

# The Unicode Character-Glyph Model: Case Studies

John H. Jenkins  
International and Text Group  
Apple Computer, Inc.

## 1. Background

One of the fundamental principles underlying the design of the Unicode Standard is embodied in the character-glyph model: that there is a distinction between the units of textual *content* (characters) and the units of textual *display* (glyphs).

Character:	Glyphs
ABCDEF...	AAAAA <i>A</i> ...

**Figure 1. Characters vs. Glyphs**

This distinction is not always a hard and fast one. Sometimes it can be hard to tell if two distinct entities are two distinct characters or two distinct appearances for the same character. Sometimes the line of demarcation can vary from language to language or expert to expert. And in a number of instances other considerations, such as the need for round-trip compatibility or political compromise, have required the Unicode standard to depart from the strict requirements of the character-glyph model.

On the whole, however, the distinction is a reasonable and most useful one. It separates text storage and operations based on the parsing of text (such as collation, searching, or content parsing) from operations based on appearance (such as display).

While the character-glyph model works very well in theory, the actual practice is another matter. Systems that implement the character-glyph model have been available

for years, but not necessarily wide-spread. There is still a very common perception that the display of a language on many systems requires a simple one-to-one mapping between character and glyph, or that all the information necessary for *complete* control of text display belongs in plain text.

We will examine here a number of specific examples of the character-glyph model in action and show how technology available in 1999 can be used to handle the model's requirements. Not all of these examples will be of Unicode text. The character-glyph model is, strictly speaking, encoding agnostic: it can be built into any number of character set encodings. Unicode and ISO/IEC 10646 depend on the character-glyph model in ways that other common character sets may not, but they are not unique in utilizing it.

One final note: The character-glyph model is not sufficient for good typography. Any system which provides support for decent typography—even in Latin—will require considerable abilities beyond what the character-glyph model specifies. Line breaking, space distribution (justification), swashes, optical alignment, kerning and tracking control—all are features that any decent typographic system must support to do its job, and they can all be done independently of the character-glyph model. Unicode provides a good solution for the computer representation of text; it does not provide the answer to the problem of beautiful type.

## **2. The technologies**

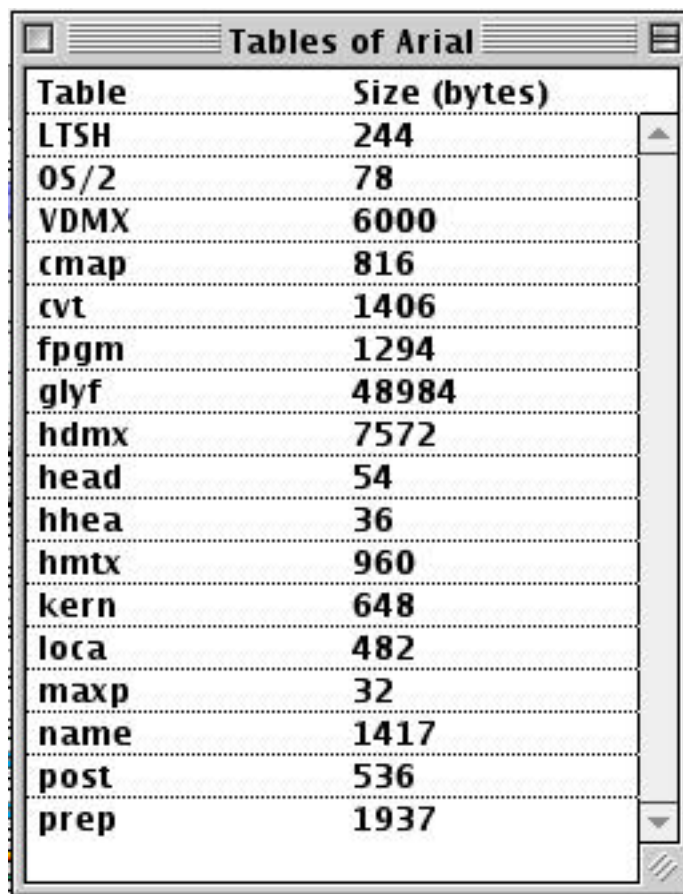
We'll focus on four technologies for the display of text, all of which are available now. Some, indeed, have been available for some time. This isn't intended to be an exhaustive list of technologies that can embody the character-glyph model; it is merely illustrative. Nor will we provide specifics on how each technology can solve each the problems discussed below. The fundamental purpose is to show how technology available *now* can implement the character-glyph model and handle the rendering of complex scripts.

### **Simple TrueType**

TrueType is an outline font format developed originally by Apple Computer. Since its original design, it has been adopted for use by Microsoft on Windows and current development and enhancement of TrueType is done by both companies working in

cooperation. The complete specification for TrueType fonts is available on-line either from <http://fonts.apple.com/TTRefMan/index.html> (the Apple version) or <http://www.microsoft.com/typography/tt/tt.htm> (the Microsoft version). The two specifications are very similar. There are some slight differences, but there is little effort involved in providing a single TrueType font that can be used on both the Mac OS and Windows.

TrueType fonts are stored in a data structure called an 'sfnt', consisting of a directory and a number of tables. The directory contains information on which tables are present, how large they are, and where they can be found within the font file.



The image shows a window titled "Tables of Arial" with a scrollable list of font tables. Each row contains a table name and its size in bytes. The tables listed are: LTSH (244), OS/2 (78), VDMX (6000), cmap (816), cvt (1406), fpgm (1294), glyf (48984), hdmx (7572), head (54), hhea (36), hmtx (960), kern (648), loca (482), maxp (32), name (1417), post (536), and prep (1937).

Table	Size (bytes)
LTSH	244
OS/2	78
VDMX	6000
cmap	816
cvt	1406
fpgm	1294
glyf	48984
hdmx	7572
head	54
hhea	36
hmtx	960
kern	648
loca	482
maxp	32
name	1417
post	536
prep	1937

**Figure 2. A TrueType Font Directory**

The tables themselves are identified using four-byte tags. While the tags can be any valid four-byte sequence, they are usually four ASCII characters to make life easier on the human programmers who use them. Thus, glyph data is stored in the 'glyf'

table; the character-glyph mapping information is stored in the 'cmap' table, and so on.

The figure above shows the directory of a Microsoft TrueType font, Arial, as distributed on the Macintosh.

Basic TrueType fonts are required to have nine tables. (A tenth, the 'OS/2' table, is required on Windows to hold OS/2 and Windows-specific metric information.) The required tables are:

<b>Tag</b>	<b>Table</b>
'cmap'	character to glyph mapping
'glyf'	outline glyph data
'head'	font header
'hhea'	horizontal header
'hmtx'	horizontal metrics
'loca'	index to glyph locations
'maxp'	maximum profile
'name'	Font and UI element name
'post'	PostScript glyph names

TrueType fonts described thus far have been a fully-integrated part of the Mac OS from version 7.0 onwards, and are also used on Windows.

