Publishing text in indigenous African languages

A workshop paper prepared by Conrad Taylor for the BarCamp Africa UK conference in London on 7 Nov 2009.

Copies downloadable from http://barcampafrica-uk.wikispaces.com/Publishing+technology, and http://www.conradiator.com

1: *Background; how the problem arose*

PUBLISHING, if we take it to mean the written word made into many copies by the scribe or the printing press, has held a privileged position for hundreds of years as a vehicle for transmitting stories & ideas, information & knowledge over long distances, free from limitations of time & space. Written mass communication is so commonplace today that perhaps we forget some of its demanding preconditions: a shared language – for which there must be a shared set of written characters; plus the skill of literacy.

Will text remain as privileged a mode of communication in the future? Text is a powerful medium, and the Internet has empowered it further with email, the Web and Google. But remote person-to-person communication today is more likely to be done by phone than email; radio and television can tell stories and bring the news to audiences who cannot read. Despite this, the ability to deal confidently with the written word is an empowering skill. And literacy is easier to achieve when you learn in your own language.

Language, literacy and cultural identity

Shared language and a shared writing system is a precondition for written communication; but some languages and writing systems have always been more 'shared' than others. Around 1000 AD, shared fluency in Latin would let a learned Anglo-Saxon correspond with a German or Italian scholar; Arabic held a similar position in the *Dar ul-Islām* across North Africa, the Middle East and Western Asia. To escape a merely local existence, someone with ambition had to learn to write and read – and usually in a language other than their mother tongue.

In the 14th–15th centuries, Europe experienced a multi-faceted cultural revolution with the Protestant Reformation and more decentralised models of intellectual authority. This period also saw alliances emerge between national monarchies and a strengthening bourgeois class. One result of this was that literacy (and publishing) increasingly 'went local'. Local vernacular languages like English, Portuguese, Spanish, Flemish and French became languages of law and administration, made it into print and became widely read.

The growth of publishing in European national languages drove the evolution of standards for orthography (spelling systems) and grammar. Indeed some European languages achieved written form for the first time as part of this wave of national linguistic rebirth.

The irony is that not long after these European languages had climbed out from under Latin's dominance, they were carried by merchant adventurers and missionaries to the ends of the earth, and came to dominate other local cultures – most markedly in the Americas and in Africa, where most cultures did not have a preiexisting writing system. In each European country's imperial sphere of influence, its own language first became a *lingua franca* for trade; then as the great late-19th century colonial landgrab of Africa got going, colonial languages became the languages of administration, education and Christian religious proselytisation.

African indigenous-language literacy: the old non-latins

How did indigenous African languages take written form? The first point to note is that literacy in Africa is very ancient. Kemetic (ancient Egyptian) scripts were probably not the first writing system in the world – Mesopotamians most likely achieved that first – but writing developed in Egypt very early, about five thousand years ago.

The old Egyptian writing system used glyphs representing ideas ('ideographs'), but later developed into a mixed system in which some signs could represent consonantal sounds. The Nubian script of Meroë (Meroitic, from about 300 BC) was a sound-representing system derived from late Egyptian.

The Egyptian empire also provided the matrix for a Late Bronze Age innovation in writing systems with revolutionary, worldwide impact. It seems the Canaanite skilled working class, employed by the Egyptian Pharaohs for quarrying and stonecarving, created an 'abjad' (an alphabet which records only consonants, not vowels) for their Afro-Asiatic language, related to the later languages Hebrew and Arabic. Later development and diffusion from this 'Proto-Canaanite' writing system gave rise to the Phoenician, Greek, Latin, Hebrew and Arabic script systems, also those of India and Southeast Asia. Most of these scripts added extra signs to represent vowels as well.

In 332 BC, the Macedonian Alexander 'the Great' conquered Egypt. On his death, his general Ptolemy established a Greek dynasty ruling Egypt. Even after the Romans took control of Egypt from Ptolemy's successor Cleopatra, Greek language and culture centred on Alexandria dominated Egypt right up until the Arabic conquest of 639 AD. To this day Coptic, similar to the language spoken by the ancient Egyptians, is used as for the liturgy of the Coptic Christian church in Egypt. Its script is an extended version of Greek, borrowing three Meroitic signs.

The Horn of Africa has its own unique modern scripts. The region long had cultural links across the Red Sea to Yemen, forming a common cultural and trading zone from the 2nd millenium BCE which became wealthy trading in aromatic spices like frankincense and myrhh. From the old South Arabian script (a descendent of the Canaanite *abjad*), a distinct Northeast African version seems to have diverged from about C9 BCE. This became Ge'ez script. From an *abjad*, Ge'ez evolved about 1,600 years ago into an *abugida*, a kind of script in which each glyph-sign records a consonant plus an attached sign for the vowel which follows it. Ge'ez is used in Ethiopian and Eritrea, for Amharic & Tigrinya.

In North Africa west of Egypt, from Libya to Morocco and down into the areas of the Saharah occupied by the Tuareg, there is a Berber-language script system called *Tifinagh*. Though not much used for public texts and daily use, some people of the old Berber communities see its use as a marker of their identity which the Arabic dominance submerged, and so the scripts have been used on monuments, flags etc. There is no one form of Tifinagh; the oldest forms are an *abjad* and it is thought it derives from the old Punic script used by the Canaanite settlers who founded Carthage, the city which over two centuries struggled with Rome for dominance of the Mediterranean.

