Using Graphite to Address Challenges in Nastaliq-style Arabic Script

Martin Hosken & Sharon Correll

SIL International

IUC 39

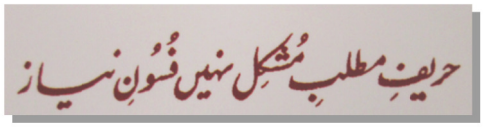
Santa Clara, CA

October 2015

Nastaliq

- Sloping calligraphic style of Arabic text
- Contrast with flat Naskh style:

حر یفِ مطلبِ مُشکِل نہیں فُسُونِ نیاز



- One of the most challenging forms of writing to implement in a computer font

Nastaliq is a sloping style of Arabic calligraphy used in Pakistan and other countries of south central Asia. The image in the slide shows text rendered in standard Naskh-style Arabic with a flat baseline (top line) compared with the same text below using Nastaliq. Note the diagonally sloping segments of text in the Nastaliq version. Due to its complex calligraphic form, Nastaliq has proved to be one of the most difficult forms of writing to support on the computer.

Where is Nastaliq used?

- Pakistan
  - Urdu: 186 million speakers (many as a second language)
  - 25 languages, 120 million speakers
    - Western Punjabi, Saraiki, Northern Pashto, Shina, Brahui, Balochi, etc.
- Afghanistan
  - 8 languages, 25 million speakers
  - Used in handwriting; not as widely on the computer
- Iran
  - 32 languages, 80 million speakers
  - Different style from Pakistani Nastaliq
- India

Nastaliq is used to write Urdu, the primary language of Pakistan. It is also used for many other languages in Pakistan and other parts of Asia, spoken by roughly 200 million speakers.

While there are several fonts that support Urdu, there are no fonts that support the lesser-known languages. Some of these languages do not yet have a standardised orthography, and for this reason, developing a font to handle them well using the trial-and-error method of fixing layout problems is not likely to be effective.

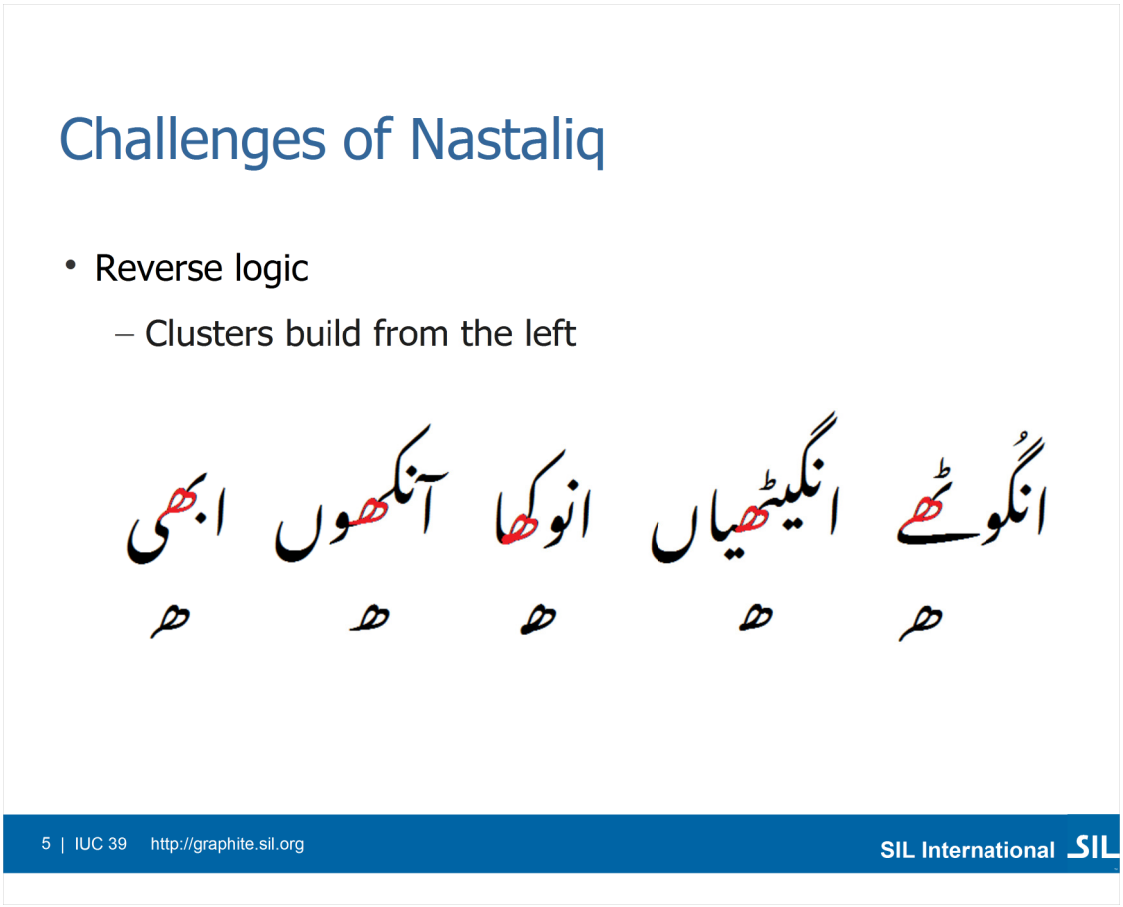
Challenges of Nastaliq

- Large number of glyphs
  - Isolate, initial, medial, final → ~64 forms
  - Special forms for every pair
  - Special Beh forms
  - 30-50 forms of each letter
  - ~700-900 base glyphs

All Arabic styles involve contextual shaping that requires initial, medial, and final forms of most of the letters of the alphabet. Nastaliq is more complicated in that it needs differing shapes corresponding to the specific neighboring letters – especially the letter following the one in question. This means that there needs to be on the order of N^2 glyphs relative to the number of basic letter forms.

Beh forms (used for the letters beh, teh, theh, peh, etc.) are particularly complicated due to the fact that a sequence of these forms must use alternating “teeth” - varying heights and shapes of the letters that help improve readability. Beh forms can also require shaping rules that take into account a larger context than most of the other forms.

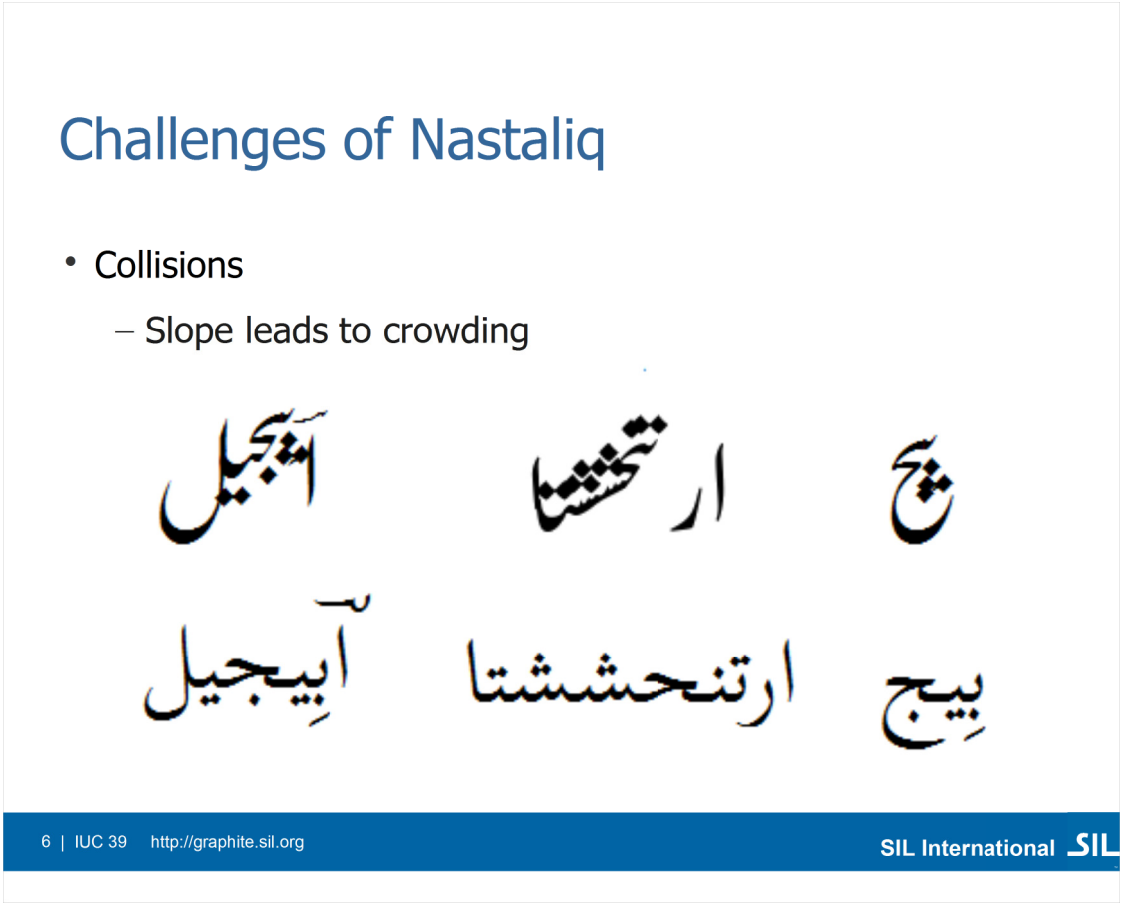
Due to these factors, Nastaliq fonts that use separate glyphs for each letter generally require about 700-900 glyphs to properly render the basic letters of the alphabet. The sheer number of glyphs is not an insurmountable obstacle in and of itself, but it does make the process to develop a Nastaliq font more complicated and quite resource-intensive.



Another challenge of Nastaliq is that the shape of most glyphs is primarily influenced by the letter that comes after it. This is in contrast to what we see in Latin handwriting.

The image above shows the varying shapes needed for the medial heh-doachashmee depending on the following letter (the one to the left). Keep in mind that while the graphic gives five examples, there are actually 15-25 forms required, for both the initial and medial positions.

The normal rendering process for a smart font is to start from the beginning of the text and work toward the end, but this does not work well for Nastaliq. For this reason OpenType has added a backward chaining mechanism that allows Nastaliq text to be processed from the end of the string to the beginning.



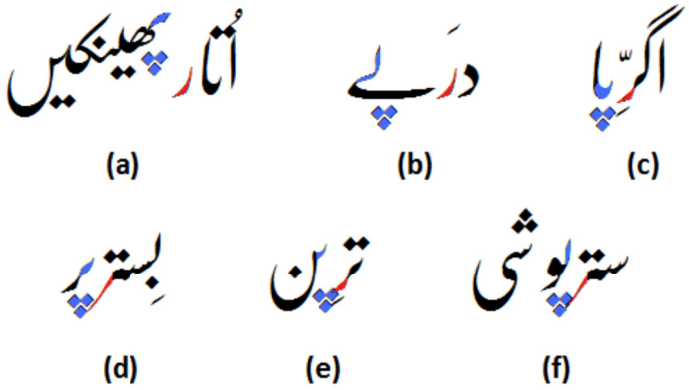
The biggest difficulty in rendering Nastaliq is the problem of glyph collisions. Limited collisions can occur in many complex fonts, but the problem is exacerbated in Nastaliq for several reasons.

First, the diagonal slope restricts the horizontal space both above and below the base forms where the nuqtas need to be drawn. Sometimes this effect is subtle and sometimes it is severe.

The graphics in the slide compares several naively-rendered Nastaliq words to the same text (below) in flat Naskh style. Collisions are not as common in Naskh style because the flat baseline generally provides for adequate space to hold a nuqta above or below the base. In Nastaliq, this space is reduced or may completely overlap with another base character.

Challenges of Nastaliq

- Collisions
  - Lots of forms makes collision prediction harder



As an example of the kerning problem, the images above show six words containing the letter reh (red) followed by the letter peh (blue). In examples (a) and (b), the rendering is correct with no collision occurring. In (c) there is a near-collision with a diacritic on the reh, and in (d), (e), and (f) there is a full collision between the reh and the triple-nuqta of the peh.

It is evident that fixing these collisions is not as simple as recognizing the sequence “reh + peh”. In the case of (a), the lack of collision is due to the length of the final sequence, which causes the peh to be positioned high enough to easily avoid the reh. So one might be tempted to use the length of the final sequence as the test for a collision. However, that is not a reliable indication in and of itself, since while most short final sequences result in a collision, (b) does not.

