International Text Layout & Typography: The Big And Future Picture

by Edward H. Trager

Introduction

The purpose of this presentation is to provide a comprehensive review of aspects of international text layout and typography which software engineers, font developers, linguists, translators and other interested stakeholders in the Free/Libre Open Source (FLOSS) community should bear in mind as we engineer the people's operating systems of the future. The author believes that the efforts of diverse groups must become unified around designing a unified text layout and rendering pipeline for such systems.

Directionality and Block Progression

Before we begin writing text, we need to consider the *direction* (also called *inline progression*) and *block progression* of that text. In the text model of the World Wide Web Consortium (W3C) standard for Cascading Style Sheets version 2 (CSS2), the *direction* property of horizontal lines of text is defined as either *left-to-right*, or *right-to-left*. Of course the problem with the CSS2 model (and existing user agents) is that vertical scripts are left out in the cold.

The text model for CSS3 finally addresses the fact that a large number of people in the world like to write and read their text in vertical columns. At the 27th International Unicode Conference, Elika J. Etemad presented a paper entitled *Robust Vertical Text Layout*[1] in which she outlined extensions to the CSS text model which provide a straightforward and comprehensive solution for typesetting scripts within the context of -- and without breaking -- the existing Unicode bidirectional algorithm (BIDI) and CSS layout models.

Readers are encouraged to read Etemad's paper as I will provide only a brief summary here with a few examples to illustrate some of the layout possibilities that one encounters in the real world. Note that I am describing a model -- not CSS3 itself -- as the model is applicable beyond CSS3-compliant user agents.

First, the *inline-progression* property has a total of four values:

- *left-to-right (ltr) --e.g. Latin text*
- *right-to-left (rtl) --e.g. Arabic text*
- *top-to-bottom (ttb) --e.g. Traditional Chinese & Japanese text*
- *bottom-to-top (btt) --e.g. Runic text*

Secondly, the *block-progression* property which describes how lines of text are stacked next to one another, has three values:

- *top-to-bottom (ttb) -- e.g., Latin text*
- *right-to-left (rtl) -- e.g., Traditional Chinese & Japanese text*
- *left-to-right (ltr) -- e.g., Traditional Mongolian text*

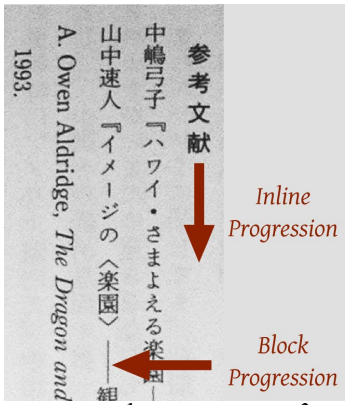

Figure 1.Modern Japanese is often typeset from top to bottom in vertical columns which progress from right to left.

Scripts are generally constrained to a subset of the possible combination of values for these two properties. For example, Latin, Greek, and Cyrillic are normally written with *inline-progression=left-to-right* and *block-progression=top-to-bottom*. The combination of the two properties is called the *writing-mode*. Some scripts are more flexible than others. For example, nowadays Chinese and Japanese often are written with the same writing mode as Latin, Greek, and Cyrillic. However, Chinese and Japanese are also often written with *inline-progression=top-to-bottom* and *block-progression=right-to-left (ttb-ltr, figure 1)*.

While text layout engines should support all of the normal writing modes that a script may assume, unusual cases can and do occur in real life with surprising frequency, especially when addressing the special needs of laying out multilingual text in dictionaries, charts, tables, and so on. Text layout engines therefore need to be designed to support *all* of the possibilities. Let's take a look at a few examples below.

In *figure 2* an excerpt from a Mongolian-Japanese dictionary is shown. Traditional Mongolian is set vertically with block progression from left to right. However because of the Japanese definitions, this dictionary is set with block progression going from right to left instead -- normal for Japanese but atypical for Mongolian. Notice that the Latin phonetics and Tibetan script are rotated 90 degrees clockwise to accommodate the top-to-bottom text progression.

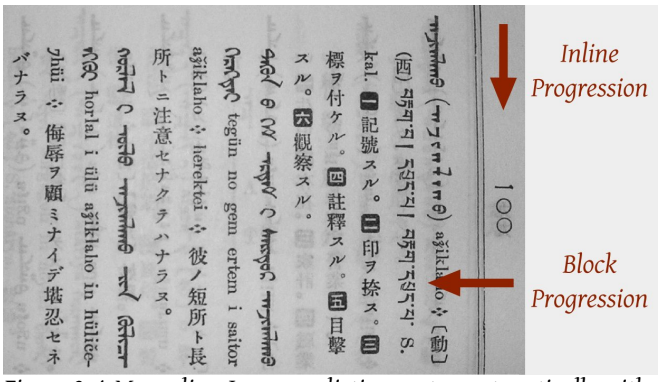

Figure 2. A Mongolian-Japanese dictionary typeset vertically with block progression from right to left.

One more example should suffice to show the variability that can occur in the real world. In *figure 3* an excerpt from a Uyghur-Chinese-Russian dictionary is shown. The Uyghur is written in Arabic script and typeset from right-to-left. Block progression is from top-to-bottom. Interestingly, the Chinese definitions are also set horizontally from right-to-left. Chinese can be typeset from right to left, and in fact it isn't that unusual to see titles in Chinese set from right to left (as shown in *figure 5)*. However, it is a bit unusual to see multiple lines of Chinese set horizontally from right to left in a block or paragraph of text with top-to-bottom progression. Nevertheless, in this case it is seen to work quite well.

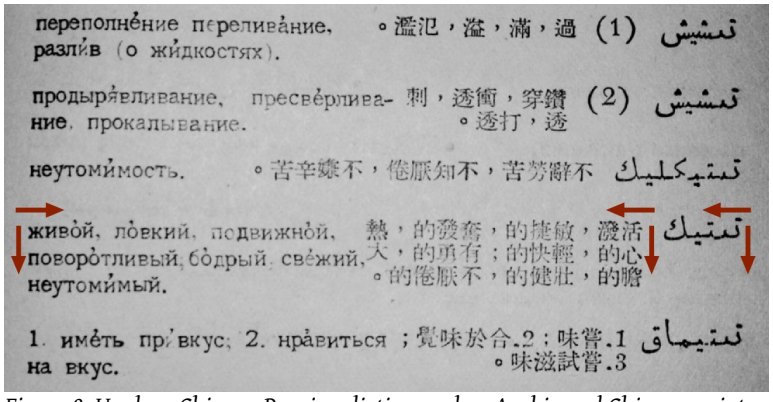

Figure 3. Uyghur-Chinese-Russian dictionary has Arabic and Chinese script both typeset from right-to-left.

Text Directionality in Titles

Figure 4. Titles on book spines are set using inline progression that may differ from the norm for paragraphs of text. As there is not just one right answer, different cultures have settled on different conventions.

Titles on book spines may be set using inline directionality that differs from the norm used to set paragraphs of text. There is usually not just one right answer, and as a result different cultures have arrived at different accepted norms. For example, the French often rotate text 90 degrees counter clockwise resulting in a *bottom-to-top* progression, while Americans and Thais prefer a clockwise rotation resulting in *top-to-bottom* progression. Chinese can just be typeset vertically from *top-to-bottom* (*figure 4*).

We don't normally think of Chinese as a *right-to-left* script, but in Taiwan and Hong Kong it is not uncommon to layout the title of an article or book horizontally from *right-to-left* -- but this is only done when the body of the text is set vertically with *right-to-left* block progression. In this way the inline progression of the title matches the block progression of the text (*figure 5*).

