Font server implementation overview

Dave Lemke

Network Computing Devices, Inc.

1. Introduction

The font server uses the same client/server model as X. The basic structure is that of the X Consortium X11R5 X server, and those who know that code should find the *os* and *difs* (device independent font server) layers familiar.

```
          +-----------------+
   +-----|      difs       |------+
   |     +-----------------+      |
   |                              |
+----+                    +------------+
| os |                    | renderers  |
+----+                    +------------+
```

Definitions

- Renderer. Code that knows how to take font data in its raw format and convert it to the font server's format.
- Font Path Element (FPE). An instance of a renderer, associated with a specific font source, (ie a directory of PCF bitmaps).

The *difs* layer interprets the requests, and handles the renderer independent work. This includes error checking of requests, and the top level font database. It also contains various utility functionality such as caching and byte swapping.

The *os* layer sets up the communications channel, reads requests and sends the raw data of replies and events. It also handles font server configuration issues, controlled by command line arguments and a configuration file.

The renderer layer contains all font-specific code, and is responsible for rendering a font (which may mean just reading a bitmap from disk, or may include scaling of outline data), computing a fonts properties and header information.

2. Startup

At startup, the font server handles any command line arguments, initializes any OS-specific data, and then sets up the communications. Various internal databases are then initialized (extensions, the font catalogue, etc).

The config file, an ordered list of font sources, cache size hints, default resolutions, and security information, is then read in. Each of these source names could be a directory name, the name of another font server, or some other string that a particular renderer can recognize.

The default font catalogue is then built up by taking each of the font source names and comparing it with the names a renderer recognizes. The one that matches this name will become attached to this source. A renderer will ‘‘understand’’ a name if it can parse the data in that directory, or recognize that it is a valid font server address, or recognizes a special string. Thus a collection of valid font path elements is built up. Each **FPE** has a set of functions to support opening a font and accessing its data.

Font information is accessed via method functions in the **Font.** When a font is first loaded, the header information and properties are loaded/computed. The font also initializes its function pointers to do the proper work. When specific metrics or bitmaps are required, they are access via the font's functions. A disk-based bitmap font will probably want to load all data when first accessed. A scaled font or FS font may want to do more selective caching. In both cases, the renderer can use the utility functions to keep track of this data. Changing values of bitmap formats could result in the font having multiple copies of data in different formats, which the renderer may use the utility functions to manage.

3. Per client processing

Each entity attaching to the server is a client. Each client has its own authorization and resolution information, and its own view of the font database. A font open to one client may not be open to another, though the font server may have it loaded.

After initialization, new clients can attach to the font server and have their requests processed. For each request that is searching for a font **(OpenBitmapFont)** or listing font names **(ListFonts, ListFontsWithXInfo),** the pattern is given to each **FPE.**

OpenBitmapFont will take the supplied name and pass it to each **FPE.** The **FPE** will return one of three things: *Success,* and the font object; *BadFont,* because it doesn't know the font; or *BadFont* and an alias name, when it has an alias for the font. If *Success* is returned, the server goes on to create an ID (or find an existing one) and return a reply. If *BadFont* is returned, it goes on to the next **FPE.** If it reaches the end without finding a font, an error is returned to the client. If an alias is returned, the search resets to the first **FPE** and starts again, using the alias as the new font name. This allows aliases to work across different **FPEs,** without any ordering restrictions.

When each **FPE** receives a font name to open, it searches for the font's existence. If it can't find, or can only find an alias, it returns *BadFont* and any alias. If it finds the font, it checks the authorization and license status of the font to that of the client. If it passes, it then creates a new font object, and reads and/or computes at least the font's header information and properties. (It may also want to produce the bitmaps and extents, but that choice is left to the renderer.)

When a font's information is accessed, the interpreter routine looks up the font ID to find the font object, and then uses the font's access functions to get the data. These functions will return the data in the format expected by the client.

4. Client shutdown

When a client disconnects, all its references to any fonts it still has opened are removed. If no other clients reference these fonts, they may be freed, though the server may choose to cache them.

5. Server reset and cleanup

A server may be reset to flush the caches, re-read the configuration file, and a new list of **FPEs** to be built, via an OS-specific outside action. In UNIX, this will be handled via signals; in VMS it could be handled via an async trap or event flag.

