O'REILLY®

Using SVG with CSS3 & HTML5

DRAWING WITH MARKUP

Amelia Bellamy-Royds & Kurt Cagle

Using SVG with CSS3 and HTML5

Vector Graphics for Web Design

Amelia Bellamy-Royds,
Kurt Cagle and Dudley Storey

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

Using SVG with CSS3 and HTML5

by Amelia Bellamy-Royds , Kurt Cagle , and Dudley Storey

Printed in the United States of America.

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

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

Editors: Simon St. Laurent and Meghan Blanchette
Production Editor: FILL IN PRODUCTION EDITOR
Copyeditor: FILL IN COPYEDITOR

Proofreader: FILL IN PROOFREADER
Indexer: FILL IN INDEXER
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

December 2016: First Edition

Revision History for the First Edition

2016-11-18: First Early Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491979013* for release details.

978-1-491-97901-3

[FILL IN]

Table of Contents

Preface

(General Intro)

A Winding Path

(Making of this book)

The Road Ahead

SVG is a complex topic, but we have tried to arrange this book in a logical progression. Part I begins with the wide view, discussing how —and why—you use SVG in web design:

- Chapter 1 sketches out the possibilities of SVG as an independent image format.
- Chapter 2 looks at SVG on the web, focusing on how it interacts with other coding languages.
- Chapter 3 describes how CSS can be used to style your SVG, and how SVG graphics can be used with CSS to style other documents.
- Chapter 4 introduces useful software for creating and testing SVG images, as well as some sources of ready-to-use SVG for less artistically-inclined web developers.

The remainder of the book will narrow in on each of the main features of SVG one chapter at a time. ??? concentrates on the core drawing elements in SVG, and how to control their geometry and layout:

- Sizing and positioning basic shapes, in ???

- Defining custom shapes and lines, in ???
- Text layout, in ???

??? dives into the technical details of how SVG documents are constructed and how vector shapes are positioned:

- Establishing coordinate systems and scale, in ???
- Re-defining coordinate systems when embedding graphics in web pages, in ???
- Re-using content and embedding images, in ???
- Transforming coordinate systems to reposition and distort graphics, in ???

??? focuses more on the graphical side of the language:

- Filling the area of shapes and text, including gradients and patterns, in ???
- Drawing outlines around shapes and text, in ???
- Adding line markers (repeated symbols on the ends or corners of custom shapes), in ???
- Controlling opacity and the blending of one graphic into its background, in ???
- Clipping and masking of graphics, in ???
- Filter effects, in ???

??? looks at how the basic structure of SVG images can be enhanced to create complete web applications, focusing on two main areas:

- Accessibility and metadata, in ???
- Animation and interaction, in ???

Once you have all the pieces in place, ??? returns to the big picture, discussing best practices for working with SVG.

Before You Begin

This book focuses on *Using SVG* in web pages. It assumes that you, the reader, are already familiar with creating web pages, using HTML, CSS, and a little bit of JavaScript. When the examples use

relatively new features of CSS 3 and HTML 5, we'll explain them briefly, but we'll assume you know a `<div>` from a `<li>`, and a `font-family` from a `font-style`.

You'll get the most out of the book by working through the code samples as you go. It will help if you have a good code editor that recognizes SVG syntax, and if you know how to use the developer tools in your web browser to inspect the document structure and styles that create the visible result.

About This Book

Whether you're casually flipping through the book, or reading it meticulously cover-to-cover, you can get more from it by understanding the following little extras used to provide additional information.

Conventions Used in This Book

(O'Reilly boilerplate on code & term formatting)

Tips like this will be used to highlight particularly tricky aspects of SVG, or simple shortcuts that might not be obvious at first glance.

Notes like this will be used for more general asides and interesting background information.

Warnings like this will highlight combatibility problems between different web browsers (or other software), or between SVG as an XML file versus SVG in HTML pages.

About the Examples

(Where to download sample files or view online, compatibility info)

(O'Reilly boilerplate on copyright & permissions)

How to Contact Us

(O'Reilly boilerplate)

Acknowledgements

(Thank you, thank you very much)

PART I

SVG on the Web

Scalable Vector Graphics (SVG) are drawings and diagrams defined using an open standard of human-readable XML code. SVG can be used in print publishing and even in technical drawings. However, SVG's true potential resides in the web browser. SVG was designed to work with the other languages to describe, style, and manipulate content on the web: HTML, CSS, and JavaScript.

The following chapters look at SVG as a whole, focusing on how it is created and used on the web, and how it intersects and overlaps other web standards.

CHAPTER 1

Graphics from Vectors

An Overview of SVG

There's a fundamental chicken-and-egg quality to creating SVG that can make teaching it a challenge. Shapes without styles are not terribly attractive; styles without shapes cannot be seen. To work with an SVG, you need to display the graphic on the web; to display a graphic, you need some SVG code to display!

This chapter presents a rough sketch of the chicken *and* the egg, so that subsequent chapters can fill in the details one topic at a time, without feeling like large parts of the picture are missing.

The chapter starts with a simple SVG graphic and then adapts it, to use different techniques and to add new functionality. The examples will introduce many key features of SVG, but will skip over many others. At the end, you should have a good idea of what an SVG file looks like, how the key elements relate to each other, and how you can edit the file to make simple changes.

The graphics in this chapter, and the rest of the book, involve building SVG directly as XML code in a text editor, rather than using a tool such as Inkscape or Adobe Illustrator. There are a couple of reasons for this:

- It helps you focus on building applications with SVG, rather than just drawing graphics—you can always extend these principles to more artistic images. To keep from having pages and

pages of SVG markup, the graphics used here are…minimalistic.

- When using graphics editors, it is easy to generate overly complex code that would distract from the key messages of the examples. If you use a code editor to view a file created by these programs, you'll discover many extra attributes and elements identified by custom XML namespaces. These are used internally by the software but don't have an effect when the SVG is displayed in a web browser.

Hand-coding SVG from scratch is only practical for simple geometric drawings. Working with the code, however, is essential for creating interactive and animated graphics. For more complex graphics, a drawing made in a visual editor can be exported to SVG, and then it can be adapted as code.

Alternatively, some graphics editors, such as Adobe Illustrator, allow you to copy individual shapes (or groups of shapes) from the editor and paste them into your text editor, with the pasted result being the SVG markup for that shape.

To follow along with the examples in this chapter, it will help if you have a basic familiarity with XML, or at least HTML (which is similar, but not the same). In future chapters, we will also assume that you are familiar with CSS and with using JavaScript to manipulate web pages. We'll always try to explain the purpose of all the code we use, but we won't discuss the basic syntax of those other web languages. If you're not comfortable with HTML, CSS, and JavaScript, you'll probably want to have additional reference books on hand as you experiment with SVG on the web.

Defining an SVG in Code

At its most basic, an SVG image consists of a series of shapes that are drawn to the screen. Everything else builds upon the shapes. Individual SVG shapes *can* be incredibly complex, made up of hundreds of distinct lines and curves. The outline of Australia (including the island of Tasmania) could be represented by a single `<path>` shape on an SVG map. For this introductory overview, however,

we're keeping it simple. We're using two shapes you're probably quite familiar with: circles and rectangles.

Figure 1-1 is a colored line drawing, such as you might find in a children's book, of a cartoon stoplight.

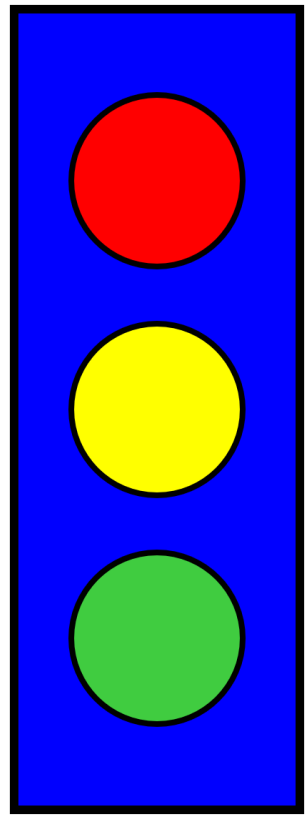


Figure 1-1. Primary color stoplight graphic

To draw the stoplight with SVG, create a new file in your code editor and save it, as unformatted text with UTF-8 encoding, with the *.svg* file extension. In that file, include the following code to define the root SVG element.

```
<svg xmlns="http://www.w3.org/2000/svg" xml:lang="en"
     height="320px" width="140px" >
     <!-- drawing goes here -->
</svg>
```

This is the root element that defines the document as an SVG file; all the graphic content will be contained in between the starting `<svg>` tag and the ending `</svg>` tag. The starting tag also contains attributes that modify the SVG element.

The first and most important of the attributes is the declaration of the SVG namespace, using `xmlns="http://www.w3.org/2000/svg"`. An SVG in its own file should be treated as XML, and most browsers will not render (draw) the SVG image without the namespace. The namespace identifier confirms that this is Scalable Vector Graphics document, as opposed to some custom XML data format that just happens to use the `svg` acronym for its root element.

Most complete SVG documents have other namespace declarations as well, indicated by an `xmlns:prefix` attribute, where the prefix will be re-used elsewhere in the document. Only one such namespace is standard in SVG (XLink, which we'll discuss in "Repetition without Redundancy" on page 20), but SVG graphics editors often add custom namespaces to hold software-specific data.

The second attribute, `xml:lang` defines the human language of any text content in the file, so that screen readers and other software can make appropriate adjustments. In this case (and every other case in this book), that language is English, as indicated by the "en" prefix. We could specify "en-US" to clarify that we're using American spelling, if we preferred.

The `xml:lang` attribute can be set on any element, applying to its child content. If you have multi-lingual diagrams, you may wish to set the attribute on individual text and metadata elements. The behavior is directly equivalent to the `lang` attribute in HTML.

You *don't* need to declare the `xml` namespace prefix: it is reserved in all XML files. To make things even simpler, SVG 2 defines a plain `lang` attribute, without XML prefixes, to replace `xml:lang` (but keep using the prefixed version for a while, until all software catches up).

The root SVG element also has `height` and `width` attributes, here defined in pixel units. This is the simplest way to define the dimensions of a graphic; we'll discuss other approaches in ???.

Although it is not required, you may want to start your SVG file with an XML declaration, indicating the XML version used or the character encoding. The following explicitly declares the default (version 1.0) XML syntax in a file with an 8-bit Unicode character set:

```
<?xml version="1.0" encoding="UTF-8" ?>
```

The XML instructions should appear on the first line of the file.

You may also include an SGML `DOCTYPE` declaration, although it is no longer recommended for SVG. The `DOCTYPE` points to the document type definition of the SVG file format, and is used by some validators and code editing tools. However, some of these validation tools will not recognize perfectly valid XML content from non-SVG namespaces.

SGML is the Standard Generalized Markup Language, the parent language for both HTML and XML. It created the idea of elements defined by angle brackets (< and >), with attributes defining their properties. SGML document type definitions (DTD files) are machine-readable files that define what elements are allowed for that document, what attributes they can have, and whether they have start and end tags, and if so what other types of elements can be included in-between. The `DOCTYPE` declaration indicates which DTD files to use.

If you do include the `DOCTYPE`, it should appear on a line in between the XML declaration and the starting `<svg>`, and should exactly

match the following (except that the amount of whitespace is flexible):

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

If you copy SVG code into another file, such as an HTML 5 document, don't include the document type or the XML declaration. They are only valid at the start of a file.

The child content of the SVG root element—the code that replaces the `<!-- drawing goes here -->` comment—can be a mix of shape elements, text elements, animation elements, structural elements, style elements, and metadata elements. For this first SVG example, we will mostly use shape elements. However, one metadata element you should always include is a title. The following code shows how it is added to the `<svg>`:

```
<svg xmlns="http://www.w3.org/2000/svg" xml:lang="en"
     height="320px" width="140px" >
     <title>Primary Color Stoplight</title>
     <!-- drawing goes here -->
</svg>
```

In graphical SVG editors, you can often set the main title using a "document properties" dialog.

When an SVG `<title>` element is included like this—as the first child of the root `<svg>`—it will be used in the same manner as an HTML `<title>`. The title text is not part of the graphic, but if you viewed the SVG in a browser "Primary Color Stoplight" would be displayed in the browser tab, and would be used for browser bookmarks.

The SVG `<title>` element is much more flexible than it's HTML equivalent; you can add titles to individual parts of a graphic as well, and they'll show up as tooltips. We'll explore titles and other metadata in depth in ???.

So far, we have a valid SVG file, with a title, but if you open that file in the browser you'll just see a plain white screen. Time to start drawing.

Simple Shapes

At its most basic, an SVG image consists of a series of shapes that are drawn to the screen. Everything else builds upon the shapes. Individual SVG shapes *can* be incredibly complex, made up of hundreds of distinct lines and curves. The outline of Australia (including the island of Tasmania) could be represented by a single `<path>` shape on an SVG map. For this introductory overview, however, we're keeping it simple. We're using two shapes you're probably quite familiar with: circles and rectangles.

There are four shapes in Figure 1-1: one rectangle and three circles. The layout, sizing, and coloring of those shapes creates the image that can be recognized as a stoplight.

Add the following code after the `<title>` to draw the blue rectangle from Figure 1-1:

```
<rect x="20" y="20" width="100" height="280"
      fill="blue" stroke="black" stroke-width="3" />
```

The `<rect>` element defines a rectangle that starts at the point given by the `x` (horizontal position) and `y` (vertical position) attributes and has an overall dimension given by the the `width` and `height` attributes. Note that you don't need to include units on the length attributes. Length units in SVG are pixels by default, although the definition of a pixel (and of every other unit) will change if the graphic is scaled.

"Pixels" when talking about lengths means CSS layout `px` units. These will not always correspond to the actual pixels (picture elements) on the monitor. The number of individual points of color per `px` unit can be affected by the type of screen (or printer) and the user's zoom setting.

In software that supports CSS 3, all other measurement units are adjusted proportional to the size of a `px` unit. An `in` (inch) unit will always equal `96px`, regardless of the monitor resolution —but it might not match the inches on your ruler!

The coordinate system used for the *x*- and *y*-positions is similar to many computer graphics and layout programs. The *x*-axis goes from

the left of the page to the right in increasing value, while the *y*-axis goes from the top of the page to the bottom. This means that the default zero-point, or origin, of the coordinate system is located at the upper left hand corner of the window. The rectangle is drawn starting 20px from the left and 20px from the top of the window.

If you're used to mathematical coordinates where the y-axis increases from bottom to top, it might help to instead think about laying out lines of text from top to bottom on a page.

The remaining attributes for the rectangle define its presentation, the styles used to draw the shape:

```
<rect x="20" y="20" width="100" height="280"
      fill="blue" stroke="black" stroke-width="3" />
```

The `fill` attribute indicates how the interior of the rectangle should be filled in. The fill value can be given as a color name or a hex color value—using the same values and syntax as CSS—to flood the rectangle with that solid color. The `stroke` and `stroke-width` attributes define the color and thickness of the lines that draw the rectangle's edges.

If you don't use XML regularly, be sure to pay attention to the `/` (forward slash) at the end of every shape tag, closing the element it creates. You could also use explicit closing tags, like `<rect` *`attributes`* `></rect>`.

With the basic rectangular shape of the stoplight now visible, it is time to draw the lights themselves. Each circular light can be drawn with a `<circle>` element. The following code draws the red light:

```
<circle cx="70" cy="80" r="30"
        fill="red" stroke="black" stroke-width="2" />
```

The first three attributes define the position and size of the shape. The `cx` (center-*x*) and `cy` (center-*y*) attributes define coordinates for the center-point of the circle, while the `r` attribute defines its radius. The `fill`, `stroke`, and `stroke-width` presentation attributes have the same meaning as for the rectangle (and for every other shape in SVG).

If you re-create the graphic in a visual editor, then look at the code later, you might not see any `<circle>` elements. A circle, and every other shape in SVG, can also be represented by the more obscure `<path>` element, which we introduce in ???.

You can probably figure out how to draw the yellow and green lights: use the code for the red light, but change the vertical position by adjusting the `cy` attribute, and set the correct fill color by changing the `fill` presentation attribute. The complete SVG markup for the stoplight is given in Example 1-1.

Example 1-1. Drawing a primary color stoplight with SVG

```
<?xml version="1.0" encoding="UTF-8" ?>
<svg xmlns="http://www.w3.org/2000/svg" xml:lang="en"
     height="320px" width="140px" >
    <title>Primary Color Stoplight</title>
    <rect x="20" y="20" width="100" height="280"
          fill="blue" stroke="black" stroke-width="3" />
    <circle cx="70" cy="80" r="30"
            fill="red" stroke="black" stroke-width="2" />
    <circle cx="70" cy="160" r="30"
            fill="yellow" stroke="black" stroke-width="2" />
    <circle cx="70" cy="240" r="30"
            fill="#40CC40" stroke="black" stroke-width="2" />
                  ❶
</svg>
```

❶ `#40CC40` is a medium green color, defined in hexadecimal RGB notation. It's brighter than the color created by the `green` keyword (`#008800`), but not quite as intense as `lime` (`#00FF00`). There's actually a `limegreen` keyword that is a pretty close match, but we wanted to emphasize that you could use hexadecimal notation to customize colors. We'll discuss more color options in ???.

The shapes are drawn on top of one another, in the order they appear in the code. Thus, the rectangle is drawn first, then each successive circle. If the rectangle had been listed after the circles, its solid blue fill would have completely obscurred them.

If you work with CSS, you know you can change the drawing order of elements using the `z-index` property. `z-index` has been added to the SVG specifications, too, but at the time of writing it is not supported in any of the major web browsers.

This code from Example 1-1 is one way to draw the stoplight in Figure 1-1, but it isn't the only way. In the next section, we explore other ways of creating the same image with SVG. The graphics will *look* the same, but they will have very different structures in the document object model (DOM).

Why is that important? If all you care about is the final image, it isn't. However, if you are going to be manipulating the graphic with JavaScript, CSS, or animations, the DOM structure is very important. Furthermore, if you modify the graphic in the future—maybe to change the sizes or styles of the shapes—you will be glad if you used clean and DRY code, where *DRY* stands for Don't Repeat Yourself.

Repetition without Redundancy

The code in Example 1-1 is somewhat redundant: many attributes are the same for all three circles. If you want to remove the black strokes or make the circles slightly larger, you need to edit the file in multiple places. For a short file like this, that might not seem like much of a problem. But if you had dozens (or hundreds) of similar shapes, instead of just three, editing each one separately would be a headache and an opportunity for error.

Some of the repetition can be removed by defining the circles inside a `<g>` element. The `<g>` or group element is one of the most commonly used elements in SVG. A group provides a logical structure to the shapes in your graphic, but it has the additional advantage that styles applied to a group will be inherited by the shapes within it. The inherited value will be used to draw the shape unless the shape element specifically sets a different value for the same property.

Groups have other uses. They can associate a single `<title>` element with a set of shapes that together make up a meaningful part of the graphic. They can be used to apply certain stylistic effects, such as masks (???) or filters (???) on the combined graphic instead of the

individual shapes. Grouping can also be used to move or even hide a collection of elements as a unit. Many vector graphic drawing programs use layers of graphics which combine to form an image; these are almost always implemented as `<g>` elements in the SVG file.

In Example 1-2, the `stroke` and `stroke-width` presentation attributes are specified once for the group containing three circles. The final graphic looks exactly the same as Figure 1-1.

Example 1-2. Grouping elements within an SVG stoplight

```
<svg xmlns="http://www.w3.org/2000/svg" xml:lang="en"
     height="320px" width="140px" >
    <title>Grouped Lights Stoplight</title>
    <rect x="20" y="20" width="100" height="280"
          fill="blue" stroke="black" stroke-width="3" />
    <g stroke="black" stroke-width="2">
        <circle cx="70" cy="80" r="30" fill="red" />
        <circle cx="70" cy="160" r="30" fill="yellow" />
        <circle cx="70" cy="240" r="30" fill="#40CC40" />
    </g>
</svg>
```

Why not specify the `cx` and `r` attributes on the group, since they are also the same for every circle? The difference is that these attributes are specific features of circles, describing their fundamental geometry. Geometric attributes are not inherited; if they aren't specified, they default to zero. And if a circle's radius is zero, it won't be drawn at all.

In contrast, fill and stroke attributes describe the styles to be used when drawing any shape. Just like CSS styles in HTML, they can be specified on a parent element and inherited by its children. In fact, presentation attributes work *exactly* like CSS styles, and you can use CSS style notation to set their values, as in Example 1-3.

Example 1-3. Using inline styles in the SVG stoplight

```
<svg xmlns="http://www.w3.org/2000/svg" xml:lang="en"
     height="320px" width="140px" >
    <title>Grouped Lights Stoplight</title>
    <rect x="20" y="20" width="100" height="280"
          style="fill:blue; stroke:black; stroke-width:3" />
    <g style="stroke:black; stroke-width:2">
        <circle cx="70" cy="80" r="30" style="fill:red" />
        <circle cx="70" cy="160" r="30" style="fill:yellow" />
```

```
        <circle cx="70" cy="240" r="30" style="fill:#40CC40" />
    </g>
</svg>
```

Again, the result of this code is exactly the same as Figure 1-1; it has just been re-written to clearly separate the geometric attributes from the styles. The interaction between CSS and presentation attributes will be discussed in more detail in Chapter 3; for now, think of presentation attributes as default styles to use if CSS styles aren't specified.