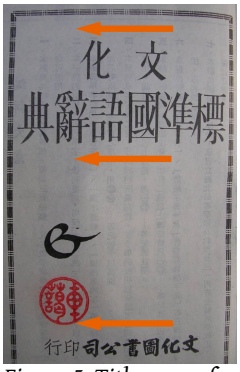

Figure 5. Title page of a Chinese dictionary from Taiwan. The title page is set from right-to-left matching the block progression of the vertically-set contents.

However the Japanese don't use this convention. The Japanese print plenty of books with vertical text and *right-to-left* block progression, but the titles and headers, if set horizontally, have *left-to-right* inline progression. The Japanese preference may be due to the presence of the hiragana and katakana syllabaries in their script. Chinese, having only hanzi (漢 字 , *kanji*) is less directionally constrained.

Right-to-left horizontal titles and headers seem to have almost disappeared in mainland China now that horizontal *left-to-right* text layouts are used almost exclusively in books and other publications.

As numerous and even unexpected combinations of inline and block progression values occur in the real world, a well-designed text layout engine should make it easy for the user to layout text in all of the ways we have seen ... and more (*as discussed below*).

Mirror Writing

Latin, Greek, and Cyrillic are not normally typeset from *right-to-left*. However, every edition of Lewis Carroll's children's classic *Through the Looking Glass* has required that the first stanza of the poem, *The Jabberwocky* be printed as if viewed in a mirror (*figure 6*).

I would therefore argue that a well-designed text layout engine should automatically interpret a *right-to-left* directive on a horizontal layout of Latin and similar *left-to-right* scripts as a request to produce a mirrored image of a *left-to-right* layout. The request will be used sparingly, to be sure, but is not difficult to implement.

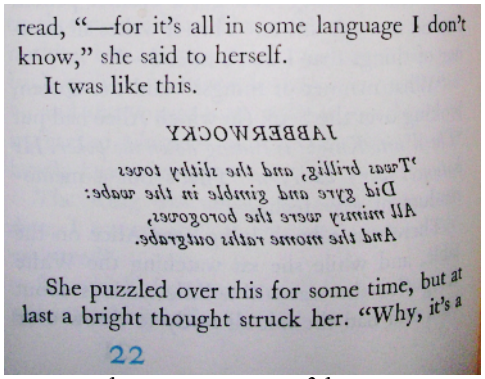
read, "—for it's all in some language I don't know," she said to herself.
It was like this.
JABBERWOCKY
'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogoves,
And the mome raths outgrabe.
She puzzled over this for some time, but at last a bright thought struck her. "Why, it's a
22

Figure 6. The opening stanza of the Jabberwocky from Lewis Carroll's Through the Looking Glass set in mirrored type.

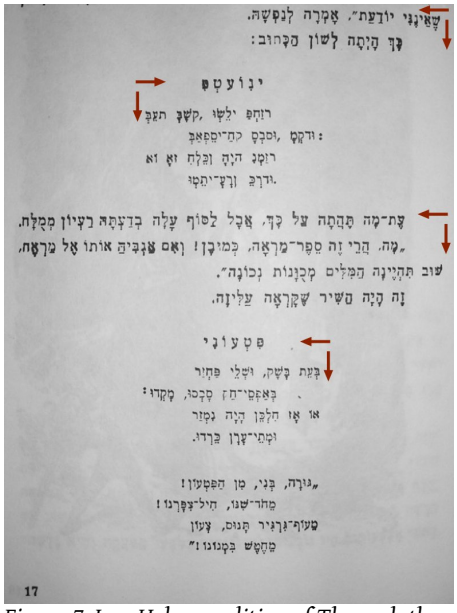

Figure 7. In a Hebrew edition of Through the Looking Glass, the first stanza of the Jabberwocky is indeed set from left-to-right, but the letters are not mirrored correctly, undoubtedly increasing Alizah's confusion ...

And, just to be pedantically complete, let's not forget about editions of Lewis Carroll's classic in languages that are written from right to left:

In *Alizah be-erets ha-mar'ah : va-asher mats'ah sham* published in Tel Aviv in 1979, the first stanza of The Jabberwocky is indeed set from left to right (*figure 7*). But, unfortunately, the letters themselves are not mirrored as they should be. Any intelligent third-grader can see that this is wrong! All the more reason for the FLOSS community to get it right ...

Bustrophedon

Bustrophedon can be considered a special case which, however, could be produced trivially by a layout engine that supported mirrored text: a calling application would merely have to alter the inline progression from *ltr* to *rtl* on a line-by-line basis when laying out a paragraph of text.

But, of course, you say, no one today uses bustrophedon. It's been out of vogue for the last 3000 years! Well, dear reader, you have never been wrong before, but this time you are wrong. Apparently fashions come and go, and sometimes they come back again. So it is interesting to note that a GPL'ed bustrophedon text reader already exists.[2] And you can even embed it in *mutt* to read all your email bustrophedonically. Perhaps this provides sufficient justification to support bustrophedonic progression in the next generation FLOSS text layout engine ...

Supporting Mongolian

FLOSS systems do not yet support Mongolian. There are a number of problems. Traditional Mongolian is written in vertical columns from *top-to-bottom* with *left-to-right* block progression. So the first problem is that FLOSS systems do not yet support vertical text. Fortunately, on this front the situation is changing rapidly. Maciej Katafiasz (Mathrick) recently implemented vertical text layout for Japanese (*Figure 8*).[3] This is a very important achievement. Adding support for left-to-right block progression, if not already available, should prove trivial.

The second problem is that FLOSS text layout engines do not yet support shaping of Mongolian text. Fixing this will require working with knowledgeable speakers of Mongolian who understand the shaping rules.

A third problem is that existing Mongolian fonts are designed with the limitations of horizontal text layout in mind and therefore have glyphs rotated 90° counter-clockwise (*Figure 9*). Fonts with rotated glyphs are the wrong answer. The right answer is to fix the technology so that fonts with rotated glyphs become unnecessary.

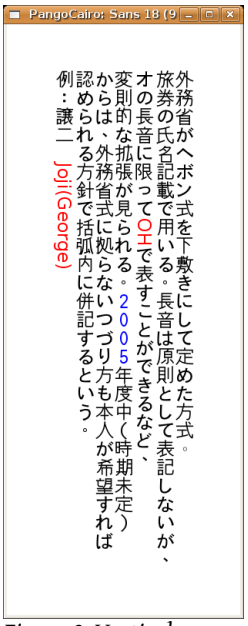

Figure 8. Vertical Japanese text rendered by Pango. With Japanese & Chinese vertical text now supported, Mongolian should be the next target.

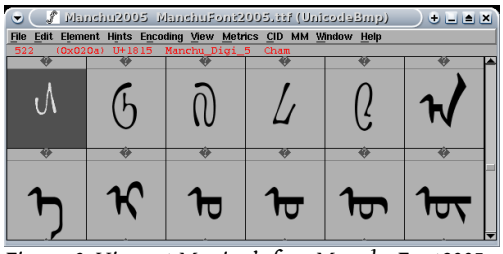

Figure 9. Vincent Magiya's free ManchuFont2005 OpenType font, like other existing Mongolian fonts, has glyphs rotated 90 counterclockwise. This is not the right model for FLOSS systems to follow.

As Pango now supports vertical text layout, the FLOSS community should now seize the opportunity to implement proper support for traditional Mongolian script. The process should begin by encouraging font developers like Vincent Magiya, the developer of the free ManchuFont2005 OpenType font[4], to create an *unrotated* Mongolian font that could be used to develop and test Mongolian support in Pango and the FLOSS layout engine of the future. Perhaps it will be possible to create a script to assist in "unrotating" glyphs from existing fonts and converting horizontal advance metrics to their vertical equivalents as a way to speed up the conversion process.

