The Xft Font Library: Architecture and Users Guide

Keith Packard
XFree86 Core Team, SuSE Inc.
keithp@keithp.com

Abstract

The X Render Extension provides a new glyph rendering architecture based on client-side glyph and font management. While this resolves many tough technical issues relating to the extension design, it places the burden of rasterizing, configuring and customizing font usage on every X client.

The Xft library was written to provide X applications a convenient interface to the FreeType font rasterizer and the Render extension. As FreeType provides for no configuration or customization, Xft also performs this task. Xft provides new font naming conventions, sophisticated font matching and selection mechanisms and sufficient abstrations to permit common applications to benefit from Render extension based text output while still working on X servers without support for this extension.

1 Introduction

The X Render Extension[Pac01] moves fontfile access and glyph image generation from the X server to the X client. Client-side glyph management provides several advantages for X applications: full font-file access, application-specific fonts, incremental rasterization and the ability to share known fonts with other parts of the environment. Additionally, as the underlying rendering mechanism refers only to images and not glyphs, the glyph rasterization technology and even the font file format itself no longer depends on the capabilities of the X server. Integrating new font technologies can now move at the rapid speed of individual application development rather than the glacial speed of the general availabilty of new X server enhancements.

With the X server no longer in charge of managing font file access and rasterization, a new library was needed to perform this task on the client side of the wire. As the Render extension was designed to support anti-aliased graphics, this new library needed to provide support for high-quality anti-aliased glyph rasterization.

The FreeType project[TT00] has developed a complete font rasterization engine for most outline font formats along with standard X PCF bitmap fonts. The Render extension takes glyph images and presents them on the screen. All that was needed was a thin layer of glue code to fit between FreeType and the Render extension to enable applications to present high-quality text on the screen.

For X servers not supporting the Render extension, the library should provide access to "core" fonts (fonts accessed through the original core X protocol). This permits applications to move to the new library without leaving legacy X server support behind.

The FreeType library shies from specifying how font files are located; instead it requires that applications provide font file names from which glyphs are loaded. This places the burden of configuring and customizing the set of available fonts outside of this library. Clearly, this new "glue" layer would have to include this ability to be usable in a desktop environment.

2 X Render Extension Glyph Management

The X Render extension provides a couple of simple abstractions for applications to manage glyphs. Each Glyph contains the "alpha" mask (a rectangular image of opacity values) covering the glyph shape, an offset from the origin of the alpha mask to the nominal glyph origin and a default escapement, with both horizontal and vertical offsets, to the next glyph. A collection of Glyphs are contained in a GlyphSet and are numbered by the application with arbitrary 32-bit indices.

Applications draw text by sending a GlyphSet identifier and a list of indices in that set. The server sets each glyph by rendering the alpha mask at the specified drawing location adjusted by the glyph offset. Successive glyphs are positioned by adding the escapement vector to the current origin. Just as with the core PolyText requests, sequences of glyphs can be alternated in the same request with position adjustments and GlyphSet changes. This allows a complex string to be rendered in a single operation.

As the required set of operating system language and locale support expands to cover more of the world's peoples, the number of glyphs included in most fonts also increases. Popular outline fonts contain thousands of glyphs today. Where incrementally rendering glyphs was considered a reasonable optimization ten years ago, it is now an essential part of any font mechanism to minimize the memory usage of each font and reduce the time taken when accessing a new font. The Render extension provides for such incremental rasterization by allowing Glyphs to be added when needed to an existing GlyphSet. No information flows from the X server to the client when adding new Glyphs which makes the process entirely asynchronous. This asychrony permits reasonable performance even in the face of high network latency.

As applications transmit images of each glyph they display, the X server conserves memory by sharing identical glyphs wherever possible.

3 FreeType Library

The FreeType project was originally formed to build a freely available rasterizer for TrueType fonts. The first version of the FreeType library provided a high-quality TrueType rasterizer essentially matching existing systems.

The second version of the FreeType library generalizes the rasterizer internals to provide support for many more formats—Type-1, OpenType and CID outline formats are now supported along with the current standard X "Portable Compiled Format" used for bitmap fonts.

FreeType provides interfaces for measuring and rasterizing glyphs as well as mechanisms to access tables within the font file for kerning and glyph substitution of various forms. This provides applications with the data needed to position glyphs for a variety of locales, as long as the underlying font includes the necessary tables.

As the FreeType project was clearly building a general font library, the burden of developing a new library within XFree86 was significantly reduced by adopting this existing system and providing "glue code" to adapt the FreeType data structures to those needed by the Render extension. This does expose applications to changes in the FreeType library, but as FreeType is a mature project, such changes are likely to be less severe than would be required from any new library developed by XFree86.

Font naming and configuration are not a part of the FreeType library. This particular chore is left to the application. As FreeType is used in many environments, some of which don't even have a file system, this design serves to ensure that FreeType remains free of system policy while providing for the widest possible use. Providing such support turned out to be the hardest part of Xft, and the part which will likely be replaced in the near future.

4 XLFD Font Naming

The core protocol provides for unstructured string font names. While font listing specifies that '?' and '*' behave as the do in the shell, font opening places no specific meaning on them. The X Logical Font Description[SG92] (XLFD) was designed to place some structure on the format of these string names. As outline fonts for desktop computing was a relative novelty when X was developed, the core protocol and the XLFD were both designed based on bitmapped fonts. The semantics and syntax surrounding scalable names was added after a significant period of XLFD-based development had already occurred.

The intent of the font name syntax in the XLFD was to provide applications sufficient information about the font from the name alone permitting font selection and font list presentation to be performed without reference to the underlying font data itself.