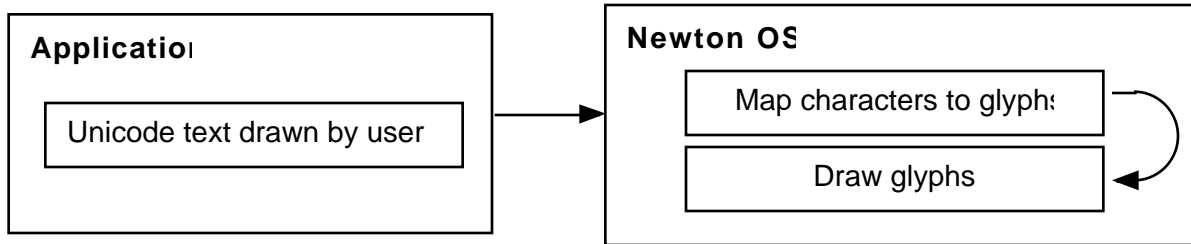
One of the values of the 'sfnt' font format used by TrueType is its versatility: other tables can be added for additional information (such as a 'vmtx' table to hold metrics for the font when used to display vertical text). Font editing tools can insert their own custom tables to simplify or speed up the editing process; such tables can be later removed before the font is actually distributed.

Apple reserves the use of tags consisting of four lower-case letters for its own use and serves as a clearing-house for the registration of new table tags; this helps avoid two designers reusing the same tag for two entirely different purposes.

It is, in point of fact, possible to completely replace the nine or ten "required" tables with other tables altogether. The first example we will consider of doing this is the version of the TrueType format used by the now-defunct Newton OS. Newton TrueType fonts lack outlines and contain only bitmaps. As such, they don't use the

`'loca'` and `'glyf'` tables, but put the glyph data in two other tables: `'bdat'` (bitmap data) and `'bloc'` (bitmap location). They also do not use or require the `'post'` table as they are, for all practical intents and purposes, unprintable.

The Newton implementation of Unicode represents Unicode at its most primitive: there is no sophisticated character-glyph mapping at all. Fonts are merely repositories for glyph and metric information, and characters map one-to-one to glyphs.



**Figure 3. Newton OS Text Drawing Model**

### **Apple Advanced Typography (AAT) and Apple Type Services for Unicode™ Imaging (ATSUI)**

Apple Advanced Typography is a TrueType enhancement that was originally introduced in 1994 as a part of QuickDraw™ GX. It is currently available on the Mac OS as a part of QuickDraw GX or through Apple Type Services for Unicode Imaging (ATSUI). Future Mac OS releases are scheduled to provide Unicode editing support through a system called Multi-lingual TextEdit (MLTE), which uses ATSUI and is a simple Unicode text editing system for use by applications. The technology underlying the character-glyph model as implemented in AAT was also used in Taligent's OpenStep operating system and is currently used in Java.

AAT enhances TrueType in two specific ways:

One, font rasterizers other than the TrueType one can be used. A custom font rasterizer can store its data in any fashion it desires, so long as it is wrapped up as a TrueType font file with a valid directory and has at least a `'cmap'` and `'name'` table. The custom rasterizer will then make data available through the system through Apple's Open Font Architecture.

Two, additional tables are provided to give control of layout information. The table germane to the character-glyph model is the `'mort'` (glyph metamorphosis) table. It is the structure of the `'mort'` table that makes AAT fonts true intelligent fonts in the sense of the character-glyph model.

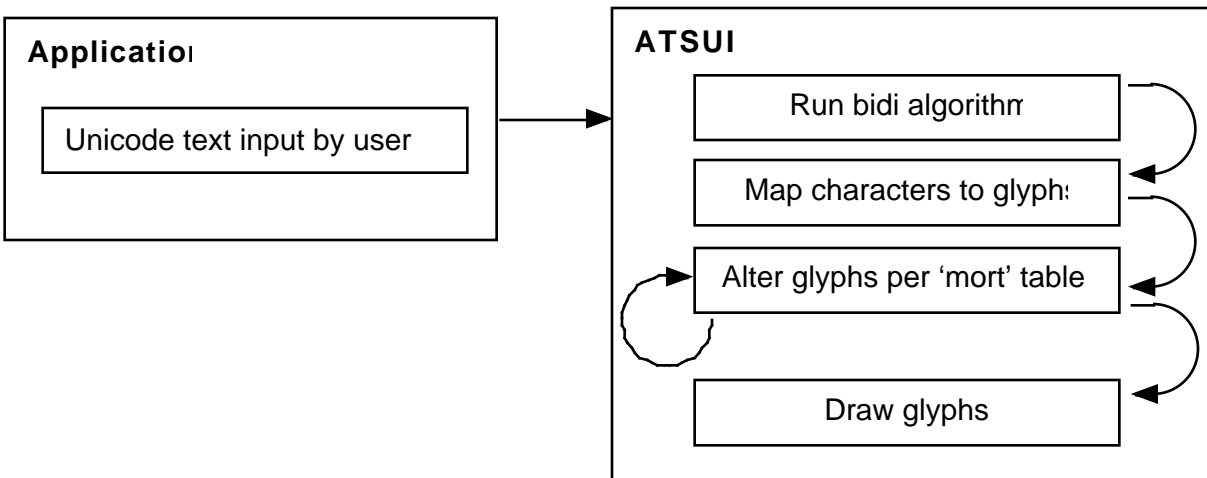
The `'mort'` table defines a series of metamorphosis actions which can be turned on and off under end-user control. These actions are grouped in terms of their general functionality (their feature type) and the specific action (the feature selector). Apple maintains a registry of feature types and selectors. Feature types and selectors are both four-bytes in size, but are usually interpreted as numbers rather than as character tags.

For example, there is a feature type for ligatures (feature type 1). Within the ligature type exist standard selectors for required ligatures, common ligatures (such as `fi` and `fl`), rare ligatures, and so on.

Font designers have complete freedom to determine what set of features and selectors their font will support, and whether those will be on or off by default. Indeed, there is no requirement that font designers restrict themselves to Apple's official list of features and types. The designers are free to create new feature types or selectors, and a properly written application will provide the user access to them.

AAT fonts are examples of fully intelligent fonts. *All* of the information on how characters map to glyphs is contained in the font, and the process of doing the mapping is done automatically by ATSUI. Since the `'mort'` table is itself built up of state tables, this means that there is actually a simple computer built into the font which can provide a large variety of complex effects which are all handled automatically by ATSUI.

(Note that other processes in ATSUI than just the glyph metamorphosis one discussed here can alter glyphs; we are presenting a somewhat simplified model.)



**Figure 4. AT/UI/AAT Text Drawing Model**

## OpenType

OpenType is a font format developed and promoted jointly by Microsoft and Adobe. As with AAT, it starts with 'sfnt' data structure as the basis; it uses fonts that adhere to the basic directory-table structure introduced with TrueType. And it enhances this structure in the same two ways:

One, OpenType allows for outline glyph data other than in the TrueType 'glyf' table. OpenType, however, is restricted to PostScript glyph outlines, which are included in the 'CFF' table. The 'CFF' table contains a compact representation of a PostScript Type 1, multiple master, or CIDFont and is structured according to Adobe specifications. (See <http://partners.adobe.com/supportservice/devrelations/opentype/cff.htm>.)

