Kerning or Positioning Using Spacer Glyphs

Positioning with Spacers

Vinod Kumar*

* Centre for Development of Advanced Computing, Gulmohar Cross Road No 9, Juhu, Mumbai 400 049, India

ABSTRACT. Scripts, especially South Asian ones, require kerning or repositioning of marks and signs. Pairwise kerning is not advisable because of the large number of pairs involved. Anchor attachment based positioning reduces the effort needed to design the kerning tables and to position the marks. But the glyph painting machinery finds it difficult to use these anchor tables. By converting the kerning offset into a sequence of fixed offset spacer glyphs, the glyph painting machinery need only paint glyph after another, dealing with one glyph at a time. The spacer glyphs for complex scripts can be generated using the unavoidable substitution tables.

RÉSUMÉ. Les écritures, et plus particulièrement celles de l’Asie du Sud-Est nécessitent souvent des mécanismes de positionnement de marques et de signes. Le crénage par paires n’est pas pertinent à cause du grand nombre de paires de crénage à générer. La technique de positionnement par le biais d’ancres facilite le développement des tables de crénage et de positionnement. Mais le mécanisme de rendu a encore des difficultés à utiliser ces tables d’ancres. En convertissant le décalage de crénage en une série de glyphs d’espacement à chasse constante, le mécanisme de rendu ne s’occupe que d’un seul glyphe à la fois. Dans le cas des écritures complexes, les glyphs d’espacement peuvent être générés par les tables du substitution.

KEYWORDS: kerning, OpenType, GPOS, GSUB, glyph metrics, spacer glyph

MOTS-CLÉS : crénage, OpenType, GPOS, GSUB, métriques de glyphe, glyphe d’espacement

DN – 9/2006. Fontes numériques, pages 69 à 85
1. Introduction

When two glyphs, say ‘A’ and ‘V’, are to be sequentially drawn, the glyph handling machinery paints ‘A’ at the current point, updates the current point by adding to it the width of ‘A’, and then paints ‘V’ at the new current point. How the two glyphs will appear together is decided by the glyph metrics and shapes of the two glyphs. But rich typography requires the relative positioning of the glyphs to be modified between certain pairs. For example, the ‘AV’ are closer than the normal. For South Asian scripts, positioning the vowel signs over or under the consonants cannot be done properly by looking at the glyph metrics alone. Figure 1 (on page 71) shows how, in Devanagari, a vowel sign $Uu$ attaches improperly and properly at the bottom of a letter $Ka$. There are three ways of properly positioning the marks (The Unicode Consortium, 2003):

- a single glyph combining the letter and the properly positioned sign;
- variations of the sign;
- kerning of the sign.

For Devanagari, forming ligatures of the letters and the signs is not feasible. There are 33 letters and at least 10 signs leading to 330 ligatures for Devanagari. Variations of the signs are feasible. For the Devanagari example in Fig. 1, a variant of the $UuMatra$ closer towards the vertical stem of the $Ka$ along with the normal $UuMatra$ positioned below the right stem of $Pa$ will reasonably cover most of the cases. But the middle stem of at least some letters, may not be exactly at the same position as that of $Ka$. Some scripts, like Oriya, has many round headed letters, unlike Devanagari with well defined stem. Variants of the signs are not feasible in these situations. The third option, kerning, where the position of the second glyph is modified from its normal position is better for South Asian scripts.

Pairwise kerning, as used with Latin fonts, is not suitable for Indic fonts. For Devanagari, the pairwise kerning table will require at least 330 entries. The size of the kerning table can be reduced by class based kerning. Anchor attachment positioning, as supported by the positioning tables (GPOS) in OpenType fonts is a better alternative to class based kerning. Moreover, OpenType fonts have glyph substitution tables (GSUB) that facilitate the many characters to many glyphs transformation (Hart et al., 1998) needed for Indic scripts (Sandeep Rao, 2005). No wonder that Indic scripts take to OpenType fonts (Microsoft Corporation, 2004) as ducks to water.

