

Extensible Rendering Technology for Web-based Data Access

Author: Sharon Correll
SIL International
E-mail: sharon_correll@sil.org

Presented at:

Linguistic Exploration: Workshop on Web-based Documentation and Description
12-15 December 2000
Philadelphia, Pennsylvania, USA

Abstract

Graphite is a software development project within SIL International whose purpose is to provide a customizable, Unicode-conformant rendering engine for complex writing systems on the Windows platform.

SIL International, as a non-profit organization engaged in linguistic research and translation work among minority language groups worldwide, has for many years experienced the need for an extensible rendering system. Rendering solutions available for non-Roman writing systems, particularly on the Windows platform, are generally aimed at major, economically significant languages. Because our work is with linguistic minorities, we often encounter situations where these solutions are not adequately flexible.

Graphite is specifically tailored to handle complexities encountered in writing systems derived from complex scripts, such as Arabic and Devanagari. Features include: bidirectionality, contextual glyph selection, ligatures, reordering, stacking diacritics, and sloping baselines. Services are provided for editing data in place, including support for split cursors.

The Graphite system includes a programming language called "Graphite Description Language" (GDL), consisting of rules to perform substitution, reordering, and positioning of glyphs according to the behavior of the writing system. GDL is compiled against a TrueType font file; the output is a font extended with several customized tables. The Graphite engine is a DLL that uses the extended font to perform the specified transformations on Unicode input and produce the rendered output.

Currently Graphite is implemented on the Windows system and is available within a simple text-editor program that has also been developed within SIL. We would like to see the power of the Graphite system more widely available, and can envision various projects to extend it, for example: porting it to other operating systems such as Linux, creating a Java implementation, integrating it with open-source web browsers, and developing an AAT compiler for the GDL language. The development team is interested in pursuing an open-source approach to accomplish these goals.

1 Introduction

One of the challenges encountered in providing access to a body of widely multilingual data via the Internet is ensuring that the user has tools to render it appropriately. Many of the languages of the world being studied by linguists are written using complex scripts for which there is no rendering support provided in the operating systems currently in popular use.

This paper describes the Graphite system, a tool providing extensible rendering for complex writing systems, and discusses how it could be extended to serve as a rendering technology for web-based applications. A more complete description of the Graphite system can be found in a paper presented at the 17th International Unicode Conference. The text of the paper, and complete documentation of the Graphite Description Language, is located on the Graphite web page: www.sil.org/computing/graphite.

2 Overview of Graphite

The Graphite system is a tool being developed by SIL International to provide rendering of complex writing systems on the Windows platform. 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 includes a compiler for a high-level, rule-based programming language, the “Graphite Description Language” (GDL). A GDL program is compiled against a TrueType font, and the results are stored in custom tables within the font, or in a file functioning as an extension to the font.

Graphite also includes an engine that uses the resulting font files for rendering. The engine is a DLL implementing a set of COM objects. Their interfaces include functions for finding line-breaks within a range of text, drawing the text on the screen, performing cursor tracking, and highlighting selections.

