# Vector Graphics: From PostScript and Flash to SVG

Steve Probets, Julius Mong,
David Evans, David Brailsford

School of Computer Science & IT
University of Nottingham
(+44) (0)115 9514230
sgp|jxm|dre|dfb@cs.nott.ac.uk

## ABSTRACT

The XML-based specification for Scalable Vector Graphics(SVG), sponsored by the World Wide Web consortium, allows for compact and descriptive vector graphics for the Web.

SVG's domain of discourse is that of graphic primitives whose optional attributes express line thickness, fill patterns, text size and so on. These primitives have very different properties from those of the traditional document components (e.g. sections, paragraphs etc.) that XML is normally called upon to express.

This paper describes a set of three tools for creating SVG, either from first principles or via the conversion of existing formats. The *ab initio* generation of SVG is effected from a server-side CGI script, using a PERL library of drawing functions; later sections highlight the problems of converting Adobe PostScript and Macromedia's Shockwave format (SWF) into SVG.

## Keywords

SVG, Flash, SWF, PDF, PostScript.

## 1. INTRODUCTION

The World Wide Web's enormous success has been achieved by marking up Web documents with a simple set of HTML tags to denote basic structural and layout concepts such as bold headings, tables etc. But perhaps the most striking weakness of today's Web technology is that all graphic material has to be expressed in one of three raster-based formats (GIF, JPEG, PNG). Such formats distort badly at any scaling factor other than 1:1, owing to the pixel-by-pixel way in which the image data is stored. Moreover, these formats are innately non-searchable and this is a severe limitation given that a common use of graphics on the Web is for representation of annotated maps and line diagrams, together with exotic characters and logotypes that lie outside conventional character sets.

In recent years Macromedia's *Flash* and its associated Shockwave format (SWF)[1] have achieved a degree of acceptance, via plug-in software to popular Web browsers, as a standard for displaying vector graphics on the Web. However, SWF uses a binary rather than a text-based format which provides few 'hooks' for text searchability or image extraction.

In early 1998 the World Wide Web consortium (W3C) invited proposals for a new standard for vector graphics on the Web. Given the widespread adoption of XML[2] meta-syntax as a means of expressing new functionality in Web documents it was to not surprising that two of the initial proposals─PGML (from Adobe Systems Inc.) and VML (from Microsoft Inc.)─were XML based. In September 1998 these proposals were subsumed into a new draft proposal called SVG (Scalable Vector Graphics) under the auspices of the W3C[3]. The SVG working group has more than 20 members representing most of the major companies in the graphics software industry. The SVG 1.0 proposal is now a W3C proposed recommendation. It would be fair to say that the graphics model of SVG owes much to PostScript, whereas the tags and their attributes have been influenced by VML. The text-based nature of SVG (and the tree structure implicit in its XML representation) should make text search and graphics transformations relatively straightforward.

The remaining sections of this paper focus first of all on generating SVG from first principles via PERL scripts, followed by a description of how PostScript and Shockwave/SWF can be converted to SVG. Firstly we focus on SVG-PL, which is a library of Perl functions for generating 'SVG on the fly' from Web CGI scripts. The great advantage of creating such a library from first principles is that its design can faithfully reflect the possibilities for object attributes, and object groupings, that are inherent in the SVG target language.

We then discuss two further software tools for converting PostScript and Flash into SVG. In both cases the faithfulness of the conversion to SVG, and the degree of code optimisation that can be achieved, depends on the degree of similarity between the underlying graphics models of source and target languages. If translation occurs from a less complex format into a richly descriptive format then, in order to make maximum use of the richer format, intelligence must be derived from the translation process. Bottom-up inference of intent is notoriously hard. For example, PostScript to SVG is an obviously desirable translation and yet most PostScript is in final form, making it difficult to decide which parts of a graphical object are inter-related and capable of being abstracted into groups.

The fact that SVG is an application of XML means that the purely syntactic aspects of the target language are well specified.

FINAL DRAFT of paper accepted for:
DocEng 2001, November 9-10, 2001, Atlanta, Georgia, USA .

But even within this well-defined framework the design of the SVG tagset has to decide whether certain graphic features shall be elements in their own right or be hierarchically nested sub-elements. Thus it will be helpful to develop some background to the XML meta-syntax, since this affects certain of the non-graphic semantic properties of SVG.

## 2. XML AND SGML

XML is designed to be a simplified subset of the Standard Generalised Mark-up Language (SGML) which has been used in the publishing community for many years. Like SGML, XML is a standard *meta-syntax* . The thing that is 'standard' about it is not that it defines a fixed tagset but, rather, that it defines the alphabet and the 'punctuation' that are used within and around the defined tags. For example, angle brackets are used to delineate a tag (so the start of a paragraph might be marked by `<para>` ) whereas a forward slash following an open angle bracket is the meta-notation for an end tag (so `</para>` would denote the end of a paragraph.) By contrast, although HTML uses SGML meta-syntax, it is not a subset of it. HTML is an *application* of SGML; it is a fixed set of tags which have been defined for straightforward interpretation by Web browser software.

