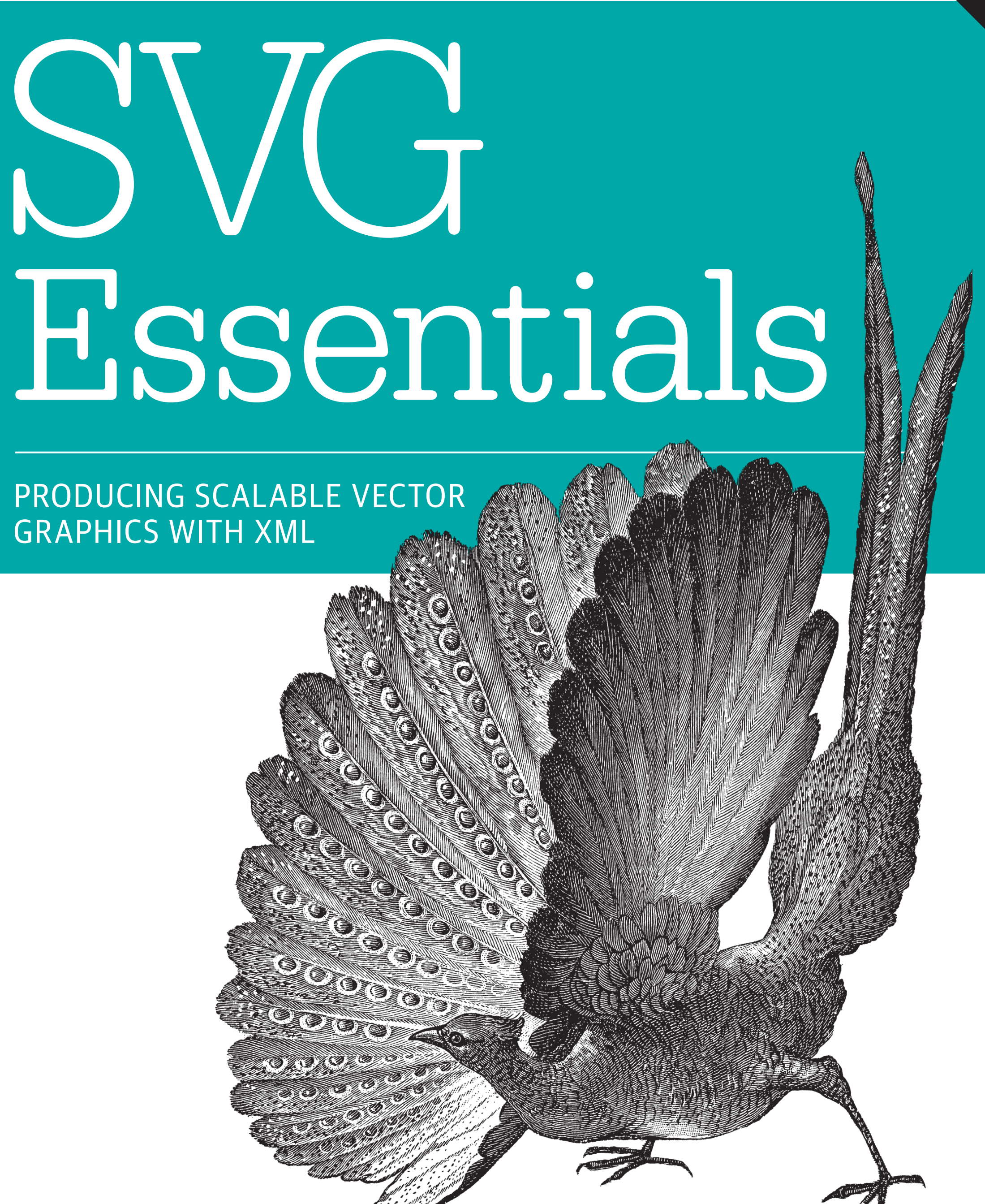
O'REILLY®

2nd Edition

SVG Essentials

PRODUCING SCALABLE VECTOR GRAPHICS WITH XML



J. David Eisenberg &
Amelia Bellamy-Royds

O'REILLY®

SVG Essentials

Learn the essentials of Scalable Vector Graphics, the markup language used by most vector drawing programs and interactive web graphics tools. *SVG Essentials* takes you through SVG's capabilities, beginning with simple line drawings and moving through complicated features such as filters, transformations, gradients, and patterns.

This thoroughly updated edition includes expanded coverage of animation, interactive graphics, and scripting SVG. Interactive examples online make it easy for you to experiment with SVG features in your web browser. Geared toward experienced designers, this book also includes appendices that explain basic concepts such as XML markup and CSS styling, so even if you have no web design experience, you can start learning SVG.

- Create and style graphics to match your web design in a way that looks great when printed or displayed on high-resolution screens
- Make your charts and decorative headings accessible to search engines and assistive technologies
- Add artistic effects to your graphics, text, and photographs using SVG masks, filters, and transformations
- Animate graphics with SVG markup, or add interactivity with CSS and JavaScript
- Create SVG from existing vector data or XML data, using programming languages and XSLT

"The first edition of *SVG Essentials* was how I first learned SVG, back in 2002. That worked out pretty well for me. It's great to see an updated edition for modern browsers and today's developers and designers."

—**Doug Schepers**
W3C, SVG Working Group

J. David Eisenberg is a programmer and instructor in San Jose, California. He's developed courses for CSS, JavaScript, CGI, and XML, and teaches Computer Information Technology courses at Evergreen Valley College. David has written *Études for Erlang* (O'Reilly) and *Let's Read Hiragana* (Eisenberg Consulting), as well as *SVG Essentials*, First Edition.

Amelia Bellamy-Royds is a freelance writer specializing in scientific and technical communication. She helps promote web standards and design through participation in online communities such as Web Platform Docs, Stack Exchange, and Codepen.

XML/WEB DESIGN

US $39.99 CAN $41.99
ISBN: 978-1-449-37435-8

Twitter: @oreillymedia
facebook.com/oreilly

SECOND EDITION

SVG Essentials

J. David Eisenberg and Amelia Bellamy-Royds

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

SVG Essentials, Second Edition

by J. David Eisenberg and Amelia Bellamy-Royds

Copyright © 2015 J. David Eisenberg and Amelia Bellamy-Royds. All rights reserved.

Printed in the United States of America.

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

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

Editors: Simon St. Laurent and Meghan Blanchette
Production Editor: Matthew Hacker
Copyeditor: Jasmine Kwityn
Proofreader: Sharon Wilkey
Indexer: Ellen Troutman Zaig
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

February 2002: First Edition

October 2014: Second Edition

Revision History for the Second Edition:

2014-10-02: First release

2015-05-01: Second release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449374358* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *SVG Essentials*, the image of a great argus pheasant, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-449-37435-8

[LSI]

To my late mother and father, for their advice and love through the years.—JDE

For Bill, who would have been so proud.—ABR

Table of Contents

Preface. xi

1. Getting Started. 1
Graphics Systems 1
Raster Graphics 1
Vector Graphics 2
Uses of Raster Graphics 2
Uses of Vector Graphics 3
Scalability 3
SVG's Role 5
Creating an SVG Graphic 6
Document Structure 6
Basic Shapes 6
Specifying Styles as Attributes 7
Grouping Graphic Objects 8
Transforming the Coordinate System 9
Other Basic Shapes 10
Paths 11
Text 12

2. Using SVG in Web Pages. 15
SVG as an Image 15
Including SVG in an <img> Element 16
Including SVG in CSS 17
SVG as an Application 18
SVG Markup in a Mixed Document 20
Foreign Objects in SVG 20
Inline SVG in XHTML or HTML5 22
SVG in Other XML Applications 25

3. **Coordinates** 27
The Viewport 27
Using Default User Coordinates 28
Specifying User Coordinates for a Viewport 30
Preserving Aspect Ratio 32
  Specifying Alignment for preserveAspectRatio 33
  Using the meet Specifier 34
  Using the slice Specifier 35
  Using the none Specifier 36
Nested Systems of Coordinates 36

4. **Basic Shapes** 39
Lines 39
Stroke Characteristics 40
  stroke-width 40
  Stroke Color 41
  stroke-opacity 43
  stroke-dasharray Attribute 44
Rectangles 45
  Rounded Rectangles 46
Circles and Ellipses 47
The <polygon> Element 48
  Filling Polygons That Have Intersecting Lines 49
The <polyline> Element 51
Line Caps and Joins 52
Basic Shapes Reference Summary 53
  Shape Elements 53
  Specifying Colors 54
  Stroke and Fill Characteristics 54

5. **Document Structure** 57
Structure and Presentation 57
Using Styles with SVG 58
  Inline Styles 58
  Internal Stylesheets 58
  External Stylesheets 59
  Presentation Attributes 60
Grouping and Referencing Objects 61
  The <g> Element 61
  The <use> Element 63
  The <defs> Element 63
  The <symbol> Element 66

The <image> Element 67

6. **Transforming the Coordinate System** 69
The translate Transformation 69
The scale Transformation 71
Sequences of Transformations 74
Technique: Converting from Cartesian Coordinates 76
The rotate Transformation 78
Technique: Scaling Around a Center Point 81
The skewX and skewY Transformations 81
Transformation Reference Summary 83
CSS Transformations and SVG 83

7. **Paths** 85
moveto, lineto, and closepath 85
Relative moveto and lineto 88
Path Shortcuts 88
The Horizontal lineto and Vertical lineto Commands 88
Notational Shortcuts for a Path 89
Elliptical Arc 90
Converting from Other Arc Formats 93
Bézier Curves 93
Quadratic Bézier Curves 94
Cubic Bézier Curves 96
Path Reference Summary 99
Paths and Filling 100
The <marker> element 100
Marker Miscellanea 104

8. **Patterns and Gradients** 107
Patterns 107
patternUnits 108
patternContentUnits 110
Nested Patterns 112
Gradients 113
The linearGradient Element 113
The radialGradient Element 118
Gradient Reference Summary 121
Transforming Patterns and Gradients 122

9. **Text** 125
Text Terminology 125

Simple Attributes and Properties of the <text> Element 126
Text Alignment 129
The <tspan> Element 129
Setting textLength 132
Vertical Text 133
Internationalization and Text 134
Unicode and Bidirectionality 134
The <switch> Element 135
Using a Custom Font 137
Text on a Path 138
Whitespace and Text 141
Case Study: Adding Text to a Graphic 142

10. Clipping and Masking. 145
Clipping to a Path 145
Masking 149
Case Study: Masking a Graphic 152

11. Filters. 155
How Filters Work 155
Creating a Drop Shadow 156
Establishing the Filter's Bounds 156
Using <feGaussianBlur> for a Drop Shadow 157
Storing, Chaining, and Merging Filter Results 158
Creating a Glowing Shadow 159
The <feColorMatrix> Element 160
More About the <feColorMatrix> Element 161
The <feImage> Filter 163
The <feComponentTransfer> Filter 164
The <feComposite> Filter 169
The <feBlend> Filter 172
The <feFlood> and <feTile> Filters 173
Lighting Effects 174
Diffuse Lighting 175
Specular Lighting 177
Accessing the Background 179
The <feMorphology> Element 181
The <feConvolveMatrix> Element 182
The <feDisplacementMap> Element 184
The <feTurbulence> Element 186
Filter Reference Summary 187

12. Animating SVG. 191
Animation Basics 192
How Time Is Measured 194
Synchronizing Animation 194
Repeated Action 196
Animating Complex Attributes 197
Specifying Multiple Values 198
Timing of Multistage Animations 199
The <set> Element 200
The <animateTransform> Element 200
The <animateMotion> Element 202
Specifying Key Points and Times for Motion 204
Animating SVG with CSS 205
  Animation Properties 206
  Setting Animation Key Frames 207
  Animating Movement with CSS 207

13. Adding Interactivity. 209
Using Links in SVG 209
Controlling CSS Animations 211
User-Triggered SMIL Animations 212
Scripting SVG 213
  Events: An Overview 216
  Listening for and Responding to Events 217
  Changing Attributes of Multiple Objects 218
  Dragging Objects 221
  Interacting with an HTML Page 225
  Creating New Elements 229

14. Using the SVG DOM. 233
Determining the Value of Element Attributes 233
SVG Interface Methods 241
Constructing SVG with ECMAScript/JavaScript 248
Animation via Scripting 251
Using JavaScript Libraries 256
Event Handling in Snap 261
  Clicking Objects 262
  Dragging Objects 262

15. Generating SVG. 265
Converting Custom Data to SVG 266
Using XSLT to Convert XML Data to SVG 270

Defining the Task 270
How XSLT Works 272
Developing an XSL Stylesheet 273

A. The XML You Need for SVG. 285

B. Introduction to Stylesheets. 299

C. Programming Concepts. 309

D. Matrix Algebra. 319

E. Creating Fonts. 327

F. Converting Arcs to Different Formats. 331

Index. 335

Preface

SVG Essentials introduces you to the Scalable Vector Graphics XML file format. SVG, a recommendation from the World Wide Web Consortium, uses XML to describe graphics that are made up of lines, curves, and text. This rather dry definition does not do justice to the scope and power of SVG.

You can add SVG graphics to an Extensible Stylesheet Language Formatting Objects (XSL-FO) document, and convert the combined document to Adobe PDF format for high-quality printouts. Mapmakers and meteorologists are using SVG to create highly detailed graphic images in a truly portable format. Web developers are embedding SVG in web pages to create high-resolution, responsive graphics with small file sizes. All of the diagrams in this book were originally created in SVG. As you learn and use SVG, you're sure to think of new and interesting uses for this technology.

Who Should Read This Book?

You should read this book if you want to

- Create SVG files in a text or XML editor
- Create SVG files from existing vector data
- Transform other XML data to SVG
- Use JavaScript to manipulate the SVG document object tree

Who Should Not Read This Book?

If you simply want to view SVG files, you need only acquire a viewer program or plug-in for the Web, download the files, and enjoy them. There's no need for you to know what's going on behind the scenes unless you wish to satisfy your lively intellectual curiosity.

If you wish to create SVG files with a drawing program that has SVG export capability, just read that program's documentation to learn how to use that program feature.

If You're Still Reading This…

If you've decided that you should indeed read this book, you should also be aware that most of the people who use this book will be fairly advanced users, quite probably from a technical background rather than a graphics design background. We didn't want to burden them with a lot of basic material up front, but we did want the book to be accessible to people with no background in XML or programming, so we created a number of introductory chapters—and then put them in the back of the book as appendixes. If you haven't used XML or stylesheets (and this could include some of the technical folks!) or have never programmed, you might want to turn first to the appendixes. A complete list of all the chapters and appendixes with details on what they contain is given later in this preface.

If you're one of the technical types, you definitely need to be aware that this book will *not* make you a better artist, any more than a book on word processing algorithms will make you a better writer. This book gives the technical details of scalable vector graphics; to create better art, you need to learn to *see*, and the book you should read in addition to this one is *The New Drawing on the Right Side of the Brain* by Dr. Betty Edwards (Tarcher).

This book gives you the essentials of SVG; if you want to find out all the details, you should go straight to the source, the W3C SVG specifications.

About the Examples

The examples in this book, except for those that involve HTML pages, have been tested with the Batik SVG viewer on a system running GNU/Linux. The Batik SVG viewer is an application of the software developed by the Apache Software Foundation's Batik project. This cross-platform software, written in Java, is available as open source under the Apache Software License and can be freely downloaded from the project website.

All the examples (including those in Chapters 2, 13, and 14 that involve JavaScript and HTML) were tested by being loaded into the Firefox and Chrome web browsers. The level of support for the more sophisticated features of SVG differs depending upon the browser.

As you look through the illustrations in this book, you will find that they are utterly lacking in artistic merit. There are reasons for this. First, each example is intended to illustrate a particular aspect of SVG, and it should do so without additional visual distractions. Second, one of the authors (David) becomes terribly depressed when he looks at other books with impossibly beautiful examples; "I can never draw anything

that looks like this," he thinks. In an effort to save you from similar distress, the examples are purposely as simple (or simplistic) as possible. As you look at them, your immediate reaction will be: "I can certainly use SVG to draw something that looks far better than this!" You can, and you will.

Organization of This Book

Chapter 1, Getting Started
: This chapter gives a brief history of SVG, compares raster and vector graphics systems, and ends with a brief tutorial introducing the main concepts of SVG.

Chapter 2, Using SVG in Web Pages
: This chapter shows you the various methods that you can use to put SVG into your HTML5 documents.

Chapter 3, Coordinates
: How do you determine the position of a point in a drawing? Which way is "up"? This chapter answers those questions, showing how to change the system by which coordinates are measured in a graphic.

Chapter 4, Basic Shapes
: This chapter shows you how to construct drawings using the basic shapes available in SVG: lines, rectangles, polygons, circles, and ellipses. It also discusses how to determine the colors for the outline and interior of a shape.

Chapter 5, Document Structure
: In a complex drawing, there are elements that are reused or repeated. This chapter tells you how to group objects together so they may be treated as a single entity and reused. It also discusses use of external images, both vector and raster.

Chapter 6, Transforming the Coordinate System
: If you draw a square on a sheet of stretchable material, and stretch the material horizontally, you get a rectangle. Skew the sides of the sheet, and you see a parallelogram. Now tilt the sheet 45 degrees, and you have a diamond. In this chapter, you will learn how to move, rotate, scale, and skew the coordinate system to affect the shapes drawn on it.

Chapter 7, Paths
: All the basic shapes are actually specific instances of the general concept of a path. This chapter shows you how to describe a general outline for a shape by using lines, arcs, and complex curves.

Chapter 8, Patterns and Gradients
: This chapter adds more to the discussion of color from Chapter 4, discussing how to create a color gradient or a fill pattern.

Chapter 9, Text
: Graphics aren't just lines and shapes; text is an integral part of a poster or a schematic diagram. This chapter shows how to add text to a drawing, both in a straight line and following a path.

Chapter 10, Clipping and Masking
: This chapter shows you how to use a clipping path to display a graphic as though it were viewed through a circular lens, keyhole, or any other arbitrary shape. It also shows how to use a mask to alter an object's transparency so that it appears to "fade out" at the edges.

Chapter 11, Filters
: Although an SVG file describes vector graphics, the document is eventually rendered on a raster device. In this chapter, you'll learn how to apply raster-oriented filters to a graphic to blur an image, transform its colors, or produce lighting effects.

Chapter 12, Animating SVG
: This chapter shows you how to use SVG's built-in animation capabilities.

Chapter 13, Adding Interactivity
: In addition to SVG's built-in animation, you can use both CSS and JavaScript to dynamically control a graphic's attributes.

Chapter 14, Using the SVG DOM
: This chapter goes further in depth with using JavaScript to manipulate the Document Object Model. It also gives a brief introduction to a JavaScript library designed for working with SVG.

Chapter 15, Generating SVG
: Although you can create an SVG file from scratch, many people will have existing vector data or XML data that they wish to display in graphic form. This chapter discusses the use of programming languages and XSLT to create SVG from these data sources.

Appendix A, The XML You Need for SVG
: SVG is an application of XML, the Extensible Markup Language. If you haven't used XML before, you should read this appendix to familiarize yourself with this remarkably powerful and flexible format for structuring data and documents.

Appendix B, Introduction to Stylesheets
: You can use stylesheets to apply visual properties to particular elements in your SVG document. These are exactly the same kind of stylesheets that can be used with HTML documents. If you've never used stylesheets before, you'll want to read this brief introduction to the anatomy of a stylesheet.

Appendix C, Programming Concepts
: If you're a graphic designer who hasn't done much programming, you'll want to find out what programmers are talking about when they throw around words like *object model* and *function*.

Appendix D, Matrix Algebra
: To fully understand coordinate transformations and filter effects in SVG, it's helpful, though not necessary, to understand matrix algebra, the mathematics used to compute the coordinates and pixels. This appendix highlights the basics of matrix algebra.

Appendix E, Creating Fonts
: TrueType fonts represent glyphs (characters) in a vector form. This appendix shows you how to take your favorite fonts and convert them to paths for use in SVG documents.

Appendix F, Converting Arcs to Different Formats
: Many applications represent arcs in a center-and-angles format. This appendix provides code to convert from that format to SVG's format for arcs and back again.

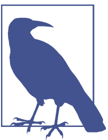
Conventions Used in This Book

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

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

`Constant width bold`
: Used to highlight a section of code being discussed in the text.

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



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

This element indicates a warning or caution.

This book uses *callouts* to denote *points of interest* in code listings. A callout is shown as a number in a filled circle; the corresponding number after the listing gives an explanation. Here's an example:

```
Roses are red,
   Violets are blue. ❶
Some poems rhyme;
   This one doesn't. ❷
```

❶ Violets actually have a color value of `#9933cc`.

❷ This poem uses the literary device known as a *surprise ending*.

Many of the examples are available to test out online; the URL is indicated in the text. Some of the online examples have markup that you can edit; click the Refresh button to see the results of your changes. You may also click the Reset button to return the example to its original state.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

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

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

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://shop.oreilly.com/product/0636920032335.do*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

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

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

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

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

Acknowledgments for the First Edition

I'd like to thank Simon St. Laurent, the editor of this book, for his guidance and comments, which were always right on the mark. He also told me in an email, "we already know that you know how to write," which is one of the nicest things anyone has ever told me.

Thanks also to Edd Dumbill, who wrote the document that I modified only slightly to create Appendix A. Of course, any errors in that appendix have been added by my modifications.

Thanks also go to the technical reviewers of this book: Antoine Quint and David Klaphaak and the SVG Quality Engineering team at Adobe, who did the technical review of the manuscript. Your comments have helped improve many aspects of this book.

Jeffrey Zeldman is the person who first put the idea in my head that I, too, could write a book, and for that I thank him most sincerely.

I also want to thank all the people, foremost among them my brother, Steven, who, when I told them I was writing a book, believed in me enough to say, "Wow, that's great."

Acknowledgments for the Second Edition

We would like to thank Shelly Powers for her excellent technical review. Our thanks also go to Simon St. Laurent and Meghan Blanchette for their fantastic job of editing and to Matthew Hacker and the O'Reilly tools and production teams for getting all the finishing touches just right, despite the best efforts of stubborn software and fussy authors.

From David: I'd like to give special thanks to Amelia Bellamy-Royds. She was initially doing technical review of the book, and her comments were so lucid and well written that I found myself lifting them verbatim and realized that she should be a coauthor. Her corrections and additions have made the book far better than I could have imagined.

From Amelia: I'd like to thank David for being decent enough to recognize when I'd exceeded my original job description and deserved extra credit. His original book was a wonderfully welcoming introduction to SVG. As someone who had puzzled through all the quirks of web browser implementations on my own, I really wanted the revised book to have clear explanations for all the things that confused me when learning SVG as it currently works in practice.

I also need to send special thanks to my husband, Chris, who has been hugely supportive, but who has also regularly reminded me when I need to step away from the computer, eat, sleep, or get some fresh air.

CHAPTER 1
Getting Started

SVG, which stands for *S*calable *V*ector *G*raphics, is an application of XML that makes it possible to represent graphic information in a compact, portable form. Interest in SVG is growing rapidly. Most modern web browsers can display SVG graphics, and most vector drawing software programs can export SVG graphics. This chapter begins with a description of the two major systems of computer graphics, and describes where SVG fits into the graphics world. The chapter concludes with a brief example that uses many of the concepts that you will explore in detail in the following chapters.

Graphics Systems

The two major systems for representing graphic information on computers are raster and vector graphics.

Raster Graphics

In *raster graphics*, an image is represented as a rectangular array of picture elements or pixels (see Figure 1-1). Each pixel is represented either by its RGB color values or as an index into a list of colors. This series of pixels, also called a *bitmap*, is often stored in a compressed format. Because most modern display devices are also raster devices, displaying an image requires a viewer program to do little more than uncompress the bitmap and transfer it to the screen.

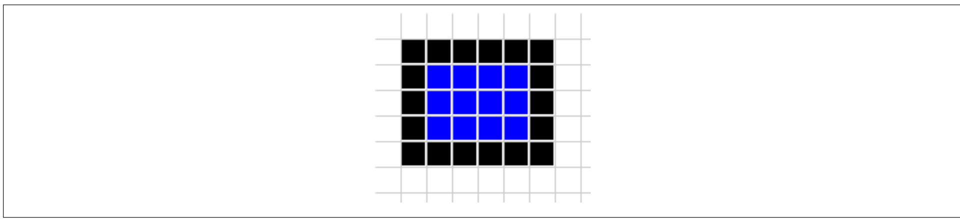


Figure 1-1. Raster graphic rectangle

Vector Graphics

In a *vector graphics* system, an image is described as a series of geometric shapes (see Figure 1-2). Rather than receiving a finished set of pixels, a vector viewing program receives commands to draw shapes at specified sets of coordinates.

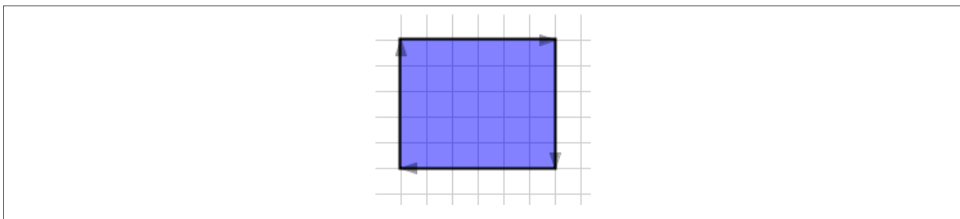


Figure 1-2. Vector graphic rectangle

If you think of producing an image on graph paper, raster graphics work by describing which squares should be filled in with which colors. Vector graphics work by describing the grid points at which lines or curves are to be drawn. Some people describe vector graphics as a set of instructions for a drawing, while bitmap graphics (rasters) are points of color in specific places. Vector graphics "understand" what they are—a square "knows" it's a square, and text "knows" that it's text. Because they are objects rather than a series of pixels, vector objects can change their shape and color, whereas bitmap graphics cannot. Also, all text is searchable because it really is text, no matter how it looks or how it is rotated or transformed.

Another way to think of raster graphics is as paint on canvas, while vector graphics are lines and shapes made of a stretchable material that can be moved around on a background.

Uses of Raster Graphics

Raster graphics are most appropriate for use with photographs, which are rarely composed of distinct lines and curves. Scanned images are often stored as bitmaps; even though the original may be *line art*, you want to store the image as a whole and don't

care about its individual components. A fax machine, for example, doesn't care what you've drawn; it simply transmits pixels from one place to another in raster form.

Tools for creating images in raster format are widespread and generally easier to use than many vector-based tools. There are many different ways to compress and store a raster image, and the internal representation of these formats is public. Program libraries to read and write images in compressed formats such as JPEG, GIF, and PNG are widely available. These are some of the reasons that web browsers have, until the arrival of SVG, supported only raster images.

Uses of Vector Graphics

Vector graphics are used in the following:

- Computer Assisted Drafting (CAD) programs, where accurate measurement and the ability to zoom in on a drawing to see details are essential.
- Programs for designing graphics that will be printed on high-resolution printers (e.g., Adobe Illustrator).
- The Adobe PostScript printing and imaging language; every character that you print is described in terms of lines and curves.
- The vector-based Macromedia Flash system for designing animations, presentations, and websites.

Because most of these files are encoded in binary format or as tightly packed bitstreams, it is difficult for a browser or other user agent to parse out embedded text, or for a server to dynamically create vector graphic files from external data. Most of the internal representations of vector graphics are proprietary, and code to view or create them is not generally available.

Scalability

Although they are not as popular as raster graphics, vector graphics have one feature that makes them invaluable in many applications—they can be scaled without loss of image quality. As an example, here are two drawings of a cat. Figure 1-3 was made with raster graphics; Figure 1-4 is a vector image. Both are shown as they appear on a screen that displays 72 pixels per inch.

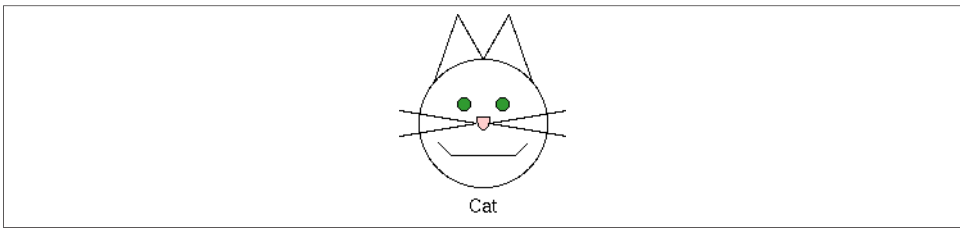


Figure 1-3. Raster image of cat

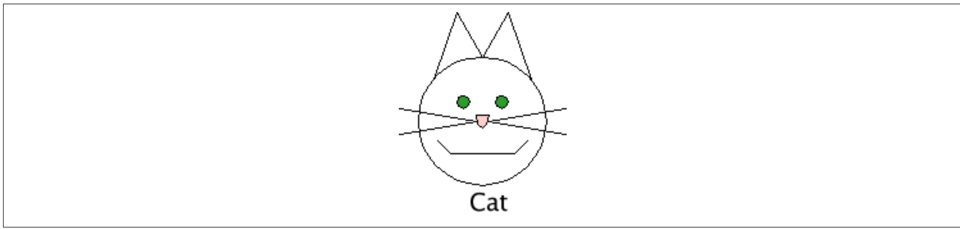


Figure 1-4. Vector image of cat

When a display program zooms in on the raster graphic, it must find some way to expand each pixel. The simplest approach to zooming in by a factor of four is to make each pixel four times as large. The results, shown in Figure 1-5, are not particularly pleasing.

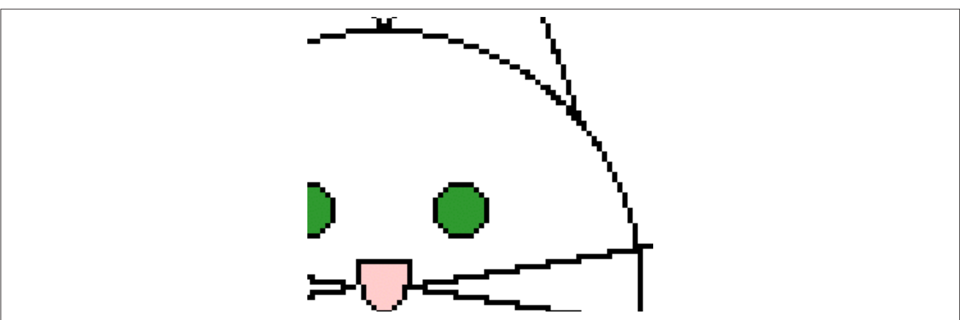


Figure 1-5. Expanded raster image

Although it is possible to use techniques such as edge detection and anti-aliasing to make the expanded image more pleasing, these techniques are time-consuming. Furthermore, since all the pixels in a raster graphic are equally anonymous, there's no guarantee that an algorithm can correctly detect edges of shapes. Anti-aliasing results in something like Figure 1-6.

Figure 1-6. Expanded anti-aliased raster image

Expanding a vector image by a factor of four, on the other hand, merely requires the display program to multiply all the coordinates of the shapes by four and redraw them at the full resolution of the display device. Thus, Figure 1-7, which is also a screenshot from a 72 dots per inch (DPI) screen, shows crisp, clear edges on the lines with significantly less of the stair-step effects of the expanded raster image.

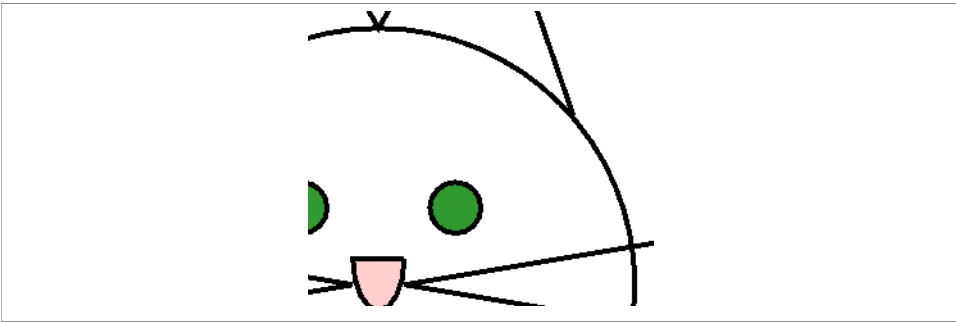


Figure 1-7. Expanded vector image

SVG's Role

In 1998, the World Wide Web Consortium formed a working group to develop a representation of vector graphics as an XML application. Because SVG is an XML application, the information about an image is stored as plain text, and it brings the advantages of XML's openness, transportability, and interoperability.

CAD and graphic design programs often store drawings in a proprietary binary format. By adding the ability to import and export drawings in SVG format, applications gain a common standard format for interchanging information.

Because it is an XML application, SVG cooperates with other XML applications. A mathematics textbook, for example, could use XSL Formatting Objects for explanatory text, MathML to describe equations, and SVG to generate the graphs for the equations.

The SVG working group's specification is an official World Wide Web Consortium Recommendation. Applications such as Adobe Illustrator and Inkscape can import and export drawings in SVG format. On the Web, SVG is natively supported in many browsers and has many of the same transformation and animation capabilities that CSS-styled HTML has. Because the SVG files are XML, text in the SVG display is available to any user agent that can parse XML.

Creating an SVG Graphic

In this section, you will see an SVG file that produces the image of the cat that you saw earlier in the chapter. This example introduces many of the concepts that you will read about in further detail in subsequent chapters. This file will be a good example of how to write an example file, which is not necessarily the way you should write an SVG file that will be part of a finished project.

Document Structure

Example 1-1 starts with the standard XML processing instruction and `DOCTYPE` declaration. The root `<svg>` element defines the `width` and `height` of the finished graphic in pixels. It also defines the SVG namespace via the `xmlns` attribute. The `<title>` element's content is available to a viewing program for use in a title bar or as a tooltip pointer, and the `<desc>` element lets you give a full description of the image.

Example 1-1. Basic structure of an SVG document

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

<svg width="140" height="170"
  xmlns="http://www.w3.org/2000/svg">
<title>Cat</title>
<desc>Stick Figure of a Cat</desc>
<!-- the drawing will go here -->
</svg>
```

Basic Shapes

You draw the cat's face by adding a `<circle>` element. The element's attributes specify the center *x*-coordinate, center *y*-coordinate, and radius. The (0,0) point is the upper-left corner of the picture. *x*-coordinates increase as you move horizontally to the right; *y*-coordinates increase as you move vertically downward.

The circle's location and size are part of the drawing's *structure*. The color in which it is drawn is part of its *presentation*. As is customary with XML applications, you should separate structure and presentation for maximum flexibility. Presentation information is contained in the `style` attribute. Its value will be a series of presentation properties and values, as described in Appendix B, in "Anatomy of a Style" on page 299. Use a stroke color of `black` for the outline, and a fill color of `none` to make the face transparent. The SVG is shown in Example 1-2, and its result in Figure 1-8.

Example 1-2. Basic shapes—circle

http://oreillymedia.github.io/svg-essentials-examples/ch01/ex01-02.html

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

<svg width="140" height="170"
  xmlns="http://www.w3.org/2000/svg">
<title>Cat</title>
<desc>Stick Figure of a Cat</desc>

<circle cx="70" cy="95" r="50" style="stroke: black; fill: none"/>

</svg>
```

Figure 1-8. Stage one—drawing a circle

Specifying Styles as Attributes

Now add two more circles for the eyes in Example 1-3. Although their fill and stroke colors are really part of the presentation, SVG does allow you to specify them as individual attributes. In this example, the `fill` and `stroke` colors are written as two separate attributes rather than together inside the `style` attribute. You probably won't use this method often; it's described further in Chapter 5, in "Presentation Attributes" on page 60. We've put it here just to prove that it can be done. The results are shown in Figure 1-9.

The `<?xml …?>` and `<!DOCTYPE?>` have been omitted to save space in the listing.

Example 1-3. Basic shapes—filled circles

http://oreillymedia.github.io/svg-essentials-examples/ch01/ex01-03.html

```
<svg width="140" height="170"
  xmlns="http://www.w3.org/2000/svg">
  <title>Cat</title>
  <desc>Stick Figure of a Cat</desc>

  <circle cx="70" cy="95" r="50" style="stroke: black; fill: none"/>
  <circle cx="55" cy="80" r="5" stroke="black" fill="#339933"/>
  <circle cx="85" cy="80" r="5" stroke="black" fill="#339933"/>
</svg>
```

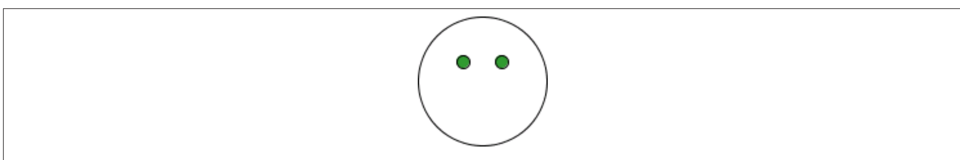


Figure 1-9. Stage two—drawing the face and eyes

Grouping Graphic Objects

Example 1-4 adds the whiskers on the right side of the cat's face with two `<line>` elements. You want to treat these whiskers as a unit (you'll see why in a moment), so enclose them in the `<g>` grouping element, and give it an `id`. You specify a line by giving the *x*- and *y*-coordinates for its starting point (`x1` and `y1`) and ending point (`x2` and `y2`). Figure 1-10 shows the result.

Example 1-4. Basic shapes—lines

http://oreillymedia.github.io/svg-essentials-examples/ch01/ex01-04.html

```
<svg width="140" height="170"
  xmlns="http://www.w3.org/2000/svg">
<title>Cat</title>
<desc>Stick Figure of a Cat</desc>

<circle cx="70" cy="95" r="50" style="stroke: black; fill: none;"/>
<circle cx="55" cy="80" r="5" stroke="black" fill="#339933"/>
<circle cx="85" cy="80" r="5" stroke="black" fill="#339933"/>
<g id="whiskers">
   <line x1="75" y1="95" x2="135" y2="85" style="stroke: black;"/>
   <line x1="75" y1="95" x2="135" y2="105" style="stroke: black;"/>
</g>
</svg>
```

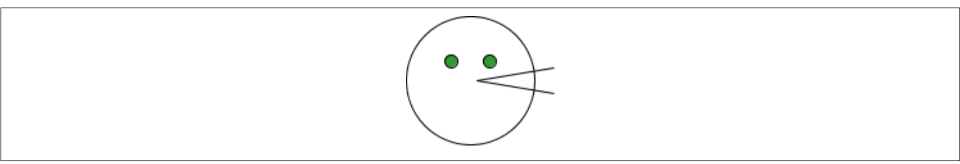


Figure 1-10. Stage three—adding whiskers on the right side

Transforming the Coordinate System

Now you will `<use>` the whiskers group and `transform` it into the left whiskers. Example 1-5 first flips the coordinate system by multiplying the *x*-coordinates by negative one in a `scale` transformation. This means that the point (75,95) is now located at the place that would have been (–75,95) in the original coordinate system. In the new scaled system, coordinates increase as you move *left*. This means you have to `translate` (move) the coordinate system 140 pixels right, the negative direction, to get them where you want them, as shown in Figure 1-11.

Example 1-5. Transforming the coordinate system

http://oreillymedia.github.io/svg-essentials-examples/ch01/ex01-05.html

```
<svg width="140" height="170"
  xmlns="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink">
  <title>Cat</title>
  <desc>Stick Figure of a Cat</desc>

  <circle cx="70" cy="95" r="50" style="stroke: black; fill: none;"/>
  <circle cx="55" cy="80" r="5" stroke="black" fill="#339933"/>
  <circle cx="85" cy="80" r="5" stroke="black" fill="#339933"/>
  <g id="whiskers">
    <line x1="75" y1="95" x2="135" y2="85" style="stroke: black;"/>
    <line x1="75" y1="95" x2="135" y2="105" style="stroke: black;"/>
  </g>
  <use xlink:href="#whiskers" transform="scale(-1 1) translate(-140 0)"/>
</svg>
```

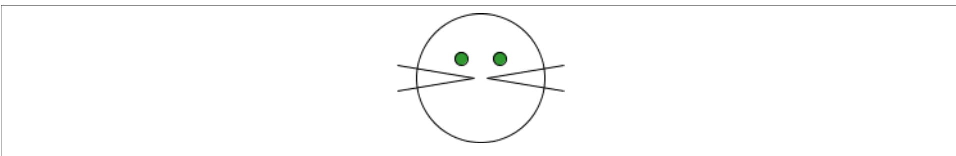


Figure 1-11. Stage four—adding whiskers on the left side

The `xlink:href` attribute in the `<use>` element is in a different *namespace* (see Appendix A for details). To make sure your SVG document will work with all SVG viewers, you must add the `xmlns:xlink` attribute to the opening `<svg>` tag.

The `transform` attribute's value lists the transformations, one after another, separated by whitespace.

Other Basic Shapes

Example 1-6 constructs the ears and mouth with the `<polyline>` element, which takes pairs of *x*- and *y*-coordinates as the `points` attribute. You separate the numbers with either blanks or commas as you please. The result is in Figure 1-12.

Example 1-6. Basic shapes—polylines

http://oreillymedia.github.io/svg-essentials-examples/ch01/ex01-06.html

```
<svg width="140" height="170"
  xmlns="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink">
  <title>Cat</title>
  <desc>Stick Figure of a Cat</desc>

  <circle cx="70" cy="95" r="50" style="stroke: black; fill: none;"/>
  <circle cx="55" cy="80" r="5" stroke="black" fill="#339933"/>
  <circle cx="85" cy="80" r="5" stroke="black" fill="#339933"/>
  <g id="whiskers">
    <line x1="75" y1="95" x2="135" y2="85" style="stroke: black;"/>
    <line x1="75" y1="95" x2="135" y2="105" style="stroke: black;"/>
  </g>
  <use xlink:href="#whiskers" transform="scale(-1 1) translate(-140 0)"/>
  <!-- ears -->
  <polyline points="108 62,  90 10,  70 45,  50, 10,  32, 62"
    style="stroke: black; fill: none;" />
  <!-- mouth -->
  <polyline points="35 110, 45 120, 95 120, 105, 110"
      style="stroke: black; fill: none;" />
</svg>
```

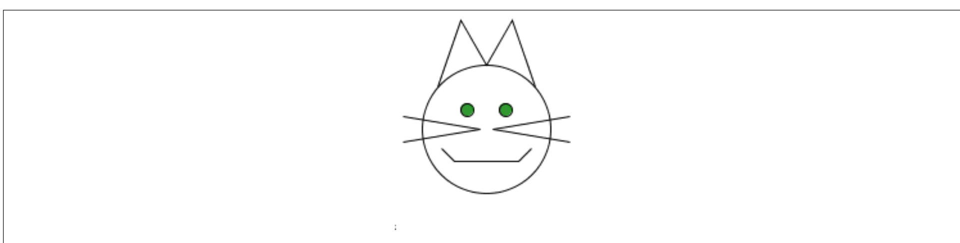


Figure 1-12. Stage five—adding ears and mouth

Paths

All of the basic shapes are actually shortcuts for the more general `<path>` element, which Example 1-7 uses to add the cat's nose. The result is in Figure 1-13. This element has been designed to make specifying a path, or sequence of lines and curves, as compact as possible. The path in Example 1-7 translates, in words, to "Move to coordinate (75,90). Draw a line to coordinate (65,90). Draw an elliptical arc with an *x*-radius of 5 and a *y*-radius of 10, ending back at coordinate (75,90)."

Example 1-7. Using the <path> element

http://oreillymedia.github.io/svg-essentials-examples/ch01/ex01-07.html

```
<svg width="140" height="170"
  xmlns="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink">
  <title>Cat</title>
  <desc>Stick Figure of a Cat</desc>

  <circle cx="70" cy="95" r="50" style="stroke: black; fill: none;"/>
  <circle cx="55" cy="80" r="5" stroke="black" fill="#339933"/>
  <circle cx="85" cy="80" r="5" stroke="black" fill="#339933"/>
  <g id="whiskers">
    <line x1="75" y1="95" x2="135" y2="85" style="stroke: black;"/>
    <line x1="75" y1="95" x2="135" y2="105" style="stroke: black;"/>
  </g>
  <use xlink:href="#whiskers" transform="scale(-1 1) translate(-140 0)"/>
  <!-- ears -->
  <polyline points="108 62,  90 10,  70 45,  50, 10,  32, 62"
    style="stroke: black; fill: none;" />
  <!-- mouth -->
  <polyline points="35 110, 45 120, 95 120, 105, 110"
      style="stroke: black; fill: none;" />
  <!-- nose -->
  <path d="M 75 90 L 65 90 A 5 10 0  0 0 75 90"
    style="stroke: black; fill: #ffcccc"/>
</svg>
```

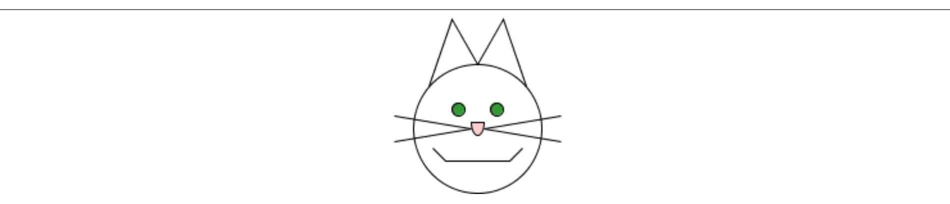


Figure 1-13. Stage six—adding a nose

Text

Finally, because this picture is so crudely drawn, there's a good chance that people will not know it is a cat. Hence, Example 1-8 adds text to the picture as a label. In the `<text>` element, the `x` and `y` attributes that specify the text's location are part of the structure. The font family and font size are part of the presentation, and thus part of the `style` attribute. Unlike the other elements you've seen, `<text>` is a container element, and its content is the text you want to display. Figure 1-14 shows the final result.

Example 1-8. Adding a label

http://oreillymedia.github.io/svg-essentials-examples/ch01/ex01-08.html

```
<svg width="140" height="170"
  xmlns="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink">
  <title>Cat</title>
  <desc>Stick Figure of a Cat</desc>

  <circle cx="70" cy="95" r="50" style="stroke: black; fill: none;"/>
  <circle cx="55" cy="80" r="5" stroke="black" fill="#339933"/>
  <circle cx="85" cy="80" r="5" stroke="black" fill="#339933"/>
  <g id="whiskers">
    <line x1="75" y1="95" x2="135" y2="85" style="stroke: black;"/>
    <line x1="75" y1="95" x2="135" y2="105" style="stroke: black;"/>
  </g>
  <use xlink:href="#whiskers" transform="scale(-1 1) translate(-140 0)"/>
  <!-- ears -->
  <polyline points="108 62,  90 10,  70 45,  50, 10,  32, 62"
    style="stroke: black; fill: none;" />
  <!-- mouth -->
  <polyline points="35 110, 45 120, 95 120, 105, 110"
      style="stroke: black; fill: none;" />
  <!-- nose -->
  <path d="M 75 90 L 65 90 A 5 10 0  0 0 75 90"
    style="stroke: black; fill: #ffcccc"/>
  <text x="60" y="165" style="font-family: sans-serif; font-size: 14pt;
    stroke: none; fill: black;">Cat</text>
</svg>
```

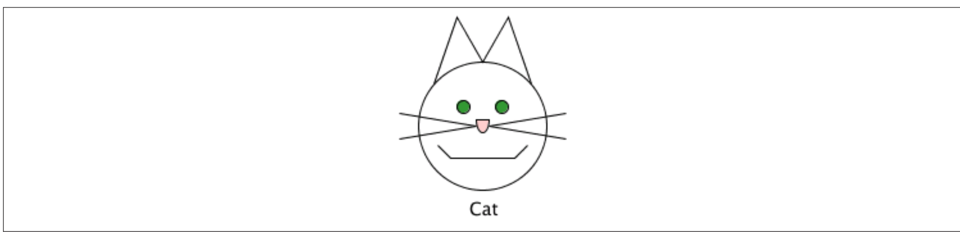


Figure 1-14. Stage seven—finished image with label

That concludes our brief overview of SVG; in the following chapters, you'll examine these concepts in depth.

CHAPTER 2
Using SVG in Web Pages

John Donne said that no man is an island, and likewise SVG does not exist in isolation. Of course, you can view SVG images on their own, as an independent file in your web browser or SVG viewer. Many of the examples in this book work that way. But in other cases, you will want your graphic to be integrated in a larger document, which contains paragraphs of text, forms, or other content that cannot easily be displayed using SVG alone. This chapter describes various ways of integrating SVG within HTML and other document types.

Figure 2-1 shows the cat drawing from the previous chapter, inserted into an HTML page in four different ways. The results look almost identical, but each method has benefits and limitations.

SVG as an Image

SVG is an image format, and as such it can be included in HTML pages in the same ways as other image types. There are two approaches: you can include the image within the HTML markup in an `<img>` element (recommended when the image is a fundamental part of the page's content); or you can insert the image as a CSS style property of another element (recommended when the image is primarily decorative).

Regardless of which method you use, including SVG as an *image* imposes certain limitations. The image will be rendered ("drawn" in the sense that the SVG code is converted to a raster image for display) separately from the main web page, and there is no way to communicate between the two. Styles defined on the main web page will have no effect on the SVG. You may need to define a default font size within your SVG code if your graphic includes text or defines lengths relative to the font size. Furthermore, scripts running on the main web page will not be able to discover or modify any of the SVG's document structure.

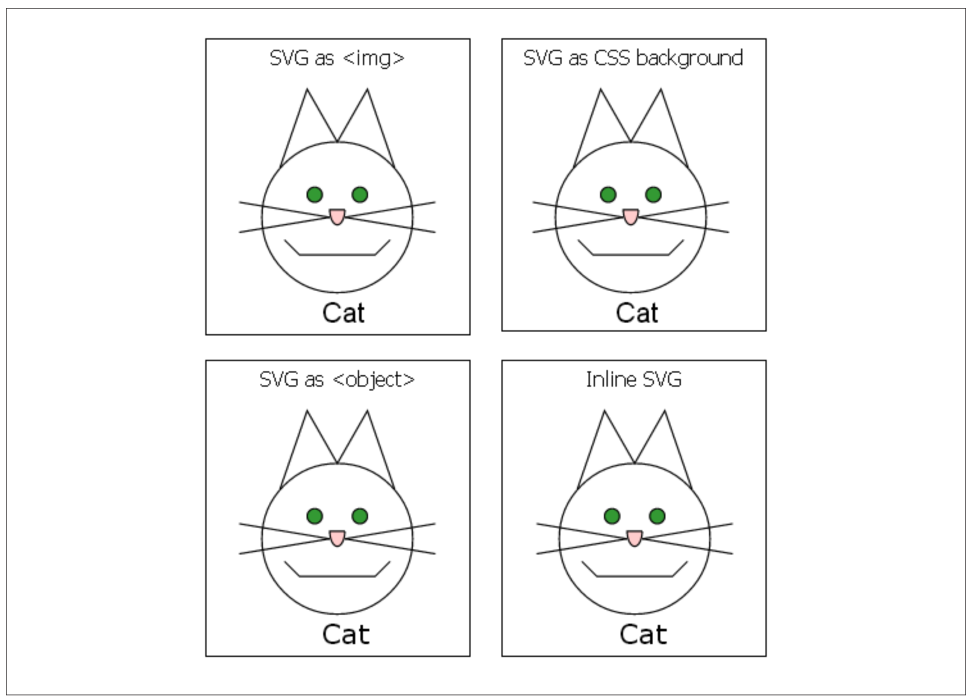


Figure 2-1. Screenshot of a web page with SVG inserted four ways

Most web browsers will not load files referenced from an SVG used as an image; this includes other image files, external scripts, and even webfont files. Depending on the browser and the user's security settings, scripts defined within the SVG file may not run, and URL fragments (the part of the URL after `#`, which indicates which part of the file you're interested in) may be ignored. Animation, as defined in Chapter 12, *is* supported within images (in browsers that support it in SVG in general).

Including SVG in an <img> Element

The HTML `<img>` element defines a space into which the browser should draw an external image file. The image file to use is specified with the `src` (source) attribute. Including an SVG image within an `<img>` element is as simple as setting the source to point to the location of your SVG file on the web server. Of course, you should also give a description with an `alt` and/or a `title` attribute so that users who cannot see the image can still understand what it represents. For example:

```
<img src="cat.svg" title="Cat Image"
     alt="Stick Figure of a Cat" />
```

Although most web browsers now support SVG as images, some older browsers will not know how to render the file and will display a broken-file icon (or nothing at all). For other browsers, you may need to confirm that your web server is configured to declare the correct media type header (`image/svg+xml`) for files ending in *.svg*.

The height and width of the image can be set using attributes or CSS properties (which take precedence). Other CSS properties control the placement of the image within the web page. If you do not specify dimensions for the `<img>` element, the intrinsic dimensions of the image file are used. If you specify only *one* of height or width, the other dimension is scaled proportionally so that the aspect ratio (the ratio of width to height) matches the intrinsic dimensions.

For raster images, the intrinsic dimension is the image size in pixels. For SVG, it's more complicated. If the root `<svg>` element in the file has explicit height and width attributes, those are used as the intrinsic dimensions of the file. If one of height *or* width is specified, but not both, and the `<svg>` has a `viewBox` attribute, then the `viewBox` will be used to calculate the aspect ratio and the image will be scaled to match the specified dimension. Otherwise, if the `<svg>` has a `viewBox` attribute but no dimensions, then the height and width parts of the `viewBox` are treated as lengths in pixels. If that all sounds incomprehensible, rest assured: we'll introduce the `viewBox` attribute properly in "Specifying User Coordinates for a Viewport" on page 30, in Chapter 3.

If neither the `<img>` element nor the root `<svg>` element has any information about the size of the image, the browser *should* apply the default HTML size for embedded content, 150 pixels tall and 300 pixels wide, but it is best not to rely on this.

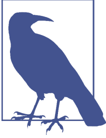
Including SVG in CSS

Various CSS style properties accept a URL to an image file as a value. The most commonly used is the `background-image` property, which draws the image (or multiple layered images) behind the text content of the element being styled.

By default, a background image is drawn at its intrinsic dimensions and repeated in both the horizontal and vertical direction to fill up the dimensions of the element. The intrinsic dimensions of an SVG file are determined in the same manner as described in "Including SVG in an <img> Element" on page 16. If there are no intrinsic dimensions, the SVG will be scaled to 100% of the height and width of the element. The size can be set explicitly using the `background-size` property, and repeat patterns and image position can be set using `background-repeat` and `background-position`:

```
div.background-cat {
 background-image: url("cat.svg");
```

```
 background-size: 100% 100%;
}
```



When using raster images for multiple small icons and logos, it is common to arrange all the images in a grid within a single image file, and then use `background-size` and `background-position` to display the correct image for each element. That way, the web browser only has to download one image file, resulting in much faster display of the web page. The compound image file is called a CSS *sprite*, named after a mythical helpful elf that magically makes things easier. SVG files can be designed as sprites, and browsers are getting better at rendering them efficiently, but you should probably avoid making the sprite file too big.

The SVG specifications define other ways to create multiple icons within a single image file; you then use URL fragments to indicate which icon to display. Ideally, these would replace sprites based on the `background-position` property. However, as mentioned previously, some browsers ignore URL fragments when rendering SVG as an image, so these features are not currently of much practical use in CSS.

In addition to background images, SVG files can be used in CSS as a `list-image` (used to create decorative bulleted lists) or `border-image` (used to create fanciful borders).

SVG as an Application

To integrate an external SVG file into an HTML page without the limitations of treating the SVG as an image, you can use an embedded object.

The `<object>` element is the general-purpose way of embedding external files in HTML (version 4 and up) and XHTML documents. It can be used to embed images, similar to `<img>`, or to embed separate HTML/XML documents, similar to an `<iframe>`. More importantly, it can also be used to embed files of any arbitrary type, so long as the browser has an application (a browser plug-in or extension) to interpret that file type. Using an object to embed your SVG can make your graphic available to users of older browsers that cannot display SVG directly, so long as they have an SVG plug-in.

The `type` attribute of the `<object>` element indicates the type of file you're embedding. The attribute should be a valid Internet media type (commonly known as a MIME type). For SVG, use `type="image/svg+xml"`.

The browser uses the file type to determine how (or if) it can display the file, without having to download it first. The location of the file itself is specified by the `data` attribute. The `alt` and `title` attributes work the same as for images.

The object element must have both a start and end tag. Any content in between the two will be rendered only if the object data itself cannot be displayed. This can be used to specify a fallback image or some warning text to display if the browser doesn't have any way of displaying SVG.[1] The following code displays both a text explanation and a raster image in browsers that don't support SVG:

```
<object data="cat.svg" type="image/svg+xml"
     title="Cat Object" alt="Stick Figure of a Cat" >
  <!-- As a fallback, include text or a raster image file -->
  <p>No SVG support! Here's a substitute:</p>
  <img src="cat.png" title="Cat Fallback"
     alt="A raster rendering of a Stick Figure of a Cat" />
</object>
```

<object> versus <embed>

Prior to the introduction of the `<object>` element, some browsers used the non-standard `<embed>` element for the same purpose. It has now been adopted into the standards, so you can use `<embed>` instead of an `<object>` element if you're worried about supporting older browsers. For even wider support, use `<embed>` as the fallback content inside the `<object>` tags.

There are two important differences between `<embed>` and `<object>`: first, the source data file is specified using a `src` attribute, not `data`; second, the `<embed>` element cannot have any child content, so there is no fallback option if the embed fails.

Although not adopted into the specifications, most browsers also support the optional `pluginspage` attribute on `<embed>` elements, which gives the URL of a page where users can download a plug-in for rendering the file type if they don't have one installed.

When you include an SVG file as an embedded object (whether with `<object>` or `<embed>`), the external file is rendered in much the same way as if it was included in an `<img>` element: it is scaled to fit the width and height of the embedding element, and it does not inherit any styles declared in the parent document.

Unlike with images, however, the embedded SVG can include external files, and scripts can communicate between the object and the parent page, as described in "Interacting with an HTML Page" on page 225.

1. In addition to fallback content, an `<object>` may also contain `<param>` elements defining parameters for the plug-in. However, these aren't used for rendering SVG data.

SVG Markup in a Mixed Document

The image and application approaches to integrating SVG in a web page are both methods to display a complete, separate, SVG file. However, it is also possible to mix SVG code with HTML or XML markup in a single file.

Combining your markup into one file can speed up your web page load times, because the browser does not have to download a separate file for the graphic. However, if the same graphic is used on many pages on your website, it can increase the total download size and time by repeating the SVG markup within each page.

More importantly, all the elements within a mixed document will be treated as a single document object when applying CSS styles and working with scripts.

Foreign Objects in SVG

One way of mixing content is to insert sections of HTML (or other) content within your SVG. The SVG specifications define a way of embedding such "foreign" content within a specified region of the graphic.

The `<foreignObject>` element defines a rectangular area into which the web browser (or other SVG viewer) should draw the child XML content. The browser is responsible for determining *how* to draw that content. The child content is often XHTML (XML-compliant HTML) code, but it could be any form of XML that the SVG viewer is capable of displaying. The type of content is defined by declaring the XML namespace on the child content using the `xmlns` attribute.

The rectangular drawing area is defined by the `x`, `y`, `width`, and `height` attributes of the `<foreignObject>` element, in a manner similar to the `<use>` or `<image>` elements, which we'll get to in Chapter 5.

The rectangle is evaluated in the local SVG coordinate system, and so is subject to coordinate system transformations (which we'll talk about in Chapter 6) or other SVG effects. The child XML document is rendered normally into a rectangular frame, and then the result is manipulated like any other SVG graphic. An SVG foreign object containing an XHTML paragraph is shown in Figure 2-2.

The `<foreignObject>` element has great potential for creating mixed SVG/XHTML documents, but is currently not well supported. Internet Explorer (at least up to version 11) does not support it at all, and there are bugs and inconsistencies in the other browsers' implementations.

If you want to define fallback content in case the SVG viewer cannot display foreign content, you can use the `<switch>` element in combination with the `requiredFeatures` attribute, as shown in Example 2-1. In browsers that support XHTML and foreign objects, that code creates Figure 2-2; in other browsers, it displays SVG text.

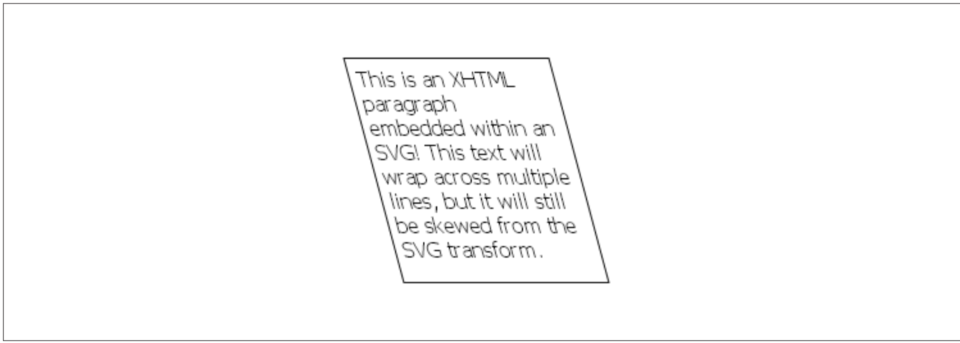
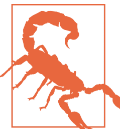


Figure 2-2. Screenshot of an SVG file containing XHTML text

The `<switch>` element instructs the SVG viewer to draw only the first direct child element (and all of *its* children) for which the `requiredFeatures`, `requiredExtensions`, and `systemLanguage` test attributes either evaluate to true or are absent. We'll discuss the use of the `systemLanguage` attribute to switch between different texts in "The <switch> Element" on page 135, in Chapter 9. When testing for required features, you use one of the URL strings given in the specifications; `<foreignObject>` support is part of the Extensibility feature.



Unfortunately, there is no consistent, cross-browser way to specify *which* type of foreign object is required. Maybe you want to use the MathML language to display a formula for your chart, with a plain-text version as a fallback for browsers that don't understand MathML. The `requiredExtensions` attribute is supposed to indicate what type of added capability is needed, but the SVG 1.1 specifications did not clearly describe how the extensions should be identified—except to say that it should be with a URL. Firefox uses the XML namespace URL, but other browsers do not.

Example 2-1. The <foreignObject> element, with a <switch>

```
<g transform="skewX(20)">
<switch>
  <!-- select one child element  -->
  <foreignObject x="1em" y="25%" width="10em" height="50%"
     requiredFeatures=
     "http://www.w3.org/TR/SVG11/feature#Extensibility">
     <body xmlns="http://www.w3.org/1999/xhtml">
        <p>This is an XHTML paragraph embedded within an SVG!
           So this text will wrap nicely around multiple lines,
           but it will still be skewed from the SVG transform.
        </p>
     </body>
```

```
  </foreignObject>
  <text x="1em" y="25%" dy="1em">
    This SVG text won't wrap, so it will get cut off...
  </text>

</switch>
</g>
```

Inline SVG in XHTML or HTML5

The other way to mix SVG with XHTML is to include your SVG markup in an XHTML document; it also works with non-XML-compliant HTML documents using the HTML5 syntax. This way of including SVG in a web page is called *Inline SVG* to distinguish it from SVG embedded as an image or object, although it really should be called In*file* SVG, because there's no requirement that your SVG code has to all appear on a single line!

Inline SVG is supported in all major desktop web browsers for versions released in 2012 and later, and most of the latest mobile browsers. For XHTML, you indicate that you're switching to SVG by defining all your SVG elements within the SVG namespace. The easiest way to do this is to set `xmlns="http://www.w3.org/2000/svg"` on the top-level `<svg>` element, which changes the default namespace for that element and all its children. For an HTML5 document (a file with `<!DOCTYPE html>`), you can skip the namespace declaration in your markup. The HTML parser will automatically recognize that `<svg>` elements and all their children—except for children of `<foreignObject>` elements—are within the SVG namespace.

Inserting SVG markup into an (X)HTML document is easier than the reverse: you don't need a separate `<foreignObject>`-like element to define where to render the SVG. Instead, you apply positioning styles to the `<svg>` element itself, making it the frame for your graphic.

By default, the SVG will be positioned with the inline display mode (meaning that it is inserted within the same line as the text before and after it), and will be sized based on the height and width attributes of the `<svg>` element. With CSS, you can change the size by setting the `height` and `width` CSS properties, and change the position with the `display`, `margin`, `padding`, and many other CSS positioning properties.[2]

Example 2-2 gives the code for a very simple SVG drawing in a very simple HTML5 document. The result is Figure 2-3. The `xmlns` attribute on the `<svg>` element is optional for HTML5. For an XHTML document, you would change the `DOCTYPE` declaration at

2. CSS positioning properties apply to top-level `<svg>` elements, ones which are direct children of HTML elements. An `<svg>` that is a child of another SVG element will be positioned based on the rules for nested SVGs, as described in Chapter 3.

the top of the file, and you would wrap the CSS code in the `<style>` element with a `<![CDATA[…]]>` block.

If you do not set the height and width of the SVG with either CSS or attributes, web browsers should apply the default 150-pixel-by-300-pixel size, but be warned! Many versions of browsers apply different defaults. Unfortunately, unlike when using an SVG file in an `<img>` element, you cannot just set one of the height or width and have the SVG scale based on the aspect ratio defined by its `viewBox` attribute.[3]

Example 2-2. Inline SVG within an HTML file

```
<!DOCTYPE html>
<html>
<head>
  <title>SVG in HTML</title>
  <style>

svg {
  display:block; ❶
  width:500px;
  height:500px;
  margin: auto;
  border: thick double navy; ❷
  background-color: lightblue;
}
body {
  font-family: cursive; ❸
}
circle {
  fill: lavender; ❹
  stroke: navy;
  stroke-width: 5;
}

  </style>
</head>
<body>
  <h1>Inline SVG in HTML Demo Page</h1>
    <svg viewBox="0 0 250 250"
         xmlns="http://www.w3.org/2000/svg">
      <title>An SVG circle</title>
      <circle cx="125" cy="125" r="100"/>
      <text x="125" y="125" dy="0.5em" text-anchor="middle">
         Look Ma, Same Font!</text>
    </svg>
  <p>And here is regular HTML again...</p>
```

3. As explained in "Specifying Alignment for preserveAspectRatio" on page 33, the `preserveAspectRatio` attribute will scale an SVG while maintaining its aspect ratio. For inline SVG, this will scale the graphic to fit within the box (height and width) you define for it; it doesn't change the size of the box within the web page.

```
</body>
</html>
```

❶ The first style rules define how the SVG should be positioned and sized within the HTML document.

❷ You can also style the box in which the SVG will be drawn using other CSS properties.

❸ Styles you define for the main document will be inherited by the SVG.

❹ You can also define styles for your SVG elements within your main stylesheet.

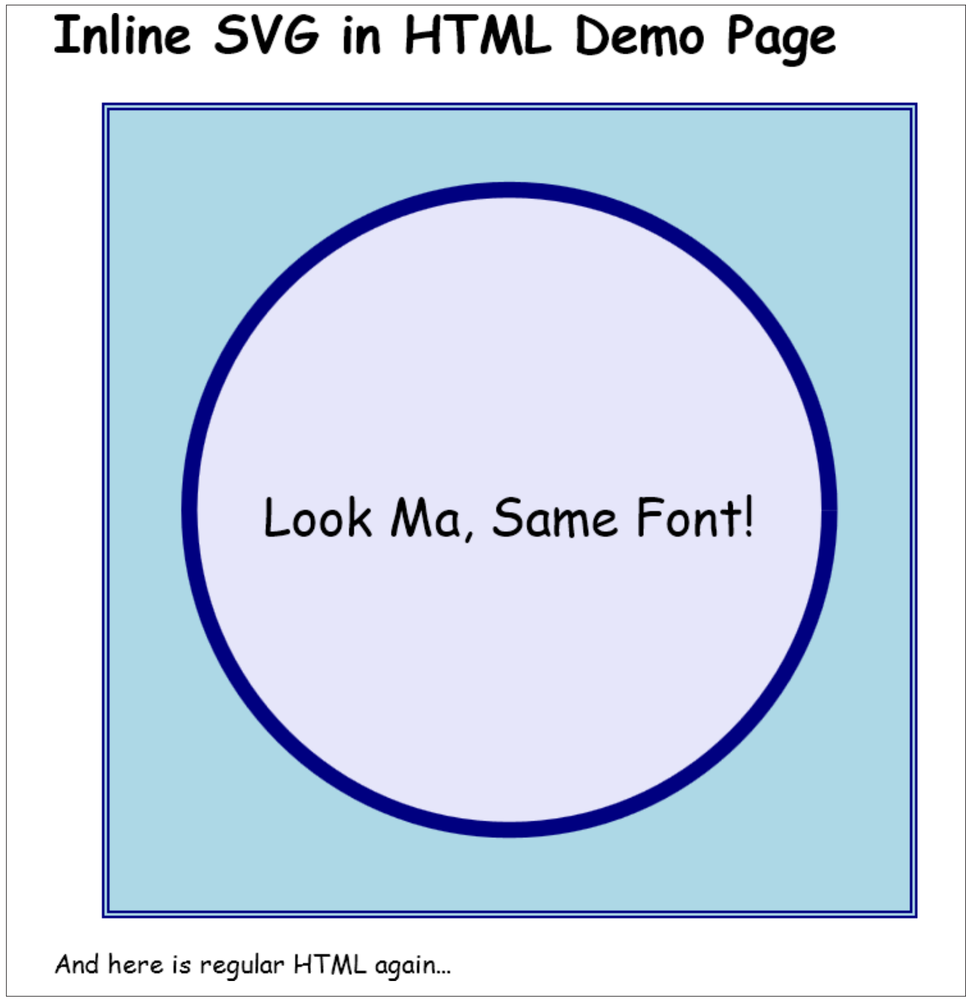


Figure 2-3. The web page from Example 2-2

SVG in Other XML Applications

XML namespaces can be used to identify SVG markup in other XML documents, not just XHTML. The details depend on the main XML document's syntax, but there are two essential requirements: the XML document must clearly define a layout box for the SVG element, and the program that will display the document must know how to draw SVG.

One type of XML document where inline SVG is commonly used is Extensible Stylesheet Language Formatting Object (XSL-FO) files. An XSL-FO file defines both the content and layout of a multipage document, and can be used in publishing or to create a PDF file. The XSL-FO data type definition includes an `<instream-foreign-object>` element, which—just like SVG's `<foreignObject>` element—defines a rectangular region to hold content from a different namespace. Inside it, you can add your SVG markup. Just make sure that the `<svg>` tag and all its children are defined within the SVG namespace, either by using a namespace prefix for all SVG elements or by changing the default namespace with an `xmlns` attribute.

Example 2-3 gives a snippet of an XSL-FO file that uses the customary `fo` namespace prefix for formatting object elements. The SVG namespace is set as the default for the `<svg>` and its children, so no prefixes are necessary within the graphical markup.

Example 2-3. SVG inside an XSL-FO document

```
<?xml version="1.0" encoding="UTF-8"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <!-- other formatting object content -->
  <fo:instream-foreign-object width="140px" height="140px">
    <svg xmlns="http://www.w3.org/2000/svg"
         width="140px" height="140px">
      <!-- SVG code goes here -->
    </svg>
  </fo:instream-foreign-object>
  <!-- rest of document -->
</fo:root>
```

CHAPTER 3

Coordinates

The world of SVG is an infinite canvas. In this chapter, we'll find out how to tell a viewer program which part of this canvas you're interested in, what its dimensions are, and how to locate points within that area.

The Viewport

The area of the canvas your document intends to use is called the *viewport*. You establish the size of this viewport with the `width` and `height` attributes on the `<svg>` element. Each attribute's value can be simply a number, which is presumed to be in pixels; this is said to be specified in user coordinates. You may also specify `width` and `height` as a number followed by a unit identifier, which can be one of the following:

`em`
: The font size of the default font, usually equivalent to the height of a line of text

`ex`
: The height of the letter *x*

`px`
: Pixels (in CSS2-supporting graphics, there are 96 pixels per inch)

`pt`
: Points (1/72 of an inch)

`pc`
: Picas (1/6 of an inch)

`cm`
: Centimeters

`mm`
: Millimeters

`in`
: Inches

Possible SVG viewport declarations include the following:

`<svg width="200" height="150">`
`<svg width="200px" height="150px">`
: Both of these specify an area 200 pixels wide and 150 pixels tall.

`<svg width="2cm" height="3cm">`
: This specifies an area 2 centimeters wide and 3 centimeters high.

`<svg width="2cm" height="36pt">`
: It is possible, though unusual, to mix units; this element specifies an area 2 centimeters wide and 36 points high.

An `<svg>` element may also specify its `width` and `height` as a percentage. When the element is nested within another `<svg>` element, the percentage is measured in terms of the enclosing element. If the `<svg>` element is the root element, the percentage is in terms of the window size. You will see nested `<svg>` elements in "Nested Systems of Coordinates" on page 36.

Using Default User Coordinates

The viewer sets up a coordinate system where the horizontal, or *x*-coordinate, increases as you go to the right, and the vertical, or *y*-coordinate, increases as you move vertically downward. The upper-left corner of the viewport is defined to have an *x*- and *y*-coordinate of 0.[1] This point, written as (0,0), is also called the origin. The coordinate system is a pure geometric system; points have neither width nor height, and the grid lines are considered infinitely thin. You can read more about this subject in Chapter 4.

Example 3-1 establishes a viewport 200 pixels wide and 200 pixels high, and then draws a rectangle whose upper-left corner is at coordinate (10,10) with a width of 50 pixels and a height of 30 pixels.[2] Figure 3-1 shows the result, with rulers and a grid to show the coordinate system.

1. In this book, coordinates are specified as a pair of numbers in parentheses, with the *x*-coordinate first. Thus, (10,30) represents an *x*-coordinate of 10 and a *y*-coordinate of 30.

2. To save space, we are leaving out the `<?xml …?>` and `<!DOCTYPE …>` lines. These are set in stone, so you can take them for granite.

Example 3-1. Using default coordinates

http://oreillymedia.github.io/svg-essentials-examples/ch03/default_coordinates.html

```
<svg width="200" height="200">
    <rect x="10" y="10" width="50" height="30"
        style="stroke: black; fill: none;"/>
</svg>
```

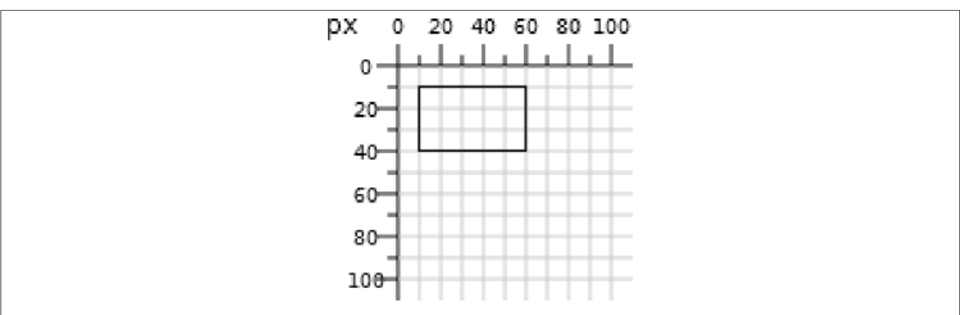


Figure 3-1. Rectangle using default coordinates

Even if you don't specify units in the viewport, you may still use them in some SVG shape elements, as in Example 3-2. Figure 3-2 shows the result, with rulers and a grid to show the coordinate system.

Example 3-2. Explicit use of units

http://oreillymedia.github.io/svg-essentials-examples/ch03/explicit_units.html

```
<svg width="200" height="200">
    <rect x="10mm" y="10mm" width="15mm" height="10mm"
        style="stroke:black; fill:none;"/>
</svg>
```

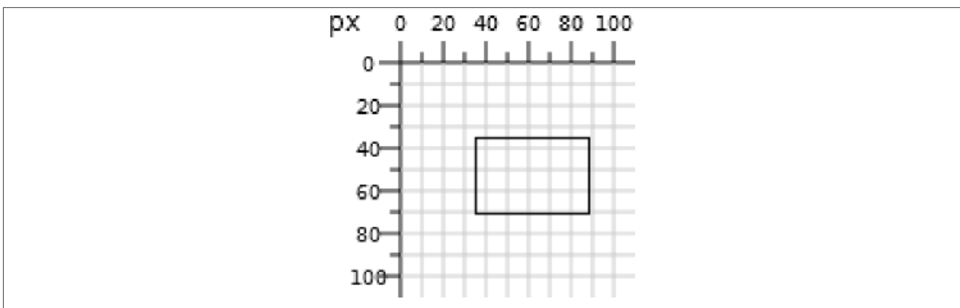


Figure 3-2. Rectangle using explicit units

Specifying units in the <svg> element does not affect coordinates given without units in other elements. Example 3-3 shows a viewport set up in millimeters, but the rectangle is still drawn at pixel (user) coordinates, as you see in Figure 3-3.

Example 3-3. Units on the svg element

http://oreillymedia.github.io/svg-essentials-examples/ch03/units_on_svg.html

```
<svg width="70mm" height="70mm">
    <rect x="10" y="10" width="50" height="30"
      style="fill: none; stroke: black;"/>
</svg>
```

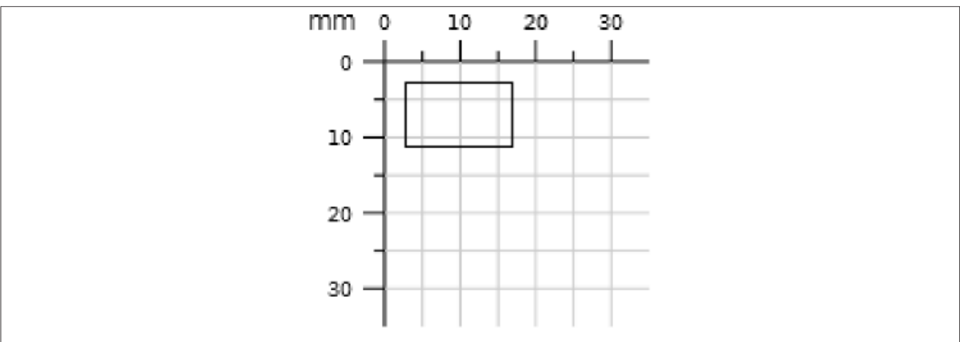


Figure 3-3. Viewport with units; rectangle without units

Specifying User Coordinates for a Viewport

In the examples so far, numbers without units have been considered to be pixels. Sometimes this is not what you want. For example, you might want to set up a system where each user coordinate represents 1/16th of a centimeter. (We're using this coordinate system to prove a point, not to show a paragon of good design.) In this system, a square that is 40 units by 40 units will display as 2.5 centimeters on a side.

To accomplish this effect, you set the viewBox attribute on the <svg> element. The value of this attribute consists of four numbers that represent the minimum *x*-coordinate, minimum *y*-coordinate, width, and height of the user coordinate system you want to superimpose on the viewport.

So, to set up the 16-units-per-centimeter coordinate system for a 4-centimeter by 5-centimeter drawing, you'd use this starting tag:

```
<svg width="4cm" height="5cm" viewBox="0 0 64 80">
```

Example 3-4 gives the SVG for a picture of a house, displayed using the new coordinate system. Figure 3-4 shows the result. The grid and darker numbers show the new user coordinate system; the lighter numbers are positioned at 1-centimeter intervals.

Example 3-4. Using a viewBox

http://oreillymedia.github.io/svg-essentials-examples/ch03/using_viewbox.html

```
<svg width="4cm" height="5cm" viewBox="0 0 64 80">
  <rect x="10" y="35" width="40" height="40"
    style="stroke: black; fill: none;"/>
  <!-- roof -->
  <polyline points="10 35, 30 7.68, 50 35"
    style="stroke:black; fill: none;"/>
  <!-- door -->
  <polyline points="30 75, 30 55, 40 55, 40 75"
    style="stroke:black; fill: none;"/>
</svg>
```

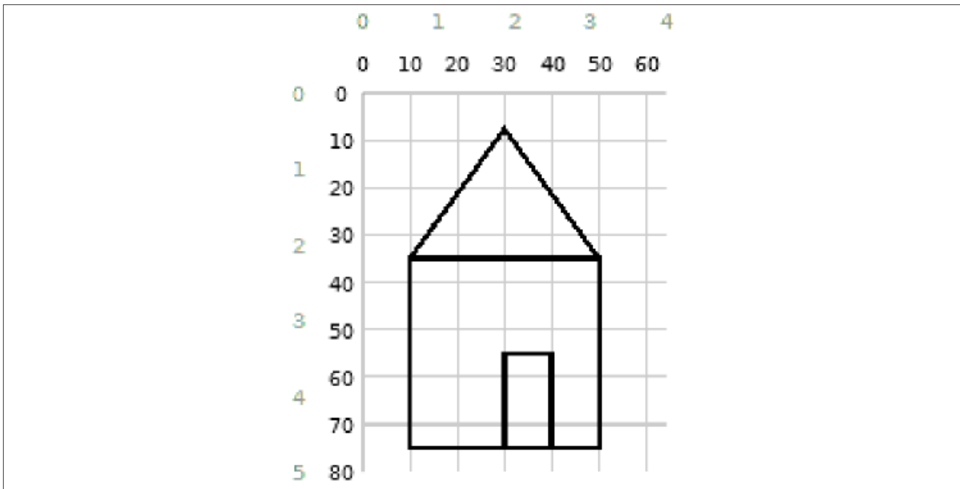
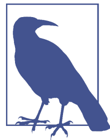


Figure 3-4. New user coordinates

The numbers you specify for the value of the `viewBox` attribute may be separated by commas or whitespace. If either the width or height is 0, none of your graphic will display. It is an error to specify a negative value for the `viewBox` width or height.



