OpenGL® Graphics with the X Window System® (Version 1.2)

Phil Karlton
Revised by: Paula Womack

OpenGL® Graphics with the X Window System®

Phil Karlton
Revised by: Paula Womack

1 Overview

This document describes GLX, the OpenGL extension to the X Window System. It refers to concepts discussed in the OpenGL specification, and may be viewed as an X specific appendix to that document. Parts of the document assume some acquaintance with both the OpenGL and X.

In the X Window System, OpenGL rendering is made available as an extension to X in the formal X sense: connection and authentication are accomplished with the normal X mechanisms. As with other X extensions, there is a defined network protocol for the OpenGL rendering commands encapsulated within the X byte stream.

Since performance is critical in 3D rendering, there is a way for OpenGL rendering to bypass the data encoding step, the data copying, and interpretation of that data by the X server. This *direct rendering* is possible only when a process has direct access to the graphics pipeline. Allowing for parallel rendering has affected the design of the GLX interface. This has resulted in an added burden on the client to explicitly prevent parallel execution when that is inappropriate.

X and the OpenGL have different conventions for naming entry points and macros. The GLX extension adopts those of the OpenGL.

2 GLX Operation

2.1 Rendering Contexts and Drawing Surfaces

The OpenGL specification is intentionally vague on how a rendering context (an abstract OpenGL state machine) is created. One of the purposes of GLX is to provide a means to create an OpenGL context and associate it with a drawing surface.

In X, a rendering surface is called a Drawable. Windows, one type of Drawable, are associated with a Visual.* The X protocol allows for a single VisualID to be instantiated at multiple depths. The GLX bindings allow only one depth for an OpenGL renderer for any given VisualID. In GLX the definition of Visual has been extended to include the types, quantities and sizes of the ancillary buffers (depth, accumulation, auxiliary, and stencil). Double buffering capability is also fixed by the Visual.† The ancillary buffers have no meaning within the core X environment. The set of extended Visuals is fixed at server startup time. One result is that a server can export multiple Visuals that differ only in the extended attributes.

The other type of X Drawable is a Pixmap, a drawing surface that is maintained off screen. The GLX equivalent to an X Pixmap is a GLXPixmap. A GLXPixmap is created using the Visual along with its extended attributes. The Visual is used to define the type and size of the Ancillary buffers associated with the Pixmap. The Pixmap is used as the front-left color buffer. A GLXDrawable is the union {Window, GLXPixmap}.

Ancillary buffers are associated with a GLXDrawable, not with a rendering context. If several OpenGL renderers are all writing to the same window, they will share those buffers. Rendering operations to one window never affect the unobscured pixels of another window, or of the corresponding pixels of ancillary buffers of that window. If an Expose event is received by the client, the values in the ancillary buffers and in the back buffers for regions corresponding to the exposed region become undefined.

A rendering context can be used with multiple GLXDrawables as long as those Drawables are *similar*. Similar means that the rendering contexts and GLXDrawables are created with the same XVisualInfo.

An application can use any rendering context (subject to the restrictions discussed in the section on address spaces) to render into any similar GLXDrawable. An implication is that multiple applications can render into the same window, each using a different rendering context.

*The association is with a {Visual, screen, depth} triple. An XVisualInfo is used by GLX functions since it can be interpreted unambiguosly.

†Any rendering system is free to use the ancillary buffers as long as it uses them in a manner consistent with the use by the OpenGL.

2.2 Using Rendering Contexts

OpenGL defines both client state and server state. Thus a rendering context consists of two parts: one to hold the client state and one to hold the server state. The client is responsible for creating a rendering context and a drawable; defaults are not supplied.

Each thread can have at most one current rendering context. In addition, a rendering context can be current for only one thread at one time.

Issuing OpenGL commands may cause the X buffer to be flushed. In particular, calling **glFlush()** will flush both the X and OpenGL rendering streams.

Some state is shared between the OpenGL and X. The pixel values in the X frame buffer are shared. The X double buffer extension (DBE) has a definition for which buffer is currently the displayed buffer. This information is shared with GLX. The state of which buffer is displayed tracks in both extensions, independent of which extension initiates a buffer swap.

2.3 Direct Rendering and Address Spaces

One of the basic assumptions of the X protocol is that if a client can name an object, then it can manipulate that object. GLX introduces the notion of an *Address Space*. A GLX object cannot be used outside of the address space in which it exists.

In a classic UNIX environment, each process is in its own address space. In a multi-threaded environment, each of the threads will share a virtual address space which references a common data region.

An OpenGL client that is rendering to a graphics engine directly connected to the executing CPU may avoid passing the tokens through the X server. This generalization is made for performance reasons. The model described here specifically allows for such optimizations, but does not mandate that any implementation support it.

When direct rendering is occurring, the address space of the renderer is that of the direct process; when direct rendering is not being used, the address space of the renderer is that of the X server. The client has the ability to reject the use of direct rendering, but there may be a performance penalty in doing so.