The XLFD also provide a standard policy for opening fonts using names containing '?' and '*'. For such names, the font selected is that which would have been returned first in response to a request to list fonts using the same pattern. Unfortunately, this doesn't ensure that '*' is a reasonable default value as the server keeps font names internally sorted within each font directory for efficient searching. For example, when attempting to use '*' for the font weight the server will list 'bold' before 'normal'.

Where this policy really failed was in the mapping of point sizes to pixel sizes. XLFD provides pixel size, point size and resolution in both axes in the font name. The standard X fonts are segregated by resolution, separate '75dpi' and '100dpi' directories contain fonts at various point sizes for those resolutions. Additional directories of fonts are generally rasterized for 75dpi screens.

The protocol directs the X server to search the font directories in the order presented by the font path. This makes the font path dictate which resolution is preferred. If the 100dpi directory is listed first, applications specifying '*' for the resolution fields will use 100dpi fonts where they exist and 75dpi fonts otherwise.

Applications specifying only point size and using '*' for resolution end up with a collection of random sized fonts—those found in the 100dpi directory will be rasterized for a 100dpi screen, the remainder of the fonts are generally rasterized for a 75dpi screen and appear smaller as a result.

The end result is that XLFD font matching is fraught with peril; applications attempting it often resort to listing available fonts and then presenting fully qualified XLFD names back to the server.

One further issue with the XLFD is that it includes the average width of the glyphs as a part of the name. While this is a useful piece of information for applications interested in selecting among various setwidths, and is easily computed for bitmapped fonts, outline fonts can't generally compute this without rasterizing all glyphs at the desired size. Simply listing the fonts available at a particular size involves rasterizing every glyph for every font.

XLFD provides useful information about the available fonts, and except for the average width field, this information is easily computed and delivered back to applications. Applications using XLFD names should avoid using server-side font matching and instead manage XLFD names locally, generating appropriate fully qualified names using information gleaned from listing the available fonts.

As XLFD names don't provide for reasonable matching semantics, a new scheme was required to permit the underlying font system to locate a suitable font given a set of constraints provided by the application. Such a system needed to be flexible enough to encompass unimagined new font properties while not requiring that applications fully specify every aspect of the font.

5 Designing a New Library

Xft interacts with its environment in three areas; with applications through a programming interface, with the system through configuration files and with users as they provide font names. While these three areas interact within the library, they are separable from the design perspective.

5.1 Application Interface Design

The primary goal of Xft was to connect the output of the FreeType rasterizer with the Render extension. However, to gain some acceptance of Xft as a replacement for the existing Xlib text output routines, a secondary goal was to also support core X fonts, albeit with some compromise on application functionality.

Because FreeType doesn't provide for font selection, a portion of Xft provides for font matching. Adopting the existing XLFD mechanism would have significantly restricted the capabilities for font matching, so Xft provides a new format. This selection mechanism is designed to always match some font; this permits applications to assume that suitable fonts exist and avoid requiring extensive fallback mechanisms at every level.

Another requirement is that the library provide reasonable matches, such as substituting oblique fonts when italic is requested and using medium weight fonts when no weight is specified. This permits applications to specify fonts without all of their characteristics and expect reasonable results. In cases where the application request contains conflicting requirements, policy about which characteristics are most important provides resolution.

To simplify text encoding options at the library level, all text output routines accept only Unicode-encoded data. Other encodings are converted by the application, either at the boundary with the operating system or between the application and Xft. As existing fonts generally provide a Unicode encoding, or one easily converted to Unicode, this dramatically simplifies the internals of the library while exposing a consistent view to the application.

Another important goal was to minimize the exposure of internal data structures to applications. As the Render extension permits incremental downloads of new glyphs, Xft is designed to rasterize new glyphs on demand. This requires that metric information be requested through a function rather than directly accessed from the data structures as is done in Xlib.

Finally, as Xft is not designed as a complete font access library, the underlying FreeType data types are exposed for applications to use directly with FreeType itself. This reduces the complexity of Xft while still allowing applications full access to the available font information.

5.2 Font Naming Design

The design of font names for Xft started with the notion that font names represent properties of the font, either properties desired by an application or properties presented by the font itself. As the requirements of an application are not fixed, and the characteristics of fonts continue to be enhanced, an idea emerged of representing a font name as a variable list of name/values pairs. As a final enhancement, one or more values could be associated with each name so that an application might place specific requests along with more general characteristics (e.g. requesting the 'Times' face, failing that, any font with serifs.)

This unifies the representation of application font requests and available fonts. An "XftPattern" is a collection of named properties; each property holds one or more values. Each available font is described as an XftPattern providing characteristics of the font. Applications build XftPatterns describing their requirements which are then matched against the XftPatterns describing all of the available fonts. The available font most closely matching the application specification is selected. This mechanism ensures that every application font request will be matched by a font.

Finally, a textual representation for XftPatterns was designed. This permits applications to be configured much as they are today by using a string to select a font.

XftPatterns provide a simple and extensible mechanism for communicating requirements and capabilities between the application and the fonts it uses. They are discussed in additional detail in Section 6.

5.3 Font Listing Design

The core protocol provides primitive support for querying the set of available fonts; a simple shell-style pattern is passed to the X server and the set of font names matching that pattern is returned. For applications interested in discovering the set of available font families, the list of returned fonts includes essentially all of the fonts in the system; the application is responsible for extracting the needed information from this mass of data.

What was needed, instead, was the ability to ask the system for specific relevant information and to have it discard redundant data. Applications could then focus on examining and manipulating the information, rather than expend significant effort parsing XLFD strings.