However, anchor attachment tables are not easily understood by the older glyph rendering machinery. The glyph rendering machinery can be expected to handle kerning pairs, not GPOS tables. There should be some way of introducing the kerning information between two glyphs at the rendering layer. Spacer glyphs are the solution. Moreover, spacer glyphs allow simplification of OpenType fonts by replacing GPOS tables by GSUB tables.
1.1. Organization of this paper

The sections that follow illustrate and justify some ideas on glyph positioning. Positioning is important and heavily used for complex scripts like Indian scripts. Kerning is the best method to achieve positioning and not variants of the marks or ligatures of consonants and marks. It can be implemented by class based kerning or by anchor attachment. For Indic scripts, anchor attachment is better. Encapsulating the correction into a spacer glyph offers the least change to the painting layer and its interface. Finally, it is possible to generate the spacer glyphs using the substitution tables instead of the positioning tables. As substitution tables cannot be dispensed with for complex scripts, avoidance of positioning tables can simplify the fonts and the font processes.

\[
\begin{align*}
<\text{कू पू}> & \quad \text{कू पू कू पू}
\end{align*}
\]

Devanagari <Ka UuMatra Pa UuMatra>

Figure 1. Devanagari string KUuPUu spaced improperly and properly

2. Notation and coordinate system

We will use the PostScript (Adobe Systems, 1991) notation for the glyph metrics to illustrate how the glyphs are positioned. Unlike it, we will choose glyph instead of character. This is according to the clear distinction made between characters and glyphs in (Hart et al., 1998).

There are two different coordinate systems coming into play when glyphs are rendered on a page or display. The Glyph Space, used by the font designers to express the font outlines and metrics, is in terms of FontUnits. A glyph has an origin with respect to which it is drawn and a glyph width (PostScript calls it character width) indicating how this glyph advances the current point for laying out the next glyph. So glyph width is alternatively known as the advance width. The origin is a two dimensional point in the Glyph Space. A glyph is drawn with its origin coinciding with the current point. The glyph width is a 2-d vector in the same coordinate system.

The page or display in User Space is in terms of units of 1/72 of an inch (Adobe Systems, 1991). The current point is the most important object in the User Space. Before an operation like drawing a text string, the current point indicates the initial starting point of the operation. After the operation, the current point refers to the final position at which a succeeding operation can begin (Fig. 4, p. 83). Positions and vectors in the two spaces may have to be converted to each other for interaction.
between the user space and the glyph space as in mouse based selection, hit testing and initial positioning of a text string.

3. Kerning

As kerning involves a correction to the normal process of laying out of glyphs, the normal process has to be understood first.

3.1. Normal glyph positions

While the upper text layout layer uses the advance width of the glyphs to get their extents and justify or line break text strings, the glyph rendering machinery at the lowest layer uses the advance width to actually paint glyph after glyph.

Originally, especially for glyphs for Latin script, the advance width was a non-negative integer expressed in FontUnits. This is understandable as the glyphs were used to display the simple linear Latin script on a dumb terminal or a line printer. But with the advent of Laser printers and Desk Top publishing for non-Latin scripts, advance width became a two-dimensional vector. Thus PostScript (Adobe Systems, 1991) states that the width (PostScript does not call it “advance width”) “is a vector in the character coordinate system; it has x and y components. Most Indo-European alphabets, including Roman, have a positive x width and a zero y width. Semitic alphabets have a negative x width. Some Asian glyphs have a non-zero y width.”

Adobe’s AFM (Adobe Systems, n.d.) too states that, for an individual glyph, \( \text{number}_x, \text{number}_y \) in the font metrics file indicates the Character width vector \((x, y)\) for the normal left to right writing direction.

3.2. Modified glyph positions

Occasionally, the spacing between two glyphs has to be modified. This is known as kerning and an example is where the letters ‘V’ and ‘A’ are drawn closer to each other than ‘V’ and ‘V’. In South Asian scripts, a vowel sign will have to attach to different points on consonants. Track kerning where the widths are uniformly changed will not be considered in this paper. The two issues in pairwise kerning are:

- how to specify the glyph pairs that need special positioning and the amount of correction, and
- how to paint these two glyphs.
3.3. Kerning table

Since it is the font that has the glyph shapes and metrics, the specification of kerning is almost always in the font. Thus, for pairwise kerning, Adobe’s AFM (Adobe Systems, n.d.) uses KP \texttt{name}_1 \texttt{name}_2 \texttt{number}_x \texttt{number}_y to specify that to position the second named glyph properly with relation to the first, it has to be moved by a vector \((x, y)\) in the standard character coordinate system.

3.4. Kerning correction

The actual corrective movement is a bigger challenge. With Latin scripts, (‘AV’ for example) the correction vector is of the form \((-30, 0)\). It means that ‘V’ has to be moved left towards ‘A’ by 30 font units. This is achieved by the glyph painting machinery, correcting the current point corresponding to the origin of ‘V’ by \(-30\) font units. The glyphs after ‘V’ are not affected by this correction. Thus the glyph painting and kerning mesh together very well for Latin script.