In order to use direct rendering, a client must create a direct rendering context. Both the client context state and the server context state of

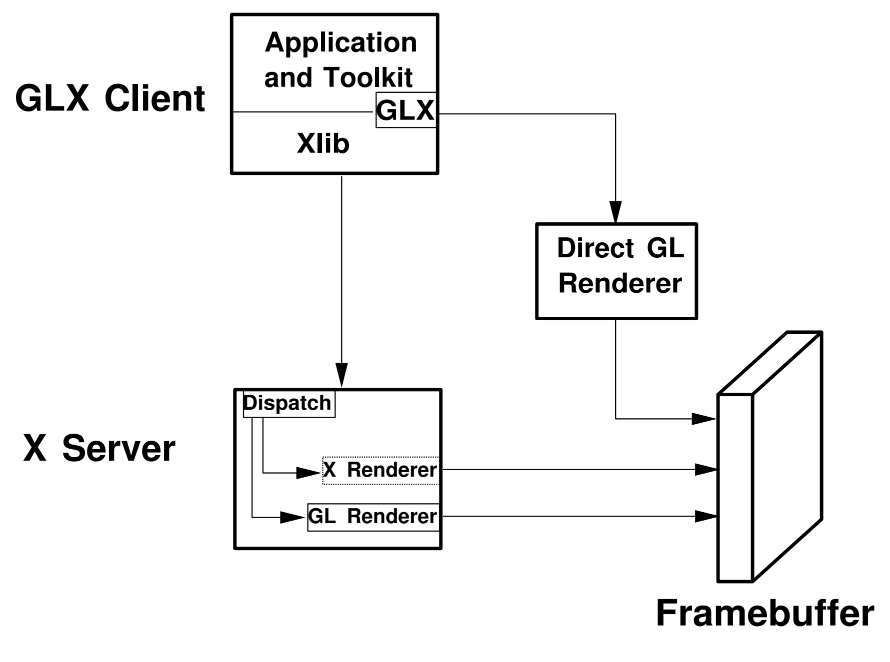

Figure 1. Direct Rendering Block Diagram.

a direct rendering context exist in the client's address space; this state cannot be shared by a client in another process. With indirect rendering contexts, the client context state is kept in the client's address space and the server context state is kept in the address space of the X server. In this case the server context state is stored in an X resource; it has an associated XID and may potentially be used by another client process.

2.4 OpenGL Display Lists

Most OpenGL state is small and easily retrieved using the **glGet*** commands. This is not true of OpenGL display lists, which are used, for example, to encapsulate a model of some physical object. First, there is no mechanism to obtain the contents of a display list from the rendering context. Second, display lists may be large and numerous. It may be desirable for multiple rendering contexts to share display lists rather than replicating that information in each context.

GLX provides for limited sharing of display lists; the lists can be shared only if the server state for the contexts share a single address space. Using this mechanism, a single set of lists can be used, for instance, by a context that supports color index rendering and a context that supports RGBA rendering.

A group of shared display lists exists until the last referencing rendering context is destroyed. All rendering contexts have equal access to using lists or defining new lists. Implementations sharing contexts must handle the case where one rendering context is using a display list when another rendering context destroys that list.

When display lists are shared between OpenGL contexts, the sharing extends only to the display lists themselves and the information about which display list numbers have been allocated. In particular, the value of the base set with **glListBase** is not shared.

In general, OpenGL commands are not atomic. **glEndList** and **glDeleteLists** are exceptions. The list named in a **glNewList** call is not created or superseded until **glEndList** is called. If one rendering context is sharing a display list with another, it will continue to use the existing definition while the second context is in the process of re-defining it.

2.5 Texture Objects

OpenGL texture state can be encapsulated in a named texture object. A texture object is created by binding an unused name to one of the texture targets (TEXTURE_1D or TEXTURE_2D) of a rendering context. When a texture object is bound, OpenGL operations on the target to which it is bound affect the bound texture object, and queries of the target to which it is bound return state from the bound texture object.

Texture objects may be shared by rendering contexts, as long as the server portion of the contexts share the same address space. OpenGL makes no attempt to synchronize access to texture objects. If a texture object is bound to more than one context, then it is up to the programmer to ensure that the contents of the object are not being changed via one context while another context is using the texture object for rendering. The results of changing a texture object while another context is using it are undefined.

A texture object will not be deleted until it is no longer bound to any rendering context.

2.6 Aligning Multiple Drawables

A client can create one window with an overlay `Visual` and a second with a main plane `Visual` and then move them independently or in concert to keep them aligned. This is a major change between the OpenGL and the previous SGI proprietary GL: allocation of overlay planes and main planes for every window is no longer done automatically. To accomplish what was done by a **drawmode/gconfig** pair in previous versions of the SGI proprietary GL, the OpenGL client can use the following paradigm:

- Make the windows which are to share the same screen area children of a single window (that will never be written). Size and position the children to completely occlude their parent. When the window combination must be moved or resized, perform the operation on the parent.

- Make the subwindows have a background of `None` so that the X server will not paint into the shared area when you restack the children.

- Select for device-related events on the parent window, not on the children. Since device-related events with the focus in one of the child windows will be inherited by the parent, input dispatching can be done directly without reference to the child on top.

2.7 Multiple Threads

It is possible to create a version of the client side library that is protected against multiple threads attempting to access the same connection. This is accomplished by having appropriate definitions for **LockDisplay** and **UnlockDisplay**. Since there is some performance penalty for doing the locking, it is implementation-dependent whether a thread safe version, a non-safe version, or both versions of the library are provided. Interrupt routines may not share a connection (and hence a rendering context) with the main thread. An application may be written as a set of co-operating processes.

