A Constraint Extension to Scalable Vector Graphics

UW Technical Report #2000-08-04

Greg J. Badros *Will Portnoy* *Jeff Nichols* *Alan Borning*
{badros,will,jwnichls,borning}@cs.washington.edu
Dept. of Computer Science and Engineering
University of Washington, Box 352350
Seattle, WA 98195-2350, USA

ABSTRACT

Scalable Vector Graphics (SVG) is a language developed by the World Wide Web Consortium for describing two dimensional vector graphics for storage and distribution on the Web. Unlike raster image formats, SVG-based images scale nicely to arbitrary resolutions and sizes. We introduce a constraint extension to SVG called Constraint Scalable Vector Graphics (CSVG) that permits a description of an image that is more flexible. With CSVG, an image can contain objects whose positions and other properties are linearly related to other attributes via constraints. For example, a rectangle can be specified to remain above a circle, and a line can be constrained to connect their centers. The various constraints each have a specified strength, and we use constraint hierarchy theory to determine an appropriate solution. CSVG enables better layouts of diagrams for a wider variety of viewing conditions and provides support for declaratively specified animation. We embedded our Cassowary constraint solving toolkit in an existing SVG renderer to produce a prototype implementation of a CSVG system.

KEYWORDS: constraints, Cassowary toolkit, CSVG, SVG, Scalable Vector Graphics, illustration.

INTRODUCTION

Scalable Vector Graphics (SVG) [17] is a language developed by the World Wide Web Consortium (W3C) for describing two dimensional vector graphics. SVG is used for storage and distribution of images on the web, and is increasingly well-supported by both commercial and free software. In contrast with raster image formats such as GIF, JPEG, and PNG, which store a matrix of individual pixels that compose an image, a Scalable Vector Graphic image contains instructions for resolution independent rendering: the same SVG file will be shown in more detail when viewed at a higher resolution (e.g., on a 1200 dots per inch typesetting device rather than a 75 dpi screen). A sample SVG image appears in Figure 1.

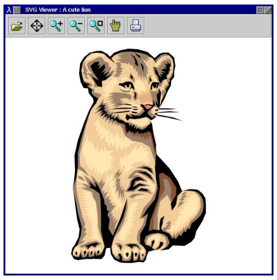

Figure 1: SVG image of a lion cub.

SVG graphics provide numerous immediate benefits besides resolution independence. SVG files are often smaller than an analogous raster image, thus web pages using them may take less time to download. Because SVG is based on XML [13], SVG files are easy to exchange, process, and analyze. SVG integrates well with Cascading Style Sheets (CSS) [12] specifications, thus enabling some separation of the content of the graphic from the visual appearance of that image. For example, the colors of a graphic can be specified in a style sheet that is independent of the SVG file itself. SVG also preserves image structure at a higher level—for example, a web browser can directly read the text included in an SVG figure. This ability, along with the separation of style from content, dramatically improves the accessibility of images for users with color-blindness or other visual impairments. Additionally, the Document Object Model (DOM) [2] and the SVG DOM [17, Appendix B] can be used to manipulate the shapes in an image dynamically to create animations and other effects.

SVG is Not Enough

Although the SVG format is a huge step forward for many kinds of images, we can do even better for diagrammatic illustrations. Contrast the illustration in Figure 2 with the lion cub in Figure 1. Figure 2 is a simpler image in which we provide a visualization of a class hierarchy. With SVG we have to specify the entire diagram fully and exactly by giving positions and sizes for all of the elements: precisely one class hierarchy diagram is described.

Full specification is important for a complex realistic image such as Figure 1, but is less important for many information visualization applications. Instead, in Fig-

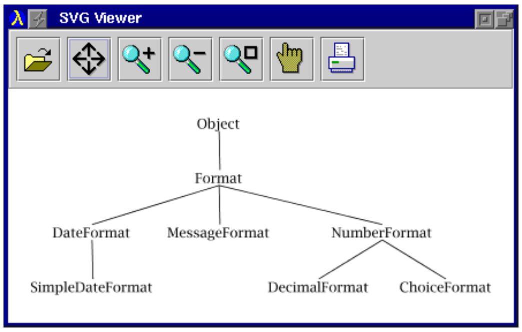

Figure 2: SVG image diagramming the object hierarchy surrounding the `Java.Text.Format` class. The SVG source for this image appears in Figure 3.

ure 2, there are certain properties of the layout that are important in conveying the desired information. For example, we want the parent class "Object" to appear above its subclasses, and want lines to connect classes to denote the inheritance relationship. If we were able to describe what is actually semantically important about a figure, we could have a single description that preserves flexibility for the renderer and would generate Figure 2 or other variations of that illustration.

Constraints are a useful approach for allowing users to state their intentions more directly. A constraint is a declarative specification of a relationship that we wish to hold true. For example, "`Format` appears above `DateFormat`" is a constraint. We can write the constraint mathematically as:

$$\mathrm{Format}.y_{bottom} + \mathrm{vert\_spacing} \leq \mathrm{DateFormat}.y_{top}$$

By stating declaratively how the two object attributes are to relate, we avoid having to give explicit values to either. Instead, we can defer that task to a constraint satisfaction algorithm that will assign values to variables. In this example, we can then use those value assignments to determine where to position the names of the various classes in the hierarchy.

Our Contributions

We describe a constraint extension to Scalable Vector Graphics, called Constraint Scalable Vector Graphics (CSVG). Our extension allows CSVG images to use arbitrary linear arithmetic constraints to control the layout of shapes, lines, paths, and font sizes. With constraints, diagrams can be under-specified, thus permitting the rendering engine greater flexibility when laying out the illustration.

