Beyond shaping

and towards a general model of OpenType typography

John Hudson, Tiro Typeworks Ltd
International Unicode Conference 39, 28 October 2015

This paper was originally presented at the 39th International Unicode Conference, in Santa Clara CA, on Wednesday 28 October 2015. It was presented as the final contribution to a joint session on 'Universal Shaping', following presentations by Roozbeh Pournader and Behdad Esfahod (Google), and the inventor of the Universal Shaping Engine, Andrew Glass (Microsoft).

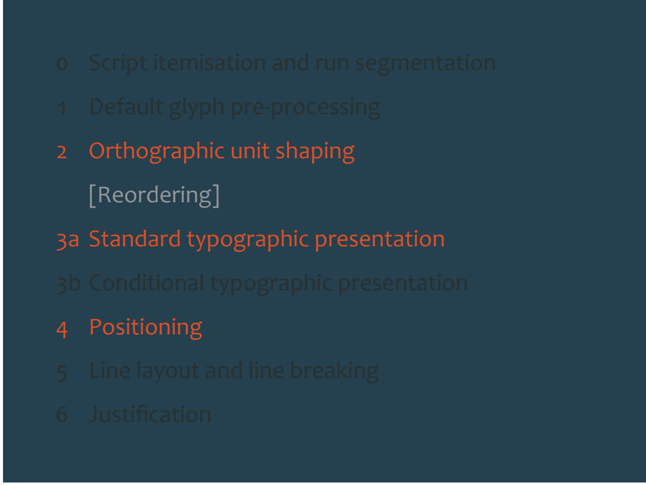
OpenType is a perplexing technology. Seen in some lights, it appears remarkably successful. It may be reasonably claimed to be the most broadly supported font format ever. It is the default format used across major operating systems, both desktop and mobile. It has been adopted, under the name Open Font Format, as an ISO standard. It is the format that almost all of the world's type designers and font manufacturers are regularly making and licensing. It is relied on by millions of users around the world to display their writing systems in appropriate, legible form. And yet…

In another light, OpenType can be seen as, if not a failure, at least a collection of unfulfilled promises. Nearly twenty years after it was first announced, support for OpenType Layout — that which most obviously distinguishes the format from preceding technologies — remains partial, piecemeal, and inconsistent. Some aspects remain completely unimplemented. While the font data format, including its large legacy component, is well-specified, there has never been an implementation specification, nor even documented agreement on best practices, or the sort of defined expectation of outcomes that would enable formal testing of implementations against something other than, well, each other.

At the recent ATypI congress in Brazil, participants put forward suggestions and feature wish lists for OpenType 2.0. Clearly, OpenType is not only an important part of our typographical present, but of the foreseeable future. Will this mean another twenty years of ad hoc and independent implementations, by developers who often fail to see the forest for the trees?

In today’s conference session, my colleagues have focused on a crucial but relatively narrow aspect of text layout, which is standard shaping and display of complex writing systems. This involves glyph processing features and reordering that occur during these three, highlighted phases of OpenType Layout: orthographic unit shaping, standard typographic presentation, and positioning. In the Universal Shaping Engine model, these phases involve substitution of necessary forms to process individual clusters prior to reordering, then, after reordering, further substitutions to refine display of the clusters and their relationship to each other and, lastly, arrangement of the glyphs using varieties of positioning features, including cursive connection, kerning, and mark positioning as appropriate to the writing system and typeface design.

I now want to go ‘beyond shaping’, and talk about what comes before and after these phases, and some of the issues that arise in the other parts of layout. I also want to talk about the whole structure, which I think is something to which not enough attention has been given.



0 Script itemisation and run segmentation

1 Default glyph pre-processing

2 Orthographic unit shaping

[Reordering]

3a Standard typographic presentation

3b Conditional typographic presentation

4 Positioning

5 Line layout and line breaking

6 Justification

OpenType Layout begins with itemisation of a string of characters, and division of that string into runs for subsequent processing, based on Unicode script property. The runs are further segmented according to additional criteria, such as direction, font and size. Script itemisation determines to which of a number of possible layout engines a run will be passed for processing, and at least some of what happens to it when it gets there. So obviously this is a crucial step, along with the cluster analysis that then occurs in the case of scripts that follow the Brahmi model, on which the orthographic unit shaping phase relies.

