VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

School of Mathematical and Computing Sciences
Computer Science

Evaluating Scalable Vector Graphics for use in Software Visualisation

Matthew Duignan, Robert Biddle, Ewan Tempero

Technical Report CS-TR-02/19
November 2002

School of Mathematical and Computing Sciences
Victoria University
PO Box 600, Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

School of Mathematical and Computing Sciences
Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341, Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

Evaluating Scalable Vector Graphics for use in Software Visualisation

Matthew Duignan, Robert Biddle, Ewan Tempero

Technical Report CS-TR-02/19
November 2002

Abstract

The W3C's new 'Scalable Vector Graphics' standard provides a new technology for deploying interactive graphical content over the Web. This paper briefly describes SVG and some of its more interesting features. We have developed a principled model for evaluating the suitability of technologies for supporting software visualisation. We briefly outline this model, and then apply it to SVG.

Author Information

Robert Biddle is a member of the academic staff in the School of Mathematical and Computing Sciences and the School of Information Management at Victoria University of Wellington. Matthew Duignan is working towards his PhD in Computer Science at Victoria University of Wellington. Ewan Tempero is a member of the academic staff of the Department of Computer Science at the University of Auckland.

Evaluating Scalable Vector Graphics for use in Software Visualisation

Matthew Duignan, Robert Biddle† **Ewan Tempero‡**

† School of Mathematics and Computing Sciences, Victoria University of Wellington
Wellington, New Zealand
‡ Department of Computer Science, University of Auckland
Auckland, New Zealand

Abstract

The W3C's new 'Scalable Vector Graphics' standard provides a new technology for deploying interactive graphical content over the Web. This paper briefly describes SVG and some of its more interesting features. We have developed a principled model for evaluating the suitability of technologies for supporting software visualisation. We briefly outline this model, and then apply it to SVG.

1 Introduction

The World Wide Web Consortium's (W3C) new 'Scalable Vector Graphics' standard provides a new technology for deploying interactive graphical content over the Web. This technology has many possible applications, one of which is in the area of software visualisation. Its potential use in deploying software visualisations *over the Web* is a particularly interesting possibility. Our current work in developing a web-based software visualisation architecture for reuse (Marshall, Jackson, McGavin, Duignan, Biddle & Tempero 2001) requires a technology that can deliver just this. For this reason, we need to evaluate Scalable Vector Graphics for use in software visualisation.

Being a new technology, Scalable Vector Graphics (or SVG) is far from proven. This is especially true in its application in software visualisation.

This paper begins by briefly describing SVG and some of its more interesting features. We then outline a principled model that has been developed for the purpose of evaluating SVG for use in software visualisation. This model is developed to ensure that it could also be applied to other software visualisation media. We apply this model to SVG to create a detailed point by point evaluation of SVG for use in software visualisation. We then discuss some issues raised by the evaluation, and then finish with conclusions and future work.

2 What is SVG

SVG is a newly-created open standard, which reached version 1.0 on the 4th of September 2001 (W3C 2001*b*). It is based on XML (W3C 2001*a*), and builds on other web standards such as those for animation (W3C 2001*c*) and colour (International Color Consortium 1999).

2.1 Bitmap VS. vector graphics

As the name suggests, SVG is based on *vector* graphics rather then traditional bitmapped graphics. The difference is primarily in the content of the graphics file format. Bitmap graphics describe the properties of each individual point of colour that makes up the image. In contrast to this, vector graphics describe the logical entities in a graphic. Instead of encoding each unit of colour (pixels), vector graphics files encode entities such as curves, circles, and words. The structure of the image is maintained in its file format. Only when the vector graphics file is to be rendered on a bitmapped display are the low level pixel values calculated.

2.2 SVG in practice

The SVG XML language defines a set of element types that describe graphics. For example, there is a circle element that defines a circle to be drawn when the SVG is shown in a renderer. The code for this might look like figure 1. The attributes *cx* and *cy* specify the centre coordinates while *r* and *style* define the radius and appearance of the circle respectively. The rendered result can be seen in figure 2.

```
<circle cx="100"
        cy="100"
        r="50"
        style="stroke:blue; fill:red;
               stroke-width:5"/>
```

Figure 1: SVG code to draw a circle.

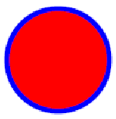

Figure 2: The circle from figure 1 displayed by an SVG browser plug-in.

In the XML way, SVG graphics are defined structurally, with lower level elements able to be combined into higher level groups. An arrow can be defined by grouping a long black line and a triangular path for the head together in a *g* element, as shown in figures 3 and 4.

This arrow group could be defined as a *symbol* that could then be *used*[1] in many places in the diagram.

[1]A *use* element can use a *symbol* element

```
<g style="fill:black; stroke:black">
  <line x1="0"
        x2="70"
        y1="10"
        y2="10"/>
  <path d="M70 5 L70 15 L80 10 z"/>
</g>
```

Figure 3: SVG code that draws an arrow as a line and a path.

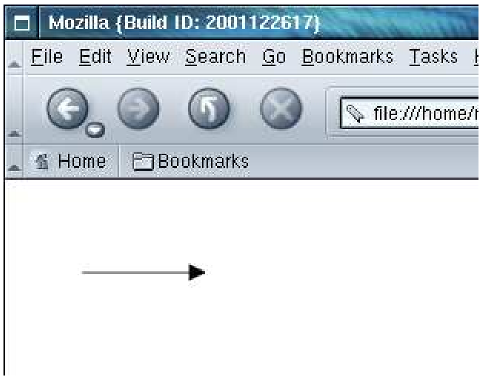

Figure 4: The arrow from figure 3 displayed in an SVG browser plug-in.

```
<symbol id="Arrow"
        preserveAspectRatio="none"
        viewBox="0 0 80 20">
  <g style="fill:black; stroke:black">
    <line x1="0"
          x2="70"
          y1="10"
          y2="10"/>
    <path d="M70 5 L70 15 L80 10 z"/>
  </g>
</symbol>

<use height="20"
     width="80"
     x="50"
     y="0"
     xlink:href="#Arrow" />
<use height="20"
     width="80"
     x="50"
     y="40"
     xlink:href="#Arrow"
     transform="rotate(165 90 40)" />
<use height="20"
     width="20"
     x="50"
     y="60"
     xlink:href="#Arrow" />
<use height="100"
     width="80"
     x="50"
     y="60"
     xlink:href="#Arrow" />
```

Figure 5: SVG code that defines an arrow as a symbol, and then uses it four times with differing coordinates, sizes and transformations applied.

