Legal Information

OpenGL Programmer's Reference

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.

The following topics present a global view of how OpenGL works:

- Primitives and Commands discusses points, line segments, and polygons as the basic units of drawing; and the processing of commands.
- OpenGL Graphic Control describes which graphic operations OpenGL controls and which it does not control.
- Execution Model discusses the client/server model for interpreting OpenGL commands.
- Basic OpenGL Operation gives a high-level description of how OpenGL processes data and produces a corresponding image in the frame buffer.

Primitives and Commands

OpenGL draws *primitives*—points, line segments, or polygons—subject to several selectable modes. You can control modes independently of one another. 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). To specify primitives, set modes, and perform other OpenGL operations, you issue 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 exceptions to this rule are cases in which 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 is consistent with complete execution of all previously issued OpenGL commands.

OpenGL Graphic Control

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. To fully understand how to use OpenGL, it helps to know the order in which it carries out its operations.

Execution Model

The model for interpretation of OpenGL commands is client/server. Application code (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 OpenGL contexts, each of which is an encapsulated OpenGL state. A client can connect to any one of these contexts. The required network protocol can be implemented by augmenting an already existing protocol or by using an independent protocol. No OpenGL commands are provided for obtaining user input.

The window system that allocates frame buffer resources ultimately controls the effects of OpenGL commands on the frame buffer. The window system:

- Determines which portions of the frame buffer OpenGL may access at any given time.
- 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.

Basic OpenGL Operation

The following abstract, high-level block diagram illustrates how OpenGL processes data. In the diagram, commands enter from the left and proceed through what can be considered a processing pipeline. Some commands specify geometric objects to be drawn, and others control how the objects are handled during the various processing stages.

{ewc msdncd, EWGraphic, bsd23540 0 /a "SDK.WMF"}

The processing stages in basic OpenGL operation are as follows:

- **Display list**. Rather than having all commands proceed immediately through the pipeline, you can choose to accumulate some of them in a display list for processing later.
- **Evaluator**. The evaluator stage of processing provides an efficient way to approximate curve and surface geometry by evaluating polynomial commands of input values.
- **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 rasterization.
- **Rasterization**. The rasterization stage 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.
- **Per-fragment operations**. These are the final operations performed on the data before it's stored as pixels in the frame buffer.

  Per-fragment 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.

Data can be input in the form of pixels rather than vertices. Data in the form of pixels, such as 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. Following pixel operations, the pixel data is either:

- Stored as texture memory, for use in the rasterization stage.
- Rasterized, with the resulting fragments merged into the frame buffer just as if they were generated from geometric data.

OpenGL Processing Pipeline

Many OpenGL functions are used specifically for drawing objects such as points, lines, polygons, and bitmaps. Other functions control the way that some of this drawing occurs (such as those that enable antialiasing or texturing). Still other functions are specifically concerned with frame buffer manipulation. The topics in this section describe how all the OpenGL functions work together to create the OpenGL processing pipeline. This section also takes a closer look at the stages in which data is actually processed, and ties these stages to OpenGL functions.

The following illustration shows a 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 into fragments, and finally into pixels in the frame buffer. This progression is discussed in more detail in Vertices, Primitives, Fragments, and Pixels.

{ewc msdncd, EWGraphic, bsd23541 0 /a "SDK.WMF"}

OpenGL Function Names

Many OpenGL functions are variations of each other, differing mostly in the data types of their arguments. Some functions differ in the number of related arguments and whether those arguments can be specified as a vector or must be specified separately in a list. For example, if you use the **glVertex2f** function, 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*. Only the base name of the function is used in the topics that follow. An asterisk indicates that there may be more to the actual function name than is shown. For example, **glVertex*** stands for all the variations of the function you use to specify vertices: **glVertex2d**, **glVertex2f**, **glVertex2i**, and so on.

The effect of an OpenGL function can vary depending on whether certain modes are enabled. For example, you need to enable lighting if the lighting-related functions are to produce a properly lit object. To enable a particular mode, use the **glEnable** function and supply the appropriate constant to identify the mode (for example, GL_LIGHTING). See **glEnable** for a complete list of the modes that can be enabled. Modes are disabled with **glDisable**.

Vertices

The topics in this section discuss the OpenGL functions that perform per-vertex operations to the processing stages shown in OpenGL Processing Pipeline.

Input Data

The OpenGL pipeline requires you to input several types of data:

- **Vertices**. Vertices describe the shape of the desired geometric object. To specify vertices, use **glVertex*** functions in conjunction with **glBegin** and **glEnd** to create a point, line, or polygon. You can also use **glRect** to describe an entire rectangle at one time.
- **Edge flag**. By default, all edges of polygons are boundary edges. Use **glEdgeFlag*** 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 to associate with a vertex of an object.

Note 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. Use functions such as **glMatrixMode**, **glMultMatrix***, **glRotate***, **glTranslate***, and **glScale*** to compose the desired transformations. Or specify matrices directly with **glLoadMatrix*** and **glLoadIdentity**. Use **glPushMatrix** and **glPopMatrix** to save and restore modelview and projection matrices on their respective stacks.

Setting Lighting and Coloring

In addition to specifying colors and normal vectors, you can define the desired lighting conditions with **glLight*** and **glLightModel***, and the desired material properties with **glMaterial***. Related functions for controlling 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 using **glTexGen***. After the texture coordinates have been specified or generated, they are transformed by the texture matrix. This matrix is controlled with the same functions that are used for matrix transformations (see Matrix Transformations).

Assembling Primitives

Once all necessary 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.

Vertices Reference

glBegin
glColor*
glColorMaterial
glEdgeFlag*
glEnd
glFrontFace
glIndex
glLight*
glLightModel*
glLoadIdentity
glLoadMatrix*
glMaterial*
glMatrixMode
glMultMatrix*
glNormal*
glPopMatrix
glPushMatrix
glRasterPos*
glRect*
glRotate*
glScale*
glShadeModel
glTexCoord*
glTexGen*
glTranslate*
glVertex*

Primitives

Primitives are converted to pixel fragments in the following steps:

- Primitives are clipped appropriately.
- Necessary corresponding adjustments are made to the color and texture data, and the relevant coordinates are transformed to window coordinates.
- Rasterization converts the clipped primitives to pixel fragments.

Clipping

Clipping occurs in two steps:

1. **Application-specific clipping**. Immediately after primitives are assembled, they're clipped in eye coordinates as necessary for any clipping planes you've defined with **glClipPlane**. (OpenGL requires support for at least six such application-specific clipping planes.)
2. **View volume clipping**. Primitives are transformed by the projection matrix into clip coordinates and clipped by the corresponding view volume. This matrix can be controlled by the matrix transformation functions (see Matrix Transformations) but is typically specified by **glFrustum** or **glOrtho**.

Points, line segments, and polygons are handled differently during clipping:

- Points are either retained in their original state (if they're inside the clip volume) or discarded (if they're outside the clip volume).
- 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 new vertices generated at the clip points.
- For line segments and polygons that are clipped, the edge flag, color, and texture information is assigned to all new vertices.

Transforming to Window Coordinates

Before clip coordinates are converted to window coordinates, they are divided by the value of *w* to yield normalized device coordinates. 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 **glDepthRange** and **glViewport**.

Rasterizing

Rasterizing 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. A point and its associated information are called a *fragment.* The current raster position, as specified with **glRasterPos***, is used in various ways during this stage for drawing pixels and bitmaps. Different issues arise when rasterizing points, line segments, and polygons. In addition, pixel rectangles and bitmaps need to be rasterized.

With OpenGL you control rasterizing using the following functions:

- **Primitives**. Control how primitives are rasterized using functions that determine dimensions and stipple patterns: **glPointSize**, **glLineWidth**, **glLineStipple**, and **glPolygonStipple**. Control how the front and back faces of polygons are rasterized with **glCullFace**, **glFrontFace**, and **glPolygonMode**.
- **Pixels**. Several functions control pixel storage and transfer modes. The function **glPixelStore*** controls the encoding of pixels in client memory, and **glPixelTransfer*** and **glPixelMap*** control how pixels are processed before being placed in the frame buffer. Specify a pixel rectangle with **glDrawPixels**; control its rasterization with **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. The **glBitmap** function specifies a bitmap.
- **Texture Memory**. When texturing is enabled, texturing maps a portion of a specified texture image onto each primitive. 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. Specify a texture image with **glTexImage2D** or **glTexImage1D**. The **glTexParameter*** and **glTexEnv*** functions control how texture values are interpreted and applied to a fragment.
- **Fog**. To blend a fog color with a rasterized fragment's post-texturing color, use a blending factor that depends on the distance between the eyepoint and the fragment. Use **glFog*** to specify the fog color and blending factor.

Primitives Reference

glBitmap
glClipPlane
glCullFace
glDepthRange
glDrawPixels
glFog*
glFrontFace
glFrustum
glLineStipple
glLineWidth
glOrtho
glPixelMap*
glPixelStore*
glPixelTransfer*
glPixelZoom
glPointSize
glPolygonMode
glPolygonStipple
glRasterPos*
glTexEnv*
glTexImage1D
glTexImage2D
glTexParameter*
glViewport

Fragments

A fragment produced by rasterization modifies the corresponding pixel in the frame buffer only if it passes the following tests:

- Pixel ownership test
- Scissor test
- Alpha test
- Stencil test
- Depth-buffer test

If it passes, the fragment's data can replace the existing frame buffer values, or you can combine it with existing data in the frame buffer, depending on the state of certain modes. You can combine the fragment with data in the frame buffer by:

- Blending
- Dithering
- Logical operations

Pixel Ownership Test

The pixel ownership test determines whether the current OpenGL context owns the pixel in the frame buffer corresponding to a particular fragment. If so, the fragment proceeds to the next test. If not, the windowing system determines whether the fragment is discarded or whether any further fragment operations will be performed with that fragment. With this test, the windowing system controls OpenGL's behavior when, for example, an OpenGL window is obscured.

Scissor Test

The **glScissor** test specifies an arbitrary screen-aligned rectangle outside of which fragments will be discarded.

Alpha Test

The alpha test (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. You specify the comparison function and reference value 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 **glStencilFunc** function specifies the comparison function 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 function. If stenciling is enabled, the result of the depth comparison also affects the stencil buffer update value.

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 may also depend on the RGB values. You control blending with **glBlendFunc**, with which you 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

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 indexes, never on RGBA values.

Fragments Reference

Pixels

Fragments are converted to pixels in the frame buffer. The frame buffer is 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 functions to control these buffers, and directly read or copy pixels from them. (Note that the particular OpenGL context you're using may not provide all these buffers.)

Frame Buffer Operations

To select the buffer into which color values are written, use **glDrawBuffer**. You use four different functions to mask the writing of bits to each of the logical frame buffers after all per-fragment operations have been performed:

glIndexMask
glColorMask
glDepthMask
glStencilMask

The **glAccum** function controls the operation of the accumulation buffer. 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 function **glReadBuffer** controls which color buffer the pixels are read or copied from.

Pixels Reference

glAccum
glClear
glClearAccum
glClearColor
glClearDepth
glClearIndex
glClearStencil
glColorMask
glCopyPixels
glDepthMask
glDrawBuffer
glIndexMask
glReadBuffer
glReadPixels
glStencilMask

Using Evaluators

The OpenGL evaluator functions allow you to use a polynomial mapping to produce vertices, normals, texture coordinates, and colors. These calculated values are then passed on to the processing pipeline as if they had been directly specified. The evaluator functions are also the basis for the NURBS (Non-Uniform Rational B-Spline) functions, which allow you to define curves and surfaces, as described in OpenGL Utility library.

The first step in using evaluators is to define the appropriate one- or two-dimensional polynomial mapping using **glMap***. You can then specify and evaluate the domain values for this map in one of two ways:

- Define a series of evenly spaced domain values to be mapped using **glMapGrid** and then evaluate a rectangular subset of that grid with **glEvalMesh**. A single point of the grid can be evaluated using **glEvalPoint**.
- Explicitly specify a desired domain value as an argument, which evaluates the maps at that value.

Evaluators Reference

glEvalCoord
glEvalMesh
glEvalPoint
glMap*
glMapGrid

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; therefore, no frame buffer modification occurs. In selection mode, you can determine which primitives will be drawn into some region of a window; in feedback mode, information about primitives that will be rasterized is fed back to the application.

You select among these three modes with **glRenderMode**.

Selection

Selection returns the current contents of the name stack, which is an array of names with integer values. 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** Call **glSelectBuffer** before you put OpenGL into selection mode with **glRenderMode**. The entire contents of the name stack aren't guaranteed to be returned until you call **glRenderMode** to take OpenGL out of selection mode.

Manipulate the name stack with **glInitNames**, **glLoadName**, **glPushName**, and **glPopName**. You can also use **gluPickMatrix** for selection.

Feedback

In feedback mode, each primitive to be rasterized generates a block of values that is copied into the feedback array. Supply this array with **glFeedbackBuffer**, which you must call before putting OpenGL 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 you call **glRenderMode** to take OpenGL out of feedback mode. You can use **glPassThrough** to supply a marker that is returned in feedback mode as if it were a primitive.

Selection and Feedback Reference

Selection and Feedback

glRenderMode

Selection

glInitNames
glLoadName
glPopName
glPushName
glSelectBuffer
gluPickMatrix

Feedback

glFeedbackBuffer
glPassThrough

Using Display Lists

A display list is a group of OpenGL functions that has been stored for subsequent execution. The **glNewList** function begins the creation of a display list, and **glEndList** ends it. With few exceptions, OpenGL functions called between **glNewList** and **glEndList** are appended to the display list. (See **glNewList** for a list of the functions that you can't store and execute 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 manage the indexes used to identify display lists with **glGenLists**, **glListBase**, and **glIsList**. To delete a set of display lists, use **glDeleteLists**.

Display Lists Reference

glCallList
glCallLists
glDeleteLists
glEndList
glGenLists
glIsList
glListBase
glNewList

Managing Modes and Execution

The effect of many OpenGL functions depends on whether a particular mode is in effect. The **glEnable** and **glDisable** functions set such modes; **glIsEnabled** determines whether a particular mode is set.

You can control the execution of previously issued OpenGL functions with **glFinish**, which forces all such functions to finish, or **glFlush**, which ensures that all such functions will be completed in a finite time.

In a particular implementation of OpenGL, you may be able to control certain behaviors with hints by using **glHint**. Such 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.

Modes and Execution Reference

glDisable
glEnable
glFinish
glFlush
glHint
glIsEnabled

Obtaining State Information

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

glGetClipPlane	**glGetPixelMap**	**glGetTexImage**
glGetLight	**glGetPolygonStipple**	**glGetTexLevelParameter**
glGetMap	**glGetTexEnv**	**glGetTexParameter**
glGetMaterial	**glGetTexGen**	

To obtain the value of other state variables, use **glGetBooleanv**, **glGetDoublev**, **glGetFloatv**, or **glGetIntegerv**, as appropriate. See **glGet** for information about how to use these functions. Other query functions you might want to use are **glGetError**, **glGetString**, and **glIsEnabled**. (See Handling Errors for more information about functions related to error handling.) To save and restore sets of state variables, use **glPushAttrib** and **glPopAttrib**.

Using the Query Functions

There are four query functions for obtaining simple state variables and one for determining whether a particular state is enabled or disabled:

glGetBooleanv
glGetIntegerv
glGetFloatv
glGetDoublev
glIsEnabled

The prototypes for the query functions are:

void glGetBooleanv(GLenum *pname*, **GLboolean** **params*);

void glGetIntegerv(GLenum *pname*, **GLint** **params*);

void glGetFloatv(GLenum *pname*, **GLfloat** **params*);

void glGetDoublev(GLenum *pname*, **GLdouble** **params*);

Respectively, the query functions obtain Boolean, integer, floating-point, or double-precision state variables. The *pname* parameter is a symbolic constant indicating the state variable to return, and *params* is a pointer to an array of the indicated type in which to place the returned data. The possible values for *pname* are listed in OpenGL State Variables. A type conversion is performed if necessary to return the desired variable as the requested data type.

The prototype for **glIsEnabled** is:

GLboolean glIsEnabled(GLenum *cap*);

If the mode specified by *cap* is enabled, **glIsEnabled** returns GL_TRUE. If the mode specified by *cap* is disabled, **glIsEnabled** returns GL_FALSE. The possible values for *cap* are listed in OpenGL State Variables.

Other specialized functions return specific state variables. To find out when to use these functions, see OpenGL State Variables and the *OpenGL Reference Manual*. For more information on OpenGL's error handling facility and the **glGetError** function, see Error Handling.

The functions that return specific state variables are:

glGetClipPlane
glGetError
glGetLight
glGetMap
glGetMaterial
glGetPixelMap
glGetPolygonStipple
glGetString
glGetTexEnv
glGetTexGen
glGetTexImage
glGetTexLevelParameter
glGetTexParameter

Error Handling

When OpenGL detects an error, it records a current error code. The function that caused the error is ignored, so it has no effect on the OpenGL state or on the frame-buffer contents. (If the error recorded was GL_OUT_OF_MEMORY, however, the results of the function are undefined.) Once recorded, the current error code isn't cleared until you call the **glGetError** query function, which returns the current error code.

Implementations of OpenGL may return multiple current error codes, each of which remains set until queried. The **glGetError** function returns GL_NO_ERROR once you've queried all the current error codes or if there is no error. Therefore, if you obtain an error code, call **glGetError** until GL_NO_ERROR is returned to be sure you've discovered all the errors. For the list of error codes, see OpenGL error codes.

You can use the **gluErrorString** GLU function to obtain a descriptive string corresponding to the error code passed in. For more information on **gluErrorString**, see Handling Errors.

> **Note** GLU functions often return error values if an error is detected. Also, the OpenGL Utility library defines the error codes GLU_INVALID_ENUM, GLU_INVALID_VALUE, and GLU_OUT_OF_MEMORY, which have the same meaning as the related OpenGL error codes.

OpenGL Error Codes

OpenGL includes the following error codes:

Error Code	Description
GL_INVALID_ENUM	**GLenum** argument out of range.
GL_INVALID_VALUE	Numeric argument out of range.
GL_INVALID_OPERATION	Operation illegal in current state.
GL_STACK_OVERFLOW	Function would cause a stack overflow.
GL_STACK_UNDERFLOW	Function would cause a stack underflow.
GL_OUT_OF_MEMORY	Not enough memory left to execute function.

Saving and Restoring Sets of State Variables

You can save and restore the values of a collection of state variables on an attribute stack with the **glPushAttrib** and **glPopAttrib** functions. The attribute stack has a depth of at least 16. To obtain the actual depth, use GL_MAX_ATTRIB_STACK_DEPTH with **glGetIntegerv**. Pushing a full stack or popping an empty one generates an error.

It's generally faster to use **glPushAttrib** and **glPopAttrib** than to get and restore the values yourself. Some values might be pushed and popped in the hardware, and saving and restoring them can be expensive. Also, if you're operating on a remote client, all the attribute data must be transferred across the network connection and back as it's saved and restored. However, your OpenGL implementation keeps the attribute stack on the server, avoiding unnecessary network delays.

The prototype of **glPushAttrib** is:

void glPushAttrib(GLbitfield *mask***);**

Using **glPushAttrib** saves all the attributes indicated by bits in *mask* by pushing them onto the attribute stack. For a list of the possible mask bits you can logically OR together to save any combination of attributes, see Attribute Groups. Each bit corresponds to a collection of individual state variables. For example, GL_LIGHTING_BIT refers to all the state variables related to lighting, which include the current material color; the ambient, diffuse, specular, and emitted light; a list of the lights that are enabled; and the directions of the spotlights. When you call **glPopAttrib**, all those variables are restored. To find out exactly which attributes are saved for particular mask values, see OpenGL State Variables.

OpenGL State Variables

The following topics list the names of state variables that can be queried:

For each variable, the topic lists a description, attribute group, initial or minimum value, and the suggested **glGet*** function to use for obtaining it.

State variables that you can obtain using **glGetBooleanv**, **glGetIntegerv**, **glGetFloatv**, or **glGetDoublev** are listed with just one of these functions—the one that is most appropriate for the type of data to be returned. You cannot obtain these state variables using **glIsEnabled**. However, you can obtain state variables for which **glIsEnabled** is listed as the query function with **glGetBooleanv**, **glGetIntegerv**, **glGetFloatv**, and **glGetDoublev**. You can obtain state variables for which any other function is listed as the query function only by using that function. If no attribute group is listed, the variable doesn't belong to any group. All state variables that you can query, except those that are implementation dependent, have initial values. To determine the initial value of a variable for which no initial value is listed, see that variable's reference topic.

State Information Reference

The following list contains the functions that get state information.

For a list of the reference pages for the state variables, see OpenGL State Variables.

glGet
glGetBooleanv
glGetClipPlane
glGetDoublev
glGetError
glGetFloatv
glGetIntegerv
glIsEnabled
glGetLight
glGetMap
glGetMaterial
glGetPixelMap
glGetPolygonStipple
glGetString
glGetTexEnv
glGetTexGen
glGetTexImage
glGetTexLevelParameter
glGetTexParameter
glPopAttrib
glPushAttrib

OpenGL Utility library

The OpenGL Utility (GLU) library contains several groups of functions that complement the core OpenGL interface by providing support for auxiliary features. These utility functions make use of core OpenGL functions, so any OpenGL implementation is guaranteed to support the utility functions.

Note The prefix for Utility library functions is "glu" rather than "gl."

For more detailed descriptions of these functions, see the *OpenGL Reference Manual*. This section groups related GLU functions, as follows:

- Initializing
- Manipulating Images for Use in Texturing
- Transforming Coordinates
- Tessellating Polygons
- Using Callback Functions
- Using Tessellation Objects
- Specifying Callbacks
- Specifying the Polygon to Be Tessellated
- Rendering Simple Surfaces
- Using NURBS Curves and Surfaces
- Handling Errors

Initializing

With GLU version 1.1 or later, **gluGetString** returns the version number of the GLU library or the version number and any vendor-specific GLU extension calls.

The prototype of **gluGetString** is:

const GLubyte *gluGetString(GLenum *name***);**

Manipulating Images for Use in Texturing

The OpenGL Utility library (GLU) provides image scaling and automatic mipmapping functions to simplify the specification of texture images. The **gluScaleImage** function scales a specified image to an accepted texture size; you can pass the resulting image to OpenGL as a texture. The automatic mipmapping functions, **gluBuild1DMipmaps** and **gluBuild2DMipmaps**, create mipmapped texture images from a specified image and pass them to **glTexImage1D** and **glTexImage2D**, respectively.

Transforming Coordinates

The OpenGL Utility library (GLU) provides several commonly used matrix transformation functions. You can set up a two-dimensional orthographic viewing region with **gluOrtho2D**, a standard perspective view volume using **gluPerspective**, or a view volume that is centered on a specified eyepoint with **gluLookAt**. Each of these functions creates the desired matrix and applies it to the current matrix using **glMultMatrix**.

The **gluPickMatrix** function simplifies selection of a picking matrix by creating a matrix that restricts drawing to a small region of the viewport. If you re-render 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 will be stored in the selection buffer. For more information about selection mode, see **Performing Selection and Feedback**.

To determine where in the window an object is being drawn, use **gluProject**, which converts the specified object coordinates *objx*, *objy*, and *objz* into window coordinates using *modelMatrix*, *projMatrix*, and *viewport*. The result is stored in *winx*, *winy*, and *winz*. If the function succeeds, the return value is GL_TRUE. If the function fails, the return value is GL_FALSE.

The **gluUnProject** function performs the inverse conversion: it transforms the specified window coordinates *winx*, *winy*, and *winz* into object coordinates using *modelMatrix*, *projMatrix*, and *viewport*. The result is stored in *objx*, *objy*, and *objz*. If the function succeeds, the return value is GL_TRUE. If the function fails, the return value is GL_FALSE.

Tessellating Polygons

OpenGL can directly display only simple convex polygons. A polygon is simple if:

- The edges intersect only at vertices.
- There are no duplicate vertices.
- Exactly two edges meet at any vertex.

To display simple nonconvex polygons or simple polygons containing holes, you must first triangulate the polygons (subdivide them into convex polygons). Such subdivision is called *tessellation*. GLU provides a collection of functions that perform tessellation. Note that the GLU tessellation functions can't handle nonsimple polygons; there is no standard OpenGL method to handle such polygons.

Because tessellation is often required and can be rather tricky, this section describes the GLU tessellation functions in detail. These functions take as input arbitrary simple polygons that might include holes, and they return some combination of triangles, triangle meshes, and triangle fans. If you don't want to deal with meshes or fans, you can specify that the tessellation functions return only triangles. However, mesh and fan information improves performance. The polygon tessellation functions triangulate a concave polygon with one or more contours.

To use polygon tessellation

1. Create a tessellation object with **gluNewTess**.
2. Use **gluTessCallBack** to define callback functions you will use to process the triangles generated by the tessellator.
3. With **gluBeginPolygon**, **gluTessVertex**, **gluNextContour**, and **gluEndPolygon**, specify the polygon with holes or the concave polygon to be tessellated.

   When the polygon description is complete, the tessellation facility invokes your callback functions as necessary.

   You can destroy unneeded tessellation objects with **gluDeleteTess**.

For more information on saving the tessellation data, see Using Callback Functions.

Using Callback Functions

The GLU callback functions, **gluBeginPolygon**, **gluTessVertex**, **gluNextContour**, and **gluEndPolygon**, are similar to the OpenGL polygon functions.

They typically save the data for the triangles, triangle meshes, and triangle fans in user-defined data structures or in OpenGL display lists. To render the polygons, other code traverses the data structures or calls the display lists. Although the callback functions could call OpenGL functions to display polygons directly, this is usually not done, as tessellation can be computationally expensive. It's a good idea to save the data if there is any chance that you want to display it again. The GLU tessellation functions are guaranteed never to return any new vertices, so interpolation of vertices, texture coordinates, or colors is never required.

Using Tessellation Objects

As a complex polygon is being described and tessellated, it requires associated data, such as the vertices, edges, and callback functions. All this data is tied to a single tessellation object. To tessellate a polygon, you first use the **gluNewTess** function which creates a new tessellation object and returns a pointer to it. A null pointer is returned if the function fails.

If you no longer need a tessellation object, you can delete it and free all associated memory with **gluDeleteTess**.

You can reuse a single tessellation object for all your tessellations. This object is required only because library functions may need to do their own tessellations, and they should be able to do so without interfering with any tessellation that your program is performing. Multiple tessellation objects are also useful if you want to use different sets of callbacks for different tessellations. Typically, however, you allocate a single tessellation object and use it for all the tessellations. There's no real need to free it, because it uses a small amount of memory. On the other hand, if you're writing a library function that uses GLU tessellation, be careful to free any tessellation objects you create.

Specifying Callbacks

You can specify up to five callback functions for a tessellation. Any functions that you do not specify are not called during the tessellation, and you do not get any information they might have returned. You specify the callback functions with **gluTessCallback**.

The **gluTessCallback** function associates the callback function *fn* with the tessellation object *tessobj*. The type of the callback is determined by the parameter *type*, which can be GLU_BEGIN, GLU_EDGE_FLAG, GLU_VERTEX, GLU_END, or GLU_ERROR. The five possible callback functions have the following prototypes.

Callback Function	Prototype
GLU_BEGIN	**void begin(GLenum** *type***);**
GLU_EDGE_FLAG	**void edgeFlag(GLboolean** *flag***);**
GLU_VERTEX	**void vertex(void** **data***);**
GLU_END	**void end(***void***);**
GLU_ERROR	**void error(GLenum** *errno***);**

To change a callback function, call **gluTessCallback** with the new function. To eliminate a callback function without replacing it with a new one, pass **gluTessCallback** a null pointer for the appropriate function.

As tessellation proceeds, the callback functions are called in a manner similar to the way you would use the OpenGL functions **glBegin**, **glEdgeFlag**, **glVertex**, and **glEnd**.

The GLU_BEGIN callback function is invoked with one of three possible parameters:

- GL_TRIANGLE_FAN
- GL_TRIANGLE_STRIP
- GL_TRIANGLES

After calling the GLU_BEGIN callback function, and before calling the callback function associated with GLU_END, some combination of the GLU_EDGE_FLAG and GLU_VERTEX callbacks is invoked. The associated vertices and edge flags are interpreted exactly as they are in OpenGL between **glBegin(**GL_TRIANGLE_FAN**)**, **glBegin(**GL_TRIANGLE_STRIP**)**, or **glBegin(**GL_TRIANGLES**)** and the matching **glEnd**.

Because edge flags make no sense in a triangle fan or triangle strip, if there is a callback function associated with GLU_EDGE_FLAG, the GLU_BEGIN callback is called only with GL_TRIANGLES. The GLU_EDGE_FLAG callback function works analogously to the OpenGL **glEdgeFlag** function.

If there is an error during the tessellation, the error callback function is invoked. The error callback function is passed a GLU error number. You can obtain a character string describing the error with the **gluErrorString** function.

Specifying the Polygon to Be Tessellated

You specify a polygon (possibly containing holes) to be tessellated using:

gluBeginPolygon
gluTessVertex
gluNextContour
gluEndPolygon

For polygons without holes, the specification process is exactly as in OpenGL:

1. Start with **gluBeginPolygon**.
2. Call **gluTessVertex** for each vertex in the boundary.
3. End the polygon with a call to **gluEndPolygon**.

If a polygon consists of multiple contours, including holes and holes within holes, you specify the contours one after the other, preceding each by **gluNextContour**. When you call **gluEndPolygon**, it signals the end of the final contour and starts the tessellation. You can omit the call to **gluNextContour** before the first contour. The **gluBeginPolygon** function begins the specification of a polygon to be tessellated and associates a tessellation object, *tessobj*, with it. The callback functions to be used are those that you bind to the tessellation object with **gluTessCallback**.

The **gluTessVertex** function specifies a vertex in the polygon to be tessellated. Call **gluTessVertex** for each vertex in the polygon. The function's *tessobj* parameter is the tessellation object to use, *v* contains the three-dimensional vertex coordinates, and *data* is an arbitrary pointer that is sent to the callback associated with GLU_VERTEX. Typically, *data* contains vertex data, texture coordinates, color information, or whatever else the application may require.

The **gluNextContour** function marks the beginning of the next contour when multiple contours make up the boundary of the polygon to be tessellated.The function's *type* parameter can be GLU_EXTERIOR, GLU_INTERIOR, GLU_CCW, GLU_CW, or GLU_UNKNOWN. These constants serve only as hints to the tessellation. If you get them right, the tessellation might go faster. If you get them wrong, they're ignored, and the tessellation still works.

For a polygon with holes, one contour is the exterior contour, and the others are interior. If you don't call **gluNextContour** immediately after **gluBeginPolygon**, the first contour is assumed to be of type GLU_EXTERIOR.

GLU_CW and GLU_CCW indicate clockwise- and counterclockwise-oriented polygons. Choosing which are clockwise and which are counterclockwise is arbitrary in three dimensions, but in any plane, there are two different orientations; use the GLU_CW and GLU_CCW types consistently. Use GLU_UNKNOWN if you don't know which to use.

The **gluEndPolygon** function indicates the end of the polygon specification. It also indicates that the tessellation can begin using the tessellation object *tessobj*.

Rendering Simple Surfaces

The GLU library includes a set of functions for drawing various simple surfaces (spheres, cylinders, disks, and parts of disks) in a variety of styles and orientations. These functions are described in detail in the *OpenGL Reference Manual.*

To render simple surfaces

1. Create a quadric object with **gluNewQuadric**.

   To destroy this object when you're finished with it, use **gluDeleteQuadric**.
2. Specify the desired rendering style, as listed below, with the appropriate function (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**
3. After specifying the rendering style, invoke the rendering function for the desired type of quadric object: **gluSphere**, **gluCylinder**, **gluDisk**, or **gluPartialDisk**.

   If an error occurs during rendering, the error-handling function you've specified with **gluQuadricCallBack** is invoked.

Use the ***Radius**, *height*, and similar arguments, rather than the **glScale** function, to scale the quadrics, so that you don't have to renormalize any unit-length normals that are generated. To force lighting calculations at a finer granularity, especially if the material specularity is high, set the *loops* and *stacks* arguments to values other than 1.

Using NURBS Curves and Surfaces

Non-Uniform Rational B-Spline (NURBS) functions provide general and powerful descriptions of curves and surfaces in two and three dimensions, converting the curves and surfaces to OpenGL evaluators. The NURBS functions can represent geometry in many computer-aided mechanical design systems. They can render curves and surfaces in a variety of styles, and they can automatically handle adaptive subdivision that tessellates the domain into smaller triangles in regions of high curvature and near silhouette edges. NURBS functions fall into the following categories.

To manage a NURBS object, use:

- **gluNewNurbsRenderer** (create a NURBS object)
- **gluDeleteNurbsRenderer** (deletes a NURBS object)
- **gluNurbsCallback** (establishes an error-handling function)

To specify the desired curves, use:

- **gluBeginCurve**
- **gluNurbsCurve**
- **gluEndCurve**

To specify the desired surfaces, use:

- **gluBeginSurface**
- **gluNurbsSurface**
- **gluEndSurface**

You can also specify a trimming region, which defines a subset of the NURBS surface domain to be evaluated so you can create surfaces that have smooth boundaries or that contain holes.

To specify the trimming region, use:

- **gluBeginTrim**
- **gluPwlCurve**
- **gluNurbsCurve**
- **gluEndTrim**

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

- Whether to discard a curve or surface whose control polyhedron lies outside the current viewport.
- The maximum length (in pixels) of edges of polygons used to render curves and surfaces.
- Whether you will take the projection matrix, modelview matrix, and viewport from the OpenGL server or supply them explictly 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 **gluErrorString** function retrieves error strings that correspond to OpenGL or GLU error codes. The currently defined OpenGL error codes are described in **glGetError**. The GLU error codes are listed in **gluErrorString**, **gluTessCallback**, **gluQuadricCallback**, and **gluNurbsCallback**.

The return value for **gluErrorString** is a pointer to a descriptive string that corresponds to the OpenGL, GLU, or GLX error number passed in the *errorCode* parameter. The defined error codes are described in the *OpenGL Reference Manual* along with the function or function that can generate them.

OpenGL on Windows NT and Windows 95

The Microsoft implementation of OpenGL in the Microsoft® Windows NT® and Windows® 95 operating systems is an implementation of the industry-standard OpenGL three-dimensional (3-D) graphics software interface with which programmers create high-quality still and animated 3-D color images. This overview describes the Windows NT and Windows 95 implementation of OpenGL.

Components

Microsoft's implementation of OpenGL in Windows NT and Windows 95 includes the following components:

- The full set of current OpenGL commands

  OpenGL contains a library of core functions for 3-D graphics operations. These basic functions are used to manage object shape description, matrix transformation, lighting, coloring, texture, clipping, bitmaps, fog, and antialiasing. The names for these core functions have a "gl" prefix.

  Many of the OpenGL commands have several variants, which differ in the number and type of their parameters. Counting all the variants, there are more than 300 OpenGL commands.
- The OpenGL Utility (GLU) library

  This library of auxiliary functions complements the core OpenGL functions. The commands manage texture support, coordinate transformation, polygon tessellation, rendering spheres, cylinders and disks, NURBS (Non-Uniform Rational B-Spline) curves and surfaces, and error handling.
- The OpenGL Programming Guide Auxiliary library

  This is a simple, platform-independent library of functions for managing windows, handling input events, drawing classic 3-D objects, managing a background process, and running a program. The window management and input routines provide a base level of functionality with which you can quickly get started programming in OpenGL.

  Do not use them, however, in a production application. Here are some reasons for this warning:
  - The message loop is in the library code.
  - There is no way to add handlers for additional WM* messages.
  - There is very little support for logical palettes.

  The library is described and used in the *OpenGL Programming Guide*.
- The WGL functions

  This set of functions connects OpenGL to the Windows NT and Windows 95 windowing system. The functions manage rendering contexts, display lists, extension functions, and font bitmaps. The WGL functions are analogous to the GLX extensions that connect OpenGL to the X Window System. The names for these functions have a "wgl" prefix.
- New Win32 functions for pixel formats and double buffering

  These functions support per-window pixel formats and double buffering (for smooth image changes) of windows. These new functions apply only to OpenGL graphics windows.

Generic Implementation and Hardware Implementations

This overview discusses the current generic implementation of OpenGL in Windows NT and Windows 95, which is the Microsoft Windows NT and Windows 95 software implementation of OpenGL. Hardware manufacturers may enhance parts of OpenGL in their drivers and may support some features not supported by the generic implementation.

Limitations

The generic implementation has the following limitations:

- Printing.

  You can print an OpenGL image directly to a printer using metafiles only. For more information, see Printing an OpenGL Image.
- OpenGL and GDI graphics cannot be mixed in a double-buffered window.

  An application can directly draw both OpenGL graphics and GDI graphics into a single-buffered window, but not into a double-buffered window.
- There are no per-window hardware color palettes.

  Windows NT and Windows 95 have a single system hardware color palette, which applies to the whole screen. An OpenGL window cannot have its own hardware palette, but can have its own logical palette. To do so, it must become a palette-aware application. For more information, see OpenGL Color Modes and Windows Palette Management.
- There is no direct support for the Clipboard, dynamic data exchange (DDE), or OLE.

  A window with OpenGL graphics does not directly support these Windows NT and Windows 95 capabilities. There are workarounds, however, for using the Clipboard. For more information, see Copying an OpenGL Image to the Clipboard.
- The Inventor 2.0 C++ class library is not included.

  The Inventor class library, built on top of OpenGL, provides higher-level constructs for programming 3-D graphics. It is not included in the current version of Microsoft's implementation of OpenGL for Windows NT and Windows 95.
- There is no support for the following pixel format features: stereoscopic images, alpha bitplanes, and auxiliary buffers.

  There is, however, support for several ancillary buffers including: stencil buffer, accumulation buffer, back buffer (double buffering), overlay and underlay plane buffer, and depth (z-axis) buffer.

Guide to Documentation

The documentation set for OpenGL in Windows NT and Windows 95 includes five elements.

The first two elements are the official OpenGL books: the *OpenGL Reference Manual* and the *OpenGL Programming Guide*.

Note The *OpenGL Reference Manual* and the *OpenGL Programming Guide* are not included with the Win32® application programming interface SDK.

The *OpenGL Reference Manual* includes an overview of how OpenGL works and a set of detailed reference pages. The reference pages cover all the 115 distinct OpenGL functions, as well as the 43 functions in the OpenGL Utility (GLU) library.

The *OpenGL Programming Guide* explains how to create graphics programs using OpenGL. It includes discussions of the following major topics:

- Drawing geometric shapes
- Pixels, bitmaps, fonts, and images
- Viewing and matrix transformations
- Texture mapping
- Display lists
- Advanced composite techniques
- Color
- Evaluators and NURBS
- Lighting
- Selection and feedback
- Blending, antialiasing, and fog
- Advanced techniques

In addition, the *OpenGL Programming Guide* contains appendixes that discuss the OpenGL Utility library and the OpenGL Programming Guide Auxiliary library.

The third documentation element is this overview. It describes the Windows NT and Windows 95 implementation of OpenGL and provides an overview of its components. It discusses the concepts of rendering contexts, pixel formats, and buffers; the WGL functions that connect OpenGL to the Windows NT and Windows 95 windowing systems; and the Win32 functions that support per-window pixel formats and double buffering of windows for OpenGL graphics windows. The WGL functions and the Win32 functions are specific to the Windows NT and Windows 95 implementation of OpenGL.

The fourth documentation element is the set of reference pages for the WGL functions, the Win32 functions just mentioned, and the **PIXELFORMATDESCRIPTOR** data structure.

The fifth documentation element is *Porting to OpenGL*. It discusses moving existing OpenGL code from other environments into Windows NT and Windows 95.

Rendering Contexts

An OpenGL *rendering context* is a port through which all OpenGL commands pass. Every thread that makes OpenGL calls must have a current rendering context. Rendering contexts link OpenGL to the Windows NT and Windows 95 windowing systems.

An application specifies a Windows NT or Windows 95 device context when it creates a rendering context. This rendering context is suitable for drawing on the device that the specified device context references. In particular, the rendering context has the same pixel format as the device context. For more information, see Rendering Context Functions.

Despite this relationship, a rendering context is not the same as a device context. A device context contains information pertinent to the graphics component (GDI) of Windows NT and Windows 95. A rendering context contains information pertinent to OpenGL. A device context must be explicitly specified in a GDI call. A rendering context is implicit in an OpenGL call. You should set a device context's pixel format before creating a rendering context.

A thread that makes OpenGL calls must have a current rendering context. If an application makes OpenGL calls from a thread that lacks a current rendering context, nothing happens; the call has no effect. An application commonly creates a rendering context, sets it as a thread's current rendering context, and then calls OpenGL functions. When it finishes calling OpenGL functions, the application uncouples the rendering context from the thread, and then deletes the rendering context. A window can have multiple rendering contexts drawing to it at one time, but a thread can have only one current, active rendering context.

A current rendering context has an associated device context. That device context need not be the same device context as that used when the rendering context was created, but it must reference the same device and have the same pixel format.

A thread can have only one current rendering context. A rendering context can be current to only one thread.

Rendering Context Functions

Five WGL functions manage rendering contexts, as described in the following table.

WGL Function	Description
wglCreateContext	Creates a new rendering context.
wglMakeCurrent	Sets a thread's current rendering context.
wglGetCurrentContext	Obtains a handle to a thread's current rendering context.
wglGetCurrentDC	Obtains a handle to the device context associated with a thread's current rendering context.
wglDeleteContext	Deletes a rendering context.

The **wglCreateContext** function takes a device context handle as its parameter and returns a rendering context handle. The created rendering context is suitable for drawing on the device referenced by the device context handle. In particular, its pixel format is the same as the device context's pixel format. After you create a rendering context, you can release or dispose of the device context. See Device Contexts for more details on creating, obtaining, releasing, and disposing of a device context.

> **Note** The device context sent to **wglCreateContext** must be a display device context, a memory device context, or a color printer device context that uses four or more bits per pixel. The device context cannot be a monochrome printer device context.

The **wglMakeCurrent** function takes a rendering context handle and a device context handle as parameters. All subsequent OpenGL calls made by the thread are made through that rendering context, and are drawn on the device referenced by that device context. The device context does not have to be the same one passed to **wglCreateContext** when the rendering context was created, but it must be on the same device and have the same pixel format. The call to **wglMakeCurrent** creates an association between the supplied rendering context and device context. You cannot release or dispose of the device context associated with a current rendering context until after you make the rendering context not current.

Once a thread has a current rendering context, it can make OpenGL graphics calls. All calls must pass through a rendering context. Nothing happens if you make OpenGL graphics calls from a thread that lacks a current rendering context.

The **wglGetCurrentContext** function takes no parameters, and returns a handle to the calling thread's current rendering context. If the thread has no current rendering context, the return value is NULL.

When you obtain a handle to the device context associated with a thread's current rendering context by calling **wglGetCurrentDC**, the association is created when a rendering context is made current.

You can break the association between a current rendering context and a thread by calling **wglMakeCurrent** with either of two handles:

- A null rendering context handle
- A handle other than the one originally called

After calling **wglMakeCurrent** with the rendering context handle parameter set to NULL, the calling thread has no current rendering context. The rendering context is released from its connection to the thread, and the rendering context's association to a device context ends. OpenGL flushes all drawing commands, and may release some resources. No OpenGL drawing will be done until the next call to **wglMakeCurrent**, because the thread can make no OpenGL graphics calls until it regains a current rendering context.

The second way to break the association between a rendering context and a thread is to call **wglMakeCurrent** with a different rendering context. After such a call, the calling thread has a new current rendering context, the former current rendering context is released from its connection to the thread, and the former current rendering context's association to a device context ends.

The **wglDeleteContext** function takes a single parameter, the handle to the rendering context to be deleted. Before calling **wglDeleteContext**, make the rendering context not current by calling **wglMakeCurrent**, and delete or release the associated device context by calling **DeleteDC** or **ReleaseDC** as appropriate.

It is an error for a thread to delete a rendering context that is another thread's current rendering context. However, if a rendering context is the calling thread's current rendering context, **wglDeleteContext** flushes all OpenGL drawing commands and makes the rendering context not current before deleting it. In this case, relying on **wglDeleteContext** to make a rendering context not current requires the programmer to delete or release the associated device context.

Pixel Formats

A *pixel format* specifies several properties of an OpenGL drawing surface. Some of the properties specified by a pixel format are:

- Whether the pixel buffer is single- or double-buffered.
- Whether the pixel data is in RGBA or color-index form.
- The number of bits used to store color data.
- The number of bits used for the depth (z-axis) buffer.
- The number of bits used for the stencil buffer.
- The number of overlay and underlay planes.
- Various visibility masks.

Microsoft's implementation of OpenGL for Windows NT and Windows 95 uses the **PIXELFORMATDESCRIPTOR** data structure to convey pixel format data. The structure's members specify the preceding properties and several others.

A given device context can support several pixel formats. Windows NT and Windows 95 identify the pixel formats that a device context supports with consecutive one-based index values (1, 2, 3, 4, and so on). A device context can have just one current pixel format, chosen from the set of pixel formats it supports.

Each window has its own current pixel format in OpenGL in Windows NT and Windows 95. This means, for example, that an application can simultaneously display RGBA and color-index OpenGL windows, or single- and double-buffered OpenGL windows. This per-window pixel format capability is limited to OpenGL windows.

Typically, you obtain a device context, set the device context's pixel format, and then create an OpenGL rendering context suitable for that device.

Note You set the pixel format before creating a rendering context because the rendering context inherits the device context's pixel format.

Pixel Format Functions

The following Win32 functions manage pixel formats.

Win32 Function	Description
ChoosePixelFormat	Obtains the device context's pixel format that is the closest match to a specified pixel format.
SetPixelFormat	Sets a device context's current pixel format to the pixel format specified by a pixel format index.
GetPixelFormat	Obtains the pixel format index of a device context's current pixel format.
DescribePixelFormat	Given a device context and a pixel format index, fills in a PIXELFORMATDESCRIPTOR data structure with the pixel format's properties.
GetEnhMetaFilePixelFormat	Retrieves pixel format information for an enhanced metafile.

The **ChoosePixelFormat** function returns a one-based pixel format index that identifies the best match from the device context's supported pixel formats.

The **SetPixelFormat** function identifies the desired format using a one-based pixel format index. Typically, you call **ChoosePixelFormat** to find a best-match pixel format, and then call **SetPixelFormat** with the result of **ChoosePixelFormat**.

If you call **SetPixelFormat** for a device context that references a window, **SetPixelFormat** also changes the pixel format of the window. Setting the pixel format of a window more than once can lead to significant complications for the Window Manager and for multithread applications, so it is not allowed. You can set the pixel format of a window only one time; after that, the window's pixel format cannot be changed.

The **GetPixelFormat** function returns a one-based pixel format index.

The **DescribePixelFormat** function takes the following as parameters:

- A handle to a device context
- A pixel format index
- A pointer to a **PIXELFORMATDESCRIPTOR** data structure

The **DescribePixelFormat** function returns with the members of **PIXELFORMATDESCRIPTOR** appropriately set.

The **GetEnhMetaFilePixelFormat** function returns the size of a metafile's pixel format and retrieves the pixel format information of the metafile.

Front, Back, and Other Buffers

OpenGL stores and manipulates pixel data in a frame buffer. The frame buffer consists of a set of logical buffers: color, depth, accumulation, and stencil buffers. The color buffer itself consists of a set of logical buffers; this set can include a front-left, a front-right, a back-left, a back-right, and some number of auxiliary buffers. A particular pixel format or OpenGL implementation may not supply all of these buffers. For example, the current version of Microsoft's implementation of OpenGL in Windows NT and Windows 95 does not support stereoscopic images, so a pixel format cannot have left and right color buffers. In addition, the current version does not support auxiliary buffers. For more information on OpenGL buffers and the OpenGL functions that operate on them, see the *OpenGL Reference Manual* and the *OpenGL Programming Guide*.

Microsoft's implementation of OpenGL in Windows NT and Windows 95 supports double buffering of images. This is a technique in which an application draws pixels to an off-screen buffer, and then, when that image is ready for display, copies the contents of the off-screen buffer to an on-screen buffer. Double buffering enables smooth image changes, which are especially important for animated images.

Two color buffers are available to applications that use double buffering: a front buffer and a back buffer. By default, drawing commands are directed to the back buffer (the off-screen buffer), while the front buffer is displayed on the screen. When the off-screen buffer is ready for display, you call **SwapBuffers**, and Windows NT or Windows 95 copies the contents of the off-screen buffer to the on-screen buffer.

The generic implementation uses a device-independent bitmap (DIB) as the back buffer and the screen display as the front buffer. Hardware devices and their drivers may use different approaches.

Double buffering is a pixel-format property. To request double buffering for a pixel format, set the PFD_DOUBLEBUFFER flag in the **PIXELFORMATDESCRIPTOR** data structure in a call to **ChoosePixelFormat**.

The OpenGL core function, **glDrawBuffer**, selects buffers for writing and clearing.

Buffer Functions

To copy the contents of an off-screen buffer to an on-screen buffer, call **SwapBuffers**. The **SwapBuffers** function takes a handle to a device context. The current pixel format for the specified device context must include a back buffer. By default, the back buffer is off-screen, and the front buffer is on-screen.

> **Note** The **SwapBuffers** function does not really swap the contents of the two buffers, but rather copies the contents of one buffer to another. The contents of the off-screen buffer are undefined after a call to **SwapBuffers**. Thus, the result of two consecutive calls to **SwapBuffers** is undefined.

The following illustration shows how the contents of the buffers are copied when calling **SwapBuffers**.

{ewc msdncd, EWGraphic, bsd23542 0 /a "SDK.WMF"}

Several OpenGL core functions also manage buffers. The **glDrawBuffer** function is the one most relevant to double buffering; it specifies the frame buffer or buffers that OpenGL draws into.

The following functions also affect buffers:

glReadBuffer
glReadPixels
glCopyPixels
glAccum
glColorMask
glDepthMask
glIndexMask
glStencilMask
glClearAccum
glClearColor
glClearDepth
glClearIndex
glClearStencil

Fonts and Text

Microsoft's implementation of OpenGL in Windows NT and Windows 95 supports GDI graphics in a single-buffered OpenGL window. It does not support GDI graphics in a double-buffered OpenGL window. Thus, you can call only the standard GDI font and text functions to draw text in a single-buffered OpenGL window; you cannot call those functions to draw text in a double-buffered OpenGL window.

There is a workaround for this restriction on text in double-buffered windows: build OpenGL display lists for bitmap images of characters, and then execute those display lists to draw characters. There are three main steps in this process:

1. Select a font for a device context, setting the font's properties as desired.
2. Create a set of bitmap display lists based on the glyphs in the device context's font, one display list for each glyph that the application will draw.
3. Draw each glyph in a string, using those bitmap display lists.

To create the display lists, call the **wglUseFontBitmaps** and **wglUseFontOutlines** functions. To draw characters in a string using those display lists, call **glCallLists**.

To create applications that are easy to localize and that use resources sparingly, the creation and storage of these glyph image display lists must be managed carefully. Many languages, unlike English, have alphabets whose character codes range over a relatively large set of values.

Font and Text Functions

Two functions can be used to manage fonts and text.

Win32 Function	Description
wglUseFontBitmaps	Creates a set of character bitmap display lists. Characters come from a specified device context's current font. Characters are specified as a consecutive run within the font's glyph set.
wglUseFontOutlines	Creates a set of display lists, based on the glyphs of the currently selected outline font of a device context, for use with the current rendering context. The display lists are used to draw 3-D characters of TrueType fonts.

The **wglUseFontBitmaps** and **wglUseFontOutlines** functions take a handle to a device context, and use that device context's current font as a source for the bitmaps. It is therefore necessary to set the device context's font and the font's properties before calling **wglUseFontBitmaps** or **wglUseFontOutlines**.

The **wglUseFontBitmaps** and **wglUseFontOutlines** functions also take a parameter that turns the first glyph in the font into a bitmap display list, and a parameter that specifies how many glyphs to turn into display lists. The function then creates display lists for the specified consecutive run of glyphs. For example:

- To create a set of 224 bitmap display lists for all of the Windows NT and Windows 95 character set glyphs, set these two parameters to 32 and 224, respectively.
- To create a set of 256 bitmap display lists for all of the OEM character set glyphs, set these two parameters to 0 and 256, respectively.
- To create a single bitmap display list for any single character set glyph, set the second of these parameters to 1.

The **wglUseFontBitmaps** and **wglUseFontOutlines** functions represent a null glyph in a font with an empty display list.

The display lists created by a call to **wglUseFontBitmaps** or **wglUseFontOutlines** are automatically numbered consecutively.

After calling the **wglUseFontBitmaps** or **wglUseFontOutlines** function, call **glCallLists** to draw a string of characters. See Drawing Text in a Double-Buffered OpenGL Window for sample code. In this context, **glCallLists** uses each character in a string as an index into the array of consecutively numbered display lists created by **wglUseFontBitmaps** or **wglUseFontOutlines**.

When you finish drawing text, call the **glDeleteLists** function to release the contiguous set of display lists created by **wglUseFontBitmaps** and **wglUseFontOutlines**.

OpenGL Color Modes and Windows Palette Management

Microsoft's implementation of OpenGL in Windows NT and Windows 95 supports two color pixel data modes: RGBA and color-index modes. Windows NT and Windows 95 provide two analogous ways of handling color: true color and palette management.

True-color devices, able to accept 16, 24, or more bits of color information per pixel, can display tens of thousands, tens of millions, or more colors simultaneously. Complexities arise, however, when an application has to manage RGBA or color-index mode on a palette-type device. Palette-type devices, such as a 256-color VGA display, are limited in the number of colors they can display simultaneously. Applications must handle a number of tricky details to successfully use palette-type devices. Because color-index mode programs don't use a hardware palette, they are more difficult to use with true-color devices than programs using the RGBA mode.

If you are unfamiliar with true-color devices or the Palette Manager, refer to the articles "Palette Awareness," "The Palette Manager: How and Why," and "Using True-Color devices" on the Microsoft Developer Network Development Library compact discs for an introduction to the essentials of Windows NT and Windows 95 color management.

Palettes and the Palette Manager

The Windows NT and Windows 95 Palette Manager, which is part of the GDI, specifically targets 8-bit display adapters with a *hardware palette* of 256 color entries. Pixels on the screen are stored as an 8-bit index into the hardware palette. Each entry in the hardware palette usually defines a 24-bit color (8 each of red, green, and blue).

The Palette Manager maintains a *system palette* that is a copy of the hardware palette. The system palette is divided into two sections: 20 reserved colors and the remaining 236 colors, which you can set using the Palette Manager.

A default 20-color *logical palette* is selected and realized into a device context. You can create and use a new logical palette. To change the system palette, select and realize the logical palette you created.

You'll probably create a logical palette to specify the colors you want displayed in your OpenGL application. Using certain GDI calls, you can temporarily replace most of the system palette with a logical palette. Using a logical palette, you can define pixel colors for the GDI using either the RGBA or the color-index mode. The maximum size of a logical palette is 256 colors for 8-bit devices and 4,096 colors on a true-color device (16, 24, and 32 bits).

For more information on the RGBA and color-index modes, see RGBA Mode and Windows Palette Management and OpenGL Color Modes and Windows Palette Management.

Palette Awareness

Your application must respond to the **WM_PALETTECHANGED**, **WM_QUERYNEWPALETTE**, and **WM_ACTIVATE** messages to be aware of and use palettes. Design your application to select and realize palettes in response to these messages.

For more information on palettes and palette awareness, see the articles "Palette Awareness," and "The Palette Manager: How and Why" on the Microsoft Developer Network Development Library compact discs.

Reading Color Values from the Frame Buffer

When using functions that read back color values from the frame buffer, be aware of the differences between reading RGBA values and color-index values on true-color devices and on palette-based devices.

On a true-color device:

- RGBA values are limited to the channel in the device.
- Color-index values are stored as RGBA values in the frame buffer. When using these values, you must perform an inverse translation from RGBA to the logical palette index. If two logical indexes have the same RGBA values, the wrong index can be returned.

On a palette-based device:

- RGBA values are read from an index in the system palette. The logical index is obtained from an inverse table, and the RGBA components are extracted.
- Color-index values are read from an index into the system palette and an inverse table is used to get the logical palette index.

Choosing Between RGBA and Color-Index Mode

In general, use the RGBA mode for your OpenGL applications; it provides more flexibility than the color-index mode for effects such as shading, lighting, color mapping, fog, antialiasing, and blending.

Consider using the color-index mode in the following cases:

- If you have a limited number of bitplanes available; the color-index mode can produce less-coarse shading than the RGBA mode.
- If you are not concerned about using "real" colors; for example, using several colors in a topographic map to designate relative elevations.
- When you're porting an existing application that uses color-index mode extensively.
- When you want to use color-map animation and effects in your application. (This is not possible on true-color devices.)

RGBA Mode and Windows Palette Management

While most GDI applications tend to use color-indexing with logical palettes, the RGBA mode is usually preferable for OpenGL applications. It works better than color mapping for several effects, such as shading, lighting, fog, and texture mapping.

The RGBA mode uses red, green, and blue (R, G, and B) color values that together specify the color of each pixel in the display. The R, G, and B values specify the intensity of each color (red, green, and blue); the values range from 0.0 (least intense) to 1.0 (most intense). The number of bits for each component varies depending on the hardware used (2, 3, 5, 6, and 8 bits are possible).The color displayed is a result of the sum of the three color values. If all three values are 0.0, the result is black. If the three values are all 1.0, the result is white. Other colors are a result of a combination of values of R, G, and B that fall between 0 and 1.0. The A (alpha) bit isn't used to specify color.

The standard super-VGA display uses palettes with eight color-bits per pixel. The eight bits are read from the buffer and used as an index in the system palette to get the R, G, and B values. When an RGB palette is selected and realized in a device context, OpenGL can render using the RGBA mode.

Because there are eight color-bits per pixel, OpenGL emphasizes the use of a three-three-two RGBA palette. "Three-three-two" refers to how the color-bit data is handled by the hardware or physical palette. Red (R) and green (G) are each specified by three bits; blue (B) is specified by two bits. Red is the least-significant bit and blue is the most-significant bit.

You determine the colors of your application's logical palette with **PALETTEENTRY** structures. Typically you create an array of **PALETTEENTRY** structures to specify the entire palette entry table of the logical palette.

RGBA Mode Palette Sample

The following code fragment shows how you can create a three-three-two RGBA palette.

```
/*
 * win8map.c - program to create an 8-bit RGB color map for
 * use with OpenGL
 *
 * For OpenGL RGB rendering you need to know red, green, & blue
 * component bit sizes and positions. On 8 bit palette devices you need
 * to create a logical palette that has the correct RGBA values for all
 * 256 possible entries. This program creates an 8 bit RGBA color cube
 * with a default gamma of 1.4
 *
 * Unfortunately, because the standard 20 colors in the system palette
 * cannot be changed,if you select this palette into an 8-bit display
 * DC, you will not realize all of the logical palette. The program
 * changes some of the entries in the logical palette to match enties in
 * the system palette using a least-squares calculation to find which
 * entries to replace
 *
 * Note: Three bits for red & green and two bits for blue; red is the
 * least-significant bit and blue is the most-significant bit
 */

#include <stdio.h>
#include <math.h>

#define DEFAULT_GAMMA 1.4F
```

```
#define MAX_PAL_ERROR (3*256*256L)

struct colorentry {
    unsigned char red;
    unsigned char green;
    unsigned char blue;
};

struct rampentry {
    struct colorentry color;
    long defaultindex;
    unsigned char flags;
};

struct defaultentry {
    struct colorentry color;
    long rampindex;
    unsigned char flags;
};

/* values for flags */
#define EXACTMATCH      0x01
#define CHANGED         0x02        /* one of the default entries is close
*/

/*
 * These arrays hold bit arrays with a gamma of 1.0
 * used to convert n bit values to 8-bit values
 */

unsigned char threeto8[8] = {
    0, 0111>>1, 0222>>1, 0333>>1, 0444>>1, 0555>>1, 0666>>1, 0377
};

unsigned char twoto8[4] = {
    0, 0x55, 0xaa, 0xff
};

unsigned char oneto8[2] = {
    0, 255
};

struct defaultentry defaultpal[20] = {
    { 0,   0,   0 },
    { 0x80,0,   0 },
    { 0,   0x80,0 },
    { 0x80,0x80,0 },
    { 0,   0,   0x80 },
    { 0x80,0,   0x80 },
    { 0,   0x80,0x80 },
    { 0xC0,0xC0,0xC0 },

    { 192, 220, 192 },
    { 166, 202, 240 },
```

```
    { 255, 251, 240 },
    { 160, 160, 164 },

    { 0x80,0x80,0x80 },
    { 0xFF,0,   0 },
    { 0,   0xFF,0 },
    { 0xFF,0xFF,0 },
    { 0,   0,   0xFF },
    { 0xFF,0,   0xFF },
    { 0,   0xFF,0xFF },
    { 0xFF,0xFF,0xFF }
};

struct rampentry rampmap[256];

void
gammacorrect(double gamma)
{
    int i;
    unsigned char v, nv;
    double dv;

    for (i=0; i<8; i++) {
        v = threeto8[i];
        dv = (255.0 * pow(v/255.0, 1.0/gamma)) + 0.5;
    nv = (unsigned char)dv;
    printf("Gamma correct %d to %d (gamma %.2f)\n", v, nv, gamma);
        threeto8[i] = nv;
    }
    for (i=0; i<4; i++) {
        v = twoto8[i];
        dv = (255.0 * pow(v/255.0, 1.0/gamma)) + 0.5;
    nv = (unsigned char)dv;
    printf("Gamma correct %d to %d (gamma %.2f)\n", v, nv, gamma);
        twoto8[i] = nv;
    }
    printf("\n");
}

main(int argc, char *argv[])
{
    long i, j, error, min_error;
    long error_index, delta;
    double gamma;
    struct colorentry *pc;

    if (argc == 2)
        gamma = atof(argv[1]);
    else
        gamma = DEFAULT_GAMMA;

    gammacorrect(gamma);

    /* First create a 256 entry RGB color cube */
```

```
for (i = 0; i < 256; i++) {
    /* BGR: 2:3:3 */
    rampmap[i].color.red = threeto8[(i&7)];
    rampmap[i].color.green = threeto8[((i>>3)&7)];
    rampmap[i].color.blue = twoto8[(i>>6)&3];
}

/* Go through the default palette and find exact matches */
for (i=0; i<20; i++) {
    for(j=0; j<256; j++) {
        if ( (defaultpal[i].color.red == rampmap[j].color.red) &&
             (defaultpal[i].color.green == rampmap[j].color.green) &&
             (defaultpal[i].color.blue == rampmap[j].color.blue)) {

            rampmap[j].flags = EXACTMATCH;
            rampmap[j].defaultindex = i;
            defaultpal[i].rampindex = j;
            defaultpal[i].flags = EXACTMATCH;
            break;
        }
    }
}

/* Now find close matches */
for (i=0; i<20; i++) {
    if (defaultpal[i].flags == EXACTMATCH)
        continue;         /* skip entries w/ exact matches */
    min_error = MAX_PAL_ERROR;

    /* Loop through RGB ramp and calculate least square error */
    /* if an entry has already been used, skip it */
    for(j=0; j<256; j++) {
        if (rampmap[j].flags != 0)      /* Already used */
            continue;

        delta = defaultpal[i].color.red - rampmap[j].color.red;
        error = (delta * delta);
        delta = defaultpal[i].color.green - rampmap[j].color.green;
        error += (delta * delta);
        delta = defaultpal[i].color.blue - rampmap[j].color.blue;
        error += (delta * delta);
        if (error < min_error) {        /* New minimum? */
            error_index = j;
            min_error = error;
        }
    }
    defaultpal[i].rampindex = error_index;
    rampmap[error_index].flags = CHANGED;
    rampmap[error_index].defaultindex = i;
}

/* First print out the color cube */

printf("Standard 8-bit RGB color cube with gamma %.2f:\n", gamma);
for (i=0; i<256; i++) {
```

```
    pc = &rampmap[i].color;
    printf("%3ld: (%3-D, %3-D, %3-D)\n", i, pc->red,
            pc->green, pc->blue);

}
printf("\n");

/* Now print out the default entries that have an exact match */

for (i=0; i<20; i++) {
    if (defaultpal[i].flags == EXACTMATCH) {
        pc = &defaultpal[i].color;
        printf("Default entry %2ld exactly matched RGB ramp entry
                  %3ld", i, defaultpal[i].rampindex);
        printf(" (%3-D, %3-D, %3-D)\n", pc->red, pc->green, pc->blue);
    }
}
printf("\n");

/* Now print out the closest entries for rest of
 * the default entries */

for (i=0; i<20; i++) {
    if (defaultpal[i].flags != EXACTMATCH) {
        pc = &defaultpal[i].color;
        printf("Default entry %2ld (%3-D, %3-D, %3-D) is close to ",
            i, pc->red, pc->green, pc->blue);
        pc = &rampmap[defaultpal[i].rampindex].color;
        printf("RGB ramp entry %3ld (%3-D, %3-D, %3-D)\n",
            defaultpal[i].rampindex, pc->red, pc->green, pc->blue);
    }
}
printf("\n");

/* Print out code to initialize a logical palette
 * that will not overflow */

printf("Here is code you can use to create a logical palette\n");

printf("static struct {\n");
printf("    WORD        palVersion;\n");
printf("    WORD        palNumEntries;\n");
printf("    PALETTEENTRY palPalEntries[256];\n");
printf("} rgb8palette = {\n");
printf("    0x300,\n");
printf("    256,\n");

for (i=0; i<256; i++) {
    if (rampmap[i].flags == 0)
        pc = &rampmap[i].color;
    else
        pc = &defaultpal[rampmap[i].defaultindex].color;

    printf("    %3-D, %3-D, %3-D,   0,    /* %ld",
            pc->red, pc->green, pc->blue, i);
```

```
        if (rampmap[i].flags == EXACTMATCH)
            printf(" - Exact match with default %d",
                        rampmap[i].defaultindex);
        if (rampmap[i].flags == CHANGED)
            printf(" - Changed to match default %d",
                        rampmap[i].defaultindex);
        printf(" */\n");
    }

    printf("};\n");
    printf("\n    * * *\n\n");
    printf("    hpal = CreatePalette((LOGPALETTE *)&rgb8palette);\n");

    return 0;
}
```

Color-Index Mode and Windows Palette Management

The color-index mode specifies colors in a logical palette with an index to a specific logical-palette entry. Most GDI programs use color-index palettes, but the RGBA mode works better for OpenGL for several effects, such as shading, lighting, fog, and texture mapping. If having the truest color isn't critical for your OpenGL application, you might choose to use the color-index mode (for example, for a topographic map that uses "false color" to emphasize the elevation gradient).

Color-Index Mode Palette Sample

The following code sets up a **PIXELFORMATDESCRIPTOR** structure that sets the flag of the **iPixelType** member to PFD_TYPE_COLORINDEX. This specifies that the application use a color-index palette.

```
BOOL bSetupPixelFormat(HDC hdc)
{
    PIXELFORMATDESCRIPTOR pfd, *ppfd;
    int pixelformat;

    ppfd = &pfd;

    ppfd->nSize = sizeof(PIXELFORMATDESCRIPTOR);
    ppfd->nVersion = 1;
    ppfd->dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL |
                    PFD_DOUBLEBUFFER;
    ppfd->dwLayerMask = PFD_MAIN_PLANE;

    /* Set to color-index mode and use the default color palette. */
    ppfd->iPixelType = PFD_TYPE_COLORINDEX;

    ppfd->cColorBits = 8;
    ppfd->cDepthBits = 16;
    ppfd->cAccumBits = 0;
    ppfd->cStencilBits = 0;

    pixelformat = ChoosePixelFormat(hdc, ppfd);

    if ( (pixelformat = ChoosePixelFormat(hdc, ppfd)) == 0 )
    {
        MessageBox(NULL, "ChoosePixelFormat failed", "Error", MB_OK);
        return FALSE;
    }

    if (SetPixelFormat(hdc, pixelformat, ppfd) == FALSE)
    {
        MessageBox(NULL, "SetPixelFormat failed", "Error", MB_OK);
        return FALSE;
    }

    return TRUE;
}
```

Overlay, Underlay, and Main Planes

You can use hardware layer planes (overlay and underlay planes) in your applications. With Windows NT and Windows 95, pixel formats describe the pixel configurations of a graphics device. Each pixel format describes the depth and other characteristics of the main color buffers and describes additional buffers (such as depth, accumulation, stencil, and auxiliary) that the main plane uses. Pixel formats are now extended to include overlay and underlay buffers.

Layer planes always have a front-left color buffer and also can include front-right and back color buffers. Each layer plane has a specific rendering context to render into the layer buffers. You cannot use GDI drawing functions in layer planes.

A window manages the color buffers of layer planes similarly to the way it manages main-plane color buffers. You can display multiple windows with overlay and/or underlay planes at the same time. You cannot have free-floating overlay windows that can move over any window in the main drawing plane. In addition, because it would obscure underlying planes in a window at all times, you cannot use hardware pop-up planes that have no transparent color.

Each layer plane in a window has an associated palette. You can set the palette of a color-index layer plane, but the palette of an RGBA color plane is fixed. You must realize the appropriate palette when a window is in the foreground. Layer planes have a transparent pixel color or index that enables any underlying layer planes to show through.

You can copy the state of a rendering context to another rendering context in a separate layer plane. You can also share display lists among rendering contexts in different layer planes.

The following functions are used with layer planes:

wglCopyContext
wglCreateLayerContext
wglDescribeLayerPlane
wglGetLayerPaletteEntries
wglRealizeLayerPalette
wglSetLayerPaletteEntries
wglSwapLayerBuffers

Sharing Display Lists

When you create a rendering context, it has its own display-list space. The **wglShareLists** function enables a rendering context to share the display-list space of another rendering context. Any number of rendering contexts can share a single display-list space.

Extending OpenGL Functions

The OpenGL library supports multiple implementations of its functions. Extension functions supported in one rendering context aren't necessarily supported in a different rendering context. For a given rendering context in an application using extension functions, use only the function addresses returned by the **wglGetProcAddress** function.

GLX and WGL/Win32

Some of the WGL functions, new Win32 functions, and existing Win32 functions are more or less analogous to GLX X Window functions. The following list shows GLX functions and their corresponding WGL/Win32 functions, if available.

GLX Functions	WGL/Win32 Functions
glXChooseVisual	**ChoosePixelFormat**
glXCopyContext	–
glXCreateContext	**wglCreateContext**
glXCreateGLXPixmap	**CreateDIBitmap**/ **CreateDIBSection**
glXDestroyContext	**wglDeleteContext**
glXDestroyGLXPixmap	**DeleteObject**
glXGetConfig	**DescribePixelFormat**
glXGetCurrentContext	**wglGetCurrentContext**
glXGetCurrentDrawable	**wglGetCurrentDC**
glXIsDirect	–
glXMakeCurrent	**wglMakeCurrent**
glXQueryExtension	**GetVersion**
glXQueryVersion	**GetVersion**
glXSwapBuffers	**SwapBuffers**
glXUseXFont	**wglUseFontBitmaps**/ **wglUseFontOutlines**
glXWaitGL	–
glXWaitX	–
XGetVisualInfo	**GetPixelFormat**
XCreateWindow	**CreateWindow**/**CreateWindowEx** and **GetDC**/**BeginPaint**
XSync	**GdiFlush**
–	**SetPixelFormat**
–	**wglGetProcAddress**
–	**wglShareLists**

For more information, refer to the *Porting Guide*.

Using OpenGL on Windows NT and Windows 95

The following topics explain how to use several features specific to Windows NT and Windows 95 of the Microsoft implementation of OpenGL.

Header Files

Applications that use:

- The core OpenGL functions must include the header file <GL\GL.H>.
- The OpenGL Utility library must include the header file <GL\GLU.H>.
- The OpenGL Programming Guide auxiliary library must include the header file <GL\GLAUX.H>.
- The WGL functions must include the header file WINDOWS.H.
- The new Win32 functions that support Microsoft's implementation of OpenGL in Windows NT and Windows 95 must include the header file WINDOWS.H.

Pixel Format Tasks

You set a device-context pixel format prior to creating an OpenGL rendering context.

Choosing and Setting a Best-Match Pixel Format

This topic explains the procedure for matching a device context to a pixel format.

To obtain a device context's best match to a pixel format

1. Specify the desired pixel format in a **PIXELFORMATDESCRIPTOR** structure.
2. Call **ChoosePixelFormat**.

   The **ChoosePixelFormat** function returns a pixel format index, which you can then pass to **SetPixelFormat** to set the best pixel format match as the device context's current pixel format.

The following code sample shows how to carry out the above steps:

```
PIXELFORMATDESCRIPTOR pfd = {
    sizeof(PIXELFORMATDESCRIPTOR),    // size of this pfd
    1,                                // version number
    PFD_DRAW_TO_WINDOW |              // support window
    PFD_SUPPORT_OPENGL |              // support OpenGL
    PFD_DOUBLEBUFFER,                 // double buffered
    PFD_TYPE_RGBA,                    // RGBA type
    24,                               // 24-bit color depth
    0, 0, 0, 0, 0, 0,                 // color bits ignored
    0,                                // no alpha buffer
    0,                                // shift bit ignored
    0,                                // no accumulation buffer
    0, 0, 0, 0,                       // accum bits ignored
    32,                               // 32-bit z-buffer
    0,                                // no stencil buffer
    0,                                // no auxiliary buffer
    PFD_MAIN_PLANE,                   // main layer
    0,                                // reserved
    0, 0, 0                           // layer masks ignored
};
HDC  hdc;
int  iPixelFormat;

// get the device context's best, available pixel format match
iPixelFormat = ChoosePixelFormat(hdc, &pfd);

// make that match the device context's current pixel format
SetPixelFormat(hdc, iPixelFormat, &pfd);
```

Examining a Device Context's Current Pixel Format

Use the **GetPixelFormat** and **DescribePixelFormat** functions to examine a device context's current pixel format, as shown in the following code fragment:

```
PIXELFORMATDESCRIPTOR  pfd;
HDC    hdc;
int    iPixelFormat;

// if the device context has a current pixel format ...
if (iPixelFormat = GetPixelFormat(hdc)) {

    // obtain a detailed description of that pixel format
    DescribePixelFormat(hdc, iPixelFormat,
                            sizeof(PIXELFORMATDESCRIPTOR), &pfd);
    }
```

Examining a Device's Supported Pixel Formats

The **DescribePixelFormat** function obtains pixel format data for a device context. It also returns an integer that is the maximum pixel format index for the device context. The following code sample shows how to use that result to step through and examine the pixel formats supported by a device:

```
// local variables
int                     iMax ;
PIXELFORMATDESCRIPTOR   pfd;
int                     iPixelFormat ;

// initialize a pixel format index variable
iPixelFormat = 1;

// keep obtaining and examining pixel format data...
do {
    // try to obtain some pixel format data
    iMax = DescribePixelFormat(hdc, iPixelFormat, sizeof(pfd), &pfd);

    // if there was some problem with that...
    if (iMax == 0)

        // return indicating failure
        return(FALSE);

    // we have successfully obtained pixel format data

    // let's examine the pixel format data...
    myPixelFormatExaminer (&pfd);
    }

// ...until we've looked at all the device context's pixel formats
while (++iPixelFormat <= iMax);
```

Rendering Context Tasks

All calls pass through a rendering context. After you set a device context's pixel format, you can create a rendering context.

Creating a Rendering Context and Making It Current

The following code sample shows how to create an OpenGL rendering context in response to a WM_CREATE message. Notice that you set up the pixel format before creating the rendering context. Also notice that in this scenario the device context is not released locally; you release it when the window is closed, after making the rendering context not current. For more information, see Deleting a Rendering Context. Finally, notice that you can use local variables for the device context and rendering context handles, because with the **wglGetCurrentContext** and **wglGetCurrentDC** functions you can obtain handles to those contexts as needed.

```
// a window has been created, but is not yet visible
case WM_CREATE:
    {
    // local variables
    HDC      hdc ;
    HGLRC    hglrc ;

    // obtain a device context for the window
    hdc = GetDC(hWnd);

    // set an appropriate pixel format
    myPixelFormatSetupFunction(hdc);

    // if we can create a rendering context ...
    if (hglrc = wglCreateContext( hdc ) ) {

        // try to make it the thread's current rendering context
        bHaveCurrentRC = wglMakeCurrent(hdc, hglrc) ;

        }

    // perform miscellaneous other WM_CREATE chores ...

    }
    break ;
```

Making a Rendering Context Not Current

To detach a rendering context from a thread, make it not current. You can do this by calling the **wglMakeCurrent** function with the parameters set to NULL. The following is a sample of this simple task:

```
// detach the current rendering context from the thread
wglMakeCurrent(NULL, NULL);
```

Deleting a Rendering Context

The following code sample shows how to delete an OpenGL rendering context when an OpenGL window is closed. It is a continuation of the scenario used in Creating a Rendering Context and Making It Current.

```
// a window is about to be destroyed
case WM_DESTROY:
    {
    // local variables
    HGLRC    hglrc;
    HDC      hdc ;

    // if the thread has a current rendering context ...
    if(hglrc = wglGetCurrentContext()) {

        // obtain its associated device context
        hdc = wglGetCurrentDC() ;

        // make the rendering context not current
        wglMakeCurrent(NULL, NULL) ;

        // release the device context
        ReleaseDC (hwnd, hdc) ;

        // delete the rendering context
        wglDeleteContext(hglrc);

        }
```

Drawing with Double Buffers

Double buffers smooth the transition between one image and another on the screen. Swapping buffers typically comes at the end of a sequence of drawing commands. By default, the Microsoft implementation of OpenGL in Windows NT and Windows 95 draws to the off-screen buffer; when drawing is complete, you call the **SwapBuffers** function to copy the off-screen buffer to the on-screen buffer. The following code sample prepares to draw, calls a drawing function, and then copies the completed image on to the screen if double buffering is available.

```
void myRedraw(void)
{
    // set up for drawing commands
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45, 1.0, 0.1, 100.0);

    // draw our objects
    myDrawAllObjects(GL_FALSE);

    // if we're double-buffering ...
    if (bDoubleBuffering)

        // ...draw the copied image to the screen
        SwapBuffers(hdc);
}
```

The following code sample obtains a window device context, renders a scene, copies the image to the screen (to show the rendering), and then releases the device context.

```
hdc = GetDC(hwnd);
mySceneRenderingFunction();
SwapBuffers(hdc);
ReleaseDC(hWnd, hdc);
```

Drawing Text in a Double-Buffered OpenGL Window

You draw text in a double-buffered OpenGL window by creating display lists for selected characters in a font, and then executing the appropriate display list for each character you want to draw. The following code sample creates a rendering context, draws a red triangle, and then labels it with text. For this sample code, we assume that there is a device context, with a font and pixel format.

```
// create an OpenGL rendering context
hglrc = wglCreateContext(hdc);

// make it this thread's current rendering context
wglMakeCurrent(hdc, hglrc);

// make the color a deep blue hue
glClearColor(0.0F, 0.0F, 0.4F, 1.0F);

// make the shading smooth
glShadeModel(GL_SMOOTH);

// clear the color buffers
glClear(GL_COLOR_BUFFER_BIT);

// specify a red triangle
glBegin(GL_TRIANGLES);
    glColor3f(1.0F, 0.0F, 0.0F);
    glVertex2f(10.0F, 10.0F);
    glVertex2f(250.0F, 50.0F);
    glVertex2f(105.0F, 280.0F);
glEnd();

// create bitmaps for the device context font's first 256 glyphs
wglUseFontBitmaps(hdc, 0, 256, 1000);

// move bottom left, southwest of the red triangle
glRasterPos2f(30.0F, 300.0F);

// set up for a string-drawing display list call
glListBase(1000);

// draw a string using font display lists
glCallLists(12, GL_UNSIGNED_BYTE, "Red Triangle");

// get all those commands to execute
glFlush();

// delete our 256 glyph display lists
glDeleteLists(1000, 256) ;

// make the rendering context not current
wglMakeCurrent (NULL, NULL) ;

// release the device context
ReleaseDC(hdc) ;
```

```
// delete the rendering context
wglDeleteContext(hglrc);
```

Printing an OpenGL Image

You can print OpenGL images rendered in enhanced metafiles. When you render to a printer device (HDC) it must be backed by a metafile spooler. OpenGL uses memory for depth, color, and other buffers to store graphics output to a printer. Because typical printer resolution requires a significant amount of memory to store the graphics output, printing an OpenGL image might require more memory than is available in the system. To overcome this limitation, the current implementation of OpenGL prints OpenGL graphics in bands. However, this increases the time it takes to print OpenGL images.

Before you print an OpenGL image, you must call the **StartDoc** function to complete the initialization of a printer device context (DC). After calling **StartDoc**, you can create rendering contexts (HGLRC) to render to the printer device. If you create rendering contexts before calling **StartDoc**, there is no way to determine whether a metafile is spooled.

The following code sample shows how to print an OpenGL image:

```
HDC             hDC;
DOCINFO         di;
HGLRC           hglrc;

// Call StartDoc to start the document
StartDoc( hDC, &di );

// Prepare the printer driver to accept data
StartPage(hDC);

// Create a rendering context using the metafile DC
hglrc = wglCreateContext ( hDCmetafile );

// Do OpenGL rendering operations here
    . . .
wglMakeCurrent(NULL, NULL); // Get rid of rendering context
    . . .
EndPage(hDC); // Finish writing to the page
EndDoc(hDC); // End the print job
```

For more information on using metafiles, see Enhanced Metafile Operations.

Copying an OpenGL Image to the Clipboard

Although the current version of the Microsoft implementation of OpenGL in Windows NT and Windows 95 does not directly support the Clipboard, you can copy a Windows NT or Windows 95 OpenGL image to the Clipboard.

To copy an OpenGL image to the Clipboard

1. Draw the image to a memory bitmap or an enhanced metafile.
2. Copy that bitmap to the Clipboard.

Multithread OpenGL Drawing Strategies

The GDI does not support multiple threads. You must use a distinct device context and a distinct rendering context for each thread. This tends to limit the performance advantages of using multiple threads with single-processor systems running OpenGL applications. However, there are ways to use threads with a single processor system to greatly increase performance. For example, you can use a separate thread to pass OpenGL rendering calls to dedicated 3-D hardware.

Symmetric multiprocessing (SMP) systems can greatly benefit from using multiple threads. An obvious strategy is to use a separate thread for each processor to handle OpenGL rendering in separate windows. For example, in a flight-simulation application you could use separate processors and threads to render the front, back, and side views.

A thread can have only one current, active rendering context. When you use multiple threads and multiple rendering contexts, you must be careful to synchronize their use. For example, use one thread only to call **SwapBuffers** after all threads finish drawing.

Using the Auxiliary Library

Using OpenGL, Silicon Graphics (SGI) created the Auxiliary Library to write simple sample programs for the *OpenGL Programming Guide*. The source code for the Auxiliary Library is supplied with the Win32 SDK, along with the OpenGL samples. To understand how the Auxiliary Library was developed from OpenGL functions and routines, examine the source code. You can use Auxiliary Library functions in your own programs. For a description of the Auxiliary Library, see the *OpenGL Programming Guide.*

Reference for Win 32 Extensions to OpenGL

The following sections contain listings of the functions and structures associated with WGL and Win32.

WGL Functions

Win32 Functions

ChoosePixelFormat
DescribePixelFormat
GetEnhMetaFilePixelFormat
GetPixelFormat
SetPixelFormat
SwapBuffers

Structures

GLYPHMETRICSFLOAT
LAYERPLANEDESCRIPTOR
PIXELFORMATDESCRIPTOR
POINTFLOAT

ChoosePixelFormat Quick Info

Overview

Group

[New - Windows 95, OEM Service Release 2]

The **ChoosePixelFormat** function attempts to match an appropriate pixel format supported by a device context to a given pixel format specification.

```
int ChoosePixelFormat(
  HDC   hdc,                                // device context to search for a best pixel format match
  CONST PIXELFORMATDESCRIPTOR *   ppfd      // pixel format for which a best match is sought
);
```

Parameters

hdc

Specifies the device context that the function examines to determine the best match for the pixel format descriptor pointed to by *ppfd*.

ppfd

Pointer to a **PIXELFORMATDESCRIPTOR** structure that specifies the requested pixel format. In this context, the members of the **PIXELFORMATDESCRIPTOR** structure that *ppfd* points to are used as follows:

nSize

Specifies the size of the **PIXELFORMATDESCRIPTOR** data structure. Set this member to **sizeof(PIXELFORMATDESCRIPTOR)**.

nVersion

Specifies the version number of the **PIXELFORMATDESCRIPTOR** data structure. Set this member to 1.

dwFlags

A set of bit flags that specify properties of the pixel buffer. You can combine the following bit flag constants by using bitwise-OR.

If any of the following flags are set, the **ChoosePixelFormat** function attempts to match pixel formats that also have that flag or flags set. Otherwise, **ChoosePixelFormat** ignores that flag in the pixel formats:

PFD_DRAW_TO_WINDOW
PFD_DRAW_TO_BITMAP
PFD_SUPPORT_GDI
PFD_SUPPORT_OPENGL

If any of the following flags are set, **ChoosePixelFormat** attempts to match pixel formats that also have that flag or flags set. Otherwise, it attempts to match pixel formats without that flag set:

PFD_DOUBLEBUFFER
PFD_STEREO

If the following flag is set, the function ignores the PFD_DOUBLEBUFFER flag in the pixel formats:

PFD_DOUBLEBUFFER_DONTCARE

If the following flag is set, the function ignores the PFD_STEREO flag in the pixel formats:

PFD_STEREO_DONTCARE

iPixelType

Specifies the type of pixel format for the function to consider:

PFD_TYPE_RGBA
PFD_TYPE_COLORINDEX

cColorBits

Zero or greater.

cRedBits
Not used.

cRedShift
Not used.

cGreenBits
Not used.

cGreenShift
Not used.

cBlueBits
Not used.

cBlueShift
Not used.

cAlphaBits
Zero or greater.

cAlphaShift
Not used.

cAccumBits
Zero or greater.

cAccumRedBits
Not used.

cAccumGreenBits
Not used.

cAccumBlueBits
Not used.

cAccumAlphaBits
Not used.

cDepthBits
Zero or greater.

cStencilBits
Zero or greater.

cAuxBuffers
Zero or greater.

iLayerType
Specifies one of the following layer type values:
PFD_MAIN_PLANE
PFD_OVERLAY_PLANE
PFD_UNDERLAY_PLANE

bReserved
Not used.

dwLayerMask
Not used.

dwVisibleMask
Not used.

dwDamageMask
Not used.

Return Values

If the function succeeds, the return value is a pixel format index (one-based) that is the closest match to the given pixel format descriptor.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

You must ensure that the pixel format matched by the **ChoosePixelFormat** function satisfies your requirements. For example, if you request a pixel format with a 24-bit RGB color buffer but the device context offers only 8-bit RGB color buffers, the function returns a pixel format with an 8-bit RGB color buffer.

The following code sample shows how to use **ChoosePixelFormat** to match a specified pixel format:

```
PIXELFORMATDESCRIPTOR pfd = {
    sizeof(PIXELFORMATDESCRIPTOR),  //  size of this pfd
    1,                     // version number
    PFD_DRAW_TO_WINDOW |   // support window
    PFD_SUPPORT_OPENGL |   // support OpenGL
    PFD_DOUBLEBUFFER,      // double buffered
    PFD_TYPE_RGBA,         // RGBA type
    24,                    // 24-bit color depth
    0, 0, 0, 0, 0, 0,      // color bits ignored
    0,                     // no alpha buffer
    0,                     // shift bit ignored
    0,                     // no accumulation buffer
    0, 0, 0, 0,            // accum bits ignored
    32,                    // 32-bit z-buffer
    0,                     // no stencil buffer
    0,                     // no auxiliary buffer
    PFD_MAIN_PLANE,        // main layer
    0,                     // reserved
    0, 0, 0                // layer masks ignored
    };
    HDC  hdc;
    int  iPixelFormat;

iPixelFormat = ChoosePixelFormat(hdc, &pfd);
```

See Also

DescribePixelFormat, **GetPixelFormat**, **SetPixelFormat**

DescribePixelFormat Quick Info

Overview

Group

[New - Windows 95, OEM Service Release 2]

The **DescribePixelFormat** function obtains information about the pixel format identified by *iPixelFormat* of the device associated with *hdc*. The function sets the members of the **PIXELFORMATDESCRIPTOR** structure pointed to by *ppfd* with that pixel format data.

```
int DescribePixelFormat(
  HDC   hdc,                          // device context of interest
  int   iPixelFormat,                 // pixel format selector
  UINT   nBytes,                      // size of buffer pointed to by ppfd
  LPPIXELFORMATDESCRIPTOR   ppfd      // pointer to structure to receive pixel format data
);
```

Parameters

hdc

Specifies the device context.

iPixelFormat

Index that specifies the pixel format. The pixel formats that a device context supports are identified by positive one-based integer indexes.

nBytes

The size, in bytes, of the structure pointed to by *ppfd*. The **DescribePixelFormat** function stores no more than *nBytes* bytes of data to that structure. Set this value to **sizeof(PIXELFORMATDESCRIPTOR)**.

ppfd

Pointer to a **PIXELFORMATDESCRIPTOR** structure whose members the function sets with pixel format data. The function stores the number of bytes copied to the structure in the structure's **nSize** member. If, upon entry, *ppfd* is NULL, the function writes no data to the structure. This is useful when you only want to obtain the maximum pixel format index of a device context.

Return Values

If the function succeeds, the return value is the maximum pixel format index of the device context. In addition, the function sets the members of the **PIXELFORMATDESCRIPTOR** structure pointed to by *ppfd* according to the specified pixel format.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The following code sample shows **DescribePixelFormat** usage:

```
PIXELFORMATDESCRIPTOR  pfd;
HDC  hdc;
int  iPixelFormat;

iPixelFormat = 1;

// obtain detailed information about
// the device context's first pixel format
DescribePixelFormat(hdc, iPixelFormat,
        sizeof(PIXELFORMATDESCRIPTOR), &pfd);
```

See Also

ChoosePixelFormat, GetPixelFormat, SetPixelFormat

GetEnhMetaFilePixelFormat Overview

Group

[New - Windows 95, OEM Service Release 2]

The **GetEnhMetaFilePixelFormat** function retrieves pixel format information for an enhanced metafile.

```
UNIT GetEnhMetaFilePixelFormat(
  HENHMETAFILE   hemf,                          // handle to an enhanced metafile
  DWORD cbBuffer,                               // buffer size
  CONST PIXELFORMATDESCRIPTOR *   ppfd          // pointer to logical pixel format specification
);
```

Parameters

hemf

Identifies the enhanced metafile.

cbBuffer

Specifies the size, in bytes, of the buffer into which the pixel format information is copied.

ppfd

Pointer to a **PIXELFORMATDESCRIPTOR** structure that contains the logical pixel format specification. The metafile uses this structure to record the logical pixel format specification.

Return Values

If the function succeeds and finds a pixel format, the return value is the size of the metafile's pixel format.

If no pixel format is present, the return value is zero.

If an error occurs and the function fails, the return value is GDI_ERROR. To get extended error information, call **GetLastError**.

Remarks

When an enhanced metafile specifies a pixel format in its **ENHMETAHEADER** structure and the pixel format fits in the buffer, the pixel format information is copied into *ppfd*. When *cbBuffer* is too small to contain the pixel format of the metafile, the pixel format is not copied to the buffer. In either case, the function returns the size of the metafile's pixel format.

For information on metafile recording and other operations, see Enhanced Metafile Operations.

See Also

ENHMETAHEADER, **PIXELFORMATDESCRIPTOR**

GetPixelFormat Quick Info

Overview

Group

[New - Windows 95, OEM Service Release 2]

The **GetPixelFormat** function obtains the index of the currently selected pixel format of the specified device context.

```
int GetPixelFormat(
  HDC   hdc     // device context whose currently selected pixel format index is sought
);
```

Parameters

hdc

Specifies the device context of the currently selected pixel format index returned by the function.

Return Values

If the function succeeds, the return value is the currently selected pixel format index of the specified device context. This is a positive, one-based index value.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The following code sample shows **GetPixelFormat** usage:

```
PIXELFORMATDESCRIPTOR  pfd;
HDC  hdc;
int  iPixelFormat;

// get the current pixel format index
iPixelFormat = GetPixelFormat(hdc);

// obtain a detailed description of that pixel format
DescribePixelFormat(hdc, iPixelFormat,
        sizeof(PIXELFORMATDESCRIPTOR), &pfd);
```

See Also

ChoosePixelFormat, **DescribePixelFormat**, **SetPixelFormat**

SetPixelFormat Quick Info

Overview

Group

[New - Windows 95, OEM Service Release 2]

The **SetPixelFormat** function sets the pixel format of the specified device context to the format specified by the *iPixelFormat* index.

```
BOOL SetPixelFormat(
  HDC   hdc,                                // device context whose pixel format the function attempts to set
  int   iPixelFormat,                       // pixel format index (one-based)
  CONST PIXELFORMATDESCRIPTOR *   ppfd      // pointer to logical pixel format specification
);
```

Parameters

hdc

Specifies the device context whose pixel format the function attempts to set.

iPixelFormat

Index that identifies the pixel format to set. The various pixel formats supported by a device context are identified by one-based indexes.

ppfd

Pointer to a **PIXELFORMATDESCRIPTOR** structure that contains the logical pixel format specification. The system's metafile component uses this structure to record the logical pixel format specification. The structure has no other effect upon the behavior of the **SetPixelFormat** function.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

If *hdc* references a window, calling the **SetPixelFormat** function also changes the pixel format of the window. Setting the pixel format of a window more than once can lead to significant complications for the Window Manager and for multithread applications, so it is not allowed. An application can only set the pixel format of a window one time. Once a window's pixel format is set, it cannot be changed.

You should select a pixel format in the device context before calling the **wglCreateContext** function. The **wglCreateContext** function creates a rendering context for drawing on the device in the selected pixel format of the device context.

An OpenGL window has its own pixel format. Because of this, only device contexts retrieved for the client area of an OpenGL window are allowed to draw into the window. As a result, an OpenGL window should be created with the WS_CLIPCHILDREN and WS_CLIPSIBLINGS styles. Additionally, the window class attribute should not include the CS_PARENTDC style.

The following code example shows **SetPixelFormat** usage:

```
PIXELFORMATDESCRIPTOR pfd = {
    sizeof(PIXELFORMATDESCRIPTOR),   // size of this pfd
    1,                     // version number
    PFD_DRAW_TO_WINDOW |   // support window
    PFD_SUPPORT_OPENGL |   // support OpenGL
    PFD_DOUBLEBUFFER,      // double buffered
```

```
    PFD_TYPE_RGBA,          // RGBA type
    24,                     // 24-bit color depth
    0, 0, 0, 0, 0, 0,       // color bits ignored
    0,                      // no alpha buffer
    0,                      // shift bit ignored
    0,                      // no accumulation buffer
    0, 0, 0, 0,             // accum bits ignored
    32,                     // 32-bit z-buffer
    0,                      // no stencil buffer
    0,                      // no auxiliary buffer
    PFD_MAIN_PLANE,         // main layer
    0,                      // reserved
    0, 0, 0                 // layer masks ignored
};
HDC  hdc;
int  iPixelFormat;

// get the best available match of pixel format for the device context
iPixelFormat = ChoosePixelFormat(hdc, &pfd);

// make that the pixel format of the device context
SetPixelFormat(hdc, iPixelFormat, &pfd);
```

See Also

ChoosePixelFormat, DescribePixelFormat, GetPixelFormat

SwapBuffers Quick Info

Overview

Group

[New - Windows 95, OEM Service Release 2]

The **SwapBuffers** function exchanges the front and back buffers if the current pixel format for the window referenced by the specified device context includes a back buffer.

```
BOOL SwapBuffers(
  HDC   hdc     // device context whose buffers get swapped
);
```

Parameters

hdc

Specifies a device context. If the current pixel format for the window referenced by this device context includes a back buffer, the function exchanges the front and back buffers.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

If the current pixel format for the window referenced by the device context does not include a back buffer, this call has no effect and the content of the back buffer is undefined when the function returns.

