Extensible Rendering for Complex Writing Systems

Sharon Correll
SIL International

1 Introduction

Those needing to work with multilingual text, particularly using any kind of complex script, commonly run into two problems: input and output. On the input side, it is often necessary to use keyboarding modules that map sequences of keys to the characters needed for the languages. On the output side, smart font rendering technology may be needed to properly display or print many complex writing systems.

In recent years several technologies have become available to provide solutions to both the input (keyboarding) and output (rendering) problems. The most well-known of these solutions are implemented as part of the operating system. These technologies are useful for the most widely-used languages of the world whose writing systems are standardized and well-documented. But the built-in solutions are often not appropriate for smaller linguistic groups whose writing systems deviate from the standard scripts.

This paper describes several technologies that can be used to provide extensible writing system implementations in situations where the solutions implemented as part of the operating system are inadequate. Specifically we will focus on Keyman, for keyboard input, and the Graphite system, for complex rendering.

2 Computers and complex scripts

Figure 1 shows how computers were originally intended to handle scripts. Because they were designed by Westerners, they historically have a simplistic model of text processing adequate to handle languages like English. It is assumed there is a one-to-one correspondence between keys presed on the keyboard, the character data stored in computer memory or in a file, and the glyphs displayed on an output device.

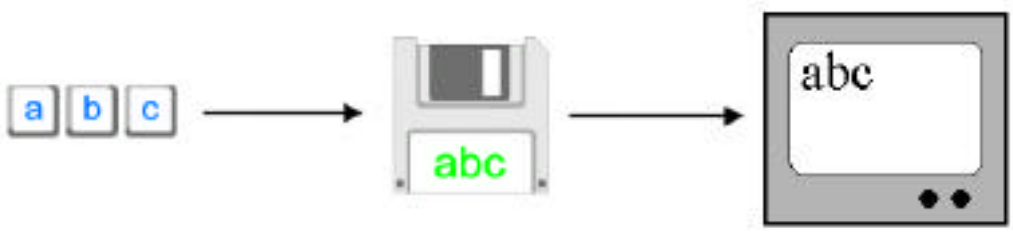

Figure 1: Processing model for simple Western scripts.

Anyone who works with non-Western scripts (and even some Western ones) quickly realizes the limitations of the simplistic model. Inadequacies appear in two areas. First there is the problem of data input: the characters on the keyboard may not match the characters needed by

the language of interest. This problem can often be solved by simply creating a new set of one-to-one mappings between the keys and the needed characters. However, it is frequently the case that there are more characters needed than there are keys present on the keyboard, and so more complex, multi-key mappings are required.

The second problem area—rendering—is more difficult to provide a solution for. Often the forms of the letter to be rendered depend on the position of the letter within the word, or the presence of other letters in the vicinity. Letter forms may need to be positioned in complex ways, including stacking and wrapping. The order in which the glyphs appear may not correspond to the order of the corresponding characters in the data, and characters may combine to form ligatures or split into two or more glyphs.

Because keyboarding has been the easier problem to solve technologically, it has been common to solve both the input and the output problem as shown in Figure 2. Complex keyboard routines are created to generate a sequence of charcter codes that can then be displayed in a straightforward way. This is called a *presentation form encoding*, because the codepoints in the data correspond exactly to the visual appearance of the rendered result. TSCII is an example of a presentation form encoding; one of the principles in its development was the desire to avoid the need for smart rendering.[1]

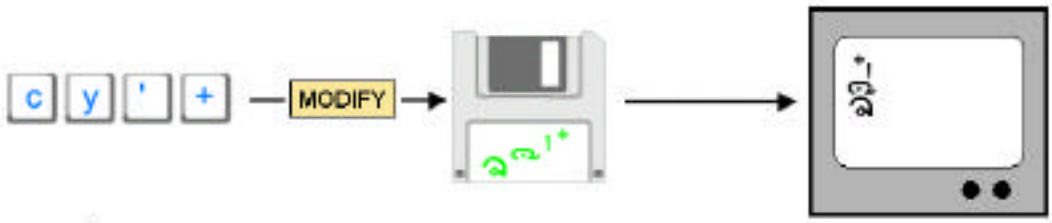

Figure 2: Processing model using presentation form encodings.

