SVG Programming: The Graphical Web

KURT CAGLE

Apress™

SVG Programming: The Graphical Web
Copyright © 2002 by Kurt Cagle

ISBN (pbk): 1-59059-019-8
Printed and bound in the United States of America 12345678910

Technical Reviewer: Don Demcsak
Editorial Directors: Dan Appleman, Peter Blackburn, Gary Cornell, Jason Gilmore, Karen Watterson, John Zukowski
Project Manager: Tracy Brown Collins
Copy Editor: Kim Wimpsett
Production Editor: Grace Wong
Compositor: Impressions Book and Journal Services, Inc.
Indexer: Ron Strauss
Cover Designer: Kurt Krames
Manufacturing Manager: Tom Debolski
Marketing Manager: Stephanie Rodriguez

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.
In the United States, phone 1-800-SPRINGER, email `orders@springer-ny.com`, or visit `http://www.springer-ny.com`.
Outside the United States, fax +49 6221 345229, email `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax: 510-549-5939, email `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section.

CHAPTER 2

Getting Started: An SVG Tutorial

THE BEST WAY to learn is to do. Although it is not always possible within this book to create step-by-step walk-throughs, you can get up and running with SVG surprisingly quickly. This chapter in particular is not meant for deep insight into the creative process or programming models—instead, it is a set of simple recipes you can use to create more sophisticated applications.

Whenever possible, I also point you to the primary area that each example most heavily uses. Like any real meaningful languages, even the simplest of Scalable Vector Graphics (SVG) code requires the use of integrated concepts. Try playing with the examples, seeing if you can come up with the solutions to the problems posed at the end of each exercise (answers are available on my Web site at `www.kurtcagle.net`).

To work through the examples in this chapter, you will need to download and install the Adobe SVG Viewer 3.0 for both Windows- and Linux-based systems. Also, the examples covered in this chapter assume Internet Explorer 5.0 for the client and Internet Information Services (IIS) for the server (when a server is needed), but the same principles can easily be ported over to Apache on Linux.

Starting with a Stop

Creating a simple stop sign is remarkably useful for illustrating a number of different concepts. You can use stop signs to do everything from creating static images that enhance the *STOP* message to a creating a forum for political commentary by replacing the *STOP* word with some other text on a rollover. Moreover, you can go with the simple, flat stop sign or instead create a graphic with a fairly high degree of realism with SVG.

Creating a Simple, Flat Stop Sign

You have to start somewhere, and a basic sign is the simplest place to begin:

1. Install the Adobe SVG Viewer 3.0 from `www.adobe.com/svg`.

2. Open your favorite text editor to an empty document, and save it as **tutStopSign1.svg**.

3. Enter the script from Listing 2-1 into `tutStopSign1.svg`.

4. Save the file, and then open Internet Explorer or Mozilla 1.0. If this is the first SVG file you have ever viewed, a dialog box will first appear containing the licensing agreement for the viewer. Accept it (the viewer will not work if you do not—though by all means read the fine print), and you will see an SVG image that looks like Figure 2-1.

Listing 2-1. `tutStopSign1.svg`

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="300" height="400" viewBox="0 0 300 400"
     preserveAspectRatio="none">
    <desc>Stop Sign 1</desc>
    <rect x="0" y="0" width="300" height="400"
           fill="black" id="background"/>
    <g transform="translate(150,150)">
        <rect x="-10" y="0" width="20" height="250"
            fill="gray" id="pole"/>
        <path id="signShape"
            d="m0,-100 l40,0 l60,60 l0,80 l-60,60 l-80,0 l-60,-60
                        l0,-80 l60,-60z"/>
        <use xlink:href="#signShape" x="0" y="0"
            fill="white" id="signWhite"/>
        <use xlink:href="#signShape" x="0" y="0" fill="red"
            transform="scale(0.95)" id="signRed"/>
        <text x="0" y="0"
            font-size="72"
            font-family="Arial Narrow"
            fill="white"
            text-anchor="middle"
            dominant-baseline="mathematical">
STOP
```

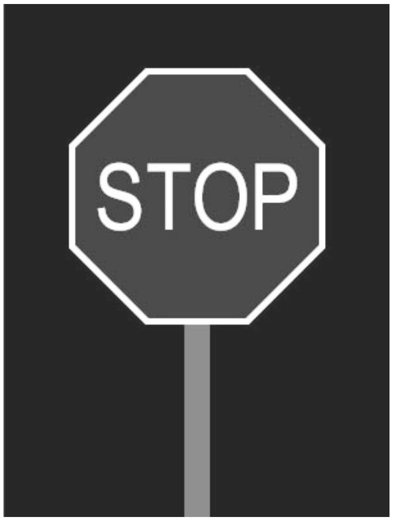


Figure 2-1. A simple stop sign with flat shapes and vectors

```
</text>
    </g>
</svg>
```

So what is going on here? A quick step-by-step analysis demonstrates how easy SVG can be to create. The first part of the `<svg>` sets the stage, so to speak, identifying the namespaces and the coordinate system:

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="300" height="400" viewBox="0 0 300 400"
     preserveAspectRatio="none">
```

Perhaps the best way of thinking about a *namespace* is to envision that it serves the same role to Extensible Markup Language (XML) as a ProgID or class name serves to COM or Java, respectively—it both uniquely identifies the role that elements with that prefix have and tells the respective processors what engine to run to make the XML actually do something useful. In this case, there are two namespaces: the default namespace that identifies SVG (`www.w3.org/2000/svg`) and a linking namespace for creating internal references to existing blocks of SVG code (`www.w3.org/1999/xlink`).

NOTE *The formal definition of a namespace is remarkably complicated because a namespace essentially defines and identifies the vocabulary used to describe a specific object, procedure, or mechanism. Informally, the namespace Uniform Resource Identifier (or namespace URI) is frequently called the namespace as well, but this URI is , in fact, a way of uniquely labeling the more conceptual namespace itself. Finally, the namespace prefix is a convenient nonunique label to identify elements in the namespace and is mapped to a namespace through an expression such as* `xmlns:svg=http://www.w3.org/2000/svg` *for the SVG namespace.*

After this and a brief internal description, the various elements are defined. The black background is established as a rectangle element:

```
<rect x="0" y="0" width="300" height="400" fill="black" id="background"/>
```

The pole is similarly a rectangle, though a gray one:

```
<rect x="-10" y="0" width="20" height="250" fill="gray" id="pole"/>
```

The sign is a more complicated shape and uses the `<path>` element to define the specific points that the sign uses:

```
<path id="signShape"
d="m0,-100 l40,0 l60,60 l0,80 l-60,60 l-80,0 l-60,-60 l0,-80 l60,-60z"/>
```

Because the shape is used twice—once for the external white part of the sign and again for the internal red part—rather than duplicating it, the SVG document invokes the `<use>` element to employ the path as a shape, scaling the dimensions of the shape down by 5 percent to draw the smaller red portion:

```
<use xlink:href="#signShape" x="0" y="0" fill="white" id="signWhite"/>
 <use xlink:href="#signShape" x="0" y="0" fill="red"
transform="scale(0.95)" id="signRed"/>
```