6. Server offloading

In order to deal with numerous clients without major performance degradation, the server must be able to clone itself, or provide the client with a substitute server via the alternate server mechanism. Since both strategies have their uses, both will be supported. For a server that has plenty of host memory or CPU, but insufficient sockets, cloning may be a good choice. For a host with limited memory, assigning an alternate server on a different host may be a good choice. The server will make this decision based on configuration options.

7. Font server data structures

The **Client** handles per-client information and interpreter status.

```
typedef struct _Client {
    int         index;
    pointer     osPrivate;
    int         noClientException;
    int         (**requestVector) ();
    pointer     requestBuffer;
    int         clientGone;
    int         sequence;
    Bool        swapped;
    long        last_request_time;
    void        (*pSwapReplyFunc) ();
    AuthContextPtr auth;
    char       *catalogues;
    int         num_catalogues;
    Mask        eventmask;
    fsResolution *resolutions;
    int         num_resolutions;
}           ClientRec, *ClientPtr;
```

The **Font** contains basic font information, including header information and properties.

```
typedef struct _font       {
         int      refcount;
         fsHeader header;
         fsBitmapFormat    format;
         int      (*get_glyphs)();
         int      (*get_metrics)();
         int      (*get_extents)();
         int      (*get_bitmaps)();
         int      (*unload_font)();
         FontPathElementPtr         fpe;
         int      *client_ids;
         Bool     restricted_font;
}        FontRec *FontPtr;
```

The **ClientFont** is a wrapper on top of **Font,** handling client specific font information.

```
typedef struct _clientfont {
         FontPtr  font;
         int      clientindex;
}        ClientFontRec, *ClientFontRec;
```

The **AuthContext** contains authorization information.

```
typedef  struct _authcontext        {
         char     *authname;
         char     *authdata;
         FSID     acid;
}        AuthContextRec *AuthContextPtr;
```

8. Font Path Element functions

These functions are associated with each renderer, and handle all aspects of font access. Font data access is controlled via another set of functions described later. These functions are intended to support the R5 X server as well as the font server. As a result, some design decisions were made to support both models. When the *difs* layer needs to access a font, it uses these functions.

```
typedef unsigned long      Mask;

typedef unsigned char      *pointer;

typedef struct _FontPathElement {
    int         name_length;
    char       *name;
    int         type;
    int         refcount;
    pointer     private;
}           FontPathElementRec, *FontPathElementPtr;
```

The FPE's reference count is incremented when it is added to the current list of FPEs and when it opens a font. It is decremented when it is no longer in the current list and when it closes a font. All reference changes are handled by the *difs* layer. The count is required to support font catalogue changes that may occur while the fontserver has fonts open, and keeps FPEs from being lost.

```
typedef struct FontNames {
    int  nnames;
    int  size;
    int  *length;
    char **names;
}            FontNamesRec, *FontNamesPtr;

typedef struct {
         Bool     (*name_check)();
         int      (*init_fpe)();
         int      (*reset_fpe)();
         int      (*free_fpe)();
         int      (*open_font)();
         int      (*close_font)();
         int      (*list_fonts)();
         int      (*start_list_fonts_with_info)();
         int      (*list_next_font_with_info)();
         int      (*wakeup_fpe)();
         int      (*client_died);
         FontNamesPtr      renderer_names;
} FPEFunctions;

int      init_fpe_type(Bool (name_func)(),
                  int (init_func)(), int (free_func)(), int (reset_func),
                  int (open_func)(), int (close_func)(),
                  int (list_func)(),
                  int (start_lfwi_func)(), int (next_lfwi_func)(),
                  int (wakeup_func)(),
                  int (client_died_func)()
                  )
```

This is called by the renderer when it is initialized at the beginning of time, and sets up an FPEFunctions entry for the renderer.