Also note that while (c), (d), (e), and (f) all contain collisions, the amount of kerning that would be needed to fix them is different for each one.

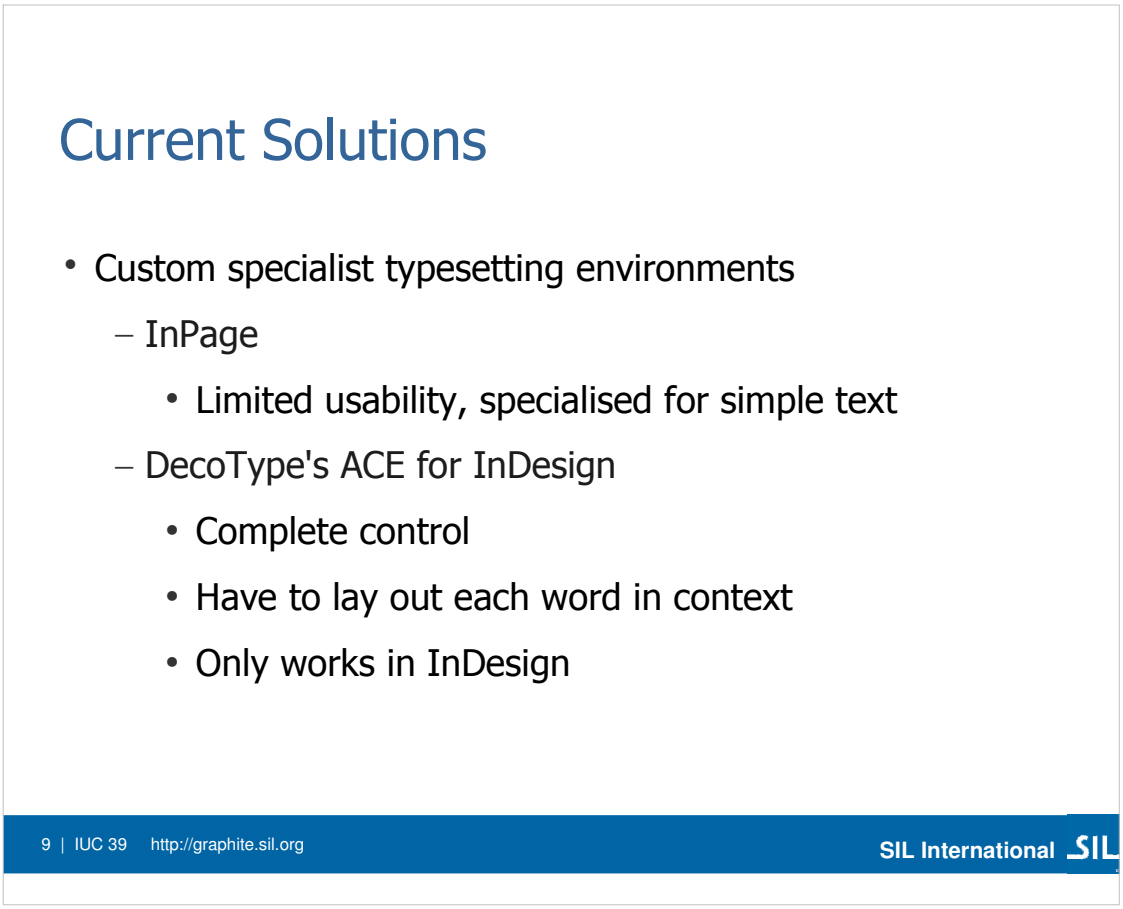
Different approaches have been used to create Nastaliq fonts using OpenType. Noori Nastaleeq solves the collision issue by using a separate glyph for each contextual sequence. This way the nuqtas can be placed in the most ideal position. The result is a very high quality font; however, it only provides good rendering for the character sequences that are anticipated and specifically added to the font. This makes it inappropriate for most languages other than Urdu. In addition, the mapping to compound glyphs cannot take diacritics into account, either in identifying the compound when a diacritic is in the sequence, or then positioning that diacritic appropriately in relation to the compound.

Most other fonts use a separate glyph for each letter and then include a plethora of special lookups to tweak the positions of the nuqtas. In Nafees Nastaleeq, for instance, about 40% of the OpenType lookups are there for the purpose of fixing collisions.

While Nafees uses a large collection of special anchor points to fine-tune nuqta positioning, Noto Nastaliq uses more specific classes for shifting, as well as nested contextual rules.

Such fonts also end up pushing the bounds of OpenType and so have problems with the inconsistencies that exist between different OpenType engines. Fonts then end up being engineered for one particular version of an engine and having problems if that engine changes or a different engine is used.

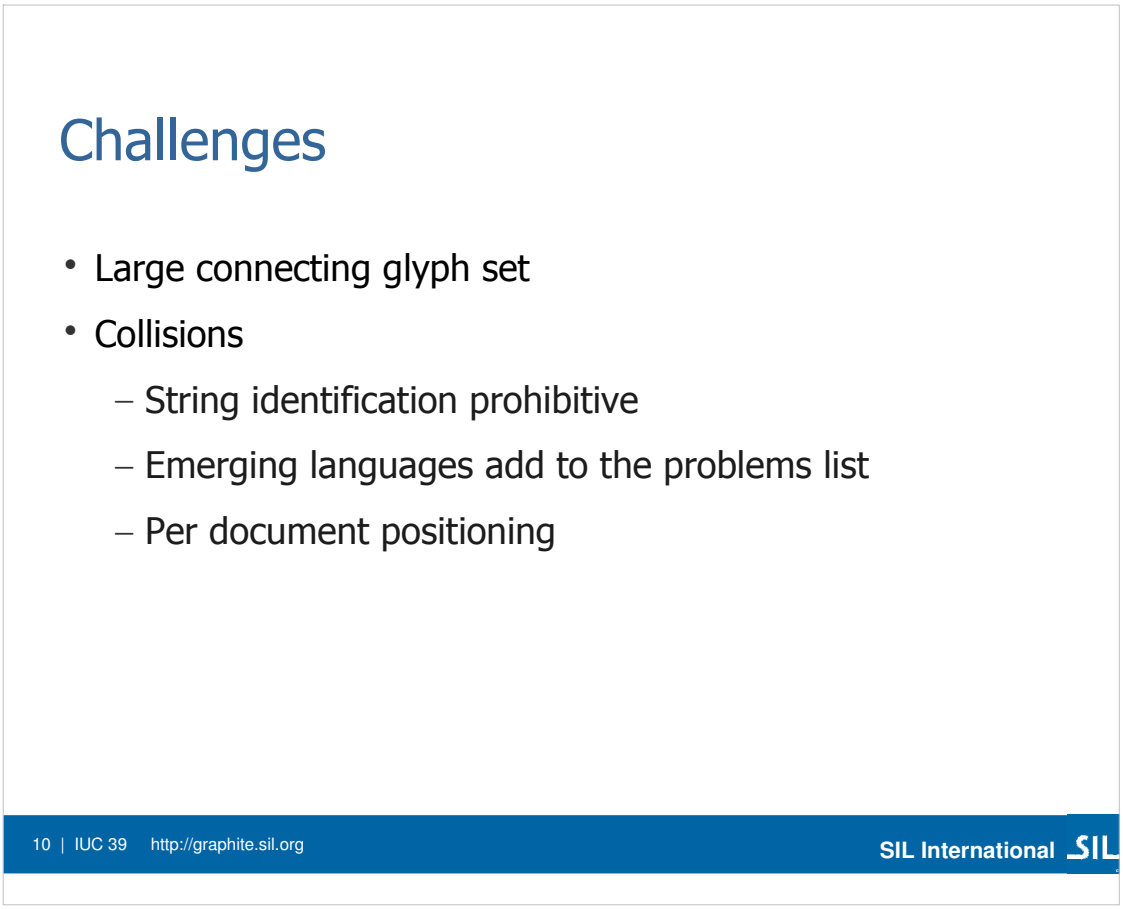
Few of the fonts handle vowel marks well. This is acceptable for Urdu, but not for other languages, such as Saraiki and Marwari, that make more extensive use of vowel marks.



InPage is a specific word processor to handle Nastaliq layout. It's OpenType-based with added kerning capability.

Thomas Milo has worked for many years on addressing the high-end typesetting problem: how can we make documents that look as good as hand calligraphy? He has a plug-in for InDesign that allows for per-word control over glyph placement. This allows for whitespace balancing and other high-end layout techniques. However, ACE requires considerable user interaction with the document, and this is beyond what most users want when creating documents. There is a need to be able to type and get a reasonably good result, without having to work through a detailed layout of each word.

So we see a need for an automatic, unguided approach that may not be as powerful as high-end typesetting but is adequate for most needs.



Our research indicates that using naive rendering (that is, with no particular effort at fixing collisions), 20-30% of words will show a collision. So fixing the collisions is not optional and requires considerable effort.

Noto Nastaliq is a recently released font that does a particularly good job of handling collisions. However, we still find that 2-3% percent of words in our dictionaries show collisions. This may be adequate for text, email, or even web pages, but is not acceptable for high-quality print publications.

In summary, the large glyph set and backward logic requirements for Nastaliq are difficult but solvable issues. The collision fixing problem, however, has proved to be quite intractable, and we think a different approach is needed.

The current unmet need

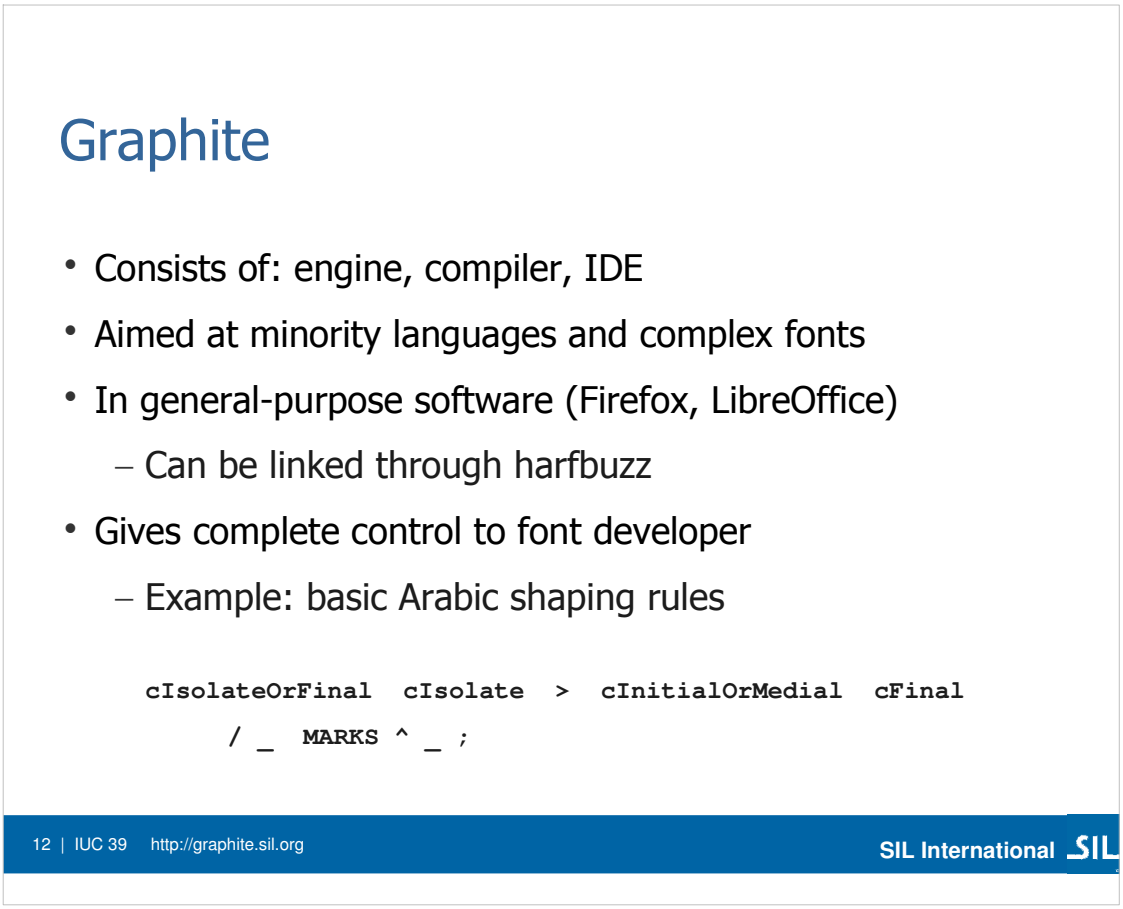
- Minority language support
  - Vowel diacritics
  - More bases and diacritics
  - More bases have nuqtas
  - More nuqtas (3, 4)
  - Unknown sequences

At SIL International, we specifically see a need for a Nastaliq font that can handle a wide variety of languages in addition to Urdu. Lesser-known languages often require more vowel diacritics than Urdu. They may use a different set of base characters and diacritics, and the base characters often include more nuqtas to represent sounds that are not present in Urdu or standard Arabic.