Our main contributions are:

- a motivation for using constraints for certain kinds of SVG illustrations;
- a description of Constraint Scalable Vector Graphics as an extension of SVG, including a Document Type Definition (DTD) for CSVG; and
- a prototype implementation of a CSVG viewer based on the CSIRO SVG viewer [35]. The prototype makes use of the sophisticated constraint solving algorithm Cassowary [11].

BACKGROUND

The Scalable Vector Graphics (SVG) language [17] is based on the eXtensible Markup Language (XML) [13]. SVG also makes use of the Cascading Style Sheets (CSS) [12] standard for partially separating visual presentation information from the basic image description itself. In this section, we provide a brief overview of each of these standards, and then discuss the Cassowary Constraint Solving Toolkit, which provides the engine behind our constraint-based extensions.

XML: eXtensible Markup Language

XML is a standardized eXtensible Markup Language [13] that is a subset of SGML, the Standard Generalized Markup Language [27]. The World Wide Web Consortium (W3C) designed XML to be lightweight and simple, while retaining compatibility with SGML. Although HTML (HyperText Markup Language) is currently the standard web document language, the W3C is positioning XHTML, an XML-based language, to be its replacement. While HTML permits authors to use only a pre-determined fixed set of tags in marking up their document, XML allows easy specification of user-defined markup tags adapted to the document and data at hand [18, 19]. XML can thus be used as the basis for many languages describing arbitrary data, not just the single XHTML language.

An XML document consists simply of text marked up with tags enclosed in angle braces. A simple example appears in Figure 3.

The `<svg>` is an open tag for the `svg` element. The `</svg>` at the end of the example is the corresponding close tag. Text and other nested tags can appear between the open and close constructs. In the example, the `svg` contains 16 immediate children elements. Empty elements are allowed and can be abbreviated with a specialized form that combines the open and close tags: `<`*`tag-name`*`/>` (e.g., each of the `line` elements). Additionally, an XML open tag can associate attribute–value pairs with an element. For example, the first `text` element has the value `200` for its `x` attribute. Attributes of an element are unordered and multiple values for the same attribute name are disallowed. In contrast, child elements are ordered, and multiple child elements of the same type may be permitted (e.g., there are eight `text` children of the `svg` element).

For an XML document to be *well-formed*, the document must conform to the syntactic rules required of XML documents (e.g., tags must be balanced and properly nested, and attribute values must be of the proper form and enclosed in quotes).

```
<?xml version="1.0"?>
<!DOCTYPE svg SYSTEM "svg.dtd">
<svg width="4.5in" height="4in"
     viewBox="0 0 100 100"
     style="fill: none; font-size: 15;
            stroke-width: 1; stroke: black;
            text-anchor: middle">
  <desc>The object hierarchy surrounding
       the class "Java.text.Format"</desc>

  <text x="200" y="30">Object</text>
  <text x="200" y="90">Format</text>
  <text x="60" y="150">DateFormat</text>
  <text x="60" y="210">SimpleDateFormat</text>
  <text x="200" y="150">MessageFormat</text>
  <text x="380" y="150">NumberFormat</text>
  <text x="310" y="210">DecimalFormat</text>
  <text x="450" y="210">ChoiceFormat</text>
  <line x1="200" y1="32" x2="201" y2="75"/>
  <line x1="200" y1="92" x2="60" y2="135"/>
  <line x1="200" y1="92" x2="201" y2="135"/>
  <line x1="200" y1="92" x2="380" y2="135"/>
  <line x1="60" y1="152" x2="61" y2="195"/>
  <line x1="380" y1="152" x2="310" y2="195"/>
  <line x1="380" y1="152" x2="450" y2="195"/>
</svg>
```

Figure 3: SVG source of the class hierarchy illustration shown in Figure 2. SVG is based on XML.

A more stringent characterization of an XML document is *validity*. An XML document is valid if and only if it both is well-formed and adheres to its specified *document type definition*, or *DTD*. A document type definition is a formal description of the grammar of the specific language to be used by a class of XML documents. It defines all the permitted element names and describes the attributes that each kind of element may possess. It also restricts the structure of the nesting within a valid XML document. Figure 3 is valid with respect to the DTD that describes Scalable Vector Graphics, `svg.dtd` [17, Appendix A].

SVG: Scalable Vector Graphics

SVG is an XML-based language for describing vector graphics. It was designed by the W3C and is intended to be the standard format for all images on the Internet. Vector graphics provide resolution independence—the description of the image is based on higher-level graphical elements, rather than the pixels used to describe a raster image. SVG uses XML elements to represent basic shapes, including rectangles, ellipses, lines, and polygons. It also supports the more general notion of an arbitrary path that can represent an outline to be filled, stroked, or clipped to. SVG is very similar in spirit to the PostScript page-description language [1], but uses XML syntax instead of postfix notation.

An SVG element describes a shape to be rendered. For example:

```
<rect x="20" y="10" width="10" height="5"/>
```

describes a rectangle whose top-left is positioned at coordinate (20,10) with a width of 10 units, and a height of 5 units. Lengths and coordinates can specify units explicitly, but when they are omitted, the user space coordinate system is used [17, Ch.7]. Unfortunately, all basic shape objects use their top-left as an anchor point, making it unduly cumbersome to position, for example, the center of an object at a specific location.

An especially powerful SVG element is `path`. Its `d` (for "data") attribute contains a string that encodes a command-based description of an arbitrary outline. For example, the element:

```
<path d="M 20 10 L 30 10 L 30 15 L 20 15 Z"/>
```

describes a rectangle path equivalent to the preceding `rect` element: first **M**ove to (20, 10), then draw **L**ines to (30,10), (30,15), and (20,15), and finally close the path (**Z**). Uppercase command characters designate the use of absolute coordinates, while lowercase denotes relative coordinates. Other `path` sub-language commands include **C**urve-to, **S**mooth curve-to, **Q**uadratic Bezier curve-to, and more.

