SVG Programming: The Graphical Web

KURT CAGLE

Apress™

SVG Programming: The Graphical Web

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 1

Why SVG?

THE FIRST WORDS that you, as the reader, see are in many respects the most important words in the book. If written well, they invite you in, motivating you to undertake the sometimes arduous task of learning, which any such book in fact requires. On the other hand, when written poorly, these same words can dissuade you, perhaps to the extent of not even purchasing the book. Hence, I must write these words with care.

This book is about pictures . . . and words. This includes not only the words that appear in presentations, alongside graphs, or on buttons but also the words used to describe and build these other words and graphics—the "programming" words, if you will.

The axiom *A picture is worth a thousand words,* when coined, was meant to illustrate the principle that we humans are fundamentally visual rather than literary beings. Yet with Scalable Vector Graphics (SVG), this simple thought takes on a life of its own, as we use words to generate the very pictures we are using to illustrate. SVG is essentially a language: It's a way of using the particular grammar imposed by Extensible Markup Language (XML) to draw pictures (and words), animate them, make them responsive to external events, generate new SVG on the fly, and interact with other languages and environments.

This is a book about SVG, but it is also about a concept beginning to gain some currency even in the workaday realm of commercial software: the idea that perhaps the real nature of programming is not to find the most efficacious way of optimizing the limited set of programming languages to accomplish a task, but rather to see language itself as a medium to be manipulated. In other words, it is not so much about programming with a language as it is about programming our concept of language.

Most people, when first encountering SVG, will try to judge it against existing applications. SVG vs. Adobe PostScript, SVG vs. Macromedia Flash. SVG vs. Microsoft PowerPoint. All of these are valid comparisons, but to view SVG this way is to miss its true applicability. Ultimately, SVG is a standard way to describe graphics and graphical interactions; it's one particular dialect of the incredibly rich family of XML technologies. It's a way of encoding a visual image that can be transported as a single unit or in pieces and a way of describing interfaces that can be built at a moment's notice, existing only so long as it's needed and then melting away in transformation.

Conceptually, this may be a hard concept to wrap your brain around because for 50 years the concept of interfaces has been (and continues to be) oriented around the idea that an interface is built. SVG is one piece of a dramatic change in the way you develop applications, use computers, work with networks, and otherwise deal with these marvelously sophisticated communication devices. Yet increasingly, these open, distributed systems tend to evolve their own priorities and protocols in response to needs over time.

Software vendors have derided Hypertext Markup Language (HTML) as being extremely simplistic and insufficient to the task of building the complex tools needed for binding communities, ensuring electronic commerce, and policing the resultant infrastructure. The irony is that HTML is still (by a wide margin) the single most widely used computer language, with several billion pieces of HTML "software" being deployed every day. Meanwhile, most vendor-driven applications have been rendered irrelevant (along with, in many cases, the vendors themselves) just a meager 10 years from the time HTML first appeared on the scene.

You can make similar arguments for other open protocols, languages, and associated tools. For instance, the most widely used Web server on the planet (by about a five-to-two margin) is the Apache Web server. Additionally, the Post Office Protocol/Simple Mail Transfer Protocol (POP3/SMTP) e-mail has not completely eliminated alternative mail systems, but there are few proprietary mail formats on the planet that don't at least have a gateway for converting mail to the open standard; otherwise, these service providers would cease to be relevant.

For that matter, the prevalent infrastructure protocols such as the Transport Control and Internet Protocols (the ubiquitous TCP/IP) have so completely rendered other network protocols obsolete that meaningful development of proprietary network protocols has all but ground to a halt. The only protocol likely to challenge TCP/IP in the long term is Block Extensible Exchange Protocol (BEEP) format, perhaps with a Simple Object Access Protocol (SOAP) layer sitting on top of it.

As I write this, SVG is roughly one year old. It has already engendered fierce controversy as well as a host of naysayers who have a vested interest in seeing that it does not succeed. In this chapter, I look at the 10,000-foot view of SVG, especially how it relates to other known protocols (including the de facto protocol PostScript) and products such as Flash. Additionally, much of this book (as much of XML itself) revolves around the notion of context. To understand where SVG is going, it doesn't hurt to understand its history. With that in mind, let's get started.

A Short History of XML and SVG

XML has sprung into existence seemingly overnight, but this appearance is deceptive. XML has, much like the Internet, gestated under a number of different forms since the late 1960s. During that decade, Charles Goldfarb, Edward Mosher, and Raymond Lorie of IBM faced a conundrum: how to effectively store electronic documentation. The solution they came up with soon came to be known as General Markup Language (GML), which was a way of providing a logically cohesive structure to a document that could in turn provide metadata about that document.