But the correction for glyphs like those for the Indic vowel signs is not that simple. Most signs are of zero advance width because they attach either to the top or bottom of a consonant. If the current point corresponding to such a sign is corrected, then any succeeding glyph will be misplaced unless the correction is undone. This is seen in Fig. 2 where the current point of \texttt{UuMatra} has been moved left, but after painting it, we have forgotten to undo the correction. So the succeeding \texttt{Pa} crashes on to the \texttt{Ka}. Compare this with the correct positioning in Fig. 1, p. 71.

![Figure 2. Devanagari KUuPUu. Uu not unspaced](image)

4. Positioning for Indic scripts

4.1. Simple and class based kerning

For Indic scripts, if we follow the AFM kerning pair list, we will end up with a lengthy one. For Devanagari with 33 consonants and 10 commonly used signs, the pairwise kerning table will require at least 330 entries. Pairwise kerning gives one the
ability to individually control the spacing between each pair of glyphs. Such power is not required for Indic scripts. The size of the kerning table can be reduced by class based kerning. Thus, in Devanagari, 23 consonants (like प फ Fig. 1) out of the 33 are with a stem on the right. The 10 signs can be classified into two: 5 top marks and 5 bottom marks. Thus, even if we cannot further classify the remaining 10 consonants, we are left with 11 consonant classes and 2 classes of signs. Classification of marks into, say top marks, implies that all these marks have to uniformly designed with their attachments at the same point. Then the class based kerning table will have only 22 entries. When painting the glyphs, a consonant will be slotted into one of the 11 classes and a sign as either a top mark or bottom mark. The class based kerning table will be searched and the kerning correction vector obtained.

Instead of class based kerning table, suppose we opt for creating variants of the marks. How many marks will we have to design to achieve the same quality as the class based kerning? This is easily answered. For the Devanagari example, a top mark will have 11 possible kerning corrections. A variant of the top mark will have to be created for each of these 11 possibilities. Hence we will need 55 variants of the 5 top marks, and 55 bottom mark variants. Thus instead of 10 marks, we end up with 110 marks. Low quality fonts reduce the consonant classes into 2 and hence they will need only 2 variants of each mark. Thus we see that variants are not an option for Indic scripts for rich typography.

For Indic scripts, the most obvious way to simplify positioning is to note that the signs attach to vertical stems or lowermost (or uppermost) point on curves.

### 4.2. Anchor point based positioning

The Indic scripts are amenable to positioning by anchor point attachment. Consider the Devanagari consonant क र and the mark उम्मत्रा in Fig. 1, p. 71. This mark attaches to the bottom point of the stem of the consonant. We can thus specify a bottom anchor point (Microsoft Corporation, 2003) for all consonants with stem. The marks to be put at the bottom too have the matching bottom anchor point. To distinguish between these two bottom anchor points, we call the anchor point on the first glyph as the Anchor Ring and the one on the mark as the Anchor Hook. Anchor attachment positions the two glyphs by moving the second glyph so that its hook coincides with the ring on the first glyph (Fig. 4, p. 83).

The font designer looks at each consonant, independently of other consonants or marks, and specifies its anchor ring. Then she looks at each mark, again independently of other glyphs, and specifies its anchor hook. If there are $m$ consonants and $n$ signs, then the design time complexity is of the order of $m + n$. With pairwise kerning, the complexity would be $mn$. 
4.2.1. Anchor positioning and class based kerning

How does anchor positioning compare with class based kerning at design time? The classification of the first glyphs (consonants) is done on the basis of inherent structural items, like the stem, of the glyphs. For the second glyphs (marks), their points of attachment are designed to fall at the same point with respect to the origin of the glyph. Thus the classification is based on the fortunate or forced coincidence of certain structural points on the glyphs. In anchor positioning, a new structural element, an anchor point, is designed into or identified in the glyph. It is the Anchor Ring for the first glyph and the Anchor Hook for the second glyph. So the designer using class based kerning has less freedom than the designer using anchor positioning. But she has to make a kerning table, listing all the pairs and the computed kerning vector for each pair in the table. The designer using anchor positioning has to identify the anchor points for each eligible glyph. After this she makes a anchor attachment table. At first glance, the anchor attachment table looks similar to the class based kerning table. But they are vastly different animals. The class based kerning table has 3 columns and several rows. In each row, the three columns contain the first glyph, the second glyph and the kerning correction vector for the second glyph if it appears after the first one. The anchor attachment positioning table (GPOS), on the other hand, has exactly one row and two columns. The first column lists all the first glyphs and the second lists all the second glyphs. The table identifies the anchor, top or bottom, on which it bases its computation. Moreover, the computation of the correction vector is at run time and not at design time.