Finally the text element is drawn, positioned so that it will always be centered with respect to the sign:

```
<text x="0" y="0"
      font-size="72"
      font-family="Arial Narrow"
      fill="white"
      text-anchor="middle"
      dominant-baseline="mathematical">STOP</text>
```

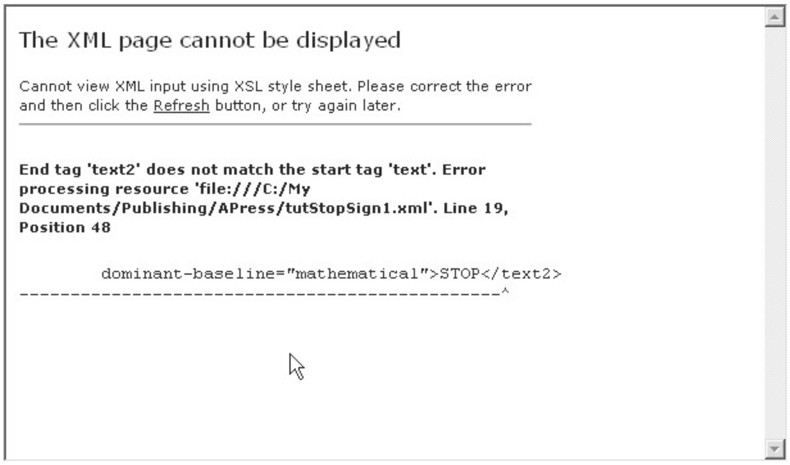
The next several chapters discuss the specific properties discussed in this section in considerably greater depth. For now, experiment with changing various properties to see what effects they have.

Debugging SVG

If everything goes properly, you should in fact see the image in Figure 2-1. However, in the real world, things seldom go completely as planned—especially with large blocks of SVG. Furthermore, the Adobe SVG Viewer does not help matters much because it is what is known as a “lazy” parser. In other words, it will basically draw the document up to the point where an error occurs, rather than just failing outright. Although this can mean you can get *most* of your graphic on the screen, it also means that tracking down problems often requires the recognition that there is a problem in the first place, something that is not always immediately obvious.

One of the best advantages of working with SVG is the fact that it is XML. If you utilize an XML processor, you can determine whether you have well-formed XML (which is where a significant portion of all problems with SVG usually stem from). Fortunately, both Internet Explorer and Mozilla 1.0 include such processors, but they are (at least on the Windows side) typically only triggered if the file itself has an .xml extension.

So, if you are having problems in the previous file, change the name from `tutStopSign1.svg` to `tutStopSign1.xml` and view it in either of the two browsers (Internet Explorer is probably more reliable on this front, but both will do). If the document is not well formed, you will probably get a message that looks something like Figure 2-2. In that case, make sure the end tags match the start tags and that nothing is improperly contained. Once you are done, check again, and if it works, rename the file to **tutStopSign1.svg** and trying viewing it again.



The XML page cannot be displayed

Cannot view XML input using XSL style sheet. Please correct the error and then click the Refresh button, or try again later.

End tag 'text2' does not match the start tag 'text'. Error processing resource 'file:///C:/My Documents/Publishing/APress/tutStopSign1.xml'. Line 19, Position 48

```
        dominant-baseline="mathematical">STOP</text2>
-----------------------------------------------^
```

Figure 2-2. A common error message

Creating a More Realistic Stop Sign

The stop sign you just created is . . . well, less than living up to its full potential. Frankly, it is ugly. Specifically, it has no sense of depth, shading, texture, or any of the other cues that help in making graphics appear more realistic. Additionally, it does not begin to show off the real capabilities of SVG. Consider a real stop sign: It is usually on some kind of metallic pole, with screws holding it into place, the light is seldom uniform over its surface, and it is usually at least a little dirty.

Is that doable in SVG? Sure, though to do so well, you have to create a little more complex SVG document:

1. Open your favorite text editor to an empty document, and save it as **tutStopSign2.svg**.

2. Enter the SVG code in Listing 2-2 into `tutStopSign2.svg`.

3. Open Internet Explorer or Mozilla and view the file (see Figure 2-3).

Listing 2-2. `tutStopSign2.svg`

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="300" height="400" viewBox="0 0 300 400"
     preserveAspectRatio="none">
    <defs>
```

```
<linearGradient id="signGrad" gradientTransform="rotate(45)">
    <stop offset="0%" stop-color="white" stop-opacity="0"/>
    <stop offset="100%" stop-color="black" stop-opacity=".4"/>
</linearGradient>
<radialGradient id="boltGrad"
        gradientTransform="translate(-0.2,-0.2)">
    <stop offset="0%" stop-color="#E0E0E0"/>
    <stop offset="70%" stop-color="#808080"/>
    <stop offset="100%" stop-color="black"/>
</radialGradient>
<linearGradient id="steel">
    <stop offset="0%" stop-color="#C0C0C0"/>
    <stop offset="20%" stop-color="#404040"/>
    <stop offset="30%" stop-color="#FFFFFF"/>
    <stop offset="60%" stop-color="#808080"/>
    <stop offset="65%" stop-color="#202020"/>
    <stop offset="70%" stop-color="#C0C0C0"/>
    <stop offset="100%" stop-color="#8080"/>
</linearGradient>
<linearGradient id="steelShadow" gradientTransform="rotate(90)">
    <stop offset="0%" stop-color="black" stop-opacity="1"/>
    <stop offset="40%" stop-color="black" stop-opacity="1"/>
    <stop offset="42%" stop-color="black" stop-opacity="0"/>
    <stop offset="100%" stop-color="black" stop-opacity="0"/>
</linearGradient>
  <filter id="Turb5" filterUnits="objectBoundingBox"
          x="0%" y="0%" width="100%" height="100%">
    <feTurbulence type="fractalNoise" baseFrequency="0.3" numOctaves="4"/>
  </filter>
<image xlink:href="ForbiddingTrimmed.jpg" x="0" y="0"
                width="300" height="400" fill="black"
                id="background"/>
<path id="signShape" d="m0,-100 l40,0 l60,60
l0,80 l-60,60 l-80,0 l-60,-60 l0,-80 l60,-60z"/>
<mask maskUnits="objectBoundingBox" id="signMask">
            x="0%" y="0%" width="100%" height="100%"
    <use xlink:href="#signShape" x="0" y="0" width="100%"
            height="100%" fill="white"/>
</mask>
<circle cx="0" cy="0" r="4" fill="url(#boltGrad)" id="bolt"/>
<g transform="translate(150,150)" id="sign">
    <rect x="-10" y="0" width="20" height="250"
            fill="url(#steel)" id="pole"/>
            <rect x="-10" y="0" width="20" height="250"
                    fill="url(#steelShadow)" id="pole"/>
            <use xlink:href="#signShape" x="0" y="0" fill="white"
                    id="signWhite"/>
            <use xlink:href="#signShape" x="0" y="0" transform="scale(0.95)"
                    id="signInner"/>
            <text x="0" y="0" font-size="72" font-family="Arial Narrow"
                    fill="white" text-anchor="middle"
                    dominant-baseline="mathematical">
                  STOP
                  </text>
            <use xlink:href="#signShape" x="0%" y="0%" filter="url(#Turb5)"
                    id="signShade"  mask="url(#signMask)" opacity="0.4"/>
            <use xlink:href="#signShape" x="0%" y="0%" fill="url(#signGrad)"
                    id="signShade"  mask="url(#signMask)"/>
            <use xlink:href="#bolt" x="0" y="-80"/>
            <use xlink:href="#bolt" x="0" y="80"/>
        </g>
        </defs>
        <use xlink:href="#background" x="0" y="0"/>
        <use xlink:href="#sign" x="0" y="0" fill="red"/>
    </svg>
```

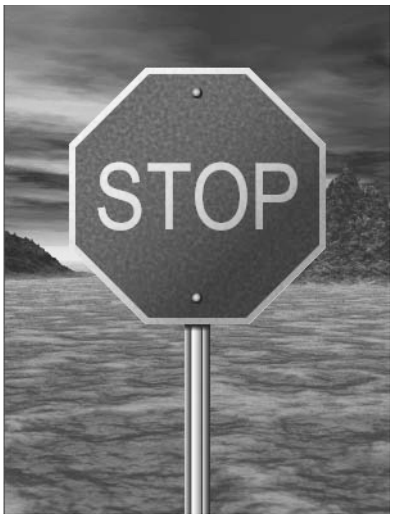


Figure 2-3. A richer stop sign, combined with a bitmap graphic

This is a little more like it. The image in the background (done in a 3D-rendering program) gives a sense of foreboding and makes the sign seem considerably more solid. The sign itself has a subtle overlay gradient that makes it appear to be slightly warped (perhaps someone did not stop quite soon enough). The pole now has depth and solidity, down to the shadow that the sign casts on it, and the steel bolts on the sign now seem to actually hold it to the pole. If you look closely enough, you will also see a faint patina of dirt on the sign.

This illustrates a little more of the true power of SVG. You can combine fairly complex gradients with Photoshop-like filters and bitmap graphics to create both realistic and abstract graphics. The fascinating thing about all of this is that the second stop sign is not all that much more complicated than the first sign in terms of the SVG document, as a little analysis will reveal. The `<svg>`-containing element is the same, defining the image as having a height of 400 pixels and a width of 300 pixels:

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="300" height="400" viewBox="0 0 300 400" preserveAspectRatio="none">
```

