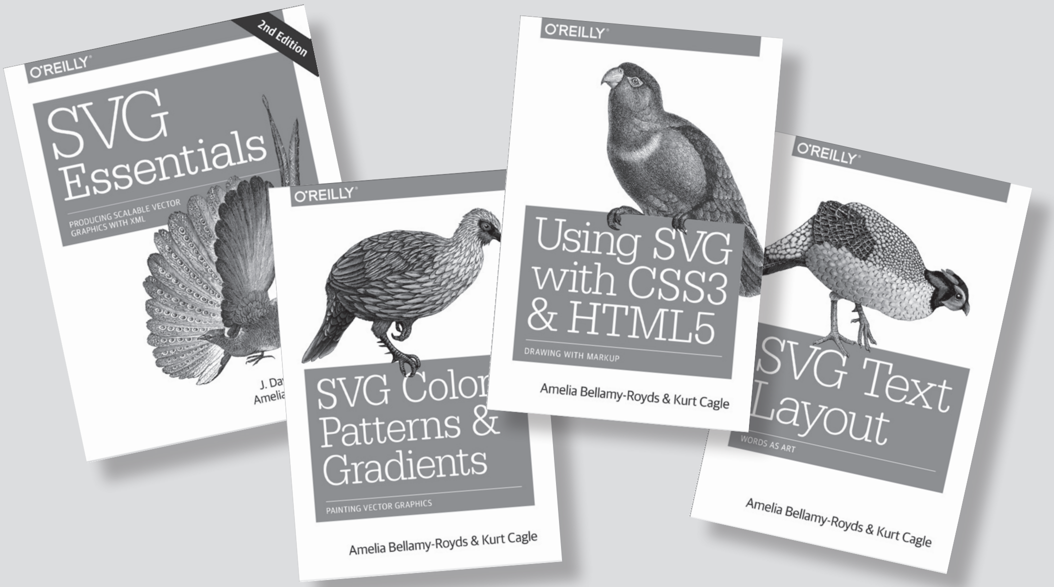
O'REILLY®

Modern SVG

A Curated Collection of Chapters from the O'Reilly SVG Library



O'REILLY
2nd Edition
SVG Essentials
PRODUCING SCALABLE VECTOR GRAPHICS WITH XML
J. Da
Amelia
O'REILLY
SVG Color Patterns & Gradients
PAINTING VECTOR GRAPHICS
Amelia Bellamy-Royds & Kurt Cagle
O'REILLY
Using SVG with CSS3 & HTML5
DRAWING WITH MARKUP
Amelia Bellamy-Royds & Kurt Cagle
O'REILLY
SVG Text Layout
WORDS AS ART
Amelia Bellamy-Royds & Kurt Cagle

Amelia Bellamy-Royds & Kurt Cagle

Modern SVG

Amelia Bellamy-Royds and Kurt Cagle

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Modern SVG
by Amelia Bellamy-Royds and Kurt Cagle

Printed in the United States of America.

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

Editor: Meg Foley
Production Editor: Kristen Brown
Proofreader: O'Reilly Production Services
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

March 2016: First Edition

Revision History for the First Edition
2016-02-25: First Release

978-1-491-95550-5

[LSI]

Table of Contents

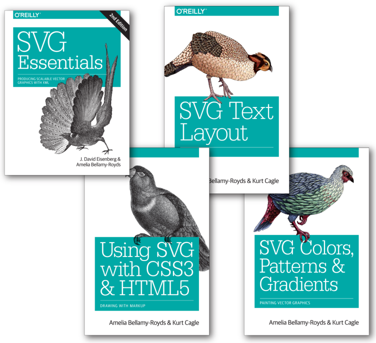
Modern SVG

A Curated Collection of Chapters from the O'Reilly SVG Library

SVG has come a long way. It may have seemed like a niche topic a few years ago, but it has evolved into a powerful tool for Web developers.

Scalable Vector Graphics (SVG) is an XML-based vector image format for two-dimensional graphics with support for interactivity and animation. SVG images and their behaviors are defined in XML text files, which means that they can be searched, indexed, scripted, and compressed. SVG images can be created and edited with any text editor, but are more often created with drawing software.

This free ebook gets you started, bringing together concepts that you need to understand before tackling your next modern SVG project. With a collection of chapters from the O'Reilly SVG library's published and forthcoming books, you'll learn about the scope and challenges that await you in the world of modern web development.


O'REILLY
SVG Essentials
2nd Edition
PRODUCING SCALABLE VECTOR GRAPHICS WITH XML
J. David Eisenberg & Amelia Bellamy-Royds
O'REILLY
SVG Text Layout
Bellamy-Royds & Kurt Cagle
Using SVG with CSS3 & HTML5
DRAWING WITH MARKUP
Amelia Bellamy-Royds & Kurt Cagle
SVG Colors, Patterns & Gradients
PAINTING VECTOR GRAPHICS
Amelia Bellamy-Royds & Kurt Cagle

CHAPTER 1

Using SVG in Web Pages

From SVG Essentials

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

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

SVG as an Image

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

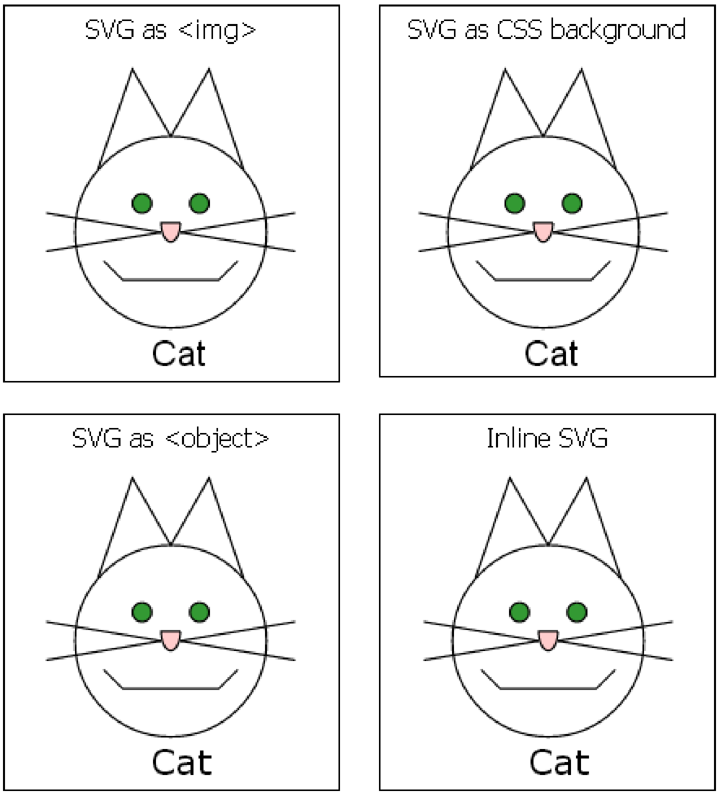


Figure 1-1. Screenshot of a web page with SVG inserted four ways

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

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

Including SVG in an <img> Element

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

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

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

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

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

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

Including SVG in CSS

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

By default, a background image is drawn at its intrinsic dimensions and repeated in both the horizontal and vertical direction to fill up the dimensions of the element. The intrinsic dimensions of an SVG file are determined in the same manner as described in "Including SVG in an <img> Element" on page 3. If there are no intrinsic dimensions, the SVG will be scaled to 100% of the height and width of the element. The size can be set explicitly using the `background-size` property, and repeat patterns and image position can be set using `background-repeat` and `background-position`:

```
div.background-cat {
 background-image: url("cat.svg");
 background-size: 100% 100%;
}
```

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

NOTE

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

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

SVG as an Application

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

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

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

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

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

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

<object> versus <embed>

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

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

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

When you include an SVG file as an embedded object (whether with `<object>` or `<embed>`), the external file is rendered in much the

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

same way as if it was included in an `<img>` element: it is scaled to fit the width and height of the embedding element, and it does not inherit any styles declared in the parent document.

Unlike with images, however, the embedded SVG can include external files, and scripts can communicate between the object and the parent page, as described in Chapter 13 in *SVG Essentials*.

SVG Markup in a Mixed Document

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

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

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

Foreign Objects in SVG

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

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

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

The rectangle is evaluated in the local SVG coordinate system, and so is subject to coordinate system transformations (which we'll talk about in Chapter 6 in *SVG Essentials*) or other SVG effects. The child XML document is rendered normally into a rectangular frame, and then the result is manipulated like any other SVG graphic. An SVG foreign object containing an XHTML paragraph is shown in Figure 1-2.

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

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

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

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

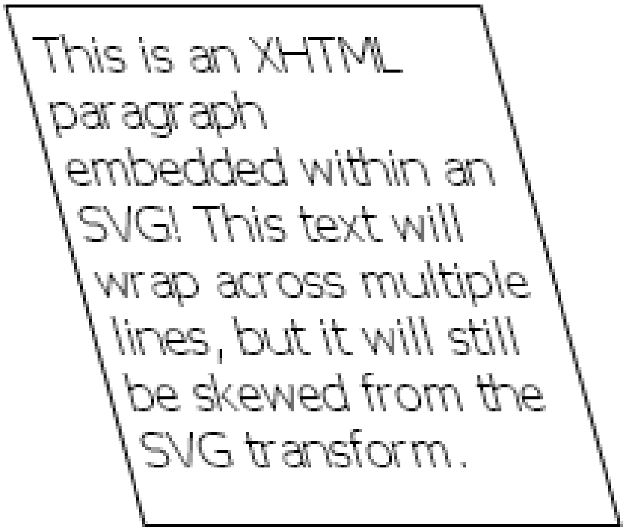

This is an XHTML paragraph embedded within an SVG! This text will wrap across multiple lines, but it will still be skewed from the SVG transform.

Figure 1-2. Screenshot of an SVG file containing XHTML text

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

```
<g transform="skewX(20)">
<switch>
  <!-- select one child element  -->
  <foreignObject x="1em" y="25%" width="10em" height="50%"
     requiredFeatures=
     "http://www.w3.org/TR/SVG11/feature#Extensibility">
     <body xmlns="http://www.w3.org/1999/xhtml">
        <p>This is an XHTML paragraph embedded within an SVG!
           So this text will wrap nicely around multiple lines,
           but it will still be skewed from the SVG transform.
        </p>
     </body>
  </foreignObject>
  <text x="1em" y="25%" dy="1em">
    This SVG text won't wrap, so it will get cut off...
  </text>
```

```
</switch>
</g>
```

Inline SVG in XHTML or HTML5

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

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

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

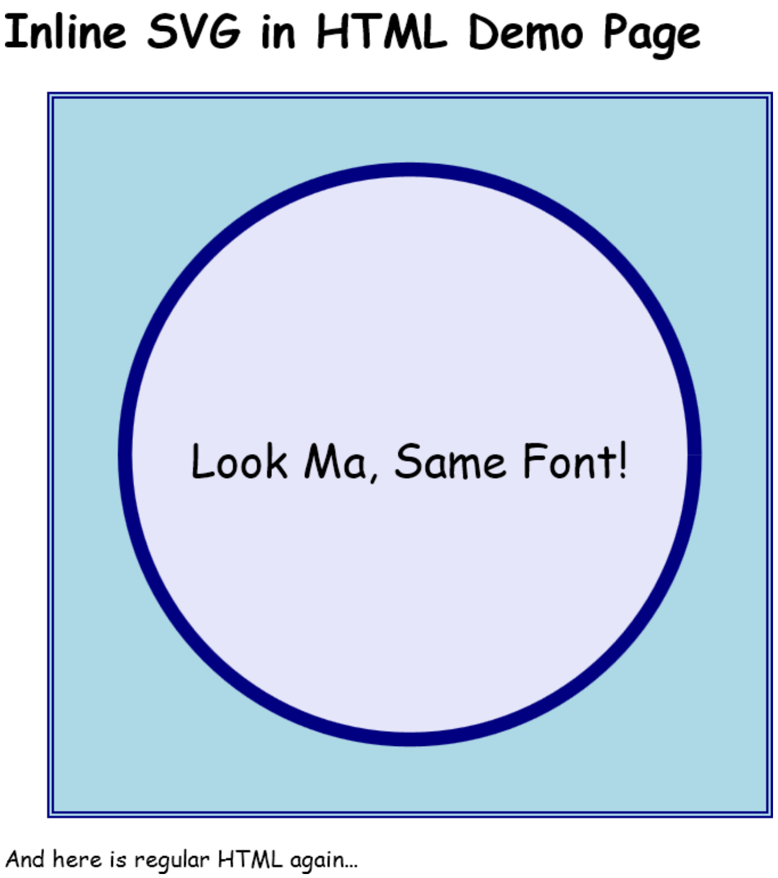
By default, the SVG will be positioned with the inline display mode (meaning that it is inserted within the same line as the text before and after it), and will be sized based on the height and width attributes of the `<svg>` element. With CSS, you can change the size by setting the `height` and `width`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

Example 1-2 gives the code for a very simple SVG drawing in a very simple HTML5 document. The result is Figure 1-3. The `xmlns` attribute on the `<svg>` element is optional for HTML5. For an XHTML document, you would change the `DOCTYPE` declaration at the top of the file, and you would wrap the CSS code in the `<style>` element with a `<![CDATA[...]]>` block.