A document type definition (DTD) can be made available to an SGML or XML parser and this sets out a syntax for the tags that will be allowed when marking up documents of a particular class. Thus, a DTD is functionally similar to a meta-syntactic definition for the syntax of a programming language and a full SGML system is capable of creating a parser for the given document class directly from the DTD itself (this is reminiscent of the use of a meta-syntax or 'two-level grammar' in languages such as Algol 68).

Although it is possible to specify, in a meta-syntax, a wide range of structural options within the generated language, this is not generally regarded as a good idea; many years of compiler-writing experience have persuaded language designers not to flirt with anything that allows valid programs in a language to be either difficult to parse or to veer towards ambiguity. Nevertheless, full SGML has an ability, in a DTD, to specify 'inclusions' and 'exclusions', together with optional end-tag and start-tag omissions. These facilities lend great power and flexibility when using SGML tags to mark up complex documents, but in some cases it can be extraordinarily hard to generate parsers for the document class in question, and equally hard to distinguish erroneous document markup from 'expert markup' which is making full use of the tag minimisation features declared in the given DTD.

For these and other reasons full SGML was deemed unsuitable for a simple easy-to-parse framework for extending the functionality of Web browsers. Instead, the XML subset of SGML was proposed in 1997 as a lightweight (but much more restrictive) meta-syntax for Web use. Like SGML, document type definitions (DTDs) are supported in XML and these DTDs specify which tags or *elements* are allowable, the syntax for the tags, their related *attributes* (which are similar in spirit to procedure arguments in programming languages) and other properties related to how the tags may be ordered and nested.

The meta-syntax for specifying the DTD itself is similar to SGML, but is neither identical to it nor a proper subset of it. Moreover a DTD is very limited in the amount of type information it can convey, either for the elements themselves or for their attributes. For these reasons XML parsers (while continuing to accept DTDs as a possibility for tagset definition) are increasingly able to accept *schemas* which specify an XML tagset (with more powerful typing information than a DTD) using XML meta-syntax.

Among the dozen or so restrictions of XML compared to full SGML, three notable restrictions are:

- No tag omissions, all start and end tags must be present.
- All attribute names (c.f. 'formal parameter names' in programming languages) must be fully specified and attribute values (c.f. 'actual parameter values' in programming language function calls) must be enclosed in quotation marks.
- Tags that do not enclose embedded content, but which may still have attributes, are allowed a special shorthand form which elides the end tag with the start tag. Examples of this would be `<BR/>` and `<EMPTY attributes = "allowed"/>`.

XML has its own specification[2] which sets out the allowed syntax for the tags, together with their related attributes and any required tag ordering constraints.

XML can represent nested document structures (e.g. sections, sub-sections and so on) and document elements which are sequenced (e.g. a sequence of paragraphs forming a section of a document). More generally it can represent arbitrary tree structures, with optional attributes attached to the internal nodes, and the Document Object Model (DOM) allows the tree structure to be traversed, inspected and modified. The particular properties of the DOM for SVG will be addressed in a later section.

## 3. PROPERTIES OF SVG

Because SVG is XML-based, the files are generally easier to parse than the binary format files of SWF, and there is the added benefit that elements within the files are stored as text, enabling search engines to index SVG files, or SVG components within HTML files. The fact that SVG is not binary makes it easier for SVG files to be created from compliant applications. SVG files are designed to be platform- and device-independent, with a hierarchical structure enabling sets of vector graphic primitives to be grouped together and manipulated as shapes. Just as in *PostScript* and *Flash*, these shapes can have transformation matrices applied to them which perform scaling and positioning. These transformation matrices can then be altered over time to create animation effects. There are a number of event-driven procedures (e.g `On-Mouse-Over` ) which provide a degree of user interactivity. Properties such as fill-colour and stroke-width are stored as XML attributes. Certain graphics elements (e.g. circle, polygon, rectangle ) and groups are defined as XML elements. The properties of these elements are defined using XML attributes to describe their appearance e.g. a red 50×50 rectangle, positioned at coordinates (100,100) could be specified by:

```
<rect id="myrect" x="100" y="100"
width="50" height="50"
style="fill: #ff0000; stroke: none"/>
```

SVG supports the Document Object Model[4], wherein the graphic is considered to be a hierarchical tree structure, with attributes (colours, line-widths etc.) able to be set at any node, and the DOM being a platform-neutral interface through which the tree structure and the node attributes can be changed. Typically the tree can be modified through a scripting interface, which alters the tree nodes and attributes in the specified way before handing back control to the enveloping SVG application to allow the tree to be re-interpreted. and the graphic updated. A variety of scripting languages could be used to modify the tree, and indeed the XML application called XSLT has been designed precisely for specifying the tree-to-tree transformations that are typical of DOM manipulation. However, for our experiments we have used Javascript (now standardised as ECMAscript[5]); its serial nature, and the resemblance to conventional programming languages, often makes it a more natural vehicle than XSLT for specifying tree modifications that correspond to dynamically evolving graphical effects.