If you were reading the code in Example 3-4 carefully, you would have noted that we used a decimal value to get the peak of the house's roof positioned just right. Nearly all numbers in SVG are floating-point decimal numbers. SVG viewers are required to support at least 32-bit precision numbers and are encouraged to use higher precision numbers for some calculations. In fact, you can even use scientific notation to work in a coordinate system with very large or small numbers, so that the point `30,7.68` *could* have been written like `3.0E+1,7.68e0`. But for readability and brevity, we wouldn't recommend it—reserve the scientific notation for when it is really necessary.

Preserving Aspect Ratio

In the previous example, the aspect ratio, or ratio of width to height, of the viewport and the `viewBox` were identical (4/5 = 64/80). What happens, though, if the aspect ratio of the viewport and the `viewBox` are not the same, as in this example, where `viewBox` has an aspect ratio of 1:1 (the width and height are equal), but the viewport has an aspect ratio of 1:3 (the height is three times as big as the width)?

```
<svg width="45px" height="135px" viewBox="0 0 90 90">
```

There are three things SVG can do in this situation:

- Scale the graphic uniformly according to the smaller dimension so the graphic will fit entirely into the viewport. In the example, the picture would become half its original width and height. You'll see examples of this in "Using the meet Specifier" on page 34.
- Scale the graphic uniformly according to the larger dimension and cut off the parts that lie outside the viewport. In the example, the picture would become one and a half times its original width and height. You'll see examples of this in "Using the slice Specifier" on page 35.
- Stretch and squash the drawing so it fits precisely into the new viewport. (That is, don't preserve the aspect ratio at all.) See the details in "Using the none Specifier" on page 36.

In the first case, because the image will be smaller than the viewport in one dimension, you must specify where to position it. In the example, the picture will be scaled uniformly to a width and height of 45 pixels. The width of the reduced graphic fits the width of the viewport perfectly, but you must now decide whether the image meets (is aligned with) the top, middle, or bottom of the 135-pixel viewport height.

In the second case, because the image will be larger than the viewport in one dimension, you must specify which area is to be sliced away. In the example, the picture will be scaled uniformly to a width and height of 135 pixels. Now the height of the graphic fits

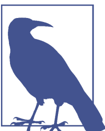
the viewport perfectly, but you must decide whether to slice off the right side, left side, or both edges of the picture to fit within the 45-pixel viewport width.

Specifying Alignment for preserveAspectRatio

The `preserveAspectRatio` attribute lets you specify the alignment of the scaled image with respect to the viewport, and whether you want it to meet the edges or be sliced off. The model for this attribute is

```
preserveAspectRatio="alignment [meet | slice]"
```

where *`alignment`* specifies the axis and location and is one of the combinations shown in Table 3-1. This alignment specifier is formed by concatenating an *x*-alignment and a *y*-alignment `min`, `mid` (middle), or `max` value. The default value for `preserveAspectRatio` is `xMidYMid meet`.



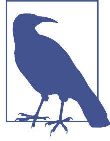
The *y*-alignment begins with a capital letter, because the *x*- and *y*-alignments are concatenated into a single word.

Table 3-1. Values for alignment portion of preserveAspectRatio

	X Alignment		
Y Alignment	`xMin` **Align minimum *x* value of `viewBox` with left edge of viewport**	`xMid` **Align midpoint *x* value of `viewBox` with horizontal center of viewport**	`xMax` **Align maximum *x* value of `viewBox` with right edge of viewport**
`yMin` **Align minimum *y* value of `viewBox` with top edge of viewport**	`xMinYMin`	`xMidYMin`	`xMaxYMin`
`yMid` **Align midpoint *y* value of `viewBox` with vertical center of viewport**	`xMinYMid`	`xMidYMid`	`xMaxYMid`
`yMax` **Align maximum *y* value of `viewBox` with bottom edge of viewport**	`xMinYMax`	`xMidYMax`	`xMaxYMax`

Thus, if you want to have the picture with a `viewBox="0 0 90 90"` fit entirely within a viewport that is 45 pixels wide and 135 pixels high, aligned at the top of the viewport, you would write the following:

```
<svg width="45px" height="135px" viewBox="0 0 90 90"
  preserveAspectRatio="xMinYMin meet">
```



In this case, because the width fits precisely, the *x*-alignment is irrelevant; you could equally well use `xMidYMin` or `xMaxYMin`. However, you normally use `preserveAspectRatio` when you don't know the aspect ratio of the viewport. For example, you might want the image to scale to fit the application window, or you might let the CSS of a parent document set the height and width. In those situations, you need to consider how you want your image to display when the viewport is too wide as well as when it is too tall.

If you don't specify a `preserveAspectRatio`, the default value is `xMidYMid meet`, which will scale down the graphic to fit the available space, and center it both horizontally and vertically.

This is all fairly abstract; the following sections give some concrete examples that show you how the combinations of alignment and `meet` and `slice` interact with one another.

Using the meet Specifier

The starting `<svg>` tags in Example 3-5 all use the `meet` specifier.

Example 3-5. Use of meet specifier

```
<!-- tall viewports -->
<svg preserveAspectRatio="xMinYMin meet" viewBox="0 0 90 90"
    width="45" height="135">

<svg preserveAspectRatio="xMidYMid meet" viewBox="0 0 90 90"
    width="45" height="135">

<svg preserveAspectRatio="xMaxYMax meet" viewBox="0 0 90 90"
    width="45" height="135">

<!-- wide viewports -->
<svg preserveAspectRatio="xMinYMin meet" viewBox="0 0 90 90"
    width="135" height="45">

<svg preserveAspectRatio="xMidYMid meet" viewBox="0 0 90 90"
    width="135" height="45">

<svg preserveAspectRatio="xMaxYMax meet" viewBox="0 0 90 90"
    width="135" height="45">
```

Figure 3-5 shows where the reduced image fits into the enclosing `viewBox`.

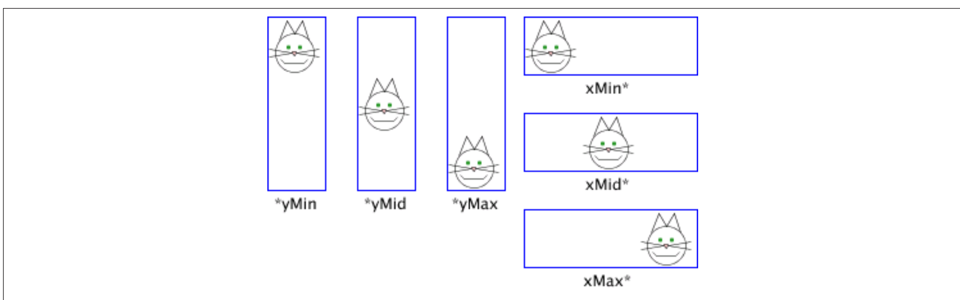


Figure 3-5. meet—viewBox fits in viewport

Using the slice Specifier

Figure 3-6 shows the use of the `slice` specifier to eliminate parts of the picture that do not fit in the viewport. They were created with the `<svg>` tags in Example 3-6.

Example 3-6. Use of slice specifier

```
<!-- tall viewports -->
<svg preserveAspectRatio="xMinYMin slice" viewBox="0 0 90 90"
    width="45" height="135">

<svg preserveAspectRatio="xMidYMid slice" viewBox="0 0 90 90"
    width="45" height="135">

<svg preserveAspectRatio="xMaxYMax slice" viewBox="0 0 90 90"
    width="45" height="135">

<!-- wide viewports -->
<svg preserveAspectRatio="xMinYMin slice" viewBox="0 0 90 90"
    width="135" height="45">

<svg preserveAspectRatio="xMidYMid slice" viewBox="0 0 90 90"
    width="135" height="45">

<svg preserveAspectRatio="xMaxYMax slice" viewBox="0 0 90 90"
    width="135" height="45">
```

The online example for this section allows you to experiment with the different `preserveAspectRatio` options to slice, shrink, and shift the cat around any sized SVG:

http://oreillymedia.github.io/svg-essentials-examples/ch03/meet_slice_specifier.html

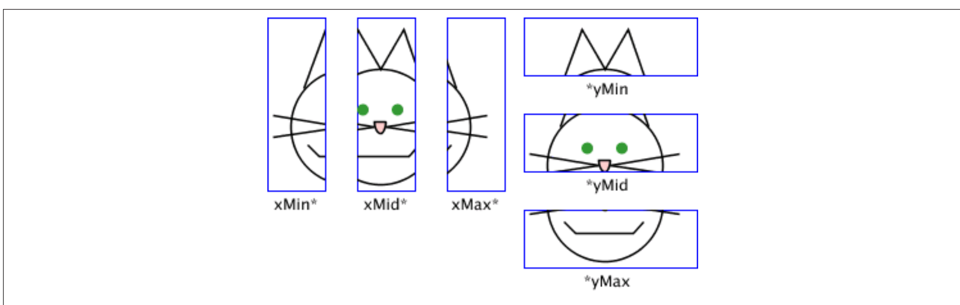


Figure 3-6. slice—graphic fills viewport

Using the none Specifier

Finally, there is the third option for scaling a graphic when the `viewBox` and viewport don't have the same aspect ratio. If you specify `preserveAspectRatio="none"`, then the graphic will be scaled nonuniformly so its user coordinates fit the viewport. Figure 3-7 shows such a "fun-house mirror" effect produced with the `<svg>` tags in Example 3-7.

Example 3-7. Aspect ratio not preserved

```
<!-- tall viewport -->
<svg preserveAspectRatio="none" viewBox="0 0 90 90"
   width="45" height="135">

<!-- wide viewport -->
<svg preserveAspectRatio="none" viewBox="0 0 90 90"
  width="135" height="45">
```

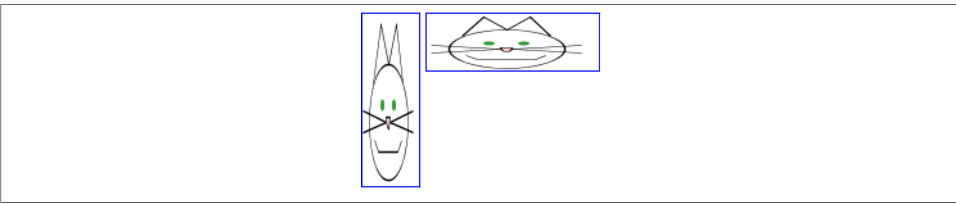


Figure 3-7. Aspect ratio not preserved

Nested Systems of Coordinates

You can establish a new viewport and system of coordinates at any time by putting another `<svg>` element into your document. The effect is to create a "mini-canvas" upon which you can draw. We used this technique to create illustrations such as Figure 3-5. Rather than drawing the rectangles, then rescaling and positioning the cat inside each one (the brute-force approach), we took these steps:

- Draw the blue rectangles on the main canvas
- For each rectangle, define a new `<svg>` element with the appropriate `preserveAspectRatio` attribute
- Draw the cat into that new canvas (with `<use>`), and let SVG do the heavy lifting

Here's a simplified example that shows a circle on the main canvas, then inside a new canvas outlined by a blue rectangle that's also on the main canvas. Figure 3-8 is the desired result.

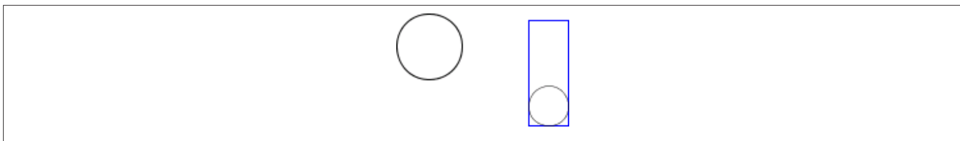


Figure 3-8. Nested viewports

First, generate the SVG for the main coordinate system and the circle (note that the user coordinates coincide exactly with the viewport in this document):

```
<svg width="200px" height="200px" viewBox="0 0 200 200">
    <circle cx="25" cy="25" r="25" style="stroke: black; fill: none;"/>
</svg>
```

The result is in Figure 3-9.

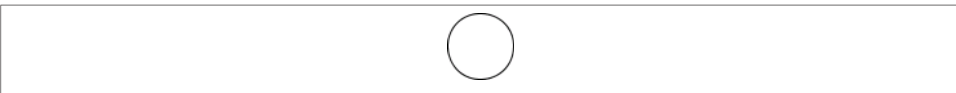


Figure 3-9. Circle in main viewport

Now, draw the boundary of the box showing where you want the new viewport to be:

```
<svg width="200px" height="200px" viewBox="0 0 200 200">
    <circle cx="25" cy="25" r="25" style="stroke: black; fill: none;"/>
    <rect x="100" y="5" width="30" height="80"
       style="stroke: blue; fill: none;"/>
</svg>
```

This produces Figure 3-10.

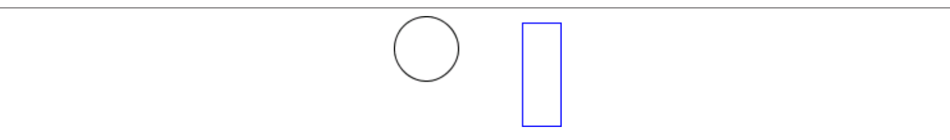


Figure 3-10. Circle and boundary box in main viewport

Now, add another `<svg>` element for the new viewport. In addition to specifying the `viewBox`, `width`, `height`, and `preserveAspectRatio` specification, you may also specify the `x` and `y` attributes—in terms of the enclosing `<svg>` element—where the new viewport is to be established (if you don't give values for `x` and `y`, they are presumed to be 0):

```
<svg width="200px" height="200px" viewBox="0 0 200 200">
    <circle cx="25" cy="25" r="25" style="stroke: black; fill: none;"/>
    <rect x="100" y="5" width="30" height="80"
        style="stroke: blue; fill: none;"/>

    <svg x="100px" y="5px" width="30px" height="80px"
      viewBox="0 0 50 50" preserveAspectRatio="xMaxYMax meet">
    </svg>
</svg>
```

Setting up the new coordinates with this nested `<svg>` element doesn't change the visual display, but it does permit you to add the circle in that new system, producing the result shown in Figure 3-8:

```
<svg width="200px" height="200px" viewBox="0 0 200 200">
  <circle cx="25" cy="25" r="25" style="stroke: black; fill: none;"/>
  <rect x="100" y="5" width="30" height="80" style="stroke: blue;
    fill: none;"/>

  <svg x="100px" y="5px" width="30px" height="80px" viewBox="0 0 50 50"
    preserveAspectRatio="xMaxYMax meet">
    <circle cx="25" cy="25" r="25" style="stroke: black;
      fill: none;"/>
  </svg>
</svg>
```

If you try to use a `meet` or `slice` value for the `preserveAspectRatio` attribute on an `<svg>` nested inside another `<svg>` with `preserveAspectRatio="none"`, the results may surprise you. The aspect ratio of the nested element's viewport will be evaluated in the squished or stretched coordinates of the parent SVG, possibly resulting in an image that is both squished *and* cropped or shrunk to fit.

CHAPTER 4
Basic Shapes

Once a coordinate system is established in the `<svg>` tag, you are ready to begin drawing. This chapter describes the basic shapes you can use to create the major elements of most drawings: lines, rectangles, polygons, circles, and ellipses.

Lines

SVG lets you draw a straight line with the `<line>` element. Just specify the *x*- and *y*-coordinates of the line's endpoints. Coordinates may be specified without units, in which case they are considered to be user coordinates, or with units such as `em`, `in`, etc. (as described in Chapter 3, in "The Viewport" on page 27).

```
<line x1="start-x" y1="start-y"
  x2="end-x" y2="end-y" />
```

The SVG in Example 4-1 draws several lines; the reference grid in Figure 4-1 is not part of the SVG you see in the example.

Example 4-1. Basic lines

http://oreillymedia.github.io/svg-essentials-examples/ch04/basic-lines.html

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
  <!-- horizontal line -->
  <line x1="40" y1="20" x2="80" y2="20" style="stroke: black;"/>
  <!-- vertical line -->
  <line x1="0.7cm" y1="1cm" x2="0.7cm" y2="2.0cm"
    style="stroke: black;"/>
   <!-- diagonal line -->
   <line x1="30" y1="30" x2="85" y2="85" style="stroke: black;"/>
</svg>
```

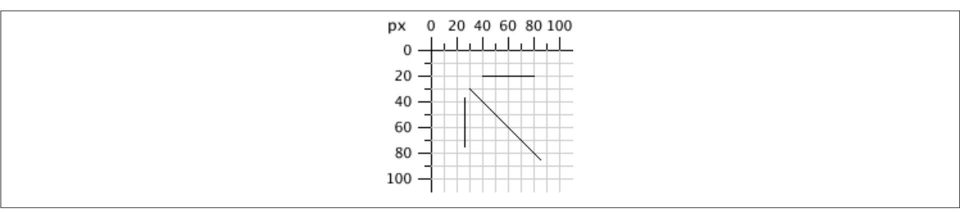


Figure 4-1. Basic lines

Stroke Characteristics

Lines are considered to be strokes of a pen that draws on the canvas. The size, color, and style of the pen stroke are part of the line's presentation. Thus, these characteristics will go into the `style` attribute.

stroke-width

As mentioned in Chapter 3, the canvas grid lines are infinitely thin. Where, then, does a line or stroke fall in relation to the grid line? The answer is that the grid line falls in the center of a stroke. Example 4-2 draws some lines where the stroke width has been set to 10 user coordinates to make the effect obvious. The result, in Figure 4-2, has the grid lines drawn in so you can see the effect clearly.

Example 4-2. Demonstration of stroke-width

http://oreillymedia.github.io/svg-essentials-examples/ch04/stroke-width.html

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
  <!-- horizontal line -->
  <line x1="30" y1="10" x2="80" y2="10"
    style="stroke-width: 10; stroke: black;"/>
  <!-- vertical line -->
  <line x1="10" y1="30" x2="10" y2="80"
    style="stroke-width: 10; stroke: black;"/>
  <!-- diagonal line -->
  <line x1="25" y1="25" x2="75" y2="75"
    style="stroke-width: 10; stroke: black;"/>
</svg>
```

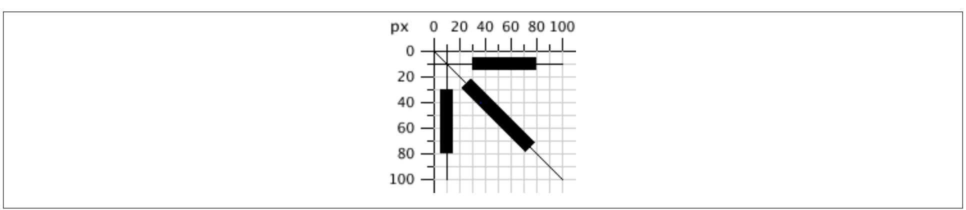
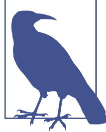


Figure 4-2. Demonstration of stroke-width



The SVG coordinate grid may be infinitely thin, but your computer screen is made of fixed-size pixels. A diagonal line can look jagged as the computer translates it to the nearest pixel blocks; this is known as *aliasing*. Alternatively, the computer can use *anti-aliasing* to soften the edges, blurring the line across all pixels it partially overlaps.

Most SVG viewers use anti-aliasing by default, and this can sometimes make a 1-pixel black line look like a 2-pixel gray line, because it is centered on the space between two pixels. You can control the use of anti-aliasing with the CSS `shape-rendering` style property. Setting this property to `crispEdges` (on an element or the SVG as a whole) will turn off anti-aliasing, resulting in clear (if sometimes jagged) lines. A value of `geometricPrecision` will emphasize smooth (if sometimes blurry) edges.

Stroke Color

You can specify the stroke color in a variety of ways:

- One of the basic color keyword names: `aqua`, `black`, `blue`, `fuchsia`, `gray`, `green`, `lime`, `maroon`, `navy`, `olive`, `purple`, `red`, `silver`, `teal`, `white`, and `yellow`. You may also use the color keywords from section 4.2 of the SVG specification.
- A six-digit hexadecimal specifier in the form `#rrggbb`, where `rr` is the red component, `gg` is the green component, and `bb` is the blue component in the range `00–ff`.
- A three-digit hexadecimal specifier in the form `#rgb`, where `r` is the red component, `g` is the green component, and `b` is the blue component in the range `0–f`. This is a shorthand form of the previous method of specifying color. To produce the six-digit equivalent, each digit of the short form is duplicated; thus `#d6e` is the same as `#dd66ee`.
- An `rgb` specifier in the form `rgb(red-value, green-value, blue-value)`, where each value is an integer in the range `0–255` or a percentage in the range `0%` to `100%`.

- The `currentColor` keyword, which uses the computed CSS `color` property for the element. The `color` property—which doesn't have a direct effect in SVG—is used in HTML to set text color, and is inherited by child elements. Using `currentColor` in an inline SVG icon (see "Inline SVG in XHTML or HTML5" on page 22) allows the icon to take on the color of the surrounding text.

Example 4-3 uses all of these methods (with the exception of `currentColor`), giving the colorful results of Figure 4-3.

Example 4-3. Demonstration of stroke color

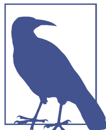
http://oreillymedia.github.io/svg-essentials-examples/ch04/stroke-color.html

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
  <!-- red -->
  <line x1="10" y1="10" x2="50" y2="10"
    style="stroke: red; stroke-width: 5;"/>

  <!-- light green -->
  <line x1="10" y1="20" x2="50" y2="20"
    style="stroke: #9f9; stroke-width: 5;"/>

  <!-- light blue -->
  <line x1="10" y1="30" x2="50" y2="30"
    style="stroke: #9999ff; stroke-width: 5;"/>

  <!-- medium orange -->
  <line x1="10" y1="40" x2="50" y2="40"
    style="stroke: rgb(255, 128, 64); stroke-width: 5;"/>

  <!-- deep purple -->
  <line x1="10" y1="50" x2="50" y2="50"
    style="stroke: rgb(60%, 20%, 60%); stroke-width: 5;"/>
</svg>
```

Figure 4-3. Demonstration of stroke color

There are yet more ways to specify color. They are taken from the CSS3 Color specification. Although widely supported in web browsers, they are not part of the SVG 1.1 specification, and may not be supported by other SVG implementations; as of this writing, for example, neither Apache Batik nor Inkscape supports them. There are three new color functions and one new keyword:

- `rgba()` specifier in the form `rgb(`*`red-value`*`, `*`green-value`*`, `*`blue-value`*`, `*`alpha-value`*`)`, where the color values are in the same format as for the `rgb()` function, and the alpha value is a decimal in the range `0–1`
- `hsl()` specifier in the form `hsl(`*`hue`*`, `*`saturation`*`, `*`lightness`*`)`, where *hue* is an integer angle from 0 to 360, and *saturation* and *lightness* are integers in the range `0–255` or percentages in the range `0%` to `100%`
- `hsla()` specifier, with the hue, saturation, and lightness values the same as for `hsl`, and the alpha value the same as for `rgba`
- `transparent` (fully transparent); this is the same as `rgba(0, 0, 0, 0)`



If you do not specify a stroke color, you won't see any lines; the default value for the `stroke` property is `none`.

stroke-opacity

Up to this point, all the lines in the example have been solid, obscuring anything beneath them. You control the opacity (which is the opposite of transparency) of a line by giving the `stroke-opacity` a value from `0.0` to `1.0`, where 0 is completely transparent and 1 is completely opaque. A value less than 0 will be changed to 0; a value greater than 1 will be changed to 1. Example 4-4 varies the opacity from 0.2 to 1 in steps of 0.2, with the result in Figure 4-4. The red line in the figure lets you see the transparency effect more clearly.

Example 4-4. Demonstration of stroke-opacity

http://oreillymedia.github.io/svg-essentials-examples/ch04/stroke-opacity.html

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
  <line x1="30" y1="0" x2="30" y2="60"
        style="stroke:red; stroke-width: 5;"/>
  <line x1="10" y1="10" x2="50" y2="10"
    style="stroke-opacity: 0.2; stroke: black; stroke-width: 5;"/>
  <line x1="10" y1="20" x2="50" y2="20"
    style="stroke-opacity: 0.4; stroke: black; stroke-width: 5;"/>
  <line x1="10" y1="30" x2="50" y2="30"
    style="stroke-opacity: 0.6; stroke: black; stroke-width: 5;"/>
  <line x1="10" y1="40" x2="50" y2="40"
    style="stroke-opacity: 0.8; stroke: black; stroke-width: 5;"/>
  <line x1="10" y1="50" x2="50" y2="50"
    style="stroke-opacity: 1.0; stroke: black; stroke-width: 5;"/>
</svg>
```

Figure 4-4. Demonstration of stroke-opacity

stroke-dasharray Attribute

If you need dotted or dashed lines, use the `stroke-dasharray` attribute, whose value consists of a list of numbers, separated by commas or whitespace, specifying dash length and gaps. The list should have an even number of entries, but if you give an odd number of entries, SVG will repeat the list so the total number of entries is even. (See the last instance in Example 4-5.)

Example 4-5. Demonstration of stroke-dasharray

http://oreillymedia.github.io/svg-essentials-examples/ch04/stroke-dasharray.html

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
   <!-- 9-pixel dash, 5-pixel gap -->
   <line x1="10" y1="10" x2="100" y2="10"
       style="stroke-dasharray: 9, 5;
       stroke: black; stroke-width: 2;"/>

   <!-- 5-pixel dash, 3-pixel gap, 9-pixel dash, 2-pixel gap -->
   <line x1="10" y1="20" x2="100" y2="20"
       style="stroke-dasharray: 5, 3, 9, 2;
       stroke: black; stroke-width: 2;"/>

   <!-- Odd number of entries is duplicated; this is equivalent to:
        9-pixel dash, 3-pixel gap,  5-pixel dash,
        9-pixel gap,  3-pixel dash, 5-pixel gap -->
   <line x1="10" y1="30" x2="100" y2="30"
       style="stroke-dasharray: 9 3 5;
       stroke: black; stroke-width: 2;"/>
</svg>
```

Figure 4-5 shows the results, zoomed in for clarity.

Figure 4-5. Demonstration of stroke-dasharray

Rectangles

The rectangle is the simplest of the basic shapes. You specify the *x*- and *y*-coordinates of the upper-left corner of the rectangle,[1] its `width`, and its `height`. The interior of the rectangle is filled with the `fill` color you specify. If you do not specify a fill color, the interior of the shape is filled with black. The fill color may be specified in any of the ways described in "Stroke Color" on page 41, or it may take the value `none` to leave the interior unfilled and thus transparent. You may also specify a `fill-opacity` in the same format as you did for `stroke-opacity` in "stroke-opacity" on page 43. Both `fill` and `fill-opacity` are presentation properties, and they belong in the `style` attribute.

After the interior is filled (if necessary), the outline of the rectangle is drawn with strokes, whose characteristics you may specify as you did for lines. If you do not specify a stroke, the value `none` is presumed, and no outline is drawn. Example 4-6 draws several variations of the `<rect>` element. Figure 4-6 shows the result, with a grid for reference.

Example 4-6. Demonstration of the rectangle element

http://oreillymedia.github.io/svg-essentials-examples/ch04/rectangle.html

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
  <!-- black interior, no outline -->
  <rect x="10" y="10" width="30" height="50"/>

  <!-- no interior, black outline -->
  <rect x="50" y="10" width="20" height="40"
    style="fill: none; stroke: black;"/>

  <!-- blue interior, thick semi-transparent red outline -->
  <rect x="10" y="70" width="25" height="30"
    style="fill: #0000ff;
      stroke: red; stroke-width: 7; stroke-opacity: 0.5;"/>

  <!-- semi-transparent yellow interior, dashed green outline -->
  <rect x="50" y="70" width="35" height="20"
    style="fill: yellow; fill-opacity: 0.5;
      stroke: green; stroke-width: 2; stroke-dasharray: 5 2"/>
</svg>
```

1. Technically, the `x` value is the smaller of the *x*-coordinate values, and the `y` is the smaller of the *y*-coordinate values of the rectangle's sides in the current user coordinate system. Because you are not yet using transformations, which are covered in Chapter 6, this is the moral equivalent of the upper-left corner.

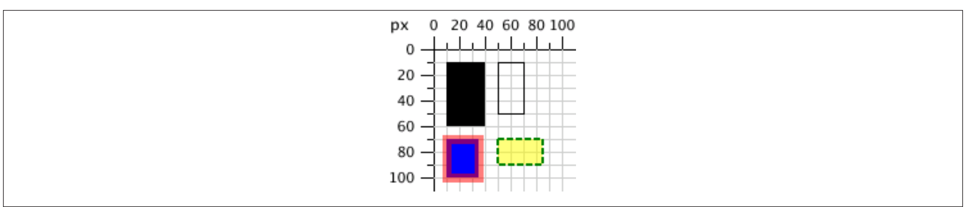
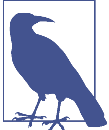


Figure 4-6. Demonstration of the rect element



The strokes that form the outline "straddle" the abstract grid lines, so the strokes will be half inside the shape and half outside the shape. Figure 4-7, a close-up of the semi-transparent red outline drawn in Example 4-6, shows this clearly.

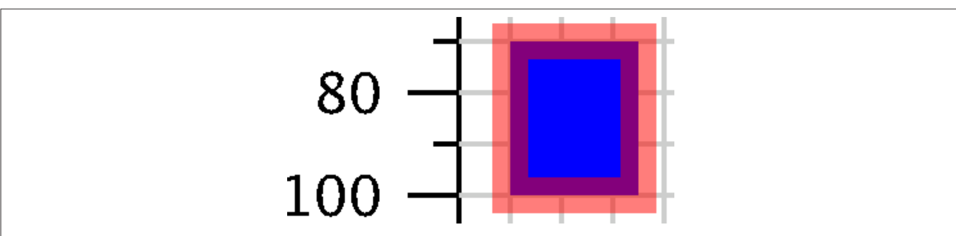


Figure 4-7. Close-up of transparent border

If you do not specify a starting `x` or `y` value, it is presumed to be 0. If you specify a `width` or `height` of 0, then the rectangle is not displayed. It is an error to provide negative values for either `width` or `height`.

Rounded Rectangles

If you wish to have rectangles with rounded corners, specify the *x*- and *y*-radius of the corner curvature. The maximum number you may specify for `rx` (the *x*-radius) is one-half the width of the rectangle; the maximum value of `ry` (the *y*-radius) is one-half the height of the rectangle. If you specify only one of `rx` or `ry`, they are presumed to be equal. Example 4-7 shows various combinations of `rx` and `ry`.

Example 4-7. Demonstration of rounded rectangles

http://oreillymedia.github.io/svg-essentials-examples/ch04/rounded-rectangles.html

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
  <!-- rx and ry equal, increasing -->
  <rect x="10" y="10" width="20" height="40" rx="2" ry="2"
```

```
    style="stroke: black; fill: none;"/>

  <rect x="40" y="10" width="20" height="40" rx="5"
    style="stroke: black; fill: none;"/>

  <rect x="70" y="10" width="20" height="40" ry="10"
    style="stroke: black; fill: none;"/>

  <!-- rx and ry unequal -->
  <rect x="10" y="60" width="20" height="40" rx="10" ry="5"
    style="stroke: black; fill: none;"/>

  <rect x="40" y="60" width="20" height="40" rx="5" ry="10"
    style="stroke: black; fill: none;"/>
</svg>
```

Figure 4-8 shows the result, with a grid in the background for reference.

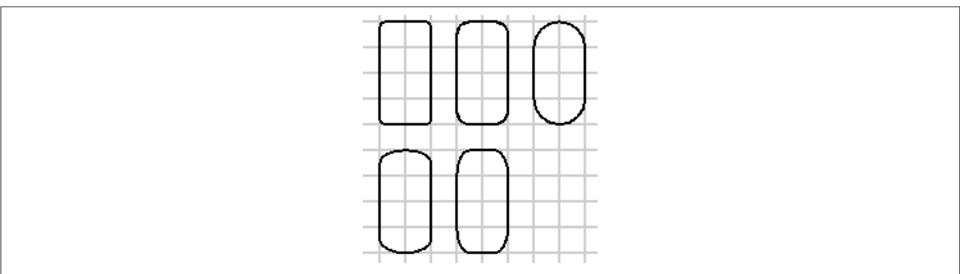
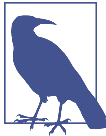


Figure 4-8. Demonstration of rounded rectangles



If you're familiar with the CSS `border-radius` property, you might know the trick of turning a rectangle into a circle or ellipse by setting the corner radius to 50% of the height and width. Although you can specify an SVG rectangle's corner radius with percent values, they will be interpreted as a percent of the viewport width (`rx`) or height (`ry`)—the same as if you used a percentage for setting the rectangle's width or height—not as a percentage of the rectangle itself. Good thing SVG has an easier way to create circles and ellipses…

Circles and Ellipses

To draw a circle, use the `<circle>` element and specify the center *x*-coordinate, center *y*-coordinate, and radius with the `cx`, `cy`, and `r` attributes. As with a rectangle, the default is to fill the circle with black and draw no outline unless you specify some other combination of `fill` and `stroke`.

An ellipse needs both an *x*-radius and a *y*-radius in addition to a center *x*- and *y*-coordinate. The attributes for these radii are named `rx` and `ry`.

In both circles and ellipses, if the `cx` or `cy` is omitted, it is presumed to be 0. If the radius is 0, no shape will be displayed; it is an error to provide a negative radius. Example 4-8 draws some circles and ellipses. They are shown in Figure 4-9.

Example 4-8. Demonstration of circles and ellipses

http://oreillymedia.github.io/svg-essentials-examples/ch04/circles-ellipses.html

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
  <circle cx="30" cy="30" r="20" style="stroke: black; fill: none;"/>
  <circle cx="80" cy="30" r="20"
    style="stroke-width: 5; stroke: black; fill: none;"/>

  <ellipse cx="30" cy="80" rx="10" ry="20"
    style="stroke: black; fill: none;"/>
  <ellipse cx="80" cy="80" rx="20" ry="10"
    style="stroke: black; fill: none;"/>
</svg>
```

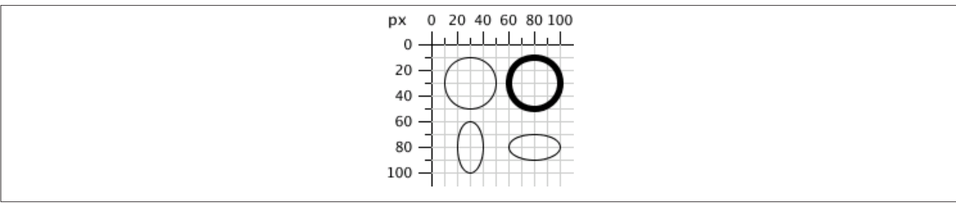


Figure 4-9. Demonstration of circle and ellipse elements

The <polygon> Element

In addition to rectangles, circles, and ellipses, you may want to draw hexagons, octagons, stars, or arbitrary closed shapes. The `<polygon>` element lets you specify a series of `points` that describe a geometric area to be filled and outlined as described earlier. The `points` attribute consists of a series of *x*- and *y*-coordinate pairs separated by commas or whitespace. You must give an even number of entries in the series of numbers. You don't have to return to the starting point; the shape will automatically be closed. Example 4-9 uses the `<polygon>` element to draw a parallelogram, a star, and an irregular shape.

Example 4-9. Demonstration of the polygon element

http://oreillymedia.github.io/svg-essentials-examples/ch04/polygon.html

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
  <!-- parallelogram -->
  <polygon points="15,10  55, 10  45, 20  5, 20"
    style="fill: red; stroke: black;"/>

  <!-- star -->
  <polygon
    points="35,37.5  37.9,46.1 46.9,46.1  39.7,51.5
      42.3,60.1  35,55  27.7,60.1  30.3,51.5
      23.1,46.1  32.1,46.1"
      style="fill: #ccffcc; stroke: green;"/>

  <!-- weird shape -->
  <polygon
    points="60 60,  65 72,  80 60,  90 90, 72 80, 72 85, 50 95"
    style="fill: yellow; fill-opacity: 0.5; stroke: black;
      stroke-width: 2;"/>
</svg>
```

The results, with a grid in the background for reference, are displayed in Figure 4-10.

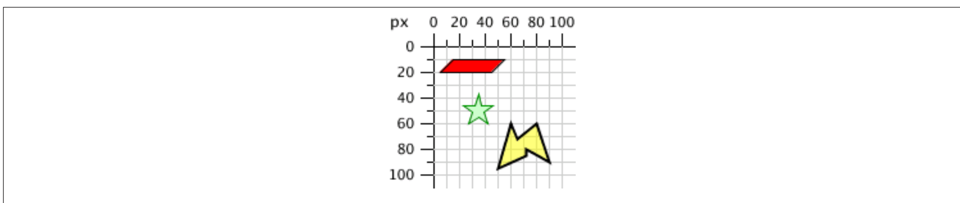


Figure 4-10. Demonstration of the polygon element

Filling Polygons That Have Intersecting Lines

For the polygons shown so far, it's been easy to fill the shape. None of the lines forming the polygon cross over one another, so the interior is easily distinguished from the exterior of the shape. However, when lines cross over one another, the determination of what is inside the polygon is not as easy. The SVG in Example 4-10 draws such a polygon. In Figure 4-11, is the middle section of the star considered to be inside or outside?

Example 4-10. Unfilled polygon with intersecting lines

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">

<polygon  points="48,16  16,96  96,48  0,48  80,96"
```

```
  style="stroke: black; fill: none;"/>

</svg>
```

Figure 4-11. Unfilled polygon with intersecting lines

SVG has two different rules for determining whether a point is inside a polygon or outside it. The `fill-rule` (which is part of presentation) has a value of either `nonzero` (the default) or `evenodd`. Depending on the rule you choose, you get a different effect. Example 4-11 uses the rules to fill two diagrams of the star. The result is shown in Figure 4-12.

Example 4-11. Effect of different fill-rules

http://oreillymedia.github.io/svg-essentials-examples/ch04/polygon-fill.html

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">

<polygon style="fill-rule: nonzero; fill: yellow; stroke: black;"
  points="48,16  16,96  96,48  0,48  80,96" />

<polygon style="fill-rule: evenodd;  fill: #00ff00; stroke: black;"
  points="148,16  116,96  196,48  100,48  180,96" />

</svg>
```

Figure 4-12. Effect of different fill-rules

Explanation of the Fill Rules

For the sake of completeness, here is how the `fill-rules` work, but don't worry—you don't need to know the details in order to use them. The `nonzero` rule determines whether a point is inside or outside a polygon by drawing a line from the point in question to infinity. It counts how many times that line crosses the polygon's lines, adding one if the polygon line is going right to left, and subtracting one if the polygon line is

going left to right. If the total comes out to zero, the point is outside the polygon. If the total is nonzero (hence the name), the point is inside the polygon.

The `evenodd` rule also draws a line from the point in question to infinity, but it simply counts how many times that line crosses your polygon's lines. If the total number of crossings is odd, then the point is inside; if even, then the point is outside.

The <polyline> Element

Finally, to round out our discussion of basic shapes, we'll return to straight lines. Sometimes you want a series of lines that does not make a closed shape. You can use multiple `<line>` elements, but if there are many lines, it might be easier to use the `<polyline>` element. It has the same `points` attributes as `<polygon>`, except that the shape is not closed. Example 4-12 draws the symbol for an electrical resistor. The result is in Figure 4-13.

Example 4-12. The polyline element

http://oreillymedia.github.io/svg-essentials-examples/ch04/polyline.html

```
<svg width="100px" height="50px" viewBox="0 0 100 50"
  xmlns="http://www.w3.org/2000/svg">

<polyline
  points="5 20, 20 20, 25 10, 35 30, 45 10,
    55 30, 65 10, 75 30, 80 20, 95 20"
  style="stroke: black; stroke-width: 3; fill: none;"/>
</svg>
```

Figure 4-13. Example of the polyline element

It's best to set the `fill` property to `none` when using `<polyline>`; otherwise, the SVG viewer attempts to fill the shape, sometimes with startling results like those in Figure 4-14.

Figure 4-14. Example of filled polyline

Line Caps and Joins

When drawing a `<line>` or `<polyline>`, you may specify the shape of the endpoints of the lines by setting the `stroke-linecap` style property to one of the values `butt`, `round`, or `square`. Example 4-13 uses these three values, with gray guide lines to show the actual endpoints of the lines. You can see in Figure 4-15 that `round` and `square` extend beyond the end coordinates; `butt`, the default, ends exactly at the specified endpoint.

Example 4-13. Values of the stroke-linecap property

http://oreillymedia.github.io/svg-essentials-examples/ch04/linecap.html

```
<line x1="10" y1="15" x2="50" y2="15"
  style="stroke: black; stroke-linecap: butt; stroke-width: 15;"/>

<line x1="10" y1="45" x2="50" y2="45"
  style="stroke: black; stroke-linecap: round; stroke-width: 15;"/>

<line x1="10" y1="75" x2="50" y2="75"
  style="stroke: black; stroke-linecap: square; stroke-width: 15;"/>

<!-- guide lines -->
<line x1="10" y1="0" x2="10" y2="100" style="stroke: #999;"/>
<line x1="50" y1="0" x2="50" y2="100" style="stroke: #999;"/>
```

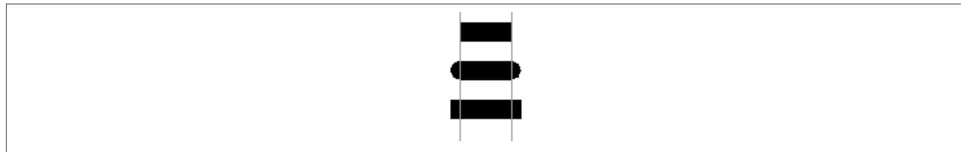


Figure 4-15. Values of the stroke-linecap attribute

You may specify the way lines connect at the corners of a shape with the `stroke-linejoin` style property, which may have the values `miter` (pointed), `round` (round—what did you expect?), or `bevel` (flat). Example 4-14 produces the result shown in Figure 4-16.

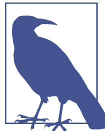
Example 4-14. Values of the stroke-linejoin attribute

http://oreillymedia.github.io/svg-essentials-examples/ch04/linejoin.html

```
<polyline
  style="stroke-linejoin: miter; stroke: black; stroke-width: 12;
  fill: none;"
  points="30 30, 45 15, 60 30"/>

<polyline
```

```
  style="stroke-linejoin: round; stroke: black; stroke-width: 12;
  fill: none;"
  points="90 30, 105 15, 120 30"/>

<polyline
  style="stroke-linejoin: bevel; stroke-width: 12; stroke: black;
  fill: none;"
  points="150 30, 165 15, 180 30"/>
```

Figure 4-16. Values of the stroke-linejoin attribute



If your lines meet at a sharp angle and have a mitered join, it's possible for the pointed part to extend far beyond the lines' thickness. You may set the ratio of the miter to the thickness of the lines being joined with the `stroke-miterlimit` style property; its default value is 4.

Basic Shapes Reference Summary

The following tables summarize the basic shapes and presentation styles in SVG.

Shape Elements

Table 4-1 summarizes the basic shapes available in SVG.

Table 4-1. Shape elements

Shape	Description
`<line x1="`*`start-x`*`" y1="`*`start-y`*`" x2="`*`end-x`*`" y2="`*`end-y`*`"/>`	Draws a line from the starting point at coordinates (*`start-x`*, *`start-y`*) to the ending point at coordinates (*`end-x`*, *`end-y`*).
`<rect x="`*`left-x`*`" y="`*`top-y`*`" width="`*`width`*`" height="`*`height`*`"/>`	Draws a rectangle whose upper-left corner is at (*`left-x`*, *`top-y`*) with the given *`width`* and *`height`*.
`<circle cx="`*`center-x`*`" cy="`*`center-y`*`" r="`*`radius`*`"/>`	Draws a circle with the given *`radius`*, centered at (*`center-x`*, *`center-y`*).
`<ellipse cx="`*`center-x`*`" cy="`*`center-y`*`" rx="`*`x-radius`*`" ry="`*`y-radius`*`"/>`	Draws an ellipse with the given *`x-radius`* and *`y-radius`* centered at (*`center-x`*, *`center-y`*).
`<polygon points="`*`points-list`*`"/>`	Draws an arbitrary closed polygon whose outline is described by the *`points-list`*. The points are specified as pairs of *x*- and *y*-coordinates. These are user coordinates only; you may not add a length unit specifier.

Shape	Description
`<polyline points="points-list"/>`	Draws an arbitrary series of connected lines as described by the *points-list*. The points are specified as pairs of *x*- and *y*-coordinates. These are user coordinates only; you may not add a length unit specifier.

When you specify a number for an attribute, it is presumed to be measured in user coordinates. In all but the last two elements of Table 4-1, you may also add a length unit specifier such as `mm`, `pt`, etc. to any number. For example:

```
<line x1="1cm" y1="30" x2="50" y2="10pt"/>
```

Specifying Colors

You may specify the color for filling or outlining a shape in one of the following ways:

- `none`, indicating that no outline is to be drawn or that the shape is not to be filled.
- A basic color name, which is one of `aqua`, `black`, `blue`, `fuchsia`, `gray`, `green`, `lime`, `maroon`, `navy`, `olive`, `purple`, `red`, `silver`, `teal`, `white`, or `yellow`.
- One of the extended color names from the SVG specifications.
- Six hexadecimal digits `#rrggbb`, each pair describing red, green, and blue values.
- Three hexadecimal digits `#rgb`, describing the red, green, and blue values. This is shorthand for the previous method; digits are replicated, so `#rgb` is equivalent to `#rrggbb`.
- `rgb(r, g, b)`, each value ranging from 0–255 or from 0% to 100%.
- `currentColor`, the computed (usually inherited) `color` property value for the element.
- One of the specifications from the CSS3 Color module (which may not be supported by all SVG implementations).

Stroke and Fill Characteristics

In order to see a line or the outline of a shape, you must specify the stroke characteristics, using the following attributes. A shape's outline is drawn after its interior is filled. All of these characteristics, summarized in Table 4-2, are presentation properties, and go in a `style` attribute.

Table 4-2. Stroke characteristics

Attribute	Values
`stroke`	The stroke color, as described in "Specifying Colors" on page 54. Default is `none`.
`stroke-width`	Width of stroke; may be given as user coordinates or with a length specifier. The stroke width is centered along the abstract grid lines. Default is `1`.
`stroke-opacity`	A number ranging from 0.0 to 1.0; 0.0 is entirely transparent; 1.0 is entirely opaque (the default).
`stroke-dasharray`	A series of numbers that tell the length of dashes and gaps with which a line is to be drawn. These numbers are in user coordinates only. The default value is `none`.
`stroke-linecap`	Shape of the ends of a line; has one of the values `butt` (the default), `round`, or `square`.
`stroke-linejoin`	The shape of the corners of a polygon or series of lines; has one of the values `miter` (pointed; the default), `round`, or `bevel` (flat).
`stroke-miterlimit`	Maximum ratio of length of the miter point to the width of the lines being drawn; the default value is 4.

You can control the way in which the interior of a shape is to be filled by using one of the fill attributes shown in Table 4-3. A shape is filled before its outline is drawn.

Table 4-3. Fill characteristics

Attribute	Values
`fill`	The fill color, as described in "Specifying Colors" on page 54. The default is `black`.
`fill-opacity`	A number ranging from 0.0 to 1.0; 0.0 is entirely transparent; 1.0 (the default) is entirely opaque.
`fill-rule`	This attribute can have the values `nonzero` (the default) or `evenodd`, which apply different rules for determining whether a point is inside or outside a shape. These rules generate different effects only when a shape has intersecting lines or "holes" in it. Details are in "Filling Polygons That Have Intersecting Lines" on page 49 earlier in this chapter.

This is only a small sample of the style properties that can apply to SVG elements; Table B-1, in Appendix B, has a complete list.

CHAPTER 5

Document Structure

We've casually mentioned that SVG lets you separate a document's structure from its presentation. In this chapter, we're going to compare and contrast the two, discuss the presentational aspects of a document in more detail, and then show some of the SVG elements you can use to make your document's structure clearer, more readable, and easier to maintain.

Structure and Presentation

As mentioned in Chapter 1, in "Basic Shapes" on page 6, one of XML's goals is to provide a way to structure data and separate this structure from its visual presentation. Consider the drawing of the cat from that chapter; you recognize it as a cat because of its structure —the position and size of the geometric shapes that make up the drawing. If we were to make structural changes, such as shortening the whiskers, rounding the nose, and making the ears longer and rounding their ends, the drawing would become one of a rabbit, no matter what the surface presentation might be. The structure, therefore, tells you what a graphic *is*.

This is not to say that information about visual style isn't important; had we drawn the cat with thick purple lines and a gray interior, it would have been recognizable as a cat, but its appearance would have been far less pleasing. These differences are shown in Figure 5-1. XML encourages you to separate structure and presentation; unfortunately, many discussions of XML emphasize structure at the expense of presentation. We'll right this wrong by going into detail about how you specify presentation in SVG.

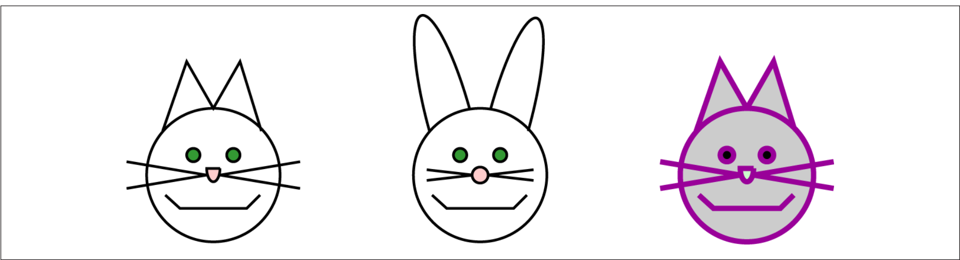


Figure 5-1. Structure versus presentation

Using Styles with SVG

SVG lets you specify presentational aspects of a graphic in four ways: with inline styles, internal stylesheets, external stylesheets, and presentation attributes. Let's examine each of these in turn.

Inline Styles

Example 5-1 uses inline styles. This is exactly the way we've been using presentation information so far; we set the value of the `style` attribute to a series of visual properties and their values as described in Appendix B, in "Anatomy of a Style" on page 299.

Example 5-1. Use of inline styles

```
<circle cx="20" cy="20" r="10"
    style="stroke: black; stroke-width: 1.5; fill: blue;
           fill-opacity: 0.6"/>
```

Internal Stylesheets

You don't need to place your styles inside each SVG element; you can create an internal stylesheet to collect commonly used styles, which you can apply to all occurrences of a particular element, or use named classes to apply styles to specific elements. Example 5-2 sets up an internal stylesheet that will draw all circles in a blue double-thick dashed line with a light yellow interior. The stylesheet is within a `<defs>` element, which we will discuss later in this chapter.

The example then draws several circles. The circles in the second row of Figure 5-2 have inline styles that override the specification in the internal stylesheet.

Example 5-2. Use of internal stylesheet

http://oreillymedia.github.io/svg-essentials-examples/ch05/internal-stylesheets.html

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
<defs>
```

```
<style type="text/css"><![CDATA[
    circle {
        fill: #ffc;
        stroke: blue;
        stroke-width: 2;
        stroke-dasharray: 5 3
    }
]]></style>
</defs>

<circle cx="20" cy="20" r="10"/>
<circle cx="60" cy="20" r="15"/>
<circle cx="20" cy="60" r="10" style="fill: #cfc"/>
<circle cx="60" cy="60" r="15"
  style="stroke-width: 1; stroke-dasharray: none;"/>
</svg>
```

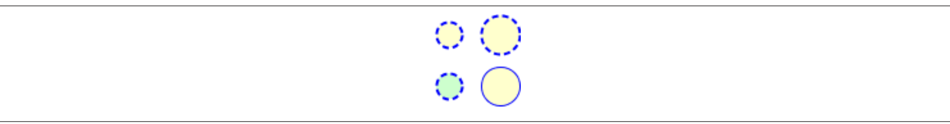


Figure 5-2. Internal stylesheet with SVG

External Stylesheets

If you want to apply a set of styles to multiple SVG documents, you could copy and paste the internal stylesheet into each of them. This, of course, is impractical for a large volume of documents if you ever need to make a global change to all the documents. Instead, you should take all the information between the beginning and ending `<style>` tags (excluding the `<![CDATA[` and `]]>`) and save it in an external file, which becomes an external stylesheet. Example 5-3 shows an external stylesheet that has been saved in a file named *ext_style.css*. This stylesheet uses a variety of selectors, including `*`, which sets a default for all elements that don't have any other style, and it, together with the SVG, produces Figure 5-3.

Example 5-3. External stylesheet

```
* { fill:none; stroke: black; } /* default for all elements */

rect { stroke-dasharray: 7 3; }

circle.yellow { fill: yellow; }

.thick { stroke-width: 5; }

.semiblue { fill:blue; fill-opacity: 0.5; }
```

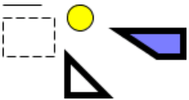


Figure 5-3. External stylesheet with SVG

Example 5-4 shows a complete SVG document (including `<?xml …?>`, `<?xml-stylesheet …?>`, and the `<!DOCTYPE>`) that references the external stylesheet.

Example 5-4. SVG file that references an external stylesheet

```
<?xml version="1.0"?>
<?xml-stylesheet href="ext_style.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 xmlns="http://www.w3.org/2000/svg"
  width="200px" height="200px" viewBox="0 0 200 200">

<line x1="10" y1="10" x2="40" y2="10"/>
<rect x="10" y="20" width="40" height="30"/>
<circle class="yellow" cx="70" cy="20" r="10"/>
<polygon class="thick" points="60 50, 60 80,  90 80"/>
<polygon class="thick semiblue"
  points="100 30, 150 30, 150 50, 130 50"/>
</svg>
```

Inline styles will almost always render more quickly than styles in an internal or external stylesheet; stylesheets and classes add rendering time due to lookup and parsing. However, stylesheets are easier to maintain, and smaller file size and caching can result in faster file-loading time.

Presentation Attributes

Although the overwhelming majority of your SVG documents will use styles for presentation information, SVG does permit you to specify this information in the form of presentation attributes. Instead of saying

```
<circle cx="10" cy="10" r="5"
    style="fill: red; stroke:black; stroke-width: 2;"/>
```

you may write each of the properties as an attribute:

```
<circle cx="10" cy="10" r="5"
    fill="red" stroke="black" stroke-width="2"/>
```

If you are thinking that this is mixing structure and presentation, you are right. Presentation attributes do come in handy, though, when you are creating SVG documents by converting an XML data source to SVG, as you will see in Chapter 15. In these cases,

it can be easier to create individual attributes for each presentation property than to create the contents of a single `style` attribute. You may also need to use presentation attributes if the environment in which you will be placing your SVG cannot support stylesheets.

Presentation attributes are at the very bottom of the priority list. Any style specification coming from an inline, internal, or external stylesheet will override a presentation attribute, although presentation attributes override inherited styles. In the following SVG document, the circle will be filled in red, not green:

```
<svg width="200" height="200"
  xmlns="http://www.w3.org/2000/svg">
  <defs>
    <style type="text/css"><![CDATA[
      circle { fill: red; }
    ]]></style>
  </defs>
  <circle cx="20" cy="20" r="15" fill="green"/>
</svg>
```

Again, we emphasize that using `style` attributes or stylesheets should always be your first choice. Stylesheets, in particular, let you apply a complex series of fill and stroke characteristics to all occurrences of certain elements within a document without having to duplicate the information into each element, as presentation attributes would require. The power and flexibility of stylesheets allow you to make significant changes in the look and feel of multiple documents with a minimum of effort.

Grouping and Referencing Objects

While it is certainly possible to define any drawing as an undifferentiated list of shapes and lines, most nonabstract art consists of groups of shapes and lines that form recognizable named objects. SVG has elements that let you do this sort of grouping to make your documents more structured and understandable.

The <g> Element

The `<g>` element gathers all of its child elements as a group and often has an `id` attribute to give that group a unique name. Each group may also have its own `<title>` and `<desc>` to identify it for text-based XML applications or to aid in accessibility for visually impaired users. Many SVG rendering agents will display a pop-up tooltip with the content of a `<title>` element when you hover over or tap any graphics within that group. Screen readers will read the contents of `<title>` and `<desc>` elements.

In addition to the conceptual clarity that comes from the ability to group and document objects, the `<g>` element also provides notational convenience. Any styles you specify in the starting `<g>` tag will apply to all the child elements in the group. In Example 5-5,

this saves us from having to duplicate the `style="fill:none; stroke:black;"` on every element shown in Figure 5-4. It is also possible to nest groups within one another, although you won't see any examples of this until Chapter 6.

The `<g>` element is analogous to the *Group Objects* function in programs such as Adobe Illustrator. It also serves a similar function to the concept of *layers* in such programs; a layer is also a grouping of related objects.

Example 5-5. Simple use of the g element

```
<svg width="240px" height="240px" viewBox="0 0 240 240"
  xmlns="http://www.w3.org/2000/svg">
<title>Grouped Drawing</title>
<desc>Stick-figure drawings of a house and people</desc>

<g id="house" style="fill: none; stroke: black;">
  <desc>House with door</desc>
  <rect x="6" y="50" width="60" height="60"/>
  <polyline points="6 50, 36 9, 66 50"/>
  <polyline points="36 110, 36 80, 50 80, 50 110"/>
</g>

<g id="man" style="fill: none; stroke: black;">
  <desc>Male human</desc>
  <circle cx="85" cy="56" r="10"/>
  <line x1="85" y1="66" x2="85" y2="80"/>
  <polyline points="76 104, 85 80, 94 104" />
  <polyline points="76 70, 85 76, 94 70" />
</g>

<g id="woman" style="fill: none; stroke: black;">
  <desc>Female human</desc>
  <circle cx="110" cy="56" r="10"/>
  <polyline points="110 66, 110 80, 100 90, 120 90, 110 80"/>
  <line x1="104" y1="104" x2="108" y2="90"/>
  <line x1="112" y1="90" x2="116" y2="104"/>
  <polyline points="101 70, 110 76, 119 70" />
</g>
</svg>
```

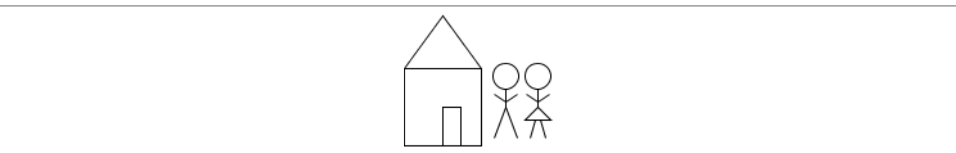


Figure 5-4. Grouped stick-figure drawing

The <use> Element

Complex graphics often have repeated elements. For example, a product brochure may have the company logo at the upper left and lower right of each page. If you were drawing the brochure with a graphic design program, you'd draw the logo once, group all its elements together, and then copy and paste them to the other location. The SVG `<use>` element gives you an analogous copy-and-paste ability with a group you've defined with `<g>` or any individual graphic element (such as a complex polygon shape that you want to define only once).

Once you have defined a group of graphic objects, you can display them again with the `<use>` tag. To specify the group you wish to reuse, give its URI in an `xlink:href` attribute, and specify the `x` and `y` location where the group's (0,0) point should be moved to. (We will see another way to achieve this effect in Chapter 6, in "The translate Transformation" on page 69.) So, to create another house and set of people, as shown in Figure 5-5, you'd put these lines just before the closing `</svg>` tag:

```
<use xlink:href="#house" x="70" y="100"/>
<use xlink:href="#woman" x="-80" y="100"/>
<use xlink:href="#man" x="-30" y="100"/>
```

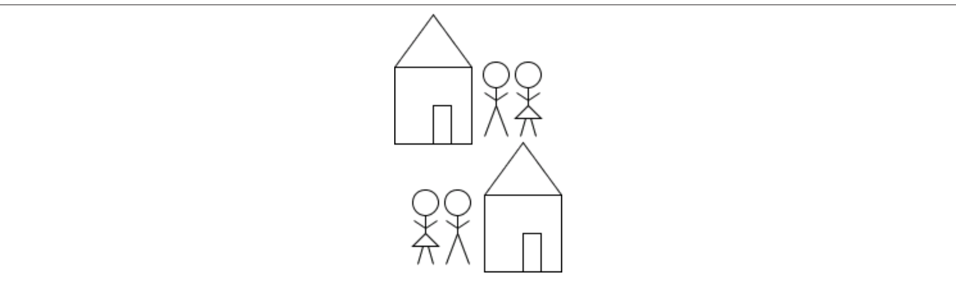


Figure 5-5. Reuse of grouped stick figures

The <defs> Element

You may have noticed some drawbacks with the preceding example:

- The math for deciding where to place the reused man and woman requires you to know the positions of the originals and use that as your base, rather than using a simple number like zero.
- The fill and stroke color for the house were established by the original, and can't be overriden by `<use>`. This means you can't make a row of multicolored houses.
- The document draws all three groups: the woman, the man, and the house. You can't "store them away" and draw only a set of houses or only a set of people.

The <defs> (definitions) element solves these problems. By putting the grouped objects between the beginning and ending <defs> tags, you instruct SVG to define them without displaying them. The SVG specification, in fact, recommends that you put all objects you wish to reuse within a <defs> element so that SVG viewers working in a streaming environment can process data more efficiently. In Example 5-6, the house, man, and woman are defined with their upper-left corner at (0,0), and the house is not given any fill color. Because the groups will be within the <defs> element, they will not be drawn on the screen right away and will serve as a "template" for future use. We have also constructed another group named `couple`, which, in turn, <use>s the `man` and `woman` groups. (Note that the bottom half of Figure 5-6 can't use `couple`, as it uses the figures in a different arrangement.)

Example 5-6. The defs element

http://oreillymedia.github.io/svg-essentials-examples/ch05/defs-example.html

```
<svg width="240px" height="240px" viewBox="0 0 240 240"
  xmlns="http://www.w3.org/2000/svg">
<title>Grouped Drawing</title>
<desc>Stick-figure drawings of a house and people</desc>

<defs>
<g id="house" style="stroke: black;">
  <desc>House with door</desc>
  <rect x="0" y="41" width="60" height="60"/>
  <polyline points="0 41, 30 0, 60 41"/>
  <polyline points="30 101, 30 71, 44 71, 44 101"/>
</g>

<g id="man" style="fill: none; stroke: black;">
  <desc>Male stick figure</desc>
  <circle cx="10" cy="10" r="10"/>
  <line x1="10" y1="20" x2="10" y2="44"/>
  <polyline points="1 58, 10 44, 19 58"/>
  <polyline points="1 24, 10 30, 19 24"/>
</g>

<g id="woman" style="fill: none; stroke: black;">
  <desc>Female stick figure</desc>
  <circle cx="10" cy="10" r="10"/>
  <polyline points="10 20, 10 34, 0 44, 20 44, 10 34"/>
  <line x1="4" y1="58" x2="8" y2="44"/>
  <line x1="12" y1="44" x2="16" y2="58"/>
  <polyline points="1 24, 10 30, 19 24" />
</g>

<g id="couple">
  <desc>Male and female stick figures</desc>
  <use xlink:href="#man" x="0" y="0"/>
  <use xlink:href="#woman" x="25" y="0"/>
```

```
</g>
</defs>

<!-- make use of the defined groups -->
<use xlink:href="#house" x="0" y="0" style="fill: #cfc;"/>
<use xlink:href="#couple" x="70" y="40"/>

<use xlink:href="#house" x="120" y="0" style="fill: #99f;"/>
<use xlink:href="#couple" x="190" y="40"/>

<use xlink:href="#woman" x="0" y="145"/>
<use xlink:href="#man" x="25" y="145"/>
<use xlink:href="#house" x="65" y="105" style="fill: #c00;"/>
</svg>
```

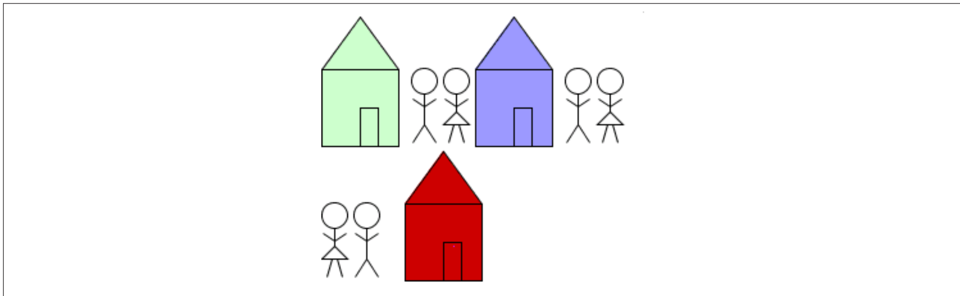


Figure 5-6. Result of using groups within defs

The `<use>` element is not restricted to using objects from the same file in which it occurs; the `xlink:href` attribute may specify any valid file or URI. This makes it possible to collect a set of common elements in one SVG file and use them selectively from other files. For example, you could create a file named *identity.svg* that contains all of the identity graphics your organization uses:

```
<g id="company_mascot">
   <!-- drawing of company mascot -->
</g>

<g id="company_logo" style="stroke: none;">
   <polygon points="0 20, 20 0, 40 20, 20 40"
      style="fill: #696;"/>
   <rect x="7" y="7" width="26" height="26"
      style="fill: #c9c;"/>
</g>

<g id="partner_logo">
    <!-- drawing of company partner's logo -->
</g>
```

and then refer to it with the following:

```
<use xlink:href="identity.svg#company_logo" x="200" y="200"/>
```

External references may not be supported in all SVG viewers, particularly web browsers, for security reasons. Some browsers (notably Internet Explorer) currently do not support external file references at all. Others only allow `<use>` elements to reference files on the same web domain or on a web server that is specifically configured to allow cross-origin use.

The <symbol> Element

The `<symbol>` element provides another way of grouping elements. Unlike the `<g>` element, a `<symbol>` is never displayed, so you don't have to enclose it in a `<defs>` specification. However, it is customary to do so, because a symbol really is something you're defining for later use. Symbols can also specify `viewBox` and `preserveAspectRatio` attributes, allowing a symbol to fit into a viewport established by adding `width` and `height` attributes to the `<use>` element. Example 5-7 shows that the `width` and `height` are ignored for a simple group (the top two octagons), but are used when displaying a symbol. The edges of the lower-right octagon in Figure 5-7 are cut off because the `preserveAspectRatio` has been set to `slice`. The `<rect>` elements are included to show the coordinates of each `<use>`.

Example 5-7. Symbols versus groups

http://oreillymedia.github.io/svg-essentials-examples/ch05/symbol.html

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
<title>Symbols vs. groups</title>
<desc>Use</desc>

<defs>
  <g id="octagon" style="stroke: black;">
    <desc>Octagon as group</desc>
    <polygon points="
      36 25, 25 36, 11 36, 0 25,
      0 11, 11 0, 25 0, 36 11"/>
  </g>

  <symbol id="sym-octagon" style="stroke: black;"
    preserveAspectRatio="xMidYMid slice" viewBox="0 0 40 40">
    <desc>Octagon as symbol</desc>
    <polygon points="
      36 25, 25 36, 11 36, 0 25,
      0 11, 11 0, 25 0, 36 11"/>
  </symbol>
</defs>
```

```
<g style="fill:none; stroke:gray">
  <rect x="40" y="40" width="30" height="30"/>
  <rect x="80" y="40" width="40" height="60"/>
  <rect x="40" y="110" width="30" height="30"/>
  <rect x="80" y="110" width="40" height="60"/>
</g>
<use xlink:href="#octagon" x="40" y="40" width="30" height="30"
  style="fill: #c00;"/>
<use xlink:href="#octagon" x="80" y="40" width="40" height="60"
  style="fill: #cc0;"/>
<use xlink:href="#sym-octagon" x="40" y="110" width="30" height="30"
  style="fill: #cfc;"/>
<use xlink:href="#sym-octagon" x="80" y="110" width="40" height="60"
  style="fill: #699;"/>
</svg>
```

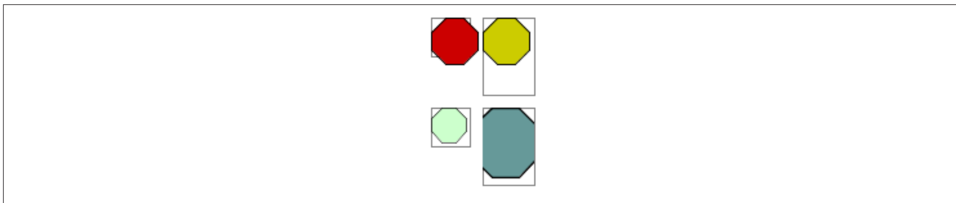


Figure 5-7. Groups versus symbols

The <image> Element

While `<use>` lets you reuse a portion of an SVG file, the `<image>` element includes an entire SVG or raster file. If you are including an SVG file, the `x`, `y`, `width`, and `height` attributes establish the viewport in which the referenced file will be drawn; if you're including a raster file, it will be scaled to fit the rectangle that the attributes specify. The SVG specs require viewers to support both JPEG and PNG raster files; viewers may support other files. For example, most web browsers will also support GIF. Example 5-8 shows how to include a JPEG image with SVG. The result is in Figure 5-8.

Example 5-8. Use of the image element

```
<svg width="310px" height="310px" viewBox="0 0 310 310"
  xmlns="http://www.w3.org/2000/svg">

<ellipse cx="154" cy="154" rx="150" ry="120" style="fill: #999999;"> ❶
<ellipse cx="152" cy="152" rx="150" ry="120" style="fill: #cceeff;"> ❷

<image xlink:href="kwanghwamun.jpg"  ❸
  x="72" y="92"      ❹
  width="160" height="120"/>      ❺

</svg>
```

❶ Create a gray ellipse to simulate a drop shadow.[1]

❷ Create the main blue ellipse. Because it occurs after the gray ellipse, it is displayed above that object.

❸ Specify the URI of the file to include.

❹ Specify the upper-left corner of the image.

❺ Specify the width and height to which the image should be scaled.

Figure 5-8. JPEG image included in an SVG file

An `<image>` element may have a `preserveAspectRatio` attribute to indicate what to do if the dimensions of the image file do not match the width and height of the element. The default value, `xMidYMid meet`, will scale the image to fit and center it in the rectangle you specify (see "Preserving Aspect Ratio" on page 32). If you are including an SVG file, you may add the keyword `defer` at the beginning of the `preserveAspectRatio` value (like `defer xMidYMin meet`); if the included image has a `preserveAspectRatio` attribute, it will be used instead.

1. In Chapter 11, we'll see another way to create a drop shadow in "Creating a Drop Shadow" on page 156.

CHAPTER 6

Transforming the Coordinate System

Up to this point, all graphics have been displayed as is—drawn exactly where and how they are defined in their attributes. There will be times when you have a graphic you would like to rotate, scale, or move to a new location. To accomplish these tasks, you add the `transform` attribute to the appropriate SVG elements. This chapter examines the details of these transformations.

The translate Transformation

In Chapter 5, you saw that you can use `x` and `y` attributes with the `<use>` element to place a group of graphic objects at a specific place. Look at the SVG in Example 6-1, which defines a square and draws it at the upper-left corner of the grid, then redraws it with the upper-left corner at coordinates (50,50). The dotted lines in Figure 6-1 aren't part of the SVG, but serve to show the part of the canvas we're interested in.

Example 6-1. Moving a graphic with use

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
    <g id="square">
        <rect x="0" y="0" width="20" height="20"
            style="fill: black; stroke-width: 2;"/>
    </g>
    <use xlink:href="#square" x="50" y="50"/>
</svg>
```

As it turns out, the `x` and `y` values are really shorthand for one form of the more general and more powerful `transform` attribute. Specifically, the `x` and `y` values are equivalent to an attribute like `transform="translate(`*`x-value`*`,` *`y-value`*`)"`, where `translate` is a fancy technical term for *move*. The *x-value* and *y-value* are measured in the current user coordinate system. Let's use `transform` to get the same effect of making a second square with its upper-left corner at (50,50). Example 6-2 lists the SVG.

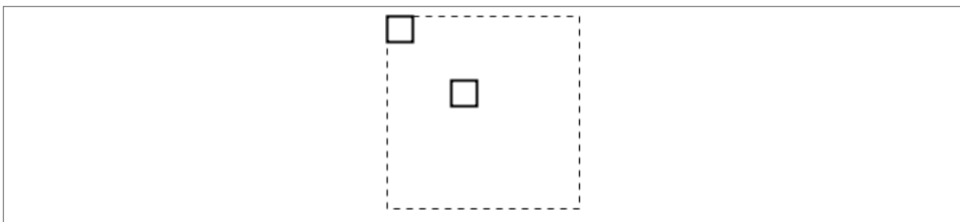


Figure 6-1. Result of moving with use

Example 6-2. Moving the coordinate system with translation

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
    <g id="square">
        <rect x="0" y="0" width="20" height="20"
            style="fill: none; stroke:black; stroke-width: 2;"/>
    </g>
    <use xlink:href="#square" transform="translate(50,50)"/>
</svg>
```

The resulting display will look exactly like that in Figure 6-1. You might think this was accomplished by moving the square to a different place on the grid, as shown conceptually in Figure 6-2, but you would be wrong.

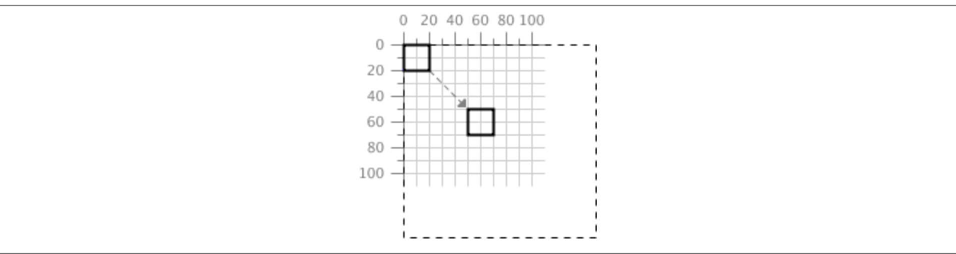


Figure 6-2. How moving appears to work (but really doesn't)

What is really going on behind the scenes is an entirely different story. Rather than moving the square, the `translate` specification picks up the *entire grid* and moves it to a new location on the canvas. As far as the square is concerned, it's still being drawn with its upper-left corner at (0,0), as depicted in Figure 6-3.

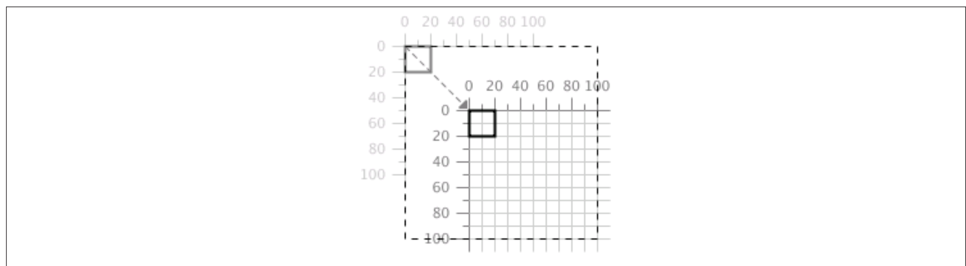


Figure 6-3. How moving with translate really works

The online example allows you to experiment with different coordinates:

http://oreillymedia.github.io/svg-essentials-examples/ch06/translate.html

A transformation *never* changes a graphic object's grid coordinates; rather, it changes the position of the grid on the canvas.

At first glance, using `translate` seems as ridiculous and inefficient as moving your couch further away from the outside wall of the house by moving the entire living room, walls and all, to a new position. Indeed, if translation were the only transformation available, moving the entire coordinate system would be wasteful. However, you will soon see other transformations and combinations of a sequence of transformations that are more mathematically and conceptually convenient if they apply to the entire coordinate system.

The scale Transformation

It is possible to make an object appear larger or smaller than the size at which it was defined by scaling the coordinate system. Such a transformation is specified as follows:

`transform="scale(`*`value`*`)"`
: Multiplies all *x*- and *y*-coordinates by the given *`value`*.

`transform="scale(`*`x-value, y-value`*`)"`
: Multiplies all *x*-coordinates by the given *`x-value`* and all *y*-coordinates by the given *`y-value`*.

Example 6-3 is an example of the first kind of scaling transformation, which uniformly doubles the scale of both axes. Once again, the dotted lines in Figure 6-4 aren't in the SVG; they simply show the area of the canvas we're interested in. Note that the square's upper-left corner is at (10,10).

Example 6-3. Uniformly scaling a graphic

http://oreillymedia.github.io/svg-essentials-examples/ch06/scale.html

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
    <g id="square">
        <rect x="10" y="10" width="20" height="20"
            style="fill: none; stroke: black;"/>
    </g>
    <use xlink:href="#square" transform="scale(2)"/>
</svg>
```

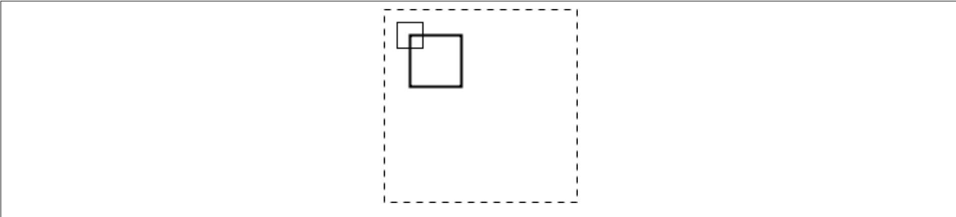


Figure 6-4. Result of using scale transformation

You might be thinking, "Wait a minute—I can understand why the square got larger. But I didn't ask for a `translate`, so why is the square in a different place?" Everything becomes clear when you look at Figure 6-5 to see what has actually occurred. The grid hasn't moved; the (0,0) point of the coordinate system is still in the same place, but each user coordinate is now twice as large as it used to be. You can see from the grid lines that the upper-left corner of the rectangle is still at (10,10) on the new, larger grid, because objects *never* move. This also explains why the outline of the larger square is thicker. The `stroke-width` is still one user unit, but that unit has now become twice as large, so the stroke thickens.

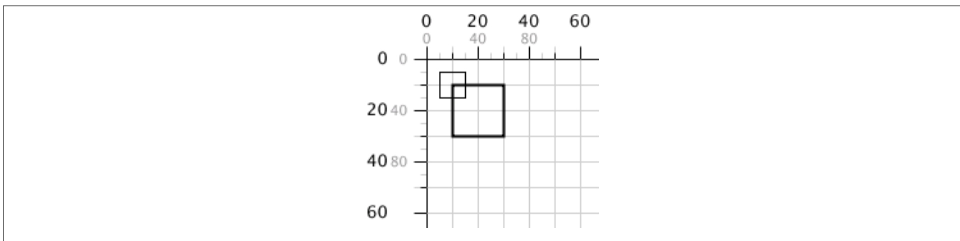


Figure 6-5. How the scale transformation works

A scaling transformation *never* changes a graphic object's grid coordinates or its stroke width; rather, it changes the size of the coordinate system (grid) with respect to the canvas.

It is possible to specify a different scale factor for the *x*-axis and *y*-axis of the coordinate system by using the second form of the `scale` transformation. Example 6-4 draws the square with the *x*-axis scaled by a factor of three and the *y*-axis scaled by a factor of one and a half. As you can see in Figure 6-6, the one-unit stroke width is also nonuniformly scaled.