The **FPEFunctions** have the following parameters:

```
Bool     name_check(char *name);
```

If *name* is something the renderer recognizes as a valid font source name, it return True, otherwise False. ie, if *name* is a directory name, or is prefixed by the renderer's prefix, and the directory contains font data the renderer can interpret, it would return True.

```
int      init_fpe(FontPathElementPtr fpe);
```

Does any initialization work for the renderer. The name in *fpe* will be one whose prefix matches the list returned when the renderer was initialized.

```
int      reset_fpe(FontPathElementPtr fpe);
```

Tells *fpe* to reset any internal state about what fonts it has available. This will typically be called because the font server's **FPE** search list has been changed. The *fpe* should reset any cached state of available fonts (ie, re-read this*fonts.dir)*when

```
int      free_fpe(FontPathElementPtr fpe);
```

Frees any renderer-specific data and closes any files or sockets.

```
int      open_font(pointer client, FontPathElementPtr fpe, Mask flags,
                 char *fontname, int namelength,
                 fsBitmapFormat format_hint, fsBitmapFormatMask format_mask,
                 XID fontid, FontPtr *ppfont, char **alias);
```

Opens the font. The bits marked by the*format_mask*in *format_hint* are used where applicable. The resulting FontPtr is returned in *ppfont.* The *client* is optional state information for use with blocking renderers. If the *fontname*
resolves to an alias, it is returned in *alias* with a *FontNameAlias* error. This tells the calling code to start searching again, using *alias* as the font name. The renderer is expected to fill in any information specified by the *flags.*

Possible flags values are:

```
#define FontLoadInfo     0x0001              /* font header info */
#define FontLoadProps    0x0002              /* font properties */
#define FontLoadMetrics  0x0004              /* font extents */
#define FontLoadBitmaps  0x0008              /* glyph bitmaps */
#define FontLoadAll      0x000f
#define FontOpenSync     0x0010              /* force synchronous loading */
```

Once a font has been opened, the server may place it and the pattern it matched into a name cache, to avoid lengthy searching if the font is reopened. If the renderer does not wish the font to be in this cache (for licensing reasons), it should set the font's *restricted_access* flag.

```
int      close_font(FontPtr pfont);
```

Frees up all the data associated with the font.

```
int      list_fonts(pointer client, FontPathElementPtr fpe,
                 char *pattern, int pattern_length, int maxnames,
                 FontNamesPtr *paths);
```

Returns in *paths* up to *maxnames* font names the fpe recognizes as matching the given pattern.

```
int      start_list_fonts_with_info(pointer client,
                 FontPathElementPtr fpe, char *pattern, int pattern_length,
                 int maxnames, pointer fpe_data);
```

Initiates a **ListFontsWithXInfo.** Typically, a disk-based renderer will do the equivalent of ListFonts to gather all the font names matching the pattern. A font server renderer will send the request. *fpe_data* provides a handle for any FPE-private data that needs to be passed in later via **list_next_font_with_info(),** eg, the list of font names for a disk-based renderer.

```
int      list_next_font_with_info(pointer client, FontPathElementPtr fpe,
                  char **name, int *namelen, FontInfoPtr &pinfo,
                  int &num_fonts, pointer fpe_data);
```

Returns the next font's information. The renderer should keep any state it requires in the *fpe_data* field. *num_fonts* contains the number of replies remaining.

These two routines are split for because of the way both disk-based renderers and font server renderers handle this request. The first function initiates the action, the second is used to gather the results. For a disk-based renderer, a list of font names matching the pattern is first built up when **start_list_fonts_with_info()** is called, and the results are gathered with each call to **list_next_font_with_info.** In a font server renderer, the first function sends the **ListFontsWithXInfo** request, and the second processes the replies.

```
int      wakeup_fpe(FontPathElementPtr fpe, unsigned long *mask)
```

Optional function which can be used for blocking renderers. Typical usage is for a font server renderer, where it is called when a reply is received, allowing the data to be read and the client to be signaled and unblocked.

```
int      client_died(pointer client, FontPathElementPtr fpe)
```

This function is called when a client dies in the middle of a blocked request, allowing the renderer to clean up.

