OpenGL® Graphics with the X Window System® (Version 1.3)

Document Editors (version 1.3): Paula Womack, Jon Leech

Contents

List of Figures

List of Tables

Chapter 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 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 such execution is inappropriate.

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

Chapter 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`. X provides two types of `Drawable`s: `Window`s which are located onscreen and `Pixmap`s which are maintained offscreen. The GLX equivalent to a `Window` is a `GLXWindow` and the GLX equivalent to a `Pixmap` is a `GLXPixmap`. GLX introduces a third type of drawable, called a `GLXPbuffer`, for which there is no X equivalent. `GLXPbuffer`s are used for offscreen rendering but they have different semantics than `GLXPixmap`s that make it easier to allocate them in non-visible frame buffer memory.

`GLXWindow`s, `GLXPixmap`s and `GLXPbuffer`s are created with respect to a `GLXFBConfig`; the `GLXFBConfig` describes the depth of the color buffer components and the types, quantities and sizes of the *ancillary buffers* (i.e., the depth, accumulation, auxiliary, and stencil buffers). Double buffering and stereo capability is also fixed by the `GLXFBConfig`.

Ancillary buffers are associated with a `GLXDrawable`, not with a rendering context. If several rendering contexts 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 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 any `GLXDrawable` that it is *compatible* with (subject to the restrictions discussed in the section on address space and the restrictions discussed under **glXCreatePixmap**). A drawable and context are compatible if they

- support the same type of rendering (e.g., RGBA or color index)
- have color buffers and ancillary buffers of the same depth. For example, a `GLXDrawable` that has a front left buffer and a back left buffer with red, green and blue sizes of 4 would not be compatible with a context that was created with a visual or `GLXFBConfig` that has only a front left buffer with red, green and blue sizes of 8. However, it would be compatible with a context that was created with a `GLXFBConfig` that has only a front left buffer if the red, green and blue sizes are 4.
- were created with respect to the same X screen

As long as the compatibility constraint is satisfied (and the address space requirement is satisfied), applications can render into the same `GLXDrawable`, using different rendering contexts. It is also possible to use a single context to render into multiple `GLXDrawable`s.

For backwards compatibility with GLX versions 1.2 and earlier, a rendering context can also be used to render into a `Window`. Thus, a `GLXDrawable` is the union {`GLXWindow`, `GLXPixmap`, `GLXPbuffer`, `Window`}. In X, `Window`s are associated with a `Visual`. In GLX the definition of `Visual` has been extended to include the types, quantities and sizes of the ancillary buffers and information indicating whether or not the `Visual` is double buffered. For backwards compatibility, a `GLXPixmap` can also be created using a `Visual`.

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.

Each thread can have at most one current rendering context. In addition, a rendering context can be current for only one thread at a time. The client is responsible for creating a rendering context and a drawable.

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

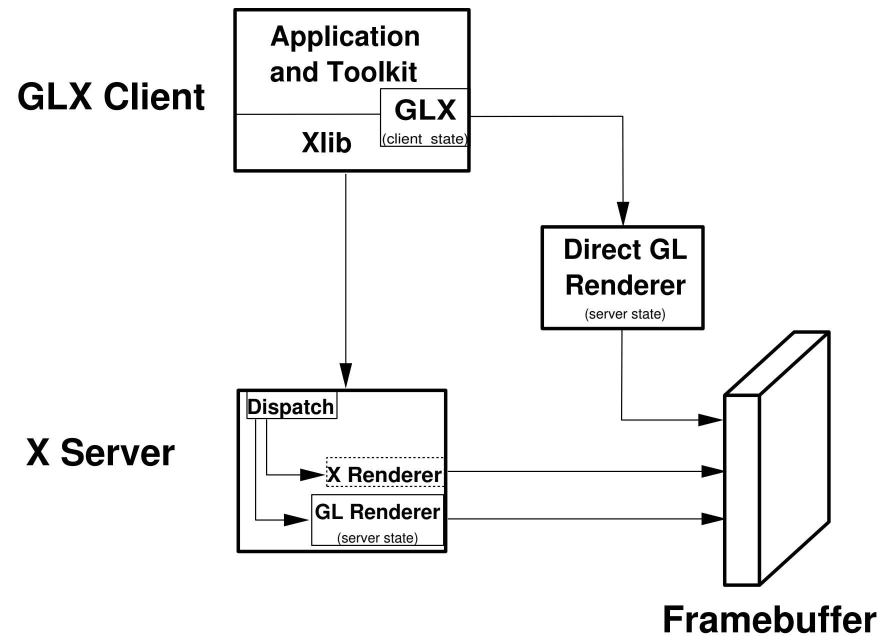

Figure 2.1. Direct and Indirect Rendering Block Diagram.

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 OpenGL implementation is that of the direct process; when direct rendering is not being used (i.e., when indirect rendering is occurring), the address space of the OpenGL implementation 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 (see figure 2.1). Both the client context state and the server context state of 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.

Although direct rendering support is optional, all implementations are required to support indirect rendering.

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. Since the lists are part of the server context state they can be shared only if the server state for the sharing contexts exists in 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.

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.

Note that the list named in a **glNewList** call is not created or superseded until **glEndList** is called. Thus 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. If one context deletes a list that is being executed by another context, the second context will continue executing the old contents of the list until it reaches the end.

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 display lists must handle the case where one rendering context is using a display list when another rendering context destroys that list or redefines it.

In general, OpenGL commands are not guaranteed to be atomic. The operation of **glEndList** and **glDeleteLists** are exceptions: modifications to the shared context state as a result of executing **glEndList** or **glDeleteLists** are atomic.

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 (`GL_TEXTURE_1D`, `GL_TEXTURE_2D` or `GL_TEXTURE_3D`) 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. (Like display lists, texture objects are part of the server context state.) 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.

All modifications to shared context state as a result of executing **glBindTexture** are atomic. Also, 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 in the overlay planes and a second in the main planes and then move them independently or in concert to keep them aligned. To keep the overlay and main plane windows aligned, the 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.

Chapter 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 **glRenderMode** `GL_FEEDBACK` or `GL_SELECT`, or a call to **glXMakeCurrent** was made between a **glBegin** and the corresponding call to **glEnd**.

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

`GLXBadCurrentWindow` The current `Window` of the calling thread is a window that is no longer valid. This error is being deprecated in favor of `GLXBadCurrentDrawable`.

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

`GLXBadFBConfig` The `GLXFBConfig` argument does not name a `GLXFBConfig`.

`GLXBadPbuffer` The `GLXPbuffer` argument does not name a `GLXPbuffer`.

`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.

`GLXBadWindow` The `GLXWindow` argument does not name a `GLXWindow`.

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 Events

GLX introduces one new event:

`GLX_PbufferClobber` The given pbuffer has been removed from framebuffer memory and may no longer be valid. These events are generated as a result of conflicts in the framebuffer allocation between two drawables when one or both of the drawables are pbuffers.

3.3 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.

3.3.1 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 and *event_base* is used to return the value of the first event code. The constant error codes and event codes should be added to these base values to get the actual value.

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.

3.3.2 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.

3.3.3 Configuration Management

A `GLXFBConfig` describes the format, type and size of the color buffers and ancillary buffers for a `GLXDrawable`. When the `GLXDrawable` is a `GLXWindow` then the `GLXFBConfig` that describes it has an associated X `Visual`; for `GLXPixmaps` and `GLXPbuffers` there may or may not be an X `Visual` associated with the `GLXFBConfig`.