X has atomicity (between clients) and sequentiality (within a single client) requirements that limit the amount of parallelism achievable when interpreting the command streams. GLX relaxes these requirements. Sequentiality is still guaranteed within a command stream, but not between the X and the OpenGL command streams. It is possible, for example, that an X command issued by a single threaded client after an OpenGL command might be executed before that OpenGL command.

The X specification requires that commands are atomic:

> If a server is implemented with internal concurrency, the overall effect must be as if individual requests are executed to completion in some serial order, and requests from a given connection must be executed in delivery order (that is, the total execution order is a shuffle of the individual streams).

OpenGL commands are not guaranteed to be atomic. Some OpenGL rendering commands might otherwise impair interactive use of the windowing system by the user. For instance calling a deeply nested display list or rendering a large texture mapped polygon on a system with no graphics hardware could prevent a user from popping up a menu soon enough to be usable.

Synchronization is in the hands of the client. It can be maintained with moderate cost with the judicious use of the **glFinish**, **glXWaitGL**, **glXWaitX**, and **XSync** commands. OpenGL and X rendering can be done in parallel as long as the client does not preclude it with explicit synchronization calls. This is true even when the rendering is being done by the X server. Thus, a multi-threaded X server implementation may execute OpenGL rendering commands in parallel with other X requests.

Some performance degradation may be experienced if needless switching between OpenGL and X rendering is done. This may involve a round trip to the server, which can be costly.

3 Functions and Errors

3.1 Errors

Where possible, as in X, when a request terminates with an error, the request has no side effects.

The error codes that may be generated by a request are described with that request. The following table summarizes the GLX-specific error codes that are visible to applications:

`GLXBadContext` A value for a `Context` argument does not name a `Context`.

`GLXBadContextState` An attempt was made to switch to another rendering context while the current context was in `RenderMode` `GL_FEEDBACK` or `GL_SELECT`, or a call to **glXMakeCurrent** was made between a **glBegin** and the corresponding call to **glEnd**.

`GLXBadCurrentWindow` The current `Drawable` of the calling thread is a window that is no longer valid.

`GLXBadDrawable` The `Drawable` argument does not name a `Drawable` configured for OpenGL rendering.

`GLXBadPixmap` The `Pixmap` argument does not name a `Pixmap` that is appropriate for OpenGL rendering.

`GLXUnsupportedPrivateRequest` May be returned in response to either a glXVendorPrivate request or a glXVendorPrivateWithReply request.

The following error codes may be generated by a faulty GLX implementation, but would not normally be visible to clients:

`GLXBadContextTag` A rendering request contains an invalid context tag. (Context tags are used to identify contexts in the protocol.)

`GLXBadRenderRequest` A glXRender request is ill-formed.

`GLXBadLargeRequest` A glXRenderLarge request is ill-formed.

3.2 Functions

GLX functions should not be called between **glBegin** and **glEnd** operations. If a GLX function is called within a **glBegin/glEnd** pair, then the result is undefined; however, no error is reported.

3.2.1 Initialization

To ascertain if the GLX extension is defined for an X server, use

> `Bool` **glXQueryExtension**`(` `Display` **dpy*, `int` **error_base*, `int` **event_base* `) ;`

dpy specifies the connection to the X server. `False` is returned if the extension is not present. *error_base* is used to return the value of the first error code. The constant error codes should be added to this base to get the actual value.

event_base is included for future extension. GLX does not currently define any events.

The GLX definition exists in multiple versions. Use

> `Bool` **glXQueryVersion**`(` `Display` **dpy*, `int` **major*, `int` **minor* `) ;`

to discover which version of GLX is available. Upon success, *major* and *minor* are filled in with the major and minor versions of the extension implementation. If the client and server both have the same major version number then they are compatible and the minor version that is returned is the minimum of the two minor version numbers.

major and *minor* do not return values if they are specified as **NULL**.

glXQueryVersion returns **True** if it succeeds and **False** if it fails. If it fails, *major* and *minor* are not updated.

3.2.2 Configuration Management

The constants shown in Table 1 are passed to **glXGetConfig** and **glXChooseVisual** to specify which attributes are being queried.

Attribute	Type	Notes
GLX_USE_GL	boolean	**True** if OpenGL rendering supported
GLX_BUFFER_SIZE	integer	depth of the color buffer
GLX_LEVEL	integer	frame buffer level
GLX_RGBA	boolean	**True** if RGBA rendering supported
GLX_DOUBLEBUFFER	boolean	**True** if color buffers have front/back pairs
GLX_STEREO	boolean	**True** if color buffers have left/right pairs
GLX_AUX_BUFFERS	integer	number of auxiliary color buffers
GLX_RED_SIZE	integer	number of bits of Red in the framebuffer
GLX_GREEN_SIZE	integer	number of bits of Green in the framebuffer
GLX_BLUE_SIZE	integer	number of bits of Blue in the framebuffer
GLX_ALPHA_SIZE	integer	number of bits in the destination alpha buffer
GLX_DEPTH_SIZE	integer	number of bits in the depth buffer
GLX_STENCIL_SIZE	integer	number of bits in the stencil buffer
GLX_ACCUM_RED_SIZE	integer	number Red bits in the accumulation buffer
GLX_ACCUM_GREEN_SIZE	integer	number Green bits in the accumulation buffer
GLX_ACCUM_BLUE_SIZE	integer	number Blue bits in the accumulation buffer
GLX_ACCUM_ALPHA_SIZE	integer	number Alpha bits in the accumulation buffer

Table 1: Configuration attributes.

