Chapter 1. Introduction to OpenGL

As a software interface for graphics hardware, OpenGL's main purpose is to render two-and three-dimensional objects into a frame buffer. These objects are described as sequences of vertices (which define geometric objects) or pixels (which define images). OpenGL performs several processing steps on this data to convert it to pixels to form the final desired image in the frame buffer.

This chapter presents a global view of how OpenGL works; it contains the following major sections:

- "OpenGL Fundamentals" briefly explains basic OpenGL concepts, such as what a graphic primitive is and how OpenGL implements a client-server execution model.
- "Basic OpenGL Operation" gives a high-level description of how OpenGL processes data and produces a corresponding image in the frame buffer.

OpenGL Fundamentals

This section explains some of the concepts inherent in OpenGL.

Primitives and Commands

OpenGL draws primitives—points, line segments, or polygons—subject to several selectable modes. You can control modes independently of each other; that is, setting one mode doesn't affect whether other modes are set (although many modes may interact to determine what eventually ends up in the frame buffer). Primitives are specified, modes are set, and other OpenGL operations are described by issuing commands in the form of function calls.

Primitives are defined by a group of one or more vertices. A vertex defines a point, an endpoint of a line, or a corner of a polygon where two edges meet. Data (consisting of vertex coordinates, colors, normals, texture coordinates, and edge flags) is associated with a vertex, and each vertex and its associated data are processed independently, in order, and in the same way. The only exception to this rule is if the group of vertices must be clipped so that a particular primitive fits within a specified region; in this case, vertex data may be modified and new vertices created. The type of clipping depends on which primitive the group of vertices represents.

Commands are always processed in the order in which they are received, although there may be an indeterminate delay before a command takes effect. This means that each primitive is drawn completely before any subsequent command takes effect. It also means that state-querying commands return data that's consistent with complete execution of all previously issued OpenGL commands.
**Procedural versus Descriptive**

OpenGL provides you with fairly direct control over the fundamental operations of two- and three-dimensional graphics. This includes specification of such parameters as transformation matrices, lighting equation coefficients, antialiasing methods, and pixel update operators. However, it doesn't provide you with a means for describing or modeling complex geometric objects. Thus, the OpenGL commands you issue specify how a certain result should be produced (what procedure should be followed) rather than what exactly that result should look like. That is, OpenGL is fundamentally procedural rather than descriptive. Because of this procedural nature, it helps to know how OpenGL works—the order in which it carries out its operations, for example—in order to fully understand how to use it.

**Execution Model**

The model for interpretation of OpenGL commands is client-server. An application (the client) issues commands, which are interpreted and processed by OpenGL (the server). The server may or may not operate on the same computer as the client. In this sense, OpenGL is network-transparent. A server can maintain several GL *contexts*, each of which is an encapsulated GL state. A client can connect to any one of these contexts. The required network protocol can be implemented by augmenting an already existing protocol (such as that of the X Window System) or by using an independent protocol. No OpenGL commands are provided for obtaining user input.

The effects of OpenGL commands on the frame buffer are ultimately controlled by the window system that allocates frame buffer resources. The window system determines which portions of the frame buffer OpenGL may access at any given time and communicates to OpenGL how those portions are structured. Therefore, there are no OpenGL commands to configure the frame buffer or initialize OpenGL. Frame buffer configuration is done outside of OpenGL in conjunction with the window system; OpenGL initialization takes place when the window system allocates a window for OpenGL rendering. (GLX, the X extension of the OpenGL interface, provides these capabilities, as described in ["OpenGL Extension to the X Window System"](GLX) )

**Basic OpenGL Operation**

The figure shown below gives an abstract, high-level block diagram of how OpenGL processes data. In the diagram, commands enter from the left and proceed through what can be thought of as a processing pipeline. Some commands specify geometric objects to be drawn, and others control how the objects are handled during the various processing stages.

Figure 1-1. OpenGL Block Diagram
As shown by the first block in the diagram, rather than having all commands proceed immediately through the pipeline, you can choose to accumulate some of them in a display list for processing at a later time.

The evaluator stage of processing provides an efficient means for approximating curve and surface geometry by evaluating polynomial commands of input values. During the next stage, per-vertex operations and primitive assembly, OpenGL processes geometric primitives—points, line segments, and polygons, all of which are described by vertices. Vertices are transformed and lit, and primitives are clipped to the viewport in preparation for the next stage.

Rasterization produces a series of frame buffer addresses and associated values using a two-dimensional description of a point, line segment, or polygon. Each fragment so produced is fed into the last stage, per-fragment operations, which performs the final operations on the data before it's stored as pixels in the frame buffer. These operations include conditional updates to the frame buffer based on incoming and previously stored z-values (for z-buffering) and blending of incoming pixel colors with stored colors, as well as masking and other logical operations on pixel values.

Input data can be in the form of pixels rather than vertices. Such data, which might describe an image for use in texture mapping, skips the first stage of processing described above and instead is processed as pixels, in the pixel operations stage. The result of this stage is either stored as texture memory, for use in the rasterization stage, or rasterized and the resulting fragments merged into the frame buffer just as if they were generated from geometric data.

All elements of OpenGL state, including the contents of the texture memory and even of the frame buffer, can be obtained by an OpenGL application.

**Chapter 2. Overview of Commands and Routines**
Many OpenGL commands pertain specifically to drawing objects such as points, lines, polygons, and bitmaps. Other commands control the way that some of this drawing occurs (such as those that enable antialiasing or texturing). Still other commands are specifically concerned with frame buffer manipulation. This chapter briefly describes how all the OpenGL commands work together to create the OpenGL processing pipeline. Brief overviews are also given of the routines comprising the OpenGL Utility Library (GLU) and the OpenGL extensions to the X Window System (GLX).

This chapter has the following main sections:

- "OpenGL Processing Pipeline" expands on the discussion in Chapter 1 by explaining how specific OpenGL commands control the processing of data.
- "Additional OpenGL Commands" discusses several sets of OpenGL commands not covered in the previous section.
- "OpenGL Utility Library" describes the GLU routines that are available.
- "OpenGL Extension to the X Window System" describes the GLX routines.

OpenGL Processing Pipeline

Now that you have a general idea of how OpenGL works from Chapter 1, let's take a closer look at the stages in which data is actually processed and tie these stages to OpenGL commands. The figure shown on the next page is a more detailed block diagram of the OpenGL processing pipeline.

For most of the pipeline, you can see three vertical arrows between the major stages. These arrows represent vertices and the two primary types of data that can be associated with vertices: color values and texture coordinates. Also note that vertices are assembled into primitives, then to fragments, and finally to pixels in the frame buffer. This progression is discussed in more detail in the following sections.

As you continue reading, be aware that we've taken some liberties with command names. Many OpenGL commands are simple variations of each other, differing mostly in the data type of arguments; some commands differ in the number of related arguments and whether those arguments can be specified as a vector or whether they must be specified separately in a list. For example, if you use the glVertex2f() command, you need to supply \( x \) and \( y \) coordinates as 32-bit floating-point numbers; with glVertex3sv(), you must supply an array of three short (16-bit) integer values for \( x \), \( y \), and \( z \). For simplicity, only the base name of the command is used in the discussion that follows, and an asterisk is included to indicate that there may be more to the actual command name than is being shown. For example, glVertex*() stands for all variations of the command you use to specify vertices.

Also keep in mind that the effect of an OpenGL command may vary depending on whether certain modes are enabled. For example, you need to enable lighting if the lighting-related commands are to have the desired effect of producing a properly lit object. To enable a particular mode, you use the glEnable() command and supply the
appropriate constant to identify the mode (for example, GL_LIGHTING). The following sections don't discuss specific modes, but you can refer to the reference page for glEnable() for a complete list of the modes that can be enabled. Modes are disabled with glDisable().

**Figure 2-1. OpenGL Pipeline**
This section relates the OpenGL commands that perform per-vertex operations to the processing stages shown in the figure on the previous page.

**Input Data**

You must provide several types of input data to the OpenGL pipeline:

- **Vertices**—Vertices describe the shape of the desired geometric object. To specify vertices, you use `glVertex*()` commands in conjunction with `glBegin()` and `glEnd()` to create a point, line, or polygon. You can also use `glRect*()` to describe an entire rectangle at once.
- **Edge flag**—By default, all edges of polygons are boundary edges. Use the `glEdgeFlag*()` command to explicitly set the edge flag.
- **Current raster position**—Specified with `glRasterPos*()`, the current raster position is used to determine raster coordinates for pixel and bitmap drawing operations.
- **Current normal**—A normal vector associated with a particular vertex determines how a surface at that vertex is oriented in three-dimensional space; this in turn affects how much light that particular vertex receives. Use `glNormal*()` to specify a normal vector.
- **Current color**—The color of a vertex, together with the lighting conditions, determine the final, lit color. Color is specified with `glColor*()` if in RGBA mode or with `glIndex*()` if in color index mode.
- **Current texture coordinates**—Specified with `glTexCoord*()`, texture coordinates determine the location in a texture map that should be associated with a vertex of an object.

When `glVertex*()` is called, the resulting vertex inherits the current edge flag, normal, color, and texture coordinates. Therefore, `glEdgeFlag*()`, `glNormal*()`, `glColor*()`, and `glTexCoord*()` must be called before `glVertex*()` if they are to affect the resulting vertex.

**Matrix Transformations**

Vertices and normals are transformed by the modelview and projection matrices before they're used to produce an image in the frame buffer. You can use commands such as `glMatrixMode()`, `glMultMatrix()`, `glRotate()`, `glTranslate()`, and `glScale()` to compose the desired transformations, or you can directly specify matrices with `glLoadMatrix()` and `glLoadIdentity()`. Use `glPushMatrix()` and `glPopMatrix()` to save and restore modelview and projection matrices on their respective stacks.

**Lighting and Coloring**

In addition to specifying colors and normal vectors, you may define the desired lighting conditions with `glLight*()` and `glLightModel*()`, and the desired material properties
with `glMaterial*()`.

Related commands you might use to control how lighting calculations are performed include `glShadeModel()`, `glFrontFace()`, and `glColorMaterial()`.

**Generating Texture Coordinates**

Rather than explicitly supplying texture coordinates, you can have OpenGL generate them as a function of other vertex data. This is what the `glTexGen*()` command does. After the texture coordinates have been specified or generated, they are transformed by the texture matrix. This matrix is controlled with the same commands mentioned earlier for matrix transformations.

**Primitive Assembly**

Once all these calculations have been performed, vertices are assembled into primitives—points, line segments, or polygons—together with the relevant edge flag, color, and texture information for each vertex.

**Primitives**

During the next stage of processing, primitives are converted to pixel fragments in several steps: primitives are clipped appropriately, whatever corresponding adjustments are necessary are made to the color and texture data, and the relevant coordinates are transformed to window coordinates. Finally, rasterization converts the clipped primitives to pixel fragments.

**Clipping**

Points, line segments, and polygons are handled slightly differently during clipping. Points are either retained in their original state (if they're inside the clip volume) or discarded (if they're outside). If portions of line segments or polygons are outside the clip volume, new vertices are generated at the clip points. For polygons, an entire edge may need to be constructed between such new vertices. For both line segments and polygons that are clipped, the edge flag, color, and texture information is assigned to all new vertices.

Clipping actually happens in two steps:

1. **Application-specific clipping**—Immediately after primitives are assembled, they're clipped in *eye coordinates* as necessary for any arbitrary clipping planes you've defined for your application with `glClipPlane()`. (OpenGL requires support for at least six such application-specific clipping planes.)

2. **View volume clipping**—Next, primitives are transformed by the projection matrix (into *clip coordinates*) and clipped by the corresponding viewing volume. This matrix can be controlled by the previously mentioned matrix transformation commands but is most typically specified by `glFrustum()` or `glOrtho()`.
Transforming to Window Coordinates

Before clip coordinates can be converted to window coordinates, they are normalized by dividing by the value of \( w \) to yield normalized device coordinates. After that, the viewport transformation applied to these normalized coordinates produces window coordinates. You control the viewport, which determines the area of the on-screen window that displays an image, with \texttt{glDepthRange()} and \texttt{glViewport()}.

Rasterization

Rasterization is the process by which a primitive is converted to a two-dimensional image. Each point of this image contains such information as color, depth, and texture data. Together, a point and its associated information are called a fragment. The current raster position (as specified with \texttt{glRasterPos*()}) is used in various ways during this stage for pixel drawing and bitmaps. As discussed below, different issues arise when rasterizing the three different types of primitives; in addition, pixel rectangles and bitmaps need to be rasterized.

Primitives. You control how primitives are rasterized with commands that allow you to choose dimensions and stipple patterns: \texttt{glPointSize()}, \texttt{glLineWidth()}, \texttt{glLineStipple()}, and \texttt{glPolygonStipple()}. Additionally, you can control how the front and back faces of polygons are rasterized with \texttt{glCullFace()}, \texttt{glFrontFace()}, and \texttt{glPolygonMode()}.

Pixels. Several commands control pixel storage and transfer modes. The command \texttt{glPixelStore*()} controls the encoding of pixels in client memory, and \texttt{glPixelTransfer*()} and \texttt{glPixelMap*()} control how pixels are processed before being placed in the frame buffer. A pixel rectangle is specified with \texttt{glDrawPixels()}; its rasterization is controlled with \texttt{glPixelZoom()}.

Bitmaps. Bitmaps are rectangles of zeros and ones specifying a particular pattern of fragments to be produced. Each of these fragments has the same associated data. A bitmap is specified using \texttt{glBitmap()}.

Texture Memory. Texturing maps a portion of a specified texture image onto each primitive when texturing is enabled. This mapping is accomplished by using the color of the texture image at the location indicated by a fragment's texture coordinates to modify the fragment's RGBA color. A texture image is specified using \texttt{glTexImage2D()} or \texttt{glTexImage1D()}. The commands \texttt{glTexParameter*()} and \texttt{glTexEnv*()} control how texture values are interpreted and applied to a fragment.

Fog. You can have OpenGL blend a fog color with a rasterized fragment's post-texturing color using a blending factor that depends on the distance between the eyepoint and the fragment. Use \texttt{glFog*()} to specify the fog color and blending factor.
Fragments

OpenGL allows a fragment produced by rasterization to modify the corresponding pixel in the frame buffer only if it passes a series of tests. If it does pass, the fragment's data can be used directly to replace the existing frame buffer values, or it can be combined with existing data in the frame buffer, depending on the state of certain modes.

Pixel Ownership Test

The first test is to determine whether the pixel in the frame buffer corresponding to a particular fragment is owned by the current OpenGL context. If so, the fragment proceeds to the next test. If not, the window system determines whether the fragment is discarded or whether any further fragment operations will be performed with that fragment. This test allows the window system to control OpenGL's behavior when, for example, an OpenGL window is obscured.

Scissor Test

With the `glScissor()` command, you can specify an arbitrary screen-aligned rectangle outside of which fragments will be discarded.

Alpha Test

The alpha test (which is performed only in RGBA mode) discards a fragment depending on the outcome of a comparison between the fragment's alpha value and a constant reference value. The comparison command and reference value are specified with `glAlphaFunc()`.

Stencil Test

The stencil test conditionally discards a fragment based on the outcome of a comparison between the value in the stencil buffer and a reference value. The command `glStencilFunc()` specifies the comparison command and the reference value. Whether the fragment passes or fails the stencil test, the value in the stencil buffer is modified according to the instructions specified with `glStencilOp()`.

Depth Buffer Test

The depth buffer test discards a fragment if a depth comparison fails; `glDepthFunc()` specifies the comparison command. The result of the depth comparison also affects the stencil buffer update value if stenciling is enabled.
Blending

Blending combines a fragment's R, G, B, and A values with those stored in the frame buffer at the corresponding location. The blending, which is performed only in RGBA mode, depends on the alpha value of the fragment and that of the corresponding currently stored pixel; it might also depend on the RGB values. You control blending with `glBlendFunc()`, which allows you to indicate the source and destination blending factors.

Dithering

If dithering is enabled, a dithering algorithm is applied to the fragment's color or color index value. This algorithm depends only on the fragment's value and its x and y window coordinates.

Logical Operations

Finally, a logical operation can be applied between the fragment and the value stored at the corresponding location in the frame buffer; the result replaces the current frame buffer value. You choose the desired logical operation with `glLogicOp()`. Logical operations are performed only on color indices, never on RGBA values.

Pixels

During the previous stage of the OpenGL pipeline, fragments are converted to pixels in the frame buffer. The frame buffer is actually organized into a set of logical buffers—the color, depth, stencil, and accumulation buffers. The color buffer itself consists of a front left, front right, back left, back right, and some number of auxiliary buffers. You can issue commands to control these buffers, and you can directly read or copy pixels from them. (Note that the particular OpenGL context you're using may not provide all of these buffers.)

Frame Buffer Operations

You can select into which buffer color values are written with `glDrawBuffer()`. In addition, four different commands are used to mask the writing of bits to each of the logical frame buffers after all per-fragment operations have been performed: `glIndexMask()`, `glColorMask()`, `glDepthMask()`, and `glStencilMask()`. The operation of the accumulation buffer is controlled with `glAccum()`. Finally, `glClear()` sets every pixel in a specified subset of the buffers to the value specified with `glClearColor()`, `glClearIndex()`, `glClearDepth()`, `glClearStencil()`, or `glClearAccum()`.
Reading or Copying Pixels

You can read pixels from the frame buffer into memory, encode them in various ways, and store the encoded result in memory with `glReadPixels()`. In addition, you can copy a rectangle of pixel values from one region of the frame buffer to another with `glCopyPixels()`. The command `glReadBuffer()` controls from which color buffer the pixels are read or copied.

Additional OpenGL Commands

This section briefly describes special groups of commands that weren't explicitly shown as part of OpenGL's processing pipeline. These commands accomplish such diverse tasks as evaluating polynomials, using display lists, and obtaining the values of OpenGL state variables.

Using Evaluators

OpenGL's evaluator commands allow you to use a polynomial mapping to produce vertices, normals, texture coordinates, and colors. These calculated values are then passed on to the pipeline as if they had been directly specified. The evaluator facility is also the basis for the NURBS (Non-Uniform Rational B-Spline) commands, which allow you to define curves and surfaces, as described later in this chapter under "OpenGL Utility Library."

The first step involved in using evaluators is to define the appropriate one- or two-dimensional polynomial mapping using `glMap*()`. The domain values for this map can then be specified and evaluated in one of two ways:

- By defining a series of evenly spaced domain values to be mapped using `glMapGrid*()` and then evaluating a rectangular subset of that grid with `glEvalMesh*()`. A single point of the grid can be evaluated using `glEvalPoint*()`.
- By explicitly specifying a desired domain value as an argument to `glEvalCoord*()`, which evaluates the maps at that value.

Performing Selection and Feedback

Selection, feedback, and rendering are mutually exclusive modes of operation. Rendering is the normal, default mode during which fragments are produced by rasterization; in selection and feedback modes, no fragments are produced and therefore no frame buffer modification occurs. In selection mode, you can determine which primitives would be drawn into some region of a window; in feedback mode, information about primitives that would be rasterized is fed back to the application. You select among these three modes with `glRenderMode()`.
Selection

Selection works by returning the current contents of the name stack, which is an array of integer-valued names. You assign the names and build the name stack within the modeling code that specifies the geometry of objects you want to draw. Then, in selection mode, whenever a primitive intersects the clip volume, a selection hit occurs. The hit record, which is written into the selection array you've supplied with `glSelectBuffer()`, contains information about the contents of the name stack at the time of the hit. (Note that `glSelectBuffer()` needs to be called before OpenGL is put into selection mode with `glRenderMode()`. Also, the entire contents of the name stack isn't guaranteed to be returned until `glRenderMode()` is called to take OpenGL out of selection mode.) You manipulate the name stack with `glInitNames()`, `glLoadName()`, `glPushName()`, and `glPopName()`. In addition, you might want to use an OpenGL Utility Library routine for selection, `gluPickMatrix()`, which is described later in this chapter under "OpenGL Utility Library."

Feedback

In feedback mode, each primitive that would be rasterized generates a block of values that is copied into the feedback array. You supply this array with `glFeedbackBuffer()`, which must be called before OpenGL is put into feedback mode. Each block of values begins with a code indicating the primitive type, followed by values that describe the primitive's vertices and associated data. Entries are also written for bitmaps and pixel rectangles. Values are not guaranteed to be written into the feedback array until `glRenderMode()` is called to take OpenGL out of feedback mode. You can use `glPassThrough()` to supply a marker that's returned in feedback mode as if it were a primitive.

Using Display Lists

A display list is simply a group of OpenGL commands that has been stored for subsequent execution. The `glNewList()` command begins the creation of a display list, and `glEndList()` ends it. With few exceptions, OpenGL commands called between `glNewList()` and `glEndList()` are appended to the display list, and optionally executed as well. (The reference page for `glNewList()` lists the commands that can't be stored and executed from within a display list.) To trigger the execution of a list or set of lists, use `glCallList()` or `glCallLists()` and supply the identifying number of a particular list or lists. You can manage the indices used to identify display lists with `glGenLists()`, `glListBase()`, and `glIsList()`. Finally, you can delete a set of display lists with `glDeleteLists()`.

Managing Modes and Execution
The effect of many OpenGL commands depends on whether a particular mode is in effect. You use `glEnable()` and `glDisable()` to set such modes and `glIsEnabled()` to determine whether a particular mode is set.

You can control the execution of previously issued OpenGL commands with `glFinish()`, which forces all such commands to complete, or `glFlush()`, which ensures that all such commands will be completed in a finite time.

A particular implementation of OpenGL may allow certain behaviors to be controlled with hints, by using the `glHint()` command. Possible behaviors are the quality of color and texture coordinate interpolation, the accuracy of fog calculations, and the sampling quality of antialiased points, lines, or polygons.

**Obtaining State Information**

OpenGL maintains numerous state variables that affect the behavior of many commands. Some of these variables have specialized query commands:

- `glGetLight()`
- `glGetMaterial()`
- `glGetClipPlane()`
- `glGetPolygonStipple()`
- `glGetTexEnv()`
- `glGetTexGen()`
- `glGetTexImage()`
- `glGetTexParameter()`
- `glGetMap()`
- `glGetPixelMap()`

The value of other state variables can be obtained with `glGetBooleanv()`, `glGetDoublev()`, `glGetFloatv()`, or `glGetIntegerv()`, as appropriate. The reference page for `glGet*()` explains how to use these commands. Other query commands you might want to use are `glGetError()`, `glGetString()`, and `glIsEnabled()` (See "Handling Errors" later in this chapter for more information about routines related to error handling.) Finally, you can save and restore sets of state variables with `glPushAttrib()` and `glPopAttrib()`.

**OpenGL Utility Library**

The OpenGL Utility Library (GLU) contains several groups of commands that complement the core OpenGL interface by providing support for auxiliary features. Since these utility routines make use of core OpenGL commands, any OpenGL implementation is guaranteed to support the utility routines. Note that the prefix for Utility Library routines is `glu` rather than `gl`. 
Manipulating Images for Use in Texturing

GLU provides image scaling and automatic mipmapming routines to simplify the specification of texture images. The routine `gluScaleImage()` scales a specified image to an accepted texture size; the resulting image can then be passed to OpenGL as a texture. The automatic mipmapming routines `gluBuild1DMipmaps()` and `gluBuild2DMipmaps()` create mipmapmed texture images from a specified image and pass them to `glTexImage1D()` and `glTexImage2D()`, respectively.

Transforming Coordinates

Several commonly used matrix transformation routines are provided. You can set up a two-dimensional orthographic viewing region with `gluOrtho2D()`, a perspective viewing volume using `gluPerspective()`, or a viewing volume that's centered on a specified eyepoint with `gluLookAt()`. Each of these routines creates the desired matrix and applies it to the current matrix using `glMultMatrix()`.

The `gluPickMatrix()` routine simplifies selection by creating a matrix that restricts drawing to a small region of the viewport. If you rerender the scene in selection mode after this matrix has been applied, all objects that would be drawn near the cursor will be selected and information about them stored in the selection buffer. See "Performing Selection and Feedback" earlier in this chapter for more information about selection mode.

If you need to determine where in the window an object is being drawn, use `gluProject()`, which converts specified coordinates from object coordinates to window coordinates; `gluUnProject()` performs the inverse conversion.

Polygon Tessellation

The polygon tessellation routines triangulate a concave polygon with one or more contours. To use this GLU feature, first create a tessellation object with `gluNewTess()`, and define callback routines that will be used to process the triangles generated by the tessellator (with `gluTessCallback()`). Then use `gluBeginPolygon()`, `gluTessVertex()`, `gluNextContour()`, and `gluEndPolygon()` to specify the concave polygon to be tessellated. Unneeded tessellation objects can be destroyed with `gluDeleteTess()`.

Rendering Spheres, Cylinders, and Disks

You can render spheres, cylinders, and disks using the GLU quadric routines. To do this, create a quadric object with `gluNewQuadric()`. (To destroy this object when you're finished with it, use `gluDeleteQuadric()`.) Then specify the desired rendering style, as listed below, with the appropriate routine (unless you're satisfied with the default values):
- Whether surface normals should be generated, and if so, whether there should be one normal per vertex or one normal per face: `gluQuadricNormals()`
- Whether texture coordinates should be generated: `gluQuadricTexture()`
- Which side of the quadric should be considered the outside and which the inside: `gluQuadricOrientation()`
- Whether the quadric should be drawn as a set of polygons, lines, or points: `gluQuadricDrawStyle()`

After you've specified the rendering style, simply invoke the rendering routine for the desired type of quadric object: `gluSphere()`, `gluCylinder()`, `gluDisk()`, or `gluPartialDisk()`. If an error occurs during rendering, the error-handling routine you've specified with `gluQuadricCallBack()` is invoked.

