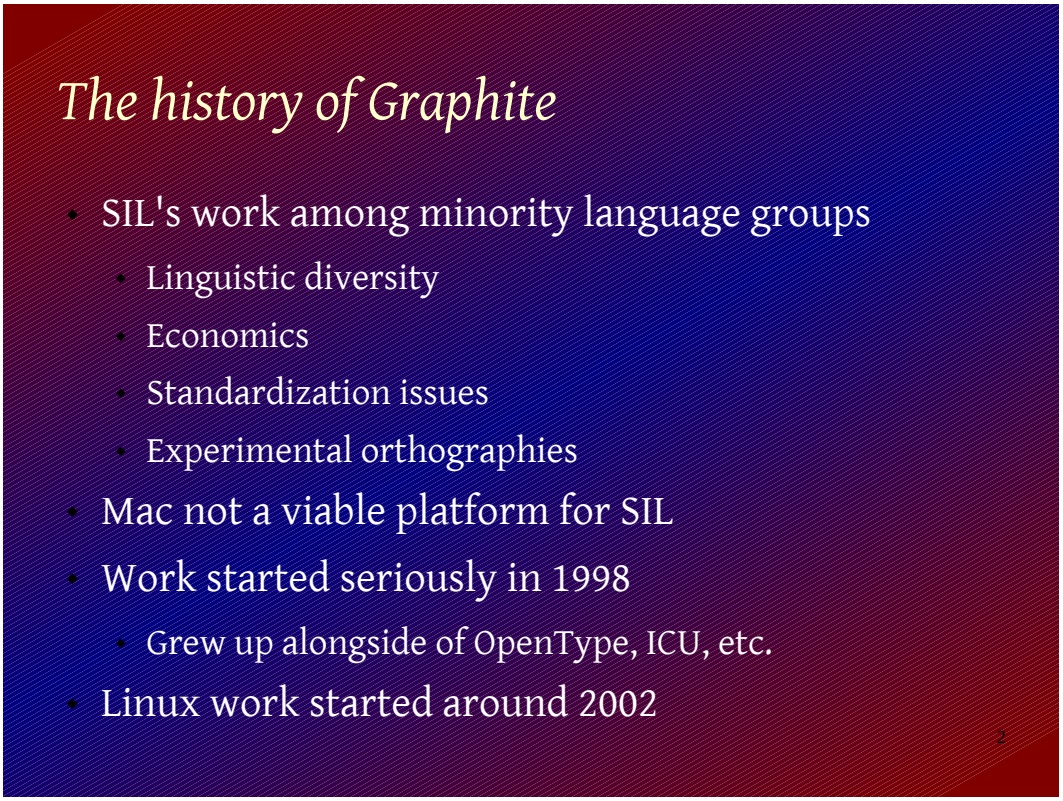
Graphite

Smart-font technology
to bridge the digital divide

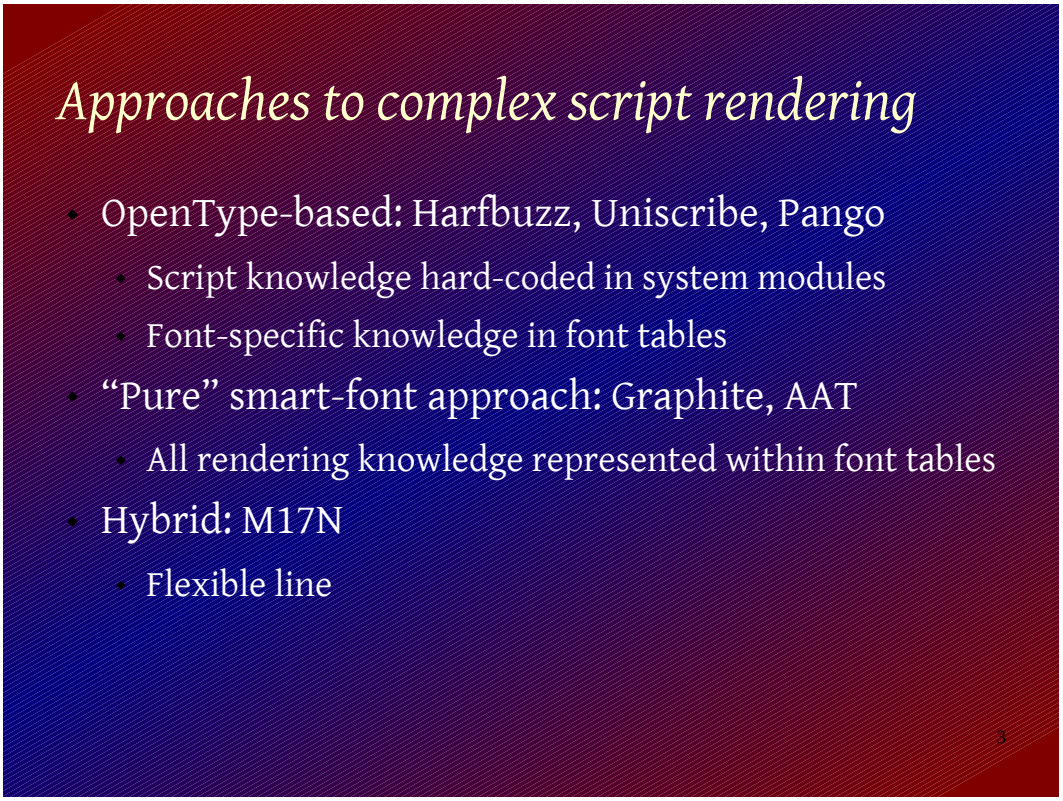
by Sharon Correll,
SIL International



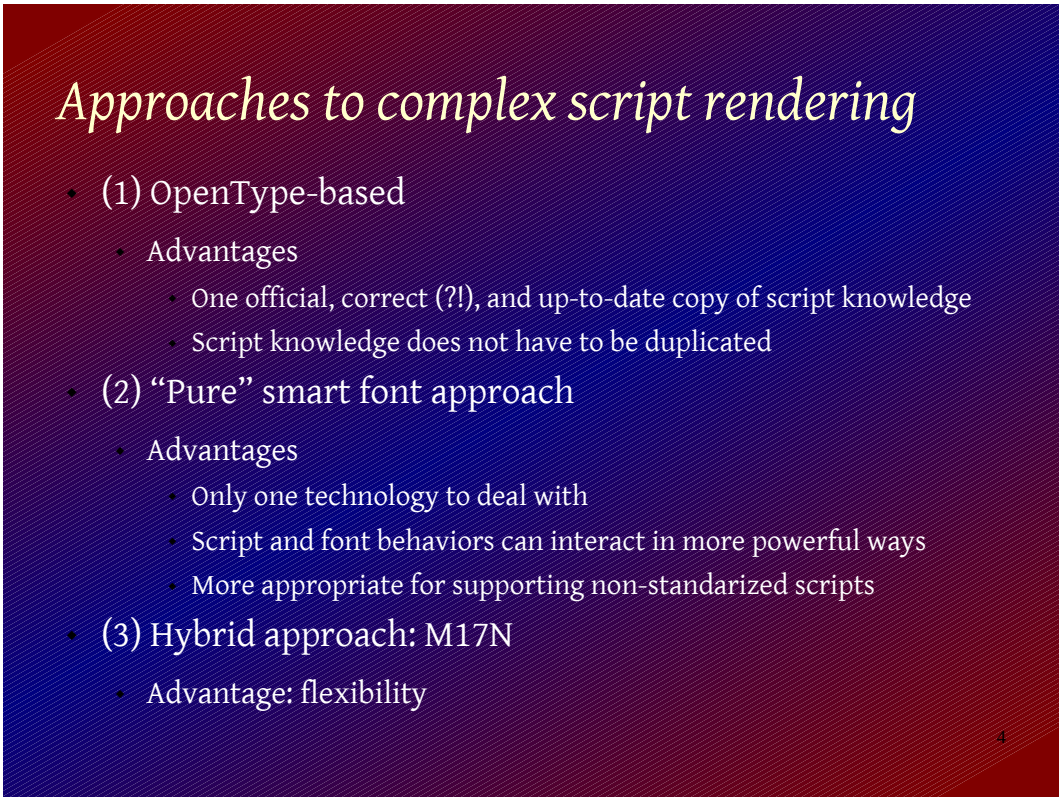
The motivation for Graphite lies in SIL's work among linguistic minority groups. Because of the linguistic diversity that exists in many countries and regions, there is often a need for a minority group to make adaptions in order to use the national script for their language. The minority language may differ profoundly from the national language, requiring unique characters, character combinations, diacritics, tone markings, etc. For reasons of economics or simple lack of knowledge, these needs are often not supported well by software developed by the major industry players. Another obstacle is the fact that the characters used by minority languages are not part of international standards such as Unicode.

In the early 1990s, SIL workers with complex script needs often used the GX technology on the Macintosh. However, there were many places around the world where the Mac did not prove to be a viable platform. Graphite was originally developed to provide extensible complex script support on the Windows platfom.

Work on Graphite started seriously around 1998, and so it “grew up” alongside of other technologies such as OpenType, ICU, etc. Work on Linux began in 2002, as it began to be clear that Linux's open-source model was a good fit with SIL's work and philosophy.

