

Transforming Characters to Glyphs in IndiX

Project IndiX

<http://www.ncst.ernet.in/projects/indix/>

1 Introduction

The Unicode standard [2] and ISO/IEC 10646 not only define the character codes, but also makes recommendations for necessary text operations like input order, rendering on display and paper, editing text, sorting and searching and compression. The passage from text characters to a sequence of glyphs that can render the text is at the heart of the rendering and editing operations.

The pipeline for an Indic script that takes in a sequence of Unicode characters and finally delivers a sequence of glyphs is not simple. Broadly, the pipeline has to reorder the characters, map the characters to the glyphs domain and then combine or, choose alternate glyphs and position the glyphs. The first step is in the character domain and the second is in the glyphs domain. OpenType fonts have been found suitable for the the glyph level processing. The technology, pioneered by Microsoft, is analysed in the light of the operational model for characters and glyphs, put forth in [4]. The model is converted to an object model. The IndiX model has many advantages over the Microsoft model.

2 The character to glyphs interface

For presentation processing [4], the text characters in the logical order (input order) are converted to a sequence of glyphs which will correctly render the syllable. An example, in the Devanagari script, will help to introduce the issues and solutions before we introduce the object model.

2.0.1 The *RKRI* example

The logical order of the six characters in the sample syllable is $\langle \text{Ra, Virama, Ka, Virama, Ra, Sign.I} \rangle$. The logical order is shown Unicode points in the first line and with their glyph forms in the second line in Fig 1. This character sequence or the glyphs sequence has to be reordered, and single or multiple glyphs have to be substituted with other glyphs. Finally a set of three glyphs is obtained on which no further processing is required. The final set of glyphs for the sample syllable is shown in the third line when they are rendered separately. The fourth line shows how the syllable will finally look like when these three final glyphs are rendered together.

```

<0930 094d 0915 094d 0930 093f>
<र     क     र     ि >
< ि     क्र     >
क्रि
    
```

Figure 1: Characters and glyphs. The *RKRI* example in Devanagari

The first two characters $\langle 0930, 094d \rangle$ have created the *reph* mark $\overset{\frown}{}$ and it is the last in the final glyph sequence. Similarly the last character $\langle 93f \rangle$ (or the $\overset{\frown}{\text{ि}}$ glyph) is responsible for the creation of the first of final glyphs $\overset{\frown}{\text{ि}}$. So there are two instances of reordering in this sample syllable. This is only an example and you should not jump to the conclusion that the glyph from the first few characters will always go to the end and the glyph from the last character will always move to the beginning! The character sequence $\langle 0915, 094d, 0930 \rangle$ is responsible for the creation of the middle glyph $\overline{\text{क}}$ in the final sequence. This is a result of substitutions on multiple glyphs.

Note the difference in form of the $\overset{\frown}{\text{ि}}$ the last on second line and the $\overset{\frown}{\text{ि}}$, the first on the third line. This is an instance of an alternate form being chosen. In this case the proper sign.I has been chosen so that its 'hook' falls on the stem of the $\overline{\text{क}}$.

3 A model for characters and glyphs and two implementations

3.1 The ISO/IEC 15285 model

The ISO/IEC Technical report 15285 [4] makes a clear distinction between characters and glyphs. The important points in the report, highly relevant and applicable to Indic scripts are:

1. The most general mapping from characters to glyphs is the mapping from a sequence of characters to a sequence of glyphs.
2. The mapping cannot easily be implemented by a table in the font that maps characters to glyphs. An Intelligent Font that has additional information on *how a sequence of coded characters is transformed into a sequence of glyph identifiers, with associated position information* is needed to implement the many-to-many mapping.