We can easily envision how a little collaboration among the relevant stakeholders could quickly yield support for Mongolian in Linux and related FLOSS systems. This would be another big win for the Open Source development model and the community of FLOSS users.

Providing Access to Advanced Typographical Features

Laying out text for many scripts requires access to advanced typographical features such as glyph substitution and precise glyph positioning. Baseline adjustments are commonly required on a per-script basis when laying out text consisting of a mixture of scripts — a phenomenon that is increasingly common in the modern world. Vertical font metrics are essential for traditional Japanese, Chinese, and Mongolian typography. And graphic designers will tell you that access to stylistic alternates, optional ligatures, swash forms, and other glyph variants is absolutely essential even for "simple" scripts like Latin.

And it doesn't stop there. Ligated forms in scripts like Devanagari or Arabic may be composed of two, three, or even more individual characters. In many usage scenarios, users should be able to highlight and edit the individual components of ligatures in a natural way within text composition software. For example, a user should be able to highlight and change just a *hamza* or a *shadda* over a letter or ligature form in Arabic without having to obliterate and then re-type the entire cluster of letters comprising that ligature all over again. Unfortunately, highlighting and cursor positioning in programs like OpenOffice.org was designed only with western scripts —not eastern scripts— in mind. It is often very difficult to tell on which part of a ligature the cursor is actually positioned because the required visual feedback is simply not there. As a result, it is often just simpler and faster to erase and retype a whole word rather than try to get the cursor positioned in exactly the right spot to fix a single mistyped character. Smart cursor and highlighting behavior as I envision it should work is not yet available in any software I am aware of.

In order to provide advanced features such as those described above, two technologies are available for exploitation by the Open Source developer community: *OpenType* and *Graphite*.

The OpenType[5] font technology was developed by Microsoft and Adobe and is the most popular technology. In the commercial world, tens of thousands of fonts are now available with OpenType features. Additionally, OpenType will soon become an ISO standard[6]. Full-featured support for OpenType should be seen as a key goal by the FLOSS development community. The Open Source *HarfBuzz* library, now being maintained and developed by Behdad Esfahbod, is one library that provides access to OpenType font features. The library originated in the FreeType project, was later developed separately in Pango and QT, and now is being merged back into a common repository

which will be used by both Pango and QT.

Advancing FLOSS Typography with Graphite

Graphite is a very exciting Open Source smart font technology from SIL with a well-designed application interface (API) and capabilities to handle the complexities of all known modern writing systems[7]. Due to recent work by Daniel Glassey and others, Graphite support is now integrated into Pango and into GTK widgets. Although now available in GTK, certain advanced features of Graphite, such as split insertion bars, discontinguous range highlighting, and the manipulation of ligature components, are not yet available in GTK. Nevertheless, the integration of the very capable Graphite into the very popular Pango/GTK libraries is an exciting development. Daniel Glassey reports from Akademy (September, 2006) that the QT developers are also interested in integrating Graphite. Let's take a brief look at some of Graphite's impressive capabilities.

First, Graphite permits the full range of character-to-glyph mappings: one-to-one, many-to-one, one-to-many, and many-to-many. And in all these cases, Graphite keeps track of the mappings in both directions, from character to glyph, and from glyph back to character.

Unlike "dumb" fonts, glyphs rendered by Graphite can have their positions adjusted both vertically and horizontally. This is needed, for example, when creating a stack of diacritical marks above or below a base character. Graphite can also modify the advance width of a glyph. In Graphite, a base glyph can have multiple attachment points for diacritics, and the diacritics themselves can form chains of attachments. Base glyphs with attached diacritics can form clusters and additional glyphs can then be positioned relative to whole clusters (*Figure 10*).

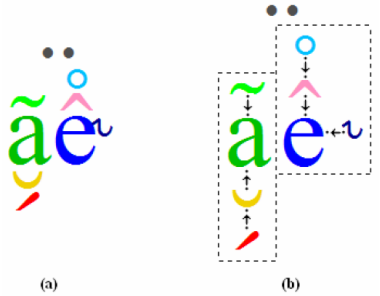

Figure 10. In Graphite diacritics can be bound to base glyphs at multiple attachment points to form glyph clusters. Additional glyphs, such as the diaeresis shown, can then be positioned relative to glyph clusters.

In Graphite, ligatures are not simply substituted monolithic glyphs. Instead a ligature in Graphite comprises both the visually rendered ligated glyph as well as the underlying characters used to create the ligature. It is possible to define rectangular areas of a ligated glyph that correspond to the underlying characters. This makes it possible to select the individual characters in a ligature for the purpose of highlighting or editing within an application. Because the highlighting and mouse selection routines are handled by the Graphite interface, no complex programming is required by the calling application (*Figure 11*).

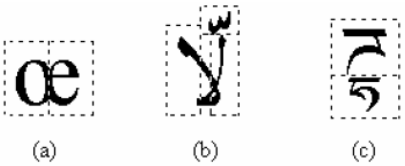

Figure 11. Rectangular areas represent the component characters within a ligature in Graphite. These can be highlighted or selected within a calling application without complex additional coding.

Graphite's advanced and flexible feature set and well-designed API clearly have an important role to play in any and all discussions around advancing text rendering in future FLOSS systems. Software developers interested in advanced typography need to take a look at Graphite to see what they've been missing. Font developers should likewise do the same to see how fonts can benefit from Graphite technology.

Seamlessly Integrating OpenType and Graphite Technologies

The sheer popularity of OpenType provides FLOSS developers with a mandate to support this

technology. At the same time, the advanced feature set of Graphite has already attracted wide interest in both the GTK+ and QT developer communities. This virtually guarantees that Graphite will become an integral part of the FLOSS text rendering pipeline alongside OpenType. These two technologies do not have identical APIs or feature sets. It is therefore imperative that the FLOSS developer community reviews both technologies comprehensively before sitting down at the drawing board to provide users with an integrated set of text layout services that expose the best features of both technologies in a seamless and natural manner.

Word Breaking & Syllabification

The following modern languages commonly do not use spaces or other markers between words when written in their native scripts:

- Thai
- Lao
- Khmer (Cambodian)
- Myanmar (Burmese)

In addition to these, a number of related minority, liturgical, and historical scripts of Southeast Asia also do not use spaces between words when used to write native languages or the Pali language of the Buddhist Canon. For example:

- Lanna (*Unicode proposal exists*)
- Tai Dam (*Viet Tai, Unicode proposal exists*)
- Tai Le (*Dehong Dai, now in Unicode*)
- New Tai Le (*now in Unicode*)

Since there are no spaces or other markers of word endings, word breaking requires a knowledge of the vocabulary and grammar of the written language. Developing word- or syllable-segmentation algorithms for these languages can be non-trivial and requires a in-depth knowledge of the specific language that one wants to analyze.

Vuthichai Ampornaramveth (Khun Hui) wrote the Thai "text-cutting" program *cttex* used as the basis of *libThai* which *Qt* and *Pango* use for Thai word boundary analysis. The basic principle is to choose the result of a word-breaking operation that yields 1) the longest matching words and also 2) the smallest word counts.[8] As a simple example, consider the Thai phrase, " ทำการบ้าน" which has three components, "ทำ" (/tʰam/, *to do something*), "การ" (/ka:n/, *the doing of something, often used a prefix*), and "บ้าน" (/ba:n/, *home, house*). The phrase could be cut as either:

1) ทำ การ บ้าน (*3 words, no errors*) -- or-- 2) ทำ การบ้าน (*2 words, no errors*)

... but option #2, "*do homework*" results in the longest matching words and smallest word count and this is the right answer. Khun Hui provides additional details and examples in his blog (*in Thai*).

