Encapsulating and Manipulating Component Object Graphics (COGs) using SVG

Alexander J. Macdonald
Document Engineering Laboratory
School of Computer Science
University of Nottingham
Nottingham, NG8 1BB, UK
ajm@cs.nott.ac.uk

David F. Brailsford
Document Engineering Laboratory
School of Computer Science
University of Nottingham
Nottingham, NG8 1BB, UK
dfb@cs.nott.ac.uk

Steven R. Bagley
Document Engineering Laboratory
School of Computer Science
University of Nottingham
Nottingham, NG8 1BB, UK
srb@cs.nott.ac.uk

ABSTRACT
Scalable Vector Graphics (SVG) has an imaging model similar to that of PostScript and PDF but the XML basis of SVG allows it to participate fully, via namespaces, in generalised XML documents.

There is increasing interest in using SVG as a Page Description Language and we examine ways in which SVG document components can be encapsulated in contexts where SVG will be used as a rendering technology for conventional page printing.

Our aim is to encapsulate portions of SVG content (SVG COGs) so that the COGs are mutually independent and can be moved around a page, while maintaining invariant graphic properties and with guaranteed freedom from side effects and mutual interference. Parellels are drawn between COG implementation within SVG's tree-based inheritance mechanisms and an earlier COG implementation using PDF.

Categories and Subject Descriptors
E.1 [Data]: Data Structures — *Trees*; I.7.2 [Document and Text Processing]: Document Preparation — *Markup Languages*; I.7.4 [Document and Text Processing]: Electronic Publishing.

General Terms: Algorithms, Documentation.

Keywords
XML, SVG, PDF, parameterization, component object graphics

1. INTRODUCTION
The World Wide Web Consortium (W3C), set up a working group in 1998, to draw up draft proposals for Scalable Vector Graphics (SVG) in response to a need for better rendering of material such as line diagrams, schematics, and maps. At present there is only limited native support within Web browsers for rendering SVG— for the most part popular browsers such as Internet Explorer need to install an SVG plug-in supplied by Adobe Systems Inc. Although the syntax of SVG is XML based, the semantics of its graphics model are similar to those of PostScript and PDF. In common with these two languages, SVG combines graphical sophistication with the ability to place text strings accurately.

Currently, with SVG 1.1, the major use is for 'vector graphic inserts' into conventional Web pages. But with SVG 1.2 close to approval there is now the prospect of SVG acquiring a page model (via *pagesets* [1]). This, in turn, means that SVG could become a viable rendering technology for a Page Description Language (PDL) for conventional hard-copy printing. Danilo and Fujisawa [2] have already surveyed some of the problems, ranging from frame buffer size to filters and ICC colour spaces, which need to be addressed before SVG can become a satisfactory PDL. Our concerns run even deeper: we want to develop ways to determine the graphical state at the root of a given SVG subtree sufficiently accurately, and to encapsulate it in some way, so that the subtree can then be extracted and used elsewhere.

FINAL DRAFT of paper accepted for:
DocEng'05 November 2–4, 2005, Bristol, United Kingdom

1.1. Previous work and present aims
In a previous paper [3] we developed the idea of Component Object Graphics (COGs) in which encapsulated graphic objects can have their page positions and certain needed resources (e.g. fonts) declared at the head of a suitable data structure. Our initial implementation of COGs was in PDF and the PDF data structure we used for encapsulation was the FormXobject.

In the **C**omponent-**O**bject **G**raphic model the pages in a PDL are no longer described as monolithic page streams (where the effect of each operator depends on the operators that have been executed before it) but rather as independent graphical objects, or COGs. A COG is a self-contained block that describes how to draw itself in a manner independent of any other COG appearing before it, while ensuring also that it does not affect the appearance of any COG imaged after it. In other words, adding or removing a COG from a page should have no visible side-effect on any other COG on that page.

A key advantage of PDF COGs is that a single COG instance can be shared throughout a document. Our COG model allows us to encapsulate PDF inside a FormXObject structure while the PDF graphical system (and its capability for saving and restoring graphic state) make it possible to establish a relative co-ordinate system within the COG and to make clear at the head of the FormXObject dictionary the resources the COG will use. The sequence of COGs on a page is invoked by executing the corresponding FormXObjects using the PDF Do operator.

The COG model is by no means limited to PDF. It is potentially applicable to any PDL that offers the possibility of encapsulation and re-use of material. So, if SVG 1.2 will truly be a future alternative to PDF we wanted to find out whether SVG COGs were feasible and, if so, how the tree-based inheritance mechanisms for graphic properties might lead to new problems and opportunities as compared to the more conventional document-based and page-based inheritance properties of PDF.

We now look at ways in which SVG COGs can be defined, manipulated on the page and made shareable.