At run time, the two approaches differ in many ways. Given a sequence of glyphs \( g_1 g_2 \), we have to verify whether the sequence appears in a row in the class based kerning table. For anchor positioning, we verify that \( g_1 \) is in the first column and \( g_2 \) is in the second column of the single row. The advantage is with anchor positioning. Once the sequence has been confirmed to be present, you can straight away read out the kerning vector from the third column of the row located in the class based kerning table. For anchor positioning, you have to read the glyph metrics and anchor points of the two glyphs and compute the vector. So, for getting the kerning vector, class based kerning holds the advantage.

4.2.2. Design with Anchor point attachment

The independent specification of the anchor point is conceptually correct. But in practical visual design, the consonant and mark may appear on the same editing screen and the operations to position them look like those for pairwise kerning. The anchor ring is moved to the appropriate position on the consonant. But the anchor hook on the mark is not moved in the same fashion. It is indirectly set by moving the whole mark so that it attaches properly to the consonant. The anchor hook on the mark is set because if you consider the same mark and another consonant whose attachment point has been set, the mark will position properly with the new consonant. You do not have to move the mark afresh for the new consonant. Similarly, if the anchor point for \( Ka \) has been set for \( UMatra \), you will not need to set it again for \( UuMatra \).
Visually, anchor points, say bottom, are specified in the following steps (Microsoft Typography, 2002):

1) select anchor attachment as opposed to pairwise adjustment;
2) select anchor position, bottom;
3) select a mark;
4) select a consonant;
5) move the bottom anchor point so that it is at the bottom of the stem of the consonant. If it is stemless, position the bottom anchor at its lowermost position.
6) Move the mark such that visually it fits with the consonant properly. Conceptually the anchor hook for the mark is being selected;
7) for each of the other consonants, move the bottom anchor point to the correct location so that the mark fits properly. Note that the mark does not have to be moved. It will follow the bottom anchor point to its correct position on the consonant;
8) select one consonant with its anchor point set. For each of the remaining marks, move the mark so that it sits pretty on the consonant. Note that the anchor point for the consonant is not moved.

4.2.3. Cascaded positioning

In Indic scripts, pairwise kerning may not suffice. Sometimes, two marks may attach to the same consonant. Or, as in Oriya, a subscript form of a consonant may attach to a full form consonant and then a matra will attach to the subscript. This is seen in Fig 3. The nominal spacing between gNgKaFull and gKaSubs is (−78, −270), and that between gKaSubs and gUMatra is (488, −60). As gKaSubs is spacing and crosses the right boundary of gNgKaFull even after the correction, the unspacing needed for gKaSubs is −(0, −270). The x unspacing becomes 0. Since the gUMatra attaches to the gKaSubs, the unspacing is postponed to after gUMatra. Hence the total unspacing after the gUMatra is −(0, −270) − (488, −60) = (−488, 330). On Fig 3 p. 77, the last row shows what happens to the gUMatra if the unspacing of gKaSubs is carried out immediately.

5. Correcting advance by rendering spacer glyphs

The lowest layer of the shaping and rendering pipeline can only take the starting point of the first glyph and render the glyph sequence one after another based upon the glyph metrics. The lowest layer takes the tuple (<Start>, <glyphseq>), moves its current point to Start and draws the <glyphseq> from there. Suppose the layer above has cranked out a glyph sequence <g1 g2 g3 g4> and there is a kerning correction, obtained from pairwise kerning table or from anchor positioning, to be applied to g3. So let us encode the correction as c3. Currently, c3 is not a glyph but something that stands for the kerning correction. The upper layer can be thought to have a sequence
Figure 3. Unspacing with cascaded positioning. Glyph sequence is Oriya \textless \textit{gNgKaFull} gKaSubs gUMatra gKaFull\textgreater. Top row shows correct positioning. All positioning disabled in middle row. Third row shows incorrectly positioned gUMatra as gKaSubs is unspaced immediately.

\textless g_1 g_2 c_3 g_3 g_4 \textgreater to be drawn from Start\textit{g_1}. The lowest layer cannot handle this sequence of dissimilar objects. The situation can be handled in three ways.

5.1. Misusing text layout interface for glyph layout

The upper layer can send two tuples (\textless \textit{Start}g_1, \textless g_1 g_2 \textgreater) followed by (\textless \textit{Start}g_3, \textless g_3 g_4 \textgreater) to the lowest layer. However it has to compute \textit{Start}g_3 somehow. There are two problems with this solution. First is that we have to find out where \textit{g_3} is going to turn up and then say that it should turn up elsewhere. Instead of telling Harry that he should step back 2 meters before jumping, we find out that he will be 10.2 meters in front of the shop and tell him that he should position himself 8.2 meters in front of the shop and then jump. The second problem with this scheme is that the upper layer
incessantly troubles the lower layer with many small jobs instead of giving it a few large jobs. The (\texttt{Start}, \texttt{glyphseq}) interface is to be used for page layout of strings and not for glyph layout.