GML proved to be something of a hit in the early 1970s, and others began to provide their own implementations of GML. Unfortunately, although there was a great deal of similarity between these various markup languages, there were also a sufficient number of differences to make building tools for GML next to impossible. As a consequence, the adopters of this technology agreed to create a standard, called Standardized Generalized Markup Language (SGML), which was adopted by the United Nations (through the International Standards Organization) as a key standard for markup languages worldwide in the early 1980s.

SGML was an extremely powerful way of expressing a wide range of documents, but the growth and fracturing of SGML meant that the standard was also so sufficiently complex that it was difficult to implement on any but the largest machines. SGML consequently went into a decline after becoming a standard; it was still widely used but with far less activity going toward improving it during the period of intense growth in the computer industry between 1980 and 1992.

Examining PostScript

Another standard emerged in that same interval, however: PostScript. This language is a page-description language, or a theoretically human-readable standard for describing graphical content, including both pictures and the letters of words. Adobe Systems created PostScript in the early 1980s as a way for printers to universally describe how a page looked, and it was such a success that PostScript-based printers currently dominate the printer market.

It's worth examining PostScript in some detail because PostScript provided an interesting paradigm that has made its way into SVG: the use of plain text to describe graphical information. For example, Listing 1-1 shows a PostScript document (generated for this particular chapter out of Open Office). The document is legible, but it looks far more like a programming language than a way of linguistically describing a page.

Listing 1-1. A PostScript Document

```
%!PS-Adobe-3.0
%%BoundingBox: 0 0 595 842
%%Creator: OpenOffice.org 641
%%For: seatails
%%CreationDate: Mon Apr 29 19:46:28 2002
%%Title: xmlSVGHistory.svg.sxw
%%LanguageLevel: 2
%%DocumentData: Clean7Bit
%%Pages: (atend)
%%PageOrder: Ascend
%%EndComments
%%BeginProlog
/ISO1252Encoding [
/psp_definefont { exch dup findfont dup length dict begin { 1 index
/FID ne
{ def } { pop pop } ifelse } forall /Encoding 3 -1 roll def
currentdict end exch pop definefont pop } def

/pathdict dup 8 dict def load begin
/rcmd { { currentfile 1 string readstring pop 0 get dup 32 gt { exit }
{ pop } ifelse } loop dup 126 eq { pop exit } if 65 sub dup 16#3 and 1
add exch dup 16#C and -2 bitshift 16#3 and 1 add exch 16#10 and 16#10
eq 3 1 roll exch } def
/rhex { dup 1 sub exch currentfile exch string readhexstring pop dup 0
get dup 16#80 and 16#80 eq dup 3 1 roll { 16#7f and } if 2 index 0 3
-1 roll put 3 1 roll 0 0 1 5 -1 roll { 2 index exch get add 256 mul }
for 256 div exch pop exch { neg } if } def
/xcmd { rcmd exch rhex exch rhex exch 5 -1 roll add exch 4 -1 roll add
1 index 1 index 5 -1 roll { moveto } { lineto } ifelse } def end
/readpath { 0 0 pathdict begin { xcmd } loop end pop pop } def

systemdict /languagelevel known not {
/xshow { exch dup length 0 1 3 -1 roll 1 sub { dup 3 index exch get
exch 2 index exch get 1 string dup 0 4 -1 roll put currentpoint 3 -1
roll show moveto 0 rmoveto } for pop pop } def
/rectangle { 4 -2 roll moveto 1 index 0 rlineto 0 exch rlineto neg 0
rlineto closepath } def
/rectfill { rectangle fill } def
/rectstroke { rectangle stroke } def } if
%%Trailer
%%Pages: 1
%%EOF
```

Of course, plain text does not always mean that the text is intelligible. To describe graphics, PostScript uses Reverse Polish Notation, a language especially well suited for programming structures called *stacks*. A stack is probably familiar to you in the context of a bunch of cafeteria trays sitting on springs—as a tray is added, the weight of the tray pushes the stack down, and pulling (or popping) the topmost tray off the stack pushes the remaining trays back up.

Stacks are remarkably efficient ways of handling any number of programming problems, but they have long been central to programming graphics. In a stack-based graphic system, when you want to draw something, you create a *graphical context* (think of it as one of those trays) and perform what drawing you need. At some point, you may need to change the context—perhaps the coordinate system (or the way the coordinate system is oriented) or some other property for that specific context. So, you create a *new* graphical context and push the old context onto the stack until you're done. When finished, you remove the active context and return to the previous coordinate system, variables, and so forth in the older context.

One advantage of such an approach is that you can essentially add as many contexts onto the stack as you need to perform short-term calculations, without having to spend a lot of time dealing with state maintenance. This was the approach that PostScript first used (and ultimately is the approach that SVG uses two decades later).