`GLX_BUFFER_SIZE` gives the total depth of the color buffer in bits. For **PseudoColor** and **StaticColor** visuals, this is equal to the depth value reported in the core X11 Visual. For **TrueColor** and **DirectColor** visuals, `GLX_BUFFER_SIZE` is the sum of `GLX_RED_SIZE`, `GLX_GREEN_SIZE`, `GLX_BLUE_SIZE`, and `GLX_ALPHA_SIZE`. Note that this value may be larger than the depth value reported in the core X11 visual since it may include alpha planes that may not be reported by X11. Also, for `TrueColor` visuals, the sum of `GLX_RED_SIZE`, `GLX_GREEN_SIZE`, and `GLX_BLUE_SIZE` may be larger than the maximum depth that core X11 can support.

To obtain a description of an OpenGL attribute exported by a `Visual` use

> `int` **glXGetConfig**`(` `Display` **dpy*, `XVisualInfo*` **visual*, `int` *attribute*, `int` **value* `)` ;

glXGetConfig returns through *value* the value of the *attribute* of *visual.*

glXGetConfig returns one of the following error codes if it fails, and Success otherwise:

`GLX_NO_EXTENSION` *dpy* does not support the GLX extension.

`GLX_BAD_SCREEN` screen of *visual* does not correspond to a screen.

`GLX_BAD_ATTRIBUTE` *attribute* is not a valid GLX attribute.

`GLX_BAD_VISUAL` *visual* does not support GLX and an attribute other than `GLX_USE_GL` was specified.

`GLX_BAD_VALUE` parameter invalid

A GLX implementation may export many visuals that support OpenGL. These visuals support either color index or RGBA rendering. Currently RGBA rendering can be supported only by Visuals of type **TrueColor** or **DirectColor** and color index rendering can be supported only by Visuals of type **PseudoColor** or **StaticColor**.

Servers are required to export at least one visual that supports RGBA rendering. At least one of the visuals that supports RGBA rendering must have at least one color buffer, a stencil buffer of at least 1 bit, a depth buffer of at least 12 bits, and an accumulation buffer; alpha bitplanes are optional. The color buffer size for this visual must be as large as that of the deepest **TrueColor**, **DirectColor**, **PseudoColor**,

Attribute	Default	Selection Criteria
GLX_USE_GL	**True**	exact
GLX_BUFFER_SIZE	0	minimum, smallest
GLX_LEVEL	0	exact
GLX_RGBA	**False**	exact
GLX_DOUBLEBUFFER	**False**	exact
GLX_STEREO	**False**	exact
GLX_AUX_BUFFERS	0	minimum, smallest
GLX_RED_SIZE	0	minimum, largest
GLX_GREEN_SIZE	0	minimum, largest
GLX_BLUE_SIZE	0	minimum, largest
GLX_ALPHA_SIZE	0	minimum, largest
GLX_DEPTH_SIZE	0	minimum, largest
GLX_STENCIL_SIZE	0	minimum, smallest
GLX_ACCUM_RED_SIZE	0	minimum, largest
GLX_ACCUM_GREEN_SIZE	0	minimum, largest
GLX_ACCUM_BLUE_SIZE	0	minimum, largest
GLX_ACCUM_ALPHA_SIZE	0	minimum, largest

Table 2: Defaults and selection criteria used by **glXChooseVisual**.

or **StaticColor** visual supported on framebuffer level zero (the main image planes), and it must be available on framebuffer level zero.

If the X server exports a **PseudoColor** or **StaticColor** visual on framebuffer level 0, a visual that supports color index rendering is also required. If color index rendering is supported then one of the visuals that supports color index rendering must have at least one color buffer, a stencil buffer of at least 1 bit, and a depth buffer of at least 12 bits. It also must have as many color bitplanes as the deepest **PseudoColor** or **StaticColor** visual supported on framebuffer level zero, and it must itself be made available on level zero.

glXChooseVisual is used to find a visual that matches the client's specified attributes.

`XVisualInfo*` **glXChooseVisual**(`Display` **dpy*, `int` *screen*, `int` **attrib_list*) ;

glXChooseVisual returns a pointer to an `XVisualInfo` structure describing the visual that best matches the specified attributes. If no matching visual exists, **NULL** is returned.

The attributes are matched in an attribute-specific manner, as shown in Table 2. Some of the attributes, such as `GLX_LEVEL`, must match the specified value exactly; others, such as, `GLX_BUFFER_SIZE` and `GLX_RED_SIZE` must meet or exceed the specified minimum values. In the case of `GLX_BUFFER_SIZE`, preference is given based on how close the visual's attribute value is to the specified value. (Attributes that are matched in this manner have `minimum, smallest` listed as their selection criteria in Table 2.) In the case of `GLX_RED_SIZE`, if the specified value is non-zero, then preference is given to visuals with the largest value for this attribute; otherwise preference is given to visuals with the smallest value. (Attributes that are matched in this manner have `minimum, largest` listed as their selection criteria in Table 2.)

If `GLX_RGBA` is in *attrib_list* then the resulting visual will be `TrueColor` or `DirectColor`. If all other attributes are equivalent, then a `TrueColor` visual will be chosen in preference to a `DirectColor` visual.

If `GLX_RGBA` is not in *attrib_list* then the returned visual will be `PseudoColor` or `StaticColor`. If all other attributes are equivalent then a `PseudoColor` visual will be chosen in preference to a `StaticColor` visual.

