Graphite Application Programmer's Guide

Implementing Graphite support in your text-processing application
Version 1.01

Sharon Correll
SIL Non-Roman Script Initiative (NRSI)

Abstract

This document describes what is required to add Graphite support to a text-processing application. It is based on version 2.0 of the Graphite API.

Sections 3 – 6 of this document include code snippets outlining how to use the Graphite API to perform various functions. The code snippets are written in C++ and assume use of the Standard Template Library.

It is suggested that when printing this document you use a color printer if possible, since a number of figures are included where color is a key aspect of the content.

1 Graphite and its capabilities

1.1 The Graphite system

Graphite is a smart-font system with power sufficiently comprehensive to handle the complexities of all known modern writing systems. The term "smart-font" refers to a system in which the knowledge about how to render (display) data is embedded in font tables, rather than coded directly in the application or operating system component. The input to the Graphite engine is character data, possibly marked up with character properties and/or font features, and the output is a set of properly positioned glyphs, along with mappings between the surface glyphs and the underlying characters. The Graphite system provides support for editing mechanisms such as mouse clicks and selection highlighting.

A font is Graphite-enabled by compiling a program written using the Graphite Description Language (GDL) into an ordinary TrueType font. The result is a TrueType font with special tables that are used by the Graphite engine.

1.2 Graphite and GTK

On the Linux system, one way to add Graphite support to an application is to make use of GTK widgets, which are Graphite-enabled by virtue of their use of Pango as the complex-script layout engine. By using GTK widgets, Graphite support comes for free. Some of the more advanced capabilities of Graphite are not available, however, such as split insertion bars, discontiguous range highlighting, and manipulation of ligature components.

Graphite support is not yet incorporated into the Windows version of Pango, so GTK widgets on Windows are not Graphite-enabled at this time.

1.3 Graphite's complex script capabilities

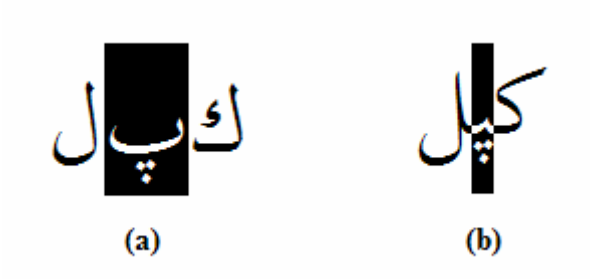
Graphite's complex script capabilities impose a fairly high level of sophistication as a requirement for any application that wants to truly support it. These capabilities are described below. (Also see examples on the Graphite web site: http://scripts.sil.org/CmplxRndExamples.)

1.3.1 Contextual shaping

Contextual shaping means that the selection of glyph for a given character may be affected by the neighboring characters or glyphs. For instance, Arabic script has four forms of each consonant, one each for word-initial, word-medial, word-final, and isolate forms. In Roman text, an "i" with a diacritic above requires a form of the glyph with the dot removed. In Tamil script, the "u" vowels take on different forms depending on which consonant they are associated with.

This means that the application can make no prior assumption about which glyph will be used to render even a fairly standard character, and cannot assume the same character will be rendered with the same glyph in two different situations.

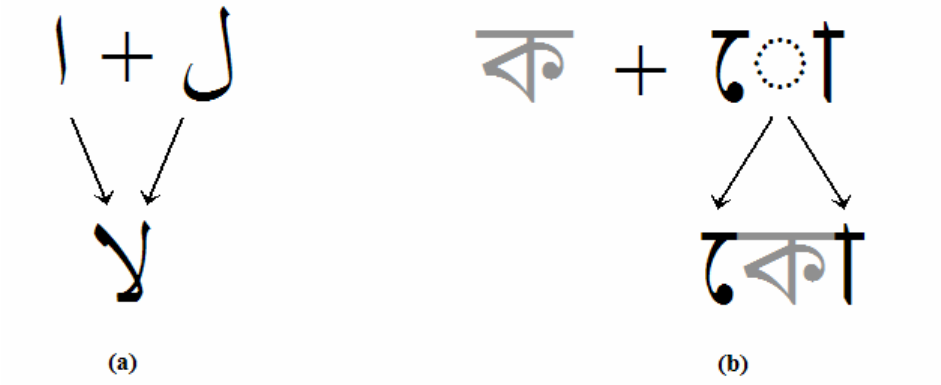
For any strings where contextual shaping can occur, it is necessary to generate the rendering as a single operation; otherwise there will not be enough context present to perform the shaping correctly. For instance, it is not appropriate to render a selected (highlighted) range as a separate operation, because this will break the contextual shaping when a single letter in a word is selected. This problem is shown in Arabic script in figure (a) below. Figure (b) shows the correct behavior.



(a) (b)

1.3.2 Complex character-to-glyph correspondences

In choosing glyphs to represent characters, Graphite permits any combination of one-to-one, one-to-many, many-to-one, and many-to-many mappings. For example, Arabic uses a single glyph to represent the lam-alef character sequence, which is a two-to-one mapping as shown in figure (a) below. Figure (b) shows how Bengali script uses two glyphs to represent the 'o' vowel, which forms a one-to-two mapping.



(a) (b)

This means that the application cannot make any assumptions about the number of glyphs that will be required to render a given sequence of characters.

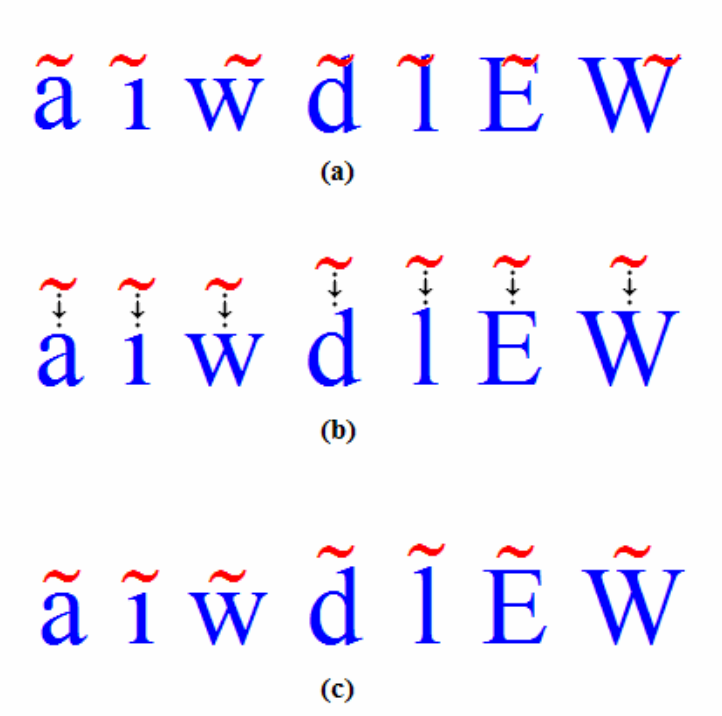
1.3.3 Positioning

Glyphs that are rendered by Graphite can have their positions adjusted either vertically or horizontally so that the rendered output looks significantly different from what would be produced by the dumb font with just basic metrics and kerning tables. A common situation where positioning is needed is to create a stack of diacritics, or to attach diacritics in subtly different locations to various base glyphs. In addition, Graphite can modify the advance width of a glyph to be different from what is specified in the font metrics, for instance, to handling kerning.

Note that this makes it quite possible for the actual height of the rendered glyphs to somewhat exceed the font height indicated by its metrics. It is also possible that glyphs are not laid out in strict left-to-right or right-to-left order, but backtrack and overlap somewhat.

1.3.3.1 Attachment and clusters

Although glyph positions can be modified by simple shift and kern operations, the most powerful and useful way to accomplish complex positioning is by means of attachment points (sometimes called anchor points). That is, two glyphs can be positioned relative to each other so that specified points on the glyphs exactly coincide. If glyphs A and B are attached, one glyph (say, B) is attached to the other (A), so that A forms a "base" for glyph B. Another glyph (say, C) can in turn be attached either to B or to A, and so forth.

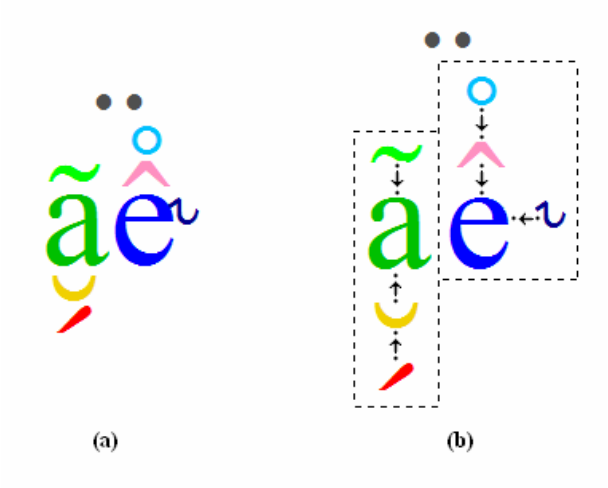


The figures above demonstrate the usefulness of attachments. As shown in figure (a), a simple overstriking diacritic rendered with a "dumb font" looks correct when attached to an average-width lowercase letter such as 'a', but on narrow, wide, tall, and uppercase letters it is either not centered correctly, collides with the upper half of the glyph, or both.[1] Font "smarts," either a shifting mechanism

[1] The solution to this problem when using a dumb font is to use "presentation forms," different versions of each diacritic in the data. For instance, one kind of tilde would be used for centering over narrow characters, another would appear over wide uppercase characters, etc. This produced a correct display but was far from ideal from the point of view of data integrity, and it is incompatible with the encoding principles on which Unicode is based.

or attachment points, are needed to position the diacritic correctly. Figure (b) shows appropriate attachment points, indicated by the small dots and arrows, and figure (c) shows the correctly rendered result.

The attachment mechanism is commonly used to handle stacked diacritics, where the first diacritic is attached to a base glyph, and in turn serves as a "base" for the next diacritic. Upper, lower, overstriking, and side-attaching diacritics can form separate chains off a single base glyph. A base glyph and all its attached glyphs can be thought of as a cluster. Graphite includes the capability to calculate metrics of clusters or sub-clusters as well as individual glyphs, and make use of those calculations in positioning operations.



The figure above shows a set of glyphs that are positioned using both shifting and attachment. Sub-figure (a) shows the rendering using Graphite, while (b) indicates the attachments, shown by the small dots and arrows. Glyph 'a' and its diacritics form a cluster, and 'e' and its diacritics form a separate cluster. The bridging diaeresis is not part of either cluster, but is positioned relative to both of them using both vertical and horizontal shifting.

It is commonly the case that editing (e.g., placing an insertion bar) within the elements of a cluster is undesirable. This behavior can be controlled in the GDL program used to generate the font smarts.[1] For instance, a font may or may not permit an insertion point between the 'a' and the breve below it, or between the circumflex and the ring. The application should make use of the mechanism provided within the Graphite engine interface to query the font with regard to the legality of potential insertion points.

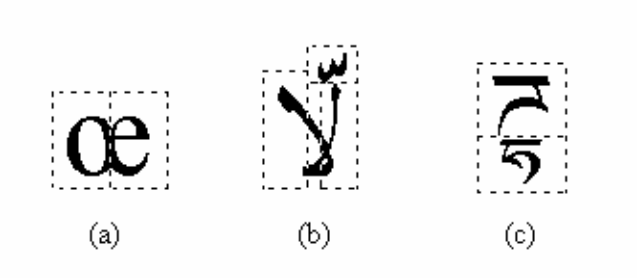
1.3.4 Ligatures

In font technology terms, a ligature is a single glyph that is used to represent two or more characters. For instance, it is common for Roman fonts to use a single glyph to represent the sequences "fi" or "fl." But while in Roman script this is generally needed only to provide typographical finesse, in other scripts this capability is essential to produce correct rendering.

In Graphite, the term "ligature" denotes not only the rendering of a sequence of characters with a single glyph, but also an association between the visual elements of that glyph and the underlying characters. For each ligature in a Graphite font, it is possible to specify a separate rectangular area that corresponds to each of the underlying characters. Selecting that character will cause the corresponding area to be highlighted, and similarly, dragging across just that area will cause the single character to be selected. The

[1] In fact, the same mechanism can be used to allow or disallow insertion between any two characters, whether or not they are attached.

figure below shows examples of ligatures and their components in (a) Roman, (b) Arabic, and (c) Tibetan script.



This capability means that it is possible for selection highlights to involve irregular areas on the screen that do not correspond to complete glyphs. The higlighting and mouse-clicking routines in the Graphite interface handle ligature components automatically, without any particular involvement by the application.

1.3.5 Reordering and splitting

Glyphs may be displayed in a different order from the corresponding underlying characters. This behavior is particularly common in scripts of South and Southeast Asia, where certain vowels are rendered to the right of the preceding consonant. Some vowels are split into several glyphs and rendered on either side or above or below the consonant. Consonants and their associated vowels can be thought of as a cluster, and (as in the case of clusters of attached glyphs), it may or may not be considered appropriate to select in the middle of one or edit the elements of the cluster separately.

Above we mentioned the possibility of one-to-many mappings; split vowels are an example of this.

When reordering and splitting occur, and editing the elements of the clusters is permitted, it is possible to get disjointed highlighting, since a range of characters in the data are not necessarily contiguous in the display. An example of this in Burmese script is shown below.

Split insertion bars are also needed in the case of reordering (see section 1.3.7 below). The Graphite API methods that perform highlighting handle these situations automatically, with a minimal amount of extra complexity in the form of a few extra parameters. (Note that the need for discontiguous highlighting assumes that the application permits only selections that contain a range of *contiguous* characters in the *underlying data*. Using Graphite for more complicated kinds of selections—that is, to handle visually contiguous selections of non-contiguous ranges of characters—is certainly possible, but requires considerably more complexity on the part of the application's text selection mechanism.)