Another characteristic of such a stack-based graphical system is that you can describe how to draw an entire document by simply keeping track of what happens when a stack is added or removed. You can think of a PostScript document as (more or less) the history of creating the document in the first place, optimized somewhat to take into account changes that were made but then undone. In other words, PostScript is also a *declarative* language; you are describing the entire state transition set that the drawing has gone through, correlating time with a position in the document. Declarative programming differs from *procedural programming*, which makes up the bulk of what most people today consider "real" programming languages: C, C++, Java, Python, and so on.

NOTE *I'd throw Perl into that mix, too, but Perl is so heavily built upon regular expressions—another form of declarative (also known as* functional*) programming—that you can think of Perl as having a foot in both worlds.*

By the way, in comparison to PostScript, Listing 1-2 shows a short SVG program (simplified a little).

Listing 1-2. A Simplified SVG Program

```
<svg width="1024" height="768">
    <defs>
        <linearGradient id="backgroundGradient" gradientTransform="rotate(90)">
            <stop offset="0%" stop-color="blue"/>
            <stop offset="100%" stop-color="navy"/>
        </linearGradient>
        <rect x="0" y="0' width="100%" height="100%" id="background"
                fill="url(#backgroundGradient)"/>
        <text dominant-baseline="mathematical" text-anchor="center" x="50%"
                y="50%" fill="yellow" id="helloWorld">
                Hello, SVG World!
        </text>
    </defs>
    <g>
        <use xlink:href="#background"/>
        <use xlink:href="#helloWorld"/>
    </g>
</svg>
```

The stack model still exists here, but it is hidden within the container/contained paradigm of XML. A `<defs>` section includes defined objects made available for later use by other entities. Graphical contexts are pushed on the stack to draw things and then released, hidden within the abstraction of the `<use>` element. An SVG document involves the creation of a tiny little world, for the most part a self-contained one, with the added benefit of persistence (that is to say, when you define an object, that object stays defined over the scope of the document). It's a powerful paradigm and is consistent with the surprisingly rich world of both HTML and XML.

The Rise of Web Markup Languages

The creation of HTML is so well documented that most grade-school children are aware of it, but I want to look at HTML as it relates to SVG and at the somewhat lesser-known underside of HTML. The original HTML that Tim Berners-Lee wrote was an extremely limited language, with perhaps a dozen tags or so. Most of the staples you think of as "crucial" to HTML, in fact, didn't even exist; form elements, tables, images, scripting blocks, and event handlers were not a part of the initial draft that Berners-Lee wrote back in the late 1980s.

Moreover, although people said HTML was written in SGML, that's actually a bit of revisionist history. Berners-Lee was attempting to solve a problem: how to make physics abstracts (summaries of articles) available to the researchers at the Center for European Nuclear Research (CERN) in Switzerland, which was where Berners-Lee worked when developing the HTTP and HTML specifications. He had encountered SGML and developed an SGML-like solution to what he perceived to be a small-scale problem. What he hadn't counted on was how quickly HTML would take off.

Meanwhile, in the United States, the U.S. Defense Advanced Research Projects Agency (DARPA) had been quietly seeding the nascent Arpanet (the precursor of the Internet) with projects to utilize SGML as a way of describing more generic entities than simple documents. In fact, DARPA was a major contributor to the National Centers for Supercomputing Applications research laboratories across the country, including the one at the University of Illinois that helped fund the creation of the Mosaic browser. It is entirely possible that had Berners-Lee not written HTML, an SGMLish language would have emerged soon thereafter because the research community had been working with SGML as data structures even around that time.

NOTE *This is not to discount the real contribution that Berners-Lee made to the effort, which was as much to give away Hypertext Transfer Protocol (HTTP) and its systems rather than charge for it as it was to develop the language in the first place.*

Unfortunately, the standards battles of the 1970s with respect to GML were repeated in the early 1990s with respect to HTML. As the number of applications for HTML increased exponentially, entrepreneurial-minded Internet browser vendors added new features to the HTML model, eventually getting to the point where there were nearly as many different versions of HTML as there were browsers.

In response to this, Berners-Lee founded a new organization, made up principally by those same vendors as well as a host of SGML experts who had wrestled with many of the same problems a generation earlier. The World Wide Web Consortium (W3C) took shape in 1994 and immediately set to work trying both to stabilize the still-morphing HTML and to try to solve some of the more egregious problems besetting the standard.

By 1996, the W3C had established a stylistic language called Cascading Style Sheets (CSS) that made it possible to separate the logical structure from its media representation. Additionally, the SGML community had attempted to create an alternative language that would use HTML/SGML-like syntax and sundry tools

but would be more generalized. This language, called HyTime, started out with noble goals—to build a cohesive mechanism for linking both across a network and over time—but for the most part failed to produce more than a specification for describing music in SGML terms (an effort that has been recast in XML, by the way). However, in the process, it spawned a couple of interesting child technologies. Specifically, a superb programmer named James Clark (who also appears later in the section) created a scripting language called Document Style Semantics and Specification Language (DSSSL) that you can use to perform more complex transformations on SGML-like documents.

