New Font Technology for X11R6

Changes for R6 Increase Font Rendering Capabilities, Improve Performance, and Enable Licensing

Nathan Meyers†

Abstract

X11R6 features a number of enhancements to X's font technology. Some of these enhancements build on capabilities first introduced with the X11R5 font server, others are new. This paper discusses the four major enhancements — the Matrix XLFD Enhancement, glyph caching, a sample authorization protocol, and scalable aliases — in terms of the capabilities they provide today, and the future capabilities they can enable.

Introduction

With the introduction of the networked font server and scalable fonts in X11R5, font technology has assumed increasing visibility and importance in X. Font technology has continued to evolve with X11R6, attempting to meet more sophisticated rendering needs, and to better utilize the distributed architecture of the X font server/client relationship.

This paper discusses four major changes introduced with X11R6:

1) **Matrix XLFD Enhancement**: Capabilities are now available, through enhancements to the existing font naming mechanism, to generate transformations of existing bitmapped and scalable fonts. Using the powerful expressive capabilities of affine transformation matrices, applications can create simple variations on fonts (wide, narrow, obliqued) or generate special effects (rotation and mirroring) for specific text rendering needs.

2) **Glyph Caching**: Glyph caching — deferred loading of character glyphs — allows X to reduce memory and computation requirements associated with generation of fonts. By deferring creation of characters that might not be needed, glyph caching can achieve significant savings — particularly for large, sparsely-used Asian fonts. X11R6 implements glyph caching in the X

†*Nathan Meyers is a Member of Technical Staff at Hewlett-Packard Company*

server's interface to the font server, reducing the X server's font memory requirements and enabling glyph caching to be implemented in individual font rasterizers.

3) **Authorization Protocol**: X11R6 contains a sample authorization protocol that can be used for access to fonts from a networked font server. The protocol allows a font server to identify the end consumer of a font, and can serve as an enabling technology for font licensing.

4) **Scalable Aliases**: Capabilities of font name aliases have been increased. Aliases can now be defined for scalable font names, and the aliasing mechanism can be used in conjunction with the Matrix XLFD Enhancement to create font variations.

The Matrix XLFD Enhancement

New semantics for the pointsize and pixelsize fields allow the specification of a matrix instead of a scalar in either or both of these fields. As described in the original proposal:

> An XLFD name presented to the server can have the POINT_SIZE or PIXEL_SIZE field begin with the character "`[`". If the first character of the field is "`[`", the character must be followed with ASCII representations of four floating point numbers and a trailing "`]`", with white space separating the numbers and optional white space separating the numbers from the "`[`" and "`]`" characters. Numbers use standard floating point syntax but use the character "`~`" to represent a minus sign in the mantissa or exponent. Numbers can optionally use the character "`+`" to represent the plus sign in the mantissa or exponent.
>
> The string `[a b c d]` represents a graphical transformation of the glyphs in the font by the matrix
>
> $$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$
>
> All transformations occur around the origin of the glyph. The relationship between the current scalar values and the matrix transformation values is that the scalar value N in the POINT_SIZE field produces the same glyphs as the matrix [`N/10 0 0 N/10`] in that field, and the scalar value N in the PIXEL_SIZE field produces the same glyphs as the matrix [`N*RESOLUTION_X/RESOLUTION_Y 0 0 N`] in that field.
>
> If matrices are specified for both the POINT_SIZE and PIXEL_SIZE, they must bear the following relationship to each other within an implementation-specific tolerance:
>
> PIXEL_SIZE_MATRIX = POINT_SIZE_MATRIX • [S_x 0 0 S_y]
>
> where
>
> S_x = RESOLUTION_X / 72.27
> S_y = RESOLUTION_Y / 72.27
>
> If either the POINT_SIZE or PIXEL_SIZE field is unspecified (either "0" or wildcarded) the preceding formulas can be used to compute one from the other.[†]

A typical 12-point font name, specified with a transformation matrix, would look like:

```
-adobe-new century schoolbook-medium-r-normal--0-[12 0 0 12]-100-100-p-0-iso8859-1
```

The numbers in the matrix are floating-point, and the value of the `FONT` property returned with a font contains matrices in both the pointsize and pixelsize fields, consistent with each other and with the resolution fields as described above.

[†]Paul Asente, "A Matrix Transformation XLFD Extension" (X Consortium *Fontwork* mailing list)

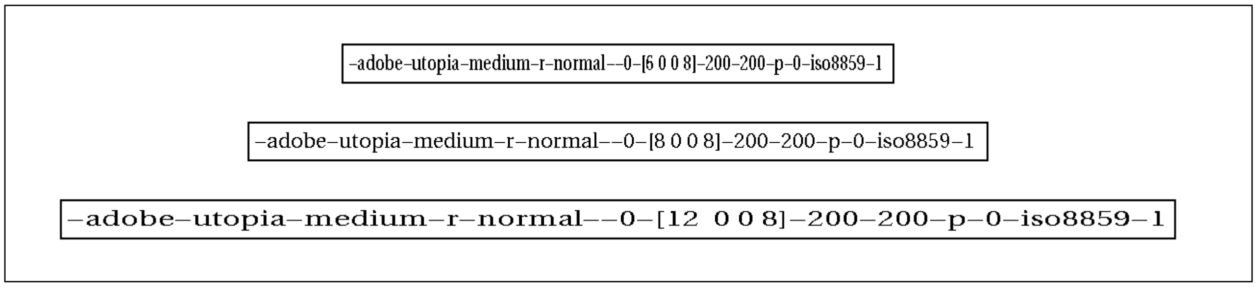
The following subsections discuss how the transformation matrix can be used to achieve font variations and effects.

Obtaining Simple Font Variations Through the Transformation Matrix

While some capabilities, such as font rotation, require advanced functionality in an X client to be useful, the Matrix XLFD Enhancement offers one capability of immediate use: the ability to generate font variations. Two transformations in particular — scaling and horizontal shearing — can generate useful variations on existing fonts.

Anamorphic Scaling

The ability to separately specify the X and Y components of the scaling matrix allows for separate specification of a font’s pointsize (vertical size) and setsize (horizontal size). By definition, a normally proportioned font is one whose setsize equals its pointsize. Figure 1 shows the variations possible with anamorphic scaling.