We must also recognise the Arabic script as a major script system of Africa. It has taken quite an effort to get computers to deal with Arabic, And not only because it runs from right to left. As a legacy from the Arabic calligraphic tradition, the script is 'cursive': consonants must join up to each other. Arabic evolved from an *abjad*, but now adds vowel markers which must be correctly floated above consonants. If columns of type are to be 'justified' (with flush left and right sides), this is achieved in Arabic by stretching the horizontal strokes rather than by increasing inter-word spaces, which is what happens when type is justified in European languages.

Despite these formidable problems, some of the national communities in the Middle East are so rich that software companies have been prepared to put in the effort, and these problems have been solved. One speculates that if the Arabic lands had no oil, and had not been so geopolitically important, there would still be no Arabic computer systems.

The 'donation' of latin scripts to African languages

The major languages of West, Central, East and South Africa now have a writing system based on 'latin' characters. It is hard to find information about how these writing systems developed, but in most cases the initiative seems to have come from Christian missionary organisations in the 19th and 20th centuries who were determined that the Bible should be translated and made available in readable form in Africa's indigenous languages, most of which did not have a writing system. As a measure of the scale of this project, at present there are about 680 of Africa's languages which have a complete version of the Bible in print.

One of the features of many Western European languages such as English and French is that they have quite crazy spelling systems (how to pronounce '*ough*' in English words?) This partly because they took 20 letters of the Latin alphabet, grabbed a couple of Greek ones such as 'z', invented a few others like 'j' and 'w' – and then, with a limited set of 26 or so letters plus some accented vowels, decided how to represent all the sounds of their language with these letters. This, plus the endless process of pronunciation drift, means that some European languages have spelling systems that are hard to learn, and lots of words that don't obey the rules.

Mereba, ma ɛnyɛ ansa
na ɛmaa kwakou dua kaa hɔ

Fig. 1 – a Twi proverb typeset in Gentium font.

Mindful of this, it seems, the missionaries and scholars who set about devising writing systems for Africa tried to create a unique and consistent way of expressing in written form each of the sounds of the target language. In some cases, a complex consonantal sound might be represented by a digraph such as *gb*. Sometimes an extra mark was added, such as in ṣ. Sometimes a specially modified letter shape was introduced, as in ɗ. For the Akan languages of Ghana, a couple of 'open' vowels were uniquely represented by phonetic notation characters, ɛ and ɔ (see sample type-setting in Fig. 1 above) and the nasalised 'ng' sound by ŋ. Additionally, in some of the tonal languages of West Africa, such as Yoruba, rising or falling tones are indicated with accent marks above the vowels.

In 1966, UNESCO organised a conference on the harmonisation of transcription system for Africa, in Bamako. A later UNESCO conference in 1978 in Niamey led to the definition of an African Reference Alphabet, which was revised in 1982.

A problematic inheritance

While these phonetic/orthographic innovations make sense and may be praised for their intent, they certainly created a store of problems for the future. Yes, a language represented with extended character sets can easily be handwritten, so it functions for teaching the language in school. As for publishing, even in the days of hot-metal letterpress typesetting, a dedicated organisation like a big printing company could equip itself with the custom fonts required to print in such a writing system.

This works – if your model for publishing is that it is a centralised, 'radiative' way of disseminating texts for mass communication: books, newspapers.

But as typewriters replaced handwriting in offices, then computers came along, languages which had had these customised writing systems solutions created for them were cursed by an inability to publish 'correctly' at low cost, and with off-the-shelf equipment. For each individual who sits at a keyboard, for each African government department which needs to produce documents in a local language, there is a dilemma: shall I invent from the font repertoire of this particular machine on my desk a means of representing our language which is not 'correct' but which I hope will be understood? Actually, what else can I do? Type, add accents afterwards with a pen, then copy?

Those African languages which do not have these special characters in their writing systems may count themselves lucky, perhaps. Decisions taken in the evolution and the standardisation of Swahili and Somali, for example, mean that they use exactly the same character set as English.

Localisation: a multi-layered problem

My focus in the rest of this paper is on African languages which use extended-latin character sets, and what is needed to make them work on a computer system. The problem has several dimensions, and there are two 'levels' of challenge.

On the one hand, there is the 'simple' issue of how to type a language like Twi into a single computer, and get from it printed output which can be published on paper, or as PDF. There is also the greater challenge of how we can enable computers (and other devices) to communicate interoperably in such languages over networks, as on the Web.

2: *Of fonts, characters and encodings*

FOR A WRITTEN LANGUAGE to be expressable in a computer publishing system, many factors must come together. It helps to think of the events that happen when you press a key on a computer keyboard, expecting a type character to appear on screen, and later to be sent to print or turned into a PDF document…

❖ **The text-processing application:** you launch a software application which allows you to choose fonts and specify the size, *etc*.

❖ **Font selection:** from a menu, you choose a font, which is a resource that contains at least a couple of hundred 'glyphs' (lettershapes).