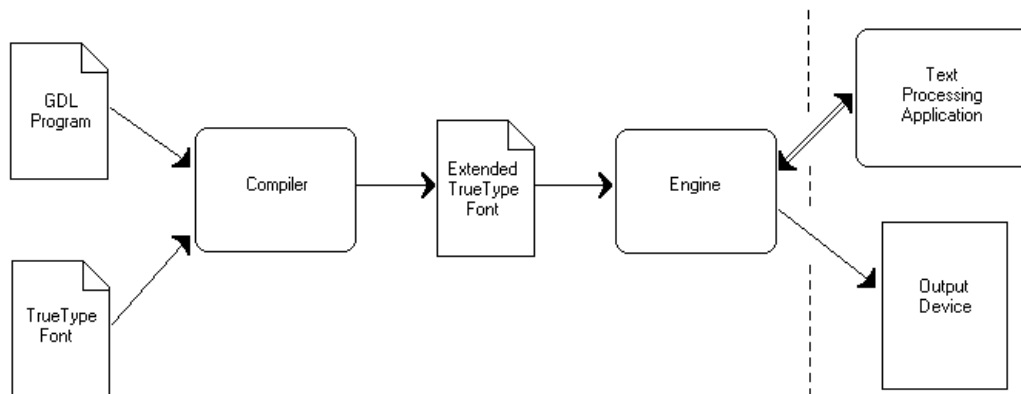


Figure 1: Overview of the Graphite system

3 The need for Graphite

3.1 Orthographic variation in minority writing systems

SIL International is a non-profit organization that performs linguistic research, literacy development, and translation work among ethnic minorities around the world. Increasingly, many of the areas in which we work use scripts that require smart rendering, characterized by extensive contextualization, glyph reordering and splitting, and complex diacritic placement. Smart rendering is especially necessary when dealing with linguistic research, because a naïve underlying representation that would make rendering straightforward would at the same time interfere with linguistic analysis.

Traditionally, much of our work has been with preliterate groups, or those that are literate only in a national or trade language. In cases where there is no written form for the target language, orthography development is a significant part of a field worker's literacy promotion efforts. And because minority languages are usually quite distinct from the national language, it is common to find linguistic phenomena in the minority language that are not present in the national language. Therefore it is often necessary to create a variation of the standard orthography to handle the minority language.

However, the majority of commercially available solutions to the complex rendering problem have been developed to handle national languages, especially the most economically significant ones. Unfortunately, several of the most promising approaches have not been designed in a way that allows them to be extended to handle the variations needed for minority languages. In particular, OpenType and Uniscribe, which comprise the smart rendering technology on the Windows platform, lack the power and extensibility needed to handle the wide variety of writing systems that linguistic researchers encounter in their work.

3.2 Graphite system requirements

Given the need for extensibility described above, we have determined the following features to be requirements of the Graphite system:

- Full Unicode compliance, including support for the Private Use Area and surrogate pairs.
- Left-to-right and right-to-left rendering and bidirectionality.
- Glyph reordering, as needed for Indic scripts.
- Extensive contextualization, including ligatures, start- and end-of-line contextual forms and contextualization across line breaks.
- Stacked diacritics.
- Kerning.
- Roman, hanging, and centered baselines.
- Full editing support, including independent editing of ligature components.
- Split cursor support.

Eventually we hope to provide support for justification and other features useful for DTP applications. Vertical text layout is another possible future addition.

3.3 Limitations

Graphite is intended to render a single line of text in a single writing system. It can also be used to suggest a line break location in a range of text. It does not perform paragraph layout and is not intended to render a mixture of writing systems, except as consecutive calls to the renderer.

This is adequate for our purposes because we have a separate development effort underway that will provide us with two other essential pieces of the puzzle: a string library that incorporates knowledge of language encoding and writing system, and a paragraph layout package that can perform multi-level bidirectional layout. (Graphite also fully supports the Unicode bidirectional algorithm, but because it is intended to be used within the context of a single writing system, extensive bidirectionality may produce some degradation of performance.)

So in the current implementation of Graphite, it is the responsibility of the calling application to break the strings to be rendered into sub-ranges consisting of a single writing system, and to perform the paragraph layout. Graphite assists in the latter by providing line break information, but the application needs to manage the paragraph layout itself.

Graphite does not provide any support for keyboarding or input. However, it does provide support for cursor tracking—resolving mouse clicks to selections in the underlying text.

4 Graphite Description Language

4.1 Overview

4.1.1 Rules

The heart of Graphite’s high-level description language, GDL, is the rules that are used to perform transformations on a stream of glyphs and position them. Those familiar with the field of linguistics might notice a similarity to phonological rules.

As an example, the following is a simple GDL rule that replaces a regular lowercase “i” with a dotless variety when it is followed by a tilde:

```
gLowercaseI > gDotlessLowercaseI / _ gTilde;
```

Notice that the rule has three parts: a *left-hand-side* (lhs), a *right-hand-side* (rhs), and a *context*. The left-hand-side, occurring to the left of the arrow (“>”), shows the contents of the glyph stream before the rule is applied, and the right-hand-side shows the corresponding change to the glyph stream. The context, following the slash, indicates surrounding glyphs that are present in the stream but are unchanged by the rule. The underscore in the context shows the location of the modified letter “i” relative to the diacritic.