**NURBS Curves and Surfaces**

NURBS (Non-Uniform Rational B-Spline) curves and surfaces are converted to OpenGL evaluators by the routines described in this section. You can create and delete a NURBS object with `gluNewNurbsRenderer()` and `gluDeleteNurbsRenderer()`, and establish an error-handling routine with `gluNurbsCallback()`.

You specify the desired curves and surfaces with different sets of routines—`gluBeginCurve()`, `gluNurbsCurve()`, and `gluEndCurve()` for curves or `gluBeginSurface()`, `gluNurbsSurface()`, and `gluEndSurface()` for surfaces. You can also specify a trimming region, which defines a subset of the NURBS surface domain to be evaluated, thereby allowing you to create surfaces that have smooth boundaries or that contain holes. The trimming routines are `gluBeginTrim()`, `gluPwlCurve()`, `gluNurbsCurve()`, and `gluEndTrim()`.

As with quadric objects, you can control how NURBS curves and surfaces are rendered:

- Whether a curve or surface should be discarded if its control polyhedron lies outside the current viewport
- What the maximum length should be (in pixels) of edges of polygons used to render curves and surfaces
- Whether the projection matrix, modelview matrix, and viewport should be taken from the OpenGL server or whether you'll supply them explicitly with `gluLoadSamplingMatrices()`

Use `gluNurbsProperty()` to set these properties, or use the default values. You can query a NURBS object about its rendering style with `gluGetNurbsProperty()`.

**Handling Errors**

The routine `gluErrorString()` is provided for retrieving an error string that corresponds to an OpenGL or GLU error code. The currently defined OpenGL error codes are described in the `glGetError()` reference page. The GLU error codes are listed in the
gluErrorString(), gluTessCallback(), gluQuadricCallback(), and gluNurbsCallback() reference pages. Errors generated by GLX routines are listed in the relevant reference pages for those routines.

**OpenGL Extension to the X Window System**

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 OpenGL's rendering commands encapsulated within the X byte stream. Since performance is critical in three-dimensional rendering, the OpenGL extension to X allows OpenGL to bypass the X server's involvement in data encoding, copying, and interpretation and instead render directly to the graphics pipeline.

This section briefly discusses the routines defined as part of GLX; these routines have the prefix `glX`. You'll need to have some knowledge of X in order to fully understand the following and to use GLX successfully.

**Initialization**

Use `glXQueryExtension()` and `glXQueryVersion()` to determine whether the GLX extension is defined for an X server, and if so, which version is bound in the server. The `glXChooseVisual()` routine returns a pointer to an XVisualInfo structure describing the visual that best meets the client's specified attributes. You can query a visual about its support of a particular OpenGL attribute with `glXGetConfig()`.

**Controlling Rendering**

Several GLX routines are provided for creating and managing an OpenGL rendering context. You can use such a context to render off-screen if you want. Routines are also provided for such tasks as synchronizing execution between the X and OpenGL streams, swapping front and back buffers, and using an X font.

**Managing an OpenGL Rendering Context**

An OpenGL rendering context is created with `glXCreateContext()`. One of the arguments to this routine allows you to request a direct rendering context that bypasses the X server as described above. (Note that in order to do direct rendering, the X server connection must be local and the OpenGL implementation needs to support direct rendering.) You can determine whether a GLX context is direct with `glXIsDirect()`.

To make a rendering context current, use `glXMakeCurrent(); glXGetCurrentContext()` returns the current context. (You can also obtain the current drawable with `glXGetCurrentDrawable()`.) Remember that only one context can be current for any thread at any one time. If you have multiple contexts, you can copy
selected groups of OpenGL state variables from one context to another with
`glXCopyContext()`. When you're finished with a particular context, destroy it with
`glXDestroyContext()`.

**Off-Screen Rendering**

To render off-screen, first create an X Pixmap and then pass this as an argument to
`glXCreateGLXPixmap()`. Once rendering is completed, you can destroy the association
between the X and GLX Pixmaps with `glXDestroyGLXPixmap()`. (Off-screen rendering
isn't guaranteed to be supported for direct renderers.)

**Synchronizing Execution**

To prevent X requests from executing until any outstanding OpenGL rendering is
completed, call `glXWaitGL()`. Then, any previously issued OpenGL commands are
guaranteed to be executed before any X rendering calls made after `glXWaitGL()`. Although the same result can be achieved with `glFinish()`, `glXWaitGL()` doesn't
require a round trip to the server and thus is more efficient in cases where the client and
server are on separate machines.

To prevent an OpenGL command sequence from executing until any outstanding X
requests are completed, use `glXWaitX()`. This routine guarantees that previously issued
X rendering calls will be executed before any OpenGL calls made after `glXWaitX()`.

**Swapping Buffers**

For drawables that are double-buffered, the front and back buffers can be exchanged by
calling `glXSwapBuffers()`. An implicit `glFlush()` is done as part of this routine.

**Using an X Font**

A shortcut for using X fonts in OpenGL is provided with the command `glXUseXFont()`.

**Chapter 3. Summary of Commands and Routines**

This chapter lists the prototypes for OpenGL, the OpenGL Utility Library, and the
OpenGL extension to the X Window System. The prototypes are grouped functionally, as
shown below:

- **OpenGL Commands**
  - "Primitives"
  - "Coordinate Transformation"
  - "Coloring and Lighting"
  - "Clipping"
  - "Rasterization"
Notation

Since some of the OpenGL commands differ from each other only by the data type of the arguments they accept, certain conventions have been used to refer to these commands in a compact way:

```c
void glVertex2{sifd}{v} (TYPE x, TYPE y);
```

In this example, the first set of braces encloses characters identifying the possible data types for the arguments listed as having data type TYPE. (The digit preceding the braces indicates how many arguments the command takes.) In this case, all the arguments have the placeholder TYPE, but in other situations some arguments may have an explicitly defined data type. The table shown below lists the set of possible data types, their corresponding characters, and the type definition OpenGL uses for referring to that data type.

<table>
<thead>
<tr>
<th>character</th>
<th>data type</th>
<th>C-language type</th>
<th>OpenGL type definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>8-bit integer</td>
<td>signed char</td>
<td>GLbyte</td>
</tr>
<tr>
<td>s</td>
<td>16-bit integer</td>
<td>short</td>
<td>GLshort</td>
</tr>
<tr>
<td>i</td>
<td>32-bit integer</td>
<td>int</td>
<td>GLint, GLsizei</td>
</tr>
<tr>
<td>f</td>
<td>32-bit floating-point</td>
<td>float</td>
<td>GLfloat, GLclampf</td>
</tr>
<tr>
<td>d</td>
<td>64-bit floating-point</td>
<td>double</td>
<td>GLdouble, GLclampd</td>
</tr>
<tr>
<td>ub</td>
<td>8-bit unsigned integer</td>
<td>unsigned char</td>
<td>GLubyte, GLboolean</td>
</tr>
<tr>
<td>character</td>
<td>data type</td>
<td>C-language type</td>
<td>OpenGL type definition</td>
</tr>
<tr>
<td>-----------</td>
<td>----------------------------</td>
<td>-----------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>us</td>
<td>16-bit unsigned integer</td>
<td>unsigned short</td>
<td>GLushort</td>
</tr>
<tr>
<td>ui</td>
<td>32-bit unsigned integer</td>
<td>unsigned int</td>
<td>GLuint, GLenum, GLbitfield</td>
</tr>
<tr>
<td></td>
<td>void</td>
<td>GLvoid</td>
<td></td>
</tr>
</tbody>
</table>

The second set of braces, if present, contains a \( v \) for the vector form of the command. If you choose to use the vector form, all the TYPE arguments are collapsed into a single array. For example, here are the nonvector and vector forms of a command, using a 32-bit floating-point data type:

```c
void glVertex2f(GLfloat x, GLfloat y);
void glVertex2fv(GLfloat v[2]);
```

Where the use of the vector form is ambiguous, both the vector and nonvector forms are listed. Note that not all commands with multiple arguments have a vector form and that some commands have only a vector form, in which case the \( v \) isn't enclosed in braces.

### OpenGL Commands

#### Primitives

Specify vertices or rectangles:

```c
void glBegin (GLenum mode);
void glEnd (void);
void glVertex2{sifd}{v}(TYPE x, TYPE y);
void glVertex3{sifd}{v}(TYPE x, TYPE y, TYPE z);
void glVertex4{sifd}{v}(TYPE x, TYPE y, TYPE z, TYPE w);
void glRect{sifd} (TYPE x1, TYPE y1, TYPE x2, TYPE y2);
void glRect{sifd}v (const TYPE *v1, const TYPE *v2);
```

Specify polygon edge treatment:

```c
void glEdgeFlag (GLboolean flag);
void glEdgeFlagv (const GLboolean *flag);
```

### Coordinate Transformation

Transform the current matrix:

```c
void glRotate{fd} (TYPE angle, TYPE x, TYPE y, TYPE z);
void glTranslate{fd} (TYPE x, TYPE y, TYPE z);
void glScale{fd} (TYPE x, TYPE y, TYPE z);
void glMultMatrix{fd} (const TYPE *m);
```
void glFrustum (GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);
void glOrtho (GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);

Replace the current matrix:

void glLoadIdentity (void);

Manipulate the matrix stack:

void glMatrixMode (GLenum mode);
void glPushMatrix (void);
void glPopMatrix (void);

Specify the viewport:

void glDepthRange (GLclampd near, GLclampd far);
void glViewport (GLint x, GLint y, GLsizei width, GLsizei height);

**Coloring and Lighting**

Set the current color, color index, or normal vector:

void glColor3{bsifd ubusui}{v} (TYPE red, TYPE green, TYPE blue);
void glColor4{bsifd ubusui}{v} (TYPE red, TYPE green, TYPE blue, TYPE alpha);
void glIndex{sifd}{v} (TYPE index);
void glNormal3{bsifd}{v} (TYPE nx, TYPE ny, TYPE nz);

Specify light source, material, or lighting model parameter values:

void glLight{if}{v} (GLenum light, GLenum pname, TYPE param);
void glMaterial{if}{v} (GLenum face, GLenum pname, TYPE param);
void glLightModel{if}{v} (GLenum pname, TYPE param);

Choose a shading model:

void glShadeModel (GLenum mode);

Specify which polygon orientation is front-facing:

void glFrontFace (GLenum dir);

Cause a material color to track the current color:
void glColorMaterial(GLenum face, GLenum mode);

Obtain light source or material parameter values:

void glGetLight{if}v(GLenum light, GLenum pname, TYPE *params);
void glGetMaterial{if}v(GLenum face, GLenum pname, TYPE *params);

Clipping

Specify a clipping plane:

void glClipPlane(GLenum plane, const GLdouble *equation);

Return clipping plane coefficients:

void glGetClipPlane(GLenum plane, GLdouble *equation);

Rasterization

Set the current raster position:

void glRasterPos2{sifd}{v}(TYPE x, TYPE y);
void glRasterPos3{sifd}{v}(TYPE x, TYPE y, TYPE z);

void glRasterPos4{sifd}{v}(TYPE x, TYPE y, TYPE z, TYPE w);

Specify a bitmap:

void glBitmap(GLsizei width, GLsizei height, GLfloat xorig, GLfloat yorig, GLfloat xmove, GLfloat ymove, const GLubyte *bitmap);

Specify the dimensions of points or lines:

void glPointSize(GLfloat size);
void glLineWidth(GLfloat width);

Specify or return a stipple pattern for lines or polygons:

void glLineStipple(GLint factor, GLushort pattern);
void glPolygonStipple(const GLubyte *mask);
void glGetPolygonStipple(GLubyte *mask);

Choose how polygons are rasterized:
void glCullFace (GLenum mode);
void glPolygonMode (GLenum face, GLenum mode);

**Pixel Operations**

Select the source for pixel reads or copies:

void glReadBuffer (GLenum mode);

Read, write, and copy pixels:

void glReadPixels (GLint x, GLint y, GLsizei width, GLsizei height, GLenum format, GLenum type, GLvoid *pixels);
void glDrawPixels (GLsizei width, GLsizei height, GLenum format, GLenum type, const GLvoid *pixels);
void glCopyPixels (GLint x, GLint y, GLsizei width, GLsizei height, GLenum type);

Specify or query how pixels are encoded or processed:

void glPixelStore{if} (GLenum pname, TYPE param);
void glPixelTransfer{if} (GLenum pname, TYPE param);
void glPixelMap{f usui}v (GLenum map, GLint mapsize, const TYPE *values);
void glGetPixelMap{f usui}v (GLenum map, TYPE *values);

Control pixel rasterization:

void glPixelZoom (GLfloat xfactor, GLfloat yfactor);

**Texture Mapping**

Control how a texture is applied to a fragment:

void glTexParameteri{if}{v} (GLenum target, GLenum pname, TYPE param);
void glTexEnv{if}{v} (GLenum target, GLenum pname, TYPE param);

Set the current texture coordinates:

void glTexCoord1{sifd}{v} (TYPE s);
void glTexCoord2{sifd}{v} (TYPE s, TYPE t);
void glTexCoord3{sifd}{v} (TYPE s, TYPE t, TYPE r);
void glTexCoord4{sifd}{v} (TYPE s, TYPE t, TYPE r, TYPE q);

Control the generation of texture coordinates:

void glTexGen{ifd}{v} (GLenum coord, GLenum pname, TYPE param);
Specify a one- or two-dimensional texture image:

```c
void glTexImage1D (GLenum target, GLint level, GLint components, GLsizei width, GLint border, GLenum format, GLenum type, const GLvoid *pixels);
void glTexImage2D (GLenum target, GLint level, GLint components, GLsizei width, GLsizei height, GLint border, GLenum format, GLenum type, const GLvoid *pixels);
```

Obtain texture-related parameter values:

```c
void glGetTexEnv{if}v (GLenum target, GLenum pname, TYPE *params);
void glGetTexGen{ifd}v (GLenum coord, GLenum pname, TYPE *params);
void glGetTexImage (GLenum target, GLint level, GLenum format, GLenum type, GLvoid *pixels);
void glGetTexLevelParameter{if}v (GLenum target, GLint level, GLenum pname, TYPE *params);
void glGetTexParameter{if}v (GLenum target, GLenum pname, TYPE *params);
```

**Fog**

Set fog parameters:

```c
void glFog{if}{v} (GLenum pname, TYPE param);
```

**Frame Buffer Operations**

Control per-fragment testing:

```c
void glScissor (GLint x, GLint y, GLsizei width, GLsizei height);
void glAlphaFunc (GLenum func, GLclampf ref);
void glStencilFunc (GLenum func, GLint ref, GLuint mask);
void glStencilOp (GLenum fail, GLenum pass, GLenum zpass);
void glDepthFunc (GLenum func);
```

Combine fragment and frame buffer values:

```c
void glBlendFunc (GLenum sfactor, GLenum dfactor);
void glLogicOp (GLenum opcode);
```

Clear some or all buffers:

```c
void glClear (GLbitfield mask);
```

Specify color, depth, and stencil values for clears:
void glClearAccum (GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha);
void glClearColor (GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);
void glClearDepth (GLclampd depth);
void glClearIndex (GLfloat c);
void glClearStencil (GLint s);

Control buffers enabled for writing:

void glDrawBuffer (GLenum mode);
void glIndexMask (GLuint mask);
void glColorMask (GLboolean red, GLboolean green, GLboolean blue, GLboolean alpha);
void glDepthMask (GLboolean flag);
void glStencilMask (GLuint mask);

Operate on the accumulation buffer:

void glAccum (GLenum op, GLfloat value);

**Evaluators**

Define a one- or two-dimensional evaluator:

void glMap1{fd} (GLenum target, TYPE u1, TYPE u2, GLint stride, GLint order, const TYPE *points);
void glMap2{fd} (GLenum target, TYPE u1, TYPE u2, GLint ustride, GLint uorder, TYPE v1, TYPE v2, GLint vstride, GLInt vorder, const TYPE *points);

Generate and evaluate a series of map domain values:

void glMapGrid1{fd} (GLint n, TYPE u1, TYPE u2);
void glMapGrid2{fd} (GLInt un, TYPE u1, TYPE u2, GLInt vn, TYPE v1, TYPE v2);
void glEvalMesh1 (GLenum mode, GLInt i1, GLInt i2);
void glEvalMesh2 (GLenum mode, GLInt i1, GLInt i2, GLInt j1, GLInt j2);
void glEvalPoint1 (GLInt i);
void glEvalPoint2 (GLInt i, GLInt j);

Evaluate one- and two-dimensional maps at a specified domain coordinate:

void glEvalCoord1{fd}{v} (TYPE u);
void glEvalCoord2{fd}{v} (TYPE u, TYPE v);

Obtain evaluator parameter values:
void glGetMap(idfv (GLenum target, GLenum query, TYPE *v);

**Selection and Feedback**

Control the mode and corresponding buffer:

GLint glRenderMode (GLenum mode);

void glSelectBuffer (GLsizei size, GLuint *buffer);
void glFeedbackBuffer (GLsizei size, GLenum type, GLfloat *buffer);

Supply a token for feedback mode:

void glPassThrough (GLfloat token);

Control the name stack for selection:

void glInitNames (void);
void glLoadName (GLuint name);
void glPushName (GLuint name);
void glPopName (void);

**Display Lists**

Create or delete display lists:

void glNewList (GLuint list, GLenum mode);
void glEndList (void);
void glDeleteLists (GLuint list, GLsizei range);

Execute a display list or set of lists:

void glCallList (GLuint list);
void glCallLists (GLsizei n, GLenum type, const GLvoid *lists);

Manage display-list indices:

GLuint glGenLists (GLsizei range);
GLboolean glIsList (GLuint list);
void glListBase (GLuint base);

**Modes and Execution**

Enable, disable, and query modes:
void glEnable (GLenum cap);
void glDisable (GLenum cap);
GLboolean glIsEnabled (GLenum cap);

Wait until all OpenGL commands have executed completely:
void glFinish (void);

Force all issued OpenGL commands to be executed:
void glFlush (void);

Specify hints for OpenGL operation:
void glHint (GLenum target, GLenum mode);

State Queries
Obtain information about an error or the current OpenGL connection:
GLenum glGetError (void);
const GLubyte * glGetString (GLenum name);

Query state variables:
void glGetBooleanv (GLenum pname, GLboolean *params);
void glGetDoublev (GLenum pname, GLdouble *params);
void glGetFloatv (GLenum pname, GLfloat *params);
void glGetIntegerv (GLenum pname, GLint *params);

Save and restore sets of state variables:
void glPushAttrib (GLbitfield mask);
void glPopAttrib (void);

GLU Routines
Texture Images
Magnify or shrink an image:
int gluScaleImage (GLenum format, GLint widthin, GLint heightin, GLenum typein, const void *datain, GLint widthout, GLint heightout, GLenum typeout, void *dataout);
Generate mipmaps for an image:

```c
int gluBuild1DMipmaps (GLenum target, GLint components, GLint width, GLenum format, GLenum type, void *data);
int gluBuild2DMipmaps (GLenum target, GLint components, GLint width, GLint height, GLenum format, GLenum type, void *data);
```

### Coordinate Transformation

Create projection or viewing matrices:

```c
void gluOrtho2D (GLdouble left, GLdouble right, GLdouble bottom, GLdouble top);
void gluPerspective (GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar);
void gluPickMatrix (GLdouble x, GLdouble y, GLdouble width, GLdouble height, GLint viewport[4]);
void gluLookAt (GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz);
```

Convert object coordinates to screen coordinates:

```c
int gluProject (GLdouble objx, GLdouble objy, GLdouble objz, const GLdouble modelMatrix[16], const GLdouble projMatrix[16], const GLint viewport[4], GLdouble *winx, GLdouble *winy, GLdouble *winz);
int gluUnProject (GLdouble winx, GLdouble winy, GLdouble winz, const GLdouble modelMatrix[16], const GLdouble projMatrix[16], const GLint viewport[4], GLdouble *objx, GLdouble *objy, GLdouble *objz);
```

### Polygon Tessellation

Manage tessellation objects:

```c
GLUtriangulatorObj* gluNewTess (void);
void gluTessCallback (GLUtriangulatorObj *tobj, GLenum which, void (*fn)());
void gluDeleteTess (GLUtriangulatorObj *tobj);
```

Describe the input polygon:

```c
void gluBeginPolygon (GLUtriangulatorObj *tobj);
void gluEndPolygon (GLUtriangulatorObj *tobj);
void gluNextContour (GLUtriangulatorObj *tobj, GLenum type);
void gluTessVertex (GLUtriangulatorObj *tobj, GLdouble v[3], void *data);
```
Quadric Objects

Manage quadric objects:

GLUquadricObj* gluNewQuadric (void);
void gluDeleteQuadric (GLUquadricObj *state);
void gluQuadricCallback (GLUquadricObj *qobj, GLenum which, void (*fn)());

Control the rendering:

void gluQuadricNormals (GLUquadricObj *quadObject, GLenum normals);
void gluQuadricTexture (GLUquadricObj *quadObject, GLboolean textureCoords);
void gluQuadricOrientation (GLUquadricObj *quadObject, GLenum orientation);
void gluQuadricDrawStyle (GLUquadricObj *quadObject, GLenum drawStyle);

Specify a quadric primitive:

void gluCylinder (GLUquadricObj *qobj, GLdouble baseRadius, GLdouble topRadius, GLdouble height, GLint slices, GLint stacks);
void gluDisk (GLUquadricObj *qobj, GLdouble innerRadius, GLdouble outerRadius, GLint slices, GLint loops);
void gluPartialDisk (GLUquadricObj *qobj, GLdouble innerRadius, GLdouble outerRadius, GLint slices, GLint loops, GLdouble startAngle, GLdouble sweepAngle);
void gluSphere (GLUquadricObj *qobj, GLdouble radius, GLint slices, GLint stacks);

NURBS Curves and Surfaces

Manage a NURBS object:

GLUnurbsObj* gluNewNurbsRenderer (void);
void gluDeleteNurbsRenderer (GLUnurbsObj *nobj);
void gluNurbsCallback (GLUnurbsObj *nobj, GLenum which, void (*fn)());

Create a NURBS curve:

void gluBeginCurve (GLUnurbsObj *nobj);
void gluEndCurve (GLUnurbsObj *nobj);
void gluNurbsCurve (GLUnurbsObj *nobj, GLint nknots, GLfloat *knot, GLint stride, GLfloat *ctlarray, GLint order, GLenum type);

Create a NURBS surface:
void gluBeginSurface (GLUnurbsObj *nobj); void gluEndSurface (GLUnurbsObj *nobj);
void gluNurbsSurface (GLUnurbsObj *nobj, GLint uknot_count, GLfloat *uknot, GLint vknot_count, GLfloat *vknot, GLint u_stride, GLint v_stride, GLfloat *ctlarray, GLint sorder, GLint torder, GLenum type);

Define a trimming region:

void gluBeginTrim (GLUnurbsObj *nobj);
void gluEndTrim (GLUnurbsObj *nobj);
void gluPwlCurve (GLUnurbsObj *nobj, GLint count, GLfloat *array, GLint stride, GLenum type);

Control NURBS rendering:

void gluLoadSamplingMatrices (GLUnurbsObj *nobj, const GLfloat modelMatrix[16], const GLfloat projMatrix[16], const GLint viewport[4]);
void gluNurbsProperty (GLUnurbsObj *nobj, GLenum property, GLfloat value);
void gluGetNurbsProperty (GLUnurbsObj *nobj, GLenum property, GLfloat *value);

Error Handling

Produce an error string from an OpenGL error code:

const GLubyte* gluErrorString (GLenum errorCode);

GLX Routines

Initialization

Determine whether the GLX extension is defined on the X server:

Bool glXQueryExtension (Display *dpy, int *errorBase, int *eventBase);
Bool glXQueryVersion (Display *dpy, int *major, int *minor);

Obtain the desired visual:

XVisualInfo* glXChooseVisual (Display *dpy, int screen, int *attribList);
int glXGetConfig (Display *dpy, XVisualInfo *vis, int attrib, int *value);

Controlling Rendering

Manage or query an OpenGL rendering context:
GLXContext glXCreateContext (Display *dpy, XVisualInfo *vis, GLXContext shareList, Bool direct);
void glXDestroyContext (Display *dpy, GLXContext ctx);
void glXCopyContext (Display *dpy, GLXContext src, GLXContext dst, GLuint mask);
Bool glXIsDirect (Display *dpy, GLXContext ctx);
Bool glXMakeCurrent (Display *dpy, GLXDrawable draw, GLXContext ctx);
GLXContext glXGetCurrentContext (void);
GLXDrawable glXGetCurrentDrawable (void);

Perform off-screen rendering:

GLXPixmap glXCreateGLXPixmap (Display *dpy, XVisualInfo *vis, Pixmap pixmap);
void glXDestroyGLXPixmap (Display *dpy, GLXPixmap pix);

Synchronize execution:

void glXWaitGL (void);
void glXWaitX (void);