With multithread applications, flush the drawing commands in any other threads drawing to the same window before calling **SwapBuffers**.

wglCreateContext Quick Info

Overview

Group

[New - Windows 95, OEM Service Release 2]

The **wglCreateContext** function creates a new OpenGL rendering context, which is suitable for drawing on the device referenced by *hdc*. The rendering context has the same pixel format as the device context.

```
HGLRC wglCreateContext(
  HDC   hdc      // device context of device that the rendering context will be suitable for
);
```

Parameters

hdc

Handle to a device context for which the function creates a suitable OpenGL rendering context.

Return Values

If the function succeeds, the return value is a valid handle to an OpenGL rendering context.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

A rendering context is not the same as a device context. Set the pixel format of the device context before creating a rendering context. For more information on setting the device context's pixel format, see the **SetPixelFormat** function.

To use OpenGL, you create a rendering context, select it as a thread's current rendering context, and then call OpenGL functions. When you are finished with the rendering context, you dispose of it by calling the **wglDeleteContext** function.

The following code example shows **wglCreateContext** usage:

```
HDC    hdc;
HGLRC  hglrc;

// create a rendering context
hglrc = wglCreateContext (hdc);

// make it the calling thread's current rendering context
wglMakeCurrent (hdc, hglrc);

// call OpenGL APIs as desired ...

// when the rendering context is no longer needed ...

// make the rendering context not current
wglMakeCurrent (NULL, NULL) ;

// delete the rendering context
wglDeleteContext (hglrc);
```

See Also

SetPixelFormat, **wglDeleteContext**, **wglGetCurrentContext**, **wglGetCurrentDC**, **wglMakeCurrent**

wglCreateLayerContext Quick Info

Overview

Group

[New - Windows 95, OEM Service Release 2]

The **wglCreateLayerContext** function creates a new OpenGL rendering context for drawing to a specified layer plane on a device context.

```
HGLRC wglCreateLayerContext(
  HDC   hdc,            // device context used for a rendering context
  int   iLayerPlane     // specifies the layer plane that a rendering context is bound to
);
```

Parameters

hdc

Specifies the device context for a new rendering context.

iLayerPlane

Specifies the layer plane to which you want to bind a rendering context. The value 0 identifies the main plane. Positive values of *iLayerPlane* identify overlay planes, where 1 is the first overlay plane over the main plane, 2 is the second overlay plane over the first overlay plane, and so on. Negative values identify underlay planes, where -1 is the first underlay plane under the main plane, -2 is the second underlay plane under the first underlay plane, and so on. The number of overlay and underlay planes is given in the **bReserved** member of the **PIXELFORMATDESCRIPTOR** structure.

Return Values

If the function succeeds, the return value is a handle to an OpenGL rendering context.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

A rendering context is a port through which all OpenGL commands pass. Every thread that makes OpenGL calls must have one current, active rendering context. A rendering context is not the same as a device context; a rendering context contains information specific to OpenGL, while a device context contains information specific to GDI.

Before you create a rendering context, set the pixel format of the device context with the **SetPixelFormat** function. You can use a rendering context in a specified layer plane of a window with identical pixel formats only.

With OpenGL applications that use multiple threads, you create a rendering context, select it as the current rendering context of a thread, and make OpenGL calls for the specified thread. When you are finished with the rendering context of the thread, call the **wglDeleteContext** function. The following code example shows how to use **wglCreateLayerContext**.

```
// The following code fragment shows how to render to overlay 1
// This example assumes that the pixel format of hdc includes
// overlay plane 1

HDC hdc;
HGLRC;

// create a rendering context for overlay plane 1
hglrc = wglCreateLayerContext(hdc, 1);
```

```
// make it the calling thread's current rendering context
wglMakeCurrent(hdc, hglrc);

// call OpenGL functions here. . .

// when the rendering context is no longer needed. . .

// make the rendering context not current
wglMakeCurrent(NULL, NULL);

// delete the rendering context
wglDeleteContext(hglrc);
```

See Also

PIXELFORMATDESCRIPTOR, **SetPixelFormat**, **wglCreateContext**, **wglDeleteContext**, **wglGetCurrentContext**, **wglGetCurrentDC**, **wglMakeCurrent**

wglCopyContext Quick Info

Overview

Group

[New - Windows 95, OEM Service Release 2]

The **wglCopyContext** function copies selected groups of rendering states from one OpenGL rendering context to another.

```
BOOL wglCopyContext(
  HGLRC   hglrcSrc,     // specifies the source of a rendering context
  HGLRC   hlglrcDst,    // specifies the destination of a rendering context
  UINT    mask          // specifies what rendering state information is copied
);
```

Parameters

hglrcSrc

Specifies the source OpenGL rendering context whose state information is to be copied.

hglrcDst

Specifies the destination OpenGL rendering context to which state information is to be copied.

mask

Specifies which groups of the *hglrcSrc* rendering state are to be copied to *hglrcDst*. It contains the bitwise-OR of the same symbolic names that are passed to the **glPushAttrib** function. You can use GL_ALL_ATTRIB_BITS to copy all the rendering state information.

Return Values

If the function succeeds, the return value is TRUE. If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

Using the **wglCopyContext** function, you can synchronize the rendering state of two rendering contexts. You can only copy the rendering state between two rendering contexts within the same process. The rendering contexts must be from the same OpenGL implementation. For example, you can always copy a rendering state between two rendering contexts with identical pixel format in the same process.

You can copy the same state information available only with the **glPushAttrib** function. You cannot copy some state information, such as pixel pack/unpack state, render mode state, select state, and feedback state. When you call **wglCopyContext**, make sure that the destination rendering context, *hglrcDst*, is not current to any thread.

See Also

glPushAttrib, **wglCreateLayerContext**, **wglCreateContext**, **wglShareLists**

wglDeleteContext Quick Info

Overview

Group

[New - Windows 95, OEM Service Release 2]

The **wglDeleteContext** function deletes a specified OpenGL rendering context.

```
BOOL wglDeleteContext(
  HGLRC   hglrc      // handle to the OpenGL rendering context to delete
);
```

Parameters

hglrc

Handle to an OpenGL rendering context that the function will delete.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

It is an error to delete an OpenGL rendering context that is the current context of another thread. However, if a rendering context is the calling thread's current context, the **wglDeleteContext** function changes the rendering context to being not current before deleting it.

The **wglDeleteContext** function does not delete the device context associated with the OpenGL rendering context when you call the **wglMakeCurrent** function. After calling **wglDeleteContext**, you must call **DeleteDC** to delete the associated device context.

See Also

DeleteDC, **wglCreateContext**, **wglGetCurrentContext**, **wglGetCurrentDC**, **wglMakeCurrent**

wglDescribeLayerPlane Quick Info

Overview

Group

[New - Windows 95, OEM Service Release 2]

The **wglDescribeLayerPlane** function obtains information about the layer planes of a given pixel format.

```
BOOL wglDescribeLayerPlane(
  HDC   hdc,                        // device context whose layer planes are of interest
  int   iPixelFormat,               // pixel format of the desired layer plane
  int   iLayerPlane,                // specifies an overlay or underlay plane
  UINT   nBytes,                    // specifies the size, in bytes, of a LAYERPLANEDESCRIPTOR structure
  LPLAYERPLANEDESCRIPTOR plpd       // points to a LAYERPLANEDESCRIPTOR structure
);
```

Parameters

hdc

Specifies the device context of a window whose layer planes are to be described.

iPixelFormat

Specifies which layer planes of a pixel format are being described.

iLayerPlane

Specifies the overlay or underlay plane. Positive values of *iLayerPlane* identify overlay planes, where 1 is the first overlay plane over the main plane, 2 is the second overlay plane over the first overlay plane, and so on. Negative values identify underlay planes, where -1 is the first underlay plane under the main plane, -2 is the second underlay plane under the first underlay plane, and so on. The number of overlay and underlay planes is given in the **bReserved** member of the **PIXELFORMATDESCRIPTOR** structure.

nBytes

Specifies the size, in bytes, of the structure pointed to by *plpd*. The **wglDescribeLayerPlane** function stores layer plane data in a **LAYERPLANEDESCRIPTOR** structure, and stores no more than *nBytes* of data. Set the value of *nBytes* to the size of **LAYERPLANEDESCRIPTOR**.

plpd

Points to a **LAYERPLANEDESCRIPTOR** structure. The **wglDescribeLayerPlane** function sets the value of the structure's data members. The function stores the number of bytes of data copied to the structure in the **nSize** member.

Return Values

If the function succeeds, the return value is TRUE. In addition, the **wglDescribeLayerPlane** function sets the members of the **LAYERPLANEDESCRIPTOR** structure pointed to by *plpd* according to the specified layer plane (*iLayerPlane*) of the specified pixel format (*iPixelFormat*).

If the function fails, the return value is FALSE.

Remarks

The numbering of planes (*iLayerPlane*) determines their order. Higher-numbered planes overlay lower-numbered planes.

See Also

DescribePixelFormat, **LAYERPLANEDESCRIPTOR**, **PIXELFORMATDESCRIPTOR**, **wglCreateLayerContext**

wglGetCurrentContext Quick Info

Overview

Group

[New - Windows 95, OEM Service Release 2]

The **wglGetCurrentContext** function obtains a handle to the current OpenGL rendering context of the calling thread.

HGLRC wglGetCurrentContext(VOID);

Parameters

This function has no parameters.

Return Values

If the calling thread has a current OpenGL rendering context, **wglGetCurrentContext** returns a handle to that rendering context. Otherwise, the return value is NULL.

Remarks

The current OpenGL rendering context of a thread is associated with a device context by means of the **wglMakeCurrent** function. You can use the **wglGetCurrentDC** function to obtain a handle to the device context associated with the current OpenGL rendering context.

See Also

wglCreateContext, **wglDeleteContext**, **wglGetCurrentDC**, **wglMakeCurrent**

wglGetCurrentDC Quick Info

Overview

Group

[New - Windows 95, OEM Service Release 2]

The **wglGetCurrentDC** function obtains a handle to the device context that is associated with the current OpenGL rendering context of the calling thread.

HDC wglGetCurrentDC(VOID);

Parameters

This function has no parameters.

Return Values

If the calling thread has a current OpenGL rendering context, the function returns a handle to the device context associated with that rendering context by means of the **wglMakeCurrent** function. Otherwise, the return value is NULL.

Remarks

You associate a device context with an OpenGL rendering context when it calls the **wglMakeCurrent** function. You can use the **wglGetCurrentContext** function to obtain a handle to the calling thread's current OpenGL rendering context.

See Also

wglCreateContext, **wglDeleteContext**, **wglGetCurrentContext**, **wglMakeCurrent**

wglGetLayerPaletteEntries Quick Info

Overview

Group

[New - Windows 95, OEM Service Release 2]

The **wglGetLayerPaletteEntries** function retrieves the palette entries from a given color-index layer plane for a specified device context.

```
int wglGetLayerPaletteEntries(
  HDC   hdc,                  // device context of a window whose layer planes are to be described
  int   iLayerPlane,          // specifies an overlay or underlay plane
  int   iStart,               // specifies the first palette entry to be set
  int   cEntries,             // specifies the number of palette entries to be set
  CONST COLORREF *pcr         // points to the first member of an array of COLORREF structures
);
```

Parameters

hdc

Specifies the device context of a window whose layer planes are to be described.

iLayerPlane

Specifies the overlay or underlay plane. Positive values of *iLayerPlane* identify overlay planes, where 1 is the first overlay plane over the main plane, 2 is the second overlay plane over the first overlay plane, and so on. Negative values identify underlay planes, where -1 is the first underlay plane under the main plane, -2 is the second underlay plane under the first underlay plane, and so on. The number of overlay and underlay planes is given in the **bReserved** member of the **PIXELFORMATDESCRIPTOR** structure.

iStart

Specifies the first palette entry to be retrieved.

cEntries

Specifies the number of palette entries to be retrieved.

pcr

Points to an array of **COLORREF** structures that contain palette RGB color values. The array must contain at least as many structures as specified by *cEntries*.

Return Values

If the function succeeds, the return value is the number of entries that were set in the palette in the specified layer plane of the window.

If the function fails or when no pixel format is selected, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

Each color-index layer plane in a window has a palette with a size 2^*n*, where *n* is the number of bit planes in the layer plane. You cannot modify the transparent index of a palette.

Use the **wglRealizeLayerPalette** function to realize the layer palette. Initially the layer palette contains only entries for white.

The **wglSetPaletteEntries** function doesn't set the palette entries of the main plane palette. To update the main plane palette, use GDI palette functions.

See Also

COLORREF, **LAYERPLANEDESCRIPTOR**, **PIXELFORMATDESCRIPTOR**, **wglDescribeLayerPlane**, **wglRealizeLayerPalette**, **wglSetLayerPaletteEntries**

wglGetProcAddress Quick Info

Overview

Group

[New - Windows 95, OEM Service Release 2]

The **wglGetProcAddress** function returns the address of an OpenGL extension function for use with the current OpenGL rendering context.

```
PROC wglGetProcAddress(
  LPCSTR   lpszProc      // name of the extension function
);
```

Parameters

lpszProc

Points to a null-terminated string that is the name of the extension function. The name of the extension function must be identical to a corresponding function implemented by OpenGL.

Return Values

When the function succeeds, the return value is the address of the extension function.

When no current rendering context exists or the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

The OpenGL library supports multiple implementations of its functions. Extension functions supported in one rendering context are not necessarily available in a separate rendering context. Thus, for a given rendering context in an application, use the function addresses returned by the **wglGetProcAddress** function only.

The spelling and the case of the extension function pointed to by *lpszProc* must be identical to that of a function supported and implemented by OpenGL. Because extension functions are not exported by OpenGL, you must use **wglGetProcAddress** to get the addresses of vendor-specific extension functions.

The extension function addresses are unique for each pixel format. All rendering contexts of a given pixel format share the same extension function addresses.

See Also

glGetString, **wglMakeCurrent**

wglMakeCurrent Quick Info

Overview

Group

[New - Windows 95, OEM Service Release 2]

The **wglMakeCurrent** function makes a specified OpenGL rendering context the calling thread's current rendering context. All subsequent OpenGL calls made by the thread are drawn on the device identified by *hdc*. You can also use **wglMakeCurrent** to change the calling thread's current rendering context so it's no longer current.

```
BOOL wglMakeCurrent(
  HDC   hdc,        // device context of device that OpenGL calls are to be drawn on
  HGLRC   hglrc     // OpenGL rendering context to be made the calling thread's current
                    rendering context
);
```

Parameters

hdc

Handle to a device context. Subsequent OpenGL calls made by the calling thread are drawn on the device identified by *hdc*.

hglrc

Handle to an OpenGL rendering context that the function sets as the calling thread's rendering context.

If *hglrc* is NULL, the function makes the calling thread's current rendering context no longer current, and releases the device context that is used by the rendering context. In this case, *hdc* is ignored.

Return Values

When the **wglMakeCurrent** function succeeds, the return value is TRUE; otherwise the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

The *hdc* parameter must refer to a drawing surface supported by OpenGL. It need not be the same *hdc* that was passed to **wglCreateContext** when *hglrc* was created, but it must be on the same device and have the same pixel format. GDI transformation and clipping in *hdc* are not supported by the rendering context. The current rendering context uses the *hdc* device context until the rendering context is no longer current.

Before switching to the new rendering context, OpenGL flushes any previous rendering context that was current to the calling thread.

A thread can have one current rendering context. A process can have multiple rendering contexts by means of multithreading. A thread must set a current rendering context before calling any OpenGL functions. Otherwise, all OpenGL calls are ignored.

A rendering context can be current to only one thread at a time. You cannot make a rendering context current to multiple threads.

An application can perform multithread drawing by making different rendering contexts current to different threads, supplying each thread with its own rendering context and device context.

If an error occurs, the **wglMakeCurrent** function makes the thread's current rendering context not current before returning.

See Also

wglCreateContext, wglDeleteContext, wglGetCurrentContext, wglGetCurrentDC

wglRealizeLayerPalette Quick Info

Overview

Group

[New - Windows 95, OEM Service Release 2]

The **wglRealizeLayerPalette** function maps palette entries from a given color-index layer plane into the physical palette or initializes the palette of an RGBA layer plane.

```
BOOL wglRealizeLayerPalette(
  HDC   hdc,           // device context whose layer plane palette is to be realized
  int   iLayerPlane,   // specifies an overlay or underlay plane
  BOOL bRealize        // indicates whether the palette is to be realized into the physical palette
);
```

Parameters

hdc

Specifies the device context of a window whose layer plane palette is to be realized into the physical palette.

iLayerPlane

Specifies the overlay or underlay plane. Positive values of *iLayerPlane* identify overlay planes, where 1 is the first overlay plane over the main plane, 2 is the second overlay plane over the first overlay plane, and so on. Negative values identify underlay planes, where -1 is the first underlay plane under the main plane, -2 is the second underlay plane under the first underlay plane, and so on. The number of overlay and underlay planes is given in the **bReserved** member of the **PIXELFORMATDESCRIPTOR** structure.

bRealize

Indicates whether the palette is to be realized into the physical palette. When *bRealize* is TRUE, the palette entries are mapped into the physical palette where available. When *bRealize* is FALSE, the palette entries for the layer plane of the window are no longer needed and might be released for use by another foreground window.

Return Values

If the function succeeds, the return value is TRUE, even if *bRealize* is TRUE and the physical palette is not available. If the function fails or when no pixel format is selected, the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

The physical palette for a layer plane is a shared resource among windows with layer planes. When more than one window attempts to realize a palette for a given physical layer plane, only one palette at a time is realized. When you call the **wglRealizeLayerPalette** function, the layer palette of a foreground window is always realized first.

When a window's layer palette is realized, its palette entries are always mapped one-to-one into the physical palette. Unlike GDI logical palettes, with **wglRealizeLayerPalette** there is no mapping of other windows' layer palettes to the current physical palette.

Whenever a window becomes the foreground window, call **wglRealizeLayerPalette** to realize its layer palettes again, even if the pixel type of the layer plane is RGBA.

Because **wglRealizeLayerPalette** doesn't realize the palette of the main plane, use GDI palette functions to realize the main plane palette.

See Also

LAYERPLANEDESCRIPTOR, PIXELFORMATDESCRIPTOR, wglDescribeLayerPlane, wglGetLayerPaletteEntries, wglRealizeLayerPalette, wglSetLayerPaletteEntries

wglSetLayerPaletteEntries Quick Info

Overview

Group

[New - Windows 95, OEM Service Release 2]

The **wglSetLayerPaletteEntries** function sets the palette entries in a given color-index layer plane for a specified device context.

```
int wglSetLayerPaletteEntries(
  HDC   hdc,                  // device context whose layer palette is to be set
  int   iLayerPlane,          // specifies an overlay or underlay plane
  int   iStart,               // specifies the first palette entry to be set
  int   cEntries,             // specifies the number of palette entries to be set
  CONST COLORREF *pcr         // points to the first member of an array of COLORREF structures
);
```

Parameters

hdc

Specifies the device context of a window whose layer palette is to be set.

iLayerPlane

Specifies an overlay or underlay plane. Positive values of *iLayerPlane* identify overlay planes, where 1 is the first overlay plane over the main plane, 2 is the second overlay plane over the first overlay plane, and so on. Negative values identify underlay planes, where -1 is the first underlay plane under the main plane, -2 is the second underlay plane under the first underlay plane, and so on. The number of overlay and underlay planes is given in the **bReserved** member of the **PIXELFORMATDESCRIPTOR** structure.

iStart

Specifies the first palette entry to be set.

cEntries

Specifies the number of palette entries to be set.

pcr

Points to the first member of an array of *cEntries* **COLORREF** structures that contain RGB color information.

Return Values

If the function succeeds, the return value is the number of entries that were set in the palette in the specified layer plane of the window. If the function fails or no pixel format is selected, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

Each color-index plane in a window has a palette with a size 2^n, where *n* is the number of bit planes in the layer plane. You cannot modify the transparent index of a palette.

Use the **wglRealizeLayerPalette** function to realize the layer palette. Initially the layer palette contains only entries for white.

The **wglSetLayerPaletteEntries** function doesn't set the palette entries of the main plane palette. To update the main plane palette, use GDI palette functions.

See Also

LAYERPLANEDESCRIPTOR, **PIXELFORMATDESCRIPTOR**, **wglDescribeLayerPlane**,

wglGetLayerPaletteEntries, **wglRealizeLayerPalette**

wglShareLists Quick Info

Overview

Group

[New - Windows 95, OEM Service Release 2]

The **wglShareLists** function enables multiple OpenGL rendering contexts to share a single display-list space.

```
BOOL wglShareLists(
  HGLRC  hglrc1,     // OpenGL rendering context with which to share display lists
  HGLRC  hglrc2      // OpenGL rendering context to share display lists
);
```

Parameters

hglrc1

Specifies the OpenGL rendering context with which to share display lists.

hglrc2

Specifies the OpenGL rendering context to share display lists with *hglrc1*. The *hglrc2* parameter should not contain any existing display lists when **wglShareLists** is called.

Return Values

When the function succeeds, the return value is TRUE.

When the function fails, the return value is FALSE and the display lists are not shared. To get extended error information, call **GetLastError**.

Remarks

When you create an OpenGL rendering context, it has its own display-list space. The **wglShareLists** function enables a rendering context to share the display-list space of another rendering context; any number of rendering contexts can share a single display-list space. Once a rendering context shares a display-list space, the rendering context always uses the display-list space until the rendering context is deleted. When the last rendering context of a shared display-list space is deleted, the shared display-list space is deleted. All the indexes and definitions of display lists in a shared display-list space are shared.

You can only share display lists with rendering contexts within the same process. However, not all rendering contexts in a process can share display lists. Rendering contexts can share display lists only if they use the same implementation of OpenGL functions. All client rendering contexts of a given pixel format can always share display lists.

All rendering contexts of a shared display list must use an identical pixel format. Otherwise the results depend on the implementation of OpenGL used.

Note The **wglShareLists** function is only available with OpenGL version 1.01 or later. To determine the version number of the implementation of OpenGL, call **glGetString**.

See Also

glGetString

wglSwapLayerBuffers Quick Info

Overview

Group

[New - Windows 95, OEM Service Release 2]

The **wglSwapLayerBuffers** function swaps the front and back buffers in the overlay, underlay, and main planes of the window referenced by a specified device context.

```
BOOL wglSwapLayerBuffers(
  HDC   hdc,          // device context whose layer plane buffers are to be swapped
  UINT  fuPlanes      // specifies the overlay, underlay and main planes to be swapped
);
```

Parameters

hdc

Specifies the device context of a window whose layer plane palette is to be realized into the physical palette.

fuPlanes

Specifies the overlay, underlay, and main planes whose front and back buffers are to be swapped. The **bReserved** member of the **PIXELFORMATDESCRIPTOR** structure specifies the number of overlay and underlay planes. The *fuPlanes* parameter is a bitwise combination of the following values.

Value	Meaning
WGL_SWAP_MAIN_PLANE	Swaps the front and back buffers of the main plane.
WGL_SWAP_OVERLAYi	Swaps the front and back buffers of the overlay plane *i*, where *i* is an integer between 1 and 15. WGL_SWAP_OVERLAY1 identifies the first overlay plane over the main plane, WGL_SWAP_OVERLAY2 identifies the second overlay plane over the first overlay plane, and so on.
WGL_SWAP_UNDERLAYi	Swaps the front and back buffers of the underlay plane *i*, where *i* is an integer between 1 and 15. WGL_SWAP_UNDERLAY1 identifies the first underlay plane under the main plane, WGL_SWAP_UNDERLAY2 identifies the second underlay plane under the first underlay plane, and so on.

Return Values

If the function succeeds, the return value is TRUE. If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

When a layer plane doesn't include a back buffer, calling the **wglSwapLayerBuffers** function has no effect on that layer plane. After you call **wglSwapLayerBuffers**, the state of the back buffer content is given in the corresponding **LAYERPLANEDESCRIPTOR** structure of the layer plane or in the

PIXELFORMATDESCRIPTOR structure of the main plane. The **wglSwapLayerBuffers** function swaps the front and back buffers in the specified layer planes simultaneously.

Some devices don't support swapping layer planes individually; they swap all layer planes as a group. When the PFD_SWAP_LAYER_BUFFERS flag of the **PIXELFORMATDESCRIPTOR** structure is set, it indicates that a device can swap individual layer planes and that you can call **wglSwapLayerBuffers**.

With applications that use multiple threads, before calling **wglSwapLayerBuffers**, clear all drawing commands in all threads drawing to the same window.

See Also

LAYERPLANEDESCRIPTOR, **PIXELFORMATDESCRIPTOR**, **SwapBuffers**

wglUseFontBitmaps Quick Info

Overview

Group

[New - Windows 95, OEM Service Release 2]

The **wglUseFontBitmaps** function creates a set of bitmap display lists for use in the current OpenGL rendering context. The set of bitmap display lists is based on the glyphs in the currently selected font in the device context. You can then use bitmaps to draw characters in an OpenGL image.

The **wglUseFontBitmaps** function creates *count* display lists, one for each of a run of *count* glyphs that begins with the *first* glyph in the *hdc* parameter's selected fonts.

```
BOOL wglUseFontBitmaps(
  HDC    hdc,          // device context whose font will be used
  DWORD  first,        // glyph that is the first of a run of glyphs to be turned into bitmap display
                       lists
  DWORD  count,        // number of glyphs to turn into bitmap display lists
  DWORD  listBase      // specifies starting display list
);
```

Parameters

hdc

Specifies the device context whose currently selected font will be used to form the glyph bitmap display lists in the current OpenGL rendering context.

first

Specifies the first glyph in the run of glyphs that will be used to form glyph bitmap display lists.

count

Specifies the number of glyphs in the run of glyphs that will be used to form glyph bitmap display lists. The function creates *count* display lists, one for each glyph in the run.

listBase

Specifies a starting display list.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

The **wglUseFontBitmaps** function defines *count* display lists in the current OpenGL rendering context. Each display list has an identifying number, starting at *listBase*. Each display list consists of a single call to **glBitmap**. The definition of bitmap *listBase+i* is taken from the glyph *first+i* of the font currently selected in the device context specified by *hdc*. If a glyph is not defined, then the function defines an empty display list for it.

The **wglUseFontBitmaps** function creates bitmap text in the plane of the screen. It enables the labeling of objects in OpenGL.

In the current version of Microsoft's implementation of OpenGL in Windows NT and Windows 95, you cannot make GDI calls to a device context that has a double-buffered pixel format. Therefore, you cannot use the GDI fonts and text functions with such device contexts. You can use the **wglUseFontBitmaps** function to circumvent this limitation and draw text in a double-buffered device context.

The function determines the parameters of each call to **glBitmap** as follows.

glBitmap Parameter	Meaning
width	The width of the glyph's bitmap, as returned in the **gmBlackBoxX** member of the glyph's **GLYPHMETRICS** structure.
height	The height of the glyph's bitmap, as returned in the **gmBlackBoxY** member of the glyph's **GLYPHMETRICS** structure.
xorig	The x offset of the glyph's origin, as returned in the **gmptGlyphOrigin.x** member of the glyph's **GLYPHMETRICS** structure.
yorig	The y offset of the glyph's origin, as returned in the **gmptGlyphOrigin.y** member of the glyph's **GLYPHMETRICS** structure.
xmove	The horizontal distance to the origin of the next character cell, as returned in the **gmCellIncX** member of the glyph's **GLYPHMETRICS** structure.
ymove	The vertical distance to the origin of the next character cell as returned in the **gmCellIncY** member of the glyph's **GLYPHMETRICS** structure.
bitmap	The bitmap for the glyph, as returned by **GetGlyphOutline** with *uFormat* equal to 1.

The following code example shows how to use **wglUseFontBitmaps** to draw some text:

```
HDC    hdc;
HGLRC  hglrc;

// create a rendering context
hglrc = wglCreateContext (hdc);

// make it the calling thread's current rendering context
wglMakeCurrent (hdc, hglrc);

// now we can call OpenGL API

// make the system font the device context's selected font
SelectObject (hdc, GetStockObject (SYSTEM_FONT));

// create the bitmap display lists
// we're making images of glyphs 0 thru 255
// the display list numbering starts at 1000, an arbitrary choice
wglUseFontBitmaps (hdc, 0, 255, 1000);

// display a string:
// indicate start of glyph display lists
glListBase (1000);
// now draw the characters in a string
glCallLists (24, GL_UNSIGNED_BYTE, "Hello Win32 OpenGL World");
```

See Also

GetGlyphOutline, **glBitmap**, **glCallLists**, **glListBase**, **GLYPHMETRICS**, **wglUseFontOutlines**

wglUseFontOutlines Quick Info

Overview

Group

[New - Windows 95, OEM Service Release 2]

The **wglUseFontOutlines** function creates a set of display lists, one for each glyph of the currently selected outline font of a device context, for use with the current rendering context. The display lists are used to draw 3-D characters of TrueType fonts. Each display list describes a glyph outline in floating-point coordinates.

The run of glyphs begins with the *first* glyph of the font of the specified device context. The em square size of the font, the notional grid size of the original font outline from which the font is fitted, is mapped to 1.0 in the x- and y-coordinates in the display lists. The extrusion parameter sets how much depth the font has in the z direction.

The *lpgmf* parameter returns a **GLYPHMETRICSFLOAT** structure that contains information about the placement and orientation of each glyph in a character cell.

```
BOOL wglUseFontOutlines(
  HDC   hdc,                    // device context of the outline font
  DWORD   first,                // first glyph to be turned into a display list
  DWORD   count,                // number of glyphs to be turned into display lists
  DWORD   listBase,             // specifies the starting display list
  FLOAT   deviation,            // specifies the maximum chordal deviation from the true outlines
  FLOAT   extrusion,            // extrusion value in the negative z direction
  int   format,                 // specifies line segments or polygons in display lists
  LPGLYPHMETRICSFLOAT   lpgmf   // address of buffer to receive glyph metric data
);
```

Parameters

hdc

Specifies the device context with the desired outline font. The outline font of *hdc* is used to create the display lists in the current rendering context.

first

Specifies the first of the set of glyphs that form the font outline display lists.

count

Specifies the number of glyphs in the set of glyphs used to form the font outline display lists. The **wglUseFontOutlines** function creates *count* display lists, one display list for each glyph in a set of glyphs.

listBase

Specifies a starting display list.

deviation

Specifies the maximum chordal deviation from the original outlines. When deviation is zero, the chordal deviation is equivalent to one design unit of the original font. The value of *deviation* must be equal to or greater than 0.

extrusion

Specifies how much a font is extruded in the negative *z* direction. The value must be equal to or greater than 0. When *extrusion* is 0, the display lists are not extruded.

format

Specifies the format, either WGL_FONT_LINES or WGL_FONT_POLYGONS, to use in the display lists. When *format* is WGL_FONT_LINES, the **wglUseFontOutlines** function creates fonts with line segments. When *format* is WGL_FONT_POLYGONS, **wglUseFontOutlines** creates fonts with

polygons.

lpgmf

Points to an array of *count* **GLYPHMETRICSFLOAT** structures that is to receive the metrics of the glyphs. When *lpgmf* is NULL, no glyph metrics are returned.

Return Values

When the function succeeds, the return value is TRUE.

When the function fails, the return value is FALSE and no display lists are generated. To get extended error information, call **GetLastError**.

Remarks

The **wglUseFontOutlines** function defines the glyphs of an outline font with display lists in the current rendering context. The **wglUseFontOutlines** function works with TrueType fonts only; stroke and raster fonts are not supported.

Each display list consists of either line segments or polygons, and has a unique identifying number starting with the *listBase* number.

The **wglUseFontOutlines** function approximates glyph outlines by subdividing the quadratic B-spline curves of the outline into line segments, until the distance between the outline and the interpolated midpoint is within the value specified by *deviation*. This is the final format used when *format* is WGL_FONT_LINES. When you specify WGL_FONT_OUTLINES, the display lists created don't contain any normals; thus lighting doesn't work properly. To get the correct lighting of lines use WGL_FONT_POLYGONS and set **glPolygonMode(**GL_FRONT, GL_LINE**)**. When you specify *format* as WGL_FONT_POLYGONS the outlines are further tessellated into separate triangles, triangle fans, triangle strips, or quadrilateral strips to create the surface of each glyph. With WGL_FONT_POLYGONS, the created display lists call **glFrontFace(**GL_CW**)** or **glFrontFace(**GL_CCW**)**; thus the current front-face value might be altered. For the best appearance of text with WGL_FONT_POLYGONS, cull the back faces as follows:

```
glCullFace(GL_BACK);
glEnable(GL_CULL_FACE);
```

A **GLYPHMETRICSFLOAT** structure contains information about the placement and orientation of each glyph in a character cell. The *lpgmf* parameter is an array of **GLYPHMETRICSFLOAT** structures holding the entire set of glyphs for a font. Each display list ends with a translation specified with the **gmfCellIncX** and **gmfCellIncY** members of the corresponding **GLYPHMETRICSFLOAT** structure. The translation enables the drawing of successive characters in their natural direction with a single call to **glCallLists**.

Note With the current release of OpenGL for Windows NT and Windows 95, you cannot make GDI calls to a device context when a pixel format is double-buffered. You can work around this limitation by using **wglUseFontOutlines** and **wglUseFontBitmaps**, when using double-buffered device contexts.

The following code example shows how to draw text using **wglUseFontOutlines**:

```
HDC    hdc;  // A TrueType font has already been selected
HGLRC  hglrc;
GLYPHMETRICSFLOAT agmf[256];

// Make hglrc the calling thread's current rendering context
wglMakeCurrent(hdc, hglrc);
```

```
// create display lists for glyphs 0 through 255 with 0.1 extrusion
// and default deviation. The display list numbering starts at 1000
// (it could be any number)
wglUseFontOutlines(hdc, 0, 255, 1000, 0.0f, 0.1f,
            WGL_FONT_POLYGONS, &agmf);

// Set up transformation to draw the string
glLoadIdentity();
glTranslate(0.0f, 0.0f, -5.0f)
glScalef(2.0f, 2.0f, 2.0f);

// Display a string
glListBase(1000); // Indicates the start of display lists for the glyphs
// Draw the characters in a string
glCallLists(24, GL_UNSIGNED_BYTE, "Hello Win32 OpenGL World.");
```

See Also

glCallLists, **glListBase**, **glTexGen**, **GLYPHMETRICSFLOAT**, **wglUseFontBitmaps**

GLYPHMETRICSFLOAT Quick Info

Overview

Group

[New - Windows 95, OEM Service Release 2]

The **GLYPHMETRICSFLOAT** structure contains information about the placement and orientation of a glyph in a character cell.

```
typedef struct _GLYPHMETRICSFLOAT { // gmf
    FLOAT      gmfBlackBoxX;
    FLOAT      gmfBlackBoxY;
    POINTFLOAT gmfptGlyphOrigin;
    FLOAT      gmfCellIncX;
    FLOAT      gmfCellIncY;
} GLYPHMETRICSFLOAT;
```

Members

gmfBlackBoxX
Specifies the width of the smallest rectangle (the glyph's black box) that completely encloses the glyph.

gmfBlackBoxY
Specifies the height of the smallest rectangle (the glyph's black box) that completely encloses the glyph.

gmfptGlyphOrigin
Specifies the x and y coordinates of the upper-left corner of the smallest rectangle that completely encloses the glyph.

gmfCellIncX
Specifies the horizontal distance from the origin of the current character cell to the origin of the next character cell.

gmfCellIncY
Specifies the vertical distance from the origin of the current character cell to the origin of the next character cell.

Remarks

The values of **GLYPHMETRICSFLOAT** are specified as notional units.

See Also

POINTFLOAT, **wglUseFontOutlines**

LAYERPLANEDESCRIPTOR Overview

Group

[New - Windows 95, OEM Service Release 2]

The **LAYERPLANEDESCRIPTOR** structure describes the pixel format of a drawing surface.

```
typedef struct tagLAYERPLANEDESCRIPTOR { // pfd
    WORD  nSize;
    WORD  nVersion;
    DWORD dwFlags;
    BYTE  iPixelType;
    BYTE  cColorBits;
    BYTE  cRedBits;
    BYTE  cRedShift;
    BYTE  cGreenBits;
    BYTE  cGreenShift;
    BYTE  cBlueBits;
    BYTE  cBlueShift;
    BYTE  cAlphaBits;
    BYTE  cAlphaShift;
    BYTE  cAccumBits;
    BYTE  cAccumRedBits;
    BYTE  cAccumGreenBits;
    BYTE  cAccumBlueBits;
    BYTE  cAccumAlphaBits;
    BYTE  cDepthBits;
    BYTE  cStencilBits;
    BYTE  cAuxBuffers;
    BYTE  iLayerPlane
    BYTE  bReserved;
    COLORREF crTransparent;
} LAYERPLANEDESCRIPTOR;
```

Members

nSize
Specifies the size of this data structure. Set this value to **sizeof(LAYERPLANEDESCRIPTOR)**.

nVersion
Specifies the version of this data structure. Set this value to 1.

dwFlags
A set of bit flags that specify properties of the layer plane. The properties are generally not mutually exclusive; any combination of bit flags can be set, with the exceptions noted. The following bit flag constants are defined.

Value	Meaning
LPD_SUPPORT_OPENGL	The layer plane supports OpenGL drawing.
LPD_SUPPORT_GDI	The layer plane supports GDI drawing. The current implementation of OpenGL doesn't support this flag.
LPD_DOUBLEBUFFER	The layer plane is double-buffered. A layer plane can be double-buffered even when the main plane is single-

	buffered and vice versa.
LPD_STEREO	The layer plane is stereoscopic. A layer plane can be stereoscopic even when the main plane is monoscopic and vice versa.
LPD_SWAP_EXCHANGE	In a double-buffered layer plane, swapping the color buffer exchanges the front buffer and back buffer contents. The back buffer then contains the contents of the front buffer before the swap. This flag is a hint only and might not be provided by a driver.
LPD_SWAP_COPY	In a double-buffered layer plane, swapping the color buffer copies the back buffer contents to the front buffer. The swap does not affect the back buffer contents. This flag is a hint only and might not be provided by a driver.
LPD_TRANSPARENT	The **crTransparent** member of this structure contains a transparent color or index value that enables underlying layers to show through this layer. All layer planes, except the lowest-numbered underlay layer, have a transparent color or index.
LPD_SHARE_DEPTH	The layer plane shares the depth buffer with the main plane.
LPD_SHARE_STENCIL	The layer plane shares the stencil buffer with the main plane.
LPD_SHARE_ACCUM	The layer plane shares the accumulation buffer with the main plane.

iPixelType

Specifies the type of pixel data. The following types are defined.

Value	Meaning
LPD_TYPE_RGBA	RGBA pixels. Each pixel has four components: red, green, blue, and alpha.
LPD_TYPE_COLORINDEX	Color-index pixels. Each pixel uses a color-index value.

cColorBits

Specifies the number of color bitplanes in each color buffer. For RGBA pixel types, it is the size of the color buffer, excluding the alpha bitplanes. For color-index pixels, it is the size of the color-index buffer.

cRedBits

Specifies the number of red bitplanes in each RGBA color buffer.

cRedShift

Specifies the shift count for red bitplanes in each RGBA color buffer.

cGreenBits

Specifies the number of green bitplanes in each RGBA color buffer.

cGreenShift
Specifies the shift count for green bitplanes in each RGBA color buffer.

cBlueBits
Specifies the number of blue bitplanes in each RGBA color buffer.

cBlueShift
Specifies the shift count for blue bitplanes in each RGBA color buffer.

cAlphaBits
Specifies the number of alpha bitplanes in each RGBA color buffer. Alpha bitplanes are not supported.

cAlphaShift
Specifies the shift count for alpha bitplanes in each RGBA color buffer. Alpha bitplanes are not supported.

cAccumBits
Specifies the total number of bitplanes in the accumulation buffer.

cAccumRedBits
Specifies the number of red bitplanes in the accumulation buffer.

cAccumGreenBits
Specifies the number of green bitplanes in the accumulation buffer.

cAccumBlueBits
Specifies the number of blue bitplanes in the accumulation buffer.

cAccumAlphaBits
Specifies the number of alpha bitplanes in the accumulation buffer.

cDepthBits
Specifies the depth of the depth (z-axis) buffer.

cStencilBits
Specifies the depth of the stencil buffer.

cAuxBuffers
Specifies the number of auxiliary buffers. Auxiliary buffers are not supported.

iLayerType
Specifies the layer plane number. Positive values of *iLayerType* identify overlay planes, where 1 is the first overlay plane over the main plane, 2 is the second overlay plane over the first overlay plane, and so on. Negative values identify underlay planes, where -1 is the first underlay plane under the main plane, -2 is the second underlay plane under the first underlay plane, and so on. The number of overlay and underlay planes is given in the **bReserved** member of the **PIXELFORMATDESCRIPTOR** structure.

bReserved
Not used. Must be zero.

crTransparent
When the LPD_TRANSPARENT flag is set, specifies the transparent color or index value. Typically the value is zero.

Remarks

Please notice, as documented above, that certain layer plane properties are not supported in the current implementation. The implementation is the Microsoft GDI software implementation of OpenGL. Hardware manufacturers that enhance parts of OpenGL may support some layer plane properties not supported by the generic implementation.

See Also

PIXELFORMATDESCRIPTOR, **wglCreateLayerContext**, **wglDescribeLayerPlane**, **wglGetLayerPaletteEntries**, **wglRealizeLayerPalette**, **wglSetLayerPaletteEntries**, **wglSwapLayerBuffers**

PIXELFORMATDESCRIPTOR Overview

Group

[New - Windows 95, OEM Service Release 2]

The **PIXELFORMATDESCRIPTOR** structure describes the pixel format of a drawing surface.

```
typedef struct tagPIXELFORMATDESCRIPTOR { // pfd
    WORD  nSize;
    WORD  nVersion;
    DWORD dwFlags;
    BYTE  iPixelType;
    BYTE  cColorBits;
    BYTE  cRedBits;
    BYTE  cRedShift;
    BYTE  cGreenBits;
    BYTE  cGreenShift;
    BYTE  cBlueBits;
    BYTE  cBlueShift;
    BYTE  cAlphaBits;
    BYTE  cAlphaShift;
    BYTE  cAccumBits;
    BYTE  cAccumRedBits;
    BYTE  cAccumGreenBits;
    BYTE  cAccumBlueBits;
    BYTE  cAccumAlphaBits;
    BYTE  cDepthBits;
    BYTE  cStencilBits;
    BYTE  cAuxBuffers;
    BYTE  iLayerType;
    BYTE  bReserved;
    DWORD dwLayerMask;
    DWORD dwVisibleMask;
    DWORD dwDamageMask;
} PIXELFORMATDESCRIPTOR;
```

Members

nSize

Specifies the size of this data structure. This value should be set to **sizeof(PIXELFORMATDESCRIPTOR)**.

nVersion

Specifies the version of this data structure. This value should be set to 1.

dwFlags

A set of bit flags that specify properties of the pixel buffer. The properties are generally not mutually exclusive; you can set any combination of bit flags, with the exceptions noted. The following bit flag constants are defined.

Value	Meaning
PFD_DRAW_TO_WINDOW	The buffer can draw to a window or device surface.
PFD_DRAW_TO_BITMAP	The buffer can draw to a memory bitmap.
PFD_SUPPORT_GDI	The buffer supports GDI drawing. This flag and

	PFD_DOUBLEBUFFER are mutually exclusive in the current generic implementation.
PFD_SUPPORT_OPENGL	The buffer supports OpenGL drawing.
PFD_GENERIC_ACCELERATED	The pixel format is supported by a device driver that accelerates the generic implementation. If this flag is clear and the PFD_GENERIC_FORMAT flag is set, the pixel format is supported by the generic implementation only.
PFD_GENERIC_FORMAT	The pixel format is supported by the GDI software implementation, which is also known as the generic implementation. If this bit is clear, the pixel format is supported by a device driver or hardware.
PFD_NEED_PALETTE	The buffer uses RGBA pixels on a palette-managed device. A logical palette is required to achieve the best results for this pixel type. Colors in the palette should be specified according to the values of the **cRedBits**, **cRedShift**, **cGreenBits**, **cGreenShift**, **cBluebits**, and **cBlueShift** members. The palette should be created and realized in the device context before calling **wglMakeCurrent**.
PFD_NEED_SYSTEM_PALETTE	Defined in the pixel format descriptors of hardware that supports one hardware palette in 256-color mode only. For such systems to use hardware acceleration, the hardware palette must be in a fixed order (for example, 3-3-2) when in RGBA mode or must match the logical palette when in color-index mode. When this flag is set, you must call **SetSystemPaletteUse** in your program to force a one-to-one mapping of the logical palette and the system palette. If your OpenGL hardware supports multiple hardware palettes and the device driver can allocate spare hardware palettes for OpenGL, this flag is typically

	clear. This flag is not set in the generic pixel formats.
PFD_DOUBLEBUFFER	The buffer is double-buffered. This flag and PFD_SUPPORT_GDI are mutually exclusive in the current generic implementation.
PFD_STEREO	The buffer is stereoscopic. This flag is not supported in the current generic implementation.
PFD_SWAP_LAYER_BUFFERS	Indicates whether a device can swap individual layer planes with pixel formats that include double-buffered overlay or underlay planes. Otherwise all layer planes are swapped together as a group. When this flag is set, **wglSwapLayerBuffers** is supported.

You can specify the following bit flags when calling **ChoosePixelFormat**.

Value	Meaning
PFD_DEPTH_DONTCARE	The requested pixel format can either have or not have a depth buffer. To select a pixel format without a depth buffer, you must specify this flag. The requested pixel format can be with or without a depth buffer. Otherwise, only pixel formats with a depth buffer are considered.
PFD_DOUBLEBUFFER_DONTCARE	The requested pixel format can be either single- or double-buffered.
PFD_STEREO_DONTCARE	The requested pixel format can be either monoscopic or stereoscopic.

With the **glAddSwapHintRectWIN** extension function, two new flags are included for the **PIXELFORMATDESCRIPTOR** pixel format structure.

Value	Meaning
PFD_SWAP_COPY	Specifies the content of the back buffer in the double-buffered main color plane following a buffer swap. Swapping the color buffers causes the content of the back buffer to be copied to the front buffer. The content of the back buffer is not affected by the swap. PFD_SWAP_COPY is a hint only and might not be

	provided by a driver.
PFD_SWAP_EXCHANGE	Specifies the content of the back buffer in the double-buffered main color plane following a buffer swap. Swapping the color buffers causes the exchange of the back buffer's content with the front buffer's content. Following the swap, the back buffer's content contains the front buffer's content before the swap. PFD_SWAP_EXCHANGE is a hint only and might not be provided by a driver.

iPixelType

Specifies the type of pixel data. The following types are defined.

Value	Meaning
PFD_TYPE_RGBA	RGBA pixels. Each pixel has four components in this order: red, green, blue, and alpha.
PFD_TYPE_COLORINDEX	Color-index pixels. Each pixel uses a color-index value.

cColorBits

Specifies the number of color bitplanes in each color buffer. For RGBA pixel types, it is the size of the color buffer, excluding the alpha bitplanes. For color-index pixels, it is the size of the color-index buffer.

cRedBits

Specifies the number of red bitplanes in each RGBA color buffer.

cRedShift

Specifies the shift count for red bitplanes in each RGBA color buffer.

cGreenBits

Specifies the number of green bitplanes in each RGBA color buffer.

cGreenShift

Specifies the shift count for green bitplanes in each RGBA color buffer.

cBlueBits

Specifies the number of blue bitplanes in each RGBA color buffer.

cBlueShift

Specifies the shift count for blue bitplanes in each RGBA color buffer.

cAlphaBits

Specifies the number of alpha bitplanes in each RGBA color buffer. Alpha bitplanes are not supported.

cAlphaShift

Specifies the shift count for alpha bitplanes in each RGBA color buffer. Alpha bitplanes are not supported.

cAccumBits

Specifies the total number of bitplanes in the accumulation buffer.

cAccumRedBits

Specifies the number of red bitplanes in the accumulation buffer.

cAccumGreenBits

Specifies the number of green bitplanes in the accumulation buffer.

cAccumBlueBits

Specifies the number of blue bitplanes in the accumulation buffer.

cAccumAlphaBits
Specifies the number of alpha bitplanes in the accumulation buffer.

cDepthBits
Specifies the depth of the depth (z-axis) buffer.

cStencilBits
Specifies the depth of the stencil buffer.

cAuxBuffers
Specifies the number of auxiliary buffers. Auxiliary buffers are not supported.

iLayerType
Ignored. Earlier implementations of OpenGL used this member, but it is no longer used.

bReserved
Specifies the number of overlay and underlay planes. Bits 0 through 3 specify up to 15 overlay planes and bits 4 through 7 specify up to 15 underlay planes.

dwLayerMask
Ignored. Earlier implementations of OpenGL used this member, but it is no longer used.

dwVisibleMask
Specifies the transparent color or index of an underlay plane. When the pixel type is RGBA, **dwVisibleMask** is a transparent RGB color value. When the pixel type is color index, it is a transparent index value.

dwDamageMask
Ignored. Earlier implementations of OpenGL used this member, but it is no longer used.

Remarks

Please notice carefully, as documented above, that certain pixel format properties are not supported in the current generic implementation. The generic implementation is the Microsoft GDI software implementation of OpenGL. Hardware manufacturers may enhance parts of OpenGL, and may support some pixel format properties not supported by the generic implementation.

See Also

ChoosePixelFormat, **DescribePixelFormat**, **GetPixelFormat**, **SetPixelFormat**

POINTFLOAT Quick Info

[New - Windows 95, OEM Service Release 2]

The **POINTFLOAT** structure contains the x and y coordinates of a point.

```
typedef struct _POINTFLOAT { // ptf
    FLOAT       x;
    FLOAT       y;
} POINTFLOAT;
```

Members

x

Specifies the horizontal (x) coordinate of a point.

y

Specifies the vertical (y) coordinate of a point.

See Also

GLYPHMETRICSFLOAT

Programming Tips

This section lists some useful programming tips and guidelines. Keep in mind that these tips are based on the intentions of the designers of OpenGL, not on any experience with actual applications and implementations. This section has the following topics:

OpenGL Correctness Tips

Follow these guidelines to create OpenGL applications that perform as you intend:

- Assume error behavior on the part of an OpenGL implementation may change in a future release of OpenGL. OpenGL error semantics may change between upward-compatible revisions.
- Use the projection matrix to collapse all geometry to a single plane. If the modelview matrix is used, OpenGL features that operate in eye coordinates (such as lighting and application-defined clipping planes) might fail.
- Do not make extensive changes to a single matrix. For example, do not animate a rotation by continually calling **glRotate** with an incremental angle. Rather, use **glLoadIdentity** to initialize the given matrix for each frame, and then call **glRotate** with the desired complete angle for that frame.
- Expect multiple passes through a rendering database to generate the same pixel fragments only if this behavior is guaranteed by the invariance rules established for a compliant OpenGL implementation. Otherwise, multiple passes might generate varying sets of fragments.
- Do not expect errors to be reported while a display list is being defined. The commands within a display list generate errors only when the list is executed.
- To optimize the operation of the depth buffer, place the near frustum plane as far from the viewpoint as possible.
- Call **glFlush** to force execution of all previous OpenGL commands. Do not count on **glGet** or **glIsEnabled** to flush the rendering stream. Query commands flush as much of the stream as is required to return valid data but don't necessarily complete all pending rendering commands.
- Turn dithering off when rendering predithered images (for example, when **glCopyPixels** is called).
- Make use of the full range of the accumulation buffer. For example, if accumulating four images, scale each by one-quarter as it's accumulated.
- To obtain exact two-dimensional rasterization, carefully specify both the orthographic projection and the vertices of primitives that are to be rasterized. Specify the orthographic projection with integer coordinates, as shown in the following example:

  ```
  gluOrtho2D(0, width, 0, height);
  ```

  The parameters *width* and *height* are the dimensions of the viewport. Given this projection matrix, place polygon vertices and pixel image positions at integer coordinates to rasterize predictably. For example, **glRecti(**0, 0, 1, 1**)** reliably fills the lower-left pixel of the viewport, and **glRasterPos2i(**0, 0**)** reliably positions an unzoomed image at the lower-left pixel of the viewport. However, point vertices, line vertices, and bitmap positions should be placed at half-integer locations. For example, a line drawn from ($x_{(1)}$, 0.5) to ($x_{(2)}$, 0.5) will be reliably rendered along the bottom row of pixels in the viewport, and a point drawn at (0.5, 0.5) will reliably fill the same pixel as **glRecti(**0, 0, 1, 1**)**.

  An optimum compromise that allows all primitives to be specified at integer positions, while still ensuring predictable rasterization, is to translate *x* and *y* by 0.375, as shown in the following code sample. Such a translation keeps polygon and pixel image edges safely away from the centers of pixels, while moving line vertices close enough to the pixel centers.

  ```
  glViewport(0, 0, width, height);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity( );
  gluOrtho2D(0, width, 0, height);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity( );
  glTranslatef(0.375, 0.375, 0.0);
  /* render all primitives at integer positions */
  ```

- Avoid using negative *w* vertex coordinates and negative *q* texture coordinates. OpenGL might not clip such coordinates correctly and can make interpolation errors when shading primitives defined by such

coordinates.

OpenGL Performance Tips

These programming practices optimize your application's performance:

- Use **glColorMaterial** when only a single material property is being varied rapidly (at each vertex, for example). Use **glMaterial** for infrequent changes, or when more than a single material property is being varied rapidly.
- Use **glLoadIdentity** to initialize a matrix, rather than loading your own copy of the identity matrix.
- Use specific matrix calls such as **glRotate**, **glTranslate**, and **glScale**, rather than composing your own rotation, translation, and scale matrices and calling **glMultMatrix**.
- Use **glPushAttrib** and **glPopAttrib** to save and restore state values. Use query functions only when your application requires the state values for its own computations.
- Use display lists to encapsulate potentially expensive state changes. For example, place all the **glTexImage** calls required to completely specify a texture (and perhaps the associated **glTexParameter**, **glPixelStore**, and **glPixelTransfer** calls as well) into a single display list. Call this display list to select the texture.
- Use display lists to encapsulate the rendering calls of rigid objects that will be drawn repeatedly.
- To minimize network bandwidth in client/server environments, use evaluators even for simple surface tessellations.
- If possible, to avoid the overhead of GL_NORMALIZE, provide unit-length normals. Because **glScale** almost always requires enabling GL_NORMALIZE, avoid using **glScale** when doing lighting.
- If smooth shading isn't required, set **glShadeModel** to GL_FLAT.
- Use a single **glClear** call per frame, if possible. Do not use **glClear** to clear small subregions of the buffers; use it only to clear the buffer completely or nearly completely.
- To draw multiple independent triangles, use a single call rather than multiple calls to **glBegin**(GL_TRIANGLES) or a call to **glBegin**(GL_POLYGON). Similarly:

  To draw even a single triangle, use GL_TRIANGLES rather than GL_POLYGON.

  Use a single call to **glBegin**(GL_QUADS) rather than calling **glBegin**(GL_POLYGON) repeatedly.

  Use a single call to **glBegin**(GL_LINES) to draw multiple independent line segments, rather than calling **glBegin**(GL_LINES) multiple times.
- In general, use the vector forms of commands to pass precomputed data, and use the scalar forms of commands to pass values that are computed near call time.
- Avoid making redundant mode changes, such as setting the color to the same value between each vertex of a flat-shaded polygon.
- When drawing or copying images, disable rasterization and per-fragment operations, which consume a lot of resources. OpenGL can apply textures to pixel images.

Attribute Groups

Mask Bit	Attribute Group
GL_ACCUM_BUFFER_BIT	accum-buffer
GL_ALL_ATTRIB_BITS	–
GL_COLOR_BUFFER_BIT	color-buffer
GL_CURRENT_BIT	current
GL_DEPTH_BUFFER_BIT	depth-buffer
GL_ENABLE_BIT	enable
GL_EVAL_BIT	eval
GL_FOG_BIT	fog
GL_HINT_BIT	hint
GL_LIGHTING_BIT	lighting
GL_LINE_BIT	line
GL_LIST_BIT	list
GL_PIXEL_MODE_BIT	pixel
GL_POINT_BIT	point
GL_POLYGON_BIT	polygon
GL_POLYGON_STIPPLE_BIT	polygon-stipple
GL_SCISSOR_BIT	scissor
GL_STENCIL_BUFFER_BIT	stencil-buffer
GL_TEXTURE_BIT	texture
GL_TRANSFORM_BIT	transform
GL_VIEWPORT_BIT	viewport

void **glPopAttrib**(void);

Restores the values of those state variables that were saved with the last **glPushAttrib**.

State Variables for Current Values and Associated Data

GL_CURRENT_COLOR

Description:	Current color
Attribute group:	current
Initial value:	1,1,1,1
Get command:	**glGetIntegerv** **glGetFloatv**

GL_CURRENT_INDEX

Description:	Current color index
Attribute group:	current
Initial value:	1
Get command:	**glGetIntegerv** **glGetFloatv**

GL_CURRENT_TEXTURE_COORDS

Description:	Current texture coordinates
Attribute group:	current
Initial value:	0,0,0,1
Get command:	**glGetFloatv**

GL_CURRENT_NORMAL

Description:	Current normal
Attribute group:	current
Initial value:	0,0,1
Get command:	**glGetFloatv**

GL_CURRENT_RASTER POSITION

Description:	Current raster position
Attribute group:	current
Initial value:	0,0,0,1
Get command:	**glGetFloatv**

GL_CURRENT_RASTER_DISTANCE

Description:	Current raster distance
Attribute group:	current
Initial value:	0
Get command:	**glGetFloatv**

GL_CURRENT_RASTER_COLOR

Description:	Color associated with raster position
Attribute group:	current
Initial value:	1,1,1,1
Get command:	**glGetIntegerv** **glGetFloatv**

GL_CURRENT_RASTER_INDEX

Description:	Color index associated with raster position
Attribute group:	current

Initial value: 1

Get command: **glGetIntegerv**
glGetFloatv

GL_CURRENT_RASTER_TEXTURE_COORDS

Description: Texture coordinates associated with raster position

Attribute group: current

Initial value: 0,0,0,1

Get command: **glGetFloatv**

GL_CURRENT_RASTER_POSITION_VALID

Description: Raster position valid bit

Attribute group: current

Initial value: GL_TRUE

Get command: **glGetBooleanv**

GL_EDGE_FLAG

Description: Edge flag

Attribute group: current

Initial value: GL_TRUE

Get command: **glGetBooleanv**

Transformation State Variables

GL_MODELVIEW_MATRIX

Description:	Modelview matrix stack
Attribute group:	–
Initial value:	Identity
Get command:	**glGetFloatv**

GL_PROJECTION_MATRIX

Description:	Projection matrix stack
Attribute group:	–
Initial value:	Identity
Get command:	**glGetFloatv**

GL_TEXTURE_MATRIX

Description:	Texture matrix stack
Attribute group:	–
Initial value:	Identity
Get command:	**glGetFloatv**

GL_VIEWPORT

Description:	Viewport origin and extent
Attribute group:	viewport
Initial value:	–
Get command:	**glGetIntegerv**

GL_DEPTH_RANGE

Description:	Depth range near and far
Attribute group:	viewport
Initial value:	0,1
Get command:	**glGetFloatv**

GL_MODELVIEW_STACK_DEPTH

Description:	Modelview matrix stack pointer
Attribute group:	–
Initial value:	1
Get command:	**glGetIntegerv**

GL_PROJECTION_STACK_DEPTH

Description:	Projection matrix stack pointer
Attribute group:	–
Initial value:	1
Get command:	**glGetIntegerv**

GL_TEXTURE_STACK_DEPTH

Description:	Texture matrix stack pointer
Attribute group:	–
Initial value:	1
Get command:	**glGetIntegerv**

GL_MATRIX_MODE

Description:	Current matrix mode
Attribute group:	transform
Initial value:	GL_MODELVIEW
Get command:	**glGetIntegerv**

GL_NORMALIZE

Description:	Current normal normalization on/off
Attribute group:	transform/enable
Initial value:	GL_FALSE
Get command:	**glIsEnabled**

GL_CLIP_PLANE*i*

Description:	User clipping plane coefficients
Attribute group:	transform
Initial value:	0,0,0,0
Get command:	**glGetClipPlane**

GL_CLIP_PLANE*i*

Description:	*i*th user clipping plane enabled
Attribute group:	transform/enable
Initial value:	GL_FALSE
Get command:	**glIsEnabled**

Coloring State Variables

GL_FOG_COLOR

Description:	Fog color
Attribute group:	fog
Initial value:	0,0,0,0
Get command:	**glGetFloatv**

GL_FOG_INDEX

Description:	Fog index
Attribute group:	fog
Initial value:	0
Get command:	**glGetFloatv**

GL_FOG_DENSITY

Description:	Exponential fog density
Attribute group:	fog
Initial value:	1.0
Get command:	**glGetFloatv**

GL_FOG_START

Description:	Linear fog start
Attribute group:	fog
Initial value:	0.0
Get command:	**glGetFloatv**

GL_FOG_END

Description:	Linear fog end
Attribute group:	fog
Initial value:	1.0
Get command:	**glGetFloatv**

GL_FOG_MODE

Description:	Fog mode
Attribute group:	fog
Initial value:	GL_EXP
Get command:	**glGetIntegerv**

GL_FOG

Description:	True if fog enabled
Attribute group:	fog/enable
Initial value:	GL_FALSE
Get command:	**glIsEnabled**

GL_SHADE_MODEL

Description:	**glShadeModel** setting
Attribute group:	lighting
Initial value:	GL_SMOOTH
Get command:	**glGetIntegerv**

Lighting State Variables

GL_LIGHTING

Description:	True if lighting is enabled
Attribute group:	lighting/enable
Initial value:	GL_FALSE
Get command:	**glIsEnabled**

GL_COLOR_MATERIAL

Description:	True if color tracking is enabled
Attribute group:	lighting
Initial value:	GL_FALSE
Get command:	**glIsEnabled**

GL_COLOR_MATERIAL_PARAMETER

Description:	Material properties tracking current color
Attribute group:	lighting
Initial value:	GL_AMBIENT_AND_DIFFUSE
Get command:	**glGetIntegerv**

GL_COLOR_MATERIAL_FACE

Description:	Face(s) affected by color tracking
Attribute group:	lighting
Initial value:	GL_FRONT_AND_BACK
Get command:	**glGetIntegerv**

GL_AMBIENT

Description:	Ambient material color
Attribute group:	lighting
Initial value:	(0.2, 0.2, 0.2, 1.0)
Get command:	**glGetMaterialfv**

GL_DIFFUSE

Description:	Diffuse material color
Attribute group:	lighting
Initial value:	(0.8, 0.8, 0.8, 1.0)
Get command:	**glGetMaterialfv**

GL_SPECULAR

Description:	Specular material color
Attribute group:	lighting
Initial value:	(0.0, 0.0, 0.0, 1.0)
Get command:	**glGetMaterialfv**

GL_EMISSION

Description:	Emissive material color
Attribute group:	lighting
Initial value:	(0.0, 0.0, 0.0, 1.0)
Get command:	**glGet**

GL_SHININESS

Description:	Specular exponent of material
Attribute group:	lighting
Initial value:	0.0
Get command:	**glGetMaterialfv**

GL_LIGHT_MODEL_AMBIENT

Description:	Ambient scene color
Attribute group:	lighting
Initial value:	(0.2, 0.2, 0.2, 0.1)
Get command:	**glGetFloatv**

GL_LIGHT_MODEL_LOCAL_VIEWER

Description:	Viewer is local
Attribute group:	lighting
Initial value:	GL_FALSE
Get command:	**glGetBooleanv**

GL_LIGHT_MODEL_TWO_SIDE

Description:	Use two-sided lighting
Attribute group:	lighting
Initial value:	GL_FALSE
Get command:	**glGetBooleanv**

GL_AMBIENT

Description:	Ambient intensity of light *i*
Attribute group:	lighting
Initial value:	(0.0, 0.0, 0.0, 1.0)
Get command:	**glGetLightfv**

GL_DIFFUSE

Description:	Diffuse intensity of light *i*
Attribute group:	lighting
Initial value:	–
Get command:	**glGetLightfv**

GL_SPECULAR

Description:	Specular intensity of light *i*
Attribute group:	lighting
Initial value:	–
Get command:	**glGetLightfv**

GL_POSITION

Description:	Position of light *i*
Attribute group:	lighting
Initial value:	(0.0, 0.0, 1.0, 0.0)
Get command:	**glGetLightfv**

GL_CONSTANT_ATTENUATION

Description:	Constant attenuation factor
Attribute group:	lighting

Initial value: 1.0
Get command: **glGetLightfv**

GL_LINEAR_ATTENUATION
Description: Linear attenuation factor
Attribute group: lighting
Initial value: 0.0
Get command: **glGetLightfv**

GL_QUADRATIC_ATTENUATION
Description: Quadratic attenuation factor
Attribute group: lighting
Initial value: 0.0
Get command: **glGetLightfv**

GL_SPOT_DIRECTION
Description: Spotlight direction of light *i*
Attribute group: lighting
Initial value: (0.0, 0.0, -1.0)
Get command: **glGetLightfv**

GL_SPOT_EXPONENT
Description: Spotlight exponent of light *i*
Attribute group: lighting
Initial value: 0.0
Get command: **glGetLightfv**

GL_SPOT_CUTOFF
Description: Spotlight angle of light *i*
Attribute group: lighting
Initial value: 180.0
Get command: **glGetLightfv**

GL_LIGHT*i*
Description: True if light *i* enabled
Attribute group: lighting/enable
Initial value: GL_FALSE
Get command: **glIsEnabled**

GL_COLOR_INDEXES
Description: $C_{(a)}$, $C_{(d)}$, and $C_{(s)}$ for color-index lighting
Attribute group: lighting/enable
Initial value: 0, 1, 1
Get command: **glGetFloatv**

Rasterization State Variables

GL_POINT_SIZE

Description:	Point size
Attribute group:	point
Initial value:	1.0
Get command:	**glGetFloatv**

GL_POINT_SMOOTH

Description:	Point aliasing on
Attribute group:	point/enable
Initial value:	GL_FALSE
Get command:	**glIsEnabled**

GL_LINE_WIDTH

Description:	Line width
Attribute group:	line
Initial value:	1.0
Get command:	**glGetFloatv**

GL_LINE_SMOOTH

Description:	Line antialiasing on
Attribute group:	line/enable
Initial value:	GL_FALSE
Get command:	**glIsEnabled**

GL_LINE_STIPPLE_PATTERN

Description:	Line stipple
Attribute group:	line
Initial value:	1's
Get command:	**glGetIntegerv**

GL_LINE_STIPPLE_REPEAT

Description:	Line stipple repeat
Attribute group:	line
Initial value:	1
Get command:	**glGetIntegerv**

GL_LINE_STIPPLE

Description:	Line stipple enable
Attribute group:	line/enable
Initial value:	GL_FALSE
Get command:	**glIsEnabled**

GL_CULL_FACE

Description:	Polygon culling enabled
Attribute group:	polygon/enable
Initial value:	GL_FALSE
Get command:	**glIsEnabled**

GL_CULL_FACE_MODE

Description:	Cull front-/back-facing polygons
Attribute group:	polygon
Initial value:	GL_BACK
Get command:	**glGetIntegerv**

GL_FRONT_FACE

Description:	Polygon front-face CW/CCW indicator
Attribute group:	polygon
Initial value:	GL_CCW
Get command:	**glGetIntegerv**

GL_POLYGON_SMOOTH

Description:	Polygon antialiasing on
Attribute group:	polygon/enable
Initial value:	GL_FALSE
Get command:	**glIsEnabled**

GL_POLYGON_MODE

Description:	Polygon rasterization mode (front and back)
Attribute group:	polygon
Initial value:	GL_FILL
Get command:	**glGetIntegerv**

GL_POLYGON_STIPPLE

Description:	Polygon stipple enable
Attribute group:	polygon/enable
Initial value:	GL_FALSE
Get command:	**glIsEnabled**

–

Description:	Polygon stipple pattern
Attribute group:	polygon-stipple
Initial value:	1's
Get command:	**glGetPolygonStipple**

Texturing State Variables

GL_TEXTURE_*x*

Description:	True if *x*-D texturing enabled (*x* is 1-D or 2-D)
Attribute group:	texture/enable
Initial value:	GL_FALSE
Get command:	**glIsEnabled**

GL_TEXTURE

Description:	*x*-D texture image at level of detail *i*
Attribute group:	–
Initial value:	–
Get command:	**glGetTexImage**

GL_TEXTURE_WIDTH

Description:	*x*-D texture image *i*'s width
Attribute group:	–
Initial value:	0
Get command:	**glGetTexLevelParameter**

GL_TEXTURE_HEIGHT

Description:	*x*-D texture image *i*'s height
Attribute group:	–
Initial value:	0
Get command:	**glGetTexLevelParameter**

GL_TEXTURE_BORDER

Description:	*x*-D texture image *i*'s border
Attribute group:	–
Initial value:	0
Get command:	**glGetTexLevelParameter**

GL_TEXTURE_COMPONENTS

Description:	Texture image components
Attribute group:	–
Initial value:	1
Get command:	**glGetTexLevelParameter**

GL_TEXTURE_BORDER_COLOR

Description:	Texture border color
Attribute group:	texture
Initial value:	0,0,0,0
Get command:	**glGetTexParameter**

GL_TEXTURE_MIN_FILTER

Description:	Texture minification function
Attribute group:	texture
Initial value:	GL_NEAREST_MIPMAP_LINEAR
Get command:	**glGetTexParameter**

GL_TEXTURE_MAG_FILTER

Description:	Texture magnification function
Attribute group:	texture
Initial value:	GL_LINEAR
Get command:	**glGetTexParameter**

GL_TEXTURE_WRAP_*x*

Description:	Texture wrap mode (*x* is S or T)
Attribute group:	texture
Initial value:	GL_REPEAT
Get command:	**glGetTexParameter**

GL_TEXTURE_ENV_MODE

Description:	Texture application function
Attribute group:	texture
Initial value:	GL_MODULATE
Get command:	**glGetTexEnviv**

GL_TEXTURE_ENV_COLOR

Description:	Texture environment color
Attribute group:	texture
Initial value:	0,0,0,0
Get command:	**glGetTexEnvfv**

GL_TEXTURE_GEN_*x*

Description:	Texgen is enabled (*x* is S, T, R, or Q)
Attribute group:	texture/enable
Initial value:	GL_FALSE
Get command:	**glIsEnabled**

GL_EYE_LINEAR

Description:	Texgen plane equation coefficients
Attribute group:	texture
Initial value:	–
Get command:	**glGetTexGenfv**

GL_OBJECT_LINEAR

Description:	Texgen object linear coefficients
Attribute group:	texture
Initial value:	–
Get command:	**glGetTexGenfv**

GL_TEXTURE_GEN_MODE

Description:	Function used for texgen
Attribute group:	texture
Initial value:	GL_EYTE_LINEAR
Get command:	**glGetTexGeniv**

Pixel Operations

GL_SCISSOR_TEST

Description:	Scissoring enabled
Attribute group:	scissor/enable
Initial value:	GL_FALSE
Get command:	**glIsEnabled**

GL_SCISSOR_BOX

Description:	Scissor box
Attribute group:	scissor
Initial value:	–
Get command:	**glGetIntegerv**

GL_STENCIL_TEST

Description:	Stenciling enabled
Attribute group:	stencil-buffer/enable
Initial value:	GL_FALSE
Get command:	**glIsEnabled**

GL_STENCIL_FUNC

Description:	Stencil function
Attribute group:	stencil-buffer
Initial value:	GL_ALWAYS
Get command:	**glGetIntegerv**

GL_STENCIL_VALUE_MASK

Description:	Stencil mask
Attribute group:	stencil-buffer
Initial value:	1's
Get command:	**glGetIntegerv**

GL_STENCIL_REF

Description:	Stencil reference value
Attribute group:	stencil-buffer
Initial value:	0
Get command:	**glGetIntegerv**

GL_STENCIL_FAIL

Description:	Stencil fail action
Attribute group:	stencil-buffer
Initial value:	GL_KEEP
Get command:	**glGetIntegerv**

GL_STENCIL_PASS_DEPTH_FAIL

Description:	Stencil depth buffer fail action
Attribute group:	stencil-buffer
Initial value:	GL_KEEP
Get command:	**glGetIntegerv**

GL_STENCIL_PASS_DEPTH_PASS

Description:	Stencil depth buffer pass action
Attribute group:	stencil-buffer
Initial value:	GL_KEEP
Get command:	**glGetIntegerv**

GL_ALPHA_TEST

Description:	Alpha test enabled
Attribute group:	color-buffer/enable
Initial value:	GL_FALSE
Get command:	**glIsEnabled**

GL_ALPHA_TEST_FUNC

Description:	Alpha test function
Attribute group:	color-buffer
Initial value:	GL_ALWAYS
Get command:	**glGetIntegerv**

GL_ALPHA_TEST_REF

Description:	Alpha test reference value
Attribute group:	color-buffer
Initial value:	0
Get command:	**glGetIntegerv**

GL_DEPTH_TEST

Description:	Depth buffer enabled
Attribute group:	depth-buffer/enable
Initial value:	GL_FALSE
Get command:	**glIsEnabled**

GL_DEPTH_FUNC

Description:	Depth buffer test function
Attribute group:	depth-buffer
Initial value:	GL_LESS
Get command:	**glGetIntegerv**

GL_BLEND

Description:	Blending enabled
Attribute group:	color-buffer/enable
Initial value:	GL_FALSE
Get command:	**glIsEnabled**

GL_BLENC_SRC

Description:	Blending source function
Attribute group:	color-buffer
Initial value:	GL_ONE
Get command:	**glGetIntegerv**

GL_BLEND_DST

Description:	Blending destination function
Attribute group:	color-buffer

	Initial value:	GL_ZERO
	Get command:	**glGetIntegerv**

GL_LOGIC_OP

Description:	Logical operation enabled
Attribute group:	color-buffer/enable
Initial value:	GL_FALSE
Get command:	**glIsEnabled**

GL_LOGIC_OP_MODE

Description:	Logical operation function
Attribute group:	color-buffer
Initial value:	GL_COPY
Get command:	**glGetIntegerv**

GL_DITHER

Description:	Dithering enabled
Attribute group:	color-buffer/enable
Initial value:	GL_TRUE
Get command:	**glIsEnabled**

Framebuffer Control State Variables

GL_DRAW_BUFFER

Description:	Buffers selected for drawing
Attribute group:	color-buffer
Initial value:	–
Get command:	**glGetIntegerv**

GL_INDEX_WRITEMASK

Description:	Color-index writemask
Attribute group:	color-buffer
Initial value:	1's
Get command:	**glGetIntegerv**

GL_COLOR_WRITEMASK

Description:	Color write enables; R, G, B, or A
Attribute group:	color-buffer
Initial value:	GL_TRUE
Get command:	**glGetBooleanv**

GL_DEPTH_WRITEMASK

Description:	Depth buffer enabled for writing
Attribute group:	depth-buffer
Initial value:	GL_TRUE
Get command:	**glGetBooleanv**

GL_STENCIL_WRITEMASK

Description:	Stencil-buffer writemask
Attribute group:	stencil-buffer
Initial value:	1's
Get command:	**glGetIntegerv**

GL_COLOR_CLEAR_VALUE

Description:	Color-buffer clear value (RGBA mode)
Attribute group:	color-buffer
Initial value:	0, 0, 0, 0
Get command:	**glGetFloatv**

GL_INDEX_CLEAR_VALUE

Description:	Color-buffer clear value (color-index mode)
Attribute group:	color-buffer
Initial value:	0
Get command:	**glGetFloatv**

GL_DEPTH_CLEAR_VALUE

Description:	Depth-buffer clear value
Attribute group:	depth-buffer
Initial value:	1
Get command:	**glGetIntegerv**

GL_STENCIL_CLEAR_VALUE

Description:	Stencil-buffer clear value
Attribute group:	stencil-buffer
Initial value:	0
Get command:	**glGetIntegerv**

GL_ACCUM_CLEAR_VALUE

Description:	Accumulation-buffer clear value
Attribute group:	accum-buffer
Initial value:	0
Get command:	**glGetFloatv**

Pixel State Variables

GL_UNPACK_SWAP_BYTES

Description:	Value of GL_UNPACK_SWAP_BYTES
Attribute group:	–
Initial value:	GL_FALSE
Get command:	**glGetBooleanv**

GL_UNPACK_LSB_FIRST

Description:	Value of GL_UNPACK_LSB_FIRST
Attribute group:	–
Initial value:	GL_FALSE
Get command:	**glGetBooleanv**

GL_UNPACK_ROW_LENGTH

Description:	Value of GL_UNPACK_ROW_LENGTH
Attribute group:	–
Initial value:	0
Get command:	**glGetIntegerv**

GL_UNPACK_SKIP_ROWS

Description:	Value of GL_UNPACK_SKIP_ROWS
Attribute group:	–
Initial value:	0
Get command:	**glGetIntegerv**

GL_UNPACK_SKIP_PIXELS

Description:	Value of GL_UNPACK_SKIP_PIXELS
Attribute group:	–
Initial value:	0
Get command:	**glGetIntegerv**

GL_UNPACK_ALIGNMENT

Description:	Value of GL_UNPACK_ALIGNMENT
Attribute group:	–
Initial value:	4
Get command:	**glGetIntegerv**

GL_PACK_SWAP_BYTES

Description:	Value of GL_PACK_SWAP_BYTES
Attribute group:	–
Initial value:	GL_FALSE
Get command:	**glGetBooleanv**

GL_PACK_LSB_FIRST

Description:	Value of GL_PACK_LSB_FIRST
Attribute group:	–
Initial value:	GL_FALSE
Get command:	**glGetBooleanv**

GL_PACK_ROW_LENGTH

Description:	Value of GL_PACK_ROW_LENGTH
Attribute group:	–
Initial value:	0
Get command:	**glGetIntegerv**

GL_PACK_SKIP_ROWS

Description:	Value of GL_PACK_SKIP_ROWS
Attribute group:	–
Initial value:	0
Get command:	**glGetIntegerv**

GL_PACK_SKIP_PIXELS

Description:	Value of GL_PACK_SKIP_PIXELS
Attribute group:	–
Initial value:	0
Get command:	**glGetIntegerv**

GL_PACK_ALIGNMENT

Description:	Value of GL_PACK_ALIGNMENT
Attribute group:	–
Initial value:	4
Get command:	**glGetIntegerv**

GL_MAP_COLOR

Description:	True if colors are mapped
Attribute group:	pixel
Initial value:	GL_FALSE
Get command:	**glGetBooleanv**

GL_MAP_STENCIL

Description:	True if stencil values are mapped
Attribute group:	pixel
Initial value:	GL_FALSE
Get command:	**glGetBooleanv**

GL_INDEX_SHIFT

Description:	Value of GL_INDEX_SHIFT
Attribute group:	pixel
Initial value:	0
Get command:	**glGetIntegerv**

GL_INDEX_OFFSET

Description:	Value of GL_INDEX_OFFSET
Attribute group:	pixel
Initial value:	0
Get command:	**glGetIntegerv**

GL_*x*_SCALE

Description:	Value of GL_*x*_SCALE; *x* is GL_RED, GL_BLUE, GL_ALPHA, or GL_DEPTH

Attribute group: pixel
Initial value: 1
Get command: **glGetFloatv**

GL_*x*_BIAS

Description: Value of GL_*x*_BIAS; *x* is GL_RED GL_BLUE, GL_ALPHA, or GL_DEPTH
Attribute group: pixel
Initial value: 0
Get command: **glGetFloatv**

GL_ZOOM_X

Description: *x* zoom factor
Attribute group: pixel
Initial value: 1.0
Get command: **glGetFloatv**

GL_ZOOM_Y

Description: *y* zoom factor
Attribute group: pixel
Initial value: 1.0
Get command: **glGetFloatv**

GL_*x*

Description: **glPixelMap** translation tables
Attribute group: pixel
Initial value: 0's
Get command: **glGetPixelMap**

GL_*x*_SIZE

Description: Size of table *x*
Attribute group: pixel
Initial value: 1
Get command: **glGetIntegerv**

GL_READ_BUFFER

Description: Read source buffer
Attribute group: pixel
Initial value: –
Get command: **glGetIntegerv**

Evaluator State Variables

GL_ORDER

Description:	1-D map order
Attribute group:	–
Initial value:	1
Get command:	**glGetMapiv**

GL_ORDER

Description:	2-D map orders
Attribute group:	–
Initial value:	1, 1
Get command:	**glGetMapiv**

GL_COEFF

Description:	1-D control points
Attribute group:	–
Initial value:	–
Get command:	**glGetMapfv**

GL_COEFF

Description:	2-D control points
Attribute group:	–
Initial value:	–
Get command:	**glGetMapfv**

GL_DOMAIN

Description:	1-D domain endpoints
Attribute group:	–
Initial value:	–
Get command:	**glGetMapfv**

GL_DOMAIN

Description:	2-D domain endpoints
Attribute group:	–
Initial value:	–
Get command:	**glGetMapfv**

GL_MAP1_x

Description:	1-D map enables: *x* is map type
Attribute group:	eval/enable
Initial value:	GL_FALSE
Get command:	**glIsEnabled**

GL_MAP2_x

Description:	2-D map enables: *x* is map type
Attribute group:	eval/enable
Initial value:	GL_FALSE
Get command:	**glIsEnabled**

GL_MAP1_GRID_DOMAIN

Description:	1-D grid endpoints
Attribute group:	eval
Initial value:	0, 1
Get command:	**glGetFloatv**

GL_MAP2_GRID_DOMAIN

Description:	2-D grid endpoints
Attribute group:	eval
Initial value:	0, 1; 0, 1
Get command:	**glGetFloatv**

GL_MAP1_GRID_SEGMENTS

Description:	1-D grid divisions
Attribute group:	eval
Initial value:	1
Get command:	**glGetFloatv**

GL_MAP1_GRID_SEGMENTS

Description:	2-D grid segments
Attribute group:	eval
Initial value:	1, 1
Get command:	**glGetFloatv**

GL_AUTO_NORMAL

Description:	True if automatic normal generation enabled
Attribute group:	eval
Initial value:	GL_FALSE
Get command:	**glIsEnabled**

Hint State Variables

GL_PERSPECTIVE_CORRECTION_HINT

Description:	Perspective correction hint
Attribute group:	hint
Initial value:	GL_DON'T CARE
Get command:	**glGetIntegerv**

GL_POINT_SMOOTH_HINT

Description:	Point smooth hint
Attribute group:	hint
Initial value:	GL_DON'T CARE
Get command:	**glGetIntegerv**

GL_LINE_SMOOTH_HINT

Description:	Line smooth hint
Attribute group:	hint
Initial value:	GL_DON'T CARE
Get command:	**glGetIntegerv**

GL_POLYGON_SMOOTH_HINT

Description:	Polygon smooth hint
Attribute group:	hint
Initial value:	GL_DON'T CARE
Get command:	**glGetIntegerv**

GL_FOG_HINT

Description:	Fog hint
Attribute group:	hint
Initial value:	GL_DON'T CARE
Get command:	**glGetIntegerv**

Implementation-Dependent State Variables

GL_MAX_LIGHTS

Description:	Maximum number of lights
Attribute group:	–
Initial value:	8
Get command:	**glGetIntegerv**

GL_MAX_CLIP_PLANES

Description:	Maximum number of user clipping planes
Attribute group:	–
Initial value:	6
Get command:	**glGetIntegerv**

GL_MAX_MODELVIEW_STACK_DEPTH

Description:	Maximum modelview-matrix stack depth
Attribute group:	–
Initial value:	32
Get command:	**glGetIntegerv**

GL_MAX_PROJECTION_STACK_DEPTH

Description:	Maximum projection-matrix stack depth
Attribute group:	–
Initial value:	2
Get command:	**glGetIntegerv**

GL_MAX_MAX_TEXTURE_STACK_DEPTH

Description:	Maximum depth of texture matrix stack
Attribute group:	–
Initial value:	2
Get command:	**glGetIntegerv**

GL_SUBPIXEL_BITS

Description:	Number of bits of subpixel precision in *x* and *y*
Attribute group:	–
Initial value:	4
Get command:	**glGetIntegerv**

GL_MAX_TEXTURE_SIZE

Description:	Maximum height or width of a texture image (without borders)
Attribute group:	–
Initial value:	64
Get command:	**glGetIntegerv**

GL_MAX_PIXEL_MAP_TABLE

Description:	Maximum size of a **glPixelMap** translation table
Attribute group:	–

Initial value: 32
Get command: **glGetIntegerv**

GL_MAX_NAME_STACK_DEPTH
Description: Maximum selection-name stack depth
Attribute group: –
Initial value: 64
Get command: **glGetIntegerv**

GL_MAX_LIST_NESTING
Description: Maximum display-list call nesting
Attribute group: –
Initial value: 64
Get command: **glGetIntegerv**

GL_MAX_EVAL_ORDER
Description: Maximum evaluator polynomial order
Attribute group: –
Initial value: 8
Get command: **glGetIntegerv**

GL_MAX_VIEWPORT_DIMS
Description: Maximum viewport dimensions
Attribute group: –
Initial value: –
Get command: **glGetIntegerv**

GL_MAX_ATTRIB_STACK_DEPTH
Description: Maximum depth of the attribute stack
Attribute group: –
Initial value: 16
Get command: **glGetIntegerv**

GL_AUX_BUFFERS
Description: Number of auxiliary buffers
Attribute group: –
Initial value: 0
Get command: **glGetBooleanv**

GL_RGBA_MODE
Description: True if color buffers store RGBA
Attribute group: –
Initial value: –
Get command: **glGetBooleanv**

GL_INDEX_MODE
Description: True if color buffers store indexes
Attribute group: –
Initial value: –
Get command: **glGetBooleanv**

GL_DOUBLEBUFFER

Description:	True if front and back buffers exist
Attribute group:	–
Initial value:	–
Get command:	**glGetBooleanv**

GL_STEREO

Description:	True if left and right buffers exist
Attribute group:	–
Initial value:	–
Get command:	**glGetFloatv**

GL_POINT_SIZE_RANGE

Description:	Range (low to high) of antialiased point sizes
Attribute group:	–
Initial value:	1, 1
Get command:	**glGetFloatv**

GL_POINT_SIZE_GRANULARITY

Description:	Antialiased point size granularity
Attribute group:	–
Initial value:	–
Get command:	**glGetFloatv**

GL_LINE_WIDTH_RANGE

Description:	Range (low to high) of antialiased line widths
Attribute group:	–
Initial value:	1, 1
Get command:	**glGetFloatv**

GL_LINE_WIDTH_GRANULARITY

Description:	Antialiased line-width granularity
Attribute group:	–
Initial value:	–
Get command:	**glGetFloatv**

Implementation-Dependent Pixel-Depth State Variables

GL_RED_BITS

Description:	Number of bits per red component in color buffers
Attribute group:	–
Initial value:	–
Get command:	**glGetIntegerv**

GL_GREEN_BITS

Description:	Number of bits per green component in color buffers
Attribute group:	–
Get command:	**glGetIntegerv**
Initial value:	–

GL_BLUE_BITS

Description:	Number of bits per blue component in color buffers
Attribute group:	–
Initial value:	–
Get command:	**glGetIntegerv**

GL_ALPHA_BITS

Description:	Number of bits per alpha component in color buffers
Attribute group:	–
Initial value:	–
Get command:	**glGetIntegerv**

GL_INDEX_BITS

Description:	Number of bits per index in color buffers
Attribute group:	–
Initial value:	–
Get command:	**glGetIntegerv**

GL_DEPTH_BITS

Description:	Number of depth-buffer bitplanes
Attribute group:	–
Initial value:	–
Get command:	**glGetIntegerv**

GL_STENCIL_BITS

Description:	Number of stencil bitplanes
Attribute group:	–
Initial value:	–
Get command:	**glGetIntegerv**

GL_ACCUM_RED_BITS

Description:	Number of bits per red component in the accumulation buffer

Attribute group: –
Initial value: –
Get command: **glGetIntegerv**

GL_ACCUM_GREEN_BITS
Description: Number of bits per green component in the accumulation buffer
Attribute group: –
Initial value: –
Get command: **glGetIntegerv**

GL_ACCUM_BLUE_BITS
Description: Number of bits per blue component in the accumulation buffer
Attribute group: –
Initial value: –
Get command: **glGetIntegerv**

GL_ACCUM_ALPHA_BITS
Description: Number of bits per alpha component in the accumulation buffer
Attribute group: –
Initial value: –
Get command: **glGetIntegerv**

Miscellaneous State Variables

GL_LIST_BASE

Description:	Setting of **glListBase**
Attribute group:	list
Initial value:	0
Get command:	**glGetIntegerv**

GL_LIST_INDEX

Description:	Number of display lists under construction; 0 if none
Attribute group:	–
Initial value:	0
Get command:	**glGetIntegerv**

GL_LIST_MODE

Description:	Mode of display list under construction; undefined if none
Attribute group:	–
Initial value:	0
Get command:	**glGetIntegerv**

GL_ATTRIB_STACK_DEPTH

Description:	Attribute stack pointer
Attribute group:	–
Initial value:	0
Get command:	**glGetIntegerv**

GL_NAME_STACK_DEPTH

Description:	Name stack depth
Attribute group:	–
Initial value:	0
Get command:	**glGetIntegerv**

GL_RENDER_MODE

Description:	**glRenderMode** setting
Attribute group:	–
Initial value:	GL_RENDER
Get command:	**glGetIntegerv**

–

Description:	Current error code(s)
Attribute group:	–
Initial value:	0
Get command:	**glGetError**

glAccum Quick Info

[New - Windows 95, OEM Service Release 2]

The **glAccum** function operates on the accumulation buffer.

```
void glAccum(
  GLenum op,
  GLfloat value
);
```

Parameters

op

The accumulation buffer operation. The accepted symbolic constants are:

GL_ACCUM
: Obtains R, G, B, and A values from the buffer currently selected for reading (see **glReadBuffer**). Each component value is divided by $2^n - 1$, where *n* is the number of bits allocated to each color component in the currently selected buffer. The result is a floating-point value in the range [0,1], which is multiplied by *value* and added to the corresponding pixel component in the accumulation buffer, thereby updating the accumulation buffer.

GL_LOAD
: Similar to GL_ACCUM, except that the current value in the accumulation buffer is not used in the calculation of the new value. That is, the R, G, B, and A values from the currently selected buffer are divided by $2^n - 1$, multiplied by *value*, and then stored in the corresponding accumulation buffer cell, overwriting the current value.

GL_ADD
: Adds *value* to each R, G, B, and A in the accumulation buffer.

GL_MULT
: Multiplies each R, G, B, and A in the accumulation buffer by *value* and returns the scaled component to its corresponding accumulation buffer location.

GL_RETURN
: Transfers accumulation buffer values to the color buffer or buffers currently selected for writing. Each R, G, B, and A component is multiplied by *value*, then multiplied by $2^n - 1$, clamped to the range [0, $2^n - 1$], and stored in the corresponding display buffer cell. The only fragment operations that are applied to this transfer are pixel ownership, scissor, dithering, and color writemasks.

value

A floating-point value used in the accumulation buffer operation. The *op* parameter determines how *value* is used.

Remarks

The accumulation buffer is an extended-range color buffer. Images are not rendered into it. Rather, images rendered into one of the color buffers are added to the contents of the accumulation buffer after rendering. You can create effects such as antialiasing (of points, lines, and polygons), motion blur, and depth of field by accumulating images generated with different transformation matrices.

Each pixel in the accumulation buffer consists of red, green, blue, and alpha values. The number of bits per component in the accumulation buffer depends on the implementation. You can examine this number by calling **glGetIntegerv** four times, with the arguments GL_ACCUM_RED_BITS, GL_ACCUM_GREEN_BITS, GL_ACCUM_BLUE_BITS, and GL_ACCUM_ALPHA_BITS, respectively. Regardless of the number of bits per component, however, the range of values stored by each component is [- 1, 1]. The accumulation buffer pixels are mapped one-to-one with frame buffer pixels.

The **glAccum** function operates on the accumulation buffer. The first argument, *op*, is a symbolic constant that selects an accumulation buffer operation. The second argument, *value*, is a floating-point

value to be used in that operation. Five operations are specified: GL_ACCUM, GL_LOAD, GL_ADD, GL_MULT, and GL_RETURN.

All accumulation buffer operations are limited to the area of the current scissor box and are applied identically to the red, green, blue, and alpha components of each pixel. The contents of an accumulation buffer pixel component are undefined if the **glAccum** operation results in a value outside the range [-1,1]. The operations are as follows:

To clear the accumulation buffer, use the **glClearAccum** function to specify R, G, B, and A values to set it to, and issue a **glClear** function with the accumulation buffer enabled.

Only those pixels within the current scissor box are updated by any **glAccum** operation.

The following functions retrieve information related to the **glAccum** function:

glGet with argument GL_ACCUM_RED_BITS
glGet with argument GL_ACCUM_GREEN_BITS
glGet with argument GL_ACCUM_BLUE_BITS
glGet with argument GL_ACCUM_ALPHA_BITS

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*op* was not an accepted value.
GL_INVALID_OPERATION	There was no accumulation buffer.
GL_INVALID_OPERATION	**glAccum** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glBlendFunc**, **glClear**, **glClearAccum**, **glCopyPixels**, **glEnd**, **glGet**, **glLogicOp**, **glPixelStore**, **glPixelTransfer**, **glReadBuffer**, **glReadPixels**, **glScissor**, **glStencilOp**

glAddSwapHintRectWIN Quick Info

[New - Windows 95, OEM Service Release 2]

The **glAddSwapHintRectWIN** function specifies a set of rectangles that are to be copied by **SwapBuffers**.

```
void glAddSwapHintRectWIN(
  GLint x,
  GLint y,
  GLsizei width,
  GLsizei height
);
```

Parameters

x
The *x*-coordinate (in window coordinates) of the lower-left corner of the hint region rectangle.

y
The *y*-coordinate (in window coordinates) of the lower-left corner of the hint region rectangle.

width
The width of the hint region rectangle.

height
The height of the hint region rectangle.

Remarks

The **glAddSwapHintRectWIN** function speeds up animation by reducing the amount of repainting between frames. With **glAddSwapHintRectWIN**, you specify a set of rectangular areas that you want copied when you call **SwapBuffers**. When you do not specify any rectangles with **glAddSwapHintRectWIN** before calling **SwapBuffers**, the entire frame buffer is swapped. Using **glAddSwapHintRectWIN** to copy only parts of the buffer that changed can significantly increase the performance of **SwapBuffers**, especially when **SwapBuffers** is implemented in software.

The **glAddSwapHintRectWIN** function adds a rectangle to the hint region. When the PFD_SWAP_COPY flag of the **PIXELFORMATDESCRIPTOR** pixel format structure is set, **SwapBuffers** uses this region to clip the copying of the back buffer to the front buffer. You don't specify PFD_SWAP_COPY; it is set by capable hardware. The hint region is cleared after each call to **SwapBuffers**. With some hardware configurations, **SwapBuffers** can ignore the hint region and exchange the entire buffer. **SwapBuffers** is implemented by the system, not by the application.

OpenGL maintains a separate hint region for each window. When you call **glAddSwapHintRectWIN** on any rendering contexts associated with a window, the hint rectangles are combined into a single region.

Call **glAddSwapHintRectWIN** with a bounding rectangle for each object drawn for a frame and for each rectangle cleared to erase previous frame objects.

Note The **glAddSwapHintRectWIN** function is an extension function that is not part of the standard OpenGL library but is part of the GL_WIN_swap_hint extension. To check whether your implementation of OpenGL supports **glAddSwapHintRectWIN**, call **glGetString(**GL_EXTENSIONS**)**. If it returns GL_WIN_swap_hint, **glAddSwapHintRectWIN** is supported. To obtain the address of an extension function, call **wglGetProcAddress**.