We could generalize the rule somewhat to say that this substitution takes place when the “i” is followed by any upper diacritic, not just a tilde:

```
gLowercaseI > gDotlessLowercaseI / _ clsUpperDiacritic;
```

Here “clsUpperDiacritic” indicates a class of glyphs rather than a single glyph. To generalize even further, we can say that any dotted letter is replaced by its dotless equivalent when followed by an upper diacritic:

```
clsDotted > clsDotless / _ clsUpperDiacritic;
```

4.1.2 Glyph classes

A class of glyphs is defined by a statement specifying the members of the class, for instance:

```
clsUpperDiacritic = (gAcute, gGrave, gTilde, gDieresis, gCircum);
clsDotted = (gLowercaseI, gLowercaseJ);
clsDotless = (gDotlessLowerI, gDotlessLowerJ);
```

Notice that the members of the `clsDotted` and `clsDotless` classes correspond by relative position in the class. When the substitution occurs, the member of `clsDotless` that is placed into the output is the one corresponding to the member of `clsDotted` that was present in the input.

Single glyphs are simply single-glyph classes, so our example above must include definitions for them in terms of functions that indicate actual glyphs in the font:

```
gLowercaseI = unicode(0x0069);
gLowercaseJ = unicode(0x006A);
gAcute = unicode(0x0301);
gGrave = unicode(0x0300);
etc.
```

4.1.3 Non-substituting rules

Some rules do not perform substitutions, but simply set attributes. These rules are used particularly to accomplish positioning of glyphs. The following rule has the effect of kerning a capital A to the left when it is preceded by a capital V or W:

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

Notice that this rule does not have an arrow; because no substitution is occurring, the left- and right-hand-sides have been merged. The rule is simply setting the *kern* attribute of the A, specifically kerning the glyph 10 units to the left. (The “m” indicates a value that is scaled in terms of the font design units.) The `kern.x` attribute is an example of a *slot attribute*, so called because it is set for a specific slot in the glyph stream.

4.1.4 Identifying glyphs

Above we saw a few examples of glyph class definitions; these are included in the *glyph table*. There are several different ways of identifying the glyphs to include in a class:

Glyph ID. It is possible to directly specify the ID numbers of the glyphs in the font:

```
glyphid(439)
glyphid(25..43)
```

Unicode ID. A glyph can be identified by its Unicode value in the font’s cmap:

```
unicode(65)
unicode(0x1F08..0x1F0F)
```

Postscript Name. A glyph can be identified by its Postscript name:

```
postscript("Ccedilla")
```

Codepoint. A glyph can be identified by an 8-bit codepoint value that is mapped through a codepage:

```
codepoint("ABC")
codepoint(65..90)
codepoint(0x41, 1252)
```

4.1.5 Glyph attributes

Above we briefly mentioned slot attributes, which are values that are set on individual occurrences within the glyph stream. *Glyph attributes*, on the other hand, are global values that are defined for every occurrence of a given glyph.

Some glyph attributes are defined by the system. Examples of these include *directionality* (indicating the glyph’s level of directionality to be used by the bidi algorithm), and *breakweight* (used when finding line breaks in a range of text).

For instance, it is possible to indicate that a certain class of glyphs function as neutrals with respect to the bidi algorithm:

```
clsNeutrals = glyph-list { directionality = DIR_NEUTRAL }
```

4.1.6 Tables and passes

Graphite is a multi-pass system, with all rules organized into a sequence of one or more passes. Passes, in turn, are organized into a series of tables. The first table establishes appropriate line-break positions; the second performs glyph substitutions (“shaping”), and the third handles positioning.

4.2 Special Rule Behaviors

The Graphite Description Language includes mechanisms to handle various complex script behaviors that are illustrated below. (For details on the GDL implementation, see “Graphite: an Extensible Rendering Engine for Complex Writing Systems.”)

4.2.1 Reordering

In many Indic scripts, the surface glyphs are reordered with respect to the order of the characters in the underlying text, as shown in Figure 2.

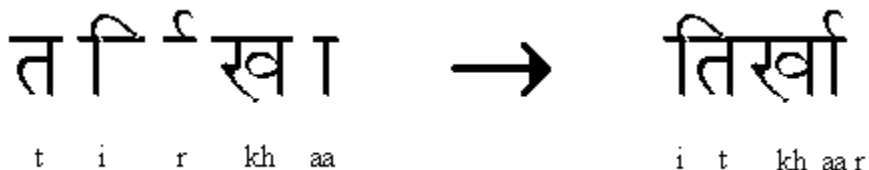


Figure 2: Reordering in Nepali.