When dealing with multiple languages, it is not feasible to use a trial-and-error approach to testing and fixing character sequences – especially since many of the languages in question do not have standardised orthographies.

Also such a font needs to be usable in general-purpose software such as LibreOffice or Firefox, and on phones and tablets, where there is no capacity for hand-fixing the collisions.

Because of the limitations we have seen with the existing fonts, and particularly the collision problem, we are implementing a Nastaliq font using Graphite extended with some extra features to address collisions.

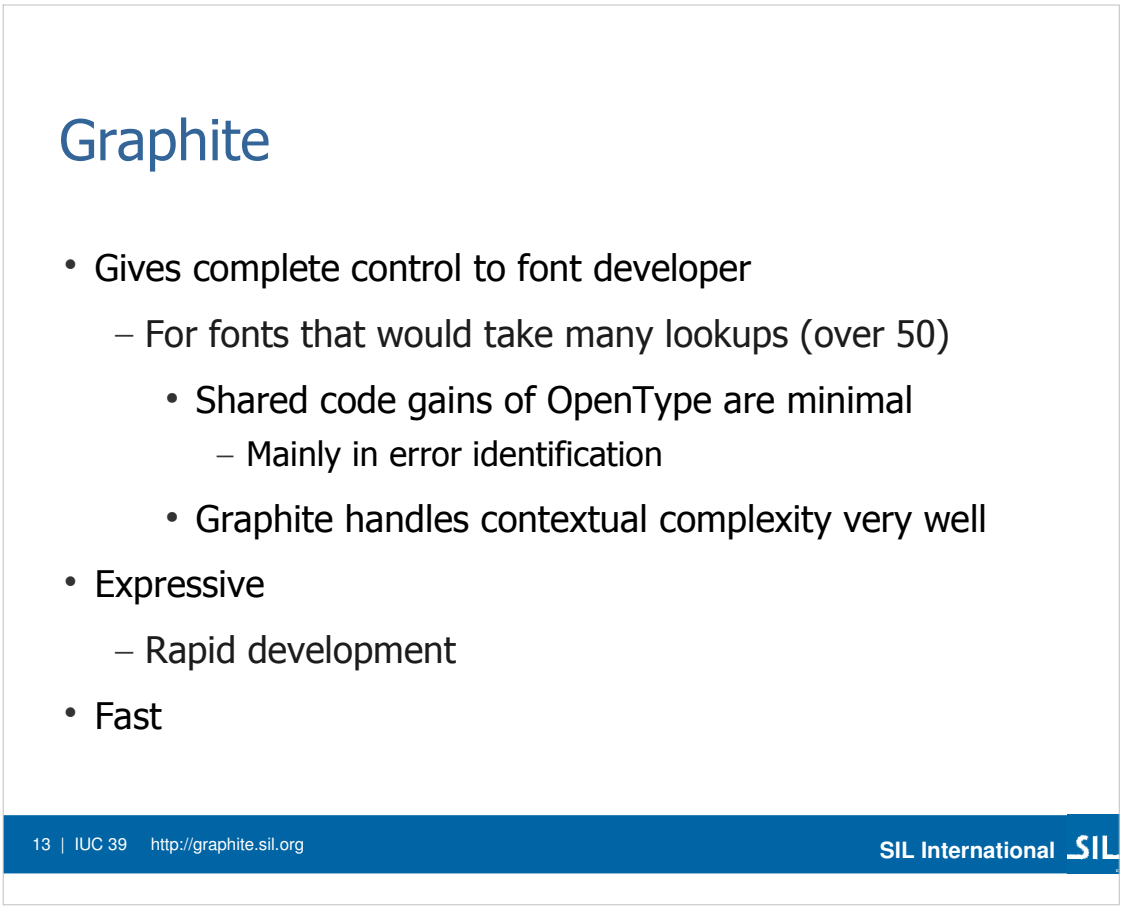


Graphite is the smart-font technology developed by SIL International and specifically designed to be flexible enough to accommodate the needs of minority languages. Unlike OpenType, Graphite does not require any knowledge about the script to be built into the rendering engine, and this makes it especially useful for scripts that are not part of industry standards or for languages that use extensions or modifications to the standardised rules of writing a script. This approach also allows script development to move forward without having to return to the application developer to update their libraries to handle shaping changes.

Some have expressed concerns about the need to reimplement basic knowledge about script behaviour in every Graphite font. But this is often necessary to support minority languages, and we believe in most other cases the cost is not at all prohibitive.

For instance, the snippet of GDL code shown above is the single rule that performs contextual shaping in Arabic. All that is necessary in addition is to define corresponding classes of isolate/initial/medial/final forms for dual-linking and right-linking letters and a list of diacritical marks. The power to customise font behaviour is well worth the bit of extra code involved – especially considering that OpenType script implementations are often inconsistent and a font that works correctly using one engine may be buggy with another.

In effect, shared behaviour across fonts is handled by shared code.



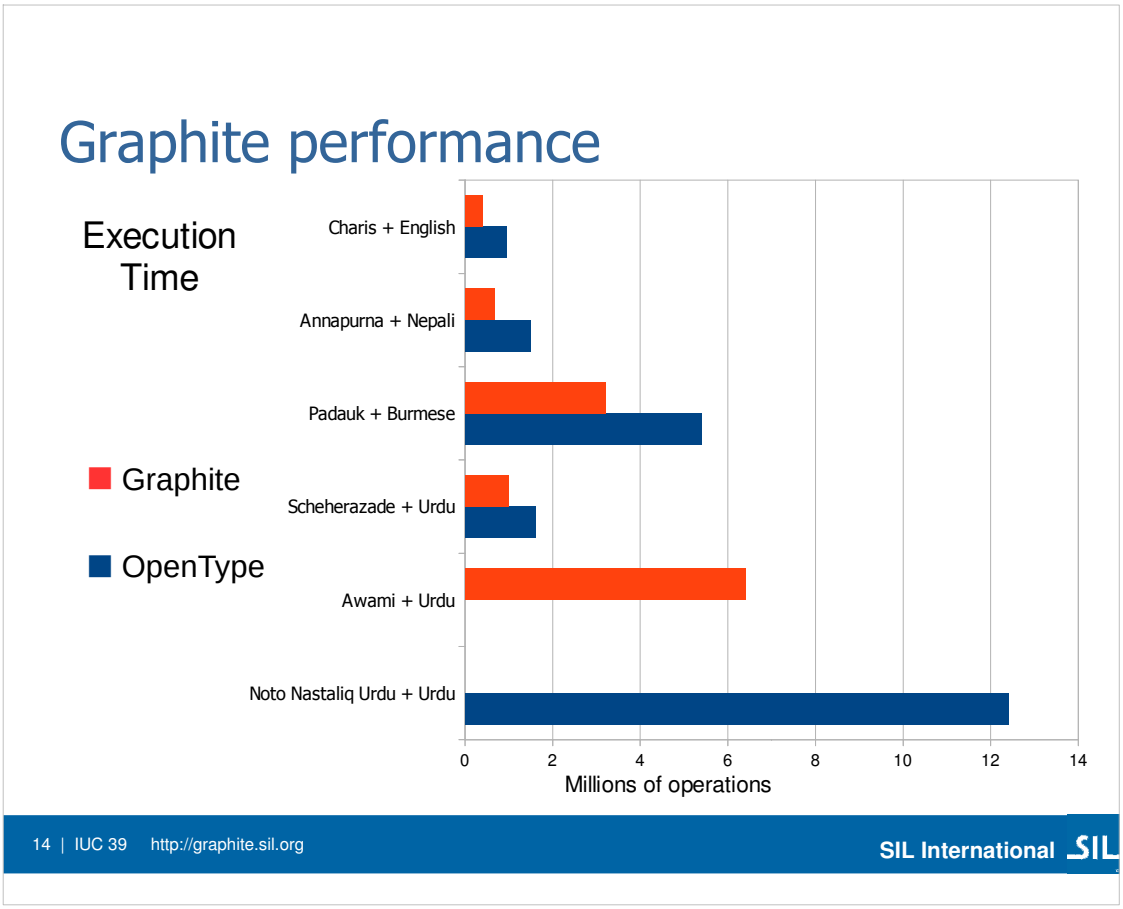
Graphite fonts are developed by writing a program using the Graphite Description Language (GDL) and compiling that program. The compiler adds Graphite-specific tables into the TrueType font.

The GDL rule is conceptually similar to an OpenType lookup, but permits considerably more flexibility:

- Optional items allow multiple rules to be generated from one rule
- Insertion and deletion of slots, allowing for reordering, all within the same rule
- Split contexts, where contexts can occur between strings as well as before and after.
- Multiple rules per pass
- Rules that move backward, or reprocess the output from a rule, within the same pass
- Rule matching constrained by tests of slot and feature attributes.

Our experience indicates that while OpenType may be a little easier to use for simple fonts, Graphite can require considerably less development effort for complex situations.

A supposed advantage of OpenType is its ability to recognise “illegal” character sequences. But this behaviour can be problematic if it means that a faulty error identification makes it impossible for a community to render the character sequence they need properly. Having complete control in the font mitigates against that. In addition, there is no need for an application upgrade to solve a shaping problem.

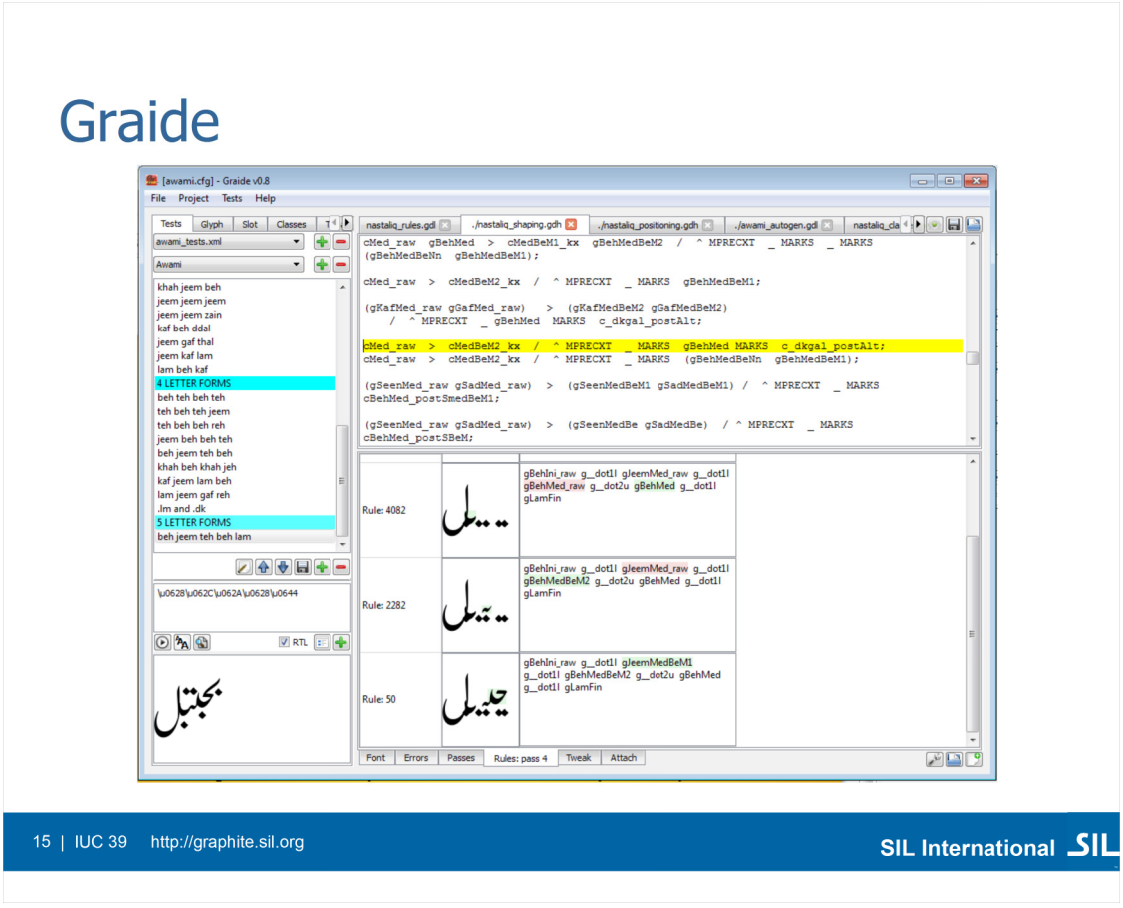


The timings here are from valgrind and are for a paragraph out of the UDHR in the given language.