–adobe–utopia–medium–r–normal––0–[6 0 0 8]–200–200–p–0–iso8859–1
–adobe–utopia–medium–r–normal––0–[8 0 0 8]–200–200–p–0–iso8859–1
–adobe–utopia–medium–r–normal––0–[12 0 0 8]–200–200–p–0–iso8859–1

Figure 1: Three variations on the same scalable font: pointsize = 8, setsize = 6, 8, and 12

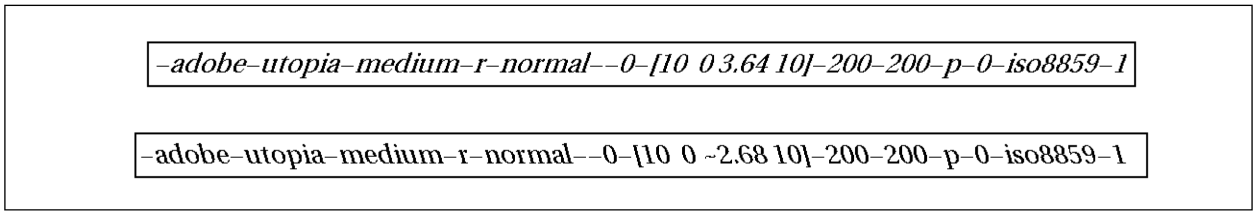
Horizontal Shearing

Horizontal shearing, or obliquing, generates slanted versions of existing fonts. The transformation for horizontal shearing is:

$$\begin{bmatrix} 1 & 0 \\ -\tan(\varphi) & 1 \end{bmatrix},$$

where φ is the slant angle counterclockwise from vertical. Figure 2 illustrates the effects of multiplying a 10-point scaling matrix by various horizontal shearing matrices:

$$\begin{bmatrix} 10 & 0 \\ 0 & 10 \end{bmatrix} \bullet \begin{bmatrix} 1 & 0 \\ -\tan(\varphi) & 1 \end{bmatrix}.$$



–adobe–utopia–medium–r–normal––0–[10 0 3.64 10]–200–200–p–0–iso8859–1
–adobe–utopia–medium–r–normal––0–[10 0 ~2.68 10]–200–200–p–0–iso8859–1

Figure 2: A 10-point font obliqued by -20 and +15 degrees CCW

Obtaining Special Effects Through the Transformation Matrix

The expressive capability of affine transformation matrices allows considerable power in specifying font transformations. Figure 3 illustrates the use of the variations discussed above, plus rotation and mirroring.

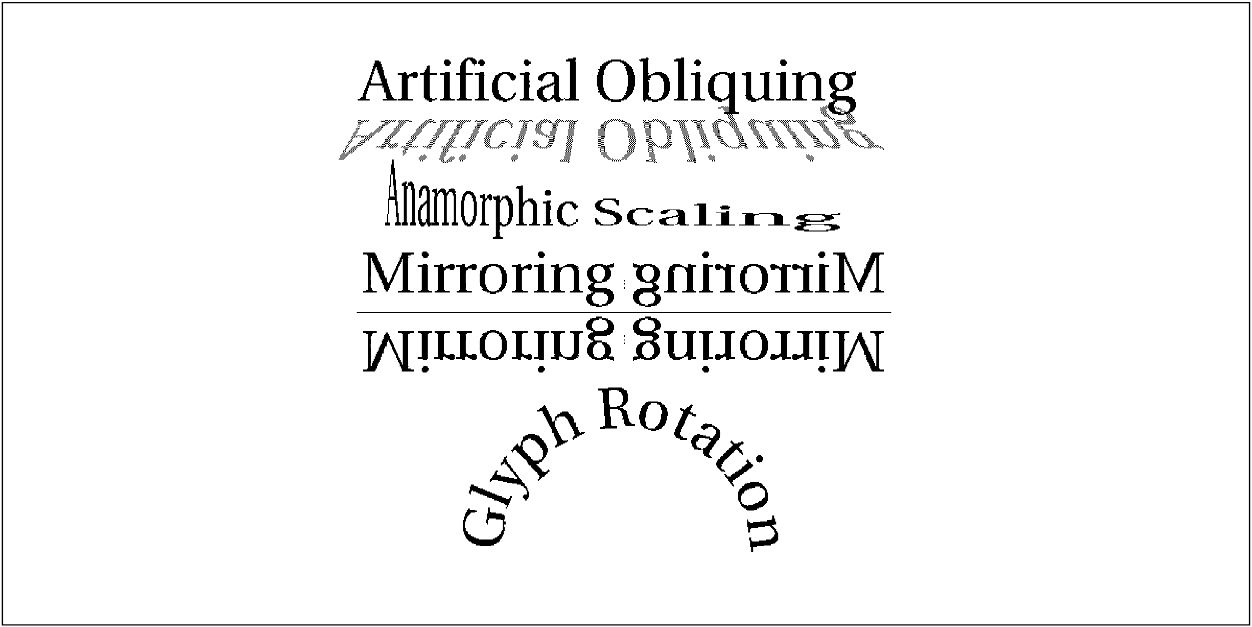


Figure 3: Some of the effects achievable with affine transformation matrices — all obtained from the same Adobe Utopia scalable font

The transformations required to achieve rotation and mirroring are shown in figure 4.

Horizontal Mirroring	Vertical Mirroring	Rotation
$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	$\begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}$

Figure 4: Matrix transformations for mirroring and rotation; θ *represents CCW rotation*

While generating rotated and mirrored fonts is straightforward, using them is more challenging: a mirrored or rotated font cannot simply be dropped into an existing application and expected to generate useful output. Additional client-side intelligence is required to make use of transformed fonts. Because X provides no functionality for drawing rotated text, the client is responsible for placement of each individual character when rendering text on a non-horizontal baseline. The remainder of this section discusses how clients must combine the transformation matrix, *a priori* knowledge of the intent of the transformation, and information returned in the font structures to properly place text.

Intent of the Transformation?

While a transformation matrix describes the mathematical manipulations to be performed on a font, it conveys no information about how text is to be rendered. For example, the matrix

$$\begin{bmatrix} 10 & 4 \\ 0 & 10 \end{bmatrix}$$

might be describing a 10-point font with 21.8˚ of vertical shearing; or it might describe an anamorphically scaled 10-point font with -21.8˚ of horizontal shearing that has been rotated 21.8˚. Indeed, any transformation matrix can represent an infinite number of histories. Figure 5 shows two possible interpretations for a font generated with the above transformation matrix.