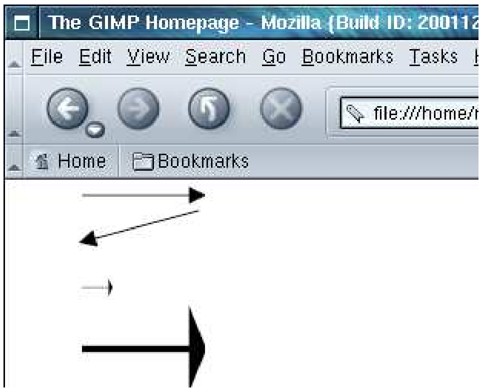

Figure 6: The SVG from figure 5 displayed in an SVG browser plug-in.

When the arrow is used, it can be stretched, rotated or moved arbitrarily (figures 5 and 6).

The other highlights of SVG's functionality include animation, interactivity, and hyper-linking as well as a myriad of graphical display constructs such as filter effects and gradients. SVG's capacity is also augmented hugely by its inclusion of script elements. Script elements contain links to (or include in-line) executable code written in a scripting language. These scripts are interpreted at runtime, and can manipulate all aspects of the SVG currently being viewed by interacting with the DOM[2]. The scripting language is specified when including a script, allowing different scripting languages to be used. The SVG specification does not make clear what scripting languages, if any, must be supported by a conformant viewer, although ECMAScript and javascript seem to be the de-facto standards. Another feature of SVG is that it is designed principally with the web in mind. The standard expects SVG to be embedded in web pages, and it provides built-in functionality such as hyper-linking to other web resources.

2.2.1 Implementations

The dominant implementation of SVG already allows the embedding of SVG graphics in web pages. This mature implementation is the Adobe SVG plug-in (Adobe Systems 2002*b*). There are versions for Macintosh, Microsoft Windows, Solaris and Linux operating systems. The Windows implementation is compliant with the Microsoft ActiveX standard, and therefore can be built into other Windows programs. Other implementations include a pure Java system for SVG rendering (The Apache Software Foundation 2002), a KDE plugin (KDE 2002), and the first native web-browser implementation in Mozilla (Mozilla.org 2002). Even with these later implementations still in development, there is clearly already an adequate base of support to make SVG a valid option for use in software visualisation.

2.3 Competing vector graphics formats

The web has long been dominated by bitmap graphics such as JPEGs and GIFs, but vector graphics are not new either. SVG has one major competitor in Macromedia Flash. Flash is the de-facto standard in web vector graphics and already has a broad development base. Macromedia provide a number of mature authoring tools for web designers, and have free

[2] In XML, the Document Object Model, or DOM, allows programmers to access an XML file by interacting with objects

viewing plug-ins for Windows, Macintosh and Linux. Macromedia Flash also provides animation, interactivity and many graphics capabilities. The core difference between Flash and SVG is in the file storage details. While SVG documents are stored in a plain text format (XML), Flash content is stored in a binary format. The implications of this, as well as further comparison of the formats is covered in section 7.1.

2.4 SVG in software visualisation

Our research project (Marshall et al. 2001), a web-based software visualisation architecture for reuse, requires a medium for delivering interactive graphics over the Web. At face value, SVG looks like it could be an ideal solution. However, because SVG is so new, there has been very little exploration of its potential for use in software visualisation. Additionally, SVG is just one of the many possible media that could be used in such an architecture.

There needs to be a way to systematically evaluate, and then compare the various media, be it Flash, Java, dynamic GIFs, or any other future display software. To this end, we have developed a model for evaluating media for use in software visualisation.

3 Constructing a model for evaluating software visualisation media

In this section, we describe the rationale for a development of a model for evaluating software visualisation media, and outline the resulting model.

Firstly, we need to define what is meant by "software visualisation media". For our purposes, we define a software visualisation "medium", and "media" to be any technology or technologies used in the creation, deployment and display of graphical images to an end user via a computer display. Note that "creation" here means only the specification of the graphics for display, *not* the whole process of generating graphics from a program execution.

The problem in developing this evaluation model is that the area of software visualisation is still so new. As we are still exploring which visualisations are useful for understanding software, it is difficult to pinpoint exactly what capabilities a medium must provide. If we limit ourselves to supporting only current software visualisations, our model is unlikely to remain relevant when a visualisation that demands new capabilities of our medium is conceived. The most practical way to address this issue is to try and identify all of the ways in which information can be visually encoded. In identifying the software visualisation designers' palette, we can evaluate a medium not just against what is required in software visualisations today, but also against what we can expect to be required in the near future. Fortunately, much of this work has already been done in the field of information visualisation. This is briefly described in section 3.1.

It is also important to explore the types of visualisations that are in use today. We need to examine existing software visualisations and utilise the existing taxonomies of software visualisations that have already been developed in the literature. This is covered in section 3.2.

