Font Selection and Font Composition for Unicode

Martin J. Dürst and Marc-Antoine Parent[1]

MultiMedia-Laboratory, Institut für Informatik der Universität Zürich
Winterthurerstrasse 190, CH-8057 Zürich, Switzerland, e-mail: mduerst@ifi.unizh.ch
and
Centre de Recherche Informatique de Montréal
1801, avenue McGill College, Bureau 800, Montréal (Québec) Canada, H3A 2N4
e-mail: Marc_Antoine.Parent@crim.ca

Note: This is a prepublication version of a paper that will be published in the Proceedings of the 7th International Unicode Conference. Copyright for this and the final version is held jointly by the authors and by Unicode, Inc. This version is not intended for wide dissemination.

The Proceedings with the final version of this paper will be available at the conference, to be held in San Jose, CA, on Sept. 14/15, 1995 (for further information, contact Global Meeting Services, 3627 Princess Avenue, North Vancouver, B.C., Canada V7N 2E4, email: conference@unicode.org, voice: +1 604 983 9157, fax: +1 604 983 9158) or after the conference from Unicode Inc. (P.O. Box 700519, San Jose, CA 95170-0519, U.S.A., email: unicode-inc@unicode.org, voice: +1 408 777 5870, fax: +1 408 777 5082).

There are some differences between this and the final version, which are mainly motivated by the problem of printing Japanese characters on printers without Japanese fonts. The current version of the paper contains EPSF bitmaps for some Japanese characters. These EPSF are optimized for 24dpmm (600dpi) printers. With other printers, dropouts or antialiasing effects may lead to suboptimal representation of the Japanese characters.

[1] The second author’s work was part of Alis Technologies Inc’s “Internet en Français” project to develop Lys, a multilingual mail agent. For information on Lys, please contact Alis: 3410 rue Griffith, Montréal, Québec, Canada, H4T 1A7; Fax: (514) 342-0318; e-mail: info@alis.ca

Font Selection and Font Composition for Unicode

Martin J. Dürst and Marc-Antoine Parent[1]

MultiMedia-Laboratory, Institut für Informatik der Universität Zürich
Winterthurerstrasse 190, CH-8057 Zürich, Switzerland, e-mail: mduerst@ifi.unizh.ch
and
Centre de Recherche Informatique de Montréal
1801, avenue McGill College, Bureau 800, Montréal (Québec) Canada, H3A 2N4
e-mail: Marc_Antoine.Parent@crim.ca

Abstract – The integration of the current scripts of the world into a single character encoding standard (Unicode/ISO 10646) poses new challenges to system software and user interface designers, typographers, and font providers. Constructing or designing all-encompassing "Unicode fonts" is not feasible for several reasons in most cases; much more flexible solutions are necessary.

The paper analyses the requirements for flexible font selection and composition mechanisms from the viewpoints of typography, user interface, programmer interface and resource usage. Based on an object-oriented application framework, an architecture to satisfy these requirements in an extensible way is proposed and implemented. The tasks of font selection and glyph mapping are reduced to the same basic concepts, which also lead to interesting solutions for problems such as CJKV glyph disambiguation, and allow the easy integration of proprietary algorithms.

1 Introduction

For most people dealing with text processing systems, from users to programmers, the relation between character codes and glyphs in a font was simple: For every character code, there was a glyph, and vice versa, and every font contained glyphs for all characters. The great majority of non-specialists expect the same for Unicode: The all-encompassing "Unicode font".

For three reasons, this expectation is not going to be fulfilled: First, the relation between characters and glyphs is not one-to-one; for some scripts, complex operations are necessary. Second, the resources needed for a single complete Unicode font, and even more for a reasonable variety of Unicode fonts, are not available. This is true both in terms of human work by font designers and, for the near future, in terms of storage space. Third, from a typographic viewpoint, a single large font is not flexible enough to address the requirements of multiscript documents.

However, once a user knows about Unicode, he is not very interested in additional explanations, and expects the system to behave as if there were one or several Unicode fonts, unless he gives more detailled specifications. To achieve such a behaviour, this paper proposes a general scheme of font composition and font and glyph selection,

[1] The second author's work was part of Alis Technologies Inc's "Internet en Français" project to develop Lys, a multilingual mail agent. For information on Lys, please contact Alis: 3410 rue Griffith, Montréal, Québec, Canada, H4T 1A7; Fax: (514) 342-0318; e-mail: info@alis.ca

which can be used in circumstances where only very few fonts are available, as well as for very high quality typography.

In Section 2, this paper gives an overview of the requirements for a font selection and composition scheme. Section 3 presents the new concepts of composite font and base font, and the three main kinds of composite fonts, namely font arrays, font cascades, and font sets. Several interesting applications of composite fonts are explained in Section 4. Section 5 gives background and details of our implementation, and Section 6 compares our concepts with existing font composition schemes.

2 Requirements

Due to the vast differences in available resources, the requirements for font selection and font composition schemes cover a wide range from low end to high end. At the low end, we should try to assure basic readability, or at least make the user aware of the fact that glyphs are missing. At the high end, we should provide mechanisms to implement optimal typographic solutions for many kinds of multilingual documents.

2.1 Requirements of Multilingual and Multiscript Typography

To understand the requirements for multiscript typography, it is important to realize that multiscript documents come in a very wide variety. This goes without saying for the number of possible combinations of scripts, as well as for the basic variety that exists even for monolingual documents. Less known, but equally important is the ratio of usage of characters from different scripts, and the length of stretches of each script. In the past, typesetting multiscript documents was difficult, and there exist only few examples. Typographic theory and practice for multiscript documents are still in their infancy, and the increasing possibilities to compose true multiscript documents will will help them growing up.