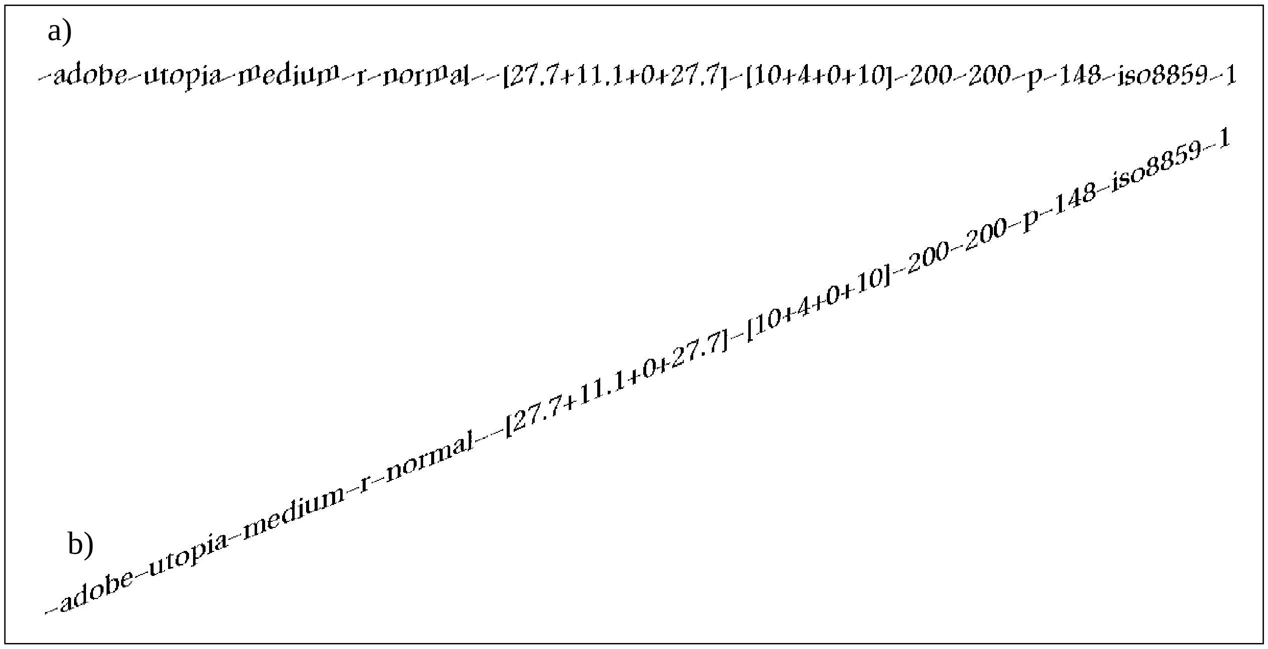

a)
-adobe-utopia-medium-r-normal--[27.7+11.1+0+27.7]-[10+4+0+10]-200-200-p-148-iso8859-1
-adobe-utopia-medium-r-normal--[27.7+11.1+0+27.7]-[10+4+0+10]-200-200-p-148-iso8859-1
b)

Figure 5: Two interpretations of the same font with the same [10 4 0 10] transformation matrix — a) as a vertically sheared unrotated font, b) as a horizontally sheared rotated font

So the transformation matrix does not, in itself, describe how to render the font; the client must place the characters based on its knowledge of the intent with which the transformation was generated. It is thus not possible to prescribe a canonical method for rendering non-horizontal text.

However, a simplification is possible — a usage model that will allow us to prescribe methods for rendering rotated text. Many applications needing rotated text are not typographically demanding or interested in unusual effects, they simply need to write normal text at an angle: for example, to label non-horizontal axes of a plot. For these applications, the process of building a font matrix can be modeled as the sequence: scale, oblique, rotate. That is, the transformation matrix is generated by applying those transformations in the order:

$$\begin{bmatrix} \mathit{setsize} & 0 \\ 0 & \mathit{pointsize} \end{bmatrix} \bullet \begin{bmatrix} 1 & 0 \\ -\tan(\varphi) & 1 \end{bmatrix} \bullet \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} .$$

For applications following this model, we can unambiguously determine the rotation angle and other relevant parameters entirely from the matrix (figure 5b, above, was rendered by interpreting the matrix according to the model). The sections below describe how to render text on a non-horizontal baseline using information from a transformation matrix generated under the model.

Font and Character Metrics for Transformed Fonts

The fundamental problem to be solved when rendering a line of text is that of applying the *escapement* — the spacing that separates one character from the next. Escapement is a vector: the character origin can move in both the horizontal and vertical direction after each character is drawn. The X protocol was designed to handle only horizontal rendering, and so treats the escapement as a scalar to be applied in the horizontal direction (the **XCharStruct** *width* field). There is no room provided in the protocol to return a two-component escapement vector.

The Matrix XLFD Enhancement solves this problem in an elegant way, by co-opting a previously unused field in the per-character **XCharStruct** metrics. The *attributes* field, whose purpose had never before been defined, now contains the character escapement that would have resulted had the font been scaled by the *pixelsize* matrix `[1000 0 0 1000]`. This *1000-pixel escapement*, in conjunction with information in the pixelsize transformation matrix, provides enough resolution for very accurate placement of glyphs — with no change to the protocol. Using the pixelsize matrix, the escapement vector for any given character in X's coordinate system can be computed as

$$\begin{bmatrix} \dfrac{1000\text{–}pixel\ escapement}{1000} & 0 \end{bmatrix} \bullet \begin{bmatrix} pixelsize\ matrix \end{bmatrix} \bullet \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} .$$

For example, if the client opens the font

```
-adobe-utopia-medium-r-normal--0-[8+4~4+8]-110-110-p-0-iso8859-1
```

it can examine the resulting FONT property:

```
-adobe-utopia-medium-r-normal--[12.2+6.09~6.09+12.2]-[8+4~4+8]-110-110-p-64-iso8859-1
```

and parse out the pixelsize matrix for use in the escapement computations. If the *attributes* field for the letter "A" reports a value of 635, the escapement vector for "A" in X's coordinate system is

$$\begin{bmatrix} \dfrac{635}{1000} & 0 \end{bmatrix} \bullet \begin{bmatrix} 12.2 & 6.09 \\ -6.09 & 12.2 \end{bmatrix} \bullet \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} 7.75 & -3.87 \end{bmatrix} .$$

Figure 6, later in this section, contains a code fragment illustrating rendering of a line of non-horizontal text, one character at a time, using the above computation.

