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.

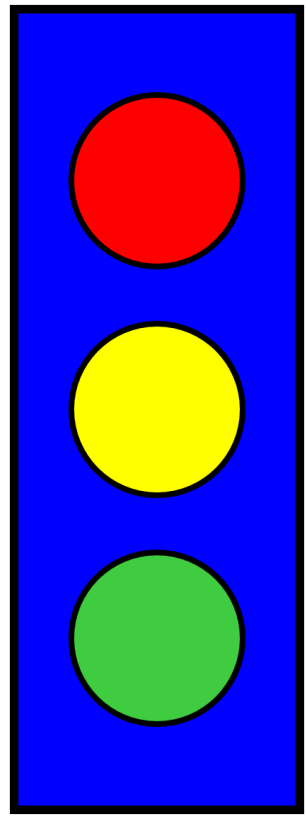

Figure 1-1. Primary color stoplight graphic

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.

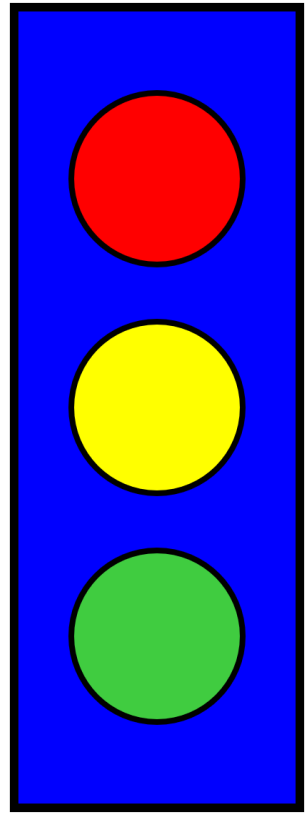

Figure 1-2. Stoplight drawn with re-used elements

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>
```

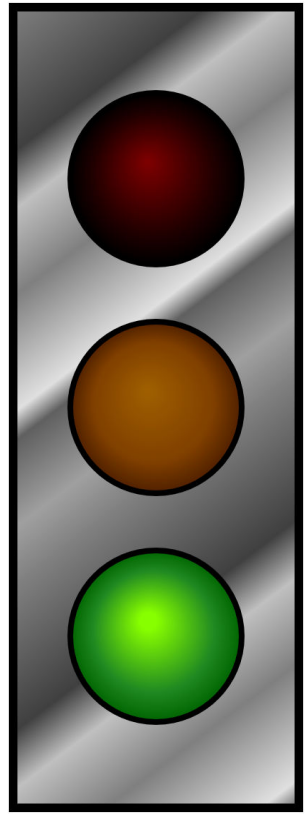

Figure 1-3. Stoplight with gradient fills

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.

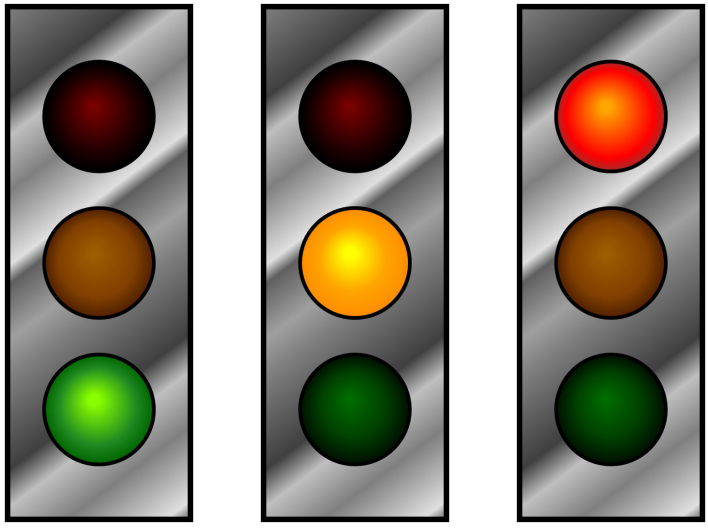

Figure 1-4. Three stages of an animated stoplight with gradient fills

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.

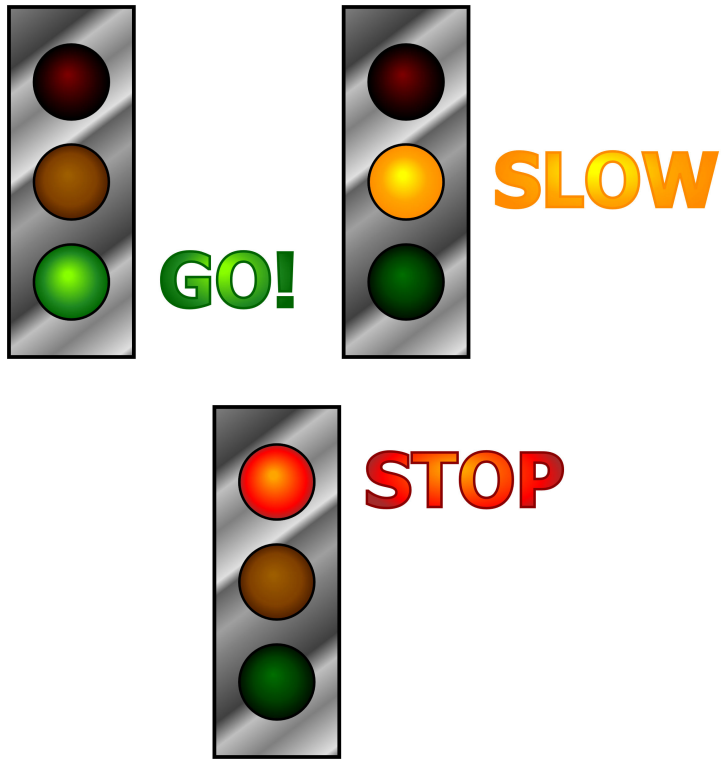

Figure 1-5. Three stages of a labelled, animated stoplight

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).

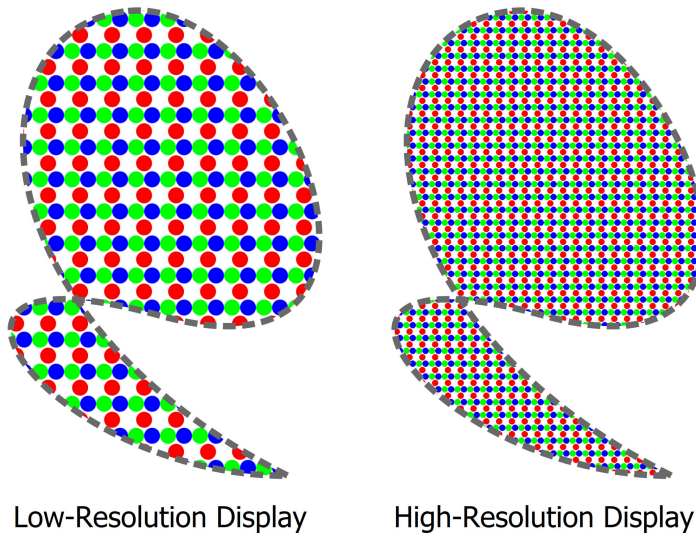

Figure 1-6. A vector graphic rasterized at different resolutions

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.