Two, OpenType allows advanced formatting data to be placed in additional tables. The table of interest for the character-glyph model is the 'GSUB' (glyph substitution) table. The text-processing client uses the 'GSUB' data to manage glyph substitution actions. 'GSUB' identifies the glyphs that are input to and output from each glyph substitution action, specifies how and where the client uses glyph substitutes, and regulates the order of glyph substitution operations. Any number of substitutions can be defined for each script or language system represented in a font.

Outside of the specific details of the differences between the formats of the `'GSUB'` and `'mort'` tables, there are some other differences between AAT and OpenType of interest regarding the character-glyph model.

OpenType fonts can distinguish between script and language-specific character-glyph transformations. AAT does not currently provide direct support for this.

AAT does not require the client to do any specific processing for certain character-glyph transformations to take place. OpenType does. That is, if a type designer comes up with a new or custom transformation specific to their own font, an AAT client would automatically provide support for it; an OpenType client would have to be specifically revised to do so.

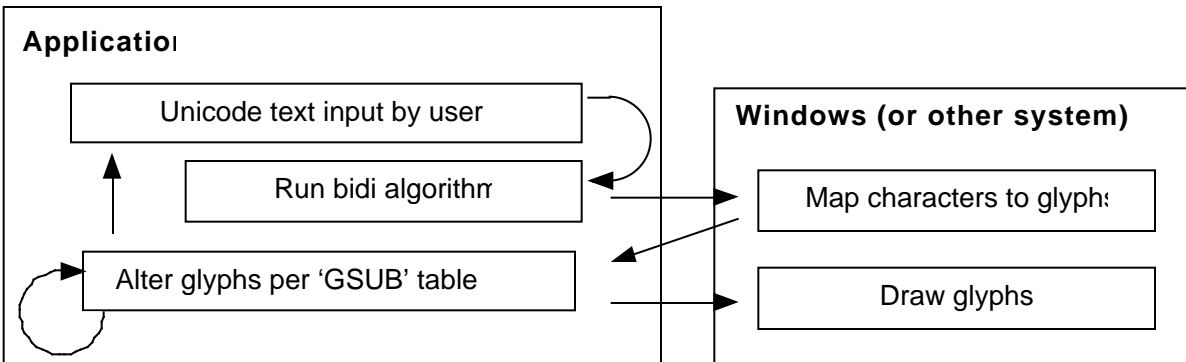
One feature of OpenType is that client applications are expected to do all glyph substitution operations common to *any* font for a particular script and language themselves. Because of this, and because the OpenType client application has to specifically interact with the font in order to do font-specific glyph substitutions, OpenType is not “fully intelligent” in terms of the character-glyph model. This should not be taken as derogatory towards OpenType, as its actual capacities are more than adequate to cover the needs of Unicode-based, high-end, multilingual typography.

The distinction is more this: A programmer utilizing an AAT-based interface such as ATSUI will not need to do any of the Unicode rendering support on their own; the system will do it all. They don't need to do any of the reordering for the bidi algorithm, nor will they need to be aware of what language or script the text is being used to represent.

A programmer utilizing OpenType, however, *will* have to include some of that in their program. They will have to do some of the bidirectional reordering themselves, and they will have to be aware of the language of the text so that they can select the appropriate glyph substitutions from the `'GSUB'` table for processing.

An OpenType programmer has to do more work; an ATSUI programmer has less control. Both approaches are legitimate. They are largely aimed, however, at slightly different sets of programmers. OpenType is more designed to be used by programmers who are writing their own text-drawing engines, such as is true for Microsoft Word and Adobe InDesign. ATSUI is aimed at programmers who want to provide

correct international support and advanced typography without writing their own text-drawing engines. Meanwhile, Microsoft is producing Uniscribe to satisfy ATSUI's target audience, and Apple is working on a lower-level interface that will satisfy OpenType's.



**Figure 5. OpenType Text Drawing Model**

### and Metafont

is a typesetting system invented by Donald E. Knuth because of his dissatisfaction with existing systems for typesetting mathematics<sup>1</sup>. As befits its inventor, it is powerful, versatile, and almost infinitely extensible.

files are plain text with mark-up directions included, rather after the fashion of HTML. The markup can vary from the trivial (such as “{\it Hello}, world.”) to the more complex (such as “ $\lim_{x \rightarrow 0} \frac{\sin x}{x} = 1$ ”). The former produces “*Hello*, world.” and the latter

$$\lim_{x \rightarrow 0} \frac{\sin x}{x} = 1$$

As befits a text markup language designed for mathematics and other technical subjects, has extensive power to handle accented letters. In this, it indeed exceeds the requirements of plain-text Unicode, in that it can do accents that stretch over two or even three letters as easily as one. At the same time, does (or can be configured to) provide considerable typographic sophistication.

<sup>1</sup> The letters in the name are actually Greek, not Latin. The name is the root of English words like “technology” and should be pronounced “tech” with the “ch” of “loch” or “Bach.” And the “K” in “Knuth” is pronounced.

Indeed, `FontView` derives much of its power from its ability to be configured with tremendous variety. Basic parameters, default macros, and information about the fonts expected to be used can all be set in configuration files.

`Standard` utilizes a related font technology, also invented by Knuth, called Metafont. Metafont provides for the definition of glyph outlines that can be converted into bitmap fonts, rather like PostScript or TrueType. `FontView` is, in fact, our first example layout technology *not* based on the 'sfnt' font structure<sup>2</sup>.

Metafont is actually a programming language that provides extensive capacity to describe and parameterize curves and their interiors. The Metafont program takes the raw text files describing the font and generates appropriate bitmaps from them, which are then used by `FontView`. Still, the ancillary data provided by Metafont fonts are limited compared to AAT or OpenType. Metafonts can have ligature tables, but there is no control over whether the ligatures are used or not; nor are more complicated substitutions directly possible.

`FontView` was originally designed for use with an extension of ASCII but can be adapted for use with other character sets. One particular extension is the Omega project, `FontView`, which is adapting `FontView` for use with Unicode<sup>3</sup>.

`FontView` allows the use of `FontView` Translation Processes ( `TPs`) in order to do complex script layout. These are applied successively to the input text stream to do the manipulations needed to get the text laid out properly. It should be noted, however, that the use of `TPs` requires coordination between `FontView` and the font. The final text character codes produced by the `FontView` will have to match the contents of the font, something that happens automatically in AAT and OpenType.