Additional flexibility over SVG graphical attributes can also be obtained by grouping elements together between `<group>` `</group>` tags. The attributes of the `<group>` tag itself can specify such things as line colour and fill pattern which will apply to all of the elements in the grouping.

Creating SVG graphics that have a useful DOM is an important part of intelligent generation of SVG. This in turn implies that the underlying semantics of the graphic primitives and attributes in SVG have to be studied carefully whenever translation from some other graphic format is being undertaken. Even if two graphics languages have similar levels of sophistication, it can be amazingly difficult to achieve efficient and optimal code if the semantics of their underlying graphic primitives differs radically.

To complicate matters further, some of the SVG graphic properties can be inherited not only from node attributes in the SVG tree but also from browser stylesheets within the CSS or XSL frameworks. This brings into stark focus an issue first identified by workers standardising the Computer Graphics Metafile[6,7] proposals: is a given graphic property *intentional* or is it purely *stylistic*? For example, the red background colour, white lettering and hexagonal shape of an international STOP sign reflect agreed intentional standards whereas the text size, background colour and general layout of a Web page may reflect nothing more than what 'looks good'. There is a strong argument that intentional properties should be expressed as XML attributes of the object in question, leaving any optional styling characteristics to be inherited from stylesheets.

## 4. CREATING SVG *ab initio* FROM PERL

Web pages are increasingly being tailored dynamically, in response to the profiles of the various users that visit a given Web site. This profile might be built up from information such as the IP address of the client machine coupled with extra information that the user has typed in to a Web form. But if the confected page has vector graphic components such as flowcharts, maps or line diagrams it is galling that current Web technology requires the preparation of a (potentially very large) library of GIF images so that the correct image can be chosen and inserted into the generated page. Each GIF insert is in itself large and incapable of being scaled elegantly by client browsers. By contrast, SVG offers the possibility of generating server-side vector graphics 'on the fly' and downloading much smaller files which have just a few line drawing and fill commands.

The technology for flexible response to Web page queries and the gathering together of customised responses is generally handled by a Common Gateway Interface (CGI) script. These scripts frequently make use of the PERL language and so it seemed natural to provide SVG capabilities for CGI via a PERL library. The SVG-PL library (which can be found at `www.ep.cs.nott.ac.uk/projects/SVG/SVGPL`) is a collection of Perl 5 modules, which define many useful and user-friendly functions to ease the process of generating valid SVG documents. The structure of SVG-PL is very similar to that of PDF-PL (`www.ep.cs.nott.ac.uk/pdf-pl/`) which generates 'PDF on the fly'. Each SVG-PL application program must have a `File` object that carries a file handle for the document being generated, and a `Graphics` object that acts as the interface through which all drawing functions are invoked.

The key advantage of generating SVG *ab initio* is that the PERL library functions can be set up to mirror exactly the functionality that SVG provides. SVG-PL provides more than a hundred functions for generating all possible SVG elements with appropriate attributes and these functions can be broadly grouped into the following five categories: Path and Shapes, Text and Fonts, Links and Rendering, Extensions and Filter Effects, Animation. PERL has the usual programming language control constructs coupled with the ability to make certain functions be 'private' and invokable only in well-defined surrounding contexts. For example, the generation of a closed path can be achieved within a SVG-PL `beginCPath` and `endCPath` environment which then, in turn, allows the use of the private unctions `moveto`, `lineto`, `closepath`, `hlineto`, `vlineto`, `curveto`, `arcto` etc. An extensive range of aliases has been defined to help in generating compact PERL code so, for example, `b()`, `e()` and `d()` will invoke the corresponding `begin...()`, `end...()`, and `draw...()` functions. Note that SVG-PL automatically decides which boundary to close when the alias function `e()` is called, although the user could still explicitly close the open boundary by calling its corresponding `end...()` function. Here is a short SVG-PL example to generate code for moving to a specified point and then generating a black-line curve having a stroke width of 5 points:

```
#cpath is a SVG-PL keyword for user
#customised paths
$g->b(cpath,style,"fill:none; stroke:black;
stroke-width:5");
  $g->moveto("abs", $xoff, $yoff);
  $g->curveto("rel", 0, 0, $xoff*1.5, 15,
$xoff*3, 0);
$g->e();
```