Exchange front and back buffers:

void glXSwapBuffers (Display *dpy, Window window);

Use an X font:

void glXUseXFont (Font font, int first, int count, int listBase);

Chapter 4. Defined Constants and Associated Commands

This chapter lists all the defined constants in OpenGL and their corresponding commands; these constants might indicate a parameter name, a value for a parameter, a mode, a query target, or a return value. The list is intended to be used as another index into the reference pages: if you remember the name of a constant, you can use this table to find out which functions use it, and then you can refer to the reference pages for those functions for more information. Note that all the constants listed can be used directly by the corresponding commands; the reference pages list additional, related commands that might be of interest.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Associated Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_2D, GL_3D, GL_3D_COLOR,</td>
<td>glFeedbackBuffer()</td>
</tr>
<tr>
<td>Constant</td>
<td>Associated Commands</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------</td>
</tr>
<tr>
<td>GL_COLOR_TEXTURE, GL_4D_COLOR_TEXTURE</td>
<td>glCallLists()</td>
</tr>
<tr>
<td>GL_2_BYTES, GL_3_BYTES, GL_4_BYTES</td>
<td>glCallLists()</td>
</tr>
<tr>
<td>GL_ACCUM</td>
<td>glAccum()</td>
</tr>
<tr>
<td>GL_ACCUM_ALPHA_BITS, GL_ACCUM_BLUE_BITS</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_ACCUM_CLEAR_BIT</td>
<td>glClear(), glPushAttrib()</td>
</tr>
<tr>
<td>GL_ACCUM_CLEAR_VALUE, GL_ACCUM_GREEN_BITS, GL_ACCUM_RED_BITS</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_ADD</td>
<td>glAccum()</td>
</tr>
<tr>
<td>GL_ALL_ATTRIB_BITS</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_ALPHA</td>
<td>glDrawPixels(), glReadPixels(), glTexImage1D(), glTexImage2D(), glGetTexImage()</td>
</tr>
<tr>
<td>GL_ALPHA_BIAS</td>
<td>glPixelTransfer*(), glGet*()</td>
</tr>
<tr>
<td>GL_ALPHA_BITS</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_ALPHA_SCALE</td>
<td>glPixelTransfer*(), glGet*()</td>
</tr>
<tr>
<td>GL_ALPHA_TEST</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_ALPHA_TEST_FUNC, GL_ALPHA_TEST_REF</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_ALWAYS</td>
<td>glAlphaFunc(), glDepthFunc(), glStencilFunc()</td>
</tr>
<tr>
<td>GL_AMBIENT</td>
<td>glLight*(), glGetLight*(), glMaterial*(), glGetMaterial*(), glColorMaterial()</td>
</tr>
<tr>
<td>GL_AMBIENT_AND_DIFFUSE</td>
<td>glMaterial*(), glGetMaterial*(), glColorMaterial()</td>
</tr>
<tr>
<td>GL_AND, GL_AND_INVERTED, GL_AND_REVERSE</td>
<td>glLogicOp()</td>
</tr>
<tr>
<td>GL_ATTRIB_STACK_DEPTH</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_AUTO_NORMAL</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_AUX0 through GL_AUX3</td>
<td>glDrawBuffer(), glReadBuffer()</td>
</tr>
<tr>
<td>Constant</td>
<td>Associated Commands</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>-------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>GL_AUX_BUFFERS</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_BACK</td>
<td>glColorMaterial(), glCullFace(), glDrawBuffer(), glReadBuffer(), glMaterial*, glGetMaterial*, glPolygonMode()</td>
</tr>
<tr>
<td>GL_BACK_LEFT, GL_BACK_RIGHT</td>
<td>glDrawBuffer(), glReadBuffer()</td>
</tr>
<tr>
<td>GL_BITMAP</td>
<td>glDrawPixels(), glReadPixels(), glTexImage1D(), glTexImage2D(), glGetTexImage()</td>
</tr>
<tr>
<td>GL_BITMAP_TOKEN</td>
<td>glPassThrough()</td>
</tr>
<tr>
<td>GL_BLEND</td>
<td>glTexEnv*, glGetTexEnv*, glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_BLEND_DST, GL_BLEND_SRC</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_BLUE</td>
<td>glDrawPixels(), glReadPixels(), glTexImage1D(), glTexImage2D(), glGetTexImage()</td>
</tr>
<tr>
<td>GL_BLUE_BIAS</td>
<td>glVertexTransfer*, glGet*()</td>
</tr>
<tr>
<td>GL_BLUE_BITS</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_BLUE_SCALE</td>
<td>glVertexTransfer*, glGet*()</td>
</tr>
<tr>
<td>GL_BYTE</td>
<td>glCallLists(), glDrawPixels(), glReadPixels(), glTexImage1D(), glTexImage2D(), glGetTexImage()</td>
</tr>
<tr>
<td>GL_CCW</td>
<td>glFrontFace()</td>
</tr>
<tr>
<td>GL_CLAMP</td>
<td>glTexParameter*()</td>
</tr>
<tr>
<td>GL_CLEAR</td>
<td>glLogicOp()</td>
</tr>
<tr>
<td>GL_CLIP_PLANE</td>
<td>glEnable(), glIsEnabled()</td>
</tr>
<tr>
<td>GL_CLIP_PLANE0 through GL_CLIP_PLANE5</td>
<td>glClipPlane(), glGetClipPlane(), glEnable(), glIsEnabled()</td>
</tr>
<tr>
<td>GL_COEFF</td>
<td>glGetMap*()</td>
</tr>
<tr>
<td>Constant</td>
<td>Associated Commands</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>GL_COLOR</td>
<td>glCopyPixels()</td>
</tr>
<tr>
<td>GL_COLOR_BUFFER_BIT</td>
<td>glClear(), glPushAttrib()</td>
</tr>
<tr>
<td>GL_COLOR_CLEAR_VALUE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_COLOR_INDEX</td>
<td>glDrawPixels(), glReadPixels(), glTexImage1D(), glTexImage2D(), glGetTexImage()</td>
</tr>
<tr>
<td>GL_COLOR_INDEXES</td>
<td>glMaterial*(), glGetMaterial*()</td>
</tr>
<tr>
<td>GL_COLOR_MATERIAL</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_COLOR_MATERIAL_FACE,</td>
<td></td>
</tr>
<tr>
<td>GL_COLOR_MATERIAL_PARAMETER</td>
<td></td>
</tr>
<tr>
<td>GL_COLOR_WRITEMASK</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_COMPILE, GL_COMPILE_AND_EXECUTE</td>
<td>glNewList()</td>
</tr>
<tr>
<td>GL_CONSTANT_ATTENUATION</td>
<td>glLight*(), glGetLight*()</td>
</tr>
<tr>
<td>GL_COPY, GL_COPY_INVERTED</td>
<td>glLogicOp()</td>
</tr>
<tr>
<td>GL_COPY_PIXEL_TOKEN</td>
<td>glPassThrough()</td>
</tr>
<tr>
<td>GL_CULL_FACE</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_CULL_FACE_MODE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_CURRENT_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_CURRENT_COLOR, GL_CURRENT_INDEX,</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_CURRENT_NORMAL,</td>
<td></td>
</tr>
<tr>
<td>GL_CURRENT_RASTER_COLOR,</td>
<td></td>
</tr>
<tr>
<td>GL_CURRENT_RASTER_INDEX,</td>
<td></td>
</tr>
<tr>
<td>GL_CURRENT_RASTER_POSITION,</td>
<td></td>
</tr>
<tr>
<td>GL_CURRENT_RASTER_POSITION_VALID,</td>
<td></td>
</tr>
<tr>
<td>GL_CURRENT_RASTER_TEXTURE_COORDS,</td>
<td></td>
</tr>
<tr>
<td>GL_CURRENT_TEXTURE_COORDS</td>
<td></td>
</tr>
<tr>
<td>GL_CW</td>
<td>glFrontFace()</td>
</tr>
<tr>
<td>GL_DECAL</td>
<td>glTexEnv*(), glGetTexEnv*()</td>
</tr>
<tr>
<td>GL_DECR</td>
<td>glStencilOp()</td>
</tr>
<tr>
<td>GL_DEPTH</td>
<td>glCopyPixels()</td>
</tr>
<tr>
<td>GL_DEPTH_BIAS</td>
<td>glPixelTransfer*(), glGet*()</td>
</tr>
<tr>
<td>GL_DEPTH_BITS</td>
<td>glGet*()</td>
</tr>
<tr>
<td>Constant</td>
<td>Associated Commands</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>GL_DEPTH_BUFFER_BIT</td>
<td>glClear(), glPushAttrib()</td>
</tr>
<tr>
<td>GL_DEPTH_CLEAR_VALUE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_DEPTH_COMPONENT</td>
<td>glDrawPixels(), glReadPixels(),</td>
</tr>
<tr>
<td></td>
<td>glTexImage1D(), glTexImage2D(),</td>
</tr>
<tr>
<td></td>
<td>glGetTexImage()</td>
</tr>
<tr>
<td>GL_DEPTH_FUNC</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_DEPTH_RANGE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_DEPTH_SCALE</td>
<td>glPixelTransfer*, glGet*()</td>
</tr>
<tr>
<td>GL_DEPTH_TEST</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_DEPTH_WRITEMASK</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_DIFFUSE</td>
<td>glLight*, glGetLight*, glMaterial*,</td>
</tr>
<tr>
<td></td>
<td>glGetMaterial*, glColorMaterial()</td>
</tr>
<tr>
<td>GL_DITHER</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_DOMAIN</td>
<td>glGetMap*()</td>
</tr>
<tr>
<td>GL_DONT_CARE</td>
<td>glHint()</td>
</tr>
<tr>
<td>GL_DOUBLEBUFFER</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_DRAW_BUFFER</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_DRAW_PIXEL_TOKEN</td>
<td>glPassThrough()</td>
</tr>
<tr>
<td>GL_DST_ALPHA, GL_DST_COLOR</td>
<td>glBlendFunc()</td>
</tr>
<tr>
<td>GL_EDGE_FLAG</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_EMISSION</td>
<td>glMaterial*, glGetMaterial*,</td>
</tr>
<tr>
<td></td>
<td>glColorMaterial()</td>
</tr>
<tr>
<td>GL_ENABLE_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_EQUAL</td>
<td>glAlphaFunc(), glDepthFunc(),</td>
</tr>
<tr>
<td></td>
<td>glStencilFunc()</td>
</tr>
<tr>
<td>GL_EQUIV</td>
<td>glLogicOp()</td>
</tr>
<tr>
<td>GL_EVAL_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_EXP, GL_EXP2</td>
<td>glFog*()</td>
</tr>
<tr>
<td>GL_EXTENSIONS</td>
<td>glGetString()</td>
</tr>
<tr>
<td>GL_EYE_LINEAR</td>
<td>glTexImage*, glGetTexImage*()</td>
</tr>
<tr>
<td>Constant</td>
<td>Associated Commands</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>GL_EYE_PLANE</td>
<td>glTexGen*()</td>
</tr>
<tr>
<td>GL_FALSE</td>
<td>glColorMask(), glGet*(), glIsEnabled(), glIsList()</td>
</tr>
<tr>
<td>GL_FASTEST</td>
<td>glHint()</td>
</tr>
<tr>
<td>GL_FEEDBACK</td>
<td>glRenderMode()</td>
</tr>
<tr>
<td>GL_FILL</td>
<td>glPolygonMode(), glEvalMesh2()</td>
</tr>
<tr>
<td>GL_FLAT</td>
<td>glShadeModel()</td>
</tr>
<tr>
<td>GL_FLOAT</td>
<td>glCallLists(), glDrawPixels(), glReadPixels(), glTexImage1D(), glTexImage2D(), glGetTexImage()</td>
</tr>
<tr>
<td>GL_FOG</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_FOG_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_FOG_COLOR, GL_FOG_DENSITY, GL_FOG_END</td>
<td>glFog*(), glGet*()</td>
</tr>
<tr>
<td>GL_FOG_HINT</td>
<td>glHint()</td>
</tr>
<tr>
<td>GL_FOG_INDEX, GL_FOG_MODE, GL_FOG_START</td>
<td>glFog*(), glGet*()</td>
</tr>
<tr>
<td>GL_FRONT</td>
<td>glColorMaterial(), glCullFace(), glDrawBuffer(), glReadBuffer(), glMaterial*, glGetMaterial*, glPolygonMode()</td>
</tr>
<tr>
<td>GL_FRONT_AND_BACK</td>
<td>glColorMaterial(), glDrawBuffer(), glMaterial*, glPolygonMode()</td>
</tr>
<tr>
<td>GL_FRONT_FACE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_FRONT_LEFT, GL_FRONT_RIGHT</td>
<td>glDrawBuffer(), glReadBuffer()</td>
</tr>
<tr>
<td>GL_GEQUAL, GL_GREATER</td>
<td>glDepthFunc(), glAlphaFunc(), glStencilFunc()</td>
</tr>
<tr>
<td>GL_GREEN</td>
<td>glDrawPixels(), glReadPixels(), glTexImage1D(), glTexImage2D(), glGetTexImage()</td>
</tr>
<tr>
<td>Constant</td>
<td>Associated Commands</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>GL_GREEN_BIAS</td>
<td>glPixelTransfer*(), glGet*()</td>
</tr>
<tr>
<td>GL_GREEN_BITS</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_GREEN_SCALE</td>
<td>glPixelTransfer*(), glGet*()</td>
</tr>
<tr>
<td>GL_HINT_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_INCR</td>
<td>glStencilOp()</td>
</tr>
<tr>
<td>GL_INDEX_BITS, GL_INDEX_CLEAR_VALUE, GL_INDEX_MODE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_INDEX_OFFSET, GL_INDEX_SHIFT</td>
<td>glPixelTransfer*(), glGet*()</td>
</tr>
<tr>
<td>GL_INDEX_WRITEMASK</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_INT</td>
<td>glCallLists(), glDrawPixels(), glReadPixels(), glTexImage1D(), glTexImage2D(), glGetTexImage()</td>
</tr>
<tr>
<td>GL_INVALID_ENUM, GL_INVALID_OPERATION, GL_INVALID_VALUE</td>
<td>glGetError()</td>
</tr>
<tr>
<td>GL_INVERT</td>
<td>glLogicOp(), glStencilOp()</td>
</tr>
<tr>
<td>GL_KEEP</td>
<td>glStencilOp()</td>
</tr>
<tr>
<td>GL_LEFT</td>
<td>glDrawBuffer(), glReadBuffer()</td>
</tr>
<tr>
<td>GL_LEQUAL, GL_LESS</td>
<td>glDepthFunc(), glAlphaFunc(), glStencilFunc()</td>
</tr>
<tr>
<td>GL_LIGHT0 through GL_LIGHT7</td>
<td>glLight*(), glGetLight*(), glEnable(), glIsEnabled()</td>
</tr>
<tr>
<td>GL_LIGHTING</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_LIGHTING_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_LIGHT_MODEL_AMBIENT,</td>
<td></td>
</tr>
<tr>
<td>GL_LIGHT_MODEL_LOCAL_VIEWER, GL_LIGHT_MODEL_TWO_SIDE</td>
<td>glGetLight*()</td>
</tr>
<tr>
<td>GL_LINE</td>
<td>gIPolygonMode(), glEvalMesh*()</td>
</tr>
<tr>
<td>GL_LINEAR</td>
<td>glFog*(), glTexParameter*()</td>
</tr>
<tr>
<td>GL_LINEAR_ATTENUATION</td>
<td>glLight*(), glGetLight*()</td>
</tr>
<tr>
<td>GL_LINEAR_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST</td>
<td>glTexParameter*()</td>
</tr>
<tr>
<td>GL_LINES</td>
<td>glBegin()</td>
</tr>
<tr>
<td>Constant</td>
<td>Associated Commands</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>--------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>GL_LINE_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_LINE_LOOP</td>
<td>glBegin()</td>
</tr>
<tr>
<td>GL_LINE_RESET_TOKEN</td>
<td>glPassThrough()</td>
</tr>
<tr>
<td>GL_LINE_SMOOTH</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_LINE_SMOOTH_HINT</td>
<td>glHint(), glGet*()</td>
</tr>
<tr>
<td>GL_LINE_STIPPLE</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_LINE_STIPPLE_PATTERN, GL_LINE_STIPPLE_REPEAT</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_LINE_STRIP</td>
<td>glBegin()</td>
</tr>
<tr>
<td>GL_LINE_TOKEN</td>
<td>glPassThrough()</td>
</tr>
<tr>
<td>GL_LINE_WIDTH, GL_LINE_WIDTH_GRANULARITY, GL_LINE_WIDTH_RANGE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_LIST_BASE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_LIST_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_LIST_INDEX, GL_LIST_MODE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_LOAD</td>
<td>glAccum()</td>
</tr>
<tr>
<td>GL_LOGIC_OP</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_LOGIC_OP_MODE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_LUMINANCE, GL_LUMINANCE_ALPHA</td>
<td>glDrawPixels(), glReadPixels(), glTexImage1D(), glTexImage2D(), glGetTexImage()</td>
</tr>
<tr>
<td>GL_MAP1_COLOR_4</td>
<td>glMap1*(), glEnable(), glIsEnabled(), glGetMap*()</td>
</tr>
<tr>
<td>GL_MAP1_GRID_DOMAIN, GL_MAP1_GRID_SEGMENTS</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_MAP1_INDEX, GL_MAP1_NORMAL, GL_MAP1_TEXTURE_COORD_1 through GL_MAP1_TEXTURE_COORD_4, GL_MAP1_VERTEX_3, GL_MAP1_VERTEX_4</td>
<td>glMap1*(), glEnable(), glIsEnabled(), glGetMap*()</td>
</tr>
<tr>
<td>GL_MAP2_COLOR_4</td>
<td>glMap2*(), glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_MAP2_GRID_DOMAIN</td>
<td>glGet*()</td>
</tr>
<tr>
<td>Constant</td>
<td>Associated Commands</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------</td>
</tr>
<tr>
<td>GL_MAP2_GRID_SEGMENTS</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_MAP2_INDEX, GL_MAP2_NORMAL, GL_MAP2_TEXTURE_COORD_1 through GL_MAP2_TEXTURE_COORD_4, GL_MAP2_VERTEX_3, GL_MAP2_VERTEX_4</td>
<td>glMap2*(), glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_MAP_COLOR, GL_MAP_STENCIL</td>
<td>glPixelTransfer*(), glGet*()</td>
</tr>
<tr>
<td>GL_MATRIX_MODE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_MODELVIEW</td>
<td>glMatrixMode()</td>
</tr>
<tr>
<td>GL_MODELVIEW_MATRIX, GL_MODELVIEW_STACK_DEPTH</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_MODULATE</td>
<td>glTexEnv*(), glGetTexEnv*()</td>
</tr>
<tr>
<td>GL_MULT</td>
<td>glAccum()</td>
</tr>
<tr>
<td>GL_NAME_STACK_DEPTH</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_NAND</td>
<td>glLogicOp()</td>
</tr>
<tr>
<td>GL_NEAREST, GL_NEAREST_MIPMAP_LINEAR, GL_NEAREST_MIPMAP_NEAREST</td>
<td>glTexParameter*()</td>
</tr>
<tr>
<td>GL_NEARER</td>
<td>gIDepthFunc(), glAlphaFunc(), glStencilFunc()</td>
</tr>
<tr>
<td>GL_NICEST</td>
<td>glHint()</td>
</tr>
<tr>
<td>GL_NONE</td>
<td>glDrawBuffer()</td>
</tr>
<tr>
<td>GL_NOOP, GL_NOR</td>
<td>glLogicOp()</td>
</tr>
<tr>
<td>GL_NORMALIZE</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_NOTEQUAL</td>
<td>gIDepthFunc(), glAlphaFunc(), glStencilFunc()</td>
</tr>
<tr>
<td>GL_NO_ERROR</td>
<td>glGetError()</td>
</tr>
<tr>
<td>GL_OBJECT_LINEAR</td>
<td>glTexImage*(), glGetTexImage*()</td>
</tr>
<tr>
<td>Constant</td>
<td>Associated Commands</td>
</tr>
<tr>
<td>-----------------------------------------------------</td>
<td>--------------------------------------------------------</td>
</tr>
<tr>
<td>GL_OBJECT_PLANE</td>
<td>glTexGen*()</td>
</tr>
<tr>
<td>GL_ONE, GL_ONE_MINUS_DST_ALPHA,</td>
<td>glBlendFunc()</td>
</tr>
<tr>
<td>GL_ONE_MINUS_DST_COLOR, GL_ONE_MINUS_SRC_ALPHA,</td>
<td></td>
</tr>
<tr>
<td>GL_ONE_MINUS_SRC_COLOR</td>
<td></td>
</tr>
<tr>
<td>GL_OR, GL_OR_INVERTED, GL_OR_REVERSE</td>
<td>glLogicOp()</td>
</tr>
<tr>
<td>GL_ORDER</td>
<td>glGetMap*()</td>
</tr>
<tr>
<td>GL_OUT_OF_MEMORY</td>
<td>glGetError()</td>
</tr>
<tr>
<td>GL_PACK_ALIGNMENT, GL_PACK_LSB_FIRST,</td>
<td>glPixelStore*(), glGet*()</td>
</tr>
<tr>
<td>GL_PACK_ROW_LENGTH, GL_PACK_SKIP_PIXELS, GL_PACK_SKIP_ROWS,</td>
<td></td>
</tr>
<tr>
<td>GL_PACK_SWAP_BYTES</td>
<td></td>
</tr>
<tr>
<td>GL_PASS_THROUGH_TOKEN</td>
<td>glPassThrough()</td>
</tr>
<tr>
<td>GL_PERSPECTIVE_CORRECTION_HINT</td>
<td>glHint(), glGet*()</td>
</tr>
<tr>
<td>GL_PIXEL_MAP_<em><em>TO</em></em></td>
<td>glGetPixelMap*, glGetPixelMap*()</td>
</tr>
<tr>
<td>GL_PIXEL_MAP_<em><em>TO</em></em>_SIZE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_PIXEL_MODE_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_POINT</td>
<td>glPolygonMode(), glEvalMesh*()</td>
</tr>
<tr>
<td>GL_POINTS</td>
<td>glBegin()</td>
</tr>
<tr>
<td>GL_POINT_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_POINT_SIZE, GL_POINT_SIZE_GRANULARITY,</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_POINT_SIZE_RANGE</td>
<td></td>
</tr>
<tr>
<td>GL_POINT_SMOOTH</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_POINT_SMOOTH_HINT</td>
<td>glHint(), glGet*()</td>
</tr>
<tr>
<td>GL_POINT_TOKEN</td>
<td>glPassThrough()</td>
</tr>
<tr>
<td>GL_POLYGON</td>
<td>glBegin()</td>
</tr>
<tr>
<td>GL_POLYGON_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_POLYGON_MODE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_POLYGON_SMOOTH</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_POLYGON_SMOOTH_HINT</td>
<td>glHint(), glGet*()</td>
</tr>
<tr>
<td>GL_POLYGON_STIPPLE</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>Constant</td>
<td>Associated Commands</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------</td>
</tr>
<tr>
<td>GL_POLYGON_STIPPLE_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_POLYGON_TOKEN</td>
<td>glPassThrough()</td>
</tr>
<tr>
<td>GL_POSITION</td>
<td>glLight*, glGetLight*()</td>
</tr>
<tr>
<td>GL_PROJECTION</td>
<td>glMatrixMode()</td>
</tr>
<tr>
<td>GL_PROJECTION_MATRIX, GL_PROJECTION_STACK_DEPTH</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_Q</td>
<td>glTexGen*, glGetTexGen*()</td>
</tr>
<tr>
<td>GL_QUADRATIC_ATTENUATION</td>
<td>glLight*, glGetLight*()</td>
</tr>
<tr>
<td>GL_QUADS, GL_QUAD_STRIP</td>
<td>glBegin()</td>
</tr>
<tr>
<td>GL_R</td>
<td>glTexGen*, glGetTexGen*()</td>
</tr>
<tr>
<td>GL_READ_BUFFER</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_RED</td>
<td>glDrawPixels(), glReadPixels(), glTexImage1D(), glTexImage2D(), glGetTexImage()</td>
</tr>
<tr>
<td>GL_RED_BIAS</td>
<td>glPixelTransfer*, glGet*()</td>
</tr>
<tr>
<td>GL_RED_BITS</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_RED_SCALE</td>
<td>glPixelTransfer*, glGet*()</td>
</tr>
<tr>
<td>GL_RENDER</td>
<td>glRenderMode()</td>
</tr>
<tr>
<td>GL_RENDERER</td>
<td>glGetString()</td>
</tr>
<tr>
<td>GL_RENDER_MODE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_REPEAT</td>
<td>glTexParameter*()</td>
</tr>
<tr>
<td>GL_REPLACE</td>
<td>glStencilOp()</td>
</tr>
<tr>
<td>GL_RETURN</td>
<td>glAccum()</td>
</tr>
<tr>
<td>GL_RGB</td>
<td>glDrawPixels(), glReadPixels(), glTexImage1D(), glTexImage2D(), glGetTexImage()</td>
</tr>
<tr>
<td>GL_RGBA</td>
<td>glDrawPixels(), glReadPixels(), glTexImage1D(), glTexImage2D(), glGetTexImage()</td>
</tr>
<tr>
<td>GL_RGBA_MODE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_RIGHT</td>
<td>glDrawBuffer()</td>
</tr>
<tr>
<td>Constant</td>
<td>Associated Commands</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>glReadBuffer()</td>
<td>glTexGen*(), glGetTexGen*()</td>
</tr>
<tr>
<td>GL_S</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_SCISSOR_BIT</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_SCISSOR_BOX</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_SCISSOR_TEST</td>
<td>glRenderMode()</td>
</tr>
<tr>
<td>GL_SELECT</td>
<td>glLogicOp()</td>
</tr>
<tr>
<td>GL_SET</td>
<td>glMaterial*(), glGetMaterial*()</td>
</tr>
<tr>
<td>GL_SHININESS</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_SHADE_MODEL</td>
<td>glCallLists(), glDrawPixels(), glReadPixels(), glTexImage1D(), glTexImage2D(),</td>
</tr>
<tr>
<td></td>
<td>glGetTexImage()</td>
</tr>
<tr>
<td>GL_SMOOTH</td>
<td>glShadeModel()</td>
</tr>
<tr>
<td>GL_SPECULAR</td>
<td>glLight*(), glGetLight*(), glMaterial*(), glGetMaterial*(), glColorMaterial()</td>
</tr>
<tr>
<td>GL_SPHERE_MAP</td>
<td>glTexGen*(), glGetTexGen*()</td>
</tr>
<tr>
<td>GL_SPOT_CUTOFF, GL_SPOT_DIRECTION,</td>
<td>glLight*(), glGetLight*()</td>
</tr>
<tr>
<td>GL_SPOT_EXPONENT</td>
<td>glBlendFunc()</td>
</tr>
<tr>
<td>GL_SRC_ALPHA, GL_SRC_ALPHA_SATURATE,</td>
<td>glGetError()</td>
</tr>
<tr>
<td>GL_SRC_COLOR</td>
<td>glCopyPixels()</td>
</tr>
<tr>
<td>GL_STACK_OVERFLOW,</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_STACK_UNDERFLOW</td>
<td>glClear(), glPushAttrib()</td>
</tr>
<tr>
<td>GL_STENCIL</td>
<td>glDrawPixels(), glReadPixels(), glTexImage1D(), glTexImage2D(), glGetTexImage()</td>
</tr>
<tr>
<td>GL_STENCIL_BITS</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_STENCIL_BUFFER_BIT</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_STENCIL_INDEX</td>
<td>glGet*()</td>
</tr>
<tr>
<td>Constant</td>
<td>Associated Commands</td>
</tr>
<tr>
<td>---------------------------------------------------</td>
<td>---------------------------------------------------------</td>
</tr>
<tr>
<td>GL_STENCIL_PASS_DEPTH_FAIL, GL_STENCIL_PASS_DEPTH_PASS, GL_STENCIL_REF</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_STENCIL_TEST</td>
<td></td>
</tr>
<tr>
<td>GL_STENCIL_VALUE_MASK, GL_STENCIL_WRITEMASK</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_STEREO</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_SUBPIXEL_BITS</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_T</td>
<td>glTexGen*(), glGetTexGen*()</td>
</tr>
<tr>
<td>GL_TEXTURE</td>
<td>glMatrixMode()</td>
</tr>
<tr>
<td>GL_TEXTURE_1D</td>
<td>glTexImage1D(), glGetTexImage(), glTexParameteri*(),</td>
</tr>
<tr>
<td></td>
<td>glGetTexParameter*, glGetTexParameter*, glEnable(),</td>
</tr>
<tr>
<td></td>
<td>glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_TEXTURE_2D</td>
<td></td>
</tr>
<tr>
<td>GL_TEXTURE_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_TEXTURE_BORDER</td>
<td>glGetTexParameter*, glGetTexParameter*, glEnable(),</td>
</tr>
<tr>
<td></td>
<td>glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_TEXTURE_BORDER_COLOR</td>
<td></td>
</tr>
<tr>
<td>GL_TEXTURE_COMPONENTS</td>
<td>glGetTexParameter*, glGetTexParameter*, glEnable(),</td>
</tr>
<tr>
<td></td>
<td>glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_TEXTURE_ENV, GL_TEXTURE_ENV_COLOR, GL_TEXTURE_ENV_MODE</td>
<td>glTexEnv*, glGetTexEnv*()</td>
</tr>
<tr>
<td>GL_TEXTURE_GEN_MODE</td>
<td>glTexGen*()</td>
</tr>
<tr>
<td>GL_TEXTURE_HEIGHT</td>
<td>glGetTexParameter*(),</td>
</tr>
<tr>
<td>Constant</td>
<td>Associated Commands</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>GL_TEXTURE_MAG_FILTER</td>
<td>glGetTexParameter*, glGetTexLevelParameter*()</td>
</tr>
<tr>
<td>GL_TEXTURE_MATRIX</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_TEXTURE_MIN_FILTER</td>
<td>glGetTexParameter*, glGetTexLevelParameter*()</td>
</tr>
<tr>
<td>GL_TEXTURE_STACK_DEPTH</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_TEXTURE_WIDTH</td>
<td>glGetTexParameter*, glGetTexLevelParameter*()</td>
</tr>
<tr>
<td>GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T</td>
<td>glGetTexParameter*, glGetTexLevelParameter*()</td>
</tr>
<tr>
<td>GL_TRANSFORM_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_TRIANGLES, GL_TRIANGLE_FAN, GL_TRIANGLE_STRIP</td>
<td>glBegin()</td>
</tr>
<tr>
<td>GL_TRUE</td>
<td>glColorMask*, glGet*, glIsEnabled*, glIsList()</td>
</tr>
<tr>
<td>GL_UNPACK_ALIGNMENT, GL_UNPACK_LSB_FIRST, GL_UNPACK_ROW_LENGTH, GL_UNPACK_SKIP_PIXELS, GL_UNPACK_SKIP_ROWS, GL_UNPACK_SWAP_BYTES</td>
<td>glGetStore*, glGet*()</td>
</tr>
<tr>
<td>GL_UNSIGNED_BYTE, GL_UNSIGNED_INT, GL_UNSIGNED_SHORT</td>
<td>glCallLists*, glDrawPixels*, glReadPixels*, glGetImage1D*, glGetImage2D*, glGetTexImage()</td>
</tr>
<tr>
<td>GL_VENDOR, GL_VERSION</td>
<td>glGetString*</td>
</tr>
<tr>
<td>GL_VIEWPORT</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_VIEWPORT_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_XOR</td>
<td>glLogicOp()</td>
</tr>
<tr>
<td>GL_ZERO</td>
<td>glBlendFunc*, glStencilOp*</td>
</tr>
<tr>
<td>GL_ZOOM_X, GL_ZOOM_Y</td>
<td>glGet*()</td>
</tr>
</tbody>
</table>
Chapter 4. Defined Constants and Associated Commands