Xft splits the data needed to return information about available fonts into two pieces. The first is a pattern which is used to select from the available fonts. The second is a set of property names used to determine what information the application is interested in receiving. The list of matching fonts would be pared to eliminate entries with duplicate values in the selected properties.

Finally, the notion of "best match" isn't useful in listing fonts; applications use font lists to generate dialogs for the user or to discover capabilities of a family of fonts. Instead of measuring the distance from the requested pattern to the available fonts, an exact match is required for all elements of the pattern.

5.4 Configuration Design

To avoid the possible proliferation of incompatible font configuration and customization mechanisms, a standard was needed. As no existing standard existed, adding configuration to Xft at least ensured that X applications could share fonts and name them in the same way.

An essential customization requirement is to allow one font to stand-in for another. These "aliases" help maintain the general appearance of text by selecting faces with similar characteristics to unavailable faces. Another useful role is in supporting generic 'mono', 'sans' and 'serif' faces which applications can use to follow the users preference in each of these common cases.

Another goal was to make adding fonts as easy as possible. The core X font configuration mechanism uses the X server font path to list the set of directories in which to find fonts. In each of these directories, two separate configuration files must be generated which map font names to file names. If these files are missing or corrupted, font selection will not work properly. By eliminating such configuration files, the overall system would be more robust.

Finally, users and administrators need to customize the rasterization of specific fonts. Some users wish to avoid anti-aliased text at a range of sizes, or for specific faces. Others need to adjust the size or spacing of certain fonts.

As it turns out, Xft is not quite the right place for all such configuration—X applications aren't the only ones interested in accessing fonts and font information. Moving this configuration mechanism out of Xft and into a library usable by non-X applications will be a focus of development in the near future.

6 Xft Font Names

Font names in Xft are represented as a list of named properties, each of which carries a list of typed values. This collection of properties is stored in an XftPattern data structure. There are routines to create XftPatterns, edit them and perform matching operations against the list of available fonts. Xft has a list of properties, shown in Table 1, which it supports internally; there is no restriction that applications use only these, Xft will ignore those it doesn't understand.

To permit XftPatterns to be transmitted and stored, a string representation exists which can capture the structure of a pattern. The format of these strings is shown in Figure 1. Each of the supported names has an implicit type which is used to parse the associated values. This reduces the need for quoting or other lexical mechanisms to distinguish amongst the various types. As is shown, named constants can take the place of either values or complete name=value pairs. As the constants uniquely determine their associated property name, there is no need to supply that as well. Refer to Table 2 for a list of the available constants. Each of these constants is represented internally by a numeric value; this permits similar matches by comparing the value from the available fonts with the value requested by the application. An application requesting 'demibold' will select 'bold' instead of 'medium'. This helps ensure that the intent of the application is fulfilled, even when the available fonts don't match those present during development and testing.

Name	Type	C Name
family	String	XFT_FAMILY
style	String	XFT_STYLE
slant	Int	XFT_SLANT
weight	Int	XFT_WEIGHT
size	Double	XFT_SIZE
pixelsize	Double	XFT_PIXEL_SIZE
encoding	String	XFT_ENCODING
spacing	Int	XFT_SPACING
foundry	String	XFT_FOUNDRY
core	Bool	XFT_CORE
antialias	Bool	XFT_ANTIALIAS
xlfd	String	XFT_XLFD
file	String	XFT_FILE
index	Int	XFT_INDEX
rasterizer	String	XFT_RASTERIZER
outline	Bool	XFT_OUTLINE
scalable	Bool	XFT_SCALABLE
rgba	Int	XFT_RGBA
scale	Double	XFT_SCALE
render	Bool	XFT_RENDER
minspace	Bool	XFT_MINSPACE
dpi	Double	XFT_DPI
charwidth	Int	XFT_CHAR_WIDTH
charheight	Int	XFT_CHAR_HEIGHT
matrix	Matrix	XFT_MATRIX

Table 1: Xft Font Name Properties.

Name	Constant	Value
weight	light	0
	medium	100
	demibold	180
	bold	200
	black	210
slant	roman	0
	italic	100
	oblique	110
spacing	proportional	0
	mono	100
	charcell	110
rgba	rgb	1
	bgr	2
	vrgb	3
	vbgr	4

Table 2: Xft Font Name Constants.

```
name          : families sizes properties
families      : family-names
              |
family-names  : family-names , string
              | string
sizes         : - size-list
              |
size-list     : size-list , number
              | number
properties    : properties : property
              |
property      : string = value
              | named-constant
value         : string
              | number
              | boolean
              | named-constant
              | matrix
matrix        : number number number number
```

Examples:

```
times,serif-12:italic
courier,mono-14:matrix=1 .1 0 1
```

Figure 1: Xft Font Name Syntax.

Additionally, while XLFD names are not ideal, they are common in existing X applications and do represent a set of desired font characteristics. Xft can convert an XLFD name into an XftPattern which can then be used to select a font using Xft matching rules rather than the XLFD matching rules described in Section 4.

Xft font names are designed to be extensible so that existing applications will continue to to operate correctly even while the Xft/FreeType2 interface continues to grow and provide pattern elements for newer systems.

7 Xft Configuration File

Using the core protocol, all applications are ensured of having access to all of the available fonts as it is the X servers responsibility to locate them. With font management now moving to the client side, locating fonts becomes the responsibility of the application. Without a central shared configuration mechanism, the set of available fonts for each application may well be different. Installing and selecting fonts would be problematic and error prone.

The main job, then, of the Xft configuration file is to specify the location of available font files. Secondary to this is the role of adjusting font selection and customizing rasterization parameters.