The attributes for a `GLXFBConfig` are shown in Table 3.1. The constants shown here are passed to **glXGetFBConfigs** and **glXChooseFBConfig** to specify which attributes are being queried.

`GLX_BUFFER_SIZE` gives the total depth of the color buffer in bits. For `GLXFBConfigs` that correspond to a `PseudoColor` or `StaticColor` visual, this is equal to the depth value reported in the core X11 `Visual`. For `GLXFBConfigs` that correspond to a `TrueColor` or `DirectColor` visual, `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 `GLXFBConfigs` that correspond to a `TrueColor` visual, 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.

The attribute `GLX_RENDER_TYPE` has as its value a mask indicating what type of `GLXContext` a drawable created with the corresponding `GLXFBConfig` can be bound to. The following bit settings are supported: `GLX_RGBA_BIT` and `GLX_COLOR_INDEX_BIT`. If both of these bits are set in the mask then drawables created with the `GLXFBConfig` can be bound to both RGBA and color index rendering contexts.

Attribute	Type	Notes
GLX_FBCONFIG_ID	XID	XID of GLXFBConfig
GLX_BUFFER_SIZE	integer	depth of the color buffer
GLX_LEVEL	integer	frame buffer level
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	no. of auxiliary color buffers
GLX_RED_SIZE	integer	no. of bits of Red in the color buffer
GLX_GREEN_SIZE	integer	no. of bits of Green in the color buffer
GLX_BLUE_SIZE	integer	no. of bits of Blue in the color buffer
GLX_ALPHA_SIZE	integer	no. of bits of Alpha in the color buffer
GLX_DEPTH_SIZE	integer	no. of bits in the depth buffer
GLX_STENCIL_SIZE	integer	no. of bits in the stencil buffer
GLX_ACCUM_RED_SIZE	integer	no. Red bits in the accum. buffer
GLX_ACCUM_GREEN_SIZE	integer	no. Green bits in the accum. buffer
GLX_ACCUM_BLUE_SIZE	integer	no. Blue bits in the accum. buffer
GLX_ACCUM_ALPHA_SIZE	integer	no. of Alpha bits in the accum. buffer
GLX_RENDER_TYPE	bitmask	which rendering modes are supported.
GLX_DRAWABLE_TYPE	bitmask	which GLX drawables are supported.
GLX_X_RENDERABLE	boolean	True if X can render to drawable
GLX_X_VISUAL_TYPE	integer	X visual type of the associated visual
GLX_CONFIG_CAVEAT	enum	any caveats for the configuration
GLX_TRANSPARENT_TYPE	enum	type of transparency supported
GLX_TRANSPARENT_INDEX_VALUE	integer	transparent index value
GLX_TRANSPARENT_RED_VALUE	integer	transparent red value
GLX_TRANSPARENT_GREEN_VALUE	integer	transparent green value
GLX_TRANSPARENT_BLUE_VALUE	integer	transparent blue value
GLX_TRANSPARENT_ALPHA_VALUE	integer	transparent alpha value
GLX_MAX_PBUFFER_WIDTH	integer	maximum width of GLXPbuffer
GLX_MAX_PBUFFER_HEIGHT	integer	maximum height of GLXPbuffer
GLX_MAX_PBUFFER_PIXELS	integer	maximum size of GLXPbuffer
GLX_VISUAL_ID	integer	XID of corresponding Visual

Table 3.1: GLXFBConfig attributes.

GLX Token Name	Description
GLX_WINDOW_BIT	GLXFBConfig supports windows
GLX_PIXMAP_BIT	GLXFBConfig supports pixmaps
GLX_PBUFFER_BIT	GLXFBConfig supports pbuffers

Table 3.2: Types of Drawables Supported by GLXFBConfig

GLX Token Name	X Visual Type
GLX_TRUE_COLOR	TrueColor
GLX_DIRECT_COLOR	DirectColor
GLX_PSEUDO_COLOR	PseudoColor
GLX_STATIC_COLOR	StaticColor
GLX_GRAY_SCALE	GrayScale
GLX_STATIC_GRAY	StaticGray
GLX_X_VISUAL_TYPE	No associated Visual

Table 3.3: Mapping of Visual Types to GLX tokens.

The attribute GLX_DRAWABLE_TYPE has as its value a mask indicating the drawable types that can be created with the corresponding GLXFBConfig (the config is said to "support" these drawable types). The valid bit settings are shown in Table 3.2.

For example, a GLXFBConfig for which the value of the GLX_DRAWABLE_TYPE attribute is

GLX_WINDOW_BIT | GLX_PIXMAP_BIT | GLX_PBUFFER_BIT

can be used to create any type of GLX drawable, while a GLXFBConfig for which this attribute value is GLX_WINDOW_BIT can not be used to create a GLXPixmap or a GLXPbuffer.

GLX_X_RENDERABLE is a boolean indicating whether X can be used to render into a drawable created with the GLXFBConfig. This attribute is True if the GLXFBConfig supports GLX windows and/or pixmaps.

If a GLXFBConfig supports windows then it has an associated X Visual. The value of the GLX_VISUAL_ID attribute specifies the XID of the Visual and the value of the GLX_X_VISUAL_TYPE attribute specifies the type of Visual. The possible values are shown in Table 3.3. If a GLXFBConfig does not support windows, then querying GLX_VISUAL_ID will return 0 and querying GLX_X_VISUAL_TYPE will return GLX_NONE.

Note that RGBA rendering may be supported for any of the six Visual types but color index rendering is supported only for PseudoColor, StaticColor, GrayScale, and StaticGray visuals (i.e., single-channel visuals). If RGBA rendering is supported for a single-channel visual (i.e., if the GLX_RENDER_TYPE attribute has the GLX_RGBA_BIT set), then the red component maps to the color buffer bits corresponding to the core X11 visual. The green and blue components map to non-displayed color buffer bits and the alpha component maps to non-displayed alpha buffer bits if their sizes are nonzero, otherwise they are discarded.

The GLX_CONFIG_CAVEAT attribute may be set to one of the following values: GLX_NONE, GLX_SLOW_CONFIG or GLX_NON_CONFORMANT_CONFIG. If the attribute is set to GLX_NONE then the configuration has no caveats; if it is set to GLX_SLOW_CONFIG then rendering to a drawable with this configuration may run at reduced performance (for example, the hardware may not support the color buffer depths described by the configuration); if it is set to GLX_NON_CONFORMANT_CONFIG then rendering to a drawable with this configuration will not pass the required OpenGL conformance tests.

Servers are required to export at least one GLXFBConfig that supports RGBA rendering to windows and passes OpenGL conformance (i.e., the GLX_RENDER_TYPE attribute must have the GLX_RGBA_BIT set, the GLX_DRAWABLE_TYPE attribute must have the GLX_WINDOW_BIT set and the GLX_CONFIG_CAVEAT attribute must not be set to GLX_NON_CONFORMANT_CONFIG). This GLXFBConfig 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; auxillary buffers are optional, and the alpha buffer may have 0 bits. The color buffer size for this GLXFBConfig must be as large as that of the deepest TrueColor, DirectColor, PseudoColor, or StaticColor visual supported on framebuffer level zero (the main image planes), and this configuration must be available on framebuffer level zero.

