Fast and Efficient Client-Side Adaptivity for SVG

Kim Marriott
Monash University
Clayton, Vic. 3800, Australia

Bernd Meyer
Monash University
Clayton, Vic. 3800, Australia

Laurent Tardif
Monash University
Clayton, Vic. 3800, Australia

[marriott | berndm | tardif]@mail.csse.monash.edu.au

ABSTRACT

The Scalable Vector Graphics format SVG is already substantially improving graphics delivery on the web, but some important issues still remain to be addressed. In particular, SVG does not support client-side adaption of documents to different viewing conditions, such as varying screen sizes, style preferences or different device capabilities. Based on our earlier work we show how SVG can be extended with constraint-based specification of document layout to augment it with adaptive capabilities. The core of our proposal is to include *one-way constraints* into SVG, which offer more expressiveness than the previously suggested class of linear constraints and at the same time require substantially less computational effort.

Categories and Subject Descriptors

I.7.2 [**Document and text processing**]: Document Preparation, Markup languages, Scripting languages, Standards

General Terms

Standardization

Keywords

SVG, Scalable Vector Graphics, constraints, differential scaling, semantic zooming, CSVG, adaptivity, interaction

1. INTRODUCTION

The way graphics is presented on the web is currently being revolutionized by the Scalable Vector Graphics format SVG [7]. However, despite its considerable success, some important issues still remain to be addressed in future SVG standards and are the focus of ongoing discussion in the community. One of the main issues is that device capabilities are becoming more and more diverse: displays are used in all sizes and aspect ratios, ranging from small PDA and mobile phone screens to large conference room displays, resolutions vary accordingly, and some devices, like mobile phones, still do not offer color. This situation is made more difficult by the fact that individual viewers may have different requirements, both in terms of *what* they want to see (e.g. level of detail) and *how* they want to see it (for example, a vision impaired user may require a larger font size). Essentially this means that the designer of a document cannot anticipate the exact conditions in which it will be viewed, and so any *fixed* design may be unsuitable in some circumstances.

WWW2002, May 7–11, 2002, Honolulu, Hawaii, USA.
ACM 1-58113-449-5/02/0005.

Consider the case of a user viewing an organizational structure diagram on an intranet (Figure 1). The fully expanded diagram is of considerable complexity and so is unsuitable for small displays. Thus, if the diagram is to be viewed on a PDA screen, we cannot use the same layout. Instead we may want to view an adapted layout that initially only shows the top level of the hierarchy and allows the viewer to zoom in hierarchically, while adapting the layout of the individual hierarchy levels to the available screen size (Figure 2). We call this *semantic zooming*.

The present paper discusses the need for adaptive graphics documents and presents an extension of SVG which provides these capabilities. Based on our earlier work [1] we extend SVG by allowing constraint-based description of relationships between document components. This allows properties, such as object sizes and positions, font sizes, as well as spatial high-level relations (like shape alignment), to be dynamically computed at the time of viewing thus allowing client-side adaptation of the document.

It is instructive to consider the different reasons for requiring adaptation of graphical documents on the web:

- *Device Capabilities* Probably the most obvious reason for adaptivity stems from the fact that many different devices are in use to display web pages, the most important varying factors being screen size and aspect ratio. The simplest way to address this issue is linear scaling (normal zooming) which is provided in SVG 1.0. However, this is often not the best way and more elaborate techniques like *differential scaling*, in which different components of the document are scaled differently, are preferable for many kinds of document.
- *Style Adaption* As with other document types, style sheets, such as CSS or XSLT sheets, can be used with SVG to determine properties, such as font size and stroke width of text. As a consequence, geometric properties of the document components may need to be adapted to the concrete values of these properties. For example, if the font in a diagram changes, sizes and positions of text boxes need to be adjusted, as text-boxes must always be large enough to accommodate the corresponding text, but should not be much larger.
- *Localized Language Versions* Another common case of adaptivity is required by language-independent docu-

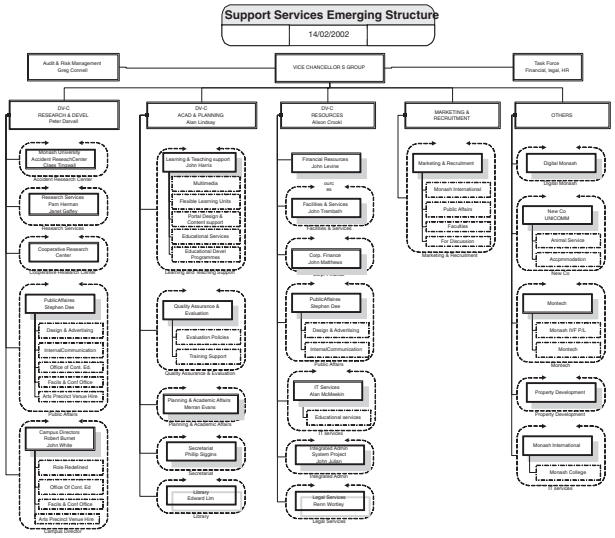

Figure 1: Full organization chart

Figure 2: Hierarchical viewing of organization chart on PDA

ments. As texts in a diagram change to a different language, their length obviously changes too, so that we are faced with much the same problems that arise from style adaption.

- *User Requirements* Not all users were created equal—even when using the same display device some users may have special requirements. This is most obvious in the case of a vision impaired user, who might require a larger font size (and maybe stronger stroke weights) to be able to read a diagram. Again, the problems that arise are the same as in style adaption and, in fact, this is just style adaption for a special reason.

- *Varying Interest in Detail* Even in a known scenario with fixed user requirements and device capabilities, different viewers may require different versions of the same document, since they are interested in varying levels of detail. Consider again the example of the organizational chart given in Figure 1. While one user may want to see only the top-level of the hierarchy, other users may want to see the full detail in all parts of the organization. Yet another user may be interested in the full detail of a particular department in the context of the overall organization, but not the full detail of the remaining departments (see Figure 3). This is another example of why semantic zooming is needed.