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

2 CSS positioning properties apply to top-level `<svg>` elements, ones which are direct children of HTML elements. An `<svg>` that is a child of another SVG element will be positioned based on the rules for nested SVGs, as described in Chapter 5 in *SVG Essentials*.

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


Inline SVG in HTML Demo Page

Look Ma, Same Font!

And here is regular HTML again…

Figure 1-3. The web page from Example 1-2

Example 1-2. Inline SVG within an HTML file

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

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

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

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

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

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

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

CHAPTER 2

Tools of the Trade

From Using SVG with CSS3 and HTML5

Software and Sources to Make SVG Easier

The SVG examples in this book were for the most part created "from scratch", by typing markup or standard JavaScript to build and manipulate the graphics. However, that's certainly not the only way to work with SVG, nor the most common one.

Most SVG art starts life inside graphical software, created by an artist working with shapes and colors instead of XML tags and attributes. Most SVG data visualizations are built using JavaScript libraries full of shorthand convenience functions. Most SVG icons are reused from one web application to another, and countless icon sets are available to license for your website.

By showing you the internal components of an SVG, stripped down to their skeletal form, we hope to give you a complete toolset to work with SVG: the skills to modify and extend *any* SVG you work with, no matter how it was created. With this programmatic approach to SVG, you will be better able to manipulate graphics created by others or by software, in order to match your web design or to enable user interaction. However, the mental toolset you'll gain by understanding SVG shouldn't detract from the software tools that other developers have created.

Software tools make it easier to create graphics, and easier to process your files so they are ready to deploy on your web server. Tools discussed in this chapter include graphical editors that emphasize visual components instead of code, code editors that provide hints and immediate feedback, code libraries that streamline the creation of dynamic graphics with JavaScript, and the many rendering programs that display the SVG or convert it to other image formats. In addition, we introduce the vast supply of free and licensable SVG content that can help you quickly enhance your web designs, even if your most artistic drawings are of the stick-figure variety.

This chapter specifically mentions some of the most popular software and services. These are not the only options, and we don't guarantee they are the best. They are given as examples of what is out there, and of how different options differ from each other. When choosing tools for your own projects, think about the features you need, and the budget you can afford. Whatever you choose, remember that any standards-compliant SVG can be opened and modified with other SVG tools, so you're not locked into any one product's workflow.

Ready-to-Use SVG

The easiest way to get started with SVG—especially if you're more of a programmer than a graphic designer—is to start with someone else's art. SVG may not be as ubiquitous on the Web as other image formats, but it's getting there. A simple web search for the word will retrieve numerous vendors and free collections of SVG, particularly for icons.

Prior to using clip art from a vendor or website, you should ascertain what kind of licenses are available for the graphic. Early in the history of SVG, most graphics were released under some kind of Creative Commons license, but increasingly, high-quality artwork is produced by professional artists working within traditional copyright domains. Although there are plenty of free-to-use graphics available (some with noncommercial restrictions or attribution requirements), others are offered under paid license systems similar to those used for stock photos or web fonts.

TIP

One benefit of SVG's dual nature as both an image format and an XML format is that it is possible to embed copyright license information directly in the file using a `<metadata>` block. We'll discuss how you can do this for your own graphics in Chapter 16 of *Using SVG with CSS3 and HTML5*.

For accessing graphics created by others, remember that creative works are by default "all rights reserved"; the absence of copyright information doesn't mean it is public domain. Don't use someone else's work unless you are sure that the original creator has offered a license compatible with your intended use.

SVG will never replace JPEG for stock photographs (which can't be efficiently represented in vectors), but it is now a standard option for vector graphic clip art and icons, including from commercial suppliers.

There are a number of tools and libraries that can convert simple SVG art into other vector formats and back again. This can increase the flexibility of the vector graphics: for example, the encapsulated PostScript (EPS) format, long a staple in the realm of clip art, is still dominant in print. For simpler graphics, icon fonts—which allow sets of single-color icons to be distributed as a web font file—are popular because they allow icon size and color to be controlled with familiar CSS properties. Nonetheless, companies that produce clip art, maps, and related graphics for the Web are increasingly shifting to SVG for their vector graphic format.

Using the vector graphic source files, the stock art companies can also generate raster images (PNG, JPEG, and GIF) of any size. For a web designer interested in purchasing raster graphics, however, it often makes more sense to license a single SVG and convert it into the associated raster format at various resolutions yourself, instead of purchasing raster graphics at different scales.

The following list of websites should help start you on your search for SVG:

Open Clip Art Project
: The Open Clip Art Library (OCAL) Project is definitely the oldest, and perhaps the largest, repository of SVG content, all of it available either through Creative Commons or public domain licenses for unrestricted commercial use. Established in 2004 by Jon Phillips and Bryce Harrington, the OCAL project was created to provide a public commons for clip art, using SVG for encoding primarily because SVG itself doesn't have the same type of vendor encumbrances or royalty restrictions as other vector formats. Moreover, because the source code for the graphics can be read with a text editor, it's also possible to decompose clip art images into separate pieces, making SVG useful as a way of packaging collections of icons or images in a single file. Figure 2-1 displays some of the diversity of artistic styles available. The project is also integrated with the Flaming

Text ImageBot graphics editor, which allows you to tweak some SVG style properties online.

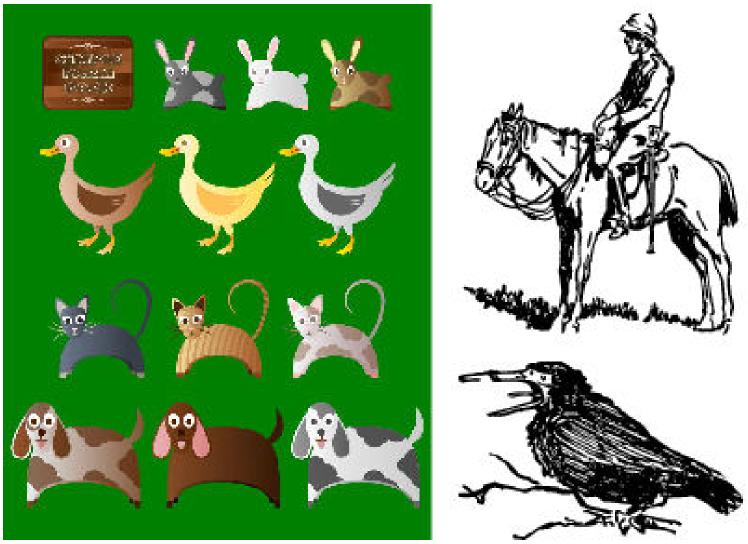


Figure 2-1. Samples from the Open Clip Art Library: on the left, Simple Farm Animals 2 by user Viscious Speed; on the right, line drawings from Sir Robert Baden-Powell's 1922 book An Old Wolf's Favourites, converted to SVG by user Johnny Automatic

Pixabay
: Another stock art library, Pixabay includes photos, illustrations, and vector graphics. However, be aware that many vector graphics are stored in Adobe Illustrator format, and you would need software able to convert them to SVG for use on the Web. All images are released under the Creative Commons' public domain license; you are encouraged to "buy a coffee" for the artists by donating funds when you download.

Wikimedia Commons
: The media repository arm of Wikipedia, Wikimedia Commons compiles images, audio, and video in a wide variety of formats. All files are available under some sort of "copyleft" license; some require attribution or are restricted to noncommercial use or to use within similarly-licensed work. Detailed license information is available on each file's catalog page.

Wikimedia is actively pushing their contributors to use the SVG format for diagrams, clip art, icons, and other vector drawings because of its flexibility and ease of editing; their servers then automatically generate raster versions in various sizes. Although the tagging and cataloguing of files is often inconsistent, making searching a little difficult, there is plenty of great SVG content if you take the time to look around. Figure 2-2 displays selections from the SVG Botanical Illustrations category, including a labelled diagram; because SVG text is easily editable, the file is available with labels in many languages.

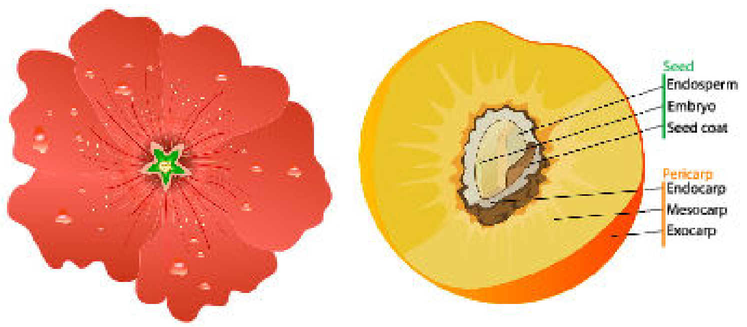


Figure 2-2. SVG from Wikimedia Commons: on the left, a hollyhock flower by user Ozgurel; on the right, a labelled diagram of a peach by Mariana Ruiz Villarreal (aka LadyofHats)

Iconic

Iconic is a commercial SVG icon library, but they offer a set of more than 200 icons completely free to use (MIT license; just ensure that license information is available with the code). This Open Iconic set includes most common user interface buttons in single-element icons that you style to any color you chose. For their paid offerings, Iconic distinguishes themselves by taking full advantage of all the possibilities of SVG, supporting multicolor styling and using scripts to substitute in more detailed versions of the icons at larger sizes. They even brag about their easy-to-read (and modify) XML markup.

The Noun Project

Another icon-focused library, the Noun Project's goal is to create a visual language for clear international communication. Access to their entire library is by monthly subscription, but their database includes many public domain and Creative Commons-licensed icons, searchable by concept using tags in dozens of languages.

There are, of course, numerous other sources for both free and for-sale SVG graphics; this is just a tiny sample. Adding the word "SVG" to your query in any search engine can help target your results; Google's Advanced image search also allows you to explicitly filter for SVG files. The difficulty is then to find the original source of the graphic to determine licensing rights.

The vast majority of ready-to-use SVG files are static content—images created by Inkscape, Adobe Illustrator, and other similar tools that contain neither scripts nor animation. A few clip art and icon libraries are starting to include animated SVG, while specialized applications such as mapping may include interactive features or links between files.

TIP

Typically, SVG in the wild is stored as text-based XML files. However, the SVG standards allow for gzip-compressed SVG—typically having an *.svgz* filename extension—to reduce the file size. This is common for high-quality, photorealistic SVG files, which can occasionally get to be bigger than their PNG counterparts; and for maps and other complex charts that embed a lot of metadata within the graphics themselves. It should also be used by a performance-minded web developer (that's you!) to compress a file for transmission from web server to browser.

Ready-to-use graphics can solve many web design problems. For common interface icons, creating your own graphics would seem like reinventing the wheel.

For other projects and other graphic types, stock art just won't do. You need to create a custom image that perfectly represents a new and unique concept. It takes a little more artistic skill, but there are plenty of tools for creating your own SVG art. After all, that's how most of the graphics in the above libraries were created in the first place.

Click, Drag, Draw

Once upon a time, one of the biggest issues facing adoption of the Scalable Vector Graphics standard was the lack of decent tools for creating SVG graphics or applications. Most SVG needed to be created by hand, or if that was too daunting, to be converted from other proprietary graphical standards. This reliance on converted graphics meant that the full features of SVG weren't always used.

On the other side, many vector graphics editors include features that are not part of standard SVG. To ensure that this extra information or features are retained when you save and reload the SVG (a process called "round-tripping"), these programs either have separate, proprietary image formats (as for Adobe Illustrator) or add extra markup to the SVG file (Inkscape). In order to create graphics that will display consistently with other software, these programs also include commands that will "flatten" the extra features into standard SVG 1.1. If your intent is to make content available for the Web, always ensure that you save a version of your graphic in standard SVG. Be aware that sometimes the multiple representations of the image may add significantly to the file size.

There are now numerous graphical software programs that can export to SVG. They all include visual, what-you-see-is-what-you-get (WYSIWYG) editors where you can position shapes with your mouse (or other pointing device) and select colors from visual palettes. They differ in how much of SVG's features they support, and in how easy they are to use.

Some vector graphics programs you could consider include:

Adobe Illustrator

Adobe Illustrator is the granddaddy of vector graphics programs, and debuted in 1991 as part of a seminal period in Adobe's history. It not only set the expectations of what a vector graphics program should look like, but has consistently been at the cutting edge of vector graphics support for the past two decades. Many aspects of SVG were inspired by the capabilities of Illustrator, and Illustrator has long supported export of its graphics to SVG format. However, it's definitely worth remembering that SVG is not a native format for the application (the *.ai* format is). What that means is that Adobe has to perform a translation from their internal vector graphics format (built primarily around Postscript) to SVG. For comparatively simple graphics, this is a straightforward proposition, but it is possible to create Illustrator images that have poor fidelity and large file sizes when rendered to SVG, with complex effects replaced by bitmap images.

The basic save-as-SVG option in Illustrator creates a complex file from which the native graphic can be reconstructed. However, in 2015 Adobe introduced a much more streamlined export-as-SVG option that creates a graphic optimized for the Web. You can also copy individual graphics components from the Illustrator interface and paste them into a text editor; the corresponding SVG code will be pasted. This is useful if you're building a complex application or animation with SVG, but want to use the visual editor to draw shapes.

Inkscape and Sodipodi

Sodipodi was one of the earliest SVG editors, initially developed for Linux systems. It drew its inspiration from Adobe Illustrator, but used SVG natively to store and retrieve content. Inkscape started as a branch of Sodipodi, and is now the more actively developed program.