Other important elements include `defs` and `use` for defining objects and later referencing them, `image` for embedding legacy raster image files (e.g., PNG or JPEG graphics), `text` for including text, and `g` for grouping sub-elements to be rendered as a single entity.

A program that reads an SVG file has access to the internals of the image via the SVG Document Object Model [17, Appendix B]. The SVG DOM is compatible with the basic XML DOM [2] and is a proper extension of the DOM Core [23]. The DOM permits access to the SVG element tree, including allowing the manipulation of element attributes. For example, to increase the size of a text element, we can write the following code in ECMAScript [16] (a standardized version of JavaScript).

```
e = document.getElementById("TextElement");
e.setAttribute("transform", "scale(2)");
```

and the selected element will be scaled to twice its normal size. The SVG DOM can be used in combination with scripting and event handlers (e.g., `mousedown`, `onclick`) to permit some useful interactive capabilities.

SVG also contains several animation elements that describe time-based perturbation of the containing object. These elements can be used to achieve motion along paths, the fading in or out of objects, changes in color, and more. For example, to animate moving a rectangle horizontally across the viewport to the right, we write:

```
<rect x="20" y="10" width="10" height="5"/>
  <animate attributeName="x"
           attributeType="XML"
           begin="0s" dur="9s" fill="freeze"
           from="20" to="120"/>
</rect>
```

Most elements contain attributes to control especially important properties of the described object, such as its position and size. Numerous other properties of objects are set using a single attribute called `style`. That attribute is the access point to a powerful style description language called Cascading Style Sheets.

CSS: Cascading Style Sheets

The Cascading Style Sheets (CSS) [12] recommendation was introduced by the W3C in association with the HTML 4.0 standard. CSS provides a rich set of "style" properties for various HTML and SVG tags. By setting the value of these properties, the document author can control how the browser will display each element.

SVG images can directly annotate elements in the document with style properties via the `style` attribute. Alternatively, the author can place this information in a separate style sheet and then link or import that file. Thus, the same document may be displayed using different style sheets and the same style sheet may be used for multiple documents, easing maintenance of a uniform look for a web site.[1] For example, in Figure 3 the `svg` element specifies a `style` attribute with the multi-part string value:

```
fill: none;            font-size: 15;
stroke: black;         stroke-width: 1;
text-anchor: middle
```

Each of the above five CSS *declarations* is a *property–value* pair. For example "font-size: 15" specifies that the property "font-size" should take on the value "15". Because all of these style properties are specified on the `svg` root element, the styles they set are inherited by each child element, unless they are overridden. The CSS standard specifies complex rules for determining the final value for a property from the multiple declarations that could influence it—this is called "cascading." An earlier paper discusses a constraint extension to CSS that declaratively formalizes these rules using constraint hierarchy theory and also demonstrates some extensions that provide greater expressiveness [5].

Cassowary Constraint Solving Toolkit

Cassowary is our constraint solving toolkit that supports arbitrary linear arithmetic constraints [4]. Constraints can be either equalities or inequalities over real-valued variables. Each constraint can be either required (hard) or preferred (soft). Arbitrarily many levels of preference can be handled, but we typically use only three: strong, medium, and weak. Applications specify sets of constraints and strengths, and the constraint solver assigns values to the variables to satisfy the constraints. All required constraints must be exactly satisfied, and the various non-required constraints are satisfied as well as possible. Cassowary handles cycles without difficulty.

Constraint hierarchy theory [9] provides a declarative semantics of what constitutes a correct solution. For Cassowary, we use the weighted-sum-better comparator for choosing a single solution from among those that satisfy all the required constraints. This comparator computes the error for a solution by summing the product of the strength and the error for each constraint that is unsatisfied. Strengths are represented as tuples: strong is $(1, 0, 0)$, medium is $(0, 1, 0)$, and weak is $(0, 0, 1)$. We order the errors lexicographically so that a strong constraint is infinitely more important than all of the medium and weak constraints.

Client applications use soft constraints to control what solutions are chosen—they are a means of manipulating the objective function for the optimization. An important use of non-required constraints is to enforce stability in graphical layout. We typically add a weak "stay" constraint on each variable's value which states that a variable's future value should be its current value. These stay constraints make objects remain in place unless some other stronger desire forces a change.

The Cassowary constraint solving algorithm is an incremental version of the simplex algorithm that we have optimized for interactive graphical applications. The simplex algorithm is a well-known and heavily studied technique for finding a solution to a collection of linear equality and inequality constraints while minimizing the value of a linear *objective function* [31, Section 2.5].

CSVG: CONSTRAINT SCALABLE VECTOR GRAPHICS

As previously mentioned, a primary advantage of Scalable Vector Graphics is resolution independence. The conventional means of delivering an image is to render the figure, then send the figure across the network in a rasterized image format such as PNG or JPEG (Figure 4). The resolution is fixed when that file is created, and the artifact the user receives is inflexible. The adoption of the SVG image format permits a different delivery mechanism (Figure 5). The high-level image description is stored in the SVG image format, preserving much of the semantic value provided by the author. That SVG file is then sent across the network, where an SVG renderer on the client side chooses the resolution and creates a rasterized display of that image specially-tuned for the display device and the desired size.

The key observation concerning the evolution from raster images to SVG is that we are sending a higher-level description across the network and moving some of the processing of the image from the server side to the client side. Thus, the artifact sent across the Internet is more flexible—it can be used as the source for generating a high-quality printout of the image, to create a low-resolution thumbnail of the image,

[1] Unfortunately, few SVG renderers currently support separating the style sheet from the SVG document—with some implementations, only style properties set via the `style` attribute are honored.