Here we propose that SVG be extended to include *one-way constraints*. In essence, one-way constraints allow attributes of document elements to be specified by an expression which is evaluated at runtime rather than requiring the document author to give an explicit value at the time of authoring (as is currently the case in SVG 1.0). As we show in Section 2, this comparatively minor extension to SVG renders it straightforward to write SVG documents which allow client-side adaptation of the form detailed above. Consider the simple example of a diagram that shows two boxes connected by an arrow. If we want the boxes to be evenly distributed in the width of the viewport, their concrete x-positions as well as the endpoints of the arrow have to be computed at runtime. A simple fragment of extended SVG code that uses one-way constraints to compute the arrow position (assuming that the boxes' URIs are `#b1` and `#b2`) is:

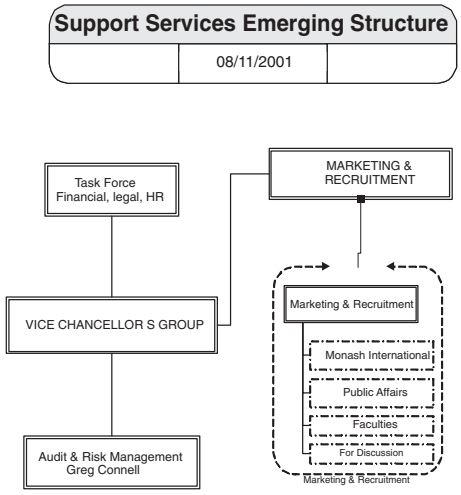

Figure 3: Partially expanded organizational chart

```
<line class="arrow"
  x1="url(#b1)_x + url(#b1)_width/2"
  y1="0.75in"
  x2="url(#b2)_x - url(#b2)_width/2"
  y2="0.75in" />
```

The key point to note is that formulas instead of concrete values are given for the graphical attributes of the objects.

Apart from improving adaptivity, adding one-way constraints to SVG has other benefits including better support for animation and some support for direct manipulation of document elements. This will be discussed in detail later in the paper. Importantly, client-side one-way constraint solving can be implemented at little computational cost and is fully compatible with the current SVG standard.

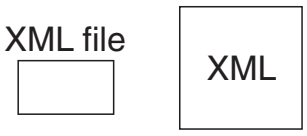

Figure 4: Label position determined by one-way constraints

Proposals to extend SVG with constraints have previously been made by the authors [1] as well as by other researchers [14]. The main difference to this earlier work, and our main technical contribution, is that we propose the use of one-way constraints while our previous work focussed on so-called *linear* constraints. The main advantages of using one-way constraints instead of linear constraints are threefold:

- *Expressiveness* Linear constraints are not expressive enough. By definition they only allow us to express relations between object properties that can be written as linear functions. This is insufficient for the specification of many common types of dependency between graphic elements, in particular relationships involving text, such as a text box which should be just big enough to enclose some text, non-linear relationships, such as a point is on the perimeter of a circle, or relations that are based on on an *if-then-else* selection. Consider the case of a diagram with fixed box sizes where we want labels to appear centered inside their corresponding box if they fit and outside otherwise (see Figure 4). This type of adaptiveness cannot be expressed with linear constraints, but one-way constraints are well-suited for it. Likewise, linear constraints cannot specify dependencies of values from discrete domains, such as colors or font sizes. All of these relationships can be captured by one-way constraints.

- *Efficiency* It is considerably simpler and more efficient to solve one-way constraints than to solve linear constraints. Although not an issue for laptop and desktop computing, this is a consideration for viewing devices, such as PDAs or mobile phones, with lower computational power. Currently such devices do not provide the computational power required to solve the relatively large sets of constraints that can arise in constraint-based SVG documents if linear constraints are used, but they do have the power to solve them if one-way constraints are used (See, for example, the Cabri system [13] which is implemented on pocket calculators).

- *Maturity* One-way constraints and constraint-solving is a very well-understood technology and one-way constraints are standard in many commercial products including widget layout in GUIs, spreadsheets, and graphic editors.

As we discuss in Section 2.4 the only disadvantage of one-way constraints is that they provide only limited support for direct manipulation. It is possible to have the best of both worlds by using multi-way constraints including linear constraints in the authoring tool but then compile these into one-way constraints in the presentation document.

There is a long history of using constraints in interfaces and interactive systems, beginning with Ivan Sutherland's pioneering Sketchpad system [18]. More specific related work includes the use of linear constraints to allow more flexible adaptation of text documents as suggested in [2] which details a constraint extension to Cascading Style Sheets and an earlier paper [3]. Another project called Madeus has used the Cassowary solver to handle a wider range of spatial and temporal constraints in multimedia documents [10]. Diehl and Keller describe constraint extensions to the Virtual Reality Markup Language (VRML) based on one-way constraints [4]. However, they do not consider adaptation nor do they consider alternate layouts.

2. ONE-WAY CONSTRAINTS FOR SVG

It is clear from the preceding examples that we would like the SVG browser to perform client-side adaptation of the layout of an SVG document. Importantly, we do not want the SVG browser to dramatically or unexpectedly change the layout specified by the document author, rather we want it to adjust or fine tune this layout using mechanisms understood by the author. Of course, we want such adaptation to occur dynamically if the viewer modifies the viewing environment by, say, resizing the browser window.

For (X)HTML documents such adaption is achieved by delaying the computation of a document element's location

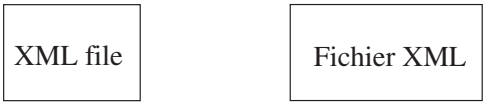

Figure 5: Language adaptivity

until display time. The author (and viewer) through style sheets, style attributes and layout widgets such as tables, implicitly specify a way to compute concrete positions, sizes, etc. at display time. It is the responsibility of the (X)HTML browser to determine the exact location of individual words. Importantly, however, the author understands the rules that will be used to determine the layout and there are few unexpected surprises.