9. Font specific functions

These functions are contained in each **Font.** For many renderers, every font will use the same functions, but some renderers may wish to use different interfaces for different fonts.

```
typedef struct {
         INT16    left B16,
                  right B16;
         INT16    width B16;
         INT16    ascent B16,
                  descent B16;
         CARD16   attributes B16;
}        fsCharInfo;

typedef struct {
    CARD8         low,
                  high;
}           fsChar2b;

typedef struct {
    fsChar2b      min_char,
                  max_char;
}           fsRange;

int      get_extents(pointer client,
                  FontPtr pfont, Mask flags, int num_ranges, fsRange *ranges,
                  int *num_extents, fsCharInfo **extents);
```

Possible flags:

```
LoadAll          /* ignore the ranges and get everything */
FinishRange      /* magic for range completion as specified by protocol */
```

Builds up the requested array of extents. The extent data (which the renderer allocates) is returned, as well as the number of extents. *closure* contains any blocking state information.

```
int      get_bitmaps(pointer client,
                 FontPtr pfont, fsBitmapFormat format, Mask flags,
                 int num_ranges, fsRange *ranges,
                 unsigned long *size, unsigned long *num_glyphs,
                 unsigned long **offsets, pointer *glyph_data);
```

Possible flags:

```
LoadAll
FinishRange      /* magic for range completion as specified by protocol */
```

Builds up the requested array of bitmaps. The glyph and offset data (which the renderer allocates) is returned, as well as the number of glyphs. The *closure* contains any blocking state information. This function will build up the bitmap data in the format specified by *format* so that the interpreter can return it without any additional modification. This should minimize data massaging, since outline renderers will hopefully be able to produce the bitmaps in the proper format.

```
void     unload_font(FontPtr pfont)
```

The render will free any allocated data. Note that the **FPE** function **close_font()** will also be called, and should handle any **FPE** data allocated for the font.

```
int      get_glyphs()
int      get_metrics()
```

These two functions are used by the X server for loading glyphs and metrics. They expect the results in a considerably different form. The *get_bitmaps()* and *get_extents()* routines both allow for better cache control by the renderer.

10. Font directories and aliases

Existing bitmap renderers already have their own concept of font organization. In the X sample server, the files **fonts.dir** and **fonts.alias** are used to list the known fonts. **fonts.dir** maps file names to font names, while **fonts.alias** maps font names to other font names.

These concepts will also be needed by other forms of fonts which the sample X server does not currently use, but the font server will, like Bitstream outlines.

11. Handling scalable fonts

For those renderers that support scalable fonts, several issues must be addressed:

- Name Parsing. An XLFD name must be parsed to determine the requested resolutions and/or sizes.
- Property scaling. Many of the standard font properties have values that depend on scaling (eg, *RESOLUTION_X. POINT_SIZE)*
- Default values. If resolution information is wildcarded, the proper default resolution should be supplied.

Name Parsing

The font name pattern supplied to **OpenBitmapFont** or **ListFonts** may require some parsing to be recognized as a scalable font known to the renderer. The **PIXEL_SIZE, POINT_SIZE, RESOLUTION_X, RESOLUTION_Y** and **AVERAGE_WIDTH** all need to determined from the font name pattern. The master font must then be found, and scaled appropriately. Any unspecified values that cannot be determined should be replaced by the proper defaults. For size fields, this is whatever the configuration specifies. For resolution fields, these should be taken from the client's resolution list, if set, or from the server's configuration.

Property scaling

Part of scaling a font is scaling its properties. Many scalable fonts will have a very large number of scalable properties. One way to deal with these is for the ‘‘master’’ outline to keep track of the property names, and supply new values for each instance of the font. If the property names are stored as Atoms, memory usage is kept to a minimum.

Using defaults

Using default values as substitutions for missing values was covered above. These defaults will also be useful in handling **ListFonts** requests. Returning a scalable font with an instance using the default values will provide the most user-friendly environment.

12. Access control

The font server will also support large grain security. It will have both a limit of the number of users, and on the hosts which it will support.

Limiting the number of users is as much a server loading issue as a security issue. The limitation will be typically be set via configuration options or OS limitations. To change it, use:

```
void     AccessSetConnectionLimit(int limit)
```

A *limit* of 0 will set it to a compiled constant based on OS resources (eg, number of file descriptors).

Client-host based access control can be used to supplement licensing, and support font server load balancing by restricting access. As with licensing, this is OS-specific code. To manipulate these functions, use:

```
typedef struct _host_address {
         int      type;
         pointer  address;
         struct _host_address *next;
} HostAddress;

typedef HostAddress        *HostList;

int      AddHost(HostList list, HostAddress *address)
int      RemoveHost(HostList list, HostAddress *address)
Bool     ValidHost(HostList list, HostAddress *address)
```

AddHost() adds a host to the *list.* **RemoveHost()** removes it, and **ValidHost()** checks to see if its on the *list.* In all functions, the *address* has will ignore any value in the *next* field.