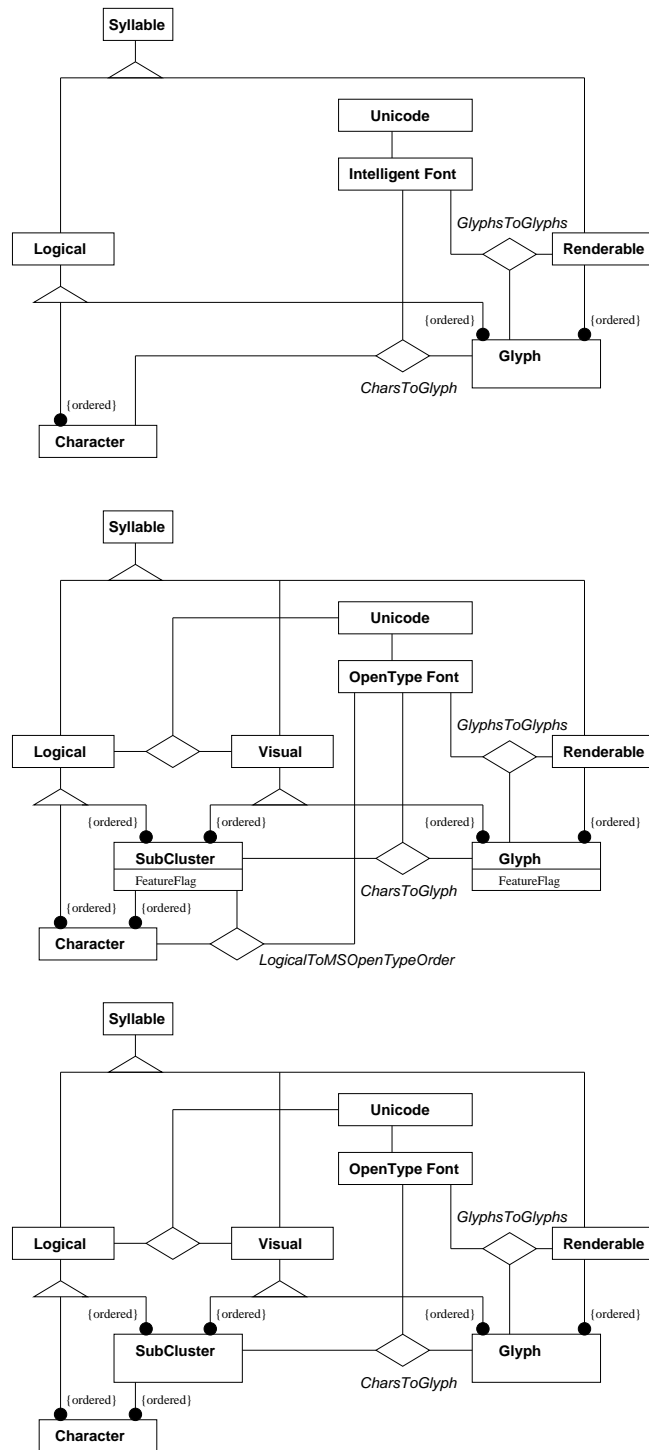


Figure 2: Object Model for characters and glyphs. (Top: ISO TR 15285, Middle: Microsoft Typography, Bottom: IndiX)

The object model of the ISO/IEC TR 15285 is shown in Fig. 2 top.

3.1.1 Syllable

We refine the sequence of characters that map to a sequence of glyphs to be a syllable. The interactions between characters will be confined to be within the syllable. A character within a syllable will not move to or change the shape of the preceding or succeeding syllable. A pipeline that converts from characters to glyphs will have to be presented with nothing short of a syllable. If the pipeline has to handle more than one syllable in the input sequence, it will have to be carefully designed to prevent unnecessary interaction between the character of a syllable and the next syllable. So, for simplifying its design, the pipeline will handle one and only one syllable at a time. A text containing multiple syllables will have to be broken down to the individual syllables by a syllable iterator [5] before being fed to the pipeline, syllable by syllable.

A **Syllable** can be **Logical** syllable or a **Renderable** syllable. Further, a **Logical** syllable can be an ordered sequence of **Character**, or an ordered sequence of **Glyph**. A **Renderable** on the other hand is an ordered sequence of **Glyph**.

Why should we differentiate a logically ordered character sequence object from the logically ordered glyph sequence object? Why not discard the three derived objects (Logical ordered **Character**, Logical ordered **Glyph**, and Renderable ordered **Glyph**) and consider the representational choice as an internal implementational matter of the **Syllable** object?

Consider a **ComplexNumber** object. The internal representation of the complex number can be a pair (x, y) denoting the value of the number to be $x + iy$. Or it can be a pair (r, θ) denoting the number as $r\angle\theta$. For adding two complex numbers, the (x, y) form is simpler whereas for multiplication the $r\angle\theta$ representation is better. Do we then maintain a **XY** object and a **RTheta** object derived from the **ComplexNumber** object? No. In most implementations, the representation is an internal matter for the object, which it manages transparently and dynamically depending upon the operations invoked on it.