The `<defs>` element defines elements to be used later in the graphic without displaying the pieces ahead of time; think of it as a place where templates are stored for defining more complex elements.

In this example, the first things to define are *gradients*: shades of color that can be assigned to specific objects. You can create gradients that run in a line from starting to ending color (`<linearGradient>` elements) or that grow in a circle (technically an oval) from a central point outward (`<radialGradient>`). Gradients can also combine varying levels of opacity in the mix, as illustrated in the `#signGrad` gradient:

```
    <linearGradient id="signGrad" gradientTransform="rotate(45)">
        <stop offset="0%" stop-color="white" stop-opacity="0"/>
        <stop offset="100%" stop-color="black" stop-opacity=".4"/>
    </linearGradient>
    <radialGradient id="boltGrad" gradientTransform="translate(-0.2,-0.2)">
        <stop offset="0%" stop-color="#E0E0E0"/>
        <stop offset="70%" stop-color="#808080"/>
        <stop offset="100%" stop-color="black"/>
    </radialGradient>
    <linearGradient id="steel">
        <stop offset="0%" stop-color="#C0C0C0"/>
        <stop offset="20%" stop-color="#404040"/>
        <stop offset="30%" stop-color="#FFFFFF"/>
        <stop offset="60%" stop-color="#808080"/>
        <stop offset="65%" stop-color="#202020"/>
        <stop offset="70%" stop-color="#C0C0C0"/>
        <stop offset="100%" stop-color="#8080"/>
    </linearGradient>
    <linearGradient id="steelShadow" gradientTransform="rotate(90)">
        <stop offset="0%" stop-color="black" stop-opacity="1"/>
        <stop offset="40%" stop-color="black" stop-opacity="1"/>
        <stop offset="42%" stop-color="black" stop-opacity="0"/>
        <stop offset="100%" stop-color="black" stop-opacity="0"/>
    </linearGradient>
```

The `<filter>` element makes it possible to apply a number of special effects to one of the various SVG shape elements, making an object blur, creating highlights and lighting effects, generating drop shadows and glows, and composing multiple elements (and potentially bitmaps) into single blocks. Filters tend to be expensive (in terms of processing time), but for static graphics they can create extremely powerful effects (see Chapter 10, "SVG Components"). The following filter, `#Turb5`, was responsible for creating the "dirty" surface on the sign:

```
<filter id="Turb5" filterUnits="objectBoundingBox"
         x="0%" y="0%" width="100%" height="100%">
    <feTurbulence type="fractalNoise" baseFrequency="0.3" numOctaves="4"/>
</filter>
```

The image element makes it possible to combine vector graphics with bitmaps. Using the XML Xlinking notation, images can reference JPEG, GIF, or PNG images universally and can also load in SVG graphics as static images:

```
<image xlink:href="ForbiddingTrimmed.jpg" x="0" y="0"
         width="300" height="400" fill="black" id="background"/>
```

As with `tutStopSign1.svg`, the `<path>` element in `tutStopSign2.svg` defines the octagonal shape of the stop sign. But in addition to its use by the white and red portions of the sign, this image also uses `<path>` to create a mask so that when the filter or gradients are applied to it they are only drawn within the dimensions of the sign itself, and not in the rest of the signs bounding rectangle:

```
    <path id="signShape"
      d="m0,-100 l40,0 l60,60 l0,80 l-60,60
            l-80,0 l-60,-60 l0,-80 l60,-60z"/>
    <mask maskUnits="objectBoundingBox" x="0%" y="0%"
            width="100%" height="100%" id="signMask">
        <use xlink:href="#signShape" x="0" y="0"
            width="100%" height="100%" fill="white"/>
    </mask>
```

The shape definition and layering is similar between the first and second version but makes more extensive use of the `<use>` element to build multiple sandwiched layers of predefined components: sign, post, text, shading and textures, and bolts, all tied together in a single package called `#sign`. For example:

```
    <circle cx="0" cy="0" r="4" fill="url(#boltGrad)" id="bolt"/>
    <g transform="translate(150,150)" id="sign">
        <rect x="-10" y="0" width="20" height="250"
                 fill="url(#steel)" id="pole"/>
        <rect x="-10" y="0" width="20" height="250"
                 fill="url(#steelShadow)" id="pole"/>
        <use xlink:href="#signShape" x="0" y="0"
                 fill="white" id="signWhite"/>
        <use xlink:href="#signShape" x="0" y="0"
                 transform="scale(0.95)" id="signInner"/>
        <text x="0" y="0" font-size="72" font-family="Arial Narrow"
                 fill="white" text-anchor="middle"
                dominant-baseline="mathematical">
                STOP
        </text>
        <use xlink:href="#signShape" x="0%" y="0%"
               filter="url(#Turb5)" id="signShade"
               mask="url(#signMask)" opacity="0.4"/>
        <use xlink:href="#signShape" x="0%" y="0%"
               fill="url(#signGrad)" id="signShade"  mask="url(#signMask)"/>
        <use xlink:href="#bolt" x="0" y="-80"/>
        <use xlink:href="#bolt" x="0" y="80"/>
    </g>
</defs>
```

Once outside the `<defs>` element, all elements that are listed actually render to the canvas of the SVG:

```
    <use xlink:href="#background" x="0" y="0"/>
    <use xlink:href="#sign" x="0" y="0" fill="red"/>
</svg>
```

Here the graphic is rendered through two simple calls: one to draw the background, and the other to draw the sign. In well-defined SVG, abstraction—hiding the irrelevant details of a drawing at successive points in the document—provides the key to both programmatic capabilities and ease of design. I will return to this point throughout the book.

