Bringing world scripts to LuaTEX: The HarfBuzz experiment

Khaled Hosny

1 HarfBuzz

Unicode includes thousands of characters and hundreds of scripts,[1] but inclusion in Unicode is just the start. Proper support for many of them is a much more involved process. (Figure 1 shows a few examples.)

To aid Unicode, there is a need for *smart* fonts; fonts that are not merely collections of glyphs. TEX's TFM fonts are a kind of smart fonts, as they contain rules for making ligatures based on certain contexts; but meeting the needs of the world scripts requires more than ligatures. These needs lead to the development of several font and layout technologies that can fulfil them.

One of these technologies is OpenType,[2] which is widely supported on major operating systems and applications, making it the de facto standard for Unicode text layout. Others include Apple Advanced Typography[3] (AAT) and Graphite.[4]

The text layout process can be seen as several successive steps. One particularly interesting and rather complex step is called *shaping*, which basically involves taking chunks of characters that have some common properties (such as having the same font, same direction, same script, and same language) and converting them into positioned font glyphs. A piece of software that does this is often called a *shaper*.

In OpenType the knowledge needed for proper shaping of a given script is split between the shapers and the fonts; a script-specific shaper has embedded knowledge about a certain script (or group of related scripts), and the fonts provide font-specific data that complements the knowledge embedded in the shaper.

One of the widely used OpenType implementations is HarfBuzz,[5] which identifies itself as a text shaping library. Although HarfBuzz was initially only an OpenType shaping engine, it now supports AAT and Graphite as well. HarfBuzz is an open source library under active development.

2 LuaTEX

LuaTEX is an extended TEX engine with Lua as an embedded scripting language. LuaTEX also supports Unicode text, among other things. The LuaTEX philosophy is that it provides solutions, not answers, so it does not come with an extended text layout engine, and instead provides hooks to its internals so that its users (or macro packages) can extend it as they see fit.

Arabic	عربي	Bengali	ক্রকি
Devanagari	क्रकि	Gujarati	ક્રકિ
Gurmukhi	ਕ੍ਰਕਿ	Kannada	ಕ್ರಕಿ
Malayalam	ക്രക്കി	Myanmar	တော်ကြီး
Oriya	କ୍ରକି	Sinhala	ක්‍රෙ
Tamil	கோ	Telugu	క్రరిు

Figure 1: Sample texts from some of the world's scripts.

خط الرقعة الجديد: رقعة عارف

خط الرقعة الجديد: رقعة عارف

Figure 2: Aref Ruqaa font as rendered with HarfBuzz integration (above) and luaotfload (below).

While this is a worthwhile goal, in practice writing a text layout engine for the Unicode age is a complex and demanding task, and takes many person-years to develop. On top of that, it is a moving target as Unicode keeps adding more scripts (both living and dead) and font technologies keep evolving as the problems at hand become better understood,[6] and it takes quite some effort to remain on top of this.

This has led to having only one mature and feature-full text layout engine for LuaTEX, written purely in Lua by the ConTEXt team. This engine is made available to LATEX users via the luaotfload package as well. It is a fast and flexible engine, and has many interesting features. But it falls short of supporting all scripts in Unicode. Even for the scripts it supports, some fonts might not work well when they utilize rarely used parts of OpenType that the ConTEXt team might not have had a chance to test (figure 2).

HarfBuzz, on the other hand, is considerably more widely used, tested, and exposed to all sorts of

[1] Unicode 12.0 has a total of 137,929 characters and 150 scripts: unicode.org/versions/Unicode12.0.0

[2] docs.microsoft.com/en-us/typography/opentype

[3] developer.apple.com/fonts/TrueType-Reference-Manual/RM06/Chap6AATIntro.html

[4] graphite.sil.org

[5] harfbuzz.github.io

[6] For example, OpenType had an initial model for shaping Indic scripts, which was later found to be inadequate and a new model was developed (keeping the old model for backward compatibility). Later, a new, more extensible model, called Universal Shaping Engine, was developed to handle many Indic and non-Indic scripts.

tricky and complex fonts.[7] It also has a larger team of dedicated developers that have spent many years enhancing and fine-tuning it.[8]

3 Integrating HarfBuzz with LuaTEX

Integrating HarfBuzz with LuaTEX would bring the benefits of HarfBuzz without giving up the capabilities of LuaTEX. There have been several attempts to do this, including the one that is going to be discussed here in some detail.

The basic idea is rather simple: get the text from LuaTEX, shape it with HarfBuzz, and then feed the result back to LuaTEX.