As with other kinds of contextualization, for any strings where reordering can occur, it is necessary to generate the rendering as a single operation; otherwise there will not be enough context present to render correctly.

1.3.5.1 Associations and cursor tracking

As mentioned above, Graphite permits a full range of character-to-glyph mappings: one-to-one, many-to-one, one-to-many, and many-to-many. Graphite keeps track of these mappings in both directions (character-to-glyph and glyph-to-character). The character-to-glyph mappings are used to draw insertion

bars and highlight selection ranges. The glyph-to-character mappings are used to convert screen locations (e.g., from mouse clicks) into a character.

To be more precise, it is the first and last character(s) associated with each glyph that are of particular significance. When the user clicks on a glyph, the click is either on the leading half or the trailing half of the glyph. A click on the leading half of the glyph results in a insertion point just before the first corresponding character, while a click on the trailing half results in an insertion point just after the last character. For this reason, Graphite keeps particular track of the first and last character associated with each glyph. Of course, in most cases, the first and last character are one and the same, but they will be different in the case of ligatures and precomposed glyphs.

In the same way, the first and last glyphs associated with a character are also of particular significance. When an insertion point occurs just before a character, the insertion bar should be placed at the leading edge of the first corresponding glyph. Similarly when an insertion point occurs just after a character, there should be an insertion bar drawn at the trailing edge of the last corresponding glyph. For this reason, Graphite keeps particular record of the first and last glyph associated with a given character. However, there must also be a complete list of *all* corresponding glyphs in order to correctly draw range highlights.

Complicated mappings may also result in the situation where selection of some ranges of characters are undesirable or nonsensical. For instance, if the sequence "oe" is rendered as a ligature (a single glyph), there is no way to visually indicate the distinction between selecting the "oe" sequence as opposed to selecting just the "o" or just the "e."[1] So this kind of selection should be disallowed, and insertion between the two characters should also be disallowed. The engine interface includes a method to indicate these kinds of situations. It is also possible for the GDL programmer to explicitly disallow insertion in certain contexts, and the same method can be used to detect those situations.

The methods for drawing insertion bars and highlighting selections use these internal mappings, and in addition, the interface provides a way for the application to look up these mappings for its own processing.

The potential for reordering has specific implications for cursor tracking and highlighting. For instance, it is not adequate to "measure a string up to character 10," in order to draw an insertion bar after character 10. When reordering is occuring, omitting character 11 may result in a different (that is, shorter) rendering for characters 1 – 10 than what actually occurs.

1.3.6 Right-to-left rendering

A Graphite-enabled font can be programmed to support left-to-right rendering, right-to-left, or both. Right-to-left rendering is needed particularly for Semitic scripts, such as Hebrew and Arabic, and a handful of others. For a right-to-left font, the Graphite engine will generate rendered output with the glyphs in the proper order. However, it is the responsibility of the application to place Graphite's output into a right-to-left (right-wrapping, right-aligned) paragraph as needed.

In a pure right-to-left environment, there are not too many complications involved in using Graphite for rendering. One does need to be aware of the fact that data returned in *logical* order is actually in *right-to-left* order. For instance, the `Segment::glyphs()` method returns glyphs in right-to-left order.

If a paragraph contains a mixture of fonts, text sizes, or properties such as bold or italic, this will result is multiple "segment" objects, where each segment corresponds to a sub-range of rendered text. While each segment is internally right-to-left, it is the responsibility of the application to position these segments out in right-to-left order.

[1] This is assuming the GDL programmer has not used the Graphite ligature component mechanism which exists to allow this very thing!

1.3.6.1 Bidirectionality

Things become more complicated in a multi-script, mixed-direction situation. Note that there are several sources of information about direction that affect different aspects of the layout:

- *Paragraph direction.* This indicates the overall flow of the paragraph. It affects the behavior of the Unicode Bidirectional Algorithm as it is implemented in Graphite, particularly the behavior of white space and other "neutral" characters at the edges of the data. (The application should also use this information to control the behavior of arrow keys: in a right-to-left paragraph, the left arrow moves forward and the right arrow moves backward.)
- *Text direction.* This is the main piece of information that affects the layout of a range of text.
- *Glyph direction.* Each glyph in the font has an associated direction. This information is used in running the Unicode Bidirectional Algorithm. Ideally these directionality codes will match what is defined for the corresponding characters in the Unicode Standard, but the defined values can be overridden in GDL, and they need to be specified for chararacters in the Private Use Area.
- *Font direction.* Each font can indicate which direction or directions it supports. Most Graphite fonts will only support one direction. This information does not actually affect the rendering, it is only available as information to the application, as it may be useful in UI mechanisms. This means that if an application requests rendering of text marked right-to-left using a font that is only intended to support left-to-right, the rendering may (depending on other factors) appear right-to-left!

So various directionality properties of the text and font work together to produce the rendered output. Note that while we said above that text direction is the primary property that affects layout, in reality glyph direction can also be crucial. For instance, if the text is marked right-to-left but all the glyphs in the font are considered left-to-right (a strange situation), the Unicode Bidirectional Algorithm will produce text that mainly appears to be left-to-right.[1]

1.3.6.1.1 *Script/font-internal bidirectionality*

Semitic scripts such as Hebrew and Arabic are inherently bidirectional. While the main flow of the text is right-to-left, numbers are written from left to right. This behavior is handled automatically by the Bidi Algorithm (assuming the font is properly programmed[1]). It means that the order of the surface glyphs is different from that of the underlying characters, which results in a need for split insertion bars (see section 1.2.7) and discontiguous range selections (as mentioned above).

1.3.6.1.2 *Multi-script directionality*

Providing proper layout of mixed-direction text that involves multiple languages, scripts, and/or fonts is the responsibility of the application, not Graphite, and is not a trivial task by any means! The Graphite API does include some hooks to help in the process.

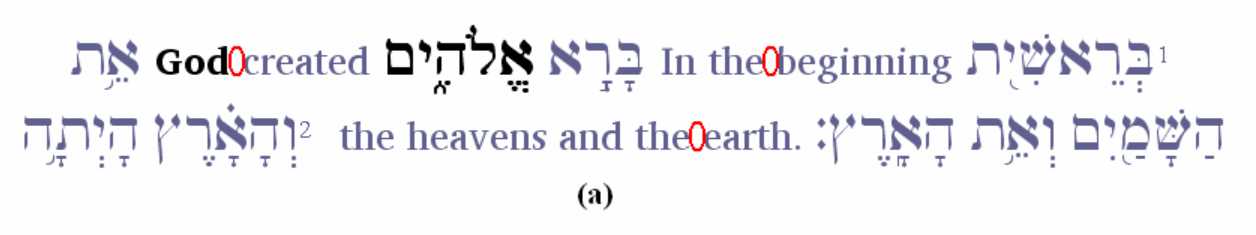
Segment reordering. When mixing text of different languages or fonts, each range will result in a separate segment. The application is responsible for laying the segments out in the correct order. Mainly this involves reversing each sequence of "upstream" segments (those flowing in the opposite direction from the paragraph). Since the upstream segment may in theory include embedded "downstream" segments, the process may need to be recursive. And of course if the overall paragraph direction is right-to-left, the entire sequence of segments on a line will need to be reversed.

The figure below shows an example of mixed-direction text. The primary direction is right-to-left (Hebrew), but there is left-to-right (English) text embedded. Note that segments 1, 3, 4, and 7 are

[1] Note that it is possible for the font programmer to turn off the Bidi Algorithm altogether using GDL. This may be useful for the sake of efficiency, but should only be done purely left-to-right scripts.

downstream, and segments 2, 5, and 6 are upstream. Since the overall paragraph direction is right-to-left, the entire sequence of segments must be put in right-to-left order. However, segments 5 and 6 (two segments because of the of bolding of segment 5) must be reversed with respect to the other segments, since together they represent a sequence of upstream text.

Trailing whitespace. One particularly tricky issue to keep in mind is that of trailing white space. When laying out text of mixed directions, trailing whitespace (the last space on a line) is always treated as "downstream", even if it is part of a script that is "upstream."



The figure above shows an example of mixed direction text, where the overall paragraph direction is right-to-left. (Notice that the paragraph is right aligned but the first line is also indented.) The circles indicate spaces in the upstream (left-to-right English) text that, when the text is wrapped as in sub-figures (b) and (c), occur logically just before the line breaks. That is, these spaces will "trail" at the end of the lines.

When this whitespace is treated as left-to-right, as in figure (b), the result is unsightly and inappropriate double spaces where the text of the two directions meet. This problem is corrected in (c) by treating the trailing whitespace as downstream (right-to-left).

(c)

Note that this way of handling of trailing whitespace is a behavior that is needed for compliance with the Unicode Bidirectional Algorithm.

The way to handle this problem in Graphite is that when creating upstream segments, the application requests the engine to make a separate segment consisting only of trailing whitespace. This way the special trailing whitespace segments can be handled specially—moved to the end of the line—wherever it is necessary. See section 6.3.1 for a discussion.

1.3.7 Split insertion bars

Split insertion bars (sometimes called split cursors) can be useful whenever the sequence of glyphs does not follow a strict horizontal linear layout with respect to the order of the corresponding characters. Examples are shown below and include:

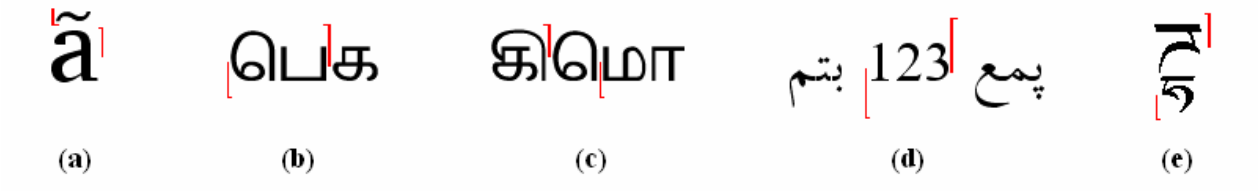
(a) vertical stacking: the insertion point is between the "a" and the tilde.

(b) reordering: the insertion point is after the Tamil consonant (ப) and before the vowel (ெ).

(c) splitting: the insertion point is between the Tamil vowel (ி) and the following consonant (ம)—the intervening glyph is part of a split vowel (ெ ீ) that occurs after the consonant ம.

(d) bidirectionality: the insertion point is after the first space (rightmost—note that the text flows from right to left) and before the "1."

(e) ligature formation: the insertion point is between the two characters which are represented by the top and bottom halves of the Tibetan ligature.



(a) (b) (c) (d) (e)

Graphite provides the means for insertion bars to logically "lean" one direction or another—forward or backward. This is signficant at a style boundary, in that it affects which of the two styles newly inserted text will take on. In the case of split insertion bars, one of the bars is considered "primary" and is drawn thicker, as shown above. Generally this indicates the actual location of the click, or where the insertion point was moved from when using the arrow keys. For instance, in sub-figure (a), clicking on the left side

of the tilde will produce the insertion bars shown; clicking on the right side of the "a" would result in a thicker (primary) to the right of the "a" and a thinner (secondary) one beside the tilde. If the text on either sides of the bars are of different styles, the primary highlight indicates which style will be adopted by newly inserted text.

Currently there is no mechanism in the SegmentPainter interface to turn off split insertions altogether. This could be achieved by subclassing the standard SegmentPainter and overriding the `drawInsertionPoint` method to always draw a single full insertion bar.

1.3.8 PUA support

Graphite was designed to correctly handle data that includes characters from Unicode's Private Use Areas (PUA). The PUA ranges are needed for writing systems that are not supported in Unicode, or are only partially supported.

For characters that are part of the Unicode Standard, there are a wide range of properties defined that may be useful to an application, such as character category (letter, number, punctuation, etc.), case mappings, directionality, etc. PUA characters, however, have no real properties defined and so they simply use default values.

Two characteristics that are of interest to Graphite are directionality and line-breaking. The line-breaking characteristic is derived from whether or not the Unicode Standard considers the character a "separator." GDL allows a font to define these properties as needed for PUA characters (and also to override them for standard characters, although this is not recommended). These properties affect the behavior of the font in rendering and line-breaking, and can also be accessed by the application as needed for its own purposes.

1.3.9 Line breaking

Graphite can serve not only as a rendering engine, but also as a line-breaking engine for scripts whose line-breaking behavior can be described by rules. (Graphite is not adequate to handle scripts that require dictionary look-up for proper line-breaking.)

There are two ways to use Graphite's line-break capabilities. The first is useful in a system where line-breaking and rendering can occur as a single operation. The application asks Graphite to render a paragraph one line at a time, filling the line out as completely as possible and breaking the line in locations considered appropriate by the font. The application specifies ideal and last-resort breakweights, from the following defined values:

- whitespace = 10
- word-level = 15
- intra-word = 20 (e.g., hyphens)
- whole-letter = 30
- clipping = 40 (permitting a break where not even a complete letter will fit in the available space).

The Graphite engine will try to choose the most desirable kind of break, gradually relaxing its constraints until it either finds a range of text that will fit or determines that no text will fit with even a last-resort kind of break.

The second way of using Graphite as a line-breaking engine involves performing line-breaking and rendering as two separate steps. In the line-breaking phase, the application asks Graphite to generate a single segment that renders an entire paragraph. It then queries the segment with regard to possible break points and lengths of various ranges of text in order to make decisions about where the break points will

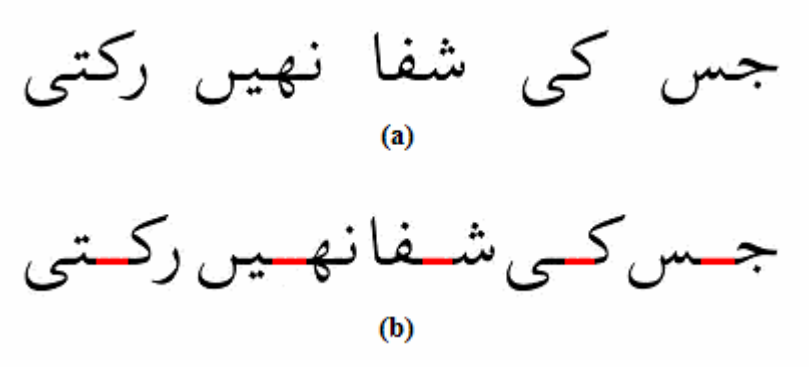
go. Once the paragraph has been broken into lines, the second phase re-renders each line as a new segment.