Network addresses are used here to avoid issues with host name aliases. The caller fills in the desired type, and an address of that form is returned. This is highly OS-specific, but values for the *type* and *address* fields could include:

```
#define  HOST_AF_INET      1
struct in_addr    *address;

#define  HOST_AF_DECnet    2
struct   dn_addr  *address;
```

The server will use a global host list, but having the list as an argument will allow licensing schemes to have their own host lists.

13. Licensing

Licensing is a tricky issue, which each renderer will support in a different way. The sample font server will attempt to provide some guidelines, and present a possible implementation of some simple licensing schemes.

Host Address licensing

This is simplistic licensing based on the client's host. With this form of licensing, a font may be accessible to some host but not others. To get the current client's host, the following is used:

```
void     GetHostAddress(HostAddress *address);
```

A renderer can also use the host access functions to keep a list of the licensed hosts, and **ValidHost()** to check a client.

Simultaneous use license

This licensing allows for a limited number of copies of the font to be open at once. Since this should be a simple per-font counter, no support should be required outside of the renderer.

14. DIFS contents

This contains the protocol dispatcher, interpreter and reply encoding routines.

The interpreter is table driven off the request code. The dispatcher gets a request from the os layer from **WaitForSomething(),** and uses the request code to determine which function to call. eg, a *CloseFont* request would call **ProcCloseFont().**

Each request's routine handles any applicable error checking, and then does as much work as it can. For font related requests, this means converting the request to the proper arguments for the renderers.

If any replies are generated, the reply data is gathered into the bytestream format, and sent via *os* write functions to the client.

If the byte order of the client and server differ, the above is modified by having the dispatcher call an intermediate function which re-orders the request to the proper byte order. Replies go through similar swapping.

Client blocking

To minimize delay caused by font server request, clients can be blocked while they wait for data to be produced. This is primarily intended for **FPEs** using a remote font server, but can be used anywhere where the font server can pause to handle other client requests while data needed to satisfy another is produced (possibly via multiple processes).

```
Bool     ClientSleep(ClientPtr client, Bool (*function)(), pointer closure)
```

Puts a client to 'sleep'. This means the client will no longer be considered while the server is dispatching requests. *function* will be called when the client is signaled, with the *client* and *closure* as its arguments.

```
Bool     ClientSignal(ClientPtr client)
```

This should be called when the client is ready to do more work. At this point, the function given to **ClientSleep()** will be called.

```
void ClientWakeup(ClientPtr client)
```

Puts the client back to its normal state processing requests.

```
Bool ClientIsAsleep(ClientPtr client)
```

Can be used to check if a client is asleep. This is useful for handling client termination, so that any requests the client is waiting upon can be properply cleaned up.

Sample Usage

For handling a font server renderer request for **OpenBitmapFont** the renderer will send the request to the remote font server, and the call **ClientSleep().** The font server will then continue processing requests from other clients, while the one making the request is blocked. When the reply returns, the renderer will notice when its **wakeup_fpe()** function is called. At this point the font server renderer will read and process the reply. **ClientSignal()** will be called, and the *closure* function will be called. It will request the data from the renderer, completing the request, and call **ClientWakeup()** to return the client to normal status.

This layer also contains the resource database, which associates fonts with IDs, extension interface functions and the server initialization and reset control.

15. OS contents

This layer contains OS specific routines for configuration, command line parsing, client/server communications, and various OS-dependent utilities such as memory management and error handling.

ReadRequestFromClient() returns a full request to the dispatcher. **WaitForSomething()** is where the server spends its idle time, waiting for any action from a client or processing any work left from a blocked client.

When a client attempts to connect, the server will call

```
int      CheckClientAuthorization(ClientPtr client, AuthPtr client_auth,
                   int *accept, int *index, int *size, char **authdata)
```

to see if the server is set to allow the client to connect. It may use licensing or configuration information to determine if the client can connect.

When then connection is established, the server will use the

```
typedef struct _alt_server {
    char        subset;
    char        namelen;
    char       *name;
}           AlternateServerRec, *AlternateServerPtr;

int ListAlternateServers(AlternateServerPtr *servers)
```

to return any alternate server information it may have.