A complete PERL program (ignoring some initialisation code) using SVG-PL, which generates a simple 'Hello World' SVG drawing would be:

```
### Hello World! ###
#OPEN FILE
my $svg = File->new();
$svg->open("Hello.svg", "public",
"encoding", "iso-8859-1", "silent");
#BEGIN GRAPHICS
my $g = $svg->beginGraphics();
# TEXT lines
$g-
>b("svg","viewBox","0","0","600","400","wid
th","400","height","300");
$g-
>b("text","xval","200","yval","200","style"
,"font-size:24pt;
       font-face:Helvetica");
 $g->printTxt("Hello World");
 $g->e();
$g->e();
$svg->close($g);   # END line
```

In the first `OPEN FILE` line, a local variable (`$svg`) is defined and initialised to be the reference to a new `File` object. The second line opens a file for output and prints the relevant headers to the SVG file. The `BEGIN GRAPHICS` line creates a `Graphics` object from the `File` object and all graphic elements must be drawn using `Graphics` methods only. The `TEXT` lines first open a drawing area, then define a text marking area for putting the text string 'Hello World' onto the page at a specified location in user co-ordinates. The two `e()` statements close the text followed by the graphics area. The `END` line closes first the Graphics object and finally the File object as well as the output file.

The generated SVG code would look like the following:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG
03March 2000//EN"
"http://www.w3.org/Graphics/SVG/SVG-
19990303.dtd>
<svg width="400" height="300" viewBox="0 0
400 300">
 <text style="font-size:48;font-
face:Helvetica;fill:black;" x="75.992"
y="174">Hello World</text>
</svg>
```

Note that in this example the number of lines in the SVG code is rather less than that in the Perl program, largely because of the fixed overhead for invoking the libraries and setting up File and Graphic objects. For larger examples the Perl code is about 50% the size of the generated SVG because of the short aliases already discussed and the availability of control constructs such as 'for' and 'while' to simplify the generation of similarly-structured SVG elements.

There are also functions which allow users define a set of x and y coordinates as a pivot point to which a text string could be aligned. For example one could indicate the top left hand corner of a text string to be the point specified by the user-defined set of x and y coordinates, and thus have the text aligned to the left and bottom against the pivot point. An example SVG drawing generated from use of such text alignment functions provided by SVG-PL is shown in figure 1.:

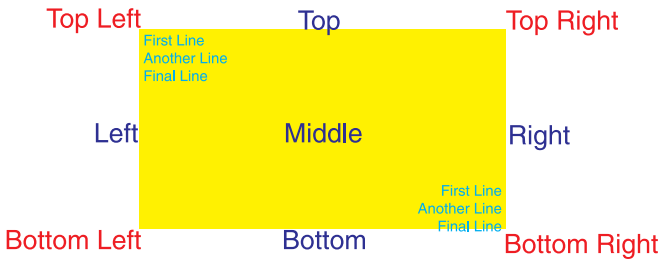


**Figure 1: SVG Drawing Generated From Text Alignment Functions**

SVG code produced using the current release of SVG-PL is, in some sense, about 80% optimised. This is due to the fact that the `begin ...()` and `end...()` constructs are a natural way of expressing SVG's groupings of graphic primitives having the same attributes and SVG-PL also has some low-level optimisations that will, for example, convert multiple `lineto` commands into a single one. It is hoped to implement better optimisations in the near future. For example, attributes within a group boundary are currently not remembered by SVG-PL; if elements within a group boundary repeat the attributes already defined in the group statement then the repeated attributes can be discarded. Moreover, there is useful functionality in the companion PDF-PL library for generating Acrobat/PDF (namely, a function to save the current graphics state) which has not yet been implemented in SVG-PL. This and other improvements are planned for the next release of SVG-PL.

Demonstration projects are available at: `www.ep.cs.nott.ac.uk/projects/SVG/svgpl/`

## 5. CONVERTING POSTSCRIPT TO SVG

PostScript is a page description language designed and implemented by Adobe Systems, with the first version being released in 1984. Since that time, PostScript has become an industry standard for quality text and graphics, and has now reached its third major version (PostScript Level 3). PostScript is an interpreted graphics description language, which uses postfix notation, and which is capable of specifying complex two-dimensional paginated graphics in a manner which is both device and resolution independent.

The Portable Document Format (PDF)[8] is another page description standard which has been developed by Adobe Systems, with an imaging model which is very similar to that of Level 2 PostScript[9].

## 5.1 PostScript/PDF and SVG

As part of our SVG project, we have been developing a Ghostscript driver to output SVG files. Ghostscript is a PostScript interpreter, which is available under the terms of the GNU Public License, and in commercial forms too. It is widely used as a free PostScript interpreter on various operating systems, both for on-screen display and for printing to various printers through its various 'back-end' printer drivers. As well as Ghostscript's wide use and extensibility, another benefit is that because of PDF's similarity to PostScript, Ghostscript has been modified so that it can interpret PDF as well as PostScript. Thus, our SVG driver will effectively convert both PostScript and PDF into SVG. In what follows, therefore, a reference to PostScript input will also be applicable to PDF input. The graphics model underlying PostScript and PDF is very similar to that of SVG, thus making conversion relatively straightforward. Vector graphics such as lines and curves can map directly from one to the other, and text can be output using Cascading Style Sheets[10] (CSS2). Text output must, of course, be careful to use appropriate entities for characters such as <, >, and & if these are to appear 'as themselves' and not be mistaken for XML metasyntax. Care must also be taken to use the PostScript font encoding vector to extract further information for characters outside the normal ASCII range, such as accented characters and symbols. Bitmapped images are converted to separate JPEG files which can then be referenced from the SVG file using an `<image>` element, which is similar to the HTML `<img>` element.

So, if care is taken with the issues outlined above, the basic framework of PostScript to SVG conversion is relatively straightforward, but problems begin to arise when attempting optimisation of the generated SVG code. We have already seen, when generating SVG directly from Perl scripts, that repeated attributes (e.g. those specifying colour, fill-type, or font), and the fact that XML is quite verbose, can lead to large file sizes. The same issues arise when converting from PostScript and, at the very least, the output needs to minimise the coordinate information, making use of SVG's ability to specify absolute or relative coordinates, and omitting `x` or `y` coordinates whenever these remain unchanged. For example, when printing text along a horizontal line, an initial `(x,y)` position is required, but afterwards only `x` positions will be required. Attributes which change infrequently are best handled by using SVG's groups to set the attribute so that it does not need to be repeated within the group. For example, if the fill colour was set to red, one would wish to generate:

```
<g style="fill:#ff0000">
<path d="M20,20 L50,10 L80,80 L70,95"/>
...
</g>
```

The problem with trying to do this in a Ghostscript driver is that GhostScript's internal interfaces are set up for writing printer drivers where code for the target printer is to be generated as quickly as possible and where optimisation is not a major issue. There is a large amount of 'graphic state' information implicitly present in GhostScript's data structures but it is not usually necessary to probe this state extensively to generate adequate code for a printer. Indeed, many target printers will be building up their own internal notion of 'state' from the stream of commands that they receive. For these reasons GhostScript output drivers generally retain little or no knowledge of what has gone before, or what is coming next.

To achieve the desirable code optimisation of grouping elements together where possible, the GhostScript SVG driver must save substantial extra state information to decide when it is worth while to start a group. This grouping of repeated graphical elements is possible in SVG by using `<symbol>` or `<g>` elements with IDs specified inside a `<defs>` section, which are then rendered using the `<use>` element. However, extracting repeated paths is difficult inside a Ghostscript driver. One possibility here, is to routinely group all paths, giving each one an ID, and to retain knowledge of that group in memory. When a subsequent matching path is found the original definition can then be re-used. Ghostscript already enables one optimisation in that it recognises rectangles internally and provides output drivers with a procedure for filling them and/or stroking the perimeter. In this case, the SVG output can use the `<rect>` element instead of a filled or stroked path containing four lines (which is all that is possible in PostScript).

## 5.2 Size and speed issues

Simple vector graphics are likely to be smaller as SVG files than as application-generated PostScript files, because the latter typically contain a sizeable overhead in the size of the PostScript prologue prepended to all files (perhaps 10KB or more). As we have already noted, an SVG file has a certain overhead in its header declarations but thereafter it need only contain the elements actually required to fully render the equivalent image. More complex graphics will generally be a similar size, or larger, in SVG because of the syntactic overhead of XML.

Rendering of SVG created from PostScript should be about as fast as that seen by displaying the PostScript on the same machine, since the SVG will no loops or procedures, and will have utilised at least some of the file size optimisations.

## 5.3 Software

It is hoped that by the time of publication the Ghostscript SVG driver will be available. For further information please see: `www.ep.cs.nott.ac.uk/projects/SVG/ps2svg/`

# 6. CONVERTING FLASH TO SVG

Macromedia's Flash format is currently the most widely used vector graphics and animation format on the Web. Flash movies are distributed as SWF (pronounced 'swiff') files, a compact binary file format that requires an additional browser plug-in to be available in order for the movie to be viewed. Objects in Flash movies are described by standard vector graphics primitives such as lines and curves. Bitmap objects can be included in Flash files, and all the objects can be animated; user interactions can be specified; and sound can be streamed into the movie.

Flash files are created using a Flash editor available from Macromedia, which generates files with a `fla` extension. These files are editable by Flash tools and the objects, images and animations can be altered. Flash movies for Web distribution are turned into final form by exporting them as SWF files from within the editor. This format stops the file from being editable and is a compact format suitable for Web distribution. The SWF format defines shapes, which are then placed onto a stage using a transformation matrix to scale and position them. The transformation matrix can be altered from frame to frame to create the animation effects. This leads to sequences of controls such as:

```
Define shape 1
Define shape 2
Place shape 1
Place shape 2
Show frame 1
Place shape 1 in new position
Place shape 2 in new position
Show frame 2
```

In comparing SWF and SVG the key differences in the semantics of the graphics primitives, their grouping into shapes and the manipulation of these shapes have to be addressed.

## 6.1 Grouping Graphics Primitives into Shapes

Both SVG and SWF allow shapes to be built up from paths described using position, line and curve primitives (note that SWF curve primitives are described using quadratic B-splines). In both languages the graphics primitives have associated properties such as fill and stroke colours, and patterns. SVG uses a similar model to PostScript and PDF in that a path is a set of points that have equivalent stroke and fill properties. SWF, on the other hand, has a more flexible method of defining paths, where adjacent primitives can be described in the same path but with different associated fill or stroke properties. This can best be described by looking at the following simple shape (figure 2).

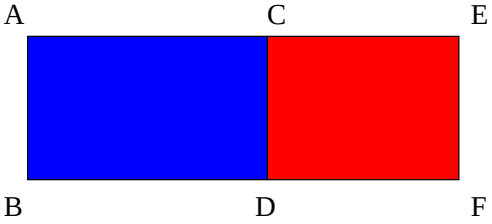


**Figure 2. A shape showing semantic path differences**

In PostScript, PDF and SVG this shape is likely to be defined by two paths. The first path (which describes the left-hand side of the rectangle) sets the fill colour to blue and is defined in the following manner:

```
move to A, line to B, line to D, line to C,
close path
```

The second path (which describes the right-hand side of the rectangle) sets the fill colour to red and may be defined as:

```
move to C, line to D, line to F, lineto E,
close path
```

For comparison, one way in which Flash could define the shape is as follows:

```
move to A, line to B with fill left = blue,
line to D with fill left = blue, line to C
with fill left = blue and fill right = red,
line to E with fill right = red, line to F
with fill right = red, line to D with fill
right = red, move to A, line to C with fill
right of blue
```

The example above illustrates that in SWF, each line can have two fill styles, one of which fills to the left of the line, the other to the right. This difference causes some problems when converting SWF into SVG. Start and end coordinates of individual lines need to be compared and those with adjoining coordinates and similar attributes can be assigned to SVG paths.

## 6.2 Properties of Groups and Shapes

A further difference between SVG and SWF is that SVG has a hierarchical model whereas Flash is essentially linear. Groupings of primitives in SVG can support sub-groups which have inherited properties, enabling common properties to be abstracted up the hierarchy and to affect whole sub-groups. Groups and sub-groups can be named, and named groups can be referenced through the Document Object Model. Properties such as styles, fill colour etc. can be manipulated through the DOM by JavaScript or other XML transformation techniques such as XSL. If the groups are designed intelligently then DOM manipulations can be used to facilitate object re-use and reduce file size. For example, let us assume the effect required is to draw the outline of a black circle within a square filled in red. If a similar shape is required, but with the circle filled in blue (i.e. in a later frame in an animation) (see figure 3.), there is no need to define two separate circles; rather, the fill-colour attribute of the circle could be altered.

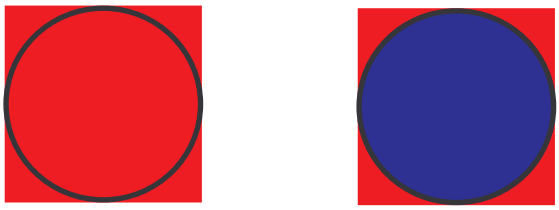


**Figure 3. How JavaScript manipulations of the DOM can alter a graphic**

There are many ways this could be achieved, one of which is as follows:

```
<g name="coloured-square-circle-shape"
style="fill: #ff0000; stroke: none;">
<rect id="outer-rect" x="0" y="0"
width="20" height="20"/>
<g name="inner-circle" style="stroke:
#000000; stroke-width="2">
<circle cx="10" cy="10" r="10"/>
```

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

This shape defines the black outlined circle in the red square. In order to change the fill colour of the circle to blue the following JavaScript could be written:

```
el=document.SVGref.getElementById(inner-
circle)
sty = el.getStyle()
sty.setProperty("fill","#0000ff")
```

Similarly, the SWF format labels shapes with unique identifiers, based upon the way in which the original Flash movie (`.fla`) file was edited from the Flash browser. These shapes can have colour or transformation matrices applied to them on a frame by frame basis. In transforming SWF to SVG it is therefore quite likely that the shapes labelled in the SWF file can be directly translated into SVG without requiring any major reworking of groups, and that transformations applied to these shapes in SWF can be represented by DOM manipulations of the associated SVG groups.

## 6.3 Animation

Animation within SVG files can be managed in one of three ways:

- using SVG's own declarative animation model;
- manipulating the document object model (DOM);
- using the Synchronised Multimedia Integration Language (SMIL) [11].

A discussion of SMIL is beyond the scope of this paper and so we shall restrict our attention to the first two animation methods listed above.

The SVG declarative animation model allows a named group to be animated over a period of time using the `<animate>` element. The object is referenced and the attribute to be animated is specified. This attribute can be any valid XML attribute such as opacity, style or a transformation matrix. The start and end values (or a list of applicable intermediate values) for the attribute are specified and a duration (or absolute start and end times) are specified. The SVG rendering engine then calculates the intermediate steps and performs the animation.

Animations on SVG elements can also be accomplished by explicitly manipulating the Document Object Model tree from a time-line and one way to do this is by using JavaScript. JavaScript functions can be used to obtain and manipulate attribute values of named SVG elements (groups of primitives) which can then be exported into the DOM. Using JavaScript these functions can be called from within a timer function. Thus by periodically altering the transformation matrix applied to a given SVG element the effect of animation is achieved. Because JavaScript is a programming language, there is more frame-by-frame flexibility in this approach than in SVG's native method and JavaScript functions can be written to calculate values for element attributes before setting them up in the altered tree.

This DOM-based manipulation of SVG elements is analogous to SWF's animation model and so, by calling Javascript functions that manipulate attributes such as a transformation matrix, SWF animations can be directly modelled in SVG.

## 6.4 Text

One of the major benefits of SVG is that it is an XML-based graphics language, and so text is searchable and indexable by Web search engines. Text in SWF is maintained in two forms, as glyphs and as codes. SWF has a font object that defines the character outlines. To reduce file size, only those characters used in the file are defined in the font definition, with a code table that defines the ASCII character codes for these characters. Thus when translating SWF to SVG, text can either by specified as character codes, or an SVG font can be built up using the glyphs and used as the character representation.

## 6.5 Performance issues

Two major performance issues in a Web animation environment are file size and rendering speed. The first of these relates to the question of code optimisation and will be addressed in the next sub-section. Turning to the question of speed, animations need to be rendered at a specified frame rate, and the rendering of the frames must occur within the time span for the frame. Additionally the graphic/animation as a whole must be as compact as possible to allow downloading from the Web in a reasonable time.

The latest SVG browsers now implement native SVG declarative animation elements. Unfortunately, SWF animations do not map very simply onto this animation model and so our converter currently supports animations via manipulation of the DOM. For simple animations the speed at which a new transformation matrix can be applied to a named graphic element is dependent on the complexity of the element. Obviously the simpler the graphic, the smaller the number of data points that have to be calculated by the application of translations, rotations and skews.

## 6.6 Code optimisation

In creating SVG files there are certain optimisations that should be undertaken and these are very similar to the optimisations already discussed for translating from PostScript. The data for a path is stored as an XML attribute, with key letters equating to positioning (`M`), line (`L`), and curve (`C`) commands. Where possible the translation software should analyse the path data and optimise the paths. Likely candidates for optimisation include using the simplified commands that draw purely horizontal or vertical lines (`H` or `V`) wherever a horizontal or vertical line is drawn; this is because lines of this type require only one data point as opposed to two. e.g. a horizontal line should not be defined as `<path d="M 10 10 L 20 10"/>` but instead as `<path d="M 10 10 H 20"/>`. Similar considerations apply to curves (especially when translating from quadratic B-splines (SWF) into cubic Béziers (SVG). A mindless translation of two adjoining quadratic B-splines simply replaces the quadratic splines with two cubic Béziers (and this inflates the amount of data since an extra set of control points is required─ although it should be noted that later releases of the SVG specification enable quadratic B-splines to be specified). However, if the tangents to the splines are equivalent at the end

of one curve and at the beginning of the other, then the two quadratic splines may be replacable by a single cubic Bézier, leading to a reduction in the path data. This kind of problem occurs in many forms in any translation from a simpler graphics language into a richer one. For example, a circle in SWF might be represented by eight quadratic B-splines: to convert this graphic to SVG the splines could be replaced intelligently as described above, or the optimisation could go a stage further in attempting to recognise the graphic as a circle and utilising the `circle` element instead, leading to a large reduction in data points.