The considerations made by Bigelow and Holmes [BH93] about their Unicode font already show that an attempt towards a full Unicode font faces many problems that have to be addressed very carefully. They mention the choice of a sans-serif style, with less cultural associations, the use of more differentiated diacritics, or the adjustment of Hebrew to a size between Latin capital height and x-height. The general objective is a 'harmonized design', regularizing basic weights and alignments, but preserving essential and meaningful differences.

Doing this for the Latin alphabet with all the Unicode extensions, and for alphabets such as Greek and Cyrillic, which share a large part of their history and typographic tradition with the Latin alphabet, is already a formidable task. Trying this for less related scripts becomes even more difficult. Very fundamental design concepts of Western typography, such as the uniform gray value of a line of text, cannot be used e.g. for Japanese character design.

Not only do the problems get larger if character history and typographic traditions are less related, but also problems increase dramatically as the number of scripts increases. While it may be relatively easy to find a solution for scripts A and B, and another for scripts B and C, this does not provide a solution for all three scripts together.

As an example, consider Hebrew, Latin, and Arabic. Matching Latin glyphs with Hebrew glyphs, the Latin glyphs should have rather large x-height and small ascenders and descenders to account for the fact that ascenders and descenders are virtually non-existent in Hebrew. On the other hand, combining Latin with Arabic, a Latin font with small x-height and large ascenders and descenders may be preferred to match similar features in the Arabic script. For a single Unicode font with uniform design, a compromise is necessary, which will be suboptimal for many texts with restricted script usage.

Another problem in high-quality typography is the ratio of usage of characters from different scripts. It may very well be that a different size ratio is necessary depending on whether some Hebrew words appear in a text written with Latin letters, or vice versa. This may even depend on the context. In an English text where Hebrew words are the subject of the discussion, Hebrew might preferably be somewhat larger or bolder than in a text where Hebrew equivalents are just given for reference, and may be read over. The role of the different scripts in the document influences their visual relation. The boundary between the composition of matching fonts (according to whatever criteria) and the explicit markup of foreign script portions, e.g. as **bold**, is of course not very well defined, and has to be chosen depending on the text.

The combination of different scripts can even lead to problems that can only be solved on the level of microtypography. The most important line of reference in the Latin script is the baseline, whereas ideographs are aligned to their center. For longer stretches of ideographs, it is best to try to align their center to that of the Latin text. On the other hand, for short sequences of ideographs and especially for single ideographs with a clearly visible baseline (such as 韭 or 画, as opposed to 中 or 林), the relation to the baseline of the Latin characters becomes more and more important. Even neighbouring characters can have an influence; ideographs that look right between standard text can look vertically deplaced between parentheses.

With all these problems, it should be clear that configurability and flexibility are very important. This applies both to the composition of Unicode fonts with a wide coverage from fonts for different scripts, as well as to the flexibility with respect to aspects such as size combinations, glyph selection, and glyph placement, where new schemes and algorithms should be easy to integrate into a system whenever desired.

The above explanations tried to show that a fixed, all-encompassing Unicode font is not the final solution for multiscript typography. However, this in no way meant to discourage the important design efforts towards Unicode fonts, such as those of Bigelow/Holmes [BH93], Everson (his shareware fonts are available from ftp://dkuug.dk/CEN/TC304/EversonMono10646 or ftp://midir.ucd.ie/mgunn/Everson/EversonMono 10646), Haralambous [HP95], and hopefully others.

Few users have a selection of fonts for several scripts wide enough to find good combinations easily. In these situations, a general design is by far preferable to a bad match of individual fonts. Also, Unicode for most scripts contains more characters than a standard font for that script; this applies particularly to the Latin script, even more so because most of the general symbols belong more to the Latin script than to any other. Extending existing fonts to cover the whole glyph set necessary in a single script for Unicode is therefore also very important.

Last but not least, trying to design well-matching type of different scripts, and adopting design ideas from other scripts, is an interesting artistic undertaking. It can lead to completely new designs of great excellence.

Whereas currently, uniform designs or the combinations of matching fonts have to be done by the type designer or the user, it might become possible to some extent in the future to automatize this process, and to provide higher quality in many specific situations with technologies such as Multiple Masters and others [MB93]. Similarly, with the number of characters increasing, more structured approaches to font design [Dür93], which are currently in the research stage only, may become more viable.

2.2 User Interface Requirements

From an user interface viewpoint, it is very important that interventions by the user are minimal. For a text with frequent small portions of foreign script, the user does not want to specify input method, font, and additional attributes such as language, size, and vertical adjustment over and over again. In general, the input method can be changed most easily with a keyboard shortcut, and has to be changed anyway. The font, on the other hand, should not have to be changed explicitly.

What is also important is that the system reacts in a predictive way and does not mess around with the user's specifications. If he selects Helvetica, he does not want that to be changed to anything else just because the rendering system has no way to render Arabic characters with a Helvetica setting.

2.3 Implementation Requirements

Implementation requirements can be split into two categories: General requirements and the specific requirements for the implementation platform we are using, the application framework ET++ [WGM89, WG94]. A more detailed discussion of our general approach to software globalization is contained in [Dür95]; this stands in contrast to earlier and more narrow attempts at localizing ET++ [Sat95].

Generally, it is very important to see that an average application programmer does not have the knowledge nor the motivation to care about multilingual issues, nor is it easy for her to test correct behaviour. This means that besides programs and components that specifically treat language-dependent aspects of text, no additional programming effort should be necessary. This of course includes font specifications; in this respect, there is little difference between a programmer and a final user. On the other hand, it is important that when really needed, an application programmer can easily introduce completely new functionality and is not bound to the limitations of fixed APIs or configuration files.