3.1 Information visualisation

The field of Information Visualisation has steadily developed a comprehensive analysis of how information can be best encoded in visual form. However, our interest here is not with what decisions to make when constructing visualisations, but with the visual pallet that should be available to visualisation designers. The basis for this work was founded by Jacques Bertin in his *Semiology of Graphics* (Bertin 1967/83) in 1967. This classic work only dealt with "that which is...on a sheet of white paper" (Bertin 1967/83) in defining the pallet, which was a reasonable limitation for the time, but obviously needs extending for our purposes. Building from this work, the Information Visualisation community have extended the pallet to include a vast array of possibilities afforded by modern graphics technology. The evaluation model incorporates these additions, drawing primarily from the summary sections of "Readings in Information Visualization: Using Vision to Think" by Card et al. (Card, Mackinlay & Shneiderman 1999). In addition, the model also utilises the discussion and summaries from the text "Information Visualization: Perception for Design" by Ware (Ware 2000).

3.2 Software visualisation

Because we are evaluating the capabilities of a graphics medium, we are primarily concerned with the end product of the software visualisation process; in other words, the visual depictions of software. However, taxonomies of software visualisation have often focused on other aspects of the visualisation process. While none of these really serve our requirements for a taxonomy of the types of graphical representation for software visualisation, Oudshoorn et al (Oudshoorn, Widjaja & Ellershaw 1996) briefly mention what would be the basis for such a taxonomy. They break down the types of "data representations" for software visualisation into three "well-known types".

- Graph-based displays

  Graph-based displays are what Ware (Ware 2000) calls node-link diagrams. They are built from nodes that represent entities, and links between them that represent various relationships. Different node attributes (shape, colour, etc.) are used to encode information about the represented entities, and similarly for the links. The most common group of these software visualisations are those from the Unified Modeling Language (or UML). UML Collaboration diagrams are a classic example of node-link diagrams with their boxes for objects, and lines for method calls.

- Statistics-based displays

  Statistics-based displays move into the area of Scientific Visualisation. Such displays can use aggregations of data to draw statistical graphs, or can display data in a massed form by encoding it visually for rapid assimilation by users. A host of examples of statistical software visualisations can be found in De Pauw et al's work in creating an architecture for visualising program behaviour (Pauw, Helm, Kimelman & Vlissides 1993). Another example by Jerding *et al* (Jerding & Stasko 1995), called the "Execution Mural" shows graphically the entire record of messages passed in a program execution. Here the colours and spatial mapping may help a developer see patterns emerging.

- Source-code-related displays

  Source-code-related displays show source code in a more visually accessible form. This can include a zoomed-out source code view with additional information encoding such as Eick et al's SeeSoft (Eick, Steffen & Summer 1992), or in providing linking from a UML class diagram to the associated source code.

The method in which a visualisation medium supports these three types of software visualisations is included in the model.

3.3 Higher level capabilities

There are a number of capabilities that are desirable in a software visualisation medium, but which are not required to achieve base functionality. These capabilities, while less tangible and measurable, can determine how useful a technology really is in practice. This comes down to determining how well the medium supports us in what we want to do. For example, different media could present the same end product, while in one medium it was radically easier to create the content, than in the other.

If visualisations are to be created directly by a human user, there will need to be good tools for them to use. If the visualisations are to be created programmatically, the medium and the libraries that facilitate diagram creation will need to provide the capabilities in a logical and intuitive manner.

3.4 Limitations

Identifying what is left out is vitally important in any evaluation model. The following areas are not included in the model:

- We are not dealing with senses outside of vision. While the use of touch, smell and sound is potentially very interesting, these fall outside of the scope of this model. However this model could be extended at some later time to include them.
- We are only dealing with 2D computer display technology. The main reason is that this is the equipment that has become cheap and accessible, and is therefore the most useful as a vehicle for software visualisation at this time. Again, expanding the outlook of this model to other possibilities would be interesting, but is left as future work.
- Because we are not examining how to best create visualisations, we are not directly including end user usability in our model. However, end user usability does make some requirements on the technology (as opposed to what we do with it) so it will be considered where appropriate.
- Finally, other than general future-proofing through a focus on the requirements for basic information visualisation, we are not trying to predict where software visualisation will be heading in the future. It is quite possible that despite our best intentions, radically new visualisations will be created that require capabilities not included in this model.

3.5 The Model

The model for the evaluation of media for software visualisations is based on the above analysis. It brings together visualisation capabilities identified in the above mentioned sources, particularly from the work of Bertin (Bertin 1967/83), Card et al. (Card et al. 1999) and Ware (Ware 2000). Additionally, what we call *higher level capabilities* are included in the model, but there seems to be little existing work in this area for the model to draw from. To deal with this, a number of important higher level capabilities are included, but it is not claimed that this list is complete.

There are two major components to the model. The first component examines a medium's basic information visualisation capabilities, and the second looks at software visualisation-specific and the higher level capabilities. The basic capabilities are broken down into *graphical capabilities* and *interaction*. Software visualisation-specific and higher level capabilities are broken down into *integration*, *higher level capabilities*, and *support for current software visualisations*.

While a detailed exposition of this model would be illustrative, the content of the model should become clear by simply applying this model in evaluating SVG. However, it is important to note that some of the reasoning behind criteria in the model may not be as clear as it would be if it were fully described here. Full details of the model can be found elsewhere (Duignan 2002).

In the following sections, SVG is evaluated point by point against the criteria of the model.

4 SVG's Basic information visualisation capabilities

4.1 Graphical capability

The first part of the model requires examining SVG's graphical capabilities. SVG performs well for most of the graphical capabilities. While it does not do particularly well in its support of the spatial substrate[3], SVG is very able to meet the remaining graphical demands of the model. SVG can compensate for its poor performance with spatial substrate capabilities through the heavy use of scripting.

4.1.1 The spatial substrate

Dimensional support SVG provides a two-dimensional coordinate system. There is no built-in support for representing additional dimensions. It is possible to implement three dimensions by calculating 3D to 2D coordinate conversions in scripts. SVG is not an ideal medium for 3D or higher dimensional use due to this lack of built-in support.