Inkscape has matured into a remarkably sophisticated, feature-rich vector graphics application, while never losing sight of its SVG roots. Its interface (Figure 2-3) is somewhat crowded with features, but with a little effort to learn all the options, it allows for considerable control over the graphic. In addition to supporting most static SVG features, it includes numerous filters and extensions to create graphical effects. There are also controls that allow you to edit nongraphical aspects of the SVG, such as element IDs, alternative text, and even simple JavaScript event handling. You can also inspect the XML file and edit nodes and attributes directly.

Inkscape implemented some features of SVG 1.2, particularly multiline text boxes, which were never widely supported by browsers; be sure to convert your text boxes to fixed-position SVG text when exporting for the Web. Other Inkscape-specific markup is removed by saving as a “Plain SVG” format. Export options (set under the “SVG Output” tab in software preferences) can help optimize the final file, stripping unused style and attributes or simplifying the data used to encode complex paths.

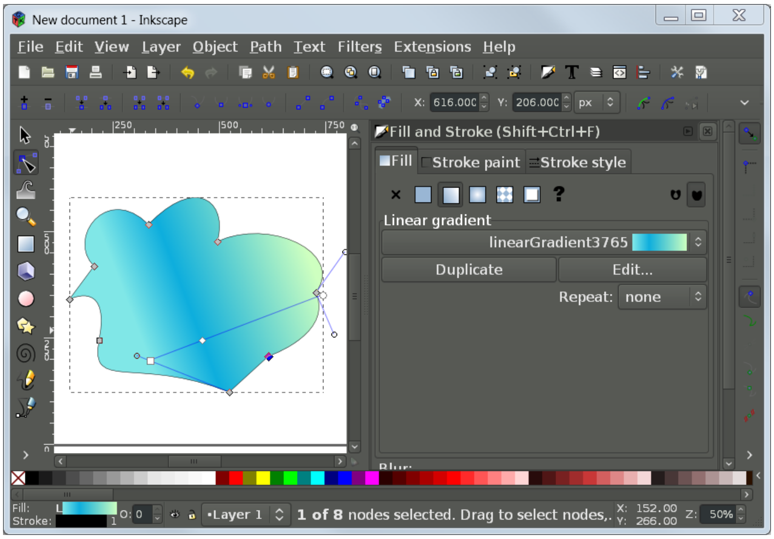

New document 1 - Inkscape
File Edit View Layer Object Path Text Filters Extensions Help
X: 616.000 Y: 206.000 px
Fill and Stroke (Shift+Ctrl+F)
Fill Stroke paint Stroke style
Linear gradient
linearGradient3765
Duplicate Edit...
Repeat: none
Blur:
Fill: Stroke:
O: 0
•Layer 1
1 of 8 nodes selected. Drag to select nodes,.
X: 152.00 Y: 266.00 Z: 50%

Figure 2-3. The open source Inkscape graphics editor

Microsoft Visio

Although not a general-purpose graphics editor, Microsoft's Visio software is used for designing charts and diagrams. SVG is remarkably well-suited for charts, and Visio provided the setting for Microsoft's first foray into SVG (as an alternative export format). It turned out to be one of the more favorably received of features for this product, as it allowed diagrams to be used outside of the normal Microsoft family of products, including on the Web.

LibreOffice and Apache Open Office

These two open-source office software suites—which share a common history but are now developed separately—both include support for vector graphics, either embedded in text documents or as stand-alone drawings. They use a conversion program (based on Batik, which is discussed later) to translate between SVG and the internal vector graphics format. The conversion is not perfect, and won't support advanced SVG features. However, these drawing programs can be user-friendly introductions to drawing basic vector graphics.

Google Docs

Google's web application alternative to desktop office software, Google Docs, uses SVG natively to encode and display drawings. Furthermore, because the SVG is being displayed live in your web browser, you are seeing it exactly as it will be displayed in a website, and you can open the file in multiple browsers to confirm consistent rendering. The interface is easy to use, but it only supports a basic set of SVG features, in order to support conversion to the drawing formats used by other office software.

SVG-edit

Another online SVG graphics application originally sponsored by Google, SVG-edit runs in your web browser either from a downloaded file or directly from the web server. In addition to most of the standard visual vector graphics options, you can easily set element `id` and `class` attributes from the editor, add hyperlinks, and can define dimensions and a title for the document. Unfortunately, at the time of writing the program is not well documented and is somewhat prone to errors when editing complex content. Development and issues can be monitored via the GitHub repository.

Draw SVG

A more complete (and more completely documented) online SVG graphics editor is Draw SVG by Joseph Liard. It implements nearly all the commonly supported SVG drawing and styling features (no filters or animation). The dialog forms that control style properties use standard SVG terminology for attributes and style properties, which is helpful if you will be alternating between using a graphics editor and writing the code yourself. The application also offers tools to create rasterized versions of the SVG, or to encode raster images as data that you can embed in the SVG itself.

The performance of the web application itself can be slow compared to desktop applications like Inkscape, and drawing complex shapes is difficult. The tool would likely be most useful for customizing the styles on icons and clip art from other sources, especially if you aren't yet comfortable writing the markup and stylesheets yourself. Figure 2-4 shows the interface.

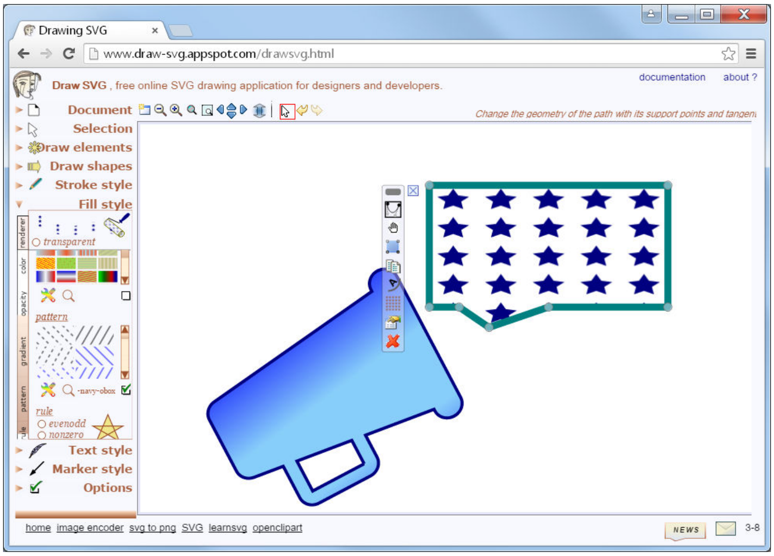

Drawing SVG
www.draw-svg.appspot.com/drawsvg.html
Draw SVG , free online SVG drawing application for designers and developers.
documentation
about ?
Document
Selection
Draw elements
Draw shapes
Stroke style
Fill style
transparent
pattern
rule
evenodd
nonzero
Text style
Marker style
Options
Change the geometry of the path with its support points and tangent
home image encoder svg to png SVG learnsvg openclipart
NEWS

Figure 2-4. The Draw SVG free online SVG graphics editor

Nearly all of these SVG editors have some ability to convert SVG graphics to raster images or other vector formats. This can be useful to create fallbacks for old browsers or to create consistent rendering in print publications. However, manually saving files in multiple formats from a graphics editor can be tedious. On many web server setups, the process can be automated using dedicated rasterization and conversion tools.

SVG Snapshots

An SVG rasterizer is a program that converts the vector graphics elements into a visual bitmap format. Broadly speaking, any program that can display an SVG on a screen is a rasterizer. However, the programs described here are single-purpose rasterizers, used when incorporating SVG into print or when generating fallback alternatives for older browsers. They are command-line programs or software libraries suitable for inclusion in automated routines.

Batik
: The Apache Batik project is a complete implementation of SVG 1.1 in Java. The project's rasterizer utility has traditionally been

used to render SVG in publishing pipelines, most typically to convert XSL Formatting Objects (XSL-FO) documents into printed output. In general, the quality of the Batik rendering is quite high, and is particularly appropriate for generating images in raster formats such as PNG or JPEG from SVG graphics. Batik supports nearly all features of SVG 1.1, but has not (at the time of writing) implemented CSS3 features that you might wish to use in SVG for modern web browsers.

Once downloaded, Batik can be run as a Java archive file. The static renderer is specifically *batik-rasterizer.jar*, part of the Batik distribution. There are a number of options for controlling output file format, width and height, area, quality, output directory, and so forth. Invoking the rasterizer without arguments should give you the list of options.

LibRSVG

The LibRSVG library is part of the Linux Gnome project, and provides support within Gnome for static SVG images as well as a library that lets developers generate SVG in their programs. It can also be run as a standalone program called *rsvg* to generate PNG or JPEG images from SVG files. It supports core vector graphics, but not advanced effects. The LibRSVG rendering engine is used in numerous other open source SVG tools.

ImageMagick

ImageMagick is best described as a Swiss army knife for command line manipulation of graphics, and is available on Windows, Macintosh, and Linux platforms. It is useful from a command line and also can be invoked from libraries in most modern processing languages, from C++ to Python, PHP, Ruby, and Node.js. Given this, it's not surprising that it supports rasterization of SVG.

At its simplest, the ImageMagick convert command is trivial:

```
convert image.svg image.png
```

This converts the file indicated by the first argument from an SVG image to a corresponding PNG image. When it is available, ImageMagick will use Inkscape's command-line interface to render SVG; in that case, it will support most SVG 1.1 features. As a second effort, ImageMagick will try to use LibRSVG. If that is not available, ImageMagick has its own rendering tools; these have less support for advanced features such as filters and stylesheets. It is generally advisable to experiment with sample SVG images to see whether ImageMagick will meet your needs.

CairoSVG
: Cairo is a complete vector graphics languages as a programming library for use by other software; it has implementations in many common programming languages such as C++ and Python. Cairo graphics can be converted to vector files as PostScript, PDF, and SVG; can be output on various screen display modes on Linux and Macintosh systems; or can be used to generate raster images. The CairoSVG library, from the web design company Kozea, parses SVG files and converts them to Cairo graphics—the result can be used to generate PDF, PostScript, or PNG versions of the SVG files. Most basic vector graphics features from SVG 1.1 are supported.

As you may have gathered, a limitation of all these rasterization tools is that they do not keep up to date with the latest developments in other web platform specifications—if they even support the full SVG standard to begin with.

A final option for creating rasterized versions of SVG files is to use an actual web browser rendering engine. To do this from a server routine or other command-line interface, you can use the PhantomJS implementation of the WebKit browser engines. The sample *rasterize.js* script can be used to convert any web page or SVG file to PNG or PDF. PhantomJS can also run your own JavaScript code, such as a script to build an SVG visualization from a data file, and then save the resulting SVG document.

With all these options for converting SVG to raster image formats, you may wonder about the reverse conversion. Can you create an SVG from a PNG or JPEG? That gets much more complicated. Although SVG code contains information about how the shapes should look, raster images don't contain information about the shapes they were constructed from.

Tracing or vectorizing tools use algorithms to try to calculate those shapes from the pixel data in a raster image, looking for high-contrast edges, then connecting them into smooth lines and curves. The more comprehensive graphics programs, such as Illustrator and Inkscape, include auto-tracing tools. There are also specialized tracing tools such as Vector Magic. These can be particularly useful if you want to draw graphics by hand, then scan them into your computer and convert to SVG.

Bringing SVG Alive

Rendering SVG to other static file formats is useful, but the reason you're using SVG in the first place is because it is so much more than a rasterized image. To see and use the full power of SVG, you need a dynamic SVG viewer that can update the graphic according to user interaction or timed animations.

When discussing web browser support for SVG, it helps to group the browsers according to the rendering engine they use. Many of the engines are open-source, and there are numerous other tools that use the same code, and therefore display web pages and SVG in the same way (although these tools may not always be up-to-date with the latest features). Knowing the rendering engine also helps you know which prefix to use when testing experimental CSS features, although CSS prefixes are going out of fashion in favor of experimental browser modes controlled by the user.

The main rendering engines are as follows:

Gecko for Firefox

The first web browser implementation of SVG was built within the Gecko rendering engine in 2003. Gecko, originally built for Netscape 6, is the basis of the Mozilla Firefox browser as well as numerous niche browsers and tools.

The original SVG implementation was basic, focusing on simple vector shapes. However, it has expanded steadily and continues to improve. Until recently, dynamic SVG could be slow and jerky in Firefox; however, over the course of 2014 significant performance improvements were made and some animations are now smoother in Firefox than in other browsers.

There are still some areas where Firefox/Gecko does not conform to the SVG specifications in the finer details, particularly around the way `<use>` elements are handled. They also did not implement many of the style properties that offer nuanced control of the layout of SVG text; some of these features are now (early 2016) being implemented in coordination with enhancements to CSS-styled HTML text. SVG rendering may also differ slightly between operating systems, as Firefox uses low-level graphical rendering tools from the operating system to improve the performance of some actions.

Experimental CSS features for Gecko used the `-moz-` (for Mozilla) prefix; however, the development team has phased out the use of prefixed experimental CSS for new features.

WebKit for Safari and iOS devices

Apple's Safari browser was built upon open-source rendering and JavaScript engines originally created for the KDE operating system (for Linux/Unix computers). Apple's branch of the code —known as WebKit—is used in all Apple devices and was also originally the basis for the Google Chrome browser, among many other tools. As previously mentioned, WebKit is also used in the PhantomJS browser simulator.