❖ **Key-press:** you press a key on the computer keyboard, which is constantly being scanned for signals coming from it. As a result, a **scancode** is sent to the computer. This is not the same as saying 'The user pressed the G key'. The numerical scancode code represents something more like 'The sixth key from the left on the third row of the keyboard was pressed.'

❖ **Scancode to character-code translation:** An important next step is the interpretation of the keyboard scancode into a code which defines which *character* has been requested. For example, on a standard UK keyboard, the 2nd key from the left on the second row translates to a **Q**. But on a French keyboard, the same keypress should resolve to an **A**. Each computer has **keyboard layout resources**, one for each language, to determine how the keypresses translate into character references.

❖ (A note, to be expanded upon later – there are many standard definitions about which numeric codes stand for which characters. Example: in the American Standard Code for Information Interchange – **ASCII** – the number **65** represents the capital letter **A**.)

❖ **Character code to glyph selection:** In effect, a software application responds to this input by saying 'please give me the glyph shape with the ASCII character code 65'. If the font you have selected was produced to fit in with the international standards, the result should be that the letter **A** appears on screen, and will later be sent to the printer, or embedded in a PDF document.

(At this point, a useful thought-experiment is to imagine that someone has re-engineered the font so that sending the scancode 65 for 'A' actually pulls onto the screen something that looks like 'Z' – or, indeed, maybe the Twi character ŋ… more on this below).

So whether the pressing of a particular key results in the production on screen or paper of the glyph-shape that you require depends on (a) whether a suitable font is installed, (b) what encoding system is in use (c) how the glyphs in the font are organised in relation to the encoding system and (d) how the keyboard layout resources map keypresses to character codes. There's a lot that could go wrong…

Character encoding: a chaos of standards

The standardisation of numerical character encodings has been a concern for a very long time. For example, without international agreement on Baudot Code and later Murray Code, telex communication in the mid 20th century would have been impossible.

ASCII encoding was developed by the X3 committee of the American National Standards Institution and released in 1963 as a standard for communication between computers and their peripherals, and between each other. US Government insistence that all federally-purchased computers work with ASCII ensured its acceptance, and America's predominance ensured worldwide propagation of ASCII's influence.

However, ASCII was a seven-bit system which encoded only 97 printable glyphs (letters, numbers, puntuation and symbols), and these were defined to suit American society only – so, the **$** and **¢** were encoded, but not the **£** or **é**. Many national variants came about; Canada for example adopted a modification that made room for French accents. Some order was brought to the chaos in 1973 with ISO 646, a 7-bit encoding standard which defined a core 'invariant set' of 32 control codes and 78 printable characters, and allowed nations to produce 'dialects' with modified definitions of the meaning of 18 other numerical codepoints. Thus for example many countries assigned the codepoint 64 to signify @, but in France it mapped to **à**, in Hungary to **Á**, in Ireland to **Ó** and in Iceland to **Đ**, the capital letter *eth*.

The eighth bit, and chaos of a new but better kind

I mentioned above that ASCII was a '7-bit' character encoding scheme, meaning that each number was a binary one with seven binary digits (bits) Thus the character reference for **A**, code **65**, was represented inside computer memory as the binary code **(x)1000001**. I have added the (x) as a reminder that with the launch of the influential IBM System/360 series of computers in 1964, the world adopted a 'byte' of eight bits width as the standard size of a packet of data, but this eighth bit was not used by ASCII or ISO 646.

It was a tempting idea to 'colonise' that final bit, and use it to more than double the number of printable characters that could be referred to with a byte-wide numeric code. And of course this did happen, but not in any kind of standardised way. One very successful encoding born in the early 1980s was Apple Computer's **MacRoman**, which (for Mac users) defined an international character set meeting the needs of North and South America and most of Western Europe.

Microsoft similarly produced an eight-bit codepage, but both Microsoft and Apple as their market expanded worldwide had to introduce lots more codepages to satisfy the needs of the market in, for example, Eastern Europe and the Balkans, Russia, Greece, India, Israel, Turkey, Thailand, Vietnam and the Arabic-writing countries. Even the very large character sets of Japan, Korea and China were tackled, though that is beyond our story here.

The Apple selection of which characters to encode was influential in publishing, because when Adobe Systems developed the **PostScript** printing system and its fonts, they followed the MacRoman selection quite closely; and, Apple had already included in their set all the special publishers' punctuation marks such as en- and em-dashes, true curved quotation marks, guillemets and so on.

Significantly, Adobe Systems didn't 'hard-code' their fonts to MacRoman or anybody else's encoding scheme. Keeping their own system (based on glyph names), and using a further process of mapping from the *encoding* to the *glyphs*, they achieved two things: (a) their fonts would work with any of the 8-bit encodings mentioned above, provided that the system was configured correctly and (b) they were able to colonise the otherwise-wasted slots which in ASCII and its derived systems stand for various system control codes. Thus they were able to bring glyphs such as the Icelandic **Ðð** and **Þþ**, and case fractions ¼½¾ into their early fonts.