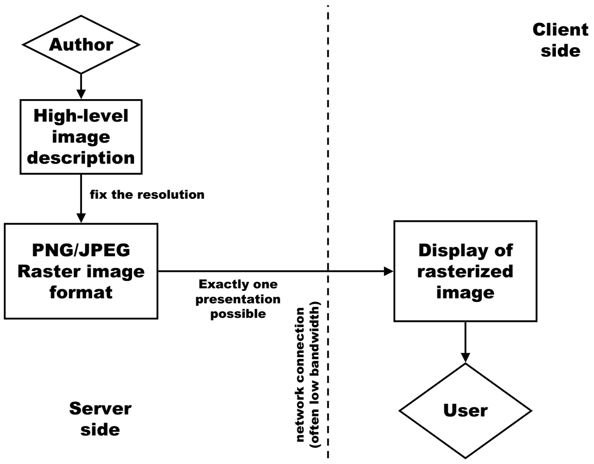

Figure 4: The conventional process of delivering a raster image across the network.

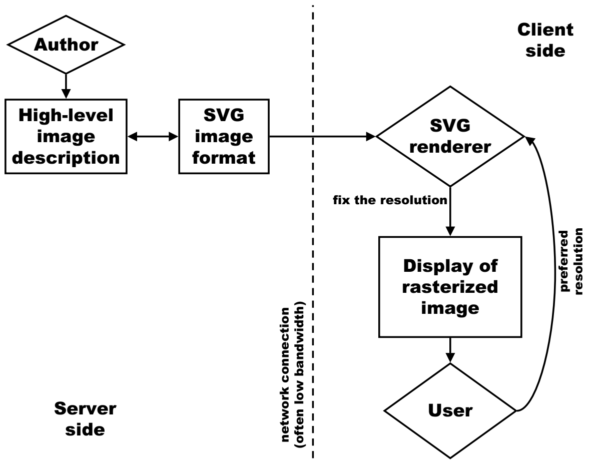

Figure 5: The process of delivering a resolution-independent SVG image across the network.

or even to "render" the image aurally using speech synthesis to describe the diagram. The decision of how to present the image is made with input from the user, her browser, and other client-side software. Style sheets provide yet another way to increase the flexibility of the image sent over the network: not only is the resolution left undetermined, but the final decision as to, for example, the coloring scheme, can be delayed until after applying style sheet declarations.

Extending SVG

Our constraint extension to SVG permits describing the author's layout intentions, and defers the actual positioning and sizing of the image's elements until just before final display for the user (Figure 6). To support this greater flexibility, we have made three extensions to the SVG language. First, we add a new element type called `constraint` and permit those elements to be children of the `svg` root element. Each `constraint` element has a required attribute, `rule`, and an optional attribute, `strength`. Second, we support identifier names in place of literal numbers in all attribute and style sheet values. Thus, we can write:

```
<constraint rule="rect_w >= rect_h"
            strength="strong"/>
<rect x="10" y="20"
      width="rect_w" height="rect_h"/>
```

to express the desire that the rectangle be at least as wide as it is tall. The rule implicitly introduces new constraint variables.[2] Third, we add several built-in read-only constraint variables. (A read-only variable is one that cannot be changed by the solver to satisfy

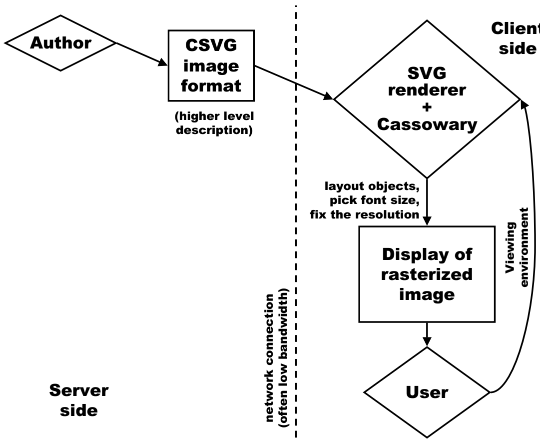

Figure 6: The process of delivering a CSVG image across the network.

the constraint in which it occurs [9].) Two variables, `viewport_width` and `viewport_height`, are used to allow the image to be influenced by the size of the display area. We expose `current_time` and `current_time_squared` which are both ever-increasing read-only variables that allow CSVG to support the declarative specification of time-based animations more directly than the `animate` elements.

CSVG permits image descriptions to be at a higher level of abstraction than an ordinary SVG file. Instead of forcing the author to specify exact values for positions and sizes, the CSVG author can use meaningful names for values and enumerate desired relationships among those values. Similar to how SVG defers choosing the display resolution to later in the delivery pipeline, CSVG

[2] Our syntax was chosen for simplicity. It may be useful to require explicit introduction of variables and to use a separate XML namespace for our extensions so that SVG renderers without a constraint engine could still handle CSVG images.

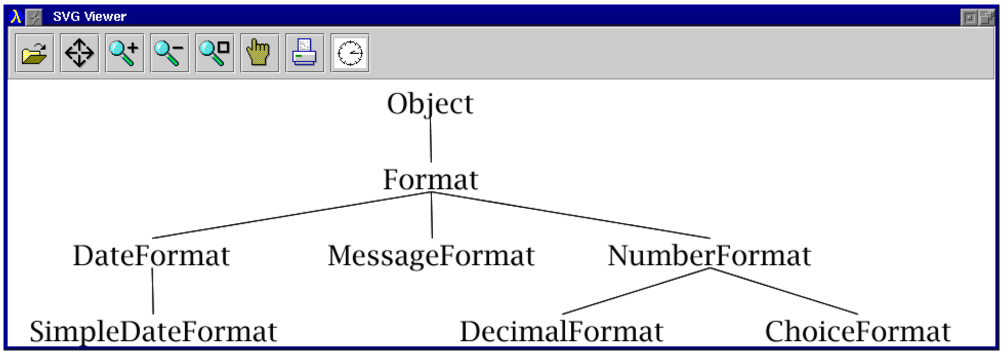

Figure 7: CSVG rendering of the `Format` class hierarchy inside a wide and short viewport.

delays finalizing the *layout* of the illustration until the client side (Table 1).