WebKit implemented most SVG 1.1 features between 2008 and 2010; many edge cases or areas of poor performance remain, but for most purposes it is a complete implementation. Many CSS3 features require a `-webkit-` prefix on Safari and related software.

Blink for newer versions of Chrome, Opera, and Android devices

In 2013, Google's Chromium project announced that they would no longer synchronize further development with the WebKit project. The Google Chrome browser at that point used WebKit code to render web pages (and SVG) but had separate code for other functions including JavaScript processing.

The branch of the rendering engine, developed as part of the Chromium project, is now known as Blink. In addition to being used by Chrome, Blink is used in the Opera browser (since version 13) and some native applications on newer Android devices.

Blink browsers still support `-webkit-` CSS properties, but they have been transitioning to using standard (unprefixed) syntax. Users can opt in to experimental features through advanced browser settings.

The development of the Google Chrome browser (and now Blink in general) has been heavily focused on performance; their SVG implementation is one of the best, and animations are generally fast and smooth (although Firefox has recently caught up). Some edge-case features are not supported, particularly in areas where the SVG specifications work differently from CSS and HTML. Blink has removed support for SVG Fonts from most platforms and the development team has indicated that they consider SVG animation elements (SMIL animation) to be deprecated. At the time of writing (early 2016), animation elements still work, but a warning is displayed in the developer's console.

Presto for older Opera versions and Opera Mini
: The Opera browser previously used its own proprietary rendering engine, known as Presto. It is still used for server-side rendering for the Opera Mini browser, converting web pages to much simpler compressed files for transmission to mobile devices with low computing power or expensive and slow Internet connections. In Opera Mini, SVG is supported as static images, but not as interactive applications.

  Presto supports nearly all of the SVG 1.1 specifications and some CSS3 properties. However, it has not been (and will not likely be) updated since 2013. Presto versions of Opera used an `-o-` prefix for experimental CSS features.

Trident for Internet Explorer and other Windows programs
: Internet Explorer was the last major browser to add support for SVG. Prior to the development of the SVG standard, Microsoft had introduced its own XML vector graphics language (the Vector Markup Language, VML), used in Microsoft Office software and supported in Internet Explorer since version 5.

  Basic SVG support was introduced (and VML phased out) with Internet Explorer version 9 in 2009. Support for additional SVG features, such as filters, was added in subsequent versions. Nonetheless, older Internet Explorer versions that do not support SVG (particularly Internet Explorer 8) continue to be used because newer versions of the software are not supported on older Windows operating systems. As of the end of 2015, slightly more than 1% of global web traffic used Internet Explorer 8, a two-third drop from a year previous but still a meaningful share for large commercial websites.[1]

1 Data from *http://caniuse.com/usage_table.php*.

As of Internet Explorer 11 (the final browser to use the Trident engine), there were a number of small quirks and bugs in SVG support, and some features that were not supported at all. The main area where Internet Explorer does not match the other web browsers is animation: there is no support for either SVG animation elements or CSS animation applied to SVG graphics. Another key missing feature is the `<foreignObject>` element, which allows XHTML content to be embedded in an SVG graphic.

The Trident rendering engine used for Internet Explorer is also used in other Microsoft programs and by some third-party software built for Windows devices. It uses the `-ms-` CSS prefix.

EdgeHTML for Microsoft Edge and Windows 10+ programs

The Microsoft Edge browser developed for Windows 10 uses a new rendering engine, built from a clean codebase to emphasize performance and cross-browser interoperability. The EdgeHTML engine is also used by other software in Windows 10.

Edge supports all the web standards supported in Internet Explorer, and many new ones. Collaboration from Adobe developers helped advance support for a number of graphics and visual effects features. Support for SVG `<foreignObject>` and CSS animations of SVG content have already been introduced, and the development team has indicated that they intend to implement many other SVG 2/CSS3 features. However, plans to eventually support SVG animation elements were shelved after the Chromium project announced their deprecation plans. Edge supports `-ms-` prefixed properties that were supported in Internet Explorer, and has also introduced support for some `-webkit-` prefixes that are commonly used in existing websites.

The browser SVG implementations (with the exception of Presto and Trident) are the subject of active development. Feature support may have changed by the time you read this. Consult the version release notes or the issue-tracking databases for the various browsers to determine if a specific feature is now supported.

Future Focus
Hardware Acceleration

When discussing SVG support, it is easy to focus on the specific elements, attributes, or style properties that software does or does not recognize. When discussing dynamic SVG, however, the efficiency of the implementation is equally important. Slow rendering speeds can cause the image to flicker and jerk when it is animated by scripts or declarative animation commands.

An ongoing area of development and improvement in dynamic SVG is hardware acceleration, using the computer display's video card to improve rendering speed. The video cards on most modern computers can combine different partially transparent image layers directly in their GPU (graphical processing unit, a specialized processor chip or chips). If the browser can divide the web page into independent layers, then changes in the opacity or relative position of the layers can be processed quickly and smoothly at the GPU level.

This level of basic acceleration operates on web content that has already been converted to rasterized forms. However, the accelertion of vector graphic calculations is an important area of new development. The Khronos Group, an industry consortium, has established a set of standards for hardware acceleration of graphics in 2D and 3D. Among their projects is the OpenVG standard, which provides a number of core libraries and hardware-standard calls for accelerating vector graphics language processing in chips.

This support has in turn made its way into graphics chips and low-level libraries produced by such vendors as nVidia, Apple, Adobe, Creative, Google, HTC, Intel, ARM, Ericsson, Nokia, Qualcomm, Samsung, Sony, and many others. Such chips usually include a vector graphic processor in addition to the 3D processors more commonly associated with graphics chips, and it is the presence of these libraries that has resulted in a significant improvement of vector graphics in general in the last few years.

This doesn't necessarily translate into full SVG support—the libraries are too low-level for that—but it does effectively mean that hardware acceleration for animation and interactivity is now available to a far wider variety of devices than held true five years ago, including many of the mobile, smartphone, and smart pad devices that are emerging today as the front-runners of the next wave of computer innovation.

In addition to the web browsers, there are a two other dynamic SVG rendering engines that have been important in the development of SVG:

Adobe SVG Viewer
: As mentioned in Chapter 2 of *Using SVG with CSS3 and HTML5*, the Adobe SVG Viewer—a plug-in for Internet Explorer—was one of the first and most complete SVG environments. Although it has not been developed for years, it can still be downloaded to enable SVG support on older Internet Explorer browsers. In order to trigger the plug-in, the SVG must be included in the page using either an `<object>` or an `<embed>` tag.

Batik Squiggle Viewer
: We mentioned the Apache Batik project in the context of SVG rasterizers; however, the rasterizer is only one part. Batik can be used to generate and display SVG in other Java-based software. It also comes with its own dynamic SVG viewer called Squiggle for viewing SVG files from your computer or the Web.

  Squiggle can display SVG animation and can process JavaScript and respond to user events, including following hyperlinks to new files. Batik supports nearly all of the SVG 1.1 specification, but has not been updated for more recent CSS, DOM, and JavaScript methods. It can also be more strict, compared to browser implementations, about requiring common values to be explicitly specified in markup and in scripts.

These dynamic SVG viewers do not merely display an image of the SVG, they present changing, interactive SVG documents. In order to create such a document, you'll need to use more than the graphical editing programs presented in "Click, Drag, Draw" on page 21. You'll need to look inside the SVG, and work with the underlying code.

Markup Management

It is possible to write SVG code in any editor that can save in a plain text format. You can open up Notepad or something similar, type in your markup, scripts, and styles, save it with a *.svg* extension, then open the same file in a web browser.

If you typed carefully, and didn't forget any required attributes or misspell any values, your SVG will appear on screen, ready to be used just as you intended. However, if you're human, chances are—at least some of the time—you'll end up with XML validation errors displayed on screen, with JavaScript errors printed to the developer's console, or simply with a graphic that doesn't look quite like you intended.

Text editors designed for writing code can help considerably. They can color-code the syntax so it's easy to detect a missing quotation mark or close bracket. They can also test for major syntax errors before you save. Many can autocomplete terms as you type. The options for code editors are too numerous to list here; many are available only for specific operating systems. Whatever you choose, be sure to confirm that the editor has—at a minimum—syntax rules for XML, or more preferably specific rules and hints for SVG.

Nonetheless, even the best syntax highlighting and code hints cannot help you *draw* with code. When working with complex shapes and graphical effects, it really helps to be able to see the graphical effect of your code as you write it.

Oxygen XML SVG Editor

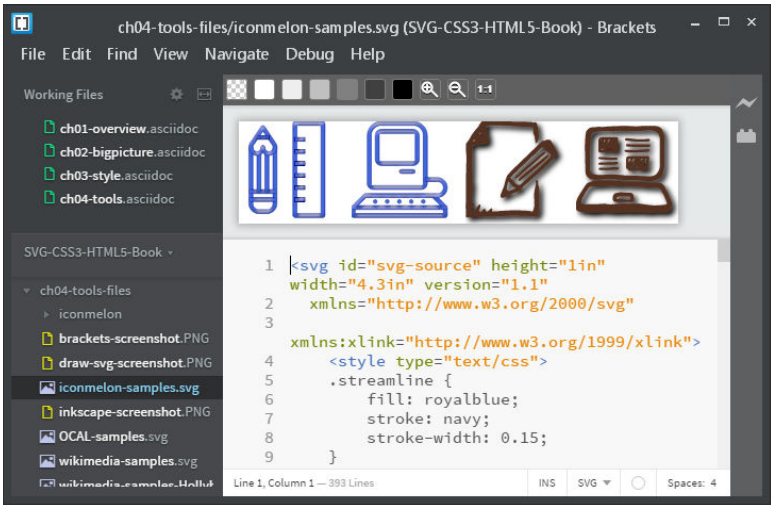
A commercial XML management program, Oxygen allows you to handle many types of XML-based data and formatting tools. The SVG editor uses Batik to render graphics, and can render both SVG markup and SVG created via scripts. It is intended primarily for creating SVG as the result of an XSLT (eXtensible Stylesheet Language Transformation) template applied to an XML data file, but can also be used for plain SVG.

Brackets plus SVG Preview

A code editor developed by Adobe primarily for web developers, Brackets includes a feature whereby you can open the web page you're working on in a browser and have it update as you type in the main editor. At the time of writing (Brackets version 1.1), this only works with HTML and CSS; SVG inline in those pages is displayed, but not updated live. For standalone SVG files, the SVG Preview extension will display a live version of the SVG in the editor as you type; however, this should currently only be used for static SVG images, as script errors in your code can crash the editor.

The SVG Preview feature—and the entire Brackets editor—uses the Blink rendering engine. The preview image is currently displayed as inline SVG code within the HTML 5 application code; this means that minor syntax errors and missing namespaces in a half-finished file do not break the preview. However, it also means that inline scripts can wreak havoc, and external files and stylesheets are not supported. Figure 2-5 shows the editor with a live preview of SVG icons.

Both the core Brackets code and extensions are being rapidly developed; however, as of early 2016 progress on SVG-focused features appear to have stalled. Adobe is also developing software (Adobe Extract) to allow users of their commercial design software (e.g., Photoshop) to easily generate matching web code in Brackets; however, at the time of writing this tool primarily focuses on CSS and does not include any SVG- or Adobe Illustrator-related features.



```
<svg id="svg-source" height="1in"
width="4.3in" version="1.1"
  xmlns="http://www.w3.org/2000/svg"

xmlns:xlink="http://www.w3.org/1999/xlink">
    <style type="text/css">
    .streamline {
        fill: royalblue;
        stroke: navy;
        stroke-width: 0.15;
    }
```

Line 1, Column 1 — 393 Lines INS SVG Spaces: 4

Figure 2-5. The Brackets code editor with SVG Preview enabled

Online live code sites

In recent years, numerous web applications have become available that allow you to write web code and see its output in separate frames of the same web page. Because the result is displayed right in your web page, you can use all the latest features supported by your web browser. Most make it easy to import common JavaScript code libraries. However, since you don't control the web server, other external files can often be limited by browser security restrictions.

All these sites currently work with HTML 5 documents, including inline SVG elements. As with the Brackets SVG preview, this means that they are more forgiving of syntax errors than SVG in XML files. Some live code sites worth mentioning include:

- JSFiddle was one of the first sites to offer live execution of web code that you can save to a publically accessible web link and send to collaborators or reference from help forums. The stripped-down interface is best for small test cases and examples.
- CodePen is a more full-featured live code site that also serves as a social media network for web designers; you can explore other coders' work, leave comments, or publish a blog with multiple embedded working examples in each post. A paid membership opens up additional collaboration tools and the ability to upload separate image files, scripts, or other resources.
- Tributary is specifically designed for data visualizations and other scripted SVG. By default, it provides you with a blank HTML page containing a single inline `<svg>` element that you can manipulate with JavaScript. You can also create separate data files accessible from the main script. The interface offers convenient tools such as visual color pickers and GIF snapshots (including animation) of your work.

When you're working on these sites, keep in mind that saving your work usually also means publishing to the Web. Some sites, such as CodePen, automatically apply a very-few-rights-reserved license to work published in this way (you can save work privately and control copyright with a paid CodePen membership).

Once you have tools that allow you to rapidly write and test your code, it becomes easier to think about SVG programmatically. Working with the code forces you to consider the graphic from the perspective of the document object model rather than simply from the perspective of its final appearance.

