OpenGL 1.2 Reference Manual

SR28-5125-01

OpenGL 1.2 Reference Manual

SR28-5125-01

Note

Before using this information and the product it supports, read the information in “Notices,” on page 553.

Second Edition (October 2000)

This edition applies to OpenGL Version 1.2 for AIX and to all subsequent releases of this product until otherwise indicated in new editions.

© Copyright International Business Machines Corporation 1994, 2002.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

© Copyright IBM Corp. 1994, 2002

About This Book

OpenGL Programmer's Reference provides reference information on the OpenGL application programming interface (API).

This publication documents the functional interface of:

- OpenGL 1.2 (first introduced in AIX 4.3.2)
- GLX 1.3 (first introduced in AIX 4.3.2)
- GLU 1.3 (first introduced in AIX 4.3.3)

It also documents several OpenGL extensions supported on this operating system.

Applications/users should query OpenGL to determine if the extension is supported (glXQueryExtensionsString, glGetString, and gluGetString) prior to making extension specific OpenGL, GLX, or GLU calls.

Further information is also avaiable in **/usr/lpp/OpenGL/README** on your installed operating system.

Who Should Use This Book

This book is intended for programmers with C programming knowledge who want to develop 3D applications.

Highlighting

The following highlighting conventions are used in this book:

Bold	Identifies commands, subroutines, keywords, files, structures, directories, and other items whose names are predefined by the system. Also identifies graphical objects such as buttons, labels, and icons that the user selects.
Italics	Identifies parameters whose actual names or values are to be supplied by the user.
`Monospace`	Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code similar to what you might write as a programmer, messages from the system, or information you should actually type.

ISO 9000

ISO 9000 registered quality systems were used in the development and manufacturing of this product.

Related Publications

The following books contain information about or related to *OpenGL Programmer's Reference*:

- *OpenGL 2.1 Reference Manual*
- *AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs*

© Copyright IBM Corp. 1994, 2002

Chapter 1. OpenGL Subroutines

Following is a list of the basic OpenGL subroutines and the purpose of each.

A	
glAccum	Operates on the accumulation buffer.
glActiveTextureARB	Specifies which texture unit is active.
glAlphaFunc	Specifies the alpha test function.
glAreTexturesResident	Determines if textures are loaded in texture memory.
glAreTexturesResidentEXT	Determines if textures are loaded in texture memory.
glArrayElement	Renders a vertex using the specified vertex array element.
glArrayElementEXT	Renders a vertex using the specified vertex array element.
B	
glBegin or **glEnd**	Delimits the vertices of a primitive or group of like primitives.
glBindTexture	Binds a named texture to a texturing target.
glBindTextureEXT	Binds a named texture to a texturing target.
glBitmap	Draws a bitmap.
glBlendColor	Sets the blend color.
glBlendColorEXT	Sets the blend color.
glBlendEquation	Specifies the RGB color blend equation.
glBlendEquationEXT	Specifies the RGB color blend equation.
glBlendFunc	Specifies pixel arithmetic.
glBlendFuncSeparateEXT	Specifies separate RGB and Alpha blend factors.
C	
glCallList	Executes a display list.
glCallLists	Executes a list of display lists.
glClear	Clears buffers within the viewport.
glClearAccum	Specifies clear values for the accumulation buffer.
glClearColor	Specifies clear values for the color buffers.
glClearDepth	Specifies the clear value for the depth buffer.
glClearIndex	Specifies the clear value for the color index buffers.
glClearStencil	Specifies the clear value for the stencil buffer.
glClientActiveTextureARB	Specifies which texture unit is active.
glClipBoundingBoxIBM or glClipBoundingSphereIBM or glClipBoundingVerticesIBM	Determines whether the specified object is trivially accepted, trivially rejected, or clipped by the current set of clipping planes.
glClipPlane	Specifies a plane against which all geometry is clipped.
glColor	Sets the current color.
glColorMask	Enables and disables the writing of frame buffer color components.
glColorMaterial	Causes a material color to track the current color.
glColorNormalVertexSUN	Specifies a color, a normal and a vertex in one call.
glColorPointer	Defines an array of colors.
glColorPointerEXT	Defines an array of colors.
glColorPointerListIBM	Defines a list of color arrays.
glColorSubTable	Defines a contiguous subset of a color lookup table.
glColorTable	Defines a color lookup table.
glColorTableParameter	Specifies attributes to be used when loading a color table.
glColorVertexSUN	Specifies a color and a vertex in one call.

© Copyright IBM Corp. 1994, 2002

glCopyColorSubTable	Loads a subset of a color lookup table from the current GL_READ_BUFFER.
glCopyColorTable	Load a color lookup table from the current GL_READ_BUFFER.
glCopyPixels	Copies pixels in the frame buffer.
glCopyTexImage1D	Defines a one-dimensional (1D) texture image.
glCopyTexImage2D	Defines a two-dimensional (2D) texture image.
glCopyTexSubImage1D	Copies a one-dimensional (1D) texture subimage.
glCopyTexSubImage2D	Copies a two-dimensional (2D) texture subimage.
glCopyTexSubImage3D	Copies a three-dimensional (3D) texture subimage.
glCopyTexSubImage3DEXT	Copies a three-dimensional (3D) texture subimage.
glCullFace	Specifies whether frontfacing or backfacing facets may be culled.

D

glDeleteLists	Deletes a contiguous group of display lists.
glDeleteTextures	Deletes named textures.
glDeleteTexturesEXT	Deletes named textures.
glDepthFunc	Specifies the function used for depth buffer comparisons.
glDepthMask	Enables or disables writing into the depth buffer.
glDepthRange	Specifies the mapping of *z* values from normalized device coordinates to window coordinates.
glDisable	Tests whether a capability is enabled.
glDisableClientState	Disables an array.
glDrawArrays	Renders primitives from array data.
glDrawArraysEXT	Renders primitives from array data.
glDrawBuffer	Specifies which color buffers are to be used for drawing.
glDrawElements	Renders primitives from array data.
glDrawPixels	Writes a block of pixels to the frame buffer.
glDrawRangeElements	Renders primitives from array data.

