Microsoft

Building Web Applications with SVG

David Dailey
Jon Frost
Domenico Strazzullo

Building Web Applications with SVG

Create rich interactivity with Scalable Vector Graphics (SVG)

Dive into SVG—and build striking, interactive visuals for your web applications. Led by three SVG experts, you'll learn step-by-step how to use SVG techniques for animation, overlays, and dynamic charts and graphs. Then you'll put it all together by building two graphic-rich applications. Get started creating dynamic visual content using web technologies you're familiar with—such as JavaScript, CSS, DOM, and AJAX.

Discover how to:

- Build client-side graphics with little impact on your web server
- Create simple user interfaces for mobile and desktop web browsers
- Work with complex shapes and design reusable patterns
- Position, scale, and rotate elements using SVG transforms
- Create animations using the Synchronized Multimedia Integration Language (SMIL)
- Build more powerful animations by manipulating SVG with JavaScript
- Apply filters to sharpen, blur, warp, reconfigure colors, and more
- Make use of programming libraries such as Pergola, D3, and Polymaps

Get code and project samples on the web
Ready to download at
http://go.microsoft.com/FWLink/?Linkid=257519

*For **system requirements**, see the Introduction.*

About the Authors

David Dailey, a professor who's taught computer science at several colleges and universities, specializes in web programming.

Jon Frost, coauthor of *Learn SVG: The Web Graphics Standard*, develops applications that include dynamic and interactive visualizations.

Domenico Strazzullo, founder and editor-in-chief of *SVG magazine*, is the author of both the Pergola JavaScript library for SVG and the open-source GEMï web operating system.

microsoft.com/mspress

ISBN: 978-0-7356-6012-0
9 780735 660120 90000

U.S.A. $34.99
Canada $36.99
[*Recommended*]

Programming/Web

Microsoft®

Microsoft®

Building Web Applications with SVG

David Dailey
Jon Frost
Domenico Strazzullo

ISBN: 978-0-7356-6012-0

1 2 3 4 5 6 7 8 9 LSI 7 6 5 4 3 2

Printed and bound in the United States of America.

Acquisitions and Developmental Editor: Russell Jones

Production Editor: Holly Bauer

Editorial Production: Zyg Group, LLC

Technical Reviewer: Domenico Strazzullo

Copyeditor: Zyg Group, LLC

Proofreader: Zyg Group, LLC

Indexer: Zyg Group, LLC

Cover Design: Twist Creative • Seattle

Cover Composition: Karen Montgomery

Interior Composition: Zyg Group, LLC

Illustrator: Rebecca Demarest

I would like to dedicate this book to my wife, Caron: my friend and companion on so many adventures.

—David Dailey

I would like to dedicate this book to my mentors in the local community, who consistently demonstrate their authentic passion for improving our town by regularly organizing events that coordinate efforts to revitalize our world, and who manage it all with an inspiring degree of heartfelt warmth and charm: Eduardo Crespi of Centro Latino, Mark Haim and Ruth Schaefer of Peace-Works and Sustainability, and Proffessor Miguel Ugarte.

—Jon Frost

I dedicate this book to the community of SVG adepts and evangelists who have given so much time and effort.

—Domenico Strazzullo

Contents at a Glance

Contents

Introduction

Scalable Vector Graphics, known as SVG, is the World Wide Web Consortium standard for graphical interactivity on the web and mobile platforms. SVG is a mature standard, first released more than a decade ago and has been under improvement by the W3C ever since. SVG is now available natively in all modern web browsers, as well as more than one billion mobile devices. SVG provides ways to create interactive graphics that can be rescaled without loss of clarity. Like HTML and HTML5, SVG coexists happily with technologies that are already familiar to web programmers, such as CSS, JavaScript, the Document Object Model, AJAX and, indeed, with HTML itself.

This book provides a comprehensive introduction to the language and how to use it for interaction and animation. The text also provides exposure to several important JavaScript packages and libraries, including D3, jQuery, and Pergola. While the book does not provide exhaustive coverage of every feature of the SVG language, it does offer essential guidance in using the key SVG components.

In addition to its coverage of basic SVG features, the book discusses a wide range of software tools for creating SVG and for embellishing it with scripted functionality. You'll also find solid introductions to complex topics such as SVG animation and filters. In many places, the book includes step by step examples and references numerous examples and downloadable sample projects that you can explore for yourself.

SVG Testimonials

Many people have been involved in the creation of SVG. As part of the Introduction to this book, we asked a handful of people who were closely involved in SVG's evolution to expound a little on what they think about SVG's past and future. Here are their statements.

Jon Ferraiolo

The W3C launched the Scalable Vector Graphics Working Group in 1998 to provide the vector graphics counterpart to HTML. The SVG WG chose to adopt all of the same general approaches as HTML (markup, DOM, scripting, styling) but replaced HTML's <div>, <p> and <span> elements with vector graphics element such as <rect>, <circle> and <path>. With various events in 2001 (SVG 1.0 Specification approval, Adobe SVG Viewer version 3 (ASV3) and bundling of ASV with Adobe Acrobat Reader 5), SVG was ubiquitous on desktop browsers, with the result that temporarily SVG took off like gangbusters, with tens of thousands of developers using SVG for various sorts of interactive graphics applications (flow charts, business graphics, and mapping). But SVG adoption dropped once Adobe abandoned ASV. Subsequently, the open source browser teams added SVG support (first Mozilla, then WebKit). With the open source project "SVGWeb" supporting older versions of SVG in IE6–8 and Microsoft's announcement of SVG support in IE9, SVG has once again regained ubiquity, and developers are now (re)discovering the power and coolness of DOM-based scriptable graphics.

The future for SVG looks quite exciting, particularly when using SVG as a component of HTML5. The W3C, in collaboration with the browser teams and the community, is generalizing many of SVG 1.0's best features (e.g., clipping, animation, filter effects) into CSS so these features will also be available to HTML, and cleaning up SVG to make it easier to use (e.g., removing SVG's XML requirement). There is active discussion about going to the next level with vector and raster graphics effects, particularly ones that are able to leverage CPUs. Given the automatic update features of the modern browser, developers will be able to take advantage of cool new features almost as soon as they are defined.

Background: Jon Ferraiolo was one of SVG's principal architects. He was the primary author of the PGML submission that served as the starting point for SVG and was the sole editor of the W3C's original SVG specification (SVG 1.0). While employed at Adobe Systems, Inc., he was the architect for several SVG-related projects at Adobe, including the Adobe SVG Viewer and Adobe Illustrator's SVG support. He is now a Distinguished Engineer at IBM.

Alex Danilo

In the early days of the web, browsers were rapidly changing and competition was fierce. When the W3C sent out a call for vector graphics proposals for the web, a collective cheer from thousands of graphics people could be heard. At last, to be free of those ancient bitmaps and bring the web into beautiful resolution and independent glory. This was the birth of SVG.

As we know, Rome wasn't built in a day, and over the years SVG was massaged and honed to perfection by an army of enthusiastic graphics aficionados. The result is a gem that's polished and can glisten with vibrant color when viewed in the right light.

SVG enables vivid interactive experiences that adapt to any display size, a way to bridge images with meaningful semantics, a powerful synergy with HTML and the DOM and just looks so good!

Background: Alex Danilo joined the W3C SVG Working Group at the start of 2002 and is now the representative of his company Abbra. Abbra's implementations both for mobile devices and web have always been at the cutting edge of the development of the SVG specification. Alex has very often produced the first proof of concept of new proposals for SVG. His current focus is development of a rich-media capable SVG engine for cross-platform application areas especially in resource constrained devices.

Cameron McCormack

It has been 10 years since the W3C Recommendation for SVG 1.0 was published, and having been involved in the SVG community for most of that time period, I can say with first-hand knowledge that SVG's fortunes have definitely been mixed. This is not an indictment on the technology itself, which is solid, but a historical problem of implementation availability.

In the early 2000s, there was a good deal of interest in SVG, as evidenced probably most clearly by the activity on the SVG Developers Yahoo Group mailing list, a forum that is still running today. Authors were creating visually rich, graphical, dynamic web applications with SVG before it became popular (or possible) to do so with other open web technologies. That this was possible at the time was, in my view, nearly entirely due to Adobe's investment in SVG and their development of the Adobe SVG Viewer plug-in. It did not matter that browsers' support for SVG was not up to scratch or did not exist at all—through the use of the Adobe plug-in, SVG was available to everyone. (Technically not everyone, of course, as the plug-in was limited to particular operating systems and architectures, but for most authors this was good enough.)

The last release of the Adobe plug-in, a preview of version 6, was made available in 2003. The preview release was somewhat unstable, but demonstrated attractive new features, including a componentization model for SVG content whose fundamental ideas even today garner interest despite a number of false starts in standardization groups. However, for a long time after this release not a word was heard out of Adobe on their plans for development. This caused growing consternation within the SVG developer community, as progress of native browser implementations had been slow to catch up to the features and performance of the plug-in. Interest in SVG began to wane, and Adobe's acquisition of Macromedia and the Flash platform only served further to fuel the notion that SVG was dead. The years following were the Dark Ages of SVG.

Although native browser implementations did improve during this time, there was still a perpetual sense by developers at large that SVG wasn't ready for prime time. What was probably the biggest impediment to authors publishing SVG content was the lack of implementation in Internet Explorer. With the arrival of one particular version of IE or Windows, I don't remember which, the unmaintained Adobe plug-in stopped working altogether. This was a blow to developers, as Microsoft had no plans to implement SVG at all, unlike the other major browser vendors who all were committed to supporting it.

In 2008, a major development occurred: the addition of SVG (and MathML) to the HTML5 specification, which allowed authors to write HTML documents with inline vector graphics without having to use mixed-namespace XML documents. This was a welcome simplification, but importantly it helped to sell SVG as being a first class part of the web platform.

By 2009—the same year that Adobe finally announced what everyone knew already, that their plug-in was no longer being maintained—sentiment had finally managed to shift away from the notion of SVG being a neat technology unsuitable for publishing on the web due to Microsoft's intransigence. This was helped by the release of SVG Web, a Flash-based SVG renderer developed by a team at Google. Once again, authors had a way to target SVG content to Internet Explorer, as most Windows computers already had Flash installed. Not only did SVG Web provide a way to render SVG in IE, it did so with reasonably complete coverage of the SVG specification and with great performance.

But perhaps the most welcome news to the SVG community came in 2010 when Microsoft announced a preview release of Internet Explorer 9, the first version of IE to support SVG. Finally it would be possible to publish SVG content using open web technologies and have all desktop browsers consume it without the need for any plug-ins or workarounds. Hooray!

Today, SVG is in its strongest position yet. Browser implementations continue to improve by leaps and bounds. Standards groups continue to draw SVG and CSS ever closer, allowing the use of SVG features such as filters, patterns, and gradients in HTML documents. The SVG Working Group itself is busy working on the next major revision of the SVG specification itself to address issues and add features that have been requested by the persevering SVG community over the years. JavaScript toolkit writers are choosing SVG as their graphical output technology.

And the developer community is reinvigorated. SVG is very much alive!

Background: Cameron McCormack has been involved in SVG since 2003 and has served as coeditor of the SVG specification and cochair of the SVG Working Group from 2007 to the present. As a graduate student at Monash University in Australia, Cameron also spearheaded the implementation of SVG in Batik—sometimes called the most extensive implementation of SVG yet. He has since gone on to work at Mozilla Corporation, where his work with SVG and other web standards continues.

Jeff Schiller

I became involved with Scalable Vector Graphics (SVG) around the time that Firefox was planning to ship its first partial implementation of SVG Full in Firefox 1.5. At that time, native support was mostly a curiosity given that there was a very mature browser plug-in (Adobe SVG Viewer) and sound alternatives to rich vector graphics in web applications (Macromedia's Flash). But what intrigued me about native SVG support was the integration with HTML: a DOM, an event model, scripting in JavaScript, styling with CSS. This would allow graphical web applications to take advantage of the AJAX bubble that was happening at the time: rich, dynamic applications that worked cross-browser without a plug-in.

More SVG Full implementations began showing up, first in Opera which set the standard for Full support, then in WebKit and finally in Internet Explorer, making it ubiquitous across the web and mobile. As native SVG support began showing up in the wild, HTML5 really started to take shape in the minds of browser vendors and I've been delighted to follow both SVG and HTML as their paths became aligned. I believe the arrival of graphics in the browsers (SVG and HTML Canvas) were essential in making the web platform compelling for application developers: a powerful markup vocabulary, a document model, a simple authoring syntax, and continuously improving support in all major browsers. Refinement of both the implementations and the specification have made SVG a really effective weapon in the web developer's arsenal and I'm constantly amazed at what people are doing with it.

Background: Jeff Schiller's name is a familiar one in the SVG community. In addition to being the originator of and contributor to the popular and useful tools SVG-Edit and Scour, he has also for many years maintained the web's most definitive site for cross-browser comparison of the completeness of the implementation of SVG. He also spearheaded and chaired the W3C's SVG Interest Group, and has made numerous contributions to the evolution of the standard itself. Jeff began his work with SVG while working at Motorola and is now a Google employee.

Doug Schepers

The fundamental idea of SVG is beautiful: take the best from popular vector programs like Illustrator, and the structure, dynamic adaptability, and hyperlinking of web formats like HTML and CSS, and then add in animation and raster effects like filters to make it fun, funky, and functional.

Now that it's supported in every modern browser, with tons of applications that output SVG, the W3C SVG Working Group is turning its eye toward SVG 2. What's in the cards? Certainly more seamless integration with HTML5 and the assorted APIs that go into making awesome web apps (though most of them already work with SVG), and a general tidying up of the language to make common tasks easier for developers and implementers, and a massive improvement to the DOM API to increase speed and usability. We're also working closely with the CSS Working Group on shared features, like filters for HTML, and we plan to adopt some new CSS features, including complex text wrapping into and around shapes, a long-standing SVG request.

And while it may sound a bit boring, we have a plan to work on smaller, more modular specs; what this means to developers and designers is more features more quickly. Look for things like parameters (highly adaptable images) and features for mapping (like non-scaling strokes and declarative level-of-detail) to come out as modules. And we are always looking for use cases and requirements that solve real-world problems for developers.

Background: Doug Schepers has been involved in SVG as a developer since the very early days, starting in 2001. He was deeply involved in raising the public's awareness of SVG. In 2007 he was hired by W3C itself to serve on the Working Group. Doug's footprints can be seen all over the SVG specification from its earlier days through the present.

Who Should Read This Book

This book is designed as both a basic introduction and a more advanced treatment that delves deeply into some of the advanced aspects of SVG. It should be equally accessible to a professional web programmer, an undergraduate student with a few semesters of computing coursework, a scientist who wants to make large datasets more interactive, or a graphical designer with a strong technical side. In short, if you are familiar with the basics of web development and computer graphics and have an interest in developing websites that are richly graphical and interactive, then this is the right book for you.

Assumptions

This book assumes some familiarity with HTML and web graphics. Prior experience with programming is not a requirement, though prior programming experience will clearly help you understand some of the chapters (such as Chapter 4 and Chapter 7) that involve programming. Familiarity with the basics of coordinate geometry and fluency with high school algebra will likely also aid in comprehension—though that would be true with any treatment of graphics involving the x-y plane—so the foray into mathematics you'll find here should prove to be a gentle one.

With a heavy focus on database concepts, this book assumes that you have a basic understanding of relational database systems such as Microsoft SQL Server, and have had brief exposure to one of the many flavors of the query language known as SQL. To go beyond this book and expand your knowledge of SQL and Microsoft's SQL Server database platform, other Microsoft Press books such as *Programming SQL Server 2012* offer both complete introductions and comprehensive information on T-SQL and SQL Server.

Who Should Not Read This Book

A graphical artist who finds notation distasteful will probably not find either SVG or this book to his or her liking. SVG is a declarative language based on XML; accordingly, it has a rigorous syntax that is not forgiving of grammar errors. If you're interested in a purely point-and-click environment, or simply want to create a graphical user interface containing drawings and illustrations, then a package such as Inkscape or Illustrator may prove to be a better direction for your creative expression.

Web authors who primarily develop web pages with a package such as Microsoft Expression Studio or Adobe Dreamweaver rather than coding HTML by hand may be interested in some of the new software tools being developed for integrating SVG and HTML. However, while this book discusses some of these tools briefly, the book is not intended as a tutorial in the use of design packages.

Organization of This Book

This book is organized in seven chapters. Chapter 1, "SVG Basics," orients the reader to SVG itself, showing how to get started, the contexts in which SVG can be created and viewed, and a diverse sampling of examples that may whet the reader's appetite. Chapter 2, "Creating and Editing SVG Graphics," and Chapter 3, "Adding Text, Style, and Transforms," get into the dynamics of the core of SVG: the basic shapes, patterns, gradients, clips, masks, and images. Chapter 4, "Motion and Interactivity," introduces the two fundamental aspects of SVG interactivity: animation and scripting. Chapter 5, "SVG Filters," discusses filters, one of the most complex and powerful parts of the graphical language. Chapter 6, "SVG Tools and Resources," and Chapter 7, "Building Web Applications: Case Studies" introduce and provide examples of the broad range of tools and libraries that support SVG development.

Conventions and Features in This Book

This book presents information using conventions designed to make the information readable and easy to follow.

- This book has numerous examples in which the reader may examine the illustration itself and the code used to create the example.
- On occasion, the code shown is an excerpt showing only the parts needed for understanding the narrative text. In such cases, a link is provided to a working example on the web, so that the reader may examine a complete working example.
- In cases of very lengthy source code, the example has been annotated in a table so that blocks of code and explanatory comments may be seen side by side.

About the Companion Content

Most of the chapters in this book include exercises that let you interactively try out new material learned in the main text. The working examples can be seen on the web at:

http://www.microsoftpressstore.com/title/9780735660120

or

http://cs.sru.edu/~svg

The examples are organized by chapter number as well as linked from the above addresses.

Installing the Code Samples

There's no need to "install" the code samples for this book—you simply need a browser that can display SVG.

System Requirements

You will need the following hardware and software to be able to follow along with the step-by-step examples in this book:

- A modern web browser: Microsoft Internet Explorer 9 or 10, Firefox 6 or higher, Opera 8 or higher, or Safari or Chrome (any version).
- For mobile users: either Opera Mobile, the Android Ice Cream Sandwich OS, or the iPhone will suffice, though in truth, there are dozens of SVG-enabled browsers too numerous (and quickly evolving) to mention.
- A simple text editor (such as NotePad) or a syntax-completion environment (such as *http://notepad-plus-plus.org/* or *http://www.htmlkit.com/*) for editing your own examples.
- If you wish to share your content on the web: a web server that serves the proper mime type for .svg files, namely as "image/*svg*+xml".
- Internet connection to view examples that accompany the book.

Acknowledgments

Jon Frost initially came up with the idea for this book; his motivation brought it about and saw it through to completion. David Dailey was instrumental in bringing the vision of the book to light through his insight and wisdom and brought a healthy down-to-earth style to the writing process. Jon and David were fortunate to be joined by Domenico Strazzullo, originally brought in as a technical reviewer. His contributions were so energetic and thorough that we just had to have him write a chapter—and who better to do that than the author of Pergola himself?

David: I'd also like to thank my family for their patience and understanding during the writing process and my academic department and university for their generous support with my SVG-related endeavors. Also to the creators of SVG and the SVG Open folks: thanks for the language and for the fun.

Jon: I am grateful for my supportive family, my super-supportive and playful wife, my super-playful and loving dog, and my good friends from cultures around the world who continue to teach me the vital necessity of sharing and caring.

Domenico: I'd like to thank Microsoft Press and the editors at O'Reilly for giving us this terrific opportunity to expose SVG to the greatest number of developers, and help it reach a long deserved status.

Errata & Book Support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site:

http://www.microsoftpressstore.com/title/ 9780735660120

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://www.microsoft.com/learning/booksurvey

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in Touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*

CHAPTER 1

SVG Basics

I decided that if I could paint that flower in a huge scale, you could not ignore its beauty.