Adding a Sunset or Sunrise

SVG is not just for static graphics, though as `tutStopSign2.svg` illustrates, you can create impressive two-dimensional graphics with SVG. However, one of the goals of the SVG language is to do more than simply provide an alternative to PostScript. Although never explicitly stated within the SVG charter, certainly one area where there is significant overlap in functionality comes from the vector animation program Flash, which utilizes a binary format for storing and manipulating shapes, images, and text. Flash started as Future Splash, which recognized that if you could just keep track of the changes in vector characteristics of an animated graphic, you could send an animation over the wire in a fraction of the space that you would need to send a digital video animation or even a GIF image.

This saving was not lost to the SVG development committee. The SVG recommendation incorporates portions of the Synchronized Multimedia Integration Language (SMIL) to perform a novel function: providing a method to indicate when specific animations should take place, how they take place, and against which properties.

You can modify the stop sign graphic through the use of these SMIL elements to perform any number of different tasks, such as changing the position of elements, changing their opacities, changing text contents, or even reacting to mouse clicks. For instance, try typing in `tutStopSign3.svg` (Listing 2-3) and see what happens (see Figure 2-4).

Listing 2-3. `tutStopSign3.svg`

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
                width="300" height="400" viewBox="0 0 300 400"
                preserveAspectRatio="none">
    <defs>
      <filter id="Turb5" filterUnits="objectBoundingBox"
               x="0%" y="0%" width="100%" height="100%">
        <feTurbulence type="fractalNoise" baseFrequency="0.3"
                numOctaves="4"/>
      </filter>
    <linearGradient id="signGrad" gradientTransform="rotate(45)">
        <stop offset="0%" stop-color="white" stop-opacity="0"/>
        <stop offset="100%" stop-color="black" stop-opacity=".4"/>
    </linearGradient>
    <radialGradient id="boltGrad" gradientTransform="translate(-0.2,-0.2)">
        <stop offset="0%" stop-color="#E0E0E0"/>
        <stop offset="70%" stop-color="#808080"/>
        <stop offset="100%" stop-color="black"/>
    </radialGradient>
```

```
<linearGradient id="steel">
    <stop offset="0%" stop-color="#C0C0C0"/>
    <stop offset="20%" stop-color="#404040"/>
    <stop offset="30%" stop-color="#FFFFFF"/>
    <stop offset="60%" stop-color="#808080"/>
    <stop offset="65%" stop-color="#202020"/>
    <stop offset="70%" stop-color="#C0C0C0"/>
    <stop offset="100%" stop-color="#8080"/>
</linearGradient>
<linearGradient id="steelShadow" gradientTransform="rotate(90)">
    <stop offset="0%" stop-color="black" stop-opacity="1"/>
    <stop offset="40%" stop-color="black" stop-opacity="1"/>
    <stop offset="42%" stop-color="black" stop-opacity="0"/>
    <stop offset="100%" stop-color="black" stop-opacity="0"/>
</linearGradient>
<image xlink:href="ForbiddingTrimmed.jpg" x="0" y="0"
        width="300" height="400" fill="black" id="background"/>
<image xlink:href="SandDune.jpg" x="0" y="0"
        width="300" height="400" fill="black" id="background2"/>
<path id="signShape"
        d="m0,-100 l40,0 l60,60 l0,80 l-60,60
              l-80,0 l-60,-60 l0,-80 l60,-60z"/>
<mask maskUnits="objectBoundingBox" x="0%" y="0%"
        width="100%" height="100%" id="signMask">
    <use xlink:href="#signShape" x="0" y="0"
        width="100%" height="100%" fill="white"/>
</mask>
<circle cx="0" cy="0" r="4" fill="url(#boltGrad)" id="bolt"/>
<rect id="night" x="0" y="0" width="300" height="400" fill="black"/>
<g transform="translate(150,150)" id="sign">
    <rect x="-10" y="0" width="20" height="250"
            fill="url(#steel)" id="pole"/>
    <rect x="-10" y="0" width="20" height="250"
            fill="url(#steelShadow)" id="pole"/>
    <use xlink:href="#signShape" x="0" y="0"
            fill="white" id="signWhite"/>
    <use xlink:href="#signShape" x="0" y="0"
            transform="scale(0.95)" id="signInner"/>
    <text x="0" y="0" font-size="72" font-family="Arial Narrow"
            fill="white" text-anchor="middle"
            dominant-baseline="mathematical">
            STOP
    </text>
    <use xlink:href="#signShape" x="0%" y="0%"
            filter="url(#Turb5)" id="signShade"
            mask="url(#signMask)" opacity="0.4"/>
    <use xlink:href="#signShape" x="0%" y="0%"
            fill="url(#signGrad)" id="signShade"  mask="url(#signMask)"/>
    <use xlink:href="#bolt" x="0" y="-80"/>
    <use xlink:href="#bolt" x="0" y="80"/>
</g>
<g transform="translate(150,150)" id="sign2">
    <rect x="-10" y="0" width="20" height="250"
               fill="url(#steel)" id="pole"/>
    <rect x="-10" y="0" width="20" height="250"
               fill="url(#steelShadow)" id="pole"/>
    <use xlink:href="#signShape" x="0" y="0" fill="white" id="signWhite"/>
    <use xlink:href="#signShape" x="0" y="0"
               transform="scale(0.95)" id="signInner"/>
    <text x="0" y="0" font-size="72" font-family="Arial Narrow"
               fill="white" text-anchor="middle"
               dominant-baseline="mathematical">
               GO
    </text>
    <use xlink:href="#signShape" x="0%" y="0%"
             filter="url(#Turb5)" mask="url(#signMask)"
             opacity="0.4" id="signShade"/>
    <use xlink:href="#signShape" x="0%" y="0%"
            fill="url(#signGrad)" mask="url(#signMask)"
            id="signShade"  />
    <use xlink:href="#bolt" x="0" y="-80"/>
    <use xlink:href="#bolt" x="0" y="80"/>
</g>
</defs>
<use xlink:href="#background" x="0" y="0">
    <set attributeName="xlink:href" attributeType="XML"
            to="#background2" begin="15s" fill="freeze"/>
</use>
<use xlink:href="#night" x="0" y="0" opacity="0">
    <animate attributeName="opacity" attributeType="CSS"
            values="0;0.9;0.9;0" dur="30s" fill="freeze"/>
</use>
<use xlink:href="#sign" x="0" y="0" fill="red">
    <set attributeName="xlink:href" attributeType="XML"
            to="#sign2" begin="15s" fill="freeze"/>
```

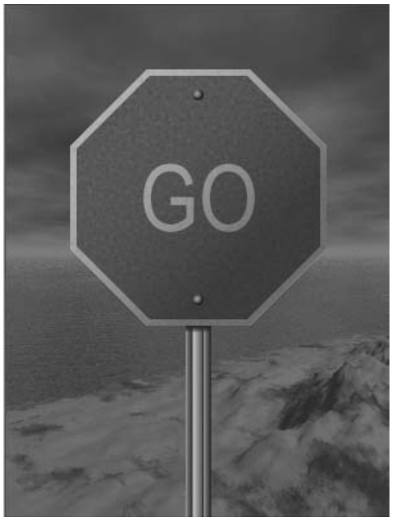


Figure 2-4. Changing opacity, text, and even background bitmaps