Table 1: Where properties of a graphic becomes fixed.

Image format	**Resolution**	**Style**	**Layout**
PNG/JPEG	server	server	server
SVG	client	server	server
SVG + CSS	client	client	server
CSVG + CSS	client	client	client

A Layout Example

We can rewrite Figure 3 to specify constraints on the layout of the class hierarchy, rather than giving exact locations for all the parts of the illustration. Our CSVG description of the image looks like the ordinary SVG image (Figure 2) under "ideal" viewing conditions. However, the CSVG file is far more flexible, and it will appear as shown in Figures 7 and 8 when the viewport dimensions are altered. An ordinary SVG file would always appear as just a uniformly scaled version of Figure 2.

For our CSVG version of the class hierarchy, we use a total of 77 constraints that reflect typical layout desires for viewing trees: nodes at the same level are aligned horizontally (4), different levels are spaced at equal vertical intervals (8), there is a minimum gap between adjacent nodes on the same level (4), and parent nodes are above and midway between their edge children (5) [31, p. 204]. Of the remaining 56 constraints, 32 are used to keep the text inside the viewport, 16 are used to declare connection points for the lines, and the last 8 are for setting the margin parameters and controlling the font size. An abridged version of the CSVG source is in Figure 10.

Of course, many of these constraints are redundant and could be eliminated through analysis. Because the Cassowary algorithm handles cycles without difficulty, the redundancies are not a problem, though they do impact performance. A CSVG image for frequent use would likely be optimized before distribution.

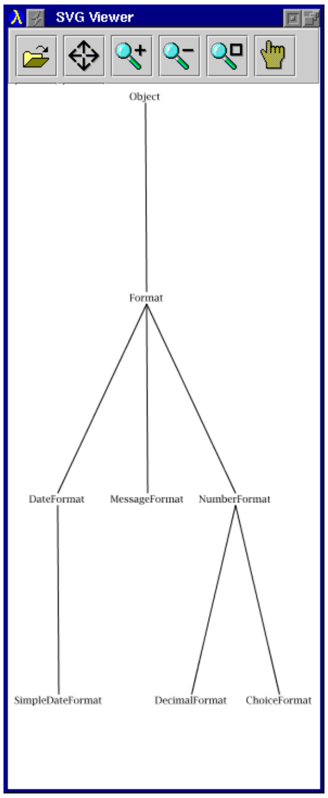

Figure 8: CSVG rendering of the `Format` class hierarchy inside a narrow and tall viewport.

An Animation Example

Constraints relating object positions to the current time can be used to support simple animations. Constraints for layout are even more compelling when parts of the image are moving: the positions of the remaining objects can be described at a high level, knowing that the solver will animate whatever other objects need to move to maintain the specified constraints.

Figure 9 shows four screenshots of our CSVG prototype rendering an animation of a ball falling on a seesaw. The `seesaw.csvg` image contains 18 constraints to support the animation: 12 for the positions of the various elements, 1 relating the ball to the `current_time_squared` built-in variable, 1 stating that the ball must remain above the left edge of the seesaw, and 4 describing that the seesaw cannot go through the floor nor through the fulcrum.

IMPLEMENTATION

On the client side of the pipeline, we have implemented a CSVG viewer to experiment with the additional expressiveness it provides. Our prototype is based on version 0.71 of the CSIRO SVG Viewer [35]. That SVG viewer is implemented in Java, and it uses IBM's XML4J parser version 2.0.15 [26]. For parsing the constraint rule expressions, we use JLex [8], a lexical analyzer gen-

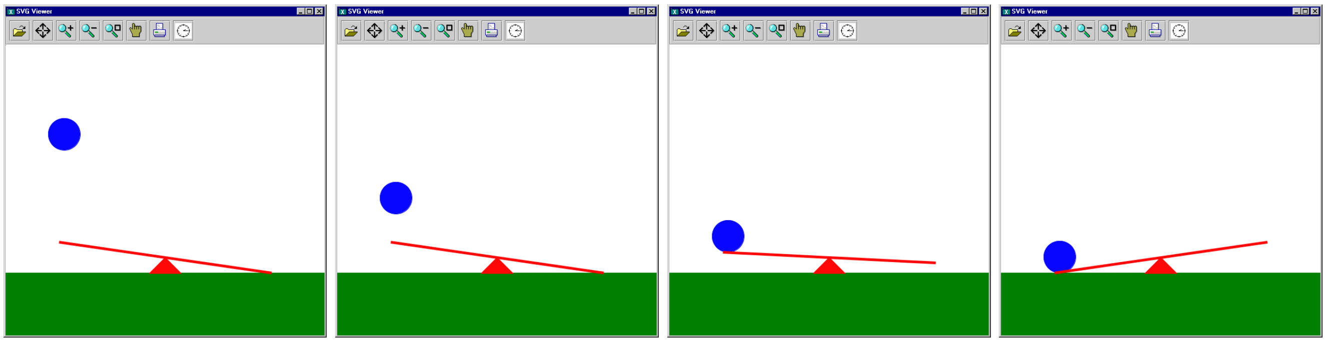

Figure 9: CSVG animation of a ball falling towards seesaw. The position of the ball is directly related to time, and the seesaw moves because of constraints describing its behavior.

erator (similar to Lex), and CUP (Constructor of Useful Parsers) [25], an LALR parser generator (similar to YACC). For solving the constraint systems and laying out the figure, we embedded our Java implementation of the Cassowary Constraint Solving Toolkit [4].