More exactly, adaptation of (X)HTML is achieved by using the style elements and predefined layout rules for each kind of document element to implicitly define for each constraint attribute x associated with some document element a function f_x which details how to compute it from environment values such as browser window width and other document element attributes $x_1, ..., x_n$. These functions are set up in such a way that the values for $x_1, ..., x_n$ will be known at the time the value for x is needed. At display time the browser takes the environment value and uses the functions in the appropriate order to compute the attributes of all document objects.

Here we argue that flexible client-side adaptation of SVG documents can be achieved in almost exactly the same way: by allowing the author to specify constraints or relations between object attributes instead of absolute values for the attributes. These constraints implicitly define the functions needed by the browser to compute the concrete values of geometric attributes such as position and size from the values of other attributes and environment values.

Functions of this form have been widely studied in the constraints community under the name of *one-way* or *function* constraints. They are the simplest type of constraints and have been employed in user interface specification and graphic editors and also for incremental attribute computation in attribute grammars and in spreadsheets.

2.1 Style and Device Capabilities

Let us now have a close look at how one-way constraints in SVG can provide the different types of adaptivity that were identified in the introduction. We do this by means of several simple examples.

Our first example illustrates how one-way constraints allow us to adapt document layout to the size of text in the diagram. This addresses issues arising from style adaptation, localized language versions and user-requirements.

The principle is illustrated by the following simple fragment of code which dynamically adapts the size of the box to accommodate the text it contains (see Figure 5).

```
 <?xml version="1.0"?>
<!DOCTYPE svg SYSTEM "svg.dtd">
<svg width="6.0in" height="1.5in">
  <desc>XSLT translation of XML into SVG</desc>

  <switch>
          <text name="t1" systemlanguage="en"
          x="1.0in" y="0.75in">XML file</text>
          <text name="t1" systemlanguage="fr"
          x="1.0in" y="0.75in">Fichier XML</text>
  </switch>
  <var  id="boxwidth"
        val="url(
        #(/descendant::text/[@name="t1"]))_width"/>
  <var  id="boxheight"
        val="url(
        #(/descendant::text/[@name="t1"]))_height"/>
  <rect id="b1" class="box"
        x="1.0in - url(#boxwidth)/2"
        y="0.75in - url(#boxheight)/2"
        width="url(#boxwidth)"
        height="url(#boxheight)"/>
</svg>
```

In the example we use a `switch` statement to choose the text to be displayed. At display time the two variables `boxwidth` and `boxheight` are dynamically set to the text width and height respectively. Note that this will take into account the user style settings for the text. Then the surrounding box has its attributes computed so as to ensure it has the appropriate height and width and is centered around the text.

Extending SVG with such one-way constraint formulas obviously leads to a number of language design issues, in particular regarding the syntax of constraint terms and the way they reference attributes. Our examples simply extend SVG 1.0's syntax by allowing any attribute to be assigned an appropriately typed expression instead of a specific value. SVG elements are referred to by means of their URI ID and an underscore is used to access their attributes. We also allow XPATH references. The advantage of this notation is that it is reasonably concise and comprehensible. However, the concrete SVG syntax chosen here is not relevant to the discussion of the technical issues and it is not intended to be a complete suggestion for extended SVG. Indeed, one could consider extending XML with one-way constraints rather than just SVG, since the need for client-side adaptation is not only a problem for SVG but also other web document languages defined by means of XML.[1]

Note how this code not only makes use of attributes specified by expressions, but also uses a `<switch>` statement based on a read-only environment variable `systemlanguage` to decide which version to display. In general we would also like to allow tests consisting of one-way constraint formula in `<switch>` statements. This allows the designer to provide alternate layouts for the document. Consider the two versions of a diagram shown in Figure 6. The layout in a tall and narrow viewport should be entirely different from that for a wide viewport. Such alternate layouts can be encoded in one-way constraints using an if-then-else as a test on each attribute to determine which layout should be used.

Our next example is more complex. Like the first example the rectangles are sized to fit the text they contain and in this example all are set to the same size. More interestingly, we also perform *differential scaling*. The idea is that if the diagram on the top of Figure 7 is displayed in some smaller viewport, we would like it to be shown in the form given at the bottom. We are not performing just a linear scaling, instead the whitespace is compressed, while

[1]Thanks to the (anonymous) reviewer of this paper who suggested this.

```
<?xml version="1.0"?>
<!DOCTYPE svg SYSTEM "svg.dtd">
<svg id="canvas" width="6.0in" height="1.5in"
     viewBox="0in 0in url(#scale)*3.0in+3*url(#boxwidth) url(#boxheight)+1.0in">
  <desc>XSLT tanslation of XML into SVG</desc>

  <!-- Definition of variables -->
  <var  id="boxwidth"
        val="max(1in, url(#t1)_width, url(#t2)_width,url(#t3)_width)"/>
  <var  id="boxheight"
        val="max(0.5in, url(#t1)_height, url(#t2)_height,url(#t3)_height)"/>

  <var  id="reqpadding" val="url(#canvas)_actualWidth -3*url(#boxwidth)"/>
  <var  id="scale" val="if reqpadding > 3.0in then 1.0
                        else if reqpadding < 1.5in then 0.5
                        else reqpadding / 3.0in "/>

  <!-- Definition of text elements -->
  <text id="t1" class="boxlabel"
        x="url(#scale)*0.5in + url(#boxwidth)/2" y="0.75in">XMLfile</text>
  <text id="t2" class="boxlabel"
        x="url(#scale)*1.5in + 3*url(#boxwidth)/2" y="0.75in">XSLT</text>
  <text id="t3" class="boxlabel"
        x="url(#scale)*2.5in + 5*url(#boxwidth)/2" y="0.75in">SVG file</text>

  <!-- Definition of boxes -->
  <rect id="b1" class="box"
        x="url(#t1)_x - url(#boxwidth)/2"   y="0.75in -url(#boxheight)/2"
        width="url(#boxwidth)"         height="url(#boxheight)" />
  <rect id="b2" class="box"
        x="url(#t1)_x - url(#boxwidth)/2"   y="0.75in - url(#boxheight)/2"
        width="url(#boxwidth)"         height="url(#boxheight)" />
  <rect id="b3" class="box"
        x="url(#t1)_x - url(#boxwidth)/2"   y="0.75in -url(#boxheight)/2"
        width="url(#boxwidth)"         height="url(#boxheight)" />

  <!-- Definition of arrow -->
  <line class="arrow"
        x1="url(#b1)_x + url(#b1)_width/2"   y1="0.75in"
        x2="url(#b2)_x - url(#b2)_width/2"   y2="0.75in" />
  <line class="arrow"
        x1="url(#b2)_x + url(#b2)_width/2"   y1="0.75in"
        x2="url(#b3)_x - url(#b3)_width/2"   y2="0.75in" />
</svg>
```