5.2. Upgrading glyph layout interface

The second approach is a variation of the first one that solves multiple invocations of the lowest layer. Effectively, the upper layer sends many tuples in one message. In the example, the message sent will be \texttt{<(Startg1, g1 g2) (Startg3, g3 g4)>>}. In practice, ICU (IBM and Others, 2003) implements a variant of this scheme where the the position information of each glyph is placed in \texttt{x} and \texttt{y} arrays that are the same size as the array of glyphs. Many font rendering machine support this method. For example, the PostScript (Adobe Systems, 1991) paints successive glyphs with \texttt{xyshow()} for such parameters. The problem here is that the lower layer interface has to be upgraded. Moreover, the \texttt{x}, \texttt{y} positions are often absolute positions in user space (indicated by the \texttt{Startg3} parameter) and not relative positions. For example, PostScript does not look at the inherent glyph advance in \texttt{xyshow()} but drives the advances from the \texttt{x}, \texttt{y} arrays alone. PostScript, while interpreting the \texttt{x}, \texttt{y} in user space, thankfully treats them as relative advance widths. They override the character’s normal width. It would have been much better if the \texttt{x}, \texttt{y} values were in glyph space and added to the individual glyph advances. Then the upper layers could keep most of the \texttt{x}, \texttt{y} array contents to be 0. In our example, only \texttt{x3}, \texttt{y3} would have nonzero value.

5.3. Using Spacer glyphs

The third approach is to replace the correction specification (\texttt{c3} for example) by a proper glyph, a spacer glyph. This approach has many advantages. Firstly, the interface of the lowest layer is not changed. Secondly, the message to lower layer is not fragmented. The messages are few and large. Thirdly, the spacer glyphs carry out the corrections relatively and not absolutely. However the success and elegance of this approach depends upon the ability of glyphs to support two dimensional glyph advance.

Changing the positions of glyphs by using spacer glyphs is a practice followed from the times of hot metal typography. Unicode standard (The Unicode Consortium, 2003) has several Space characters like 3 PER EM, 4 PER EM, 5 PER EM, 6 PER EM, THIN SPACE, HAIR SPACE etc in the U+2000 to U+20FF code chart. These appear to be relics from the hot metal era. With digital typography, the correction is two dimensional and hence the spacer should be able to advance in two dimensions. Thus the advance width too should be two dimensional. So we suggest spacers like 3 PER EM X, 3 PER EM X, MINUS 3 PER EM X, MINUS 3 PER EM Y, etc.
5.4. **Spacer glyphs in IndiX**

IndiX is a project to “identify and embed the minimal, logical and required changes to text processing at all layers of a widely used and deployed software platform (GNU/Linux Graphical User Interface) so that the system handles all nine Indian scripts as easily as it handles the Latin script.” Indian scripts follow the \(m - n\) characters to glyphs mapping as opposed to the \(1 - 1\) mapping in Latin. The second difference is the heavy need for glyph positioning. The spacer glyph approach is the best for meeting the positioning requirements.

We have seen that the number of spacer glyphs needed for a font, does not run into thousands. It is roughly of the order of the number glyphs in the font. For RKRaghuHindi font (see the website under (Sandeep Rao, 2005)), 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 (Table 1). 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.

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? Alternatively, the U+2000 to U+20FF code chart can be populated with spacers like MINUS 3 PER EM X.

5.4.1. **The Oriya Example**

In the Oriya NgKKU KA example (Fig. 3), the spacer between NgKaFull and KaSubs was stated as \((-78, -270)\). This spacing is achieved by four spacers \(\langle A9202A9204A9187A9193\rangle\).

5.4.2. **Implementations**

PostScript supports advance width as two dimensional vector. Moreover, the font resource dictionary can be populated with these glyphs along with the glyphs corresponding to the Indic glyphs. Hence positioning by using the fixed size spacer glyphs in the table is eminently workable and is working.