0 Script itemisation and run segmentation

1 Default glyph pre-processing

2 Orthographic unit shaping

[Reordering]

3a Standard typographic presentation

3b Conditional typographic presentation

4 Positioning

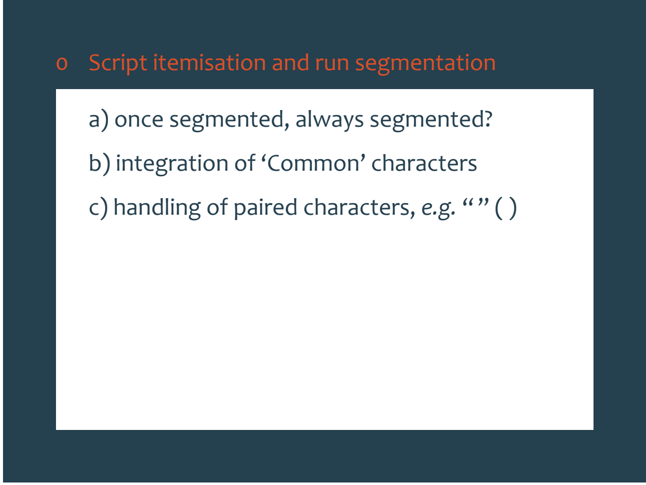
5 Line layout and line breaking

6 Justification

Necessary though this itemisation and segmentation is, once segmented things tend to stay segmented and, as we'll see later, this introduces problems for later aspects of layout, in which the relationship of adjacent glyphs at run boundaries are not resolved.

Other implementation issues arise in this phase, and would benefit from more explicit specification of the correct way to do things. I've had developers tell me that it is 'obvious' how characters with the Unicode script property 'Common' — *e.g.* punctuation — should be rolled into runs with adjacent characters of specific scripts to enable preferred form substitutions or kerning. Yet, over the years, I've seen plenty of inconsistency in how this is done by different software makers, and in different versions of software from the same makers. There does seem, now, to be convergence on a common practice, but it would surely have been helpful to have a formal specification of the expected way to do this from the outset.

There are related issues around display of paired common characters — such as opening and closing quotes or parentheses — enclosing text in multiple scripts, which potentially involve distinctive preferred forms or positioning. What gets precedence? Should paired characters always correspond to each other in form and position, or should they react to the script to which they are individually adjacent? There isn't an obviously correct answer to that question: very likely there will be varying preference among users. The issues are in some respects similar to those that arise in bi-directional layout, but involving decisions about application of glyph processing features in one place based on what has happened in another, and with no standard algorithm that says what is the proper default thing to do.



0 Script itemisation and run segmentation

a) once segmented, always segmented?

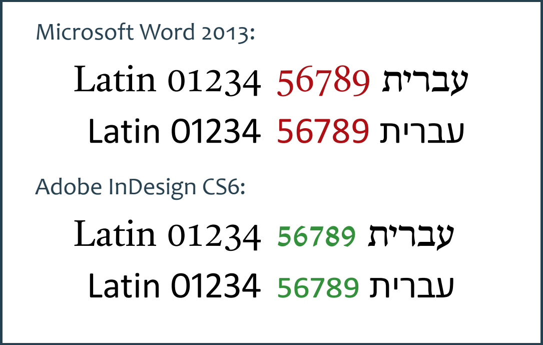
b) integration of 'Common' characters

c) handling of paired characters, *e.g.* “ ” ( )

Finally, there are inconsistencies in the interpretation of the relationship of script and language system tags in fonts, and how these relate to document language tagging. So, for example, if a font contains variant numeral glyphs that align in style or height with Hebrew letters, how are these activated? In this example, the text is tagged as 'Hebrew' in the respective documents, but only InDesign interprets this as meaning that a Localised Forms <locl> feature substitution for the numerals under the font's Hebrew script and language system tag should be applied.