Figure 8: Extended SVG Code for Differential Scaling

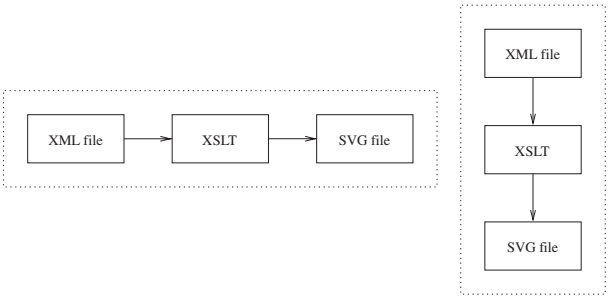

Figure 6: Alternate layouts

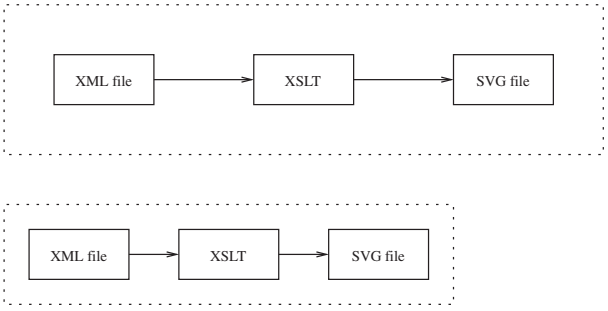

Figure 7: Differential scaling

the box-sizes are maintained. Our code does this by computing a scale factor `scale` for the arrow length and the amount of surrounding white space in the horizontal direction. This is always between 1/2 and 1. We assume that the `svg` element has additional attributes `actualWidth` and `actualHeight` which are the width and height of the viewport that the svg element will be displayed in (computed after negotiation with the parent and taking into account the *desired* width and height of the `svg` element given by its `width` and `height` attributes). In this case we compute the `viewBox` after differential scaling so that if required ad-

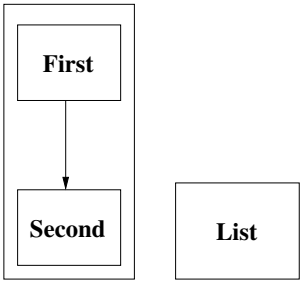

Figure 10: Interactive zooming

ditional linear scaling can be applied. Extended SVG code that uses constraints to implement this example is given in Figure 8.

2.2 Interactive Semantic Zooming

As we have discussed in the introduction, it is often convenient to be able to interactively change the level of detail in a diagram when viewing it. Consider the example of the organizational chart given in Figures 1 and 3. Extending SVG with one-way constraints provides this capability. We can implement alternative levels of detail between which the user can switch, effectively zooming in and out, while the constraints are used to adapt the remaining structure and layout of the diagram by maintaining specified invariants, such as the topology of the graph and the fact that some nodes may be above others (because they are higher up in the hierarchy). For example, we specify the end coordinates of particular edges to be identical to the positions of particular nodes, and providing only default values for the concrete geometric properties.

As an example consider the expansion of a node in the diagram shown in Figure 10. The code for this is given in Figure 9. We introduce a Boolean attribute `visible` that controls whether the node is to be expanded or not. This is appropriately initialized to take into account the size of the actual viewing area. We assume that user interaction can be used to change the value of the variable. Note how one-way constraint propagation will ensure that if the variable's value is changed the layout will also be changed.

2.3 Animation

In the case of document adaptation the idea is that the attributes of document elements are defined in terms of those of other document elements and attributes of the viewing environment. In a similar fashion one-way constraints provide a simple form of animation by making object properties a function of a read-only constraint variable *time* that is automatically incremented by the browser [5, 6]. Consider the case where we want two balls to bounce up and down simultaneously and the start of the motion is determined by a mouse-click on the object. The code for this is given below.

```
<?xml version="1.0"?>
<!DOCTYPE svg SYSTEM "svg.dtd">
<svg width="5.0in" height="5.0in"/>
  <desc>Two bouncing circles</desc>
  <circle id="c1"
          cx="1in" cy="1in"
          r="0.5in" style="fill:red">
    <animate attributeName="cy"
             dur="2sec" begin="user.click"
             val="1in + time*time" />
    <animate attributeName="cy" dur="2sec"
             val="1in - time*time"
             additive="cumulative"/>
    <animate attributeName="cy" dur="infinite"
             val="1in" additive="cumulative"/>
  </circle>
  <circle id="c2" cx="4in" cy="1in" r="0.5in"
          style="fill:red">
    <animate attributeName="cy" dur="2sec"
             begin="user.click"
             val="1in + time*time" />
    <animate attributeName="cy" dur="2sec"
             val="1in - time*time"
             additive="cumulative"/>
    <animate attributeName="cy" dur="infinite"
             val="1in" additive="cumulative"/>
  </circle>

  <line x1="url(#c1)_cx"   y1="url(#c1)_cy"
  x2="url(#c2)_cx"   y2="url(#c2)_cy"/>
</svg>
```

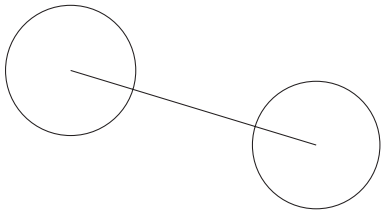

Figure 11: Bouncing balls

2.4 Direct Manipulation

The web is an inherently interactive medium. Given that one-way constraints are good at handling user initiated animation and user directed semantic zooming, it is natural to ask if they can also provide more powerful sorts of interaction. For example, a user presented with a graph that shows the structure of a web-site may want to re-arrange the nodes representing the pages in order to match the visualization with their conceptual view of the site. Such facilities are particularly useful for educational and instructional material. We might, for example, provide diagrams representing mechanical or physical devices that allow the user to directly manipulate key components in the device and see how moving them will affect other components in the device.