Native SVG files, being XML based, are unlikely to be as small as SWF files, which have a very compact representation. There are many factors which affect SVG file sizes e.g. whether text is maintained as glyphs or as codes and which animation model is used. SVG files use XML links to point to included bitmap images, whereas SWF files tend to embed the images into the movie. Thus if one particular image is used in different movies being disseminated over the Web there is a likelihood of it being cached in SVG, whereas in SWF it will be downloaded again. Comparing SWF with SVG directly is therefore misleading, it is more accurate to pre-compress the SVG files before doing the comparison. In this way, even allowing for the above discrepancies, at least comparison is between two like files (both compressed binary). Tests on the movies indicate that a compressed SVG file is about one and a half to two times the size of its SWF counterpart.

## 6.7 Software

An application has been written in C, that outputs SVG from an input SWF movie. The application can be accessed through a Web forms interface, whereby a SWF file is uploaded and a SVG file is returned. At present the application has restrictions in that only version 2 of Flash is supported and features such as sprites and sound are unsupported. The converter can be accessed from:

`www.ep.cs.nott.ac.uk/projects/SVG/flash2svg`

# 7. CONCLUSIONS

By the end of 2001, SVG is very likely to be adopted as a vector graphics standard for the Web. It will mark the first time that XML, with its innate tree model, and tree-node attributes, has been used as the framework for a graphics format. Insofar as SVG's elements give a standardised vocabulary for Web graphics, with easy text searchability then the new standard is to be welcomed. Equally, the large number of XML parsers, many of them in the public domain, will enable SVG to be analysed and transformed. However, as we have discovered in this paper there is a need for something more than mere tree transformations when rendering quality graphics. In the end SVG is ‘an interpreted data structure’ rather than a full programming language. The graphic semantics of the attributes at the SVG tree's internal nodes are of crucial importance; the suitability of JavaScript tree-manipulation functions, for modelling the higher-order functionality found in other graphics languages, has to be tackled.

The definition of SVG has clearly been motivated by the fact that much of the existing graphic material to be converted, from languages such as PostScript and SWF, will already be in ‘final format’ with absolute co-ordinates in place and little or no need for any dynamic behaviour. At this level SVG does an adequate job and will provide a sound framework for Web graphics.

Different graphics formats have different internal structures and different degrees of ‘object orientation’. In order for graphics to be considered ‘smart’, the underlying format must support intelligent structural descriptions of the graphic. If graphics can be generated that make maximum use of hierarchical groupings then the graphics become more usable. Consider a vector drawing of an aeroplane: if the fuselage, wings and windows are grouped together, and the attributes describing the colour, are extrapolated to the top of these groups, then a template aeroplane can be created. This allows simple DOM manipulations to create a blue aeroplane with red wings and black windows, or a purple aeroplane with green wings and transparent windows (opacity is supported in SVG).

Interestingly, the existence of similar data structures in two given graphic formats is not necessarily helpful in translating from one to the other. PDF has an innate tree structure of pages which is of no help at all in translating from PDF to SVG where the tree structure represents hierarchical relationships of graphic objects within a single page. More generally, graphics can only be considered smart if the internal structures of the format accurately reflect the abstract structure and projected use of the graphic—the generation and translation tools then need to be able to generate the correct groupings and labellings of the individual graphical elements. What we find is that formats such as SWF give sufficient ‘object’ information to enable translation to a rich format such as SVG to be undertaken, with only a minimal need for intelligent feature extraction, whereas other formats, such as PostScript, prove difficult, largely because much of its semantics is implicit in a complex ‘graphic state’ coupled to a procedure and argument passing mechanism more reminiscent of a low-level assembly language.

By contrast, at the level of graphical primitives, those in PostScript and PDF are well aligned with SVG in terms of the way that lines are stroked and mitred, shapes are filled and so on. Here the conversion from Flash, with its ideas of having fill patterns on either side of a line, rather than as part of an implied closed shape, can cause some very tricky translation problems.

The interesting open questions are whether SVG's various options for animation will be robust enough for real-life use and whether and how SVG will develop beyond the ‘final form’ approach to allow for Web schematics (e.g. something as procedurally simple as Brian Kernighan's PIC[12]) to be directly implemented in SVG, procedures and all, with the procedural mechanisms not necessarily involving the dynamic rewriting and re-interpretation of tree-like data structures

# 8. REFERENCES


[1] World Wide Web Consortium. Scalable Vector Graphics. http://www.w3.org/Graphics/SVG/Overview.html

[2] World Wide Web Consortium. Extensible Markup Language. http://www.w3.org/XML

[3] World Wide Web Consortium. Scalable Vector Graphics. http://www.w3.org/Graphics/SVG/Overview.html.

[4] World Wide Web Consortium. Document Object Model. http://www.w3.org/DOM/.

[5] European Computer Manufacturer's Association. Standard ECMA-262 ECMAScript. http://www.ecma.ch/.

[6] Petrotechnical Open Software Corporation. How to get the ISO CGM Specifications. http://www.posc.org/technical/cgmpip/get_cgm.shtml.

[7] Lofton Henderson, Anne Mumford. The CGM Handbook. Academic Press. 1993.

[8] Adobe Systems. Portable Document Format Reference Manual. 1993. Addison Wesley.

[9] Adobe Systems. PostScript Language Reference Manual. 1985. Addison Wesley

[10] World Wide Web Consortium. Cascading Style Sheets. http://www.w3.org/Style/CSS.

[11] World Wide Web Consortium. Synchronised Multimedia. http://www.w3.org/AudioVideo/

[12] Brian W. Kernighan. PIC, A Graphics Language for Typesetting. Software Practice and Experience. 12(1). January 1982