Similarly, let the internal representation of the syllable be a sequence of logically ordered characters, or renderable glyph sequence, decided by the object depending on the operations invoked on it. At first, this viewpoint looks attractive, but it fails to make a crucial distinction between the two situations. The form of the internal representation of a **ComplexNumber** is independent of external objects or even the operations invoked on it. The internal form is changed by the object only if the implementation wants to optimize its behaviour. But the internal representations

of the **Syllable** are dependent on the **Intelligent Font** object and not something entirely decided by the **Syllable** alone. Hence, Logical ordered **Character** *is-a* **Syllable**, Logical ordered **Glyph** *is-a* **Syllable**, and Renderable ordered **GLyph** *is-a* **Syllable**.

3.1.2 The Intelligent Font

The form and function of the **Intelligent Font** can be surmised from the relationships it has with the other objects. Since, it is related to **Unicode**, it will have a CharMap from Unicode points to its glyphs, and these glyphs will have the form as indicated by the Unicode standard. This is also clear from the ternary *CharsToGlyph* relationship between the **Character**, the **Glyph** and the **Intelligent Font**. Functionally, an instance of the Logical ordered **Glyph** syllable is created from an instance of the Logical ordered **Character** syllable, character by character by using the CharMap from the instance of the **Intelligent Font**.

The **Intelligent Font** enters into another ternary *GlyphsToGlyphs* relationship with the **Glyph** and the **Renderable** ordered Glyph. Functionally, what this means is that the the glyphs in the ordered **Glyph** syllable are reordered, combined, replaced and positioned using tables, rules and routines associated with the **Intelligent Font** to create the **Renderable** syllable.

3.1.3 The *RKRI* example

For the Devanagari sample syllable shown in Fig 1, the line one is the instance of Logical ordered **Character** syllable, line two is the instance of Logical ordered **Glyph** syllable, and line three is the instance of Renderable ordered **Glyph** syllable.

3.2 The Microsoft Typography implementation

The Microsoft Typography approach will be explained using the object model in Fig. 2 middle and the *RKRI* example.

3.2.1 The visually ordered syllable

An important point about the **Intelligent Font** is that it should have the ability to reorder **Glyphs**:- to convert them from **Logical** order to **Renderable** order. Microsoft Typography implemented the **Intelligent Font** using **OpenType Font** technology (Fig 2 middle). In OpenType, it is possible to substitute a sequence of glyphs with another *single* glyph. It is not possible to substitute with another sequence of glyphs. Hence, it is not possible to *reorder* a glyph sequence using OpenType technology. Microsoft Typography hence specified a **Visual** syllable object between the **Logical** ordered Character syllable and the **Renderable** ordered Glyph.

The reordering rules in the Unicode standard are in terms of the glyphs. In any concrete implementation, the glyphs have to be denoted or encoded. Denoting a glyph by its index in the font glyphs table would be an impractical solution. Microsoft Typography chose to denote a glyph by the Unicode character that gives rise to the glyph. They introduced a reordering component external to the OpenType technology. Instead of reordering the glyphs, this component would reorder the characters.

Unfortunately, there are some simple glyphs which cannot be attributed to a single character. With Indic scripts, especially for what are called consonant marks (reph or vattu), a subcluster of characters gives rise to the consonant mark. So a consonant mark glyph is denoted by a subcluster of characters and not by a single character. So the reordering will be on the subclusters. The four Logical **SubClusters** for the *RKRI* example are shown in Fig. 3 between vertical bars.

<| 0930 094d | 0915 | 094d 0930 | 093f |>

Figure 3: Logical ordered subclusters of characters. The *RKRI* example

The **Logical** syllable of ordered subclusters is reordered as per the **Unicode** standard to form the **Visual** syllable of ordered subclusters. Functionally, the Logical **SubClusters** are repositioned. The four Visual **SubClusters** of Character for the *RKRI* example are shown in Fig. 4 between vertical bars.