LuaTEX provides hooks, called *callbacks*, that allow modifying its internals and adding code to be executed when LuaTEX is about to do certain tasks.

HarfBuzz provides a C API and there are several ways to call such an API from LuaTEX; each has its pros and cons:

FFI Originally part of LuaJIT, but available now for regular LuaTEX as well, this allows binding C APIs without the need for writing separate bindings in C. However, it requires duplicating the C headers of the library inside the Lua code. Using FFI in LuaTEX requires using the less-safe `--shell-escape` command-line option.

Loadable Lua C modules Written in C, this uses the Lua C API for interacting with the Lua interpreter; it can link to any library with a C API (either dynamically or statically). It can be dynamically loaded at runtime like any Lua module (e.g. using `require`), but it is not as well supported by LuaTEX on all platforms.

Built-in Lua C modules Instead of dynamically loading Lua C modules at runtime, they can be statically linked into the LuaTEX binary, making them work on all platforms. This however, requires either building a new independent engine based on LuaTEX, or convincing the LuaTEX team to include the new module.

Making a loadable Lua C module was chosen for this experiment, utilizing the existing luaharfbuzz project[9] and extending it as needed to expose additional HarfBuzz APIs.

In addition to the luaharfbuzz module, additional Lua code is needed to extract input from LuaTEX, feed it to HarfBuzz and back to LuaTEX, and do any conversion and processing necessary for both input and output to be in the form that both ends can utilize.

4 Loading fonts

LuaTEX's `define_font` callback allows for overriding the internal handling of the `\font` primitive, which can be used to extend the syntax as well as to load additional font formats beyond what LuaTEX natively supports. Although HarfBuzz is not specifically a font loading library, it provides APIs to get enough information for LuaTEX to use the font.

HarfBuzz's font loading functions support only fonts using the SFNT container format,[10] which basically means it supports OpenType fonts (and by extension TrueType fonts, which are a subset of OpenType). It is possible to support other formats by using the FreeType library[11] to load the fonts instead of HarfBuzz's own font loading functions, but for the sake of simplicity and to avoid depending on another library this was not attempted. In practice (outside of the TEX world, that is) all new fonts are essentially SFNT fonts of some sort.

Font data in SFNT containers are organized into different *tables*. Each table serves a specific purpose (or several purposes) and has a tag that identifies it. For example, the `name` table contains various font names, the `cmap` table maps Unicode characters to font glyphs, and so on.

The LuaTEX manual describes the structure that should be returned by this callback. Basically, some information about the font is needed, plus some information about the glyphs in the font.

4.1 Loading font-wide data

Loading most font-wide data (font names, format, etc.) is straightforward since HarfBuzz has APIs that expose such information.

There are two main OpenType font flavours based on what kind of Bézier curves is used to describe glyph shapes in the font; cubic Bézier curves (also called PostScript curves, as these are the kind of curves used in PostScript fonts), and quadratic curves (also called TrueType curves, as these are the kind of curves used in TrueType fonts). The main difference between the two is the glyph shapes table: cubic curves use the `CFF` table, while quadratics use the `glyf` and `loca` tables.

LuaTEX wants the format to be indicated in the font structure returned by the callback (possibly to decide which glyph shapes table to look for, though

[7] HarfBuzz is used by Mozilla Firefox, Google Chrome, ChromeOS, GNOME, LibreOffice, Android, some versions of Adobe products and many open source libraries, not to mention XETEX; in all, it has billions of users.

[8] HarfBuzz started its life around the year 2000 as the OpenType layout code of the FreeType 1 library, long before it was named HarfBuzz.

[9] `github.com/ufyTeX/luaharfbuzz` by Deepak Jois

[10] `en.wikipedia.org/wiki/SFNT`

[11] `freetype.org`

Figure 3: Random Unicode emojis using Noto Color Emoji font which embeds bitmap PNGs instead of outline glyphs. (Grayscaled in the print version.)

that seems redundant as it can easily detect it itself). It is easy to determine the format by checking which tables are present in the font, so that is not an issue. However, there is now a different OpenType flavour that does not include any of these tables and instead uses different tables that embed colored glyph bitmaps (used mainly for colored emoji; a few are shown in figure 3). LuaTEX does not support embedding such fonts in PDF files. To work around this, such fonts are identified during font loading, and during shaping (see below) this is detected, and the PNG bitmaps for font glyphs are extracted and embedded as graphics in the document, avoiding the need for including the font itself in the PDF.