The eight-bit trap

Such was the situation as it stabilised for users of latin-script computer systems in the 1980s, and which persisted until around the millenium when new possibilities came about through the advent of Unicode, OpenType etc. But it must be said that most of the time we still find ourselves stuck in that eight-bit encoding trap. That is why there are still often 'weird' character substitutions in emails and exchanged files, and why some Web pages don't display correctly.

Apart from those annoyances, the eight-bit trap also had these consequences for publishing in African indigenous languages using an extended latin character set:

- ❖ As long as latin character set encodings were limited to an eight-bit space, there was an insufficient incentive for font-designing companies to extend their character range beyond the set required by the lucrative European and American core market.
- ❖ Without a truly international character encoding system that addresses all languages equally, other languages needing solutions within this space could only find them by what I shall call various kinds of 'creative cheating' – private solutions limited to closed groups of users.

3: *Font editing and 'creative cheating'*

ADOBE's POSTSCRIPT LANGUAGE and its associated outline font system made its début at the beginning of 1984 when Apple launched its LaserWriter, the first printer with a PostScript controller and scaleable fonts embedded. Two years later, Jim von Ehr's company Altsys released a font editing system called **Fontographer** for Apple Macintosh, making it possible for anyone to create scaleable outline fonts at a fraction of the cost of competing systems. (The German URW Ikarus font digitisation system at the time cost over $100,000).

In the late 1980s I investigated the business of electronic font editing and invested in a copy of Fontographer, because my wife Sang-usa Suttitanakul (แศงอุษา สุทธิธนกูล) – also known as 'Noi' – was keen to get our Macintosh computers working in Thai so we could more effectively produce our newsletter for Thai people in Europe. We had already made a visit to the Bitstream digital type foundry in Massachusetts, and met the type designer Matthew Carter; and I had been struck by just what a huge undertaking it is to take on the design of a whole typeface. People who repeatedly show an aptitude for the creation of entirely new typefaces, such as Adrian Frutiger, Hermann Zapf, Charles Bigelow, Sumner Stone or Robert Slimbach whose 'Myriad' design you are reading at the moment truly deserve our admiration.

But Noi and I had also made a visit to the Thai distributor of Apple computers in Bangkok – Sahaviraya Systems. In exchange for my giving them some early Adobe Illustrator Thai-themed artwork, they had given us a copy of the Thai PostScript fonts they had created for Macintosh, on the understanding that we would not distribute them but keep them for our own use only – which we did.

There was, however, a problem: the Sahaviraya fonts were encoded to work with the Thai adaptation of ASCII (so-called 'TASCII'). We installed them and tried them, but only English characters came out, because the Thai characters were all placed in the upper register of 8-bit extended ASCII space.

We were undeterred. We had Fontographer. I did not have to design the Thai fonts from scratch. We'd already used a spirit AV marker to write Thai labels onto the appropriate physical keys of the Mac Plus that Noi had adopted as her own machine. In Fontographer, I simply moved the glyphs around – so that even using a British keyboard encoding, so long as the correct font Thai font was chosen, the right Thai output would appear on screen and be sent to print. (We also learned how to make PostScript files from PageMaker, with fonts embedded, so we could send our Thai pages to a bureau for output at 2,540 dpi on a Linotronic imagesetter).

What this story shows is that it is possible, with font editing software, either to create a new typeface from scratch (hard) or move the glyphs around in an existing one (much easier) to be able to 'cheat creatively' and thus work comfortably with an 'exotic' font inside an existing encoding scheme.

On a much later occasion, I was faced with a need to write documents about mediæval Islamic culture in Andalusia and North Africa. I wanted to have a font in which I could record

Arabic personal and place names with the same distinctions as found in the multi-volume *UNESCO History of Africa*. Taking Slimbach's font 'Minion', I opened it in Fontographer, freed up some character slots I knew I would not need, copied some vowels across to the newly-opened-up character slots, and added the accents I needed. I called the resulting font *Khadim*. It worked for my purpose, but I must emphasise that doing this is only legal if you have a valid licence to the font you are modifying, and if you do not distribute the edited version of the font to anyone else. Which is not going to be a world-changing solution…

Since then, I have met many situations in which problems of typesetting African languages have been (kind-of) 'solved' by someone working with a font editor like Fontographer to create a modified font, assigning the newly created glyph forms to replaceothers that can be dispensed with, for use within some existing Western encoding scheme for Mac or Windows. But this is very much a private solution, which can produce printed or PDF output from one computer, but cannot make 'live' text files that can be shared.

4: *Unicode rides (slowly) to the rescue*

ALREADY IN 1988, it was clear to a few computer scientists working at Xerox and Apple that a proper multilingual character encoding scheme was needed to address all the world's languages and scripts, and so the Unicode project was born. To quote a 1988 document by Joe Becker of Xerox:

Unicode is intended to address the need for a workable, reliable world text encoding. Unicode could be roughly described as 'wide-body ASCII' that has been stretched to 16 bits to encompass the characters of all the world's living languages. In a properly engineered design, 16 bits per character are more than sufficient for this purpose.

This estimate was based on the fact that a 16-bit system has enough space for 65,536 character codepoints.

Others joined the working group from Sun Microsystems, Microsoft, NeXT and the Research Libraries Group, and the work began. The Unicode Consortium was incorporated in January 1991. By 1996 it was clear that 16 bits would not be enough if historical writing systems such as Egyptian and Mayan hieroglyphs were also to be incorporated, and ways of extending Unicode were invented making over a million codepoints available.

