Extending SVG Fonts with Graphite

Chris Lilley, W3C
Sharon Correll, SIL International

1. Introduction

Scalable Vector Graphics (SVG) is an XML format developed by the W3C for the description of vector graphics. It includes its own font format. Care was taken in the internationalization aspects of this format and it is capable of rendering text in simple writing systems. SVG content creators would love support of more languages, but SVG implementers are wary of adding an open-ended amount of special knowledge to the font rendering system particularly on resource-constrained mobile devices.

Graphite is an extensible system that allows the declarative description of features of new writing systems.

By transferring some of the language support effort from SVG implementers to SVG content creators, it is thought that a way forwards can be found. A system for declaring the properties of an unknown writing system gives implementers a fixed target (an engine to understand the description) instead of requiring open-ended, near omniscient knowledge of all the worlds current and historical writing systems. The simple should be easy and the hard, possible.

This paper presents some initial thoughts on ways in which concepts from the Graphite system could be used to extend the SVG Font system. It raises more questions than it answers, and is intended as the opening stage in a dialogue, not a closing summary.

2. SVG

SVG represents graphics as a tree of vector objects - that is, objects which are described by their geometry rather than by an array of pixels. Since text is a frequent component of graphics, SVG also allows textual content, which is stored as plain marked-up text, in whatever encoding the XML file uses (typically UTF-8 or UTF-16). Characters which cannot conveniently be entered by the keyboard or input methods available to the content creator can use Numeric Character References (NCRs) thus making available the entire Unicode character repertoire.

For portability across different platforms and to give greater artistic control, SVG also includes its own font format. SVG fonts are referenced from and applied to text with CSS style sheets, using the CSS2 WebFont mechanism. The glyph shapes use the same attribute syntax as the general SVG path element, thus promoting code reuse, and additional elements are used for glyphs, kern pairs and font metadata. The entire system is written in declarative XML.

Basic international text in simple scripts is supported, including bidirectional and vertical writing, ligatures, alternate glyphs, language-specific glyphs and the four basic contextual forms

of Arabic glyphs; however the format is currently unable to describe more complex writing systems.

3. Graphite

Graphite is a tool developed by SIL International to provide rendering of complex writing systems. It is particularly oriented toward languages of South and Southeast Asia and the Middle East which are characterized by complex reordering and a high degree of contextualization and ligation. Graphite has three components:

- a high-level, rule-based programming language, the "Graphite Description Language" (GDL)
- a GDL compiler, which stores its results in custom tables in a TrueType font or in a file functioning as an extension to the font
- a rendering engine that uses the resulting TrueType font files

A declarative language is a clear candidate for expression in XML, and could be used to extend the capabilities of SVG Fonts by describing exactly what processing a given writing system requires.

4. Existing features

4.1 Character to glyph mapping

In SVG, the font element holds the metadata such as font name, weight, number of units on the em square (defining the coordinate space withing which glyphs are laid out) and so on. Each font element has a 'missing glyph' child and any number of 'glyph' children each of which holds a single glyph definition. The geometry of the glyph and its metadata (width, etc) are attributes of the glyph element. There is no explicit cmap instead, each glyph has a 'unicode' attribute that holds the character (in fact, the string – see below) for which it is a glyph. Thus, multiple glyphs can map to the same character.