4.2 Loading glyph data

Other than font-wide data, LuaTEX also wants to know some information about the font glyphs. Ideally, such information should be queried only when the glyphs are actually being used, and OpenType tables are carefully structured to allow for fast loading of any needed glyph information on demand without having to parse the whole font (which can be time consuming, especially for large CJK fonts containing tens of thousands of glyphs). However, the way things are structured in LuaTEX requires loading all basic glyph information up front. Thankfully, HarfBuzz's font loading is fast enough that the slowness implicit in loading all required glyph information is not a critical problem.

Although it is theoretically possible not to load any glyphs initially, but wait until after shaping and update the font with information about glyphs that were actually used, this would be very slow as it would happen thousands of times, requiring recreating the LuaTEX font each time a new glyph is used. Also, in my experiments, sometimes glyphs would fail to show in the final document if they weren't loaded when the font was initially created.

LuaTEX requires all glyphs in the font to have a corresponding character (the font structure seems to make no distinction between characters and glyphs), but not all glyphs in the font are mapped to Unicode characters (some glyphs are only used after glyph substitutions, e.g. ligatures and small caps). To work around this, pseudo-Unicode code points are assigned to each glyph; LuaTEX characters are full 32-bit numbers, but Unicode is limited to 21-bit values (no code point larger than this will ever be used by Unicode, for compatibility reasons), so the trick is to use the glyph index in the font and then prefix it by 0x110000 (the maximum possible Unicode code point + 1), thus keeping LuaTEX happy by having a character assigned to each glyph. This way any glyph in the font can be accessed by LuaTEX, while not clashing with any valid code point. The downside of this is that any LuaTEX message that tries to print font characters (like overfull box messages) will show meaningless bytes instead. LuaTEX has callbacks that could be used to potentially fix this.

Some font-wide data like ascent, descent and cap-height do not have corresponding entries in LuaTEX fonts and LuaTEX checks instead for the metrics of hard-coded set of characters to derive this information from. To work around this, and since we don't provide any entries for real characters, we can create fake entries for these characters using the font-wide data instead of the actual character metrics.

Math fonts seem to be tricky as some of the information LuaTEX requires is not exposed by HarfBuzz in a way that can easily be used at font loading time. For example, HarfBuzz has an API to get the math kerning between a pair of glyphs at given positions, but LuaTEX wants the raw math kerning data from the font to do the calculation itself. Handling this properly would require changes to either HarfBuzz or LuaTEX.

5 Shaping

For shaping there are basically two problems to solve: converting LuaTEX's text representation into something that can be fed to HarfBuzz, and converting HarfBuzz output to a form that can be given back to LuaTEX.

5.1 Converting LuaTEX nodes to text strings

LuaTEX's default text layout can be overridden with the `pre_linebreak_filter` and `hpack_filter` callbacks. They are called right before LuaTEX is ready to break lines into a paragraph, which is just the right moment to shape the text.

By the time the callbacks are called, LuaTEX has converted its input into a list of *nodes*. Nodes represent different items of the LuaTEX input. Some represent characters/glyphs, some represent glue, while others represent kerning, etc.; there are also

modes for non-textual material like graphics and PDF literals.

HarfBuzz, on the other hand, takes as input strings of Unicode characters, in the form of UTF-8, UTF-16 or UTF-32 text strings, or an array of numbers representing Unicode code points.

Converting character nodes is straightforward; the characters they represent are inserted into the text string. Glue nodes are converted to SPACE (U+0020), and discretionary hyphenation nodes are converted to SOFT HYPHEN (U+00AD). Any other node is converted to OBJECT REPLACEMENT CHARACTER (U+FFFC), which serves as a placeholder that does not usually interact with other characters during shaping, but its presence helps with later converting the HarfBuzz output to LuaTEX nodes.

Now the text is almost ready to be fed to HarfBuzz, but not quite: first it needs to be "itemized". HarfBuzz takes as input a contiguous run of characters that use the same font and have the same Unicode script, text direction, and language.

Font itemization is grouping together any contiguous run of character nodes that use the same font, along with any intervening non-character nodes (so that glue nodes, for example, are shaped with the text they belong to).

The same goes for Unicode script itemization, except that this depends on the *Unicode Character Database*,[12] which collects many Unicode character properties, including their scripts (HarfBuzz has an API to access these properties). Some characters don't have an explicit script property, though. Some characters have the script property *Inherit* and these, as you might guess, inherit the script of the preceding character (they are usually combining marks, like accents). Others have the script property *Common*, and they take the script of the surrounding text (they are usually characters that do not belong to a specific script, like common punctuation characters). Unicode Standard Annex #24 describes a heuristic[13] to handle common characters which suggests special handling of paired characters (e.g. parentheses) so that matching ones get assigned the same script.