If the X server exports a PseudoColor or StaticColor visual on framebuffer level 0, a GLXFBConfig that supports color index rendering to windows and passes OpenGL conformance is also required (i.e., the GLX_RENDER_TYPE attribute must have the GLX_COLOR_INDEX_BIT set, the GLX_DRAWABLE_TYPE attribute must have the GLX_WINDOW_BIT set, and the GLX_CONFIG_CAVEAT attribute must not be set to GLX_NON_CONFORMANT_CONFIG). This GLXFBConfig 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 the configuration must be made available on level zero.

The attribute GLX_TRANSPARENT_TYPE indicates whether or not the configuration supports transparency, and if it does support transparency, what type of transparency is available. If the attribute is set to `GLX_NONE` then windows created with the `GLXFBConfig` will not have any transparent pixels. If the attribute is `GLX_TRANSPARENT_RGB` or `GLX_TRANSPARENT_INDEX` then the `GLXFBConfig` supports transparency. `GLX_TRANSPARENT_RGB` is only applicable if the configuration is associated with a `TrueColor` or `DirectColor` visual: a transparent pixel will be drawn when the red, green and blue values which are read from the framebuffer are equal to `GLX_TRANSPARENT_RED_VALUE`, `GLX_TRANSPARENT_GREEN_VALUE` and `GLX_TRANSPARENT_BLUE_VALUE`, respectively. If the configuration is associated with a `PseudoColor`, `StaticColor`, `GrayScale` or `StaticGray` visual the transparency mode `GLX_TRANSPARENT_INDEX` is used. In this case, a transparent pixel will be drawn when the value that is read from the framebuffer is equal to `GLX_TRANSPARENT_INDEX_VALUE`.

If `GLX_TRANSPARENT_TYPE` is `GLX_NONE` or `GLX_TRANSPARENT_RGB`, then the value for `GLX_TRANSPARENT_INDEX_VALUE` is undefined. If `GLX_TRANSPARENT_TYPE` is `GLX_NONE` or `GLX_TRANSPARENT_INDEX`, then the values for `GLX_TRANSPARENT_RED_VALUE`, `GLX_TRANSPARENT_GREEN_VALUE`, and `GLX_TRANSPARENT_BLUE_VALUE` are undefined. When defined, `GLX_TRANSPARENT_RED_VALUE`, `GLX_TRANSPARENT_GREEN_VALUE`, and `GLX_TRANSPARENT_BLUE_VALUE` are integer framebuffer values between 0 and the maximum framebuffer value for the component. For example, `GLX_TRANSPARENT_RED_VALUE` will range between 0 and (2**`GLX_RED_SIZE`)-1. (`GLX_TRANSPARENT_ALPHA_VALUE` is for future use.)

`GLX_MAX_PBUFFER_WIDTH` and `GLX_MAX_PBUFFER_HEIGHT` indicate the maximum width and height that can be passed into **glXCreatePbuffer** and `GLX_MAX_PBUFFER_PIXELS` indicates the maximum number of pixels (width times height) for a `GLXPbuffer`. Note that an implementation may return a value for `GLX_MAX_PBUFFER_PIXELS` that is less than the maximum width times the maximum height. Also, the value for `GLX_MAX_PBUFFER_PIXELS` is static and assumes that no other pbuffers or X resources are contending for the framebuffer memory. Thus it may not be possible to allocate a pbuffer of the size given by `GLX_MAX_PBUFFER_PIXELS`.

Use

GLXFBConfig ***glXGetFBConfigs**(Display **dpy*, int *screen*, int **nelements*);

to get the list of all `GLXFBConfig`s that are available on the specified screen. The call returns an array of `GLXFBConfig`s; the number of elements in the array is returned in *nelements*.

Use

```
GLXFBConfig *glXChooseFBConfig(Display *dpy, int
    screen, const int *attrib_list, int *nelements);
```

to get GLXFBConfigs that match a list of attributes.

This call returns an array of GLXFBConfigs that match the specified attributes (attributes are described in Table 3.1). The number of elements in the array is returned in *nelements*.

If *attrib_list* contains an undefined GLX attribute, *screen* is invalid, or *dpy* does not support the GLX extension, then NULL is returned.

All attributes in *attrib_list*, including boolean attributes, are immediately followed by the corresponding desired value. The list is terminated with None. If an attribute is not specified in *attrib_list*, then the default value (listed in Table 3.4) is used (it is said to be specified implicitly). For example, if GLX_STEREO is not specified then it is assumed to be False. If GLX_DONT_CARE is specified as an attribute value, then the attribute will not be checked. GLX_DONT_CARE may be specified for all attributes except GLX_LEVEL. If *attrib_list* is NULL or empty (first attribute is None), then selection and sorting of GLXFBConfigs is done according to the default criteria in Tables 3.4 and 3.1, as described below under **Selection** and **Sorting**.

Selection of GLXFBConfigs

Attributes are matched in an attribute-specific manner, as shown in Table 3.4. The match criteria listed in the table have the following meanings:

Smaller GLXFBConfigs with an attribute value that meets or exceeds the specified value are returned.

Larger GLXFBConfigs with an attribute value that meets or exceeds the specified value are returned.

Exact Only GLXFBConfigs whose attribute value exactly matches the requested value are considered.

Mask Only GLXFBConfigs for which the set bits of attribute include all the bits that are set in the requested value are considered. (Additional bits might be set in the attribute).

Some of the attributes, such as GLX_LEVEL, must match the specified value exactly; others, such as GLX_RED_SIZE must meet or exceed the specified minimum values.

To retrieve an GLXFBConfig given its XID, use the GLX_FBCONFIG_ID attribute. When GLX_FBCONFIG_ID is specified, all other attributes are ignored, and only the GLXFBConfig with the given XID is returned (NULL is returned if it does not exist).

If GLX_MAX_PBUFFER_WIDTH, GLX_MAX_PBUFFER_HEIGHT, GLX_MAX_PBUFFER_PIXELS, or GLX_VISUAL_ID are specified in *attrib_list*, then they are ignored (however, if present, these attributes must still be followed by an attribute value in *attrib_list*). If GLX_DRAWABLE_TYPE is specified in *attrib_list* and the mask that follows does not have GLX_WINDOW_BIT set, then the GLX_X_VISUAL_TYPE attribute is ignored.

If GLX_TRANSPARENT_TYPE is set to GLX_NONE in *attrib_list*, then inclusion of GLX_TRANSPARENT_INDEX_VALUE, GLX_TRANSPARENT_RED_VALUE, GLX_TRANSPARENT_GREEN_VALUE, GLX_TRANSPARENT_BLUE_VALUE, or GLX_TRANSPARENT_ALPHA_VALUE will be ignored.

If no GLXFBConfig matching the attribute list exists, then NULL is returned. If exactly one match is found, a pointer to that GLXFBConfig is returned.

Sorting of GLXFBConfigs

If more than one matching GLXFBConfig is found, then a list of GLXFBConfigs, sorted according to the *best* match criteria, is returned. The list is sorted according to the following precedence rules that are applied in ascending order (i.e., configurations that are considered equal by lower numbered rule are sorted by the higher numbered rule):

1. By GLX_CONFIG_CAVEAT where the precedence is GLX_NONE, GLX_SLOW_CONFIG, GLX_NON_CONFORMANT_CONFIG.

2. Larger total number of RGBA color bits (GLX_RED_SIZE, GLX_GREEN_SIZE, GLX_BLUE_SIZE, plus GLX_ALPHA_SIZE). If the requested number of bits in *attrib_list* for a particular color component is 0 or GLX_DONT_CARE, then the number of bits for that component is not considered.