Having discussed the newly defined **XCharStruct** *attributes* field, we briefly examine how the existing **XCharStruct** fields and the other components of the **XFontStruct** are affected by the Matrix XLFD Enhancement.

XCharStruct *ascent*, *descent, lbearing*, and *rbearing*: These fields function as always, describing the pixel extent of each glyph horizontally and vertically relative to the origin. They do not — indeed, cannot — apply in a transformed coordinate space; doing so would render them useless to the X server's glyph rendering logic.

XCharStruct *width*: This field contains the horizontal component of the escapement: the rounded X value from the escapement computation (above). For fonts with no rotation component, this corresponds to the existing *width* definition — so existing text rendering calls will function correctly for mirrored or obliqued fonts (horizontally mirrored fonts will, as expected, be rendered in the reverse direction from normal). For fonts with a rotation component, the *width* values do not contain enough information for character placement, and the *attributes* values must be used.

XFontStruct *ascent* and *descent*: These fields contain the vertical component — after transformation by the matrix — of the font overall ascent and descent. For fonts with no rotation or vertical mirroring, these correspond to the existing definitions. These metrics are useful for any mirrored or obliqued font, but inadequate for fonts with a rotation component; in that case, meaningful values must be computed from the `RAW_ASCENT` and `RAW_DESCENT` properties (see below).

Font Properties for Transformed Fonts

The Matrix XLFD Enhancement modifies the definitions of properties as follows:

> All font properties that represent horizontal widths or displacements have as their value the X component of the transformed width or displacement. All font properties that represent vertical heights or displacements have as their value the Y component of the transformed height or displacement. Each such property will be accompanied by a new zproperty, named as the original except prefixed with `RAW_`, that gives the value of the width, height, or displacement in 1000 pixel metrics.[†]

In other words, for a font with the pointsize[pixelsize] transformation matrix

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix},$$

the vertical properties (`POINT_SIZE`, `CAP_HEIGHT`, etc.) will have the same values as those for a normal (i.e., without a transformation matrix specified) *d*-point[pixel] font, and the horizontal properties (`NORM_SPACE`, `AVERAGE_WIDTH`, etc.) will have the same values as those for a normal *a*-point[pixel] font. If *a* or *d* is negative, the corresponding properties are negated.[‡]

These new definitions of the existing properties match existing definitions for normal fonts. And they continue to be useful for fonts with obliquing and mirroring. But they are generally not useful for fonts with a rotation component ($b \neq 0$, in our usage model). More information is needed for this case, which is provided by the `RAW_` properties.

The Matrix XLFD Enhancement defines new properties, corresponding to all existing numeric vertical and horizontal properties, but prefixed with "`RAW_`". These properties describe the metrics for an untransformed 1000-pixel version (`[1000 0 0 1000]`) of the font. For example: `RAW_PIXEL_SIZE`, `RAW_UNDERLINE_POSITION`, `RAW_MAX_SPACE`, etc. These 1000-

[†]Paul Asente, "A Matrix Transformation XLFD Extension" (X Consortium *Fontwork* mailing list)

[‡]*To support the possibility of negative values, all numeric horizontal and vertical properties are now INT32's. Clients using normal fonts will not suddenly see negative POINT_SIZE properties or other such surprises, but the change allows any property to assume a negative value if the corresponding component of the transformation matrix is negative.*

pixel metrics, like the *attributes* fields discussed above, provide high-resolution values that can be scaled for use with transformed fonts.

Two additional properties are defined: `RAW_ASCENT` and `RAW_DESCENT`. These provide the 1000-pixel versions of the overall font ascent and descent values, corresponding to *ascent* and *descent* from the **XFontStruct**.

Applying Font Properties to Transformed Fonts

For fonts with a rotation component, the `RAW_` properties provide the information needed to apply the corresponding metrics to the transformed text. For properties representing horizontal displacements or offsets (`RAW_MIN_SPACE`, `RAW_QUAD_WIDTH`, etc.), the transformation is the same as that used for the escapements (above):

$$\begin{bmatrix} \frac{RAW\ PROPERTY}{1000} & 0 \end{bmatrix} \bullet \begin{bmatrix} pixelsize\ matrix \end{bmatrix} \bullet \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} .$$

The result, as with the escapement computation, is a vector in X’s coordinate system that applies in the direction of writing.

The transformation of properties representing vertical displacements or offsets (`RAW_ASCENT`, `RAW_DESCENT`, etc.) is somewhat trickier: the matrix must be modified to remove any obliquing component (as understood in the context of our usage model) that would skew the resulting vectors. Describing the pixelsize matrix as

$$\mathrm{P} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} ,$$

define a new matrix

$$\mathrm{P}' = \begin{bmatrix} a & b \\ \frac{b^2c - abd}{a^2 + b^2} & \frac{a^2d - abc}{a^2 + b^2} \end{bmatrix} .$$

P′ is, in the context of our usage model, the pixelsize matrix with the obliquing component removed — now ready for use with the raw font properties. To transform the raw font properties representing vertical offsets or displacements, apply the new matrix thus:

$$\begin{bmatrix} 0 & \frac{RAW\ PROPERTY}{1000} \end{bmatrix} \bullet \mathrm{P}' \bullet \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} .$$