<| 093f | 0915 | 094d 0930 | 0930 094d |>

Figure 4: Visual ordered subclusters of characters. The *RKRI* example

3.2.2 Subcluster according to OpenType font

After a **SubCluster** of ordered Characters is formed and repositioned within the syllable, Microsoft Typography took a peculiar step. The order of characters in some **SubClusters** is changed. The new order of characters within a subcluster is determined by the substitution rules within the **OpenType Font**. Since two distinct subclusters with different character order could be reordered within the subcluster to the same character sequence, the need arose to distinguish between these two subclusters by using the flags associated with the substitution rules. So the Flag associated with the applicable substitution table in the **OpenType Font** is set in the a FeatureFlag of the **SubCluster**. Hence, the **OpenType Font** is the third party to the ternary *LogicalToMSOpenTypeOrder* relationship between the Logical ordered **Character** SubCluster and the **SubCluster** ordered and flagged according to the **OpenType Font**. In the *RKRI* example, the third Visual **Subcluster**, the vattu, is reordered. The flagged Visual ordered **Subclusters** of characters are shown in Fig. 5.

```
<| 093f | 0915 | 0930 094d | 0930 094d |>
    | pres | vatu | blwf+vatu | rphf          |
```

Figure 5: Visual ordered subclusters of characters according to Microsoft Typography. The *RKRI* example

3.2.3 The syllable of visually ordered glyphs

The OpenType Font enters into relation *CharsToGlyph* between Visual ordered **Subcluster** of Character and Visual ordered **Glyph** as well. The passage from the **Character** domain to **Glyph** domain in the ISO TR 15285 model was controlled by the **Intelligent Font**. Microsoft Typography used the **OpenType Font** as the **Intelligent Font**. Instead of **Character**, they use flagged **SubCluster** of Character. If the **SubCluster** has a single character, then the **Glyph** will be formed using the Unicode CharMap in the font. If there are many characters, then each character in the **SubCluster** is first mapped to a glyph using the CharMap and then the glyph sequence is replaced with a single glyph using a substitution (GSUB) rule in the **OpenType Font**. The two step process should be abstractly viewed as a mapping from the **SubCluster** of character to a glyph. Thus a Visual ordered **Glyph** Syllable is created.

The first set of Visual ordered **Glyphs** for the *RKRI* example are shown in Fig. 6.

| | | | | | | | | | | | | |
|---|--|------|--|------|--|-----------|---|------|---|---|--|---|
| < | | ँ | | क | | र | \ | | र | \ | | > |
| | | pres | | vatu | | blwf+vatu | | rphf | | | | |

Figure 6: Visual ordered Glyphs(First step). The *RKRI* example

The second set of Visual ordered **Glyphs** for the *RKRI* example are shown in Fig. 7.

| | | | | | | | | | | |
|---|--|------|--|------|--|-----------|--|------|--|---|
| < | | ँ | | क | | ^ | | ^ | | > |
| | | pres | | vatu | | blwf+vatu | | rphf | | |

Figure 7: Visual ordered Glyphs(Second step). The *RKRI* example

3.2.4 The OpenType Font

The OpenType standard [7] decomposes the many to *many* mapping from glyphs to *glyphs* in terms of a computational process involving the application of a sequence of many to *one* mapping from glyphs to *glyph*. An OpenType font has ordered substitution rules organized into what are called GSUB tables. The tables are ordered and tagged with flags. The substitution rules are in terms of glyph indices and not Unicode character points.

An extract of the substitution tables from a Devanagari OpenType font organized according to the recommendations of Microsoft Typography is shown in Table 1. In the extract, the substitution table named Vattu Variants is associated with a flag *vatu*. Two substitutions are grouped under the *vatu* flag. A full OpenType font will have many more lookup tables and many substitution rules in most tables as indicated by the vertical dots (:) in Table 1.

The conversion from glyph sequence to the glyph sequence has several stages.

1. When the input character string is parsed, and the visual ordered syllable of glyphs is formed, we know which glyphs correspond to Reph, which correspond to pre-base mark and so on. A glyph is associated with the corresponding Flag by setting it in the FeatureFlag of the glyph.
2. Substitute glyphs and glyph sequences using the GSUB tables in the order of the tables and the rules within each table. If the left hand side of a substitution rule matches a substring in the input glyph sequence, the context

| Lookup table name | Flag | Substitution rule (GSUB) | | | |
|-----------------------|------|--------------------------|---------------|------------------------------|--------------|
| | | context before | source glyphs | context after | target glyph |
| Reph | rphf | | र ˘ | | र̣ |
| Below-base Form | blwf | | र ˘ | | र̣ |
| ⋮ | | | | | |
| Vattu Variants | vatu | | क ˘ ठ ˘ | | क़ ठ़ |
| ⋮ | | | | | |
| Pre-base Substitution | pres | | रि रि | { क़, क, ... } { र, ... } | रि रि |
| ⋮ | | | | | |

Table 1: A Devanagari OpenType Font sample following Microsoft Typography

of the rule is satisfied in the string around the substring and the substring has *the Flag associated with the rule set on all its glyphs* then the matched glyphs in the substring are replaced with the target glyph specified in the rule.