In a localized sense, *libThai* solves the word boundary analysis problem for Thai. But, as Theppitak Karoonboonyanan told me, "*To share the code of (the cttex) word break module with other scripts, I'm not*

sure if the underlying mechanism is ready for that."[9] *Cttex* operates on TIS-620 text and therefore is not a candidate for extension or modification, even for the very closely-related languages like Northern Thai (Lanna) or Lao.

Jens Herden of *Khmeros.info,* also using a dictionary-based approach, has written software to solve the problem for Khmer.[10] Although researchers are actively working on word boundary analysis for languages like Myanmar[11, 12] and Lao[13], to the best of my knowledge the results of such research have not yet become practical software available in Linux distributions.

What the FLOSS community needs is a lightweight but powerful object-oriented framework for word boundary analysis that could be plugged into a unified text layout and rendering pipeline. We can imagine having a virtual base class from which two sub-classes would be immediately derived: one for *dictionary-based* segmentation (needed for *Thai, Khmer, Lanna*), and another for *rule-based* segmentation (*Lao*). These could be further sub-classed as necessary.

IBM's International Components for Unicode (ICU) library implements a word break iterator that includes code for Thai.[14] Word break iterators for other Southeast Asian languages do not yet exist in ICU. Perhaps ICU's class structure will be useful when thinking about how to design a unified text layout and rendering pipeline that includes robust word-boundary analysis for all of the scripts mentioned above and more.

SIL's Graphite API currently provides classes for rule-based but not yet for dictionary-based word and syllable segmentation. Perhaps extensions to Graphite's API could include dictionary-based word segmentation at some point in the future.

With the right foundation, creating a fast, lightweight, and extensible library for word boundary analysis to handle the scripts of Southeast Asia is possible. However, only the coordinated efforts of knowledgeable people in the world-wide FLOSS developer community can make the possibility become a reality.

Segmentation of Pali Texts -- Esoteric or A Natural Outcome of Good Foundational Designs?

Before leaving the subject of word segmentation, I'll mention an "esoteric" idea. Pali, the principle language of the Theravadan Buddhist Canon, has historically been, and continues to be, written in the numerous scripts of Southeast Asia.

When written in any of the scripts of Southeast Asia that do not place spaces between words, one can certainly envision how the fixed or nearly-fixed spelling used in the texts could be mapped from a source script into another script used for the dictionary-based lookup needed to perform word segmentation. In other words, it might be possible to write a single Pali word segmentation algorithm that would work equally well for Pali texts written in Thai, Khmer, Myanmar, Lanna and other scripts.

Extensive computerized dictionaries for Pali, such as the Pali Text Society's dictionary[15], already exist. If there also existed a well-designed Open Source class library for handling dictionary-based word segmentation and related tasks such as spell checking for Southeast Asian scripts, Pali scholars with a bent for software design might find the possibilities quite intriguing.

While the Pali example here might sound esoteric, the idea of creating well-designed foundational infrastructure which can serve as the basis for many avenues of unforseen and creative exploration in FLOSS systems is most certainly *not* esoteric. It's just good design practice.

Hyphenation

A problem closely related to the problem of word breaking for Southeast Asian typography is the problem of hyphenation in Western typography. John D. Berry, a typographer who has been writing about type for the past 15 years, points out that hyphenation is a big subject[16]. First of all, the algorithms used in major commercial programs like *PageMaker* and *QuarkXPress* are sufficiently imperfect that some manual line breaking is almost always employed by professional typographers. Secondly, Berry points out that hyphenation rules for English differ between the Americans and the British —including within their respective spheres of linguistic influence.

One library for hyphenation in the FLOSS world is Raph Levien's *libhnj*[17]. According to Peter Moulder[18], several different programs —for example *Scribus* and *OpenOffice.org*— have independently modified *libhnj* and have even used incompatible hyphenation dictionaries. The forking originally occurred due to *libhnj*'s inadequate support for Unicode. Moulder thinks it would nice if these could be unified, especially if the code could share dictionaries with TEX again (a task that may be difficult due to the embedding of various TEX commands and macros in the TEX dictionaries). If FLOSS systems are to achieve a degree of consistent behavior across different software applications, it certainly would make sense to provide hyphenation as a shared system service.

Fonts are the First Step in Making Advanced Typography Both Possible And Convenient

It is fair to say that we are currently witnessing just the beginning of a new trend to produce high-quality free/libre fonts with advanced typographical features made possible by technologies like Graphite and OpenType.

For many scripts, much work remains before it will be actually *convenient* to produce high-quality typography on FLOSS systems. For other scripts, much work remains before it will even be *possible* to produce high-quality typography.

Let's use Arabic as an example where it is *possible* but not yet *convenient* to produce high-quality typography. A quick survey of statistics from Ethnolgue[19], SIL[20], and Wikipedia[21] suggest that there are at least 266 million people who use Arabic as the primary script for writing their language (*Arabic, Farsi, Urdu, Uyghur*, etc.). Based on the high rates of population growth in the Middle East alone, this estimate could be off by 100 million[22]. There should be no question that this is an important market segment deserving serious attention by the FLOSS community. Software packages like *ArabTEX*[23] are capable of advanced layout of Arabic, including specifying ligatures, presenting fully vocalized text, and precision glyph positioning. However *TEX* is probably not what most not-technical users would call *convenient.*

All usable Arabic fonts require OpenType or Graphite tables in order to typeset contextual glyph forms correctly. The vast majority of the free/libre Arabic fonts available today do not have glyphs for the common ligatures like initial ت /tā'/ or ي /yā'/ followed by م /mīm/ (*figure 12*), or medial ي /yā'/ followed by final ن /nūn/ or ر /rā'/. Sometimes this lack of ligature forms is intentional, as the forms are perhaps deemed unnecessary in modern font styles.

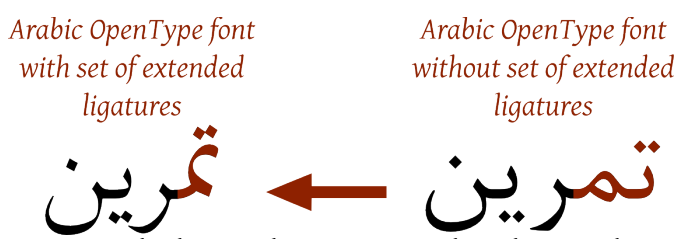

Figure 12.Technologies such as OpenType and Graphite provide numerous opportunities for developers to create fonts that result in higher quality typography.

However, another view holds that using such fonts is essentially no better than typing Arabic on an old mechanical typewriter. The computer as a tool should open up new avenues for beautiful and creative typography rather than simply reinforce the limitations of the technology of the previous era. While the availability of optional ligated forms in Arabic fonts is no guarantee of better Arabic typography, it is a first step in the right direction.

Another problem with a great many of the currently-available Free Arabic fonts is that fully vocalized text is not supported correctly in popular programs such as OpenOffice.org. It is not clear to me whether the origin of this problem is in the fonts, in the applications, or somewhere at the crossroads of interaction between the fonts and the applications.

As Arabic fonts with extensive ligature sets and correct diacritic placement require more work to produce, it is not surprising that few are available under free/libre licenses. The Uyghur Computer Science Association[24] is the only organization I am aware of which is providing freely downloadable Arabic fonts with extended ligature sets. This situation will undoubtedly change in the future. It will have to change if we want to make FLOSS operating systems more appealing to a wider audience of Arabic script users.