Lacking a clear implementation specification, I can't tell you which of these is technically correct, although it isn't difficult to see which is more useful.

0 Script itemisation and run segmentation



The default glyph pre-processing phase is relatively straightforward. The interaction of layout engine and font begins with the location of the default glyph mapped to each Unicode character in the font cmap table. Glyph pre-processing features perform substitutions on these default glyphs, such as substituting language-specific forms, composing combinations of base and combining mark characters as diacritic glyphs or, conversely, decomposing default glyphs into one or more component elements. What all these features have in common is that they set up the initial sequence of glyphs to be processed by subsequent phases.

The only significant issue regarding this phase is identification of exactly which features should be processed in it and, for the font developer, the order in which these will be applied.

Most of the rest of this conference session has dealt with implementation of the orthographic unit shaping phase, in the context of the Universal Shaping Engine. I won't say much more about it, other than to reiterate that for many writing systems it is dependent on discrete cluster analysis: a further level of segmentation. One of the huge improvements of the Universal Shaping Engine model is that it makes clear that this cluster segmentation is only applicable to this phase, and that once the orthographic forms are arrived at and reordered, processing of subsequent phases should be applied across the whole glyph run and not just at the individual cluster level. This is, of course, essential to enabling contextual interaction between adjacent clusters, the impossibility of which was a major flaw in implementations of previous Indic shaping engines. Unfortunately, a fix to those engines that was agreed last April has, to date, only been implemented in Apple's CoreText engine and the open-source Harfbuzz engine. Microsoft and Adobe's implementation remain unable to support cross-cluster interaction.

0 Script itemisation and run segmentation

1 Default glyph pre-processing

2 Orthographic unit shaping

[Reordering]

3a Standard typographic presentation

3b Conditional typographic presentation

4 Positioning

5 Line layout and line breaking

6 Justification

The purpose of the standard typographic presentation phase is to arrive at an appropriate default visual representation of the encoded text, taking into account the norms of the writing system as well as the conventional text culture with its traditions, aesthetic canons, and reader expectations. In practice, complex scripts are better supported in this regard, through shaping engines, than ostensibly simple scripts such as Latin, because standard typographic presentation forms — ligatures, contextual variants, etc. — are able to piggy-back on the need for orthographic shaping. Where such features are perceived by software developers as optional extras, rather than part of the obligatory representation of text, they have seldom been prioritised.

It should be noted that the standard typographic presentation phase includes both required OpenType Layout features, by which I mean those that must be active at all times, and the aptly names standard features, which are expected to be active by default but might be disabled by the user. Lack of consistency in support for these features, means of course that a common standard and default presentation of text across software remains unrealised.

0 Script itemisation and run segmentation

1 Default glyph pre-processing

2 Orthographic unit shaping

[Reordering]

3a Standard typographic presentation

3b Conditional typographic presentation

4 Positioning

5 Line layout and line breaking

6 Justification

In previous talks and papers, I have referred to the next phase as 'discretionary' typographic presentation, in the sense that the features are off by default but a user might turn them on at his or her discretion. I recently realised, though, that this term might encourage the very thinking that I want to eliminate: the idea that this whole phase is somehow optional, at the discretion of the software maker to ignore. So I am now calling this the conditional typographic presentation phase. The conditions in question may be defined by standards for display of particular kinds of text — for instance 'ruby' notation of Japanese kanji —, or by the particular intention of an individual user regarding the styling of a paragraph, a word, or even a single character.

This is where we can start to consider the forest as well as the individual trees. So far, we've initiated a process, the end goal of which is typography. In its broadest sense, typography is the articulation of text through the design and arrangement of type. In fundamental terms, this means making decisions about how text is presented, so it should stand to reason that a typographic technology is one that enables such decisions to be made. Implementations of OpenType Layout that support only partially the standard typographic presentation phase, and the conditional typographic presentation phase not at all, are not typographic technologies. Their makers have come as far as they think they need to present this or that script and language according to their own understanding of a minimal acceptable level of legibility and orthography, and then walked away from the rest of the task.