As with any XML language, CSVG is defined by its Document Type Definition. Our CSVG DTD is a straightforward extension of the SVG DTD: we added the `constraint` element and specified its two attributes, `rule` (required) and `strength` (implicit, defaulting to `strong`):

```
<!ELEMENT constraint EMPTY >
<!ATTLIST constraint
   rule CDATA #REQUIRED
   strength CDATA #IMPLIED>
```

Additionally, we added the `constraint` element to the list of permissible children of `svg` elements:

```
<!ELEMENT svg (defs?,desc?,title?,
  (path|text|...|constraint)*)>
```

No other changes to the SVG DTD were necessary to support using identifiers inside of attribute expressions. (However, further changes would be necessary with the more sophisticated data description that XML Schema allows.)

After the XML parser reads in the SVG document, we handle constraint elements by creating a new constrainable variable for each unique identifier contained in a constraint rule. For each variable, we add a stay constraint on it to ensure stability of the resulting figure. Then, for each constraint element, we create a constraint object by parsing the rule attribute's string. Finally, we add each constraint to the global solver.

As we build the internal representation of the image, we store the names of variable identifiers that are used as an attribute's value. Then, whenever we render the figure, we retrieve the values of attributes as usual, with one extra step: if the attribute is an identifier, we then look up that constraint variable's value and use it. For `path` elements, we prefix names of constraint variables with the `$` symbol to avoid ambiguity. For example, we write:

```
<path d="M $x $y l $dx $dy"/>
```

to move to the absolute coordinates held in `x` and `y`, and then draw a line to the relative coordinates contained in variables `dx` and `dy`.

On our Xeon Pentium III 550 MHz test machine running Java 1.3beta-0 with the HotSpot virtual machine under Windows NT 4.0, the performance of our prototype is very good. For our class hierarchy example that contains 77 constraints, the adding of the constraints and the initial solve requires only 360 ms. Subsequent re-solves of the constraint system after resizing the window require less than 200 ms each. Thus, re-rendering the figure after changing the viewport size takes only slightly longer than for the ordinary SVG viewer. Performance would be even better if we removed redundant constraints or if we further optimized our implementation.

On the server side, our class hierarchy diagram example was largely mechanically-derived from an XML-based representation of Java source code, JavaML [3]. Using XSLT [14], it is reasonably straightforward to generate CSVG from the JavaML representation.

RELATED WORK

As mentioned earlier, style-sheet technologies, such as CSS (Cascading Style Sheets) [12], PSL (Proteus Style Language) [30], DSSSL (Document Style Semantics and Specification Language) [28], and XSL (eXtensible Style Language) [14], each delay finalizing various presentational attributes of a figure until later in the delivery process, closer to the viewing user. None of these style languages, however, attempt to preserve layout desires to perform layout dynamically on the client side.

Our constraint extensions to Cascading Style Sheets, CCSS, demonstrate how CSS can be understood in terms of constraints, and they add expressiveness given that more general framework [5]. Our CSVG motivation and philosophy is analogous to that of CCSS, and CCSS is directly applicable to controlling style properties of CSVG documents as well. The primary addition

```
<?xml version="1.0"?>
<!DOCTYPE svg SYSTEM "csvg.dtd">
<svg width="4.5in" height="4in"
     viewBox="0 0 100 100"
     style="fill: none; stroke: black;
            stroke-width: 1">
 <desc>The object hierarchy surrounding class
       "Java.text.Format"</desc>
 <constraint rule="fh >= 9"/>
 <constraint
   rule="vert_spacing = vp_height / 3.5"/>
 <constraint rule="text_w * 4 = vp_width"/>
 ...
 <!-- stay inside viewport  -->
 <constraint
   rule="o_x >= side_margin + h_text_w"/>
 <constraint
   rule="o_x &lt;=
         vp_width - side_margin - h_text_w"/>
 <constraint
   rule="o_y >= top_margin + fh"/>
 <constraint
   rule="o_y &lt;= vp_height - top_margin"/>
 ...
 <!-- layout between children and parents -->
 <constraint
   rule="(dtf_x + nf_x) / 2 = f_x"
   strength="strong"/>
 <constraint
   rule="f_y >= o_y + vert_spacing"
   strength="strong"/>
 ...
 <!-- same level at same y coordinate -->
 <constraint rule="dtf_y = mf_y"/>
 ...
 <!-- same level spread out horizontally -->
 <constraint rule="dtf_x + text_w &lt;= mf_x"/>
 ...
 <!-- the text elements for each class -->
 <g style="font-size: fh; text-anchor: middle">
   <text x="o_x" y="o_y">Object</text>
   <text x="f_x" y="f_y">Format</text>
   <text x="dtf_x" y="dtf_y">DateFormat</text>
   <text x="mf_x" y="mf_y">MessageFormat</text>
   <text x="nf_x" y="nf_y">NumberFormat</text>
   ...
 </g>

 <!-- lines connecting parents to children -->
 <line x1="o_x" y1="o_y_b"
       x2="f_x" y2="f_y_t"/>
 <line x1="f_x" y1="f_y_b"
       x2="dtf_x" y2="dtf_y_t"/>
 ...
</svg>
```

Figure 10: CSVG source of the object hierarchy surrounding the `Java.text.Format` class. The `<` inside of rule attribute values is an XML entity that represents the “<” symbol.

of CSVG beyond CCSS is the ability to control non-style properties of SVG elements. This feature is necessary to control layout because the positions of those objects are determined not by style properties but by element attributes. An earlier paper [10] had goals similar to CCSS, but did not integrate well with the emerging web standards.