Unicode is now being developed in collaboration with the International Organization for Standardization (ISO), and the Unicode standard shares the same repertoire and codepoints with ISO/IEC 10646, the Universal Character Set standard. At present, Unicode covers 90 script systems .

Codepoints and planes

Unicode codepoint references are written using hexadecimal numeric notation – base 16 arithmetic, where the digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. (To write decimal 20 in hexadecimal, you would use 'F4'.)

Full Unicode has made provision for 16 planes of 65,536 codepoints. In practice, for the latin-based African scripts we are interested only in the **Basic Multilingual Plane**, in which codepoint references are conventionally written as **U+** followed by four hexadecimal numbers. Thus for example Ɓ, a character in the Hausa language of Nigeria written in the Boko script introduced by the British colonial administration, is referenced as U+0181.

In practice, Unicode is handled in computers using various encoding schemes, the most popular of which are the two Unicode Transformation Schemes, **UTF-8** and **UTF-16**.

❖ **UTF-8** uses one byte per code point for characters within the basic ASCII/Latin repertoire 'if it can get away with it', and includes an escape mechanism to jump up to two, three or four bytes if required. It has the advantage of being a compact way to store and transmit text that is basically of latin form, and has emerged as a *de facto* standard for Unicode text interchange.

❖ **UTF-16** uses two bytes for every character – unless there is a need to venture beyond the Basic Multilingual Plane, in which case like UTF-8 it 'escapes' to use more bytes as needed. It is the favoured method whereby operating systems and software represent Unicode text internally, for example in Windows NT and its successors, Mac OS X, KDE for Linux, and the Java and .NET environments.

As can be seen, modern operating systems are certainly now Unicode-aware. But we need more than that. We need our text processing and other applications to handle Unicode properly; we need fonts that include those characters; and we need mechanisms to access those characters.

5: *Fonts for the nations*

Together with Unicode, another liberating technical development was the rise of new font formats that were able to accommodate large numbers of glyphs.

TrueType and OpenType

TrueType was an Apple development, later adopted by Microsoft. Apple had had some clever ideas about handling large, complex character sets as part of 'QuickDraw GX', an extension of their system-wide type handler which sadly was picked up by very few application developers. However, this seems to have influenced the the TrueType specification, which can potentially support a very large set of glyphs.

OpenType is a later font standard jointly developed by Microsoft and Adobe Systems. OpenType is agnostic about whether glyph shapes are drawn in TrueType's quadratic or PostScript's cubic curves, which has made it easier to transfer professional font libraries over to the format. Also, a single OpenType font file can be used on modern Mac, Windows or Unix systems.

As with TrueType, OpenType fonts can contain very large numbers of glyphs, and they can be referenced by Unicode encodings. Additionally, the way characters relate to and can substitute for each other can be programmed, so that when an **f** and a **j** occur next to each other, the single combination

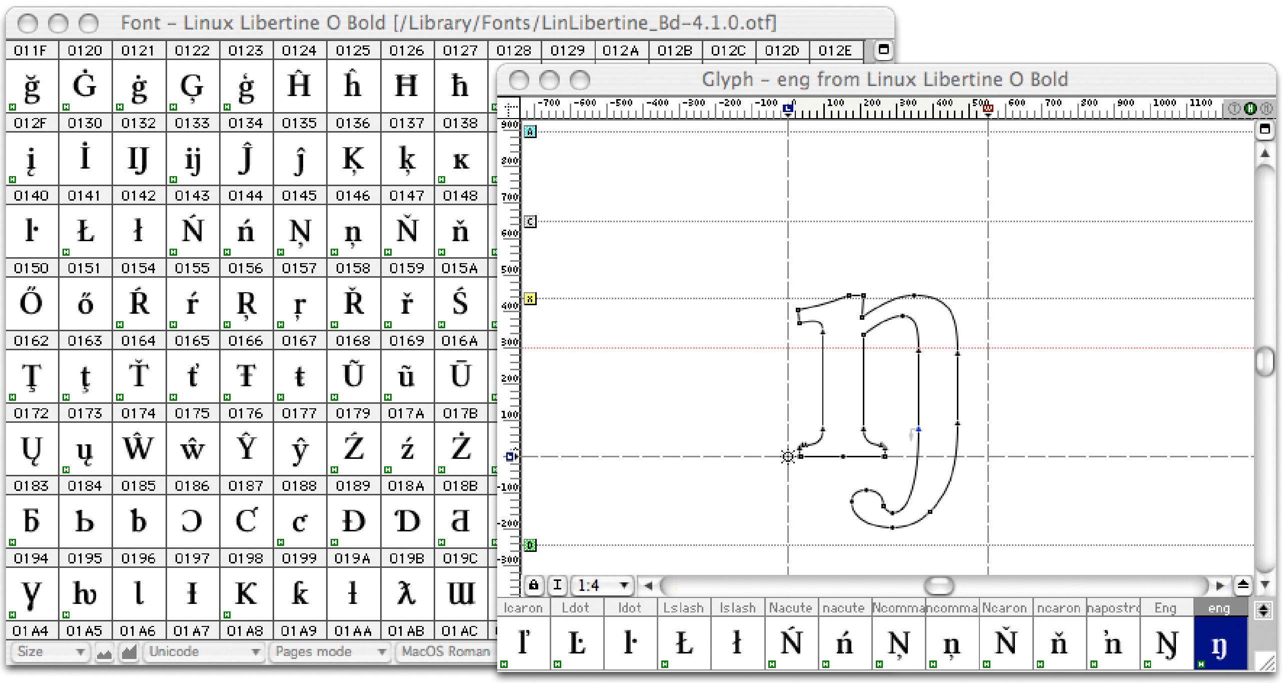
Font – Linux Libertine O Bold [/Library/Fonts/LinLibertine_Bd-4.1.0.otf]
Glyph – eng from Linux Libertine O Bold
Size
Unicode
Pages mode
MacOS Roman
1:4