Another implementation requirement is efficiency, both in space and time. For applications with graphical user interfaces that contain many short text items, these items should be as lightweight as possible. Having to specify runs for different fonts on such texts makes them more heavy than if only a single font can be specified that can render whatever characters are used.

The use an object-oriented application framework such as ET++ both provides the base to realize the above requirements, and defines requirements of its own. The most important aspect is the general portability designed into ET++ [Wei92][2]. This means that the rendering model has to be very general and on a rather low level. In fact, for text rendering there is a single function that connects the application framework with system-specific code; this function draws a single glyph. Another aspect is that as a publicly available framework, ET++ has to rely on openly available fonts, which come in a very wide variety of glyph sets and encodings. On the other hand, ET++ should not disallow the use of proprietary fonts by users who have acquired them.

3 Composite Fonts

The requirements of the previous section can be fulfilled only by a system that makes a basic distinction between two kinds of fonts, namely *base fonts* and *composite fonts*. The use of composition is a very important principle in object-oriented software [GHJV95], and here again shows its strengths. Base fonts are the fonts that we all know from conventional systems; they cover a usually rather small subset of Unicode, with a consistent design. Composite fonts, on the other hand, are responsible to cover Unicode as generally as possible.

3.1 Glyph Mapping and Font Selection

Both base fonts and composite fonts can incorporate table-based and algorithmic functionality. Base fonts will care for glyph selection and mapping, whereas composite fonts have to select appropriate subfonts.

The simplest base font will just do a one-to-one mapping from the Unicode characters it covers to the glyph encoding it uses; more sophisticated base fonts will take into account the script-specific and font-specific character-to-glyph mappings. Assigning this functionality to fonts, and not building it into the core text rendering routines of a system, allows to use fonts with different encodings and different ligaturing mechanisms for the same script in the same system. This makes the system easily extensible for new scripts and new glyph mapping schemes. Even proprietary glyph mapping schemes for highest-quality typography can be added.

Composite fonts also incorporate functionality. They have to decide what portions of a text are rendered with what base font. Many ways to do this can be imagined; we will introduce three important ones starting in Section 3.3. Some interesting and not immediately obvious applications will be presented in Section 4.

3.2 Virtual Fonts and Real Fonts

With the above distinction between composite fonts and base fonts, how can we assure that when a user specifies Helvetica, which obviously exists as base font, he gets Japa-

[2] At present, there exist ports for X11, SunView, OpenGL, the Macintosh, and Windows NT.

nese drawn with Japanese letters, which are not usually contained in Helvetica, and which therefore have to be selected by a composite font? To achieve this, a font can play both the role of a composite font and of a base font. When Helvetica is used for rendering a text, it is addressed as a composite font. The composition is constructed so that it will address Helvetica again as a base font for those characters that can indeed be rendered with the base font. For the other characters, the composition will reference other fonts as base fonts, maybe with intermediate composite fonts.

We call the fonts that play both roles *real fonts*. Of course, a user or programmer may not be satisfied to always have the same fixed combinations. Additional combinations can be created, e.g. a font called `MyNewSansSerif`. Such fonts are adressable only as composite fonts, but not as base fonts, and are therefore called *virtual fonts*.

Also, it is possible that due to system limitations (e.g. only 256 glyphs per font), font composition is used below the level accessible to the user. Such fonts are not adressable as composite fonts, and therefore are called *hidden fonts*.

3.3 Font Arrays

A first kind of composite font is the font array. A font array is a very deterministic composite font; it knows by itself which of its components can draw what characters. Usually, components are responsible for a single range of Unicode characters. Font arrays can be used to simulate large fonts if these are not available due to limitations of the underlying rendering system. Another use is the splitting of fonts designed or rearranged for Unicode into smaller parts to avoid loading the whole font if only small parts of it are used. As Section 6 shows, schemes similar to font arrays are available in several display or printing systems.

3.4 Font Cascades

A font cascade is a composite font with clear priorities, for use in situations that are not as clearcut as in the case of a font array. A font cascade does not know by itself what glyphs its components can render. It starts by giving fonts with higher order priority a chance to render the text. Text portions that cannot be rendered by fonts with higher priority are passed to fonts with lower priority in turn. This procedure guarantees that the font specified by the user is used whenever possible. With the correct sequence, this also makes separate masking unnecessary. Masking otherwise has to take care that e.g. Latin characters are not taken from a Greek font, or Kana from a Korean font, even if they are available there.

Long cascades can lead to inefficiencies. But if the fonts of more frequently used scripts are arranged before the less used scripts in the cascade, this is not a problem. Finding the position of the next character that can be rendered with a given base font, so as to pass the intermediate stretch down the cascade, is very fast.

3.5 Font Sets

A font set treats its component fonts in a more equal way than a cascade, incorporates more functionality, and gives a wider selection of possibilities. It can also address some very specific problems, such as the selection of glyph variants from different East Asian typographic traditions (see Section 4.2). The basic implementation of a font set looks for the component font that can render the longest stretch of text. This can provide well-matching glyphs and can be more efficient on a display system that has a large overhead for font switching.

Other strategies for deciding between different fonts can be implemented, for example any kind of quality rating. However, the more complex the strategy, the more time will be used for evaluation, even if in many cases, most fonts drop out very early because there is a character they cannot render. For higher efficiency, it is possible to restrict lookahead to a given number of characters.

As a special case, a font set can decide whether in the rendering of combining characters, priority is given to a separate rendering with a high-quality font, or integrated rendering, maybe with a font that does not exactly match the font selected by the user.