Although widely used, there are problems with the presentation form approach. One significant problem is that there are multiple ways of representing a single "grapheme"—a distinct script element from the perspective of the user. This greatly complicates searching, sorting, and other linguistically-based processes. For instance, if there are three or four forms of a given vowel, the process of finding all instances of this vowel must involve specifying all the forms. If several charcters combine to form a ligature, searching for the instances of one of the components of the ligature is difficult. Find-and-replace operations are made difficult by the need for contextual rendering, that is, the requirement that a specific form of a character be used depending on the neighboring characters. A further disadvantage is that when editing a body of text, more characters must be retyped in order to generate the correct forms.

Figure 3 shows a more ideal solution. In this approach, the encoding—the collection of codepoints representing the data—is defined in terms of an abstract set of characters, regardless of their final appearance. The use of a linguistically pure encoding greatly simplifies the various kinds of processing mentioned above. A smart keyboard exists to transform a sequence of keystrokes into these characters, and a corresponding smart renderer

[1] K. Kalyanasundaram. 1997. "A Proposal for A Tamil Standard Code For Information Exchange." Available on-line at www.tamil.net/tscii/tscii_draft.html.

generates the appropriate form for display. Unicode is an example of a linguistically pure encoding that fits this model.[2]

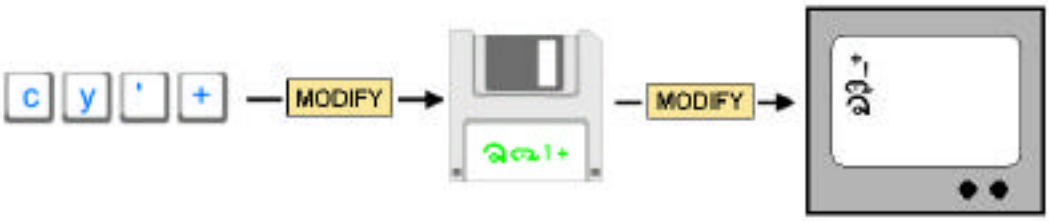

Figure 3: Processing model permitting a linguistically pure encoding.

3 Current technological solutions

A number of high quality "smart modules" are provided with various operating systems. On the Windows system, for instance, there are a wide variety of keyboards available for many common languages. On the rendering side, the Uniscribe rendering engine is built into Windows 2000 and later systems to provide complex rendering for a number of scripts. Similar technology is under development on the Linux system and is built into some Java implementations.

There are many languages, however, for which these hand-crafted operating system solutions are inadequate. The lesser-known languages of the world, generally spoken by ethnic minority groups, often use non-standard forms of the more well-known scripts. Because they represent a relatively small market share of the computer users of the world, and generally not an economically powerful share, it is not to the advantage of the corporate developers to create expensive, hand-crafted solutions for these lesser-known writing systems. In addition, many of these writing systems have not undergone a standardization process, preventing them from being supported in much commercial software.

Among those doing literacy development in many parts of the world, there is a need for experimental orthographies that can be progressively changed and improved as might be indicated by research and testing.

In summary, it is simply not feasible for modules built into the operating system to handle the wide variety of existing writing systems. Therefore, the minority language groups of the world need technologies for building customized keyboarding and rendering solutions. This technology needs to be accessible to ordinary programmers and users without the need to tweak or hack the operating system.

3.1 Customizable solutions

Fortunately, a handful of technologies exist that provide these customized solutions. On the input side there is Keyman (Keyboard Manager), produced by Tavultesoft. For complex rendering, the technologies include Apple's ATSUI and AAT (successor to GX) for use on the Macintosh, and Graphite, developed by SIL International and currently available on the Windows platform. An advantage of the Graphite system is its rule-based programming language which facilitates font development.

[2] The Unicode standard does include presentation forms for some scripts, but only for the purpose of backwards compatibility with earlier standards.

A question may arise about whether OpenType, developed by Microsoft and Adobe, can also provide this sort of customized solution. OpenType is indeed a customizable smart *font* technology, but it was not designed to handle the full range of complex writing system behaviors. It was designed on the assumption that an outer module (e.g., Uniscribe) would handle the needs of the writing system itself. So while OpenType can potentially be used to provide some complex rendering, it is not adequate for all situations.