X11 Server (Barkakati, 1995) does not have Y positioning. Moreover, the basic X11 server that we have modified to render Indic text, uses bit map fonts converted from the outlines. They do not have Y advance nor negative X advance. Hence it was not possible to fill the font with appropriate spacers. To get over a difficult situation, without drastic changes, we overload unused fields in the spacer glyph metrics (they
Table 1. Proposed spacers codes, names, and metrics

<table>
<thead>
<tr>
<th>Code</th>
<th>gName1</th>
<th>gName2</th>
<th>XAdv</th>
<th>YAdv</th>
<th>Bounding Box</th>
</tr>
</thead>
<tbody>
<tr>
<td>23e0</td>
<td>/X-2187</td>
<td>A9184</td>
<td>{ -2187</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>23e1</td>
<td>/X-729</td>
<td>A9185</td>
<td>{ -729</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>23e2</td>
<td>/X-243</td>
<td>A9186</td>
<td>{ -243</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>23e3</td>
<td>/X-81</td>
<td>A9187</td>
<td>{ -81</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>23e4</td>
<td>/X-27</td>
<td>A9188</td>
<td>{ -27</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>23e5</td>
<td>/X-9</td>
<td>A9189</td>
<td>{ -9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>23e6</td>
<td>/X-3</td>
<td>A9190</td>
<td>{ -3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>23e7</td>
<td>/X-1</td>
<td>A9191</td>
<td>{ -1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>23e8</td>
<td>/X+1</td>
<td>A9192</td>
<td>{ 1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>23e9</td>
<td>/X+3</td>
<td>A9193</td>
<td>{ 3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>23ea</td>
<td>/X+9</td>
<td>A9194</td>
<td>{ 9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>23eb</td>
<td>/X+27</td>
<td>A9195</td>
<td>{ 27</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>23ec</td>
<td>/X+81</td>
<td>A9196</td>
<td>{ 81</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>23ed</td>
<td>/X+243</td>
<td>A9197</td>
<td>{ 243</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>23ee</td>
<td>/X+729</td>
<td>A9198</td>
<td>{ 729</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>23ef</td>
<td>/X+2187</td>
<td>A9199</td>
<td>{ 2187</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>23f0</td>
<td>/Y-2187</td>
<td>A9200</td>
<td>{ 0</td>
<td>-2187</td>
<td>0</td>
</tr>
<tr>
<td>23f1</td>
<td>/Y-729</td>
<td>A9201</td>
<td>{ 0</td>
<td>-729</td>
<td>0</td>
</tr>
<tr>
<td>23f2</td>
<td>/Y-243</td>
<td>A9202</td>
<td>{ 0</td>
<td>-243</td>
<td>0</td>
</tr>
<tr>
<td>23f3</td>
<td>/Y-81</td>
<td>A9203</td>
<td>{ 0</td>
<td>-81</td>
<td>0</td>
</tr>
<tr>
<td>23f4</td>
<td>/Y-27</td>
<td>A9204</td>
<td>{ 0</td>
<td>-27</td>
<td>0</td>
</tr>
<tr>
<td>23f5</td>
<td>/Y-9</td>
<td>A9205</td>
<td>{ 0</td>
<td>-9</td>
<td>0</td>
</tr>
<tr>
<td>23f6</td>
<td>/Y-3</td>
<td>A9206</td>
<td>{ 0</td>
<td>-3</td>
<td>0</td>
</tr>
<tr>
<td>23f7</td>
<td>/Y-1</td>
<td>A9207</td>
<td>{ 0</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>23f8</td>
<td>/Y+1</td>
<td>A9208</td>
<td>{ 0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>23f9</td>
<td>/Y+3</td>
<td>A9209</td>
<td>{ 0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>23fa</td>
<td>/Y+9</td>
<td>A9200</td>
<td>{ 0</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>23fb</td>
<td>/Y+27</td>
<td>A9211</td>
<td>{ 0</td>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td>23fc</td>
<td>/Y+81</td>
<td>A9212</td>
<td>{ 0</td>
<td>81</td>
<td>0</td>
</tr>
<tr>
<td>23fd</td>
<td>/Y+243</td>
<td>A9213</td>
<td>{ 0</td>
<td>243</td>
<td>0</td>
</tr>
<tr>
<td>23fe</td>
<td>/Y+729</td>
<td>A9214</td>
<td>{ 0</td>
<td>729</td>
<td>0</td>
</tr>
<tr>
<td>23ff</td>
<td>/Y+2187</td>
<td>A9215</td>
<td>{ 0</td>
<td>2187</td>
<td>0</td>
</tr>
</tbody>
</table>

have no outlines, for example) and manage to load the glyph cache with the spacer glyphs. While rendering, the current point is advanced in two-dimensions, looking at the normal width (x) and the y width from the overloaded glyph metrics. The implementations would have been cleaner if all the fonts supported 2-dimensional vector as advance width.
6. Substituting positioning with substituting

We have shown how kerning correction vector can be replaced with spacer glyphs. The spacers are fixed too. Hence a GPOS table for pairwise kerning can be substituted by GSUB tables. Looking at the Oriya example 3, there will be two tables.