Georgia O'Keefe

In this chapter:

Scalable Vector Graphics (SVG) is a graphical standard maintained and endorsed by the World Wide Web Consortium (W3C), the same group that created and continues to maintain HTML, CSS, XML, and other technologies that constitute the World Wide Web.

The What, Why, and Where of SVG

SVG is much more than its name suggests. It is true that SVG is a language that allows for the creation of two-dimensional vector elements, which are simply mathematical representations of graphical objects, and that these vectors are infinitely scalable and can be transformed within the bounds of the 2D coordinate system. However, SVG is unique in that it is an open standard defined by the W3C (*http://w3c.org/svg/*), and like other W3C languages such as HTML and XML, it has its own Document Object Model (DOM) that brings with it many benefits, and it's interoperable with other open standard languages such as JavaScript, CSS, and HTML.

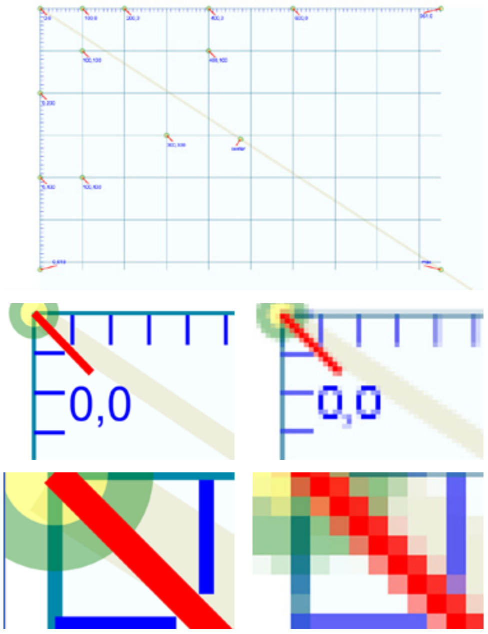
SVG has been in the works over the past decade and has matured a great deal during that time, with collaboration from interested parties around the world. The great appeal of SVG is that, like HTML, it's easy to read and edit, while allowing for complex interactivity and animations through scripting and Synchronized Multimedia Integration Language (SMIL), which is another W3C standard. Browsers have matured over the last few years, and all the major ones now natively support much of the SVG specification, so you no longer need to fuss with proprietary SVG plug-ins. All of these capabilities allow for a much greater degree of creativity, with complex interactivity mixing with animation and real-time data, all within the context of SVG-enhanced web applications. This is ideal for modern designers and developers, as demonstrated throughout this book.

The What

SVG is based on vectors rather than pixels. While a pixel-based approach (used by programs such as Adobe Photoshop) places pigment or color at xy-coordinates for each pixel in a bitmap, a vector-based approach (used by programs such as Adobe Illustrator) composes a picture out of shapes, each described by a relatively simple formula and filled with a *texture* (a term used broadly here to refer to a mixture of colors, gradients, and patterns).

SVG is scalable. As you may already know, if you zoom in on pixel-based art, you will eventually reach a maximum resolution. Even with the 10-megapixel cameras that are now commonplace (or the 100-megapixel cameras that can be had for a small fortune), increasing the zoom factor much beyond screen resolution will cause pixelation. Scalability is a tremendous advantage for the emergence of the mobile web, as well as for very-large-display devices (as for outdoor advertising).

The following image shows the difference between what happens when you zoom into a vector graphic (left) and a bitmap (right).



The Why

Some of the advantages of SVG are now discussed, with brief explanations:

- **Client-side graphics** Because SVG uses client-side graphics, its impact on your web server is light. In addition to being scalable, SVG is dynamic and interactive. A user can interactively explore the data underlying a picture in novel ways.
- **Open source (XML)** Anyone can view the source code that underlies the graphic. It's readable by humans and looks a lot like HTML.
- **Accessibility** Because the SVG source code is written in XML, it is also readable by screen readers and search engines. While a picture might be worth a thousand words, a megapixel image is not worth much at all to someone who can't see it. The ability of SVG to bring geometry to those who cannot see it extends its reach into many domains that pixel-based imagery just cannot go.
- **Open standard** Because it was created by the W3C (the same organization that brought us HTML and the web itself), SVG is nonproprietary and vendor neutral.
- **Familiar technologies** SVG uses technologies already familiar to web programmers: DOM, JavaScript, CSS, and AJAX. Rather than having to learn entire realms of technology, programming languages, and terminology to deal with the complex and technical area of computer graphics, designers, programmers, and web professionals can leverage skills learned elsewhere.
- **Web applications** SVG is suitable for incorporation with HTML5, web-based applications, and rich Internet applications (RIAs). The last 10 years have seen a great elevation of the status of the phrase *web-based application*. Not so many years ago, people in the web community used to respond with sarcasm or disbelief when someone talked about wanting to create a web-based application that lived primarily in the browser. A cursory inspection of the history of HTML5 reveals that the creation of web applications was one of the primary intentions behind the development of this emerging specification. The incorporation of inline SVG into the HTML5 specification is a great advantage for web developers.
- **SMIL** SMIL is a W3C declarative language supporting multimedia and animation for nonprogrammers. SMIL is partially incorporated into the SVG specification. Those who have had more than a cursory exposure to programming animation in JavaScript may find themselves enamored of the ease with which certain complex animations can be authored using SVG animation (or SMIL), as well as the ability to update many objects on the screen almost concurrently. While SVG also supports scripted animation through JavaScript, SMIL brings convenience, parsimony, and elegance to the table.
- **The adoption of SVG** As of 2010, SVG is supported natively by the most current versions of the five major web browsers. Additionally, it can be found in the chip sets aboard several hundred million mobile phones, with major support being offered from Nokia, Ikivo, Sony

Ericsson, Opera Mobile, Samsung, iPhone, and several others. This will be discussed further in the next section.

- **Other technologies** SVG has overlap with Flash, Vector Markup Language (VML), and Silverlight—but it has the advantages of being nonproprietary, standardized, cross platform, and interoperable with other XML languages and W3C standards.

The Where

Vector graphics are everywhere. The art world, for example, is replete with examples of the use of vector graphics. As Professor Jerrold Maddox wrote in "SVG and Art: Expanding the possibilities, different times and different places," "Image making based on vector-like forms is the way most of the art of the world is and has been made" (*http://www.personal.psu.edu/jxm22/svgopen/*). He continues, "The Song [dynasty] in China and Renaissance Europe are only times and places where tonal art ever took off—and photography made it seem like the only way to do it" (personal correspondence, 2011). Accordingly, from a global and historical perspective, we might see images that are not vector based as more the anomaly than the rule.

SVG, nowadays, is also pretty much everywhere. As of this writing, an estimated 1.5 billion devices in the world are SVG enabled (from *http://en.wikipedia.org/wiki/Usage_share_of_web_browsers* and *http://marketshare.hitslink.com/browser-market-share.aspx?spider=1&qprid=2*).

If we add to this the two mobile manufacturers whose devices are SVG enabled (Ikivo with 350 million users [*http://www.ikivo.com/04about.html*] and Apple, whose iPhone boasts another 100 million [*http://mashable.com/2011/03/02/100-million-iphones/*]) and Abbra's estimate that "Today over 700 million mobile phones have been shipped with in-built support for SVG version 1.1—more than twice as many as the nearest competing technology—FlashLite" (*http://abbra.com/products.html*), then our estimate rises to close to two billion devices that are SVG ready!

Adobe provided the first support for SVG in the browser (via a plug-in known as ASV 3) as early as 2000, though support in other applications (such as CORELDraw and Microsoft Visio) came earlier (*http://www.w3.org/G6raphics/SVG/History*). SVG has had considerable support in drawing programs, including Illustrator, CORELDraw, and Inkscape, for many years now, and it's also supported in a variety of Internet Protocol Television (IPTV) applications and in the popular KDE desktop environment for Linux.

In the browser market, Konqueror was the first browser to support SVG natively, in 2004 (*http://en.wikipedia.org/wiki/Scalable_Vector_Graphics#Native_support*). As of early 2005, the Opera browser had fairly extensive SVG support, and Firefox developed support for basic SVG shortly thereafter in version 2. By mid-2007, Safari had implemented basic support as well. Google debuted its Chrome browser in 2008, and in 2009 Microsoft announced that Internet Explorer would finally have native support, rounding out SVG support for all the major browsers.

Beyond browsers, there are several dozen software applications that read or export SVG content (see the list at *http://en.wikipedia.org/wiki/Scalable_Vector_Graphics*).

Getting Started: A Simple Overview

You'll see a more detailed step-by-step example at the end of this chapter, but it is important that you gain some idea of what's involved in viewing and creating SVG at the outset.

Viewing SVG

Start up any modern browser and point it at the website related to this book, *http://cs.sru.edu/~svg.com*. Internet Explorer, Firefox, Chrome, Safari, and Opera all support viewing SVG on the web, so you can use any of those. The most important exception is this: if you are using Internet Explorer, you will either need to upgrade to Internet Explorer 9 (which requires Microsoft Windows Vista or later), or you will need to download the free SVG plug-in (ASV version 3.03) from Adobe, at *http://www.adobe.com/svg/viewer/install/mainframed.html*. For all the other browsers listed, using the latest version will always prove helpful, because all of these browsers are making steady and frequent progress on their implementations of the SVG specification.

SVG is a big specification—one that's not trivial to implement. SVG 1.1 is generally the version against which browsers are compared. As of this writing, no browser implements all of SVG 1.1, despite the specification having reached recommendation status (meaning that it is officially a W3C standard) in 2003 (*http://www.w3.org/Graphics/SVG/History*). Improvements to browser support tend to appear on a monthly basis, so it is best to make sure that you're using the latest release of whatever browsers you use.

As another example of the importance of using current browser versions, Firefox 3.6 does not support SMIL animation, while Firefox 4.0 does. You'll see more about the idiosyncrasies of browser support in the discussions of the relevant topics, but note that the parts of SVG that pertain to animation, filters, and fonts are most likely to show browser differences.

Writing SVG

There are many different paths that one can follow to develop SVG. This book will show you several of those in more detail in Chapter 6, "SVG Tools and Resources." In the meantime, we recommend using any simple text editor—for example, Notepad for Windows or TextEdit (properly configured for Mac; see *http://support.apple.com/kb/HT2523*)—or any of the plethora of editing tools in Linux or UNIX (nano, pico, emacs, vi, ed, kate, vim, kwrite, gEdit, etc.).

First, enter this very simple SVG file into your text editor, and save the file with the name ***myfirstfile.svg*** (you can save the file to your local hard drive or a remote server, so long as you know how to get to it from your web browser):

```
<svg xmlns="http://www.w3.org/2000/svg">
<circle r="50"/>
</svg>
```

The file is also visible at *http://cs.sru.edu/~ddailey/svg/simplest.svg* should you have any problems seeing the file you've created.

You'll see information about more advanced editing environments at the end of this chapter, and you'll of course see many more examples of SVG code throughout the rest of the book.

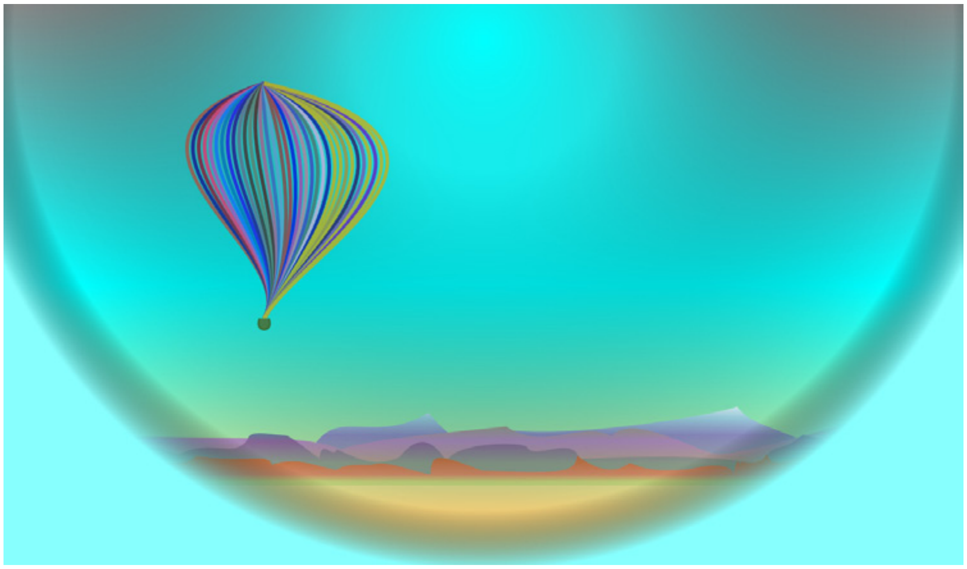
Thirteen Examples That Show the Capabilities of SVG

To fully appreciate the power of SVG, complete with its interactivity and animation capabilities, I encourage you to take a look at the tutorial page on this book's website (*http://cs.sru.edu/~svg*), which contains links to interesting examples, and also to explore and read the examples illustrated and briefly discussed below.

Note We haven't yet defined the terms for the effects described below, but we'll make them clear later on. At this point, we simply want to ensure that you have some idea of what SVG can accomplish before you begin working with it. How else to know the lay of the land?

Example 1: Dynamic Random Landscape Drawn with JavaScript and SVG

The scenery here, inspired by one author's frequent drives from his homeland in New Mexico to his graduate school in Colorado, shows the effect of motion parallax on the various mountain ranges leading from the foothills to the Continental Divide. As the vantage point moves continually northward toward the badlands of Wyoming, a slightly impressionistic hot-air balloon follows. Its vertical position, speed, and wind deformation change somewhat randomly as we move. The various layers of mountains recede behind us to the left, with the taller peaks remaining visible longer. Owing to the use of random elements, no two landscapes will ever be the same (ignoring the infinitesimal probability of extreme coincidence). The example can be seen at (*http://srufaculty.sru.edu/david.dailey/svg/balloon.svg*).



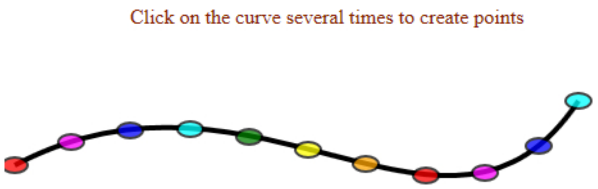
Here's how it's done:

- **The sky** The sky consists of two rectangles. One, the background, is simply filled with a linear gradient consisting of colors that move from brighter shades of sky blue to gray, from bottom to top. The second rectangle provides a snow globe effect. The foreground and smog, due to the overpopulation of communities along the front range, are simulated through the color transitions in the foothills and the overlay of gray stemming from the background and foreground.
- **The snow globe effect** This is produced using a radial gradient of varying transparency in the foreground. With SVG gradients, you vary not only the colors as they change gradually from one to another, but also their relative opacity.
- **The balloon** The balloon is entirely handled through JavaScript. A series of almost parallel Bézier curves is created with start points and endpoints that coincide. The control points differ and change over time. The entire group (a *<g>* element in SVG) then has its horizontal and vertical positions varied through a timed loop that refreshes the screen every 10 milliseconds.
- **The drawing of the mountains** There are four layers of mountains, each filled with a linear gradient that changes from yellow-brownish in the plains and foothills to the blue-white of the snowcapped peaks of the Continental Divide. The hint of green in the second range behind the foothills is meant to suggest the presence of the forests there. The heights of the peaks are randomly determined, with an array of random xy-coordinates being first generated and then sorted by their x-values. Then they are divided into triplets so that the peaks can be connected by a series of curves, each having the previous endpoint and the next separated by points in a cubic Bézier curve.

- **The movement of the mountains** The foreground layers are simply shifted leftward more quickly than the layers in the back. Each array has its first element removed so that another random element can be added onto the end of the array without the array becoming arbitrarily large. Any memory of what has happened is systematically purged.

Example 2: Equidistant Positioning Points along a Bézier Curve

The mathematics of Bézier curves, while quite accessible to a mathematician, are not trivial. Bézier curves were, after all, not discovered until 1959 (see *http://en.wikipedia.org/wiki/B%C3%A9zier_curve*), 130 years after Évariste Galois resolved the theory of roots of polynomials and laid the foundation for much of the algebra of the 19th and 20th centuries. Fortunately, SVG (following the lead of Adobe Illustrator 88) gives direct and intuitive access to these wonderfully expressive curves in terms of the ability to draw, measure, subdivide, orient, and animate them. In this example (visible at *http://srufaculty.sru.edu/david.dailey/svg/curve.svg*), the curve is drawn with a simple set of markup commands, and each time the user clicks the curve or near it, JavaScript is used to measure the curve and divide it into an increasingly larger number of parts, with the option of animating the process ultimately being offered to the user.



Here's how it's done:

- **Drawing the curve** The markup used is quite simple:

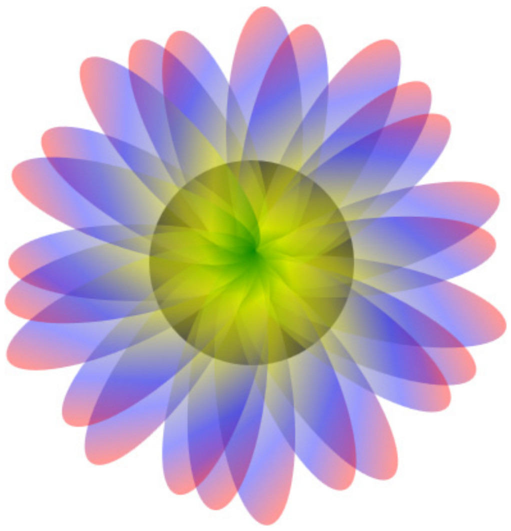
```
<path d="M 10 150 C 200 80 350 300 450 100" id="B"
      stroke="black" fill="none" stroke-width="4"/>
```

Note The drawing of SVG paths is one of the most powerful and expressive aspects of the language; it's covered in Chapter 2.

- **Measuring and subdividing the curve** The JavaScript language binding of SVG allows you to interrogate properties of things that have been drawn either through markup or dynamically, and to manipulate them using methods. In this case, we are using two function calls: *L = B.getTotalLength();* and *P = B.getPointAtLength(L * i / n);*. The first measures the path, B, and returns a numeric value; the second returns a point (an object with both x and y values) a given fraction of the distance along B. Script is then used to create new ellipses of different colors at those fractional mileposts.

Example 3: Simple Animation (Just 38 Lines of Markup and No Script)

This example, visible at *http://srufaculty.sru.edu/david.dailey/svg/ovaling.svg*, has been cited by others for the richness it achieves even with such simplicity. The example uses SMIL animation to simultaneously vary 4 different attributes of 26 different objects. At the SVG Open 2010 conference in Paris, one of Microsoft's demonstrations showed that this particular example could be animated using one of several SMIL emulators for SVG, although as of this writing, most browsers can run the animation without additional assistance. Creating such a rich animation with other technologies, such as the HTML5 <*canvas*> tag or Java Applets, would take much more code, thought, experimentation, and time to develop.