Example 6-4. Nonuniform scaling of a graphic

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
  <g id="square">
    <rect x="10" y="10" width="20" height="20"
      style="fill: none; stroke: black;"/>
  </g>
  <use xlink:href="#square" transform="scale(3, 1.5)"/>
</svg>
```

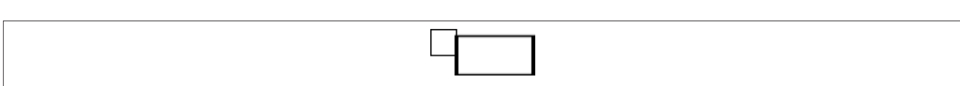


Figure 6-6. Result of using nonuniform scale transformation

To this point, the examples have applied the `transform` attribute to only the `<use>` element. You can apply a transformation to a series of elements by grouping them and transforming the group:

```
<g id="group1" transform="translate(3, 5)">
  <line x1="10" y1="10" x1="30" y2="30"/>
  <circle cx="20" cy="20" r="10"/>
</g>
```

You may also apply a transformation to a single object or basic shape. For example, here is a rectangle whose coordinate system is scaled by a factor of three:

```
<rect x="15" y="20" width="10" height="5"
  transform="scale(3)"
  style="fill: none; stroke: black;"/>
```

It's fairly clear that the width and height of the scaled rectangle should be three times as large as the unscaled rectangle. However, you may wonder if the *x*- and *y*-coordinates are evaluated before or after the rectangle is scaled. The answer is that SVG applies transformations to the coordinate system before it evaluates any of the shape's

coordinates. Example 6-5 is the SVG for the scaled rectangle, shown in Figure 6-7 with grid lines that are drawn in the unscaled coordinate system.

Example 6-5. Transforming a single graphic

```
<!-- grid guide lines in non-scaled coordinate system -->
<line x1="0" y1="0" x2="100" y2="0" style="stroke: black;"/>
<line x1="0" y1="0" x2="0" y2="100" style="stroke: black;"/>
<line x1="45" y1="0" x2="45" y2="100" style="stroke: gray;"/>
<line x1="0" y1="60" x2="100" y2="60" style="stroke: gray;"/>

<!-- rectangle to be transformed -->
<rect x="15" y="20" width="10" height="5"
    transform="scale(3)"
    style="fill: none; stroke: black;"/>
```

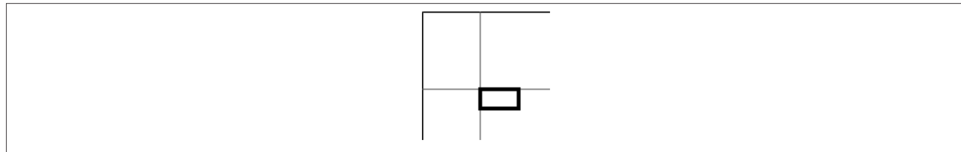
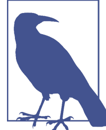


Figure 6-7. Result of transforming a single graphic



The effect of applying a transformation to a shape is the same as if the shape were enclosed in a transformed group. In the preceding example, the scaled rectangle is equivalent to this SVG:

```
<g transform="scale(3)">
   <rect x="15" y="20" width="10" height="5"
     style="fill: none; stroke: black;"/>
</g>
```

Sequences of Transformations

It is possible to do more than one transformation on a graphic object. You just put the transformations, optionally separated by whitespace or a comma, in the value of the `transform` attribute. Here is a rectangle that undergoes two transformations, a translation followed by a scaling. (The axes are drawn to show that the rectangle has, indeed, moved.)

```
<!-- draw axes -->
<line x1="0" y1="0" x2="0" y2="100" style="stroke: gray;"/>
<line x1="0" y1="0" x2="100" y2="0" style="stroke: gray;"/>

<rect x="10" y="10" height="15" width="20"
  transform="translate(30, 20) scale(2)"
  style="fill: gray;"/>
```

This is the equivalent of the following sequence of nested groups, and both will produce what you see in Figure 6-8:

```
<g transform="translate(30, 20)">
  <g transform="scale(2)">
  <rect x="10" y="10" height="15" width="20"
    style="fill: gray;"/>
  </g>
</g>
```

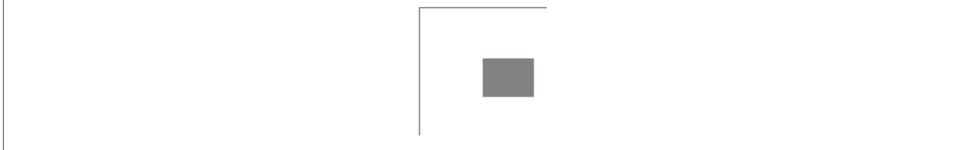


Figure 6-8. Result of translate followed by scale

Figure 6-9 shows what is happening at each stage of the transformation.

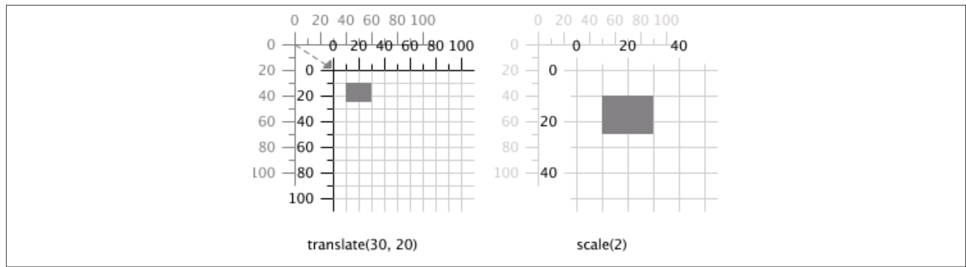
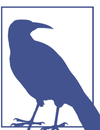


Figure 6-9. How translate followed by scale works



The order in which you do a sequence of transformations affects the result. In general, transformation A followed by transformation B will not give the same result as transformation B followed by transformation A.

Example 6-6 draws the same rectangle as in the previous example, in gray. Then it draws the rectangle again in black, but does the `scale` before the `translate`. As you can see from the result in Figure 6-10, the rectangles end up in very different places on the canvas.

Example 6-6. Sequence of transformations—scale followed by translate

```
<!-- draw axes -->
<line x1="0" y1="0" x2="0" y2="100" style="stroke: gray;"/>
<line x1="0" y1="0" x2="100" y2="0" style="stroke: gray;"/>
```

```
<rect x="10" y="10" width="20" height="15"
    transform="translate(30, 20) scale(2)" style="fill: gray;"/>

<rect x="10" y="10" width="20" height="15"
    transform="scale(2) translate(30, 20)"
    style="fill: black;"/>
```

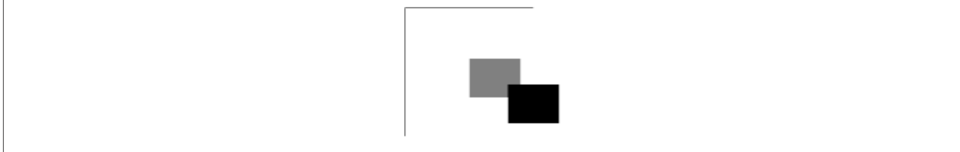


Figure 6-10. Result of scale followed by translate

The reason the black rectangle ends up farther away from the origin is that the scaling is applied first, so the translate of 20 units in the *x*-direction and 10 units in the *y*-direction is done with units that are now twice as large, as shown in Figure 6-11.

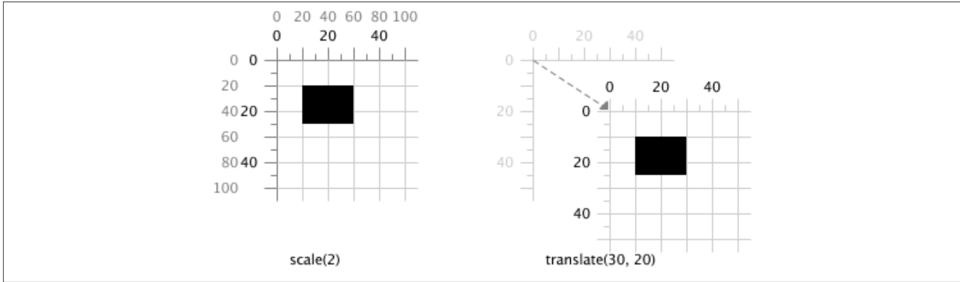


Figure 6-11. How scale followed by translate works

In the online example, you can experiment with any sequence of transformations, and compare the transformed rectangle against the original:

http://oreillymedia.github.io/svg-essentials-examples/ch06/sequence.html

Technique: Converting from Cartesian Coordinates

If you are transferring data from other systems to SVG, you may have to deal with vector drawings that use Cartesian coordinates (the ones you learned about in high school algebra) to represent data. In this system, the (0,0) point is at the lower left of the canvas, and *y*-coordinates increase as you move upward. Figure 6-12 shows the coordinates of a trapezoid drawn with Cartesian coordinates.

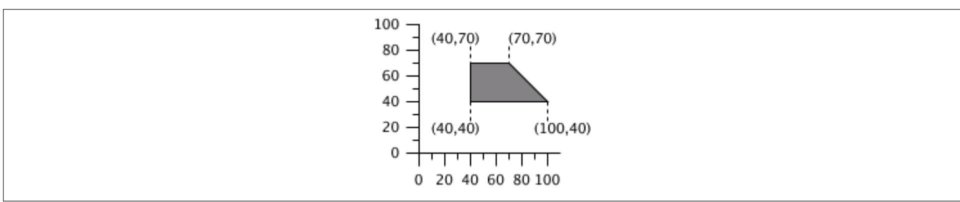


Figure 6-12. Trapezoid drawn with Cartesian coordinates

The *y*-axis is "upside-down" relative to the SVG default, so the coordinates need to be recalculated. Rather than do it by hand, you can use a sequence of transformations to have SVG do all the work for you. First, translate the picture into SVG, with the coordinates exactly as shown in Example 6-7. (The example also includes the axes as a guide.) To nobody's surprise, the picture will come out upside-down. Note that the image in Figure 6-13 is *not* left-to-right reversed, because the *x*-axis points in the same direction in both Cartesian coordinates and the default SVG coordinate system.

Example 6-7. Direct use of Cartesian coordinates

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
  <!-- axes -->
  <line x1="0" y1="0" x2="100" y2="0" style="stroke: black;"/>
  <line x1="0" y1="0" x2="0" y2="100" style="stroke: black;"/>

  <!-- trapezoid -->
  <polygon points="40 40, 100 40, 70 70, 40 70"
      style="fill: gray; stroke: black;"/>
</svg>
```

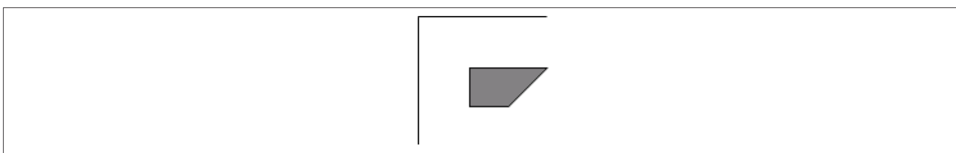
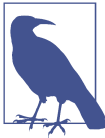


Figure 6-13. Result of using original Cartesian coordinates

To flip the image back right-side-up, you can take advantage of the fact that scaling a shape by a negative value reverses the order of coordinates. However, because the entire grid ends up flipped to the other side of the 0 coordinate, you also need to translate the shape back onto the visible part of the canvas. The conversion follows these steps:

1. Find the maximum *y*-coordinate in the original drawing. In this case, it turns out to be 100, the endpoint of the *y*-axis in the original.
2. Enclose the entire drawing in a `<g>` element.

3. Enter a translate that moves the coordinate system downward by the maximum y value: `transform="translate(0, max-y)"`.
4. The next transform will be to scale the *y*-axis by a factor of –1, flipping it upside-down: `transform="translate(0, max-y) scale(1, -1)"`.



You don't want to change the *x*-axis values, but you still need to specify an *x*-value for both the translate and scale functions. The do-nothing value for the translate is `0`, but the do-nothing value for the scale transform is `1`, because coordinates are *multiplied* by the scale factor. A `scale(0)` transform would collapse your shape to a single point (because every coordinate, multiplied by zero, would become zero).

Example 6-8 incorporates this transformation, producing a right-side-up trapezoid in Figure 6-14.

Example 6-8. Transformed Cartesian coordinates

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
  <g transform="translate(0,100) scale(1,-1)">
    <!-- axes -->
    <line x1="0" y1="0" x2="100" y2="0" style="stroke: black;"/>
    <line x1="0" y1="0" x2="0" y2="100" style="stroke: black;"/>

    <!-- trapezoid -->
    <polygon points="40 40, 100 40, 70 70, 40 70"
        style="fill: gray; stroke: black;"/>
  </g>
</svg>
```

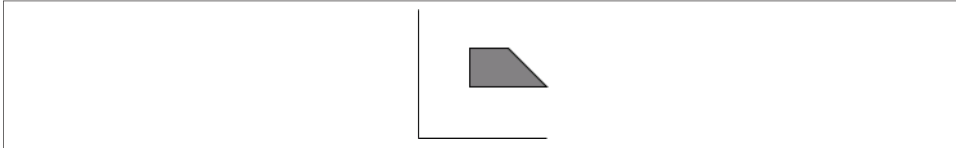


Figure 6-14. Transformed Cartesian coordinates

The rotate Transformation

It is also possible to rotate the coordinate system by a specified angle. In the default coordinate system, angle measure increases as you rotate clockwise, with a horizontal line having an angle of 0 degrees, as shown in Figure 6-15.

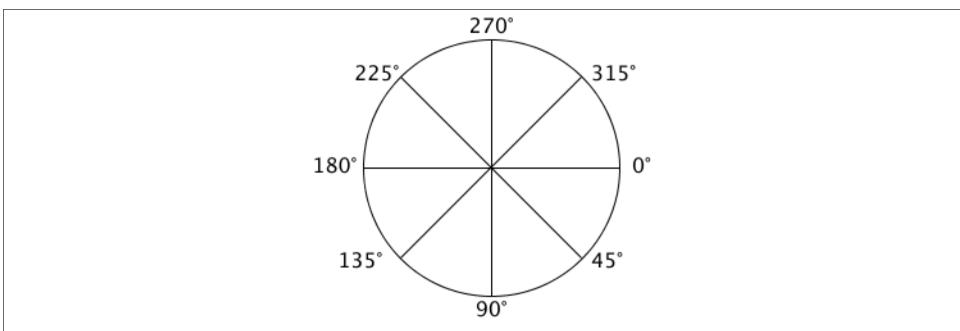


Figure 6-15. Default measurement of angles

Unless you specify otherwise, the *center of rotation* (a fancy term for the *pivot point*) is presumed to be (0,0). Example 6-9 shows a square drawn in gray, then drawn again in black after the coordinate system is rotated 45 degrees. The axes are also shown as a guide. Figure 6-16 shows the result. If you're surprised that the square has appeared to move, you shouldn't be. Remember, as shown in Figure 6-17, the entire coordinate system has been rotated.[1]

Example 6-9. Rotation around the origin

http://oreillymedia.github.io/svg-essentials-examples/ch06/rotate.html

```
<!-- axes -->
<polyline points="100 0, 0 0, 0 100" style="stroke: black; fill: none;"/>

<!-- normal and rotated square -->
<rect x="70" y="30" width="20" height="20" style="fill: gray;"/>
<rect x="70" y="30" width="20" height="20"
    transform="rotate(45)" style="fill: black;"/>
```

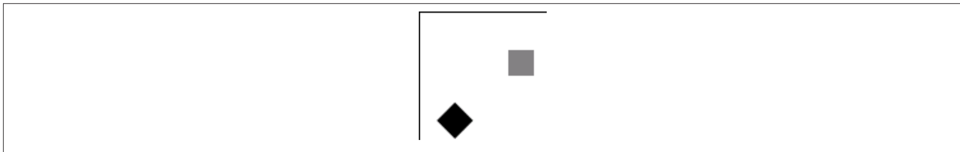


Figure 6-16. Result of rotation around the origin

1. All the figures in this chapter are static pictures. This one shows two squares (one rotated and one unrotated). To show an animation of a rotating square, use `<animateTransform>`, which we will discuss in Chapter 12, in "The <animateTransform> Element" on page 200.

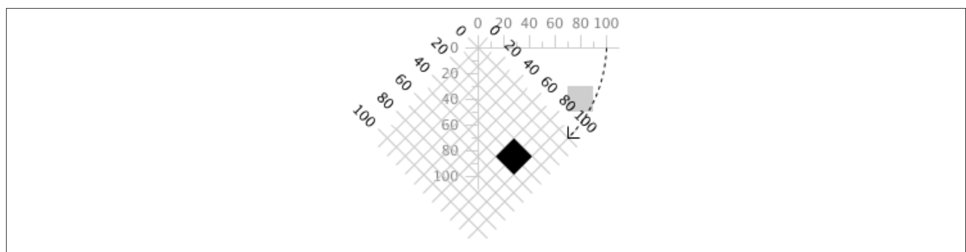


Figure 6-17. How rotation around the origin works

Most of the time, you will not want to rotate the entire coordinate system around the origin; you'll want to rotate a single object around a point other than the origin. You can do that via this series of transformations: `translate(centerX, centerY) rotate(angle) translate(-centerX, -centerY)`. SVG provides another version of `rotate` to make this common task easier. In this second form of the `rotate` transformation, you specify the angle and the center point around which you want to rotate:

```
rotate(angle, centerX, centerY)
```

This has the effect of temporarily establishing a new system of coordinates with the origin at the specified center *x* and *y* points, doing the rotation, and then re-establishing the original coordinates. Example 6-10 shows this form of `rotate` to create multiple copies of an arrow, shown in Figure 6-18.

Example 6-10. Rotation around a center point

http://oreillymedia.github.io/svg-essentials-examples/ch06/rotate-center.html

```
<!-- center of rotation -->
<circle cx="50" cy="50" r="3" style="fill: black;"/>

<!-- non-rotated arrow -->
<g id="arrow" style="stroke: black;">
    <line x1="60" y1="50" x2="90" y2="50"/>
    <polygon points="90 50, 85 45, 85 55"/>
</g>

<!-- rotated around center point -->
<use xlink:href="#arrow" transform="rotate(60, 50, 50)"/>
<use xlink:href="#arrow" transform="rotate(-90, 50, 50)"/>
<use xlink:href="#arrow" transform="rotate(-150, 50 50)"/>
```

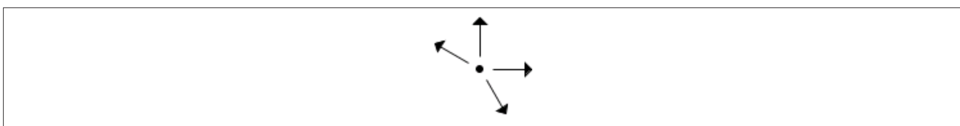


Figure 6-18. Result of rotation around a center point

Technique: Scaling Around a Center Point

While it's possible to rotate around a point other than the origin, there is no corresponding capability to `scale` around a point. You can, however, make concentric symbols with a simple series of transformations. To scale an object by a given factor around a center point, do this:

```
translate(-centerX*(factor-1), -centerY*(factor-1))
scale(factor)
```

You may also want to divide the `stroke-width` by the scaling factor so the outline stays the same width while the object becomes larger. Example 6-11 draws the set of concentric rectangles shown in Figure 6-19.[2]

Example 6-11. Scaling around a center point

```
<!-- center of scaling -->
<circle cx="50" cy="50" r="2" style="fill: black;"/>

<!-- non-scaled rectangle -->
<g id="box" style="stroke: black; fill: none;">
    <rect x="35" y="40" width="30" height="20"/>
</g>

<use xlink:href="#box" transform="translate(-50,-50) scale(2)"
    style="stroke-width: 0.5;"/>
<use xlink:href="#box" transform="translate(-75,-75) scale(2.5)"
    style="stroke-width: 0.4;"/>
<use xlink:href="#box" transform="translate(-100,-100) scale(3)"
    style="stroke-width: 0.33;"/>
```

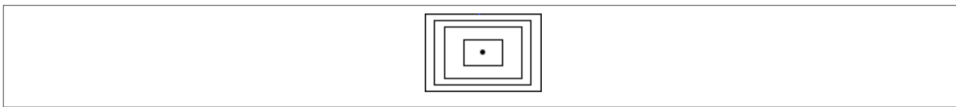


Figure 6-19. Result of scaling around a center point

The skewX and skewY Transformations

SVG also has two other transformations: `skewX` and `skewY`, which let you skew one of the axes. The general form is `skewX(`*`angle`*`)` and `skewY(`*`angle`*`)`. The `skewX` transformation "pushes" all *x*-coordinates by the specified angle, leaving *y*-coordinates

2. This is also a static picture, a "square bull's-eye." If you want to show an animation of an expanding square, you'll use `<animateTransform>`, which we will discuss in Chapter 12, in "The <animateTransform> Element" on page 200.

unchanged. skewY skews the *y*-coordinates, leaving *x*-coordinates unchanged, as shown in Figure 6-20, which is drawn with the code in Example 6-12.

Example 6-12. skewX and skewY

http://oreillymedia.github.io/svg-essentials-examples/ch06/skew.html

```
<!-- guide lines --> ❶
<g style="stroke: gray; stroke-dasharray: 4 4;">
    <line x1="0" y1="0" x2="200" y2="0"/>
    <line x1="20" y1="0" x2="20" y2="90"/>
    <line x1="120" y1="0" x2="120" y2="90"/>
</g>

<g transform="translate(20, 0)">      ❷
    <g transform="skewX(30)">     ❸
        <polyline points="50 0, 0 0, 0 50"       ❹
             style="fill: none; stroke: black; stroke-width: 2;"/>
        <text x="0" y="60">skewX</text>      ❺
    </g>
</g>

<g transform="translate(120, 0)">      ❻
    <g transform="skewY(30)">
        <polyline points="50 0, 0 0, 0 50"
            style="fill: none; stroke: black; stroke-width: 2;"/>
        <text x="0" y="60">skewY</text>
    </g>
</g>
```

❶ These dashed lines are drawn in the default coordinate system, before any transformation has occurred.

❷ This will move the entire skewed "package" to the desired location.

❸ Skew the *x*-coordinates 30 degrees. This transformation doesn't change the origin, which will still be at (0,0) in the new coordinate system.

❹ To make things easier, we draw the object at the origin.

❺ Text will be covered in more detail in Chapter 9.

❻ These elements are organized exactly like the preceding ones, except the *y*-coordinates are skewed.

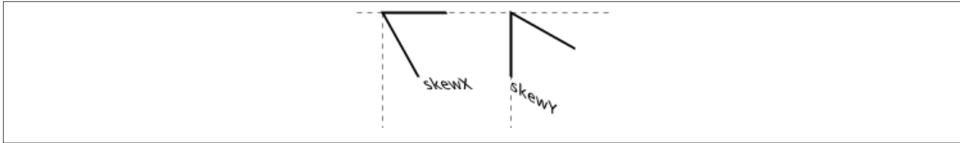


Figure 6-20. Result of skewX and skewY transformations

Notice that `skewX` leaves the horizontal lines horizontal, and `skewY` leaves the vertical lines untouched. Go figure.

Transformation Reference Summary

Table 6-1 gives a quick summary of the transformations available in SVG.

Table 6-1. SVG transformations

Transformation	Description
`translate(x, y)`	Moves the user coordinate system by the specified *x* and *y* amounts. Note: If you don't specify a *y* value, 0 is assumed.
`scale(xFactor, yFactor)`	Multiplies all user coordinates by the specified *xFactor* and *yFactor*. The factors may be fractional or negative.
`scale(factor)`	Same as `scale(factor, factor)`.
`rotate(angle)`	Rotates the user coordinate system by the specified *angle*. The center of rotation is the origin (0,0). In the default coordinate system, angle measure increases as you rotate clockwise, with a horizontal line having an angle of 0 degrees.
`rotate(angle, centerX, centerY)`	Rotates the user coordinate system by the specified *angle*. The center of rotation is specified by *centerX* and *centerY*.
`skewX(angle)`	Skews all *x*-coordinates by the specified *angle*. Visually, this makes vertical lines appear at an angle.
`skewY(angle)`	Skews all *y*-coordinates by the specified *angle*. Visually, this makes horizontal lines appear at an angle.
`matrix(a b c d e f)`	Specifies a transformation in the form of a transformation matrix of six values. See Appendix D.

CSS Transformations and SVG

As of this writing, CSS also has a working draft of a transformations module. Because it is a working draft, details may change and browser support may vary. If you are already using CSS transforms, there are some important differences from SVG:

- SVG 1.1 transforms are in user units or implicit degrees. CSS transforms use CSS length and angle units, although the specification would also allow implicit user units when applied to SVG elements.
- SVG 1.1 transforms are structural attributes, whereas CSS transforms can be specified in stylesheets. Stylesheet specifications override attribute values.
- In CSS, you cannot have space between the transform type and the opening parenthesis, and you must use commas to separate numerical values.

- CSS transforms include a separate property to specify the origin for rotation and scaling. In SVG, the rotation origin is part of the `rotate()` function, and you cannot specify an origin for scaling.
- CSS transforms include 3D effects.

CHAPTER 7
Paths

All of the basic shapes described in Chapter 4 are really shorthand forms for the more general `<path>` element. You are well advised to use these shortcuts; they help make your SVG more readable and more structured. The `<path>` element is more general; it draws the outline of any arbitrary shape by specifying a series of connected lines, arcs, and curves. This outline can be filled and drawn with a stroke, just as the basic shapes are. Additionally, these paths (as well as the shorthand basic shapes) may be used to define the outline of a clipping area or a transparency mask, as you will see in Chapter 10.

All of the data describing an outline is in the `<path>` element's `d` attribute (the `d` stands for *data*). The path data consists of one-letter commands, such as `M` for *moveto* or `L` for *lineto*, followed by the coordinate information for that particular command.

moveto, lineto, and closepath

Every path must begin with a *moveto* command.

The command letter is a capital `M` followed by an *x*- and *y*-coordinate, separated by commas or whitespace. This command sets the current location of the "pen" that's drawing the outline.

This is followed by one or more *lineto* commands, denoted by a capital `L`, also followed by *x*- and *y*- coordinates, and separated by commas or whitespace. Example 7-1 has three paths. The first draws a single line, the second draws a right angle, and the third draws two 30-degree angles. When you "pick up" the pen with another *moveto*, you are starting a new subpath. Notice that you can use either a comma or whitespace to separate the *x*- and *y*-coordinates, as shown in all three paths. The result is Figure 7-1.

Example 7-1. Using moveto and lineto

http://oreillymedia.github.io/svg-essentials-examples/ch07/moveto-lineto.html

```
<svg width="150px" height="150px" viewBox="0 0 150 150"
  xmlns="http://www.w3.org/2000/svg">
<g style="stroke: black; fill: none;">
  <!-- single line -->
  <path d="M 10 10 L 100 10"/>

  <!-- a right angle -->
  <path d="M 10, 20  L 100, 20  L  100,50"/>

  <!-- two 30-degree angles -->
  <path d="M 40 60 L 10, 60 L 40 42.68
    M 60, 60 L 90 60 L 60, 42.68"/>
</g>
</svg>
```

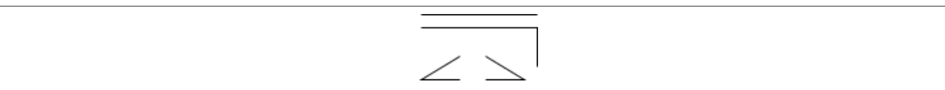
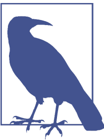


Figure 7-1. Result of using moveto and lineto

Examining the last path more closely, with commas replaced by whitespace:

Value	Action
`M 40 60`	Move pen to (40,60)
`L 10 60`	Draw a line to (10,60)
`L 40 42.68`	Draw a line to (40,42.68)
`M 60 60`	Start a new subpath; move pen to (60,60)—no line is drawn
`L 90 60`	Draw a line to (90,60)
`L 60 42.68`	Draw a line to (60,42.68)



You may have noticed the path data doesn't look very much like the typical values for XML attributes. Because the entire path data is contained in one attribute rather than an individual element for each point or line segment, a path takes up less memory when read into a Document Object Model structure by an XML parser. Additionally, a path's compact notation allows a complex graphic to be transmitted without requiring a great deal of bandwidth.

If you want to use a `<path>` to draw a rectangle, you can draw all four lines, or you can draw the first three lines and then use the *closepath* command, denoted by a capital `Z`, to draw a straight line back to the beginning point of the current subpath. Example 7-2

is the SVG for Figure 7-2, which shows a rectangle drawn the hard way, a rectangle drawn with *closepath*, and a path that draws two triangles by opening and closing two subpaths.

Example 7-2. Using closepath

```
<g style="stroke: black; fill: none;">
  <!-- rectangle; all four lines -->
  <path d="M 10, 10 L 40, 10 L 40, 30 L 10, 30 L 10, 10"/>

  <!-- rectangle with closepath -->
  <path d="M 60 10 L 90 10 L 90 30 L 60 30 Z"/>

  <!-- two 30-degree triangles -->
  <path d="M 40 60 L 10 60 L 40 42.68 Z
     M 60 60 L 90 60 L 60 42.68 Z"/>
</g>
```

Examining the last path more closely:

Value	Action
`M 40 60`	Move pen to (40,60)
`L 10 60`	Draw a line to (10,60)
`L 40 42.68`	Draw a line to (40,42.68)
`Z`	Close path by drawing a straight line to (40,60), where this subpath began
`M 60 60`	Start a new subpath; move pen to (60,60)—no line is drawn
`L 90 60`	Draw a line to (90,60)
`L 60 42.68`	Draw a line to (60,42.68)
`Z`	Close path by drawing a straight line to (60,60), where this subpath began

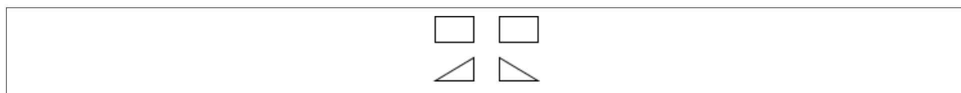


Figure 7-2. Result of using closepath

There is another difference between drawing the rectangle with all four lines and using a *closepath* command. When you close the path, the start and end lines are joined together to form a continuous shape for stroking styles. The difference is noticeable if you are using wide strokes or `stroke-linecap` and `stroke-linejoin` effects. Example 7-3 uses a larger stroke width, and Figure 7-3 shows the result, magnified to make the difference more clearly visible.

Example 7-3. Individual lines versus closepath

```
<g style="stroke: gray; stroke-width: 8; fill: none;">

    <!-- rectangle; all four lines -->
    <path d="M 10 10 L 40 10 L 40 30 L 10 30 L 10 10"/>

    <!-- rectangle with closepath -->
    <path d="M 60 10 L 90 10 L 90 30 L 60 30 Z"/>
</g>
```

Figure 7-3. Result of individual strokes versus closepath

Relative moveto and lineto

The preceding commands are all represented by uppercase letters, and the coordinates are presumed to be *absolute* coordinates. If you use a lowercase command letter, the coordinates are interpreted as being *relative* to the current pen position. Thus, the following two paths are equivalent:

```
    <path d="M 10 10 L 20 10 L 20 30  M 40 40 L 55 35"
        style="stroke: black;"/>
    <path d="M 10 10 l 10  0 l  0 20  m 20 10 l 15 -5"
        style="stroke: black;"/>
```

If you start a path with a lowercase `m` (*moveto*), its coordinates will be interpreted as an absolute position, as there's no previous pen position from which to calculate a relative position. All the other commands in this chapter also have the same upper- and lowercase distinction. An uppercase command's coordinates are absolute, and a lowercase command's coordinates are relative. The *closepath* command, which has no coordinates, has the same effect in both upper- and lowercase.

Path Shortcuts

If content is king and design is queen, then bandwidth efficiency is the royal courtier who keeps the palace running smoothly. Because any nontrivial drawing will have paths with many tens of coordinate pairs, the `<path>` element has shortcuts that allow you to represent a path in as few bytes as possible.

The Horizontal lineto and Vertical lineto Commands

Horizontal and vertical lines are common enough to warrant shortcut commands. A path may specify a horizontal line with an `H` command followed by an absolute

x-coordinate, or an `h` command followed by a relative *x*-coordinate. Similarly, a vertical line is specified with a `V` command followed by an absolute *y*-coordinate, or a `v` command followed by a relative *y*-coordinate.

The following table compares the short and long way to draw horizontal and vertical lines:

Shortcut	Equivalent to	Effect
`H 20`	`L 20 current_y`	Draws a line to absolute location (20,`current_y`)
`h 20`	`l 20 0`	Draws a line to (`current_x` + 20,`current_y`)
`V 20`	`L current_x 20`	Draws a line to absolute location (`current_x`,20)
`v 20`	`l 0 20`	Draws a line to location (`current_x`, `current_y` + 20)

Thus, the following path draws a rectangle 15 units in width and 25 units in height, with the upper-left corner at coordinates (12,24).

```
<path d="M 12 24 h 15 v 25 h -15 z"/>
```

Notational Shortcuts for a Path

Paths can also be made shorter by applying the following two rules:

- You may place multiple sets of coordinates after an `L` or `l`, just as you do in the `<polyline>` element. The following six paths all draw the same diamond shown in Figure 7-4; the first three are in absolute coordinates and the last three in relative coordinates. The third and sixth paths have an interesting twist—if you place multiple pairs of coordinates after a *moveto*, all the pairs after the first are presumed to be preceded by a *lineto*: [1]

```
<g style="fill:none; stroke: black">
  <path d="M 30 30 L 55 5 L 80 30 L 55 55 Z"/>
  <path d="M 30 30 L 55 5 80 30 55 55 Z"/>
  <path d="M 30 30 55 5 80 30 55 55 Z"/>
  <path d="m 30 30 l 25 -25 l 25 25 l -25 25 z"/>
  <path d="m 30 30 l 25 -25 25 25 -25 25 z"/>
  <path d="m 30 30 25 -25 25 25 -25 25 z"/>
</g>
```

1. You can also put multiple single coordinates after a *horizontal lineto* or *vertical lineto*, although you'll only notice an effect if you're using line markers, which we haven't discussed yet. `H 25 35 45` is the same as `H 45`, and `v 11 13 15` is the same as `v 39`.

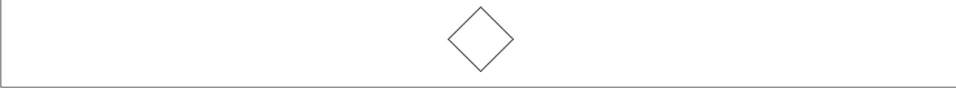


Figure 7-4. Result of drawing a diamond with a path

- Any unnecessary whitespace may be eliminated. You don't need a blank after a command letter because all commands are one letter only. You don't need a blank between a number and a command because the command letter can't be part of the number. You don't need a blank between a positive and a negative number because the leading minus sign of the negative number can't be a part of the positive number. This lets you reduce the third and sixth paths in the preceding listing even further:

  ```
  <path d="M30 30 55 5 80 30 55 55Z"/>
  <path d="m30 30 25-25 25 25-25 25z"/>
  ```

  Another example of the whitespace elimination rule in action is shown by the example that drew a rectangle 15 units in width and 25 units in height, with the upper-left corner at coordinates (12,24):

  ```
  <path d="M 12 24 h 15 v 25 h -15 z"/> <!-- original -->
  <path d="M12 24h15v25h-15z"/> <!-- shorter -->
  ```

Elliptical Arc

Lines are simple; two points on a path uniquely determine the line segment between them. Because an infinite number of curves can be drawn between two points, you must give additional information to draw a curved path between them. The simplest of the curves we will examine is the elliptical arc—that is, drawing a section of an ellipse that connects two points.

Although arcs are visually the simplest curves, specifying a unique arc requires the *most* information. The first pieces of information you need to specify are the *x*- and *y*-radii of the ellipse on which the points lie. This narrows it down to two possible ellipses, as you can see in section (a) of Figure 7-5. The two points divide the two ellipses into four arcs. Two of them, (b) and (c), are arcs that measure less than 180 degrees. The other two, (d) and (e), are greater than 180 degrees. If you look at (b) and (c), you will notice they are differentiated by their direction; (b) is drawn in the direction of increasing negative (counterclockwise) angle, and (c) in the direction of increasing positive (clockwise) angle. The same relationship holds true between (d) and (e).

But wait—that still doesn't uniquely specify the potential arcs! There's no law that says the ellipse has to have its *x*-radius parallel to the *x*-axis. Part (f) of Figure 7-5 shows the two points with their candidate ellipses rotated 30 degrees with respect to the *x*-axis.

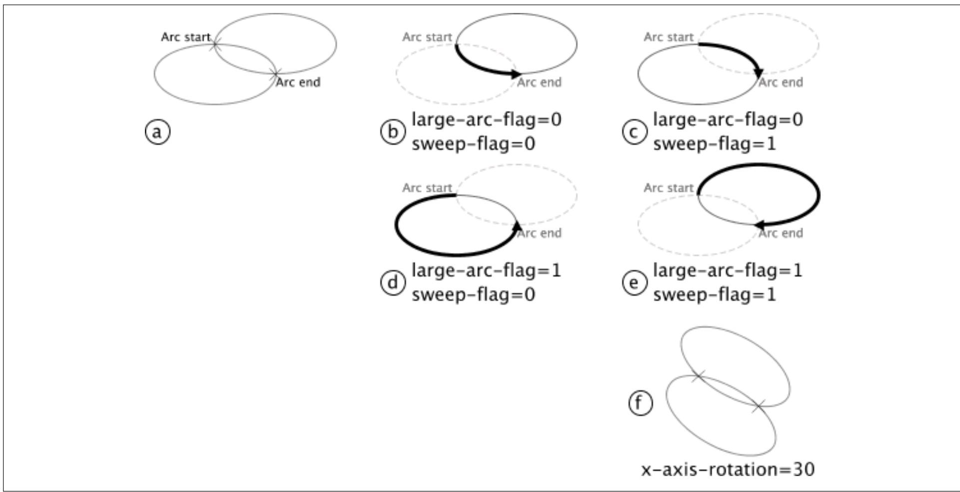


Figure 7-5. Variations of the elliptical arc command

(Figure 7-5 is adapted from the one found in section 8.3.8 of the World Wide Web Consortium's SVG specification.)

Thus, an arc command begins with the `A` abbreviation for absolute coordinates or `a` for relative coordinates, and is followed by seven parameters:

- The *x*- and *y*-radius of the ellipse on which the points lie.
- The *`x-axis-rotation`* of the ellipse.
- The *`large-arc-flag`*, which is 0 if the arc's measure is less than 180 degrees, or 1 if the arc's measure is greater than or equal to 180 degrees.
- The *`sweep-flag`*, which is 0 if the arc is to be drawn in the negative angle direction, or 1 if the arc is to be drawn in the positive angle direction.
- The ending *x*- and *y*- coordinates of the ending point. (The starting point is determined by the last point drawn or the last *moveto* command.)

Here are the paths used to draw the elliptical arcs in sections (b) through (e) of Figure 7-5:

```
<path d="M 125,75 A100,50 0 0,0 225,125"/> <!-- b -->
<path d="M 125,75 A100,50 0 0,1 225,125"/> <!-- c -->
<path d="M 125,75 A100,50 0 1,0 225,125"/> <!-- d -->
<path d="M 125,75 A100,50 0 1,1 225,125"/> <!-- e -->
```

Online, you can experiment with all the arc parameters to see what they do:

http://oreillymedia.github.io/svg-essentials-examples/ch07/arc.html

As a further example, let's enhance the background we started in Example 5-8 to complete the yin-yang symbol that is part of the South Korean flag. Example 7-4 keeps the

full ellipses as `<ellipse>` elements, but creates the semicircles it needs with paths. The result is shown in Figure 7-6.

Example 7-4. Using elliptical arc

```
<svg width="400px" height="300px" viewBox="0 0 400 300"
  xmlns="http://www.w3.org/2000/svg">
  <!-- gray drop shadow -->
  <ellipse cx="154" cy="154" rx="150" ry="120" style="fill: #999999;"/>

  <!-- light blue ellipse -->
  <ellipse cx="152" cy="152" rx="150" ry="120" style="fill: #cceeff;"/>

  <!-- large light red semicircle fills upper half,
      followed by small light red semicircle that dips into
      lower-left half of symbol -->
  <path d="M 302 152 A 150 120, 0, 1, 0, 2 152
      A 75 60, 0, 1, 0, 152 152" style="fill: #ffcccc;"/>

  <!-- light blue semicircle rises into upper-right half of symbol -->
  <path d="M 152 152 A 75 60, 0, 1, 1, 302 152" style="fill: #cceeff;"/>
</svg>
```

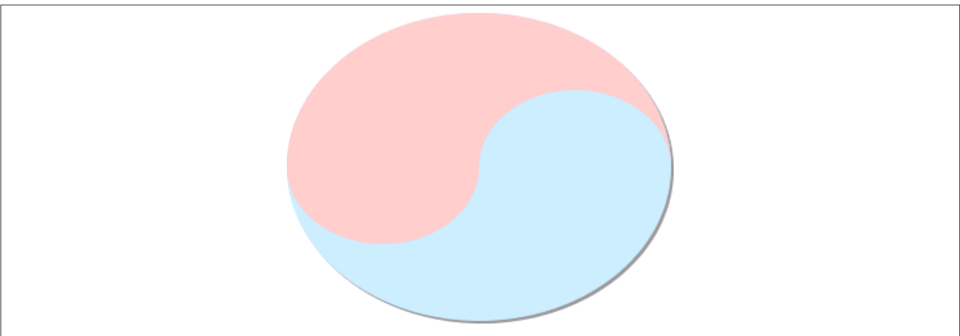


Figure 7-6. Result of using elliptical arc

You cannot draw a full ellipse with a single path command; if the starting and ending points of the arc are the same, there are infinite ways to position the ellipse. SVG viewers will skip such an arc command. If you specify an ellipse that is too small to reach between the starting and ending points, the SVG viewer will scale up the ellipse until it is just big enough.

For exact details on how out-of-range parameters are handled, see the specification's arc implementation notes.

Converting from Other Arc Formats

You may be wondering why you can't specify an arc by defining a center point for the ellipse, its *x*- and *y*-radius, the starting angle, and the extent of the angle's arc, as some other vector graphics systems do. This is a straightforward method of specification, and is excellent for drawing arcs as single objects. This, paradoxically, is exactly why SVG instead chooses such a seemingly eccentric method to specify arcs. In SVG, an arc is not presumed to be living in lonely splendor; it is intended to be part of a connected path of lines and curves. (For example, a rounded rectangle is precisely that—a series of lines and elliptical arcs.) Thus, it makes sense to specify an arc by its endpoints.

Sometimes, though, you do want an isolated semicircle (or, more accurately, semi-ellipse). Presume you have an ellipse specified as follows:

```
<ellipse cx="cx" cy="cy" rx="rx" ry="ry"/>
```

Here are the paths to draw the four possible semi-ellipses (items in parentheses are intended as algebraic expressions to be calculated):

```
<!-- northern hemisphere -->
<path d="M (cx - rx) cy
  A rx ry 0 1 1 (cx + rx) cy"/>
<!-- southern hemipshere -->
<path d="M (cx - rx) cy
  A rx ry 0 1 0 (cx + rx) cy"/>
<!-- eastern hemisphere -->
<path d="M cx (cy - ry)
  A rx ry 0 1 1 cx (cy + ry)"/>
<!-- western hemisphere -->
<path d="M cx (cy - ry)
  A rx ry 0 1 0 cx (cy + ry)"/>
```

Sometimes you may want to draw an arbitrary arc that has been specified in *center-and-angles* notation and wish to convert it to SVG's *endpoint-and-sweep* format. In other cases, you may want to convert an arc from the SVG format to a center-and-angles format. The mathematics for this second case is rather complex, and is detailed in the SVG specification. You can see a JavaScript version of these conversions in Appendix F.

Bézier Curves

Arcs can be characterized as clean and functional, but one would rarely use the word *graceful* to describe them. If you want graceful, you need to use curves that are produced by graphing quadratic and cubic equations. Mathematicians have known about these curves for literally hundreds of years, but drawing them was always a computationally demanding task. This changed when Pierre Bézier, an engineer who worked for French car manufacturer Rénault, and Paul de Casteljau, a physicist and mathematician who

worked for Citroën, developed and promoted a computationally convenient way to generate these curves.

If you have used graphics programs like Adobe Illustrator, you draw these Bézier curves by specifying two points and then moving a "handle" as shown in the following diagram. The end of this handle is called the *control point*, because it controls the shape of the curve. As you move the handle, the curve changes in a way that, to the uninitiated, is completely mystifying. Mike Woodburn, a graphic designer at Key Point Software, suggests Figure 7-7 as a way to visualize how the control point and the curve interact: imagine the line is made of flexible metal. Inside the control point is a magnet; the closer a point is to the control point, the more strongly it is attracted.

Figure 7-7. How graphics programs draw Bézier curves

Another way to visualize the role of the control point is based on the de Casteljau method of constructing the curves. We will use this approach in the following sections. See further details on the underlying mathematics, presented in a remarkably lucid fashion.

Quadratic Bézier Curves

The simplest of the Bézier curves is the quadratic curve. You specify a beginning point, an ending point, and a control point. Imagine two tent poles placed at the endpoints of the line. These tent poles meet at the control point. Stretched between the centers of the tent poles is a rubber band. The place where the curve bends is tied to the exact center of that rubber band. This situation is shown in Figure 7-8.

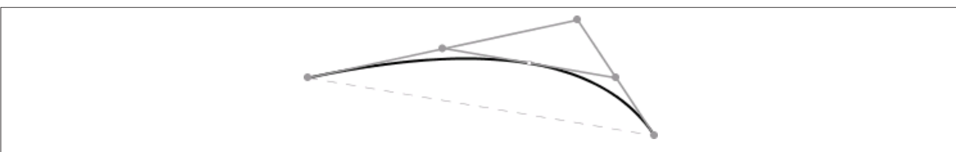


Figure 7-8. Visualizing a quadratic Bézier curve

The lines between the start and endpoints and the control point are tangent to the start and end of the curve. The curve starts by following the line to the control point, but then bends over to reach the midpoint heading in the same direction as the "tent pole" line. The curve ends by sliding up alongside the line from the control point to the endpoint. Programs like Adobe Illustrator show you only one of the "tent poles." The

next time you're using such a program, mentally add in the second pole and the resulting curves will be far less mysterious.

That's the concept; now for the practical matter of actually producing such a curve in SVG. You specify a quadratic curve in a `<path>` data with the `Q` or `q` command. The command is followed by two sets of coordinates that specify a control point and an endpoint. The uppercase command implies absolute coordinates; lowercase implies relative coordinates. The curve in Figure 7-8 was drawn from (30,75) to (300,120) with the control point at (240,30), and was specified in SVG as follows:

```
<path d="M30 75 Q240 30, 300 120" style="stroke: black; fill: none;"/>
```

The online example shows it with and without the "tent poles":

http://oreillymedia.github.io/svg-essentials-examples/ch07/quadratic-bezier.html

You may specify several sets of coordinates after a quadratic curve command. This will generate a poly-Bézier curve. Presume you want a `<path>` that draws a curve from (30,100) to (100,100) with a control point at (80,30) and then continues with a curve to (200,80) with a control point at (130,65). Here is the SVG for this path, with control point coordinates in bold. The result is shown in the left half of Figure 7-9; the control points and lines are shown in the right half of the figure:

```
<path d="M30 100 Q 80 30, 100 100, 130 65, 200 80"/>
```

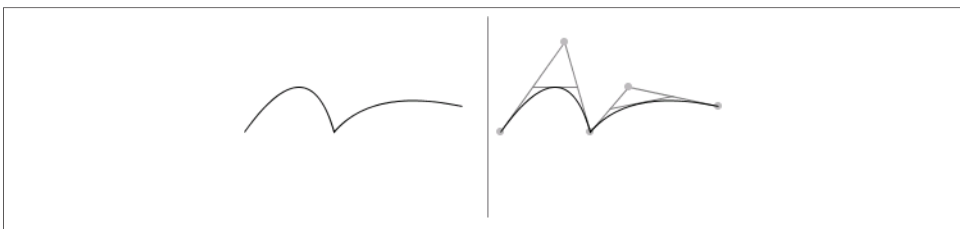
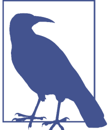


Figure 7-9. Quadratic poly-Bézier curve

You are probably wondering, "What happened to *graceful*? That curve is just lumpy." This is an accurate assessment. Just because curves are connected doesn't mean they will look good together. That's why SVG provides the *smooth quadratic curve* command, which is denoted by the letter `T` (or `t` if you want to use relative coordinates). The command is followed by the next endpoint of the curve; the control point is calculated automatically, as the specification says, by "reflection of the control point on the previous command relative to the current point."



For the mathematically inclined, the new control point *x2*, *y2* is calculated from the previous segment's endpoint *x*, *y* and the previous control point *x1*, *y1* with these formulas:

```
x2 = x + (x - x1) = 2 * x - x1
y2 = y + (y - y1) = 2 * y - y1
```

Here is a quadratic Bézier curve drawn from (30,100) to (100,100) with a control point at (80,30) and then smoothly continued to (200,80). The left half of Figure 7-10 shows the curve; the right half shows the control points. The reflected control point is shown with a dashed line. Gracefulness has returned!

```
<path d="M30 100 Q 80 30, 100 100 T 200 80"/>
```

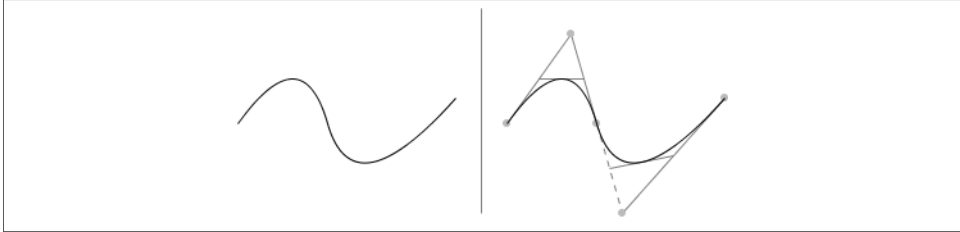


Figure 7-10. Smooth quadratic poly-Bézier curve

The online example allows you to experiment with the quadratic poly-Bézier curve:

http://oreillymedia.github.io/svg-essentials-examples/ch07/smooth-quadratic-bezier.html

Cubic Bézier Curves

A single quadratic Bézier curve has exactly one peak or valley per curve segment. While these curves are more versatile than simple arcs, you can do even better by using cubic Bézier curves, which can have both a peak and a valley in the same segment, among other possible shapes. In other words, a cubic curve can contain an inflection point (the point where the curve changes from bending in one direction to bending in the other).

The difference between the quadratic and cubic curves is that the cubic curve has two control points, one for each endpoint. The technique for generating the cubic curve is similar to that for generating the quadratic curve. As you can see in Figure 7-11, you draw three lines that connect the endpoints and control points (a), and connect their midpoints. That produces two lines (b). You connect *their* midpoints, and that produces

one line (c), whose midpoint determines one of the points on the final curve.[2] Notice that the start, end, and middle angles of the curve are tangent to (they "follow") the control lines.

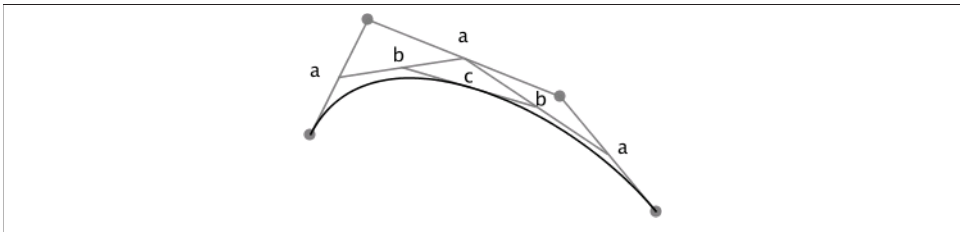


Figure 7-11. Visualizing a cubic Bézier curve

To specify such a cubic curve, use the `C` or `c` command. The command is followed by three sets of coordinates that specify the control point for the start point, the control point for the endpoint, and the endpoint. As with all the other path commands, an uppercase command implies absolute coordinates; lowercase implies relative coordinates. The curve in the preceding diagram was drawn from (20,80) to (200,120) with control points at (50,20) and (150,60). The SVG for the path was as follows:

```
<path d="M20 80 C 50 20, 150 60, 200 120"
    style="stroke: black; fill: none;"/>
```

There are many interesting curves you can draw, depending upon the relationship of the control points (see Figure 7-12). To make the graphic cleaner, we show only the lines from each endpoint to its control point.

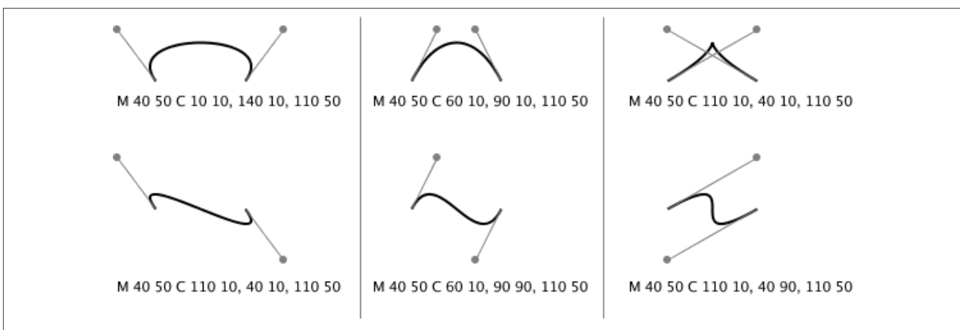


Figure 7-12. Result of cubic Bézier control point combinations

2. We're dispensing with the tent analogy; it gets too unwieldy. Curves based on yurts and geodesic domes are left as exercises for the reader.

Experiment with these combinations and more in the online example:

http://oreillymedia.github.io/svg-essentials-examples/ch07/cubic-bezier.html

As with quadratic curves, you can construct a cubic poly-Bézier by specifying several sets of coordinates after a cubic curve command. The last point of the first curve becomes the first point of the next curve, and so on. Here is a `<path>` that draws a cubic curve from (30,100) to (100,100) with control points at (50,50) and (70,20); it is immediately followed by a curve that doubles back to (65,100) with control points at (110,130) and (45,150). Here is the SVG for this path, with control point coordinates in bold:

```
<path d="M30 100 C 50 50, 70 20, 100 100,
                  110, 130, 45, 150, 65, 100"/>
```

The result is shown in the left half of Figure 7-13; the control points and lines are shown in the right half of the diagram.

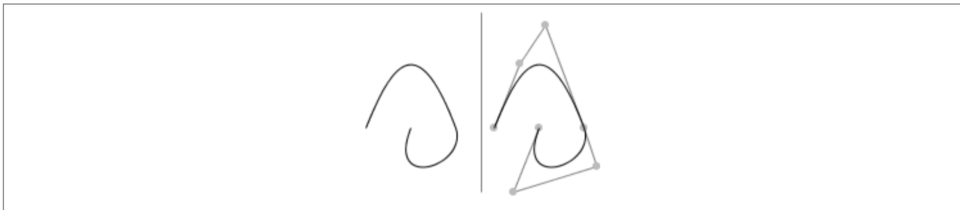


Figure 7-13. Cubic poly-Bézier curve

If you want to guarantee a smooth join between curves, you can use the `S` command (or `s` if you want to use relative coordinates). In a manner analogous to that of the `T` command for quadratic curves, the new curve will take the previous curve's endpoint as its starting point, and its first control point will be the reflection of the previous ending control point. All you need to supply is the control point for the next endpoint on the curve, followed by the next endpoint itself.

Here is a cubic Bézier curve drawn from (30,100) to (100,100) with control points at (50,30) and (70,50). It continues smoothly to (200,80), using (150,40) as its ending control point. The left half shows the curve; the right half shows the curve with the control points. The reflected control point is shown with a dashed line in Figure 7-14:

```
<path d="M30 100 C 50 30, 70 50, 100 100 S 150 40, 200 80"/>
```

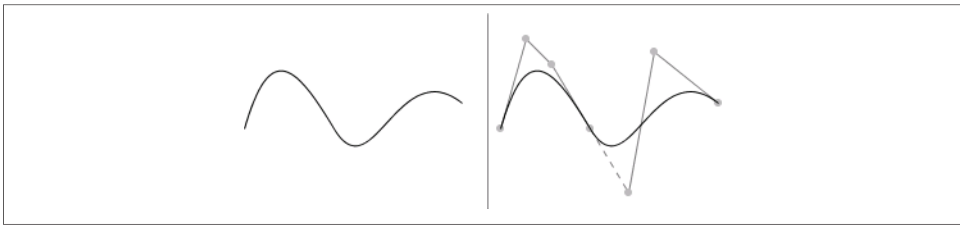


Figure 7-14. Smooth cubic poly-Bézier curve

Path Reference Summary

In Table 7-1, uppercase commands use absolute coordinates, and lowercase commands use relative coordinates.

Table 7-1. Path commands

Command	Arguments	Effect
`M m`	*`x y`*	Move to given coordinates.
`L l`	*`x y`*	Draw a line to the given coordinates. You may supply multiple sets of coordinates to draw a polyline.
`H h`	*`x`*	Draw a horizontal line to the given *x*-coordinate.
`V v`	*`y`*	Draw a vertical line to the given *x*-coordinate.
`A a`	*`rx ry x-axis-rotation large-arc sweep x y`*	Draw an elliptical arc from the current point to (*x*,*y*). The points are on an ellipse with *x*-radius *`rx`* and *y*-radius *`ry`*. The ellipse is rotated *`x-axis-rotation`* degrees. If the arc is less than 180 degrees, *`large-arc`* is 0; if greater than 180 degrees, *`large-arc`* is 1. If the arc is to be drawn in the positive direction, *`sweep`* is 1; otherwise it is 0.
`Q q`	*`x1 y1 x y`*	Draw a quadratic Bézier curve from the current point to (*x*,*y*) using control point (*x1*,*y1*).
`T t`	*`x y`*	Draw a quadratic Bézier curve from the current point to (*x*,*y*). The control point will be the reflection of the previous `Q` command's control point. If there is no previous curve, the current point will be used as the control point.
`C c`	*`x1 y1 x2 y2 x y`*	Draw a cubic Bézier curve from the current point to (*x*,*y*) using control point (*x1*,*y1*) as the control point for the beginning of the curve and (*x2*,*y2*) as the control point for the endpoint of the curve.
`S s`	*`x2 y2 x y`*	Draw a cubic Bézier curve from the current point to (*x*,*y*), using (*x2*,*y2*) as the control point for this new endpoint. The first control point will be the reflection of the previous `C` command's ending control point. If there is no previous curve, the current point will be used as the first control point.

Paths and Filling

The information described in Chapter 4 in "Filling Polygons That Have Intersecting Lines" on page 49 is also applicable to paths, which not only can have intersecting lines, but also can have "holes" in them. Consider the paths in Example 7-5, both of which draw nested squares. In the first path, both squares are drawn clockwise; in the second path, the outer square is drawn clockwise and the inner square is drawn counterclockwise.

Example 7-5. Using different fill-rule values on paths

```
<!-- both paths clockwise -->
<path d="M 0 0, 60 0, 60 60, 0 60 Z
    M 15 15, 45 15, 45 45, 15 45Z"/>

<!-- outer path clockwise; inner path counterclockwise -->
<path d="M 0 0, 60 0, 60 60, 0 60 Z
    M 15 15, 15 45, 45 45, 45 15Z"/>
```

Figure 7-15 shows there is a difference when you use a `fill-rule` of `nonzero`, which takes into account the direction of the lines when determining whether a point is inside or outside a path. Using a `fill-rule` of `evenodd` produces the same result for both paths; it uses total number of lines crossed and ignores their direction.

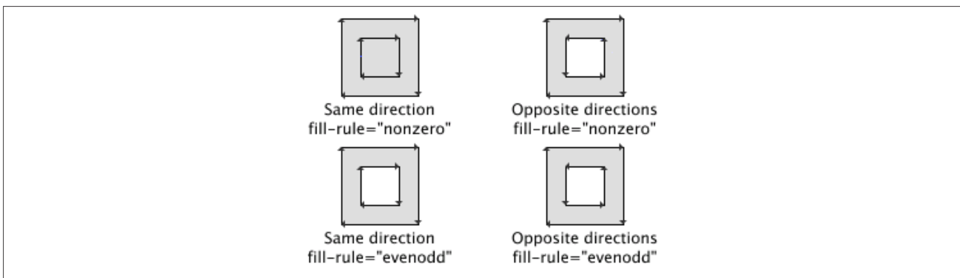


Figure 7-15. Result of using different fill-rule values

The <marker> element

Consider the following path, which uses a line, an elliptical arc, and another line to draw the rounded corner in Figure 7-16:

```
<path d="M 10 20 100 20 A 20 30 0 0 1 120 50 L 120 110"
    style="fill: none; stroke: black;"/>
```

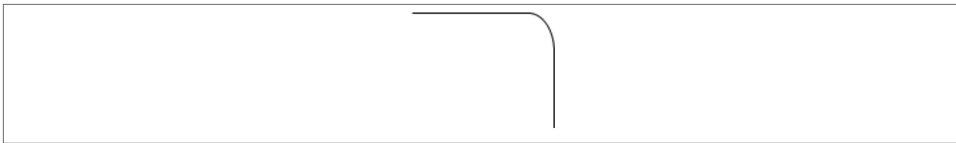


Figure 7-16. Lines and arc

Presume you want to mark the direction of the path by putting a circle at the beginning, a solid triangle at the end, and arrowheads at the other vertices, as shown in Figure 7-17. To achieve this effect, you need to construct three `<marker>` elements and tell the `<path>` to reference them.

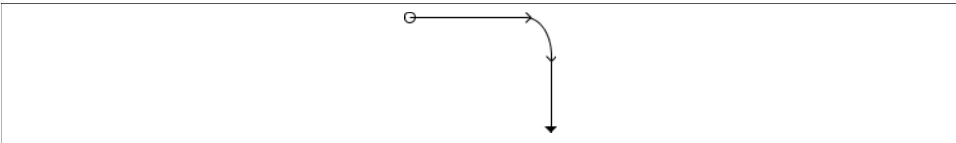


Figure 7-17. Lines and arc with markers

As a first step, consider Example 7-6, which adds the circular marker. A marker is a "self-contained" graphic with its own private set of coordinates, so you have to specify its `markerWidth` and `markerHeight` in the starting `<marker>` tag. That is followed by the SVG elements required to draw the marker, and ends with a closing `</marker>`. A `<marker>` element does not display by itself, but we are putting it in a `<defs>` element because that's where reusable elements belong.

We want the circle to be at the beginning of the path, so we add a `marker-start` to the `style` in the `<path>`.[3] The value of this property is a URL reference to the `<marker>` element we've just created.

Example 7-6. First attempt at circular marker

```
<defs>
<marker id="mCircle" markerWidth="10" markerHeight="10">
    <circle cx="5" cy="5" r="4" style="fill: none; stroke: black;"/>
</marker>
</defs>

<path  d="M 10 20 100 20 A 20 30 0 0 1 120 50 L 120 110"
    style="marker-start: url(#mCircle);
    fill: none; stroke: black;"/>
```

3. Yes, markers are considered to be part of presentation rather than structure. This is one of those gray areas where you could argue either case.

The result in Figure 7-18 is not quite what you planned…

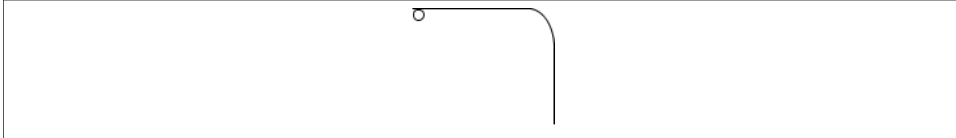


Figure 7-18. Misplaced circular marker

The reason the circle appears in the wrong place is that, by default, the start marker's (0,0) point is aligned with the beginning coordinate of the path. Example 7-7 adds `refX` and `refY` attributes specifying which coordinates (in the marker's system) are to align with the beginning coordinate. Once these are added, the circular marker appears exactly where it is desired in Figure 7-19.

Example 7-7. Correctly placed circular marker

```
<marker id="mCircle" markerWidth="10" markerHeight="10"
        refX="5" refY="5">
    <circle cx="5" cy="5" r="4" style="fill: none; stroke: black;"/>
</marker>
```

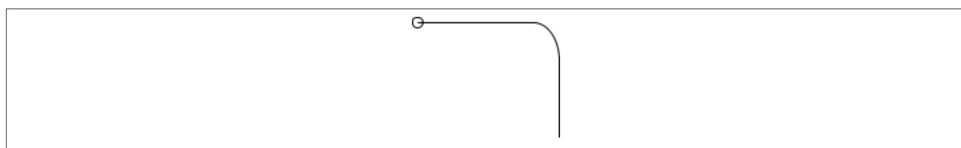


Figure 7-19. Correctly placed circular marker

Given this information, you can now write Example 7-8, which adds the triangular marker and references it as the `marker-end` for the path. Then we can add the arrowhead marker and reference it as the `marker-mid`. The `marker-mid` will be attached to every vertex except the beginning and end of the path. Notice that the `refX` and `refY` attributes have been set so the wide end of the arrowhead aligns with the intermediate vertices, while the tip of the solid triangle aligns with the ending vertex. Figure 7-20 shows the result, which draws the first marker correctly but not the others.

Example 7-8. Attempt to use three markers

```
<defs>
    <marker id="mCircle" markerWidth="10" markerHeight="10"
        refX="5" refY="5">
        <circle cx="5" cy="5" r="4" style="fill: none; stroke: black;"/>
    </marker>

    <marker id="mArrow" markerWidth="4" and markerHeight="8"
```

```
        refX="0" refY="4">
        <path d="M 0 0 4 4 0 8" style="fill: none; stroke: black;"/>
    </marker>

    <marker id="mTriangle" markerWidth="5" markerHeight="10"
        refX="5" refY="5">
        <path d="M 0 0 5 5 0 10 Z" style="fill: black;"/>
    </marker>
</defs>

<path d="M 10 20 100 20 A 20 30 0 0 1 120 50 L 120 110"
    style="marker-start: url(#mCircle);
        marker-mid: url(#mArrow);
        marker-end: url(#mTriangle);
        fill: none; stroke: black;"/>
```

Figure 7-20. Incorrectly oriented markers

To get the effect you want, you must explicitly set a marker's `orient` attribute to `auto`. This makes the marker automatically rotate to match the direction of the path.[4] (You may also specify a number of degrees, in which case the marker will always be rotated by that amount.) Here in Example 7-9 are the markers, set to produce the effect shown in Figure 7-17. You don't need to orient the circle; it looks the same no matter how it's rotated.

Example 7-9. Correctly oriented markers

```
<defs>
    <marker id="mCircle" markerWidth="10" markerHeight="10"
        refX="5" refY="5">
        <circle cx="5" cy="5" r="4" style="fill: none; stroke: black;"/>
    </marker>

    <marker id="mArrow" markerWidth="6" markerHeight="10"
        refX="0" refY="4" orient="auto">
        <path d="M 0 0 4 4 0 8" style="fill: none; stroke: black;"/>
    </marker>

    <marker id="mTriangle" markerWidth="5" markerHeight="10"
        refX="5" refY="5" orient="auto">
```

4. To be exact, the rotation is the average of the angle of the direction of the line going into the vertex and the direction of the line going out of the vertex.

```
        <path d="M 0 0 5 5 0 10 Z" style="fill: black;"/>
    </marker>
</defs>

<path d="M 10 20 100 20 A 20 30 0 0 1 120 50 L 120 110"
    style="marker-start: url(#mCircle);
        marker-mid: url(#mArrow);
        marker-end: url(#mTriangle);
        fill: none; stroke: black;"/>
```

Another useful attribute is the `markerUnits` attribute. If set to `strokeWidth`, the marker's coordinate system is set so one unit equals the stroke width. This makes your marker grow in proportion to the stroke width; it's the default behavior and it's usually what you want. If you set the attribute to `userSpaceOnUse`, the marker's coordinates are presumed to be the same as the coordinate system of the object that references the marker. The marker will remain the same size irrespective of the stroke width.

Marker Miscellanea

If you want the same marker at the beginning, middle, and end of a path, you don't need to specify all of the `marker-start`, `marker-mid`, and `marker-end` properties. Just use the `marker` property and have it reference the marker you want. Thus, if you wanted all the vertices to have a circular marker, as shown in Figure 7-21, you'd write the SVG in Example 7-10.

Example 7-10. Using a single marker for all vertices

```
<defs>
    <marker id="mCircle" markerWidth="10" markerHeight="10"
        refX="5" refY="5">
        <circle cx="5" cy="5" r="4" style="fill: none; stroke: black;"/>
    </marker>
</defs>

<path d="M 10 20 100 20 A 20 30 0 0 1 120 50 L 120 110"
    style="marker: url(#mCircle); fill: none; stroke: black;"/>
```

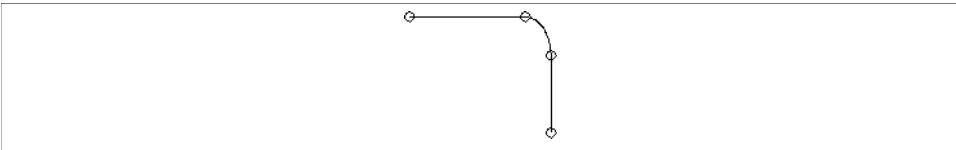


Figure 7-21. Using a single marker for all vertices

It is also possible to set the `viewBox` and `preserveAspectRatio` attributes on a `<marker>` element to gain even more control over its display. For example, you can use a `viewBox`

to define the grid so that the (0,0) coordinate is in the center of your marker; you may want to do this instead of using `refX` and `refY`. `viewBox` and `preserveAspectRatio` work exactly as described in "Specifying User Coordinates for a Viewport" on page 30 and in "Preserving Aspect Ratio" on page 32.

You may reference a `<marker>` in a `<polygon>`, `<polyline>`, or `<line>` element as well as in a `<path>`.

The following thought may have occurred to you: "If a marker can have a path in it, can *that* path have a marker attached to it as well?" The answer is yes, it can, but the second marker must fit into the rectangle established by the first marker's `markerWidth` and `markerHeight`. Remember that just because a thing can be done does not mean it should be done. If you need such an effect, you are probably better off to draw the secondary marker as a part of the primary marker rather than attempting to nest markers.

Make sure you don't define a part of a marker to use itself as a marker. This could happen if you had a CSS rule like this to give all your paths a star as a marker:

```
path {marker: url(#star)}
```

If the `<marker>` with id `star` has a `<path>` in it, that path would refer to itself in an infinite loop. To prevent this, you would add a CSS rule that says not to put any marker on a path that is part of the star marker:

```
path {marker: url(#star)}
marker#star path {marker: none}
```

CHAPTER 8
Patterns and Gradients

To this point, you have used only solid colors to fill and outline graphic objects. You are not restricted to using solid colors; you may also use a pattern or a gradient to fill or outline a graphic. That's what we'll examine in this chapter.

Patterns

To use a pattern, you define a graphic object that is replicated horizontally and vertically to fill another object (or stroke). This graphic object is called a *tile*, because the act of filling an object with a pattern is very much like covering an area of a floor with tiles. In this section, we will use the quadratic curve drawn by the SVG in Example 8-1 as our tile. It's outlined in gray to show its area (20 by 20 user units) clearly.

Example 8-1. Path for a pattern tile

```
<path d="M 0 0 Q 5 20 10 10 T 20 20"
    style="stroke: black; fill: none;"/>
<path d="M 0 0 h20 v20 h-20 z"
    style="stroke: gray; fill: none;"/>
```

Figure 8-1 is zoomed in so you can see it in detail.

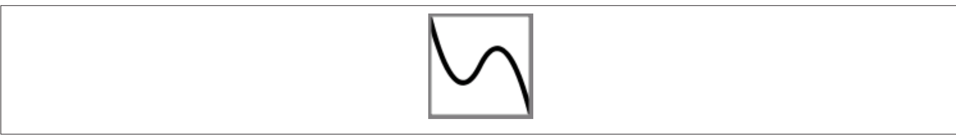


Figure 8-1. Zoomed-in view of pattern tile

patternUnits

To create a pattern tile, you must enclose the <path> elements that describe your tile in a <pattern> element, and then make several decisions. The first decision is how you wish to space the tiles, and this is reflected in the patternUnits attribute. Do you want the tiles spaced to fill a certain percentage of each object they're applied to, or do you want them spaced at equal intervals, no matter what the size of the object they're filling?

If you want the tile dimensions on an object-by-object basis, you specify the pattern's upper-left x and y coordinates, and its width and height as percentages or decimals in the range 0 to 1, and set the patternUnits attribute to objectBoundingBox. An object's *bounding box* is the smallest rectangle that completely encloses a particular graphic object. Example 8-2 creates a sample tile that will be replicated five times horizontally and five times vertically in any object that it fills.

Example 8-2. Tiles spaced with patternUnits set to objectBoundingBox

```
<defs>
<pattern id="tile" x="0" y="0" width="20%" height="20%"
    patternUnits="objectBoundingBox">
<path d="M 0 0 Q 5 20 10 10 T 20 20"
    style="stroke: black; fill: none;"/>
<path d="M 0 0 h 20 v 20 h -20 z"
    style="stroke: gray; fill: none;"/>
</pattern>
</defs>

<rect x="20" y="20" width="100" height="100"
    style="fill: url(#tile); stroke: black;"/>
<rect x="135" y="20" width="70" height="80"
    style="fill: url(#tile); stroke: black;"/>
<rect x="220" y="20" width="150" height="130"
    style="fill: url(#tile); stroke: black;"/>
```

In Figure 8-2, the leftmost rectangle, which is 100 user units wide and tall, provides an exact fit for five tiles that are each 20 user units wide and tall. In the middle rectangle, the width and height aren't great enough to show any one pattern tile completely, so they are truncated. In the rightmost rectangle, extra space is added, because the rectangle's width and height exceeds five times the space required for a single tile. In all cases, because x and y are both set, the upper-left corner of the tile coincides with the upper-left corner of the rectangle.

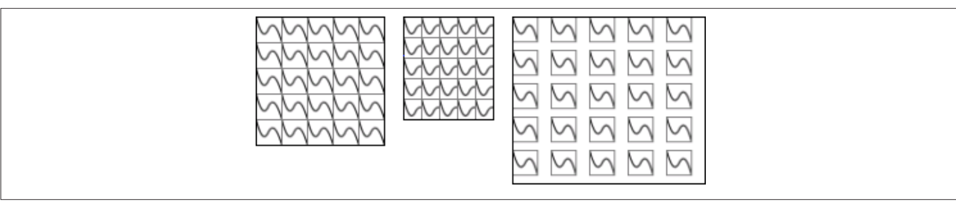


Figure 8-2. Tiles spaced by objectBoundingBox

If you're used to most graphics programs, this behavior comes as somewhat of a shock. Typical graphic editing programs put tiles directly next to one another to fill the area, no matter what its size. There is never extra padding between tiles, and tiles are cut off only by the edge of the object they're filling. If this is the behavior that you want, you must set the `patternUnits` attribute to `userSpaceOnUse`, and specify the `x` and `y` coordinates, and the `width` and `height` of the tile in user units. Example 8-3 uses the same sample tile, set to its exact width and height of 20 user units.

Example 8-3. Changing patternUnits to userSpaceOnUse

http://oreillymedia.github.io/svg-essentials-examples/ch08/patternunits.html

```
<defs>
<pattern id="tile" x="0" y="0" width="20" height="20"
    patternUnits="userSpaceOnUse">
<path d="M 0 0 Q 5 20 10 10 T 20 20"
    style="stroke: black; fill: none;"/>
<path d="M 0 0 h 20 v 20 h -20 z"
    style="stroke: gray; fill: none;"/>
</pattern>
</defs>

<rect x="20" y="20" width="100" height="100"
    style="fill: url(#tile); stroke: black;"/>
<rect x="135" y="20" width="70" height="80"
    style="fill: url(#tile); stroke: black;"/>
<rect x="220" y="20" width="150" height="130"
    style="fill: url(#tile); stroke: black;"/>
```

In Figure 8-3, the tiles have constant size in all three rectangles. Their alignment is, however, dependent upon the underlying coordinate system. The middle rectangle, for example, has an *x*-coordinate that is not a multiple of 20, so the rectangle's upper-left corner doesn't coincide with a tile's upper-left corner. (The top edges do align, because the upper *y*-coordinate of all three rectangles was carefully chosen to be a multiple of 20.)

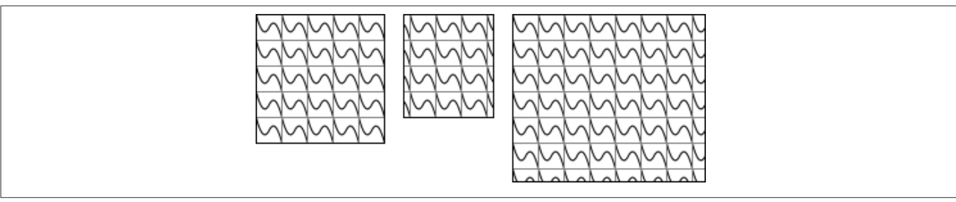
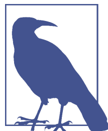


Figure 8-3. Tiles spaced by userSpaceOnUse

If you do not specify a value for `patternUnits`, the default is set to `objectBoundingBox`.

patternContentUnits

You must next decide what units are to be used to express the pattern data itself. By default, the `patternContentUnits` attribute is set to `userSpaceOnUse`. If you set the attributes to `objectBoundingBox`, the path data points are expressed in terms of the object being filled. Example 8-4 shows the SVG that produces Figure 8-4.



If you use `userSpaceOnUse` for your `patternContentUnits`, you should draw any objects to be filled with the upper-left corner of their bounding boxes at the origin (0,0).

If you use `objectBoundingBox`, you will have to reduce the `stroke-width` in the pattern data. The width will also be scaled proportionally to the bounding box, not measured in user units, so a `stroke-width` of 1 would cover the entire tile. In the example, the stroke is set to 0.01, or 1% of the average of the object bounding box's height and width.

Example 8-4. patternContentUnits set to objectBoundingBox

```
<defs>
<pattern id="tile"
    patternUnits="objectBoundingBox"
    patternContentUnits="objectBoundingBox"
     x="0" y="0" width=".2" height=".2">
    <path d="M 0 0 Q .05 .20 .10 .10 T .20 .20"
        style="stroke: black; fill: none; stroke-width: 0.01;"/>
    <path d="M 0 0 h 0.2 v 0.2 h-0.2z"
        style="stroke: black; fill: none; stroke-width: 0.01;"/>
</pattern>

</defs>
```

```
<g transform="translate(20,20)">
<rect x="0" y="0" width="100" height="100"
    style="fill: url(#tile); stroke: black;"/>
</g>

<g transform="translate(135,20)">
<rect x="0" y="0" width="70" height="80"
    style="fill: url(#tile); stroke: black;"/>
</g>

<g transform="translate(220,20)">
<rect x="0" y="0" width="150" height="130"
    style="fill: url(#tile); stroke: black;"/>
</g>
```

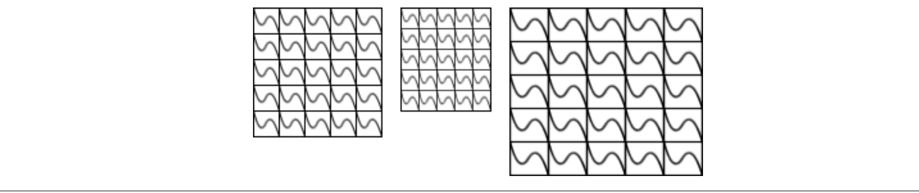


Figure 8-4. patternContentUnits set to objectBoundingBox

If you want to reduce an existing graphic object for use as a tile, it's easier to use the `viewBox` attribute to scale it. Specifying `viewBox` will override any `patternContentUnits` information. Another possible option is to use the `preserveAspectRatio` attribute, as described in "Preserving Aspect Ratio" on page 32. Example 8-5 uses a scaled-down version of the cubic poly-Bézier curve from Figure 7-13 as a tile. The `stroke-width` is set to 5; otherwise, when scaled down, the pattern you see in Figure 8-5 would not be visible.

Example 8-5. Using viewBox to scale a pattern

```
<defs>
<pattern id="tile"
    patternUnits="userSpaceOnUse"
    x="0" y="0" width="20" height="20"
    viewBox="0 0 150 150">
    <path d="M30 100 C 50 50, 70 20, 100 100,
                  110, 130, 45, 150, 65, 100"
        style="stroke: black; stroke-width: 5; fill: none;"/>
</pattern>
</defs>