By default, the Xft configuration file 'XftConfig' is found in /usr/X11R6/lib/X11. This can be overridden by specifying an alternate filename in the XFT_CONFIG environment variable.

Comments can be placed anywhere in the file with the '#' character and extend to the end of the line. The remaining syntax will be exposed in subsequent sections.

7.1 Font Directories

Xft takes a simplistic view in configuring where fonts are to be found. A list of directories are specified and Xft searches those for font files, all of the font files it finds are added to the list of available fonts for matching. The order within the directories is irrelevant; Xft always searches for the best match among all of the fonts. Directories are specified in the config file with a line of the form:

```
dir "/usr/X11R6/lib/X11/fonts/Type1"
```

No special configuration of the directory is required, Xft scans the directory looking for font files automatically.

7.2 Nested Configuration Files

To split the configuration file into manageable parts, and to allow for per-user customization of the library, the Xft configuration file allows for lines of the form:

```
include "/usr/local/lib/XftAliases"

includeif "~/.xftconfig"
```

The only difference is that the first form will cause Xft to emit a warning message when the referred file can't be found. The '~' character refers to the user's home directory; as the Xft configuration file is parsed by the user's applications themselves, this permits per-user customization without referring to the X display.

7.3 Font Pattern Editing

To provide for font substitution and other adjustments in the font matching process along with allowing for the configuration of the rasterization process, the Xft configuration file can contain operations which modify the XftPattern before the matching process is done. These operations are called "editing commands" and operate by matching the incoming pattern and modifying it. Each command is executed in the order it appears in the configuration file. The syntax for these commands is found in Figure 2.

The match clauses select which patterns are selected for editing; all must be true for the edit to be performed. Matches prefixed with 'any' are true if any of the values associated with the named field satisfy the condition. Matches prefixed with 'all' are true only when all of the values satisfy the condition.

The edit clauses operate on the fields of the pattern and can either replace or amend the values found for the associated field. If the match clause contains a reference to the same element of the pattern, the matching value is marked.

Edits using the '+=' operator will insert their value before the marked value. Edits using the '=+' operator will insert their value after the marked value. Edits using the '=' operator will replace the marked value with their value. If no value is marked, the '+=' and '=+' operators prepend/append their associated value to the list of values while the '=' operator replaces all existing values with its associated value.

The operators within the expressions operate in the obvious manner, one possibly unexpected operation is that the '+' operator can operate on strings and will concatenate them. Fields within the font may be referred to within an expression; the first value associated with the field is used.

After using this mechanism for several months, it has become clear that it is very complex and yet incapable of performing expected tasks. One deficiency is that the edits affect only the incoming pattern and do not operate on the matched font. This makes the example of disabling anti-aliasing for LuciduxSerif fonts produce unexpected results when presented with a name of the form:

Times,LuciduxSerif,serif-10

The 'match' clause will succeed with this pattern, causing this Times font to be displayed without anti-aliasing.

Another issue is that font aliases are difficult to specify and still don't provide precisely the desired semantics. There are two possible semantics desired for font aliases; the first is that certain faces should *always* be substituted for others, the second is that certain faces *may* be substituted for others when those are not found. An explicit statement of this kind would both clarify the file format while improving the font matching semantics.

8 Font Matching

The goal for font matching in Xft is to accept a collection of characteristics from the application and return the best font from all of those available. The application provides a font specification in the form of an XftPattern to the XftFontMatch function. The pattern is edited as described in Section 7.3. X resources are then used to modify the pattern as shown in Table 3. If a pixel size was not specified by the application, it is computed by taking the specified point size (if specified, else 12.0), multiplying that by the scale factor and then converting it from points to pixels using the specified dpi value. The

```
command   : match tests edit edits
tests     : test tests
          |
test      : any | all name compare value
compare   : == | != | < | <= | > | >=
edits     : edit edits
          |
edit      : name eqop expr ;
eqop      : = | += | =+
expr      | value
          | name
          | expr binop expr
          | ! expr
          | expr ? expr : expr
binop     : || | && | == | != | < | <= | > | >= | + | - | * | /
```

Examples:

```
# Use LuciduxSerif as default serif'ed font
match any family == "serif"
edit      family += "LuciduxSerif";
# Avoid using anti-aliasing at some sizes for LuciduxSerif face
match any family == "LuciduxSerif" any size < 14 any size > 8
edit      antialias = false;
```

Figure 2: Xft Pattern Editing Syntax.

X Resource	Type	Effect	Default
Xft.render	Bool	Directs Xft to use client-side fonts	HasRender
Xft.core	Bool	Directs Xft to use server-side fonts	!HasRender
Xft.antialias	Bool	Selects whether glyphs are anti-aliased	True
Xft.rgba	Number	Specifies sub-pixel order on LCD screens	0
Xft.minspace	Bool	Eliminates extra leading between lines	False
Xft.scale	Number	Scales point size of all fonts	1.0
Xft.dpi	Number	Used to convert point size into pixel size	Vertical screen dpi

Table 3: Adjusting Xft values with X Resources.

Order	Name	Type
1	foundry	String
2	encoding	String
3	antialias	Bool
4	family	String
5	spacing	Number
6	pixelsize	Number
7	style	String
8	slant	Number
9	weight	Number
10	rasterizer	String
11	outline	Bool

Table 4: Matching Order of Xft Field Comparisons.

other X resources are used as default values for the associated pattern elements.

The fully specified pattern is then compared with all available fonts. Only the fields shown in table 4 are used in this comparison. The order of fields in this table is significant; any mismatch in elements earlier in the table override all matches in later elements. Missing elements in a pattern or font are tacitly matched. Numbers are measured by the magnitude of their difference. This is how 'oblique' selects 'italic' instead of 'roman'.