What about the geometric attributes? Many graphics contain repeated geometric shapes, and those shapes are often much more complicated than simple circles. SVG has its own approach to avoiding redundant code in those cases: the `<use>` element. Example 1-4 defines the basic circle once, and then re-uses it three times, with different vertical positions and fill colors.

Example 1-4. Re-using elements to draw an SVG stoplight

```
<svg xmlns="http://www.w3.org/2000/svg" xml:lang="en"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     height="320px" width="140px" >
    <title>Re-usable Lights Stoplight</title>
    <defs>
        <circle id="light" cx="70" r="30" />
    </defs>
    <rect x="20" y="20" width="100" height="280"
          fill="blue" stroke="black" stroke-width="3" />
    <g stroke="black" stroke-width="2">
        <use xlink:href="#light" y="80" fill="red" />
        <use xlink:href="#light" y="160" fill="yellow" />
        <use xlink:href="#light" y="240" fill="#40CC40" />
    </g>
</svg>
```

Let's break that example down to clearly explain what's going on. The first change is a new attribute on the `<svg>` itself. The `xmlns:xlink` attribute defines a second XML namespace, `"http://www.w3.org/1999/xlink"`, which will be identified by the `xlink` prefix. XLink was a W3C Standard for defining relationships between XML elements or files. The `xlink:href` attribute is fundamental to many SVG elements in SVG 1, and other XLink attributes were also adopted to describe hyperlinks. However, XLink isn't really used anywhere else on the web other than SVG, so the name-

space and attributes have been deprecated. When browsers fully support SVG 2, you'll be able to use the `href` attribute without any namespace.

Because a stand-alone SVG file is XML, you *could* use any prefix you choose to represent the XLink namespace; all that matters is the `"http://www.w3.org/1999/xlink"` namespace URL. However, when you use SVG within HTML files—which don't support XML namespaces—only the standard `xlink:href` attribute name will be recognized.

At the time of writing (late 2016), skipping the namespace altogether (and just using `href`) is supported in Microsoft Edge, Internet Explorer, and the very latest Firefox and Blink browsers. It isn't supported in WebKit / Safari.

The next new feature is the `<defs>` element, which contains definitions of SVG content for later use. Children of a `<defs>` element are never drawn directly. In Example 1-4, one element is defined in this way: the circle. The `cx` and `r` attributes which were previously repeated for each light are now included only once on this pre-defined circle. However, the circle has no `cy` attribute—it will default to zero—and no styles: it will inherit styles whenever it is used.

Most importantly of all, the circle has the `id` attribute `"light"`. Without an ID, there would be no way to indicate that this is the graphic to be re-used later. The SVG `id` attribute has the same role and restrictions as `id` in HTML or `xml:id` in other XML documents:

- It must start with a letter, and contain only letters, numbers, periods (`.`), and hyphens (`-`).
- It must be completely unique within a document.

The final change to the SVG code in Example 1-4 is that each `<circle>` in the main graphic has been replaced by a `<use>` element that refers back to the single pre-defined circle with the `xlink:href` attribute.

The content of `xlink:href` is always a URI (Universal Resource Identifier). To identify another element in the same document, you

use a target fragment: a hash mark (#) followed by the other element's id value, the same as you would use for internal hyperlinks within an HTML document.

The URI format may make you wonder if it is possible to re-use elements from separate SVG files. The SVG specifications allow it, but there are important browser security and support restrictions which we'll discuss in ???.

The `<use>` elements have other attributes. The `y` attribute tells the browser to shift the re-used graphic vertically so that the *y*-axis from the original graphic now lines up with the specified *y*-position. A similar `x` attribute could have been used for a horizontal shift. Since the original circle was defined with its vertical center (`cy`) as the default zero, the effect is to move the center of the circle to the given value of `y`. Finally, the `fill` value specified on each `<use>` element becomes the inherited fill color for that instance of the circle.

Again, although we've made considerable changes to the document structure, the final graphic still looks the same, as shown in Figure 1-2.

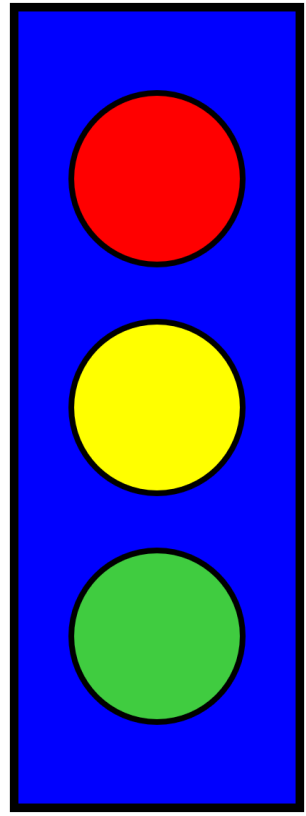


Figure 1-2. Stoplight drawn with re-used elements

Exciting, right? Or maybe not. All that fussing with document structure and we've still got the exact same picture. It is important to know that you can draw the same graphic many different ways without changing its appearance. But it is equally important to know how to dress up that graphic with some new styles.

Graduating to Gradients

If your target audience is over the age of ten, you might find the blocks of solid color in Figure 1-2 a tad simplistic. One option for

enhancing the graphic would be to draw in extra details with additional shapes. Another option is to work with the shapes we have, but `fill` them with something other than solid colors.

At first glance, `fill` would appear to be just another term for color. However, this is a little misleading. You can fill—and also stroke—shapes with gradients or patterns (which we'll discuss more in ???) instead of solid colors.

The gradients and patterns are defined as separate elements within the SVG code, but they are never drawn directly. Instead, the gradient or pattern is drawn within the area of the shape that references it. In a way, this is similar to a web domain serving up an image for a browser to draw within a specified region of an HTML file. For this reason, the gradient or pattern is known as a paint server.

The server analogy isn't just superficial. In theory, you should be able to take an external SVG document and put in multiple paint servers—gradients or patterns—then reference the file and element using a URI like `gradients.svg#metal`. However, as mentioned above, support for references between files is subject to browser security limitations.

Example 1-5 defines four different gradients for the three lights and the stoplight frame. The result is shown in Figure 1-3.

Example 1-5. Using gradient fills to enhance a vector graphic stoplight

```
<svg xmlns="http://www.w3.org/2000/svg" xml:lang="en"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     height="320px" width="140px" >
    <title>Gradient-Filled Stoplight</title>
    <defs>
        <circle id="light" cx="70" r="30" />
        <radialGradient id="red-light-off" fx="0.45" fy="0.4">
            <stop stop-color="maroon" offset="0"/>
            <stop stop-color="#220000" offset="0.7"/>
            <stop stop-color="black" offset="1.0"/>
        </radialGradient>
        <radialGradient id="yellow-light-off" fx="0.45" fy="0.4">
            <stop stop-color="#A06000" offset="0"/>
            <stop stop-color="#804000" offset="0.7"/>
            <stop stop-color="#502000" offset="1"/>
        </radialGradient>
```

```
        <radialGradient id="green-light-on" fx="0.45" fy="0.4">
            <stop stop-color="#88FF00" offset="0.1"/>
            <stop stop-color="forestGreen" offset="0.7"/>
            <stop stop-color="darkGreen" offset="1.0"/>
        </radialGradient>
        <linearGradient id="metal" spreadMethod="repeat"
                        gradientTransform="scale(0.7) rotate(75)">
            <stop stop-color="#808080" offset="0"/>
            <stop stop-color="#404040" offset="0.25"/>
            <stop stop-color="#C0C0C0" offset="0.35"/>
            <stop stop-color="#808080" offset="0.5"/>
            <stop stop-color="#E0E0E0" offset="0.7"/>
            <stop stop-color="#606060" offset="0.75"/>
            <stop stop-color="#A0A0A0" offset="0.9"/>
            <stop stop-color="#808080" offset="1"/>
        </linearGradient>
    </defs>
    <rect x="20" y="20" width="100" height="280"
          fill="url(#metal)" stroke="black" stroke-width="3" />
    <g stroke="black" stroke-width="2">
        <use xlink:href="#light" y="80"
             fill="url(#red-light-off)" />
        <use xlink:href="#light" y="160"
             fill="url(#yellow-light-off)" />
        <use xlink:href="#light" y="240"
             fill="url(#green-light-on)" />
    </g>
</svg>
```

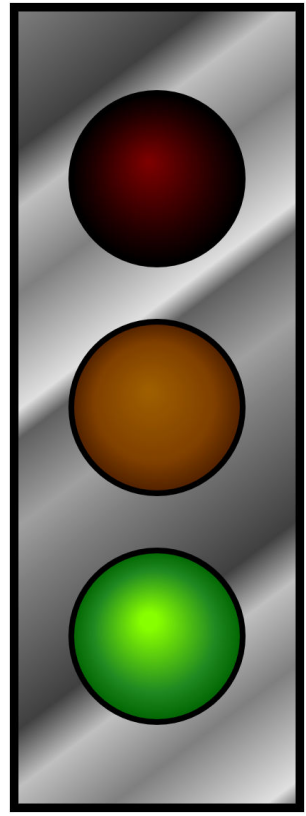


Figure 1-3. Stoplight with gradient fills

The gradients, defined within the `<defs>` section of the file, come in two types: `<radialGradient>` for the circular lights and `<linearGradient>` for the frame. Each gradient element has an easy-to-remember `id` attribute that will be used to reference it.

The radial gradients also have `fx` and `fy` attributes, which create the off-center effect, while the linear gradient contains `spreadMethod` and `gradientTransform` attributes to control the angle, scale, and repetition of the gradient. Each gradient contains `<stop>` elements

that define the color transition. If you absolutely must know more *now*, you can jump ahead to ??? for more details.

Still here? OK, then let's look at the rest of Example 1-5:

```
<rect x="20" y="20" width="100" height="280"
      fill="url(#metal)" stroke="black" stroke-width="3" />
<g stroke="black" stroke-width="2">
    <use xlink:href="#light" y="80"
         fill="url(#red-light-off)" />
    <use xlink:href="#light" y="160"
         fill="url(#yellow-light-off)" />
    <use xlink:href="#light" y="240"
         fill="url(#green-light-on)" />
</g>
```

It's mostly the same as Example 1-4, except for the fill values. Instead of color names or RGB hash values, each `fill` attribute is of the form `url(#gradient-id)`. Why the extra `url()` notation? Partly, it's because presentation attributes need to be compatible with CSS, and CSS uses `url(`*`reference`*`)`. More importantly, it's because `fill` and `stroke` and other presentation attributes can be specified as a URL *or* as other data types, and you need to be able to clearly distinguish between them. Without `url()`, how would you know if `#fabdad` referred to a paint server element or to a light pink color?

To add a bit of realism, the gradients were defined so that the green light appeared to be lit (bright green), while the red and yellow lights were dim (dark maroon and mustard brown). But a *real* stoplight wouldn't stay green all the time.

It's fairly straightforward to edit the code to switch the stoplight to red: copy the red light gradient, change its `id` to `red-light-on`, then change the `stop-color` values to something brighter. Copy the green gradient, change its `id` to `green-light-off`, then change the colors to something darker. Finally, change the `fill` values to reference the new gradients. There: you have a red stoplight. But you still don't have a *working* stoplight. For that, you need animation.

Activating Animation

Animation was a core part of the original SVG specifications. Not only was there the option of animating elements with JavaScript, but there was a way of declaring animations as their own elements.

These animation elements (such as `<animate>` and `<set>`) were based

However, by the time web browsers really started working on their SVG implementations, there was a competing proposal for declarative animation on the web: CSS transitions and keyframe animations. Microsoft browser developers at first refused to implement either for SVG, until a common model was defined.

That combined animations model, the Web Animations framework, is starting to gain traction. But in the meantime, developers of Google Chrome announced that they'd prefer to just get rid of all their SMIL-related implementation code. (The wider concept of SMIL, for synchronizing multimedia, had never caught on in browsers and was made obsolete by HTML 5 audio and video elements.) Chrome hasn't removed support, yet, but new implementation and bug fixes for SVG animation elements has stalled across the board.

Sigh… The web ain't easy. We'll talk a bit more about your animation options in ???, or you can keep your eyes out for Sarah Drasner's forthcoming *SVG Animations* book for more.[1]

The short version: CSS animation is not yet a full replacement for the SVG/SMIL animation elements—but for the animations it can handle, it currently has the better browser support.

For our animated stoplight, there are a few different ways to approach the problem with CSS. Option 1 is to directly animate the `fill` property. That works fine for solid-color fills. But browsers are currently buggy about animation when the fill value is a `url()` reference to a paint server. As an alternative, we can create *two* versions of each light, one with the "off" gradient and one with the "on" gradient, layered on top of each other. Then we can animate the visibility of the top layer.

Example 1-6 provides the code for implementing this approach: first the markup for the layered structure, then the CSS code that brings it to life. The CSS code is contained in an SVG `<style>` element, which is much the same as an HTML `<style>` element. It allows us to include `@keyframes` rules for the animation, and also to assign

1 *http://shop.oreilly.com/product/0636920045335.do*

styles by class. Figure 1-4 shows the three states of the stoplight—but to get the full effect, run the code in a web browser!

Example 1-6. Animating the vector graphic stoplight using CSS keyframes

```
<svg xmlns="http://www.w3.org/2000/svg" xml:lang="en"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     height="320px" width="140px" >
    <title>Animated Stoplight, using CSS Keyframes</title>
    <defs>
        <circle id="light" cx="70" r="30" />
        <radialGradient id="red-light-on" fx="0.45" fy="0.4">
           <stop stop-color="orange" offset="0.1"/>
           <stop stop-color="red" offset="0.8"/>
           <stop stop-color="brown" offset="1.0"/>
        </radialGradient>
        <radialGradient id="red-light-off" fx="0.45" fy="0.4"> ❶
            <stop stop-color="maroon" offset="0"/>
            <stop stop-color="#220000" offset="0.7"/>
            <stop stop-color="black" offset="1.0"/>
        </radialGradient>
        <!-- More gradients -->                                ❷
    </defs>
    <style>
        /* CSS styles (see below) */                          ❸
    </style>
    <rect x="20" y="20" width="100" height="280"
          fill="url(#metal)" stroke="black" stroke-width="3" />
    <g stroke="black" stroke-width="2">
        <g class="red light">                                 ❹
            <use xlink:href="#light" y="80" fill="url(#red-light-off)" />
            <use class="lit"
                 xlink:href="#light" y="80" fill="url(#red-light-on)" />
        </g>                                                  ❺
        <g class="yellow light">
            <use xlink:href="#light" y="160" fill="url(#yellow-light-off)" />
            <use class="lit"
                 xlink:href="#light" y="160" fill="url(#yellow-light-on)"
                 visibility="hidden" />                       ❻
        </g>
        <g class="green light">
            <use xlink:href="#light" y="240" fill="url(#green-light-off)" />
            <use class="lit"
                 xlink:href="#light" y="240" fill="url(#green-light-on)"
                 visibility="hidden" />
        </g>
    </g>
</svg>
```

❶ New radial gradients are added to represent the lit and off states of each light.

❷ But to keep this example short, the repetitive code isn't printed here; all the gradients follow the same structure, just with different colors and different `id` values.

❸ The `<style>` element can be included anywhere, but it's usually best to keep it before or after the `<defs>`, near the top of the file.

❹ The changed markup replaces each light in the stoplight with a group (`<g>`) containing two different `<use>` versions of the circle. The first one (bottom layer) has the "off" gradient. Each group is distinguished by a class describing which light it is.

❺ The second `<use>` in each group (top layer) has the "on" gradient. It also has the class "lit" so that we can access it from the CSS.

❻ The "lit" layers for the green and yellow lights are hidden by default, using the presentation attribute for the `visibility` property. We use `visibility` (and not `display`) because `display` cannot be animated with CSS. We use presentation attributes (and not inline styles), so that our CSS rules will override them: these are just the default values that apply if CSS animations are not supported.

```
@keyframes cycle {
    33.3% { visibility: visible; }
    100%  { visibility: hidden;  }          ❶
}
.lit {
    animation: cycle 9s step-start infinite; ❷
}
.red    .lit { animation-delay: -3s; }
.yellow .lit { animation-delay: -6s; }      ❸
.green  .lit { animation-delay:  0s; }
```

❶ The animation states are defined with a `@keyframes` block, which names this animation `cycle`. There are two states in the animation: hidden and visible. The time selectors say that after one-third (33.3%) of the animation cycle, we want the light to be visible, and at the end of the cycle we want it to be hidden.

❷ The animation is assigned to all the layers with class "lit" using the shorthand `animation` property. Translated to English, the value means: "use the `cycle` animation keyframes; advance through all the keyframes in a 9-second duration; for each frame, jump immediately to the new value at the start of each frame's time period; repeat the entire animation infinitely." The `step-start` value is important for the way we've defined the keyframes: the animation will *start* in the `visible` state, and switch to the `hidden` state as soon as the 33.3% time point is past.

❸ All the lights have the same animation keyframes, but we don't want them all to turn on and off at the same time. The `animation-delay` property staggers the animation cycles for each light. Negative values mean that the animation starts running, from a point partway through, when the file loads. The delay offsets are multiples of one-third of the 9s total cycle time, matching the proportions used in the keyframes.

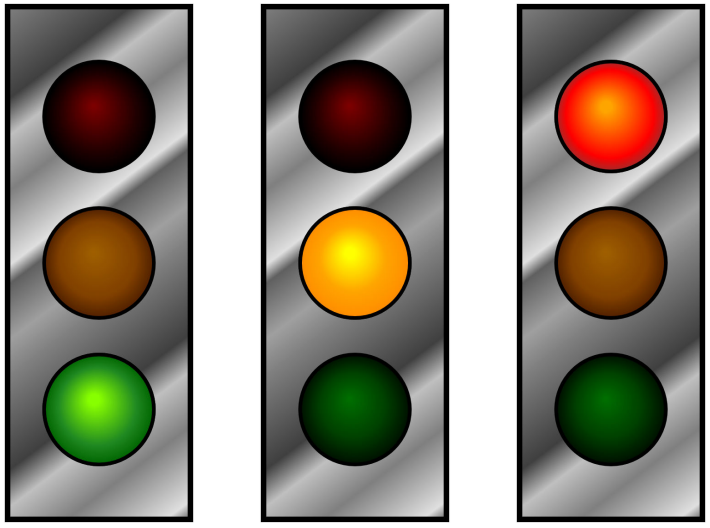


Figure 1-4. Three stages of an animated stoplight with gradient fills

Be aware that this animation still won't display in Internet Explorer browsers (it works fine in MS Edge), or in other older browsers. As we discuss in Chapter 2, some other browsers have issues when the SVG is embedded as an image.

You *could* increase support in some older browsers by adding prefixed versions of the animation properties (like `-webkit-animation-delay`) and by duplicating the keyframes rule and giving it a prefix, too (`@-webkit-keyframes`). But the number of users who would benefit from this code duplication decreases every month, and the hassle of maintaining it remains the same!

For a purely decorative effect, the lack of perfect browser support may be acceptable. You still get a stoplight in the other browsers; it's just stuck on the red light. In other cases, however, the animation is essential to your content, and you will need to use JavaScript to create the animated effect. We'll explore adding JavaScript to the stoplight example in Chapter 2.

Chapter 2 will also show how SVG images can be integrated into text-heavy HTML web pages. HTML is not *required* to associate words with your graphics; SVG can display text directly. However, displaying text in SVG has as much (or more) in common with drawing SVG shapes as it does with laying out text documents with HTML and CSS.

Talking with Text

Although it may not be immediately obvious, text has a significant role in the realm of graphics, and a surprisingly large amount of the SVG specification is devoted towards the placement of and display of text.

When the information in your graphic is essential, you often need to spell it out in words as well as images. Metadata such as the `<title>` element can help, especially for screen readers, but sometimes you need words on screen where everyone can see them.

Drawing text in an SVG is done with the creatively-named `<text>` element. We'll take more about text in ???, but the basics are as follows:

- The words (or other characters) to be drawn are the *child content* of the element, enclosed between starting and ending `<text>` and `</text>` tags.
- The text is positioned (by default) in a single line around an *anchor* point; the anchor is set with `x` and `y` attributes.
- The text is painted using the `fill` and `stroke` properties, the same as with shapes, and not with the CSS `color` property.

Example 1-7 shows the added or changed code, relative to Example 1-6. Figure 1-5 shows the three states of the animated result.

Example 1-7. Adding text labels to the animated stoplight

```
<svg xmlns="http://www.w3.org/2000/svg" xml:lang="en"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     height="320px" width="400px" > ❶
```

❶ To make room for the labels, the width of the graphic has been increased.

```
text {
    font: bold 60px sans-serif; ❶
}
```

❶ A new CSS rule (added to the `<style>` block) assigns font styles to all the `<text>` elements, using the same shorthand `font` property used in the rest of CSS text styling.

```
    <g stroke="black" stroke-width="2">
        <g class="red light">
            <use xlink:href="#light" y="80" fill="url(#red-light-off)" />
            <g class="lit" fill="url(#red-light-on)">     ❶
                <use xlink:href="#light" y="80" />
                <text x="140" y="100"
                      stroke="darkRed">STOP</text>        ❷
            </g>
        </g>
        <g class="yellow light">
            <use xlink:href="#light" y="160" fill="url(#yellow-light-off)" />
            <g class="lit" visibility="hidden" fill="url(#yellow-light-on)" >
                <use xlink:href="#light" y="160" />
                <text x="140" y="180"
                      stroke="darkOrange">SLOW</text>     ❸
            </g>
        </g>
```

```
    <g class="green light">
        <use xlink:href="#light" y="240" fill="url(#green-light-off)" />
        <g class="lit" visibility="hidden" fill="url(#green-light-on)" >
            <use xlink:href="#light" y="240" />
            <text x="140" y="260"
                  stroke="darkGreen">GO!</text>
        </g>
    </g>
</g>
```

❶ Each "lit" version of the light has now been grouped together with a matching text label. The class has been moved to the `<g>` element, so that the entire group will be hidden or revealed as the animation changes the lights. The `fill` presentation attribute has also been moved to the group: both the shape and the text will inherit the value.