If an attribute is not specified in *attrib_list*, then the default value is used. See Table 2 for a list of defaults.

Default specifications are superseded by the attributes included in *attrib_list*. Integer attributes are immediately followed by the corresponding desired value. Boolean attributes appearing in *attrib_list* have an implicit **True** value; such attributes are *never* followed by an explicit **True** or **False** value. The list is terminated with `None`.

To free the data returned, use **XFree**.

NULL is returned if an undefined GLX attribute is encountered.

3.2.3 Off Screen Rendering

To create an off screen rendering area, first create an X `Pixmap` of the depth specified by the desired `Visual`, then call

`GLXPixmap` **glXCreateGLXPixmap**(`Display` **dpy*, `XVisualInfo*` *visual*, `Pixmap` *pixmap*) ;

glXCreateGLXPixmap creates an off screen rendering area and returns its XID. Any GLX rendering context created with respect to *visual* can be used to render into this off screen area.

pixmap is used for the RGB planes of the front-left buffer of the resulting GLX off screen rendering area. The alpha buffer and ancillary buffers specified by *visual* are created without externally visible names. GLX pixmaps may be created with a *visual* that includes back buffers and stereoscopic buffers. However, **glXSwapBuffers** is ignored for these pixmaps.

A direct rendering context might not be able to be made current with a `GLXPixmap`.

If the depth of *pixmap* does not match the depth value reported by core X11 for *visual*, or if *pixmap* was not created with respect to the same screen as *visual*, then a `BadMatch` error is generated. If *visual* is not valid (e.g., if GLX does not support it), then a `BadValue` error is generated. If *pixmap* is not a valid pixmap id, then a `BadPixmap` error is generated. Finally, if the server cannot allocate the new GLX pixmap, a `BadAlloc` error is generated.

A `GLXPixmap` is destroyed by calling

```
void glXDestroyGLXPixmap( Display *dpy,
    GLXPixmap pixmap ) ;
```

This request deletes the association between the resource ID *pixmap* and the GLX pixmap. The storage will be freed when it is not current to any client.

If *pixmap* is not a valid GLX pixmap then a `GLXBadPixmap` error is generated.

3.2.4 Rendering Contexts

To create an OpenGL rendering context call

```
GLXContext glXCreateContext( Display *dpy,
    XVisualInfo* visual, GLXContext share_list, Bool direct
    ) ;
```

glXCreateContext returns **NULL** if it fails. If **glXCreateContext** succeeds, it initializes the rendering context to the default OpenGL state and returns a handle to it. This handle can be used to render to both windows and GLX pixmaps.

If *share_list* is not **NULL**, then all display lists and texture objects except texture objects named 0 will be shared by *share_list* and the newly created rendering context. An arbitrary number of **GLXContexts** can share a single display list and texture object space. All sharing contexts must also share a single address space or a `BadMatch` error is generated.

If *direct* is true, then a direct rendering context will be created if the implementation supports direct rendering and the connection is to an X server that is local. If *direct* is **False**, then a rendering context that renders through the X server is created.

Direct rendering contexts may be a scarce resource in some implementations. If *direct* is true, and if a direct rendering context cannot be created, then **glXCreateContext** will attempt to create an indirect context instead.

glXCreateContext can generate the following GLX extension errors: `GLXBadContext` if *share_list* is neither zero nor a valid GLX rendering context; `BadValue` if *visual* is not a valid X Visual or if GLX does not support it; `BadMatch` if *share_list* defines an address space that cannot be shared with the newly created context or if *share_list* was created on a different screen than the one referenced by *visual*; `BadAlloc` if the server does not have enough resources to allocate the new context.

To determine if an OpenGL rendering context is direct call

`Bool` **glXIsDirect**`(  Display` **dpy*`, GLXContext` *ctx* `) ;`

glXIsDirect returns **True** if *ctx* is a direct rendering context, **False** otherwise. If *ctx* is not a valid GLX rendering context, a `GLXBadContext` error is generated.

An OpenGL rendering context is destroyed by calling

`void` **glXDestroyContext**`(  Display` **dpy*`, GLXContext` *ctx* `) ;`

If *ctx* is still current to any thread, *ctx* is not destroyed until it is no longer current. In any event, the associated XID will be destroyed and *ctx* cannot subsequently be made current to any thread.

glXDestroyContext will generate a `GLXBadContext` error if *ctx* is not a valid rendering context.

To copy OpenGL rendering state from one context to another, use

`void` **glXCopyContext**`(  Display` **dpy*`, GLXContext` *source*`, GLXContext` *dest*`, unsigned long` *mask* `) ;`

glXCopyContext copies selected groups of state variables from *source* to *dest*. *mask* indicates which groups of state variables are to be copied; it contains the bitwise OR of the symbolic names for the attribute groups. The symbolic names are the same as those used by **glPushAttrib**, described in the OpenGL Specification. Also, the order in which the attributes are copied to *dest* as a result of the **glXCopyContext** operation is the same as the order in which they are popped off of the stack when **glPopAttrib** is called. The single symbolic constant `GL_ALL_ATTRIB_BITS` can be used to copy the maximum possible portion of the rendering state. It is not an error to specify *mask* bits that are undefined.