Axis folding SVG provides no built-in support for axis folding. Every visual element must have an exact position specified with coordinates. Axis folding functionality could be implemented through heavy use of scripts.

Axis types SVG only supports straight axes. Other axes could be supported through scripting.

Axis distortions Arbitrary axis distortions cannot be established in native SVG. Again, this must be done via scripting. This can be achieved by ensuring that all use of coordinates on the distorted axis is mediated by specialised scripts. These scripts need to convert the virtual distorted axis values onto the real coordinates that are used in the native SVG. However, SVG does allow the specification of “transformations” on a coordinate system. These include moving, (universal) scaling, rotation, or skewing a coordinate

[3]A term used to describe the underlying *spatial* characteristics of a graphic

space. This could be used to provide only simple axis distortion effects. These do not provide the programmer with a single global coordinate space with flexible distortions inside it. Rather, this cuts the space into a number of different coordinate spaces, each with their own different distortions.

Viewpoint control by system Pan and zoom changes can be specified declaratively in SVG. This is done by using animate elements that control transformations on the root SVG graphic. These animations can be staggered to create the moving viewpoint effect. Scripting can also control the "currentScale" and "currentTranslate" values to get the same effect.

Recursion SVG provides explicit support for subdivided space by allowing SVG elements to be embedded within each other recursively.

4.1.2 Marks and their properties

Size Size can be specified when shapes and paths are created. SVG has a competent scaling system.

Colour SVG has a sophisticated colour system that includes support for the International Colour Consortium's colour profile standard (International Color Consortium 1999). SVG allows a full range of colour specification as well as providing control via Cascading Style Sheets (W3C 1999).

Orientation Orientation can be specified via SVG's competent translation capabilities.

Shape SVG allows the definition of paths that can include cubic and quadratic Bézier curves and elliptical arcs, as well as straight lines.

Points, lines, and areas SVG allows for exact positioning of points in two dimensional space. Lines can be specified as described above. Areas can be defined by a path description and used for masking and controlling the area of graphic effects.

Filter Effects SVG allows a wide variety of filter effects.

Transparency SVG allows the specification of the transparency of any visible element.

4.1.3 Temporal encoding

Encoding time SVG has a timing model to allow changes to graphics over time. The animate, set, animateMotion, animateColour, and animateTransform elements all take timing attributes so they can be cued at the appropriate points. Also, scripting allows programmatic control of the graphic.

Encoding identity The capability to specify animation declaratively (as opposed to manually setting values to fake animation) makes it easy to have visual elements move smoothly — and thus preserve identity.

Variation of retinal encodings Almost all graphical attributes of elements in an SVG graphic can be changed declaratively with the animate and set elements.

4.2 Interaction

SVG is almost entirely dependent on scripting for interaction. Having said this, with scripting SVG is capable of a wide range of interaction types and is highly flexible.

Graphic malleability (after creation) Only with the addition of scripting can the structure and content of SVG graphics be changed at runtime. However, scripting can alter almost any aspect of the graphic arbitrarily at runtime. SVG's declarative modification elements (such as *set*) can only modify the attributes of the elements that are already in the document when it is first rendered.

Events The SVG specification covers a full range of pointer events, as well as focusin, focusout, and activate that could be driven from a keyboard. However, full keyboard support is not part of the standard. There are a series of event types cued when things happen to the SVG document, such as loading, resizing and closing. The standard also specifies DOM events which are triggered when the document is modified via the DOM.

Computation The only computation that can be done in native SVG is to allow alternative viewings to suit an SVG renderer's capabilities. A switch element allows conditional processing of parts of a SVG document based on these capabilities. Of course, the inclusion of scripting allows full computational abilities.

User notation SVG has no in built-in capability to allow users to annotate graphics. However, this can be provided with scripting.

View refinement / navigation SVG renderers are expected to provide pan and zoom controls. The Adobe viewer does this via the context menu (right click) or by holding down the ALT or CTRL keys and left clicking.

Information hiding Graphical SVG elements have a visibility attribute that can be set to "hidden". They can be unhidden by setting the attribute to "visible".

Time control Time in an SVG graphic begins once the SVG is loaded. While animations can be restarted through native SVG event handling, accurate control of the "current time" can only be controlled via scripting. An SVG element has a setCurrentTime operation that can be passed the new "current time" by using the DOM.

4.3 Performance

SVG seems to have acceptable performance for small to medium-sized graphics. Scripting and interaction seem to impose a large performance overhead.

Scalability Increasing the complexity of of a graphic will always have an impact on performance, but there is no indication that there are any inherent scalability problems in the graphic's specification of the SVG standard. On the other hand, SVG might have a weak point as scripting may begin to show serious scalability problems. This is because it is interpreted at runtime which imposes an additional computational overhead.

Current implementations The only fully featured implementation is the Adobe SVG viewer. It has now reached version 3 and its performance has steadily improved with each release. The viewer's current performance seems comparable to Macromedia Flash. While no solid empirical tests were conducted, very large versions of the message mural inspired example shown in figure 7 were tested. On a Pentium 4 1700MHz, 512 Megabytes of RAM running Windows 98 with the Adobe web plug-in, it performed fairly well. The test was fully scrollable and zoomable with fair responsiveness when there were up to 20000 line elements. Once the test was pushed up to 32000 line elements it became unusable, functioning with large delays. While drawing only straight lines is far less complex than curved shapes or text, this shows that even the current viewer is capable of displaying fairly complex diagrams. Another static example is a large class diagram from the Apache Batik project. Even though it is very large, this graphic could be panned with ease. Zooming in and out worked, but with some delay. While Adobe's plug-in performed quite well with these static graphics, with interaction and animation examples it seemed to struggle a little.