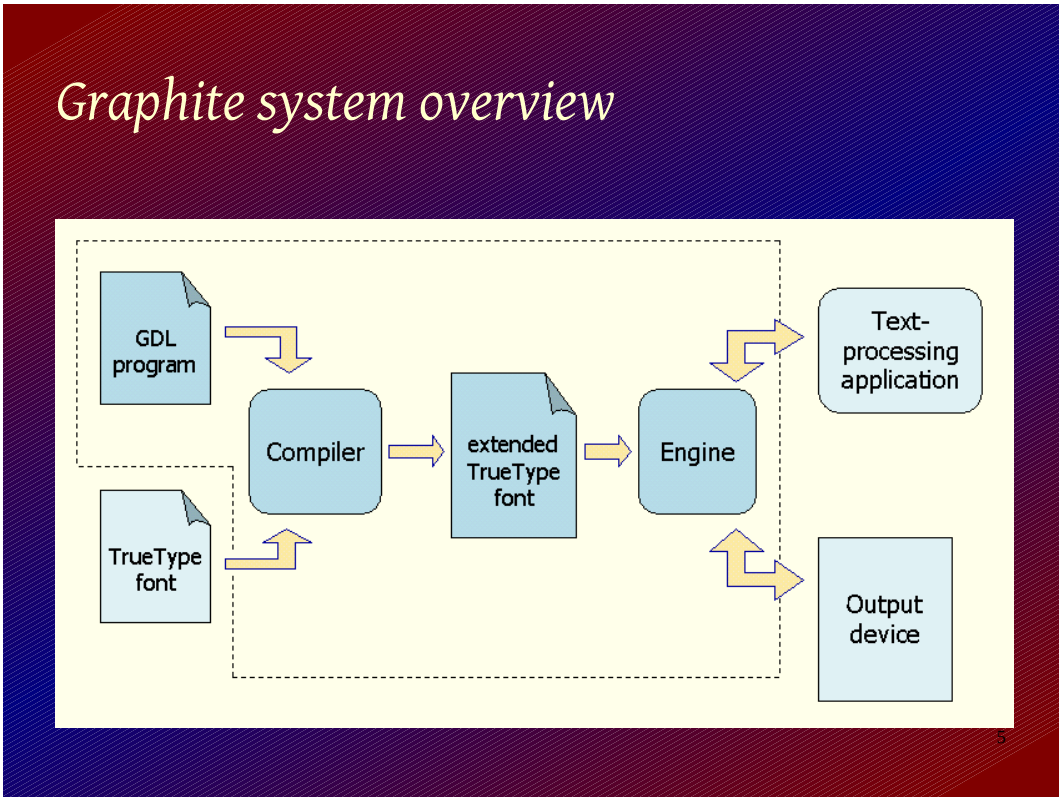


There are two main approaches to complex-script rendering. Technologies that are based on OpenType, such as Harfbuzz, Uniscribe, and Pango, separate script knoweldge from font knowledge, with the former incorporated into standard software modules and the latter into font tables. (In fact, OpenType itself does not have the power to handle all script-related behaviors, such as reordering.) Graphite and AAT, on the other hand, are both “pure” smart-font technologies, in that all the knowledge governing rendering, both script-related and font-specific, is represented by font tables. M17N represents a third or hybrid approach, creating a flexible line between what is put in software modules and what goes in the font.



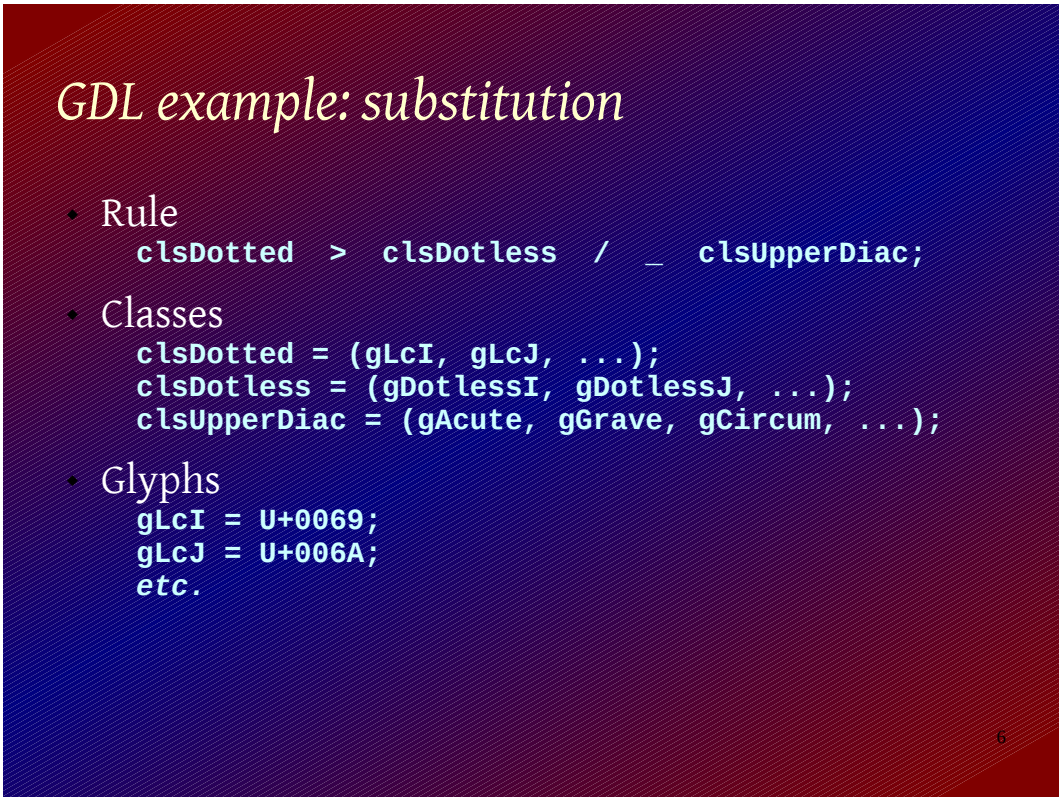
Each of the approaches to script support has advantages and disadvantages. The main advantage of the OpenType model is that general script knowledge is implemented once and is made available by means of a single system module. This results in relatively consistent behavior regardless of the font.

One of the advantages of the "pure" smart font approach is that implementing support for a script involves dealing with a single technology. Also, because the script- and font-knowledge are handled by the same process, the two kinds of information can interact in more complex ways, resulting in a more powerful system. Often minority languages need support for a non-standardized set of characters or some experimental behaviors, and while it would generally not be desirable or practical to implement such support in standard system module, it is quite feasible to do so by means of special-purpose font.



A Graphite font is created by writing a GDL program to coordinate with a font. GDL (Graphite Description Language) is a programming language that uses transformation rules to specify the font's behavior. (Utilities are available to help auto-generate some of the GDL code based on information from the font.) The program and the font are compiled together to create a font with extra Graphite tables. The font is used by the Graphite engine, which serves as the back-end of a text-processing application. The engine provides support for drawing on an output device.

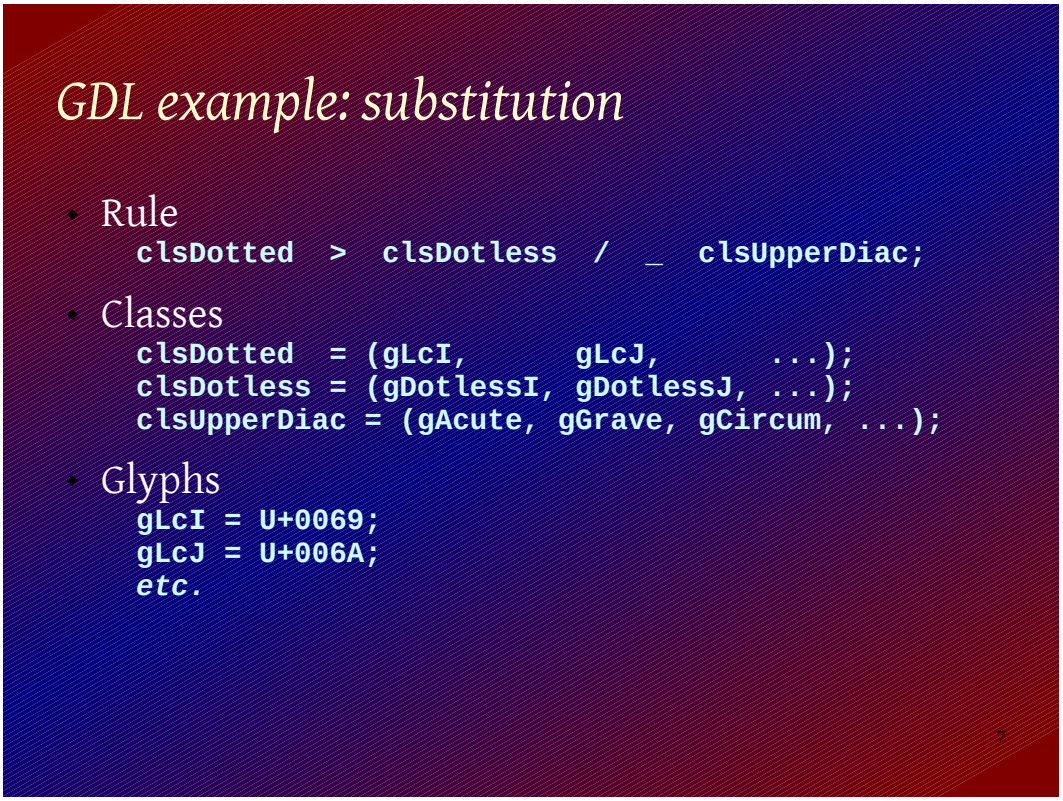
Currently Graphite only handles TrueType fonts, but it may be possible to extend it to handle other formats.