See Also

glGetString, **PIXELFORMATDESCRIPTOR**, **SwapBuffers**, **wglGetProcAddress**

glAlphaFunc Quick Info

[New - Windows 95, OEM Service Release 2]

The **glAlphaFunc** function specifies the alpha test function.

```
void glAlphaFunc(
  GLenum func,
  GLclampf ref
);
```

Parameters

func

The alpha comparison function. The following are the accepted symbolic constants and their meanings.

Symbolic Constant	Meaning
GL_NEVER	Never passes.
GL_LESS	Passes if the incoming alpha value is less than the reference value.
GL_EQUAL	Passes if the incoming alpha value is equal to the reference value.
GL_LEQUAL	Passes if the incoming alpha value is less than or equal to the reference value.
GL_GREATER	Passes if the incoming alpha value is greater than the reference value.
GL_NOTEQUAL	Passes if the incoming alpha value is not equal to the reference value.
GL_GEQUAL	Passes if the incoming alpha value is greater than or equal to the reference value.
GL_ALWAYS	Always passes. This is the default.

ref

The reference value to which incoming alpha values are compared. This value is clamped to the range 0 through 1, where 0 represents the lowest possible alpha value and 1 the highest possible value. The default reference is 0.

Remarks

The alpha test discards fragments depending on the outcome of a comparison between the incoming fragments' alpha values and a constant reference value. The **glAlphaFunc** function specifies the reference and comparison function. The comparison is performed only if alpha testing is enabled. (For more information on GL_ALPHA_TEST, see **glEnable**.)

The *func* and *ref* parameters specify the conditions under which the pixel is drawn. The incoming alpha value is compared to *ref* using the function specified by *func*. If the comparison passes, the incoming fragment is drawn, conditional on subsequent stencil and depth-buffer tests. If the comparison fails, no change is made to the frame buffer at that pixel location.

The **glAlphaFunc** function operates on all pixel writes, including those resulting from the scan conversion of points, lines, polygons, and bitmaps, and from pixel draw and copy operations. The **glAlphaFunc** function does not affect screen clear operations.

Alpha testing is done only in RGBA mode.

The following functions retrieve information related to the **glAlphaFunc** function:

glGet with argument GL_ALPHA_TEST_FUNC
glGet with argument GL_ALPHA_TEST_REF
glIsEnabled with argument GL_ALPHA_TEST

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*func* was not an accepted value.
GL_INVALID_OPERATION	**glAlphaFunc** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glBlendFunc**, **glClear**, **glDepthFunc**, **glEnable**, **glEnd**, **glGet**, **glIsEnabled**, **glStencilFunc**

glAreTexturesResident

[New - Windows 95, OEM Service Release 2]

The **glAreTexturesResident** function determines whether specified texture objects are resident.

```
GLboolean glAreTexturesResident(
  GLsizei n,
  GLuint * textures
  GLboolean *
residences
  );
```

Parameters

n
The number of textures to be queried.

textures
The address of an array containing the names of the textures to be queried.

residences
The address of an array in which the texture residence status is returned. The residence status of a texture named by an element of *textures* is returned in the corresponding element of *residences*.

Remarks

On machines with a limited amount of texture memory, OpenGL establishes a ``working set'' of textures that are resident in texture memory. These textures can be bound to a texture target much more efficiently than textures that are not resident.

The **glAreTexturesResident** function queries the texture residence status of the *n* textures named by the elements of *textures*. If all the named textures are resident, **glAreTexturesResident** returns GL_TRUE, and the contents of *residences* are undisturbed. If any of the named textures are not resident, **glAreTexturesResident** returns GL_FALSE, and detailed status is returned in the *n* elements of *residences*.

If an element of *residences* is GL_TRUE, then the texture named by the corresponding element of *textures* is resident.

To query the residence status of a single bound texture, call **glGetTexParameter** with the *target* parameter set to the target texture to which the target is bound and set the *pname* parameter to GL_TEXTURE_RESIDENT. You must use this method to query the resident status of a default texture.

You cannot include **glAreTexturesResident** in display lists.

Note The **glAreTexturesResident** function is only available in OpenGL version 1.1 or later.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_VALUE	*n* was a negative value.
GL_INVALID_VALUE	An element in *textures* was zero or did not contain a texture name.
GL_INVALID_OPERATION	**glAreTexturesResident** was called between a call to **glBegin** and the

corresponding call to **glEnd**.

See Also

glBegin, **glBindTexture**, **glEnd**, **glGetTexParameter**, **glPrioritizeTextures**, **glTexImage1D**, **glTexImage2D**

glArrayElement

[New - Windows 95, OEM Service Release 2]

The **glArrayElement** function specifies the array elements used to render a vertex.

```
void glArrayElement(
  GLint index
);
```

Parameters

index
An index in the enabled arrays.

Remarks

Use the **glArrayElement** function within **glBegin** and **glEnd** pairs to specify vertex and attribute data for point, line, and polygon primitives. The **glArrayElement** function specifies the data for a single vertex using vertex and attribute data located at the *index* of the enabled vertex arrays.

You can use **glArrayElement** to construct primitives by indexing vertex data, rather than by streaming through arrays of data in first-to-last order. Because **glArrayElement** specifies a single vertex only, you can explicitly specify attributes for individual primitives. For example, you can set a single normal for each individual triangle.

When you include calls to **glArrayElement** in display lists, the necessary array data, determined by the array pointers and enable values, is entered in the display list also. Array pointer and enable values are determined when display lists are created, not when display lists are executed.

You can read and cache static array data at any time with **glArrayElement**. When you modify the elements of a static array without specifying the array again, the results of any subsequent calls to **glArrayElement** are undefined.

When you call **glArrayElement** without first calling **glEnableClientState(**GL_VERTEX_ARRAY**)**, no drawing occurs, but the attributes corresponding to enabled arrays are modified. Although no error is generated when you specify an array within **glBegin** and **glEnd** pairs, the results are undefined.

See Also

glBegin, **glColorPointer**, **glDrawArrays**, **glEdgeFlagPointer**, **glEnableClientState**, **glEnd**, **glGetPointerv**, **glGetString**, **glIndexPointer**, **glNormalPointer**, **glTexCoordPointer**, **glVertexPointer**

glBegin, glEnd

[New - Windows 95, OEM Service Release 2]

The **glBegin** and **glEnd** functions delimit the vertices of a primitive or a group of like primitives.

```
void glBegin(
  GLenum mode
);

void glEnd(
  void
);
```

Parameters

mode

The primitive or primitives that will be created from vertices presented between **glBegin** and the subsequent **glEnd**. The following are accepted symbolic constants and their meanings:

GL_POINTS
: Treats each vertex as a single point. Vertex *n* defines point *n*. *N* points are drawn.

GL_LINES
: Treats each pair of vertices as an independent line segment. Vertices *2n - 1* and *2n* define line *n*. *N/2* lines are drawn.

GL_LINE_STRIP
: Draws a connected group of line segments from the first vertex to the last. Vertices *n* and *n+1* define line *n*. *N - 1* lines are drawn.

GL_LINE_LOOP
: Draws a connected group of line segments from the first vertex to the last, then back to the first. Vertices *n* and *n+1* define line *n*. The last line, however, is defined by vertices *N* and *1*. *N* lines are drawn.

GL_TRIANGLES
: Treats each triplet of vertices as an independent triangle. Vertices *3n - 2*, *3n -1*, and *3n* define triangle *n*. *N/3* triangles are drawn.

GL_TRIANGLE_STRIP
: Draws a connected group of triangles. One triangle is defined for each vertex presented after the first two vertices. For odd *n*, vertices *n*, *n + 1*, and *n + 2* define triangle *n*. For even *n*, vertices *n + 1*, *n*, and *n + 2* define triangle *n*. *N - 2* triangles are drawn.

GL_TRIANGLE_FAN
: Draws a connected group of triangles. One triangle is defined for each vertex presented after the first two vertices. Vertices *1*, *n + 1*, and *n + 2* define triangle *n*. *N - 2* triangles are drawn.

GL_QUADS
: Treats each group of four vertices as an independent quadrilateral. Vertices *4n - 3*, *4n - 2*, *4n - 1*, and *4n* define quadrilateral *n*. *N/4* quadrilaterals are drawn.

GL_QUAD_STRIP
: Draws a connected group of quadrilaterals. One quadrilateral is defined for each pair of vertices presented after the first pair. Vertices *2n - 1*, *2n*, *2n + 2*, and *2n + 1* define quadrilateral *n*. *N* quadrilaterals are drawn. Note that the order in which vertices are used to construct a quadrilateral from strip data is different from that used with independent data.

GL_POLYGON
: Draws a single, convex polygon. Vertices *1* through *N* define this polygon.

Remarks

The **glBegin** and **glEnd** functions delimit the vertices that define a primitive or a group of like primitives.

The **glBegin** function accepts a single argument that specifies which of ten ways the vertices are interpreted. Taking *n* as an integer count starting at one, and *N* as the total number of vertices specified, the interpretations are as follows:

- You can use only a subset of OpenGL functions between **glBegin** and **glEnd**. The functions you can use are:

  glVertex
  glColor
  glIndex
  glNormal
  glTexCoord
  glEvalCoord
  glEvalPoint
  glMaterial
  glEdgeFlag

  You can also use **glCallList** or **glCallLists** to execute display lists that include only the preceding functions. If any other OpenGL function is called between **glBegin** and **glEnd**, the error flag is set and the function is ignored.
- Regardless of the value chosen for *mode* in **glBegin**, there is no limit to the number of vertices you can define between **glBegin** and **glEnd**. Lines, triangles, quadrilaterals, and polygons that are incompletely specified are not drawn. Incomplete specification results when either too few vertices are provided to specify even a single primitive or when an incorrect multiple of vertices is specified. The incomplete primitive is ignored; the complete primitives are drawn.
- The minimum specification of vertices for each primitive is:

Minimum Number of Vertices	Type of Primitive
1	point
2	line
3	triangle
4	quadrilateral
3	polygon

  Modes that require a certain multiple of vertices are GL_LINES (2), GL_TRIANGLES (3), GL_QUADS (4), and GL_QUAD_STRIP (2).

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*mode* was set to an unaccepted value.
GL_INVALID_OPERATION	A function other than **glVertex**, **glColor**, **glIndex**, **glNormal**, **glTexCoord**, **glEvalCoord**, **glEvalPoint**, **glMaterial**, **glEdgeFlag**, **glCallList**, or **glCallLists** was called between **glBegin** and the corresponding **glEnd**.
GL_INVALID_OPERATION	**glEnd** was called before the corresponding **glBegin** was called, or **glBegin** was called within a

glBegin/**glEnd** sequence.

See Also

glCallList, **glCallLists**, **glColor**, **glEdgeFlag**, **glEvalCoord**, **glEvalPoint**, **glIndex**, **glMaterial**, **glNormal**, **glTexCoord**, **glVertex**

glBindTexture

[New - Windows 95, OEM Service Release 2]

The **glBindTexture** function enables the creation of a named texture that is bound to a texture target.

```
void glBindTexture(
  GLenum target,
  GLuint texture
);
```

Parameters

target
The target to which the texture is bound. Must have the value GL_TEXTURE_1D or GL_TEXTURE_2D.

texture
The name of a texture; the texture name cannot currently be in use.

Remarks

The **glBindTexture** function enables you to create a named texture. Calling **glBindTexture** with *target* set to GL_TEXTURE_1D or GL_TEXTURE_2D, and *texture* set to the name of the new texture you have created binds the texture name to the appropriate texture target. When a texture is bound to a target, the previous binding for that target is no longer in effect.

Texture names are unsigned integers with the value zero reserved to represent the default texture for each texture target. Texture names and the corresponding texture contents are local to the shared display-list space of the current OpenGL rendering context; two rendering contexts share texture names only if they also share display lists. You can generate a set of new texture names using **glGenTextures**.

When a texture is first bound, it assumes the dimensionality of its texture target; a texture bound to GL_TEXTURE_1D becomes one-dimensional and a texture bound to GL_TEXTURE_2D becomes two-dimensional. Operations you perform on a texture target also affect a texture bound to the target. When you query a texture target, the return value is the state of the texture bound to it. Texture targets become aliases for textures currently bound to them.

When you bind a texture with **glBindTexture**, the binding remains active until a different texture is bound to the same target or you delete the bound texture with the **glDeleteTextures** function. Once you create a named texture you can bind it to a texture target that has the same dimensionality as often as needed.

It is usually much faster to use **glBindTexture** to bind an existing named texture to one of the texture targets than it is to reload the texture image using **glTexImage1D** or **glTexImage2D**. For additional control of texturing performance, use **glPrioritizeTextures**.

You can include calls to **glBindTexture** in display lists.

Note The **glBindTexture** function is only available in OpenGL version 1.1 or later.

The following functions retrieve information related to **glBindTexture**:

glGet with argument GL_TEXTURE_1D_BINDING

glGet with argument GL_TEXTURE_2D_BINDING

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*target* was not an accepted value.
GL_INVALID_OPERATION	*texture* did not have the same dimensionality as *target*.
GL_INVALID_OPERATION	**glBindTexture** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glAreTexturesResident, **glDeleteTextures**, **glGenTextures**, **glGet**, **glGetTexParameter**, **glIsTexture**, **glPrioritizeTextures**, **glTexImage1D**, **glTexImage2D**, **glTexParameter**

glBitmap Quick Info

[New - Windows 95, OEM Service Release 2]

The **glBitmap** function draws a bitmap.

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

Parameters

width, height
The pixel width and height of the bitmap image.

xorig, yorig
The location of the origin in the bitmap image. The origin is measured from the lower-left corner of the bitmap, with right and up directions being the positive axes.

xmove, ymove
The *x* and *y* offsets to be added to the current raster position after the bitmap is drawn.

bitmap
The address of the bitmap image.

Remarks

A bitmap is a binary image. When drawn, the bitmap is positioned relative to the current raster position, and frame buffer pixels corresponding to 1s in the bitmap are written using the current raster color or index. Frame-buffer pixels corresponding to zeros in the bitmap are not modified.

The bitmap image is interpreted like image data for the **glDrawPixels** function, with *width* and *height* corresponding to the width and height arguments of that function, and with *type* set to GL_BITMAP and *format* set to GL_COLOR_INDEX. Modes you specify using **glPixelStore** affect the interpretation of bitmap image data; modes you specify using **glPixelTransfer** do not.

If the current raster position is invalid, **glBitmap** is ignored. Otherwise, the lower-left corner of the bitmap image is positioned at the following window coordinates:

$$x_{(w)} = \lfloor x_{(r)} - x_{(o)} \rfloor$$
$$y_{(w)} = \lfloor y_{(r)} - y_{(o)} \rfloor$$

In these coordinates, (*xr, yr)* is the raster position, and (*xo, yo*) is the bitmap origin. Fragments are then generated for each pixel corresponding to a 1 in the bitmap image. These fragments are generated using the current raster *z*-coordinate, color or color index, and current raster texture coordinates. They are then treated just as if they had been generated by a point, line, or polygon, including texture mapping, fogging, and all per-fragment operations such as alpha and depth testing.

After the bitmap has been drawn, the *x* and *y* coordinates of the current raster position are offset by *xmove* and *ymove*. No change is made to the *z*-coordinate of the current raster position, or to the current raster color, index, or texture coordinates.

The following functions retrieve information related to the **glBitmap** function:

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_COLOR
glGet with argument GL_CURRENT_RASTER_INDEX
glGet with argument GL_CURRENT_RASTER_TEXTURE_COORDS
glGet with argument GL_CURRENT_RASTER_POSITION_VALID

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_VALUE	*width* or *height* is negative.
GL_INVALID_OPERATION	**glBitmap** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glDrawPixels**, **glEnd**, **glPixelStore**, **glPixelTransfer**, **glRasterPos**

glBlendFunc Quick Info

[New - Windows 95, OEM Service Release 2]

The **glBlendFunc** function specifies pixel arithmetic.

```
void glBlendFunc(
  GLenum sfactor,
  GLenum dfactor
);
```

Parameters

sfactor

Specifies how the red, green, blue, and alpha source-blending factors are computed. Nine symbolic constants are accepted: GL_ZERO, GL_ONE, GL_DST_COLOR, GL_ONE_MINUS_DST_COLOR, GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA, and GL_SRC_ALPHA_SATURATE.

dfactor

Specifies how the red, green, blue, and alpha destination-blending factors are computed. Eight symbolic constants are accepted: GL_ZERO, GL_ONE, GL_SRC_COLOR, GL_ONE_MINUS_SRC_COLOR, GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_DST_ALPHA, and GL_ONE_MINUS_DST_ALPHA.

Remarks

In RGB mode, pixels can be drawn using a function that blends the incoming (source) RGBA values with the RGBA values that are already in the frame buffer (the destination values). By default, blending is disabled. Use **glEnable** and **glDisable** with the GL_BLEND argument to enable and disable blending.

When enabled, **glBlendFunc** defines the operation of blending. The *sfactor* parameter specifies which of nine methods is used to scale the source color components. The *dfactor* parameter specifies which of eight methods is used to scale the destination color components. The eleven possible methods are described in the following table. Each method defines four scale factors, one each for red, green, blue, and alpha.

In the table and in subsequent equations, source and destination color components are referred to as ($R_{(s)}, G_{(s)}, B_{(s)}, A_{(s)}$) and ($R_{(d)}, G_{(d)}, B_{(d)}, A_{(d)}$). They are understood to have integer values between zero and ($k_{(R)}, k_{(G)}, k_{(B)}, k_{(A)}$), where

$$k_{(c)} = 2^{m_{(c)}} - 1$$

and ($m_{(R)}, m_{(G)}, m_{(B)}, m_{(A)}$) is the number of red, green, blue, and alpha bitplanes.

Source and destination scale factors are referred to as ($s_{(R)}, s_{(G)}, s_{(B)}, s_{(A)}$) and ($d_{(R)}, d_{(G)}, d_{(B)}, d_{(A)}$). The scale factors described in the table, denoted ($f_{(R)}, f_{(G)}, f_{(B)}, f_{(A)}$), represent either source or destination factors. All scale factors have range [0,1].

Parameter	***(f(R), f(G), f(B), f(A))***
GL_ZERO	(0,0,0,0)
GL_ONE	(1,1,1,1)
GL_SRC_COLOR	($R_{(s)}/k_{(R)}, G_{(s)}/k_{(G)}, B_{(s)}/k_{(B)}, A_{(s)}/k_{(A)}$)
GL_ONE_MINUS_SRC_COLOR	(1,1,1,1). ($R_{(s)}/k_{(R)}, G_{(s)}/k_{(G)}, B_{(s)}/k_{(B)}, A_{(s)}/k_{(A)}$)

GL_DST_COLOR	$(R_{(d)}/k_{(R)}, G_{(d)}/k_{(G)}, B_{(d)}/k_{(B)}, A_{(d)}/k_{(A)})$
GL_ONE_MINUS_DST_COLOR	(1,1,1,1)
GL_SRC_ALPHA	$(R_{(d)}/k_{(R)}, G_{(d)}/k_{(G)}, B_{(d)}/k_{(B)}, A_{(d)}/k_{(A)})$ $(A_{(s)}/k_{(A)}, A_{(s)}/k_{(A)}, A_{(s)}/k_{(A)}, A_{(s)}/k_{(A)})$
GL_ONE_MINUS_SRC_ALPHA	(1,1,1,1) $(A_{(s)}/k_{(A)}, A_{(s)}/k_{(A)}, A_{(s)}/k_{(A)}, A_{(s)}/k_{(A)})$
GL_DST_ALPHA	$(A_{(d)}/k_{(A)}, A_{(d)}/k_{(A)}, A_{(d)}/k_{(A)}, A_{(d)}/k_{(A)})$
GL_ONE_MINUS_DST_ALPHA	(1,1,1,1) $(A_{(d)}/k_{(A)}, A_{(d)}/k_{(A)}, A_{(d)}/k_{(A)}, A_{(d)}/k_{(A)})$
GL_SRC_ALPHA_SATURATE	(*i*,*i*,*i*,1)

In the table,

$i = \min(A_{(s)}, k_{(A)} - A_{(d)}) / kA$

To determine the blended RGBA values of a pixel when drawing in RGB mode, the system uses the following equations:

$R_{(d)} = \min(kR, RssR+RddR)$
$G_{(d)} = \min(kG, GssG+GddG)$
$B_{(d)} = \min(kB, BssB+BddB)$
$A_{(d)} = \min(kA, AssA+AddA)$

Despite the apparent precision of the above equations, blending arithmetic is not exactly specified, because blending operates with imprecise integer color values. However, a blend factor that should be equal to one is guaranteed not to modify its multiplicand, and a blend factor equal to zero reduces its multiplicand to zero. Thus, for example, when *sfactor* is GL_SRC_ALPHA, *dfactor* is GL_ONE_MINUS_SRC_ALPHA, and $A_{(s)}$ is equal to $k_{(A)}$, the equations reduce to simple replacement:

$R_{(d)} = R_{(s)}$
$G_{(d)} = G_{(s)}$
$B_{(d)} = B_{(s)}$
$A_{(d)} = A_{(s}$
)

Examples

Transparency is best implemented using **glBlendFunc(**GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA**)** with primitives sorted from farthest to nearest. Note that this transparency calculation does not require the presence of alpha bitplanes in the frame buffer.

You can also use **glBlendFunc(**GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA**)** for rendering antialiased points and lines in arbitrary order.

To optimize polygon antialiasing, use **glBlendFunc(**GL_SRC_ALPHA_SATURATE, GL_ONE**)** with polygons sorted from nearest to farthest. (See the GL_POLYGON_SMOOTH argument in **glEnable** for information on polygon antialiasing.) Destination alpha bitplanes, which must be present for this blend function to operate correctly, store the accumulated coverage.

Incoming (source) alpha is a material opacity, ranging from 1.0 ($K_{(A)}$), representing complete opacity, to

0.0 (0), representing complete transparency.

When you enable more than one color buffer for drawing, each enabled buffer is blended separately, and the contents of the buffer is used for destination color. (See **glDrawBuffer**.)

Blending affects only RGB rendering. It is ignored by color-index renderers.

The following functions retrieve information related to **glBlendFunc**:

glGet with argument GL_BLEND_SRC
glGet with argument GL_BLEND_DST
glIsEnabled with argument GL_BLEND

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	Either *sfactor* or *dfactor* was not an accepted value.
GL_INVALID_OPERATION	**glBlendFunc** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glAlphaFunc, **glBegin**, **glClear**, **glDisable**, **glDrawBuffer**, **glEnable**, **glGet**, **glIsEnabled**, **glLogicOp**, **glStencilFunc**

glCallList Quick Info

[New - Windows 95, OEM Service Release 2]

The **glCallList** function executes a display list.

```
void glCallList(
  GLuint list
);
```

Parameters

list

The integer name of the display list to be executed.

Remarks

You start execution of the named display list with **glCallList**. The functions saved in the display list are executed in order, just as if you called them without using a display list. If *list* has not been defined as a display list, **glCallList** is ignored.

The **glCallList** function can appear inside a display list. To avoid the possibility of infinite recursion resulting from display lists calling one another, a limit is placed on the nesting level of display lists during display-list execution. This limit is at least 64, and it depends on the implementation.

The OpenGL state is not saved and restored across a call to **glCallList**. Thus, changes made to the OpenGL state during the execution of a display list remain after execution of the display list is completed. To preserve the OpenGL state across **glCallList** calls, use **glPushAttrib**, **glPopAttrib**, **glPushMatrix**, and **glPopMatrix**.

You can execute display lists between a call to **glBegin** and the corresponding call to **glEnd**, as long as the display list includes only functions that are allowed in this interval.

The following functions retrieve information related to **glCallList**:

glGet with argument GL_MAX_LIST_NESTING

glIsList

See Also

glBegin, **glCallLists**, **glDeleteLists**, **glEnd**, **glGenLists**, **glGet**, **glIsList**, **glNewList**, **glPopAttrib**, **glPopMatrix**, **glPushAttrib**, **glPushMatrix**

glCallLists Quick Info

[New - Windows 95, OEM Service Release 2]

The **glCallLists** function executes a list of display lists.

```
void glCallLists(
  GLsizei n,
  GLenum type,
  const GLvoid * lists
);
```

Parameters

n

The number of display lists to be executed.

type

The type of values in *lists*. The following symbolic constants are accepted:

GL_BYTE
The *lists* parameter is treated as an array of signed bytes, each in the range - 128 through 127.

GL_UNSIGNED_BYTE
The *lists* parameter is treated as an array of unsigned bytes, each in the range 0 through 255.

GL_SHORT
The *lists* parameter is treated as an array of signed two-byte integers, each in the range - 32768 through 32767.

GL_UNSIGNED_SHORT
The *lists* parameter is treated as an array of unsigned two-byte integers, each in the range 0 through 65535.

GL_INT
The *lists* parameter is treated as an array of signed four-byte integers.

GL_UNSIGNED_INT
The *lists* parameter is treated as an array of unsigned four-byte integers.

GL_FLOAT
The *lists* parameter is treated as an array of four-byte, floating-point values.

GL_2_BYTES
The *lists* parameter is treated as an array of unsigned bytes. Each pair of bytes specifies a single display-list name. The value of the pair is computed as 256 times the unsigned value of the first byte plus the unsigned value of the second byte.

GL_3_BYTES
The *lists* parameter is treated as an array of unsigned bytes. Each triplet of bytes specifies a single display list name. The value of the triplet is computed as 65536 times the unsigned value of the first byte, plus 256 times the unsigned value of the second byte, plus the unsigned value of the third byte.

GL_4_BYTES
The *lists* parameter is treated as an array of unsigned bytes. Each quadruplet of bytes specifies a single display list name. The value of the quadruplet is computed as 16777216 times the unsigned value of the first byte, plus 65536 times the unsigned value of the second byte, plus 256 times the unsigned value of the third byte, plus the unsigned value of the fourth byte.

lists

The address of an array of name offsets in the display list. The pointer type is void because the offsets can be bytes, shorts, ints, or floats, depending on the value of *type*.

Remarks

The **glCallLists** function causes each display list in the list of names passed as *lists* to be executed. As a result, the functions saved in each display list are executed in order, just as if they were called without using a display list. Names of display lists that have not been defined are ignored.

The **glCallLists** function provides an efficient means for executing display lists.The *n* parameter specifies the number of lists with various name formats (specified by the *type* parameter) **glCallLists** executes.

The list of display list names is not null-terminated. Rather, *n* specifies how many names are to be taken from *lists*.

The **glListBase** function makes an additional level of indirection available. The **glListBase** function specifies an unsigned offset that is added to each display list name specified in *lists* before that display list is executed.

The **glCallLists** function can appear inside a display list. To avoid the possibility of infinite recursion resulting from display lists calling one another, a limit is placed on the nesting level of display lists during display list execution. This limit must be at least 64, and it depends on the implementation.

The OpenGL state is not saved and restored across a call to **glCallLists**. Thus, changes made to the OpenGL state during the execution of the display lists remain after execution is completed. Use **glPushAttrib**, **glPopAttrib**, **glPushMatrix**, and **glPopMatrix** to preserve the OpenGL state across **glCallLists** calls.

You can execute display lists between a call to **glBegin** and the corresponding call to **glEnd**, as long as the display list includes only functions that are allowed in this interval.

The following functions retrieve information related to the **glCallLists** function:

glGet with argument GL_LIST_BASE

glGet with argument GL_MAX_LIST_NESTING

glIsList

See Also

glBegin, **glCallList**, **glDeleteLists**, **glEnd**, **glGenLists**, **glGet**, **glIsList**, **glListBase**, **glNewList**, **glPopAttrib**, **glPopMatrix**, **glPushAttrib**, **glPushMatrix**

glClear Quick Info

[New - Windows 95, OEM Service Release 2]

The **glClear** function clears buffers within the viewport.

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

Parameters

mask

Bitwise OR of masks that indicate the buffers to be cleared. The four masks are as follows.

Mask	Buffer to be Cleared
GL_COLOR_BUFFER_BIT	The buffers currently enabled for color writing.
GL_DEPTH_BUFFER_BIT	The depth buffer.
GL_ACCUM_BUFFER_BIT	The accumulation buffer.
GL_STENCIL_BUFFER_BIT	The stencil buffer.

Remarks

The **glClear** function sets the bitplane area of the window to values previously selected by **glClearColor**, **glClearIndex**, **glClearDepth**, **glClearStencil**, and **glClearAccum**. You can clear multiple color buffers simultaneously by selecting more than one buffer at a time using **glDrawBuffer**.

The pixel-ownership test, the scissor test, dithering, and the buffer writemasks affect the operation of **glClear**. The scissor box bounds the cleared region. The alpha function, blend function, logical operation, stenciling, texture mapping, and *z*-buffering are ignored by **glClear**.

The **glClear** function takes a single argument (*mask*) that is the bitwise OR of several values indicating which buffer is to be cleared.

The value to which each buffer is cleared depends on the setting of the clear value for that buffer.

If a buffer is not present, a **glClear** call directed at that buffer has no effect.

The following functions retrieve information related to **glClear**:

glGet with argument GL_ACCUM_CLEAR_VALUE
glGet with argument GL_DEPTH_CLEAR_VALUE
glGet with argument GL_INDEX_CLEAR_VALUE
glGet with argument GL_COLOR_CLEAR_VALUE
glGet with argument GL_STENCIL_CLEAR_VALUE

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_VALUE	Any bit other than the four defined bits was set in *mask.*
GL_INVALID_OPERATION	**glClear** was called between a call to **glBegin** and the corresponding

call to **glEnd**.

See Also

glClearAccum, **glClearColor**, **glClearDepth**, **glClearIndex**, **glClearStencil**, **glDrawBuffer**, **glGet**, **glScissor**

glClearAccum Quick Info

[New - Windows 95, OEM Service Release 2]

The **glClearAccum** function clears values for the accumulation buffer.

```
void glClearAccum(
  GLfloat red,
  GLfloat green,
  GLfloat blue,
  GLfloat alpha
);
```

Parameters

red, green, blue, alpha

The red, green, blue, and alpha values used when the accumulation buffer is cleared. The default values are all zero.

Remarks

The **glClearAccum** function specifies the red, green, blue, and alpha values used by **glClear** to clear the accumulation buffer.

Values specified by **glClearAccum** are clamped to the range [- 1,1].

The following function retrieves information related to **glClearAccum**:

glGet with argument GL_ACCUM_CLEAR_VALUE

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glClearAccum** was called between a call to **glBegin** and the corresponding call to **glEnd**

See Also

glBegin, **glClear**, **glEnd**, **glGet**

glClearColor Quick Info

[New - Windows 95, OEM Service Release 2]

The **glClearColor** function specifies clear values for the color buffers.

```
void glClearColor(
  GLclampf red,
  GLclampf green,
  GLclampf blue,
  GLclampf alpha
);
```

Parameters

red, green, blue, alpha

The red, green, blue, and alpha values used when the color buffers are cleared. The default values are all zero.

Remarks

The **glClearColor** function specifies the red, green, blue, and alpha values used by **glClear** to clear the color buffers. Values specified by **glClearColor** are clamped to the range [0,1].

The following functions retrieve information related to the **glClearColor** function:

glGet with argument GL_ACCUM_CLEAR_VALUE
glGet with argument GL_COLOR_CLEAR_VALUE

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glClearColor** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glClear**, **glEnd**, **glGet**

glClearDepth Quick Info

[New - Windows 95, OEM Service Release 2]

The **glClearDepth** function specifies the clear value for the depth buffer.

```
void glClearDepth(
  GLclampd depth
);
```

Parameters

depth

The depth value used when the depth buffer is cleared.

Remarks

The **glClearDepth** function specifies the depth value used by **glClear** to clear the depth buffer. Values specified by **glClearDepth** are clamped to the range [0,1].

The following function retrieves information related to the **glClearDepth** function:

glGet with argument GL_DEPTH_CLEAR_VALUE

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glClearDepth** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glClear**, **glEnd**, **glGet**

glClearIndex Quick Info

[New - Windows 95, OEM Service Release 2]

The **glClearIndex** function specifies the clear value for the color-index buffers.

```
void glClearIndex(
  GLfloat c
);
```

Parameters

c

The index used when the color-index buffers are cleared. The default value is zero.

Remarks

The **glClearIndex** function specifies the index used by **glClear** to clear the color-index buffers. The *c* parameter is not clamped. Rather, *c* is converted to a fixed-point value with unspecified precision to the right of the binary point. The integer part of this value is then masked with 2^*m* - 1, where *m* is the number of bits in a color index stored in the frame buffer.

The following functions retrieve information related to **glClearIndex**:

glGet with argument GL_INDEX_CLEAR_VALUE
glGet with argument GL_INDEX_BITS

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glClearIndex** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glClear**, **glEnd**, **glGet**

glClearStencil Quick Info

[New - Windows 95, OEM Service Release 2]

The **glClearStencil** function specifies the clear value for the stencil buffer.

```
void glClearStencil(
  GLint s
);
```

Parameters

s

The index used when the stencil buffer is cleared. The default value is zero.

Remarks

The **glClearStencil** function specifies the index used by **glClear** to clear the stencil buffer. The *s* parameter is masked with 2^*m* - 1, where *m* is the number of bits in the stencil buffer.

The following functions retrieve information related to the **glClearStencil** function:

glGet with argument GL_STENCIL_CLEAR_VALUE
glGet with argument GL_STENCIL_BITS

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glClearStencil** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glClear**, **glEnd**, **glGet**

glClipPlane Quick Info

[New - Windows 95, OEM Service Release 2]

The **glClipPlane** function specifies a plane against which all geometry is clipped.

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

Parameters

plane
The clipping plane that is being positioned. Symbolic names of the form GL_CLIP_PLANE*i*, where *i* is an integer between 0 and GL_MAX_CLIP_PLANES - 1, are accepted.

equation
The address of an array of four double-precision floating-point values. These values are interpreted as a plane equation.

Remarks

Geometry is always clipped against the boundaries of a six-plane frustum in *x*, *y*, and *z*. The **glClipPlane** function allows the specification of additional planes, not necessarily perpendicular to the *x*- , *y*- , or *z*-axis, against which all geometry is clipped. Up to GL_MAX_CLIP_PLANES planes can be specified, where GL_MAX_CLIP_PLANES is at least six in all implementations. Because the resulting clipping region is the intersection of the defined half-spaces, it is always convex.

The **glClipPlane** function specifies a half-space using a four-component plane equation. When you call **glClipPlane**, *equation* is transformed by the inverse of the modelview matrix and stored in the resulting eye coordinates. Subsequent changes to the modelview matrix have no effect on the stored plane-equation components. If the dot product of the eye coordinates of a vertex with the stored plane equation components is positive or zero, the vertex is *in* with respect to that clipping plane. Otherwise, it is *out*.

Use the **glEnable** and **glDisable** functions to enable and disable clipping planes. Call clipping planes with the argument GL_CLIP_PLANE*i*, where *i* is the plane number.

By default, all clipping planes are defined as (0,0,0,0) in eye coordinates and are disabled.

It is always the case that GL_CLIP_PLANE*i* = GL_CLIP_PLANE0 + *i*.

The following functions retrieve information related to **glClipPlane**:

glGetClipPlane
glIsEnabled with argument GL_CLIP_PLANE*i*

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*plane* was not an accepted value.
GL_INVALID_OPERATION	**glClipPlane** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glDisable**, **glEnable**, **glEnd**, **glGetClipPlane**, **glIsEnabled**

glColor

[New - Windows 95, OEM Service Release 2]

glColor3b, glColor3d, glColor3f, glColor3i, glColor3s, glColor3ub, glColor3ui, glColor3us, glColor4b, glColor4d, glColor4f, glColor4i, glColor4s, glColor4ub, glColor4ui, glColor4us, glColor3bv, glColor3dv, glColor3fv, glColor3iv, glColor3sv, glColor3ubv, glColor3uiv, glColor3usv, glColor4bv, glColor4dv, glColor4fv, glColor4iv, glColor4sv, glColor4ubv, glColor4uiv, glColor4usv

These functions set the current color.

```
void glColor3b(
  GLbyte red,
  GLbyte green,
  GLbyte blue
);

void glColor3d(
  GLdouble red,
  GLdouble green,
  GLdouble blue
);

void glColor3f(
  GLfloat red,
  GLfloat green,
  GLfloat blue
);

void glColor3i(
  GLint red,
  GLint green,
  GLint blue
);

void glColor3s(
  GLshort red,
  GLshort green,
  GLshort blue
);

void glColor3ub(
  GLubyte red,
  GLubyte green,
  GLubyte blue
);

void glColor3ui(
  GLuint red,
  GLuint green,
  GLuint blue
);

void glColor3us(
  GLushort red,
  GLushort green,
  GLushort blue
```

```
);

void glColor4b(
  GLbyte red,
  GLbyte green,
  GLbyte blue,
  GLbyte alpha
);

void glColor4d(
  GLdouble red,
  GLdouble green,
  GLdouble blue,
  GLdouble alpha
);

void glColor4f(
  GLfloat red,
  GLfloat green,
  GLfloat blue,
  GLfloat alpha
);

void glColor4i(
  GLint red,
  GLint green,
  GLint blue,
  GLint alpha
);

void glColor4s(
  GLshort red,
  GLshort green,
  GLshort blue,
  GLshort alpha
);

void glColor4ub(
  GLubyte red,
  GLubyte green,
  GLubyte blue,
  GLubyte alpha
);

void glColor4ui(
  GLuint red,
  GLuint green,
  GLuint blue,
  GLuint alpha
);

void glColor4us(
  GLushort red,
  GLushort green,
  GLushort blue,
  GLushort alpha
);
```

Parameters

red, green, blue

New red, green, and blue values for the current color.

alpha

A new alpha value for the current color. Included only in the four-argument **glColor4** function.

```
void glColor3bv(
  const GLbyte *v
);

void glColor3dv(
  const GLdouble *v
);

void glColor3fv(
  const GLfloat *v
);

void glColor3iv(
  const GLint *v
);

void glColor3sv(
  const GLshort *v
);

void glColor3ubv(
  const GLubyte *v
);

void glColor3uiv(
  const GLuint *v
);

void glColor3usv(
  const GLushort *v
);

void glColor4bv(
  const GLbyte *v
);

void glColor4dv(
  const GLdouble *v
);

void glColor4fv(
  const GLfloat *v
);

void glColor4iv(
  const GLint *v
);

void glColor4sv(
  const GLshort *v
);

void glColor4ubv(
  const GLubyte *v
```

```
);

void glColor4uiv(
  const GLuint *v
);

void glColor4usv(
  const GLushort *v
);
```

Parameters

v

A pointer to an array that contains red, green, blue, and (sometimes) alpha values.

Remarks

OpenGL stores both a current single-valued color index and a current four-valued RGBA color. The **glColor** function sets a new four-valued RGBA color.

There are two major variants to **glColor**:

- The **glColor3** variants specify new red, green, and blue values explicitly, and set the current alpha value to 1.0 implicitly.
- The **glColor4** variants specify all four color components explicitly.

The **glColor3b**, **glColor4b**, **glColor3s**, **glColor4s**, **glColor3i**, and **glColor4i** functions take three or four signed byte, short, or long integers as arguments. When you append *v* to the name, the color functions can take a pointer to an array of such values.

Current color values are stored in floating-point format, with unspecified mantissa and exponent sizes. Unsigned integer color components, when specified, are linearly mapped to floating-point values such that the largest representable value maps to 1.0 (full intensity), and zero maps to 0.0 (zero intensity). Signed integer color components, when specified, are linearly mapped to floating-point values such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly.

Neither floating-point nor signed integer values are clamped to the range [0,1] before updating the current color. However, color components are clamped to this range before they are interpolated or written into a color buffer.

You can update the current color at any time. In particular, you can call **glColor** between a call to **glBegin** and the corresponding call to **glEnd**.

The following functions retrieve information related to the **glColor** functions:

glGet with argument GL_CURRENT_COLOR

glGet with argument GL_RGBA_MODE

See Also

glBegin, **glEnd**, **glGet**, **glIndex**

glColorMask Quick Info

[New - Windows 95, OEM Service Release 2]

The **glColorMask** function enables and disables writing of frame-buffer color components.

```
void glColorMask(
  GLboolean red,
  GLboolean green,
  GLboolean blue,
  GLboolean alpha
);
```

Parameters

red, green, blue, alpha

Specify whether red, green, blue, and alpha can or cannot be written into the frame buffer. The default values are all GL_TRUE, indicating that the color components can be written.

Remarks

The **glColorMask** function specifies whether the individual color components in the frame buffer can or cannot be written. If *red* is GL_FALSE, for example, no change is made to the red component of any pixel in any of the color buffers, regardless of the drawing operation attempted.

Changes to individual bits of components cannot be controlled. Rather, changes are either enabled or disabled for entire color components.

The following functions retrieve information related to **glColorMask**:

glGet with argument GL_COLOR_WRITEMASK

glGet with argument GL_RGBA_MODE

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glColorMask** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glColor**, **glDepthMask**, **glEnd**, **glGet**, **glIndex**, **glIndexMask**, **glStencilMask**

glColorMaterial Quick Info

[New - Windows 95, OEM Service Release 2]

The **glColorMaterial** function causes a material color to track the current color.

```
void glColorMaterial(
  GLenum face,
  GLenum mode
);
```

Parameters

face

Specifies whether front, back, or both front and back material parameters should track the current color. Accepted values are GL_FRONT, GL_BACK, and GL_FRONT_AND_BACK. The default value is GL_FRONT_AND_BACK.

mode

Specifies which of several material parameters track the current color. Accepted values are GL_EMISSION, GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, and GL_AMBIENT_AND_DIFFUSE. The default value is GL_AMBIENT_AND_DIFFUSE.

Remarks

The **glColorMaterial** function specifies which material parameters track the current color. When you enable GL_COLOR_MATERIAL, the material parameter or parameters specified by *mode*, of the material or materials specified by *face*, track the current color at all times. You enable and disable GL_COLOR_MATERIAL with the functions **glEnable** and **glDisable**, which you call with GL_COLOR_MATERIAL as their argument. By default, GL_COLOR_MATERIAL is disabled.

With **glColorMaterial**, you can change a subset of material parameters for each vertex using only the **glColor** function, without calling **glMaterial**. If you are going to specify only such a subset of parameters for each vertex, it is better to do so with **glColorMaterial** than with **glMaterial**.

The following functions retrieve information related to **glColorMaterial**:

glGet with argument GL_COLOR_MATERIAL_PARAMETER

glGet with argument GL_COLOR_MATERIAL_FACE

glIsEnabled with argument GL_COLOR_MATERIAL

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*face* or *mode* was not an accepted value.
GL_INVALID_OPERATION	**glColorMaterial** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glColor**, **glDisable**, **glEnable**, **glEnd**, **glGet**, **glIsEnabled**, **glLight**, **glLightModel**, **glMaterial**

glColorPointer

[New - Windows 95, OEM Service Release 2]

The **glColorPointer** function defines an array of colors.

```
void glColorPointer(
  GLint size,
  GLenum type,
  GLsizei stride,
  GLsizei count,
  const GLvoid * pointer
);
```

Parameters

size
The number of components per color. The value must be either 3 or 4.

type
The data type of each color component in a color array. Acceptable data types are specified with the following constants: GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, or GL_DOUBLE.

stride
The byte offset between consecutive colors. When *stride* is zero, the colors are tightly packed in the array.

count
The number of static colors, counting from the first color.

pointer
A pointer to the first component of the first color element in a color array.

Remarks

The **glColorPointer** function specifies the location and data format of an array of color components to use when rendering. The *stride* parameter determines the byte offset from one color to the next, enabling the packing of vertex attributes in a single array or storage in separate arrays. In some implementations, storing vertex attributes in a single array can be more efficient than the use of separate arrays. Starting from the first color array element, *count* indicates the total number of static elements. You can modify static elements, but once the elements are modified, you must explicitly specify the array again before using the array for any rendering. Nonstatic color array elements are not accessed until you call **glDrawArrays** or **glArrayElement**.

The color array is enabled when you specify the GL_COLOR_ARRAY constant with **glEnableClientState**. Calling **glArrayElement**, **or glDrawArrays** uses the color array that is thus enabled. By default, the color array is disabled. The **glColorPointer** calls are not entered in display lists.

When you specify a color array using **glColorPointer**, the values of all the function's color array parameters are saved in a client-side state, and you can cache static array elements. Because the color array parameters are in a client-side state, **glPushAttrib** and **glPopAttrib** do not save or restore the parameters' values.

Although specifying the color array within **glBegin** and **glEnd** pairs does not generate an error, the results are undefined.

The following functions retrieve information related to the **glColorPointer** function:

glIsEnabled with argument GL_COLOR_ARRAY

glGet with argument GL_COLOR_ARRAY_SIZE
glGet with argument GL_COLOR_ARRAY_TYPE
glGet with argument GL_COLOR_ARRAY_STRIDE
glGet with argument GL_COLOR_ARRAY_COUNT
glGetPointerv with argument GL_COLOR_ARRAY_POINTER

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_VALUE	*size* was not 3 or 4.
GL_INVALID_ENUM	*type* was not an accepted value.
GL_INVALID_VALUE	*stride* or *count* was negative.

See Also

glArrayElement, **glBegin**, **glDrawArrays**, **glEdgeFlagPointer**, **glEnableClientState**, **glEnd**, **glGet**, **glGetString**, **glGetPointerv**, **glIndexPointer**, **glIsEnabled**, **glNormalPointer**, **glPopAttrib**, **glPushAttrib**, **glTexCoordPointer**, **glVertexPointer**

glColorTableEXT

[New - Windows 95, OEM Service Release 2]

The **glColorTableEXT** function specifies the format and size of a palette for targeted paletted textures.

```
void glColorTableEXT(
  GLenum target,
  GLenum internalFormat,
  GLsizei width,
  GLenum format,
  GLenum type,
  const GLvoid * data
);
```

Parameters

target

The target texture that is to have its palette changed. Must be TEXTURE_1D, TEXTURE_2D, PROXY_TEXTURE_1D, or PROXY_TEXTURE_2D.

internalFormat

The internal format and resolution of the palette. This parameter can assume one of the following symbolic values:

Constant	Base Format	R Bits	G Bits	B Bits	A Bits
GL_R3_G3_B2	GL_RGB	3	3	2	–
GL_RGB4	GL_RGB	4	4	4	–
GL_RGB5	GL_RGB	5	5	5	–
GL_RGB8	GL_RGB	8	8	8	–
GL_RGB10	GL_RGB	10	10	10	–
GL_RGB12	GL_RGB	12	12	12	–
GL_RGB16	GL_RGB	16	16	16	–
GL_RGBA2	GL_RGBA	2	2	2	2
GL_RGBA4	GL_RGBA	4	4	4	4
GL_RGB5_A1	GL_RGBA	5	5	5	1
GL_RGBA8	GL_RGBA	8	8	8	8
GL_RGB10_A2	GL_RGBA	10	10	10	2
GL_RGB12	GL_RGBA	12	12	12	12
GL_RGBA16	GL_RGBA	16	16	16	16

width

The size of the palette. Must be 2^*n* ≥ 1 for some integer *n*.

format

The format of the pixel data. The following symbolic constants are accepted:

GL_RGBA

Each pixel is a group of four components in this order: red, green, blue, alpha. The RGBA format is determined in this way:

1. The **glColorTableEXT** function converts floating-point values directly to an internal format with unspecified precision. Signed integer values are mapped linearly to the internal format such that the most positive representable integer value maps to 1.0, and the most negative representable integer value maps to -1.0. Unsigned integer data is mapped similarly: the largest integer value maps to 1.0, and zero maps to 0.0.
2. The **glColorTableEXT** function multiplies the resulting color values by GL_c_SCALE and

adds them to GL_c_BIAS, where *c* is RED, GREEN, BLUE, and ALPHA for the respective color components. The results are clamped to the range [0,1].

3. If GL_MAP_COLOR is TRUE, **glColorTableEXT** scales each color component by the size of lookup table GL_PIXEL_MAP_c_TO_c, then replaces the component by the value that it references in that table; *c* is R, G, B, or A, respectively.

4. The **glColorTableEXT** function converts the resulting RGBA colors to fragments by attaching the current raster position *z*-coordinate and texture coordinates to each pixel, then assigning *x* and *y* window coordinates to the *n*th fragment such that

$x_{(n)} = x_{(r)} + n \bmod width$

$y_{(n)} = y_{(r)} + \lfloor n/width \rfloor$

where $(x_{(r)}, y_{(r)})$ is the current raster position.

5. These pixel fragments are then treated just like the fragments generated by rasterizing points, lines, or polygons. The **glColorTableEXT** function applies texture mapping, fog, and all the fragment operations before writing the fragments to the frame buffer.

GL_RED

Each pixel is a single red component.

The **glColorTableEXT** function converts this component to the internal format in the same way that the red component of an RGBA pixel is, then converts it to an RGBA pixel with green and blue set to 0.0, and alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.

GL_GREEN

Each pixel is a single green component.

The **glColorTableEXT** function converts this component to the internal format in the same way that the green component of an RGBA pixel is, and then converts it to an RGBA pixel with red and blue set to 0.0, and alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.

GL_BLUE

Each pixel is a single blue component.

The **glColorTableEXT** function converts this component to the internal format in the same way that the blue component of an RGBA pixel is, and then converts it to an RGBA pixel with red and green set to 0.0, and alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.

GL_ALPHA

Each pixel is a single alpha component.

The **glColorTableEXT** function converts this component to the internal format in the same way that the alpha component of an RGBA pixel is, and then converts it to an RGBA pixel with red, green, and blue set to 0.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.

GL_RGB

Each pixel is a group of three components in this order: red, green, blue.

The **glColorTableEXT** function converts each component to the internal format in the same way that the red, green, and blue components of an RGBA pixel are. The color triple is converted to an RGBA pixel with alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.

GL_BGR_EXT

Each pixel is a group of three components in this order: blue, green, red.

GL_BGR_EXT provides a format that matches the memory layout of Windows device-independent bitmaps (DIBs). Thus your applications can use the same data with Win32 function calls and OpenGL pixel function calls.

GL_BGRA_EXT

Each pixel is a group of four components in this order: blue, green, red, alpha.

GL_BGRA_EXT provides a format that matches the memory layout of Windows device-independent bitmaps (DIBs). Thus your applications can use the same data with Win32 function calls and OpenGL pixel function calls.

type

The data type for *data*. The following symbolic constants are accepted: GL_UNSIGNED_BYTE, GL_BYTE, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, and GL_FLOAT.

The following table summarizes the meaning of the valid constants for the *type* parameter.

Constant	Meaning
GL_UNSIGNED_BYTE	Unsigned 8-bit integer
GL_BYTE	Signed 8-bit integer
GL_UNSIGNED_SHORT	Unsigned 16-bit integer
GL_SHORT	Signed 16-bit integer
GL_UNSIGNED_INT	Unsigned 32-bit integer
GL_INT	32-bit integer
GL_FLOAT	Single-precision floating-point value

data

A pointer to the paletted texture data. The data is treated as single pixels of a 1-D texture palette entry for a palette entry.

Remarks

Paletted textures are defined with a palette of colors and a set of image data that is composed of indexes to color entries of a palette (a color table).

The **glColorTableEXT** function specifies the texture palette of a targeted texture. It takes the *data* from memory and converts the data as if each palette entry is a single pixel of a 1-D texture. The **glColorTableEXT** function unpacks and converts the data and translates it into an internal format that matches the given *format* as closely as possible.

If a palette's *width* is greater than the range of the color indexes in the texture data, some of the palette entries are unused. If a palette's *width* is less than the range of the color indexes in the texture data, the most significant bits of the texture data are ignored and only the appropriate number of bits in the index are used when accessing the palette. When you specify a proxy *target* using PROXY_TEXTURE_1D or PROXY_TEXTURE_2D, the palette of the proxy texture is resized and its parameters are set but no data is transferred or accessed.

When the *target* parameter is GL_PROXY_TEXTURE_1D or GL_PROXY_TEXTURE_2D, and the implementation does not support the values specified for either *format* or *width*, **glColorTableEXT** can fail to create the requested color table. In this case, the color table is empty and all parameters retrieved will be zero. You can determine whether OpenGL supports a particular color table format and size by calling **glColorTableEXT** with a proxy target, and then calling **glGetColorTableParameterivEXT** or **glGetColorTableParameterfvEXT** to determine whether the width parameter matches that set by **glColorTableEXT**. If the retrieved width is zero, the color table request by **glColorTable** failed. If the retrieved width is not zero, you can call **glColorTable** with the real target with TEXTURE_1D or TEXTURE_2D to set the color table.

Note The **glColorTableEXT** function is an extension function that is not part of the standard OpenGL library but is part of the GL_EXT_paletted_texture extension. To check whether your implementation of OpenGL supports **glColorTableEXT**, call **glGetString**(GL_EXTENSIONS). If it returns GL_EXT_paletted_texture, **glColorTableEXT** is supported. To obtain the function address of an extension function, call **wglGetProcAddress**.

To retrieve the actual color table data specified by the **glColorTableEXT** function, call **glGetColorTableEXT**. To retrieve the parameters, such as *width* and *format*, of the color table specified by the **glColorTableEXT** function, call the **glGetColorTableParameterivEXT** or **glGetColorTableParameterfvEXT** function.

Error Codes

The following are the errors generated and their conditions.

Error Code	Condition
GL_INVALID_VALUE	*width* was an invalid integer.
GL_INVALID_ENUM	*target*, *internalFormat*, *format*, or *type* was not an accepted value.
GL_INVALID_OPERATION	**glColorTableEXT** was called between **glBegin** and **glEnd** pairs.

See Also

glBegin, **glColorSubTableEXT**, **glEnd**, **glGetColorTableEXT**, **glGetColorTableParameterfvEXT**, **glGetColorTableParameterivEXT**, **wglGetProcAddress**

glColorSubTableEXT

[New - Windows 95, OEM Service Release 2]

The **glColorSubTableEXT** function specifies a portion of the targeted texture's palette to be replaced.

```
void glColorSubTableEXT(
  GLenum target,
  GLsizei start,
  GLsizei count,
  GLenum format,
  GLenum type,
  const GLvoid * data
);
```

Parameters

target
The target paletted texture that is to have its palette changed. Must be TEXTURE_1D or TEXTURE_2D.

start
The starting palette index entry of the palette to be changed.

count
The number of palette index entries of the palette to be changed beginning at *start*. The *count* parameter determines the range of palette index entries that are changed.

format
The format of the pixel data. The following symbolic constants are accepted:

GL_RGBA
Each pixel is a group of four components in the following order: red, green, blue, alpha. The RGBA format is determined in this way:

1. The **glColorSubTableEXT** function converts floating-point values directly to an internal format with unspecified precision. Signed integer values are mapped linearly to the internal format such that the most positive representable integer value maps to 1.0, and the most negative representable value maps to - 1.0. Unsigned integer data is mapped similarly: the largest integer value maps to 1.0, and zero maps to 0.0.
2. The **glColorSubTableEXT** function multiplies the resulting color values by GL_c_SCALE and adds them to GL_c_BIAS, where *c* is RED, GREEN, BLUE, and ALPHA for the respective color components. The results are clamped to the range [0,1].
3. If GL_MAP_COLOR is TRUE, **glColorSubTableEXT** scales each color component by the size of lookup table GL_PIXEL_MAP_c_TO_c, then replaces the component by the value that it references in that table; *c* is R, G, B, or A, respectively.
4. The **glColorSubTableEXT** function converts the resulting RGBA colors to fragments by attaching the current raster position *z*-coordinate and texture coordinates to each pixel, then assigning *x* and *y* window coordinates to the *n*th fragment such that

$$x_{(n)} = x_{(r)} + n \bmod width$$

$$y_{(n)} = y_{(r)} + \lfloor n/width \rfloor$$

where $(x_{(r)}, y_{(r)})$ is the current raster position.

5. These pixel fragments are then treated just like the fragments generated by rasterizing points, lines, or polygons. The **glColorSubTableEXT** function applies texture mapping, fog, and all the fragment operations before writing the fragments to the frame buffer.

GL_RED
Each pixel is a single red component.

The **glColorSubTableEXT** function converts this component to the internal format in the same way

that the red component of an RGBA pixel is, then converts it to an RGBA pixel with green and blue set to 0.0, and alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.

GL_GREEN
Each pixel is a single green component.

The **glColorSubTableEXT** function converts this component to the internal format in the same way that the green component of an RGBA pixel is, and then converts it to an RGBA pixel with red and blue set to 0.0, and alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.

GL_BLUE
Each pixel is a single blue component.

The **glColorSubTableEXT** function converts this component to the internal format in the same way that the blue component of an RGBA pixel is, and then converts it to an RGBA pixel with red and green set to 0.0, and alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.

GL_ALPHA
Each pixel is a single alpha component.

The **glColorSubTableEXT** function converts this component to the internal format in the same way that the alpha component of an RGBA pixel is, and then converts it to an RGBA pixel with red, green, and blue set to 0.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.

GL_RGB
Each pixel is a group of three components in this order: red, green, blue.

The **glColorSubTableEXT** function converts each component to the internal format in the same way that the red, green, and blue components of an RGBA pixel are. The color triple is converted to an RGBA pixel with alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.

GL_BGR_EXT
Each pixel is a group of three components in this order: blue, green, red.

GL_BGR_EXT provides a format that matches the memory layout of Windows device-independent bitmaps (DIBs). Thus your applications can use the same data with Win32 function calls and OpenGL pixel function calls.

GL_BGRA_EXT
Each pixel is a group of four components in this order: blue, green, red, alpha.

GL_BGRA_EXT provides a format that matches the memory layout of Windows device-independent bitmaps (DIBs). Thus your applications can use the same data with Win32 function calls and OpenGL pixel function calls.

type

The data type for *data*. The following symbolic constants are accepted: GL_UNSIGNED_BYTE, GL_BYTE, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, and GL_FLOAT.

The following table summarizes the meaning of the valid constants for the *type* parameter.

Constant	Meaning
GL_UNSIGNED_BYTE	Unsigned 8-bit integer
GL_BYTE	Signed 8-bit integer
GL_UNSIGNED_SHORT	Unsigned 16-bit integer
GL_SHORT	Signed 16-bit integer
GL_UNSIGNED_INT	Unsigned 32-bit integer
GL_INT	32-bit integer
GL_FLOAT	Single-precision floating-point value

data

A pointer to the paletted texture data. The data is treated as single pixels of a 1-D texture palette entry for a palette entry.

Remarks

The **glColorSubTableEXT** function specifies portions of the current targeted texture's palette to be replaced. Unlike **glColorTableEXT**, you cannot specify the *target* parameter to be a proxy texture palette.

Note The **glColorSubTableEXT** function is an extension function that is not part of the standard OpenGL library but is part of the GL_EXT_paletted_texture extension. To check whether your implementation of OpenGL supports **glColorSubTableEXT**, call **glGetString(**GL_EXTENSIONS**)**. If it returns GL_EXT_paletted_texture, **glColorSubTableEXT** is supported. To obtain the function address of an extension function, call **wglGetProcAddress**.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_VALUE	*start* or *count* was an invalid integer.
GL_INVALID_ENUM	*target*, *format*, or *type* was not an accepted value.
GL_INVALID_OPERATION	**glColorSubTableEXT** was called between **glBegin** and **glEnd** pairs.

See Also

glBegin, **glColorTableEXT**, **glEnd**, **glGetColorTableEXT**, **glGetColorTableParameterfvEXT**, **glGetColorTableParameterivEXT**, **glGetString**, **wglGetProcAddress**

glCopyPixels Quick Info

[New - Windows 95, OEM Service Release 2]

The **glCopyPixels** function copies pixels in the frame buffer.

```
void glCopyPixels(
  GLint x,
  GLint y,
  GLsizei width,
  GLsizei height,
  GLenum type
);
```

Parameters

x, y

The window coordinates of the lower-left corner of the rectangular region of pixels to be copied.

width, height

The dimensions of the rectangular region of pixels to be copied. Both must be nonnegative.

type

Specifies whether **glCopyPixels** is to copy color values, depth values, or stencil values. The acceptable symbolic constants are:

GL_COLOR

The **glCopyPixels** function reads indexes or RGBA colors from the buffer currently specified as the read source buffer (see **glReadBuffer**).

If OpenGL is in color-index mode:

1. Each index that is read from this buffer is converted to a fixed-point format with an unspecified number of bits to the right of the binary point.
2. Each index is shifted left by GL_INDEX_SHIFT bits, and added to GL_INDEX_OFFSET.

If GL_INDEX_SHIFT is negative, the shift is to the right. In either case, zero bits fill otherwise unspecified bit locations in the result.

3. If GL_MAP_COLOR is true, the index is replaced with the value that it references in lookup table GL_PIXEL_MAP_I_TO_I.
4. Whether the lookup replacement of the index is done or not, the integer part of the index is then **AND**ed with 2^*b* - 1, where *b* is the number of bits in a color-index buffer.

If OpenGL is in RGBA mode:

1. The red, green, blue, and alpha components of each pixel that is read are converted to an internal floating-point format with unspecified precision.
2. The conversion maps the largest representable component value to 1.0, and component value zero to 0.0.
3. The resulting floating-point color values are then multiplied by GL_c_SCALE and added to GL_c_BIAS, where *c* is RED, GREEN, BLUE, and ALPHA for the respective color components.
4. The results are clamped to the range [0,1].
5. If GL_MAP_COLOR is true, each color component is scaled by the size of lookup table GL_PIXEL_MAP_c_TO_c, and then replaced by the value that it references in that table; *c* is R, G, B, or A, respectively.

The resulting indexes or RGBA colors are then converted to fragments by attaching the current raster position *z*-coordinate and texture coordinates to each pixel, and then assigning window coordinates ($x_{(r)} + i$, $y_{(r)} + j$), where ($x_{(r)}$ $y_{(r)}$) is the current raster position, and the pixel was the pixel in the *i* position in the *j* row. These pixel fragments are then treated just like the fragments generated by rasterizing points, lines, or polygons. Texture mapping, fog, and all the fragment

operations are applied before the fragments are written to the frame buffer.

GL_DEPTH

Depth values are read from the depth buffer and converted directly to an internal floating-point format with unspecified precision. The resulting floating-point depth value is then multiplied by GL_DEPTH_SCALE and added to GL_DEPTH_BIAS. The result is clamped to the range [0,1].

The resulting depth components are then converted to fragments by attaching the current raster position color or color index and texture coordinates to each pixel, then assigning window coordinates ($x_{(r)} + i$, $y_{(r)} + j$), where ($x_{(r)}$, $y_{(r)}$) is the current raster position, and the pixel was the pixel in the *i* position in the *j* row. These pixel fragments are then treated just like the fragments generated by rasterizing points, lines, or polygons. Texture mapping, fog, and all the fragment operations are applied before the fragments are written to the frame buffer.

GL_STENCIL

Stencil indexes are read from the stencil buffer and converted to an internal fixed-point format with an unspecified number of bits to the right of the binary point. Each fixed-point index is then shifted left by GL_INDEX_SHIFT bits, and added to GL_INDEX_OFFSET. If GL_INDEX_SHIFT is negative, the shift is to the right. In either case, zero bits fill otherwise unspecified bit locations in the result. If GL_MAP_STENCIL is true, the index is replaced with the value that it references in lookup table GL_PIXEL_MAP_S_TO_S. Whether the lookup replacement of the index is done or not, the integer part of the index is then **AND**ed with 2^*b* - 1, where *b* is the number of bits in the stencil buffer. The resulting stencil indexes are then written to the stencil buffer such that the index read from the *i* location of the *j* row is written to location ($x_{(r)} + i$, $y_{(r)} + j$), where ($x_{(r)}$, $y_{(r)}$) is the current raster position. Only the pixel-ownership test, the scissor test, and the stencil writemask affect these writes.

Remarks

The **glCopyPixels** function copies a screen-aligned rectangle of pixels from the specified frame buffer location to a region relative to the current raster position. Its operation is well defined only if the entire pixel source region is within the exposed portion of the window. Results of copies from outside the window, or from regions of the window that are not exposed, are hardware dependent and undefined.

The *x* and *y* parameters specify the window coordinates of the lower-left corner of the rectangular region to be copied. The *width* and *height* parameters specify the dimensions of the rectangular region to be copied. Both *width* and *height* must be nonnegative.

Several parameters control the processing of the pixel data while it is being copied. These parameters are set with three functions: **glPixelTransfer**, **glPixelMap**, and **glPixelZoom**. This topic describes the effects on **glCopyPixels** of most, but not all, of the parameters specified by these three functions.

The **glCopyPixels** function copies values from each pixel with the lower-left corner at ($x + i$, $y + j$) for $0 \le i < width$ and $0 \le j < height$. This pixel is said to be the *i* pixel in the *j* row. Pixels are copied in row order from the lowest to the highest row, left to right in each row.

The *type* parameter specifies whether color, depth, or stencil data is to be copied.

The rasterization described thus far assumes pixel zoom factors of 1.0. If you use **glPixelZoom** to change the *x* and *y* pixel zoom factors, pixels are converted to fragments as follows. If ($x_{(r)}$, $y_{(r)}$) is the current raster position, and a given pixel is in the *i* location in the *j* row of the source pixel rectangle, then fragments are generated for pixels whose centers are in the rectangle with corners at

($x_{(r)} + zoom_{(x)}\ i$, $y_{(r)} + zoom_{(y)}\ j$)

and

($x_{(r)} + zoom_{(x)}\ (i + 1)$, $y_{(r)} + zoom_{(y)}\ (j + 1)$)

where $zoom_{(x)}$ is the value of GL_ZOOM_X and $zoom_{(y)}$ is the value of GL_ZOOM_Y.

Modes specified by **glPixelStore** have no effect on the operation of **glCopyPixels**.

The following functions retrieve information related to **glCopyPixels**:

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID

To copy the color pixel in the lower-left corner of the window to the current raster position, use

glCopyPixels(0, 0, 1, 1, GL_COLOR**)**;

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*type* was not an accepted value.
GL_INVALID_VALUE	Either *width* or *height* was negative.
GL_INVALID_OPERATION	*type* was GL_DEPTH and there was no depth buffer.
GL_INVALID_OPERATION	*type* was GL_STENCIL and there was no stencil buffer.
GL_INVALID_OPERATION	**glCopyPixels** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glDepthFunc**, **glDrawBuffer**, **glDrawPixels**, **glEnd**, **glGet**, **glPixelMap**, **glPixelStore**, **glPixelTransfer**, **glPixelZoom**, **glRasterPos**, **glReadBuffer**, **glReadPixels**, **glStencilFunc**

glCopyTexImage1D

[New - Windows 95, OEM Service Release 2]

The **glCopyTexImage1D** function copies pixels from the frame buffer into a one-dimensional texture image.

```
void glCopyTexImage1D(
  GLenum target,
  GLint level,
  GLenum
internalFormat,
  GLint x,
  GLint y,
  GLsizei width,
  GLint border
);
```

Parameters

target

The target for which the image data will be changed. Must have the value GL_TEXTURE_1D.

level

The level-of-detail number. Level 0 is the base image. Level *n* is the *n*th mipmap reduction image.

internalFormat

The internal format and resolution of the texture data. The values 1, 2, 3, and 4 are not accepted for *internalFormat*. This parameter can assume one of the following symbolic values:

Constant	Base Format	R Bits	G Bits	B Bits	A Bits	L Bits	I Bits
ALPHA	–	–	–	–	–	–	–
ALPHA4	ALPHA	–	–	–	4	–	–
ALPHA8	ALPHA	–	–	–	8	–	–
ALPHA12	ALPHA	–	–	–	12	–	–
ALPHA16	ALPHA	–	–	–	16	–	–
LUMINANCE	–	–	–	–	–	–	–
LUMINANCE4	LUMINANCE	–	–	–	–	4	–
LUMINANCE8	LUMINANCE	–	–	–	–	8	–
LUMINANCE12	LUMINANCE	–	–	–	–	12	–
LUMINANCE16	LUMINANCE	–	–	–	–	16	–
LUMINANCE_ALPHA	–	–	–	–	–	–	–
LUMINANCE4_ALPHA4	LUMINANCE_ALPHA	–	–	–	4	4	–
LUMINANCE6_ALPHA2	LUMINANCE_ALPHA	–	–	–	2	6	–
LUMINANCE8_ALPHA8	LUMINANCE_ALPHA	–	–	–	8	8	–
LUMINANCE12_ALPHA4	LUMINANCE_ALPHA	–	–	–	4	12	–
LUMINANCE12_ALPHA12	LUMINANCE_ALPHA	–	–	–	12	12	–
LUMINANCE16_ALPHA16	LUMINANCE_ALPHA	–	–	–	16	16	–

INTENSITY	–	–	–	–	–	–	–
INTENSITY4	INTENSITY	–	–	–	–	–	4
INTENSITY8	INTENSITY	–	–	–	–	–	8
INTENSITY12	INTENSITY	–	–	–	–	–	12
INTENSITY16	INTENSITY	–	–	–	–	–	16
GL_RGB	–	–	–	–	–	–	–
GL_R3_G3_B2	GL_RGB	3	3	2	–	–	–
GL_RGB4	GL_RGB	4	4	4	–	–	–
GL_RGB5	GL_RGB	5	5	5	–	–	–
GL_RGB8	GL_RGB	8	8	8	–	–	–
GL_RGB10	GL_RGB	10	10	10	–	–	–
GL_RGB12	GL_RGB	12	12	12	–	–	–
GL_RGB16	GL_RGB	16	16	16	–	–	–
GL_RGBA	–	–	–	–	–	–	–
GL_RGBA2	GL_RGBA	2	2	2	2	–	–
GL_RGBA4	GL_RGBA	4	4	4	4	–	–
GL_RGB5_A1	GL_RGBA	5	5	5	1	–	–
GL_RGBA8	GL_RGBA	8	8	8	8	–	–
GL_RGB10_A2	GL_RGBA	10	10	10	2	–	–
GL_RGBA12	GL_RGBA	12	12	12	12	–	–
GL_RGBA16	GL_RGBA	16	16	16	16	–	–

x, *y*
The window coordinates of the lower-left corner of the row of pixels to be copied.

width
The width of the texture image. Must be 2^*n* + 2 * *border* for some integer *n*. The height of the texture image is 1.

border
The width of the border. Must be either zero or 1.

Remarks

The **glCopyTexImage1D** function defines a one-dimensional texture image using pixels from the current frame buffer, rather than from main memory as is the case for **glTexImage1D**.

Using the mipmap level specified with *level*, texture arrays are defined as a pixel row aligned with the lower-left corner of the window at the coordinates specified by *x* and *y*, with a length equal to *width* + 2 * *border*. The internal format of the texture array is specified with the *internalFormat* parameter.

The **glCopyTexImage1D** function processes the pixels in a row in the same way as **glCopyPixels**, except that before the final conversion of the pixels, all pixel component values are clamped to the range [0, 1] and converted to the texture's internal format for storage in the texture array. Pixel ordering is determined with lower *x* coordinates corresponding to lower texture coordinates. If any of the pixels within a specified row of the current frame buffer are outside the window associated with the current rendering context, then their values are undefined.

You cannot include calls to **glCopyTexImage1D** in display lists.

Note The **glCopyTexImage1D** function is only available in OpenGL version 1.1 or later.

Texturing has no effect in color-index mode. The **glPixelStore** and **glPixelTransfer** functions affect

texture images in exactly the way they affect **glDrawPixels**.

The following function retrieves information related to **glCopyTexImage1D**:

glIsEnabled with argument GL_TEXTURE_1D

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*target* was not an accepted value.
GL_INVALID_VALUE	*level* was less than zero or greater than log sub 2(*max*), where *max* is the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_VALUE	*border* was not zero or 1.
GL_INVALID_VALUE	*width* was less than zero, greater than 2 + GL_MAX_TEXTURE_SIZE; or *width* cannot be represented as 2^*n* + 2 * *border* for some integer *n*.
GL_INVALID_OPERATION	**glCopyTexImage1D** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glCopyPixels, **glCopyTexImage2D**, **glDrawPixels**, **glFog**, **glPixelStore**, **glPixelTransfer**, **glTexEnv**, **glTexGen**, **glTexImage1D**, **glTexImage2D**, **glTexParameter**

glCopyTexImage2D

[New - Windows 95, OEM Service Release 2]

The **glCopyTexImage2D** function copies pixels from the frame buffer into a two-dimensional texture image.

```
void glCopyTexImage2D(
  GLenum target,
  GLint level,
  GLenum internalFormat,
  GLint x,
  GLint y,
  GLsizei width,
  GLsizei height,
  GLint border
);
```

Parameters

target
The target to which the image data will be changed. Must have the value GL_TEXTURE_2D.

level
The level-of-detail number. Level 0 is the base image. Level *n* is the *n*th mipmap reduction image.

internalFormat
The internal format and resolution of the texture data. The values 1, 2, 3, and 4 are not accepted for *internalFormat*. The parameter can assume one of the following symbolic values:

Constant	Base Format	R Bits	G Bits	B Bits	A Bits	L Bits	I Bits
ALPHA	–	–	–	–	–	–	–
ALPHA4	ALPHA	–	–	–	4	–	–
ALPHA8	ALPHA	–	–	–	8	–	–
ALPHA12	ALPHA	–	–	–	12	–	–
ALPHA16	ALPHA	–	–	–	16	–	–
LUMINANCE	–	–	–	–	–	–	–
LUMINANCE4	LUMINANCE	–	–	–	–	4	–
LUMINANCE8	LUMINANCE	–	–	–	–	8	–
LUMINANCE12	LUMINANCE	–	–	–	–	12	–
LUMINANCE16	LUMINANCE	–	–	–	–	16	–
LUMINANCE_ALPHA	–	–	–	–	–	–	–
LUMINANCE4_ALPHA4	LUMINANCE_ALPHA	–	–	–	4	4	–
LUMINANCE6_ALPHA2	LUMINANCE_ALPHA	–	–	–	2	6	–
LUMINANCE8_ALPHA8	LUMINANCE_ALPHA	–	–	–	8	8	–
LUMINANCE12_ALPHA4	LUMINANCE_ALPHA	–	–	–	4	12	–
LUMINANCE12_ALPHA12	LUMINANCE_ALPHA	–	–	–	12	12	–
LUMINANCE16_ALPHA16	LUMINANCE_ALPHA	–	–	–	16	16	–

INTENSITY	–	–	–	–	–	–	–
INTENSITY4	INTENSITY	–	–	–	–	–	4
INTENSITY8	INTENSITY	–	–	–	–	–	8
INTENSITY12	INTENSITY	–	–	–	–	–	12
INTENSITY16	INTENSITY	–	–	–	–	–	16
GL_RGB	–	–	–	–	–	–	–
GL_R3_G3_B2	GL_RGB	3	3	2	–	–	–
GL_RGB4	GL_RGB	4	4	4	–	–	–
GL_RGB5	GL_RGB	5	5	5	–	–	–
GL_RGB8	GL_RGB	8	8	8	–	–	–
GL_RGB10	GL_RGB	10	10	10	–	–	–
GL_RGB12	GL_RGB	12	12	12	–	–	–
GL_RGB16	GL_RGB	16	16	16	–	–	–
GL_RGBA	–	–	–	–	–	–	–
GL_RGBA2	GL_RGBA	2	2	2	2	–	–
GL_RGBA4	GL_RGBA	4	4	4	4	–	–
GL_RGB5_A1	GL_RGBA	5	5	5	1	–	–
GL_RGBA8	GL_RGBA	8	8	8	8	–	–
GL_RGB10_A2	GL_RGBA	10	10	10	2	–	–
GL_RGBA12	GL_RGBA	12	12	12	12	–	–
GL_RGBA16	GL_RGBA	16	16	16	16	–	–

x, *y*
The window coordinates of the lower-left corner of the rectangular region of pixels to be copied.

width
The width of the texture image. Must be 2^*n* + 2 * *border* for some integer *n*.

height
The height of the texture image. Must be 2^*n* + 2 * *border* for some integer *n*.

border
The width of the border. Must be either zero or 1.

Remarks

The **glCopyTexImage2D** function defines a two-dimensional texture image using pixels from the current frame buffer, rather than from main memory as is the case for **glTexImage2D**.

Using the mipmap level specified with *level*, texture arrays are defined as a rectangle of pixels with the lower-left corner located at the coordinates *x* and *y*, width equal to *width* + (2 * *border*), and a height equal to *height* + (2 * *border*). The internal format of the texture array is specified with the *internalFormat* parameter.

The **glCopyTexImage2D** function processes the pixels in a row in the same way as **glCopyPixels** except that before the final conversion of the pixels, all pixel component values are clamped to the range [0, 1] and converted to the texture's internal format for storage in the texture array. Pixel ordering is determined with lower *x* and *y* coordinates corresponding to lower *s* and *t* texture coordinates. If any of the pixels within a specified row of the current frame buffer are outside the window associated with the current rendering context, then their values are undefined.

You cannot include calls to **glCopyTexImage2D** in display lists.

Note The **glCopyTexImage2D** function is only available in OpenGL version 1.1 or later.

Texturing has no effect in color-index mode. The **glPixelStore** and **glPixelTransfer** functions affect texture images in exactly the way they affect **glDrawPixels**.

The following function retrieves information related to **glCopyTexImage2D**:

glIsEnabled with argument GL_TEXTURE_2D

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*target* was not an accepted value.
GL_INVALID_VALUE	*level* was less than zero or greater than log sub 2(*max*), where *max* is the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_VALUE	*border* was not zero or 1.
GL_INVALID_VALUE	*width* was less than zero, greater than 2 + GL_MAX_TEXTURE_SIZE; or *width* cannot be represented as 2^*n* + 2 * *border* for some integer *n*.
GL_INVALID_OPERATION	**glCopyTexImage2D** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glCopyTexImage1D**, **glDrawPixels**, **glEnd**, **glFog**, **glPixelStore**, **glPixelTransfer**, **glTexEnv**, **glTexGen**, **glTexImage1D**, **glTexImage2D**, **glTexParameter**

glCopyTexSubImage1D

[New - Windows 95, OEM Service Release 2]

The **glCopyTexSubImage1D** function copies a sub-image of a one-dimensional texture image from the frame buffer.

```
void glCopyTexSubImage1D(
  GLenum target,
  GLint level,
  GLint xoffset,
  GLint x,
  GLint y,
  GLsizei width
);
```

Parameters

target
The target to which the image data will be changed. Must have the value GL_TEXTURE_1D.

level
The level-of-detail number. Level 0 is the base image. Level *n* is the *n*th mipmap reduction image.

xoffset
The texel offset within the texture array.

x, *y*
The window coordinates of the lower-left corner of the row of pixels to be copied.

width
The width of the sub-image of the texture image. Specifying a texture sub-image with zero width has no effect.

Remarks

The **glCopyTexSubImage1D** function replaces a portion of a one-dimensional texture image using pixels from the current frame buffer, rather than from main memory as is the case for **glTexSubImage1D**.

A row of pixels beginning with the window coordinates specified by *x* and *y* and with the length *width* replaces the portion of the texture array with the indexes *xoffset* through *xoffset* + (*width* - 1). The destination in the texture array cannot include any texels outside the originally specified texture array.

The **glCopyTexSubImage1D** function processes the pixels in a row in the same way as **glCopyPixels** except that before the final conversion of the pixels, all pixel component values are clamped to the range [0, 1] and converted to the texture's internal format for storage in the texture array. Pixel ordering is determined with lower *x* coordinates corresponding to lower texture coordinates. If any of the pixels within a specified row of the current frame buffer are outside the window associated with the current rendering context, then their values are undefined.

No change is made to the *internalFormat*, *width*, or *border* parameter of the specified texture array or to texel values outside the specified texture sub-image.

You cannot include calls to **glCopyTexSubImage1D** in display lists.

Note The **glCopyTexSubImage1D** function is only available in OpenGL version 1.1 or later.

Texturing has no effect in color-index mode. The **glPixelStore** and **glPixelTransfer** functions affect texture images in exactly the way they affect the way pixels are drawn using **glDrawPixels**.

The following functions retrieve information related to **glCopyTexSubImage1D**:

glGetTexImage
glIsEnabled with argument GL_TEXTURE_1D

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*target* was not an accepted value.
GL_INVALID_VALUE	*level* was less than zero or greater than log sub 2(*max*), where *max* is the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_VALUE	*xoffset* was less than *border* or (*xoffset* + *width*) was greater than (*w* + *border*), where *w* is GL_TEXTURE_WIDTH and *border* is GL_TEXTURE_BORDER. Note that *w* includes twice the *border* width.
GL_INVALID_VALUE	*width* was less than *border* or *y* was less than *border*, where *border* is the border width of the texture array.
GL_INVALID_OPERATION	The texture array was not defined by a previous **glTexImage1D** operation.
GL_INVALID_OPERATION	**glCopyTexSubImage1D** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glCopyTexSubImage2D**, **glDrawPixels**, **glEnd**, **glFog**, **glPixelStore**, **glPixelTransfer**, **glTexEnv**, **glTexGen**, **glTexImage1D**, **glTexImage2D**, **glTexSubImage1D**, **glTexSubImage2D**, **glTexParameter**

glCopyTexSubImage2D

[New - Windows 95, OEM Service Release 2]

The **glCopyTexSubImage2D** function copies a sub-image of a two-dimensional texture image from the frame buffer.

```
void glCopyTexSubImage2D(
  GLenum target,
  GLint level,
  GLint xoffset,
  GLint yoffset,
  GLint x,
  GLint y,
  GLsizei width,
  GLsizei height
);
```

Parameters

target
The target to which the image data will be changed and can only have the value GL_TEXTURE_2D.

level
The level-of-detail number. Level 0 is the base image. Level *n* is the *n*th mipmap reduction image.

xoffset
The texel offset in the *x* direction within the texture array.

yoffset
The texel offset in the *y* direction within the texture array.

x, *y*
The window coordinates of the lower-left corner of the row of pixels to be copied.

width
The width of the sub-image of the texture image. Specifying a texture sub-image with zero width has no effect.

height
The height of the sub-image of the texture image. Specifying a texture sub-image with zero width has no effect.

Remarks

The **glCopyTexSubImage2D** function replaces a rectangular portion of a two-dimensional texture image with pixels from the current frame buffer, rather than from main memory as is the case for **glTexSubImage2D**.

A rectangle of pixels beginning with the *x* and *y* window coordinates and with the dimensions *width* and *height* replaces the portion of the texture array with the indexes *xoffset* through *xoffset* + (*width* - 1), with the indexes *yoffset* through *yoffset* + (*width* - 1) at the mipmap level specified by *level*. The destination rectangle in the texture array cannot include any texels outside the originally specified texture array.

The **glCopyTexSubImage2D** function processes the pixels in a row in the same way as **glCopyPixels** except that before the final conversion of the pixels, all pixel component values are clamped to the range [0, 1] and converted to the texture's internal format for storage in the texture array. Pixel ordering is determined with lower *x* coordinates corresponding to lower texture coordinates. If any of the pixels within a specified row of the current frame buffer are outside the window associated with the current rendering context, then their values are undefined.

If any of the pixels within the specified rectangle of the current frame buffer are outside the read window

associated with the current rendering context, then the values obtained for those pixels are undefined. No change is made to the *internalFormat*, *width*, *height*, or *border* parameter of the specified texture array or to texel values outside the specified texture sub-image.

You cannot include calls to **glCopyTexSubImage2D** in display lists.

Note The **glCopyTexSubImage2D** function is only available in OpenGL version 1.1 or later.

Texturing has no effect in color-index mode. The **glPixelStore** and **glPixelTransfer** functions affect texture images in exactly the way they affect the way pixels are drawn using **glDrawPixels**.

The following functions retrieve information related to **glCopyTexSubImage2D**:

glGetTexImage
glIsEnabled with argument GL_TEXTURE_2D.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*target* was not an accepted value.
GL_INVALID_VALUE	*level* was less than zero or greater than log sub 2(*max*), where *max* is the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_VALUE	*xoffset* was less than *border*, (*xoffset* + *width*) was greater than (*w* + *border*), *yoffset* was less than *border,* or (*yoffset* + *height*) was greater than (*h* + *border*), where *w* is GL_TEXTURE_WIDTH, *h* is GL_TEXTURE_HEIGHT, and *border* is GL_TEXTURE_BORDER. Note that *w* includes twice the border width.
GL_INVALID_VALUE	*width* was less than *border* or *y* was less than *border*, where *border* is the border width of the texture array.
GL_INVALID_OPERATION	The texture array was not defined by a previous **glTexImage2D** operation.
GL_INVALID_OPERATION	**glCopyTexSubImage2D** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glCopyPixels**, **glCopyTexSubImage1D**, **glDrawPixels**, **glEnd**, **glFog**, **glPixelStore**, **glPixelTransfer**, **glTexEnv**, **glTexGen**, **glTexImage2D**, **glTexSubImage2D**, **glTexParameter**

glCullFace Quick Info

[New - Windows 95, OEM Service Release 2]

The **glCullFace** function specifies whether front- or back-facing facets can be culled.

```
void glCullFace(
  GLenum mode
);
```

Parameters

mode

Specifies whether front- or back-facing facets are candidates for culling. Symbolic constants GL_FRONT and GL_BACK are accepted. The default value is GL_BACK.

Remarks

The **glCullFace** function specifies whether front- or back-facing facets are culled (as specified by *mode*) when facet culling is enabled. You enable and disable facet culling using **glEnable** and **glDisable** with the argument GL_CULL_FACE. Facets include triangles, quadrilaterals, polygons, and rectangles.

The **glFrontFace** function specifies which of the clockwise and counterclockwise facets are front-facing and back-facing.

The following functions retrieve information related to **glCullFace**:

glGet with argument GL_CULL_FACE_MODE
glIsEnabled with argument GL_CULL_FACE

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*mode* was not an accepted value.
GL_INVALID_OPERATION	**glCullFace** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glDisable**, **glEnable**, **glEnd**, **glFrontFace**, **glGet**, **glIsEnabled**

glDeleteLists Quick Info

[New - Windows 95, OEM Service Release 2]

The **glDeleteLists** function deletes a contiguous group of display lists.

```
void glDeleteLists(
  GLuint list,
  GLsizei range
);
```

Parameters

list
The integer name of the first display list to delete.

range
The number of display lists to delete.

Remarks

The **glDeleteLists** function causes a contiguous group of display lists to be deleted. The *list* parameter is the name of the first display list to be deleted, and *range* is the number of display lists to delete. All display lists *d* with $list \leq d \leq list + range - 1$ are deleted.

All storage locations allocated to the specified display lists are freed, and the names are available for reuse at a later time. Names within the range that do not have an associated display list are ignored. If *range* is zero, nothing happens.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_VALUE	*range* was negative.
GL_INVALID_OPERATION	**glDeleteLists** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glCallList**, **glCallLists**, **glEnd**, **glGenLists**, **glIsList**, **glNewList**

glDeleteTextures

[New - Windows 95, OEM Service Release 2]

The **glDeleteTextures** function deletes named textures.

```
  void glDeleteTextures(
    GLsizei n,
    const GLuint *
textures
  );
```

Parameters

n
The number of textures to be deleted.

textures
An array of textures to be deleted.

Remarks

The **glDeleteTextures** function deletes *n* textures named by the elements of the array *textures*. After a texture is deleted, it has no contents or dimensionality, and its name is free for reuse (for example, by **glGenTextures**). The **glDeleteTextures** function ignores zeros and names that do not correspond to existing textures.

If a texture that is currently bound is deleted, the binding reverts to zero (the default texture).

You cannot include calls to **glDeleteTextures** in display lists.

Note The **glDeleteTextures** function is only available in OpenGL version 1.1 or later.

The following function retrieves information related to **glDeleteTextures**:

glIsTexture

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_VALUE	*n* was a negative value.
GL_INVALID_OPERATION	**glDeleteTextures** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glAreTexturesResident, glBegin, glBindTexture, glEnd, glGenTextures, glGet, glGetTexParameter, glIsTexture, glPrioritizeTextures, glTexGen, glTexImage1D, glTexImage2D, glTexParameter

glDepthFunc Quick Info

[New - Windows 95, OEM Service Release 2]

The **glDepthFunc** function specifies the value used for depth-buffer comparisons.

```
void glDepthFunc(
  GLenum func
);
```

Parameters

func

The depth-comparison function. The following symbolic constants are accepted.

Symbolic Constant	Meaning
GL_NEVER	Never passes.
GL_LESS	Passes if the incoming *z* value is less than the stored *z* value. This is the default value.
GL_EQUAL	Passes if the incoming *z* value is equal to the stored *z* value.
GL_LEQUAL	Passes if the incoming *z* value is less than or equal to the stored *z* value.
GL_GREATER	Passes if the incoming *z* value is greater than the stored *z* value.
GL_NOTEQUAL	Passes if the incoming *z* value is not equal to the stored *z* value.
GL_GEQUAL	Passes if the incoming *z* value is greater than or equal to the stored *z* value.
GL_ALWAYS	Always passes.

Remarks

The **glDepthFunc** function specifies the function used to compare each incoming pixel *z* value with the *z* value present in the depth buffer. The comparison is performed only if depth testing is enabled. (See **glEnable** with the argument GL_DEPTH_TEST.)

Initially, depth testing is disabled.

The following functions retrieve information related to **glDepthFunc**:

glGet with argument GL_DEPTH_FUNC

glIsEnabled with argument GL_DEPTH_TEST

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*func* was not an accepted value.
GL_INVALID_OPERATION	**glDepthFunc** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, glDepthRange, glEnable, glEnd, glGet, glIsEnabled

glDepthMask Quick Info

[New - Windows 95, OEM Service Release 2]

The **glDepthMask** function enables or disables writing into the depth buffer.

```
void glDepthMask(
  GLboolean flag
);
```

Parameters

flag

Specifies whether the depth buffer is enabled for writing. If *flag* is zero, depth-buffer writing is disabled. Otherwise, it is enabled. Initially, depth-buffer writing is enabled.

Remarks

The following function retrieves information related to **glDepthMask**:

glGet with argument GL_DEPTH_WRITEMASK

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glDepthMask** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glColorMask**, **glDepthFunc**, **glDepthRange**, **glEnd**, **glGet**, **glIndexMask**, **glStencilMask**

glDepthRange Quick Info

[New - Windows 95, OEM Service Release 2]

The **glDepthRange** function specifies the mapping of *z* values from normalized device coordinates to window coordinates.

```
void glDepthRange(
  GLclampd znear,
  GLclampd zfar
);
```

Parameters

znear

The mapping of the near clipping plane to window coordinates. The default value is 0.

zfar

The mapping of the far clipping plane to window coordinates. The default value is 1.

Remarks

After clipping and division by *w*, *z*-coordinates range from -1.0 to 1.0, corresponding to the near and far clipping planes. The **glDepthRange** function specifies a linear mapping of the normalized *z*-coordinates in this range to window *z*-coordinates. Regardless of the actual depth buffer implementation, window coordinate depth values are treated as though they range from 0.0 through 1.0 (like color components). Thus, the values accepted by **glDepthRange** are both clamped to this range before they are accepted.

The default mapping of 0,1 maps the near plane to 0 and the far plane to 1. With this mapping, the depth-buffer range is fully utilized.

It is not necessary that *znear* be less than *zfar*. Reverse mappings such as 1,0 are acceptable.

The following function retrieves information related to **glDepthRange**:

glGet with argument GL_DEPTH_RANGE

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glDepthRange** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glDepthFunc**, **glEnd**, **glGet**, **glViewport**

glDrawArrays

[New - Windows 95, OEM Service Release 2]

The **glDrawArrays** function specifies multiple primitives to render.

```
void glDrawArrays(
  GLenum mode,
  GLint first,
  GLsizei count
);
```

Parameters

mode
The kind of primitives to render. The following constants specify acceptable types of primitives: GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES, GL_QUAD_STRIP, GL_QUADS, and GL_POLYGON.

first
The starting index in the enabled arrays.

count
The number of indexes to render.

Remarks

With **glDrawArrays**, you can specify multiple geometric primitives to render. Instead of calling separate OpenGL functions to pass each individual vertex, normal, or color, you can specify separate arrays of vertices, normals, and colors to define a sequence of primitives (all the same kind) with a single call to **glDrawArrays**.

When you call **glDrawArrays**, *count* sequential elements from each enabled array are used to construct a sequence of geometric primitives, beginning with the *first* element. The *mode* parameter specifies what kind of primitive to construct and how to use the array elements to construct the primitives.

After **glDrawArrays** returns, the values of vertex attributes that are modified by **glDrawArrays** are undefined. For example, if GL_COLOR_ARRAY is enabled, the value of the current color is undefined after **glDrawArrays** returns. Attributes not modified by **glDrawArrays** remain defined. When GL_VERTEX_ARRAY is not enabled, no geometric primitives are generated but the attributes corresponding to enabled arrays are modified.

You can include **glDrawArrays** in display lists. When you include **glDrawArrays** in a display list, the necessary array data, determined by the array pointers and the enables, are generated and entered in the display list. The values of array pointers and enables are determined during the creation of display lists.

You can read static array data at any time. If any static array elements are modified and the array is not specified again, the results of any subsequent calls to **glDrawArrays** are undefined.

Although no error is generated when you specify an array more than once within **glBegin** and **glEnd** pairs, the results are undefined.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_VALUE	*count* was negative.
GL_INVALID_ENUM	*mode* was not an accepted value.

GL_INVALID_OPERATION	**glDrawArrays** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glArrayElement, **glBegin**, **glColorPointer**, **glEdgeFlagPointer**, **glEnd**, **glGetPointerv**, **glGetString**, **glIndexPointer**, **glNormalPointer**, **glTexCoordPointer**, **glVertexPointer**

glDrawBuffer Quick Info

[New - Windows 95, OEM Service Release 2]

The **glDrawBuffer** function specifies which color buffers are to be drawn into.

```
void glDrawBuffer(
  GLenum mode
);
```

Parameters

mode

Specifies up to four color buffers to be drawn into with the following acceptable symbolic constants:

GL_NONE
: No color buffers are written.

GL_FRONT_LEFT
: Only the front-left color buffer is written.

GL_FRONT_RIGHT
: Only the front-right color buffer is written.

GL_BACK_LEFT
: Only the back-left color buffer is written.

GL_BACK_RIGHT
: Only the back-right color buffer is written.

GL_FRONT
: Only the front-left and front-right color buffers are written. If there is no front-right color buffer, only the front left-color buffer is written.

GL_BACK
: Only the back-left and back-right color buffers are written. If there is no back-right color buffer, only the back-left color buffer is written.

GL_LEFT
: Only the front-left and back-left color buffers are written. If there is no back-left color buffer, only the front-left color buffer is written.

GL_RIGHT
: Only the front-right and back-right color buffers are written. If there is no back-right color buffer, only the front-right color buffer is written.

GL_FRONT_AND_BACK
: All the front and back color buffers (front-left, front-right, back-left, back-right) are written. If there are no back color buffers, only the front-left and front-right color buffers are written. If there are no right color buffers, only the front-left and back-left color buffers are written. If there are no right or back color buffers, only the front-left color buffer is written.

GL_AUX*i*
: Only the auxiliary color buffer *i* is written; *i* is between 0 and GL_AUX_BUFFERS -1. (GL_AUX_BUFFERS is not the upper limit; use **glGet** to query the number of available aux buffers.)

The default value is GL_FRONT for single-buffered contexts, and GL_BACK for double-buffered contexts.

Remarks

When colors are written to the frame buffer, they are written into the color buffers specified by **glDrawBuffer**.

If more than one color buffer is selected for drawing, then blending or logical operations are computed

and applied independently for each color buffer and can produce different results in each buffer.

Monoscopic contexts include only left buffers, and stereoscopic contexts include both left and right buffers. Likewise, single-buffered contexts include only front buffers, and double-buffered contexts include both front and back buffers. The context is selected at OpenGL initialization.

It is always the case that GL_AUX*i* = GL_AUX0 + *i*.

The following functions retrieve information related to the **glDrawBuffer** function:

glGet with argument GL_DRAW_BUFFER
glGet with argument GL_AUX_BUFFERS

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*mode* was not an accepted value.
GL_INVALID_OPERATION	None of the buffers indicated by *mode* existed.
GL_INVALID_OPERATION	**glDrawBuffer** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glBlendFunc**, **glColorMask**, **glEnd**, **glGet**, **glIndexMask**, **glLogicOp**, **glReadBuffer**

glDrawElements

[New - Windows 95, OEM Service Release 2]

The **glDrawElements** function renders primitives from array data.

```
void glDrawElements(
  GLenum mode,
  GLsizei count,
  GLenum type,
  const GLvoid *indices
);
```

Parameters

mode
The kind of primitives to render. It can assume one of the following symbolic values: GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES, GL_QUAD_STRIP, GL_QUADS, and GL_POLYGON.

count
The number of elements to be rendered.

type
The type of the values in indices. Must be one of GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, or GL_UNSIGNED_INT.

indices
A pointer to the location where the indices are stored.