<rect x="20" y="20" width="100" height="100"
    style="fill: url(#tile); stroke: black;"/>
```

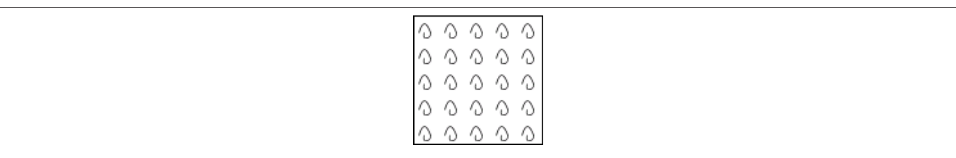


Figure 8-5. Pattern scaled with viewBox

Nested Patterns

Again, this may have occurred to you: "If an object can be filled with a pattern, can *that* pattern be filled with a pattern as well?" The answer is yes. As opposed to nested markers, which are rarely necessary, there are some effects you can't easily achieve without nested patterns. Example 8-6 creates a rectangle filled with circles, all filled with horizontal stripes. This produces the unusual, but valid, striped polka-dot effect shown in Figure 8-6.

Example 8-6. Nested patterns

```
<defs>
  <pattern id="stripe"
    patternUnits="userSpaceOnUse"
    x="0" y="0" width="6" height="6">
    <path d="M 0 0 6 0"
      style="stroke: black; fill: none;"/>
  </pattern>

  <pattern id="polkadot"
    patternUnits="userSpaceOnUse"
    x="0" y="0" width="36" height="36">
    <circle cx="12" cy="12" r="12"
      style="fill: url(#stripe);  stroke: black;"/>
  </pattern>
</defs>

<rect x="36" y="36" width="100" height="100"
    style="fill: url(#polkadot); stroke: black;"/>
```

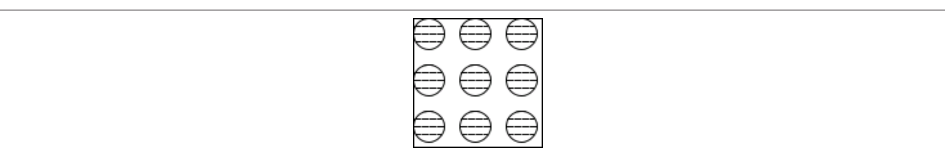


Figure 8-6. Patterns within patterns

Gradients

Rather than filling an object with a solid color, you can fill it with a *gradient*, a smooth color transition from one shade to another. Gradients can be *linear*, where the color transition occurs along a straight line, or *radial*, where the transition occurs as you radiate outward from a center point.

The linearGradient Element

A linear gradient is a transition through a series of colors along a straight line. You specify the colors you want at specific locations, called *gradient stops*. The stops are part of the structure of the gradient; the colors are part of the presentation. Example 8-7 shows the SVG for a gradient that fills a rectangle with a smooth transition from gold to cyan. The result is in Figure 8-7.

Example 8-7. Simple two-color gradient

http://oreillymedia.github.io/svg-essentials-examples/ch08/linear_gradient.html

```
<defs>
  <linearGradient id="two_hues">
    <stop offset="0%" style="stop-color: #ffcc00;"/>
    <stop offset="100%" style="stop-color: #0099cc;"/>
  </linearGradient>
</defs>

<rect x="20" y="20" width="200" height="100"
  style="fill: url(#two_hues);  stroke: black;"/>
```

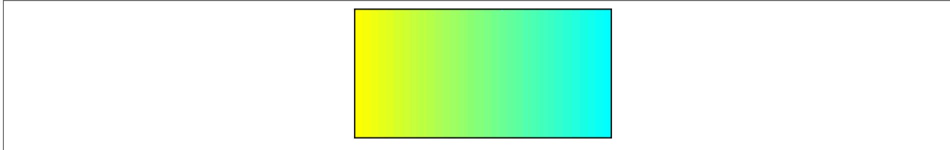


Figure 8-7. Simple two-color gradient

The <stop> element

Let's examine the `<stop>` element more closely. It has two required attributes: `offset` and `stop-color`. The `offset` tells the point along the line at which the color should be equal to the `stop-color`. The `offset` is expressed as a percentage from 0 to 100% or as a decimal value from 0 to 1.0. While you don't need to place stops at 0% and 100%, you usually will. The `stop-color` is specified here in a `style`, but you may also specify it as an attribute. Example 8-8 is a slightly more complex linear gradient, with stops for gold at 0%, reddish-purple at 33.3%, and light green at 100%. The result is shown in Figure 8-8.

Example 8-8. Three-color gradient

http://oreillymedia.github.io/svg-essentials-examples/ch08/three_stop_gradient.html

```
<defs>
  <linearGradient id="three_stops">
    <stop offset="0%" style="stop-color: #ffcc00;"/>
    <stop offset="33.3%" style="stop-color: #cc6699"/>
    <stop offset="100%" style="stop-color: #66cc99;"/>
  </linearGradient>
</defs>

<rect x="20" y="20" width="200" height="100"
    style="fill: url(#three_stops); stroke: black;"/>
```

Figure 8-8. Three-stop gradient

You can also use a `stop-opacity` attribute when specifying a stop color, with 1 being totally opaque and 0 being totally transparent. Example 8-9 creates a gradient that fades out dramatically up to the halfway point, then fades slightly toward the end. The result is shown in Figure 8-9.

Example 8-9. Three-opacity gradient

http://oreillymedia.github.io/svg-essentials-examples/ch08/stop_opacity.html

```
<defs>
  <linearGradient id="three_opacity_stops">
    <stop offset="0%" style="stop-color: #906; stop-opacity: 1.0"/>
    <stop offset="50%" style="stop-color: #906; stop-opacity: 0.3"/>
    <stop offset="100%" style="stop-color: #906; stop-opacity: 0.10"/>
  </linearGradient>
</defs>

<rect x="20" y="20" width="200" height="100"
    style="fill: url(#three_opacity_stops); stroke: black;"/>
```

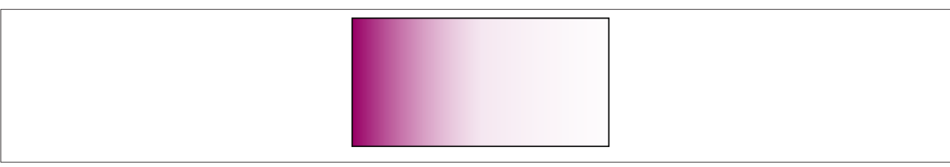


Figure 8-9. Gradient using stop-opacity

Establishing a transition line for a linear gradient

The default behavior of a linear gradient is to transition along a horizontal line from the left side of an object to its right side. If you want the transition of colors to occur across a vertical line or a line at an angle, you must specify the line's starting point with the `x1` and `y1` attributes and its ending points with the `x2` and `y2` attributes. By default, these are also expressed as percentages from 0% to 100% or decimals from 0 to 1. Example 8-10 uses the same color stops in a horizontal, vertical, and diagonal gradient. Rather than duplicate the stops into each `<linearGradient>` element, the example uses the `xlink:href` attribute to refer to the original left-to-right gradient. The stops will be inherited, but the *x*- and *y*-coordinates will be overridden by each individual gradient. The arrows in Figure 8-10 do not appear in the SVG of Example 8-10.

Example 8-10. Defining vectors for a linear gradient

http://oreillymedia.github.io/svg-essentials-examples/ch08/transition_line.html

```
<defs>
<linearGradient id="three_stops">
  <stop offset="0%" style="stop-color: #ffcc00;"/>
  <stop offset="33.3%" style="stop-color: #cc6699"/>
  <stop offset="100%" style="stop-color: #66cc99;"/>
</linearGradient>

<linearGradient id="right_to_left"
  xlink:href="#three_stops"
  x1="100%" y1="0%" x2="0%" y2="0%"/>

<linearGradient id="down"
  xlink:href="#three_stops"
  x1="0%" y1="0%" x2="0%" y2="100%"/>

<linearGradient id="up"
  xlink:href="#three_stops"
  x1="0%" y1="100%" x2="0%" y2="0%"/>

<linearGradient id="diagonal"
  xlink:href="#three_stops"
  x1="0%" y1="0%" x2="100%" y2="100%"/>
</defs>

<rect x="40" y="20" width="200" height="40"
```

```
  style="fill: url(#three_stops); stroke: black;"/>

<rect x="40" y="70" width="200" height="40"
  style="fill: url(#right_to_left); stroke: black;"/>

<rect x="250" y="20" width="40" height="200"
  style="fill: url(#down); stroke: black;"/>

<rect x="300" y="20" width="40" height="200"
  style="fill: url(#up); stroke: black;"/>

<rect x="40" y="120" width="200" height="100"
  style="fill: url(#diagonal); stroke: black;"/>
```

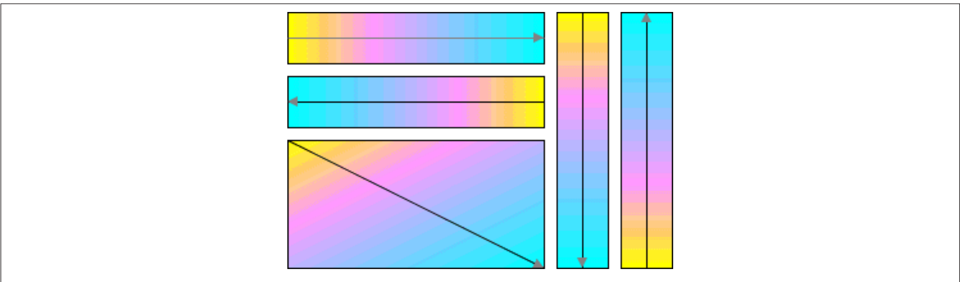


Figure 8-10. Defining vectors for a linear gradient

If you wish to establish the transition line using user space coordinates instead of percentages, set the `gradientUnits` to `userSpaceOnUse` instead of the default value, which is `objectBoundingBox`.

The spreadMethod attribute

The transition line does not have to go from one corner of an object to another. What happens if you say that the transition line goes from (20%,30%) to (40%,80%)? What happens to the part of the object outside that line? You can set the `spreadMethod` attribute to one of these values:

`pad`
: The beginning and ending stop colors will be extended to the edges of the object.

`repeat`
: The gradient will be repeated start-to-end until it reaches the edges of the object being filled.

`reflect`
: The gradient will be reflected end-to-start, start-to-end until it reaches the edges of the object being filled.

Figure 8-11 shows the leftmost square's gradient padded, the middle square's gradient repeated, and the right square's gradient reflected. The original transition line has been added to the SVG for each square in Example 8-11 to make the effect easier to detect.

Example 8-11. Effects of spreadMethod values on a linear gradient

http://oreillymedia.github.io/svg-essentials-examples/ch08/spread_method.html

```
<defs>
<linearGradient id="partial"
  x1="20%" y1="30%" x2="40%" y2="80%">
  <stop offset="0%" style="stop-color: #ffcc00;"/>
  <stop offset="33.3%" style="stop-color: #cc6699"/>
  <stop offset="100%" style="stop-color: #66cc99;"/>
</linearGradient>

<linearGradient id="padded"
  xlink:href="#partial"
  spreadMethod="pad"/>

<linearGradient id="repeated"
  xlink:href="#partial"
  spreadMethod="repeat"/>

<linearGradient id="reflected"
  xlink:href="#partial"
  spreadMethod="reflect"/>

<line id="show-line" x1="20" y1="30" x2="40" y2="80"
  style="stroke: white;"/>
</defs>

<rect x="20" y="20" width="100" height="100"
  style="fill: url(#padded); stroke: black;"/>
<use xlink:href="#show-line" transform="translate (20,20)"/>

<rect x="130" y="20" width="100" height="100"
  style="fill: url(#repeated); stroke: black;"/>
<use xlink:href="#show-line" transform="translate (130,20)"/>

<rect x="240" y="20" width="100" height="100"
  style="fill: url(#reflected); stroke: black;"/>
<use xlink:href="#show-line" transform="translate (240,20)"/>
```

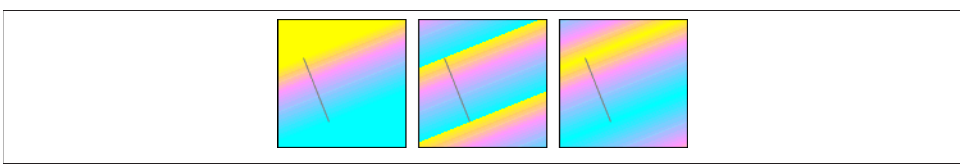


Figure 8-11. spreadMethod values pad, repeat, and reflect for a linear gradient

The radialGradient Element

The other type of gradient you can use is the radial gradient, where each color stop represents a circular path, radiating outward from a focus point.[1] It's set up in much the same way as a linear gradient. Example 8-12 sets three stops: orange, green, and purple. The result is shown in Figure 8-12.

Example 8-12. Radial gradient with three stops

http://oreillymedia.github.io/svg-essentials-examples/ch08/three_stop_radial.html

```
<defs>
  <radialGradient id="three_stops">
    <stop offset="0%" style="stop-color: #f96;"/>
    <stop offset="50%" style="stop-color: #9c9;"/>
    <stop offset="100%" style="stop-color: #906;"/>
  </radialGradient>
</defs>

<rect x="20" y="20" width="100" height="100"
  style="fill: url(#three_stops); stroke: black;"/>
```

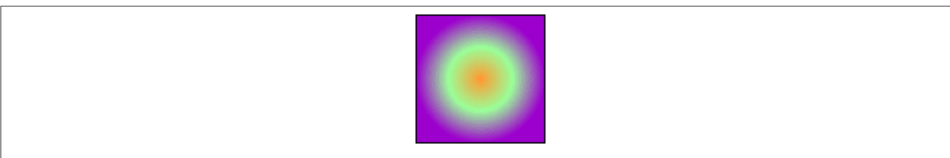


Figure 8-12. Radial gradient with three stops

Establishing transition limits for a radial gradient

Instead of using a line to determine where the 0% and 100% stop points should be, a radial gradient's limits are determined by a circle; the center is the 0% stop point, and the outer circumference defines the 100% stop point. You define the outer circle with the `cx` (center *x*), `cy` (center *y*), and `r` (radius) attributes. All of these are in terms of

1. If the bounding box of the object being filled is not square, the transition path will become elliptical to match the aspect ratio of the bounding box.

percentages of the object's bounding box. The default value for all these attributes is 50%. Example 8-13 draws a square with a radial gradient with the zero point centered at the upper left of the square and the outer edge at the lower right. The result is shown in Figure 8-13.

Example 8-13. Setting limits for a radial gradient

http://oreillymedia.github.io/svg-essentials-examples/ch08/radial_limits.html

```
<defs>
  <radialGradient id="center_origin"
    cx="0%" cy="0%" r="141%">
      <stop offset="0%" style="stop-color: #f96;"/>
      <stop offset="50%" style="stop-color: #9c9;"/>
      <stop offset="100%" style="stop-color: #906;"/>
  </radialGradient>
</defs>

<rect x="20" y="20" width="100" height="100"
  style="fill: url(#center_origin); stroke: black;"/>
```

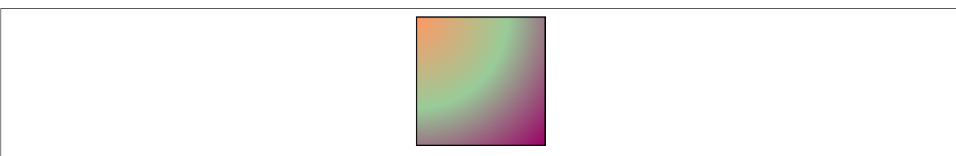
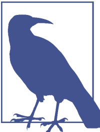


Figure 8-13. Setting limits for a radial gradient



In the preceding example, the `radialGradient`'s `r` was set to 141% instead of 100%. This is because the unit used to measure the radius is the average of the height and width of the object's bounding box, not the box's diagonal. The ratio of the diagonal to the side of a square is the square root of two, or 1.41.

The 0% stop point, also called the focal point, is by default placed at the center of the circle that defines the 100% stop point. If you wish to have the 0% stop point at some point other than the center of the limit circle, you must change the `fx` and `fy` attributes. The focal point should be within the circle established for the 100% stop point. If it's not, the SVG viewer program will automatically move the focal point to the outer circumference of the end circle.

In Example 8-14, the circle is centered at the origin with a radius of 100%, but the focal point is at (50%,50%). As you see in Figure 8-14, this has the visual effect of moving the "center."

Example 8-14. Setting focal point for a radial gradient

http://oreillymedia.github.io/svg-essentials-examples/ch08/radial_focus.html

```
<defs>
  <radialGradient id="focal_set"
    cx="0%" cy="0%" fx="50%" fy="50%" r="100%">
      <stop offset="0%" style="stop-color: #f96;"/>
      <stop offset="50%" style="stop-color: #9c9;"/>
      <stop offset="100%" style="stop-color: #906;"/>
  </radialGradient>
</defs>

<rect x="20" y="20" width="100" height="100"
  style="fill: url(#focal_set); stroke: black;"/>
```

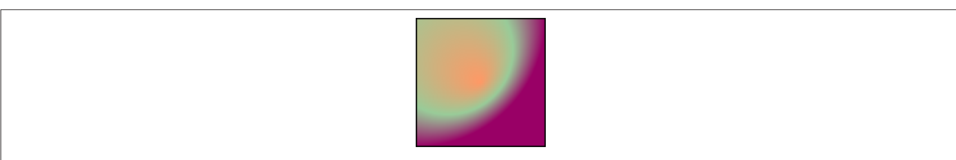
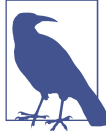


Figure 8-14. Setting focal point for a radial gradient

The default values for the limit-setting attributes of a `<radialGradient>` are as follows:

Attribute	Default value
`cx`	50% (horizontal center of object bounding box)
`cy`	50% (vertical center of object bounding box)
`r`	50% (half the width/height of object bounding box)
`fx`	Same as `cx`
`fy`	Same as `cy`



If you wish to establish the circle limits using user space coordinates instead of percentages, set the `gradientUnits` to `userSpaceOnUse` instead of the default value, which is `objectBoundingBox`.

The spreadMethod attribute for radial gradients

In the event that the limits you've described don't reach to the edges of the object, you can set the `spreadMethod` attribute to one of the values `pad`, `repeat`, or `reflect` as described earlier in "The spreadMethod attribute" on page 116 to fill up the remaining space as you wish. Example 8-15 has all three effects; Figure 8-15 shows the leftmost

square's gradient padded, the middle square's gradient repeated, and the right square's gradient reflected.

Example 8-15. Effects of spreadMethod values on a radial gradient

http://oreillymedia.github.io/svg-essentials-examples/ch08/radial_spread_method.html

```
<defs>
  <radialGradient id="three_stops"
    cx="0%" cy="0%" r="70%">
      <stop offset="0%" style="stop-color: #f96;"/>
      <stop offset="50%" style="stop-color: #9c9;"/>
      <stop offset="100%" style="stop-color: #906;"/>
  </radialGradient>

  <radialGradient id="padded" xlink:href="#three_stops"
    spreadMethod="pad"/>
  <radialGradient id="repeated" xlink:href="#three_stops"
    spreadMethod="repeat"/>
  <radialGradient id="reflected" xlink:href="#three_stops"
    spreadMethod="reflect"/>
</defs>

<rect x="20" y="20" width="100" height="100"
  style="fill: url(#padded); stroke: black;"/>
<rect x="130" y="20" width="100" height="100"
  style="fill: url(#repeated); stroke: black;"/>
<rect x="240" y="20" width="100" height="100"
  style="fill: url(#reflected); stroke: black;"/>
```

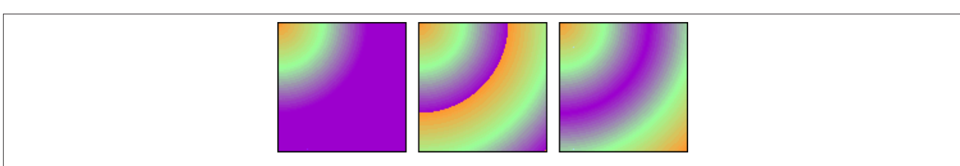


Figure 8-15. spreadMethod values pad, repeat, and reflect for a radial gradient

Gradient Reference Summary

Linear and radial gradients describe a smooth transition of colors used to fill an object. The object in question has a bounding box, defined as the smallest rectangle that entirely contains the object. The `<linearGradient>` and `<radialGradient>` elements are both containers for a series of `<stop>` elements. Each of these `<stop>` elements specifies a `stop-color`, an `offset`, and optionally a `stop-opacity`. For linear gradients, the offset is a percentage of the distance along the gradient's linear vector. For radial gradients, it is a percentage of the distance along the gradient's radius.

For a linear gradient, the starting point of the vector (which has the 0% stop color) is defined by the attributes `x1` and `y1`; the ending point (which has the 100% stop color) by the attributes `x2` and `y2`.

For a radial gradient, the focal point (which has the 0% stop color) is defined by the attributes `fx` and `fy`; the circle that has the 100% stop color is defined by its center coordinates `cx` and `cy` and its radius `r`.

If the `gradientUnits` attribute has the value `objectBoundingBox`, the coordinates are taken as a percentage of the bounding box's dimensions (this is the default). If the value is set to `userSpaceOnuse`, the coordinates are taken to be in the coordinate system used by the object being filled.

If the vector for a linear gradient or the circle for a radial gradient does not reach to the boundaries of the object being filled, the remaining space will be colored as determined by the value of the `spreadMethod` attribute: `pad`, the default, extends the start and end colors to the boundaries; `repeat` repeats the gradient start-to-end until it reaches the boundaries; and `reflect` replicates the gradient end-to-start and start-to-end until it reaches the object boundaries.

Transforming Patterns and Gradients

Sometimes you may need to skew, stretch, or rotate a pattern or gradient. You're not transforming the object being filled; you're transforming the pattern or the color spectrum used to fill the object. The `gradientTransform` and `patternTransform` attributes let you do just that, as written in Example 8-16 and shown in Figure 8-16.

Example 8-16. Transforming patterns and gradients

http://oreillymedia.github.io/svg-essentials-examples/ch08/pattern_gradient_transform.html

```
<defs>
  <pattern id="tile" x="0" y="0" width="20%" height="20%"
      patternUnits="objectBoundingBox">
  <path d="M 0 0 Q 5 20 10 10 T 20 20"
      style="stroke: black; fill: none;"/>
  <path d="M 0 0 h 20 v 20 h -20 z"
      style="stroke: gray; fill: none;"/>
  </pattern>

  <pattern id="skewed-tile"
    patternTransform="skewY(15)"
    xlink:href="#tile"/>

  <linearGradient id="plain">
    <stop offset="0%" style="stop-color: #ffcc00;"/>
    <stop offset="33.3%" style="stop-color: #cc6699"/>
```

```
    <stop offset="100%" style="stop-color: #66cc99;"/>
  </linearGradient>

  <linearGradient id="skewed-gradient"
    gradientTransform="skewX(10)"
    xlink:href="#plain"/>
</defs>

<rect x="20" y="10" width="100" height="100"
    style="fill: url(#tile); stroke: black;"/>
<rect x="135" y="10" width="100" height="100"
    style="fill: url(#skewed-tile); stroke: black;"/>

<rect x="20" y="120" width="200" height="50"
    style="fill: url(#plain); stroke: black;"/>
<rect x="20" y="190" width="200" height="50"
    style="fill: url(#skewed-gradient); stroke: black;"/>
```

Figure 8-16. Transformation of a pattern and gradient

One final note about gradients and patterns—although these examples have applied them to only the filled area of a shape, you may also apply them to the `stroke`. This lets you produce a multicolored or patterned outline for an object. You'll usually set the `stroke-width` to a number greater than 1 in order to make the effect more clearly visible.

The `objectBoundingBox` is based on the extent of the shape *before* adding the stroke. Because straight vertical and horizontal lines have a zero-width or zero-height bounding box, a gradient or pattern using `objectBoundingBox` units will be ignored when used as the stroke value for these lines. This means that the line won't be drawn at all unless you specify a fallback stroke value like `stroke: url(#rainbow) red;` in your style.

Fallback fill and stroke options are also a good idea if your pattern or gradient is defined in a separate file, in case that file cannot be loaded or the SVG viewer doesn't support external references.

CHAPTER 9

Text

While it may be true that every picture tells a story, it's perfectly all right to use words to help tell the story. Thus, SVG has several elements that let you add text to your graphics.

Text Terminology

Before investigating the primary method of adding text, the <text> element, we should define some terms you'll see if you read the SVG specification or if you work with text in any graphic environment:

Character
: A character, as far as an XML document is concerned, is a byte or bytes with a numeric value according to the Unicode standard. For example, what we call the letter *g* is the character with Unicode value 103.

Glyph
: A glyph is the visible representation of a character or characters. A single character can have many different glyphs to represent it. Figure 9-1 shows the word *glyphs* written with two different sets of glyphs—look particularly at the initial *g*—it's the same character, but the glyphs are markedly different.

glyphs glyphs

Figure 9-1. Two sets of glyphs

Multiple characters can reduce to a single glyph; some fonts have separate glyphs for the letter combinations *fl* and *ff* to make their spacing look better (these are called *ligatures*). Other times, a single character can be composed of multiple glyphs;

a print program might create the character *é* (which has Unicode value 233) by combining the *e* glyph with a nonspacing accent mark (´).

Font
: A font is a collection of glyphs representing a certain set of characters.

All the glyphs in a font will normally have the following characteristics in common:

Baseline, ascent, and descent
: All the glyphs in a font line up on the baseline. The distance from the baseline to the top of the tallest character in the font is the ascent; the distance from the baseline to the bottom of the deepest character is the descent. The total height of the character, which is the sum of the ascent and descent, is also called the em-height. The em-box is a square that has a width as large as an em-height.

Cap-height, ex-height
: The cap-height is the height of a capital letter above the baseline. The ex-height is, logically enough, the distance from the baseline to the top of a lowercase letter *x*. The ex-height is often a better measure of the subjective size (and readability) of a font than the em-height.

The baseline, ascent, and descent of a typical Roman-letter font are marked in Figure 9-2. The upper dotted line shows the cap-height, while the lower dotted line marks the ex-height.

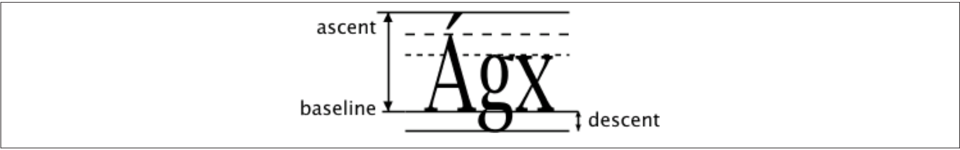


Figure 9-2. Glyph measurements

Simple Attributes and Properties of the <text> Element

The simplest form of the `<text>` element requires only two attributes, `x` and `y`, which define the point where the baseline of the first character of the element's content is placed. The default style for text, as with all objects, is to have a fill color of black and no outline. This, as it turns out, is precisely what you want for text. If you set the outline as well as the fill, the text looks uncomfortably thick. If you set only the outline, you can get a fairly pleasant set of outlined glyphs, especially if you lower the stroke width. Example 9-1 uses the placement and stroke/fill characteristics for `<text>`; the result is Figure 9-3.

Example 9-1. Text placement and outlining

```
<!-- guide lines -->
<path d="M 20 10, 20 120 M 10 30 100 30 M 10 70 100 70
   M 10 110 100 110" style="stroke: gray;"/>

<text x="20" y="30">Simplest Text</text>
<text x="20" y="70" style="stroke: black;">Outlined/filled</text>
<text x="20" y="110" style="stroke: black; stroke-width: 0.5;
    fill: none;">Outlined only</text>
```

Figure 9-3. Text placement and outlining

Many of the other properties that apply to text are the same as they are in the Cascading Style Sheets standard. Here are some of the CSS properties and values that are implemented in the Apache Batik viewer version 1.7. They also work in most (but not all) browsers:

font-family
: The value is a whitespace-separated list of font family names or generic family names. This is a list of fallback values; the SVG viewer will use the first family name it recognizes. The generic names must be the last in the list. The SVG viewer is required to recognize generic names and have fonts available for them. The generic family names are serif, sans-serif, monospace, fantasy, and cursive. Serif fonts have little "hooks" at the ends of the strokes; sans-serif fonts don't. In Figure 9-1, the word at the left is in a serif font and the word on the right is in a sans-serif font. Both serif and sans-serif fonts are proportional; the width of a capital M is not the same as the width of a capital I. A monospace font, which may or may not have serifs, is one where all the glyphs have the same width, like the letters of a typewriter. The fantasy and cursive default fonts can vary considerably from one browser or SVG viewer to another.

font-size
: The value is the baseline-to-baseline distance of glyphs if you were to have more than one line of text. (In SVG, you must position multiline <text> content yourself, so the concept is somewhat abstract.) If you use units on this attribute, as in style="font-size: 12pt", the size will be converted to user units before being rendered, so it can be affected by transformations and the SVG viewBox. If you use

relative units (`em`, `ex`, or percentages), they are calculated relative to the inherited font size.

`font-weight`
: The two most commonly used values of this property are `bold` and `normal`. You need the `normal` value in case you want to place nonbold text in a group that has been set to `style="font-weight: bold"`.

`font-style`
: The two most commonly used values of this property are `italic` and `normal`.

`text-decoration`
: Possible values of this property are `none`, `underline`, `overline`, and `line-through`.

`word-spacing`
: The value of this property is a length, either in explicit units such as `pt` or in user units. Make this a positive number to increase the space between words, set it to `normal` to keep normal space, or make it negative to tighten up the space between words. The length you specify is *added* to the normal spacing.

`letter-spacing`
: The value of this property is a length, either in explicit units such as `pt` or in user units. Make this a positive number to increase the space between individual letters, set it to `normal` to keep normal space, or make it negative to tighten up the space between letters. The length you specify is *added* to the normal spacing.

Example 9-2 uses these styles to produce Figure 9-4, with effects you'd expect from any competent text application.

Example 9-2. Text weight, style, decoration, and spacing attributes

```
<g style="font-size: 18pt">
<text x="20" y="20" style="font-weight:bold;">bold</text>
<text x="120" y="20" style="font-style:italic;">italic</text>
<text x="20" y="60" style="text-decoration:underline;">under</text>
<text x="120" y="60" style="text-decoration:overline;">over</text>
<text x="200" y="60" style="text-decoration:line-through;">through</text>
<text x="20" y="90" style="word-spacing: 10pt;">more word space</text>
<text x="20" y="120" style="word-spacing: -3pt;">less word space</text>
<text x="20" y="150" style="letter-spacing: 5pt;">wide letter space</text>
<text x="20" y="180"
      style="letter-spacing: -6pt;">narrow letter space</text>
</g>
```

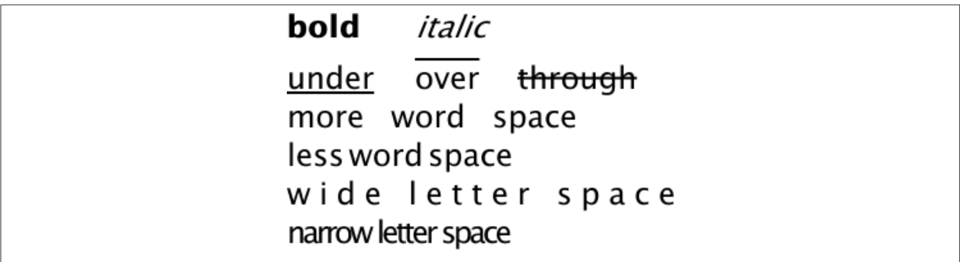


Figure 9-4. Text weight, style, decoration, and spacing

Text Alignment

The `<text>` element lets you specify the starting point, but you don't know, *a priori*, its ending point. This would make it difficult to center or right-align text, were it not for the `text-anchor` property. You set it to a value of `start`, `middle`, or `end`. For fonts that are drawn left-to-right, these are equivalent to left, center, and right alignment. For fonts that are drawn in other directions (see "Internationalization and Text" on page 134), these have a different effect. Example 9-3 shows three text strings, all starting at an *x*-location of 100, but with differing values of `text-anchor`. A guide line is drawn to show the effect more clearly in the result, Figure 9-5.

Example 9-3. Use of text-anchor

http://oreillymedia.github.io/svg-essentials-examples/ch09/text_alignment.html

```
<g style="font-size: 14pt;">
  <path d="M 100 10 100 100" style="stroke: gray; fill: none;"/>
  <text x="100" y="30" style="text-anchor: start">Start</text>
  <text x="100" y="60" style="text-anchor: middle">Middle</text>
  <text x="100" y="90" style="text-anchor: end">End</text>
</g>
```

Start
Middle
End

Figure 9-5. Result of using text-anchor

The <tspan> Element

Another consequence of not knowing a text string's length in advance is that it is difficult to construct a string with varying text attributes, such as this sentence, which switches among *italic*, normal, and **bold** text. If you had only the `<text>` element, you'd need to

experiment to find where each differently styled segment of text ended in order to space them properly. To solve this problem, SVG provides the `<tspan>`, or text span element. Analogous to the (X)HTML `<span>` element, `<tspan>` is a *tabula rasa* that may be embedded in text content, and upon which you may impose style changes. The `<tspan>` remembers the text position, so you don't have to. Thus, Example 9-4, which produces the display in Figure 9-6.

Example 9-4. Using tspan to change styles

```
<text x="10" y="30" style="font-size:12pt;">
  Switch among
  <tspan style="font-style:italic">italic</tspan>, normal,
  and <tspan style="font-weight:bold">bold</tspan> text.
</text>
```

Switch among *italic*, normal, and **bold** text.

Figure 9-6. Styles changed with tspan

In addition to changing presentation properties such as font size, color, weight, etc., you can also use attributes with `<tspan>` to change the positioning of individual letters or sets of letters. If, for example, you want superscripts or subscripts, you can use the `dy` attribute to offset characters within a span. The value you assign to this attribute is added to the vertical position of the characters, and continues to affect text even outside the span. Negative values are allowed. A similar attribute, `dx`, offsets characters horizontally. Example 9-5 uses vertical offsets to create the "falling letters" in Figure 9-7.

Example 9-5. Using dy to change vertical positioning within text

```
<text x="10" y="30" style="font-size:12pt;">
  F <tspan dy="4">a</tspan>
  <tspan dy="8">l</tspan>
  <tspan dy="12">l</tspan>
</text>
```

F a l l

Figure 9-7. Vertical positioning with dy

If you wish to express the offsets in absolute terms rather than relative terms, use the `x` and `y` attributes. This is handy for doing multiline runs of text. As you will see in "Whitespace and Text" on page 141, SVG never displays newline characters in text, so you need to manually reset the `x` value for each line, and use `y` or `dy` to position it vertically. You should always use `<tspan>`s within a `<text>` element to group related lines, not

only to allow them to be selected as a unit, but also because it adds structure to your document. Example 9-6 presents a verse of Edward Lear's *The Owl and the Pussycat* using `<tspan>` elements with absolute *x*-coordinates and a mix of `y` and `dy` values.

Example 9-6. Use of absolute positioning with tspan

```
<text x="10" y="30" style="font-size:12pt;">
  They dined on mince, and slices of quince,
  <tspan x="20" y="50">Which they ate with a
    runcible spoon;</tspan>
  <tspan x="10" y="70">And hand in hand, on the edge
    of the sand,</tspan>
  <tspan x="20" dy="20">They danced by the light of the moon.</tspan>
</text>
```

There's no visual evidence in Figure 9-8 that all the text is in one `<text>` element, but trust us—they're all connected.

They dined on mince, and slices of quince,
Which they ate with a runcible spoon;
And hand in hand, on the edge of the sand,
They danced by the light of the moon.

Figure 9-8. Absolutely positioned poetry

You may also rotate a letter or series of letters within a `<tspan>` by using the `rotate` attribute, whose value is an angle in degrees.

If you have to modify the positions of several characters, you can do it easily by specifying a series of numbers for any of the `x`, `y`, `dx`, `dy`, and `rotate` attributes. The numbers you specified will be applied, one after another, to the characters within the `<tspan>`. This is shown in Example 9-7.

Example 9-7. Use of multiple values for dx, dy, and rotate in a text span

```
<text x="30" y="30" style="font-size:14pt">It’s
<tspan dx="0 4 -3 5 -4 6" dy="0 -3 7 3 -2 -8"
  rotate="5 10 -5 -20 0 15">shaken</tspan>,
not stirred.
</text>
```

Notice in Figure 9-9 that the effects of `dx` and `dy` persist after the `<tspan>` ends. The text after the closing `</tspan>` is at the same offsets as the letter `n` in `shaken`. The text does not return to the baseline established by the first letter in the `<tspan>`.

It’s s h a k e n, not stirred.

Figure 9-9. Multiple horizontal and vertical offsets

Although you can use the `dy` attribute to produce superscripts and subscripts, it’s easier to use the `baseline-shift` style, as in Example 9-8. This style property has values of `super` and `sub`. You may also specify a length, such as `0.5em`, or a percentage, which is calculated in terms of the font size. `baseline-shift`’s effects are restricted to the span in which it occurs.

Example 9-8. Use of baseline-shift

```
<text x="20" y="25" style="font-size: 12pt;">
C<tspan style="baseline-shift: sub;">12</tspan>
H<tspan style="baseline-shift: sub;">22</tspan>
O<tspan style="baseline-shift: sub;">11</tspan> (sugar)
</text>

<text x="20" y="70" style="font-size: 12pt;">
6.02 x 10<tspan baseline-shift="super">23</tspan>
(Avogadro's number)
</text>
```

In Figure 9-10, the subscripted numbers appear too large. In an ideal case, you’d set the `font-size` as well, but we wanted this example to concentrate on only one concept.

$C_{12}H_{22}O_{11}$ (sugar)

6.02×10^{23}(Avogadro's number)

Figure 9-10. Subscripts and superscripts

Setting textLength

Although, as mentioned previously, there’s no *a priori* way to determine the endpoint of a segment of text, you can explicitly specify the length of text as the value of the `textLength` attribute. SVG will then fit the text into the given space. It does so by adjusting the space between glyphs and leaving the glyphs themselves untouched, or it can fit the words by adjusting both the spacing and glyph size. If you want to adjust space only, set the value of the `lengthAdjust` to `spacing` (this is the default). If you want SVG to fit the words into a given length by adjusting both spacing and glyph size, set `lengthAdjust` to `spacingAndGlyphs`. Example 9-9 uses these attributes to achieve the results of Figure 9-11.

Example 9-9. Use of textLength and lengthAdjust

http://oreillymedia.github.io/svg-essentials-examples/ch09/text_length.html

```
<g style="font-size: 14pt;">
<path d="M 20 10 20 70 M 220 10 220 70" style="stroke: gray;"/>
<text x="20" y="30"
  textLength="200" lengthAdjust="spacing">Two words</text>
<text x="20" y="60"
  textLength="200" lengthAdjust="spacingAndGlyphs">Two words</text>

<text x="20" y="90">Two words
  <tspan style="font-size: 10pt;">(normal length)</tspan></text>

<path d="M 20 100 20 170 M 100 100 100 170" style="stroke: gray;"/>
<text x="20" y="120"
  textLength="80" lengthAdjust="spacing">Two words</text>
<text x="20" y="160"
  textLength="80" lengthAdjust="spacingAndGlyphs">Two words</text>
</g>
```

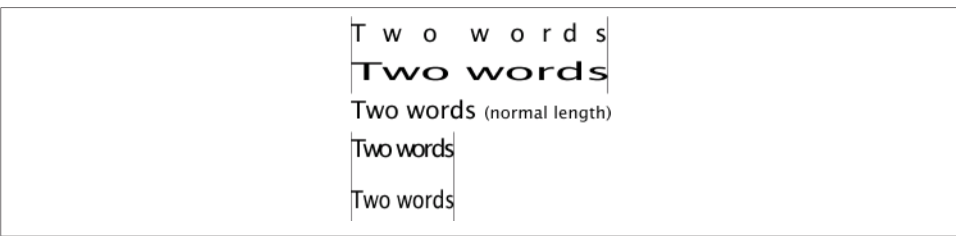


Figure 9-11. Effects of varying textLength and lengthAdjust

Vertical Text

When you use SVG to create charts, graphs, or tables, you will often want labels running down the vertical axes. One way to achieve vertically oriented text is to use a transformation to `rotate` the text 90 degrees. Another way to achieve the same effect is to change the value of the `writing-mode` style property to the value `tb` (meaning *t*op to *b*ottom).

Sometimes, though, you want the letters to appear in a vertical column with no rotation.

Example 9-10 does this by setting the `glyph-orientation-vertical` property with a value of 0. (Its default value is 90, which is what rotates top-to-bottom text 90 degrees.) In Figure 9-12, this setting tends to display the inter-letter spacing as unnaturally large. Setting a small negative value for `letter-spacing` solves this problem.

Example 9-10. Producing vertical text

```
<text x="10" y="20" transform="rotate(90,10,20)">Rotated 90</text>
<text x="50" y="20" style="writing-mode: tb;">Writing Mode tb</text>
```

```
<text x="90" y="20" style="writing-mode: tb;
   glyph-orientation-vertical: 0;">Vertical zero</text>
```

If you have been trying out these examples, you may have noticed that a number of features (such as `baseline-shift`, spacing, and vertical text) are poorly supported in some browsers. It's always a good idea to test out your designs in any SVG viewers that you want to support.

Rotated 90

Writing Mode tb

Vertical zero

Figure 9-12. Vertical text

Internationalization and Text

If your graphic has text that needs to be translated into multiple languages, SVG's support for Unicode and ability to display many languages in a single document will save you the trouble of creating separate documents for each language.

Unicode and Bidirectionality

XML is based on the Unicode standard (fully documented at the Unicode Consortium's website). This lets text display in any language the underlying viewer software can display, as you can see in Figure 9-13. Some languages such as Arabic and Hebrew are written right to left, so when text in these languages is mixed with text written left to right, as English is, the text is bidirectional, or *bidi* for short. The system software knows which characters go in which direction and works out their positions accordingly. Example 9-11 also overrides the implicit directionality of a segment of text by setting its `direction` style property to `rtl`, which stands for *r*ight-*t*o-*l*eft. If you wish to change the direction of Hebrew or Arabic text, set it to `ltr`, which is *l*eft-*t*o-*r*ight. You must also explicitly override the underlying Unicode bidirectionality algorithm by setting the `unicode-bidi` style property to `bidi-override`.

Example 9-11. International text using Unicode

```
<g style="font-size: 14pt;">

<text x="10" y="30">Greek: </text>
```

```
<text x="100" y="30">
  αβγδε
</text>

<text x="10" y="50">Russian:</text>
<text x="100" y="50">
  абвгд
</text>

<text x="10" y="70">Hebrew:</text>
<text x="100" y="70">
   אבגדה (written right to left)
</text>

<text x="10" y="90">Arabic:</text>
<text x="100" y="90">
  ا ب ج د (written right to left)
</text>

<text x="10" y="130">
  This is
    <tspan style="direction: rtl; unicode-bidi: bidi-override;
      font-weight: bold;">right-to-left</tspan>
  writing.
</text>
</g>
```

Greek: αβγδϵ
Russian: абвгд
Hebrew: אבגדה (written right to left)
Arabic: ابةت (written right to left)

This is **tfel–ot–thgir** writing.

Figure 9-13. Multilingual text

The <switch> Element

The ability to display multiple languages in a single document is useful for such things as a brochure for an event that receives international visitors. Sometimes, though, you would like to create one document with content in two languages—say, Spanish and Russian. People viewing the document with Spanish system software would see the Spanish text, and Russians would see Russian text.

SVG provides this capability with the `<switch>` element. This element searches through all its children until it finds one whose `systemLanguage` attribute has a value that

matches the language the user has chosen in the viewer software's preferences.[1] The value of systemLanguage is a single value or comma-separated list of language names. A language name is either a two-letter language code, such as ru for Russian, or a language code followed by a country code, which specifies a sublanguage. For instance, fr-CA denotes Canadian French, while fr-CH denotes Swiss French.

Once a matching child element is found, all its children will be displayed. All the other children of the <switch> will be bypassed. Example 9-12 shows text in UK English, US English, Spanish, and Russian. A match of language code alone is considered a match, and country codes are used only to "break a tie," so the text for UK English must come first.

Example 9-12. Use of the switch element

```
<circle cx="40" cy="60" r="20" style="fill: none; stroke: black;"/>
<g font-size="12pt">
  <switch>
    <g systemLanguage="en-UK">
      <text x="10" y="30">A circle</text>
      <text x="10" y="100">without colour.</text>
    </g>
    <g systemLanguage="en">
      <text x="10" y="30">A circle</text>
      <text x="10" y="100">without color.</text>
    </g>
    <g systemLanguage="es">
      <text x="10" y="30">Un círculo</text>
      <text x="10" y="100">sin color.</text>
    </g>
    <g systemLanguage="ru">
      <text x="10" y="30">Круг</text>
      <text x="10" y="100">без света.</text>
    </g>
  </switch>
</g>
```

Figure 9-14 is a combination of screenshots taken with the language set to each of the choices in Example 9-12. You should normally provide a fallback (a group without any systemLanguage attribute, as the last element in the <switch> block) to display *something* in case none of the languages match. Ideally, you would like to give users a way of selecting a language from the ones you have available.

1. The <switch> element can also be used for other tests; in "Foreign Objects in SVG" on page 20, we showed how to use a switch to test for support for specific features. If you use multiple test attributes on the children of a <switch> element, all of them must match for the content to be displayed.

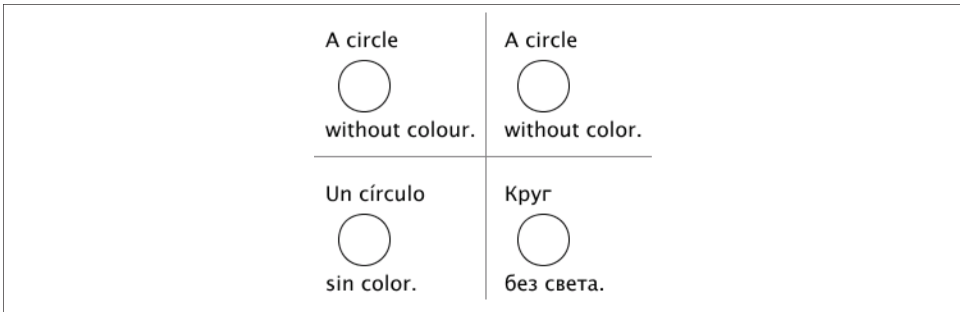


Figure 9-14. Combined screenshots as seen with different language preferences

Using a Custom Font

Sometimes you need special symbols that are not represented in Unicode, or you want a subset of the Unicode characters without having to install an entire font. An example is Figure 9-15, which needs only a few of the over 2,000 Korean syllables. You can create a custom font as described in Appendix E and give its starting `<font>` tag a unique `id`. Here is the relevant portion of a file containing six of the Korean syllables exported from the Batang TrueType font. The file is called *kfont.svg*:

```
<font id="kfont-defn" horiz-adv-x="989" vert-adv-y="1200"
  vert-origin-y="0">
  <font-face font-family="bakbatn"
    units-per-em="1000"
    panose-1="2 3 6 0 0 1 1 1 1 1"
    ascent="800" descent="-200" baseline="0" />
    <missing-glyph horiz-adv-x="500" />
    <!-- glyph definitions go here -->
  </font-face>
</font>
```

서울 – 대한민국

Figure 9-15. Korean syllables from an external font

Once that is done, Example 9-13 can reference the font in that external file. For the sake of consistency, the value of the `font-family` you use in this SVG file should match the value in the external file.

Example 9-13. Use of an external font

```
<defs>
  <font-face font-family="bakbatn">
    <font-face-src>
      <font-face-uri xlink:href="kfont.svg#kfont-defn">
        <font-face-format string="svg" />
```

```
      </font-face-uri>
    </font-face-src>
  </font-face>
</defs>

<text font-size="28" x="20" y="40"
  style="font-family: bakbatn, serif;">
    서울 - 대한민국
</text>
```

SVG fonts are currently not supported in Internet Explorer browsers (including version 11) or Firefox browsers (version 30). For these browsers, you can include a second `<font-face-src>` element, with the URI of an alternate font file in a different format. Alternatively, you can use a `<font-face-name>` element with the single attribute `name`, containing the name of a system font to use. All of these elements have the same interpretation as their equivalent CSS font-face properties.

If none of your specified fonts can be used, the browser will try to find any font on its system that can display the Unicode characters used in the text.

Text on a Path

Text does not have to go in a straight horizontal or vertical line. It can follow any arbitrary path; simply enclose the text in a `<textPath>` element that uses an `xlink:href` attribute to refer to a previously defined `<path>` element. Letters will be rotated to stand "perpendicular" to the curve (i.e., the letter's baseline will be tangent to the curve). Text along a gently curving and continuous path is easier to read than text that follows a sharply angled or discontinuous path.

Referencing a `<path>` within a `<textPath>` element does not automatically display that path. In Example 9-14, the `<path>`s are defined in a `<defs>` section, so they wouldn't normally be displayed. The example has `<use>` elements to draw the visible lines.

Example 9-14. Examples of textPath

http://oreillymedia.github.io/svg-essentials-examples/ch09/text_path.html

```
<defs>
<path id="curvepath"
    d="M30 40 C 50 10, 70 10, 120 40 S 150 0, 200 40"
    style="stroke: gray; fill: none;"/>

<path id="round-corner"
```

```
    d="M250 30 L 300 30 A 30 30 0 0 1 330 60 L 330 110"
    style="stroke: gray; fill: none;"/>

<path id="sharp-corner"
    d="M 30 110 100 110 100 160"
    style="stroke: gray; fill: none;"/>

<path id="discontinuous"
    d="M 150 110 A 40 30 0 1 0 230 110 M 250 110 270 140"
    style="stroke: gray; fill: none;"/>
</defs>

<g style="font-family: 'Liberation Sans';
  font-size: 10pt;">
  <use xlink:href="#curvepath"/>
  <text>
    <textPath xlink:href="#curvepath">
    Following a cubic Bézier curve.
    </textPath>
  </text>

  <use xlink:href="#round-corner"/>
  <text>
    <textPath xlink:href="#round-corner">
    Going 'round the bend
    </textPath>
  </text>

  <use xlink:href="#sharp-corner"/>
  <text>
    <textPath xlink:href="#sharp-corner">
    Making a quick turn
    </textPath>
  </text>

  <use xlink:href="#discontinuous"/>
  <text>
      <textPath xlink:href="#discontinuous">
      Text along a broken path
      </textPath>
  </text>
</g>
```

Example 9-14 produces Figure 9-16; Figure 9-17 shows you what it looks like if we draw the text without the underlying paths.

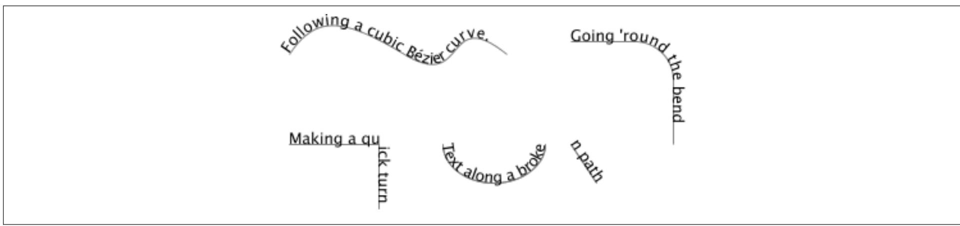


Figure 9-16. Text along a path (with paths shown)

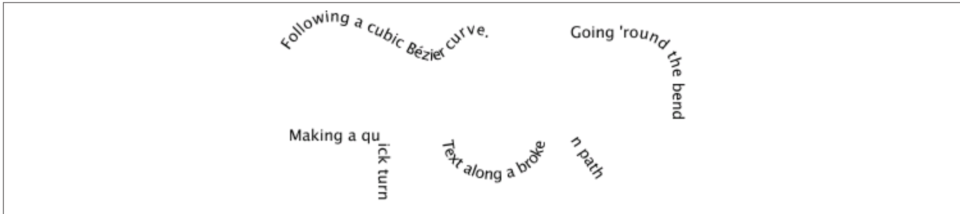


Figure 9-17. Text along a path (paths not shown)

You may adjust the beginning point of the text along its path by setting the `startOffset` attribute to a percentage or to a length. For example, `startOffset="25%"` will start the text one-fourth of the distance along the path, and `startOffset="30"` will start the text at a distance of 30 user units from the beginning of the path. If you wish to center text on a path, as in Example 9-15, set `textanchor="middle"` on the `<text>` element and `startOffset="50%"` on the `<textPath>` element. Text falling beyond the ends of the path will not be displayed, as shown in the left half of Figure 9-18.

Example 9-15. Text length and startOffset

http://oreillymedia.github.io/svg-essentials-examples/ch09/start_offset.html

```
<defs>
  <path id="short-corner" transform="translate(40,40)"
      d="M0 0 L 30 0 A 30 30 0 0 1 60 30 L 60 60"
    style="stroke: gray; fill: none;"/>

  <path id="long-corner" transform="translate(140,40)"
      d="M0 0 L 50 0 A 30 30 0 0 1 80 30 L 80 80"
    style="stroke: gray; fill: none;"/>
</defs>

<g style="font-family: 'Liberation Sans'; font-size: 12pt">
  <use xlink:href="#short-corner"/>
  <text>
    <textPath xlink:href="#short-corner">
      This text is too long for the path.
      </textPath>
```

```
  </text>

  <use xlink:href="#long-corner"/>
  <text style="text-anchor: middle;">
    <textPath xlink:href="#long-corner" startOffset="50%">
      centered
    </textPath>
  </text>
</g>
```

Figure 9-18. Effects of long text and startOffset

Whitespace and Text

You may change the way SVG handles whitespace (blanks, tabs, and newline characters) within text by changing the value of the `xml:space` attribute. If you specify a value of `default` (which, coincidentally, is the default value), SVG will handle whitespace as follows:

- Remove all newline characters
- Change all tabs to blanks
- Remove all leading and trailing blanks
- Change any run of intermediate blanks to a single blank

Thus, this string, where *t* represents a tab and *n* represents a newline, and an underscore represents a blank:

```
\n\n___abc_\t\t_def_\n\n__ghi
```

will render as:

```
abc_def_ghi
```

The other setting of `xml:space` is `preserve`. With this setting, SVG will simply convert all newline and tab characters to blanks, and then display the result, including leading and trailing blanks. The same text:

```
\n\n___abc_\t\t_def_\n\n__ghi
```

then renders as:

```
_____abc____def_____ghi
```

SVG's handling of whitespace is not like that of HTML. SVG's default handling eliminates all newlines; HTML changes internal newlines to a space. SVG's `preserve` method converts newlines to blanks; HTML's `<pre>` element does not. There is no newline in SVG 1.0; this bothers people until they realize that SVG text is oriented toward graphic display, not textual content (as in XHTML).

Case Study: Adding Text to a Graphic

Figure 9-19 adds Korean and English text to the Korean national symbol shown in Figure 7-6. The text is centered along an elliptical path. The additional SVG in Example 9-16 is shown in boldface.

Example 9-16. Text case study

```
<defs>
  <font-face font-family="bakbatn">
    <font-face-src>
      <font-face-uri xlink:href="kfont.svg#kfont-defn">
        <font-face-format string="svg" />
      </font-face-uri>
    </font-face-src>
  </font-face>

  <path id="upper-curve" d="M -8 154 A 162 130 0 1 1 316 154"/>
  <path id="lower-curve" d="M -21 154 A 175 140 0 1 0 329 154"/>
</defs>

<ellipse cx="154" cy="154" rx="150" ry="120" style="fill: #999999;"/>
<ellipse cx="152" cy="152" rx="150" ry="120" style="fill: #cceeff;"/>

<!--
  large light red semicircle fills upper half,
  followed by small light red semicircle that dips into
  lower-left half of symbol
-->
<path d="M 302 152 A 150 120, 0, 1, 0, 2 152
  A 75 60, 0, 1, 0, 152 152" style="fill: #ffcccc;"/>

<!--
  light blue semicircle rises
  into upper-right half of symbol
-->
<path d="M 152 152 A 75 60, 0, 1, 1, 302 152"
  style="fill: #cceeff;"/>

<text font-family="bakbatn, serif"
  style="font-size: 24pt; text-anchor: middle;">
  <textPath xlink:href="#upper-curve" startOffset="50%">
    서울 - 대한민국
```

```
  </textPath>
</text>

<text style="font-size: 14pt; text-anchor: middle;">
  <textPath xlink:href="#lower-curve" startOffset="50%">
  Seoul - Republic of Korea
  </textPath>
</text>
```

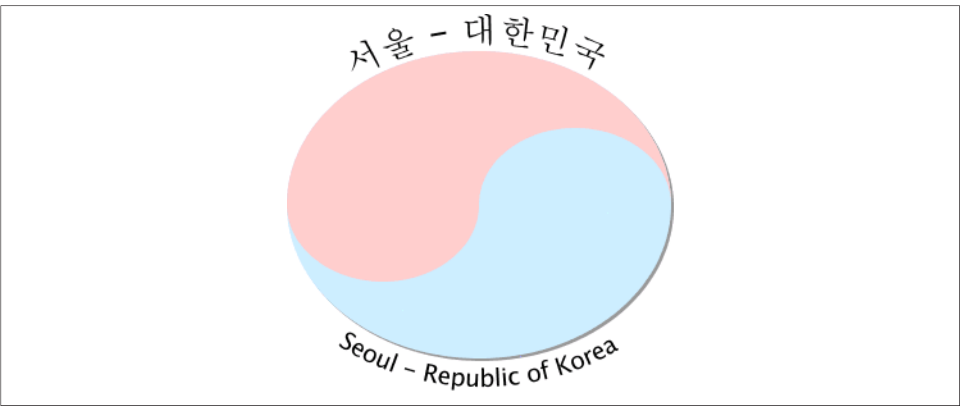


Figure 9-19. Text along path added to graphic

CHAPTER 10

Clipping and Masking

Sometimes you don't want to see an entire picture. For example, you might wish to draw a picture as though it were seen through binoculars or a keyhole; everything outside the boundary of the eyepieces or keyhole will be invisible. Or, you might want to set a mood by showing an image as though viewed through a translucent curtain. SVG accomplishes such effects with clipping and masking.

Clipping to a Path

When you create an SVG document, you establish its viewport by specifying the width and height of the area you're interested in. This becomes by default your clipping area; anything drawn outside these limits will not be displayed.[1] You can establish a clipping area of your own with the `<clipPath>` element.

Here's the simplest case: establishing a rectangular clip path. Inside the `<clipPath>` element will be the `<rect>` you want to clip to. The rectangle itself is not displayed; we only love it for its coordinates. Thus, you are free to add any fill or stroke styles you wish to the elements within the `<clipPath>`. On the object to be clipped, you add a `clip-path` style property whose value references the `<clipPath>` element. Note that the property is hyphenated and not capitalized; the element is capitalized and not hyphenated. In Example 10-1, the object being clipped is a small version of the cat picture from Chapter 1. The result is in Figure 10-1.

Example 10-1. Clipping to a rectangular path

http://oreillymedia.github.io/svg-essentials-examples/ch10/clip_path.html

```
<svg width="350" height="200" viewBox="0 0 350 200"
  xmlns="http://www.w3.org/2000/svg"
```

1. You can change this behavior by setting the `overflow` style property to `visible`.

```
  xmlns:xlink="http://www.w3.org/1999/xlink">
  <defs>
  <clipPath id="rectClip">
    <rect id="rect1" x="15" y="15"
      width="40" height="45"
      style="stroke: gray; fill: none;"/>
  </clipPath>
  </defs>

  <!-- clip to rectangle -->
  <use xlink:href="minicat.svg#cat"
    style="clip-path: url(#rectClip);"/>

  <!--
    for reference, show entire picture
    with clipping area outlined
  -->
  <g transform="translate(100,0)">
    <use xlink:href="#rect1"/>    <!-- show clip rectangle -->
    <use xlink:href="minicat.svg#cat"/>
  </g>
</svg>
```

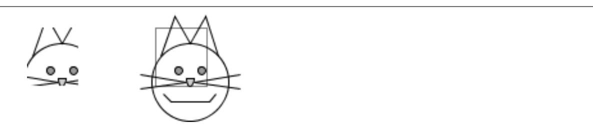


Figure 10-1. Simple rectangular clipping

As the name `<clipPath>` implies, you can clip to any arbitrary path. Indeed, the `<clipPath>` element can contain any number of basic shapes, `<path>` elements, or `<text>` elements. Example 10-2 shows a group of shapes clipped to a curved path and the same group of shapes clipped by text.

Example 10-2. Complex clip paths

http://oreillymedia.github.io/svg-essentials-examples/ch10/complex_clip_path.html

```
<defs>
  <clipPath id="curveClip">
    <path id="curve1"
      d="M5 55 C 25 5, 45 -25, 75 55, 85 85, 20 105, 40 55 Z"
      style="stroke: black; fill: none;"/>
  </clipPath>

  <clipPath id="textClip">
    <text id="text1" x="20" y="20" transform="rotate(60)"
      style="font-family: 'Liberation Sans';
        font-size: 48pt; stroke: black; fill: none;">
```

```
      CLIP
    </text>
  </clipPath>

  <g id="shapes">
    <rect x="0" y="50" width="90" height="60" style="fill: #999;"/>
    <circle cx="25" cy="25" r="25" style="fill: #666;"/>
    <polygon points="30 0 80 0 80 100" style="fill: #ccc;"/>
  </g>
</defs>

<!-- draw with curved clip-path -->
<use xlink:href="#shapes" style="clip-path: url(#curveClip);" />

<!-- draw with text as clip-path -->
<use transform="translate(100, 0)"
  xlink:href="#shapes" style="clip-path: url(#textClip);"/>

<g transform="translate(0, 150)">
  <use xlink:href="#shapes"/>
  <use xlink:href="#curve1"/>   <!-- show clip path -->
</g>

<g transform="translate(100,150)">
  <use xlink:href="#shapes"/>
  <use xlink:href="#text1"/>
</g>
```

To help you see the areas better, the preceding SVG draws the clipping path above the entire figure; you see this in the right half of Figure 10-2.

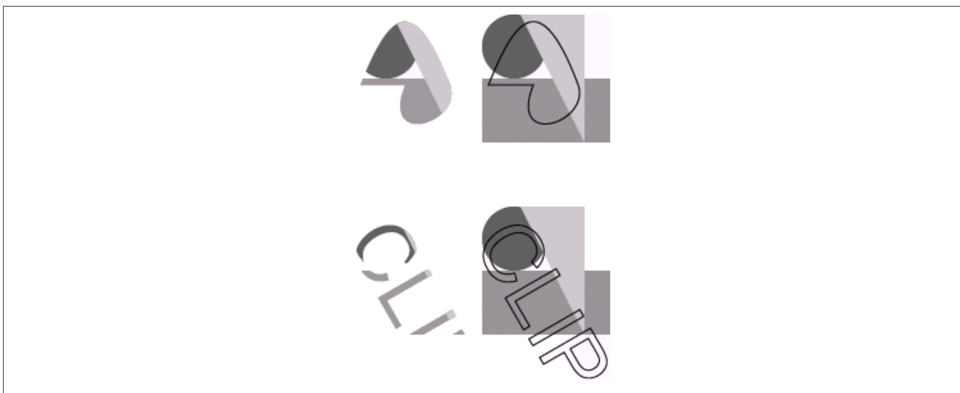


Figure 10-2. Complex path clipping

The coordinates for the preceding clip paths have been specified in user coordinates. If you wish to express coordinates in terms of the object bounding box, then set

clipPathUnits to objectBoundingBox (the default is userSpaceOnUse). Example 10-3 uses a clip path that will produce a circular (or oval) window on any object it's applied to.

Example 10-3. clipPathUnits using objectBoundingBox

```
<defs>
 <clipPath id="circularPath" clipPathUnits="objectBoundingBox">
   <circle cx="0.5" cy="0.5" r="0.5"/>
 </clipPath>

  <g id="shapes">
    <rect x="0" y="50" width="100" height="50" style="fill: #999;"/>
    <circle cx="25" cy="25" r="25" style="fill: #666;"/>
    <polygon points="30 0 80 0 80 100" style="fill: #ccc;"/>
  </g>

  <g id="words">
    <text  x="0"  y="19" style="font-family: 'Liberation Sans';
      font-size: 14pt;">
    <tspan x="0"  y="19">If you have form'd a circle</tspan>
    <tspan x="12" y="35">to go into,</tspan>
    <tspan x="0"  y="51">Go into it yourself</tspan>
    <tspan x="12" y="67">and see how you would do.</tspan>
    <tspan x="50" y="87">&#8212;William Blake</tspan>
    </text>
  </g>
</defs>

<use xlink:href="#shapes" style="clip-path: url(#circularPath);" />
<use xlink:href="#words" transform="translate(110,0)"
    style="clip-path: url(#circularPath);"/>
```

In Figure 10-3, the geometric figures happen to have a square bounding box, so the clipping appears circular. The text is bounded by a rectangular area, so the clipping area appears to be an oval.

For <marker>, <symbol>, and <svg> elements, which define their own viewport, you can also clip content to the viewport by using a style of overflow: hidden. However, if the content has a meet value for preserveAspectRatio, the viewport may be larger than the viewBox. To clip to the viewBox, create a <clipPath> element containing a rectangle that matches the minimum *x*, minimum *y*, width, and height of the viewBox.

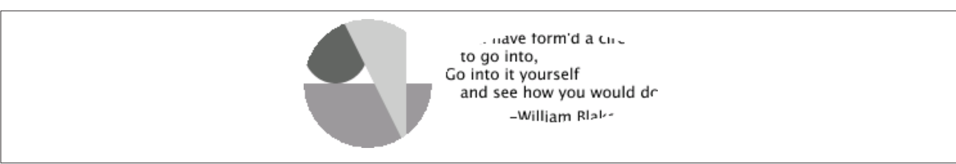


Figure 10-3. Use of a circular/oval clipping path

Masking

A mask in SVG is the exact opposite of the mask you wear to a costume party. With a costume party mask, the parts that are opaque hide your face; the parts that are translucent let people see your face dimly, and the holes (which are transparent) let people see your face clearly. An SVG mask, on the other hand, transfers its transparency to the object it masks. Where the mask is opaque, the pixels of the masked object are opaque. Where the mask is translucent, so is the object, and the transparent parts of the mask make the corresponding parts of the masked object invisible.

You use the <mask> element to create a mask. You may specify the mask's dimensions with the x, y, width, and height attributes. These dimensions are in terms of the masked objectBoundingBox. If you want the dimensions to be in terms of user space coordinates, set maskUnits to userSpaceOnUse.

Between the beginning <mask> and ending </mask> tags are any basic shapes, text, images, or paths you wish to use as the mask. The coordinates on these elements are expressed in user coordinate space by default. If you wish to use the object bounding box for the contents of the mask, set maskContentUnits to objectBoundingBox. (The default is userSpaceOnUse.)

The question then becomes: how does SVG determine the transparency, or alpha value, of the mask? We know each pixel is described by four values: its red, green, and blue color value, and its opacity. While at first glance it would seem logical to use only the opacity value, SVG decides to use all the information available to it rather than throwing away three-fourths of a pixel's information. SVG uses this formula:

```
(0.2125 * red value + 0.7154 * green value + 0.0721 * blue value) * opacity value
```

where all of the values are floating-point numbers in the range 0 to 1. You may be surprised that the proportions aren't equal, but if you look at fully saturated red, green, and blue, the green appears to be the brightest, red darker, and blue the darkest. (You can see this in Figure 10-4.) The darker the color, the smaller the resulting alpha value will be, and the less opaque the masked object will be.

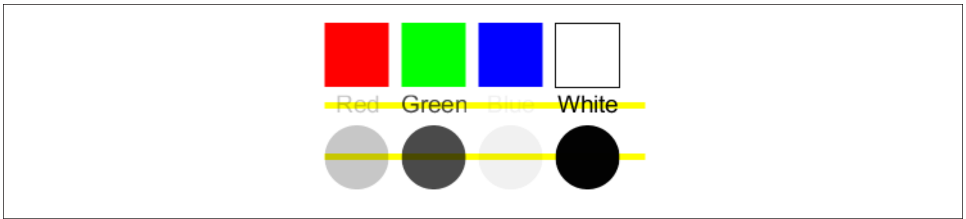


Figure 10-4. Effect of mask color values on transparency

Figure 10-4 was drawn with Example 10-4, which creates black text and a black circle masked by a totally opaque red, green, blue, and white square. The text and circle are grouped together, and the group uses a `mask` style property to reference the appropriate mask. The yellow horizontal bars in the background show you that, indeed, the text and circles are partially transparent.

Example 10-4. Masking with opaque colors

```
<defs>
  <mask id="redmask" x="0" y="0" width="1" height="1"
    maskContentUnits="objectBoundingBox">
    <rect x="0" y="0" width="1" height="1" style="fill: #f00;"/>
  </mask>

  <mask id="greenmask" x="0" y="0" width="1" height="1"
    maskContentUnits="objectBoundingBox">
    <rect x="0" y="0" width="1" height="1" style="fill: #0f0;"/>
  </mask>

  <mask id="bluemask" x="0" y="0" width="1" height="1"
    maskContentUnits="objectBoundingBox">
    <rect x="0" y="0" width="1" height="1" style="fill: #00f;"/>
  </mask>

  <mask id="whitemask" x="0" y="0" width="1" height="1"
    maskContentUnits="objectBoundingBox">
    <rect x="0" y="0" width="1" height="1" style="fill: #fff;"/>
  </mask>
</defs>

<!-- display the colors to show relative brightness (luminance) -->
<rect x="10" y="10" width="50" height="50" style="fill: #f00;"/>
<rect x="70" y="10" width="50" height="50" style="fill: #0f0;"/>
<rect x="130" y="10" width="50" height="50" style="fill: #00f;"/>
<rect x="190" y="10" width="50" height="50"
    style="fill: #fff; stroke: black;"/>

<!-- background content to show transparency -->
<rect x="10" y="72" width="250" height="5" style="fill: yellow"/>
<rect x="10" y="112" width="250" height="5" style="fill: yellow"/>
```

```
<g style="mask: url(#redmask);
    font-size: 14pt; text-anchor: middle;">
  <circle cx="35" cy="115" r="25"  style="fill: black;"/>
  <text x="35" y="80">Red</text>
</g>

<g style="mask: url(#greenmask);
    font-size: 14pt; text-anchor: middle;">
  <circle cx="95" cy="115" r="25" style="fill: black;"/>
  <text x="95" y="80">Green</text>
</g>

<g style="mask: url(#bluemask);
    font-size: 14pt; text-anchor: middle;">
  <circle cx="155" cy="115" r="25" style="fill: black;"/>
  <text x="155" y="80">Blue</text>
</g>

<g style="mask: url(#whitemask);
    font-size: 14pt; text-anchor: middle;">
  <circle cx="215" cy="115" r="25" style="fill: black;"/>
  <text x="215" y="80">White</text>
</g>
```

Figuring out the interaction between color, opacity, and final alpha value is not exactly intuitive. If you fill and/or stroke the mask contents in white, the "color factor" adds up to 1.0, and the opacity will then be the only factor that controls the mask's alpha value. Example 10-5 is written this way, and the result is in Figure 10-5.

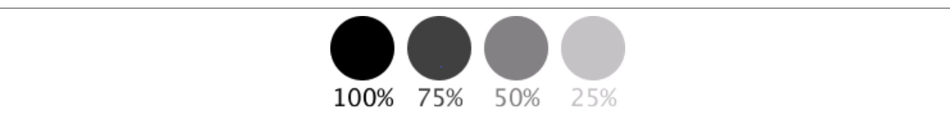


Figure 10-5. Alpha value equal to opacity

Example 10-5. Mask alpha using opacity only

http://oreillymedia.github.io/svg-essentials-examples/ch10/alpha_opacity_mask.html

```
<defs>
<mask id="fullmask" x="0" y="0" width="1" height="1"
    maskContentUnits="objectBoundingBox">
    <rect x="0" y="0" width="1" height="1"
        style="fill-opacity: 1.0; fill: white;"/>
</mask>

<mask id="three-fourths" x="0" y="0" width="1" height="1"
    maskContentUnits="objectBoundingBox">
    <rect x="0" y="0" width="1" height="1"
        style="fill-opacity: 0.75; fill: white;"/>
```

```
</mask>

<mask id="one-half" x="0" y="0" width="1" height="1"
    maskContentUnits="objectBoundingBox">
    <rect x="0" y="0" width="1" height="1"
        style="fill-opacity: 0.5; fill: white;"/>
</mask>

<mask id="one-fourth" x="0" y="0" width="1" height="1"
    maskContentUnits="objectBoundingBox">
    <rect x="0" y="0" width="1" height="1"
        style="fill-opacity: 0.25; fill: white;"/>
</mask>
</defs>

<g style="font-size: 14pt; text-anchor:middle; fill:black;">
    <g style="mask: url(#fullmask);">
    <circle cx="35" cy="35" r="25"/>
    <text x="35" y="80">100%</text>
    </g>

    <g style="mask: url(#three-fourths);">
    <circle cx="95" cy="35" r="25"/>
    <text x="95" y="80">75%</text>
    </g>

    <g style="mask: url(#one-half);">
    <circle cx="155" cy="35" r="25"/>
    <text x="155" y="80">50%</text>
    </g>

    <g style="mask: url(#one-fourth);">
    <circle cx="215" cy="35" r="25"/>
    <text x="215" y="80">25%</text>
    </g>
</g>
```

Case Study: Masking a Graphic

Example 10-6 adds a JPG image to the graphic that was constructed in "Case Study: Adding Text to a Graphic" on page 142. As you can see in Figure 10-6 (reduced to save space), the image obscures the curve inside the main ellipse, and the blue sky intrudes horribly on the pale red section.

Example 10-6. Unmasked <image> inside a graphic

```
<defs>
  <font-face font-family="bakbatn">
    <font-face-src>
      <font-face-uri xlink:href="kfont.svg#kfont-defn"/>
    </font-face-src>
```

```
  </font-face>
</defs>

<!-- draws ellipse and text -->
<use xlink:href="ksymbol.svg#ksymbol"/>

<image xlink:href="kwanghwamun.jpg" x="72" y="92"
    width="160" height="120"/>
```

Figure 10-6. Unmasked <image> inside a graphic

The solution is to fade out the edges of the picture, which is easily done by using a radial gradient as a mask. Here's the code to be added to the `<defs>` section of the document:

```
<radialGradient id="fade">
  <stop offset="0%" style="stop-color: white; stop-opacity: 1.0;"/>
  <stop offset="85%" style="stop-color: white; stop-opacity: 0.5;"/>
  <stop offset="100%" style="stop-color: white; stop-opacity: 0.0;"/>
</radialGradient>
<mask id="fademask" maskContentUnits="objectBoundingBox">
  <rect x="0" y="0" width="1" height="1"
    style="fill: url(#fade);"/>
</mask>
```

Then add a mask reference to the `<image>` tag, resulting in Figure 10-7:

```
<image xlink:href="kwanghwamun.jpg" x="72" y="92"
  width="160" height="120"
  style="mask: url(#fademask);"/>
```

Figure 10-7. Masked image

Using less of the picture can substantially improve the graphic as a whole.

CHAPTER 11
Filters

The preceding chapters have given you a basis for creating graphics that convey information with great precision and detail. If you're going on a spring picnic, you want a precise map. When you look in the newspaper for the graphics that describe the weather forecast, you want "just the facts."

If you're asked later to describe the day of the picnic, nobody wants a crisp recitation of meteorological statistics. Similarly, nobody wants to see a graphic of a spring flower composed of pure vectors; Figure 11-1 fails totally to convey any warmth or charm.

Figure 11-1. Flower composed of plain vectors

Graphics are often designed to evoke feelings or moods as much as they are meant to convey information. Artists who work with bitmap graphics—and therefore work with the *appearance* of an object instead of its geometrical definition—have many tools at their disposal to add such effects. They can produce blurred shadows, selectively thicken or thin lines, add textures to part of the drawing, or make an object appear to be embossed or beveled.

How Filters Work

Although SVG is not a bitmap description language, it still lets you use some of these same tools. When an SVG viewer program processes a graphic object, it will render the

object to some bitmapped output device; at some point, the program will convert the object's description into the appropriate set of pixels that appear on the output device. Now let's say that you use the SVG <filter> element to specify a set of operations (also called *primitives*) that display an object with a blurred shadow offset slightly to the side, and attach that filter to an object:

```
<filter id="drop-shadow">
  <!-- filter operations go here -->
</filter>

<g id="spring-flower"
  style="filter: url(#drop-shadow);"/>
  <!-- drawing of flower goes here -->
</g>
```

Because the flower uses a filter in its presentation style, SVG will not render the flower directly to the final graphic. Instead, SVG will render the flower's pixels into a temporary bitmap. The operations specified by the filter will be applied to that temporary area and their result will be rendered into the final graphic.

The size of the temporary bitmap will by default depend on the resolution and size of the display screen on which the image will be rendered. This means that some filter effects can have different appearances at different sizes, even if all the SVG code is the same. The specifications define attributes to control the effective resolution of the filter effects, but these are not consistently implemented in SVG viewers and are not discussed here.

Creating a Drop Shadow

Example 5-8 created a drop shadow by offsetting a gray ellipse underneath a colored ellipse. It worked, but it wasn't elegant. Let's investigate a way to create a better-looking drop shadow with a filter.

Establishing the Filter's Bounds

The <filter> element has attributes that describe the clipping region for a filter. You specify an `x`, `y`, `width`, and `height` in terms of the percentage of the filtered object's bounding box. (That is the default.) Any portion of the resulting output that's outside the bounds will not be displayed. If you are intending to apply a filter to many objects, you may want to omit these attributes altogether and take the default values of `x` equal to –10%, `y` equal to –10%, `width` equal to 120%, and `height` equal to 120%. This gives extra space for filters—such as the drop shadow that we're constructing—that produce output larger than their input.

These attributes are in terms of the filtered object's bounding box; specifically, `filterUnits` has a value of `objectBoundingBox` by default. If you wish to specify boundaries in user units, then set the attribute's value to `userSpaceOnUse`.

You can also specify the units used in the filter primitives with the `primitiveUnits` attribute. Its default value is `userSpaceOnUse`, but if you set it to `objectBoundingBox`, then you can express units as a percent of the graphic's size.

Using <feGaussianBlur> for a Drop Shadow

Between the beginning and ending `<filter>` tags are the filter primitives that perform the operations you desire. Each primitive has one or more inputs, and exactly one output. An input can be the original graphic, specified as `SourceGraphic`, the alpha (opaqueness) channel of the graphic, specified as `SourceAlpha`, or the output of a previous filtering primitive. The alpha source is useful when you're interested in only the shape of the graphic, regardless of color; it avoids the interactions of alpha and color, as described in Chapter 10, in "Masking" on page 149.

Example 11-1 is a first attempt to produce a drop shadow on the flower, using the `<feGaussianBlur>` filter primitive. We specify `SourceAlpha` as its input (the `in` attribute), and the amount of blur with the `stdDeviation` attribute. The larger this number, the greater the blur. If you give two numbers separated by whitespace as the value for `stdDeviation`, the first number is taken as the blur in the *x*-direction and the second as the blur in the *y*-direction.

Example 11-1. First attempt to produce a drop shadow

```
<defs>
  <filter id="drop-shadow">
      <feGaussianBlur in="SourceAlpha" stdDeviation="2"/>
  </filter>
</defs>

<g id="flower" filter="url(#drop-shadow)">
    <!-- drawing here -->
</g>
```

Figure 11-2 shows the result, which is probably not what you thought it would be.

Don't be surprised; remember, the filter returns the output, which is a blurred alpha channel, *instead* of the original source graphic. You could get the effect you want by putting the flower within the `<defs>` section of the document and changing the SVG to read as follows:

```
<use xlink:href="#flower" filter="url(#drop-shadow)"
    transform="translate(4, 4)"/>
<use xlink:href="#flower"/>
```

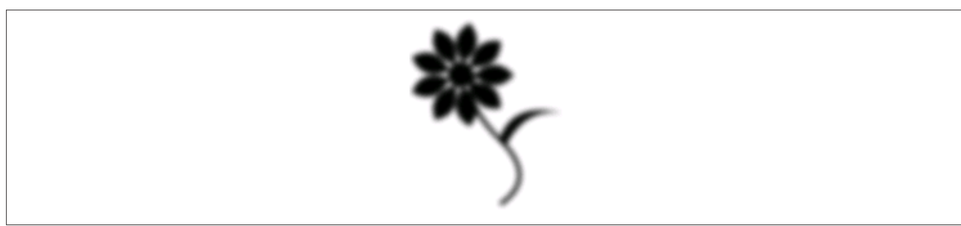


Figure 11-2. Result of first attempt at a drop shadow

However, that would require SVG to execute all the elements that make up the flower twice. Instead, the solution is to add more filter primitives, so that all the work can be handled during rendering.

Storing, Chaining, and Merging Filter Results

Example 11-2 is the updated filter.

Example 11-2. Improved drop shadow filter

```
<filter id="drop-shadow">
  <feGaussianBlur in="SourceAlpha" stdDeviation="2" result="blur">  ❶
  <feOffset in="blur" dx="4" dy="4" result="offsetBlur">  ❷
  <feMerge>    ❸
        <feMergeNode in="offsetBlur">
        <feMergeNode in="SourceGraphic"/>
  </feMerge>
</filter>
```

❶ The `result` attribute specifies that the result of this primitive can be referenced later by the name `blur`. This isn't like an XML `id`; the name you give is a local name that's valid only for the duration of the primitives contained in the current `<filter>`.

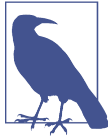
❷ The `<feOffset>` primitive takes its input, in this case the `blur` result from the Gaussian blur, offsets it by the specified `dx` and `dy` values, and stores the resulting bitmap under the name `offsetBlur`.

❸ The `<feMerge>` primitive encloses a list of `<feMergeNode>` elements, each of which specifies an input. The inputs are stacked one on top of another in the order that they appear. In this case, you want the `offsetBlur` below the original `SourceGraphic`.

You can now refer to this improved drop shadow filter sequence when drawing the flower, producing a surprisingly pleasant image in Figure 11-3.