3. Smaller GLX_BUFFER_SIZE.

4. Single buffered configuration (GLX_DOUBLE_BUFFER being False) precedes a double buffered one.

5. Smaller GLX_AUX_BUFFERS.

Attribute	Default	Selection and Sorting Criteria	Sort Priority
GLX_FBCONFIG_ID	GLX_DONT_CARE	*Exact*	
GLX_BUFFER_SIZE	0	*Smaller*	3
GLX_LEVEL	0	*Exact*	
GLX_DOUBLEBUFFER	GLX_DONT_CARE	*Exact*	4
GLX_STEREO	False	*Exact*	
GLX_AUX_BUFFERS	0	*Smaller*	5
GLX_RED_SIZE	0	*Larger*	2
GLX_GREEN_SIZE	0	*Larger*	2
GLX_BLUE_SIZE	0	*Larger*	2
GLX_ALPHA_SIZE	0	*Larger*	2
GLX_DEPTH_SIZE	0	*Larger*	6
GLX_STENCIL_SIZE	0	*Larger*	7
GLX_ACCUM_RED_SIZE	0	*Larger*	8
GLX_ACCUM_GREEN_SIZE	0	*Larger*	8
GLX_ACCUM_BLUE_SIZE	0	*Larger*	8
GLX_ACCUM_ALPHA_SIZE	0	*Larger*	8
GLX_RENDER_TYPE	GLX_RGBA_BIT	*Mask*	
GLX_DRAWABLE_TYPE	GLX_WINDOW_BIT	*Mask*	
GLX_X_RENDERABLE	GLX_DONT_CARE	*Exact*	
GLX_X_VISUAL_TYPE	GLX_DONT_CARE	*Exact*	9
GLX_CONFIG_CAVEAT	GLX_DONT_CARE	*Exact*	1
GLX_TRANSPARENT_TYPE	GLX_NONE	*Exact*	
GLX_TRANSPARENT_INDEX_VALUE	GLX_DONT_CARE	*Exact*	
GLX_TRANSPARENT_RED_VALUE	GLX_DONT_CARE	*Exact*	
GLX_TRANSPARENT_GREEN_VALUE	GLX_DONT_CARE	*Exact*	
GLX_TRANSPARENT_BLUE_VALUE	GLX_DONT_CARE	*Exact*	
GLX_TRANSPARENT_ALPHA_VALUE	GLX_DONT_CARE	*Exact*	

Table 3.4: Default values and match criteria for GLXFBConfig attributes.

6. Larger GLX_DEPTH_SIZE.

7. Smaller GLX_STENCIL_BITS.

8. Larger total number of accumulation buffer color bits (GLX_ACCUM_RED_SIZE, GLX_ACCUM_GREEN_SIZE, GLX_ACCUM_BLUE_SIZE, plus GLX_ACCUM_ALPHA_SIZE). If the requested number of bits in *attrib_list* for a particular color component is 0 or GLX_DONT_CARE, then the number of bits for that component is not considered.

9. By GLX_X_VISUAL_TYPE where the precedence is GLX_TRUE_COLOR, GLX_DIRECT_COLOR, GLX_PSEUDO_COLOR, GLX_STATIC_COLOR, GLX_GRAY_SCALE, GLX_STATIC_GRAY.

Use **XFree** to free the memory returned by **glXChooseFBConfig**.

To get the value of a GLX attribute for a GLXFBConfig use

```
int glXGetFBConfigAttrib(Display *dpy, GLXFBConfig
    config, int attribute, int *value);
```

If **glXGetFBConfigAttrib** succeeds then it returns Success and the value for the specified attribute is returned in *value*; otherwise it returns one of the following errors:

GLX_BAD_ATTRIBUTE *attribute* is not a valid GLX attribute.

Refer to Table 3.1 and Table 3.4 for a list of valid GLX attributes.

A GLXFBConfig has an associated X Visual only if the GLX_DRAWABLE_TYPE attribute has the GLX_WINDOW_BIT bit set. To retrieve the associated visual, call:

```
XVisualInfo *glXGetVisualFromFBConfig(Display
    *dpy, GLXFBConfig config);
```

If *config* is a valid GLXFBConfig and it has an associated X visual then information describing that visual is returned; otherwise NULL is returned. Use **XFree** to free the data returned.

3.3.4 On Screen Rendering

To create an onscreen rendering area, first create an X `Window` with a visual that corresponds to the desired `GLXFBConfig`, then call

GLXWindow **glXCreateWindow**(Display **dpy*,
GLXFBConfig *config*, Window *win*, const int
**attrib_list*);

glXCreateWindow creates a `GLXWindow` and returns its XID. Any GLX rendering context created with a compatible `GLXFBConfig` can be used to render into this window.

attrib_list specifies a list of attributes for the window. The list has the same structure as described for **glXChooseFBConfig**. Currently no attributes are recognized, so *attrib_list* must be `NULL` or empty (first attribute of `None`).

If *win* was not created with a visual that corresponds to *config*, then a `BadMatch` error is generated. (i.e., **glXGetVisualFromFBConfig** must return the visual corresponding to *win* when the `GLXFBConfig` parameter is set to *config*.) If *config* does not support rendering to windows (the `GLX_DRAWABLE_TYPE` attribute does not contain `GLX_WINDOW_BIT`), a `BadMatch` error is generated. If *config* is not a valid `GLXFBConfig`, a `GLXBadFBConfig` error is generated. If *win* is not a valid window XID, then a `BadWindow` error is generated. If there is already a `GLXFBConfig` associated with *win* (as a result of a previous **glXCreateWindow** call), then a `BadAlloc` error is generated. Finally, if the server cannot allocate the new GLX window, a `BadAlloc` error is generated.

A `GLXWindow` is destroyed by calling

glXDestroyWindow(Display **dpy*, GLXWindow *win*);

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

If *win* is not a valid GLX window then a `GLXBadWindow` error is generated.

3.3.5 Off Screen Rendering

GLX supports two types of offscreen rendering surfaces: `GLXPixmaps` and `GLXPbuffers`. `GLXPixmaps` and `GLXPbuffers` differ in the following ways:

1. `GLXPixmaps` have an associated X pixmap and can therefore be rendered to by X. Since a `GLXPbuffer` is a GLX resource, it may not be possible to render to it using X or an X extension other than GLX.

2. The format of the color buffers and the type and size of any associated ancillary buffers for a `GLXPbuffer` can only be described with a `GLXFBConfig`. The older method of using extended X `Visuals` to describe the configuration of a `GLXDrawable` cannot be used. (See section 3.4 for more information on extended visuals.)

3. It is possible to create a `GLXPbuffer` whose contents may be asynchronously lost at any time.

4. If the GLX implementation supports direct rendering, then it must support rendering to `GLXPbuffers` via a direct rendering context. Although some implementations may support rendering to `GLXPixmaps` via a direct rendering context, GLX does not require this to be supported.