Issues with quality font availability are by no means confined to the Middle East. We could just as easily talk about problems with CJK fonts for East Asia. For example, a *Hei* style (黑體) font to serve as the default *sans* font for Chinese doesn't even exist yet, and some consider the freely-available Japanese fonts inadequate for Japanese.

Preparing for New Scripts in Unicode

At the 21st International Unicode Conference in Dublin, Ireland in 2002, Michael Everson presented a paper entitled *Leaks in the Unicode Pipeline: script, script, script...*[25] In his paper, Everson noted that although there were —at that time— 52 scripts allocated in the Unicode Standard, at least another 96 remained to be encoded! Of the 96 noted by Everson, a whopping 33 (one-third) represent scripts that continue to be used to write modern spoken languages or continue to be used for liturgical purposes. The rest are historical scripts which remain important for scholarship and study.

Since 2002, a some of the scripts mentioned by Everson, such as *Tifinagh* and *N'Ko* have now become incorporated into the Standard. A review of Deborah Anderson's *Scripts Encoding Initiative* web site at University of California at Berkeley[26] shows that of the now over 100 scripts listed, just 21 --one-fifth of the total-- have so far been approved for inclusion. Of those not yet approved, proposals have not yet been written for a good portion. And of those scripts for which proposals do exist, the quality of the proposals varies (excepting the high-quality proposals written by Everson who is an old hand at it by now). We can conclude that the Unicode "script incorporation rate" is slow, as a large amount of work and research is usually necessary before a script can be added to the Standard.

Of course many of these unencoded scripts are related to other scripts already encoded in the standard. Some are simple left-to-right alphabets and syllabaries without special features, and we can conclude that it will not be difficult to produce fonts for the simpler scripts once encoded.

However a number of the other scripts have special features. Some have unique features not shared by other scripts, while others are quite similar to other scripts already encoded in the Standard.

As an example of this latter category, let's take a brief look at the traditional script of Northern Thailand called *Lanna*. Lanna exhibits features typical of Indic scripts, such as having vowels that may precede, sit on top, or hang below base consonants. However, unlike central Thai but akin to scripts of South India like Kannada, Lanna also has consonant glyph forms that hang below (phonetically) preceding consonants. For example, looking at *figure 13*, we can see that the second letter ๓ /p/ in the Pali word "nippan" (meaning *nirvana*) hangs below the preceding ๓ /p/. In addition to that, the example above shows that these hanging consonantal forms usually have completely different shapes from the normal forms -- compare the word *nippan* in the middle with the form on the right in the first row where you can see how the ๓ /p/ is written as ◌๎ in the subjoined form.

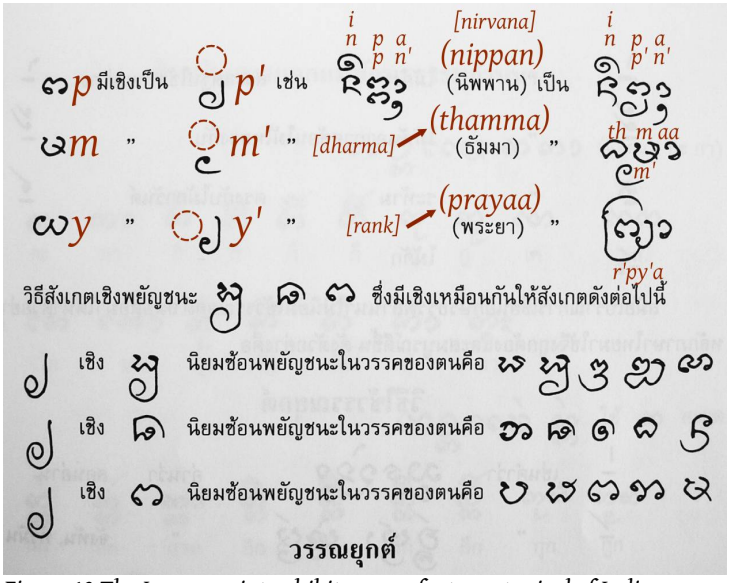

Figure 13.The Lanna script exhibits many features typical of Indic scripts. This traditional script of Northern Thailand is much more complicated to write —or typeset— than the central Thai script.

Also observe that these subjoined consonant forms don't always hang directly beneath the preceding consonant. In the second row, the first letter ๓ /m/ in the word "thamma" (*dharma*) appears to be hanging in between the initial ๐/th/ and the second ๓ /m/. Finally, in the third row in the word "prayaa" we see that the letter /r/ *precedes* the letter ๓ /p/.

The central Thai script doesn't have anything like these features, although other modern Indic scripts do have similar features. Whereas modern Thai fonts do not require special Graphite or OpenType features, it should be evident that making good Lanna fonts will probably require a bit of work and a very good understanding of the orthography of the script in addition to an understanding of Graphite or OpenType.

The Lanna script was recently considered for inclusion in Unicode at the JTC1/SC2/WG2 - ISO/IEC 10646 - UCS meeting in Tokyo in September, so it might not be long before this script becomes available in FLOSS systems.

Font Configuration and Customization in FLOSS Systems

Recent Freedesktop.org-sponsored IRC meetings[27] have highlighted a number of issues related to font configuration and customization on FLOSS systems.

At first glance, it would appear that all that is needed is to come up with a list of high-quality FLOSS fonts for the major scripts in Unicode, and create a master Fontconfig *fonts.conf* file to serve as the *One Ring* to rule and bind them all. Alas, the reality is perhaps not so simple as that. Careful inspection reveals that there are subtle issues because different cultures can have very specific cultural preferences -- even when those cultures share a common script.

As a first example, let's use Modern Vietnamese which now shares with much of the Western world the use of the Latin script for writing the national language, *Quốc Ngữ*.

Vietnamese orthography differs from that used by many Western languages in having more diacritical marks, especially on the vowel letters. As diacritical marks take up additional space, designed-for-Vietnamese fonts tend to have greater vertical line spacing than similar fonts designed for use in the West. This helps alleviate any appearance of "crowding" caused by the plethora of diacritics. In *figure 14* we can see that the VU Pho Tho sans-serif font has a much greater default line spacing compared to DejaVu Sans and this leads to an airier appearance.

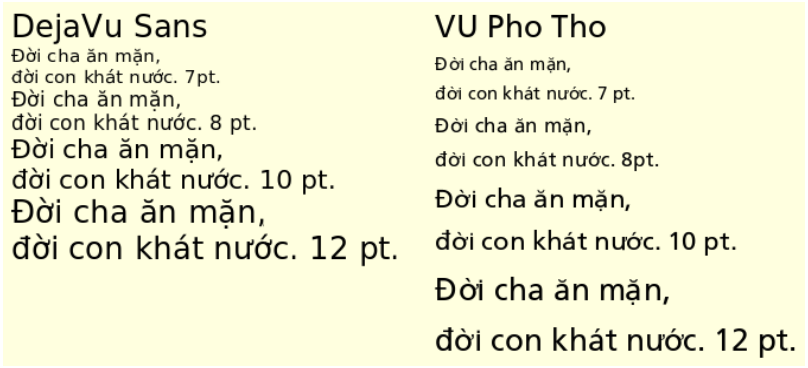

Figure 14. Waterfall presentations of DejaVu Sans and VU Pho Tho. While DejaVu Sans has very readable diacritics, only the designed-for-Vietnamese font Vu Pho Tho on the right has the larger line spacing that Vietnamese readers expect.