When the client limit is reached, the font server may attempt to copy itself, by calling

```
int CloneMyself()
```

This function will (if the configuartion options allow) start a new font server process. This is done in such a way that no pending connections should be lost, and that the original server will accept no new connections. Once the original server has no more clients, it will exit.

Catalogue manipulation

Catalogues are configuration dependent, and hence sent by OS-dependent methods. In order for the *difs* layer to get them, it uses

```
int      ListCatalogues(char *pattern, int pattern_length,
         int maxnames, char **catalogues, int *len)
```

which returns the list of all catalogues it supports which match the pattern. This function will be used by the catalogue manipulation requests, as well as by renderers when they give their **ListFonts** results.

```
int ValidateCatalogues(int number, char *catalogues)
```

Can be used to validate a list of catalogues, returning True if the list is acceptable.

16. Utility functions

Client data functions

These provide access to the current client's resolution and authorization data. This form of interface is supplied rather than passing it to all renderers in the **FPE** functions because the data may be complex and/or uninteresting to all renderers.

```
AuthContextPtr    GetClientAuthorization()
```

Returns the authorization data for the current client.

```
fsResolution      *GetClientResolutions(int   *num_resolutions)
```

Returns the list of resolutions that the current client has set.

Caching functions

These are functions that simplify caching of renderer data. These are for use by renderers that take significant resources to produce data. The data must be re-creatable -- the cache is not meant for general storage. The data may also be moved by the cache, so it should only be accessed by CacheID.

```
typedef void (*CacheFree)();
typedef unsigned long      CacheID;
typedef unsigned long      Cache;
```

```
Cache CacheInit(int renderer_id)
```

Initializes a cache object for the renderer. the returned ID should be passed to **CacheStoreMemory()** when adding an object to the cache.

```
void CacheStats(Cache cid, unsigned long *num_entries,
        unsigned long *max_storage, unsigned long *current_storage,
        unsigned long *num_lookups, unsigned long *hit_ratio)
```

Returns statistics on the cache. Useful if the renderer wants some hints about whether to place an object in the cache. If the cache is nearly full, and the priority low, it may want to take different action.

```
CacheID  CacheStoreMemory(Cache cacheid, pointer data, unsigned long size,
                          CacheFree free_func)
```

The renderer hands the cache some chunk of contiguous memory, which the cache timestamps and stores. When it needs to remove them, it calls the *free_func,* which must take responsibility for properly freeing the data. *size* is primarily a hint to the cache, so that cache limits can be properly calculated. A return value of zero means the store failed, probably because the given size was over the cache limit. If the given data is too large for the current cache, it will attempt to free old data to make room. The returned ID is a unique value that refers both to the object and the cache in which it was placed.

```
pointer CacheFetchMemory(CacheID cid, Bool update)
```

Returns the memory attached to the id. If *update* is set, the timestamp is updated. (some accesses may wish to be 'silent', which allows some control over the freeing scheduling.) If the cid is invalid, *NULL* is returned.

```
int      CacheFreeMemory(CacheID cid, Bool notify)
```

Allows the cache to flush the data. If *notify* is set, the CacheFree function passed in when the data was cached will also be called.

```
void     MemoryFreed(CacheID cid, pointer data, int reason)
```

Callback function from the cache to the renderer notifying it that its data has been flushed. This function then has the responsibility to free that data. *reason* may be one of:

```
CacheReset        /* all cache freed because of server reset */
CacheEntryFreed   /* explicit request via free_memory() */
CacheEntryOld     /* cache hit limit, and memory being freed because its old */
```

and is supplied so that the renderer may choose how to deal with the free request. (It will probably be ignored by most, but some may want to keep the memory around by bypassing the cache, or re-inserting it.) Note that the cache will consider the data gone, so it **must** be re-inserted to keep it alive.

```
void     CacheSimpleFree(CacheID cid, pointer data, int reason)
```

Just calls **free()** on the data. Simple CacheFree defined here to prevent it being redefined in each renderer.

Typical usage of the cache is for the renderer to store a CacheID rather than a pointer to the cacheable data. The renderer is responsible for both allocating and freeing the data, as well as keeping track of just what it is. When the renderer needs the cached data, it will request it from the cache. If it fails, it must rebuild it.