Remarks

The **glDrawElements** function enables you to specify multiple geometric primitives with very few function calls. Instead of calling an OpenGL function to pass each individual vertex, normal, or color, you can specify separate arrays of vertexes, normals, and colors beforehand and use them to define a sequence of primitives (all of the same type) with a single call to **glDrawElements**.

When you call the **glDrawElements** function, it uses *count* sequential elements from *indices* to construct a sequence of geometric primitives. The *mode* parameter specifies what kind of primitives are constructed, and how the array elements are used to construct these primitives. If GL_VERTEX_ARRAY is not enabled, no geometric primitives are generated.

Vertex attributes that are modified by **glDrawElements** have an unspecified value after **glDrawElements** returns. For example, if GL_COLOR_ARRAY is enabled, the value of the current color is undefined after **glDrawElements** executes. Attributes that aren't modified remain unchanged.

You can include the **glDrawElements** function in display lists. When **glDrawElements** is included in a display list, the necessary array data (determined by the array pointers and enables) is also entered into the display list. Because the array pointers and enables are client-side state variables, their values affect display lists when the lists are created, not when the lists are executed.

Note The **glDrawElements** function is only available in OpenGL version 1.1 or later.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*mode* was not an accepted value.

GL_INVALID_VALUE	*count* was a negative value.
GL_INVALID_OPERATION	**glDrawElements** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glArrayElement, **glBegin**, **glColorPointer**, **glDrawArrays**, **glEdgeFlagPointer**, **glEnd**, **glGetPointerv**, **glIndexPointer**, **glNormalPointer**, **glTexCoordPointer**, **glVertexPointer**

glDrawPixels Quick Info

[New - Windows 95, OEM Service Release 2]

The **glDrawPixels** function writes a block of pixels to the frame buffer.

```
void glDrawPixels(
  GLsizei width,
  GLsizei height,
  GLenum format,
  GLenum type,
  const GLvoid *pixels
);
```

Parameters

width, height

The dimensions of the pixel rectangle that will be written into the frame buffer.

format

The format of the pixel data. Acceptable symbolic constants are:

GL_COLOR_INDEX

Each pixel is a single value, a color index.

1. The **glDrawPixels** function converts each pixel to fixed-point format, with an unspecified number of bits to the right of the binary point, regardless of the memory data type. Floating-point values convert to true fixed-point values. The **glDrawPixels** function converts signed and unsigned integer data with all fraction bits set to zero. The function converts bitmap data to either 0.0 or 1.0.
2. The **glDrawPixels** function shifts each fixed-point index left by GL_INDEX_SHIFT bits and adds it to GL_INDEX_OFFSET. If GL_INDEX_SHIFT is negative, the shift is to the right. In either case, zero bits fill otherwise unspecified bit locations in the result.
3. When in RGBA mode, **glDrawPixels** converts the resulting index to an RGBA pixel using the GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, and GL_PIXEL_MAP_I_TO_A tables. When in the color-index mode and GL_MAP_COLOR is true, the index is replaced with the value that **glDrawPixels** references in lookup table GL_PIXEL_MAP_I_TO_I.
4. Whether the lookup replacement of the index is done or not, the integer part of the index is **AND**ed with $2^b - 1$, where b is the number of bits in a color-index buffer.
5. The resulting indexes or RGBA colors are then converted to fragments by attaching the current raster position z-coordinate and texture coordinates to each pixel, and then assigning x and y window coordinates to the nth fragment such that

$$x_{(n)} = x_{(r)} + n \bmod width$$

$$y_{(n)} = y_{(r)} + \lfloor n/width \rfloor$$

where $(x_{(r)}, y_{(r)})$ is the current raster position.

6. The **glDrawPixels** function treats these pixel fragments just like the fragments generated by rasterizing points, lines, or polygons. It applies texture mapping, fog, and all the fragment operations before writing the fragments to the frame buffer.

GL_STENCIL_INDEX

Each pixel is a single value, a stencil index.

1. The **glDrawPixels** function converts it to fixed-point format, with an unspecified number of bits to the right of the binary point, regardless of the memory data type. Floating-point values convert to true fixed-point values. The **glDrawPixels** function converts signed and unsigned integer data with all fraction bits set to zero. Bitmap data converts to either 0.0 or 1.0.
2. The **glDrawPixels** function shifts each fixed-point index left by GL_INDEX_SHIFT bits, and adds it to GL_INDEX_OFFSET. If GL_INDEX_SHIFT is negative, the shift is to the right. In

either case, zero bits fill otherwise unspecified bit locations in the result.

3. If GL_MAP_STENCIL is true, the index is replaced with the value that **glDrawPixels** references in lookup table GL_PIXEL_MAP_S_TO_S.

4. Whether the lookup replacement of the index is done or not, the integer part of the index is then **AND**ed with $2^b - 1$, where b is the number of bits in the stencil buffer. The resulting stencil indexes are then written to the stencil buffer such that the nth index is written to location

$$x_{(n)} = x_{(r)} + n \bmod width$$

$$y_{(n)} = y_{(r)} + \lfloor n / width \rfloor$$

where $(x_{(r)}, y_{(r)})$ is the current raster position. Only the pixel ownership test, the scissor test, and the stencil writemask affect these writes.

GL_DEPTH_COMPONENT

Each pixel is a single-depth component.

1. The **glDrawPixels** function converts floating-point data directly to an internal floating-point format with unspecified precision. Signed integer data is mapped linearly to the internal floating-point format such that the most positive representable integer value maps to 1.0, and the most negative representable value maps to -1.0. Unsigned integer data is mapped similarly: the largest integer value maps to 1.0, and zero maps to 0.0.

2. The **glDrawPixels** function multiplies the resulting floating-point depth value by GL_DEPTH_SCALE and adds it to GL_DEPTH_BIAS. The result is clamped to the range [0,1].

3. The **glDrawPixels** function converts the resulting depth components to fragments by attaching the current raster position color or color index and texture coordinates to each pixel, and then assigning x and y window coordinates to the nth fragment such that

$$x_{(n)} = x_{(r)} + n \bmod width$$

$$y_{(n)} = y_{(r)} + \lfloor n / width \rfloor$$

where $(x_{(r)}, y_{(r)})$ is the current raster position.

4. These pixel fragments are then treated just like the fragments generated by rasterizing points, lines, or polygons. The **glDrawPixels** function applies texture mapping, fog, and all the fragment operations before writing the fragments to the frame buffer.

GL_RGBA

Each pixel is a four-component group in this order: red, green, blue, alpha.

1. The **glDrawPixels** function converts floating-point values directly to an internal floating-point format with unspecified precision. Signed integer values are mapped linearly to the internal floating-point format such that the most positive representable integer value maps to 1.0, and the most negative representable value maps to - 1.0. Unsigned integer data is mapped similarly: the largest integer value maps to 1.0, and zero maps to 0.0.

2. The **glDrawPixels** function multiplies the resulting floating-point color values by GL_c_SCALE and adds them to GL_c_BIAS, where c is RED, GREEN, BLUE, and ALPHA for the respective color components. The results are clamped to the range [0,1].

3. If GL_MAP_COLOR is true, **glDrawPixels** scales each color component by the size of lookup table GL_PIXEL_MAP_c_TO_c, and then replaces the component by the value that it references in that table; c is R, G, B, or A, respectively.

4. The **glDrawPixels** function converts the resulting RGBA colors to fragments by attaching the current raster position z-coordinate and texture coordinates to each pixel, then assigning x and y window coordinates to the nth fragment such that

$$x_{(n)} = x_{(r)} + n \bmod width$$

$$y_{(n)} = y_{(r)} + \lfloor n / width \rfloor$$

where $(x_{(r)}, y_{(r)})$ is the current raster position.

5. These pixel fragments are then treated just like the fragments generated by rasterizing points, lines, or polygons. The **glDrawPixels** function applies texture mapping, fog, and all the fragment operations before writing the fragments to the frame buffer.

GL_RED
Each pixel is a single red component.

The **glDrawPixels** function converts this component to the internal floating-point format in the same way that the red component of an RGBA pixel is, and then converts it to an RGBA pixel with green and blue set to 0.0, and alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.

GL_GREEN
Each pixel is a single green component.

The **glDrawPixels** function converts this component to the internal floating-point format in the same way that the green component of an RGBA pixel is, and then converts it to an RGBA pixel with red and blue set to 0.0, and alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.

GL_BLUE
Each pixel is a single blue component.

The **glDrawPixels** function converts this component to the internal floating-point format in the same way that the blue component of an RGBA pixel is, and then converts it to an RGBA pixel with red and green set to 0.0, and alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.

GL_ALPHA
Each pixel is a single alpha component.

The **glDrawPixels** function converts this component to the internal floating-point format in the same way that the alpha component of an RGBA pixel is, and then converts it to an RGBA pixel with red, green, and blue set to 0.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.

GL_RGB
Each pixel is a group of three components in this order: red, green, blue.

The **glDrawPixels** function converts each component to the internal floating-point format in the same way that the red, green, and blue components of an RGBA pixel are. The color triple is converted to an RGBA pixel with alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.

GL_LUMINANCE
Each pixel is a single luminance component.

The **glDrawPixels** function converts this component to the internal floating-point format in the same way that the red component of an RGBA pixel is, and then converts it to an RGBA pixel with red, green, and blue set to the converted luminance value, and alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.

GL_LUMINANCE_ALPHA
Each pixel is a group of two components in this order: luminance, alpha.

The **glDrawPixels** function converts the two components to the internal floating-point format in the same way that the red component of an RGBA pixel is, and then converts them to an RGBA pixel with red, green, and blue set to the converted luminance value, and alpha set to the converted alpha value. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.

GL_BGR_EXT
Each pixel is a group of three components in this order: blue, green, red.

GL_BGR_EXT provides a format that matches the memory layout of Windows device-independent bitmaps (DIBs). Thus your applications can use the same data with Win32 function calls and OpenGL pixel function calls.

GL_BGRA_EXT
Each pixel is a group of four components in this order: blue, green, red, alpha.

GL_BGRA_EXT provides a format that matches the memory layout of Windows device-independent bitmaps (DIBs). Thus your applications can use the same data with Win32 function calls and OpenGL pixel function calls.

type

The data type for *pixels*. The following symbolic constants are accepted: GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, and GL_FLOAT.

The following table summarizes the meaning of the valid constants for the *type* parameter.

Type	Meaning
GL_UNSIGNED_BYTE	Unsigned 8-bit integer
GL_BYTE	Signed 8-bit integer
GL_BITMAP	Single bits in unsigned 8-bit integers
GL_UNSIGNED_SHORT	Unsigned 16-bit integer
GL_SHORT	Signed 16-bit integer
GL_UNSIGNED_INT	Unsigned 32-bit integer
GL_INT	32-bit integer
GL_FLOAT	Single-precision floating-point

pixels

A pointer to the pixel data.

Remarks

The **glDrawPixels** function reads pixel data from memory and writes it into the frame buffer relative to the current raster position. Use **glRasterPos** to set the current raster position, and use **glGet** with argument GL_CURRENT_RASTER_POSITION to query the raster position.

Several parameters define the encoding of pixel data in memory and control the processing of the pixel data before it is placed in the frame buffer. These parameters are set with four functions: **glPixelStore**, **glPixelTransfer**, **glPixelMap**, and **glPixelZoom**. This topic describes the effects on **glDrawPixels** of many, but not all, of the parameters specified by these four functions.

Data is read from *pixels* as a sequence of signed or unsigned bytes, signed or unsigned shorts, signed or unsigned integers, or single-precision floating-point values, depending on *type*. Each of these bytes, shorts, integers, or floating-point values is interpreted as one color or depth component, or one index, depending on *format*. Indexes are always treated individually. Color components are treated as groups of one, two, three, or four values, again based on *format*. Both individual indexes and groups of components are referred to as pixels. If *type* is GL_BITMAP, the data must be unsigned bytes, and *format* must be either GL_COLOR_INDEX or GL_STENCIL_INDEX. Each unsigned byte is treated as eight 1-bit pixels, with bit ordering determined by GL_UNPACK_LSB_FIRST (see **glPixelStore**).

The *width* by *height* pixels are read from memory, starting at location *pixels*. By default, these pixels are taken from adjacent memory locations, except that after all *width* pixels are read, the read pointer is advanced to the next 4-byte boundary. The **glPixelStore** function specifies the 4-byte row alignment with argument GL_UNPACK_ALIGNMENT, and you can set it to 1, 2, 4, or 8 bytes. Other pixel store parameters specify different read pointer advancements, both before the first pixel is read, and after all *width* pixels are read.

The **glPixelStore** function operates on each of the *width*-by-*height* pixels that it reads from memory in the same way, based on the values of several parameters specified by **glPixelTransfer** and **glPixelMap**. The details of these operations, as well as the target buffer into which the pixels are drawn, are specific to the format of the pixels, as specified by *format*.

The rasterization described thus far assumes pixel zoom factors of 1.0. If you use **glPixelZoom** to change the *x* and *y* pixel zoom factors, pixels are converted to fragments as follows. If $(x_{(r)}, y_{(r)})$ is the current raster position, and a given pixel is in the *n* column and *m* row of the pixel rectangle, then fragments are generated for pixels whose centers are in the rectangle with corners at

$(x_{(r)} + zoom_{(x)}\ n,\ y_{(r)} + zoom_{(y)}\ m)$
$(x_{(r)} + zoom_{(x)}\ (n + 1),\ y_{(r)} + zoom_{(y)}\ (m + 1))$

where $zoom_{(x)}$ is the value of GL_ZOOM_X and $zoom_{(y)}$ is the value of GL_ZOOM_Y.

The following functions retrieve information related to **glDrawPixels**:

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_VALUE	Either *width* or *height* was negative.
GL_INVALID_ENUM	Either *format* or *type* was not an accepted value.
GL_INVALID_OPERATION	*format* was GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_BGR_EXT, GL_BGRA_EXT, GL_LUMINANCE, or GL_LUMINANCE_ALPHA, and OpenGL was in color-index mode.
GL_INVALID_ENUM	*type* was GL_BITMAP and *format* was not either GL_COLOR_INDEX or GL_STENCIL_INDEX.
GL_INVALID_OPERATION	*format* was GL_STENCIL_INDEX and there was no stencil buffer.
GL_INVALID_OPERATION	**glDrawPixels** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glAlphaFunc, **glBegin**, **glBlendFunc**, **glCopyPixels**, **glDepthFunc**, **glEnd**, **glGet**, **glLogicOp**, **glPixelMap**, **glPixelStore**, **glPixelTransfer**, **glPixelZoom**, **glRasterPos**, **glReadPixels**, **glScissor**, **glStencilFunc**

glEdgeFlag, glEdgeFlagv

[New - Windows 95, OEM Service Release 2]

The **glEdgeFlag** and **glEdgeFlagv** functions flag edges as either boundary or nonboundary.

```
void glEdgeFlag(
  GLboolean flag
);
```

Parameters

flag

In **glEdgeFlag**, specifies the current edge flag value, either TRUE or FALSE.

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

Parameters

flag

In **glEdgeFlagv**, specifies a pointer to an array that contains a single Boolean element, which replaces the current edge flag value.

Remarks

Each vertex of a polygon, separate triangle, or separate quadrilateral specified between a **glBegin**/**glEnd** pair is marked as the start of either a boundary or nonboundary edge. If the current edge flag is TRUE when the vertex is specified, the vertex is marked as the start of a boundary edge. If the current edge flag is FALSE, the vertex is marked as the start of a nonboundary edge. The **glEdgeFlag** function sets the edge flag to TRUE if *flag* is nonzero, FALSE otherwise.

The vertices of connected triangles and connected quadrilaterals are always marked as boundary, regardless of the value of the edge flag.

Boundary and nonboundary edge flags on vertices are significant only if GL_POLYGON_MODE is set to GL_POINT or GL_LINE. See **glPolygonMode**.

Initially, the edge flag bit is TRUE.

The current edge flag can be updated at any time. In particular, **glEdgeFlag** can be called between a call to **glBegin** and the corresponding call to **glEnd**.

The following function retrieves information related to the **glEdgeFlag** function:

glGet with argument GL_EDGE_FLAG

See Also

glBegin, **glEnd**, **glGet**, **glPolygonMode**

glEdgeFlagPointer

[New - Windows 95, OEM Service Release 2]

The **glEdgeFlagPointer** function defines an array of edge flags.

```
void glEdgeFlagPointer(
  GLsizei stride,
  GLsizei count,
  const GLboolean * pointer
);
```

Parameters

stride
The byte offset between consecutive edge flags. When *stride* is zero, the edge flags are tightly packed in the array.

count
The number of edge flags, counting from the first, that are static.

pointer
A pointer to the first edge flag in the array.

Remarks

The **glEdgeFlagPointer** function specifies the location and data of an array of Boolean edge flags to use when rendering. The *stride* parameter determines the byte offset from one edge flag to the next, which enables the packing of vertices and attributes in a single array or storage in separate arrays. In some implementations, storing the vertices and attributes in a single array can be more efficient than using separate arrays.

Starting from the first edge-flag array element, *count* indicates the total number of static elements. Your application can modify static elements, but once the elements are modified, the application must explicitly specify the array again before using it for any rendering. Nonstatic array elements are not accessed until you call **glDrawArrays** or **glArrayElement**.

Static array data can be read at any time. If any static array elements are modified and the array is not specified again, the results of any subsequent calls to **glEdgeFlagPointer** are undefined.

An edge-flag array is enabled when you specify the GL_EDGE_FLAG_ARRAY constant with **glEnableClientState**. When enabled, **glDrawArrays** or **glArrayElement** uses the edge-flag array. By default the edge-flag array is disabled.

You cannot include **glEdgeFlagPointer** in display lists.

When you specify an edge-flag array using **glEdgeFlagPointer**, the values of all the function's edge-flag array parameters are saved in a client-side state and static array elements can be cached. Because the edge-flag array parameters are in a client-side state, **glPushAttrib** and **glPopAttrib** do not save or restore their values.

Although calling **glEdgeFlagPointer** within a **glBegin**/**glEnd** pair does not generate an error, the results are undefined.

The following functions retrieve information related to the **glEdgeFlagPointer** function:

glIsEnabled with argument GL_EDGE_FLAG_ARRAY
glGet with argument GL_EDGE_FLAG_ARRAY_STRIDE
glGet with argument GL_EDGE_FLAG_ARRAY_COUNT

glGetPointerv with argument GL_EDGE_FLAG_ARRAY_POINTER

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*stride* or *count* was negative.

See Also

glArrayElement, **glBegin**, **glColorPointer**, **glDrawArrays**, **glEnableClientState**, **glEnd**, **glGet**, **glGetPointerv**, **glGetString**, **glIndexPointer**, **glIsEnabled**, **glNormalPointer**, **glPopAttrib**, **glPushAttrib**, **glTexCoordPointer**, **glVertexPointer**

glEnable, glDisable

[New - Windows 95, OEM Service Release 2]

The **glEnable** and **glDisable** functions enable or disable OpenGL capabilities.

```
void glEnable(
  GLenum cap
);

void glDisable(
  GLenum cap
);
```

Parameters

cap

A symbolic constant indicating an OpenGL capability.

For discussion of the values *cap* can take, see the following Remarks section.

Remarks

The **glEnable** and **glDisable** functions enable and disable various capabilities. Use **glIsEnabled** or **glGet** to determine the current setting of any capability.

Both **glEnable** and **glDisable** take a single argument, *cap*, which can assume one of the following values:

GL_ALPHA_TEST
: If enabled, do alpha testing. See **glAlphaFunc**.

GL_AUTO_NORMAL
: If enabled, compute surface normal vectors analytically when either GL_MAP2_VERTEX_3 or GL_MAP2_VERTEX_4 has generated vertices. See **glMap2**.

GL_BLEND
: If enabled, blend the incoming RGBA color values with the values in the color buffers. See **glBlendFunc**.

GL_CLIP_PLANE*i*
: If enabled, clip geometry against user-defined clipping plane *i*. See **glClipPlane**.

GL_COLOR_MATERIAL
: If enabled, have one or more material parameters track the current color. See **glColorMaterial**.

GL_CULL_FACE
: If enabled, cull polygons based on their winding in window coordinates. See **glCullFace**.

GL_DEPTH_TEST
: If enabled, do depth comparisons and update the depth buffer. See **glDepthFunc** and **glDepthRange**.

GL_DITHER
: If enabled, dither color components or indexes before they are written to the color buffer.

GL_FOG
: If enabled, blend a fog color into the post-texturing color. See **glFog**.

GL_LIGHT*i*
: If enabled, include light *i* in the evaluation of the lighting equation. See **glLightModel** and **glLight**.

GL_LIGHTING
: If enabled, use the current lighting parameters to compute the vertex color or index. If disabled, associate the current color or index with each vertex. See **glMaterial**, **glLightModel**, and **glLight**.

GL_LINE_SMOOTH

If enabled, draw lines with correct filtering. If disabled, draw aliased lines. See **glLineWidth**.

GL_LINE_STIPPLE
If enabled, use the current line stipple pattern when drawing lines. See **glLineStipple**.

GL_LOGIC_OP
If enabled, apply the currently selected logical operation to the incoming and color-buffer indexes. See **glLogicOp**.

GL_MAP1_COLOR_4
If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** generate RGBA values. See also **glMap1**.

GL_MAP1_INDEX
If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** generate color indexes. See also **glMap1**.

GL_MAP1_NORMAL
If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** generate normals. See also **glMap1**.

GL_MAP1_TEXTURE_COORD_1
If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** generate *s* texture coordinates. See also **glMap1**.

GL_MAP1_TEXTURE_COORD_2
If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** generate *s* and *t* texture coordinates. See also **glMap1**.

GL_MAP1_TEXTURE_COORD_3
If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** generate *s*, *t*, and *r* texture coordinates. See also **glMap1**.

GL_MAP1_TEXTURE_COORD_4
If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** generate *s*, *t*, *r*, and *q* texture coordinates. See also **glMap1**.

GL_MAP1_VERTEX_3
If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** generate *x*, *y*, and *z* vertex coordinates. See also **glMap1**.

GL_MAP1_VERTEX_4
If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** generate homogeneous *x*, *y*, *z*, and *w* vertex coordinates. See also **glMap1**.

GL_MAP2_COLOR_4
If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** generate RGBA values. See also **glMap2**.

GL_MAP2_INDEX
If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** generate color indexes. See also **glMap2**.

GL_MAP2_NORMAL
If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** generate normals. See also **glMap2**.

GL_MAP2_TEXTURE_COORD_1
If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** generate *s* texture coordinates. See also **glMap2**.

GL_MAP2_TEXTURE_COORD_2
If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** generate *s* and *t* texture coordinates. See also **glMap2**.

GL_MAP2_TEXTURE_COORD_3
If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** generate *s*, *t*, and *r* texture coordinates. See also **glMap2**.

GL_MAP2_TEXTURE_COORD_4

If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** generate *s*, *t*, *r*, and *q* texture coordinates. See also **glMap2**.

GL_MAP2_VERTEX_3
If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** generate *x*, *y*, and *z* vertex coordinates. See also **glMap2**.

GL_MAP2_VERTEX_4
If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** generate homogeneous *x*, *y*, *z*, and *w* vertex coordinates. See also **glMap2**.

GL_NORMALIZE
If enabled, normal vectors specified with **glNormal** are scaled to unit length after transformation. See **glNormal**.

GL_POINT_SMOOTH
If enabled, draw points with proper filtering. If disabled, draw aliased points. See **glPointSize**.

GL_POLYGON_SMOOTH
If enabled, draw polygons with proper filtering. If disabled, draw aliased polygons. See **glPolygonMode**.

GL_POLYGON_STIPPLE
If enabled, use the current polygon stipple pattern when rendering polygons. See **glPolygonStipple**.

GL_SCISSOR_TEST
If enabled, discard fragments that are outside the scissor rectangle. See **glScissor**.

GL_STENCIL_TEST
If enabled, do stencil testing and update the stencil buffer. See **glStencilFunc** and **glStencilOp**.

GL_TEXTURE_1D
If enabled, one-dimensional texturing is performed (unless two-dimensional texturing is also enabled). See **glTexImage1D**.

GL_TEXTURE_2D
If enabled, two-dimensional texturing is performed. See **glTexImage2D**.

GL_TEXTURE_GEN_Q
If enabled, the *q* texture coordinate is computed using the texture-generation function defined with **glTexGen**. Otherwise, the current *q* texture coordinate is used.

GL_TEXTURE_GEN_R
If enabled, the *r* texture coordinate is computed using the texture generation function defined with **glTexGen**. If disabled, the current *r* texture coordinate is used.

GL_TEXTURE_GEN_S
If enabled, the *s* texture coordinate is computed using the texture generation function defined with **glTexGen**. If disabled, the current *s* texture coordinate is used.

GL_TEXTURE_GEN_T
If enabled, the *t* texture coordinate is computed using the texture generation function defined with **glTexGen**. If disabled, the current *t* texture coordinate is used.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*cap* was not one of the values listed in the preceding Remarks section.
GL_INVALID_OPERATION	**glEnable** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glAlphaFunc, glArrayElement, glBegin, glBlendFunc, glClipPlane, glColorMaterial, glColorPointer, glCullFace, glDepthFunc, glDepthRange, glDrawArrays, glEdgeFlagPointer, glEnd, glEvalCoord1, glEvalMesh1, glEvalPoint1, glFog, glGet, glIndexPointer, glIsEnabled, glLight, glLightModel, glLineWidth, glLineStipple, glLogicOp, glMap1, glMap2, glMaterial, glNormal, glNormalPointer, glPointSize, glPolygonMode, glPolygonStipple, glScissor, glStencilFunc, glStencilOp, glTexCoordPointer, glTexGen, glTexImage1D, glTexImage2D

glEnableClientState, glDisableClientState

[New - Windows 95, OEM Service Release 2]

The **glEnableClientState** and **glDisableClientState** functions enable and disable arrays respectively.

```
void glEnableClientState(
  GLenum array
);

void glDisableClientState(
  GLenum array
);
```

Parameters

array
A symbolic constant for the array you want to enable or disable. This parameter can assume one of the following values:

GL_COLOR_ARRAY
If enabled, use color arrays with calls to **glArrayElement**, **glDrawElements**, or **glDrawArrays**. See also **glColorPointer**.

GL_EDGE_FLAG_ARRAY
If enabled, use edge flag arrays with calls to **glArrayElement**, **glDrawElements**, or **glDrawArrays**. See also **glEdgeFlagPointer**.

GL_INDEX_ARRAY
If enabled, use index arrays with calls to **glArrayElement**, **glDrawElements**, or **glDrawArrays**. See also **glIndexPointer**.

GL_NORMAL_ARRAY
If enabled, use normal arrays with calls to **glArrayElement**, **glDrawElements**, or **glDrawArrays**. See also **glNormalPointer**.

GL_TEXTURE_COORD_ARRAY
If enabled, use texture coordinate arrays with calls to **glArrayElement**, **glDrawElements**, or **glDrawArrays**. See also **glTexCoordPointer**.

GL_VERTEX_ARRAY
If enabled, use vertex arrays with calls to **glArrayElement**, **glDrawElements**, or **glDrawArrays**. See also **glVertexPointer**.

Remarks

The **glEnableClientState** and **glDisableClientState** functions enable and disable various individual arrays. Use **glIsEnabled** or **glGet** to determine the current setting of any capability.

Calling **glEnableClientState** and **glDisableClientState** between calls to **glBegin** and the corresponding call to **glEnd** can cause an error. If no error is generated, the behavior is undefined.

Note The **glEnableClientState** and **glDisableClientState** functions are only available in OpenGL version 1.1 or later.

Error Codes

The following is the error code generated and its condition.

Error Code	Condition
GL_INVALID_ENUM	*array* was not an accepted value.

See Also

glArrayElement, **glBegin**, **glColorPointer**, **glDrawArrays**, **glDrawElements**, **glEdgeFlagPointer**, **glEnable**, **glEnd**, **glGetPointerv**, **glIndexPointer**, **glInterleavedArrays**, **glNormalPointer**, **glTexCoordPointer**, **glVertexPointer**

glEvalCoord

[New - Windows 95, OEM Service Release 2]

glEvalCoord1d, glEvalCoord1f, glEvalCoord2d, glEvalCoord2f, glEvalCoord1dv, glEvalCoord1fv, glEvalCoord2dv, glEvalCoord2fv

These functions evaluate enabled one- and two-dimensional maps.

```
void glEvalCoord1d(
  GLdouble u
);

void glEvalCoord1f(
  GLfloat u
);

void glEvalCoord2d(
  GLdouble u,
  GLdouble v
);

void glEvalCoord2f(
  GLfloat u,
  GLfloat v
);
```

Parameters

u

In **glEvalCoord1d**, **glEvalCoord1f**, **glEvalCoord2d**, and **glEvalCoord2f**, specifies a value that is the domain coordinate *u* to the basis function defined in a previous **glMap1** or **glMap2** function.

v

A value that is the domain coordinate *v* to the basis function defined in a previous **glMap2** function. This argument is not present in a **glEvalCoord1** function.

```
void glEvalCoord1dv(
  const GLdouble * u
);

void glEvalCoord1fv(
  const GLfloat * u
);

void glEvalCoord2dv(
  const GLdouble * u
);

void glEvalCoord2fv(
  const GLfloat * u
);
```

Parameters

u

In **glEvalCoord1dv**, **glEvalCoord1fv**, **glEvalCoord2dv**, and **glEvalCoord2fv**, specifies a pointer to an array containing either one or two domain coordinates. The first coordinate is *u*. The second coordinate is *v*, which is present only in **glEvalCoord2** versions.

Remarks

The **glEvalCoord1** function evaluates enabled one-dimensional maps at argument *u*. The **glEvalCoord2** function does the same for two-dimensional maps using two domain values, *u* and *v*. Maps are defined with **glMap1** and **glMap2**, and are enabled and disabled with **glEnable** and **glDisable**.

When one of the **glEvalCoord** functions is issued, all currently enabled maps of the indicated dimension are evaluated. Then, for each enabled map, it is as if the corresponding OpenGL function were issued with the computed value. That is, if GL_MAP1_INDEX or GL_MAP2_INDEX is enabled, a **glIndex** function is simulated. If GL_MAP1_COLOR_4 or GL_MAP2_COLOR_4 is enabled, a **glColor** function is simulated. If GL_MAP1_NORMAL or GL_MAP2_NORMAL is enabled, a normal vector is produced, and if any of GL_MAP1_TEXTURE_COORD_1, GL_MAP1_TEXTURE_COORD_2, GL_MAP1_TEXTURE_COORD_3, GL_MAP1_TEXTURE_COORD_4, GL_MAP2_TEXTURE_COORD_1, GL_MAP2_TEXTURE_COORD_2, GL_MAP2_TEXTURE_COORD_3, and GL_MAP2_TEXTURE_COORD_4 is enabled, then an appropriate **glTexCoord** function is simulated.

OpenGL uses evaluated values instead of current values for those evaluations that are enabled, and current values otherwise, for color, color index, normal, and texture coordinates. However, the evaluated values do not update the current values. Thus, if **glVertex** functions are interspersed with **glEvalCoord** functions, the color, normal, and texture coordinates associated with the **glVertex** functions are not affected by the values generated by the **glEvalCoord** functions, but only by the most recent **glColor**, **glIndex**, **glNormal**, and **glTexCoord** functions.

No functions are issued for maps that are not enabled. If more than one texture evaluation is enabled for a particular dimension (for example, GL_MAP2_TEXTURE_COORD_1 and GL_MAP2_TEXTURE_COORD_2), then only the evaluation of the map that produces the larger number of coordinates (in this case, GL_MAP2_TEXTURE_COORD_2) is carried out. GL_MAP1_VERTEX_4 overrides GL_MAP1_VERTEX_3, and GL_MAP2_VERTEX_4 overrides GL_MAP2_VERTEX_3, in the same manner. If neither a three- nor four-component vertex map is enabled for the specified dimension, **glEvalCoord** is ignored.

If automatic normal generation is enabled, **glEvalCoord2** calls **glEnable** with argument GL_AUTO_NORMAL to generate surface normals analytically, regardless of the contents or enabling of the GL_MAP2_NORMAL map. Let

{ewc msdncd, EWGraphic, bsd23543 0 /a "SDK.BMP"}

The generated normal **n** is

{ewc msdncd, EWGraphic, bsd23543 1 /a "SDK.BMP"}

If automatic normal generation is disabled, the corresponding normal map GL_MAP2_NORMAL, if enabled, is used to produce a normal. If neither automatic normal generation nor a normal map is enabled, no normal is generated for **glEvalCoord2** functions.

The following functions retrieve information related to the **glEvalCoord** functions:

- **glIsEnabled** with argument GL_MAP1_VERTEX_3
- **glIsEnabled** with argument GL_MAP1_VERTEX_4
- **glIsEnabled** with argument GL_MAP1_INDEX
- **glIsEnabled** with argument GL_MAP1_COLOR_4
- **glIsEnabled** with argument GL_MAP1_NORMAL
- **glIsEnabled** with argument GL_MAP1_TEXTURE_COORD_1
- **glIsEnabled** with argument GL_MAP1_TEXTURE_COORD_2
- **glIsEnabled** with argument GL_MAP1_TEXTURE_COORD_3

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_4
glIsEnabled with argument GL_MAP2_VERTEX_3
glIsEnabled with argument GL_MAP2_VERTEX_4
glIsEnabled with argument GL_MAP2_INDEX
glIsEnabled with argument GL_MAP2_COLOR_4
glIsEnabled with argument GL_MAP2_NORMAL
glIsEnabled with argument GL_MAP2_TEXTURE_COORD_1
glIsEnabled with argument GL_MAP2_TEXTURE_COORD_2
glIsEnabled with argument GL_MAP2_TEXTURE_COORD_3
glIsEnabled with argument GL_MAP2_TEXTURE_COORD_4
glIsEnabled with argument GL_AUTO_NORMAL
glGetMap

See Also

glBegin, **glColor**, **glDisable**, **glEnable**, **glEvalMesh**, **glEvalPoint**, **glGetMap**, **glIndex**, **glIsEnabled**, **glMap1**, **glMap2**, **glMapGrid**, **glNormal**, **glTexCoord**, **glVertex**

glEvalMesh1, glEvalMesh2

[New - Windows 95, OEM Service Release 2]

The **glEvalMesh1** and **glEvalMesh2** functions compute a one- or two-dimensional grid of points or lines.

```
void glEvalMesh1(
  GLenum mode,
  GLint i1,
  GLint i2
);
```

Parameters

mode
In **glEvalMesh1**, specifies whether to compute a one-dimensional mesh of points or lines. The following symbolic constants are accepted: GL_POINT and GL_LINE.

i1, i2
The first and last integer values for grid domain variable *i*.

```
void glEvalMesh2(
  GLenum mode,
  GLint i1,
  GLint i2,
  GLint j1,
  GLint j2
);
```

Parameters

mode
In **glEvalMesh2**, specifies whether to compute a two-dimensional mesh of points, lines, or polygons. The following symbolic constants are accepted: GL_POINT, GL_LINE, and GL_FILL.

i1, i2
The first and last integer values for grid domain variable *i*.

j1, j2
The first and last integer values for grid domain variable *j*.

Remarks

Use **glMapGrid** and **glEvalMesh** in tandem to efficiently generate and evaluate a series of evenly spaced map domain values. The **glEvalMesh** function steps through the integer domain of a one- or two-dimensional grid, whose range is the domain of the evaluation maps specified by **glMap1** and **glMap2**. The *mode* parameter determines whether the resulting vertices are connected as points, lines, or filled polygons.

In the one-dimensional case, **glEvalMesh1**, the mesh is generated as if the following code fragment were executed:

```
glBegin(type) ;
for (i = i1; i <= i2; i += 1)
    glEvalCoord1(i·Δu + u(1) )
glEnd( );
```

where

$\Delta u = (u_{(2)} - u_{(1)}) / n$

and n, $u_{(1)}$, and $u_{(2)}$ are the arguments to the most recent **glMapGrid1** function. The *type* parameter is GL_POINTS if *mode* is GL_POINT, or GL_LINES if *mode* is GL_LINE. The one absolute numeric requirement is that if $i = n$, then the value computed from $i \cdot \Delta u + u_{(1)}$ is exactly $u_{(2)}$.

In the two-dimensional case, **glEvalMesh2**, let

$\Delta u = (u_{(2)} - u_{(1)})/n$
$\Delta v = (v_{(2)} - v_{(1)})/m$,

where n, $u_{(1)}$, $u_{(2)}$, m, $v_{(1)}$, and $v_{(2)}$ are the arguments to the most recent **glMapGrid2** function. Then, if *mode* is GL_FILL, **glEvalMesh2** is equivalent to:

```
for (j = j1; j < j2; j += 1) {
    glBegin(GL_QUAD_STRIP);
    for (i = i1; i <= i2; i += 1) {
        glEvalCoord2(i·Δ u + u(1) , j ·Δ v + v(1) );
        glEvalCoord2(i·Δ u + u(1) , (j+1) ·Δ v + v(1) );
    }
    glEnd( );
}
```

If *mode* is GL_LINE, then a call to **glEvalMesh2** is equivalent to:

```
for (j = j1; j <= j2; j += 1) {
    glBegin(GL_LINE_STRIP);
    for (i = i1; i <= i2; i += 1)
        glEvalCoord2(i·Δ u + u(1) , j·Δ v + v(1) );
    glEnd( );
}
    for (i = i1; i <= i2; i += 1) {
    glBegin(GL_LINE_STRIP);
    for (j = j1; j <= j1; j += 1)
        glEvalCoord2(i·Δ u + u(1) , j·Δ v + v(1) );
    glEnd( );
}
```

And finally, if *mode* is GL_POINT, then a call to **glEvalMesh2** is equivalent to:

```
glBegin(GL_POINTS);
for (j = j1; j <= j2; j += 1) {
    for (i = i1; i <= i2; i += 1) {
        glEvalCoord2(i·Δ u + u(1) , j·Δ v + v(1) );
    }
}
glEnd( );
```

In all three cases, the only absolute numeric requirements are that if $i = n$, then the value computed from $i \cdot \Delta u + u_{(1)}$ is exactly $u_{(2)}$, and if $j = m$, then the value computed from $j \cdot \Delta v + v_{(1)}$ is exactly $v_{(2)}$.

The following functions retrieve information relating to **glEvalMesh**:

glGet with argument GL_MAP1_GRID_DOMAIN
glGet with argument GL_MAP2_GRID_DOMAIN
glGet with argument GL_MAP1_GRID_SEGMENTS
glGet with argument GL_MAP2_GRID_SEGMENTS

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*mode* was not an accepted value.
GL_INVALID_OPERATION	**glEvalMesh** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glEvalCoord**, **glEvalPoint**, **glMap1**, **glMap2**, **glMapGrid**

glEvalPoint1, glEvalPoint2

[New - Windows 95, OEM Service Release 2]

The **glEvalPoint1** and **glEvalPoint2** functions generate and evaluate a single point in a mesh.

```
void glEvalPoint1(
  GLint i
);

void glEvalPoint2(
  GLint i,
  GLint j
);
```

Parameters

i
The integer value for grid domain variable *i*.

j
The integer value for grid domain variable *j* (**glEvalPoint2** only).

Remarks

The **glMapGrid** and **glEvalMesh** functions are used in tandem to efficiently generate and evaluate a series of evenly spaced map domain values. You can use **glEvalPoint** to evaluate a single grid point in the same gridspace that is traversed by **glEvalMesh**. Calling **glEvalPoint1** is equivalent to calling

glEvalCoord1(i·$\Delta u + u_{(1)}$);

where

$\Delta u = (u_{(2)} - u_{(1)}) / n$

and n, $u_{(1)}$, and $u_{(2)}$ are the arguments to the most recent **glMapGrid1** function. The one absolute numeric requirement is that if $i = n$, then the value computed from $i \cdot \Delta u + u_{(1)}$ is exactly $u_{(2)}$.

In the two-dimensional case, **glEvalPoint2**, let

$\Delta u = (u_{(2)} - u_{(1)})/n$
$\Delta v = (v_{(2)} - v_{(1)})/m$

where n, $u_{(1)}$, $u_{(2)}$, m, $v_{(1)}$, and $v_{(2)}$ are the arguments to the most recent **glMapGrid2** function. Then the **glEvalPoint2** function is equivalent to calling

glEvalCoord2(i·$\Delta u + u_{(1)}$, j·$\Delta v + v_{(1)}$);

The only absolute numeric requirements are that if $i = n$, then the value computed from $i \cdot \Delta u + u_{(1)}$ is exactly $u_{(2)}$, and if $j = m$, then the value computed from $j \cdot \Delta v + v_{(1)}$ is exactly $v_{(2)}$.

The following functions retrieve information relating to **glEvalPoint1** and **glEvalPoint2**:

glGet with argument GL_MAP1_GRID_DOMAIN
glGet with argument GL_MAP2_GRID_DOMAIN
glGet with argument GL_MAP1_GRID_SEGMENTS
glGet with argument GL_MAP2_GRID_SEGMENTS

See Also

glEvalCoord, glEvalMesh, glGet, glMap1, glMap2, glMapGrid

glFeedbackBuffer Quick Info

[New - Windows 95, OEM Service Release 2]

The **glFeedbackBuffer** function controls feedback mode.

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

Parameters

size
The maximum number of values that can be written into *buffer*.

type
A symbolic constant that describes the information that will be returned for each vertex. The following symbolic constants are accepted: GL_2D, GL_3D, GL_3D_COLOR, GL_3D_COLOR_TEXTURE, and GL_4D_COLOR_TEXTURE.

buffer
Returns the feedback data.

Remarks

The **glFeedbackBuffer** function controls feedback. Feedback, like selection, is an OpenGL mode. The mode is selected by calling **glRenderMode** with GL_FEEDBACK. When OpenGL is in feedback mode, no pixels are produced by rasterization. Instead, information about primitives that would have been rasterized is fed back to the application using OpenGL.

The **glFeedbackBuffer** function has three arguments:

- *buffer* is a pointer to an array of floating-point values into which feedback information is placed.
- *size* indicates the size of the array.
- *type* is a symbolic constant describing the information that is fed back for each vertex.

You must issue **glFeedbackBuffer** before feedback mode is enabled (by calling **glRenderMode** with argument GL_FEEDBACK). Setting GL_FEEDBACK without establishing the feedback buffer, or calling **glFeedbackBuffer** while OpenGL is in feedback mode, is an error.

Take OpenGL out of feedback mode by calling **glRenderMode** with a parameter value other than GL_FEEDBACK. When you do this while OpenGL is in feedback mode, **glRenderMode** returns the number of entries placed in the feedback array. The returned value never exceeds *size*. If the feedback data required more room than was available in *buffer*, **glRenderMode** returns a negative value.

While in feedback mode, each primitive that would be rasterized generates a block of values that get copied into the feedback array. If doing so would cause the number of entries to exceed the maximum, the block is partially written so as to fill the array (if there is any room left at all), and an overflow flag is set. Each block 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. Feedback occurs after polygon culling and **glPolygonMode** interpretation of polygons has taken place, so polygons that are culled are not returned in the feedback buffer. It can also occur after polygons with more than three edges are broken up into triangles, if the OpenGL implementation renders polygons by performing this decomposition.

You can insert a marker into the feedback buffer with **glPassThrough**.

The following is the grammar for the blocks of values written into the feedback buffer. Each primitive is indicated with a unique identifying value followed by some number of vertices. Polygon entries include an integer value indicating how many vertices follow. A vertex is fed back as some number of floating-point values, as determined by *type*. Colors are fed back as four values in RGBA mode and one value in color-index mode.

feedbackList ← feedbackItem feedbackList | feedbackItem
feedbackItem ← point | lineSegment | polygon | bitmap | pixelRectangle | passThru
point ← GL_POINT_TOKEN vertex
lineSegment ← GL_LINE_TOKEN vertex vertex | GL_LINE_RESET_TOKEN vertex vertex
polygon ← GL_POLYGON_TOKEN n polySpec
polySpec ← polySpec vertex | vertex vertex vertex
bitmap ← GL_BITMAP_TOKEN vertex
pixelRectangle ← GL_DRAW_PIXEL_TOKEN vertex | GL_COPY_PIXEL_TOKEN vertex
passThru ← GL_PASS_THROUGH_TOKEN value
vertex ← 2d | 3d | 3dColor | 3dColorTexture | 4dColorTexture
2d ← value value
3d ← value value value
3dColor ← value value value color
3dColorTexture ← value value value color tex
4dColorTexture ← value value value value color tex
color ← rgba | index
rgba ← value value value value
index ← value
tex ← value value value value

The *value* parameter is a floating-point number, and *n* is a floating-point integer giving the number of vertices in the polygon. The following are symbolic floating-point constants: GL_POINT_TOKEN, GL_LINE_TOKEN, GL_LINE_RESET_TOKEN, GL_POLYGON_TOKEN, GL_BITMAP_TOKEN, GL_DRAW_PIXEL_TOKEN, GL_COPY_PIXEL_TOKEN, and GL_PASS_THROUGH_TOKEN. GL_LINE_RESET_TOKEN is returned whenever the line stipple pattern is reset. The data returned as a vertex depends on the feedback *type*.

The following table gives the correspondence between *type* and the number of values per vertex; *k* is 1 in color-index mode and 4 in RGBA mode.

Type	Coordinates	Color	Texture	Total Number of Values
GL_2D	*x*, *y*			2
GL_3D	*x*, *y*, *z*			3
GL_3D_COLOR	*x*, *y*, *z*	k		3 + k
GL_3D_COLOR_TEXTURE	*x*, *y*, *z*,	k	4	7 + k
GL_4D_COLOR_TEXTURE	*x*, *y*, *z*, *w*	k	4	8 + k

Feedback vertex coordinates are in window coordinates, except *w*, which is in clip coordinates. Feedback colors are lighted, if lighting is enabled. Feedback texture coordinates are generated, if texture coordinate generation is enabled. They are always transformed by the texture matrix.

The **glFeedbackBuffer** function, when used in a display list, is not compiled into the display list but rather

is executed immediately.

The following function retrieves information related to **glFeedbackBuffer**:

glGet with argument GL_RENDER_MODE

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*type* was not an accepted value.
GL_INVALID_VALUE	*size* was negative.
GL_INVALID_OPERATION	**glFeedbackBuffer** was called while the render mode was GL_FEEDBACK, or **glRenderMode** was called with argument GL_FEEDBACK before **glFeedbackBuffer** was called at least once.
GL_INVALID_OPERATION	**glFeedbackBuffer** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glGet**, **glLineStipple**, **glPassThrough**, **glPolygonMode**, **glRenderMode**, **glSelectBuffer**

glFinish Quick Info

[New - Windows 95, OEM Service Release 2]

The **glFinish** function blocks until all OpenGL execution is complete.

```
void glFinish(
  void
);
```

Remarks

The **glFinish** function does not return until the effects of all previously called OpenGL functions are complete. Such effects include all changes to the OpenGL state, all changes to the connection state, and all changes to the frame buffer contents.

The **glFinish** function requires a round trip to the server.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glFinish** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glFlush**

glFlush Quick Info

[New - Windows 95, OEM Service Release 2]

The **glFlush** function forces execution of OpenGL functions in finite time.

```
void glFlush(
  void
);
```

Remarks

Different OpenGL implementations buffer commands in several different locations, including network buffers and the graphics accelerator itself. The **glFlush** function empties all these buffers, causing all issued commands to be executed as quickly as they are accepted by the actual rendering engine. Though this execution may not be completed in any particular time period, it does complete in finite time.

Because any OpenGL program might be executed over a network, or on an accelerator that buffers commands, be sure to call **glFlush** in all programs whenever they require that all of their previously issued commands have been completed. For example, call **glFlush** before waiting for user input that depends on the generated image.

The **glFlush** function can return at any time. It does not wait until the execution of all previously issued OpenGL functions is complete.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glFlush** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glFinish**

glFogf, glFogi, glFogfv, glFogiv

[New - Windows 95, OEM Service Release 2]

These functions specify fog parameters.

```
void glFogf(
  GLenum pname,
  GLfloat param
 );

void glFogi(
  GLenum pname,
  GLint param
 );

void glFogfv(
  GLenum pname,
  const GLfloat * params
 );

void glFogiv(
  GLenum pname,
  const GLint * params
 );
```

Parameters

pname

In **glFogf** and **glFogi**, specifies a single-valued fog parameter.

In **glFogfv** and **glFogiv**, specifies a fog parameter.

The **glFogf**, **glFogi**, **glFogfv**, and **glFogiv** functions accept the following values:

GL_FOG_MODE
: The *params* parameter is a single integer or floating-point value that specifies the equation to be used to compute the fog blend factor, *f*. Three symbolic constants are accepted: GL_LINEAR, GL_EXP, and GL_EXP2. The equations corresponding to these symbolic constants are defined in the following Remarks section. The default fog mode is GL_EXP.

GL_FOG_DENSITY
: The *params* parameter is a single integer or floating-point value that specifies *density*, the fog density used in both exponential fog equations. Only nonnegative densities are accepted. The default fog density is 1.0.

GL_FOG_START
: The *params* parameter is a single integer or floating-point value that specifies *start*, the near distance used in the linear fog equation. The default near distance is 0.0.

GL_FOG_END
: The *params* parameter is a single integer or floating-point value that specifies *end*, the far distance used in the linear fog equation. The default far distance is 1.0.

GL_FOG_INDEX
: The *params* parameter is a single integer or floating-point value that specifies $i_{(f)}$, the fog color index. The default fog index is 0.0.

The **glFogfv** and **glFogiv** functions also accept GL_FOG_COLOR:

GL_FOG_COLOR
: The *params* parameter contains four integer or floating-point values that specify $C_{(f)}$, the fog color. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly.

After conversion, all color components are clamped to the range [0,1]. The default fog color is (0,0,0,0).

param

In **glFogf** and **glFogi**, specifies the value that *pname* will be set to.

params

In **glFogfv** and **glFogiv**, specifies the value or values to be assigned to *pname*. GL_FOG_COLOR requires an array of four values. All other parameters accept an array containing only a single value.

Remarks

You enable and disable fog with **glEnable** and **glDisable**, using the argument GL_FOG. While enabled, fog affects rasterized geometry, bitmaps, and pixel blocks, but not buffer-clear operations.

The **glFog** function assigns the value or values in *params* to the fog parameter specified by *pname*.

Fog blends a fog color with each rasterized pixel fragment's posttexturing color using a blending factor *f*. Factor *f* is computed in one of three ways, depending on the fog mode. Let *z* be the distance in eye coordinates from the origin to the fragment being fogged. The equation for GL_LINEAR fog is:

{ewc msdncd, EWGraphic, bsd23544 0 /a "SDK.BMP"}

The equation for GL_EXP fog is:

{ewc msdncd, EWGraphic, bsd23544 1 /a "SDK.BMP"}

The equation for GL_EXP2 fog is:

{ewc msdncd, EWGraphic, bsd23544 2 /a "SDK.BMP"}

Regardless of the fog mode, *f* is clamped to the range [0,1] after it is computed. Then, if OpenGL is in RGBA color mode, the fragment's color $C_{(r)}$ is replaced by

{ewc msdncd, EWGraphic, bsd23544 3 /a "SDK.BMP"}

In color-index mode, the fragment's color index $i_{(r)}$ is replaced by

{ewc msdncd, EWGraphic, bsd23544 4 /a "SDK.BMP"}

The following functions retrieve information related to the **glFog** functions:

glGet with argument GL_FOG_COLOR
glGet with argument GL_FOG_INDEX
glGet with argument GL_FOG_DENSITY
glGet with argument GL_FOG_START
glGet with argument GL_FOG_END
glGet with argument GL_FOG_MODE
glIsEnabled with argument GL_FOG

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*pname* was not an accepted value.
GL_INVALID_OPERATION	**glFinish** was called between a call to **glBegin** and the corresponding call to

glEnd.

See Also

glBegin, **glDisable**, **glEnable**, **glEnd**, **glGet**, **glIsEnabled**

glFrontFace Quick Info

[New - Windows 95, OEM Service Release 2]

The **glFrontFace** function defines front- and back-facing polygons.

```
void glFrontFace(
  GLenum mode
);
```

Parameters

mode

The orientation of front-facing polygons. GL_CW and GL_CCW are accepted. The default value is GL_CCW.

Remarks

In a scene composed entirely of opaque closed surfaces, back-facing polygons are never visible. Eliminating these invisible polygons has the obvious benefit of speeding up the rendering of the image. You enable and disable elimination of back-facing polygons with **glEnable** and **glDisable** using argument GL_CULL_FACE.

The projection of a polygon to window coordinates is said to have clockwise winding if an imaginary object following the path from its first vertex, its second vertex, and so on, to its last vertex, and finally back to its first vertex, moves in a clockwise direction about the interior of the polygon. The polygon's winding is said to be counterclockwise if the imaginary object following the same path moves in a counterclockwise direction about the interior of the polygon. The **glFrontFace** function specifies whether polygons with clockwise winding in window coordinates, or counterclockwise winding in window coordinates, are taken to be front-facing. Passing GL_CCW to *mode* selects counterclockwise polygons as front-facing; GL_CW selects clockwise polygons as front-facing. By default, counterclockwise polygons are taken to be front-facing.

The following function retrieves information about **glFrontface**:

glGet with argument GL_FRONT_FACE

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*mode* was not an accepted value.
GL_INVALID_OPERATION	**glFrontFace** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glCullFace**, **glDisable**, **glEnable**, **glEnd**, **glGet**, **glLightModel**

glFrustum Quick Info

[New - Windows 95, OEM Service Release 2]

The **glFrustum** function multiplies the current matrix by a perspective matrix.

```
void glFrustum(
  GLdouble left,
  GLdouble right,
  GLdouble bottom,
  GLdouble top,
  GLdouble znear,
  GLdouble zfar
);
```

Parameters

left, right
The coordinates for the left and right vertical clipping planes.

bottom, top
The coordinates for the bottom and top horizontal clipping planes.

znear, zfar
The distances to the near and far depth clipping planes. Both distances must be positive.

Remarks

The **glFrustum** function describes a perspective matrix that produces a perspective projection. The (*left*, *bottom*, *znear*) and (*right*, *top*, *znear*) parameters specify the points on the near clipping plane that are mapped to the lower-left and upper-right corners of the window, respectively, assuming that the eye is located at (0, 0, 0). The *zfar* parameter specifies the location of the far clipping plane. Both *znear* and *zfar* must be positive. The corresponding matrix is:

{ewc msdncd, EWGraphic, bsd23544 5 /a "SDK.BMP"}

{ewc msdncd, EWGraphic, bsd23544 6 /a "SDK.BMP"}

The **glFrustum** function multiplies the current matrix by this matrix, with the result replacing the current matrix. That is, if M is the current matrix and F is the frustum perspective matrix, then **glFrustum** replaces M with M • F.

Use **glPushMatrix** and **glPopMatrix** to save and restore the current matrix stack.

Depth-buffer precision is affected by the values specified for *znear* and *zfar*. The greater the ratio of *zfar* to *znear* is, the less effective the depth buffer will be at distinguishing between surfaces that are near each other. If

{ewc msdncd, EWGraphic, bsd23544 7 /a "SDK.BMP"}

roughly $\log_{(2)} r$ bits of depth buffer precision are lost. Because *r* approaches infinity as *znear* approaches zero, you should never set *znear* to zero.

The following functions retrieve information about **glFrustum**:

glGet with argument GL_MATRIX_MODE
glGet with argument GL_MODELVIEW_MATRIX
glGet with argument GL_PROJECTION_MATRIX
glGet with argument GL_TEXTURE_MATRIX

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_VALUE	*znear* or *zfar* was not positive.
GL_INVALID_OPERATION	**glFrustum** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glGet**, **glMatrixMode**, **glMultMatrix**, **glOrtho**, **glPopMatrix**, **glPushMatrix**, **glViewport**

glGenLists Quick Info

[New - Windows 95, OEM Service Release 2]

The **glGenLists** function generates a contiguous set of empty display lists.

```
GLuint glGenLists(
  GLsizei range
);
```

Parameters

range
The number of contiguous empty display lists to be generated.

Remarks

The **glGenLists** function has one argument, *range*. It returns an integer *n* such that *range* contiguous empty display lists, named *n*, *n*+1, . . ., *n*+*range* - 1, are created. If *range* is zero, if there is no group of *range* contiguous names available, or if any error is generated, then no display lists are generated and zero is returned.

The following function retrieves information related to **glGenLists**:

glIsList

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_VALUE	*range* is negative.
GL_INVALID_OPERATION	**glGenLists** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glCallList**, **glCallLists**, **glDeleteLists**, **glEnd**, **glIsList**, **glNewList**

glGenTextures

[New - Windows 95, OEM Service Release 2]

The **glGenTextures** function generates texture names.

```
void glGenTextures(
  GLsizei n,
  GLuint * textures
);
```

Parameters

n

The number of texture names to be generated.

textures

A pointer to the first element of an array in which the generated texture names are stored.

Remarks

The **glGenTextures** function returns *n* texture names in the *textures* parameter. The texture names are not necessarily a contiguous set of integers, however, none of the returned names can have been in use immediately prior to calling the **glGenTextures** function. The generated textures assume the dimensionality of the texture target to which they are first bound with the **glBindTexture** function. Texture names returned by **glGenTextures** are not returned by subsequent calls to **glGenTextures** unless they are first deleted by calling **glDeleteTextures**.

You cannot include **glGenTextures** in display lists.

Note The **glGenTextures** function is only available in OpenGL version 1.1 or later.

The following function retrieves information related to **glGenTextures**:

glIsTexture

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_VALUE	*n* was a negative value.
GL_INVALID_OPERATION	**glGenTextures** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glBindTexture**, **glDeleteTextures**, **glEnd**, **glGet**, **glGetTexParameter**, **glIsTexture**, **glTexImage1D**, **glTexImage2D**, **glTexParameter**

glGetBooleanv, glGetDoublev, glGetFloatv, glGetIntegerv

[New - Windows 95, OEM Service Release 2]

These functions return the value or values of a selected parameter.

```
void glGetBooleanv(
  GLenum pname,
  GLboolean * params
);

void glGetDoublev(
  GLenum pname,
  GLdouble * params
);

void glGetFloatv(
  GLenum pname,
  GLfloat * params
);

void glGetIntegerv(
  GLenum pname,
  GLint * params
);
```

Parameters

pname

The parameter value to be returned. The following symbolic constants are accepted:

GL_ACCUM_ALPHA_BITS
: The *params* parameter returns one value: the number of alpha bitplanes in the accumulation buffer.

GL_ACCUM_BLUE_BITS
: The *params* parameter returns one value: the number of blue bitplanes in the accumulation buffer.

GL_ACCUM_CLEAR_VALUE
: The *params* parameter returns four values: the red, green, blue, and alpha values used to clear the accumulation buffer. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glClearAccum**.

GL_ACCUM_GREEN_BITS
: The *params* parameter returns one value: the number of green bitplanes in the accumulation buffer.

GL_ACCUM_RED_BITS
: The *params* parameter returns one value: the number of red bitplanes in the accumulation buffer.

GL_ALPHA_BIAS
: The *params* parameter returns one value: the alpha bias factor used during pixel transfers. See **glPixelTransfer**.

GL_ALPHA_BITS
: The *params* parameter returns one value: the number of alpha bitplanes in each color buffer.

GL_ALPHA_SCALE
: The *params* parameter returns one value: the alpha scale factor used during pixel transfers. See **glPixelTransfer**.

GL_ALPHA_TEST

The *params* parameter returns a single Boolean value indicating whether alpha testing of fragments is enabled. See **glAlphaFunc**.

GL_ALPHA_TEST_FUNC
The *params* parameter returns one value: the symbolic name of the alpha test function. See **glAlphaFunc**.

GL_ALPHA_TEST_REF
The *params* parameter returns one value: the reference value for the alpha test. See **glAlphaFunc**. An integer value, if requested, is linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value.

GL_ATTRIB_STACK_DEPTH
The *params* parameter returns one value: the depth of the attribute stack. If the stack is empty, zero is returned. See **glPushAttrib**.

GL_AUTO_NORMAL
The *params* parameter returns a single Boolean value indicating whether 2-D map evaluation automatically generates surface normals. See **glMap2**.

GL_AUX_BUFFERS
The *params* parameter returns one value: the number of auxiliary color buffers.

GL_BLEND
The *params* parameter returns a single Boolean value indicating whether blending is enabled. See **glBlendFunc**.

GL_BLEND_DST
The *params* parameter returns one value: the symbolic constant identifying the destination blend function. See **glBlendFunc**.

GL_BLEND_SRC
The *params* parameter returns one value: the symbolic constant identifying the source blend function. See **glBlendFunc**.

GL_BLUE_BIAS
The *params* parameter returns one value: the blue bias factor used during pixel transfers. See **glPixelTransfer**.

GL_BLUE_BITS
The *params* parameter returns one value: the number of blue bitplanes in each color buffer.

GL_BLUE_SCALE
The *params* parameter returns one value: the blue scale factor used during pixel transfers. See **glPixelTransfer**.

GL_CLIP_PLANE*i*
The *params* parameter returns a single Boolean value indicating whether the specified clipping plane is enabled. See **glClipPlane**.

GL_COLOR_CLEAR_VALUE
The *params* parameter returns four values: the red, green, blue, and alpha values used to clear the color buffers. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glClearColor**.

GL_COLOR_MATERIAL
The *params* parameter returns a single Boolean value indicating whether one or more material parameters are tracking the current color. See **glColorMaterial**.

GL_COLOR_MATERIAL_FACE
The *params* parameter returns one value: a symbolic constant indicating which materials have a parameter that is tracking the current color. See **glColorMaterial**.

GL_COLOR_MATERIAL_PARAMETER
The *params* parameter returns one value: a symbolic constant indicating which material parameters are tracking the current color. See **glColorMaterial**.

GL_COLOR_WRITEMASK
The *params* parameter returns four Boolean values: the red, green, blue, and alpha write enables for the color buffers. See **glColorMask**.

GL_CULL_FACE
The *params* parameter returns a single Boolean value indicating whether polygon culling is enabled. See **glCullFace**.

GL_CULL_FACE_MODE
The *params* parameter returns one value: a symbolic constant indicating which polygon faces are to be culled. See **glCullFace**.

GL_CURRENT_COLOR
The *params* parameter returns four values: the red, green, blue, and alpha values of the current color. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glColor**.

GL_CURRENT_INDEX
The *params* parameter returns one value: the current color index. See **glIndex**.

GL_CURRENT_NORMAL
The *params* parameter returns three values: the *x*, *y*, and *z* values of the current normal. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glNormal**.

GL_CURRENT_RASTER_COLOR
The *params* parameter returns four values: the red, green, blue, and alpha values of the current raster position. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glRasterPos**.

GL_CURRENT_RASTER_DISTANCE
The *params* parameter returns one value: the distance from the eye to the current raster position. See **glRasterPos**.

GL_CURRENT_RASTER_INDEX
The *params* parameter returns one value: the color index of the current raster position. See **glRasterPos**.

GL_CURRENT_RASTER_POSITION
The *params* parameter returns four values: the *x*, *y*, *z*, and *w* components of the current raster position. The *x*, *y*, and *z* components are in window coordinates, and *w* is in clip coordinates. See **glRasterPos**.

GL_CURRENT_RASTER_TEXTURE_COORDS
The *params* parameter returns four values: the *s*, *t*, *r*, and *q* current raster texture coordinates. See **glRasterPos** and **glTexCoord**.

GL_CURRENT_RASTER_POSITION_VALID
The *params* parameter returns a single Boolean value indicating whether the current raster position is valid. See **glRasterPos**.

GL_CURRENT_TEXTURE_COORDS
The *params* parameter returns four values: the *s*, *t*, *r*, and *q* current texture coordinates. See **glTexCoord**.

GL_DEPTH_BIAS
The *params* parameter returns one value: the depth bias factor used during pixel transfers. See **glPixelTransfer**.

GL_DEPTH_BITS
The *params* parameter returns one value: the number of bitplanes in the depth buffer.

GL_DEPTH_CLEAR_VALUE
The *params* parameter returns one value: the value that is used to clear the depth buffer. Integer

values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glClearDepth**.

GL_DEPTH_FUNC
The *params* parameter returns one value: the symbolic constant that indicates the depth comparison function. See **glDepthFunc**.

GL_DEPTH_RANGE
The *params* parameter returns two values: the near and far mapping limits for the depth buffer. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glDepthRange**.

GL_DEPTH_SCALE
The *params* parameter returns one value: the depth scale factor used during pixel transfers. See **glPixelTransfer**.

GL_DEPTH_TEST
The *params* parameter returns a single Boolean value indicating whether depth testing of fragments is enabled. See **glDepthFunc** and **glDepthRange**.

GL_DEPTH_WRITEMASK
The *params* parameter returns a single Boolean value indicating if the depth buffer is enabled for writing. See **glDepthMask**.

GL_DITHER
The *params* parameter returns a single Boolean value indicating whether dithering of fragment colors and indexes is enabled.

GL_DOUBLEBUFFER
The *params* parameter returns a single Boolean value indicating whether double buffering is supported.

GL_DRAW_BUFFER
The *params* parameter returns one value: a symbolic constant indicating which buffers are being drawn to. See **glDrawBuffer**.

GL_EDGE_FLAG
The *params* parameter returns a single Boolean value indicating whether the current edge flag is true or false. See **glEdgeFlag**.

GL_FOG
The *params* parameter returns a single Boolean value indicating whether fogging is enabled. See **glFog**.

GL_FOG_COLOR
The *params* parameter returns four values: the red, green, blue, and alpha components of the fog color. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glFog**.

GL_FOG_DENSITY
The *params* parameter returns one value: the fog density parameter. See **glFog**.

GL_FOG_END
The *params* parameter returns one value: the end factor for the linear fog equation. See **glFog**.

GL_FOG_HINT
The *params* parameter returns one value: a symbolic constant indicating the mode of the fog hint. See **glHint**.

GL_FOG_INDEX
The *params* parameter returns one value: the fog color index. See **glFog**.

GL_FOG_MODE
The *params* parameter returns one value: a symbolic constant indicating which fog equation is selected. See **glFog**.

GL_FOG_START
The *params* parameter returns one value: the start factor for the linear fog equation. See **glFog**.

GL_FRONT_FACE
The *params* parameter returns one value: a symbolic constant indicating whether clockwise or counterclockwise polygon winding is treated as front-facing. See **glFrontFace**.

GL_GREEN_BIAS
The *params* parameter returns one value: the green bias factor used during pixel transfers.

GL_GREEN_BITS
The *params* parameter returns one value: the number of green bitplanes in each color buffer.

GL_GREEN_SCALE
The *params* parameter returns one value: the green scale factor used during pixel transfers. See **glPixelTransfer**.

GL_INDEX_BITS
The *params* parameter returns one value: the number of bitplanes in each color-index buffer.

GL_INDEX_CLEAR_VALUE
The *params* parameter returns one value: the color index used to clear the color-index buffers. See **glClearIndex**.

GL_INDEX_MODE
The *params* parameter returns a single Boolean value indicating whether OpenGL is in color-index mode (TRUE) or RGBA mode (FALSE).

GL_INDEX_OFFSET
The *params* parameter returns one value: the offset added to color and stencil indexes during pixel transfers. See **glPixelTransfer**.

GL_INDEX_SHIFT
The *params* parameter returns one value: the amount that color and stencil indexes are shifted during pixel transfers. See **glPixelTransfer**.

GL_INDEX_WRITEMASK
The *params* parameter returns one value: a mask indicating which bitplanes of each color-index buffer can be written. See **glIndexMask**.

GL_LIGHT*i*
The *params* parameter returns a single Boolean value indicating whether the specified light is enabled. See **glLight** and **glLightModel**.

GL_LIGHTING
The *params* parameter returns a single Boolean value indicating whether lighting is enabled. See **glLightModel**.

GL_LIGHT_MODEL_AMBIENT
The *params* parameter returns four values: the red, green, blue, and alpha components of the ambient intensity of the entire scene. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glLightModel**.

GL_LIGHT_MODEL_LOCAL_VIEWER
The *params* parameter returns a single Boolean value indicating whether specular reflection calculations treat the viewer as being local to the scene. See **glLightModel**.

GL_LIGHT_MODEL_TWO_SIDE
The *params* parameter returns a single Boolean value indicating whether separate materials are used to compute lighting for front- and back-facing polygons. See **glLightModel**.

GL_LINE_SMOOTH
The *params* parameter returns a single Boolean value indicating whether antialiasing of lines is enabled. See **glLineWidth**.

GL_LINE_SMOOTH_HINT
The *params* parameter returns one value: a symbolic constant indicating the mode of the line antialiasing hint. See **glHint**.

GL_LINE_STIPPLE
The *params* parameter returns a single Boolean value indicating whether stippling of lines is enabled. See **glLineStipple**.

GL_LINE_STIPPLE_PATTERN
The *params* parameter returns one value: the 16-bit line stipple pattern. See **glLineStipple**.

GL_LINE_STIPPLE_REPEAT
The *params* parameter returns one value: the line stipple repeat factor. See **glLineStipple**.

GL_LINE_WIDTH
The *params* parameter returns one value: the line width as specified with **glLineWidth**.

GL_LINE_WIDTH_GRANULARITY
The *params* parameter returns one value: the width difference between adjacent supported widths for antialiased lines. See **glLineWidth**.

GL_LINE_WIDTH_RANGE
The *params* parameter returns two values: the smallest and largest supported widths for antialiased lines. See **glLineWidth**.

GL_LIST_BASE
The *params* parameter returns one value: the base offset added to all names in arrays presented to **glCallLists**. See **glListBase**.

GL_LIST_INDEX
The *params* parameter returns one value: the name of the display list currently under construction. Zero is returned if no display list is currently under construction. See **glNewList**.

GL_LIST_MODE
The *params* parameter returns one value: a symbolic constant indicating the construction mode of the display list currently being constructed. See **glNewList**.

GL_LOGIC_OP
The *params* parameter returns a single Boolean value indicating whether fragment indexes are merged into the frame buffer using a logical operation. See **glLogicOp**.

GL_LOGIC_OP_MODE
The *params* parameter returns one value: a symbolic constant indicating the selected logic operational mode. See **glLogicOp**.

GL_MAP1_COLOR_4
The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates colors. See **glMap1**.

GL_MAP1_GRID_DOMAIN
The *params* parameter returns two values: the endpoints of the 1-D maps grid domain. See **glMapGrid**.

GL_MAP1_GRID_SEGMENTS
The *params* parameter returns one value: the number of partitions in the 1-D maps grid domain. See **glMapGrid**.

GL_MAP1_INDEX
The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates color indexes. See **glMap1**.

GL_MAP1_NORMAL
The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates normals. See **glMap1**.

GL_MAP1_TEXTURE_COORD_1
The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 1-D texture coordinates. See **glMap1**.

GL_MAP1_TEXTURE_COORD_2
The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 2-D texture coordinates. See **glMap1**.

GL_MAP1_TEXTURE_COORD_3

The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 3-D texture coordinates. See **glMap1**.

GL_MAP1_TEXTURE_COORD_4
The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 4-D texture coordinates. See **glMap1**.

GL_MAP1_VERTEX_3
The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 3-D vertex coordinates. See **glMap1**.

GL_MAP1_VERTEX_4
The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 4-D vertex coordinates. See **glMap1**.

GL_MAP2_COLOR_4
The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates colors. See **glMap2**.

GL_MAP2_GRID_DOMAIN
The *params* parameter returns four values: the endpoints of the 2-D maps *i* and *j* grid domains. See **glMapGrid**.

GL_MAP2_GRID_SEGMENTS
The *params* parameter returns two values: the number of partitions in the 2-D maps *i* and *j* grid domains. See **glMapGrid**.

GL_MAP2_INDEX
The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates color indexes. See **glMap2**.

GL_MAP2_NORMAL
The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates normals. See **glMap2**.

GL_MAP2_TEXTURE_COORD_1
The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 1-D texture coordinates. See **glMap2**.

GL_MAP2_TEXTURE_COORD_2
The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 2-D texture coordinates. See **glMap2**.

GL_MAP2_TEXTURE_COORD_3
The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 3-D texture coordinates. See **glMap2**.

GL_MAP2_TEXTURE_COORD_4
The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 4-D texture coordinates. See **glMap2**.

GL_MAP2_VERTEX_3
The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 3-D vertex coordinates. See **glMap2**.

GL_MAP2_VERTEX_4
The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 4-D vertex coordinates. See **glMap2**.

GL_MAP_COLOR
The *params* parameter returns a single Boolean value indicating whether colors and color indexes are to be replaced by table lookup during pixel transfers. See **glPixelTransfer**.

GL_MAP_STENCIL
The *params* parameter returns a single Boolean value indicating whether stencil indexes are to be replaced by table lookup during pixel transfers. See **glPixelTransfer**.

GL_MATRIX_MODE
The *params* parameter returns one value: a symbolic constant indicating which matrix stack is

currently the target of all matrix operations. See **glMatrixMode**.

GL_MAX_ATTRIB_STACK_DEPTH
The *params* parameter returns one value: the maximum supported depth of the attribute stack. See **glPushAttrib**.

GL_MAX_CLIP_PLANES
The *params* parameter returns one value: the maximum number of application-defined clipping planes. See **glClipPlane**.

GL_MAX_EVAL_ORDER
The *params* parameter returns one value: the maximum equation order supported by 1-D and 2-D evaluators. See **glMap1** and **glMap2**.

GL_MAX_LIGHTS
The *params* parameter returns one value: the maximum number of lights. See **glLight**.

GL_MAX_LIST_NESTING
The *params* parameter returns one value: the maximum recursion depth allowed during display-list traversal. See **glCallList**.

GL_MAX_MODELVIEW_STACK_DEPTH
The *params* parameter returns one value: the maximum supported depth of the modelview matrix stack. See **glPushMatrix**.

GL_MAX_NAME_STACK_DEPTH
The *params* parameter returns one value: the maximum supported depth of the selection name stack. See **glPushName**.

GL_MAX_PIXEL_MAP_TABLE
The *params* parameter returns one value: the maximum supported size of a **glPixelMap** lookup table. See **glPixelMap**.

GL_MAX_PROJECTION_STACK_DEPTH
The *params* parameter returns one value: the maximum supported depth of the projection matrix stack. See **glPushMatrix**.

GL_MAX_TEXTURE_SIZE
The *params* parameter returns one value: the maximum width or height of any texture image (without borders). See **glTexImage1D** and **glTexImage2D**.

GL_MAX_TEXTURE_STACK_DEPTH
The *params* parameter returns one value: the maximum supported depth of the texture matrix stack. See **glPushMatrix**.

GL_MAX_VIEWPORT_DIMS
The *params* parameter returns two values: the maximum supported width and height of the viewport. See **glViewport**.

GL_MODELVIEW_MATRIX
The *params* parameter returns 16 values: the modelview matrix on the top of the modelview matrix stack. See **glPushMatrix**.

GL_MODELVIEW_STACK_DEPTH
The *params* parameter returns one value: the number of matrices on the modelview matrix stack. See **glPushMatrix**.

GL_NAME_STACK_DEPTH
The *params* parameter returns one value: the number of names on the selection name stack. See **glPushMatrix**.

GL_NORMALIZE
The *params* parameter returns a single Boolean value indicating whether normals are automatically scaled to unit length after they have been transformed to eye coordinates. See **glNormal**.

GL_PACK_ALIGNMENT
The *params* parameter returns one value: the byte alignment used for writing pixel data to memory. See **glPixelStore**.

GL_PACK_LSB_FIRST
The *params* parameter returns a single Boolean value indicating whether single-bit pixels being written to memory are written first to the least significant bit of each unsigned byte. See **glPixelStore**.

GL_PACK_ROW_LENGTH
The *params* parameter returns one value: the row length used for writing pixel data to memory. See **glPixelStore**.

GL_PACK_SKIP_PIXELS
The *params* parameter returns one value: the number of pixel locations skipped before the first pixel is written into memory. See **glPixelStore**.

GL_PACK_SKIP_ROWS
The *params* parameter returns one value: the number of rows of pixel locations skipped before the first pixel is written into memory. See **glPixelStore**.

GL_PACK_SWAP_BYTES
The *params* parameter returns a single Boolean value indicating whether the bytes of 2-byte and 4-byte pixel indexes and components are swapped before being written to memory. See **glPixelStore**.

GL_PERSPECTIVE_CORRECTION_HINT
The *params* parameter returns one value: a symbolic constant indicating the mode of the perspective correction hint. See **glHint**.

GL_PIXEL_MAP_A_TO_A_SIZE
The *params* parameter returns one value: the size of the alpha-to-alpha pixel-translation table. See **glPixelMap**.

GL_PIXEL_MAP_B_TO_B_SIZE
The *params* parameter returns one value: the size of the blue-to-blue pixel-translation table. See **glPixelMap**.

GL_PIXEL_MAP_G_TO_G_SIZE
The *params* parameter returns one value: the size of the green-to-green pixel-translation table. See **glPixelMap**.

GL_PIXEL_MAP_I_TO_A_SIZE
The *params* parameter returns one value: the size of the index-to-alpha pixel translation table. See **glPixelMap**.

GL_PIXEL_MAP_I_TO_B_SIZE
The *params* parameter returns one value: the size of the index-to-blue pixel translation table. See **glPixelMap**.

GL_PIXEL_MAP_I_TO_G_SIZE
The *params* parameter returns one value: the size of the index-to-green pixel-translation table. See **glPixelMap**.

GL_PIXEL_MAP_I_TO_I_SIZE
The *params* parameter returns one value: the size of the index-to-index pixel-translation table. See **glPixelMap**.

GL_PIXEL_MAP_I_TO_R_SIZE
The *params* parameter returns one value: the size of the index-to-red pixel-translation table. See **glPixelMap**.

GL_PIXEL_MAP_R_TO_R_SIZE
The *params* parameter returns one value: the size of the red-to-red pixel-translation table. See **glPixelMap**.

GL_PIXEL_MAP_S_TO_S_SIZE
The *params* parameter returns one value: the size of the stencil-to-stencil pixel translation table. See **glPixelMap**.

GL_POINT_SIZE
The *params* parameter returns one value: the point size as specified by **glPointSize**.

GL_POINT_SIZE_GRANULARITY
The *params* parameter returns one value: the size difference between adjacent supported sizes for antialiased points. See **glPointSize**.

GL_POINT_SIZE_RANGE
The *params* parameter returns two values: the smallest and largest supported sizes for antialiased points. See **glPointSize**.

GL_POINT_SMOOTH
The *params* parameter returns a single Boolean value indicating whether antialiasing of points is enabled. See **glPointSize**.

GL_POINT_SMOOTH_HINT
The *params* parameter returns one value: a symbolic constant indicating the mode of the point antialiasing hint. See **glHint**.

GL_POLYGON_MODE
The *params* parameter returns two values: symbolic constants indicating whether front-facing and back-facing polygons are rasterized as points, lines, or filled polygons. See **glPolygonMode**.

GL_POLYGON_SMOOTH
The *params* parameter returns a single Boolean value indicating whether antialiasing of polygons is enabled. See **glPolygonMode**.

GL_POLYGON_SMOOTH_HINT
The *params* parameter returns one value: a symbolic constant indicating the mode of the polygon antialiasing hint. See **glHint**.

GL_POLYGON_STIPPLE
The *params* parameter returns a single Boolean value indicating whether stippling of polygons is enabled. See **glPolygonStipple**.

GL_PROJECTION_MATRIX
The *params* parameter returns 16 values: the projection matrix on the top of the projection matrix stack. See **glPushMatrix**.

GL_PROJECTION_STACK_DEPTH
The *params* parameter returns one value: the number of matrices on the projection matrix stack. See **glPushMatrix**.

GL_READ_BUFFER
The *params* parameter returns one value: a symbolic constant indicating which color buffer is selected for reading. See **glReadPixels** and **glAccum**.

GL_RED_BIAS
The *params* parameter returns one value: the red bias factor used during pixel transfers. See **glPixelTransfer**.

GL_RED_BITS
The *params* parameter returns one value: the number of red bitplanes in each color buffer.

GL_RED_SCALE
The *params* parameter returns one value: the red scale factor used during pixel transfers. See **glPixelTransfer**.

GL_RENDER_MODE
The *params* parameter returns one value: a symbolic constant indicating whether OpenGL is in render, select, or feedback mode. See **glRenderMode**.

GL_RGBA_MODE
The *params* parameter returns a single Boolean value indicating whether OpenGL is in RGBA mode (TRUE) or color-index mode (FALSE). See **glColor**.

GL_SCISSOR_BOX
The *params* parameter returns four values: the *x* and *y* window coordinates of the scissor box, followed by its width and height. See **glScissor**.

GL_SCISSOR_TEST
The *params* parameter returns a single Boolean value indicating whether scissoring is enabled.

See **glScissor**.

GL_SHADE_MODEL
The *params* parameter returns one value: a symbolic constant indicating whether the shading mode is flat or smooth. See **glShadeModel**.

GL_STENCIL_BITS
The *params* parameter returns one value: the number of bitplanes in the stencil buffer.

GL_STENCIL_CLEAR_VALUE
The *params* parameter returns one value: the index to which the stencil bitplanes are cleared. See **glClearStencil**.

GL_STENCIL_FAIL
The *params* parameter returns one value: a symbolic constant indicating what action is taken when the stencil test fails. See **glStencilOp**.

GL_STENCIL_FUNC
The *params* parameter returns one value: a symbolic constant indicating what function is used to compare the stencil reference value with the stencil buffer value. See **glStencilFunc**.

GL_STENCIL_PASS_DEPTH_FAIL
The *params* parameter returns one value: a symbolic constant indicating what action is taken when the stencil test passes, but the depth test fails. See **glStencilOp**.

GL_STENCIL_PASS_DEPTH_PASS
The *params* parameter returns one value: a symbolic constant indicating what action is taken when the stencil test passes and the depth test passes. See **glStencilOp**.

GL_STENCIL_REF
The *params* parameter returns one value: the reference value that is compared with the contents of the stencil buffer. See **glStencilFunc**.

GL_STENCIL_TEST
The *params* parameter returns a single Boolean value indicating whether stencil testing of fragments is enabled. See **glStencilFunc** and **glStencilOp**.

GL_STENCIL_VALUE_MASK
The *params* parameter returns one value: the mask that is used to mask both the stencil reference value and the stencil buffer value before they are compared. See **glStencilFunc**.

GL_STENCIL_WRITEMASK
The *params* parameter returns one value: the mask that controls writing of the stencil bitplanes. See **glStencilMask**.

GL_STEREO
The *params* parameter returns a single Boolean value indicating whether stereo buffers (left and right) are supported.

GL_SUBPIXEL_BITS
The *params* parameter returns one value: an estimate of the number of bits of subpixel resolution that are used to position rasterized geometry in window coordinates.

GL_TEXTURE_1D
The *params* parameter returns a single Boolean value indicating whether 1-D texture mapping is enabled. See **glTexImage1D**.

GL_TEXTURE_2D
The *params* parameter returns a single Boolean value indicating whether 2-D texture mapping is enabled. See **glTexImage2D**.

GL_TEXTURE_ENV_COLOR
The *params* parameter returns four values: the red, green, blue, and alpha values of the texture environment color. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glTexEnv**.

GL_TEXTURE_ENV_MODE
The *params* parameter returns one value: a symbolic constant indicating which texture

environment function is currently selected. See **glTexEnv**.

GL_TEXTURE_GEN_S
The *params* parameter returns a single Boolean value indicating whether automatic generation of the S texture coordinate is enabled. See **glTexGen**.

GL_TEXTURE_GEN_T
The *params* parameter returns a single Boolean value indicating whether automatic generation of the T texture coordinate is enabled. See **glTexGen**.

GL_TEXTURE_GEN_R
The *params* parameter returns a single Boolean value indicating whether automatic generation of the R texture coordinate is enabled. See **glTexGen**.

GL_TEXTURE_GEN_Q
The *params* parameter returns a single Boolean value indicating whether automatic generation of the Q texture coordinate is enabled. See **glTexGen**.

GL_TEXTURE_MATRIX
The *params* parameter returns 16 values: the texture matrix on the top of the texture matrix stack. See **glPushMatrix**.

GL_TEXTURE_STACK_DEPTH
The *params* parameter returns one value: the number of matrices on the texture matrix stack. See **glPushMatrix**.

GL_UNPACK_ALIGNMENT
The *params* parameter returns one value: the byte alignment used for reading pixel data from memory. See **glPixelStore**.

GL_UNPACK_LSB_FIRST
The *params* parameter returns a single Boolean value indicating whether single-bit pixels being read from memory are read first from the least significant bit of each unsigned byte. See **glPixelStore**.

GL_UNPACK_ROW_LENGTH
The *params* parameter returns one value: the row length used for reading pixel data from memory. See **glPixelStore**.

GL_UNPACK_SKIP_PIXELS
The *params* parameter returns one value: the number of pixel locations skipped before the first pixel is read from memory. See **glPixelStore**.

GL_UNPACK_SKIP_ROWS
The *params* parameter returns one value: the number of rows of pixel locations skipped before the first pixel is read from memory. See **glPixelStore**.

GL_UNPACK_SWAP_BYTES
The *params* parameter returns a single Boolean value indicating whether the bytes of 2-byte and 4-byte pixel indexes and components are swapped after being read from memory. See **glPixelStore**.

GL_VIEWPORT
The *params* parameter returns four values: the *x* and *y* window coordinates of the viewport, followed by its width and height. See **glViewport**.

GL_ZOOM_X
The *params* parameter returns one value: the *x* pixel zoom factor. See **glPixelZoom**.

GL_ZOOM_Y
The *params* parameter returns one value: the *y* pixel zoom factor. See **glPixelZoom**.

params
Returns the value or values of the specified parameter.

Remarks

These four functions return values for simple state variables in OpenGL. The *pname* parameter is a symbolic constant indicating the state variable to be returned, and *params* is a pointer to an array of the

indicated type in which to place the returned data.

Type conversion is performed if *params* has a different type from the state variable value being requested. If you call **glGetBooleanv**, a floating-point or integer value is converted to GL_FALSE if and only if it is zero. Otherwise, it is converted to GL_TRUE.

If you call **glGetIntegerv**, Boolean values are returned as GL_TRUE or GL_FALSE, and most floating-point values are rounded to the nearest integer value. Floating-point colors and normals, however, are returned with a linear mapping that maps 1.0 to the most positive representable integer value and -1.0 to the most negative representable integer value.

If you call **glGetFloatv** or **glGetDoublev**, Boolean values are returned as GL_TRUE or GL_FALSE, and integer values are converted to floating-point values.

You can query many of the Boolean parameters more easily with **glIsEnabled**.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*pname* was not an accepted value.
GL_INVALID_OPERATION	**glGet** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glAccum, **glAlphaFunc**, **glBegin**, **glBlendFunc**, **glCallList**, **glClearAccum**, **glClearColor**, **glClearDepth**, **glClearIndex**, **glClearStencil**, **glClipPlane**, **glColor**, **glColorMask**, **glColorMaterial**, **glCullFace**, **glDepthFunc**, **glDepthMask**, **glDepthRange**, **glDrawBuffer**, **glEdgeFlag**, **glEnd**, **glFog**, **glFrontFace**, **glGetClipPlane**, **glGetError**, **glGetLight**, **glGetMap**, **glGetMaterial**, **glGetPixelMap**, **glGetPolygonStipple**, **glGetString**, **glGetTexEnv**, **glGetTexGen**, **glGetTexImage**, **glGetTexLevelParameter**, **glGetTexParameter**, **glHint**, **glIndex**, **glIndexMask**, **glIsEnabled**, **glLight**, **glLightModel**, **glLineStipple**, **glLineWidth**, **glListBase**, **glLogicOp**, **glMap1**, **glMap2**, **glMapGrid**, **glMatrixMode**, **glNewList**, **glNormal**, **glPixelMap**, **glPixelStore**, **glPixelTransfer**, **glPixelZoom**, **glPointSize**, **glPolygonMode**, **glPolygonStipple**, **glPushAttrib**, **glPushMatrix**, **glPushName**, **glRasterPos**, **glReadPixels**, **glScissor**, **glShadeModel**, **glStencilFunc**, **glStencilMask**, **glStencilOp**, **glTexCoord**, **glTexEnv**, **glTexGen**, **glTexImage1D**, **glTexImage2D**, **glViewport**

glGetClipPlane Quick Info

[New - Windows 95, OEM Service Release 2]

The **glGetClipPlane** function returns the coefficients of the specified clipping plane.

```
void glGetClipPlane(
  GLenum plane,
  GLdouble * equation
);
```

Parameters

plane

A clipping plane. The number of clipping planes depends on the implementation, but at least six clipping planes are supported. They are identified by symbolic names of the form GL_CLIP_PLANE*i* where 0 ≤ *i* < GL_MAX_CLIP_PLANES.

equation

Returns four double-precision values that are the coefficients of the plane equation of *plane* in eye coordinates.

Remarks

The **glGetClipPlane** function returns in *equation* the four coefficients of the plane equation for *plane*.

It is always the case that GL_CLIP_PLANE*i* = GL_CLIP_PLANE0 + *i*.

If an error is generated, no change is made to the contents of *equation*.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*plane* was not an accepted value.
GL_INVALID_OPERATION	**glGetClipPlane** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glClipPlane**, **glEnd**

glGetColorTableEXT

[New - Windows 95, OEM Service Release 2]

The **glGetColorTableEXT** function gets the color table data of the current targeted texture palette.

```
void glGetColorTableEXT(
  GLenum target,
  GLenum format,
  GLenum type,
  const GLvoid * data
);
```

Parameters

target

The target texture that is to have its palette changed. Must be TEXTURE_1D or TEXTURE_2D.

format

The format of the pixel data. The following symbolic constants are accepted:

GL_RGBA

Each pixel is a group of four components in the following order: red, green, blue, alpha. The RGBA format is determined in this way:

1. The **glGetColorTableEXT** function converts floating-point values directly to an internal format with unspecified precision. Signed integer values are mapped linearly to the internal format such that the most positive representable integer value maps to 1.0, and the most negative representable integer value maps to - 1.0. Unsigned integer data is mapped similarly: the largest integer value maps to 1.0, and zero maps to 0.0.
2. The **glGetColorTableEXT** function multiplies the resulting color values by GL_c_SCALE and adds them to GL_c_BIAS, where *c* is RED, GREEN, BLUE, and ALPHA for the respective color components. The results are clamped to the range [0,1].
3. If GL_MAP_COLOR is TRUE, **glGetColorTableEXT** scales each color component by the size of lookup table GL_PIXEL_MAP_c_TO_c, then replaces the component by the value that it references in that table; *c* is R, G, B, or A, respectively.
4. The **glGetColorTableEXT** function converts the resulting RGBA colors to fragments by attaching the current raster position z-coordinate and texture coordinates to each pixel, then assigning *x* and *y* window coordinates to the *n*th fragment such that

$$x_{(n)} = x_{(r)} + n \bmod width$$

$$y_{(n)} = y_{(r)} + \lfloor n/width \rfloor$$

where $(x_{(r)}, y_{(r)})$ is the current raster position.

5. These pixel fragments are then treated just like the fragments generated by rasterizing points, lines, or polygons. The **glGetColorTableEXT** function applies texture mapping, fog, and all the fragment operations before writing the fragments to the frame buffer.

GL_RED

Each pixel is a single red component.

The **glGetColorTableEXT** function converts this component to the internal format in the same way that the red component of an RGBA pixel is, then converts it to an RGBA pixel with green and blue set to 0.0, and alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.

GL_GREEN

Each pixel is a single green component.

The **glGetColorTableEXT** function converts this component to the internal format in the same way that the green component of an RGBA pixel is, and then converts it to an RGBA pixel with red and blue set to 0.0, and alpha set to 1.0. After this conversion, the pixel is treated just as if it had been

read as an RGBA pixel.

GL_BLUE
Each pixel is a single blue component.

The **glGetColorTableEXT** function converts this component to the internal format in the same way that the blue component of an RGBA pixel is, and then converts it to an RGBA pixel with red and green set to 0.0, and alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.

GL_ALPHA
Each pixel is a single alpha component.

The **glGetColorTableEXT** function converts this component to the internal format in the same way that the alpha component of an RGBA pixel is, and then converts it to an RGBA pixel with red, green, and blue set to 0.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.

GL_RGB
Each pixel is a group of three components in this order: red, green, blue.

The **glGetColorTableEXT** function converts each component to the internal format in the same way that the red, green, and blue components of an RGBA pixel are. The color triple is converted to an RGBA pixel with alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.

GL_BGR_EXT
Each pixel is a group of three components in this order: blue, green, red.

GL_BGR_EXT provides a format that matches the memory layout of Windows device-independent bitmaps (DIBs). Thus your applications can use the same data with Win32 function calls and OpenGL pixel function calls.

GL_BGRA_EXT
Each pixel is a group of four components in this order: blue, green, red, alpha.

GL_BGRA_EXT provides a format that matches the memory layout of Windows device-independent bitmaps (DIBs). Thus your applications can use the same data with Win32 function calls and OpenGL pixel function calls.

type

The data type for *data*. The following symbolic constants are accepted: GL_UNSIGNED_BYTE, GL_BYTE, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, and GL_FLOAT.

The following table summarizes the meaning of the valid constants for the *type* parameter.

Constant	Meaning
GL_UNSIGNED_BYTE	Unsigned 8-bit integer
GL_BYTE	Signed 8-bit integer
GL_UNSIGNED_SHORT	Unsigned 16-bit integer
GL_SHORT	Signed 16-bit integer
GL_UNSIGNED_INT	Unsigned 32-bit integer
GL_INT	32-bit integer
GL_FLOAT	Single-precision floating-point value

data

Points to the location where returned color table information is to be stored. Each color table entry is stored as if it was a single pixel of a 1-D texture. Because all textures have a default palette, **glGetColorTableEXT** always returns palette information even if the texture data is not in a paletted format.

Remarks

The **glGetColorTableEXT** function gets the actual color table data specified by **glColorTableEXT** and

glColorSubTableEXT.

Note The **glGetColorTableEXT** function is an extension function that is not part of the standard OpenGL library but is part of the GL_EXT_paletted_texture extension. To check whether your implementation of OpenGL supports **glGetColorTableEXT**, call **glGetString**(GL_EXTENSIONS). If it returns GL_EXT_paletted_texture, **glGetColorTableEXT** is supported. To obtain the function address of an extension function, call **wglGetProcAddress**.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*target*, *format*, or *type* was not an accepted value.
GL_INVALID_OPERATION	**glGetColorTableEXT** was called between **glBegin** and **glEnd** pairs.

See Also

glColorSubTableEXT, **glColorTableEXT**, **glGetColorTableParameterfvEXT**, **glGetColorTableParameterivEXT**, **wglGetProcAddress**

glGetColorTableParameterfvEXT, glGetColorTableParameterivEXT

[New - Windows 95, OEM Service Release 2]

The **glGetColorTableParameterfvEXT** and **glGetColorTableParameterivEXT** functions get palette parameters from color tables.

```
void glGetColorTableParameterfvEXT(
  GLenum target,
  GLenum pname,
  GLint * params
);

void glGetColorTableParameterivEXT(
  GLenum target,
  GLenum pname,
  GLint * params
);
```

Parameters

target

The target texture of the palette for which you want parameter data. Must be TEXTURE_1D, TEXTURE_2D, PROXY_TEXTURE_1D, or PROXY_TEXTURE_2D.

pname

A symbolic constant for the type of palette parameter data pointed to by *params*.

The following table summarizes the meaning of the valid constants for the *pname* parameter.