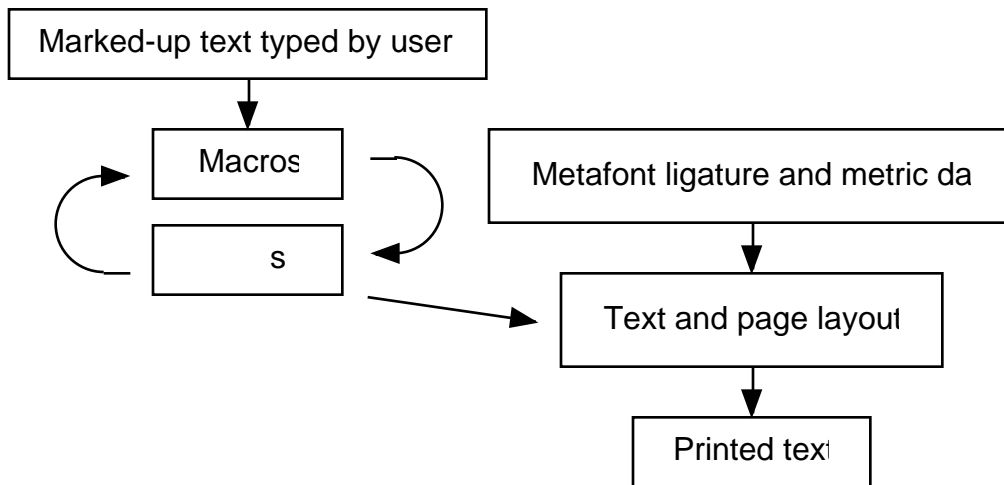
One of the major advantages of `FontView` over either AAT or OpenType is its pervasiveness and cross-platform availability. Knuth developed `FontView` and Metafont in the early 1980's and implementations are available on Windows, the Mac OS, and Unix.

---

<sup>2</sup> It is possible to implement `FontView` on top of TrueType, however. Jonathan Kew of the Summer Institute of Linguistics has developed a `FontView` implementation for the Mac OS that uses AAT fonts. It would also be possible to embed Metafonts in a TrueType font file and create a custom OFA scaler for them, making Metafonts available for all AAT clients.

<sup>3</sup> The home page for the Omega project can be found at <http://www.gutenberg.eu.org/omega/>.

One reason for choosing `TeX` to illustrate the character-glyph model is that it was developed long before the character-glyph model. `TeX`—even `TeX`—doesn't make as clear distinction between character and glyph in its own processing as do systems built on top of the 'sfnt' font format (which was a font format designed with the character-glyph model in mind). Nonetheless, it is possible to achieve the same results through careful use of the structures available.



**Figure 6. The `TeX` / Text Drawing Model**

### 3. The Problems

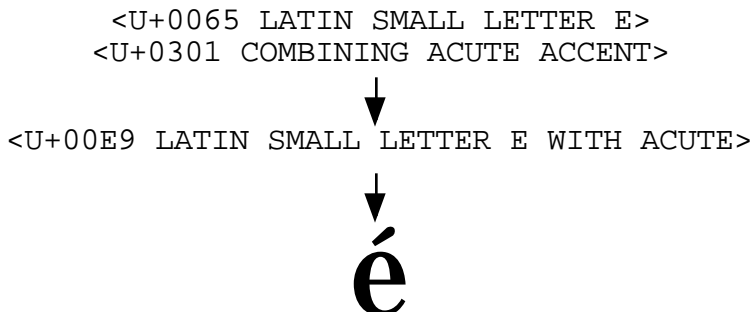
#### Problem number one: Accented Latin, Greek, and Cyrillic

Each of our exemplar technologies can handle the simplest of Unicode's "complex script" problems, accented Latin, Greek, and Cyrillic. We'll discuss this case at some length, because the general principles involved can be used for other scripts and rendering problems.

Approach A: Precompose the Unicode before drawing

The first way to represent accented Latin, Greek, and Cyrillic is to use precomposed Unicode as much as possible. You don't use `<U+0065 LATIN SMALL LETTER E><U+0301 COMBINING ACUTE ACCENT>`, you use `<U+00E9 LATIN SMALL LETTER E WITH ACUTE>`. If the text entered by the user contains `<U+0065 LATIN SMALL LETTER E><U+0301 COMBINING ACUTE ACCENT>`, the application could either transform it to `<U+00E9 LATIN SMALL`

LETTER E WITH ACUTE> before drawing, or it could actually change the contents of the document to contain <U+00E9 LATIN SMALL LETTER E WITH ACUTE> instead of <U+0065 LATIN SMALL LETTER E><U+0301 COMBINING ACUTE ACCENT>.



**Figure 7. Approach A**

The first drawback to this approach is that it requires a Unicode code point to work. If there isn't one in the formal standard—that is, if you're dealing with an unencoded entity such as LATIN SMALL LETTER Z WITH TILDE, you can assign it a Private Use code point before rendering.

The second is that it requires the font to have the accented character as a glyph before it can be drawn at all. If the font doesn't have a ž glyph, your character U+E000 LATIN SMALL LETTER Z WITH TILDE will show up as a blank box or something equally unpleasant. There is also the problem of trying to find an appropriate character in the Private Use Area to use, and in coordinating the use of the Private Use Area between the font and the layout engine.

Because and s use alternate character codes as intermediates in processing, and the specific use of these character codes isn't exposed in interchanged text, there is more freedom to use code points *outside* of the Private Use Area in such cases. The Unicode standard tolerates a degree of nonconformity, so long as that nonconformity is never publicly exposed.

Approach B: Draw precomposed glyphs using ligatures

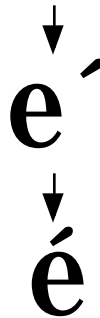
A similar solution is to use ligatures to represent accents. This is the approach favored by AAT, based on the assumption that a hand-tweaked glyph will always look better than an algorithmically generated one. This is not a terribly different approach

than using precomposed Unicode, except that it allows for unencoded combinations. For example, a font designer may create a *z̃* in their font, even though `LATIN SMALL LETTER Z WITH TILDE` is not encoded in Unicode.

AAT, OpenType, and `fontspec` all allow for the formation of ligatures. Ligature formation is typically done in “glyph space,” rather than “character space.” That is, AAT and OpenType define the formation of ligatures in terms of glyphs and not in terms of characters. (`fontspec` is slightly different, since it doesn’t make a distinction between characters and glyphs. In the case of `fontspec`, you simply add a ligature table to your metafont that gives the rules for creating a single character out of a pair.)

That is, AAT and OpenType do *not* define the “fi” ligature as formed by the sequence of characters `<U+0066 LATIN SMALL LETTER F><U+0069 LATIN SMALL LETTER I>`. Rather, it is defined as an operation resulting from the glyph “f” being followed by the glyph “i.”

`<U+0065 LATIN SMALL LETTER E>`  
`<U+0301 COMBINING ACUTE ACCENT>`



**Figure 8. Approach B**

Defining ligatures as being formed out of glyphs, not characters, allows the ligature-formation rules to be independent of the character set used for the text—an advantage for AAT specifically—and it allows ligatures to be formed at the tail end of a long series of other glyphic transformations—an advantage for both AAT and OpenType.