5. The intent of the pbuffer semantics is to enable implementations to allocate pbuffers in non-visible frame buffer memory. Thus, the allocation of a `GLXPbuffer` can fail if there is insufficient framebuffer resources. (Implementations are not required to virtualize pbuffer memory.) Also, clients should deallocate `GLXPbuffers` when they are no longer using them – for example, when the program is iconified.

To create a `GLXPixmap` offscreen rendering area, first create an X `Pixmap` of the depth specified by the desired `GLXFBConfig`, then call

```
GLXPixmap glXCreatePixmap(Display *dpy, GLXFBConfig
    config, Pixmap pixmap, const int *attrib_list);
```

glXCreatePixmap creates an offscreen rendering area and returns its XID. Any GLX rendering context created with a `GLXFBConfig` that is compatible with *config* can be used to render into this offscreen area.

pixmap is used for the RGB planes of the front-left buffer of the resulting GLX offscreen rendering area. GLX pixmaps may be created with a *config* that includes back buffers and stereoscopic buffers. However, **glXSwapBuffers** is ignored for these pixmaps.

attrib_list specifies a list of attributes for the pixmap. The list has the same structure as described for **glXChooseFBConfig**. Currently no attributes are recognized, so *attrib_list* must be `NULL` or empty (first attribute of `None`).

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

If *pixmap* was not created with respect to the same screen as *config*, then a BadMatch error is generated. If *config* is not a valid GLXFBConfig or if it does not support pixmap rendering then a GLXBadFBConfig error is generated. If *pixmap* is not a valid Pixmap XID, 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

glXDestroyPixmap(Display **dpy*, GLXPixmap *pixmap*);

This request deletes the association between the XID *pixmap* and the GLX pixmap. The storage for the GLX pixmap will be freed when it is not current to any client. To free the associated X pixmap, call **XFreePixmap**.

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

To create a GLXPbuffer call

GLXPbuffer **glXCreatePbuffer**(Display **dpy*,
GLXFBConfig *config*, const int **attrib_list*);

This creates a single GLXPbuffer and returns its XID. Like other drawable types, GLXPbuffers are shared; any client which knows the associated XID can use a GLXPbuffer.

attrib_list specifies a list of attributes for the pbuffer. The list has the same structure as described for **glXChooseFBConfig**. Currently only four attributes can be specified in *attrib_list*: GLX_PBUFFER_WIDTH, GLX_PBUFFER_HEIGHT, GLX_PRESERVED_CONTENTS and GLX_LARGEST_PBUFFER.

attrib_list may be NULL or empty (first attribute of None), in which case all the attributes assume their default values as described below.

GLX_PBUFFER_WIDTH and GLX_PBUFFER_HEIGHT specify the pixel width and height of the rectangular pbuffer. The default values for GLX_PBUFFER_WIDTH and GLX_PBUFFER_HEIGHT are zero.

Use GLX_LARGEST_PBUFFER to get the largest available pbuffer when the allocation of the pbuffer would otherwise fail. The width and height of the allocated pbuffer will never exceed the values of GLX_PBUFFER_WIDTH and GLX_PBUFFER_HEIGHT, respectively. Use **glXQueryDrawable** to retrieve the dimensions of the allocated pbuffer. By default, GLX_LARGEST_PBUFFER is False.

If the GLX_PRESERVED_CONTENTS attribute is set to False in *attrib_list*, then an *unpreserved* pbuffer is created and the contents of the pbuffer may be lost at any time. If this attribute is not specified, or if it is specified as `True` in *attrib_list*, then when a resource conflict occurs the contents of the pbuffer will be *preserved* (most likely by swapping out portions of the buffer from the framebuffer to main memory). In either case, the client can register to receive a pbuffer clobber event which is generated when the pbuffer contents have been preserved or have been damaged. (See **glXSelectEvent** in section 3.3.8 for more information.)

The resulting pbuffer will contain color buffers and ancillary buffers as specified by *config*. It is possible to create a pbuffer with back buffers and to swap the front and back buffers by calling **glXSwapBuffers**. Note that pbuffers use framebuffer resources so applications should consider deallocating them when they are not in use.

If a pbuffer is created with `GLX_PRESERVED_CONTENTS` set to `False`, then portions of the buffer contents may be lost at any time due to frame buffer resource conflicts. Once the contents of a unpreserved pbuffer have been lost it is considered to be in a *damaged* state. It is not an error to render to a pbuffer that is in this state but the effect of rendering to it is the same as if the pbuffer were destroyed: the context state will be updated, but the frame buffer state becomes undefined. It is also not an error to query the pixel contents of such a pbuffer, but the values of the returned pixels are undefined. Note that while this specification allows for unpreserved pbuffers to be damaged as a result of other pbuffer activity, the intent is to have only the activity of visible windows damage pbuffers.

Since the contents of a unpreserved pbuffer can be lost at anytime with only asynchronous notification (via the pbuffer clobber event), the only way a client can guarantee that valid pixels are read back with **glReadPixels** is by grabbing the X server. (Note that this operation is potentially expensive and should not be done frequently. Also, since this locks out other X clients, it should be done only for short periods of time.) Clients that don't wish to do this can check if the data returned by **glReadPixels** is valid by calling **XSync** and then checking the event queue for pbuffer clobber events (assuming that these events had been pulled off of the queue prior to the **glReadPixels** call).

When **glXCreatePbuffer** fails to create a `GLXPbuffer` due to insufficient resources, a `BadAlloc` error is generated. If *config* is not a valid `GLXFBConfig` then a `GLXBadFBConfig` error is generated; if *config* does not support `GLXPbuffer`s then a `BadMatch` error is generated.

A `GLXPbuffer` is destroyed by calling:

void **glXDestroyPbuffer**(Display **dpy*, GLXPbuffer *pbuf*);

The XID associated with the GLXPbuffer is destroyed. The storage for the GLXPbuffer will be destroyed once it is no longer current to any client.

If *pbuf* is not a valid GLXPbuffer then a GLXBadPbuffer error is generated.

3.3.6 Querying Attributes

To query an attribute associated with a GLXDrawable call:

void **glXQueryDrawable**(Display **dpy*, GLXDrawable *draw*, int *attribute*, unsigned int **value*);

attribute must be set to one of GLX_WIDTH, GLX_HEIGHT, GLX_PRESERVED_CONTENTS, GLX_LARGEST_PBUFFER, or GLX_FBCONFIG_ID.

To get the GLXFBConfig for a GLXDrawable, first retrieve the XID for the GLXFBConfig and then call **glXChooseFBConfig**.

If *draw* is not a valid GLXDrawable then a GLXBadDrawable error is generated. If *draw* is a GLXWindow or GLXPixmap and *attribute* is set to GLX_PRESERVED_CONTENTS or GLX_LARGEST_PBUFFER, then the contents of *value* are undefined.

3.3.7 Rendering Contexts

To create an OpenGL rendering context, call

GLXContext **glXCreateNewContext**(Display **dpy*, GLXFBConfig *config*, int *render_type*, GLXContext *share_list*, Bool *direct*);

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

If *render_type* is set to GLX_RGBA_TYPE then a context that supports RGBA rendering is created; if *render_type* is set to GLX_COLOR_INDEX_TYPE then a context that supports color index rendering is created.

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. The server context state for all sharing contexts must exist in 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 **glXCreateNewContext** will attempt to create an indirect context instead.

glXCreateNewContext can generate the following errors: `GLXBadContext` if *share_list* is neither zero nor a valid GLX rendering context; `GLXBadFBConfig` if *config* is not a valid `GLXFBConfig`; `BadMatch` if the server context state for *share_list* exists in 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 *config*; `BadAlloc` if the server does not have enough resources to allocate the new context; `BadValue` if *render_type* does not refer to a valid rendering type.