Constant	Meaning
GL_COLOR_TABLE_FORMAT_EXT	Return the internal format specified by the most recent call to **glColorTableEXT** or the default value.
GL_COLOR_TABLE_WIDTH_EXT	Return the width of the current palette.
GL_COLOR_TABLE_RED_SIZE_EXT	Return the actual size used internally to store the red component of the palette data.
GL_COLOR_TABLE_GREEN_SIZE_EXT	Return the actual size used internally to store the green component of the palette data.
GL_COLOR_TABLE_BLUE_SIZE_EXT	Return the actual size used internally to store the blue component of the palette data.
GL_COLOR_TABLE_ALPHA_SIZE_EXT	Return the actual size used internally to store the alpha component of the palette data.

params

Points to the color table parameter data specified by the *pname* parameter.

Remarks

You use the **glGetColorTableParameterivEXT** and **glGetColorTableParameterfvEXT** functions to

retrieve specific parameter data from color tables set with **glColorTableEXT** for targeted texture palettes. Also you can use these functions to determine the number of color table entries that **glGetColorTableEXT** returns.

When the *target* parameter is GL_PROXY_TEXTURE_1D or GL_PROXY_TEXTURE_2D, and the implementation does not support the values specified for either *format* or *width*, **glColorTableEXT** can fail to create the requested color table. In this case, the color table is empty and all parameters retrieved will be zero. You can determine whether OpenGL supports a particular color table format and size by calling **glColorTableEXT** with a proxy target, and then calling **glGetColorTableParameterivEXT** or **glGetColorTableParameterfvEXT** to determine whether the width parameter matches that set by **glColorTableEXT**. If the retrieved width is zero, the color table request by **glColorTable** failed. If the retrieved width is not zero, you can call **glColorTable** with the real target with TEXTURE_1D or TEXTURE_2D to set the color table.

Note The **glGetColorTableParameterivEXT** and **glGetColorTableParameterfvEXT** functions are extension functions that are not part of the standard OpenGL library but are part of the GL_EXT_paletted_texture extension. To check whether your implementation of OpenGL supports **glGetColorTableParameterivEXT** and **glGetColorTableParameterfvEXT**, call **glGetString**(GL_EXTENSIONS). If it returns GL_EXT_paletted_texture, **glGetColorTableParameterivEXT** and **glGetColorTableParameterfvEXT** are supported. To obtain the function address of an extension function, call **wglGetProcAddress**.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*target* or *pname* was not an accepted value.
GL_INVALID_OPERATION	**glGetColorTableParameterivEXT** or **glGetColorTableParameterfvEXT** was called between **glBegin** and **glEnd** pairs.

See Also

glColorSubTableEXT, **glColorTableEXT**, **glGetColorTableEXT**, **wglGetProcAddress**

glGetError Quick Info

[New - Windows 95, OEM Service Release 2]

The **glGetError** function returns error information.

```
GLenum glGetError(
  void
);
```

Remarks

The **glGetError** function returns the value of the error flag. Each detectable error is assigned a numeric code and symbolic name. When an error occurs, the error flag is set to the appropriate error code value. No other errors are recorded until **glGetError** is called, the error code is returned, and the flag is reset to GL_NO_ERROR. If a call to **glGetError** returns GL_NO_ERROR, there has been no detectable error since the last call to **glGetError**, or since OpenGL was initialized.

To allow for distributed implementations, there may be several error flags. If any single error flag has recorded an error, the value of that flag is returned and that flag is reset to GL_NO_ERROR when **glGetError** is called. If more than one flag has recorded an error, **glGetError** returns and clears an arbitrary error flag value. If all error flags are to be reset, you should always call **glGetError** in a loop until it returns GL_NO_ERROR.

Initially, all error flags are set to GL_NO_ERROR.

The following are the currently defined errors:

GL_NO_ERROR
: No error has been recorded. The value of this symbolic constant is guaranteed to be zero.

GL_INVALID_ENUM
: An unacceptable value is specified for an enumerated argument. The offending function is ignored, having no side effect other than to set the error flag.

GL_INVALID_VALUE
: A numeric argument is out of range. The offending function is ignored, having no side effect other than to set the error flag.

GL_INVALID_OPERATION
: The specified operation is not allowed in the current state. The offending function is ignored, having no side effect other than to set the error flag.

GL_STACK_OVERFLOW
: This function would cause a stack overflow. The offending function is ignored, having no side effect other than to set the error flag.

GL_STACK_UNDERFLOW
: This function would cause a stack underflow. The offending function is ignored, having no side effect other than to set the error flag.

GL_OUT_OF_MEMORY
: There is not enough memory left to execute the function. The state of OpenGL is undefined, except for the state of the error flags, after this error is recorded.

When an error flag is set, results of an OpenGL operation are undefined only if GL_OUT_OF_MEMORY has occurred. In all other cases, the function generating the error is ignored and has no effect on the OpenGL state or frame buffer contents.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glGetError** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**

glGetLightfv, glGetLightiv

[New - Windows 95, OEM Service Release 2]

The **glGetLightfv** and **glGetLightiv** functions return light source parameter values.

```
void glGetLightfv(
  GLenum light,
  GLenum pname,
  GLfloat * params
);

void glGetLightiv(
  GLenum light,
  GLenum pname,
  GLint * params
);
```

Parameters

light

A light source. The number of possible lights depends on the implementation, but at least eight lights are supported. They are identified by symbolic names of the form GL_LIGHT*i* where $0 \leq i <$ GL_MAX_LIGHTS.

pname

A light source parameter for *light*. The following symbolic names are accepted:

GL_AMBIENT

The *params* parameter returns four integer or floating-point values representing the ambient intensity of the light source. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined.

GL_DIFFUSE

The *params* parameter returns four integer or floating-point values representing the diffuse intensity of the light source. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined.

GL_SPECULAR

The *params* parameter returns four integer or floating-point values representing the specular intensity of the light source. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined.

GL_POSITION

The *params* parameter returns four integer or floating-point values representing the position of the light source. Integer values, when requested, are computed by rounding the internal floating-point values to the nearest integer value. The returned values are those maintained in eye coordinates. They will not be equal to the values specified using **glLight**, unless the modelview matrix was identified at the time **glLight** was called.

GL_SPOT_DIRECTION

The *params* parameter returns three integer or floating-point values representing the direction of the light source. Integer values, when requested, are computed by rounding the internal floating-point values to the nearest integer value. The returned values are those maintained in eye coordinates. They will not be equal to the values specified using **glLight**, unless the modelview

matrix was identified at the time **glLight** was called. Although spot direction is normalized before being used in the lighting equation, the returned values are the transformed versions of the specified values prior to normalization.

GL_SPOT_EXPONENT
: The *params* parameter returns a single integer or floating-point value representing the spot exponent of the light. An integer value, when requested, is computed by rounding the internal floating-point representation to the nearest integer.

GL_SPOT_CUTOFF
: The *params* parameter returns a single integer or floating-point value representing the spot cutoff angle of the light. An integer value, when requested, is computed by rounding the internal floating-point representation to the nearest integer.

GL_CONSTANT_ATTENUATION
: The *params* parameter returns a single integer or floating-point value representing the constant (not distance-related) attenuation of the light. An integer value, when requested, is computed by rounding the internal floating-point representation to the nearest integer.

GL_LINEAR_ATTENUATION
: The *params* parameter returns a single integer or floating-point value representing the linear attenuation of the light. An integer value, when requested, is computed by rounding the internal floating-point representation to the nearest integer.

GL_QUADRATIC_ATTENUATION
: The *params* parameter returns a single integer or floating-point value representing the quadratic attenuation of the light. An integer value, when requested, is computed by rounding the internal floating-point representation to the nearest integer.

params
: Returns the requested data.

Remarks

The **glGetLight** function returns in *params* the value or values of a light source parameter. The *light* parameter names the light and is a symbolic name of the form GL_LIGHT*i* for $0 \leq i <$ GL_MAX_LIGHTS, where GL_MAX_LIGHTS is an implementation-dependent constant that is greater than or equal to eight. The *pname* parameter specifies one of ten light source parameters, again by symbolic name.

It is always the case that GL_LIGHT*i* = GL_LIGHT0 + *i*.

If an error is generated, no change is made to the contents of *params*.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*light* or *pname* was not an accepted value.
GL_INVALID_OPERATION	**glGetLight** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glLight**

glGetMapdv, glGetMapfv, glGetMapiv

[New - Windows 95, OEM Service Release 2]

These functions return evaluator parameters.

```
void glGetMapdv(
  GLenum target,
  GLenum query,
  GLdouble * v
);

void glGetMapfv(
  GLenum target,
  GLenum query,
  GLfloat * v
);

void glGetMapiv(
  GLenum target,
  GLenum query,
  GLint * v
);
```

Parameters

target

The symbolic name of a map. The following are accepted values: GL_MAP1_COLOR_4, GL_MAP1_INDEX, GL_MAP1_NORMAL, GL_MAP1_TEXTURE_COORD_1, GL_MAP1_TEXTURE_COORD_2, GL_MAP1_TEXTURE_COORD_3, GL_MAP1_TEXTURE_COORD_4, GL_MAP1_VERTEX_3, GL_MAP1_VERTEX_4, GL_MAP2_COLOR_4, GL_MAP2_INDEX, GL_MAP2_NORMAL, GL_MAP2_TEXTURE_COORD_1, GL_MAP2_TEXTURE_COORD_2, GL_MAP2_TEXTURE_COORD_3, GL_MAP2_TEXTURE_COORD_4, GL_MAP2_VERTEX_3, and GL_MAP2_VERTEX_4.

query

Specifies which parameter to return. The following symbolic names are accepted:

GL_COEFF

The *v* parameter returns the control points for the evaluator function. One-dimensional evaluators return *order* control points, and two-dimensional evaluators return *uorder*x*vorder* control points. Each control point consists of one, two, three, or four integer, single-precision floating-point, or double-precision floating-point values, depending on the type of the evaluator. Two-dimensional control points are returned in row-major order, incrementing the *uorder* index quickly, and the *vorder* index after each row. Integer values, when requested, are computed by rounding the internal floating-point values to the nearest integer values.

GL_ORDER

The *v* parameter returns the order of the evaluator function. One-dimensional evaluators return a single value, *order*. Two-dimensional evaluators return two values, *uorder* and *vorder*.

GL_DOMAIN

The *v* parameter returns the linear *u* and *v* mapping parameters. One-dimensional evaluators return two values, *u*1 and *u*2, as specified by **glMap1**. Two-dimensional evaluators return four values (*u*1, *u*2, *v*1, and *v*2) as specified by **glMap2**. Integer values, when requested, are computed by rounding the internal floating-point values to the nearest integer values.

v

Returns the requested data.

Remarks

The **glGetMap** function returns evaluator parameters. (The **glMap1** and **glMap2** functions define evaluators.) The *target* parameter specifies a map, *query* selects a specific parameter, and *v* points to storage where the values will be returned.

The acceptable values for the *target* parameter are described in **glMap1** and **glMap2**

If an error is generated, no change is made to the contents of *v*.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*target* or *query* was not an accepted value.
GL_INVALID_OPERATION	**glGetMap** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glEvalCoord**, **glMap1**, **glMap2**

glGetMaterialfv, glGetMaterialiv

[New - Windows 95, OEM Service Release 2]

The **glGetMaterialfv** and **glGetMaterialiv** functions return material parameters.

```
void glGetMaterialfv(
  GLenum face,
  GLenum pname,
  GLfloat * params
);

void glGetMaterialiv(
  GLenum face,
  GLenum pname,
  GLint * params
);
```

Parameters

face

Specifies which of the two materials is being queried. GL_FRONT or GL_BACK are accepted, representing the front and back materials, respectively.

pname

The material parameter to return. The following values are accepted:

GL_AMBIENT

The *params* parameter returns four integer or floating-point values representing the ambient reflectance of the material. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined.

GL_DIFFUSE

The *params* parameter returns four integer or floating-point values representing the diffuse reflectance of the material. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined.

GL_SPECULAR

The *params* parameter returns four integer or floating-point values representing the specular reflectance of the material. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined.

GL_EMISSION

The *params* parameter returns four integer or floating-point values representing the emitted light intensity of the material. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined.

GL_SHININESS

The *params* parameter returns one integer or floating-point value representing the specular exponent of the material. Integer values, when requested, are computed by rounding the internal floating-point value to the nearest integer value.

GL_COLOR_INDEXES

The *params* parameter returns three integer or floating-point values representing the ambient,

diffuse, and specular indexes of the material. Use these indexes only for color-index lighting. (The other parameters are all used only for RGBA lighting.) Integer values, when requested, are computed by rounding the internal floating-point values to the nearest integer values.

params
Returns the requested data.

Remarks

The **glGetMaterial** function returns in *params* the value or values of parameter *pname* of material *face*.

If an error is generated, no change is made to the contents of *params*.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*face* or *pname* was not an accepted value.
GL_INVALID_OPERATION	**glGetMaterial** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glMaterial**

glGetPixelMapfv, glGetPixelMapuiv, glGetPixelMapusv

[New - Windows 95, OEM Service Release 2]

These functions return the specified pixel map.

```
void glGetPixelMapfv(
  GLenum map,
  GLfloat * values
);

void glGetPixelMapuiv(
  GLenum map,
  GLuint * values
);

void glGetPixelMapusv(
  GLenum map,
  GLushort * values
);
```

Parameters

map
The name of the pixel map to return. Accepted values are GL_PIXEL_MAP_I_TO_I, GL_PIXEL_MAP_S_TO_S, GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, GL_PIXEL_MAP_I_TO_A, GL_PIXEL_MAP_R_TO_R, GL_PIXEL_MAP_G_TO_G, GL_PIXEL_MAP_B_TO_B, and GL_PIXEL_MAP_A_TO_A.

values
Returns the pixel map contents.

Remarks

See **glPixelMap** for a description of the acceptable values for the *map* parameter. The **glGetPixelMap** function returns in *values* the contents of the pixel map specified in *map*. Use pixel maps during the execution of **glReadPixels**, **glDrawPixels**, **glCopyPixels**, **glTexImage1D**, and **glTexImage2D** to map color indexes, stencil indexes, color components, and depth components to other values.

Unsigned integer values, if requested, are linearly mapped from the internal fixed or floating-point representation such that 1.0 maps to the largest representable integer value, and 0.0 maps to zero. Return unsigned integer values are undefined if the map value was not in the range [0,1].

To determine the required size of *map*, call **glGet** with the appropriate symbolic constant.

If an error is generated, no change is made to the contents of *values*.

The following functions retrieve information related to **glGetPixelMap**:

glGet with argument GL_PIXEL_MAP_I_TO_I_SIZE
glGet with argument GL_PIXEL_MAP_S_TO_S_SIZE
glGet with argument GL_PIXEL_MAP_I_TO_R_SIZE
glGet with argument GL_PIXEL_MAP_I_TO_G_SIZE
glGet with argument GL_PIXEL_MAP_I_TO_B_SIZE
glGet with argument GL_PIXEL_MAP_I_TO_A_SIZE
glGet with argument GL_PIXEL_MAP_R_TO_R_SIZE

glGet with argument GL_PIXEL_MAP_G_TO_G_SIZE
glGet with argument GL_PIXEL_MAP_B_TO_B_SIZE
glGet with argument GL_PIXEL_MAP_A_TO_A_SIZE
glGet with argument GL_MAX_PIXEL_MAP_TABLE

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*map* was not an accepted value.
GL_INVALID_OPERATION	**glGetPixelMap** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glCopyPixels**, **glDrawPixels**, **glEnd**, **glGet**, **glPixelMap**, **glPixelTransfer**, **glReadPixels**, **glTexImage1D**, **glTexImage2D**

glGetPointerv

[New - Windows 95, OEM Service Release 2]

The **glGetPointerv** function returns the address of a vertex data array.

```
void glGetPointerv(
  GLenum pname,
  GLvoid * * params
);
```

Parameters

pname

The type of array pointer to return from the following symbolic constants: GL_VERTEX_ARRAY_POINTER, GL_NORMAL_ARRAY_POINTER, GL_COLOR_ARRAY_POINTER, GL_INDEX_ARRAY_POINTER, GL_TEXTURE_COORD_ARRAY_POINTER, and GL_EDGE_FLAG_ARRAY_POINTER.

params

Returns the value of the array pointer specified by *pname*.

Remarks

The **glGetPointerv** function returns array pointer information. The *pname* parameter is a symbolic constant specifying the kind of array pointer to return, and *params* is a pointer to a location to place the returned data.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*pname* was not an accepted value.

See Also

glArrayElement, **glColorPointer**, **glDrawArrays**, **glEdgeFlagPointer**, **glGetString**, **glIndexPointer**, **glNormalPointer**, **glTexCoordPointer**, **glVertexPointer**

glGetPolygonStipple Quick Info

[New - Windows 95, OEM Service Release 2]

The **glGetPolygonStipple** function returns the polygon stipple pattern.

```
void glGetPolygonStipple(
  GLubyte * mask
);
```

Parameters

mask
Returns the stipple pattern.

Remarks

The **glGetPolygonStipple** function returns to *mask* a 32x32 polygon stipple pattern. The pattern is packed into memory as if **glReadPixels** with both *height* and *width* of 32, *type* of GL_BITMAP, and *format* of GL_COLOR_INDEX were called, and the stipple pattern were stored in an internal 32x32 color-index buffer. Unlike **glReadPixels**, however, pixel-transfer operations (shift, offset, and pixel map) are not applied to the returned stipple image.

If an error is generated, no change is made to the contents of *mask*.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glGetPolygonStipple** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glPixelStore**, **glPixelTransfer**, **glPolygonStipple**, **glReadPixels**

glGetString Quick Info

[New - Windows 95, OEM Service Release 2]

The **glGetString** function returns a string describing the current OpenGL connection.

```
const GLubyte * glGetString(
  GLenum name
);
```

Parameters

name

One of the following symbolic constants:

GL_VENDOR
Returns the company responsible for this OpenGL implementation. This name does not change from release to release.

GL_RENDERER
Returns the name of the renderer. This name is typically specific to a particular configuration of a hardware platform. It does not change from release to release.

GL_VERSION
Returns a version or release number.

GL_EXTENSIONS
Returns a space-separated list of supported extensions to OpenGL.

Remarks

The **glGetString** function returns a pointer to a static string describing some aspect of the current OpenGL connection.

Because OpenGL does not include queries for the performance characteristics of an implementation, it is expected that some applications will be written to recognize known platforms and will modify their OpenGL usage based on known performance characteristics of these platforms. The strings GL_VENDOR and GL_RENDERER together uniquely specify a platform, and will not change from release to release. They should be used by such platform recognition algorithms.

The format and contents of the string that **glGetString** returns depend on the implementation, except that:

- Extension names will not include space characters and will be separated by space characters in the GL_EXTENSIONS string.
- All strings are null-terminated.

If an error is generated, **glGetString** returns zero.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*name* was not an accepted value.
GL_INVALID_OPERATION	**glGetString** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**

glGetTexEnvfv Quick Info

[New - Windows 95, OEM Service Release 2]

The **glGetTexEnvfv** and **glGetTexEnviv** functions return texture environment parameters.

```
void glGetTexEnvfv(
  GLenum target,
  GLenum pname,
  GLfloat * params
);

void glGetTexEnviv(
  GLenum target,
  GLenum pname,
  GLint * params
);
```

Parameters

target
A texture environment. Must be GL_TEXTURE_ENV.

pname
The symbolic name of a texture environment parameter. The following values are accepted:

GL_TEXTURE_ENV_MODE
The *params* parameter returns the single-valued texture environment mode, a symbolic constant.

GL_TEXTURE_ENV_COLOR
The *params* parameter returns four integer or floating-point values that are the texture environment color. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer, and -1.0 maps to the most negative representable integer.

params
Returns the requested data.

Remarks

The **glGetTexEnv** function returns in *params* selected values of a texture environment that was specified with **glTexEnv**. The *target* parameter specifies a texture environment. Currently, only one texture environment is defined and supported: GL_TEXTURE_ENV.

The *pname* parameter names a specific texture environment parameter.

If an error is generated, no change is made to the contents of *params*.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*target* or *pname* was not an accepted value.
GL_INVALID_OPERATION	**glGetTexEnv** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glTexEnv**

glGetTexGendv, glGetTexGenfv, glGetTexGeniv

[New - Windows 95, OEM Service Release 2]

These functions return texture coordinate generation parameters.

```
void glGetTexGendv(
  GLenum coord,
  GLenum pname,
  GLdouble * params
);

void glGetTexGenfv(
  GLenum coord,
  GLenum pname,
  GLfloat * params
);

void glGetTexGeniv(
  GLenum coord,
  GLenum pname,
  GLint * params
);
```

Parameters

coord

A texture coordinate. Must be GL_S, GL_T, GL_R, or GL_Q.

pname

The symbolic name of the value(s) to be returned. Must be either GL_TEXTURE_GEN_MODE or the name of one of the texture generation plane equations: GL_OBJECT_PLANE or GL_EYE_PLANE. These values are as follows:

GL_TEXTURE_GEN_MODE

The *params* parameter returns the single-valued texture-generation function, a symbolic constant.

GL_OBJECT_PLANE

The *params* parameter returns the four plane equation coefficients that specify object linear-coordinate generation. Integer values, when requested, are mapped directly from the internal floating-point representation.

GL_EYE_PLANE

The *params* parameter returns the four plane equation coefficients that specify eye linear-coordinate generation. Integer values, when requested, are mapped directly from the internal floating-point representation. The returned values are those maintained in eye coordinates. They are not equal to the values specified using **glTexGen**, unless the modelview matrix was identified at the time **glTexGen** was called.

params

Returns the requested data.

Remarks

The **glGetTexGen** function returns in *params* selected parameters of a texture-coordinate generation function that you specified with **glTexGen**. The *coord* parameter names one of the (*s*,*t*,*r*,*q*) texture coordinates, using the symbolic constant GL_S, GL_T, GL_R, or GL_Q.

If an error is generated, no change is made to the contents of *params*.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*coord* or *pname* was not an accepted value.
GL_INVALID_OPERATION	**glGetTexGen** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glTexGen**

glGetTexImage Quick Info

[New - Windows 95, OEM Service Release 2]

The **glGetTexImage** function returns a texture image.

```
void glGetTexImage(
  GLenum target,
  GLint level,
  GLenum format,
  GLenum type,
  GLvoid * pixels
);
```

Parameters

target
Specifies which texture is to be obtained. GL_TEXTURE_1D and GL_TEXTURE_2D are accepted.

level
The level-of-detail number of the desired image. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.

format
A pixel format for the returned data. The supported formats are GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_LUMINANCE, GL_BGR_EXT, GL_BGRA_EXT, and GL_LUMINANCE_ALPHA.

type
A pixel type for the returned data. The supported types are GL_UNSIGNED_BYTE, GL_BYTE, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, and GL_FLOAT.

pixels
Returns the texture image. Should be a pointer to an array of the type specified by *type*.

Remarks

The **glGetTexImage** function returns a texture image into *pixels*. The *target* parameter specifies whether the desired texture image is one specified by **glTexImage1D(**GL_TEXTURE_1D**)** or by **glTexImage2D(**GL_TEXTURE_2D**)**. The *level* parameter specifies the level-of-detail number of the desired image. The *format* and *type* parameters specify the format and type of the desired image array. For a description of the acceptable values for the *format* and *type* parameters, respectively, see **glTexImage1D** and **glDrawPixels**.

Operation of **glGetTexImage** is best understood by considering the selected internal four-component texture image to be an RGBA color buffer the size of the image. The semantics of **glGetTexImage** are then identical to those of **glReadPixels** called with the same *format* and *type*, with *x* and *y* set to zero, *width* set to the width of the texture image (including border if one was specified), and *height* set to one for 1-D images, or to the height of the texture image (including border, if one was specified) for 2-D images.

Because the internal texture image is an RGBA image, pixel formats GL_COLOR_INDEX, GL_STENCIL_INDEX, and GL_DEPTH_COMPONENT are not accepted, and pixel type GL_BITMAP is not accepted.

If the selected texture image does not contain four components, the following mappings are applied. Single-component textures are treated as RGBA buffers with red set to the single-component value, and green, blue, and alpha set to zero.

Two-component textures are treated as RGBA buffers, with red set to the value of component zero, alpha

set to the value of component one, and green and blue set to zero. Finally, three-component textures are treated as RGBA buffers with red set to component zero, green set to component one, blue set to component two, and alpha set to zero.

To determine the required size of *pixels*, use **glGetTexLevelParameter** to ascertain the dimensions of the internal texture image, and then scale the required number of pixels by the storage required for each pixel, based on *format* and *type*. Be sure to take the pixel-storage parameters into account, especially GL_PACK_ALIGNMENT.

If an error is generated, no change is made to the contents of *pixels*.

The following functions retrieve information related to **glGetTexImage**:

glGetTexLevelParameter with argument GL_TEXTURE_WIDTH
glGetTexLevelParameter with argument GL_TEXTURE_HEIGHT
glGetTexLevelParameter with argument GL_TEXTURE_BORDER
glGetTexLevelParameter with argument GL_TEXTURE_COMPONENTS
glGet with argument GL_PACK_ALIGNMENT and others

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*target*, *format*, or *type* was not an accepted value.
GL_INVALID_VALUE	*level* is less than zero or greater than $\log_{(2)} max$, where *max* is the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_OPERATION	**glGetTexImage** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glDrawPixels**, **glEnd**, **glGetTexLevelParameter**, **glReadPixels**, **glTexImage1D**, **glTexImage2D**

glGetTexLevelParameterfv, glGetTexLevelParameteriv

[New - Windows 95, OEM Service Release 2]

The **glGetTexLevelParameterfv** and **glGetTexLevelParameteriv** functions return texture parameter values for a specific level of detail.

```
void glGetTexLevelParameterfv(
  GLenum target,
  GLint level,
  GLenum pname,
  GLfloat * params
);

void glGetTexLevelParameteriv(
  GLenum target,
  GLint level,
  GLenum pname,
  GLint * params
);
```

Parameters

target
The symbolic name of the target texture: either GL_TEXTURE_1D or GL_TEXTURE_2D.

level
The level-of-detail number of the desired image. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.

pname
The symbolic name of a texture parameter. The following parameter names are accepted:

GL_TEXTURE_WIDTH
The *params* parameter returns a single value: the width of the texture image. This value includes the border of the texture image.

GL_TEXTURE_HEIGHT
The *params* parameter returns a single value: the height of the texture image. This value includes the border of the texture image.

GL_TEXTURE_COMPONENTS
The *params* parameter returns a single value: the number of components in the texture image.

GL_TEXTURE_BORDER
The *params* parameter returns a single value: the width in pixels of the border of the texture image.

params
Returns the requested data.

Remarks

The **glGetTexLevelParameter** function returns in *params* texture parameter values for a specific level-of-detail value, specified as *level*. The *target* parameter defines the target texture—either GL_TEXTURE_1D or GL_TEXTURE_2D—to specify one- or two-dimensional texturing. The *pname* parameter specifies the texture parameter whose value or values will be returned.

If an error is generated, no change is made to the contents of *params*.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*target* or *pname* was not an accepted value.
GL_INVALID_VALUE	*level* is less than zero or greater than $\log_{(2)}$ *max*, where *max* is the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_OPERATION	**glGetTexLevelParameter** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glGetTexParameter**, **glTexImage1D**, **glTexImage2D**, **glTexParameter**

glGetTexParameterfv, glGetTexParameteriv

[New - Windows 95, OEM Service Release 2]

The **glGetTexParameterfv** and **glGetTexParameteriv** functions return texture parameter values.

```
void glGetTexParameterfv(
  GLenum target,
  GLenum pname,
  GLfloat * params
);

void glGetTexParameteriv(
  GLenum target,
  GLenum pname,
  GLint * params
);
```

Parameters

target
The symbolic name of the target texture. GL_TEXTURE_1D and GL_TEXTURE_2D are accepted.

pname
The symbolic name of a texture parameter. The following values are accepted:

GL_TEXTURE_MAG_FILTER
Returns the single-valued texture magnification filter, a symbolic constant.

GL_TEXTURE_MIN_FILTER
Returns the single-valued texture minification filter, a symbolic constant.

GL_TEXTURE_WRAP_S
Returns the single-valued wrapping function for texture coordinate *s*, a symbolic constant.

GL_TEXTURE_WRAP_T
Returns the single-valued wrapping function for texture coordinate *t*, a symbolic constant.

GL_TEXTURE_BORDER_COLOR
Returns four integer or floating-point numbers that comprise the RGBA color of the texture border. Floating-point values are returned in the range [0,1]. Integer values are returned as a linear mapping of the internal floating-point representation such that 1.0 maps to the most positive representable integer and -1.0 maps to the most negative representable integer.

params
Returns the texture parameters.

Remarks

The **glGetTexParameter** function returns in *params* the value or values of the texture parameter specified as *pname*. The *target* parameter defines the target texture—either GL_TEXTURE_1D or GL_TEXTURE_2D—to specify one- or two-dimensional texturing. The *pname* parameter accepts the same symbols as **glTexParameter**, with the same interpretations.

If an error is generated, no change is made to the contents of *params*.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition

GL_INVALID_ENUM	*target* or *pname* was not an accepted value.
GL_INVALID_OPERATION	**glGetTexParameter** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glTexParameter**

glHint Quick Info

[New - Windows 95, OEM Service Release 2]

The **glHint** function specifies implementation-specific hints.

```
void glHint(
  GLenum target,
  GLenum mode
);
```

Parameters

target

A symbolic constant indicating the behavior to be controlled. The following symbolic constants, along with suggested semantics, are accepted:

GL_FOG_HINT
Indicates the accuracy of fog calculation. If per-pixel fog calculation is not efficiently supported by the OpenGL implementation, hinting GL_DONT_CARE or GL_FASTEST can result in per-vertex calculation of fog effects.

GL_LINE_SMOOTH_HINT
Indicates the sampling quality of antialiased lines. Hinting GL_NICEST can result in more pixel fragments being generated during rasterization, if a larger filter function is applied.

GL_PERSPECTIVE_CORRECTION_HINT
Indicates the quality of color and texture coordinate interpolation. If perspective-corrected parameter interpolation is not efficiently supported by the OpenGL implementation, hinting GL_DONT_CARE or GL_FASTEST can result in simple linear interpolation of colors and/or texture coordinates.

GL_POINT_SMOOTH_HINT
Indicates the sampling quality of antialiased points. Hinting GL_NICEST can result in more pixel fragments being generated during rasterization, if a larger filter function is applied.

GL_POLYGON_SMOOTH_HINT
Indicates the sampling quality of antialiased polygons. Hinting GL_NICEST can result in more pixel fragments being generated during rasterization, if a larger filter function is applied.

mode

A symbolic constant indicating the desired behavior. The following symbolic constants are accepted:

GL_FASTEST
The most efficient option should be chosen.

GL_NICEST
The most correct, or highest quality, option should be chosen.

GL_DONT_CARE
The client doesn't have a preference.

Remarks

When there is room for interpretation, you can control certain aspects of OpenGL behavior with hints. You specify a hint with two arguments. The *target* parameter is a symbolic constant indicating the behavior to be controlled, and *mode* is another symbolic constant indicating the desired behavior.

Though the implementation aspects that can be hinted are well defined, the interpretation of the hints depends on the implementation.

The **glHint** function can be ignored.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*target* or *mode* was not an accepted value.
GL_INVALID_OPERATION	**glHint** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**

glIndex

[New - Windows 95, OEM Service Release 2]

glIndexd, glIndexf, glIndexi, glIndexs, glIndexdv, glIndexfv, glIndexiv, glIndexsv

These functions set the current color index.

```
void glIndexd(
  GLdouble c
);

void glIndexf(
  GLfloat c
);

void glIndexi(
  GLint c
);

void glIndexs(
  GLshort c
);
```

Parameters

c
The new value for the current color index.

```
void glIndexdv(
  const GLdouble *c
);

void glIndexfv(
  const GLfloat *c
);

void glIndexiv(
  const GLint *c
);

void glIndexsv(
  const GLshort *c
);
```

Parameters

c
A pointer to a one-element array that contains the new value for the current color index.

Remarks

The **glIndex** function updates the current (single-valued) color index. It takes one argument: the new value for the current color index.

The current index is stored as a floating-point value. Integer values are converted directly to floating-point values, with no special mapping.

Index values outside the representable range of the color-index buffer are not clamped. However, before an index is dithered (if enabled) and written to the frame buffer, it is converted to fixed-point format. Any

bits in the integer portion of the resulting fixed-point value that do not correspond to bits in the frame buffer are masked out.

The current index can be updated at any time. In particular, **glIndex** can be called between a call to **glBegin** and the corresponding call to **glEnd**.

The following function retrieves information related to **glIndex**:

glGet with argument GL_CURRENT_INDEX

See Also

glBegin, **glColor**, **glEnd**, **glGet**

glIndexMask Quick Info

[New - Windows 95, OEM Service Release 2]

The **glIndexMask** function controls the writing of individual bits in the color-index buffers.

```
void glIndexMask(
  GLuint mask
);
```

Parameters

mask

A bit mask to enable and disable the writing of individual bits in the color-index buffers. Initially, the mask is all ones.

Remarks

The **glIndexMask** function controls the writing of individual bits in the color-index buffers. The least significant *n* bits of *mask*, where *n* is the number of bits in a color-index buffer, specify a mask. Wherever a one appears in the mask, the corresponding bit in the color-index buffer (or buffers) is made writable. Where a zero appears, the bit is write-protected.

This mask is used only in color-index mode, and it affects only the buffers currently selected for writing (see **glDrawBuffer**). Initially, all bits are enabled for writing.

The following function retrieves information related to **glIndexMask**:

glGet with argument GL_INDEX_WRITEMASK

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glIndexMask** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glDepthMask**, **glDrawBuffer**, **glEnd**, **glIndex**, **glStencilMask**

glIndexPointer

[New - Windows 95, OEM Service Release 2]

The **glIndexPointer** function defines an array of color indexes.

```
void glIndexPointer(
  GLenum type,
  GLsizei stride,
  GLsizei count,
  const GLvoid *pointer
);
```

Parameters

type
The data type of each color index in the array using the following symbolic constants: GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE.

stride
The byte offset between consecutive color indexes. When *stride* is zero, the color indexes are tightly packed in the array.

count
The number of color indexes, counting from the first, that are static.

pointer
A pointer to the first color index in the array.

Remarks

The **glIndexPointer** function specifies the location and data of an array of color indexes to use when rendering. The *type* parameter specifies the data type of each color index and *stride* determines the byte offset from one color index to the next, enabling the packing of vertices and attributes in a single array or storage in separate arrays. In some implementations storing the vertices and attributes in a single array can be more efficient than using separate arrays. Starting from the first color-index element, *count* indicates the total number of static elements. Your application can modify static elements, but once the elements are modified, the application must explicitly specify the array again before using the array for any rendering. Non-static array elements are not accessed until you call **glDrawArrays** or **glArrayElement**.

A color-index array is enabled when you specify the GL_INDEX_ARRAY constant with **glEnableClientState**. When enabled, **glDrawArrays** and **glArrayElement** use the color-index array. By default the color-index array is disabled.

You cannot include **glIndexPointer** in display lists.

When you specify a color-index array using **glIndexPointer**, the values of all the function's color-index array parameters are saved in a client-side state and static array elements can be cached. Because the color-index array parameters are client-side state, their values are not saved or restored by **glPushAttrib** and **glPopAttrib**.

Although no error is generated when you call **glIndexPointer** within **glBegin** and **glEnd** pairs, the results are undefined.

The following functions retrieve information related to **glIndexPointer**:

glIsEnabled with argument GL_INDEX_ARRAY
glGet with argument GL_INDEX_ARRAY_STRIDE
glGet with argument GL_INDEX_ARRAY_COUNT

glGet with argument GL_INDEX_ARRAY_TYPE
glGet with argument GL_INDEX_ARRAY_SIZE
glGetPointerv with argument GL_INDEX_ARRAY_POINTER

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*type* was not an accepted value.
GL_INVALID_VALUE	*stride* or *count* was negative.

See Also

glArrayElement, **glColorPointer**, **glDrawArrays**, **glEdgeFlagPointer**, **glGetPointerv**, **glGetString**, **glNormalPointer**, **glPushAttrib**, **glTexCoordPointer**, **glVertexPointer**

glInitNames Quick Info

[New - Windows 95, OEM Service Release 2]

The **glInitNames** function initializes the name stack.

```
void glInitNames(
  void
);
```

Remarks

The **glInitNames** function causes the name stack to be initialized to its default empty state. The name stack is used during selection mode to allow sets of rendering commands to be uniquely identified. It consists of an ordered set of unsigned integers.

The name stack is always empty while the render mode is not GL_SELECT. Calls to **glInitNames** while the render mode is not GL_SELECT are ignored.

The following functions retrieve information related to **glInitNames**:

glGet with argument GL_NAME_STACK_DEPTH
glGet with argument GL_MAX_NAME_STACK_DEPTH

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glInitNames** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glLoadName**, **glPushName**, **glRenderMode**, **glSelectBuffer**

glInterleavedArrays

[New - Windows 95, OEM Service Release 2]

The **glInterleavedArrays** function simultaneously specifies and enables several interleaved arrays in a larger aggregate array.

```
void glInterleavedArrays(
  GLenum format,
  GLsizei stride,
  const GLvoid * pointer
);
```

Parameters

format

The type of array to enable. The parameter can assume one of the following symbolic values: GL_V2F, GL_V3F, GL_C4UB_V2F, GL_C4UB_V3F, GL_C3F_V3F, GL_N3F_V3F, GL_C4F_N3F_V3F, GL_T2F_V3F, GL_T4F_V4F, GL_T2F_C4UB_V3F, GL_T2F_C3F_V3F, GL_T2F_N3F_V3F, GL_T2F_C4F_N3F_V3F, or GL_T4F_C4F_N3F_V4F.

stride

The offset in bytes between each aggregate array element.

pointer

A pointer to the first element of an aggregate array.

Remarks

With the **glInterleavedArrays** function you can simultaneously specify and enable several interleaved color, normal, texture, and vertex arrays whose elements are part of a larger aggregate array element. For some memory architectures this is more efficient than specifying the arrays separately.

If the *stride* parameter is zero then the aggregate array elements are stored consecutively; otherwise *stride* bytes occur between aggregate array elements.

The *format* parameter serves as a "key" that describes how to extract individual arrays from the aggregate array:

- If *format* contains a T, then texture coordinates are extracted from the interleaved array.
- If C is present, color values are extracted.
- If N is present, normal coordinates are extracted.
- Vertex coordinates are always extracted.
- The digits 2, 3, and 4 denote how many values are extracted.
- F indicates that values are extracted as floating point values.
- If 4UB follows the C, colors may also be extracted as 4 unsigned bytes. If a color is extracted as 4 unsigned bytes the vertex array element that follows is located at the first possible floating point aligned address.

If you call **glInterleavedArrays** while compiling a display list, it is not compiled into the list but is executed immediately.

You cannot include calls to **glInterleavedArrays** in **glDisableClientState** between calls to **glBegin** and the corresponding call to **glEnd**.

Note The **glInterleavedArrays** function is only available in OpenGL version 1.1 or later.

The **glInterleavedArrays** function is implemented on the client side with no protocol. Because the vertex array parameters are client-side state, they are not saved or restored by **glPushAttrib** and **glPopAttrib**. Use **glPushClientAttrib** and **glPopClientAttrib** instead.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*format* was not an accepted value.
GL_INVALID_VALUE	*stride* was a negative value.
GL_INVALID_OPERATION	**glInterleavedArrays** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glArrayElement, **glColorPointer**, **glDrawArrays**, **glDrawElements**, **glEdgeFlagPointer**, **glEnableClientState**, **glGetPointerv**, **glIndexPointer**, **glNormalPointer**, **glPushAttrib**, **glPushClientAttrib**, **glTexCoordPointer**, **glVertexPointer**

glIsEnabled Quick Info

[New - Windows 95, OEM Service Release 2]

The **glIsEnabled** function tests whether a capability is enabled.

```
GLboolean glIsEnabled(
  GLenum cap
);
```

Parameters

cap

A symbolic constant indicating an OpenGL capability. The following capabilities are accepted:

GL_ALPHA_TEST	See **glAlphaFunc**
GL_AUTO_NORMAL	See **glEvalCoord**
GL_BLEND	See **glBlendFunc**
GL_CLIP_PLANE*i*	See **glClipPlane**
GL_COLOR_MATERIAL	See **glColorMaterial**
GL_CULL_FACE	See **glCullFace**
GL_DEPTH_TEST	See **glDepthFunc** and **glDepthRange**
GL_DITHER	See **glEnable**
GL_FOG	See **glFog**
GL_LIGHT*i*	See **glLightModel** and **glLight**
GL_LIGHTING	See **glMaterial**, **glLightModel**, and **glLight**
GL_LINE_SMOOTH	See **glLineWidth**
GL_LINE_STIPPLE	See **glLineStipple**
GL_LOGIC_OP	See **glLogicOp**
GL_MAP1_COLOR_4	See **glMap1**
GL_MAP1_INDEX	See **glMap1**
GL_MAP1_NORMAL	See **glMap1**
GL_MAP1_TEXTURE_COORD_1	See **glMap1**
GL_MAP1_TEXTURE_COORD_2	See **glMap1**
GL_MAP1_TEXTURE_COORD_3	See **glMap1**
GL_MAP1_TEXTURE_COORD_4	See **glMap1**
GL_MAP1_VERTEX_3	See **glMap1**
GL_MAP1_VERTEX_4	See **glMap1**
GL_MAP2_COLOR_4	See **glMap2**
GL_MAP2_INDEX	See **glMap2**
GL_MAP2_NORMAL	See **glMap2**
GL_MAP2_TEXTURE_COORD_1	See **glMap2**
GL_MAP2_TEXTURE_COORD_2	See **glMap2**
GL_MAP2_TEXTURE_COORD_3	See **glMap2**
GL_MAP2_TEXTURE_COORD_4	See **glMap2**
GL_MAP2_VERTEX_3	See **glMap2**
GL_MAP2_VERTEX_4	See **glMap2**

GL_NORMALIZE	See **glNormal**
GL_POINT_SMOOTH	See **glPointSize**
GL_POLYGON_SMOOTH	See **glPolygonMode**
GL_POLYGON_STIPPLE	See **glPolygonStipple**
GL_SCISSOR_TEST	See **glScissor**
GL_STENCIL_TEST	See **glStencilFunc** and **glStencilOp**
GL_TEXTURE_1D	See **glTexImage1D**
GL_TEXTURE_2D	See **glTexImage2D**
GL_TEXTURE_GEN_Q	See **glTexGen**
GL_TEXTURE_GEN_R	See **glTexGen**
GL_TEXTURE_GEN_S	See **glTexGen**
GL_TEXTURE_GEN_T	See **glTexGen**

Remarks

The **glIsEnabled** function returns GL_TRUE if *cap* is an enabled capability and returns GL_FALSE otherwise.

Error Codes

If an error is generated, **glIsEnabled** returns zero.

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*cap* was not an accepted value.
GL_INVALID_OPERATION	**glIsEnabled** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnable**, **glEnd**

glIsList Quick Info

[New - Windows 95, OEM Service Release 2]

The **glIsList** function tests for display list existence.

```
GLboolean glIsList(
  GLuint list
);
```

Parameters

list

A potential display list name.

Remarks

The **glIsList** function returns GL_TRUE if *list* is the name of a display list and returns GL_FALSE otherwise.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glIsList** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glCallList**, **glCallLists**, **glDeleteLists**, **glEnd**, **glGenLists**, **glNewList**

glIsTexture

[New - Windows 95, OEM Service Release 2]

The **glIsTexture** function determines if a name corresponds to a texture.

```
GLboolean glIsTexture(
  GLuint texture
);
```

Parameters

texture
A value that is the name of a texture.

Remarks

If the *texture* parameter is currently the name of a texture, the **glIsTexture** function returns GL_TRUE. If *texture* is zero, is a non-zero value that is not currently the name of a texture, or if an error occurs, **glIsTexture** returns GL_FALSE.

You cannot include calls to **glIsTexture** in display lists.

Note The **glIsTexture** function is only available in OpenGL version 1.1 or later.

Error Codes

The following is the error code generated and its condition.

Error Code	Condition
GL_INVALID_OPERATION	**glIsTexture** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glBindTexture**, **glEnd**, **glGenTextures**, **glGet**, **glGetTexParameter**, **glTexImage1D**, **glTexImage2D**, **glTexParameter**

glLightf, glLighti, glLightfv, glLightiv

[New - Windows 95, OEM Service Release 2]

These functions set light source parameters.

```
void glLightf(
  GLenum light,
  GLenum pname,
  GLfloat param
);

void glLighti(
  GLenum light,
  GLenum pname,
  GLint param
);
```

Parameters

light

A light. The number of lights depends on the implementation, but at least eight lights are supported. They are identified by symbolic names of the form GL_LIGHTi where 0 ≤ *i* < GL_MAX_LIGHTS.

pname

A single-valued light source parameter for *light*. The following values are accepted.

GL_SPOT_EXPONENT

The *params* parameter is a single integer or floating-point value that specifies the intensity distribution of the light. Integer and floating-point values are mapped directly. Only values in the range [0,128] are accepted.

Effective light intensity is attenuated by the cosine of the angle between the direction of the light and the direction from the light to the vertex being lighted, raised to the power of the spot exponent. Thus, higher spot exponents result in a more focused light source, regardless of the spot cutoff angle (see the following paragraph). The default spot exponent is 0, resulting in uniform light distribution.

GL_SPOT_CUTOFF

The *params* parameter is a single integer or floating-point value that specifies the maximum spread angle of a light source. Integer and floating-point values are mapped directly. Only values in the range [0,90], and the special value 180, are accepted. If the angle between the direction of the light and the direction from the light to the vertex being lighted is greater than the spot cutoff angle, then the light is completely masked. Otherwise, its intensity is controlled by the spot exponent and the attenuation factors. The default spot cutoff is 180, resulting in uniform light distribution.

GL_CONSTANT_ATTENUATION
GL_LINEAR_ATTENUATION
GL_QUADRATIC_ATTENUATION

The *params* parameter is a single integer or floating-point value that specifies one of the three light attenuation factors. Integer and floating-point values are mapped directly. Only nonnegative values are accepted. If the light is positional, rather than directional, its intensity is attenuated by the reciprocal of the sum of: the constant factor, the linear factor multiplied by the distance between the light and the vertex being lighted, and the quadratic factor multiplied by the square of the same distance. The default attenuation factors are (1,0,0), resulting in no attenuation.

param

The value to which parameter *pname* of light source *light* will be set.

```
void glLightfv(
  GLenum light,
```

```
  GLenum pname,
  const GLfloat *params
);

void glLightiv(
  GLenum light,
  GLenum pname,
  const GLint *params
);
```

Parameters

light

A light. The number of lights depends on the implementation, but at least eight lights are supported. They are identified by symbolic names of the form GL_LIGHT*i* where $0 \leq i <$ GL_MAX_LIGHTS.

pname

A light source parameter for *light*. The following values are accepted:

GL_AMBIENT

The *params* parameter contains four integer or floating-point values that specify the ambient RGBA intensity of the light. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default ambient light intensity is (0.0, 0.0, 0.0, 1.0).

GL_DIFFUSE

The *params* parameter contains four integer or floating-point values that specify the diffuse RGBA intensity of the light. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default diffuse intensity is (0.0, 0.0, 0.0, 1.0) for all lights other than light zero. The default diffuse intensity of light zero is (1.0, 1.0, 1.0, 1.0).

GL_SPECULAR

The *params* parameter contains four integer or floating-point values that specify the specular RGBA intensity of the light. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default specular intensity is (0.0, 0.0, 0.0, 1.0) for all lights other than light zero. The default specular intensity of light zero is (1.0, 1.0, 1.0, 1.0).

GL_POSITION

The *params* parameter contains four integer or floating-point values that specify the position of the light in homogeneous object coordinates. Both integer and floating-point values are mapped directly. Neither integer nor floating-point values are clamped.

The position is transformed by the modelview matrix when **glLight** is called (just as if it were a point), and it is stored in eye coordinates. If the *w* component of the position is 0.0, the light is treated as a directional source. Diffuse and specular lighting calculations take the lights direction, but not its actual position, into account, and attenuation is disabled. Otherwise, diffuse and specular lighting calculations are based on the actual location of the light in eye coordinates, and attenuation is enabled. The default position is (0,0,1,0); thus, the default light source is directional, parallel to, and in the direction of the -z axis.

GL_SPOT_DIRECTION

The *params* parameter contains three integer or floating-point values that specify the direction of the light in homogeneous object coordinates. Both integer and floating-point values are mapped directly. Neither integer nor floating-point values are clamped.

The spot direction is transformed by the inverse of the modelview matrix when **glLight** is called (just as if it were a normal), and it is stored in eye coordinates. It is significant only when GL_SPOT_CUTOFF is not 180, which it is by default. The default direction is (0,0,-1).

GL_SPOT_EXPONENT
The *params* parameter is a single integer or floating-point value that specifies the intensity distribution of the light. Integer and floating-point values are mapped directly. Only values in the range [0,128] are accepted.

Effective light intensity is attenuated by the cosine of the angle between the direction of the light and the direction from the light to the vertex being lighted, raised to the power of the spot exponent. Thus, higher spot exponents result in a more focused light source, regardless of the spot cutoff angle (see the following paragraph). The default spot exponent is 0, resulting in uniform light distribution.

GL_SPOT_CUTOFF
The *params* parameter is a single integer or floating-point value that specifies the maximum spread angle of a light source. Integer and floating-point values are mapped directly. Only values in the range [0,90], and the special value 180, are accepted. If the angle between the direction of the light and the direction from the light to the vertex being lighted is greater than the spot cutoff angle, then the light is completely masked. Otherwise, its intensity is controlled by the spot exponent and the attenuation factors. The default spot cutoff is 180, resulting in uniform light distribution.

GL_CONSTANT_ATTENUATION
GL_LINEAR_ATTENUATION
GL_QUADRATIC_ATTENUATION
The *params* parameter is a single integer or floating-point value that specifies one of the three light attenuation factors. Integer and floating-point values are mapped directly. Only nonnegative values are accepted. If the light is positional, rather than directional, its intensity is attenuated by the reciprocal of the sum of: the constant factor, the linear factor multiplied by the distance between the light and the vertex being lighted, and the quadratic factor multiplied by the square of the same distance. The default attenuation factors are (1,0,0), resulting in no attenuation.

params
A pointer to the value or values to which parameter *pname* of light source *light* will be set.

Remarks

The **glLight** function sets the values of individual light source parameters. The *light* parameter names the light and is a symbolic name of the form:

GL_LIGHT*i*, where $0 \leq i <$ GL_MAX_LIGHTS

The *pname* parameter specifies one of ten light source parameters, again by symbolic name. The *params* parameter is either a single value or a pointer to an array that contains the new values.

Lighting calculation is enabled and disabled using **glEnable** and **glDisable** with argument GL_LIGHTING. When lighting is enabled, light sources that are enabled contribute to the lighting calculation. Light source *i* is enabled and disabled using **glEnable** and **glDisable** with argument GL_LIGHT*i*.

It is always the case that GL_LIGHT*i* = GL_LIGHT0 + *i*.

The following functions retrieve information related to the **glLight** function:

glGetLight
glIsEnabled with argument GL_LIGHTING

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*light* or *pname* was not an accepted value.

GL_INVALID_VALUE	a spot exponent value was specified outside the range [0,128], or if spot cutoff was specified outside the range [0,90] (except for the special value 180), or if a negative attenuation factor was specified.
GL_INVALID_OPERATION	**glLight** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glColorMaterial**, **glEnd**, **glLightModel**, **glMaterial**

glLightModelf, glLightModeli, glLightModelfv, glLightModeliv

[New - Windows 95, OEM Service Release 2]

These functions set the lighting model parameters.

```
void glLightModelf(
  GLenum pname,
  GLfloat param
);

void glLightModeli(
  GLenum pname,
  GLint param
);
```

Parameters

pname

A single-valued lighting model parameter. The following values are accepted:

GL_LIGHT_MODEL_LOCAL_VIEWER

The *params* parameter is a single integer or floating-point value that specifies how specular reflection angles are computed. If *params* is 0 (or 0.0), specular reflection angles take the view direction to be parallel to and in the direction of the -z axis, regardless of the location of the vertex in eye coordinates. Otherwise specular reflections are computed from the origin of the eye coordinate system. The default is 0.

GL_LIGHT_MODEL_TWO_SIDE

The *params* parameter is a single integer or floating-point value that specifies whether one- or two-sided lighting calculations are done for polygons. It has no effect on the lighting calculations for points, lines, or bitmaps. If *params* is 0 (or 0.0), one-sided lighting is specified, and only the *front* material parameters are used in the lighting equation. Otherwise, two-sided lighting is specified. In this case, vertices of back-facing polygons are lighted using the *back* material parameters, and have their normals reversed before the lighting equation is evaluated. Vertices of front-facing polygons are always lighted using the *front* material parameters, with no change to their normals. The default is 0.

param

The value to which *param* will be set.

```
void glLightModelfv(
  GLenum pname,
  const GLfloat *params
);

void glLightModeliv(
  GLenum pname,
  const GLint *params
);
```

Parameters

pname

A lighting model parameter. The following values are accepted:

GL_LIGHT_MODEL_AMBIENT

The *params* parameter contains four integer or floating-point values that specify the ambient RGBA intensity of the entire scene. Integer values are mapped linearly such that the most positive

representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default ambient scene intensity is (0.2, 0.2, 0.2, 1.0).

GL_LIGHT_MODEL_LOCAL_VIEWER
The *params* parameter is a single integer or floating-point value that specifies how specular reflection angles are computed. If *params* is 0 (or 0.0), specular reflection angles take the view direction to be parallel to and in the direction of the -z axis, regardless of the location of the vertex in eye coordinates. Otherwise specular reflections are computed from the origin of the eye coordinate system. The default is 0.

GL_LIGHT_MODEL_TWO_SIDE
The *params* parameter is a single integer or floating-point value that specifies whether one- or two-sided lighting calculations are done for polygons. It has no effect on the lighting calculations for points, lines, or bitmaps. If *params* is 0 (or 0.0), one-sided lighting is specified, and only the *front* material parameters are used in the lighting equation. Otherwise, two-sided lighting is specified. In this case, vertices of back-facing polygons are lighted using the *back* material parameters, and have their normals reversed before the lighting equation is evaluated. Vertices of front-facing polygons are always lighted using the *front* material parameters, with no change to their normals. The default is 0.

params
A pointer to the value or values to which *params* will be set.

Remarks

The **glLightModel** function sets the lighting model parameter. The *pname* parameter names a parameter and *params* gives the new value.

In RGBA mode, the lighted color of a vertex is the sum of the material emission intensity, the product of the material ambient reflectance and the lighting model full-scene ambient intensity, and the contribution of each enabled light source. Each light source contributes the sum of three terms: ambient, diffuse, and specular. The ambient light source contribution is the product of the material ambient reflectance and the light's ambient intensity. The diffuse light source contribution is the product of the material diffuse reflectance, the light's diffuse intensity, and the dot product of the vertex's normal with the normalized vector from the vertex to the light source. The specular light source contribution is the product of the material specular reflectance, the light's specular intensity, and the dot product of the normalized vertex-to-eye and vertex-to-light vectors, raised to the power of the shininess of the material. All three light source contributions are attenuated equally based on the distance from the vertex to the light source and on light source direction, spread exponent, and spread cutoff angle. All dot products are replaced with zero if they evaluate to a negative value.

The alpha component of the resulting lighted color is set to the alpha value of the material diffuse reflectance.

In color-index mode, the value of the lighted index of a vertex ranges from the ambient to the specular values passed to **glMaterial** using GL_COLOR_INDEXES. Diffuse and specular coefficients, computed with a (.30, .59, .11) weighting of the light's colors, the shininess of the material, and the same reflection and attenuation equations as in the RGBA case, determine how much above ambient the resulting index is.

The following functions retrieve information related to **glLightModel**:

glGet with argument GL_LIGHT_MODEL_AMBIENT
glGet with argument GL_LIGHT_MODEL_LOCAL_VIEWER
glGet with argument GL_LIGHT_MODEL_TWO_SIDE
glIsEnabled with argument GL_LIGHTING

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*pname* was not an accepted value.
GL_INVALID_OPERATION	**glLightModel** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glLight**, **glMaterial**

glLineStipple Quick Info

[New - Windows 95, OEM Service Release 2]

The **glLineStipple** function specifies the line stipple pattern.

```
void glLineStipple(
  GLint factor,
  GLushort pattern
);
```

Parameters

factor

A multiplier for each bit in the line stipple pattern. If *factor* is 3, for example, each bit in the pattern will be used three times before the next bit in the pattern is used. The *factor* parameter is clamped to the range [1, 256] and defaults to one.

pattern

A 16-bit integer whose bit pattern determines which fragments of a line will be drawn when the line is rasterized. Bit zero is used first, and the default pattern is all ones.

Remarks

The **glLineStipple** function specifies the line stipple pattern. Line stippling masks out certain fragments produced by rasterization; those fragments will not be drawn. The masking is achieved by using three parameters: the 16-bit line stipple pattern *pattern*, the repeat count *factor*, and an integer stipple counter *s*.

Counter *s* is reset to zero whenever **glBegin** is called, and before each line segment of a **glBegin(**GL_LINES**)**/**glEnd** sequence is generated. It is incremented after each fragment of a unit width aliased line segment is generated, or after each *i* fragments of an *i* width line segment are generated. The *i* fragments associated with count *s* are masked out if *pattern* bit (*s* *factor*) mod 16 is zero, otherwise these fragments are sent to the frame buffer. Bit zero of *pattern* is the least significant bit.

Antialiased lines are treated as a sequence of 1x*width* rectangles for purposes of stippling. Rectangle *s* is rasterized or not based on the fragment rule described for aliased lines; it counts rectangles rather than groups of fragments.

Line stippling is enabled or disabled using **glEnable** and **glDisable** with argument GL_LINE_STIPPLE. When enabled, the line stipple pattern is applied as described above. When disabled, it is as if the pattern were all ones. Initially, line stippling is disabled.

The following functions retrieve information related to **glLineStipple**:

glGet with argument GL_LINE_STIPPLE_PATTERN
glGet with argument GL_LINE_STIPPLE_REPEAT
glIsEnabled with argument GL_LINE_STIPPLE

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glLineStipple** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, glEnd, glLineWidth, glPolygonStipple

glLineWidth Quick Info

[New - Windows 95, OEM Service Release 2]

The **glLineWidth** function specifies the width of rasterized lines.

```
void glLineWidth(
  GLfloat width
);
```

Parameters

width
The width of rasterized lines. The default is 1.0.

Remarks

The **glLineWidth** function specifies the rasterized width of both aliased and antialiased lines. Using a line width other than 1.0 has different effects, depending on whether line antialiasing is enabled. Line antialiasing is controlled by calling **glEnable** and **glDisable** with argument GL_LINE_SMOOTH.

If line antialiasing is disabled, the actual width is determined by rounding the supplied width to the nearest integer. (If the rounding results in the value 0, it is as if the line width were 1.) If $|\Delta x| \geq |\Delta y|$, *i* pixels are filled in each column that is rasterized, where *i* is the rounded value of *width*. Otherwise, *i* pixels are filled in each row that is rasterized.

If antialiasing is enabled, line rasterization produces a fragment for each pixel square that intersects the region lying within the rectangle having width equal to the current line width, length equal to the actual length of the line, and centered on the mathematical line segment. The coverage value for each fragment is the window coordinate area of the intersection of the rectangular region with the corresponding pixel square. This value is saved and used in the final rasterization step.

Not all widths can be supported when line antialiasing is enabled. If an unsupported width is requested, the nearest supported width is used. Only width 1.0 is guaranteed to be supported; others depend on the implementation. The range of supported widths and the size difference between supported widths within the range can be queried by calling **glGet** with arguments GL_LINE_WIDTH_RANGE and GL_LINE_WIDTH_GRANULARITY.

The line width specified by **glLineWidth** is always returned when GL_LINE_WIDTH is queried. Clamping and rounding for aliased and antialiased lines have no effect on the specified value.

Non-antialiased line width may be clamped to an implementation-dependent maximum. Although this maximum cannot be queried, it must be no less than the maximum value for antialiased lines, rounded to the nearest integer value.

The following functions retrieve information related to **glLineWidth**:

glGet with argument GL_LINE_WIDTH
glGet with argument GL_LINE_WIDTH_RANGE
glGet with argument GL_LINE_WIDTH_GRANULARITY
glIsEnabled with argument GL_LINE_SMOOTH

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_VALUE	*width* was less than or equal to zero.

GL_INVALID_OPERATION	**glLineWidth** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnable**, **glEnd**, **glIsEnabled**

glListBase Quick Info

[New - Windows 95, OEM Service Release 2]

The **glListBase** function sets the display list base for **glCallLists.**

```
void glListBase(
  GLuint base
);
```

Parameters

base

An integer offset that will be added to **glCallLists** offsets to generate display list names. Initial value is zero.

Remarks

The **glListBase** function specifies an array of offsets. Display list names are generated by adding *base* to each offset. Names that reference valid display lists are executed; others are ignored.

The following function retrieves information related to **glListBase**:

glGet with argument GL_LIST_BASE

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glListBase** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glCallLists**, **glEnd**

glLoadIdentity Quick Info

[New - Windows 95, OEM Service Release 2]

The **glLoadIdentity** function replaces the current matrix with the identity matrix.

```
void glLoadIdentity(
  void
);
```

Remarks

The **glLoadIdentity** function replaces the current matrix with the identity matrix. It is semantically equivalent to calling **glLoadMatrix** with the identity matrix

{ewc msdncd, EWGraphic, bsd23545 0 /a "SDK.BMP"}

but in some cases it is more efficient.

The following functions retrieve information related to **glLoadIdentity**:

glGet with argument GL_MATRIX_MODE
glGet with argument GL_MODELVIEW_MATRIX
glGet with argument GL_PROJECTION_MATRIX
glGet with argument GL_TEXTURE_MATRIX

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glLoadIdentity** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glLoadMatrix**, **glMatrixMode**, **glMultMatrix**, **glPushMatrix**

glLoadMatrixd, glLoadMatrixf

[New - Windows 95, OEM Service Release 2]

The **glLoadMatrixd** and **glLoadMatrixf** functions replace the current matrix with an arbitrary matrix.

```
void glLoadMatrixd(
  const GLdouble *m
);

void glLoadMatrixf(
  const GLfloat *m
);
```

Parameters

m

A pointer to a 4x4 matrix stored in column-major order as 16 consecutive values.

Remarks

The **glLoadMatrix** function replaces the current matrix with the one specified in *m*. The current matrix is the projection matrix, modelview matrix, or texture matrix, determined by the current matrix mode (see **glMatrixMode**).

The *m* parameter points to a 4x4 matrix of single- or double-precision floating-point values stored in column-major order. That is, the matrix is stored as follows:

{ewc msdncd, EWGraphic, bsd23545 1 /a "SDK.WMF"}

The following functions retrieve information related to **glLoadMatrix**:

glGet with argument GL_MATRIX_MODE
glGet with argument GL_MODELVIEW_MATRIX
glGet with argument GL_PROJECTION_MATRIX
glGet with argument GL_TEXTURE_MATRIX

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glLoadMatrix** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glLoadIdentity**, **glMatrixMode**, **glMultMatrix**, **glPushMatrix**

glLoadName Quick Info

[New - Windows 95, OEM Service Release 2]

The **glLoadName** function loads a name onto the name stack.

```
void glLoadName(
  GLuint name
);
```

Parameters

name

A name that will replace the top value on the name stack.

Remarks

The **glLoadName** function causes *name* to replace the value on the top of the name stack, which is initially empty. The name stack is used during selection mode to allow sets of rendering commands to be uniquely identified. It consists of an ordered set of unsigned integers.

The name stack is always empty while the render mode is not GL_SELECT. Calls to **glLoadName** while the render mode is not GL_SELECT are ignored.

The following functions retrieve information related to **glLoadName**:

glGet with argument GL_NAME_STACK_DEPTH

glGet with argument GL_MAX_NAME_STACK_DEPTH

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glLoadName** was called while the name stack was empty.
GL_INVALID_OPERATION	**glLoadName** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glInitNames**, **glPushName**, **glRenderMode**, **glSelectBuffer**

glLogicOp Quick Info

[New - Windows 95, OEM Service Release 2]

The **glLogicOp** function specifies a logical pixel operation for color index rendering.

```
void glLogicOp(
  GLenum opcode
);
```

Parameters

opcode

A symbolic constant that selects a logical operation. The following symbols are accepted:

Opcode	Resulting Value
GL_CLEAR	0
GL_SET	1
GL_COPY	s
GL_COPY_INVERTED	!s
GL_NOOP	d
GL_INVERT	!d
GL_AND	s & d
GL_NAND	!(s & d)
GL_OR	s | d
GL_NOR	!(s | d)
GL_XOR	s ^ d
GL_EQUIV	!(s ^ d)
GL_AND_REVERSE	s & !d
GL_AND_INVERTED	!s & d
GL_OR_REVERSE	s | !d
GL_OR_INVERTED	!s | d

Remarks

The **glLogicOp** function specifies a logical operation that, when enabled, is applied between the incoming color index and the color index at the corresponding location in the frame buffer. The logical operation is enabled or disabled with **glEnable** and **glDisable** using the symbolic constant GL_LOGIC_OP.

The *opcode* parameter is a symbolic constant chosen from the list below. In the explanation of the logical operations, *s* represents the incoming color index and *d* represents the index in the frame buffer. Standard C-language operators are used. As these bitwise operators suggest, the logical operation is applied independently to each bit pair of the source and destination indexes.

Logical pixel operations are not applied to RGBA color buffers.

When more than one color-index buffer is enabled for drawing, logical operations are done separately for each enabled buffer, using the contents of that buffer for the destination index (see **glDrawBuffer**).

The *opcode* parameter must be one of the 16 accepted values. Other values result in an error.

The following functions retrieve information related to **glLogicOp**:

glGet with argument GL_LOGIC_OP_MODE

glIsEnabled with argument GL_LOGIC_OP

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*opcode* was not an accepted value.
GL_INVALID_OPERATION	**glLogicOp** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glAlphaFunc, **glBegin**, **glBlendFunc**, **glDrawBuffer**, **glEnable**, **glEnd**, **glIsEnabled**, **glStencilOp**

glMap1d, glMap1f

[New - Windows 95, OEM Service Release 2]

The **glMap1d** and **glMap1f** functions define a one-dimensional evaluator.

```
void glMap1d(
  GLenum target,
  GLdouble u1,
  GLdouble u2,
  GLint stride,
  GLint order,
  const GLdouble *points
);

void glMap1f(
  GLenum target,
  GLfloat u1,
  GLfloat u2,
  GLint stride,
  GLint order,
  const GLfloat *points
);
```

Parameters

target
The kind of values that are generated by the evaluator. Symbolic constants. The *target* parameter is a symbolic constant that indicates what kind of control points are provided in *points*, and what output is generated when the map is evaluated. It can assume one of nine predefined values:

GL_MAP1_VERTEX_3
Each control point is three floating-point values representing *x, y,* and *z*. Internal **glVertex3** commands are generated when the map is evaluated.

GL_MAP1_VERTEX_4
Each control point is four floating-point values representing *x, y, z,* and *w*. Internal **glVertex4** commands are generated when the map is evaluated.

GL_MAP1_INDEX
Each control point is a single floating-point value representing a color index. Internal **glIndex** commands are generated when the map is evaluated. However, the current index is not updated with the value of these **glIndex** commands.

GL_MAP1_COLOR_4
Each control point is four floating-point values representing red, green, blue, and alpha. Internal **glColor4** commands are generated when the map is evaluated. However, the current color is not updated with the value of these **glColor4** commands.

GL_MAP1_NORMAL
Each control point is three floating-point values representing the *x, y,* and *z* components of a normal vector. Internal **glNormal** commands are generated when the map is evaluated. However, the current normal is not updated with the value of these **glNormal** commands.

GL_MAP1_TEXTURE_COORD_1
Each control point is a single floating-point value representing the *s* texture coordinate. Internal **glTexCoord1** commands are generated when the map is evaluated. However, the current texture coordinates are not updated with the value of these **glTexCoord** commands.

GL_MAP1_TEXTURE_COORD_2
Each control point is two floating-point values representing the *s* and *t* texture coordinates. Internal **glTexCoord2** commands are generated when the map is evaluated. However, the current texture

coordinates are not updated with the value of these **glTexCoord** commands.

GL_MAP1_TEXTURE_COORD_3
Each control point is three floating-point values representing the *s, t,* and *r* texture coordinates. Internal **glTexCoord3** commands are generated when the map is evaluated. However, the current texture coordinates are not updated with the value of these **glTexCoord** commands.

GL_MAP1_TEXTURE_COORD_4
Each control point is four floating-point values representing the *s, t, r,* and *q* texture coordinates. Internal **glTexCoord4** commands are generated when the map is evaluated. However, the current texture coordinates are not updated with the value of these **glTexCoord** commands.

u1, u2
A linear mapping of *u*, as presented to **glEvalCoord1**, to *û*, the variable that is evaluated by the equations specified by this command.

stride
The number of floats or doubles between the beginning of one control point and the beginning of the next one in the data structure referenced in *points*. This allows control points to be embedded in arbitrary data structures. The only constraint is that the values for a particular control point must occupy contiguous memory locations.

order
The number of control points. Must be positive.

points
A pointer to the array of control points.

Remarks

Evaluators provide a way to use polynomial or rational polynomial mapping to produce vertices, normals, texture coordinates, and colors. The values produced by an evaluator are sent to further stages of OpenGL processing just as if they had been presented using **glVertex**, **glNormal**, **glTexCoord**, and **glColor** commands, except that the generated values do not update the current normal, texture coordinates, or color.

All polynomial or rational polynomial splines of any degree (up to the maximum degree supported by the OpenGL implementation) can be described using evaluators. These include almost all splines used in computer graphics, including B-splines, Bezier curves, Hermite splines, and so on.

Evaluators define curves based on Bernstein polynomials. Define **p**(û) as

{ewc msdncd, EWGraphic, bsd23545 2 /a "SDK.BMP"}

where $\mathbf{R}_{(i)}$ is a control point and $B_i^n(\hat{u})$ is the *i*th Bernstein polynomial of degree *n* (*order* = *n* + 1):

{ewc msdncd, EWGraphic, bsd23545 3 /a "SDK.BMP"}

Recall that

{ewc msdncd, EWGraphic, bsd23545 4 /a "SDK.BMP"}

The **glMap1** function is used to define the basis and to specify what kind of values are produced. Once defined, a map can be enabled and disabled by calling **glEnable** and **glDisable** with the map name, one of the nine predefined values for *target* described above. The **glEvalCoord1** function evaluates the one-dimensional maps that are enabled. When **glEvalCoord1** presents a value *u*, the Bernstein functions are evaluated using *û*, where

{ewc msdncd, EWGraphic, bsd23545 5 /a "SDK.WMF"}

The *stride*, *order*, and *points* parameters define the array addressing for accessing the control points. The *points* parameter is the location of the first control point, which occupies one, two, three, or four

contiguous memory locations, depending on which map is being defined. The *order* parameter is the number of control points in the array. The *stride* parameter tells how many float or double locations to advance the internal memory pointer to reach the next control point.

As is the case with all OpenGL commands that accept pointers to data, it is as if the contents of *points* were copied by **glMap1** before it returned. Changes to the contents of *points* have no effect after **glMap1** is called.

The following functions retrieve information related to **glMap1**:

glGet with argument GL_MAX_EVAL_ORDER
glGetMap
glIsEnabled with argument GL_MAP1_VERTEX_3
glIsEnabled with argument GL_MAP1_VERTEX_4
glIsEnabled with argument GL_MAP1_INDEX
glIsEnabled with argument GL_MAP1_COLOR_4
glIsEnabled with argument GL_MAP1_NORMAL
glIsEnabled with argument GL_MAP1_TEXTURE_COORD_1
glIsEnabled with argument GL_MAP1_TEXTURE_COORD_2
glIsEnabled with argument GL_MAP1_TEXTURE_COORD_3
glIsEnabled with argument GL_MAP1_TEXTURE_COORD_4

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*target* was not an accepted value.
GL_INVALID_VALUE	*u1* was equal to *u2*.
GL_INVALID_VALUE	*stride* was less than the number of values in a control point.
GL_INVALID_VALUE	*order* was less than one or greater than GL_MAX_EVAL_ORDER.
GL_INVALID_OPERATION	**glMap1** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glColor**, **glEnable**, **glEnd**, **glEvalCoord**, **glEvalMesh**, **glEvalPoint**, **glMap2**, **glMapGrid**, **glNormal**, **glTexCoord**, **glVertex**

glMap2d, glMap2f

[New - Windows 95, OEM Service Release 2]

The **glMap2d** and **glMap2f** functions define a two-dimensional evaluator.

```
void glMap2d(
  GLenum target,
  GLdouble u1,
  GLdouble u2,
  GLint ustride,
  GLint uorder,
  GLdouble v1,
  GLdouble v2,
  GLint vstride,
  GLint vorder,
  const GLdouble *points
);

void glMap2f(
  GLenum target,
  GLfloat u1,
  GLfloat u2,
  GLint ustride,
  GLint uorder,
  GLfloat v1,
  GLfloat v2,
  GLint vstride,
  GLint vorder,
  const GLfloat *points
);
```

Parameters

target

The kind of values that are generated by the evaluator. The following symbolic constants are accepted:

GL_MAP2_VERTEX_3
: Each control point is three floating-point values representing *x, y,* and *z*. Internal **glVertex3** commands are generated when the map is evaluated.

GL_MAP2_VERTEX_4
: Each control point is four floating-point values representing *x, y, z,* and *w*. Internal **glVertex4** commands are generated when the map is evaluated.

GL_MAP2_INDEX
: Each control point is a single floating-point value representing a color index. Internal **glIndex** commands are generated when the map is evaluated. The current index is not updated with the value of these **glIndex** commands, however.

GL_MAP2_COLOR_4
: Each control point is four floating-point values representing red, green, blue, and alpha. Internal **glColor4** commands are generated when the map is evaluated. The current color is not updated with the value of these **glColor4** commands, however.

GL_MAP2_NORMAL
: Each control point is three floating-point values representing the *x, y,* and *z* components of a normal vector. Internal **glNormal** commands are generated when the map is evaluated. The current normal is not updated with the value of these **glNormal** commands, however.

GL_MAP2_TEXTURE_COORD_1
Each control point is a single floating-point value representing the *s* texture coordinate. Internal **glTexCoord1** commands are generated when the map is evaluated. The current texture coordinates are not updated with the value of these **glTexCoord** commands, however.

GL_MAP2_TEXTURE_COORD_2
Each control point is two floating-point values representing the *s* and *t* texture coordinates. Internal **glTexCoord2** commands are generated when the map is evaluated. The current texture coordinates are not updated with the value of these **glTexCoord** commands, however.

GL_MAP2_TEXTURE_COORD_3
Each control point is three floating-point values representing the *s, t,* and *r* texture coordinates. Internal **glTexCoord3** commands are generated when the map is evaluated. The current texture coordinates are not updated with the value of these **glTexCoord** commands, however.

GL_MAP2_TEXTURE_COORD_4
Each control point is four floating-point values representing the *s, t, r,* and *q* texture coordinates. Internal **glTexCoord4** commands are generated when the map is evaluated. The current texture coordinates are not updated with the value of these **glTexCoord** commands, however.

u1, u2
A linear mapping of *u*, as presented to **glEvalCoord2**, to $\hat{u}$, one of the two variables that is evaluated by the equations specified by this command.

ustride
The number of floats or doubles between the beginning of control point $\mathbf{R}_{(ij)}$ and the beginning of control point $\mathbf{R}_{((i+1)j)}$, where *i* and *j* are the *u* and *v* control point indexes, respectively. This allows control points to be embedded in arbitrary data structures. The only constraint is that the values for a particular control point must occupy contiguous memory locations.

uorder
The dimension of the control point array in the *u*-axis. Must be positive.

v1, v2
A linear mapping of *v*, as presented to **glEvalCoord2**, to $\hat{v}$, one of the two variables that is evaluated by the equations specified by this command.

vstride
The number of floats or doubles between the beginning of control point $\mathbf{R}_{(ij)}$ and the beginning of control point $\mathbf{R}_{(i(j+1))}$, where *i* and *j* are the *u* and *v* control point indexes, respectively. This allows control points to be embedded in arbitrary data structures. The only constraint is that the values for a particular control point must occupy contiguous memory locations.

vorder
The dimension of the control point array in the *v*-axis. Must be positive.

points
A pointer to the array of control points.

Remarks

Evaluators provide a way to use polynomial or rational polynomial mapping to produce vertices, normals, texture coordinates, and colors. The values produced by an evaluator are sent on to further stages of OpenGL processing just as if they had been presented using **glVertex**, **glNormal**, **glTexCoord**, and **glColor** commands, except that the generated values do not update the current normal, texture coordinates, or color.

All polynomial or rational polynomial splines of any degree (up to the maximum degree supported by the OpenGL implementation) can be described using evaluators. These include almost all surfaces used in computer graphics, including B-spline surfaces, NURBS surfaces, Bezier surfaces, and so on.

Evaluators define surfaces based on bivariate Bernstein polynomials. Define $\mathbf{p}(\hat{u},\hat{v})$ as

{ewc msdncd, EWGraphic, bsd23545 6 /a "SDK.BMP"}

where **R** *(ij)* is a control point, $B_i^n(\hat{u})$ is the *i*th Bernstein polynomial of degree

n (*uorder* = *n* + 1)
{ewc msdncd, EWGraphic, bsd23545 7 /a "SDK.BMP"}
and ($B_j^m(\hat{v})$) is the *j*th Bernstein polynomial of degree *m* (*vorder* = *m* + 1)
{ewc msdncd, EWGraphic, bsd23545 8 /a "SDK.BMP"}
Recall that
{ewc msdncd, EWGraphic, bsd23545 9 /a "SDK.BMP"}
The **glMap2** function is used to define the basis and to specify what kind of values are produced. Once defined, a map can be enabled and disabled by calling **glEnable** and **glDisable** with the map name, one of the nine predefined values for *target*, described above. When **glEvalCoord2** presents values *u* and *v*, the bivariate Bernstein polynomials are evaluated using $\hat{u}$ and Quick Info, where

{ewc msdncd, EWGraphic, bsd23545 10 /a "SDK.BMP"}
{ewc msdncd, EWGraphic, bsd23545 11 /a "SDK.BMP"}

The *target* parameter is a symbolic constant that indicates what kind of control points are provided in *points*, and what output is generated when the map is evaluated.

The *ustride*, *uorder*, *vstride*, *vorder*, and *points* parameters define the array addressing for accessing the control points. The *points* parameter is the location of the first control point, which occupies one, two, three, or four contiguous memory locations, depending on which map is being defined. There are *uorder*x*vorder* control points in the array. The *ustride* parameter tells how many float or double locations are skipped to advance the internal memory pointer from control point **R** *(ij)* to control point **R** *((i+1)j)*. The *vstride* parameter tells how many float or double locations are skipped to advance the internal memory pointer from control point **R** *(ij)* to control point **R** *(i(j+1))*.

As is the case with all OpenGL commands that accept pointers to data, it is as if the contents of *points* were copied by **glMap2** before it returned. Changes to the contents of *points* have no effect after **glMap2** is called.

The following functions retrieve information related to **glMap2**:

glGet with argument GL_MAX_EVAL_ORDER
glGetMap
glIsEnabled with argument GL_MAP2_VERTEX_3
glIsEnabled with argument GL_MAP2_VERTEX_4
glIsEnabled with argument GL_MAP2_INDEX
glIsEnabled with argument GL_MAP2_COLOR_4
glIsEnabled with argument GL_MAP2_NORMAL
glIsEnabled with argument GL_MAP2_TEXTURE_COORD_1
glIsEnabled with argument GL_MAP2_TEXTURE_COORD_2
glIsEnabled with argument GL_MAP2_TEXTURE_COORD_3
glIsEnabled with argument GL_MAP2_TEXTURE_COORD_4

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*target* was not an accepted value.
GL_INVALID_VALUE	*u1* was equal to *u2*, or if *v1* was equal to *v2*.
GL_INVALID_VALUE	either *ustride* or *vstride* was less than

	the number of values in a control point.
GL_INVALID_VALUE	either *uorder* or *vorder* was less than one or greater than GL_MAX_EVAL_ORDER.
GL_INVALID_OPERATION	**glMap2** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glColor**, **glEnable**, **glEnd**, **glEvalCoord**, **glEvalMesh**, **glEvalPoint**, **glMap1**, **glMapGrid**, **glNormal**, **glTexCoord**, **glVertex**

glMapGrid1d, glMapGrid1f, glMapGrid2d, glMapGrid2f

[New - Windows 95, OEM Service Release 2]

These functions define a one- or two-dimensional mesh.

```
void glMapGrid1d(
  GLint un,
  GLdouble u1,
  GLdouble u2
);

void glMapGrid1f(
  GLint un,
  GLfloat u1,
  GLfloat u2
);

void glMapGrid2d(
  GLint un,
  GLdouble u1,
  GLdouble u2,
  GLint vn,
  GLdouble v1,
  GLdouble v2
);

void glMapGrid2f(
  GLint un,
  GLfloat u1,
  GLfloat u2,
  GLint vn,
  GLfloat v1,
  GLfloat v2
);
```

Parameters

un
The number of partitions in the grid range interval [*u1*, *u2*]. Must be positive.

u1, u2
The mappings for integer grid domain values *i* = 0 and *i* = *un*.

vn
The number of partitions in the grid range interval [*v1*, *v2*] (**glMapGrid2** only).

v1, v2
The mappings for integer grid domain values *j* = 0 and *j* = *vn* (**glMapGrid2** only).

Remarks

The **glMapGrid** and **glEvalMesh** functions are used in tandem to efficiently generate and evaluate a series of evenly spaced map domain values. The **glEvalMesh** function steps through the integer domain of a one- or two-dimensional grid, whose range is the domain of the evaluation maps specified by **glMap1** and **glMap2**.

The **glMapGrid1** and **glMapGrid2** functions specify the linear grid mappings between the *i* (or *i* and *j*)

integer grid coordinates, to the *u* (or *u* and *v*) floating-point evaluation map coordinates. See **glMap1** and **glMap2** for details of how *u* and *v* coordinates are evaluated.

The **glMapGrid1** function specifies a single linear mapping such that integer grid coordinate 0 maps exactly to *u1*, and integer grid coordinate *un* maps exactly to *u2*. All other integer grid coordinates *i* are mapped such that:

u = *i*(*u*2 - *u*1)/*un* + *u1*

The **glMapGrid2** function specifies two such linear mappings. One maps integer grid coordinate *i* = 0 exactly to *u1*, and integer grid coordinate *i* = *un* exactly to *u2*. The other maps integer grid coordinate *j* = 0 exactly to *v1*, and integer grid coordinate *j* = *vn* exactly to *v2*. Other integer grid coordinates *i* and *j* are mapped such that

u = *i*(*u*2 - *u*1)/*un* + *u*1
v = *j* (*v*2 - *v*1)/*vn* + *v*1

The mappings specified by **glMapGrid** are used identically by **glEvalMesh** and **glEvalPoint**.

The following functions retrieve information related to **glMapGrid**:

glGet with argument GL_MAP1_GRID_DOMAIN
glGet with argument GL_MAP2_GRID_DOMAIN
glGet with argument GL_MAP1_GRID_SEGMENTS
glGet with argument GL_MAP2_GRID_SEGMENTS

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_VALUE	Either *un* or *vn* was not positive.
GL_INVALID_OPERATION	**glMapGrid** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glEvalCoord**, **glEvalMesh**, **glEvalPoint**, **glMap1**, **glMap2**

glMaterialf, glMaterialiv, glMaterialfv, glMaterialiv

[New - Windows 95, OEM Service Release 2]

These functions specify material parameters for the lighting model.

```
void glMaterialf(
  GLenum face,
  GLenum pname,
  GLfloat param
);

void glMateriali(
  GLenum face,
  GLenum pname,
  GLint param
);
```

Parameters

face
The face or faces that are being updated. Must be one of GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK.

pname
The single-valued material parameter of the face or faces being updated. Must be GL_SHININESS.

param
The value that parameter GL_SHININESS will be set to.

```
void glMaterialfv(
  GLenum face,
  GLenum pname,
  const GLfloat *params
);

void glMaterialiv(
  GLenum face,
  GLenum pname,
  const GLint *params
);
```

Parameters

face
The face or faces that are being updated. Must be one of GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK.

pname
The material parameter of the face or faces being updated. The parameters that can be specified using **glMaterial**, and their interpretations by the lighting equation, are as follows:

GL_AMBIENT
The *params* parameter contains four integer or floating-point values that specify the ambient RGBA reflectance of the material. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default ambient reflectance for both front- and back-facing materials is (0.2, 0.2, 0.2, 1.0).

GL_DIFFUSE

The *params* parameter contains four integer or floating-point values that specify the diffuse RGBA reflectance of the material. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default diffuse reflectance for both front- and back-facing materials is (0.8, 0.8, 0.8, 1.0).

GL_SPECULAR
The *params* parameter contains four integer or floating-point values that specify the specular RGBA reflectance of the material. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default specular reflectance for both front- and back-facing materials is (0.0, 0.0, 0.0, 1.0).

GL_EMISSION
The *params* parameter contains four integer or floating-point values that specify the RGBA emitted light intensity of the material. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default emission intensity for both front- and back-facing materials is (0.0, 0.0, 0.0, 1.0).

GL_SHININESS
The *params* parameter is a single integer or floating-point value that specifies the RGBA specular exponent of the material. Integer and floating-point values are mapped directly. Only values in the range [0,128] are accepted. The default specular exponent for both front- and back-facing materials is 0.

GL_AMBIENT_AND_DIFFUSE
Equivalent to calling **glMaterial** twice with the same parameter values, once with GL_AMBIENT and once with GL_DIFFUSE.

GL_COLOR_INDEXES
The *params* parameter contains three integer or floating-point values specifying the color indexes for ambient, diffuse, and specular lighting. These three values, and GL_SHININESS, are the only material values used by the color-index mode lighting equation. Refer to **glLightModel** for a discussion of color-index lighting.

params
A pointer to the value or values to which *pname* will be set.

Remarks

The **glMaterial** function assigns values to material parameters. There are two matched sets of material parameters. One, the *front-facing* set, is used to shade points, lines, bitmaps, and all polygons (when two-sided lighting is disabled), or just front-facing polygons (when two-sided lighting is enabled). The other set, *back-facing*, is used to shade back-facing polygons only when two-sided lighting is enabled. Refer to **glLightModel** for details concerning one- and two-sided lighting calculations.

The **glMaterial** function takes three arguments. The first, *face*, specifies whether the GL_FRONT materials, the GL_BACK materials, or both GL_FRONT_AND_BACK materials will be modified. The second, *pname*, specifies which of several parameters in one or both sets will be modified. The third, *params*, specifies what value or values will be assigned to the specified parameter.

Material parameters are used in the lighting equation that is optionally applied to each vertex. The equation is discussed in **glLightModel**.

The material parameters can be updated at any time. In particular, **glMaterial** can be called between a call to **glBegin** and the corresponding call to **glEnd**. If only a single material parameter is to be changed per vertex, however, **glColorMaterial** is preferred over **glMaterial**.

The following function retrieves information related to **glMaterial**:

glGetMaterial

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	Either *face* or *pname* was not an accepted value.
GL_INVALID_VALUE	A specular exponent outside the range [0,128] was specified.

See Also

glColorMaterial, **glLight**, **glLightModel**

glMatrixMode Quick Info

[New - Windows 95, OEM Service Release 2]

The **glMatrixMode** function specifies which matrix is the current matrix.

```
void glMatrixMode(
  GLenum mode
);
```

Parameters

mode

The matrix stack that is the target for subsequent matrix operations. The *mode* parameter can assume one of three values:

GL_MODELVIEW
: Applies subsequent matrix operations to the modelview matrix stack.

GL_PROJECTION
: Applies subsequent matrix operations to the projection matrix stack.

GL_TEXTURE
: Applies subsequent matrix operations to the texture matrix stack.

Remarks

The **glMatrixMode** function sets the current matrix mode.

The following function retrieves information related to **glMatrixMode**:

glGet with argument GL_MATRIX_MODE

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*mode* was not an accepted value.
GL_INVALID_OPERATION	**glMatrixMode** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glLoadMatrix**, **glPushMatrix**

glMultMatrixd, glMultMatrixf

[New - Windows 95, OEM Service Release 2]

The **glMultMatrixd** and **glMultMatrixf** functions multiply the current matrix by an arbitrary matrix.

```
void glMultMatrixd(
  const GLdouble *m
);

void glMultMatrixf(
  const GLfloat *m
);
```

Parameters

m

A pointer to a 4x4 matrix stored in column-major order as 16 consecutive values.

Remarks

The **glMultMatrix** function multiplies the current matrix by the one specified in *m*. That is, if M is the current matrix and T is the matrix passed to **glMultMatrix**, then M is replaced with M • T.

The current matrix is the projection matrix, modelview matrix, or texture matrix, determined by the current matrix mode (see **glMatrixMode**).

The *m* parameter points to a 4x4 matrix of single- or double-precision floating-point values stored in column-major order. That is, the matrix is stored as:

{ewc msdncd, EWGraphic, bsd23545 12 /a "SDK.BMP"}

The following functions retrieve information related to **glMultMatrix**:

glGet with argument GL_MATRIX_MODE
glGet with argument GL_MODELVIEW_MATRIX
glGet with argument GL_PROJECTION_MATRIX
glGet with argument GL_TEXTURE_MATRIX

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glMultMatrix** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glLoadIdentity**, **glLoadMatrix**, **glMatrixMode**, **glPushMatrix**

glNewList, glEndList

[New - Windows 95, OEM Service Release 2]

The **glNewList** and **glEndList** functions create or replace a display list.

```
void glNewList(
  GLuint list,
  GLenum mode
);

void glEndList(
  void
);
```

Parameters

list
The display list name.

mode
The compilation mode. The following values are accepted:

GL_COMPILE
Commands are merely compiled.

GL_COMPILE_AND_EXECUTE
Commands are executed as they are compiled into the display list.

Remarks

Display lists are groups of OpenGL commands that have been stored for subsequent execution. The display lists are created with **glNewList**. All subsequent commands are placed in the display list, in the order issued, until **glEndList** is called.

The **glNewList** function has two parameters. The first parameter, *list*, is a positive integer that becomes the unique name for the display list. Names can be created and reserved with **glGenLists** and tested for uniqueness with **glIsList**. The second parameter, *mode*, is a symbolic constant that can assume one of the two preceding values.

Certain commands are not compiled into the display list, but are executed immediately, regardless of the display list mode. These commands are **glIsList**, **glGenLists**, **glDeleteLists**, **glFeedbackBuffer**, **glSelectBuffer**, **glRenderMode**, **glReadPixels**, **glPixelStore**, **glFlush**, **glFinish**, **glIsEnabled**, and all of the **glGet** routines.

When the **glNewList** function is encountered, the display list definition is completed by associating the list with the unique name *list* (specified in the **glNewList** command). If a display list with name *list* already exists, it is replaced only when **glEndList** is called.

The **glCallList** and **glCallLists** functions can be entered into display lists. The commands in the display list or lists executed by **glCallList** or **glCallLists** are not included in the display list being created, even if the list creation mode is GL_COMPILE_AND_EXECUTE.

The following function retrieves information related to **glNewList**:

glGet with argument GL_MATRIX_MODE

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_VALUE	*list* was zero.
GL_INVALID_ENUM	*mode* was not an accepted value.
GL_INVALID_OPERATION	**glEndList** was called without a preceding **glNewList**, or if **glNewList** was called while a display list was being defined.
GL_INVALID_OPERATION	**glNewList** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glCallList**, **glCallLists**, **glDeleteLists**, **glEnd**, **glGenLists**, **glIsList**

glNormal

[New - Windows 95, OEM Service Release 2]

glNormal3b, glNormal3d, glNormal3f, glNormal3i, glNormal3s, glNormal3bv, glNormal3dv, glNormal3fv, glNormal3iv, glNormal3sv

These functions set the current normal vector.

```
void glNormal3b(
  GLbyte nx,
  GLbyte ny,
  GLbyte nz
);

void glNormal3d(
  GLdouble nx,
  GLdouble ny,
  GLdouble nz
);

void glNormal3f(
  GLfloat nx,
  GLfloat ny,
  GLfloat nz
);

void glNormal3i(
  GLint nx,
  GLint ny,
  GLint nz
);

void glNormal3s(
  GLshort nx,
  GLshort ny,
  GLshort nz
);
```

Parameters

nx, ny, nz
The *x, y,* and *z* coordinates of the new current normal. The initial value of the current normal is (0,0,1).

```
void glNormal3bv(
  const GLbyte *v
);

void glNormal3dv(
  const GLdouble *v
);

void glNormal3fv(
  const GLfloat *v
);

void glNormal3iv(
  const GLint *v
);
```

```
void glNormal3sv(
  const GLshort *v
);
```

Parameters

v

A pointer to an array of three elements: the *x, y,* and *z* coordinates of the new current normal.

Remarks

The current normal is set to the given coordinates whenever **glNormal** is issued. Byte, short, or integer arguments are converted to floating-point format with a linear mapping that maps the most positive representable integer value to 1.0, and the most negative representable integer value to -1.0.

Normals specified with **glNormal** need not have unit length. If normalization is enabled, then normals specified with **glNormal** are normalized after transformation. Normalization is controlled using **glEnable** and **glDisable** with the argument GL_NORMALIZE. By default, normalization is disabled.

The current normal can be updated at any time. In particular, **glNormal** can be called between a call to **glBegin** and the corresponding call to **glEnd**.

The following functions retrieve information related to **glNormal**:

glGet with argument GL_CURRENT_NORMAL
glIsEnable with argument GL_NORMALIZE

See Also

glBegin, **glColor**, **glEnd**, **glIndex**, **glTexCoord**, **glVertex**

glNormalPointer

[New - Windows 95, OEM Service Release 2]

The **glNormalPointer** function defines an array of normals.

```
void glNormalPointer(
  GLenum type,
  GLsizei stride,
  GLsizei count,
  const GLvoid *pointer
);
```

Parameters

type
The data type of each coordinate in the array using the following symbolic constants: GL_BYTE, GL_SHORT, GL_INT, GL_FLOAT, and GL_DOUBLE.

stride
The byte offset between consecutive normals. When *stride* is zero, the normals are tightly packed in the array.

count
The number of normals, counting from the first, that are static.

pointer
A pointer to the first normal in the array.

Remarks

The **glNormalPointer** function specifies the location and data of an array of normals to use when rendering. The *type* parameter specifies the data type of each normal coordinate. The *stride* parameter determines the byte offset from one normal to the next, enabling the packing of vertices and attributes in a single array or storage in separate arrays. In some implementations storing the vertices and attributes in a single array can be more efficient than using separate arrays. Starting from the first normal element, *count* indicates the total number of static elements. Your application can modify static elements, but once the elements are modified, the application must explicitly specify the array again before using the array for any rendering. Non-static array elements are not accessed until you call **glDrawArrays** or **glArrayElement**.

A normal array is enabled when you specify the GL_NORMAL_ARRAY constant with **glEnableClientState**. When enabled, **glDrawArrays** and **glArrayElement** use the normal array. By default the normal array is disabled.

You cannot include **glNormalPointer** in display lists.

When you specify a normal array using **glNormalPointer**, the values of all the function's normal array parameters are saved in a client-side state and static array elements can be cached. Because the normal array parameters are saved in a client-side state, their values are not saved or restored by **glPushAttrib** and **glPopAttrib**.

Although no error is generated when you call **glNormalPointer** within **glBegin** and **glEnd** pairs, the results are undefined.

The following functions are associated with **glNormalPointer**:

glGet with argument GL_NORMAL_ARRAY_STRIDE
glGet with argument GL_NORMAL_ARRAY_COUNT
glGet with argument GL_NORMAL_ARRAY_TYPE

glGetPointerv with argument GL_NORMAL_ARRAY_POINTER
glIsEnabled with argument GL_NORMAL_ARRAY

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*type* was not an accepted value.
GL_INVALID_VALUE	*stride* or *count* was negative.