```
        <set attributeName="fill" attributeType="CSS"
                to="green" begin="15s" fill="freeze"/>
    </use>
    <use xlink:href="#night" x="0" y="0" opacity="0">
        <animate attributeName="opacity" attributeType="CSS"
values="0;0.9;0.9;0" dur="30s" fill="freeze"/>
    </use>
</svg>
```

There are relatively few differences between this and the previous sign in the `<defs>` section: A second sign (`#sign2`) was added that was identical to the first except that instead of the word *STOP*, the sign has the word *GO* on it. Additionally, I have defined a black rectangle entitled `#night` to handle some special effects.

The primary changes, however, come in the *rendered* portion of the graphic. Each of the visible `<use>` elements now include a set of animations that tell that element to change the value of a particular property, either continuously (such as changing the opacity of the `#night` graphic) or via discreet jumps (changing the `xlink` property in the middle of the animation sequence to point to a different graphic):

```
   <use xlink:href="#background" x="0" y="0">
        <set attributeName="xlink:href" attributeType="XML"
                to="#background2" begin="15s" fill="freeze"/>
   </use>
```

```
<use xlink:href="#night" x="0" y="0" opacity="0">
    <animate attributeName="opacity" attributeType="CSS"
            values="0;0.9;0.9;0" dur="30s" fill="freeze"/>
</use>
<use xlink:href="#sign" x="0" y="0" fill="red">
    <set attributeName="xlink:href" attributeType="XML"
            to="#sign2" begin="15s" fill="freeze"/>
    <set attributeName="fill" attributeType="CSS"
            to="green" begin="15s" fill="freeze"/>
</use>
<use xlink:href="#night" x="0" y="0" opacity="0">
    <animate attributeName="opacity" attributeType="CSS"
            values="0;0.9;0.9;0" dur="30s" fill="freeze"/>
</use>
```

The animation in the `<use>` element that references `#background` changes the background 15 seconds after the document is loaded. The `#night` referencing element (essentially a black rectangle that sits behind the sign but in front of the background) fades up from being completely transparent to nearly completely opaque. At 15 seconds after the show begins, the sign changes color from red to green while simultaneously changing the referenced text from *STOP* to *GO*. Finally, another `#night` layer sits on top of the sign itself, providing a different level of "sunset" effects for the sign than for the background.

NOTE *Although the opacity patterns are the same in both cases, the opacities multiply for the background. In other words, at the darkest point, the sign is only 90-percent opaque, while the background becomes 99-percent opaque [100%-(100%–90%)*(100%–90%)=99%].*

Generating a Graph

You can often overlook the fact that SVG is an XML format when you are using it to create "artistic" pieces, but it is in fact a powerful reason for exploring SVG as a way of creating *information graphics*. Graphs, charts, and maps all lend themselves to *auto-generation*, or using an XML data source to drive the way that such a map gets displayed and then using XML Stylesheet Language for Transformations (XSLT) to actually performing the mapping of the data into the SVG structure. Certainly you can use XSLT to handle translation to other graphics formats (such as PostScript), but the difficulty is that XSLT is really optimized for working with XML (or XML-like languages such as HTML). The mix of XML, SVG, and XSLT thus makes for a potent combination in the development of information graphics that can change in a timely fashion.

For example, consider a graph that displays traffic volume for an Internet service. The purpose of the graph is to determine, at any given time of day, whether the volume is extraordinarily higher or lower than the average for that time. This graph displays the activity as two bars: a blue bar that shows the current activity and an overlaying red bar (slightly transparent) that shows the expected average for that time of day. Significant discrepancies between the two will show up clearly in such a graph (see Figure 2-5).

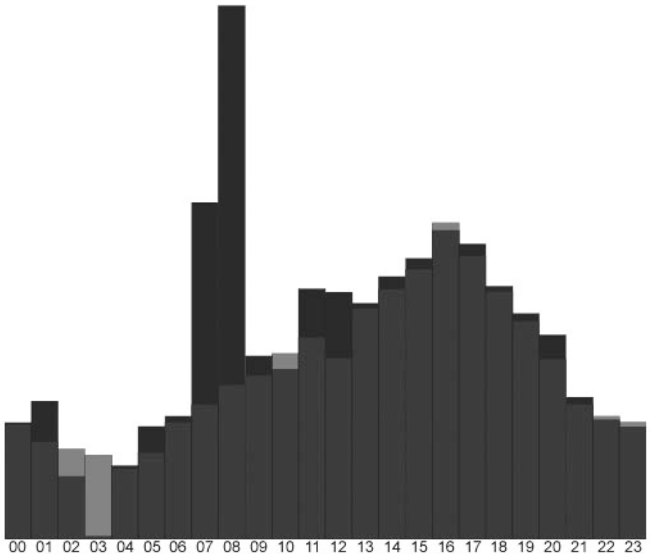


Figure 2-5. Creating an information graphic

To create an information graphic, follow these steps:

1. Create the following XML file (or load it from the Web site at `www.kurtcagle.net` or the Apress site at `www.apress.com`) and save it as **activity.xml**:

```
<activity>
    <report time="2002-04-05T00:00:00" current="1824" average="1792"/>
    <report time="2002-04-05T01:00:00" current="2159" average="1524"/>
    <report time="2002-04-05T02:00:00" current="981" average="1412"/>
    <report time="2002-04-05T03:00:00" current="52" average="1316"/>
    <report time="2002-04-05T04:00:00" current="1152" average="1102"/>
    <report time="2002-04-05T05:00:00" current="1762" average="1355"/>
    <report time="2002-04-05T06:00:00" current="1922" average="1825"/>
    <report time="2002-04-05T07:00:00" current="5255" average="2101"/>
    <report time="2002-04-05T08:00:00" current="7332" average="2411"/>
    <report time="2002-04-05T09:00:00" current="2864" average="2557"/>
    <report time="2002-04-05T10:00:00" current="2655" average="2903"/>
    <report time="2002-04-05T11:00:00" current="3912" average="3152"/>
    <report time="2002-04-05T12:00:00" current="3855" average="2825"/>
    <report time="2002-04-05T13:00:00" current="3688" average="3604"/>
    <report time="2002-04-05T14:00:00" current="4102" average="3902"/>
    <report time="2002-04-05T15:00:00" current="4387" average="4217"/>
    <report time="2002-04-05T16:00:00" current="4822" average="4943"/>
    <report time="2002-04-05T17:00:00" current="4611" average="4421"/>
    <report time="2002-04-05T18:00:00" current="3951" average="3871"/>
    <report time="2002-04-05T19:00:00" current="3527" average="3402"/>
    <report time="2002-04-05T20:00:00" current="3194" average="2814"/>
    <report time="2002-04-05T21:00:00" current="2219" average="2107"/>
    <report time="2002-04-05T22:00:00" current="1865" average="1924"/>
    <report time="2002-04-05T23:00:00" current="1755" average="1825"/>
</activity>
```

2. For the next part, you will need a command-line XSLT processor. Although there are several you can use, I like the Saxon Java-based parser (specifically, Saxon7), which you can download from Michael Kay's Saxon project on Source Forge (`saxon.sourceforge.org`). To use Saxon, you also need to have a version of Java, generally 1.2 or above (I use 1.3.1 in these samples).

3. Enter the following stylesheet, or download it from `www.kurtcagle.net`:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns="http://www.w3.org/2000/svg"
    version="1.0">
    <xsl:output method="xml" media-type="image/svg-xml"
             omit-xml-declaration="yes" indent="yes"/>
    <xsl:variable name="maxValue">
        <xsl:call-template name="max">
            <xsl:with-param name="list"
                     select="//report/@current|//report/@average"/>
        </xsl:call-template>
    </xsl:variable>
    <xsl:template match="/">
    <xsl:apply-templates/>
</xsl:template>

<xsl:template match="activity">
    <svg>
        <g transform="translate(50,50)">
        <svg x="0" y="0" width="500" height="500"
                viewBox="0 0 1000 {$maxValue * 1.2}"
                preserveAspectRatio="none">
        <g transform="translate(0,{$maxValue * 1.2}),scale(1,-1)">
        <xsl:apply-templates select="report"/>
        </g>
        </svg>
        <g transform="translate(0,510)">
            <xsl:apply-templates select="report" mode="labels"/>
        </g>
        </g>
    </svg>
</xsl:template>

<xsl:template match="report">
    <rect
          x="{format-number((position() - 1)
           *1000 div count(//report),'0.0')}"
           y="0"
           width="{format-number(1000 div count(//report),'0.0')}"
           height="{@current}"
           fill="blue" stroke="black" stroke-width="2"/>
    <rect
           x="{format-number((position() - 1)
           *1000 div count(//report),'0.0')}"
           y="0"
           width="{format-number(1000 div count(//report),'0.0')}"
           height="{@average}"
           fill="red" stroke="black"
           stroke-width="2" opacity="0.7"/>
</xsl:template>

<xsl:template match="report" mode="labels">
    <text
            x="{format-number((position() - 0.5)
            * 500 div count(//report),'0')}"
            y="0"
```

```
                              text-anchor="middle">
                                      <xsl:value-of select="substring(@time,12,2)"/>
                         </text>
                    </xsl:template>

                    <xsl:template name="min">
                         <xsl:param name="list"/>
                         <xsl:variable name="minValue">
                         <xsl:for-each select="$list">
                              <xsl:sort select="." order="ascending" data-type="number"/>
                              <xsl:if test="position()=1">
                              <item><xsl:value-of select="number(.)"/></item>
                              </xsl:if>
                         </xsl:for-each>
                         </xsl:variable>
                         <xsl:value-of select="$minValue"/>
                    </xsl:template>

                    <xsl:template name="max">
                         <xsl:param name="list"/>
                         <xsl:variable name="maxValue">
                         <xsl:for-each select="$list">
                              <xsl:sort select="." order="descending" data-type="number"/>
                              <xsl:if test="position()=1">
                              <item><xsl:value-of select="number(.)"/></item>
                              </xsl:if>
                         </xsl:for-each>
                         </xsl:variable>
                         <xsl:value-of select="$maxValue"/>
                    </xsl:template>


               </xsl:stylesheet>
```

4. Apply the stylesheet to the data via a command-line call, saving the file as **activity.svg** (you will need to change the location given to reflect the path to your XSLT processor):

```
c:\> java -jar c:\winnt\saxon7.jar activity.xml
                              showActivity.xsl > activity.svg
```