Kim Marriott (a co-author on our Cassowary and CCSS work) and his colleagues have independently done some preliminary work on constraint extensions to SVG. They use MathML to describe constraints (instead of a string), use the QOCA algorithm which uses a least-squares-better comparator but is otherwise similar to Cassowary, and support a limited form of disjunctions modeled after our preconditions for CCSS [39]. Diehl and Keller describe constraint extensions to the Virtual Reality Markup Language (VRML) basd on a local propagation based solver that is unable to handle cycles or inequality constraints [15].

The animation aspects of SVG and CSVG are related to the Synchronized Multimedia Integration Langauge (SMIL) [24]. Another project called Madeus has used the Cassowary solver to handle a wider range of constraints in multimedia documents [38]. Madeus provides support for both temporal and spatial relationships, and it includes a rudimentary authoring environment.

There is a long history of using constraints in interfaces and interactive systems, beginning with Ivan Sutherland's pioneering Sketchpad system [37]. Juno-2 is a more recent constraint-based drawing application [22]. Constraints have also been used in several other layout applications. IDEAL [40] is an early system specifically designed for page layout applications. Harada, Witkin, and Baraff [20] describe the use of physically-based modeling for a variety of interactive modeling tasks, including page layout. GLIDE [36] uses visual organization features (VOFs) to control layout of arbitrary graphs using a spring metaphor and an iterative numeric solver. Numerous systems use constraints for widget layout [32, 33], and Badros [7] uses constraints for window layout.

CONCLUSIONS AND FUTURE WORK

Our constraint extension to SVG provides useful new expressiveness for describing illustration graphics at a higher semantic level. CSVG permits deferring the actual layout of the objects in the figure until the final rendering, thus resulting in greater flexibility in dealing with varied viewing environments and user desires. The implementation of our prototype system was straightforward because we were able to leverage our Cassowary constraint solving toolkit.

There are substantial opportunities for future improvements of CSVG. Currently, there are no authoring environments that preserve the author's intentions sufficiently well to generate CSVG at the appropriate level of abstraction. It is essential that a drawing program permit users to specify constraints interactively, dynamically maintain them throughout editing, and ultimately reflect those constraints in the saved CSVG file. Noth's CDA [34] or an SVG-capable editor such as Adobe Illustrator^{tm} or Sketch [21] may provide a useful starting point.

Even in the presence of graphical editing tools for CSVG, it may be beneficial to provide some syntactic sugar for CSVG. Future versions of CSVG could support referencing other elements' attributes directly. Additionally, CSVG could easily support using arbitrary expressions, instead of just identifiers, for attribute values. Such expressions would provide non-linear and non-numeric constraints over read-only variables. Extending the power of the constraint solving algorithms would permit some of these kinds of constraints over read-write variables. For example, a text element in a CSVG document could be constrained to display the coordinates of a circle: moving the circle would update the string, and editing the string would move the circle.

It may also be useful to permit even higher-level constraint abstractions in the CSVG source. For example:

```
<align dir="horizontal" anchor="middle">
  <!-- arbitrary basic shape objects here -->
</align>
```

would permit easier specification of the intention that a set of basic shapes are aligned in a row by their vertical centers. Constraints at this level also avoid problems that arise when object structure changes. Suppose a basic shape is removed from a diagram (e.g., using the SVG DOM): should indirect relationships through that object remain or be removed? If only the primitive constraints are present, the situation is ambiguous. With multiple objects being aligned with a single declaration, the answer is more clearly that those objects should remain aligned.

Another area for future work is to better describe the semantics of the SVG in terms of constraints and constraint hierarchy theory. This direction is similar to what we did for Constraint Cascading Style Sheets [5] and it may provide a unifying implementation mechanism for parts of SVG as well. In particular, some of the scripting events, such as `onMouseMove`, may be handled within this framework: a discrete action (such as a button press) establishes a connection that then is managed via a constraint relationship until a subsequent action removes the constraint [29].

Overall, CSVG provides a surprising amount of expressiveness at a minimal implementation complexity, and at a low performance cost.

ACKNOWLEDGMENTS

Thanks to Vincent Hardy of the SVG working group for his helpful feedback on our work, and to Denise Pinnel for her comments on a draft of this paper. This research has been funded in part by both a National Science Foundation Graduate Research Fellowship and the University of Washington Computer Science and Engineering Wilma Bradley fellowship for Greg Badros, and in part by NSF Grant No. IIS-9975990.

Availability

Our prototype CSVG renderer, complete versions of both examples described here, and the Cassowary constraint solving toolkit are all freely available on the Internet [4, 6] and are distributed under the terms of the GNU General Public License.

REFERENCES

1. Adobe Systems Incorporated. *PostScript Language Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2nd edition, 1990.
2. V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. Le Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, and L. Wood. Document object model (DOM) level 1. W3C Recommendation, October 1998. `http://www.w3.org/TR/REC-DOM-Level-1`.
3. Greg J. Badros. JavaML: A markup language for java source code. In *Proceedings of the Ninth International Conference on the World Wide Web*, Amsterdam, The Netherlands, May 2000. Elsevier Science B. V. `http://www.cs.washington.edu/homes/gjb/JavaML`.
4. Greg J. Badros and Alan Borning. The Cassowary linear arithmetic constraint solving algorithm: Interface and implementation. Technical Report UW-CSE-98-06-04, University of Washington, Seattle, Washington, June 1998. `http://www.cs.washington.edu/research/constraints/cassowary/cassowary-tr.pdf`.
5. Greg J. Badros, Alan Borning, Kim Marriott, and Peter Stuckey. Constraint cascading style sheets for the web. In *Proceedings of the 1999 ACM Conference on User Interface Software and Technology*, November 1999.
6. Greg J. Badros, Will Portnoy, Jeff Nichols, and Alan Borning. CSVG—Constraint Scalable Vector Graphics. Web page, 2000. `http://www.cs.washington.edu/homes/gjb/csvg/`.
7. Greg J. Badros and Maciej Stachowiak. Scwm—The Scheme Constraints Window Manager. Web page, 1999. `http://scwm.mit.edu/scwm/`.
8. Elliott Berk and C. Scott Ananian. Jlex: A lexical analyzer generator for java. Web Page, 2000. `http://www.cs.princeton.edu/~appel/modern/java/JLex/`.
9. Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, September 1992. `http://www.cs.washington.edu/research/constraints/theory/hierarchies-92.html`.