This chapter lists all the defined constants in OpenGL and their corresponding commands; these constants might indicate a parameter name, a value for a parameter, a mode, a query target, or a return value. The list is intended to be used as another index into the reference pages: if you remember the name of a constant, you can use this table to find out which functions use it, and then you can refer to the reference pages for those functions for more information. Note that all the constants listed can be used directly by the corresponding commands; the reference pages list additional, related commands that might be of interest.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Associated Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_2D, GL_3D, GL_3D_COLOR, GL_COLOR_TEXTURE, GL_4D_COLOR_TEXTURE</td>
<td>glFeedbackBuffer()</td>
</tr>
<tr>
<td>GL_2_BYTES, GL_3_BYTES, GL_4_BYTES</td>
<td>glCallLists()</td>
</tr>
<tr>
<td>GL_ACCUM</td>
<td>glAccum()</td>
</tr>
<tr>
<td>GL_ACCUM_ALPHA_BITS, GL_ACCUM_BLUE_BITS</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_ACCUM_BUFFER_BIT</td>
<td>glClear(), glPushAttrib()</td>
</tr>
<tr>
<td>GL_ACCUM_CLEAR_VALUE, GL_ACCUM_GREEN_BITS, GL_ACCUM_RED_BITS</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_ADD</td>
<td>glAccum()</td>
</tr>
<tr>
<td>GL_ALL_ATTRIB_BITS</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_ALPHA</td>
<td>glDrawPixels(), glReadPixels(), glTexImage1D(), glTexImage2D(), glGetTexImage()</td>
</tr>
<tr>
<td>GL_ALPHA_BIAS</td>
<td>glPixelTransfer*, glGet*()</td>
</tr>
<tr>
<td>GL_ALPHA_BITS</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_ALPHA_SCALE</td>
<td>glPixelTransfer*, glGet*()</td>
</tr>
<tr>
<td>GL_ALPHA_TEST</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_ALPHA_TEST_FUNC, GL_ALPHA_TEST_REF</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_ALWAYS</td>
<td>glAlphaFunc(), glDepthFunc(), glStencilFunc()</td>
</tr>
<tr>
<td>GL_AMBIENT</td>
<td>glLight*, glGetLight*, glMaterial*,</td>
</tr>
<tr>
<td>Constant</td>
<td>Associated Commands</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------</td>
</tr>
<tr>
<td>GL_AMBIENT_AND_DIFFUSE</td>
<td>glGetMaterial*(), glColorMaterial()</td>
</tr>
<tr>
<td>GL_AND, GL_AND_INVERTED, GL_AND_REVERSE</td>
<td>glLogicOp()</td>
</tr>
<tr>
<td>GL_ATTRIB_STACK_DEPTH</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_AUTO_NORMAL</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_AUX0 through GL_AUX3</td>
<td>glDrawBuffer(), glReadBuffer()</td>
</tr>
<tr>
<td>GL_AUX_BUFFERS</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_BACK</td>
<td>glColorMaterial(), glCullFace(), glDrawBuffer(), glReadBuffer(), glMaterial*(), glGetMaterial*(), glPolygonMode()</td>
</tr>
<tr>
<td>GL_BACK_LEFT, GL_BACK_RIGHT</td>
<td>glDrawBuffer(), glReadBuffer()</td>
</tr>
<tr>
<td>GL_BITMAP</td>
<td>glDrawPixels(), glReadPixels(), glTexImage1D(), glTexImage2D(), glGetTexImage()</td>
</tr>
<tr>
<td>GL_BITMAP_TOKEN</td>
<td>glPassThrough()</td>
</tr>
<tr>
<td>GL_BLEND</td>
<td>glTexEnv*(), glGetTexEnv*(), glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_BLEND_DST, GL_BLEND_SRC</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_BLUE</td>
<td>glDrawPixels(), glReadPixels(), glTexImage1D(), glTexImage2D(), glGetTexImage()</td>
</tr>
<tr>
<td>GL_BLUE_BIAS</td>
<td>glPixelTransfer*(), glGet*()</td>
</tr>
<tr>
<td>GL_BLUE_BITS</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_BLUE_SCALE</td>
<td>glPixelTransfer*(), glGet*()</td>
</tr>
<tr>
<td>GL_BYTE</td>
<td>glCallLists(), glDrawPixels(),...</td>
</tr>
<tr>
<td>Constant</td>
<td>Associated Commands</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>-------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>glReadPixels(),</td>
<td></td>
</tr>
<tr>
<td>glTexImage1D(),</td>
<td></td>
</tr>
<tr>
<td>glTexImage2D(),</td>
<td></td>
</tr>
<tr>
<td>glGetTexImage()</td>
<td></td>
</tr>
<tr>
<td>glFrontFace()</td>
<td></td>
</tr>
<tr>
<td>glTexParameteri*()</td>
<td></td>
</tr>
<tr>
<td>glLogicOp()</td>
<td></td>
</tr>
<tr>
<td>glEnable(), glIsEnabled()</td>
<td></td>
</tr>
<tr>
<td>glClipPlane(), glGetClipPlane(),</td>
<td></td>
</tr>
<tr>
<td>glEnable(), glIsEnabled()</td>
<td></td>
</tr>
<tr>
<td>glDrawPixels(),</td>
<td></td>
</tr>
<tr>
<td>glGet*()</td>
<td></td>
</tr>
<tr>
<td>glCopyPixels()</td>
<td></td>
</tr>
<tr>
<td>glClear(), glPushAttrib()</td>
<td></td>
</tr>
<tr>
<td>glGet*()</td>
<td></td>
</tr>
<tr>
<td>glMaterial*(), glGetMaterial*()</td>
<td></td>
</tr>
<tr>
<td>glEnable(), glIsEnabled(), glGet*()</td>
<td></td>
</tr>
<tr>
<td>glGet*()</td>
<td></td>
</tr>
<tr>
<td>glGetLight*()</td>
<td></td>
</tr>
<tr>
<td>glLogicOp()</td>
<td></td>
</tr>
<tr>
<td>glPassThrough()</td>
<td></td>
</tr>
<tr>
<td>glEnable(), glIsEnabled(), glGet*()</td>
<td></td>
</tr>
<tr>
<td>glPushAttrib()</td>
<td></td>
</tr>
<tr>
<td>glGet*()</td>
<td></td>
</tr>
<tr>
<td>Constant</td>
<td>Associated Commands</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------</td>
</tr>
<tr>
<td>GL_CURRENT_RASTER_COLOR, GL_CURRENT_RASTER_INDEX, GL_CURRENT_RASTER_POSITION, GL_CURRENT_RASTER_POSITION_VALID, GL_CURRENT_RASTER_TEXTURE_COORDS, GL_CURRENT_TEXTURE_COORDS</td>
<td></td>
</tr>
<tr>
<td>GL_CW</td>
<td>glFrontFace()</td>
</tr>
<tr>
<td>GL_DECAL</td>
<td>glTexEnv*(), glGetTexEnv*()</td>
</tr>
<tr>
<td>GL_DECOR</td>
<td>glStencilOp()</td>
</tr>
<tr>
<td>GL_DEPTH</td>
<td>glCopyPixels()</td>
</tr>
<tr>
<td>GL_DEPTH_BIAS</td>
<td>glPixelTransfer*(), glGet*()</td>
</tr>
<tr>
<td>GL_DEPTH_BITS</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_DEPTH_BUFFER_BIT</td>
<td>glClear(), glPushAttrib()</td>
</tr>
<tr>
<td>GL_DEPTH_CLEAR_VALUE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_DEPTH_COMPONENT</td>
<td>glDrawPixels(), glReadPixels(), glTexImage1D(), glTexImage2D(), glGetTexImage()</td>
</tr>
<tr>
<td>GL_DEPTH_FUNC</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_DEPTH_RANGE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_DEPTH_SCALE</td>
<td>glPixelTransfer*(), glGet*()</td>
</tr>
<tr>
<td>GL_DEPTH_TEST</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_DEPTH_WRITEMASK</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_DIFFUSE</td>
<td>glLight*(), glGetLight*(), glMaterial*(), glGetMaterial*(), glColorMaterial()</td>
</tr>
<tr>
<td>GL_DITHER</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_DOMAIN</td>
<td>glGetMap*()</td>
</tr>
<tr>
<td>GL_DONT_CARE</td>
<td>glHint()</td>
</tr>
<tr>
<td>GL_DOUBLEBUFFER</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_DRAW_BUFFER</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_DRAW_PIXEL_TOKEN</td>
<td>glPassThrough()</td>
</tr>
<tr>
<td>GL_DST_ALPHA, GL_DST_COLOR</td>
<td>glBlendFunc()</td>
</tr>
<tr>
<td>Constant</td>
<td>Associated Commands</td>
</tr>
<tr>
<td>---------------------------</td>
<td>--------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>GL_EDGE_FLAG</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_EMISSION</td>
<td>glVertex*(), glGetMaterial*(), glColorMaterial()</td>
</tr>
<tr>
<td>GL_ENABLE_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_EQUAL</td>
<td>glAlphaFunc(), glDepthFunc(), glStencilFunc()</td>
</tr>
<tr>
<td>GL_EQUIV</td>
<td>glLogicOp()</td>
</tr>
<tr>
<td>GL_EVAL_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_EXP, GL_EXP2</td>
<td>glFog*()</td>
</tr>
<tr>
<td>GL_EXTENSIONS</td>
<td>glGetString()</td>
</tr>
<tr>
<td>GL_EYE_LINEAR</td>
<td>glTexGen*(), glGetTexGen*()</td>
</tr>
<tr>
<td>GL_EYE_PLANE</td>
<td>glTexGen*()</td>
</tr>
<tr>
<td>GL_FALSE</td>
<td>glColorMask(), glGet*, glIsEnabled(), glIsList()</td>
</tr>
<tr>
<td>GL_FASTEST</td>
<td>glHint()</td>
</tr>
<tr>
<td>GL_FEEDBACK</td>
<td>glRenderMode()</td>
</tr>
<tr>
<td>GL_FILL</td>
<td>glPolygonMode(), glEvalMesh2()</td>
</tr>
<tr>
<td>GL_FLAT</td>
<td>glShadeModel()</td>
</tr>
<tr>
<td>GL_FLOAT</td>
<td>glCallLists(), glDrawPixels(), glGet*(), glTexImage1D(), glTexImage2D(), glGetTexImage()</td>
</tr>
<tr>
<td>GL_FOG</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_FOG_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_FOG_COLOR, GL_FOG_DENSITY, GL_FOG_END</td>
<td>glFog*, glGet*()</td>
</tr>
<tr>
<td>GL_FOG_HINT</td>
<td>glHint()</td>
</tr>
<tr>
<td>GL_FOG_INDEX, GL_FOG_MODE, GL_FOG_START</td>
<td>glFog*, glGet*()</td>
</tr>
<tr>
<td>GL_FRONT</td>
<td>glColorMaterial(), glCullFace(), glDrawBuffer(), glReadBuffer(), glGetMaterial*, glGetMaterial*, glPolygonMode()</td>
</tr>
<tr>
<td>Constant</td>
<td>Associated Commands</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>GL_FRONT_AND_BACK</td>
<td>glColorMaterial(), glDrawBuffer(), glMaterial*(), glPolygonMode()</td>
</tr>
<tr>
<td>GL_FRONT_FACE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_FRONT_LEFT, GL_FRONT_RIGHT</td>
<td>glDrawBuffer(), glReadBuffer()</td>
</tr>
<tr>
<td>GL_GEQUAL, GL_GREATER</td>
<td>glDepthFunc(), glAlphaFunc(), glStencilFunc()</td>
</tr>
<tr>
<td>GL_GREEN</td>
<td>glDrawPixels(), glReadPixels(), glTexImage1D(), glTexImage2D(), glGetTexImage()</td>
</tr>
<tr>
<td>GL_GREEN_BIAS</td>
<td>glPixelTransfer*(), glGet*()</td>
</tr>
<tr>
<td>GL_GREEN_BITS</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_GREEN_SCALE</td>
<td>glPixelTransfer*(), glGet*()</td>
</tr>
<tr>
<td>GL_HINT_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_INCR</td>
<td>glStencilOp()</td>
</tr>
<tr>
<td>GL_INDEX_BITS, GL_INDEX_CLEAR_VALUE, GL_INDEX_MODE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_INDEX_OFFSET, GL_INDEX_SHIFT</td>
<td>glPixelTransfer*(), glGet*()</td>
</tr>
<tr>
<td>GL_INDEX_WRITEMASK</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_INT</td>
<td>glCallLists(), glDrawPixels(), glReadPixels(), glTexImage1D(), glTexImage2D(), glGetTexImage()</td>
</tr>
<tr>
<td>GL_INVALID_ENUM, GL_INVALID_OPERATION, GL_INVALID_VALUE</td>
<td>glGetError()</td>
</tr>
<tr>
<td>GL_INVERT</td>
<td>glLogicOp(), glStencilOp()</td>
</tr>
<tr>
<td>GL_KEEP</td>
<td>glStencilOp()</td>
</tr>
<tr>
<td>GL_LEFT</td>
<td>glDrawBuffer(), glReadBuffer()</td>
</tr>
<tr>
<td>GL_LEQUAL, GL_LESS</td>
<td>glDepthFunc(), glAlphaFunc(), glStencilFunc()</td>
</tr>
<tr>
<td>GL_LIGHT0 through GL_LIGHT7</td>
<td>glLight*(), glGetLight*(), glEnable(), glIsEnabled()</td>
</tr>
<tr>
<td>GL_LIGHTING</td>
<td>glEnable(), glIsEnabled()</td>
</tr>
<tr>
<td>Constant</td>
<td>Associated Commands</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>GL_LIGHTING_BIT</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_LIGHT_MODEL_AMBIENT, GL_LIGHT_MODEL_LOCAL_VIEWER, GL_LIGHT_MODEL_TWO_SIDE</td>
<td>glLightModel*, glGet*()</td>
</tr>
<tr>
<td>GL_LINE</td>
<td>glPolygonMode(), glEvalMesh*()</td>
</tr>
<tr>
<td>GL_LINE_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_LINE_LOOP</td>
<td>glBegin()</td>
</tr>
<tr>
<td>GL_LINE_RESET_TOKEN</td>
<td>glPassThrough()</td>
</tr>
<tr>
<td>GL_LINE_SMOOTH</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_LINE_SMOOTH_HINT</td>
<td>glHint(), glGet*()</td>
</tr>
<tr>
<td>GL_LINE_STIPPLE</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_LINE_STIPPLE_PATTERN, GL_LINE_STIPPLE_REPEAT</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_LINE_STRIP</td>
<td>glBegin()</td>
</tr>
<tr>
<td>GL_LINE_TOKEN</td>
<td>glPassThrough()</td>
</tr>
<tr>
<td>GL_LINE_WIDTH, GL_LINE_WIDTH_GRANULARITY, GL_LINE_WIDTH_RANGE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_LIST_BASE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_LIST_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_LIST_INDEX, GL_LIST_MODE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_LOAD</td>
<td>glAccum()</td>
</tr>
<tr>
<td>GL_LOGIC_OP</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_LOGIC_OP_MODE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_LUMINANCE, GL_LUMINANCE_ALPHA</td>
<td>glDrawPixels(), glReadPixels(), glTexImage1D(),...</td>
</tr>
<tr>
<td>Constant</td>
<td>Associated Commands</td>
</tr>
<tr>
<td>-------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>GL_MAP1_COLOR_4</td>
<td>glTexImage2D(), glGetTexImage()</td>
</tr>
<tr>
<td>GL_MAP1_GRID_DOMAIN, GL_MAP1_GRID_SEGMENTS</td>
<td>glMap1*(), glEnable(), glIsEnabled(), glGetMap*()</td>
</tr>
<tr>
<td>GL_INDEX, GL_MAP1_NORMAL, GL_MAP1_TEXTURE_COORD_1 through GL_MAP1_TEXTURE_COORD_4, GL_MAP1_VERTEX_3, GL_MAP1_VERTEX_4</td>
<td>glMap1*(), glEnable(), glIsEnabled(), glGetMap*()</td>
</tr>
<tr>
<td>GL_MAP2_COLOR_4</td>
<td>glMap2*(), glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_MAP2_GRID_DOMAIN</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_MAP2_GRID_SEGMENTS</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_INDEX, GL_MAP2_NORMAL, GL_MAP2_TEXTURE_COORD_1 through GL_MAP2_TEXTURE_COORD_4, GL_MAP2_VERTEX_3, GL_MAP2_VERTEX_4</td>
<td>glMap2*(), glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_MAP_COLOR, GL_MAP_STENCIL</td>
<td>glPixelTransfer*(), glGet*()</td>
</tr>
<tr>
<td>GL_MATRIX_MODE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_MODELVIEW</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_MODELVIEW_MATRIX, GL_MODELVIEW_STACK_DEPTH</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_MODULATE</td>
<td>glTexImage2D*, glGetTexEnv*()</td>
</tr>
<tr>
<td>GL_MULT</td>
<td>glAccum()</td>
</tr>
<tr>
<td>GL_NAME_STACK_DEPTH</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_NAND</td>
<td>glLogicOp()</td>
</tr>
<tr>
<td>GL_NEAREST, GL_NEAREST_MIPMAP_LINEAR,</td>
<td>glTexImage2D*, glGet*()</td>
</tr>
<tr>
<td>Constant</td>
<td>Associated Commands</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>GL_NEAREST_MIPMAP_NEAREST</td>
<td>glDepthFunc(), glAlphaFunc(), glStencilFunc()</td>
</tr>
<tr>
<td>GL_NEVER</td>
<td>glDepthFunc(), glAlphaFunc(), glStencilFunc()</td>
</tr>
<tr>
<td>GL_NICEST</td>
<td>glHint()</td>
</tr>
<tr>
<td>GL_NONE</td>
<td>glDrawBuffer()</td>
</tr>
<tr>
<td>GL_NOOP, GL_NOR</td>
<td>glLogicOp()</td>
</tr>
<tr>
<td>GL_NORMALIZE</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_NOTEQUAL</td>
<td>glDepthFunc(), glAlphaFunc(), glStencilFunc()</td>
</tr>
<tr>
<td>GL_NO_ERROR</td>
<td>glGetError()</td>
</tr>
<tr>
<td>GL_OBJECT_LINEAR</td>
<td>glTexGen*(), glGetTexGen*()</td>
</tr>
<tr>
<td>GL_OBJECT_PLANE</td>
<td>glTexGen*()</td>
</tr>
<tr>
<td>GL_ONE, GL_ONE_MINUS_DST_ALPHA, GL_ONE_MINUS_DST_COLOR, GL_ONE_MINUS_SRC_ALPHA, GL_ONE_MINUS_SRC_COLOR</td>
<td>glBlendFunc()</td>
</tr>
<tr>
<td>GL_OR, GL_OR_INVERTED, GL_OR_REVERSE</td>
<td>glLogicOp()</td>
</tr>
<tr>
<td>GL_ORDER</td>
<td>glGetMap*()</td>
</tr>
<tr>
<td>GL_OUT_OF_MEMORY</td>
<td>glGetError()</td>
</tr>
<tr>
<td>GL_PACK_ALIGNMENT, GL_PACK_LSB_FIRST, GL_PACK_ROW_LENGTH, GL_PACK_SKIP_PIXELS, GL_PACK_SKIP_ROWS, GL_PACK_SWAP_BYTES</td>
<td>PixelStore*(), glGet*()</td>
</tr>
<tr>
<td>GL_PASS_THROUGH_TOKEN</td>
<td>glPassThrough()</td>
</tr>
<tr>
<td>GL_PERSPECTIVE_CORRECTION_HINT</td>
<td>glHint(), glGet*()</td>
</tr>
<tr>
<td>GL_PIXEL_MAP_* TO _</td>
<td>glPixelMap*(), glGetPixelMap*()</td>
</tr>
<tr>
<td>GL_PIXEL_MAP_* TO _ SIZE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_PIXEL_MODE_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_POINT</td>
<td>glPolygonMode(), glGetMesh*()</td>
</tr>
<tr>
<td>GL_POINTS</td>
<td>glBegin()</td>
</tr>
<tr>
<td>GL_POINT_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_POINT_SIZE, GL_POINT_SIZE_GRANULARITY, GL_POINT_SIZE_RANGE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>Constant</td>
<td>Associated Commands</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>GL_POINT_SMOOTH</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_POINT_SMOOTH_HINT</td>
<td>glHint(), glGet*()</td>
</tr>
<tr>
<td>GL_POINT_TOKEN</td>
<td>glPassThrough()</td>
</tr>
<tr>
<td>GL_POLYGON</td>
<td>glBegin()</td>
</tr>
<tr>
<td>GL_POLYGON_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_POLYGON_MODE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_POLYGON_SMOOTH</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_POLYGON_SMOOTH_HINT</td>
<td>glHint(), glGet*()</td>
</tr>
<tr>
<td>GL_POLYGON_STIPPLE</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_POLYGON_STIPPLE_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_POLYGON_TOKEN</td>
<td>glPassThrough()</td>
</tr>
<tr>
<td>GL_POSITION</td>
<td>glLight*, glGetLight*()</td>
</tr>
<tr>
<td>GL_PROJECTION</td>
<td>glMatrixMode()</td>
</tr>
<tr>
<td>GL_PROJECTION_MATRIX, GL_PROJECTION_STACK_DEPTH</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_Q</td>
<td>glTexImage1D(), glTexImage2D(), glGetTexImage()</td>
</tr>
<tr>
<td>GL_QUADRATIC_ATTENUATION</td>
<td>glLight*, glGetLight*()</td>
</tr>
<tr>
<td>GL_QUADS, GL_QUAD_STRIP</td>
<td>glBegin()</td>
</tr>
<tr>
<td>GL_R</td>
<td>glTexImage1D(), glGetTexImage()</td>
</tr>
<tr>
<td>GL_READ_BUFFER</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_RED</td>
<td>glTexImage1D(), glTexImage2D(), glGetTexImage()</td>
</tr>
<tr>
<td>GL_RED_BIAS</td>
<td>glPixelTransfer*, glGet*()</td>
</tr>
<tr>
<td>GL_RED_BITS</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_RED_SCALE</td>
<td>glPixelTransfer*, glGet*()</td>
</tr>
<tr>
<td>GL_RENDER</td>
<td>glRenderMode()</td>
</tr>
<tr>
<td>GL_RENDERER</td>
<td>glGetString()</td>
</tr>
<tr>
<td>GL_RENDERER_MODE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_REPLACE</td>
<td>glStencilOp()</td>
</tr>
<tr>
<td>Constant</td>
<td>Associated Commands</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>GL_RETURN</td>
<td>glAccum()</td>
</tr>
<tr>
<td>GL_RGB</td>
<td>glDrawPixels(), glReadPixels(), glTexImage1D(), glTexImage2D(), glGetTexImage()</td>
</tr>
<tr>
<td>GL_RGBA</td>
<td>glDrawPixels(), glReadPixels(), glTexImage1D(), glTexImage2D(), glGetTexImage()</td>
</tr>
<tr>
<td>GL_RGBA_MODE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_RIGHT</td>
<td>glDrawBuffer(), glReadBuffer()</td>
</tr>
<tr>
<td>GL_S</td>
<td>gl TexGen*(), glGetTexGen*()</td>
</tr>
<tr>
<td>GL_SCISSOR_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_SCISSOR_BOX</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_SCISSOR_TEST</td>
<td>glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_SELECT</td>
<td>glRenderMode()</td>
</tr>
<tr>
<td>GL_SET</td>
<td>glLogicOp()</td>
</tr>
<tr>
<td>GL_SHININESS</td>
<td>glMaterial*(), glGetMaterial*()</td>
</tr>
<tr>
<td>GL_SHADE_MODE</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_SHORT</td>
<td>glCallLists(), glDrawPixels(), glReadPixels(), glTexImage1D(), glTexImage2D(), glGetTexImage()</td>
</tr>
<tr>
<td>GL_SMOOTH</td>
<td>glShadeModel()</td>
</tr>
<tr>
<td>GL_SPECULAR</td>
<td>glLight*, glGetLight*, glMaterial*, glGetMaterial*, glColorMaterial*</td>
</tr>
<tr>
<td>GL_SPHERE_MAP</td>
<td>gl TexGen*(), glGetTexGen*()</td>
</tr>
<tr>
<td>GL_SPOT_CUTOFF, GL_SPOT_DIRECTION, GL_SPOT_EXPONENT</td>
<td>glLight*, glGetLight*()</td>
</tr>
<tr>
<td>GL_SRC_ALPHA, GL_SRC_ALPHA_SATURATE, GL_SRC_COLOR</td>
<td>glBlendFunc()</td>
</tr>
<tr>
<td>Constant</td>
<td>Associated Commands</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>GL_STACK_OVERFLOW, GL_STACK_UNDERFLOW</td>
<td>glGetError()</td>
</tr>
<tr>
<td>GL_STENCIL</td>
<td>glCopyPixels()</td>
</tr>
<tr>
<td>GL_STENCIL_BITS</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_STENCIL_BUFFER_BIT</td>
<td>glClear(), glPushAttrib()</td>
</tr>
<tr>
<td>GL_STENCIL_INDEX</td>
<td>glDrawPixels(), glReadPixels(), glTexImage1D(), glTexImage2D(), glGetTexImage()</td>
</tr>
<tr>
<td>GL_STENCIL_CLEAR_VALUE, GL_STENCIL_FAIL,</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_STENCIL_FUNC, GL_STENCIL_PASS_DEPTH_FAIL,</td>
<td></td>
</tr>
<tr>
<td>GL_STENCIL_PASS_DEPTH_PASS, GL_STENCIL_REF</td>
<td>glDisable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_STENCIL_TEST</td>
<td>glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_STENCIL_VALUE_MASK, GL_STENCIL_WRITEMASK</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_STEREO</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_SUBPIXEL_BITS</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_T</td>
<td>glTexGen*, glGetTexGen*()</td>
</tr>
<tr>
<td>GL_TEXTURE</td>
<td>glMatrixMode()</td>
</tr>
<tr>
<td>GL_TEXTURE_1D</td>
<td>glTexImage1D(), glGetTexImage(), glTexParameteri*, glGetTexParameter*, glGetTexLevelParameter*, glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_TEXTURE_2D</td>
<td>glTexImage2D(), glGetTexImage(), glTexParameteri*, glGetTexParameter*, glGetTexLevelParameter*, glEnable(), glIsEnabled(), glGet*()</td>
</tr>
<tr>
<td>GL_TEXTURE_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_TEXTURE_BORDER</td>
<td>glGetTexParameter*, glGetTexLevelParameter*()</td>
</tr>
<tr>
<td>Constant</td>
<td>Associated Commands</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>GL_TEXTURE_BORDER_COLOR</td>
<td>glTexParameter*(), glGetTexParameter*(),</td>
</tr>
<tr>
<td></td>
<td>glGetTexLevelParameter*()</td>
</tr>
<tr>
<td>GL_TEXTURE_COMPONENTS</td>
<td>glGetTexParameter*(), glGetTexLevelParameter*()</td>
</tr>
<tr>
<td>GL_TEXTURE_ENV, GL_TEXTURE_ENV_COLOR,</td>
<td>glTexParameteri*, glGetTexParameter*(),</td>
</tr>
<tr>
<td>GL_TEXTURE_ENV_MODE</td>
<td>glGetTexLevelParameter*()</td>
</tr>
<tr>
<td>GL_TEXTURE_GEN_MODE</td>
<td>glTexParameteri*, glGetTexParameter*(),</td>
</tr>
<tr>
<td>GL_TEXTURE_GEN_Q, GL_TEXTURE_GEN_R,</td>
<td>glGetTexParameter*(), glGetTexLevelParameter*()</td>
</tr>
<tr>
<td>GL_TEXTURE_GEN_S, GL_TEXTURE_GEN_T</td>
<td></td>
</tr>
<tr>
<td>GL_TEXTURE_HEIGHT</td>
<td>glGetTexParameter*(), glGetTexLevelParameter*()</td>
</tr>
<tr>
<td>GL_TEXTURE_MAG_FILTER</td>
<td>glTexParameteri*, glGetTexParameter*(),</td>
</tr>
<tr>
<td></td>
<td>glGetTexLevelParameter*()</td>
</tr>
<tr>
<td>GL_TEXTURE_MATRIX</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_TEXTURE_MIN_FILTER</td>
<td>glTexParameteri*, glGetTexParameter*(),</td>
</tr>
<tr>
<td></td>
<td>glGetTexLevelParameter*()</td>
</tr>
<tr>
<td>GL_TEXTURE_STACK_DEPTH</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_TEXTURE_WIDTH</td>
<td>glTexParameteri*, glGetTexParameter*(),</td>
</tr>
<tr>
<td></td>
<td>glGetTexLevelParameter*()</td>
</tr>
<tr>
<td>GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T</td>
<td>glTexParameteri*, glGetTexParameter*(),</td>
</tr>
<tr>
<td></td>
<td>glGetTexLevelParameter*()</td>
</tr>
<tr>
<td>GL_TRANSFORM_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_TRIANGLES, GL_TRIANGLE_FAN,</td>
<td>glBegin()</td>
</tr>
<tr>
<td>GL_TRIANGLE_STRIP</td>
<td></td>
</tr>
<tr>
<td>GL_TRUE</td>
<td>glColorMask(), glGet*, glIsEnabled(), glIsList()</td>
</tr>
<tr>
<td>GL_UNSIGNED_BYTE, GL_UNSIGNED_INT,</td>
<td>glPixelStore*, glGet*()</td>
</tr>
<tr>
<td>GL_UNSIGNED_SHORT</td>
<td>glCallLists(), glDrawPixels(), glReadPixels(),</td>
</tr>
<tr>
<td></td>
<td>glTexImage1D(),</td>
</tr>
</tbody>
</table>
### Chapter 6. GLU Reference Pages