3. The substitution rules will be applied in sequence to the initial glyphs. A glyph pattern in the middle may get substituted by another glyph. A new pattern will be formed and it may take part in a substitution specified by a *subsequent* rule.
4. When the last rule in the GSUB tables has been tried, the process terminates. The glyph sequence, if rendered now, will be correct but for the positioning of marks.

The requirement that the Flag associated with a substitution rule has to be set on all glyphs of the input substring may lead to two problems. First, derived from *has to be set* is that the OpenType font will have to be standardized. The Flags should be defined and registered. That is why Microsoft Typography has registered features like *rphf*, *blwf*, *pres* and so on. The parser will have to follow the OpenType standards and set the correct flags on the input glyph string. Second, derived from *on all glyphs* is that the flags can be set only on the initial sequence of glyphs (Visual ordered Glyph syllable) and cannot be set directly on the intermediate

glyphs formed during the application of the substitution rules. Since the only way of setting a flag on an intermediately generated glyph is to set the flag on all the source glyphs that generated the intermediate glyph, we may have to locate the leaf glyphs of a substitution rule and set the flag on all of them. Unless the OpenType font is known or the substitution tree is standardized, this will be an impossible task. One way out is to state the substitution rules such that each rule is distinct. Two substitution rules will differ either in the source glyph string or the contexts. The flag set on the input glyphs is then redundant. Then we can safely set `allf` (denoting all flags *O*Red together) with the input glyphs and only the matching substitution rule will be activated. When a substitution happens, the target glyph inherits the `allf` of the source glyphs and so the chain of substitutions continues.

If two substitution rules are identical, then there is no way out other than setting the specific flag on the input glyph substring. Unfortunately, that is not the end of the story. Other undesirable coupling between setting of `allf` for glyphs and specific flags with some glyphs will be illustrated with the *RKRI* example.

In addition to GSUB tables, there are tables (GPOS) that help in relative positioning of glyphs. Instead of pairwise kerning tables, GPOS tables reduce the number of rules by specifying attachment points to glyphs. For Indic scripts, the consonant glyphs can have above base, below base, pre-base and post-base attachment points. The signs that attach to other glyphs (matras) have a corresponding attachment points. When a consonant is followed by an above base vowel sign, the GPOS tables are searched for their above base attachment points and any correction required to align them is effected. If there are m consonants and n vowel signs, the GPOS tables help to reduce the problem of positioning them to a problem of $m + n$ space complexity. The pair-wise kerning approach, on the other hand, would be of space complexity mn .

3.2.5 The Renderable ordered Glyph syllable

Technically, rendering the Visual ordered **Glyph** syllable will produce reasonably correct display, but it will be difficult to read and aesthetically and culturally lacking. For the *RKRI* example, if the glyphs in Fig. 7 are displayed together, the syllable will look like किं.

Some visual glyphs will have to be combined to form ligatures, some replaced with other forms and some marks positioned with respect to the base glyphs. The Unicode standard has a good list of ligature forms, but it does not say much about alternate forms or about proper placement of vowel marks. The font designer, as she designs the font glyph faces, is the best person to specify these substitution and

positioning rules. It also allows the font designer to give free flow to her creativity. Standards like INSFOC [1] that define a fixed set of glyph outlines and their indices will be useful for situations like web pages on a display, where the font machinery has to be simple, fast and not of high quality. For high quality rendering, a more sophisticated technology is required. The glyph faces and the rules are packaged into the **OpenType Font**. The **OpenType Font** then enters into the ternary *GlyphsToGlyphs* relationship with Visual ordered **Glyph** syllable to form the **Renderable** syllable. For the *RKRI* example, Fig. 1 shows the **Renderable Glyphs** of the syllable separately on line 3 and together on line 4.