In that same year, the W3C recognized that if HTML were to truly evolve, it would need to become far more flexible; essentially, it would need to become a meta-language itself, like SGML. The unfortunate downside of SGML, though, came from the twenty-plus years of legacy adaptations that had made writing an SGML application still an exercise requiring complex parsers (the programs that interpret the SGML to create the document structures in memory). SGML was simply too big and flexible to fit nicely on the Web. So, the W3C stripped out all but the most basic parts of SGML to create a new language: XML.

XML was originally envisioned as being a way of creating various document structures, and consequently the first XML experts were the SGML gurus who had worked so heavily with the older language. By December 1997, the W3C had published a formal XML specification, and the language emerged to . . . well, yawns. Sold originally as a way of replacing HTML (a goal that it is in fact finally beginning to do), XML was simply too abstract and document-oriented for most people to really care.

However, one of the characteristics of XML (which had been foreseen by DARPA several years earlier) was that a consistent, simple-to-use markup language could actually encode a large number of things beyond documents. For example, it could describe data structures such as those used in object-oriented programming, and it could readily describe stack structures. Programmers (myself included, at the time) saw an XML parser as a superb way of being able to describe binary trees and hierarchical data. By late 1998, the revolution had been hijacked, and XML became synonymous with data.

By late 1998, the W3C started putting together the pieces of the Web "as they should have been built." If HTML was a page-description language with links, it would be much better to have a whole language that existed to style XML documents into a specific media representation. This effort led to the creation of the XML Stylesheet Language (XSL), which in turn consisted of one part that described the specific layout, PostScript-like, called XSL and one part for styling the XSL from the XML based on DSSSL but using XML as its description language. This language was in turn called the XML Stylesheet Language for Transformations (XSLT) and was shaped in great part by the aforementioned James Clark.

Remember those data guys? The programmers who saw XML as a way of representing data? They recognized a very subtle truth: XSLT is a compiler.

XSLT takes a string (an XML document), parses it into lexical units (XML nodes), arranges those nodes in a tree structure, and then maps that tree structure onto a new string—the output stream. Compilers are nice tools to have, especially when the output of those streams could in fact be another XML document (it doesn't have to be HTML). XSLT soon ballooned out of proportion to its XSL sibling to such an extent that XSL became interchangeable with the transformation language. Belatedly, XSL became XSL-Formatting Objects (XSL-FO), which is a way of describing page content that may end up becoming more important to the printing industry than it would be to browser manufacturers.

From Stormy Beginnings, a Common Standard

1998 was a good year for XML. Not only did it see the publication of the XML specification in December 1997, but people began looking for potential needs that weren't being met by HTML but that might be met by XML. A big one loomed in the need for a vector graphics standard.

PostScript has long been the de facto standard for describing *vector-based graphics*, or graphics that use equations rather than pixels to depict images. However, as is demonstrated previously, PostScript is a little too low level; it is a language that an expert could potentially write, but in general was far too optimized for machine use to be useful in the same way that HTML can be. HTML, however, includes nothing in the way of graphics primitives for drawing, so the field was largely ceded to proprietary standards written by third-party vendors.

A few years before, a small company named Future Splash worked out a way of encoding vector graphics in a small bundle to allow for dynamic animations without the tedious download times of digital video. Future Splash became popular—so popular that in 1996 Macromedia (one of the largest established players in the multimedia field) purchased it, renaming it Flash. Flash, rendered through Macromedia's Shockwave plug-in, has become the most common piece of proprietary software on the planet.

Adobe and Macromedia have traditionally had a rivalry, and arguably they have been battling back and forth in the multimedia production and playback arena for several years. In 1998, Adobe submitted a proposal called the Precision Graphics Markup Language (PGML) to the W3C that basically described an XML-based graphics language.

That same year, Microsoft had been looking at creating a vector markup language standard to complement its efforts with the XML, XSLT, and XML Schema (a way of defining data types associating with specific XML elements or attributes) specifications currently under submission. Macromedia at the time had released the specification for Flash but because it wasn't XML-based had chosen

instead to support Microsoft's alternative: the Vector Markup Language (VML). As a proof of concept, Microsoft also created a viewer for VML that could be run as a COM component and then incorporated this viewer into Internet Explorer and Microsoft Office (where it is to this day).

The issue came to a head in the spring of 1999 within the newly formed W3C Graphics Working Group. The final consensus of the group (though one reached only after a great deal of argument) was that neither submission would be considered final. Instead, the two standards would merge into a new working draft document called the Scalable Vector Graphics language. Although both Macromedia and Microsoft were signatories to the original group and the resulting document, both companies essentially ceased to participate in the development of the standard after this point, and currently neither has provided any indication that they will support SVG in their line of products (the computer world can change overnight, however; this may not be true by the time you read this book). The SVG 1.0 Working Draft became a formal recommendation in September 2001.