Here's how it's done:

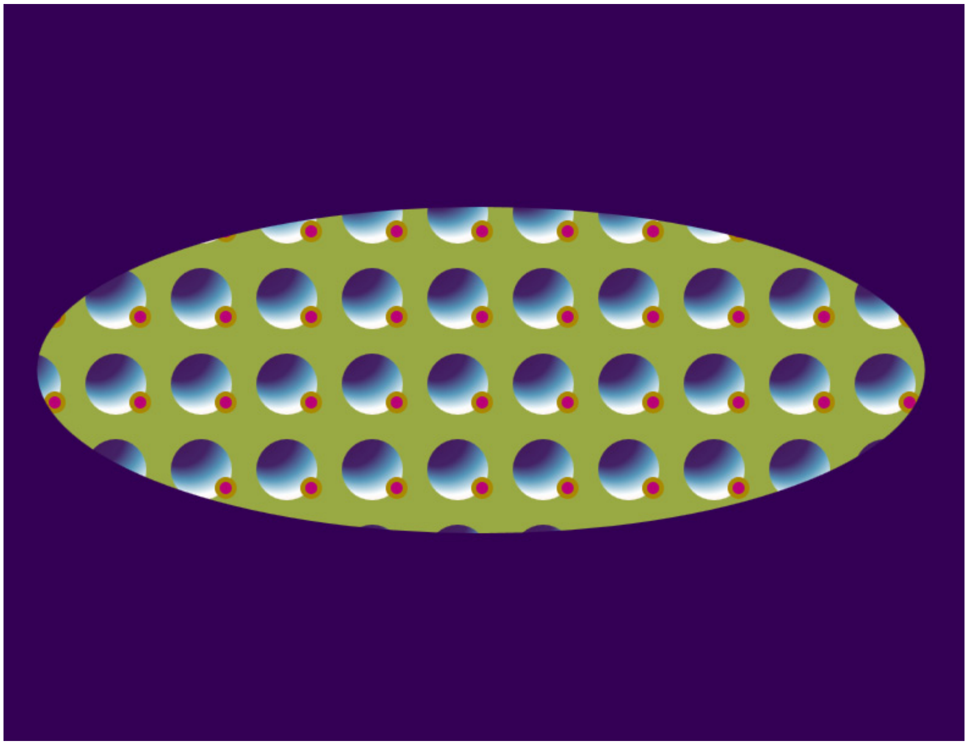
- **Drawing one petal of the flower** An ellipse is drawn with a given centroid and differing radii in the x and y directions. It is made slightly more transparent than opaque (the *opacity* is set to 0.4). It is then filled with a gradient (in this case, a linear gradient moving from red to blue and then through green to yellow).
- **Replicating the petal** SVG allows considerable reuse of code. In this case, the initial petal is reused four times through a series of <*use*> elements, each of which applies a different rotation to the petal. This creates a petal cluster, which itself is then grouped and reused 5 more times, for a total of 25 petals being drawn with only 9 lines of markup.

- **Animating the illustration** The initial petal of the flower (which is later replicated) has three separate animations applied to it. The first gradually changes its orientation from 0 to 360 degrees over a period of 7 seconds. The next 2 animations vary the x value of the centroid and the radius in the y direction over, respectively, 8 seconds and 3 seconds. Because 3, 7, and 8 are relatively prime, the entire animation will repeat every 168 seconds (3 × 7 × 8 = 168). Because the animation is applied to a petal that is then reused 24 times, each of the 25 petals inherits the same animation, with the rotation and repositioning being applied relative to each differing initial position. One more circle at the center of the composition has its own color animated to add a pleasant bit of chromatic variety.

Example 4: Use of Gradients and Patterns

This example, visible at *http://srufaculty.sru.edu/david.dailey/svg/grid2.svg*, consists of just 19 lines of markup (not counting its animations) and no JavaScript. It demonstrates that some rather intriguing results can be concocted by juxtaposing some quite simple SVG elements.

After you have grown accustomed to SVG, animations of this sort will be remarkably easy to create and experiment with on your own.



Here's how it's done:

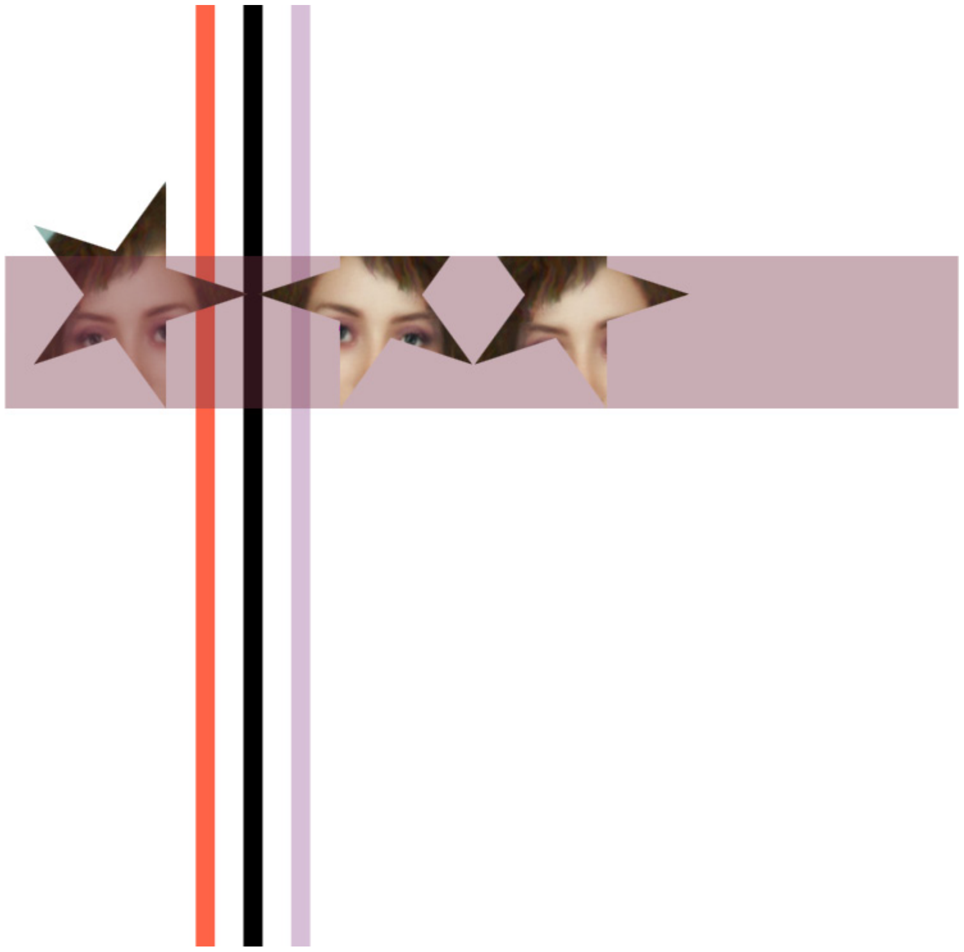
- **Creating the repeating pattern** In this case, the pattern consists of two circles (one filled with an off-center radial gradient and the other with a flat color and a different-colored stroke).
- **Restricting the pattern to a shape** The pattern is then applied to an ellipse (which of course is animated).

Example 5: Intersecting Clip Paths

The example at *http://srufaculty.sru.edu/david.dailey/svg/newstuff/clipPath4.svg* demonstrates four things:

- SVG allows bitmapped images (.png, .jpg, and .gif) to be imported and used in conjunction with other graphical primitives. As you will see later, this is done through the <*image*> element.
- Images and other shapes can be clipped to the confines defined by a given shape (in this case, a five-pointed star) using the <*clipPath*> element.
- There is more than one way of making clip paths intersect. Here, the lavender rectangle intersects the five-pointed stars in two rather different ways.
- Like almost all things in SVG, clip paths can be animated. The example uses SMIL animation to rotate the stars, revealing different parts of the underlying faces.

This particular example, first constructed in 2006, has served as a mini-benchmark test for browsers. Originally, it only worked properly in Internet Explorer with ASV. Over time, Opera came to handle the multiple clip paths and the animation, and each of the other browsers has been gradually phasing in correct handling of intersected clip paths as well.

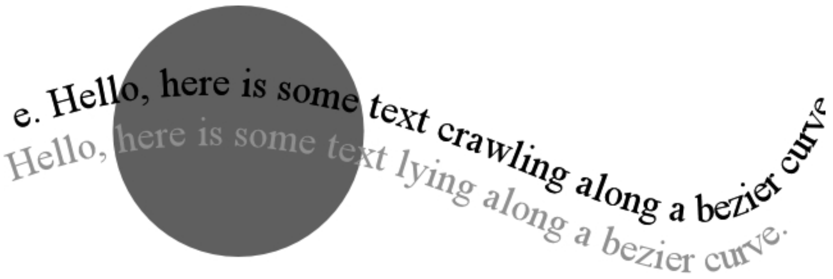


Here's how it's done:

- **Clipping an image by a shape** The leftmost image is defined by an *<image>* element. Its attribute *clip-path="url(#CPST)"* references the element *<clipPath id="CPST">*, which itself contains a star-shaped *<path>* element.
- **Clipping a clip path** This is done in either of two ways in this example: First, the *<image>* element to which a clip path has been applied is reused with a *<use>* element. The *<use>* element then has another clip path applied to it (which happens to consist of the lavender rectangle). The two clip paths intersect as would be expected. The other approach is to build a *<clipPath>* that has its own *clip-path* attribute defined. This works in Internet Explorer 9, Opera, and Internet Explorer with ASV, as you would expect, and is the same regardless of whether the secondary clip path is applied to the parent *<clipPath>* or the elements within it. The other browsers show a variety of idiosyncratic responses to this approach.

Example 6: Animated Text Crawling Along a Bézier Curve

To anyone who enjoyed the excitement of new applications being unveiled in the Macintosh environment of the mid-to-late 1980s, Adobe Illustrator's ability to allow the layout of text to follow an arbitrary curve, using a simple graphical user interface (GUI), fell in the category of "utterly cool." The example at *http://srufaculty.sru.edu/david.dailey/svg/newstuff/textpath1.svg* demonstrates that SVG can do this—and go one step further: it can animate the text moving along that curve!


e. Hello, here is some text crawling along a bezier curve
Hello, here is some text lying along a bezier curve.

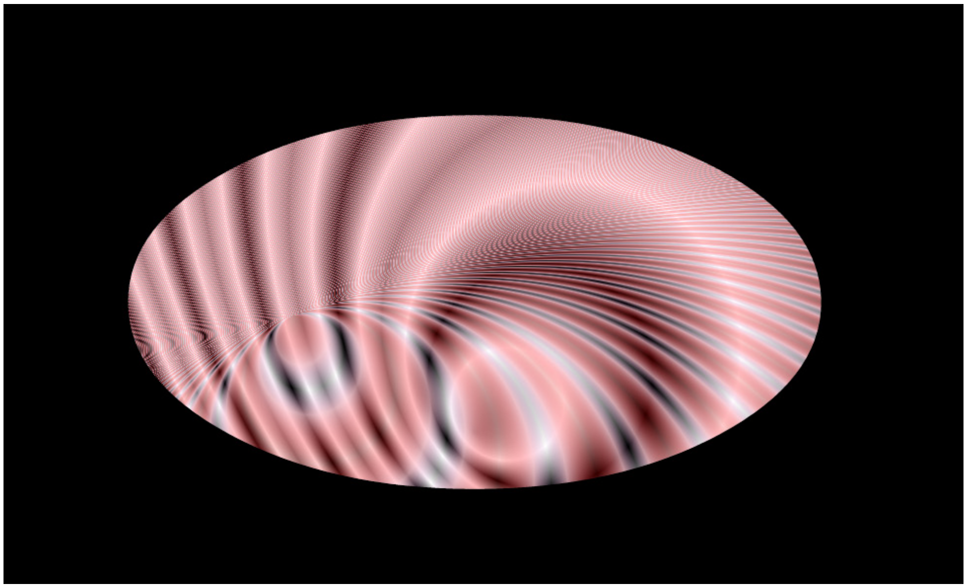
Here's how it's done:

- **Laying text along a path** While this will be discussed with examples later in the book, it works rather like this: First, running text (a sequence of characters) is placed in an SVG <*text*> element. Also in the <*text*> goes a <*textPath*> element that has a simple URI reference to the ID of the <*path*> element.
- **Animation of text following a path** One attribute of <*textPath*> is *startOffset*. Its value determines an offset for the initial position of the text. That is, a value of zero means that the text will begin at the start of the path; higher values mean that the text will begin closer to the endpoint. The effect is accomplished by simply animating that value with an SVG <*animate*> element.

Example 7: Animated Reflected Gradients with Transparency

Some of the effects offered by SVG seem to be more interesting than useful. This is often true of things like pure mathematics, until one's imagination discovers (or invents) their utility. The radial gradients available in SVG have the ability to repeat bands of color, using the values of *reflect* or *repeat*.

When seen in motion at *http://srufaculty.sru.edu/david.dailey/svg/newstuff/gradient11c.svg*, this example is quite impressive. It's best rendered by Chrome and ASV.

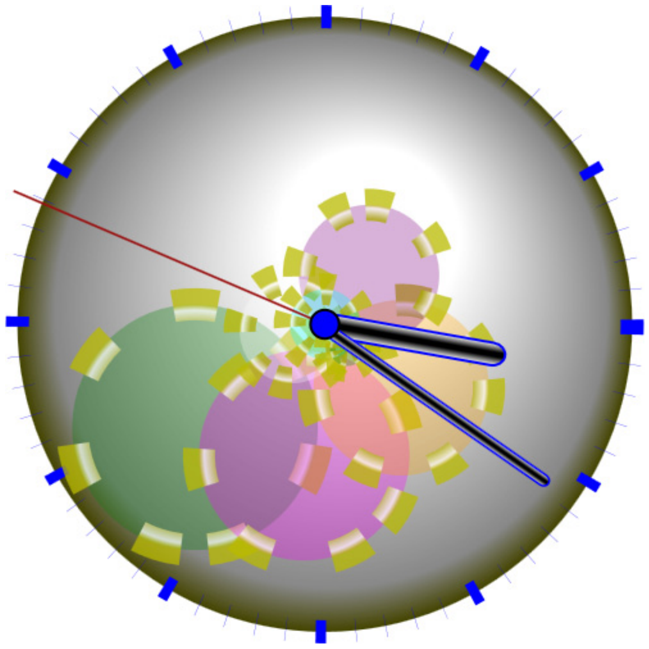


Here's how it's done:

- **The two swirly gradients** SVG has two primary sorts of gradients: linear and radial. The radial gradient allows for a special type called a *reflected* gradient. In this case, two identical ellipses are located one atop the other. Both have alternating bands of opacity and transparency coinciding with their alternating colors, which allows us to see through to the background.
- **The animation** The center and focal points of the reflected gradients are then independently animated using SMIL animation.

Example 8: Clock with Impressionist Tinge

There are lots of SVG clocks on the Web. Displaying time is a medium of expression ripe with opportunity, it seems. This particular one is probably not the most artful, elegant, appealing, fanciful, decorative, or marketable version available, but its ability to do what it does with only 79 lines of code (about half JavaScript and half SVG) may help to illustrate the ease and brevity with which you can achieve rich effects. You can find an animated version of this (for browsers that support SMIL animation) at *http://srufaculty.sru.edu/david.dailey/svg/ballclock.svg*.

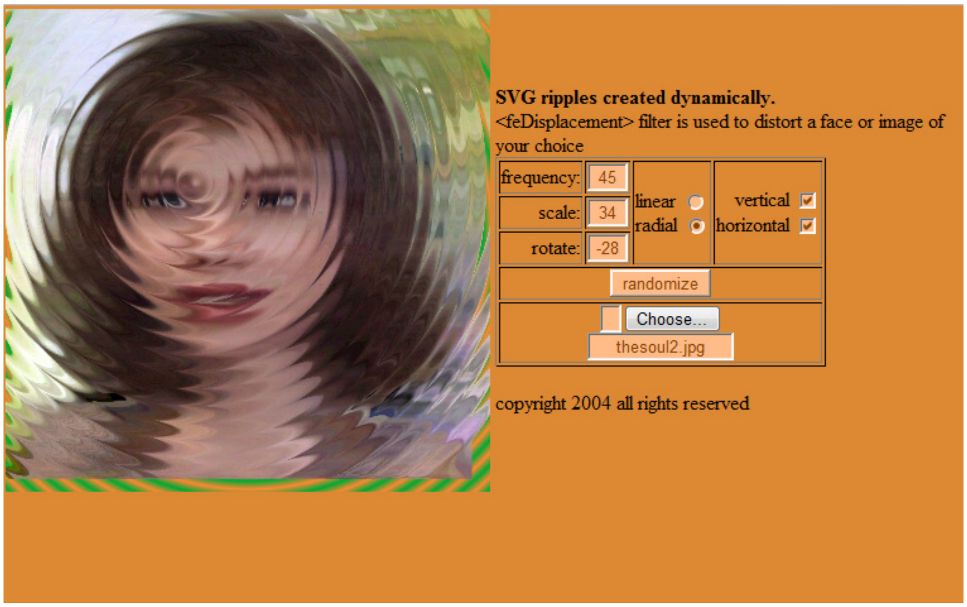


Here's how it's done:

- **The animation** All animation is handled declaratively (using SMIL). That is, there are no JavaScript statements involving *setTimeout()* or *setInterval()* (used for conventional web animation). A generic animation that handles the rotation of the clock's hands is declared in markup and then cloned through JavaScript, with its properties being modified in a simple loop that handles the details of how fast each hand should move. Likewise, the gears are each cloned from one protogear, with the dash patterns around their edges and their rotations being assigned different speeds.
- **The markup** The markup is kept minimal by using script to replicate many copies of similar things. SVG does not yet have a <*replicate*> element that might allow some of this script to be handled declaratively. In the meantime, we can use markup and script for what each does best—SVG allows the pleasant intermingling of both. The JavaScript is also used to assign colors, sizes, and speeds to the various gears, and to determine the actual time of day so the clock's hands may be initialized.
- **The clock face** The hour marks are also done declaratively by setting the *dash-array* attribute of the stroke around the clock face. The appearance of a slight curvature to the clock face is provided through a radial gradient.

Example 9: Using a Filter to Create Pond Ripples over an Image

This example shows some of the more advanced aspects of scripted animated gradients used in conjunction with filters to distort an image. The animated version shows ripples (customizable by the user using HTML input elements) moving across an image—much as ripples would disrupt the reflection of an image in a pond.



SVG ripples created dynamically.
<feDisplacement> filter is used to distort a face or image of your choice

frequency:	45	linear ○	vertical ✓
scale:	34	radial [✓ ●]	horizontal ✓
rotate:	-28		

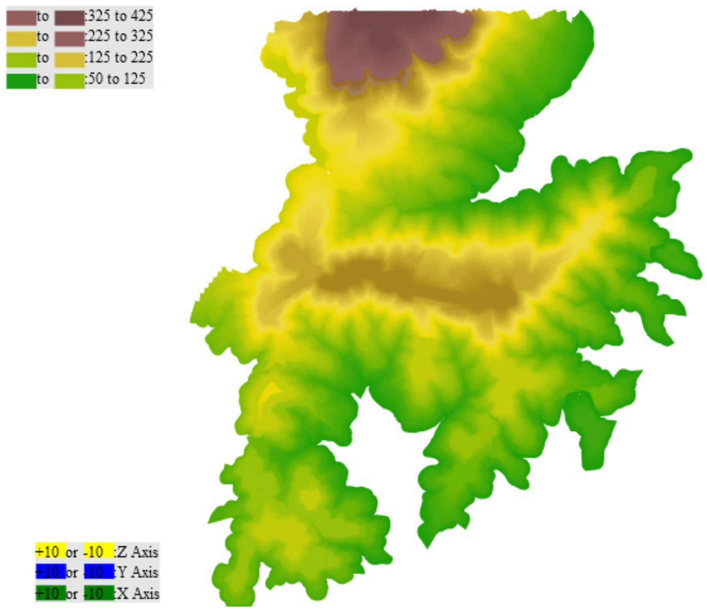
copyright 2004 all rights reserved

Here's how it's done:

- **Creating concentric circles** The circles are created with script. A gradient can have different color bands, called *stops*, defined within it. In this case, a series of concentric stops (orange and green) is created through script and added to a gradient, which is then applied to an ellipse under the image of a face.
- **Animating concentric circles** The radius (or offset) of each stop is then modified gradually through subtle changes in a *setTimeout* loop defined in JavaScript. Interestingly, the script for this example resides in the HTML rather than within the SVG, and the SVG DOM is accessed from there.
- **Distorting the image** Once the above two things have been done, the rest is rather easy. A filter is created that brings in both the concentric circles and the face as layers, and then distortion is applied through a filter effect known as *<feDisplacementMap>*, using the red channel of the gradient to determine the degree of distortion associated with the image. Because green doesn't contain red but orange does, *<feDisplacementMap>* provides the differential distortion in concentric bands.