<table>
<thead>
<tr>
<th>Context before</th>
<th>Source $\rightarrow$ target</th>
<th>Context after</th>
</tr>
</thead>
<tbody>
<tr>
<td>NgKaFull</td>
<td>KaSubs $\rightarrow$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>KaSubs SpacersOf${78, 270}$</td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Unspacer substitution

<table>
<thead>
<tr>
<th>Context before</th>
<th>Source $\rightarrow$ target</th>
<th>Context after</th>
</tr>
</thead>
<tbody>
<tr>
<td>NgKaFull</td>
<td>$\rightarrow$</td>
<td>KaSubs</td>
</tr>
<tr>
<td></td>
<td>NgKaFull SpacersOf${-78, -270}$</td>
<td></td>
</tr>
</tbody>
</table>

Table 3. Spacer substitution

$SpacersOf\{-78, -270\}$ are A9202 A9204 A9187 A9193. $SpacersOf\{78, 270\}$ are A9196 A9190 A9213 A9211. So the two substitutions tables achieve:

$$NgKaFull \text{ KaSubs} \rightarrow NgKaFull A9202 A9204 A9187 A9193 \text{ KaSubs}$$

$$A9196 A9190 A9213 A9211$$

The latter is a nice stream of glyphs and the glyph painting machinery can uniformly paint it without suddenly getting a surprise of a kerning information.

6.1. Single table substitutions

If we have a set of standard spacer glyphs with each OpenType font, the GPOS tables could be replaced by GSUB tables. At first glance, it appears that for each pair, the spacer glyphs will have to be found and the complexity falls back to $mn$. But going back to the model of alignment of anchor points on the consonant with that on the sign, we get a $m + n$ solution.

6.2. Mechanism of Anchors

The basic ideas behind anchor attachment are the attachment ring on the first glyph, the attachment hook on the second glyph and the shifting of the second glyph till the two attachment points coincide. The attachment point is always specified with respect to the origin of the glyph. The origin of the second glyph without adjustment will be at the origin of first glyph added with its advance width.
Consider the table 4 of Devanagari glyphs.

The *U matra* will attach without any spacer to *Pa* as the anchor of *U* will be at \((800,0) + (-50,0)\) which is same as the anchor point of *Pa* \((750,0)\). For *Ka*, the anchor is at \((500,0)\) but that of a succeeding *U* will be at \((1000,0) + (-50,0) = (950,0)\). They can be made to coincide by adding a \((-450,0)\) spacer between them. Similarly, *Pha* and *U matra* will require \((-550,0)\) between them.

Thus we see in Fig. 4 p. 83 that a consonant glyph *G* \(_1\) with adv width \(w_1\) and anchor ring at \(r_1\), when succeeded by a glyph *G* \(_2\) with anchor hook at \(h_2\) will need a spacer of \((-w_1 + r_1 - h_2)\). With respect to the origin of the second glyph, the anchor of the first is at \(-w_1 + r_1\). Its anchor is at \(h_2\). To make them coincident glyph *G* \(_2\) has to be moved by \(-w_1 + r_1 - h_2\). In Fig 4 p. 83, we have shown two consonant characters that look like a *U* and a *C*. The sign is similar to the Kannada length mark. The correction vector is indicated as *s* (spacer) and the unspacer as *u*. The first row shows the advance widths and anchor positions of the three glyphs. The second row shows how the *U* and the mark will render without correction and with correction. The third row shows the similar relationship between *C* and the mark.

### 6.3. Splitting the Spacer substitution tables

Ignoring the unspacers, the spacer substitution table for glyph *G* \(_1\) and glyph *G* \(_2\) will be:

<table>
<thead>
<tr>
<th>Context before</th>
<th>Source (\rightarrow) target</th>
<th>Context after</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>G</em> (_1) (\rightarrow) <em>G</em> (_1)</td>
<td><strong>Spacer</strong>(_{(w_1 + r_1 - h_2)}) (\rightarrow) <em>G</em> (_2)</td>
<td></td>
</tr>
</tbody>
</table>

*Table 5. General pairwise spacer substitution*

Here the spacers depend on both *G* \(_1\) and *G* \(_2\). Hence the advantage of anchor attachment is lost. For each pair of glyphs *G* \(_1\) and *G* \(_2\), a separate table like Tab. 5 would be required. Hence there will be \(mn\) such tables. It is not possible to have a single table with \(mn\) entries. Clearly pairwise substitution for spacers is not feasible with Indic scripts.