When Microsoft and Adobe first announced OpenType, in 1996, the two companies had different priorities: respectively, complex script shaping and typographic design. This found expression in the initial set of layout features registered by each company, and in the priorities adopted in the implementation of OpenType Layout in their software. So, for example, Microsoft registered features and implemented support for Arabic and Indic script shaping, but was slow to implement support for typographic features for European scripts, such as common ligatures and smallcaps, while Adobe registered and supported features for many typographic refinements for European and East Asian scripts, but was slow to enable complex script shaping and even whole categories of OTL lookup types.

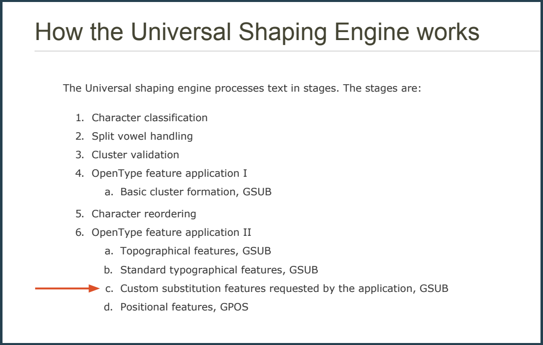
0 Script itemisation and run segmentation
1 Default glyph pre-processing
2 Orthographic unit shaping
[Reordering]
3a Standard typographic presentation
3b Conditional typographic presentation
4 Positioning
5 Line layout and line breaking
6 Justification

Within their respective areas of priority, each company established de facto standards for feature implementation, in the absence of any formal specification of OpenType Layout beyond the font format documentation and supplementary, script-specific shaping specifications produced by Microsoft (these were often written ex post facto, and did not always accurately describe the behaviour of the layout engines that they ostensibly specified). Unfortunately, these implementations tended to perpetuate the initial difference in priorities between script shaping and typography, such that very different levels of typographic sophistication now pertain to different writing systems, affecting the ability of publishers, typographic designers, and font makers to produce consistent quality in their work across multiple scripts and languages.

We are now in a situation where the surely desirable goal of an equitable, high level of typographic richness and control — regardless of script or language —, seems as likely to be stymied by two-decades of legacy behaviour and backwards compatibility concerns as to find eventual fulfillment. Within this situation, the Universal Shaping Engine provides to newly encoded scripts a rapid and well-specified path not just to orthographic shaping but also to typographic control. Andrew Glass' published explanation of how the Universal Shaping Engine works follows a similar structure to my own 'general model'. Note that 'Custom substitution features requested by the application' anticipates conditional typographic presentation: the new shaping engine is ready to receive those decisions made by the user about the appearance and articulation of text.

Do I need to draw attention to the irony that a technology designed to facilitate support of latterly encoded minority and historical scripts — the sort of scripts that might otherwise not get supported at all — provides a higher level of typographic functionality than that afforded to most of the world's major writing systems in much current software?

3b Conditional typographic presentation



How the Universal Shaping Engine works

The Universal shaping engine processes text in stages. The stages are:

1. Character classification
2. Split vowel handling
3. Cluster validation
4. OpenType feature application I
    a. Basic cluster formation, GSUB
5. Character reordering
6. OpenType feature application II
    a. Topographical features, GSUB
    b. Standard typographical features, GSUB
    c. Custom substitution features requested by the application, GSUB
    d. Positional features, GPOS

Most issues around positioning have, over the years, involved limited and piecemeal support. Again, complex scripts have tended to come out ahead, due to the very obvious need for dynamic mark positioning. Some assumptions persist, regarding the kinds of positioning that are expected for a given writing system, so while, for instance, cursive connection lookups are supported in Arabic layout engines, they are not necessarily supported for Latin fonts, even cursively connecting ones. This kind of assumption limits the potential for type designers to invent new styles of font. Again, the Universal Shaping Engine model does much better in this regard, by making every lookup type and feature available to every script that is passed through the engine.