The distinction between whitespace and word-level breaks is needed for scripts that do not use whitespace between words. In fact, some fonts may define differing levels of word breaks, using, say, 13 to indicate a "very good" word break and 18 to indicate a not-so-good one. The same could be true of hyphenation-level breaks, and the application should be aware of this possibility. If the application wants to insist on, say, a word-level break, it may be appropriate use the value `klbHyphenBreak - 1` rather than `klbWordBreak` to make sure it can handle this kind of granularity.

1.3.10 Justification

Justification involves adjusting the width of a line of text so that it exactly fills the available space. In Roman script, this is generally accomplished by adding or removing space between words and letters. For many non-Roman scripts, however, other strategies are needed. These may include:

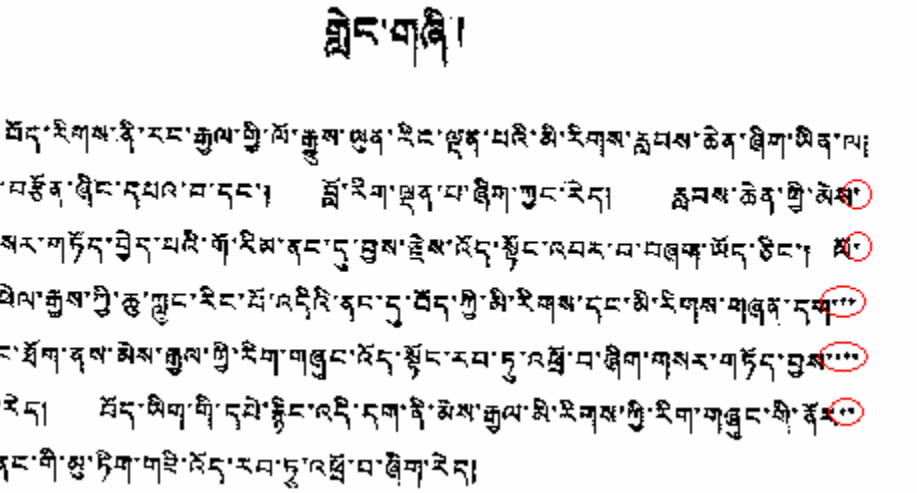
- creation or removal of optional ligatures.
- insertion of kashidas—short horizontal segments used to extend cursive connections, shown in red in figure (b) below. This is a primary strategy in Arabic script.



(a)

(b)

- insertion of line-filler characters, as shown below in Tibetan script.



- substitution of alternate (wider or narrower) glyphs.

Much of the burden for achieving justification falls on the font programmer. A default white-space justification is included by default in all Graphite fonts (unless inhibited). But there are several key things for the application programmer to be aware of.

A main consideration is that justification by means of inserting space between words or letters cannot be seen as a special or even desirable strategy. In Arabic script, for example, insertion of kashidas is preferred, and unnecessary whitespace between words is considered undesirable and aesthetically displeasing. In the Arabic example above, figure (b) is preferred over figure (a).

Justification may be achieved by either stretching a line that is not quite full, or shrinking a line that is slightly over-full. Currently Graphite does not support shrinking using the one-pass line-breaking approach, where each line is filled and rendered in a single operation (see section 6.2.1). Using this approach, the only way to perform justification is by stretching.

Justification using Graphite is handled by means of a call-back object, an instance of a class called Justifier. The Justifier object encapsulates the application's justification algorithm. It is passed to the Graphite engine, and the engine "calls back" to the Justifier at the point in its processing where it needs to make layout decisions that affect justification. This approach allows an application to implement its own justification algorithm. A fairly sophisticated implementation of Justifier is available in the open-source code, however, and is quite adequate for most applications.

Graphite was designed to permit multi-level justification, where the various levels would incorporate different strategies. For instance, the most basic strategy might involve stretching of inter-word space, and a more "radical" strategy, if the basic one proved inadequate for a given range of text, would involve adding or removing optional ligatures. However, this multi-level approach has not yet been implemented.

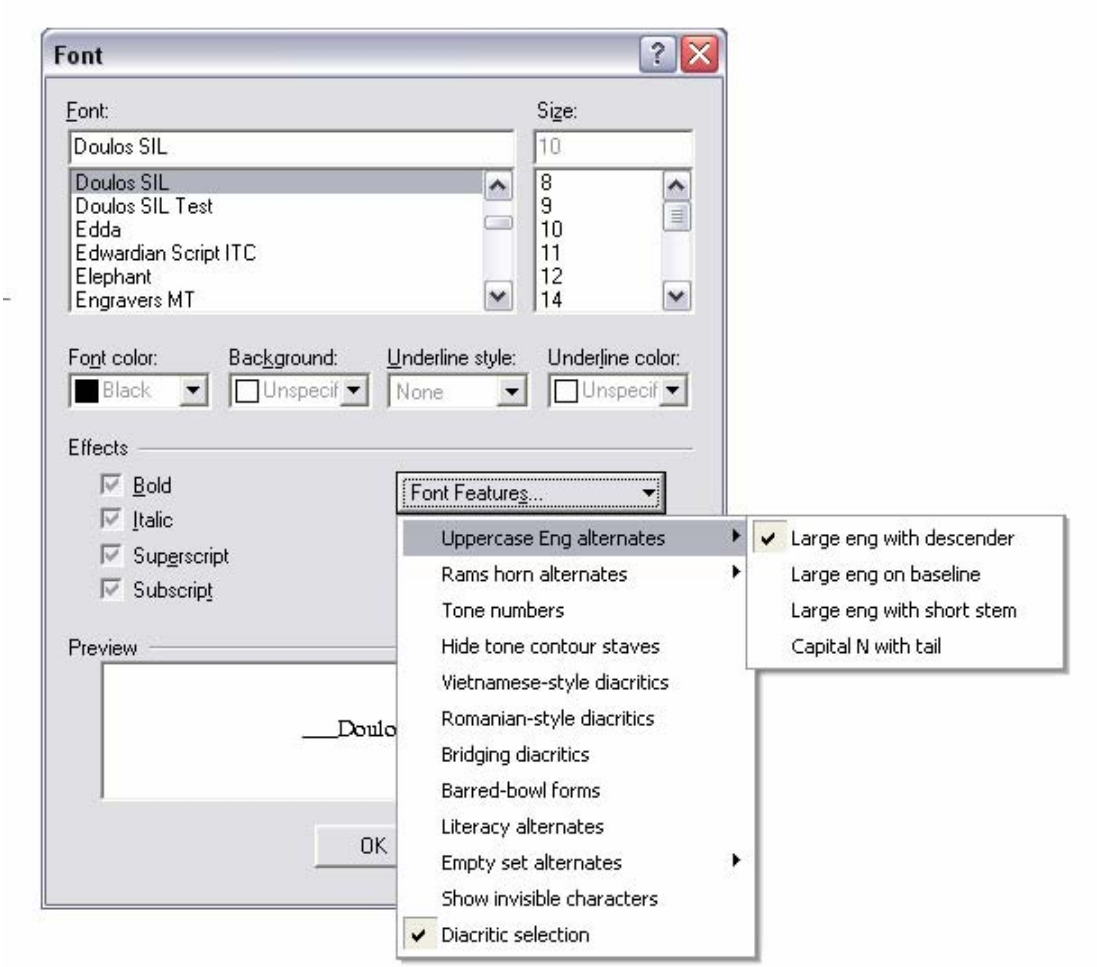
1.4 Font features

Graphite fonts can include variations that are defined by features. Features can be thought of as parameters that can affect the way the rendering behaves. Each font specifies default settings for its features, and if wanted, the application can specify features settings for any given range of text.

Features can be binary- or multi-valued. Examples of possible features include:

- Alternate glyph shapes
- Alternate diacritic positioning
- Showing/hiding invisible characters
- Optional ligatures
- Allowing selection of ligature components
- Allowing selection within glyph clusters

To be fully Graphite-enabled, an application should provide a user interface that allows the user to set features—ideally for any range of text, but at the very least as a default for the application. Graphite's API includes methods to query the font for a list of features and their possible values, including labels appropriate for populating a UI mechanism. The figure below shows a possible menu-based mechanism.

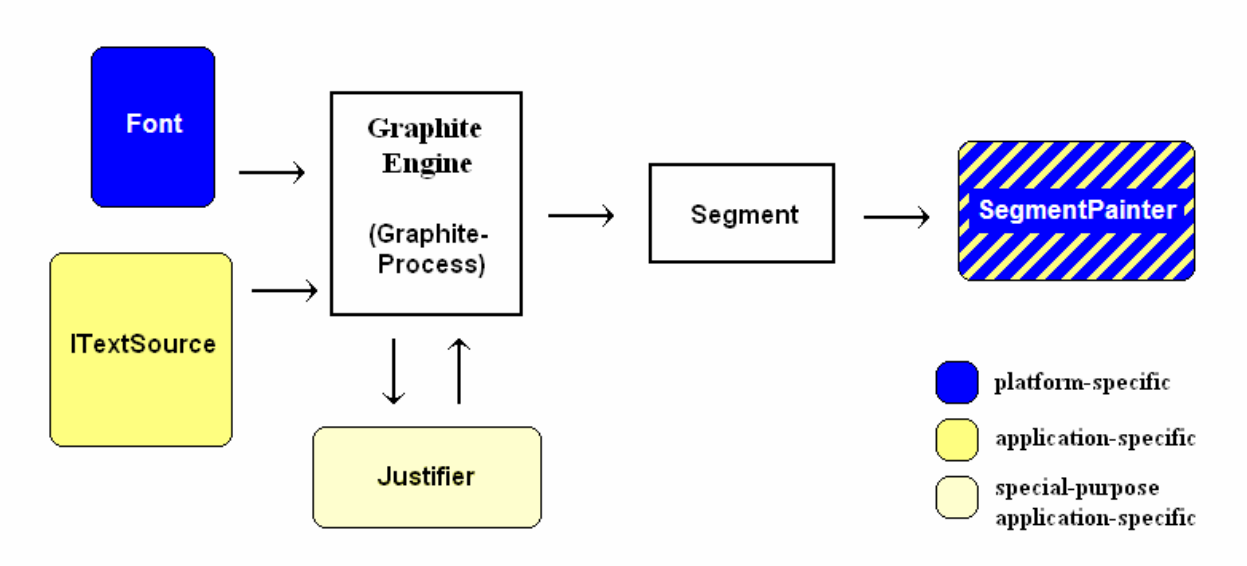


When it is not feasible to extend an application's UI significantly, another possible approach is to use a configuration file to associate feature settings with character style names. Then the application's UI can be used to define and assign character styles, which will have the affect of applying the associated features to the text. Below is an example of a section of a configuration file for the InDesign NRComposer plug-in that associates font features in the Charis SIL font with character styles. The format of each line is `CharacterStyle=feature-ID:feature-value;feature-ID:feature-value;….`

```
[Graphite Features]
Literacy=1032:1
Romanian=1041:1
Vietnamese=1029:1
Vietnamese Literacy=1032:1;1029:1
```

2 The Graphite API class model

Graphite rendering occurs when an application asks the Graphite engine to generate a series of *segments* based on a *text-source* using a *font*. The application may choose to draw the segment or interact with it using a *segment-painter*. A *justifier* can be used to achieve justification, which uses the *GraphiteProcess* interface to communicate with the Graphite engine.



The following classes are part of the Graphite engine interface.

2.1 ITextSource

A *text-source* represents the text to be rendered. It implements methods to access the characters and their style properties. The name of the abstract interface class is `ITextSource`; implementation classes must be a subclass and provide an implementation of all the methods.

A simple stock implementation class is available in the open-source code, called SimpleTextSource. This simple implementation supports only left-to-right text with no features or style changes. It is likely that an application will want to implement its own subclass, in order to handle whatever structure and properties are required for the application's text model.

A subclass of ITextSource called IColorTextSource is also part of the open-source code. It defines an extra method called `getColor` that returns foreground and background color for any character in the text. This will be of interest to applications that want to display colored text. Also note that any application that uses WinSegmentPainter will need to make sure its text-source is a subclass of IColorTextSource, since this is what is expected by WinSegmentPainter.

Summary

What requires an implementation of this class: most applications.

Existing open-source implementations: SimpleTextSrc

Which other classes need to correspond: if a SegmentPainter is being used, it needs to handle any text style properties that are of interest to the application, such as color or underline.

Interface documentation: "Graphite API Version 2: TextSource" (`V2 TextSource Interface.rtf`)

2.2 Font

A *font* object is used to access Graphite tables and TrueType data from a Graphite-enabled font. The tables that are of interest to the font are the following:

- `cmap`
- `head`
- `name`

- `os2`
- `Silf`
- `Feat` (Graphite feature table, not to be confused with `feat` which is the AAT feature table)
- `Gloc`
- `Glat`
- `Sill`

In addition the Font object will need to provide access to font and glyph metrics and details about glyph curves.

Separate implementations of the Font superclass are generally needed for different platforms, such as Windows or Linux, and possibly for different programming frameworks. Implementations of Font that currently exist include:

- WinFont: uses the Windows font handle (`HFONT`) selected into a Windows device context to read the tables and metrics.
- FileFont: reads the font tables from a file with a specified name.
- XftGrFont: uses FreeType to read the font tables and metrics using either an XftFont or FT_Face handle. It is suitable for systems using X with the Xft extension.