Fig 2: The character 'eng' opened in the font editor FontLab. Note in the foremost window how the font outline is a series of straight lines and curves that run between digitisation points. In the window below, the repertoire of glyphs is shown: the numbers over them are Unicode codepoints.

ligature glyph ﬁ will be used for display and print without having to be explicitly specified.

Such 'glyph substitution' is vital for Arabic script, where a consonant takes a completely different form depending on whether it is at the beginning, the middle or the end of a word, or stands on its own. It is also significant for Indic (Indian and Southeast Asian) scripts which replace certain consonant pairs with a 'conjunct', which is a bit like a vertically-stacked ligature.

Fonts which accommodate many script systems

There is a handful of fonts that have been developed to cover a large part of the Basic Multilingual Plane. An early example was **Lucida Sans Unicode**, developed by Charles Bigelow and Kris Holmes, containing 1,776 character glyphs, which shipped in 1993 for Windows NT 3.1. A later revision found on Macintosh systems is Lucida Grande, which has 2,826 glyphs.

The current 'record holder' is **GNU Unifont**, developed by Roman Czyborra with over 63,000 glyphs – but it is a bitmap font, unusable for scaling up and down to different sizes as required for publishing applications.

Code2000, a font project by James Kass, is another big hitter. It is an OpenType outline font with over 61,000 glyphs, covering extended the latin scripts plus Greek, Cyrillic, Arabic, Chinese, Japanese, most of the South Asian languages – and Ethiopic Ge'ez script too. Oh, plus JRR Tolkein's invented 'Tengwar' Elvish script from *The Lord of the Rings*. But in my opinion Code 2000 has many aesthetic defects (by which I mean I find it ugly!)

Such very large repertoire fonts are going to be the rare exception rather than the rule, and fonts that attempt to cover the known universe was never what the Unicode project was about anyway. Instead, we more expect to see is fonts which address the needs of a defined group of linguistic communities – say, the non-latin script systems of Africa, or the Ge'ez scripts – but which are organised and encoded using the now-prevalent Unicode scheme.

Really, what we seeking is some kind of rationality and interoperability. What we want is that it should be possible to use entirely standard mechanisms to use the characters in these fonts – without resorting to the kind of private, non-standard 'creative cheating' methods I described towards the end of section 3.

Open Source multilingual fonts, free of charge

There is a small number of extended-character-set fonts available free of charge under the **Open Font Licence (OFL)**, or some other Open Source licencing scheme. Some of these multilingual projects got a kick-start when the Bitstream foundry donated their designs **Charter** (by Matthew Carter) and **Vera** (by Jim Lyles) to the open source community, enabling further multilingual development.

❖ **DejaVu** is a project to develop an extended-repertoire type family with matched serif and sans-serif faces in a number of weights, and with italics, based on Bitstream Vera; the project was initiated by Štěpán Roh.

❖ **Charis SIL** is a development of Bitstream Charter. The SIL.org organisation, formerly the Summer Institute of Linguistics, has a background in Bible translation and has paid particular attention in its font projects to serving the publishing needs of African communities.

❖ **Gentium** (the name is Latin for '*of the nations*') is the font I have used in this document to show samples of various African letterforms. Gentium was initiated as a private project by Victor Gaultney while he was at the University of Reading (UK) Department of Typography and Graphic

Communication. Victor designed Gentium from scratch, rather than basing it on a previous set of outlines, and in my opinion it is rather beautiful. Gentium fonts are distributed by SIL.org.

❖ **Linux Libertine** is an ongoing Open Font Licence font design project initiated by Philip H Poll. The intention was to create a free (and better-looking) alternative to Times Roman, and at present there is support for all European languages including those which use Greek and Cyrillic, many African languages, and Hebrew. Versions have been produced in both TrueType and OpenType formats.

More fonts please; but funded how?

So, there *are* now beginning to be made available a few fonts which are organised with Unicode encoding and include the glyph signs specified in the African Reference Alphabet for typesetting such languages Akan/Twi, Baule, Dinka, Ewe, Fulfulde, Hausa, Igbo, Krio, Lingala, Wolof, Yorúbà etc. But there is not enough variety.