0 Script itemisation and run segmentation

1 Default glyph pre-processing

2 Orthographic unit shaping

[Reordering]

3a Standard typographic presentation

3b Conditional typographic presentation

4 Positioning

5 Line layout and line breaking

6 Justification

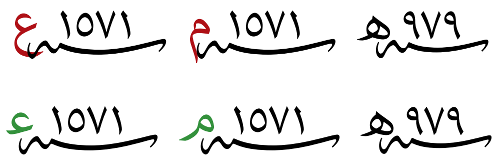
We move now into what may be considered the forward observation post of OpenType typography. What I've discussed so far is stuff that has been implemented, however imperfectly, and which is supported fairly widely, however inconsistently. The outcome is a sequence of glyphs of appropriate form, interaction, and positioning according to the initial itemisation and segmentation, script and language-specific pre-processing, orthographic unit shaping, standard and conditional typographic presentation, and positioning. This is the stage where one might consider the work to be done. In fact what one may have is a sequence of glyphs in which adjacent shapes in different runs may be colliding or otherwise in need of additional refinement, and in which determination and application of line breaking may require extra shaping behaviour.

0	Script itemisation and run segmentation
1	Default glyph pre-processing
2	Orthographic unit shaping
	[Reordering]
3a	Standard typographic presentation
3b	Conditional typographic presentation
4	Positioning
5	Line layout and line breaking
6	Justification

The Arabic date sign and its calendar abbreviation provide a particularly tricky example of the run boundary problem. Here we have an example of year notation for the Islamic Hijri calendar, identified by the Arabic letter *hā',* and the corresponding year in the Gregorian calendar, identified by the letter mīm (from *al-miladi*) in Arabic, and the letter *'ayn* (from *'issawi*) in Urdu. In this font, the standard, descending forms of *mīm* and *'ayn* collide with the tail of the year sign, and would be better replaced by the variant forms shown in green. At present, there is no way to automate this substitution within OpenType Layout, because the change of direction between the left-to-right numerals and the right-to-left letter means that there's no way to express the context. This is where the 'once segmented, always segmented' issue that we identified in the first phase of text processing comes back to bite us. Although this sequence of characters forms a graphical unit, it is processed as discrete, segmented runs determined by the directionality of the characters.

A number of ideas have been put forward regarding ways to address this kind of line layout issue, both specific — *e.g.* treating the Arabic year sign element as a special construct, aided by layout intelligence outside the font — and general, *e.g.* de-segmenting runs at some stage (maybe earlier than this phase) and either applying a common glyph directionality or introducing a mechanism for lookups to specify directionality for processing. As I said, we're now in the forward observation post, looking at the territory to be covered, and figuring out how to proceed.

U+0601 ARABIC SIGN SANAH



Line breaking may introduce its own complexities, of which my favourite example is Uyghur hyphenation. Uyghur is a Turkic language written in a variant of the Perso-Arabic script. Since Turkic languages tend to longer words than are found in Arabic, Uyghur typesetters introduced word breaking and hyphenation, both of which are alien to most other uses of the script. As you can see in the highlighted example the line break between the third and fourth lines, the practice is to retain the shaping of the letters as if they were still connected. There's much about this style of typography that contravenes general practice of Perso-Arabic script grammar — including the original application of that grammar, in the *nasta'līq* style, to the Uyghur language —, but it is what a large modern user community is used to. More interesting, to me at least, is the implication of Uyghur hyphenation practice for the general model of OpenType typography, which is, of course, that elements of Arabic script shaping, which have been dealt with already back in the orthographic unit shaping and standard typographic phases, need also to be apply during the line breaking phase. This suggests that at least some aspects of OpenType glyph substitution and positioning may need to be applied more than once.