Text direction itemization requires first applying the *Unicode Bidirectional Algorithm*,[14] but this was out of scope for this experiment, so users are expected to mark right-to-left segments of the text manually using LuaTEX's direction primitives, and the code uses this to determine the direction of the text.

[12] unicode.org/ucd
[13] unicode.org/reports/tr24/#Common
[14] unicode.org/reports/tr9

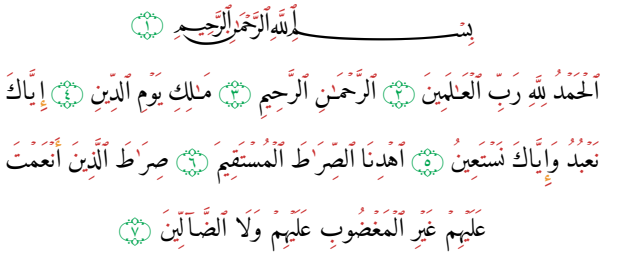

Figure 4: Text using Amiri Quran Colored font which uses colored glyph layers to make a distinction between the consonantal text and the later developments of the Arabic script. Black for the base consonants (they just use the text color), dark red for diacritical dots and vowel marks, golden yellow for *hamzah*, and pale green for non-textual elements like the "circled" *āyah* numbers. (The print version is grayscaled.)

5.2 Shaping with HarfBuzz

After feeding the input text to HarfBuzz and getting back output, some post-processing is needed.

Some OpenType flavours contain only bitmaps for glyphs (in the `CBDT` table[15]), not outlines; LuaTEX doesn't know how to embed such fonts in PDF files. These fonts are detected during font loading, and after shaping, the PNG data of such glyphs is extracted using HarfBuzz, then saved to temporary files and finally embedded as graphics in LuaTEX's node list (it would be better to skip the temporary files step, but there wasn't any obvious way to do this in LuaTEX). This way the font can be used with LuaTEX without having to actually embed it in the PDF output.

There are also layered color fonts (see figure 4), where the font contains, in addition to regular outline glyphs, a table (`COLR`[16]) that maps some glyphs to layers (composed of other glyphs) and color indices, and another table (`CPAL`[17]) that specifies the colors for each color index. Since LuaTEX doesn't keep these tables in the font when embedding it into the PDF file (and even if it did, PDF viewers and other PDF workflows are unlikely to handle them), instead the glyphs are decomposed into layers using the relevant HarfBuzz API and the corresponding colors are added using the regular PDF mechanisms for coloring text. (Color transparency is not handled, though, as it requires support from macro packages to manage PDF resources.)

[15] docs.microsoft.com/en-us/typography/opentype/spec/cbdt
[16] docs.microsoft.com/en-us/typography/opentype/spec/colr
[17] docs.microsoft.com/en-us/typography/opentype/spec/cpal

5.3 Converting HarfBuzz glyphs to LuaTEX nodes

HarfBuzz outputs positioned glyphs. Output glyph information includes things such as the glyph index in the font and the index of the character it corresponds to in the input string (called *cluster* by HarfBuzz). Glyph positions tell how a given glyph is positioned relative to the previous one, in both X and Y directions (called *offset* by HarfBuzz), as well as how much the line should advance after this glyph in both directions (called *advance* by HarfBuzz, but unlike offsets, only one direction is active at a time, so for horizontal layout the Y advance will always be zero, and for vertical layout the X advance will be zero).

To feed HarfBuzz output back into LuaTEX, a new node list based on the original needs to be synthesized. Using the HarfBuzz *cluster* of each output glyph to identify the node from the original list that this glyph belongs to, we can re-use it in the new list, thus preserving any LuaTEX attributes and properties of the original node.

Character nodes The original node is turned into a glyph node, using the glyph index + 0x110000 as its character (see font loading section above for explanation). If more than one glyph belongs to this node, each gets copied as needed and inserted into the node list, so that all the glyphs inherit the properties of the original node. If the advance width of the glyph is different from the font width of the glyph, a `kern` node is also inserted (after the glyph for left-to-right text, and before it for right-to-left text).

Glue nodes The advance width of the output glyph is used to set the natural width of the glue. This way fonts can have OpenType rules that change the width of the space (e.g. some fonts use a narrower space for Arabic text than for Latin, some fonts kern the space when followed by certain glyphs, and so on).

Discretionary hyphenation nodes The existing pre-line breaking, post-line breaking and replacement node lists[18] of the original node need to be shaped as well. Special handling is needed when characters around a discretionary hyphen form a ligature; when no line breaking happens at that discretionary hyphen then the ligature needs to be kept intact, but when line breaking does happen the text should be shaped as if a real hyphen had been there from the start.