Noto Nastaliq Urdu used to be much faster than this (about 9.3m), but all the special cases required for collision fixing is costly in terms of speed. One of the complaints we received about Nastaliq text processing is how painfully slow rendering can be.

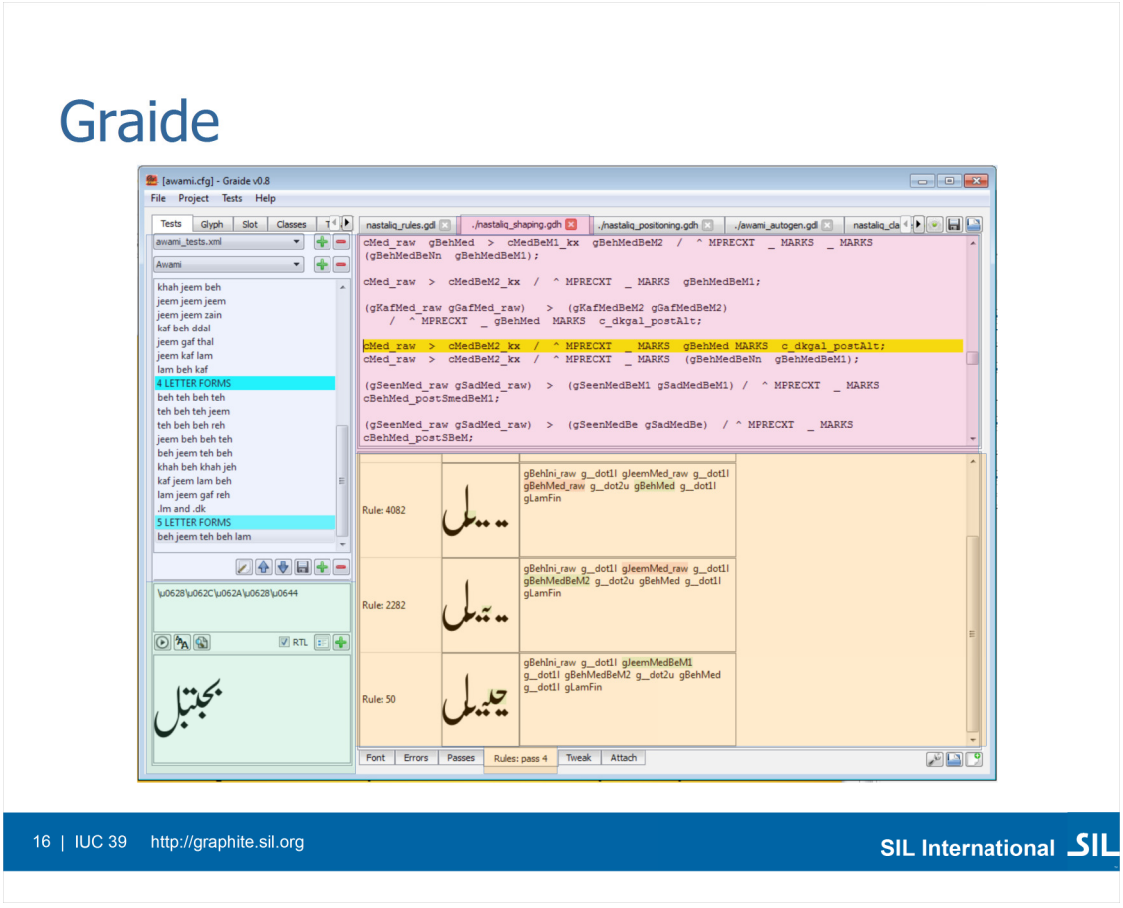
20% of the processing time for Padauk is spent testing for illegal character sequences.

One thing this graph shows is that doing part of the shaping in the engine is does not necessarily make it faster.



A Graphite development tool called Graide is available and, while still at version 0.8, has been used effectively for several font projects including SIL's Nastaliq font. Graide allows the user to edit code, build the font, see rendering results, debug, and manage a suite of test cases.

This screenshot shows Graide (GRAphite Integrated Development Environment) being used for Graphite font development.



Some different areas of the tool are highlighted:

- The column to the left (blue) shows a suite of short text samples that can be used for testing and debugging.
- The rendering of the selected sample is displayed in the lower left pane (green).
- One of the GDL source code files appears in the main upper region (pink).
- The lower pane (orange) shows a sequence of rules that have been fired during the selected pass (pass #4), with the intermediate state of the rendering displayed at each step. The user has clicked on one of the rules, and the corresponding line of source code is highlighted in yellow in the pane above.

But wait. There is more!

While Graphite has somewhat more expressive power than OpenType, we felt that it was still not adequate to handle the very difficult problem of collision handling. So we have extended the Graphite system with a mechanism to detect and fix collisions automatically.

Could we come up with a way to address the layout needs of Nastaliq that would alleviate all those special case rules, do a good enough, if not perfect, job, and be quick enough? (It doesn't need to be *absolutely* perfect, because the font author can always fall back on the absolute positioning of contextual rules for the relative handful of really hard cases.)

This is our goal.

Collision Avoidance

- Overview of how it works
  - Outline approximation
  - Collision avoidance
  - Weighted resolution
  - Graphite control
  - Kerning

There are a number of sub-problems within the overall problem that needed to be addressed.

Comparing two glyph outlines for overlap is a hard problem that takes too long. Is there some way to reduce the outlines to something close and manageable?

If we can spot a collision, what do we do then? Can we position things so that there is no collision?

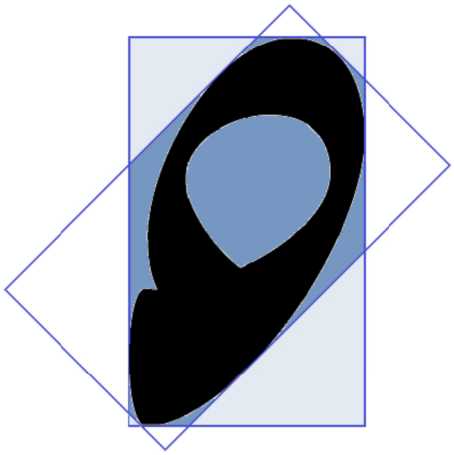
Even if we can position things so they don't collide, can we position them in a way that fits nicely with the principles of Nastaliq calligraphy?

Once we have a processing model for all this, what are the smart-font controls and parameters that will allow a font engineer to get the results they want? We don't want to lock things down so tightly that font designers can't generate fonts with different stylings, etc.

Finally, there is the issue of kerning. Traditionally, Nastaliq is written with fairly tight kerning between the sloping text segments, but typical approaches to glyph position cannot handle this well. Shape-based kerning, however, makes this possible.

Outline Approximation

- Representation of glyph shape
  - Polygon defined by up to 8 line segments = 8 numbers



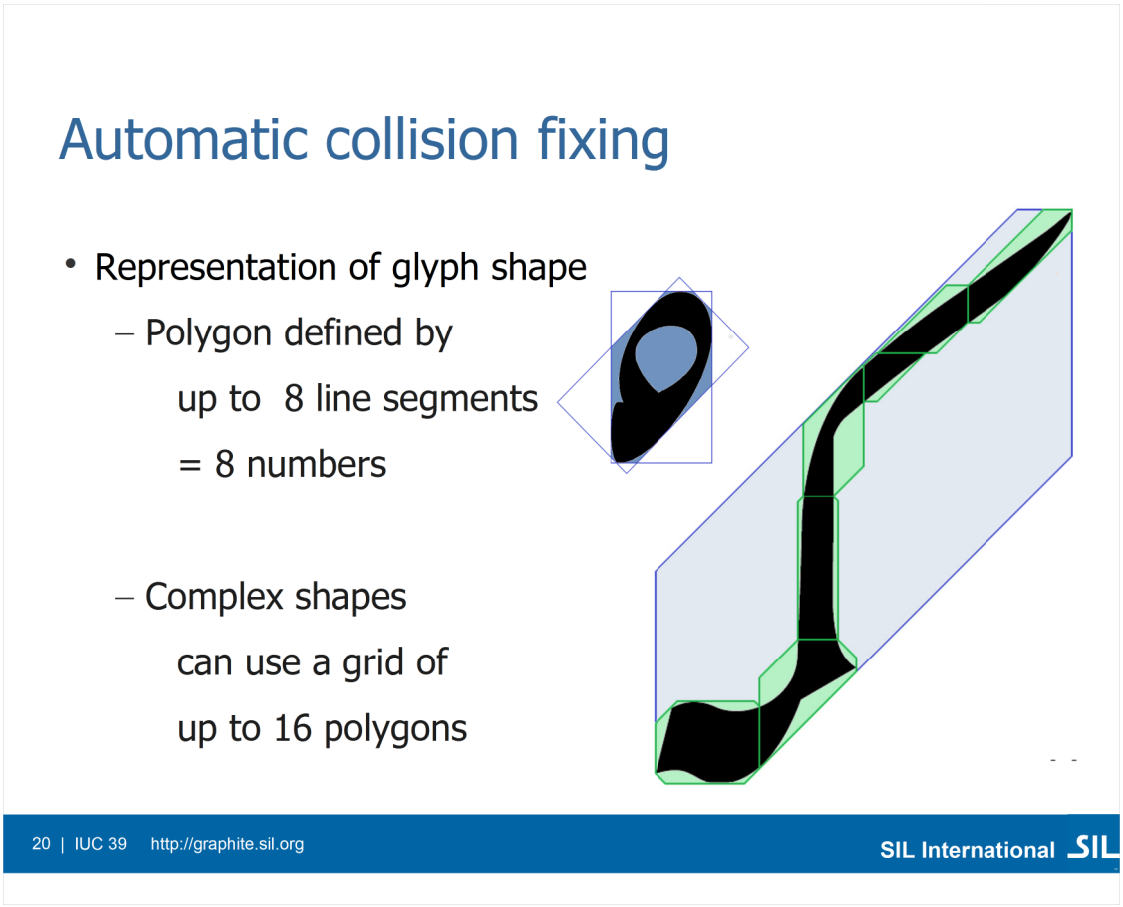
The first question we need to ask when dealing with collisions is: do these two glyphs collide? For that we need to compare the outlines of the two glyphs. If we do this using the actual outline data, it will take forever. So we came up with an approximation to the outline.

A simple bounding box is too inaccurate, but is there something that is both cheap and more accurate? The approach we use is to represent each glyph by a polygon created by two overlapping rectangles, one at a 45-degree angle. The overlapping area of the two rectangles serves as an approximation to the glyph.

Each polygon contains (at most) eight sides represented by eight lines, two horizontal, two vertical, and four diagonal. The vertical and horizontal lines, as expected, are represented by their intersections on the corresponding axes. Diagonal lines at 45 degrees also can be defined by a constant – the sum or difference between the x and y values on the line. Thus each of the eight lines can be represented by a single number.