Ligature formation is mediated by the contents of the `'mort'` table in AAT. The table uses the feature type 1 (ligatures), and various feature selectors depending on whether the ligature is absolutely required by the language (as would be the case for

an “é” drawn as a ligature), or generally accepted (as in a “fi” ligature), or rare. Other possibilities are allowed, including the definition of user-specified ligatures.

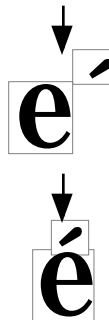
OpenType uses a number of feature tags for ligature data. One is the `'liga'` tag for standard ligatures such as “fi”, the `'hlig'` tag for historical ligatures, the `'clig'` tag for contextual ligatures, the `'dlig'` tag for discretionary ligatures—that is, those used for special effect—and the `'dpng'` tag for diphthongs. These last two tags may trigger changes in the original text. If, for example, an “AE” was turned into an “Æ” by the `'dpng'` tag, the original text should be changed appropriately to include both `<U+00C6 LATIN CAPITAL LETTER AE>` (as a normalized form for searching and replacing) and `<U+0041 LATIN CAPITAL LETTER A><U+0045 LATIN CAPITAL LETTER E>` (for decomposition of the ligature, if that should prove necessary).

The main disadvantage of this approach is that it still requires the font designer to have explicitly included the glyph in the font. At least, however, if the glyph is missing its pieces might still be present. If you don't see “ž”, at least you see “z~”, which is better than “ ”.

Approach C: Create new accented forms on the fly

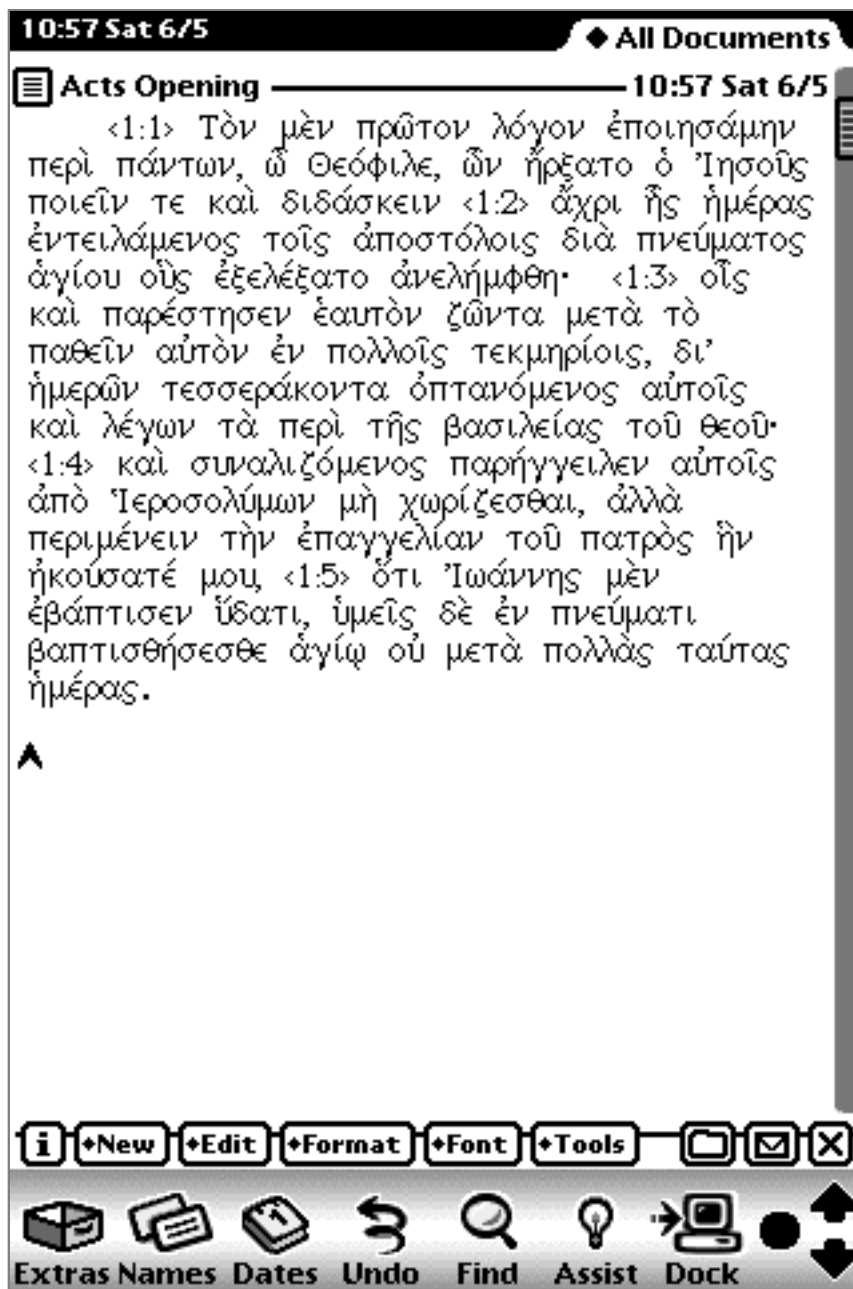
This is the best approach and is readily supported by both `Unicode` and OpenType. Alternately, one can simply use the combining diacritical marks in Unicode. `Unicode`, indeed, would be utterly useless to do mathematical and technical texts if it couldn't put a tilde or any other accent above an “z” as easily as above an “n”.

`<U+0065 LATIN SMALL LETTER E>`  
`<U+0301 COMBINING ACUTE ACCENT>`



**Figure 9. Approach C**

Again, there are alternate approaches.



**Figure 10. New Testament Greek on a Newton**

The simplest way to do this to simply give the accent zero width and a negative side-bearing. This approach doesn't strictly speaking even use the character-glyph mode, and so it is can be used by the Newton OS, for example. The result is shown in figure above, which shows the opening of the Acts of the Apostles drawn on a Newton using combining diacritical marks. The result is barely satisfactory for screen text and

would be entirely unsatisfactory for a printed book. Note, for example, how the circumflex tends to sit too far to the right on a lower-case omega.

and OpenType allow a font designer to define a way for an accent to attach to a particular letter. In essence, this allows the line layout system to move the accent appropriately to optimize the display. This allows for an algorithmic construction of a new glyph on-the-fly as it is needed and thereby provides the greatest flexibility.

But that flexibility may not be perfect. Even the most advanced algorithm may not do quite as good a job as a skilled type designer. Font designers would be best served by filling their fonts with large repertoires of accented letters.

There is one additional wrinkle for Greek: Greek has one letter (ς), which regularly changes its shape to " at the end of words—exemplary of the problem of contextual forms which are pervasive in other scripts. Unicode actually has the latter as a separate character: U+03C2 GREEK SMALL LETTER FINAL SIGMA. A system like the Newton OS has no choice but to use it.