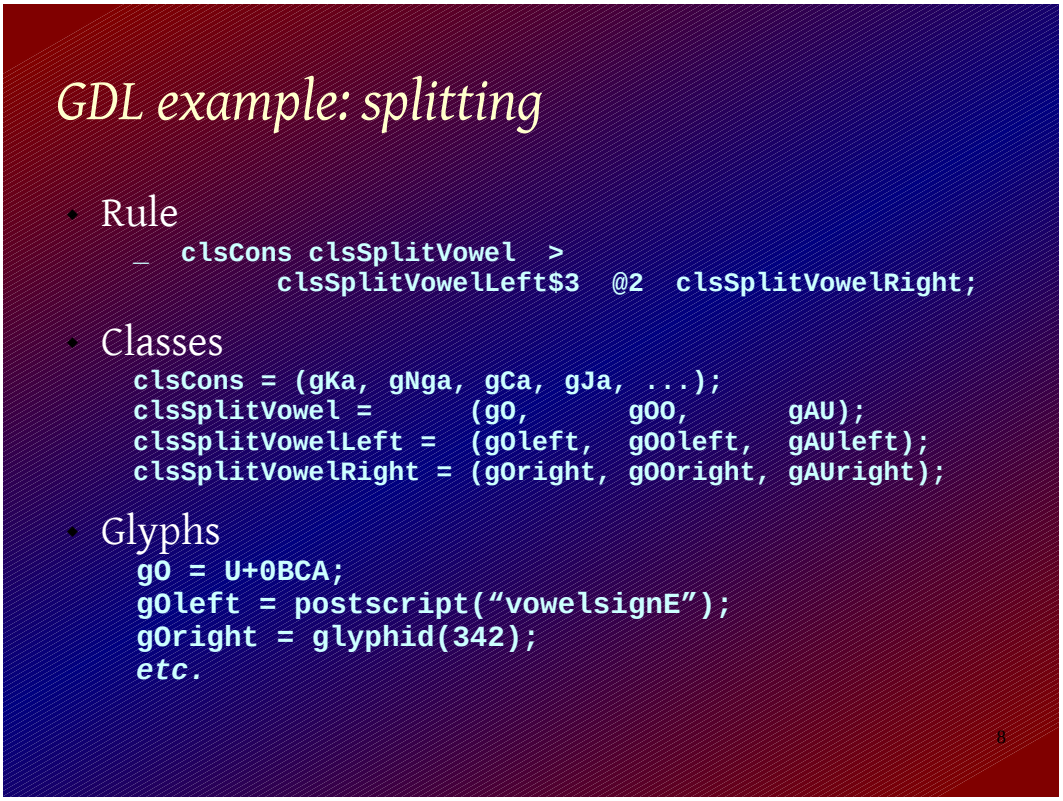
GDL (Graphite Description Language) is a programming language that uses transformation rules to describe font behavior. The rules have some similarities to phonological rules seen in linguistic study.

The slide above shows an example of a GDL substitution rule. The purpose of the rule is to replace a dotted letter with its dotless version in the context where it is followed by an upper diacritic. (The part of the rule that follows the slash indicates the context, and the underscore in the context represents the location of the dotted base character to the diacritic.)

The items used in the rules must also be defined in GDL. The `clsDotted` class contains dotted glyphs such as lower case i and j. The `clsDotless` class contains the dotless equivalents. The individual glyphs above are defined in terms of Unicode characters, but they can also be defined using glyph ID numbers or Postscript names from the font.



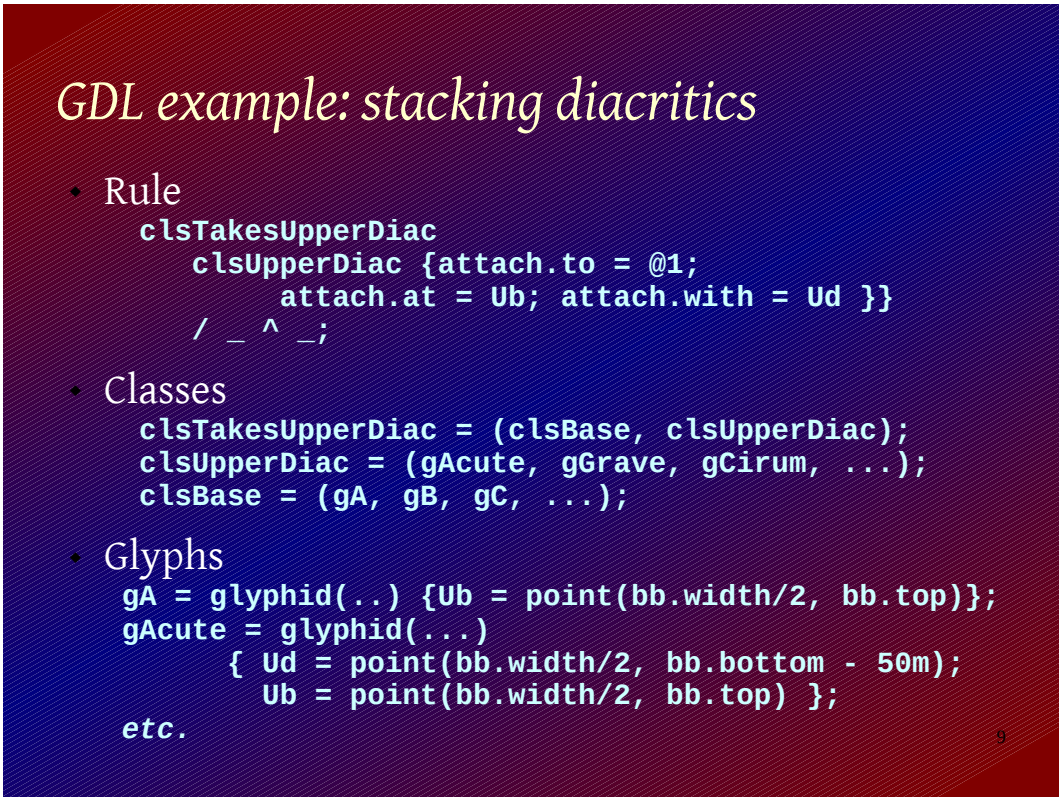
Note that the items in the `clsDotted` and `clsDotless` classes must be listed corresponding order, due to the way substitution rules works. That is, the substitution process determines the index of the input glyph in the input class, `clsDotted`, selects the corresponding item from `clsDotless`, and places it in the output.



A more complex example involves splitting, as is often seen in scripts from South Asia. The rule above replaces a vowel that must be split with its left- and right-hand halves. More accurately, it inserts the left half into the output and replaces the original glyph with the right half. The underscore in the input represents the location where the insertion will occur, that is, before the consonant. The "@2" syntax indicates that the consonant should be copied into the output without any change.

Again, the classes and glyphs used in the rule must be defined. In this case, there are three corresponding classes, the original vowel, the left-hand half, and the right-hand half. The "$3" syntax in the rule indicates that the index of the third item in the rule (`clsSplitVowel`) is what should be used to select from `clsSplitVowelLeft`.

The glyph definitions show examples of using the Unicode mappings, glyph IDs, and Postscript names.

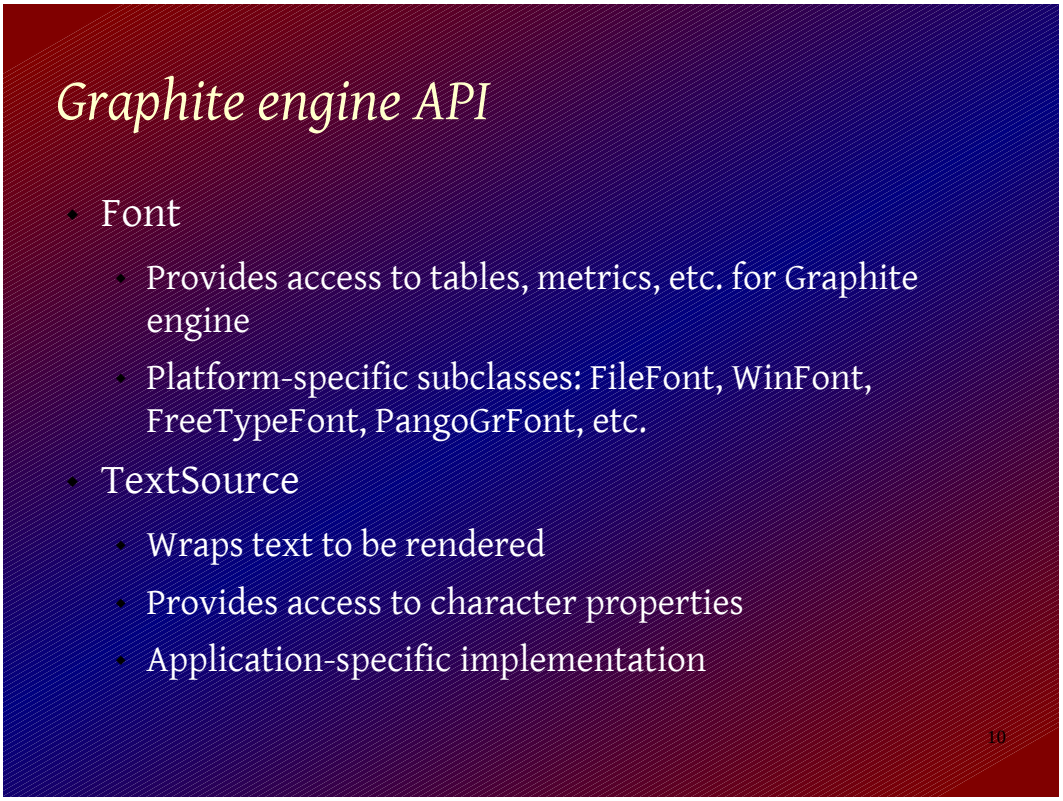


The rule above handles attaching an upper diacritic to a base glyph or to another diacritic. The statement `attach.to = @1` says to attach the diacritic to the first item in the rule (`clsTakesUpperDiac`), and `attach.at = Ub` says to attach it to the point named "Ub" on that first item. The statement `attach.with = Ud` says to use the point named "Ud" on the diacritic. The two glyphs are positioned so that the attachment points exactly coincide.