The \(mn\) tables can be replaced by two tables. The first one (Tab. 6) contributes the spacers corresponding to *G* \(_1\) alone. It will have *m* entries. The second position is

<table>
<thead>
<tr>
<th>Glyph</th>
<th>Advance width</th>
<th>Anchor type</th>
<th>Anchor position</th>
</tr>
</thead>
<tbody>
<tr>
<td>FullKa</td>
<td>(1000,0)</td>
<td>ring</td>
<td>(500,0)</td>
</tr>
<tr>
<td>FullPha</td>
<td>(1200,0)</td>
<td>ring</td>
<td>(600,0)</td>
</tr>
<tr>
<td>FullPa</td>
<td>(800,0)</td>
<td>ring</td>
<td>(750,0)</td>
</tr>
<tr>
<td>UMatra</td>
<td>(0,0)</td>
<td>hook</td>
<td>(-50,0)</td>
</tr>
</tbody>
</table>

*Table 4. Four Devanagari glyphs, three of them with anchor rings and one with hook.*
Figure 4. Calculation of Spacing and Unspacing vectors. Row 1 shows the individual glyphs, two consonants and a mark. Each glyph has advance width and either anchor ring or hook. Rows 2 and 3 show how the Spacing correction $s$ is computed (left) from the distance between the ring and the hook. The correction is applied (right) for each of the consonant-mark pairs when the origin of the second glyph is moved by $s$.

now not a glyph, but a class of glyphs, say TopMarks. The anchor point of the first glyph will be chosen depending on the type of second glyph. If the second glyph is a top mark, then the top anchor ring of the first glyph is chosen.

<table>
<thead>
<tr>
<th>Context before</th>
<th>Source $\rightarrow$ target</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G_1 \rightarrow G_1$ SpacerOf ($-w_1 + r_1$)</td>
<td>{ $G_2$ }</td>
</tr>
</tbody>
</table>

Table 6. Appending spacers of first glyph
The second table (Tab. 7) contributes the spacers corresponding to the second glyph. It will have \( n \) entries. Since it is known whether the second glyph is a top or bottom mark, its anchor hook position is known context freely.

<table>
<thead>
<tr>
<th>Context before</th>
<th>Source ( \rightarrow ) target</th>
<th>Context after</th>
</tr>
</thead>
<tbody>
<tr>
<td>( G_2 \rightarrow \text{SpacerOf}(-h_2) \ G_2 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7. **Prepending spacers of second glyph**

Thus each table depends on the glyph metrics and anchor point of a single glyph and do not have to consider them as a pair. The space complexity reduces to \( m + n \). The small disadvantage would be the increased number of spacers.

### 6.4. Further simplification

If the anchor point \( h_2 \) is made into a constant \( hm \) then the constant can be added to the first table (Tab. 8). The table for prepending the spacers of second glyph becomes redundant. The physical meaning is that there is a popular attachment ring on the first glyph and all the marks are designed such that their hooks would attach to this ideal location. The actual ring on the first glyph could be offset from the ideal position and that correction will be independent of the second glyph. So a tool to produce the spacer substitution table should support the specification of the actual attachment ring and the ideal ring for the first glyphs. The ideal ring will be at a fixed offset from the glyphs advance width. The ideal ring for all first glyphs would be same. The second glyph will have its hook in such a way that it will coincide with the ideal ring.

<table>
<thead>
<tr>
<th>Context before</th>
<th>Source ( \rightarrow ) target</th>
<th>Context after</th>
</tr>
</thead>
<tbody>
<tr>
<td>( G_1 \rightarrow G_1 \text{SpacerOf}(-w_1 + r_1 - hm) )</td>
<td>{( G_2 )}</td>
<td></td>
</tr>
</tbody>
</table>

Table 8. **Spacers of first glyph with ideal attachment point**

### 6.5. Is GPOS replacement with GSUB feasible?

Introducing spacer glyphs between two glyphs \( \langle g_1g_2 \rangle \) is feasible using GSUB tables. Introducing the unspacer glyphs is difficult. The unspacers after \( g_2 \) depend on \( g_1 \). Unless a copy of \( g_1 \) is moved after \( g_2 \), we cannot create unspacer components that depend on \( g_1 \). OpenType has lots of difficulties in reordering glyphs. So, GPOS tables cannot be easily eliminated. The software that looks at the GPOS tables can extract the glyph metrics including anchor data, compute the kerning correction and convert it into spacer glyphs.
7. Conclusion

Converting correction vectors for kerning into sequence of fixed two dimensional spacers glyphs offers many advantages. The glyph painting machinery has to deal with only one type of work, painting glyph after glyph without getting distracted by looking at kerning tables, calculating and making those corrections. The font gets simplified, with substitution tables alone and no positioning tables. The font designer is not led to a disadvantage as well as there can be a friendly user interface to retain the $m+n$ advantage of anchor based positioning schemes. For the scheme to succeed, fonts should be standardized such that the advance width of each glyph is treated as a two dimensional vector.

8. References