Example 10: Using <replicate> to Simulate Digital Elevation Maps

SVG is still evolving. Version 2 of the specification is presently under deliberation by the W3C's Working Group. While the language currently has only two types of gradients (linear and radial), several proposals exist for increasing that number. One, the proposal to allow declarative markup to create many objects that are tweened from one another—like animation, only spatial rather than temporal—is to use *<replicate>*. While *<replicate>* would handle a wide variety of issues (such as this rotatable 3D portrayal of a geographic landform), other proposals are considerably less broad in scope.

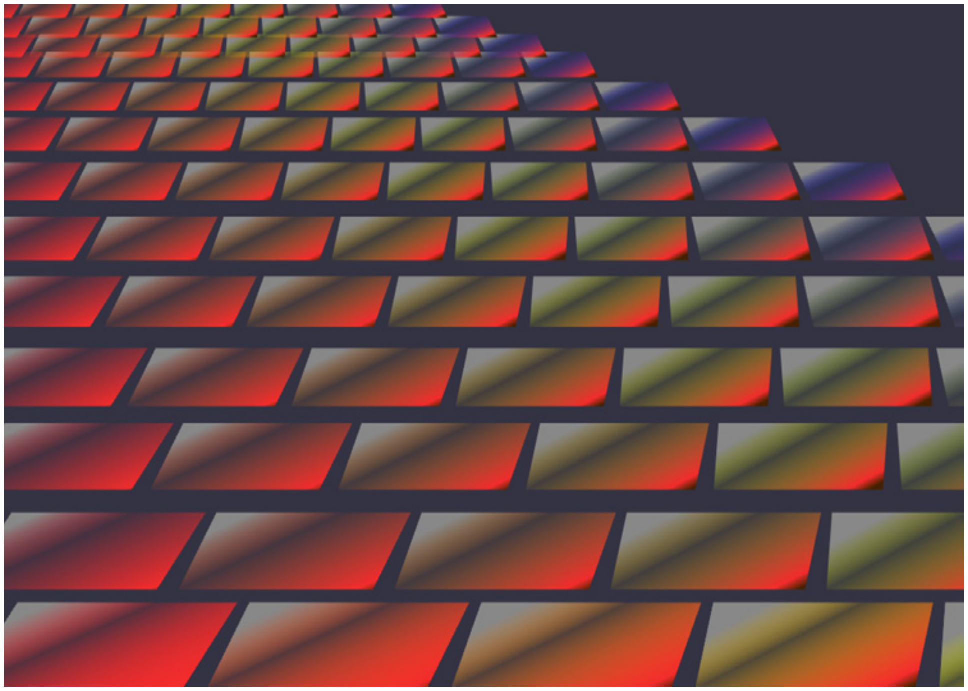


Here's how it's done:

- **Interpolating between paths** In this example, which you can find at *http://srufaculty.sru.edu/david.dailey/svg/dem/DEM_1.svg*, many concentric polygons (with varying numbers of points in their definition) are defined through interpolation and then cloned.
- **Simulating 3D rotation** Script then manipulates the data to enable rotation in three dimensions.

Example 11: Non-Affine Cobblestones

Here's another example showing the use of *<replicate>* (see *http://srufaculty.sru.edu/david.dailey/svg/replicate/repRectsGrad2g.svg*). This example replicates interpolated polygons in two directions. While *<replicate>* is not (yet) supported by the SVG specification, it is supported through an open source JavaScript initiative that allows SVG-like declarative markup to be interspersed with actual SVG to create a wide range of effects.

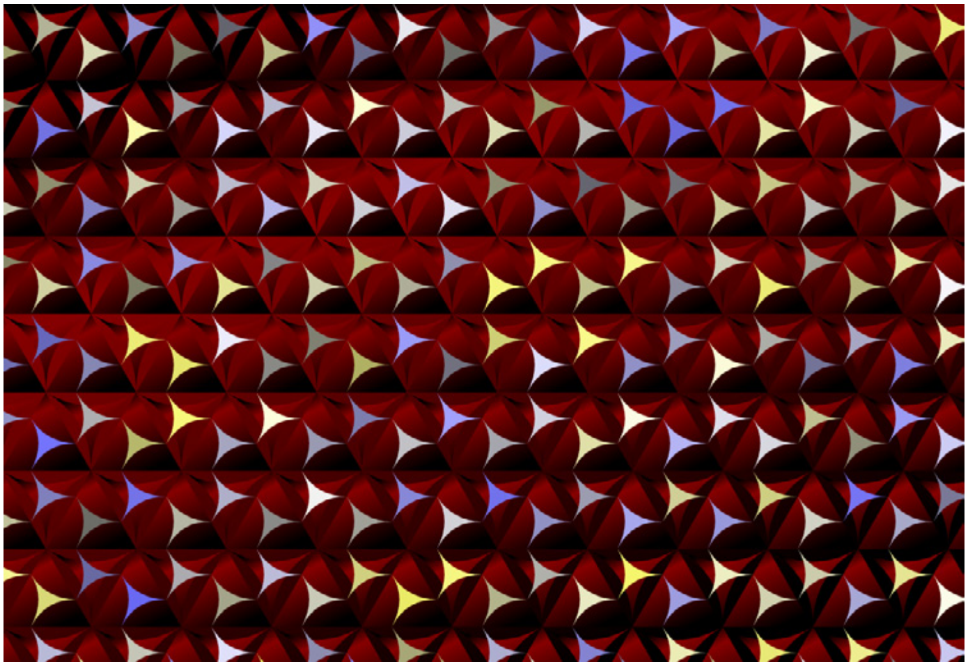


Here's how it's done:

- **The basic shape** First, a quadrilateral is drawn with SVG using a <*path*> element.
- **Replicating from left to right** It is then replicated by placing a <*replicate*> element inside the <*path*>. The <*replicate*> element instructs the quadrilateral's shape and position to be gradually duplicated from left to right. Additionally, the gradient applied to the quadrilateral is retrieved, and one of its defining color bands (or *stops*) is gradually changed from red to green, and finally to purple (with the color values of that gradient being modified as well).
- **Replicating vertically** The results of the first replication are viewed as part of a group that is then replicated upward, with its scale being modified as it is cloned.

Example 12: Triangular Tiling

While SVG has a <*pattern*> element, which allows the creation of repeated rectangular tiles, if you wish to use nonrectangular tilings or to individually modify the elements from one part of a pattern to another, then script may be the way to go. The juxtaposition of opacity, rotation, gradients, and triangles is something easily done in SVG. The rotation of the inner triangles creates a bivalent appearance of either clover leaves or honeycombs, depending on orientation. You can see this example at *http://srufaculty.sru.edu/david.dailey/svg/triBraids4.svg*.

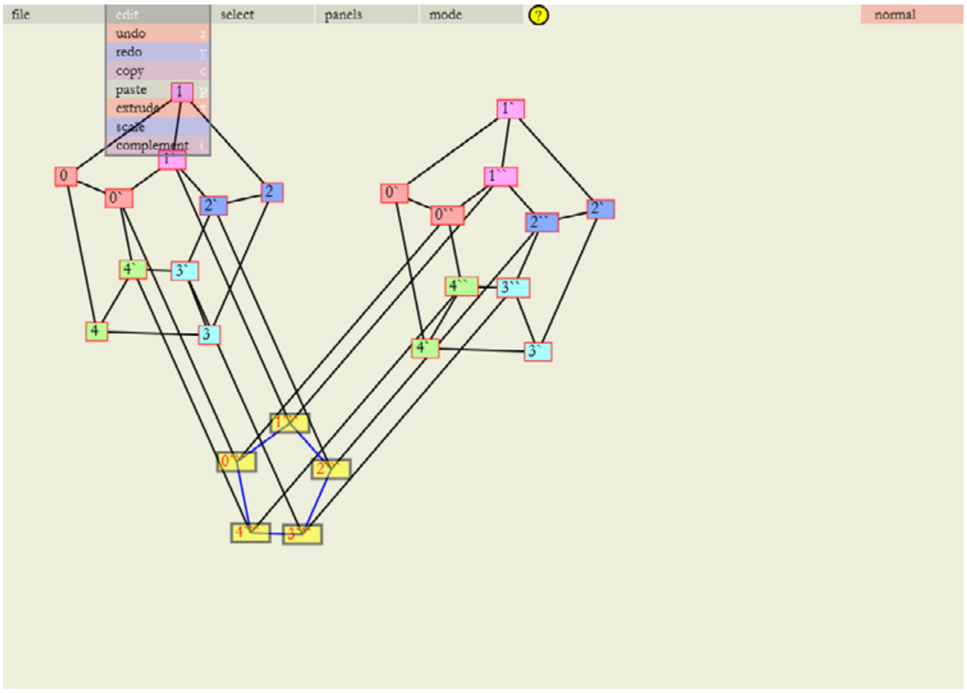


Here’s how it’s done:

- **The basic shapes** We begin with two triangles, having different orientations and gradient fills. Some opacity in the gradients is used to allow the background to be seen.
- **Duplication using script** In this case, script is used to build a triangular tiling through cloning of the initial triangles. The center of each triangle is then filled with another triangular shape filled with random colors (from a very select range of possibilities).
- **Finishing effects** An underlying gradient is applied and slowly animated to give an almost subliminal sense of “atmosphere.” For browsers that support SMIL animation, some of these effects, including rotation, are animated.

Example 13: A Web Application for Drawing Graphs (Networks)

This particular application has been built and rebuilt by its authors in many different languages (cT, HyperTalk, Java, VML, and now SVG with JavaScript) over the past 25 years. It has proven invaluable for the teaching of discrete mathematics to undergraduate students. Basically, using a few thousand lines of JavaScript, it builds a click-and-drag GUI interface to allow the creation, editing, replication, storage, and retrieval of finite graphs. Like many emerging web applications, the SVG here is sort of secondary, with JavaScript and event handling consuming the predominant effort. SVG can be used to play a crucial role in the increasingly important realm of web applications.



Here's how it's done:

- **Drawing** One advantage of using SVG—instead of comparatively lightweight graphical technologies, such as <*canvas*> in HTML—for building web applications is that objects in SVG are in the DOM. This means that events can be easily attached to objects and SVG event handling, much like those in the nongraphical parts of HTML. Thus, mouse coordinates (as well as the targets of events) can easily be interrogated to allow the creation and repositioning of objects, in the classical sense of a GUI.
- **Connecting** Again, because SVG objects are in the DOM, it is easy to build JavaScript referents to those objects so that arrays of objects and their properties may be maintained along with connections back to their visible instantiations. It is easy to connect and disconnect nodes of graphs, precisely because they are objects, both in JavaScript and SVG.
- **Interface** The example shown here uses JavaScript to build a menuing system along with dialog boxes, and the ability to export and import the user's drawings. However, much of this functionality can also be provided through higher-level tools, such as D3, and Pergola (discussed in later chapters).

Diving In: A Step-by-Step Approach to Building a Simple SVG Document

The following exercises are presented at a deliberately slow pace. Once you get the hang of how SVG works (in some general way), the pace will quicken a good deal.

A first file

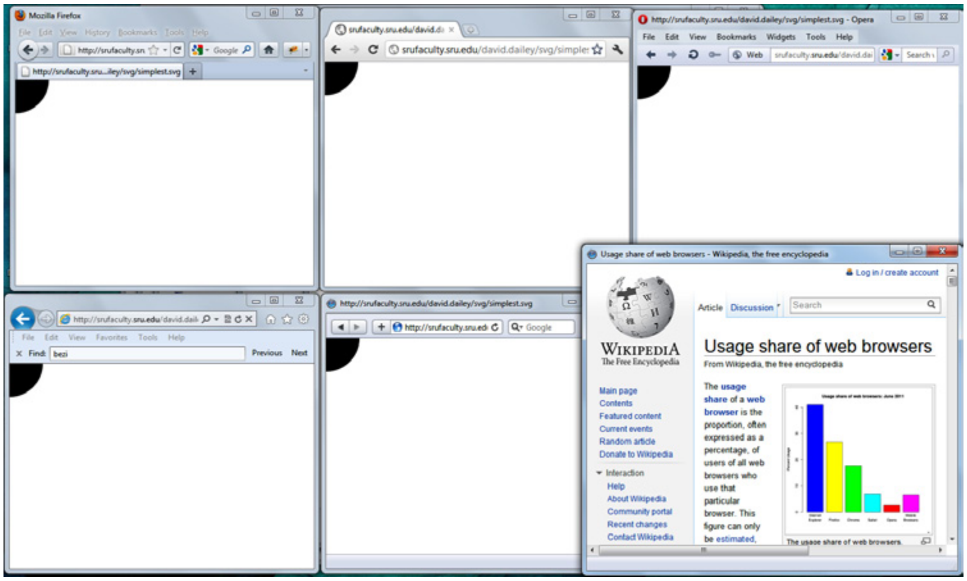
We already introduced a very simple example of an SVG file in this chapter, in the "Writing SVG" section. Let us recommence at that point:

1. Open a trusty text editor (something that allows you to see and save plain text—typically plain ASCII or UTF-8 in .txt format).
2. Create a file containing the following lines of code, and save it as *first.svg*:

```
<svg xmlns="http://www.w3.org/2000/svg">
  <circle r="50"/>
</svg>
```

3. Open the same file in a web browser. You can leave your text editor open because you may wish to later revise the file to add new things. For your browser, you may use a current version of Chrome, Firefox, Internet Explorer (see notes on this from the "Viewing SVG" section earlier in the chapter), Opera, or Safari.

   You should see something that looks like the image below, which shows screen shots of Firefox, Chrome, Opera, Internet Explorer, and Safari (from left to right, top to bottom).



4. If you wish to serve this file from your own server, then make sure that the server is serving its mime type as *Content-Type: "image/svg+xml"*. You may have to contact your systems administrator to make sure the server is properly configured. If problems arise, please refer to the document SVG MIME Type, at *http://planetsvg.com/tools/mime.php*.

Intermission and Analysis

Next, we'll discuss the code from the preceding exercise so you can see what it does.

```
<svg xmlns="http://www.w3.org/2000/svg">
  <circle r="50"/>
</svg>
```

SVG As XML

The first and last lines show that SVG, as an XML dialect, is a markup language. Each element—in the simplest case, a single word between the angle brackets—must have a beginning (the *<svg>* in this case) and an end (the *</svg>* in this example). You can end the tag like this:

```
<svg></svg>
```

Or you can end it like this:

```
  <circle ... />
```

This second example is called a self-terminating tag, because the slash (/) occurs at the end of the tag itself. Note that the second line in this example is indented as a convention for making the code more readable—the indentation isn't required.

Attributes

All SVG elements have a collection of attributes that are divided into two categories: regular attributes and presentation attributes (*http://www.w3.org/TR/SVG/attindex.html*). The first category includes, for example, geometrical attributes, such as *x*, *cx*, and *width*. The second category includes, for example, paint attributes, such as *fill*, *stroke-width*, *display*, and *opacity*.

The *<circle>* element, for example, has an attribute *r* (meaning *radius*). The fact that the *r* attribute has a value of 50 means (in the simplest and standard case) that the circle's radius will be 50 pixels.

The SVG Namespace

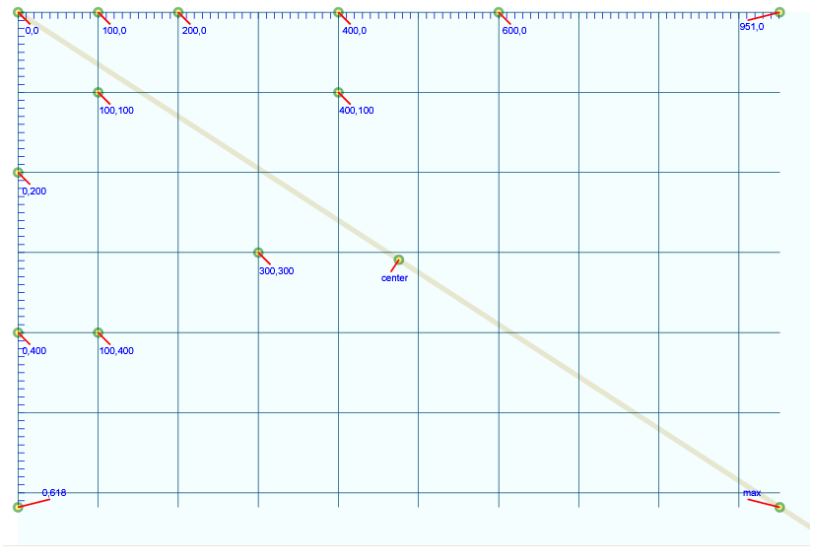
The *<svg>* element has the attribute/value pair *xmlns="http://www.w3.org/2000/svg"* (meaning that the XML namespace used to interpret the document will be one specified by the W3C).

The *xmlns* attribute (which appears not to have been a part of the language originally, because the Adobe and Opera viewers are unique in not requiring it) is necessary for most browsers to be able to display the code as SVG.

Essentially, the *xmlns* attribute merely tells the browser that it will be speaking a new dialect of XML. This is because most browsers of the 20th century assumed that the only language they would need to know was HTML. Writing <*svg*> isn't sufficient to let the browser know this, because the XML specification requires a namespace. It is rather unfortunate, from the perspective of teachers and learners, that the computer languages we learn are filled with mysteries that have no apparent purpose until one becomes a guru. However, you can think of the code *xmlns="http://www.w3.org/2000/svg"* within the <*svg*> element as just that: a mysterious incantation probably placed in the language to make sure that casual learners know to be on their guard. It turns out that it is not all that important to understand.

Screen Coordinates

Before beginning the second exercise, in which you'll begin experimenting with SVG, consider the drawing space itself—the browser window. Each point within the drawing space (also known as the Cartesian plane) is identified by a pair of coordinates (x and y). The upper-left corner of the screen is the point (0,0) and—depending on screen resolution and the current size of the window—the lower-right corner could have coordinates such as (800,640), (951,651), or (1440,900). The number of pixels determines the resolution of the screen. The resolution on mobile devices varies considerably; 240×320 pixels is a popular size for smaller and older devices.

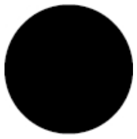


Modifying your code and experimenting

In this exercise, you'll experiment with the circle you drew in the previous exercise by changing its location, size, and color, rebuilding it so that only its outer boundary remains black.

1. Move the circle to the center of the screen. You do this by setting the x and y coordinates of the center of the circle (*cx* and *cy*, respectively) to 50 percent, which is measured relative to the width and height of the browser window.