3.2.6 Why not set all flags?

If you examine the Visual ordered **Glyphs** syllable for the *RKRI* example in Fig. 6, from where the *GlyphsToGlyphs* processing of the OpenType font is started, you will see that specific flags are set for all the glyphs even though only the Reph and the Below-base Form have to be distinguished by setting the *rphf* and the *blwf* on the corresponding glyphs. Can we then create the **Renderable** syllable by setting these unavoidable flags and *allf* for the other glyphs as in Fig. 8? Unfortunately, the Vattu Variant form क़ will not be created.

| | | | | | | | | | | |
|---|---|------|---|--|------|---|--|------|---|---|
| < | ि | | क | | र | ् | | र | ् | > |
| | | allf | | | allf | | | blwf | | |
| | | | | | rphf | | | | | |

Figure 8: Unsuccessful setting of *allf* . The *RKRI* example

The correct flag setting involving *allf* and the the unavoidable flags (*rphf*, *blwf*) is shown in Fig. 9. The unsetting of a flag from the *allf* is indicated by the $-$ operator on the flag. It is quite counter-intuitive. For the vattu subcluster identified in the logical character string, you have to unset the *rphf* and for the reph subcluster, you have to unset the *blwf*!

| | | | | | | | | | | |
|---|---|------|---|--|-----------|---|--|-----------|---|---|
| < | ि | | क | | र | ् | | र | ् | > |
| | | allf | | | allf | | | allf-rphf | | |
| | | | | | allf-blwf | | | | | |

Figure 9: Correct setting of *allf* . The *RKRI* example

3.2.7 Some problems with the Microsoft Typography approach

Notice the step involving reordering within the Vattu subcluster $\langle 094d\ 0930 \rangle$ from Fig. 4 to Fig. 5. Microsoft Typography says that this is according to the Unicode standard. The Unicode standard has a specific rule ([2] Chap 9.1 Devanagari, R6 Consonant RA Rules) which states: *R6 Except for the dead consonant RA_d , when a dead consonant C_d precedes the live consonant RA_l , then C_d is replaced with its nominal form C_n , and RA is replaced by the subscript nonspacing mark RA_{sub} , which is positioned so that it applies to C_n .*

So according to Unicode, the sequence $\langle Ka\ Virama\ Ra \rangle$ will be handled as follows:

1. Combine *Ka Virama* to form Ka_d .
2. Finding Ra after Ka_d , replace Ka_d with Ka (equivalent to Ka_n).
3. Replace Ra with Ra_{sub}

The gist of the Unicode rules is that the Virama is associated with Ra and not with the Ka. Microsoft Typography has understood this. An additional step that they have taken is to move the Virama after the Ra to cement the idea that this Virama is not with Ka but with Ra. Unfortunately this creates the ambiguity that can only be resolved by using flags. Given a reordered subcluster $\langle Ra\ Virama \rangle$ should it become a reph (RA_{sup}) or a vattu (RA_{sub})? Microsoft Typography uses the flag `blwf` for vattu and `rphf` for the reph form. Notice that the source glyphs of the substitution rules in the GSUB tables of the OpenType font for these two cases are same (Tab. 1). So, at the character reordering level, we have to worry about, what flag has been chosen by the lower layer font for its substitution rules. If a font designer includes the vattu rule (Below-base Form) in a table with flag `vatt`, the character level algorithm has to set `vatt` on the $\langle Ra\ Virama \rangle$ subcluster.

3.3 The IndiX implementation

The difficulties in implementing the Microsoft Typography shaping architecture for the twelve Indic scripts, made us reexamine the Microsoft model. The IndiX model (Fig. 2 bottom) is based on the following observations:

1. Substitution rules in the OpenType Font should be distinguishable either on the source glyphs or on their contexts.

2. There is no need to reorder the characters of a **SubCluster**. The **OpenType Font** need not enter into the relationship *LogicalToMSOpenTypeOrder* with the Logical ordered **Character** SubCluster to form a flagged **SubCluster** of Character, ordered according to the OpenType Font substitution rules.
3. The flags within the OpenType Font are artifacts within it. The **Logical Syllable** does not require any flag for determining how the syllable has to be rendered. This is true for the **Visual** syllable of SubClusters of Character as well. If the OpenType Font is designed with distinguishable substitution rules then there is no need for any flag even for Visual ordered **Glyph**
4. The **Visual** syllable, especially the ordered Subcluster (of Character), can be considered to be an enhancement to the ISO TR 15285 model, associated with the **Logical** Syllable and not as an enhancement to the model for the **Intelligent Font** or **Renderable** Syllable.

3.3.1 A Devanagari OpenType Font according to IndiX

In IndiX we have cut all effects of the **OpenType Font** from the **Logical** and **Visual** Syllable of Characters. Instead of the **OpenType Font** deciding how the character level processing has to happen, the Unicode standard decides how the characters are reordered. The Unicode standard then determines the OpenType Font tables as well. An extract of the substitution tables from a Devanagari OpenType font organized according to the recommendations of Unicode standard is shown in Table 2.

