Constraint SVG

Cameron McCormack
Monash University
Clayton, Vic. 3800, Australia

Kim Marriott
Monash University
Clayton, Vic. 3800, Australia

Bernd Meyer
Monash University
Clayton, Vic. 3800, Australia

firstname.lastname
@infotech.monash.edu.au

ABSTRACT

We believe it is important for web graphic standards such as SVG to support user interaction and diagrams that can adapt their layout and appearance to their viewing context so as to take into account viewing device charateristics and the viewer's requirements. In [1] we suggested that adding expression-based attributes to SVG and using one-way constraints to evaluate these dynamically would considerably improve SVG's support for adaptive layout and user interaction. We describe a minimal backward compatible extension to SVG 1.1, called Constraint SVG (CSVG), that provides such expression-based attributes and its implementation on top of Batik. CSVG also provides another significant extension to SVG 1.1: it allows the author to define new custom elements using XSLT.

Categories and Subject Descriptors

H.4.m [**Information Systems**]: Miscellaneous; D.2 [**Software**]: Software Engineering; H.5.2 [**Information Systems**]: Information Interfaces and Presentation

General Terms

Constraint-based Graphics, Document Formats

Keywords

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

1. CONSTRAINT SVG

With the advent of a large array of different devices for web browsing, adaptive document layout is becoming an increasingly important issue. This not only applies to page layout, but also to the layout of individual diagrams. Therefore, as detailed in [1], an important requirement for web graphics is that a diagram can adapt its layout and appearance to its viewing context so as to take into account viewing device charateristics and the viewer's requirements. Another important aspect of advanced web-based graphics is that a diagram should be able to directly support interaction.

Unfortunately, the current vector-based graphics standard SVG 1.1 does not explicitly support these. Instead, the document author typically has to use scripting in combination with SVG. In [1] we suggested that adding expression-based attributes to SVG and using one-way constraints to evaluate these dynamically would considerably improve SVG's support for adaptive layout and user interaction and remove much of the need for scripting.

In this poster we describe a minimal backward compatible extension to SVG 1.1, called Constraint SVG (CSVG), that provides such expression-based attributes. CSVG also provides another significant extension to SVG 1.1 and our earlier proposals: it allows the author to define new custom elements using XSLT. This feature of CSVG was influenced by public documents from the SVG working group describing proposed features in SVG 1.2.

Expression-based attributes and custom elements combine synergistically: the dynamic evaluation of expression-based attributes allows us to specify user interaction and animation directly in terms of the custom elements. This is not supported in the current SVG 1.2 proposal. The attributes of the custom elements are linked to the attributes of the base SVG elements in the shadow tree via one-way constraints. Efficient constraint solving allows us to propagate attribute changes to the shadow tree elements without the need for expensive regeneration of the shadow tree.

To illustrate the capabilities of CSVG consider the simple box-and-arrow diagram in Figure 1. We use custom elements to model box and labelled arrows. We use expressions to compute the size of the boxes from the size of their enclosed text, allowing the browser to appropriately adjust the layout for different font sizes or languages, and expressions to ensure that the arrow runs from one box to the other and that the arrow label remains above the arrow's mid-point. To demonstrate animation we have added CSVG code to let the boxes bounce.

CSVG extends SVG to allow any animatable attribute to have an expression assigned to it rather than an actual value. The expression is specified using a syntax similar to that for animation. This syntax, like SMIL, provides backwards compatability by allowing the user to give an absolute value for a constraint attribute which can be used by SVG browsers that do not support constraints.

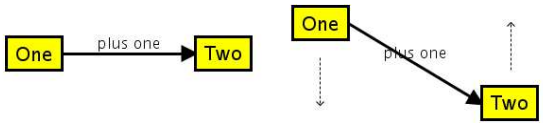

Figure 1: CSVG Example when (a) `$animate` = false, and (b) `$animate` = true

WWW2004, May 17–22, 2004, New York, New York, USA.
ACM 1-58113-912-8/04/0005.

```
<extensionDefs namespace="http://mcc.id.au/example/charts">
  <xsl:stylesheet id="chartXSL" xmlns:chart="http://mcc.id.au/example/charts" ... >
    <xsl:template match="chart:node">
        <c:property name="textWidth"
                    value="c:width(c:bbox(following-sibling::svg:text))"/>
        <c:property name="textHeight"
                    value="c:height(c:bbox(following-sibling::svg:text))"/>
        <rect x="{@x}" y="{@y}" width="{@width}" height="{@height}" onclick="{@onclick}">
          <c:constraint attributeName="x" value="c:instance()/@x"/>
          <c:constraint attributeName="y" value="c:instance()/@y"/>
          <c:constraint attributeName="width" value="c:instance()/@width"/>
          <c:constraint attributeName="height" value="c:instance()/@height"/>
        </rect>
        <text x="{@x + @width div 2}" y="{@y + @height div 2}"
              text-anchor="middle" font-size="20" pointer-events="none">
          <xsl:value-of select="."/>
          <c:constraint attributeName="x"
                        value="c:instance()/@x + c:instance()/@width div 2"/>
          <c:constraint attributeName="y"
                        value="c:instance()/@y + c:instance()/@height div 2
                               + $textHeight div 2"/>
        </text>
    </xsl:template>
    <xsl:template match="chart:arrow">
        <line x1="{@x1}" y1="{@y1}" x2="{@x2}" y2="{@y2}" stroke="black" stroke-width="3">
          <c:constraint attributeName="x1" value="c:instance()/@x1"/>
          <c:constraint attributeName="y1" value="c:instance()/@y1"/>
          <c:constraint attributeName="x2" value="c:instance()/@x2"/>
          <c:constraint attributeName="y2" value="c:instance()/@y2"/>
        </line>
        <text x="{(@x1 + @x2) div 2}" y="{(@y1 + @y2) div 2 - 5}"
              text-anchor="middle" font-size="16">
          <xsl:value-of select="."/>
          <c:constraint attributeName="x"
                        value="(c:instance()/@x1 + c:instance()/@x2) div 2"/>
          <c:constraint attributeName="y"
                        value="(c:instance()/@y1 + c:instance()/@y2) div 2 - 5"/>
        </text>
    </xsl:template>
  </xsl:stylesheet>
  <elementDef name="node"> <transformer xlink:href="#chartXSL" type="text/xsl"/></elementDef>
  <elementDef name="arrow"><transformer xlink:href="#chartXSL" type="text/xsl"/></elementDef>
</extensionDefs>
```