❷ The `<text>` elements each have an `x` value that positions them to the right of the stoplight, and a `y` value that positions the base of the text near the bottom of the circle. The text elements also have a solid-colored stroke assigned directly with a presentation attribute.

❸ The remaining lights follow the same structure, except that the lit layer, including the label, is hidden by default.

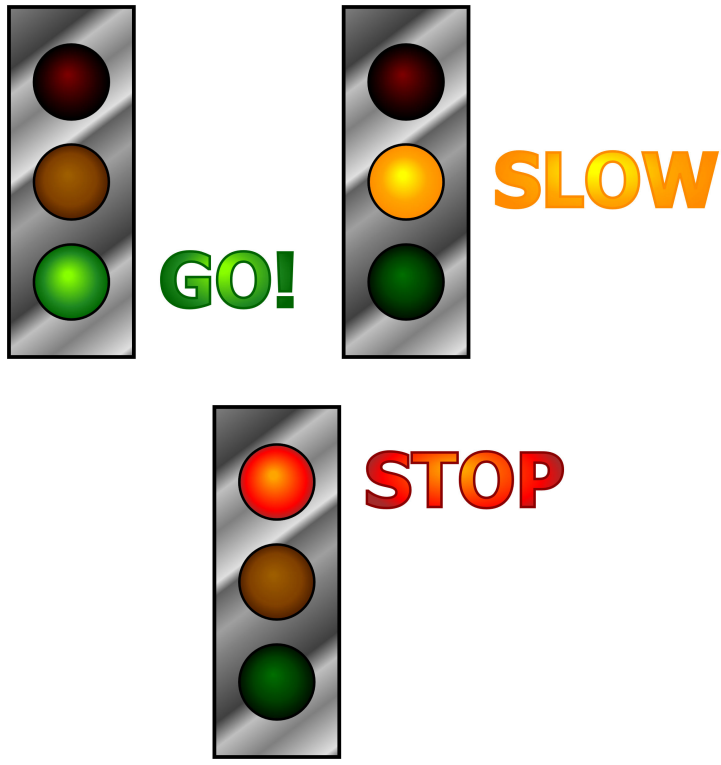


Figure 1-5. Three stages of a labelled, animated stoplight

The stoplight example now contains shapes, paint servers, text, animation: most of the key features used in SVG clip art, icons, or data visualizations. Of course there's much more to learn—this is only Chapter 1! But, by now you should understand enough to start making tweaks and adjustments to a clip art SVG file—or one you created with a drawing program—by editing the code directly.

Equally importantly, you should be starting to understand how SVG works as a structured graphical document, not simply as a picture. The drawing is divided into meaningful parts, and these parts can be styled or modified independently.

To understand how SVG balances its dual nature as a document and as an image, it helps to step back and think about how vector graphics work overall. The stoplights we've been drawing so far are all vec-

tor graphics, but what does that mean? How do these SVG graphics differ from the bitmap images that you can create in a basic Paint application?

Understanding Vector Graphics

Vectors, in mathematics, are numerical representations of movement or change, of how to get from here to there. Usually represented as a list of numbers, they break down the overall movement into the total displacement in each dimension. For 2D graphics, that's usually horizontal (x) and vertical (y).

Vector graphics consist of mathematical descriptions of shapes as a combination of these numerical vectors. Because the vector graphics define the *path* that the lines should take, rather than a set of points along the way, they can be used to calculate any number of points, and so can be used at any image resolution.

Vector graphics form the basis of many computer applications, and they've been around almost as long as the personal computer. The DWG ("drawing") format—used by AutoCAD and other computer-assisted drafting software—and the PostScript language—used in Adobe's desktop publishing software—were both developed in the early 1980s. For drafting, the mathematical precision of vector graphics is essential. For publishing, the key benefit was the ability to create high-resolution printed documents, without requiring the position of every drop of ink to be encoded in the computer file.

Computer printers do not (usually) draw vector graphics directly, tracing along the lines like you would draw a shape with a pen. However, by using a set of vector shapes to define the boundaries of what a letter or image looks like, a program can test whether any given point is within that boundary. As the printer head moves across the page, the software tells it whether to drop or not drop a spot of ink at any given point.

The same principles are used in TrueType and other vector font formats: rather than storing separate, pixelated representations for each possible size of text, vector fonts only need one set of shapes regardless of size. The software can calculate which pixels on the screen would fit within the letter outlines, in a process called *rasterization*. The word comes from the latin for "rake", and refers to the way the

image is calculated as a series of parallel lines, like the lines made by the tines of a rake across the ground.

This description of vector fonts is an over-simplification; most font formats also include hints or constraints that are used to adjust the appearance of the letters at low resolutions.

Figure 1-6 simulates the rasterization process. The gray dashed line represents the vector outline of the shape, which is the same mathematically-defined curve regardless of the display resolution. The dots of ink, or pixels of light, fill in the shape in neat rows. The number of rows and dots that fill in the shape are quite different between a low resolution and a high resolution printer or display. However, the overall size and shape is the same, except for a slight jaggedness (pixelation or aliasing) around the edges on the low-resolution display, caused by the fixed size and spacing of the dots. This jaggedness is often minimized by partially coloring the edge pixels to smooth out the curves (anti-aliasing).

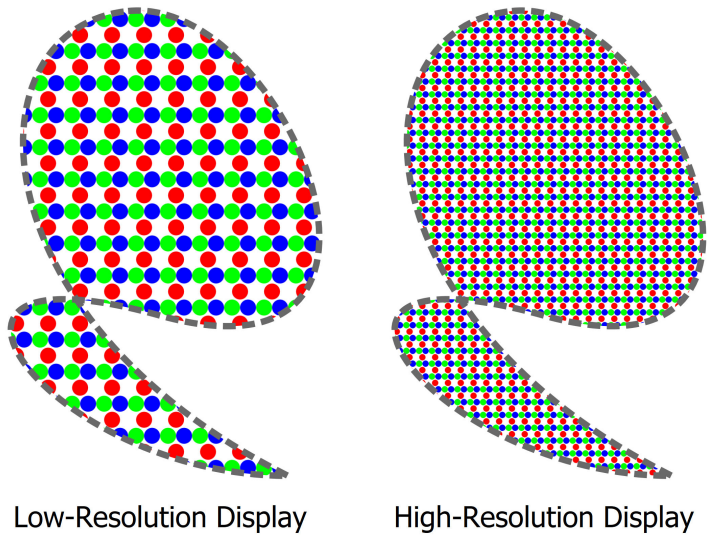


Figure 1-6. A vector graphic rasterized at different resolutions

Raster graphics are the opposite of vector graphics; these are directly encoded as the rows (raster lines) of final pixel colors for each point in the image grid. Viewed at the correct resolution, the human eye connects up those points into lines and shapes, but the underlying graphic doesn't include any information about those shapes.

Raster graphics are also known as *bitmap* graphics, because originally the data (bits) in the file could be directly mapped to corresponding pixels in the output. Today, most raster graphic formats employ some form of compression, to reduce the total number of bytes necessary to encode the graphic.

When SVG was first proposed in the late 1990s, bandwidth was at a premium. The principle advantage of vector graphics on the web was that high-resolution (but graphically simple) images could be defined with small file sizes. You could display the graphics at any size, for screen or print, without causing smooth curves or diagonal lines to degrade into jagged steps. The images scaled cleanly whether they were rendered at 1% of the original size or 100 times the original size. Mathematical transformations—scaling, rotating, or skewing—of the image did not change the underlying information about the graphic itself.

Raster graphics, on the other hand, are considerably more sensitive to scaling. If you double an image's dimensions, the display will have to interpolate new color values from the old, making the image appear soft and out of focus. If you then reduce that image to one half its original size, that same graphic has to throw out information. Take that reduced image and blow it up by a factor of four, and the image ends up interpolating data that may or may not be accurate, and the image gets even farther out of focus. Rotations can create jagged artifacts, skewing can introduce more artifacts, and successive transformations can leave an image looking like it went through a wormhole in space.

In other words, instruction-based (or declarative) vector graphics are scalable, while raster graphics for the most part are not. Thus, *Scalable Vector Graphics* are images that are described by instructions or functions rather than rasters, that can scale in response to transformations without losing information, and that typically require smaller files to encode content.

The same is true for declarative animation, which adds a new dimension to the vectors—time! Declarative vector animations

(including both CSS animation and SMIL animation elements) define how a property should change over time, and let the browser fill in the specific in-between values. While videos and other animated image formats (such as GIF) define a fixed number of individual frames, vector animations will adapt in frame rate according to the capabilities of the device. Slowing down or speeding up the animation is also simple, just like scaling the image in space.

Because vector graphics languages describe images and animation as a series of discrete instructions for each shape or line, they could logically be translated into an element-based document object model of the sort used for HTML and XML. And that's where SVG comes in.

The SVG Advantage

The basic concept for SVG is simple: use the descriptive power of XML to create overlapping lines, shapes, masks, filters, and text that —when combined—create illustrations. In computer graphics terms, these shapes are either predefined (rectangles, circles, ovals, and so forth), or are constructed by sequences of vector instructions.

The SVG specification, originally finalized in 2002, was complex and extensive. The specifications include more than just the XML markup. They define many new CSS style properties for SVG content (some of which have since been adopted for CSS styling of other content) and a complete set of custom DOM interfaces for manipulating SVG elements.

It has taken time for SVG to reach its potential. Some would argue it is not there yet. Unlike HTML, SVG did not develop in concert with extensive real-world experience from software implementations and web designers.

For the first few years, there were only two implementations of the complete standard: the Adobe SVG viewer, a plug-in for Internet Explorer, and the Apache Batik Squiggle viewer, an open source Java-based tool. Limited implementations of SVG were available in other tools, however, including a version integrated into Mozilla Firefox in 2003.

The Adobe SVG viewer was discontinued after Adobe merged with Macromedia, makers of Flash. Batik remains of limited use, primar-

ily as a component of other Java-based tools. However, by 2009, all major browsers either had or were planning native (no plug-in required) SVG implementations. The browser-based SVG tools have only recently reached the performance and support levels of the old Adobe plug-in. This delay resulted in a shift in focus: from extending SVG as a stand-alone dynamic graphics tool, towards integrating SVG within the rest of the web platform.

Bandwidth availability has improved dramatically since SVG was first proposed. On most broadband networks, the time to download a large raster diagram is comparable to the time for a vector graphics program to calculate how to render a complex image. Nonetheless, the benefits of SVG go well beyond file size:

Familiar syntax
: SVG is XML content, so it can be generated by external data feeds and processes on the fly, making it a natural part of web server pipelines. It can also be integrated within other XML document types, which includes the file formats used by major word processing and publishing systems.

  Because SVG is XML, this means that it can be edited in any XML editor, several of which are remarkably sophisticated. Most of the examples in this book were written using the OxygenXML or Brackets code editors, which include visualization tools to display the SVG even as you type the code. Given that SVG has an established grammar and schema, these tools can also check for syntax errors or help fill in values quickly.

Dynamic and interactive
: The vector elements in SVG describe not only what the graphics *look like* but also what they *are*. If the elements are modified, their appearance can be re-calculated. SVG on the web can be interactive and dynamic, using scripts to manipulate the document in response to user interaction or based on data retrieved from separate files or web services.

  Even without JavaScript, SVG can be dynamic: animation elements and CSS selectors can show, hide, or alter content as the user interacts with the graphic. And of course, SVG can be hyperlinked to other documents on the Internet.

Accessible and extendable
: SVG supports text-based metadata, not just about the image as a whole, but about individual components within the picture. Maps can internally identify roads, buildings, geographic boundaries, and more; diagrams can provide relevant explanatory and even interactive information; metadata systems can read an SVG document and derive from it a very rich and sophisticated view of the meaning behind the image.

Resolution-independent
: SVG, as a vector format, automatically adapts to the capabilities of the display hardware. There is no need to create new files for the latest higher resolution screens. When working with SVG, you can apply and undo infinite transformations or filter effects without any irreversible degradation of image quality.

It is worth noting that many raster image tools also provide support for rudimentary vector graphics encoding; even photo-processing applications like Photoshop or GIMP use vector graphics to combine layers, manipulate text, or apply some effects. However, most of these vector encodings are not generally available to end users or developers—they are locked up deep within the application layers of these respective programs. (To the extent that the vector data *is* available, it is often via SVG export!)

In a similar way, SVG can incorporate bitmap graphics within specific layers. Such bitmaps do not get the scaling benefit of vector graphics; while they can certainly be scaled, the images will lose resolution the same way scaling any bitmap will. However, the SVG code can be used to modify bitmap images or composite multiple bitmaps—mask areas off, scale areas, apply underlays or overlays, generate drop shadows, and so forth—and these effects can be applied regardless of the resolution of the bitmap image. This makes it possible to build layout tools for raster graphics with SVG.

Nonetheless, the biggest advantage for SVG on the web remains the way in which it is integrated with other web platform languages. Portable Document Format (PDF) files, which can contain PostScript vector graphics code, are widely available on the Internet. But PDF documents exist separate and apart from the web sites that link to or embed them. SVG images, in contrast, are part of the web, and can interact with other web technologies such as HTML, XML, CSS, and JavaScript.

Summary: An Overview of SVG

This chapter has breezed through many different features of SVG, and skipped over many more. The intent has been to give you the lay of the land, so you can keep your bearings as we start exploring in detail.

One of the key ideas, beyond the general structure of SVG and its element or attribute names, is that SVG can (and in many cases should) be approached programmatically. There are often multiple ways to create the same picture, but each will differ in how they can be used. Creating effective interactive applications with SVG requires seeing the language as being, like HTML, a complex toolset of interconnected parts.

While you can use tools such as Adobe Illustrator or Inkscape to draw graphical pieces, the language comes into its own when you treat it as a powerful way to build interfaces—widgets, maps, charts, game controls, and more. Although SVG can replace icons or art that you currently represent as static images (or animated GIFs), the true advantages of SVG come in the ways it is different from any other image type—in particular, in the ways it interacts with other web design languages.

CHAPTER 2

The Big Picture

SVG and the Web

In the last few years, a quiet revolution has been taking place in web browsers and operating systems. Solid implementations of SVG have become standard in both desktop and mobile browsers.

As this has happened, web developers and designers have become more confident in using SVG to display content that moves beyond HTML layout. They have also been combining SVG with the power of the newer, efficient and standardized JavaScript engines to build sophisticated information graphics and interactive games. Anyone working with data visualization on the web is now gaining familiarity with SVG as a tool.

The history of SVG has not been straightforward. As with HTML, CSS, and the other standards that make up the web, the development of SVG has been a process of back-and-forth compromises between the authors of specifications, the builders of web browsers that implement them, and the designers of web pages that use them. Unlike those other languages, however, the SVG specification did not develop slowly and incrementally—it was created fully formed, as an incredibly complex graphics language.

If you work with HTML and CSS web design, you will find many aspects of SVG familiar—and a few quite different. The SVG standard was built upon other web standards, most notably XML and CSS, and has a complex DOM that can be manipulated with JavaScript. In that way, it is very similar to HTML. But because the pri-

mary focus of SVG is graphics, not text, it intersects and connects the parts of the web platform that you usually try to keep separate: content, formatting, and functionality.

This chapter starts with a refresher about the main web languages and their separate roles. It then looks at how SVG interacts with these languages. We adapt the stoplight example from Chapter 1 to show how you can build on a simple SVG to create complete web pages.

The Web Platform

The web platform—the foundation on which all web sites are built, from global forces like Google and Facebook to niche blogs and web apps—consists of a half-dozen core coding standards, and countless extensions of them. Each standard is a plank that connects the browsers (and other software) that interpret the web page code to the developers that write it. So long as both ends, browsers and developers, stay firmly attached to a common standard, the structure stays sound and web sites function as expected in any browser, on any operating system.

However, all of these components—standards, browsers, and web sites—are in constant flux, adding new features and building new structures. If a browser doesn't quite reach the level of the standard a developer is using, something needs to be cobbled together to connect them, or the page ends up broken in appearance or function. In many ways, the web platform is more of a ramshackle raft than anything you would want to build a house upon. But somehow it all hangs together, most of the time.

If you are only interested in SVG as an image format—as a tool to create static, unchanging pictures—you don't need to worry too much about the other web standards. To take full advantage of SVG as a graphical web application, however, you will need to leverage the entire web platform to build and extend your graphics. You'll need to understand what each language adds to the web in general, and to SVG in particular.

Since the web standards movement first gained hold in the late 1990s, a key organizing principle—to help control the chaos—has been the *separation of concerns*. Different aspects of a web site or application should be kept in separate code blocks or files, in order

to support the greatest flexibility and re-usability of code. Figure 2-1 visualizes how the main web languages each have their separate, but interconnected, concerns.

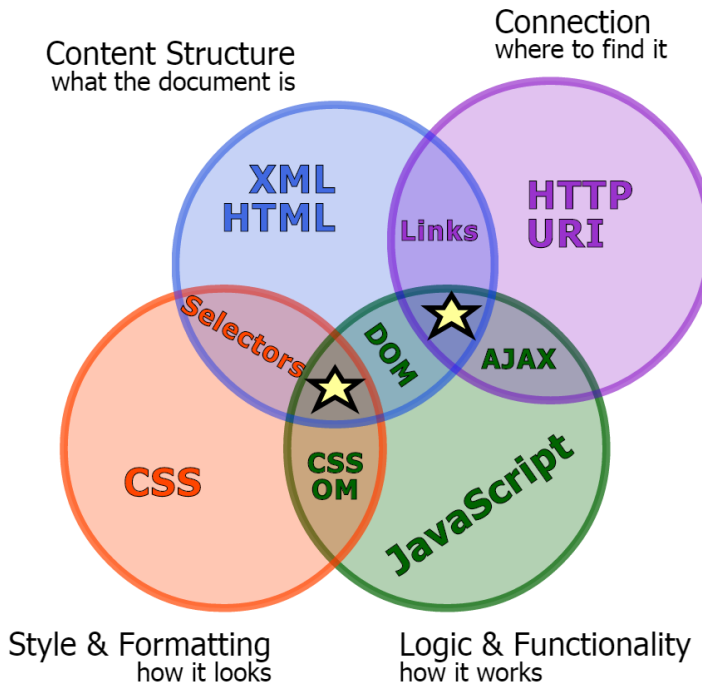


Figure 2-1. The core web platform languages, their roles and their areas of overlap. Web pages and web sites exist at the intersection of all four regions.

Each component has a clear role to play:

HTML and XML
: The language most strongly associated with the web is HTML, HyperText Markup Language. *Hypertext* meaning that the text in one document is connected to further information stored elsewhere; *markup* meaning that the structure is described by codes included in the main text. Not all information can be represented as HTML, whose markup tags are designed to describe

text documents. The eXtensible Markup Language (XML) allows diverse data and document formats to be represented with a consistent syntax. Programs to edit, validate, and manipulate XML documents only needed to know the XML syntax rules, not the particulars of the document type. XML, while the basis of countless file types, has not—as initially planned—replaced the looser syntax of HTML; nonetheless, a document can be both HTML and XML compatible when required, a format known as XHTML.

HTTP and URI

Web pages may be written in HTML, but the web is *connected* with the Universal Resource Identifier system (URI, formerly Universal Resource Locators, URL) that identifies resources on the web, and the HyperText Transfer Protocol (HTTP) that transmits them from server to browser.

These standards overlap the HTML language in one area: links. The URI/URL addressing system identifies hyperlink destinations and embedded resources such as images. HTTP mostly operates behind the scenes, in the browser source codes. Front-end web page developers should at least be aware of the HTTP form-submission methods (GET and POST) for sending data to the server, and the HTTP header system for identifying files from the server and for sending user information (e.g., web cookies) with file requests.

CSS

Cascading Style Sheets (CSS) were developed as an alternative to the formatting-focused HTML tags that flourished in the early years of the web. CSS offered two main benefits: presentation information could be separated into style blocks or stylesheet files, and multiple stylesheets could be applied to a single document.

By separating the styles, one stylesheet could be used for many web pages on a site, and the HTML markup could focus on the content. By *cascading* style rules through multiple stylesheets, users could retain some control over appearance, by setting their own defaults and by marking certain features as `! important` if, for example, poor eyesight makes large text or high-contrast colors essential.

The benefits of CSS were such that HTML 4.01 deprecated—discouraged the use of—formatting markup when it was finalized in 1999 by the World Wide Web Consortium (W3C). Web designers were, and are, advised to separate structure (HTML) from presentation (CSS) as much as possible, connecting the two only by the selectors in the CSS and the class attributes in the HTML. Ideally, it should be possible to completely change the style of a web page without altering the HTML.

JavaScript

Dynamic web pages, which could react to user actions in the same way desktop computer programs did, were another new feature developed by early web browsers trying to distinguish themselves. Netscape Navigator 2 was the first to include programming code, using JavaScript, a flexible, informal scripting language whose syntax was (very) loosely inspired by the then-dominant Java. Other browsers developed similar-but-different scripting rules.