E

glEdgeFlag	Marks edges as either boundary or nonboundary.
glEdgeFlagPointer	Defines an array of edge flags.
glEdgeFlagPointerEXT	Defines an array of edge flags.
glEdgeFlagPointerListIBM	Defines a list of edge flag arrays.
glEnable or **glDisable**	Tests whether a capability is enabled.
glEnableClientState or**glDisableClientState**	Enables or disables an array.
glEnd	Delimits the vertices of a primitive or group of like primitives.
glEvalCoord	Evaluates enabled one-dimensional (1D) and two-dimensional (2D) maps.
glEvalMesh	Computes a one-dimensional (1D) or two-dimensional (2D) grid of points or lines.
glEvalPoint	Generates and evaluates a single point in a mesh.

F

glFeedbackBuffer	Controls the feedback mode.
glFinish	Blocks until all GL execution is complete.
glFlush	Forces the running of GL subroutines in finite time.
glFog	Specifies fog parameters.
glFogCoordEXT	Specifies a Fog Coordinate.
glFogCoordPointerEXT	Specifies an array of fog coordinates.
glFogCoordPointerListIBM	Defines a list of arrays of fog coordinates.
glFrontFace	Defines frontfacing and backfacing polygons.

glFrustum	Multiplies the current matrix by a perspective matrix.
G	
glGenLists	Generates a contiguous set of empty display lists.
glGenTextures	Generate texture names.
glGenTexturesEXT	Generates texture names.
glGet	Returns the value or values of a selected parameter.
glGetClipPlane	Returns the coefficients of the clipping plane.
glGetColorTable	Returns a color lookup table to the user.
glGetColorTableParameter	Returns attributes used when loading a color table.
glGetError	Returns error information.
glGetLight	Returns light source parameter values.
glGetMap	Returns evaluator parameters.
glGetMaterial	Returns material parameters.
glGetPixelMap	Returns the specified pixel map.
glGetPointerv	Returns the address of the specified pointer.
glGetPointervEXT	Returns the address of a vertex data array.
glGetPolygonStipple	Returns the polygonstipple pattern.
glGetString	Returns a string describing the current GL connection.
glGetTexEnv	Returns texture environment parameters.
glGetTexGen	Returns texture coordinate generation parameters.
glGetTexImage	Returns a texture image.
glGetTexLevelParameter	Returns texture parameter levels for a specific level of detail.
glGetTexParameter	Returns texture parameter values.
H	
glHint	Specifies implementation-specific hints.
I	
glIndex	Sets the current color index.
glIndexMask	Controls the writing of individual bits in the color index buffers.
glIndexPointer	Defines an array of color indexes.
glIndexPointerEXT	Defines an array of color indexes.
glIndexPointerListIBM	Defines a list of color index arrays.
glInitNames	Initializes the name stack.
glInterleavedArrays	Simultaneously specifies and enables several interleaved arrays.
glIsEnabled	Tests whether a capability is enabled.
glIsList	Tests for display list existence.
glIsTexture	Determines if a name corresponds to a texture.
glIsTextureEXT	Determines if a name corresponds to a texture.
L	
glLight	Sets light source parameters.
glLightModel	Sets the lighting model parameters.
glLineStipple	Specifies the line stipple pattern.
glLineWidth	Specifies the width of rasterized lines.
glListBase	Sets the display-list base for the **glCallLists** subroutine.
glLoadIdentity	Replaces the current matrix with the identity matrix.
glLoadMatrix	Replaces the current matrix with an arbitrary matrix.
glLoadName	Loads a name onto the name stack.
glLoadNamedMatrixIBM	Loads a pre-defined matrix into the top of the named matrix stack.

glLoadTransposeMatrixARB	Loads a matrix in row-major order, rather than column-major order.
glLockArraysEXT	Locks the currently enabled vertex arrays.
glLogicOp	Specifies a logical pixel operation for color index rendering.
M	
glMap1	Defines a one-dimensional (1D) evaluator.
glMap2	Defines a two-dimensional (2D) evaluator.
glMapGrid	Defines a one-dimensional (1D) or two-dimensional (2D) mesh.
glMaterial	Specifies material parameters for the lighting model.
glMatrixMode	Specifies the current matrix.
glMultiDrawArraysEXT	Renders multiple primitives from array data.
glMultiDrawElementsEXT	Renders multiple primitives from array data.
glMultiModeDrawArraysIBM	Renders primitives of multiple primitive types from array data.
glMultiModeDrawElementsIBM	Renders primitives of multiple primitive types from array data.
glMultiTexCoordARB	Sets the current texture coordinates.
glMultMatrix	Multiplies the current matrix by an arbitrary matrix.
glMultTransposeMatrixARB	Multiplies the current matrix by a matrix specified in row-major order, rather than column-major order.
N	
glNewList	Creates or replaces a display list.
glNormal	Sets the current normal vector.
glNormalPointer	Defines an array of normals.
glNormalPointerEXT	Defines an array of normals.
glNormalPointerListIBM	Defines a list of normal arrays.
glNormalVertexSUN	Specifies a normal and a vertex in one call.
O	
glOrtho	Multiplies the current matrix by an orthographic matrix.
P	
glPassThrough	Places a marker in the feedback buffer.
glPixelMap	Sets up pixel transfer maps.
glPixelStore	Sets pixel storage modes.
glPixelTransfer	Sets pixel transfer modes.
glPixelZoom	Specifies the pixel zoom factors.
glPointSize	Specifies the diameter of rasterized points.
glPolygonMode	Selects a polygon rasterization mode.
glPolygonOffset	Sets the scale and bias used to calculate depth values.
glPolygonOffsetEXT	Sets the scale and bias used to calculate z values.
glPolygonStipple	Sets the polygon stippling pattern.
glPrioritizeTextures	Sets texture residence priority.
glPrioritizeTexturesEXT	Sets texture residence priority.
glPushAttrib or **glPopAttrib**	Pushes and pops the attribute stack.
glPushClientAttrib or **glPopClientAttrib**	Pushes and pops the attribute stack.
glPushMatrix or **glPopMatrix**	Pushes and pops the current matrix stack.
glPushName or **glPopName**	Pushes and pops the name stack.
R	
glRasterPos	Specifies the raster position for pixel operations.
glReadBuffer	Selects a color buffer source for pixels.