More crucial than efficiency is stability. A font set treats its component fonts in a more equal way than a font cascade. It is therefore more volatile to changes in a text, and to the way text stretches are used for rendering. A single insertion or deletion may lead to a longer run for a certain font, or give it a better quality rating. Also, if rendering is done on units of lines or paragraphs for efficiency reasons, or even parts of lines, a different font may be chosen if the font selection process starts at a different character. Such changes are highly undesirable for the user. Without care, this can even lead to nasty loops during reformatting. Thus font sets should be implemented and used with great care.

4 Applications of Composite Fonts

Besides standard font and glyph selection, composite fonts in their various forms allow to cover a wide range of related problems. In the following, some particularly interesting examples are given.

4.1 Last Resort Font

A special position is taken by the last font in a font cascade; it is a kind of "lender of last resort". If it is asked to render a text portion, this means that the character cannot be rendered as desired, and that there is most probably no appropriate glyph. Still, there are many things that can be done, one after another or alternatively. The selection can be made by the system implementor, or in more elaborate systems by the application programmer or the user.

A composite font may have been defined to include a font of each of the scripts used, with the understanding that this covers all the necessary characters. Still, it is possible that a given font for a script does not contain a character, whereas another font in a different style, and maybe from a different supplier, contains that character. Therefore, a first thing the last resort font can do is to try to search all the fonts loaded in the application, defined in the application, or available on the system. As this uses progressively more resources, it has to be carefully implemented, if at all.

Another possibility, maybe after the above has failed, is to display some mechanically generate identification of the character. One variant is the use of four hexadecimal digits, arranged in a square, as on the Macintosh. Depending on the context, readable glyph sequences for unavailable combinations of combining characters (e.g. u" for ü) can be provided.

Another variant is to display a glyph indicating the script or category (e.g. symbols, dingbats,…) of the character only. It has been argued that such glyphs could be added to the character set of Unicode, but this should not be done[3], for two reasons: First, these are not characters themselves, they are only used on the meta-level to speak *about* characters and glyphs (respectively their absence). Second, they should not be provided in a font, and much less in a character set, but should be part of the application framework itself to assure display in any situation.

An even simpler solution is to display the same shape for all characters that cannot be rendered. In some cases, width information about the corresponding glyph may be available, and the shape may be adapted appropriately. It may be similar to, but should be distinguishable from, the glyph for U+FFFD, the replacement character for characters that cannot be *represented* (as opposed to displayed) in Unicode. In a configuration debugging mode, a popup dialog might also appear, trying to give installation hints to the user.

Additional resources may be made available through the network. In the context of the worldwide web, for example, unavailable glyphs might be replaced by inline images obtained from a special 'glyph server', or by a single active inline image representing an unknown character, which on activation would display additional information or trigger further attempts at higher-quality rendering, going as far as having the server send the whole document in bitmap form. Having the user trigger additional work, instead of doing this as a default, is reasonable because the demand on the system can by heavy, and in many cases, a user will have downloaded a document accidentally and will understand as much with replacement characters as he would with a high-quality rendering.

Similar solutions can also be envisioned for cases where transmission is requested in a form that cannot directly encode all characters of the document. ISO 10646, code-by-code identical to Unicode, is very likely to become the document character set for HTML [BC95, work in progress]. The document character set mainly defines the interpretation of numeric character references (`&#nnn;` , where nnn is the decimal representation of the character); it will take some more time until it is accepted as the default character encoding for documents that go beyond ISO 8859-1 (Latin-1).

4.2 CJKV Glyph Disabiguation

A font set can provide a simple and efficient solution to choose between different glyph variants for the typographic traditions of East Asia. The basic idea for this solution is due to Glenn Adams [personal communication, 1994]. Chinese (traditional or

[3] For ideographs, the Geta mark (U+3013) is available for historic reasons.

simplified), Japanese, Korean, and historical Vietnamese[4] all use Han ideographs. These characters have been unified in Unicode, as this has been done for all other scripts. Because of the great number of characters and the variety of similarities and historical developments, an explicit, well-defined model [Uni92, p. 15] was introduced, based on criteria from existing Japanese standards. This model assures basic readability and round-trip conversion. However, for high quality rendering, different glyphs are frequently necessary even in the same typeface [Lun94], although the mainstream typefaces are different for each typographic tradition.

With a small amount of lookahead, disambiguation is easily possible in an appropriately configured font set. If fonts for the corresponding local standards are used, disambiguation will come at no additional programming cost. A font of Japanese origin, containing the glyphs of the Japanese standard JIS X 208 [Lun93], will on average not be able to render more than two or three subsequent characters in a Chinese text, and vice versa.

In general, the following features can be used for disambiguation: First, simplifications above a certain degree have separate codepoints in Unicode; these are used in distinguishing simplified Chinese and Japanese. Second, additional characters have been invented locally; this applies for Vietnamese and Japanese. Third, the frequency distribution of certain characters is widely differing, to the extent that some characters are well-used somewhere, whereas they are obsolete somewhere else. Fourth, phonetic scripts of different nature complement the ideographs; this applies for Japanese (Kana) and Korean (Hangul).

This scheme works correctly for single-language documents and documents with different languages in each paragraph or line. On the other hand, very short foreign language pieces, such as person or place names, may not be detected, but in these cases, 'native' glyphs are used anyway in standard typographic practice, such as newspapers. More problems arise with isolated entries in menus and similar places, where localization mechanisms [Dür95] can provide appropriate solutions.

The integration of CJKV glyph disambiguation into the general font selection mechanism by means of a font set demonstrates the general usefulness of composite fonts, assures that glyph disambiguation is available for a wide range of cases automatically, and hopefully removes some of the mostly unsubstantiated concerns against Unicode in this point.