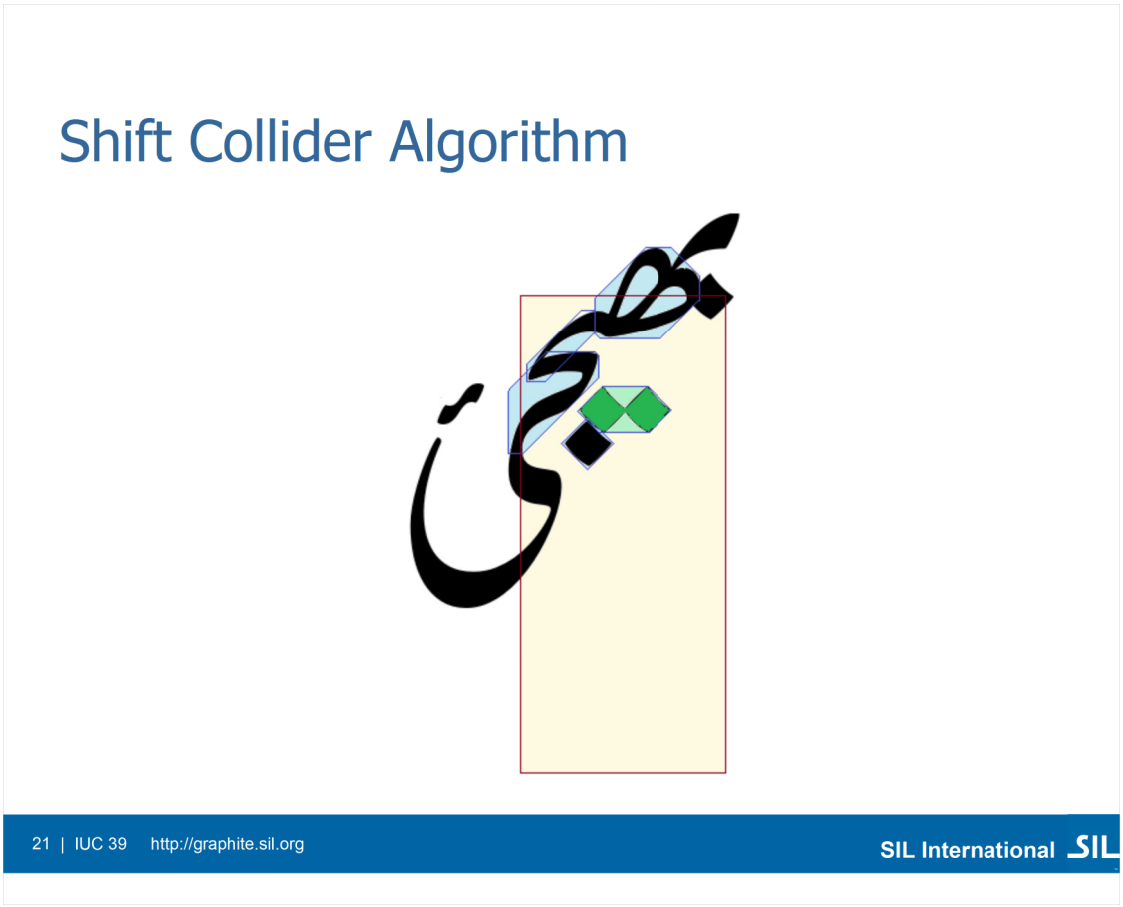
The image above shows the polygon for one form of the medial feh.

These polygons are calculated for each glyph by the Graphite compiler based on the glyphs' curves, and are stored in one of the special Graphite tables in the font. There is no need for a user to define the outline; the compiler does the work.



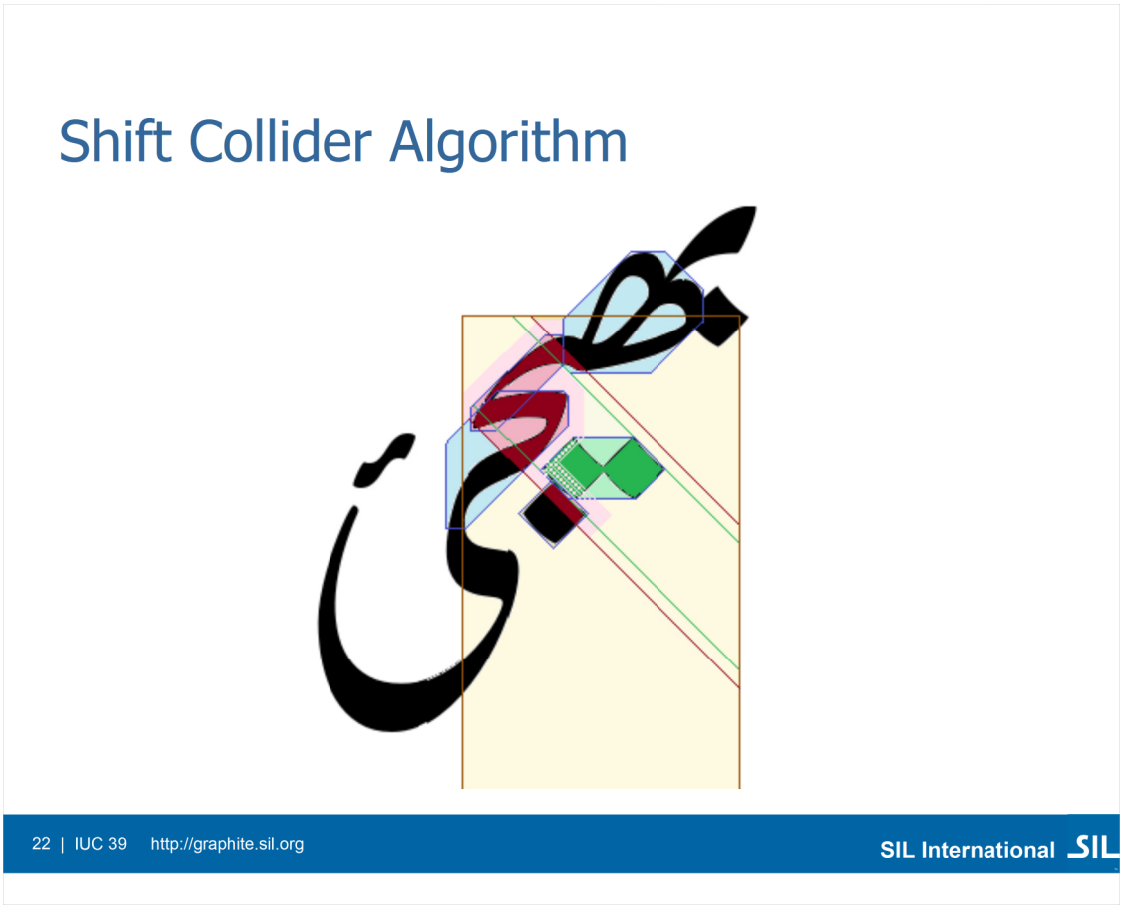
A single polygon is adequate to represent many simple glyphs. But it is not adequate for complex curves, where, for instance, a nuqta might need to be positioned inside the concave area of the curve. For these cases, a set of smaller polygons can be defined to more closely approximate the curve. We define a grid of 4x4 sub-rectangles which allows up to 16 sub-polygons.

The second graphic above shows one form of the letter kaf. Notice that the single polygon, shown by the light blue background, is not a very close approximation to the glyph shape. Instead, a set of seven polygons, shown by the green outlines, is used.



The image shows an example of text with some collisions that needs to be fixed. Let's focus on the green double-nuqta, which collides with single nuqta and is also too close to the jeem form above and to the left. The blue shapes shown around some of the neighboring glyphs are the polygons that are used to approximate these glyphs.

The yellow area is the “movement limit rectangle” - the area within which the double-nuqta is permitted to move.



The shift collider algorithm tries to figure out where some legal areas are for the colliding glyph to move to. It examines four axes – horizontal, vertical, and diagonal at 45 degrees. One of the diagonal axes is shown above. The diagonal lines show the areas where the nuqta could move into if it were to move along this axis.

The red areas indicate places that are illegal because they are taken up by other glyphs. Notice that each glyph has some margin around it, indicated by pink. Some of the pink and red area extends across the green double-nuqta, indicating that it is indeed in collision.

GDL Parameters

- Movement limit rectangle
- Margin
- Kerning
- Overlaps
- Exclusion areas – e.g., above kaf upper stroke
- Flow of sequences
  - diagonal flow of nuqtas
- Weighting

Another requirement for collision fixing is a way for the font developer to specify the parameters of the behaviour. The GDL language contains a set of mechanisms to allow this.

•The movement limit rectangle indicates how far the glyph can be moved from its original position in four directions.

•The margin indicates how much space should be preserved or created between this glyph and others.

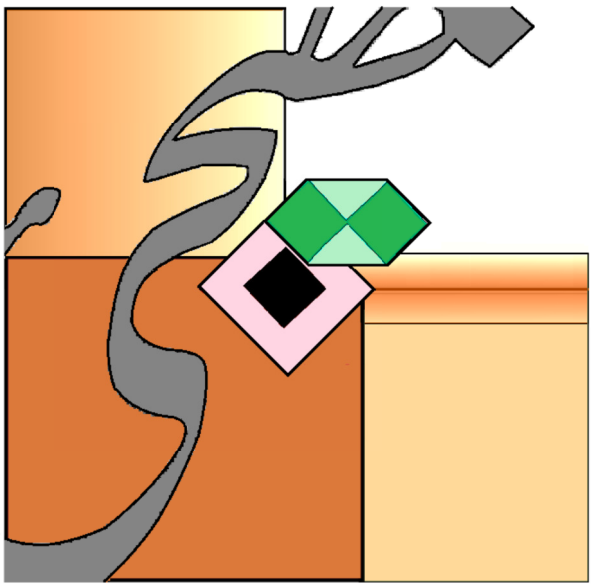
•The programmer can indicate how much kerning is allowed after a base glyph. The same mechanism that kerns apart to fix collisions can be used to kern together in order to create the diagonal overlap of sequences typical of the Nastaliq style.

•Exclusion areas allow a glyph to restrict space around it beyond the actual glyph curve approximation. For instance, kaf forms use an exclusion area above their upper strokes to prevent nuqtas from jumping above them.

•There is a sequencing mechanism that encourages nuqtas to position themselves in specific ways relative to other glyphs - for instance, to create a diagonal flow that roughly parallels the diagonal flow of the base glyphs.

•At some point we might also implement a way to substitute alternate (i.e., smaller) glyphs specifically to fix collisions.

Sequence Flow Regions



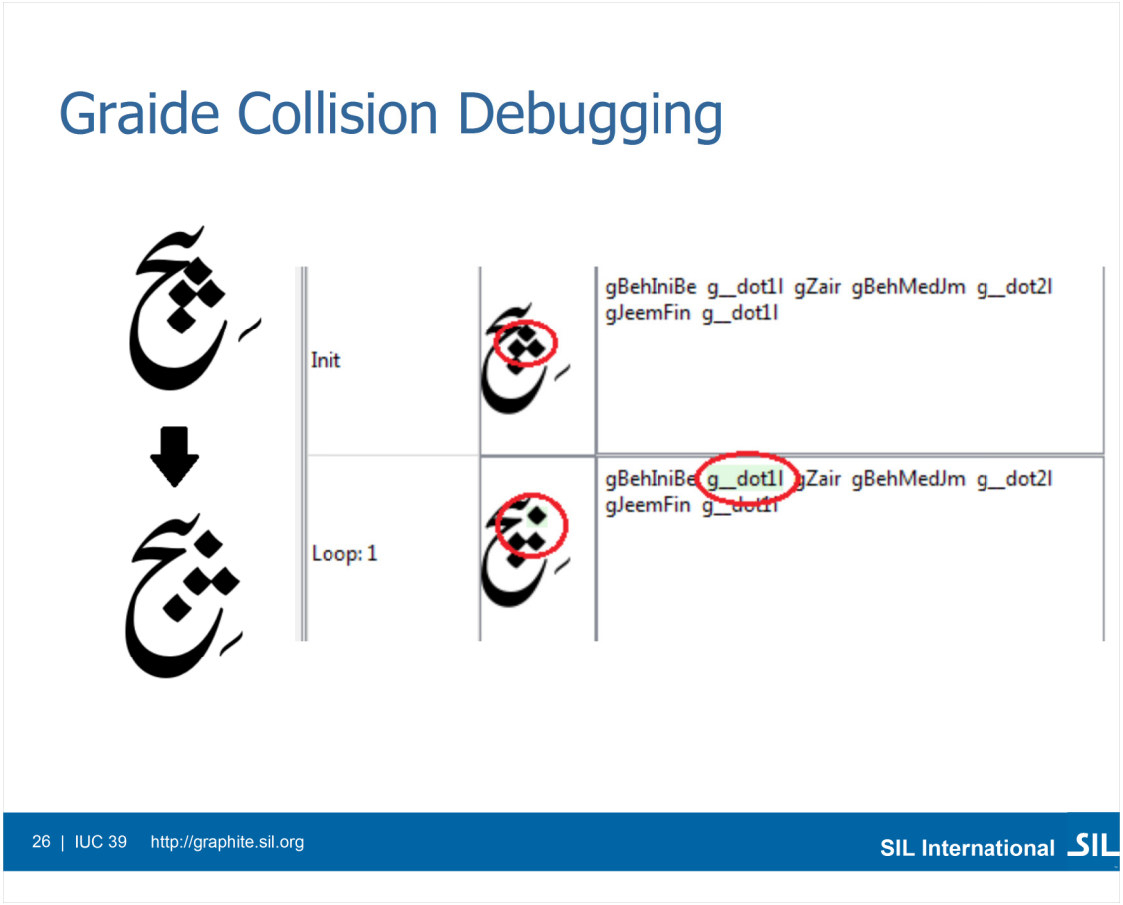
The brown regions show areas that are affected by the sequence flow mechanism. These regions are set up to try to keep the green double-nuqta roughly to the right and above the black single nuqta. The deeper the brown colour, the more undesirable the location and the higher the cost.