Seeing the small diacritical marks of Vietnamese on a computer display can be difficult unless the font's diacritical marks are large enough. This may require a larger default font size, or careful hinting of the diacritical marks, or both. While the display of diacritical marks in DejaVu Sans is quite readable, Vietnamese users of FLOSS systems may find that the default line spacing in DejaVu is too small. As a default font, DejaVu may prove to be a sub-optimal choice for the Vietnamese market, and VU Pho Tho may be preferred.

The Vietnamese example above is perhaps not a familiar one among Western developers. A better-known example involves Chinese and Japanese. Although it seems that the actual differences in the designs of glyphs between Chinese and Japanese fonts are really quite small, they are apparently large enough to create acrimony among Chinese and Japanese users forced to use each other's fonts. Japanese users want to have Japanese fonts. And Chinese users want to have Chinese fonts.

Examples of two characters which are said to typically differ in glyph form between Chinese and Japanese fonts are 直 (Chinese *zhí* ㄓˊ : *straight, vertical*) and 骨 (Chinese *gú* ㄍㄨˊ : *bone*), shown in *figure 15*. The Japanese tend to only accept the "直" form as being correct whereas both "直" and "直" forms appear in Chinese fonts. A similar situation holds for "骨" where the little hook in the box on the top can go to either the left or right in Chinese fonts, but only to the right in Japanese fonts. As my father used to say, *de gustibus non disputandem est*[28].

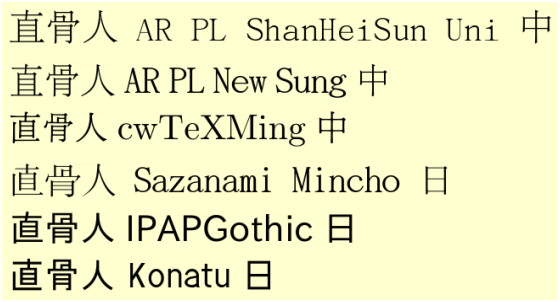

Figure 15. Differences in the designs of Chinese (中) and Japanese (日) glyphs are subtle ... but users still complain.

With the two previous examples now in mind, it should be evident that the following snippet of a *fonts.conf* file from the *Fedora Core 6* development branch is just *not* the answer, is it? :

```
<alias>
        <family>DejaVu Serif</family>
        <family>Bitstream Vera Serif</family>
        <family>Times New Roman</family>
         ...
        <family>Luxi Serif</family>
        <family>Kochi Mincho</family>
        <family>Sazanami Mincho</family>
        <family>AR PL ZenKai Uni</family>
        <family>AR PL SungtiL GB</family>
        <family>AR PL Mingti2L Big5</family>
        <family>ＭＳ 明朝</family>
        <family>Baekmuk Batang</family>
        <family>FreeSerif</family>
        <family>MgOpen Canonica</family>
        <default><family>serif</family></default>
</alias>
```

In fact, this snippet of *fonts.conf* code is wrong for a lot of other reasons too. Can you see what some of them are?

One solution would be to create different *fonts.conf* files for different locales. For example, a *fonts.conf.zh* file would clearly place Chinese font families ahead of Japanese, while *fonts.conf.ja* would do exactly the opposite. And a *fonts.conf.el* file for Grecian locales would certainly never place *MgOpen Canonica* at the bottom of the list!

But that solution is, I'm afraid, not ideal. The problem is that many people in the world want -- and need-- to work in more than just one script or orthography. Some people might have a need to commonly work in three or four or even more scripts or orthographies all in a day's work. Take the hypothetical example of a Chinese graphics designer who frequently does business for clients in Japan: to be constrained by any system that always gave priority to Chinese fonts would be ridiculous.

Configuring Fonts by Script and Orthography

Suppose for a moment that Fontconfig were expanded to recognize some new xml tags: *<script>* and *<orthography>*. We could then write *fonts.conf* snippets like the following:

```
<script>
        <name>latn</name>
        <orthography>
                <name>vn</name>
                <alias>
                        <family>VU Pho Tho</family>
                        <family>DejaVu Sans</family>
                        <default><family>sans</family></default>
                </alias>
        </orthography>
</script>
```

This would would tell Fontconfig to prefer the VU Pho Tho font as the default *sans* font for any Vietnamese text. Obviously this rule would be used at runtime for users with an environment set to a Vietnamese locale. But --and this is perhaps the more interesting case-- the rule would also be applied when a user *--regardless of locale--* visited a *sans-serif* Vietnamese web page that had been tagged with a *lang* or *xml:lang* tag for Vietnamese.

Although not shown in the short example above, it should be obvious that the *<script>* section for *latn* could contain any number of nested *<orthography>* sections, including of course a *default* orthography. The default orthography section for Latin would now look a lot cleaner than the Fedora example we saw earlier, as now only Latin font families would be included.

Let's see at what the situation would look like for Chinese Japanese users. A CJK snippet should look something like the following (*note that the ISO 15924 code*[29] *for CJK ideographs comprising Hanzi, Kanji, and Hanja* is "*hani*"):

```
<script>
        <name>hani</name>
        <orthography>
                <name>ja</name>
                <alias>
                        <family>Sazanami Mincho</family>
                        <default><family>serif</family></default>
                </alias>
        </orthography>
        <orthography>
                <name>zh</name>
                <alias>
                        <family>AR PL ShanHeiSun Uni</family>
                        <default><family>serif</family></default>
                </alias>
        </orthography>
</script>
```

... which says to prefer *Sazanami Mincho* as the *default serif* font for Japanese text, and to prefer *AR PL ShanHeiSun Uni* as the *default serif* font for Chinese text. Any Chinese or Japanese user who didn't like that could change it -- preferably with a GUI tool.

Configuring fonts hierarchically by script and orthography as suggested above would be a boon for other scripts such as Arabic too. For example, the preferred font style for Urdu speakers is quite a bit different than that for some other languages written with the Arabic script. On top of this, many languages, including Urdu, Farsi, and Uyghur, require additional characters that are not present in all Arabic fonts. The model suggested here would allow very precise tuning of default fonts for all scripts and orthographies and would thus eliminate many of the problems that people are currently experiencing.

Configuring Fonts for Screen and Print

The Fontconfig library currently defines sections for *serif, sans*, and *monospace* fonts. It is useful to recognize that the terms *serif* and *sans* (being an abbreviation of *sans-serif*) are only really applicable in Western typography. Non-Western script traditions do not have a native concept of "serifs." It would be more accurate in a global typographical context to replace the word *serif* with *modulated*, and the word *sans* with *unmodulated. Modulation* refers to changes in the stroke width of an imaginary pen used to draw a glyph. *Serif* fonts are thus a subset, and specifically the Western subset, of a larger global set of all modulated fonts, while *sans-serif* fonts are the Western subset of a larger global set of all visually unmodulated fonts. I am not sure whether such a change in the accepted terminology, regardless of the inadequacy of the accepted terminology, will ever occur.

In any case, the dichotomous *serif* and *sans* categories represent only two of many legitimate ways of categorizing fonts. *Monospace* is another legitimate category referring to a very narrow set of Western fonts designed to behave like letters typed on a typewriter. Since these categories are only three of many possible legitimate categories, it is worth asking what additional categories would be the most useful to have available for defining default font sets?

The almost Herculean efforts by the Wen Quan Yi[30] project to produce high-quality bitmap fonts for Chinese at common screen-viewing sizes highlights the importance of fonts designed specifically for viewing on computer displays. While the Wen Quan Yi project is tackling the problem for Chinese, many other language community organizations around the world are working to provide solutions for their own scripts. It therefore seems that creating a category in Fontconfig for *screen* fonts would be judicious.