This approach is particularly useful when dealing with dynamic and interactive SVG and when creating data visualizations. If those areas interest you, the next set of tools you'll want to investigate are those that make manipulating the DOM easier.

Ready-to-Use Code

There are two ways to create an SVG by writing code: writing out the XML markup and attributes, or writing JavaScript to create the corresponding DOM elements dynamically. Scripting is preferred when you have a lot of similar elements or when the geometric attributes should be calculated based on a data file. This book uses both approaches in the examples.

NOTE

There's actually a third way to code SVG (which we mentioned briefly when discussing the Oxygen XML editor): using an XSLT stylesheet applied to an XML data file.

The XSLT stylesheet is an XML file. It consists of SVG markup templates interspersed with formatting instruction elements that indicate how the data should be processed and inserted into the SVG file. XSLT is therefore another way to create SVG that should correspond with underlying data. However, unlike with scripting, the XSL transformation can only be applied once, when the file is processed; it cannot update with new data or respond to user interactions. With standardized JavaScript being well supported and efficiently implemented in browsers, the use of XSLT to generate SVG is falling out of favor.

The popularity of using JavaScript for the creation and manipulation of SVG has much to do with the availability of open-source tools to make this easier. These libraries of JavaScript code provide shorthand methods to perform common tasks, allowing your script to focus on the graphics instead of on the underlying DOM function calls. The following JavaScript libraries are particularly important when working with SVG:

Raphaël and Snap.svg
The Raphaël library by Dmitry Baranovskiy was important in getting dynamic SVG into production web pages. It provides a

single interface that can be used to create either SVG graphics or Microsoft VML graphics, depending on which one the browser supports. The library is therefore essential if you want to provide dynamic vector graphics to users of Internet Explorer 8. The number of features Raphaël supports, however, is limited to the shared features of the two vector graphics languages.

The terminology used by Raphaël includes a number of convenient shorthands that do not always directly correspond to the standard SVG element and attribute names. The same terminology is used in the newer Snap.svg library, produced by Baranovskiy through his new employer, Adobe. Unlike Raphaël, Snap.svg does not include support for VML graphics. This allows the library code files to be smaller, and allows support for features such as clipping, masking, filters, and even groups, which aren't supported in VML. Snap can also load in existing SVG code, in order to manipulate complex graphics created in WYSIWYG editors. Both Snap and Raphaël have convenient functions to create smooth JavaScript animations, allowing you to animate graphics in any version of Internet Explorer.

D3.js

The D3.js library, originally developed by Mike Bostock, has become the standard tool for creating dynamic SVG data visualizations. D3 is short for *Data-Driven Documents*, and it reflects how the library works by associating JavaScript data objects with elements in the document object model (DOM).

The core D3 library is open-ended, allowing you to manipulate groups of DOM elements (SVG or HTML) simultaneously by defining how their attributes and styles should be calculated from the corresponding data objects. Changes in values can be set to smoothly transition over time to create animated effects.

D3 includes a number of convenient functions for calculating the geometric properties of common data visualization layouts, such as the angles in a pie graph. It also includes SVG-specific convenience functions for converting data and geometrical values into the actual instructions you'll use in the attributes of an SVG `<path>` element. However, D3 does not draw the charts directly; many extensions and add-ons have been developed to make it easier to draw common chart types.

GSAP

An animation-focused commercial library, the GreenSock Animation Platform focuses on making animated HTML and SVG content fast, smooth, and cross-browser compatible. The GSAP library can be freely used on many commercial projects (and most noncommercial ones); a paid license is required if the website's end users pay subscription or other fees, or to access various extra plug-in scripts. A number of those plug-ins are specifically focused on working with SVG paths, or circumventing browser support issues at the intersection of SVG and CSS3.

Learning to use these JavaScript libraries is worth a book of its own (and many great books are available). However, they don't replace an understanding of the underlying SVG graphics. It's difficult to effectively manipulate SVG with scripts unless you already know what SVG is (and isn't) capable of.

Processing and Packaging

You have your SVG ready to go, whether it came from a clip art library, was drawn in a graphics editor, or was carefully written as code. There are still a few tools that you may want to use while uploading SVG to your web server. A sample routine, which could be automated, would be to:

1. Run your SVG code through an optimizing tool such as SVGO or Scour to eliminate extra markup from graphics tools and to otherwise condense the file (being sure not to use any settings that will remove IDs or classes that you'll use in scripts or stylesheets).
2. Generate raster fallback images for Internet Explorer 8 and Android 2.3 users (using any of the rasterization tools mentioned in "SVG Snapshots" on page 25).
3. Compile a folder full of all individual SVG icons into a single file that can be sent to the user all at once (the SVGStore Grunt plugin does this on a Node/Grunt server configuration).
4. Use gzip compression to further reduce file size (being sure that your server is set to correctly indicate the compression scheme in the HTTP headers sent with the file).

There are almost certainly many more tools and techniques that can be used, depending on how your website and server are set up, and on how you intend to use the SVG. These examples should get you started.

Summary: Software and Sources to Make SVG Easier

The purpose of this chapter hasn't been to tell you what software or which websites to use, although hopefully it has given you some suggestions if you did not know where to start.

More importantly, the review should have helped you understand the diversity of ways you can create and use SVG. Furthermore, it should have helped remind you of the compatibility issues that must always be kept in mind when working on the web. And finally, it should have helped you get some SVG files onto your computer—whether downloaded from a clip art library or created yourself in an editor—that you can experiment with as you work through the rest of the book.

This chapter has been revised many times since it was first started in 2011, in part due to the dramatic changes in the SVG software landscape. It will surely be the first chapter in the book to become obsolete. In the past few years, SVG has in many ways become a *de facto* standard for maps and information graphics on the Web, is becoming a commercially viable alternative for clip art, is making its way into graphics usage for component diagrams of everything from houses to aircraft to cars, and is factoring into web interfaces (and even operating system interfaces) in subtle but increasingly ubiquitous ways.

While the example sites and applications given here are a good start, other places to find out more about SVG include Infographics and Data Visualization Meetups, code libraries such as Google Code Projects, or online forums on LinkedIn or Google+ (both of which have a number of active SVG and data visualization groups).

As you're following along with the rest of the book, feel free to use downloaded clip art or SVG you created in a graphics program to experiment with styles and effects. It's what you'll often do in practice. Opening the files in a code editor that highlights the syntax (and particularly one that can "tidy up" or "pretty print" the XML) can help you identify the core structure of group and shape elements in the code. From there, you can add new attributes or CSS classes.

CHAPTER 3

Beyond Straight Lines

From SVG Text Layout

Baselines ensure that glyphs are positioned to create a pleasing line of text. However, we've already said that, in graphical layout, you don't always *want* text to display in perfectly straight lines. Sometimes it's fun to make text move out of those boring lines and into more complex curves—circles, spirals, around the edges of various objects, and so forth.

This chapter introduces the `<textPath>` element, which allows you to use SVG path geometry to describe complex text layouts.

Creating Curved Text

We've shown (in Chapter 5 in *SVG Text Layout*) how you can position and rotate individual characters. For some simple designs, that's enough. Example 3-1 spaces the letters of the word "Sunrise" in a semicircle, each letter positioned every 30° and rotated to match, as shown in Figure 3-1.

Example 3-1. Arranging letters around a shape with character position attributes

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xml:lang="en"
     width="4in" height="2.3in" viewBox="0 0 400 230">
    <title>Text Positioned in a Curve</title>
    <style type="text/css">
        text {
            font: bold italic 48px "Times New Roman", Times, serif;
            fill: gold;
            stroke: orangeRed;
        }
    </style>
    <rect fill="#CEE" width="100%" height="100%" />
```

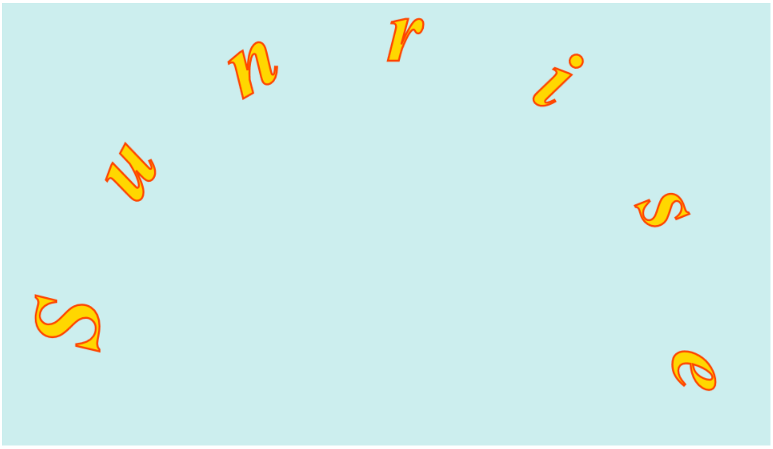


Figure 3-1. Curved text positioned with x, y, and rotate

```
    <g transform="translate(200,180)">                           ❶
        <text x="-150 -130 -75 0 75 130 150"
              y="0 -75 -130 -150 -130 -75 0"
              rotate="-90 -60 -30 0 30 60 90"
              >Sunrise</text>                                   ❷
    </g>
</svg>
```

❶ To make the trigonometry *slightly* easier, the coordinate system origin is translated to the center of the semicircle.

❷ There are seven letters in "Sunrise," so there are seven values in each of the positioning attribute lists.

The exact positions of each letter required a little bit of trigonometry to calculate. Even then, it doesn't look quite right: because the letters *start* at the specified anchor point, the final "e" sticks out below the starting "S" even though their anchors are on the same horizontal line. Using `text-anchor: middle` doesn't help; it centers each letter over the anchor point *before* rotating them, so they end up shifted to the side, not shifted around the circle.

If we had more letters, we'd need more trigonometry, and longer lists of `x`, `y`, and `rotate` attributes. And if we had enough letters that we wanted the word to look like a continuous string of text, we'd have to deal with the fact that each letter should be spaced differently according to its own unique dimensions. This clearly isn't a practical solution for pleasing text layout.

For cursive scripts such as Arabic, there's another problem with absolutely positioning letters: no matter how close each letter is to the next, they are rendered as isolated letters, not parts of a continuous word.

This is where text on a path comes in handy. Text on a path is exactly what it says—each letter is placed such that its baseline is on the tangent of a curve, spaced out along that curve according to the normal spacing of that text sequence.

> Although text on a path *should* be perfect for creating decorative layouts with cursive text, actual browser implementations are another matter, particularly for right-to-left scripts such as Arabic.

SVG text on a path is created with the `<textPath>` element. The content of the `<textPath>` is aligned along the outline of a separate `<path>` element. Which path to use is specified with an `xlink:href` attribute.

> TIP
>
> Just like a `<tspan>`, a `<textPath>` *must* be within a `<text>` element; it does not draw anything on its own.

Example 3-2 arranges the longer string "from Sunrise to Sunset" around a semicircular path (actually a cubic Bézier curve). The result is shown in Figure 3-2.

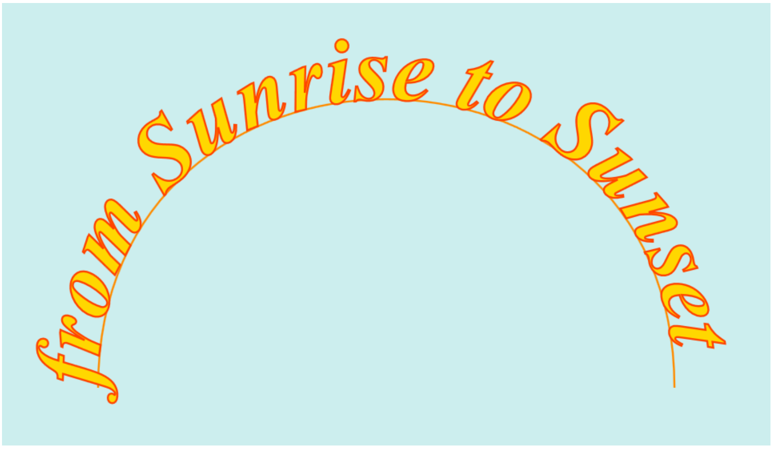


Figure 3-2. Curved text positioned along a path

Example 3-2. Arranging a text string around a shape with textPath

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xml:lang="en"
     width="4in" height="2.3in" viewBox="0 0 400 230">
    <title>Text on a Curved Path</title>
    <style type="text/css">
        text {
            font: bold italic 48px "Times New Roman", Times, serif;
            fill: gold;
            stroke: orangeRed;
        }
    </style>
    <rect fill="#CEE" width="100%" height="100%" />
    <path id="path" d="M50,200 C50,0 350,0 350,200"
          fill="none" stroke="darkOrange" />
    <text>
        <textPath xlink:href="#path">from Sunrise
            to Sunset</textPath>
    </text>
</svg>
```

The letters in Figure 3-2 are spaced much more smoothly than you could expect to achieve by placing each character yourself.

Blink/WebKit browsers currently render each letter within text on a path as if it was its own text chunk. This doesn't affect the spacing, but it has other consequences. For right-to-left scripts within a left-to-right layout, this means that the letters are not rearranged into the correct reading order. In cursive scripts, it also means that the isolated glyph forms for each letter are used instead of the word forms.