glReadPixels	Reads a block of pixels from the frame buffer.
glRect	Draws a rectangle.
glRenderMode	Sets rasterization mode.
glRotate	Multiplies the current matrix by a rotation matrix.
S	
glScale	Multiplies the current matrix by a general scaling matrix.
glScissor	Defines the scissor box.
glSecondaryColorEXT	Specifies an RGB color used by the Color Sum stage.
glSecondaryColorPointerEXT	Specifies an array of secondary colors.
glSecondaryColorPointerListIBM	Defines a list of arrays of secondary colors.
glSelectBuffer	Establishes a buffer for selection mode values.
glShadeModel	Selects flat or smooth shading.
glStencilFunc	Sets function and reference values for stencil testing.
glStencilMask	Controls the writing of individual bits in the stencil planes.
glStencilOp	Sets stencil test actions.
T	
glTexCoord	Sets the current texture coordinates.
glTexCoordColorNormalVertexSUN	Specifies a texture coordinate, a color, a normal and a vertex in one call.
glTexCoordColorVertexSUN	Specifies a texture coordinate, a color, and a vertex in one call.
glTexCoordNormalVertexSUN	Specifies a texture coordinate, a normal and a vertex in one call.
glTexCoordPointer	Defines an array of texture coordinates.
glTexCoordPointerEXT	Defines an array of texture coordinates.
glTexCoordPointerListIBM	Defines a list of texture coordinate arrays.
glTexCoordVertexSUN	Specifies a texture coordinate and a vertex in one call.
glTexEnv	Sets texture environment parameters.
glTexGen	Controls the generation of texture coordinates.
glTexImage1D	Specifies a one-dimensional (1D) texture image.
glTexImage2D	Specifies a two-dimensional (2D) texture image.
glTexImage3D	Specifies a three-dimensional (3D) texture image.
glTexImage3DEXT	Specifies a three-dimensional (3D) texture image.
glTexParameter	Sets texture parameters.
glTexSubImage1D	Specifies a one-dimensional (1D) texture subimage.
glTexSubImage1DEXT	Specifies a one-dimensional (1D) texture subimage.
glTexSubImage2D	Specifies a two-dimensional (2D) texture subimage.
glTexSubImage2DEXT	Specifies a two-dimensional (2D) texture subimage.
glTexSubImage3D	Specifies a three-dimensional (3D) texture subimage.
glTexSubImage3DEXT	Specifies a three-dimensional (3D) texture subimage.
glTranslate	Multiplies the current matrix by a translation matrix.
U	
glUnLockArraysEXT	Unlocks the currently enabled vertex arrays.
V	
glVertex	Specifies a vertex.
glVertexPointer	Defines an array of vertex data.
glVertexPointerEXT	Defines an array of vertex data.
glVertexPointerListIBM	Defines a list of vertex arrays.
glViewport	Sets the viewport.
glVisibilityBufferIBM	Specifies the array in which visibility calculation results are stored.

glVisibilityThresholdIBM	Specifies the number of visible fragments rendered before a visibility hit is registered.

glAccum Subroutine

Purpose

Operates on the accumulation buffer.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glAccum(GLenum  Operation,
             GLfloat  Value)
```

Description

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. Effects such as antialiasing (of points, lines, and polygons), motion-blur, and depth of field can be created by accumulating images generated with different transformation matrices.

Each pixel in the accumulation buffer consists of red, green, blue, and alpha (RGBA) values. The number of bits per component in the accumulation buffer depends on the implementation. You can examine this number by calling **glGetInteger** four times, with arguments **GL_ACCUM_RED_BITS**, **GL_ACCUM_GREEN_BITS**, **GL_ACCUM_BLUE_BITS**, and **GL_ACCUM_ALPHA_BITS**, respectively. (See the **glGet** subroutine for more information on **glGetInteger**.) 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 1-to-1 with frame buffer pixels.

The **glAccum** subroutine operates on the accumulation buffer. The first argument, *Operation*, 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_LOAD**, **GL_ACCUM**, **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 RGBA 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:

GL_ACCUM	Obtains RGBA values from the buffer currently selected for reading. (See **glReadBuffer**.) Each component value is divided by 2n-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	Functions similarly 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 RGBA values from the currently selected buffer are divided by 2n-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 RGBA 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 RGBA component is multiplied by *Value*, then multiplied by 2n-1, clamped to the range [0, 2n-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.

The accumulation buffer is cleared by specifying R, G, B, A values to set it to with the **glClearAccum** directive, and then issuing a **glClear** subroutine with the accumulation buffer enabled.

Parameters

Operation	Specifies the accumulation buffer operation. Symbolic constants **GL_LOAD**, **GL_ACCUM**, **GL_MULT**, **GL_ADD**, and **GL_RETURN** are accepted.
Value	Specifies a floating-point value used in the accumulation buffer operation. The *Operation* parameter determines how *Value* is used.

Notes

All **glAccum** operations update only those pixels within the current scissor box.

Errors

GL_INVALID_ENUM	*Operation* is set to an unaccepted value.
GL_INVALID_OPERATION	There is no accumulation buffer.
GL_INVALID_OPERATION	The **glAccum** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glAccum** subroutine are as follows. (See the **glGet** subroutine for more information.)

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**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glBlendFunc** subroutine, **glClear** subroutine, **glClearAccum** subroutine, **glCopyPixels** subroutine, **glLogicOp** subroutine, **glPixelStore** subroutine, **glPixelTransfer** subroutine, **glReadBuffer** subroutine, **glReadPixels** subroutine, **glScissor** subroutine, **glStencilOp** subroutine.

glActiveTextureARB Subroutine

Purpose

Specify which texture unit is active.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
void glActiveTextureARB(GLenum  texture)
```

Description

glActiveTextureARB selects which texture unit subsequent texture state calls will affect. The number of texture units an implementation supports is implementation dependent, but must be at least two. The texture parameter must be one of **GL_TEXTUREi_ARB**, where 0 <= i < **GL_MAX_TEXTURE_UNITS_ARB**. The initial value is **GL_TEXTURE0_ARB**.

Parameters

texture	specifies which texture unit to make active.

Notes

Vertex arrays are client-side GL resources, which are selected by the **glClientActiveTextureARB** routine.

If the **GL_ARB_multitexture** extension is NOT present, then the number of texture units supported by the implementation is one, not two, as described above.