4.3 Character Replacement

Besides trying to get the best possible result with the available font resources, composite fonts can also easily be used for more fancy effects. Transliteration, e.g. from Cyrillic to Latin, can be a last resort solution when a Cyrillic font is not available, but it can also be a means to display phonetically readable characters to an user who does not know the Cyrillic alphabet, even if corresponding glyphs are available. Transliter-

[4] Contemporary Vietnamese does not use ideograms any more. Also, Unicode currently does not cover some ideograms particular to Vietnamese; their addition is planned for a future version.

ation is in many cases not very easy, and may interfere with the user's interest of seeing the base data, so that on-demand transliteration into a separate window, and implementation of transliteration as a higher-level text-changing process, similar to spell checking, hyphenation, and so on, may be more desirable. Nevertheless, that font composition can be used in this context shows the versatility of this concept.

Another application of composite fonts is the visualization of usually invisible control characters. Such a function, visualizing tabulators, paragraph breaks, and so on, is available in many text editors. Both an implementation in the application code and an implementation in the main rendering code are complicated and can be rather slow. The solution is to prepend a special font to the font cascade set by the user so that the characters are remapped appropriately.

5 Implementation

The font composition and selection scheme described above has been successfully implemented as part of the ongoing effort of globalizing the application framework ET++ [WGM89]. An application framework is a collection of cooperating object-oriented software components providing most of the functionality for applications with graphical user interfaces. ET++ itself has pioneered many object-oriented concepts [GHJV95], comes with a wide variety of sample implementations [WG94] and multimedia extensions [Ack94], and has also been used in commercial applications.

The current effort of globalizing ET++, known under the name of UNET++ (pronounced 'unity++'), is aimed at researching new concepts in the area of multiscript processing and localization while exploiting the features of application frameworks and demonstrating the special suitability of an application framework for the implementation of multiscript support.

5.1 Basic Multiscript Support in UNET++

Here, we give a short overview only of the base of UNET++; a detailed description is available [DW94]. A solution for compiler support for wide string constants is described in [Dür94]. All characters and character strings in UNET++ are uniformly encoded in Unicode, but for storage efficiency reasons, both wide (16 bit) and narrow (8 bit) string implementations are used, which are hidden from the application programmer by a common `String` class. A flexible `Mapping` class is responsible for on-demand loading and efficient storage of conversion tables. A wide range of `Converter`s has been implemented for easy conversion from and to external character encodings.

The class `KeyboardFrontend` cares for keyboard input conversions ranging from simple remapping to aggregation and server-based input conversion for East-Asian languages. `KeyboardFrontend`s are composable in much the same manner as composite fonts to achieve flexibility with a small selection of basic components. The various input methods can be selected from a "Keyboard" menu to the left of the help menu, in a similar way to the keyboard menu icon on the Macintosh [App92]. From the same menu, a dialog is available that allows to change the language of the user interface during runtime, either for the whole application or for certain windows individually [Dür95]. Even in this case, although this is a major source of programming work in other approaches (see [ODo94], p. 281), by using the inherent advantages of the application framework, in the general case the programmer does not have to change a single line of her code to make the application localizable.

5.2 *Text Rendering Classes in ET++*

ET++ uses object-oriented abstractions mainly for high-level concepts, such as `Text` (the actual text model, including stable information for styles and so on), `TextView` (high-level text display and control functionality), `TextFormatter` (line breaking and other formatting tasks), `TextPainter` (intermediate level display and control functionality), and `Port` (low-level abstraction of the rendering system capabilities).

In contrast to other systems, the text is not directly represented with objects for individual glyphs[5] [GHJV95, especially p. 34] or layout components such as runs of uniform directionality [How94]. The former approach is very flexible as long as there is a one-to-one relation between characters and glyphs, but not easily extended for more complicated cases. The later approach is using composition (of text from smaller text portions) for a minor aspect that is not familiar to the general programmer and can change very dynamically.

ET++ stores intermediate formatting information such as line divisions as runs that apply to a contiguous sequence of characters, similar to font and style information. Rendering and other operations that convert from internal representation to display representation work on stretches of text, usually lines, in the `TextPainter`. This has the advantage that the necessary operations, which are one of the major performance bottlenecks of an interactive system, can be implemented with small and fast loops.

5.3 *Fonts, Font Families, and Glyph Mappers*

In the area of fonts, ET++ uses the classes `Font`, `FontFamily`, and `FontManager` [Wei92]. A `Font` originally was a base font with implicit glyph mapping only. For simple font combinations, we introduced a single-step cascade in the form of one backup font for each font [DW94]. Now, as described in Section 3.2, a `Font` has to play both the role of a composite font and of a base font. This is achieved by relegating the actual font and glyph selection functionality to a new object, the `GlyphMapper`. Every `Font` can return two different `GlyphMapper`s that correspond to its roles as composite and base font.

Some readers may think of glyph mapping (in base fonts) and font selection (in composite fonts) as two rather unrelated tasks. The reason to have them done by one and the same object is twofold. First, flexible composition requires that both leaf components and composite components share the same superclass. Second, the functionality, seen on a more abstract level, is indeed the same, namely to map a sequence of characters to a sequence of font-glyph pairs.

[5] Note that the meaning of the term glyph here is different from its use in the field of character encoding.

The central method of `GlyphMapper` is `TranslateText`. It maps an array of characters, or part of it, and fills an array of font identifiers, an array of glyph indices, and an array of glyph-to-character indices, which allows correct cursor placement and similar functions in the case of complex character-to-glyph relations. Each subclass of `GlyphMapper` also has to provide a method that decides if it can map a character or not. Figure 1 shows the class hierarchy of `GlyphMapper`. Figure 2 shows an example of an instance hierarchy. Methods to find the next character that cannot be rendered in a given font, or the next character that can again be rendered, can be implemented for increased efficiency, but default implementations are provided by the base `GlyphMapper` class. In addition, there are some methods used on initialization.