The result is a vector, in X’s coordinate system, perpendicular to the direction of writing. This vector is used analogously to the original properties: added to the current position if the corresponding property is normally added (e.g., `UNDERLINE_POSITION`), subtracted from the current position if the corresponding property is normally subtracted (e.g., `STRIKEOUT_ASCENT`). Use of this transformation is illustrated in figure 6, below.

```
  char *stringptr;              /* Null-terminated string to write */
  int xorigin, yorigin;         /* Location of string origin */
  double pixmatrix[4];          /* The pixel matrix, parsed from font name */
  XFontStruct *fontinfo;        /* The fontstruct for this font */
  int raw_underline_position;   /* Offset from baseline of 1000-pixel font to
                                   underline; from RAW_UNDERLINE_POSITION
                                   property or otherwise computed */
  double offset, underline_offset_x, underline_offset_y;        /* stuff */

  /* Output the text */

#define IROUND(x) (int)((x) > 0 ? (x) + .5 : (x) - .5)

  offset = 0.0;
  while (*stringptr)
  {
    /* Computation of charno and char_metric_offset specific to 8-bit
       text. */
    int charno = *(unsigned char *)stringptr;
    int char_metric_offset = charno - fontinfo->min_char_or_byte2;

    XDrawString(display, drawable, gc,
                IROUND((double)xorigin + offset * pixmatrix[0] / 1000.0),
                IROUND((double)yorigin - offset * pixmatrix[1] / 1000.0),
                stringptr, 1);
    stringptr++;

    offset += (double)(fontinfo->per_char ?
                       fontinfo->per_char[char_metric_offset].attributes :
                       fontinfo->min_bounds.attributes);
  }

  /* Output an underline */

#define A (pixmatrix[0])
#define B (pixmatrix[1])
#define C (pixmatrix[2])
#define D (pixmatrix[3])

  underline_offset_x = -(double)raw_underline_position *
                        (B*B*C - A*B*D)/(A*A + B*B);

  underline_offset_y = (double)raw_underline_position *
                       (A*A*D - A*B*C)/(A*A + B*B);

  XDrawLine(display, drawable, gc,
            IROUND((double)xorigin + underline_offset_x / 1000.0),
            IROUND((double)yorigin + underline_offset_y / 1000.0),
            IROUND((double)xorigin +
                   (underline_offset_x + offset * pixmatrix[0]) / 1000.0),
            IROUND((double)yorigin +
                   (underline_offset_y - offset * pixmatrix[1]) / 1000.0));
```

Figure 6: Code fragment illustrating use of the 1000-pixel metrics with the pixelsize transformation matrix for placement of non-horizontal text. The attributes field is used with the pixelsize matrix for escapement computations, and the value of the RAW_UNDERLINE_POSITION property is used with the P′ matrix to place an underline.

What Values and Properties Can Clients Expect?

Given the possibility that clients may encounter some font sources that support the Matrix XLFD Enhancement and some that do not, it can count on the following to be true if and only if a font is successfully opened with a matrix in the pointsize and/or pixelsize field:

- The **XCharStruct** fields *width* and *attributes* will be defined according to their new definitions (above).
- The **XFontStruct** fields *ascent* and *descent* will be defined according to their new definitions (above).
- The `RAW_ASCENT` and `RAW_DESCENT` properties will be provided.
- For every numeric property provided representing an X or Y displacement or offset (such as `POINT_SIZE`, `UNDERLINE_POSITION`, etc.), a corresponding `RAW_` property will be provided.

Conversely, if a font is opened without a matrix in the pointsize or pixelsize fields, there is no guarantee that the above will be true: clients cannot rely on finding useful *attributes* fields or `RAW_` properties in a font opened with scalar values in the pointsize and pixelsize fields.

(There is an unfortunate exception to the above guarantees: pre-R6 font servers zeroed out the *attributes* fields in the **difs** code. If a font is obtained through a chained pre-R6 font server, the *attributes* fields will be lost.)

Support Utilities

As of this writing, there is an unfilled need for client-side support utilities to enable easy use of the XLFD transformation matrix. Among the requirements: assembling an XLFD name from a raw name and a transformation matrix; parsing a matrix out of an XLFD name; generation of matrices for common transformations; DrawString and DrawImageString implementations for 8-bit and 16-bit fonts and for font sets.

Charset Subsetting

The Matrix XLFD Enhancement has an admitted weakness: it must be applied when a font is opened; it cannot be applied at render time. For clients generating special effects that require a different transformation for virtually every character (such as in figure 3), this means opening a different font for every character.

Opening dozens of fonts is slow — and particularly burdensome when very few of the characters being generated will be needed. While glyph caching (discussed later in this document) offers the eventual hope that X and font servers will not waste time building unneeded characters, that hope is still distant and very unlikely to be fully realized.

So X11R6 includes another capability in support of the Matrix XLFD Enhancement: the ability to hint — through the font name — which characters are of interest. The charset subset specification, appended at the end of the XLFD name, is of the form:

[value[_value][value[_value][value[_value] ...]] **]**

A value is a hex or decimal number; two values separated by a "_" specify a range. Values and ranges are separated by " ". So, for example, the name

```
-adobe-utopia-medium-r-normal--0-120-0-0-p-0-iso8859-1[65_67 0xe0_255 32]
```

indicates that the client is interested in the characters 32, 65-67, and 224-255. The font source can use this hint to save computation by generating a font containing only these characters (or any superset of these characters).

Advertising of Capabilities

If a client is to use the Matrix XLFD Enhancement on a font, it must be able determine which fonts can support it. This need to query capability is supported by **ListFonts**: if a capability is included in the name specification passed to **ListFonts**, only fonts whose rasterizers support that capability will be returned. For example, passing the name:

```
-*-*-*-*-*-*-*-[1 0 0 1]-*-*-*-*-*-*
```

to **ListFonts** (the particular choice of numbers in the matrix is immaterial) will return only font names that can support the Matrix XLFD Enhancement.

Support for charset subsetting is similarly advertised through **ListFonts**; including a charset subset specification in the name will return only font names that support charset subsetting.

Changes Required to Rasterizers

The API between libfont and font rasterizers has changed in several ways to support the Matrix XLFD Enhancement, necessitating corresponding changes to any vendor-supplied rasterizers. Specifically:

- The **FontRendererRec** structure (used by rasterizers to register their supported font file types) has an additional field to support advertising of capabilities.
- The **FontScalableRec** structure has replaced its *pixel* and *point* fields with matrices representing the pixelsize and pointsize, and a bitmap identifying what capabilities are specified in the fontname. This structure is passed to all rasterizer **OpenScalable**() procedures, often used within the rasterizers themselves, and passed by rasterizers as a parameter to the important **FontParseXLFDName**() utility. It is certainly possible for a rasterizer *not* to support matrices, but it must, at a minimum, know how to use the new fields in **FontScalableRec** to ascertain and specify point and pixel sizes.

A detailed description of the API changes is beyond the scope of this document. However, the R6 sample implementations of the Type 1, bitscale, and Speedo rasterizers contain good examples of how to use the changed structures.