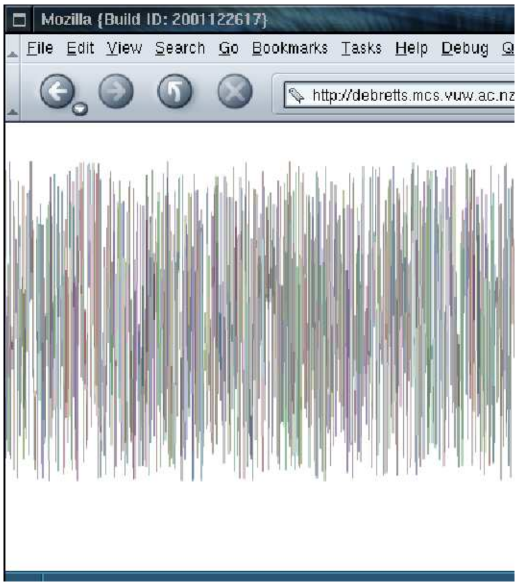

Figure 7: An SVG visualisation of random data mimicking Jerding and Stasko's message mural (Jerding & Stasko 1995).

5 SVG's software visualisation-specific and higher level capabilities

5.1 Integration

SVG has a huge strength in how it integrates with other technologies. Due to its XML basis, as well as it being developed for the web, it has unique advantages over other graphics formats.

Creation mechanism Since SVG is based on the plain text XML standard, there are a wide range of options for creating graphics with it. A SVG document can be hand-crafted in a text editor. A vector graphics drawing program can be used such as Adobe Illustrator (Adobe Systems 2002*a*). XSLT can be used to create an SVG document from an XML data source. Standard XML tools can be used to create the diagram programmatically.

Deployment SVG already has the MIME (Internet Engineering Task Force 1996) type of image/svg+xml (W3C 2001*b*). MIME types are a standardised way of declaring and transmitting non ASCII file types over the standard ASCII protocols of the Internet[4]. By using this MIME type, SVG can be transferred by existing web servers over the Internet as requested. Web pages can embed or link to SVG content for easy access. SVG can also be sent via email, or distributed in all other typical forms. Users will currently need an SVG viewer installed on their machine, but these are free and easily available. In the future, SVG will likely move to being included inline in XHTML, and web browsers will render SVG natively. Indeed, Mozilla already supports this (Mozilla.org 2002). However, SVG can not easily support live program software visualisation viewing. This is discussed further in section 6.2.

Linkages with other technologies at display time SVG is very strongly integrated with scripting. The scripting language to be used is not dictated by the SVG specification, and it can therefore integrate with any scripting language and implementation needed. The most popular choice is a variation on javascript or ECMAScript. Once the scripting language is specified, the SVG document can contain script elements as well as script event attributes in the chosen language. SVG integrates well with the web. Any visual element can be a hyper-link to any other web resource.

View coordination with other visual media SVG is dependent on scripting to coordinate multiple views across media. The only limitation is in the scripting environment used in an implementation. In principle, this is not limited by SVG. However, in practice, multiple SVG views can be coordinated within a web browser's scripting environment. Testing with interaction with other media (e.g. Java) was not conducted.

5.2 Higher level capabilities

SVG is disappointingly lacking in higher level capabilities. After becoming accustomed to programming in well developed user interface frameworks, moving to SVG is a shock. SVG is relatively low-level. This is fine for working with simple graphics, but becomes troublesome when applied to more complex applications. Again, scripting is necessary for any hope at achieving these capabilities. The only exception to this is in built-in graphical capabilities where SVG performs rather well.

Higher order graphics SVG provides support for a number of basic shapes including rectangles, circles, ellipses, lines, polylines, and polygons. At specification they can all be given appropriate attributes such as position, width and height. Lines and curves can be used to build non-standard shapes. These non-standard shapes can be specified as "symbols" and used repeatedly in a graphic. Text is well supported in SVG with provision for multiple languages and fonts.

[4]MIME stands for Multipurpose Internet Mail Extensions, which betrays its origins as a standard for email attachment types.

Sophisticated layering and filtering (as already mentioned) are part of the standard. Gradients and patterns (textures) are supported.

Data/display independence There is little room in SVG to store data independently from display. An SVG document is only designed to describe a graphic, not to store other forms of information. There is a metadata element that can be included to describe any SVG element, although this is meant to be used for descriptive information rather than raw data. However, scripting can be used to store the data while it creates the content of the SVG on the fly.

Referencable Entities/Objects Every graphical element in an SVG graphic can be referenced via the DOM or XML ID attributes.

Layout constraints SVG provides no support for layout constraints.

Structure Elements can be structured with group (g) elements. However, these groups determine layering order (which elements lie on top of or below others) and so are very limited in their use for grouping elements logically, rather then visually.

5.3 Support for current software visualisations

SVG is able to display all three of the common forms of software visualisation.

Node-link diagrams SVG can cope well with drawing node-link diagrams. However, the lack of layout constraints makes SVG a much less convenient medium to work with than it could be.

Statistical diagrams The performance of the message mural inspired test has already been described in the performance section of this evaluation. This indicated that SVG was quite capable of displaying non-trivially sized raw data displays. It would not perform well in more interactive and dynamic raw data displays. Aggregated data is displayed with ease.

Source-code-related diagrams While SVG can display text, most implementations would probably have trouble rendering the huge quantities of text that make up the source code of large programs. Also, SVG is missing higher level text support such as word-wrapping etc. However, SVG's integration with HTML more than makes up for this in many cases due to HTML browsers' relatively good performance with text. For many source-code-related diagrams, it would be possible to have the graphical segments in SVG while the source code segments are in HTML. The SVG would simply have to be embedded in the appropriate places in the HTML, with scripting used to let the HTML and SVG work together. However, it would be possible to conceive visualisations that could not be divided like this, and they would push SVG renderers to their limits.