See Also

glArrayElement, **glColorPointer**, **glDrawArrays**, **glEdgeFlagPointer**, **glGetPointerv**, **glIndexPointer**, **glIsEnabled**, **glTexCoordPointer**, **glVertexPointer**, **glGetString**

glOrtho Quick Info

[New - Windows 95, OEM Service Release 2]

The **glOrtho** function multiplies the current matrix by an orthographic matrix.

```
void glOrtho(
  GLdouble left,
  GLdouble right,
  GLdouble bottom,
  GLdouble top,
  GLdouble near,
  GLdouble far
);
```

Parameters

left, right
The coordinates for the left and right vertical clipping planes.

bottom, top
The coordinates for the bottom and top horizontal clipping planes.

near, far
The distances to the nearer and farther depth clipping planes. These distances are negative if the plane is to be behind the viewer.

Remarks

The **glOrtho** function describes a perspective matrix that produces a parallel projection. The (*left*, *bottom*, - *near*) and (*right*, *top*, - *near*) parameters specify the points on the near clipping plane that are mapped to the lower-left and upper-right corners of the window, respectively, assuming that the eye is located at (0, 0, 0). The -*far* parameter specifies the location of the far clipping plane. Both *near* and *far* can be either positive or negative. The corresponding matrix is

{ewc msdncd, EWGraphic, bsd23545 13 /a "SDK.BMP"}

where

{ewc msdncd, EWGraphic, bsd23545 14 /a "SDK.BMP"}

The current matrix is multiplied by this matrix with the result replacing the current matrix. That is, if M is the current matrix and O is the ortho matrix, then M is replaced with M • O.

Use **glPushMatrix** and **glPopMatrix** to save and restore the current matrix stack.

The following functions retrieve information related to **glOrtho**:

glGet with argument GL_MATRIX_MODE
glGet with argument GL_MODELVIEW_MATRIX
glGet with argument GL_PROJECTION_MATRIX
glGet with argument GL_TEXTURE_MATRIX

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glOrtho** was called between a call to

glBegin and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glFrustum**, **glMatrixMode**, **glMultMatrix**, **glPushMatrix**, **glViewport**

glPassThrough Quick Info

[New - Windows 95, OEM Service Release 2]

The **glPassThrough** function places a marker in the feedback buffer.

```
void glPassThrough(
  GLfloat token
);
```

Parameters

token

A marker value to be placed in the feedback buffer. It is indicated with the following unique identifying value:

GL_PASS_THROUGH_TOKEN

The order of **glPassThrough** commands with respect to the specification of graphics primitives is maintained.

Remarks

Feedback is an OpenGL render mode. The mode is selected by calling **glRenderMode** with GL_FEEDBACK. When OpenGL is in feedback mode, no pixels are produced by rasterization. Instead, information about primitives that would have been rasterized is fed back to the application using OpenGL. See **glFeedbackBuffer** for a description of the feedback buffer and the values in it.

The **glPassThrough** function inserts a user-defined marker in the feedback buffer when it is executed in feedback mode. The *token* parameter is returned as if it were a primitive.

The **glPassThrough** function is ignored if OpenGL is not in feedback mode.

The following function retrieves information related to **glPassThrough**:

glGet with argument GL_RENDER_MODE

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glPassThrough** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glFeedbackBuffer**, **glRenderMode**

glPixelMapfv, glPixelMapuiv, glPixelMapusv

[New - Windows 95, OEM Service Release 2]

These functions set up pixel transfer maps.

```
void glPixelMapfv(
  GLenum map,
  GLint mapsize,
  const GLfloat *values
);

void glPixelMapuiv(
  GLenum map,
  GLint mapsize,
  const GLuint *values
);

void glPixelMapusv(
  GLenum map,
  GLint mapsize,
  const GLushort *values
);
```

Parameters

map
A symbolic map name. The ten maps are as follows:

GL_PIXEL_MAP_I_TO_I
Maps color indexes to color indexes.

GL_PIXEL_MAP_S_TO_S
Maps stencil indexes to stencil indexes.

GL_PIXEL_MAP_I_TO_R
Maps color indexes to red components.

GL_PIXEL_MAP_I_TO_G
Maps color indexes to green components.

GL_PIXEL_MAP_I_TO_B
Maps color indexes to blue components.

GL_PIXEL_MAP_I_TO_A
Maps color indexes to alpha components.

GL_PIXEL_MAP_R_TO_R
Maps red components to red components.

GL_PIXEL_MAP_G_TO_G
Maps green components to green components.

GL_PIXEL_MAP_B_TO_B
Maps blue components to blue components.

GL_PIXEL_MAP_A_TO_A
Maps alpha components to alpha components.

mapsize
The size of the map being defined.

values
An array of *mapsize* values.

Remarks

The **glPixelMap** function sets up translation tables, or *maps*, used by **glDrawPixels**, **glReadPixels**, **glCopyPixels**, **glTexImage1D**, and **glTexImage2D**. Use of these maps is described completely in the **glPixelTransfer** topic, and partly in the topics for the pixel and texture image commands. Only the specification of the maps is described in this topic.

The *map* parameter is a symbolic map name, indicating one of ten maps to set. The *mapsize* parameter specifies the number of entries in the map, and *values* is a pointer to an array of *mapsize* map values.

The entries in a map can be specified as single-precision floating-point numbers, unsigned short integers, or unsigned long integers. Maps that store color component values (all but GL_PIXEL_MAP_I_TO_I and GL_PIXEL_MAP_S_TO_S) retain their values in floating-point format, with unspecified mantissa and exponent sizes. Floating-point values specified by **glPixelMapfv** are converted directly to the internal floating-point format of these maps, and then clamped to the range [0,1]. Unsigned integer values specified by **glPixelMapusv** and **glPixelMapuiv** are converted linearly such that the largest representable integer maps to 1.0, and zero maps to 0.0.

Maps that store indexes, GL_PIXEL_MAP_I_TO_I and GL_PIXEL_MAP_S_TO_S, retain their values in fixed-point format, with an unspecified number of bits to the right of the binary point. Floating-point values specified by **glPixelMapfv** are converted directly to the internal fixed-point format of these maps. Unsigned integer values specified by **glPixelMapusv** and **glPixelMapuiv** specify integer values, with all zeros to the right of the binary point.

The following table shows the initial sizes and values for each of the maps. Maps that are indexed by either color or stencil indexes must have *mapsize* = 2^*n* for some *n* or results are undefined. The maximum allowable size for each map depends on the implementation and can be determined by calling **glGet** with argument GL_MAX_PIXEL_MAP_TABLE. The single maximum applies to all maps, and it is at least 32.

GL_PIXEL_MAP_I_TO_I

Lookup Index:	color index
Lookup Value:	color index
Initial Size:	1
Initial Value:	0.0

GL_PIXEL_MAP_S_TO_S

Lookup Index:	stencil index
Lookup Value:	stencil index
Initial Size:	1
Initial Value:	0.0

GL_PIXEL_MAP_I_TO_R

Lookup Index:	color index
Lookup Value:	R
Initial Size:	1
Initial Value:	0.0

GL_PIXEL_MAP_I_TO_G

Lookup Index:	color index
Lookup Value:	G

Initial Size: 1
Initial Value: 0.0

GL_PIXEL_MAP_I_TO_B

Lookup Index: color index
Lookup Value: B
Initial Size: 1
Initial Value: 0.0

GL_PIXEL_MAP_I_TO_A

Lookup Index: color index
Lookup Value: A
Initial Size: 1
Initial Value: 0.0

GL_PIXEL_MAP_R_TO_R

Lookup Index: R
Lookup Value: R
Initial Size: 1
Initial Value: 0.0

GL_PIXEL_MAP_G_TO_G

Lookup Index: G
Lookup Value: G
Initial Size: 1
Initial Value: 0.0

GL_PIXEL_MAP_B_TO_B

Lookup Index: B
Lookup Value: B
Initial Size: 1
Initial Value: 0.0

GL_PIXEL_MAP_A_TO_A

Lookup Index: A
Lookup Value: A
Initial Size: 1
Initial Value: 0.0

The following functions retrieve information related to **glPixelMap**:

glGet with argument GL_PIXEL_MAP_I_TO_I_SIZE
glGet with argument GL_PIXEL_MAP_S_TO_S_SIZE
glGet with argument GL_PIXEL_MAP_I_TO_R_SIZE
glGet with argument GL_PIXEL_MAP_I_TO_G_SIZE
glGet with argument GL_PIXEL_MAP_I_TO_B_SIZE

glGet with argument GL_PIXEL_MAP_I_TO_A_SIZE
glGet with argument GL_PIXEL_MAP_R_TO_R_SIZE
glGet with argument GL_PIXEL_MAP_G_TO_G_SIZE
glGet with argument GL_PIXEL_MAP_B_TO_B_SIZE
glGet with argument GL_PIXEL_MAP_A_TO_A_SIZE
glGet with argument GL_MAX_PIXEL_MAP_TABLE

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*map* was not an accepted value.
GL_INVALID_VALUE	*mapsize* was negative or larger than GL_MAX_PIXEL_MAP_TABLE.
GL_INVALID_VALUE	*map* was GL_PIXEL_MAP_I_TO_I, GL_PIXEL_MAP_S_TO_S, GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, or GL_PIXEL_MAP_I_TO_A, and *mapsize* was not a power of two.
GL_INVALID_OPERATION	**glPixelMap** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glCopyPixels**, **glDrawPixels**, **glEnd**, **glPixelStore**, **glPixelTransfer**, **glReadPixels**, **glTexImage1D**, **glTexImage2D**

glPixelStoref, glPixelStorei

[New - Windows 95, OEM Service Release 2]

The **glPixelStoref** and **glPixelStorei** functions set pixel storage modes.

```
void glPixelStoref(
  GLenum pname,
  GLfloat param
);

void glPixelStorei(
  GLenum pname,
  GLint param
);
```

Parameters

pname

The symbolic name of the parameter to be set. Six of the twelve storage parameters affect how pixel data is returned to client memory, and are therefore significant only for **glReadPixels** commands. They are as follows:

GL_PACK_SWAP_BYTES

If true, byte ordering for multibyte color components, depth components, color indexes, or stencil indexes is reversed. That is, if a four-byte component is made up of bytes $b_{(0)}$, $b_{(1)}$, $b_{(2)}$, $b_{(3)}$, it is stored in memory as $b_{(3)}$, $b_{(2)}$, $b_{(1)}$, $b_{(0)}$ if GL_PACK_SWAP_BYTES is true. GL_PACK_SWAP_BYTES has no effect on the memory order of components within a pixel, only on the order of bytes within components or indexes. For example, the three components of a GL_RGB format pixel are always stored with red first, green second, and blue third, regardless of the value of GL_PACK_SWAP_BYTES.

GL_PACK_LSB_FIRST

If true, bits are ordered within a byte from least significant to most significant; otherwise, the first bit in each byte is the most significant one. This parameter is significant for bitmap data only.

GL_PACK_ROW_LENGTH

If greater than zero, GL_PACK_ROW_LENGTH defines the number of pixels in a row. If the first pixel of a row is placed at location *p* in memory, then the location of the first pixel of the next row is obtained by skipping

{ewc msdncd, EWGraphic, bsd23545 15 /a "SDK.BMP"}

components or indexes, where *n* is the number of components or indexes in a pixel, *l* is the number of pixels in a row (GL_PACK_ROW_LENGTH if it is greater than zero, the width argument to the pixel routine otherwise), *a* is the value of GL_PACK_ALIGNMENT, and *s* is the size, in bytes, of a single component (if $a < s$, then it is as if $a = s$). In the case of 1-bit values, the location of the next row is obtained by skipping

{ewc msdncd, EWGraphic, bsd23545 16 /a "SDK.BMP"}

components or indexes.

The word *component* in this description refers to the nonindex values red, green, blue, alpha, and depth. Storage format GL_RGB, for example, has three components per pixel: first red, then green, and finally blue.

GL_PACK_SKIP_PIXELS and GL_PACK_SKIP_ROWS

These values are provided as a convenience to the programmer; they provide no functionality that cannot be duplicated simply by incrementing the pointer passed to **glReadPixels**. Setting GL_PACK_SKIP_PIXELS to *i* is equivalent to incrementing the pointer by *i n* components or indexes, where *n* is the number of components or indexes in each pixel. Setting

GL_PACK_SKIP_ROWS to *j* is equivalent to incrementing the pointer by *j k* components or indexes, where *k* is the number of components or indexes per row, as computed above in the GL_PACK_ROW_LENGTH section.

GL_PACK_ALIGNMENT
Specifies the alignment requirements for the start of each pixel row in memory. The allowable values are 1 (byte-alignment), 2 (rows aligned to even-numbered bytes), 4 (word alignment), and 8 (rows start on double-word boundaries).

The other six of the twelve storage parameters affect how pixel data is read from client memory. These values are significant for **glDrawPixels**, **glTexImage1D**, **glTexImage2D**, **glBitmap**, and **glPolygonStipple**. They are as follows:

GL_UNPACK_SWAP_BYTES
If true, byte ordering for multibyte color components, depth components, color indexes, or stencil indexes is reversed. That is, if a four-byte component is made up of bytes $b_{(0)}$, $b_{(1)}$, $b_{(2)}$, $b_{(3)}$, it is taken from memory as $b_{(3)}$, $b_{(2)}$, $b_{(1)}$, $b_{(0)}$ if GL_UNPACK_SWAP_BYTES is true. GL_UNPACK_SWAP_BYTES has no effect on the memory order of components within a pixel, only on the order of bytes within components or indexes. For example, the three components of a GL_RGB format pixel are always stored with red first, green second, and blue third, regardless of the value of GL_UNPACK_SWAP_BYTES.

GL_UNPACK_LSB_FIRST
If true, bits are ordered within a byte from least significant to most significant; otherwise, the first bit in each byte is the most significant one. This is significant for bitmap data only.

GL_UNPACK_ROW_LENGTH
If greater than zero, GL_UNPACK_ROW_LENGTH defines the number of pixels in a row. If the first pixel of a row is placed at location *p* in memory, then the location of the first pixel of the next row is obtained by skipping

{ewc msdncd, EWGraphic, bsd23545 17 /a "SDK.BMP"}

components or indexes, where n is the number of components or indexes in a pixel, *l* is the number of pixels in a row (GL_UNPACK_ROW_LENGTH if it is greater than zero, the width argument to the pixel routine otherwise), *a* is the value of GL_UNPACK_ALIGNMENT, and *s* is the size, in bytes, of a single component (if $a < s$, then it is as if $a = s$). In the case of 1-bit values, the location of the next row is obtained by skipping

{ewc msdncd, EWGraphic, bsd23545 18 /a "SDK.BMP"}

components or indexes.

The word *component* in this description refers to the nonindex values red, green, blue, alpha, and depth. Storage format GL_RGB, for example, has three components per pixel: first red, then green, and finally blue.

GL_UNPACK_SKIP_PIXELS and GL_UNPACK_SKIP_ROWS
These values are provided as a convenience to the programmer; they provide no functionality that cannot be duplicated simply by incrementing the pointer passed to **glDrawPixels**, **glTexImage1D**, **glTexImage2D**, **glBitmap**, or **glPolygonStipple**. Setting GL_UNPACK_SKIP_PIXELS to *i* is equivalent to incrementing the pointer by *i n* components or indexes, where *n* is the number of components or indexes in each pixel. Setting GL_UNPACK_SKIP_ROWS to *j* is equivalent to incrementing the pointer by *j k* components or indexes, where *k* is the number of components or indexes per row, as computed above in the GL_UNPACK_ROW_LENGTH section.

GL_UNPACK_ALIGNMENT
Specifies the alignment requirements for the start of each pixel row in memory. The allowable values are 1 (byte-alignment), 2 (rows aligned to even-numbered bytes), 4 (word alignment), and 8 (rows start on double-word boundaries).

param
The value that *pname* is set to.

Remarks

The **glPixelStore** function sets pixel storage modes that affect the operation of subsequent **glDrawPixels** and **glReadPixels** as well as the unpacking of polygon stipple patterns (see **glPolygonStipple**), bitmaps (see **glBitmap**), and texture patterns (see **glTexImage1D** and **glTexImage2D**).

The following table gives the type, initial value, and range of valid values for each of the storage parameters that can be set with **glPixelStore**.

Pname	Type	Initial Value	Valid Range
GL_PACK_SWAP_BYTES	Boolean	false	true or false
GL_PACK_SWAP_BYTES	Boolean	false	true or false
GL_PACK_ROW_LENGTH	integer	0	[0,)
GL_PACK_SKIP_ROWS	integer	0	[0,)
GL_PACK_SKIP_PIXELS	integer	0	[0,)
GL_PACK_ALIGNMENT	integer	4	1, 2, 4, or 8
GL_UNPACK_SWAP_BYTES	Boolean	false	true or false
GL_UNPACK_LSB_FIRST	Boolean	false	true or false
GL_UNPACK_ROW_LENGTH	integer	0	[0,)
GL_UNPACK_SKIP_ROWS	integer	0	[0,)
GL_UNPACK_SKIP_PIXELS	integer	0	[0,)
GL_UNPACK_ALIGNMENT	integer	4	1, 2, 4, or 8

The **glPixelStoref** function can be used to set any pixel store parameter. If the parameter type is Boolean, and if *param* is 0.0, then the parameter is false; otherwise it is set to true. If *pname* is an integer type parameter, then *param* is rounded to the nearest integer.

Likewise, the **glPixelStorei** function can also be used to set any of the pixel store parameters. Boolean parameters are set to false if *param* is 0 and true otherwise. The *param* parameter is converted to floating point before being assigned to real-valued parameters.

The pixel storage modes in effect when **glDrawPixels**, **glReadPixels**, **glTexImage1D**, **glTexImage2D**, **glBitmap**, or **glPolygonStipple** is placed in a display list control the interpretation of memory data. The pixel storage modes in effect when a display list is executed are not significant.

The following functions retrieve information related to **glPixelStore**:

glGet with argument GL_PACK_SWAP_BYTES
glGet with argument GL_PACK_LSB_FIRST
glGet with argument GL_PACK_ROW_LENGTH
glGet with argument GL_PACK_SKIP_ROWS
glGet with argument GL_PACK_SKIP_PIXELS
glGet with argument GL_PACK_ALIGNMENT
glGet with argument GL_UNPACK_SWAP_BYTES
glGet with argument GL_UNPACK_LSB_FIRST
glGet with argument GL_UNPACK_ROW_LENGTH
glGet with argument GL_UNPACK_SKIP_ROWS
glGet with argument GL_UNPACK_SKIP_PIXELS
glGet with argument GL_UNPACK_ALIGNMENT

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*pname* was not an accepted value.
GL_INVALID_VALUE	A negative row length, pixel skip, or row skip value was specified, or if alignment was specified as other than 1, 2, 4, or 8.
GL_INVALID_OPERATION	**glPixelStore** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glBitmap**, **glDrawPixels**, **glEnd**, **glPixelMap**, **glPixelTransfer**, **glPixelZoom**, **glPolygonStipple**, **glReadPixels**, **glTexImage1D**, **glTexImage2D**

glPixelTransferf, glPixelTransferi

[New - Windows 95, OEM Service Release 2]

The **glPixelTransferf** and **glPixelTransferi** functions set pixel transfer modes.

```
void glPixelTransferf(
  GLenum pname,
  GLfloat param
);

void glPixelTransferi(
  GLenum pname,
  GLint param
);
```

Parameters

pname

The symbolic name of the pixel transfer parameter to be set. The following table gives the type, initial value, and range of valid values for each of the pixel transfer parameters that are set with **glPixelTransfer**.

Pname	Type	Initial Value	Valid Range
GL_MAP_COLOR	Boolean	false	true/false
GL_MAP_STENCIL	Boolean	false	true/false
GL_INDEX_SHIFT	integer	0	(- ,)
GL_INDEX_OFFSET	integer	0	(- ,)
GL_RED_SCALE	float	1.0	(- ,)
GL_GREEN_SCALE	float	1.0	(- ,)
GL_BLUE_SCALE	float	1.0	(- ,)
GL_ALPHA_SCALE	float	1.0	(- ,)
GL_DEPTH_SCALE	float	1.0	(- ,)
GL_RED_BIAS	float	0.0	(- ,)
GL_GREEN_BIAS	float	0.0	(- ,)
GL_BLUE_BIAS	float	0.0	(- ,)
GL_ALPHA_BIAS	float	0.0	(- ,)
GL_DEPTH_BIAS	float	0.0	(- ,)

param

The value that *pname* is set to.

Remarks

The **glPixelTransfer** function sets pixel transfer modes that affect the operation of subsequent **glDrawPixels**, **glReadPixels**, **glCopyPixels**, **glTexImage1D**, and **glTexImage2D** commands. The algorithms that are specified by pixel transfer modes operate on pixels after they are read from the frame buffer (**glReadPixels** and **glCopyPixels**) or unpacked from client memory (**glDrawPixels**, **glTexImage1D**, and **glTexImage2D**). Pixel transfer operations happen in the same order, and in the same manner, regardless of the command that resulted in the pixel operation. Pixel storage modes (**glPixelStore**) control the unpacking of pixels being read from client memory, and the packing of pixels being written back into client memory.

Pixel transfer operations handle four fundamental pixel types: *color, color index, depth*, and *stencil*. *Color* pixels are made up of four floating-point values with unspecified mantissa and exponent sizes, scaled

such that 0.0 represents zero intensity and 1.0 represents full intensity. *Color indexes* comprise a single fixed-point value, with unspecified precision to the right of the binary point. *Depth* pixels comprise a single floating-point value, with unspecified mantissa and exponent sizes, scaled such that 0.0 represents the minimum depth buffer value, and 1.0 represents the maximum depth buffer value. Finally, *stencil* pixels comprise a single fixed-point value, with unspecified precision to the right of the binary point.

The pixel transfer operations performed on the four basic pixel types are as follows:

Color

Each of the four color components is multiplied by a scale factor, and then added to a bias factor. That is, the red component is multiplied by GL_RED_SCALE, and then added to GL_RED_BIAS; the green component is multiplied by GL_GREEN_SCALE, and then added to GL_GREEN_BIAS; the blue component is multiplied by GL_BLUE_SCALE, and then added to GL_BLUE_BIAS; and the alpha component is multiplied by GL_ALPHA_SCALE, and then added to GL_ALPHA_BIAS. After all four color components are scaled and biased, each is clamped to the range [0,1]. All color scale and bias values are specified with **glPixelTransfer**.

If GL_MAP_COLOR is true, each color component is scaled by the size of the corresponding color-to-color map, and then replaced by the contents of that map indexed by the scaled component. That is, the red component is scaled by GL_PIXEL_MAP_R_TO_R_SIZE, and then replaced by the contents of GL_PIXEL_MAP_R_TO_R indexed by itself. The green component is scaled by GL_PIXEL_MAP_G_TO_G_SIZE, and then replaced by the contents of GL_PIXEL_MAP_G_TO_G indexed by itself. The blue component is scaled by GL_PIXEL_MAP_B_TO_B_SIZE, and then replaced by the contents of GL_PIXEL_MAP_B_TO_B indexed by itself. The alpha component is scaled by GL_PIXEL_MAP_A_TO_A_SIZE, and then replaced by the contents of GL_PIXEL_MAP_A_TO_A indexed by itself. All components taken from the maps are then clamped to the range [0,1]. GL_MAP_COLOR is specified with **glPixelTransfer**. The contents of the various maps are specified with **glPixelMap**.

Color index

Each color index is shifted left by GL_INDEX_SHIFT bits, filling with zeros any bits beyond the number of fraction bits carried by the fixed-point index. If GL_INDEX_SHIFT is negative, the shift is to the right, again zero filled. GL_INDEX_OFFSET is then added to the index. GL_INDEX_SHIFT and GL_INDEX_OFFSET are specified with **glPixelTransfer**.

From this point, operation diverges depending on the required format of the resulting pixels. If the resulting pixels are to be written to a color-index buffer, or if they are being read back to client memory in GL_COLOR_INDEX format, the pixels continue to be treated as indexes. If GL_MAP_COLOR is true, then each index is masked by 2^*n* - 1, where *n* is GL_PIXEL_MAP_I_TO_I_SIZE, and then replaced by the contents of GL_PIXEL_MAP_I_TO_I indexed by the masked value. GL_MAP_COLOR is specified with **glPixelTransfer**. The contents of the index map are specified with **glPixelMap**.

If the resulting pixels are to be written to an RGBA color buffer, or if they are being read back to client memory in a format other than GL_COLOR_INDEX, the pixels are converted from indexes to colors by referencing the four maps GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, and GL_PIXEL_MAP_I_TO_A. Before being dereferenced, the index is masked by 2_n - 1, where *n* is GL_PIXEL_MAP_I_TO_R_SIZE for the red map, GL_PIXEL_MAP_I_TO_G_SIZE for the green map, GL_PIXEL_MAP_I_TO_B_SIZE for the blue map, and GL_PIXEL_MAP_I_TO_A_SIZE for the alpha map. All components taken from the maps are then clamped to the range [0,1]. The contents of the four maps are specified with **glPixelMap**.

Depth

Each depth value is multiplied by GL_DEPTH_SCALE, added to GL_DEPTH_BIAS, and then clamped to the range [0,1].

Stencil

Each index is shifted GL_INDEX_SHIFT bits just as a color index is, and then added to GL_INDEX_OFFSET. If GL_MAP_STENCIL is true, each index is masked by 2^*n* - 1, where *n* is GL_PIXEL_MAP_S_TO_S_SIZE, then replaced by the contents of GL_PIXEL_MAP_S_TO_S indexed by the masked value.

The **glPixelTransferf** function can be used to set any pixel transfer parameter. If the parameter type is Boolean, 0.0 implies false and any other value implies true. If *pname* is an integer parameter, *param* is rounded to the nearest integer.

Likewise, **glPixelTransferi** can also be used to set any of the pixel transfer parameters. Boolean parameters are set to false if *param* is 0 and true otherwise. The *param* parameter is converted to floating point before being assigned to real-valued parameters.

If a **glDrawPixels**, **glReadPixels**, **glCopyPixels**, **glTexImage1D**, or **glTexImage2D** command is placed in a display list (see **glNewList** and **glCallList**), the pixel transfer mode settings in effect when the display list is *executed* are the ones that are used. They may be different from the settings when the command was compiled into the display list.

The following functions retrieve information related to **glPixelTransfer**:

glGet with argument GL_MAP_COLOR
glGet with argument GL_MAP_STENCIL
glGet with argument GL_INDEX_SHIFT
glGet with argument GL_INDEX_OFFSET
glGet with argument GL_RED_SCALE
glGet with argument GL_RED_BIAS
glGet with argument GL_GREEN_SCALE
glGet with argument GL_GREEN_BIAS
glGet with argument GL_BLUE_SCALE
glGet with argument GL_BLUE_BIAS
glGet with argument GL_ALPHA_SCALE
glGet with argument GL_ALPHA_BIAS
glGet with argument GL_DEPTH_SCALE
glGet with argument GL_DEPTH_BIAS

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*pname* was not an accepted value.
GL_INVALID_OPERATION	**glPixelTransfer** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glCallList**, **glCopyPixels**, **glDrawPixels**, **glEnd**, **glNewList**, **glPixelMap**, **glPixelStore**, **glPixelZoom**, **glReadPixels**, **glTexImage1D**, **glTexImage2D**

glPixelZoom Quick Info

[New - Windows 95, OEM Service Release 2]

The **glPixelZoom** function specifies the pixel zoom factors.

```
void glPixelZoom(
  GLfloat xfactor,
  GLfloat yfactor
);
```

Parameters

xfactor, yfactor

The *x* and *y* zoom factors for pixel write operations.

Remarks

The **glPixelZoom** function specifies values for the *x* and *y* zoom factors. During the execution of **glDrawPixels** or **glCopyPixels**, if ($x_{(r)}$, $y_{(r)}$) is the current raster position, and a given element is in the *n*th row and *m*th column of the pixel rectangle, then pixels whose centers are in the rectangle with corners at

{ewc msdncd, EWGraphic, bsd23545 19 /a "SDK.BMP"}

are candidates for replacement. Any pixel whose center lies on the bottom or left edge of this rectangular region is also modified.

Pixel zoom factors are not limited to positive values. Negative zoom factors reflect the resulting image about the current raster position.

The following functions retrieve information related to **glPixelZoom**:

glGet with argument GL_ZOOM_X
glGet with argument GL_ZOOM_Y

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glPixelZoom** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glCopyPixels**, **glDrawPixels**, **glEnd**

glPointSize Quick Info

[New - Windows 95, OEM Service Release 2]

The **glPointSize** function specifies the diameter of rasterized points.

```
void glPointSize(
  GLfloat size
);
```

Parameters

size

The diameter of rasterized points. The default is 1.0.

Remarks

The **glPointSize** function specifies the rasterized diameter of both aliased and antialiased points. Using a point size other than 1.0 has different effects, depending on whether point antialiasing is enabled. Point antialiasing is controlled by calling **glEnable** and **glDisable** with argument GL_POINT_SMOOTH.

If point antialiasing is disabled, the actual size is determined by rounding the supplied size to the nearest integer. (If the rounding results in the value 0, it is as if the point size were 1.) If the rounded size is odd, then the center point (*x, y*) of the pixel fragment that represents the point is computed as

$$(\lfloor x_{(w)} \rfloor + .5, \lfloor y_{(w)} \rfloor + .5)$$

where *w* subscripts indicate window coordinates. All pixels that lie within the square grid of the rounded size centered at (*x, y*) make up the fragment. If the size is even, the center point is

$$(\lfloor x_{(w)} + .5 \rfloor, \lfloor y_{(w)} + .5 \rfloor)$$

and the rasterized fragment's centers are the half-integer window coordinates within the square of the rounded size centered at (*x, y*). All pixel fragments produced in rasterizing a nonantialiased point are assigned the same associated data; that of the vertex corresponding to the point.

If antialiasing is enabled, then point rasterization produces a fragment for each pixel square that intersects the region lying within the circle having diameter equal to the current point size and centered at the points ($x_{(w)}$, $y_{(w)}$). The coverage value for each fragment is the window coordinate area of the intersection of the circular region with the corresponding pixel square. This value is saved and used in the final rasterization step. The data associated with each fragment is the data associated with the point being rasterized.

Not all sizes are supported when point antialiasing is enabled. If an unsupported size is requested, the nearest supported size is used. Only size 1.0 is guaranteed to be supported; others depend on the implementation. The range of supported sizes and the size difference between supported sizes within the range can be queried by calling **glGet** with arguments GL_POINT_SIZE_RANGE and GL_POINT_SIZE_GRANULARITY.

The point size specified by **glPointSize** is always returned when GL_POINT_SIZE is queried. Clamping and rounding for aliased and antialiased points have no effect on the specified value.

Non-antialiased point size may be clamped to an implementation-dependent maximum. Although this maximum cannot be queried, it must be no less than the maximum value for antialiased points, rounded to the nearest integer value.

The following functions retrieve information related to **glPointSize**:

glGet with argument GL_POINT_SIZE
glGet with argument GL_POINT_SIZE_RANGE
glGet with argument GL_POINT_SIZE_GRANULARITY
glIsEnabled with argument GL_POINT_SMOOTH

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_VALUE	*size* was less than or equal to zero.
GL_INVALID_OPERATION	**glPointSize** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnable**, **glEnd**, **glIsEnabled**

glPolygonMode Quick Info

[New - Windows 95, OEM Service Release 2]

The **glPolygonMode** function selects a polygon rasterization mode.

```
void glPolygonMode(
  GLenum face,
  GLenum mode
);
```

Parameters

face

The polygons that *mode* applies to. Must be GL_FRONT for front-facing polygons, GL_BACK for back-facing polygons, or GL_FRONT_AND_BACK for front- and back-facing polygons.

mode

The way polygons will be rasterized. The following modes are defined and can be specified in *mode*. The default is GL_FILL for both front- and back-facing polygons.

GL_POINT

Polygon vertices that are marked as the start of a boundary edge are drawn as points. Point attributes such as GL_POINT_SIZE and GL_POINT_SMOOTH control the rasterization of the points. Polygon rasterization attributes other than GL_POLYGON_MODE have no effect.

GL_LINE

Boundary edges of the polygon are drawn as line segments. They are treated as connected line segments for line stippling; the line stipple counter and pattern are not reset between segments (see **glLineStipple**). Line attributes such as GL_LINE_WIDTH and GL_LINE_SMOOTH control the rasterization of the lines. Polygon rasterization attributes other than GL_POLYGON_MODE have no effect.

GL_FILL

The interior of the polygon is filled. Polygon attributes such as GL_POLYGON_STIPPLE and GL_POLYGON_SMOOTH control the rasterization of the polygon.

Remarks

The **glPolygonMode** function controls the interpretation of polygons for rasterization. The *face* parameter describes which polygons *mode* applies to: front-facing polygons (GL_FRONT), back-facing polygons (GL_BACK), or both (GL_FRONT_AND_BACK). The polygon mode affects only the final rasterization of polygons. In particular, a polygon's vertices are lit and the polygon is clipped and possibly culled before these modes are applied.

To draw a surface with filled back-facing polygons and outlined front-facing polygons, call

glPolygonMode(GL_FRONT, GL_LINE**)**;

Vertices are marked as boundary or nonboundary with an edge flag. Edge flags are generated internally by OpenGL when it decomposes polygons, and they can be set explicitly using **glEdgeFlag**.

The following function retrieves information related to **glPolygonMode**:

glGet with argument GL_POLYGON_MODE

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition

GL_INVALID_ENUM	Either *face* or *mode* was not an accepted value.
GL_INVALID_OPERATION	**glPolygonMode** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEdgeFlag**, **glEnd**, **glLineStipple**, **glLineWidth**, **glPointSize**, **glPolygonStipple**

glPolygonStipple Quick Info

[New - Windows 95, OEM Service Release 2]

The **glPolygonStipple** function sets the polygon stippling pattern.

```
void glPolygonStipple(
  const GLubyte *mask
);
```

Parameters

mask

A pointer to a 32x32 stipple pattern that will be unpacked from memory in the same way that **glDrawPixels** unpacks pixels.

Remarks

The **glPolygonStipple** function sets the polygon stippling pattern. Polygon stippling, like line stippling (see **glLineStipple**), masks out certain fragments produced by rasterization, creating a pattern. Stippling is independent of polygon antialiasing.

The *mask* parameter is a pointer to a 32x32 stipple pattern that is stored in memory just like the pixel data supplied to **glDrawPixels** with *height* and *width* both equal to 32, a pixel *format* of GL_COLOR_INDEX, and data *type* of GL_BITMAP. That is, the stipple pattern is represented as a 32x32 array of 1-bit color indexes packed in unsigned bytes. The **glPixelStore** function parameters, such as GL_UNPACK_SWAP_BYTES and GL_UNPACK_LSB_FIRST, affect the assembling of the bits into a stipple pattern. Pixel transfer operations (shift, offset, and pixel map) are not applied to the stipple image, however.

Polygon stippling is enabled and disabled with **glEnable** and **glDisable**, using argument GL_POLYGON_STIPPLE. If enabled, a rasterized polygon fragment with window coordinates $x_{(w)}$ and $y_{(w)}$ is sent to the next stage of OpenGL if and only if the ($x_{(w)}$ mod 32)th bit in the ($y_{(w)}$ mod 32)th row of the stipple pattern is one. When polygon stippling is disabled, it is as if the stipple pattern were all ones.

The following functions retrieve information related to **glPolygonStipple**:

glGetPolygonStipple

glIsEnabled with argument GL_POLYGON_STIPPLE

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glPolygonStipple** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glDrawPixels**, **glEnd**, **glLineStipple**, **glPixelStore**, **glPixelTransfer**

glPrioritizeTextures

[New - Windows 95, OEM Service Release 2]

The **glPrioritizeTextures** function sets the residence priority of textures.

```
void glPrioritizeTextures(
  GLsizei n,
  GLuint * textures,
  GLclampf * priorities
);
```

Parameters

n
The number of textures to be prioritized.

textures
A pointer to the first element of an array containing the names of the textures to be prioritized.

priorities
A pointer to the first element of an array containing the texture priorities. A priority given in an element of the *priorities* parameter applies to the texture named by the corresponding element of the *textures* parameter.

Remarks

The **glPrioritizeTextures** function assigns the *n* texture priorities specified in the *priorities* parameter to the *n* textures named in the *textures* parameter. On machines with a limited amount of texture memory, OpenGL establishes a ``working set'' of textures that are resident in texture memory. These textures can be bound to a texture target much more efficiently than textures that are not resident.

By specifying a priority for each texture, the **glPrioritizeTextures** function enables you to determine which textures should be resident.

The texture priorities elements in *priorities* are clamped to the range [0.0, 1.0] before being assigned. Zero indicates the lowest priority; thus textures with priority zero are least likely to be resident. The value 1.0 indicates the highest priority; thus textures with priority 1.0 are most likely to be resident. However, textures are not guaranteed to be resident until they are bound.

The **glPrioritizeTextures** function ignores attempts to prioritize textures with a *priorities* value of zero or any texture name that does not correspond to an existing texture. None of the functions named by the *textures* parameter need to be bound to a texture target.

If a texture is currently bound, you can also use the **glTexParameter** function to set its priority. This is the only way to set the priority of a default texture.

You can include **glPrioritizeTextures** in display lists.

Note The **glPrioritizeTextures** function is only available in OpenGL version 1.1 or later.

The following function retrieves the priority of a currently-bound texture related to **glPrioritizeTextures**:

glGetTexParameter with parameter name GL_TEXTURE_PRIORITY.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_VALUE	*n* was a negative value.
GL_INVALID_OPERATION	**glPrioritizeTextures** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glAreTexturesResident, **glBegin**, **glEnd**, **glGetTexParameter**, **glTexImage1D**, **glTexImage2D**, **glTexParameter**

glPushAttrib, glPopAttrib

[New - Windows 95, OEM Service Release 2]

The **glPushAttrib** and **glPopAttrib** functions push and pop the attribute stack.

```
void glPushAttrib(
  GLbitfield mask
);
```

Parameter

mask

A mask that indicates which attributes to save. The symbolic mask constants and their associated OpenGL state are as follows (the indented paragraphs list which attributes are saved):

GL_ACCUM_BUFFER_BIT
- Accumulation buffer clear value

GL_COLOR_BUFFER_BIT
- GL_ALPHA_TEST enable bit
- Alpha test function and reference value
- GL_BLEND enable bit
- Blending source and destination functions
- GL_DITHER enable bit
- GL_DRAW_BUFFER setting
- GL_LOGIC_OP enable bit
- Logic op function
- Color-mode and index-mode clear values
- Color-mode and index-mode writemasks

GL_CURRENT_BIT
- Current RGBA color
- Current color index
- Current normal vector
- Current texture coordinates
- Current raster position
- GL_CURRENT_RASTER_POSITION_VALID flag
- RGBA color associated with current raster position
- Color index associated with current raster position
- Texture coordinates associated with current raster position
- GL_EDGE_FLAG flag

GL_DEPTH_BUFFER_BIT
- GL_DEPTH_TEST enable bit
- Depth buffer test function
- Depth buffer clear value
- GL_DEPTH_WRITEMASK enable bit

GL_ENABLE_BIT
- GL_ALPHA_TEST flag
- GL_AUTO_NORMAL flag
- GL_BLEND flag
- Enable bits for the user-definable clipping planes
- GL_COLOR_MATERIAL
- GL_CULL_FACE flag
- GL_DEPTH_TEST flag
- GL_DITHER flag
- GL_FOG flag
- GL_LIGHT*i* where 0 <= *i* < GL_MAX_LIGHTS

- GL_LIGHTING flag
- GL_LINE_SMOOTH flag
- GL_LINE_STIPPLE flag
- GL_LOGIC_OP flag
- GL_MAP1_*x* where *x* is a map type
- GL_MAP2_*x* where *x* is a map type
- GL_NORMALIZE flag
- GL_POINT_SMOOTH flag
- GL_POLYGON_SMOOTH flag
- GL_POLYGON_STIPPLE flag
- GL_SCISSOR_TEST flag
- GL_STENCIL_TEST flag
- GL_TEXTURE_1D flag
- GL_TEXTURE_2D flag
- Flags GL_TEXTURE_GEN_*x* where *x* is S, T, R, or Q

GL_EVAL_BIT
- GL_MAP1_*x* enable bits, where *x* is a map type
- GL_MAP2_*x* enable bits, where *x* is a map type
- 1-D grid endpoints and divisions
- 2-D grid endpoints and divisions
- GL_AUTO_NORMAL enable bit

GL_FOG_BIT
- GL_FOG enable flag
- Fog color
- Fog density
- Linear fog start
- Linear fog end
- Fog index
- GL_FOG_MODE value

GL_HINT_BIT
- GL_PERSPECTIVE_CORRECTION_HINT setting
- GL_POINT_SMOOTH_HINT setting
- GL_LINE_SMOOTH_HINT setting
- GL_POLYGON_SMOOTH_HINT setting
- GL_FOG_HINT setting

GL_LIGHTING_BIT
- GL_COLOR_MATERIAL enable bit
- GL_COLOR_MATERIAL_FACE value
- Color material parameters that are tracking the current color
- Ambient scene color
- GL_LIGHT_MODEL_LOCAL_VIEWER value
- GL_LIGHT_MODEL_TWO_SIDE setting
- GL_LIGHTING enable bit
- Enable bit for each light
- Ambient, diffuse, and specular intensity for each light
- Direction, position, exponent, and cutoff angle for each light
- Constant, linear, and quadratic attenuation factors for each light
- Ambient, diffuse, specular, and emissive color for each material
- Ambient, diffuse, and specular color indexes for each material
- Specular exponent for each material
- GL_SHADE_MODEL setting

GL_LINE_BIT
- GL_LINE_SMOOTH flag
- GL_LINE_STIPPLE enable bit
- Line stipple pattern and repeat counter

- Line width

GL_LIST_BIT
- GL_LIST_BASE setting

GL_PIXEL_MODE_BIT
- GL_RED_BIAS and GL_RED_SCALE settings
- GL_GREEN_BIAS and GL_GREEN_SCALE values
- GL_BLUE_BIAS and GL_BLUE_SCALE
- GL_ALPHA_BIAS and GL_ALPHA_SCALE
- GL_DEPTH_BIAS and GL_DEPTH_SCALE
- GL_INDEX_OFFSET and GL_INDEX_SHIFT values
- GL_MAP_COLOR and GL_MAP_STENCIL flags
- GL_ZOOM_X and GL_ZOOM_Y factors
- GL_READ_BUFFER setting

GL_POINT_BIT
- GL_POINT_SMOOTH flag
- Point size

GL_POLYGON_BIT
- GL_CULL_FACE enable bit
- GL_CULL_FACE_MODE value
- GL_FRONT_FACE indicator
- GL_POLYGON_MODE setting
- GL_POLYGON_SMOOTH flag
- GL_POLYGON_STIPPLE enable bit

GL_POLYGON_STIPPLE_BIT
- Polygon stipple image

GL_SCISSOR_BIT
- GL_SCISSOR_TEST flag
- Scissor box

GL_STENCIL_BUFFER_BIT
- GL_STENCIL_TEST enable bit
- Stencil function and reference value
- Stencil value mask
- Stencil fail, pass, and depth buffer pass actions
- Stencil buffer clear value
- Stencil buffer writemask

GL_TEXTURE_BIT
- Enable bits for the four texture coordinates
- Border color for each texture image
- Minification function for each texture image
- Magnification function for each texture image
- Texture coordinates and wrap mode for each texture image
- Color and mode for each texture environment
- Enable bits GL_TEXTURE_GEN_*x*; *x* is S, T, R, and Q
- GL_TEXTURE_GEN_MODE setting for S, T, R, and Q
- **glTexGen** plane equations for S, T, R, and Q

GL_TRANSFORM_BIT
- Coefficients of the six clipping planes
- Enable bits for the user-definable clipping planes
- GL_MATRIX_MODE value
- GL_NORMALIZE flag

GL_VIEWPORT_BIT
- Depth range (near and far)
- Viewport origin and extent

```
void glPopAttrib(
  void
);
```

Remarks

The **glPushAttrib** function takes one argument, a mask that indicates which groups of state variables to save on the attribute stack. Symbolic constants are used to set bits in the mask. The *mask* parameter is typically constructed by **OR**ing several of these constants together. The special mask GL_ALL_ATTRIB_BITS can be used to save all stackable states.

The **glPopAttrib** function restores the values of the state variables saved with the last **glPushAttrib** command. Those not saved are left unchanged.

It is an error to push attributes onto a full stack, or to pop attributes off an empty stack. In either case, the error flag is set and no other change is made to the OpenGL state.

Initially, the attribute stack is empty.

Not all values for the OpenGL state can be saved on the attribute stack. For example, pixel pack and unpack state, render mode state, and select and feedback state cannot be saved.

The depth of the attribute stack depends on the implementation, but it must be at least 16.

The following functions retrieve information related to **glPushAttrib** and **glPopAttrib**:

glGet with argument GL_ATTRIB_STACK_DEPTH
glGet with argument GL_MAX_ATTRIB_STACK_DEPTH

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_STACK_OVERFLOW	**glPushAttrib** was called while the attribute stack was full.
GL_STACK_UNDERFLOW	**glPopAttrib** was called while the attribute stack was empty.
GL_INVALID_OPERATION	**glPushAttrib** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glGet**, **glGetClipPlane**, **glGetError**, **glGetLight**, **glGetMap**, **glGetMaterial**, **glGetPixelMap**, **glGetPolygonStipple**, **glGetString**, **glGetTexEnv**, **glGetTexGen**, **glGetTexImage**, **glGetTexLevelParameter**, **glGetTexParameter**, **glIsEnabled**

glPushClientAttrib, glPopClientAttrib

[New - Windows 95, OEM Service Release 2]

The **glPushClientAttrib** and **glPopClientAttrib** functions save and restore groups of client-state variables on the client-attribute stack.

void glPushClientAttrib(
  GLbitfield *mask*
);

Parameters

mask
A mask that indicates which attributes to save. The following are the symbolic mask constants and their associated OpenGL client state:

GL_CLIENT_PIXEL_STORE_BIT
Pixel storage mode attributes.

GL_CLIENT_VERTEX_ARRAY_BIT
Vertex array attributes.

GL_CLIENT_ALL_ATTRIB_BITS
All stackable client-state attributes.

void glPopClientAttrib(
  void *void*
);

Remarks

The **glPushClientAttrib** function uses its *mask* parameter to determine which groups of client-state variables are saved on the client-attribute stack. You can **OR** together accepted symbolic constants to set bits and construct a *mask*.

The **glPopClientAttrib** function restores the values of the client-state variables last saved with **glPushClientAttrib**. Client-state variables not previously saved are left unchanged. Pushing attributes onto a full client-attribute stack or popping attributes off an empty stack sets an error flag and no other change is made to the OpenGL state. By default the client attribute stack is empty.

Some OpenGL client-state values cannot be saved on the client-attribute stack. For example, you cannot save the select or feedback states on the client-attribute stack. The depth of the client-attribute stack is at least 16.

The **glPushClientAttrib** and **glPopClientAttrib** functions are not compiled into display lists, but are executed immediately.

The **glPushClientAttrib** and **glPopClientAttrib** functions can only push and pop pixel storage modes and vertex array client states. You must use **glPushAttrib** and **glPopAttrib** to push and pop states that are kept on the server.

Note The **glPushClientAttrib** and **glPopClientAttrib** functions are only available in OpenGL version 1.1 or later.

The following functions retrieve information related to **glPushClientAttrib** and **glPopClientAttrib**:

glGet with argument GL_CLIENT_ATTRIB_STACK_DEPTH
glGet with argument GL_MAX_CLIENT_ATTRIB_STACK_DEPTH

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_STACK_OVERFLOW	**glPushClientAttrib** was called while the client-attribute stack was full.
GL_STACK_UNDERFLOW	**glPopClientAttrib** was called while the client-attribute stack was empty.

See Also

glColorPointer, **glDisableClientState**, **glEdgeFlagPointer**, **glEnableClientState**, **glGet**, **glGetError**, **glIndexPointer**, **glNormalPointer**, **glNewList**, **glPixelStore**, **glPushAttrib**, **glTexCoordPointer**, **glVertexPointer**

glPushMatrix, glPopMatrix

[New - Windows 95, OEM Service Release 2]

The **glPushMatrix** and **glPopMatrix** functions push and pop the current matrix stack.

```
void glPushMatrix(
  void
);

void glPopMatrix(
  void
);
```

Remarks

There is a stack of matrices for each of the matrix modes. In GL_MODELVIEW mode, the stack depth is at least 32. In the other two modes, GL_PROJECTION and GL_TEXTURE, the depth is at least 2. The current matrix in any mode is the matrix on the top of the stack for that mode.

The **glPushMatrix** function pushes the current matrix stack down by one, duplicating the current matrix. That is, after a **glPushMatrix** call, the matrix on the top of the stack is identical to the one below it. The **glPushMatrix** function pops the current matrix stack, replacing the current matrix with the one below it on the stack. Initially, each of the stacks contains one matrix, an identity matrix.

The following functions retrieve information related to **glPushMatrix** and **glPopMatrix**:

glGet with argument GL_MATRIX_MODE
glGet with argument GL_MODELVIEW_MATRIX
glGet with argument GL_PROJECTION_MATRIX
glGet with argument GL_TEXTURE_MATRIX
glGet with argument GL_MODELVIEW_STACK_DEPTH
glGet with argument GL_PROJECTION_STACK_DEPTH
glGet with argument GL_TEXTURE_STACK_DEPTH
glGet with argument GL_MAX_MODELVIEW_STACK_DEPTH
glGet with argument GL_MAX_PROJECTION_STACK_DEPTH
glGet with argument GL_MAX_TEXTURE_STACK_DEPTH

Error Codes

It is an error to push a full matrix stack, or to pop a matrix stack that contains only a single matrix. In either case, the error flag is set and no other change is made to the OpenGL state.

The following are the error codes generated and their conditions.

Error Code	Condition
GL_STACK_OVERFLOW	**glPushMatrix** was called while the current matrix stack was full.
GL_STACK_UNDERFLOW	**glPopMatrix** was called while the current matrix stack contained only a single matrix.
GL_INVALID_OPERATION	**glPushMatrix** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, glEnd, glFrustum, glLoadIdentity, glLoadMatrix, glMatrixMode, glMultMatrix, glOrtho, glRotate, glScale, glTranslate, glViewport

glPushName, glPopName

[New - Windows 95, OEM Service Release 2]

The **glPushName** and **glPopName** functions push and pop the name stack.

```
void glPushName(
  GLuint name
);
```

Parameters

name
A name that will be pushed onto the name stack.

```
void glPopName(
  void
);
```

Remarks

The **glPushName** function causes *name* to be pushed onto the name stack, which is initially empty. The **glPopName** function pops one name off the top of the stack. The name stack is used during selection mode to allow sets of rendering commands to be uniquely identified. It consists of an ordered set of unsigned integers.

The name stack is always empty while the render mode is not GL_SELECT. Calls to **glPushName** or **glPopName** while the render mode is not GL_SELECT are ignored.

The following functions retrieve information related to **glPushName** and **glPopName**:

glGet with argument GL_NAME_STACK_DEPTH
glGet with argument GL_MAX_NAME_STACK_DEPTH

Error Codes

It is an error to push a name onto a full stack, or to pop a name off an empty stack. It is also an error to manipulate the name stack between a call to **glBegin** and the corresponding call to **glEnd**. In any of these cases, the error flag is set and no other change is made to the OpenGL state.

The following are the error codes generated and their conditions.

Error Code	Condition
GL_STACK_OVERFLOW	**glPushName** was called while the name stack was full.
GL_STACK_UNDERFLOW	**glPopName** was called while the name stack was empty.
GL_INVALID_OPERATION	**glPushName** or **glPopName** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glInitNames**, **glLoadName**, **glRenderMode**, **glSelectBuffer**

glRasterPos

[New - Windows 95, OEM Service Release 2]

glRasterPos2d, glRasterPos2f, glRasterPos2i, glRasterPos2s, glRasterPos3d, glRasterPos3f, glRasterPos3i, glRasterPos3s, glRasterPos4d, glRasterPos4f, glRasterPos4i, glRasterPos4s, glRasterPos2dv, glRasterPos2fv, glRasterPos2iv, glRasterPos2sv, glRastePos3dv, glRasterPos3fv, glRasterPos3iv, glRasterPos3sv, glRasterPos4dv, glRasterPos4fv, glRasterPos4iv, glRasterPos4sv

These functions specify the raster position for pixel operations.

```
void glRasterPos2d(
  GLdouble x,
  GLdouble y
 );

void glRasterPos2f(
  GLfloat x,
  GLfloat y
 );

void glRasterPos2i(
  GLint x,
  GLint y
 );

void glRasterPos2s(
  GLshort x,
  GLshort y
 );

void glRasterPos3d(
  GLdouble x,
  GLdouble y,
  GLdouble z
 );

void glRasterPos3f(
  GLfloat x,
  GLfloat y,
  GLfloat z
 );

void glRasterPos3i(
  GLint x,
  GLint y,
  GLint z
 );

void glRasterPos3s(
  GLshort x,
  GLshort y,
  GLshort z
 );

void glRasterPos4d(
  GLdouble x,
```

```
  GLdouble y,
  GLdouble z,
  GLdouble w
);

void glRasterPos4f(
  GLfloat x,
  GLfloat y,
  GLfloat z,
  GLfloat w
);

void glRasterPos4i(
  GLint x,
  GLint y,
  GLint z,
  GLint w
);

void glRasterPos4s(
  GLshort x,
  GLshort y,
  GLshort z,
  GLshort w
);
```

Parameters

x, y, z, w
The *x*, *y*, *z*, and *w* object coordinates (if present) for the raster position.

```
void glRasterPos2dv(
  const GLdouble *v
);

void glRasterPos2fv(
  const GLfloat *v
);

void glRasterPos2iv(
  const GLint *v
);

void glRasterPos2sv(
  const GLshort *v
);

void glRasterPos3dv(
  const GLdouble *v
);

void glRasterPos3fv(
  const GLfloat *v
);

void glRasterPos3iv(
  const GLint *v
);

void glRasterPos3sv(
```

```
  const GLshort *v
);

void glRasterPos4dv(
  const GLdouble *v
);

void glRasterPos4fv(
  const GLfloat *v
);

void glRasterPos4iv(
  const GLint *v
);

void glRasterPos4sv(
  const GLshort *v
);
```

Parameters

v

A pointer to an array of two, three, or four elements, specifying *x, y, z,* and *w* coordinates, respectively.

Remarks

OpenGL maintains a 3-D position in window coordinates. This position, called the raster position, is maintained with subpixel accuracy. It is used to position pixel and bitmap write operations. See **glBitmap**, **glDrawPixels**, and **glCopyPixels**.

The current raster position consists of three window coordinates (*x, y, z*), a clip coordinate *w* value, an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The *w* coordinate is a clip coordinate, because *w* is not projected to window coordinates. The **glRasterPos4** function specifies object coordinates *x, y, z,* and *w* explicitly. The **glRasterPos3** function specifies object coordinates *x, y,* and *z* explicitly, while *w* is implicitly set to one. The **glRasterPos2** function uses the argument values for *x* and *y* while implicitly setting *z* and *w* to zero and one.

The object coordinates presented by **glRasterPos** are treated just like those of a **glVertex** command. They are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, then it is projected and scaled to window coordinates, which become the new current raster position, and the GL_CURRENT_RASTER_POSITION_VALID flag is set. If the vertex is culled, then the valid bit is cleared and the current raster position and associated color and texture coordinates are undefined.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then GL_CURRENT_RASTER_COLOR, in RGBA mode, or the GL_CURRENT_RASTER_INDEX, in color-index mode, is set to the color produced by the lighting calculation (see **glLight**, **glLightModel**, and **glShadeModel**). If lighting is disabled, current color (in RGBA mode, state variable GL_CURRENT_COLOR) or color index (in color-index mode, state variable GL_CURRENT_INDEX) is used to update the current raster color.

Likewise, GL_CURRENT_RASTER_TEXTURE_COORDS is updated as a function of GL_CURRENT_TEXTURE_COORDS, based on the texture matrix and the texture generation functions (see **glTexGen**). Finally, the distance from the origin of the eye coordinate system to the vertex, as transformed by only the modelview matrix, replaces GL_CURRENT_RASTER_DISTANCE.

Initially, the current raster position is (0,0,0,1), the current raster distance is 0, the valid bit is set, the associated RGBA color is (1,1,1,1), the associated color index is 1, and the associated texture

coordinates are (0, 0, 0, 1). In RGBA mode, GL_CURRENT_RASTER_INDEX is always 1; in color-index mode, the current raster RGBA color always maintains its initial value.

Note The raster position is modified both by **glRasterPos** and by **glBitmap**.

When the raster position coordinates are invalid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to the OpenGL state).

The following functions retrieve information related to **glRasterPos**:

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID
glGet with argument GL_CURRENT_RASTER_DISTANCE
glGet with argument GL_CURRENT_RASTER_COLOR
glGet with argument GL_CURRENT_RASTER_INDEX
glGet with argument GL_CURRENT_RASTER_TEXTURE_COORDS

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glRasterPos** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glBitmap**, **glCopyPixels**, **glDrawPixels**, **glEnd**, **glLight**, **glLightModel**, **glShadeModel**, **glTexCoord**, **glTexGen**, **glVertex**

glReadBuffer Quick Info

[New - Windows 95, OEM Service Release 2]

The **glReadBuffer** function selects a color buffer source for pixels.

```
void glReadBuffer(
  GLenum mode
);
```

Parameters

mode

A color buffer. Accepted values are GL_FRONT_LEFT, GL_FRONT_RIGHT, GL_BACK_LEFT, GL_BACK_RIGHT, GL_FRONT, GL_BACK, GL_LEFT, GL_RIGHT, and GL_AUX*i*, where *i* is between 0 and GL_AUX_BUFFERS -1.

Remarks

The **glReadBuffer** function specifies a color buffer as the source for subsequent **glReadPixels** and **glCopyPixels** commands. The *mode* parameter accepts one of twelve or more predefined values. (GL_AUX0 through GL_AUX3 are always defined.) In a fully configured system, GL_FRONT, GL_LEFT, and GL_FRONT_LEFT all name the front-left buffer, GL_FRONT_RIGHT and GL_RIGHT name the front-right buffer, and GL_BACK_LEFT and GL_BACK name the back-left buffer.

Nonstereo double-buffered configurations have only a front-left and a back-left buffer. Single-buffered configurations have a front-left and a front-right buffer if stereo, and only a front-left buffer if nonstereo. It is an error to specify a nonexistent buffer to **glReadBuffer**.

By default, *mode* is GL_FRONT in single-buffered configurations, and GL_BACK in double-buffered configurations.

The following function retrieves information related to **glReadBuffer**:

glGet with argument GL_READ_BUFFER

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*mode* was not one of the twelve (or more) accepted values.
GL_INVALID_OPERATION	*mode* specified a buffer that does not exist.
GL_INVALID_OPERATION	**glReadBuffer** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glCopyPixels**, **glDrawBuffer**, **glEnd**, **glReadPixels**

glReadPixels Quick Info

[New - Windows 95, OEM Service Release 2]

The **glReadPixels** function reads a block of pixels from the frame buffer.

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

Parameters

x, y

The window coordinates of the first pixel that is read from the frame buffer. This location is the lower-left corner of a rectangular block of pixels.

width, height

The dimensions of the pixel rectangle. The *width* and *height* parameters of one correspond to a single pixel.

format

The format of the pixel data. The following symbolic values are accepted:

GL_COLOR_INDEX

Color indexes are read from the color buffer selected by **glReadBuffer**. Each index is converted to fixed point, shifted left or right depending on the value and sign of GL_INDEX_SHIFT, and added to GL_INDEX_OFFSET. If GL_MAP_COLOR is GL_TRUE, indexes are replaced by their mappings in the table GL_PIXEL_MAP_I_TO_I.

GL_STENCIL_INDEX

Stencil values are read from the stencil buffer. Each index is converted to fixed point, shifted left or right depending on the value and sign of GL_INDEX_SHIFT, and added to GL_INDEX_OFFSET. If GL_MAP_STENCIL is GL_TRUE, indexes are replaced by their mappings in the table GL_PIXEL_MAP_S_TO_S.

GL_DEPTH_COMPONENT

Depth values are read from the depth buffer. Each component is converted to floating point such that the minimum depth value maps to 0.0 and the maximum value maps to 1.0. Each component is then multiplied by GL_DEPTH_SCALE, added to GL_DEPTH_BIAS, and finally clamped to the range [0,1].

GL_RED
GL_GREEN
GL_BLUE
GL_ALPHA
GL_RGB
GL_RGBA
GL_BGR_EXT
GL_BGRA_EXT
GL_LUMINANCE
GL_LUMINANCE_ALPHA

Processing differs depending on whether color buffers store color indexes or RGBA color components. If color indexes are stored, they are read from the color buffer selected by **glReadBuffer**. Each index is converted to fixed point, shifted left or right depending on the value and sign of GL_INDEX_SHIFT, and added to GL_INDEX_OFFSET. Indexes are then replaced by

the red, green, blue, and alpha values obtained by indexing the GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, and GL_PIXEL_MAP_I_TO_A tables.

If RGBA color components are stored in the color buffers, they are read from the color buffer selected by **glReadBuffer**. Each color component is converted to floating point such that zero intensity maps to 0.0 and full intensity maps to 1.0. Each component is then multiplied by GL_c_SCALE and added to GL_c_BIAS, where *c* is GL_RED, GL_GREEN, GL_BLUE, and GL_ALPHA. Each component is clamped to the range [0,1]. Finally, if GL_MAP_COLOR is GL_TRUE, each color component *c* is replaced by its mapping in the table GL_PIXEL_MAP_c_TO_c, where *c* again is GL_RED, GL_GREEN, GL_BLUE, and GL_ALPHA. Each component is scaled to the size of its corresponding table before the lookup is performed.

Finally, unneeded data is discarded. For example, GL_RED discards the green, blue, and alpha components, while GL_RGB discards only the alpha component. GL_LUMINANCE computes a single component value as the sum of the red, green, and blue components, and GL_LUMINANCE_ALPHA does the same, while keeping alpha as a second value.

type

The data type of the pixel data. Must be one of the following values:

Type	Index Mask	Component Conversion
GL_UNSIGNED_BYTE	2^*8*-1	(2^*8*-1)*c*
GL_BYTE	2^*7*-1	[2^*7*-1]*c*-1]/2
GL_BITMAP	1	1
GL_UNSIGNED_SHORT	2^*16*-1	(2^*16*-1) *c*
GL_SHORT	2^*15*-1	[(2^*15*-1) *c*-1] / 2
GL_UNSIGNED_INT	2^*32*-1	(2^*32*-1) *c*
GL_INT	2^*31*-1	[(2^31-1) *c*-1] / 2
GL_FLOAT	none	*c*

pixels

Returns the pixel data.

Remarks

The **glReadPixels** function returns pixel data from the frame buffer, starting with the pixel whose lower-left corner is at location (*x*, *y*), into client memory starting at location *pixels*. Several parameters control the processing of the pixel data before it is placed into client memory. These parameters are set with three commands: **glPixelStore**, **glPixelTransfer**, and **glPixelMap**. This topic describes the effects on **glReadPixels** of most, but not all of the parameters specified by these three commands.

The **glReadPixels** function returns values from each pixel with lower-left corner at (*x* + i, *y* + j) for $0 \leq i <$ *width* and $0 \leq j <$ *height*. This pixel is said to be the *i*th pixel in the *j*th row. Pixels are returned in row order from the lowest to the highest row, left to right in each row.

The shift, scale, bias, and lookup factors described above are all specified by **glPixelTransfer**. The lookup table contents are specified by **glPixelMap**.

The final step involves converting the indexes or components to the proper format, as specified by *type*. If *format* is GL_COLOR_INDEX or GL_STENCIL_INDEX and *type* is not GL_FLOAT, each index is masked with the mask value given in the following table. If *type* is GL_FLOAT, then each integer index is converted to single-precision floating-point format.

If *format* is GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_BGR_EXT, GL_BGRA_EXT, GL_LUMINANCE, or GL_LUMINANCE_ALPHA and *type* is not GL_FLOAT, each component is multiplied by the multiplier shown in the preceding table. If type is GL_FLOAT, then each component is passed as is (or converted to the client's single-precision floating-point format if it is different

from the one used by OpenGL).

Return values are placed in memory as follows. If *format* is GL_COLOR_INDEX, GL_STENCIL_INDEX, GL_DEPTH_COMPONENT, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, or GL_LUMINANCE, a single value is returned and the data for the *i*th pixel in the *j*th row is placed in location (*j*) *width* + *i*. GL_RGB and GL_BGR_EXT return three values, GL_RGBA and GL_BGRA_EXT return four values, and GL_LUMINANCE_ALPHA returns two values for each pixel, with all values corresponding to a single pixel occupying contiguous space in *pixels*. Storage parameters set by **glPixelStore**, such as GL_PACK_SWAP_BYTES and GL_PACK_LSB_FIRST, affect the way that data is written into memory. See **glPixelStore** for a description.

Values for pixels that lie outside the window connected to the current OpenGL context are undefined.

If an error is generated, no change is made to the contents of *pixels*.

The following function retrieves information related to **glReadPixels**:

- **glGet** with argument GL_INDEX_MODE

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*format* or *type* was not an accepted value.
GL_INVALID_VALUE	either *width* or *height* was negative.
GL_INVALID_OPERATION	*format* was GL_COLOR_INDEX and the color buffers stored RGBA or BGRA color components.
GL_INVALID_OPERATION	*format* was GL_STENCIL_INDEX and there was no stencil buffer.
GL_INVALID_OPERATION	*format* was GL_DEPTH_COMPONENT and there was no depth buffer.
GL_INVALID_OPERATION	**glReadPixels** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glCopyPixels**, **glDrawPixels**, **glEnd**, **glPixelMap**, **glPixelStore**, **glPixelTransfer**, **glReadBuffer**

glRectd, glRectf, glRecti, glRects, glRectdv, glRectfv, glRectiv, glRectsv

[New - Windows 95, OEM Service Release 2]

These functions draw a rectangle.

```
void glRectd(
  GLdouble x1,
  GLdouble y1,
  GLdouble x2,
  GLdouble y2
);

void glRectf(
  GLfloat x1,
  GLfloat y1,
  GLfloat x2,
  GLfloat y2
);

void glRecti(
  GLint x1,
  GLint y1,
  GLint x2,
  GLint y2
);

void glRects(
  GLshort x1,
  GLshort y1,
  GLshort x2,
  GLshort y2
);
```

Parameters

x1, y1
One vertex of a rectangle.

x2, y2
The opposite vertex of the rectangle.

```
void glRectdv(
  const GLdouble *v1,
  const GLdouble *v2
);

void glRectfv(
  const GLfloat *v1,
  const GLfloat *v2
);

void glRectiv(
  const GLint *v1,
  const GLint *v2
);
```

```
void glRectsv(
  const GLshort *v1,
  const GLshort *v2
);
```

Parameters

v1

A pointer to one vertex of a rectangle.

v2

A pointer to the opposite vertex of the rectangle.

Remarks

The **glRect** function supports efficient specification of rectangles as two corner points. Each rectangle command takes four arguments, organized either as two consecutive pairs of (*x*, *y*) coordinates, or as two pointers to arrays, each containing an (*x*,*y*) pair. The resulting rectangle is defined in the *z* = 0 plane.

The **glRect(***x1***,** *y1***,** *x2***,** *y2***)** function is exactly equivalent to the following sequence:

glBegin(GL_POLYGON**)**;
glVertex2(*x1*, *y1***)**;
glVertex2(*x2*, *y1***)**;
glVertex2(*x2*, *y2***)**;
glVertex2(*x1*, *y2***)**;
glEnd( );

Notice that if the second vertex is above and to the right of the first vertex, the rectangle is constructed with a counterclockwise winding.

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glRect** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glVertex**

glRenderMode Quick Info

[New - Windows 95, OEM Service Release 2]

The **glRenderMode** function sets the rasterization mode.

```
GLint glRenderMode(
  GLenum mode
);
```

Parameters

mode

The rasterization mode. The following three values are accepted. The default value is GL_RENDER.

GL_RENDER
: Render mode. Primitives are rasterized, producing pixel fragments, which are written into the frame buffer. This is the normal mode and also the default mode.

GL_SELECT
: Selection mode. No pixel fragments are produced, and no change to the frame buffer contents is made. Instead, a record of the names of primitives that would have been drawn if the render mode was GL_RENDER is returned in a select buffer, which must be created (see **glSelectBuffer**) before selection mode is entered.

GL_FEEDBACK
: Feedback mode. No pixel fragments are produced, and no change to the frame buffer contents is made. Instead, the coordinates and attributes of vertices that would have been drawn had the render mode been GL_RENDER are returned in a feedback buffer, which must be created (see **glFeedbackBuffer**) before feedback mode is entered.

Remarks

The **glRenderMode** function takes one argument, *mode*, which can assume one of three predefined values above.

The return value of the **glRenderMode** function is determined by the render mode at the time **glRenderMode** is called, rather than by *mode*. The values returned for the three render modes are as follows:

GL_RENDER
: Zero.

GL_SELECT
: The number of hit records transferred to the select buffer.

GL_FEEDBACK
: The number of values (not vertices) transferred to the feedback buffer.

Refer to **glSelectBuffer** and **glFeedbackBuffer** for more details concerning selection and feedback operation.

If an error is generated, **glRenderMode** returns zero regardless of the current render mode.

The following function retrieves information related to **glRenderMode**:

glGet with argument GL_RENDER_MODE

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*mode* was not one of the three accepted values.
GL_INVALID_OPERATION	**glSelectBuffer** was called while the render mode was GL_SELECT, or if **glRenderMode** was called with argument GL_SELECT before **glSelectBuffer** was called at least once.
GL_INVALID_OPERATION	**glFeedbackBuffer** was called while the render mode was GL_FEEDBACK, or if **glRenderMode** was called with argument GL_FEEDBACK before **glFeedbackBuffer** was called at least once.
GL_INVALID_OPERATION	**glRenderMode** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glFeedbackBuffer**, **glInitNames**, **glLoadName**, **glPassThrough**, **glPushName**, **glSelectBuffer**

glRotated, glRotatef

[New - Windows 95, OEM Service Release 2]

The **glRotated** and **glRotatef** functions multiply the current matrix by a rotation matrix.

```
void glRotated(
  GLdouble angle,
  GLdouble x,
  GLdouble y,
  GLdouble z
);

void glRotatef(
  GLfloat angle,
  GLfloat x,
  GLfloat y,
  GLfloat z
);
```

Parameters

angle
The angle of rotation, in degrees.

x, y, z
The *x*, *y*, and *z* coordinates of a vector, respectively.

Remarks

The **glRotate** function computes a matrix that performs a counterclockwise rotation of *angle* degrees about the vector from the origin through the point (*x*, *y*, *z*).

The current matrix (see **glMatrixMode**) is multiplied by this rotation matrix, with the product replacing the current matrix. That is, if M is the current matrix and R is the translation matrix, then M is replaced with M•R.

If the matrix mode is either GL_MODELVIEW or GL_PROJECTION, all objects drawn after **glRotate** is called are rotated. Use **glPushMatrix** and **glPopMatrix** to save and restore the unrotated coordinate system.

The following functions retrieve information related to **glRotate**:

glGet with argument GL_RENDER_MODE
glGet with argument GL_MATRIX_MODE
glGet with argument GL_MODELVIEW_MATRIX
glGet with argument GL_PROJECTION_MATRIX
glGet with argument GL_TEXTURE_MATRIX

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glRotate** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, glEnd, glMatrixMode, glMultMatrix, glPushMatrix, glScale, glTranslate

glScaled, glScalef

[New - Windows 95, OEM Service Release 2]

The **glScaled** and **glScalef** functions multiply the current matrix by a general scaling matrix.

```
void glScaled(
  GLdouble x,
  GLdouble y,
  GLdouble z
);

void glScalef(
  GLfloat x,
  GLfloat y,
  GLfloat z
);
```

Parameters

x, y, z
Scale factors along the *x*, *y*, and *z* axes, respectively.

Remarks

The **glScale** function produces a general scaling along the *x*, *y*, and *z* axes. The three arguments indicate the desired scale factors along each of the three axes. The resulting matrix is

{ewc msdncd, EWGraphic, bsd23545 20 /a "SDK.BMP"}

The current matrix (see **glMatrixMode**) is multiplied by this scale matrix, with the product replacing the current matrix. That is, if M is the current matrix and S is the scale matrix, then M is replaced with M•S.

If the matrix mode is either GL_MODELVIEW or GL_PROJECTION, all objects drawn after **glScale** is called are scaled. Use **glPushMatrix** and **glPopMatrix** to save and restore the unscaled coordinate system.

If scale factors other than 1.0 are applied to the modelview matrix and lighting is enabled, automatic normalization of normals should probably also be enabled (**glEnable** and **glDisable** with argument GL_NORMALIZE).

The following functions retrieve information related to **glScale**:

glGet with argument GL_MATRIX_MODE
glGet with argument GL_MODELVIEW_MATRIX
glGet with argument GL_PROJECTION_MATRIX
glGet with argument GL_TEXTURE_MATRIX

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glScale** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, glEnd, glMatrixMode, glMultMatrix, glPushMatrix, glRotate, glTranslate

glScissor Quick Info

[New - Windows 95, OEM Service Release 2]

The **glScissor** function defines the scissor box.

```
void glScissor(
  GLint x,
  GLint y,
  GLsizei width,
  GLsizei height
);
```

Parameters

x, y
The lower-left corner of the scissor box. Initially (0,0).

width, height
The width and height of the scissor box. When an OpenGL context is *first* attached to a window, *width* and *height* are set to the dimensions of that window.

Remarks

The **glScissor** function defines a rectangle, called the scissor box, in window coordinates. The first two parameters, *x* and *y*, specify the lower-left corner of the box. The *width* and *height* parameters specify the width and height of the box.

The scissor test is enabled and disabled using **glEnable** and **glDisable** with argument GL_SCISSOR_TEST. While the scissor test is enabled, only pixels that lie within the scissor box can be modified by drawing commands. Window coordinates have integer values at the shared corners of frame buffer pixels, so **glScissor(**0,0,1,1**)** allows only the lower-left pixel in the window to be modified, and **glScissor(**0,0,0,0**)** disallows modification to all pixels in the window.

When the scissor test is disabled, it is as though the scissor box includes the entire window.

The following functions retrieve information related to **glScissor**:

glGet with argument GL_SCISSOR_BOX
glIsEnabled with argument GL_SCISSOR_TEST

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_VALUE	either *width* or *height* was negative.
GL_INVALID_OPERATION	**glScissor** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnable**, **glEnd**, **glIsEnabled**, **glViewport**

glSelectBuffer Quick Info

[New - Windows 95, OEM Service Release 2]

The **glSelectBuffer** function establishes a buffer for selection mode values.

```
void glSelectBuffer(
  GLsizei size,
  GLuint *buffer
);
```

Parameters

size
The size of *buffer*.

buffer
Returns the selection data.

Remarks

The **glSelectBuffer** function has two parameters: *buffer* is a pointer to an array of unsigned integers, and *size* indicates the size of the array. The *buffer* parameter returns values from the name stack (see **glInitNames**, **glLoadName**, **glPushName**) when the rendering mode is GL_SELECT (see **glRenderMode**). The **glSelectBuffer** function must be issued before selection mode is enabled, and it must not be issued while the rendering mode is GL_SELECT.

Selection is used by a programmer to determine which primitives are drawn into some region of a window. The region is defined by the current modelview and perspective matrices.

In selection mode, no pixel fragments are produced from rasterization. Instead, if a primitive intersects the clip volume defined by the viewing frustum and the user-defined clipping planes, this primitive causes a selection hit. (With polygons, no hit occurs if the polygon is culled.) When a change is made to the name stack, or when **glRenderMode** is called, a hit record is copied to *buffer* if any hits have occurred since the last such event (either a name stack change or a **glRenderMode** call). The hit record consists of the number of names in the name stack at the time of the event; followed by the minimum and maximum depth values of all vertices that hit since the previous event; followed by the name stack contents, bottom name first.

Returned depth values are mapped such that the largest unsigned integer value corresponds to window coordinate depth 1.0, and zero corresponds to window coordinate depth 0.0.

An internal index into *buffer* is reset to zero whenever selection mode is entered. Each time a hit record is copied into *buffer*, the index is incremented to point to the cell just past the end of the block of names—that is, to the next available cell. If the hit record is larger than the number of remaining locations in *buffer*, as much data as can fit is copied, and the overflow flag is set. If the name stack is empty when a hit record is copied, that record consists of zero followed by the minimum and maximum depth values.

Selection mode is exited by calling **glRenderMode** with an argument other than GL_SELECT. Whenever **glRenderMode** is called while the render mode is GL_SELECT, it returns the number of hit records copied to *buffer*, resets the overflow flag and the selection buffer pointer, and initializes the name stack to be empty. If the overflow bit was set when **glRenderMode** was called, a negative hit record count is returned.

The contents of *buffer* are undefined until **glRenderMode** is called with an argument other than GL_SELECT.

The **glBegin**/**glEnd** primitives and calls to **glRasterPos** can result in hits.

The following function retrieves information related to **glSelectBuffer**:

glGet with argument GL_NAME_STACK_DEPTH

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_VALUE	*size* was negative.
GL_INVALID_OPERATION	**glSelectBuffer** was called while the render mode was GL_SELECT, or if **glRenderMode** was called with argument GL_SELECT before **glSelectBuffer** was called at least once.
GL_INVALID_OPERATION	**glSelectBuffer** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glFeedbackBuffer**, **glInitNames**, **glLoadName**, **glPushName**, **glRenderMode**

glShadeModel Quick Info

[New - Windows 95, OEM Service Release 2]

The **glShadeModel** function selects flat or smooth shading.

```
void glShadeModel(

  GLenum mode
);
```

Parameters

mode

A symbolic value representing a shading technique. Accepted values are GL_FLAT and GL_SMOOTH. The default is GL_SMOOTH.

Remarks

OpenGL primitives can have either flat or smooth shading. Smooth shading, the default, causes the computed colors of vertices to be interpolated as the primitive is rasterized, typically assigning different colors to each resulting pixel fragment. Flat shading selects the computed color of just one vertex and assigns it to all the pixel fragments generated by rasterizing a single primitive. In either case, the computed color of a vertex is the result of lighting, if lighting is enabled, or it is the current color at the time the vertex was specified, if lighting is disabled.

Flat and smooth shading are indistinguishable for points. Counting vertices and primitives from one, starting when **glBegin** is issued, each flat-shaded line segment *i* is given the computed color of vertex *i* + 1, its second vertex. Counting similarly from one, each flat-shaded polygon is given the computed color of the vertex listed in the following table. This is the last vertex to specify the polygon in all cases except single polygons, where the first vertex specifies the flat-shaded color.

Primitive Type of Polygon i	Vertex
Single polygon (*i*≡1)	1
Triangle strip	*i* + 2
Triangle fan	*i* + 2
Independent triangle	3*i*
Quad strip	2*i* + 2
Independent quad	4*i*

Flat and smooth shading are specified by **glShadeModel** with *mode* set to GL_FLAT and GL_SMOOTH, respectively.

The following function retrieves information related to **glShadeModel**:

glGet with argument GL_SHADE_MODEL

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*mode* was any value other than GL_FLAT or GL_SMOOTH.
GL_INVALID_OPERATION	**glShadeModel** was called between a call to **glBegin** and the corresponding

call to **glEnd**.

See Also

glBegin, **glColor**, **glEnd**, **glLight**, **glLightModel**

glStencilFunc Quick Info

[New - Windows 95, OEM Service Release 2]

The **glStencilFunc** function sets the function and reference value for stencil testing.

```
void glStencilFunc(
  GLenum func,
  GLint ref,
  GLuint mask
);
```

Parameters

func

The test function. The following eight tokens are valid:

GL_NEVER
Always fails.

GL_LESS
Passes if (*ref* & *mask*) < (*stencil* & *mask*).

GL_LEQUAL
Passes if (*ref* & *mask*) ≤ (*stencil* & *mask*).

GL_GREATER
Passes if (*ref* & *mask*) > (*stencil* & *mask*).

GL_GEQUAL
Passes if (*ref* & *mask*) ≥ (*stencil* & *mask*).

GL_EQUAL
Passes if (*ref* & *mask*) = (*stencil* & *mask*).

GL_NOTEQUAL
Passes if (*ref* & *mask*) ≠ (*stencil* & *mask*).

GL_ALWAYS
Always passes.

ref

The reference value for the stencil test. The *ref* parameter is clamped to the range *[0,2^n - 1],* where *n* is the number of bitplanes in the stencil buffer.

mask

A mask that is **AND**ed with both the reference value and the stored stencil value when the test is done.

Remarks

Stenciling, like *z*-buffering, enables and disables drawing on a per-pixel basis. You draw into the stencil planes using OpenGL drawing primitives, then render geometry and images, using the stencil planes to mask out portions of the screen. Stenciling is typically used in multipass rendering algorithms to achieve special effects, such as decals, outlining, and constructive solid geometry rendering.

The stencil test conditionally eliminates a pixel based on the outcome of a comparison between the reference value and the value in the stencil buffer. The test is enabled by **glEnable** and **glDisable** with argument GL_STENCIL. Actions taken based on the outcome of the stencil test are specified with **glStencilOp**.

The *func* parameter is a symbolic constant that determines the stencil comparison function. It accepts one of the eight values shown above. The *ref* parameter is an integer reference value that is used in the stencil comparison. It is clamped to the range [0,2^*n* - 1], where *n* is the number of bitplanes in the stencil buffer. The *mask* parameter is bitwise **AND**ed with both the reference value and the stored stencil value,

with the **AND**ed values participating in the comparison.

If *stencil* represents the value stored in the corresponding stencil buffer location, the preceding list shows the effect of each comparison function that can be specified by *func*. Only if the comparison succeeds is the pixel passed through to the next stage in the rasterization process (see **glStencilOp**). All tests treat *stencil* values as unsigned integers in the range [0,2^*n* - 1], where *n* is the number of bitplanes in the stencil buffer.

Initially, the stencil test is disabled. If there is no stencil buffer, no stencil modification can occur and it is as if the stencil test always passes.

The following functions retrieve information related to **glStencilFunc**:

glGet with argument GL_STENCIL_FUNC
glGet with argument GL_STENCIL_VALUE_MASK
glGet with argument GL_STENCIL_REF
glGet with argument GL_STENCIL_BITS
glIsEnabled with argument GL_STENCIL_TEST

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*func* was not one of the eight accepted values.
GL_INVALID_OPERATION	**glStencilFunc** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glAlphaFunc, **glBegin**, **glBlendFunc**, **glDepthFunc**, **glEnable**, **glEnd**, **glIsEnabled**, **glLogicOp**, **glStencilOp**

glStencilMask Quick Info

[New - Windows 95, OEM Service Release 2]

The **glStencilMask** function controls the writing of individual bits in the stencil planes.

```
void glStencilMask(
  GLuint mask
);
```

Parameters

mask

A bit mask to enable and disable writing of individual bits in the stencil planes. Initially, the mask is all ones.

Remarks

The **glStencilMask** function controls the writing of individual bits in the stencil planes. The least significant *n* bits of *mask*, where *n* is the number of bits in the stencil buffer, specify a mask. Wherever a one appears in the mask, the corresponding bit in the stencil buffer is made writable. Where a zero appears, the bit is write-protected. Initially, all bits are enabled for writing.

The following functions retrieve information related to **glStencilMask**:

glGet with argument GL_STENCIL_WRITEMASK

glGet with argument GL_STENCIL_BITS

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glStencilMask** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glColorMask**, **glDepthMask**, **glEnd**, **glIndexMask**, **glStencilFunc**, **glStencilOp**

glStencilOp Quick Info

[New - Windows 95, OEM Service Release 2]

The **glStencilOp** function sets the stencil test actions.

```
void glStencilOp(
  GLenum fail,
  GLenum zfail,
  GLenum zpass
);
```

Parameters

fail

The action to take when the stencil test fails. The following six symbolic constants are accepted:

GL_KEEP
Keeps the current value.

GL_ZERO
Sets the stencil buffer value to zero.

GL_REPLACE
Sets the stencil buffer value to *ref*, as specified by **glStencilFunc**.

GL_INCR
Increments the current stencil buffer value. Clamps to the maximum representable unsigned value.

GL_DECR
Decrements the current stencil buffer value. Clamps to zero.

GL_INVERT
Bitwise inverts the current stencil buffer value.

zfail

Stencil action when the stencil test passes, but the depth test fails. Accepts the same symbolic constants as *fail.*

zpass

Stencil action when both the stencil test and the depth test pass, or when the stencil test passes and either there is no depth buffer or depth testing is not enabled. Accepts the same symbolic constants as *fail*.

Remarks

Stenciling, like *z*-buffering, enables and disables drawing on a per-pixel basis. You draw into the stencil planes using OpenGL drawing primitives, then render geometry and images, using the stencil planes to mask out portions of the screen. Stenciling is typically used in multipass rendering algorithms to achieve special effects, such as decals, outlining, and constructive solid geometry rendering.

The stencil test conditionally eliminates a pixel based on the outcome of a comparison between the value in the stencil buffer and a reference value. The test is enabled with **glEnable** and **glDisable** calls with argument GL_STENCIL, and controlled with **glStencilFunc**.

The **glStencilOp** function takes three arguments that indicate what happens to the stored stencil value while stenciling is enabled. If the stencil test fails, no change is made to the pixel's color or depth buffers, and *fail* specifies what happens to the stencil buffer contents.

Stencil buffer values are treated as unsigned integers. When incremented and decremented, values are clamped to 0 and 2^*n* - 1, where *n* is the value returned by querying GL_STENCIL_BITS.

The other two arguments to **glStencilOp** specify stencil buffer actions should subsequent depth buffer

tests succeed (*zpass*) or fail (*zfail*). (See **glDepthFunc**.) They are specified using the same six symbolic constants as *fail*. Note that *zfail* is ignored when there is no depth buffer, or when the depth buffer is not enabled. In these cases, *fail* and *zpass* specify stencil action when the stencil test fails and passes, respectively.

Initially the stencil test is disabled. If there is no stencil buffer, no stencil modification can occur and it is as if the stencil tests always pass, regardless of any call to **glStencilOp**.

The following functions retrieve information related to **glStencilOp**:

glGet with argument GL_STENCIL_FAIL
glGet with argument GL_STENCIL_PASS_DEPTH_PASS
glGet with argument GL_STENCIL_PASS_DEPTH_FAIL
glGet with argument GL_STENCIL_BITS
glIsEnabled with argument GL_STENCIL_TEST

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*fail*, *zfail*, or *zpass* was any value other than the six defined constant values.
GL_INVALID_OPERATION	**glStencilOp** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glAlphaFunc, **glBegin**, **glBlendFunc**, **glDepthFunc**, **glEnable**, **glEnd**, **glIsEnabled**, **glLogicOp**, **glStencilFunc**

glTexCoord

[New - Windows 95, OEM Service Release 2]

glTexCoord1d, glTexCoord1f, glTexCoord1i, glTexCoord1s, glTexCoord2d, glTexCoord2f, glTexCoord2i, glTexCoord2s, glTexCoord3d, glTexCoord3f, glTexCoord3i, glTexCoord3s, glTexCoord4d, glTexCoord4f, glTexCoord4i, glTexCoord4s, glTexCoord1dv, glTexCoord1fv, glTexCoord1iv, glTexCoord1sv, glTexCoord2dv, glTexCoord2fv, glTexCoord2iv, glTexCoord2sv, glTexCoord3dv, glTexCoord3fv, glTexCoord3iv, glTexCoord3sv, glTexCoord4dv, glTexCoord4fv, glTexCoord4iv, glTexCoord4sv

These functions set the current texture coordinates.

```
void glTexCoord1d(
  GLdouble s
);

void glTexCoord1f(
  GLfloat s
);

void glTexCoord1i(
  GLint s
);

void glTexCoord1s(
  GLshort s
);

void glTexCoord2d(
  GLdouble s,
  GLdouble t
);

void glTexCoord2f(
  GLfloat s,
  GLfloat t
);

void glTexCoord2i(
  GLint s,
  GLint t
);

void glTexCoord2s(
  GLshort s,
  GLshort t
);

void glTexCoord3d(
  GLdouble s,
  GLdouble t,
  GLdouble r
);

void glTexCoord3f(
  GLfloat s,
  GLfloat t,
  GLfloat r
```

```
);

void glTexCoord3i(
  GLint s,
  GLint t,
  GLint r
);

void glTexCoord3s(
  GLshort s,
  GLshort t,
  GLshort r
);

void glTexCoord4d(
  GLdouble s,
  GLdouble t,
  GLdouble r,
  GLdouble q
);

void glTexCoord4f(
  GLfloat s,
  GLfloat t,
  GLfloat r,
  GLfloat q
);

void glTexCoord4i(
  GLint s,
  GLint t,
  GLint r,
  GLint q
);

void glTexCoord4s(
  GLshort s,
  GLshort t,
  GLshort r,
  GLshort q
);
```

Parameters

s, t, r, q
The *s*, *t*, *r*, and *q* texture coordinates. Not all parameters are present in all forms of the command.

```
void glTexCoord1dv(
  const GLdouble *v
);

void glTexCoord1fv(
  const GLfloat *v
);

void glTexCoord1iv(
  const GLint *v
);

void glTexCoord1sv(
```

```
  const GLshort *v
);

void glTexCoord2dv(
  const GLdouble *v
);

void glTexCoord2fv(
  const GLfloat *v
);

void glTexCoord2iv(
  const GLint *v
);

void glTexCoord2sv(
  const GLshort *v
);

void glTexCoord3dv(
  const GLdouble *v
);

void glTexCoord3fv(
  const GLfloat *v
);

void glTexCoord3iv(
  const GLint *v
);

void glTexCoord3sv(
  const GLshort *v
);

void glTexCoord4dv(
  const GLdouble *v
);

void glTexCoord4fv(
  const GLfloat *v
);

void glTexCoord4iv(
  const GLint *v
);

void glTexCoord4sv(
  const GLshort *v
);
```

Parameters

v

A pointer to an array of one, two, three, or four elements, which in turn specify the *s, t*, *r,* and *q* texture coordinates.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data that is associated with polygon vertices.

The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The **glTexCoord1** function sets the current texture coordinates to (*s*, 0, 0, 1); a call to **glTexCoord2** sets them to (*s*, *t*, 0, 1). Similarly, **glTexCoord3** specifies the texture coordinates as (*s*, *t*, *r*, 1), and **glTexCoord4** defines all four components explicitly as (*s*, *t*, *r*, *q*).

The current texture coordinates can be updated at any time. In particular, **glTexCoord** can be called between a call to **glBegin** and the corresponding call to **glEnd**.

The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

See Also

glVertex

glTexCoordPointer

[New - Windows 95, OEM Service Release 2]

The **glTexCoordPointer** function defines an array of texture coordinates.

```
void glTexCoordPointer(
  GLint size,
  GLenum type,
  GLsizei stride,
  GLsizei count,
  const GLvoid *pointer
);
```

Parameters

size
The number of coordinates per array element. The value of *size* must be 1, 2, 3, or 4.

type
The data type of each texture coordinate in the array using the following symbolic constants: GL_SHORT, GL_INT, GL_FLOAT, and GL_DOUBLE.

stride
The byte offset between consecutive array elements. When *stride* is zero, the array elements are tightly packed in the array.

count
The number of array elements, counting from the first, that are static.

pointer
A pointer to the first coordinate of the first element in the array.

Remarks

The **glTexCoordPointer** function specifies the location and data of an array of texture coordinates to use when rendering.The *size* parameter specifies the number of coordinates used for each element of the array.The *type* parameter specifies the data type of each texture coordinate. The *stride* parameter determines the byte offset from one array element to the next, enabling the packing of vertices and attributes in a single array or storage in separate arrays. In some implementations, storing the vertices and attributes in a single array can be more efficient than using separate arrays. Starting from the first array element, *count* indicates the total number of static elements. Your application can modify static elements, but once the elements are modified, the application must explicitly specify the array again before using the array for any rendering. Non-static array elements are not accessed until you call **glDrawArrays** or **glArrayElement**.

A texture coordinate array is enabled when you specify the GL_TEXTURE_COORD_ARRAY constant with **glEnableClientState**. When enabled, **glDrawArrays** and **glArrayElement** use the texture coordinate array. By default the texture coordinate array is disabled.

You cannot include **glTexCoordPointer** in display lists.

When you specify a texture coordinate array using **glTexCoordPointer**, the values of all the function's texture coordinate array parameters are saved in a client-side state, and static array elements can be cached. Because the texture coordinate array parameters are client-side state, their values are not saved or restored by **glPushAttrib** and **glPopAttrib**.

Although no error is generated when you call **glTexCoordPointer** within **glBegin** and **glEnd** pairs, the results are undefined.

The following functions retrieve information related to **glTexCoordPointer**:

glIsEnabled with argument GL_TEXTURE_COORD_ARRAY
glGet with argument GL_TEXTURE_COORD_ARRAY_SIZE
glGet with argument GL_TEXTURE_COORD_ARRAY_STRIDE
glGet with argument GL_TEXTURE_COORD_ARRAY_COUNT
glGet with argument GL_TEXTURE_COORD_ARRAY_TYPE
glGetPointerv with argument GL_TEXTURE_COORD_ARRAY_POINTER

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_VALUE	*size* was not 1, 2, 3, or 4.
GL_INVALID_ENUM	*type* was not an accepted value.
GL_INVALID_VALUE	*stride* or *count* was negative.

See Also

glArrayElement, **glColorPointer**, **glDrawArrays**, **glEdgeFlagPointer**, **glGetPointerv**, **glGetString**, **glIndexPointer**, **glIsEnabled**, **glNormalPointer**, **glVertexPointer**

glTexEnvf, glTexEnvi, glTexEnvfv, glTexEnviv

[New - Windows 95, OEM Service Release 2]

These functions set texture environment parameters.

```
void glTexEnvf(
  GLenum target,
  GLenum pname,
  GLfloat param
);
```

```
void glTexEnvi(
  GLenum target,
  GLenum pname,
  GLint param
);
```

Parameters

target
A texture environment. Must be GL_TEXTURE_ENV.

pname
The symbolic name of a single-valued texture environment parameter. Must be GL_TEXTURE_ENV_MODE.

param
A single symbolic constant, one of GL_MODULATE, GL_DECAL, or GL_BLEND.

```
void glTexEnvfv(
  GLenum target,
  GLenum pname,
  const GLfloat *params
);
```

```
void glTexEnviv(
  GLenum target,
  GLenum pname,
  const GLint *params
);
```

Parameters

target
A texture environment. Must be GL_TEXTURE_ENV.

pname
The symbolic name of a texture environment parameter. Accepted values are GL_TEXTURE_ENV_MODE and GL_TEXTURE_ENV_COLOR.

params
A pointer to an array of parameters: either a single symbolic constant or an RGBA color.

Remarks

A texture environment specifies how texture values are interpreted when a fragment is textured. The *target* parameter must be GL_TEXTURE_ENV. The *pname* parameter can be either GL_TEXTURE_ENV_MODE or GL_TEXTURE_ENV_COLOR.

If *pname* is GL_TEXTURE_ENV_MODE, then *params* is (or points to) the symbolic name of a texture function. Three texture functions are defined: GL_MODULATE, GL_DECAL, and GL_BLEND.

A texture function acts on the fragment to be textured using the texture image value that applies to the fragment (see **glTexParameter**) and produces an RGBA color for that fragment. The following table shows how the RGBA color is produced for each of the three texture functions that can be chosen. *C* is a triple of color values (RGB) and *A* is the associated alpha value. RGBA values extracted from a texture image are in the range [0,1]. The subscript *f* refers to the incoming fragment, the subscript *t* to the texture image, the subscript *c* to the texture environment color, and subscript *v* indicates a value produced by the texture function.

A texture image can have up to four components per texture element (see **glTexImage1D** and **glTexImage2D**). In a one-component image, $L_{(t)}$ indicates that single component. A two-component image uses $L_{(t)}$ and $A_{(t)}$. A three-component image has only a color value, $C_{(t)}$. A four-component image has both a color value $C_{(t)}$ and an alpha value $A_{(t)}$.