A set of open-source utilities are available for reading and interpreting TrueType tables. These can greatly assist in implementing a subclass of Font. These are available in the open-source code, in the files starting with "Tt…" (see `Tt.h`, `TtfUtil.cpp`, etc).

Summary

What requires an implementation of this class: platforms and programming frameworks.

Existing open-source implementations: WinFont, FileFont, XftGrFont

Which other classes need to correspond: some implementations of SegmentPainter may require a specific subclass of Font.

Interface documentation: "Graphite API Version 2: Font and GraphiteProcess"
(`V2 Engine Interface.rtf`)

2.3 Segment

A *segment* represents a range of laid-out text ready to be displayed on an output device. Each segment represents a range of text using a single font and set of style properties that can be displayed on a single line. Some lines will require several segments, others require only one. It is the responsibility of the application to organize segments into a properly laid-out paragraph.

Differences in style properties such as font size, bold, and italic require a separate segment, while styles like color, underline, and strikethrough do not. See the TextSource interface document for a more complete list.

Generating segments constitutes the bulk of the work of the Graphite engine. There are several subclasses of Segment that represent different ways of constructing segments. It is not expected, however, that an application will ever create its own subclass.

A Segment object consists of a list of glyphs with their metrics and positions, as well as mappings between the underlying characters and the glyphs. Other available information includes each glyph's direction, whether it is legal to place an insertion bar before its associated character, whether the glyph is a ligature, and to what extent it is legal to break the line before or after the character. All of this information can be queried from the Segment object.

Summary

What requires an implementation of this class: nothing.

Which other classes need to correspond: none.

Interface documentation: "Graphite API Version 2: Segment, SegmentPainter, and Justifier" (`V2 Segment Interface.rtf`)

2.4 SegmentPainter

A *segment-painter* provides a way to draw or highlight the segment on the output device, and also interact with it in other ways, such as requesting highlight locations, handling arrow key movement, converting mouse clicks to character indices, and querying the legality of insertion points.

The implementation of SegmentPainter that exists in the open-source code includes quite sophisticated approaches to highlighting, interpreting mouse-clicks, handling arrow keys, and most other behaviors. However, one basic operation that is completely missing is the `paint` method to draw the segment. Applications must use a concrete subclass of SegmentPainter, such as WinSegmentPainter or a custom version, that implements this method for the relevant platform or programming framework. It is also appropriate for subclasses to override any of the other methods for which the application wants to supply its own customized behavior.

Some applications may choose to handle these operations without the use of a SegmentPainter object at all. This is often appropriate when the Graphite support is being added in to an existing application or operating system module that uses a very different framework for text layout and editing. In this case, the application interacts directly with the Segment to obtain glyph information and reorganizes it into data structures appropriate for the existing framework. Some capabilities are likely to be lost in this case, such as split insertion bars, manipulation of ligature components, and inserting into the middle of reordered character clusters.

Occasionally it may be convenient to create a segment-painter that is not intended for painting at all, but only for some of the other kinds of segment interactions, such as discerning legitimate insertion points or handling arrow keys. A class called SegmentNonPainter exists and is appropriate for this need.

A SegmentPainter implementation may be based on specific subclasses of the other interface object types. For instance, WinSegmentPainter requires a segment created with a WinFont, since it uses the WinFont's device context for drawing. Also WinSegmentPainter handles color, so it requires an IColorTextSource, a subclass of ItextSource which includes a method to return color properties for the text.

Summary

What requires an implementation of this class:

- platforms and programming frameworks
- applications that want customized drawing and highlighting.

Existing open-source implementation: WinSegmentPainter

Which other classes need to correspond:

- may require a specific kind of Font.
- may require a specific kind of TextSource, e.g., to make style properties available.

Interface documentation: "Graphite API Version 2: Segment, SegmentPainter, and Justifier" (`V2 Segment Interface.rtf`)

2.5 IGrJustifier

The Graphite system allows an application to implement its own justification algorithm. This is accomplished by creating a subclass of the IGrJustifier interface class and passing an instance of this class to the Graphite engine.

This justifier object is a call-back object that is used during the layout process. Midway through the process of creating a Graphite segment, the engine calls on the justifier to make decisions about how much and where to stretch, shrink, and adjust in order to achieve justified output. The justifier interacts with the GraphiteProcess interface to get information about the state of the layout and to communicate back with the engine about its decisions; see the GraphiteProcess interface.

A complete and fairly sophisticated implementation of IGrJustifier is available in the open-source code. Only applications with especially high-end or special-purpose justification requirements will need to implement their own. The best way to do this may be to copy the provided implementation and modify it.

Summary

What requires an implementation of this class: applications requiring very sophisticated or special-purpose high-end justification.

Existing open-source implementations: GrJustifier

Which other classes need to correspond: none

Interface documentation: "Graphite API Version 2: Segment, SegmentPainter, and Justifier" (`V2 Segment Interface.rtf`)

2.5.1 GraphiteProcess

GraphiteProcess provides an interface that allows the Justifier object to communicate with the Graphite engine as it is running. This is needed because the Justifier is a call-back object that is invoked during the process of laying out a segment.

Summary

What requires an implementation of this class: nothing.

Which other classes need to correspond: none.

Interface documentation: "Graphite API Version 2: Font and GraphiteProcess" (`V2 Engine Interface.rtf`)

3 Basic rendering

3.1 Creating a segment

Basic rendering involves creating a text-source based on the text to render, and then asking the Graphite engine to generate a segment or a series of segments.

3.1.1 A simple one-line paragraph

The following code creates a segment that puts the entire text on one line. By default, the RangeSegment constructor includes the entire range of text.

```
// Create a text-source.
MyTextSrc textsrc("Hello World.");
```

```
// Create a Graphite font object, initializing it with the relevant
// style information.
MyGraphiteFont font(...);

LayoutEnvironment layout;  // use all the defaults

// Create the segment.
RangeSegment seg(&font, &text, &layout);
```

Details of how to create the Font object are omitted, because this depends on the implementation of the Font subclass. The following section gives an example of setting up a WinFont.

3.1.1.1 Setting up a WinFont

A WinFont uses the Windows device context for handling font metrics and tables. Below we use the `LOGFONT` structure to select the desired font, size, and style into the device context, and then create a Graphite WinFont using the device context.

This code creates a WinFont that uses a font called “Doulos SIL” at 12 points.

```
HDC hdc = ... // get it from the window

// Set up the Windows font.
HFONT hfont;
LOGFONT lf;
memset(&lf, ‘\0’, sizeof(LOGFONT));
lf.lfCharSet = DEFAULT_CHARSET;
lf.lfHeight = -MulDiv(12, GetDeviceCaps(hdc, LOGPIXELSY, 72));
lf.lfItalic = false; // true for italic
lf.lfWeight = 400;   // 700 for bold
strcpy(lf.lfFaceName, “Doulos SIL”);
hfont = CreateFontIndirect(&lf);

// Select the font into the device context.
HFONT hfontOld = (HFONT)::SelectObject(hdc, hfont);

// Create a Graphite font object.
WinFont font(hdc);
```

3.1.2 Simple line wrapping

To lay out a wrapping paragraph, the simplest strategy is to ask Graphite to create successive lines of the paragraph, fitting as much text as possible on a single line. The LineFillSegment constructor is used for this purpose.

The following is a simple loop that assumes that only one segment is needed per line (which will be the case when all the text uses the same font, size, and styles).

```
size_t charNext = 0;
size_t charEnd = textsrc.getLength();

while (charNext < charEnd)
{
   LayoutEnvironment layout; // use defaults
   Segment * pseg = new LineFillSegment(&font, &textsrc, &layout,
      charNext);

   // ADD HERE: store the segment in a data structure.

   charNext = pseg->stopCharacter();

}
```

3.1.3 Styled and multilingual text wrapping

When the text is complex enough to combine multiple fonts, sizes, or styles, the paragraph may require multiple segments per line. Any style that causes a change in font metrics, such as size, bold, italic, shadow, or outline, will require a separate segment. (Changes in styles such as underline, color, or strikethrough can be handled by a single segment. See the TextSource API documentation for a complete summary of styles that require a segment break.) Also, mixing directions on a line will require multiple segments. If a font supports multiple scripts, such as Roman and Thai, a change in script will not require a change in segment, unless the directions of the scripts differ.

The following code demonstrates the process of laying out a paragraph that may require multiple segments per line.

```
size_t charNext = 0;
size_t charEnd = textsrc.getLength();

size_t lineCount = 0;
bool firstOnLine = true;

float widthLeft = paragraphWidth;

while (charNext < charEnd)
{
   // Get style information from text-source. (These methods are
   // not part of the Graphite API but must be supplied by the
   // application's implementation of ITextSource.)
   std::wstring fontName = textsrc.getFontNameForApp(charNext);
   int fontSize = textsrc.getFontSizeForApp(charNext);
   // ADD: bold, italic, etc.

   // Create an appropriate font object using this style information.
   MyGrFont font(...);

   LayoutEnvironment layout;
   layout.setStartOfLine(firstOnLine);
   // endOfLine = true by default

   if (firstOnLine)
      // For first segment on the line, make sure to include something,
      // even if it requires a very bad break.
      layout.setWorstBreak(klbClipBreak);
   else
      // When there is already something on the line, only add more
      // if we can find a good line break.
      layout.setWorstBreak(klbWordBreak);

   Segment * pseg = new LineFillSegment(&font, &textsrc, &layout,
      charNext, charEnd, widthLeft);

   SegEnd est = pseg->segmentTermination();
   if (est == kestMoreLines || est == kestHardBreak)
   {
      // Line is full; go on to the next.
      lineCount++;
      firstOnLine = true;
      widthLeft = paragraphWidth;
   }
```

```
    else if (est == kestNoMore)
    {
       // Paragraph is completely laid out.
       lineCount++;
    }
    else
    {
       // More segments can be added to this line.
       // Segment just created is no longer at the end of the line;
       // replace it with one with the proper end-of-line flag.
       // This is needed so that the width of the segment will include
       // any trailing white space.
       Segment * psegX = new LineContextSegment(&pseg, firstOnLine, false);
       delete pseg;
       pseg = psegX;
       firstOnLine = false;
       widthLeft -= pseg->advanceWidth();
    }
    // ADD HERE: Store the new segment in a data structure.
    charNext = pseg->stopCharacter();
}
```

(Note that this code sample does not handle the issue of trailing whitespace in mixed-direction text; see sections 1.3.6.1.2 and 6.3.1 for a discussion.)

3.1.4 Backtracking

And it gets worse! When combining multiple segments on a line, the paragraph layout routine may need to rethink its decision about what will fit on the line. This happens when the end of an already-created segment is not a legitimate break point, and it is not possible to fit another complete word on the line.

Consider the sentence "In the beginning, God (אֱלֹהִים) created the heavens and the earth." Note that there will be segment boundaries at the parentheses which are definitely not good locations to break the line. First the paragraph layout routine generates a segment representing the text "In the beginning, God (" and then, believing there is more room on the line, it will try to generate a Hebrew segment for "אֱלֹהִים". But suppose the Hebrew word will not fit on the line (and there is no reasonable way to break that word). In that case, the application needs to backtrack into the previous English segment and find an earlier legitimate break point.

This is handled by the `backtracking` argument in the LineFillSegment constructor. This argument forces the Graphite engine to find a legitimate break point before the `stopChar` argument. If such a break point cannot be found, the returned segment is invalid (as indicated by `kestNothingFit`).

```
startChar = segWithBadBreak->startCharacter();
newStopChar = segWithBadBreak->stopCharacter() – 1;
Segment * segShorter = new LineFillSegment(&font, &textsrc, &layout,
   startChar, newStopChar, widthLeft,
   true); // backtracking
```

```
if (segShorter->segmentTermination() == kestNothingFit)
{
   delete segShorter;
   // Backtrack further if necessary.
}
else
{
   // Replace segWithBadBreak with segShorter in the paragraph.
}
```

Note that it may be necessary to backtrack through several segments before finding a legitimate break point. In our example, if it is not until the point of adding the *closing* parenthesis that the line overflows, backtracking will be necessary through both the Hebrew segment and then the preceding English one.

3.2 Displaying a segment

Once a segment has been created, it can be displayed on an output device by creating a suitable kind of segment-painter that incorporates the segment, and then asking it to paint. Each subclass of SegmentPainter will have its own way of being initialized with the segment, an indication of the output device, and so forth.

```
MySegmentPainter painter(pseg, ...);
painter.paint();
```

3.2.1 WinSegmentPainter

A WinSegmentPainter requires a valid device context to use in the paint operation. This might very well *not* be the same device context as was used in creating the segment originally.

```
WinSegmentPainter painter(pseg, dc);
painter.paint();
```

WinSegmentPainter caches data from the segment in a form that is optimized for drawing on the Windows platform. For this reason it may be useful to cache the segment painter itself and use it for multiple paint operations. There is method in the WinSegmentPainter interface that allows the application to reset the device context as would be needed for this situation.

```
WinSegmentPainter painterCached = new WinSegmentPainter(pseg, dc1);

// Later...
painterCached.setDC(dc2);
painterCached.paint();
painterCached.setDC(0);
```

3.2.2 Coordinate-system transformations

It is possible to use a segment-painter to do coordinate-system transformations. These can involve both the *layout* system (the coordinate system in which the segment was created, based on the resolution of the font used to create it) as well as the *device* coordinates in which it is to be drawn (e.g., pixels).

To position segments at successively lower locations on the pane, set the *origin* of the segment painter to increasing values. To handle scrolling, set the *position* of the segment painter. For a zoom effect, set the *scaling factor*.