The remainder of this paper will focus on Keyman and Graphite.

3.1.1 Keyman

The Keyman package is available from Tavultesoft and can be downloaded from the web site. It makes use of a rule-based programming language that maps sequences of keystrokes onto characters. The programming environment includes an integrated keyboard editor convenient for testing the effects of the program. Keyman version 5.0 can be used to generate both eight-bit and Unicode data.

Figure 4 shows a technical overview of Keyman. A Keyman keyboard is implemented via a program written in Keyman's programming language. This program is compiled using TIKE (Tavultesoft Integrated Keyboard Editor), which produces binary form of the keyboard, generally given a .KMX extension. This executable is installed into the Keyman system using the Keyman configuration program, from which the driver is initialized. When the Keyman driver is running, it intercepts keystrokes from the operating system and transforms them into characters using the active keyboard, then passes those characters on to the current application.

The modules shown within the dotted lines are those that are part of the Keyman system.

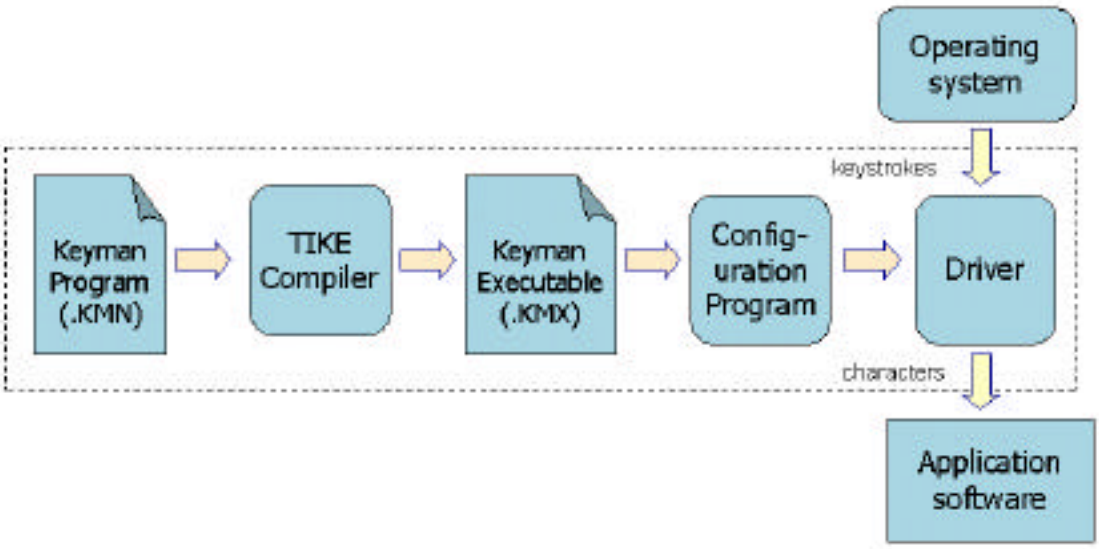

Figure 4: Technical overview of Keyman.

3.1.2 Graphite

Graphite is a relatively new technology providing complex rendering on the Windows platform. It handles a wide variety of complex rendering behaviors, such as contextual glyph selection, ligatures, stacking diacritics, complex glyph positioning, reordering. It also includes support for Unicode's bidirectional algorithm.

Figure 5 shows a technical overview of Graphite. A Graphite rendering implementation starts with a program written using Graphite's rule-based programming language, Graphite Description Language (GDL). The GDL program is compiled against a TrueType font, and the output is a copy of the font extended with several tables specifically intended for use by Graphite. This extended font serves as input to the Graphite engine. The engine can be hooked up as the back-end of a text-processing application, providing the service of generating rendered output on a computer screen or printed page.