```
<g id="flower" filter="url(#drop-shadow)">
    <!-- drawing here -->
</g>
```

Figure 11-3. Result of improved drop shadow



When you first start working with filters, we strongly recommend that you do things in stages, testing filters one at a time. I created large numbers of stunningly ugly results during botched attempts to discover how a filter really works. You probably will too. We'll just keep it as our little secret.

Similarly, when you first learn about filters, you will be tempted to apply as many of them as possible to a drawing, just to see what will happen. Your purpose is experimentation, so go ahead. Once you finish experimenting and begin production work, the purpose of the filter changes. Filters should support and enhance your message, not overwhelm it. Judicious use of one or two filters is a buoy; a flotilla of filters almost always sinks the message.

Creating a Glowing Shadow

The drop shadow works well on the flower, but looks totally unimpressive when applied to text, as you can see in Figure 11-4.

Figure 11-4. Drop shadow applied to text

The text would look better with a glowing turquoise area surrounding it, and you can create this effect with the <feColorMatrix> primitive to change black to a different color.

The <feColorMatrix> Element

The `<feColorMatrix>` element allows you to change color values in a very generalized way. The sequence of primitives used to create a glowing turquoise shadow is shown in Example 11-3.

Example 11-3. Glow filter

```
<filter id="glow">
  <feColorMatrix type="matrix"
    values=
      "0 0 0 0   0
       0 0 0 0.9 0
       0 0 0 0.9 0
       0 0 0 1   0"/>
  <feGaussianBlur stdDeviation="2.5"
    result="coloredBlur"/>
  <feMerge>
    <feMergeNode in="coloredBlur"/>
    <feMergeNode in="SourceGraphic"/>
  </feMerge>
</filter>
```

The `<feColorMatrix>` is a very versatile primitive, allowing you to modify any of the color or alpha values of a pixel. When the `type` attribute equals `matrix`, you must set the `value` to a series of 20 numbers describing the transformation. The 20 numbers are best understood when written in four rows of five columns each. Each row represents an algebra equation defining how to calculate the output R, G, B, or A value (in order by row). The numbers in the row are multiplied by the input pixel's value for R, G, B, and A and the constant 1 (in order by column), and then added together to get the output value. To set up a transformation that paints all opaque areas the same color, you can ignore the input colors and the constant, and just set the values in the alpha column. The model for such a matrix looks like this:

```
values=
  "0 0 0 red 0
   0 0 0 green 0
   0 0 0 blue 0
   0 0 0 1 0"
```

where the red, green, and blue values are decimal numbers that usually range from 0 to 1. In Example 11-3, red is set to to 0, and the green and blue values are set to 0.9, which will produce a bright cyan color.

You'll note that the example does not have an `in` attribute for the input to this primitive; the default is to use the `SourceGraphic`. There is also no `result` attribute on this primitive. This means that the color matrix operation's output is available only as the implicit input to the next filter primitive. If you use this shortcut, then the next filter primitive must *not* have an `in` attribute.

In the example, the result of `<feColorMatrix>` is a cyan-colored source. The rest of the filter uses a Gaussian blur to spread it out; the resulting cyan-colored blur is stored for future reference as `coloredBlur`. Finally, `<feMerge>` produces the output glow underneath the object in question.

With these two filters, you can create the new, improved Figure 11-5 with SVG like this:

```
<g id="flower" style="filter: url(#drop-shadow);">
  <!-- draw the flower -->
</g>
<text x="120" y="50"
  style="filter: url(#glow); fill: #003333; font-size:18;"">
Spring <tspan x="120" y="70">Flower</tspan>
</text>
```

Figure 11-5. Drop shadow and glowing text

More About the <feColorMatrix> Element

The preceding example used the most general kind of color matrix, where you get to specify any values you wish. There are three other values for the `type` attribute. Each of these "built-in" color matrices accomplishes a particular visual task and has its own way of specifying `values`:

`hueRotate`
: The `values` is a single number that tells how many degrees the color values should be rotated. The mathematics used to accomplish this are very similar to those used in the `rotate` transformation as described in "The rotate Transformation" on page 78 in Chapter 6. The relation between rotation and resulting color is not at all obvious, as shown in Figure 11-6; you may want to experiment with the online version:

  http://oreillymedia.github.io/svg-essentials-examples/ch11/hue_rotate.html

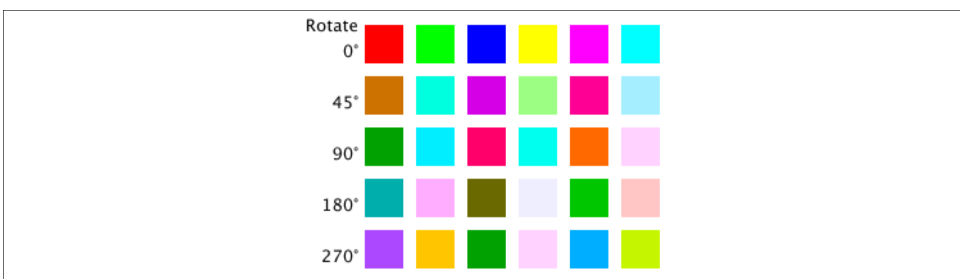


Figure 11-6. Result of hueRotate on fully saturated colors

`saturate`

The `values` attribute specifies a single number in the range 0 to 1. The smaller the number, the more "washed out" the colors will be, as you see in Figure 11-7. A value of 0 converts the graphic to black and white. The filter can be used only to desaturate (wash out) an image; you cannot increase saturation (to make a normal image technicolor) with this method.

Test out other saturation values with the online example:

http://oreillymedia.github.io/svg-essentials-examples/ch11/saturate.html

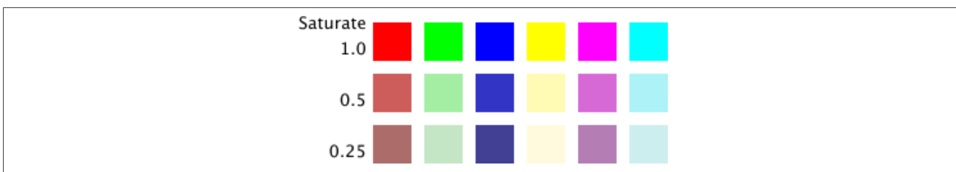


Figure 11-7. Result of saturate on primary and secondary colors

`luminanceToAlpha`

This filter creates an alpha channel based upon a color's luminance. The luminance is the inherent "brightness" of a color, as described in "Masking" on page 149 in Chapter 10. In Figure 11-8, the luminance of the colored squares is used as an alpha channel for solid black squares. The filter discards the color of the original squares, with the result being solid black with varying opacity levels. The lighter a color, the less the transparency it confers upon the filtered object. The `values` attribute is ignored for this `type`.

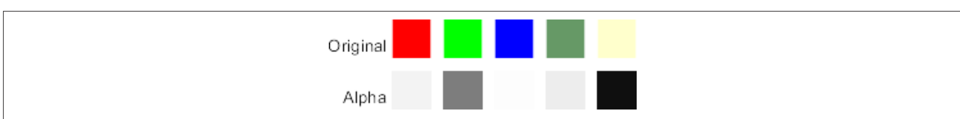


Figure 11-8. Result of luminanceToAlpha

The <feImage> Filter

Up to this point, you have seen only the original graphic or its alpha channel as input to a filter. SVG's `<feImage>` element lets you use any JPG, PNG, or SVG file—or an SVG element with an `id` attribute—as input to a filter. Example 11-4 imports a picture of the sky with a cloud in it to use as a background for the picture of the flower.

Example 11-4. Using the feImage element

```
<defs>
<filter id="sky-shadow" filterUnits="objectBoundingBox">
    <feImage xlink:href="sky.jpg" result="sky"
         x="0" y="0" width="100%" height="100%"
         preserveAspectRatio="none"/>
    <feGaussianBlur in="SourceAlpha" stdDeviation="2" result="blur"/>
    <feOffset in="blur" dx="4" dy="4" result="offsetBlur"/>
    <feMerge>
        <feMergeNode in="sky"/>
        <feMergeNode in="offsetBlur"/>
        <feMergeNode in="SourceGraphic"/>
    </feMerge>
</filter>
</defs>

<g id="flower" style="filter: url(#sky-shadow)">
    <!-- flower graphic goes here -->
</g>

<!-- show original image -->
<image xlink:href="sky.jpg" x="170" y="10"
    width="122" height="104"/>
```

Figure 11-9 shows the result, with the original picture of the sky shown at right at its true size. The image is stretched to fit the filter region defined on the `<filter>` element by default (there are no dimensions on the filter, so the default filter region is 10% padding around the object bounding box). You can set explicit height, width, and x/y offsets on the `<feImage>` element. By default, these measurements are in `userSpaceOnUse` units; however, any percentage values are calculated relative to the filter region. You can use the `primitiveUnits` attribute on the `<filter>` element to switch to `objectBoundingBox` units, but this will affect all the elements in the filter.

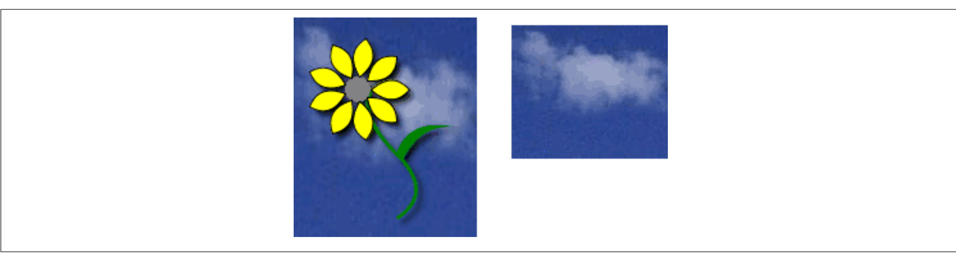
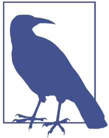


Figure 11-9. Result of feImage



In addition to importing a complete image file, you can use a URL fragment in the `xlink:href` attribute to import part of an SVG graphic—from a file or from elsewhere in the same SVG—into your filter. Unfortunately, image fragments in `<feImage>` are not yet supported in Mozilla Firefox. You can monitor Bugzilla bug 455986 to see if that changes.

The <feComponentTransfer> Filter

The problem with the background is that it is too dark. Using `saturate` isn't the answer; it changes the intensity of the color, but maintains its brightness (luminance). To lighten the image, you need to increase the value of each color channel. You could do this with a custom color matrix, but `<feComponentTransfer>` provides an easier and more flexible way to manipulate each channel (component) separately. It also lets you adjust each color channel differently, so you can make the blue sky both lighter *and* less intense by increasing the level of green and red more than the blue level.

You adjust the levels of red, green, blue, and alpha by placing a `<feFuncR>`, `<feFuncG>`, `<feFuncB>`, and `<feFuncA>` element inside `<feComponentTransfer>`. Each of these subelements may independently specify a `type` attribute telling how that particular channel is to be modified.

To simulate the effect of a brightness control, you specify the `linear` function, which places the current color value `C` into the formula *`slope`* `* C +` *`intercept`*. The *`intercept`* provides a "base value" for the result; the *`slope`* is a simple scaling factor. Example 11-5 uses a filter that adds a brightened sky to the flower with the drop shadow. Note that the red and green channels are adjusted differently than the blue channel. This dramatically brightens the sky in Figure 11-10.

Example 11-5. Changing brightness with feComponentTransfer

http://oreillymedia.github.io/svg-essentials-examples/ch11/linear_transfer.html

```
<filter id="brightness-shadow" filterUnits="objectBoundingBox">
  <feImage xlink:href="sky.jpg" result="sky"/>
  <feComponentTransfer in="sky" result="sky">
    <feFuncB type="linear" slope="3" intercept="0"/>
    <feFuncR type="linear" slope="1.5" intercept="0.2"/>
    <feFuncG type="linear" slope="1.5" intercept="0.2"/>
  </feComponentTransfer>
  <feGaussianBlur in="SourceAlpha" stdDeviation="2" result="blur"/>
  <feOffset in="blur" dx="4" dy="4" result="offsetBlur"/>
  <feMerge>
    <feMergeNode in="sky"/>
    <feMergeNode in="offsetBlur"/>
    <feMergeNode in="SourceGraphic"/>
  </feMerge>
</filter>
```

Figure 11-10. Result of linear component transfer

A simple linear adjustment will add and multiply the same amount to every color value within a channel. This is not the case with the `gamma` function, which places the current color value `C` into the formula *`amplitude`* `*` `C`^{exponent} `+` *`offset`*. The *`offset`* provides a "base value" for the result; the *`amplitude`* is a simple scaling factor, and *`exponent`* makes the result a curved line rather than a straight line. Because the color value is always between 0 and 1, the larger your exponent, the *smaller* the modified value will be. Figure 11-11 shows the curves generated with exponent values of 0.6 (the solid black line), 0.3 (the dashed line), and 1.66667 (the gray line).

Looking at the dashed line, you can see that a low original color value such as 0.1 will be boosted to 0.5, a 400% increase. An original value of 0.5, on the other hand, will increase only 60% to 0.8. The effect is to brighten the image as a whole and to increase the contrast in dark areas while reducing contrast in light areas. For exponents greater than 1 (the gray line), the modified values are smaller than the original, darkening the image while increasing the contrast in bright areas. Note that the solid gray and black lines are symmetrical around the diagonal: the gamma value of 1.6667 is the inverse of

a gamma of 0.6. In any case, the exponent has no effect when the original value is either 0 or 1.

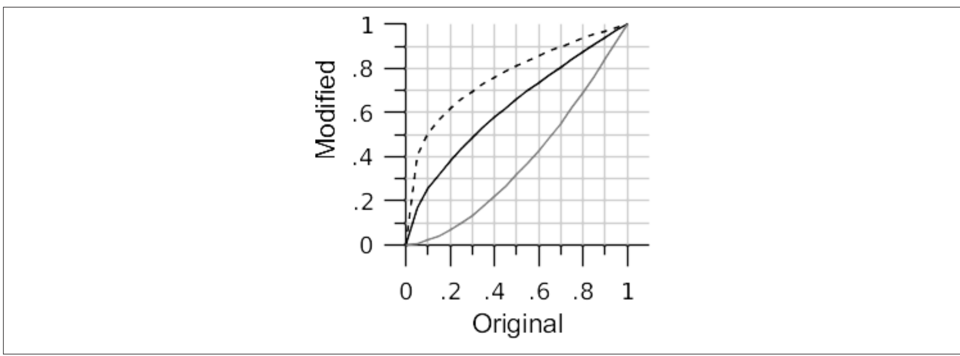


Figure 11-11. Gamma curve functions

When you specify a gamma filter, you set the `amplitude`, `exponent`, and `offset` attributes to correspond to the values in the preceding formula. Example 11-6 uses gamma correction to adjust the sky. In this particular case, the differences between Figures 11-10 and 11-12 are minor, but there are some images that can be improved much more by one method than by the other.

Example 11-6. Gamma adjustment with feComponentTransfer

http://oreillymedia.github.io/svg-essentials-examples/ch11/gamma_transfer.html

```
<feImage xlink:href="sky.jpg" result="sky"/>
<feComponentTransfer in="sky" result="sky">
  <feFuncB type="gamma"
    amplitude="1" exponent="0.2" offset="0"/>
  <feFuncR type="gamma"
    amplitude="1" exponent="0.707" offset="0"/>
  <feFuncG type="gamma"
    amplitude="1" exponent="0.707" offset="0"/>
</feComponentTransfer>
```

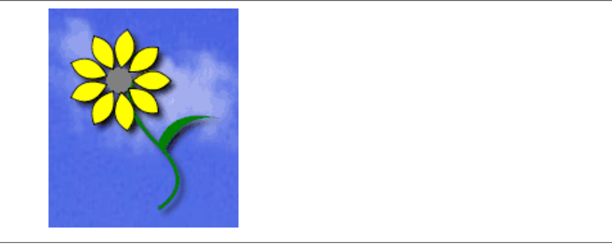
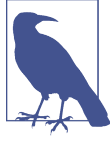


Figure 11-12. Result of using gamma correction



The astute reader (that's you) may have observed that both linear and gamma functions can produce color values greater than 1.0. The SVG specification says this is not an error; after each filter primitive, the SVG processor will clamp the values to a valid range. Thus, any value greater than 1.0 is reduced to 1.0, and any value less than 0 is set to 0.

`<feComponentTransfer>` has other options for the `type` attribute. Note that you may mix and match any of these; you can gamma-correct the red values while brightening the green values with a linear function:

`identity`
: A "do-nothing" function. This lets you explicitly state that a color channel should remain unaffected. (This is the default if you don't provide an `<feFunc`*`X`*`>` element for a particular channel.)

`table`
: Lets you divide the color values into a series of equal intervals, each of which will be proportionately scaled. Consider the following remapping, which doubles the value of the lowest quarter of the color range, squeezes the next quarter into a range of one-tenth, keeps the third quarter in exact proportion, and then squeezes the last quarter of the values into the remaining 15% of the color range:

Original value range	Modified value range
0.00–0.25	0.00–0.50
0.25–0.50	0.50–0.60
0.50–0.75	0.60–0.85
0.75–1.00	0.85–1.00

You would specify this mapping for the green channel by listing the endpoints of the remapped range in the `tableValues` attribute:

```
<feFuncG type="table"
  tableValues ="0.0, 0.5, 0.6, 0.85, 1.0"/>
```

If you are dividing the input spectrum into *n* different sections, you must provide *n*+1 items in `tableValues`, separated by whitespace or commas:

`discrete`
: Lets you divide the color values into a series of equal intervals, each of which will be mapped to a single discrete color value. Consider the following remapping, which maps the value of the lowest quarter of the color range to 0.125, sets the next quarter to 0.375, the third quarter to 0.625, and remaining quarter to 0.875 (i.e., each quarter of the range is mapped to its center point):

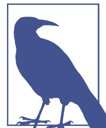
Original value range	Modified value
0.00–0.25	0.125
0.25–0.50	0.375
0.50–0.75	0.625
0.75–1.00	0.875

You would specify this mapping for the green channel by listing the discrete values, separated by commas or whitespace, in the `tableValues` attribute:

```
<feFuncG type="discrete"
  tableValues ="0.125 0.375 0.625 0.875"/>
```

Dividing the input channel into *n* sections requires *n* entries in the `tableValues` attribute. *Exception*: If you want to remap all the input values to a single output value, you must place that entry into `tableValues` twice. Thus, to set any input value of the blue channel to 0.5, you would say:

```
<feFuncB type="discrete" tableValues="0.5 0.5"/>
```



If you want to invert the range of color values for a channel (i.e., change increasing values from a minimum to maximum into decreasing values from the maximum to the minimum), use this:

```
<feFuncX type="table"
  tableValues="maximum minimum"/>
```

Figure 11-13 shows the result of using discrete and table transfers as well as inversion via a table transfer.

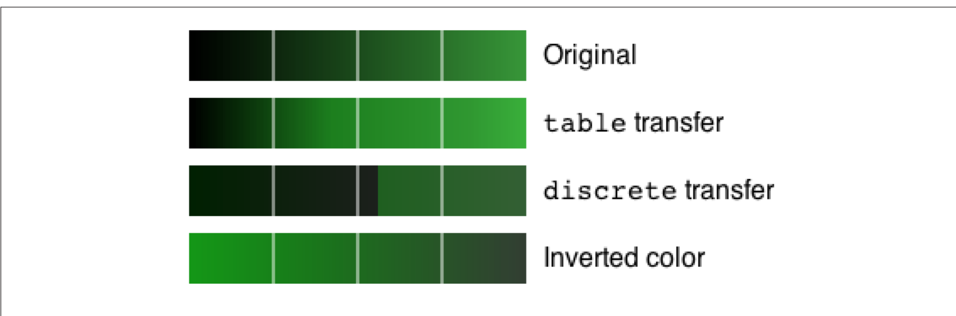


Figure 11-13. Result of using table and discrete transfers

Defining the Color Space

Ordinarily, the values for red, green, or blue run in a straight line from 0 to 1, with 0 being none of the color and 1 being 100% of the color. This is called a *linear color space*. However, when SVG calculates the color values between gradient stops (as described in Chapter 8, in "Gradients" on page 113), SVG uses a special way of representing color such that the values do not follow a straight line from 0 to 1. This representation is called the standard RGB or *sRGB* color space, and its use can make gradients much more natural-looking. Figure 11-14 shows a comparison. The first gradient goes from black to green, the second from red to green to blue, and the third from black to white.

By default, filter arithmetic calculates any interpolated ("in-between") values in the linear RGB space, so if you apply a filter to an object that has been filled with a gradient, you will get results that aren't at all what you expect. In order to get the correct result, you must tell the filter to do its calculations in sRGB space by adding a `color-interpolation-filters="sRGB"` attribute to your `<filter>` element. As an alternative, you may choose to leave the filter alone and apply `color-interpolation="linearRGB"` to the `<gradient>` element, so that it uses the same color space as the default for filters.

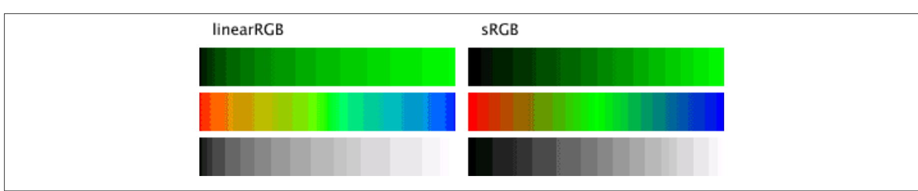


Figure 11-14. Comparison of linearRGB and sRGB

The <feComposite> Filter

So far we have combined the results of filters by using `<feMerge>` to layer the intermediate results one over another. The much more general `<feComposite>` element takes two inputs, specified with the `in` and `in2` attributes, and an `operator` that tells how the two are to be combined. In the following explanation, presume you've specified `result="A"` and `result="B"` for previous filter primitive outputs:

`<feComposite operator="`**`over`**`" in="A" in2="B"/>`
: Produces the result of layering `A` over `B`, exactly as `<feMergeNode>` does. In fact, `<feMergeNode>` is really just a convenient shortcut for an `<feComposite>` element that specifies an `over` operation. (`<feMergeNode>` also allows you to layer more than two graphics at a time.)

`<feComposite operator="in" in="A" in2="B"/>`
: The result is the parts of `A` that overlap the opaque areas of `B`. It is similar to a masking effect, but the mask is based on only the alpha channel of `B`, not its color luminance. Don't confuse the name of this attribute value with the `in` attribute.

`<feComposite operator="out" in="A" in2="B"/>`
: The result is the parts of `A` that are outside the opaque areas of `B` (with a reverse-masking effect for partially transparent areas).

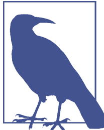
`<feComposite operator="atop" in="A" in2="B"/>`
: The result is the part of `A` that is inside B, as well as the part of `B` outside A. To quote the article in which these operators were first defined: "…*paper* atop *table* includes *paper* where it is on top of *table*, and *table* otherwise; area beyond the edge of the table is out of the picture."[1]

`<feComposite operator="xor" in="A" in2="B"/>`
: The result is the part of `A` that is outside `B` together with the part of `B` that is outside `A`.

`<feComposite in="A" in2="B" operator="arithmetic".../>`
: The ultimate in flexibility. You provide four coefficients: `k1`, `k2`, `k3`, and `k4`. The result for each channel of each pixel is calculated as follows:

```
k1 * A * B + k2 * A + k3 * B + k4
```

where `A` and `B` are the values for that channel and pixel from the input graphics.



The arithmetic operator is useful for doing a "dissolve" effect. If you want to have a resulting image that is *a*% of image A and *b*% of image B, set `k1` and `k4` to 0, `k2` to *a*/100, and `k3` to *b*/100. For example, to make a blend with 30% of `A` and 70% of `B`, you'd use this:

```
<feComposite in="A" in2="B" result="combined"
  k1="0" k2="0.30" k3="0.70" k4="0"/>
```

Figure 11-15 shows the combinations we've described, with red text as the `in` image and a blurred offset shadow as the `in2` image; the arithmetic blend is 50% of the text and 50% of its shadow.

1. "Compositing Digital Images," T. Porter, T. Duff, SIGGRAPH '84 Conference Proceedings, Association for Computing Machinery, Volume 18, Number 3, July 1984.

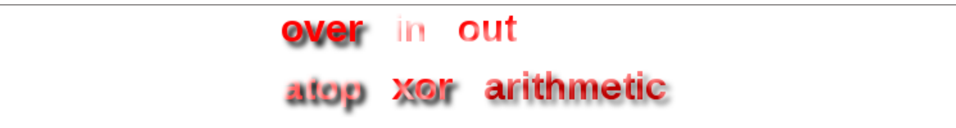


Figure 11-15. Result of using feComposite operators

Example 11-7 uses the `in` and `out` operators to do “cut-outs.” The drop shadow has been eliminated from this example to produce a more visually pleasing result in Figure 11-16.

Example 11-7. Use of feComposite in and out

```
<defs>
<filter id="sky-in" filterUnits="objectBoundingBox">
  <feImage xlink:href="sky.jpg" result="sky"
    x="0" y="0" width="100%" height="100%"
    preserveAspectRatio="none"/>
  <feComposite in="sky" in2="SourceGraphic"
    operator="in"/>
</filter>

<filter id="sky-out" filterUnits="objectBoundingBox">
  <feImage xlink:href="sky.jpg" result="sky"
    x="0" y="0" width="100%" height="100%"
    preserveAspectRatio="none"/>
  <feComposite in="sky" in2="SourceGraphic"
    operator="out"/>
</filter>

<g id="flower">
  <!-- flower graphic goes here -->
</g>
</defs>

<use xlink:href="#flower" transform="translate(10,10)"
  style="filter: url(#sky-in);"/>

<use xlink:href="#flower" transform="translate(170,10)"
  style="filter: url(#sky-out);"/>
```

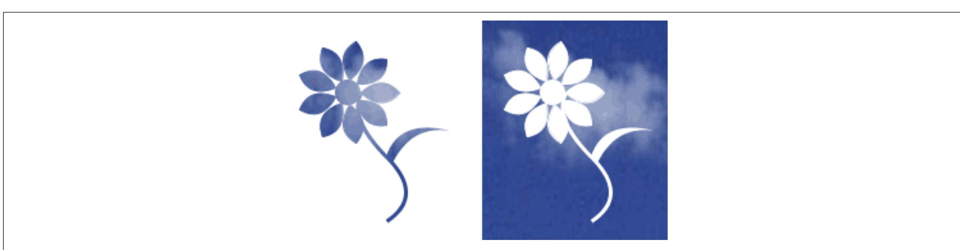


Figure 11-16. Result of feComposite in and out

The <feBlend> Filter

But wait, there's more! Yes, filters provide yet another way to combine images. The `<feBlend>` element requires two inputs, specified with the `in` and `in2` attributes, and a `mode` that tells how the inputs are to be blended. The possible values are: `normal`, `multiply`, `screen`, `lighten`, and `darken`. Given opaque inputs `<feBlend in="A" in2="B" mode="m"/>`, the following describes the color of the resulting pixel for each mode:

`normal`
: `B` only; this is the same as the `over` operator in `<feComposite>`.

`multiply`
: Multiplies (as the name suggests) `A`'s value and `B`'s value for each color channel. Because color values are in the range 0–1, multiplying them makes them *smaller*. This darkens colors, with the strongest effect for dark colors or very different intense colors, and no effect when one of the colors is white. The result is similar to creating photographic slides for both images and then stacking them together in the same slide projector—only the light that passes through both is visible.

`screen`
: Adds the color values together for each channel, and then subtracts their product. Bright or light colors tend to dominate over dark colors, but colors of similar brightness get combined. The result is similar to having two different slide projectors, one for each image, shining on the same screen—bright light from one projector overpowers shadows from the other.

`darken`
: Takes the minimum of `A` and `B` in each channel. This is the darker color, hence the name.

`lighten`
: Takes the maximum of `A` and `B` in each channel. This is the lighter color, hence the name.

Note that the appropriate calculation is done independently for each of the red, green, and blue values. So, if you were to darken a pure red square with RGB values of (100%, 0%, 0%) and a gray square with RGB values of (50%, 50%, 50%), the resulting color would be (50%, 0%, 0%). If the inputs are not opaque, then all the modes except for `screen` factor in the transparencies when making the calculations.

Finally, once the color value is calculated, the opacity of the result is determined by the formula `1 - (1 - opacity of A) * (1 - opacity of B)`. Using this formula, two opaque items will still be opaque; two items that are 50% opaque will combine to one that is 75% opaque.

Figure 11-17 shows the result of blending an opaque, white-to-black gradient bar with opaque and 50% opaque color squares that have RGB values of black (`#000`), yellow (`#ff0`), red (`#f00`), medium-bright green (`#0c0`), and dark blue (`#009`).

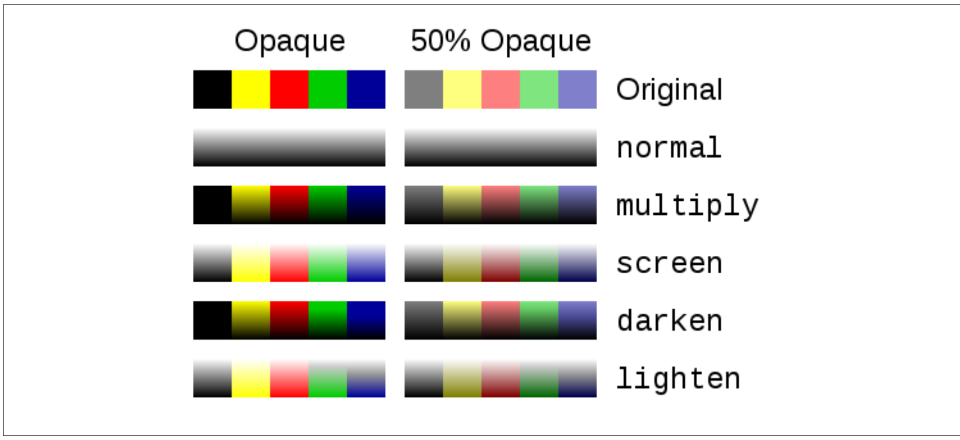


Figure 11-17. Result of feBlend

The <feFlood> and <feTile> Filters

The `<feFlood>` and `<feTile>` elements are *utility filters*. Much like `<feOffset>`, they allow you to carry out certain common operations within a series of filter primitives rather than having to create extra SVG elements in your main graphic.

`<feFlood>` provides a solid-colored area for use in compositing or merging. You provide the `flood-color` and `flood-opacity`, and the filter does the rest.

`<feTile>` takes its input and tiles it horizontally and vertically to fill the area specified in the filter. The size of the tile itself is specified by the size of the input to `<feTile>`.

Example 11-8 uses `<feComposite>` to cut out the flooded and tiled area to the shape of a flower. The image used as a tile is shown for reference at the upper right of Figure 11-18.

Example 11-8. Example of feFlood and feTile

```
<defs>
<filter id="flood-filter" x="0" y="0" width="100%" height="100%">
  <feFlood flood-color="#993300" flood-opacity="0.8" result="tint"/>
  <feComposite in="tint" in2="SourceGraphic"
    operator="in"/>
</filter>

<filter id="tile-filter" x="0" y="0" width="100%" height="100%">
  <feImage xlink:href="cloth.jpg" width="32" height="32"
    result="cloth"/>
```

```
  <feTile in="cloth" result="cloth"/>
  <feComposite in="cloth" in2="SourceGraphic"
    operator="in"/>
</filter>

<g id="flower">
    <!-- flower graphic goes here -->
</g>
</defs>

<use xlink:href="#flower" transform="translate(0, 0)"
  style="filter: url(#flood-filter);"/>
<use xlink:href="#flower" transform="translate(110,0)"
  style="filter: url(#tile-filter);"/>
<image xlink:href="cloth.jpg" x="220" y="10"
  width="32" height="32"/>
```

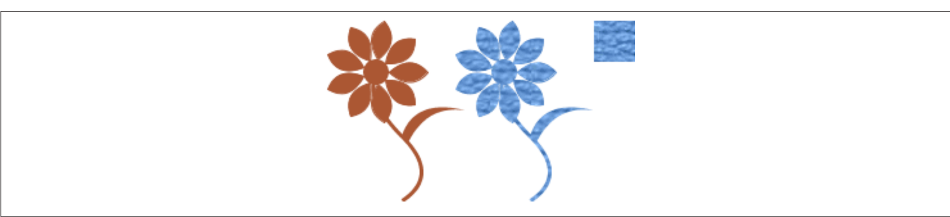


Figure 11-18. Result of feFlood and feTile elements

Lighting Effects

If you draw a bright green circle with SVG, it looks like a refugee from a traffic signal, glowing by its own light and otherwise lying flat on the screen. If you look at a circle cut out of green construction paper, it looks more "real" because it is lit from an outside source and has some texture. A circle cut from green plastic not only is lit from outside; it also has reflected highlights. Light from an outside source is called *diffuse lighting*, and the highlights that reflect off a surface are called *specular lighting*, from the Latin *speculum*, meaning *mirror*.

In order to achieve these effects, you must specify the following:

- The type of lighting you want (`<feDiffuseLighting>` or `<feSpecularLighting>`)
- The object you want to light
- The color of light you are using
- The type of light source you want (`<fePointLight>`, `<feDistantLight>`, or `<feSpotLight>`) and its location

You specify the location of a light source in three dimensions; this means you will need a *z*-value in addition to *x*- and *y*-values. As with two-dimensional graphics, the positive *x*-axis goes from left to right, and the positive *y*-axis goes from top to bottom. The positive *z*-axis is "coming out of the screen" and pointing at you.

Both these lighting effects use the alpha channel of the object they are illuminating as a *bump map*; higher alpha values are presumed to be "raised" above the surface of the object.

Diffuse Lighting

The best way to show how the `<feDiffuseLighting>` element works is to jump right into Example 11-9, which shines a pale yellow light on a green circle, textured with the curve pattern from Example 8-1.

Example 11-9. Diffuse lighting with a point light source

```
<defs>
  <path id="curve" d="M 0 0 Q 5 20 10 10 T 20 20"      ❶
      style="stroke: black; fill: none;"/>      ❷

  <filter id="diff-light" color-interpolation-filters="sRGB"
    x="0" y="0" width="100%" height="100%">     ❸

    <feImage xlink:href="#curve" result="tile"
        width="20" height="20"/>      ❹

    <feTile in="tile" result="tile"/>

    <feDiffuseLighting in="tile"      ❺
        lighting-color="#ffffcc"
        surfaceScale="1"      ❻
        diffuseConstant="0.5"      ❼
        result="diffuseOutput">      ❽
        <fePointLight x="0" y="50" z="50"/>      ❾
    </feDiffuseLighting>      ❿

    <feComposite in="diffuseOutput" in2="SourceGraphic"
        operator="in" result="diffuseOutput"/>      ⓫

    <feBlend in="diffuseOutput" in2="SourceGraphic"
        mode="screen"/>      ⓬
  </filter>
</defs>

<circle id="green-light" cx="50" cy="50" r="50"
    style="fill: #060; filter: url(#diff-light)"/>      ⓭
```

❶ Define the curve to be used as the tile.

❷ Set the color interpolation method and the boundaries for the filter.

❸ Tile the area of the filter with the `curve` image. This will become the bump map.

❹ This tiled area is the input to the `<feDiffuseLighting>` element, which is illuminated with a pale yellow light, as specified by the `lighting-color` attribute.

❺ The `surfaceScale` attribute tells the height of the surface for an alpha value of 1. (More generally, it's the factor by which the alpha value is multiplied.)

❻ `diffuseConstant` is a multiplicative factor that is used in determining the final RGB values of a pixel. It must have a value greater than or equal to 0; its default value is 1. The brighter your `lighting-color`, the smaller this number should be (unless you like having your picture washed out).

❼ The result of this filter will be named `diffuseOutput`.

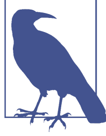
❽ This example uses a *point light source*, which means a source that radiates light in all directions. It is positioned at the left center of the area we wish to illuminate, and it is 50 units in front of the screen. The farther you set the light away from the object, the more evenly the object is illuminated. In this example, the light is up close and personal to get the greatest possible effect.

❾ The end of the `<feDiffuseLighting>` element.

❿ `<feComposite>`'s `in` operator clips the filter's output to the boundaries of the source graphic (the circle).

⓫ Finally, `<feBlend>` in `screen` mode, which tends to lighten the input, creates the final part of the filter.

⓬ Activate the filter on the desired object to produce Figure 11-19.



The input to this filter is a four-color graphic, but only the alpha channel is used. However, when I (David) inserted a `<feColorMatrix type="luminanceToAlpha">` and used its output as the input to the filter, I did not get the desired effect. Remember that `luminanceToAlpha` converts black areas (zero luminance) to complete transparency (zero alpha). That leaves no difference in alpha levels between the black squiggle pattern and the transparent empty background, and therefore no texture for the lighting effects.

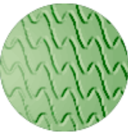


Figure 11-19. Result of applying diffuse lighting filter

Specular Lighting

Specular lighting, on the other hand, gives highlights rather than illumination. Example 11-10 shows how this works.

Example 11-10. Specular lighting with a distant light

```
<defs>
  <path id="curve" d="M 0 0 Q 5 20 10 10 T 20 20"
   style="stroke: black; fill: none;"/>     ❶

  <filter id="spec-light" color-interpolation-filters="sRGB"
    x="0" y="0" width="100%" height="100%">     ❷

    <feImage xlink:href="#curve" result="tile"
        width="20" height="20"/>     ❸

    <feTile in="tile" result="tile"/>

    <feSpecularLighting in="tile"     ❹
        lighting-color="#ffffcc"
        surfaceScale="1"     ❺
        specularConstant="1"     ❻
        specularExponent="4"     ❼
        result="specularOutput">     ❽
        <feDistantLight elevation="25" azimuth="0"/>     ❾
    </feSpecularLighting>     ❿

    <feComposite in="specularOutput" in2="SourceGraphic"
        operator="in" result="specularOutput"/>     ⓫

    <feComposite in="specularOutput" in2="SourceGraphic"
        operator="arithmetic" k1="0" k2="1" k3="1" k4="0"/>     ⓬
  </filter>
</defs>

<circle id="green-light" cx="50" cy="50" r="50"
  style="fill: #060; filter: url(#spec-light)"/>     ⓭
```

❶ As in the previous example, define the curve.

❷ The only difference between this and the previous example is the filter name.

❸ As in the previous example, this section tiles the curve.

❹ Starts the definition of the `<feSpecularLighting>` filter and specifies the `lighting-color` to be a pale yellow light.

❺ The `surfaceScale` attribute tells the height of the surface for an alpha value of 1. (Specifically, it's the factor by which the alpha value is multiplied.)

❻ `specularConstant` is a multiplicative factor used in determining the final RGB values of a pixel. It must have a value greater than or equal to 0; its default value is 1. The brighter your `lighting-color`, the smaller this number should be. The effect of this number is also moderated by the `specularExponent` attribute.

❼ `specularExponent` is another factor used in determining the final RGB values of a pixel. This attribute must have a value from 1 to 128; the default value is 1. The larger this number, the more "shiny" the result.

❽ The result of this filter will be named `specularOutput`.

❾ This example uses a distant light source, one which is so far away from the image that its light hits all parts of the image at the same angle. Instead of specifying the position of the light source, you specify the angle that the light is coming from.

The `elevation` and `azimuth` attributes let you specify the angle in three dimensions. Specifically, `elevation` gives the angle of light above the plane of the screen: `elevation="0"` is light shining flat across the image, while `elevation="90"` is light shining straight down.

The `azimuth` specifies the angle within the plane; when `elevation` is 0, `azimuth="0"` specifies light coming from the right of the image (more generally, the positive end of the *x*-axis); `azimuth="90"` is from the bottom (positive end of the *y*-axis), `azimuth="180"` is from the left, and `azimuth="270"` is from the top.

❿ The end of the `<feSpecularLighting>` element. Note that the input to this filter was an alpha channel; the output contains both alpha *and* color information (unlike `<feDiffuseLighting>`, which always produces an opaque result).

⓫ Use `<feComposite>`'s `in` operator to clip the filter's output to the boundaries of the source graphic (the circle).

⓬ Finally, use `<feComposite>` with the `arithmetic` operator to do a straight addition of the lighting and the source graphic.

⓭ Activate the filter on the circle, producing the highlighting relief effect in Figure 11-20.

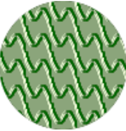
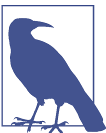


Figure 11-20. Result of applying specular lighting filter



An excellent tutorial on lighting effects in three dimensions is available. We're working in only two dimensions, but much of the information is applicable.

A third type of light source, `<feSpotLight>`, is specified with these attributes: `x`, `y`, and `z`, the location of the spotlight (default value is 0); `pointsAtX`, `pointsAtY`, and `pointsAtZ`, the place the spotlight is pointing at (default value is 0); `specularExponent`, a value that controls the focus for the light source (default value is 1); and `limitingConeAngle`, which restricts the region where the light is projected. This is the angle between the spotlight axis and the cone. Thus, if you want a 30-degree spread for the entire cone, specify the angle as 15. (The default value is to allow unlimited spread.)

Accessing the Background

In addition to the `SourceGraphic` and `SourceAlpha` filter inputs, a filtered object may access the part of the image that has already been rendered onto the canvas when you invoke a filter. These parts are called `BackgroundImage` (*not* `BackgroundGraphic`) and `BackgroundAlpha`. In order to access these inputs, the filtered object must be within a container element that has set the `enable-background` attribute to the value `new`. Example 11-11 performs a Gaussian blur on the background image.

Example 11-11. Accessing the background image

```
<defs>
  <filter id="blur-background">   ❶
    <feGaussianBlur in="BackgroundImage" stdDeviation="2" result="blur" />
    <feComposite in="blur" in2="SourceGraphic" operator="in" />
    <feOffset dx="4" dy="4" result="offsetBlur"/>
  </filter>
</defs>

<g enable-background="new">   ❷
  <rect x="0" y="0" width="60" height="60"
    style="fill: lightblue; stroke: blue; stroke-width:10" />   ❸
  <circle cx="40" cy="40" r="30"
```

```
    style="fill: #fff; filter: url(#blur-background);" />   ❹
</g>
```

❶ This is similar to the blur filter used for drop shadows, except the input is now the `BackgroundImage` rather than the `SourceAlpha`.

❷ Because `<g>` is a container element, it is a perfect candidate for placing the `enable-background`. All the children of this element will have access to the background image and alpha.

❸ The rectangle is drawn onto the canvas *and* into a background buffer.

❹ The circle does not display directly; the filter blurs the background image (which does not include the circle) and composites `in` the `SourceGraphic`. Figure 11-21 shows the result.

As of the time of writing, no web browser implements `enable-background` or the `BackgroundImage` and `BackgroundAlpha` inputs. If you use those inputs in a filter in a browser that does not support them, the filter will not return anything—meaning that the filtered part of your graphic will disappear.

An alternative is to separate out the background into its own `<g>` element and use `<feImage>` to import it into your filter. Adapting Example 11-11 in this way results in the following code:

```
<defs>
  <filter id="blur-background">
    <feImage xlink:href="#background" result="bg"/>
    <feGaussianBlur in="bg" stdDeviation="2" result="blur" />
    <feComposite in="blur" in2="SourceGraphic" operator="in" />
    <feOffset dx="4" dy="4" result="offsetBlur"/>
  </filter>
</defs>

<g id="background">
  <rect x="0" y="0" width="60" height="60"
    style="fill: lightblue; stroke: blue; stroke-width:10" />
</g>
<circle cx="40" cy="40" r="30"
    style="fill: #fff; filter: url(#blur-background);" />
```

The result is the same as Figure 11-21 in browsers that support the use of SVG fragments for `<feImage>` (it currently does not work in Mozilla Firefox).

Figure 11-21. Result of accessing background image

The <feMorphology> Element

The `<feMorphology>` element lets you "thin" or "thicken" a graphic. You specify an `operator` with a value of `erode` to thin or `dilate` to thicken a graphic. The `radius` attribute tells us how much the lines are to be thickened or thinned. It's ordinarily applied to alpha channels. Example 11-12 erodes and dilates a simple line drawing. As you see in Figure 11-22, erosion can wreak havoc on a drawing that has thin lines to begin with.

Example 11-12. Thickening and thinning with feMorphology

http://oreillymedia.github.io/svg-essentials-examples/ch11/fe_morphology.html

```
<defs>
  <g id="cat" stroke-width="2">
      <!-- drawing of a cat -->
  </g>

  <filter id="erode1">
    <feMorphology operator="erode" radius="1"/>
  </filter>

  <filter id="dilate2">
    <feMorphology operator="dilate" radius="2"/>
  </filter>
</defs>

<use xlink:href="#cat"/>
<text x="75" y="170" style="text-anchor: middle;">Normal</text>

<use xlink:href="#cat" transform="translate(150,0)"
    style="filter: url(#erode1);"/>
<text x="225" y="170" style="text-anchor: middle;">Erode 1</text>

<use xlink:href="#cat" transform="translate(300,0)"
    style="filter: url(#dilate2);"/>
<text x="375" y="170" style="text-anchor: middle;">Dilate 2</text>
```

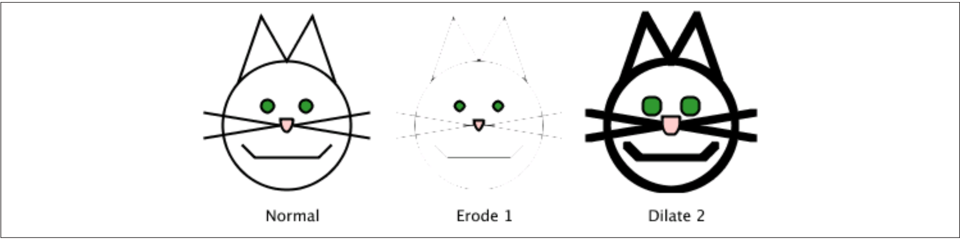


Figure 11-22. Result of using feMorphology

The <feConvolveMatrix> Element

The `<feConvolveMatrix>` element lets you calculate a pixel's new value in terms of the values of its neighboring pixels. This filter lets you do effects such as blurring, sharpening, embossing, and beveling. It works by combining a pixel with its neighboring pixels to produce a resulting pixel value. Imagine a pixel P and its eight neighboring pixels (the usual case that is used with this filter):

```
A  B  C
D  P  E
F  G  H
```

You then specify a list of nine numbers in the `kernelMatrix` attribute. These numbers tell how much to multiply each pixel by. These products will be added up. The sum could well come out to be greater than 1 (if all the factors are positive, for example), so, to even the intensity, the result is divided by the total of the factors. Let's say you specify these nine numbers (spaced out to show them as a matrix):

```
<feConvolveMatrix kernelMatrix="
   0  1  2
   3  4  5
   6  7  8"/>
```

The new value of pixel P will then be as follows:

```
P' = ((0*A) + (1*B) + (2*C) +
     (3*D) + (4*P) + (5*E) +
     (6*F) + (7*G) + (8*H)) / ( 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8)
```

The exception is if all your matrix values sum to 0; in this case, no division is performed.

You can also specify a `bias` property, which shifts the output range of the filter by adding the specified offset value to each pixel. The `bias` is added after the division, but before the result is clamped to the 0–1 allowable range.

Example 11-13 achieves the embossing effect shown in Figure 11-23 by taking the upper-left neighbor minus the lower-right neighbor of each pixel.[2] The `bias` of 0.5 is added after the `kernelMatrix` summation and division is applied. The bias causes pixels with identical top-left and bottom-right neighbors to be displayed as gray (versus black without a bias shift). Pixels where the top-left neighbor is brighter than the bottom-left neighbor display in dark colors, while pixels where the bottom-left neighbor is brighter display in bright colors. As a result, diagonal edges are highlighted as if the image were raised up and lit from the side. The transparent background pixels are considered to have a color of black.

The default behavior of `<feConvolveMatrix>` is to apply the calculations to all the channels, including alpha. With that setting, only the edges of each shape—the ones for which the upper-left neighbors have a higher alpha value than the (transparent background) lower-right neighbors—would be displayed. In order to apply calculations only to the red, green, and blue values, we specified `preserveAlpha` as `true`; the default value is `false`.

Example 11-13. Embossing with feConvolveMatrix

```
<defs>
  <filter id="emboss">
    <feConvolveMatrix
      preserveAlpha="true"
      kernelMatrix="1 0 0 0 0 0 0 0 -1"
      bias="0.5"/>
  </filter>

  <g id="flower">
    <!-- flower graphic goes here -->
  </g>
</defs>

<use xlink:href="#flower" style="filter: url(#emboss);"/>
```

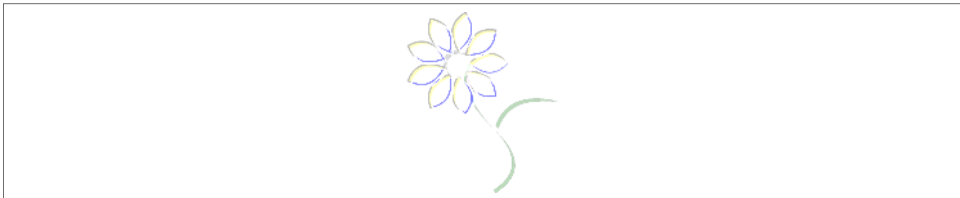


Figure 11-23. Result of using feConvolveMatrix

2. Filters containing `<feConvolveMatrix>` elements caused rendering errors when tested in Apache Batik version 1.7; the example works as expected when it was tested in web browsers.

Although the default matrix size is three columns by three rows, you can specify any size you want with the `order` attribute. If you specify `order="4"`, then the matrix will require sixteen numbers (4 by 4) in the `kernelMatrix` attribute. A matrix with three columns and two rows would be specified by `order="3 2"` and would require six numbers. The larger your kernel matrix, the more computation is required to produce the result.

For a pixel in the middle of a graphic, the neighbors are easy to identify. What do you do with the pixels on the edges of the graphic? Who are their neighbors? This decision is made by the setting you give the `edgeMode` attribute. If you set its value to be `duplicate` (the default), then `<feConvolveMatrix>` duplicates the edge values in the required direction to produce a neighbor. The value `wrap` wraps around to the opposite side to find a neighbor. For example, the neighbor above a pixel at the top is the pixel at the bottom, and the neighbor to the left of a pixel at the left edge is the corresponding pixel at the right edge. This behavior is useful if the image being modified will be used as a repeating tile. The value of `none` will provide a transparent black pixel (red, green, blue, and alpha values of zero) for any missing neighbors.

All the possibilities from `<feConvolveMatrix>` can't possibly be described here. Experiment with the online example and see what you can come up with:

http://oreillymedia.github.io/svg-essentials-examples/ch11/convolve.html

The <feDisplacementMap> Element

This fascinating filter uses the color values of its second input to decide how far to move the pixels in the first input. You specify which color channel should be used to affect the *x*-coordinate of a pixel with the `xChannelSelector` attribute; the `yChannelSelector` attribute specifies the color channel used to affect the *y*-coordinate. The legal values for these selectors are `"R"`, `"G"`, `"B"`, and `"A"` (for the alpha channel). You must specify how far to displace pixels; the `scale` attribute gives the appropriate scaling factor. If you don't specify this attribute, the filter won't do anything.

Example 11-14 creates a gradient rectangle as the second input. The displacement factor will be set to 10, the red channel will be used as an *x* offset, and the green channel will be used as a *y* offset. Figure 11-24 shows the result of applying this displacement to the flower.

Example 11-14. Using a gradient as a displacement map

```
<defs>
  <linearGradient id="gradient">
    <stop offset="0" style="stop-color: #ff0000;" />
    <stop offset="0.5" style="stop-color: #00ff00;"/>
    <stop offset="1" style="stop-color: #000000;"/>
  </linearGradient>
```

```
  <rect id="rectangle" x="0" y="0" width="100" height="200"
    style="fill: url(#gradient);"/>

  <filter id="displace">
    <feImage xlink:href="#rectangle" result="grad"/>

    <feDisplacementMap
      scale="10"
      xChannelSelector="R"
      yChannelSelector="G"
      in="SourceGraphic" in2="grad"/>
  </filter>
  <g id="flower">
    <!-- flower graphic goes here -->
  </g>
</defs>

<use xlink:href="#flower" style="filter: url(#displace);"/>
```

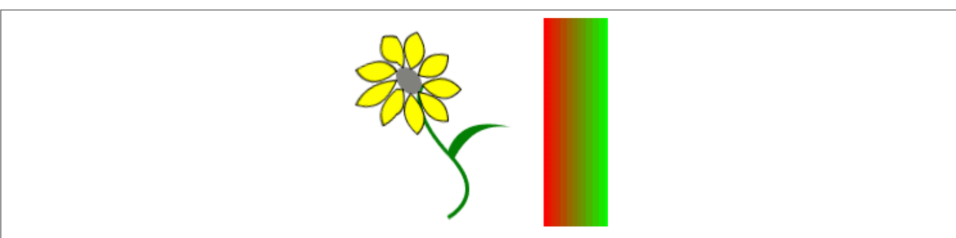


Figure 11-24. Result of using feDisplacementMap

It's possible to use the same graphic for both inputs. This means a graphic's displacement is controlled by its own coloration. This effect, as written in Example 11-15 and displayed in Figure 11-25, can be quite eccentric.

Example 11-15. Using a graphic as its own displacement map

```
<defs>
<filter id="self-displace">
  <feDisplacementMap
    scale="10"
    xChannelSelector="R"
    yChannelSelector="G"
    in="SourceGraphic" in2="SourceGraphic"/>
</filter>

<g id="flower">
  <!-- flower graphic goes here -->
</g>
</defs>
```

```
<use xlink:href="#flower" style="filter: url(#self-displace);"/>
```

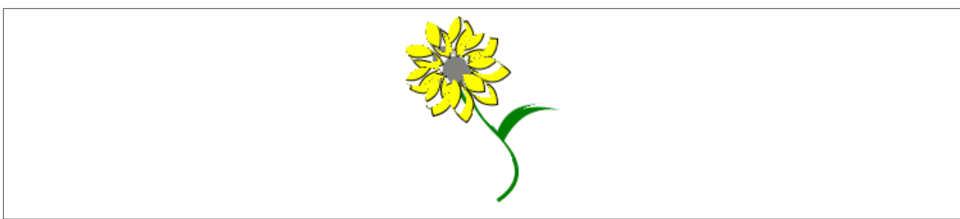


Figure 11-25. Same graphic used as both inputs to feDisplacementMap

The <feTurbulence> Element

The `<feTurbulence>` element lets you produce artificial textures for effects like marble, clouds, etc. by using equations developed by Ken Perlin.This is referred to as Perlin noise. You specify these attributes:

`type`
: One of `turbulence` or `fractalNoise`. Fractal noise is smoother in appearance.

`baseFrequency`
: The larger the number you give as the value for this attribute, the more quickly colors change in the result. This number must be greater than 0 and should be less than 1. You may also give two numbers for this attribute; the first will be the frequency in the *x* direction, and the second will be the frequency in the *y* direction.

`numOctaves`
: This is the number of noise functions that should be added together when generating the final result. The larger this number, the more fine-grained the texture. The default value is 1.

`seed`
: The starting value for the random number generator this filter uses. The default value is 0; change it to get some variety in the result.

Figure 11-26 is a screenshot of an SVG file showing various values of the first three of these attributes.

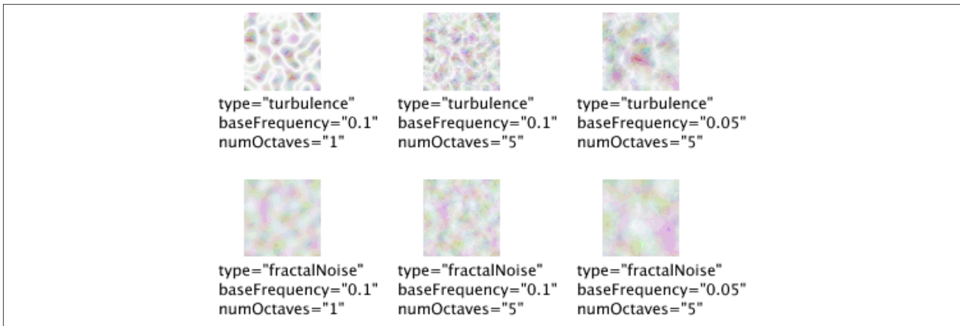


Figure 11-26. Various values of feTurbulence attributes

The online example allows you to experiment with all three parameters:

http://oreillymedia.github.io/svg-essentials-examples/ch11/turbulence.html

Filter Reference Summary

The `<filter>` element contains a series of filter primitives, each of which takes one or more inputs and provides a single result for use with other filters. The result of the last filter in the series is rendered into the final graphic. You specify the dimensions of the canvas to which the filter applies with the `x`, `y`, `width`, and `height` attributes. Use `filterUnits` to specify the units used to define the filter region, and `primitiveUnits` to specify the coordinate system for the various length values within the filter primitives.

Table 11-1 presents a filter reference summary. Each of the filter primitive elements has an `in` attribute that gives the source for the primitive, and may also specify an `x`, `y`, `width`, and `height`.

Table 11-1. Filter reference summary

Element	Attributes

`<feBlend>`	`in2="`*`second source`*`"`
	`mode="normal" | "multiply" | "screen" | "darken" | "lighten"`
	(default is `normal`)

`<feColorMatrix>`	`type="matrix" | "saturate" | "hueRotate" |`
	`"luminanceToAlpha"`
	`values="`*`matrix values`*`" | "`*`saturation value (0-1)`*`" |`
	`"`*`rotate degrees`*`"`

`<feComponentTransfer>`	Container for `<feFuncR>`, `<feFuncG>`, `<feFuncB>`, and `<feFuncA>` elements

Element	Attributes
<feFunc*X*>	type="identity" | "table" | "discrete" | "linear" | "gamma" tableValues="*intervals for table, steps for discrete*" slope="*linear slope*" intercept="*linear intercept*" amplitude="*gamma amplitude*" exponent="*gamma exponent*" offset="*gamma offset*"
<feComposite>	in2="*second source*" operator="over" | "in" | "out" | "atop" | "xor" | "arithmetic" The following attributes are used with arithmetic (any attributes that are not specified have a default value of 0): k1="*factor for in1*in2*" k2="*factor for in1*" k3="*factor for in2*" k4="*additive offset*"
<feConvolveMatrix>	order="*columns rows*" (default 3 by 3) kernel="*values*" bias="*offset value*"
<feDiffuseLighting>	Container for a light source element surfaceScale="*height*" (default 1) diffuseConstant="*factor*" (must be non-negative; default 1)
<feDisplacementMap>	scale="*displacement factor*" (default 0) xChannelSelector="R" | "G" | "B" | "A" yChannelSelector="R" | "G" | "B" | "A" in2="*second input*"
<feFlood>	flood-color="*color specification*" flood-opacity="*value (0-1)*"
<feGaussianBlur>	stdDeviation="*blur spread*" (larger is blurrier; default 0)
<feImage>	xlink:href="*image source*"
<feMerge>	Container for <feMergeNode> elements
<feMergeNode>	in="*intermediate result*"
<feMorphology>	operator="erode" | "dilate" radius="*x-radius y-radius*" radius="*radius*"
<feOffset>	dx="*x offset*" (default 0) dy="*y offset*" (default 0)
<feSpecularLighting>	Container for a light source element surfaceScale="*height*" (default 1) specularConstant="*factor*" (must be non-negative; default 1) specularExponent="*exponent*" (range 1-128; default 1)
<feTile>	Tiles the *in* layer

Element	Attributes

`<feTurbulence>`	`type="turbulence" | "fractalNoise"`
	`baseFrequency="`*`x-frequency y-frequency`*`"`
	`baseFrequency="`*`frequency`*`"`
	`numOctaves="`*`integer`*`"`
	`seed="`*`number`*`"`

`<feDistantLight>`	`azimuth="`*`degrees`*`"` (default 0) `elevation="`*`degrees`*`"` (default 0)
`<fePointLight>`	`x="`*`coordinate`*`"` (default 0) `y="`*`coordinate`*`"` (default 0) `z="`*`coordinate`*`"` (default 0)
`<feSpotLight>`	`x="`*`coordinate`*`"` (default 0) `y="`*`coordinate`*`"` (default 0) `z="`*`coordinate`*`"` (default 0) `pointsAtX="`*`coordinate`*`"` (default 0) `pointsAtY="`*`coordinate`*`"` (default 0) `pointsAtZ="`*`coordinate`*`"` (default 0) `specularConstant="`*`focus control`*`"` (default 1) `limitingConeAngle="`*`degrees`*`"`

CHAPTER 12
Animating SVG

Up to this point, all the images you have seen are static images; once constructed, they never change. In this chapter, we will examine two of three methods of making graphic images move. The first method, SMIL-based animation, should be used for movement that is a fundamental part of what the graphic represents, and for which the motion can be defined ahead of time. CSS animations should be used for stylistic effects and simple feedback (like highlighting an element on focus/hover). Scripting should be used for more complex interaction, and it is covered in Chapter 13.

In Chapter 11, we suggested that filters should be used as a means to enhance a graphic's message, not as an end in themselves. This suggestion is even more crucial with animation. Drunk with the power of animation, you will be tempted to turn your every graphic into an all-dancing, all-singing, Broadway spectacular. As long as your goal is experimentation, this is fine. If your goal is to convey a message, however, nothing is worse than gratuitous use or overuse of animation. Let me state this clearly: nobody except the company CEO is interested in repeated viewing of a spinning, flashing, color-changing, strobe-lit version of the company logo.

In this chapter, the message *is* the animation, so most of the examples will be remarkably free of any content while, of course, avoiding gratuitous and overwrought animation as much as possible.

Internet Explorer browsers (through version 11, the latest at the time of writing) do not support either the SMIL-based animation or CSS animation applied to SVG elements. There are JavaScript-based solutions such as SMILscript and FakeSMILe that can convert SMIL-based animation into scripted animation for Internet Explorer.

Animation Basics

The animation features of SVG are based on the World Wide Web Consortium's Synchronized Multimedia Integration Language Level 3 (SMIL3) specification. In this system, you specify the starting and ending values of the attribute, color, motion, or transformation you wish to animate; the time at which the animation should begin; and the duration of the animation. Example 12-1 gives the basic code.

Example 12-1. The incredible shrinking rectangle

http://oreillymedia.github.io/svg-essentials-examples/ch12/simple_animation.html

```
<rect x="10" y="10" width="200" height="20" stroke="black" fill="none">
  <animate
    attributeName="width"
    attributeType="XML"
    from="200" to="20"
    begin="0s" dur="5s"
    fill="freeze" />
</rect>
```

The first thing to notice is that the `<rect>` element is no longer an empty element; the animation is contained within the element.

The `<animate>` element specifies the following:

- The `attributeName` whose value should change over time; in this case, `width`.
- The `attributeType`. The `width` attribute is an `XML` attribute. The other common value of `attributeType` is `CSS`, indicating that the property you want to change is a CSS property. If you omit this specification, the default value of `auto` is used; it searches through CSS properties first and then XML attributes.
- The starting (`from`) and ending (`to`) values for the attribute. In this example, the starting value is 200, and the ending value is 20. The `from` value is optional; if you leave it out, the starting value is whatever was specified in the parent element.There is also a `by` attribute, which you may use instead of `to`; it is an offset added to the starting `from` value; the result is the ending value.
- The beginning and duration times for the animation. In this example, time is measured in seconds, specified by the `s` after the number. Other ways to define time are described in the next section, "How Time Is Measured" on page 194.
- What to do when the animation finishes. In this example, after the 5-second duration, the attribute will "freeze" at the `to` value. This is the SMIL `fill` attribute, which tells the animation engine how to fill up the remaining time. Don't confuse it with SVG's `fill` attribute, which tells SVG how to paint an object. If you remove this

line, the default value (`remove`) will return the `width` attribute to its original value of 200 after the 5-second animation has finished.

Figures 12-1 and 12-2 show the beginning and ending stages of the animation. They can't do justice to the actual effect, so we strongly recommend you try it out within your browser.

Figure 12-1. Beginning of animation

Figure 12-2. Ending of animation

Example 12-2 is a bit more ambitious. It starts with a 20 by 20 green square that will grow to 250 by 200 over the space of 8 seconds. For the first 3 seconds, the opacity of the green will increase, and then decrease for the next 3 seconds. Note that `fill-opacity` is referred to with `attributeType="CSS"` because it was set in a `style`.

Example 12-2. Multiple animations on a single object

http://oreillymedia.github.io/svg-essentials-examples/ch12/multiple_animation.html

```
<rect x="10" y="10" width="20" height="20"
  style="stroke: black; fill: green; style: fill-opacity: 0.25;">
  <animate attributeName="width" attributeType="XML"
    from="20" to="200" begin="0s" dur="8s" fill="freeze"/>
  <animate attributeName="height" attributeType="XML"
    from="20" to="150" begin="0s" dur="8s" fill="freeze"/>
  <animate attributeName="fill-opacity" attributeType="CSS"
    from="0.25" to="1" begin="0s" dur="3s" fill="freeze"/>
  <animate attributeName="fill-opacity" attributeType="CSS"
    from="1" to="0.25" begin="3s" dur="3s" fill="freeze"/>
</rect>
```

The last simple example, Example 12-3, animates a square and a circle. The square will expand from 20 by 20 to 120 by 120 over the space of 8 seconds. Two seconds after the beginning of the animation, the circle's radius will start expanding from 20 to 50 over the space of four seconds. Figure 12-3 shows a combined screenshot of the animation at four times: 0 seconds, when the animation begins; 2 seconds, when the circle starts to grow; 6 seconds, when the circle finishes growing; and 8 seconds, when the animation is finished.

Example 12-3. Simple animation of multiple objects

http://oreillymedia.github.io/svg-essentials-examples/ch12/multiple_animation2.html

```
<rect x="10" y="10" width="20" height="20"
  style="stroke: black; fill: #cfc;">
  <animate attributeName="width" attributeType="XML"
     begin="0s" dur="8s" from="20" to="120" fill="freeze"/>
  <animate attributeName="height" attributeType="XML"
     begin="0s" dur="8s" from="20" to="120" fill="freeze"/>
</rect>

<circle cx="70" cy="70" r="20"
  style="fill: #ccf; stroke: black;">
  <animate attributeName="r" attributeType="XML"
    begin="2s" dur="4s" from="20" to="50" fill="freeze"/>
</circle>
```

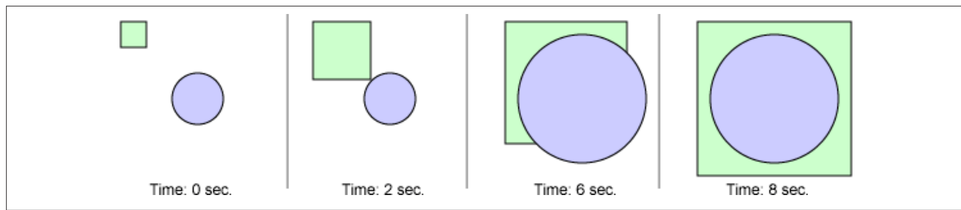


Figure 12-3. Stages of multiobject animation

How Time Is Measured

SVG's animation clock starts ticking when the SVG has finished loading, and it stops ticking when the user leaves the page. You may specify a beginning or duration for a particular animation segment as a numeric value in one of these ways:

- A full clock value in hours, minutes, and seconds (`1:20:23`).
- A partial clock value in minutes and seconds (`02:15`).
- A time value followed by an abbreviation that is one of `h` (hours), `min` (minutes), `s` (seconds), or `ms` (milliseconds), for example, `dur="3.5s" begin="1min"`. You may not put any whitespace between the value and the unit.

If no unit is specified, the default is seconds.

Synchronizing Animation

Instead of defining each animation's start time as the document loading time, you can tie an animation's beginning time to the beginning or end of another animation.

Example 12-4 animates two circles; the second one will start expanding as soon as the first one has stopped shrinking. Figure 12-4 shows the important stages of the animation.

Example 12-4. Synchronization of animations

```
<circle cx="60" cy="60" r="30" style="fill: #f9f; stroke: gray;">
  <animate id="c1" attributeName="r" attributeType="XML"
    begin="0s" dur="4s" from="30" to="10" fill="freeze"/>
</circle>

<circle cx="120" cy="60" r="10" style="fill: #9f9; stroke: gray;">
  <animate attributeName="r" attributeType="XML"
    begin="c1.end" dur="4s" from="10" to="30" fill="freeze"/>
</circle>
```

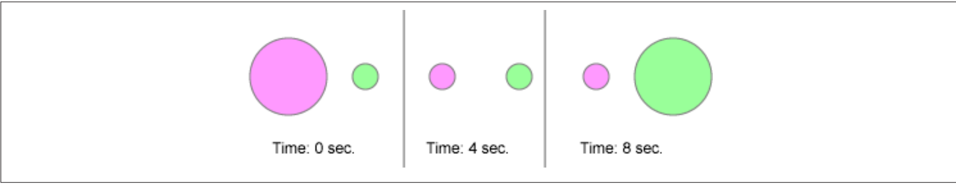


Figure 12-4. Stages of synchronized animations

It is also possible to add an offset to a synchronization. To make an animation start 2 seconds after another animation, you would use a construction of the form `begin="otherAnim.end+2s"`. (You may add whitespace around the plus sign.) In Example 12-5, the second circle begins to grow 1 1/4 seconds after the first circle begins shrinking.

Example 12-5. Synchronization of animations with offsets

http://oreillymedia.github.io/svg-essentials-examples/ch12/sync_with_offset.html

```
<circle cx="60" cy="60" r="30" style="fill: #f9f; stroke: gray;">
  <animate id="c1" attributeName="r" attributeType="XML"
    begin="0s" dur="4s" from="30" to="10" fill="freeze"/>
</circle>

<circle cx="120" cy="60" r="10" style="fill: #9f9; stroke: gray;">
  <animate attributeName="r" attributeType="XML"
    begin="c1.begin+1.25s" dur="4s" from="10" to="30" fill="freeze"/>
</circle>
```

Now that you know about synchronizing animations, we can introduce the `end` attribute, which sets an end time for an animation. This is *not* a substitute for the `dur` attribute! The following animation will start 6 seconds after the page loads. It will last for 12 seconds or until an animation named `otherAnim` ends, whichever comes first:

```
<animate attributeName="width" attributeType="XML"
  begin="6s" dur="12s" end="otherAnim.end"
  from="10" to="100" fill="freeze"/>
```

You can, of course, set the value of end to a specific time; this is useful for halting an animation partway through so you can see if everything is in the right place. This is how we were able to create Figure 12-3. The following animation starts at 5 seconds and should last for 10 seconds, but is halted 9 seconds after the document loads (4 seconds after the animation starts). The animation is stopped 40% of the way through, so the width will freeze at a value of 140 (40% of the distance from 100 to 200):

```
<animate attributeName="width" attributeType="XML"
    begin="5s" dur="10s" end="9s"
    from="100" to="200" fill="freeze"/>
```

Repeated Action

The animations so far occur exactly once; `fill` is set to `freeze` to keep the final stage of the animation. If you want to have the object return to its pre-animation state, omit the attribute. (This is equivalent to setting `fill` to the default value of `remove`.)

Two other attributes allow you to repeat an animation. The first of them, `repeatCount`, is set to an integer value telling how many times you want a particular animation to repeat. The second, `repeatDur`, is set to a time telling how long the repetition should last. If you want an animation to repeat until the user leaves the page, set either `repeatCount` or `repeatDur` to the value `indefinite`. You will usually use only one of the two, not both. If you do specify both `repeatCount` and `repeatDur`, the one that specifies the end time that occurs first will be used.

The animation in Example 12-6 shows two circles. The upper circle moves from left to right in two repetitions of 5 seconds each. The second circle moves from right to left for a total of 8 seconds.

Example 12-6. Example of repeated animation

http://oreillymedia.github.io/svg-essentials-examples/ch12/repeated_action.html

```
<circle cx="60" cy="60" r="30" style="fill: none; stroke: red;">
  <animate attributeName="cx" attributeType="XML"
    begin="0s" dur="5s" repeatCount="2"
    from="60" to="260" fill="freeze"/>
</circle>

<circle cx="260" cy="90" r="30" style="fill: #ccf; stroke: black;">
  <animate attributeName="cx" attributeType="XML"
    begin="0s" dur="5s" repeatDur="8s"
    from="260" to="60" fill="freeze"/>
</circle>
```

Just as it is possible to synchronize an animation with the beginning or ending of another animation, you can tie the start of one animation to the start of a specific repetition of another animation. You give the first animation an *id*, and then set the `begin` of the second animation to `id.repeat(count)`, where *count* is a number beginning at 0 for the first repetition. Example 12-7 shows an upper circle moving from left to right three times, requiring 5 seconds for each repetition. The lower square will go right to left only once, and will not begin until halfway through the second repetition.

Example 12-7. Synchronizing an animation with a repetition

http://oreillymedia.github.io/svg-essentials-examples/ch12/sync_repetition.html

```
<circle cx="60" cy="60" r="15"
  style="fill: none; stroke: red;">
  <animate id="circleAnim" attributeName="cx" attributeType="XML"
    begin="0s" dur="5s" repeatCount="3"
    from="60" to="260" fill="freeze"/>
</circle>

<rect x="230" y="80" width="30" height="30"
  style="fill: #ccf; stroke: black;">
  <animate attributeName="x" attributeType="XML"
    begin="circleAnim.repeat(1)+2.5s" dur="5s"
    from="230" to="30" fill="freeze"/>
</rect>
```

Animating Complex Attributes

Animation is not limited to simple numbers and lengths. You can animate nearly any attribute or style where you can calculate a smooth transition between two values.

To animate a color, simply make the `from` and `to` attributes valid color values, as described in Chapter 4, in "Stroke Color" on page 41. The color is treated as a vector of three numbers for the calculations; the `R`, `G`, and `B` values will each transition from their value in one color to the other.[1] Example 12-8 animates the fill and stroke colors of a circle, changing the fill from light yellow to red, and the gray outline to blue. Both animations start 2 seconds after the page loads; this gives you time to see the original colors.

Example 12-8. Example of animating color

http://oreillymedia.github.io/svg-essentials-examples/ch12/animate_color.html

```
<circle cx="60" cy="60" r="30"
        style="fill: #ff9; stroke: gray; stroke-width: 10;">
```

1. The color change is affected by the `color-interpolation` property, as described in "Defining the Color Space" on page 169. The default interpolation, `sRGB`, usually produces pleasant results.