One-way constraints can support such interaction, but only in a limited way. They can, e.g., support direct interaction in which the viewer can move nodes in a graph and the edges follow the nodes. Basically they support the kind of direct manipulation provided in graphic authoring tools such as Visio.

However, as an indication of their limitations consider the abacus shown in Figure 12. One would like to allow the user to select beads and move them along the rod, with other beads being appropriately "pushed" by the bead being manipulated. It is not possible to provide such interaction using one-way constraints. The problem is that there is no one fixed flow of information between attributes in the figure. A bead's position depends on the beads below it if one of these is being directly manipulated, on those above it if one of those is being manipulated. This leads to a cycle in the one-way constraints which is not allowed.

```
<?xml version="1.0"?>
<!DOCTYPE svg SYSTEM "svg.dtd">
<svg id="canvas"
     width="6.0in" height="1.5in"
     viewBox="url(#canvas)_actualX, url(#canvas)_actualY,
              url(#canvas)_actualHeight,url(#canvas)_actualWidth">

 <desc>Sample file </desc>
 <g style="fill:none;stroke:black;stroke-width:0.01in;text-anchor:middle;
           font-style:normal;font-size:12pt;marker-end:url(#arrow-end)">

 <!-- Display the first box-->
 <switch att="visible">
   <rect id="box1" visible="url(#canvas)_actualHeight >= 1in" class="box"
         x="0.5in" y="0.5in" width="1in" height="1.5in" name="box1"/>
   <rect id="box1bis" visible="url(#canvas)_ActualHeight < 1in" class="box"
         x="0.5in" y="1.5in" width="1in" height="0.5in" name="box1"/>
 </switch>
 <!-- to simplify the reading of the sample, we call
      now box1 : /descent::rect[@name="box1"] -->

 <!-- definition of variables -->
 <var  id="x1" val=" url(#box1)_x"/>
 <var  id="y1" val="url(#box1)_y"/>

 <switch att="test">
    <g test="url(#box1)_visible=true">

      <!-- Display all the element in the box-->
      <text class="boxlabel" name="text1" x="url(#x1)+0.2in"
            y="url(#y1)+ url(#box1_1)_Height +0.1 in"> First</text>
      <rect id="box1_1" class="box" x="url(#x1)+0.1in"
            y="url(#text1)_y - url(#text1)_Height+0.1in"/>
      <text class="boxlabel" name="text2" x="url(#box1_1)_x"
            y="url(#box1_1)_y+2*url(#box1_1)_Height+0.2in">Second</text>
      <rect id="box1_2" class="box" x="url(#box1)_x+0.1in"
            y="url(#text2)_y - url(#text2)_Height+0.1in"/>

      <line id="line_box1_1box1_2" class="arrow"
         x1="url(#box1_1)_x+url(#box1_1)_width/2 in" y1="url(#box1_1)_y in"
         x2="url(line_box1_box2)_x1 in" y2="url(#box1_2)_y+url(#box1_2)_height"/>
     </g>

    <g test="false">
      <!-- Display only the label of the box1 -->
      <text class="boxlabel" x="url(#x1)+0.1" y="url(#y1)+url(#box1)_height-0.2in">List</text>
    </g>
   </switch>
  </g>
</svg>
```

Figure 9: Extended SVG Code for Semantic Zooming

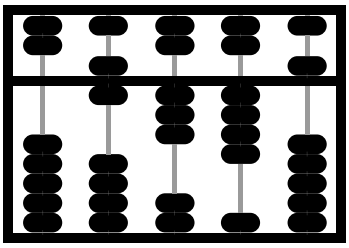

Figure 12: Interactive abacus

The key problem with one-way constraints is that they have a fixed direction of information flow: the value of the left-hand side variable depends on the value of the variables in the right-hand side but not vice versa. For this reason more powerful constraints and constraint solving techniques are required to support interaction in which the information flow cannot be determined when authoring the document. Constraints which allow more flexible information flow are said to be *multi-way constraints*. However, it is important to note that, leaving aside direct manipulation, we can solve the problems we have discussed above with one way constraints alone. It is also worth noting that, as we discuss in Section 4, multi-way constraints can in principle be compiled into one-way constraints for a specific mode of interaction.

2.5 Alternative Approaches

One might argue that one-way constraints do not need to be included in SVG at all. For example, it might be argued that providing alternative versions of the same document on the server side, which are used under different viewing conditions, is a sufficient solution. However, one can only provide a limited number of pre-designed versions, so that alternative versions cannot account for the full spectrum of available devices or for an individual user's requirements. This is particularly apparent in the presence of style sheets. Only flexible, adaptive documents that can automatically adapt to different viewing conditions provide the capability to handle these conditions gracefully.

However, the adaptive approach should not be used in isolation either: adaptive documents alone cannot cover all possible cases. Even if a document provides smart ways of adaptive zooming, etc., it is unlikely that such adaptions can go beyond a certain level of variation. It would be very difficult to design a document that could automatically adapt for the whole range from mobile phone displays to wide-screen television in a reasonable way, because here the changes are not only quantitative but qualitative. Therefore, we advocate a combination of alternative versions with one-way constraint based adaptation.

We should also note that some effects, such as the semantic zooming in our organizational chart example, could be achieved by using specialized viewer applets, such as a tree viewer, that allow the user to dynamically expand and collapse the diagram. However, this does not solve the general problem either, as the same situation may arise with very different kinds of diagrams and we can only provide such viewers for a limited number of diagram types.