The following OpenGL subroutines will be routed to different texture units based on this call:

- **glEnable (GL_TEXTURE_GEN_*)**
- **glDisable (GL_TEXTURE_GEN_*)**
- **glTexGen***
- **glTexEnv***
- **glTexImage***
- **glTexSubImage***
- **glCopyTexImage***
- **glCopyTexSubImage***
- **glBindTexture**

Errors

GL_INVALID_OPERATION	is generated if texture is not one of the accepted values.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glEnableClientState** or **glDisableClientState** subroutine, the **glMultiTexCoordARB** subroutine, the **glTexCoordPointer**.

glAlphaFunc Subroutine

Purpose

Specifies the alpha test function.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glAlphaFunc(GLenum  Function,
          GLclampf  Reference)
```

Description

The alpha test discards fragments conditional on the outcome of a comparison between the incoming fragment's alpha value and a constant reference value. The **glAlphaFunc** subroutine specifies the reference and comparison function. The comparison is performed only if alpha testing is enabled. (See **glEnable or glDisable** of **GL_ALPHA_TEST**.)

The *Function* and *Reference* parameters specify the conditions under which the pixel is drawn. The incoming alpha value is compared to the *Reference* parameter using the function specified by *Function*. 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 comparison functions are:

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.

The **glAlphaFunc** subroutine operates on all pixel write operations, including those resulting from the scan conversion of points, lines, polygons, and bitmaps, and those resulting from pixel draw and copy operations. The **glAlphaFunc** subroutine does not affect screen clear operations.

Parameters

Function	Specifies the alpha comparison function. Symbolic constants **GL_NEVER**, **GL_LESS**, **GL_EQUAL**, **GL_LEQUAL**, **GL_GREATER**, **GL_NOTEQUAL**, **GL_GEQUAL**, and **GL_ALWAYS** are accepted. The default function is **GL_ALWAYS**.
Reference	Specifies the reference value to which incoming alpha values are compared. This value is clamped to the range 0 (zero) through 1 (one), where 0 represents the lowest possible alpha value, and 1 the highest possible value. The default reference is 0.

Notes

Alpha testing is done only in RGBA mode.

Errors

GL_INVALID_ENUM	*Function* is set to an unaccepted value.
GL_INVALID_OPERATION	The **glAlphaFunc** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glAlphaFunc** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_ALPHA_TEST_FUNC**

glGet with argument **GL_ALPHA_TEST_REF**

glIsEnabled with argument **GL_ALPHA_TEST**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glBlendFunc** subroutine, **glClear** subroutine, **glDepthfunc** subroutine, **glEnable** or **glDisable** subroutine, **glStencilFunc** subroutine.

glAreTexturesResident Subroutine

Purpose

Determines if textures are loaded in texture memory.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLboolean glAreTexturesResident(GLsizei  n,
      const GLuint * textures,
      GLboolean * residences)
```

Description

On machines with a limited amount of texture memory, OpenGL establishes a “working set” of textures that are resident in texture memory. These textures may be bound to a texture target much more efficiently than textures that are not resident.

The **glAreTexturesResident** subroutine 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 not all the named textures are 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.

The residence status of a single bound texture may also be queried by calling **glGetTexParameter** with the target argument set to the target to which the texture is bound, and the parameter name argument set to **GL_TEXTURE_RESIDENT**. This is the only way that the residence status of a default texture can be queried.

The **glAreTexturesResident** subroutine is not included in display lists.

Parameters

n	Specifies the number of textures to be queried.
textures	Specifies an array containing the names of the textures to be queried.
residences	Specifies 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*.

Notes

The **glAreTexturesResident** subroutine is available only if the GL version is 1.1 or greater.

The **glAreTexturesResident** subroutine returns the residency status of the textures at the time of invocation. It does not guarantee that the textures will remain resident at any other time.

If textures live in virtual memory (there is no texture memory) they are considered always resident.

Errors

GL_INVALID_VALUE is generated if *n* is negative.

GL_INVALID_VALUE is generated if any element in textures is zero or does not name a texture. In that case, the function returns **GL_FALSE** and the contents of residences is indeterminate.

GL_INVALID_OPERATION is generated if **glAreTexturesResident** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glGetTexParameter with parameter name **GL_TEXTURE_RESIDENT** retrieves the residence status of a currently-bound texture.

Related Information

The **glBindTexture** subroutine, **glPrioritizeTextures** subroutine, **glTexImage1D** subroutine, **glTexImage2D** subroutine, **glTexParameter** subroutine.

glAreTexturesResidentEXT Subroutine

Purpose

Renders a vertex using the specified vertex array element.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLboolean glAreTexturesResidentEXT(GLsizei  n,
     const GLuint * textures,
     GLboolean    * residences)
```

Description

On machines with a limited amount of texture memory, OpenGL establishes a “working set” of textures that are resident in texture memory. These textures may be bound to a texture target much more efficiently than textures that are not resident.

The **glAreTexturesResidentEXT** subroutine queries the texture residence status of the *n* textures named by the elements of *textures*. If all the named textures are resident, **glAreTexturesResidentEXT** returns **GL_TRUE** and the contents of *residences* are undisturbed. If not all the named textures are resident, **glAreTexturesResidentEXT** 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.

The residence status of a single bound texture may also be queried by calling **glGetTexParameter** with the target argument set to the target to which the texture is bound, and the parameter name argument set to **GL_TEXTURE_RESIDENT_EXT**. This is the only way that the residence status of a default texture can be queried.

The **glAreTexturesResidentEXT** subroutine is not included in display lists.

Parameters

n	Specifies the number of textures to be queried.
textures	Specifies an array containing the names of the textures to be queried.
residences	Specifies 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*.

Notes

The **glAreTexturesResidentEXT** subroutine is part of the **EXT_texture_object** extension, not part of the core GL command set. If **GL_EXT_texture_object** is included in the string returned by **glGetString** (when called with argument **GL_EXTENSIONS**), extension **EXT_texture_object** is supported by the connection.

Errors

GL_INVALID_VALUE is generated if *n* is negative.

GL_INVALID_VALUE is generated if any element in *textures* is zero or does not name a texture.

GL_INVALID_OPERATION is generated if **glAreTexturesResidentEXT** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glGetTexParameter with parameter name **GL_TEXTURE_RESIDENT_EXT** retrieves the residence status of a currently-bound texture.

Files

/usr/include/GL/glext.h	Contains extensions to C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBindTextureEXT** subroutine, **glPrioritizeTexturesEXT** subroutine, **glTexImage1D** subroutine, **glTexImage2D** subroutine, **glTexParameter** subroutine.

glArrayElement Subroutine

Purpose

Renders a vertex using the specified vertex array element.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glArrayElement(GLint  i)
```

Description

The **glArrayElement** commands are used within **glBegin**/**glEnd** pairs to specify vertex and attribute data for point, line, and polygon primitives. If **GL_VERTEX_ARRAY** is enabled when **glArrayElement** is called, a single vertex is drawn, using vertex and attribute data taken from location i of the enabled arrays. If **GL_VERTEX_ARRAY** is not enabled, no drawing occurs but the attributes correspoinding to the enabled arrays are modified.