A possible configuration parameter is the size of the cache. when the cache is filled (with the calculation based on the given size), it sweeps the cache and frees old data. The amount of memory actually freed may wish to be tunable: some systems may want to keep the cache as full as possible, others may want to free some percentage such that sweeps occur less frequently.

Cache statistics may want to be available for administrators. They could be dumped to a file when a signal is received. (SNMP seems like a perfect match, but apparently the technology isn't there yet.

Cached data could also be compressed, if the memory/CPU tradeoffs make it worthwhile.

ISSUE: Is a time-based freeing schedule sufficient? Should priorities or size also be taken into account? [No. Anything that the renderer thinks should have a higher priority should probably not be placed into the cache.]

Byte swapping

Functions for swapping a 4-byte quantity, a 2-byte quantity and inverting a byte.

```
void     BitOrderInvert(pointer buffer, unsigned long num_bytes)
void     TwoByteSwap(pointer buffer, unsigned long num_shorts)
void     FourByteSwap(pointer buffer, unsigned long num_longs)
```

Bitmap padding

Functions taking a desired extents and a bitmap that will return the bitmap properly padded.

```
int      RepadBitmap(pointer src, pointer dst, fsFormat src_format,
                   fsFormat dst_format, int width, int height)
```

Takes a bitmap in *src_format* and converts it to one in *dst_format.*

Atoms

Existing bitmap-based renderers use atoms to store strings for property information. Rather than duplicate this code in each renderer, it lives in the *util* directory.

Atoms will be especially useful for property information, to prevent many copies of the same strings from being saved. Using atoms for comparison when modifying properties after scaling is also more efficient. Since *atoms* will will exist until the server is reset, they may want to be used sparingly for property values to avoid extraneous string data.

```
typedef unsigned long      Atom;

Atom     MakeAtom(char *string, unsigned int length, Bool create)
```

Returns the atom associated with *string.* If *create* is true, a new atom will be created.

```
char     *NameForAtom(Atom atom)
```

Returns the string associated with *atom.*

17. Server request details

This section describes in-depth the action of each protocol request. In all cases, the request is first error checked for simple length or value errors, with the server immediately returning an error if one is encountered.

17.1. Connection

When a new client attempts to connect, the server first checks its initial authorization information to see if the server is willing to talk to it. This will be handled in some OS-specific form using

CheckClientAuthorization(). If it passes this test, and the server has sufficient to resources to talk to it, the server sends accepts the connection and returns its connection block. If the connection fails, the server returns the proper status and a list of any alternate servers it may know of (gathered from **ListAlternate-Servers().)**

17.2. ListExtension

Returns the list of extensions the server knows about. Any extensions will be initialized when the server is first started.

17.3. QueryExtension

Returns the information about the requested extension, which was set when the extension was initialized.

17.4. ListCatalogues

Returns the catalogues the server recognizes (the results of **ListCatalogues().)**

17.5. SetCatalogues

Sets the requesting client's catalogues after verifying them with the supported catalogues.

17.6. GetCatalogues

Returns the requesting client's catalogues.

17.7. CreateAC

Creates a new authorization context and fills it in. The list of authorization protocols is then checked by the server with **CheckClientAuthorization().** If any are accepted, the **AC** is placed in the resource database and *Success* is returned with the name of the accepted protocol. If more than one is accepted, *Continue* is returned with each of the accepted protocols, until the last one which has status *Success* Otherwise *Denied* is returned.

17.8. FreeAC

Looks up the **AC** in the resource database, and frees it if it finds it. Otherwise an *Access* error is returned.

17.9. SetAuthorization

Looks up the **AC** in the resource database, and set the client's AuthContextPtr to its value if it is found. Otherwise it sends an *Access* error.

17.10. SetResolution

Sets the requesting client's resolution list to the supplied list.

17.11. GetResolution

Returns the requesting client's list of resolutions.

17.12. ListFonts

Iterates over each open FPE, calling the FPE's **list_fonts()** routine passing it the pattern. When all FPE's have been processed, the list that has been built up is returned. Note that the same **FontNamesPtr** is sent to each FPE in turn, so that one list is built up. An FPE may restrict the fonts it returns based on the client's catalogue.