While HTML and CSS were standardized by the World Wide Web Consortium (W3C), scripting was standardized by the organization ECMA International (formerly the European Computer Manufacturers Association) and is officially known as ECMAScript. This book will follow the terminology used by the vast majority of web developers, and call it JavaScript. Thankfully, most browsers now support the ECMAScript standards, so the distinction is no longer relevant.

ECMAScript only provides half of the functionality required for dynamic HTML. It defines basic data types, functions, and control structures. It *doesn't* define how the parts of a web page work. The Document Object Model (DOM) connects JavaScript to HTML/XML. It defines a web document, not as a stream of text with formatting marks, but as a collection of data objects. The DOM specifications define how these objects are nested within one another, along with their properties and methods. Unlike JavaScript, the DOM was defined via the W3C, originally with the intent it could be implemented in any programming language. There is also a CSS object model (CSS OM) that allows scripts to manipulate the rules in a stylesheet.

Web page scripts can also use HTTP to access other files or database gateways on the Internet. Commonly known as AJAX,

for *Asynchronous JavaScript And XML*, the technique relies on the `XMLHttpRequest` API, first introduced as an add-on to Internet Explorer but now widely supported and being standardized. [1] Although initially intended for accessing XML data files, it can import any information accessible via HTTP, and can manipulate the HTTP headers sent with the request.

The web exists at the intersection of these components (in the starred regions of Figure 2-1). Individual web pages are built from HTML/XML, CSS, and Javascript. Complex web sites are held together by hyperlinks, AJAX, URIs and HTTP.

Where does SVG fit on the web platform? SVG is an XML langauge, describing a structured document. However, because SVG is a *graphics* format, the structure of the document cannot be separated from its visual presentation. SVG elements represent geometric shapes, text, and embedded images that will be displayed on screen according to a clearly defined geometric layout. Other markup defines complex artistic effects to be applied to the graphical elements.

SVG exists because not all documents can be displayed with HTML and CSS. Sometimes content and layout are inseparable. In charts and diagrams, the position of text conveys its meaning as much as the words it contains. Other meaning is conveyed by symbols, colors, and shapes—without any words at all. Any complete representation of the content of these documents includes the layout and graphical features.

The same could be said about any image, and images have been part of web pages since the early days of HTML. The W3C even standardized an image format (Portable Network Graphics, PNG) that encodes icons and diagrams in compact files and can be used royalty-free by any software or developer.

The difference is that all other image formats on the web are displayed as single, complete entities. SVG, as an XML document, has structured content with a corresponding DOM. Stylesheets and

1 The standard definition of `XMLHttpeRequest` is at *https://xhr.spec.whatwg.org/*, managed by WHATWG, the Web Hypertext Application Technology Working Group, which is now also a key driver of HTML and DOM standardization.

scripts can access and modify the components of the graphic. Search engines and assistive technologies can read text labels and metadata.

SVG on the web can be used as independent files, with hyperlinks to other files and JavaScript-based interaction. However, SVG was not designed to display large blocks of text. Nor does it include form-input elements or other specialized features of HTML. The most effective use of SVG is therefore not as a *replacement* for HTML, but as a complement to it. SVG web applications are nearly always presented as part of larger HTML web pages, either as embedded objects or—with increasing frequency—directly included inline within the HTML markup.

SVG and the Modern Web

As SVG has been integrated into web pages, the web pages themselves have changed. The HTML 5 specification and the many CSS Level 3 (and beyond!) specifications have significantly changed the relationship between SVG and the rest of the web platform. The dramatically improved performance of JavaScript since 2009 and the rise of JavaScript-based SVG libraries, such as Snap.svg and D3.js, are smoothing away many of the rough edges between implementations.

This book uses the terms HTML 5 and CSS 3 fairly generically. One of the main references for HTML, the WHATWG Living Standard, doesn't use any version numbers; the competing spec at W3C is progressing through version numbers 5.1 to 5.2. These incremental changes can all be thought of as HTML 5+.

On the CSS side, the Level 3 and 4 specifications of existing features are being developed in sync with Level 1 and 2 specifications for new features. All of these (essentially, anything beyond the old CSS 2.1 specification) can be thought of as CSS 3+.

Nonetheless, when working with SVG in the browser, one must always be aware that implementations of the SVG standards are imperfect, incomplete, and frequently changing. This becomes all the more important when taking advantage of the features of HTML

5 and CSS 3 that directly integrate SVG. Cross-browser compatibility is an issue not only as it relates to support of the SVG standard, but also as it relates to support for new features in HTML, CSS, DOM, and JavaScript.

At the time of writing, the best supported version of SVG is 1.1; this specification was created in 2005, without adding any major new features to the original SVG standard. A proposed SVG 2 standard was published in September 2016. It adds many commonly requested features, clarifies numerous details, and improves coordination with HTML and CSS. Other more advanced SVG features are being proposed through additional modules.

There have been other SVG efforts, but these have not had a significant impact on the web. The draft SVG 1.2 standard was ambitious, adding features that would put it in line to replace Microsoft Powerpoint, as well as advanced vector manipulation of graphics. It was abandoned as unworkable when it became clear that the future of SVG would be in the browsers, not specialized software.

A simplified standard, SVG Tiny, was developed for mobile devices. The SVG Tiny 1.2 standard *was* finalized, with some new features from the main SVG 1.2 proposal; however, it fell out of favour as mobile browsers shifted towards displaying standard web pages.

The other web platform languages have also been revised, in parallel to the SVG work. Figure 2-2 sketches out a rough timeline of the past quarter-century of web standards. At the time SVG 1.1 was finalized, the established standards in other areas of the web included CSS level 2, DOM level 2, and ECMAScript (ES) level 3. Many SVG-centric tools, such as Apache Batik and libRSVG, have not significantly updated their CSS implementations since then, nor (in the case of Batik) their DOM and JavaScript implementations. In contrast, as of SVG 2's publication, the latest web browsers all support DOM level 3 (and much of DOM 5) and ES 5.1 (and some ES 6), as well as at least some CSS 3+ features. And more features are added with every browser update.

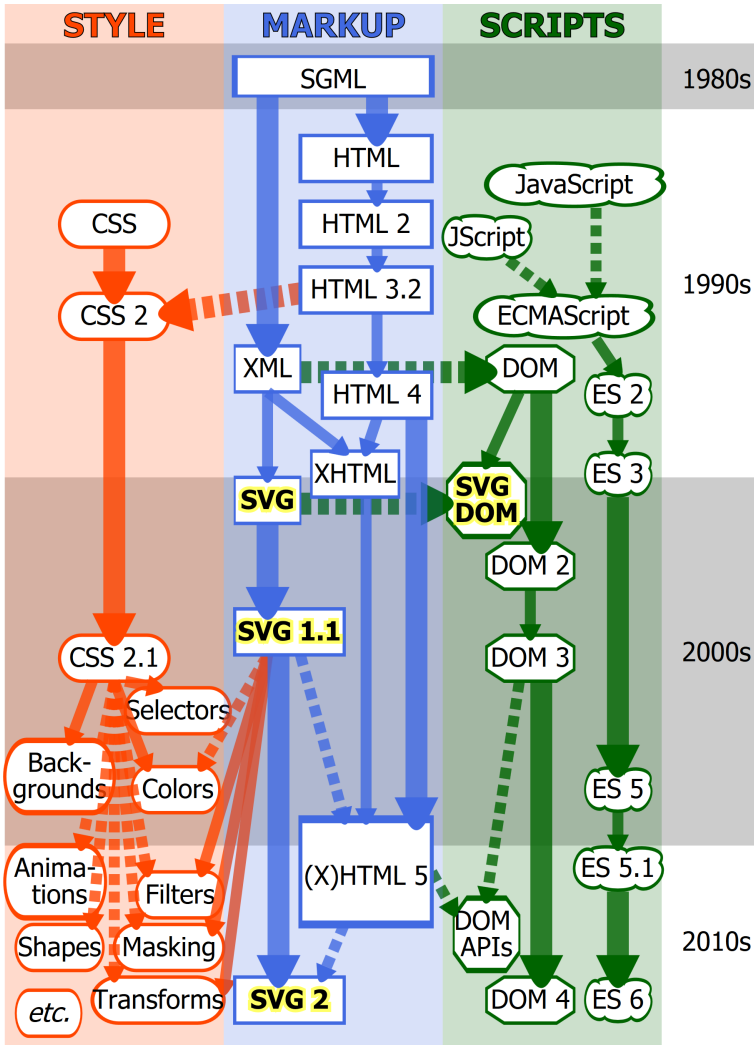


Figure 2-2. Timeline of Web Platform Standards. Solid arrows indicate direct extensions of existing standards; dashed arrows represent more indirect inspiration. Specifications that were abandoned, such as SVG 1.2 or ECMAScript (ES) 4 are not included.

This book focuses on SVG in the web browser, and it will often take advantage of the new features from other web specifications. However, many older web browsers and other software are still in use.

We will identify areas where you're likely to stumble across backwards-compatibility issues, and suggest workaround or fallback options.

This book is also very aware of the fact that SVG is still developing. New CSS modules are extending SVG functionality, and SVG 2 introduces a variety of changes. Throughout the book we will highlight these proposed features which, while not quite ready for production work, are useful to keep in mind as you learn the language. A graphic that is difficult to create now may be much easier with a new tool. The possibilities will be highlighted with "Focus on the Future" side bars like the following:

Future Focus

A Crystal Ball

In these boxes, we'll try to predict the future of SVG. Sometimes the predictions will be clear, because there is wide agreement about what features should be adopted, and it's just a matter of waiting for wider browser support. Other times, the image in our crystal ball will be murky and out-of-focus, because different proposals are still being debated. But in either case, you should be aware that some of the recommendations and best practices discussed in this book may change as SVG matures and adapts to its role on the web.

While there is still some inconsistency between the various SVG implementations, most browsers now have sufficient support for static scalable vector graphics and JavaScript-based dynamic content that it is possible to build complete graphical web applications with SVG.

JavaScript in SVG

If you've used JavaScript to create a dynamic HTML page, you can use it to create dynamic SVG. SVG elements inherit all the core DOM methods to get and set attributes and styles.

Of course, there are a few complications: SVG is a namespaced XML language, so you'll need to use the namespace-sensitive DOM meth-

ods when creating elements or setting `xlink` attributes. Also be aware that some methods you may be familiar with from HTML are **not** part of the core DOM specifications, such as the `.innerHTML()` method to parse a string of markup or the `.focus()` method to control keyboard focus. (Focus and blur control have been added to SVG 2, but aren't universally implemented yet.) Even worse, some features such as `.className` look similar but are structured differently.

As a sort of consolation prize for the missing HTML DOM methods, the SVG specifications introduced a variety of SVG-specific methods and properties to make graphical calculations easier. Unfortunately, many of these features were not universally implemented, and some have become obsolete as SVG switches to an animation model more compatible with CSS. Nonetheless, there are still a few useful features that can be relied on in most web browsers.

To include a script within a stand-alone SVG file, you can include a `<script>` element anywhere between the opening and closing `<svg>` tags. To include an external JavaScript file, use an `xlink:href` attribute (*not* `src` like in HTML) to give the file location. If you instead include the script in between the `<script>` tags, remember to wrap it in an XML character data block, so that less-than and greater-than operators do not cause XML validation errors.

You can specify the scripting language using the `type` attribute on individual `<script>` elements, or the `contentScriptType` attribute on the `<svg>` element. However, it really isn't necessary; the default type, `application/ecmascript` (standard JavaScript) is the only type currently supported in most web browsers.

Most browsers—though not necessarily other tools—will also recognize the types `application/javascript` and `text/javascript` as synonyms. But avoid confusion by relying on the default.

We've already alluded to one common use of scripted SVG: cross-browser animation support. Example 2-1 shows JavaScript that could be used to replace the CSS animation from Example 1-6 or Example 1-7 from Chapter 1. The code uses classes to select the ele-

ments, so works with either the simple or labelled SVG markup: just remember to remove the CSS animation code from the `<style>` element! The JavaScript should be inclued at the end of the file (right before the closing `</svg>` tag), inside a `<script><![CDATA[ /* script goes here */ ]]></script>` block.

Example 2-1. Using JavaScript to animate an SVG stoplight

```
(function(){                                                    ❶

    var lights = ["green", "yellow", "red"];                    ❷
    var nLights = lights.length;
    var lit = 2;                                                ❸

    function cycle() {                                          ❹
        lit = (lit + 1) % nLights;                              ❺

        var litElement, selector;
        for (var i=0; i < nLights; i++ ) {                      ❻
            selector = "." + lights[i] + " .lit";
            litElement = document.querySelector(selector);      ❼

            litElement.style.setProperty("visibility",
                (i==lit)? "visible" : "hidden",
                null);                                          ❽
        }
    }

    cycle()
    setInterval(cycle, 3000);                                   ❾

})()                                                            ❿
```

❶ The code is contained in an anonymous function, which creates a closure to encapsulate variables. Although not required, this is good coding practice to avoid conflicts between different scripts on a page.

❷ The `lights` array holds the class names that distinguish each light group. Since the number of lights won't be changing, we can store it in a variable as well.

❸ The red light is initially lit in the markup; this corresponds to index 2 in the `lights` array; JavaScript array indices start at 0 for the first element.

❹ The `cycle()` function will change the lights.

❺ To start the cycle, the `lit` variable is advanced by one; the modulus operator (`%`) ensures that it cycles back to 0 when it reaches the length of the array.

❻ At each stage of the cycle, each color of light will be modified to either hide or show the "lit" graphic.

❼ The "lit" `<use>` element for each color is retrieved using the `querySelector()` method; a CSS selector of the form ".red .lit" will select the first element of class "lit" that is a child of an element with class "red".

❽ The display style property is set to either `block` or `none` according to whether this light should be lit. Modifying the style property sets an inline style, which over-rides the presentation attribute in the markup.

❾ The cycle function is run once to turn the light green, and then is called at regular intervals on a 3-second (3000ms) timer.

❿ The anonymous function is run immediately: the syntax `(function(){ /*...*/ })()` parses and runs the encapsulated code.

Although this isn't a full web application—there is no interaction with the user—it demonstrates how SVG elements can be accessed and modified from a script.

There's nothing SVG-specific about the JavaScript code: it selects elements using CSS selectors and the `document.querySelector()` method, and sets the `visibility` style according to which lit bulb should be displaying. A timer runs the cycle again after every three-second interval.

The `querySelector()` method and its sibling `querySelectorAll()` are convenient ways to locate elements using a combination of tag names, class names, and relationships to other elements. However, versions of Webkit and Blink browsers prior to mid-2015 had a bug that prevented them from working for SVG mixed case tag and attribute names, such as `linearGradient` or `viewBox` in HTML documents. To improve backwards compatibility, use classes on these elements if you need to select them in your code.

JavaScript and SVG can do more than re-create simple animations, of course. Scripts can be used to implement complex logic and user interaction. They are also useful for calculating the coordinates of shapes in geometric designs and data visualizations. You *could* even use JavaScript to implement your own form-input elements or wrapped-text blocks in SVG. But it's generally easier to use HTML for that.

The stoplight graphic now cycles through green, yellow, and red lights when you view it in any modern web browser, and even in Internet Explorer 9 (the earliest IE to have SVG support). However, you won't get *any* animation, in any browser, if you reference the SVG from the `src` attribute of an HTML `<img>` element!

It's an extra complication when dealing with SVG in web pages: the behavior of SVG on the web can be quite different, depending on how the SVG is incorporated in the page.

Embedding SVG in Web Pages

If you're using SVG on the web you'll usually want to integrate the graphics within larger HTML files. There are a number of different ways that SVG can be added to web pages, each of which has advantages and disadvantages.

The most straightforward method is to use a self-contained SVG file as an image, and include it with the HTML `<img>` tag. Example 2-2 provides the code for a super-simple web page using the CSS-animated SVG stoplight from Example 1-6. Figure 2-3 shows the result.

Example 2-2. SMIL-animated stop light as an image in a web page

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>SVG Images within HTML</title>
    <style>
        html, body {
            background-color: lightYellow;
            margin: 0; padding: 0;
            min-height: 100%;
        }
        header {
            height: 10em;
            background: #444;
            font-family: serif;
        }
        header h1 {
            margin-top: 0;
            color: red;
            text-shadow: yellow 0 0 4px, orange 0 0 2px;
            font-size: 400%;
        }
        header img {
            height: 8em;
            float: left;
            margin: 1em 2em;
        }
        p { padding: 0.5em; }
    </style>
</head>
<body>
    <header>
        <img src="../ch01-overview-files/animated-stoplight-css.svg" />
        <h1>Tony's Towing</h1>
    </header>
    <main>
        <p>Main text goes here, but <em>WOW</em>
           look at that image in the header!</p>
    </main>
</body>
</html>
```

Figure 2-3. Sample web page using an SVG image file

There are a couple features to emphasize in this example. First, note that the height of the image is being set in em-units, versus the height of 320px that was set in the SVG file (???). Just like other image types, the SVG can be scaled to fit. Unlike other image types, the image will be drawn at whatever resolution is needed to fill the given size. The width of the image isn't set in Example 2-2; since both height and width were set in the SVG file, the image will scale in proportion to the height.

Or at least, that's how it *usually* works. Internet Explorer scales the width of the image area in proportion to the height you set, but doesn't scale the actual drawing to fit into that area. There's an easy solution, however; it involves the `viewBox` attribute that we'll discuss in ???.

The second thing to note is that this is the *animated* SVG that is embedded. Declarative animation, including CSS animation and SVG/SMIL animation elements, runs as normal within SVG used as images—in browsers that support that animation type at all.

Or at least, that's how it is *supposed* to work. At the time of writing (late 2016), MS Edge will not animate the SVG when it is embedded as an image in a web page, even though it supports CSS animation elsewhere. The same problem exists in Firefox browsers prior to version 51 (stable release at the end of 2016), even though the same versions support CSS animation in regular files *and* SMIL-style animation elements in images!

Why not use the scripted SVG animation from Example 2-1, which has better browser support? Because scripts *do not* run within images. That's not a bug, it's defined in the HTML spec: for security and performance reasons, files loaded as images can't have scripted content.

There are some other important limitations of SVG used as images:

- SVG in images won't load external files (such as external stylesheets or embedded photos).
- SVG in images won't receive user interaction events (such as mouse clicks or hover movements).
- SVG in images can't be modified by the parent web page's scripts or styles.

The limitations are the same if you reference an SVG image from within CSS, as a background image or other decorative graphic (an embedding option we'll discuss more thoroughtly in Chapter 3).

If you want to use an external SVG file without (most of) the limitations of images, you can use an embedded `<object>`, replacing the `<img>` tag from Example 2-2 with the following:

```
<object data="animated-stoplight-scripted.svg"
        type="image/svg+xml" >
</object>
```

You could also use an `<embed>` element or `<iframe>` with the same effect, although different support in older browsers. However, if you use an `<iframe>` it will never automatically scale to match the proportions of your drawing. Objects will scale, but only according to the `viewBox` attribute (which we haven't gotten to yet), but not when

the SVG dimensions have been defined using height and width (as in these examples).

Embedded objects can load external files, run scripts, and (with a little extra work and some security restrictions) use those scripts to interact with the main document. However, they are still separate documents, and they have separate stylesheets.

While embedded objects *should* be interactive and accesible parts of the main web page, just like `<iframe>`-embedded documents in HTML, they can be a little buggy in browsers. Test carefully, including keyboard interaction and screen reader exposure, if you're embedding interactive SVG as an `<object>`.

Using SVG with HTML 5

Perhaps the biggest step towards establishing SVG as *the* vector graphics language for the web came from the W3C's HTML 5 working group. When SVG was first proposed, as an XML language, it was expected that SVG content would be inserted directly into other XML documents, including XHTML, using XML namespaces to indicate the switch in content type. But most web authors weren't interested in adopting the stricter syntax of XML when browsers rendered their HTML just fine.

With HTML developing separately from XML, it could have easily left SVG behind as another too-complicated coding language. Instead, the HTML 5 standard welcomed the idea of SVG content mixed in with HTML markup, just without the need for XML namespaces.

By making SVG a de facto extension of HTML, the HTML 5 working group acknowledged that SVG was a fundamental part of the future of the web. This has had—and will continue to have—a huge impact on SVG.

HTML 5 also introduced a number of features to further separate the structure of a document from its presentation, by making it easier to completely define the web page structure in the markup (or in the DOM, for dynamic pages):

- new elements to describe the structure of complex web sites that consist of more than a single flow of information
- new form input types to capture different types of data
- adoption of the Web Accessibility Initiative's Accessible Rich Internet Applications (ARIA) attributes, which allow authors to identify the structure and function of custom content
- new elements and attributes to better support mixed-language content or content from languages that aren't written as a single string of characters

The examples in this book will touch on some of these features but won't go into too much detail; there are plenty of great resources on using HTML 5 out there. For using SVG, the most important thing to know about HTML 5 is the `<svg>` element.

The HTML 5 `<svg>` element represents an SVG graphic to be included in the document. Unlike other ways of embedding SVG in web pages, the graphic isn't contained in a separate file; instead, the SVG content is included directly within the HTML file, as child content of the `<svg>` element.

It's also possible to include HTML elements as children of SVG content, using the SVG `<foreignObject>` element. It creates a layout box in which an HTML document fragment can be displayed.

The `<foreignObject>` was never supported in Internet Explorer, although it is available in Microsoft Edge. Foreign objects in SVG are somewhat quirky in most web browsers, and are best used only for small amounts of content.

There are some key differences to keep in mind between using SVG code in HTML 5 documents versus using it in stand-alone SVG files.

The most common area of difficulty is XML namespaces. The HTML parser ignores them.

If a web page is sent to the browser as an HTML file, namespace declarations have no effect. Namespace prefixes will be interpretted as part of the element or attribute name they preceed (except for a few attributes like `xlink:href` and `xml:lang`, which are hard-coded into the parser). If the webpage is sent to the browser as an XHTML

file, or if the same markup is used in other XML content, the document object model that results may be different.

This book will try to use the most universally compatible syntax for SVG, and to identify any areas where you're likely to have problems.

To further confuse matters, the DOM *is* sensitive to namespaces. If you are dynamically creating SVG, you need to be aware of XML namespaces regardless of whether or not you're working inside an HTML 5 document.

Another important feature of SVG in HTML 5 is that the `<svg>` element has a dual purpose: it is the parent element of the SVG graphic, but it also describes the box within the web page where that graphic should be inserted. The box can be positioned and styled using CSS, the same as you would an `<img>` or `<object>` referencing an external SVG file.

By including SVG content within your primary HTML file, scripts can manipulate both HTML and SVG content as one cohesive document. CSS styles inherit from HTML parent elements to SVG children. Among other benefits, this means that you can use dynamic CSS pseudoclass selectors—such as `:hover` or `:checked`—on HTML elements to control the appearance of child or sibling SVG content.

Example 2-3 uses HTML 5 form validation and the `:valid` and `:invalid` psuedoclasses to turn the an inline SVG version of the stoplight graphic into a warning light. If any of the user's entries in the form are invalid, the stoplight will display red. If the form is valid and ready to submit, the light will be green. And finally, if the browser doesn't support these pseudoclasses, the light will stay yellow regardless of what the user types.

Example 2-3. Controlling inline SVG with HTML form validation pseudoclasses