```
  <animate attributeName="fill"
    begin="2s" dur="4s" from="#ff9" to="red" fill="freeze"/>
  <animate attributeName="stroke"
    begin="2s" dur="4s" from="gray" to="blue" fill="freeze"/>
</circle>
```

You can also animate attributes that are lists of numbers, so long as the *number* of numbers in the list does not change; each value in the list is transitioned separately. That means you can animate path data or a polygon's points, so long as you maintain the number of points and the types of path segments; Example 12-9 shows animations of both a `<polygon>` and a `<path>`.

Example 12-9. Example of animating path and polygon

http://oreillymedia.github.io/svg-essentials-examples/ch12/animate_path_poly.html

```
<polygon points="30 30 70 30 90 70 10 70"
  style="fill:#fcc; stroke:black">
  <animate id="animation"
    attributeName="points"
    attributeType="XML"
    to="50 30 70 50 50 90 30 50"
    begin="0s" dur="5s" fill="freeze" />
</polygon>

<path d="M15 50 Q 40 15, 50 50, 65 32, 100 40"
  style="fill:none; stroke: black" transform="translate(0,50)">
  <animate attributeName="d"
    attributeType="XML"
    to="M50 15 Q 15 40, 50 50, 32 65, 40 100"
    begin="0s" dur="5s" fill="freeze"/>
</path>
```

Specifying Multiple Values

All the animation elements presented so far give a starting (`from` or default) value and an ending (`to`) value, and let the computer calculate how to get from one to the other. It is possible to give specific intermediary values for an animation, allowing a single `<animate>` element to define complex sequences of changes. Instead of animating the color in Example 12-8 from light yellow to red, you can give a semicolon-separated list of `values` that the animation will use over the duration. Example 12-10 shows a circle that animates color using values of light yellow, light blue, pink, and light green.

Example 12-10. Animating color by specific values

http://oreillymedia.github.io/svg-essentials-examples/ch12/animating_values.html

```
<circle cx="50" cy="50" r="30"
  style="fill: #ff9; stroke:black;">
  <animate attributeName="fill"
```

```
    begin="2s" dur="4s" values="#ff9;#99f;#f99;#9f9"
    fill="freeze"/>
</circle>
```

The `values` attribute can also be used to make repeating animations alternate back and forth between two values, using the format `values="*start*; *end*; *start*;"`.

Timing of Multistage Animations

When an animation has multiple values, the duration of the animation (the `dur` attribute) is the time it takes to cycle through *all* the values. By default, the duration of the animation is divided into equal time periods for each transition. Example 12-10 used four color values, so there are three color transitions; the total duration is 4 seconds, so each transition lasts 4/3 of a second.

The `keyTimes` attribute allows you to divide the duration in other ways. The format of `keyTimes` is also a semicolon-separated list, and it must have the same number of entries as `values`. The first entry is always `0` and the last is always `1`; the intermediary times are expressed as decimal numbers between 0 and 1, representing the proportion of the animation duration that should pass by the time the corresponding value is reached.

More options for controlling timing are created with the `calcMode` attribute. There are four possible values for `calcMode`:

`paced`
: The SVG viewer will calculate the distance between subsequent values and divide up the duration so that the rate of change is constant (any `keyTimes` attribute will be ignored). Paced animation mode works with colors and simple numbers or lengths, but is not possible for lists of points or path data.

`linear`
: The default for `<animate>` elements; each transition will proceed at a steady pace, but the time alotted to each transition is equal (if `keyTimes` aren't specified) or is determined by `keyTimes`.

`discrete`
: The animation will jump from one value to the next without transitioning. If you animate a property that doesn't support transitions (like `font-family`), discrete mode will be used automatically.

`spline`
: The animation will accelerate and decelerate according to the values of the `keySplines` attribute; you can read more about it in the SVG specifications.

The <set> Element

All of these animations have modified values over time. Sometimes, particularly for non-numeric attributes or properties that can't transition, you simply want to change the value at a chosen point in the animation sequence.

For example, you might want an initially invisible text item to become visible at a certain time; there's no real need for both a `from` and `to`. Thus, SVG has the convenient shorthand of the `<set>` element, which needs only a `to` attribute and the proper timing information. Example 12-11 shrinks a circle down to 0, then reveals text 1/2 second after the circle is gone.

Example 12-11. Example of set element

http://oreillymedia.github.io/svg-essentials-examples/ch12/animation_set.html

```
<circle cx="60" cy="60" r="30" style="fill: #ff9; stroke: gray;">
  <animate id="c1" attributeName="r" attributeType="XML"
    begin="0s" dur="4s" from="30" to="0" fill="freeze"/>
</circle>

<text text-anchor="middle" x="60" y="60" style="visibility: hidden;">
  <set attributeName="visibility" attributeType="CSS"
    to="visible" begin="4.5s" dur="1s" fill="freeze"/>
  All gone!
</text>
```

The <animateTransform> Element

The `<animate>` element doesn't work with rotate, translate, scale, or skew transformations because they're all "wrapped up" inside the `transform` attribute. This is where the `<animateTransform>` element comes to the rescue. You set its `attributeName` to `transform`. The `type` attribute's value then specifies the transformation whose values should change (one of `translate`, `scale`, `rotate`, `skewX`, or `skewY`). The `from` and `to` values are specified as appropriate for the transform you're animating. As of this writing, most implementations currently support only `<animateTransform>` on the XML `transform` attribute rather than the CSS3 transformations.

Example 12-12 stretches a rectangle from normal scale to a scale of four times in the horizontal direction and two times in the vertical direction. Note that the rectangle is centered around the origin so it doesn't move as it scales; it is inside a `<g>` so it can be translated to a more convenient location. Figure 12-5 shows the beginning and end of the animation.

Example 12-12. Example of animateTransform

http://oreillymedia.github.io/svg-essentials-examples/ch12/animate_transform.html

```
<g transform="translate(100,60)">
  <rect x="-10" y="-10" width="20" height="20"
    style="fill: #ff9; stroke: black;">
    <animateTransform attributeType="XML"
      attributeName="transform" type="scale"
      from="1" to="4 2"
      begin="0s" dur="4s" fill="freeze"/>
  </rect>
</g>
```

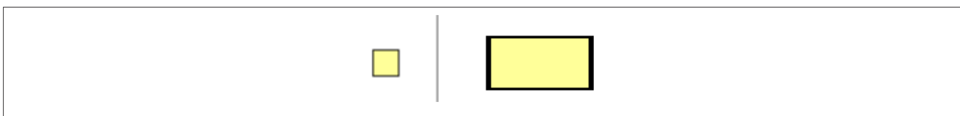


Figure 12-5. animateTransform—before and after

If you intend to animate more than one transformation, you must use the `additive` attribute. The default value of `additive` is `replace`, which replaces the specified transformation in the object being animated. This won't work in a series of transformations, because the second animation would override the first one. By setting `additive` to `sum`, SVG will accumulate the transformations. Example 12-13 stretches and rotates the rectangle. The before and after pictures are in Figure 12-6.

Example 12-13. Example of multiple animateTransform elements

http://oreillymedia.github.io/svg-essentials-examples/ch12/additive_transform.html

```
<rect x="-10" y="-10" width="20" height="20"
  style="fill: #ff9; stroke: black;">
  <animateTransform attributeName="transform" attributeType="XML"
    type="scale" from="1" to="4 2"
    additive="sum" begin="0s" dur="4s" fill="freeze"/>
  <animateTransform attributeName="transform" attributeType="XML"
    type="rotate" from="0" to="45"
    additive="sum" begin="0s" dur="4s" fill="freeze"/>
</rect>
```

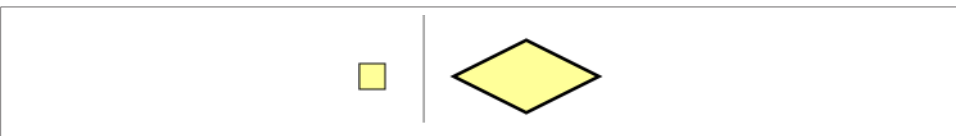
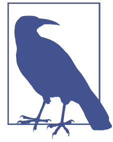


Figure 12-6. Multiple animateTransforms—before and after



You can also use additive="sum" to combine the effects of animation elements that control numerical and color attributes. If the animations are specified using to, adding them together causes subsequent animations to use the current value of the previous animation as their starting-point. If the animations use the by attribute to define their effects, or use *both* from and to, then the final value will be the sum of all the individual changes.

The <animateMotion> Element

You can cause an object to animate along a straight-line path by using translate with the <animateTransform> element. However, if you wanted to move the object in a more complicated pattern, you would need an extended series of transform animations timed to start one after another. The <animateMotion> element makes it easier to animate an object along an arbitrary path, whether a straight line or a series of overlapping loops.

If you want to use <animateMotion> for straight-line motion, you simply set the from and to attributes, assigning them each a pair of (*x*,*y*) coordinates. The coordinates specify the position where the (0,0) point of the shape's coordinate system will be moved, similar to how translate(x,y) works. Example 12-14 moves a grouped circle and rectangle from (0,0) to (60,30).

Example 12-14. Animation along a linear path

http://oreillymedia.github.io/svg-essentials-examples/ch12/linear_animateMotion.html

```
<g>
  <rect x="0" y="0" width="30" height="30" style="fill: #ccc;"/>
  <circle cx="30" cy="30" r="15" style="fill: #cfc; stroke: green;"/>
  <animateMotion from="0,0" to="60,30" dur="4s" fill="freeze"/>
</g>
```

Multiple points can be specified with values, but the motion will still be a series of straight lines. If you want a more complex path to follow, use the path attribute instead; its value is in the same format as the d attribute in the <path> element. Example 12-15, adapted from the SVG specification, animates a triangle along a cubic Bézier curve path.

Example 12-15. Animation along a complex path

http://oreillymedia.github.io/svg-essentials-examples/ch12/complex_animate_motion.html

```
<!-- show the path along which the triangle will move -->
<path d="M50,125 C 100,25 150,225, 200, 125"
  style="fill: none; stroke: blue;"/>

<!-- Triangle to be moved along the motion path.
   It is defined with an upright orientation with the base of
```

```
   the triangle centered horizontally just above the origin. -->
<path d="M-10,-3 L10,-3 L0,-25z" style="fill: yellow; stroke: red;">
  <animateMotion
    path="M50,125 C 100,25 150,225, 200, 125"
    dur="6s" fill="freeze"/>
</path>
```

As you can see in Figure 12-7, the triangle stays upright throughout its entire path.

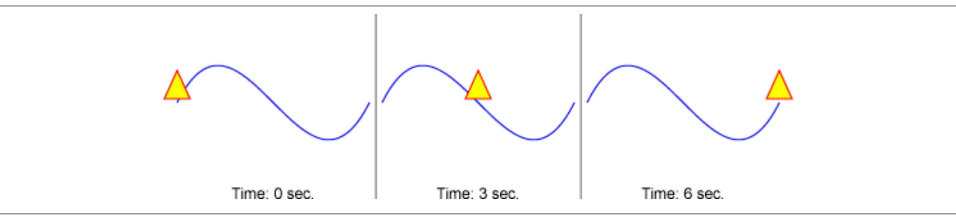


Figure 12-7. animateMotion along a complex path

If you would prefer that the object tilt so its *x*-axis is always parallel to the slope of the path, just add the `rotate` attribute with a value of `auto` to the `<animateMotion>` element. Example 12-16 shows the SVG, and Figure 12-8 shows screenshots taken at various stages of the animation.

Example 12-16. Animation along a complex path with auto-rotation

http://oreillymedia.github.io/svg-essentials-examples/ch12/animate_motion_rotate.html

```
<!-- show the path along which the triangle will move -->
<path d="M50,125 C 100,25 150,225, 200, 125"
  style="fill: none; stroke: blue;"/>

<!-- Triangle to be moved along the motion path.
   It is defined with an upright orientation with the base of
   the triangle centered horizontally just above the origin. -->
<path d="M-10,-3 L10,-3 L0,-25z" style="fill: yellow; stroke: red;" >
  <animateMotion
    path="M50,125 C 100,25 150,225, 200, 125"
    rotate="auto"
    dur="6s" fill="freeze"/>
</path>
```

Put simply, when you leave off the `rotate` attribute, you get the default value of 0, and the object acts like a hot-air balloon floating along the path. If you set `rotate` to `auto`, the object acts like a car on a roller coaster, tilting up and down as the path does.

You can also set `rotate` to a numeric value, which will set the rotation of the object throughout the animation. Thus, if you wanted an object rotated 45 degrees no matter what direction the path took, you'd use `rotate="45"`.

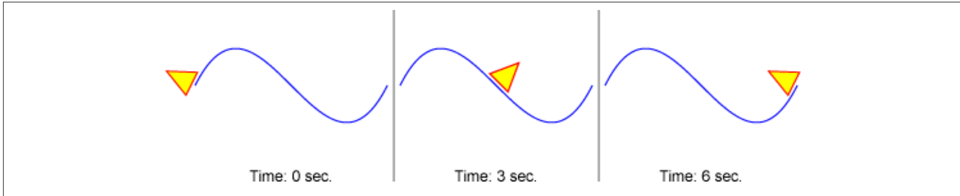


Figure 12-8. animateMotion along a complex path with auto-rotation

Example 12-16 drew the path in blue so it was visible, and then duplicated the path in the `<animateMotion>` element. You can avoid this duplication by adding an `<mpath>` element within the `<animateMotion>` element. The `<mpath>` will contain an `xlink:href` attribute that references the path you want to use. This also comes in handy when you have one path you wish to use to animate multiple objects. Here's the preceding example, rewritten as Example 12-17, using `<mpath>`.

Example 12-17. Motion along a complex path using mpath

```
<path id="cubicCurve" d="M50,125 C 100,25 150,225, 200, 125"
  style="fill: none; stroke: blue;"/>

<path d="M-10,-3 L10,-3 L0,-25z" style="fill: yellow; stroke: red;" >
  <animateMotion dur="6s" rotate="auto" fill="freeze">
    <mpath xlink:href="#cubicCurve"/>
  </animateMotion>
</path>
```

Specifying Key Points and Times for Motion

In Example 12-16, the triangle moved at a steady pace. This *paced* animation is the default for `<animateMotion>` (equivalent to `calcMode="paced"`); the amount of time it takes to move between subsequent points is directly proportional to the distance between them.[2]

In "Timing of Multistage Animations" on page 199, we introduced the `keyTimes` attribute, which can be used to control the rate an animation transitions between different values. You can use `keyTimes` for motion animation too, but if you're using a path instead

2. There's one exception to the distance rule for paced `<animateMotion>`: if your path has any moveto commands, these are counted as zero distance, meaning your object will immediately jump from the end of one subpath to the beginning of the next.

of a `values` list to define the motion, you need to specify key points along the path using (you guessed it) the `keyPoints` attribute.

Like `keyTimes`, `keyPoints` is a semicolon-separated list of decimal numbers. Each point represents how far along the path the object should have moved at the corresponding entry in the `keyTimes` list. Just as `keyTimes` ranges from 0 (beginning of animation) to 1 (end of animation), `keyPoints` ranges from 0 (beginning of the path) to 1 (end of the path). Example 12-18 shows the triangle moving more slowly as it goes uphill.

Although `keyTimes` must be given in order from 0 to 1, `keyPoints` may start or end in the middle of the path and go in either direction. However, the `keyPoints` and `keyTimes` lists must have the same number of entries, and you must set `calcMode="linear"` (or `"spline"`, but that's beyond the scope of this book).

Example 12-18. Variable speed motion along a path using keyPoints and keyTimes

http://oreillymedia.github.io/svg-essentials-examples/ch12/key_points.html

```
<path d="M-10,-3 L10,-3 L0,-25z" style="fill: yellow; stroke: red;" >
  <animateMotion
    path="M50,125 C 100,25 150,225, 200, 125"
    rotate="auto"
    keyPoints="0;0.2;0.8;1"
    keyTimes="0;0.33;0.66;1"
    calcMode="linear"
    dur="6s" fill="freeze"/>
</path>
```

Animating SVG with CSS

Modern browsers allow you to animate both HTML and SVG elements with CSS. This is a two-stage process. In the first stage, you select the element you want to animate and set the properties of the animation as a whole. In the second stage, you say which properties of the selected element are to change, and at what stages of the animation; these are defined in a `@keyframes` specifier.

Consider the following task: display a green star that fades to a white interior as its border becomes thicker; the effect is that the color is being "drained" into the border. Here is the SVG for the star:

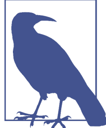
```
<svg width="200" height="200" viewBox="0 0 200 200">
  <defs>
    <g id="starDef">
      <path d="M 38.042 -12.361 9.405 -12.944 -0.000 -40.000
        -9.405 -12.944 -38.042 -12.361 -15.217 4.944
        -23.511 32.361 0.000 16.000 23.511 32.361 15.217 4.944 Z"/>
    </g>
  </defs>
```

```
  <use id="star" class="starStyle" xlink:href="#starDef"
    transform="translate(100, 100)"
    style="fill: #008000; stroke: #008000"/>
</svg>
```

Animation Properties

These are properties you'll set in the CSS for the element being animated:

- `animation-name` is the name of `@keyframes` specifier.
- `animation-duration` determines how long the animation should last; this is a number followed by a time unit as described in "How Time Is Measured" on page 194.
- `animation-timing-function` tells how intermediate values are calculated (e.g., should an animation ease in or out, or work in discrete steps).
- `animation-iteration-count` tells how many times to repeat an animation, with `infinite` looping continuously.
- `animation-direction` determines whether an animation should go in a forward or reverse direction, and whether it should alternate between the two or not.
- `animation-play-state` can be set to `running` or `paused`.
- `animation-delay` tells how long to wait to start the animation after the style is applied.
- `animation-fill-mode` tells what properties to use when the animation is not executing. This can be `forwards` (applies properties for the time the animation ended), `backwards` (applies properties for the time the animation began), or `both`.



In order to use these properties in a WebKit-based browser, you must, as of this writing, prefix them with `-webkit-`; thus, for example, `-webkit-animation-name` or `-webkit-animation-duration`.

Example 12-19 shows the CSS to set up the star animation. It will be repeated four times, with 2 seconds per iteration.

Example 12-19. CSS setup for animation

```
.starStyle {
  animation-name: starAnim;
  animation-duration: 2s;
  animation-iteration-count: 4;
  animation-direction: alternate;
  animation-timing-function: ease;
```

```
  animation-play-state: running;
}
```

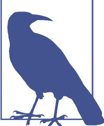
Setting Animation Key Frames

You set the properties to change at each stage of the animation by using the `@keyframes` media type, followed by the name of the animation being controlled. Inside the `@keyframes`, you list keyframe selectors, which are percentages that tell when properties should change. For each of those selectors, list the properties and values that the animation should take on. Example 12-20 shows the key frames for the star animation. For WebKit-based browsers, use `@-webkit-keyframes`. Figure 12-9 shows three stages of the animation.

Example 12-20. Key frame specification in CSS

http://oreillymedia.github.io/svg-essentials-examples/ch12/svg_css_anim1.html

```
@keyframes starAnim {
  0% {
    fill-opacity: 1.0;
    stroke-width: 0;
  }

  100% {
    fill-opacity: 0;
    stroke-width: 6;
  }
}
```



You can use `from` and `to` as synonyms for `0%` and `100%`.

Figure 12-9. Animation at beginning, middle, and end

Animating Movement with CSS

If you want to use pure CSS to animate movement, you can't use the `transform` attribute. Instead, you have to use CSS styles to translate, rotate, and scale your SVG.

Luckily, the CSS `transform` property's value looks very much like the SVG `transform` attribute, though there are differences, as noted in "CSS Transformations and SVG" on page 83. If, for example, you want an SVG element to be translated to (100,50), scaled by a factor of 1.5, and then rotated 90 degrees, the property would be as follows:

```
transform: translate(100px, 50px) scale(1.5) rotate(90deg);
```

Example 12-21 shows the key frames required for making the star move upward and rotate, and then descend to its starting point. Because the 100% key frame doesn't specify a `translate`, the star will return to its original position (specified in the SVG). This is why the 50% and 80% key frames, in addition to the 20% key frame, must specify the `translate` so that the star does not move vertically during that portion of the animation.

Example 12-21. Specifying transformations in CSS

```
@keyframes starAnim  {
  0% {
    fill-opacity: 1.0;
    stroke-width: 0;
  }

  20% {
    transform: translate(100px, 50px)
  }

  50% {
    transform: translate(100px, 50px) rotate(180deg)
  }

  80% {
    transform: translate(100px, 50px) rotate(360deg)
  }

  100% {
    fill-opacity: 0.0;
    stroke-width: 6;
  }
}
```

Create your own keyframes, and experiment with the timing properties, with the online example:

http://oreillymedia.github.io/svg-essentials-examples/ch12/svg_css_anim2.html

At the time of this writing, browsers tend to be buggy and inconsistent when applying CSS animations and transitions to SVG graphics that are duplicated by `<use>` elements.

CHAPTER 13
Adding Interactivity

To this point, you, the author of the SVG document, have made all the decisions about a graphic. You decide what a static image should look like, and if there are any animations, you decide when they start and stop. In this chapter, you will see how to hand some of that control over to the person who is viewing your document.

The lowest level of interactivity is *declarative interactivity*—animation or other style changes created by telling the browser what should happen under certain situations, without directly controlling the effect with a script. SVG provides a limited set of built-in interactive states.

Using Links in SVG

The easiest sort of interactivity to provide is linking, accomplished with the `<a>` element. By enclosing a graphic in this element, it becomes active; when clicked, you go to the URL specified in the `xlink:href` attribute. You can link to another SVG file or, depending upon your environment, a web page. In Example 13-1, clicking the word "Cat" will link to an SVG drawing of a cat; clicking the red, green, and blue shapes will link to the World Wide Web Consortium's SVG page. All the items within the second link are individually linked to the same destination, not the entire bounding box. When you test this example and move the cursor between the shapes, you will see that those areas do not respond to clicks.

Example 13-1. Links in SVG

http://oreillymedia.github.io/svg-essentials-examples/ch13/svg_link.svg

```
<a xlink:href="cat.svg">
  <text x="100" y="30" style="font-size: 12pt;">Cat</text>
</a>

<a xlink:href="http://www.w3.org/SVG/">
```

```
  <circle cx="50" cy="70" r="20" style="fill: red;"/>
  <rect x="75" y="50" width="40" height="40" style="fill: green;"/>
  <path d="M120 90, 140 50, 160 90 Z" style="fill: blue;"/>
</a>
```

In the `<use>` element, `xlink:href` specifies a resource that becomes part of your graphic, as described in "The <use> Element" on page 63. For the `<a>` element, the `xlink:href` attribute specifies a different resource to jump to.

Links in HTML are recognizable by color and underlining effects. Figure 13-1 shows the results of Example 13-1; there's nothing to tell you that the graphics are actually linked, unless you notice the change of the cursor from an arrow to a "hand" icon. Keyboard users have the same difficulty: some browsers outline elements if they have keyboard focus, but others do not. You can use CSS pseudoclasses to give users some feedback about the interactive elements of your graphic. Like a CSS class, a *pseudo-class* is used to apply styles to select instances of an element; unlike true classes, they are applied automatically, not assigned in the `class` attribute.[1]

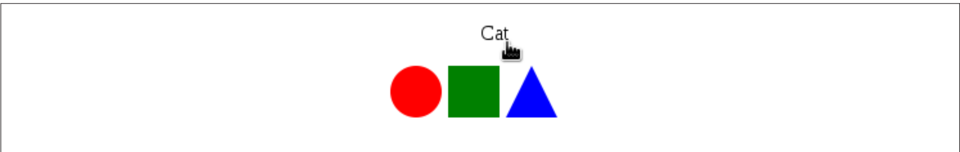


Figure 13-1. A hyperlinked SVG, the results of Example 13-1

The `:hover` pseudoclass applies when the main mouse pointer is over an element, while the `:focus` pseudoclass applies when an element has the keyboard focus. You can often use the same styles for both *`element`*`:hover` and *`element`*`:focus`, because both indicate the potential for user action.

Example 13-2 uses the same graphics as Example 13-1, but now the links give feedback when they are hovered or focused. The text will become bold and underlined, and the shapes will get a light blue border.

Example 13-2. Links in SVG highlighted with CSS

http://oreillymedia.github.io/svg-essentials-examples/ch13/svg_css_link.svg

```
<style type="text/css"><![CDATA[

    a.words:hover, a.words:focus {
       text-decoration: underline;
       font-weight:bold;
    }
```

1. The Apache Batik SVG viewer, version 1.7, does not support CSS pseudoclass selectors.

```
    a.shapes:hover, a.shapes:focus {
      stroke: #66f;
      stroke-width: 2;
      outline: none; /* override default focus formatting */
    }
    ]]>
  </style>

  <a class="words" xlink:href="cat.svg">
    <text x="100" y="30" style="font-size: 12pt;">Cat</text>
  </a>

  <a class="shapes" xlink:href="http://www.w3.org/SVG/">
    <circle cx="50" cy="70" r="20" style="fill: red;"/>
    <rect x="75" y="50" width="40" height="40" style="fill: green;"/>
    <path d="M120 90, 140 50, 160 90 Z" style="fill: blue;"/>
  </a>
```

Controlling CSS Animations

What if you wanted more dynamic user feedback? CSS animations are defined as style properties, so they can also be controlled by pseudoclasses. Example 13-3 starts an animation when you hover the mouse over the shapes.

Example 13-3. Animating a link with :hover

http://oreillymedia.github.io/svg-essentials-examples/ch13/anim_css_link.svg

```
<style type="text/css"><![CDATA[

  a.animatedLink {
    animation-name: animKeys;
    animation-iteration-count: infinite;
    animation-duration: 0.5s;
    animation-direction: alternate;
    animation-play-state: paused;
  }

  a.animatedLink:hover {
    animation-play-state: running;
  }

  @keyframes animKeys {
    0% {fill-opacity: 1.0;}
    100% {fill-opacity: 0.5;}
  }
  ]]>
  </style>

<a class="animatedLink" xlink:href="http://www.w3.org/SVG/">
  <circle cx="50" cy="70" r="20" style="fill: red;"/>
```

```
  <rect x="75" y="50" width="40" height="40" style="fill: green;"/>
  <path d="M120 90, 140 50, 160 90 Z" style="fill: blue;"/>
</a>
```

The example isn't a great demonstration of user design: if you move the mouse out of the link area while the animation is going on, it simply pauses but doesn't revert back to full opacity. If you're interested in CSS animations, the latest draft specifications describe all the possibilities.

User-Triggered SMIL Animations

CSS animations are limited in what sort of changes they can create and what sort of events they can respond to. If you are using SMIL animation elements in your SVG, you can use an alternative format for the `begin` and `end` attributes to declare that they should respond to user actions.

The format for an interactive animation timing attribute is `elementID.eventName`. The element referenced in the ID does not have to be the same element that is being animated.

Interaction using the SMIL begin and end attributes is *event based*: once an animation is started, it will continue until the duration of the animation is complete or an ending event occurs. In contrast, interaction using CSS pseudoclasses is *state based*: the style or animation is only applied for so long as the state is true. For example, the CSS `#myElement:hover` pseudoclass selector describes the state of the element with ID `myElement` when the mouse pointer is over it; to define an animation to occur during the same period, you would set the animation attributes `begin="myElement.mouseover"` and `end="myElement.mouseout"`.[2]

For greater control, you can optionally add a time offset, of the form `elementID.eventName + offset`. The offset is specified in the same format as for other SMIL animation timing attributes (see "How Time Is Measured" on page 194, in Chapter 12), with a unit like `1.5min` or as a stopwatch time like `01:30`. You could even use a negative time offset, but since there is no such thing as Psychic Vector Graphics, the animation won't actually start before the event occurs. Instead, as soon as the event occurs the computer will skip the first part of the animation (which "should" have occurred before the event) and continue on with the rest of the animation.

Example 13-4 creates a trapezoid and a button. When you click the button, the trapezoid rotates 360 degrees. Screenshots, before and after a click, are shown in Figure 13-2.

2. If you're familiar with JavaScript event handling, you'll recognize that the event names are the same as used in DOM event handling. If you're not familiar with JavaScript and DOM events, keep reading.

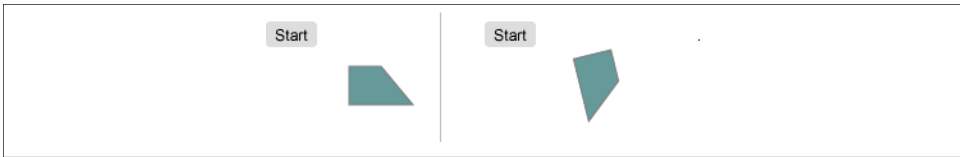


Figure 13-2. Screenshot of two stages of scripting with animation

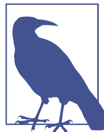
Example 13-4. Interactive Animation

http://oreillymedia.github.io/svg-essentials-examples/ch13/smil_event_animation.svg

```
<g id="button">     ❶
  <rect x="10" y="10" width="40" height="20" rx="4" ry="4"
    style="fill: #ddd;"/>
  <text x="30" y="25"
    style="text-anchor: middle; font-size: 8pt">Start</text>
</g>

<g transform="translate(100, 60)">
  <path d="M-25 -15, 0 -15, 25 15, -25 15 Z"
    style="stroke: gray; fill: #699;">

    <animateTransform id="trapezoid" attributeName="transform"
      type="rotate" from="0" to="360"
      begin="button.click"
      dur="6s"/>   ❷
  </path>
</g>
```

❶ The start button is a simple rounded rectangle with text. The entire group gets the `id`.

❷ Instead of giving the begin time for the animation in terms of seconds, we begin whenever a `click` event is detected on the `button` object.



It is often easier to design the SVG first and add the scripting later. One advantage of this method is that you can see if the base drawing looks good before you start making it react to events.

Scripting SVG

The next step up from these declarative animations—and it's a big step—is scripting. You can write a program in ECMAScript to interact with an SVG graphic. (ECMAScript is the standardized version of what is commonly called JavaScript, as defined by ECMA, an organization formerly known as the European Computer Manufacturer's

Association.) If you're new to ECMA/JavaScript—or programming in general—you'll want to read Appendix C.

As the SVG viewer reads the markup in an SVG document, it creates a tree of *nodes*, which are objects in memory that correspond to the structure and content of the markup. This is the *Document Object Model*, and it is accessible to your scripts.

The first thing you need to do in order to deal with the DOM is to access the nodes. The main function you will probably use to deal with the DOM is `document.getElementById(idString)`. This function takes a string that is the `id` of an SVG element and returns a reference to that element's node in the DOM. If you want all of the elements in a document that have a particular tag name (the tag name is the "svg" in `<svg>` or "rect" in `<rect>`), you can use another `document` method, `getElementsByTagName(name)`; this returns an array of nodes.

Once you have an element's node, you can:

- Read its attributes by calling `element.getAttribute(attributeName)`, which returns the attribute value as a string.
- Change an attribute's value by calling `element.setAttribute(name, newValue)`; if the attribute with the given *name* does not exist, it will be created.
- Remove attributes by calling `element.removeAttribute(name)`.

You could modify inline styles using `element.setAttribute("style", newStyleValue)`, but this overwrites *all* styles on the element. Instead, you can work with the `element.style` property. Use:

- `element.style.getPropertyValue(propertyName)` to access a specific style,
- `element.style.setProperty(propertyName, newValue, priority)` to change it (`priority` is usually null, but could be "important"), and
- `element.style.removeProperty(propertyName)` to delete it.

If you *do* want to set all styles at once, then you can directly modify `element.style.cssText`, which is a string representation of all the styles in `property-name: value` format.[3]

If you need to access or modify the text content of any node, use the `element.textContent` property. When you read this property, it returns the

3. Most browsers also allow you get or set CSS properties using the form `element.style.propertyName` or `element.style["property-name"]`. However, this isn't part of the CSS object model standards and isn't supported in some other SVG viewers, or even consistently between browsers.

concatenated text of all of the node's descendants. If you set it, you will replace any descendant nodes with a single text block.[4]

Example 13-5 uses these functions to access the attributes of an SVG element, display them in text format, and modify an attribute. (This isn't interactive, but bear with us; we're building suspense in the plot line.)

Example 13-5. Accessing SVG with the DOM

http://oreillymedia.github.io/svg-essentials-examples/ch13/basic_dom_example.svg

```
<svg width="300" height="100" viewBox="0 0 300 100"
  xmlns="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink">

  <title>Accessing Content in SVG</title>

  <rect id="rectangle" x="10" y="20" width="30" height="40"
    style="stroke:gray; fill: #ff9; stroke-width:3"/> ❶
  <text  id="output" x="10" y="80" style="font-size:9pt"></text>

  <script type="application/ecmascript">
  // <![CDATA[ ❷
    var txt = document.getElementById("output"); ❸
    var r = document.getElementById("rectangle");
    var msg =  r.getAttribute("x") + ", " + ❹
      r.getAttribute("y") + " " +
      r.style.getPropertyValue("stroke") + " " +
      r.style.getPropertyValue("fill");
    r.setAttribute("height", "30"); ❺
    txt.textContent= msg; ❻
    // ]]>
  </script>
</svg>
```

❶ In order to easily access an element from a script, give it a unique `id`.

❷ The `<![CDATA[` is used to ensure that any stray < or > signs are not interpreted as markup.

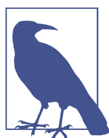
❸ Select elements from the document by `id`, and save the results in variables.

❹ The results of `getAttribute()` and `style.getPropertyValue()` are strings; these are concatenated together with + to create the message string.

❺ This changes the height of the rectangle to convert it to a square.

❻ Finally, set the content of the `<text>` element to display the attributes.

4. If you're familiar with using the `.innerHTML` property to modify the combined text and markup of all descendants of an element, be warned that this property is only defined in the specifications for nodes of type `HTMLElement`, and many browsers and SVG viewers do not support it on SVG elements.



The script in Example 13-5 is included in the SVG file *after* the `<rect>` and `<text>` elements that it uses. This ensures that the elements exist in the DOM before the script is run.

Events: An Overview

Interaction occurs when graphic objects respond to events. There are several categories of events. The text for many of these descriptions comes directly from the World Wide Web Consortium's specification.

User interface events
: The `focusIn` and `focusOut` events occur when an element receives or loses focus, such as selecting or unselecting text. The `activate` element occurs when an element is activated through a mouse click or keypress.

Mouse events
: The `mousedown` and `mouseup` events occur when a pointing device button is pressed or released on an element. If the screen location for these events is the same, then a `click` event is generated.

The `mouseover`, `mousemove`, and `mouseout` events occur when the pointing device is moved over an element, moved while over an element, and moved away from an element.

Mutation events
: The SVG viewer will generate events when the DOM changes (by another script); for example, `DOMNodeInserted` occurs when a node has been added as a child of another node; `DOMAttrModified` occurs when an attribute has been modified on a node. This book will not cover these events in detail.

Document events
: The `SVGLoad` event is triggered when the SVG viewer has fully parsed a document and is ready to act upon it (e.g., display it on a device). `SVGUnload` occurs when a document is removed from a window. `SVGAbort` occurs when page loading is stopped before loading completes; `SVGError` occurs when an element does not load properly or an error occurs during script execution.

The `SVGResize`, `SVGScroll`, and `SVGZoom` events occur when the viewer changes the document in the way that the name suggests.

Animation events
: The SVG viewer generates `beginEvent`, `endEvent`, and `repeatEvent` when an animation element begins, ends, or repeats; `repeatEvent` is not generated for the first iteration.

Key events
: There are no events built into SVG for keypresses, but some viewers may support nonstandard `keydown` and `keyup` events.

Listening for and Responding to Events

To allow an object to respond to an event, you must first tell the object to listen for the event. You do this by calling the `addEventListener()` function. This function has two required arguments. The first is a string with the name of the type of event you want to listen for. The second argument is the name of a function that will handle the event. An optional third argument is a boolean that tells whether you want to respond to the event when the viewer is passing the event down the DOM hierarchy from parent elements to children to find the specific target ("capture" stage). A value of `false` (the usual) causes your listener to wait to handle the event until after any child elements have dealt with it; this is the "bubbling" stage, as events float to the top of the DOM tree.[5]

The function that handles the event takes one argument: an event object that contains information about the event that triggered the call. The most important property of the event object is the `target` property, which is the object to which the event was dispatched. Other important event properties are `clientX` and `clientY`, which give the coordinates at which the event occurred relative to the DOM implementation's client area, which is the area occupied by the entire *.svg* file or web page.

Example 13-6 adds listeners for the `mouseover`, `mouseout`, and `click` events to a circle. Moving the mouse in and out of the circle will cause its radius to grow or shrink; clicking the mouse will increase or decrease the circle's stroke width.

Example 13-6. Adding mouse movement listeners

http://oreillymedia.github.io/svg-essentials-examples/ch13/simple_event.svg

```
<circle id="circle" cx="50" cy="50" r="20"
  style="fill: #ff9; stroke:black; stroke-width: 1"/>

<script type="application/ecmascript"><![CDATA[

  function grow(evt) {
    var obj = evt.target;
    obj.setAttribute("r", "30");
  }

  function shrink(evt) {
    this.setAttribute("r", "20");
  }
```

5. This is a very simplistic definition. See the DOM Events specification for the full details.

```
    function reStroke(evt) {
      var w = evt.target.style.getPropertyValue("stroke-width");
      w = 4 - parseFloat(w); /* toggle between 1 and 3 */
      evt.target.style.setProperty("stroke-width", w, null);
    }

    var c = document.getElementById("circle");
    c.addEventListener("mouseover", grow);
    c.addEventListener("mouseout", shrink);
    c.addEventListener("click", reStroke);
    // ]]>
  </script>
```

The first event handler, `grow()`, uses `evt.target` to access the element that received the event, and stores it in a separate variable. The second event handler, `shrink()`, uses the reserved word `this` to refer to the element that is attached to the event listener (which in this case, but not always, is the same as the event target). The last event handler, `reStroke()`, again uses `evt.target`, but without the use of a temporary variable.

While we *could* have used `c` instead of `evt.target` (because it is the only element with a listener), this would have been a terrible idea, because you will often need to attach the same event handler to many different targets, as in the next example.

Changing Attributes of Multiple Objects

Sometimes you will want an event that occurs on object A to affect attributes of both object A and some other object B. Example 13-7 presents possibly the world's crudest example of SVGcommerce. Figure 13-3 shows a T-shirt whose size changes as the user clicks each labeled button. The currently selected size button is highlighted in light yellow.

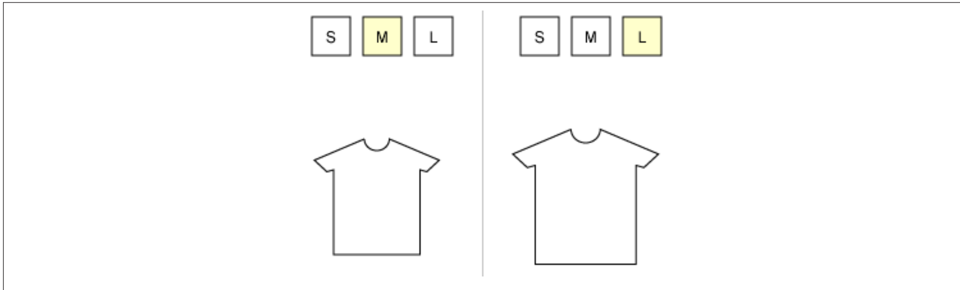


Figure 13-3. Screenshots of different selections

Example 13-7. Changing multiple objects in a script

http://oreillymedia.github.io/svg-essentials-examples/ch13/shirt1.svg

The XML code:

```
<svg width="400" height="250" viewBox="0 0 400 250"
  xmlns="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  onload="init(evt)"> ❶

  <defs>
    <style type="text/css" > <![CDATA[
      /* style rules will go here */
    ]]></style>
    <script type="application/ecmascript"> <![CDATA[
      /* script will go here */
    ]]></script>

    <path id="shirt-outline"
      d="M -6 -30 -32 -19 -25.5 -13 -22 -14 -22 30 23 30
        23 -14 26.5 -13 33 -19 7 -30
        A 6.5 6 0 0 1 -6 -30"/> ❷
  </defs>

  <g id="shirt" >
    <use xlink:href="#shirt-outline" x="0" y="0"/>
  </g>

  <g id="scale0" >
    <rect x="100" y="10" width="30" height="30" />
    <text x="115" y="30">S</text>
  </g>

  <g id="scale1" class="selected"> ❸
    <rect x="140" y="10" width="30" height="30" />
    <text x="155" y="30">M</text>
  </g>
```

```
  <g id="scale2" >
    <rect x="180" y="10" width="30" height="30" />
    <text x="195" y="30">L</text>
  </g>
</svg>
```

❶ As soon as the document finishes loading, the `SVGLoad` event occurs, and the `onload` handler will call the init function, passing it the event information. Many scripts will use this event handler to make sure all their variables are set up properly. This allows you to put the `<script>` before the SVG elements to be manipulated. Attributes of the form `on`*`eventname`* can be used to listen to many events, but are discouraged (in favor of `addEventListener()`) because they mix your scripting functionality with your XML structure; document loading is an exception.

❷ The shirt outline is centered at (0, 0), so that it will scale from the center, and is then positioned with transformations in the script.

❸ The medium button is selected initially.

The styles:

```
svg { /* default values */
   stroke: black;
   fill: white;
}
g.selected rect {
   fill: #ffc; /* light yellow */
}
text {
   stroke: none;
   fill:black;
   text-anchor: middle;
}
```

The "selected" class is used to indicate which size option is active, by filling the button in light yellow.

The script:

```
var scaleChoice = 1;  ❶
var scaleFactor = [1.25, 1.5, 1.75];

function init(evt) { ❷
  var obj;
  for (var i = 0; i < 3; i++) {
    obj = document.getElementById("scale" + i);
    obj.addEventListener("click", clickButton, false);
  }
  transformShirt();
}
```

```
function clickButton(evt) {
  var choice = evt.target.parentNode; ❸
  var name = choice.getAttribute("id");
  var old = document.getElementById("scale" + scaleChoice);
  old.removeAttribute("class"); ❹
  choice.setAttribute("class", "selected");

  scaleChoice = parseInt(name[name.length - 1]); ❺
  transformShirt();
}

function transformShirt() { ❻
  var factor = scaleFactor[scaleChoice];
  var obj = document.getElementById("shirt");
  obj.setAttribute("transform",
    "translate(150, 150) " +
    "scale(" + factor + ")");
  obj.setAttribute("stroke-width",
    1 / factor);
}
```

❶ This script works by keeping track of which button (S, M, or L) has been chosen, and indexing into the corresponding entry in the `scaleFactor` array. The default is index number one, medium.

❷ The `init()` function gets each rectangle-and-text `<g>` and tells it to listen for a `click` event. The function then displays the shirt at its current size.

❸ A click could occur on either the text or the interior of the rectangle. Using `parentNode` puts the `<g>` object into variable `choice`.

❹ Remove the "selected" class from the button `<g>` corresponding to the previously selected size, and add it to the newly selected `<g>`.

❺ Extract the number at the end of the button group's `id`; this is the new `scaleChoice`. It's stored in a global variable, accessible by the `transformShirt()` function.

❻ The shirt is resized and positioned by setting its `transform` attribute. The stroke width is rescaled by the inverse factor, so that it appears to stay the same when the shirt is scaled up and down.

Dragging Objects

Let us expand this example by adding "sliders" that can be dragged to set the color of the shirt, as shown in Figure 13-4.

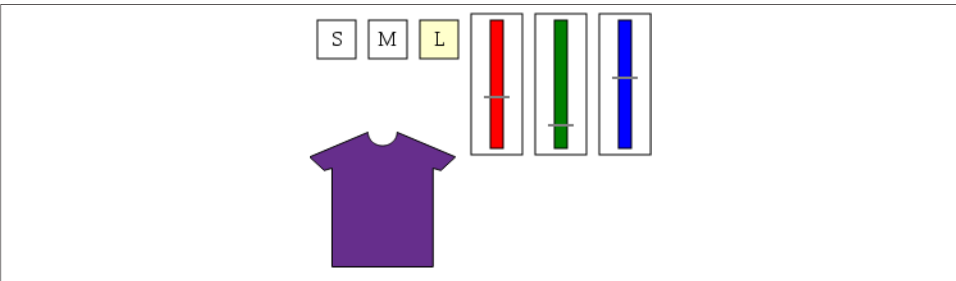


Figure 13-4. Screenshot of color sliders

You can experiment with the slider in the online example:

http://oreillymedia.github.io/svg-essentials-examples/ch13/drag_objects.svg

This script needs some more global variables. The first of these, `slideChoice`, tells which slider (0, 1, or 2) is currently being dragged; its initial value is -1, meaning no slider is active. The script also uses an array called `rgb` to hold the percent of red, green, and blue; the initial values are all 100, because the shirt is initially white:

```
var slideChoice = -1;
var rgb = [100, 100, 100];
```

Next, draw the sliders themselves. The color bar and the slide indicator are drawn on a white background, and they are grouped together. The `id` attribute goes on the indicator `<line>` element, because its *y*-coordinate will be changing. The event handlers will be attached to the enclosing `<g>` element. The group will then capture the mouse events that happen on any of its child elements (this is why we drew the white rectangle; the mouse will still track even if you drag outside the colored bar):

```
<g id="sliderGroup0" transform="translate( 230, 10 )">
  <rect x="-10" y="-5" width="40" height="110"/>
  <rect x="5" y="0" width="10" height="100" style="fill: red;"/>
  <line id="slide0" class="slider"
    x1="0" y1="0" x2="20" y2="0" />
</g>

<g id="sliderGroup1" transform="translate( 280, 10 )">
  <rect x="-10" y="-5" width="40" height="110"/>
  <rect x="5" y="0" width="10" height="100" style="fill: green;"/>
  <line id="slide1" class="slider"
    x1="0" y1="0" x2="20" y2="0" />
</g>

<g id="sliderGroup2" transform="translate( 330, 10 )">
  <rect x="-10" y="-5" width="40" height="110"/>
  <rect x="5" y="0" width="10" height="100" style="fill: blue;"/>
  <line id="slide2" class="slider"
```

```
    x1="0" y1="0" x2="20" y2="0" />
</g>
```

New style rules handle everything except for the slider-specific colors:

```
line.slider {
   stroke: gray;
   stroke-width: 2;
}
```

In the `init()` function, add three event listeners to each slider group:

```
obj = document.getElementById("sliderGroup" + i);
obj.addEventListener("mousedown", startColorDrag, false);
obj.addEventListener("mousemove", doColorDrag, false);
obj.addEventListener("mouseup", endColorDrag, false);
```

The corresponding functions are as follows.

```
function startColorDrag(evt) { ❶
  var sliderId = evt.target.parentNode.getAttribute("id");
  endColorDrag( evt );
  slideChoice = parseInt(sliderId[sliderId.length - 1]);
}

function endColorDrag(evt) { ❷
  slideChoice = -1;
}

function doColorDrag(evt) { ❸
  var sliderId = evt.target.parentNode.getAttribute("id");
  chosen = parseInt(sliderId[sliderId.length - 1]);

  if (slideChoice >= 0 && slideChoice == chosen) {  ❹

    var obj = evt.target; ❺
    var pos = evt.clientY - 10;
    if (pos < 0) { pos = 0; }
    if (pos > 100) { pos = 100; }

    obj = document.getElementById("slide" + slideChoice); ❻
    obj.setAttribute("y1", pos);
    obj.setAttribute("y2", pos);

    rgb[slideChoice] = 100-pos; ❼

    var colorStr = "rgb(" + rgb[0] + "%," +  ❽
      rgb[1] + "%," + rgb[2] + "%)";
    obj = document.getElementById("shirt");
    obj.style.setProperty("fill", colorStr, null);
  }
}
```

❶ `startColorDrag(evt)` is called on mousedown. It stops dragging the current slider (if any) and sets the current slider to the one specified (`0` = red, `1` = green, `2` = blue).

❷ `endColorDrag(evt)` is called on mouseup or by other functions. It sets the slider choice to -1, indicating that no slider is being dragged. No access to the event is needed for this function.

❸ `doColorDrag(evt)` is called on mousemove. It uses both the event's `target` property (to determine which slider is being dragged) and the `clientY` property to determine the mouse position relative to the top of the SVG.

❹ Check that a slider is active and that the event is on the chosen slider.

❺ Get the slider indicator line object, and the mouse position (adjusted by the position of the top of the color bar). Clamp the position values to the range 0–100.

❻ Move the slider line to the new mouse position.

❼ Calculate the new color value for this slider.

❽ Compile the color values into `rgb()` notation and change the shirt's color accordingly.

There's only one minor point to take care of—the document will respond to an `onmouseup` only if it occurs within the slider area. So, if you click the mouse on the red color bar, drag the mouse down to the shirt, then release the mouse button, the document will be unaware of it. When you then move the mouse over the red slider again, it will still follow the mouse. To solve this problem, we insert a transparent rectangle that completely covers the viewport, and it responds to a `mouseup` event by calling `stopColorDrag`. It will be the first, and therefore bottom-most object in the graphic. To make the rectangle as unobtrusive as possible, it will be set to `style="fill: none;"`. "But wait," you interject. "A transparent area cannot respond to an event!" No, ordinarily it can't, but you can set the `pointer-events` attribute to `visible`, meaning that an object can respond to events as long as it is visible, no matter what its opacity:[6]

```
<rect id="eventCatcher" x="0" y="0" width="400" height="300"
  style="fill: none;" pointer-events="visible" />
```

The `init()` function is modified to add the appropriate event listener:

```
document.getElementById("eventCatcher").
  addEventListener("mouseup", endColorDrag, false);
```

6. Other values for `pointer-events` let you respond to an object's events in the filled areas only (`fill`), outline areas only (`stroke`), or the fill and outline together (`painted`), whether visible or not. Corresponding attribute values of `visibleFill`, `visibleStroke`, and `visiblePainted` take the object's visibility into account as well.

Interacting with an HTML Page

There are two ways to put interactive SVG into an HTML document, as described in Chapter 2. If you have a small amount of SVG, just put it directly into the HTML. If you have a large SVG graphic, you can include it by using the `<object>` element. The following markup shows how you would do this for the preceding SVG example:

```
<object id="externalShirt" data="shirt_interact.svg"
  type="image/svg+xml">
  <p>Alas, your browser does not support SVG.</p>
</object>
```

The relevant attributes are the `data` source for the graphic (a URL—in this case, a relative pathname) and the `type` attribute, which will be `image/svg+xml`. The HTML between the opening and closing `<object>` tags is only displayed if the object cannot be loaded. You can now add code to the SVG script and the HTML page's script so they can communicate with one another; the `id` attribute will come into play for that.

The web page will have a form that lets users type in the red, green, and blue percentages. The values they enter will be reflected in the sliders. If users adjust the sliders, the values in the form fields will be updated accordingly.

Here is the HTML document, with references to the (as yet unwritten) `updateSVG()` function. This function will take the input field number and the value currently within the input field as its arguments:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8">
  <title>SVG and HTML</title>
  <style type="text/css">
    /* make form entries begin on a new line */
    label {display: block;}
    h1 {font-size: 125%;}
  </style>
  <script type="text/javascript">
    /* script goes here */
  </script>
</head>

<body>
<h1>SVG and HTML</h1>
<div style="text-align:center">
  <object id="shirt" data="shirt_interact.svg"
    type="image/svg+xml">
    <p>Alas, your browser cannot load this SVG file.</p>
  </object>

  <form id="rgbForm">
```

```
    <label>Red: <input id="fld0" type="text" size="5" value="100"
      onchange="updateSVG(0, this.value)" />% </label>
    <label>Green: <input id="fld1" type="text" size="5" value="100"
      onchange="updateSVG(1, this.value)" />% </label>
    <label>Blue: <input id="fld2" type="text" size="5" value="100"
      onchange="updateSVG(2, this.value)" />%</label>
  </form>
</div>
</body>
</html>
```

In the interest of keeping the code sample short, we've used poor coding style and mixed our script in with our HTML. The attribute `onchange="updateSVG(0, this.value)"` is equivalent to adding a listener for the `<input>` element's `change` event, and setting it to run the command `updateSVG(0, this.value)`.

Here is the script that goes into the head of the HTML document. Function `updateSVG` checks to see that the input value is an integer (it will discard any decimal part), and, if so, calls function `setShirtColor`. This is actually a reference to a function that exists in the SVG document, and it will be the SVG document's responsibility to connect the function to this HTML reference. (You will see this happen later in the chapter.)

Function `updateHTMLField` will be called from the SVG document's script. It will receive a form field number and a percent value, which it will display in the appropriate form field:

```
function updateSVG(which, amount) {
  amount = parseInt(amount);
  if (!isNaN(amount) && window.setShirtColor) {
    window.setShirtColor(which, amount);
  }
}

function updateHTMLField(which, percent) {
  document.getElementById("fld" + which).value = percent;
}
```

Now you have to modify the SVG document. There are now two ways to set shirt color —from the slider and from the HTML. Thus, the first task is to separate the setting of the shirt color from the slider dragging. The modified `doColorDrag(evt)` function detects the active slider and calculates the position of the slider, but it then calls a new `svgSetShirtColor` function to implement the change:

```
function doColorDrag(evt) {
  if (slideChoice >= 0) {
    var sliderId = evt.target.parentNode.getAttribute("id");
    chosen = parseInt(sliderId[sliderId.length - 1]);
    if (slideChoice == chosen) {
```

```
        svgSetShirtColor(slideChoice, 100 - (evt.clientY - 10));
      }
    }
  }
```

Function `svgSetShirtColor` will do what the remainder of `doColorDrag` used to do, with two major differences. It uses the slider number it is given as the first parameter, not the global `slideChoice` variable. Also, it takes the percentage as its second parameter. These are the sort of changes you have to make when you decide to modularize simple code that was written for an ad hoc example:

```
function svgSetShirtColor(which, percent) {
  var obj;
  var colorStr;
  var newText;

  if (percent < 0) { percent = 0; } ❶
  if (percent > 100) { percent = 100; }

  obj = document.getElementById("slide" + which); ❷
  obj.setAttribute("y1", 100 - percent);
  obj.setAttribute("y2", 100 - percent);
  rgb[which] = percent;

  colorStr = "rgb(" + rgb[0] + "%," + ❸
    rgb[1] + "%," + rgb[2] + "%)";
  obj = document.getElementById("shirt");
  obj.style.setProperty("fill", colorStr, null);
}
```

❶ You still have to check that the percent value is within the correct range.

❷ The slider line is moved, to the new position.

❸ The color-changing code is the same.

Now, use the reserved word `parent` in the `init` function to connect the SVG document's `svgSetShirtColor` function to the HTML page's `setShirtColor` reference. This works because when one document is embedded in another, the `parent` global variable in the child document will be a reference to the other document's `window` object. As `setShirtColor` will be a property of the window in which the web page is running, the HTML document will be able to access it. The following code accomplishes the HTML to SVG communication. Before using the `parent` variable, it tests to confirm that the parent object exists (meaning that the SVG is actually embedded in another document):

```
function init( evt ) {
  // add event listeners
  if (parent) {
        parent.setShirtColor = svgSetShirtColor;
   }
```

```
    transformShirt();
  }
```

The last step is to communicate from SVG back to HTML if the user decides to choose colors with the slider. Rather than continuously update the HTML fields, we made the design decision to update the HTML when the mouse drag stops. Add the boldface code to function `endColorDrag`. If a slider was being moved, this sends the slider number and its value back to the updateHTMLField function in the parent web browser window (if it exists):

```
  function endColorDrag( ) {

    if (slideChoice >= 0) {
      if (parent)
        parent.updateHTMLField(slideChoice, rgb[slideChoice]);
    }

    // In any case, nobody's being dragged now
    slideChoice = -1;
  }
```

The result is shown in Figure 13-5; the screenshot has been edited to eliminate unnecessary whitespace.

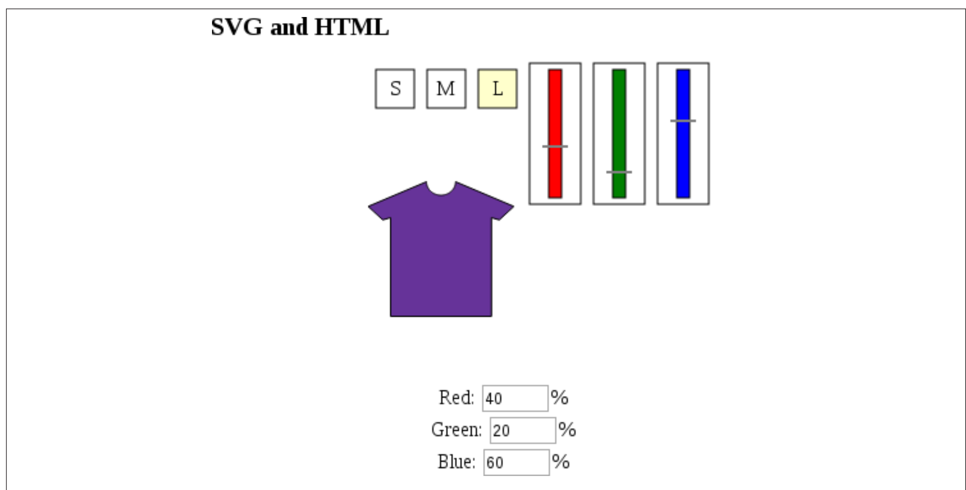


Figure 13-5. Screenshot of HTML and SVG interaction

Try it out yourself online:

http://oreillymedia.github.io/svg-essentials-examples/ch13/shirt_interact.html

Creating New Elements

In addition to modifying the attributes of existing elements, a script can also create new elements. The next addition we will make to the shirt example is the ability to add rings to the shirt in a bull's-eye pattern. Figure 13-6 shows the result, which we're sure will soon be a hugely popular fashion.

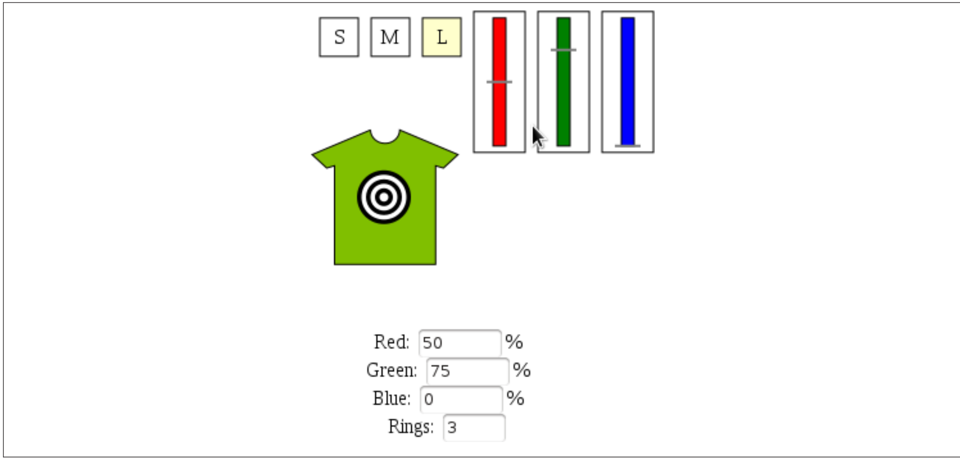


Figure 13-6. Screenshots of different selections

You can create your own designs with the online example:

http://oreillymedia.github.io/svg-essentials-examples/ch13/shirt_create.html

The HTML has to be modified to add a new form field for specifying the number of rings:

```
<label>Rings: <input id="nRings" type="text" size="3" value="0"
  onchange="createRings(this.value)" /></label>
```

The SVG lives in an external document, and the script in the HTML file needs a way to access it. In the previous section, the script in the SVG file used `parent` to access the environment where the HTML scripts were running. In this example, the HTML script will use the `<object>` element's `getSVGDocument()` method to directly access—and then modify—the SVG DOM. The SVG file is the same as in the previous section.

To establish the connection with the SVG document, an initialization function is called on the web page's `<body>` when the entire page has loaded:

```
<body onload="init()">
```

The `init()` function accesses the SVG document and stores it in a global variable:

```
var svgDoc;

function init() {
  var obj = document.getElementById("shirt");
  svgDoc = obj.getSVGDocument();
}
```

With the stage set, the function `createRings()` in the HTML page's script can add and remove elements in the SVG document, as shown in Example 13-8.

Example 13-8. JavaScript code to create elements

```
function createRings(nRings) {
  var shirt = svgDoc.getElementById("shirt"); ❶
  var rings = shirt.getElementsByTagName("circle"); ❷
  var i;
  var radius;
  var circle;

  for (i = rings.length - 1; i >= 0; i--) { ❸
    shirt.removeChild(rings[i]);
  }

  /* Pin the range to 0-5 */
  if (nRings < 0) { nRings = 0; }
  else if (nRings > 5) { nRings = 5; }

  radius = nRings * 4;
  for (i = 0; i < nRings * 2; i++) {
    circle = svgDoc.createElementNS("http://www.w3.org/2000/svg",  ❹
      "circle");
    circle.setAttribute("cx", "0");
    circle.setAttribute("cy", "0");
    circle.setAttribute("r", radius);

    if (i % 2 == 0) { ❺
        circle.style.cssText = "fill:black; stroke:none";
    }
    else {
        circle.style.cssText = "fill:white; stroke:none;";
    }
    shirt.appendChild(circle); ❻
    radius -= 2;
  }
}
```

❶ Get the `<g>` from the SVG document

❷ Retrieve all the `<circle>` elements in that group.

❸ Remove all the rings (in reverse order) by calling `removeChild(`*`nodeToRemove`*`)` on the parent `<g>` stored in the `shirt` variable.

❹ The element that is being created is part of an SVG document, so you must create an element in the namespace (`NS`) for SVG. Note that you define the namespace using the namespace URL string, not a prefix.

❺ The assigned style alternates for even and odd numbered rings. Because we're setting multiple styles on a new, unstyled element, we set them as a block using *`element`*`.style.cssText`.

❻ Finally, append the newly created `<circle>` element and reduce the radius in preparation for the next circle by calling `addChild(`*`newNode`*`)` on the `shirt` group. The new nodes will be inserted in the DOM in the order they are added, largest to smallest, so the smaller circles will be drawn on top of the larger ones.

CHAPTER 14
Using the SVG DOM

The methods and properties used in the scripting examples in Chapter 13—creating and selecting elements, getting and setting styles and attributes—are part of the Document Object Model (DOM) specification, and are not unique to SVG. The SVG 1.1 specifications define numerous additional methods to make it easier to work with the two-dimensional graphical layout of SVG. These methods allow you to figure out exactly where your text or path elements have been drawn, control the timing of your animation elements, and convert between transformed coordinate systems.

Determining the Value of Element Attributes

In "Scripting SVG" on page 213, we used `getAttribute` and `setAttribute` to access element attributes. Those methods treat the attributes as simple strings, without worrying about what they represent. If you need to calculate the difference in width between an element with `width="10em"` and an element with `width="100px"`, you would need to extract the numbers from the strings, find out the font size, and convert the measurements.

To make things easier, SVG element objects have properties representing the key attributes for that element type. An `SVGCircleElement` object, which represents a `<circle>` in your SVG code, has properties `cx`, `cy`, and `r`. An `SVGRectElement` (a `<rect>` in markup) has properties `x`, `y`, `width`, and `height`.

These properties don't store simple numbers, however. In most cases, the properties each contain two subproperties: `baseVal` and `animVal`. The `animVal` object is read-only, and is updated as an object is animated, so it always gives the current displayed state of the attribute.

Equally important, the `baseVal` and `animVal` subproperties are themselves complex data objects designed to make it easy to deal with attributes specified in different units. For

lengths and angles, `baseVal.value` and `animVal.value` always contain the value of the property in user units (degrees for angles), regardless of what unit was used to set the attribute. The `baseVal` and `animVal` objects also have methods to convert between units.

Example 14-1 shows different ways of accessing information about the *x*- and *y*-radii of an animated ellipse. The output is shown in Figure 14-1 (the exact output will depend on the default font size of your system).

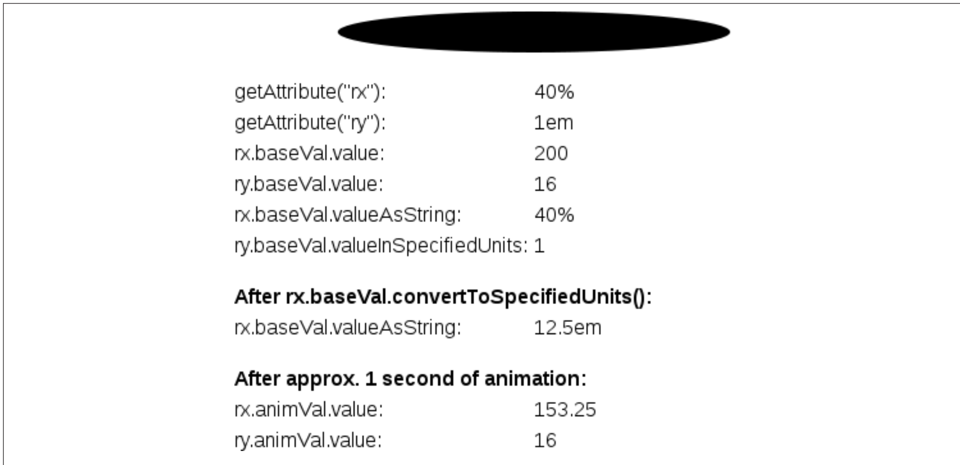


Figure 14-1. Screenshot of animation with baseVal and animVal

Example 14-1. Using the baseVal and animVal properties

http://oreillymedia.github.io/svg-essentials-examples/ch14/baseval_animval.svg

Animation markup

```
<ellipse id="el" cx="50%" cy="20" rx="40%" ry="1em">
    <animate id="animation" attributeName="rx" to="20%"
             begin="indefinite" dur="2s" fill="freeze"/>
</ellipse>
```

Markup for text output

```
<text y="3em">
    <tspan x="1em" dy="1.5em">getAttribute("rx"):</tspan>
        <tspan x="50%" id="getRx"/>
    <tspan x="1em" dy="1.5em">getAttribute("ry"):</tspan>
        <tspan x="50%" id="getRy"/>
    <tspan x="1em" dy="1.5em">rx.baseVal.value:</tspan>
        <tspan x="50%" id="rxBase"/>
    <tspan x="1em" dy="1.5em">ry.baseVal.value:</tspan>
        <tspan x="50%" id="ryBase"/>
    <tspan x="1em" dy="1.5em">rx.baseVal.valueAsString:</tspan>
        <tspan x="50%" id="rxBaseString"/>
```

```
    <tspan x="1em" dy="1.5em">ry.baseVal.valueInSpecifiedUnits:</tspan>
        <tspan x="50%" id="ryBaseUnits"/>

    <tspan style="font-weight:bold;" x="1em" dy="2.5em">After
        rx.baseVal.convertToSpecifiedUnits():</tspan>
    <tspan x="1em" dy="1.5em">rx.baseVal.valueAsString:</tspan>
        <tspan x="50%" id="rxBaseUnits"/>

    <tspan style="font-weight:bold;" x="1em" dy="2.5em">After
        approx. 1 second of animation:</tspan>
    <tspan x="1em" dy="1.5em">rx.animVal.value:</tspan>
        <tspan x="50%" id="rxAnim"/>
    <tspan x="1em" dy="1.5em">ry.animVal.value:</tspan>
        <tspan x="50%" id="ryAnim"/>
</text>
```

Script

```
var doc = document;
var el = doc.getElementById("el"); ❶
doc.getElementById("getRx").textContent = el.getAttribute("rx"); ❷
doc.getElementById("getRy").textContent = el.getAttribute("ry");
doc.getElementById("rxBase").textContent = el.rx.baseVal.value; ❸
doc.getElementById("ryBase").textContent = el.ry.baseVal.value;
doc.getElementById("rxBaseString").textContent =
 el.rx.baseVal.valueAsString; ❹
doc.getElementById("ryBaseUnits").textContent =
 el.ry.baseVal.valueInSpecifiedUnits; ❺

el.rx.baseVal.convertToSpecifiedUnits(SVGLength.SVG_LENGTHTYPE_EMS); ❻
doc.getElementById("rxBaseUnits").textContent =
  el.rx.baseVal.valueAsString;

var animate = doc.getElementById("animation"); ❼
try {
  animate.beginElement(); //start the animation
} catch(e){/* catch exception if animation not supported */}
setTimeout(getAnimatedValue, 1000); ❽

function getAnimatedValue() { ❾
  try {
    animate.endElement(); //freeze the animation
  } catch(e){}
  doc.getElementById("rxAnim").textContent = el.rx.animVal.value;
  doc.getElementById("ryAnim").textContent = el.ry.animVal.value;
}
```

❶ `el` is the "`SVGEllipseElement` object” representing the ellipse in the markup.

❷ `el.getAttribute("rx")` returns the string value of the attribute with the units used in the markup.

❸ `el.rx.baseVal.value` is the `rx` value converted to a number in user units.

❹ `el.rx.baseVal.valueAsString` is the complete string, with units.

❺ `el.ry.baseVal.valueInSpecifiedUnits` is the value of the attribute as a number, but in the units used when the attribute was set.

❻ The `convertToSpecifiedUnits(`*`unitConstant`*`)` method can be used to convert the value to any unit you choose (in this case, `ems`). The method doesn't return anything directly, but the data object's `valueAsString` and `valueInSpecifiedUnits` properties will be changed. The `value` property (which is always in user units) isn't affected.

❼ The markup defines an animation element that modifies the `rx` attribute of the ellipse, but which has `begin="indefinite"`, preventing it from starting automatically. Instead, we start it by calling the `beginElement()` method. The method is put in a try/catch block to avoid errors in browsers that don't support animation.

❽ The `setTimeout()` function call tells the computer to wait 1 second (1000ms), and then run the `getAnimatedValue` function.

❾ `getAnimatedValue()` halts the animation with `animate.endElement()`, then queries the `animVal` properties for each attribute. Although `ry` is not being animated, `ry.animVal` still exists; it is exactly equal to `ry.baseVal`.

The `rx` and `ry` properties used in Example 14-1 are both instances of `SVGAnimatedLength` objects. To describe different types of geometrical data, SVG defines a variety of custom objects. Table 14-1 lists some of the most important objects and what you can do with them.

Table 14-1. SVG Data Objects

Object Name	Description	Properties and Methods
SVGLength	A length with units. The units are defined as constants of the form SVGLength.SVG_LENGTHTYPE_*unit*, where *unit* is one of NUMBER (for user units), PERCENTAGE, EMS (em units), EXS (ex units), PX, CM, MM, IN, PT, PC.	**Properties:** unitType one of the constants defining allowed units; value the length in user units; valueInSpecifiedUnits the length in unitType units; valueAsString the value and units together as a string. **Methods:** newValueSpecifiedUnits(*unitType*, *valueInSpecifiedUnits*) sets the value and the units; convertToSpecifiedUnits(*unitType*) change the unit type, while maintaining the same value in user units.
SVGAngle	An angle with units. The unit constants are of the form SVGAngle.SVG_ANGLETYPE_*unit*, where *unit* is one of UNSPECIFIED (degrees by default) DEG, RAD, or GRAD.	Same as for SVGLength.
SVGRect	A rectanglular area in user coordinates. The SVGRect object is not the same as a <rect> element (which is represented by the SVGRectElement interface).	**Properties:** x, y, width *and* height all numbers in user coordinates.

Object Name	Description	Properties and Methods

`SVGPoint`	A point in user space.	**Properties:**
		`x`, `y`
		both numbers in user coordinates.
		
		Method:
		`matrixTransform(matrix)`
		returns the value of the point transformed by the given matrix, which must be an `SVGMatrix` object.

`SVGMatrix`	A matrix in the form used for transformations, as described in Appendix D.	**Properties:**
		`a,b,c,d,e,f`
		all numbers, representing the variable values of the matrix in top-to-bottom, left-to-right order.
		
		Methods:
		`multiply(`*`secondMatrix`*`)`
		returns a matrix equal to the two matrixes multiplied together;
		`inverse()`
		creates the invert matrix if possible, or throws an exception if the matrix is not invertable (for example if it represents a `scale(0)` operation);
		`translate(`*`x`*`, `*`y`*`)`, `scale(`*`scaleFactor`*`)`, `scaleNonUniform(`*`scaleFactorX`*`, `*`scaleFactorY`*`)`, `rotate(`*`angle`*`)`, `rotateFromVector(`*`x`*`, `*`y`*`)`, `flipX()`, `flipY()`, `skewX(`*`angle`*`)`, `skewY(`*`angle`*`)`
		returns a new matrix that is this matrix multiplied by the specified transformation; `rotateFromVector` calculates the angle required to rotate the *x*-axis to align with that vector; `flipX` and `flipY` are equivalent to scaling by an *x* or *y* factor of -1.

Object Name	Description	Properties and Methods

`SVGTransform`	A single transformation command. The type of command is defined as a constant of the form `SVG_TRANSFORM_`*`type`*, where *type* is one of `MATRIX`, `TRANSLATE`, `SCALE`, `ROTATE`, `SKEWX` or `SKEWY`.	**Properties:**
		`type`
		one of the type constants;
		`matrix`
		the `SVGMatrix` object representing this transformation;
		`angle`
		the angle of rotation or skew in degrees, if applicable, or zero.
		Methods:
		`setMatrix(`*`matrix`*`)`, `setTranslate(`*`x`*`, `*`y`*`)`, `setScale(`*`scaleFactorX`*`, `*`scaleFactorY`*`)`, `setRotate(`*`angle`*`, `*`cx`*`, `*`cy`*`)`, `setSkewX(`*`angle`*`)`, `setSkewY(`*`angle`*`)`
		change this `SVGTransform` object to the specified transformation.

`SVGTransformList`	A list of transformations (`SVGTransform` objects), in the order they would be specified in a `transform` attribute. In addition to the specific methods for transform lists, these objects also implement the `numberOfItems` property and all the methods described below for generic lists.	**Methods:**
		`createSVGTransformFromMatrix(`*`matrix`*`)`
		create a new `SVGTransform` object from an `SVGMatrix` object;
		`consolidate()`
		combine all the transformations in this list into a single matrix transformation, and then return that `SVGTransform` object.

Object Name	Description	Properties and Methods
SVG*Xxx*List	For any data type that might be used in a list or an array (and most animated data types), there is a SVG*Datatype*List. Most Javascript implementations of the SVG DOM use arrays to represent these lists, so you may also be able to use array notation to set or retrieve items.	**Property:** numberOfItems the length of the list. **Methods:** clear() removes all elements from the list; initialize(*newItem*) clears all existing items from the list, so that it only contains the one new item; getItem(*index*) accesses the list item at *index* (the first index is 0); insertItemBefore(*newItem*, *index*), replaceItem(*newItem*, *index*), re moveItem(*index*), appendItem(*newItem*) self-explanatory methods for modifying the list.
SVGAnimated*Xxx*	For nearly every data type, there is an SVGAnimated*Datatype* interface, which is used as the value of a property representing an animatable attribute. For example, the cx property of an SVGCircleElement object is an SVGAnimatedLength object, while a transform property is of type SVGAnimatedTransformList.	**Properties:** baseVal an object containing the official value of the attribute or property; animVal a read-only object containing the current displayed value of the attribute or property, after adjusting for animation.

SVG Interface Methods

Sometimes when scripting SVG, you want to calculate a geometric property that isn't defined directly by an attribute. For example, you might want to draw a rectangle around a text label that will fit neatly, regardless of what font is used for the text. Or you might want to find out where you are in the timing cycle of an animation at the time the JavaScript code is run. Other times, you might want to manipulate part of a complex attribute: one curve segment within a path, or one command within a transform attribute.

There are a variety of useful properties and methods that can be called on element objects —the objects returned by `document.getElementById(id)`—to perform these sorts of calculations and manipulations. The specifications define these in terms of interfaces, descriptions of common functionality that are implemented by specific object types.

Table 14-2 lists some of the interface features available for the elements described in this book. The list is not exhaustive, but it should get you started.

Table 14-2. Interfaces for SVG elements

Applies to	Method or property	Result
SVGElement (any element in the SVG namespace)	.ownerSVGElement()	Returns a reference to the nearest ancestor <svg> in the hierarchy; returns null if called on a top-level SVG element.
	.viewportElement()	Returns a reference to the <svg>, <pattern>, <symbol>, or <marker> element that establishes the viewport for this element.
SVGLocatable (any element that takes up coordinate space or takes a transform attribute: graphics elements, <g> or <svg>)	.nearestViewportElement	The <svg>, <pattern>, <symbol>, or <marker> element that establishes the viewport for this element.
	.farthestViewportElement	The top-level <svg> element that contains this element.
	.getBBox()	Returns the object bounding box as an SVGRect object with properties x, y, width, and height, representing the coordinates and extent of the smallest rectangle in the current user coordinate system that can contain the graphic. The bounding box is not affected by stroke width, clipping, masking, or filter effects.
	.getCTM()	Returns a cumulative transformation matrix, an SVGMatrix object representing the net transformation from the coordinate system of this element to the coordinate system of the nearestViewportElement.
	.getScreenCTM()	Returns a cumulative transformation matrix to convert from the user coordinate system of this element to the "screen" or client coordinates used to represent points on the root-level document. This method is useful in event handlers when converting between mouse/pointer coordinates and the coordinate system of your graphics.
	.getTransformToElement(*SVGElement*)	Returns an SVGMatrix representing the net transformations required to convert between the coordinate system for this element and the coordinate system for the other element.
SVGTransformable (any element that can take a transform attribute)	.transform	The SVGAnimatedTransformList representing the base and animated values of the transformations defined on this element.

Applies to	Method or property	Result
`SVGStylable` (any element that can take a `style` attribute)	`.style`	A `CSSStyleDeclaration` object representing any inline styles set on this element. See "Scripting SVG" on page 213 in Chapter 13 for methods of the `style` object.
`SVGSVGElement` (`<svg>`)	`.suspendRedraw(` *`maxWaitTimeInMilliseconds`*`)`	Tells the browser to hold off on redrawing the graphic for the specified wait time (max one minute). Useful if you are going to be making a lot of changes and you want them all to apply at once. The method returns an ID number which you can then pass to `unsuspendRedraw` function.
	`.unsuspendRedraw(`*`suspendID`*`)`	Cancels the specific `suspendRedraw` call associated with the ID value.
	`.unsuspendRedrawAll()`	Cancels all `suspendRedraw` calls on the SVG, causing the graphic to refresh again.
	`.pauseAnimations()`	Pauses the time clock for all SMIL animations within the SVG.
	`.unpauseAnimations()`	Resumes the time clock for all SMIL animations within the SVG.
	`.animationsPaused()`	Returns true or false, depending on whether animations have been paused using the above methods.
	`.getCurrentTime()`	Returns the value of the time clock used for SMIL animations. Normally, this is the number of seconds since the document was loaded, but it can be affected by pausing or unpausing animations or setting the time directly.
	`.setCurrentTime(`*`timeInSeconds`*`)`	Sets the SMIL time clock to the specified value, affecting all animations
	`.getIntersectionList(` *`rectangle`*`,` *`referenceElement`*`)`	Returns a list of elements that (a) overlap the specified rectangle (an `SVGRect` object) based on the current `<svg>` element's coordinate system, and that (b) are children of the reference element. Only parts of the element that are sensitive to pointer events (based on the `pointer-events` property, by default any visible painted areas) are considered when determining overlap. The reference element can be null, in which case all children of this `<svg>` are included.[a]
	`.getEnclosureList(` *`rectangle`*`,` *`referenceElement`*`)`	Similar to `getIntersectionList()`, except that it only returns elements that are entirely within the rectangle

Applies to	Method or property	Result
	.checkIntersection(*element*, *rectangle*)	Returns true or false, depending on whether the given element intersects the given SVGRect object, in this <svg> element's coordinate system.
	.checkEnclosure(*element*, *rectangle*)	Returns true or false, depending on whether the given element is completely contained in the given SVGRect object, in this <svg> element's coordinate system.
	.createSVG*Xxx*()	The SVGSVGElement supports methods to create new instances of each of the data object types (SVGPoint, SVGAngle, SVGMatrix, etc.) from Table 14-1. The methods don't take any parameters; the resulting objects are initialized with their properties set to 0 (except for createSVGMatrix(), which returns an identity matrix).
SVGUseElement (<use>)	.instanceRoot	Contains the top-level node of a shadow DOM tree representing the graphics drawn by the <use> element. The elements in the shadow DOM (SVGElementInstance objects) have limited function: you cannot manipulate attributes or styles directly, but they can be targets of user events. Each SVGElementInstance has a correspondingElement attribute that links to the source graphics element that it duplicates, and a correspondingUseElement property that links back to the <use> element.[b]
SVGPathElement (<path>)	.getTotalLength()	Returns the computed length, in user units, of the entire path (not counting any move-to commands). This value may not be exactly the same in every agent because approximations may be used for some curves.
	.getPointAtLength(distance)	Returns an SVGPoint object with properties x and y, representing the coordinates of the point on the path that is distance user units from the start, using the same method of calculating path length as for getTotalLength().

Applies to	Method or property	Result
SVGPathData (<path> elements and any other elements that support a path data attribute, such as <animateMotion> elements)	.pathSegList	Contains a list of object representing each segment of the path. The list can then be modified or queried in an object-oriented manner; see the SVG specifications for methods. The pathSegList property returns the path corresponding to the actual d attribute value; use the animatedPathSegList property to access the current state of the path in case the d attribute is being animated.
	.normalizedPathSegList	A simplified version of the path segment list, where each segment has been converted to either a move, line-to, cubic curve, or close path command in absolute coordinates. There is also an animatedNormalizedPathSegList property.
SVGAnimatedPoints (<polygon> and <polyline> elements)	.points	Returns the points associated with this element as an SVGPointList. The property animatedPoints is similar but is updated when the points attribute is animated.
SVGTextContentElement (any text element, including <text>, <tspan>, and <textPath>)	.getNumberOfChars()	Returns the total number of characters in this element, including text in all child <tspan> elements. Multibyte unicode characters are counted according to the number of UTF-16 characters required to represent them.
	.getComputedTextLength()	Returns the length, in user coordinate units, required to write out the text after applying all CSS properties and dx and dy attributes. Does not include any adjustments made based on a textLength attribute.
	.getSubStringLength(*charNum*, *nChars*)	Returns the computed text length for a substring of the text, starting with the character that is charNum number of characters into the text (the first character is numbered 0), and continuing for nChars or until the end of the text, whichever comes first.
	.getStartPositionOfChar(*charNum*)	Returns an SVGPoint object with properties x and y, representing the position of the specified character within user coordinate space. How the character is drawn relative to this point depends on the writing mode (vertical or horizontal, left-to-right or right-to-left) and the baseline-alignment property. For default properties on left-to-right text, the point will be where the left edge of the character intercepts the baseline.

Applies to	Method or property	Result
	`.getEndPositionOfChar(`*charNum*`)`	Similar to `getStartPositionOfChar` but returns the point where the baseline meets the end edge of the character.
	`.getExtentOfChar(`*charNum*`)`	Returns an `SVGRect` object, similar to `getBBox()`, except that it only represents the boundaries for a single character.
	`.getRotationOfChar(`*charNum*`)`	Returns the rotation value (angle in degrees) for the specified character, after taking into consideration any `rotate` attribute and any rotation from text on a path, but not including any transformation of the coordinate system.
	`.getCharNumAtPosition(`*point*`)`	Returns the index number for the character that includes the specified point, or returns -1 if no characters in the string contain that point. The point is specified as an `SVGPoint` object, either one returned from another interface method, or one created by calling `createSVGPoint()` on an `<svg>` element and then setting the returned object's `x` and `y` properties.
`ElementTimeControl` and `SVGAnimationElement` (any of the SVG animation elements: `<animate>`, `<set>`, `<animateTransform>`, and `<animateMotion>`)[c]	`.targetElement`	The `SVGElement` that this animation element modifies.
	`.beginElement()`	Start the specified animation immediately, if it isn't prevented by a `restart` attribute of `never` or `whenNotActive`.
	`.beginElementAt(`*offset*`)`	Start the specified animation after *offset* number of seconds; if offset is negative, the animation will start immediately but be calculated as if it had started that many seconds previously.
	`.endElement()`	End the current run of the animation (including all repeats) immediately.
	`.endElementAt(`*offset*`)`	End the animation after *offset* number of seconds.
	`.getStartTime()`	If the animation is running, returns the start time at which it began, in seconds relative to the time clock for the SVG. If the animation is waiting to begin, returns the time at which it will begin. Otherwise, throws an exception.
	`.getCurrentTime()`	Returns the time in seconds of the animation time clock for the SVG that this element is part of, the same as calling `getCurrentTime()` on that SVG.

Applies to	Method or property	Result
	`.getSimpleDuration()`	Returns the number of seconds for the duration of each cycle of the animation (the value of the `dur` attribute); throws an exception if the duration is undefined.

[a] At the time of writing, Firefox (version 30) does not support the methods `getIntersectionList`, `getEnclosureList`, `checkIntersection`, or `checkEnclosure`.

[b] Firefox (as of version 30) also does not make the shadow DOM tree of a `<use>` element accessible to scripts, and does not implement the `SVGElementInstance` interface.

[c] None of the animation-related properties and methods are implemented in Internet Explorer (version 11), which does not support SMIL animation. The Apache Batik SVG viewer, version 1.7, throws errors when using `beginElement` and `beginElementAt`.

Constructing SVG with ECMAScript/JavaScript

The next example is a simple analog clock, as shown in Figure 14-2; Example 14-2 gives the SVG code.