The path itself is stroked in Example 3-2, but it does not have to be; you can define the path within a `<defs>` section without drawing it at all. Here, we draw it to emphasize that the baseline of the text is aligned with the path. If you used a different baseline, the characters would move in or out relative to the curve.

Regardless of whether the path itself was drawn to the screen or not, the text will be positioned as if the path was drawn in the same coordinate system as the `<text>` element itself.

Positioning on a Path

The text string in Example 3-2 doesn't quite fit the full length of the path, making it appear slightly off-balance. A `text-anchor: middle` setting could center the text, but only if we can correctly position the anchor point. By default, it's at the start of the path. If we centered the text around that point, half of it would disappear off the start of the path.

TIP

Any text that extends beyond the length of the path—at the beginning or end—will not be drawn at all.

The `startOffset` attribute of a `<textPath>` element defines the position along the path at which the text should be anchored. It can be given as a length—measured from the normal start of the path—or as a percentage of the path's length. To center text within the path length, you can therefore use `text-anchor: middle` with a `startOffset` of 50%, as follows:

```
<text text-anchor="middle">
    <textPath xlink:href="#path" startOffset="50%"
        >from Sunrise to Sunset</textPath>
</text>
```

Figure 3-3 shows the much more balanced result of this change.

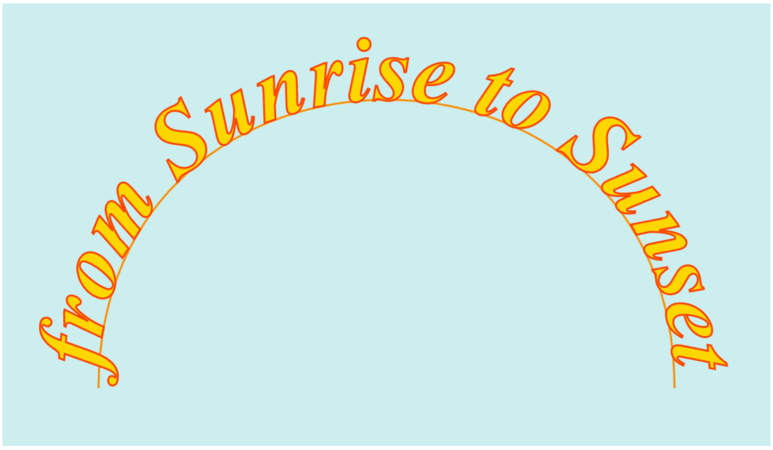


Figure 3-3. Curved text centered on a path

The `startOffset` attribute is particularly useful when arranging text around an existing shape, which might not have been drawn with the start of the text in mind. When animated, a changing `startOffset` can create a marquee effect, sliding text the length of the path.

! Blink/WebKit browsers currently treat negative `startOffset` values as 0, and do not draw any text with a `startOffset` greater than 100% of the path length, even when using an `end` value for `text-anchor`. This rather limits the potential for scrolling marquees of appearing and disappearing text.

On closed shapes (such as a complete circle), be aware that text will not continue from the end of the path back to the beginning. To position text across this seam in the path definition, create a version of the path with the entire path string duplicated. Then reduce any percentage values for `startOffset` by half, to account for the fact that the path is now twice as long.[1]

1 Thanks to Israel Eisenberg for the doubled-path solution for text on a closed shape.

The SVG 1.1 specifications did not define how `startOffset`—or text paths in general—would work for right-to-left text direction. Firefox is currently the only browser that provides legible results with correct bidirectional ordering.

> Internet Explorer and Blink/WebKit browsers do not correctly process right-to-left and bidirectional text on a path. For cursive scripts (such as Arabic) Blink/WebKit also use the isolated glyph versions of each character.

Even for Firefox (version 40) the `startOffset` value is used as an *end*-offset position when `text-anchor` is `start` and the reverse for a `text-anchor` of `end`. This ensures that text is still visible with the default offset value of 0, but it is otherwise unintuitive and inconsistent with the rest of SVG text layout.

Figure 3-4 shows how Arabic text on a path could look (screenshot from Firefox 40). The code is given in Example 3-3.

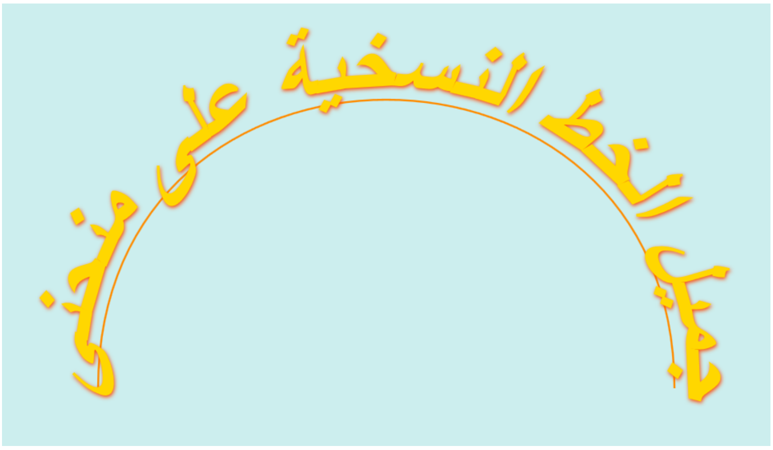


Figure 3-4. Arabic text on a path, with drop-shadow filter

Example 3-3. Displaying right-to-left cursive text within textPath

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="4in" height="2.3in" viewBox="0 0 400 230">
    <title xml:lang="en">Arabic Text on a Curved Path</title>
    <style type="text/css">
        text {
            font: bold italic 48px "Times New Roman", Times, serif;
            fill: gold;
            stroke: orangeRed;
        }
        @supports (filter: drop-shadow(0 0 0)){
            text {
                stroke: none;
                filter: drop-shadow(orangeRed 0.5px 1px 1px);
            }
        }
    </style>
    <rect fill="#CEE" width="100%" height="100%" />
    <path id="path" d="M50,200 C50,0 350,0 350,200"
          fill="none" stroke="darkOrange" />
    <text text-anchor="middle" dir="rtl" xml:lang="ar">
        <textPath xlink:href="#path" startOffset="50%"
            >جميل الخط النسخية على منحنى</textPath>
    </text>
</svg>
```

Although mostly similar to the centered version of Example 3-2, Example 3-3 uses a drop-shadow filter instead of a stroke to avoid interrupting the cursive connections. When letters are stroked, Firefox currently does not apply ligatures; even if it did, the strokes would include the edges between each glyph. A text shadow is not an improvement; it is also painted one glyph at a time, and so also overlaps the cursive connections. In contrast, the drop-shadow filter (introduced in the CSS Filters module, although it could also be created with SVG filter elements) is applied to the final shaped text, after combining all the glyphs into a single layer.

By comparison, Figure 3-5 shows the result with stroked text (also in Firefox 40). Although the connections between letters are much more awkward, this is still much closer to proper Arabic typography than any of the other web browsers are able to render at the time of writing.

The SVG 2 specifications will include new rules for how text on a path *should* behave. At the time of writing, they have not been finalized. One option would be to replace the default `startOffset` with an `auto` value that adapts according to the text direction.

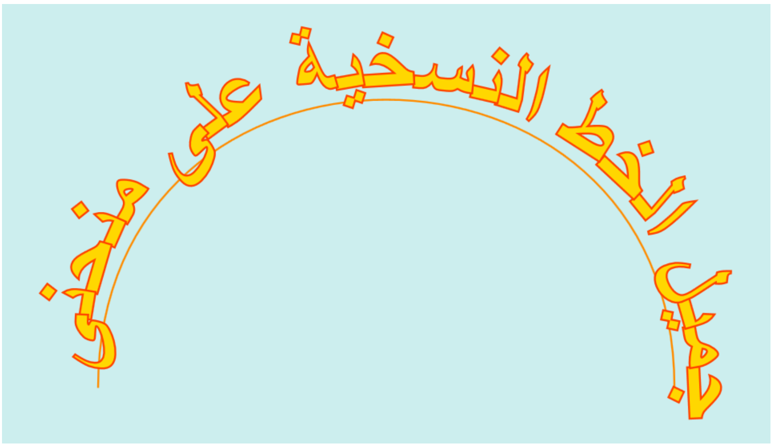


Figure 3-5. Arabic text on a path, with stroked letters

Integrating Other Text Effects

Most of the other text features we have discussed so far can be used with text on a path, some with more success than others.

The text within a `<textPath>` element can have `<tspan>` sections that change the styling. Example 3-4 adds stroke and fill changes for the keywords in the string, as displayed in Figure 3-6.

Example 3-4. Styling text spans within textPath

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xml:lang="en"
     width="4in" height="2.3in" viewBox="0 0 400 230">
    <title>Styled Text on a Curved Path</title>
    <style type="text/css">
        text {
            font: bold italic 48px "Times New Roman", Times, serif;
            fill: orangeRed;
        }
        .bright {
            fill: gold;
            stroke: orangeRed;
        }
    </style>
    <rect fill="#CEE" width="100%" height="100%" />
    <path id="path" d="M50,200 C50,0 350,0 350,200"
```

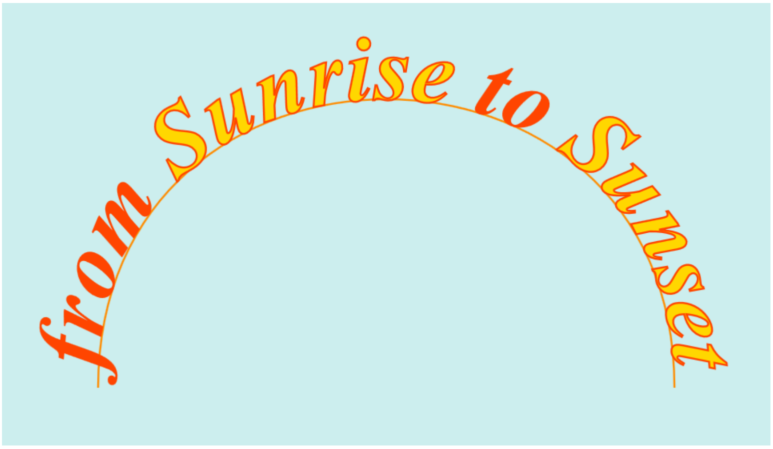


Figure 3-6. Curved text on a path, with styled spans

```
          fill="none" stroke="darkOrange" />
    <text text-anchor="middle">
        <textPath xlink:href="#path" startOffset="50%"
            >from
            <tspan class="bright">Sunrise</tspan>
            to
            <tspan class="bright">Sunset</tspan
                ></textPath>
    </text>
</svg>
```

The styles on the `<tspan>` elements are applied with the `bright` class, to override the styles set on the `<text>` as a whole.

> **!** Avoid setting styles using a CSS selector for the `textPath` tag name. Older Blink and WebKit browsers do not correctly match mixed-case tag names for SVG elements (case sensitive) that are inline in HTML 5 documents (case insensitive). The latest versions of both platforms have workarounds for this problem.

If you can use `<tspan>` within `<textPath>`, can you use the normal text path positioning attributes? You can, but they don't have the normal effect. Instead of moving text horizontally or vertically, they move them along the path or perpendicular to the path.

TIP

Although the `x`, `y`, `dx`, `dy`, and `rotate` positioning attributes on `<text>` and `<tspan>` *affect* the characters within a `<textPath>`, SVG 1.1 does not allow you to use these attributes directly on the `<textPath>` element.

For horizontal writing mode, therefore, a positive `dx` attribute moves characters futher along the path. A positive `dy` value shifts the text toward the inside of the path, while a negative `dy` value shifts it outward.

!

Internet Explorer does not render any of the text if you use `dx` or `dy` in combination with `text-anchor: middle`; it renders these offsets correctly for `text-anchor: start`.

The other browsers have no problem with relative position attributes, but every browser tested was inconsistent and buggy when absolute positioning (`x` and `y` attributes) was used.

For vertical `writing-mode`, *y*-offsets move the text along the path and *x*-offsets move it perpendicularly. For right-to-left character sequences—whether or not they are embedded in a left-to-right layout—you would need to use negative `dx` values to add space between characters, the same as for normal SVG text.

!

Browsers are currently very inconsistent about how `<textPath>` contents are laid out when the layout direction (i.e., `direction` property) is right to left. Unfortunately, the SVG 1.1 specifications did not discuss this situation carefully.

Example 3-5 uses `dx` and `dy` on both the `<text>` element that contains the `<textPath>` and the `<tspan>` elements within it. The resulting layout is shown in Figure 3-7.