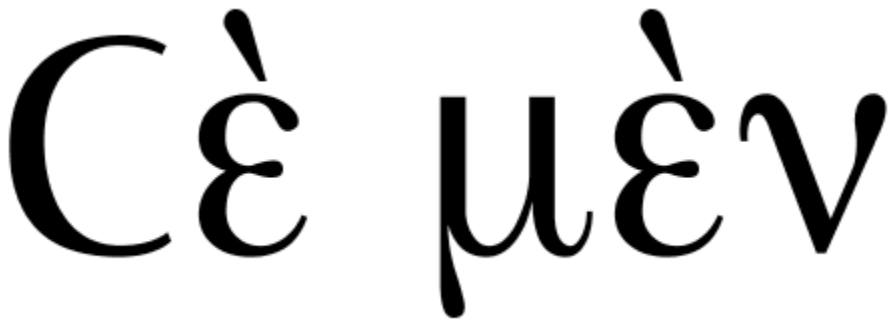
Here is a simple example font, derived from SIL Galatia but converted to SVG and re-encoded to use Unicode; the file also contains some Greek text (including the upper case lunate Sigma introduced in Unicode 4) rendered in that font:

```
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/SVG/DTD/svg10.dtd">
<svg width="100%" height="100%" viewBox="0 0 200 80">
    <defs>
        <font horiz-adv-x="478">
            <font-face font-family="SVG SIL Galatia-U" units-per-em="1024"
            panose-1="0 0 4 0 0 0 0 0 0 0" ascent="939" descent="-230" alphabetic="0"/>
            <missing-glyph horiz-adv-x="512" d="M64 0V824H448V0H64ZM128 64H384V760H128V64Z"/>
            <glyph unicode="&#x20;" glyph-name="space" horiz-adv-x="325"/>
            <glyph unicode="&#x1f72;" glyph-name="epsilon-with-varia" horiz-adv-x="428" d="M264 552H239L133 720Q127 729
127 741Q127 757 137 770T164 784Q198 784 211 740L264 552ZM395 116Q353 -12 212 -12Q139 -12 93 26T46 125Q46 178 81
220Q105 250 135 264Q46 289 46 373Q46 431 89 470Q136 512 216 512Q282 512 327 481Q367 453 367 423Q367 409 356
401T333 392Q313 392 301 415T275 461Q256 485 224 485Q132 485 132 373Q132 326 157 301Q177 281 207 281Q215 281 238
289T286 298Q317 298 317 270Q317 243 282 243Q263 243 240 250T209
258Q183 258 162 233Q135 199 135 135Q135 85 169 53Q200 24 245 24Q287 24 323 51T378 123L395 116Z"/>
            <glyph unicode="&#x3bc;" glyph-name="mu" horiz-adv-x="586" d="M560 135Q560 74 538 33Q514 -12 473 -12Q370
-12 366 169Q352 95 317 46Q276 -12 218 -12Q127 -12 95 92Q95 16 122 -66T149 -171Q149 -207 113 -207Q70 -207 70
```

```
-93V502H152V135Q152 94 174 59T227 24Q284 24 324 115Q357 190 357 256V502H439V125Q439 74 455 47Q469 24 489
24Q508 24 525 57T542 135H560Z"/>
            <glyph unicode="&#x3bd;" glyph-name="nu" horiz-adv-x="523" d="M491 467Q491 436 472 393L297 0H267L111
411Q97 447 79 447Q67 447 58 433Q45 415 45 380Q45 372 46 360H23Q20 379 20 389Q20 512 107 512Q160 512 199 409L312
110L383 272Q396 302 396 360Q396 369 393 392T390 426Q390 459 402 482Q418 512 450 512Q467 512 479 499T491 467Z"/>
            <glyph unicode="&#x3f9;" glyph-name="Sigmalunate" horiz-adv-x="601" d="M559 39Q500 -12 391 -12Q38 -12 38
347Q38 507 128 600Q225 700 407 700Q486 700 543 666L529 637Q469 667 390 667Q291 667 223 590Q150 505 150 363Q150
221 208 129Q274 25 394 25Q488 25 547 68L559 39Z"/>
            <hkern g1="epsilon" g2="mu" k="16"/>
        </font>
    </defs>
    <g style="font-family: 'SVG SIL Galatia-U'; font-size:48;fill:black">
        <text x="20" y="60">&#x3f9;è µèv</text>
    </g>
</svg>
```

As expected, the bulk of this file is actually the Bezier curve descriptions (the ‘d’ attribute); however a complete, working example was felt to be beneficial. Only those glyphs needed to render the sample text are defined in this example.

This is what the Greek SVG Font file looks like when rendered to show the sample text.



4.2 Ligatures

SVG can express ligatures of any length, merely by putting a Unicode string rather than a single character in the ‘unicode’ attribute on the glyph. For example, a ‘Chris’ ligature could easily be defined. This facility is often used for logos.

As an example, here is an SVG font that contains a lam-alef ligature as well as the isolated lam and alef glyphs. (The glyph data has been truncated to save space)

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/SVG/DTD/svg10.dtd">
<svg xmlns="http://www.w3.org/2000/svg" width="100%" height="100%">
  <font horiz-adv-x="573">
    <font-face font-family="SWAS" units-per-em="1000" panose-1="5 1 1 1 1 1 1 1 1 1"
      ascent="1025" descent="-399" alphabetic="0"/>
    <missing-glyph horiz-adv-x="500" d="M31 0V800H469V0H31ZM438 31V769H62V31H438Z"/>
    <glyph unicode="&#x644;&#x627;" glyph-name="lam-alef-ligature" horiz-adv-x="530" d="M474 and so on 89Z"/>
    <glyph unicode="ا" glyph-name="alef" arabic-form="isolated" horiz-adv-x="288" d="M288 and so on H288V0Z"/>
    <glyph unicode="ل" glyph-name="lam-isolate"  arabic-form="isolated" horiz-adv-x="518" d="M467  and so on 5Z"/>
  </font>
</svg>
```

This font has three glyphs (besides the 'missing glyph') – isolated alef, isolated lam, and a lam-alef ligature. For clarity and to show that the 'unicode' attribute contains two characters, numeric character references have been used for the ligature. The 'glyph-name' attribute is primarily for documentation purposes in this example.

To avoid the complexities and ambiguities of 'longest string' matches, SVG specifies that the first matching glyph is used. This means that ligatures need to be ahead of the single-character glyphs in the SVG font.

Notice the hard-coded 'arabic' attribute that gives at least basic legibility for Arabic text by allowing the four contextual glyph forms of each character to be used in a font. A more generic and open ended solution would be better in the long run than a succession of specific and non-extensible attributes.

Graphite, too, can express ligatures. Here for example is an OE ligature:

```
clsOELig = codepoint((140, 156)) {
   component.o = box(0,0, 50m,100m);
   component.e = box(50m,0, 100m,100m)
}
clsO clsE > clsOELig:(1 2) {component {o.ref=@1; e.ref=@2}} _;
```

The first part defines the glyph for the ligature, in terms of two components (for example, to allow sub-glyph cursor positioning); the second part defines the rule that generates the ligature and the respective attachment points of the two component glyphs.

Not usually described as ligatures, but using the same mechanisms, glyphs for sequences of decomposed characters can also be described in SVG fonts in this way, simply by putting the decomposed sequence into the Unicode attribute.

Graphite also can describe composing sequences in this manner. For example:

```
clsBase = glyph-list {
      upperAttPt = point(50m, 75m);
      lowerAttPt = point(50m, 0) }
clsUpperDiac = glyph-list { attPt = point(50m, 0) }
clsLowerDiac = glyph-list { attPt = point(50m, 20m) }
clsBase clsUpperDiac {
      attach.to = @1;
      attach.at = upperAttPt;
      attach.with = attPt }
clsBase clsLowerDiac {
      attach.to = @1;
      attach.at = lowerAttPt;
      attach.with = attPt } ;
```

In SVG, space can be saved by referencing the shapes of other glyphs, and merely positioning the components correctly (for example, the height and position of a diacritic would depend on the shape of the base character). If no transformations are used, the glyphs will just stack on to p of one another which is often good enough. However, in languages such as Vietnamese, the position of a diacritic depends on the base letter and the other diacritics, since there can be several. In these cases, explicit positioning is required.

Graphite allows special paths (in practice, points) in the glyph definition to be defined as attachment points – places where one glyph, such as a diacritic or . This moves responsibility for correct attachment of diacritics from the content creator to the font designer, who may be better

prepared to make these sort of design decisions. Such a facility could usefully be added to SVG, and would remove the need for explicit ligatures for many combining character combinations.

4.3 Kerning

SVG allows for pairwise kerning, for both horizontal and vertical writing, using the ‘hkern’ and ‘vkern’ elements. An example was provided in the Greek font above and is reproduced again here. The ‘g1’ and ‘g2’ attributes give the names of the two glyphs to be kerned, and the ‘k’ attribute specifies the amount to move them – horizontal movement for hkern and vertical movement for vkern – in the coordinate units of the em square. This ensures that kerning does not shift in weird ways at different font sizes, but scales uniformly with font size changes.

```
<hkern g1="epsilon" g2="mu" k="16"/>
```

The kern mechanism is one way that could have been used for correct positioning of diacritics, if both horizontal and vertical movement had been allowed for each kern type.

Graphite defines pairwise kerning in a similar way:

```
gUppercaseA { kern.x = -10m } / clsUppercaseVW _;
```

The difference here is that GDL can reference both individual glyphs (the naming convention being a starting ‘g’) and classes (‘cls’) the latter being in fact more like arrays than sets, as they have ordering. This particular class merely contains the upper case V and W glyphs. Use of classes in SVG could significantly reduce the number of kern elements required, saving download time and memory.

4.4 Alternate glyphs

SVG allows alternate glyph forms, but in a rather restricted manner. We have already met the ‘arabic’ attribute that allows up to four glyph forms for the same character, for Arabic and similar languages that use this contextual positioning.

It is also possible to specify a language (or list of languages) for which a glyph is appropriate. This is matched against the value of the ‘xml:lang’ attribute of the text to which the font is applied. If the attribute is not set, then the glyph can be used for all languages.

Here is an example where three different glyphs have been defined for the same character – U+9AA8, a unified Han ideograph called ‘bone’. When this character is rendered, different glyph forms are expected in Japanese text, in Traditional Chinese text and in Simplified Chinese text. SVG Fonts allow all three glyphs to be used in the same font.

Unicode 9AA8

Glyphs can be referenced in the definition of other glyphs – this is frequently done when making glyphs for composed sequences – and those glyphs do not have to be in the same font, or even in the same file, as a URI is used for the reference.

Lastly, SVG has an ‘altGlyph’ element which can be used directly in the text to set a specific glyph to be used for one particular instance of a character or character string. This overrides all other font processing and essentially treats the font as a symbol library. This method is commonly used for inserting pictorial symbols into a run of text, especially if those symbols have no Unicode value; it is also used to produce swash forms, for example forms which are only used at the end or start of a line. It is however a nuisance and a poor separation of content from presentation to have to put specific markup in the text.

In SVG 1.0 and 1.1, all line breaking was the responsibility of the generating application. The SVG renderer never broke lines. In SVG 1.2, this is no longer true and wrapping text inside arbitrary shapes is supported. This means that detection of linebreak-sensitive swash opportunities should be moved to the renderer; the content creator does not necessarily know where the line breaks will occur.

Graphite has a way to match on line start and line end, by defining a special escape character ‘#’ to match the ends of the line. Again, use of glyph classes makes this much more compact than would otherwise have been the case:

```
clsSwashable > clsWithInitialSwash / # _; // start of line
clsSwashable > clsWithFinalSwash / _ #; // end of line
```

This simply says that glyphs that can be swashed are replaced with their corresponding start of line swash form when the glyph is at the start of a line – and similarly for the end of the line. It

also illustrates that the classesare in fact arrays, since they are ordered so that the corresponding entry in another class can be determined.

5. Additional glyph substitution and rearrangement

Many languages require much more complex glyph substitution that SVG can currently provide, and also require reordering of glyphs compared to the order of the characters. Here for example is a sample of GDL code that shows how to handle vowel reordering in Bengali.

* The members of the "clsVowelSecondHalf" class are really glyphs that happen to be referenced by the given Unicode values. They correspond in number and order to the items in clsVowelSplit.

* In the first rule, the "@" notation replaces the given glyph from the input into the output and also maintains an association of that glyph with the underlying character.

* In the second rule:

- the underscore corresponds to the insertion of the first half of the vowel.

- the ":3" notation associates the first half of the vowel with the third underlying character in the rule (ie, the original vowel).

- when clsVowelSplit is replaced with clsVowelSecondHalf, it automatically chooses the corresponding item based on the order of the items in the two classes.

* Nothing is need to handle the clsVowelPost class, since those vowels are already properly ordered. The class could be left out; I just included it for completeness.

6. Compilation and the Finite State Automaton

Beyond a certain point, multiple layers of indirection through glyph substitutions become unwieldy. The rendering software would need to organize the information in some way to make processing more efficient. However, such organization would imply a possibly lengthy startup time before the first text was rendered, which is unacceptable in many situations.

The Graphite system overcomes this problem by compiling each pass in the GDL and creating instructions for a state machine, a Finite State Automata, or in simple terms the way to go from the input to the output with the fewest number of steps possible. The output of the current GDL compiler is binary information; although this could be stored directly in the XML file, for example using the base64 encoding, it would be preferable to express this essentially hierarchical information in terms of XML provided the resulting data size was not too large. This would allow generation and manipulation by regular XML tools, for example when subsetting a font for delivery.

An additional benefit of compilation is that error checking can be performed before the content is created and delivered. This is particularly important in SVG where dynamically created and modified text is common. Errors in a complex font might not be apparent with the initial state of the displayed text.

As a result of adding the compilation step, creation of a Graphite-assisted SVG font would not be a simple hand-coding affair, but would rely on automated processing. Since most existing font creation in SVG already uses automated tools, this is not seen as a major drawback in practice. The result would be a set of content-specific instructions that are merely followed by the SVG renderer—thus making the implementers happy—with faster startup time and improved capability to deal with complex scripts, thus making content creators happy.

7. Bibliography

Apple Computer, *AAT Font Feature Registry*. Available at http://developer.apple.com/fonts/Registry/index.html

Bos, Bert; Lie, Håkon Wium; Lilley, Chris; Jacobs, Ian (eds) *Cascading Style Sheets, level 2 CSS2 Specification*, W3C Recommendation, 12 May 1998. Available at http://www.w3.org/TR/REC-CSS2/

Bray, Tim; Paoli, Jean; Sperberg-McQueen, C.M.; Maler, Eve (eds) *Extensible Markup Language (XML) 1.0 (Second Edition)* W3C Recommendation, 6 October 2000. Available at http://www.w3.org/TR/REC-xml.html

Correll, Sharon; *Graphite: An Extensible Rendering Engine for Complex Writing Systems,* 17th International Unicode Conference, San Jose, California, September 2000. Available at http://graphite.sil.org/pdf/IUC17_paper.pdf

Correll, Sharon; *Graphite Requirements version 0.91*, SIL International. Available at http://graphite.sil.org/pdf/Graphite_Requirements.pdf

Ferraiolo, Jon; 藤沢 淳 (FUJISAWA Jun); Jackson, Dean (eds); *Scalable Vector Graphics (SVG) 1.1 Specification*, W3C Recommendation, 14 January 2003. Available at http://www.w3.org/TR/SVG11/

Lilley, Chris; *SVG: Unicode Meets Vector Graphics*, 22nd International Unicode Conference, San Jose, California, September 2002.