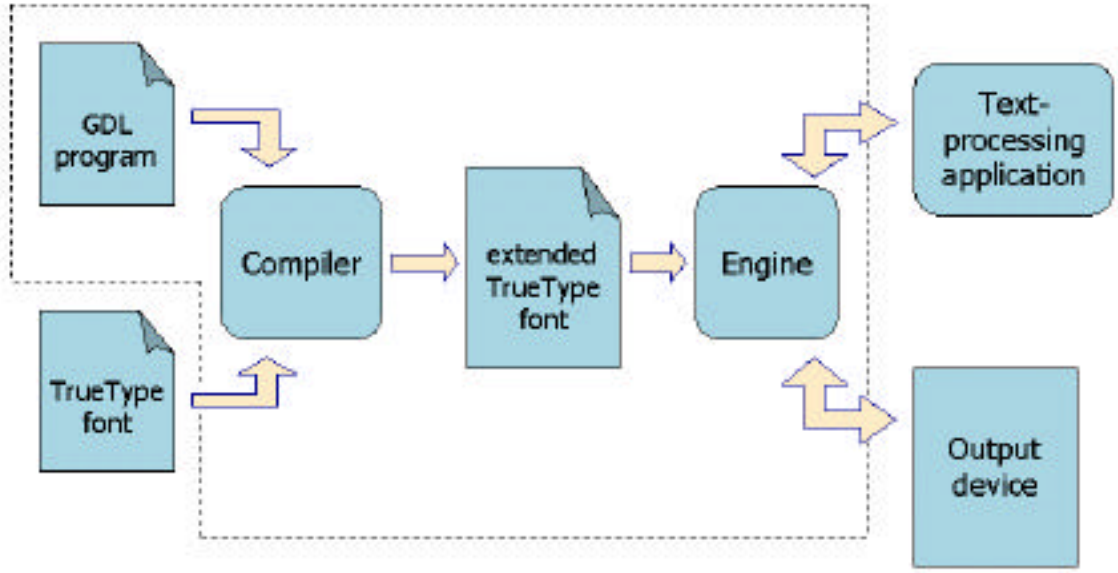

Figure 5: Technical overview of Graphite.

Graphite is a general purpose smart rendering system and thus can handle a wide variety of complex behaviors found in writing systems used in many parts of the world.

3.1.2.1 Contextual glyph selection

In a number of complex scripts, the exact form of characters take on when displayed is dependent on neighboring letters. Semitic scripts, such as Arabic and Hebrew, are particularly known for this, as shown in Figure 6. The black letters show differing forms of Hebrew consonants, where the form dispplayed depends on the position of the character within the word. Figure 7 shows a similar phenomenon in traditional Tamil (ORNL) script.

Figure 6: Regular and word-final letter forms in Biblical Hebrew: (a) kaf, (b) mem, and (c) nun.

Figure 7: Contextual glyph selection in Tamil (old-style).

3.1.2.2 Ligatures

In many complex scripts, a sequence of two or more adjacent characters can combine to form a ligature. In Graphite, it is possible to specify arbitrary rectangular portions of the ligature to correspond to the individual components. These components can then be selected and manipulated independently. Note that, unlike many complex rendering systems, the ligatures do not have to be side-by-side, and may visually overlap. Figure 8 shows some examples of ligatures. In Tamil, however, it can be challenging to find simple rectangular forms that correspond to the ligature components!

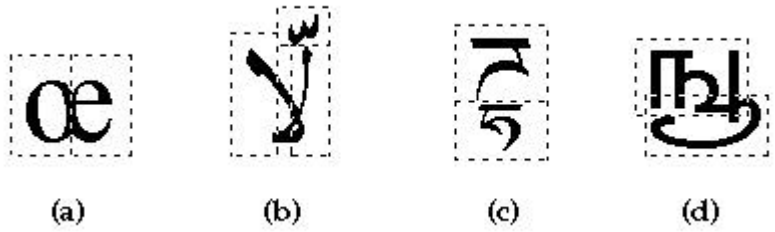

Figure 8: Ligature components in (a) Latin script, (b) Arabic, (c) Tibetan, and (d) Tamil.

3.1.2.3 Reordering

Reordering is a common phenomenon in scripts of South Asia, where the order of the characters in the data does not match the visual order of the corresponding glyphs. Figure 9 shows an example of reordering in Nepali, where in addition to the reordering that occurs within the cluster "ti," the "r" is also moved to the end of the word.

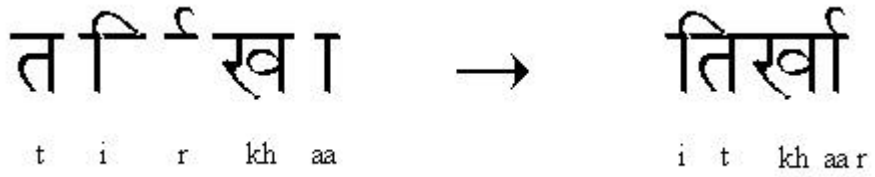

Figure 9: Reordering in Nepali.