3.4 Indic shaping in IndiX: The *RKRI* example

The four Logical **SubClusters** for the *RKRI* example are shown in Fig. 3 between vertical bars. The **Logical** syllable of ordered subclusters is reordered as per the **Unicode** standard to form the **Visual** syllable of ordered subclusters. Functionally, the Logical **SubClusters** are repositioned. The four Visual **SubClusters** of Character for the *RKRI* example are shown in Fig. 4 between vertical bars. The four Visual **SubClusters** of Glyph for the *RKRI* example are shown in Fig. 10. This syllable is created through the *CharsToGlyph* relation implemented using the CharMap table in the **OpenType Font**.

In the next step, the **Renderable** syllable in Fig. 11 is formed. Note that *allf* is not seen to be set on any of the glyphs. This is because it has to be set on all glyphs. It is not part of the variable state of any glyph and so there is no information to be

| Lookup table name | Flag | Substitution rule (GSUB) | | | |
|-----------------------|------|--------------------------|---------------|---------------|--------------|
| | | context before | source glyphs | context after | target glyph |
| Below-base Form | blwf | | ू र | | ू |
| Reph | rphf | | र ू | | ू |
| ⋮ | | | | | |
| Vattu Variants | vatu | | क ू ठ ू | | क्र ठ्र |
| ⋮ | | | | | |
| Half forms | half | | क ू | { क, ठ,..} | क् |
| ⋮ | | | | | |
| Pre-base Substitution | pres | | ू | { क्र, क,..} | ू |
| | | | ू | { र,..} | ू |
| ⋮ | | | | | |

Table 2: A Devanagari OpenType Font sample according to IndiX

<| ि | क | ँ | र | र | ँ |>

Figure 10: Visual ordered Glyphs in IndiX (First step). The *RKRI* example

gained in showing that the flag has to be set on all the glyphs. In fact, for the FreeType library [3], if the FeatureFlag is cleared it means that all flags are set. For the FreeType library, if you want the Below-base form to be not enabled as in Fig. 9, then you have to set the bit corresponding to blwf in the FeatureFlag.

<| ि | क | ँ | र | र | ँ |>

Figure 11: Visual ordered Glyphs in IndiX (Second step). The *RKRI* example

From the second step, the *GlyphsToGlyphs* relation, independent of FeatureFlags, generates the final **Renderable** syllable as shown in Fig. 1 separately on line 3 and together on line 4.

3.5 The new IndiX shaping architecture

Some points to be noted from the *RKRI* example and the IndiX OpenType Font in Tab. 2 are:

1. The source glyphs or the contexts of all substitution rules are distinct.
2. The vattu substitution (Below-base Form) precedes the reph.
3. The half consonant rules (Half forms) are after the reph.

The order of these rules is important. Consider the input sequence $\langle 0915\ 094d\ 0930\ 094d\ 0920 \rangle$. No subcluster needs repositioning for this sequence. After the first level glyph generation, the sequence of glyphs is: $\langle क \ \grave{\text{र}} \ \grave{\text{ठ}} \rangle$. If the Half forms were before the Reph, the $\langle क \ \grave{\text{र}} \rangle$ will be substituted by क^{h} and the Reph will shape the next two glyphs. The final glyphs will be $\langle \text{क}^{\text{h}} \ \grave{\text{ठ}} \rangle$ which is incorrect. Similarly, if the vattu (Below-base Form) comes after the Reph, the final output will again be $\langle \text{क}^{\text{h}} \ \grave{\text{ठ}} \rangle$.

The correct glyph sequence is $\langle क \ \grave{\text{र}} \ \grave{\text{ठ}} \rangle$. The idea is that the sequence \langle

२) should be given priority over any other rule that may take away the

4. Neither the character subclusters or the glyphs are flagged. While the GSUB tables in the IndiX OpenType font are tagged with flags, this information is irrelevant for repositioning characters as well as for substituting glyphs. All that matters is the proper ordering and form of the substitution rules. The constraints on the font are derivable from the Unicode reordering (repositioning) rules. But the OpenType font does not place any constraints on the character reordering logic.

The last point is the most significant advantage of the new IndiX shaping architecture. In the earlier architecture, the lower font layer imposed unnecessary constraints and requirements on the upper layer. The upper layers became complicated. More complex upper layer made it difficult to handle the various Indic scripts. Independent development of OpenType fonts, the font level and the character level processing routines suffered. Only those groups, who knew how to examine and modify the fonts tables and the character level reordering could make progress.