This chapter contains the reference pages, in alphabetical order, for all the routines comprising the OpenGL Utility Library (GLU).

**gluBeginCurve**

**NAME**

`gluBeginCurve`, `gluEndCurve` - delimit a NURBS curve definition

**C SPECIFICATION**

```c
void gluBeginCurve( GLUnurbsObj *nobj )
void gluEndCurve( GLUnurbsObj *nobj )
```

**PARAMETERS**

`nobj`  Specifies the NURBS object (created with `gluNewNurbsRenderer`).

**DESCRIPTION**

Use `gluBeginCurve` to mark the beginning of a NURBS curve definition. After calling `gluBeginCurve`, make one or more calls to `gluNurbsCurve` to define the attributes of the curve. Exactly one of the calls to `gluNurbsCurve` must have a curve type of `GL_MAP1_VERTEX_3` or `GL_MAP1_VERTEX_4`. To mark the end of the NURBS curve definition, call `gluEndCurve`.

OpenGL evaluators are used to render the NURBS curve as a series of line segments. Evaluator state is preserved during rendering with `glPushAttrib(GL_EVAL_BIT)` and `glPopAttrib()`. See the "glPushAttrib" reference page for details on exactly what state these calls preserve.

---

<table>
<thead>
<tr>
<th>Constant</th>
<th>Associated Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_VENDOR, GL_VERSION</td>
<td>glGetString()</td>
</tr>
<tr>
<td>GL_VIEWPORT</td>
<td>glGet*()</td>
</tr>
<tr>
<td>GL_VIEWPORT_BIT</td>
<td>glPushAttrib()</td>
</tr>
<tr>
<td>GL_XOR</td>
<td>glLogicOp()</td>
</tr>
<tr>
<td>GL_ZERO</td>
<td>glBlendFunc(), glStencilOp()</td>
</tr>
<tr>
<td>GL_ZOOM_X, GL_ZOOM_Y</td>
<td>glGet*()</td>
</tr>
</tbody>
</table>
EXAMPLE

The following commands render a textured NURBS curve with normals; texture coordinates and normals are also specified as NURBS curves:

```c
gluBeginCurve(nobj);
    gluNurbsCurve(nobj, ..., GL_MAP1_TEXTURE_COORD_2);
    gluNurbsCurve(nobj, ..., GL_MAP1_NORMAL);
    gluNurbsCurve(nobj, ..., GL_MAP1_VERTEX_4);
gluEndCurve(nobj);
```

SEE ALSO

"gluBeginSurface", "gluBeginTrim", "gluNewNurbsRenderer", "gluNurbsCurve", 
glPopAttrib, "glPushAttrib"

gluBeginPolygon

NAME

`gluBeginPolygon`, `gluEndPolygon` - delimit a polygon description

C SPECIFICATION

```c
void gluBeginPolygon( GLUtriangulatorObj *tobj )
void gluEndPolygon( GLUtriangulatorObj *tobj )
```

PARAMETERS

`tobj`  Specifies the tessellation object (created with `gluNewTess`).

DESCRIPTION

`gluBeginPolygon` and `gluEndPolygon` delimit the definition of a nonconvex polygon. To define such a polygon, first call `gluBeginPolygon`. Then define the contours of the polygon by calling `gluTessVertex` for each vertex and `gluNextContour` to start each new contour. Finally, call `gluEndPolygon` to signal the end of the definition. See the "gluTessVertex" and "gluNextContour" reference pages for more details.

Once `gluEndPolygon` is called, the polygon is tessellated, and the resulting triangles are described through callbacks. See "gluTessCallback" for descriptions of the callback functions.
EXAMPLE

A quadrilateral with a triangular hole in it can be described like this:

```c
gluBeginPolygon(tobj);
    gluTessVertex(tobj, v1, v1);
    gluTessVertex(tobj, v2, v2);
    gluTessVertex(tobj, v3, v3);
    gluTessVertex(tobj, v4, v4);
    gluNextContour(tobj, GLU_INTERIOR);
    gluTessVertex(tobj, v5, v5);
    gluTessVertex(tobj, v6, v6);
    gluTessVertex(tobj, v7, v7);
gluEndPolygon(tobj);
```

SEE ALSO

"gluNewTess", "gluNextContour", "gluTessCallback", "gluTessVertex"

gluBeginSurface

NAME

gluBeginSurface, gluEndSurface - delimit a NURBS surface definition

C SPECIFICATION

```c
void gluBeginSurface( GLUnurbsObj *nobj )
void gluEndSurface( GLUnurbsObj *nobj )
```

PARAMETERS

nobj Specifies the NURBS object (created with gluNewNurbsRenderer).

DESCRIPTION

Use gluBeginSurface to mark the beginning of a NURBS surface definition. After calling gluBeginSurface, make one or more calls to gluNurbsSurface to define the attributes of the surface. Exactly one of these calls to gluNurbsSurface must have a surface type of GL_MAP2_VERTEX_3 or GL_MAP2_VERTEX_4. To mark the end of the NURBS surface definition, call gluEndSurface.
Trimming of NURBS surfaces is supported with `gluBeginTrim`, `gluPwlCurve`, `gluNurbsCurve`, and `gluEndTrim`. Refer to the `gluBeginTrim` reference page for details.

OpenGL evaluators are used to render the NURBS surface as a set of polygons. Evaluator state is preserved during rendering with `glPushAttrib(GL_EVAL_BIT)` and `glPopAttrib()`. See the "glPushAttrib" reference page for details on exactly what state these calls preserve.

EXAMPLE

The following commands render a textured NURBS surface with normals; the texture coordinates and normals are also described as NURBS surfaces:

```c
  glBeginSurface(nobj);
  gluNurbsSurface(nobj, ..., GL_MAP2_TEXTURE_COORD_2);
  gluNurbsSurface(nobj, ..., GL_MAP2_NORMAL);
  gluNurbsSurface(nobj, ..., GL_MAP2_VERTEX_4);
  glEndSurface(nobj);
```

SEE ALSO

"gluBeginCurve", "gluBeginTrim", "gluNewNurbsRenderer", "gluNurbsCurve", "gluNurbsSurface", "gluPwlCurve"

**gluBeginTrim**

NAME

`gluBeginTrim`, `gluEndTrim` - delimit a NURBS trimming loop definition

C SPECIFICATION

```c
void gluBeginTrim( GLUnurbsObj *nobj )
void gluEndTrim( GLUnurbsObj *nobj )
```

PARAMETERS

`nobj`  Specifies the NURBS object (created with `gluNewNurbsRenderer`).

DESCRIPTION
Use **gluBeginTrim** to mark the beginning of a trimming loop, and **gluEndTrim** to mark the end of a trimming loop. A trimming loop is a set of oriented curve segments (forming a closed curve) that define boundaries of a NURBS surface. You include these trimming loops in the definition of a NURBS surface, between calls to **gluBeginSurface** and **gluEndSurface**.

The definition for a NURBS surface can contain many trimming loops. For example, if you wrote a definition for a NURBS surface that resembled a rectangle with a hole punched out, the definition would contain two trimming loops. One loop would define the outer edge of the rectangle; the other would define the hole punched out of the rectangle. The definitions of each of these trimming loops would be bracketed by a **gluBeginTrim**/**gluEndTrim** pair.

The definition of a single closed trimming loop can consist of multiple curve segments, each described as a piecewise linear curve (see "**gluPwlCurve**") or as a single NURBS curve (see "**gluNurbsCurve**"), or as a combination of both in any order. The only library calls that can appear in a trimming loop definition (between the calls to **gluBeginTrim** and **gluEndTrim**) are **gluPwlCurve** and **gluNurbsCurve**.

The area of the NURBS surface that is displayed is the region in the domain to the left of the trimming curve as the curve parameter increases. Thus, the retained region of the NURBS surface is inside a counterclockwise trimming loop and outside a clockwise trimming loop. For the rectangle mentioned earlier, the trimming loop for the outer edge of the rectangle runs counterclockwise, while the trimming loop for the punched-out hole runs clockwise.

If you use more than one curve to define a single trimming loop, the curve segments must form a closed loop (that is, the endpoint of each curve must be the starting point of the next curve, and the endpoint of the final curve must be the starting point of the first curve). If the endpoints of the curve are sufficiently close together but not exactly coincident, they will be coerced to match. If the endpoints are not sufficiently close, an error results (see "**gluNurbsCallback**").

If a trimming loop definition contains multiple curves, the direction of the curves must be consistent (that is, the inside must be to the left of all of the curves). Nested trimming loops are legal as long as the curve orientations alternate correctly. Trimming curves cannot be self-intersecting, nor can they intersect one another (or an error results).

If no trimming information is given for a NURBS surface, the entire surface is drawn.

**EXAMPLE**

This code fragment defines a trimming loop that consists of one piecewise linear curve, and two NURBS curves:
gluBeginTrim(nobj);
    gluPwlCurve(..., GLU_MAP1_TRIM_2);
    gluNurbsCurve(..., GLU_MAP1_TRIM_2);
    gluNurbsCurve(..., GLU_MAP1_TRIM_3);
    gluEndTrim(nobj);

SEE ALSO

"gluBeginSurface", "gluNewNurbsRenderer", "gluNurbsCallback", "gluNurbsCurve", "gluPwlCurve"

**gluBuild1DMipmaps**

**NAME**

**gluBuild1DMipmaps** - create 1-D mipmaps

**C SPECIFICATION**

```c
int gluBuild1DMipmaps( GLenum target, GLint components, GLint width, GLenum format, GLenum type, void *data )
```

**PARAMETERS**

- **target** Specifies the target texture. Must be **GL_TEXTURE_1D**.
- **components** Specifies the number of color components in the texture. Must be 1, 2, 3, or 4.
- **width** Specifies the width of the texture image.
- **format** Specifies the format of the pixel data. Must be one of **GL_COLOR_INDEX, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_LUMINANCE, and GL_LUMINANCE_ALPHA**.
- **type** Specifies the data type for **data**. Must be one of **GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, or GL_FLOAT**.
- **data** Specifies a pointer to the image data in memory.

**DESCRIPTION**

**gluBuild1DMipmaps** obtains the input image and generates all mipmap images (using **gluScaleImage**) so that the input image can be used as a mipmaped texture image. **glTexImage1D** is then called to load each of the images. If the width of the input image
is not a power of two, then the image is scaled to the nearest power of two before the
mipmaps are generated.

A return value of zero indicates success. Otherwise, a GLU error code is returned (see
"gluErrorString").

Please refer to the glTexImage1D reference page for a description of the acceptable
values for the format parameter. See the "glDrawPixels" reference page for a description
of the acceptable values for the type parameter.

SEE ALSO

"glTexImage1D", "gluBuild2DMipmaps", "gluErrorString", "gluScaleImage"

gluBuild2DMipmaps

NAME

gluBuild2DMipmaps - create 2-D mipmaps

C SPECIFICATION

int gluBuild2DMipmaps( GLenum target, GLint components, GLint width, GLint
height, GLenum format, GLenum type, void *data )

PARAMETERS

target Specifies the target texture. Must be GL_TEXTURE_2D.

components Specifies the number of color components in the texture. Must be 1, 2, 3,
or 4.

width, height Specifies the width and height, respectively, of the texture image.

format Specifies the format of the pixel data. Must be one of:
GL_COLOR_INDEX, GL_RED, GL_GREEN, GL_BLUE,
GL_ALPHA, GL_RGB, GL_RGBA, GL_LUMINANCE, and
GL_LUMINANCE_ALPHA.

type Specifies the data type for data. Must be one of:
GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP,
GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT,
GL_INT, or GL_FLOAT.

data Specifies a pointer to the image data in memory.

DESCRIPTION
**gluBuild2DMipmaps** obtains the input image and generates all mipmap images (using **gluScaleImage**) so that the input image can be used as a mipmapped texture image. **glTexImage2D** is then called to load each of the images. If the dimensions of the input image are not powers of two, then the image is scaled so that both the width and height are powers of two before the mipmaps are generated.

A return value of 0 indicates success. Otherwise, a GLU error code is returned (see "gluErrorString").

Please refer to the **glTexImage1D** reference page for a description of the acceptable values for the **format** parameter. See the "glDrawPixels" reference page for a description of the acceptable values for the **type** parameter.

**SEE ALSO**

"glDrawPixels", "glTexImage1D", "glTexImage2D", "gluBuild1DMipmaps", "gluErrorString", "gluScaleImage"

gluCylinder

**NAME**

gluCylinder - draw a cylinder

**C SPECIFICATION**

```c
void gluCylinder( GLUquadricObj *qobj, GLdouble baseRadius, GLdouble topRadius, GLdouble height, GLint slices, GLint stacks )
```

**PARAMETERS**

- **qobj** Specifies the quadrics object (created with **gluNewQuadric**).
- **baseRadius** Specifies the radius of the cylinder at \( z = 0 \).
- **topRadius** Specifies the radius of the cylinder at \( z = \text{height} \).
- **height** Specifies the height of the cylinder.
- **slices** Specifies the number of subdivisions around the z axis.
- **stacks** Specifies the number of subdivisions along the z axis.

**DESCRIPTION**

**gluCylinder** draws a cylinder oriented along the z axis. The base of the cylinder is placed at \( z = 0 \), and the top at \( z = \text{height} \). Like a sphere, a cylinder is subdivided around the z axis into slices, and along the z axis into stacks.
Note that if topRadius is set to zero, then this routine will generate a cone.

If the orientation is set to GLU_OUTSIDE (with gluQuadricOrientation), then any generated normals point away from the z axis. Otherwise, they point toward the z axis.

If texturing is turned on (with gluQuadricTexture), then texture coordinates are generated so that \( t \) ranges linearly from 0.0 at \( z = 0 \) to 1.0 at \( z = \text{height} \), and \( s \) ranges from 0.0 at the +y axis, to 0.25 at the +x axis, to 0.5 at the -y axis, to 0.75 at the -x axis, and back to 1.0 at the +y axis.

SEE ALSO
"gluDisk", "gluNewQuadric", "gluPartialDisk", "gluQuadricTexture", "gluSphere"

**gluDeleteNurbsRenderer**

NAME

gluDeleteNurbsRenderer - destroy a NURBS object

C SPECIFICATION

```c
void gluDeleteNurbsRenderer( GLUnurbsObj *nobj )
```

PARAMETERS

*nojb* Specifies the NURBS object to be destroyed (created with gluNewNurbsRenderer).