The only alternative approach that provides the same degree of flexibility to that we can achieve with constraint-based SVG is to use script-based manipulation of the SVG document. Appropriate authoring tools could even hide the scripting from the author. However, we believe this is the wrong approach and that it is better to include one-way constraints in SVG itself. The reasons are:

- *Single language* Scripts force the document to be expressed in two languages, only one of which is XML compliant. This means that the documents are hard to understand and reason about by either humans or machines. In particular, scripts are difficult to edit and interchange between authoring tools is difficult.

- *Better implementation* Another good reason for providing one-way constraints as part of SVG is that it allows the SVG browser to provide a fast, more sophisticated implementation than is possible in a scripting language. In particular, it allows initial static solving of the constraints when the document is first rendered and incremental resolving of one-way constraints during animation, interaction and resizing of the browser window. It also allows for lazy evaluation of constraints and hence incremental downloading of documents.

- *Better support for multiple documents* A related problem is that unless one actually implements a one-way constraint solver in the scripting language itself, scripts hardwire the order in which attribute values are evaluated. This means that essentially the document author or authoring tool must determine how to solve the constraint graph at the time of authoring, rather than letting the solver do it at the time of display. This limits dependencies between documents. In practice, it means that interdependent documents must really be authored and downloaded as a single document.

We note that many of the above reasons are analogous to those used to support animation in SVG 1.0 and SMIL rather than requiring the author to write animation scripts.

3. IMPLEMENTATION

From the above examples it should be clear that extending SVG by adding one-way constraints allows the document author to explicitly state how their document should adapt to the viewing context. However, for this approach to be practical we need to be sure that the one-way constraints can be efficiently solved on the client-side with possibly limited computational resources.

Fortunately, there has been considerable research into how to solve one-way constraints efficiently (see [9], for example). Algorithms for solving one-way constraints are relatively simple and very fast, our experience shows that thousands of constraints can be solved in one or two milliseconds.

Indeed one-way constraints now form the basis for a variety of commercial products including spreadsheets and the customizable graphic editors Visio[2] and ConceptDraw.[3] In Visio and ConceptDraw one-way constraints provide the glue for ensuring that graphical objects behave naturally during

[2]http://www.microsoft.com/office/visio/
[3]http://www.conceptdraw.com

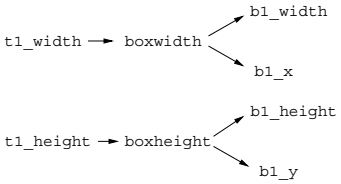

Figure 13: Example constraint graph

interaction by, for instance, ensuring a text box remains large enough to contain its component text or that lines or arrows connecting two objects follow the objects when they are moved. The reason one-way constraints are used is they provide a unifying implementation technology, they are fast and they allow users of Visio and ConceptDraw to readily construct customized editors for their own particular diagrammatic notation.

Algorithms for solving one-way constraints come in two forms: *static* for solving a system from scratch, and *incremental*, for resolving the system when some of the values change. In both cases the algorithms are couched in terms of the *constraint graph*. This is a directed graph with a node for each variable x which has an associated function $f_x(y_1, ..., y_n)$ for computing the value of x from the variables $y_1, ..., y_n$ and for each y_i a directed edge from the node representing y_i to the node representing x. For example, the constraint graph for the SVG document given at the start of Section 2.1 is shown in Figure 13.

The constraint graph induces a partial ordering on the variables reflecting the dependencies, i.e. variable x depends on y if there is a path from y to x. Static algorithms for solving one-way constraints first compute a total ordering for the variables which is consistent with the dependency ordering in the constraint graph, and then compute the value of the variables in this order. This can be done in time linear in the number of variables.

It is easy to modify this algorithm to be incremental. If a variable x's value changes, then we first determine all variables whose value depends on x. Then we compute a total ordering for these dependent variables which is consistent with the dependency ordering in the constraint graph and recompute their value in this order. Again this can be done in time linear in the number of variables.

One issue is that we need to ensure that the collection of functions is well-defined in the sense that no attribute value depends recursively on itself, i.e. the constraint graph must be *acyclic*. We note that an authoring tool could easily check whether a document is well-defined.

The above algorithms are *eager* in the sense that all nodes in the graph have the values computed. In the case of variables in different branches in a `switch` or an `if-then-else` we may not wish to do this, instead first evaluating the test condition in the `switch` or an `if-then-else` and only then add the variables in the chosen graph. This lazy approach although slightly more difficult to implement will generally be faster and also has the considerable advantage that it works well with incremental downloading of SVG documents.

4. AUTHORING OF EXTENDED SVG

An important issue is how authoring tools can be extended to support authoring of SVG with one-way constraints.

Authoring of graphical and multi-media documents with constraints has previously been discussed in [11, 10, 16]. In our context, it is particularly instructive to consider the Visio and ConceptDraw approach to authoring of diagrams with one-way constraints. The main approach is to provide pre-defined objects and constraint templates for different classes of diagrammatic notations, e.g. house plans and UML diagrams. Pre-defined objects consist of simpler graphic shapes with built-in one-way constraints dictating their behavior such as text boxes or connectors. Pre-defined constraint templates allow the author to add additional constraints such as vertical alignment to existing objects. When all else fails, the author can use a spreadsheet-like interface for textually specifying how to compute an attribute of an object from the attributes of other objects. Finally, there is the ability to add new pre-defined objects and templates and so to create a customized graphics editor for a new diagrammatic notation.

All of these approaches make sense in the context of SVG, however, they do not directly support authoring of adaptation (which is not surprising given that this was not the aim of either Visio or ConceptDraw.) We need to devise additional authoring metaphors that support differential scaling and semantic zooming. Regardless, however, we believe it is important to keep the ability for the author to add arbitrary one-way constraints since layout of graphical objects is more diverse than that of text and there is no comprehensive set of layout idioms. For this reason it is important for the underlying SVG to allow the specification of a rich variety of one-way constraints.