Field with multiple values are biased to prefer fonts which match earlier in the list; this way the name

Times,LuciduxSerif,serif-10

will prefer to match 'Times' over 'LuciduxSerif'.

Once a font has been selected, an XftPattern is built which describes that font precisely allowing the application to discover what font was actually selected. This pattern includes information about the underlying font file itself allowing applications direct access to any information unavailable through FreeType. Fields found in the pattern that don't occur in the font are added; this allows applications to communicate information in font patterns, both to rasterizers and also within the application itself.

9 Core X Font Handling

While the main goal of Xft is to provide client-side font support using FreeType and the Render extension, it seemed reasonable to also provide some application compatibility by using core X fonts where the Render extension was unavailable. This, by necessity, restricts the capabilities of the library, but even so, many applications can use Xft functions with either core or Render fonts without change. Others need only minor adjustments to notice when FreeType functions cannot be used.

Core font handling takes two pieces; selection of fonts and rendering. The selection of fonts is done by listing the available X fonts and converting those into XftPattern data structures and then accepting application patterns for matching. As the matching is based on a "nearness" metric instead of simple shell pattern matching, the resulting matching is generally more useful than that specified by the XLFD. This can simplify applications which before had significant mechanism devoted to core X font matching.

Once a core font is selected, rendering becomes a simple matter of having the Xft routines call the standard Xlib text drawing routines. The only difficulty is in mapping the Unicode glyphs provided by the application into the encoding supported by the font.

One effect of this merging is that the general Xft rendering routines expose no capabilities not provided by both Render and core fonts. Applications may specify a translucent color value for drawing, but it will be ignored when using core fonts. Similarly, there's no provision for selecting a raster operation or compositing operator as each is supported by only one underlying renderer.

The result is a new way of using the core fonts which is in many ways easier to use and more powerful than the Xlib equivalents.

10 An Overview of Xft Interfaces

The Xft programming interface can be easily separated into three distinct areas: those dealing with XftPatterns and matching fonts, those dealing with drawing glyphs on the screen and a small section providing an interface to the underlying FreeType library.

10.1 Xft Data Structures

XftValue

```
typedef enum _XftType {
    XftTypeVoid,
```

```
        XftTypeInteger,
        XftTypeDouble,
        XftTypeString,
        XftTypeBool,
        XftTypeMatrix
    } XftType;

    typedef struct _XftValue {
        XftType       type;
        union {
            char      *s;
            int       i;
            Bool      b;
            double    d;
            XftMatrix *m;
        } u;
    } XftValue;
```

An XftValue holds one value for an element of an XftPattern. It should be treated as a tagged union; set and check the ‘type’ field before using the union elements. Storage for the string and matrix elements is separate. Xft always copies the data from these pointers when receiving XftValue structures from the application, this leaves applications free to use static or stack-based storage as appropriate.

XftPattern

```
typedef struct _XftPattern
    XftPattern;
```

An XftPattern is an opaque structure holding a list of named elements, each of which holds a list of XftValues. Xft provides interfaces to build and query these patterns.

XftFont

```
typedef struct _XftFont {
    int ascent;
    int descent;
    int height;
    int max_advance_width;
    Bool core;
    XftPattern *pattern;
    union {
        struct {
            XFontStruct *font;
        } core;
        struct {
            XftFontStruct *font;
        } ft;
    } u;
} XftFont;
```

The XftFont data structure is returned when opening fonts and is used in drawing glyphs. The visible members provide a modicum of information about the font. The ‘core’ value will be True if the underlying font is a core X font, in which case the u.core.font field will point at an XFontStruct. Otherwise, the underlying font is a FreeType font and the u.ft.font field refers to an XftFontStruct.

XftFontStruct

```
typedef struct _XftFontStruct
    XftFontStruct;

struct _XftFontStruct {
    FT_Face face;
    GlyphSet glyphset;
    int min_char;
    int max_char;
    FT_F26Dot6 size;
    int ascent;
    int descent;
    int height;
    int max_advance_width;
    int spacing;
    int rgba;
    Bool antialias;
    int charmap;
    XRenderPictFormat *format;
    XGlyphInfo **realized;
    int nrealized;
    Bool transform;
    FT_Matrix matrix;
};
```

This structure should probably become opaque with suitable accessor functions to reach the appropriate internal fields. The most useful member is likely to be the ‘face’ which refers to the underlying FreeType object, however, as keeping that object resident in memory is rather expensive, it may be that this needs to become cached and loaded on demand. Applications wanting to use this field are wise to prepare for its eventual disappearance by wrapping access in a macro. An audit of which other fields of this structure should be visible is needed.

XftDraw

```
typedef struct _XftDraw
    XftDraw;
```

An XftDraw encapsulates the state needed to render glyphs to an X drawable. For the Render extension, a Picture is needed while for core X, a GC and associated pixel values are required.

XftColor

```
typedef struct _XftColor {
    unsigned long   pixel;
    XRenderColor    color;
```

```
} XftColor;
```

The Render extension needs RGBA while core X requires a pixel value. An XftColor holds both, and routines exist to initialize and free any associated resources. For TrueColor visuals, the allocation routine avoids a round trip by computing the pixel value locally, so there's no performance penalty in that case. When it is know that the Render extension will be used, the 'color' member can be initialized manually leaving the 'pixel' value unset.

XftObjectSet

```
typedef struct _XftObjectSet
    XftObjectSet;
```

The font listing mechanism within Xft uses an XftObjectSet to restrict the amount of data returned to the application. An XftObjectSet holds a list of field names.