4.2.2 Insertions and deletions

Substitution rules can perform insertions and deletions. Thus it is possible to represent a single underlying character with several surface glyphs, or vice versa.

4.2.3 Glyph selection

When performing replacements or insertions, it is possible to select a glyph to place in the output based on a corresponding glyph in the input. Figure 3 shows an example of this in Biblical Hebrew. In each word, the two black glyphs render the same underlying letter of the alphabet—when these consonants appear in word-final (left-most) position, they are replaced by alternate forms.



Figure 3: Regular and word-final letter forms in Biblical Hebrew: (a) kaf, (b) mem, and (c) nun.

4.2.4 Associations

In order to make the surface rendering editable in any meaningful sense, there needs to be a mapping between the surface glyphs and the underlying characters. This mapping can be discontinuous, as shown in Figure 4. In that example, the second character in the underlying text is a vowel that needs to be rendered as a pair of glyphs surrounding the preceding consonant. In the rendered form, the first and third glyphs must both be associated with the second character in the underlying text.

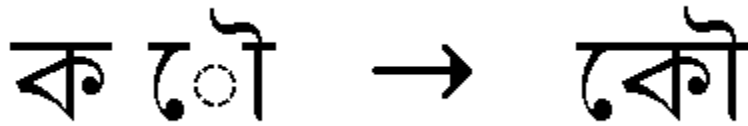


Figure 4: Split vowels in Bengali.

4.2.5 Ligatures

When a single surface glyph is used to represent multiple underlying characters, it is possible to define rectangular areas of the glyph that function as components of the ligature, each mapping to a single underlying character. Examples are shown in Figure 5.

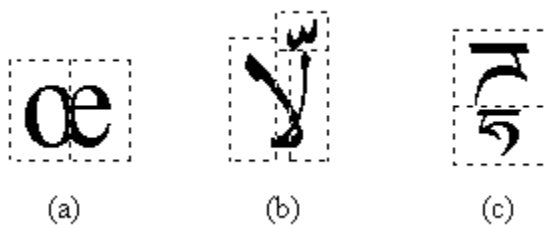


Figure 5: Ligature components in (a) Latin script, (b) Arabic, and (c) Tibetan.

4.2.6 Attachment points

It is possible to specify attachment points on glyphs (for instance, on base characters and diacritics), and write rules that position the glyphs such that their attachment points coincide.

4.2.7 Line breaking

Since Graphite performs rendering and therefore controls the visual space needed by a range of text, one of the system's essential capabilities is to create line breaks in the rendered text. The

application can specify the kind of line break that is preferred, with a worst-case value to be allowed if the preferred type of break is not possible

4.2.8 Line boundary contextualization

Graphite rules can support start- and end-of-line contextualization as well as contextualization across line boundaries.

4.2.9 Bidirectionality

Several Middle Eastern scripts are bidirectional, in the sense that the main flow of text is right-to-left, but numbers in particular are written from left to right. Graphite provides support for these scripts by means of a special pass that implements the Unicode bidirectional algorithm, which accounts for this sort of behavior.

4.3 Features

The Graphite features mechanism provides a way to produce rendering variations for a writing system. For instance, a Greek-based writing system might have two variations, one that renders the final sigma using the alternate form and one that does not. Such a variation would be defined as a feature, and rules can be invoked conditionally based on the value of the feature in the text to be rendered.

4.4 Font interaction

Generally, GDL programs must be written with respect to a single font. It is possible to extract some of the more general writing system behaviors and allow them to be shared among GDL programs intended for various fonts. However, many of the behaviors, especially those relating to positioning and attachment points, must be font specific.

To assist the GDL program and the font in working together, there are several mechanisms in the language to provide access to information in the font

- The `glyphid` function (mentioned above).
- Glyph metrics: bounding box, side bearing, advance width, ascent and descent.
- Point and path functions, making it possible to define attachment points in terms of the actual curves that define the glyph within the font.
- The ability to scale measurements with respect to the design units on which the font is based.