{ewc msdncd, EWGraphic, bsd23545 21 /a "SDK.BMP"}

If *pname* is GL_TEXTURE_ENV_COLOR, *params* is a pointer to an array that holds an RGBA color consisting of four values. Integer color components are interpreted linearly such that the most positive integer maps to 1.0, and the most negative integer maps to -1.0. The values are clamped to the range [0,1] when they are specified. $C_{(c)}$ takes these four values.

GL_TEXTURE_ENV_MODE defaults to GL_MODULATE and GL_TEXTURE_ENV_COLOR defaults to (0,0,0,0).

The following function retrieves information related to **glTexEnv**:

glGetTexEnv

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*target* or *pname* was not one of the accepted defined values, or when *params* should have had a defined constant value (based on the value of *pname*) and did not.
GL_INVALID_OPERATION	**glTexEnv** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glTexImage1D**, **glTexImage2D**, **glTexParameter**

glTexGend, glTexGenf, glTexGeni, glTexGendv, glTexGenfv, glTexGeniv

[New - Windows 95, OEM Service Release 2]

These functions control the generation of texture coordinates.

```
void glTexGend(
  GLenum coord,
  GLenum pname,
  GLdouble param
);

void glTexGenf(
  GLenum coord,
  GLenum pname,
  GLfloat param
);

void glTexGeni(
  GLenum coord,
  GLenum pname,
  GLint param
);
```

Parameters

coord
A texture coordinate. Must be one of the following: GL_S, GL_T, GL_R, or GL_Q.

pname
The symbolic name of the texture-coordinate generation function. Must be GL_TEXTURE_GEN_MODE.

param
A single-valued texture generation parameter, one of GL_OBJECT_LINEAR, GL_EYE_LINEAR, or GL_SPHERE_MAP.

```
void glTexGendv(
  GLenum coord,
  GLenum pname,
  const GLdouble *params
);

void glTexGenfv(
  GLenum coord,
  GLenum pname,
  const GLfloat *params
);

void glTexGeniv(
  GLenum coord,
  GLenum pname,
  const GLint *params
);
```

Parameters

coord

A texture coordinate. Must be one of the following: GL_S, GL_T, GL_R, or GL_Q.

pname

The symbolic name of the texture-coordinate generation function or function parameters. Must be GL_TEXTURE_GEN_MODE, GL_OBJECT_PLANE, or GL_EYE_PLANE.

params

A pointer to an array of texture generation parameters. If *pname* is GL_TEXTURE_GEN_MODE, then the array must contain a single symbolic constant, one of GL_OBJECT_LINEAR, GL_EYE_LINEAR, or GL_SPHERE_MAP. Otherwise, *params* holds the coefficients for the texture-coordinate generation function specified by *pname*.

Remarks

The **glTexGen** function selects a texture-coordinate generation function or supplies coefficients for one of the functions. The *coord* parameter names one of the (*s*,*t*,*r*,*q*) texture coordinates, and it must be one of these symbols: GL_S, GL_T, GL_R, or GL_Q. The *pname* parameter must be one of three symbolic constants: GL_TEXTURE_GEN_MODE, GL_OBJECT_PLANE, or GL_EYE_PLANE. If *pname* is GL_TEXTURE_GEN_MODE, then *params* chooses a mode, one of GL_OBJECT_LINEAR, GL_EYE_LINEAR, or GL_SPHERE_MAP. If *pname* is either GL_OBJECT_PLANE or GL_EYE_PLANE, *params* contains coefficients for the corresponding texture generation function.

If the texture generation function is GL_OBJECT_LINEAR, the function

{ewc msdncd, EWGraphic, bsd23545 22 /a "SDK.BMP"}

is used, where *g* is the value computed for the coordinate named in *coord*; $p_{(1)}$, $p_{(2)}$, $p_{(3)}$, and $p_{(4)}$ are the four values supplied in *params*; and $x_{(o)}$, $y_{(o)}$, $z_{(o)}$, and $w_{(o)}$ are the object coordinates of the vertex. This function can be used to texture-map terrain using sea level as a reference plane (defined by $p_{(1)}$, $p_{(2)}$, $p_{(3)}$, and $p_{(4)}$). The altitude of a terrain vertex is computed by the GL_OBJECT_LINEAR coordinate generation function as its distance from sea level; that altitude is used to index the texture image to map white snow onto peaks and green grass onto foothills, for example.

If the texture generation function is GL_EYE_LINEAR, the function

{ewc msdncd, EWGraphic, bsd23545 23 /a "SDK.BMP"}

is used, where

{ewc msdncd, EWGraphic, bsd23545 24 /a "SDK.BMP"}

and $x_{(e)}$, $y_{(e)}$, $z_{(e)}$, and $w_{(e)}$ are the eye coordinates of the vertex, $p_{(1)}$, $p_{(2)}$, $p_{(3)}$, and $p_{(4)}$ are the values supplied in *params*, and *M* is the modelview matrix when **glTexGen** is invoked. If *M* is poorly conditioned or singular, texture coordinates generated by the resulting function may be inaccurate or undefined.

Note that the values in *params* define a reference plane in eye coordinates. The modelview matrix that is applied to them may not be the same one in effect when the polygon vertices are transformed. This function establishes a field of texture coordinates that can produce dynamic contour lines on moving objects.

If *pname* is GL_SPHERE_MAP and *coord* is either GL_S or GL_T, *s* and *t* texture coordinates are generated as follows. Let **u** be the unit vector pointing from the origin to the polygon vertex (in eye coordinates). Let $\mathbf{n}'$ be the current normal, after transformation to eye coordinates. Let $f = (f_{(x)}\, f_{(y)}\, f_{(z)})^T$ be the reflection vector such that

{ewc msdncd, EWGraphic, bsd23545 25 /a "SDK.BMP"}

Finally, let

{ewc msdncd, EWGraphic, bsd23545 26 /a "SDK.BMP"}

Then the values assigned to the *i* and *t* texture coordinates are

{ewc msdncd, EWGraphic, bsd23545 27 /a "SDK.BMP"}

A texture-coordinate generation function is enabled or disabled using **glEnable** or **glDisable** with one of the symbolic texture-coordinate names (GL_TEXTURE_GEN_S, GL_TEXTURE_GEN_T, GL_TEXTURE_GEN_R, or GL_TEXTURE_GEN_Q) as the argument. When enabled, the specified texture coordinate is computed according to the generating function associated with that coordinate. When disabled, subsequent vertices take the specified texture coordinate from the current set of texture coordinates. Initially, all texture generation functions are set to GL_EYE_LINEAR and are disabled. Both *s* plane equations are (1,0,0,0); both *t* plane equations are (0,1,0,0); and all *r* and *q* plane equations are (0,0,0,0).

The following functions retrieve information related to **glTexGen**:

glGetTexGen
glIsEnabled with argument GL_TEXTURE_GEN_S
glIsEnabled with argument GL_TEXTURE_GEN_T
glIsEnabled with argument GL_TEXTURE_GEN_R
glIsEnabled with argument GL_TEXTURE_GEN_Q

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*coord* or *pname* was not an accepted defined value, or when *pname* was GL_TEXTURE_GEN_MODE and *params* was not an accepted defined value.
GL_INVALID_ENUM	*pname* was GL_TEXTURE_GEN_MODE, *params* was GL_SPHERE_MAP, and *coord* was either GL_R or GL_Q.
GL_INVALID_OPERATION	**glTexGen** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glGetTexGen**, **glIsEnabled**, **glTexEnv**, **glTexImage1D**, **glTexImage2D**, **glTexParameter**

glTexImage1D Quick Info

[New - Windows 95, OEM Service Release 2]

The **glTexImage1D** function specifies a one-dimensional texture image.

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

Parameters

target
The target texture. Must be GL_TEXTURE_1D.

level
The level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.

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

width
The width of the texture image. Must be 2^*n* + 2(*border*) for some integer *n*. The height of the texture image is 1.

border
The width of the border. Must be either 0 or 1.

format
The format of the pixel data. It can assume one of nine symbolic values:

GL_COLOR_INDEX
Each element is a single value, a color index. It is converted to fixed point (with an unspecified number of 0 bits to the right of the binary point), shifted left or right depending on the value and sign of GL_INDEX_SHIFT, and added to GL_INDEX_OFFSET (see **glPixelTransfer**). The resulting index is converted to a set of color components using the GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, and GL_PIXEL_MAP_I_TO_A tables, and clamped to the range [0,1].

GL_RED
Each element is a single red component. It is converted to floating point and assembled into an RGBA element by attaching 0.0 for green and blue, and 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

GL_GREEN
Each element is a single green component. It is converted to floating point and assembled into an RGBA element by attaching 0.0 for red and blue, and 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

GL_BLUE
Each element is a single blue component. It is converted to floating point and assembled into an RGBA element by attaching 0.0 for red and green, and 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and

clamped to the range [0,1] (see **glPixelTransfer**).

GL_ALPHA
: Each element is a single red component. It is converted to floating point and assembled into an RGBA element by attaching 0.0 for red, green, and blue. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

GL_RGB
: Each element is an RGB triple. It is converted to floating point and assembled into an RGBA element by attaching 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

GL_RGBA
: Each element is a complete RGBA element. It is converted to floating point. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

GL_BGR_EXT
: Each pixel is a group of three components in this order: blue, green, red.

  GL_BGR_EXT provides a format that matches the memory layout of Windows device-independent bitmaps (DIBs). Thus your applications can use the same data with Win32 function calls and OpenGL pixel function calls.

GL_BGRA_EXT
: Each pixel is a group of four components in this order: blue, green, red, alpha.

  GL_BGRA_EXT provides a format that matches the memory layout of Windows device-independent bitmaps (DIBs). Thus your applications can use the same data with Win32 function calls and OpenGL pixel function calls.

GL_LUMINANCE
: Each element is a single luminance value. It is converted to floating point, and then assembled into an RGBA element by replicating the luminance value three times for red, green, and blue, and attaching 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

GL_LUMINANCE_ALPHA
: Each element is a luminance/alpha pair. It is converted to floating point, and then assembled into an RGBA element by replicating the luminance value three times for red, green, and blue. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

type
: The data type of the pixel data. The following symbolic values are accepted: GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, and GL_FLOAT.

pixels
: A pointer to the image data in memory.

Remarks

The **glTexImage1D** function specifies a one-dimensional texture image. Texturing maps a portion of a specified *texture image* onto each graphical primitive for which texturing is enabled. One-dimensional texturing is enabled and disabled using **glEnable** and **glDisable** with argument GL_TEXTURE_1D.

Texture images are defined with **glTexImage1D**. The arguments describe the parameters of the texture image, such as width, width of the border, level-of-detail number (see **glTexParameter**), and number of color components provided. The last three arguments describe the way the image is represented in memory. These arguments are identical to the pixel formats used for **glDrawPixels**.

Data is read from *pixels* as a sequence of signed or unsigned bytes, shorts or longs, or single-precision floating-point values, depending on *type*. These values are grouped into sets of one, two, three, or four values, depending on *format*, to form elements. If *type* is GL_BITMAP, the data is considered as a string of unsigned bytes (and *format* must be GL_COLOR_INDEX). Each data byte is treated as eight 1-bit elements, with bit ordering determined by GL_UNPACK_LSB_FIRST (see **glPixelStore**).

A texture image can have up to four components per texture element, depending on *components*. A one-component texture image uses only the red component of the RGBA color extracted from *pixels*. A two-component image uses the R and A values. A three-component image uses the R, G, and B values. A four-component image uses all of the RGBA components.

Texturing has no effect in color-index mode.

The texture image can be represented by the same data formats as the pixels in a **glDrawPixels** command, except that GL_STENCIL_INDEX and GL_DEPTH_COMPONENT cannot be used. The **glPixelStore** and **glPixelTransfer** modes affect texture images in exactly the way they affect **glDrawPixels**.

A texture image with zero width indicates the null texture. If the null texture is specified for level-of-detail 0, it is as if texturing were disabled.

The following functions retrieve information related to **glTexImageID**:

glGetTexImage
glIsEnabled with argument GL_TEXTURE_1D

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*target* was not GL_TEXTURE_1D.
GL_INVALID_ENUM	*format* was not an accepted *format* constant. Format constants other than GL_STENCIL_INDEX and GL_DEPTH_COMPONENT were accepted.
GL_INVALID_ENUM	*type* was not a *type* constant.
GL_INVALID_ENUM	*type* was GL_BITMAP and *format* was not GL_COLOR_INDEX.
GL_INVALID_VALUE	*level* was less than zero or greater than log (2) *max*, where *max* was the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_VALUE	*components* was not 1, 2, 3, or 4.
GL_INVALID_VALUE	*width* was less than zero or greater than 2 + GL_MAX_TEXTURE_SIZE, or if it could not be represented as 2^*n* + 2(*border*) for some integer value of *n*.
GL_INVALID_VALUE	*border* was not 0 or 1.
GL_INVALID_OPERATION	**glTexImage1D** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, glDrawPixels, glEnd, glFog, glGetTexImage, glIsEnabled, glPixelStore, glPixelTransfer, glTexEnv, glTexGen, glTexImage2D, glTexParameter

glTexImage2D Quick Info

[New - Windows 95, OEM Service Release 2]

The **glTexImage2D** function specifies a two-dimensional texture image.

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

Parameters

target
The target texture. Must be GL_TEXTURE_2D.

level
The level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.

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

width
The width of the texture image. Must be 2^*n* + 2(*border*) for some integer *n*.

height
The height of the texture image. Must be 2^*m* + 2(*border*) for some integer *m*.

border
The width of the border. Must be either 0 or 1.

format
The format of the pixel data. It can assume one of nine symbolic values:

GL_COLOR_INDEX
Each element is a single value, a color index. It is converted to fixed point (with an unspecified number of 0 bits to the right of the binary point), shifted left or right depending on the value and sign of GL_INDEX_SHIFT, and added to GL_INDEX_OFFSET (see **glPixelTransfer**). The resulting index is converted to a set of color components using the GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, and GL_PIXEL_MAP_I_TO_A tables, and clamped to the range [0,1].

GL_RED
Each element is a single red component. It is converted to floating point and assembled into an RGBA element by attaching 0.0 for green and blue, and 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

GL_GREEN
Each element is a single green component. It is converted to floating point and assembled into an RGBA element by attaching 0.0 for red and blue, and 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

GL_BLUE
Each element is a single blue component. It is converted to floating point and assembled into an

RGBA element by attaching 0.0 for red and green, and 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

GL_ALPHA
Each element is a single red component. It is converted to floating point and assembled into an RGBA element by attaching 0.0 for red, green, and blue. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

GL_RGB
Each element is an RGB triple. It is converted to floating point and assembled into an RGBA element by attaching 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

GL_RGBA
Each element is a complete RGBA element. It is converted to floating point. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

GL_BGR_EXT
Each pixel is a group of three components in this order: blue, green, red.

GL_BGR_EXT provides a format that matches the memory layout of Windows device-independent bitmaps (DIBs). Thus your applications can use the same data with Win32 function calls and OpenGL pixel function calls.

GL_BGRA_EXT
Each pixel is a group of four components in this order: blue, green, red, alpha.

GL_BGRA_EXT provides a format that matches the memory layout of Windows device-independent bitmaps (DIBs). Thus your applications can use the same data with Win32 function calls and OpenGL pixel function calls.

GL_LUMINANCE
Each element is a single luminance value. It is converted to floating point, and then assembled into an RGBA element by replicating the luminance value three times for red, green, and blue, and attaching 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

GL_LUMINANCE_ALPHA
Each element is a luminance/alpha pair. It is converted to floating point, and then assembled into an RGBA element by replicating the luminance value three times for red, green, and blue. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

type
The data type of the pixel data. The following symbolic values are accepted: GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, and GL_FLOAT.

pixels
A pointer to the image data in memory.

Remarks

The **glTexImage2D** function specifies a two-dimensional texture image. Texturing maps a portion of a specified *texture image* onto each graphical primitive for which texturing is enabled. Two-dimensional texturing is enabled and disabled using **glEnable** and **glDisable** with argument GL_TEXTURE_2D.

Texture images are defined with **glTexImage2D**. The arguments describe the parameters of the texture image, such as height, width, width of the border, level-of-detail number (see **glTexParameter**), and number of color components provided. The last three arguments describe the way the image is

represented in memory. These arguments are identical to the pixel formats used for **glDrawPixels**.

Data is read from *pixels* as a sequence of signed or unsigned bytes, shorts or longs, or single-precision floating-point values, depending on *type*. These values are grouped into sets of one, two, three, or four values, depending on *format*, to form elements. If *type* is GL_BITMAP, the data is considered as a string of unsigned bytes (and *format* must be GL_COLOR_INDEX). Each data byte is treated as eight 1-bit elements, with bit ordering determined by GL_UNPACK_LSB_FIRST (see **glPixelStore**). Please see **glDrawPixels** for a description of the acceptable values for the *type* parameter.

A texture image can have up to four components per texture element, depending on *components*. A one-component texture image uses only the red component of the RGBA color extracted from *pixels*. A two-component image uses the R and A values. A three-component image uses the R, G, and B values. A four-component image uses all of the RGBA components.

Texturing has no effect in color-index mode.

The texture image can be represented by the same data formats as the pixels in a **glDrawPixels** command, except that GL_STENCIL_INDEX and GL_DEPTH_COMPONENT cannot be used. The **glPixelStore** and **glPixelTransfer** modes affect texture images in exactly the way they affect **glDrawPixels**.

A texture image with zero height or width indicates the null texture. If the null texture is specified for level-of-detail 0, it is as if texturing were disabled.

The following functions retrieve information related to **glTexImage2D**:

glGetTexImage
glIsEnabled with argument GL_TEXTURE_2D

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*target* was not GL_TEXTURE_2D.
GL_INVALID_ENUM	*format* was not an accepted *format* constant. Format constants other than GL_STENCIL_INDEX and GL_DEPTH_COMPONENT were accepted.
GL_INVALID_ENUM	*type* was not a *type* constant.
GL_INVALID_ENUM	*type* was GL_BITMAP and *format* was not GL_COLOR_INDEX.
GL_INVALID_VALUE	*level* was less than zero or greater than $\log_{(2)}$ max, where *max* was the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_VALUE	*components* was not 1, 2, 3, or 4.
GL_INVALID_VALUE	*width* or *height* was less than zero or greater than 2 + GL_MAX_TEXTURE_SIZE, or if either could not be represented as 2^*k* + 2(*border*) for some integer value of *k*.
GL_INVALID_VALUE	*border* was not 0 or 1.

GL_INVALID_OPERATION	**glTexImage2D** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glDrawPixels**, **glEnd**, **glFog**, **glIsEnabled**, **glPixelStore**, **glPixelTransfer**, **glTexEnv**, **glTexGen**, **glTexImage1D**, **glTexParameter**

glTexParameterf, glTexParameteri, glTexParameterfv, glTexParameteriv

[New - Windows 95, OEM Service Release 2]

These functions set texture parameters.

```
void glTexParameterf(
  GLenum target,
  GLenum pname,
  GLfloat param
);

void glTexParameteri(
  GLenum target,
  GLenum pname,
  GLint param
);
```

Parameters

target

The target texture, which must be either GL_TEXTURE_1D or GL_TEXTURE_2D.

pname

The symbolic name of a single-valued texture parameter. The following symbols are accepted in *pname*:

GL_TEXTURE_MIN_FILTER

The texture minifying function is used whenever the pixel being textured maps to an area greater than one texture element. There are six defined minifying functions. Two of them use the nearest one or nearest four texture elements to compute the texture value. The other four use mipmaps.

A mipmap is an ordered set of arrays representing the same image at progressively lower resolutions. If the texture has dimensions 2^*n*x2^*m* there are **max**(*n*, *m*) + 1 mipmaps. The first mipmap is the original texture, with dimensions 2^*n*x2^*m*. Each subsequent mipmap has dimensions 2^*k*-1x2^*l*-1 where 2^*k*x2^*l* are the dimensions of the previous mipmap, until either *k* = 0 or *l* = 0. At that point, subsequent mipmaps have dimension 1x2^*l-1* or 2^*k-1*x1 until the final mipmap, which has dimension 1x1. Mipmaps are defined using **glTexImage1D** or **glTexImage2D** with the level-of-detail argument indicating the order of the mipmaps. Level 0 is the original texture; level bold max(*n, m*) is the final 1x1 mipmap.

GL_TEXTURE_MAG_FILTER

The texture magnification function is used when the pixel being textured maps to an area less than or equal to one texture element. It sets the texture magnification function to either GL_NEAREST or GL_LINEAR.

GL_TEXTURE_WRAP_S

Sets the wrap parameter for texture coordinate *s* to either GL_CLAMP or GL_REPEAT. GL_CLAMP causes *s* coordinates to be clamped to the range [0,1] and is useful for preventing wrapping artifacts when mapping a single image onto an object. GL_REPEAT causes the integer part of the *s* coordinate to be ignored; OpenGL uses only the fractional part, thereby creating a repeating pattern. Border texture elements are accessed only if wrapping is set to GL_CLAMP. Initially, GL_TEXTURE_WRAP_S is set to GL_REPEAT.

GL_TEXTURE_WRAP_T

Sets the wrap parameter for texture coordinate *t* to either GL_CLAMP or GL_REPEAT. See the discussion under GL_TEXTURE_WRAP_S. Initially, GL_TEXTURE_WRAP_T is set to GL_REPEAT.

param

The value of *pname*.

```
void glTexParameterfv(
  GLenum target,
  GLenum pname,
  const GLfloat *params
);

void glTexParameteriv(
  GLenum target,
  GLenum pname,
  const GLint *params
);
```

Parameters

target

The target texture, which must be either GL_TEXTURE_1D or GL_TEXTURE_2D.

pname

The symbolic name of a texture parameter. The following symbols are accepted in *pname*:

GL_TEXTURE_MIN_FILTER

The texture minifying function is used whenever the pixel being textured maps to an area greater than one texture element. There are six defined minifying functions. Two of them use the nearest one or nearest four texture elements to compute the texture value. The other four use mipmaps.

A mipmap is an ordered set of arrays representing the same image at progressively lower resolutions. If the texture has dimensions 2^*n*x2^*m* there are **max**(*n*, *m*) + 1 mipmaps. The first mipmap is the original texture, with dimensions 2^*n*x2^*m*. Each subsequent mipmap has dimensions 2^k_{-1}x2^l_{-1} where 2^*k*x2^*l* are the dimensions of the previous mipmap, until either *k* = 0 or *l* = 0. At that point, subsequent mipmaps have dimension 1x2^*l-1* or 2^*k-1*x1 until the final mipmap, which has dimension 1x1. Mipmaps are defined using **glTexImage1D** or **glTexImage2D** with the level-of-detail argument indicating the order of the mipmaps. Level 0 is the original texture; level bold max(*n, m*) is the final 1x1 mipmap.

GL_TEXTURE_MAG_FILTER

The texture magnification function is used when the pixel being textured maps to an area less than or equal to one texture element. It sets the texture magnification function to either GL_NEAREST or GL_LINEAR.

GL_TEXTURE_WRAP_S

Sets the wrap parameter for texture coordinate *s* to either GL_CLAMP or GL_REPEAT. GL_CLAMP causes *s* coordinates to be clamped to the range [0,1] and is useful for preventing wrapping artifacts when mapping a single image onto an object. GL_REPEAT causes the integer part of the *s* coordinate to be ignored; OpenGL uses only the fractional part, thereby creating a repeating pattern. Border texture elements are accessed only if wrapping is set to GL_CLAMP. Initially, GL_TEXTURE_WRAP_S is set to GL_REPEAT.

GL_TEXTURE_WRAP_T

Sets the wrap parameter for texture coordinate *t* to either GL_CLAMP or GL_REPEAT. See the discussion under GL_TEXTURE_WRAP_S. Initially, GL_TEXTURE_WRAP_T is set to GL_REPEAT.

GL_TEXTURE_BORDER_COLOR

Sets a border color. The *params* parameter contains four values that comprise the RGBA color of the texture border. Integer color components are interpreted linearly such that the most positive integer maps to 1.0, and the most negative integer maps to -1.0. The values are clamped to the range [0,1] when they are specified. Initially, the border color is (0, 0, 0, 0).

params

A pointer to an array where the value or values of *pname* are stored. The *params* parameter supplies a function for minifying the texture as one of the following:

GL_NEAREST
: Returns the value of the texture element that is nearest (in Manhattan distance) to the center of the pixel being textured.

GL_LINEAR
: Returns the weighted average of the four texture elements that are closest to the center of the pixel being textured. These can include border texture elements, depending on the values of GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T, and on the exact mapping. GL_NEAREST is generally faster than GL_LINEAR, but it can produce textured images with sharper edges because the transition between texture elements is not as smooth. The default value of GL_TEXTURE_MAG_FILTER is GL_LINEAR.

GL_NEAREST_MIPMAP_NEAREST
: Chooses the mipmap that most closely matches the size of the pixel being textured and uses the GL_NEAREST criterion (the texture element nearest to the center of the pixel) to produce a texture value.

GL_LINEAR_MIPMAP_NEAREST
: Chooses the mipmap that most closely matches the size of the pixel being textured and uses the GL_LINEAR criterion (a weighted average of the four texture elements that are closest to the center of the pixel) to produce a texture value.

GL_NEAREST_MIPMAP_LINEAR
: Chooses the two mipmaps that most closely match the size of the pixel being textured and uses the GL_NEAREST criterion (the texture element nearest to the center of the pixel) to produce a texture value from each mipmap. The final texture value is a weighted average of those two values.

GL_LINEAR_MIPMAP_LINEAR
: Chooses the two mipmaps that most closely match the size of the pixel being textured and uses the GL_LINEAR criterion (a weighted average of the four texture elements that are closest to the center of the pixel) to produce a texture value from each mipmap. The final texture value is a weighted average of those two values.

Remarks

Texture mapping is a technique that applies an image onto an object's surface as if the image were a decal or cellophane shrink-wrap. The image is created in texture space, with an (*s, t*) coordinate system. A texture is a one- or two-dimensional image and a set of parameters that determine how samples are derived from the image.

The **glTexParameter** function assigns the value or values in *params* to the texture parameter specified as *pname*. The *target* parameter defines the target texture, either GL_TEXTURE_1D or GL_TEXTURE_2D.

As more texture elements are sampled in the minification process, fewer aliasing artifacts will be apparent. While the GL_NEAREST and GL_LINEAR minification functions can be faster than the other four, they sample only one or four texture elements to determine the texture value of the pixel being rendered and can produce moire patterns or ragged transitions. The default value of GL_TEXTURE_MIN_FILTER is GL_NEAREST_MIPMAP_LINEAR.

Suppose texturing is enabled (by calling **glEnable** with argument GL_TEXTURE_1D or GL_TEXTURE_2D) and GL_TEXTURE_MIN_FILTER is set to one of the functions that requires a mipmap. If either the dimensions of the texture images currently defined (with previous calls to **glTexImage1D** or **glTexImage2D**) do not follow the proper sequence for mipmaps, or there are fewer texture images defined than are needed, or the set of texture images have differing numbers of texture components, then it is as if texture mapping were disabled.

Linear filtering accesses the four nearest texture elements only in 2-D textures. In 1-D textures, linear filtering accesses the two nearest texture elements.

The following function retrieves information related to **glTexParameterf**, **glTexParameteri**, **glTexParameterfv**, and **glTexParameteriv**:

glGetTexParameter

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*target* or *pname* was not one of the accepted defined values, or when *params* should have had a defined constant value (based on the value of *pname*) and did not.
GL_INVALID_OPERATION	**glTexParameter** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glGetTexParameter**, **glTexEnv**, **glTexGen**, **glTexImage1D**, **glTexImage2D**

glTexSubImage1D

[New - Windows 95, OEM Service Release 2]

The **glTexSubImage1D** function specifies a portion of an existing one-dimensional texture image. You cannot define a new texture with **glTexSubImage1D**.

```
void glTexSubImage1D(
  GLenum target,
  GLint level,
  GLint xoffset,
  GLsizei width,
  GLenum format,
  GLenum type,
  const GLvoid *pixels
);
```

Parameters

target
The target texture. Must be GL_TEXTURE_1D.

level
The level-of-detail number. Level 0 is the base image. Level *n* is the *n*th mipmap reduction image.

xoffset
A texel offset in the *x* direction within the texture array.

width
The width of the texture sub-image.

format
The format of the pixel data. This parameter can assume one of the following symbolic values:

GL_COLOR_INDEX
Each element is a single value, a color index. It is converted to fixed point format (with an unspecified number of 0 bits to the right of the binary point), shifted left or right depending on the value and sign of GL_INDEX_SHIFT, and added to GL_INDEX_OFFSET (see **glPixelTransfer**). The resulting index is converted to a set of color components using the GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, and GL_PIXEL_MAP_I_TO_A tables, and clamped to the range [0,1].

GL_RED
Each element is a single red component. It is converted to floating point format and assembled into an RGBA element by attaching 0.0 for green and blue, and 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

GL_GREEN
Each element is a single green component. It is converted to floating point format and assembled into an RGBA element by attaching 0.0 for red and blue, and 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

GL_BLUE
Each element is a single blue component. It is converted to floating point format and assembled into an RGBA element by attaching 0.0 for red and green, and 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

GL_ALPHA
Each element is a single alpha component. It is converted to floating point format and assembled into an RGBA element by attaching 0.0 for red, green, and blue. Each component is then multiplied

by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

GL_RGB
Each element is an RGB triple. It is converted to floating point and assembled into an RGBA element by attaching 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

GL_RGBA
Each element is a complete RGBA element. It is converted to floating point format. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

GL_LUMINANCE
Each element is a single luminance value. It is converted to floating point format, and then assembled into an RGBA element by replicating the luminance value three times for red, green, and blue, and attaching 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

GL_LUMINANCE_ALPHA
Each element is a luminance/alpha pair. It is converted to floating point format, and then assembled into an RGBA element by replicating the luminance value three times for red, green, and blue. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

type
The data type of the pixel data. The following symbolic values are accepted: GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, and GL_FLOAT.

pixels
A pointer to the image data in memory.

Remarks

One-dimensional texturing for a primitive is enabled using **glEnable** and **glDisable** with the argument GL_TEXTURE_1D. During texturing, part of a specified texture image is mapped onto each enabled primitive. You use the **glTexSubImage1D** function to specify a contiguous sub-image of an existing one-dimensional texture image for texturing.

The texels referenced by *pixels* replace a region of the existing texture array with *x* indexes of *xoffset* and *xoffset* + (*width* - 1) inclusive. This region cannot include any texels outside the range of the originally specified texture array.

Specifying a sub-image with a *width* of zero has no effect and does not generate an error.

Texturing has no effect in color-index mode.

In general, texture images can be represented by the same data formats as the pixels in a **glDrawPixels** command, except that GL_STENCIL_INDEX and GL_DEPTH_COMPONENT cannot be used. The **glPixelStore** and **glPixelTransfer** modes affect texture images in exactly the way they affect **glDrawPixels**.

The following functions retrieve information related to **glTexSubImage1D**:

glGetTexImage
glIsEnabled with argument GL_TEXTURE_1D

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*target* was not GL_TEXTURE_1D.
GL_INVALID_ENUM	*format* was not an accepted constant.
GL_INVALID_ENUM	*type* was not a constant.
GL_INVALID_ENUM	*type* was GL_BITMAP and *format* was not GL_COLOR_INDEX.
GL_INVALID_VALUE	*level* was less than zero or greater than $\log_{(2)} max$, where *max* was the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_VALUE	*xoffset* was less than -*b*, or *xoffset* + *width* was greater than *w* - *b*, where *w* is the GL_TEXTURE_WIDTH, and *b* is the width of the GL_TEXTURE_BORDER of the texture image being modified. Note that *w* includes twice the border width.
GL_INVALID_VALUE	*width* was less than -*b*, where *b* is the border width of the texture array.
GL_INVALID_VALUE	*border* was not zero or 1.
GL_INVALID_OPERATION	The texture array was not defined by a previous **glTexImage1D** operation.
GL_INVALID_OPERATION	**glTexSubImage1D** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glDrawPixels, **glEnable**, **glFog**, **glGetTexImage**, **glIsEnabled**, **glPixelStore**, **glPixelTransfer**, **glTexEnv**, **glTexGen**, **glTexImage1D**, **glTexParameter**

glTexSubImage2D

[New - Windows 95, OEM Service Release 2]

The **glTexSubImage2D** function specifies a portion of an existing one-dimensional texture image. You cannot define a new texture with **glTexSubImage2D**.

```
void glTexSubImage2D(
  GLenum target,
  GLint level,
  GLint xoffset,
  GLint yoffset,
  GLsizei width,
  GLsizei height,
  GLenum format,
  GLenum type,
  const GLvoid *pixels
);
```

Parameters

target
The target texture. Must be GL_TEXTURE_2D.

level
The level-of-detail number. Level 0 is the base image. Level *n* is the *n*th mipmap reduction image.

xoffset
A texel offset in the *x* direction within the texture array.

yoffset
A texel offset in the *y* direction within the texture array.

width
The width of the texture sub-image.

height
The height of the texture sub-image.

format
The format of the pixel data. It can assume one of the following symbolic values:

GL_COLOR_INDEX
Each element is a single value, a color index. It is converted to fixed point format (with an unspecified number of 0 bits to the right of the binary point), shifted left or right depending on the value and sign of GL_INDEX_SHIFT, and added to GL_INDEX_OFFSET (see **glPixelTransfer**). The resulting index is converted to a set of color components using the GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, and GL_PIXEL_MAP_I_TO_A tables, and clamped to the range [0,1].

GL_RED
Each element is a single red component. It is converted to floating point format and assembled into an RGBA element by attaching 0.0 for green and blue, and 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

GL_GREEN
Each element is a single green component. It is converted to floating point format and assembled into an RGBA element by attaching 0.0 for red and blue, and 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

GL_BLUE
Each element is a single blue component. It is converted to floating point format and assembled

into an RGBA element by attaching 0.0 for red and green, and 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

GL_ALPHA
Each element is a single alpha component. It is converted to floating point format and assembled into an RGBA element by attaching 0.0 for red, green, and blue. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

GL_RGB
Each element is an RGB triple. It is converted to floating point format and assembled into an RGBA element by attaching 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

GL_RGBA
Each element is a complete RGBA element. It is converted to floating point. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

GL_LUMINANCE
Each element is a single luminance value. It is converted to floating point format, and then assembled into an RGBA element by replicating the luminance value three times for red, green, and blue, and attaching 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

GL_LUMINANCE_ALPHA
Each element is a luminance/alpha pair. It is converted to floating point format, and then assembled into an RGBA element by replicating the luminance value three times for red, green, and blue. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

type
The data type of the pixel data. The following symbolic values are accepted: GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, and GL_FLOAT.

pixels
A pointer to the image data in memory.

Remarks

Two-dimensional texturing for a primitive is enabled using **glEnable** and **glDisable** with the argument GL_TEXTURE_2D. During texturing, part of a specified texture image is mapped onto each enabled primitive. You use the **glTexSubImage2D** function to specify a contiguous sub-image of an existing one-dimensional texture image for texturing.

The texels referenced by *pixels* replace a region of the existing texture array with *x* indexes of *xoffset* and *xoffset* + (*width* - 1) inclusive and *y* indexes of *yoffset* and *Yoffset* + (*height* - 1) inclusive. This region cannot include any texels outside the range of the originally specified texture array.

Specifying a sub-image with a *width* of zero has no effect and does not generate an error.

Texturing has no effect in color-index mode.

In general, texture images can be represented by the same data formats as the pixels in a **glDrawPixels** command, except that GL_STENCIL_INDEX and GL_DEPTH_COMPONENT cannot be used. The **glPixelStore** and **glPixelTransfer** modes affect texture images in exactly the way they affect **glDrawPixels**.

The following functions retrieve information related to **glTexSubImage2D**:

glGetTexImage
glIsEnabled with argument GL_TEXTURE_2D

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_ENUM	*target* was not GL_TEXTURE_2D.
GL_INVALID_ENUM	*format* was not an accepted constant.
GL_INVALID_ENUM	*type* was not an accepted constant.
GL_INVALID_ENUM	*type* was GL_BITMAP and *format* was not GL_COLOR_INDEX.
GL_INVALID_VALUE	*level* was less than zero or greater than $\log_{(2)} max$, where *max* was the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_VALUE	*xoffset* was less than -*b*; or *xoffset* + *width* was greater than *w* - *b*; or *yoffset* was less than -*b*; or *yoffset* + *height* was greater than *h* - *b*, where *w* is the GL_TEXTURE_WIDTH, h is the GL_TEXTURE_HEIGHT, and *b* is the width of the GL_TEXTURE_BORDER of the texture image being modified. Note that *w* and *h* include twice the border width.
GL_INVALID_VALUE	*width* was less than -*b*, where *b* is the border width of the texture array.
GL_INVALID_VALUE	*border* was not zero or 1.
GL_INVALID_OPERATION	The texture array was not defined by a previous **glTexImage2D** operation.
GL_INVALID_OPERATION	**glTexSubImage2D** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glDrawPixels, **glEnable**, **glFog**, **glGetTexImage**, **glIsEnabled**, **glPixelStore**, **glPixelTransfer**, **glTexEnv**, **glTexGen**, **glTexImage2D**, **glTexParameter**

glTranslated, glTranslatef

[New - Windows 95, OEM Service Release 2]

The **glTranslated** and **glTranslatef** functions multiply the current matrix by a translation matrix.

```
void glTranslated(
  GLdouble x,
  GLdouble y,
  GLdouble z
);

void glTranslatef(
  GLfloat x,
  GLfloat y,
  GLfloat z
);
```

Parameters

x, y, z
The *x*, *y*, and *z* coordinates of a translation vector.

Remarks

The **glTranslate** function moves the coordinate system origin to the point specified by (*x*, *y*, *z*). The translation vector is used to compute a 4x4 translation matrix:

{ewc msdncd, EWGraphic, bsd23545 28 /a "SDK.BMP"}

The current matrix (see **glMatrixMode**) is multiplied by this translation matrix, with the product replacing the current matrix. That is, if M is the current matrix and T is the translation matrix, then M is replaced with M•T.

If the matrix mode is either GL_MODELVIEW or GL_PROJECTION, all objects drawn after **glTranslate** is called are translated. Use **glPushMatrix** and **glPopMatrix** to save and restore the untranslated coordinate system.

The following functions retrieve information related to **glTranslated** and **glTranslatef**:

glGet with argument GL_MATRIX_MODE
glGet with argument GL_MODELVIEW_MATRIX
glGet with argument GL_PROJECTION_MATRIX
glGet with argument GL_TEXTURE_MATRIX

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_OPERATION	**glTranslate** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEnd**, **glMatrixMode**, **glMultMatrix**, **glPushMatrix**, **glRotate**, **glScale**

glVertex

[New - Windows 95, OEM Service Release 2]

glVertex2d, glVertex2f, glVertex2i, glVertex2s, glVertex3d, glVertex3f, glVertex3i, glVertex3s, glVertex4d, glVertex4f, glVertex4i, glVertex4s, glVertex2dv, glVertex2fv, glVertex2iv, glVertex2sv, glVertex3dv, glVertex3fv, glVertex3iv, glVertex3sv, glVertex4dv, glVertex4fv, glVertex4iv, glVertex4sv

These functions specify a vertex.

```
void glVertex2d(
  GLdouble x,
  GLdouble y
);

void glVertex2f(
  GLfloat x,
  GLfloat y
);

void glVertex2i(
  GLint x,
  GLint y
);

void glVertex2s(
  GLshort x,
  GLshort y
);

void glVertex3d(
  GLdouble x,
  GLdouble y,
  GLdouble z
);

void glVertex3f(
  GLfloat x,
  GLfloat y,
  GLfloat z
);

void glVertex3i(
  GLint x,
  GLint y,
  GLint z
);

void glVertex3s(
  GLshort x,
  GLshort y,
  GLshort z
);

void glVertex4d(
  GLdouble x,
  GLdouble y,
```

```
  GLdouble z,
  GLdouble w
);

void glVertex4f(
  GLfloat x,
  GLfloat y,
  GLfloat z,
  GLfloat w
);

void glVertex4i(
  GLint x,
  GLint y,
  GLint z,
  GLint w
);

void glVertex4s(
  GLshort x,
  GLshort y,
  GLshort z,
  GLshort w
);
```

Parameters

x, y, z, w
: The *x*, *y*, *z*, and *w* coordinates of a vertex. Not all parameters are present in all forms of the command.

```
void glVertex2dv(
  const GLdouble *v
);

void glVertex2fv(
  const GLfloat *v
);

void glVertex2iv(
  const GLint *v
);

void glVertex2sv(
  const GLshort *v
);

void glVertex3dv(
  const GLdouble *v
);

void glVertex3fv(
  const GLfloat *v
);

void glVertex3iv(
  const GLint *v
);

void glVertex3sv(
```

```
  const GLshort *v
);

void glVertex4dv(
  const GLdouble *v
);

void glVertex4fv(
  const GLfloat *v
);

void glVertex4iv(
  const GLint *v
);

void glVertex4sv(
  const GLshort *v
);
```

Parameters

v

A pointer to an array of two, three, or four elements. The elements of a two-element array are *x* and *y*; of a three-element array, *x*, *y*, and *z*; and of a four-element array, *x*, *y*, *z*, and *w*.

Remarks

The **glVertex** function commands are used within **glBegin**/**glEnd** pairs to specify point, line, and polygon vertices. The current color, normal, and texture coordinates are associated with the vertex when **glVertex** is called.

When only *x* and *y* are specified, *z* defaults to 0.0 and *w* defaults to 1.0. When *x, y,* and *z* are specified, *w* defaults to 1.0.

Invoking **glVertex** outside of a **glBegin**/**glEnd** pair results in undefined behavior.

See Also

glBegin, **glCallList**, **glColor**, **glEdgeFlag**, **glEnd**, **glEvalCoord**, **glIndex**, **glMaterial**, **glNormal**, **glRect**, **glTexCoord**

glVertexPointer

[New - Windows 95, OEM Service Release 2]

The **glVertexPointer** function defines an array of vertex data.

```
void glVertexPointer(
  GLint size,
  GLenum type,
  GLsizei stride,
  GLsizei count,
  const GLvoid *pointer
);
```

Parameters

size
The number of coordinates per vertex. The value of *size* must be 2, 3, or 4.

type
The data type of each coordinate in the array using the following symbolic constants: GL_SHORT, GL_INT, GL_FLOAT, and GL_DOUBLE.

stride
The byte offset between consecutive vertices. When *stride* is zero, the vertices are tightly packed in the array.

count
The number of vertices, counting from the first, that are static.

pointer
A pointer to the first coordinate of the first vertex in the array.

Remarks

The **glVertexPointer** function specifies the location and data of an array of vertex coordinates to use when rendering. The *size* parameter specifies the number of coordinates per vertex. The *type* parameter specifies the data type of each vertex coordinate. The *stride* parameter determines the byte offset from one vertex to the next, enabling the packing of vertices and attributes in a single array or storage in separate arrays. In some implementations, storing the vertices and attributes in a single array can be more efficient than using separate arrays. Starting from the first vertex element, the *count* parameter indicates the total number of static elements. Your application can modify static elements, but once the elements are modified, the application must explicitly specify the array again before using the array for any rendering. Non-static array elements are not accessed until you call **glDrawArrays** or **glArrayElement**.

A vertex array is enabled when you specify the GL_VERTEX_ARRAY constant with **glEnableClientState**. When enabled, **glDrawArrays** and **glArrayElement** use the vertex array. By default, the vertex array is disabled.

You cannot include **glVertexPointer** in display lists.

When you specify a vertex array using **glVertexPointer**, the values of all the function's vertex array parameters are saved in a client-side state and static array elements can be cached. Because the vertex array parameters are client-side state, their values are not saved or restored by **glPushAttrib** and **glPopAttrib**.

Although no error is generated if you call **glVertexPointer** within **glBegin** and **glEnd** pairs, the results are undefined.

The following functions retrieve information related to **glVertexPointer**:

glGet with argument GL_VERTEX_ARRAY_SIZE
glGet with argument GL_VERTEX_ARRAY_STRIDE
glGet with argument GL_VERTEX_ARRAY_COUNT
glGet with argument GL_VERTEX_ARRAY_TYPE
glGetPointerv with argument GL_VERTEX_ARRAY_POINTER
glIsEnabled with argument GL_VERTEX_ARRAY

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_VALUE	*size* was not 2, 3, or 4.
GL_INVALID_ENUM	*type* was not an accepted value.
GL_INVALID_VALUE	*stride* or *count* was negative.

See Also

glArrayElement, **glColorPointer**, **glDrawArrays**, **glEdgeFlagPointer**, **glEnableClientState**, **glGetPointerv**, **glGetString**, **glIndexPointer**, **glIsEnabled**, **glNormalPointer**, **glTexCoordPointer**

glViewport Quick Info

[New - Windows 95, OEM Service Release 2]

The **glViewport** function sets the viewport.

```
void glViewport(
  GLint x,
  GLint y,
  GLsizei width,
  GLsizei height
);
```

Parameters

x, y

The lower-left corner of the viewport rectangle, in pixels. The default is (0,0).

width, height

The width and height, respectively, of the viewport. When an OpenGL context is *first* attached to a window, *width* and *height* are set to the dimensions of that window.

Remarks

The **glViewport** function specifies the affine transformation of *x* and *y* from normalized device coordinates to window coordinates. Let (x_{nd}, y_{nd}) be normalized device coordinates. The window coordinates (x_w, y_w) are then computed as follows:

{ewc msdncd, EWGraphic, bsd23545 29 /a "SDK.BMP"}

Viewport width and height are silently clamped to a range that depends on the implementation. This range is queried by calling **glGet** with argument GL_MAX_VIEWPORT_DIMS.

The following functions retrieve information related to **glViewport**:

glGet with argument GL_VIEWPORT

glGet with argument GL_MAX_VIEWPORT_DIMS

Error Codes

The following are the error codes generated and their conditions.

Error Code	Condition
GL_INVALID_VALUE	Either *width* or *height* was negative.
GL_INVALID_OPERATION	**glViewport** was called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glDepthRange**

Introduction to Porting to OpenGL for Windows NT and Windows 95

OpenGL is designed for compatibility across hardware and operating systems. This design makes it easier for programmers to port OpenGL programs from one system to another. While each operating system has unique requirements, much of the OpenGL code in your current programs can be used as is. To port your OpenGL application to Microsoft® Windows NT® and Windows® 95 operating systems, you'll have to modify your programs to work with the Windows NT and Windows 95 windowing systems.

In general applications are ported to OpenGL for Windows NT and Windows 95 from one of two platforms:

- OpenGL applications developed for the X Window System and the X library (Xlib)
- IRIS GL applications

The following topics describe how to port your applications from each of these platforms. The topics discuss porting OpenGL and window management code only; there is no discussion of other operating system port issues such as reading files, messaging, thread creation, and so on. This porting guide focuses on specific porting issues and assumes that you have an understanding of OpenGL and Windows NT and Windows 95 programming.

Porting X Window System Applications

Like Windows NT and Windows 95, the X Window System is an event-handling, message-based system that uses window controls and menus. The OpenGL code in your X Window System application is probably located in areas that roughly correspond to where it will appear when you port it to Windows NT and Windows 95. Most of your OpenGL code will not change, but you must rewrite any code that is specific to the X Window System. For more information on Win32® application programming interface User and GDI calls, refer to the *Win32 Programmer's Reference*. For more information on the X Window System and UNIX, refer to your X Window System and UNIX operating system documentation.

Quick Info

Use the following general procedure to port your X Window System OpenGL programs to Windows NT and Windows 95

1. Rewrite the X Window System specific code using equivalent Win32 code. Locate window-creation and event-handling code. The X Window System, Windows NT, and Windows 95 are event-handling, message-based windowing systems, which makes it easier to determine where to make the appropriate changes. (However, especially for large applications, rewriting an application from one operating system to another can be a complex and difficult undertaking.)
2. Locate any code that uses GLX functions. These are the functions you'll translate to their equivalent Win32 functions.
3. Translate GLX pixel format functions and Visual/Drawable functions to appropriate Win32/OpenGL pixel format and device context functions.
4. Translate GLX rendering context functions to Win32/OpenGL rendering context functions.
5. Translate GLX Pixmap functions to equivalent Win32 functions.
6. Translate GLX framebuffer and other GLX functions to the appropriate Win32 functions.

Translating the GLX Library

OpenGL X Window System programs use the OpenGL Extension with the X Window System (GLX) library. The library is a set of functions and routines that initialize the pixel format, control rendering, and perform other OpenGL specific tasks. It connects the OpenGL library to the X Window System by managing window handles and rendering contexts. You must translate these functions to their equivalent Windows NT and Windows 95 functions. The following table lists the X Window System GLX functions and their equivalent Win32 functions.

GLX/Xlib Function	Win32 Function
glXChooseVisual	**ChoosePixelFormat**
glXCopyContext	Not applicable.
glXCreateContext	**wglCreateContext**
glXCreateGLXPixmap	**CreateDIBitmap**/**CreateDIBSection**
glXDestroyContext	**wglDeleteContext**
glXDestroyGLXPixmap	**DeleteObject**
glXGetConfig	**DescribePixelFormat**
glXGetCurrentContext	**wglGetCurrentContext**
glXGetCurrentDrawable	**wglGetCurrentDC**
glXIsDirect	Not applicable.
glXMakeCurrent	**wglMakeCurrent**
glXQueryExtension	**GetVersion**
glXQueryVersion	**GetVersion**
glXSwapBuffers	**SwapBuffers**
glXUseXFont	**wglUseFontBitmaps**
XGetVisualInfo	**GetPixelFormat**
XCreateWindow	**CreateWindow**/**CreateWindowEx** and **GetDC**/**BeginPaint**
XSync	**GdiFlush**
Not applicable.	**SetPixelFormat**

Some GLX functions don't have an equivalent Win32 function. To port these functions to Win32, rewrite your code to achieve the same functionality. For example, **glXWaitGL** has no equivalent Win32 function but you can achieve the same result, executing any pending OpenGL commands, by calling **glFinish**.

The following topics describe how to port GLX functions that set the pixel format, and manage rendering contexts, pixmaps and bitmaps.

Porting Device Contexts and Pixel Formats

Each window in the Microsoft implementation of OpenGL for Windows NT and Windows 95 has its own current pixel format. A pixel format is defined by a **PIXELFORMATDESCRIPTOR** data structure. Because each window has its own pixel format, you obtain a device context, set the pixel format of the device context, and then create an OpenGL rendering context for the device context. The rendering context automatically uses the pixel format of its device context.

The X Window System also uses a data structure, **XVisualInfo**, to specify the properties of pixels in a window. **XVisualInfo** structures contain a **Visual** data structure that describes how color resources are used in a specific screen.

In the X Window System, **XVisualInfo** is used to create a window by setting the window to the pixel format you want. The returned structure is used to create the window and a rendering context. In Windows NT and Windows 95, you first create a window and get a handle to a device context (HDC) of the window. The HDC is then used to set the pixel format for the window. The rendering context uses the pixel format of the window.

The following table compares the X Window System and GLX visual functions with their equivalent Win32 pixel format functions.

X Window/GLX Visual Function	Win32 Pixel Format Function
XVisualInfo* **glXChooseVisual(**Display **dpy*, int *screen*, int **attribList***)**	int **ChoosePixelFormat(**HDC *hdc*, PIXELFORMATDESCRIPTOR **ppfd***)**
int **glXGetConfig(**Display **dpy*, XVisualInfo **vis*, int **attribList*, int **value***)**	int **DescribePixelFormat(**HDC *hdc*, int *iPixelFormat*, UINT *nBytes*, LPPIXELFORMATDESCRIPTOR *ppfd***)**
XVisualInfo* **XGetVisualInfo(**Display **dpy*, long *vinfo_mask*, XVisualInfo **vinfo_templ*, int **nitems***)**	int **GetPixelFormat(**HDC *hdc***)**
The *visual* returned by **glxChooseVisual** is used when a window is created.	BOOL **SetPixelFormat(**HDC *hdc*, int *iPixelFormat*, PIXELFORMATDESCRIPTOR **ppfd***)**

The following sections give examples of pixel format code fragments for an X Window System program, and the same code after it has been ported to Windows NT and Windows 95.

For more information on pixel formats, see Pixel Formats.

GLX Pixel Format Code Sample

The code sample below shows how an X Window System OpenGL program uses GLX visual/pixel formatting functions.

```
/* X globals, defines, and prototypes */
Display *dpy;
Window glwin;
static int attributes[] = {GLX_DEPTH_SIZE, 16, GLX_DOUBLEBUFFER, None};

    /* find an OpenGL-capable Color Index visual with depth buffer */
    vi = glXChooseVisual(dpy, DefaultScreen(dpy), attributes);
    if (vi == NULL) {
        fprintf(stderr, "could not get visual\n");
        exit(1);
    }
```

The Visual can be used to create a window and a rendering context.

Win32 Pixel Format Code Sample

The following code sample shows a function that sets the pixel format using Win32 functions:

```
BOOL bSetupPixelFormat(HDC hdc)
{
    PIXELFORMATDESCRIPTOR pfd, *ppfd;
    int pixelformat;

    ppfd = &pfd;

    ppfd->nSize = sizeof(PIXELFORMATDESCRIPTOR);
    ppfd->nVersion = 1;
    ppfd->dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL |
                        PFD_DOUBLEBUFFER;
    ppfd->dwLayerMask = PFD_MAIN_PLANE;
    ppfd->iPixelType = PFD_TYPE_COLORINDEX;
    ppfd->cColorBits = 8;
    ppfd->cDepthBits = 16;
    ppfd->cAccumBits = 0;
    ppfd->cStencilBits = 0;

    pixelformat = ChoosePixelFormat(hdc, ppfd);

    if ( (pixelformat = ChoosePixelFormat(hdc, ppfd)) == 0 )
    {
        MessageBox(NULL, "ChoosePixelFormat failed", "Error", MB_OK);
        return FALSE;
    }

    if (SetPixelFormat(hdc, pixelformat, ppfd) == FALSE)
    {
        MessageBox(NULL, "SetPixelFormat failed", "Error", MB_OK);
        return FALSE;
    }

    return TRUE;
}
```

Porting Rendering Contexts

The X Window System, Windows NT, and Windows 95 render through rendering contexts. Six GLX functions manage rendering contexts and five of them have an equivalent Win32 function.

The following table lists the GLX rendering functions and their equivalent Win32 functions.

GLX Rendering Context Function	**Win32 Rendering Context Function**
GLXContext **glXCopyContext(**Display **dpy*, GLXContext *src*, GLXContext *dst*, GLuint *mask***)**	Not applicable.
GLXContext **glXCreateContext(**Display **dpy*, XVisualInfo **vis*, GLXContext *shareList*, Bool *direct***)**	HGLRC **wglCreateContext(**HDC *hdc***)**
void **glXDeleteContext (**Display **dpy*, GLXContext *ctx***)**	BOOL **wglDeleteContext(**HGLRC *hglrc***)**
GLXContext **glXGetCurrentContext (***void***)**	HGLRC **wglGetCurrentContext(***VOID***)**
GLXDrawable **glXGetCurrentDrawable(***void***)**	HDC **wglGetCurrentDC(***VOID***)**
Bool **glXMakeCurrent(**Display **dpy*, GLXDrawable *draw*, GLXContext *ctx***)**	BOOL **wglMakeCurrent(**HDC *hdc*, HGLRC *hglrc***)**

Return types and other types have different names in the X Window System than in Windows NT and Windows 95. You can search for occurrences of GLXContext to help find parts of your code that need to be ported.

The following sections compare rendering context code samples in an X Window System program and the same code after it has been ported to Windows NT and Windows 95.

For more information on rendering contexts, see Rendering Contexts.

GLX Rendering Context Code Sample

The following code sample shows how an X Window System OpenGL program uses GLX rendering context functions.

```
Display *dpy;             /* display variable */
XVisualInfo *vi;          /* visual variable */
Window win;               /* window variable */
GLXDrawable drawable;     /* drawable variable */
GLXContext cx, cxTemp;    /* rendering context variables */

/* Code to open a display and get a visual. */

/* Create a GLX context. */
cx = glXCreateContext(dpy, vi, 0, GL_FALSE);
if (!cx) {
    fprintf(stderr, "Cannot create context.\n");
    exit(-1);
}

        .
/* Connect the context to the window. */
glXMakeCurrent(dpy, win, cx);

        .
/* When it's time to destroy the rendering context. . . */
cx = glXGetCurrentContext;
glXDestroyContext(dpy, cx);
```

Win32 Rendering Context Code Sample

The following code sample shows how the GLX rendering context code in the previous section looks when it has been ported to Windows NT and Windows 95 using Win32 functions.

```
HGLRC hRC;           // rendering context variable

/* Create and initialize a window */

       .
/* Window message switch in a window procedure */
case WM_CREATE:      // Message when window is created
{
    HDC hDC, hDCTemp;    // device context handles

    /* Get the handle of the windows device context. */
    hDC = GetDC(hWnd);

    /* Create a rendering context and make it the current context */
    hRC = wglCreateContext(hDC);
    if (!hRC)
    {
        MessageBox(NULL, "Cannot create context.", "Error", MB_OK);
        return FALSE;
    }
    wglMakeCurrent(hDC, hRC);
}
break;

        .
case WM_DESTROYED:   // Message when window is destroyed
{
    HGLRC hRC        // rendering context handle
    HDC   hDC;       // device context handle

    /* Release and free the device context and rendering context. */
    hDC = wglGetCurrentDC;
    hRC = wglGetCurrentContext;

    wglMakeCurrent(NULL, NULL);

    if (hRC)
        wglDeleteContext(hRC);

    if (hDC)
        ReleaseDC(hWnd, hDC);

    PostQuitMessage (0);
}
break;
```

Porting GLX Pixmap Code

The X Window System uses *pixmaps*, which are off-screen virtual drawing surfaces in the form of a three-dimensional array of bits. You can think of a pixmap as a stack of bitmaps: a two-dimensional array of pixels with each pixel having a value from 0 to 2^N-1 where N is the depth of the pixmap.

For OpenGL programs you use the GLX functions, **glXCreateGLXPixmap** and **glXDestroyGLXPixmap**, to create and destroy GLX pixmaps used for off-screen rendering.

Windows NT and Windows 95 use device-independent bitmaps that serve the same function as X Window System pixmaps. Use the standard Win32 bitmap functions to create and destroy bitmaps.

The following table lists the GLX pixmap functions and their equivalent Win32 bitmap functions.

GLX Pixmap and Font Function	**Win32 Bitmap and Font Function**
GLXPixmap **glXCreateGLXPixmap(**Display **dpy*, XVisualInfo **vis*, Pixmap *pixmap***)**	HBITMAP **CreateDIBitmap(**HDC *hdc*, LPBITMAPINFOHEADER *lpbmih*, DWORD *fdwInit*, CONST BYTE **lpbInit*, LPBITMAPINFO *lpbmi*, UINT *fuUsage***)** HBITMAP **CreateDIBSection(**HDC *hdc*, LPBITMAPINFO *lpbmi*, DWORD *fInit*, DWORD *iUsage***)**
void **glXDestroyGLXPixmap(**Display **dpy*, GLXPixmap *pix***)**	BOOL **DeleteObject(**HGDIOBJ *hObject***)**

Porting Other GLX Code

In addition to the Xlib and GLX functions described in the preceding sections, your program probably contains some of the other GLX or Xlib functions listed in Translating the GLX Library. Rewrite your X Window System code to Windows NT and Windows 95 code, substituting the appropriate functions.

A Porting Sample

It's easier to understand how to modify your X Window System OpenGL program for a Windows NT or Windows 95 program if you can compare before and after samples, and you can better see how the translated code is used in the proper context. This section presents an example of an X Window System OpenGL program, and then shows how the program looks after it has been ported to Windows NT and Windows 95. Note that the OpenGL code is the same in both programs.

An X Window System OpenGL Program

The following program is an X Window System OpenGL program with the same OpenGL code used in the AUXEDEMO.C sample program supplied with the Win32 SDK. Compare this program with the Win32 OpenGL program in The Program Ported to Win32.

```
/*
 * Example of an X Window System OpenGL program.
 * OpenGL code is taken from auxdemo.c in the Win32 SDK
 */
#include <GL/glx.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <X11/keysym.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <stdio.h>

/* X globals, defines, and prototypes */
Display *dpy;
Window glwin;
static int attributes[] = {GLX_DEPTH_SIZE, 16, GLX_DOUBLEBUFFER, None};

#define SWAPBUFFERS glXSwapBuffers(dpy, glwin)
#define BLACK_INDEX     0
#define RED_INDEX       1
#define GREEN_INDEX     2
#define BLUE_INDEX      4
#define WIDTH           300
#define HEIGHT          200


/* OpenGL globals, defines, and prototypes */
GLfloat latitude, longitude, latinc, longinc;
GLdouble radius;

#define GLOBE    1
#define CYLINDER 2
#define CONE     3

GLvoid resize(GLsizei, GLsizei);
GLvoid initializeGL(GLsizei, GLsizei);
GLvoid drawScene(GLvoid);
void polarView( GLdouble, GLdouble, GLdouble, GLdouble);

static Bool WaitForMapNotify(Display *d, XEvent *e, char *arg)
{
    if ((e->type == MapNotify) && (e->xmap.window == (Window)arg)) {
    return GL_TRUE;
    }
    return GL_FALSE;
}

void
```

```
main(int argc, char **argv)
{
    XVisualInfo    *vi;
    Colormap        cmap;
    XSetWindowAttributes swa;
    GLXContext      cx;
    XEvent          event;
    GLboolean       needRedraw = GL_FALSE, recalcModelView = GL_TRUE;
    int             dummy;

    dpy = XOpenDisplay(NULL);
    if (dpy == NULL){
        fprintf(stderr, "could not open display\n");
        exit(1);
    }

    if(!glXQueryExtension(dpy, &dummy, &dummy)){
        fprintf(stderr, "could not open display");
        exit(1);
    }

    /* find an OpenGL-capable Color Index visual with depth buffer */
    vi = glXChooseVisual(dpy, DefaultScreen(dpy), attributes);
    if (vi == NULL) {
        fprintf(stderr, "could not get visual\n");
        exit(1);
    }

    /* create an OpenGL rendering context */
    cx = glXCreateContext(dpy, vi,  None, GL_TRUE);
    if (cx == NULL) {
        fprintf(stderr, "could not create rendering context\n");
        exit(1);
    }

    /* create an X colormap since probably not using default visual */
    cmap = XCreateColormap(dpy, RootWindow(dpy, vi->screen),
                           vi->visual, AllocNone);
    swa.colormap = cmap;
    swa.border_pixel = 0;
    swa.event_mask = ExposureMask | KeyPressMask | StructureNotifyMask;
    glwin = XCreateWindow(dpy, RootWindow(dpy, vi->screen), 0, 0, WIDTH,
                      HEIGHT, 0, vi->depth, InputOutput, vi->visual,
                      CWBorderPixel | CWColormap | CWEventMask, &swa);
    XSetStandardProperties(dpy, glwin, "xogl", "xogl", None, argv,
                           argc, NULL);

    glXMakeCurrent(dpy, glwin, cx);

    XMapWindow(dpy, glwin);
    XIfEvent(dpy,  &event,  WaitForMapNotify,  (char *)glwin);

    initializeGL(WIDTH, HEIGHT);
    resize(WIDTH, HEIGHT);
```

```
    /* Animation loop */
    while (1) {
    KeySym key;

    while (XPending(dpy)) {
        XNextEvent(dpy, &event);
        switch (event.type) {
        case KeyPress:
                XLookupString((XKeyEvent *)&event, NULL, 0, &key, NULL);
        switch (key) {
        case XK_Left:
            longinc += 0.5;
            break;
        case XK_Right:
            longinc -= 0.5;
            break;
        case XK_Up:
            latinc += 0.5;
            break;
        case XK_Down:
            latinc -= 0.5;
            break;
        }
        break;
        case ConfigureNotify:
        resize(event.xconfigure.width, event.xconfigure.height);
        break;
        }
    }
    drawScene();
    }
}

/* OpenGL code */

GLvoid resize( GLsizei width, GLsizei height )
{
    GLfloat aspect;

    glViewport( 0, 0, width, height );

    aspect = (GLfloat) width / height;

    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluPerspective( 45.0, aspect, 3.0, 7.0 );
    glMatrixMode( GL_MODELVIEW );
}

GLvoid createObjects()
{
    GLUquadricObj *quadObj;

    glNewList(GLOBE, GL_COMPILE);
        quadObj = gluNewQuadric ();
```

```
        gluQuadricDrawStyle (quadObj, GLU_LINE);
        gluSphere (quadObj, 1.5, 16, 16);
    glEndList();

    glNewList(CONE, GL_COMPILE);
        quadObj = gluNewQuadric ();
        gluQuadricDrawStyle (quadObj, GLU_FILL);
        gluQuadricNormals (quadObj, GLU_SMOOTH);
        gluCylinder(quadObj, 0.3, 0.0, 0.6, 15, 10);
    glEndList();

    glNewList(CYLINDER, GL_COMPILE);
        glPushMatrix ();
        glRotatef ((GLfloat)90.0, (GLfloat)1.0, (GLfloat)0.0, (GLfloat)0.0);
        glTranslatef ((GLfloat)0.0, (GLfloat)0.0, (GLfloat)-1.0);
        quadObj = gluNewQuadric ();
        gluQuadricDrawStyle (quadObj, GLU_FILL);
        gluQuadricNormals (quadObj, GLU_SMOOTH);
        gluCylinder (quadObj, 0.3, 0.3, 0.6, 12, 2);
        glPopMatrix ();
    glEndList();
}

GLvoid initializeGL(GLsizei width, GLsizei height)
{
    GLfloatmaxObjectSize, aspect;
    GLdouble     near_plane, far_plane;

    glClearIndex( (GLfloat)BLACK_INDEX);
    glClearDepth( 1.0 );

    glEnable(GL_DEPTH_TEST);

    glMatrixMode( GL_PROJECTION );
    aspect = (GLfloat) width / height;
    gluPerspective( 45.0, aspect, 3.0, 7.0 );
    glMatrixMode( GL_MODELVIEW );

    near_plane = 3.0;
    far_plane = 7.0;
    maxObjectSize = 3.0F;
    radius = near_plane + maxObjectSize/2.0;

    latitude = 0.0F;
    longitude = 0.0F;
    latinc = 6.0F;
    longinc = 2.5F;

    createObjects();
}

void polarView(GLdouble radius, GLdouble twist, GLdouble latitude,
           GLdouble longitude)
{
    glTranslated(0.0, 0.0, -radius);
```

```
    glRotated(-twist, 0.0, 0.0, 1.0);
    glRotated(-latitude, 1.0, 0.0, 0.0);
    glRotated(longitude, 0.0, 0.0, 1.0);

}

GLvoid drawScene(GLvoid)
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    glPushMatrix();

        latitude += latinc;
        longitude += longinc;

        polarView( radius, 0, latitude, longitude );

        glIndexi(RED_INDEX);
        glCallList(CONE);

        glIndexi(BLUE_INDEX);
        glCallList(GLOBE);

        glIndexi(GREEN_INDEX);
        glPushMatrix();
            glTranslatef(0.8F, -0.65F, 0.0F);
            glRotatef(30.0F, 1.0F, 0.5F, 1.0F);
            glCallList(CYLINDER);
        glPopMatrix();

    glPopMatrix();

    SWAPBUFFERS;
}
```

The Program Ported to Win32

The following program is a Win32 OpenGL program with the same OpenGL code used in the AUXDEMO.C sample program supplied with the Win32 SDK. Compare this program with the X Window System OpenGL program in An X Window System OpenGL Program.

```
/*
 * Example of a Win32 OpenGL program.
 * The OpenGL code is the same as that used in
 * the X Window System sample
 */
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>

/* Windows globals, defines, and prototypes */
CHAR szAppName[]="Win OpenGL";
HWND  ghWnd;
HDC   ghDC;
HGLRC ghRC;

#define SWAPBUFFERS SwapBuffers(ghDC)
#define BLACK_INDEX     0
#define RED_INDEX       13
#define GREEN_INDEX     14
#define BLUE_INDEX      16
#define WIDTH           300
#define HEIGHT          200

LONG WINAPI MainWndProc (HWND, UINT, WPARAM, LPARAM);
BOOL bSetupPixelFormat(HDC);

/* OpenGL globals, defines, and prototypes */
GLfloat latitude, longitude, latinc, longinc;
GLdouble radius;

#define GLOBE    1
#define CYLINDER 2
#define CONE     3

GLvoid resize(GLsizei, GLsizei);
GLvoid initializeGL(GLsizei, GLsizei);
GLvoid drawScene(GLvoid);
void polarView( GLdouble, GLdouble, GLdouble, GLdouble);

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
lpCmdLine, int nCmdShow)
{
    MSG        msg;
    WNDCLASS   wndclass;

    /* Register the frame class */
    wndclass.style         = 0;
    wndclass.lpfnWndProc   = (WNDPROC)MainWndProc;
```

```
    wndclass.cbClsExtra    = 0;
    wndclass.cbWndExtra    = 0;
    wndclass.hInstance     = hInstance;
    wndclass.hIcon         = LoadIcon (hInstance, szAppName);
    wndclass.hCursor       = LoadCursor (NULL,IDC_ARROW);
    wndclass.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
    wndclass.lpszMenuName  = szAppName;
    wndclass.lpszClassName = szAppName;

    if (!RegisterClass (&wndclass) )
        return FALSE;

    /* Create the frame */
    ghWnd = CreateWindow (szAppName,
             "Generic OpenGL Sample",
         WS_OVERLAPPEDWINDOW | WS_CLIPSIBLINGS | WS_CLIPCHILDREN,
             CW_USEDEFAULT,
             CW_USEDEFAULT,
             WIDTH,
             HEIGHT,
             NULL,
             NULL,
             hInstance,
             NULL);

    /* make sure window was created */
    if (!ghWnd)
        return FALSE;

    /* show and update main window */
    ShowWindow (ghWnd, nCmdShow);

    UpdateWindow (ghWnd);

    /* animation loop */
    while (1) {
        /*
         *  Process all pending messages
         */

        while (PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE) == TRUE)
        {
            if (GetMessage(&msg, NULL, 0, 0) )
            {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
            } else {
                return TRUE;
            }
        }
        drawScene();
    }
}

/* main window procedure */
```

```
LONG WINAPI MainWndProc (
    HWND    hWnd,
    UINT    uMsg,
    WPARAM  wParam,
    LPARAM  lParam)
{
    LONG    lRet = 1;
    PAINTSTRUCT    ps;
    RECT rect;

    switch (uMsg) {

    case WM_CREATE:
        ghDC = GetDC(hWnd);
        if (!bSetupPixelFormat(ghDC))
            PostQuitMessage (0);

        ghRC = wglCreateContext(ghDC);
        wglMakeCurrent(ghDC, ghRC);
        GetClientRect(hWnd, &rect);
        initializeGL(rect.right, rect.bottom);
        break;

    case WM_PAINT:
        BeginPaint(hWnd, &ps);
        EndPaint(hWnd, &ps);
        break;

    case WM_SIZE:
        GetClientRect(hWnd, &rect);
        resize(rect.right, rect.bottom);
        break;

    case WM_CLOSE:
        if (ghRC)
            wglDeleteContext(ghRC);
        if (ghDC)
            ReleaseDC(hWnd, ghDC);
        ghRC = 0;
        ghDC = 0;

        DestroyWindow (hWnd);
        break;

    case WM_DESTROY:
        if (ghRC)
            wglDeleteContext(ghRC);
        if (ghDC)
            ReleaseDC(hWnd, ghDC);

        PostQuitMessage (0);
        break;

    case WM_KEYDOWN:
        switch (wParam) {
```

```
        case VK_LEFT:
            longinc += 0.5F;
            break;
        case VK_RIGHT:
            longinc -= 0.5F;
            break;
        case VK_UP:
            latinc += 0.5F;
            break;
        case VK_DOWN:
            latinc -= 0.5F;
            break;
        }

    default:
        lRet = DefWindowProc (hWnd, uMsg, wParam, lParam);
        break;
    }

    return lRet;
}

BOOL bSetupPixelFormat(HDC hdc)
{
    PIXELFORMATDESCRIPTOR pfd, *ppfd;
    int pixelformat;

    ppfd = &pfd;

    ppfd->nSize = sizeof(PIXELFORMATDESCRIPTOR);
    ppfd->nVersion = 1;
    ppfd->dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL |
                        PFD_DOUBLEBUFFER;
    ppfd->dwLayerMask = PFD_MAIN_PLANE;
    ppfd->iPixelType = PFD_TYPE_COLORINDEX;
    ppfd->cColorBits = 8;
    ppfd->cDepthBits = 16;
    ppfd->cAccumBits = 0;
    ppfd->cStencilBits = 0;

    pixelformat = ChoosePixelFormat(hdc, ppfd);

    if ( (pixelformat = ChoosePixelFormat(hdc, ppfd)) == 0 )
    {
        MessageBox(NULL, "ChoosePixelFormat failed", "Error", MB_OK);
        return FALSE;
    }

    if (SetPixelFormat(hdc, pixelformat, ppfd) == FALSE)
    {
        MessageBox(NULL, "SetPixelFormat failed", "Error", MB_OK);
        return FALSE;
    }

    return TRUE;
```

```
}

/* OpenGL code */

GLvoid resize( GLsizei width, GLsizei height )
{
    GLfloat aspect;

    glViewport( 0, 0, width, height );

    aspect = (GLfloat) width / height;

    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluPerspective( 45.0, aspect, 3.0, 7.0 );
    glMatrixMode( GL_MODELVIEW );
}

GLvoid createObjects()
{
    GLUquadricObj *quadObj;

    glNewList(GLOBE, GL_COMPILE);
        quadObj = gluNewQuadric ();
        gluQuadricDrawStyle (quadObj, GLU_LINE);
        gluSphere (quadObj, 1.5, 16, 16);
    glEndList();

    glNewList(CONE, GL_COMPILE);
        quadObj = gluNewQuadric ();
        gluQuadricDrawStyle (quadObj, GLU_FILL);
        gluQuadricNormals (quadObj, GLU_SMOOTH);
        gluCylinder(quadObj, 0.3, 0.0, 0.6, 15, 10);
    glEndList();

    glNewList(CYLINDER, GL_COMPILE);
        glPushMatrix ();
        glRotatef ((GLfloat)90.0, (GLfloat)1.0, (GLfloat)0.0, (GLfloat)0.0);
        glTranslatef ((GLfloat)0.0, (GLfloat)0.0, (GLfloat)-1.0);
        quadObj = gluNewQuadric ();
        gluQuadricDrawStyle (quadObj, GLU_FILL);
        gluQuadricNormals (quadObj, GLU_SMOOTH);
        gluCylinder (quadObj, 0.3, 0.3, 0.6, 12, 2);
        glPopMatrix ();
    glEndList();
}

GLvoid initializeGL(GLsizei width, GLsizei height)
{
    GLfloat     maxObjectSize, aspect;
    GLdouble    near_plane, far_plane;

    glClearIndex( (GLfloat)BLACK_INDEX);
    glClearDepth( 1.0 );
```

```
    glEnable(GL_DEPTH_TEST);

    glMatrixMode( GL_PROJECTION );
    aspect = (GLfloat) width / height;
    gluPerspective( 45.0, aspect, 3.0, 7.0 );
    glMatrixMode( GL_MODELVIEW );

    near_plane = 3.0;
    far_plane = 7.0;
    maxObjectSize = 3.0F;
    radius = near_plane + maxObjectSize/2.0;

    latitude = 0.0F;
    longitude = 0.0F;
    latinc = 6.0F;
    longinc = 2.5F;

    createObjects();
}

void polarView(GLdouble radius, GLdouble twist, GLdouble latitude,
           GLdouble longitude)
{
    glTranslated(0.0, 0.0, -radius);
    glRotated(-twist, 0.0, 0.0, 1.0);
    glRotated(-latitude, 1.0, 0.0, 0.0);
    glRotated(longitude, 0.0, 0.0, 1.0);

}

GLvoid drawScene(GLvoid)
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    glPushMatrix();

        latitude += latinc;
        longitude += longinc;

        polarView( radius, 0, latitude, longitude );

        glIndexi(RED_INDEX);
        glCallList(CONE);

        glIndexi(BLUE_INDEX);
        glCallList(GLOBE);

    glIndexi(GREEN_INDEX);
        glPushMatrix();
            glTranslatef(0.8F, -0.65F, 0.0F);
            glRotatef(30.0F, 1.0F, 0.5F, 1.0F);
            glCallList(CYLINDER);
        glPopMatrix();

    glPopMatrix();
```

```
    SWAPBUFFERS;
}
```

Porting Applications from IRIS GL

This section lists important differences between IRIS GL and OpenGL and describes the basic steps for porting code from IRIS GL to OpenGL. For a complete list of the differences between IRIS GL and Open GL, see IRIS GL and OpenGL Differences.

Porting IRIS GL programs to OpenGL for Windows NT and Windows 95 requires considerably more work than converting OpenGL programs from the X Window System. While IRIS GL programs are designed to run with specific hardware and software, OpenGL was designed for portability among various systems.

The following table lists some of the key differences between OpenGL and IRIS GL programs.

OpenGL Code	IRIS GL Code
Operating system independent; contains no functions for windowing, event handling, buffer allocation/management, and so on.	Dependent on operating system; windowing-system functions are mixed with rendering functions. There is no windows manager in IRIS GL.
Uses a standard, common naming convention. OpenGL functions and defined types begin with a "gl" prefix to prevent conflicts with other libraries.	Does not use a common naming convention for functions and defined types.
Manages state variables (such as color, fog, texture, lighting, and so on) directly and consistently. Does not use tables to load state-variable values.	Uses tables to manage state variables and must bind variables to table values.
Display lists cannot be edited.	Display lists can be edited.
Does not provide a file format for fonts.	Provides functions to handle fonts and text strings and a file format for fonts.
Includes a GL Utility (GLU) library that contains additional functions and routines (such as NURBS and quadratic rendering routines).	Does not support the GLU library.

Quick Info

Use the following general procedure to port your IRIS GL programs to OpenGL

1. Rewrite any code that makes calls to a window manager, window configuration, device, or event, or where you load a color map to equivalent Win32 code. Rewriting an application from one operating system to another can be complex and difficult. This subject is beyond the scope of this section.
2. Locate any code that uses IRIS GL functions and routines. Translate these functions to their equivalent OpenGL functions. For a complete listing of IRIS GL functions and routines and their equivalent OpenGL counterparts, see OpenGL Functions and Their IRIS GL Equivalents.
3. Change IRIS GL code as described in Special IRIS GL Porting Issues.

Special IRIS GL Porting Issues

The following topics describe techniques for porting specific parts of your IRIS GL code to OpenGL code.

Porting greset

OpenGL replaces the IRIS GL function, **greset**, with the functions, **glPushAttrib** and **glPopAttrib**. Use these functions to save and restore groups of state variables. For example,

```
void glPushAttrib( GLbitfield mask );
```

This example takes a bitwise OR of symbolic constants, indicating which groups of state variables to push onto an attribute stack. Each constant refers to a group of state variables. The following table shows the attribute groups with their equivalent symbolic constant names. For a complete list of the OpenGL state variables associated with each constant, see **glPushAttrib**.

Attribute	**Constant**
accumulation buffer clear value	GL_ACCUM_BUFFER_BIT
color buffer	GL_COLOR_BUFFER_BIT
current	GL_CURRENT_BIT
depth buffer	GL_DEPTH_BUFFER_BIT
enable	GL_ENABLE_BIT
evaluators	EGL_VAL_BIT
fog	GL_FOG_BIT
GL_LIST_BASE setting	GL_LIST_BIT
hint variables	GL_HINT_BIT
lighting variables	GL_LIGHTING_BIT
line drawing mode	GL_LINE_BIT
pixel mode variables	GL_PIXEL_MODE_BIT
point variables	GL_POINT_BIT
polygon	GL_POLYGON_BIT
polygon stipple	GL_POLYGON_STIPPLE_BIT
scissor	GL_SCISSOR_BIT
stencil buffer	GL_STENCIL_BUFFER_BIT
texture	GL_TEXTURE_BIT
transform	GL_TRANSFORM_BIT
viewport	GL_VIEWPORT_BIT
–	GL_ALL_ATTRIB_BITS

To restore the values of the state variables to those saved with the last **glPushAttrib**, simply call **glPopAttrib**. The variables you didn't save will remain unchanged. The attribute stack has a finite depth of at least 16.

Porting IRIS GL "Get" Functions

IRIS GL "get" functions take the following form:

```
int getthing();
```

and

```
int getthings( int *a, int *b);
```

Your IRIS GL code probably includes get function calls that look something like:

```
thing = getthing();
if (getthing() == THING) { /* some stuff here */ }
getthings (&a, &b);
```

In OpenGL you use one of the following four types of **glGet** functions in place of equivalent IRIS GL get functions:

- **glGetBooleanv**
- **glGetIntegerv**
- **glGetFloatv**
- **glGetDoublev**

The functions have the following syntax:

```
glGet<Datatype>v( value, *data );
```

where *value* is of type **GLenum** and data is of type **GLdatatype**. When you call **glGet** and it returns a type different from the type expected, the type is converted appropriately. For a complete list of **glGet** parameters, see **glGet**.

Porting Code that Requires a Current Graphics Position

OpenGL does not maintain a current graphics position. IRIS GL functions that depend on the current graphics position, such as **move**, **draw**, and **rmv**, have no equivalents in OpenGL.

Older versions of IRIS GL included drawing commands that relied upon the current graphics position, though their use has been discouraged. You will need to rewrite your code if you relied on the current graphics position in any way, or used any of the following routines:

- **draw** and **move**
- **pmv**, **pdr**, and **pclos**
- **rdr**, **rmv**, **rpdr**, and **rpmv**
- **getgpos**

OpenGL has a concept of raster position that corresponds to IRIS GL's current character position. For more information on raster positioning, see Porting Pixel Operations.

Porting Screen and Buffer Clearing Commands

OpenGL replaces a variety of IRIS GL **clear** functions (such as **zclear**, **aclear**, **sclear**, and so on) with a single function, **glClear**. Specify exactly what you want to clear by passing masks to **glClear**.

Keep the following points in mind when porting screen and buffer commands:

- OpenGL maintains clearing colors separately from drawing colors, with calls like **glClearColor** and **glClearIndex**. Be sure to set the clear color for each buffer before clearing.
- Instead of using one of several differently named clear calls, you now clear several buffers with one call, **glClear**, by **OR**ing together buffer masks. For example, **czclear** is replaced by:

  ```
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT )
  ```

- IRIS GL references the polygon stipple and the color writemask. OpenGL ignores the polygon stipple but references the color writemask. (The **czclear** function ignores both the polygon stipple and the color writemask.)

The following table lists the various IRIS GL clear functions with their equivalent OpenGL functions.

IRIS GL Call	OpenGL Call	Meaning
acbuf(AC_CLEAR)	**glClear(**GL_ACCUM_BUFFER_BIT**)**	Clear the accumulation buffer.
–	**glClearColor**	Set the RGBA clear color.
–	**glClearIndex**	Set the clear-color index.
clear	**glClear(**GL_COLOR_BUFFER_BIT**)**	Clear the color buffer.
–	**glClearDepth**	Specify the clear value for the depth buffer.
zclear	**glClear(**GL_DEPTH_BUFFER_BIT**)**	Clear the depth buffer.
czclear	**glClear(**GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT**)**	Clear the color buffer and the depth buffer.
–	**glClearAccum**	Specify clear values for the accumulation buffer.
–	**glClearStencil**	Specify the clear value for the stencil buffer.
sclear	**glClear(**GL_STENCIL_BUFFER_BIT**)**	Clear the stencil buffer.

When your IRIS GL code uses both **gclear** and **sclear**, you can combine them into a single **glClear** call—this can improve your program's performance.

Porting Matrix and Transformation Functions

IRIS GL and OpenGL handle matrices and transformations in a similar manner. But there are several differences to keep in mind when porting code from IRIS GL:

- In OpenGL you are always in double-matrix mode; there is no single-matrix mode.
- Angles are measured in degrees, instead of tenths of degrees.
- Projection matrix calls, like **glFrustum** and **glOrtho**, now multiply onto the current matrix, instead of being loaded onto the current matrix.
- The OpenGL function, **glRotate**, is very different from **rotate**. You can rotate around any arbitrary axis, instead of being confined to the x-, y-, and z- axes. For example, you can translate:

  ```
  rotate(200*(i+1), 'z');
  ```

  to:

  ```
  glRotate(.1*(200*(i+1), 0.0, 0.0, 1.0);
  ```

  When translating from **rotate** to **glRotate** you switch to degrees from tenths of degrees and replace 'z' with a vector for the z-axis.
- OpenGL has no equivalent to the **polarview** function. You can replace it easily with a translation and three rotations. For example, you can translate:

  ```
  polarview(distance, azimuth, incidence, twist);
  ```

  to:

  ```
  glTranslatef( 0.0, 0.0, -distance);
  glRotatef( -twist * 10.0, 0.0, 0.0, 1.0);
  glRotatef( -incidence * 10.0, 1.0, 0.0, 0.0);
  glRotatef( -azimuth * 10.0, 0.0, 0.0, 1.0);
  ```

The following table lists the OpenGL matrix functions and their equivalent IRIS GL functions.

IRIS GL Function	OpenGL Function	Meaning
mmode	**glMatrixMode**	Set current matrix mode.
—	**glLoadIdentity**	Replace current matrix with the identity matrix.
loadmatrix	**glLoadMatrixf**, **glLoadMatrixd**	Replace current matrix with the specified matrix.
multmatrix	**glMultMatrixf**, **glMultMatrixd**	Post-multiply current matrix with the specified matrix (note that **multmatrix** pre-multiplied).
mapw, **mapw2**	**gluUnProject**	Project world-space coordinates to object space (see also **gluProject**).
ortho	**glOrtho**	Multiply current matrix by an orthographic projection matrix.
ortho2	**gluOrtho2D**	Define a two-dimensional orthographic projection matrix.
perspective	**gluPerspective**	Define a perspective projection matrix.
picksize	**gluPickMatrix**	Define a picking region.

popmatrix	**glPopMatrix**	Pop current matrix stack, replacing the current matrix with the one below it.
pushmatrix	**glPushMatrix**	Push current matrix stack down by one, duplicating the current matrix.
rotate, **rot**	**glRotated**, **glRotatef**	Rotate current coordinate system by the given angle about the vector from the origin through the given point. Note that **rotate** rotated only about the x-, y-, and z-axes.
scale	**glScaled**, **glScalef**	Multiply current matrix by a scaling matrix.
translate	**glTranslatef**, **glTranslated**	Move coordinate-system origin to the point specified, by multiplying the current matrix by a translation matrix.
window	**glFrustum**	Given coordinates for clipping planes, multiply the current matrix by a perspective matrix.

OpenGL has three matrix modes, which are set with **glMatrixMode**. The following table lists the modes available as parameters for **glMatrixMode**.

IRIS GL Matrix Mode	OpenGL Mode	Meaning	Min Stack Depth
MTEXTURE	GL_TEXTURE	Operate on the texture matrix stack.	2
MVIEWING	GL_MODELVIEW	Operate on the model view matrix stack.	32
MPROJECTION	GL_PROJECTION	Operate on the projection matrix stack.	2

Porting MSINGLE Mode Code

OpenGL has no equivalent for **MSINGLE**, single-matrix mode. Though use of this mode has been discouraged, it is the default for IRIS GL. If your IRIS GL program uses the single-matrix mode, you need to rewrite it to use double-matrix mode only. OpenGL is always in double-matrix mode, and is initially in GL_MODELVIEW mode.

Most IRIS GL code in **MSINGLE** mode looks like this:

```
projectionmatrix();
```

where **projectionmatrix** is one of: **ortho**, **ortho2**, **perspective**, or **window**. To port to OpenGL, replace the **MSINGLE**-mode **projectionmatrix** function with:

```
glMatrixMode( GL_PROJECTION );
glLoadMatrix( identity matrix );

/* call one of these functions here: */
/* glFrustrum(), glOrtho(), glOrtho2(), gluPerspective()}; */

glMatrixMode( GL_MODELVIEW );
glLoadMatrix( identity matrix );
```

Porting Functions that Get Matrices and Transformations

The following table lists the IRIS GL functions that get the state of matrices and transformations and their OpenGL equivalents.

IRIS GL Matrix Query	OpenGL glGet Matrix Query	Meaning
getmmode	GL_MATRIX_MODE	Return the current matrix mode.
getmatrix in **MVIEWING** mode	GL_MODELVIEW_MATRIX	Return a copy of the current model-view matrix.
getmatrix in **MPROJECTION** mode	GL_PROJECTION_MATRIX	Return a copy of the current projection matrix.
getmatrix in **MTEXTURE** mode	GL_TEXTURE_MATRIX	Return a copy of the current texture matrix.
Not applicable.	GL_MAX_MODELVIEW_ STACK_DEPTH	Return maximum supported depth of the model-view matrix stack.
Not applicable.	GL_MAX_PROJECTION_ STACK_DEPTH	Return maximum supported depth of the projection matrix stack.
Not applicable.	GL_MAX_TEXTURE_ STACK_DEPTH	Return maximum supported depth of the texture matrix stack.
Not applicable.	GL_MODELVIEW_ STACK_DEPTH	Returns number of matrices on the model view stack.
Not applicable.	GL_PROJECTION_ STACK_DEPTH	Returns number of matrices on the projection stack.
Not applicable.	GL_TEXTURE_STACK_ DEPTH	Returns number of matrices on the texture stack.

Porting Viewports, Screenmasks, and Scrboxes

The following IRIS GL viewport functions have no OpenGL equivalent:

- **reshapeviewport**
- **scrbox**
- **getscrbox**

With the IRIS GL **viewport** function, you specify the x coordinates (in pixels) for the left and right of a viewport rectangle and the y coordinates for the top and bottom. With the OpenGL **glViewport** function, however, you specify the x and y coordinates (in pixels) of the lower-left corner of the viewport rectangle along with its width and height.

The following table lists IRIS GL viewport functions and their equivalent OpenGL functions.

IRIS GL Function	OpenGL Function	Meaning
viewport(left, right, bottom, top**)**	**glViewport(**x, y, width, height**)**	Set the viewport.
popviewport	**glPopAttrib**	Push and pop the stack.
pushviewport	**glPushAttrib(**GL_VIEWPORT_BIT**)**	
getviewport	**glGet(**GL_VIEWPORT**)**	Returns viewport dimensions.

Porting Clipping Planes

OpenGL implements clipping planes similarly to IRIS GL. In addition, in OpenGL you can query clipping planes. The following table lists IRIS GL clipping plane functions and their equivalent OpenGL functions.

IRIS GL Function	OpenGL Function	Meaning
clipplane(i, **CP_ON**, params)	**glEnable(**GL_CLIP_PLANEi**)**	Enable clipping on plane i.
clipplane(i, **CP_DEFINE**, plane**)**	**glClipPlane(**GL_CLIP_PLANEi, plane**)**	Define clipping plane.
–	**glGetClipPlane**	Returns clipping plane equation.
–	**glIsEnabled(**GL_CLIP_PLANEi**)**	Returns true if clip plane i is enabled.
scrmask	**glScissor**	Defines the scissor box.
getscrmask	**glGet(**GL_SCISSOR_BOX**)**	Return the current scissor box.

To turn on the scissor test, call **glEnable** using GL_SCISSOR_BOX as the parameter.

Porting Drawing Functions

The following sections discuss how to port IRIS GL drawing primitives.

The IRIS GL Sphere Library

OpenGL doesn't support the IRIS GL sphere library. You can replace your sphere library calls with quadrics routines from the GLU library. For more information about the GLU library, see the *Open GL Programming Guide* and OpenGL Utility library.

The following table lists the OpenGL quadrics functions.

OpenGL Function	Meaning
gluNewQuadric	Create a new quadric object.
gluDeleteQuadric	Delete a quadric object.
gluQuadricCallback	Associate a callback with a quadric object, for error handling.
gluQuadricNormals	Specify normals: no normals, one per face, or one per vertex.
gluQuadricOrientation	Specify direction of normals: outward or inward.
gluQuadricTexture	Turn texture-coordinate generation on or off.
gluQuadricDrawstyle	Specify drawing style: polygons, lines, points, and so on.
gluSphere	Draw a sphere.
gluCylinder	Draw a cylinder or cone.
gluPartialDisk	Draw an arc.
gluDisk	Draw a circle or disk.

You can use one quadric object for all quadrics you want to render in similar ways. The following code sample uses two quadric objects to draw four quadrics, two of them textured.

```
GLUquadricObj    *texturedQuad, *plainQuad;

texturedQuad = gluNewQuadric(void);
gluQuadricTexture(texturedQuad, GL_TRUE);
gluQuadricOrientation(texturedQuad, GLU_OUTSIDE);
gluQuadricDrawStyle(texturedQuad, GLU_FILL);

plainQuad = gluNewQuadric(void);
gluQuadricDrawStyle(plainQuad, GLU_LINE);

glColor3f (1.0, 1.0, 1.0);

gluSphere(texturedQuad, 5.0, 20, 20);
glTranslatef(10.0, 10.0, 0.0);
gluCylinder(texturedQuad, 2.5, 5, 5, 10, 10);
glTranslatef(10.0, 10.0, 0.0);
gluDisk(plainQuad, 2.0, 5.0, 10, 10);
glTranslatef(10.0, 10.0, 0.0);
gluSphere(plainQuad, 5.0, 20, 20);
```

Porting v Functions

In IRIS GL, you use variations on the **v** function to specify vertices. The equivalent OpenGL function is **glVertex**: Below are examples of **glVertex**.

```
glVertex2[d|f|i|s][v]( x, y );
glVertex3[d|f|i|s][v]( x, y, z);
glVertex4[d|f|i|s][v]( x, y, z, w);
```

The **glVertex** function takes suffixes the same way other OpenGL calls do. The vector versions of the call take arrays of the proper size as arguments. In the 2-D version, z=0 and w=1. In the 3-D version, w=1.

Porting bgn/end Commands

IRIS GL uses the begin/end paradigm but has a different function for each graphics primitive. For example, you probably use **bgnpolygon** and **endpolygon** to draw polygons, and **bgnline** and **endline** to draw lines. In OpenGL, you use the **glBegin**/**glEnd** structure for both. In OpenGL you draw most geometric objects by enclosing a series of functions that specify vertices, normals, textures, and colors between pairs of **glBegin** and **glEnd** calls. For example:

```
void glBegin( GLenum mode) ;
   /* vertex list, colors, normals, textures, materials */
void glEnd( void );
```

The **glBegin** function takes a single parameter that specifies the drawing mode, and thus the primitive. Here's an OpenGL code sample that draws a polygon and then a line:

```
glBegin( GL_POLYGON ) ;
   glVertex2f(20.0, 10.0);
   glVertex2f(10.0, 30.0);
   glVertex2f(20.0, 50.0);
   glVertex2f(40.0, 50.0);
   glVertex2f(50.0, 30.0);
   glVertex2f(40.0, 10.0);
glEnd();
glBegin( GL_LINES ) ;
   glVertex2i(100,100);
   glVertex2i(500,500);
glEnd();
```

With OpenGL, you draw different geometric objects by specifying different parameters for **glBegin**. The following table lists the OpenGL **glBegin** parameters that correspond to equivalent IRIS GL functions.

IRIS GL Function	Value of glBegin Mode	Meaning
bgnpoint	GL_POINTS	Individual points.
bgnline	GL_LINE_STRIP	Series of connected line segments.
bgnclosedline	GL_LINE_LOOP	Series of connected line segments, with a segment added between first and last vertices.
–	GL_LINES	Pairs of vertices interpreted as individual line segments.
bgnpolygon	GL_POLYGON	Boundary of a simple convex polygon.
–	GL_TRIANGLES	Triples of vertices interpreted as triangles.
bgntmesh	GL_TRIANGLE_STRIP	Linked strips of triangles.
–	GL_TRIANGLE_FAN	Linked fans of triangles.
–	GL_QUADS	Quadruples of vertices interpreted as quadrilaterals.
bgnqstrip	GL_QUAD_STRIP	Linked strips of quadrilaterals.