```
<!DOCTYPE html>                                                    ❶
<html lang="en">
<head>
  <meta charset="utf-8" >
  <title>Inline SVG within HTML</title>
  <link rel="stylesheet" type="text/css"
```

```
       href="svg-inline-styles.css" >                                    ❷
</head>
<body>
<form id="contactForm" method="post" >                                   ❸
  <h1>How can we contact you?</h1>
  <svg viewBox="20 20 140 320" width="140" height="320"
       preserveAspectRatio="xMinYMin meet" >                             ❹
      <defs>
        <circle id="light" cx="70" r="30" />
      </defs>
      <rect x="20" y="20" width="100" height="280"
            fill="url(gradients.svg#metal) silver"
            stroke="black" stroke-width="3" />                           ❺
      <g stroke="black" stroke-width="2">
        <g class="red light" >
          <use xlink:href="#light" y="80"
            fill="url(gradients.svg#red-light-off) maroon" />
          <use class="lit" xlink:href="#light" y="80" visibility="hidden"
            fill="url(gradients.svg#red-light-on) red" >
            <title>STOP,                                                 ❻
              the form is incomplete or there is an error</title>
          </use>
        </g>
        <g class="yellow light">
          <use xlink:href="#light" y="160"
            fill="url(gradients.svg#yellow-light-off) #705008" />
          <use class="lit" xlink:href="#light" y="160"
            fill="url(gradients.svg#yellow-light-on) yellow" />
        </g>
        <g class="green light">
          <use xlink:href="#light" y="240"
            fill="url(gradients.svg#green-light-off) #002804" />
          <use class="lit" xlink:href="#light" y="240" visibility="hidden"
            fill="url(gradients.svg#green-light-on) lime" >
            <title>GO, your information is ready to submit</title>
          </use>
        </g>
      </g>
  </svg>                                                                 ❼
  <label>
    <input type="text" name="CustomerName" required />
    Full Name <abbr title="Required">*</abbr>
  </label>
  <label>
    <input type="email" name="CustomerEmail" required />
    Email Address <abbr title="Required">*</abbr>
  </label>
  <label>
    <input type="tel" name="CustomerTelephone"
           pattern="\s*\(?\d{3}\)?[-\s]*\d{3}[-\s]*\d{4}\s*"
           title="10-digit number including North American area code.
```

```
(Optional—Leave empty to be contacted by email only.)" />
    Telephone Number
  </label>
  <button type="submit" >Send</button>
</form>
</body>
</html>
```

❶ The `DOCTYPE` declaration tells the browser to use the latest HTML standards when reading the file.

❷ A linked stylesheet contains the CSS for both the HTML form content and the inline SVG.

❸ The body of the web page contains a single `<form>` element. It holds both the form input elements and the SVG that will give feedback about the user's entries.

❹ The root `<svg>` element is included as a direct child of `<form>`, and then contains all the graphical markup (as normal). For a pure HTML document, no namespaces are required (for XHTML, they would be); the HTML 5 parser knows to switch to SVG mode when it reaches an opening `<svg>` tag. The width and height attributes provide default sizes; however, the final size of the SVG will be controlled by CSS. Two new attributes (`viewBox` and `preserveAspectRatio`) control the scaling of the graphic.[1]

❺ The `fill` values of each shape refer to gradients defined in a separate file, *gradients.svg*; the fill value also includes a solid color fallback value.[2]

❻ This version of the stoplight graphic doesn't include the `<text>` labels, but it does include `<title>` values, which will show up as tooltips, and should also be available to screen readers.[3]

Within the `<body>` of the HTML page, a single form element contains a heading and then a modified version of the stoplight SVG

1 See ??? for all about how `viewBox` and `preserveAspectRatio` interact.

2 See ??? for more on the complete fill syntax.

3 See ??? for more on how and why to use `<title>`.

code from Example 1-4. The first change is the addition of `viewBox` and `preserveAspectRatio` attributes to the `<svg>` element. We'll discuss these in detail in ???; for now, just trust that these are the magic that make the SVG drawing scale to fit the dimensions we give it within the HTML page. The particular settings also ensure that the graphic will be drawn flush against the top left edges of the SVG area, making it line up neatly with the heading text.

Inside the SVG, the main change is that we've removed all the gradient definitions and put them in a separate file. As mentioned in Chapter 1, many browsers do not support this; it's used here to shorten the code. In ??? we'll discuss how scripting can be used to avoid this limitation by importing SVG content from another file; here, we use the SVG option of defining fallback colors as an alternative to paint servers. If the gradients cannot be accessed, the shapes will be filled with the solid colors instead.

The graphic doesn't include animation label text. However, `<title>` elements have been added to the illuminated versions of the red and green lights to clarify the meaning of the graphic. Each title will be available as a tooltip and as screen-reader accessible text when the corresponding "lit" shape is displayed. Since the yellow light is only displayed when we *don't* have any information for the user, it hasn't been given a title.

The rest of the code describes the form itself. Various HTML 5 form validation attributes have been used that will trigger invalid states:

- The first two fields are `required`, and so will be invalid when empty.
- The second field is of `type="email"`; browsers that recognize this type will mark it as invalid unless the content meets the standard format of an email address.
- The third field is of `type="tel"`, representing a telephone number. On its own, that doesn't define a specific format, but the field also has a `pattern` attribute that will only accept numbers in the format used in the United States and Canada. The format is expressed as a JavaScript regular expression; it requires groups of 3, 3, and 4 digits, optionally separated by whitespace or common punctuation. An HTML `title` attribute is used to provide a tooltip with guidance to the user.

The `:valid` and `:invalid` pseudoclass selectors are supported on `<form>` elements (as opposed to individual `<input>` elements) in Firefox (versions 13+) and Blink browsers (Chrome and Opera, since early 2015). All other browsers currently display the indeterminate yellow light.

For more universal browser support, you could use a script to listen for changes in focus between input elements, and set regular CSS classes on the `<form>` element based on whether any of the input elements are in the invalid state. The stylesheet would also need to be modified so that these classes also control the SVG styles.

Since this isn't a book about JavaScript, we're not going to write out that script in detail. Once again, nothing about the script would be SVG-specific. The SVG effects are controlled entirely by the (pseudo-)classes on the parent `<form>`.

The styles that control the appearance of both the form and the SVG are in a separate file, linked from within the HTML `<head>`. Example 2-4 presents the CSS code that controls the interaction.

Example 2-4. CSS stylesheet for the code in Example 2-3

```
@charset "UTF-8";

/* Form styles */
form {
    display: block;
    margin: 1em;
    padding: 1em;
    background: lightyellow;
    border: double navy thick;
    border-radius: 1em;
    overflow: auto;
    color: #002;
}
label {
    display: block;
    clear: right;
    padding: 0 0 1.5em;
    font-family: sans-serif;
}
input, button {
    display: block;
```

```
    float: right;
    min-width: 6em;
    max-width: 70%;
    border-width: 3px;
    padding: 0.2em;
}
input:invalid {
    border-color: red;
    box-shadow: none; /* override browser defaults */
}
abbr[title="Required"] {
    color: red;
    font-weight: bold;
    text-decoration: none;
    border: none;
}
form:invalid button[type="submit"] {
    color: gray;
}

/* SVG styles */
form svg {
    float: left;
    width: 6em;
    height: 14em;
    max-width: 25%;
    max-height: 80vh;
    overflow: visible;
    padding: 1em; /* hack for browsers that don't */
    margin: -1em; /* support overflow on SVG */
}
.lit {
    /* Set the lights off to start */
    visibility: hidden;
}
form .yellow .lit {
    /* By default (if HTML 5 form validation is not supported)
       the yellow light will display */
    visibility: visible;
}
form:valid .green .lit {
    /* If the validator thinks all form elements are ok,
       the green light will display */
    visibility: visible;
}
form:invalid .red .lit {
    /* If the validator detects a problem in the form,
       the red light will display */
    visibility: visible;
}
form:valid .yellow .lit, form:invalid .yellow .lit {
```

```
    /* If either validator class is recognized,
       turn off the yellow light */
    visibility: hidden;
}
```

The first batch of style rules define the appearance of the HTML form elements, including using the `:invalid` selector to style individual inputs that cannot be submitted as-is. The `<svg>` element itself is then styled as a floated box with a standard width and height that will shrink on small screens. The `overflow` is set to visible to prevent the strokes of the rectangle from being clipped, now that the rectangle has been moved flush against the edge of the SVG. Older Webkit/Blink browsers only support SVG overflow onto the padding region, so padding is added and then cancelled out with a negative margin.

The remaining rules control the light behaviour. The `.lit` versions of each light are hidden to start, then the yellow light in particular is revealed. If the form matches the `:valid` selector, the bright green light is revealed; if it matches `:invalid`, the bright red light is displayed. Finally, if *either* of those selectors is recognized by the browser, the illuminated version of the yellow light is hidden again.

Figure 2-4 shows the web page in action: when the form is empty (and invalid because of missing required fields), after the required fields are complete (and the form is valid), when there is a formatting error in a field (invalid), and when it is complete (and green light once again). The screenshots are from Firefox, which at the time of writing is the only browser that supports both the new pseudoclasses and SVG gradients from external files.

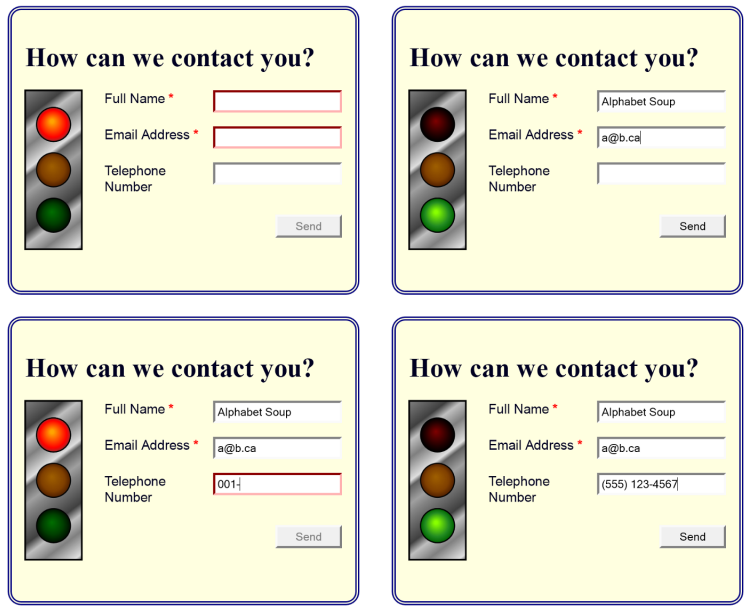


Figure 2-4. A web page using inline SVG to enhance form validation feedback

This demo uses familiar CSS approaches (classes and pseudoclasses) to style SVG in a dynamic way. But this is only the beginning of how SVG and CSS can be integrated.

Using SVG with CSS 3

CSS has also advanced considerably since SVG 1.1 was introduced. First, the core CSS specification was updated to version 2.1; work on that was finalized in 2011. At that time, work was already progressing on a variety of CSS modules, each focusing on specific topics and collectively known as CSS Level 3. There is no single CSS 3 specification, although the W3C CSS working group did publish a "snapshot" defining the current state of stable CSS recommendations.

As of the 2010 CSS snapshot, CSS included

- basic document layout and text formatting features

- an expanded set of selectors to target specific elements based on their position within the document or the state of form input elements
- support for XML namespaces
- expanded options for defining colors, including semi-transparent colors
- the ability to selectively apply certain style rules by querying the size, orientation, color support, or other features of the display media

These features are widely implemented in web browsers, and can usually be used without problem when designing SVG for the web. SVG tools that still rely exclusively on the SVG 1.1 specifications should ignore the new style declarations.

An updated snapshot includes newer specifications, some of which are still not entirely stable. This includes many graphical effects, such as masking, filters, and transformations, that are direct extensions of features from SVG. These standards replace the corresponding SVG definitions (there are no equivalent chapters in SVG 2), allowing the same syntax to be used for all CSS-styled content. Support for these effects in browsers is more erratic; many bugs remain but they are slowly being squashed.

Chapter 3 goes into further detail about how CSS applies to SVG.

Not all aspects of CSS 3 have increased collaboration with SVG. Other new CSS features introduced similar-but-different graphical features that have put CSS in direct competition with SVG for some simple vector graphics. Whenever this book discusses a feature of SVG that has a CSS equivalent, we'll highlight the similarities and differences using "CSS vs. SVG" notes like this:

CSS Versus SVG

Style versus Graphics

In these asides, we'll compare different ways of achieving the same graphical effect, and identify effects that can only be achieved with one language or the other. This should help you, the web designer, decide which tool is best for the job you're trying to do.

CSS 3 also makes manipulating raster graphics easier. Multiple image files can be layered in an element's background, and the images used or their positions can be adjusted according to properties of the display or states of user interaction. In many ways, working with raster images (using CSS or HTML) is more straightforward and predictable than using SVG. However, that predictability is also inflexibility, reflecting the fact that raster images are treated by the browser as a single unit with a predefined presentation.

The relationship between CSS and SVG is so complex, and so important, that we've given it a separate chapter. Chapter 3 will look at the ways CSS can be used to enhance SVG, and SVG can be used to enhance SVG. It will also consider the ways how CSS has started to replace SVG for simple graphics like the stoplight example we've been using so far.

Summary: SVG and the Web

This chapter aimed to provide a big picture of SVG on the web, considering both the role of SVG on the web and the way it can complement (and be complemented by) other web technologies.

The web is founded on the intersection of many different coding languages and standards, each with their own role to play. For the most part, web authors are encouraged to separate their web page code into the document text and structure (HTML), its styles and layout (CSS), and its logic and functionality (JavaScript).

SVG re-defines this division to support documents where layout and graphical appearance is a fundamental part of the structure and meaning. It provides a way to describe an image as a structured document, with distinct elements defined by their geometric presentation and layout, that can be styled with CSS and modified with JavaScript.

The SVG standard has developed in fits and starts. The practical use of SVG on the web is only just starting to achieve its potential. There are still countless quirks and areas of cross-browser incompatibility, which we'll mention whenever possible in the rest of the book. Hopefully, the messy history of SVG and the inter-dependent web standards has reinforced the fact that the web, in general, is far from

a perfect or complete system, and SVG on the web is still relatively new.

The goal of this book is to help you work with SVG on the web, focusing on the way SVG is currently supported in web browsers, rather than the way the language was originally defined. In many cases, this will include warnings about browser incompatibilities and suggestions of work-arounds. However, support for SVG may have changed by the time you read this, so open up your web browser(s) and test out anything you're curious about.

CHAPTER 3

A Sense of Style

Working with CSS

On the web, style means CSS. Cascading Style Sheets are used to indicate how the plain text of HTML should be formatted into the colourful diversity of web sites and applications that you interact with every day.

SVG and CSS have an intertwined relationship. SVG incorporates CSS styling of decorative aspects of the drawing, but uses a basic layout model completely independent of CSS layout. CSS has been expanded to include many graphical effects formerly only available in SVG, until it has become a rudimentary vector graphics language of its own. Other new CSS properties are direct extensions of—and replacements for—corresponding SVG features.

This chapter covers how to use CSS styles to modify your SVG graphics, and how to reference SVG images and elements inside CSS code used to style HTML documents. It also discusses the benefits and limitations of using CSS+HTML to create graphics, including its similarities and differences with SVG, and outlines factors to consider when deciding between the two.

CSS in SVG

CSS is not required for SVG; it is perfectly possible to define a complete SVG graphic using presentation attributes. However, using CSS

to control presentation makes it easier to create a consistent look and feel. It also makes it easier to change the presentation later.

A complete list of SVG style properties, including their default values, is provided in ???. Most will be discussed in context throughout the rest of the book.

Style declarations

There are four different ways to define presentation properties for an SVG element: presentation attributes, inline styles, internal stylesheets (`<style>` blocks), and external stylesheets.

Presentation attributes

Most style properties used in SVG may be specified as an XML attribute. For the most part, the effect is the same as if CSS was used. Properties that are normally inherited will be inherited, and the `inherit` keyword can be used to force inheritance on other properties. Things to note:

- There are no shorthand versions of the presentation attributes (e.g., use `font-size`, `font-family`, and so on, not `font`).
- The XML parser is case-sensitive for property names (must be lowercase) and for keyword values (although SVG 2 will change this, so values are parsed the same way as CSS).
- You cannot include multiple declarations for the same property in order to provide fallbacks for browsers that do not support the latest feature.
- You cannot use the `!important` modifier in presentation attributes.
- Any values specified on the same element using CSS will take priority over a presentation attribute. However, presentation attributes supercede inherited style values, even if the inherited value had been specified with CSS.

Inline styles

All SVG elements can have a `style` attribute. Similar to its HTML equivalent, it accepts a string of CSS `property: value` pairs. Inline

styles declared this way supercede both presentation attributes and values from stylesheets, except for `!important` values.

A `<style>` block

You can include an internal stylesheet within your SVG document using a `<style>` element, similar to the `<style>` element in HTML. The element can be placed anywhere, but is usually at the top of the file or inside a `<defs>` section. In addition, when your SVG code is included inline within another document, such as HTML 5, any stylesheets declared for that document—or declared in other SVG graphics within the document—will affect your graphic.

The `<style>` block can include any valid CSS stylesheet content, but the main content will be CSS style rules consisting of a CSS *selector* followed, within braces (curly brackets), by a list of `property: value` pairs that will apply to elements matching that selector:

```
selector {
  property1: value;
  property2: value; /* comment */
}
```

If you're not familiar with CSS and CSS selectors, you'll want to consult a CSS-specific reference guide, such as Eric Meyer's *CSS Pocket Reference* or the CSS-Tricks online almanac. Most browsers now support CSS Level 3 selectors (and some Level 4 selectors) but older browsers and SVG tools that have not been updated since SVG 1.1 may only support CSS 2 selectors.

Although the SVG `<style>` element works much the same way as its HTML counterpart, HTML introduced additional DOM interfaces that allow you to access and modify the stylesheet using JavaScript, which SVG did not initially match. SVG 2 harmonizes the two, but implementations have not all caught up.

Some other details to consider when using internal stylesheets in SVG files (especially if you're not used to working with CSS in XML):

- The SVG 1.1 specifications did *not* define a default stylesheet type. Although all web browsers will assume `type="text/css"`,

other tools (notably Apache Batik) will ignore the style block if the type isn't declared. SVG 2 makes the *de facto* default official.

- The content of the `<style>` block can be surrounded by character data markers to avoid parsing errors if your comments contain stray `<`, `>`, or `&` characters. The start of the character data region is indicated by `<![CDATA[` and the end by `]]>`.
- CSS has its own way of handling XML namespaces, that is completely distinct from any namespace prefixes declared in the XML markup.

By default in XML, CSS selectors apply to attributes and elements defined in any namespace. To select an element tag or attribute defined in a specific namespace, you declare a namespace prefix with an `@namespace` rule, then use it in your selector, separated from the selector value with a `|` (vertical bar or pipe) character:

```
@namespace svg "http://www.w3.org/2000/svg";
@namespace x "http://www.w3.org/1999/xlink";

a         { /* These rules would apply to any `a` elements. */
          }
svg|a     { /* These rules would apply to SVG links,
               but not HTML links. */
          }
[x|href]  { /* These rules apply to any element with an
               `xlink:href` attribute. Note that you can use
               any prefix you choose, it does not have to match
               the one used in the markup. */
          }
```

You can also declare a default namespace by omitting the prefix in the `@namespace` rule. The specified namespace will apply to any selectors without namespaces:

```
@namespace "http://www.w3.org/2000/svg";

a         { /* These rules now only apply
               to `a` elements within the SVG namespace. */
          }
```

HTML does not fully recognize XML namespaces. If you use unrecognized namespace prefixes in your markup (for example, to add custom data attributes) the HTML parser will include the prefix *and* the `:` (colon) character in the attribute or tag name. To select these in CSS, you could use regular tag or attribute selectors based on the complete name, escaping the `:` character (which has special meaning

in CSS) by preceding it with a \ (backslash). However, it's better to avoid trying to use unrecognized XML prefixes in an HTML document.

For recognized namespace prefixes, such as `xlink`, the namespace *is* properly parsed in HTML 5 documents; declaring specific CSS namespace prefixes works as intended. However, you cannot select these elements by omitting the namespace selector; the browser applies the HTML namespace as the default. To select elements or attributes within any namespace, use a namespace selector with a wildcard (`*`) in place of the namespace prefix:

```
*|a       { /* These rules apply to `a` elements,
              in any namespace, regardless of prefixes. */
           }
[*|href] { /* These rules apply to elements with an `href`
              attribute from any namespace,
              including `xlink`. */
           }
```

External Stylesheets

Style rules may be collected into external *.css* files so that they can be used by multiple documents. There are three ways to include external stylesheets for SVG, all of which allow the stylesheet to be restricted to certain media types:

- Using an include rule at the top of another CSS stylesheet or `<style>` block, like

  ```
  @include "style.css";
  @include url("print.css") print;
  ```

- Using an XML stylesheet processing instruction in the prolog of an SVG or XML file (the "prolog" being any code before the opening `<svg>` or other root tag), like:

  ```
  <?xml-stylesheet href="style.css" type="text/css"?>
  <?xml-stylesheet href="print.css" media="print"
                   type="text/css"?>
  ```

- Using a link element in the `<head>` of an HTML 5 document that includes inline SVG, like:

  ```
  <link href="style.css" rel="stylesheet" type="text/css">
  <link href="print.css" rel="stylesheet"
        media="print" type="text/css">
  ```

SVG 2 allows the HTML `<link>` element for stylesheets in stand-alone SVG files, by expressly declaring the XHTML namespace on the element:

```
<svg xmlns="http://www.w3.org/2000/svg">
  <link xmlns="http://www.w3.org/1999/xhtml"
        rel="stylesheet"
        href="mystyles.css" type="text/css" />
```

Test support before using!

In all these cases, the first stylesheet (*style.css*) would be used for all media, while the second (*print.css*) would only be used when printing out the graphic.

External stylesheets (or any external file resources) are never loaded from SVG files that are used as images in web pages.

The cascade

With all the possible ways of declaring styles, it frequently occurs that multiple values for a given style property could apply to a single element. This is where the *cascade* comes in: the rules that govern the waterfall of different style declarations, one replacing another until a final value is determined.

For the most part, the CSS cascade works the same in SVG as it does in CSS-styled HTML. However, presentation attributes add an extra twist.

The order of precedence of CSS rules is determined by four factors:

1. The *source* of the style declaration: browser defaults, web page author values, or user settings.
2. The *importance* of the rule (either `!important` or not).
3. The *specificity* of the CSS selector.
4. The state of *animations or transitions*; animations override other rules of the same source, importance, and specificity. Transitions override animations.

5. The *order* in which the declarations are encountered by the CSS parser; when all else is equal, later declarations replace previous ones.

Regular style declaration sources are ranked in the following order, from the least to most important:

1. The defaults from the CSS property definitions. For the root element, this is the initial value. For example, SVG `fill` is by default black. For all other elements, it is either `initial` or `inherit`, depending on whether the property is normally inherited.
2. Web browser (or other user agent) default styles for a particular element or pseudoclass, or defaults declared in the specifications. There are very few such styles for SVG: one example is that `overflow` is hidden on some SVG elements.
3. User preferences set in the web browser. For example, many browsers allow users to specify a default font.
4. Styles defined by the web page, in the markup, in `<style>` blocks, or from external stylesheets.

For `!important` rules, the ranking of sources is reversed: important user preferences override all author styles, including important ones. These are usually related to accessibility issues, such as setting a minimum font size or requiring high-contrast color combinations. Important browser styles are even stronger, and usually relate to fundamental features of the language, such as the fact that `linearGradient` elements do not display, regardless of the `display` value an author might set.

Because SVG text is often artistic, browsers may or may not apply user's accessibility preferences to modify the author's styles.

When multiple style rules declared by the same source could apply to a given element, they are ranked according to the specificity of the CSS selector used. There are three components to selector specificity, in order from least to most important:

1. Element tag names and pseudo-element selectors.
2. Classes, pseudo-classes, and attribute-based selectors.
3. ID values.

A given selector may have many components, so the total number of components of each category is counted. If two selectors are tied for the number of `id` references (possibly none), then the number of class and class-like components is counted. If still tied, the number of element components is used. A universal selector, `*`, has zero specificity when comparing stylesheet rules, but still trumps over presentation attributes, inherited values, and browser defaults.

Presentation attributes are treated as if they had selectors with a specificity slightly less than zero. The `style` attribute acts like the most specific selector of all.

After all styles from all sources have been cascaded, if the final value for an element is still `inherit` or `initial`, the corresponding value is substituted in.

The cascading and specificity rules can be used to create a stylesheet that completely overrules presentation attributes in the code, for example to create a separate set of styles for black-and-white printing. Example 3-1 presents such a stylesheet for the stoplight with gradients code from Example 1-5 in Chapter 1. The only modification to the SVG code (which had used presentation attributes) is a class name added to the `<g>` element that contains the separate lights —and a link to include the print stylesheet. The result is shown in Figure 3-1.

Example 3-1. Stylesheet for Monochrome Printing of an SVG

```
* {                                  ❶
    fill: inherit;                  ❷
    stroke: inherit;
    stroke-width: inherit;
}
g.lights {                          ❸
    fill: lightgray;
    stroke: dimgray;
    stroke-width: 1;
```

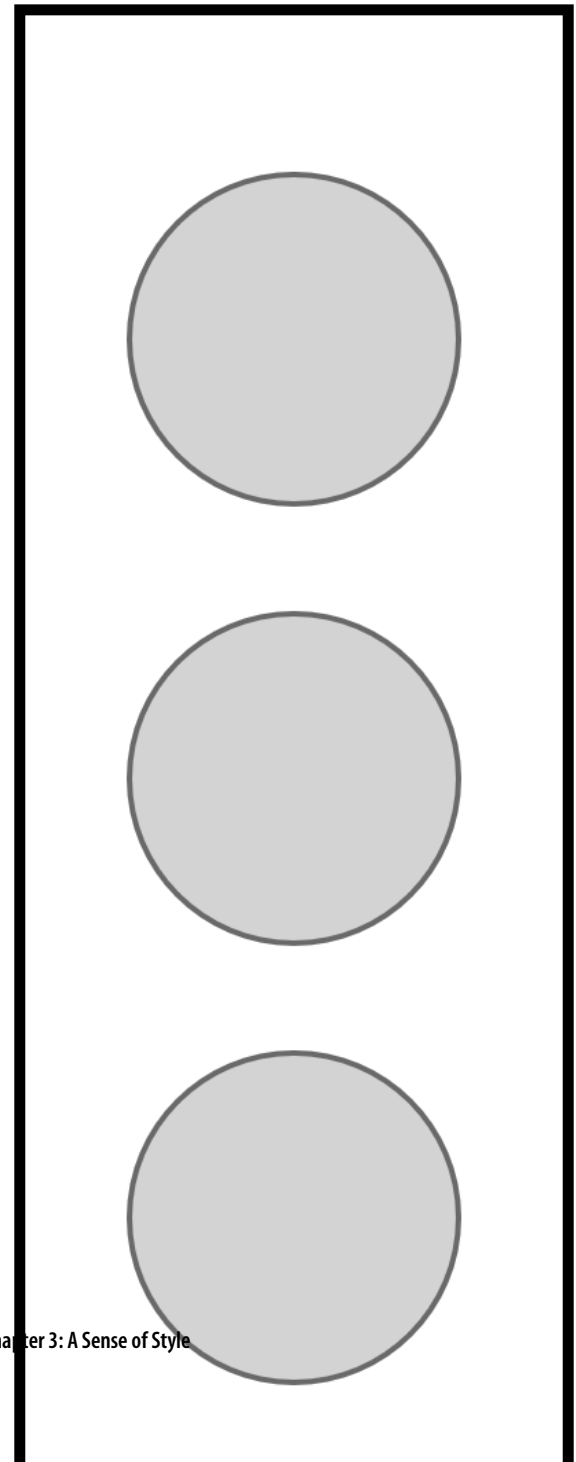
```
}
rect {                          ❹
    fill: none;
    stroke: black;
    stroke-width: 2;
}
```

❶ The universal selector (`*`) is used to reset the styles declared on all elements using presentation attributes.

❷ The `inherit` keyword is used, instead of setting a value directly on every element, so that style inheritance will work normally.

❸ All the lights are given the same appearance, by setting the styles once on the `<g>` element (which has been given a class name to aid CSS styling).

❹ The `<rect>` is styled separately. Note that, because we reset the `stroke` and `stroke-width` presentation attributes for *all* elements, the styles have to be set in the CSS even when, like the black stroke, they have not been changed from the original.



If there are multiple valid stylesheet rules with the same specificity, the final value is used. To determine which value comes last, different stylesheets and blocks are concatenated together in the order in which they are included in the document. This allows you to use an external stylesheet for many documents and then modify it for a particular document using a `<style>` block. It also allows you to specify fallback styles for new features that may not be universally supported.

Conditional styles

Declaring styles in CSS stylesheets is often simply a convenience compared to using presentation attributes, allowing you to apply the same style rules to many elements (or many files) without repeating your code. However, using CSS can create much greater flexibility in your graphic, by using fallback values, media queries, and interactive pseudoclasses to create styles that adapt to the context.

The most basic, and universally supported, conditional CSS is the fallback value. CSS error handling rules require browsers to skip any invalid style declarations, and continue reading the rest of the file. If you use a new feature the browser doesn't support, it will ignore the declaration.

In combination with the cascade rules, this means that you can provide fallback values for new property features simply by declaring—earlier in the stylesheet—a more widely supported value for the same property. Browsers will apply the *last* value that they recognize and support. For example, the following code would provide a fallback for a semi-transparent `rgba()` color value (which was introduced in the CSS Level 3 Color specification):

```
.stained-glass {
  fill: #FF8888; /* solid pink */
  fill: rgba(100%, 0, 0, 0.5); /* transparent red */
}
```

Fallback styles allow you to use new features while providing alternatives for tools that only support the SVG 1.1 specifications. There are currently only a few such features relevant to styling SVG, such as CSS 3 colors and units, and new filter and masking options. However the number of inconsistently-supported style values will likely increase as new CSS 3 and SVG 2 specifications are rolled out.

One limitation of CSS error-handling fallbacks is that they only apply to a single property. Sometimes you need to coordinate multiple property values to provide fallback support for a complex feature. The new `@supports` rule would allow this type of coordination. The basic structure looks like the following:

```
.stained-glass {
  fill: #FF0000; /* solid red */
  fill-opacity: 0.5;
}
@supports ( fill: rgba(100%, 0, 0, 0.5) ) {
    .stained-glass {
      fill: rgba(100%, 0, 0, 0.5); /* transparent red */
      fill-opacity: 1; /* reset */
    }
}
```

With this code, the SVG `fill-opacity` property is used as an alternative to CSS 3 transparent colors. Again, thanks to CSS error-handling rules, browsers that don't recognize the `@supports` rule will skip that entire block.

CSS `@supports` is supported in all the latest browsers, but older browsers (such as Internet Explorer) will skip over the entire block; make sure you still have decent fallbacks if "nothing" is supported by the `@supports` test!

Be aware that `@supports` only tests whether the CSS parser recognizes a particular property / value combination. It can't test whether that property will be applied when styling a particular element. This is particularly frustrating when working with SVG in web browsers. Certain properties (such as `filter` or `mask`) may be recognized, but only applied on SVG elements, despite applying to all elements in the latest specs. Other properties (such as `text-shadow` or `z-index`) may be recognized and applied for CSS box model content, but not for SVG.

The `@supports` rule is a type of *conditional* CSS rule block. A more established conditional CSS rule is the `@media` rule, which applies certain styles according to the type of output medium used to display the document. We already discussed how it is possible to use the `media` attribute to limit the use of external stylesheets; `@media`

rules allow those same conditions to be included within a single stylesheet or `<style>` block:

```
@media print {
  /* These styles only apply when the document is printed */
}
```

Originally, CSS used a fixed number of media type descriptions, such as `screen`, `print`, `tv`, and `handheld`. However, this proved limiting. Handheld devices today are quite different from what they were 15 years ago, and screens come in all shapes and sizes. The only media type distinction that is still relevant is `screen` versus `print`. For all other distinctions, use feature-based media queries, which directly test whether the output device is a certain size, or a certain resolution, or able to display full color images (among other features).

For general information on media query syntax and options, consult a CSS reference. For SVG, there are a couple complications to keep in mind.

The first thing to realize is that the media query is evaluated before any of the SVG code; in other words, before any scaling effects within the SVG change the definition of what a `px` or a `cm` is.

The second thing to be aware of, particularly if switching between separate SVG files and inline SVG code, is that the media being tested is the output area for the *document* containing the SVG code. If the SVG is inline within HTML 5, that means the entire frame containing the web page. However, if the SVG is embedded as an `<object>` or `<img>`, it means the specific frame area created to draw the graphic. Example 3-2 demonstrates the difference. The same CSS file is used for both the inline SVG code and the embedded SVG object; it uses media queries to test the size of the document region, and reduces the thick stroke width when the document is small.

Example 3-2. Interpretation of Media Queries in Inline versus Embedded SVG

CSS STYLES: *MEDIAQUERY.CSS*

```
rect {
    fill: lightGreen;
    stroke: darkGreen;
    stroke-width: 20px;
```

```
}
svg.inline, object {
    width: 100px; /* fallback if viewport units not supported */
    height: 150px;
    width: 40vw;
    height: 80vh;
    border: solid thin navy;
}
figure {
    display: inline-block;
    margin: 2%;
    padding: 0;
    font-family: sans-serif;
}

@media (max-width: 200px), (max-height: 200px) {
    rect {
        stroke-width: 5px;
    }
}
```

Embedded SVG file: *mediaquery-embedded.svg*

```
<?xml-stylesheet href="mediaquery.css" ?>
<svg xmlns="http://www.w3.org/2000/svg" xml:lang="en" >
    <title>Stretchable Rectangle</title>
    <rect x="10%" y="10%" height="80%" width="80%" />
</svg>
```

HTML markup:

```
<!DOCTYPE html>
<html>
<head>
    <title>Using Media Queries with Embedded and Inline SVG</title>
    <link rel="stylesheet" href="mediaquery.css" />
</head>
<body>
    <figure>
        <figcaption>Inline SVG</figcaption>
        <svg class="inline" >
            <title>Stretchable Rectangle</title>
            <rect x="10%" y="10%" height="80%" width="80%" />
        </svg>
    </figure>
    <figure>
        <figcaption>Embedded Object</figcaption>
        <object type="image/svg+xml" data="mediaquery-embedded.svg"></object>
    </figure>
</body>
</html>
```

A screenshot of the result on a small screen is shown in Figure 3-2. At this size, the height and width for the embedded object have triggered the media query, but the overall document dimensions, used for the styles on the inline SVG, have not.

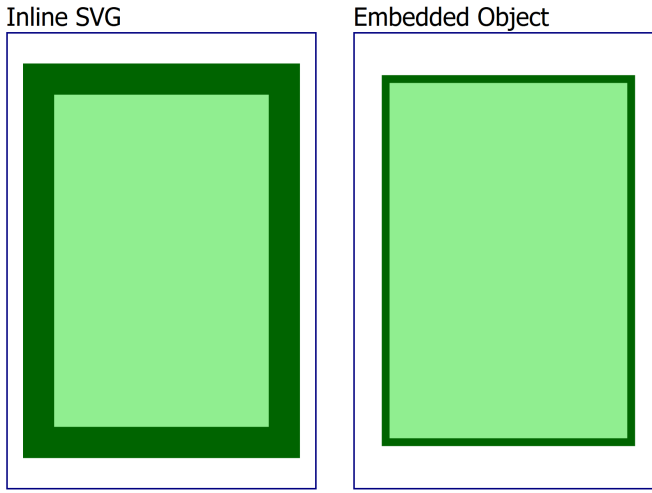


Figure 3-2. Media Queries in Inline versus Embedded SVG, on a 4-inch wide display

Although this example was simple, media queries add considerable flexibility to your SVG code. In ???, we'll discuss using media queries to adjust text size when the scale of an image changes.

It is clear that a thorough understanding of CSS can help you make the most of your SVG graphics. However, that is not the end of the relationship; it also works the other way.

SVG in CSS

There are two distinct ways in which you can reference SVG files from within a CSS file: using the entire SVG file as an image, or using specific SVG effects elements. Complete SVG images can be used like other image types in CSS-styled HTML or XML documents. There are not currently any CSS properties that use images to style SVG content (although there are proposals to change this).

SVG element references were initially a feature unique to SVG styling properties; however, many of these properties are now being expanded to other types of CSS-styled content.

Using SVG images within CSS

The ability to reference image files from CSS has existed from the earliest versions of the language. Any HTML element that participates in the document layout flow—i.e., anything that takes up space on a web page—has a background. The ability to modify this background was one of the first CSS capabilities implemented in contemporary browsers, and as such is quite robust, fully supported by all popular browsers still in common use.

The `background-image` property takes a URL, either local (such as */images/myImage.jpg*) if on the same server as the web page, or global (*http://www.example.com/images/myImage.jpg*) if not. Like most CSS properties that require a URL, the path should be contained within the url() function:

```
background-image: url('/images/myImage.jpg');
```

The quotes around the argument in the `url()` function are recommended, but can generally be omitted so long as there is no whitespace in the URL.

The HTML 5 specification indicates that SVG should be considered a valid format for use with background-image. This means that if you have an SVG file, such as *myImage.svg*, you can use it in exactly the same manner as you could use a JPEG, GIF, or PNG file:

```
background-image: url('/images/myImage.svg');
```

If you want to provide fallback images for older browsers, you have a few options:

- Use a JavaScript tool, such as Modernizr, to test whether SVG images are supported and change the classes (and therefore style rules) on your elements accordingly.
- Use a server script to identify the old browsers and edit your web page to use different style rules.
- Use Internet Explorer conditional comments to include a stylesheet that over-rides your main style rules (this doesn't help with older mobile browsers that don't support SVG).

- Use other modern CSS syntax not supported by the older browsers, such as layered background images or CSS gradients, to make the browser ignore the background image declaration that includes SVG, and apply fallback declaration instead.

The layered-image fallback approach looks like the following:

```
background-image: url('/images/myImage.jpg'); /*fallback*/
background-image: url('/images/myImage.svg'), none;
```

This works on Internet Explorer 8, currently the most commonly used browser that doesn't support SVG, but fails on some early Android browsers that support layered backgrounds but not SVG. A variation replaces the `none` layer with a transparent CSS gradient (see ??? for syntax), reducing the number of older mobile browsers that will try to download the SVG.

Although backgrounds are the most common use for images in CSS, there are currently two other properties which accept image values: `list-style-image` and `border-image`. The first is used to specify a custom graphic to replace the bullet or number for a list element (technically, any element with CSS display type `list-item`). The second is used to generate decorative frames for elements.

Example 3-3 applies all three properties to an HTML list, using card suit shapes that we'll learn how to draw in ???. The (somewhat intense-looking) webpage that results is shown in Figure 3-3.

Example 3-3. Using CSS properties that accept an SVG image value

HTML MARKUP:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" >
    <title>Using SVG in CSS for an HTML page</title>
    <link rel="stylesheet" href="svg-in-css.css" />
</head>
<body>
    <ul>
        <li>First point</li>
        <li>Second point</li>
        <li>Third</li>
        <li>Fourth</li>
        <li>And another</li>
        <li>One more</li>
```

```
        <li>In conclusion</li>
        <li>This is the last point</li>
    </ul>
</body>
</html>
```

CSS STYLES: *SVG-IN-CSS.CSS*

```
body {
    margin: 0;
    padding: 2em;
    font-family: sans-serif;

    background-color: red;                                          ❶

    background-image: url('../ch06-path-files/spade.svg'),
                      url('../ch06-path-files/heart.svg'),
                      url('../ch06-path-files/club.svg');           ❷
    background-size: 40px 40px;
    background-position: 0 0, 20px 0, 20px 20px;                    ❸
}
ul {
    background-color: white;
    background-color: rgba(100%, 100%, 100%, 0.8);
    max-width: 70%;
    margin: auto;
    padding: 1em;
    padding-left: calc(1em + 10%);

    border: solid #999 5%;                                          ❹
    border-radius: 10%;

    border-image-source: url(svg-in-css-border-gradient.svg);       ❺
    border-image-slice:  5% 5%;
    border-image-width:  4.7%; /* 5% / 105% */
    border-image-repeat: stretch;
}
li {
    line-height: 2em;
}
ul li:nth-of-type(4n+1) {                                           ❻
    list-style-image: url('../ch06-path-files/spade.svg');
}
ul li:nth-of-type(4n+2) {
    list-style-image: url('../ch06-path-files/heart.svg');
}
ul li:nth-of-type(4n+3) {
    list-style-image: url('../ch06-path-files/club.svg');
}
ul li:nth-of-type(4n+4) {
```

```
    list-style-image: url('../ch06-path-files/diamond.svg');
}
```

❶ The solid red background color will be visible on any parts of the web page not covered by the background images; it will also provide a fallback if SVG images are not supported.