2. MOTIVATION FOR THIS WORK

Most PDLs have the problem that when small alterations are needed to the final appearance of a document it is often necessary to regenerate at least a whole page (and sometimes the whole document) from scratch. Similar problems of "monolithic" transformations apply to XML documents that are transformed by XSLT scripts. Any small change to the output effect will generally require that the XSLT script be modified and then re-applied to the entire document.

For certain sorts of application such as product catalogues and posters this monolithic approach is not desirable. A page of a catalogue typically consists of several component blocks of material and each of these should correspond to a separate COG of the sort we have described. One needs the ability to move these objects around the page for best aesthetic effect and to have each COG be equipped with clearly identifiable declarations for the resources used.

3. IMPLEMENTATION OF SVG COGS

Turning to SVG, we find that its syntax is rather more transparent than that of PDF; properties such as fill colour or line width are expressed by style attributes on the nodes that draw the object, rather than being an ordered sequence of commands to create the effect. The listings in Figures 3.2 and 3.3 illustrate the different PDF and SVG codings required to render the text 'Helvetica House' in 12pt Times-Roman.

```
/F1 1 Tf
12 0 0 12 100 100 Tm
(Helvetica House) Tj
```

Figure 3.1 — PDF code to render 'Helvetica House'

```
<text x="100" y="100" font-family="Times"
font-size="12pt">
Helvetica House
</text>
```

Figure 3.2 — SVG equivalent of the PDF code in Figure 3.1

Unfortunately (as far as node extraction is concerned) SVG allows graphical properties to be inherited down the tree, and so the appearance of the `<text>` node in the above example may depend upon its position within the tree. However, SVG provides a grouping node (`<g>`) that can be used to group together sets of nodes that share common properties. As an example, the `<text>` node in Figure 3.3, when rendered, will appear twice as big as identical node in Figure 3.2 due to the scale operation attached to the entire group. This same scale operation will also affect the *x* and *y* positioning, which will change from (100,100) to (200,200).

```
<g transform=”scale(2)”>
<text x="100" y="100" font-family="Times"
font-size="12pt">
Helvetica House
</text>
</g>
```

Figure 3.3 — Scaled text node in SVG

On the face of it these `<g>` groupings give us a means of encapsulating portions of SVG to form SVG COGs However, great care is needed because the possible nesting of groups within groups means that the inner grouping inherits the properties of the outer group. These problems are not insurmountable (it is a simple matter to walk back up the tree towards the root node and concatenate all the properties together) but they do complicate the process of extracting content.

The same problem can be found when trying to reinsert the extracted graphic into another SVG document. The appearance will depend upon where the extracted grouping is inserted into the destination SVG document tree. Unless all the rendering properties are explicitly specified, as attributes, on the extracted graphic it will inherit these values from its newly-acquired ancestors in the destination tree and may therefore not render as desired.

SVG is also beset by the same problems encountered in PDF regarding extracted content. While we can see that SVG has a neater and more regular syntax than PDF, it is still possible for the various commands that draw a particular graphical object to be located far away from each other within the SVG document. Thus the problems described in [3], in working out the exact graphic state, still apply here.

With all the above problems in mind it can be seen that there is a need for a COG-like model within SVG just as there was in PDF. Fortunately, as with PDF, SVG provides the necessary tools to implement such a model. The `<use/>` element defined in the SVG specification permits a reference to be made to an element that exists elsewhere in the document and to display that element as if the code being referenced existed at this point in the rendering tree. In programming language terms, it is the equivalent of a procedure call.

By storing the graphical content of each COG inside a `<g>` element labelled with a unique identifier we can then display this COG later on using a `<use>` element. Now if this COG definition existed as an immediate child of the SVG document root node then it would be rendered even if it was never referred to. To stop the COG definition from being rendered it can be placed inside a `<defs>` element. Any SVG located within a `<defs>` element remains a part of the Document Object Model (DOM) but the SVG renderer knows it should not render it until it is explicitly called out and used. A rendered SVG COG version of a menu for a restaurant called 'Helvetica House' is shown in Figure 3.4 along with part of the SVG code, in Figure 3.5, that defines and displays the heading COG.

Figure 3.4 — Helvetica House menu rendered in SVG COGs

3.1. Graphical independence of SVG COGs

By using the previously mentioned <g> element it is very easy to group together all of the graphical content of a COG. However, SVG 1.2 introduces the ref() matrix, which computes the inverse of the current viewport's transformation matrix. Thus if the SVG code is left unchecked it could effectively break out of the COG nature of the document and position itself using absolute page co-ordinates rather than relative to the origin of its parent. This can be prevented by additionally wrapping the COGs inside <svg> elements so that they each exist as a separate document fragment. This also means COGs are explicitly clipped to their bounding box, and so more closely match the behaviour of PDF COGs. Another benefit is that resources such as gradients and embedded fonts can be stored in the COG.