Use **glArrayElement** to construct primitives by indexing vertex data, rather than by streaming through arrays of data in first-to-last order. Because each call specifies only a single vertex, it is possible to explicitly specify per- primitive attributes, such as a single normal per individual triangle.

Changes made to array data between the execution of **glBegin** and the corresponding execution of **glEnd** may affect calls to **glArrayElement** that are made within the same glBegin/glEnd period in non-sequential ways. That is, a call to **glArrayElement** that precedes a change to array data may access the changed data, and a call that follows a change to array data may access original data.

Parameters

i	Specifies an index into the enabled vertex data arrays.

Notes

The **glArrayElement** subroutine is available only if the GL version is 1.1 or greater.

The **glArrayElement** subroutine is included in display lists. If **glArrayElement** is entered into 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, their values affect display lists when the lists are created, not when the lists are executed.

Related Information

The **glClientActiveTextureARB** subroutine, **glColorPointer** subroutine, **glDrawArrays** subroutine, **glEdgeFlagPointer** subroutine, **glGetPointerv** subroutine, **glIndexPointer** subroutine, **glInterleavedArrays** subroutine, **glNormalPointer** subroutine, **glTexCoordPointer** subroutine, **glVertexPointer** subroutine.

glArrayElementEXT Subroutine

Purpose

Specifies the array elements used to render a vertex.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

void glArrayElementEXT(GLint *i* **)**

Description

The **glArrayElementEXT** commands are used within **glBegin**/**glEnd** pairs to specify vertex and attribute data for point, line and polygon primitives. When **glArrayElementEXT** is called, a single vertex is drawn, using vertex and attribute data taken from location *i* of the enabled arrays.

Use **glArrayElementEXT** to construct primitives by indexing vertex data, rather than by streaming through arrays of data in first-to-last order. Because each call specifies only a single vertex, it is possible to explicitly specify perprimitive attributes, such as a single normal per individual triangle.

Parameters

i	Specifies an index in the enabled arrays.

Notes

The **glArrayElementEXT** subroutine may be included in display lists. If **glArrayElementEXT** is entered into 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, their values affect display lists when the lists are created, not when the lists are executed.

Static array data may be read and cached by the implementation at any time. If static array elements are modified and the arrays are not respecified, the results of any subsequent calls to **glArrayElementEXT** are undefined.

The **glArrayElementEXT** subroutine executes even if **GL_VERTEX_ARRAY_EXT** is not enabled. No drawing occurs in this case, but the attributes corresponding to enabled arrays are modified.

Although it is not an error to respecify an array between the execution of **glBegin** and the corresponding execution of **glEnd**, the result of such respecification is undefined.

The **glArrayElementEXT** subroutine is part of the _extname(EXT_vertex_array) extension, not part of the core GL command set. If _extstring(EXT_vertex_array) is included in the string returned by **glGetString**, when called with argument **GL_EXTENSIONS**, extension _extname(EXT_vertex_array) is supported.

File

/usr/include/GL/glext.h	Contains extensions to C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glClientActiveTextureARB** subroutine, **glColorPointerEXT** subroutine, **glDrawArraysEXT** subroutine, **glEdgeFlagPointerEXT** subroutine, **glGetPointervEXT** subroutine, **glIndexPointerEXT** subroutine, **glInterleavedArrays** subrou **glNormalPointerEXT** subroutine, **glTexCoordPointerEXT** subroutine, **glVertexPointerEXT** subroutine.

glBegin or glEnd Subroutine

Purpose

Delimits the vertices of a primitive or group of like primitives.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glBegin(GLenum  mode)
void glEnd(void)
```

Description

The **glBegin** and **glEnd** subroutines delimit the vertices that define a primitive or group of like primitives. The **glBegin** subroutine accepts a single argument that specifies which of 10 ways the vertices will be interpreted. Taking *n* as an integer count starting at 1 (one), and *N* as the total number of vertices specified, the interpretations are:

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 2*n*-1 and 2*n* 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 3*n*-2, 3*n*-1, and 3*n* 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 4*n*-3, 4*n*-2, 4*n*-1, and 4*n* 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 2*n*-1, 2*n*, 2*n*+2, and 2*n*+1 define quadrilateral *n*. *N*/2-1 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.

Only a subset of GL subroutines can be used between the **glBegin** and **glEnd** subroutines. The subroutines are: **glVertex**, **glColor**, **glIndex**, **glNormal**, **glTexCoord**, **glEvalCoord**, **glEvalPoint**, **glMaterial**, and **glEdgeFlag**. Also, it is acceptable to use **glCallList** or **glCallLists** to execute display lists that include only the preceding subroutines. If any other GL subroutine is called between the **glBegin** and **glEnd** subroutines, the error flag is set and the subroutine is ignored.

Regardless of the value chosen for *mode*, there is no limit to the number of vertices that can be defined between the **glBegin** and **glEnd** subroutines. 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 rest are drawn.

The minimum specification of vertices for each primitive is as follows: 1 for a point, 2 for a line, 3 for a triangle, 4 for a quadrilateral, and 3 for a polygon. Modes that require a certain multiple of vertices are: **GL_LINES** (2), **GL_TRIANGLES** (3), **GL_QUADS** (4), and **GL_QUAD_STRIP** (2).

Parameters

mode Specifies the primitive or primitives that will be created from vertices presented between **glBegin** and the subsequent **glEnd**. Ten symbolic constants are accepted: **GL_POINTS**, **GL_LINES**, **GL_LINE_STRIP**, **GL_LINE_LOOP**, **GL_TRIANGLES**, **GL_TRIANGLE_STRIP**, **GL_TRIANGLE_FAN**, **GL_QUADS**, **GL_QUAD_STRIP**, and **GL_POLYGON**.

Errors

INVALID_ENUM	Indicates that *mode* is set to an unaccepted value.
GL_INVALID_OPERATION	Indicates that a subroutine other than **glVertex**, **glColor**, **glIndex**, **glNormal**, **glTexCoord**, **glEvalCoord**, **glEvalPoint**, **glMaterial**, **glEdgeFlag**, **glCallList**, or **glCallLists** subroutine is called between **glBegin** and the corresponding **glEnd**.
GL_INVALID_OPERATION	Indicates that **glEnd** is called before the corresponding **glBegin** is called.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glArrayElement** subroutine, **glArrayElementEXT** subroutine, **glColor** subroutine, **glCallList** subroutine, **glCallLists** subroutine, **glEdgeFlag** subroutine, **glEvalCoord** subroutine, **glEvalPoint** subroutine, **glIndex** subroutine, **glMaterial** subroutine, **glNormal** subroutine, **glTexCoord** subroutine, **glVertex** subroutine.

glBindTexture Subroutine

Purpose

Binds a named texture to a texturing target.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glBindTexture(GLenum  target,
     GLuint  texture)
```

Description

The **glBindTexture** subroutine lets you create or use a named texture. Calling **glBindTexture** with *target* set to **GL_TEXTURE_1D**, **GL_TEXTURE_2D**, **GL_TEXTURE_3D**, or **GL_TEXTURE_3D_EXT** and *texture* set to the name of the new texture binds the texture name to the target. When a texture is bound to a target, the previous binding for that target is automatically broken.