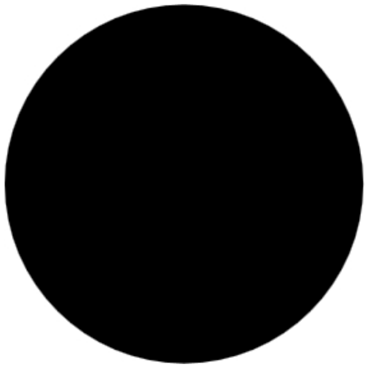
```
 <svg xmlns="http://www.w3.org/2000/svg">
  <circle r="50" cx="50%" cy="50%"/>
</svg>
```



2. Increase the radius and set it as a fixed proportion of the browser's width.

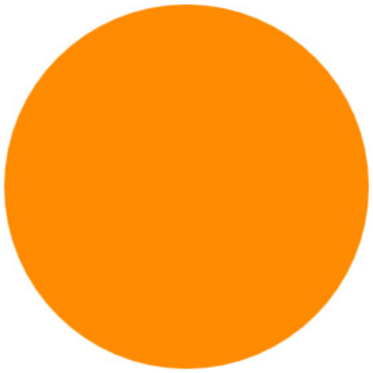
Note A geometric attribute, such as *cx*, *cy*, or *r* in this case, can be set as either a proportional value (relative to window size) or an absolute value (pixels, by default).

```
<svg xmlns="http://www.w3.org/2000/svg">
  <circle r="25%" cx="50%" cy="50%"/>
</svg>
```



3. Change its color. You can do this by setting the *fill* attribute to a named color, or in a variety of other ways (e.g., using CSS or HTML hexadecimal values, RGB values, or HSB values).

```
<svg xmlns="http://www.w3.org/2000/svg">
  <circle r="25%" cx="50%" cy="50%" fill="darkorange"/>
</svg>
```

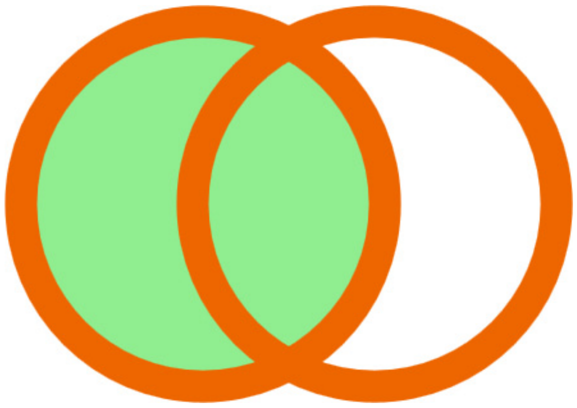


4. Change the code so that just the outside of the circle is colored. This actually involves three tasks: setting the fill of the circle to *none* so that its interior is transparent, setting its stroke to some color (e.g., *darkorange*), and defining a width for the stroke. The code below also adjusts the color from the named color *darkorange* to *#e60*, which is a bit lower on the red channel and a good bit lower on the green channel than the *darkorange* hexadecimal equivalent, #FF8C00.

```
<svg xmlns="http://www.w3.org/2000/svg">
  <circle r="25%" cx="50%" cy="50%" fill="none" stroke="#e60" stroke-width="25"/>
</svg>
```

5. Make the inside transparent. You can accomplish this by putting another opaque circle behind it and scooting it a bit to the left. Note that the first object defined appears behind objects that appear later in the document tree. In this case, we've also added another namespace identifier (which is not strictly needed here, but will become necessary for elements that use the SVG linking facilities to link to external documents or code fragments defined elsewhere within the same document). It's here so you can become accustomed to seeing it, because it's part of the standard declaration of a typical SVG document.

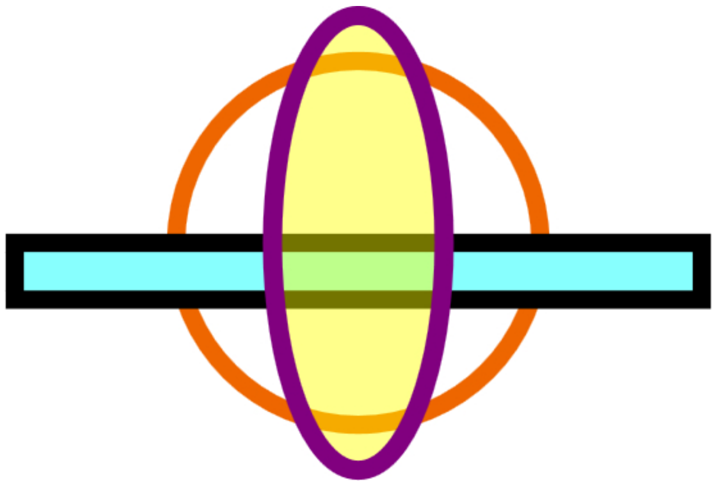
```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink" >
  <circle cx="30%" cy="50%" r="25%"
          fill="lightgreen" stroke="#e60" stroke-width="25"/>
  <circle cx="50%" cy="50%" r="25%"
          fill="none" stroke="#e60" stroke-width="25"/>
</svg>
```



Accomplishing a given effect

In this exercise, we will present a picture and ask you to analyze and then try to draw it.

1. Observe the following SVG drawing, referred to as "the objective":



2. Identify the type of objects that seem to be used in the drawing.

Tip Until we introduce the full range of graphical primitives, we will restrict the drawing to circles, ellipses, and rectangles, all of which are somewhat self-explanatory once you see the syntax.

In this case, it appears that there are three objects: a circle (that very much resembles our earlier one), an oval (called an ellipse in SVG), and a rectangle.

3. Identify the order in which the objects are drawn. The front-most object appears to be the ellipse, and its yellowish fill pattern appears slightly transparent, because you can see through it to the objects behind it. From back to front (which coincides with the order in which the objects will be drawn), there appears to be a circle, then a rectangle, and finally an ellipse. It's important to note that the topmost object (the ellipse) is transparent in its interior but not its boundary.

4. Determine whether the objects seem to be drawn using relative values (percentages) or absolute values (pixels) for their geometric attributes.

   You may find it useful to view the drawing on the web, where you can see how the drawing is affected by resizing the browser. You can find the drawing here: *http://granite.sru.edu/~ddailey/svg/lesson3.svg*.

   In this case, it makes sense to begin with the assumption that the geometry has been drawn relative to the window size for three reasons: because the circle seems to be the same as in the second exercise (which used relative values for its geometry), because the rectangle's top line seems to coincide with the center of the circle, and because the ellipse appears to share the same center as the circle.

5. Start off with the same file you created at the end of the previous exercise, because it appears that the two files share the same circle, and the objective illustration would involve placing that circle beneath the other objects—which means earlier in the markup code.

```
<svg xmlns="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink" >
  <title>Collage involving &lt;rect&gt; , &lt;circle&gt; and &lt;ellipse&gt; </title>
  <circle cx="50%" cy="50%" r="25%" fill="none" stroke="#e60" stroke-width="25"/>
</svg>
```

   The preceding code contains one additional line: a *<title>* element. As an image format, SVG has great potential to address issues of accessibility for visually impaired people, so it is best to get in the habit of adding a title to all your documents. You'll see more about accessibility later in this book, because it is an important topic, particularly for SVG.

6. Add a light-blue rectangle on top of the circle and adjust its size, position, color, stroke, and stroke width:

```
<svg xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" >
  <title>Collage involving &lt;rect&gt; , &lt;circle&gt; and &lt;ellipse&gt; </title>
  <circle cx="50%" cy="50%" r="25%" fill="none"
    stroke="#e60" stroke-width="25"/>
  <rect x="10%" width="80%" y="50%" height="10%"
    fill="#8ff" stroke="black" stroke-width="6" />
</svg>
```

The <rect> element, like the <*circle*>, can have *fill*, *stroke*, *stroke-width*, and other attributes. The *x* and *y* attributes specify the rectangle's upper-left corner, and *height* and *width* specify its size. Because you want the top of the rectangle to coincide with the center of the window, you can set *x* to "50%". You also want it centered on the screen horizontally, so the distance of its rightmost extent (specified by *x* + *width*) to the right edge of the window should equal *x*. By experimenting a bit with different values of *x* and the corresponding value of *width* (determined by the centering constraint), you can visually estimate the values above (or similar values). Note that 100% – (80% + 10%) = 10%, which means that the rectangle will be centered horizontally, even though it is not centered vertically. The values for *stroke-width* and *fill* can likewise be estimated through experimentation.

7. Put an oval atop everything and fill it with a transparent shade of yellow, while keeping its stroke opaque:

```
<svg xmlns="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink" >
  <title>Collage involving &lt;rect&gt; , &lt;circle&gt; and &lt;ellipse&gt; </title>
  <circle cx="50%" cy="50%" r="25%" fill="none" stroke="#e60" stroke-width="25"/>
  <rect x="10%" width="80%" y="50%" height="10%" fill="#8ff"
    stroke="black" stroke-width="6" />
  <ellipse cx="50%" cy="50%" rx="10%" ry="40%" fill="yellow" fill-opacity=".45"
    stroke="purple" stroke-width="15" />
</svg>
```

An ellipse, like a circle, has a center defined by *cx* and *cy*. However, owing to the difference in its vertical and horizontal extents, it has two radii: *ry* and *rx*, respectively. Because this oval is taller than it is wide, you can approximate the values above fairly closely by testing a few values and seeing what happens. Alternatively, you could actually measure the drawing on the screen to duplicate the effect more precisely.

The preceding code introduces a new attribute: *opacity*. All the typical drawn objects (such as *rect*, *circle*, *polygon*, *ellipse*, and *path*) all have an *opacity* attribute. When *opacity* is set to “1.0”, an object’s stroke and fill are completely opaque. When *opacity* is set to “0.0”, the object is completely invisible. If you don’t specify *opacity*, the browser assumes that *opacity* is 1. If you wish to specify different levels of opacity for an object’s stroke and fill, you can do so using the attributes *stroke-opacity* and *fill-opacity*.

The code you end up with for this exercise should closely match the code of the example at *http://granite.sru.edu/~ddailey/svg/lesson3.svg*.

Summary

With this chapter, we hope to have given you a sense of how useful, elegant, and important SVG is for building informative and appealing graphics. You can accomplish a broad range of effects with this technology, ranging from practical to artistic, while making your graphics both dynamic and interactive. SVG is a powerful technology, and yet it allows you to easily begin the process of experimenting and learning. We feel it is a valuable technology that is just beginning its ascent to widespread deployment.

CHAPTER 2

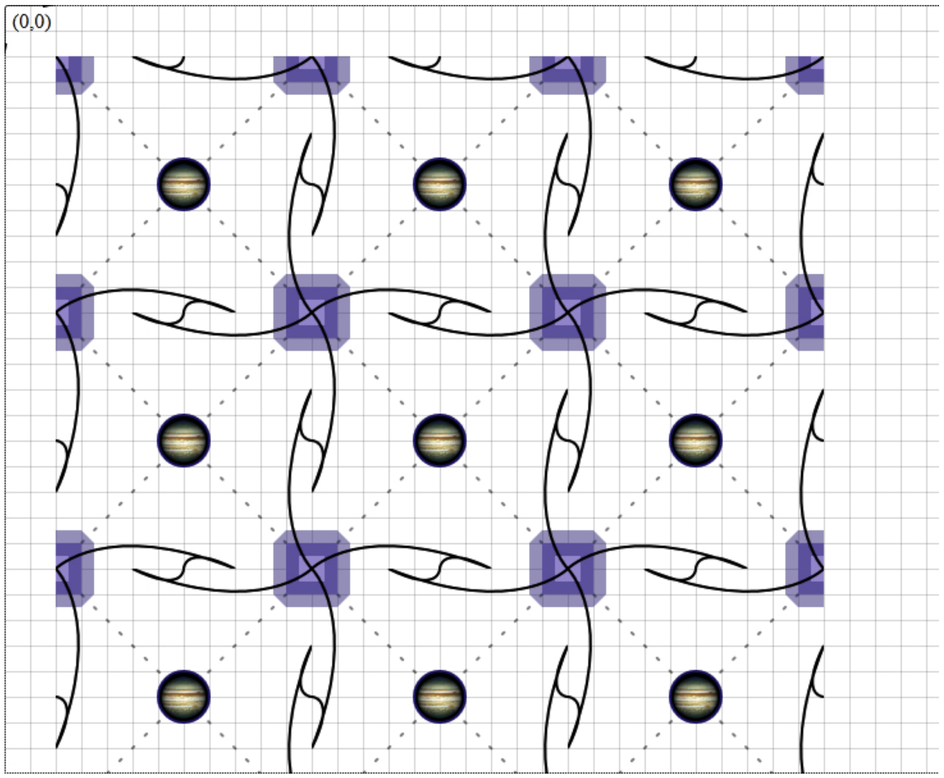
Creating and Editing SVG Graphics

Order becomes beauty
beyond infinite planes
and the undeciphered dense text
a mosaic flower, fiery,
chaos tamed in fullness,
spring.

Orides Fontela

In this chapter:

By the end of this chapter, you will have explored the core concepts and practiced the basic skills to begin tapping into your visual creativity. One great thing about programming graphics is that you can usually visualize your work almost immediately. To demonstrate this, you'll walk through a process that uses all of the basic shape elements of SVG. As a teaser, here's a look at one of the graphics that you will build in this chapter.



This graphic incorporates all of the basic shapes, the simple Bézier curve, more complex cubic Bézier curves, and bitmap images. It also demonstrates the logical grouping and reusing of related graphics, and finally, how to pull everything together into a reusable tiling pattern, which is also known as *tessellation of the plane*.

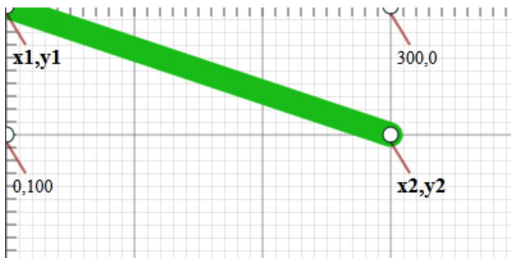
Note Although mathematical functions underlie the creation of SVG, and getting the most out of SVG requires a decent grasp of mathematical concepts, those of us who have limited mathematical talents can still harness the power and creative potential of SVG.

Creating Basic Vector Shapes

To get started, we'll go over the six basic shape elements: *<line>*, *<rect>*, *<circle>*, *<ellipse>*, *<polyline>*, and *<polygon>*.

Lines

To create a visible line in SVG, simply set the *x2* and *y2* values of the *<line>* element. You can set the line's color and other properties as well using the stroke-related attributes.



```
<line x2="300" y2="100" stroke="green" stroke-width="10" stroke-linecap="round" />
```

Note By default, most SVG shape properties have initial or default values. For example, the initial value of most positioning properties is zero, which is why you do not have to specify the *x1* and *y1* values for the *<line>* element. Also, the default fill color for shapes is black, so the shape *<circle r="50" /> or <polygon points="850,75 850,325 742,262 742,137" />* will appear black even though the fill has not been specified.

Brief Review of SVG Presentation Attributes

Besides the command attributes that define a shape's position, radius, width, and height, SVG also has many attributes that define a shape's style. You are probably already familiar with attributes such as *display, visibility, font,* and *letter-spacing*. SVG also has many SVG-specific styling properties (as in the example above, which shows how the *stroke* attribute allows you to define the color of the line).

SVG presentation attributes can help you quickly set the paint and geometrical values of SVG elements; apply gradients, filters, and clipping; and control the interactive behavior. Chapter 3, "Adding Text, Style, and Transforms" covers presentation attributes in more depth, but Table 2-1 provides a quick reference for common properties that you will be using in this chapter.

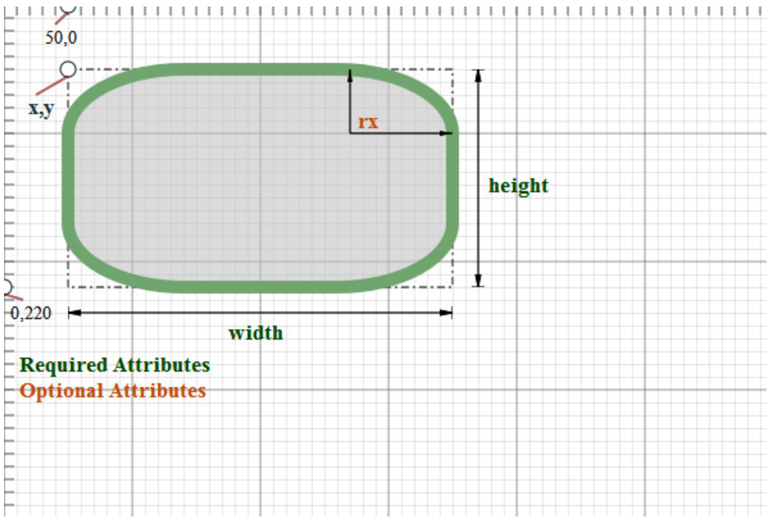
TABLE 2-1 Common SVG Presentation Attributes

Attribute	Values
stroke	This specifies the color of the stroke. The valid color values are the same as in CSS3 and HTML5: named color (e.g., "blue"), hexadecimal (e.g., "#f34a12"), RGB (e.g., "rgb(255,255,255)"), HSL (e.g., "rgb(100%,50%,90%)"),%), and so on. More detail about SVG colors can be found here: *http://www.w3.org/TR/SVG/color.html*.
stroke-width	This specifies the width of the stroke for a shape or text using either a percentage or a length value. When using a length value, we recommend specifying the type of unit (px, cm, etc.) to prevent cross-browser issues. It is worth pointing out that the units specified in the outermost *<svg>* tag are inherited by all descendants, and that the default value is *px*. You can find more details about possible length values here: *http://www.w3.org/TR/SVG/types.html#DataTypeLength*. Note that the stroke is centered on the edge of a shape, so if *stroke-width* is set to a large enough value, the shape's fill may not even display.
stroke-opacity	This is a number between 1.0 and 0.0. A value of 1 makes the stroke entirely opaque and 0 makes it invisible.
stroke-dasharray	This is a list of user coordinate values (px) that determines the length or pattern of the invisible spacing to be drawn between segments along the stroke of text or a shape.

Attribute	Values
stroke-linecap	This defines the shape at both ends of a line. The options are *butt* (the default), *round*, and *square*.
stroke-linejoin	This determines the shape to be used at the corners of paths or basic shapes. The options are *miter* (the default), *round*, and *bevel*.
fill	This specifies the color of the shape or text.
fill-opacity	This is similar to the stroke opacity. Note that if the opacity is between 0 and 1, and the stroke value is set to a different color or opacity than the fill color, then the inner portion of the stroke will be a different color than the outer portion of the stroke, which can create some nice effects.
fill-rule	This determines which portions of a shape will be filled. The options are *nonzero* (the default) and *evenodd*. Note that this is usually straightforward, but for more interesting or complex shapes, the result of *fill-rule* can be less obvious, as explained in the "Fill Properties: nonzero and evenodd" section.