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 make a context current, call

```
Bool glXMakeContextCurrent(Display *dpy,
    GLXDrawable draw, GLXDrawable read, GLXContext
    ctx);
```

glXMakeContextCurrent binds *ctx* to the current rendering thread and to the *draw* and *read* drawables. *draw* is used for all OpenGL operations except:

- Any pixel data that are read based on the value of `GL_READ_BUFFER`. Note that accumulation operations use the value of `GL_READ_BUFFER`, but are not allowed unless *draw* is identical to *read*.
- Any depth values that are retrieved by **glReadPixels** or **glCopyPixels**.
- Any stencil values that are retrieved by **glReadPixels** or **glCopyPixels**.

These frame buffer values are taken from *read*. Note that the same `GLXDrawable` may be specified for both *draw* and *read*.

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 *draw* or *read* are not compatible with *ctx* a `BadMatch` error is generated. If *ctx* is current to some other thread, then **glXMakeContextCurrent** 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`. If *ctx* is not a valid GLX rendering context, `GLXBadContext` is generated. If either *draw* or *read* are not a valid GLX drawable, a `GLXBadDrawable` error is generated. If the X `Window` underlying either *draw* or *read* is no longer valid, a `GLXBadWindow` error is generated. If the previous context of the calling thread has unflushed commands, and the previous drawable is no longer valid, `GLXBadCurrentDrawable` is generated. Note that the ancillary buffers for *draw* and *read* 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.

In addition, implementations may generate a `BadMatch` error under the following conditions: if *draw* and *read* cannot fit into framebuffer memory simultaneously; if *draw* or *read* is a `GLXPixmap` and *ctx* is a direct rendering context; if *draw* or *read* is a `GLXPixmap` and *ctx* was previously bound to a `GLXWindow` or `GLXPbuffer`; if *draw* or *read* is a `GLXWindow` or `GLXPbuffer` and *ctx* was previously bound to a `GLXPixmap`.

Other errors may arise when the context state is inconsistent with the drawable state, as described in the following paragraphs. Color buffers are treated specially because the current GL_DRAW_BUFFER and GL_READ_BUFFER context state can be inconsistent with the current draw or read drawable (for example, when GL_DRAW_BUFFER is GL_BACK and the drawable is single buffered).

No error will be generated if the value of GL_DRAW_BUFFER in *ctx* indicates a color buffer that is not supported by *draw*. In this case, all rendering will behave as if GL_DRAW_BUFFER was set to NONE. Also, no error will be generated if the value of GL_READ_BUFFER in *ctx* does not correspond to a valid color buffer. Instead, when an operation that reads from the color buffer is executed (e.g., **glReadPixels** or **glCopyPixels**), the pixel values used will be undefined until GL_READ_BUFFER is set to a color buffer that is valid in *read*. Operations that query the value of GL_READ_BUFFER or GL_DRAW_BUFFER (i.e., **glGet**, **glPushAttrib**) use the value set last in the context, independent of whether it is a valid buffer in *read* or *draw*.

Note that it is an error to later call **glDrawBuffer** and/or **glReadBuffer** (even if they are implicitly called via **glPopAttrib** or **glXCopyContext**) and specify a color buffer that is not supported by *draw* or *read*. Also, subsequent calls to **glReadPixels** or **glCopyPixels** that specify an unsupported ancillary buffer will result in an error.

If *draw* is destroyed after **glXMakeContextCurrent** is called, then subsequent rendering commands will be processed and the context state will be updated, but the frame buffer state becomes undefined. If *read* is destroyed after **glXMakeContextCurrent** then pixel values read from the framebuffer (e.g., as result of calling **glReadPixels**, **glCopyPixels** or **glCopyColorTable**) are undefined. If the X Window underlying the GLXWindow *draw* or *read* drawable is destroyed, rendering and readback are handled as above.

To release the current context without assigning a new one, set *ctx* to NULL and set *draw* and *read* to None. If *ctx* is NULL and *draw* and *read* are not None, or if *draw* or *read* are set to None and *ctx* is not NULL, then a BadMatch error will be generated.

The first time *ctx* is made current, the viewport and scissor dimensions are set to the size of the *draw* drawable (as though **glViewport**(0, 0, w, h) and **glScissor**(0, 0, w, h) were called, where w and h are the width and height of the drawable, respectively). However, the viewport and scissor dimensions are not modified when *ctx* is subsequently made current; it is the clients responsibility to reset the viewport and scissor in this case.

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.

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.

Not all GL state values can be copied. For example, client side state such as pixel pack and unpack state, vertex array state and select and feedback state cannot be copied. Also, some server state such as render mode state, the contents of the attribute and matrix stacks, display lists and texture objects, cannot be copied. The state that can be copied is exactly the state that is manipulated by **glPushAttrib**.

If *source* and *dest* were not created on the same screen or if the server context state for *source* and *dest* does not exist in the same address space, a `BadMatch` error is generated (*source* and *dest* may be based on different `GLXFBConfig`s 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 no longer valid, a `GLXBadCurrentDrawable` error 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

Attribute	Type	Description
GLX_FBCONFIG_ID	XID	XID of GLXFBConfig associated with context
GLX_RENDER_TYPE	int	type of rendering supported
GLX_SCREEN	int	screen number

Table 3.5: Context attributes.

larger number of current rendering contexts is essential for general-purpose systems, but may not be necessary for turnkey applications.)

To get the current context, call

GLXContext **glXGetCurrentContext**(void);

If there is no current context, NULL is returned.

To get the XID of the current drawable used for rendering, call

GLXDrawable **glXGetCurrentDrawable**(void);

If there is no current *draw* drawable, None is returned.

To get the XID of the current drawable used for reading, call

GLXDrawable **glXGetCurrentReadDrawable**(void);

If there is no current *read* 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.

To obtain the value of a context's attribute, use

int **glXQueryContext**(Display **dpy*, GLXContext *ctx*,
int *attribute*, int **value*);

glXQueryContext returns through *value* the value of *attribute* for *ctx*. It may cause a round trip to the server.

The values and types corresponding to each GLX context attribute are listed in Table 3.5.

glXQueryContext returns GLX_BAD_ATTRIBUTE if *attribute* is not a valid GLX context attribute and Success otherwise. If *ctx* is invalid and a round trip to the server is involved, a GLXBadContext error is generated.

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.

3.3.8 Events

GLX events are returned in the X11 event stream. GLX and X11 events are selected independently; if a client selects for both, then both may be delivered to the client. The relative order of X11 and GLX events is not specified.

A client can ask to receive GLX events on a `GLXWindow` or a `GLXPbuffer` by calling

```
void glXSelectEvent(Display *dpy, GLXDrawable draw,
   unsigned long event_mask);
```

Calling **glXSelectEvent** overrides any previous event mask that was set by the client for *draw*. Note that the GLX event mask is private to GLX (separate from the core X11 event mask), and that a separate GLX event mask is maintained in the server state for each client for each drawable.

If *draw* is not a valid `GLXPbuffer` or a valid `GLXWindow`, a `GLXBadDrawable` error is generated.