How can we achieve a situation in which fonts supporting African education, culture and development are available in greater variety, and are diffused more generally? Consider the situation in developed countries: **most people who own a computer do not in fact purchase fonts**. They use a core set of fonts provided along with the operating system. Extra font sets are available for purchase from commercial 'font foundries', at a cost typically of several hundred pounds. We cannot expect African users to pay this much for fonts.

What Africa needs is more suitable fonts which are free of charge and quite legal to download and install. This could be achieved in a couple of ways. The first would be if more font foundries were to follow the example of Bitstream and turn over designs to the Open Source community, so that a community of volunteer designers could collaborate to extend their repertoire.

The second approach is not mutually exclusive to the first, but would involve raising funds to engage the services of professional font designers with a proven track record, to create new and original fonts, on condition that the fonts would then be released free-of-charge, perhaps with some form of permissive licence permitting others to develop the fonts further. I recommend this for consideration because over the years it has become evident to me that the design of attractive, well thought out fonts is not easy, and there are undoubted experts in the field.

There are font design companies who already undertake equivalent 'private' commissions, such as London-based Dalton Maag, who have designed corporate fonts for such brands as UBS, British Telecom, Skoda, Tesco, BMW, Toyota and Vodafone. Suppose an international agency such as UNESCO or a private philanthropic trust were to engage the services of such a firm; I estimate the cost would be in the region of £30,000 for a family of fonts, but the funder would be assured of a quality result, professional project management and timely delivery of the product.

6: *Typesetting applications & browsers*

THERE ARE A FEW MORE BARRIERS. One is, that having an extended-repertoire font will do you no good if your applications are unwilling to play along.

Let me give you an example. I am currently working on a publication design project. The client uses **FrameMaker 7.2** (Windows) as the publication typesetting tool. The fonts we have chosen are Adobe OpenType Pro fonts which support an extended range of latin-script characters – certainly all the Eastern European ones – plus Greek and Cyrillic. And yet there is no way to use characters such as **Љ** or **ű** or **ř** within FrameMaker, even if they are in the font. This is *not* an issue of how to type the characters, because even if you copy and paste across from the Windows Character Map utility all you get is question marks. FrameMaker 7.2 basically does not recognise Unicode encoding.

In contrast, to compose this document you are reading, I have used Adobe InDesign CS3 (on Apple Macintosh). This modern high-quality publication design tool uses Unicode as its method for encoding all text content, so it can be used with all the extended-repertoire latin fonts I have been discussing. I have also been able to copy blocks of Chinese text across from Wikipedia displayed in Firefox, and paste them into InDesign, where I have a range of Apple-provided Chinese system fonts available to style them.

Adobe have recognised that there will be some situations in which people want to use a few characters within a font which they cannot access from the keyboard. To meet this need, they provide a **Glyph Palette** – a window that displays the entire repertoire of glyphs in any installed font on the computer system (see screenshot above). Simply by double-clicking on a glyph in the palette, it is inserted into the text at the text cursor's current location.

Similar Unicode support, and a glyph palette for access, is I believe provided at much lower cost in Serif Software's DTP application, **PagePlus**.

One typesetting system which provides excellent quality composition but absolutely for free is Donald Knuth's **TEX** (pronounced tech), which scholars and scientists have been using since the late seventies for its maths-typesetting abilities. A recent extension of the **TEX** system is **XƎTEX**, which does use Unicode encoding and also works well with

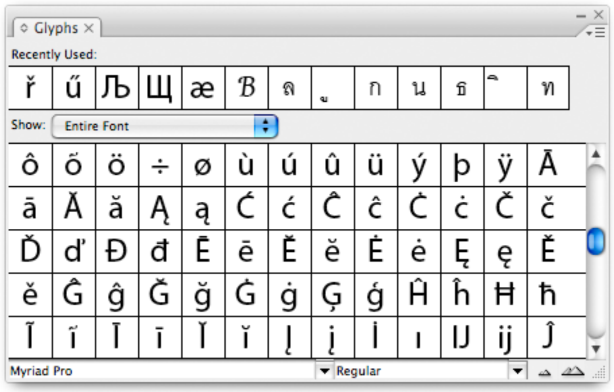
Glyphs
Recently Used:
ř ű Љ Щ æ ℬ
Show: Entire Font
ô ő ö ÷ ø ù ú û ü ý þ ÿ Ā
ā Ă ă Ą ą Ć ć Ĉ ĉ Ċ ċ Č č
Ď ď Đ đ Ē ē Ĕ ĕ Ė ė Ę ę Ě
ě Ĝ ĝ Ğ ğ Ġ ġ Ģ ģ Ĥ ĥ Ħ ħ
Ĭ ĭ Ī ī Ĭ ĭ Į į İ ı Ĳ ĳ Ĵ
Myriad Pro
Regular

Fig 3: The Glyph Palette character picker in Adobe InDesign

extended-repertoire OpenType fonts. However, it is not a 'what you see is what you get' composition tool, and there is a steep learning curve. XƎTEX is distributed by SIL.org.

Right-to-left and other composition issues

A further problem is encountered if you try to use software such as InDesign to typeset a language such as Arabic, Hebrew, Hindi, Tibetan or Khmer. In the first two cases, the text runs from right to left: the ordinary version of InDesign cannot do that, and you must purchase a special Middle East version of the software which does support mixed left-to-right and right-to-left setting. In the latter three cases, these are languages which have complex behavioural rules for collapsing groups of adjacent consonants into 'conjuncts' and combining them with superscript and subscript vowels.