Notice that this can result in discontiguous visual selections, as shown in figure 10. When the underlying characters indicated by the gray box in (a) are selected, the result is the visual highlight shown in (b).

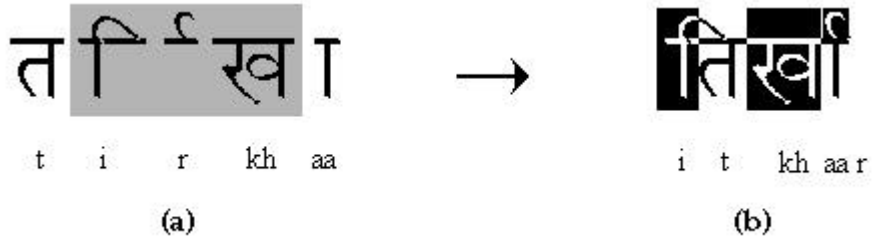

Figure 10: Discontiguous highlighting of reordered characters.

3.1.2.4 Split characters

Another common behavior of South Asian scripts involves characters that are rendered discontiguously. An example from Tamil is shown in figure 11.

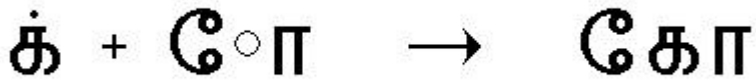

Figure 11: Split vowels in Tamil.

Graphite allows the ability to produce this kind of rendering with the behavior that all of the rendered glyphs are "aware" of their underlying association with the single character. As shown in figure 12, this means that dragging the mouse over either half of the split character (a) will result in the entire underlying character being selected (b), with both halves visually highlighted (c).

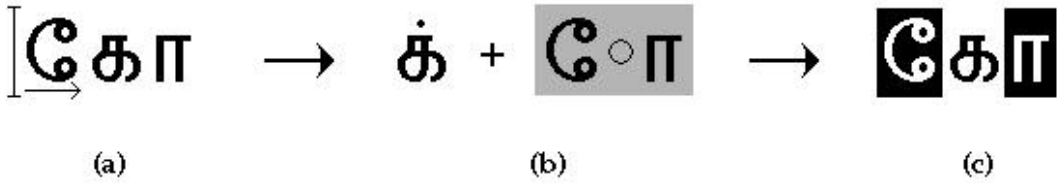

Figure 12: Selection of split characters.

3.1.2.5 Positioning

Positioning can be accomplished in two ways Graphite. The first way is by straightfoward shifting and kerning of glyphs, in either the horizontal or vertical direction (or both). The second way is by means of attachment points. Attachment points can be defined on glyphs, specifying for instance the locations of upper and lower diacritics. Then a simple rule is used to position the glyphs such that the attachment points coincide. Figure 13 shows possible uses of attachment points in scripts with multiple diacritics. The small cross-hatches indicate the attachment points.

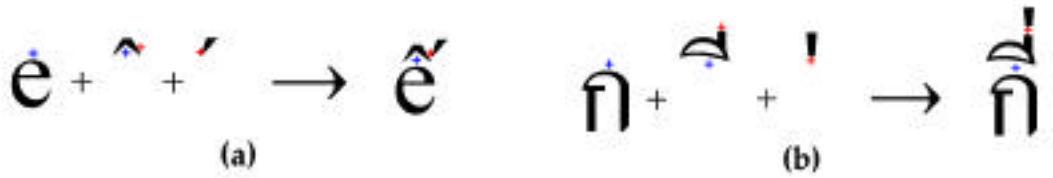

Figure 13: Attachment points in (a) Vietnamese and (b) Thai.

4 Creating customized script solutions

At this point we are going to work through an example of how one would use Keyman and Graphite to create a customized solution for a complex writing system with special character needs. For this example we are going to use Arabic, which has a number of interesting complex behaviors, and for which fairly complete Keyman and Graphite implementations exist. I expect that the principles can be extrapolated in a straightforward way to other scripts, such as Tamil.