The class of glyphs that can take upper diacritics includes not only base glyphs (`clsBase`) but also upper diacritics themselves (`clsUpperDiac`). This allows the diacritics to stack.

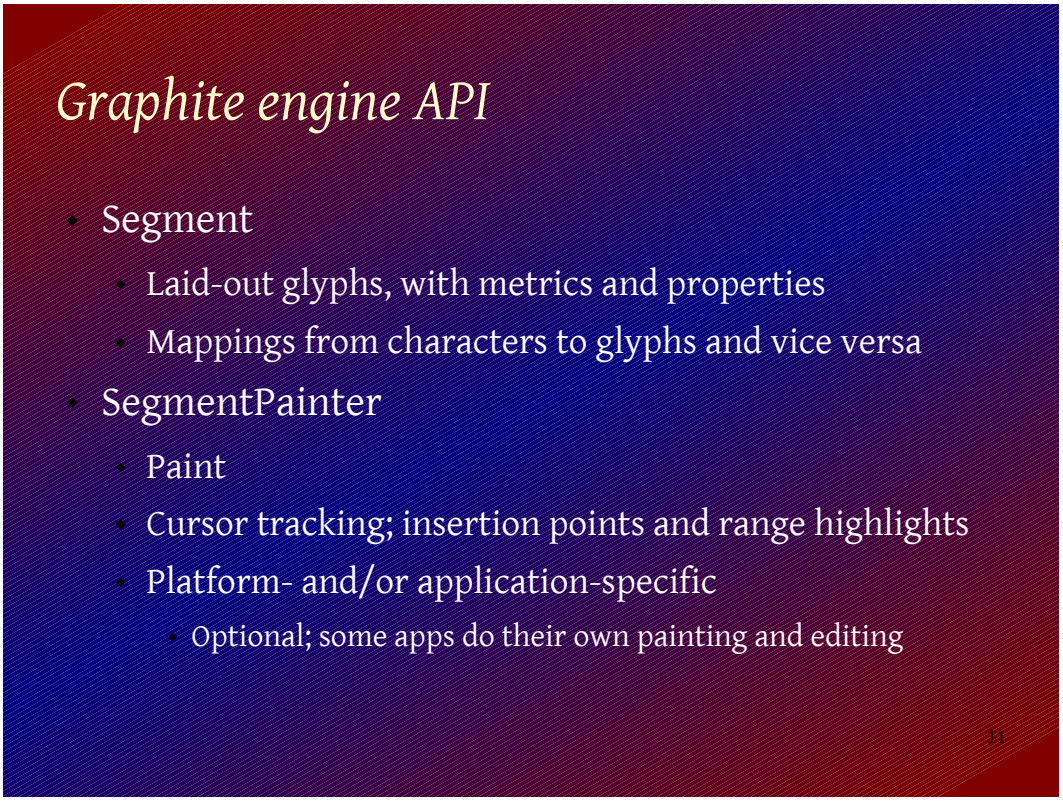
Another key aspect of allowing stacking can be seen in the context of the rule, "`/ _ ^ _`". This code indicates that after running the rule, the processing position must be backed up so that the second item, the diacritic becomes the place from where the next rule is run. This allows the rule to be fired again with the diacritic serving as the base (`clsTakesUpperDiac`).

Every glyph in `clsTakesUpperDiac` must define the Ub attachment point. In this example the point is defined in terms of the glyph's bounding box (bb), but they can also use the advance width, font ascent or descent, numerical values, or points on the glyph curve. Because diacritics are part of both `clsTakesUpper-Diacritic` and `clsUpperDiac`, they must define both the Ub and the Ud attachment points.



The Graphite API consists of four main classes. The **Font** class provides access to the font tables and glyph metrics. Font is an abstract class; a number of subclasses exist corresponding to various platforms and programming environments.

TextSource is a wrapper for the text to be rendered. Its interface allows Graphite to query for the characters to render, character properties, etc. Generally each application will implement its own version of TextSource, depending on the complexity of its text model. A simple version is available as part of the open-source code.

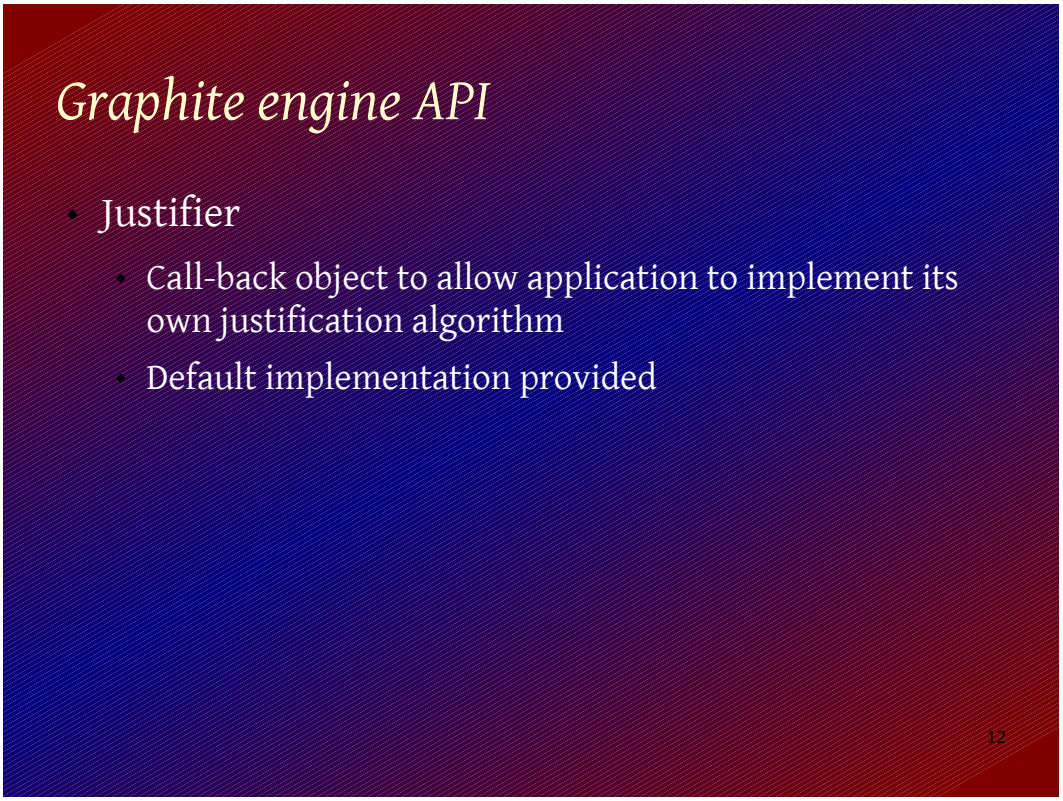


The **Segment** object represents a sequence of glyphs laid out on a single line and ready to be drawn. It also records mappings from characters to glyphs and vice versa. A Segment can be queried for these mappings as well as glyph metrics and properties.