There are some reasonably-priced word processors which work well with Arabic and Hebrew. Anyone using a Mac should check out the excellent **Mellel** software.

African fonts in Internet communications

When using a Unicode-compliant design tool to produce a publication, the best way to distribute it making sure it displays as you intended is to save or convert it to Adobe's Portable Document Format (PDF). That's what I have done for this document. It can be read on screen in the Adobe Reader free-of-charge software application, or printed.

As for Web, that is a bit more complicated because the person viewing the Web page has to have suitable fonts installed on their own computer. There are moves afoot to have fonts 'embedded' in Web sites, or more strictly, to have a font located on the Web at a URL and then make a reference link to it, so that the Web browser can use the font to render that page. But I confess I do not know much about the status of these developments.

At least it can be said that the latest Web browsers do work well with Unicode-encoded text, helping us make progress towards the multilingual Internet. (Earlier I mentioned being able to view texts in a number of script systems due to the combination of some Apple-provided fonts such as Lucida Grande, and Firefox's recognition of Unicode.)

Within the BarCamp Africa UK team we have made some experiments including Twi, Yoruba and Hausa characters in Skype text-chat, between my Macintosh and my colleagues' Windows computers. And it partly works (the Twi and Yoruba seemed to work, the Hausa didn't).

7: *Hacking the keyboard*

SO, LET US ASSUME that you have got appropriate fonts installed, with Unicode encoding, and on an operating system that recognises them. Also, that you have found an application which works with those fonts and Unicode to typeset your document.

There is still the problem of the keyboard. You will want to have a keyboard in which the keys are labelled with the appropriate letters. One DIY solution is to use a fine spirit marker to write on the keys, then cover them with a neatly-cut patch of non-reflective adhesive tape such as 3M's Magic Tape. A better (industrial) solution would be to have the keyboard equipped with proper screen-printed lettering, such as in the pan-Nigerian keyboard for the One Laptop Per Child XO computer (see figure 5 at bottom of page).

Recall the 'stack', the chain of causation that I explained under Section 2. When you press a key on the keyboard, a scancode is sent to the computer, and a look-up table turns that into a character code. The **keyboard layout resource** is the software which does this.

I may have to go find a custom keyboard layout and install it on my computer. Fig. 4 below tells the story: I located and downloaded and installed a Yoruba keyboard layout for Macintosh. Now it is one of the layouts available to me on a pull-down menu, and if I call up the keyboard viewer, I can see which characters live with which keys. Note that the tinted keys are 'dead keys' – if I press one, it combines with the next key I press to get the appropriate character.

There are utilities available for creating and editing new Unicode-compliant keyboard layouts: **Keyman** for Windows, or **Ukelele** for Macintosh OS X.

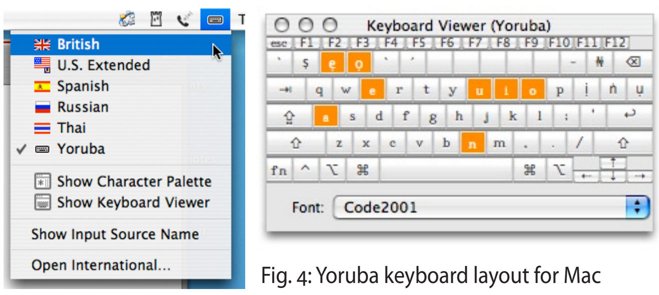

Fig. 4: Yoruba keyboard layout for Mac

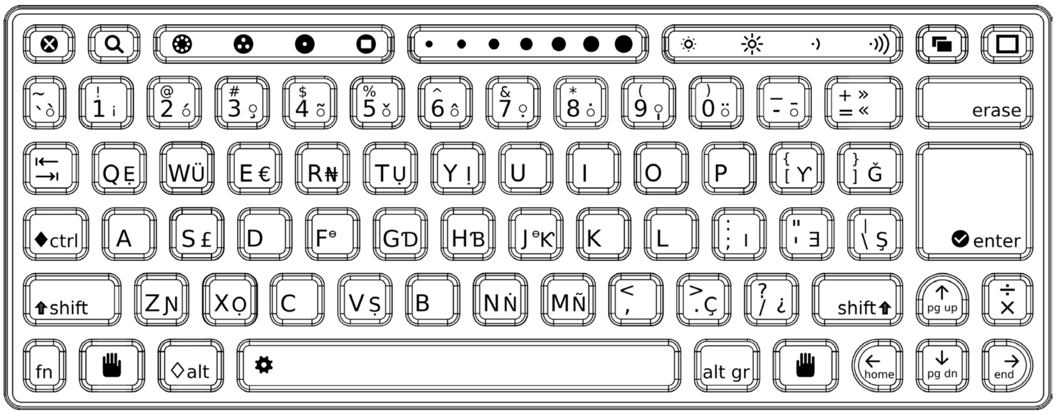

Fig 5: Pan-Nigerian keyboard for the One Laptop Per Child XO computer