❷ For browsers that support SVG and the CSS 3 Backgrounds and Borders specification, a complex pattern consisting of three overlapping images will be used as the background. The backgrounds will be layered top to bottom in the same order they are given in the CSS.

❸ A list of values is given for `background-position`, setting the initial offset for each corresponding graphic in the list of background images. Each shape will have the 40px square size set by `background-size`. The positions are the initial offset, but each image will repeat in a tiled pattern (set the `background-repeat` property for a different behavior).

❹ The list will be surronded by a decorative border image; however, a solid gray border is defined as a fallback. The `border-radius` curvature is used to ensure that the padding area will not extend beyond the curved corners of the border image.

❺ The border image consists of a rounded rectangle with a repeating diagonal gradient. The other properties control how the image is divided into edges and corners to fill the border region. CSS border images are a complex topic, with many options, which do not take advantage of any SVG features. You may find it easier to use layered background images to achieve the same effect.

❻ To create a rotating series of custom bullets, the `nth-of-type` CSS pseudoclass selector is used to assign the different list images. The selector `li:nth-of-type(4n+1)` will apply to every `<li>` element that is one more than a multiple of four (when counting all list-item elements that are children of the same parent). In other words, the first, fifth, and ninth item and so on. Unlike JavaScript, CSS does not count the first element in a set as index 0.

Figure 3-3. A web page using SVG graphics as CSS backgrounds, bullets, and borders

The stylesheet presented in Example 3-3 references five separate SVG image files. Each one is only a few hundred bytes in size before compression, but requesting each file from the web server may slow display of the web page.

The HTTP/2 protocol, now available in the latest browsers and many web servers, reduces the time spent by the browser requesting each new file. With HTTP/2, you probably wouldn't worry about having five different image files. But you might worry about 100.

This is especially true because the server's ability to compress the file size depends on how much repetition there is in a single file: repetition that is divided across many different files cannot be compressed away.

There are various options to reduce the number of file requests when you have a large number of small SVG graphics:

- Construct a single *sprite* image, with the separate icons arranged in a row or grid, to use as a `background-image` file for multiple

elements. Then use the `background-size` and `background-position` properties to position the sprite file in such a way that the correct icon is visible.

This technique has been used for years with raster image sprites, there is nothing specific to SVG about it. However, it only works for background images—there are no similar size and position options to reposition a list image file.

- Create a sprite image as above, but add `<view>` elements (which we'll discuss in ???) which describe the target region for each icon. Then specify the id of the relevant view as the target fragment (after the # mark) in the URI.

  This SVG-specific sprite feature could be used for any CSS (or HTML) image, since all the information about position and size is contained in the SVG code and the URL fragment. However, at the time of writing it is not supported for CSS images in any Blink or Webkit browsers, which ignore URL fragments when rendering CSS images.

- Create an SVG *stack* file, with your different icons overlapping in the same position. Then use the CSS `:target` pseudoclass to only display each graphic if it is referenced in the URL target fragment.

  This option has the same browser compatibility limitation as for SVG view sprites; it also makes working with your icon files more difficult, since no icons will display if you open up the SVG file without a target fragment.

- Convert each image reference to a data URI, which allows you to pass an entire file's contents in the form of a URI value.

To use data URIs, you specify the `data:` file protocol, the file type, optional encoding information, and then the file contents as a URI-safe string. Many browsers allow you to pass an SVG file as plain text markup, with only the `%`, `#`, and `?` characters encoded (since these have special meaning in URIs). However, Internet Explorer will not read the data URI unless it is completely URL-encoded (including whitespace and `=`, `<`, and `>` markup characters). You can use the JavaScript `encodeURIcomponent(`*`string`*`)` method to encode your plain text markup. The result would be entered in the following code to create a data URI:

```
url("data:image/svg+xml,URI-encoded ASCII text file");
url("data:image/svg+xml;charset=utf-8,URI-encoded
            Unicode file");
```

Alternatively, you can use the Base 64 encoding algorithm, which converts the raw binary data of a file to a string using 64 URL-safe characters; the algorithm can also encode raster image files (which don't have a plain text representation). The JavaScript function `btoa(`*`data`*`)` function converts file data to base-64 encoding. The result is embedded as follows:

```
url("-encoded
            ASCII text file");
url("data:image/svg+xml;charset=utf-8,base64-encoded
            Unicode file");
```

Be sure to use a complete, valid SVG file to create your data URI, including XML namespaces. However, you can *minify* the file as much as possible before encoding, in particular to remove extra whitespace.

There are no extra complications when combining large number of SVG graphical effects into a single file; in these case, the URL references always target a specific element.

Using SVG effects within CSS

We have already seen, in Chapter 1, how an SVG presentation attribute can reference another SVG element using the `url()` notation. In that case, it was a fill property referencing a gradient element.

Other SVG style properties follow the same syntax: you define the details of the graphical effect you want to apply in the SVG markup, and then apply that effect to another element using a `url()` reference. In the case of masks, clipping paths, and filters (which we'll discuss in ??? and ???) these graphical effects can now also be applied to non-SVG content in the latest web browsers.

The URL value may be relative or absolute, but it must always contain a # targetting the reference to a specific element id. Because the graphical effects are considered interactive content, most web browsers will apply cross-origin file security restrictions. If the file with the SVG content isn't on the same web domain as the CSS file that references it, the web server must provide HTTP headers that specifically allow its use by the referencing web domain.

If the URL is either a local target fragment like `url(#filter)` or a relative file path like `url("../assets/filters.svg#blur")`, the URL will be resolved relative to the file that contains the CSS rule. This means that local target fragments can only be used for `<style>` blocks, inline styles, and presentation attributes—never an external stylesheet, which cannot contain valid SVG elements. The location of relative URLs, *including* local target fragments, will also be affected by the HTML `<base>` tag or `xml:base` attributes, which instruct the browser to treat all relative URLs as being relative to a different web address.

SVG 2 and the latests CSS specs propose special rules for URLs that *only* have a target fragment (i.e., they start with a `#` character), so that they would not be affected by `<base>` change, and could be used in external stylesheets. However, browser support isn't consistent yet, and some of the specification details may still change.

CSS versus SVG

This chapter has so far emphasized the ways in which CSS and SVG are interdependent and complementary. However, there are also many ways in which the two are contradictory and competitive. As CSS has expanded to include more graphical options, it has included many features that were previously the exclusive domain of SVG, such as gradients, complex shapes, and animation. Although some of the new CSS graphical effect have been coordinated with work on SVG, others have implemented completely new rules and syntax. As a result, switching between CSS graphics and SVG can be confusing.

Styling Documents versus Drawing Graphics

The tension between CSS and SVG is driven by their differing goals of formatting documents versus rendering graphics. Both are involved in handling layout, both control the incorporation of images, colors and patterns, and both determine how text gets rendered. It's easy to get lost in the overlap; some CSS properties work in SVG just the same as in HTML, others are completely different.

It helps to focus on the different purposes of the two languages. CSS was designed to describe the presentation of text documents. It

assumes that most of the elements in the document contain text. CSS rules define the regions of the web page in which the text should be arranged (layout boxes), the decoration of those boxes, and the styling of the text itself. For the most part, the text is treated as a continuous stream that can be wrapped from one line to another to fit within the layout boxes; when you change the available space for each line, the layout will be re-arranged, with text wrapping at different points, boxes expanding to fit more or fewer lines of text, and the overall page layout shifting to accomodate them.

In contrast, SVG defines a two-dimensional graphic. There is no single flow of content that can be wrapped to a new line if there isn't enough space on this one. The entire SVG expands or contracts together, preserving the relative positions of all the elements in both horizontal and vertical directions.

The shape and position of SVG elements are a fundamental part of their meaning and purpose. In contrast, the fundamental meaning of HTML elements is contained in their text content and the semantic meaning associated with the HTML tags; the shape and position of the CSS layout boxes is purely decoration.

When you add a border to a CSS layout box, it takes up extra space around the outside of the box, and the layout adjusts to accomodate it. In contrast, when you add a stroke to an SVG shape, it is positioned exactly centered over the geometric edge of the shape. If that stroke overlaps something else, it's up to you to decide whether to move or resize the shapes to accomodate it; the browser makes no assumptions about why you're drawing graphics in the particular places you specify.

Many other syntax differences come from this difference between styling independent, flexible boxes versus drawing graphics that are explicitly positioned by *x*- and *y*-coordinates. Even for the CSS features that have been adapted from SVG, such as transformations (see ???) or masks (???), new rules were required to apply the effects to elements that aren't part of a fixed coordinate system.

Nonetheless, although CSS layout and style properties were created to format text documents, they can just as easily be applied to empty elements in order to construct purely graphical content. It is with this usage that CSS becomes a direct competitor to SVG.

CSS as a Vector Graphics Language

As described in Chapter 1, vector graphics describe where and how a computer should draw an image, rather than describing the pixelated result. In this way, the combination of an HTML document plus CSS stylesheet can be seen as a vector language; together, they describe how the browser should display the web page.

Most web pages do not use the precise coordinate-system layout usually associated with vector graphics. However, CSS absolute positioning can be used to provide coordinate-like positioning, placing elements at a certain point on the page regardless of the flowing layout of the text. CSS also allows you to set width and height of elements exactly, regardless of the width available or the height required to display their text content.

With these properties—and the many styles available to decorate CSS boxes with borders, backgrounds, and more—CSS can turn a nested series of HTML elements into a vector graphic. Example 3-4 uses CSS and HTML to recreate the original, primary-color stoplight from Chapter 1. Figure 3-4 shows the result, which is very close to Figure 1-1.

Example 3-4. A simple stoplight drawn with CSS and HTML

HTML MARKUP:

```
<!DOCTYPE html>
<html>
<head>
    <title>Stoplight Drawn Using CSS Styles</title>
    <style>
        /* Style rules go here (or as an external stylesheet) */
    </style>
</head>
<body>
    <figure aria-label="A stoplight">                    ❶
        <div class="stoplight-frame" >                   ❷
            <div class="stoplight-light red" ></div>     ❸
            <div class="stoplight-light yellow" ></div>
            <div class="stoplight-light green" ></div>   ❹
        </div>
    </figure>
</body>
</html>
```

❶ An HTML 5 `<figure>` element is used to group the elements that will be part of the graphic. An `aria-label` attribute adds a text description for accessibility purposes.

❷ The frame and the lights are each `<div>` elements; by default, these will be displayed as independent boxes, but they have no other meaning or default styles. A class attribute is used so that the custom styles can be applied in the stylesheet.

❸ The three elements that will represent the lights are contained inside the element that will represent the frame; unlike SVG shapes, HTML `<div>` elements can be nested, which allows you to position smaller elements relative to the boundaries of larger components.

❹ Each light has been given two class names: one (`stoplight-light`) will be used to assign the common features (size and shape); the other (e.g., `green`) will be used to assign the unique features (color and position).

CSS STYLES:

```
figure {                              ❶
    margin: 0;
    padding: 0;
}
.stoplight-frame {                    ❷
    margin: 20px;
    width: 100px;
    height: 280px;
    background-color: blue;
    border: solid black 3px;
    position: relative;               ❸
}
.stoplight-light {                    ❹
    width: 60px;
    height: 60px;
    border-radius: 30px;
    border: solid black 2px;
    position: absolute;               ❺
    left: 20px;
}
.stoplight-light.red {                ❻
    background-color: red;
    top: 30px;
}
```

```
.stoplight-light.yellow {
    background-color: yellow;
    top: 110px;
}
.stoplight-light.green {
    background-color: #40CC40;
    top: 190px;
}
```

❶ Most browsers inset `<figure>` elements compared to the rest of the text; here, the margin and padding are reset so that positions of the graphic components will be relative to the browser window.

❷ The element with class `stoplight-frame` is offset from the edge of the window using margin spacing, then is given a fixed width and height. It is filled in using `background-color` and given a stroke-like effect with `border`.

❸ The stoplight frame element is also given the `position: relative` property. This does not affect the display of this component, but it defines the component as the reference coordinate system for its absolutely-positioned child elements.

❹ All three of the elements representing the lights will share the `.stoplight-light` styles. The `width` and `height` make the element square, then `border-radius` rounds it into a circle and `border` adds a stroke effect.

❺ After declaring the element to be absolutely positioned, the `left` property sets the horizontal position of each light, as an offset from the left edge of the frame element.

❻ Each individual light has its color set according to its class. The vertical position is set with the `top` property, again as an offset from the frame element.

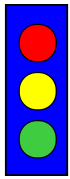


Figure 3-4. The CSS vector graphic stoplight

The example may be simple, but it demonstrates some common features in CSS vector graphics:

- Although CSS elements are by default rectangular, the `border-radius` feature can be used to create circles or ellipses.
- Elements can be positioned using margin and padding, but it is usually more reliable to position them absolutely.
- You can use one element as the reference frame for absolutely positioning other elements by nesting them in the HTML.

In ???, we will expand upon this simple stoplight example to recreate the gradient effects from Figure 1-3 using CSS gradients.

Which to Choose?

If you can create vector graphics with CSS, why bother learning SVG? It really depends on what type of graphics you're trying to create.

For advanced graphics, SVG has the undoubted advantage. CSS can create simple rectangles and circles, but not complex shapes. There is also much better browser support for graphical effects in SVG, and more flexible options for decorative text.

For graphical decorations on a text document, however, CSS styling may provide a simple solution. For a simple background gradient, styling with CSS is easier than creating a separate SVG file, encoding it, and embedding it as a data URI in your CSS file. By taking advantage of the `::before` and `::after` pseudoelements available on most elements, you can create moderately complex decorations such as turned-down corners, menu icons, or stylistic dividing rules.

The `::before` and `::after` pseudoelements do not apply to "replaced content" such as images, form input elements, or SVG content. They *do* apply to other void (always empty) HTML elements such as `<hr/>` (horizontal rule) and `<br/>` (line break), which are rendered using the CSS layout model.

When the CSS graphic requires more than the three layout boxes you can create from a single element, simplicity is compromised. Creating a scaffolding of HTML elements to represent each part of the graphic, as we did in Example 3-4, divides your graphical code between the CSS and the markup. Although the same could be said about inline SVG graphics with external stylesheets, it is generally easier to distinguish SVG graphical markup from the rest of the HTML content of the web page. SVG's `<use>` element, in particular, makes it easier to organize your graphical markup into a single section of your HTML file.

Compatibility adds another layer of complexity. For the most part, browsers either support SVG or they don't; certain effects may not always render the same, but the geometric structure will be consistent. In contrast, when building CSS vectors, the geometric appearance often depends on relatively recent features of the language. Without support for `border-radius`, the stoplight would have been barely recognizable.

This book focuses on SVG, and so it will for the most part emphasize the SVG way of creating vector graphics. However, this is also intended to be a practical reference for web designers, so the question of CSS versus SVG will be re-visited regularly throughout the rest of the book. Whenever an SVG feature is introduced that has a CSS counterpart, the two will be compared and contrasted to highlight the key differences in how they work.

Summary: Working with CSS

CSS and SVG have an interdependent relationship, which has both been enhanced and complicated by the development of Level 3 CSS specifications.

CSS 3 is big, consisting of more than two dozen different documents representing new functionality beyond what is supported in CSS 2.1.

There are specifications covering animations, 2D and 3D transformations, transitions, text to speech, print layout, advanced selectors, and more. Throughout this book, where appropriate, each chapter will cover the CSS capabilities in comparison with SVG features.

It's worth noting that the CSS specification are a perpetual work in progress. One of the key roles of this chapter was to point out features that were available and relevant to SVG. If you are working with advanced CSS features, it is always worth spending some time looking at the current state of work on CSS specifications at the W3C and experimenting to see what is and is not implemented in your target browsers.

When using CSS to style SVG content, new CSS features such as media queries can increase the functionality and flexibility of your graphics. Because most browsers update their CSS and SVG implementations independently, you can use these new CSS features in your SVG files in any web browser that supports them, even though they did not exist at the time the SVG 1.1 specifications were finalized. The future-focused CSS error-handling rules allow you to define limited fallback options for software that has not implemented the latest features. The `@supports` rule allows more nuanced control.

SVG has also become an important part of CSS styling for text documents. This chapter has discussed the use of complete SVG images in CSS; other chapters will explore the SVG graphical effects which can now be used in the latest browsers to manipulate the appearance of HTML content.

At the same time, CSS 3 has developed alternatives to many SVG features, allowing it to be used to directly create vector graphics from empty elements in your HTML code. This chapter has given a hint of the possibilities; by the end of the book, you should have a clearer understanding of what is possible with CSS and HTML alone, and what is made much easier with SVG.

CHAPTER 4

Tools of the Trade

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 into a text editor or IDE 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 drawings and original art start their life inside some kind of graphical software, created by an artist or designer working with shapes and colors rather than XML tags and attributes. Most SVG data visualizations are created using JavaScript, and visualization libraries offer different degrees of abstaction between the author and the SVG code. In most projects, SVG icons are imported from existing icon sets, with the SVG files manipulated entirely by project build scripts.

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. But this 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 process files so they are ready to deploy on your web server and display on your pages. The tools discussed in this chapter include graphical editors that emphasize visual components rather than code, editors that provide hints and immediate feedback, libraries that streamline the creation of dynamic graphics with JavaScript, and rendering programs that display the SVG or convert it into other image formats. In addition, we introduce the vast panalopy of free and licensable SVG content that can help you quickly enhance your web development process and designs, even if your personal artistic skills don't extend beyond stick figures.

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 file can usually be opened and modified with other SVG tools, so you're not always locked into the workflow of a particular product or vendor.

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.

The simplest method is to start searching for your term of interest plus "SVG", which will yeild a broad result in most search engines. For more precision, you will want to refine your search: Google Images (*http://images.google.com*) is a good start, although you can refine your search by using the "Search tools" option, and choosing "Line Art" or "Clip Art" under the "Type" menu option. Alternatively, typing "*subject* filetype:SVG" directly into Google will also work.

Prior to using clip art from a vendor or website, you should ascertain what license it is provided under. Early in the history of SVG, most graphics were released under a Creative Commons license:footnote[*https://creativecommons.org/*], but increasingly, high-quality artwork is produced by professional artists working

within the strictures of traditional copyright. Although there are plenty of free-to-use graphics available (some with non-commercial restrictions or attribution requirements), others are offered under paid licence systems similar to those used for stock photos or web fonts.

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

For accessing graphics created by others, remember that creative works are by default "all rights reserved"; *the absence of a copyright declaration does not mean a work is public domain*. Don't use someone else's work unless you are sure that the original creator has offered a licence 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 those provided by 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 a vector graphic as a source file, stock art companies can generate raster images (PNG, JPEG, and GIF) of any size on demand. For a web designer interested in purchasing raster graphics, however, it often makes more sense to licence a single SVG and convert it into the required raster format at the needed resolutions yourself, rather than purchasing raster graphics at different scales.

The following list of sites 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 the format typically doesn't have the same type of vendor encumbrances or royalty restrictions as other proprietary 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 4-1 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.

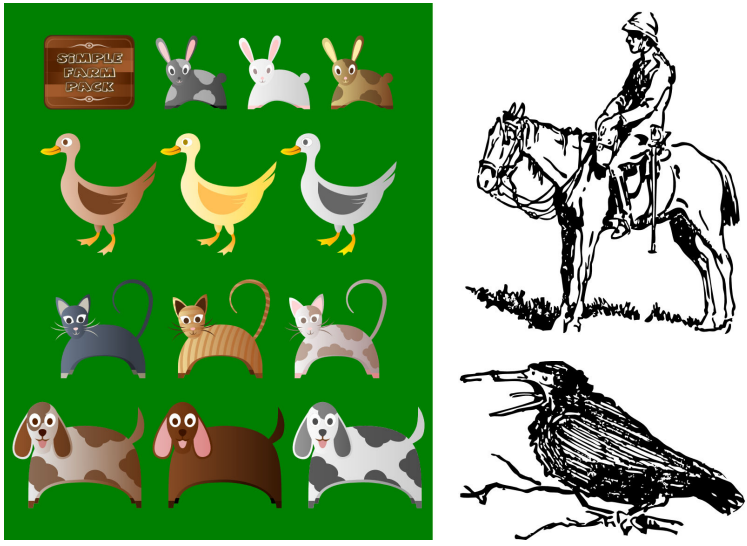


Figure 4-1. Samples from the Open Clip Art Library: on the left, Simple Farm Animals 2 by user Viscious Speed; on the right, line drawings

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's *.ai* format, so you'll need software to convert those files into SVG. All images are released under the Creative Commons' public domain licence; you are encouraged to "buy a coffee" for the artists through donated 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" licence; some require attribution or are restricted to non-commercial use or to use within similarly-licenced work. Detailed licence information is available on each file's catalogue 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. The tagging and cataloguing of files on Wikipedia is often inconsistent, making searching a little difficult, there is plenty of great SVG content if you take the time to look around. Figure 4-2 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.

Wikipedia also contains most maps in SVG format, together with most logos and diagrams on the service.

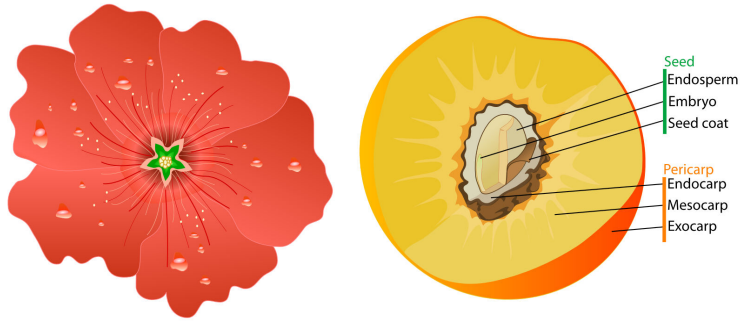


Figure 4-2. SVG from Wikimedia Commons: on the left, a hollyhock flower by user Ozgurel; on the right, a labelled diagram of a peach by Mariana Ruiz Villarreal (aka LadyofHats).