Notice the deeper brown area directly to the right of the single nuqta – this region attempts to keep the nuqtas offset slightly in order to enhance readability.

Weighted parameters

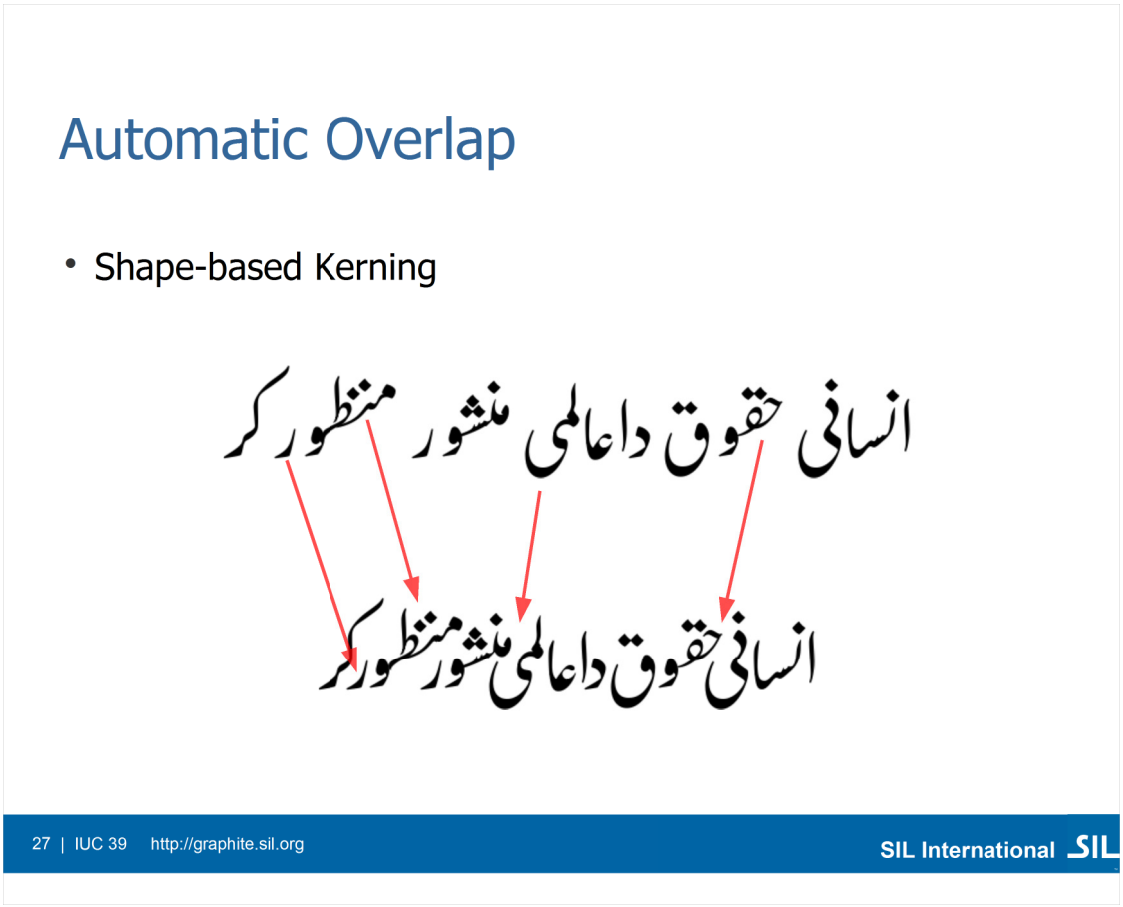
- Distance moved
- Margin encroachment
- Sequence flow

Several of the factors can also have weights associated with them. The GDL programmer can specify weights associated with the amount of distance the glyph is moved from the original position and the encroachment into the specified margin. This allows the algorithm to, for instance, move a glyph closer to a neighboring glyph than the margin would indicate rather than moving the glyph a long way from it's original position. Similarly weights can be associated with the regions that are defined for arranging a sequence of glyphs. For instance, you may prefer to violate the sequence rules rather than move two glyphs closer together.



In addition to showing each rule that fires, the Graide utility allows the user to step through the process that the Graphite algorithm uses to make position adjustments.

One step of the process is shown above. The single nuqta has been moved up and away from the double nuqta that follows it. To the right, the name of the adjusted glyph is highlighted in green.

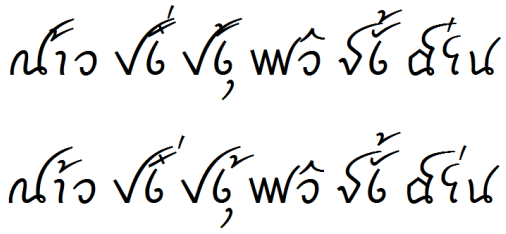


Just as we use the approximate outlines of the glyphs for moving diacritics, we can use those same approximations to aid in kerning that is outline-based and allows for overlapping of the diagonal segments. This is important in Nastaliq, since it gives back the calligraphic feel of the text which has been lost in most computer implementations.

The image above shows first a typical rendering with definitive spacing between the text segments, and below that, the rendering with overlap between segments. The amount of overlap can be controlled by the GDL program.

Automatic collision fixing

- Possible use in other scripts:
  - Naskh-style Arabic
  - Roman kerning
  - Tai Viet



While the automatic collision fixing mechanism is motivated by Nastaliq, we see potential for it to be applied to other styles and scripts as well. For example, the image shows a number of sequences in Tai Viet script that require collision fixing.

Collision fixing

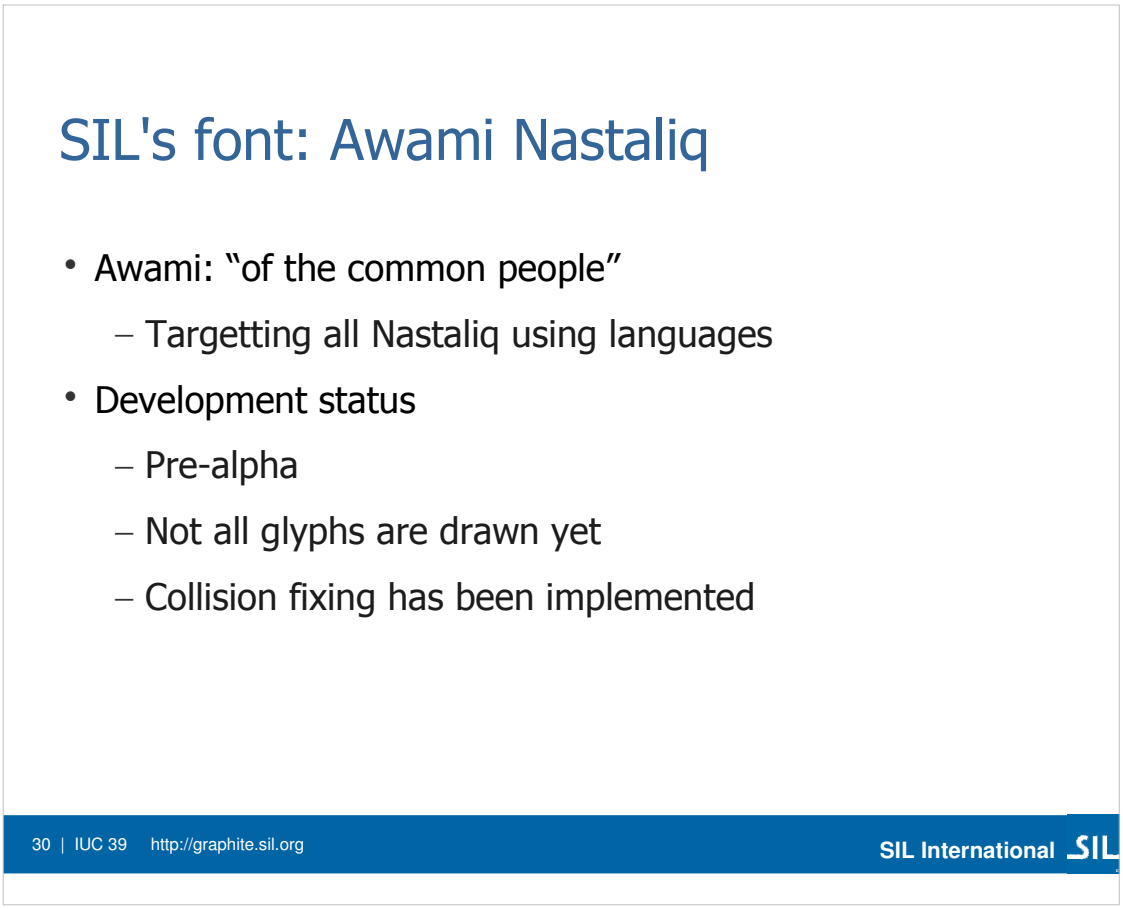
- Fixed forms of the examples we showed earlier:

Here are some examples of the results of the collision avoidance system. Notice that there are no sequence-specific rules here; all the fixes are a result of the basic collision fixing algorithm. This is a real solution to the Nastaliq rendering problems shown earlier. The impact on font development is enormous:

- Addition of extra glyphs does not require addition of many rules to handle nukta collisions.
- Changing glyph shapes does not require retesting of all the collision avoidance test cases.