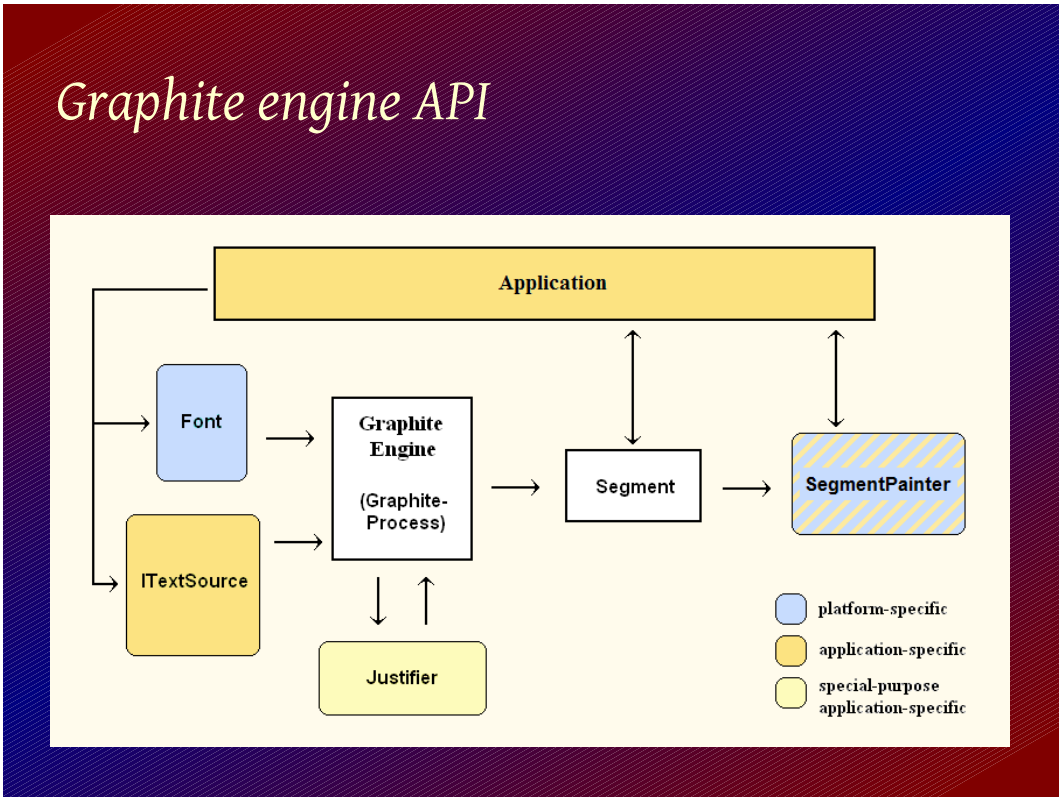
If multiple fonts and/or styles are used on a single line, the line will require multiple Segments. A segment corresponds to what other systems often call an "item."

A **SegmentPainter** is an object that is capable of painting a segment on an output device and handling editing operations. The SegmentPainter can draw an insertion bar (or split bars) at a given character location, or highlight a range of text. It can return the location of the highlight for the benefit of scrolling routines. It can convert a mouse click location to a charater index, which may involve distinguishing areas of ligatures that correspond to distinct characters.

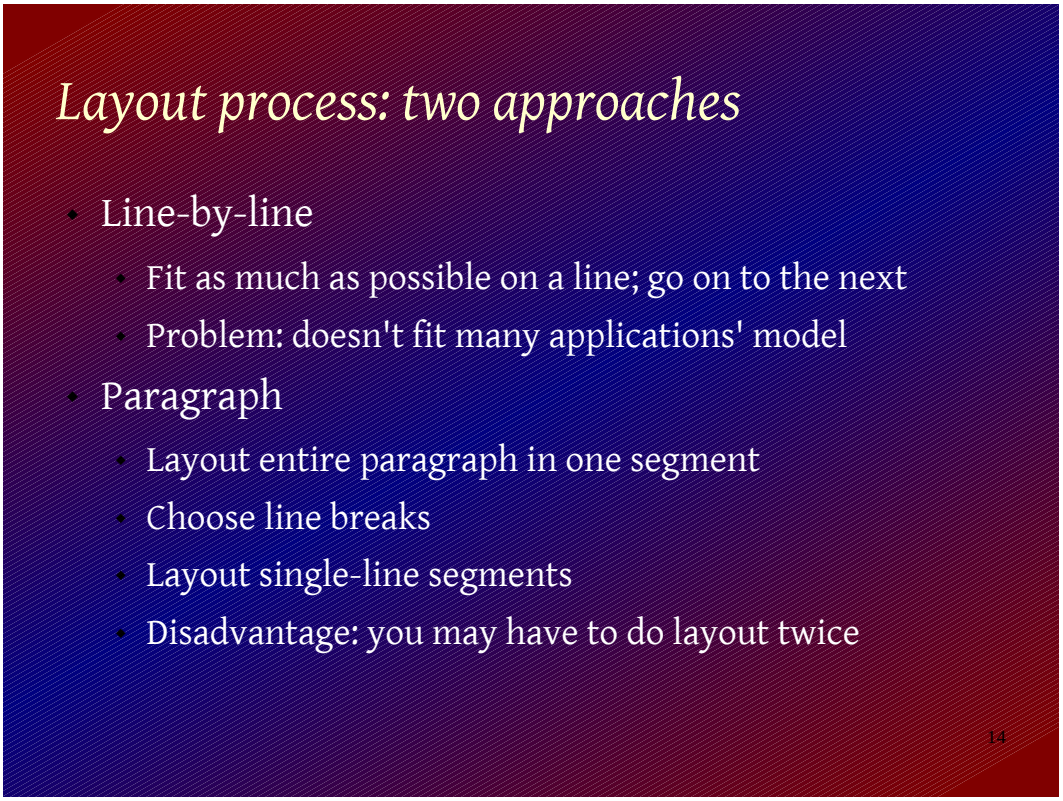
A SegmentPainter works in conjunction with a Font, and so there are platform- or environment-specific subclasses. An application may also override part or all of the standard functionality of SegmentPainter. Or it may implement editing support directly (via queries directly to the Segment) without using a SegmentPainter at all.



An additional class (less important than the others) is Justifier, which allows the application to implement its own justification algorithm. A stock implementation of Justifier exists that is quite adequate for most applications.



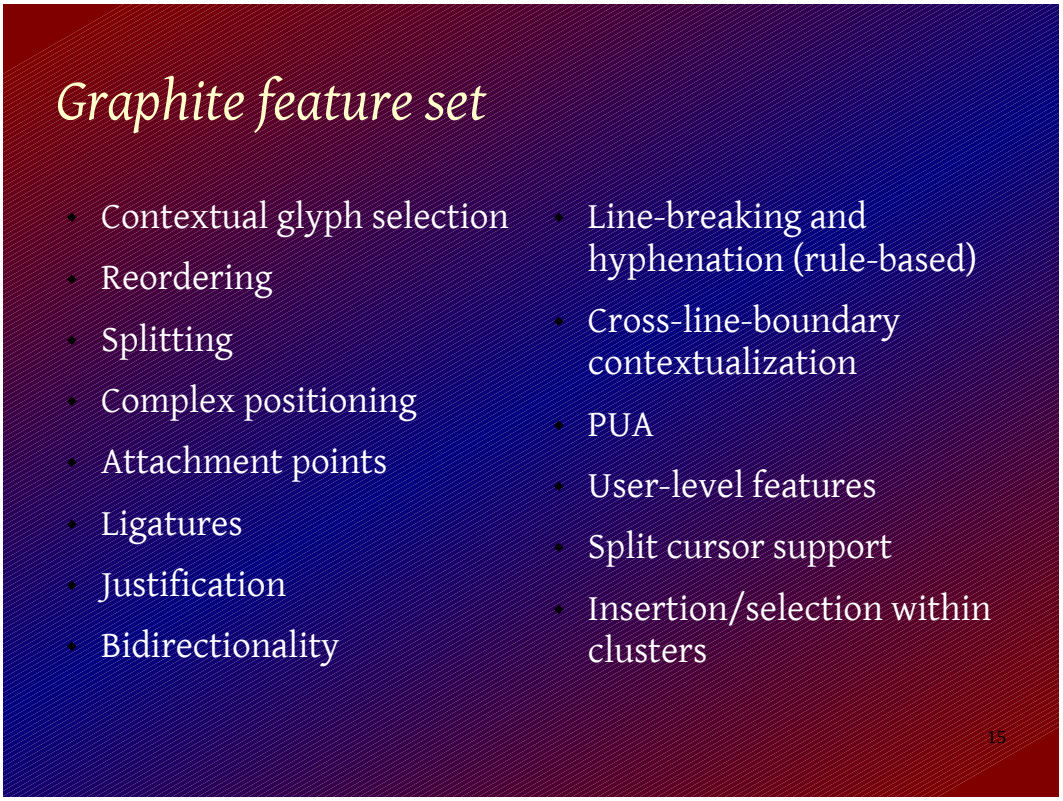
The above chart shows the process of rendering with Graphite. The application creates a Font and TextSource and sends them to the Graphite engine. The engine may possibly interact with the Justifier to lay out justified text. The output from the engine is a Segment. To support editing behaviors, the application can either interact with the Segment directly or create a SegmentPainter that wraps the Segment.



The Graphite API supports two models of paragraph-level layout. The first is to render a line at a time, asking Graphite to lay out as much as will fit and and return both the laid-out text and the line-break index. The next line will start at that point.

This was the approach assumed by version 1 of the Graphite API. The disadvantage we discovered is that this does not fit the model of many applications (OpenOffice in particular) that perform line-breaking as a very separate operation from layout.

So version 2 of the Graphite API also supports a different approach. This is to lay out an entire paragraph as one segment, which will provide all the information needed to make decisions about line-breaks—breakweights, metrics, etc. One these decisions have been made, each line is laid out as a separate segment. Of course, a disadvantage of this approach is that all text must be laid out twice.