Despite the apparent discord, the people and companies who participated in developing the standard is an exhaustive list of the key Internet players in the late 1990s. In addition to those mentioned earlier, other participants included Autodesk, IBM, Netscape, Apple, Sun Microsystems, Xerox, Corel, Visio, Hewlett-Packard, and Quark, as well as the XML standards organization OASIS, Oxford Brookes University, and a number of experts on graphical standards such as David Dodds, Chris Lilley, Philip Mansfield, and David Duce. This broad base of support has meant that SVG has had a chance to evolve across a number of different applications and uses even before it became widely known to the public.

The formal SVG standard is now about a year old, and its impact is already beginning to be felt. SVG is a critical part of the open-source Apache XML Project (supported heavily by IBM) and is in use with both Cocoon (the Apache Web publishing platform) and Batik (an open-source SVG component). The Adobe SVG component, which is the primary SVG platform used in this book, provides perhaps the most complete implementation of the SVG standard for multimedia playback, and graphic tool manufacturers such as Adobe, Corel, AutoDesk, and others are now incorporating it as either an import or an export format (and in most companies as both).

SVG is very much a version 1.0 technology. Like all 1.0 technologies, that means that far from being the ultimate graphics format, SVG is really just beginning to be truly field tested. Already this process has revealed a few deficiencies in the language; but as these are discovered, they are being incorporated into the SVG 1.1 specification currently underway. The working group has also proposed significant new functionality for SVG 2.0, a document that acts more as a way of defining the goals for the next generation of SVG applications. Additionally, the

W3C is likely to adopt the second versions of W3C recommendations, such as XSLT 2.0, XML Query, and XPath 2.0, within 2002 or early 2003. Further, a number of working drafts of new technologies such as XML Forms and XSL-FO either have just been recommended or are close to being so, and the W3C Graphics Working Group is coordinating closely with these other groups to ensure compatibility between the various standards.

SVG as a Piece of the Puzzle

I've been talking about SVG as a graphical format, but this is in fact somewhat misleading. You can certainly describe a graphic using SVG (indeed, almost every graphic in this book was originally generated using SVG), but it is important to realize that SVG is a language intended to work *in conjunction with* other languages, most notably the Synchronized Multimedia Integration Language (SMIL) but also scripting languages such as JavaScript, statically typed languages such as Java or C#, XSLT, XHTML (the XML version of HTML), and many others.

To that end, it's worthwhile to think of SVG as being a way of exposing graphical objects that other applications can use. For instance, SMIL is an XML-based language that in essence provides a way of correlating other languages and time- or event-based processes. The SVG animation elements work by providing an SVG structure on which to hang SMIL properties. Similarly, the SVG `<script>` element serves as a way to not only run programs within SVG documents but also to manipulate the SVG using a Document Object Model (DOM) that the SVG viewer exposes. Chapter 9, "Integrating SVG and HTML," and Chapter 10, "SVG Components," in this book also explore the fairly intimate way that SVG can work in conjunction with HTML or XHTML and how SVG can enhance HTML without replacing the benefits of using it.

Finally, XSLT is a powerful tool for both generating SVG from XML resources and for extracting key information from SVG documents. You should not overlook this particular point because it opens up a view of a world where many significant processes are actually handled through the transformation of XML data into a variety of forms, including SVG. XSLT is quietly performing a transformation on programming and application development as well, increasingly appearing as a core component in both Web and stand-alone applications, serving as a generic engine within embedded systems, and performing more of the necessary but seemingly unimportant programming that makes networked computer systems interoperate.

Consequently, when working with SVG, you should always remember that SVG by itself is far less powerful than SVG used in conjunction with something else. This is a point I hope to prove throughout this book.

Using SVG and Flash

Chances are pretty good that if you have heard anything about SVG, you will have heard about how SVG is sort of like Flash and may even be a "Flash killer."

I'll confess here, in the name of full disclosure. I worked as the Technical Editor for the *Macromedia User Journal* and am intimately familiar with both Macromedia Director and Flash applications. I personally think that Flash is a superb product and has a well-deserved reputation for producing some of the most cutting-edge multimedia on the Internet. Moreover, I think that in the area where Flash has initially been targeted, it is the dominant mechanism for both creating and displaying multimedia far and away.

So why am I not writing a book about the next version of Flash? There are actually any number of reasons, but most of them boil down to the fact that the principle thing that the Web really needs is not a superb way of building flashy presentations . . . instead, SVG will succeed because it satisfies the need for an easy-to-use, inexpensive, dynamic, nonproprietary way to build graphical interfaces.

Let me address each of these four points individually.