If *source* and *dest* do not share an address space or were not created on the same screen, a `BadMatch` error is generated. (*source* and *dest* may be based on different X visuals and still share an address space; **glXCopyContext** will work correctly in such cases.) If the destination context is current for some thread then a `BadAccess` error is generated. If the source context is the same as the current context of the calling thread, and the current drawable of the calling thread is a window that is no longer valid, a `GLXBadCurrentWindow` is generated. Finally, if either *source* or *dest* is not a valid GLX rendering context, a `GLXBadContext` error is generated.

glXCopyContext performs an implicit **glFlush()** if *source* is the current context for the calling thread.

Only one rendering context may be in use, or *current*, for a particular thread at a given time. The minimum number of current rendering contexts that must be supported by a GLX implementation is one. (Supporting a larger number of current rendering contexts is essential for general-purpose systems, but may not be necessary for turnkey applications.)

To make a context current, call

> `Bool` **glXMakeCurrent**`(  Display` **dpy*`, GLXDrawable` *drawable*`, GLXContext` *ctx* `) ;`

If the calling thread already has a current rendering context, then that context is flushed and marked as no longer current. *ctx* is made the current context for the calling thread.

If the *drawable* and *ctx* are not similar, a `BadMatch` error is generated. If *ctx* is current to some other thread, then **glXMakeCurrent** will generate a `BadAccess` error. `GLXBadContextState` is generated if there is a current rendering context and its render mode is either **GL_FEEDBACK** or **GL_SELECT**. `GLXBadContextState` will also be generated if **glXMakeCurrent** is called between a **glBegin** and its corresponding **glEnd**. If *ctx* is not a valid GLX rendering context, `GLXBadContext` is generated. If *drawable* is not a valid GLX drawable, a `GLXBadDrawable` error is generated. If the previous context of the calling thread has unflushed commands, and the previous drawable is a window that is no longer valid, `GLXBadCurrentWindow` is generated. Finally, note that the ancillary buffers for *drawable* need not be allocated until they are needed. A `BadAlloc` error will be generated if the server does not have enough resources to allocate the buffers.

If *drawable* is destroyed after **glXMakeCurrent** is called then subsequent rendering commands will behave as if *drawable* is bound to the NULL clip. The commands will be processed and the context state will be updated, but no output will appear on the display.

To release the current context without assigning a new one, use `NULL` for *ctx* and `None` for *drawable*. If *ctx* is `NULL` and *drawable* is not `None`, or if *drawable* is `None` and *ctx* is not `NULL`, then a `BadMatch` error will be generated.

The first time *ctx* is made current to a `GLXDrawable`, its initial viewport is set. That viewport must be reset by the client when *ctx* is subsequently made current.

Note that when multiple threads are using their current contexts to render to the same drawable, OpenGL does not guarantee atomicity of fragment update operations. In particular, programmers may not assume that depth-buffering will automatically work correctly; there is a race condition between threads that read and update the depth buffer. Clients are responsible for avoiding this condition. They may use vendor-specific extensions or they may arrange for separate threads to draw in disjoint regions of the framebuffer, for example.

glXGetCurrentContext returns the current context.

`GLXContext` **glXGetCurrentContext**`( void ) ;`

If there is no current context, **NULL** is returned.

glXGetCurrentDrawable returns the XID of the current drawable.

`GLXDrawable` **glXGetCurrentDrawable**`( void ) ;`

If there is no current drawable, `None` is returned.

To get the display associated with the current context and drawable, call

`Display*` **glXGetCurrentDisplay**`( void ) ;`

If there is no current context, **NULL** is returned. This routine is available only if the GLX version is 1.2 or later.

glXGet* calls retrieve client-side state and do not force a round trip to the X server. Unlike most X calls (including the **glXQuery*** calls) that return a value, these calls do not flush any pending requests.

3.2.5 Synchronization Primitives

To prevent X requests from executing until any outstanding OpenGL rendering is done, call

`void` **glXWaitGL**`( void ) ;`

OpenGL calls made prior to **glXWaitGL** are guaranteed to be executed before X rendering calls made after **glXWaitGL**. While the same result can be achieved using **glFinish**, **glXWaitGL** does not require a round trip to the server, and is therefore more efficient in cases where the client and server are on separate machines.

glXWaitGL is ignored if there is no current rendering context. If the drawable associated with the calling thread's current context is a window that is no longer valid, a `GLXBadCurrentWindow` error is generated.

To prevent the OpenGL command sequence from executing until any outstanding X requests are completed, call

`void` **glXWaitX**`( void ) ;`

X rendering calls made prior to **glXWaitX** are guaranteed to be executed before OpenGL rendering calls made after **glXWaitX**. While the same result can be achieved using **XSync**, **glXWaitX** does not require a round trip to the server, and may therefore be more efficient.

glXWaitX is ignored if there is no current rendering context. If the drawable associated with the calling thread's current context is a window that is no longer valid, a `GLXBadCurrentWindow` error is generated.

3.2.6 Double Buffering

For drawables that are double buffered, the contents of the back buffer can be made potentially visible (i.e., become the contents of the front buffer) by calling

`void` **glXSwapBuffers** `( Display` **dpy*, `GLXDrawable` *drawable* `) ;`

The contents of the back buffer then become undefined. This operation is a no-op if *drawable* was created with a non-double-buffered visual, or if *drawable* is a `GLXPixmap`.

All GLX rendering contexts share the same notion of which are front buffers and which are back buffers for a given drawable. This notion is also shared with the X double buffer extension (DBE).