The following example shows a series of segments being displayed in a vertical column. The lines are spaced 25 units apart (units being those of the layout system) with 15 units of indentation. There are 6 pixels (device units) of whitespace at the left of the window, and the window is scrolled vertically by 100 pixels. Note that to scroll down, `setPosition` is passed a negative y-value.

```
for (int segIndex = 0; segIndex < segCount; segIndex++)
{
   MySegmentPainter painter(segmentList + segmentIndex, ...);
   painter.setOrigin(segmentIndex * 25, 15); // layout units
   painter.segPosition(6, -100);             // device units
   painter.setScalingFactors(2.0, 2.0);      // zoom to 200%
   painter.paint();
}
```

3.2.3 Obtaining glyph information from the segment

Some applications may choose to not use a SegmentPainter to draw the segment, but read the glyphs and positioning information directly from the segment and do their own drawing. This is accomplished by using the GlyphIterator class to access each glyph in the segment.

```
vector<wchar_t> glyphs;    // 16-bit words
vector<float> origins;
vector<float> advWidths;
vector<float> yOffsets;

std::pair<GlyphIterator, GlyphIterator> iterPair = seg.glyphs();
GlyphIterator glyphsBegin = iterPair.first;
GlyphIterator glyphsEnd = iterPair.second;

GlyphIterator glyphThis = glyphsBegin;
while (glyphThis != glyphsEnd)
{
   GlyphIterator glyphNext = glyphThis;
   ++glyphNext;

   glyphs.pushback((*glyphThis).glyph());

   origins.pushback((*glyphsThis).origin());

   if (glyphNext == glyphsEnd)
      // Last glyph in the segment:
      advWidths.pushback(seg.advanceWidth() – (*glyphThis).origin());
   else
      advWidths.pushback((*glyphNext).origin() – (*glyphThis).origin());

   yOffsets.pushback((*glyphThis).yOffset());

   glyphThis = glyphNext;
}
```

4 Basic editing

The SegmentPainter interface can be used to provide support for editing operations as well as drawing. These methods are defined on SegmentPainter due to the fact that it is most commonly the same code modules that handle editing as well as drawing.

Note that if coordinate-system transformations have been introduced during drawing, it is important to use the same transformations in the painter that is used for these editing operations.

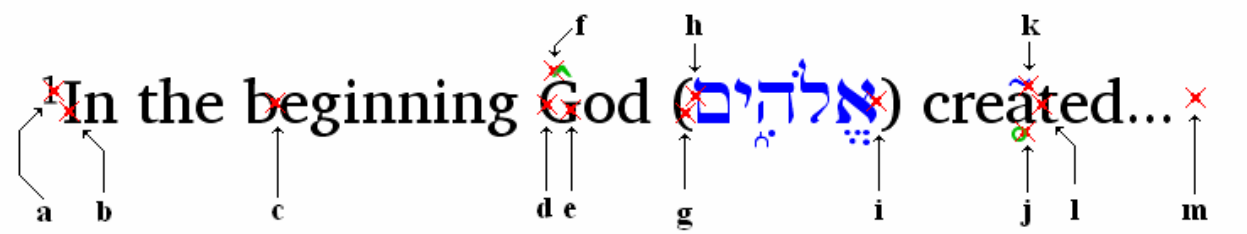
4.1 Mouse clicks

The following code passes a screen location to a Graphite segment in order to return a character index. It is the responsibility of the application to determine the appropriate segment to handle the request.

```
POINT point;
point.x = ...;  // device units where click occurred
point.y = ...;

MySegmentPainter painter(&seg);
int charIndex;
bool leansBackward;
painter.pointToChar(point, &charIndex, &leansBackward);
```

There are two return values. The `charIndex` argument returns the index of character that the click was *before*, where the indices are zero-based. If the click was between characters 3 and 4, `charIndex` will equal 4. If the click was before the first character, `charIndex` will equal 0.

The second return argument, `leansBackward`, indicates whether the resulting insertion bar is more closely associated with the previous text ("leaning backward") or the following text ("leaning forward"). In other words, the character toward which the insertion point logically "leans" is the character whose styles should be applied to newly inserted text. Clicking on the trailing edge (the right edge, in left-to-right text) of a character will cause the insertion to lean backwards; clicking on the leading edge will cause it to lean forward. The figure below shows some sample clicks followed by a description of the results of the standard `pointToChar` method. (Note that the character indices assume decomposed—NFD—data, which means that each diacritic is a separate character, and that there are 9 Hebrew characters.)



a: `charIndex` = 1, `leansBackward` = true; click is between the '1' and the 'I'; newly inserted text will take on superscript style.

b: `charIndex` = 1, `leansBackward` = false; click is between the '1' and the 'I'; newly inserted text will take on regular style.

c: `charIndex` = 9, `leansBackward` = true.

d: `charIndex` = 18, `leansBackward` = false; click is between the space and the 'G'; newly inserted text will take on regular style.

e: `charIndex` = 19, `leansBackward` = true; click is between the 'G' and the diacritic; newly inserted text will take on regular style (black).

f: `charIndex` = 19, `leansBackward` = false; click is between the 'G' and the diacritic; newly inserted text will be green (the color of the circumflex).

g: `charIndex` = 24, `leansBackward` = true; click is between opening parenthesis and Hebrew א; newly inserted text will be English and black.

h: `charIndex` = 33, `leansBackward` = true; click is between Hebew ם and closing parenthesis; newly inserted text will be Hebrew and blue.

i: `charIndex` = 24, `leansBackward` = false; click is between opening parenthesis and Hebrew א; newly inserted text will be Hebrew and blue.

j: `charIndex` = 39, `leansBackward` = true; click is between ring below and tilde above; newly inserted text will be green (the color of the ring).

k: `charIndex` = 40, `leansBackward` = true; click is between tilde and ‘t’; newly inserted text will be blue (the color of the tilde).

l: `charIndex` = 40, `leansBackward` = false; click is between tilde and ‘t’; newly inserted text will be black.

m: `charIndex` = 46, `leansBackward` = true; click is at the end of the segment; newly inserted text will take on regular style.

Remember that “leaning” is always logical, so “leaning backward” means leaning toward the beginning of the text string, which may mean leaning towards the right in right-to-left text.

(Note that the examples involving Hebrew and English are given to illustrate the principle of “leaning” insertions. Since in actuality text in different languages will usually be in different segments, it is the responsibility of the application to determine the value of `leansBackward` and use it appropriately—that is, if the click is on the logically first segment, `leansBackward` would be true, etc.)

4.1.1 Determining the validity of insertion points

A Graphite font has the capacity to specify that insertions are not permitted between specific characters. For instance, the font programmer may prohibit insertions between a base character and its diacritics, or in the middle of clusters of characters where reordering occurs. The `pointToChar` method takes this information into account automatically, and will never return an invalid insertion point.

In addition, a method called `isValidInsertionPoint` is available to help the application ensure that selections created programmatically, for instance as a result of arrow keys, are always valid.

4.2 Drawing insertion points

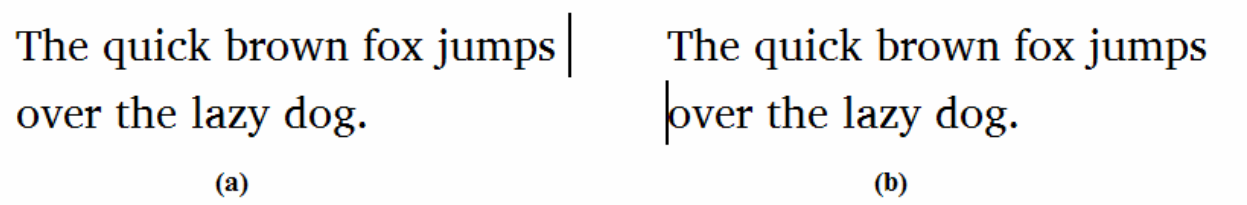
The following code draws an insertion bar at a given character index. It assumes the text all flows in one direction, so only one segment needs to draw. Note that it is the responsibility of the application to determine which segment is responsible for drawing the insertion bar.

```
// Input parameters: segList, charIndex, leansBackward
// If the insertion is at a segment boundary, leansBackward
// determines which segment should draw it.

int lastCharIndex = segments[segList.size()-1]->stopCharacter();
if (charIndex == 0)
   // At the very beginning, leaning backward doesn’t make sense.
   leansBackward = false;
else if (charIndex == lastCharIndex)
   // At the very end, leaning forward doesn’t make sense.
   leansBackward = true;

// Loop through all segments, finding one to draw the insertion.
for (int segIndex = 0; segIndex < segList.size(); segIndex++)
{
```

```
    Segment * pseg = segList[segIndex];
    int start = pseg->startCharacter();
    int stop = pseg->stopCharacter();
    if (start < charIndex || (start == charIndex && !leansBackward))
       && (charIndex < stop || (charIndex == stop && leansBackward)))
    {
       // This segment should draw it.

       MySegmentPainter painter(pseg);
       painter.drawInsertionPoint(charIndex, leansBackward, true, false);
       break; // assumes only one segment needs to draw it
    }
}
```

The second argument (`leansBackward`) is used by the SegmentPainter to determine how to draw split insertion bars (see section 6.1). It has no effect for standard insertion bars. Even in simpler applications that do not support truly multilingual text, split insertions may still be used for ligatures or stacking diacritics, depending on the implementation of the Graphite font.

At the application level, the `leansBackward` flag is also used to determine which segment is responsible for drawing the insertion bar. This becomes signficant when the insertion point occurs at a line boundary, as shown in the figure below.



If `leansBackward` is true, the bar should be drawn at the end of the previous line, as shown in sub-figure (a); if false, it should be drawn as in sub-figure (b). This reflects the fact that generally `leansBackward` would be true if the selection was created by a click or arrow movement at the end of the first line, and false if if it was created by clicking or arrowing in the second line.

The third argument of `drawInsertionPoint` indicates whether the insertion bar should be drawn or erased. In the standard implementation of SegmentPainter, however, this argument is ignored, and the insertion bar is simply inverted (that is, drawn if not present, or erased if it is).

The fourth argument, `forceSplit`, is need to properly handle multilingual text that involves reordering and bidirectionality. See section 6.1 below for a discussion.

4.3 Drawing range selections

The following code draws a range selection, which may extend over multiple segments. As with insertion points, it is the responsibility of the application to determine which segments need to perform the highlighting. The `lineTop` and `lineBottom` arguments are present to allow for the fact that the highlight may need to include some of the “leading”—vertical space between lines of text.

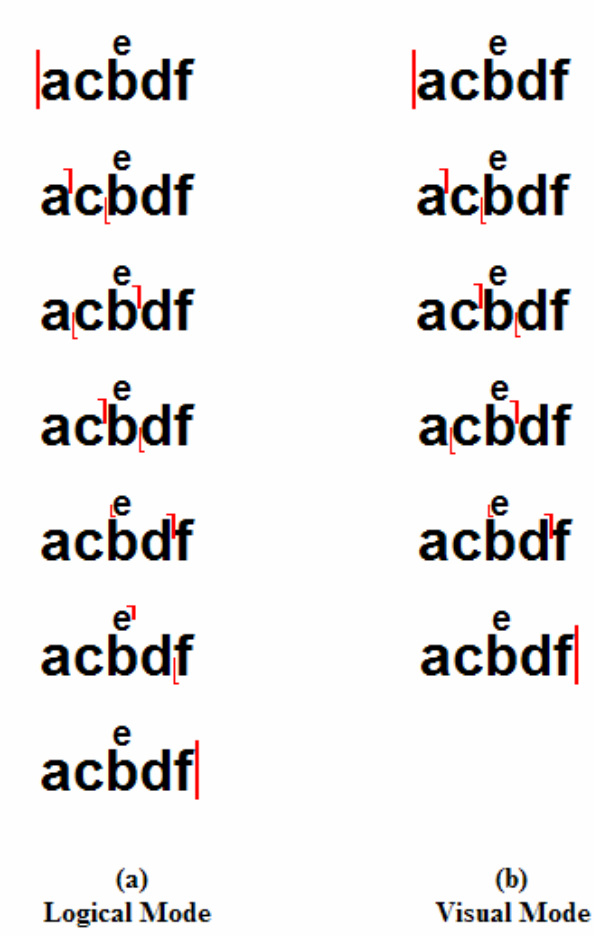
```
// Input parameters: segList, selStart, selStop, lineTop, lineBottom

// Loop through all segments, finding one(s) that need to draw
// at least part of the range.
for (int segIndex = 0; segIndex < segList.size(); segIndex++)
{
```

```
        Segment * pseg = segList[segIndex];
        int start = max(pseg->startCharacter(), selStart);
        int stop = min(pseg->stopCharacter(), selStop);
        if (start < stop)
        {
            // This segment should draw a section of it.

            MySegmentPainter painter(pseg);
            painter.drawSelectionRange(start, stop, lineTop, lineBottom, true);
        }

        // Keep going; another segment may need to handle part of it.
    }
```

There may be a need for discontiguous ranges due to reordering (for example, see the figure in section 1.3.5). This is handled automatically by the standard implementation of SegmentPainter.

4.4 Arrow keys

There are two basic modes of handling horizontal arrow keys: logical and visual. The two modes involve almost identical behavior in simple scripts like Roman, but can be quite different when working with multilingual data and complex scripts. The figure below compares the two modes, showing the movement of the insertion bar (according to the stock implementation of SegmentPainter) on successive presses of the right arrow through text “abcdef” that includes reordering and stacking.



4.4.1 Logical mode

In “logical mode,” the movement of the insertion point as a result of an arrow key reflects the logical ordering of the characters in the text. This can be implemented by simply incrementing or decrementing the character index and drawing the insertion bar at the new position. Range selections are either extended or contracted by incrementing the appropriate end-point of the range.

One complication when working with Graphite fonts is that the application should ensure that the new insertion point is in fact a valid position according to the rules of the Graphite font. It is common for the font programmer to disallow insertions between certain characters, such as base characters and diacritics, or where reordering is occuring.

The routine should also set the “leaning” direction of the insertion point. It should lean toward the character that it just moved over.