Texture names are unsigned integers. The value zero is 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 (see glXCreateContext) of the current GL rendering context; two rendering contexts share texture names only if they also share display lists.

You can use **glGenTextures** to generate a set of new texture names.

When a texture is first bound, it assumes the dimensionality of its target: A texture first bound to **GL_TEXTURE_1D** becomes one-dimensional (1D), a texture first bound to **GL_TEXTURE_2D** becomes two-dimensional (2D), a texture first bound to **GL_TEXTURE_3D** becomes three-dimensional (3D), a texture first bound to **GL_TEXTURE_3D_EXT** becomes three-dimensional (3D). The state of a (1D) texture immediately after it is first bound is equivalent to the state of the default **GL_TEXTURE_1D** at GL initialization, and similarly for 2D and 3D textures.

While a texture is bound, GL operations on the target to which it is bound affect the bound texture, and queries of the target to which it is bound return state from the bound texture. If texture mapping of the dimensionality of the target to which a texture is bound is active, the bound texture is used. In effect, the texture targets become aliases for the textures currently bound to them, and the texture name zero refers to the default textures that were bound to them at initialization.

A texture binding created with **glBindTexture** remains active until a different texture is bound to the same target, or until the bound texture is deleted with **glDeleteTextures**.

Once created, a named texture may be rebound to the target of the matching 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 over performance, use **glPrioritizeTextures**.

The **glBindTexture** subroutine is included in display lists.

Parameters

target	Specifies the target to which the texture is bound. Must be either **GL_TEXTURE_1D**, **GL_TEXTURE_2D**, **GL_TEXTURE_3D**, or **GL_TEXTURE_3D_EXT** (EXT_texture3D).
texture	Specifies the name of a texture.

Errors

GL_INVALID_ENUM is generated if *target* is not one of the allowable values.

GL_INVALID_OPERATION is generated if *texture* has a dimensionality which doesn't match that of *target*.

GL_INVALID_OPERATION is generated if **glBindTexture** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glGet with argument **GL_TEXTURE_1D_BINDING**

glGet with argument **GL_TEXTURE_2D_BINDING**

glGet with argument **GL_TEXTURE_3D_BINDING**

glGet with argument **GL_TEXTURE_3D_BINDING_EXT**

Related Information

The **glAreTexturesResident** subroutine, **glDeleteTextures** subroutine, **glGenTextures** subroutine, **glGet** subroutine, **glGetTexParameter** subroutine, **glIsTexture** subroutine, **glPrioritizeTextures** subroutine, **glTexImage1D** subroutine, **glTexImage2D** subroutine, **glTexImage3DEXT** subroutine, **glTexParameter** subroutine.

---

glBindTextureEXT Subroutine

Purpose

Binds a named texture to a texturing target.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glBindTextureEXT(GLenum  target,
     GLuint  texture)
```

Description

glBindTextureEXT is part of the **EXT_texture_object** extension. This extension makes it possible to use named 1-, 2-dimensional textures in addition to the usual OpenGL texture targets designated by **GL_TEXTURE_1D**, **GL_TEXTURE_2D**, **GL_TEXTURE_3D_EXT**, etc.

Texture names are unsigned integers. The value zero is 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 (see **glXCreateContext**) of the current OpenGL rendering context; two rendering contexts will share texture names only if they also share display lists.

To create a named texture, simply bind a previously-unused texture name to one of the texture targets listed above. This can be accomplished by calling **glBindTextureEXT** with *target* set to the appropriate texture target, and *texture* set to the name of the new texture. When a texture is bound to a target, the previous binding for that target is automatically broken.

Note that **glGenTexturesEXT** may be used to generate a set of fresh texture names.

When a texture is first bound, it assumes the dimensionality of its target: A texture first bound to **GL_TEXTURE_1D** becomes one-dimensional (1D), a texture first bound to **GL_TEXTURE_2D** becomes two-dimensional (2D), a texture first bound to **GL_TEXTURE_3D_EXT** becomes three-dimensional (3D). The state of a (1D) texture immediately after it is first bound is equivalent to the state of the default **GL_TEXTURE_1D** at GL initialization, and similarly for 2D and 3D textures.

While a texture is bound, GL operations on the target towhich it is bound affect the bound texture, and queries of the target to which it is bound return state from the bound texture. If texture mapping of the dimensionality of the target to which a texture is bound is active, the bound texture is used. In effect, the texture targets become aliases for the textures currently bound to them, and the texture name zero refers to the default textures that were bound to them at initialization.

A texture binding created with **glBindTextureEXT** remains active until a different texture is bound to the same target, or until the bound texture is deleted with **glDeleteTexturesEXT**.

Once created, a named texture may be re-bound to the appropriate target as often as needed. It is usually much faster to bind an existing named texture to one of the texture targets using **glBindTextureEXT** than it is to reload the texture image using **glTexImage***. For additional control over performance, consider using **glPrioritizeTexturesEXT**.

glBindTextureEXT is included in display lists.

Parameters

target	The target to which the texture will be bound. Must be one of **GL_TEXTURE_1D**, **GL_TEXTURE_2D**, or **GL_TEXTURE_3D_EXT** (EXT_texture3D).
texture	The name of a texture.

Notes

glBindTextureEXT is part of the **EXT_texture_object** extension, not part of the core GL command set. If **GL_EXT_texture_object** is included in the string returned by **glGetString**, when called with argument **GL_EXTENSIONS**, extension **EXT_texture_object** is supported by the connection.

Errors

GL_INVALID_ENUM	Generated if *target* is not one of the allowable values.
GL_INVALID_OPERATION	Generated if *texture* has a dimensionality and it doesn't match that of *target*.
GL_INVALID_OPERATION	Generated if **glBindTextureEXT** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glGet with argument **GL_TEXTURE_1D_BINDING_EXT**

glGet with argument **GL_TEXTURE_2D_BINDING_EXT**

glGet with argument **GL_TEXTURE_3D_BINDING_EXT**

Files

/usr/include/GL/glext.h	Contains extensions to C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glDeleteTexturesEXT** subroutine, **glGenTexturesEXT** subroutine, **glGet** subroutine, **glGetTexParameter** subroutine, **glIsTexture** subroutine, **glTexImage1D** subroutine, **glTexImage2D** subroutine, **glTexImage3DEXT** subroutine, **glTexParameter** subroutine.

glBitmap Subroutine

Purpose

Draws a bitmap.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glBitmap(GLsizei  Width,
      GLsizei  Height,
      GLfloat  xOrigin,
      GLfloat  yOrigin,
      GLfloat  xMove,
      GLfloat  yMove,
      const GLubyte * Bitmap)
```