History

Interest in advanced font rendering capabilities has been around almost as long as has X. For years, that interest centered around the creation of an X Typographic Extension — a bundle of capabilities that would address X's many text imaging weaknesses. Among the desired capabilities: subpixel character placement, kerning, resolution-independent character metrics, anti-aliasing, render-time transformation of characters, and more. The X Typographic Extension suffered the usual fate of innovations that try to do too much: it ended up mired in a bog of conflicting requirements. Despite occasional attempts to resurrect the Extension, it is considered dead. The growing popularity of Display PostScript®[†] for solving advanced text imaging problems virtually seals the fate of the X Typographic Extension.

There is, however, a large gulf between basic text rendering and full typographic capability — and there are clients whose needs fall into that void. The Matrix XLFD Enhancement works to address that gulf, but efforts to do so actually go back several years. The first attempt to provide advanced text capabilities through the XLFD name was the Hewlett-Packard XLFD Enhancements, introduced with its X11R4 product offering. Using very different modifications to the XLFD name, and a different strategy for font and glyph metrics, the HP XLFD Enhancements provided many of the capabilities discussed above: anamorphic scaling, obliquing, rotation, and mirroring. The HP XLFD Enhancements were the starting point for the discussion that ultimately led to the design of the Matrix XLFD Enhancement. Interested readers are referred to a 1992 article by Deininger and Meyers[‡] for more information about the HP XLFD Enhancements.

Glyph Caching

In the X11R5 sample implementation, obtaining a font from a font server causes the following sequence of transactions between the X client, the X server, the font server, and the rasterizers within the font server:

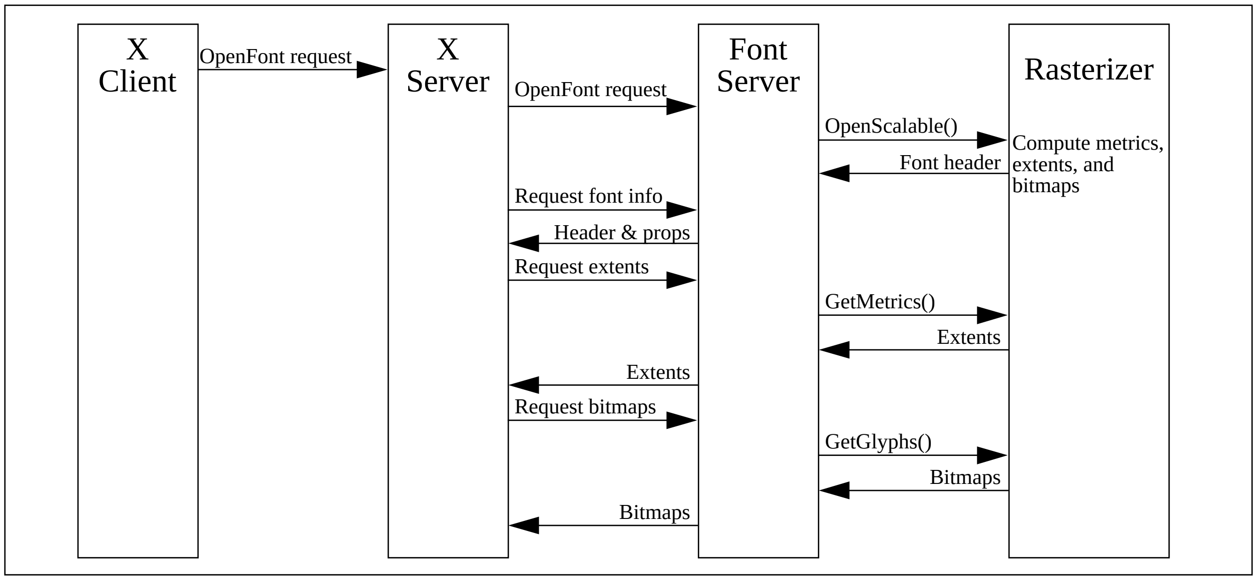


Figure 7: Transactions between the various font system components when a font is opened

What is wrong with this picture? Before the client has issued a single text rendering request using this font, two complete copies of the font and its glyphs already exist: in the font server and in the

[†]PostScript® *and* Display PostScript® *are registered trademarks of Adobe Systems Incorporated.*

[‡]Axel Deininger and Nathan Meyers, "Using the New Font Capabilities of HP-Donated Font Server Enhancements", *The X Resource* Journal, Issue 3, Summer 1992, pp. 97-119.

X server. The rasterizer consumed the cycles to build the glyphs and the memory to store them, and the X server took the network bandwidth and memory to obtain and store its own complete copy of the font.

Ideally, this should not happen. In the interest of memory usage and performance, only glyphs that are needed should be built. This is particularly critical for Asian fonts: ideographic fonts such as Japanese Kanji have thousands of glyphs, but are very sparsely used.

Glyph caching is a term for the practice of managing memory caches containing glyphs that have been selectively realized. One aspect of glyph caching, *deferred loading*, has been implemented in *libfont*'s interface to the font server.

When deferred loading is enabled, the last transaction shown in figure 7 — obtaining the bitmaps — is deferred. Instead, bitmaps are obtained a few at a time as they are referenced in text rendering requests. All other transactions still occur at font-open time; deferring them would be pointless, since virtually all X clients request the font header, properties, and extents immediately after opening a font.

Turning on Glyph Caching

Default behavior of the X and font servers is not to perform deferred loading. Deferred loading can be enabled in the X server with the command-line option:

```
-deferglyphs none|16|all
```

Specifying "16" causes the X server to defer loading for 16-bit fonts; "all" defers loading for all fonts.

Deferred loading is controlled in the font server through the config file. The command:

```
deferglyphs=none|16|all
```

functions similarly to the X server's -deferglyphs option. The option affects the font server's behavior when it obtains fonts from an upstream font server.

Requirements for X Servers to Support Glyph Caching

Figure 8 illustrates how an X server obtains glyphs on behalf of an X client.