6 Discussion

Now that we have briefly covered how SVG performed in each of our model's capabilities, a more detailed discussion of the interesting points that were raised can be undertaken.

6.1 SVG creation

SVG has a number of interesting options available for programmatic generation and creation.

6.1.1 XSLT

XSLT is a declarative style sheet language for styling XML documents. It is often used for XML data conversion, moving data from one XML document type to another. As such, it would seem to be a logical choice for creating SVG software visualisations derived from an XML source. However, XSLT is primarily a stylesheet language, and its declarative nature makes it awkward for the types of processing needed for visualisation creation. While XSLT is good for simply changing the form of XML data, it would be difficult to use XSLT to implement some of the complex layout algorithms needed to convert raw data into visualisations. However, XSLT can be used to generate script content for an SVG that will be run automatically when the SVG is loaded. This SVG script content can then build and control the graphical elements required in the visualisation.

What this amounts to is developing visualisation creation transformations in two contrasting environments. One, (server based) uses XSLT. The other (client based) uses script. The end result is a transformation system of far more complexity than is needed. This is intensified by the fact that the scripting element (likely some variant on javascript) is not standardised across SVG renderers, web browsers, and platforms. Even for simple scripting, moving between the Adobe SVG viewer in Internet Explorer on Windows 98 and the Adobe SVG viewer in Mozilla on Linux often showed strange incompatibilities and inconsistencies. Additionally, a generalised scripting development environment (as it needs to be for multiple platforms etc.) typically fails to give the support of a more robust programming environment. So while XSLT may have a place in some SVG software visualisation generation systems, it is unlikely to offer a complete solution.

6.1.2 Server side generation

The other alternative is to complete the generation of the visualisation entirely on the server side. In this scenario, the environment is controlled much more tightly by the developer. As most of the work is done on the server in the developer's choice of programming language, rather than on an undetermined client system with an unpredictable setup, things are simplified greatly. Also, using server side tools allows the use of quality XML toolkits such as the Apache project's Xerces (The Apache Software Foundation 2000) and Batik (The Apache Software Foundation 2002).

6.1.3 Writing SVG by hand

Creating a complete software visualisation by hand is possible, but time consuming and difficult. Editing SVG at the source code level is more suited for either building basic components for use in programmatic creation, or for fine-tuning SVG output from a drawing or conversion tool.

6.1.4 Drawing tools

Using Adobe Illustrator (Adobe Systems 2002*a*) proved to be an easy way to create one-off SVG diagrams. However, the code it produces tends not to be as conceptually tidy as what would be produced by a human. For this reason, this is not an ideal route if lots of work needs to be done to the SVG after export (such as the addition of complex scripting).

6.1.5 UML CASE tool file conversion

Finding a way to create SVG from a CASE tool designed for creation of software visualisations is an attractive option for some situations. However, this is limited to creating only the well known, supported visualisation types. This would be too limiting for some applications.

6.2 Streaming SVG

SAX's simple event triggering design makes it ideal for streaming conversion of XML content. A form of streaming could be used to animate a live visualisation of a program execution on a server. However, doing this with SVG is problematic. While some implementations of SVG renderers will support a crude form of streaming in that they display the SVG as it becomes available, this is too limiting for running even moderately interesting animations. This is because a stream of SVG can only add to the SVG that has already been rendered at the client — not manipulate or change what is already there. To do serious live program visualisation with SVG over the Internet, another agent would need to be present on the client. This agent would read an event stream and modify the SVG via the DOM. Perhaps this could be done via scripting, but Java would be a more attractive option. However, this would raise the question of why not just use Java's graphical capabilities directly.

6.3 The importance of scripting

A theme emerging from the evaluation so far has been SVG's reliance on scripting to support many of the capabilities that would prove useful for software visualisation. The "dirty" nature of scripting, as implemented in practice, combined with the fact that the SVG specification defines no standard scripting language, means that the developer must be very careful. While doing simple scripting work in a homogeneous environment is acceptable, the implementation of the complex script capabilities required for SVG software visualisation could cause problems. However, if SVG with scripting was used, the following could be achieved:

XSLT and client side generation The details of this have already been discussed in section 6.1.1.

3D with shadows Adobe's Chemical Markup Language example (shown in figure 8) shows a 3D visualisation with shadow effects on SVG's 2D plane. This shows that SVG can be used simply as a raw display medium while graphics libraries are implemented on top in a scripting language. However, travelling too far down this road would certainly raise serious performance problems. At some point it will need to be asked, why not use some other medium that supports these capabilities natively without the performance hit of interpreted scripting? Other options could include 3D markup languages such as VRML (The Web3D Consortium 1997), the emerging X3D (The Web3D Consortium 2002), or alternatively Java with Java3D.

Improved node-link diagrams While SVG is more than capable of displaying node-link diagrams, the underlying representations that are possible are far from ideal. Unfortunately, for reasons discussed shortly in section 6.5, SVG does not allow a line to be specified as linking two other graphic entities. This becomes a concern when one or more of these entities needs to be moved. For example, a user might want to modify the layout for clarity. If the graphical entity is moved, any lines that linked to it will now be orphans. The only way to address this is to track the "nodes" and "links" in scripts, and demote the SVG further towards the the role of a raw medium, rather then a structured and developer-friendly graphics language.