XftFontSet

```
typedef struct
    _XftFontSet {
    int nfont;
    int sfont;
    XftPattern **fonts;
} XftFontSet;
```

XftFontSets reference a set of patterns and are used as the return value from the font listing functions.

XftResult

```
typedef enum _XftResult {
    XftResultMatch,
    XftResultNoMatch,
    XftResultTypeMismatch,
    XftResultNoId
} XftResult;
```

Functions that return a search result use this data type to present their findings. XftResultMatch indicates that the object was found, XftResultNoMatch says that no matching object was found. XftResultTypeMismatch says that an object was found, but it was of the wrong type while XftResultNoId says that an object was found, but that there were fewer values than requested.

10.2 Font Pattern Manipulation

Many Xft operations involve XftPattern objects. As those should not be directly accessed by the application, a set of routines exist to manipulate them.

XftPatternCreate

```
XftPattern *
XftPatternCreate (
    void)
```

This creates an empty pattern.

XftPatternDuplicate

```
XftPattern *
XftPatternDuplicate (
    XftPattern *p)
```

A new pattern is created which contains all of the values from 'p'. Values referring to other memory (strings and matrices) are copied to newly allocated storage as well.

XftPatternDestroy

```
void
XftPatternDestroy (
    XftPattern *p)
```

All referenced storage, including any strings and matrices referred to by values within the pattern, is freed.

XftPatternAdd

```
Bool
XftPatternAdd (
    XftPattern *p,
    const char *object,
    XftValue value,
    Bool append)
```

'value' is added to the list of values for the field 'object'. If 'append' is True, the value is added to the end of the list, otherwise it is insert at the head. If 'value' refers to a string or matrix, XftPatternAdd will allocate new storage and copy the object into it. XftPatternAdd returns False only when allocating space for this operation fails. This function forms the basis for the remaining XftPatternAdd functions.

```
Bool
XftPatternAddInteger (
    XftPattern *p,
    const char *object,
    int i)

Bool
XftPatternAddDouble (
    XftPattern *p,
    const char *object,
    double d)

Bool
XftPatternAddString (
    XftPattern *p,
```

```
        const char *object,
        const char *s)

    Bool
    XftPatternAddMatrix (
        XftPattern *p,
        const char *object,
        const XftMatrix *s)

    Bool
    XftPatternAddBool (
        XftPattern *p,
        const char *object,
        Bool b)
```

Each of these functions creates a temporary XftValue with the appropriate type and value and passes it to XftPatternAdd with the 'append' value set to True.

XftPatternGet

```
    XftResult
    XftPatternGet (
        XftPattern *p,
        const char *object,
        int id,
        XftValue *v)
```

XftPatternGet searches the indicated pattern for an element whose name matches 'object'. It then searches along the list of values for that name for the 'id'th element (starting at zero) and stores the resulting value in v. It does not allocate new storage for strings or matrices, so applications must ensure that the returned value not be referenced beyond the life of the XftPattern itself. This function forms the basis for the remaining XftPatternGet functions.

```
    XftResult
    XftPatternGetInteger (
        XftPattern *p,
        const char *object,
        int n,
        int *i)

    XftResult
    XftPatternGetDouble (
        XftPattern *p,
        const char *object,
        int n,
        double *d)

    XftResult
    XftPatternGetString (
        XftPattern *p,
        const char *object,
        int n,
        char **s)

    XftResult
    XftPatternGetMatrix (
        XftPattern *p,
        const char *object,
        int n,
        XftMatrix **s)

    XftResult
    XftPatternGetBool (
        XftPattern *p,
        const char *object,
        int n,
        Bool *b)
```

Each of these convenience functions calls XftPatternGet. If the datatype of the resulting value doesn't match the type of the function, these functions return XftResultTypeMismatch, otherwise they return the same thing as XftPatternGet.

XftPatternBuild

```
    XftPattern *
    XftPatternBuild (
        XftPattern *orig,
        ...)
```

The arguments after 'orig' form a list of pattern names, types and values to add to the specified pattern. If 'orig' is null, a new pattern is allocated. The pattern arguments are of the form:

```
    char *object,
    XftType type,
    union {
        char *s;
        int i;
        Bool b;
        double d;
        XftMatrix *m;
    } u
```

The list is terminated with a NULL object. Here's an example:

```
p = XftPatternBuild (
    0,
    XFT_FAMILY,
      XftTypeString,
      "mono",
```

```
    XFT_SIZE,
      XftTypeDouble,
      12.0,
    0);
```

This function is a convenience function encapsulating several calls to XftPatternAdd in one statement, the semantics are identical to making the equivalent set of XftPatternAdd calls.

```
XftPattern *
XftPatternVaBuild (
    XftPattern *orig,
    va_list va)
```

This takes the a varargs list of arguments in the same format as XftPatternBuild, yielding identical results.

10.3 Font Selection

Xft exposes several levels of font selection functions. The patterns involved can either be implicitly generated using a simple font name string, or they can be explicitly managed by the application and manipulated before being presented to the underlying interface for matching. Starting with the most primitive level, Xft provides:

XftFontMatch

```
XftPattern *
XftFontMatch (
    Display *dpy,
    int screen,
    XftPattern *pattern,
    XftResult *result)
```

This function takes the application-generated XftPattern, performs the config file edits and X resource substitutions and then matches the result against the set of available fonts. The closest matching font name is returned as another XftPattern which contains enough information to open the font with the next function, XftFontOpenPattern. In case of failure, the return value is 0 and an indication of the error is placed in result.