5 TrueType font extensions

The output of the Graphite compiler is an extended TrueType font, identical to the original input font but with several customized tables added. One table holds control data for the Graphite engine, such as the passes and rules, the glyph classes, and special purpose constants. A second table, structured similarly to the standard TrueType “`loca`” table, stores information about glyph attribute values for each glyph in the font. Graphite also makes use of tables to store textual information and the feature data.

6 Using the Graphite engine

6.1 Functions of the Graphite DLL

The Graphite engine is a DLL that implements two COM interfaces—`IRenderEngine` and `ILgSegment`—corresponding to a rendering engine and a range of rendered text in a single writing system. Together these interfaces provide functions to perform the following:

- Create a “segment,” a range of text that can be rendered in a given amount of horizontal space with a given type of line break.
- Render the segment in a specified location on the screen.
- Return the metrics of the segment: height, width, ascent, descent.
- Draw an insertion point (possibly split) or a range selection corresponding to location(s) in the underlying text.
- Return an indication of the insertion point in the underlying text that corresponds to a given screen location.
- Return the visual location of a selection, for the purposes of scrolling.
- Indicate the appropriate adjusted position following the press of an arrow key.

6.2 Responsibilities of the application

It is the responsibility of the calling application, for each string to be rendered, to determine ranges of the string that are in a single language and writing system. For each range, successive calls to Graphite return consecutive “segments” of the range that fit on successive lines, which the application must then arrange into paragraphs. If bidirectionality is occurring, the application must reorder the segments according to the direction of each and the hierarchical structure of the textual data.

The application is also responsible for managing the selections. Graphite can suggest an insertion point in the underlying text corresponding to a mouse click location, and for a given position in the underlying string, draw a (possibly split) insertion point or answer the visual location of the selection so that the application can scroll it into view. Note that if complicated reordering is occurring, a split insertion point may be displayed as part of two different segments (i.e., on two separate lines); it is the responsibility of the application to make the successive calls to the separate segments as necessary.

Graphite provides no support for keyboarding operations; these must be supplied by other processes.

7 Graphite as a web-based rendering technology

Because of Graphite’s extensibility, it has the capacity to render obscure writing systems that are not supported by system software, and thus is well suited to serve as a general-purpose renderer to handle the wide variety of language data that would be encountered in a comprehensive archiving effort. Since Graphite is currently implemented only on the Windows platform, the system would need to be ported and/or extended in several ways to function in this capacity. The following are initial ideas; research is needed to determine which are truly viable options.

7.1 Porting the Graphite engine to other platforms

7.1.1 Java implementation

Because of Java's high degree of cross-platform compatibility, porting Graphite to Java would make it available on a wider range of platforms than is presently the case. Research is needed to determine the best way to fit Graphite into Java's class hierarchy and to convert the operating system API calls.

7.1.2 FreeType integration

FreeType is the renderer that is part of the Linux operating system. Currently there is no smart rendering available on the Linux system, so Graphite has the potential to fill a significant need on that platform.

7.1.3 Macintosh compatibility

The ability to compile a GDL program into AAT tables would create compatibility between Graphite fonts on the Mac and other platforms. However, it is likely that only a subset of GDL would be compatible with AAT.

7.2 Integrating the renderer with a web browser

Possibilities here include Mozilla, the version of the Netscape web browser that is being developed using an open-source approach, and Amaya, an editor/browser under development by the W3C.

7.3 Application integration

Integrating Graphite with projects such as OpenOffice (the open-source development of Star Office) would further extend the range of applications that could be used to view multilingual data that may need customized complex rendering.

7.4 Graphite system extensions

Some extensions may be needed to allow Graphite to fully support all complex scripts. For instance, vertical text has not yet been implemented. Also there are some enhancements that may be needed to handle Nastaliq—the sloping form of Arabic used for writing Urdu. True surrogate-pair support has not been implemented, although this is low priority because the scripts that most need surrogates are not particularly appropriate for implementing in Graphite.

8 Summary

SIL is seriously considering the possibility of pursuing an open-source approach to the on-going development of Graphite in order to extend the power of the system and make it more widely useful within a variety of applications and platforms. We are looking for contacts with an interest in a project similar to the ones listed above, either as an open-source effort or as an independently funded, cooperative project.