Simplicity

HTML became popular not because it was the most powerful solution out there; as mentioned earlier, HTML was originally intended to describe physics abstracts, documents that no one but maybe a practicing scientist is ever likely to see. What was more important for HTML's success was that it originally had a small enough set of tags and a simple enough way of using them that you didn't *have to be* a programmer to build sophisticated applications.

This is the HyperCard principle. HTML owes a great deal to HyperCard, Apple's groundbreaking application tool that defined everything from hypertext linking to drag-and-drop language development. The people who took to HyperCard weren't programmers (at least such programmers wouldn't admit it); they were grade-school students and teachers, soldiers, and small-store owners. It was a language that let people who didn't know the arcane syntax and convoluted logic of C++ still write applications that solved *their* needs.

The HyperCard principle is one I personally think has been forgotten (or deliberately displaced) by software vendors. In the rush to build into the business sector, to get those all important e-commerce systems contracts, the creators of

too many software tools have sought to create "solutions" that are incredibly complex and only manageable by other tools (which of course those vendors provide). This holds as much for Web application development as it does for stand-alone, shrink-wrapped software. Declarative language solutions may help here, but these solutions will likely emerge from the open-source community before the commercial software industry rediscovers why it's no longer making any money.

Inexpensive

Expensive tools are another facet of this same problem. For example, my oldest daughter is nine years old. She's learning how to build Web pages on her computer (a hand-me down) on which I installed Linux. The tools she's using to build the pages are open-source—part of a distribution I paid about $50 for, primarily for the convenience of not having to burn CDs. She's now teaching her classmates how to do the same thing on the Apple computers at school, this time using inexpensive software that the school purchased.

The pages aren't fancy (unless you *really* like the Powerpuff Girls and Sailor Moon), but that's unimportant. HTML is both simple enough that she can learn it with relatively little instruction and inexpensive enough that she doesn't have to be wealthy to learn how to work with this technology.

SVG is a lot like that. Because it is an open standard, there are numerous tools emerging for the creation of SVG, some of them bare-bones, some of them quite sophisticated. If you want a full Integrated Development Environment (IDE) and development suite to create SVG applications, then you can (and should) pay a premium for them. But just as you can create HTML applications in a text editor, you can also do so with SVG (as I did in almost every instance in this book). This means the people learning SVG may not necessarily be the ones with the deepest pockets but rather the people who are most motivated to learn, and that is one of the keys that will likely make SVG the HTML of this decade.

Dynamic

Dynamic is one of those great marketing words that has been rendered largely meaningless in this day and age, but in this particular case *dynamic* has a strict technical meaning. SVG is a highly mutable language because of the combination of its XML foundation and its DOM interfaces. In many respects, SVG is one giant text string, with pieces that can be pulled out or added in as the need arises. This means it is in fact an ideal language for the broad category of graphics that are least well served on the Net: information graphics.

Information is not static; it has relevance only in a limited context and typically for only a limited duration. Some of that information can be readily expressed as text, yet the irony is that the Web is actually relatively information-poor. We are visual beings, yet most of the information on the Web is still presented to us in the form of text. Using an XSLT transformation, you can generate graphics based upon dynamic information, and you can even show how specific data points change over time, either dynamically (through DOM) or based upon previous history (by setting specific animation points within a SMIL statement). Such animations are easy to do in SVG, but they would be challenging at best within Flash or even a dedicated chart component.

Moreover, it is possible to build SVG using external libraries (though currently you need to use a lot of DOM to help the process along—see Chapter 10, "SVG Components"), and you can make the mechanisms for describing the links to these resources dynamic. This means you could configure the same core SVG document to display different kinds of output depending upon user preferences, the type of data involved, and the kind of browser being used.

Nonproprietary

Who owns the Web? This is not an academic question, and SVG was actually a central case around which this question hinged. The W3C, since its inception, has adopted a uniform and open license on all standards it has published—in essence stating that the W3C acts as the owner of all W3C material licenses and makes them freely available to anyone who wants to use them, with the proviso that no one can consequently license modified versions of these standards.

This principle is a variant upon the General Public License (GPL, or CopyLeft, as it is sometimes known) that recognizes that the standards being published are essentially being kept as part of a public trust.

An attempt was made to seriously undermine that principle in September 2001, though the issue had surfaced with SVG a few months earlier. In essence, the SVG case involved Adobe making a claim of prior patent on the SVG specification, saying it could prove it had a patent on key technologies used within SVG and consequently could enforce that patent if need be.

The implications of this are not clear, though they probably wouldn't have been good. W3C patents had, up until that point, been strictly royalty-free; that is to say, even if a member company had prior patent claims, they would need either to declare that such patents were royalty-free and not claim royalties for the use of the patent or to withdraw that particular technology from the specification.