XftFontOpenPattern

```
XftFont *
XftFontOpenPattern (
    Display *dpy,
    XftPattern *pattern)
```

XftFontOpenPattern accepts a matched font pattern and creates an XftFont structure for the font. The returned XftFont contains a reference to the passed pattern, this pattern will be destroyed when the XftFont is passed to XftFontClose.

XftFontOpen

```
XftFont *
XftFontOpen (
    Display *dpy,
    int screen,
    ...)
```

Arguments after 'screen' form an implicit XftPattern as seen in the description of XftPatternBuild in Section 10.2. Here's an example:

```
f = XftFontOpen (
    dpy,
    DefaultScreen(dpy),
    XFT_FAMILY,
      XftTypeString,
      "mono",
    XFT_SIZE,
      XftTypeDouble,
      12.0,
    0);
```

A pattern is created from the arguments and passed to XftFontMatch, the result of that is passed to XftFontOpenPattern which produces the return value for this function.

XftFontOpenName

```
XftFont *
XftFontOpenName (
    Display *dpy,
    int screen,
    const char *name)
```

Similar to XftFontOpen, the 'name' argument forms an implicit XftPattern, XftFontMatcn and XftFontOpenPattern are used in the same fashion to produce a matching XftFont.

XftFontOpenXlfd

```
XftFont *
XftFontOpenXlfd (
    Display *dpy,
    int screen,
    const char *xlfd)
```

This functions precisely as XftFontOpenName except that the XftPattern is generated by parsing the XLFD name 'xlfd'.

XftFontClose

```
void
XftFontClose (
    Display *dpy,
```

```
    XftFont *font)
```

The underlying core or FreeType font object is closed and the pattern referenced by the font is destroyed.

10.4 XftDraw Manipulation

Xft provides an abstraction that masks the differences between core X and Render extension rendering. The XftDraw object provides the target for either mode, wrapping suitable information for either rendering model.

XftDrawCreate

```
XftDraw *
XftDrawCreate (
    Display *dpy,
    Drawable drawable,
    Visual *visual,
    Colormap colormap)
```

This routine creates an XftDraw object referencing mentioned drawable along with its associated visual and colormap. The visual argument is required even for pixmap rendering as it specifies the eventual format for the pixel values.

XftDrawCreateBitmap

```
XftDraw *
XftDrawCreateBitmap (
    Display *dpy,
    Pixmap bitmap)
```

When the rendering target is a 1-bit bitmap, this function is used in place of XftDrawCreate.

XftDrawChange

```
void
XftDrawChange (
    XftDraw *draw,
    Drawable drawable)
```

This function switches the underlying rendering target without affecting any other properties of the XftDraw object. Applications are responsible for ensuring that the drawable uses the same visual as the original drawable.

**XftDrawDisplay,XftDrawDrawable,
XftDrawColormap,XftDrawVisual**

```
Display *
XftDrawDisplay (
    XftDraw *draw)

Drawable
XftDrawDrawable (
    XftDraw *draw)

Colormap
XftDrawColormap (
    XftDraw *draw)

Visual *
XftDrawVisual (
    XftDraw *draw)
```

These functions simply return the associated value from the opaque XftDraw structure.

XftDrawDestroy

```
void
XftDrawDestroy (
    XftDraw *draw)
```

This destroys the XftDraw object and any private data allocated therein. The X drawable referenced is not destroyed..

10.5 Glyph Rendering

With an XftFont and an XftDraw in hand, the next step is to use them together and present some text on the screen. This is relatively simple as the encoding is always Unicode and the only thing that changes is how those glyphs are stored. This section also documents a few other convenience functions for applications wanting to use similar data structures for other operations.

XftTextExtents

```
void
XftTextExtents<8,16,32,Utf8> (
    Display *dpy,
    XftFont *font,
    XftChar<8,16,32,8> *string,
    int len,
    XGlyphInfo *extents)
```

These functions measure the extents of the specified string and returns the metrics in the ‘extents’ structure.

XftDrawString

```
void
XftDrawString<8,16,32,Utf8> (
    XftDraw *d,
    XftColor *color,
    XftFont *font,
    int x,
```

```
        int y,
        XftChar<8,16,32,8> *string,
        int len)
```

Each of these functions displays a single string. For Render text, the string is painted using the Over operator. For core text, it is painted with GXcopy and a full planemask.

XftDrawRect

```
    void
    XftDrawRect (
        XftDraw *d,
        XftColor *color,
        int x,
        int y,
        unsigned int width,
        unsigned int height)
```

This simple function paints a single rectangle in the specified color.

XftDrawSetClip

```
    Bool
    XftDrawSetClip (
        XftDraw *d,
        Region r)
```

The clip list for the specified XftDraw is set from the specified region. Alternative functions taking lists of rectangles should probably be added.

10.6 Font Listing

Xft provides a relatively complex mechanism for listing available fonts. This allows the same mechanism to list available faces as well as available styles for a particular face without returning extraneous data. This is accomplished by having the application declare which fields of the font patterns are important and separately provide a pattern to select matching fonts. For each matching font, an entry for the returned list is generated by extracting only those fields selected by the XftObjectList provided by the application. A unique set of such entries is returned to the application.

Because listing fonts is a fundamentally separate process from opening a single font, the semantics for how patterns match fonts in this process is different than in Section 10.3. Matching in this context requires that each of the elements of the pattern must have one value which precisely matches the associated font element.

XftObjectSetCreate

```
    XftObjectSet *
    XftObjectSetCreate (
        void)
```

Creates an empty XftObjectSet.