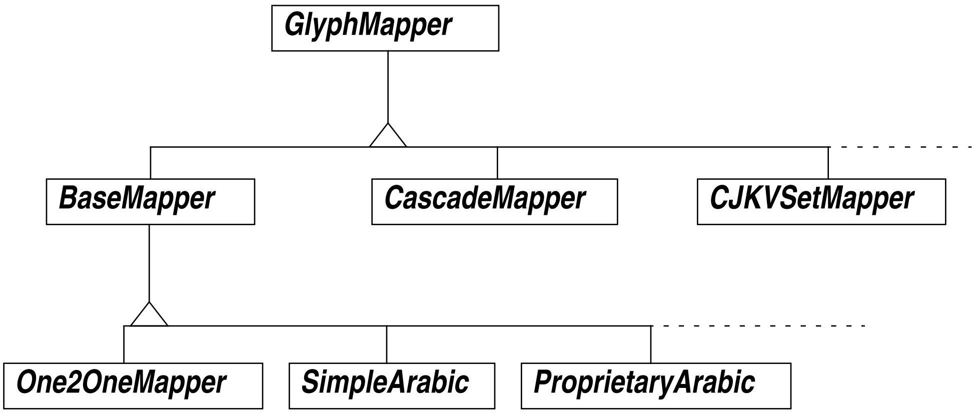

Figure 1: Part of the class hierarchy of GlyphMapper.

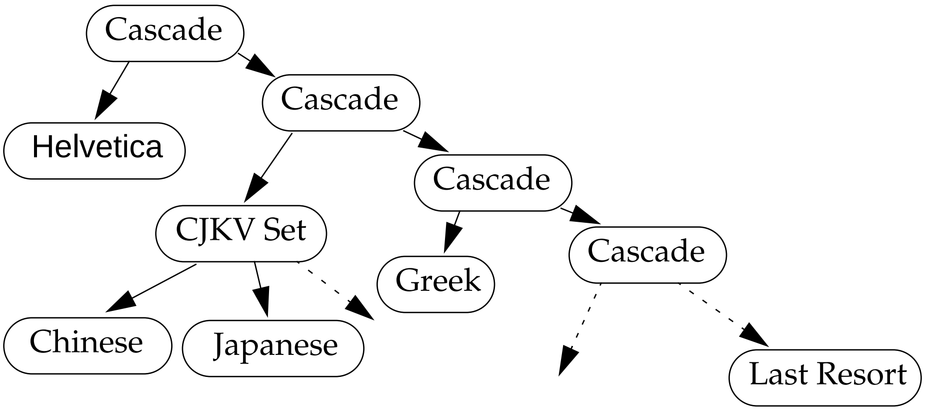

Figure 2: A configuration of composite and base fonts
for Helvetica with generic fonts for non-Latin scripts.

`Font` (together with `FontManager`) is subclassed for each display system, so that relegating font selection to `GlyphMapper` has the additional advantage of separating display system dependent subclassing from subclassing for the implementation of new font and glyph selection schemes (this is the bridge pattern from [GHWV95]). Although it is possible that a certain form of font selection is necessary or available only on a single system, in general similar font selection schemes will be used on all display systems. An additional advantage of separating `GlyphMapper` and `Font` is that different `GlyphMapper` strategies can be used with the same `Font`. Different strategies can be used either in the same backup cascade, especially in connection with last resort schemes, or in different cascades, e.g. masking out different parts or amounts of a font depending on what other fonts it is combined with.

The actual references to `GlyphMapper`s are not stored in `Font`, but in `FontFamily`. Whereas `Font` represents font instances with individual styles (Roman, *Italic*, **Bold**,…), and individual sizes for bitmap fonts, `FontFamily` represents a single consistent typeface such as Times or Helvetica in all its appearances. Although there might be situations where a backup cascade or a glyph mapping algorithm is different for different styles (e.g. an *Italic* version containing more ligatures than a Roman version of a font), this is not the general case. If necessary, additional `FontFamilies` (subfamilies) can be introduced.

5.4 Lazy Evaluation

Another important aspect of associating `GlyphMapper`s with `FontFamilie`s instead of `Font`s is that loading of actual fonts is delayed, in a kind of lazy evaluation. In the average case, with complete backup chains but only very few scripts used, this can save large amounts of system memory.

Lazy evaluation also reduces errors on document transfer. Future document formats may include definitions of composite fonts. Often, a font will be included in a backup chain as part of a general definition, but not actually used in the document, because the originator knows very well that some of his local scripts will not be readable in other locations. A system that tries to build up a full backup chain of `Font`s from the definition in the document will quickly produce an error, whereas this will not happen if the fonts are evaluated lazily.

Another small advantage of lazy evaluation can appear depending on how the height of a text is calculated. Evaluating the height of each character individually is often too slow, or impossible because the necessary information is not available from the rendering system. On the other hand, calculating the maximal vertical extension of a composite font may lead to unnecessarily large line spacing. Doing height calculations by base font can be a very reasonable solution, as there is not so much height variation is an average base font.

5.5 Advanced Features

The parameters passed to `GlyphMapper::TranslateText` as explained so far can handle basic font selection and character-to-glyph mapping including many-to-many cases. This interface can be extended to include more sophisticated features. One line of extension would be to pass a reference to the overall context, usually the text to be rendered. This can be used by a font set to obtain whatever additional information is necessary.