DESCRIPTION

gluDeleteNurbsRenderer destroys the NURBS object and frees any memory used by it. Once gluDeleteNurbsRenderer has been called, nojb cannot be used again.

SEE ALSO

"gluNewNurbsRenderer"

**gluDeleteQuadric**

NAME

gluDeleteQuadric - destroy a quadrics object
C SPECIFICATION

void gluDeleteQuadric( GLUquadricObj *state )

PARAMETERS

state Specifies the quadrics object to be destroyed (created with gluNewQuadric).

DESCRIPTION

gluDeleteQuadric destroys the quadrics object and frees any memory used by it. Once gluDeleteQuadric has been called, state cannot be used again.

SEE ALSO

"gluNewQuadric"

gluDeleteTess

NAME

gluDeleteTess - destroy a tessellation object

C SPECIFICATION

void gluDeleteTess( GLUtriangulatorObj *tobj )

PARAMETERS

tobj Specifies the tessellation object to destroy (created with gluNewTess).

DESCRIPTION

gluDeleteTess destroys the indicated tessellation object and frees any memory that it used.

SEE ALSO

"gluBeginPolygon", "gluNewTess", "gluTessCallback"

gluDisk

NAME
gluDisk - draw a disk

C SPECIFICATION

void gluDisk( GLUquadricObj *qobj, GLdouble innerRadius, GLdouble outerRadius, GLint slices, GLint loops )

PARAMETERS

qobj Specifies the quadrics object (created with gluNewQuadric).
innerRadius Specifies the inner radius of the disk (may be 0).
outerRadius Specifies the outer radius of the disk.
slices Specifies the number of subdivisions around the z axis.
loops Specifies the number of concentric rings about the origin into which the disk is subdivided.

DESCRIPTION

gluDisk renders a disk on the z = 0 plane. The disk has a radius of outerRadius, and contains a concentric circular hole with a radius of innerRadius. If innerRadius is 0, then no hole is generated. The disk is subdivided around the z axis into slices (like pizza slices), and also about the z axis into rings (as specified by slices and loops, respectively).

With respect to orientation, the +z side of the disk is considered to be "outside" (see "gluQuadricOrientation" ). This means that if the orientation is set to GLU_OUTSIDE, then any normals generated point along the +z axis. Otherwise, they point along the -z axis.

If texturing is turned on (with gluQuadricTexture), texture coordinates are generated linearly such that where \( r = \text{outerRadius} \), the value at \((r, 0, 0)\) is \((1, 0.5)\), at \((0, r, 0)\) it is \((0.5, 1)\), at \((-r, 0, 0)\) it is \((0, 0.5)\), and at \((0, -r, 0)\) it is \((0.5, 0)\).

SEE ALSO

"gluCylinder", "gluNewQuadric", "gluPartialDisk", "gluQuadricOrientation", "gluQuadricTexture", "gluSphere"

 gluErrorString

NAME

 gluErrorString - produce an error string from an OpenGL or GLU error code

C SPECIFICATION
const GLubyte* gluErrorString ( GLenum errorCode )

PARAMETERS

errorCode Specifies an OpenGL or GLU error code.

DESCRIPTION

gluErrorString produces an error string from an OpenGL or GLU error code. The string is in an ISO Latin 1 format. For example, gluErrorString(GL_OUT_OF_MEMORY) returns the string out of memory.

The standard GLU error codes are GLU_INVALID_ENUM, GLU_INVALID_VALUE, and GLU_OUT_OF_MEMORY. Certain other GLU functions can return specialized error codes through callbacks. Refer to the glGetError reference page for the list of OpenGL error codes.

SEE ALSO

"glGetError", "gluNurbsCallback", "gluQuadricCallback", "gluTessCallback"

gluGetNurbsProperty

NAME

gluGetNurbsProperty - get a NURBS property

C SPECIFICATION

void gluGetNurbsProperty( GLUnurbsObj *nobj, GLenum property, GLfloat *value )

PARAMETERS

nobj Specifies the NURBS object (created with gluNewNurbsRenderer).

property Specifies the property whose value is to be fetched. Valid values are GLU_CULLING, GLU_SAMPLING_TOLERANCE, GLU_DISPLAY_MODE, and GLU_AUTO_LOAD_MATRIX.

value Specifies a pointer to the location into which the value of the named property is written.

DESCRIPTION
**gluGetNurbsProperty** is used to retrieve properties stored in a NURBS object. These properties affect the way that NURBS curves and surfaces are rendered. Please refer to the **gluNurbsProperty** reference page for information about what the properties are and what they do.

**SEE ALSO**

"gluNewNurbsRenderer", "gluNurbsProperty"

**gluLoadSamplingMatrices**

**NAME**

**gluLoadSamplingMatrices** - load NURBS sampling and culling matrices

**C SPECIFICATION**

```c
void gluLoadSamplingMatrices( GLUnurbsObj *nobj, const GLfloat modelMatrix[16],
                          const GLfloat projMatrix[16], const GLint viewport[4]);
```

**PARAMETERS**

- `nobj` Specifies the NURBS object (created with **gluNewNurbsRenderer**).
- `modelMatrix` Specifies a modelview matrix (as from a **glGetFloatv** call).
- `projMatrix` Specifies a projection matrix (as from a **glGetFloatv** call).
- `viewport` Specifies a viewport (as from a **glGetIntegerv** call).

**DESCRIPTION**

**gluLoadSamplingMatrices** uses `modelMatrix`, `projMatrix`, and `viewport`; to recompute the sampling and culling matrices stored in `nobj`. The sampling matrix determines how finely a NURBS curve or surface must be tessellated to satisfy the sampling tolerance (as determined by the **GLU_SAMPLING_TOLERANCE** property). The culling matrix is used in deciding if a NURBS curve or surface should be culled before rendering (when the **GLU_CULLING** property is turned on).

**gluLoadSamplingMatrices** is necessary only if the **GLU_AUTO_LOAD_MATRIX** property is turned off (see "**gluNurbsProperty**" ). Although it can be convenient to leave the **GLU_AUTO_LOAD_MATRIX** property turned on, there can be a performance penalty for doing so. (A round trip to the OpenGL server is needed to fetch the current values of the modelview matrix, projection matrix, and viewport.)

**SEE ALSO**
gluLookAt

NAME

gluLookAt - define a viewing transformation

C SPECIFICATION

void gluLookAt( GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz )

PARAMETERS

eyex, eyey, eyez
  Specifies the position of the eye point.

centerx, centery, centerz
  Specifies the position of the reference point.

upx, upy, upz
  Specifies the direction of the up vector.

DESCRIPTION

gluLookAt creates a viewing matrix derived from an eye point, a reference point indicating the center of the scene, and an up vector. The matrix maps the reference point to the negative z axis and the eye point to the origin, so that, when a typical projection matrix is used, the center of the scene maps to the center of the viewport. Similarly, the direction described by the up vector projected onto the viewing plane is mapped to the positive y axis so that it points upward in the viewport. The up vector must not be parallel to the line of sight from the eye to the reference point.

The matrix generated by gluLookAt postmultiplies the current matrix.

SEE ALSO

"gluFrustum", "gluPerspective"

 gluNewNurbsRenderer

NAME

 gluNewNurbsRenderer - create a NURBS object
C SPECIFICATION

GLUnurbsObj* gluNewNurbsRenderer( void )

DESCRIPTION

gluNewNurbsRenderer creates and returns a pointer to a new NURBS object. This object must be referred to when calling NURBS rendering and control functions. A return value of zero means that there is not enough memory to allocate the object.

SEE ALSO

"gluBeginCurve", "gluBeginSurface", "gluBeginTrim", "gluDeleteNurbsRenderer", "gluNurbsCallback", "gluNurbsProperty"

gluNewQuadric

NAME

gluNewQuadric - create a quadrics object

C SPECIFICATION

GLUquadricObj* gluNewQuadric( void )

DESCRIPTION

gluNewQuadric creates and returns a pointer to a new quadrics object. This object must be referred to when calling quadrics rendering and control functions. A return value of zero means that there is not enough memory to allocate the object.

SEE ALSO

"gluCylinder", "gluDeleteQuadric", "gluDisk", "gluPartialDisk", "gluQuadricCallback", "gluQuadricDrawStyle", "gluQuadricNormals", "gluQuadricOrientation", "gluQuadricTexture", "gluSphere"

gluNewTess

NAME

gluNewTess - create a tessellation object

C SPECIFICATION
GLUtriangulatorObj* gluNewTess( void )

DESCRIPTION

gluNewTess creates and returns a pointer to a new tessellation object. This object must be referred to when calling tessellation functions. A return value of zero means that there is not enough memory to allocate the object.

SEE ALSO

"gluBeginPolygon", "gluDeleteTess", "gluTessCallback"

gluNextContour

NAME

gluNextContour - mark the beginning of another contour

C SPECIFICATION

void gluNextContour( GLUtriangulatorObj *tobj, GLenum type )

PARAMETERS

tobj Specifies the tessellation object (created with gluNewTess).
type Specifies the type of the contour being defined. Valid values are GLU_EXTERIOR, GLU_INTERIOR, GLU_UNKNOWN, GLU_CCW, and GLU_CW.

DESCRIPTION

gluNextContour is used in describing polygons with multiple contours. After the first contour has been described through a series of gluTessVertex calls, a gluNextContour call indicates that the previous contour is complete and that the next contour is about to begin. Another series of gluTessVertex calls is then used to describe the new contour. This process can be repeated until all contours have been described.

type defines what type of contour follows. The legal contour types are as follows:

GLU_EXTERIOR An exterior contour defines an exterior boundary of the polygon.
GLU_INTERIOR An interior contour defines an interior boundary of the polygon (such as a hole).
GLU_UNKNOWN
An unknown contour is analyzed by the library to determine if it is interior or exterior.

GLU_CCW, GLU_CW
The first GLU_CCW or GLU_CW contour defined is considered to be exterior. All other contours are considered to be exterior if they are oriented in the same direction (clockwise or counterclockwise) as the first contour, and interior if they are not. If one contour is of type GLU_CCW or GLU_CW, then all contours must be of the same type (if they are not, then all GLU_CCW and GLU_CW contours will be changed to GLU_UNKNOWN). Note that there is no real difference between the GLU_CCW and GLU_CW contour types.

gluNextContour can be called before the first contour is described to define the type of the first contour. If gluNextContour is not called before the first contour, then the first contour is marked GLU_EXTERIOR.

EXAMPLE
A quadrilateral with a triangular hole in it can be described as follows:

```c
gluBeginPolygon(tobj);
    gluTessVertex(tobj, v1, v1);
    gluTessVertex(tobj, v2, v2);
    gluTessVertex(tobj, v3, v3);
    gluTessVertex(tobj, v4, v4);
    gluNextContour(tobj, GLU_INTERIOR);
    gluTessVertex(tobj, v5, v5);
    gluTessVertex(tobj, v6, v6);
    gluTessVertex(tobj, v7, v7);
gluEndPolygon(tobj);
```

SEE ALSO
"gluBeginPolygon", "gluNewTess", "gluTessCallback", "gluTessVertex"

gluNurbsCallback

NAME
gluNurbsCallback - define a callback for a NURBS object

C SPECIFICATION
void gluNurbsCallback( GLUnurbsObj *nobj, GLenum which, void (*fn)() )

PARAMETERS

nobj    Specifies the NURBS object (created with gluNewNurbsRenderer).
which   Specifies the callback being defined. The only valid value is GLU_ERROR.
fn      Specifies the function that the callback calls.

DESCRIPTION

gluNurbsCallback is used to define a callback to be used by a NURBS object. If the specified callback is already defined, then it is replaced. If fn is NULL, then any existing callback is erased.

The one legal callback is GLU_ERROR:

GLU_ERROR The error function is called when an error is encountered. Its single argument is of type GLenum, and it indicates the specific error that occurred. There are 37 errors unique to NURBS named GLU_NURBS_ERROR1 through GLU_NURBS_ERROR37. Character strings describing these errors can be retrieved with gluErrorString.

SEE ALSO

"gluErrorString", "gluNewNurbsRenderer"

gluNurbsCurve

NAME

gluNurbsCurve - define the shape of a NURBS curve

C SPECIFICATION

void gluNurbsCurve( GLUnurbsObj *nobj, GLint nknots, GLfloat *knot, GLint stride, GLfloat *ctlarray, GLint order, GLenum type )

PARAMETERS

nobj    Specifies the NURBS object (created with gluNewNurbsRenderer).
nknots  Specifies the number of knots in knot. nknots equals the number of control points plus the order.
**knot**  Specifies an array of *nknots* nondecreasing knot values.

**stride**  Specifies the offset (as a number of single-precision floating-point values) between successive curve control points.

**ctlarray**  Specifies a pointer to an array of control points. The coordinates must agree with *type*, specified below.

**order**  Specifies the order of the NURBS curve. *order* equals degree + 1, hence a cubic curve has an order of 4.

**type**  Specifies the type of the curve. If this curve is defined within a `gluBeginCurve`/`gluEndCurve` pair, then the type can be any of the valid one-dimensional evaluator types (such as `GL_MAP1_VERTEX_3` or `GL_MAP1_COLOR_4`). Between a `gluBeginTrim`/`gluEndTrim` pair, the only valid types are `GLU_MAP1_TRIM_2` and `GLU_MAP1_TRIM_3`.

**DESCRIPTION**

Use `gluNurbsCurve` to describe a NURBS curve.

When `gluNurbsCurve` appears between a `gluBeginCurve`/`gluEndCurve` pair, it is used to describe a curve to be rendered. Positional, texture, and color coordinates are associated by presenting each as a separate `gluNurbsCurve` between a `gluBeginCurve`/`gluEndCurve` pair. No more than one call to `gluNurbsCurve` for each of color, position, and texture data can be made within a single `gluBeginCurve`/`gluEndCurve` pair. Exactly one call must be made to describe the position of the curve (a *type* of `GL_MAP1_VERTEX_3` or `GL_MAP1_VERTEX_4`).

When `gluNurbsCurve` appears between a `gluBeginTrim`/`gluEndTrim` pair, it is used to describe a trimming curve on a NURBS surface. If *type* is `GLU_MAP1_TRIM_2`, then it describes a curve in two-dimensional (u and v) parameter space. If it is `GLU_MAP1_TRIM_3`, then it describes a curve in two-dimensional homogeneous (u, v, and w) parameter space. See the "`gluBeginTrim`" reference page for more discussion about trimming curves.

**EXAMPLE**

The following commands render a textured NURBS curve with normals:

```c
    gluBeginCurve(nobj);
    gluNurbsCurve(nobj, ..., GL_MAP1_TEXTURE_COORD_2);
    gluNurbsCurve(nobj, ..., GL_MAP1_NORMAL);
    gluNurbsCurve(nobj, ..., GL_MAP1_VERTEX_4);
    gluEndCurve(nobj);
```

**SEE ALSO**
gluNurbsProperty

NAME

gluNurbsProperty - set a NURBS property

C SPECIFICATION

void gluNurbsProperty( GLUnurbsObj *nobj, GLenum property, GLfloat value )

PARAMETERS

nobj Specifies the NURBS object (created with gluNewNurbsRenderer).
property Specifies the property to be set. Valid values are
GLU_SAMPLING_TOLERANCE, GLU_DISPLAY_MODE,
GLU_CULLING, and GLU_AUTO_LOAD_MATRIX.
value Specifies the value to which to set the indicated property.

DESCRIPTION

gluNurbsProperty is used to control properties stored in a NURBS object. These properties affect the way that a NURBS curve is rendered. The legal values for property are as follows:

GLU_SAMPLING_TOLERANCE

value specifies the maximum length, in pixels, of line segments or edges of polygons used to render NURBS curves or surfaces. The NURBS code is conservative when rendering a curve or surface, so the actual length can be somewhat shorter. The default value is 50.0 pixels.

GLU_DISPLAY_MODE

value defines how a NURBS surface should be rendered. value can be set to GLU_FILL,
GLU_OUTLINE_POLYGON, or
GLU_OUTLINE_PATCH. When set to
GLU_FILL, the surface is rendered as a set of polygons. GLU_OUTLINE_POLYGON instructs the NURBS library to draw only the outlines of the polygons created by tessellation. GLU_OUTLINE_PATCH causes just the outlines of patches and trim curves defined by the
user to be drawn. The default value is GLU_FILL.

**GLU_CULLING**

`value` is a Boolean value that, when set to `GL_TRUE`, indicates that a NURBS curve should be discarded prior to tessellation if its control points lie outside the current viewport. The default is `GL_FALSE` (because a NURBS curve cannot fall entirely within the convex hull of its control points).

**GLU_AUTO_LOAD_MATRIX**

`value` is a Boolean value. When set to `GL_TRUE`, the NURBS code downloads the projection matrix, the modelview matrix, and the viewport from the OpenGL server to compute sampling and culling matrices for each NURBS curve that is rendered. Sampling and culling matrices are required to determine the tesselation of a NURBS surface into line segments or polygons and to cull a NURBS surface if it lies outside of the viewport. If this mode is set to `GL_FALSE`, then the user needs to provide a projection matrix, a modelview matrix, and a viewport for the NURBS renderer to use to construct sampling and culling matrices. This can be done with the `gluLoadSamplingMatrices` function. The default for this mode is `GL_TRUE`. Changing this mode from `GL_TRUE` to `GL_FALSE` does not affect the sampling and culling matrices until `gluLoadSamplingMatrices` is called.

**SEE ALSO**

"gluGetNurbsProperty", "gluLoadSamplingMatrices", "gluNewNurbsRenderer"

**gluNurbsSurface**

**NAME**

`gluNurbsSurface` - define the shape of a NURBS surface

**C SPECIFICATION**
void gluNurbsSurface( GLUnurbsObj *nobj, GLint sknot_count, GLfloat *sknot, GLint tknot_count, GLfloat *tknot, GLint s_stride, GLint t_stride, GLfloat *ctlarray, GLint sorder, GLint torder, GLenum type )

PARAMETERS

nobj Specifies the NURBS object (created with gluNewNurbsRenderer).
sknot_count Specifies the number of knots in the parametric u direction.
sknot Specifies an array of sknot_count nondecreasing knot values in the	parametric u direction.
tknot_count Specifies the number of knots in the parametric v direction.
tknot Specifies an array of tknot_count nondecreasing knot values in the
parametric v direction.
s_stride Specifies the offset (as a number of single-precision floating point values)
between successive control points in the parametric u direction in ctlarray.
t_stride Specifies the offset (in single-precision floating-point values) between
successive control points in the parametric v direction in ctlarray.
ctlarray Specifies an array containing control points for the NURBS surface. The
offsets between successive control points in the parametric u and v
directions are given by s_stride and t_stride.
sorder Specifies the order of the NURBS surface in the parametric u direction.
The order is one more than the degree, hence a surface that is cubic in u
has a u order of 4.
torder Specifies the order of the NURBS surface in the parametric v direction.
The order is one more than the degree, hence a surface that is cubic in v
has a v order of 4.
type Specifies type of the surface. type can be any of the valid two-dimensional
evaluator types (such as GL_MAP2_VERTEX_3 or
GL_MAP2_COLOR_4).

DESCRIPTION

Use gluNurbsSurface within a NURBS (Non-Uniform Rational B-Spline) surface
definition to describe the shape of a NURBS surface (before any trimming). To mark the
beginning of a NURBS surface definition, use the gluBeginSurface command. To mark
the end of a NURBS surface definition, use the gluEndSurface command. Call
gluNurbsSurface within a NURBS surface definition only.

Positional, texture, and color coordinates are associated with a surface by presenting each
as a separate gluNurbsSurface between a gluBeginSurface gluEndSurface pair. No
more than one call to gluNurbsSurface for each of color, position, and texture data can
be made within a single gluBeginSurface gluEndSurface pair. Exactly one call must be
made to describe the position of the surface (a type of GL_MAP2_VERTEX_3 or GL_MAP2_VERTEX_4).

A NURBS surface can be trimmed by using the commands gluNurbsCurve and gluPwlCurve between calls to gluBeginTrim and gluEndTrim.

Note that a gluNurbsSurface with sknot_count knots in the u direction and tknot_count knots in the v direction with orders sorder and torder must have (sknot_count - sorder) x (tknot_count - torder) control points.

EXAMPLE

The following commands render a textured NURBS surface with normals; the texture coordinates and normals are also NURBS surfaces:

```c
gluBeginSurface(nobj);
    gluNurbsSurface(nobj, ..., GL_MAP2_TEXTURE_COORD_2);
    gluNurbsSurface(nobj, ..., GL_MAP2_NORMAL);
    gluNurbsSurface(nobj, ..., GL_MAP2_VERTEX_4);
    gluEndSurface(nobj);
```

SEE ALSO

"gluBeginSurface", "gluBeginTrim", "gluNewNurbsRenderer", "gluNurbsCurve", "gluPwlCurve"

**gluOrtho2D**

NAME

**gluOrtho2D** - define a 2-D orthographic projection matrix

C SPECIFICATION

```c
void gluOrtho2D( GLdouble left, GLdouble right, GLdouble bottom, GLdouble top )
```

PARAMETERS

- `left`, `right` Specify the coordinates for the left and right vertical clipping planes.
- `bottom`, `top` Specify the coordinates for the bottom and top horizontal clipping planes.

DESCRIPTION
gluOrtho2D sets up a two-dimensional orthographic viewing region. This is equivalent to calling glOrtho with \( \text{near} = -1 \) and \( \text{far} = 1 \).

SEE ALSO

"glOrtho" , "gluPerspective"

**gluPartialDisk**

**NAME**

gluPartialDisk - draw an arc of a disk

**C SPECIFICATION**

```c
void gluPartialDisk( GLUquadricObj *qobj, GLdouble innerRadius, GLdouble outerRadius, GLInt slices, GLInt loops, GLdouble startAngle, GLdouble sweepAngle )
```

**PARAMETERS**

- **qobj** Specifies a quadrics object (created with gluNewQuadric).
- **innerRadius** Specifies the inner radius of the partial disk (can be zero).
- **outerRadius** Specifies the outer radius of the partial disk.
- **slices** Specifies the number of subdivisions around the \( z \) axis.
- **loops** Specifies the number of concentric rings about the origin into which the partial disk is subdivided.
- **startAngle** Specifies the starting angle, in degrees, of the disk portion.
- **sweepAngle** Specifies the sweep angle, in degrees, of the disk portion.

**DESCRIPTION**

**gluPartialDisk** renders a partial disk on the \( z = 0 \) plane. A partial disk is similar to a full disk, except that only the subset of the disk from \( \text{startAngle} \) through \( \text{startAngle} + \text{sweepAngle} \) is included (where 0 degrees is along the \( +y \) axis, 90 degrees along the \( +x \) axis, 180 along the \( -y \) axis, and 270 along the \( -x \) axis).

The partial disk has a radius of \( \text{outerRadius} \), and contains a concentric circular hole with a radius of \( \text{innerRadius} \). If \( \text{innerRadius} \) is zero, then no hole is generated. The partial disk is subdivided around the \( z \) axis into slices (like pizza slices), and also about the \( z \) axis into rings (as specified by \( \text{slices} \) and \( \text{loops} \), respectively).

With respect to orientation, the \( +z \) side of the partial disk is considered to be outside (see "gluQuadricOrientation"). This means that if the orientation is set to GLU_OUTSIDE,
then any normals generated point along the +z axis. Otherwise, they point along the -z axis.

If texturing is turned on (with `gluQuadricTexture`), texture coordinates are generated linearly such that where \( r = \text{outerRadius} \), the value at \((r, 0, 0)\) is \((1, 0.5)\), at \((0, r, 0)\) it is \((0.5, 1)\), at \((-r, 0, 0)\) it is \((0, 0.5)\), and at \((0, -r, 0)\) it is \((0.5, 0)\).

**SEE ALSO**

"gluCylinder", "gluDisk", "gluNewQuadric", "gluQuadricOrientation", "gluQuadricTexture", "gluSphere"

**gluPerspective**

**NAME**

**gluPerspective** - set up a perspective projection matrix

**C SPECIFICATION**

```c
void gluPerspective( GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar )
```

**PARAMETERS**

- `fovy` Specifies the field of view angle, in degrees, in the y direction.
- `aspect` Specifies the aspect ratio that determines the field of view in the x direction. The aspect ratio is the ratio of x (width) to y (height).
- `zNear` Specifies the distance from the viewer to the near clipping plane (always positive).
- `zFar` Specifies the distance from the viewer to the far clipping plane (always positive).

**DESCRIPTION**

`gluPerspective` specifies a viewing frustum into the world coordinate system. In general, the aspect ratio in `gluPerspective` should match the aspect ratio of the associated viewport. For example, `aspect = 2.0` means the viewer's angle of view is twice as wide in x as it is in y. If the viewport is twice as wide as it is tall, it displays the image without distortion.

The matrix generated by `gluPerspective` is multiplied by the current matrix, just as if `glMultMatrix` were called with the generated matrix. To load the perspective matrix onto
the current matrix stack instead, precede the call to `gluPerspective` with a call to `glLoadIdentity`.