XftObjectSetAdd

```
    Bool
    XftObjectSetAdd (
        XftObjectSet *os,
        const char *object)
```

Add a single field name to an XftObjectSet. Storage is allocated for the name and it is copied from the argument permitting the application to release or reuse its own copy.

XftObjectSetDestroy

```
    void
    XftObjectSetDestroy (
        XftObjectSet *os)
```

The XftObjectSet is destroyed along with any other referenced storage.

XftObjectSetBuild, XftObjectSetVaBuild

```
    XftObjectSet *
    XftObjectSetBuild (
        const char *first,
        ...)

    XftObjectSet *
    XftObjectSetVaBuild (
        const char *first,
        va_list va)
```

XftObjectSetBuild creates an XftObjectSet from a list of object names terminated with NULL. This permits the quick construction of a constant set of names without a long sequence of calls and is semantically equivalent to such a sequence of calls. XftObjectSetVaBuild takes an existing varargs list and does the same thing.

XftListFontsPatternObjects

```
    XftFontSet *
    XftListFontsPatternObjects (
        Display     *dpy,
        int     screen,
        XftPattern     *pattern,
        XftObjectSet    *os)
```

For each font matching 'pattern', a new pattern is generated including only those elements specified in 'os'. This function then merges each new pattern into the return value so that only unique patterns are included.

XftListFonts

```
XftFontSet *
XftListFonts (
    Display *dpy,
    int screen,
    ...)
```

The variable arguments include a list of pattern elements as seen in the description of XftPatternBuild in Section 10.2 followed by a NULL. Following that, a list of field names as described by XftObjectSetBuild in this section followed by another null. With the pattern and object set generated, this function calls XftListFontPatternObjects and returns the same result.

10.7 FreeType Access

Where more complex interactions with the underlying FreeType library or Render extension are required, Xft provides some functions to access those more directly.

XftFreeTypeGet

```
XftFontStruct *
XftFreeTypeGet (
    XftFont *font)
```

This returns the underlying XftFontStruct object for the given font, or NULL if the underlying font is not a FreeType font.

XftRenderString

```
void
XftRenderString<8,16,32,Utf8> (
    Display *dpy,
    Picture src,
    XftFontStruct *font,
    Picture dst,
    int srcx, int srcy,
    int x, int y,
    XftChar<8,16,32,8> *string,
    int len)
```

These four related functions all provide a bit more control over the drawing of text with the Render extension. In particular, they provide for an arbitrary source picture which can be used to pattern the text. Applications might also be able to perform more efficient caching of the source Picture data with these routines;

11 Font Information Cache

As described in Section 7.1, the library generates the set of available fonts by scanning each of the directories listed in the configuration files. Discovering the properties of a font requires that the font file be opened by FreeType and a significant amount of the file processed. This can take quite some time, especially with a large number of fonts.

To improve the performance of this operation, Xft caches the results of this search in two places. Each directory may contain a file named `XftCache` which contains a list of fonts, one per line with the font file name, the index within the file of the font and information about the font represented as an XftPattern in string form. For any fonts not found in an `XftCache` file, Xft generates a `.xftcache` file in the users home directory containing the font file name, font index, the file modification time and the XftPattern.

XftCache files are built by the `xftcache` program, a standard part of the XFree86 release. Users are encouraged to run this on other directories added to the Xft configuration file, and to re-run it whenever the contents of those directories change. Xft will automatically manage the contents of each per-user `.xftcache` file to save unknown fonts while eliminating information present in an `XftCache` file.

Xft still scans each directory at startup time, but it first checks the per-directory and per-user caches for the filename before attempting to open it with FreeType. This dramatically reduces applications startup time while retaining the accuracy provided by querying directory contents.

12 Future Directions

Xft started as a relatively small effort to bind the X Render extension with the FreeType rasterizer. It gained new capabilities as needed to provide support for more applications. All of this new functionality has been necessary, but some of it has outgrown this X-centric library. In particular, the font configuration mechanism must be extracted from Xft and placed in a separate library to be shared with printer drivers and other non-X font consumers. This will ensure that fonts can be easily installed and managed for the benefit of all applications, not just those displaying information on the screen.

Xft also needs additional capabilities to support internationalization. Right now, the XftFont object refers to a single FreeType font. If that font doesn't contain all of the glyphs necessary to render a particular document, the missing glyphs will not be displayed correctly. Some mechanism for finding replacement glyphs is needed. The XftFont structure should be extended to optionally perform automatic glyph substitution using multiple underlying FreeType faces.

For applications not wanting automatic substitution, it should be possible to match fonts based on required subsets of Unicode glyphs; the information about glyph coverage is present in each font file, applications should be able to specify the needed range of glyphs and have Xft match fonts based on those requirements.

13 Conclusion

The development of Xft has followed a different model from most of X development. Early versions of the library were released and integrated into other projects; other X projects have waited until they were essentially stable before being reviewed by the community at large.

As use of the library has grown, some major improvements have been added without significantly impacting those other projects. Future enhancements may require changes to existing applications, but that should be viewed as a part of this process; much of this library is breaking new ground for the XFree86 community and only through widespread review and use can appropriate architecture be developed.

The design and implementation of Xft has progressed rapidly. Even though Xft was started only one year ago it has already become an essential part of many current X projects.

References

[Pac01] Keith Packard. Design and Implementation of the X Rendering Extension. In *FREENIX Track, 2001 Usenix Annual Technical Conference*, Boston, MA, June 2001. USENIX.

[SG92] Robert W. Scheifler and James Gettys. *X Window System*. Digital Press, third edition, 1992.

[TT00] David Turner and The FreeType Development Team. The design of FreeType 2, 2000. http://www.freetype.org/freetype2/docs/design/.