In passing, I'll also note that the particular forms of those connecting letters on either side of the hyphenated word break are easily determined in a simplified style such as shown here. In a richer script style, the contextual selection of appropriate form for actually connected letters may differ from the forms to be used at a linebreak, being determined both by what comes before and what comes after.

5 Line layout and line breaking

Uyghur hyphenation

...تۇتۇپ ، يولنىڭ ياقىسىغا تۇرغۇزۇپ
قويغان ئەخلەت ماشىنىسىنىڭ ساندۇ-
قىغاتاشلىماقچى بوپتۇ. بۇنى كۆرگەن ھې-
لىقى يىگىت.

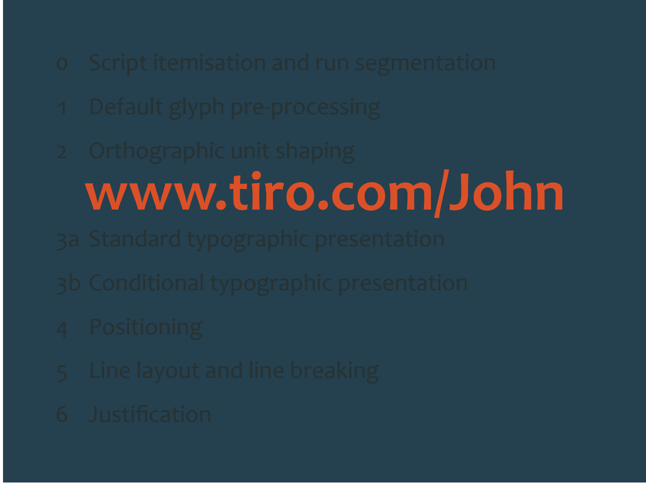
We come at last to justification. We've shaped text, we've applied suitable typographic features, we've positioned the pieces, and we've broken the text into lines to fit the frame (and possible re-applied some aspects of shaping to clean up the lines and the line breaks). Now we want to optically balance the margins by adjusting aspects of those lines to make them all the same length. It may be news to many people that since the mid-90s the OpenType format has included a Justification table (JSTF) that is designed to assist this phase: it is, to my knowledge, entirely unimplemented in line layout software, fonts, or tools for making fonts. The table provides for a prioritised set of methods to be tried during justification to obtain the best results. It is appropriately a font-level mechanism, because the best results in justification are by their nature design dependent, not merely script or language dependent, nor reducible to a crude algorithm of increasing or decreasing word spacing regardless of script and type style. The JSTF table was designed to be able to handle an iterative approach to justification, in which several methods in succession — variant wider or narrower letterforms, extended connections, spacing adjustments — are applied to achieve progressively finer results.

I've heard some developers question whether the JSTF table is implementable, and others express concern about the processing costs of running an iterative approach to justification that may, in turn, require iterative re-application of some aspects of shaping. These are not insignificant concerns, but given just how crude justification of many non-European writing systems is in current software, there is clearly room for improvement. Even if the JSTF table as currently specified isn't the ultimate solution, we should be considering how to fulfill this particular promise of OpenType, twenty years after it was made.

So there you have, in broad structural terms, a general model of OpenType typography. A white paper on categorisation of OTL features within the model, as well as a grab-bag of other presentations and articles to which I've alluded, is available at the URL shown in this slide.

There are, of course, many details that I have omitted or passed over too briefly within each of the phases I have described. But I want to encourage you all to think about the whole, and not just the parts that perhaps pertain most directly to your individual tasks and responsibilities, or to your company's immediate priorities.

Unless we have a sense of the goal — of OpenType typography as a structured, integrated technology for making and applying decisions about the presentation of text — we risk continuing as we have for two decades: treating it as a bucket of arbitrary features to be added to products in ad hoc and piecemeal ways, without regard to consistency, usability, or any overall strategy.



0 Script itemisation and run segmentation

1 Default glyph pre-processing

2 Orthographic unit shaping

www.tiro.com/John

3a Standard typographic presentation

3b Conditional typographic presentation

4 Positioning

5 Line layout and line breaking

6 Justification