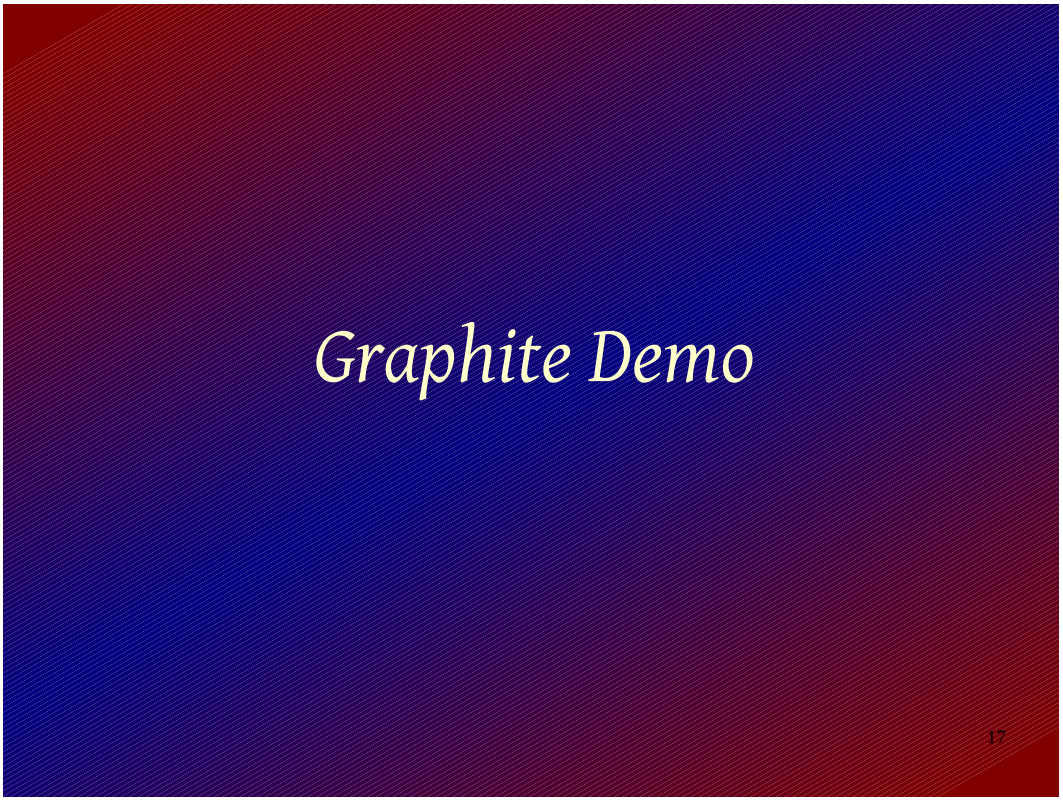


Above are listed the various complex behaviors supported by Graphite. These provide adequate support for all known modern writing systems.

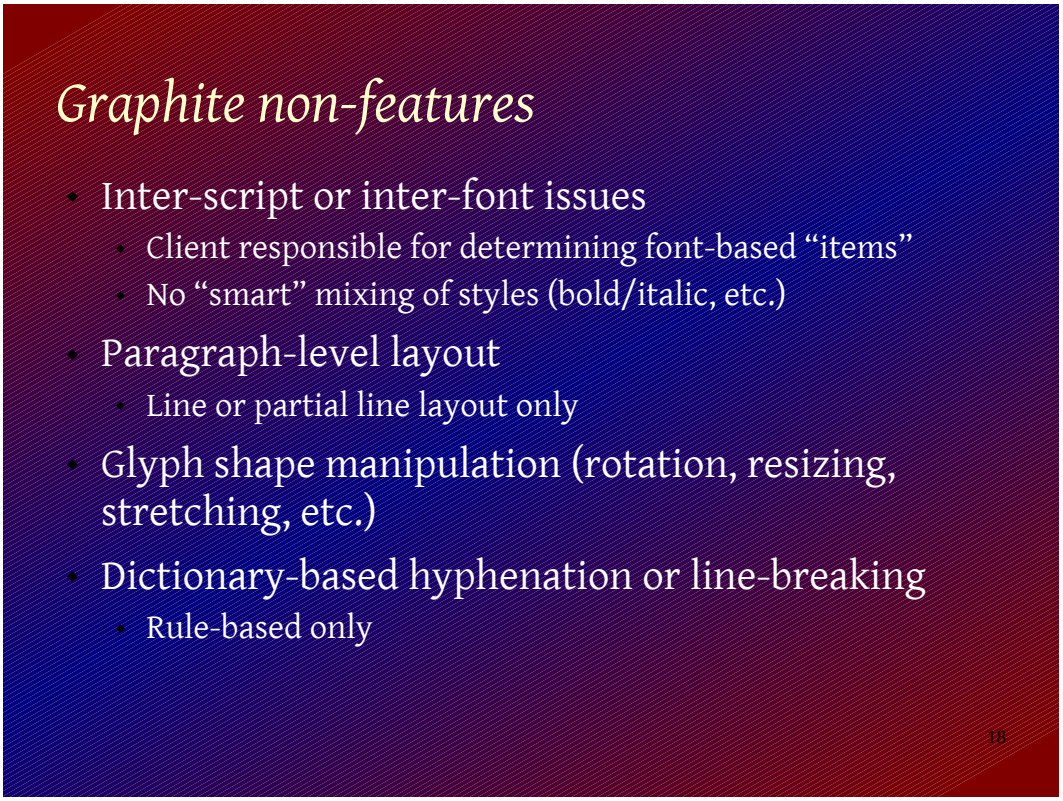
One feature that we feel has been quite costly and not particularly useful has been cross-line-boundary contextualization. This would allow, for instance, reordering across a line-break, permitting a hyphenation-break in the middle of a word that involves reordering. We are currently considering whether to it might be as well to remove this capability from future versions. At the very least, it is probably not necessary for a unified approach to support this behavior.

An important capacity in Graphite is the ability provide user-level features. These allow the user to specify variations in the way many characters are displayed or behave. Features may involve selecting alternate glyphs, adjusting the positioning of diacritics, or changing the way editing mechanisms behave.

SIL's Roman/Cyrillic and Arabic fonts include quite a number of user-level features. Some of these are need for special purposes such as literacy materials, others are language-related or regional preferences, and others represent archaic forms.



The demo will show examples of user-level features and some of the advanced editing capabilities.



Above are listed some of the limitations of Graphite support. Graphite is only intended to provide single line layout; it is the responsibility of the application to do paragraph-level layout. A segment includes text involving only one font and style (bold/italic); the application is responsible to create and lay out multiple segments on a line where needed, and there is no “smart” contextual behavior between segments.

Graphite does not perform any glyph manipulation such as stretching or rotating.

Graphite provides support for hyphenation and line-breaking that can be described by transformation rules. More complex needs such as dictionary-based line-breaking is not supported.

Graphite was originally implemented on Windows, and has been ported to Linux. It also runs on Mac X11, but not under the native Aqua graphics environment.

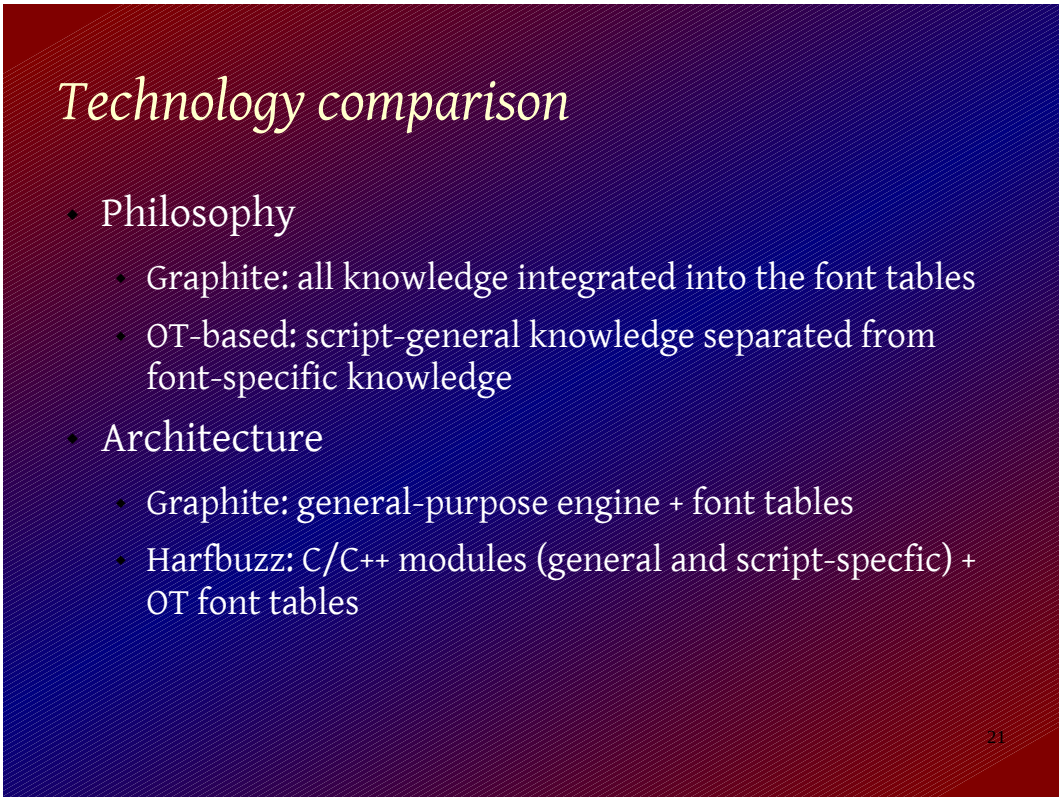
Graphite support has been added to Pango, OpenOffice, and XeTeX (Unicode-based implementation of TeX by SIL's Jonathan Kew).