10. Alan Borning, Richard Lin, and Kim Marriott. Constraints for the web. In *Proceedings of 1997 ACM Multimedia Conference*, 1997.

11. Alan Borning, Kim Marriott, Peter Stuckey, and Yi Xiao. Solving linear arithmetic constraints for user interface applications. In *Proceedings of the 1997 ACM Symposium on User Interface Software and Technology*, October 1997.

12. Bert Bos, Håkon Wium Lie, Chris Lilley, and Ian Jacobs. Cascading style sheets, level 2. W3C Working Draft, January 1998. `http://www.w3.org/TR/WD-css2/`.

13. Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0. W3C Recommendation, February 1998. `http://www.w3.org/TR/REC-xml`.

14. James Clark. XSL transformations. W3C Recommendation, November 1999. `http://www.w3.org/TR/xslt`.

15. Stephan Diehl and Jörg Keller. VRML with constraints. In *Proceedings of the Web3D-VRML 2000 fifth symposium on Virtual reality modeling language*, Monterey, California, February 2000. `http://www.cs.uni-sb.de/RW/users/diehl/VRMLCONSTR/VRMLConstr.html`.

16. ECMAScript language specification, 3rd ed., December 1999. `ftp://ftp.ecma.ch/ecma-st/Ecma-262.pdf`.

17. Jon Ferraiolo. Scalable vector graphics (SVG) 1.0 specification. W3C Working Draft, December 1999. `http://www.w3.org/TR/1999/WD-SVG-19991203/`.

18. Charles F. Goldfarb and Paul Prescod. *The XML Handbook*. Prentice Hall PTR, 1998.

19. Michael Goosens and Sebastian Rahtz. *The LaTeX Web Companion*. Addison Wesley Longman, 1999.

20. Mikako Harada, Andrew Witkin, and David Baraff. Interactive physically-based manipulation of discrete/continuous models. In *Proceedings of SIGGRAPH 1995*, pages 199–208, Los Angeles, August 1995. ACM.

21. Bernhard Herzon. Sketch, a vector drawing program for unix. Web page, 2000. `http://sketch.sourceforge.net/`.

22. Allan Heydon and Greg Nelson. The Juno-2 constraint-based drawing editor. Technical Report 131a, Digital Systems Research Center, Palo Alto, California, December 1994.

23. Arnaud Le Hors, Mike Champion, Steve Byrne, Gavin Nicol, and Lauren Wood. Document object model core. W3C Working Draft, September 1999. `http://www.w3.org/TR/1999/WD-DOM-Level-2-19990923/core.html`.

24. Philipp Hoschka. Synchronized multimedia integration language. W3C Recommendation, June 1998. `http://www.w3.org/TR/REC-smil/`.

25. Scott Hudson and C. Scott Ananian. CUP parser generator for Java. Web page, 1999–2000. `http://www.cs.princeton.edu/~appel/modern/java/CUP/`.

26. IBM AlphaWorks. XML for Java. `http://www.alphaworks.ibm.com/tech/xml4j`.

27. ISO. Standard generalized markup language (SGML). ISO 8879, 1986. `http://www.iso.ch/cate/d16387.html`.

28. ISO/IEC. Document style semantics and specification language (DSSSL). ISO/IEC 10179, 1996.

29. Robert J. K. Jacob, Leonida Deligiannidis, and Stephen Morrison. A software model and specification language for non-wimp user interfaces. *ACM Transactions on Computer-Human Interaction*, 6(1):1–46, March 1999. `http://www.acm.org/pubs/articles/journals/tochi/1999-6-1/p1-jacob/p1-jacob.pdf`.

30. Philip M. Marden, Jr. and Ethan V. Munson. PSL: An alternate approach to style sheet languages for the world wide web. *Journal of Universal Computer Science*, 4(10), 1998. http://www.cs.uwm.edu/~multimedia.

31. Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, Cambridge, Massachusetts, 1998.

32. Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive support for graphical highly interactive user interfaces. *IEEE Computer*, November 1990.

33. Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and Patrick Doane. The Amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6):347–365, June 1997.

34. Michael Noth. Constraint drawing applet. Web page, 1998. `http://www.cs.washington.edu/research/constraints/cda/info.html`.

35. Bella Robinson and Dean Jackson. SVG toolkit. Web page, 1999–2000. `http://sis.cmis.csiro.au/svg/`.

36. Kathy Ryall, Joe Marks, and Stuart Shieber. An interactive constraint-based system for drawing graphs. In *Proceedings of UIST 1997*, Banff, Alberta Canada, October 1997.

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

38. Laurent Tardif, Frédéric Bes, and Cécile Roisin. Constraints for multimedia documents. In *Proceedings of the Second International Conference and Exhibition on the Practical Application of Constraint Technology and Logic Programming*, Manchester, United Kingdom, April 2000.

39. Jojada J. Tirtowidjojo, Kim Marriott, and Bernd Meyer. Extending svg with constraints. In *Proceedings of the Sixth Australian World Wide Web Conference*, Cairns, Queensland Australia, June 2000. Poster to appear. `http://www.dgs.monash.edu.au/~jojada/ConstraintSVG.html`.

40. Christopher J. van Wyk. A high-level language for specifying pictures. *ACM Transactions on Graphics*, 1(2):163–182, April 1982.