With the new architecture, implementation dependent choices may remain but are reduced. So the upper layers will become more general. It will also help to replace one technology by another at the lower layer, say OpenType with INSFOC, without having to make choices inappropriate for that level. For example, in our architecture, the INSFOC handling layer does not have to recognize some flag called blwf.

3.6 Handling positioning

The positioning of a sequence of glyphs for Latin script, is a simple procedure implemented by the font machinery. The layer above the font layer supplies a Latin character string to the font machinery through an interface which looks like: *DrawString(atPosition, aString)*. The font machinery converts the character string to a glyph string, and calls a lower layer interface *DrawGlyphString (atPosition, aGlyphString)*. *DrawGlyphString* positions the current point at atPosition and commences rendering the glyphs one after another. The rendering engine draws a glyph at a point, advances the current point according to the glyph metrics and draws the next glyph in correct relation to the previous one. When the GPOS tables of an OpenType font are used to get positioning information, there are at least three ways of handling it.

The first is to call *DrawGlyphString* repeatedly with a single glyph at a time, updating the point at which the glyph has to be drawn from the GPOS tables. This was the method used with IndiX initially. As the *DrawGlyphString* routine and the OpenType handling were both on the XServer, there was no appreciable degradation of performance. But if the *DrawGlyphString* were implemented on the XClient, the degradation would have been appreciable.

The second method, promoted by ICU [5], is to move the positioning information into separate x and y arrays which are of the same size as the array of glyphs. Many font rendering machines support rendering of glyphs in this fashion. For example, the PostScript font resources [6] support *xyshow()* for such parameters.

The third method, promoted by IndiX, uses spacer and unspacer glyphs to correctly position the glyphs with respect to each other. Suppose there is a glyph corresponding to a non-spacing vowel mark and GPOS says that it has to be moved relative to its nominal position by (x, y). We create a spacer glyph which advances the current position by (x, y) and place it before the non-spacing glyph. After the non-spacing glyph, we place an unspacer glyph which advances the current point by (-x, -y). Handling of spacing glyphs that need to be corrected is only slightly more complicated.

We have seen that the number of spacer glyphs does not run into thousands. It is roughly of the order of the number glyphs in the font. For Raghu font, which has about 600 glyphs, we required about 200 spacers and unspacers. It is better to create separate xspacers and yspacers. Generally, these spacers have to be created dynamically. If this is a problem or the number is too large, then we have a static scheme with a small number of appropriately valued spacers. Any spacing can be achieved by a sequence of such fixed valued spacers just as any quantity of Rupees can be made up by appropriate number and combination of Rs 1, Rs 2, Rs 5, Rs 10 currency notes.

Since we need both x and -x as the spacing value, we have found a simple method to assign the nominal spacers. The initial few nominal spacers will advance the current point by 1, -1, 3, -3, 9, -9, 27, -27. These numbers can generate any number from -40 to 40 using one of them only once. This scheme has been implemented in our printing tools. For the X11 display, we are investigating dynamic creation of these small set of spacers. It is suggested that all Unicode fonts should support certain glyphs, like the dotted circle. Can we similarly suggest that all OpenType fonts should contain such a set of spacer glyphs?

4 Detailed standards

5 Conclusion

References

- [1] *Indian Standard Font Code (INSFOC)*. <http://tdil.mit.gov.in/insfoc.pdf>, July 2002.
- [2] *The Unicode Standard, Version 4.0: The Unicode Consortium*. Addison-Wesley, Reading, MA, USA, 2003.
- [3] David Turner. *A Free, High-Quality, and Portable Font Engine*. <http://www.freetype.org/>, 2003.
- [4] Edwin Hart and Alan Griffiee. An operational model for characters and glyphs. Technical Report ISO/IEC TR 15285, 1998.
- [5] IBM and Others. *International Components for Unicode (ICU) ICU 2.8 Documentation*. <http://oss.software.ibm.com/icu/apiref/>, 2003.
- [6] Adobe Systems Incorporated. *PostScript Language Reference Manual*. Addison Wesley, San Fransisco, CA, USA, 1991.
- [7] Microsoft Corporation. *Creating and supporting OpenType fonts for Indic scripts* . <http://www.microsoft.com/typography/otfntdev/indicot/default.htm> also from <http://www.microsoft.com/typography/specs/default.htm>, 2003.