For a detailed discussion of the differences between triangle meshes, strips, and fans, see Porting Triangles.

There is no limit to the number of vertices you can specify between a **glBegin**/**glEnd** pair.

In addition to specifying vertices inside a **glBegin**/**glEnd** pair, you can specify a current normal, current texture coordinates, and a current color. The following table lists the commands valid inside a **glBegin**/**glEnd** pair.

IRIS GL Function	OpenGL Function	Meaning
v2, **v3**, **v4**	**glVertex**	Set vertex coordinates.
RGBcolor, **cpack**	**glColor**	Set current color.
color	**glIndex**	Set current color index.
n3f	**glNormal**	Set normal vector coordinates.
—	**glEvalCoord**	Evaluate enabled one- and two-dimensional maps.
callobj	**glCallList**, **glCallLists**	Execute display list(s).
t2	**glTexCoord**	Set texture coordinates.
—	**glEdgeFlag**	Control drawing edges.
lmbind	**glMaterial**	Set material properties.

Note If you use any OpenGL function other than those listed in the preceding table inside a **glBegin**/**glEnd** pair, you'll get unpredictable results, or possibly an error.

Porting Points

OpenGL has no command to draw a single point. Otherwise, porting point functions is straightforward. The following table lists IRIS GL functions for drawing points and their equivalent OpenGL functions.

IRIS GL Function	OpenGL Function	Meaning
pnt	–	Draw a single point.
bgnpoint, **endpoint**	**glBegin**(GL_POINTS), **glEnd**	Interpret vertices as points.
pntsize	**glPointSize**	Set point size in pixels.
pntsmooth	**glEnable**(GL_POINT_SMOOTH)	Turn on point antialiasing. (For more information on point antialiasing, see Porting Antialiasing Functions.)

For information about related get functions, see **glPointSize**.

Porting Lines

Porting IRIS GL code that draws lines is fairly straightforward, though you should note the differences in the way OpenGL stipples. The following table lists IRIS GL functions for drawing lines and their equivalent OpenGL functions.

IRIS GL Function	OpenGL Function	Meaning
bgnclosedline, **endclosedline**	**glBegin**(GL_LINE_LOOP) **glEnd**	Draw a closed line.
bgnline	**glBegin**(GL_LINE_STRIP)	Draw line segments.
linewidth	**glLineWidth**	Set line width.
getlwidth	**glGet**(GL_LINE_WIDTH)	Return current line width.
deflinestyle, **setlinestyle**	**glLineStipple**	Specify a line stipple pattern.
lsrepeat	factor argument of **glLineStipple**	Set a repeat factor for the line style.
getlstyle	**glGet**(GL_LINE_STIPPLE_PATTERN)	Return line stipple pattern.
getlsrepeat	**glGet**(GL_LINE_STIPPLE_REPEAT)	Return repeat factor.
linesmooth, **smoothline**	**glEnable**(GL_LINE_SMOOTH)	Turn on line antialiasing (For more information on antialiasing, see Porting Antialiasing Functions.)

OpenGL doesn't use tables for line stipples; it maintains only one line-stipple pattern. You can use **glPushAttrib** and **glPopAttrib** to switch between different stipple patterns.

Older IRIS GL line style functions (such as **draw**, **lsbackup**, **getlsbackup**, and so on) are not supported by OpenGL.

For information on drawing antialiased lines, see Porting Antialiasing Functions.

Porting Polygons and Quadrilaterals

Keep the following points in mind when porting polygons and quadrilaterals:

- There is no direct equivalent for **concave(TRUE)**. Instead you can use the tessellation routines in the GLU, described in Tessellating Polygons.
- Polygon modes are set differently.
- These polygon drawing functions have no direct equivalents in OpenGL: the **poly** family of routines; the **polf** family of routines; **pmv**, **pdr**, and **pclos**; **rpmv** and **rpdr**; **splf**; and **spclos**.

If your IRIS GL code uses these functions, you'll have to rewrite the code using **glBegin(**GL_POLYGON**)**.

The following table lists the IRIS GL polygon drawing functions and their equivalent OpenGL functions.

IRIS GL Function	OpenGL Function	Meaning
bgnpolygon **endpolygon**	**glBegin(**GL_POLYGON**)** **glEnd**	Vertices define boundary of a simple convex polygon.
–	**glBegin(**GL_QUADS**)** **glEnd**	Interpret quadruples of vertices as quadrilaterals.
bgnqstrip **endqstrip**	**glBegin(**GL_QUAD_STRIP**)** **glEnd**	Interpret vertices as linked strips of quadrilaterals.
–	**glEdgeFlag**	
polymode	**glPolygonMode**	Set polygon drawing mode.
rect **rectf**	**glRect**	Draw a rectangle.
sbox **sboxf**	–	Draw a screen-aligned rectangle.

Porting Polygon Modes

The OpenGL function **glPolygonMode** lets you specify which side of a polygon (the back or the front) the mode applies to. Its syntax is:

```
void glPolygonMode( GLenum face, GLenum mode );
```

where face is one of:

GL_FRONT	mode which applies to front-facing polygons
GL_BACK	mode which applies to back-facing polygons
GL_FRONT_AND_BACK	mode which applies to both front- and back-facing polygons

The GL_FRONT_AND_BACK mode is equivalent to the IRIS GL **polymode** function. The following table lists IRIS GL polygon modes and their equivalent OpenGL modes.

IRIS GL Mode	OpenGL Mode	Meaning
PYM_POINT	GL_POINT	Draw vertices as points.

PYM_LINE	GL_LINE	Draw boundary edges as line segments.
PYM_FILL	GL_FILL	Draw polygon interior filled.
PYM_HOLLOW	—	Fill only interior pixels at the boundaries.

Porting Polygon Stipples

When porting IRIS GL polygon stipples, keep the following points in mind:

- OpenGL doesn't use tables for polygon stipples; only one stipple pattern is kept. You can use display lists to store different stipple patterns.
- The OpenGL polygon stipple bitmap size is always a 32x32 bit pattern.
- Stipple encoding is affected by **glPixelStore**.

For more information on porting polygon stipples, see Porting Pixel Operations.

The following table lists IRIS GL polygon stipple functions and their equivalent OpenGL functions.

IRIS GL Function	OpenGL Function	Meaning
defpattern	**glPolygonStipple**	Set the stipple pattern.
setpattern	—	OpenGL keeps only one polygon stipple pattern.
getpattern	**glGetPolygonStipple**	Return the stipple bitmap (used to return an index).

In OpenGL, you enable and disable polygon stippling by passing GL_POLYGON_STIPPLE as a parameter for **glEnable** and **glDisable**.

The following OpenGL code sample demonstrates polygon stippling:

```
void display(void)
{
    GLubyte fly[] = {
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
      0x03, 0x80, 0x01, 0xC0, 0x06, 0xC0, 0x03, 0x60,
      0x04, 0x60, 0x06, 0x20, 0x04, 0x30, 0x0C, 0x20,
      0x04, 0x18, 0x18, 0x20, 0x04, 0x0C, 0x30, 0x20,
      0x04, 0x06, 0x60, 0x20, 0x44, 0x03, 0xC0, 0x22,
      0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
      0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
      0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
      0x66, 0x01, 0x80, 0x66, 0x33, 0x01, 0x80, 0xCC,
      0x19, 0x81, 0x81, 0x98, 0x0C, 0xC1, 0x83, 0x30,
      0x07, 0xe1, 0x87, 0xe0, 0x03, 0x3f, 0xfc, 0xc0,
      0x03, 0x31, 0x8c, 0xc0, 0x03, 0x33, 0xcc, 0xc0,
      0x06, 0x64, 0x26, 0x60, 0x0c, 0xcc, 0x33, 0x30,
      0x18, 0xcc, 0x33, 0x18, 0x10, 0xc4, 0x23, 0x08,
      0x10, 0x63, 0xC6, 0x08, 0x10, 0x30, 0x0c, 0x08,
      0x10, 0x18, 0x18, 0x08, 0x10, 0x00, 0x00, 0x08
    };
    GLubyte halftone[] = {
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
```

```
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55
  };

  glClear (GL_COLOR_BUFFER_BIT);

/*  draw all polygons in white*/
    glColor3f (1.0, 1.0, 1.0);

/*  draw one solid, unstippled rectangle,*/
/*  then two stippled rectangles*/
    glRectf (25.0, 25.0, 125.0, 125.0);
    glEnable (GL_POLYGON_STIPPLE);
    glPolygonStipple (fly);
    glRectf (125.0, 25.0, 225.0, 125.0);
    glPolygonStipple (halftone);
    glRectf (225.0, 25.0, 325.0, 125.0);
    glDisable (GL_POLYGON_STIPPLE);

    glFlush ();
}
```

Porting Tessellated Polygons

In IRIS GL, you use **concave(TRUE)** and then **bgnpolygon** to draw concave polygons. The OpenGL GLU includes functions you can use to draw concave polygons.

Quick Info To draw a concave polygon with OpenGL

1. Create a tessellation object.
2. Define callbacks that will be used to process the triangles generated by the tessellator.
3. Specify the concave polygon to be tessellated.

The following table lists the OpenGL functions for drawing tessellated polygons.

OpenGL GLU Function	Meaning
gluNewTess	Create a new tessellation object.
gluDeleteTess	Delete a tessellation object.
gluTessCallback	–
gluBeginPolygon	Begin the polygon specification.
gluTessVertex	Specify a polygon vertex in a contour.
gluNextContour	Indicate that the next series of vertices describe a new contour.

gluEndPolygon	End the polygon specification.

Porting Triangles

You can draw three types of triangles in OpenGL: separate triangles, triangle strips, and triangle fans.

OpenGL has no equivalent for the IRIS GL **swaptmesh** function. You can achieve the same effect using a combination of triangles, triangle strips, and triangle fans.

The following table lists the IRIS GL functions for drawing triangles and their equivalent OpenGL functions.

IRIS GL Function	Equivalent glBegin Parameter	Meaning
—	GL_TRIANGLES	Triples of vertices interpreted as triangles.
bgntmesh **endtmesh**	GL_TRIANGLE_STRIP	Linked strips of triangles.
—	GL_TRIANGLE_FAN	Linked fans of triangles.

Porting Arcs and Circles

With OpenGL, filled arcs and circles are drawn with the same calls as unfilled arcs and circles. The following table lists the IRIS GL arc and circle functions and their equivalent OpenGL (GLU) functions.

IRIS GL Function	OpenGL Function	Meaning
arc **arcf**	**gluPartialDisk**	Draw an arc.
circ **circf**	**gluDisk**	Draw a circle or disk.

You can do some things with OpenGL arcs and circles that you can't do with IRIS GL. OpenGL calls arcs and circles, disks and partial disks respectively.

When porting arcs and circles, keep the following points about OpenGL in mind:

- Angles are measured in degrees, and not in tenths of degrees.
- The start angle is measured from the positive y-axis, and not from the x-axis.
- The sweep angle is now clockwise instead of counterclockwise.

Porting Spheres

When porting spheres to OpenGL, keep the following points in mind:

- You cannot control the type of primitives used to draw the sphere. You can control drawing precision in another way: use the slices and stacks parameters. Slices are longitudinal; stacks are latitudinal.
- Spheres are drawn centered at the origin. Instead of specifying the location, as you do with the IRIS GL **sphdraw** function, precede a call to the GLU **gluSphere** function with a translation.
- The sphere library is not yet available for OpenGL.

The following table lists the IRIS GL functions for drawing spheres and their equivalent GLU functions where available.

IRIS GL Function	GLU Function	Meaning
sphobj	**gluNewQuadric**	Create a new sphere object.
sphfree	**gluDeleteQuadric**	Delete sphere object and free memory used.
sphdraw	**gluSphere**	Draw a sphere.
sphmode	—	Set sphere attributes.
sphrotmatrix	—	Control sphere orientation.
sphgnpolys	—	Return number of polygons in current sphere.

Porting Color, Shading, and Writemask Code

When porting color, shading, and writemask code to OpenGL, keep the following points in mind:

- Though you can set color-map indexes with the OpenGL **glIndex** function, OpenGL does not have a function for loading color-map indexes.
- Color values are normalized to their data type. (For information about color values, see **glColor**).
- There is no simple equivalent for **cpack**.
- You may have to translate code that includes the **c** or **color** functions to **glClearColor** or **glClearIndex** instead of **glColor** or **glIndex**.
- The RGBA writemask applies to each component but not for each bit.
- IRIS GL provides defined color constants: BLACK, BLUE, RED, GREEN, MAGENTA, CYAN, YELLOW, and WHITE. OpenGL does not provide these constants.

Porting Color Calls

The following table lists IRIS GL color functions and their equivalent OpenGL functions.

IRIS GL Function	OpenGL Function	Meaning
c	**glColor**	Set RGB color.
color	**glIndex**	Set the color index.
getcolor	**glGet(**GL_CURRENT_INDEX**)**	Return the current color index.
getmcolor	–	Get a copy of the RGB values for a color map entry.
gRGBcolor	**glGet(**GL_CURRENT_COLOR**)**	Get the current RGB color values.
mapcolor	–	
RGBcolor	**glColor**	Set RGB color.
writemask	**glIndexMask**	Set the color-index mode color mask.
wmpack	**glColorMask**	Set the RGB color mode mask.
RGBwritemask		
getwritemask	**glGet(**GL_COLOR_WRITEMASK**)**	Get the color mask.
	glGet(GL_INDEX_WRITEMASK**)**	
gRGBmask	**glGet(**GL_COLOR_WRITEMASK**)**	Get the color mask.
zwritemask	**glDepthMask**	–

Note Be careful when replacing **zwritemask** with **glDepthMask**; **glDepthMask** takes a Boolean argument, whereas **zwritemask** takes a bit field.

If you want to use multiple color maps, you need to use the appropriate Win32 color map functions. Therefore, **multimap**, **onemap**, **getcmmode**, **setmap**, and **getmap** have no OpenGL equivalents.

Porting Shading Models

Like IRIS GL, OpenGL lets you switch between smooth (Gouraud) shading and flat shading. The following table lists the IRIS GL shading and dithering functions and their equivalent OpenGL functions.

IRIS GL Function	OpenGL Function	Meaning
shademodel(FLAT)	**glShadeModel(**GL_FLAT**)**	Do flat shading.
shademodel(GOURAUD)	**glShadeModel(**GL_SMOOTH**)**	Do smooth shading.
getsm	**glGet(**GL_SHADE_MODEL**)**	Return current shade model.
dither(DT_ON)	**glEnable(**GL_DITHER**)**	Turn dithering on/off.
dither(DT_OFF)	**glDisable(**GL_DITHER**)**	

Porting Pixel Operations

When porting code that involves pixel operations, keep the following points in mind:

- Logical pixel operations are not applied to RGBA color buffers. For more information, see **glLogicOp**.
- In general, IRIS GL uses the format ABGR for pixels, whereas OpenGL uses RGBA. You can change the format with **glPixelStore**.
- When porting **lrectwrite** functions, be careful to note where **lrectwrite** is writing (for example, it could be writing to the depth buffer).

OpenGL gives you some additional flexibility in pixel operations. The following table lists IRIS GL functions for pixel operations and their equivalent OpenGL functions.

IRIS GL Function	OpenGL Function	Meaning
lrectread, **rectread**, **readRGB**	**glReadPixels**	Read a block of pixels from the frame buffer.
lrectwrite, **rectwrite**	**glDrawPixels**	Write a block of pixels to the frame buffer.
rectcopy	**glCopyPixels**	Copy pixels in the frame buffer.
rectzoom	**glPixelZoom**	Specify pixel zoom factors for **glDrawPixels** and **glCopyPixels**.
cmov	**glRasterPos**	Specify raster position for pixel operations.
readsource	**glReadBuffer**	Select a color buffer source for pixels.
pixmode	**glPixelStore**, **glPixelTransfer**	Set pixel storage modes. Set pixel transfer modes.
logicop	**glLogicOp**	Specify a logical operation for pixel writes.
—	**glEnable**(GL_LOGIC_OP)	Turn on pixel logic operations.

For a complete list of possible logical operations, see **glLogicOp**.

This IRIS GL code sample shows a typical pixel write:

```
unsigned long *packedRaster;
..
packedRaster[k] = 0x00000000;
..
lrectwrite(0, 0, xSize, ySize, packedRaster);
```

The preceding code looks like this when translated to OpenGL:

```
glRasterPos2i( 0, 0);
glDrawPixels( xSize + 1, ySize + 1, GL_RGBA, GL_UNSIGNED_BYTE,
              packedRaster);
```

Porting Depth Cueing and Fog Commands

When porting depth-cueing and fog commands, keep the following points in mind:

- The IRIS GL call, **fogvertex**, sets a mode and the parameters affecting that mode. In OpenGL, you call **glFog** once to set the mode, and then again twice or more to set various parameters.
- In OpenGL, depth cueing is not a separate feature. Use linear fog instead of depth cueing. (This section gives an example of how to do this.) The following IRIS GL functions have no direct OpenGL equivalent:

  depthcue

  IRGBrange

  lshaderange

  getdcm
- To adjust fog quality, use **glHint(**GL_FOG_HINT**)**.

The following table lists the IRIS GL functions for managing fog and their equivalent OpenGL functions.

IRIS GL Function	OpenGL Function	Meaning
fogvertex	**glFog**	Set various fog parameters.
fogvertex(FG_ON**)**	**glEnable(**GL_FOG**)**	Turn fog on.
fogvertex(FG_OFF**)**	**glDisable(**GL_FOG**)**	Turn fog off.
depthcue	**glFog(**GL_FOG_MOD, GL_LINEAR**)**	Use linear fog for depth cueing.

The following table lists the parameters you can pass to **glFog**.

Fog Parameter	Meaning	Default
GL_FOG_DENSITY	Fog density.	1.0
GL_FOG_START	Near distance for linear fog.	0.0
GL_FOG_END	Far distance for linear fog.	1.0
GL_FOG_INDEX	Fog color index.	0.0
GL_FOG_COLOR	Fog RGBA color.	(0, 0, 0, 0)
GL_FOG_MODE	Fog mode.	See the following table.

The fog-density parameter of OpenGL differs from the one in IRIS GL. They are related as follows:

- if **fogMode** = EXP2
  openGLfogDensity = (*irisGLfogDensity*) (**sqrt**(- **log**(1 / 255)))
- if **fogMode** = EXP
  openGLfogDensity = (*irisGLfogDensity*) (- **log**(1 / 255))

where **sqrt** is the square root operation, **log** is the natural logarithm, *irisGLfogDensity* is the IRIS GL fog density, and *openGLfogDensity* is the OpenGL fog density.

To switch between calculating fog in per-pixel mode and per-vertex mode, use **glHint(**GL_FOG_HINT, **hintMode)**. Two hint modes are available:

- GL_NICEST per-pixel fog calculation
- GL_FASTEST per-vertex fog calculation

The following table lists the IRIS GL fog modes and their OpenGL equivalents.

IRIS GL Fog Mode	OpenGL Fog Mode	Hint Mode	Meaning
FG_VTX_EXP, FG_PIX_EXP	GL_EXP	GL_FASTEST, GL_NICEST	Heavy fog mode (default).
FG_VTX_EXP2, FG_PIX_EXP2	GL_EXP2	GL_FASTEST, GL_NICEST	Haze mode.
FG_VTX_LIN, FG_PIX_LIN	GL_LINEAR	GL_FASTEST, GL_NICEST	Linear fog mode. (Use for depth cueing.)

The following code example demonstrates depth cueing in OpenGL:

```
/*
 *  depthcue.c
 *  This program draws a wire frame model, which uses
 *  intensity (brightness) to give clues to distance
 *  Fog is used to achieve this effect
 */
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"

/*  Initialize linear fog for depth cueing
 */
void myinit(void)
{
    GLfloat fogColor[4] = {0.0, 0.0, 0.0, 1.0};

    glEnable(GL_FOG);
    glFogi (GL_FOG_MODE, GL_LINEAR);
    glHint (GL_FOG_HINT, GL_NICEST);  /*  per pixel  */
    glFogf (GL_FOG_START, 3.0);
    glFogf (GL_FOG_END, 5.0);
    glFogfv (GL_FOG_COLOR, fogColor);
    glClearColor(0.0, 0.0, 0.0, 1.0);

    glDepthFunc(GL_LEQUAL);
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_FLAT);
}

/*  display() draws an icosahedron
 */
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0);
    auxWireIcosahedron(1.0);
    glFlush();
}
```

```
void myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective (45.0, (GLfloat) w/(GLfloat) h, 3.0, 5.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity ();
    glTranslatef (0.0, 0.0, -4.0); /*move object into view*/
}
/*  Main Loop
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
    auxInitPosition (0, 0, 400, 400);
    auxInitWindow (argv[0]);
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
}
```

Porting Curve and Surface Functions

OpenGL doesn't support equivalents to the IRIS GL functions for curves and surface patches. You'll need to rewrite your code if it includes any of the following calls:

- **defbasis**
- **curvebasis**, **curveprecision**, **crv**, **crvn**, **rcrv**, **rcrvn**, and **curveit**
- **patchbasis**, **patchcurves**, **patchprecision**, **patch**, and **rpatch**

Porting NURBS Objects

OpenGL treats NURBS as objects, similar to the way it treats quadrics: you create a NURBS object and then specify how it should be rendered. The following table lists the OpenGL GLU functions for managing NURBS objects.

OpenGL GLU Function	Meaning
gluNewNurbsRenderer	Create a new NURBS object.
gluDeleteNurbsRenderer	Delete a NURBS object.
gluNurbsCallback	Associate a callback with a NURBS object, for error handling.

When porting IRIS GL NURBS code to OpenGL, keep the following points in mind:

- NURBS control points are floats, not doubles.
- The stride parameter is counted in floats, not bytes.
- If you're using lighting and you're not specifying normals, call **glEnable** with GL_AUTO_NORMAL as the parameter to generate normals automatically.

Porting NURBS Curves

The OpenGL functions for drawing NURBS curves are very similar to the IRIS GL functions. You specify knot sequences and control points using a **gluNurbsCurve** function, which must be contained within a **gluBeginCurve** /**gluEndCurve** pair.

The following table lists the IRIS GL functions for drawing NURBS curves and their equivalent OpenGL functions.

IRIS GL Function	OpenGL Function	Meaning
bgncurve	**gluBeginCurve**	Begin a curve definition.
nurbscurve	**gluNurbsCurve**	Specify curve attributes.
endcurve	**gluEndCurve**	End a curve definition.

Associate position, texture, and color coordinates by presenting each as a separate **gluNurbsCurve** inside the begin/end pair. You can make no more than one call to **gluNurbsCurve** for each piece of color, position, and texture data within a single **gluBeginCurve/gluEndCurve** pair. You must make exactly one call to describe the position of the curve (a GL_MAP1_VERTEX_3 or GL_MAP1_VERTEX_4 description). When you call **gluEndCurve**, the curve is tessellated into line segments and then rendered.

The following table lists IRIS GL and OpenGL NURBS curve types.

IRIS GL Type	OpenGL Type	Meaning
N_V3D	GL_MAP1_VERTEX_3	Polynomial curve.
N_V3DR	GL_MAP1_VERTEX_4	Rational curve.
–	GL_MAP1_TEXTURE_COORD_*	Control points are texture coordinates.
–	GL_MAP1_NORMAL	Control points are normals.

For more information on available evaluator types, see **glMap1**.

Porting Trimming Curves

OpenGL trimming curves are very similar to IRIS GL trimming curves. The following table lists the IRIS GL functions for defining trimming curves and their equivalent OpenGL functions.

IRIS GL Function	OpenGL Function	Meaning
bgntrim	**gluBeginTrim**	Begin trimming-curve definition.
pwlcurve	**gluPwlCurve**	Define a piecewise linear curve.
nurbscurve	**gluNurbsCurve**	Specify trimming-curve attributes.
endtrim	**gluEndTrim**	End trimming-curve definition.

Porting NURBS Surfaces

The following table lists the IRIS GL functions for drawing NURBS surfaces and their equivalent OpenGL functions.

IRIS GL Function	OpenGL Function	Meaning
bgnsurface	**gluBeginSurface**	Begin a surface definition.
nurbssurface	**gluNurbsSurface**	Specify surface attributes.
endsurface	**gluEndSurface**	End a surface definition.

The following table lists IRIS GL parameters for surface types and their equivalent OpenGL parameters.

IRIS GL Type	OpenGL Type	Meaning
N_V3D	GL_MAP2_VERTEX_3	Polynomial curve.
N_V3DR	GL_MAP2_VERTEX_4	Rational curve.
N_C4D	GL_MAP2_COLOR_4	Control points define color surface in (R,G,B,A) form.
N_C4DR	–	–
N_T2D	GL_MAP2_TEXTURE_COORD_2	Control points are texture coordinates.
N_T2DR	GL_MAP2_TEXTURE_COORD_3	Control points are texture coordinates.
–	GL_MAP2_NORMAL	Control points are normals.

For more information on available evaluator types, see **glMap2**.

The following code sample draws a trimmed NURBS surface:

```
/*
 *  trim.c
 *  This program draws a NURBS surface in the shape of a
 *  symmetrical hill, using both a NURBS curve and pwl
 *  (piecewise linear) curve to trim part of the surface
 */
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"

GLfloat ctlpoints[4][4][3];

GLUnurbsObj *theNurb;

/*
 *  Initializes the control points of the surface to
 *  a small hill. The control points range from -3 to
 *  +3 in x, y, and z
 */
void init_surface(void)
{
    int u, v;
```

```
    for (u = 0; u < 4; u++) {
        for (v = 0; v < 4; v++) {
            ctlpoints[u][v][0] = 2.0*((GLfloat)u - 1.5);
            ctlpoints[u][v][1] = 2.0*((GLfloat)v - 1.5);

            if ( (u == 1 || u == 2) && (v == 1 || v == 2))
                ctlpoints[u][v][2] = 3.0;
            else
                ctlpoints[u][v][2] = -3.0;
        }
    }
}

/*  Initialize material property and depth buffer
 */
void myinit(void)
{
    GLfloat mat_diffuse[] = { 0.6, 0.6, 0.6, 1.0 };
    GLfloat mat_specular[] = { 0.9, 0.9, 0.9, 1.0 };
    GLfloat mat_shininess[] = { 128.0 };

    glClearColor (0.0, 0.0, 0.0, 1.0);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glDepthFunc(GL_LEQUAL);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_AUTO_NORMAL);
    glEnable(GL_NORMALIZE);

    init_surface();

    theNurb = gluNewNurbsRenderer();
    gluNurbsProperty(theNurb, GLU_SAMPLING_TOLERANCE, 50.0);
    gluNurbsProperty(theNurb, GLU_DISPLAY_MODE, GLU_FILL);
}

void display(void)
{
    GLfloat knots[8] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
    GLfloat edgePt[5][2] = /* counter clockwise */
    {{0.0, 0.0}, {1.0, 0.0}, {1.0, 1.0}, {0.0, 1.0},
     {0.0, 0.0}};
    GLfloat curvePt[4][2] = /* clockwise */
    {{0.25, 0.5}, {0.25, 0.75}, {0.75, 0.75}, {0.75, 0.5}};
    GLfloat curveKnots[8] =
        {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
    GLfloat pwlPt[4][2] = /* clockwise */
        {{0.75, 0.5}, {0.5, 0.25}, {0.25, 0.5}};

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
```

```
    glRotatef(330.0, 1.,0.,0.);
    glScalef (0.5, 0.5, 0.5);

    gluBeginSurface(theNurb);
    gluNurbsSurface(theNurb,
            8, knots,
            8, knots,
            4 * 3,
            3,
            &ctlpoints[0][0][0],
            4, 4,
            GL_MAP2_VERTEX_3);
    gluBeginTrim (theNurb);
        gluPwlCurve (theNurb, 5, &edgePt[0][0], 2,
                     GLU_MAP1_TRIM_2);
    gluEndTrim (theNurb);
    gluBeginTrim (theNurb);
        gluNurbsCurve (theNurb, 8, curveKnots, 2,
                &curvePt[0][0], 4, GLU_MAP1_TRIM_2);
        gluPwlCurve (theNurb, 3, &pwlPt[0][0], 2,
                     GLU_MAP1_TRIM_2);
    gluEndTrim (theNurb);
    gluEndSurface(theNurb);

    glPopMatrix();
    glFlush();
}

void myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective (45.0, (GLdouble)w/(GLdouble)h, 3.0, 8.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef (0.0, 0.0, -5.0);
}

/*  Main Loop
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow (argv[0]);
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
}
```

Porting Antialiasing Functions

In OpenGL the subpixel mode is always on, consequently the IRIS GL function **subpixel(TRUE)** is not necessary and has no OpenGL equivalent. The following topics describe aspects of porting IRIS GL antialiasing code.

Porting Blending Code

In IRIS GL, when drawing to both front and back buffers, blending is done by reading one of the buffers, blending with that color, and then writing the result to both buffers. In OpenGL, however, each buffer is read in turn, blended, and then written.

The following table lists IRIS GL blending functions and their equivalent OpenGL functions.

IRIS GL Function	OpenGL Function	Meaning
–	**glEnable**(GL_BLEND)	Turn on blending.
blendfunction	**glBlendFunc**	Specify a blend function.

The OpenGL **glBlendFunc** function and the IRIS GL **blendfunction** function are almost identical. The following table lists IRIS GL blend factors and their OpenGL equivalents.

IRIS GL	OpenGL	Notes
BF_ZERO	GL_ZERO	
BF_ONE	GL_ONE	
BF_SA	GL_SRC_ALPHA	
BF_MSA	GL_ONE_MINUS_SRC_ALPHA	
BF_DA	GL_DST_ALPHA	
BF_MDA	GL_ONE_MINUS_DST_ALPHA	
BF_SC	GL_SRC_COLOR	
BF_MSC	GL_ONE_MINUS_SRC_COLOR	Destination only.
BF_DC	GL_DST_COLOR	Source only.
BF_MDC	GL_ONE_MINUS_DST_COLOR	Source only.
BF_MIN_SA_MDA	GL_SRC_ALPHA_SATURATE	

Porting afunction Test Functions

The following table lists the available IRIS GL alpha test functions and their equivalent OpenGL functions.

afunction	glAlphaFunc
AF_NOTEQUAL	GL_NOTEQUAL
AF_ALWAYS	GL_ALWAYS
AF_NEVER	GL_NEVER
AF_LESS	GL_LESS
AF_EQUAL	GL_EQUAL
AF_LEQUAL	GL_LEQUAL
AF_GREATER	GL_GREATER
AF_GEQUAL	GL_GEQUAL

Using Antialiasing Functions

The following table lists IRIS GL antialiasing functions and their equivalent OpenGL functions.

IRIS GL Function	OpenGL Function	Meaning
pntsmooth	**glEnable**(GL_POINT_SMOOTH)	Enable antialiasing of points.
linesmooth	**glEnable**(GL_LINE_SMOOTH)	Enable antialiasing of lines.
polysmooth	**glEnable**(GL_POLYGON_SMOOTH)	Enable antialiasing of polygons.

Use the equivalent **glDisable** calls to turn off antialiasing.

In IRIS GL, you can control the quality of the antialiasing by calling:

```
linesmooth(SML_ON + SML_SMOOTHER);
```

OpenGL provides similar control—use **glHint**:

```
glHint(GL_POINT_SMOOTH_HINT, hintMode);
glHint(GL_LINE_SMOOTH_HINT, hintMode);
glHint(GL_POLYGON_SMOOTH_HINT, hintMode);
```

where *hintMode* is one of the following:

- GL_NICEST (Use the highest quality smoothing.)
- GL_FASTEST (Use the most efficient smoothing.)
- GL_DONT_CARE

IRIS GL also permits end-correction by calling:

```
linesmooth(SML_ON +  SML_END_CORRECT);
```

OpenGL has no equivalent for this function.

Porting Accumulation Buffer Calls

You must allocate your accumulation buffer by requesting the appropriate pixel format with the OpenGL **auxInitDisplayMode** or **ChoosePixelFormat** function. The following table lists IRIS GL functions that affect the accumulation buffer and their equivalent OpenGL functions.

IRIS GL Function	OpenGL Function	Meaning
acsize	**auxInitDisplayMode** or **ChoosePixelFormat**	Specify number of bitplanes per color component in the accumulation buffer.
acbuf	**glAccum**	Operate on the accumulation buffer.
–	**glClearAccum**	Set clear values for accumulation buffer.
acbuf(AC_CLEAR**)**	**glClear(**GL_ACCUM_BUFFER_BIT**)**	Clear the accumulation buffer.

The following table lists the IRIS GL **acbuf** parameters along with the equivalent OpenGL **glAccum** parameters.

IRIS GL Parameter	OpenGL Parameter
AC_ACCUMULATE	GL_ACCUM
AC_CLEAR_ACCUMULATE	GL_LOAD
AC_RETURN	GL_RETURN
AC_MULT	GL_MULT
AC_ADD	GL_ADD

Porting Stencil Plane Calls

In OpenGL, you allocate stencil planes by requesting the appropriate pixel format with the OpenGL **auxInitDisplayMode** or **ChoosePixelFormat**. functions. The following table lists IRIS GL functions that affect the stencil planes and their equivalent OpenGL functions.

IRIS GL Function	OpenGL Function	Meaning
stensize	**ChoosePixelFormat**	–
stencil(TRUE, ...**)**	**glEnable(**GL_STENCIL_TEST**)**	Enable stencil tests.
stencil	**glStencilOp**	Set stencil test actions.
stencil(... func, ...**)**	**glStencilFunc**	Set function and reference value for stencil testing.
swritemask	**glStencilMask**	Specify which stencil bits can be written.
–	**glClearStencil**	Specify the clear value for the stencil buffer.
sclear	**glClear(**GL_STENCIL_BUFFER_BIT**)**	–

Stencil-comparison functions and stencil pass/fail operations are nearly equivalent in OpenGL and IRIS GL. The IRIS GL stencil-function flags are prefaced with SF; the OpenGL flags with GL. IRIS GL pass/fail operation flags are prefaced with ST; the OpenGL flags with GL.

Porting Display Lists

The OpenGL implementation of display lists is similar to the IRIS GL implementation, with two exceptions: in OpenGL you can't edit display lists once you've created them and you can't call functions from within display lists.

Because you can't edit or call functions from within display lists, these IRIS GL functions have no equivalent in OpenGL:

- **editobj**
- **objdelete**, **objinsert**, and **objreplace**
- **maketag**, **gentag**, **istag**, and **deltag**
- **callfunc**

In IRIS GL, you use the **makeobj** and **closeobj** functions to create display lists. In OpenGL, you use **glNewList** and **glEndList**.

The following table lists the IRIS GL display list functions with their equivalent OpenGL functions.

IRIS GL Function	OpenGL Function	Meaning
makeobj	**glNewList**	Create a new display list.
closeobj	**glEndList**	Signal end of display list.
callobj	**glCallList**, **glCallLists**	Execute display list or lists.
isobj	**glIsList**	Test for display list existence.
delobj	**glDeleteLists**	Delete contiguous group of display lists.
genobj	**glGenLists**	Generate the given number of contiguous empty display lists.

Porting the bbox2 Function

There is no OpenGL equivalent for the IRIS GL **bbox2** function.

Quick Info

To port code that contains bbox2 functions

1. Create a new (OpenGL) display list that contains everything in the equivalent IRIS GL display list except the call to **bbox2**.
2. Use appropriate Win32 code to draw a rectangle the same size as the IRIS GL **bbox**.

Porting Edited Display Lists

Although you can't edit OpenGL display lists, you can get similar results by nesting display lists and then destroying and creating new versions of the sublists. For example:

```
glNewList (1, GL_COMPILE);
    glIndexi(MY_RED);
glEndlist();

glNewList(2, GL_COMPILE);
    glScalef(1.2, 1.2, 1.0);
glEndList();

glNewList(3, GL_COMPILE);
    glCallList(1);
    glCallList(2);
glEndList();

glDeleteLists(1, 2);
glNewList(1, GL_COMPILE);
    glIndexi(MY_CYAN);
glEndList();
glNewList(2, GL_COPILE);
    glScalef(0.5, 0.5, 1.0);
glEndList;
```

A Sample Port of a Display List

This topic gives an IRIS GL sample of code that defines three display lists; one of the display lists refers to the others in its definition. Following the IRIS GL sample is a sample of what the code looks like when ported to OpenGL.

IRIS GL Sample Display List Code

```
makeobj(10);  // 10 object
    cpack(0x0000FF);
    recti(164, 33, 364, 600);   // Hollow rectangle
closeobj();

makeobj(20);  // 20 object
    cpack(0xFFFF00);
    circle(0, 0, 25);           // Unfilled circle
    recti(100, 100, 200, 200);  // Filled rectangle
closeobj();

makeobj(30);  // 30 object
    callobj(10);
    cpack(0xFFFFFF);
    recti(400, 100, 500, 300);  // Draw filled rectangle
    callobj(20);
closeobj();

// Now draw by calling the lists
call(30);
```

OpenGL Sample Display List Code

Here is the preceding IRIS GL code translated to OpenGL:

```
glNewList(10, GL_COMPILE);            // List #10
    glColor3f(1, 0, 0);
    glRecti(164, 33, 364, 600);
glEndList();

glNewList(20, GL_COMPILE);            //List #20
    glColor3f(1, 1, 0);               // Set color to YELLOW
    glPolygonMode(GL_BOTH, GL_LINE);  // Unfilled mode
    glBegin(GL_POLYGON);  // Use polygon to approximate a circle
        for(i=0; i<100; i++) {
            cosine = 25 * cos(i * 2 * PI/100.0);
            sine =   25 * sin(i * 2 * PI/100.0);
            glVertex2f(cosine, sine);
        }
    glEnd();
    glBegin(GL_QUADS);
        glColorf(0, 1, 1);            // Set color to CYAN
        glVertex2i(100, 100);
        glVertex2i(100, 200);
        glVertex2i(200, 200);
        glVertex2i(100, 200);
    glEnd();
glEndList();
```

```
glNewList(30, GL_COMPILE);            // List #30
    glCallList(10);
        glColorf(1, 1, 1);            // Set color to WHITE
        glRecti(400, 100, 500, 300);
    glCallList(20);
glEndList();

// Execute the display lists
glCallList(30);
```

Porting Defs, Binds, and Sets

OpenGL doesn't have tables of stored definitions; you can't define lighting models, material, textures, line styles, or patterns as separate objects as you can in IRIS GL. Thus OpenGL has no direct equivalents to the following IRIS GL functions:

- **lmdef** and **lmbind**
- **tevdef** and **tevbind**
- **textdef** and **textbind**
- **definestyle** and **setstyle**
- **defpattern** and **setpattern**

You can use OpenGL display lists to mimic the IRIS GL def/bind mechanism. For example, here is a material definition in IRIS GL:

```
float mat() = {
    AMBIENT, .1, .1, .1,
    DIFFUSE, 0, .369, .165,
    SPECULAR, .5, .5, .5,
    SHININESS, 10,
    LMNULL
};
lmdef(DEFMATERIAL, 1, 0, mat);
lmbind(MATERIAL, 1);
```

The following OpenGL code sample defines the same material in a display list that is referred to by the list number defined by **MYMATERIAL**.

```
#define MYMATERIAL 10

GLfloat        mat_amb[] = {.1, .1, .1, 1.0};
GLfloat        mat_dif[] = {0, .369, .165, 1.0};
GLfloat        mat_spec[] = {.5, .5, .5, 1.0};

glNewList(MYMATERIAL, GL_COMPILE);
    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_amb);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_dif);
    glMaterialfv(GL_FRONT, GL_SHININESS, 10);
glEndList();

glCallList( MYMATERIAL );
```

Porting Lighting and Materials Functions

OpenGL functions for lighting and materials differ substantially from the IRIS GL functions. Unlike IRIS GL, OpenGL has separate functions for setting lights, light models and materials.

Keep the following points in mind when porting lighting and materials functions:

- OpenGL has no table of stored definitions. You can use display lists to mimic the IRIS GL def/bind mechanism. For more information on defs and binds, see Porting Defs, Binds, and Sets.
- With OpenGL, attenuation is associated with each light source, rather than the overall lighting model.
- Diffuse and specular components are separated in OpenGL light sources.
- OpenGL light sources have an alpha component. When porting your IRIS GL code, set this alpha component to 1.0, indicating 100 percent opaque. The alpha values are then determined by the alpha component of your materials only, so the objects in your scene will look the same as they did in IRIS GL.

The following table lists IRIS GL lighting and materials functions and their equivalent OpenGL functions.

IRIS GL Function	OpenGL Function	Meaning
lmdef(DEFLIGHT, ...**)**	**glLight**	Define a light source.
lmdef(DEFLMODEL, ...**)**	**glLightModel**	Define a lighting model.
lmbind	**glEnable(**GL_LIGHT*i***)**	Enable light *i*.
lmbind	**glEnable(**GL_LIGHTING**)**	Enable lighting.
lmdef(DEFMATERIAL, ...**)**	**glMaterial**	Define a material.
lmcolor	**glColorMaterial**	Change the effect of color commands while lighting is active.
–	**glGetMaterial**	Get material parameters.

The following table lists various IRIS GL material parameters and their equivalent OpenGL parameters.

lmdef Index	glMaterial Parameter	Default	Meaning
ALPHA	GL_DIFFUSE	–	The fourth value in the GL_DIFFUSE parameter specifies the alpha value.
AMBIENT	GL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	Ambient color.
DIFFUSE	GL_DIFFUSE	(0.8, 0.8, 0.8, 1.0)	Diffuse color.
SPECULAR	GL_SPECULAR	(0.0, 0.0, 0.0, 1.0)	Emissive color.
SHININESS	GL_SHININESS	0.0	Specular exponent.
	GL_AMBIENT_ AND_DIFFUSE		Equivalent to calling **glMaterial** twice with the same values.
COLORINDEXES	GL_COLOR_ INDEXES	–	Color indexes for ambient, diffuse, and specular lighting.

When the first parameter of **lmdef** is DEFMODEL, the equivalent OpenGL translation is the function **glLightModel**. The exception is when the parameter following DEFMODEL is ATTENUATION: then the equivalent OpenGL function is **glLight**.

The following table lists the equivalent lighting model parameters for IRIS GL and OpenGL.

lmdef Index	glLightModel Parameter	Default	Meaning
AMBIENT	GL_LIGHT_MODEL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	Ambient color of scene.
ATTENUATION	–	–	See **glLight**.
LOCALVIEWER	GL_LIGHT_MODEL_LOCAL_VIEWER	GL_FALSE	Viewer local (TRUE) or infinite (FALSE).
TWOSIDE	GL_LIGHTMODEL_TWO_SIDE	GL_FALSE	Use two-sided lighting when TRUE.

When the first parameter of **lmdef** is DEFLIGHT, the equivalent OpenGL translation is the **glLight** function.

The following table lists the equivalent lighting parameters for IRIS GL and OpenGL.

lmdef Index	glLight Parameter	Default	Meaning
AMBIENT	GL_AMBIENT	(0.0, 0.0, 0.0, 1.0)	Ambient intensity.
	GL_DIFFUSE	(1.0, 1.0, 1.0, 1.0)	Diffuse intensity.
	GL_SPECULAR	(1.0, 1.0, 1.0, 1.0)	Specular intensity.
LCOLOR	No equivalent.	–	–
POSITION	GL_POSITION	(0.0, 0.0, 1.0, 0.0)	Position of light.
SPOTDIRECTION	GL_SPOT_DIRECTION	(0, 0, -1)	Direction of spotlight.
SPOTLIGHT	GL_SPOT_EXPONENT	0	Intensity distribution.
	GL_SPOT_CUTOFF	180	Maximum spread angle of light source.
DEFMODEL, ATTENUATION	GL_CONSTANT_ATTENUATION	(1, 0, 0)	Attenuation factors.
	GL_LINEAR_ATTENUATION		
	GL_QUADRATIC_ATTENUATION		

Porting Texture Functions

When porting IRIS GL texture functions to OpenGL, keep the following points in mind:

- OpenGL doesn't maintain tables of textures; it uses either 1-D texture and 2-D texture only. To reuse the textures from your IRIS GL code, put them in a display list.
- OpenGL doesn't automatically generate mipmaps. If you're using mipmaps, you must first call the **gluBuild2DMipmaps** function.
- In OpenGL, you use **glEnable** and **glDisable** to turn texturing capabilities on and off.
- In OpenGL, texture size is more strictly regulated than in IRIS GL. The size of an OpenGL texture must be:

  2^n+2b

  where n is an integer and b is:
  - 0, if the texture has no border
  - 1, if the texture has a border pixel (OpenGL textures can have 1-pixel borders.)

The following table lists IRIS GL texture functions and their general OpenGL equivalents.

IRIS GL Function	OpenGL Function	Meaning
textdef2d	**glTexImage2D** **glTexParameter** **gluBuild2DMipmaps**	Specify a 2-D texture image.
textbind	**glTexImage2D** **glTexParameter** **gluBuild2DMipmaps**	Select a texture function.
tevdef	**glTexEnv**	Define a texture-mapping environment.
tevbind	**glTexEnv** **glTexImage1D**	Select a texture environment.
t2	**glTexCoord**	Set the current texture coordinates.
texgen	**glTexGen** **glGetTexParameter** **gluBuild1DMipmaps** **gluBuild2DMipmaps** **gluScaleImage**	Control generation of texture coordinates. Scale an image to an arbitrary size.

For more information on texturing, see the *OpenGL Programming Guide*.

Translating tevdef

The following code example is an IRIS GL texture-environment definition that specifies the TV_DECAL texture-environment parameter:

```
float tevprops[] = {TV_DECAL, TV_NULL};

tevdef(1, 0, tevprops);
```

and the same code translated to OpenGL:

```
glTexEnvfv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
```

The following table lists the IRIS GL texture-environment parameters and their equivalent OpenGL parameters.

IRIS GL Parameter	OpenGL Parameter
TV_MODULATE	GL_MODULATE
TV_DECAL	GL_DECAL
TV_BLEND	GL_BLEND
TV_COLOR	GL_TEXTURE_ENV_COLOR
TV_ALPHA	No direct OpenGL equivalent.
TV_COMPONENT_SELECT	No direct OpenGL equivalent.

For more information about texture-environment parameters, see **glTexEnv**.

Translating texdef

The following code example is an IRIS GL texture definition:

```
float texprops[] = { TX_MINFILTER, TX_POINT,
                         TX_MAGFILTER, TX_POINT,
                         TX_WRAP_S, TX_REPEAT,
                         TX_WRAP_T, TX_REPEAT,
                         TX_NULL };

textdef2d(1, 1, 6, 6, granite_texture, 7, texprops);
```

In the preceding example, **texdef** specifies the TX_POINT filter as both the magnification and the minimizing filter, and TX_REPEAT as the wrapping mechanism. It also specifies the texture image: *granite_texture*.

In OpenGL, **glTexImage** specifies the image and **glTexParameter** sets the property. To translate IRIS GL texture definitions, replace the **textdef** function with **glTexImage** and one or more calls to **glTexParameter**.

The preceding IRIS GL code looks like this when translated to OpenGL:

```
GLfloat nearest[] = {GL_NEAREST};
GLfloat repeat = {GL_REPEAT};
glTexParameterfv( GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, nearest);
glTexParameterfv( GL_TEXTURE_1D, GL_TEXTURE_MAGILTER, nearest);
glTexParameterfv( GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, repeat);
glTexParameterfv( GL_TEXTURE_1D, GL_TEXTURE_WRAP_T, nearest);
glTexImage1D( GL_TEXTURE_1D, 0, 1, 6, 0, GL_RGB,
                  GL_UNSIGNED_SHORT, granite_texture);
```

The following table lists the IRIS GL texture parameters and their equivalent OpenGL parameters.

IRIS GL Texture Parameter	OpenGL Texture Parameter
TX_MINFILTER	GL_TEXTURE_MIN_FILTER
TX_MAGFILTER	GL_TEXTURE_MAG_FILTER
TX_WRAP, TX_WRAP_S	GL_TEXTURE_WRAP_S
TX_WRAP, TX_WRAP_T	GL_TEXTURE_WRAP_T
	GL_TEXTURE_BORDER_COLOR

The following table lists the possible values of the IRIS GL texture parameters and their equivalent OpenGL parameters.

IRIS GL Texture Parameter	OpenGL Texture Parameter
TX_POINT	GL_NEAREST
TX_BILINEAR	GL_LINEAR
TX_MIPMAP_POINT	GL_NEAREST_MIPMAP_NEARE ST
TX_MIPMAP_BILINEAR	GL_LINEAR_MIPMAP_NEAREST
TX_MIPMAP_LINEAR	GL_NEAREST_MIPMAP_LINEAR
TX_TRILINEAR	GL_LINEAR_MIPMAP_LINEAR

Translating texgen

The IRIS GL function **texgen** is translated to **glTexGen** for OpenGL.

With IRIS GL, you call **texgen** twice: once to set the mode and plane equation simultaneously, and once to enable texture-coordinate generation. For example:

```
texgen(TX_S, TG_LINEAR, planeParams);
texgen(TX_S, TG_ON, NULL);
```

With OpenGL, you make three calls: two to **glTexGen** (once to set the mode and once to set the plane equation), and one to **glEnable**. For example, the OpenGL equivalent to the IRIS GL code above is:

```
glTexGen(GL_S, GLTEXTURE_GEN_MODE, modeName);
glTextGen(GL_S, GL_OBJECT_PLANE, planeParameters);
glEnable(GL_TEXTURE_GEN_S);
```

The following table lists the IRIS GL texture-coordinate names and their OpenGL equivalents.

IRIS GL Texture Coordinate	OpenGL Texture Coordinate	glEnable Argument
TX_S	GL_S	GL_TEXTURE_GEN_S
TX_T	GL_T	GL_TEXTURE_GEN_T
TX_R	GL_R	GL_TEXTURE_GEN_R
TX_Q	GL_Q	GL_TEXTURE_GEN_Q

The following table lists the IRIS GL texture-generation modes and their equivalent OpenGL texture modes and plane names.

IRIS GL Texture Mode	OpenGL Texture Mode	OpenGL Plane Name
TG_LINEAR	GL_OBJECT_LINEAR	GL_OBJECT_PLANE
TG_CONTOUR	GL_EYE_LINEAR	GL_EYE_PLANE
TG_SPHEREMAP	GL_SPHERE_MAP	

Porting Picking Functions

All IRIS GL picking functions have OpenGL equivalents, with the exception of **clearhitcode**. The following table lists the IRIS GL picking functions and their equivalent OpenGL functions.

IRIS GL Function	OpenGL Function	Meaning
clearhitcode	Not supported.	Clears global variable and hitcode.
pick **select**	**glRenderMode(**GL_SELECT**)**	Switch to selection or picking mode.
endpick **endselect**	**glRenderMode(**GL_RENDER**)**	Switch to rendering mode.
picksize	**gluPickMatrix** **glSelectBuffer**	Set the return array.
initnames	**glInitNames**	
pushname	**glPushName**	
popname	**glPopName**	
loadname	**glLoadName**	

For more information on picking, see **gluPickMatrix**.

Porting Feedback Functions

With IRIS GL, the way that feedback is handled differs depending on the computer running IRIS GL. OpenGL standardizes feedback functions so you can rely on consistent feedback among various hardware platforms. The following table lists the IRIS GL feedback functions and their equivalent OpenGL functions.

IRIS GL Function	OpenGL Function	Meaning
feedback	**glRenderMode(**GL_FEEDBACK**)**	Switch to feedback mode.
endfeedback	**glRenderMode(**GL_RENDER**)**	Switch to rendering mode.
	glFeedbackBuffer	
passthrough	**glPassThrough**	Place a token marker in the feedback buffer.

OpenGL Functions and Their IRIS GL Equivalents

This appendix lists IRIS GL functions and their equivalent OpenGL functions. The first column is an alphabetic list of IRIS GL functions, the second column contains the corresponding functions to use in OpenGL.

Note The following OpenGL functions listed may behave somewhat differently from the IRIS GL commands, and the parameters may be different as well. For more information on the differences between IRIS GL and OpenGL, see IRIS GL and OpenGL Differences.

IRIS GL Function	OpenGL,GLU, or Win32 Function
acbuf	**glAccum**
acsize	**ChoosePixelFormat**
addtopup	Use Win32 for menus.
afunction	**glAlphaFunc**
arc	**gluPartialDisk**
backbuffer	**glDrawBuffer**(GL_BACK)
backface	**glCullFace**(GL_BACK)
bbox2	Not supported.
bgnclosedline	**glBegin**(GL_LINE_LOOP)
bgncurve	**gluBeginCurve**
bgnline	**glBegin**(GL_LINE_STRIP)
bgnpoint	**glBegin**(GL_POINTS)
bgnpolygon	**glBegin**(GL_POLYGON)
bgnqstrip	**glBegin**(GL_QUAD_STRIP)
bgnsurface	**gluBeginSurface**
bgntmesh	**glBegin**(GL_TRIANGLE_STRIP)
bgntrim	**gluBeginTrim**
blankscreen	Use Win32 for windowing.
blanktime	Use Win32 for windowing.
blendfunction	**glBlendFunc**
blink	Use Win32 for color maps.
blkqread	Use Win32 for event handling.
c	**glColor**
callfunc	Not supported.
callobj	**glCallList**
charstr	**glCallLists**
chunksize	Not needed.
circ	**gluDisk**
clear	**glClear**(GL_COLOR_BUFFER_BIT)
clearhitcode	Not supported.
clipplane	**glClipPlane**
clkon	Use Win32 for keyboard management.

clkoff	Use Win32 for keyboard management.
closeobj	**glEndList**
cmode	**ChoosePixelFormat**
cmov	**glRasterPos3**
cmov2	**glRasterPos2**
color	**glIndex**
compactify	Not needed.
concave	**gluBeginPolygon**
cpack	**glColor**
crv	Not supported.
crvn	Not supported.
curorigin	Use Win32 for cursors.
cursoff	Use Win32 for cursors.
curson	Use Win32 for cursors.
curstype	Use Win32 for cursors.
curvebasis	**glMap1**
curveit	**glEvalMesh1**
curveprecision	Not supported.
cyclemap	Use Win32 for color maps.
czclear	**glClear(**GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT**)**
dbtext	Not supported.
defbasis	**glMap1**
defcursor	Use Win32 for cursors.
deflinestyle	**glLineStipple**
defpattern	**glPolygonStipple**
defpup	Use Win32 for menus.
defrasterfont	**wglUseFontBitmaps**
delobj	**glDeleteLists**
deltag	Not supported.
depthcue	**glFog**
dglclose	Not needed. (OpenGL is network transparent.)
dglopen	Not needed. (OpenGL is network transparent.)
dither	**glEnable(**GL_DITHER**)**
dopup	Use Win32 for menus.
doublebuffer	**ChoosePixelFormat**
draw	**glBegin(**GL_LINES**)**
drawmode	**wglMakeCurrent**
editobj	Not supported.
endclosedline	**glEnd**
endcurve	**gluEndCurve**
endfeedback	**glRenderMode(**GL_RENDER**)**
endfullscreen	Not supported.
endline	**glEnd**

endpick	**glRenderMode(**GL_RENDER**)**
endpoint	**glEnd**
endpolygon	**glEnd**
endpupmode	Use Win32 for menus.
endqstrip	**glEnd**
endselect	**glRenderMode(**GL_RENDER**)**
endsurface	**gluEndSurface**
endtmesh	**glEnd**
endtrim	**gluEndTrim**
feedback	**glFeedbackBuffer**
finish	**glFinish**
fogvertex	**glFog**
font	**glListBase**
foreground	Use Win32 for windowing.
freepup	Use Win32 for menus.
frontbuffer	**glDrawBuffer(**GL_FRONT)
frontface	**glCullFace**
fudge	Use Win32 for windowing.
fullscrn	Not supported.
gammaramp	Use Win32 for color maps.
gbegin	Use Win32 for windowing.
gconfig	No equivalent. (Not needed.)
genobj	**glGenLists**
gentag	Not supported.
getbackface	**glGet**
getbuffer	**glGet**
getbutton	Use Win32 for windowing.
getcmmode	**wglGetCurrentContext**
getcolor	**glGet**
getcpos	**glGet**
getcursor	Not supported.
getdcm	**glIsEnabled**
getdepth	**glGet**
getdescender	Use Win32 for fonts.
getdev	Not supported.
getdisplaymode	**glGet** **wglGetCurrentContext**
getdrawmode	**wglGetCurrentContext**
getfont	Use Win32 for fonts.
getgdesc	**glGet** **DescribePixelFormat** **wglGetCurrentContext** **wglGetCurrentDC**
getgpos	Not supported.
getheight	Use Win32 for fonts.
gethitcode	Not supported.

getlsbackup	Not supported.
getlsrepeat	**glGet**
getlstyle	**glGet**
getlwidth	**glGet**
getmap(void)	Not supported.
getmatrix	**glGet(**GL_MODELVIEW_MATRIX**)** **glGet(**GL_PROJECTION_MATRIX**)**
getmcolor	Not supported.
getmmode	**glGet(**GL_MATRIX_MODE**)**
getmonitor	Not supported.
getnurbsproperty	**gluGetNurbsProperty**
getopenobj	Not supported.
getorigin	Use Win32 for windowing.
getpattern	**glGetPolygonStipple**
getplanes	**glGet(**GL_RED_BITS**)** **glGet(**GL_GREEN_BITS**)** **glGet(**GL_BLUE_BITS**)**
getport	Use Win32 for windowing.
getresetls	Not supported.
getscrbox	Not supported.
getscrmask	**glGet(**GL_SCISSOR_BOX**)**
getshade	**glGet(**GL_CURRENT_INDEX**)**
getsize	Use Win32 for windowing.
getsm	**glGet(**GL_SHADE_MODEL**)**
getvaluator	Use Win32 for event handling
getvideo	Not supported.
getviewport	**glGet(**GL_VIEWPORT**)**
getwritemask	**glGet(**GL_INDEX_WRITEMASK**)**
getwscrn	Use Win32 for windowing.
getzbuffer	**glIsEnabled(**GL_DEPTH_TEST**)**
gexit	Use Win32 for windowing.
gflush	**glFlush**
ginit	Use Win32 for windowing.
glcompat	Not supported.
greset	Not supported.
gRGBcolor	**glGet(**GL_CURRENT_RASTER_COLOR**)**
gRGBcursor	Use Win32 for cursors.
gRGBmask	**glGet(**GL_COLOR_WRITEMASK**)**
gselect	**glSelectBuffer**
gsync	Use Win32 for windowing.
gversion	**glGetString(**GL_RENDERER**)**
iconsize	Use Win32.
icontitle	Use Win32.
imakebackground	Use Win32 for event handling.
initnames	**glInitNames**

ismex	Not supported.
isobj	**glIsList**
isqueued	Use Win32 for event handling.
istag	Not supported.
keepaspect	Use Win32 for windowing.
lampoff	Not supported.
lampon	Not supported.
linesmooth	**glEnable**(GL_LINE_SMOOTH)
linewidth	**glLineWidth**
linewidthf	**glLineWidth**
lmbind	**glEnable**(GL_LIGHTING) **glEnable**(GL_LIGHT)
lmcolor	**glColorMaterial**
lmdef	**glMaterial** **glLight** **glLightModel**
loadmatrix	**glLoadMatrix**
loadname	**glLoadName**
logicop	**glLogicOp**
lookat	**gluLookAt**
lrectread	**glReadPixels**
lrectwrite(])	**glDrawPixels**
lRGBrange	Not supported. (See **glFog**.)
lsbackup	Not supported.
lsetdepth	**glDepthRange**
lshaderange	Not supported. (See **glFog**.)
lsrepeat	**glLineStipple**
makeobj	**glNewList**
maketag	Not supported.
mapcolor	Use Win32 for color maps.
mapw	**gluProject**
maxsize	Use Win32 for windowing.
minsize	Use Win32 for windowing.
mmode	**glMatrixMode**
move	Not supported.
mswapbuffers	Use Win32 for windowing.
multimap	Use Win32 for color maps.
multmatrix	**glMultMatrix**
n3f	**glNormal3fv**
newpup	Use Win32 for Menus.
newtag	Not supported.
nmode	**glEnable**(GL_NORMALIZE)
noborder	Use Win32 for windowing.
noise	Use Win32 for event handling.
noport	Use Win32 for windowing.
normal	**glNormal3fv**

nurbscurve	**gluNurbsCurve**
nurbssurface	**gluNurbsSurface**
objdelete	Not supported.
objinsert	Not supported.
objreplace	Not supported.
onemap	Use Win32 for color maps.
ortho	**glOrtho**
ortho2	**gluOrtho2D**
overlay	Use Win32.
pagecolor	Not supported.
passthrough	**glPassThrough**
patch	**glEvalMesh2**
patchbasis	**glMap2**
patchcurves	**glMap2**
patchprecision	Not supported.
pclos	Not supported. (See **glEnd**.)
pdr	Not supported. (See **glVertex**.)
perspective	**gluPerspective**
pick	**gluPickMatrix** **glRenderMode(**GL_SELECT**)**
picksize	**gluPickMatrix**
pixmode	**glPixelTransfer** and 3
pmv	Not supported. (See **glBegin** and **glVertex**.)
pnt	**glBegin(**GL_POINTS**)**
pntsize	**glPointSize**
pntsizef	**glPointSize**
pntsmooth	**glEnable(**GL_POINT_SMOOTH**)**
polarview	Not supported. (See **glRotate** and **glTranslate**.)
polf	Not supported.
poly	Not supported.
polymode	**glPolygonMode**
polysmooth	**glEnable(**GL_POLYGON_SMOOTH**)**
popattributes	**glPopAttrib**
popmatrix	**glPopMatrix**
popname	**glPopName**
popviewport	**glPopAttrib**
prefposition	Use Win32 for windowing.
prefsize	Use Win32 for windowing.
pupmode	Use Win32 for windowing.
pushattributes	**glPushAttrib**
pushmatrix	**glPushMatrix**
pushname	**glPushName**
pushviewport	**glPushAttrib(**GL_VIEWPORT**)**
pwlcurve	**gluPWLCurve**

qcontrol	Use Win32 for event handling.
qdevice	Use Win32 for event handling.
qenter	Use Win32 for event handling.
qgetfd	Use Win32 for event handling.
qread	Use Win32 for event handling.
qreset	Use Win32 for event handling.
qtest	Use Win32 for event handling.
rcrv	Not supported.
rcrvn	Not supported.
rdr	Not supported.
readdisplay	Not supported.
readRGB	Not supported.
readsource	**glReadBuffer**
rect	**glRect** **glPolygonMode**
rectf	**glRect**
rectcopy	**glCopyPixels**
rectread	**glReadPixels**
rectwrite	**glDrawPixels**
rectzoom	**glPixelZoom**
resetls	Not supported.
reshapeviewport	Not supported.
RGBcolor	**glColor**
RGBcursor	Use Win32 for cursors.
RGBmode	Use Win32 for windowing.
RGBrange	Not supported.
RGBwritemask	**glColorMask**
ringbell	Not supported.
rmv	Not supported.
rot	**glRotate**
rotate	**glRotate**
rpatch	Not supported.
rpdr	Not supported.
rpmv	Not supported.
sbox	**glRect**
scale	**glScale**
sclear	**glClear**(GL_STENCIL_BUFFER_BIT)
scrbox	Not supported.
screenspace	Not supported.
scrmask	**glScissor**
scrnattach	Use Win32 for windowing.
scrnselect	Use Win32 for windowing.
scrsubdivide	Not supported.
select	**glRenderMode**
setbell	Not supported.

setcursor	Use Win32 for cursors.
setdblights	Not supported.
setdepth	**glDepthRange**
setlinestyle	**glLineStipple**
setmap	Use Win32 for color maps.
setmonitor	Not supported.
setnurbsproperty	**gluNurbsProperty**
setpattern	**glPolygonStipple**
setpup	Use Win32 for menus.
setvaluator	Use Win32 for devices.
setvideo	Not supported.
shademodel	**glShadeModel**
shaderange	**glFog**
singlebuffer	Use Win32 for windowing.
smoothline	**glEnable**(GL_LINE_SMOOTH)
spclos	Not supported.
splf	Not supported. (See **glBegin**.)
stencil	**glStencilFunc** **glStencilOp**
stensize	**glStencilMask**
stepunit	Use Win32 for windowing.
strwidth	Use Win32 for fonts and strings.
subpixel	Not needed.
swapbuffers	**SwapBuffers**
swapinterval	Use Win32 for windowing.
swaptmesh	Not supported. (See **glBegin**(GL_TRIANGLE_FAN).)
swinopen	Use Win32 for windowing.
swritemask	**glStencilMask**
t2	**glTexCoord2**
tevbind	**glTexEnv**
tevdef	**glTexEnv**
texbind	**glTexImage2D** **glTexParameter** **gluBuild2DMipmaps**
texdef2d	**glTexImage2D** **glTexParameter** **gluBuild2DMipmaps**
texgen	**glTexGen**
textcolor	Not supported.
textinit	Not supported.
textport	Not supported.
tie	Use Win32 for event handling.
tpoff	Not supported.
tpon	Not supported.
translate	**glTranslate**

underlay	**ChoosePixelFormat**
unqdevice	Use Win32 for event handling.
v	**glVertex**
videocmd	Not supported.
viewport	**glViewport**
winattach	Use Win32 for windowing.
winclose	**wglDeleteContext** **CloseWindow**
winconstraints	Use Win32 for windowing.
windepth	Use Win32 for windowing.
window	**glFrustum**
winget	**wglGetCurrentContext**
winmove	Use Win32 for windowing.
winopen	Use Win32 for windowing.
winpop	Use Win32 for windowing.
winposition	Use Win32 for windowing.
winpush	Use Win32 for windowing.
winset	Use Win32 for windowing.
wintitle	Use Win32 for windowing.
wmpack	**glColorMask**
writemask	**glIndexMask**
writepixels	**glDrawPixels**
writeRGB	**glDrawPixels**
xfpt	Not supported.
zbuffer	**glEnable**(GL_DEPTH_TEST)
zclear	**glClear**(GL_DEPTH_BUFFER_BIT)
zdraw	Not supported.
zfunction	**glDepthFunc**
zsource	Not supported.
zwritemask	**glDepthMask**

IRIS GL and OpenGL Differences

This appendix lists the differences between OpenGL and IRIS GL. A term for each difference is given, followed by a description.

accumulation wrapping
: The OpenGL accumulation buffer operation is not defined when component values exceed 1.0 or drop below -1.0.

antialiased lines
: OpenGL stipples antialiased lines. IRIS GL does not.

arc
: OpenGL supports arcs in its utility library.

attribute lists
: The attributes pushed by IRIS GL **pushattributes** differ from any of the attribute sets pushed by OpenGL **glPushAttrib**. Because all OpenGL states can be read back, however, you can implement any desired push/pop semantics using OpenGL.

automatic texture scaling
: The OpenGL texture interface does not support automatic scaling of images to power-of-two dimensions. However, the GLU supports image scaling.

bbox
: OpenGL doesn't support conditional execution of display lists.

callfunc
: OpenGL doesn't support callback from display lists. Note that IRIS GL doesn't support this functionality either, when client and server are on different platforms.

circle
: OpenGL supports circles with the GLU. In OpenGL both circles and arcs (disks and partial disks) can have holes. Also, you can change subdivision of the primitives in OpenGL, and the primitives' surface normals are available for lighting.

clear options
: OpenGL actually clears buffers. It doesn't apply currently specified pixel operations, such as blending and logicop, regardless of their modes. To clear using such features, you must render a window-size polygon.

closed lines
: OpenGL renders all single-width aliased lines such that abutting lines share no pixels. This means that the "last" pixel of an independent line is not drawn.

color/normal flag
: OpenGL lighting is explicitly enabled or disabled. When enabled, it is effective regardless of the order in which colors and normals are specified.

  You cannot enable or disable lighting between OpenGL **glBegin** and **glEnd** commands. To disable lighting between **glBegin** and **glEnd**, specify zero ambient, diffuse, and specular material reflectance, and then set the material emission to the desired color.

concave polygons
: The core OpenGL API doesn't handle concave polygons, but the GLU supports decomposing concave, non-self-intersecting contours into triangles. These triangles can either be drawn immediately or returned.

current computed color
: OpenGL has no equivalent to a current computed color. If you're using OpenGL as a lighting engine, you can use feedback to obtain colors generated by lighting calculations.

current graphics position
: OpenGL doesn't maintain a current graphics position. IRIS GL commands that depend on current graphics position, such as relative lines and polygons, are not included in OpenGL.

curves
: OpenGL does not support IRIS GL curves. Use of NURBS curves is recommended.

defs/binds
: OpenGL doesn't have the concept of material, light, or texture objects; only of material, light, and texture properties. You can use display lists to create their own objects, however.

depthcue
: OpenGL provides no direct support for depth cueing, but its fog support is a more general capability that you can easily use to emulate the IRIS GL **depthcue** function.

display list editing
: OpenGL display lists can't be edited, only created and destroyed. Because you specify display list names, however, you can redefine individual display lists in a hierarchy.

  OpenGL display lists are designed for data caching, not for database management. They are guaranteed to be stored on the server in client/server environments, so they are not limited by network bandwidth during execution.

  OpenGL display lists can be called between **glBegin** and **glEnd** commands, so the display list hierarchy can be made fine enough that it can, in effect, be edited.

error checking
: OpenGL checks for errors more carefully than IRIS GL. For example, all OpenGL functions that are not accepted between **glBegin** and **glEnd** are detected as errors, and have no other effect.

error return values
: When an OpenGL command that returns a value detects an error, it always returns zero. OpenGL commands that return data through passed pointers make no change to the array contents if an error is detected.

error side effects
: When an OpenGL command results in an error, its only side effect is to update the error flag to the appropriate value. No other state changes are made. (An exception is the OUT_OF_MEMORY error, which is fatal.)

feedback
: Feedback is standardized in OpenGL so it doesn't change from machine to machine.

fonts and strings
: OpenGL requires character glyphs to be manipulated as individual display lists. It provides a display list calling function that accepts a list of display list names, each name represented as 1, 2, or 4 bytes. The **glCallLists** function adds a separately specified offset to each display list name before the call, allowing lists of display list names to be treated as strings.

  This mechanism provides all the functionality of IRIS GL fonts, and considerably more. For example, characters comprised of triangles can be easily manipulated.

frontbuffer
: IRIS GL has complex rules for rendering to the front buffer in single buffer mode. OpenGL handles rendering to the front buffer in a straightforward way.

hollow polygons
: You can use the OpenGL stencil capacity to render hollow polygons. OpenGL doesn't support other means for creating hollow polygons.

index clamping
: Where possible, OpenGL treats color and stencil indexes as bit fields rather than numbers. Thus indexes are masked, rather than clamped, to the supported range of the framebuffer.

integer colors
: Signed integer color components (red, green, blue, or alpha) are mapped linearly to floating points such that the most negative integer maps to -1.0 and the most positive integer maps to 1.0. This mapping occurs when you specify the color, before OpenGL replaces the current color.

  Unsigned integer color components are mapped linearly to floating points such that 0 maps to 0.0 and the largest integer maps to 1.0. This mapping occurs when you specify the color, before OpenGL replaces the current color.

integer normals
: Integer normal components are mapped just like signed color components. The most negative integer maps to -1.0, and the most positive integer maps to 1.0. pixel fragments.

  Pixels drawn by **glDrawPixels** or **glCopyPixels** are always rasterized and converted to fragments. The resulting fragments are textured, fogged, depth buffered, blended, and so on, just as if they were generated from geometric points. Fragment data that isn't provided by the source pixels is augmented from the current raster position. For example, RGBA pixels take the raster position Z and texture coordinates. Depth pixels take the raster position color and texture coordinates.

invariance
: OpenGL guarantees certain consistency that IRIS GL doesn't. For example, OpenGL guarantees that identical code sequences sent to the same system, differing only in the specified blending function, will generate the same pixel fragments. (The fragments differ, however, if blending is enabled and then disabled.)

lighting equation
: The OpenGL lighting equation differs slightly from the IRIS GL equation. OpenGL supports separate attenuation for each light source, rather than a single attenuation for all the light sources like IRIS GL. OpenGL adjusts the equation so that ambient, diffuse, and specular lighting contributions are all attenuated. Also, OpenGL allows you to specify separate colors for the ambient, diffuse, and specular intensities of light sources, as well as for the ambient, diffuse, and specular reflectance of materials. All OpenGL light and material colors include alpha.

  Setting the specular exponent to zero does not defeat specular lighting in OpenGL.

mapw
: OpenGL utilities support mapping between object and window coordinates.

matrix mode
: Where the IRIS GL **ortho**, **ortho2**, **perspective**, and **window** functions operate on a particular matrix, all OpenGL matrix operations work on the current matrix. All OpenGL matrix operations except **glLoadIdentity** and **glLoadMatrix** multiply the current matrix rather than replacing it (as do **ortho**, **ortho2**, **perspective**, and **window** in the IRIS GL).

mipmaps, automatic generation
: The OpenGL texture interface does not support automatic generation of mipmap images. However, the GLU supports the automatic generation of mipmap images for both 1-D and 2-D textures.

move/draw/pmove/pdraw/pclos
: OpenGL supports only Begin/End style graphics, because it does not maintain a current graphics position. The scalar parameter specification of the old move/draw commands is accepted by OpenGL for all vertex related commands, however.

mprojection mode
: IRIS GL doesn't transform geometry by the modelview matrix while in projection matrix mode. OpenGL always transforms by both the modelview and the projection matrix, regardless of matrix mode.

multi-buffer drawing
: OpenGL renders to each color buffer individually, rather than computing a single new color value based on the contents of one color buffer and writing it to all the enabled color buffers, as IRIS GL does.

NURBS
: OpenGL supports NURBS with a combination of core capability (evaluators) and GLU support. All IRIS GL NURBS capabilities are supported.

old polygon mode
: Aliased OpenGL polygons are always point-sampled. IRIS GL's polygon compatibility mode, where pixels outside the polygon perimeter are included in its rasterization, is not supported. If your code uses this polygon mode, it's probably for rectangles. Old polygon mode rectangles appear one pixel wider and higher.

packed color formats
: OpenGL accepts colors as 8-bit components, but these components are treated as an array of bytes

rather than as bytes packed into larger words. By encouraging array indexing rather than shifting, OpenGL promotes endian-invariant programming.

Just as IRIS GL accepts packed colors both for geometric and pixel rendering, OpenGL accepts arrays of color components for geometric and pixel rendering.

patches
: OpenGL doesn't support IRIS GL patches.

per-bit color writemask
: OpenGL writemasks for color components enable or disable changes to the entire component (red, green, blue, or alpha), not to individual bits of components. Note that per-bit writemasks are supported for both color indexes and stencil indexes, however.

per-bit depth writemask
: OpenGL writemasks for depth components enable or disable changes to the entire component, not to individual bits of the depth component.

pick
: The OpenGL Utility library includes support for generating a pick matrix.

pixel coordinates
: In both OpenGL and IRIS GL, the origin of a window's coordinate system is at its lower left corner. OpenGL places the origin at the lower left corner of this pixel, however, while IRIS GL places it at the center of the lower left pixel.

pixel zoom
: OpenGL negative zoom factors reflect about the current graphics position. IRIS GL doesn't define the operation of negative zoom factors, and instead provides RIGHT_TO_LEFT and TOP_TO_BOTTOM reflection pixmodes. These reflection modes reflect in place, rather than about the current raster position. OpenGL doesn't define reflection modes.

pixmode
: OpenGL pixel transfers operate on individual color components, rather than on packed groups of four 8-bit components as does IRIS GL. While OpenGL provides substantially more pixel capability than IRIS GL, it doesn't support packed color constructs, and it doesn't enable color components to be reassigned (red to green, red to blue, and so on) during pixel copy operations.

polf/poly
: OpenGL provides no direct support for vertex lists other than display lists. Functions like **polf** and **poly** can be implemented easily using the OpenGL API, however.

polygon provoking vertex
: Flat shaded IRIS GL polygons take the color of the last vertex specified, while OpenGL polygons take the color of the first vertex specified.

polygon stipple
: With IRIS GL the polygon stipple pattern is relative to the screen. With OpenGL it is relative to a window.

polygon vertex count
: There is no limit to the number of vertices between **glBegin** and **glEnd** with OpenGL, even for **glBegin(**POLYGON**)**. With IRIS GL, polygons are limited to no more than 255 vertices.

readdisplay
: Reading pixels outside window boundaries is properly a window system capability, rather than a rendering capability. Use Win32 functions to replace the IRIS GL **readdisplay** command.

relative move/draw/pmove/pdraw/pclos
: OpenGL doesn't maintain a current graphics position, and therefore doesn't support relative vertex operations.

RGBA logicop()
: OpenGL does not support logical operations on RGBA buffers.

sbox()
: **sbox** is an IRIS GL rectangle primitive that is well-defined only if transformed without rotation. It is designed to be rendered faster than standard rectangles. While OpenGL doesn't support such a

primitive, it can be tuned to render rectangles very quickly when the matrices and other modes are in states that simplify calculations.

scalar arguments
: All OpenGL commands that are accepted between **glBegin** and **glEnd** have entry points that accept scalar arguments. For example, **glColor4f(**red, green, blue, alpha**)**.

scissor
: The OpenGL **glScissor** function doesn't track the viewport. The IRIS GL **viewport** command automatically updates the **scrmask**.

scrbox()
: OpenGL doesn't support bounding box computation.

scrsubdivide()
: OpenGL doesn't support screen subdivision.

single matrix mode
: OpenGL always maintains two matrices: ModelView and Projection. While an OpenGL implementation can consolidate these into a single matrix for performance reasons, it must always present the two-matrix model to the programmer.

subpixel mode
: All OpenGL rendering is subpixel positioned—subpixel mode is always on.

swaptmesh()
: OpenGL doesn't support the **swaptmesh** capability. It does offer two types of triangle meshes, however: one that corresponds to the default "strip" behavior of the IRIS GL, and another that corresponds to calling **swaptmesh** prior to the third and all subsequent vertices when using IRIS GL.

vector arguments
: All OpenGL commands that are accepted between **glBegin** and **glEnd** have entry points that accept vector arguments. For example, **glColor4fv**.

window management
: OpenGL includes no window system commands. It is always supported as an extension to a window or operating system that includes capability for device and window control. Each extension provides a system-specific mechanism for creating, destroying, and manipulating OpenGL rendering contexts. For example, the OpenGL extension to the X window system (GLX) includes roughly 10 commands for this purpose.

  IRIS GL commands such as **gconfig** and **drawmode** are not implemented by OpenGL.

window offset
: IRIS GL returns viewport and character positions in screen, rather than window, coordinates. OpenGL always uses window coordinates.

z rendering
: OpenGL doesn't support rendering colors to the depth buffer. It does allow for additional color buffers, which can be implemented using the same memory that is used for depth buffers in other window configurations. But these additional color buffers cannot share memory with the depth buffer in any single configuration.

A

aliasing

A rendering technique that assigns to pixels the color of the primitive being rendered, regardless of whether that primitive covers all of the pixel's area or only a portion of the pixel's area. This results in jagged edges, or **jaggies**.

alpha

A fourth color component typically used to control color blending. The alpha component is never displayed directly. By convention, OpenGL alpha corresponds to opacity rather than transparency, meaning an alpha value of 1.0 implies complete opacity, and an alpha value of 0.0 implies complete transparency.

animation

The generation of repeated renderings of a scene quickly enough, with smoothly changing viewpoint or object positions, so that the illusion of motion is achieved. OpenGL animation almost always uses double-buffering.

antialiasing

A rendering technique that assigns colors to pixels based on the fraction of the pixel area that is covered by the primitive being rendered. Antialiased rendering reduces or eliminates the jaggies that result from aliased rendering. See also **jaggies**, **rendering**.

application-specific clipping

Clipping of primitives against planes in eye coordinates. The planes are specified by the application using **glClipPlane**. See also **eye coordinates**.

B

back face
See **face**.

bit
Binary digit. A state variable that has only two possible values: 0 or 1. Binary numbers are constructions of one or more bits.

bitmap
A rectangular array of bits. Also, the primitive rendered by the **glBitmap** command, which uses its *bitmap* parameter as a mask.

bitplane
A rectangular array of bits mapped one-to-one with pixels.

blending
Reducing two color components to one component, usually as a linear interpolation between the two components.

buffer
A group of bitplanes that store a single component (such as depth or green) or a single index (such as the color index or the stencil index). Sometimes the red, green, blue, and alpha buffers together are referred to as the color buffer, rather than the color buffers.

C

client

The computer from which OpenGL commands are issued. The computer that issues OpenGL commands can be connected through a network to a different computer that executes the commands, or commands can be issued and executed on the same computer. See also **server**.

client memory

The main memory (where program variables are stored) of the client computer.

clip coordinates

The coordinate system that follows transformation by the projection matrix and that precedes perspective division. View-volume clipping is done in clip coordinates, but application-specific clipping is not. See also **application-specific clipping**.

clipping

Eliminating the portion of a geometric primitive that is outside the half-space defined by a clipping plane. Points are simply rejected if outside. The portion of a line or of a polygon that is outside the half-space is eliminated, and additional vertices are generated as necessary to complete the primitive within the clipping half-space. Geometric primitives and the current raster position (when specified) are always clipped against the six half-spaces defined by the left, right, bottom, top, near, and far planes of the view volume. Applications can specify optional application-specific clipping planes to be applied in eye coordinates.

color index

A single value that represents a color by name, rather than by value. OpenGL color indexes are treated as continuous values (for example, floating-point numbers) while operations such as interpolation and dithering are performed on them. Color indexes stored in the frame buffer are always integer values, however. Floating-point indexes are converted to integers by rounding to the nearest integer value.

color-index mode

Mode of an OpenGL context in which its color buffers store color indexes, rather than red, green, blue, and alpha color components.

color map

A table of index-to-RGB mappings that is accessed by the display hardware. Each color index is read from the color buffer, converted to an RGB triple by lookup in the color map, and sent to the monitor.

component

A single, continuous (for example, floating-point) value that represents an intensity or quantity. Usually, a component value of zero represents the minimum value or intensity, and a component value of one represents the maximum value or intensity, though other normalizations are sometimes used. Because component values are interpreted in a normalized range, they are specified independently of actual resolution. For example, the RGB triple (1, 1, 1) is white, regardless of whether the color buffers store 4, 8, or 12 bits each.

Out-of-range components are typically clamped to the normalized range, not truncated or otherwise interpreted. For example, the RGB triple (1.4, 1.5, 0.9) is clamped to (1.0, 1.0, 0.9) before it's used to update the color buffer. Red, green, blue, alpha, and depth are always treated as components, never as indexes.

context

A complete set of OpenGL state variables. Note that frame buffer contents are not part of the OpenGL state, but that the configuration of the frame buffer is.

convex

Condition of a polygon in which no straight line in the plane of the polygon intersects the polygon more than twice.

convex hull

The smallest convex region enclosing a specified group of points. In two dimensions, the convex hull is found conceptually by stretching a rubber band around the points so that all of the points lie within the band.

coordinate system

In *n*-dimensional space, a set of *n* linearly independent vectors anchored to a point (called the origin). A group of coordinates specifies a point in space (or a vector from the origin) by indicating how far to travel along each vector to reach the point (or tip of the vector).

culling

The process of eliminating a front face or back face of a polygon so that the face isn't drawn.

current matrix

A matrix that transforms coordinates in one coordinate system to coordinates of another system. There are three current matrices in OpenGL: the modelview matrix, which transforms object coordinates (coordinates specified by the programmer) to eye coordinates; the perspective matrix, which transforms eye coordinates to clip coordinates; and the texture matrix, which transforms specified or generated texture coordinates as described by the matrix. Each current matrix is the top element on a stack of matrices. Each of the three stacks can be manipulated with OpenGL matrix-manipulation commands.

current raster position

A window coordinate position that specifies the placement of an image primitive when it's rasterized. The current raster position, and other current raster parameters, are updated when **glRasterpos** is called.

D

depth

Generally refers to the *z* window coordinate.

depth-cueing

A rendering technique that assigns color based on distance from the viewpoint.

display list

A named list of OpenGL commands. Display lists are always stored on the server, so display lists can be used to reduce the network traffic in client/server environments. The contents of a display list may be preprocessed, and might therefore execute more efficiently than the same set of OpenGL commands executed in immediate mode. Such preprocessing is especially important for computing intensive commands such as **glTexImage**.

dithering

A technique for increasing the perceived range of colors in an image at the cost of spatial resolution. Adjacent pixels are assigned differing color values. When viewed from a distance, these colors seem to blend into a single intermediate color. The technique is similar to the half-toning used in black-and-white publications to achieve shades of gray.

double-buffering

Using OpenGL contexts in which both front and back color buffers are double-buffered. Smooth animation is accomplished by rendering into only the back buffer (which isn't displayed), then causing the front and back buffers to be swapped.

E

element
A single component or index.

evaluation
The OpenGL process of generating object-coordinate vertices and parameters from previously specified Bézier equations.

execute
To call an OpenGL command in immediate mode or to call the display list that the command is a part of.

eye coordinates
The coordinate system that follows transformation by the modelview matrix and that precedes transformation by the projection matrix. Lighting and application-specific clipping are done in eye coordinates.

F

face

One side of a polygon. Each polygon has two faces: a front face and a back face. Only one face is ever visible in the window. Whether the back or front face is visible is effectively determined after the polygon is projected onto the window. After this projection, if the polygon's edges are directed clockwise, one of the faces is visible; if directed counterclockwise, the other face is visible. Whether clockwise corresponds to front or back (and counterclockwise corresponds to back or front) is determined by the OpenGL programmer.

flat shading

Refers to coloring a primitive with a single, constant color across its extent, rather than smoothly interpolating colors across the primitive. See **Gouraud shading**.

fog

A rendering technique that can be used to simulate atmospheric effects such as haze, fog, and smog by fading object colors to a background color based on distance from the viewer. Fog also aids in the perception of distance from the viewer, giving a depth cue. See also **depth-cueing**.

font

A group of graphical character representations usually used to display strings of text. The characters may be roman letters, mathematical symbols, Asian ideograms, Egyptian hieroglyphs, and so on.

fragment

Graphic data generated by the rasterization of primitives. Each fragment corresponds to a single pixel and includes color, depth, and sometimes texture-coordinate values.

frame buffer

A stack of bitplanes. All the buffers of a given window or context. Sometimes includes all the pixel memory of the graphics hardware accelerator. See also **bitplane**.

front face

See **face**.

frustum

The view volume warped by perspective division.

G

gamma correction
A function applied to colors stored in the frame buffer to correct for the nonlinear response of the eye (and sometimes of the monitor) to linear changes in color-intensity values.

geometric model
The object-coordinate vertices and parameters that describe an object. Note that OpenGL doesn't define a syntax for geometric models, but rather a syntax and semantics for the rendering of geometric models.

geometric object
A geometric model.

geometric primitive
A point, line, or polygon.

Gouraud shading
Smooth interpolation of colors across a polygon or line segment. Colors are assigned at vertices and linearly interpolated across the primitive to produce a relatively smooth variation in color. Also called *smooth shading*.

group
A group of one, two, three, or four elements that represents each pixel of an image in client memory. Thus, in the context of a client memory image, a group and a pixel are the same thing.

H

half-space

The result of a plane dividing space. A plane divides space into two half-spaces.

homogenous coordinates

A set of *n*+1 coordinates used to represent points in *n*-dimensional projective space. Points in projective space can be thought of as points in Euclidean space together with some points at infinity. The coordinates are homogenous because a scaling of each of the coordinates by the same non-zero constant doesn't alter the point to which the coordinates refer. Homogeneous coordinates are useful in the calculations of projective geometry, and thus in computer graphics, where scenes must be projected onto a window.

I

image
A rectangular array of pixels, either in client memory or in the frame buffer.

image primitive
A bitmap or an image.

immediate mode
Mode in which an OpenGL command is called directly, rather than from a display list. No immediate-mode bit exists; the *mode* in immediate mode refers to usage of OpenGL, rather than to a specific bit of OpenGL state.

index
A single value that is interpreted as an absolute value, rather than as a normalized value in a specified range (as is a component). Color indexes are the names of colors, which are dereferenced by the display hardware using the color map. Indexes are typically masked, rather than clamped, when out of range. For example, the index 0xf7 is masked to 0x7 when written to a 4-bit buffer (color or stencil). Color indexes and stencil indexes are always treated as indexes, never as components.

IRIS GL
Silicon Graphics' proprietary graphics library, developed from 1982 through 1992. OpenGL was designed with IRIS GL as a starting point.

JK

jaggies

Artifacts of aliased rendering. The edges of primitives that are rendered with aliasing are jagged rather than smooth. A near-horizontal aliased line, for example, is rendered as a set of horizontal lines on adjacent pixel rows, rather than as a smooth, continuous line.

L

lighting

The process of computing the color of a vertex based on current lights, material properties, and lighting-model modes.

line

A straight region of finite width between two vertices. (Unlike mathematical lines, OpenGL lines have finite width and length.) Each segment of a strip of lines is itself a line.

luminance

The perceived brightness of a surface. Often refers to a weighted average of red, green, and blue color values that gives the perceived brightness of the combination.

M

matrices

Plural of *matrix*. See **matrix**.

matrix

A two-dimensional array of values. OpenGL matrices are all 4x4, though when they are stored in client memory they're treated as 1x16 single-dimension arrays.

modelview matrix

The 4x4 matrix that transforms points, lines, polygons, and raster positions from object coordinates to eye coordinates.

monitor

The device that displays the image in the frame buffer.

motion blurring

A technique that simulates what you get on a piece of film when you take a picture of a moving object, or when you move the camera when you take a picture of a stationary object. In animations without motion blur, object motion can appear jerky.

N

network
A connection between two or more computers that allows each to transfer data to and from the others.

nonconvex
A state of a polygon in which a line in the plane of the polygon intersects the polygon more than twice.

normal
A three-component plane equation that defines the angular orientation, but not position, of a plane or surface.

normalize
To divide each of the components of a normal by the square root of the sum of their squares. Then, if the normal is thought of as a vector from the origin to the point (nx′, ny′, nx′), this vector has unit length:

$nx' = nx/factor$
$ny' = ny/factor$
$nz' = nz/factor$

normal vector
See **normal**.

NURBS
Non-Uniform Rational B-Spline. A common way to specify parametric curves and surfaces.

O

object
An object-coordinate model that is rendered as a collection of primitives.

object coordinates
Coordinate system prior to any OpenGL transformation.

orthographic
Nonperspective projection, as in some engineering drawings, with no foreshortening.

P

parameter
A value passed as an argument to an OpenGL command. Sometimes one of the values passed by reference to an OpenGL command.

perspective division
The division of *x*, *y*, and *z* by *w*, carried out in clip coordinates. See also **clip coordinates**.

pixel
Picture element. The bits at location (*x*, *y*) of all the bitplanes in the frame buffer constitute the single pixel (*x*, *y*). In an image in client memory, a pixel is one group of elements. In OpenGL window coordinates, each pixel corresponds to a 1.0x1.0 screen area. The coordinates of the lower left corner of the pixel names *x*, *y* are (*x*, *y*), and the upper right corner are (*x*+1, *y*+1).

point
An exact location in space, which is rendered as a finite-diameter dot.

polygon
A near-planar surface bounded by edges specified by vertices. Each triangle of a triangle mesh is a polygon, as is each quadrilateral of a quadrilateral mesh. The rectangle specified by **glRect** is also a polygon.

primitive
A shape (such as a point, line, polygon, bitmap or image) that can be drawn, stored, and manipulated as a discrete entity; elements from which large graphic designs are created.

projection matrix
The 4x4 matrix that transforms points, lines, polygons, and raster positions from eye coordinates to clip coordinates.

Q

quadrilateral
A polygon with four edges.

R

rasterize

To convert a projected point, line, or polygon, or the pixels of a bitmap or image, to fragments, each corresponding to a pixel in the frame buffer. Note that all primitives are rasterized, not just points, lines, and polygons.

rectangle

A quadrilateral whose alternate edges are parallel to each other in object coordinates. Polygons specified with **glRect*( )** are always rectangles; other quadrilaterals might be rectangles.

rendering

Conversion of primitives specified in object coordinates to an image in the frame buffer. Rendering is the primary operation of OpenGL.

RGBA

The red, green, blue, and alpha color components of the RGBA mode.

RGBA mode

An OpenGL context in which its color buffers store red, green, blue, and alpha color components, rather than color indexes.

S

server

The computer on which OpenGL commands are executed. This might differ from the computer from which commands are issued. See also **client**.

shading

The process of interpolating color within the interior of a polygon, or between the vertices of a line, during rasterization.

single-buffering

An OpenGL context without a back color buffer.

stipple

A one- or two-dimensional binary pattern that defeats the generation of fragments where its value is zero. Line stipples are one-dimensional and are applied relative to the start of a line. Polygon stipples are two-dimensional and are applied with a fixed orientation to the window.

T

tessellation
Reduction of a portion of an analytic surface to a mesh of polygons, or of a portion of an analytic curve to a sequence of lines.

texel
A texture element. A texel is obtained from texture memory and represents the color of the texture to be applied to a corresponding fragment. See also **fragment**.

texture
A one- or two-dimensional image used to modify the color of fragments produced by rasterization. See also **rasterize**.

texture mapping
The process of applying an image (the texture) to a primitive. Texture mapping is often used to add realism to a scene. For example, you could apply a picture of a building facade to a polygon representing a wall. See also **texture**.

texture matrix
The 4x4 matrix that transforms texture coordinates from the coordinates that they're specified in to the coordinates that are used for interpolation and texture lookup.

transformation
A warping of space. In OpenGL, transformations are limited to projective transformations that include anything that can be represented by a 4x4 matrix. Such transformations include rotations, translations, (nonuniform) scalings along the coordinate axes, perspective transformations, and combinations of these.

triangle
A polygon with three edges. Triangles are always convex.

U V

vertex

A point in three-dimensional space.

vertices

Preferred plural of vertex. See **vertex**.

viewpoint

The origin of either the eye- or the clip-coordinate system, depending on context. For example, when discussing lighting, the viewpoint is the origin of the eye-coordinate system. When discussing projection, the viewpoint is the origin of the clip-coordinate system. With a typical projection matrix, the eye-coordinate and clip-coordinate origins are at the same location.

view volume

The volume in clip coordinates whose coordinates satisfy the three conditions

- $-w \leq x \leq w$
- $-w \leq y \leq w$
- $-w \leq z \leq w$

W

window

A subregion of the frame buffer, usually rectangular, whose pixels all have the same buffer configuration. An OpenGL context renders to one window at a time.

window-aligned

When referring to line segments or polygon edges, implies that these are parallel to the window boundaries. (In OpenGL, the window is rectangular, with horizontal and vertical edges). When referring to a polygon pattern, implies that the pattern is fixed relative to the window origin.

window coordinates

The coordinate system of a window. It's important to distinguish between the names of pixels, which are discrete, and the window-coordinate system, which is continuous. For example, the pixel at the lower-left corner of a window is pixel (0, 0); the window coordinates of the center of this pixel are (0.5, 0.5, *z*). Note that window coordinates include a depth, or *z*, component, and that this component is continuous as well.

wireframe

A representation of an object that contains line segments only. Typically, the line segments indicate polygon edges.

X Y Z

X Window System

A window system used by many of the machines on which OpenGL is implemented.

About OpenGL

OpenGL, originally developed by Silicon Graphics Incorporated (SGI) for their graphics workstations, lets applications create high-quality color images independent of windowing systems, operating systems, and hardware.

The OpenGL Architecture Review Board (ARB), an industry consortium, is currently responsible for defining OpenGL. Members of the ARB include Silicon Graphics Incorporated, Microsoft Corporation, Intel, IBM, and Digital Equipment Corporation.

The official reference document for OpenGL, version 1, is the *OpenGL Reference Manual*, by the OpenGL Architecture Review Board (ISBN 0-201-63276-4). The official guide to learning OpenGL, version 1, is the *OpenGL Programming Guide*, by Jackie Neider, Tom Davis, and Mason Woo (ISBN 0-201-63274-8). Both books are published by Addison-Wesley.