5. View the results in Listing 2-4 in Internet Explorer or Mozilla (the result should look like Figure 2-4).

Listing 2-4. Resulting SVG Output (Reformatted for Legibility)

```
<svg xmlns="http://www.w3.org/2000/svg">
   <g transform="translate(50,50)">
      <svg x="0" y="0" width="500" height="500"
                 viewBox="0 0 1000 9998.4"
                 preserveAspectRatio="none">
         <g transform="translate(0,9998.4),scale(1,-1)">
            <rect x="0.0" y="0" width="41.7" height="1824"
                    fill="blue" stroke="black" stroke-width="2"/>
            <rect x="0.0" y="0" width="41.7" height="1792"
                    fill="red" stroke="black" stroke-width="2" opacity="0.7"/>
            <rect x="41.7" y="0" width="41.7" height="2159"
                    fill="blue" stroke="black" stroke-width="2"/>
            <rect x="41.7" y="0" width="41.7" height="1524"
                    fill="red" stroke="black" stroke-width="2" opacity="0.7"/>
            <rect x="83.3" y="0" width="41.7" height="981"
                    fill="blue" stroke="black" stroke-width="2"/>
            <rect x="83.3" y="0" width="41.7" height="1412"
                    fill="red" stroke="black" stroke-width="2" opacity="0.7"/>
            <rect x="125.0" y="0" width="41.7" height="52"
                    fill="blue" stroke="black" stroke-width="2"/>
            <rect x="125.0" y="0" width="41.7" height="1316"
                    fill="red" stroke="black" stroke-width="2" opacity="0.7"/>
            <rect x="166.7" y="0" width="41.7" height="1152"
                    fill="blue" stroke="black" stroke-width="2"/>
            <rect x="166.7" y="0" width="41.7" height="1102"
                    fill="red" stroke="black" stroke-width="2" opacity="0.7"/>
            <rect x="208.3" y="0" width="41.7" height="1762"
                    fill="blue" stroke="black" stroke-width="2"/>
            <rect x="208.3" y="0" width="41.7" height="1355"
                    fill="red" stroke="black" stroke-width="2" opacity="0.7"/>
            <rect x="250.0" y="0" width="41.7" height="1922"
                    fill="blue" stroke="black" stroke-width="2"/>
            <rect x="250.0" y="0" width="41.7" height="1825"
                    fill="red" stroke="black" stroke-width="2" opacity="0.7"/>
            <rect x="291.7" y="0" width="41.7" height="5255"
                    fill="blue" stroke="black" stroke-width="2"/>
            <rect x="291.7" y="0" width="41.7" height="2101"
                    fill="red" stroke="black" stroke-width="2" opacity="0.7"/>
```

```
<rect x="333.3" y="0" width="41.7" height="8332"
        fill="blue" stroke="black" stroke-width="2"/>
<rect x="333.3" y="0" width="41.7" height="2411"
        fill="red" stroke="black" stroke-width="2" opacity="0.7"/>
<rect x="375.0" y="0" width="41.7" height="2864"
        fill="blue" stroke="black" stroke-width="2"/>
<rect x="375.0" y="0" width="41.7" height="2557"
        fill="red" stroke="black" stroke-width="2" opacity="0.7"/>
<rect x="416.7" y="0" width="41.7" height="2655"
        fill="blue" stroke="black" stroke-width="2"/>
<rect x="416.7" y="0" width="41.7" height="2903"
        fill="red" stroke="black" stroke-width="2" opacity="0.7"/>
<rect x="458.3" y="0" width="41.7" height="3912"
        fill="blue" stroke="black" stroke-width="2"/>
<rect x="458.3" y="0" width="41.7" height="3152"
        fill="red" stroke="black" stroke-width="2" opacity="0.7"/>
<rect x="500.0" y="0" width="41.7" height="3855"
        fill="blue" stroke="black" stroke-width="2"/>
<rect x="500.0" y="0" width="41.7" height="2825"
        fill="red" stroke="black" stroke-width="2" opacity="0.7"/>
<rect x="541.7" y="0" width="41.7" height="3688"
        fill="blue" stroke="black" stroke-width="2"/>
<rect x="541.7" y="0" width="41.7" height="3604"
        fill="red" stroke="black" stroke-width="2" opacity="0.7"/>
<rect x="583.3" y="0" width="41.7" height="4102"
        fill="blue" stroke="black" stroke-width="2"/>
<rect x="583.3" y="0" width="41.7" height="3902"
        fill="red" stroke="black" stroke-width="2" opacity="0.7"/>
<rect x="625.0" y="0" width="41.7" height="4387"
        fill="blue" stroke="black" stroke-width="2"/>
<rect x="625.0" y="0" width="41.7" height="4217"
        fill="red" stroke="black" stroke-width="2" opacity="0.7"/>
<rect x="666.7" y="0" width="41.7" height="4822"
        fill="blue" stroke="black" stroke-width="2"/>
<rect x="666.7" y="0" width="41.7" height="4943"
        fill="red" stroke="black" stroke-width="2" opacity="0.7"/>
<rect x="708.3" y="0" width="41.7" height="4611"
        fill="blue" stroke="black" stroke-width="2"/>
<rect x="708.3" y="0" width="41.7" height="4421"
        fill="red" stroke="black" stroke-width="2" opacity="0.7"/>
<rect x="750.0" y="0" width="41.7" height="3951"
        fill="blue" stroke="black" stroke-width="2"/>
<rect x="750.0" y="0" width="41.7" height="3871"
        fill="red" stroke="black" stroke-width="2" opacity="0.7"/>
```

```
      <rect x="791.7" y="0" width="41.7" height="3527"
              fill="blue" stroke="black" stroke-width="2"/>
      <rect x="791.7" y="0" width="41.7" height="3402"
              fill="red" stroke="black" stroke-width="2" opacity="0.7"/>
      <rect x="833.3" y="0" width="41.7" height="3194"
              fill="blue" stroke="black" stroke-width="2"/>
      <rect x="833.3" y="0" width="41.7" height="2814"
              fill="red" stroke="black" stroke-width="2" opacity="0.7"/>
      <rect x="875.0" y="0" width="41.7" height="2219"
              fill="blue" stroke="black" stroke-width="2"/>
      <rect x="875.0" y="0" width="41.7" height="2107"
              fill="red" stroke="black" stroke-width="2" opacity="0.7"/>
      <rect x="916.7" y="0" width="41.7" height="1865"
              fill="blue" stroke="black" stroke-width="2"/>
      <rect x="916.7" y="0" width="41.7" height="1924"
              fill="red" stroke="black" stroke-width="2" opacity="0.7"/>
      <rect x="958.3" y="0" width="41.7" height="1755"
              fill="blue" stroke="black" stroke-width="2"/>
      <rect x="958.3" y="0" width="41.7" height="1825"
              fill="red" stroke="black" stroke-width="2" opacity="0.7"/>
   </g>
</svg>
<g transform="translate(0,510)">
   <text x="10" y="0" text-anchor="middle">00</text>
   <text x="31" y="0" text-anchor="middle">01</text>
   <text x="52" y="0" text-anchor="middle">02</text>
   <text x="73" y="0" text-anchor="middle">03</text>
   <text x="94" y="0" text-anchor="middle">04</text>
   <text x="115" y="0" text-anchor="middle">05</text>
   <text x="135" y="0" text-anchor="middle">06</text>
   <text x="156" y="0" text-anchor="middle">07</text>
   <text x="177" y="0" text-anchor="middle">08</text>
   <text x="198" y="0" text-anchor="middle">09</text>
   <text x="219" y="0" text-anchor="middle">10</text>
   <text x="240" y="0" text-anchor="middle">11</text>
   <text x="260" y="0" text-anchor="middle">12</text>
   <text x="281" y="0" text-anchor="middle">13</text>
   <text x="302" y="0" text-anchor="middle">14</text>
   <text x="323" y="0" text-anchor="middle">15</text>
   <text x="344" y="0" text-anchor="middle">16</text>
   <text x="365" y="0" text-anchor="middle">17</text>
   <text x="385" y="0" text-anchor="middle">18</text>
   <text x="406" y="0" text-anchor="middle">19</text>
```

```
            <text x="427" y="0" text-anchor="middle">20</text>
            <text x="448" y="0" text-anchor="middle">21</text>
            <text x="469" y="0" text-anchor="middle">22</text>
            <text x="490" y="0" text-anchor="middle">23</text>
         </g>
      </g>
</svg>
```

To understand what is going on in this example, it is probably best to start from the final SVG document and work backward. In `activity.svg`, the structure is divided into two distinct sections. The first contains the graph itself, with no text contents. For ease of display, this particular code shifts the graphic down and to the right by 50 pixels through the use of a translation transform on a containing `<g>` element:

```
<svg xmlns="http://www.w3.org/2000/svg">
   <g transform="translate(50,50)">
```

In the second section, the graph in turn takes advantage of “local” coordinate systems. It makes the assumption that the “interior” height of the graph (in other words, the height as seen by the graph elements themselves) is sized to be about 20-percent taller than the height of the highest element. This ensures that the graph will always show all of the data for each time period, though it does so at the cost of no longer being consistent in scale. The external dimensions, on the other hand, are set to be a consistent 500x500 pixels. This automatically forces the page to rescale itself to fit in that dimension, regardless of the dimensions used within the SVG:

```
<svg x="0" y="0" width="500" height="500"
    viewBox="0 0 1000 9998.4" preserveAspectRatio="none">
```

The real goal in working with SVG is to try to do as little work in computation as possible. One of the problems in creating graphs is the fact that the coordinate system that SVG uses (one in which vertical coordinates go *down* the page rather than up) are well suited to computer graphics but not at all suited to most charts (which work in the reverse direction). Consequently, rather than trying to calculate the difference in height between the top of the document and the baseline—a pretty complicated undertaking—I simply draw the boxes down from a given baseline and then flip the entire picture along the horizontal axis with the command scale `(1,-1)`. Once this happens, the graphic is now *above* the top of the graphic, so I have to move it (translate) back down by the generated height of the graph:

```
<g transform="translate(0,9998.4),scale(1,-1)">
```