4.1 Step 1: Extend the encoding

The first step in creating a writing system extension is to properly extend the encoding. In other words, it is necessary to define exactly which codepoints will be used to represent the special-purpose characters The encoding is the central element, because it establishes the

semantics and integrity of the data. Also, the keyboarding and rendering extensions are defined in terms of the encoding extensions, so the encoding must be established first.

Note that extending the encoding may mean adding new characters (for instance, to the Unicode Private Use Area), or combining existing characters in new ways.

Example
In our example we are going to add a retroflex "s" character to an Arabic-based writing system. We'll choose an arbitrary character from Unicode's Private Use Area, say U+E000[3].

4.2 Step 2: Extend the font

It maybe necessary to add glyphs to the font, new shapes that are needed to properly display the new character. This can be done using a commercial program like FontLab. Be sure to check the license for your font to be sure you have permission before modifying a font.

Example
Our new retroflex-s character will be visually simlar to the Arabic *seen* character which is used for the "s" sound, but will have four dots above it. Moreover, as is typical in Arabic script, we will need four forms of the new character shown in figure 14: an isolate (stand-alone) form (a), a word-initial form (b), a word-medial form (c), and a word-final form (d). In the font's cmap, we will map our Unicode codepoint U+E000 to the isolate form, which parallels the way the rest of the characters in the font are handled. This means that when no smart rendering is occuring, the isolate form is the one that will be displayed. It can also be helpful, although not required, to define postscript names for the new glyphs.

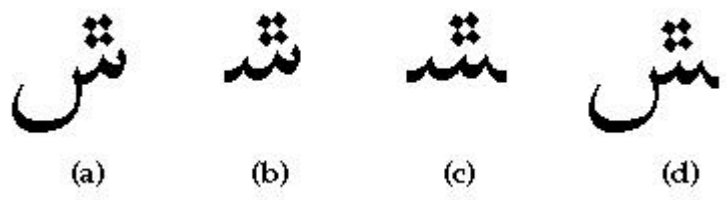

Figure 14: Retroflex seen *forms for Arabic example.*

4.3 Step 3: Extend the keyboard

The next step is to extend the Keyman program to handle the new characters. This typically involves adding a new rule to the keyboard program, recompiling it, and reinstalling the resulting keyboard executable.

Example
The key sequence we are going to use to indicate a retroflex *seen* is "s~". There is already a rule present in the program that converts the keystroke "s" into an Arabic seen (U+0633), as follows:

```
+ 's'  >  U+0633
```

(The blank space to the left of the plus sign means the preceding context to be taken into consideration is null, that is, the substitution happens regardless of what was previously

[3] In Unicode, codepoints are typically represented by four digit hexidecimal numbers, preceded by "U+" to indicate the Unicode encoding. The Private Use Area is a range of characters specifically reserved for user assignments.

typed.) The new rule we add to the program says that when the immediately preceding context is a *seen* (as generated by the first rule), and the user types a "~", we convert this sequence into a our new retroflex *seen* character, U+E000:

```
U+0633  +  '~'  >  U+E000
```

After adding this rule to the Keyman program and recompiling, we can type a retroflex *seen* into our data. Since a basic form of the glyph has been added to the font and mapped to the character, we can see that glyph displayed on the screen. However, the proper contextual form will not be rendered; in figure 15, notice the lack of cursive connections between the retroflex *seens* and the neighboring (gray) characters. To provide for proper display we need to extend our Graphite rendering module.