An alpha version of a plug-in for InDesign exists on Windows; it does not handle right-to-left layout.

Some initial work has been done to create a Java-based wrapper.

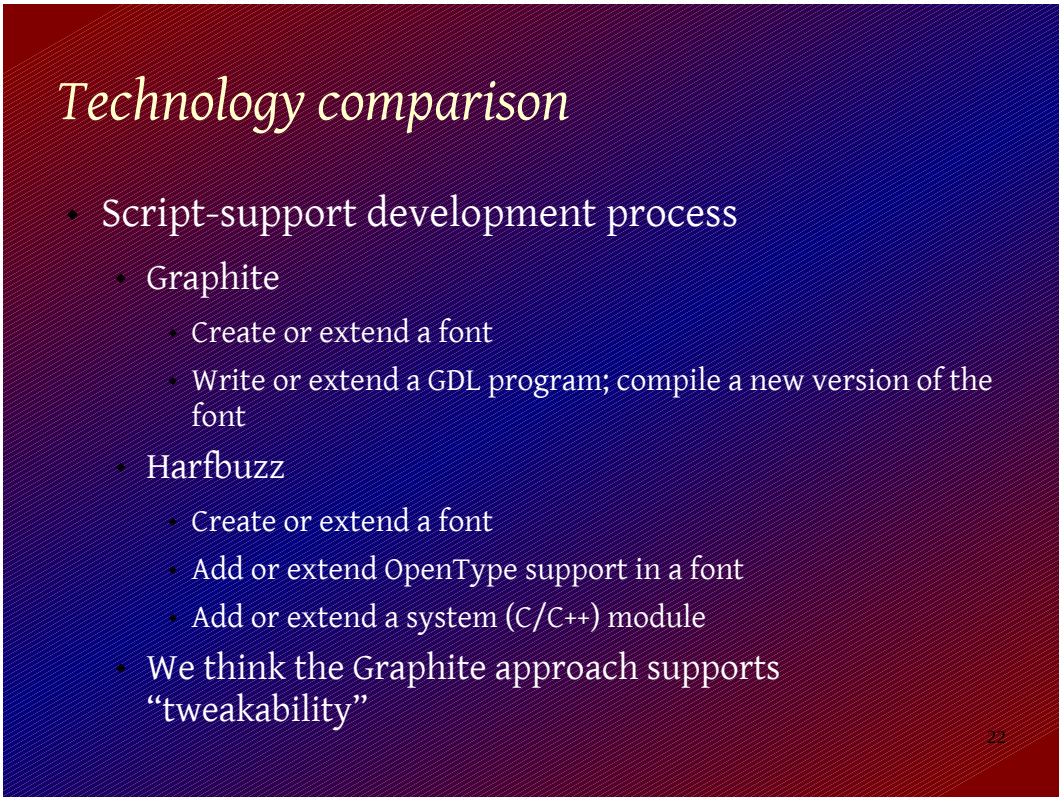
No serious attempt has yet been made to optimize the Graphite engine. Currently the performance seems usable but a little sluggish, depending on the application's approach to layout.

The above list shows the Graphite fonts that have been developed or are expected to be available in the near future.



As previously discussed, there are two main philosophies that govern complex-script systems: one that treats script- and font-related knowledge separately and one that integrates them.

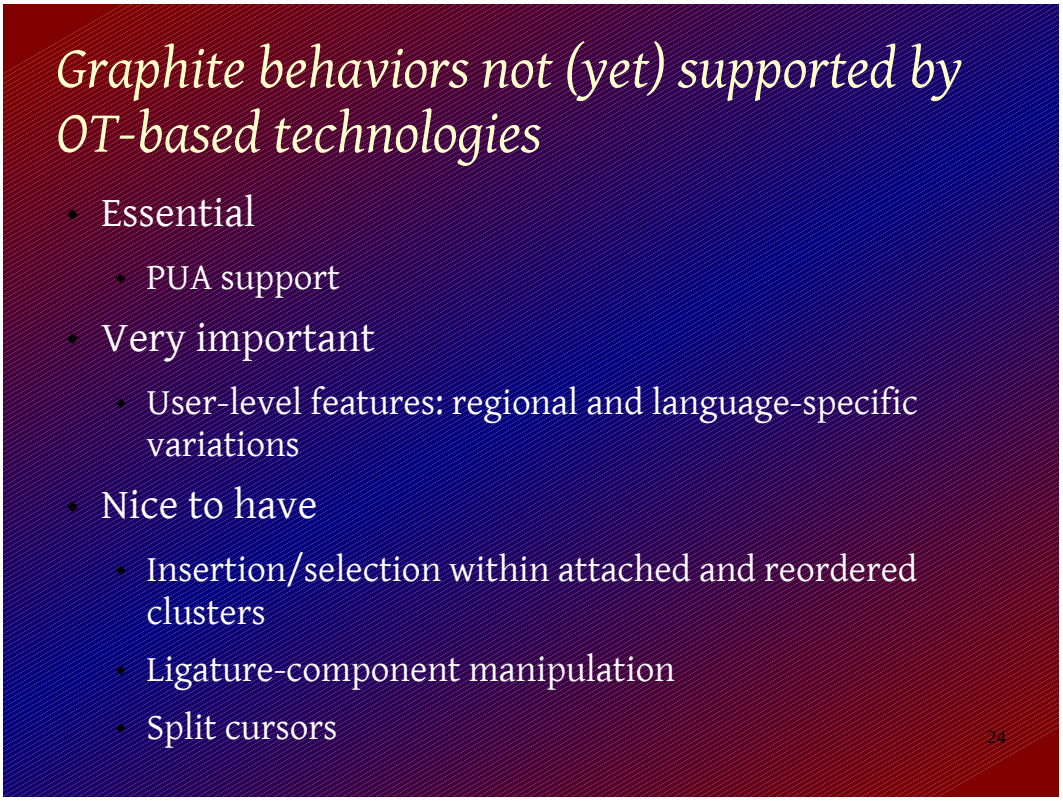
With regards to system architecture, Graphite consists of a single general-purpose engine that reads and interprets font tables. OpenType-based technologies include script-specific modules as well as general-purpose OT-driver modules that make use of the OT-font tables.



In order to create support for a script or language-specific writing system in Graphite, one creates a font or adds the appropriate glyphs to an existing font, and then creates or extends the corresponding GDL program and invokes the Graphite compiler. In contrast, script support in an OpenType-based technology like Harfbuzz involves not only OT programming but programming a script module in a standard language like C or C++. Because Graphite support involves only one font technology and one special-purpose programming language, we feel that it is a better approach to enable support for minority languages, many of which require "tweaking" of an existing implementation. For this reason, Graphite can be a signficant technology in helping bridge the "Digital Divide."

The basic approach to developing a Graphite font is to write a GDL program and compile it into font. There is a Perl module that can assist with much of the tedium of setting up glyph classes and attachment (anchor) points by extracting information from the font and related tools.

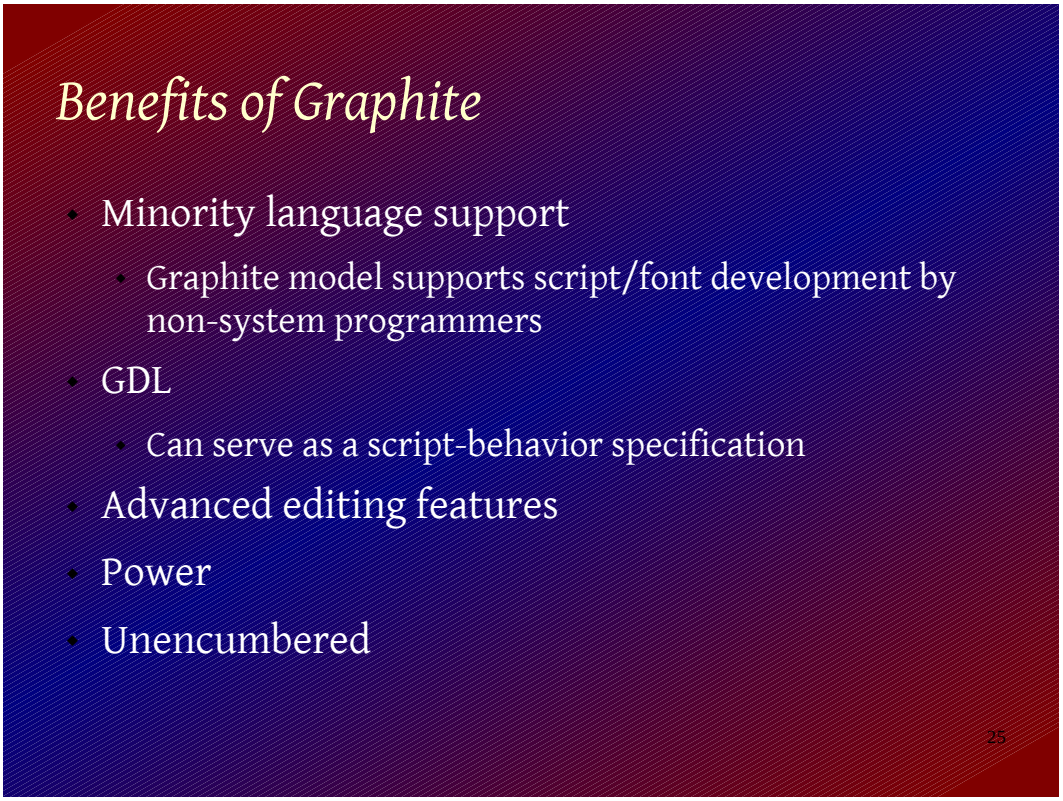
Unfortunately, there is no equivalent to VOLT for Graphite. In particular, a tool to visually place attachment points (anchor points) would be extremely useful. Currently there are two second-best approaches: create anchor points in FontLab and export them to an XML file and from there to GDL, or create anchor points using VOLT and extract them into GDL.