As a complement to the *screen* category, a *print* category is also advisable. Once again, Chinese provides a good example of why. To read Chinese on screen without excessive blurriness at small sizes —in web pages, for example— requires a bitmap font like Wen Quan Yi. However, such a bitmap font is completely inadequate for printing. Substituting a *sans* font for printing —which in the case of Chinese would be a *Hei* style (黑體) font— is also not ideal, as a *Song* style (宋體) font (such as AR PL ShanHeiSun) is generally preferred for reading printed documents, just as westerners generally prefer a Roman serif font for reading printed documents.

Blacklisting Glyphs

Since it is easier and less time-consuming to write short snippets of XML code in Fontconfig's *fonts.conf* file than it is to try to get certain fonts "fixed" upstream, the idea of providing a mechanism to blacklist so-called "terminally ill glyphs" from within Fontconfig has been widely discussed. I believe at least several different patches to fontconfig have been suggested and there is now general agreement that some blacklisting mechanism will need to be incorporated into Fontconfig. However, the exact nature and extent of that mechanism has not been agreed upon.

One of the issues has been the desire to be able to blacklist a block of glyphs for specific scripts. For example, people want to be able to enforce rules such as "don't use the Latin, Greek, or Cyrillic glyphs from such-and-such a CJK font because they look awful." Implementing hierarchical categorization of default fonts by script and orthography in Fontconfig, as suggested above, will eliminate much of the need for these kind of rules. Under such a scenario, blacklisting could be used for the much more occasional occurrence of an otherwise "good" font having just one or two "bad apples." Problems such as having really awful Greek glyphs from a CJK font appearing as the default

choice for a passage of Greek text would simply not occur.

Enhancing Development Tools for Better Typography

Just as *GCC* is an indispensable tool for Free software, George Williams' *Fontforge* has played an important role in advancing FLOSS typography. Fontforge represents a revolution in type design because it significantly lowers the barrier to entry for aspiring typographers worldwide. Yet, like other tools, Fontforge can still be improved. Some would like to see Fontforge get a GTK+ facelift, but I agree with Williams that this is not an interesting task as it fails to improve functionality in any meaningful way. Let's look at two much better ways to improve Fontforge:

First, I mentioned earlier that SIL's Graphite technology has an important place in the future of FLOSS typography. If Graphite development tools can be integrated into the well-known and popular Fontforge, this could help spur development of Graphite-enabled FLOSS fonts.

The second improvement is that Fontforge needs to provide features that assist in the process of distributed font development for projects like DejaVu and Wen Quan Yi. Let's take the DejaVu project as an example. Currently, the DejaVu development team is struggling with the fact that the DejaVu sans development files are already greater than 2MB in size. Even if a developer only changes a few glyphs, his or her colleagues are forced to receive a 2MB file update in order to sync their local repositories. The problem is that a Fontforge sfd file is just a local file-based database. What is needed instead is an option to save glyphs to a distributed network database instead. This idea is not not new. Back in April, Raph Levien wrote on the OpenFontLibrary discussion list[31]:

> *I'm primarily thinking about network access to the font repository. This basically boils down to which version control tool is chosen, and how it's set up. Should each font be one big text file, or perhaps a directory with a separate file for each glyph?*

A simple but very effective solution would be to modify Fontforge to provide an option to save fonts in a special extended-SVG/XML filesystem or directory-based database format. In place of writing out a single file representing a font, Fontforge would, as Levien was thinking, write a file for each glyph in an appropriate subdirectory:

```
...
DejaVu/sans/glyphs/g201c.svg
DejaVu/sans/glyphs/g201d.svg
DejaVu/sans/glyphs/g201e.svg
...
```

To my knowledge, the current SVG font format does not support features such as having one glyph reference the outlines of other glyphs. Therefore the current SVG font format would have to be extended with additional attributes and tags. However, once the details of the XML had been worked out, one could then treat the font repository just like any other distributed source code repository.

Another big advantage of such a solution would be that development teams would remain free to choose whichever source code management (SCM) system they wanted, be it *git*, *SVN*, *SVK*, or something else. The only enhancement required to Fontforge would be the addition of a new *Save* format. Fontforge would remain agnostic about network and SCM protocols, just as it should. It would now also become trivial to write Perl or Python scripts to perform custom management tasks such as glyph subsetting, calculation of coverage statistics, and so on.

Wrap Up: Opportunities in the Development Pipeline

In this paper, I have attempted to cover some of the big issues related to text layout on FLOSS systems. Hopefully I have succeeded to some degree, even though there are additional topics that I have knowingly avoided discussing. In particular, I failed to mention tools designed to make the lives of users easier. GUI-based font management and advanced font dialog issues are the topic of another paper, and readers are encouraged to look at my discussion, *Designing a Better Font Selection Widget,* as a starting point[32]. How text is actually typed into the computer has been almost completely ignored, excepting the brief mention of smart cursor positioning within ligatures. Although word breaking in Southeast Asian scripts was given some treatment, hyphenation in Western typography was treated much more briefly. The related issues of justification and kerning —in both Western and Eastern scripts— were not discussed at all. Satisfactory treatment of these topics could easily fill another paper.

Despite these shortcomings, let's wrap up by looking at the really big picture and asking a few questions about the roles that each of us individually and collectively can play in the process. Despite numerous imperfections, the Unicode Standard is an enormous success. But it would be a mistake to think of Unicode as some sort of *fait accompli* toward which we can take a passive attitude.

Instead, I believe the FLOSS community has a duty and responsibility to take a much more active stance in pushing the Standard forward to the next milestone. Collaboration with groups who were historically outside of the traditional FLOSS development network, such as the Scripts Encoding Initiative at UC Berkeley and SIL International, should prove particularly fruitful. Although the Unicode *script incorporation rate* is slow, nevertheless the FLOSS community and collaborating stakeholders should be ready to "move into high gear" every time a new script is added into Unicode to insure that fonts and input methods become available on FLOSS systems in a timely manner. Martin Hosken, the co-author with Michael Everson of the Lanna Unicode proposal, recently remarked to me[33]:

> *A good target should be that an implementation of a script be available for every new script appearing in a release of Unicode when the new version of Unicode is released. There is a long lag time between scripts being accepted by Unicode and their appearing in a new version of Unicode ... so there is time to test implementations and have something ready by the time the real release occurs.*

Hosken points out that SIL is now working on a tool dubbed *Scriptforge* which is designed to be a clearing house of information about writing systems and their implementations. The FLOSS community would do well to consider the possibilities here, not only for new scripts, but also for many already encoded scripts where the level of support in FLOSS systems is still sorely lacking. For example, recall that a Free *Hei (sans)* style font for Chinese is not available, and substituting an available Japanese font to fill that gap is a very crass idea.

So, as food for thought, here are a few questions:

- *How does a script progress from not even being encoded in Unicode at all to eventually becoming just another script which we can use to compose text on a computer?*
- *How can the FLOSS community contribute to make this process better and faster?*
- *What does the overall development pipeline look like?*
- *Is the process supervised at some level within humanity, or does it all just happen autonomously?*

- *Where are the gaps and holes in the process?*
- *Where are the gaps and holes in support for existing scripts?*
- *What existing organizations and projects are most capable of filling in the gaps?*
- *Where are the opportunities for establishing or enhancing communication between existing organisations and projects in order to streamline the process?*
- *Should more formal relationships be established between existing organizations in order to better track the development processes and monitor quality assurance?*
- *What aspects are entirely missing from the existing implementation processes?*

When we think about the current explosion of interest in FLOSS systems combined with new initiatives like the One Laptop Per Child (OLPC)[34] program, I think you'll agree that these are relevant questions to ask. And to answer.