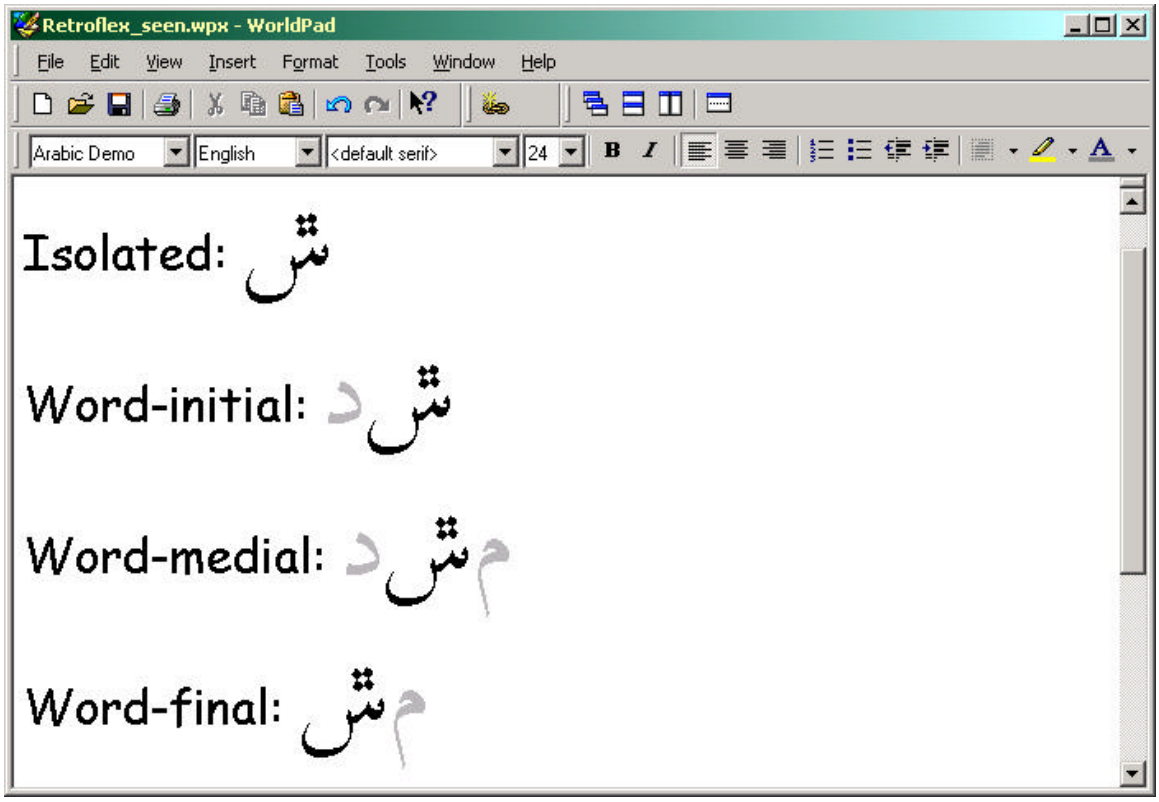

Figure 15: Rendering of the retroflex seen *after step 3.*

4.4 Step 4: Extend the renderer

Extending a GDL program involves a little more work than extending a Keyman program. First, certain attributes must be set for any new characters that are not part of the Unicode standard. These attributes include breakweight, which is used to determine appropriate line-break locations, and directionality, used in applying the Unicode bidirectional algorithm (needed for bidirectional scripts like Arabic). For standard characters, these attributes are normally read from a Unicode character properties database, but of course this is not possible for non-standard characters, so the values must be stated explicitly.

Apart from line-breaking and directionality, there are several ways to add new glyph rendering behavior to the program. GDL is a rule-based programming language, so the most obvious extension is to add a new rule to the body of rules that define the renderer's behavior. But it might be possible to use a simpler approach: assign the new glyphs to classes of glyphs that already exist. If the existing rules have been written in terms of glyph classes, the newly

added glyphs will automatically take on the behavior already set up for the other members of the class.

A final extension that might be needed is to set up attributes for the new glyphs such as those that specify attachment points or other positioning information.

Example

For our example, we define the four retroflex-*seen* glyphs within the GDL program. Because the character we are adding is a word-forming letter, we don't need to set the breakweight attribute; word-forming letters are the default. However, we will need to inform the program that this letter is a right-to-left Arabic letter, so it will be handled properly by the bidirectional algorithm. This is done by setting the directionality attribute. The GDL statements to define the new glyphs are as follows:

```
seenFourDotsAboveIso  =  unicode(0xE000) // default is isolate
     form
     { directionality = DIR_ARABIC };

seenFourDotsAboveIni = postscript("seenFourDotsAboveIni")
     { directionality = DIR_ARABIC };

seenFourDotsAboveMed = postscript("seenFourDotsAboveMed")
     { directionality = DIR_ARABIC };

seenFourDotsAboveFin = postscript("seenFourDotsAboveFin")
     { directionality = DIR_ARABIC };
```

The `postscript` function accesses the glyph with the given Postscript name in the font, while the `unicode` function accesses the glyph that the given Unicode value maps to within the cmap. (It is also possible to refer to a glyph directly using its glyph ID number.)