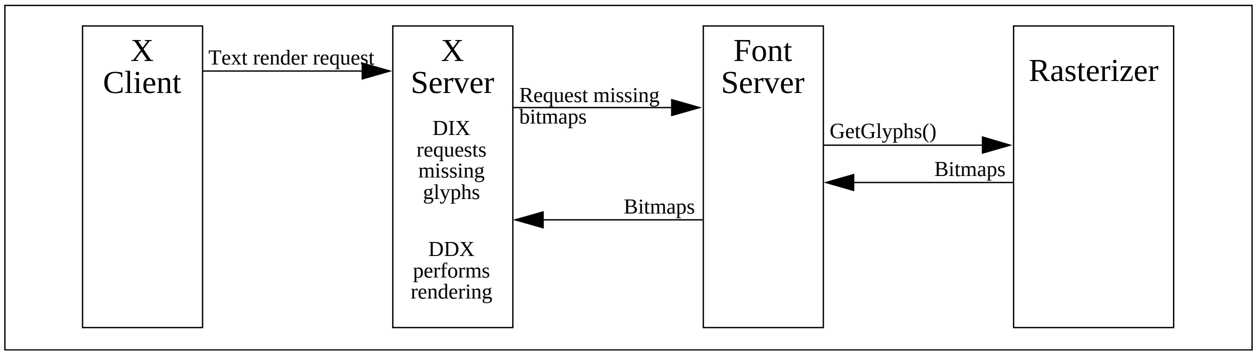


Figure 8: Glyphs are obtained as they are referenced in text rendering requests

Over time, as more of the font's characters are referenced, the X server's glyph cache grows to contain more of the font. Successive requests are satisfied with glyphs already in the cache, requiring fewer round trips to the font server after the font has been in use for a while. (The cache discipline in the sample implementation is grow-only until the font is closed; some future or proprietary implementation might choose to manage the cache more actively — perhaps removing glyphs using an LRU algorithm.)

There is a very important assumption implicit in this architecture: the X server's DDX code must never assume that glyph bitmaps are present (i.e., obtainable through the **GetGlyphs**() call) *except while handling a text rendering request referencing those glyphs*. At any other time, it is possible that a glyph has not yet been loaded, or that a previously loaded glyph has been unloaded from the cache.

Most X servers, including the sample implementation, meet this assumption and can support glyph caching today. But not *all* servers do: a common DDX performance-enhancement practice from times past — copying all glyphs into offscreen memory at font-open time — will cause an X server not to meet this assumption. DDX code that does not support glyph caching must indicate so at screen initialization time, by calling **SetGlyphCachingMode**() to register the highest level of glyph caching (`CACHING_OFF`, `CACHE_16_BIT_GLYPHS`, `CACHE_ALL_GLYPHS`) it supports.

What Next?

There are many unsolved problems remaining before glyph caching can be considered complete:

- The simple cache management discipline practiced in the sample implementation could be improved, optimizing memory usage by dumping little-used glyphs.
- When missing glyphs are referenced in a text rendering request, a decision must be made about which glyphs to obtain from the font server. Should only the missing glyphs be obtained? Should some additional glyphs be obtained with the hope of avoiding future costly round trips to the font server? The decision must be based on such factors as network speed, memory requirements and limitations, rasterizer speed, and even usage patterns for the font (ideographic fonts are sparsely used, phonetic fonts are densely used). The sample implementation uses a simplistic model that is in need of refinement.
- Finally, and most importantly, deferred rasterization support must be added to the rasterizers themselves. All current rasterizers in the sample implementation build the entire font at font-open time, rather than deferring creation of glyphs until they are needed. Unfortunately, deferring rasterization is a very difficult problem: the rasterizer must be able to accurately generate all font and glyph metrics and extents when the font is opened... a difficult problem to solve without taking the considerable time required to build the glyphs. For this reason, there are likely always to be rasterizers that cannot partake of glyph caching's performance benefits.

Sample Authorization Protocol

X11R6 includes a sample font authorization protocol: *hp-hostname-1*, a non-authenticating protocol to identify the end consumer of a font.

Consider an arbitrary network of X display and font servers:

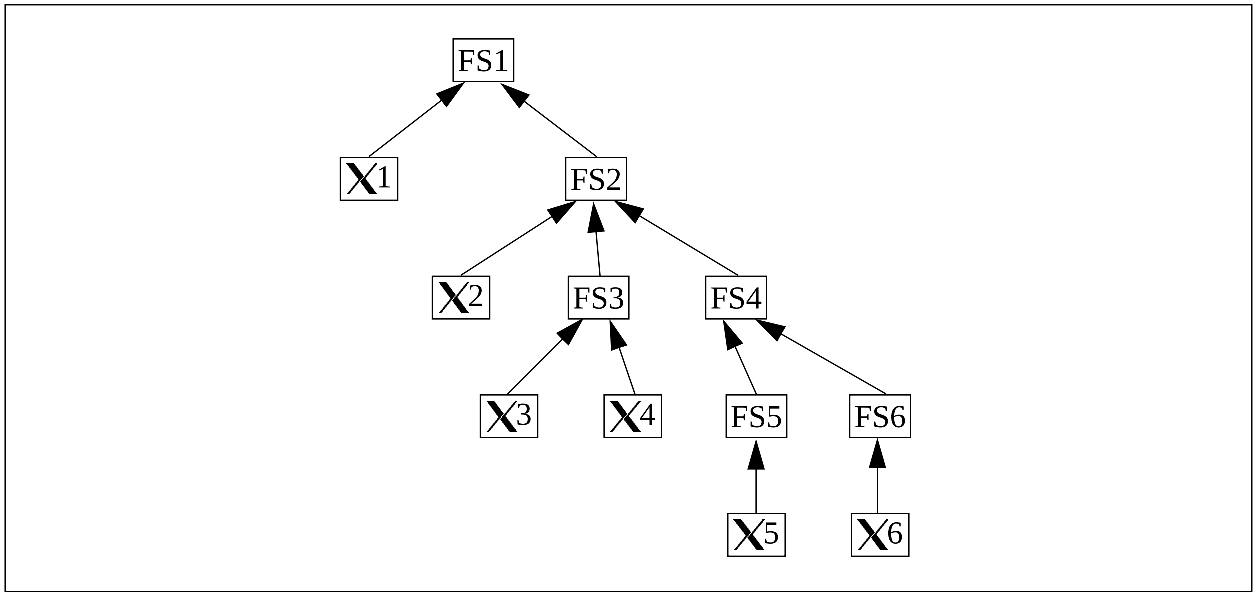


Figure 9: A network of X display and font servers; edges indicate client → server relationship

The font server **FS1** in figure 9 can receive requests from 6 X servers, passing through as many as three chained font servers. For **OpenFont**, **ListFonts**, and **ListFontsWithInfo** requests from any X server, the *hp-hostname-1* protocol identifies which X server has issued the request.