Rectangles

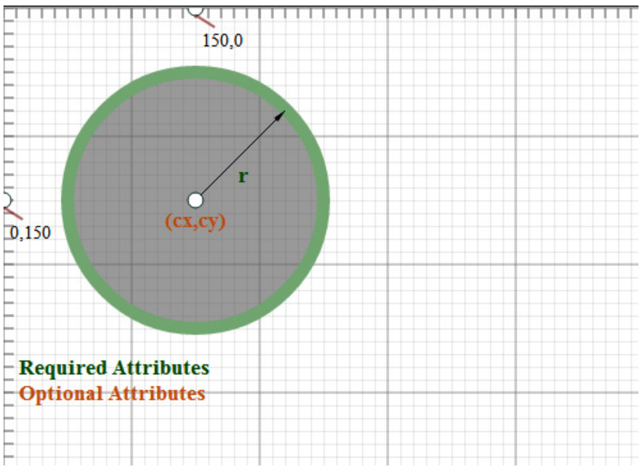
The rectangle element (<*rect*>) requires *width* and *height* attributes, but you can also specify *x* and *y* attributes, which specify the position in relation to the top-left corner of the SVG canvas. If they are not specified, they default to (0,0). Optional *rx* and *ry* attributes are also available, which apply a uniform rounding to all the corners. If only *rx* is specified, *ry* is equal to *rx*.



```
<rect x="50" y="50" width="300" height="170" rx="90" ry="50"
      stroke="darkseagreen" stroke-width="10"
      fill="lightgray" fill-opacity="0.6" />
```

Circles

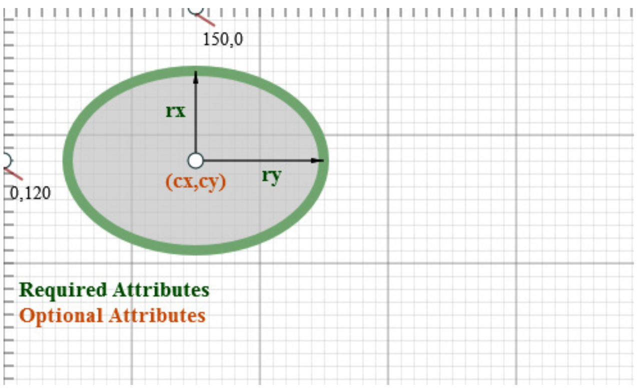
As mentioned in Chapter 1, "Stepping into SVG," the SVG <*circle*> element only requires a value for the radius. In the following image, the *cx* and *cy* values are set to (100,50).



```
<circle cx="150" cy="150" r="100"
        stroke="darkseagreen" stroke-width="10" fill="grey" fill-opacity="0.6"/>
```

Ellipses

The *<ellipse>* element provides the additional attribute *ry* so that both the x and y radius values can be set as shown below:



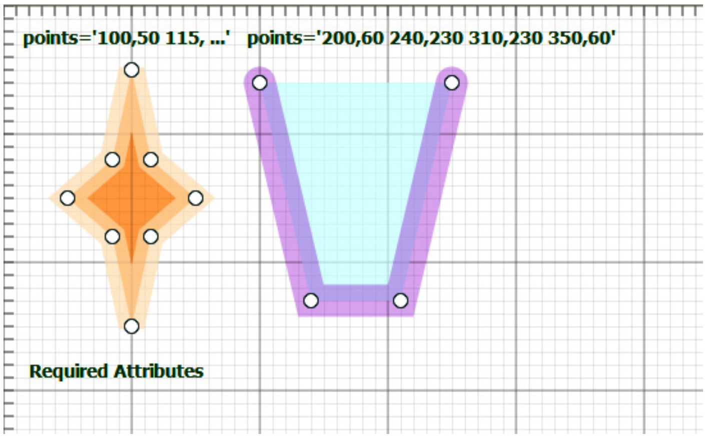
```
<ellipse cx="110" cy="55" rx="70" ry="35"
         stroke="darkseagreen" stroke-width="0.8"
  fill="lightgray" fill-opacity="0.6" />
```

Polylines and Polygons

There are just two additional basic shapes: the polyline and the polygon. They are very similar to each other in that they both simply require the *points* attribute, which contains a list of x,y value pairs. Both of these shapes allow for drawing a series of straight lines, as if a pen were set down and used to draw on paper.

The primary difference between the *<polyline>* and *<polygon>* shape elements in SVG is that the polyline path will not be closed by default—that is, the two endpoints will not be connected unless you specify that they should be. If you wish a polyline shape to be closed, you need to specifically draw an endpoint that meets back up with the starting point. The polygon, on the other hand, will automatically close the shape from the last specified point, as shown in this example:

```
// open
<polyline points="200,60 240,230 310,230 350,60"
          fill="lightcyan" fill-opacity="0.7"  stroke="darkviolet"
          stroke-width="25" stroke-linecap="round" stroke-opacity="0.2" />

// closed
<polygon points="100,50 115,120 150,150 115,180 100,250 85,180 50,150 85,120"
         fill="darkorange" fill-opacity="0.5" stroke="papayawhip"
         stroke-width="20" stroke-opacity="0.7" stroke-linejoin="miter"/>
```



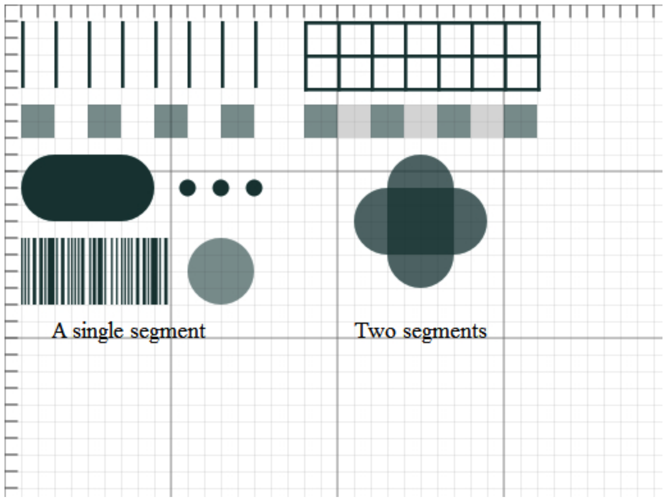
Note how the *stroke-linejoin* and *stroke-linecap* attributes affect the shape.

Note All of these basic shapes have been purposely designed by the W3C community for ease of use, and each type of shape element carries an inherent semantic meaning as well. As discussed in the previous chapter, there are many benefits of semantic languages, and the well-defined shape elements of SVG have inherent benefits for projects such as mapping, CAD, and graphic design.

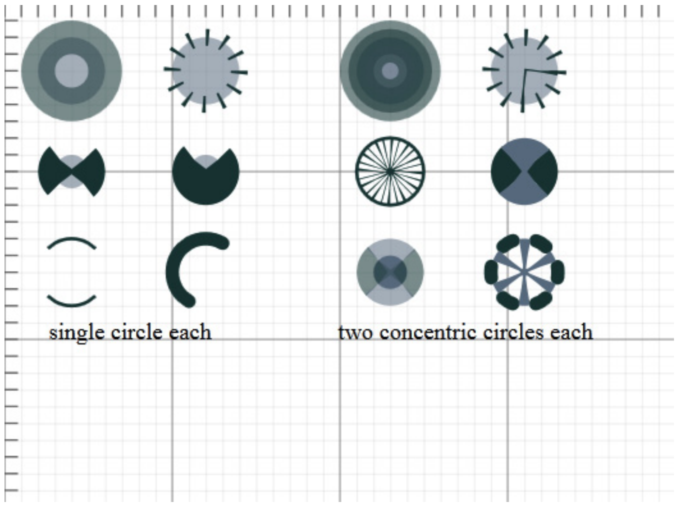
Creativity with Basic Shapes

The beauty of the SVG language is that with just this basic knowledge, you are already able to start building some complex vector graphics that will render in all the major browsers. As an example, this next image demonstrates some of the fancy things that you can already do with a little creativity and knowledge of SVG shape properties.

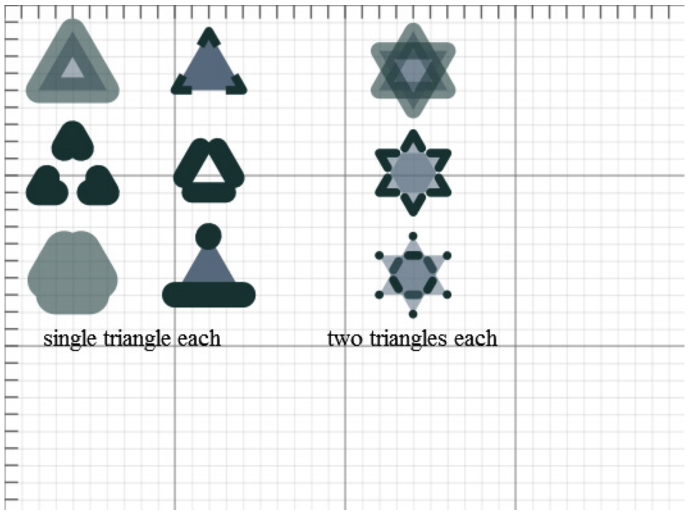
This next example shows how the <*line*> element can be styled, with surprising results. All of the shapes on the left of the figure were created with a single <*line*> element, and all of the shapes on the right were created with just two <*line*> elements.



Even more interesting, the following minimalistic example demonstrates how to create fancy circular shapes using just one or two <*circle*> elements.



This third example uses only one or two <*polygon*> elements—again with interesting results:



I encourage you to open the code samples (see the Introduction for instructions on downloading the code samples) that come with this book to better understand how these interesting shapes were created. This will provide you with valuable insights into the workings of presentation attributes such as *stroke*, *opacity*, *dash array*, and others.

In addition to the basic shape elements, SVG provides the much more expressive <*path*> element, which allows you to create any type of two-dimensional shape.

Paths in SVG

The <*path*> element is the most flexible drawing primitive in SVG. It contains subcommands that allow it to mimic all of the other basic shapes. As such, it is a bit trickier to learn.

Like other drawing primitives such as <*rect*> and <*ellipse*>, <*path*> can take attributes such as *fill*, *stroke*, and *dash array*. On the other hand, <*path*> uses a special syntax to describe the way it actually visits points on a plane. It borrows some of its origin (at least ideologically) from *turtle graphics* (*http://en.wikipedia.org/wiki/Turtle_graphics*), which are used in the Logo programming language to help introduce younger children to the basics of computer programming.

The SVG <*path*> element is very expressive due to the range of powerful path commands that it uses. As with the HTML5 <*canvas*> element, paths can be used to draw pen-up and pen-down movements, quadratic and cubic Bézier curves, and elliptical arcs, all within a single path. That is, you move the pen (or drawing point) from position to position, raise it and lower it, and make strokes of varying types. These instructions within the <*path*> syntax are called *subcommands* of the path object. In SVG, you'll find them in the data attribute (*d*) of the <*path*>.

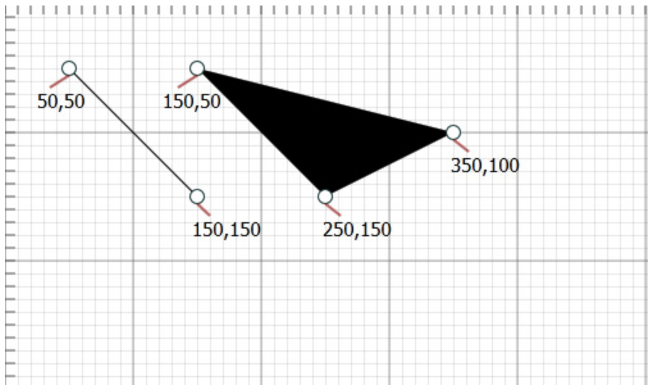
Paths typically begin with the *M* subcommand, which instructs the drawing to begin at a specific (x,y) point, such as (100,100), like so:

```
d = "M 100,100 ..."
```

From there, you continue adding points—that is, (x,y) pairs—describing segments to be joined along the path. The following section shows how this works.

<path> Subcommands: M and L

Start by specifying where the drawing will begin. As the first command for the *d* attribute, you insert a notation such as *M x y*, where *x* and *y* are numbers. You can think of *M x y* as meaning "Move the pen to the coordinates (x,y)." From there, you have the option of drawing a line (*L*), a quadratic curve (*Q*), a cubic curve (*C*), or an arc (*A*). For example, *d="M 50 50 L 150 150"* would draw a diagonal line from the point (50,50) to the point (150,150).



```
<path stroke="black"
      d="M 50 50 L 150 150"/>
<path d="M 150 50
      L 250 150 350 100"/>
```

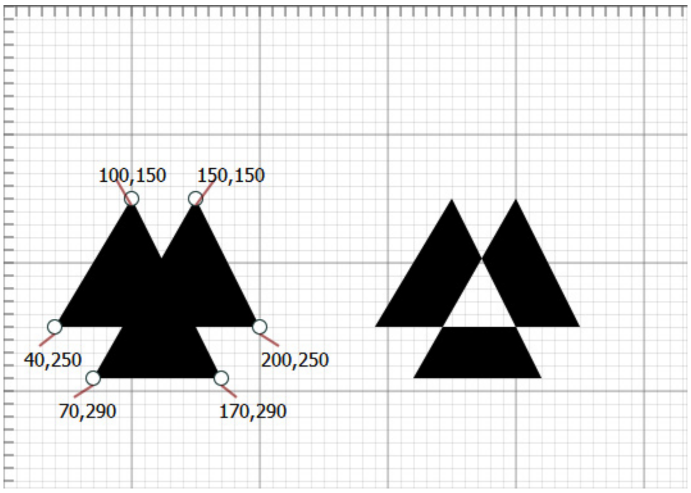
You should note several things about this example:

- The second path does not specify a stroke; by default, the figure is filled with black. If you specify *fill*="*none*", the figure will be invisible unless you specify a stroke.
- You can, for the sake of legibility, use commas between pairs of coordinates, in which case the space after the comma is optional.
- You can omit the command letter on subsequent commands if the same command is used multiple times in a row, as shown in the second path, where the *L* command is followed by two pairs of values. Note also that if a MoveTo command (*M* or *m*) is directly followed by multiple pairs of coordinates, the subsequent pairs are treated as implicit LineTo commands.

Fill Properties: nonzero and evenodd

Since a path is filled with black by default, it is natural to wonder what happens when a path crosses itself. As mentioned in Table 2-1, the default *fill-rule* value is *nonzero*, which means that by default, the union of the regions traversed by the path is filled unless you specify otherwise. You can find more information on this in the "Fill Properties" section of the SVG specification, at *http://www.w3.org/TR/SVG/painting.html#FillProperties*.

Here is an example to show the difference between the *fill-rule* values *nonzero* and *evenodd*.



```
<path d="M 70,290 L 150,150 200,250 40,250 100,150 170,290"/>
<path d="M 70,290 L 150,150 200,250 40,250 100,150 170,290"
      fill-rule="evenodd" transform="translate(250,0)"/>
```

This example demonstrates the default fill technique, as well as the *evenodd* fill rule on a shape that intersects itself in more than one place.

Note To demonstrate how the *fill-rule* attribute rule works in this example, we moved the second path shape, which has *fill-rule*="*evenodd*" applied to it, 250 units along the x-axis through the use of the *translate* method of the *transform* attribute. You will learn more about the *transform* capabilities in Chapter 3.

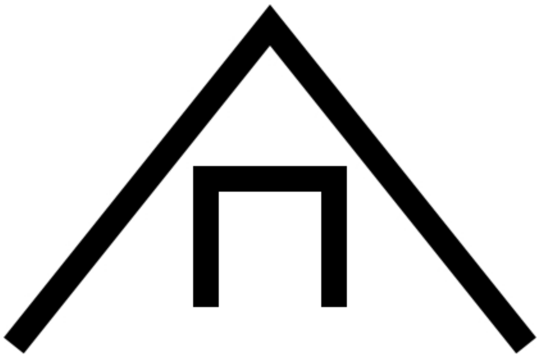
An Example of Building Complex Shapes

This section shows how you can use the pen-down command *M* to make more complex shapes with *<path>*.

The following code creates two paths, with one apparently drawn inside the other (in the sense that the coordinates of one are contained inside the polygon defined by the other):

```
<path d="M 100,350 300,100 500,350" fill="none" stroke="black" stroke-width="20"/>
<path d="M 250,320 250,220 350,220 350,320" fill="none" stroke="black" stroke-width="20"/>
```

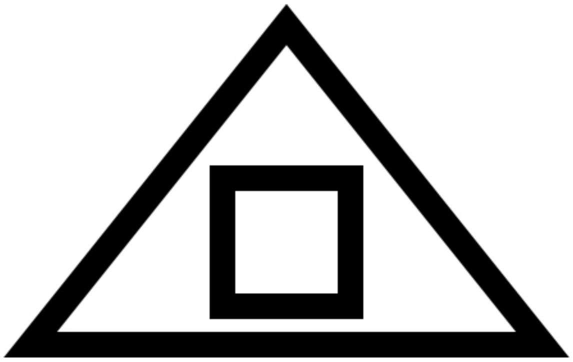
The figure contains two paths: one with three points, the other with four. Note how the triangle encompasses the rectangle.



Next, we add the simple *z* subcommand (shown below in bold) at the end of each of the strings, which closes the path by drawing a final line back to the path's starting point. After we do that, the paths will be closed rather than left open between endpoints.

```
<path d="M 100,350 300,100 500,350 z" fill="none" stroke="black" stroke-width="20"/>
<path d="M 250,320 250,220 350,220 350,320 z" fill="none" stroke="black" stroke-width="20"/>
```

The new rendered SVG image looks like this:



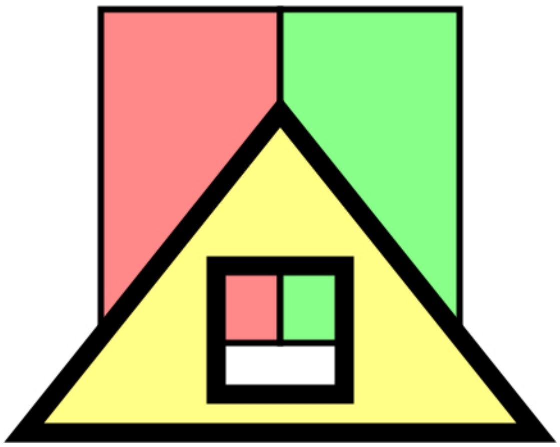
Alternatively, you could create the preceding image using only a single *path* object (see *http://www.w3techcourses.com/svg_images/onepath.svg*), as follows:

```
<path d="M 100,350 300,100 500,350 z
               M 250,320 250,220 350,220 350,320 z"
          fill="none" stroke="black" stroke-width="20"/>
```

This method can save a bit on markup, but is a little harder. However, there are some additional benefits to this approach that are worth considering. By combining the two shapes above into one compound path, you can define the fill rule of that path as *evenodd*. The net effect of this is that the shape's fill color will not be applied to the interior region (although it would be applied to regions inside the interior region). Try this combined code:

```
<path d="M 100,350 300,100 500,350 z
M 250,320 250,220 350,220 350,320 z"
  fill="#ff8" stroke="black" stroke-width="15" fill-rule="evenodd"/>
```

You can see the advantage in the next graphic. The rectangles that underlie the triangle are visible through the rectangular hole in the shape. This effect would be difficult to produce if the two parts of this compound path were separate paths, because to be visible, the rectangle would have to be on top of the triangle—but in that case, nothing inside it other than the triangle itself would be visible.



We have just demonstrated how to create a complex vector graphic shape using a single SVG path element that contains a yellow triangle with a rectangular hole showing pink and green rectangles underneath. The next section discusses creating shapes using Bézier curves.

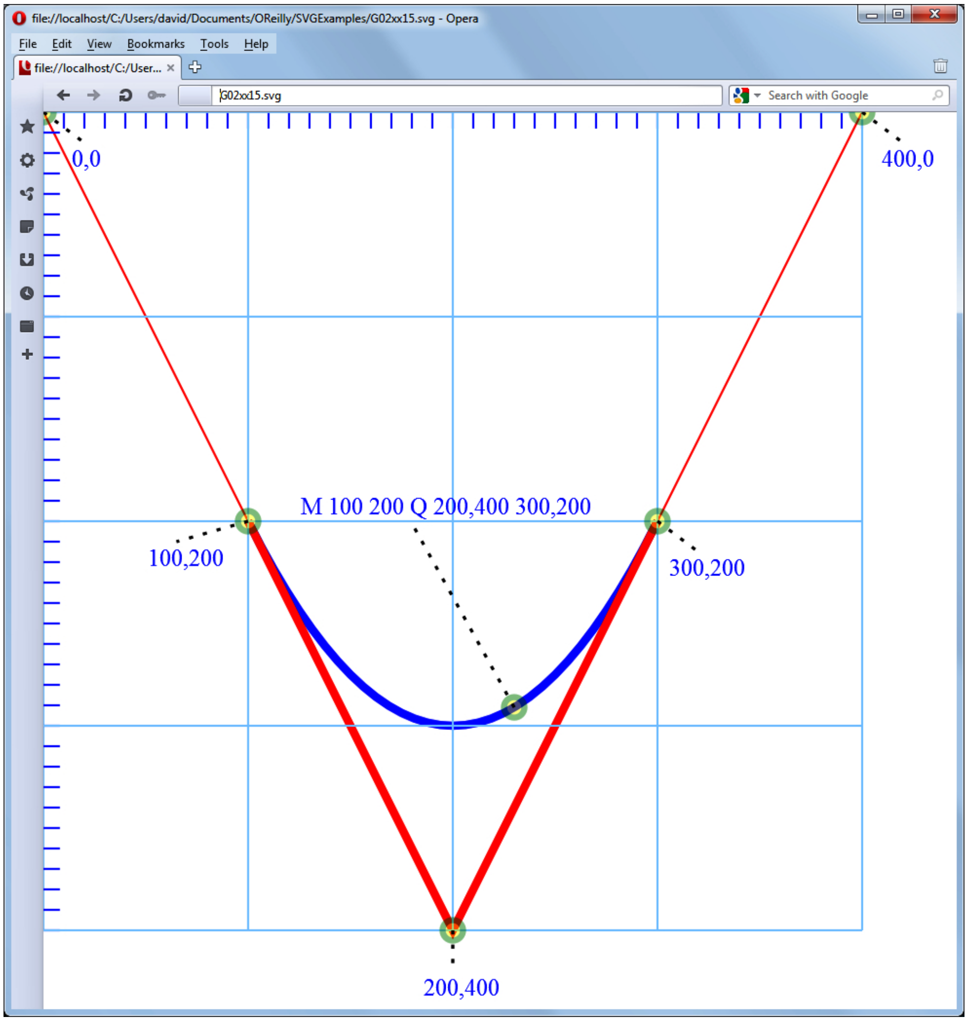
Quadratic Bézier Curves: The Q Subcommand

I became aware of Bézier curves in the mid-1980s when I discovered that Adobe Illustrator had the ability to draw amazing curves quickly. You can find good treatment of the subject on Wikipedia, at *http://en.wikipedia.org/wiki/B%C3%A9zier_curve#Quadratic_curves*.