**SEE ALSO**

"glFrustum", "glLoadIdentity", "glMultMatrix", "gluOrtho2D"

**gluPickMatrix**

**NAME**

`gluPickMatrix` - define a picking region

**C SPECIFICATION**

```c
```

**PARAMETERS**

- `x, y` Specify the center of a picking region in window coordinates.
- `width, height` Specify the width and height, respectively, of the picking region in window coordinates.
- `viewport` Specifies the current viewport (as from a `glGetIntegerv` call).

**DESCRIPTION**

`gluPickMatrix` creates a projection matrix that can be used to restrict drawing to a small region of the viewport. This is typically useful to determine what objects are being drawn near the cursor. Use `gluPickMatrix` to restrict drawing to a small region around the cursor. Then, enter selection mode (with `glRenderMode` and rerender the scene. All primitives that would have been drawn near the cursor are identified and stored in the selection buffer.

The matrix created by `gluPickMatrix` is multiplied by the current matrix just as if `glMultMatrix` is called with the generated matrix. To effectively use the generated pick matrix for picking, first call `glLoadIdentity` to load an identity matrix onto the perspective matrix stack. Then call `gluPickMatrix`, and finally, call a command (such as `gluPerspective`) to multiply the perspective matrix by the pick matrix.

When using `gluPickMatrix` to pick NURBS, be careful to turn off the NURBS property `GLU_AUTO_LOAD_MATRIX`. If `GLU_AUTO_LOAD_MATRIX` is not turned off, then any NURBS surface rendered is subdivided differently with the pick matrix than the way it was subdivided without the pick matrix.
EXAMPLE

When rendering a scene as follows:

```c
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(...);
glMatrixMode(GL_MODELVIEW);
/* Draw the scene */
```

a portion of the viewport can be selected as a pick region like this:

```c
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPickMatrix(x, y, width, height, viewport);
gluPerspective(...);
glMatrixMode(GL_MODELVIEW);
/* Draw the scene */
```

SEE ALSO

"glGet", "glLoadIdentity", "glMultMatrix", "glRenderMode", "gluPerspective"

**gluProject**

NAME

**gluProject** - map object coordinates to window coordinates

C SPECIFICATION

```c
int gluProject(GLdouble objx, GLdouble objy, GLdouble objz, const GLdouble modelMatrix[16], const GLdouble projMatrix[16], const GLint viewport[4], GLdouble *winx, GLdouble *winy, GLdouble *winz)
```

PARAMETERS

- **objx, objy, objz**: Specify the object coordinates.
- **modelMatrix**: Specifies the current modelview matrix (as from a **glGetDoublev** call).
- **projMatrix**: Specifies the current projection matrix (as from a **glGetDoublev** call).
- **viewport**: Specifies the current viewport (as from a **glGetIntegerv** call).
winx, winy, winz

Return the computed window coordinates.

DESCRIPTION

**gluProject** transforms the specified object coordinates into window coordinates using `modelMatrix`, `projMatrix`, and `viewport`. The result is stored in `winx`, `winy`, and `winz`. A return value of `GL_TRUE` indicates success, and `GL_FALSE` indicates failure.

SEE ALSO

"glGet", "gluUnProject"

**gluPwlCurve**

NAME

**gluPwlCurve** - describe a piecewise linear NURBS trimming curve

C SPECIFICATION

```c
void gluPwlCurve( GLUnurbsObj *nobj, GLint count, GLfloat *array, GLint stride, GLenum type )
```

PARAMETERS

- `nobj` Specifies the NURBS object (created with `gluNewNurbsRenderer`).
- `count` Specifies the number of points on the curve.
- `array` Specifies an array containing the curve points.
- `stride` Specifies the offset (a number of single-precision floating-point values) between points on the curve.
- `type` Specifies the type of curve. Must be either `GLU_MAP1_TRIM_2` or `GLU_MAP1_TRIM_3`.

DESCRIPTION

**gluPwlCurve** describes a piecewise linear trimming curve for a NURBS surface. A piecewise linear curve consists of a list of coordinates of points in the parameter space for the NURBS surface to be trimmed. These points are connected with line segments to form a curve. If the curve is an approximation to a real curve, the points should be close enough that the resulting path appears curved at the resolution used in the application.
If type is \texttt{GLU\_MAP1\_TRIM\_2}, then it describes a curve in two-dimensional \((u\text{ and }v)\) parameter space. If it is \texttt{GLU\_MAP1\_TRIM\_3}, then it describes a curve in two-dimensional homogeneous \((u, v, \text{ and }w)\) parameter space. Please refer to the \texttt{gluBeginTrim} reference page for more information about trimming curves.

\textbf{SEE ALSO}

"gluBeginCurve", "gluBeginTrim", "gluNewNurbsRenderer", "gluNurbsCurve"

\textbf{gluQuadricCallback}

\textbf{NAME}

\textit{gluQuadricCallback} - define a callback for a quadrics object

\textbf{C SPECIFICATION}

\begin{verbatim}
void gluQuadricCallback( GLUquadricObj *qobj, GLenum which, void (*fn)( )
\end{verbatim}

\textbf{PARAMETERS}

- \texttt{qobj} Specifies the quadrics object (created with \texttt{gluNewQuadric}).
- \texttt{which} Specifies the callback being defined. The only valid value is \texttt{GLU\_ERROR}.
- \texttt{fn} Specifies the function to be called.

\textbf{DESCRIPTION}

\textit{gluQuadricCallback} is used to define a new callback to be used by a quadrics object. If the specified callback is already defined, then it is replaced. If \texttt{fn} is \texttt{NULL}, then any existing callback is erased.

The one legal callback is \texttt{GLU\_ERROR}:

- \texttt{GLU\_ERROR} The function is called when an error is encountered. Its single argument is of type \texttt{GLenum}, and it indicates the specific error that occurred. Character strings describing these errors can be retrieved with the \texttt{gluErrorString} call.

\textbf{SEE ALSO}

"gluErrorString", "gluNewQuadric"

\textbf{gluQuadricDrawStyle}
NAME

gluQuadricDrawStyle - specify the draw style desired for quadrics

C SPECIFICATION

void gluQuadricDrawStyle( GLUquadricObj *quadObject, GLenum drawStyle )

PARAMETERS

quadObject Specifies the quadrics object (created with gluNewQuadric).
drawStyle Specifies the desired draw style. Valid values are GLU_FILL, GLU_LINE, GLU_SILHOUETTE, and GLU_POINT.

DESCRIPTION

gluQuadricDrawStyle specifies the draw style for quadrics rendered with quadObject. The legal values are as follows:

GLU_FILL Quadrics are rendered with polygon primitives. The polygons are drawn in a counterclockwise fashion with respect to their normals (as defined with gluQuadricOrientation).
GLU_LINE Quadrics are rendered as a set of lines.
GLU_SILHOUETTE Quadrics are rendered as a set of lines, except that edges separating coplanar faces will not be drawn.
GLU_POINT Quadrics are rendered as a set of points.

SEE ALSO

"gluNewQuadric", "gluQuadricNormals", "gluQuadricOrientation", "gluQuadricTexture"

 gluQuadricNormals

NAME

gluQuadricNormals - specify what kind of normals are desired for quadrics

C SPECIFICATION

void gluQuadricNormals( GLUquadricObj *quadObject, GLenum normals )
PARAMETERS

quadObject Specifies the quadrics object (created with gluNewQuadric).

normals Specifies the desired type of normals. Valid values are GLU_NONE, GLU_FLAT, and GLU_SMOOTH.

DESCRIPTION

gluQuadricNormals specifies what kind of normals are desired for quadrics rendered with quadObject. The legal values are as follows:

GLU_NONE No normals are generated.
GLU_FLAT One normal is generated for every facet of a quadric.
GLU_SMOOTH One normal is generated for every vertex of a quadric. This is the default.

SEE ALSO

"gluNewQuadric", "gluQuadricDrawStyle", "gluQuadricOrientation", "gluQuadricTexture"

gluQuadricOrientation

NAME

gluQuadricOrientation - specify inside/outside orientation for quadrics

C SPECIFICATION

void gluQuadricOrientation( GLUquadricObj *quadObject, GLenum orientation )

PARAMETERS

quadObject Specifies the quadrics object (created with gluNewQuadric).

orientation Specifies the desired orientation. Valid values are GLU_OUTSIDE and GLU_INSIDE.

DESCRIPTION

gluQuadricOrientation specifies what kind of orientation is desired for quadrics rendered with quadObject. The orientation values are as follows:
GLU_OUTSIDE  Quadrics are drawn with normals pointing outward.
GLU_INSIDE   Normals point inward. The default is GLU_OUTSIDE.

Note that the interpretation of *outward* and *inward* depends on the quadric being drawn.

SEE ALSO

"gluNewQuadric", "gluQuadricDrawStyle", "gluQuadricNormals", "gluQuadricTexture"

**gluQuadricTexture**

**NAME**

gluQuadricTexture - specify if texturing is desired for quadrics

**C SPECIFICATION**

void gluQuadricTexture( GLUquadricObj *quadObject, GLboolean textureCoords )

**PARAMETERS**

quadObject  Specifies the quadrics object (created with gluNewQuadric).
textureCoords  Specifies a flag indicating if texture coordinates should be generated.

**DESCRIPTION**

**gluQuadricTexture** specifies if texture coordinates should be generated for quadrics rendered with *quadObject*. If the value of *textureCoords* is GL_TRUE, then texture coordinates are generated, and if *textureCoords* is GL_FALSE, they are not. The default is GL_FALSE.

The manner in which texture coordinates are generated depends upon the specific quadric rendered.

**SEE ALSO**

"gluNewQuadric", "gluQuadricDrawStyle", "gluQuadricNormals", "gluQuadricOrientation"

**gluScaleImage**
NAME

gluScaleImage - scale an image to an arbitrary size

C SPECIFICATION

```c
int gluScaleImage( GLenum format, GLint widthin, GLint heightin, GLenum typein,
               const void *datain, GLint widthout, GLint heightout, GLenum typeout, void *dataout )
```

PARAMETERS

- **format**: Specifies the format of the pixel data. The following symbolic values are valid: `GL_COLOR_INDEX`, `GL_STENCIL_INDEX`, `GL_DEPTH_COMPONENT`, `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`, `GL_RGB`, `GL_RGBA`, `GL_LUMINANCE`, and `GL_LUMINANCE_ALPHA`.

- **widthin**, **heightin**: Specify the width and height, respectively, of the source image that is scaled.

- **typein**: Specifies the data type for `datain`. Must be one of `GL_UNSIGNED_BYTE`, `GL_BYTE`, `GL_BITMAP`, `GL_UNSIGNED_SHORT`, `GL_SHORT`, `GL_UNSIGNED_INT`, `GL_INT`, or `GL_FLOAT`.

- **datain**: Specifies a pointer to the source image.

- **widthout**, **heightout**: Specify the width and height, respectively, of the destination image.

- **typeout**: Specifies the data type for `dataout`. Must be one of `GL_UNSIGNED_BYTE`, `GL_BYTE`, `GL_BITMAP`, `GL_UNSIGNED_SHORT`, `GL_SHORT`, `GL_UNSIGNED_INT`, `GL_INT`, or `GL_FLOAT`.

- **dataout**: Specifies a pointer to the destination image.

DESCRIPTION

gluScaleImage scales a pixel image using the appropriate pixel store modes to unpack data from the source image and pack data into the destination image.

When shrinking an image, gluScaleImage uses a box filter to sample the source image and create pixels for the destination image. When magnifying an image, the pixels from the source image are linearly interpolated to create the destination image.

A return value of zero indicates success, otherwise a GLU error code is returned indicating what the problem was (see "gluErrorString").
Please refer to the `glReadPixels` reference page for a description of the acceptable values for the `format`, `typein`, and `typeout` parameters.

**SEE ALSO**

"glDrawPixels", "glReadPixels", "gluBuild1DMipmaps", "gluBuild2DMipmaps", "gluErrorString"

### gluSphere

**NAME**

gluSphere - draw a sphere

**C SPECIFICATION**

```c
void gluSphere( GLUquadricObj *qobj, GLdouble radius, GLint slices, GLint stacks )
```

**PARAMETERS**

- `qobj` Specifies the quadrics object (created with `gluNewQuadric`).
- `radius` Specifies the radius of the sphere.
- `slices` Specifies the number of subdivisions around the z axis (similar to lines of longitude).
- `stacks` Specifies the number of subdivisions along the z axis (similar to lines of latitude).

**DESCRIPTION**

`gluSphere` draws a sphere of the given radius centered around the origin. The sphere is subdivided around the z axis into slices and along the z axis into stacks (similar to lines of longitude and latitude).

If the orientation is set to `GLU_OUTSIDE` (with `gluQuadricOrientation`), then any normals generated point away from the center of the sphere. Otherwise, they point toward the center of the sphere.

If texturing is turned on (with `gluQuadricTexture`), then texture coordinates are generated so that \( t \) ranges from 0.0 at \( z = -radius \) to 1.0 at \( z = radius \) (\( t \) increases linearly along longitudinal lines), and \( s \) ranges from 0.0 at the +y axis, to 0.25 at the +x axis, to 0.5 at the -y axis, to 0.75 at the -x axis, and back to 1.0 at the +y axis.

**SEE ALSO**
**NAME**

gluTessCallback - define a callback for a tessellation object

**C SPECIFICATION**

```c
void gluTessCallback( GLUtriangulatorObj *tobj, GLenum which, void (*fn)( )
```

**PARAMETERS**

- `tobj` Specifies the tessellation object (created with gluNewTess).
- `which` Specifies the callback being defined. The following values are valid: GLU_BEGIN, GLU_EDGE_FLAG, GLU_VERTEX, GLU_END, and GLU_ERROR.
- `fn` Specifies the function to be called.

**DESCRIPTION**

gluTessCallback is used to indicate a callback to be used by a tessellation object. If the specified callback is already defined, then it is replaced. If `fn` is NULL, then the existing callback is erased.

These callbacks are used by the tessellation object to describe how a polygon specified by the user is broken into triangles.

The legal callbacks are as follows:

**GLU_BEGIN**

The begin callback is invoked like glBegin to indicate the start of a (triangle) primitive. The function takes a single argument of type GLenum that is either GL_TRIANGLE_FAN, GL_TRIANGLES, or GL_TRIANGLES.

**GLU_EDGE_FLAG**

The edge flag callback is similar to glEdgeFlag. The function takes a single Boolean flag that indicates which edges of the created triangles were part of the original polygon defined by the user, and which were created by the tessellation process. If the flag is GL_TRUE, then each vertex that follows begins an edge that was part of the original polygon. If the flag is GL_FALSE, then each vertex that follows begins an edge that was generated...
by the tessellator. The edge flag callback (if defined) is invoked before the first vertex callback is made.

Since triangle fans and triangle strips do not support edge flags, the begin callback is not called with \texttt{GL\_TRIANGLE\_FAN} or \texttt{GL\_TRIANGLE\_STRIP} if an edge flag callback is provided. Instead, the fans and strips are converted to independent triangles.

\textbf{GLU\_VERTEX}

The vertex callback is invoked between the begin and end callbacks. It is similar to \texttt{glVertex}, and it defines the vertices of the triangles created by the tessellation process. The function takes a pointer as its only argument. This pointer is identical to the opaque pointer provided by the user when the vertex was described (see "glTessVertex").

\textbf{GLU\_END}

The end callback serves the same purpose as \texttt{glEnd}. It indicates the end of a primitive and it takes no arguments.

\textbf{GLU\_ERROR}

The error callback is called when an error is encountered. The one argument is of type \texttt{GLenum}, and it indicates the specific error that occurred. There are eight errors unique to polygon tessellation, named \texttt{GLU\_TESS\_ERROR1} through \texttt{GLU\_TESS\_ERROR8}. Character strings describing these errors can be retrieved with the \texttt{gluErrorString} call.

\section*{EXAMPLE}

Polygons tessellated can be rendered directly like this:

\begin{verbatim}
gluTessCallback(tobj, GLU\_BEGIN, glBegin); gluTessCallback(tobj, GLU\_VERTEX, glVertex3dv); gluTessCallback(tobj, GLU\_END, glEnd); glBeginPolygon(tobj);   gluTessVertex(tobj, v, v); ... glEndPolygon(tobj);
\end{verbatim}

Typically, the tessellated polygon should be stored in a display list so that it does not need to be retessellated every time it is rendered.

\section*{SEE ALSO}

"glBegin", "glEdgeFlag", "glVertex", "gluDeleteTess", "gluErrorString", "gluNewTess", "gluTessVertex"

\texttt{gluTessVertex}
NAME

`gluTessVertex` - specify a vertex on a polygon

C SPECIFICATION

```c
void gluTessVertex( GLUtessCallbackObj *tobj, GLdouble v[3], void *data )
```

PARAMETERS

- `tobj` Specifies the tessellation object (created with `gluNewTess`).
- `v` Specifies the location of the vertex.
- `data` Specifies an opaque pointer passed back to the user with the vertex callback (as specified by `gluTessCallback`).

DESCRIPTION

`gluTessVertex` describes a vertex on a polygon that the user is defining. Successive `gluTessVertex` calls describe a closed contour. For example, if the user wants to describe a quadrilateral, then `gluTessVertex` should be called four times. `gluTessVertex` can only be called between `gluBeginPolygon` and `gluEndPolygon`.

`data` normally points to a structure containing the vertex location, as well as other per-vertex attributes such as color and normal. This pointer is passed back to the user through the `GLU_VERTEX` callback after tessellation (see the "`gluTessCallback`" reference page).

EXAMPLE

A quadrilateral with a triangular hole in it can be described as follows:

```c
 gluBeginPolygon(tobj);
    gluTessVertex(tobj, v1, v1);
    gluTessVertex(tobj, v2, v2);
    gluTessVertex(tobj, v3, v3);
    gluTessVertex(tobj, v4, v4);
    gluNextContour(tobj, GLU_INTERIOR);
    gluTessVertex(tobj, v5, v5);
    gluTessVertex(tobj, v6, v6);
    gluTessVertex(tobj, v7, v7);
 gluEndPolygon(tobj);
```

SEE ALSO
gluUnProject

NAME

gluUnProject - map window coordinates to object coordinates

C SPECIFICATION

int gluUnProject( GLdouble winx, GLdouble winy, GLdouble winz, const GLdouble modelMatrix[16], const GLdouble projMatrix[16], const GLint viewport[4], GLdouble *objx, GLdouble *objy, GLdouble *objz )

PARAMETERS

  winx, winy, winz  Specify the window coordinates to be mapped.
  modelMatrix  Specifies the modelview matrix (as from a glGetDoublev call).
  projMatrix  Specifies the projection matrix (as from a glGetDoublev call).
  viewport  Specifies the viewport (as from a glGetIntegerv call).
  objx, objy, objz  Returns the computed object coordinates.

DESCRIPTION

gluUnProject maps the specified window coordinates into object coordinates using modelMatrix, projMatrix, and viewport. The result is stored in objx, objy, and objz. A return value of GL_TRUE indicates success, and GL_FALSE indicates failure.

SEE ALSO

"glGet", "gluProject"

Chapter 7. GLX Reference Pages

OpenGL Reference Manual
gives an overview of OpenGL in the X Window System; you might want to start with this page.

**glXChooseVisual**

**NAME**

`glXChooseVisual` - return a visual that matches specified attributes

**C SPECIFICATION**

`XVisualInfo* glXChooseVisual( Display *dpy, int screen, int *attribList )`

**PARAMETERS**

- `dpy` Specifies the connection to the X server.
- `screen` Specifies the screen number.
- `attribList` Specifies a list of Boolean attributes and integer attribute/value pairs. The last attribute must be `None`.

**DESCRIPTION**

`glXChooseVisual` returns a pointer to an `XVisualInfo` structure describing the visual that best meets a minimum specification. The Boolean GLX attributes of the visual that is returned will match the specified values, and the integer GLX attributes will meet or exceed the specified minimum values. If all other attributes are equivalent, then TrueColor and PseudoColor visuals have priority over DirectColor and StaticColor visuals, respectively. If no conforming visual exists, `NULL` is returned. To free the data returned by this function, use `XFree`.

All Boolean GLX attributes default to `False` except `GLX_USE_GL`, which defaults to `True`. All integer GLX attributes default to zero. Default specifications are superseded by attributes included in `attribList`. Boolean attributes included in `attribList` are understood to be `True`. Integer attributes are followed immediately by the corresponding desired or minimum value. The list must be terminated with `None`.

The interpretations of the various GLX visual attributes are as follows:

- **GLX_USE_GL** Ignored. Only visuals that can be rendered with GLX are considered.
- **GLX_BUFFER_SIZE** Must be followed by a nonnegative integer that indicates the desired color index buffer size. The smallest index buffer of at least the specified size is
preferred. Ignored if GLX_RGBA is asserted.

**GLX_LEVEL**
Must be followed by an integer buffer-level specification. This specification is honored exactly. Buffer level zero corresponds to the default frame buffer of the display. Buffer level one is the first overlay frame buffer, level two the second overlay frame buffer, and so on. Negative buffer levels correspond to underlay frame buffers.

**GLX_RGBA**
If present, only TrueColor and DirectColor visuals are considered. Otherwise, only PseudoColor and StaticColor visuals are considered.

**GLX_DOUBLEBUFFER**
If present, only double-buffered visuals are considered. Otherwise, only single-buffered visuals are considered.

**GLX_STEREO**
If present, only stereo visuals are considered. Otherwise, only monoscopic visuals are considered.

**GLX_AUX_BUFFERS**
Must be followed by a nonnegative integer that indicates the desired number of auxiliary buffers. Visuals with the smallest number of auxiliary buffers that meets or exceeds the specified number are preferred.

**GLX_RED_SIZE**
Must be followed by a nonnegative minimum size specification. If this value is zero, the smallest available red buffer is preferred. Otherwise, the largest available red buffer of at least the minimum size is preferred.

**GLX_GREEN_SIZE**
Must be followed by a nonnegative minimum size specification. If this value is zero, the smallest available green buffer is preferred. Otherwise, the largest available green buffer of at least the minimum size is preferred.

**GLX_BLUE_SIZE**
Must be followed by a nonnegative minimum size specification. If this value is zero, the smallest available blue buffer is preferred. Otherwise, the largest available blue buffer of at least the minimum size is preferred.

**GLX_ALPHA_SIZE**
Must be followed by a nonnegative minimum size
specification. If this value is zero, the smallest available alpha buffer is preferred. Otherwise, the largest available alpha buffer of at least the minimum size is preferred.

**GLX_DEPTH_SIZE**

Must be followed by a nonnegative minimum size specification. If this value is zero, visuals with no depth buffer are preferred. Otherwise, the largest available depth buffer of at least the minimum size is preferred.

**GLX_STENCIL_SIZE**

Must be followed by a nonnegative integer that indicates the desired number of stencil bitplanes. The smallest stencil buffer of at least the specified size is preferred. If the desired value is zero, visuals with no stencil buffer are preferred.

**GLX_ACCUM_RED_SIZE**

Must be followed by a nonnegative minimum size specification. If this value is zero, visuals with no red accumulation buffer are preferred. Otherwise, the largest possible red accumulation buffer of at least the minimum size is preferred.

**GLX_ACCUM_GREEN_SIZE**

Must be followed by a nonnegative minimum size specification. If this value is zero, visuals with no green accumulation buffer are preferred. Otherwise, the largest possible green accumulation buffer of at least the minimum size is preferred.

**GLX_ACCUM_BLUE_SIZE**

Must be followed by a nonnegative minimum size specification. If this value is zero, visuals with no blue accumulation buffer are preferred. Otherwise, the largest possible blue accumulation buffer of at least the minimum size is preferred.

**GLX_ACCUM_ALPHA_SIZE**

Must be followed by a nonnegative minimum size specification. If this value is zero, visuals with no alpha accumulation buffer are preferred. Otherwise, the largest possible alpha accumulation buffer of at least the minimum size is preferred.

**EXAMPLES**

```c
attribList {GLX_RGBA, GLX_RED_SIZE, 4, GLX_GREEN_SIZE, 4,
```
Specifies a single-buffered RGB visual in the normal frame buffer, not an overlay or underlay buffer. The returned visual supports at least four bits each of red, green, and blue, and possibly no bits of alpha. It does not support color index mode, double-buffering, or stereo display. It may or may not have one or more auxiliary color buffers, a depth buffer, a stencil buffer, or an accumulation buffer.

NOTES

XVisualInfo is defined in Xutil.h. It is a structure that includes visual, visualID, screen, and depth elements.

glxXChooseVisual is implemented as a client-side utility using only XGetVisualInfo and glXGetConfig. Calls to these two routines can be used to implement selection algorithms other than the generic one implemented by glXChooseVisual.

GLX implementers are strongly discouraged, but not proscribed, from changing the selection algorithm used by glXChooseVisual. Therefore, selections may change from release to release of the client-side library.

There is no direct filter for picking only visuals that support GLXPixmaps. GLXPixmaps are supported for visuals whose GLX_BUFFER_SIZE is one of the Pixmap depths supported by the X server.

ERRORS

NULL is returned if an undefined GLX attribute is encountered in attribList.

SEE ALSO

"glXCreateContext", "glXGetConfig"

glxXCopyContext

NAME

glxXCopyContext - copy state from one rendering context to another

C SPECIFICATION

void glXCopyContext( Display *dpy, GLXContext src, GLXContext dst, GLuint mask )

PARAMETERS
Specifies the connection to the X server.

Specifies the source context.

Specifies the destination context.