One important observation is that we do not have to be restricted to one-way constraints in authoring SVG code just because the browser provides only one-way constraints. Multi-way constraints offer a more powerful technology for supporting user interaction and in particular arbitrary direct manipulation. Therefore it would make considerable sense to support multi-way constraints in the authoring tool. The original multi-way constraints then have to be compiled into one-way constraints for the final constraint-based SVG document. This is possible because constraints in SVG are only used in limited ways. Compilation of multi-way constraints into one-way constraints is the subject of recent research, see for instance [17, 8]. It is clear in principle that it can be done (in a sense one is simply partially evaluating the multi-way constraint solver for a particular type of direct manipulation), but algorithms are needed to do this efficiently.

There are a number of metaphors that can be used for differential scaling. One approach is to use constraint templates similar to the standard "distribution" tools provided in most graphic editors or layout widgets in user interfaces. By this we mean that the author would use a tool to select a set of objects to distribute or align them. Instead of apply the adjustment operation only once, this would apply a constraint that keeps objects aligned even when one of them is moved.

A natural extension of this approach is a *spring model*. This can also be viewed as an extension of the *hfill* model of LaTeX or of the padding model that is used in many popular GUI toolkits such as Java AWT, Swing and TCL/Tk. The idea is that the document author can annotate the diagram with horizontal and vertical springs that are attached to objects or to the surrounding viewport (which is shown as an explicit surrounding frame). Each spring has a non-negative

natural, minimum and maximum length and a non-negative compression and extension strength. When the document is resized the objects are appropriately scaled in order to minimize the overall energy in the springs. However, scaling will not allow a spring to extend beyond its maximum length or decrease beyond its minimum length. Of course the author can directly manipulate the viewport to see how the diagram appears in different conditions. As an example see the spring annotation for three boxes which stay vertically aligned and horizontally distributed in the middle of the viewport (Figure 14). Spring models have previously been used for the authoring of multi-media documents in [12]. We believe the spring model is reasonably intuitive, however, there are difficulties in combining it with arbitrary one-way constraints such as boxes being large enough to contain their text. Neither spring constraints nor distribution constraints are one-way, so we need to be able to compile them into one-way constraints as discussed previously. We are currently investigating this approach.

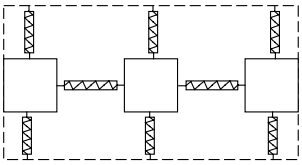

Figure 14: Visualising constraints with springs

Another approach to authoring differential scaling is to use *example-based inference* of the scaling constraints. The idea is for the author to create snapshots of how they wish the diagram to appear for various size view boxes and then for the system to infer the underlying constraints. Example based inference of constraints has been tried in both user-interface design and in generic graphic editors, for example [15]. Arguably, it has not proven overly successful in these areas. This is because in an unbounded context the system must see many examples before it has any hope of determining what the user is trying to achieve. In the context of specifying differential scaling, however, the problem is considerably simpler since the system can start with a predefined class of constraints with only a few parameters to be determined. This is possible if we make the object properties dependent on a few well-defined parameters only. For example, one might assume that every attribute in the x dimension, such as width or a x coordinate, is a linear function of the viewbox's width and the attribute's initial value, while those attributes in the y dimension are a linear function of the viewbox height and their initial value. Figure 15 shows the two examples that would be required to specify that two rectangles always stay top-aligned in the middle of the viewport and evenly distributed in width. As with the spring approach, there is a difficulty in combining this approach with (other) one-way constraints.

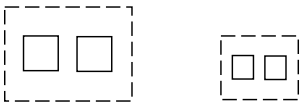

Figure 15: Inferring constraints from examples

Authoring metaphors for semantic zooming and alternative layout seem fairly straightforward: one needs to be able to indicate that an object has more than one layout and use a new canvas to author each of these layouts. The spring-based approach can be used to specify the applicability of the individual different layouts by giving the springs minimum and maximum lengths beyond which they cannot extend.

5. PROTOTYPE IMPLEMENTATION

As a proof-of-concept we have implemented a prototype authoring and browsing tool for a one-way constraint-based extension to SVG. This was reasonably straightforward since we could leverage from the Visio diagram editor discussed previously. Visio was ideal for our purpose because not only is it constraint-based, but it is also designed to be easily extensible.

One might argue that Visio itself is already a suitable authoring tool for constraint SVG. However the use of constraints in our prototype is quite different from the use envisaged by the designers of Visio. In Visio (and ConceptDraw, for that matter) constraints exist to support the actual editing process, e.g., by keeping two object aligned when either is moved. Importantly, once the editing is finished and the document is finalized, the author is not interested in the constraints any more: the final document provides an absolute layout of the graphics. In contrast, when producing constraint SVG the constraints themselves are part of the final document as well as the graphic elements and are just as important since they are ultimately what determines the concrete layout of the document in given circumstances.

Our prototype tool can read and write a restricted version of SVG together with one-way constraints. In accord with the philosophy of Visio it is not a generic graphics editor but rather an editor for specialized class of diagrams, in this case adaptive organizational charts of the form shown in Figure 1. By providing a customized editor specialized for particular classes of diagrammatic notations we can remove the need for the user to explicitly add most constraints by providing pre-defined object *templates*. The major component of our prototype are Viso plug-ins for reading and writing SVG and predefined templates for adaptive organizational charts. Using these pre-defined object types, the author can easily create organizational charts without having to take care of the implicit constraints. Additional explicit constraints, such as alignment, can be added with standard tools.

The second purpose of the prototype implementation was to show that documents based on one-way constraints can be efficiently processed. In order to this we created 7 different organizational charts with 80 to 1440 objects. For each document we created two versions, the first using one-way constraints to compute the values of attributes, the second containing no one-way constraints only concrete values for the attributes. For each of the two versions of these documents we measured the document processing time, i.e. the time for loading and displaying the document including constraint solving, with Visio 2001 running on a 1.4GHz PC clone running Windows 2000.