The most fundamental behavior to set up is the selection of the correct contextual shape—isolate, word-initial, word-medial, or word-final—as the character's position in the word requires. But we already have rules in our GDL program that provide this basic behavior. So our main task will consist of adding each of the four glyphs to the appropriate class.

In the glyph definitions above, we gave our four glyphs user-friendly names. So we can simply add these names to the list of glyphs that comprise the contextual classes:

```
dualLinkIso = (yehIso, behIso, tehIso, ...,
     seenFourDotsAboveIso);

dualLinkIni = (yehIni, behIni, tehIni, ...,
     seenFourDotsAboveIni);

dualLinkMed = (yehMed, behMed, tehMed, ...,
     seenFourDotsAboveMed);

dualLinkFin = (yehFin, behFin, tehFin, ...,
     seenFourDotsAboveFin);
```

Note that the items within the three classes for substitution are in corresponding order (yeh, beh, teh, etc.), and they must be kept so in order to achieve the correct behavior. On the other

hand, it is not necessary for us to make any changes to the body of rules; the existing rule will handle the newly added "seenFourDotsAbove" in exactly the same way it handles the other characters needing contextual shaping.

Ideally, we'll also define some basic attachment points which indicate how to position associated diacritical marks:

```
seenFourDotsAbove { TopCenter = point(360m, 705m);
     BottomCenter = point(300m, -600m) };

seenFourDotsAboveIni { TopCenter = point(355m, 980m);
     BottomCenter = point(450m, -150m) };

seenFourDotsAboveMed { TopCenter = point(430m, 980m);
     BottomCenter = point(450m, -150m) };

seenFourDotsAboveFin { TopCenter = point(350m, 600m);
     BottomCenter = point(350m, -600m) };
```

The `TopCenter` point is used to position marks above the glyph, and the `BottomCenter` point is used to attach marks beneath. The attachment points are defined in terms of units in the font's em-square. Again, rules already exist that recognize the situation where marks occur and perform the needed attachments.

After the modifictions to the GDL program have been completed, the program is recompiled to create a new version of the Graphite-enabled font. Now it is possible to see the proper contextual forms rendered within our application, as shown in figure 16.

Figure 16: Correct contextual forms in Arabic, rendered with Graphite.

5 Status of Graphite

The Graphite software is currently available in beta version 0.8. It is currently implemented only on the Windows platform, and there is one basic text editing application that can use Graphite for rendering. Because Graphite is not implemented as part of the operating system, applications must specifically be written to use its interface, and therefore Graphite is not available within standard commercial applications.

The Graphite package code is available through open-sourcing, and the development team is very interested in working with others who are interested in using it as part of their application or making extensions and improvements.

Currently there is serious interest in porting Graphite to the Linux platform, and also some interest has been expressed in seeing a Java implementation. Integration into various kinds of applications such as word-processors, publishing programs, and Internet browsers would make the Graphite technology more useful in a wide variety of situations. Other possibilities for open-source development include extensions to the Graphite system itself and the development of a programming environment that would make Graphite font development a more productive process.

6 Summary

The Graphite and Keyman technologies provide means by which writing system implementations can be provided for those language communities who use lesser-known and non-standardized writing systems for which there is no support provided by the operating system. In many cases, the extensions needed to be made to Graphite and Keyman programs represent, not months and years of development, but effort on the order of hours or days. The modest level of resources required puts this technology within the reach of the communities who have need of it, as well as agencies who may be working with them in areas such as anthropological research and literature and literacy development.

Without this level of extensible support, many language communities around the world will remain firmly on the opposite side of the “digital divide,” and their literature and culture will be largely inaccessible to those on this side of the gap who could benefit from it.

For these reasons, it is important that this kind of extensible technology be made available on a large number of platforms and be integrated into a wide range of applications.

7 For more information

For more information on Graphite and Keyman:

- Visit the following web sites:
  - SIL International: www.sil.org
  - Graphite: : graphite.sil.org
  - Tavultesoft: www.tavultesoft.com/keyman
- Sign up on the Graphite mailing lists: graphite.sil.org/mailman