Because all positioning in SVG is done relative to the origin of the current coordinate system we can apply a transformation, to the group as a whole, which will alter that system. This allows the entire COG to be moved, scaled or rotated in any way whilst preserving the internal positioning of the contents of the COG. To demonstrate the graphical independence of the SVG COGs an ECMAScript was created that allows the user to move, scale and rotate any of the COGs in a document. In effect it mimics the effect of the PDF COG manipulation program (written in C++) that was described in [3]. It should be noted that while the Acrobat plug-in extends the Acrobat viewer (and so is available to all COG PDF documents) the ECMAScript must be included, or linked to, in the COG SVG document before it can be used.

It was encouraging to find that the ECMAscript SVG COG manipulator worked correctly and, as a given COG is moved around the page, all other COGs on the page are unaffected. This is demonstrated in Figure 3.6 where the ECMAscript COG manipulator has been used to shift the "OTHERS" COG in the Helvetica House menu. Also visible are shaded squares, which are controls to switch between translation, rotation and scaling of the selected COG.

```
<svg:svg id="Cog4e3e983e">
<svg:text x="100.375" y="21.87" font-
family="helvetica" font-size="30pt">
Helvetica House
</svg:text>
…
</svg:svg>
…
<svg:use xlink:href="#Cog4e3e983e"
transform="translate(96.625 32.23999)"
id="cog-instance-Cog4e3e983e-1"/>
```

Figure 3.5 — An SVG COG definition and a reference to it.

3.2. Creating SVG COGs

In addition to hand-coding SVG COGs we are presently in the process of converting several of our XSLT-based SVG code-generation facilities into a form such that they can generate SVG COGs. A further useful source of SVG COGs is also available to us because it has proved possible to modify Mong's PDF-to-SVG converter [4] so that it converts PDF COGs into SVG COGs.

4. CONCLUSIONS AND FUTURE WORK

All the effects of PDF COGs described in [3] have been successfully replicated in SVG . The use of <defs> and <svg> to encapsulate SVG code parallels the use of a FormXObject structure in PDF whereas SVG's <use> operator parallel's PDF's execution of a FormXObject via Do.

Note carefully that the simple SVG COGs we have created are

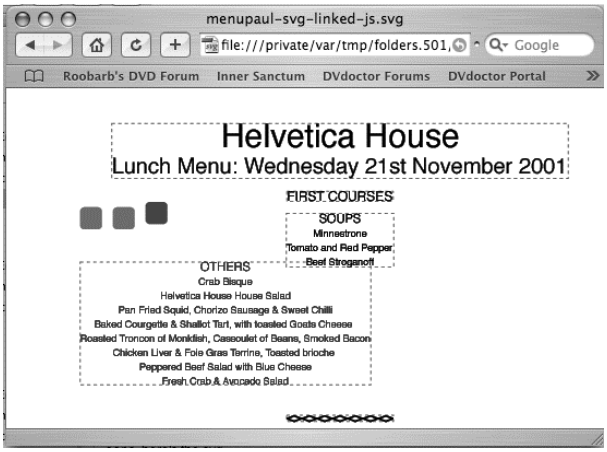

Figure 3.6 — ECMAScript demonstration of the graphical independence of SVG COGs

just like their PDF counterparts in being invariant in their appearance wherever they might be invoked. This behaviour is a consequence of the fact that PDF is a final-form PDL, with no facilities for parameterised late-bound adjustments to appearance.

By contrast, SVG is not a final-form PDL. It is parameterisable via XML attributes (see for example the <text> node in Figure 3.4). If a more powerful mechanism for parameter forwarding and binding can be devised one might achieve parameterised COGs. An example would be the insertion of formal parameters inside text strings to enable late-bound items such as the price of an item, or the start and finish dates of a special offer, to be substituted at the last possible moment, just before the page is rendered and printed. Work continues on exploring this exciting possibility.

5. ACKNOWLEDGEMENTS

Thanks are due to Hewlett Packard (UK) and EPSRC for supporting Alex Macdonald's PhD studentship. In particular we thank John Lumley and Tony Wiley of HP (UK) for technical insights and administrative help. Thanks are also due to Jon Ferraiolo of Adobe Systems Inc for information on SVG 1.2.

6. REFERENCES

[1] SVG 1.2 – Multiple Pages. http://www.w3.org/TR/2004/WD-SVG12-20041027/multipage.html

[2] Alex Danilo and Jun Fujisawa, "SVG as a Page Description Language" http://www.svgopen.org/2002/papers

[3] Steven Bagley, David Brailsford, and Matthew Hardy, ''Creating reusable well-structured PDF as a sequence of Component Object Graphic (COG) elements.,'' in *Proceedings of the ACM Symposium on Document Engineering (DocEng'03)*, pp. 58–67, ACM Press, 20–22 November 2003.

[4] Julius Mong and David Brailsford, ''Some experiments in using SVG as the rendering model for structured and graphically complex Web material,'' in *Proceedings of the ACM Symposium on Document Engineering (DocEng'03)*, pp. 88–91, ACM Press, 20–22 November 2003.