Graphite includes some capabilities are currently not available in the OpenType-based systems. The ability to define rendering behavior for PUA characters is an essential aspect of SIL's strategy to handle nonstandardized writing systems. This is difficult in systems where some aspects of the behavior must be hard-coded into the engine.

The ability to define user-level features is quite important to handling the wide variety of regional and language-related variations that we encounter. (While there are ways to get around it, they are awkward and cumbersome at best.)

Graphite includes smart character manipulation to a degree not allowed by any other system. Ideally any unified text-rendering API would include hooks to make these capacities available to the application for any underlying technologies that support them.



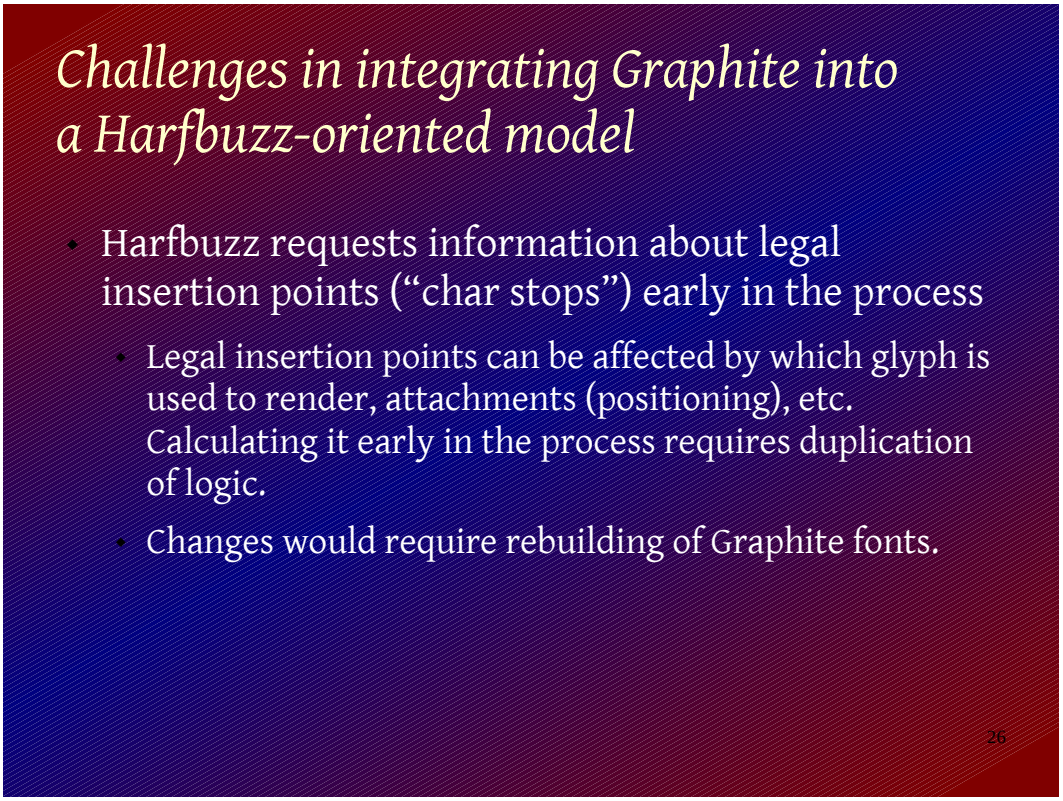
We believe that Graphite's pure smart font approach provides a good model for complex script support in many situations where Linux system developers are not likely to be involved.

The GDL programming language provides a way to specify rendering behavior that is both powerful and natural enough to serve as documentation for the script. This can in turn facilitate implementation using other technologies.

Graphite's advanced editing features, while not essential, can be very useful in writing systems that make extensive use of diacritics, reordering, and ligation. User-level features or application options could be used to make some of these behaviors optional.

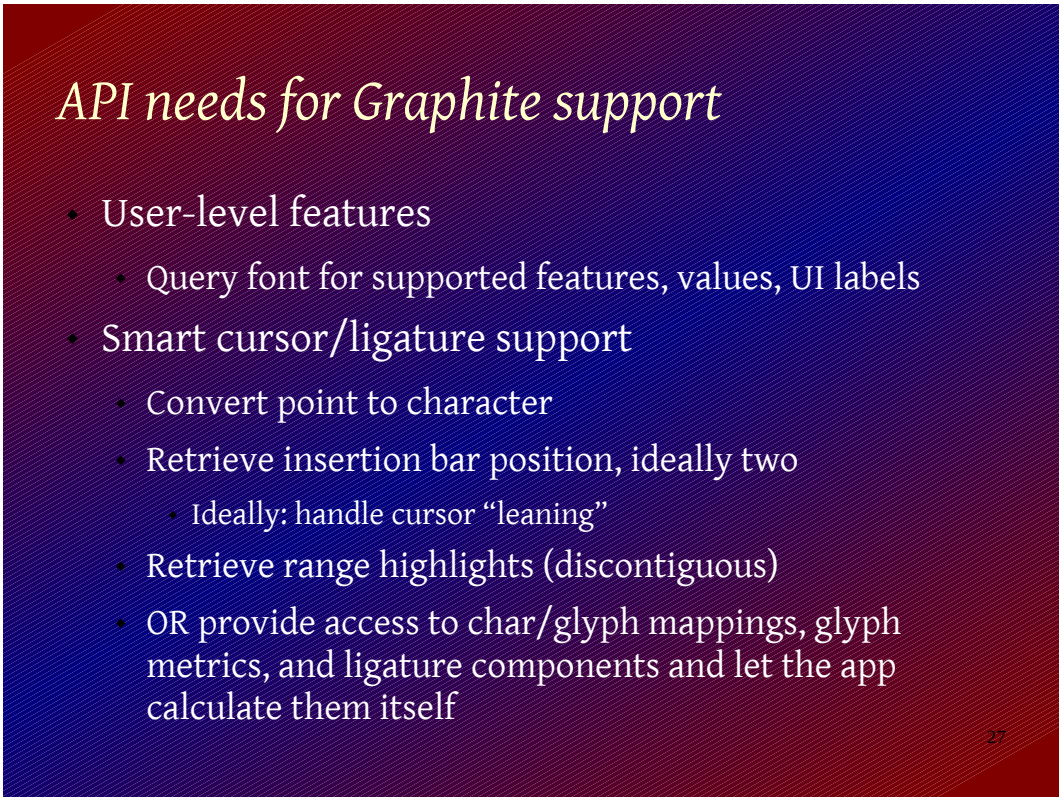
Graphite is the most powerful smart-font system in existence. Experience has shown us that many features that take weeks or months to implement in OpenType can be accomplished in a straightforward way using Graphite.

While OpenType is widely used and available, commercial stakes in the technology can be of concern to the open-source community. Graphite, on the other hand, is completely open-source and unencumbered, and SIL is committed that it remain so.



At this point, the main difficulty we envision in integrating Graphite into the current Harfbuzz-oriented model lies in the fact that Harfbuzz requires information about legal insertion indices (“character stops”) at an earlier stage in the process than it is available from Graphite. Graphite uses the approach it does in order to allow editing behavior, including insertion indices, to be based on the selection of glyphs or their positions. For instance, if you choose to represent tone using diacritics, you might not want to allow insertion between the base character and the tone marker, but if you use superscript numbers (possibly due to a user-level feature), it makes perfect sense to allow insertion before the tone.

Graphite's model could possibly be changed, but it would require some duplication of logic on the part of the GDL programmer (e.g., testing for a feature both at the point of calculating the insertion points as well as during glyph selection; or knowing about diacritic attachment early in the process). Also this change would require reprogramming and rebuilding all of the existing Graphite fonts in order for them to work correctly within the Harfbuzz model.



In order to provide support for Graphite's most useful features, a general text-layout API would need to include methods to query a font with regard to user-level features and defined settings. This includes not only numerical IDs but also strings that can be used to populate a UI mechanism, such as a feature menu.

Less essential but still very useful would be methods to provide support for advanced editing behaviors. This would involve either a way to make use of the Graphite SegmentPainter class, or a way for the application to implement the equivalent of SegmentPainter. The former would include methods to allow the text-layout engine to convert a point (mouse click) into a character position, to draw highlights, and to return highlight locations for the purposes of scrolling.

Alternately, the application could implement the equivalent of SegmentPainter directly. This would mean making available the glyph-to-character mappings, glyph positions and metrics, so that the application can perform hit testing and draw highlights. Full support would also include information about ligature component mappings and metrics—rectangular glyph areas that correspond to distinct underlying characters.

Currently Graphite's API includes the concept of insertion bar "leaning", which in particular affects the behavior of split cursors. Ideally, a general API would also support this concept.

Questions?

- Contact information:
  - Email: sharon_correll@sil.org
  - Graphite web site: graphite.sil.org
  - NRSI web site: scripts.sil.org