Table 1 displays the performance on the different sized documents (timings are the average over 10 runs). For smaller documents, the overhead due to the initialization of the solver results in a larger difference in performance between processing with and without constraints. However, as the document size grows, the initialization time has less

Number Objects	Number Constraints	Without Constraints (ms)	With Constraints (ms)	Difference (%)
80	640	320	400	25.0
160	1280	380	441	16.0
240	1920	405	461	12.2
320	2560	475	530	11.3
400	3200	490	551	12.0
640	5120	611	681	11.4
1280	10240	1175	1308	11.9

Table 1: Timings for Constraint Processing.

influence on the performance difference. Instead, the performance difference is primarily determined by the time taken to process the constraints. For larger documents, the performance difference stabilizes at approximately a 12%.

6. CONCLUSION

We believe the case for adding one-way constraints to SVG is strong. One-way constraints directly address limitations in SVG 1.0, by giving the SVG document author the ability to specify how to perform flexible client side adaptation of the document in response to viewer requirements and the viewing environment. In particular this allows differential scaling and semantic zooming. More precisely, our proposed extension is that:

- We allow attributes of SVG document elements to be specified by an expression which is evaluated at display time rather than requiring the document author to give an explicit value at the time of authoring.
- We allow these expressions and also the `switch` statement to include tests for applicability which are checked at display time.
- There is a slightly more complex negotiation between the SVG document and its surrounding context in which the document provides a desired display region size and the browser sets an actual display region size.

We suggest that this is integrated with an ability to specify alternate documents for radically different kinds of viewing devices.

The main advantage of one-way constraints over the previously suggested approach of using linear constraints is that one-way constraints are more expressive allowing virtually any desired adaptive behavior to be specified, they can be efficiently evaluated even on viewing devices with low computational power such as PDAs, and that they are well-established technology, easily implemented and already they are used in number of commercial graphics tools including Visio and ConceptDraw as well as widget layout in GUI toolkits and spreadsheets. Thus we believe it would be relatively easy to incorporate them into current SVG browsers.

As evidence for this, with relatively little work we have added a module to Visio which is both a browser and an authoring tool for (a restricted subset) of SVG with one-way constraints.

7. ACKNOWLEDGEMENTS

We would like to acknowledge the input of Greg Badros, Alan Borning, Jojada J. Tirtowidjojo and Peter Stuckey, with whom we have collaborated on earlier versions of constraint SVG and other related constraint-based graphics projects. Their input has strongly influenced our development of Constraint SVG. We would also like to thank Dean Jackson for comments on earlier drafts.

8. REFERENCES

[1] G. Badros, J. J. Tirtowidjojo, K. Marriott, B. Meyer, W. Portnoy, and A. Borning. A constraint extension to scalable vector graphics. In *Proc. 10th World Wide Web Conference*, Hong Kong, May 2001.

[2] G. J. Badros, A. Borning, K. Marriott, and P. Stuckey. Constraint cascading style sheets for the web. In *Proceedings of the 1999 ACM Conference on User Interface Software and Technology*, November 1999.

[3] A. Borning, R. Lin, and K. Marriott. Constraints for the web. In *Proceedings of ACM Multimedia 1997*, Nov. 1997.

[4] S. Diehl and J. Keller. VRML with constraints. In *Proceedings of the Web3D-VRML 2000 fifth symposium on virtual reality modeling language*, Monterey, California, February 2000.

[5] R. Duisberg. Animus: A constraint based animation system. In *CHI'86 Proceedings*, pages 131–136, Boston, Apr. 1986.

[6] R. Duisberg. Animation using temporal constraints: An overview of the Animus system. *Human-Computer Interaction*, 3(3):275–308, 1987.

[7] J. Ferraiolo. Scalable vector graphics (SVG) 1.0 specification. W3C Working Draft, December 1999.

[8] W. Harvey, P. Stuckey, and A. Borning. Compiling constraint solving using projection. In *Proceedings of the 1997 Conference on Principles and Practice of Constraint Programming (CP97)*, pages 491–505, Oct. 1997.

[9] S. Hudson. Incremental attribute evaluation: A flexible algorithm for lazy update. *ACM Transactions on Programming Languages and Systems*, 13(3):315–341, July 1991.

[10] M. Jourdan, N. Layaida, C. Roisin, L. Sabry-Ismail, and L. Tardif. Madeus, an authoring environment for interactive multimedia documents. In *ACM Multimedia '98*, Bristol UK., 1998.

[11] R. K. Harada, E. Tanaka and Y. Hara. Anecdote: A multimedia storyboarding system with seemless authoring support. In *ACM Multimedia 1996*, pages 341–351, November 1996.

[12] M. Kim. Creative multimedia for children: Isis story builder. In *Companion Proceedings of the Conference*

on Human Factors and Computing, (CHI'95), Denver/USA, 1995.

[13] J. Laborde and C.Laborde. Des connaissances abstraites aux ralits artificielles, le concept de micromonde cabri. In *Environnements interactifs d'apprentissage avec ordinateur*, volume 2, pages 29–41, Paris, 1995. Eyrolles.

[14] P. Mansfiled. A theory of constrained linear transformations for computer graphics, January 1999. Draft report.

[15] B. Myers. Peridot: Creating user interface by demonstration. In *Watch What I DO: Programming by Demonstration, Allen Cypher Editor*, pages 125–153, Cambridge, 1993. MIT Press.

[16] D. Saade, L. Soares, F. Costa, and G. Souza Filho. Graphical structured-editing of multimedia documents with temporal and spatial constraint. In *4th Conference on Multimedia Modelling*, pages 279–295, 1997.

[17] M. Sannella, J. Maloney, B. Freeman-Benson, and A. Borning. Multi-way versus one-way constraints in user interfaces: Experience with the DeltaBlue algorithm. *Software—Practice and Experience*, 23(5):529–566, May 1993.

[18] I. Sutherland. Sketchpad: A man-machine graphical communication system. In *Proceedings of the Spring Joint Computer Conference*, pages 329–346. IFIPS, 1963.