AAT and OpenType both have the capacity, however, to form contextual forms as part of their regular glyph shaping. In OpenType, this is a matter of the 'fina' tag in the 'GSUB' table. In the case of AAT, it would mean using a feature of type 8 (smart swashes) and a word final swashes on/off selector.

would let you do the same thing by defining a `final U+03C3 GREEK SMALL LETTER SIGMA` that transforms a final `U+03C3 GREEK SMALL LETTER SIGMA` into `U+03C2 GREEK SMALL LETTER FINAL SIGMA` wherever it occurs at the end of a word. Since this would map to an existing Unicode code point, the problems of assuming that the font has the correct glyph at the correct place are minimized. We note that `final U+03C3 GREEK SMALL LETTER SIGMA` already does an exceptionally good job of distinguishing letters used in text and letters used in mathematical expressions, so the problem of accidentally turning ς into " in a formula is minimized, something that OpenType and AAT would have more trouble with.

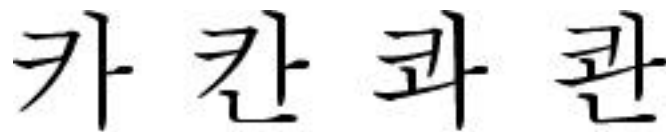
Thus we have two general solutions to the problem of contextual forms, which parallel those for accented letters or other glyphic combinations: map to an existing Unicode code point (either permanently in the text data or, preferably, just before rendering); or use data in the font to do an alteration of the glyph data after the initial character-glyph mapping.

## Problem number two: Conjoining Jamos

The Unicode Standard allows for the display of Korean using two entirely different systems. One involves the use of precomposed hangul syllables; the other conjoining jamos to make the syllables piece-by-piece. Interconversion between the two is simple and algorithmic.

Displaying Korean using precomposed hangul syllables presents no challenge; conjoining jamos, however, do.

Consider the four hangul syllables below:



**Figure 11. Ka, Kan, Kwa, and Kwan**

In each of these cases, the syllable starts with the character U+110F HANGUL CHOSEONG KHIEUKH. Its shape, however, alters drastically over the course of the sequence. In the first syllable, it has the left half of the glyph all to itself and can extend very nearly to the bottom. In the second, it has to become a bit more compact to make room for the U+11AB HANGUL JONGSEONG NIEUN that finishes the syllable. In the third hangul, the *khieukh* must shorter even further to stay above the U+1189 HANGUL JUNGSEONG U-A which crosses nearly the middle of the glyph; and when room is needed both for the final *nieun* and the double-vowel, the *khieukh* becomes cramped, indeed.

Clearly, the naïve approach used by the Newton OS won't work here. A single glyph for the initial *khieukh* would simply not work, either colliding with some other jamos or leaving half the glyph empty.

In this case, the correct solution is for the font to contain a *glyph* for the hangul syllable that will be drawn in place of the glyphs for the individual jamos. Access to this glyph can be provided by defining it as a ligature form for the individual jamos. With AAT, the correct font feature type to use would be *Ligature* (number 1) and the feature selector would be *Required* (number 0). AAT (through ATSUI) provides no alternative, as the model does no internal manipulation of the text data other than running the bidi algorithm.

Since *any* Korean font would support the same sort of mapping between conjoining jamos and precomposed hangul, the OpenType expectation would be that the OpenType client application would precompose conjoining jamos to hangul syllables before accessing the font data, rather than replicate this mapping as ligature table information within every Korean font.

would most reasonably do the same and use a            to map the conjoining jamos to precomposed hangul as an intermediate form before accessing the font data.

### **Problem number three: Biblical Hebrew**

Modern Hebrew is generally written without vowel points, but they are used in Biblical Hebrew. Fortunately, the problem of placing the vowel points is on the same order of complexity as the problem of placing accents for Latin, Greek, and Cyrillic. The main difference is that Unicode does not encode pointed Hebrew letters as precomposed forms, so that solution is unavailable without using the Private Use Area.

The main challenge of Hebrew in general is its requiring the use Unicode's bidirectional algorithm. This means that unadorned TrueType as implemented on the Newton OS cannot do it. This isn't so much a fault of TrueType as it is of the Newton OS: no bidirectional processing is done at all.

Hebrew highlights one of the main differences between OpenType and AAT/ATSUI. OpenType requires client applications to run the bidi algorithm themselves as a part of the linguistic processing common to any font representing a script.

AAT also requires a certain amount of bidi processing to be done by its clients. However, as it is encoding neutral and doesn't assume Unicode, it does much of the work itself and even allows for fonts to contain directionality information. A Unicode client of AAT would have to at least handle the Unicode directionality overrides and do initial assignation of levels based on code point; AAT would do the rest.

ATSUI as an AAT client and programmer's interface, however, *does* assume Unicode and therefore does *all* the bidirectional processing itself. An ATSUI client therefore need do nothing different for Biblical Hebrew than they did for accented Latin.

Two caveats: As with the final " in Greek, Hebrew has five letters that regularly take a different shape when at the end of a word: kaf (k/°), mem (m/μ), nun (n/ˆ), pe (p/ ), and tsadi (x/ ). Unlike Greek, these final forms occur *almost* always, but not quite always, particularly for languages other than Hebrew itself written with the Hebrew script, such as Yiddish. It is probably wisest *not* to do any automatic alteration of the non-final forms to the final ones in Hebrew.

It should also be noted that properly written Biblical Hebrew uses a number of sigla other than vowel points. From a technological standpoint for the character-glyph model, these do not alter the basic approaches to use; but they are important to remember for proper typography.

#### Problem number four: Devanagari and other South Asian scripts

South Asian languages represent—next to Arabic—perhaps the most complicated problems that rendering engines have to solve in order to display Unicode properly. *The Unicode Standard, Version 2.0* devotes twelve pages to describing the proper rendering of Devanagari. Letters are drawn in a different order from that in which they are typed; complex ligatures are formed; glyphs need to be inserted or rearranged. The actual amount of work necessary to translate a series of Unicode Devanagari code points into a series of Devanagari glyphs is enormous.