17.13. ListFontsWithXInfo

Iterates over each FPE, calling its **start_list_fonts_with_info()** function to prime the FPE's renderer. It then calls the FPE's **list_next_font_with_info(),** sending each font's data to the client until no more fonts remain. When all FPEs have been processed, the final reply with a zero-length name is then sent to mark the end of the replies. An FPE may restrict the fonts it returns based on the client's catalogue. Note: an issue exists with font aliases which may require this to change, since an FPE may contain an alias pointing to another FPE, and cannot therefore return the font's info.

17.14. OpenBitmapFont

The pattern is first searched for in the font server's name cache. If it doesn't find it, the server iterates over each FPE, calling its **open_font** function with the supplied pattern. This will return one of the following values:

- an **Access** error, which means the renderer has the font but the client does not have access to it because of some form of licensing restriction
- a **Font** error and a NULL *alias* parameter, which will cause the next FPE to be tried
- a **Font** error but a non-NULL *alias,* which will cause the search to start over with the first FPE using *alias* as the new font pattern
- **Success,** in which case a valid font has been found.

If the end of the FPE list is reached without having found the font, an error is returned to the client. If an **Access** error was encountered, it is returned, otherwise a **Font** error is returned. If a valid font is found, its reference count will be incremented and it will be checked to see if the client has already opened it before. If so, the previous ID will be returned. Otherwise the font will be placed in the resource database.

The renderer will fill in the font's header and property information, and may also choose to load or create the font's metrics or glyphs. If the glyphs are built, they will use any supplied *format hint*.

Whenever a new font is successfuly opened, the font and its name pattern will be placed in a name cache. This cache exists to minimize the amount of work spent searching for a font. It will be flushed when the font catalogue is modified. Client's with private font catalogues will require private name caches.

17.15. QueryXInfo

The *fontid* is looked up in the resource database, and the font's header and property info is returned.

17.16. QueryXExtents8 QueryXExtents16

The *fontid* is looked up in the resource database. The supplied list of characters (interpreted according to request type) is then translated into a list of ranges. The font's **get_extents()** function is then called. It builds the requested list of extents, and returns them along with the number of extents. The results are properly swapped and sent to the client.

17.17. QueryXBitmaps8 QueryXBitmaps16

The *fontid* is looked up in the resource database. The supplied list of characters (interpreted according to request type) is then translated into a list of ranges. The font's **get_bitmaps()** function is called, and the renderer will build up the requested bitmaps, using the specified *format,* and returns the bitmaps, the number of glyphs and the offsets. The offsets are properly swapped and the offsets and bitmaps are sent to the clients.

17.18. CloseFont

The font's reference count is decremented. If this was the last reference, the font's **unload_font()** function is called to free the renderer's data, and the font's FPE **close_font()** function is called to free up any FPE specific data.

18. Configuration

The configuration mechanism is a simple keyword-value pair, separated by an '='.

Configuration types:

cardinal	non-negative number
boolean	"[Yy]es", "[Yy]" "on", "1", "[Nn]o", "[Nn]", "off", "0"
resolution	*cardinal,cardinal*
list of foo	1 or more of foo, separated by commas

Here is an incomplete list of the supported keywords:

#	in the first column, a comment character
catalogue (list of string)	Ordered list of font path element names.
alternate-servers (list of string)	List of alternate servers for this FS.
client-limit (cardinal)	Number of clients this FS will support before refusing service.
clone-self (boolean)	Whether this FS should attempt to clone itself or use delegates when it reachs the client-limit.
default-point-size (cardinal)	The default pointsize (in decipoints) for fonts that don't specify.
default-resolutions (list of resolutions)	Resolutions the server supports by default. This information may be used as a hint for pre-rendering.
error-file (string)	Filename of the error file. All warnings and errors will be logged here.
port (cardinal)	The TCP port on which the server will listen for connections.
use-syslog (boolean)	Whether syslog(3) is to be used for errors.

Each renderer may also want private configuration options. The names should be prefixed by the renderer name, ie *pcf-, atm-*.

Examples:

```
# allow a ~a megabyte of memory to be reserved for cache data
cache-size = 1000000
```

catalogue = pcf:/usr/lib/X11/fonts/misc,speedo:/usr/lib/fonts/speedo