During one meeting of the W3C Patent Working Group, a second category of license agreements was created called the Reasonable and Non-Discriminatory (RAND) license. RAND licenses would give the licensees the opportunity to charge

a reasonable fee if someone used the standard in an application. The problem of course comes down to such terms as *reasonable* and *non-discriminatory*. A reasonable fee could easily bankrupt a smaller company that did lots of SVG, and a non-discriminatory license was essentially unenforceable.

SVG (and potentially other already-extant standards) were quietly shifted into the RAND category, and an accelerated response period was instituted to get it into the bylaws as quickly as possible. Fortunately, an alert reader notified the Web community through a number of developer Web sites (most notable Slashdot), and the response was enormous . . . and highly critical of the RAND license. More than 5,000 respondents wrote, including many of the major names in the Web and XML communities. As of January 2002, the W3C rescinded the RAND licenses and converted to royalty-free licenses on SVG, with Adobe giving up its claim to any prior technology that could have been used against other companies or individuals.

Why is an obscure legalistic question about patents so important to SVG developers? A RAND license, if it had been allowed to continue, would have essentially meant that any person who created anything using the standard (such as an SVG graphic or an application for editing the same) would have had to pay Adobe a usage fee—even if they weren't using any Adobe-developed software. It would have also meant that companies could not use SVG to create applications without the possibility that they would have to pay license fees to Adobe, even if they weren't planning on using the Adobe SVG Viewer.

The real effect of such an action wouldn't have been the enrichment of Adobe's coffers (save perhaps via legal fees). Instead, it would have meant that both people and companies would have simply stopped doing anything with SVG, and the language would have become just one more has-been technology.

As it stands now, however, with the RAND issue rendered moot, a number of companies (even small one- or two-person companies) are looking at SVG not just as a way to deploy multimedia presentations but also as the foundation for core technology or as an integral part of new applications. This doesn't mean that people can use the Adobe SVG Viewer as part of their own work; the Adobe SVG Viewer is very much Adobe's own implementation of the SVG standard and should by all rights be licensable. However, should you implement your own version of SVG (or license someone else's), you do not have to worry about the possibility, sometime in the future, that all of your work could be invalidated on the basis of a "prior claim." Nor do you have to worry that Adobe will change the specification beneath you in an unadvertised fashion so that your work will no longer work with the language itself.

In the debate between Flash and SVG, the status of Macromedia's Flash is considerably murkier in this regard. Macromedia has openly published the Flash format, which means you could in theory write applications that consume or produce Flash without potential litigation. However, Macromedia still holds all patents to Flash itself and has not submitted the format to any recognized stan-

dards group—and to be honest should not be expected to do so. This may prove to be more problematic as Macromedia attempts to position Flash as an alternative to complete Web page development, especially as a key provision within the Macromedia Flash license states: "You may not decompile, reverse engineer, disassemble, or otherwise reduce the Macromedia software to a human-perceivable form."

This particular story is far from over. The impetus that open-source software licenses have had on the development of new technology is forcing many people to reconsider the conditions under which they both create and license technology. It is entirely possible (though in the realm of speculation at this point) that Macromedia could choose to make SVG a format for at least reading and potentially writing Flash content. Here again, Macromedia would own the patents on its own engine, but the data format that it's using would be open-source and could only be altered through extensive public debate within the overall development community. In many ways it is this ability to have the debate in public, with the potential for input on the part of all interested parties, that illustrates why SVG should remain a royalty-free standard.

Programming and SVG

This book isn't filled with lots of fancy graphics—indeed, in comparison to what you can do with the language, the content here is distinctly pedestrian. One reason for that (my own inability as a graphic designer notwithstanding) is that I wanted to place more focus on the task of working with SVG from the standpoint of a programmer.

Programming with SVG might well be seen as something of an oxymoron, especially given that I don't really start into serious DOM programming until Chapter 8, "Animating SVG," or so in this book. However, this actually misses the point about SVG, in that the language itself includes any number of features that make it a valid programming environment even before you write the first line of JavaScript or Java.

Declarative programming differs considerably from procedural programming in that regard. In a procedural program, you focus upon specific actions: You assign this variable to this value, you turn this flag on or off, you perform this calculation, and then you move the frog over the result in pixels. In essence, procedural programming involves the creation of a running thread of actions, and you can typically, at any point in the program, say that this event will happen at this time.

Declarative programming, on the other hand, involves the creation of environments: You build primitive objects that in turn go toward the making of more complex objects, which in turn build even more complex objects. However, at any stage, the increasing complexity is also handled at an increasingly high level

of abstraction so that by the time you reach the end of the document, you are in fact dealing with objects at an abstract level.

One side effect of such declarative programming arises over the issue of time. Traditional procedural programming by its nature correlates the position of the code in a routine with time; in other words, any coding written following a particular line of code will be executed after that code (within the same routine, of course).