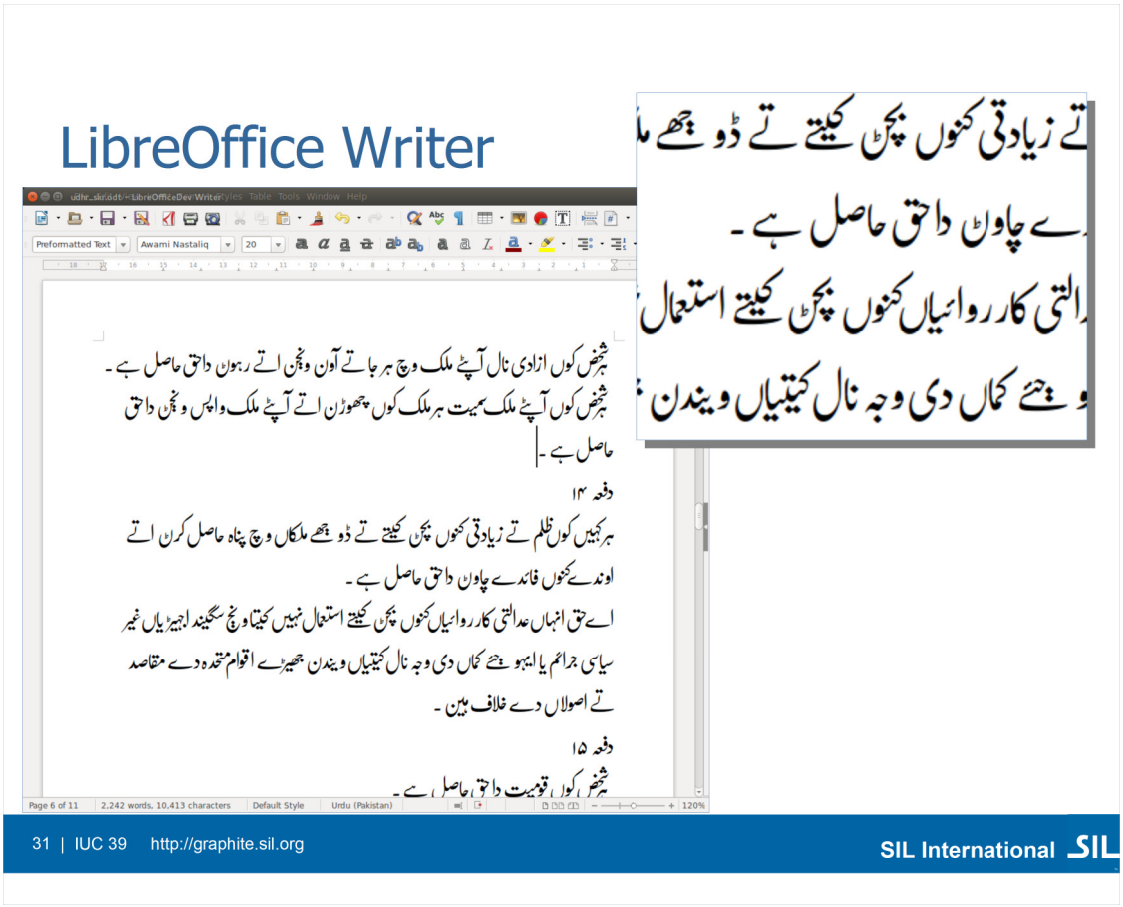
The collision avoidance system does not preclude the need for font testing, but it does provide a large safety net so that even if unexpected sequences occur that have not been tested for, the font will handle them well, if not perfectly.

This is the first time that there is a general font technology that can provide this capability in general applications.

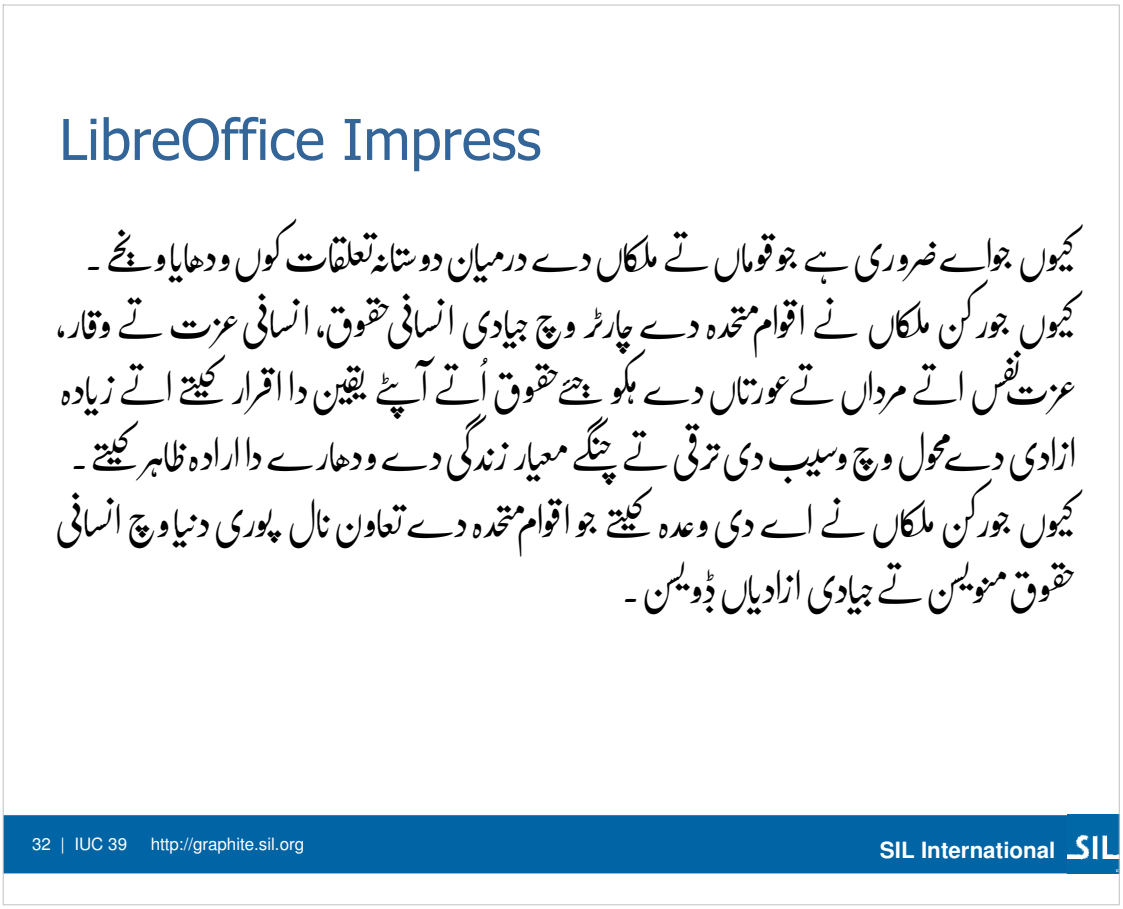


The font SIL is developing is called "Awami Nastaliq." *Awami* is an Urdu word meaning "of the common people," reflecting our intention that this font be able to support a wide variety of lesser-known languages.

The font is currently at pre-alpha stage. About 700 of the roughly 1000 Nastaliq glyphs have been drawn, but we've done enough to implement and test our new collision fixing mechanism.



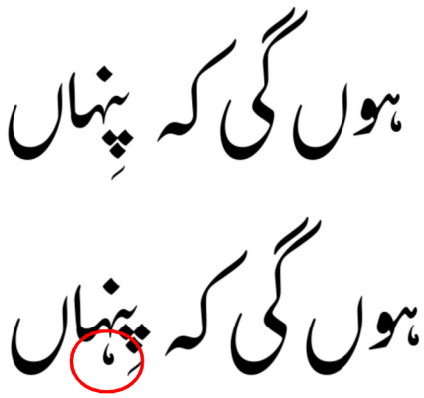
This screenshot shows Saraiki text displayed in LibreOffice Writer.



This slide shows Saraiki text is being displayed directly in this LibreOffice Impress presentation. (Note that we are using an early build of LibreOffice that is not yet officially released.)

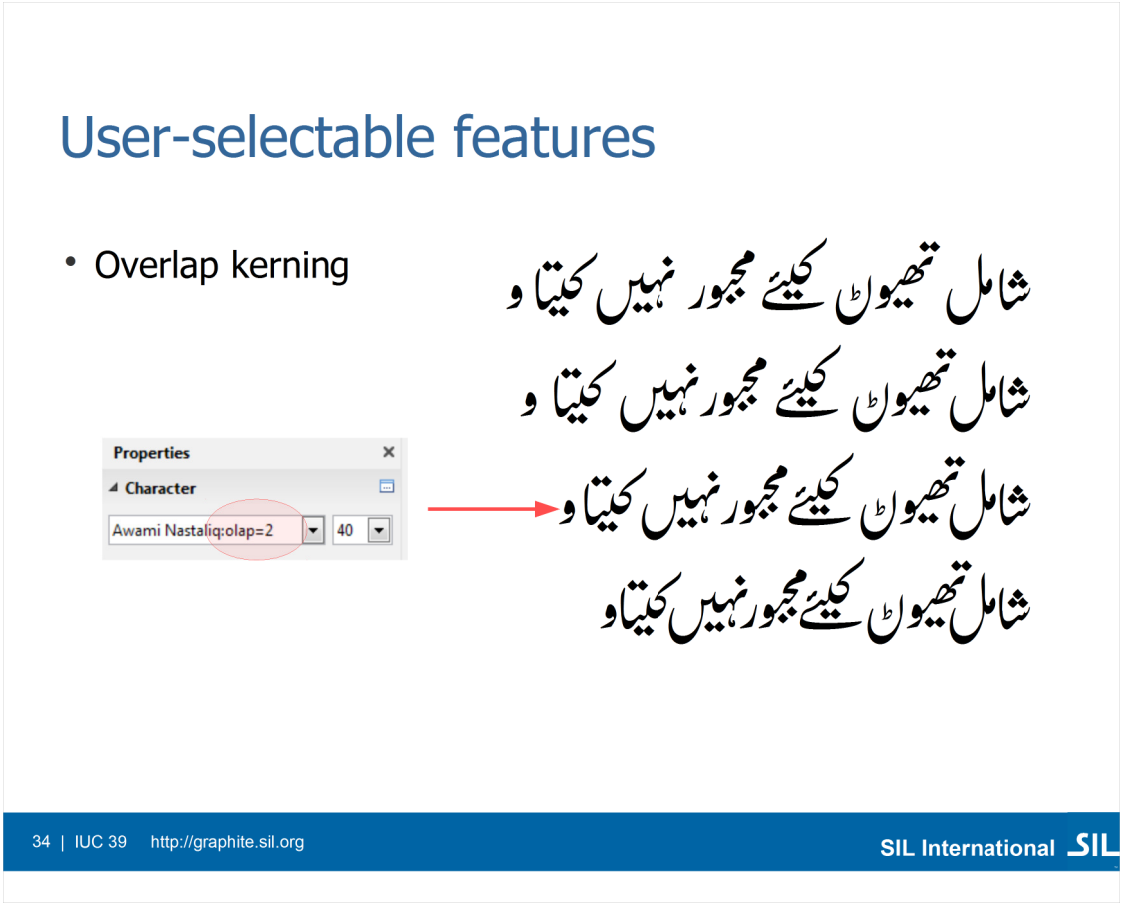
User-selectable features

- Heh-goal medial hook



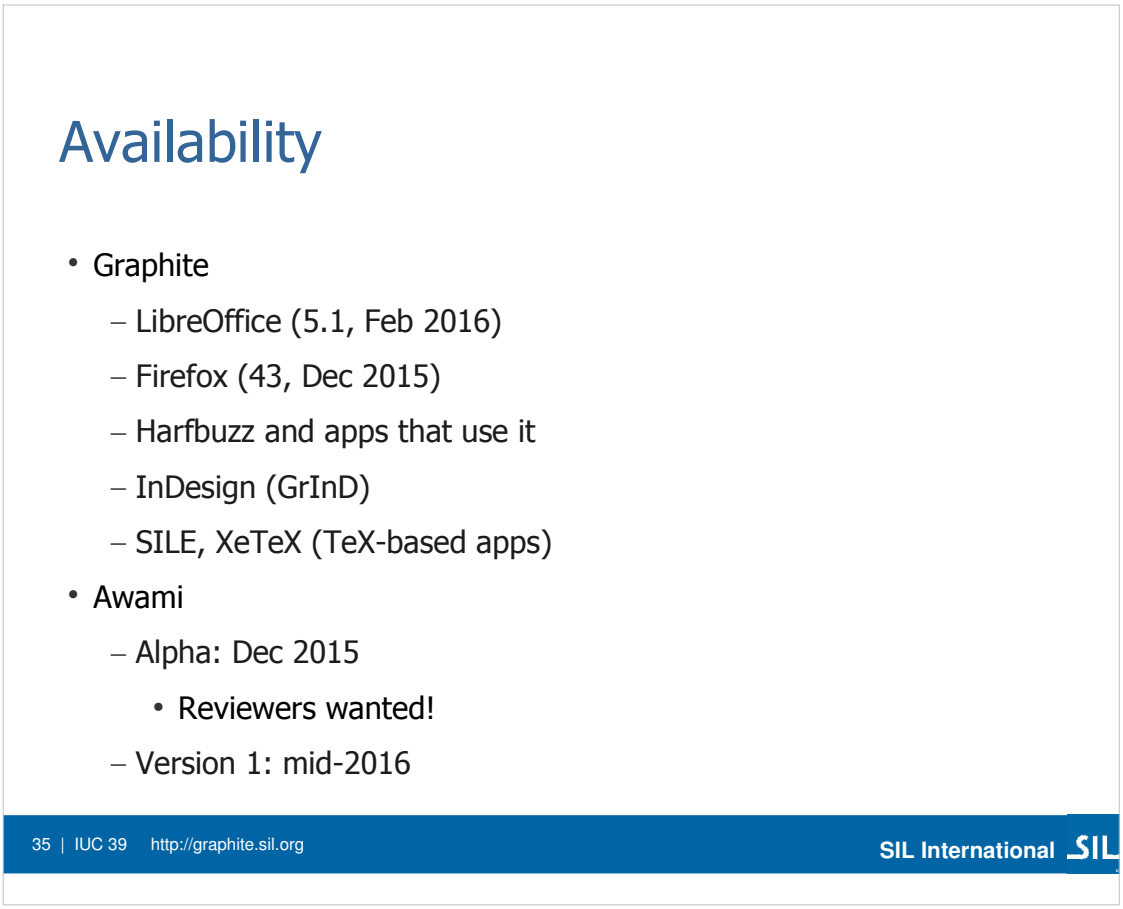
We are adding a number of user-selectable variations in Awami. For instance, some Nastaliq writers omit the hook on the medial heh-goal forms, while others include it. Awami will support either style by means of a user feature.