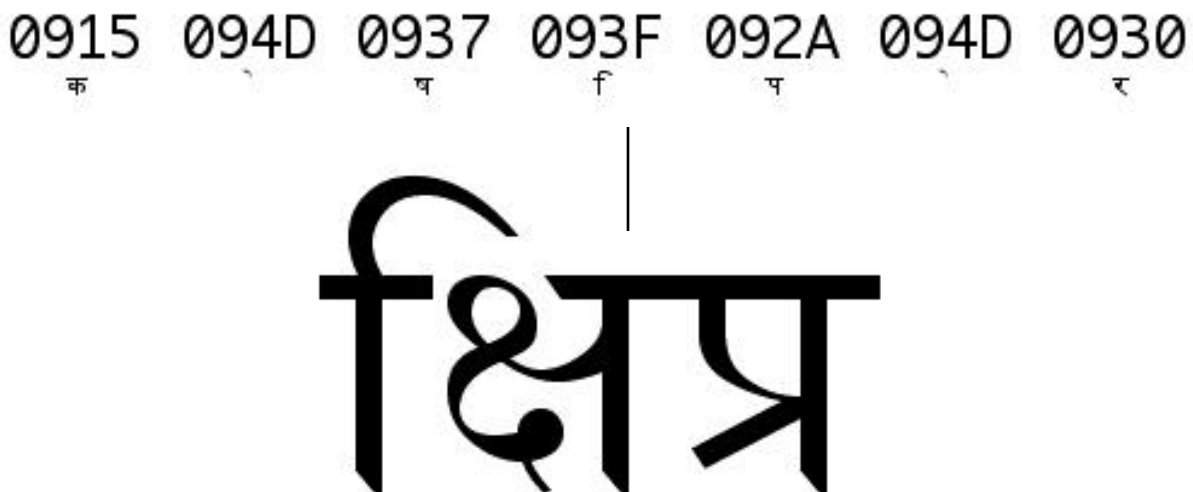


Figure 12. Devanagari

It should come as no surprise, therefore, that the Newton OS couldn't handle it at all<sup>4</sup>. What *is* surprising is that , OpenType, and ATSUI can do it already. The basic approaches already outlined can be used to handle all of the complex mappings that are required. The problem is in getting all of the details right for any particular font and script.

The story is slightly more involved, however for OpenType. OpenType doesn't only specify *what* fonts and applications must do in order to provide proper support for particular scripts or languages, it very often specifies details of *how* that is to be done. In the case of south Asian scripts, although basic support is in place the full details haven't been worked out as of yet. OpenType client applications would therefore need to do some reworking when those details are finalized.

#### Problem number five: Unihan Glyph Selection

This is a significant problem that text-drawing engines need to address. It derives from the fact that Unicode encodes characters, and not glyphs. The various writing systems of East Asia that use ideographs use a common set of ideographs—on this, everyone is basically agreed—but these ideographs are *drawn* differently in different places. A character may have one shape in mainland China, another in Japan, a third in Taiwan, and a fourth in Korea. These differences are often minor; they're on the order of American and British spellings for words like "color" or "center." A Chinese reader may find a Japanese glyph for a character odd-looking, but it would be recognizable.

Accordingly, the ideographic repertoire of Unicode (frequently called "Unihan" for short) unified the various typographic traditions as different reflections of a common set of characters. The difficulty is that actual text display should reflect the preferences of the intended readers. It would be unacceptable for a book published for Japanese to use Chinese typography.

The Unicode approach has always been to recommend fonts as the proper locus for solving the problem. Ideally, a user would select a font designed for use with Japanese when displaying or printing Japanese text. This requires information external to the text stream—namely the font, or ideally the font and locale—but even the simplest of text editors allow for multiple fonts nowadays, so this isn't an issue. Even the Newton could draw Japanese with Japanese glyphs and Chinese with Chinese glyphs


---

<sup>4</sup> Actually, there are rumors of a Devanagari support cobbled together for the Newton using a custom soft keyboard, the Private Use Area, and a fully precomposed set of glyphs—but that would be the only way to do it.

(assuming the user had enough memory for both fonts).  
PRC Taiwan Japan Korea

### Figure 13. Unihan Glyph Selection


A wrinkle has developed, however. The original assumption that type designers would produce and users utilise fonts designed for a specific region exclusively has proven untrue. Type designers want to produce a single typeface appropriate for use in Japan *and* Taiwan, not two typefaces, one for Japan and one for Taiwan. And users want to take advantage of these fonts.

Neither  nor the Newton OS could really handle the case where a different glyph is chosen from a single font based on locale. Both AAT and OpenType do support it. The simplest solution is one offered by ATSUI: a different 'cmap' subtable can be used based on locale tags set on the text. This allows a font to have one mapping of characters to glyphs for text intended for Taiwan and another mapping for text intended for Japan.



Beyond this, OpenType defines a series of feature tags (e.g., 'jajp', 'kokr') which a font designer could include to allow locale-sensitive selection of glyphs. AAT similarly provides a character shape feature type (20) which can be used to indicate a particular shape for an ideograph.

### Problem number six: Ideographic Description Sequences

In terms of the character-glyph model, Ideographic Description Sequences represent something of a solution in search of a problem. Unicode 3.0 will define twelve "Ideographic Description Characters" and define how these can be used to create Ideographic Description Sequences which describe unencoded ideographs. These descriptions are intended to be parsed by human beings and not automatically and are intended to give human readers a good sense of what is meant, rather on the order of the phrase, "a z with a squiggle on top."

Thus the unencoded ideograph 乾 would be described by an Ideographic Description Sequence that looks like  (an admittedly pathological example).

Unicode 3.0 will specifically *not* require any sophisticated display, UI interaction, or parsing on the part of systems using Ideographic Description Sequences. Even the Newton OS could handle them. At the same time, the standard will not *preclude* sophisticated display.

As a result, the ligature-formation power of OpenType, AAT, and  could be harnessed to provide support for specific instances of Ideographic Description Sequences in specific fonts. This is most complicated in the case of , as usual,

where coordination needs to be maintained between what expects the font to contain and what the font actually contains; but with OpenType and ATSUI, the support comes automatically.<sup>5</sup>

Such an approach would be useful for example in the preparation of a scholarly text of an ancient document containing unencoded ideographs. In this case, specific glyphs could be used in a customized typeface and generated by Ideographic Description Sequences in the text. Any well-written high end publishing application would then allow the unencoded ideograph to appear in the printed text.

(In theory, it should be possible to write an OFA2 scaler for use with AAT that can generate new glyphs for Ideographic Description Sequences on the fly. In practice, this would be very difficult and in any event, algorithmically generated glyphs rarely look quite as good as their hand-drawn or hand-tuned counterparts.)

Problem number seven: Surrogates

There are currently no characters encoded in Unicode using surrogates, but this is expected to change in the near future. WG2 is currently working on the definition of part 2 of ISO/IEC 10646. Part 1 defines the basic structure of ISO/IEC 10646, allocates a single plane, plane 0, the Basic Multilingual Plan, for use and encodes characters therein.

Part 2 doesn't alter the architecture of 10646, but it does define three new planes: the Secondary Multilingual Plane (SMP, plane 1), the Supplementary Plane for CJK Unified Ideographs (plane 2), and the General Purpose Plane (plane 14). The current proposal for ISO/IEC 10646-2 also begins the process of encoding characters within these three planes.

The SMP is initially populated with three alphabets (Etruscan, Gothic, and the Deseret Alphabet) and two sets of symbols (Byzantine Musical Symbols and Western Musical Symbols).

The Supplementary Plane for CJK Unified Ideographs will be initially populated with the CJK Unified Ideographs Extension B, a set of nearly 41,000 additional rare Han ideographs.

The GPP contains a small set of tag characters to be used by external protocols.