Figure 2: XSLT Template Definitions

In order to simplify constraints and to allow factorisation of common sub-expressions we allow the author to declare new variables whose value is given by an expression. For instance, in Figure 3 `$w` is set to the maximum length of the text inside the boxes so as to ensure that the boxes have the same size but are large enough to contain their text.

As custom elements can generate constraint definitions as well as standard SVG elements, the layout of the final SVG document is a two-phase process: First the shadow tree elements are generated and attributes in the shadow tree are bound to expressions instead of values where required. This allows shadow tree element placement to be specified in terms of other elements in the shadow tree. Thus for instance in our example above, a rectangle is specified to be large enough to contain its text. In the second phase the actual value of the attributes are automatically determined by the constraint solver and transparently re-computed when required, e.g. in the case of resizing or user interaction.

It is not possible to specify such layout if XSLT is simply generating standard SVG elements since this requires all attribute values to be explicitly computed by the XSLT processor. The XSLT processor, however, has no means to determine implicit quantities, such as the size of text or any transformed SVG elements. Thus using XSLT to generate SVG alone does not support adaptive layout. We need the semantic properties of the generated SVG elements. Allowing the shadow tree elements to contain expressions for attribute is a simple way of achieving this.

Animation can be understood as a special type of expression which gives an attribute's value in terms of the time since the start of the animation. The current animation expressions provided by SVG can be understood as convenient shorthand for more common types of animation. We allow the author to specify an animation using expressions that

```
<c:variable id="animate" name="animate" value="false()"/>
<c:variable name="w" value="c:max(c:property(id('n1'), 'textWidth'),
                                  c:property(id('n2'), 'textWidth'))"/>
<c:variable name="h" value="c:max(c:property(id('n1'), 'textHeight'),
                                  c:property(id('n2'), 'textHeight'))"/>

<chart:node id="n1" x="100" y="100" width="100" height="100" onclick="toggleAnimation()">
  One
  <c:constraint attributeName="width" value="$w + 20"/>
  <c:constraint attributeName="height" value="$h + 20"/>
  <c:constraint attributeName="x" value="140 - $w div 2"/>
  <c:constraint attributeName="y"
                value="c:if($animate, ((c:time() mod 10) div 10) * (300 - ($h + 20)),
                                      140 - $h div 2)"/>
</chart:node>
<chart:arrow x1="200" y1="150" x2="300" y2="150">
  plus one
  <c:constraint attributeName="x1" value="id('n1')/@x + id('n1')/@width"/>
  <c:constraint attributeName="y1" value="id('n1')/@y + id('n1')/@height div 2"/>
  <c:constraint attributeName="x2" value="id('n2')/@x"/>
  <c:constraint attributeName="y2" value="id('n2')/@y + id('n2')/@height div 2"/>
</chart:arrow>
<chart:node id="n2" x="300" y="100" width="100" height="100" onclick="toggleAnimation()">
  Two
  <c:constraint attributeName="width" value="$w + 20"/>
  <c:constraint attributeName="height" value="$h + 20"/>
  <c:constraint attributeName="x" value="340 - $w div 2"/>
  <c:constraint attributeName="y"
                value="c:if($animate, (1 - ((c:time() mod 10) div 10)) * (300 - ($h + 20)),
                                      140 - $h div 2)"/>
</chart:node>
```

Figure 3: CSVG Example using XSLT Templates

refer to `time()` which yields the time since the animation of the respective element started. As the example illustrates we allow custom element attributes to be animated as well.

With expressions and variables it is also quite simple to specify quite complex user interation, such as semantic zooming, i.e. zooming in which elements do not only change their size but their shape and appearance, such as expanding or collapsing sub-trees [1].

2. IMPLEMENTATION

Our implementation is based on Batik and utilises one-way constraint solving algorithms developed for a variety of applications including GUIs, spreadsheets and customisable graphic editors, such as Visio [2]. The one-way constraint solver is responsible for updating the value of expressions and for analyzing the dependencies between the attributes in order to determine an evaluation order that allows the updates to be computed efficiently. As the order of expression evaluation is independent of the order of SVG elements in the document, the author is free to specify what they want without worrying about how it should be computed.

Further information about CSVG and its implementation is available at `http://www.csse.monash.edu.au/~clm/csvg/` which also provides many realistic examples demonstrating the usefulness of extending SVG with dynamic attribute evaluation that fully supports custom elements. To create these examples using scripting instead of CSVG, significantly more implementation effort would have been required. The CSVG browser, which can be downloaded from the website, proves that CSVG can efficiently be rendered for real-time interaction and animation support.

3. REFERENCES

[1] K. Marriott, B. Meyer, and L. Tardif. Fast and efficient client-side adaptivity for SVG. In *Proc. 11th International World Wide Web Conference*, May 2002.

[2] B. Vander Zanden et al. Lessons learned about one-way, dataflow constraints in the Garnet and Amulet graphical toolkits. *ACM Transactions on Programming Languages and Systems*, 23(6):776–796, 2001.