Here's basically how a quadratic Bézier curve works in SVG. You define an initial point (e.g., 100,200) using a pen-down command. From there, you set a course heading toward the next point; however, instead of actually moving to the next point, you just aim in that direction. So, for example, while "*M 100 200 L 200 400*" will make you actually arrive at the point (200,400), "*M 100 200 Q 200 400…*" will merely point you in that direction. Ultimately, you also need a final destination, which is the final coordinate pair required for a quadratic Bézier curve. In the example that follows, the command "*M 100,200 L 200,400 300,200*" draws a red path between (and reaching each of) the three points indicated. But simply replacing the *L* with a *Q* (i.e., "*M 100,200 Q 200,400 300,200*") produces a curve that passes through both endpoints and is a tangent to the associated lines of the allied line path at the endpoints of the segments.

Bézier Curve Example

This example clearly shows how the quadratic Bézier curve is created.



file://localhost/C:/Users/david/Documents/OReilly/SVGExamples/G02xx15.svg - Opera
File Edit View Bookmarks Tools Help
file://localhost/C:/User...
G02xx15.svg
Search with Google
0,0
400,0
M 100 200 Q 200,400 300,200
100,200
300,200
200,400

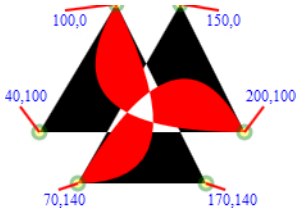
```
<path d="M 100 200 Q 200,400 300,200" fill="none" stroke="blue" />
<path d="M 100 200 L 200,400 300,200" fill="none" stroke="red"/>
```

While an infinite number of curves are tangent to both the line "*M 100 200 L 200 300*" at (100,200) and "*M 200 400 L 300 200*" at (300,200), only one quadratic shares these properties, even if you allow for rotations (in the sense of parametric equations) of the quadratic. That is, those three points in the plane uniquely define a specific curve. Likewise, any three noncollinear points in the plane determine one quadratic Bézier curve.

Revisiting the earlier example, which modified the fill rule to produce an empty space in the middle of the curve, you can draw the same curve with quadratic splines instead of lines to see the effect.

Here's an example of a graphic that uses a quadratic spline:

```
<path fill-rule="evenodd"
  d="M 70 140 L 150,0 200,100 L 40,100 100,0 L 170,140 70 140"/>

<path fill="red" fill-rule="evenodd"
  d="M 70 140 Q 150,0 200,100 Q 40,100 100,0 Q 170,140 70 140"/>
```



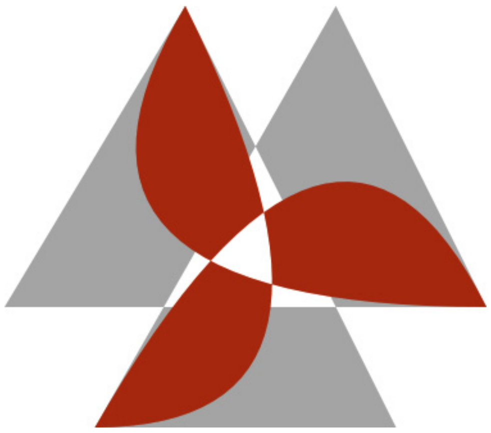
Note how the above example

```
<path id="H" fill="#bbb" fill-rule="evenodd"
  d="M 70 140 L 150,0 200,100 L 40,100 100,0 L 170,140 70 140"/>
```

can have its *L*s modified to *Q*s:

```
<path id="X" fill="#b42" fill-rule="evenodd"
  d="M 70 140 Q 150,0 200,100 Q 40,100 100,0 Q 170,140 70 140"/>
```

That produces a shape similar to the following (we've changed the colors and added in identifiers to the paths for easy reference in the text here):

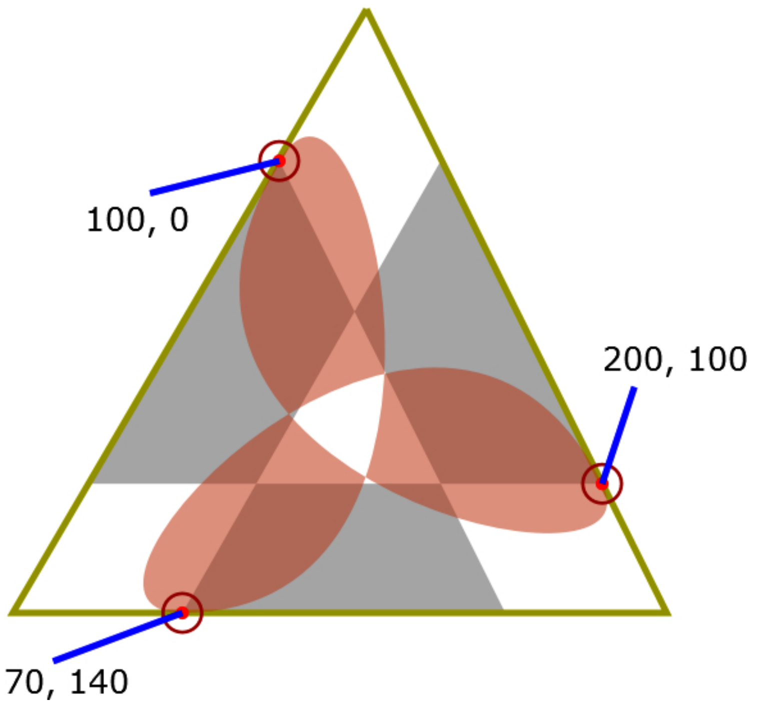


The figure shows two paths produced from the preceding code. Both paths have the same points, but one is linear (*id*="*H*") and the other is quadratic (*id*="*X*").

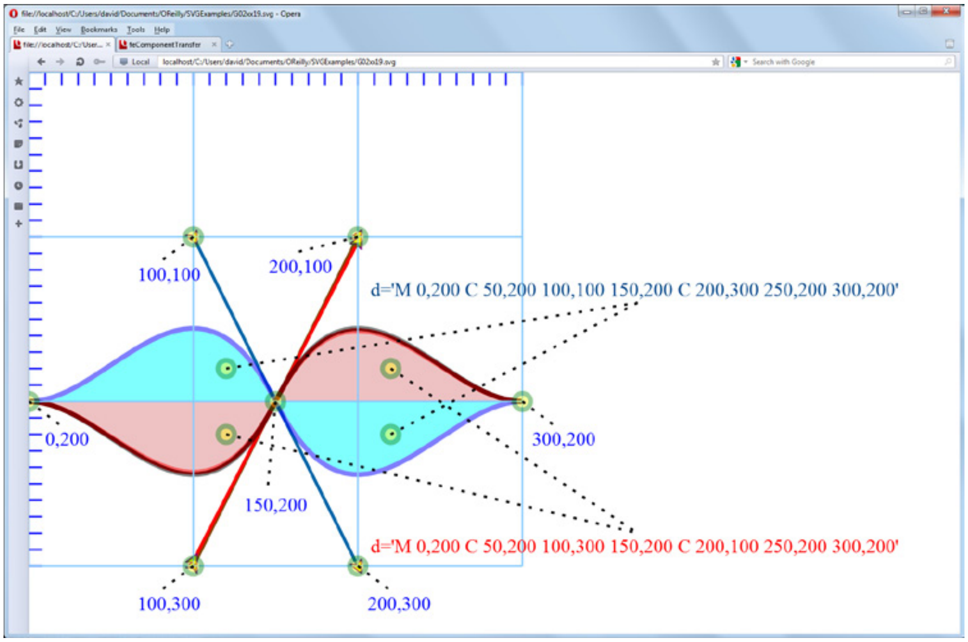
Observe that the angles of the reddish shape (*X*) at which the curves actually meet are sharp rather than rounded. Let's look more closely. If you're familiar with trefoil knots (see *http://en.wikipedia.org/wiki/Trefoil_knot*), then that is the sort of shape we'll be aiming toward.

First, observe that if the desired shape were to pass through any of the six points of the linear path *H*, then in order for the parts of the curve that meet there to be smooth, and for any of them to be tangent to lines of *H*, the new curve would have to extend beyond the bounds of *X*. You could extend the lines of *X* into a larger equilateral triangle and then work on building your trefoil knot. You could do this with cubic Bézier curves by defining a curve that passes through the same three endpoints (*http://www.w3techcourses.com/svg_images/lineOutCub.svg*) that it already does, but that is guided by the control points consisting of the three points of the circumscribed triangle (shown in the next figure as the light green line).

```
<path fill="#c53" fill-rule="evenodd" opacity=".5"
  d="M 70 140 C 17.5 ,140 150,0 200,100 C 220, 140 40,100 100,0 C 127,-47 170,140 70 140"/>
```



As a final example, the following demonstrates how to stitch Bézier curves together smoothly. For this to happen, the slopes of the lines at either side of a segment's endpoint must be the same.



```
d='M 0,200 C 50,200 100,100 150,200 C 200,300 250,200 300,200'
d='M 0,200 C 50,200 100,300 150,200 C 200,100 250,200 300,200'
```

Notice that the brown and blue paths share the same beginning points and endpoints, initial and final control points, and midpoints (150,200). They differ only in terms of the control points surrounding the midpoint. The blue path aims toward (100,100) and then changes direction toward (200,300), passing through the midpoint on its way and tangent to the line, as shown. Because the three relevant points, (100,100), (150,200), and (200,300), are collinear, the slopes of both segments are the same at the point where they meet, implying that the curve is smooth (continuously differentiable) at that point.

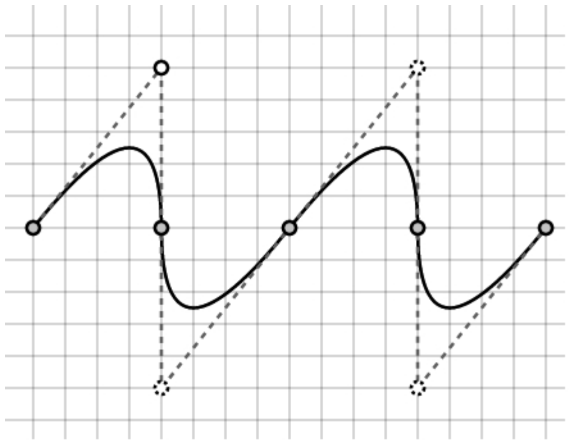
Creating Smooth Curves: The S and T Subcommands

These shortcut commands help with creating smooth curves, and they require fewer data points than constructing cubic and quadratic Bézier curves without these shortcut commands. This is because one of the Bézier curve points is used simply as a reference point, which is then reflected to create a smooth curve.

You use the *S* command to draw a smooth cubic Bézier spline segment from the current point to a new point (x,y). The previous segment must also be a smooth cubic Bézier spline, and that second control point is then reused via reflection relative to the current point as the segment's first control point. The second control must be explicitly specified.

You use the *T* command to draw a smooth quadratic Bézier spline segment from the current point to a new point (x,y). The previous segment must also be a smooth quadratic Bézier spline, and that control point is then reused via reflection relative to the current point.

The following image demonstrates the automatic reflection process for both these commands:

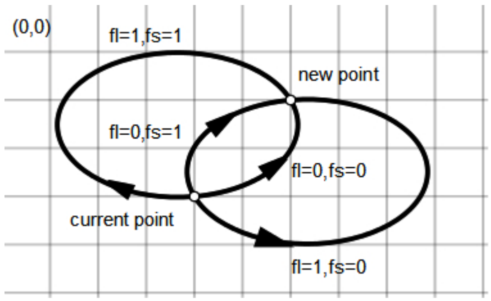


As you have seen, the *<path>* element can express both simple and complex shapes using the *L*, *H*, *V*, *Q*, and *C* commands. The geometric calculations involved are quite complex, which is why vector-drawing programs such as Inkscape, Illustrator, SVG-Edit, and Visio are very helpful in the SVG design process.

Elliptical Arc Example

One other often-used path command is the elliptical arc command (*A*), which allows you to quickly draw subsets of ellipses or intersecting ellipses. The arc subcommand of the *<path>* element has the following syntax: *A rx ry XAR large-arc-flag sweep-flag x y*.

The arc begins at the current point (which is determined by the last coordinate specified) and ends at (x,y), as demonstrated below:



You now have the choice of four elliptical arc segments: two small ones and two large ones. These arc segments can have a positive angular orientation (clockwise) or a negative orientation. The *large-arc-flag (fl)* controls the angular orientation of the larger arc segment via *fl = 0 : small*, *fl = 1 : large*. The *sweep-flag (fs)* controls the angular orientation analogously, via *fs = 0 : positive*, and *fs = 1 : negative*.

Note For the special case where the endpoint coordinates (x,y) are equal to the *current point's* coordinates, the arc will not get rendered. Because this behavior is not intuitive when *large-arc-flag* is set to 1, this might be changed in the SVG 2.0 specification.

Using this elliptical arc information, here's the code to create a simple spiral:

```
<svg width="600" height="400" viewBox="0 0 400 300">
<path stroke="darkslategray" stroke-width="6" fill="none"
  stroke-linecap="round"
  d="M50,100
    A100,50 0 0 1 250,100
    A80,40 0 0 1 90,100
    A60,30 0 0 1 210,100
    A40,20 0 0 1 130,100
    A20,10 0 0 1 170,100" />
</svg>
```

That code produces the following spiral:

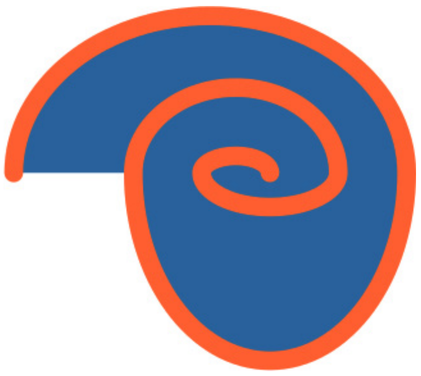


Table 2-2 provides a quick reference for the path commands and properties.

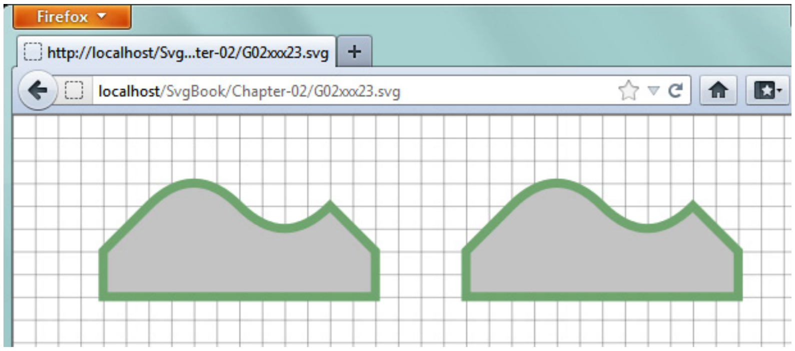
TABLE 2-2 Path Commands

Commands	Parameters	Instruction
M, m	*x, y*	Move to a new point (x,y).
L, l	*x, y*	Draw a line from the current point to a new point (x,y).
H, h	*x*	Draw a horizontal line from the current point to a new point (x,current-point-y).
V, v	*y*	Draw a vertical line from the current point to a new point (current-point-x,y).
A, a	*rx, ry, x-axis-rotation, large-arc-flag, sweep-flag, x, y*	Draw an elliptical arc from the current point to a new point (x,y). The arc belongs to an ellipse that has radii *rx* and *ry* and a rotation with respect to the positive x-axis of *x-axis-rotation* (in degrees). If *large-arc-flag* is 0 (zero), then the small arc (less than 180 degrees) is drawn. A value of 1 results in the large arc (greater than 180 degrees) being drawn. If *sweep-flag* is 0, then the arc is drawn in a negative angular direction (counterclockwise); if it is 1, then the arc is drawn in a positive angular direction (clockwise).

Commands	Parameters	Instruction
Q, q	*x1, y1* *x, y*	Draw a quadratic Bézier curve from the current point to a new point (x,y) using (x1,y1) as the control point.
T, t	*x, y*	Draw a smooth quadratic Bézier curve segment from the current point to a new point (x,y). The control point is computed automatically as the reflection of the control point on the previous command relative to the current point. If there is no previous command or if the previous command was not a *Q, q, T*, or *t*, the control point is coincident with the current point.
C, c	*x1, y1* *x2, y2* *x, y*	Draw a cubic Bézier curve from the current point to a new point (x,y) using (x1,y1) and (x2,y2) as control points.
S, s	*x2, y2* *x, y*	Draw a smooth cubic Bézier curve segment from the current point to a new point (x,y). The first control point is computed automatically as the reflection of the control point on the previous command relative to the current point. If there is no previous command or if the previous command was not a *C, c, S*, or *s*, the first control point is coincident with the current point. (x2,y2) is the second control point.

Relative vs. Absolute Path Coordinates

This next example uses a mixture of MoveTo (*M*), Vertical (*V*), LineTo (*L*), Bézier (*Q*), HorizontalTo (*H*), and ClosePath (*Z*) commands to generate a fairly elegant shape, as shown on the left of the following image. The example on the right requires less spatial brain power to generate the same shape because it uses *relative* versions of commands (i.e., lowercase commands). The coordinates of the new point are relative to the position of the previous point (40,80).


Firefox
http://localhost/Svg...ter-02/G02xxx23.svg
localhost/SvgBook/Chapter-02/G02xxx23.svg

Note The data of the path's *d* attribute actually follows a specific set of rules, called the Backus-Naur Form (BNF). You can find more detailed information on these rules at *http://www.w3.org/TR/SVG/paths.html#PathDataBNF*.

Accessing and Reusing Graphics

From buttons, icons, and window UIs, to building graphs and gaming graphics, there are many logical use cases for accessing and reusing raster and vector graphics in SVG.

Linking to both internal and external image data is worth a quick mention here because it is a common method for accessing and reusing SVG.

Referencing Vector and Bitmap Images

The SVG language provides the <*image*> element, which can reference other SVG images, as well as PNG and JPEG bitmap images. The syntax for the <*image*> element is similar to the <*rect*> element in that it has *x*, *y*, *width*, and *height* attributes.

The <*image*> element has the additional attribute *xlink:href,* which allows you to specify the location of the referenced image. Similar to HTML's *href* attribute, the *xlink:href* attribute allows the referenced image to be stored either locally or on the Internet. The code for referencing a bitmap image is as follows:

```
<image xlink:href="GrandMothersParty-121YO.png" x="340" y="0" width="140"
  height="160" opacity="0.5"/>
```

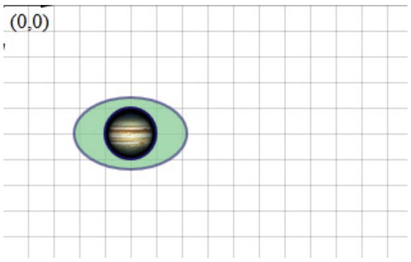
Referencing other SVG images is just as easy and becomes very useful in many application scenarios, such as reusing the same vector symbol on a page or dynamically loading vector images on demand.