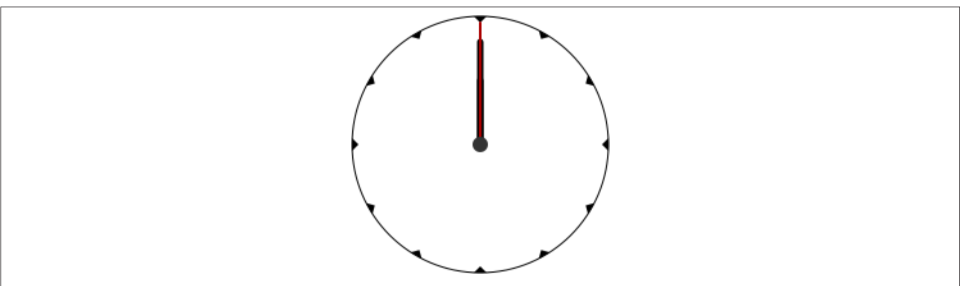


Figure 14-2. Analog clock

Example 14-2. SVG for a basic analog clock

```
<svg xmlns="http://www.w3.org/2000/svg"
    id="clock" width="250" height="250" viewBox="0 0 250 250">
<title>SVG Analog Clock</title>

<circle id="face" cx="125" cy="125" r="100"
        style="fill: white; stroke: black"/>
<g id="ticks" transform="translate(125,125)">
    <path d="M95,0 L100,-5 L100,5 Z" transform="rotate(30)"  />
    <path d="M95,0 L100,-5 L100,5 Z" transform="rotate(60)"  />
    <path d="M95,0 L100,-5 L100,5 Z" transform="rotate(90)"  />
    <path d="M95,0 L100,-5 L100,5 Z" transform="rotate(120)" />
    <path d="M95,0 L100,-5 L100,5 Z" transform="rotate(150)" />
    <path d="M95,0 L100,-5 L100,5 Z" transform="rotate(180)" />
    <path d="M95,0 L100,-5 L100,5 Z" transform="rotate(210)" />
    <path d="M95,0 L100,-5 L100,5 Z" transform="rotate(240)" />
    <path d="M95,0 L100,-5 L100,5 Z" transform="rotate(270)" />
    <path d="M95,0 L100,-5 L100,5 Z" transform="rotate(300)" />
    <path d="M95,0 L100,-5 L100,5 Z" transform="rotate(330)" />
    <path d="M95,0 L100,-5 L100,5 Z" transform="rotate(360)" />
</g>

<g id="hands" style="stroke: black;
                     stroke-width: 5px;
                     stroke-linecap: round;">
  <path id="hour" d="M125,125 L125,75"
        transform="rotate(0, 125, 125)"/>
  <path id="minute" d="M125,125 L125,45"
        transform="rotate(0, 125, 125)"/>
  <path id="second" d="M125,125 L125,30"
```

```
        transform="rotate(0, 125, 125)"
        style="stroke: red; stroke-width: 2px" />
</g>
<circle id="knob" r="6" cx="125" cy="125" style="fill: #333;"/>
</svg>
```

The code isn't particularly complicated, but it is rather redundant. Each of the 12 hour-marker elements has nearly the exact same syntax, just a different `rotation` attribute. You could use `<use>` elements to avoid repeating the path data, but it really wouldn't simplify things very much. If you wanted to add numbers for the hours, you'd have 12 `<text>` elements, too. And if you wanted to add minute markers, you'd need 60 separate elements for the marks.

In programming, whenever you're doing the same thing many times, you really ought to be using a loop or function to do it. In "Creating New Elements" on page 229 in Chapter 13, JavaScript was used to create an arbitrary number of SVG elements based on user input. The same methods can also be used to create the initial graphic, and they can be especially helpful if your graphic is repetitive and geometrical.

Example 14-3 creates the same output as the previous one in an SVG viewer that supports JavaScript. It uses both the basic DOM methods and the SVG DOM features introduced in this chapter. The `<svg>` contains only two markup elements: a `<title>` and a `<script>`.

Example 14-3. Basic analog clock created with ECMAScript

```
<!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"
    id="clock" width="250" height="250" viewBox="0 0 250 250"
     onload="init()" >
<title>Scripted Analog Clock</title>

<script type="application/ecmascript"> <![CDATA[

  /* the <svg> object that will contain the drawing */
  var clock; ❶

  function init() { ❷
    /* select the empty <svg> */
    clock = document.getElementById("clock");
    var svgns = clock.namespaceURI,
        doc   = document;

    clock.suspendRedraw(1000); ❸

    /* create the clock face */ ❹
    var face = doc.createElementNS(svgns, "circle");
    face.cx.baseVal.value = 125;
    face.cy.baseVal.value = 125;
```

```
    face.r.baseVal.value = 100;
    face.style.cssText = "fill: white; stroke: black";
    clock.appendChild( face );

    /* create a group for the ticks */
    var ticks = clock.appendChild(
          doc.createElementNS(svgns, "g") );
    ticks.setAttribute("transform", "translate(125,125)" );

    /* create the tick marks */
    var tickMark;
    for (var i = 1; i <= 12; i++) { ❺
      tickMark = doc.createElementNS(svgns, "path");
      tickMark.setAttribute( "d",
                             "M95,0 L100,-5 L100,5 Z" );
      tickMark.setAttribute( "transform",
                             "rotate(" + (30*i) + ")" );
      ticks.appendChild( tickMark );
    }

    /* create the hands */
    var hands = clock.appendChild(
                    doc.createElementNS(svgns, "g") );
    hands.style.cssText =
      "stroke: black; stroke-width:5px; stroke-linecap: round;";

    var hourHand = hands.appendChild(
                    doc.createElementNS(svgns, "path") );
    hourHand.id = "hour";
    hourHand.setAttribute("d", "M125,125 L125,75"); ❻
    hourHand.setAttribute("transform", "rotate(0, 125, 125)");

    /* similar code for minute and second hands
      omitted to save space */

    /* add the center knob */ ❼
    var knob = doc.createElementNS(svgns, "circle");
    knob.setAttribute("cx", "125");
    knob.setAttribute("cy", "125");
    knob.setAttribute("r", "6");
    knob.style.setProperty("fill", "#333", null);
    clock.appendChild( knob );

    clock.unsuspendRedrawAll(); ❽
  }

  // ]]>
</script>
</svg>
```

1. The script declares a global variable to hold the <svg> element; the bulk of the code, however, is in an initialization function (`init()`), which will run when the SVG loads.
2. The `init()` function starts by selecting the <svg> element. It also declares convenience variables; accessing the `.namespaceURI` property on any existing SVG element is a good way to avoid retyping the URL each time you create a new element.
3. Although not required, suspending drawing updates on your SVG before making a lot of DOM changes can improve performance in some viewers.
4. The clock face <circle> is created, styles and attributes are set using SVG DOM properties, and then it is added to the SVG.
5. Here's the magic of looping. The `for` loop runs the same code 12 times to create each hour marker, and calculates the rotation angle for each. Note that for initializing complex attributes like `d` or `transform`, passing a string to `setAttribute()` is generally easier than manipulating the complex DOM objects that represent the values.
6. The clock hands are initialized pointing to midnight, with zero-rotation transformation attributes to make it easier to later change the rotation.
7. For comparison with the face circle, the knob <circle> is initialized using setter methods instead of DOM properties. It still requires six lines of code.
8. Don't forget to cancel any `suspendRedraw()` calls when you're finished building the DOM!

Using scripting shortened the code required to make the tick-marks, but it considerably increased the amount of code required to draw simple elements like the circles for the clock face and knob. This is the main reason why JavaScript libraries like Snap.svg are so popular; they have shortcut functions for the most common operations, like creating elements and setting attributes.

We'll get to that in a moment, but until we get to a discussion of libraries (the easy way to do it), we'll continue to draw the clock face using plain SVG. After all, there's a more pressing issue: our clock doesn't tell time.

Animation via Scripting

One option to get the clock ticking is to use <animateTransform> elements on the hands. You can set the second hand to rotate 360 degrees every minute, the minute hand

to rotate 360 degrees every hour, and the hour hand to rotate 360 degrees every 12 hours. What `<animateTransform>` cannot do is get the hands to show the correct time.[1]

There are other things animation elements can't do easily. They can't keep track of past user events and change the way they respond to new events accordingly. If your animation relies on logic, data, or complex user interactions, it makes more sense to control it from a script (with the other application functionality) than to try to define it in the XML (the document structure).

To create an animation with JavaScript, you repeatedly modify the attribute or style property to change it from the starting value to the ending value. If you're just starting out in programming, you might think that you could create a loop to continually update the attributes by wrapping the update code in a `while(true)` block.

This would ensure your clock is always up-to-date, but we wouldn't recommend it. The `while(true)` technique is akin to constantly looking at your watch and asking, "Is it time yet?" It gives you no time to sleep or do anything else. In an SVG script, it gives the computer no chance to attend to other tasks.

It takes your computer only a tiny fraction of a millisecond to run through a simple loop like the one in the preceding script; each increment of the clock hands will be essentially meaningless. In comparison, most film and video consists of images that are updated 30–60 times per second. That speed is called the *frame rate* of the video, and it is sufficient to convince your eyes that you're watching smooth motion.

Computer displays have a frame rate, too. As content changes, part or all of the display will be updated to match. However, the effective frame rate of the computer display depends on how much other work the computer is doing in the background. If you bog down the computer with endless loops, it won't have time to repaint the screen, and your animation will become slow and choppy, regardless of how often you update the attributes.

To persuade your computer to create smooth animations, you have to be polite. By calling the method `requestAnimationFrame(`*`animationfunction`*`)`, you are effectively saying, "Computer, next time you're ready to repaint the screen, please run this function first." The function will be passed a timestamp value, which will be the same for all functions being called for a given frame, allowing you to coordinate multiple animation functions. (The timestamp is based on the document time clock, not the system clock, so we won't be using it to set the time.)

If the last line of your animation function also calls `requestAnimationFrame` and passes itself as the function to run, then it will be called as often as the computer can draw the material to the screen, allowing you to create a smooth animation without exhausting

1. The SMIL specifications do define a format for synchronizing the start time of animations with the system clock, but it isn't implemented in most browsers or SVG viewers.

your computer's resources.[2] Importantly, if the window that contains this script is currently minimized or hidden, then the function won't be called at all until there is something visible to animate.

Mimicking requestAnimationFrame with setTimeout

The `requestAnimationFrame()` method is a relatively new addition to the DOM specifications, and is not supported in older browsers or Batik. To allow your script to run smoothly, you can test whether the function exists, and create a substitute method if it doesn't, as shown here. The substitute method makes use of the `setTimeout(`*`function`*`,`*`waitTime`*`)` method, which tells the computer to run your function after the specified `waitTime` (in milliseconds). This code should go at the very beginning of your `<script>`:

```
if (!window.requestAnimationFrame) { ❶

  window.requestAnimationFrame = function(animationFunction) { ❷

    function wrapperFunction() { ❸
      animationFunction(Date.now());
    }

    setTimeout(wrapperFunction, 30); ❹
  }
}
```

❶ If a `requestAnimationFrame` method doesn't exist…

❷ Create your own function and save it as the `requestAnimationFrame` property of the global object. Your new function must accept an animation callback function as a parameter.

❸ Take the passed-in animation function, and wrap it in a function that can be run without parameters. The wrapper function calls the animation function with a timestamp value as a parameter. The timestamp returned by `Date.now()` is just the system timestamp as an integer.

❹ The `setTimeout` method calls the animation function when the computer is free, but no sooner than 30 milliseconds, which works out as approximately 33 frames per second.

The `setTimeout()` function is not as polite as `requestAnimationFrame()`, because it doesn't adjust for anything else your computer might be doing, and it will run regardless of whether the animation will be visible. This sample code also does not replace all the

2. You may have heard of a programming technique called *recursion*, which happens when a function calls itself. This is not recursion, because your function calls `requestAnimationFrame()`, not itself.

functionality of `requestAnimationFrame()`; specifically, it does not synchronize multiple animation calls, synchronize with the SMIL animation clock, or provide a way to cancel an animation frame request. Nonetheless, with a sensible wait time, it should provide acceptably smooth animation for simple programs. Just insert it at the top of your script (or in its own script tag at the top of your file).

Example 14-4 presents the code for updating the clock's hands, the polite way, using `requestAnimationFrame()`.

Example 14-4. Animating your scripted SVG clock

Global variables

```
/* references to the SVGPathElements for the clock hands */
var hourHand,
    minuteHand,
    secondHand;
/* references to SVGTransform object that rotates each hand*/
var secondTransform,
    minuteTransform,
    hourTransform;
/* time conversion constants */
var secPerMinute  = 60,
    secPerHour    = 60*60,
    secPer12Hours = 60*60*12;
```

Variable initialization (in the `init()` function)

```
function init() {
  /*
    Access the SVGPathElements for the clock hands
  */
  hourHand = document.getElementById("hour");
  minuteHand = document.getElementById("minute");
  secondHand = document.getElementById("second");

  /* Access the SVGTransform objects that represent
     the current rotate(0, 125, 125) transform on each hand:
  */
  secondTransform = secondHand.transform.baseVal.getItem(0);
  minuteTransform = minuteHand.transform.baseVal.getItem(0);
  hourTransform   = hourHand.transform.baseVal.getItem(0);
  updateClock(); /* start the clock going */
}
```

updateClock function

```
function updateClock() {
    /* get the system time */ ❶
    var date = new Date();
    /* calculate the number of seconds since midnight */
    var time = date.getMilliseconds()/1000 +
```

```
            date.getSeconds() +
            date.getMinutes()*60 +
            date.getHours()*60*60; ❷

    /* calculate the rotation angles */ ❸
    var s = 360*( time % secPerMinute )/secPerMinute,
        m = 360*( time % secPerHour )/secPerHour,
        h = 360*( time % secPer12Hours )/secPer12Hours;

    /* use SVGTransform.setRotate(angle, cx, cy)
       to update the rotation angle:
    */
    secondTransform.setRotate( s, 125, 125); ❹
    minuteTransform.setRotate( m, 125, 125);
    hourTransform.setRotate( h, 125, 125);

    window.requestAnimationFrame( updateClock ); ❺
    // repeat for the next frame
}
```

❶ The constructor `new Date()` returns the local system date and time (with precision of thousandths of a second) as a JavaScript object.

❷ The `date.getPart()` functions return part of the date as an integer. Using these methods, the time of day is calculated as the number of seconds since midnight.

❸ Using the predefined constants, the rotational position of each clock hand (in number of degrees past midnight) is calculated. The `%` is the *modulus* operator, which returns the remainder part of integer division.

❹ In the initialization function, we access the first (and only) item in the `SVGTransformList` in the `baseVal` of each hand's `transform` attribute, and store that `SVGTransform` object in a variable. At each update, we directly change the rotation of each transform. By modifying the objects directly, instead of using `setAttribute`, we skip the time required for the browser to parse the attribute string.

❺ The final line of `updateClock` uses `requestAnimationFrame` to schedule itself to be run again the next time the screen refreshes.

The animation cycle is started by the `updateClock()` call at the end of the initialization function. The initialization function is called by adding an `onload="init()"` attribute to the opening `<svg>` tag. The final, working clock can be seen on the book's website:

http://oreillymedia.github.io/svg-essentials-examples/ch14/animated_clock_js.svg

Using JavaScript Libraries

In Example 14-3, we created the entire clock by constructing each element "by hand" with JavaScript calls. As we noted at the time, building elements this way can be painfully slow. There must be a better way, and, indeed, there is. You can use free, open source JavaScript libraries such as D3.js, Raphaël, or Snap.svg to simplify the task. These libraries are all just external script files as far as the web browser is concerned. For you, however, they are collections of useful functions that you can "borrow" from as needed.

Which library to use depends on your needs. D3.js is "a JavaScript library for manipulating documents based on data." It is best for manipulating sets of similar elements all at once, defining their attributes, styles, and reaction to events according to a corresponding value in a data array. If you wanted, for example, a highly interactive bar chart, D3.js would be an excellent choice.

Raphaël and Snap.svg are more-generic libraries with a goal of making it easy to modify or create animated graphics. Raphaël is compatible with older browsers, converting your graphics commands to forms they understand. Snap, created by the Adobe Web Platform team, is designed for modern browsers and uses SVG exclusively. This example will use Snap.

All three libraries work by *wrapping* the DOM element objects in their own custom objects, which have additional properties and methods for you to use. In Snap, the `<svg>` element that holds your graphic is wrapped up in a `Paper` object. Snap provides function calls that let you add graphics elements to the paper, modify their attributes, and handle events that occur in them.

There are numerous JavaScript libraries available designed to work with HTML. Be cautious about using them for SVG. It's not only that they do not have any methods for specifically dealing with graphics; even basic tasks can become problematic if the library is not aware of XML namespaces. For example, the popular JQuery library does not (at the time of writing) have any way of creating new elements in the SVG namespace; if you ask JQuery to make you a circle, it will return an `HTMLUnknownElement` object. In contrast, SVG-aware libraries like D3 and Snap recognize the SVG element names and will know that *circle* means `SVGCircleElement`.

Here is the XML for a Snap-based version of the animated clock:

```
<!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"
     id="clock" width="250" height="250" viewBox="0 0 250 250"
     onload="init()" >
```

```
    <title>Snap.svg Analog Clock</title>

    <script type="application/ecmascript"
            xlink:href="snap.svg-min.js"></script>
    <script type="application/ecmascript">

    /* Initialization and update functions go here */

    </script>
  </svg>
```

The first `<script>` element brings in the Snap library. We're hosting the source code directly on our server in a minified form, meaning that the code has been processed to remove whitespace and comments and shorten variable names. The minified version of Snap.svg v0.3.0 is a 72 KB file, without file compression. In other words, downloading the Snap library uses up about the same bandwidth for your web page's visitors as downloading a moderately complex PNG diagram.

To use an external library in an HTML file, use the attribute `src` for the file URL, and be sure to include both an opening and closing tag; empty script elements will not work in older browsers.

The second script will follow the same structure as the previous examples, with an `init()` function to draw the clock and an `updateClock()` to get it ticking.

If you're trying out these examples yourself, be sure you're using the latest version of *snap.svg-min.js*. The library was initially designed to modify inline SVG code within an HTML page, and versions prior to 0.3.0 had bugs when run in standalone SVG files.

Example 14-5 presents the Snap script to draw the clock. It follows the same structure (and creates the same graphic) as Example 14-3, but demonstrates many of Snap's shortcut methods.

Example 14-5. Drawing a basic analog clock with Snap.svg

```
/* the Paper object where the clock is drawn */
var clock;
/* references to the Snap Elements for the clock hands */
var hourHand,
    minuteHand,
    secondHand; ❶

/* time conversion constants */
var secPerMinute  = 60,
    secPerHour    = 60*60,
    secPer12Hours = 60*60*12;

function init() { ❷
```

```
  /* select the empty <svg> as a snap Paper object */
  clock = Snap("#clock");

  /* create the clock face */ ❸
  var face = clock.circle(125, 125, 100);
  face.attr({fill: "white", stroke: "black"});

  /* create a group for the ticks */
  var ticks = clock.g();
  ticks.transform("t125,125"); ❹

  var tickMark;
  for (var i = 1; i <= 12; i++) { ❺
    tickMark = clock.path("M95,0 L100,-5 L100,5 Z");
    tickMark.transform("rotate("+ (30*i) + ")");
    ticks.add(tickMark);
  }

  /* create the hands */
  hourHand = clock.path("M125,125 L125,75");
  minuteHand = clock.path("M125,125 L125,45");
  secondHand = clock.path("M125,125 L125,30");

  var hands = clock.g(hourHand, minuteHand, secondHand);  ❻
  hands.attr({stroke: "black",
              "stroke-width": 5, ❼
              "stroke-linecap": "round"});
  secondHand.attr({stroke: "red", strokeWidth: "2px"});

  /* add the center knob */
  clock.circle(125, 125, 6).attr({fill: "#333"}); ❽

  updateClock(); ❾
}

function updateClock()
{
  /* adjust the hands */
}
```

❶ The global variable names are the same, but their content will differ; `clock` will be a Snap `Paper` object, and the variables for the hands will point to the Snap-wrapped elements.

❷ In the `init()` function, the `Snap(`*`selector`*`)` method creates the `Paper` object containing the `<svg>` identified by the query string (in CSS selector format). An alternative version of the function, `Snap(`*`width`*`, `*`height`*`)` will create a new SVG of the specified dimensions.

❸ *Paper*.circle(*cx*,*cy*,*r*) creates a <circle> with the given center coordinates and radius, and adds it to the SVG. The Snap-wrapped circle element is assigned to a variable so that we can change its attributes on the next line. The *Element*.attr(*attrValues*) function sets multiple attributes at once, given in JavaScript object notation. (We're using presentation attributes because, as of v0.3.0, Snap doesn't have a shorthand way to set inline styles.)

❹ An empty <g> element, ticks, is created with *Paper*.g(). The transform attribute could be set with ticks.attr(), but there's a Snap shorthand method for this common task. There's also a shorthand notation for the transformation commands: "t125,125" is short for "translate(125,125)".

❺ The for loop creates the hour marks with *Paper*.path(*pathData*); they are then moved into the <g> with the command ticks.add(tickMark). Just to show it can be done, the transform attribute is set with standard SVG notation.

❻ The clock hands are created, and then grouped together all in one line: the *Paper*.g(*Element*, *Element*, …) both creates the group and moves the specified elements into it.

❼ When using JavaScript object notation to set attributes that have a hyphen (-) in the name, you either quote the name ("stroke-width") or convert it to camel case form (strokeWidth).

❽ Because the Snap methods that create elements also return those elements, you can *chain* multiple method calls together with dot notation.

❾ As before, the last line of the initialization function is a call to updateClock().

If you load this file into your browser, you will see that it looks exactly the same as Figure 14-2. The preceding SVG does what Example 14-3 did, but in a much more readable fashion.

The only task remaining is to get our Snap-ified SVG clock working again. Example 14-6 gives the code. Instead of animating the motion ourselves with requestAnimationFrame, we let Snap's *Element*.animate() function handle the movement. It uses requestAnimationFrame behind the scenes and, for old browsers, defines its own substitute method that coordinates multiple animation calls.

Element.animate() has two required and two optional parameters:

- An attributes object (in the same form as *Element*.attr()), giving the *final* value for each attribute in the animation.
- A duration in milliseconds indicating how long the animation should take to reach the final value.

- Optionally, a function defining the rate of change of the attribute over the course of the duration (an *easing* function). The Snap source code defines a number of functions you can use as properties of Snap's `mina` object: `mina.easeinout` will give you smooth acceleration and deceleration, while `mina.bounce` will hit the final value quickly, and then bounce back a few times before settling down. For steady motion throughout the duration, use `mina.linear`.
- Optionally, a callback function that will run when the animation is complete. This can be used to cause the animation to repeat indefinitely.

Example 14-6. Animating a clock with Snap.svg

```
function updateClock()
{
    /* get the system time */ ❶
    var date = new Date();
    /* calculate the number of seconds since midnight */
    var time = date.getMilliseconds()/1000 +
               date.getSeconds() +
               date.getMinutes()*60 +
               date.getHours()*60*60;

    /* calculate the rotation angles */
    var s = 360*( time % secPerMinute)/secPerMinute,
        m = 360*( time % secPerHour )/secPerHour,
        h = 360*( time % secPer12Hours )/secPer12Hours;

    secondHand.transform("r" + s +",125,125"); ❷
    minuteHand.transform("r" + m +",125,125");
    hourHand.transform("r" + h +",125,125");

   secondHand.animate({transform: "r" + [s + 360, 125, 125]}, ❸
                        60000, mina.linear);
   minuteHand.animate({transform: "r" + [m + 6, 125, 125]},
                        60000, mina.linear);
   hourHand.animate({transform: "r" + [h + 0.5, 125, 125]},
                        60000, mina.linear, updateClock); ❹
}
```

❶ The calculations are the same as in Example 14-4.

❷ Snap shorthand functions and transformation notation are used to set the time.

❸ The `animate` calls get the clock ticking. The first parameter to each `animate` function gives the position of the hand in 1 minute's time; the second hand turns 360 degrees, the minute hand moves 6 degrees, while the hour hand moves half a degree. The second parameter indicates that the animation should take 60,000 milliseconds (1 minute) to run, and the third parameter sets the animation to move at a steady (linear) rate.

❹ When the minute is up, the callback parameter passed to the final animate function causes `updateClock()` to be run again. This resynchronizes the animated clock with the system clock, and triggers another minute's worth of animated motion. Note that only one of the animate functions needs the callback, as running `updateClock` will restart all three.

You can see the clock in action online:

http://oreillymedia.github.io/svg-essentials-examples/ch14/snap_animated_clock.svg

Event Handling in Snap

Using a library like Snap also makes event handling much easier. The following example, which uses Snap in an HTML page, is very simple: it displays a circle and a button. You can drag the circle, and clicking the button returns the circle to the center.

The main Snap functions you need for this script are `Snap()`, `click()`, and `drag()`. The `Snap()` function takes a string in the form `#idName` and returns the corresponding element as an object that wraps a DOM element with extra functionality . Once you have an element, you can use its `click()` and `drag()` functions to have that element respond to the appropriate events.

Example 14-7 shows the necessary HTML.

Example 14-7. HTML for Snap events example

```
<html xml:lang="en" lang="en"
    xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Click and Drag Events in Snap</title>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <script type="text/javascript" src="snap.svg-min.js"></script>
  <script type="text/javascript">
    function init() {
    }
  </script>
</head>

<body onload="init()">
  <h1>Click and Drag Events in Snap</h1>

<div style="text-align:center">
  <svg width="200" height="200" viewBox="0 0 200 200"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">

    <circle id="circle" cx="100" cy="100" r="30"
      style="fill:#663399; stroke: black"/>
```

```
    <rect id="button" x="60" y="170"
        rx="5" ry="5" width="80" height="25"
        style="stroke:black; fill:#ddd; cursor:pointer"/>
    <text id="buttonText" x="100" y="187" class="buttonText"
      style="fill:black; stroke:none;
      font-family: sans-serif; font-size: 12pt;
      text-anchor:middle; cursor:pointer">Reset</text>
  </svg>
  </div>
</body>
</html>
```

Clicking Objects

To set a click handler, call a Snap element's `click()` function and pass it the name of the function that handles the click. Here is the code you need to insert in order to handle a click on the button and its text:

```
function init() {
  Snap("#button").click(resetFcn);
  Snap("#buttonText").click(resetFcn);
}

function resetFcn(evt) {
  Snap("#circle").attr({cx: 100, cy: 100});
}
```

The handler function gets the triggering event as its parameter, but in this case, the `resetFcn()` function doesn't need to use it. If you try the code at this point, nothing will appear to happen, as the circle is already at the center of the drawing. Change the value of either `cx` or `cy` to see that the handler is really working.

Dragging Objects

Now that the button is handled, you can add drag handling to the circle. The `drag()` method has three arguments: the name of a function to handle moving, the name of a function to handle the drag start, and the name of a function to handle the drag end.

The drag start function takes three parameters: the starting *x* postion, the starting *y* position, and the DOM event object that triggered the start.

The drag end function takes only one parameter: the DOM event object at the end of the drag.

The drag move function has five parameters:

- `dx`, the shift in *x* from the start point
- `dy`, the shift in *y* from the start point

- `x`, the *x* position of the mouse
- `y`, the *y* position of the mouse
- `event`, the DOM event for the mouse movement

You will need to remember where the circle's starting point is:

```
var startX = 100;
var startY = 100;
```

Here is the code you need to add to `init()` to assign the drag handlers to the circle:

```
Snap("#circle").drag(dragMove, dragStart, dragEnd);
```

And here are those functions (in logical order of start, move, end):

```
function dragStart(x, y, evt) {
  // figure out where the circle currently is
  startX = parseInt(Snap("#circle").attr("cx"), 10);
  startY = parseInt(Snap("#circle").attr("cy"), 10);
}

function dragMove(dx, dy, x, y, evt) {
  Snap("#circle").attr({cx: (startX + dx), cy: (startY + dy)});
}

function dragEnd(evt) {
  // no action required
}
```

Figure 14-3 shows the result, edited to save vertical space.

Figure 14-3. Screenshot of circle being dragged

To click and drag it yourself, test out the online interactive version:

http://oreillymedia.github.io/svg-essentials-examples/ch14/snap_events.html

These examples are only the very minimum of what you can do with the Snap library. For much more sophisticated examples, go to the website and see the demos. D3, Snap, and Raphaël are not the only SVG libraries out there, but they all have one thing in common: they make it easy for you to use JavaScript to create and manipulate SVG dynamically.

CHAPTER 15

Generating SVG

The previous chapters have described the major features of SVG. All the examples have been relatively modest and have been written in an ordinary text editor. For graphics of any great complexity, though, few people will write SVG from scratch. Let's face it: almost nobody does this by hand. Instead, graphic designers will use some sort of graphic tool that outputs SVG, and programmers will take existing raw data and convert it to SVG with a script.

If you're dealing with a graphic program's output that is already in SVG format, you can sit back and relax; all the heavy lifting has been done for you. If you ever take a look at the SVG that it generated, it may be hard to read. Some programs, for example, may not use groups (the `<g>` element) efficiently or they may not optimize paths. When you use these programs, you are trading off the ease of generating SVG for the absolute control you have when you write the entire file by hand. Hopefully, with a better understanding of what's going on "under the hood," you will be better able to write code to adapt and interact with the graphics program's output.

Generating SVG code from a data file is a trickier topic. The possibilities will depend on what type of data you have to start with and what type of programming languages you can work with.

One option for generating SVG is to build up an SVG document object model using the methods presented in Chapter 14. D3.js, which we alluded to briefly in "Using JavaScript Libraries" on page 256, is specifically designed for using data files to dynamically build SVG charts and graphs in the web browser. Scott Murray's *Interactive Data Visualization for the Web* (O'Reilly) is a good introduction for beginners. There are also many good tutorials by the library's original author, Mike Bostock, and others listed on the project's wiki page.

A different approach to dynamically generating an SVG file is to piece together the markup using the string manipulation and file-writing methods of your favorite

programming language. The first section of this chapter outlines how you could use a custom program to convert geographical mapping data that's not in an XML format to an SVG file.

If your data is already in XML format, you may just need to extract the pertinent data and plug it into an SVG framework. In such a case, you can use tools that implement Extensible Stylesheet Language Transformations (XSLT). XSLT is an XML syntax for defining how to convert one XML file into another. The second part of this chapter shows how to use XSLT to convert an XML-formatted aeronautical weather report to SVG.

Converting Custom Data to SVG

If anyone lives a life that revolves around graphics display, it's a mapmaker. Cartographers are finding XML markup in general and SVG in particular to be excellent vehicles for putting data into a portable format. At present, though, much of the data that is currently available is in custom or proprietary formats.

One such proprietary format was developed by Environmental Systems Research Institute for use by their ArcInfo Geographic Information System. Data created in this system can be exported in an ASCII *ungenerate* form. Such a file contains a series of polygon descriptions, followed by a line with the word `END` on it. Each polygon starts with a line that consists of an integer polygon identification number and the *x*- and *y*-coordinates of the polygon's centroid. This line is followed by the *x*- and *y*-coordinates of the polygon's vertices, one vertex per line. A line with the word `END` on it marks the end of the polygon. Here is a sample file:

```
         1      -0.122432044171565E+03       0.378635608621089E+02
      -0.122418712172884E+03       0.378527169597E+02
      -0.122434402770255E+03       0.378524342437443E+02
      -0.122443301934511E+03       0.378554484803880E+02
      -0.122446316168374E+03       0.378610463416856E+02
      -0.122438565286068E+03       0.378683666259093E+02
      -0.122418712172884E+03       0.378527169591107E+02
END
         2      -122.36                      37.82
      -122.378                     37.826
      -122.377                     37.831
      -122.370                     37.832
      -122.378                     378.826
END
END
```

Converting such a file to SVG is a simple task of inserting the coordinates into the points attribute of `<polygon>` elements. The only twist is that ARC/INFO stores data in Cartesian coordinates, so we will have to flip the *y*-coordinates upside-down. The program

we'll describe will take two parameters: the *input-file* name and the desired *width* of the resulting SVG graphic in pixels.

In addition to those parameters, we're going to need a few global variables to keep track of the data:

- `lineBuffer`, an array of tokens from parsing each line of the file into whitespace-separated words and numbers.
- `singlePolygon`, an array of coordinates for the current polygon.
- `polygonList`, an array of `points` strings for all the polygons.
- `minX`, `minY`, `maxX`, `maxY`, numbers representing the extreme coordinates observed so far in each direction. The minimum variables should be initialized to equal positive Infinity (or the maximum number possible in the programming language), and the maximum variables to negative Infinity. That way, any finite number compared to them will become the new maximum or minimum.

The following algorithm assumes that you have a way of reading the input file one line at a time, and printing the results to an output stream or file. The exact ways to do so will depend on your programming language, but nearly every language can do both:

1. Create a utility subroutine that grabs one token at a time from the input file:

```
function get_token()
{
  if ( lineBuffer is empty ) // out of data?
  {
    read the next line from input-file;
    get rid of leading and trailing whitespace;
    split on whitespace to create an array of tokens;
    put into lineBuffer;
  }
  remove first item in lineBuffer and return it
}
```

2. The main program (after validating and storing the input parameters and initializing the other variables), uses nested loops to process the data file. The outer loop handles the start and end of each polygon, and the inner loop processes the pairs of coordinates. Each polygon starts with an index number and ends with the token `END`. The file as a whole also ends with an additional `END`. Read an index number, initialize `singlePolygon` as a coordinate array, read coordinates until you reach `END`, add that coordinate array to the *polygonList*, and then repeat unless the next token is also `END`:

```
open input-file;

while ((polygonNumber = get_token()) is not "END" )
{
```

```
    singlePolygon = empty list;

    while ((xCoord = get_token()) is not "END" )
    {
      yCoord = get_token();
      append (xCoord, yCoord) to singlePolygon;

      // keep track of minimum and maximum coordinates
      minX = min(xCoord, minX);
      maxX = max(xCoord, maxX);
      minY = min(yCoord, minY);
      maxY = max(yCoord, maxY);
    }
    append singlePolygon to polygonList;
  }

  close input-file ;
```

3. After processing the input file, `polygonList` is an array of arrays of coordinate pairs, and `minX`, `minY`, `maxX`, and `maxY` hold the largest and smallest coordinates in each dimension. But before you can start building your SVG, you need to determine an appropriate scale to fit the *x*-range of the data into the pixel width requested by the user. Initialize additional variables to keep track of the input and output dimensions:

```
deltaX = maxX - minX;
deltaY = maxY - minY;
scale = width / deltaX;
height = deltaY * scale;
height = int(height + 0.5); // round up to integer
```

4. The SVG file itself is constructed by printing markup to a file, substituting in the values from the data:

```
open output;

print the following to output, replacing variables in {}
with their values:

  <!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

  <svg width=" {width} " height=" {height} "
    viewBox="0 0 {deltaX}  {deltaY} "
    xmlns="http://www.w3.org/2000/svg">
  <title>Map constructed from {input-file} </title>
  <g style="fill: none; stroke: black;">
```

5. Process the data arrays to create polygon objects:

```
polygonNumber = 1;
foreach singlePolygon in polygonList
{
```

```
    print ' <polyline id="poly {polygonNumber} " points=" ';
    remove first set of coordinates in singlePolygon;

    n = 0; //coordinate index
    foreach coordinate in singlePolygon
    {
      if (n % 2 == 1)    // y-coordinate
      {
        coordinate = (maxY - coordinate); // invert y coords
      }
      else
      {
        coordinate = (coordinate - minX);
      }
      print the coordinate followed by a space;
    }

    // To avoid long lines, place only 8 coordinates per line
      n = (n + 1) % 8;
      if (n == 0) { print a newline }
    }
    print ' /> ';    // close off the polyline
    polygonNumber++;
  }
```

6. Close off open tags and close the file:

```
print ' </g>\n</svg>\n ';
close output;
```

This pseudocode algorithm is a simplified version of a real program written in Perl. Running the Perl program with the data for the state of Michigan with an output width of 250 pixels produces Figure 15-1. Michigan was chosen because it requires several polygons to draw, and its outline is more visually interesting than that of, say, Colorado. This data came from the US Census Bureau Cartographic Boundary Files website.

Figure 15-1. Conversion from ARC/INFO ungenerate to SVG

Using XSLT to Convert XML Data to SVG

If your data is in XML format, then Extensible Stylesheet Language Transformations (XSLT) may be your best choice for doing the conversion to SVG.

Defining the Task

This example uses XSLT to extract information from an XML file and insert it into an SVG file. The source data is a weather report that is retrieved from *http://w1.weather.gov/xml/current_obs/NNNN.xml*, where *NNNN* represents a four-letter weather station identifier. It is formatted as a Weather Observation Markup Format (OMF) document, according to the National Oceanic and Atmospheric Organization's definition of the data format. Here is some sample data (edited for length) from station KSJC:

```
<current_observation version="1.0">
  <credit>NOAA's National Weather Service</credit>
  <credit_URL>http://weather.gov/</credit_URL>
  <location>San Jose International Airport, CA</location>
  <station_id>KSJC</station_id>
  <latitude>37.37</latitude>
  <longitude>-121.93</longitude>
  <observation_time>Last Updated on Jul 15 2014, 7:53 am PDT
    </observation_time>
  <observation_time_rfc822>Tue, 15 Jul 2014 07:53:00 -0700
    </observation_time_rfc822>
  <weather>Overcast</weather>
  <temperature_string>62.0 F (16.7 C)</temperature_string>
  <temp_f>62.0</temp_f>
  <temp_c>16.7</temp_c>
  <wind_string>West at 5.8 MPH (5 KT)</wind_string>
  <wind_dir>West</wind_dir>
  <wind_degrees>290</wind_degrees>
  <wind_mph>5.8</wind_mph>
  <visibility_mi>10.00</visibility_mi>
  <copyright_url>http://weather.gov/disclaimer.html</copyright_url>
</current_observation>
```

The objective is to extract the reporting station, the date and time, temperature, wind speed and direction, and visibility from the report. The data will be filled into the graphic template of Figure 15-2.

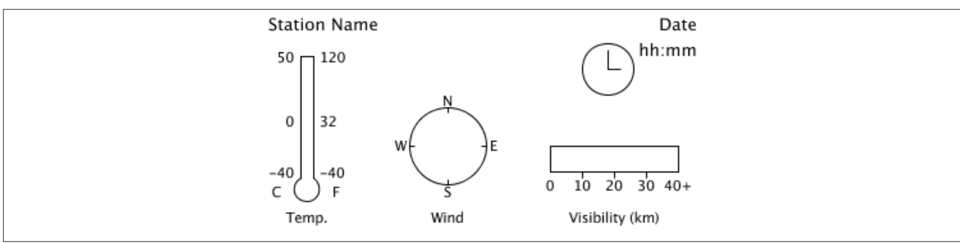


Figure 15-2. Graphic weather template

The elements we're interested in are listed here, along with the plan for displaying them in the final graphic:

`<observation_time_rfc822>`
: In the final graphic, the date and time will be represented in text, and the time will also be shown on an analog clock. The color of the clock face will be light yellow to indicate hours between 6 a.m. and 6 p.m., and light blue for evening and night hours.

`<station_id>`
: The reporting station's call letters. The final graphic will represent this as text.

`<temp_c>`
: The air temperature in degrees Celsius. This will be displayed by coloring in the thermometer to the appropriate level. If the temperature is greater than 0, the coloring will be red; if less than or equal to 0, it will be blue.

`<wind_degrees>`
: The direction is measured in degrees; 0 indicates wind blowing from true north, and 270 indicates wind from the west. This will be represented by a line on the compass.

`<wind_mph>`
: The wind speed is expressed in miles per hour, which will be converted to meters per second.

`<wind_gust_mph>`
: If there are wind gusts, this element will give the speed in miles per hour, which will also be converted to meters per second.

`<visibility_mi>`
: Surface visibility in miles, which will be converted to kilometers. The final graphic will represent this by filling in a horizontal bar. Any visibility above 40 kilometers will be presumed to be unlimited.

How XSLT Works

To convert an OMF source file to its destination SVG format, we will create a list of specifications that tells which elements and attributes in OMF are of interest. These specifications will then detail what SVG elements to generate whenever the processor encounters an item of interest. If you were asking a human to do the transformation by hand, you could write out an English language description:

1. Begin a new SVG document by typing this:

```
<!DOCTYPE svg PUBLIC "-//W3C/DTD SVG 1.0//EN",
 "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
```

2. Go through the source document. As you find each element, look for instructions on how to process it.

3. To process the `<current_observation>` element, add the following code to your file, and then process any child elements as specified in comments:

```
<svg viewBox="0 0 350 200" height="200" width="350">
    <!-- process any child elements from <current_observation> -->
</svg>
```

4. To process a `<station_id>` element, add this code and fill in the blanks:

```
<text font-size="10pt" x="10" y="20">
    <!-- fill in the value of the element's content -->
</text>
```

5. To process a `<temp_c>` element, use its content when following the instructions for "how to draw a thermometer."

6. To process a `<wind_dir>` element, use its contents as you follow the instructions for "how to draw a wind compass."

Similarly, for each other element type, you would indicate where to find the instructions to follow. Then you'd have supplemental instructions like these:

- To draw a thermometer:
  - — Calculate the height of the bar as 50 minus the value you got from the caller.
  - — Determine the appropriate color (`red` or `blue`) based on whether or not the value is greater than 0.
  - — Insert those values where you see the italicized text in the following:

```
<path
  d = "M 25 height 25 90
  A 10 10 0 1 0 35 90
  L 35 height Z"
  style="stroke: none; fill: color ;"/>
<path
```

```
d= "M 25 0 25 90 A 10 10 0 1 0 35 90 L 35 0 Z"
style="stroke: black; fill: none;"/>
```

There would be equally detailed instructions for "how to draw a wind compass" and all the other elements.

Rather than writing the specifications in English and handing them to a human to perform, you write the specifications in the XSLT markup format. You can then hand the XSLT file, along with the XML file, to an XLST processor, and it will process elements and fill in the blanks to produce an output SVG file.

Here is a quick English-to-XSLT translation guide:

English	XSLT
Create an output document of a given type	`<xsl:output method="xml"` `  doctype-public="..." doctype-system="...">`

Process an *element* element	`<xsl:template match="element">`
	`  <!-- output to produce -->`
	`</xsl:template>`

Process any *items* within the current element	`<xsl:apply-templates select="items"/>`
Fill in the value of an *item*	`<xsl:value-of select="item"/>`

Use the value of an *item* as a variable named *var*	`<xsl:variable name="var">`
	`  <!-- instructions to produce item's value -->`
	`</xsl:variable>`

Call another template named *some-name*, and give it a *parameter* with *some-value*	`<xsl:call-template name="some-name">`
	`  <xsl:with-param name="parameter"`
	`    select="some-value"/>`
	`</xsl:call-template>`

Add the following content if the data passes a test	`<xsl:if test="some-test">`
	`   <!-- content -->`
	`</xsl:if>`

Add content if data passes a test; otherwise, add other content	`<xsl:choose>`
	`   <xsl:when test="some-test">`
	`      <!-- content -->`
	`   </xsl:when>`
	`   <xsl:otherwise>`
	`      <!-- other content -->`
	`   </xsl:otherwise>`
	` </xsl:choose>`

Developing an XSL Stylesheet

We'll add details as we proceed, but this is more than enough to start. The XSLT file begins like this, and, for the purposes of this example, is stored in a file named *weather.xsl*:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns="http://www.w3.org/2000/svg">

<xsl:output method="xml" indent="yes"
  doctype-public="-//W3C//DTD SVG 1.0//EN"
  doctype-system=
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd"/>      ❶

<xsl:template match="current_observation">      ❷
<svg width="350" height="200" viewBox="0 0 350 200"
  xmlns="http://www.w3.org/2000/svg">
  <g style="font-family: sans-serif">

  <!-- Process all child elements -->
  <xsl:apply-templates />      ❸
  </g>
</svg>
</xsl:template>
```

❶ The <xsl:output> specifies that the output will be an XML file and that it should be indented nicely. It also generates the appropriate <!DOCTYPE …> instruction.

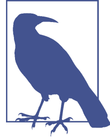
❷ <xsl:template> directs the XSLT processor to generate the specified output whenever it encounters a <current_observation> element. This template will be called only once, because there's only one such element in the source document. It creates the outermost <svg> element and a <g> element for later use.

❸ After outputting the <svg> and <g>, <xsl:apply-templates> directs the processor to find any child elements and generate whatever is specified by their <xsl:template> elements.

This is the markup to process the station_id element:

```
<xsl:template match="station_id">
 <text font-size="10pt" x="10" y="20">
    <xsl:value-of select="."/>
  </text>
</xsl:template>
```

The <xsl:value-of> inserts the value of the selected item. In this case, the . means "the current element."



So far, this example uses only element names as the values of a `match` or `select`. In reality, you can put any XPath expression as a value. XPath is a notation that lets you select parts of an XML document with extreme precision. For example, while processing an XHTML document, you could select only the odd `<td>` elements that are within `<tr>` elements that have a `title` attribute.

While it would be possible to output all the relevant SVG for the temperature within one `<xsl:template>`, a modular approach is easier to read and maintain. XSLT lets you create templates that act somewhat like functions; they don't correspond to any element in the source document, but you may explicitly call them by name and pass parameters to them. Here is the code to draw the thermometer:

```
<xsl:template match="temp_c">
  <xsl:call-template name="draw-thermometer">
    <xsl:with-param name="t" select="."/>
  </xsl:call-template>
</xsl:template>
```

If the value of a parameter is an attribute value or the content of an element, the easiest way to set it is with a `select`. Another way to set the value is to put the content between a beginning and ending tag.

Now you can write the template for `draw-thermometer`. The passed-in parameter determines the height to fill the thermometer and whether the thermometer should be filled with red or blue. Let's build this up in stages. First, extract the parameter and draw the parts that are static:

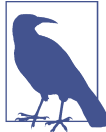
```
<xsl:template name="draw-thermometer">
  <xsl:param name="t" select="0"/>
  <g id="thermometer" transform="translate(10, 40)">
    <path id="thermometer-path" stroke="black" fill="none"
      d= "M 25 0 25 90 A 10 10 0 1 0 35 90 L 35 0 Z"/>

    <g id="thermometer-text" font-size="8pt" font-family="sans-serif">
      <text x="20" y="95" text-anchor="end">-40</text>
      <text x="20" y="55" text-anchor="end">0</text>
      <text x="20" y="5" text-anchor="end">50</text>
      <text x="10" y="110" text-anchor="end">C</text>
      <text x="40" y="95">-40</text>
      <text x="40" y="55">32</text>
      <text x="40" y="5">120</text>
      <text x="50" y="110">F</text>
      <text x="30" y="130" text-anchor="middle">Temp.</text>
  </g>
</g>
</xsl:template>
```

The `<xsl:param>` element lets you provide a default value (in this case, `0`) if no parameter is passed in.

Next, add the code to display the temperature as text. If there was no temperature present, display the value `N/A`. Notice that the parameter name is `t`, but to access its contents, you must say `$t`. This code goes inside the `<g id="thermometer-text">`:

```
<text x="30" y="145" text-anchor="middle">
  <xsl:choose>
    <xsl:when test="$t != ''">
      <xsl:value-of select="round($t)"/>&#176;C /
      <xsl:value-of select="round($t div 5 * 9 + 32)"/>&#176;F
    </xsl:when>
    <xsl:otherwise>N/A</xsl:otherwise>
  </xsl:choose>
</text>
```

The text is set conditionally with the `<xsl:choose>` element, which contains one or more `<xsl:when>` elements. The first one whose `test` succeeds is the one whose output goes into the final document. The `<xsl:otherwise>` element is a catch-all in case all the preceding tests fail.



The formula for conversion of Celsius to Fahrenheit uses `div` for division; this is because the forward slash is already used in XPath to separate levels of element nesting.

The next step is to fill the thermometer. This should be done only if the value of the `t` parameter is not the empty string. The following code uses `<xsl:variable>` to create a variable named `tint` and sets its value to either `red` or `blue`, depending on whether the temperature is above 0 degrees Celsius or not. Variables in XSL are *single-assignment*. Every time the template is called, the variable is set to an initial value, but for the duration of the template, it cannot be changed further. Place this after the closing `</g>` tag of the `thermometer-text` group:

```
<xsl:if test="$t != ''">
  <xsl:variable name="tint">
  <xsl:choose>
    <xsl:when test="$t &gt; 0">red</xsl:when>
    <xsl:otherwise>blue</xsl:otherwise>
  </xsl:choose>
  </xsl:variable>
  <!-- remainder of code -->
</xsl:if>
```

Again, the code uses `<xsl:choose>` to conditionally set the variable. The `test` uses the entity reference `>` for a greater-than symbol to avoid problems with some XSLT processors; if you ever want to produce a less-than symbol, it *must* be written as `<`.

Here is the remainder of the code for filling the thermometer:

```
<!-- "fill" the thermometer by drawing a solid
  rectangle and clipping it to the shape of
  the thermometer -->
<xsl:variable name="h">
  <xsl:choose>
    <xsl:when test="$t &lt; -55">
      <xsl:value-of select="105"/>
    </xsl:when>
    <xsl:when test="$t &gt; 50">
      <xsl:value-of select="0"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="50 - $t"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:variable>

<clipPath id="thermoclip">
  <use xlink:href="#thermometer-path"/>
</clipPath>
<path d="M 10 {$h} h40 V 120 h-40 Z"
  fill="{$tint}" clip-path="url(#thermoclip)"/>
```

The `<xsl:choose>` in the preceding code has two `<xsl:when>` clauses; they limit the height of the "mercury" in case temperatures go above or below the thermometer's limits. The `<xsl:otherwise>` clause sets the height for all other in-range temperatures.

When referring to parameters or variables in the values of attributes of the output document, as in the final `<path>` element, you must enclose them with curly braces.

This would be a good time to test the transformation so far. Before you can test, you have to add an empty template to handle text nodes. XSLT processors are set up with default templates to ensure that they will visit all the elements and text in the source document. The default behavior is to send the text within elements directly to the destination document. In this transformation, you want to throw away any text that you don't specifically want to process, so there's an empty template for text nodes; they will not appear in the SVG file. Finally, you need the closing `</xsl:stylesheet>` tag:

```
<xsl:template match="text()"/>

</xsl:stylesheet>
```

Invoke your XSLT processor on the XML file that contains the weather report. The resulting graphic, Figure 15-3, shows the station name and the thermometer. If you

don't have a standalone XSLT processor, you can add the following lines at the top of the XML file:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="weather.xsl"?>
```

Then open the file in your browser, and it will do the transformation for you.

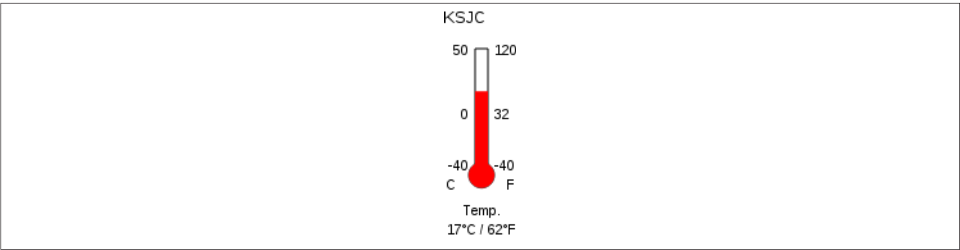


Figure 15-3. XSL-generated SVG file showing thermometer

You have seen that XSLT can do simple arithmetic; it can also do a reasonable amount of string manipulation. Here is the XSLT to display the day and time. It uses the `substring` function to extract the necessary information.

```
<xsl:template match="observation_time_rfc822">
  <xsl:variable name="time" select="."/> ❶

  <text font-size="10pt" x="345" y="20" text-anchor="end">
    <xsl:value-of select="substring($time, 6, 11)"/> ❷
  </text>

  <xsl:call-template name="draw-time-and-clock"> ❸
    <xsl:with-param name="hour"
      select="number(substring($time, 18, 2))"/>
    <xsl:with-param name="minute"
      select="number(substring($time, 21, 2))"/>
  </xsl:call-template>
</xsl:template>
```

❶ For convenience, store the string in a variable rather than having to do many `<xsl:value-of>`s.

❷ The `substring()` function needs the string, starting character index, and number of characters to extract. The first character of the string is index number 1, not 0, as in many other programming languages.

❸ Pass the hour and minute to a named template to do the heavy lifting. The `number()` function converts its string parameter to a true numeric value.

And here is the template that draws the clock face and displays the time as text (the only new construct here is the `format-number()` function):

```
<xsl:template name="draw-time-and-clock">
  <xsl:param name="hour">0</xsl:param>
  <xsl:param name="minute">0</xsl:param>

  <!-- clock face is light yellow from 6 a.m. to 6 p.m.,
       otherwise light blue -->
  <xsl:variable name="tint">
    <xsl:choose>
      <xsl:when test="$hour &gt;= 6 and $hour &lt; 18"
           >#ffffcc</xsl:when>
      <xsl:otherwise>#ccccff</xsl:otherwise>
    </xsl:choose>
  </xsl:variable>

  <!-- calculate angles for hour and minute hand
    of analog clock -->
  <xsl:variable name="hourAngle"
    select="(30 * ($hour mod 12 + $minute div 60)) - 90"/>
  <xsl:variable name="minuteAngle"
    select="($minute * 6) - 90"/>

  <text font-size="10pt" x="345" y="40" text-anchor="end">
    <xsl:value-of select="format-number($hour,00)"/> ❶
    <xsl:text>:</xsl:text> ❷
    <xsl:value-of select="format-number($minute,00)"/>
  </text>
  <g id="clock" transform="translate(255, 30)">
    <circle cx="20" cy="20" r="20" fill="{$tint}"
            stroke="black"/>
    <line transform="rotate({$minuteAngle}, 20, 20)"
      x1="20" y1="20" x2="38" y2="20" stroke="black"/>
    <line transform="rotate({$hourAngle}, 20, 20)"
      x1="20" y1="20" x2="33" y2="20" stroke="black"/>
  </g>
</xsl:template>
```

❶ The `format-number($hour,00)` ensures that the output will have a leading zero if it is less than two digits long.

❷ The `<xsl:text>` element places its contents, which must be pure text, into the output document verbatim. Using `<xsl:text>` helps avoid problems with whitespace; if I had not used it, the newline and indentation would have made its way into the resultant SVG `<text>` element, which would have produced extra space around the colon in the final graphic.

Here is the markup to draw the wind speed indicator:

```
<xsl:template match="wind_degrees">
  <xsl:call-template name="draw-wind">
    <xsl:with-param name="dir" select="number(.)"/>
    <xsl:with-param name="speed"
      select="number(../wind_mph) * 1609.344 div 3600"/> ❶
    <xsl:with-param name="gust"
      select="number(following-sibling::wind_gust_mph) *
        1609.344 div 3600"/>
  </xsl:call-template>
</xsl:template>

<xsl:template name="draw-wind">
  <xsl:param name="dir">0</xsl:param>
  <xsl:param name="speed">0</xsl:param>
  <xsl:param name="gust">0</xsl:param>

  <g id="compass" font-size="8pt" font-family="sans-serif"
    transform="translate(110, 70)">
    <circle cx="40" cy="40" r="30" stroke="black" fill="none"/>
    <!-- tick marks at cardinal directions -->
    <path stroke="black" fill="none"
      d= "M 40 10 L 40 14
      M 70 40 L 66 40
      M 40 70 L 40 66
      M 10 40 L 14 40"/>
    <xsl:if test="$speed &gt;= 0">
      <path d="M 40 40 h 25"
        fill="none" stroke="black"
        transform="rotate({$dir - 90},40,40)"/> ❷
    </xsl:if>
    <text x="40" y="9" text-anchor="middle">N</text>
    <text x="73" y="44">E</text>
    <text x="40" y="80" text-anchor="middle">S</text>
    <text x="8" y="44" text-anchor="end">W</text>
    <text x="40" y="100" text-anchor="middle">Wind (m/sec)</text>
    <text x="40" y="115" text-anchor="middle"> ❸
      <xsl:choose>
        <xsl:when test="$speed &gt;= 0">
          <xsl:value-of select="format-number($speed, 0.)"/>
        </xsl:when>
        <xsl:otherwise>N/A</xsl:otherwise>
      </xsl:choose>
      <xsl:if test="$gust &gt; 0">
        <xsl:text> - </xsl:text>
        <xsl:value-of select="format-number($gust, 0.)"/>
      </xsl:if>
    </text>
  </g>
</xsl:template>
```

❶ Here is a more complex XPath expression. `..` means "this node's parent," so `../wind_mph` will find all the `<wind_mph>` elements that are children of the parent of the `<wind_degree>` element (in this XML file, there is only one such element). The expression that gets the wind gust (if any) uses the more verbose `following-sibling::` specification.

❷ The NOAA specification says that a true north wind is 360 degrees, with 0 degrees implying no wind. You have to subtract 90 degrees, because "north" is –90 degrees in SVG.

❸ This logic for displaying the wind speed (and gusts) as text works even if there is no `<wind_mph>` or `<wind_gust_mph>` element in the weather report. When a nonexistent element's content is converted to a number, the result is `NaN` (Not a Number). When you do any comparison with `NaN`, the result is *always* false. Thus, if there is no `<wind_mph>`, the resulting text is `N/A`, and when there is no `<wind_gust_mph>`, the dash and second number are never output.

Here are the XSLT commands to draw the visibility bar. The first template converts the visibility to kilometers as it passes it to the second template. The visibility bar is 100 pixels wide, so any visibility greater than 40 km is set to 100; anything less is scaled:

```
<xsl:template match="visibility_mi">
  <xsl:call-template name="draw-visibility">
    <xsl:with-param name="v" select="number(.) * 1.609344"/> ❶
  </xsl:call-template>
</xsl:template>

<xsl:template name="draw-visibility">
  <xsl:param name="v">0</xsl:param>
  <g id="visbar" transform="translate(220,110)"
    font-size="8pt" text-anchor="middle">

  <!-- fill in the rectangle if there is a visibility value -->
  <xsl:if test="$v &gt;= 0">
    <xsl:variable name="width"> ❷
      <xsl:choose>
      <xsl:when test="$v &gt; 40">100</xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="$v * 100.0 div 40.0"/>
      </xsl:otherwise>
      </xsl:choose>
    </xsl:variable>
    <rect style="fill:green; stroke:none;"
      x="0" y="0" width="{$width}" height="20"/>
  </xsl:if>

  <rect x="0" y="0" width="100" height="20"
        style="stroke:black; fill:none"/>
```

```
    <path fill="none" stroke="black"
      d="M 25 20 L 25 25 M 50 20 L 50 25 M 75 20 L 75 25"/>

    <text x="0" y="35">0</text>
    <text x="25" y="35">10</text>
    <text x="50" y="35">20</text>
    <text x="75" y="35">30</text>
    <text x="100" y="35">40+</text>
    <text x="50" y="60">
      Visibility (km)
    </text>
    <text x="50" y="75">
      <xsl:choose>
         <xsl:when test="$v &gt;= 0">
           <xsl:value-of select="format-number($v,'0.###')"/> ❸
        </xsl:when>
        <xsl:otherwise>N/A</xsl:otherwise>
      </xsl:choose>
    </text>
  </g>
  </xsl:template>
```

❶ The first template passes the visibility, converted to kilometers, to the next template.

❷ The visibility bar is 100 pixels wide. Visibility greater than 40 km is set to 100; anything less than that is scaled to 100 pixels.

❸ This format string will print a leading zero before the decimal point and three digits after the decimal point.

Putting this all together produces Figure 15-4.

This is only a small sample of what you can do with XSLT. For more information, get *XSLT* by Doug Tidwell (O'Reilly). Chapter 9 of that marvelous book also contains an example of using XSLT to generate SVG from an XML file. If you're serious about manipulating XML, you would be well advised to have that book on your shelf.

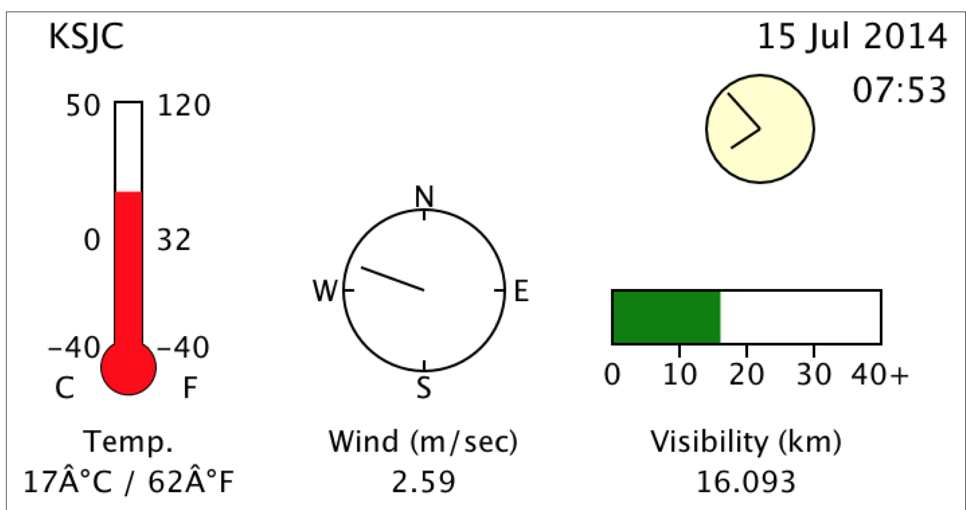


Figure 15-4. XSLT-generated SVG file showing complete data

APPENDIX A

The XML You Need for SVG

The purpose of this appendix is to introduce you to XML. A knowledge of XML is essential if you wish to write SVG documents directly rather than having them generated by some graphics utility.

If you're already acquainted with XML, you don't need to read this appendix. If not, read on. The general overview of XML given in this appendix should be more than sufficient to enable you to work with the SVG documents that you will create. For further information about XML, the O'Reilly books *Learning XML* by Erik T. Ray and *XML in a Nutshell* by Elliotte Rusty Harold and W. Scott Means are invaluable guides.

Note that this appendix makes frequent reference to the formal XML 1.0 specification, which can be used for further investigation of topics that fall outside the scope of SVG. Readers are also directed to Tim Bray's "Annotated XML Specification", which provides an illuminating explanation of the XML 1.0 specification, and Norm Walsh's technical introduction to XML.

You may have noticed that these are not recent publications. Don't be surprised; XML is a solid, long-established standard.

What Is XML?

XML, the Extensible Markup Language, is an Internet-friendly format for data and documents, invented by the World Wide Web Consortium (W3C). The *Markup* denotes a way of expressing the structure of a document within the document itself. XML has its roots in a markup language called SGML (Standard Generalized Markup Language), which is used in publishing and shares this heritage with HTML. XML was created to do for machine-readable documents on the Web what HTML did for human-readable documents—that is, provide a commonly agreed-upon syntax so that processing the underlying format becomes a commodity and documents are made accessible to all users.

Unlike HTML, though, XML comes with very little predefined. HTML developers are accustomed both to the notion of using angle brackets < > for denoting elements (i.e., *syntax)*, and also to the set of element names themselves (i.e., head, body, etc.). XML shares only the former feature, the notion of using angle brackets for denoting elements. Unlike HTML, XML has no predefined elements, but is merely a set of rules that lets you write other languages like HTML.[1] Because XML defines so little, it is easy for everyone to agree to use the XML syntax, and then to build applications on top of it. It's like agreeing to use a particular alphabet and set of punctuation symbols, but not saying which language to use. However, if you're coming to XML from an HTML background, then prepare yourself for the shock of having to choose what to call your tags!

Knowing that XML's roots lie with SGML should help you understand some of XML's features and design decisions. Note that, although SGML is essentially a document-centric technology, XML's functionality also extends to data-centric applications, including SVG. Commonly, data-centric applications do not need all the flexibility and expressiveness that XML provides and limit themselves to employing only a subset of XML's functionality.

Anatomy of an XML Document

The best way to explain how an XML document is composed is to present one. The following example shows an XML document you might use to describe authors:

```
<?xml version="1.0" encoding="us-ascii"?>
<authors>
    <person id="lear">
        <name>Edward Lear</name>
        <nationality>British</nationality>
    </person>
    <person id="asimov">
        <name>Isaac Asimov</name>
        <nationality>American</nationality>
    </person>
    <person id="mysteryperson"/>
</authors>
```

The first line of the document is known as the *XML declaration*. This tells a processing application which version of XML you are using (the version indicator is mandatory) and which character encoding you have used for the document. In the previous example, the document is encoded in ASCII. (The significance of character encoding is covered later in this chapter.) If the XML declaration is omitted, a processor will make certain assumptions about your document. In particular, it will expect it to be encoded in UTF-8, an encoding of the Unicode character set. However, it is best to use the XML

1. To clarify XML's relationship with SGML: XML is an *SGML subset*. By contrast, HTML is an *SGML application*. SVG uses XML to express its operations and thus is an *XML application*.

declaration wherever possible, both to avoid confusion over the character encoding and to indicate to processors which version of XML you're using.

Elements and Attributes

The second line of the example begins an *element*, which has been named *authors*. The contents of that element include everything between the right angle bracket (`>`) in `<authors>` and the left angle bracket (`<`) in `</authors>`. The actual syntactic constructs `<authors>` and `</authors>` are often referred to as the element *start tag* and *end tag*, respectively. Do not confuse tags with elements! Note that elements may include other elements, as well as text. An XML document must contain exactly one *root element*, which contains all other content within the document. The name of the root element defines the type of the XML document.

Elements that contain both text and other elements simultaneously are classified as *mixed content*. The SVG `<text>` element is such an element; it can contain text and `<tspan>` elements.

The sample *authors* document uses elements named `person` to describe the authors themselves. Each person element has an *attribute* named `id`. Unlike elements, attributes can contain only textual content. Their values must be surrounded by quotes. Either single quotes (`'`) or double quotes (`"`) may be used, as long as you use the same kind of closing quote as the opening one.

Within XML documents, attributes are frequently used for *metadata* (i.e., data about data)—describing properties of the element's contents. This is the case in our example, where `id` contains a unique identifier for the person being described.

As far as XML is concerned, it does not matter in what order attributes are presented in the element start tag. For example, these two elements contain exactly the same information as far as an XML 1.0 conformant processing application is concerned:

```
<animal name="dog" legs="4"/>
<animal legs="4" name="dog"/>
```

On the other hand, the information presented to an application by an XML processor on reading the following two lines will be different for each animal element because the ordering of elements is significant:

```
<animal><name>dog</name><legs>4</legs></animal>
<animal><legs>4</legs><name>dog</name></animal>
```

XML treats a set of attributes like a bunch of stuff in a bag—there is no implicit ordering —while elements are treated like items on a list, where ordering matters.

New XML developers frequently ask when it is best to use attributes to represent information and when it is best to use elements. As you can see from the authors example,

if order is important to you, then elements are a good choice. In general, there is no hard-and-fast "best practice" for choosing whether to use attributes or elements.

The final author described in our document has no information available. All we know about this person is his or her ID, `mysteryperson`. The document uses the XML shortcut syntax for an empty element. The following is a reasonable alternative:

```
<person id="mysteryperson"></person>
```

Name Syntax

XML 1.0 has certain rules about element and attribute names. In particular:

- Names are case-sensitive: e.g., `<person/>` is not the same as `<Person/>`.
- Names beginning with *xml* (in any permutation of uppercase or lowercase) are reserved for use by XML 1.0 and its companion specifications.
- A name must start with a letter or an underscore, not a digit, and may continue with any letter, digit, underscore, or period.[2]

A precise description of names can be found in section 2.3 of the XML 1.0 specification. The rules for names in XML 1.1 are slightly different, primarily with regard to Unicode characters. SVG uses the XML 1.0 rules.

Well-Formed

An XML document that conforms to the rules of XML syntax is known as *well-formed*. At its most basic level, well-formedness means that elements should be properly matched, and all opened elements should be closed. A formal definition of well-formedness can be found in section 2.1 of the XML 1.0 specification. Table A-1 shows some XML documents that are not well-formed.

2. Actually, a name may also contain a colon, but the colon is used to delimit a *namespace prefix* and is not available for arbitrary use. For more information, see Tim Bray's "XML Namespaces by Example."

Table A-1. Examples of poorly formed XML documents

Document	Reason why it's not well-formed
<foo> <bar> </foo> </bar>	The elements are not properly nested because foo is closed while inside its child element bar.
<foo> <bar> </foo>	The bar element was not closed before its parent, foo, was closed.
<foo baz> </foo>	The baz attribute has no value. While this is permissible in HTML (e.g., <table border>), it is forbidden in XML.
<foo baz=23> </foo>	The baz attribute value, 23, has no surrounding quotes. Unlike HTML, all attribute values must be quoted in XML.

Comments

As in HTML, it is possible to include *comments* within XML documents. XML comments are intended to be read only by people. With HTML, developers have occasionally employed comments to add application-specific functionality. For example, the server-side include functionality of most web servers uses instructions embedded in HTML comments. XML provides other means of indicating application-processing instructions,[3] so comments should not be used for any purpose other than those for which they were intended.

The start of a comment is indicated with <!--, and the end of the comment with -->. Any sequence of characters, aside from the string --, may appear within a comment.

Comments tend to be used more in XML documents intended for human consumption than those intended for machine consumption. The <desc> and <title> elements in SVG obviate much of the need for comments.

Entity References

Another feature of XML that is occasionally useful when writing SVG documents is the mechanism for *escaping* characters.

Because some characters have special significance in XML, there needs to be a way to represent them. For example, in some cases the < symbol might really be intended to mean *less than* rather than to signal the start of an element name. Clearly, just inserting

3. A discussion of *processing instructions (PIs)* is outside the scope of this book. For more information on PIs, see section 2.6 of the XML 1.0 specification, at *http://www.w3.org/TR/REC-xml#sec-pi*.

the character without any escaping mechanism would result in a poorly formed document because a processing application would assume you were starting another element. Another instance of this problem is needing to include both double quotes and single quotes simultaneously in an attribute's value. Here's an example that illustrates both these difficulties:

```
<badDoc>
  <para>
    I'd really like to use the < character
  </para>
  <note title="On the proper 'use' of the " character"/>
</badDoc>
```

XML avoids this problem by the use of the *predefined entity reference*. The word *entity* in the context of XML simply means a unit of content. The term *entity reference* means just that, a symbolic way of referring to a certain unit of content. XML predefines entities for the following symbols: left angle bracket (`<`), right angle bracket (`>`), apostrophe (`'`), double quote (`"`), and ampersand (`&`).

An entity reference is introduced with an ampersand (`&`), which is followed by a name (using the word *name* in its formal sense, as defined by the XML 1.0 specification), and terminated with a semicolon (`;`). Table A-2 shows how the five predefined entities can be used within an XML document.

Table A-2. Predefined entity references in XML 1.0

Literal character	Entity reference
<	<
>	>
'	'
"	"
&	&

Here's the problematic document revised to use entity references:

```
<badDoc>
  <para>
    I'd really like to use the &lt; character
  </para>
  <note title="On the proper &apos;use&apos;
              of the &quot; character"/>
</badDoc>
```

Being able to use the predefined entities is all you need for SVG; in general, entities are provided as a convenience for human-created XML. XML 1.0 allows you to define your own entities and use entity references as "shortcuts" in your document. Section 4 of the XML 1.0 specification describes the use of entities.

Character References

You are likely to find *character references* in the context of SVG documents. Character references allow you to denote a character by its numeric position in the Unicode character set (this position is known as its *code point*). Table A-3 contains a few examples that illustrate the syntax.

Table A-3. Example character references in UTF-8

Actual character	Character reference
`1`	`1`
`A`	`A`
`Ñ`	`Ñ`
`®`	`®`

Note that the code point can be expressed in decimal or, with the use of *x* as a prefix, in hexadecimal.

Character Encodings

The subject of character encodings is frequently a mysterious one for developers. Most code tends to be written for one computing platform and, normally, to run within one organization. Although the Internet is changing things quickly, most of us have never had cause to think too deeply about internationalization.

XML, designed to be an Internet-friendly syntax for information exchange, has internationalization at its very core. One of the basic requirements for XML processors is that they support the Unicode standard character encoding. Unicode attempts to include the requirements of all the world's languages within one character set. Consequently, it is very large!

Unicode Encoding Schemes

Unicode 3.0 has over 57,700 code points, each of which corresponds to a character.[4] If one were to express a Unicode string by using the position of each character in the character set as its encoding (in the same way as ASCII does), expressing the whole range of characters would require four octets[5] for each character. Clearly, if a document is written in 100 percent American English, it will be four times larger than required—

4. You can obtain charts of all these characters online by visiting *http://www.unicode.org/charts/*.

5. An *octet* is a string of 8 binary digits, or bits. A *byte* is commonly, but not always, considered the same thing as an octet.

all the characters in ASCII fitting into a 7-bit representation. This places a strain both on storage space and on memory requirements for processing applications.

Fortunately, two encoding schemes for Unicode alleviate this problem: *UTF-8* and *UTF-16*. As you might guess from their names, applications can process documents in these encodings in 8- or 16-bit segments at a time. When code points are required in a document that cannot be represented by one chunk, a bit pattern is used that indicates that the following chunk is required to calculate the desired code point. In UTF-8, this is denoted by the most significant bit of the first octet being set to 1.

This scheme means that UTF-8 is a highly efficient encoding for representing languages using Latin alphabets, such as English. All of the ASCII character set is represented natively in UTF-8—an ASCII-only document and its equivalent in UTF-8 are byte-for-byte identical.

This knowledge will also help you debug encoding errors. One frequent error arises because of the fact that ASCII is a proper subset of UTF-8—programmers get used to this fact and produce UTF-8 documents, but use them as if they were ASCII. Things start to go awry when the XML parser processes a document containing, for example, characters such as *Á*. Because this character cannot be represented using only one octet in UTF-8, this produces a two-octet sequence in the output document; in a non-Unicode viewer or text editor, it looks like a couple of characters of garbage.

Other Character Encodings

Unicode, in the context of computing history, is a relatively new invention. Native operating system support for Unicode is by no means universal. For instance, older systems like Windows 95 and 98 do not have it.

XML 1.0 allows a document to be encoded in any character set registered with the Internet Assigned Numbers Authority (IANA). European documents are commonly encoded in one of the *ISO Latin* character sets, such as ISO-8859-1. Japanese documents commonly use *Shift-JIS*, and Chinese documents use *GB2312* and *Big 5*.

A full list of registered character sets is maintained by the Internet Assigned Numbers Authority (IANA).

XML processors are not required by the XML 1.0 specification to support any more than UTF-8 and UTF-16, but most commonly support other encodings, such as US-ASCII and ISO-8859-1. Although most SVG transactions are currently conducted in ASCII (or the ASCII subset of UTF-8), there is nothing to stop SVG documents from containing, say, Korean text. You will, however, probably have to dig into the encoding support of your computing platform to find out if it is possible for you to use alternative encodings.

Validity

In addition to well-formedness, XML 1.0 offers another level of verification, called *validity*. To explain why validity is important, let's take a simple example. Imagine you invented a simple XML format for your friends' telephone numbers:

```
<phonebook>
  <person>
    <name>Albert Smith</name>
    <number>123-456-7890</number>
  </person>
  <person>
    <name>Bertrand Jones</name>
    <number>456-123-9876</number>
  </person>
</phonebook>
```

Based on your format, you also construct a program to display and search your phone numbers. This program turns out to be so useful, you share it with your friends. However, your friends aren't so hot on detail as you are, and try to feed your program this phone book file with a `<phone>` element instead of a `<number>` element:

```
<phonebook>
  <person>
    <name>Melanie Green</name>
    <phone>123-456-7893</phone>
  </person>
</phonebook>
```

Note that, although this file is perfectly well-formed, it doesn't fit the format you prescribed for the phone book, and you find you need to change your program to cope with this situation. If your friends had used `number` as you did to denote the phone number, and not `phone`, there wouldn't have been a problem. However, as it is, this second file is not a valid phone book document.

For validity to be a useful general concept, you need a machine-readable way of saying what a valid document is; that is, which elements and attributes must be present and in what order. XML 1.0 achieves this by introducing *document type definitions* (DTDs). For the purposes of SVG, you don't need to know much about DTDs. Rest assured that SVG does have a DTD, and it spells out in detail exactly which combinations of elements and attributes make up a valid document.

Document Type Definitions (DTDs)

The purpose of a DTD is to express the allowed elements and attributes in a certain document type and to constrain the order in which they must appear within that document type. A DTD contains declarations defining the element types and attribute lists. A DTD may span more than one file, and the SVG 1.1 specification uses a modularized

DTD spread over more than a dozen files. However, the mechanism for including one file inside another—parameter entities—is outside the scope of this book. It is common to mistakenly conflate element and element types. The distinction is that an element is the actual instance of the structure as found in an XML document, whereas the element type is the kind of element that the instance is.

Putting It Together

What *is* important to you is knowing how to link a document to its defining DTD. This is done with a document type declaration `<!DOCTYPE ...>`, inserted at the beginning of the XML document, after the XML declaration in our fictitious example:

```
<?xml version="1.0" encoding="us-ascii"?>
<!DOCTYPE authors SYSTEM "http://example.com/authors.dtd">
<authors>
  <person id="lear">
    <name>Edward Lear</name>
    <nationality>British</nationality>
  </person>
  <person id="asimov">
    <name>Isaac Asimov</name>
    <nationality>American</nationality>
  </person>
  <person id="mysteryperson"/>
</authors>
```

This example assumes the DTD file has been placed on a web server at *example.com*. Note that the document type declaration specifies the root element of the document, not the DTD itself. You could use the same DTD to define *person*, *name*, or *nationality* as the root element of a valid document. Certain DTDs, such as the DocBook DTD for technical documentation,[6] use this feature to good effect, allowing you to provide the same DTD for multiple document types.

A validating XML processor is obliged to check the input document against its DTD. If it does not validate, the document is rejected. To return to the phone book example, if your application validated its input files against a phone book DTD, you would have been spared the problems of debugging your program and correcting your friend's XML because your application would have rejected the document as being invalid. Some of the programs that read SVG files have a validating XML processor built into them to assure they have valid input (and to keep you honest!). The kinds of XML processors that are available are discussed in "Tools for Processing XML" on page 295.

6. See *http://www.docbook.org*.

XML Namespaces

XML 1.0 lets developers create their own elements and attributes, but this leaves open the potential for overlapping names. `<title>` may mean the name of a book in one context, but it may mean the prefix for a person's name (Ms., Dr., etc.) in a different context. The Namespaces in XML specification provides a mechanism developers can use to identify particular vocabularies using Uniform Resource Identifiers (URIs).

SVG uses the URI *http://www.w3.org/2000/svg* for its namespace. The URI is just an identifier—opening that page in a web browser reveals some links to the SVG, XML 1.0, and Namespaces in XML specifications. Programs processing documents with multiple vocabularies can use the namespaces to figure out which vocabulary they are handling at any given point in a document.

SVG applies the namespace in the root element of SVG documents:

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

The `xmlns` attribute, which defines the namespace, is actually provided as a default value by the SVG DTD. However, some browsers will not render an SVG document if you don't use the namespace explicitly. (If the namespace does appear, it must have the exact value shown earlier.) The namespace declaration applies to all of the elements contained by the element in which the declaration appears, including the containing element. This means that the element named *svg* is in the namespace `http://www.w3.org/2000/svg`.

SVG uses the "default namespace" for its content, using the SVG element names without any prefix. Namespaces can also be applied using prefixes, as shown here:

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

In this case, the namespace URI `http://www.w3.org/2000/svg` would apply to all elements using a prefix of *svgns*. The SVG 1.0 DTD won't validate against such documents.

Namespaces are very simple on the surface but are a well-known field of combat in XML arcana. For more information on namespaces, see *XML in a Nutshell* or *Learning XML* (both O'Reilly).

Tools for Processing XML

Many parsers exist for using XML with many different programming languages. Most are freely available, the majority being open source.

Selecting a Parser

An XML parser typically takes the form of a library of code that you interface with your own program. The SVG program hands the XML over to the parser, and it hands back information about the contents of the XML document. Typically, parsers do this either via events or via a Document Object Model (DOM).

With event-based parsing, the parser calls a function in your program whenever a parse event is encountered. Parse events include things like finding the start of an element, the end of an element, or a comment. Most Java event-based parsers follow a standard API called SAX, which is also implemented for other languages such as Python and Perl.

DOM-based parsers work in a markedly different way. They consume the entire XML input document and hand back a tree-like data structure that the SVG software can interrogate and alter. The DOM is a W3C standard with its own documentation.

As XML matures, hybrid techniques that give the best of both worlds are emerging. If you're interested in finding out what's available and what's new for your favorite programming language, keep an eye on the following online sources:

XML.com Resource Guide
  http://www.xml.com/resourceguide/

Free XML Tools Guide
  http://www.garshol.priv.no/download/xmltools/

XSLT Processors

Many XML applications involve transforming one XML document into another XML document or into HTML. The W3C has defined a special language called *XSLT* for doing transformations. XSLT processors are becoming available for all major programming platforms.

XSLT works by using a *stylesheet*, which contains templates that describe how to transform elements from an XML document. These templates typically specify what XML to output in response to a particular element or attribute. Using a W3C technology called *XPath* gives you the flexibility to say not only "do this for every *person* element," but to give instructions as complex as "do this for the third *person* element whose *name* attribute is *Fred*."