To find out which GLX events are selected for a `GLXWindow` or `GLXPbuffer` call

```
void glXGetSelectedEvent(Display *dpy, GLXDrawable
   draw, unsigned long *event_mask);
```

If *draw* is not a GLX window or pbuffer then a `GLXBadDrawable` error is generated.

Currently only one GLX event can be selected, by setting *event_mask* to `GLX_PBUFFER_CLOBBER_MASK`. The data structure describing a pbuffer clobber event is:

```
typedef struct {
     int event_type; /* GLX_DAMAGED or GLX_SAVED */
     int draw_type; /* GLX_WINDOW or GLX_PBUFFER */
     unsigned long serial; /* number of last request processed by server */
     Bool send_event; /* event was generated by a SendEvent request */
     Display *display; /* display the event was read from */
     GLXDrawable drawable; /* XID of Drawable */
     unsigned int buffer_mask; /* mask indicating which buffers are affected */
     unsigned int aux_buffer; /* which aux buffer was affected */
     int x, y;
     int width, height;
```

Bitmask	Corresponding buffer
GLX_FRONT_LEFT_BUFFER_BIT	Front left color buffer
GLX_FRONT_RIGHT_BUFFER_BIT	Front right color buffer
GLX_BACK_LEFT_BUFFER_BIT	Back left color buffer
GLX_BACK_RIGHT_BUFFER_BIT	Back right color buffer
GLX_AUX_BUFFERS_BIT	Auxillary buffer
GLX_DEPTH_BUFFER_BIT	Depth buffer
GLX_STENCIL_BUFFER_BIT	Stencil buffer
GLX_ACCUM_BUFFER_BIT	Accumulation buffer

Table 3.6: Masks identifying clobbered buffers.

```
    int count; /* if nonzero, at least this many more */
} GLXPbufferClobberEvent;
```

If an implementation doesn't support the allocation of pbuffers, then it doesn't need to support the generation of `GLXPbufferClobberEvents`.

A single X server operation can cause several pbuffer clobber events to be sent (e.g., a single pbuffer may be damaged and cause multiple pbuffer clobber events to be generated). Each event specifies one region of the `GLXDrawable` that was affected by the X Server operation. *buffer_mask* indicates which color or ancillary buffers were affected; the bits that may be present in the mask are listed in Table 3.6. All the pbuffer clobber events generated by a single X server action are guaranteed to be contiguous in the event queue. The conditions under which this event is generated and the value of *event_type* varies, depending on the type of the `GLXDrawable`.

When the `GLX_AUX_BUFFERS_BIT` is set in *buffer_mask*, then *aux_buffer* is set to indicate which buffer was affected. If more than one aux buffer was affected, then additional events are generated as part of the same contiguous event group. Each additional event will have only the `GLX_AUX_BUFFERS_BIT` set in *buffer_mask*, and the *aux_buffer* field will be set appropriately. For non-stereo drawables, `GLX_FRONT_LEFT_BUFFER_BIT` and `GLX_BACK_LEFT_BUFFER_BIT` are used to specify the front and back color buffers.

For preserved pbuffers, a pbuffer clobber event, with *event_type* `GLX_SAVED`, is generated whenever the contents of a pbuffer has to be moved to avoid being damaged. The event(s) describes which portions of the pbuffer were affected. Clients who receive many pbuffer clobber events, referring to different save actions, should consider freeing the pbuffer resource in order to prevent the system from thrashing due to insufficient resources.

For an unpreserved pbuffer a pbuffer clobber event, with *event_type* `GLX_DAMAGED`, is generated whenever a portion of the pbuffer becomes invalid.

For GLX windows, pbuffer clobber events with *event_type* `GLX_SAVED` occur whenever an ancillary buffer, associated with the window, gets moved out of offscreen memory. The event contains information indicating which color or ancillary buffers, and which portions of those buffers, were affected. GLX windows don't generate pbuffer clobber events when clobbering each others' ancillary buffers, only standard X11 damage events

3.3.9 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 no longer valid, a `GLXBadCurrentDrawable` 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 no longer valid, a `GLXBadCurrentDrawable` error is generated.

3.3.10 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
    draw);
```

The contents of the back buffer then become undefined. This operation is a no-op if *draw* was created with a non-double-buffered `GLXFBConfig`, or if *draw* 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 *draw* 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 *draw* is not a valid GLX drawable, **glXSwapBuffers** generates a `GLXBadDrawable` error. If *dpy* and *draw* are the display and drawable associated with the calling thread's current context, and if *draw* is a window that is no longer valid, a `GLXBadCurrentDrawable` error is generated. If the X `Window` underlying *draw* is no longer valid, a `GLXBadWindow` error is generated.

3.3.11 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` 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. `GLXBadCurrentDrawable` is generated if the drawable associated with the calling thread's current context is no longer valid.

3.4 Backwards Compatibility

`GLXFBConfig`s were introduced in GLX 1.3. Also, new functions for managing drawable configurations, creating pixmaps, destroying pixmaps, creating contexts and making a context current were introduced. The 1.2 versions of these functions are still available and are described in this section. Even though these older function calls are supported their use is not recommended.

3.4.1 Using Visuals for Configuration Management

In order to maintain backwards compatibility, visuals continue to be overloaded with information describing the ancillary buffers and color buffers for `GLXPixmap`s and `Windows`. Note that `Visual`s cannot be used to create `GLXPbuffer`s. Also, not all configuration attributes are exported through visuals (e.g., there is no visual attribute to describe which drawables are supported by the visual.)

The set of extended `Visual`s is fixed at server start up time. Thus a server can export multiple `Visual`s that differ only in the extended attributes. Implementors may choose to export fewer `GLXDrawable` configurations through visuals than through `GLXFBConfig`s.

The X protocol allows a single `VisualID` to be instantiated at multiple depths. Since GLX allows only one depth for any given `VisualID`, an `XVisualInfo` is used by GLX functions. An `XVisualInfo` is a {`Visual`, `Screen`, `Depth`} triple and can therefore be interpreted unambiguously.

The constants shown in Table 3.7 are passed to **glXGetConfig** and **glXChooseVisual** to specify which attributes are being queried.

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

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 color buffer
GLX_GREEN_SIZE	integer	number of bits of Green in the color buffer
GLX_BLUE_SIZE	integer	number of bits of Blue in the color buffer
GLX_ALPHA_SIZE	integer	number of bits of Alpha in the color 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
GLX_FBCONFIG_ID	integer	XID of most closely associated GLXFBConfig

Table 3.7: GLX attributes for Visuals.

```
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. RGBA rendering can be supported only by `Visual`s of type `TrueColor` or `DirectColor` (unless `GLXFBConfig`s are used), and color index rendering can be supported only by `Visual`s of type `PseudoColor` or `StaticColor`.

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 3.8. The definitions for the selection criteria `Smaller`, `Larger`, and `Exact` are given in section 3.3.3.

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.