Specifically, language information could be obtained by a font set for use in fully deterministic CJKV disambiguation, instead of the more ad-hoc schemes described in Section 4.2. Language information can also be used for other decisions in highest quality rendering (see [Bai94] for some examples). However, care should be taken not to assume the availability of language information throughout a framework, for several reasons: First, the average user is unwilling to supply language information for every single bit of text. Second, languages do not form a single level partition of the idioms spoken and written; there are hierarchies as well as mixtures. Third, because there is a constant exchange of new words between languages, it is often difficult to decide to what language a word belongs. Fourth, language is often not the information needed for rendering purposes. The distinction between traditional and simplified Chinese, for example, is only marginally a distinction of language; it is much more a distinction of writing system and typographic tradition. On the other hand, the difference between Mandarin and Cantonese is a language difference, but is not relevant for CJKV glyph disambiguation.

Passing a general context parameter might be difficult because of the varieties of contexts; a full-fledged `Text` and a simple `String` do not have common access methods. So more specific parameters can be passed, and corresponding arrays can filled. A first possibility is to pass a size parameter, which can be used in various composite fonts to choose fonts of different nominal, but matching visual size. If an array of glyph positioning offsets is also provided, baseline adjustment between different base fonts and sophisticated diacritics processing (Arabic) can be integrated, and scripts that stack characters vertically such as Tibetan could use a single glyph for the same shape at different vertical offsets.

Another problem is the passing of context before and after the characters that actually should be rendered for ligaturing scripts. The solution we are currently pursuing is to allow `TranslateText` to look at characters before and after those it has to translate, and to limit this range with null characters. This is not only a problem of programming technique, but also of desired functionality. Whereas there is no reason to break context-dependent rendering on a change of attributes such as color, for a font or size change this is less clear because it may not be possible to link the characters nicely.

6 Comparisons

Adobe for their PostScript language and printers provides a wide variety of functionality. Type 0 fonts [Ado90] available in Level 2 implementations also provide composite and base fonts. Although the terms coincide, the functionality is different. The main aim is to interpret a stream of bytes to select characters from base fonts limited to a size of 256 glyphs. Switching schemes or font arrays as defined in Section 3.3 are possible. The newer CID/CMAP technology [Lun94] in addition allows easy sharing of common glyphs, e.g. between horizontal and vertical versions of a font.

In both cases, more sophisticated font selection schemes, e.g. using lookahead, are not possible. This may not be that important because it can be assumed that the application decides on exactly what glyphs from what fonts should be used. On the other hand, as PostScript is a full programming language, it is also possible to send a program to the printer that implements the composition techniques described in this paper. In the ET++ PostScript `PrinterPort`, we will most probably follow the former alternative, but we are currently waiting for fonts from various scripts to become available at low cost.

Release 5 of version 11 of the X Window System provides the concept of a `FontSet`, which corresponds again to a font array in our terminology, and is linked to the interpretation of character strings in a locale-dependent manner [Fla91]. The Unicode environment on AIX, based on XPG4, contains a universal font set, with a fixed association of characters to fonts and subsequent glyph mapping [Kun94]. Plan 9 [PT93, PTH94] also implements font arrays, mainly for sharing fonts of rare scripts, and in its $8\frac{1}{2}$ window system contains a very efficient caching mechanism. Unfortunately, character-to-glyph mappings seem to be limited to one-to-one.

Mule, the multilingual extension of emacs [NHT93], uses fontsets to assign fonts to character sets. A character set in a fixed way links a character tag to the number of bytes for internal representation, the character width in display cells, and the character encoding. Nonproportional display in not possible; it is simulated in the case of Arabic by separating the Arabic glyph set into wide and narrow glyphs and using two fonts and two encodings. Some backup is provided by using a font from the default fontset if the corresponding font is not available.

The Macintosh uses fonts to determine scripts, which comprise encoding interpretation, display behaviour, and associated input software [App92, App93]. Conforming applications can easily handle multiscript text, but as the concept of a composite font is lacking, in mixed text frequent explicit font changes are necessary. QuickDraw GX [App94] provides a very wide range of typographic functionality with many ways to control glyph selection and layout, but the function-based interface results in a rather static approach.

7 Conclusions and Future Work

The concepts of font selection and font composition presented in Section 3 to 5 of this paper largely satisfy the requirements put forward in Section 2. The flexible and expandible architecture we have designed and implemented allows to address issues on many different levels of typographic quality, leads to an efficient use of resources wherever necessary, and gives the programmer and the end user the benefits of a “Unicode font” while avoiding its problems. The architecture also allows to easily integrate solutions for the missing glyph problem or CJKV glyph disambiguation, as well as high-quality proprietary rendering algorithms.

To fully exploit the advantages of our approach, we plan to implement some new `GlyphMapper`s, e.g. for Indic scripts or Tibetan, and to further expand the pragmatic aspects, e.g. GUI support for virtual font construction. Ideally, there should be some program code with “typographic intelligence”, advising the user on optimal configurations or creating them automatically, but as long as there is not even a consistent metric for character height, this idea remains largely a dream.

With respect to multilingualization of ET++ in general, working solutions are available for most major aspects (input, conversion, display,…), and existing ET++ applications as well as new applications successfully use these concepts. The main areas where work still has to be done are the Unicode bidirectionality algorithm, search with regular expressions, a Unicode OS interface, character properties, collation, and so on. In addition, programming for the world means that there are always new scripts, languages, and cultures that require some adaptions or additions.

The multilingualization effort of ET++, known as UNET++, has for some time now been carried out separated from the main versions of ET++, but the features described in this and previous papers are gradually integrated back into the main version of ET++. In the meantime, we would be willing to share our code with other researchers and developers interested in collaboration and experimentation. Interested parties should contact the first author.

Acknowledgements

The first author thanks André Weinand for his continuous cooperation, Glenn Adams for his advice on Vietnamese, and Peter Stucki for his continuous support. The second author wishes to acknowledge that this work was jointly funded by Alis Technologies Inc. and the Canadian government's CANARIE program.

References

[Ack94] Ph. Ackermann, Direct manipulation of temporal structures in an object-oriented multimedia application framework, in: *ACM Multimedia 94 Conference Proceedings*, ACM, 1994.