Notice that the automatic collision fixing mechanism has adjusted the position of the triple-nuqta on the peh to avoid a collision with the added hook, without any particular attention by the programmer.



Another user feature allows control of the tightness or looseness of the overlap kerning. The image above shows four levels of kerning: None, Wide, Medium, and Tight.

Graphite features can be selected in LibreOffice by appending the feature ID and value to the font name, as shown.

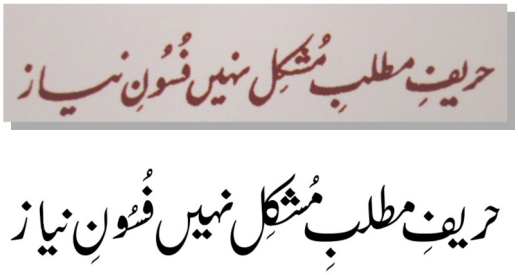


While Graphite is not yet available in some popular applications like Microsoft Office or Chrome, it is available in LibreOffice and Firefox. The GrInD plug-in for InDesign and integration with TeX-based software provides Graphite support for higher-quality typesetting. This makes it possible to have a complete work chain that is Graphite-based.

An alpha of the Awami font will be released in December. Please contact us if you would like to provide feedback. We expect version 1 to be available in mid-2016.

Summary

- Adequate automatic collision avoidance
- Shape based kerning for calligraphic feel
- Fonts are quick and now much smaller



Text samples:

- Top line: taken from “Mirror of Waiting,” by Ghalib; calligraphy from *Wine of Passion: The Urdu Ghazals of Ghalib* by Safaraz K Niazi, p. 69.
- Bottom line: the same text rendered with Awami Nastaliq

Summary

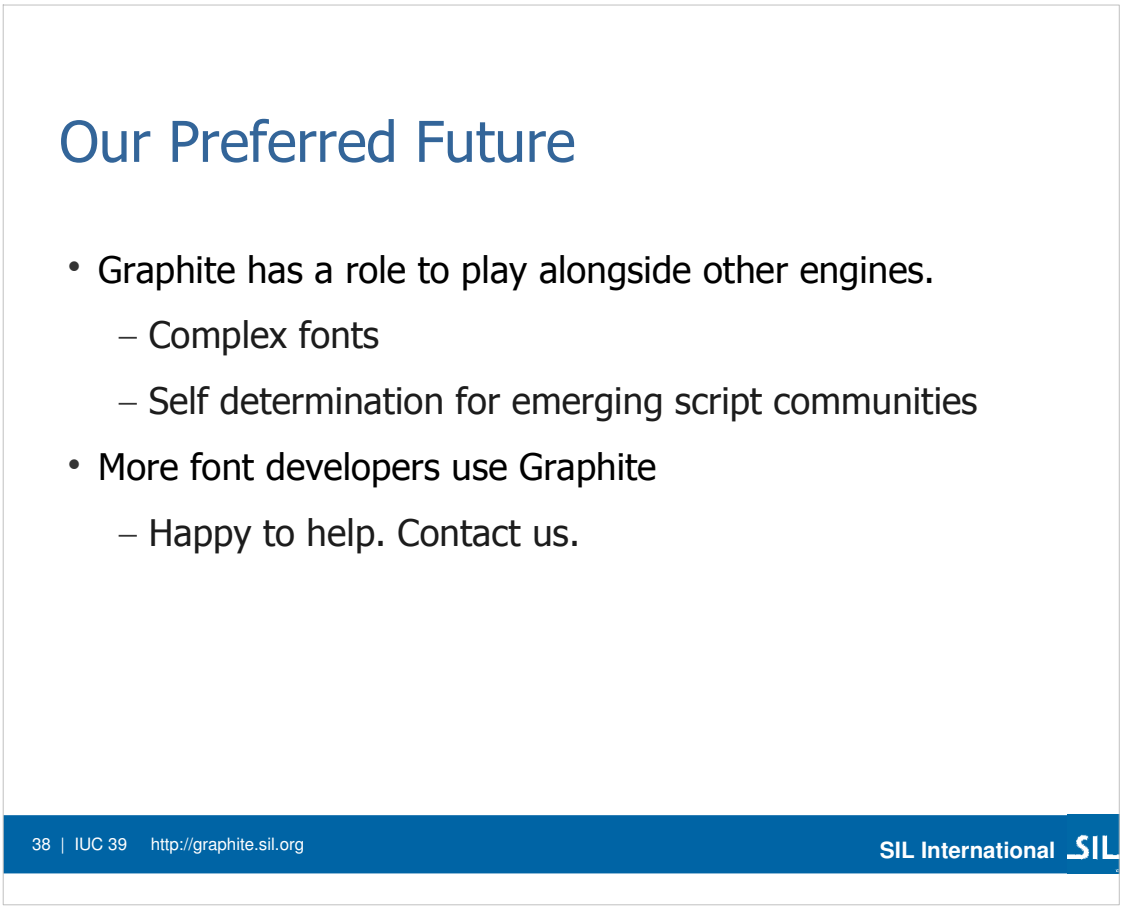
- Consistent cross application behaviour
  - Shaping not dependent on application
- Graphite engine in good shape
  - Careful Design, ongoing fuzzing
  - Open Source (https://github.com/silnrsi/graphite)
- Graphite engine is small (ARM: 83K), fast and flexible
- Graphite engine, compiler, IDE – available today!

One of the struggles we have with OpenType is that not all engines behave in quite the same way. This means that we end up having to test on different platforms and then try to square some circles because Engine A needs us to describe the behaviour one way, while Engine B requires it in another. Currently Graphite is a single engine, and even then the engine behaviour is well defined and not dependent on the vagaries of particular script needs. We are also careful to maintain backward compatibility.

We've done quite a bit of work on the engine lately, and while we have always had security in mind as we develop, we've improved our fuzz testing and so security has improved further as a result.

In spite of the additional collision avoidance mechanism, the engine size hasn't grown much; it is still only 83K on ARM7 (Acorn RISC Machines, the most common phone CPU technology).

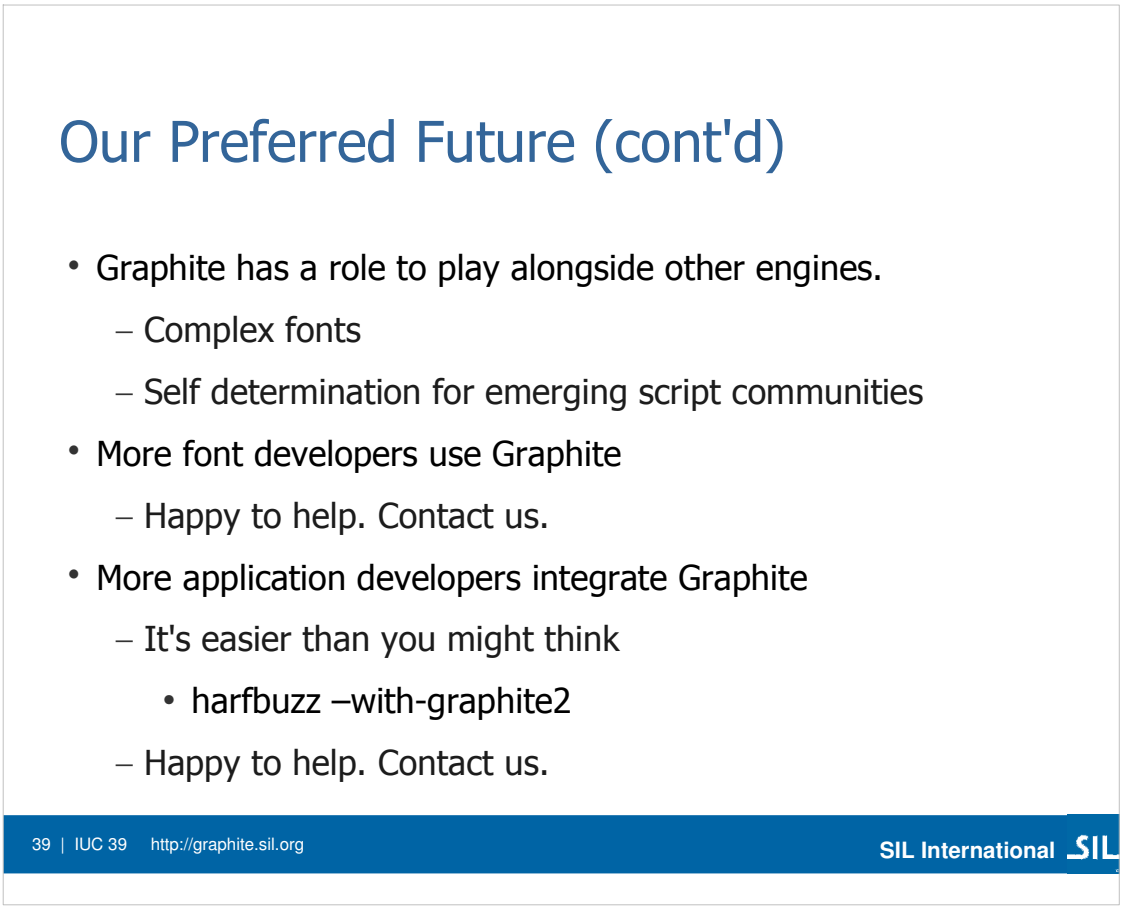
Graphite is not vapourware. We have been using it for a number of years and it has maturity. We commend it as a technology that you can use today to solve real world problems.



What would we like to see happen?

It's not a question of Graphite *or* OpenType, because the two are not incompatible. You can make a font with both OpenType and Graphite tables in, and we generally do. At the application level, you can support both OpenType and Graphite. They are not competing technologies.

Graphite has a particular role in handling complex fonts. Such fonts are *much* quicker to develop in Graphite and run faster as well. The other particular role is in handling fonts for communities whose script is just emerging or where they have needs for sequences considered illegal or not well supported in OpenType.



Bear in mind that supporting a shaping change in OpenType involves all the OpenType engine developers agreeing on the change, and then waiting for that change to be made and then deployed. Even then, few systems backport such changes to existing applications. Graphite allows such problems to be solved in the font itself, allowing communities to solve their own script problems.

Graphite suffers from the network problem: Since OpenType is available in MS Office, OpenType is the de facto solution, so many fonts have OpenType support. Meanwhile, integrating Graphite into an app seems superfluous because there are so few fonts using it; similarly, nobody wants to create Graphite fonts because few applications support it. This Catch-22 makes it very hard to make Graphite a viable solution, which directly impacts those communities who might want to use it to solve their particular community needs to their own time schedules.

It is also an unnecessary obstacle, because integrating Graphite is not hard. For example, Harfbuzz has a –with-graphite2 configure option which means that Harfbuzz-based applications can get Graphite support immediately if they choose to enable Graphite.

Connect with us

- Awami input:
  - http://scripts.sil.org/AwamiNastaliq
- Integration champions needed:
  - Chromium / Android
- Making fonts or adding Graphite to apps:
  - graphite_nrsi@sil.org
  - https://lists.sourceforge.net/lists/listinfo/silgraphite-fonts