Appendix

The following chart, although incomplete, is provided as a tool for readers interested in mulling over questions such as those raised at the conclusion of the paper:

Who?	***What?***	***Issues?***
** SEI - Script Encoding Initiative, UC Berkeley* ** SIL* ** Other Independent Proposal Writers*	** Find funding to do initial research on scripts* ** Prepares proposals to submit to ISO/IEC 10646 / Unicode*	** SEI is underfunded.* ** Too many scripts to encode.* ** Proposal quality varies.* ** Many scripts still have no proposals.*
** Unicode.org* ** ISO/IEC 10646*	** Approve script proposals.* ** Publish Unicode standard.*	** History shows that numerous headaches occur at implementation time for scripts that were approved without sufficient research or input from stakeholders at the time the proposals were submitted to Unicode for approval.*
Numerous developers and projects in the FLOSS Community: ** Freedesktop.org* ** X11.org* ** Fontconfig* ** FreeType* ** Gnome (GTK, Pango, etc.)* ** KDE (QT, etc.)* ** DejaVu* ** Wen Quan Yi* *... etc. etc. etc. ...*	** Develop the font and text layout rendering infrastructure for the Free Desktop.*	** Of course there are none :-)*
** Individual Font Developers* ** Language Community NGOs* ** Language Community Govt. Orgs. (GOs)*	** Develop and release fonts* ** Develop keyboard drivers and input methods.*	** Individual Font Developers, GOs and NGOs in many parts of the world don't have a clue about FLOSS licensing, don't understand*

Who?	What?	Issues?
* FLOSS Community and LUGs * SIL International	* SIL now releasing fonts under new OFL license.	GPL, have never heard of OFL. * Keyboard drivers and input method development done piecemeal: no guarantee of development for all major platforms (FLOSS, Apple, Microsoft) even though simultaneous development would save time and work.
* Individual Font Developers ? * Language Community NGOs ? * Language Community Govt. Orgs. (GOs) ? * FLOSS Community and LUGs * SIL International	* Comprehensively test fonts for compatibility on FLOSS and other platforms.	* Of course organizations like SIL and projects like DejaVu and Wen Quan Yi do test their fonts. But the testing record is much spottier for other GOs, NGOs, LUGs, Individual developers, etc. * Lack of standardized test suites.
* SIL * Unifont.org * Other Font Websites ?	* Publicize FLOSS fonts and related software so that language communities know about it.	* How many people outside of the development communities look at these resources?
* SIL ? * LUGs ?	* Provides workshops and educational seminars in how to use FLOSS software, fonts, input methods.	
* Language Communities	* Adopt and use FLOSS fonts and software systems.	* Many language communities still don't know about FLOSS at all.

References

1. Etemad, Elika J. *Robust Vertical Text Layout*. 27th Internationalization and Unicode Conference, Berlin, Germany, April 2005. *http://www.unicode.org/notes/tn22/RobustVerticalLayout.pdf.*

2. *The Boustrophedon Text Reader. http://traevoli.com/boust/screen.php*.

3. Katafiasz, Maciej (Mathrick). *The End Is Near*. *http://mathrick.org/blog/archives/2006/08/21/the-end-is-near/*.

4. *ManchuFont2005. http://sourceforge.net/project/showfiles.php?group_id=118623*.

5. *OpenType Specification. http://www.microsoft.com/typography/otspec/*.

6. Wikipedia. *OpenType. http://en.wikipedia.org/wiki/OpenType*.

7. Correll, Sharon. *Graphite Application Programmers Guide*, Version 1.01. SIL Non-Roman script Initiative (NRSI), July, 2006.
http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&cat_id=RenderingGraphite

8. Ampornaramveth, Vuthichai (Khun Hui). *การตัดคำของ cttex*. Hui's Blog, วันพฤหัสบดี, กรกฎาคม 08, 2547 (July 08, 2004). *http://vuthi.blogspot.com/2004/07/cttex.html*.

9. Karoonboonyanan, Theppitak. *Pers. comm.*, May, 2005.

10. Herden, Jens. *KDE Cross Cultural - Experiences From The KDE Localization Process In Cambodia.* Talk given at KDE Akademy 2006. Dublin, Ireland, September, 2006.
http://conference2006.kde.org/conference/talks/5.php

11. Tun, Ngwe. *Myanmar Localization Efforts and Issues.* Myanmar Unicode and NLP Research Center,

2006. *http://www.tcllab.org/events/uploads/ngwe-myanmar1.pdf .*

12. Htay, Hla Hla & Kavi Narayana Murthy. *Myanmar Word Segmentation.* Thesis work by the author in the Department of Computer Science, University of Hyderabad, India. *hla_hla_htay@yahoo.co.uk.*

13. Dalaloy, Valaxay. *Lao Syllabification for Line Breaking.* August, 2006.
http://www.tcllab.org/events/uploads/valaxay-lao.pdf

14. *Text Element Boundary Analysis.* ICU Users Guide.
http://icu.sourceforge.net/userguide/boundaryAnalysis.html.

15. The Pali Text Society. *Pali-English Dictionary. http://dsal.uchicago.edu/dictionaries/pali/ .*

16. Berry, John D. *dot-font: The justification for Hyphenation.*
http://www.creativepro.com/story/feature/8658.html.

17. Levien, Raph. *Libhnj. http://www.levien.com/.*

18. Moulder, Peter. *Pers. comm.*, September, 2006.

19. *Ethnologue Languages of the World. http://www.ethnologue.com/*

20. *Documentation for ISO 639 identifier: ara.* SIL International.

http://www.sil.org/iso639-3/documentation.asp?id=ara.

21. *List of Languages by Number of Native Speakers.* Wikipedia article.
http://en.wikipedia.org/wiki/List_of_languages_by_number_of_native_speakers.

22. Population Resource Center. *Executive Summary: The Middle East.*
http://www.prcdc.org/summaries/middleeast/middleeast.html

23. *ArabTEX. http://www.informatik.uni-stuttgart.de/ifi/bs/research/arab_e.html.*

24. *Uyghur Computer Science Association Fonts. http://www.ukij.org/fonts/.*

25. Everson, Michael. 2002. *Leaks in the Unicode pipeline: script, script, script...* 21st International Unicode Conference, Dublin, Ireland, May 2002. *http://www.unicode.org/notes/tn4/everson-iuc21pap.pdf*

26. Anderson, Deborah. *Script Encoding Initiative* web site at the Dept. of Linguistics, UC Berkeley.
http://www.linguistics.berkeley.edu/sei/who.html

27. Freedesktop.org. *Font Configuration IRC Meeting Archives.*

http://www.freedesktop.org/wiki/Software_2fFonts_2fConfiguration_2fArchives.

28. *De gustibus non disputandem est: "*There is no accounting for tastes.*"*

29. Unicode.org. *ISO 15924 Codes*, *http://www.unicode.org/iso15924/iso15924-codes.html.*

30. Wen Quan Yi (文泉驿). *Wen Quan Yi Open source CJK font project.*

http://wqy.sourceforge.net/cgi-bin/enindex.cgi.

31. Levien, Raph. *Re: Free Design Software Community Roadmap.*
http://lists.freedesktop.org/archives/openfontlibrary/2006-April/000060.html

32. Trager, Edward. *Designing a Better Font Selection Widget.* September, 2005.
http://unifont.org/fontdialog/.

33. Hosken, Martin. *Pers. comm.* September, 2006.

34. *One Laptop Per Child. http://laptop.org/*