It is likely that within the next year or two, Part Two of 10646 will be approved by WG2 and the characters with surrogates would then be formally adopted as part of

---

<sup>5</sup> Since, however, East Asian scripts use ligatures but rarely, it may be that an OpenType client application would not do any ligature formation for East Asian text. Any ATSUI client or OpenType client that allows the formation of ligatures in East Asian text would have the required form show up once the font is installed.

the Unicode standard. Since some of the ideographs included in CJK Unified Ideographs Extension B are required for full coverage of JIS X 0213—an important new Japanese standard—actual support for characters encoded with surrogates will be needed as soon as they're formally encoded. The time is *now* for Unicode implementations to begin to prepare to support them.

Unfortunately, nobody is quite there yet. The *proper* way to support surrogates would be to directly map surrogate pairs to glyphs. `Font` doesn't have the concept of an arbitrary mapping from character space to glyph space; glyphs in the fonts for `Font` can't have indices beyond 65,536. This means that `Font` can't expand the surrogate pair to its corresponding scaler value and then use the scaler value as a lookup for the character's glyph—the scaler value exceeds this limit.

The TrueType specification also currently assumes that character codes run from `0x0000` through `0xFFFF`, although, Apple and Microsoft are actively working on extending the TrueType spec to allow the direct mapping of surrogate pairs to glyphs both as paired surrogates and as single scaler values.

Meanwhile other mechanisms will have to serve.

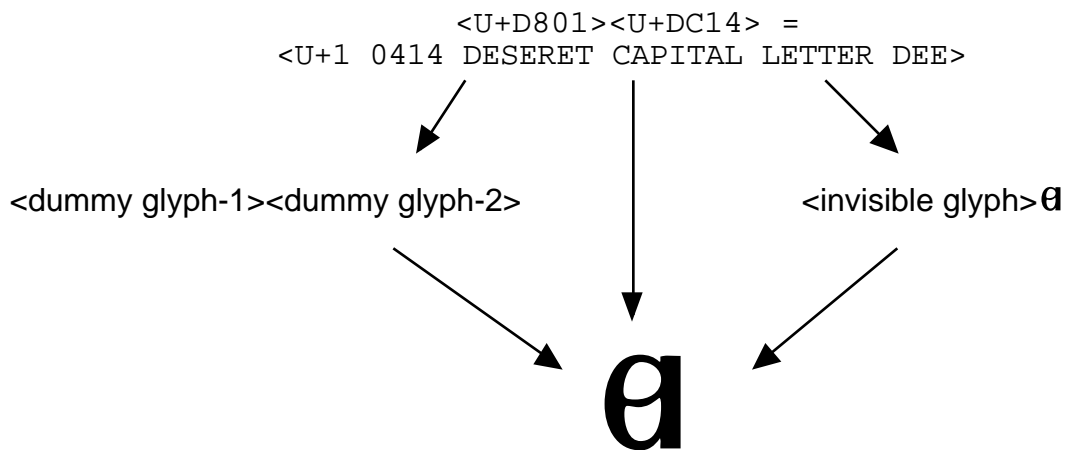
In OpenType, it would be possible to map the surrogate pairs into the Private Use Area. This is an ideal solution for small scripts like the Deseret Alphabet, which already has a semi-standard encoding in the Private Use Area; but it won't work for large collections such as the CJK Unified Ideographs Extension B. There aren't 41,000 code points in the two-byte Private Use Area.

Using ligatures is a possible solution. It has the advantage of being available now and will work. It does mean defining “dummy glyphs” for the intermediate stages in ligature formation, glyphs which would never be seen. One dummy glyph would be needed for each high surrogate and low surrogate covered.

Consider the case of the Deseret Alphabet, which is currently targeted for encoding from `U+10400` through `U+1044F`. This corresponds to a high surrogate of `U+D801` and low surrogates from `U+DC00` through `U+DC4F`. (Four of these code points are unassigned.) This means 77 dummy glyphs in the font—not an impossible overhead, but annoying. This becomes more difficult to do for the Western Musical Symbols block, which requires 221 dummy glyphs and very difficult for CJK Unified Ideographs Vertical Extension B. The 41,000 characters in Extension B would

require 41,000 actual glyphs in addition to some 2069 dummy glyphs (21 for high surrogates and 2048 for all the low surrogates).

One alternative does exist for fonts which provide coverage for scripts which never re-use low surrogates, that is, which doesn't cover a range of encoded characters using more than one high surrogate. A font designed for the Deseret Alphabet and no other surrogate-encoded script, for example, could give its single high surrogate glyph zero width and no outlines, and use the correct Deseret Alphabet shapes for its low surrogate glyphs.



**Figure 14. Handling Surrogates**

This approach would even work on the Newton OS—but it does put a severe limitation on the font to covering no more than 2048 characters encoded with surrogates, and it makes cursor movement and similar actions behave unexpectedly. A similar approach would be to take advantage of the ability of OpenType and AAT to use contextual forms to have glyphs vanish altogether. Instead of drawing the Deseret Alphabet letter `ᑕ` using a zero-width invisible glyph followed by the `ᑕ` glyph, the `'GSUB'` or `'mort'` table could simply direct that the glyph pair “`<invisible high surrogate glyph>ᑕ`” be transformed into the single glyph “`ᑕ`”, which will behave properly in the UI.



The current Pollard proposal, however, does not use precomposed characters; the character-glyph model allows us to encode Pollard using combining characters, and the ligature-formation capacities for `compositing`, AAT/ATSUI, and OpenType can adequately handle Pollard.

Pollard, however, illustrates one of the key differences between OpenType and AAT/ATSUI.

There is currently no consideration given in the OpenType specification for how to do Pollard. OpenType certainly allows for non-standard feature tags to be used, but there is the risk that OpenType client applications will not use them, or that the non-standard feature tag used now will be replaced by a different one later and that support would then be broken.

AAT, on the other hand, is more forgiving for experimental or non-standard features in fonts and less stringent that the specific details of the feature registry be followed. Once a standard encoding is available, a keyboard and a font would be all that is necessary; with those, any ATSUI-based application would automatically work and continue to work even in the “standard” way of using font features with Pollard were to change.

Of course, implementation of unencoded scripts is always risky, as the code points (if nothing else) will be changed if the script is ever formally adopted by Unicode, as will certainly be the case for Pollard. Nonetheless, less work would be involved for ATSUI client applications to provide support once Pollard is in the standard than would be the case for OpenType.

The same is true for `compositing`, which would require an alteration of the font and the `compositing` to use with Pollard. Here the delay would be relatively slight, as `compositing` already has the mechanisms in place to use the new data as soon as it is available.

## **4. Conclusion**

At first glance, the character-glyph model and the complexity of some of the scripts covered by Unicode would appear to create nearly insuperable obstacles for proper rendering; and it is certainly true that all the details for rendering complex scripts have not been ironed out in every case for existing systems.

Nonetheless, the industry has now reached a point where major platforms have solutions in place which can be used to handle even the most complex rendering problems that proper Unicode support would require.