LuaTEX handles this with a `replacement` node list which contains the nodes that should appear if no line breaking happens, and a `pre` node list that contain what comes before a line break, and `post` for what comes after it. Since ligatures can't just be cut into parts, the text needs to be shaped two times: once with the whole text without a hyphen, and once with the text split into two parts and a hyphen inserted at the end of the first part. It would be very inefficient to reshape the whole paragraph in this manner, and it would also be impractical to store full paragraphs in `replacement`, `pre`, and `post` node lists.

One solution is to reshape just the ligature, but sometimes the shaping output can be different based on the surrounding characters, so cutting the ligature out and shaping it all by itself can produce the wrong result. Fortunately, HarfBuzz has a flag attached to output glyphs that says whether breaking the text before this glyph and shaping each part separately would give the same output or not. We use this flag to find the smallest part of the text that is safe to reshape separately from the rest of the paragraph, starting from the discretionary hyphen, and re-shape only that part. (See figure 5.)

[18] See LuaTEX manual for detailed explanation of these.

office coffee HAVANA
of-
fice
cof-
fee
HA-
VANA

Figure 5: Ligatures and kerning are formed correctly around discretionary hyphens, when no line breaking happens, and correctly broken at line breaks.

Characters that are not supported by the font are ignored by TEX (no output is shown in the typeset document), and by default only a message is printed in the log file. This is a bit unfortunate as it can be easily missed. With HarfBuzz, unsupported characters return glyph index zero (often named as the `.notdef` glyph), which is usually a box glyph and sometimes has an X inside it to mark unsupported characters. The code will thus insert this glyph into the node list, and since this will effectively disable the missing character messages that LuaTEX outputs, the code emulates LuaTEX behaviour and outputs such messages itself. One side effect of using glyph zero is that even though the character is not shown, the text is preserved in the PDF file and can be searched or copied.

5.4 Handling text extraction from PDF

To extract text from a PDF file (e.g. copying or searching), the PDF viewer needs to know how to reverse map glyphs back to Unicode characters. The simplest way to do this is to set the mapping in the font's `/ToUnicode` dictionary, which can handle one-to-one and one-to-many glyph to character mappings (i.e. simple glyphs and ligatures).

Getting one-to-one glyph to character mappings can be partially done at font loading time by reversing the font's `cmap` table. This, however, covers only glyphs that are mapped directly from Unicode characters. In OpenType, not all glyphs are mapped this way, for example, small cap glyphs are not mapped directly from Unicode characters as they are only activated when a certain font feature is on (the characters are first mapped to regular lowercase glyphs, then a *small caps* feature maps those to small cap glyphs), and detecting what characters they came from can happen only after shaping.[19] Because of this, there is still a need to modify the fonts after shaping each part of the text, to update the `ToUnicode` values for each glyph, and for large documents this is rather slow.

Furthermore, `/ToUnicode` can't handle all cases. With HarfBuzz there can be glyph-to-character relationships that are any of one-to-one, one-to-many, many-to-one and many-to-many. With `/ToUnicode` the first two can be handled, but the last two can't. Also, the `/ToUnicode` mapping is required to be unique for each glyph; the same glyph can't be used for different Unicode characters, but that is a possibility in OpenType and other modern font formats.

Fortunately there is another, more general, mechanism in PDF; the `/ActualText` spans, which can enclose any number of glyphs and represent any number of characters (not all PDF viewers support them, though, but we can't help that).

After shaping, HarfBuzz *clusters* are used to group glyphs that belong to one or more characters and that information is stored in the node list. Then, after line breaking, `/ActualText` spans are added for any group that can't be represented in the `/ToUnicode` dictionary. This is done after line breaking (in the `post_linebreak_filter` callback) since there are many restrictions on the kind of nodes that can appear in the node lists of discretionary hyphenation nodes.

[19] The alternative would be to decode the font features and parse them, which requires a substantial effort and would still not handle all cases since features can do different things for different scripts and languages, and there might be more than one way to arrive at the same glyph.

6 Conclusion

Integrating HarfBuzz with LuaTEX is possible and can bring many benefits to LuaTEX and enable more users to enjoy its capabilities. There are some technical issues to solve and rough edges to round, but nothing that would substantially prevent such integration.

The code described here was made possible thanks to generous support from the TUG development fund (`tug.org/tc/devfund`). The code and the required luaharfbuzz module are available at:

`github.com/khaledhosny/harf`
`github.com/ufyTeX/luaharfbuzz`

◇ Khaled Hosny
`github.com/khaledhosny`