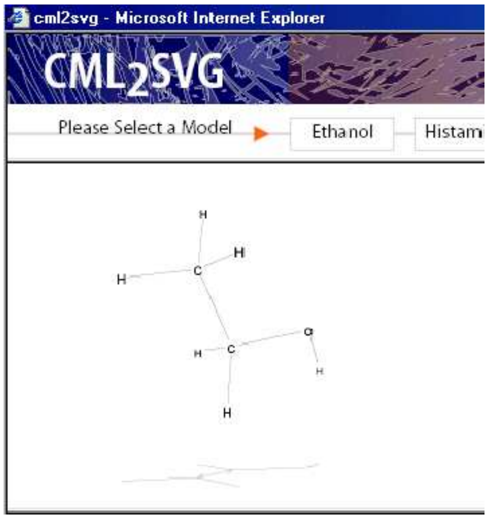

Figure 8: A scientific visualisation of a chemical implemented in SVG. The visualisations were created with XSLT from an XML source. (Adobe Systems)

Data display independence Again, data display independence is reliant on scripting to do the work behind the scenes, while the SVG takes a passive role. In this case, creating a visualisation would involve determining the logical elements that will be present in the diagram. The SVG should probably arrive at the client in an empty or basic initial state, except for the scripts that would take control when the SVG is viewed. Any interaction would need to be fully mediated by the scripts, which would update their state, and then feed this through to the display (implemented by the SVG).

Logical structure Logical structure, as opposed to the largely graphically determined structure imposed by SVG, would also be facilitated by the data display independence just outlined.

6.4 Creating objects from symbols

SVG seems to have a strange weakness in how symbols are created and used. To make SVG easier to use (and more efficient for downloading) recurring groups of SVG elements can be defined as a symbol. These symbols can then be used as many times as needed in a graphic. This is done by specifying a *use* tag, and giving coordinates for its location in its attributes. Unfortunately, there seems to be no easy way to specify other customisations at creation time. For example, in creating the rectangles at the top of a UML sequence diagram, all that needs to be customised is the value of the text element that displays the object's name. However, this cannot be done through the declarative means of the SVG.

6.5 Layout constraints

A weakness that seems to be a result of SVG's philosophy is the lack of any support for layout constraints (Tirtowidjojo, Marriott & Meyer 2000). SVG's philosophy seems to be that all graphical elements should be placed in exact coordinates. In other words, SVG makes no effort to help in the intelligent placement of objects. The programmer must do all the work. For example:

- There is no way to automate line wrapping for text that runs off the screen or outside an area.
- There is no way to specify an element's coordinate values as being an expression to be calculated.
- There is no way to specify behaviour rules for specific elements (e.g. different zoom factors when a zoom is triggered).

These limits can be overcome through diligent script hacking, but it is unfortunate that they were not included in the base SVG specification.

7 Alternatives to SVG

While this is primarily an evaluation of SVG as a medium for software visualisation, it is also prudent to at least mention some of the alternatives. However, this discussion will be limited to a brief comparison with SVG. A more complete exploration would require measuring each of the alternatives against the evaluation model in order to usefully develop the points of comparison.

7.1 Macromedia Flash

Flash provides very similar basic capabilities to SVG. In fact, there is a working (but still developing) conversion script from SVG to Flash's file format (Freshmeat 2001). However, Flash has a number of crucial differences. Firstly, while Flash has a vector based file format, it is a binary format. While this can result in the possibility of mildly better performance, it has a major draw back. Creating new authoring systems is made very difficult. For both hand-authoring and programmatic generation, Flash has only a few possible choices for tools (Ming 2002, Macromedia 2002*b*, Macromedia 2002*a*). Compared to the breadth of support for authoring SVG via general XML tools, Flash's options are limited. Additionally, because SVG is backed by standard XML, high quality authoring environments already have SVG support (Adobe Systems 2002*a*), and more will probably follow. An interesting development in the latest version of Flash (Flash 5) is that it now has direct support for XML data communication (Macromedia 2002*c*). What this means is that a Flash graphic being viewed on a client machine can now access XML data from the web as needed, as well as send XML to servers. Importantly however, this does not overcome Flash's central weakness. The XML integration is only an addition to its capabilities, *not* its file format. This means that developers are still limited to using the limited set of tools that are available for Flash authoring. Finally, one major advantage of Flash is that it has support for streaming "movies", which would be useful for live program visualisation.

7.2 VRML and X3D

Another option is to use VRML (The Web3D Consortium 1997), or the still emerging X3D (The Web3D Consortium 2002) as media. Like SVG, these technologies are for the display of graphics over the web and are also based on markup languages. However, they would probably not be ideal for the two dimensional applications that dominate much of the software visualisation field. Further examination of these media would be necessary in order to provide further comment.

7.3 Java

Another option would be to utilise Java. Java Applets (Sun Microsystems 2002) allow programs written and compiled in Java to run in a web browser on (potentially) any hardware/software platform. This is a very interesting option, as it provides a sophisticated graphical user interface toolkit, as well as full network capabilities for intelligent communication with a server after deployment. However, Java is clearly a heavy-weight solution, while SVG is the light-weight alternative. SVG allows the developer to deploy simple descriptions of graphical content, rather then executable programs. For small software visualisations, such as some UML diagrams, SVG would be a perfect choice. Larger, more dynamic visualisations would probably be better served by Java applets. Interestingly, SVG content can be included in Java applications through the Apache Batik libraries (The Apache Software Foundation 2002), so SVG visualisations could be displayed from within larger Java visualisations.

8 Conclusions

SVG has a number of factors that make it interesting for use in software visualisation. Firstly, it is a W3C standard (W3C 2001*b*), which helps ensure that it is open and encourages community support. This, coupled with SVG's XML basis, has encouraged the rapid growth of SVG tools for both developers and users. SVG's integration with other web standards allows it to leverage existing technology implementations, and gives developers the advantage of reusing their current knowledge. SVG is particularly useful where we want to view software visualisations over the web (Marshall et al. 2001) as it is specifically designed for this sort of deployment. To determine whether SVG was really useful for software visualisation, we developed a model to evaluate it.

Our principled model leverages the general understanding of what techniques are and can be used in software visualisations, and expressed them as capabilities that are desirable in a software visualisation medium. Additionally, our model includes a number of higher level capabilities that extends the model to evaluate how the medium supports developers' needs.