Description

A bitmap is a binary image. When drawn, the bitmap is positioned relative to the current raster position, and frame buffer pixels corresponding to 1's in the bitmap are written using the current raster color or index. Frame buffer pixels corresponding to 0's in the bitmap are not modified.

The **glBitmap** subroutine takes seven arguments. The first pair of arguments specify the width and height of the bitmap image. The second pair of arguments specify the location of the bitmap origin relative to the lower left corner of the bitmap image. The final pair of arguments specify *x* and *y* offsets to be added to the current raster position after the bitmap has been drawn. The final argument is a pointer to the bitmap image itself.

The bitmap image is interpreted like image data for the **glDrawPixels** subroutine, with *Width* and *Height* corresponding to the width and height arguments of that subroutine, and with *Type* set to **GL_BITMAP** and *Format* set to **GL_COLOR_INDEX**. Modes specified using the **glPixelStore** subroutine affect the interpretation of bitmap image data; modes specified using the **glPixelTransfer** subroutine do not.

If the current raster position is not valid, the **glBitmap** subroutine is ignored. Otherwise, the lower left corner of the bitmap image is positioned at the following window coordinates:

```
xw =  [xr - xo]
yw =  [yr - yo]
```

where (*x*r, *y*r) is the raster position, and (*x*o, *y*o) 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.

Parameters

Width	Specifies the pixel width of the bitmap image.
Height	Specifies the pixel height of the bitmap image.
xOrigin	Specifies the location of the *x* origin in the bitmap image. The *x* origin is measured from the lower left corner of the bitmap, with right and up being the positive axes.
yOrigin	Specifies the location of the *y* origin in the bitmap image. The y origin is measured from the lower left corner of the bitmap, with right and up being the positive axes.
xMove	Specifies the *x* offset to be added to the current raster position after the bitmap is drawn.
yMove	Specifies the *y* offset to be added to the current raster position after the bitmap is drawn.
Bitmap	Specifies the address of the bitmap image.

Errors

GL_INVALID_VALUE	Either *Width* or *Height* is negative.
GL_INVALID_OPERATION	The **glBitmap** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glBitmap** subroutine are as follows. (See the **glGet** subroutine for more information.)

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**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glDrawPixels** subroutine, **glPixelStore** subroutine, **glPixelTransfer** subroutine, **glRasterPos** subroutine.

glBlendColor Subroutine

Purpose

Sets the blend color. This subroutine is part of OpenGL 1.2 ARB Imaging subset extension.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glBlendColor(GLclampf   red,
                  GLclampf   green,
                  GLclampf   blue,
                  GLclampf   alpha)
```

Description

The **GL_BLEND_COLOR** may be used to calculate the source and destination blending factors. See **glBlendFunc** for a complete description of the blending operations. Initially the **GL_BLEND_COLOR** is set to (0, 0, 0, 0).

Parameters

red, green, blue, alpha	Specify the components of **GL_BLEND_COLOR**.

Notes

The **glBlendColor** subroutine is available only if the GL version is 1.1 or greater.

Errors

GL_INVALID_OPERATION	The **glBlendColor** is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

glGet with argument **GL_BLEND_COLOR**.

Related Information

The **glBlendFunc** subroutine, **glGetString** subroutine.

---

glBlendColorEXT Subroutine

Purpose

Sets the blend color. This subroutine is part of OpenGL 1.2 ARB Imaging subset extension.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glBlendColorEXT(GLclampf  red,
                     GLclampf  green,
                     GLclampf  blue,
                     GLclampf  alpha)
```

Description

The **GL_BLEND_COLOR_EXT** may be used to calculate the source and destination blending factors. See **glBlendFunc** for a complete description of the blending operations. Initially the **GL_BLEND_COLOR_EXT** is set to (0, 0, 0, 0).

Parameters

red, green, blue, alpha	Specify the components of **GL_BLEND_COLOR_EXT**.

Notes

The **glBlendColorEXT** subroutine is available only if the GL version is 1.1 or greater.

Errors

GL_INVALID_OPERATION	The **glBlendColorEXT** is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

glGet with argument **GL_BLEND_COLOR_EXT**.

Related Information

The **glBlendFunc** subroutine, **glGetString** subroutine.

glBlendEquation Subroutine

Purpose

Specifies the RGB color blend equation. This subroutine is part of the OpenGL 1.2 ARB Imaging subset.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

void glBlendEquation(GLenum *mode***)**

Description

Blending combines corresponding source and destination color components according to the blending operation specified by the mode. The blend equations are:

GL_FUNC_ADD	min(Cs*sf + Cd*df, 1)
GL_FUNC_SUBTRACT	max(Cs*sf - Cd*df, 0)
GL_FUNC_REVERSE_SUBTRACT	max(Cd*df - Cs*sf, 0)
GL_LOGIC_OP	Cs Lop Cd
GL_MIN	min(Cs, Cd)
GL_MAX	max(Cs, Cd)

where Cs and Cd are the source and destination color components, respectively; sf and df are the source and destination blending factors are specified by **glBlendFunc**; Lop is one of the 16 bitwise operators specified by **glLogicOp**.

Parameters

mode Specifies how source and destination RGBA color components are combined. The symbolic constants **GL_FUNC_ADD**, **GL_MIN**, **GL_MAX**, **GL_FUNC_SUBTRACT**, **GL_REVERSE_SUBTRACT** are accepted. The initial mode is **GL_FUNC_ADD**.