[Ado90] Adobe Systems Incorporated, *PostScript® Language Reference Manual, Second Edition*, Addison-Wesley, Reading, MA, 1990.

[App92] Apple Computer, Inc., *Guide to Macintosh Software Localization*, Addison-Wesley, Reading, MA, 1992.

[App93] Apple Computer, Inc., *Inside Macintosh – Text*, Addison-Wesley, Reading, MA, 1993.

[App94] Apple Computer, Inc., *QuickDraw GX – Typography*, Addison-Wesley, Reading, MA, 1994.

[Bai94] B. Bailey, Unicode as a glyph identification system, *Proc. Unicode Implementers' workshop 6*, Unicode, Inc., San Jose, CA, 1994.

[BC95] T. Berners-Lee and D. Connolly, Hypertext Markup Language – 2.0, Internet-Draft, June 16, 1995. (available as ftp://nic.nordu.net/internet-drafts/draft-ietf-html-spec-04.txt)

[BH93] Ch. Bigelow and K. Holmes, The design of a Unicode font, *Electronic Publishing – Origination, Dissemination, and Design*, Vol. 6, No. 3, Sept. 1993 (Proc. RIDT'94), pp. 289-305. (Also contained in *Proc. Unicode Implementers' workshop 6*, Unicode, Inc., San Jose, CA, 1994.)

[Dür93] M.J. Dürst, Coordinate-independent font description using Kanji as an example, *Electronic Publishing – Origination, Dissemination, and Design*, Vol. 6, No. 3, Sept. 1993 (Proc. RIDT'94), pp. 133-143.

[Dür94] M.J. Dürst, Uniprep – Preparing a C/C++ Compiler for Unicode, *ACM SIGPLAN Notices*, Vol. 29, No. 1, Jan. 1994, p. 53.

[DW94] M.J. Dürst and André Weinand, Introducing Unicode into an Application Framework, *Proc. Unicode Implementers' workshop 6*, Unicode, Inc., San Jose, CA, 1994.

[Dür95] M.J. Dürst, Localization Facilities for ET++, *Proc. ET++ Workshop on Developing Building Blocks and Frameworks*, Dept. of Computer Science, University of Zurich, Switzerland, July 1995.

[Fla91] D. Flanagan, *Programmer's Supplement for Release 5*, O'Reilly & Associates, Inc., Sebastopol, CA, 1991.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.

[HP95] Y. Haralambous and J. Plaice, Ω + virtual METAFONT = Unicode + Typography, to appear in *Cahiers GUTemberg*, 1995.

[How94] W.H. Howry, Bidirectional text in an object oriented environment, *Proc. Unicode Implementers' workshop 6*, Unicode, Inc., San Jose, CA, 1994.

[Kun94] M. Kung, Unicode and XPG4, *Proc.Unicode Implementers' Workshop 6*, Unicode, Inc., San Jose, CA, 1994.

[Lun93] K. Lunde, *Understanding Japanese Information Processing*, O'Reilly & Associates, Inc., Sebastopol, CA, 1993.

[Lun94] K. Lunde, Creating Fonts for the Unicode Kanji Set: Problems & Solutions, *Proc.Unicode Implementers' Workshop 6*, Unicode, Inc., San Jose, CA, 1994.

[MB93] C.D. McQueen III and R.G. Beausoleil, Infinifont: a parametric font generation system, *Electronic Publishing – Origination, Dissemination, and Design*, Vol. 6, No. 3, Sept. 1993 (Proc. RIDT'94), pp. 117-132.

[NHT93] M. Nishikimi, K. Handa, and S. Tomura, Mule: MULtilingual enhancement to GNU Emacs, *Proc. INET'93 (Internet Workshop '93)*. (available as ftp://etlport.etl.go.jp/pub/mule/papers/INET93.ps.gz)

[ODo94] S.M. O'Donnell, *Programming for the World: A Guide to Internationalization*, Prentice Hall, Englewood Cliffs, NJ, 1994.

[PT93] R. Pike and K. Thomson, Hello world or καλημέρα κόσμε or こんにちは世界, *Proceedings of the Winter 1993 USENIX Conference*, USENIX Association, Berkeley, CA, 1993, pp. 43-50. (Also contained in *Proc. Unicode Implementers' workshop 6*, Unicode, Inc., San Jose, CA, 1994.)

[PTH94] R. Pike, K. Thomson, and H. Trickey, Unicode in Plan 9, *Proc. Unicode Implementers' workshop 6*, Unicode, Inc., San Jose, CA, 1994.

[Sat95] K. Sato, *Class Libraries Unrestricted – Introduction to Application Frameworks and Design Patterns*, Toppan, Ltd., Tokyo, Japan, 1995 (in Japanese).

[Uni92] The Unicode Consortium, *The Unicode Standard – Worldwide Character Encoding, Version 1.0, Volume 2*, Addison-Wesley, Reading, MA, 1992.

[WGM89] A. Weinand, E. Gamma, and R. Marty, Design and Implementation of ET++, a Seamless Object-Oriented Application Framework, *Structured Programming*, Vol. 10, No. 2, 1989, pp. 63-87.

[Wei92] A. Weinand, *Objektorientierte Architektur für graphische Benutzeroberflächen* (in German), Springer-Verlag, Berlin, 1992.

[WG94] A. Weinand and E. Gamma, ET++ – a Portable, Homogenous Class Library and Application Framework, *Computer Science at UBILAB – Strategy and Projects* (Proc. UBILAB Conference '94, Zurich), W.R. Bischofberger and H.-P. Frei, Eds., Universitätsverlag Konstanz, Konstanz, 1994, pp. 66-92.