The code below shows a logical approach to arrow key movement in left-to-right text.

```
// Input parameters: charIndex, leansBackward, segList, movingLeft

// Figure out which segment knows about the text.
int iseg = 0;
for ( ; iseg < segList.size()-1; iseg++)
{
   if (segList[iseg]->startCharacter <= charIndex
         && charIndex <= segList[iseg]->stopCharacter)
      break;
}

// Turn off the old highlight.
MySegmentPainter painter1(segList[iseg]);
painter1.drawInsertionPoint(charIndex, leansBackward, false, false);

lastChar = segList[segList.size()-1]->stopCharacter();

while (true)
{
   if (movingLeft)
   {
      if (charIndex == 0)
         return;  // can’t go any further
      charIndex--;
      leansBackward = false;
   }
   else
   {
      if (charIndex == lastChar)
         return;  // can’t go any further
      charIndex++;
      leansBackward = true;
   }

   // Figure out which segment knows about the text.
   while (charIndex < segList[iseg]->startCharacter)
      iseg--;
   while (charIndex > segList[iseg]->stopCharacter)
      iseg++;
```

```
      // Ask that segment about the validity of the insertion point.
      MySegmentPainter painter2(segList[iseg]);
      LgIPValidResult ipvr = painter.isValidInsertionPoint(charIndex);
      if (ipvr == kipvrOkay);
         break;
      else if (ipvr == kipvrUnknown)
         // The selection is at a segment boundary; ideally the adjacent
         // segment should be consulted, but for now just assume it's okay.
         break;
      // Otherwise: invalid insertion point; keep going.

   }

   // Turn on the new highlight.
   MySegmentPainter painter3(segList[iseg]);
   painter3.drawInsertionPoint(charIndex, leansBackward, true, false);
   // Also: scroll the selection into view.
```

This routine would require some enhancement to properly handle right-to-left and bidirectional text. To begin with, in paragraphs whose overall direction is right-to-left, the left arrow needs to move forward and the right arrow moves backward. This should be true even in embedded ranges of "upstream text." However, the behavior at the boundary between paragraphs of different directions is particularly tricky: since the arrow keys have opposite meanings, successive presses of a key will bounce back and forth between the end of one paragraph and the beginning of the next! We have not found a good solution to this problem and have concluded that this is just one of the challenges of working with mixed-direction text! If you find a better approach, let us know!

4.4.2 Visual mode

Visual mode involves quite a few more complications when complex scripts are involved, but it can be more intuitive to the naïve user. In this mode, the movement of the insertion bar reflects the physical layout of the glyphs used to render the text. The Graphite SegmentPainter interface provides two methods to help the application implement visual arrow key movement, using its knowledge of the layout of the glyphs: `arrowKeyPosition` and `extendSelectionPosition`. The following sample shows how to move an insertion point in visual mode.

```
   // INPUT: charIndex, leansBackward, movingRight

   // ADD HERE: Figure out which segment the insertion point is in;
   // turn off the old highlight (see 4.4.1).

   // If the insertion point is purely internal to the segment,
   // let the segment just handle it.
   bool inThisSeg = true;
   int newIndex = segment.arrowKeyPosition(charIndex,
      // This argument passes the old value in and gets the new value out:
      &leansBackward,
      movingRight, &inThisSeg);

   while (!inThisSeg)
   {
      // No insertion point is possible within the same segment.
      // Find an adjacent segment.
      if (movingRight)
         segment = // the next segment to the right
      else
         segment = // the next segment to the left
```

```
      if ( /* couldn't find an adjacent segment */ )
         break;

      inThisSeg = false;
      newIndex = segment.arrowKeyPosition(charIndex, &leansBackward,
         movingRight, &inThisSeg;
   }

   if (!inThisSeg)
      // No movement is possible; leave insertion point unchanged.
      newIndex = charIndex;

   // ADD HERE: Turn on the new highlight; scroll selection into view.
```

4.4.3 Vertical arrow keys

There is no special support provided in Graphite to handle up- and down-arrow keys, which have a purely visual function. The basic approach we recommend is to determine the physical location of the insertion point (`positionsOfIP`), increment or decrement the y-coordinate until it coincides with a different line of text, find the appropriate segment in that line, then convert the new location to a character position in the new segment (`pointToChar`).

Ideally, successive presses of the up- and down-arrow keys should remember the horizontal position of the insertion bar as accurately as possible. The `positionsOfIP` method can be used for that. This allows, for example, up-arrowing past a short line at the end of a paragraph to a position in another paragraph that is aligned with the original position.

4.5 Scrolling selections into view

When selections are created programmatically, it is often necessary for the application to automatically scroll them into view. To do this, the `positionsOfIP` and `positionsOfRange` methods can be used to retrieve the screen positions that need to be made visible. As usual, these functions are complicated by the possible need for the split insertion bars and discontiguous ranges that are often needed for complex scripts.

The process for using these methods is similar to drawing insertion points and ranges. But instead of asking the segment painter to draw, rather you request a rectangle indicating the location where the selection would be drawn. The code sample below shows how to achieve this (it assumes no split insertion bars are needed).

```
   // INPUT: segList, selStart, selStop, leansBackwards, maxScrollPosition
   // lineTop, lineBottom.

   int top = maxScrollPosition;
   int bottom = 0;

   // Loop over segments...
   for (int segIndex = 0; segIndex < segList.size(); segIndex++)
   {

      if (selStart == selStop)
      {  // Insertion point.
```

```
        if ( /* this segment is responsible to draw, as in 4.2 */ )
        {
           MySegmentPainter painter(pseg);
           gr::Rect position;
           painter.positionsOfIP(selStart, leansBackward, false,
              &position, NULL);
           // Invalid rectangles are returned as (0,0,0,0).
           if (position.top != 0 || position.bottom != 0
                 || position.left != 0 || position.right != 0)
           {
              // Rectangle is valid.
              top = min(top, position.top);
              bottom = max(bottom, position.bottom);
           }

        }
     else
     {  // Range
        if ( /* this segment is responsible to draw, as in 4.3 */ )
        {
           MySegmentPainter painter(pseg);
           gr::Rect position;
           if (painter.positionsOfRange(selStart, selStop,
                 lineTop, lineBottom, &position))
           {
              // Rectangle is valid.
              top = min(top, position.top);
              bottom = max(bottom, position.bottom);
           }

        }

     }
  }
  // ADD HERE: scroll as needed to make 'top' and 'bottom' visible.
  // If both do not fit, it is generally appropriate to scroll to the
  // bottom of the selection.
```

5 Font features

To fully support Graphite fonts, applications should provide a way for the user to set the font features. At the very least it should be possible to specify the default features for the entire application, but ideally it should possible to also set features on ranges of text. In general, anywhere it is possible to indicate a font, it should be possible to set the font features.

The Graphite API includes methods to query the font for its features. The following shows how to build a menu that offers the user a list of features, using the Windows GUI interface.

```
// INPUT: hmenu, font, currentSettings
// currentSettings is a list of values ordered the same
// as the list generated by getFeatures; a value of kDefault means that
// the feature is unset

const unsigned int eng = 0x0409; // English language

int itemID = kFirstItemID; // each item needs a unique identifier
```

```
std::pair<FeatureIterator, FeatureIterator> featIterPair =
        font.getFeatures();
FeatureIterator featBegin = featIterPair.first;
FeatureIterator featEnd = featIterPair.second;
for (int ifeat=0, FeatureIterator feat = featBegin;
   feat != featEnd;
   ++feat, ++ifeat)
{
   // 1 is a special “language” feature that shouldn’t be included:
   if ((*feat) == 1)
      continue;

   std::pair<FeatureSettingIterator,FeatureSettingIterator> setIterPair =
         font.getFeatureSettings(feat);
   FeatureSettingIterator settingBegin = setIterPair.first;
   FeatureSettingIterator settingEnd = setIterPair.second;

   FeatureSettingIterator default = font.getDefaultFeatureValue(feat);

   // Figure out whether a simple toggle will work for this feature.
   // Here we say this is true if there are two settings,
   // the values are 0 and 1, and the labels are “True” and “False”.

   bool binaryToggle = false;
   if (settingEnd – settingBegin == 2
      && ((*settingBegin) == 0) && ((*settingEnd) == 1))
   {
      FeatureSettingIterator setting1 = settingBegin;
      FeatureSettingIterator setting2 = settingBegin; ++setting2;
      utf16 label1[128];
      utf16 label2[128];
      font.getFeatureSettingLabel(setting1, eng, label1);
      font.getFeatureSettingLabel(setting2, eng, label2);

      if (wcscmp(label1, “False”) == 0 && wcscmp(label2, “True”) == 0)
         binaryToggle = true;
      // Add tests for other strings like “On” and “Off”, “Yes” and “No”,
      // etc. It may be wise to allow for all lowercase strings as well.
   }

   utf16 featureLabel[128];
   font.getFeatureLabel(feat, eng, featureLabel);

   if (binaryToggle)
   {
      bool checked = currentSettings[ifeat] == 1
         || (currentSettings[ifeat] == kDefault && (*default) == 1);

      int checkFlag = (checked) ? MF_CHECKED : MF_UNCHECKED;
      ::AppendMenu(hmenu, MF_STRING | checkFlag, itemID,
         featureLabel);
      itemID++;
   }
```

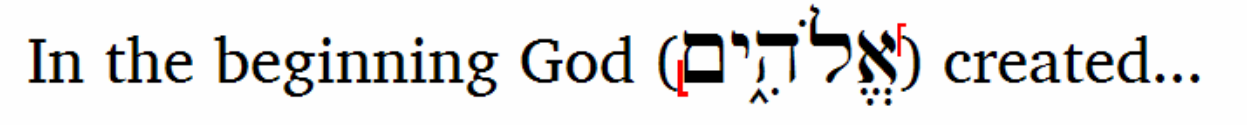
```
    else
    {
       // Create a submenu listing the settings.
       HMENU subMenu = ::CreatePopupMenu();
       ::AppendMenu(hmenu, MF_POPUP, (UINT_PTR)subMenu, featureLabel);
       for (FeatureSettingIterator setting = settingBegin;
          setting != settingEnd;
          ++setting)
       {
          utf16 settingLabel[128];
          font.getFeatureSettingLabel(setting, eng, settingLabel);
          bool checked = (currentSettings[ifeat] == (*setting)
             || (currentSettings[ifeat] == kDefault
                      && (*default) == (*setting)));

          int checkFlag = (checked) ? MF_CHECKED : MF_UNCHECKED;
          ::AppendMenu(hmenu, MF_STRING | checkFlag, itemID,
             settingLabel);
          itemID++;
       }
    }
}
```

6 Advanced issues

6.1 Split insertion bars

Split insertion bars (sometimes called split cursors) can be useful particularly in the case where reordering or other complexities in rendering cause ambiguity as to the exact meaning of the highlight. Refer to section 1.3.7 for an overview of the kinds of situations where split insertion bars can be used.

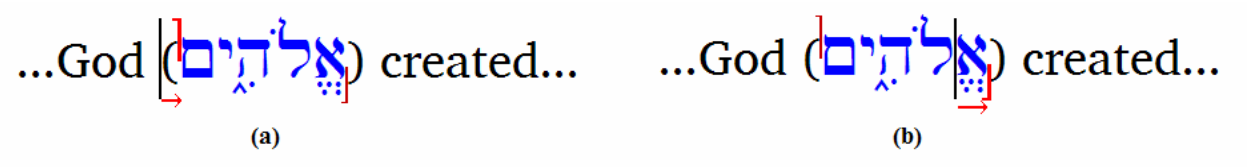
In a split insertion point, one half of the bar is “backward-oriented” and one is “forward-oriented.” The standard implementation of SegmentPainter uses a serif on the insertion bar to indicate the orientation. Also, in an ideal implementation, there is a primary half and a secondary half. In the standard implementation, the primary half is drawn slightly thicker than the secondary half, as shown in the figure below. When typing at a split insertion point, the primary highlight indicates the text from which the language and style of newly inserted text should be adopted. For instance, in the figure below, the “Hebrew side” of insertion point is primary and the “English side” is secondary. In other words, the insertion point “leans backward” toward the Hebrew text, so one would expect newly inserted text to be in Hebrew.



It is the responsibility of the application to record the “leaning” of the insertion point and handle inserted text correctly. The `pointToChar` method returns a recommended direction in which to lean, based on where the given screen point lies relative to the laid-out glyphs. For instance, if the user clicked on a Hebrew character, the insertion point should lean towards the Hebrew.

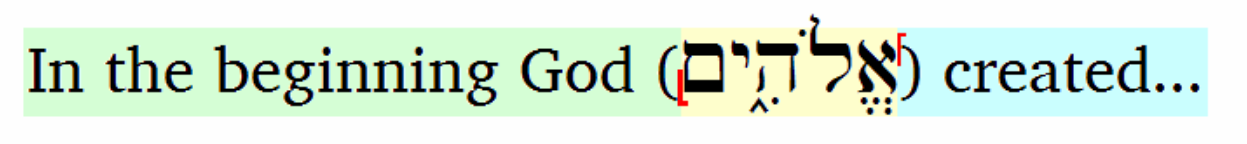
In addition, leaning should be affected by arrow key movement. In general, the insertion point should lean in the direction from which it moved. The application is responsible for keeping track of this information. In sub-figure (a) below, moving from within English text to a direction boundary should result in a

selection that leans toward the English, while (b) shows that a selection that moves from within Hebrew text would then lean toward Hebrew. Note that the resulting character index is the same for both selections; the only difference is the direction in which they lean.

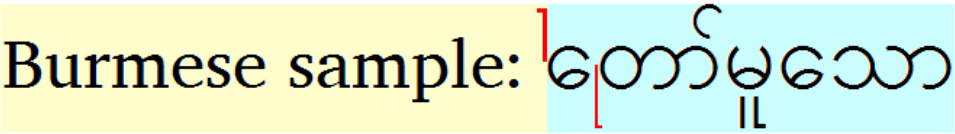


The `drawInsertionPoint` and `positionsOfIP` methods include parameters that support the handling of split insertion bars. The `drawInsertionPoint` method includes an argument called `leansBackward` which indicates how split insertions should be drawn, that is, which half should be considered primary.