Attribute	Default	Selection Criteria
GLX_USE_GL	True	*Exact*
GLX_BUFFER_SIZE	0	*Smaller*
GLX_LEVEL	0	*Exact*
GLX_RGBA	False	*Exact*
GLX_DOUBLEBUFFER	False	*Exact*
GLX_STEREO	False	*Exact*
GLX_AUX_BUFFERS	0	*Smaller*
GLX_RED_SIZE	0	*Larger*
GLX_GREEN_SIZE	0	*Larger*
GLX_BLUE_SIZE	0	*Larger*
GLX_ALPHA_SIZE	0	*Larger*
GLX_DEPTH_SIZE	0	*Larger*
GLX_STENCIL_SIZE	0	*Smaller*
GLX_ACCUM_RED_SIZE	0	*Larger*
GLX_ACCUM_GREEN_SIZE	0	*Larger*
GLX_ACCUM_BLUE_SIZE	0	*Larger*
GLX_ACCUM_ALPHA_SIZE	0	*Larger*

Table 3.8: Defaults and selection criteria used by **glXChooseVisual**.

If `GLX_FBCONFIG_ID` is specified in *attrib_list*, then it is ignored (however, if present, it must still be followed by an attribute value).

If an attribute is not specified in *attrib_list*, then the default value is used. See Table 3.8 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.

3.4.2 Off Screen Rendering

A `GLXPixmap` can be created using by calling

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

Calling **glXCreateGLXPixmap**(*dpy*, *visual*, *pixmap*) is equivalent to calling **glXCreatePixmap**(*dpy*, *config*, *pixmap*, `NULL`) where *config* is the `GLXFBConfig` identified by the `GLX_FBCONFIG_ID` attribute of *visual*. Before calling **glXCreateGLXPixmap**, clients must first create an X `Pixmap` of the depth specified by *visual*. The `GLXFBConfig` identified by the `GLX_FBCONFIG_ID` attribute of *visual* is associated with the resulting pixmap. Any compatible GLX rendering context can be used to render into this offscreen area.

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` created by **glXCreateGLXPixmap** can be destroyed by calling

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

This function is equivalent to **glXDestroyPixmap**; however, `GLXPixmap`s created by calls other than **glXCreateGLXPixmap** should not be passed to **glXDestroyGLXPixmap**.

3.5 Rendering Contexts

An OpenGL rendering context may be created by calling

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

Calling **glXCreateContext**(*dpy*, *visual*, *share_list*, *direct*) is equivalent to calling **glXCreateNewContext**(*dpy*, *config*, *render_type*, *share_list*, *direct*) where *config* is the `GLXFBConfig` identified by the `GLX_FBCONFIG_ID` attribute of *visual*. If *visual*'s `GLX_RGBA` attribute is `True` then *render_type* is taken as `GLX_RGBA_TYPE`, otherwise `GLX_COLOR_INDEX_TYPE`. The `GLXFBConfig` identified by the `GLX_FBCONFIG_ID` attribute of *visual* is associated with the resulting context.

glXCreateContext can generate the following 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 make a context current, call

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

Calling **glXMakeCurrent**(*dpy*, *draw*, *ctx*) is equivalent to calling **glXMakeContextCurrent**(*dpy*, *draw*, *draw*, *ctx*). Note that *draw* will be used for both the draw and read drawable.

If *ctx* and *draw* are not compatible then a `BadMatch` error will be generated. Some implementations may enforce a stricter rule and generate a `BadMatch` error if *ctx* and *draw* were not created with the same `XVisualInfo`.

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`. If *ctx* is not a valid GLX rendering context, `GLXBadContext` is generated. If *draw* is not a valid `GLXPixmap` or a valid `Window`, 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 *draw* 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.

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

Chapter 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

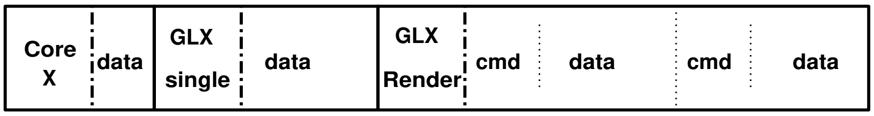

Figure 4.1. GLX byte stream.

glIsTexture
glNewList
glPixelStoref
glPixelStorei
glReadPixels
glRenderMode
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 4.1.

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 OpenGL commands whose encoding is longer than the maximum

X request size, a series of **GLXRenderLarge** commands are 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
glXGetCurrentReadDrawable
glXGetConfig
glXGetFBConfigAttrib
glXGetFBConfigs
glXGetSelectedEvent
glXGetVisualFromFBConfig

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

glXChooseFBConfig
glXChooseVisual
glXCreateContext
glXCreateGLXPixmap
glXCreateNewContext
glXCreatePbuffer
glXCreatePixmap
glXCreateWindow
glXDestroyContext
glXDestroyGLXPixmap
glXDestroyPbuffer
glXDestroyPixmap
glXDestroyWindow
glXMakeContextCurrent
glXMakeCurrent
glXIsDirect
glXQueryContext
glXQueryDrawable
glXQueryExtension
glXQueryExtensionsString
glXQueryServerString
glXQueryVersion
glXSelectEvent
glXWaitGL
glXSwapBuffers (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.

Chapter 5

Extending OpenGL

OpenGL implementors may extend OpenGL by adding new OpenGL commands or additional enumerated values for existing OpenGL commands. When a new vendor-specific command is added, GLX protocol must also be defined. If the new command is one that cannot be added to a display list, then protocol for a new **glXVendorPrivate** or **glXVendorPrivateWithReply** request is required; otherwise protocol for a new rendering command that can be sent to the X Server as part of a **glXRender** or **glXRenderLarge** request is required.

The OpenGL Architectural Review Board maintains a registry of vendor-specific enumerated values; opcodes for vendor private requests, vendor private with reply requests, and OpenGL rendering commands; and vendor-specific error codes and event codes.

New names for OpenGL functions and enumerated types must clearly indicate 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 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`.

Implementors may also extend GLX. As with OpenGL, the new names must indicate whether or not the feature is vendor-specific. (e.g., SGI might add new GLX commands and constants of the form **glXNewCommandSGI** and `GLX_NEW_DEFINITION_SGI`). When a new GLX command is added, protocol for a new **glXVendorPrivate** or **glXVendorPrivate-**

WithReply request is required.

Chapter 6

GLX Versions

Each version of GLX supports all versions of OpenGL up to the version shown in Table 6.1 corresponding to the given GLX version.

6.1 New Commands in GLX Version 1.1

The following GLX commands were added in GLX Version 1.1:

glXQueryExtensionsString
glXGetClientString
glXQueryServerString

6.2 New Commands in GLX Version 1.2

The following GLX commands were added in GLX Version 1.2:

GLX Version	Highest OpenGL Version Supported
GLX 1.0	OpenGL 1.0
GLX 1.1	OpenGL 1.0
GLX 1.2	OpenGL 1.1
GLX 1.3	OpenGL 1.2

Table 6.1: Relationship of OpenGL and GLX versions.

glXGetCurrentDisplay

6.3 New Commands in GLX Version 1.3

The following GLX commands were added in GLX Version 1.3:

glXChooseFBConfig
glXGetFBConfigAttrib
glXGetVisualFromFBConfig
glXCreateWindow
glXDestroyWindow
glXCreatePixmap
glXDestroyPixmap
glXCreatePbuffer
glXDestroyPbuffer
glXQueryDrawable
glXCreateNewContext
glXMakeContextCurrent
glXGetCurrentReadDrawable
glXQueryContext
glXSelectEvent
glXGetSelectedEvent

Chapter 7

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.

Compatible an OpenGL rendering context is compatible with (may be used to render into) a `GLXDrawable` if they meet the constraints specified in section 2.1.

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 and textures.

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. `Windows` 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.

Index of GLX Commands