Iconic
: Iconic is a commercial SVG icon library, but they offer a set of more than 200 icons completely free to use (MIT licence; you must ensure that licence information is available in the file). 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 multi-color 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-licenced icons, searchable by concept using tags in dozens of languages.

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, photo-realistic 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.

Gzip should also be used by a performance-minded web developer (that's you!) to compress a file for transmission from web server to browser, but this is indicated via HTTP headers rather than with a file extension.

Ready-to-use graphics can solve many web design problems. For common interface icons, creating your own graphics may often feel like reinventing the wheel. But 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 collections above 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. 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 the standard SVG specification. To ensure that this extra information and features are retained when you save and reload the SVG (a process called "round-tripping"), these programs either have separate, proprietary image formats (like Adobe Illustrator's *.ai* format) or add extra markup to the SVG file (as Inkscape does). 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 export the final version of your graphic in standard SVG*. Without this step, the file may be many times larger than the "pure" SVG version, which will slow and complicate your site. We discuss additional SVG optimisation tools at the end of this chapter.

There are now numerous graphical applications that can export files as SVG. The defining feature of the apps described here are the visual, what-you-see-is-what-you-get (WYSIWYG) editors where you can position shapes with your mouse (or stylus, or finger) and select colors from on-screen palettes. They differ in the range of SVG features they support, and in how easy they are to use.

Regardless of which program you use, there are some common steps you can add to your workflow to make the final output more optimized for web use:

Define your Artboard or Drawing Size

The first step of creating a graphic should be to choose the right reference system for your work in your application of choice. While vector shapes are indifferent to size and scale, it's logical to use measurements that mirror the purpose of your SVG file. For the web, this usually means pixels. If your work is a precise illustration of a real-life object that must be faithfully repesented, you might choose inches, centimeters or millimeters.

For most software and export options, the size you choose for your artboard defines the coordinate system for the SVG `viewBox`. Making the artboard oversized will leave whitespace when the file is displayed, potentially ruining the display. You can always change the artboard size after initially setting it, and many programs allow you to crop the view to just fit the current drawing.

While it's important to size the artboard appropriately, it's also important not to draw to the very edges of the canvas: SVG elements that are exported while touching the edges may display trimmed anti-aliasing (the effects that help "smooth" display on raster screens) poorly. Leaving one or two pixels from the edge to any non-rectangular SVG elements that are meant to be fully displayed in the viewBox is usually a good idea.

Set up color preferences for web use
: Most vector illustration applications were originally designed for print, and therefore have their color space set to CMYK. RGB is much more appropriate for screen use, and has a wider gamut (range of colors) than CMYK; be sure to set this as your preference before you begin drawing.

Structure your design
: When you draw in SVG you're not just making images: you're creating *data*. For that reason, it's important to name each relevant object as you make it (lowercase, without any spaces, so it can be directly converted into a valid `id` attribute). It's much easier to create these references during visual editing, rather than later. Most applications will identify exported SVG elements by their layer or group name, but in some cases, you must explicitly turn on settings to use the custom names as `id` values.

  Use the application's grouping and layering feature wisely, for any sets of graphics you'll want to transform or style as a block. Groups are also important for the logical structure of your document, as they can be given alternative text (as described in ???).

Simplify Shapes
: In most SVG drawings, the list of coordinate points that describe each shape is the largest contribution to file size. When drawing, therefore, you want to use the fewest possible number of points to create a shape. This will in turn produce the best vector illustration, at the lowest file size.

  To help with this, the more advanced graphics programs over path simplification tools, which smooth out curves, reducing the number of points and keeping things manageable. Similarly, merging related shapes can simplify the file, unless you will be animating or dynamically styling them individually.

Decide whether to convert text
: An important feature of SVG graphics is that text can be searchable and easily editable. However, to keep it looking the way you designed on the web, you need to ensure that the end user has the correct fonts. This either means only using common fonts

or providing web fonts; in either case, you'll need to edit the CSS yourself to ensure proper fallbacks.

If precise text rendering is more important than editability—for example, in a logo—many programs allow you to convert styled text elements into graphical shapes for each letter. Adobe Illustrator will also provide you with the option to convert type to shapes when you export the SVG file.

Decide how to create a backdrop
: SVG backgrounds are transparent by default. If you want to create the appearance of a background color in a SVG document, you have three choices:

1. If the SVG is used as an image on a web page, provide a background-color for the image in CSS.
2. Add a `background` style for the `<svg>` root element in the SVG itself (or as part of your main document CSS, if you'll be using the code as inline SVG).
3. Include a colored rectangular backdrop shape in the SVG, that takes up the entire artboard, and make it the rearmost layer in the document.

The rest of this section introduces a sample of vector graphics programs for creating SVG, and offers some tips and warnings about how to get web-quality output from them.

Adobe Illustrator

Adobe Illustrator is the granddaddy of vector graphics programs, and debuted in 1991, a seminal period in Adobe's history. Illustrator 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). This means means is that Illustrator must perform a translation from it's internal vector graphics format (built primarily around Postscript) to SVG. For comparatively simple graphics, this is a straightforward process, but it is possible to create Illustrator images that have poor fidelity & large file sizes

when rendered to SVG, with the application replacing complex vector with embedded bitmap images.

The basic save-as-SVG option in Illustrator creates a complex file from which the native graphic can be reconstructed. However, a much more streamlined export-as-SVG option was introduced in Adobe Illustrator CC 2015, which creates a vector graphic optimized for the web.

You can also copy individual graphics components from Illustrator 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 in SVG, and want to use the visual editor to draw shapes. Unfortunately, it doesn't work with text elements: only the plain text content will be pasted, not the SVG markup.

Avoid using "Illustrator filters" in your graphic, as the application doesn't yet translate them well into SVG; use the "SVG filters" instead. Similarly, blending modes (Multiply, Dissolve, etc.) aren't yet translated into SVG-compatible CSS blend modes (as described in ???). If you use these, remove them before export and then edit the CSS yourself to add them back in. Finally, be sure all stroked shapes use centered strokes, as inside/outside strokes are not yet supported in SVG.

Adobe PhotoShop

The most recent versions of Adobe PhotoShop can also export SVG documents. While PhotoShop is primarily a bitmap editor, it supports vector shapes and text, making this a useful option to be aware of.

To export SVG from PhotoShop, name the layer or group with the intended name of the file: for example, `rect.svg`. Then, use `File / Generate / Image Assets`.

The generated SVG document will be exported into a folder provided with the name of the originating PSD file, with your named SVG documents inside the folder. By default, this folder will appear on your desktop, although you can change this in PhotoShop's preferences.

At the time of writing this book, there are several features of the exported SVG file to keep in mind:

1. The exported SVG is responsive (it has a `viewBox` attribute on the root svg element) but with default sizes (it also has `width` and `height` attributes).
2. The viewBox is automatically cropped to the edges of the largest vector shape in the current design.
3. Elements are given their own unique classes in an embedded style in the SVG.
4. Any bitmap images will be turned into inline base64 data URI in the SVG.

As an alternative to generating an SVG as an image asset, you can use PhotoShop's File / Export, including setting up preferences to make SVG the default "Quick Export" option. One upside of the "Generate Image Assets" approach is that PhotoShop will automatically keep the exported SVG updated with any changes made to the original PSD document, making updates and changes aautomatically. It's expected that Adobe Illustrator will take on a similar feature in the near future.

Sketch

Sketch is a Mac-only program that has proved extremely popular with user interface designers. Unfortunately, the SVG export capabilities of version 3.5.2 (the latest at the time of writing) is stuck where Adobe Illustrator was at five years ago… although that's not to say that the result can't be improved and cleaned up.

First, Sketch must be informed that a shape can be exported as SVG: you'll find this option at the bottom of the properties panel for a selected shape. Sketch will deliver each shape as its own separate SVG file, unless multiple shapes are merged.

By default, Sketch applies stroke to elements on the inside, which (as previously discussed) SVG does not yet support. If you intend your export to be similar to your drawing, ensure that Sketch's preferences are set to stroke paths in their center.

A typical export of SVG code for a simple path looks something like this (with some line breaks added, so it will fit on a page:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg width="553px" height="119px" viewBox="0 0 553 119"
     version="1.1" xmlns="http://www.w3.org/2000/svg" xml:lang="en"
```

```
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xmlns:sketch="http://www.bohemiancoding.com/sketch/ns">
    <!-- Generator: Sketch 3.5.2 (25235)
         - http://www.bohemiancoding.com/sketch -->
    <title>Path 1</title>
    <desc>Created with Sketch.</desc>
    <defs></defs>
    <g id="Page-1" stroke="none" stroke-width="1" fill="none"
       fill-rule="evenodd" sketch:type="MSPage">
        <path d="M2.90625,41.7265625 C88.1132812,-5.6484375
                 202.457031,-9.69140625 286.011719,47.7382813
                 C369.566406,105.167969 451.902344,130.687498
                 551.433594,102.425781"
                 id="Path-1" stroke="#F96262" stroke-width="9"
                 sketch:type="MSShapeGroup"></path>
    </g>
</svg>
```

A lot of this code is extraneous; Sketch has snuck it's own proprietary code in the output, under the `sketch` namespace.

To create the cleanest possible SVG output from Sketch, adhere to the same rules we established for Illustrator, whch a few additions:

1. Create an artboard for each drawing (`Insert / Artboard`), and one drawing (such as an icon) per artboard.
2. Remove any bounding boxes from the drawing.
3. Don't attempt to rotate your drawing in Sketch before export, as doing so will (in current versions) significantly distort the SVG export.

Following these rules will eliminate many of the current issues with SVG export from Sketch, but not all of them; if you find the process too arduous, you may find it easier to copy and paste your Sketch drawings into another veoctor application, using it's native SVG support for export instead.

Adobe Animate

Part of the original intention behind SVG was to create a viable, open and free alternative to Macromedia Flash, the standard for vector graphics and animation on at the earliest inception of the web. When Adobe bought Macromedia, it quickly added SVG export to the options in the application. This option has been inherited by Adobe Animate, the modern version of Adobe Flash, and is available under `File / Export / Image`. The exported code is typically

overblown, and needs to be trimmed and optimized (see the applications and utilities at the end of this chapter).

Adobe Animate will likely prove useful for exporting documents developed in older versions of Flash (with which it remains compatible) to SVG. However, basic SVG export doesn't include any animation; it only exports the first frame. Other export options and plugins can convert animation to JavaScript.

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 4-3) 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 non-graphical 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.

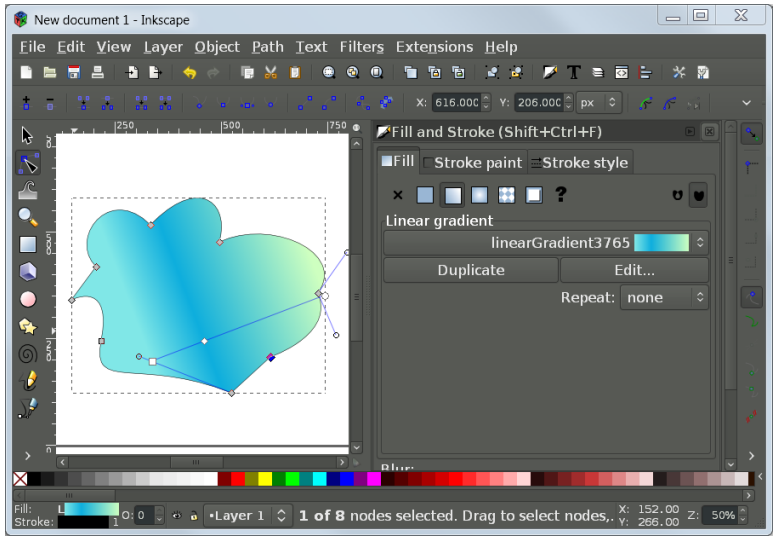


Figure 4-3. The open source Inkscape graphics editor

There are a few features in Inkscape that can make SVG exports easier:

1. Use File / Clean up document to remove unused `<def>` elements.
2. If you are copying an element for use elsewhere in the same drawing, create a Clone (`Edit / Clone`), which generates a `<use>` reference to the original element. This reduces output code size and makes later editing far easier.
3. Use `Path / Simplify` to reduce the number of points in an over-drawn element, further reducing file size.

Inkscape uses SVG as its own native format, but with extra application-specific data in its own XML namespaces. In addition, 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.

When saving your file, you'll need to be very clear exactly what kind of SVG you are saving from Inkscape: "Inkscape" SVG, with its extra embedded code, or "plain" SVG. The confusion is added to by the

fact that Inkscape actually has multiple SVG output "formats", all of which have a similar file extension, but which preserve different amounts of data:

Inkscape SVG
: the fully infused file, with Inkscape XML extensions; use this for your working copy, particularly if you use any of Inkscape's unique features, such as "live" vector effects. Your "SVG Output" settings in the Inkscape preferences dialog are used to determine how much numerical precision to preserve, how to encode styles, and whether to remove unrecognized attributes (unselect this if you custom-edit the XML).

Plain SVG
: the basic SVG file, without Inkscape-specific data, but otherwise unmodified. Again, the "SVG Output" settings are used.

Compressed Inkscape SVG and Compressed Plain SVG
: the gzipped version of the file. Generally, this is more trouble than its worth, since it prevents any edits to the file in a text editor; use server-based gzip compression instead.

Optimized SVG
: a plain SVG which has been modified to reduce file size; this should be your choice in most cases when exporting the final copy for the web. Important options to check include:

  - Shorten color values converts colors to hexadecimal values, in the most compact syntax if possible (`#rgb` rather than `#rrggbb`).
  - Group collapsing eliminates empty or redundant groups (be careful of this if you created nested groups on purpose for animations)
  - Create groups for similar attributes reduces the number of repeated style declarations (which is not as good as using CSS classes, but could make it easier to convert to classes yourself).
  - Embed rasters means that bitmap image files will be converted into inline base64 data URI (use this if your SVG file needs to work as a stand-alone image file; uncheck if the SVG will be copied into inline HTML or used as an embedded object, with access to the image in an assets folder).

- Remove metadata and comments takes out Inkscape editor additions (be careful if you've added your own metadata).
- Remove XML declaration eliminates the XML prolog that is unnecessary for web use.
- Enable viewboxing adds a viewBox attribute to the file, essential for proper scaling (as described in ???).
- The significant digits setting rounds path coordinates to the specified numerical precision; smaller numbers can substantially reduce file size, but too small and you'll distort the graphic (3 or 4 is reasonable for most web use).

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 below) 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 web site, 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 4-4 shows the interface.

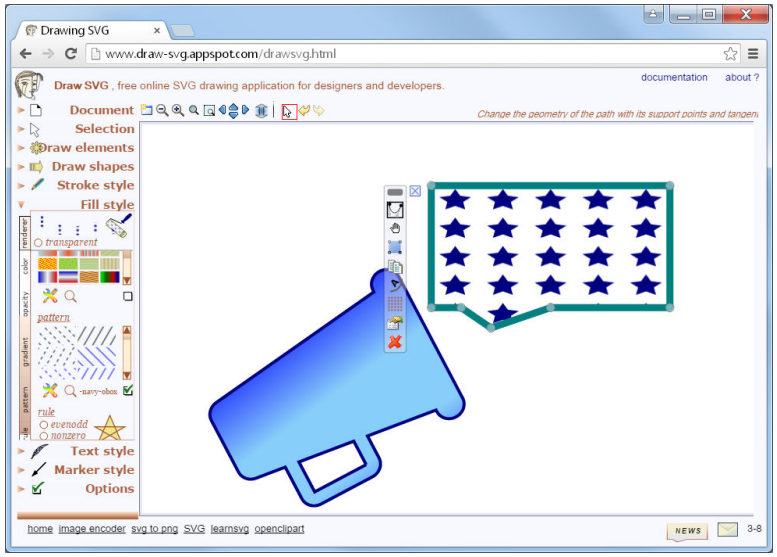


Figure 4-4. The Draw SVG free online SVG graphics editor

Boxy SVG

A recent addition to the list of browser-based SVG editors, Boxy SVG is a full graphical editor in the form of a Chrome plug-in; in use, it is indistinguishable from a desktop application. The website demo also works in any web browser. It has an attractive interface and excellent support for SVG features, including new features such as blend modes. Equally importantly, it is being actively developed.

Nearly all of the SVG editors described in this section 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 set-ups, the process can be automated using dedicated rasterization and conversion tools.

SVG Snapshots

Rasterization is the conversion of a vector graphic into a bitmap format. Broadly speaking, any application that can display an SVG on a screen is a rasterizer. 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 CSS 3 features which 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 a "Swiss army knife" for command line manipulation of graphics, and is available on Windows, Macintosh, and Linux platforms. It can be used 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 vector graphics language, used as a programming library in other software; it has implementations in many common programming languages such as C++ and Python. Cairo graphics can be converted into 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 neccessarily 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.

Similar "headless" versions of the other browser rendering engines have been proposed, with a headless Chromium browser expected soon.

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 before scanning them into your computer and converting the drawings to SVG.

Bringing SVG Alive

Rendering SVG into other static file formats is useful, but the reason you're using SVG in the first place is because it contains 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 these 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. As of mid-2016, all the major browsers have pledged not to introduce new prefixed CSS properties for web content. New features are now usually enabled with experimental browser modes (or "flags") controlled by the user. Conversely, some of the most widely-used prefixed properties have now been adopted into the specifications as deprecated synonyms, supported by all new browsers. Content creators should, of course, use the standard un-prefixed versions.

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 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; since mid-2016, Firefox also supports the most common `-webkit-` properties.

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 CSS 3 features require a `-webkit-` prefix on Safari and related software, although the development team has now committed to transitioning away from prefixes.

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 in native applications on newer Android devices. It is also used by other new browsers, such as Vivaldi and Brave, and by the Samsung Internet browser on Samsung Android devices.

Blink browsers still support `-webkit-` CSS properties, although not necessarily those introduced since the split.

Initial development of the Google Chrome browser (and now Blink in general) was heavily focused on performance; their SVG implementation is one of the best, and animations are generally fast and smooth (although Firefox has since 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 in favor of CSS or scripted animations. At the time of writing (mid-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 CSS 3 properties. However, it has not been (and will not likely be) substantially updated since 2013. Presto versions of Opera used an `-o-` prefix for experimental CSS features, but it is unlikely to be useful in modern websites.

Trident for Internet Explorer and other Windows programs

Internet Explorer was the last major browsers to introduce SVG support. 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]

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 devel-

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

opers 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/CSS 3 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 also introduced support for some `-webkit-` prefixes that are commonly used in existing websites.

Servo
: The Mozilla foundation is sponsoring the development of a new browser rendering engine, Servo, that may one day replace Gecko at the core of Firefox. It is being built from scratch in Rust, a programming language optimized for parallel computing environments. At the time of writing, there has been no work on SVG rendering within Servo, although they have an open issue to add support.

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
SVG on the GPU

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 rasterization and for some filter effect 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.

Unfortunately, the other common use of hardware acceleration—for compositing different layers of a web page—is not currently used by the web browsers to optimize animations within an SVG canvas, although they use it for CSS box model layers.

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, 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 111. 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* extensions, then open the same file in a web browser.

If you typed carefully, and didn't forget any required attributes or mis-spell 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 that are 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 auto-complete 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 stand-alone 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 wreck havoc and external files and stylesheets are not supported. Figure 4-5 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.

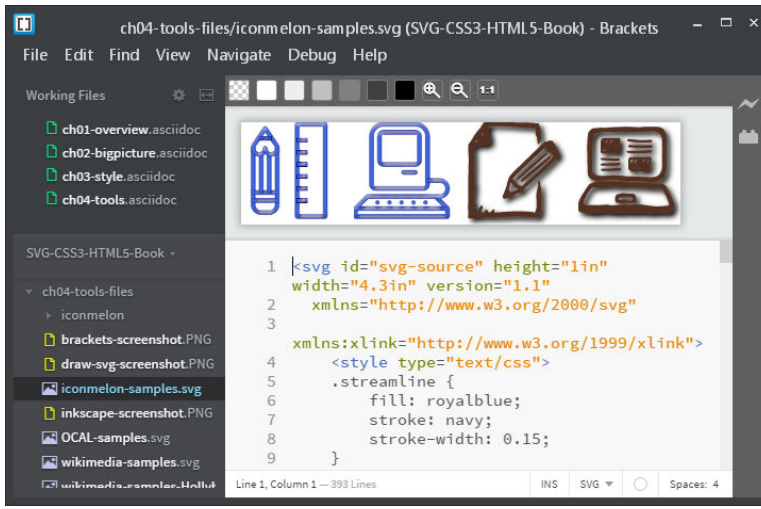


Figure 4-5. The Brackets code editor with SVG Preview enabled

Online live code sites

In recent years, numerous web applications have become available which 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 that you can 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 "Pro" 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 licence 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.

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 be updated 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 terminol-

ogy 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 non-commercial ones); a paid licence is required if the site's end users pay a 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 CSS 3.

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 which you may want to use while uploading SVG to your web server. A sample routine, which could be automated, would be to:

- 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).
- 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 123).
- 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).
- 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.

For ???, however, we will focus on creating graphics entirely with code, defining the raw shapes that make up your image using elements and attributes.