When multiple threads are rendering to the same drawable, only one of them need call `glXSwapBuffers` and all of them will see the effect of the swap. The client must synchronize the threads that perform the swap and the rendering, using some means outside the scope of GLX, to insure that each new frame is completely rendered before it is made visible.

If *dpy* and *drawable* are the display and drawable for the calling thread's current context, `glXSwapBuffers` performs an implicit `glFlush()`. Subsequent OpenGL commands can be issued immediately, but will not be executed until the buffer swapping has completed, typically during vertical retrace of the display monitor.

If *drawable* is not a valid GLX drawable, **glXSwapBuffers** generates a `GLXBadDrawable` error. If *dpy* and *drawable* are the display and drawable associated with the calling thread's current context, and if *drawable* is a window that is no longer valid, a `GLXBadCurrentWindow` error is generated.

3.2.7 Access to X Fonts

A shortcut for using X fonts is provided by the command

> `void` **glXUseXFont**`(` `Font` *font*`,` `int` *first*`,` `int` *count*`,` `int` *list_base* `) ;`

count display lists are defined starting at *list_base*, each list consisting of a single call on **glBitmap**. The definition of bitmap *list_base* + i is taken from the glyph *first* + i of *font*. If a glyph is not defined, then an empty display list is constructed for it. The `width`, `height`, `xorig`, and `yorig` of the constructed bitmap are computed from the font metrics as `rbearing-lbearing`, `ascent+descent`, `-lbearing`, and `descent-1` respectively. `xmove` is taken from the `width` metric and `ymove` is set to zero.

Note that in the direct rendering case, this requires that the bitmaps be copied to the client's address space.

glXUseXFont performs an implicit `glFlush()`.

glXUseXFont is ignored if there is no current GLX rendering context. `BadFont` is generated if *font* is not a valid X font id. `GLXBadContextState` is generated if the current GLX rendering context is in display list construction mode. `GLXBadCurrentWindow` is generated if the drawable associated with the calling thread's current context is a window and is no longer valid.

3.2.8 GLX Versioning

The following functions are available only if the GLX version is 1.1 or later.

> `const char*` **glXQueryExtensionsString(** `Display` **dpy*, `int` *screen* `) ;`

glXQueryExtensionsString returns a pointer to a string describing which GLX extensions are supported on the connection. The string is zero-terminated and contains a space-seperated list of extension names. The extension names themselves do not contain spaces. If there are no extensions to GLX, then the empty string is returned.

> `const char*` **glXGetClientString(** `Display` **dpy*, `int` *name* `);`

glXGetClientString returns a pointer to a static, zero-terminated string describing some aspect of the client library. The possible values for *name* are `GLX_VENDOR`, `GLX_VERSION`, and `GLX_EXTENSIONS`. If *name* is not set to one of these values then **NULL** is returned. The format and contents of the vendor string is implementation dependent, and the format of the extension string is the same as for **glXQueryExtensionsString**. The version string is laid out as follows:

> <major_version.minor_version><space><vendor-specific info>

Both the major and minor portions of the version number are of arbitrary length. The vendor-specific information is optional. However, if it is present, the format and contents are implementation specific.

> `const char*` **glXQueryServerString(** `Display` **dpy*, `int` *screen*, `int` *name* `) ;`

glXQueryServerString returns a pointer to a static, zero-terminated string describing some aspect of the server's GLX extension. The possible values for *name* and the format of the strings is the same as for **glXGetClientString**. If *name* is not set to a recognized value then **NULL** is returned.

4 Encoding on the X Byte Stream

In the remote rendering case, the overhead associated with interpreting the GLX extension requests must be minimized. For this reason, all commands have been broken up into two categories: OpenGL and GLX commands that are each implemented as a single X extension request and OpenGL rendering requests that are batched within a `GLXRender` request.

4.1 Requests that hold a single extension request

Each of the commands from glx.h (that is, the **glX*** commands) is encoded by a separate X extension request. In addition, there is a separate X extension request for each of the OpenGL commands that cannot be put into a display list. That list consists of all the **glGet*** commands plus

glAreTexturesResident
glDeleteLists
glDeleteTextures
glEndList
glFeedbackBuffer
glFinish
glFlush
glGenLists
glGenTextures
glIsEnabled
glIsList
glIsTexture
glNewList
glPixelStoref
glPixelStorei
glReadPixels
glRenderMode

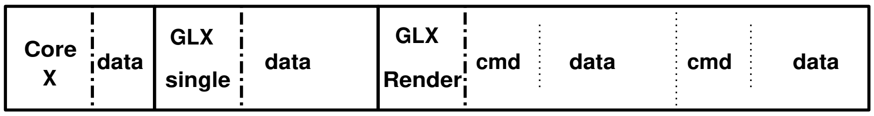

Figure 2. GLX byte stream.

glSelectBuffer

The two **PixelStore** commands (**glPixelStorei** and **glPixelStoref**) are exceptions. These commands are issued to the server only to allow it to set its error state appropriately. Pixel storage state is maintained entirely on the client side. When pixel data is transmitted to the server (by **glDrawPixels**, for example), the pixel storage information that describes it is transmitted as part of the same protocol request. Implementations may not change this behavior, because such changes would cause shared contexts to behave incorrectly.