The graphic rectangles themselves are actually straightforward. The width is calculated so that however many you have in the sample (in this case, 24), the total will always add up to 500, give or take a few pixels from rounding. Each graph point consists of two rectangles, one blue and solid, the other red and 70-percent opaque. The opacity makes it easy to overlay one over the other to see where the greatest discrepancies are between the average and the actual value:

```
<rect x="0.0" y="0" width="41.7" height="1824"
                      fill="blue" stroke="black" stroke-width="2"/>
<rect x="0.0" y="0" width="41.7" height="1792"
                      fill="red" stroke="black" stroke-width="2" opacity="0.7"/>
<rect x="41.7" y="0" width="41.7" height="2159"
                      fill="blue" stroke="black" stroke-width="2"/>
<rect x="41.7" y="0" width="41.7" height="1524"
                      fill="red" stroke="black" stroke-width="2" opacity="0.7"/>
<!-- more along this line -->
```

I deliberately separated out the labels from the rest of the graph because of the reflection and translation issue; drawing the text into the graph would have made the letters appear upside down in the final output. Instead, a new <g> element was defined to move the starting positions of the letters to just under the graph, and each text item was positioned to be centered under their corresponding block. Note that the hours are given in military time, ranging from 00 for midnight to 23 for 11 p.m. Not only is this the easiest way to generate this information (in essence, reading it from the activity timestamps), but it also keeps the labels both big enough to see and small enough to avoid overlap.
For example:

```
      <g transform="translate(0,510)">
         <text x="10" y="0" text-anchor="middle">00</text>
         <text x="31" y="0" text-anchor="middle">01</text>
         <text x="52" y="0" text-anchor="middle">02</text>
         <text x="73" y="0" text-anchor="middle">03</text>
            <!-- more along this line -->
```

That is it. You could do more with this graph, of course—making the elements dynamic, including coordinate grids, and so forth, but the principle in working with those elements is essentially the same as shown in this section.

Examining the XSLT

The XSLT to generate such a graph is fairly simple, though it can be daunting if you have not worked a lot with XSLT in the past. Because XSLT is somewhat outside of the immediate scope of this chapter, I want to just briefly discuss what each template does.

The initial header creates the stylesheet namespaces and also identifies the namespace for SVG, something that is reinforced by the `<xsl:output>` object that defines the `method` attribute (which describes the overall structural language) as being `xml` and the media-type as being `images/svg-xml`:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns="http://www.w3.org/2000/svg"
    version="1.0">
    <xsl:output method="xml" media-type="image/svg-xml"
        omit-xml-declaration="yes" indent="yes"/>
```

Perhaps the only really flashy part of the XSLT transformation involves the generation of the maximum value from the data sets. Because the maximum could in fact come from either the current timestamps (an extraordinarily high-hit response) or from the average timestamp for that period (if, for instance, a denial of service attack hit the server for more than a day), the maximum value needs to be calculated from the union of the two sets and is then assigned into the variable `$maxValue`:

```
    <xsl:variable name="maxValue">
        <xsl:call-template name="max">
            <xsl:with-param name="list"
                        select="//report/@current|//report/@average"/>
        </xsl:call-template>
    </xsl:variable>
```

NOTE *XSLT variables start with the dollar (*$*) sign except at the time of their definition, and I use the same notation to refer to them within the text body.*

The routine that actually does the sorting, well, er . . . cheats. It takes the list of nodes in the XML source, then uses the `<xsl:sort>` attribute on an `<xsl:for-each>` to automatically order the nodes by the requisite attributes.

By indicating whether the sorted list is ascending or descending in order, you can then be sure that the first element will always be either the lowest or highest value in the list. For example:

```
<xsl:template name="min">
    <xsl:param name="list"/>
    <xsl:variable name="minValue">
    <xsl:for-each select="$list">
        <xsl:sort select="." order="ascending" data-type="number"/>
        <xsl:if test="position()=1">
        <item><xsl:value-of select="number(.)"/></item>
        </xsl:if>
    </xsl:for-each>
    </xsl:variable>
    <xsl:value-of select="$minValue"/>
</xsl:template>
```

The root template match passes the processing to the next node, `activity`, which lays out the initial SVG document structure, then processes the records twice—first to generate the graph, then to generate the labels. Note the use of bracketed XPath evaluations to calculate the height and offset of the graph region:

```
<xsl:template match="/">
    <xsl:apply-templates/>
</xsl:template>

<xsl:template match="activity">
    <svg>
        <g transform="translate(50,50)">
        <svg x="0" y="0" width="500" height="500"
                    viewBox="0 0 1000 {$maxValue * 1.2}"
                    preserveAspectRatio="none">
        <g transform="translate(0,{$maxValue * 1.2}),scale(1,-1)">
        <xsl:apply-templates select="report" mode="graph"/>
        </g>
        </svg>
        <g transform="translate(0,510)">
            <xsl:apply-templates select="report" mode="labels"/>
        </g>
        </g>
    </svg>
</xsl:template>
```

The report template, graph mode, is where the heavy lifting is done. Normally, calculations done on numbers return those numbers with a mantissa of eight significant figures. Unfortunately, although more accurate, this makes things much harder to read; thus, most of the calculations are cleaned up with the format-number XPath function:

```
<xsl:template match="report" mode="graph">
    <rect
           x="{format-number((position() - 1)*1000 div count(//report),'0.0')}"
           y="0"
           width="{format-number(1000 div count(//report),'0.0')}"
           height="{@current}"
           fill="blue" stroke="black" stroke-width="2"/>
    <rect
           x="{format-number((position() - 1)
           *1000 div count(//report),'0.0')}"
           y="0"
           width="{format-number(1000 div count(//report),'0.0')}"
           height="{@average}"
           fill="red" stroke="black" stroke-width="2"
           opacity="0.7"/>
</xsl:template>
```

The label generator is in fact a fairly simple variant of the rectangle drawing routine. The positioning mechanism is nearly identical, save for a half-bar offset to ensure that the text is centered on the bar's midpoint rather than on the bar's leftmost point. The routine that retrieves the hour extracts the hour field from the `@time` attribute for each report (as `substring(@time,12,2)`):

```
<xsl:template match="report" mode="labels">
    <text
           x="{format-number((position() - 0.5)
           * 500 div count(//report),'0')}"
           y="0"
           text-anchor="middle">
                 <xsl:value-of select="substring(@time,12,2)"/>
    </text>
</xsl:template>
```

That is pretty much it. As with XSLT in general, the trick to using SVG with XSLT is to pass to each language the task that they are best suited for—in other words, SVG for rendering, changing graphical context, and defining entities, and XSLT for positioning, identifying pieces, building core objects, and performing conditional logic.

Summary

The tutorials in this chapter were intended to give you a blitz of what real SVG documents look like, and a great deal of information was glossed over in the interest of trying to pull together a sampler of what you can do with the language.

The next several chapters look at the various pieces of the SVG puzzle: coordinate systems, shapes, fills, strokes, images, and text, as well as animation, interactivity, the DOM, and then ultimately back to XSLT again. If you have trouble understanding this chapter, I recommend you come back here after each chapter and see how later explanations relate to this content.

Learning is seldom a linear process, no matter how much you may try to make it so. Instead, knowledge and wisdom come only when a solid foundation exists upon which to build.