Because of this flexibility, some applications have sprung up for XSLT that aren't really transformation applications at all, but take advantage of the ability to trigger actions on certain element patterns and sequencers. Combined with XSLT's ability to execute custom code via extension functions, the XPath language has enabled applications such as document indexing to be driven by an XSLT processor. You can see a brief introduction to XSLT in Chapter 15.

The W3C specifications for XSLT and XPath can be found at *http://w3.org/TR/xslt* and *http://w3.org/TR/xpath*, respectively.

APPENDIX B
Introduction to Stylesheets

As mentioned in Chapter 5, some attributes of SVG elements control the element's geometry. An example of one such attribute would be the `cx` (center x) attribute of a `<circle>`. Other attributes, such as `fill`, control the element's presentation. Stylesheets provide a way for you to separate the presentation from the geometric structure; this lets you control the visual display of many different SVG elements (and even documents) by changing one stylesheet referenced by all the documents.

Anatomy of a Style

A *style* is a specification of a visual property for an element and the value that you would like that property to have. The property name and the value are separated by a colon. For example, to say that you want the stroke color for some element to be blue, the appropriate style specifier would be `stroke: blue`.

To specify multiple properties in a style, you separate the specifiers with semicolons. The following style specifier sets the stroke color to red, the stroke width to three pixels, and the fill color to a light blue. The last property-value pair is followed by a semicolon. This is not necessary, but is done to give the style a more consistent look.

```
stroke: red; stroke-width: 3px; fill: #ccccff;
```

Inline Styles: The style Attribute

Once you have determined the visual properties you'd like, you must select the element or elements to which they apply. The simplest way to apply a style specification to a single element is to make that specification the value of a `style` attribute. So, if you want the preceding specification to apply to a particular `<circle>` in your document, you write this:

```
<circle cx="50" cy="40" r="12"
    style="stroke: red; stroke-width: 3px; fill: #ccccff;"/>
```

Internal Stylesheets

If you want the style specification to apply to all <circle> elements in a single document, add an internal stylesheet. A stylesheet consists of *selectors* (the names of the elements you want to affect) and the style specifications for those selectors. The style specification is enclosed in curly braces. The following applies styles to <circle> and <rect> elements:

```
<style type="text/css"><![CDATA[
  circle {
    stroke: red; stroke-width: 3px;
    fill: #ccccff;
  }
  rect { fill: gray; stroke: black; }
]]></style>
```

When you put a <style> element into an SVG document, you should enclose its contents within <![CDATA[and]]>. This notation tells XML parsers that the contents are pure character *data* and should not, under any circumstances, be treated as information for XML to parse.

Because this stylesheet is within a document, it applies to that document alone. If you have many documents, all of whose circles and rectangles appear as specified in the preceding example, take the specifiers, without the <style> or <![CDATA[tags, and put them into a separate file named *myStyle.css*. In each SVG document, insert the following processing instruction:

```
<?xml-stylesheet href="myStyle.css" type="text/css"?>
```

Then, at a later point, if you decide that all rectangles should be filled with a light green and outlined in dark green, you can simply change the specification in *myStyle.css* to read as follows:

```
rect {fill: #ccffcc; stroke: #006600;}
```

and all your documents, once redisplayed, will have green rectangles instead of gray rectangles.

Style Selector Classes

The preceding stylesheet affects all <rect> and <circle> elements. Let's say, though, that you want only some circles in your documents to be styled. Write your stylesheet with a class specifier as follows, where the dot after circle indicates that the following identifier is a class name:

```
circle.special {
  stroke: red; stroke-width: 3px;
  fill: #ccccff;
}
```

If, in your SVG document, you had the following elements, the first circle would show up as the default (black fill, no stroke), and the second would take on the style attributes as its class name matches the class identifier in the stylesheet:

```
<circle cx="40" cy="40" r="20"/>
<circle cx="60" cy="20" r="10" class="special"/>
```

It is possible to specify a generic class that can apply to any element. Presume that several different graphic objects serve as warning symbols. You would like them to have a yellow fill and a red border. You could write this selector, which consists only of a class name and its style specifier:

```
.warning { fill: yellow; stroke: red; }
```

This generic class may now be applied to any SVG element. In the following example, both the rectangle and triangle will have yellow interiors and red outlines:

```
<rect class="warning" x="5" y="10" width="20" height="30"/>
<polygon class="warning" points="40 40, 40 60, 60 50"/>
```

The `class` attribute may contain the names of several classes separated by whitespace; their combined properties will be applied to the element in question. The following markup adds a generic class named `seeThrough` for translucency to the previous example and then applies both classes to the polygon:

```
<svg width="100" height="100" viewBox="0 0 100 100">
  <style type="text/css"><![CDATA[
    .warning { fill: yellow; stroke: red; }
    .seeThrough { fill-opacity: 0.25; stroke-opacity: 0.5; }
  ]]></style>
  <rect class="warning" x="5" y="10" width="20" height="30"/>
  <polygon class="warning seeThrough" points="40 40, 40 60, 60 50"/>
</svg>
```

Using CSS with SVG

The question then becomes: which attributes in SVG elements can also be specified in a stylesheet? Table B-1 is a list of the properties you may use in a stylesheet, the valid values (with default value shown in boldface where appropriate), and the elements to

which they may be applied. It is a modified version of the property index from the SVG specification.[1]

The value of `fill` and `stroke` is a *`paint`* value, which is one of the following:

- `none`
- `currentColor`
- A color specification, as described in "Stroke Color" on page 41
- A construction of the form `url(…)` that refers to a gradient or pattern

You can specify fallback paint options, in case there is an error loading a gradient or pattern; the options are given in a whitespace-separated list starting with the preferred value.

1. Copyright © 2001 World Wide Web Consortium (Massachusetts Institute of Technology, Institut National de Recherche en Informatique et en Automatique, Keio University). All Rights Reserved. *http://www.w3.org/Consortium/Legal/*.

Table B-1. CSS property table for SVG

Name	Values	Applies to
alignment-baseline	**auto** | baseline | before-edge | text-before-edge | middle | after-edge | text-after-edge | ideographic | alphabetic | hanging | mathematical	<tspan>, <tref>, <altGlyph>, <textPath>
baseline-shift	**baseline** | sub | super | *percentage* | *length*	<tspan>, <tref>, <altGlyph>, <textPath> elements
clip-path	*uri*	Container elements and graphics elements
clip-rule	nonzero | evenodd | class=noxref	Graphics elements within a <clipPath> element
color	*color*	Used to provide a potential indirect value (currentColor) for fill, stroke, stop-color, flood-color, and lighting-color
color-interpolation	auto | **sRGB** | linearRGB	Container elements, graphics elements, and <animateColor>
color-interpolation-filters	auto | sRGB | **linearRGB**	Filter primitives
color-profile	**auto** | sRGB | *name* | *uri*	<image> elements that refer to raster images
color-rendering	**auto** | optimizeSpeed | optimizeQuality	Container elements, graphics elements and <animateColor>
cursor	*uri* | **auto** | crosshair | default | pointer | move | e-resize | ne-resize | nw-resize | n-resize | se-resize | sw-resize | s-resize | w-resize | text | wait | help	Container elements and graphics elements
direction	**ltr** | rtl	<text>, <tspan>, <tref>, and <textPath> elements

<table>
<tr><th>Name</th><th>Values</th><th>Applies to</th></tr>
<tr><td>display</td><td>inline | block | list-item | run-in | compact | marker | table | inline-table | table-row-group | table-header-group | table-footer-group | table-row | table-column-group | table-column | table-cell | table-caption | none</td><td><svg>, <g>, <switch>, <a>, <foreignObject>, graphics elements (including the <text> element), and text subelements (i.e., <tspan>, <tref>, <altGlyph>, <textPath>). All values except none are treated the same for elements in an SVG, enabling display of the graphic.</td></tr>
<tr><td>dominant-baseline</td><td>auto | use-script | no-change | reset-size | alphabetic | hanging | ideographic | mathematical | central | middle | text-after-edge | text-before-edge | text-top | text-bottom</td><td>Text content elements</td></tr>
<tr><td>enable-background</td><td>accumulate | new [x y width height]</td><td>Container elements</td></tr>
<tr><td>fill</td><td>See description of paint at end of table for possible values; the default is black</td><td>Shapes and text content elements</td></tr>
<tr><td>fill-opacity</td><td>opacity-value (default 1)</td><td>Shapes and text content elements</td></tr>
<tr><td>fill-rule</td><td>nonzero | evenodd</td><td>Shapes and text content elements</td></tr>
<tr><td>filter</td><td>uri | none</td><td>Container elements and graphics elements</td></tr>
<tr><td>flood-color</td><td>currentColor | color specifier (default black)</td><td><feFlood> elements</td></tr>
<tr><td>flood-opacity</td><td>alphavalue (default 1)</td><td><feFlood> elements</td></tr>
<tr><td>font</td><td>font-style, font-variant, font-weight, font-size line-height, font-family | caption | icon | menu | message-box | small-caption | status-bar</td><td>Text content elements</td></tr>
<tr><td>font-family</td><td>series of family-name or generic-family</td><td>Text content elements</td></tr>
<tr><td>font-size</td><td>absolute-size | relative-size | length | percentage</td><td>Text content elements</td></tr>
</table>

Name	Values	Applies to
`font-size-adjust`	*number* | **none**	Text content elements
`font-stretch`	**normal** | wider | narrower | ultra-condensed | extra-condensed | condensed | semi-condensed | semi-expanded | expanded | extra-expanded | ultra-expanded	Text content elements
`font-style`	**normal** | italic | oblique	Text content elements
`font-variant`	**normal** | small-caps	Text content elements
`font-weight`	**normal** | bold | bolder | lighter | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900	Text content elements
`glyph-orientation-horizontal`	*angle* (default 0deg)	Text content elements
`glyph-orientation-vertical`	**auto** | *angle*	Text content elements
`image-rendering`	**auto** | optimizeSpeed | optimizeQuality	Images
`kerning`	**auto** | *length*	Text content elements
`letter-spacing`	**normal** | *length*	Text content elements
`lighting-color`	currentColor | *color specification* (default white)	<feDiffuseLighting> and <feSpecularLighting> elements
`marker`, `marker-end`, `marker-mid`, `marker-start`	**none** | *uri*	<path>, <line>, <polyline>, and <polygon> elements
`mask`	*uri* | **none**	Container elements and graphics elements
`opacity`	*alphavalue* (default 1)	Container elements and graphics elements

<table>
<tr><th>Name</th><th>Values</th><th>Applies to</th></tr>
<tr><td>overflow</td><td>visible | hidden | scroll | auto</td><td>Elements that establish a new viewport, <pattern> elements, and <marker> elements</td></tr>
<tr><td>pointer-events</td><td>**visiblePainted** | visibleFill | visibleStroke | visible | painted | fill | stroke | all | none</td><td>Graphics elements</td></tr>
<tr><td>shape-rendering</td><td>**auto** | optimizeSpeed | crispEdges | geometricPrecision</td><td>Shapes</td></tr>
<tr><td>stop-color</td><td>currentColor | *color specification* (default black)</td><td><stop> elements</td></tr>
<tr><td>stop-opacity</td><td>*alphavalue* (default 1)</td><td><stop> elements</td></tr>
<tr><td>stroke</td><td>See description of *paint* at end of table for possible values; the default is **none**</td><td>Shapes and text content elements</td></tr>
<tr><td>stroke-dasharray</td><td>**none** | *dasharray*</td><td>Shapes and text content elements</td></tr>
<tr><td>stroke-dashoffset</td><td>*dashoffset* (default 0)</td><td>Shapes and text content elements</td></tr>
<tr><td>stroke-linecap</td><td>**butt** | round | square</td><td>Shapes and text content elements</td></tr>
<tr><td>stroke-linejoin</td><td>**miter** | round | bevel</td><td>Shapes and text content elements</td></tr>
<tr><td>stroke-miterlimit</td><td>*miterlimit* (default 4)</td><td>Shapes and text content elements</td></tr>
<tr><td>stroke-opacity</td><td>*opacity-value* (default 1)</td><td>Shapes and text content elements</td></tr>
<tr><td>stroke-width</td><td>*width* (default 1)</td><td>Shapes and text content elements</td></tr>
<tr><td>text-anchor</td><td>**start** | middle | end</td><td>Text content elements</td></tr>
<tr><td>text-decoration</td><td>**none** | underline | overline | line-through | blink</td><td>Text content elements</td></tr>
</table>

<table>
<tr><th>Name</th><th>Values</th><th>Applies to</th></tr>
<tr><td>text-rendering</td><td>auto | optimizeSpeed | optimizeLegibility | geometricPrecision</td><td><text> elements</td></tr>
<tr><td>unicode-bidi</td><td>normal | embed | bidi-override</td><td>Text content elements</td></tr>
<tr><td>visibility</td><td>visible | hidden | collapse</td><td>Graphics elements (including the <text> element) and text sub-elements (i.e., <tspan>, <tref>, <altGlyph>, <textPath>, and <a>)</td></tr>
<tr><td>word-spacing</td><td>normal | length</td><td>Text content elements</td></tr>
<tr><td>writing-mode</td><td>lr-tb | rl-tb | tb-rl | lr | rl | tb</td><td><text> elements</td></tr>
</table>

APPENDIX C

Programming Concepts

Many graphic designers want to use the scripting capability of SVG as described in Chapter 13. If they're not familiar with programming, they tend to practice what might be called *voodoo scripting*. In the popular-culture stereotype,[1] voodoo works by reciting a mysterious spell and hoping that your enemies die horribly. Voodoo scripting works by copying someone else's mysterious script into your SVG document and hoping that your document continues to live. We're under no illusion (nor even a spell) that reading this brief, purposely oversimplified summary will turn you into a master programmer. Our goal is simply to introduce enough of the elementary programming concepts to remove some of the mystery from the scripts that you copy and modify. The particular programming language that we will discuss in this appendix is called ECMAScript; it is the standardized version of the JavaScript language. The concepts used in ECMAScript are common to many other programming languages.

Constants

A *constant* is a fancy word for a number or string of characters that never changes. Examples are `2`, `2.71828`, `"message"`, and `'communication'`. The last two are called *string constants*. In ECMAScript, you can use either single or double quotes to mark the boundaries of a string. This is good if you ever need to write things like `"O'Reilly Media"` or `'There is no "there" there.'`

You will sometimes see the two Boolean constants `true` and `false`, which are used for yes-or-no situations.

1. Unlike the actual practice of Santería and voudon, which are much more complex and not inherently evil.

Variables

A *variable* is a block of memory reserved to hold some value that may change from time to time. You can think of it as a mailbox with a name on it; the mailbox holds a slip of paper with information written on it. Let's say you need to keep track of the current width of a rectangle and need to store a changeable message; in ECMAScript you define these variables like this:

```
var currentWidth;
var message;
```

You may visualize them as shown in Figure C-1.

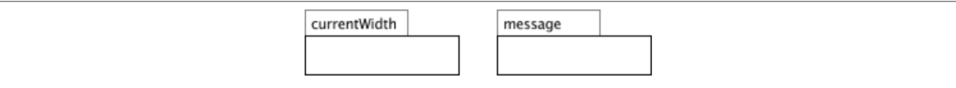


Figure C-1. Two empty variables

Variables defined this way have nothing in their "mailbox"; the technical term is that these variables contain the undefined value. Variable names must start with a letter or an underscore and can contain only letters, digits, and underscores. They are case sensitive, so `width`, `Width`, and `WIDTH` are names of three different variables.

Assignment and Operators

You can put a value into a variable by using an assignment statement, which starts with a variable name, an equal sign, and the value. Examples:

```
currentWidth = 32;
message = "I love SVG.";
```

You can read these as "set the value of `currentWidth` equal to `32`" and "set the value of `message` equal to `"I love SVG."`" In reality, this statement works from right to left; whatever is on the righthand side of the equal sign is placed into the variable on the lefthand side. Note that all our ECMAScript statements end with a semicolon. There are cases where you don't need one, but we'd rather have the semicolon and not need it than need it and not have it. Figure C-2 shows the "after" picture for these assignments.

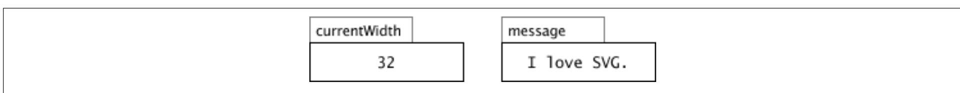


Figure C-2. Two assigned variables

Actually, we told you a small lie a few sentences ago. Whatever the righthand side of the equal sign *works out to* goes into the variable on the lefthand side. This lets us do mathematical operations, as in the following code:

```
var info;        ❶
info = 7 + 2;      ❷
info = 7 * 2;     ❸
info = info + 1;      ❹
info = "door";      ❺
info = info + "bell"      ❻
```

❶ Create an empty variable named `info`.

❷ `info` will now contain the value `9` (7 plus 2). You use the minus sign (`-`) for subtraction.

❸ The asterisk is used for multiplication, because the multiplication symbol is too easily confused with the letter `x`. `info` will contain `14` (7 times 2). The previous value of 9 will be discarded. You use the forward slash (`/`) for division.

❹ This is illegal in high school algebra, but not in ECMAScript. Start with the righthand side of the equal sign: take the current value of `info`, which is `14`, and add 1 to it. The righthand side works out to `15`. This result goes into the variable on the left side of the equal, which just happens to be `info`. At the end of this statement, `info` will contain the value `15`.

❺ This takes the string `"door"` and puts it into `info`. In ECMAScript, a variable can hold any sort of information at any time.

❻ This takes the current value of `info`, which is `"door"`, and "adds" (appends) the string constant `"bell"` to the end of it. The resulting value is the word `"doorbell"`, which goes back into `info` on the left side of the equal sign. The plus sign is the only operator that works with strings; you can't subtract, multiply, or divide them. Be careful when mixing addition and string concatenation: `"The answer is " + 2 + 2` produces `"The answer is 22"`; `"The answer is " + (2 + 2)` produces `"The answer is 4"`

You can define a variable and set its initial value all in one fell swoop. This is called *initializing* a variable. You can have more than one operation on the righthand side of the equal sign. In the following code, an empty variable named `celsius` is created, then a variable named `fahrenheit` with value `212`, and then the Fahrenheit temperature is converted to Celsius:

```
var celsius;
var fahrenheit = 212;
celsius = ((fahrenheit - 32) / 9 ) * 5;
```

Arrays

An *array* is an ordered collection of data, indexed by number. We compared a simple variable to a mailbox (send mail to the "Smith Residence"). An array is like a set of numbered apartment mailboxes (send mail to the "A-List Apartments, #12"). The only difference is that array index numbers start at 0 rather than 1.[2] Here's a declaration of an array of radius sizes for circles. This form initializes the array. The second statement sets the value of the last element of the array to 9. You access one of the "slots" of an array by putting its index number in square brackets. Figure C-3 shows the results after the code has finished:

```
var radiusSizes = [8.5, 6.4, 12.2, 7];
radiusSizes[3] = 9;
```

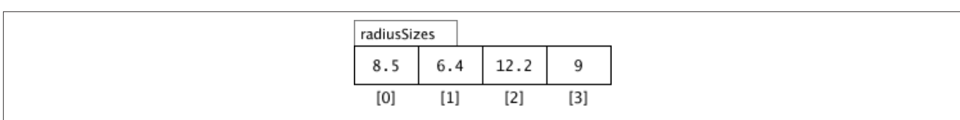


Figure C-3. Depiction of an array

Comments

Comments provide a way to document your programs so that other people can figure out what you did. There are two kinds of comments in ECMAScript. If you place two forward slashes in a row (`//`), they and everything to the end of that line are considered to be a comment. If you want a multiline comment, start with `/*` and end with `*/` as shown here:

```
var interest;   // this is accumulated on a daily basis
var rate;       // expressed as a decimal; 75% is a rate of 0.75

/* Figure out the payment amount given a principal
   of $10,000 and 180 monthly payments. */
```

Conditional Statements

Ordinarily, your program statements are carried out in the order in which they appear. Sometimes you may want to do different calculations depending upon some condition. You use the `if` statement to do this. Here's a calculation for wages that depends upon the number of hours worked. We presume that all the `var` statements have been set up appropriately:

2. This is not done to be contrary; it's because programs often use a mathematical formula to select the relevant item. These formulas are invariably easier when you start counting at zero.

```
if (hours <= 40)     ❶
{
  pay = hours * rate;     ❷
}
else     ❸
{
  pay = 40 * rate + (hours - 40) * rate * 1.5;     ❹
}
```

❶ The expression in the parentheses is called the *condition*. It always asks a yes-or-no question. In this case, the question is "is the value of the `hours` variable less than or equal to 40?" Other comparison operations are less than (<), greater than (>) greater than or equal (>=), equal (==), and not equal (!=).[3] Note that asking if two things are equal to each other requires two equal signs, not one!

❷ If the answer to the question is yes, then the program will do everything between the opening and closing braces `{` and `}`…

❸ …otherwise (`else`)…

❹ …do everything between the other set of curly braces. The curly braces are used to signify that one or more statements should be grouped together, much in the way that XML opening and closing tags tell where content begins and ends.

Repeated Actions

Sometimes you want to repeat an action a specific number of times ("fill 10 2-liter containers from a large water tank"). You use a `for` loop to do the first sort of task. It's called a *loop* because, if you were to draw arrows representing the path the computer takes through your program, they would form a loop as the program repeated the actions. Here's the container-filling scenario translated into ECMAScript, with variables for the water tank and the containers presumed to be defined. The loop body, that is, the actions you want repeated, are enclosed in curly braces:

```
var i;              // a counter variable
for (i = 0;         // start counting at zero
     i < 10;        // up to (but not including) 10
     i++)           // add one to the count at every repetition
{
  container[i] = 2;               // fill container number "i"
  waterTank = waterTank - 2;  // take 2 liters out of the tank
}
```

3. There are also the `===` and `!==` operators; when you use them, Boolean values and strings containing numbers won't automatically be converted into numbers when compared against a number.

Other times, you want to repeat an action as long as some condition is true ("keep filling 2-liter containers from a large water tank as long as there is any water left"). For this, you use a `while` loop:

```
i = 0;                          // start with container number zero
while ( waterTank > 0 )         // while there is water left
{
  container[i] = 2;                // fill container number "i"
  waterTank = waterTank - 2;  // take 2 liters out of the tank
  i = i + 1;                  // move on to the next container
}
```

Functions

You can accomplish some surprisingly sophisticated tasks with this small number of programming concepts. You collect sets of ECMAScript statements designed to perform a specific task into functions. Think of a function as a recipe card that gives a list of ingredients and instructions which, when followed, create a specific dish. A function starts with the keyword `function` followed by the function name. The name should be indicative of the task that it does, and it follows the same rules that variable names do. Following the function name, in parentheses, are the parameters of the function. A parameter is extra information that the function needs when it does its task. Consider this imaginary recipe:

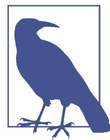
> Korean Kimch'i Surprise
>
> Take 100 grams of kimch'i per serving, 25 grams of ko-ju-jang red pepper paste per serving, and 50 grams of mushrooms per serving. Mix well. Serve.

Before you can make the recipe, you have to supply some extra information—the number of servings you intend to make. Our script might look like this:

```
function makeKimchiSurprise(numberOfServings)
{
  var kimchi = 100 * numberOfServings;
  var kojujang = 25 * numberOfServings;
  var mushrooms = 50 * numberOfServings;
  var surprise = kimchi + kojujang + mushrooms;
}
```

This is only the definition of the function. It does absolutely *nothing* until it is invoked, or called on. (You may have hundreds of recipe cards in a file box at home. They just sit there, inactive, until someone asks you to pull one card out and perform the cooking tasks.) You will often call a function as the result of an event. In the following example, a click on the blue rectangle will call the function. The number `5` in the parentheses will fill in the "extra information" required by the `numberOfServings` parameter:

```
<rect x="10" y="10" width="100" height="30" style="fill: blue;"
  onclick="makeKimchiSurprise( 5 )" />
```



Even if the function doesn't need any parameters, you still have to put the parentheses after its name in order to call it. Without the parentheses, `area` is just another object to be utilized. Unlike other languages, ECMAScript doesn't let you declare both a variable `area` and a function `area()` in the same namespace.

Functions can also call other functions. For example, a function that calculates compound interest might need to call upon another function that determines whether a year is a leap year or not. A parameter lets the interest function tell the leap year function what year it's interested in. The `return` statement will let the leap year function communicate its result back to the caller. This allows you to modularize a program into generally useful building blocks. In cooking terms, the `makeHollandaiseSauce()` function can be called from the `makeEggsBenedict()` function as well as from `makeChickenFlorentine()`.

Objects, Properties, and Methods

Take a power supply with its on-off switch, a plastic dial, a lever with a spring, and a metal chassis with rectangular slots and coils of wire in it. Put all these parts together, and you get a toaster.

Each of these parts is an object. Some of them have characteristics that are of interest: the power supply has a voltage of 110 or 220 volts, the chassis has a color and a number of slots, and the dial has a minimum and maximum setting. (The lever has no interesting characteristics.)

You do actions with each of these objects: you push the lever down or pop it up, you insert bread into the slots, you turn the power supply on or off, and you turn the dial to the desired setting.

Let's take this toaster into the world of ECMAScript. Now the mailboxes can hold many slips of paper, each representing different data properties of the object. Simple data types (like numbers or strings) are written down on the slip of paper directly. More complex data types require their own "mailboxes" to hold all the information, so the "slip of paper" contains information on where to find the full object (a *pointer* to the data); as a consequence, multiple variables can reference the same object. A mailbox can also hold a set of instructions (functions, like our recipe cards) so that they can perform actions. When a variable is inside another mailbox, we call the inner variable a *property*. When a function is inside a mailbox, we call it a *method*. The diagram for the toaster looks like Figure C-4.

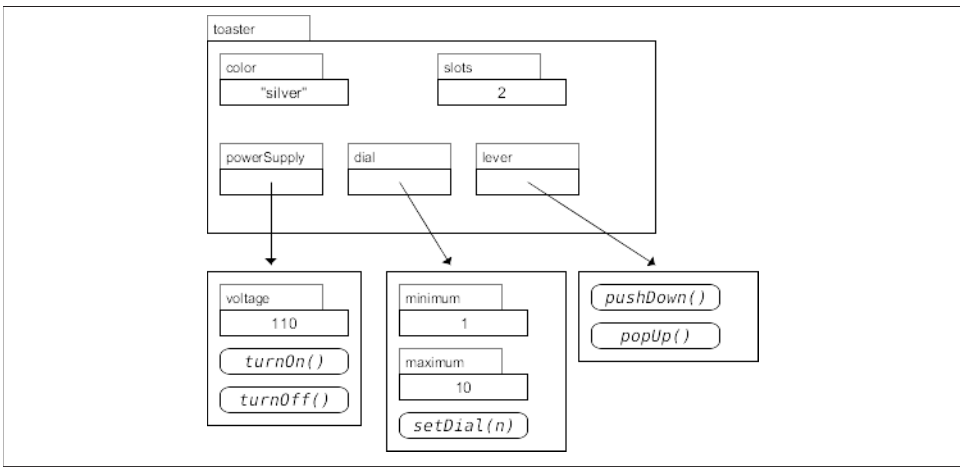


Figure C-4. The object diagram for a toaster

This is a very flexible way of modeling a toaster, but it's introduced a problem. To set the toaster's color or voltage or to pop out the bread, you can't just say things like this:

```
color = "gold";
voltage = 220;
popUp();
```

The `color` property is nested inside the `toaster` variable, the `voltage` really belongs to the `powerSupply` inside the `toaster`, and it's the `toaster`'s `lever` that does the `popUp` function. For these reasons, you must say:

```
toaster.color = "gold";
toaster.powerSupply.voltage = 220;
toaster.lever.popUp();
```

These are easy to figure out if you read them from right to left and say "of" whenever you see a period: "Put gold into the color of the toaster." "Put 220 into the voltage of the power supply of the toaster." "Call the pop up method of the lever of the toaster."[4] Think of this as the grown-up version of nested objects and methods that you learned as a child: "This is the dog that chased the cat that killed the rat that ate the malt that lay in the house that Jack built."

By using objects to model the behavior of a toaster, you've built a "Toaster Object Model." Similarly, there is a Document Object Model (DOM) that lets ECMAScript access a document's properties and invoke its methods. Almost all of your access to an SVG

4. If you insist upon reading left to right, adapt the suggestion for reading path names made by Elizabeth Castro in her book, *Visual Quickstart Guide to HTML for the World Wide Web*: read the period as "contains." Then `toaster.powerSupply.voltage = 220;` is read as "the toaster contains a power supply, which contains a voltage. Set that voltage to 220."

document will be through methods that begin with the word `set` or `get`. To set the radius of a `<circle>` element with an `id` of `wheel`, you might write `svgDocument.getElementById("wheel").setAttribute("r", 3)`. In some cases, you will use properties. For example, if you receive a mouse click event and want to find its *x*-coordinate, you would write `evt.clientX`.

The SVG Document Object Model is actually a superset of the XML Document Object Model; once you learn to manipulate the structure of an SVG document, you can immediately apply that knowledge to other XML documents, so the time you spend in learning the DOM will be amply repaid.

What, Not How

We've given you an overview of the *what* of programming, which can serve as a base for reading programs that other people have written and making sense of them (the programs, not the people). *How* you define a task and lay out the programming steps to solve it is another problem altogether, and far beyond the scope of this book. If you enjoy solving crossword puzzles or brain teasers, or just solving problems in general, you may well enjoy writing programs. If you would like to go in depth with JavaScript, we recommend *JavaScript: The Definitive Guide, 6th Edition* by David Flanagan (O'Reilly).

APPENDIX D

Matrix Algebra

Matrix algebra is a branch of mathematics that defines operations on *matrices*, which are series of numbers arranged in rows and columns. In addition to its many uses in science and engineering, matrix algebra makes the computations for graphic operations very efficient. The purpose of this appendix is to introduce you to the fundamental concepts of matrix algebra that SVG uses "behind the scenes."

Matrix Terminology

We describe a matrix by its number of rows and columns. Figure D-1 displays a matrix that arranges a series of daily temperatures over a two-week period into two rows of seven columns each. This matrix is called a *2-by-7 matrix*. Matrices are enclosed in square brackets when written.

$$\begin{bmatrix} 22.3 & 26 & 27.2 & 24.8 & 28 & 25 & 28.2 \\ 30.3 & 30.4 & 28 & 26.4 & 29.9 & 27.2 & 26 \end{bmatrix}$$

Figure D-1. 2-by-7 matrix of daily temperatures

Here are some other terms that you may encounter when dealing with matrix operations: a *square matrix* is a matrix with the same number of rows as columns. A *vector* is a matrix with only one row, and a *column vector* is a matrix with only one column. The individual numbers in a matrix are called *entries*, and the technical term for a plain number is a *scalar*. Now you can bring these sure-fire conversation stoppers to the next party you attend.

Applying the concept of a matrix to SVG, you might express a set of *x*- and *y*-coordinates as a 2-by-1 matrix. This isn't the way we'll ultimately end up representing coordinates,

but it's a good place to start, as it's easy to understand. In this representation, the point (3,5) is expressed as shown in Figure D-2.

$$\begin{bmatrix} 3 \\ 5 \end{bmatrix}$$

Figure D-2. Coordinates expressed as a matrix

Matrix Addition

The easiest matrix operation is addition. To add two matrices, you add their corresponding elements. This, of course, requires that your matrices have exactly the same number of rows and columns. Figure D-3 shows the addition of two matrices, each 3 by 2.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} + \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 1+7 & 2+8 \\ 3+9 & 4+10 \\ 5+11 & 6+12 \end{bmatrix} = \begin{bmatrix} 8 & 10 \\ 12 & 14 \\ 16 & 18 \end{bmatrix}$$

Figure D-3. Addition of two 3-by-2 matrices

You can see that the `translate` transformation in SVG could be accomplished easily by matrix addition. For example, the matrix addition in Figure D-4 would implement `transform="translate(7, 2)"` for any point (*x*,*y*).

$$\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 7 \\ 2 \end{bmatrix} = \begin{bmatrix} x+7 \\ y+2 \end{bmatrix}$$

Figure D-4. Simple method to translate coordinates

The order in which you add matrices doesn't matter. Mathematicians say that matrix addition is *commutative* (A + B = B + A). It is also *associative*; given three matrices A, B, and C: (A + B) + C is the same as A + (B + C). There is such a thing as matrix subtraction; just subtract the corresponding elements of the two matrices. Just as with regular subtraction, matrix subtraction is not commutative.

Matrix Multiplication

You may be thinking that matrix multiplication works in a similar manner, and that's how you can do a `scale()` transformation. Unfortunately, the easy way doesn't work this time. Matrix multiplication is significantly more complicated than matrix addition.

In the first examples that follow, this complexity appears to be needless. Later in this appendix, you'll see that the usefulness of matrix multiplication far outweighs its difficulty.

In order to multiply two matrices, the number of *columns* of the first matrix must equal the number of *rows* of the second matrix. Such matrices are called *compatible*. This means you can multiply a 3-by-5 matrix times a 5-by-4 matrix, but not a 3-by-5 matrix times a 3-by-2 matrix. The matrices in Figure D-5 are compatible and can be multiplied.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \bullet \begin{bmatrix} 7 & 10 \\ 8 & 11 \\ 9 & 12 \end{bmatrix}$$

Figure D-5. Two matrices to be multiplied

The resulting matrix will have the same number of rows as the first matrix, and the same number of columns as the second matrix. Thus, the example's 2-by-3 matrix times the 3-by-2 matrix will result in a 2-by-2 matrix.

The entry that will go in row one, column one of the result matrix is the dot product of the first row of the first matrix and the first column of the second matrix. *Dot product* is a fancy way of saying "add up the products of the corresponding entries in a row and column" as shown in Figure D-6.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \bullet \begin{bmatrix} 7 & 10 \\ 8 & 11 \\ 9 & 12 \end{bmatrix} = \begin{bmatrix} 1 \cdot 7 + 2 \cdot 8 + 3 \cdot 9 & - \\ - & - \end{bmatrix} = \begin{bmatrix} 50 & - \\ - & - \end{bmatrix}$$

Figure D-6. First entry in multiplied matrices

To find the quantity to place in row two, column one (the lower left) of the result matrix, take the dot product of row two in the first matrix and column one in the second matrix, as shown in Figure D-7.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \bullet \begin{bmatrix} 7 & 10 \\ 8 & 11 \\ 9 & 12 \end{bmatrix} = \begin{bmatrix} 1 \cdot 7 + 2 \cdot 8 + 3 \cdot 9 & - \\ 4 \cdot 7 + 5 \cdot 8 + 6 \cdot 9 & - \end{bmatrix} = \begin{bmatrix} 50 & - \\ 122 & - \end{bmatrix}$$

Figure D-7. Second entry in multiplied matrices

Calculating the remaining items produces the result shown in Figure D-8.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \bullet \begin{bmatrix} 7 & 10 \\ 8 & 11 \\ 9 & 12 \end{bmatrix} = \begin{bmatrix} 1 \cdot 7 + 2 \cdot 8 + 3 \cdot 9 & 1 \cdot 10 + 2 \cdot 11 + 3 \cdot 12 \\ 4 \cdot 7 + 5 \cdot 8 + 6 \cdot 9 & 4 \cdot 10 + 5 \cdot 11 + 6 \cdot 12 \end{bmatrix} = \begin{bmatrix} 50 & 68 \\ 122 & 167 \end{bmatrix}$$

Figure D-8. Completed multiplication of matrices

Given this information, you can now use matrix multiplication to express the calculations needed to scale a point (x,y) by a factor of 3 horizontally and a factor of 1.5 vertically. The transformation matrix will have to have two rows and two columns so it is compatible with the two-row by one-column coordinates, as shown in Figure D-9.

$$\begin{bmatrix} 3 & 0 \\ 0 & 1.5 \end{bmatrix} \bullet \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 3 \cdot x + 0 \cdot y \\ 0 \cdot x + 1.5 \cdot y \end{bmatrix} = \begin{bmatrix} 3x \\ 1.5y \end{bmatrix}$$

Figure D-9. Simple scaling by multiplying matrices

Unlike multiplication of single numbers, matrix multiplication is not commutative. If two matrices, A and B, aren't square matrices, then A·B won't have the same number of rows and columns as B·A (if they're even compatible in both directions). Even if A and B are both 3-by-3 square matrices, there's no guarantee that A times B will result in the same answer as B times A. In fact, they'll come out equal in only a very few special cases.

There is another limited form of multiplication: multiplying a matrix by a *scalar* (a plain number) will multiply every item in the matrix by the scalar, as shown in Figure D-10. We didn't mention this in conjunction with scaling, because this construct scales uniformly, and SVG scaling isn't always the same horizontally and vertically.

$$5 \bullet \begin{bmatrix} 3 & 2 \\ 5 & 6 \end{bmatrix} = \begin{bmatrix} 15 & 10 \\ 25 & 30 \end{bmatrix}$$

Figure D-10. Multiplying a scalar by a matrix

There is no such thing as matrix division *per se*. There is a construct called a *matrix inverse*, which is analogous to the reciprocal of a number; if you multiply matrix A times B times the inverse of B, the result is simply A. Another way of describing an invert matrix is that the inverse of B is the matrix that, when multiplied by B, creates an *identity matrix*. An identity matrix is a square matrix with values of 1 along the diagonal and values of 0 in every other position; multiplying by an identity matrix does not change the original matrix.

In SVG, inverse matrix calculations are used for converting between coordinate systems, where it is necessary to calculate the reverse of a transformation. Not all matrix transformations are invertable, however; if the original transformation resulted in the loss of information about one or more dimensions of the matrix, no subsequent multiplication can re-create it. While multiplying by the identity matrix is analagous to multiplying a number by 1, multiplying by a noninvertable matrix is analagous to multiplying a number by 0. And just like trying to divide by 0, trying to invert a noninvertable matrix will cause your processor to throw an error.

How SVG Uses Matrix Algebra for Transformations

The approach we've taken to translation and scaling works, but it's not ideal. For instance, if you want to translate a point and then scale it, you need to do one matrix addition and then a matrix multiplication. SVG uses a clever trick to represent coordinates and transformation matrices so you can do scaling and translation all with one operation. First, it adds a third value, which is always equal to 1, to the coordinate matrix. This means that the point (3,5) will be represented as shown in Figure D-11.

$$\begin{bmatrix} 3 \\ 5 \\ 1 \end{bmatrix}$$

Figure D-11. SVG representation of a coordinate

SVG uses a 3-by-3 matrix, again set up with extra zeros and ones, to specify the transformation. Figure D-12 shows a matrix that translates a point by a horizontal distance of `tx` and a vertical distance of `ty`.

$$\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} 3 \\ 5 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \cdot 3 + 0 \cdot 5 + tx \cdot 1 \\ 0 \cdot 3 + 1 \cdot 5 + ty \cdot 1 \\ 0 \cdot 3 + 0 \cdot 5 + 1 \cdot 1 \end{bmatrix} = \begin{bmatrix} 3 + tx \\ 5 + ty \\ 1 \end{bmatrix}$$

Figure D-12. SVG representation of translation

Figure D-13 shows a transformation matrix that will scale a point by a factor of `sx` in the horizontal direction and `sy` in the vertical direction.

$$\begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} 3 \\ 5 \\ 1 \end{bmatrix} = \begin{bmatrix} sx \cdot 3 + 0 \cdot 5 + 0 \cdot 1 \\ 0 \cdot 3 + sy \cdot 5 + 0 \cdot 1 \\ 0 \cdot 3 + 0 \cdot 5 + 1 \cdot 1 \end{bmatrix} = \begin{bmatrix} sx \cdot 3 \\ sy \cdot 5 \\ 1 \end{bmatrix}$$

Figure D-13. SVG representation of scaling

What this buys you is a consistent notation; all the transformations, including rotation and skewing, can be represented with 3-by-3 matrices. Furthermore, because everything is 3-by-3, you can construct a chain of transformations by multiplying those matrices together; they're guaranteed to be compatible. For example, to do a translation followed by a scaling, you multiply the matrices in that order (see Figure D-14).

$$\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} 3 \\ 5 \\ 1 \end{bmatrix}$$

Figure D-14. Translation followed by scaling

Again, it seems as if this is needlessly complicating matters. In order to transform the point (3,5), you now need two matrix multiplications. To transform another point would require two more multiplications. Given a `<path>` element with several hundred points, this could run into some serious computing time.

Here's where SVG does something clever: it multiplies the first two matrices together, and stores that result, as shown in Figure D-15.

$$\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} sx & 0 & tx \\ 0 & sy & ty \\ 0 & 0 & 1 \end{bmatrix}$$

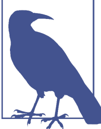
Figure D-15. Result of multiplying translation and scaling matrices

This "pre-multiplied" matrix now embodies both of the transformations. By multiplying this new matrix times a coordinate point's matrix, the translation and scaling occurs with a single matrix multiplication (see Figure D-16). Now conversion of a hundred points would require only 100 multiplications, not 200.

$$\begin{bmatrix} sx & 0 & tx \\ 0 & sy & ty \\ 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} 3 \\ 5 \\ 1 \end{bmatrix} = \begin{bmatrix} sx \cdot 3 + 0 \cdot 5 + tx \cdot 1 \\ 0 \cdot 3 + sy \cdot 5 + ty \cdot 1 \\ 0 \cdot 3 + 0 \cdot 5 + 1 \cdot 1 \end{bmatrix} = \begin{bmatrix} sx \cdot 3 + tx \\ sy \cdot 5 + ty \\ 1 \end{bmatrix}$$

Figure D-16. Result of premultiplying translation and scaling matrices

If you had to do a translation followed by a rotation followed by a scale, you'd create three 3-by-3 matrices; one to do the translation, one to do the rotation, and one to do the scaling. You'd multiply them all together (in that order), and the resulting matrix would embody all the calculations needed to do all three transformations.



As mentioned in "Matrix Multiplication" on page 320, matrix multiplication is not commutative. If you change the order of the transformation matrices, you get a different result. This is the mathematics behind the fact that the sequence of transformations makes a difference in the resulting graphic, as described in Chapter 6, in "Sequences of Transformations" on page 74.

This, then, is the power of matrix algebra; it lets you combine the information about as many transformations as you want into one single 3-by-3 matrix, thus dramatically reducing the amount of calculation necessary to transform points. The matrices in Figure D-17 are the ones used to specify a rotation by an angle `a`, a skew along the *x*-axis of `ax`, and a skew along the *y*-axis of `ay`.

$$\begin{bmatrix} \cos(a) & -\sin(a) & 0 \\ \sin(a) & \cos(a) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & \tan(ax) & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ \tan(ay) & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure D-17. Rotate, skew x, and skew y transformation matrices

You can use the `matrix(a, b, c, d, e, f)` transformation to specify six numbers that fill in the entries in the transformation matrix; the relationship of the numbers to the matrix is shown in Figure D-18.

$$\begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix}$$

Figure D-18. Generic transformation matrix

SVG also uses matrix algebra quite heavily in the calculations associated with filters, which are described in Chapter 11. There, a pixel's red, green, blue, and alpha (opaqueness) values are described as a matrix with five rows and one column. It also adds a fifth row so that a 5-by-5 transformation matrix can add a constant amount to any of the values as well as multiply them by any desired factor. The economies of scale we get by pre-multiplying coordinate transformation matrices work equally well when pre-multiplying pixel manipulation matrices.

The `<feColorMatrix>` filter, described in Chapter 11 in "The <feColorMatrix> Element" on page 160, lets you specify all 20 values. Thus, this markup:

```
<feColorMatrix type="matrix"
  values=
   "a0  a1  a2  a3  a4
    a5  a6  a7  a8  a9
    a10 a11 a12 a13 a14
    a15 a16 a17 a18 a19"/>
```

would be placed in the pixel transformation matrix, as shown in Figure D-19.

$$
\begin{bmatrix}
a_0 & a_1 & a_2 & a_3 & a_4 \\
a_5 & a_6 & a_7 & a_8 & a_9 \\
a_{10} & a_{11} & a_{12} & a_{13} & a_{14} \\
a_{15} & a_{16} & a_{17} & a_{18} & a_{19} \\
0 & 0 & 0 & 0 & 1
\end{bmatrix}
$$

Figure D-19. Color transformation matrix

APPENDIX E
Creating Fonts

The fonts built into the system that renders your SVG documents will take care of the vast majority of your needs. Sometimes, though, you will want to use a custom font. It is possible to create a font for use with SVG from scratch. In brief, you use a `<font>` element tag to describe the origin and default width of the font's glyphs. Inside the `<font>` is the `<font-face>` element, which has an immense number of attributes that describe the font's dimensions in excessive detail. Table E-1 summarizes some of the more useful attributes. You can see them all in detail in the SVG specification.

The SVG font specification was intended to allow designers to create accessible logos and graphics. Search engines and screen readers would understand the text as a sequence of characters, but the design could be completely customized and would look the same on every system. At the time of writing, this ideal has not yet been realized, because two major web browsers (Internet Explorer and Firefox) have not implemented SVG fonts. If you create a custom SVG font, there are web services that can convert it to other font formats for use with these browsers.

Table E-1. font-face attributes

`font-family`	A list of font family names
`font-style`	Values of `normal`, `italic`, or `oblique`.
`font-variant`	`normal` or `small-caps`.
`font-weight`	`normal`, `bold`, or a number from 100 to 900 in steps of 100.
`font-stretch`	Indicates the `condensed` or `expanded` nature of the face relative to others in its font family. Possible prefixes for `condensed` or `expanded` are `ultra-`, `extra-`, and `semi-`.
`font-size`	`all`, suitable for most scalable fonts, or if a font is designed for a restricted range of sizes, a list of lengths (such as `18pt`).
`unicode-range`	The range of Unicode characters covered by this font, in the form `U`*`start-end`*.

`font-family`	A list of font family names
`units-per-em`	Coordinate units for the *em square*. This establishes a coordinate system for the font. The following are all measured in these units.
`cap-height`	Height of uppercase glyphs.
`x-height`	Height of lowercase glyphs.
`accent-height`	Distance from the origin to the top of accent characters.
`ascent`	Maximum unaccented height of the font.
`descent`	Maximum unaccented depth of the font.
`widths`	A list of widths for the glyph corresponding to each character.
`bbox`	The maximal bounding box for the font; a box in which the largest character will fit.
`underline-position`	Ideal position of an underline.
`underline-thickness`	Ideal thickness of an underline.

Following the `<font-face>` are `<glyph>` elements, which contain path descriptions for each of the glyphs you wish to have in your font.

While it is possible to create fonts from scratch, it's a lot of work, and often a duplication of effort, as the glyphs you need may be in an already-existing font. If you already have a TrueType font with the desired glyphs, you are in luck. The quadratic Bézier curves used in TrueType can be easily converted to SVG glyphs. Just be sure to include the standard TrueType font as a fallback option (as a web font or as a reference to a local font name) for web browsers that don't support SVG fonts.

The ttf2svg Utility

The Apache Batik project has created a utility that will convert your TrueType fonts to SVG. The following summary is adapted from the Batik project's documentation (copyright 2013 The Apache Software Foundation. All rights reserved).

If you are using the Batik binary distribution, type the following at the command line:

```
java -jar batik-ttf2svg.jar [options]
```

If you are using the Batik developer distribution, type the following at the command line:

```
build ttf2svg [options]
```

In both cases, the options are the same (these options will all go on the same line when typed at the command line; they are placed on separate lines here for ease of reading):

```
ttf-path
[-l range-being]
[-h range-end]
[-ascii]
```

```
    [id id]
    [-o output-path]
    [-testcard]
```

The options have the following meanings:

ttf-path
: Specifies the TrueType font file that contains the characters to be converted.

`-l` *range-begin*
`-h` *range-end*
: The low and high value of the range of characters to be converted to SVG (ASCII or Unicode values).

`-ascii`
: Forces usage of the ASCII character map.

`-id` *id*
: Specifies the value for the `id` attribute of the generated `<font>` element.

`-o` *output-path*
: Specifies the path for the generated SVG font file. If not specified, output goes to the Java console.

`-testcard`
: Specifies that a set of SVG `<text>` elements should be appended to the SVG font file to visualize and test the characters in the SVG font. This provides an easy way to validate the generated SVG font file visually.

For example, to convert characters 48 to 57, that is, the characters *0* to *9*, in *myFont.ttf* into their SVG equivalent in the *mySVGFont.svg* file, appending a test card so that the font can be visualized easily, you would use this command:

```
java -jar batik-ttf2svg.jar /usr/home/myFont.ttf -l 48 -h 57
   -id MySVGFont -o mySVGFont.svg -testcard
```

Make sure you have the right to embed a font before you embed it in an SVG file. TrueType font files contain a flag that defines the "embeddability" of a font, and there are tools for checking that flag.

Converting Arcs to Different Formats

Converting from Center and Angles to SVG

The following JavaScript code converts an arc specified in center-and-angles format to a form suitable for placing into an SVG `<path>`:

```
/*
  Convert an elliptical arc based around a central point
  to an elliptical arc parameterized for SVG.

  Parameters are:
    center x coordinate
    center y coordinate
    x-radius of ellipse
    y-radius of ellipse
    beginning angle of arc in degrees
    arc extent in degrees
    x-axis rotation angle in degrees

  Return value is an array containing:
    x-coordinate of beginning of arc
    y-coordinate of beginning of arc
    x-radius of ellipse
    y-radius of ellipse
    x-axis rotation angle in degrees
    large-arc-flag as defined in SVG specification
    sweep-flag  as defined in SVG specification
    x-coordinate of endpoint of arc
    y-coordinate of endpoint of arc
*/

function centeredToSVG(cx, cy, rx, ry, theta, delta, phi)
{
  var endTheta, phiRad;
  var x0, y0, x1, y1, largeArc, sweep;
```

```
  /*
    Convert angles to radians. I need a separate variable for phi as
    radians, because I must preserve phi in degrees for the
    return value.
  */
  theta = theta * Math.PI / 180.0;
  endTheta = (theta + delta) * Math.PI / 180.0;
  phiRad = phi * Math.PI / 180.0;

  /*
    Figure out the coordinates of the beginning and ending points
  */
  x0 = cx + Math.cos(phiRad) * rx * Math.cos(theta) +
    Math.sin(-phiRad) * ry * Math.sin(theta);

  y0 = cy + Math.sin(phiRad) * rx * Math.cos(theta) +
    Math.cos(phiRad) * ry * Math.sin(theta);

  x1 = cx + Math.cos(phiRad) * rx * Math.cos(endTheta) +
    Math.sin(-phiRad) * ry * Math.sin(endTheta);

  y2 = cy + Math.sin(phiRad) * rx * Math.cos(endTheta) +
    Math.cos(phiRad) * ry * Math.sin(endTheta);

  largeArc = (delta > 180) ? 1 : 0;
  sweep = (delta > 0) ? 1 : 0;

  return [x0, y0, rx, ry, phi, largeArc, sweep, x1, y1];
}
```

Converting from SVG to Center and Angles

The following code, adapted from the Apache Batik project, converts an SVG-style arc to a center-and-angles format:

```
/*
  Convert an elliptical arc specified as SVG path parameters
  to an arc based around a central point.

  Parameters are:
    x-coordinate of beginning of arc
    y-coordinate of beginning of arc
    x-radius of ellipse
    y-radius of ellipse
    x-axis rotation angle in degrees
    large-arc-flag as defined in SVG specification
    sweep-flag  as defined in SVG specification
    x-coordinate of endpoint of arc
    y-coordinate of endpoint of arc

  Return value is an array containing:
```

```
    center x coordinate
    center y coordinate
    x-radius of ellipse
    y-radius of ellipse
    beginning angle of arc in degrees
    arc extent in degrees
    x-axis rotation angle in degrees
*/

function convertArc(x0, y0, rx, ry, xAngle, largeArcFlag,
  sweepFlag, x, y)
{
  // Step 1: compute half the distance between the current
  // and final point.
  var dx2 = (x0 - x) / 2.0;
  var dy2 = (y0 - y) / 2.0;

  // convert angle from degrees to radians
  var xAngle = Math.PI * (xAngle % 360.0) / 180.0;
  var cosXAngle = Math.cos(xAngle);
  var sinXAngle = Math.sin(xAngle);

  // Compute x1, y1
  var x1 = (cosXAngle * dx2 + sinXAngle * dy2);
  var y1 = (-sinXAngle * dx2 + cosXAngle * dy2);

  // Ensure radii are large enough
  rx = Math.abs(rx);
  ry = Math.abs(ry);
  var rxSq = rx * rx;
  var rySq = ry * ry;
  var x1Sq = x1 * x1;
  var y1Sq = y1 * y1;

  var radiiCheck = x1Sq / rxSq + y1Sq / rySq
  if (radiiCheck > 1) {
    rx = Math.sqrt(radiiCheck) * rx;
    ry = Math.sqrt(radiiCheck) * ry;
    rxSq = rx * rx;
    rySq = ry * ry;
  }

  // Step 2: Compute (cx1, cy1)
  var sign = (largeArcFlag == sweepFlag) ? -1 : 1;
  var sq = ((rxSq * rySq) - (rxSq * y1Sq) - (rySq * x1Sq)) /
    ((rxSq * y1Sq) + (rySq * x1Sq));
  sq = (sq < 0) ? 0 : sq;
  var coef = (sign * Math.sqrt(sq));
  var cx1 = coef * ((rx * y1) / ry);
  var cy1 = coef * -((ry * x1) / rx);

  // Step 3 : Compute (cx, cy) from (cx1, cy1)
```

```
  var sx2 = (x0 + x) / 2.0;
  var sy2 = (y0 + y) / 2.0;
  var cx = sx2 + (cosXAngle * cx1 - sinXAngle * cy1);
  var cy = sy2 + (sinXAngle * cx1 + cosXAngle * cy1);

  // Step 4 : Compute the angleStart and the angleExtent
  var ux = (x1 - cx1) / rx;
  var uy = (y1 - cy1) / ry;
  var vx = (-x1 - cx1) / rx;
  var vy = (-y1 - cy1) / ry;
  var p, n;
  // Compute the angle start
  n = Math.sqrt((ux * ux) + (uy * uy));
  p = ux; // (1 * ux) + (0 * uy)
  sign = (uy < 0) ? -1.0 : 1.0;
  var angleStart = 180.0 * (sign * Math.acos(p / n)) / Math.PI;

  // Compute the angle extent
  n = Math.sqrt((ux * ux + uy * uy) * (vx * vx + vy * vy));
  p = ux * vx + uy * vy;
  sign = (ux * vy - uy * vx < 0) ? -1.0 : 1.0;
  var angleExtent = 180.0 * (sign * Math.acos(p / n)) / Math.PI;
  if(!sweepFlag && angleExtent > 0)
  {
    angleExtent -= 360.0;
  }
  else if (sweepFlag && angleExtent < 0)
  {
    angleExtent += 360.0;
  }
  angleExtent %= 360;
  angleStart %= 360;

  return( [cx, cy, rx, ry, angleStart, angleExtent, xAngle] );
}
```

Index

Symbols

@keyframes CSS rule, 205, 207

A

a element, 209
A or a command (elliptical arcs), 91
absolute versus relative path notation, 88
addEventListener function, 217
additive attribute, animation elements, 201
  combining effects controlling numerical and color attributes, 202
aliasing, 41
  (see also anti-aliasing)
alignment
  specifying for preserveAspectRatio, 33–36
  text, 129
alpha value
  changing with feColorMatrix element, 160
  determining for masks, 149
  feConvolveMatrix filter and, 183
  final, interaction with color and opacity in masks, 151
  for rgba/hsla color functions, 43
  luminanceToAlpha filter preset, 162
  SourceAlpha filter input, 157
animate element, 192
  (see also SMIL animation)
animateMotion element, 202
  animation along a complex path, 202
  animation along a linear path, 202
  rotate attribute, 203–204
  specifying key points and times for motion, 204
  using mpath, 204
animateTransform element, 200
animation, 191
  (see also CSS; JavaScript/ECMAScript; SMIL animation)
  basics, 192
  frame rate, 252
  via scripting, 251–256
animation events (SMIL), 216
animVal and baseVal properties, 233
anti-aliasing, 4, 41
Apache Software Foundation Batik project, xii
application, SVG as, 18
arcs, 90–93
  converting to different formats, 93, 331–334
    from center and angles to SVG, 331
    from SVG to center and angles, 332
  path syntax, 91
arrays, in JavaScript, 312
ascent, in typography, 126
aspect ratio, preserving, 32
  (see also preserveAspectRatio attribute)
assignment, 310

We'd like to hear your suggestions for improving our indexes. Send email to index@oreilly.com.

attributeName, animation elements, 192
attributes
accessing and changing using the DOM, 214, 218, 233
in XML markup, 287
presentation, 7, 60
(see also styles)
attributeType, animation elements, 192

B

background-image style, 17
background-position style, 18
background-size style, 18
BackgroundAlpha filter input, 179
BackgroundImage filter input, 179
baseline, 126
baseline-shift style, 132
baseVal and animVal properties, 233
Batik SVG viewer, xii
Bézier curves, 93–99
animation along a cubic Bézier curve path, 202
bidirectional text, 134
bitmaps, 1
temporary, filter operations on, 156
blanks, SVG handling of, 141
border-radius style (HTML), compared with rectangle corner radius attributes, 47
brightness, changing with feComponentTransfer, 164
browsers (see web browsers)
bump map, in lighting filters, 175

C

C or c command (cubic Bézier curves), 97
CAD (Computer Assisted Drafting) programs, 3
calcMode attribute, animation elements, 199
cap-height, 126
Cartesian coordinates, converting from, 76, 266
character encodings, 291
other than Unicode, 292
character references, 291
characters, defined, 125
circles, 6, 47
SVGCircleElement object and <circle> element, 233
class attribute, 301
classes (CSS), 300
(see also pseudoclasses)
click events, 216, 217
click handler in Snap.svg, 262
in user-triggered SMIL animations, 212–213
clientX and clientY event properties, 217
clip-path style, 145
clipPath element, 145
basic shapes, path elements, or text elements in, 146
clipPathUnits attribute, 147
clipping area, 145
clipping graphics, 145–149
closepath command, 86–88, 88
code point, in character encodings, 291
color, 41, 54
color style, 42
colors
animating, 197, 198
changing with feColorMatrix, 160–163
CSS3 color specification, 42
defining the color space, 169
fill color, 45, 303
gradients, 113–122
interaction with opacity and final alpha value in masks, 151
mask color values and transparency, 149
masking with opaque colors, 150
stroke color, 41, 303
comments
in ECMAScript, 312
in XML, 289
conditional statements, in scripting, 312
(see also switch element)
constants, 309
coordinate system, transforming, 9, 69–84
converting from Cartesian coordinates, 76
reference summary of transformations in SVG, 83
rotate transformation, 78
scale transformation, 71
scaling around a center point, 81
sequences of transformations, 74
skewX and skewY transformations, 81
translate transformation, 69
coordinates, 27–38
absolute or relative, moveto and lineto commands, 88
default user coordinates, using, 28

nested systems of, 36
preserving aspect ratio, 32
specifying for lines, 39
specifying user coordinates for a viewport, 27, 30
CSS, 299
animating SVG with, 205–208
animating movement, 207
animation properties, 206
controlling using pseudoclasses, 211
setting animation key frames, 207
attributeType for animation elements, 192
CSS3 color specification, 42
properties
for img element in a web page, 17
for svg element inline in (X)HTML, 22
for text, 127
SVG file as value of, 17
table of properties and values for SVG, 301
pseudoclasses, 210
transformations and SVG, 83, 200
CSS sprites, 18
cubic Bézier curves, 96
currentColor keyword, 42

D

d (data) attribute, path element, 85
dashed lines, 44
data objects in SVG DOM, table of, 236
declarative interactivity, 209
defer specifier (preserveAspectRatio, 68
defs element, 63
desc element, 6
descent, in typography, 126
diffuse lighting, 175
document events, 216
document object
getElementById function, 214
getElementsByTagName function, 214
document structure, 57–68
basic, in SVG, 6
grouping and referencing objects, 61–68
structure and presentation, 7, 57
using styles with SVG, 58–61
Document Type Definitions (DTD), 293
DOM (Document Object Model), 214
accessing SVG with, 215
DOM-based XML parsers, 296
events, 217
compared with CSS pseudoclasses, 212
in Snap.svg, 261
to control SMIL animation, 212–213
mutation events, 216
SVG DOM methods and properties, 233–263
animVal and baseVal properties, 233
constructing SVG with ECMAScript/JavaScript, 248
data objects and methods, table of, 236
determining value of element attributes, 233
SVG interface methods, 241
with JavaScript libraries, 256
dragging objects, 221, 262
drop shadow, creating, 156–159
DTD (Document Type Definitions), 293
dur (duration) attribute, animation elements, 199

E

ECMAScript, 213
(see also JavaScript/ECMAScript)
elements
creating new, with scripts, 229
reference tables
basic shapes, 53
DOM interfaces, 241
filters, 187
path commands, 99
XML, 287
ellipses, 48
elliptical (see arcs)
embed element, 19
embossing effect, using feConvolveMatrix, 183
enable-background attribute, 179
end attribute, animation elements, 195
entity references (XML), 290
escaping characters, 289
event-based interaction, 212
events, 216
categories of, 216
compared with CSS pseudoclasses, 212
event handling in Snap, 261
clicking objects, 262
dragging objects, 262
listening for and responding to, 217, 223
to control SMIL animation, 212–213

ex-height, 126
examples in this book, about, xii
Extensible Stylesheet Language Formatting Objects (see XSL-FO files)

F

feBlend filter, 172
feColorMatrix filter, 160–163
feComponentTransfer filter, 164–169
  gamma adjustment with, 165
feComposite filter, 169, 178
feConvolveMatrix filter, 182
  embossing effect, 183
feDiffuseLighting filter, 175
feDisplacementMap filter, 184
feFlood and feTile filters, 173
feFuncR, feFuncG, feFuncB, and feFuncA elements, 164
feGaussianBlur filter, 157
feImage filter, 163, 180
feMerge filter, 169
feMergeNode element, 158, 169
feMorphology filter, 181
feSpecularLighting filter, 177
feSpotLight filter, 179
feTurbulence filter, 186
files, external
  for patterns and gradients, 123
  for stylesheets, 59
  for SVG fonts, 137
  in SVGs inserted in web pages, 15, 18
  with use element, 65
fill attribute, animation elements, 196
fill characteristics, table of, 55
fill style
  for text, 126
  specifying color, 7, 45
  using gradients, 113–122
  using patterns, 107–113
  with polyline element, 51
fill-opacity style, 45, 55
fill-rule style
  for paths, 100
  for polygons with intersecting lines, 49
filters, 155–189
  accessing the background, 179
  creating a drop shadow, 156
    establishing filter's bounds, 156
    storing, chaining, and merging filter results, 158
    using feGaussianBlur, 157
  creating a glowing shadow, 159
  how they work, 155
  lighting effects, 174
    diffuse lighting, 175
    specular lighting, 177
  reference summary, 187
  resolution of, 156
filterUnits attribute, 157
Flash, 3
focal point (radial gradients), 119
:focus pseudoclass, 210
font-face element, 137, 327
  attributes for (table), 327
font-face-name element, 138
font-face-src element, 138
font-family style, 127
  matching value in external font file, 137
font-size style, 127
font-style style, 128
font-weight style, 128
fonts
  creating, 327–329
  defined, 126
  font family and font size for text label, 12
  using a custom font, 137
for loop, 313
foreign objects, instream-foreign-object element in XSL-FO, 25
foreignObject element, 20
frame rate, 252
functions, 314

G

g element (group), 8, 61
generating SVG, 265–282
  converting custom data to SVG, 266–270
  using XSLT to convert XML data to SVG, 270–282
getSVGDocument function, 229
glyph-orientation-vertical style, 133
glyphs, 125
gradients, 113–122
  color values between gradient stops, 169
  linear, 113
    establishing a transition line, 115
    spreadMethod attribute, 116

stop element, 113
radial, 118
spreadMethod attribute, 120
transition limits for, 118
using as a mask, 153
reference summary, 121
transforming, 122
using as a displacement map, 184
gradientTransform attribute, 122
gradientUnits attribute, 116, 120, 122
graphics programs, drawing Bézier curves, 94
graphics systems, raster versus vector, 1–3
grouping, 8, 61
transforming the group, 73
using symbol element, 66
within defs element, 63
within use element, 63

H

H or h command (horizontal lines), 88
horizontal lineto command, 88
:hover pseudoclass, 210
hsl and hsla color functions, 43
HTML (see HTML)
handling of whitespace, 142
inline SVG in, 22
inline SVG in HTML5, 22
interacting with embedded SVG, 225–229
SVG foreign object containing XHTML text, 20

I

icons within a single image file, 18
image element, 67
images
feImage filter, 163
masking, 152
img element, including SVG in, 16
inline styles, 58, 299
modifying, 214
stylesheets versus, 60
inline SVG, in XHTML or HTML5, 22–25
instream-foreign-object element, 25
interactivity, adding, 209–231
controlling CSS animations, 211
scripting SVG, 213–231
user-triggered SMIL animation, 212
using links in SVG, 209
interfaces for SVG elements, 241
internationalization, text and, 134–138
Internet media types, 18

J

JavaScript/ECMAScript, 213–231
animation with, 251–256
animVal property, 233
requestAnimationFrame function, 252
using Snap.svg, 259
attributes, accessing and changing, 214, 218
creating new elements, 229
dragging objects, 221
events
listening for and responding to, 217
overview, 216
interacting with an HTML page, 225–229
getSVGDocument function, 229
programming concepts, 309–317
styles, accessing and modifying, 214
using code libraries, 256–261

K

key events, 217
key frames, setting for animation, 207
keyPoints attribute, animateMotion element, 205
keyTimes attribute
animateMotion element, 204
timing of multistage animations, 199

L

L or l command (lineto) (see lineto command)
languages
switching, 135
Unicode and bidirectional text, 134
letter-spacing style, 128, 133
lighting effects, 174
diffuse lighting, 175
specular lighting, 177
line caps and joins, 52
line element, 8, 39
stroke characteristics, 40–45
linear gradients, 113
establishing a transition line, 115
using spreadMethod attribute, 116
specifying stop-opacity, 114

stop element in, 113
linear RGB color space, 169
lineto command, 85
horizontal and vertical lineto commands, 88
multiple pairs of coordinates after, 89
relative coordinates, 88
links in SVG, 209
highlighting with SVG pseudoclasses, 210, 211
loops, 313

M

M or m command (moveto), 88
marker element, 100–105
marker-end style, 102
marker-mid style, 102
marker-start style, 101
orient attribute, 103
setting viewBox and preserveAspectRatio to control display, 104
markerUnits attribute, marker element, 104
mask element, 149
mask style, 150
maskContentUnits attribute, 149
masking graphics, 149–154
maskUnits attribute, 149
matrix algebra, 319–326
matrix multiplication, 320
matrix terminology, 319
use by SVG for transformations, 323
measurement units
explicit use of, 29
specifying for the viewport, 27
meet specifier (preserveAspectRatio), 34
metadata, 287
methods, of JavaScript/ECMAScript objects, 315
MIME types, 18
mouse events, 216
adding mouse movement listeners, 217
moveto command, 85
multiple pairs of coordinates after, 89
relative coordinates, 88
mpath element, using in animateMotion element, 204
mutation events, DOM, 216

N

namespaces, 10, 295
nested markers, 105
nested patterns, 112
newlines, SVG handling of, 141
nodes, 214
none specifier (preserveAspectRatio), 36
notational shortcuts (paths), 89
number function, XSLT, 278

O

object element, HTML, 18, 225
embed element versus, 19
objectBoundingBox units
stroke and, 123
with clipPathUnits, 147
with filterUnits, 157
with gradientUnits, 116, 122
with maskContentUnits, 149
with maskUnits, 149
with patternContentUnits, 110
with patternUnits, 108
with primitiveUnits, 157
objects, in scripting, 315
object interfaces for SVG elements, 233, 242
SVG data objects (table), 236
offset attribute, stop element, 113
opacity
fill-opacity style, 45
interaction between color, opacity, and final alpha value in masks, 151
stop-opacity style, gradient stops, 114
stroke-opacity style, 43
operators, in JavaScript/ECMAScript, 311
orient attribute, marker element, 103
origin (coordinate system), 28
rotation around, 79
overflow style, 148

P

paced animation, 204
parsers (XML), 296
path element, 85
paths, 11, 85–105
animating, 198
Bézier curves, 93–99
clipping to, 145

closepath command, 86
converting from other arc formats, 93
elliptical arcs, 90
filling, 100
for a pattern tile, 107
key points for animateMotion element, 205
lineto command, 85
marker element, 100–105
moveto command, 85
reference summary of commands, 99
relative moveto and lineto, 88
shortcuts, 88
horizontal and vertical lineto commands, 88
notational shortcuts, 89
text on, 138
patternContentUnits attribute, 110
patterns, 107–113
nested, 112
tiles, 107
transforming, 122
patternTransform attribute, 122
patternUnits attribute, 108
pluginspage attribute, embed element, 19
points attribute, polygon and polyline elements, 48, 51, 51
poly-Bézier curve, 95
polygon element, 48–51
animating, 198
ArcInfo Geographic Information in, 266
having intersecting lines, filling, 49
polyline element, 10, 51, 89
PostScript, 3
presentation
as attributes, 60
versus structure, 7, 57
preserveAspectRatio attribute
default value, 34
defer specifier, 68
meet specifier, 34
none specifier, 36
slice specifier, 35
specifying alignment, 33
using to scale a pattern, 111
with marker element, 104
preserveAspectRation attribute
defer specifier, 68
primitives, filter, 156
primitiveUnits attribute, filter element, 157
programming concepts, 309–317
arrays, 312
assignment and operators, 310
comments, 312
conditional statements, 312
constants, 309
functions, 314
objects, properties, and methods, 315
repeated actions, 313
variables, 310
properties
CSS property table for SVG, 302
of JavaScript/ECMAScript objects, 316
of SVG element objects, 233
pseudoclasses, 210–212

Q

Q or q command (quadratic Bézier curves), 95
quadratic Bézier curves, 94

R

radialGradient element, 118
establishing transition limits for, 118
spreadMethod attribute, 120
using as a mask, 153
raster graphics, 1
and filter effects, 155
scalability, lack of, 4
uses of, 2
rectangles, 45–47
default attributes, 46
rounded, 46
SVGRectElement object, 233
using explicit units, 30
referencing external files (see files, exteral)
relative coordinates, in path commands, 88
repeated actions, in JavaScript/ECMAScript, 313
repeated animations, 196
alternating between two values, 199
requestAnimationFrame function, 252
requiredExtensions attribute, 21
requiredFeatures attribute, 21
reusing graphics with use element, 63
using defs element, 63
using symbol element, 66
rgb color function, 41
rgba color function, 43

rotate transformation, 78

S

S or s command (smooth cubic Bézier curve), 98
scale transformations, 9, 71
  scale followed by translate, 75
  scaling around a center point, 81
  translate followed by scale, 75
  when converting from Cartesian coordinates, 78
scripts (see JavaScript/ECMAScript)
set element, 200
setTimeout function, 253
shape-rendering style, 41
shapes
  basic, 6, 39–55
    circles, 7
    circles and ellipses, 47
    line caps and joins, 52
    lines, 8, 39–45
    paths, 11
    polygon element, 48–51
    polyline element, 10, 51
    rectangles, 45–47
    summary table, 53–55
  in clipPath element, 146
  transformation applied to, 74
skewX and skewY transformations, 81
slice specifier (preserveAspectRatio), 35
SMIL animation, 192–205
  animate element, 192
  animateMotion element, 202–204
  animateTransform element, 200
  animation events, 216
  beginning and ending of, 193
  complex attributes, 197
  key points and times for motion, 204
  key times and calculation, 199
  multiple animations on an object, 193
  multiple values, 198
  repeated action, 196
  set element, 200
  synchronizing, 194
  time measurement, 194
  user-triggered, 212–213
SMIL3 specification, 192
smooth cubic curve command, 98
smooth quadratic curve command, 95
Snap.svg library (JavaScript), 256
  animate function and easing options, 259
  drawing graphics, 257
  event handling in, 261
    clicking objects, 262
    dragging objects, 262
  including the library script, 256
  Paper object, 256, 258
specular lighting, 177
spotlight filter, 179
spreadMethod attribute, 116, 122
  for linear gradients, 116
  for radial gradients, 120
sprites, CSS image, 18
src attribute, img element, 16
standard RGB (sRGB) color space, 169
startOffset attribute, textPath element, 140
state-based interaction, 212
stop element, 113
stop-color attribute, 113
stop-opacity style, 114
stroke characteristics, table of, 54
stroke style
  for rectangles, 45
  for text, 126
  path closure and, 87
  position relative to shape coordinates, 46
  scale transformations and, 72
    nonuniform scaling, 73
  specifying color, 7, 41
  using gradients and patterns, 123
stroke-dasharray style, 44
stroke-linecap style, 52
stroke-linejoin style, 52
stroke-miterlimit style, 53
stroke-opacity style, 43
stroke-width style, 40
  scaling transformations and, 81
structure (see document structure)
style attribute, 7, 299
styles, 58–61, 299
  font family and font size for text, 12
  inline, 58, 299
  presentation attributes versus, 7, 60
  table of style properties for SVG, 301
  text, changing with tspan element, 130
stylesheets, 299–308
  classes as style selectors, 300
  external, 59

inline styles versus, 60
internal, 58, 300
XSLT stylesheet, 273
subscripts and superscripts
using baseline-shift with tspan element, 132
using dy attribute of tspan, 130
SVG
defined, 1
role of, 5
scalability of vector graphics, 3
svg element, 6
viewBox attribute, 30
width and height attributes, 27
xmlns:xlink attribute, 10
SVGAngle object, 237
SVGAnimatedLength object, 236, 240
SVGCircleElement object, 233
SVGEllipseElement object, 235
SVGLength object, 237
SVGMatrix object, 238
SVGPoint object, 238
SVGRect object, 237
SVGRectElement object, 233
SVGTransform object, 239
SVGTransformList object, 239
switch element, 20, 135
symbol element, 66
synchronization of animations, 194
with repetition, 197
Synchronized Multimedia Integration Language (see SMIL animation)
systemLanguage attribute, 21, 135

T

T or t command (smooth quadratic curves), 95
tabs, SVG handling of, 141
target property, events, 217
text, 125–143
adding to a graphic, 12, 142–143
alignment, 129
CSS styles and values for, 127
drop shadow applied to, 159
glowing shadow for, 159
in vector versus raster graphics, 2
internationalization, 134
custom font, 137
switching languages with switch element, 135
Unicode and bidirectionality, 134
lengthAdjust attribute, 132
modifying text content of a node, 214
on a path, 138
overview, 126
terminology, 125
text elements in clipPath element, 146
textLength attribute, 132
tspan element, 129
(see also tspan element)
vertical, 133
whitespace and, 141
text element, 12
text-anchor style, 129, 140
text-decoration style, 128
textPath element, 138
referencing a path element in, 138
startOffset attribute, 140
this keyword, 218
tiles, 107
(see also patterns)
feTile filter, 173
spacing in patterns, 108
time measurement for animation, 194, 212
timing of animations, 199
title element, 6
transform style (see transformations, CSS)
transformations, 9, 69
(see also coordinate system, transforming)
animateTransform element, 200
converting from Cartesian coordinates, 76
CSS, 83, 207
matrix used for, 238
of patterns and gradients, 122
rotate transformation, 78
scale transformation, 71
sequences of, 74
skewX and skewY transformations, 81
summary table, 83
use of matrix algebra by SVG for, 323
transition limits, in gradients
and spreadMethod attribute, 116
for radial gradients, 118
transition line for linear gradients, 115
translate transformation, 9, 69
in Cartesian coordinates conversion, 78
scale followed by translate, 75
translate followed by scale, 75
using CSS, 208
transparency (see alpha value; opacity)

transparent specifier (CSS3), 43
tspan element, 129
  rotate attribute, 131
  using absolute positioning with, 130
  using baseline-shift, 132
  using multiple values for dx, dy, and rotate attributes, 131
  vertical positioning with dy attribute, 130
ttf2svg utility, 328
type attribute, object element, 18

U

Unicode, 134
  code points, 291
  encoding schemes, 291
use element, 63
  height and width attributes, 66
  including SVG file with, 65
  moving a graphic with, 69
  transform attribute, 69
  xlink:href attribute, 210
user interface events, 216
userSpaceOnUse setting
  for clipPathUnits, 148
  for filterUnits, 157
  for gradientUnits, 116, 120, 122
  for markerUnits, 104
  for maskContentUnits, 149
  for maskUnits, 149
  for patternContentUnits, 110
  for patternUnits, 109
  for primitiveUnits, 157

V

V or v command (vertical lines), 89
validity (XML), 293
values attribute, animation elements, 198
variables, 310
  assigning values to, 310
vector graphics, 2
  scalability, 3
  uses of, 3
vertical lineto command, 88
viewBox attribute
  clipping content to, 148
  marker element, 104
  svg element, 30
  using to scale a pattern, 111
viewport, 27
  aspect ratio, 32
  clipping area, 145
  clipping content to, 148
  nested, 37
  possible declarations, 28
  specifying user coordinates for, 30

W

web browsers
  and SVG used as an image, 16
  background filter inputs and, 180
  file types, determining support for, 18
  foreign object support, 20
  Internet Explorer and SVG animation, 191
  support for SVG as images, 17
  SVG fonts and, 138
  SVG support, 6
web pages, using SVG in, 15–25
  foreign objects in SVG, 20
  including SVG in an img element, 16
  inline SVG in (X)HTML, 22–25
  SVG as an application, 18
  SVG files as CSS property values, 17
  SVG in other XML applications, 25
well-formed XML, 288
while loop, 314
whitespace
  in path commands, 90
  SVG handling of, in text, 141
word-spacing style, 128
World Wide Web Consortium Recommendation (SVG), 5
writing-mode style, 133

X

xlink:href attribute
  a element, 209
  feImage element, 164
  image element, 67
  linearGradient and radialGradient elements, 115
  mpath element, 204
  textPath element, 138
  use element, 63
XML, 285–297
  attributeType in animation elements, 192, 200

character encodings, 291
character references, 291
comments, 289
defined, 285
document structure, 286
 for SVG graphic, 6
Document Type Definitions (DTDs), 293
elements and attributes, 287
entity references, 289
linking a document to its DTD, 294
name syntax, 288
namespaces, 6, 10, 295
parsers, 296
SVG as XML application, 5
SVG in other XML documents, 25
validity, 293
well-formed, 288
XML declaration, 286
xml:space attribute, 141
xmlns attribute, 295
 svg element, 6
xmlns:xlink attribute, 10
XPath, 275, 276, 281, 296
XSL-FO files, xi
 SVG in, 25
XSLT (Extensible Stylesheet Language Transformations), 266
 converting XML data to SVG, 270–282
 how XSLT works, 272
 objectives, 270
 stylesheet structure, 273
 processors, 296

Z

Z command (closepath), 86

About the Authors

J. David Eisenberg is a programmer and instructor living in San Jose, California. David has a talent for teaching and explaining. He has developed courses for HTML and CSS, JavaScript, XML, and Perl. He teaches computer and information technology courses at Evergreen Valley College in San Jose. He has also developed online courses providing introductory tutorials for Korean, Modern Greek, and Russian. David has been developing education software since 1975, when he worked with the Modern Foreign Language project at the University of Illinois to develop computer-assisted instruction on the PLATO system. He is coauthor of *Introducing Elixir*. When not programming, David enjoys digital photography, caring for a feral cat colony at work, and riding his bicycle.

Amelia Bellamy-Royds is a freelance writer specializing in scientific and technical communication. She helps promote web standards and design through participation in online communities such as Web Platform Docs, Stack Exchange and Codepen. Her interest in SVG stems from work in data visualization and builds upon the programming fundamentals she learned while earning a BSc in bioinformatics. A policy research job for the Canadian Library of Parliament convinced her that she was more interested in discussing the big-picture applications of scientific research than doing the laboratory work herself, leading to graduate studies in journalism. She currently lives in Edmonton, Alberta. If she isn't at a computer, she's probably digging in her vegetable garden or out enjoying live music.

Colophon

The animal on the cover of *SVG Essentials* is a great argus pheasant (*Argusianus argus*). This pheasant can be found in Malaysia, Thailand, Sumatra, and Borneo, where it lives in tropical rainforests. The males have blue faces, black crowns, and short crests; their underparts are mottled brown. The iridescent spots on their wings and tail feathers aid in attracting females. Female argus pheasants are smaller than males and lack their ornate plumage.

The great argus pheasant's wings can continue to grow into the bird's sixth year. Its tail feathers are the longest of all birds, measuring up to 5.7 feet. Some cultures use these feathers in their headdresses.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is a 19th-century engraving from the Dover Pictorial Archive. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.