4.2 Request that holds multiple OpenGL commands

The remaining OpenGL commands are those that may be put into display lists. Multiple occurrences of these commands are grouped together into a single X extension request (**GLXRender**). This is diagrammed in Figure 2.

The grouping minimizes dispatching within the X server. The library packs as many OpenGL commands as possible into a single X request (without exceeding the maximum size limit). No OpenGL command may be split across multiple **GLXRender** requests.

For long OpenGL commands (those longer than a maximum X request size), a series of **GLXRenderLarge** commands is issued. The structure of the OpenGL command within **GLXRenderLarge** is the same as for **GLXRender**.

Note that it is legal to have a **glBegin** in one request, followed by **glVertex** commands, and eventually the matching **glEnd** in a subsequent request. A command is not the same as an OpenGL primitive.

4.3 Wire representations and byte swapping

Unsigned and signed integers are represented as they are represented in the core X protocol. Single and double precision floating point numbers are sent and received in IEEE floating point format. The X byte stream and network specifications make it impossible for the client to assure that double precision floating point numbers will be naturally aligned within the transport buffers of the server. For those architectures that require it, the server or client must copy those floating point numbers to a properly aligned buffer before using them.

Byte swapping on the encapsulated OpenGL byte stream is performed by the server using the same rule as the core X protocol. Single precision floating point values are swapped in the same way that 32-bit integers are swapped. Double precision floating point values are swapped across all 8 bytes.

4.4 Sequentiality

There are two sequences of commands: the X stream, and the OpenGL stream. In general these two streams are independent: Although the commands in each stream will be processed in sequence, there is no guarantee that commands in the separate streams will be processed in the order in which they were issued by the calling thread.

An exception to this rule arises when a single command appears in *both* streams. This forces the two streams to rendezvous.

Because the processing of the two streams may take place at different rates, and some operations may depend on the results of commands in a different stream, we distinguish between commands assigned to each of the X and OpenGL streams.

The following commands are processed on the client side and therefore do not exist in either the X or the OpenGL stream:

glXGetClientString
glXGetCurrentContext
glXGetCurrentDisplay
glXGetCurrentDrawable
glXGetConfig

The following commands are in the X stream and obey the sequentiality guarantees for X requests:

glXCreateContext
glXDestroyContext
glXMakeCurrent
glXIsDirect
glXQueryExtensionsString
glXQueryServerString
glXQueryVersion
glXWaitGL
glXCreateGLXPixmap
glXDestroyGLXPixmap
glXChooseVisual
glXSwapBuffers (but see below)
glXCopyContext (see below)

glXSwapBuffers is in the X stream if and only if the display and drawable are not those belonging to the calling thread's current context; otherwise it is in the OpenGL stream. **glXCopyContext** is in the X stream alone if and only if its source context differs from the calling thread's current context; otherwise it is in both streams.

Commands in the OpenGL stream, which obey the sequentiality guarantees for OpenGL requests are:

glXWaitX
glXSwapBuffers (see below)
All OpenGL Commands

glXSwapBuffers is in the OpenGL stream if and only if the display and drawable are those belonging to the calling thread's current context; otherwise it is in the X stream.

Commands in both streams, which force a rendezvous are:

glXCopyContext (see below)
glXUseXFont

glXCopyContext is in both streams if and only if the source context is the same as the current context of the calling thread; otherwise it is in the X stream only.

5 Extending OpenGL

OpenGL is extended by adding new GLX requests, OpenGL requests or additional enumerated values to the OpenGL requests. The OpenGL Architectural Review Board maintains a registry of indexes for each vendor to use as they wish.

New names must clearly indicate to clients whether some particular feature is in the core OpenGL or is vendor specific. To make a vendor-specific name, append a company identifier (in upper case) and any additional vendor-specific tags (e.g. machine names). For instance, SGI might add new commands and manifest constants of the form **glNewCommandSGI** and **GL_NEW_DEFINITION_SGI**. If SGI wanted to provide extensions that were specific to its Reality Engine, then the names might be of the form **glNewCommandSGIre** and **GL_NEW_DEFINITION_SGI_RE**. If two or more licensees agree in good faith to implement the same extension, and to make the specification of that extension publicly available, the procedures and tokens that are defined by the extension can be suffixed by **EXT**.

6 Glossary

Address Space the set of objects or memory locations accessible through a single name space. In other words, it is a data region that one or more processes may share through pointers.

Client an X client. An application communicates to a server by some path. The application program is referred to as a client of the window system server. To the server, the client is the communication path itself. A program with multiple connections is viewed as multiple clients to the server. The resource lifetimes are controlled by the connection lifetimes, not the application program lifetimes.

Connection a bidirectional byte stream that carries the X (and GLX) protocol between the client and the server. A client typically has only one connection to a server.

(Rendering) Context a OpenGL rendering context. This is a virtual OpenGL machine. All OpenGL rendering is done with respect to a context. The state maintained by one rendering context is not affected by another except in case of shared display lists.

GLXContext a handle to a rendering context. Rendering contexts consist of client side state and server side state.

Similar a potential correspondence among `GLXDrawable`s and rendering contexts. `Window`s and `GLXPixmap`s are similar to a rendering context are similar if, and only if, they have been created with respect to the same `VisualID` and root window.

Thread one of a group of processes all sharing the same address space. Typically, each thread will have its own program counter and stack pointer, but the text and data spaces are visible to each of the threads. A thread that is the only member of its group is equivalent to a process.