The standard implementation will automatically draw split insertion bars where one would expect them—at changes in direction, reordering, stacking locations, etc. (see section 1.3.7). There are also occasions where the application needs to inform Graphite that a split insertion bar is needed. The fourth argument of `drawInsertionPoint`, `forceSplit`, is used to indicate this.



For example, the text in the figure above requires three segments. A split insertion bar is appropriate between the Hebrew word and the following English text, as shown. However, neither segment has adequate information to realize this on its own, so it must be determined by the application by comparing the directions of the two segments, and then passed to `drawInsertionPoint` via the fourth argument.



The second example, above, shows a slightly different situation. In this case the two segments are of the same direction, but the split insertion bar is needed due to reordering in the second segment. Again, there is no way for the English segment to determine that its half of the insertion bar should be split; the application must pass this information in the `drawInsertionPoint` call. The `doBoundaries-Coincide` method can be used to help determine whether split insertions are needed at segment boundaries.

The following code demonstrates how to draw insertion bars when there is the possibility that split forms are needed. The gray code is unchanged from section 4.2.

```
// Input parameters: segList, charIndex, leansBackward, paraLTR

int lastCharIndex = segList[segList.size()-1]->stopCharacter();
if (charIndex == 0)
   // At the very beginning, leaning backward doesn't make sense.
   leansBackward = false;
else if (charIndex == lastCharIndex)
   // At the very end, leaning forward doesn't make sense.
   leansBackward = true;

int lastSegI = segList.size() – 1;
```

```
// Loop through all segments, finding the one(s) that should draw.
for (int segIndex = 0; segIndex < segList.size(); segIndex++)
{
   Segment * pseg = segList[segIndex];
   Segment * psegPrev = (segIndex == 0) ? NULL : segList[segIndex-1];
   Segment * psegNext =
               (segIndex == lastSegI) ? NULL : segList[segIndex+1];
   int start = pseg->startCharacter();
   int stop = pseg->stopCharacter(); // one past the end of the segment
   if (start <= charIndex && charIndex <= stop)
   {
      // This segment should draw it.
      MySegmentPainter painter(pseg);
      bool forceSplit = false;
      if (psegPrev && charIndex == start) // at segment's leading boundary
      {
         MySegmentPainter painterPrev(psegPrev);
         if (!painterPrev.doBoundariesCoincide(true, paraLTR)
            forceSplit = true;
      }
      if (psegNext && charIndex == stop) // at segment's trailing boundary
      {
         MySegmentPainter painterNext(psegNext);
         if (!painterNext.doBoundariesCoincide(false, !paraLTR)
            forceSplit = true;
      }
      painter.drawInsertionPoint(charIndex, leansBackward, true,
               forceSplit);
      // Keep going; possibly other segments need to draw it as well.
   }
}
```

The `positionsOfIP` method also includes an argument to assist with split insertion points. It can return two rectangles, one for the primary highlight and one for the secondary highlight. The code sample below shows how the routine in section 4.5, which scrolls a selection into view, should be extended to handle split selections.

```
// INPUT: segList, selStart, selStop, leansBackwards, maxScrollPosition
// lineTop, lineBottom.
int top = maxScrollPosition;
int bottom = 0;
// Loop over segments...
for (int segIndex = 0; segIndex < segList.size(); segIndex++)
{
   if (selStart == selStop)
   {  // Insertion point.
```

```
        if ( /* this segment is responsible to draw */ )
        {
           MySegmentPainter painter(pseg);
           gr::Rect primaryPos, secondaryPos;
           painter.positionsOfIP(selStart, leansBackward, false,
              &primaryPos, &secondaryPos);
           // Invalid rectangles are returned as (0,0,0,0).
           if (primaryPos.top != 0 || primaryPos.bottom != 0
                 || primaryPos.left != 0 || primaryPos.right != 0)
           {
              // Rectangle is valid.
              top = min(top, primaryPos.top);
              bottom = max(bottom, primaryPos.bottom);
           }

           if (secondaryPos.top != 0 || secondaryPos.bottom != 0
                 || secondaryPos.left != 0 || secondaryPos.right != 0)
           {
              // Rectangle is valid.
              top = min(top, secondaryPos.top);
              bottom = max(bottom, secondaryPos.bottom);
           }

        }
     else
     {  // Range: no change needed from above routine

     }
  }
  // ADD HERE: scroll as needed to make 'top' and 'bottom' visible.
```

6.2 Line breaking

Graphite can serve not only as a rendering engine, but also as a line-breaking engine for scripts whose line-breaking behavior can be described by GDL-style rules.

There are two ways to use Graphite's line-break capabilities. The first approach involves performing line-breaking and layout simultaneously, and the second treats them as two separate operations.

6.2.1 One-pass approach

The first approach is appropriate when the decisions about where to break lines can be made in a fairly straightforward way. Using this approach, the application lays out the text line by line, filling each line as completely as possible. For each line, it calls the Graphite engine with as much text as could possibly be included in a single segment. Graphite fills the line with text, finding a reasonable breakpoint at which to break the line if necessary.

The code samples in section 3.1.2 and 3.1.3 demonstrate the one-pass approach.

6.2.2 Two-pass approach

The two-pass approach is appropriate for higher-quality typesetting, when the application needs to take more complicated criterion into consideration when deciding where to break lines. It is also useful when line-breaking decisions have to be made in a different code module from where the rendering takes place.

In this approach, the application first asks Graphite to render the entire paragraph as one large segment. The application can query this segment with regards to the potential break points, the breakweights at those locations, the length of the segments that would be formed if breaks occurred at specified places, and justification options. This allows the application to do more sophisticated paragraph layout such as balancing the length of lines, avoiding hyphenation on successive lines, shrinking, and so forth. The application uses all this information to chooses the break points. The second pass occurs when the application requests Graphite to generate individual segments based on these breaks.

The following code lays out a single-style paragraph using the two-pass approach. It tries to avoid less-than-ideal (hyphen) breaks on more than one line in a row, unless this would create an extremely short line.

```
// INPUT: textSource, paraWidth

// Create a Graphite font object, initializing it with the relevant
// style information.
MyGraphiteFont font(...);

LayoutEnvironment layout;  // use all the defaults

// Pass 1: Create the segment that includes the entire paragraph.
RangeSegment segPara(&font, &textSource, &layout);

toffset currBreak = 0;

// Allow hyphen-breaks unless the previous line had a hyphen break,
// or a really short line would result. For the first line, allow one.
LineBrk lbNextWorst = klbHyphenBreak;

vector<toffset> breaks;

while (true)
{
   LineBrk lbNext;
   toffset nextBreak = segPara.findNextBreakPoint(currBreak,
         klbWhiteSpace, lbWorst, paraWidth, &lbNext);

   // But if this break would result in a really short segment,
   // allow hyphenation anyway.
   if (lbWorst < klbHyphen)
   {
      float nextWidth = segPara.getRangeWidth(currBreak, nextBreak);
      if (nextWidth < paraWidth * 0.75)
         nextBreak = segPara.findNextBreakPoint(currBreak,
            klbWhiteSpace, klbWordBreak, paraWidth, &lbNext);
   }

   breaks.pushback(nextBreak);

   currBreak = nextBreak;

   // Allow a hyphen on the next line unless we had one on this line.
   lbNextWorst = (lbNext == klbHyphen) ? klbWordBreak : klbHyphenBreak;

}

// Pass 2: Generate actual segments based on these breaks.
```

```
vector<Segment*> segList;
for (int seg = 0; seg < breaks.size() - 1; seg++)
{
   Segment * pseg = new RangeSegment(&font, &textSource, &layout,
         breaks[seg], breaks[seg+1]);
   segList.pushback(pseg);
}
```

The code above assumes that the entire paragraph can be rendered in a single segment; considerable enhancements would be needed to handle combinations of languages, styles, and direction that would require multiple segments.

6.3 Bidirectional issues

6.3.1 Trailing whitespace

Proper handling of trailing whitespace is essential for any application that intends to support mixing of right-to-left and left-to-right text in a single paragraph. The issues involved are discussed in section 1.3.6.1.2.

Segments in a mixed-direction paragraph can be thought of as of two types: “downstream” and “upstream.” Downstream segments flow in the same direction as the paragraph they are a part of; upstream segments flow in the opposite direction. In a left-to-right paragraph, English or Russian text would be “downstream,” Arabic or Hebrew would be “upstream.” In a right-to-left paragraph, the opposite is true: Arabic and Hebrew are downstream, while English and Russian are upstream. However, trailing whitespace must always be treated as downstream.

The approach supported by Graphite is to generate upstream segments in two steps. The first step creates a segment in the normal way but *omits* the trailing whitespace; the second step creates a segment that *only* includes the trailing whitespace. The whitespace segment can then be treated as downstream if necessary (only segments that actually trail at the end of the line should be treated as downstream). Whitespace segments are somewhat special in that they know how to convert themselves easily from upstream to downstream.

The following example shows how the code in section 3.1.3 would be extended to handle trailing whitespace. The gray text is unchanged.

```
size_t charNext = 0;
size_t charEnd = textsrc.getLength();

size_t lineCount = 0;
bool firstOnLine = true;

float widthLeft = paragraphWidth;

LgTrailingWsHandling twsh = ktwshAll; // normal, downstream case

int topDepth = (paraRTL) ? 1 : 0;

while (charNext < charEnd)
{
   // Get style information from text-source.
   std::wstring fontName = textsrc.getFontNameForApp(charNext);
   int fontSize = textsrc.getFontSizeForApp(charNext);
   // ADD: bold, italic, etc.

   // Create an appropriate font object using this style information.
   MyGrFont font(...);
```

```
      LayoutEnvironment layout;
      layout.setStartOfLine(firstOnLine);

      if (firstOnLine)
         layout.setWorstBreak(klbClipBreak);
      else
         layout.setWorstBreak(klbWordBreak);

      int dirDepth = textsrc.getDirectionDepth(charNext);
      if (dirDepth < topDepth)
         dirDepth += 2; // embedded text always has a larger depth
      if (dirDepth == topDepth)
         // Downstream
         twsh = ktwshAll;
      else {
         // Upstream: toggle between visible and whitespace segments.
         if (twsh == ktwshNoWs)
            twsh = ktwshOnlyWs;     // whitespace only
         else
            twsh = ktwshNoWs;       // omit whitespace
      }
      layout.setTrailingWs(twsh);

      Segment * pseg = new LineFillSegment(&font, &textsrc, &layout,
         charNext, charEnd, widthLeft);

      SegEnd est = pseg->segmentTermination();

      // But it's possible we didn't get the kind we were expecting;
      // try the other. This can happen in the case of oddly formatted text.
      if (est == kestNothingFit && twsh != ktwshAll)
      {
         if (twsh == ktwshOnlyWs)
            twsh = ktwshNoWs;
         else
            twsh = ktwshOnlyWs;
         layout.setTrailingWs(twsh);

         pseg = new LineFillSegment(&font, &textsrc, &layout,
            charNext, charEnd, widthLeft);
      }

      SegEnd est = pseg->segmentTermination();

      // etc.

      // ADD HERE: add the segment to a data structure.
      // Also record the segment's direction and embedding depth
      // for later reordering.

   }
```

Once the entire line has been composed, if the final segment on the line is upstream whitespace, it should be changed to downstream. These kind of segments can be identified by the fact that the direction is opposite to the paragraph direction, as well as the fact that their directionality is "weak."

In fact, to be truly compliant with the Unicode Bidi Algorithm, all "weak" segments should have their directionality set to that of the "outermost" of the adjacent segments (that is, the one with the least level of embedding). For segments at the starting or ending edge of the line, this is the directionality of the paragraph. The following code demonstrates this process.

```
for (iseg = 0; iseg < segments.size(); iseg++)
{
   Segment * pseg = segments[iseg];
   bool weak;
   int dirDepth = pseg->directionDepth(&weak);

   if (weak)
   {
      // Weak segment: look at the adjacent strong segments to see
      // what the directionality of this segment should be.

      // First look at the previous segments.
      int depthPrev = topDepth;  // in case we don't find a strong segment
      bool weakPrev;
      for (int iPrev = i-1; iPrev >= 0; iPrev--)
      {
         int depthTmp = segments[iPrev]->directionDepth(&weakPrev);
         if (!weakPrev)
         {
            depthPrev = depthTmp;
            break;   // found a strong segment
         }
      }

      // Do the same for the following segments.
      int depthNext = topDepth;
      bool weakNext;
      for (int iNext = i+1; iNext < segments.size(); iNext++)
      {
         int depthTmp = segments[iNext]->directionDepth(&weakNext);
         if (!weakNext)
         {
            depthNext = depthTmp;
            break;
         }
      }

      // Now change the directionality of the weak segment as needed.

      int newDepth = min(depthPrev, depthNext); // min means least embedded
      if (dirDepth != newDepth)
      {
         Segment * psegNew = Segment::WhiteSpaceSegment(*pseg, newDepth);
         segments[iseg] = psegNew;
         delete pseg;
      }

   }
}
```

6.3.2 Segment reordering

Once all the segments have been assigned the appropriate direction and level of embedding, actual reordering must occur.