Example 3-5. Using relative positioning attributes within textPath

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="4in" height="2.3in" viewBox="0 0 400 230">
    <title>Text Offset from a Curved Path</title>
    <style type="text/css">
        /* omitted to save space */                    ❶
    </style>
    <rect fill="#CEE" width="100%" height="100%" />
    <path id="path" d="M50,200 C50,0 350,0 350,200"
          fill="none" stroke="darkOrange" />
    <text dy="0.5ex" text-anchor="middle">              ❷
        <textPath xlink:href="#path" startOffset="50%"
            >from
            <tspan class="bright" dy="-1ex" dx="10px"
            >Sunrise</tspan>                            ❸
            <tspan dy="+1ex">to</tspan>                 ❹
            <tspan class="bright" dy="+1ex" dx="10px"
            >Sunset</tspan></textPath>                  ❺
    </text>
</svg>
```

❶ The styles would be the same as for Example 3-4.

❷ A `dy` attribute on the `<text>` element applies to the first character on the path. It shifts the first chunk of text on the path down by 0.5ex, so the lowercase letters half-overlap the path. Using a `middle` baseline would have had much the same effect, if it could be relied upon for consistent browser support.

❸ The “Sunrise” span starts with some extra spacing (`dx`), and is shifted outward (`dy`) by the full ex-height, so it ends up 0.5ex beyond the path.

❹ A span around the word “to” is used to cancel out the `dy` value and reset the baseline; if `baseline-shift` had better browser support, it could have been used on “Sunrise” instead, and this extra span would not be required.

❺ The “Sunset” span also starts with a `dx` offset, but its `dy` value shifts it down, into the interior of the path.

In Figure 3-7, you’ll notice that the letters in “Sunrise” are spaced farther apart than usual, while the letters in “Sunset” look rather cramped. This is because each letter is shifted perpendicular to its particular point on the path. On a curved path, those different perpendicular lines spread out on one side and come together on the other.

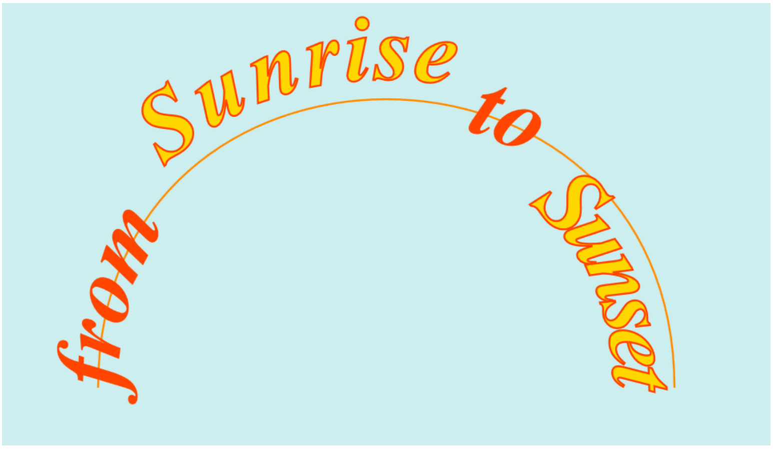


Figure 3-7. Curved text on a path, with spans offset both along and perpendicular to the path

TIP

For tightly curved paths, the cramped or stretched effect can be visible even without a perpendicular shift. Convex curves, like this, will space out letters above the baseline and compress them below the baseline. Concave curves will do the opposite. As you can guess, the choice of baseline will also affect whether or not the letters end up uncomfortably spaced.

The SVG specifications include two other attributes to control text path layout, neither of which currently have an effect in browsers:

`spacing`
: How the glyphs should be positioned along the path. The default value supported in browsers is `exact`: each glyph takes up the same space on the path as it would in a straight line of text. The alternative value, `auto`, would allow the SVG rendering agent to adjust the spacing "in order to achieve visually appealing results," although what that means is not defined.

`method`
: How the text string should be bent to fit along the path. The default value supported in browsers is `align` (each glyph is aligned with the path without distorting it); the alternative,

unsupported value is `stretch` (the tops and bottoms of each glyph are stretched or condensed to fill the available space).

The lack of support for the `stretch` method is particularly problematic with cursive scripts and fonts, whose glyphs may no longer overlap each other correctly when each character has a different rotation. In the Arabic text from Example 3-3, this is visible as cracks between adjacent glyphs, as shown in Figure 3-8 (a zoomed-in view of Figure 3-4). Nonetheless, the lack of support is perhaps not surprising, considering that there is no support anywhere else in SVG for a stretch-type distortion effect (technically called a non-affine transformation).

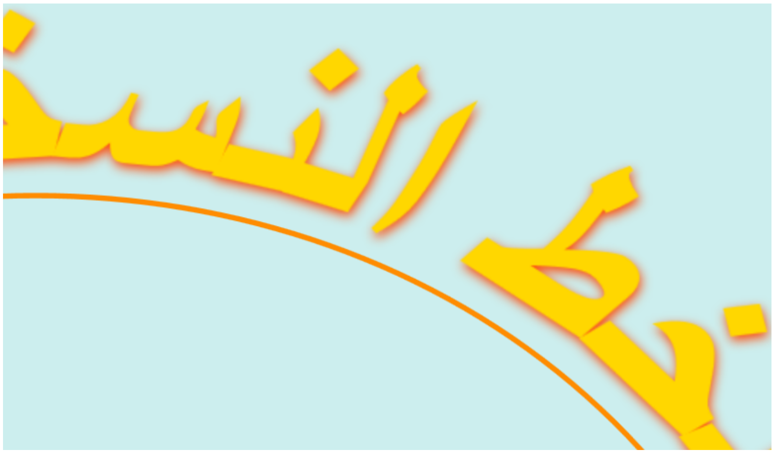


Figure 3-8. Discontinuities visible in cursive text on a path without stretch support

As mentioned briefly earlier, you can also use the `x` and `y` attributes to set absolute positioning within text on a path. These are only supposed to have an effect in the direction of the text, creating a new start offset; absolute positions perpendicular to the path would be ignored. In other words, for horizontal text on a path, the `x` attribute could be used to reposition the offset along the path. In combination with a `dy` attribute, this could (theoretically) be used to create multi-line text above and below a path.

The specifications were short on details of how this would work, and browser implementations are correspondingly inconsistent. If you want to create multiline text on a path, use two `<textPath>` elements referencing the same `<path>` shape, with different `dy` offsets for each. Example 3-6 uses this approach to create the multiline text shown in Figure 3-9.

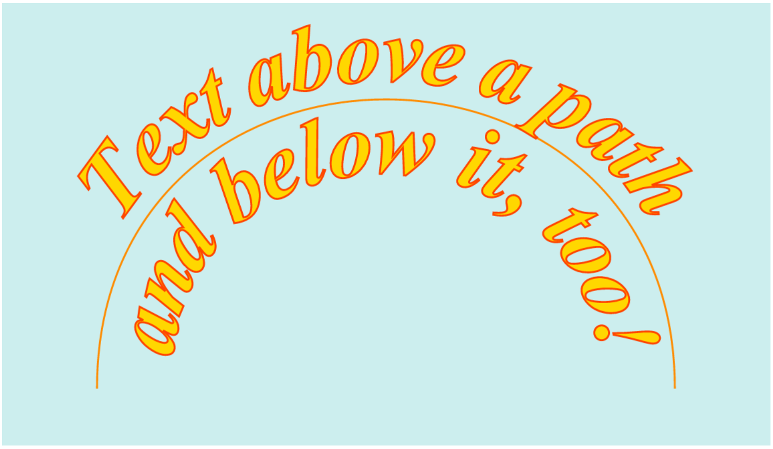


Figure 3-9. Multiline text arranged around a single path

Example 3-6. Using multiple textPath elements to create multiline text on a path

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xml:lang="en"
     width="4in" height="2.3in" viewBox="0 0 400 230">
    <title>Multiline Text on a Curved Path</title>
    <style type="text/css">
        text {
            font: bold italic 48px "Times New Roman", Times, serif;
            fill: gold;
            stroke: orangeRed;
        }
    </style>
    <rect fill="#CEE" width="100%" height="100%" />
    <path id="path" d="M50,200 C50,0 350,0 350,200"
          fill="none" stroke="darkOrange" />
    <text text-anchor="middle">
        <textPath xlink:href="#path" startOffset="50%"
            letter-spacing="-2.5px"><tspan dy="-0.2em"
            >Text above a path</tspan></textPath>
        <textPath xlink:href="#path" startOffset="50%"
            letter-spacing="5px"><tspan dy="0.8em"
            >and below it, too!</tspan></textPath>
    </text>
</svg>
```

The `<textPath>` elements in Example 3-6 use `letter-spacing` to adjust for the expansion and compression caused by the vertical offsets from the curved path. As mentioned previously, `letter-spacing` is not currently supported for SVG text in Firefox; the screenshot is from Chrome version 44.

Future Focus

Changes to Text on a Path

It's likely that SVG 2 will include a number of improvements and clarifications related to `<textPath>`, as well as a few new features.

Some changes that have already been decided:

- `<textPath>` will include a `d` attribute. It would allow you to specify the path directly, instead of having to define a separate `<path>` element.
- Alternatively, a `<textPath>` element could reference any shape element (circle, rectangle, polygon, etc.) instead of a `<path>`. Each shape has a canonical path representation that defines where a 0% start offset would be positioned.
- For closed shapes, text would continue smoothly from the end to the beginning.
- A new `side` attribute will allow you to define which side of the path the text should appear on, effectively reversing the path definition.

It will probably also be possible to specify positioning attributes directly on the `<textPath>` element, eliminating the need to have extra `<tspan>` elements.

The new specifications should also provide clearer definitions for details of text on a path layout that are currently inconsistently implemented (or not implemented at all), particularly with respect to right-to-left text layout.

CHAPTER 4

Serving Paint

From SVG Colors, Patterns, & Gradients

When the fill or stroke is more complicated than a single color (transparent or otherwise), SVG uses a concept called a *paint server* to describe how the graphic is rendered.

The paint servers are defined using their own SVG elements. Those elements—gradients and patterns—do not directly create any visible graphics. They are only used through the `fill` and `stroke` properties of shapes and text. However, by using XML markup to define the paint server, it can be infinitely variable: any SVG graphics can be used to generate an SVG pattern, including other patterns!

In contrast, when using CSS to style HTML content, all the information about how to paint an element must be contained within the CSS style rules. In CSS 2.1, the only way to create patterns was to reference external image files. Since then, CSS has introduced many graphical effects that were previously only possible with SVG, such as gradients and improved image positioning. Although the end result may look similar, the all-CSS syntax for these properties is quite different from their SVG equivalent. Throughout the rest of the book, the two approaches will be compared in "CSS Versus SVG" sidebars.

This chapter introduces the basic paint server model, and then demonstrates how it can be used in the simplest case, to serve up a single color of paint.

Paint and Wallpaper

A key feature of all SVG paint servers is that they generate a rectangular region of graphical content. This can be limiting at times, but it allows for underlying performance optimizations.

An SVG paint server doesn't need to know anything about the shape it is told to fill—it just slops on paint indiscriminately all over the wall. The shape itself acts as a mask or stencil that blocks off which parts of the paint actually reach the drawing, in the same way that a wall painter covers (*masks*) floorboards, ceilings, light fixtures, etc., so that only the wall gets painted.

Another way of thinking about paint servers—particularly when talking about gradients and patterns—is to picture the paint content as a large sheet of wallpaper. The shape is cut out from that sheet, as imagined in Figure 4-1.

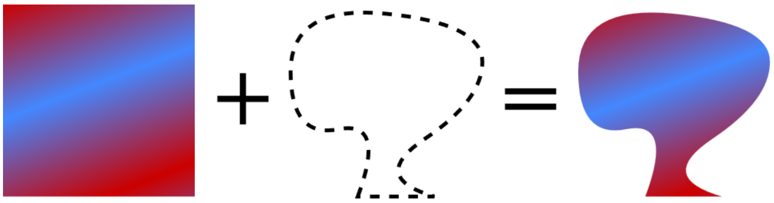


Figure 4-1. A filled shape can be thought of as a shape cut out of a rectangular sheet of patterned paper

The computer doesn't use paper and scissors, of course; instead, as it rasterizes (scans across) the shape, for every point that is "inside" the filled region, the computer looks up the corresponding (*x*,*y*) point from the paint server. A paint server can therefore be any object that can assign a specific color value to each (*x*,*y*) value.

In theory, the "paint" could be anything: a single color, one or more gradients, repeating patterns, bitmap graphics, text, even other SVG files. In practice, SVG 1.1 has two types of paint servers, gradients and repeating patterns. However, those core elements can be used to create all of the options just mentioned, as the rest of the book will demonstrate.

Identifying Your Assets

The name "server" suggests an external source for multiple resources. Theoretically, you can create a separate asset file containing all your paint servers and reference it from the `fill` or `stroke` property, but this currently has poor browser support. More generally, the name *paint server* refers to the fact that each gradient or pattern object can serve paint (rendering instructions) to multiple SVG shapes.

At the time of writing, external paint servers are only supported in Firefox and in pre-Blink versions of Opera that use the Presto rendering engine.

In order to use a paint server, you reference the paint server element using URL syntax, wrapped in the CSS `url()` functional notation. Because of the browser support limitation, this URL is nearly always an internal reference like `url(#myReference)`. The hash mark (`#`) indicates that what follows is a target toward a specific element; the fact that there is nothing *before* the hash mark indicates that the browser should look for that element in the current document. Specifically, it should look for an element with an `id` attribute that matches the target fragment (i.e., `<pattern id="myReference">`).

Thus, referencing a paint server with an ID of `"customBlue"` as a fill could look something like:

```
<rect fill="url(#customBlue)" width="100" height="100"/>
```

Because `fill` is a presentation attribute, you could also use a `<style>` block elsewhere in the document to set the value:

```
rect {
    fill: url(#customBlue);
}
```

The preceding rule would set all rectangles in the document to use that paint server, provided that the style wasn't overridden by more specific CSS rules.

Relative URLs in *external* stylesheets are always relative to the CSS file location, not the location of the document using those styles. This includes local URL fragments like `#customBlue`, which will never match anything if specified in an external CSS file. In combination with the lack of support for external paint servers, this unfortunately means that you cannot effectively use external stylesheets to set paint server references.