Specifies which portions of src state are to be copied to dst.

**DESCRIPTION**

`glXCopyContext` copies selected groups of state variables from src to dst. mask indicates which groups of state variables are to be copied. mask contains the bitwise OR of the same symbolic names that are passed to the OpenGL command `glPushAttrib`. The single symbolic constant `GL_ALL_ATTRIB_BITS` can be used to copy the maximum possible portion of rendering state.

The copy can be done only if the renderers named by src and dst share an address space. Two rendering contexts share an address space if both are nondirect using the same server, or if both are direct and owned by a single process. Note that in the nondirect case it is not necessary for the calling threads to share an address space, only for their related rendering contexts to share an address space.

Not all values for OpenGL state can be copied. For example, pixel pack and unpack state, render mode state, and select and feedback state are not copied. The state that can be copied is exactly the state that is manipulated by OpenGL command `glPushAttrib`.

An implicit `glFlush` is done by `glXCopyContext` if src is the current context for the calling thread.

If src is not the current context for the thread issuing the request, then the state of the src context is undefined.

**NOTES**

Two rendering contexts share an address space if both are nondirect using the same server, or if both are direct and owned by a single process.

A process is a single execution environment, implemented in a single address space, consisting of one or more threads.

A thread is one of a set of subprocesses that share a single address space, but maintain separate program counters, stack spaces, and other related global data. A thread that is the only member of its subprocess group is equivalent to a process.

**ERRORS**

BadMatch is generated if rendering contexts src and dst do not share an address space or were not created with respect to the same screen.
BadAccess is generated if $dst$ is current to any thread (including the calling thread) at the time `glXCopyContext` is called.

GLXBadCurrentWindow is generated if $src$ is the current context and the current drawable is a window that is no longer valid.

GLX_Bad_Context is generated if either $src$ or $dst$ is not a valid GLX context.

BadValue is generated if undefined $mask$ bits are specified.

SEE ALSO

"glPushAttrib", "glXCreateContext", "glXIsDirect"

**glXCreateContext**

**NAME**

`glXCreateContext` - create a new GLX rendering context

**C SPECIFICATION**

```c
GLXContext glXCreateContext( Display *dpy, XVisualInfo *vis, GLXContext shareList, Bool direct )
```

**PARAMETERS**

- `$dpy$` Specifies the connection to the X server.
- `$vis$` Specifies the visual that defines the frame buffer resources available to the rendering context. It is a pointer to an `XVisualInfo` structure, not a visual ID or a pointer to a `Visual`.
- `$shareList$` Specifies the context with which to share display lists. NULL indicates that no sharing is to take place.
- `$direct$` Specifies whether rendering is to be done with a direct connection to the graphics system if possible (`True`) or through the X server (`False`).

**DESCRIPTION**

`glXCreateContext` creates a GLX rendering context and returns its handle. This context can be used to render into both windows and GLX pixmaps. If `glXCreateContext` fails to create a rendering context, NULL is returned.

If `$direct$` is `True`, then a direct rendering context is 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 always created. Direct rendering provides a performance advantage in some implementations. However, direct rendering contexts cannot be shared outside a single process, and they cannot be used to render to GLX pixmaps.

If `shareList` is not NULL, then all display-list indexes and definitions are shared by context `shareList` and by the newly created context. An arbitrary number of contexts can share a single display-list space. However, all rendering contexts that share a single display-list space must themselves exist in the same address space. Two rendering contexts share an address space if both are nondirect using the same server, or if both are direct and owned by a single process. Note that in the nondirect case, it is not necessary for the calling threads to share an address space, only for their related rendering contexts to share an address space.

**NOTES**

`XVisualInfo` is defined in `Xutil.h`. It is a structure that includes `visual`, `visualID`, `screen`, and `depth` elements.

A *process* is a single execution environment, implemented in a single address space, consisting of one or more threads.

A *thread* is one of a set of subprocesses that share a single address space, but maintain separate program counters, stack spaces, and other related global data. A *thread* that is the only member of its subprocess group is equivalent to a *process*.

**ERRORS**

`NULL` is returned if execution fails on the client side.

`BadMatch` is generated if the context to be created would not share the address space or the screen of the context specified by `shareList`.

`BadValue` is generated if `vis` is not a valid visual (e.g., if the GLX implementation does not support it).

`GLX_Bad_Context` is generated if `shareList` is not a GLX context and is not `NULL`.

`BadAlloc` is generated if the server does not have enough resources to allocate the new context.

**SEE ALSO**

"glXDestroyContext" . "glXGetConfig" . "glXIsDirect" . "glXMakeCurrent"
glXCreateGLXPixmap

NAME

glXCreateGLXPixmap - create an off-screen GLX rendering area

C SPECIFICATION

GLXPixmap glXCreateGLXPixmap( Display *dpy, XVisualInfo *vis, Pixmap pixmap )

PARAMETERS

dpy  Specifies the connection to the X server.
vis  Specifies the visual that defines the structure of the rendering area. It is a pointer to an XVisualInfo structure, not a visual ID or a pointer to a Visual.
pixmap  Specifies the X pixmap that will be used as the front left color buffer of the off-screen rendering area.

DESCRIPTION

glXCreateGLXPixmap creates an off-screen rendering area and returns its XID. Any GLX rendering context that was created with respect to vis can be used to render into this off-screen area. Use glXMakeCurrent to associate the rendering area with a GLX rendering context.

The X pixmap identified by pixmap is used as the front left buffer of the resulting off-screen rendering area. All other buffers specified by vis, including color buffers other than the front left buffer, are created without externally visible names. GLX pixmaps with double-buffering are supported. However, glXSwapBuffers is ignored by these pixmaps.

Direct rendering contexts cannot be used to render into GLX pixmaps.

NOTES

XVisualInfo is defined in Xutil.h. It is a structure that includes visual, visualID, screen, and depth elements.

ERRORS

BadMatch is generated if the depth of pixmap does not match the GLX_BUFFER_SIZE value of vis, or if pixmap was not created with respect to the same screen as vis.
**BadValue** is generated if *vis* is not a valid XVisualInfo pointer (e.g., if the GLX implementation does not support this visual).

**BadPixmap** is generated if *pixmap* is not a valid pixmap.

**BadAlloc** is generated if the server cannot allocate the GLX pixmap.

SEE ALSO

"glXCreateContext", "glXIsDirect", "glXMakeCurrent"

**glXDestroyContext**

NAME

**glXDestroyContext** - destroy a GLX context

C SPECIFICATION

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

PARAMETERS

*dpy*   Specifies the connection to the X server.
*ctx*   Specifies the GLX context to be destroyed.

DESCRIPTION

If GLX rendering context *ctx* is not current to any thread, **glXDestroyContext** destroys it immediately. Otherwise, *ctx* is destroyed when it becomes not current to any thread. In either case, the resource ID referenced by *ctx* is freed immediately.

ERRORS

**GLX_Bad_Context** is generated if *ctx* is not a valid GLX context.

SEE ALSO

"glXCreateContext", "glXMakeCurrent"

**glXDestroyGLXPixmap**

NAME
glXDestroyGLXPixmap - destroy a GLX pixmap

C SPECIFICATION

void glXDestroyGLXPixmap( Display *dpy, GLXPixmap pix )

PARAMETERS

dpy  Specifies the connection to the X server.
pix  Specifies the GLX pixmap to be destroyed.

DESCRIPTION

If GLX pixmap pix is not current to any client, glXDestroyGLXPixmap destroys it immediately. Otherwise, pix is destroyed when it becomes not current to any client. In either case, the resource ID is freed immediately.

ERRORS

GLX_Bad_Pixmap is generated if pix is not a valid GLX pixmap.

SEE ALSO

"glXCreateGLXPixmap", "glXMakeCurrent"

glXGetConfig

NAME

glXGetConfig - return information about GLX visuals

C SPECIFICATION

int glXGetConfig( Display *dpy, XVisualInfo *vis, int attrib, int *value )

PARAMETERS

dpy  Specifies the connection to the X server.
vis  Specifies the visual to be queried. It is a pointer to an XVisualInfo structure, not a visual ID or a pointer to a Visual.
attrib  Specifies the visual attribute to be returned.
value  Returns the requested value.
DESCRIPTION

`glXGetConfig` sets value to the attrib value of windows or GLX pixmaps created with respect to vis. `glXGetConfig` returns an error code if it fails for any reason. Otherwise, zero is returned.

attrib is one of the following:

**GLX_USE_GL**
- **True** if OpenGL rendering is supported by this visual, **False** otherwise.

**GLX_BUFFER_SIZE**
Number of bits per color buffer. For RGBA visuals, `GLX_BUFFER_SIZE` is the sum of `GLX_RED_SIZE`, `GLX_GREEN_SIZE`, `GLX_BLUE_SIZE`, and `GLX_ALPHA_SIZE`. For color index visuals, `GLX_BUFFER_SIZE` is the size of the color indexes.

**GLX_LEVEL**
Frame buffer level of the visual. Level zero is the default frame buffer. Positive levels correspond to frame buffers that overlay the default buffer, and negative levels correspond to frame buffers that underlay the default buffer.

**GLX_RGBA**
- **True** if color buffers store red, green, blue, and alpha values, **False** if they store color indexes.

**GLX_DOUBLEBUFFER**
- **True** if color buffers exist in front/back pairs that can be swapped, **False** otherwise.

**GLX_STEREO**
- **True** if color buffers exist in left/right pairs, **False** otherwise.

**GLX_AUX_BUFFERS**
Number of auxiliary color buffers that are available. Zero indicates that no auxiliary color buffers exist.

**GLX_RED_SIZE**
Number of bits of red stored in each color buffer. Undefined if `GLX_RGBA` is **False**.

**GLX_GREEN_SIZE**
Number of bits of green stored in each color buffer. Undefined if `GLX_RGBA` is **False**.

**GLX_BLUE_SIZE**
Number of bits of blue stored in each color buffer. Undefined if `GLX_RGBA` is **False**.

**GLX_ALPHA_SIZE**
Number of bits of alpha stored in each color buffer.
GLX_DEPTH_SIZE  Undefined if GLX_RGB is False.

GLX_STENCIL_SIZE  Number of bits in the stencil buffer.

GLX_ACCUM_RED_SIZE  Number of bits of red stored in the accumulation buffer.

GLX_ACCUM_GREEN_SIZE  Number of bits of green stored in the accumulation buffer.

GLX_ACCUM_BLUE_SIZE  Number of bits of blue stored in the accumulation buffer.

GLX_ACCUM_ALPHA_SIZE  Number of bits of alpha stored in the accumulation buffer.

The X protocol allows a single visual ID to be instantiated with different numbers of bits per pixel. Windows or GLX pixmaps that will be rendered with OpenGL, however, must be instantiated with a color buffer depth of GLX_BUFFER_SIZE.

Although a GLX implementation can export many visuals that support OpenGL rendering, it must support at least two. One is an RGBA visual with 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 in this visual. However, its color buffer size must be as great as that of the deepest TrueColor, DirectColor, PseudoColor, or StaticColor visual supported on level zero, and it must itself be made available on level zero.

The other required visual is a color index one with at least one color buffer, a stencil buffer of at least 1 bit, and a depth buffer of at least 12 bits. This visual must have as many color bitplanes as the deepest PseudoColor or StaticColor visual supported on level zero, and it must itself be made available on level zero.

Applications are best written to select the visual that most closely meets their requirements. Creating windows or GLX pixmaps with unnecessary buffers can result in reduced rendering performance as well as poor resource allocation.

NOTES

XVisualInfo is defined in Xutil.h. It is a structure that includes visual, visualID, screen, and depth elements.
ERRORS

GLX_NO_EXTENSION is returned if dpy does not support the GLX extension.
GLX_BAD_SCREEN is returned if the screen of vis does not correspond to a screen.
GLX_BAD_ATTRIB is returned if attrib is not a valid GLX attribute.
GLX_BAD_VISUAL is returned if vis doesn't support GLX and an attribute other than
GLX_USE_GL is requested.

SEE ALSO

"glXChooseVisual", "glXCreateContext"

glXGetCurrentContext

NAME

glXGetCurrentContext - return the current context

C SPECIFICATION

GLXContext glXGetCurrentContext( void )

DESCRIPTION

glXGetCurrentContext returns the current context, as specified by glXMakeCurrent.
If there is no current context, NULL is returned. glXGetCurrentContext returns client-
side information. It does not make a round trip to the server.

SEE ALSO

"glXCreateContext", "glXMakeCurrent"

glXGetCurrentDrawable

NAME

glXGetCurrentDrawable - return the current drawable

C SPECIFICATION

GLXDrawable glXGetCurrentDrawable( void )

DESCRIPTION
glXGetCurrentDrawable returns the current drawable, as specified by glXMakeCurrent. If there is no current drawable, None is returned.
glXGetCurrentDrawable returns client-side information. It does not make a round trip to the server.

SEE ALSO
"glXCreateGLXPixmap", "glXMakeCurrent"

glXIntro

NAME

glXIntro - Introduction to OpenGL in the X window system

OVERVIEW

OpenGL is a high-performance 3-D-oriented renderer. It is available in the X window system through the GLX extension. Use glXQueryExtension and glXQueryVersion to establish whether the GLX extension is supported by an X server, and if so, what version is supported. GLX extended servers make a subset of their visuals available for OpenGL rendering. Drawable created with these visuals can also be rendered using the core X renderer and with the renderer of any other X extension that is compatible with all core X visuals. GLX extends drawables with several buffers other than the standard color buffer. These buffers include back and auxiliary color buffers, a depth buffer, a stencil buffer, and a color accumulation buffer. Some or all are included in each X visual that supports OpenGL. To render using OpenGL into an X drawable, you must first choose a visual that defines the required OpenGL buffers. glXChooseVisual can be used to simplify selecting a compatible visual. If more control of the selection process is required, use XGetVisualInfo and glXGetConfig to select among all the available visuals. Use the selected visual to create both a GLX context and an X drawable. GLX contexts are created with glXCreateContext, and drawables are created with either XCreateWindow or glXCreateGLXPixmap. Finally, bind the context and the drawable together using glXMakeCurrent. This context/drawable pair becomes the current context and current drawable, and it is used by all OpenGL commands until glXMakeCurrent is called with different arguments. Both core X and OpenGL commands can be used to operate on the current drawable. The X and OpenGL command streams are not synchronized, however, except at explicitly created boundaries generated by calling glXWaitGL, glXWaitX, XSync, and glFlush.

EXAMPLES

Below is the minimum code required to create an RGBA-format, OpenGL-compatible X window and clear it to yellow. The code is correct, but it does not include any error checking. Return values dpy, vi, cx, cmap, and win should all be tested.
```c
#include <GL/glx.h>
#include <GL/gl.h>
#include <unistd.h>
static int attributeList[] = { GLX_RGBA, None };
static Bool WaitForNotify(Display *d, XEvent *e, char *arg) {
    return (e->type == MapNotify) && (e->xmap.window == (Window)arg);
}
int main(int argc, char **argv) {
    Display *dpy;
    XVisualInfo *vi;
    Colormap cmap;
    XSetWindowAttributes swa;
    Window win;
    GLXContext cx;
    XEvent event;
    /* get a connection */
    dpy = XOpenDisplay(0);
    /* get an appropriate visual */
    vi = glXChooseVisual(dpy, DefaultScreen(dpy), attributeList);
    /* create a GLX context */
    cx = glXCreateContext(dpy, vi, 0, GL_FALSE);
    /* create a color map */
    cmap = XCreateColormap(dpy, RootWindow(dpy, vi->screen),
        vi->visual, AllocNone);
    /* create a window */
    swa.colormap = cmap;
    swa.border_pixel = 0;
    swa.event_mask = StructureNotifyMask;
    win = XCreateWindow(dpy, RootWindow(dpy, vi->screen), 0, 0, 100, 100,
        0, vi->depth, InputOutput, vi->visual,
        CWBorderPixel|CWColormap|CWEventMask, &swa);
    XMapWindow(dpy, win);
    XIfEvent(dpy, &event, WaitForNotify, (char*)win);
    /* connect the context to the window */
    glXMakeCurrent(dpy, win, cx);
    /* clear the buffer */
    glClearColor(1,1,0,1);
    glClear(GL_COLOR_BUFFER_BIT);
    glFlush();
    /* wait a while */
    sleep(10);
}
```

**NOTES**

A color map must be created and passed to `XCreateWindow`. See the example code above.
A GLX context must be created and attached to an X drawable before OpenGL commands can be executed. OpenGL commands issued while no context/drawable pair is current are ignored.

Exposure events indicate that all buffers associated with the specified window may be damaged and should be repainted. Although certain buffers of some visuals on some systems may never require repainting (the depth buffer, for example), it is incorrect to code assuming that these buffers will not be damaged.

GLX commands manipulate XVisualInfo structures rather than pointers to visuals or visual IDs. XVisualInfo structures contain visual, visualID, screen, and depth elements, as well as other X-specific information.

SEE ALSO


glXIsDirect

NAME

glxIsDirect - indicate whether direct rendering is enabled

C SPECIFICATION

Bool glXIsDirect( Display *dpy, GLXContext ctx )

PARAMETERS

dpy Specifies the connection to the X server.
ctx Specifies the GLX context that is being queried.

DESCRIPTION

glxIsDirect returns True if ctx is a direct rendering context, False otherwise. Direct rendering contexts pass rendering commands directly from the calling process's address space to the rendering system, bypassing the X server. Nondirect rendering contexts pass all rendering commands to the X server.

ERRORS
GLX_Bad_Context is generated if ctx is not a valid GLX context.

SEE ALSO
"glXCreateContext"

**glXMakeCurrent**

**NAME**

**glXMakeCurrent** - attach a GLX context to a window or a GLX pixmap

**C SPECIFICATION**

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

**PARAMETERS**

- **dpy** Specifies the connection to the X server.
- **drawable** Specifies a GLX drawable. Must be either an X window ID or a GLX pixmap ID.
- **ctx** Specifies a GLX rendering context that is to be attached to **drawable**.

**DESCRIPTION**

**glXMakeCurrent** does two things: It makes ctx the current GLX rendering context of the calling thread, replacing the previously current context if there was one, and it attaches ctx to a GLX drawable, either a window or a GLX pixmap. As a result of these two actions, subsequent OpenGL rendering calls use rendering context ctx to modify GLX drawable drawable. Because **glXMakeCurrent** always replaces the current rendering context with ctx, there can be only one current context per thread.

Pending commands to the previous context, if any, are flushed before it is released.

The first time ctx is made current to any thread, its viewport is set to the full size of drawable. Subsequent calls by any thread to **glXMakeCurrent** with ctx have no effect on its viewport.

To release the current context without assigning a new one, call **glXMakeCurrent** with drawable and ctx set to None and NULL respectively.

**glXMakeCurrent** returns True if it is successful, False otherwise. If False is returned, the previously current rendering context and drawable (if any) remain unchanged.
NOTES

A process is a single-execution environment, implemented in a single address space, consisting of one or more threads.

A thread is one of a set of subprocesses that share a single address space, but maintain separate program counters, stack spaces, and other related global data. A thread that is the only member of its subprocess group is equivalent to a process.

ERRORS

BadMatch is generated if drawable was not created with the same X screen and visual as ctx. It is also generated if drawable is None and ctx is not None.

BadAccess is generated if ctx was current to another thread at the time glXMakeCurrent was called.

GLX_Bad_Drawable is generated if drawable is not a valid GLX drawable.

GLX_Bad_Context is generated if ctx is not a valid GLX context.

GLX_Bad_Context_State is generated if the rendering context current to the calling thread has OpenGL renderer state GL_FEEDBACK or GL_SELECT.

GLX_Bad_Current_Window is generated if there are pending OpenGL commands for the previous context and the current drawable is a window that is no longer valid.

BadAlloc may be generated if the server has delayed allocation of ancillary buffers until glXMakeCurrent is called, only to find that it has insufficient resources to complete the allocation.

SEE ALSO

"glXCreateContext", "glXCreateGLXPixmap"

glXQueryExtension

NAME

glXQueryExtension - indicate whether the GLX extension is supported

C SPECIFICATION

Bool glXQueryExtension( Display *dpy, int *errorBase, int *eventBase )
PARAMETERS

dpy     Specifies the connection to the X server.
errorBase Returns the base error code of the GLX server extension.
eventBase Returns the base event code of the GLX server extension.

DESCRIPTION

glxQueryExtension returns True if the X server of connection dpy supports the GLX extension, False otherwise. If True is returned, then errorBase and eventBase return the error base and event base of the GLX extension. Otherwise, errorBase and eventBase are unchanged.

errorBase and eventBase do not return values if they are specified as NULL.

NOTES

eventBase is included for future extensions. GLX does not currently define any events.

SEE ALSO

"glXQueryVersion"

glxQueryVersion

NAME

glxQueryVersion - return the version numbers of the GLX extension

C SPECIFICATION

Bool glxQueryVersion( Display *dpy, int *major, int *minor )

PARAMETERS

dpy     Specifies the connection to the X server.
major   Returns the major version number of the GLX server extension.
minor   Returns the minor version number of the GLX server extension.

DESCRIPTION

glxQueryVersion returns the major and minor version numbers of the GLX extension implemented by the server associated with connection dpy. Implementations with the
same major version number are upward compatible, meaning that the implementation
with the higher minor number is a superset of the version with the lower minor number.

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

**ERRORS**

`glXQueryVersion` returns *False* if it fails, *True* otherwise. *major* and *minor* are not
updated when *False* is returned.

**SEE ALSO**

"glXQueryExtension"

**glXSwapBuffers**

**NAME**

`glXSwapBuffers` - make back buffer visible

**C SPECIFICATION**

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

**PARAMETERS**

- *dpy* Specifies the connection to the X server.
- *drawable* Specifies the window whose buffers are to be swapped.

**DESCRIPTION**

`glXSwapBuffers` promotes the contents of the back buffer of *drawable* to become the
contents of the front buffer of *drawable*. The contents of the back buffer then become
undefined. The update typically takes place during the vertical retrace of the monitor,
rather than immediately after `glXSwapBuffers` is called. All GLX rendering contexts
share the same notion of which are front buffers and which are back buffers.

An implicit `glFlush` is done by `glXSwapBuffers` before it returns. Subsequent OpenGL
commands can be issued immediately after calling `glXSwapBuffers`, but are not
executed until the buffer exchange is completed.

If *drawable* was not created with respect to a double-buffered visual, `glXSwapBuffers`
has no effect, and no error is generated.
NOTES

Synchronization of multiple GLX contexts rendering to the same double-buffered window is the responsibility of the clients. The X Synchronization Extension can be used to facilitate such cooperation.

ERRORS

GLX_Bad_Drawable is generated if drawable is not a valid GLX drawable.

GLX_Bad_Current_Window is generated if dpy and drawable are respectively the display and drawable associated with the current context of the calling thread, and drawable identifies a window that is no longer valid.

SEE ALSO

"glFlush"

glxUseXFont

NAME

glxUseXFont - create bitmap display lists from an X font

C SPECIFICATION

void glxUseXFont( Font font, int first, int count, int listBase )

PARAMETERS

font Specifies the font from which character glyphs are to be taken.
first Specifies the index of the first glyph to be taken.
count Specifies the number of glyphs to be taken.
listBase Specifies the index of the first display list to be generated.

DESCRIPTION

glxUseXFont generates count display lists, named listBase through listBase + count - 1, each containing a single glBitmap command. The parameters of the glBitmap command of display list listBase + i are derived from glyph first + i. Bitmap parameters xorig, yorig, width, and height are computed from font metrics as descent-1, -lbearing, rbearing-lbearing, and ascent+descent, respectively. xmove is taken from the glyph's width metric, and ymove is set to zero. Finally, the glyph's image is converted to the appropriate format for glBitmap.
Using glXUseXFont may be more efficient than accessing the X font and generating the display lists explicitly, both because the display lists are created on the server without requiring a round trip of the glyph data, and because the server may choose to delay the creation of each bitmap until it is accessed.

Empty display lists are created for all glyphs that are requested and are not defined in font. glXUseXFont is ignored if there is no current GLX context.

ERRORS

BadFont is generated if font is not a valid font.

GLX_Bad_Context_State is generated if the current GLX context is in display-list construction mode.

GLX_Bad_Current_Window is generated if the drawable associated with the current context of the calling thread is a window, and that window is no longer valid.

SEE ALSO

"glBitmap", "glXMakeCurrent"

glXWaitGL

NAME

glXWaitGL - complete GL execution prior to subsequent X calls

C SPECIFICATION

void glXWaitGL( void )

DESCRIPTION

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

glXWaitGL is ignored if there is no current GLX context.

NOTES

glXWaitGL may or may not flush the X stream.
ERRORS

GLX_Bad_Current_Window is generated if the drawable associated with the current context of the calling thread is a window, and that window is no longer valid.

SEE ALSO

"glFinish", "glFlush", "glXWaitX", XSync

glXWaitX

NAME

glXWaitX - complete X execution prior to subsequent OpenGL calls

C SPECIFICATION

void glXWaitX( void )

DESCRIPTION

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

glXWaitX is ignored if there is no current GLX context.

NOTES

glXWaitX may or may not flush the OpenGL stream.

ERRORS

GLX_Bad_Current_Window is generated if the drawable associated with the current context of the calling thread is a window, and that window is no longer valid.

SEE ALSO

"glFinish", "glFlush", "glXWaitGL", XSync