Notice that this may need to happen recursively at multiple levels if there are multiple levels of embedding. Consider, for instance, the case of German text embedded within Hebrew text embedded within English; this text would have a maximum embedding depth of 2. The routine below would first

reverse all the Hebrew and German segments, and then re-reverse the German segments with respect to the Hebrew.

```
// ADD HERE: calculate maximum depth of embedding for all the segments
// on the line.

for (int idepth = 1; depth <= maxEmbeddingDepth; depth++)
{
   for (int iseg = 0; iseg < segments.size(); iseg++)
   {
      dirDepth = pseg->directionDepth();

      if (dirDepth > topDepth)
      {
         // We have at least one segment of a different direction.
         // Find the end of the range to reverse.
         int isegLast = iseg;
         while (isegLast < segments.size()-1; isegLast++)
         {
            if (segments[isegLast+1]->directionDepth() > topDepth)
               iLast++;
            else
               break;
         }

         // Reverse the range.
         for (int iLp = 0; iLp < (isegLast–iseg)/2; iLp++)
         {
            Segment * psegTemp = segments[iseg+iLp];
            segments[iseg+iLp] = segments[iLast–iLp];
            segments[iLast–iLp] = psegTemp;
         }
      }
   }

}

// Now that the segments are in physical order, they can be
// positioned simply from left to right.
```

6.4 Justification

6.4.1 Basic justification

Justification is achieved by first generating a segment at its “natural” width, calculating the desired width, and then requesting a new segment of the desired width based on the original segment. The Justifier object, which makes decisions about how to stretch or shrink, is stored in the original segment and then used by the justified segment.

For a single-segment line, this is quite simple.

```
LayoutEnvironment layout;
layout.setJustifier(&justifier);
Segment * psegNatural = RangeSegment(&font, &text, &layout);

Segment * psegJustified =
              Segment::JustifiedSegment(psegNatural, fullWidth);
```

The more common case is that there are several segments on the line that need to be adjusted proportionally.

```
// Calculate the width of the entire line.
int totalNaturalWidth = 0;
for (int iseg = 0; iseg < segments.size(); iseg++)
   totalNaturalWidth += segments[iseg]->advanceWidth();

// To do this nicely, we want to divide the extra width needed among the
// segments in a way that is roughly proportional to the width of each
// segment, but not more than what each segment is capable of.
// This may take several passes to get right, because we're not sure
// at the outset where the stretch can occur and where we're going to
// hit the limits of stretchability.

vector<int> stretches;     // how much each segment should be stretched
stretches.resize(segments.size());     // initialize with zeros

int widthToDistribute = totalWidthDesired – totalNaturalWidth;
while (widthToDistribute > 0)
{
   // Add up the natural width of all the segments that can still
   // be stretched.
   int stretchableWidth = 0;
   for (int iseg = 0; iseg < segments.size(); iseg++)
      if (stretches[iseg] < segments[iseg]->maxStretch(1);
         stretchableWidth += segments[iseg]->advanceWidth();

   int widthToAddThisPass = widthToDistribute;
   for (iseg = 0; iseg < segments.size(); iseg++)
   {
      // Add stretch roughly proportional to the width of this segment.
      int incStretch = (segment[iseg]->advanceWidth()
                                * widthToAddThisPass / stretchableWidth);

      // But don't stretch more than the segment is capable of.
      incStretch = min(incStretch,
            segments[iseg]->maxStretch(1) – stretches[iseg]);
      stretches[iseg] += incStretch;

      widthToDistribute -= incStretch;
   }
}

// Create a new set of adjusted segments.
for (int iseg = 0; iseg < segments.size(); iseg++)
{
   Segment * psegNatural = segments[iseg];
   Segment * psegJust = Segment::JustifiedSegment(psegNatural,
         psegNatural->advanceWidth() + stretches[iseg]);
   segments[iseg] = psegJust;
   delete psegNatural;
}
```

This routine may need to be enhanced slightly in order to avoid rounding errors.

6.4.2 Justification and two-pass line-breaking

As described in section 6.2.2, the two-pass line-breaking approach involves determining all breakpoints up front, and then generating the segments based on those breakpoints. If justification is in effect, it may

be desirable to take its effect into consideration at the point of making decisions about where breaks will go. For instance, you might want to avoid stretching beyond a certain percentage of the natural width of the text.

The following code shows an example of incorporating justification issues into the two-pass line-breaking approach. It ensures that it is indeed possible to stretch a given range of text to achieve justification to the desired paragraph width.

```
// Create a Graphite font object, initializing it with the relevant
// style information.
MyGraphiteFont font(...);

LayoutEnvironment layout;  // use all the defaults
layout.setJustifier(&justifier);

// Create the segment that includes the entire paragraph.
RangeSegment segPara(&font, &textSource, &layout);

toffset currBreak = 0;

vector<toffset> breaks;

while (true)
{
   // Find the next really good (whitespace or word-level) break.
   LineBrk lbNext;
   toffset nextBreak = segPara.findNextBreakPoint(currBreak,
         klbWhiteSpace, klbWordSpace, paraWidth, &lbNext);

   int totalStretch = 0;
   int naturalWidth;

   if (nextBreak < currBreak)
   {
      // No good break found, probably because we're near the end of the
      // paragraph. Just make sure the text can fit on the line.
      naturalWidth = segPara.getRangeWidth(currBreak, textSource.getLength());
      if (naturalWidth <= widthDesired)
         // End of paragraph--we're done.
         break;

      // Otherwise we definitely need a less-desirable break,
      // regardless of potential stretch.
   }

   else
   {
      // See if the text can actually be stretched to the full paragraph
      // width. To do this, loop over the glyphs. For any glyphs that
      // would be part of this range, add up their stretchability.
      naturalWidth = segPara.getRangeWidth(currBreak, nextBreak);
```

```
            std::pair<GlyphIterator, GlyphIterator> allGlyphs = segPara.glyphs();
            GlyphIterator gitBegin = allGlyphs.first;
            GlyphIterator gitEnd = allGlyphs.second;
            for (GlyphIterator git = gitBegin; git != gitEnd; ++git)
            {
               // We simplify here by assuming that it is not possible for
               // a character to be part of more than one line.
               if ((*git).firstCharacter >= currBreak
                     && (*git).lastCharacter <= nextBreak)
               {
                  totalStretch += (*git).maxStretch();
               }
            }
         }

         if (naturalWidth + totalStretch < widthDesired)
         {
            // Can't stretch enough--allow less desirable breaks.
            nextBreak = segPara.findNextBreakPoint(currBreak,
                  klbWhiteSpace, klbHyphenBreak, paraWidth, &lbNext);
         }

         if (nextBreak < currBreak)
            break;  // no reasonable break

         breaks.pushback(nextBreak);

         currBreak = nextBreak;

      }

      // ADD HERE: Generate actual segments based on these breaks.
```

6.4.3 Implementing your own justification algorithm

Justification in Graphite has been architected to allow an application to include its own justification module. While the standard Justifier supplied in the open-source code is quite powerful, it is possible that some applications may want to use a more specific algorithm. This is done by implementing the Justifier class and its main method, `adjustGlyphWidths`.

The `adjustGlyphWidths` method makes decisions about where shrinking and stretching should occur into order to achieve justified output. The basic procedure is as follows:

1. Calculate total amount of adjustment needed—the difference between the natural segment width and the desired width (both of these are passed from the engine).
2. Determine how much adjustment can be made for each glyph, by querying the Graphite engine for the relevant attributes (via `GraphiteProcess::getGlyphAttribute`). Adjustment includes total stretch/shrink amounts, weights, and "steps." Steps indicate the unit-multiples by which some glyphs must be adjusted—for instance, glyphs using a kashida insertion strategy can only be stretched by multiples of the width of the kashida. Highly weighted glyphs should be adjusted more than those with lower weights.
3. Distribute the needed width among the adjustable glyphs, with consideration given to the weights. As well, care must be taken not to violate the step constraints. Rounding errors are a particular concern at this point.
4. Assign stretch or shrink values to all adjusted glyphs, using the `GraphiteProcess::setGlyph-Attribute` method.

It is recommended that you study the stock Justifier code to familiarize yourself with the issues and possible solutions to the justification challenges before implementing your own version. The following are things to keep in mind.

- Shrinking and stretching are quite different processes with possibly very different strategies, so it is important to obtain the appropriate information from the engine—that is, the "shrink" attribute should be used for shrinking and the "stretch" attribute for stretching.
- It is expected that the Justifier will need to perform its calculations in terms of pixels or some output device unit, in order to produce a very accurate result from the point of view of the final layout. Avoiding rounding errors is a concern in any case, but it is a particular challenge in the context of the Graphite Justifier due to two factors. (1) The engine is working with glyph metrics in terms of font units, and additional rounding will likely occur when converting between this coordinate system and that of the Justifier. (2) This tendency becomes even more pronounced in dealing with "step" values. A mechanism is provided to allow the Graphite engine to return the "stretch-in-steps," rather than having the justifier calculate it, which seems to avoid the worst of the rounding problems.
- A glyph with a weight of 2 should ideally stretched twice as much as a glyph with weight 1; similarly, a glyph with weight 5 should be stretched 5 times as much as the glyph with weight 1. However, the overall stretchability for the more-highly-weighted glyph may be actually be less than that of the low-weighted glyph, requiring the low-weighted glyph to take on more than its fair share in a situation where considerable stretching is needed.

6.4.4 Multi-level justification

Multi-level justification is not yet supported in Graphite. It will involve requesting the level of stretch or shrink available at various levels, and choosing the level of justification required.

6.5 Line-boundary contextualization

Graphite supports the ability to define special rendering behaviors for text that occurs at or across line boundaries. This ability can be useful in the following contexts:

- hyphenation
- reordering within a "hyphenated" word (whether or not there is a hyphen-like character)
- high-end typographical features such as start- and end-of-line swashes
- hanging punctuation

To support these features, each segment must be created with an indication of whether or not it occurs at the start of a line or the end of the line, or both. If a segment is created with the end-of-line flag equal to true and then a further segment is added to the line, the previous segment must be changed to be non-end-of-line.

Also, the previous segment (the last segment from the previous line) must be passed to the segment-creation routine, as well as the entire paragraph text-source (not just the sub-range needed for the particular segment). Passing the entire text-source is good practice in any case. The `LayoutEnvironment::setPrevSegment` is the way to supply the previous segment.

A generally useful technique, particularly when using the one-pass approach to line layout (section 6.2.1) is to always first create new segments with end-of-line equal to true. This means that the algorithm can be assured that the newly created segment will fit on the line even if nothing further does.

The `LayoutEnvironment::setStartOfLine` and `setEndOfLine` methods are used to specify these flags. To change the flags of an existing segment, make a copy of the segment using the

`LineContextSegment` constructor (this will make a copy as efficiently as possible). It may be helpful to temporarily keep both versions of the segments available as part of the layout process.

Note that keeping track of end-of-line flags is also necessary in order to properly handle layout of segments that end with whitespace. An end-of-line segment does not include the width of the whitespace in its width (as returned by the `Segment::advanceWidth` method); a non-end-of-line segment does include the width of the whitespace.

The sample code in section 3.1.3 includes the use of `setStartOfLine` and the `LineContextSegment` constructor.

6.6 Generating debugger output

The Graphite engine includes a feature that allows it to output a log of the transduction process it went through in rendering. The format of this log is described in a document called "Transduction Logging in Graphite" (`TransductionLog.rtf`).

The log file can be generated by storing an output file stream in the LayoutEnvironment object that is passed to the segment creation routine.

```
std::ofstream logFile = open("graphiteLog.txt");
LayoutEnvironment layout;
layout.setLoggingStream(logFile);

Segment * pseg = new RenderRangeSegment(&font, &textsrc, &layout);
```

Note that this will overwrite the file every time a new segment is generated. In some situations it may be appropriate to open the file in "append" mode.

7 Appendix: Graphite documentation and resources

- *Graphite web site:* http://graphite.sil.org
- *Examples of complex rendering:* http://scripts.sil.org/CmplxRndExamples
- *Glossary of script-related terminology:* http://scripts.sil.org/Glossary
- *Graphite API Version 2* (`V2 Engine Interface.pdf, V2 Segment Interface.pdf, V2 TextSource Interface.pdf`): documents the classes and methods that are to be used by an application to interact with the Graphite engine. Included with open-source code and available on the web site.
- *Graphite Requirements:* specification that drove the implementation of version 1 of Graphite, annoted with regards to actual implementation. Available for download from the web site.
- *Graphite Table Format (*`GTF_3_0.rtf`*):* format of Graphite font tables. Included with open-source code.

For programming and debugging Graphite fonts:

- *GDL* (`GDL.pdf`): definitive documentation of the Graphite Description Language for programming Graphite fonts. Does not include justification features. Available for download from the web site.
- *GDL Tutorial:* introduction to GDL programming, with exercises and solutions. Available for download from the web site. Available for download from the web site.
- *Compiler Debug Files* (`Compiler Debug Files.rtf`): documents the files that are output by compiler that can be used in debugging a Graphite font. Supplied with the Graphite compiler.

- *Stack Machine Commands* (`Stack Machine Commands.rtf`): describes the stack machine commands that implement the Graphite rules. Useful for interpreting the debugger output from the compiler. Supplied with the Graphite compiler.
- *Transduction Logging in Graphite* (`Transduction Log.rtf`): documents the output of the log file generated by the Graphite engine. Supplied with the Graphite compiler.

Graphite engine source code documentation—contact the development team to obtain these:

- *Graphite Overview* (`GraphiteOverview.pdf`): overview of the Graphite engine. Some of it is obsolete, based on version 1.0 of the API.
- *FSM* (`FSM.pdf`): describes the finite state machines that are used for input recognition
- *Graphite Slot Streams* (`SlotStreams.pdf`): describes the slot-stream mechanism that is used for Graphite processing.
- *Stack Machine Commands* (`Stack Machine Commands.rtf`): describes the stack machine commands that implement the Graphite rules.
- *WinRend Multi-pass Data Transform Engine* (`WR Data Transform Engine.pdf`): early design document describing some of the complexities needed to be handled by the engine.
- *WinRend Finite State Machines* (`WR FSM Design.pdf`): early design document describing the purpose and construction of the finite state tables.

8 Revision history

1.00 18 July 2006. Version 1 released.

1.01 Slight change to API class diagram.

9 File Name

GraphiteAppProgGuide.pdf