Table 1: Semantics of the hp-hostname-1 Authorization Protocol

Name	"hp-hostname-1" (null-terminated)
X Server Semantics	data = full TCP hostname (null-terminated)
Font Server Semantics	data = data field of client on whose behalf request is issued
Challenges	none expected or handled

When, for example, server **X5** issues an OpenFont request, it sends an authorization protocol packet with the name field "hp-hostname-1" and the data field containing its TCP hostname. All chained font servers — **FS5**, **FS4**, and **FS2** — pass this information upstream, allowing **FS1** to ascertain the hostname of the X server requesting the font by examining the client structure. This information can be used by the font server to implement host-based font licensing.

For X servers or font servers not supporting the protocol, upstream font servers will see either an empty data string or no authorization protocol in the client structure. For example, if **FS3** does not support *hp-hostname-1*, **FS1** and **FS2** will see no *hp-hostname-1* data for requests from **X3** or **X4**.

Limitations

The protocol is non-authenticating, and therefore not secure. For licensing applications for which this is unsatisfactory, the sample implementation of the *hp-hostname-1* protocol can serve as a template for a more robust mechanism.

Related Work

Some related work carried out in the X11R6 font code:

Hp-printername-1 Protocol

A related protocol, *hp-printername-1*, is supported by the sample font server to enable licensing for printers. As with *hp-hostname-1*, the font server logic passes *hp-printername-1* data to upstream font servers. The contents of the data field have not yet been defined.

Font Caching

Implementation of *hp-hostname-1* was accompanied by some fixes and enhancements to the font-caching logic: the *cachable* flag in the **FontInfoRec** and **fsOpenBitmapFontReply** structures is now observed. If a font obtained from a rasterizer or an upstream font server is cachable, any font or X server will willingly share that font with other clients. If the font is not cachable, a request to open the font is always referred back to the font source.

Referring to figure 9, if a font obtained by **X5** from **FS1** is not cachable, and **X6** requests the same font, **FS4**, **FS2**, and **FS1** will not automatically share the font. **FS1** will receive the request and pass it to the rasterizer that created the font. That rasterizer, after exercising its licensing logic, must provide the font. A new parameter has been added to the end of the **OpenScalable**() and **OpenBitmap**() argument lists in the rasterizers: a **FontPtr** pointing (if non-null) to the existing instance of the non-cachable font. If the rasterizer decides to share the font, it can return that pointer instead of creating another instance of the font.

Scalable Aliases

The font name aliasing mechanism has been enhanced to support scalable names. If an alias in **fonts.alias** consists of a scalable font name in both the source and destination names, then the alias applies to all sizes of the font, and enhancements specified in the source name are applied to the destination name.

For example, an alias entry of:

```
-foo-bar-medium-r-normal--0-0-0-0-c-0-iso8859-1 \
  -misc-fixed-medium-r-normal--0-0-0-0-c-0-iso8859-1
```

will map any request for the "`-foo-bar`..." font to an equivalent request for the "`-misc-fixed`..." font. An attempt to open font

```
-foo-bar-medium-r-normal--0-[24 0 0 12]-110-110-c-0-iso8859-1[65_67]
```

will result in opening

```
-misc-fixed-medium-r-normal--0-[24 0 0 12]-110-110-c-0-iso8859-1[65_67] .
```

Scalable Aliases and the Matrix XLFD Enhancement

Scalable alias entries can incorporate matrices to generate font variations. A matrix specified in either the pixelsize or pointsize (but not both) field of the destination name will be multiplied by the source pointsize and pixelsize matrices when generating the destination. Multiplication order is: [*destination matrix*] = [*alias matrix*] • [*source matrix*].

For example, an alias entry of:

```
-misc-fixed-medium-o-normal--0-0-0-0-c-0-iso8859-1 \
  "-misc-fixed-medium-r-normal--0-[1 0 .3 1]-0-0-c-0-iso8859-1"
```

defines an obliqued version of the font; requesting

```
-misc-fixed-medium-o-normal--0-120-110-110-c-0-iso8859-1
```

(or its XLFD matrix equivalent) will result in opening the obliqued font:

```
-misc-fixed-medium-r-normal--0-[12 0 3.6 12]-110-110-c-0-iso8859-1 .
```

Similar entries could be devised to achieve other variations:

```
-misc-fixed-medium-r-wide--0-0-0-0-c-0-iso8859-1 \
  "-misc-fixed-medium-r-normal--0-[1.5 0 0 1]-0-0-c-0-iso8859-1"
-misc-fixed-medium-r-narrow--0-0-0-0-c-0-iso8859-1 \
  "-misc-fixed-medium-r-normal--0-[.75 0 0 1]-0-0-c-0-iso8859-1" .
```

Conclusion

X11 font technology is still very much in evolution. This document has examined four of the major font technology enhancements offered in X11R6, and pointed the way to possible future work: utilities for client-side support of the Matrix XLFD Enhancement; smarter glyph caching; implementation of deferred glyph rasterization in the sample rasterizers; use of font authorization protocols for licensing.

Acknowledgments

The design of the Matrix XLFD Enhancement is due primarily to Paul Asente (Adobe Systems Inc.) Paul’s design evolved through lively discussion in the fontwork mailing list with other fanatically interested parties, including Jim Graham (Sun Microsystems), Axel Deininger (Hewlett-Packard Company), Bob Scheifler, Stephen Gildea, and Dave Wiggins (all of the X Consortium staff), and the author.

The HP XLFD Enhancements were designed primarily by Axel Deininger and Bill Frolik (Hewlett-Packard Company).

Glyph caching was designed to meet the performance needs of HP’s Japanese System Environment, where it received extensive exposure and testing thanks to the efforts of Takumi Ohtani, Hiroyuki Date, Daisaburo Muraoka, and Hideyuki Hayashi.

Scalable aliases were developed independently by Hewlett-Packard Company and Sun Microsystems. The idea of adding matrix transformation capabilities to scalable aliases came from Jim Graham of Sun Microsystems.

Author Information

Nathan Meyers (*nathanm@cv.hp.com*) is a Member of Technical Staff at Hewlett-Packard Company, and lead engineer for HP’s X Font Technology. He authored the sample implementation of the enhancements described in this article.