The results of applying this model to SVG are mixed. It is clear that SVG can meet the requirements of a software visualisation medium adequately. It seems to be primarily attractive for light-weight web-based software visualisations. Under these conditions, SVG is a good candidate, as it integrates well with other web technologies, and can be created dynamically with standard XML tools.

However, the core of SVG is disappointingly low level, forcing developers to utilise scripting to accomplish anything beyond the most basic tasks. In particular, the lack of an in-built layout constraints system, or any ability for data display independence are missed. For complex and highly interactive visualisations, we would advise a more supportive medium such as Java.

8.1 Current and future work

To aid in the creation of software visualisations in SVG, we are currently developing a Javascript library that provides some of the higher level functionality that is not built into SVG. This library provides software visualisation-specific functionality that should alleviate some of SVG's weaknesses for use in this area.

In addition to future extensions to this library, future work could include extending the model to examine three-dimensional media, and other communication means such as sound. Additionally, other potential software visualisation media such as Flash and Java could be evaluated against the current model, to provide a strong basis for comparison.

References

Adobe Systems (2002*a*), 'Adobe illustrator 10'. `http://www.adobe.com/products/illustrator/`.

Adobe Systems (2002*b*), 'SVG Zone'. `http://www.adobe.com/svg/basics/intro.htm`.

Bertin, J. (1967/83), *Semiology of Graphics: Diagrams, Networks, Maps*, University of Wisconsin Press, Madison, WI. W. J. Berg, Trans.

Card, S. K., Mackinlay, J. D. & Shneiderman, B. (1999), *Readings in Information Visualisation: Using Vision to Think*, Morgan Kaufmann.

Duignan, M. (2002), Evaluating scalable vector graphics for use in software visualisation, Master's thesis, Victoria University of Wellington.

Eick, S. G., Steffen, J. L. & Summer, E. E. (1992), 'Seesoft-a tool for visualizing line oriented software statistics', *IEEE Transactions on Software Engineering* **18**(11), 957–968.

Freshmeat (2001), 'svg2swf by Rob Lanphier'. `http://freshmeat.net/projects/svg2swf/`.

International Color Consortium (1999), 'Document icc.1a:1999-04 addendum 2 to spec. icc.1:1998-09'. `http://www.color.org/ICC-1A\_1999-04.PDF`.

Internet Engineering Task Force (1996), 'Multipurpose internet mail extensions: Mime, rfcs 2045 - 2049'. `http://www.rfc-editor.org`.

Jerding, D. F. & Stasko, J. T. (1995), The information mural: A technique for displaying and navigating large information spaces, *in* 'Proceedings of the IEEE Visualization '95 Symposium on Information Visualization', Atlanta, GA, pp. 43–50.

KDE (2002), 'KSVG', `http://svg.kde.org/`.

Macromedia (2002*a*), 'Macromedia Flash 5'. `http://www.macromedia.com/software/flash/`.

Macromedia (2002*b*), 'Macromedia Flash File Format (SWF) SDK'. `http://www.macromedia.com/software/flash/open/licensing/fileformat/`.

Macromedia (2002*c*), 'XML Transfer and HTML Text Support'. `http://www.macromedia.com/software/flash/productinfo/features/new\_feat%ures\_tour/14xml\_html.htm`.

Marshall, S., Jackson, K., McGavin, M., Duignan, M., Biddle, R. & Tempero, E. (2001), Visualising reusable software over the web, *in* 'Information Visualisation 2001. Australian Symposium on Information Visualisation', Vol. 9, pp. 103–111.

Ming (2002), 'Ming - an SWF output library and PHP module'. `http://www.opaque.net/ming/`.

Mozilla.org (2002), 'Mozilla SVG Project'. `http://www.mozilla.org/projects/svg`.

Oudshoorn, M. J., Widjaja, H. & Ellershaw, S. K. (1996), Aspects and taxonomy of program visualisation, *in* P. D. Eades & K. Zhang, eds, 'Software Visualisation', Vol. 7, World Scientific, Singapore, pp. 3–26.

Pauw, W. D., Helm, R., Kimelman, D. & Vlissides, J. (1993), An architecture for visualising the behavior of object-oriented systems, *in* 'OOPSLA 1993', pp. 326–337.

Sun Microsystems (2002), 'Applets'. `http://java.sun.com/applets/index.html`.

The Apache Software Foundation (2000), 'Xerces Java Parser'. `http://xml.apache.org/xerces-j/index.html`.

The Apache Software Foundation (2002), 'Batik SVG Toolkit'. `http://xml.apache.org/batik/`.

The Web3D Consortium (1997), 'The Virtual Reality Modeling Language'. `http://www.web3d.org/Specifications/VRML97/`.

The Web3D Consortium (2002), 'Extensible 3D (X3D) Graphics'. `http://www.web3d.org/x3d.html`.

Tirtowidjojo, J. J., Marriott, K. & Meyer, B. (2000), Extending svg with constraints, *in* 'Sixth Australian World Wide Web Conference'.

W3C (1999), 'Cascading style sheets, level 1. w3c recommendation 17 dec 1996, revised 11 jan 1999'. `http://www.w3.org/TR/REC-CSS1`.

W3C (2001*a*), 'Extensible markup language (xml) 1.0 (second edition). w3c recommendation 06 october 2001'. `http://www.w3.org/TR/2000/REC-xml-20001006`.

W3C (2001*b*), 'Scalable vector graphics (svg) 1.0 specification . w3c recommendation 04 september 2001'. `http://www.w3.org/TR/2001/REC-SVG-20010904/`.

W3C (2001*c*), '*SMIL Animation* w3c recommendation 04-september-2001', `http://www.w3.org/TR/2001/REC-smil-animation-20010904/`.

Ware, C. (2000), *Information Visualization: Perception for Design*, Morgan Kaufmann, San Fancisco.