TIP

Relative URLs are also affected by the `xml:base` attribute or the HTML `<base>` element; using either can cause your paint server references to fail.

In theory (or if you only need to support Firefox), if you had a set of colors that are predefined in a file called *brand.svg*, you could provide the relative path to that resource, then use the target fragment to point to the specific element:

```
<rect fill="url(brand.svg#customBlue)"
      width="100" height="100"/>
```

Or you could even provide the absolute URI to that same resource—assuming the external file could be securely accessed from your web domain:

```
<rect fill="url(//example.com/assets/brand.svg#customBlue)"
      width="100" height="100"/>
```

The lack of support for this option is unfortunate, because the server concept can be thought of as being just another form of asset library, a way of storing commonly used colors, gradients, patterns, masks, and other resources in a single file. For now, if you have paint servers that are used by multiple SVGs, you need to incorporate them directly in each document, either by using some preprocessing on your server or by using AJAX techniques to import them with client-side JavaScript.

Because numerous things might interfere with the ability to load an external resource—even separate from browser support—the SVG `fill` and `stroke` properties allow you to specify a fallback color value. The color is given after the `url()` reference, separated by whitespace, like the following:

```
rect {
    fill: url(brandColors.svg#customBlue) mediumBlue;
}
```

Or, using presentation attributes and hex color syntax:

```
<rect fill="url(brandColors.svg#customBlue) #0000CD"
      width="100" height="100"/>
```

If the referenced paint server cannot be loaded, the rectangles will be painted with the specified solid blue color.

Future Focus
Layered Fill Paint and Fallbacks

SVG 2 introduces layered fills or layered strokes, similar to how CSS box layout supports layered background images.

As with multiple background images, the multiple paint options will be specified using a comma-separated list of layers from top to bottom. A fallback color will still be allowed, at the end of the list, separated by whitespace.

Unlike with the CSS `background` shorthand—which sets both the list of `background-image` values and the single `background-color` value—that final color would not normally be drawn underneath the other layers.

A sample declaration would look something like the following:

```
.typeA {
    fill: url(#pattern1), url(#gradient) mediumBlue;
}
.typeB {
    fill: url(#pattern2), url(#gradient) darkSeaGreen;
}
```

If the paint servers are loaded correctly, the `typeA` and `typeB` graphics would be distinguished by different patterns layered overtop of the same gradient. If the paint servers could not be found (perhaps your AJAX script did not run successfully), then the two classes would be drawn with different solid colors.

If you *did* want a solid color to be drawn underneath a pattern or gradient, you would separate the color into its own layer using a comma:

```
.typeA {
    fill: url(#pattern), mediumBlue;
}
.typeB {
    fill: url(#pattern), darkSeaGreen;
}
```

In this case, both classes of graphics use the same pattern (which maybe adds a textured effect), but layered over different solid colors.

The Solid Gradient

Oftentimes, especially when working with commercial uses of color, a designer will give that color a specific name. The same color may show up in many graphics related to the brand: different versions of the company logo, heading text, product labels, and so on. Rather than having to keep a list of RGB values for each color, it is much easier to define them once, give them a name, and then use that name in the content. This also makes it much easier if you decide to change one of the colors later on in the design process!

An SVG paint server is ideally suited for this task. It can be referenced by ID in the `fill` or `stroke` properties of multiple graphics, but the actual color value is only specified once and can be easily updated (or animated, as we'll show in Chapter 14 in *SVG Colors, Patterns, and Gradients*).

The original SVG specifications did not explicitly include a solid color paint server, but all browsers allow you to use a gradient with a single, un-changing color to this effect. Example 4-1 demonstrates this strategy; it uses `<linearGradient>` elements to define four named colors that are used in the branding strategy for the fictional company ACME. The colors are then used to draw a company logo, which is shown in Figure 4-2.

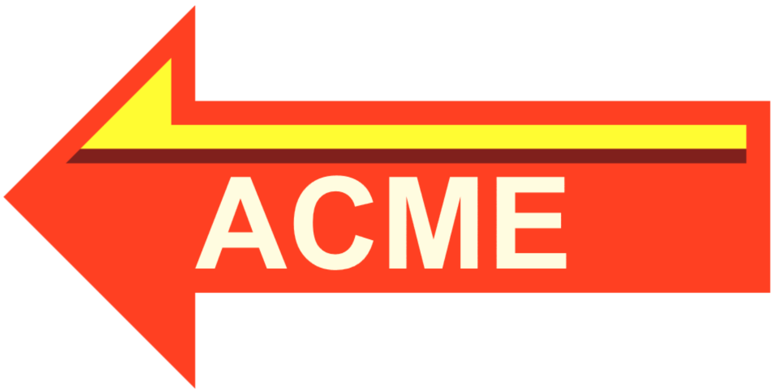


Figure 4-2. ACME Logo using named colors

Example 4-1. Defining named colors for consistent branding using single-color gradients

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xml:lang="en"
     width="100mm" height="50mm">                          ❶
    <title>ACME Logo</title>
    <defs>
        <linearGradient id="AcmeRed">                      ❷
            <stop stop-color="#FF4022" />
        </linearGradient>
        <linearGradient id="AcmeMaroon">
            <stop stop-color="#80201C" />
        </linearGradient>
        <linearGradient id="AcmeGold">
            <stop stop-color="#FFFC32" />
        </linearGradient>
        <linearGradient id="AcmeWhiteGold">
            <stop stop-color="#FFFCE0" />
        </linearGradient>
        <symbol id="AcmeLogo" viewBox="0,-40 160,80" >     ❸
            <path d="M0,0 L40,-40 L40,-20 L160,-20
                     L160,20 L40,20 L40,40z"
                  fill="url(#AcmeRed)"/>                   ❹
            <path d="M16,-10 L35,-29 L35,-15 L155,-15 L155,-10 z"
                  fill="url(#AcmeGold)"/>
            <path d="M13,-7 L16,-10 L155,-10 L155,-7 z"
                  fill="url(#AcmeMaroon)"/>
            <text x="40" y="15"
                  style="font-family:Arial; font-weight:bold;
                         font-size:20pt;"
                  fill="url(#AcmeWhiteGold)">ACME</text>
        </symbol>
    </defs>
    <use xlink:href="#AcmeLogo" />                         ❺
</svg>
```

❶ The SVG does not have a `viewBox`; scaling is controlled by the `<symbol>` element that contains the logo. However, default `width` and `height` values ensure that the image has the correct intrinsic aspect ratio and a reasonable default size when embedded in other web pages.

❷ The company has four brand colors, AcmeRed, AcmeMaroon, AcmeGold, and AcmeWhiteGold. Each color is defined as a paint-server using a `<linearGradient>` with a single `<stop>` element.

❸ The logo itself is defined inside a `<symbol>` element for easy reuse in other graphics. The `viewBox` creates a coordinate system that is centered on the vertical axis.

❹ Each shape within the symbol uses one of the predefined paint servers to set the `fill` color.

❺ The logo is drawn within the SVG with a `<use>` element. The `<use>` element does not have any positioning or sizing attributes, so the reused `<symbol>` will scale to fill the entire SVG area.

Examining the gradients more closely, each consists of two elements, `<linearGradient>` and `<stop>`:

```
<linearGradient id="AcmeRed">
    <stop stop-color="#FF4022" />
</linearGradient>
```

The `<linearGradient>` defines the paint server, and gives it the `id` value that will be used to reference it. This gradient element is also a container for the `<stop>` element that defines the color. For a normal gradient, there would be multiple stops defining the initial, final, and intermediary colors.

The color is specified using the `stop-color` presentation attribute. There is also a `stop-opacity` presentation attribute, similar to `fill-opacity` or `stroke-opacity`; by default, colors are fully opaque.

Although Example 4-1 works as intended in every web browser tested, it fails in Apache Batik, which is more strict on syntax. To make it work, the `<stop>` elements also require an `offset` attribute, which we'll discuss in Chapter 6 of *SVG Colors, Patterns, and Gradients*.

Because the colors are defined in a single location, they can be changed easily and consistently, or animated uniformly. Because `stop-color` is a presentation attribute, you don't even need to edit the XML to change the color; you can override it with CSS rules.

As a result, you can use conditional CSS rules to change the color. A stylesheet with media queries can be used to assign print colors for high-quality printers, or for grayscale printing. Because the color is used by reference in the rest of the graphic, the stylesheet does not need to identify all the elements that use each color, nor does it need to distinguish between `fill` and `stroke` values.

TIP

Although `stop-color` is a presentation attribute, it is not inherited by default. It must be explicitly set on the `<stop>` element, either directly or by using the `inherit` keyword.

Example 4-2 gives a sample set of print styles. For color printing, it redefines the colors using HSL notation, which can then be mapped to the full color gamut used on the print device. For monochrome printing, it assigns each color to a shade of gray that will create stronger contrast than if the colors were converted to gray automatically. The grayscale version is shown in Figure 4-3.

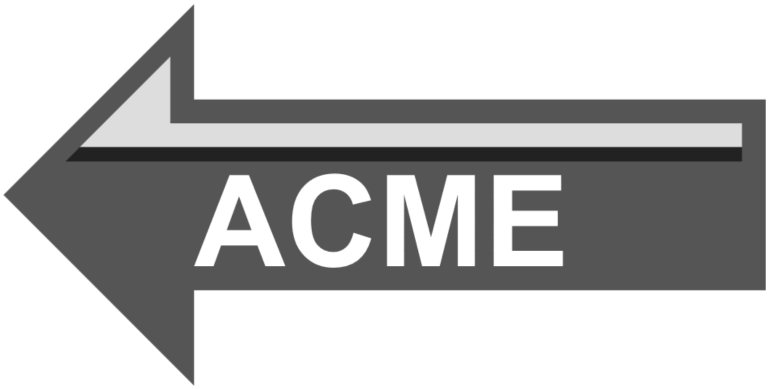


Figure 4-3. ACME Logo using named colors, converted to monochrome

Example 4-2. Redefining named colors for print graphics

```
@media print AND (color) {
    #AcmeRed stop       { stop-color: hsl(10, 100%, 60%); }
    #AcmeMaroon stop    { stop-color: hsl( 0,  65%, 30%); }
    #AcmeGold stop      { stop-color: hsl(60, 100%, 60%); }
    #AcmeWhiteGold stop { stop-color: hsl(55, 100%, 90%); }
}
@media print AND (monochrome) {
    #AcmeRed stop       { stop-color: #555; }
    #AcmeMaroon stop    { stop-color: #222; }
    #AcmeGold stop      { stop-color: #DDD; }
    #AcmeWhiteGold stop { stop-color: #FFF; }
}
```

Although most browsers correctly apply CSS print styles when printing a web page, they do not always apply monochrome styles when the user chooses to print in black and white on a color printer.

Using paint servers to name nonstandard colors in this way has the additional advantage that it makes your code easier for others to read. By using meaningful `id` values, the color and purpose of each element becomes apparent to any programmer who has to adapt your work in the future.

Future Focus
The <solidcolor> Paint Server

Named color paint servers have many benefits. However, using a single-color gradient to create a named color is a bit of a hack; it certainly was not the original purpose of these elements.

SVG 2 therefore uses the `<solidcolor>` element to create a single-color paint server with no hackery. It uses the `solid-color` and `solid-opacity` presentation attributes to set the color value.

Using `<solidcolor>` elements, the four brand colors from Example 4-1 could be defined as follows:

```
<solidcolor id="AcmeRed"       solid-color="#FF4022" />
<solidcolor id="AcmeMaroon"    solid-color="#80201C" />
<solidcolor id="AcmeGold"      solid-color="#FFFC32" />
<solidcolor id="AcmeWhiteGold" solid-color="#FFFCE0" />
```

This not only reduces the amount of markup, it also makes the purpose of your code more readily apparent.

The `<solidColor>` element (note the capital C!) was included in the SVG Tiny 1.2 specification, so it is supported in some graphics programs; you would need to explicitly set the `version="1.2"` attribute on the root `<svg>` element. (In contrast, web browsers ignore `version` and use the latest spec for all SVG content.) The latest draft SVG 2 specification changes the capitalization to make the element more HTML-friendly, which unfortunately breaks compatibility in case-sensitive XML viewers.

At the time of writing, neither `<solidColor>` nor `<solidcolor>` are supported in the stable version of any major web browser. However, an implementation is under development in Firefox.

About the Authors

Amelia Bellamy-Royds is a freelance writer specializing in scientific and technical communication. She helps promote web standards and design through participation in online communities such as Web Platform Docs, Stack Exchange and Codepen. Her interest in SVG stems from work in data visualization, and builds upon the programming fundamentals she learned while earning a B.Sc. in bioinformatics. A policy research job for the Canadian Library of Parliament convinced her that she was more interested in discussing the big-picture applications of scientific research than doing the laboratory work herself, leading to graduate studies in journalism. She currently lives in Edmonton, Alberta. If she isn't at a computer, she's probably digging in her vegetable garden or out enjoying live music.

Kurt Cagle worked as a member of the SVG Working Group, and wrote one of the first SVG books on the market in 2004. After consulting to a number of Fortune 500 media, transportation and publishing companies as well as having worked as an architect with the US National Archives and the Affordable Care Act, Kurt founded Semantical, LLC in 2015 to develop applications for data visualization, virtualization and enrichment.