The declarative nature of SVG is much less temporal in nature—to the extent that you could easily invert the structure of an SVG document so the definitions for objects occur after the objects themselves are invoked, with no real effect on the way the world is rendered. Because of this, in the SVG model, time becomes just one more parameter to work with (usually as part of the various SMIL animation elements). Consequently, creating multimedia applications using SVG is surprisingly easy, especially once you understand that in multimedia programming, the graphics, sound, and video are easy. Unfortunately, it's time management that generally proves to be the nightmare.

Coding against the DOM breaks this model somewhat because you reintroduce time into the equation. Currently, that is unavoidable; the language is still evolving dramatically, and most practitioners of SVG would rather have something that requires patches now than an idealized version of the language that would never actually appear. As more the growing number of SVG developers raise more issues, you'll likely begin seeing less of a need to jump into procedural code to extend the functionality of the language.

As you begin to play with SVG, you should keep this idea in your head: The best SVG code is the one that least breaks the declarative nature of the language. When I started working with SVG, I had a tendency to assume that the language was too simplistic for what I wanted to do, only to discover after having coded some hairy procedure in JavaScript that a little more digging would have revealed an already existing capability that did pretty much everything I had needed. I've rewritten a lot of the code in this book as I've become more adept with the language, and I'm astonished and amused to find that my final SVG code is much less procedural than when I started out. I hope as you get a chance to play with this technology that you will discover this same realization. It's a powerful one, Zen-like in its profundity, and it provides a good indication why I think declarative language programming will become the predominant way to "program."

Installing the Adobe SVG Viewer

Adobe has been one of the major backers of the SVG standard since its inception, and if it succeeds, it will almost certainly be directly because of the efforts of Adobe. One of the key pieces of Adobe's strategy with regard to SVG is the Adobe SVG Viewer.

The Adobe SVG Viewer 3.0 is actually two distinct products: an automation component COM object (EXE) used by Internet Explorer and a set of Java classes used by the Mozilla 0.94+ browser, as well as any Netscape browser that uses that engine (Netscape 6.0+). The components are available for free directly from Adobe at `www.adobe.com/svg`.

Installation is straightforward. Either load the application file to your system and double-click it to run the installation wizard or choose the Open option to run it once it downloads. This component is roughly 3.2MB.

Loading the Adobe SVG Viewer under Linux is a little less straightforward. From the SVG download page, you'll see a paragraph at the bottom indicating archives of older versions:

> *Older, unsupported versions of the Adobe SVG Viewer are available for developers and users who are not yet able to upgrade to the latest version of the Viewer. These versions are not supported by Adobe, and Adobe recommends that you install the latest version, above, unless you are sure that you need to install an older version. Click here to download an older version of the Adobe SVG Viewer.*

This text is a little misleading because the link also contains binaries for the Red Hat Linux distribution (Red Hat 7.3, though it works fine under Mandrake 8.1 and Suse 7.3) and a Solaris version. To install the Linux viewer, you should do the following:

1. In a terminal console, switch to super-user by typing **su**, followed by your root password.

2. Download the file `adobe-svg3.0-linux-i386.tar.gz` to a temporary directory.

3. Run **tar adobe-svg3.0-linux-i386.tar.gz** to extract it to a folder.

4. Enter **./install.sh** to run the installation routine. This will place the relevant files into the Netscape 6.0 and Mozilla 0.9.4 browser's plug-in folders.

5. Exit the console, create or download an SVG file, and then pass the URL to Mozilla. Depending upon the speed of your computer, a few seconds to a minute will pass before a license screen appears. Indicate your assent, and you should then be

able to see your SVG (the license screen is only shown the first time you view an SVG document).

NOTE *If you include an* `<embed>` *tag pointing to an SVG document from HTML, you'll similarly be able to start the viewer. More details about embed tagging are covered in Chapter 2, "Getting Started: An SVG Tutorial," and Chapter 10, "SVG Components."*

Adobe SVG Viewer 3.0 enables you to view SVG 1.0 documents. Check back on the site for updates that will support SVG 1.1 and SVG 2.0.

Summary

Context is everything. This chapter looked at how and why SVG was created and some of the design principles underlying it. Understanding these principles is paramount to using SVG for anything more complex than the creation of static graphics. But even with static graphics it can turn an autogenerated monstrosity into a rich library of reusable objects. Additionally, the open-source nature of SVG means that there will be many different implementations of the specification for any number of different target applications—from printing to full multimedia extravaganzas and from interface design to page descriptions.

SVG can be a little daunting. The next chapter provides a step-by-step walkthrough of several SVG documents and applications to show what goes into them. There's not a lot of how or why available yet (that's why I wrote the book in the first place), but in trying the exercises you'll get a little better feel for the language's flow. It's still early in the game, so sit back, pull up a chair, and immerse yourself in SVG.