The Group Element

The SVG group element, <*g*>, is great for logically grouping sets of related graphical objects. This group capability makes it easy to add styles, transformations, interactivity, and even animations to entire groups of objects. The following code groups a circle and a bitmap image together into a group named *iris*, which is then grouped together with an ellipse shape into another group named *eye*.

```
<!-- Group containing the eye. -->
<g id="eye">
  <!-- Draw the ellipse. -->
  <ellipse fill="#a1d9ad" fill-opacity="0.7" fill-rule="nonzero"
           stroke="#32287d" stroke-width="1" stroke-opacity="0.5" />

  <!-- Group containing the eye's iris. -->
  <g id="iris"
     cx="50" cy="50" rx="20" ry="14" />

     <!-- Draw the circle. -->
     <circle fill="black" fill-opacity="1" fill-rule="nonzero"
             stroke="#32287d" stroke-width="1" stroke-linecap="butt"
             stroke-linejoin="bevel" stroke-miterlimit="4"
             id="path3395" cx="50" cy="50" r="10" />
```

```
      <!-- Reference the bitmap image (PNG) -->
      <image id="bitmapCentralBall"
              width="5.5%" height="5.5%"
              x="39px" y="42px"
              xlink:href="iris-small.png"
              alt="NASA Photo of Jupiter" />
  </g>
</g>
```



With some creativity, you could then add some scripted interactivity such that the *iris* group could follow the mouse, while the *eye* group could blink randomly or at set intervals.

You'll see another great use for *<g>* during the discussion of transformations and interactivity in SVG in Chapter 4, "Motion and Interactivity." You can associate items together in a group and then define transformations to move, scale, or rotate all the items together so that their spatial relations to one another are maintained. Through the use of interactivity in SVG, you can assign, for example, an *onclick* event to an entire group so that all elements within the group respond to the event.

The <use> Element

The *<use>* element lets you reuse existing elements and thus write less code. Like the *<image>* element, *<use>* takes *x*, *y*, *height*, and *width* attributes, and it references other content using the *xlink:href* attribute.

As an example, you can reuse the following rectangle

```
<!-- Draw the upper-right rectangle. -->
<rect fill="#ada1d9" fill-opacity="1" fill-rule="nonzero"
    stroke="#32287d" stroke-width="10" stroke-linecap="butt"
    stroke-linejoin="bevel" stroke-miterlimit="4" stroke-opacity="0.4"
    id="rectangle" width="20" height="20" x="90" y="-10" />
```

by referencing it with the *<use>* element:

```
<!-- Reuse the first rectangle element and move it to a different position. -->
<use x="" y="" xlink:href="#rectangle" />
```

Creating Patterns

The SVG language helps you create and reuse patterns with ease. Patterns are extremely useful—in fact, the grid background found in many of this book's examples is just a simple pattern that consists of a single 10-by-10-pixel rectangle. The *<defs>* element can be used to store content that will not be directly displayed. This stored hidden content can then be referenced and displayed by other SVG elements, which makes it ideal for things such as patterns that contain reusable graphics.

To create a basic pattern in SVG, first place a rectangle within a *<pattern>* element, and then put everything inside of a *<defs>* element.

```
<defs>
  <pattern id="Pattern01" width="10" height="10" patternUnits="userSpaceOnUse">
    <rect width="10" height="10" fill="#FFFFFF" stroke="#000000" stroke-width="0.1"/>
  </pattern>
</defs>
```

Now, to use this pattern anywhere in your SVG graphic, simply set your element's *fill* attribute value to the *id* of the pattern, like this: *url(#Pattern01)*.

```
<rect id="Background" x="0" y="0" width="100%" height="100%"
  fill="url(#Pattern01)" stroke-width="0.5" stroke="#000000" />
```

Case Study: Designing a Reusable Pattern

The example in this section gives you a closer look at how to write SVG code that generates a pattern composed of both vector and bitmap graphics.

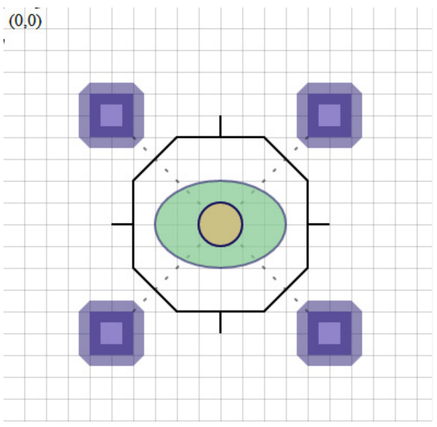
Adding Basic Shapes

Building upon your knowledge up to this point, you'll walk through each step of the design and creation process.

1. Create and save a file named *tile.svg* that contains the following lines of code:

```
<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink"
  version="1.1"
  width="800" height="600"
  viewBox="0 0 400 300" preserveAspectRatio="none">
  <g id="layer1"></g>
</svg>
```

2. With this framework in place, you can start adding some basic shapes. The next example shows a simple pattern design. Tile patterns are known mathematically as *tessellations of the plane*.



To create this pattern in SVG code, first create the following line:

```
<line stroke="#000000" stroke-width="1" stroke-linecap="round"
          stroke-linejoin="round" stroke-miterlimit="4" stroke-opacity="0.4"
          stroke-dasharray="1, 6" stroke-dashoffset="0"
          x1="90" y1="10" x2="10" y2="90"
          id="patternLine1" />
```

Now, as mentioned earlier, you can reuse the line. By changing the *x* and *y* values you can effectively rotate the line by 90 degrees. Also, to mix the pattern up a bit, you can override *stroke-opacity* and other style attributes that would otherwise be inherited from the referenced element:

```
<use stroke-opacity="1"
       transform="rotate(90, 50, 50)"
         xlink:href="#patternLine1"
         id="patternLine2" />
```

3. Next, draw the rest of the elements that you want to include in your pattern—for example:

```
<ellipse fill="#a1d9ad" fill-opacity="0.7" fill-rule="nonzero"
              stroke="#32287d" stroke-width="1" stroke-opacity="0.5"
              id="path3389" cx="50" cy="50" rx="30" ry="20" />

<!-- Draw the upper-right rectangle. -->
<rect fill="#ada1d9" fill-opacity="1" fill-rule="nonzero"
       stroke="#32287d" stroke-width="10" stroke-linecap="butt"
       stroke-linejoin="bevel" stroke-miterlimit="4" stroke-opacity="0.4"
       id="patternRect-upperRight"
       width="20" height="20" x="90" y="-10" />

<!-- Reuse the first rectangle element and rotate it 90 degrees each time. -->
```

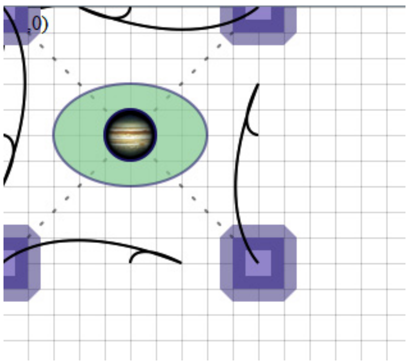
```
<use transform="rotate(90, 50, 50)"
         xlink:href="#patternRect-upperRight"
         id="patternRect-lowerRight" />
<use transform="rotate(180, 50, 50)"
         xlink:href="#patternRect-upperRight"
         id="patternRect-lowerLeft" />
<use transform="rotate(270, 50, 50)"
         xlink:href="#patternRect-upperRight"
         id="patternRect-upperLeft" />

<!-- Draw the circle. -->
<circle fill="#d9d2a1" fill-opacity="1" fill-rule="nonzero"
            stroke="#32287d" stroke-width="1" stroke-linecap="butt"
            stroke-linejoin="bevel" stroke-miterlimit="4"
            id="path3395" cx="50" cy="50" r="10" />

<!-- Draw the path using "relative"coordinates via lowercase path commands.
     Note that we can easily switch to using the Polyline element by changing
     the "d" attribute to "points". -->
<path fill="none" stroke="#000000" strGoke-width="1px" stroke-linecap="butt"
          stroke-linejoin="miter" stroke-opacity="1"
          d="m 0,50 10,0 0,20 20,20 0,0 0,0 20,0 0,10"
          id="patternPath-lowerLeft" />

<!-- Reuse the first path, rotate it 90 more degrees for each of the four corners. -->
<use transform="rotate(90, 50, 50)"
         xlink:href="#patternPath-lowerLeft"
         id="patternPath-upperLeft" />
<use transform="rotate(180, 50, 50)"
         xlink:href="#patternPath-lowerLeft"
         id="patternPath-upperRight" />
<use transform="rotate(270, 50, 50)"
         xlink:href="#patternPath-lowerLeft"
         id="patternPath-lowerRight" />
```

These SVG elements form the basis for the pattern that you will create in the next step. You may have noticed the use of the *transform* attribute. You can see how the referenced rectangle and path shapes were moved into a different position via a rotate command. The next chapter will cover the usefulness of transformations in greater detail.

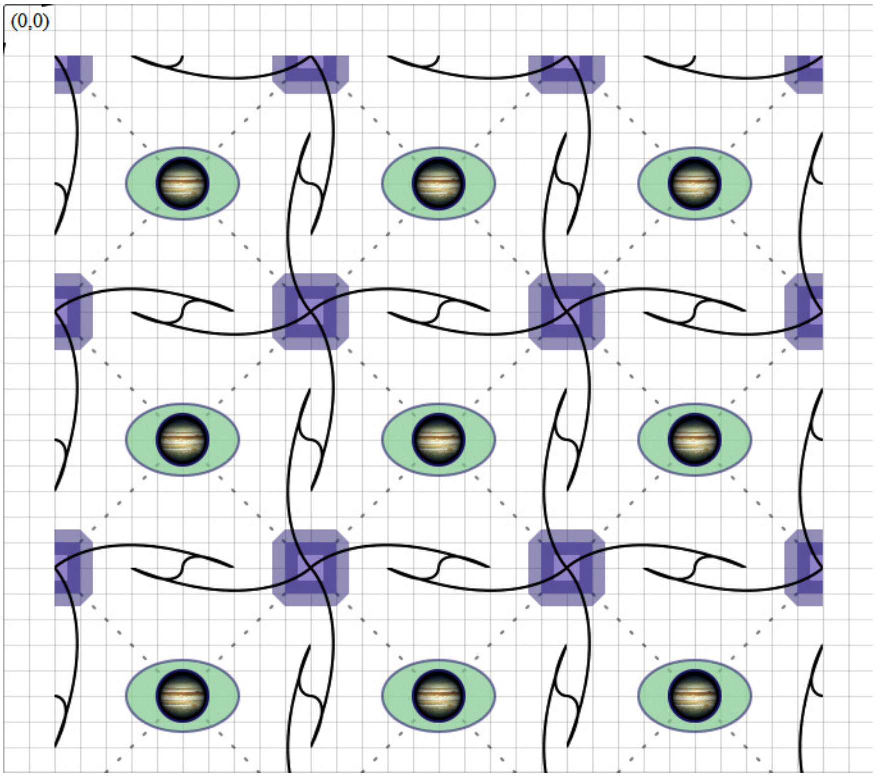
4. To create a more interesting pattern design, rather than using simple MoveTo (*M*) path commands, simply alter the <*path*> element's values to use a relatively positioned smooth quadratic Bézier curve using the *s* command, and an absolutely positioned cubic Bézier curve using *C*. So, the path's data becomes the following:

```
d="M 0,50 s 10,0 0,20 C 20,20 0,0 0,0"
```

5. Add a reference to a bitmap image of the planet Jupiter and position the image at the center. Also, move the graphics to the origin of the coordinate system, which equals the x,y value of (0,0), to complete your initial tile design. Now the tile looks like this:



6. Finally, add the <*pattern*> element inside of a <*defs*> element and move the tile design graphics inside of the <*pattern*>.



```
<defs>
  <pattern id="gridPatternWithTessellation"
                 x="20" y="20" width="100" height="100
                 patternUnits="userSpaceOnUse">
      <!--Insert the tile elements here.  -->
  </pattern>
</defs>
```

Below the *<defs>*, you then simply create a rectangle, path, or any other SVG shape and set its *fill* value to be the pattern, as shown at the end of the full code listing below.

```
<svg
   xmlns="http://www.w3.org/2000/svg"
   xmlns:xlink="http://www.w3.org/1999/xlink"
   id="chapter2-ShapesPatternsGroupsUse"
   version="1.1"
   width="800" height="600"
   viewBox="0 0 400 300" preserveAspectRatio="none"
>
  <defs>
    <!-- Begin Example -->
    <pattern id="gridPatternWithTessellation" x="20" y="20" width="100" height="100"
      patternUnits="userSpaceOnUse">
      <!-- Draw the lines. -->
      <line stroke="black" stroke-width="1" stroke-linecap="round" stroke-
linejoin="round"
         stroke-miterlimit="4" stroke-opacity="0.4" stroke-dasharray="1, 6"
         stroke-dashoffset="0"
         x1="90" y1="10" x2="10" y2="90"
         id="patternLine1" />
      <!-- Reuse the first line, rotate it 90 degrees, and update the style attributes.
-->
      <!-- For appendix or wiki - note that currently most browsers do not support
styling
      of Use elements using either CSS or SVG attributes -->
      <use stroke-opacity="1"
        transform="rotate(90, 50, 50)"
         xlink:href="#patternLine1"
         id="patternLine2" />
      <!-- Draw the upper-right rectangle. -->
      <rect fill="#ada1d9" fill-opacity="1" fill-rule="nonzero" stroke="#32287d"
         stroke-width="10" stroke-linecap="butt" stroke-linejoin="bevel"
         stroke-miterlimit="4" stroke-opacity="0.4"
         id="patternRect-upperRight"
         width="20"
         height="20"
         x="90"
         y="-10" />
      <!-- Reuse the first rectangle element and rotate it 90 degrees each time. -->
      <use transform="rotate(90, 50, 50)"
         xlink:href="#patternRect-upperRight"
         id="patternRect-lowerRight" />
      <use transform="rotate(180, 50, 50)"
         xlink:href="#patternRect-upperRight"
         id="patternRect-lowerLeft" />
      <use transform="rotate(270, 50, 50)"
         xlink:href="#patternRect-upperRight"
         id="patternRect-upperLeft" />
      <!-- Group containing the eye. -->
      <g id="eye">
        <!-- Draw the ellipse. -->
        <ellipse fill="#a1d9ad" fill-opacity="0.7" fill-rule="nonzero"
          stroke="#32287d" stroke-width="1" stroke-opacity="0.5"
        cx="50" cy="50" rx="22" ry="14" />
      <!-- Group containing the eye's iris. -->
      <g id="iris">
         id="path3389"
         cx="50" cy="50" rx="20" ry="14" />

        <!-- Draw the circle. -->
        <circle fill="black" fill-opacity="1" fill-rule="nonzero" stroke="#32287d"
           stroke-width="1" stroke-linecap="butt" stroke-linejoin="bevel"
           stroke-miterlimit="4"
           id="path3395"
           cx="50" cy="50" r="10" />
        <!-- Reference the bitmap image (PNG) -->
        <image id="bitmapCentralBall"
           width="5.5%" height="5.5%"
           x="39px" y="42px"
           xlink:href="iris-small.png"
           alt="NASA Photo of Jupiter" />
      </g>
    </g>
    <!-- Draw the path using "relative" coordinates via lowercase path commands.
    Note that we can easily switch to using the Polyline element by changing
   the "d"
    attribute to "points". -->
    <path fill="none" stroke="black" stroke-width="1px" stroke-linecap="butt"
       stroke-linejoin="miter" stroke-opacity="1"
       d="M 0,50 s 10,0 0,20 C 20,20 0,0 0,0"
       id="patternPath-lowerLeft" />
       <!-- Other interesting paths
            MoveTo Polyline-like d="m 0,50 10,0 0,20 20,20 0,0 0,0 20,0 0,10"
            Quadratic d="M 0,50 Q 10,0 0,20 S 20,20 0,0"
            Smooth Quadratic d="M 0,50 S 10,0 0,20 Q 20,20 0,0"
            Cubic d="M 0,50 C 10,0 0,20 20,20 S 0,0 0,0"
            Smooth Quadratic & Cubic d="M 0,50 s 10,0 0,20 C 20,20 0,0 0,0" -->

            -->
    <!-- Reuse the first path, rotate it 90 more degrees for each of the
    four corners. -->
    <use
      transform="rotate(90, 50, 50)"
       xlink:href="#patternPath-lowerLeft"
       id="patternPath-upperLeft" />
    <use
      transform="rotate(180, 50, 50)"
       xlink:href="#patternPath-lowerLeft"
       id="patternPath-upperRight" />
    <use
      transform="rotate(270, 50, 50)"
       xlink:href="#patternPath-lowerLeft"
       id="patternPath-lowerRight" />
  </pattern>
  </g>
  <pattern id="gridPattern" width="10" height="10" patternUnits="userSpaceOnUse">
     <path d="M10 0 L0 0 L0 10" fill='none' stroke='gray' stroke-width='0.25'/>
  </pattern>
</defs>
<g id="layer1">
```

```
        <!-- background grid -->
        <rect id="grid" width="100%" height="100%" x="0" y="0"
           stroke='gray' stroke-width='0.25' fill='url(#gridPattern)'/>
        <!-- grid illustrations -->
        <use xlink:href="#coords"/>
        <text x="3" y="9" font-size='8'>(0,0)</text>
        <!-- Begin Example -->
        <rect id="gridWithTessellation" width="300" height="300" x="20" y="20"
           fill='url(#gridPatternWithTessellation)' />
      </g>
      <rect id="gridWithTessellation"
              x="20" y="20" width="300" height="300"
              fill='url(#gridPatternWithTessellation)' />
    </svg>
```

With just these lines of code, you have created an interesting work of art and a useful tiling pattern that has all the benefits of SVG. To meet the needs of your company, group, or imagination, you only need to edit the base tile to create an entirely different design for your application.

Because there is a bitmap image within the pattern, if end users zoom in they will see a slightly pixelated graphic surrounded by the smoother, unpixelated vector graphics. You should consider bitmap pixelation when your project requires high-fidelity printouts.

Note There are several programs that assist with creating tiles for patterns. Inkscape, for example, has some excellent built-in pattern creation features, and there is a powerful pattern creation program on the LearnSVG.com website as well, which was originally developed by Michel Hirtzler (see *http://pilat.free.fr/tiling_loc/tile.svg*).

Summary

At this point, you should be off to a great start exploring the expressive language of SVG. This chapter showed you the basics of working with SVG in a very condensed form, including creating basic vector shapes and paths, building more complex shapes, and creating and working with patterns. In the next chapter, we will delve into animations and scripting, as well as gradient rotation, scaling, and other transformations.

Index

About the Authors

DAVID DAILEY was born and raised in Albuquerque, NM, receiving his bachelor's degree from the University of New Mexico and his doctorate from the University of Colorado. Having taught mathematics, psychology, and computer science at the Universities of Wyoming, Tulsa, and Alaska, he later moved east with appointments at Vassar, Williams, and Bay Path College, before settling in at Slippery Rock University in Pennsylvania where he is Professor of Computer Science teaching mainly in areas of web programming. He is married, has four children, and enjoys creating art, food, music, and games.

JON FROST is a seasoned developer who has worked with SVG for more than a decade. The SVG applications he has developed include interactive web applications and dynamic reports. He dreamt up and collaborated on the books *Learn SVG: The Web Graphics Standard* and *Building Web Applications with SVG*.

DOMENICO STRAZZULLO, founder and editor-in-chief of *SVG magazine*, is the author of both the Pergola JavaScript library for SVG and the open-source GEMï web operating system.