Notes

The mode **GL_LOGIC_OP** is part of the EXT_blend_logic_op extension, not part of the core GL command set. If GL_EXT_blend_logic_op is included in the string returned by **glGetString**, when called with argument **GL_EXTENSIONS**, extension EXT_blend_logic_op is supported by the connection.

Errors

GL_INVALID_ENUM	The *mode* parameter is not an accepted or supported value.
GL_INVALID_OPERATION	The **glBlendEquation** is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

glGet with argument **GL_BLEND_EQUATION**.

Related Information

The **glBlendFunc** subroutine, **glEnable** or **glDisable** subroutine, **glGet** subroutine, **glLogicOp** subroutine.

glBlendEquationEXT Subroutine

Purpose

Specifies the RGB color blend equation.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glBlendEquationEXT(GLenum  mode)
```

Description

Blending combines corresponding source and destination color components according to the blending operation specified by the mode. The blend equations are:

GL_FUNC_ADD_EXT	min(Cs*sf + Cd*df, 1)
GL_FUNC_SUBTRACT_EXT	max(Cs*sf - Cd*df, 0)
GL_FUNC_REVERSE_SUBTRACT_EXT	max(Cd*df - Cs*sf, 0)
GL_LOGIC_OP	Cs Lop Cd
GL_MIN_EXT	min(Cs, Cd)
GL_MAX_EXT	max(Cs, Cd)

where Cs and Cd are the source and destination color components, respectively; sf and df are the source and destination blending factors are specified by **glBlendFunc**; Lop is one of the 16 bitwise operators specified by **glLogicOp**.

Parameters

mode Specifies how source and destination RGBA color components are combined. The symbolic constants **GL_FUNC_ADD_EXT**, **GL_MIN_EXT**, **GL_MAX_EXT**, **GL_FUNC_SUBTRACT_EXT**, **GL_REVERSE_SUBTRACT_EXT** are accepted. The initial mode is **GL_FUNC_ADD_EXT**.

Notes

The modes **GL_FUNC_SUBTRACT_EXT** and **GL_FUNC_REVERSE_SUBTRACT_EXT** are part of the EXT_blend_subtract extension, not part of the core GL command set. If GL_EXT_blend_subtract is included in the string returned by **glGetString**, when called with argument **GL_EXTENSIONS**, extension EXT_blend_subtract is supported by the connection.

The mode **GL_LOGIC_OP** is part of the EXT_blend_logic_op extension, not part of the core GL command set. If GL_EXT_blend_logic_op is included in the string returned by **glGetString**, when called with argument **GL_EXTENSIONS**, extension EXT_blend_logic_op is supported by the connection.

The modes **GL_MIN_EXT** and **GL_MAX_EXT** are part of the EXT_blend_minmax extension, not part of the core GL command set. If GL_EXT_blend_minmax is included in the string returned by **glGetString**, when called with argument **GL_EXTENSIONS**, extension EXT_blend_minmax is supported by the connection.

Errors

GL_INVALID_ENUM	The *mode* parameter is not an accepted or supported value.
GL_INVALID_OPERATION	The **glBlendEquation** is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

glGet with argument **GL_BLEND_EQUATION_EXT**.

Related Information

The **glBegin** subroutine, **glBlendFunc** subroutine, **glEnable** or **glDisable** subroutine, **glGet** subroutine, **glGetString** subroutine, **glLogicOp** subroutine.

glBlendFunc Subroutine

Purpose

Specifies pixel arithmetic.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glBlendFunc(GLenum  SourceFactor,
                 GLenum  DestinationFactor)
```

Description

In RGB mode, pixels can be drawn using a function that blends the incoming (source) red, green, blue, and alpha (RGBA) values with the RGBA values that are already in the frame buffer (the destination values). By default, blending is disabled. Use the **glEnable** and **glDisable** subroutines with argument **GL_BLEND** to enable and disable blending.

When blending is enabled, **glBlendFunc** and **glBlendEquationEXT** determine the blending operation. *SourceFactor* and *DestinationFactor* specify the scaling rules used for scaling the source and destination color components, respectively. Each rule defines four scale factors, one each for red, green, blue, and alpha. The rules are described in the table below.

In the table and in subsequent equations, source color components are referred to as:

```
(Rs, Gs, Bs, As)
```

Destination color components are referred to as:

```
(Rd, Gd, Bd, Ad)
```

Constant color components are referred to as:

```
(Rc, Gc, Bc, Ac)
```

They are understood to have integer values between 0 (zero) and:

```
(kR, kG, kB, kA)
```

where

```
(kc = 2mc - 1)
(m R, m G, m B, m A)
```

represents the number of RGBA bit planes.

Source scale factors are referred to as:

```
(s R, s G, s B, s A)
```

Destination scale factors are referred to as:

```
(d R, d G, d B, d A)
```

The scale factors:

```
(fR, fG, fB, fA)
```

represent either source or destination factors. All scale factors have the 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) - (*R*d/*k*R, *G*d/*k*G, *B*d/*k*B, *A*d/*k*A)
GL_SRC_ALPHA	(*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_CONSTANT_COLOR	(*R*c/*k*R, *G*c/*k*G, *B*c/*k*B, *A*c/*k*A)
GL_ONE_MINUS_CONSTANT_COLOR	(1, 1, 1, 1) - (*R*c/*k*R, *G*c/*k*G, *B*c/*k*B, *A*c/*k*A)
GL_CONSTANT_ALPHA	(*A*c/*k*A, *A*c/*k*A, *A*c/*k*A, *A*c/*k*A)
GL_ONE_MINUS_CONSTANT_ALPHA	(1, 1, 1, 1) - (*A*c/*k*A, *A*c/*k*A, *A*c/*k*A, *A*c/*k*A)
GL_SRC_ALPHA_SATURATE	(*i*, *i*, *i*, 1)

i = min (*A*s, *k*A - *A*d)/*k*A

To determine the blended RGBA values of a pixel when drawing in RGB mode, the system uses the following equations:

```
Rd = min (kR, RssR + RddR)
Gd = min (kG, GssG + GddG)
Bd = min (kB, BssB + BddB)
Ad = min (kA, AssA + AddA)
```

Blending combines corresponding source and destination color components according to the blending operation specified by **GL_BLEND_EQUATION_EXT**. The blending operations are:

GL_BLEND_EQUATION_EXT	*Binary Operation*
GL_FUNC_ADD_EXT	*min(C*s *x s*C+*C*d *x d*C,*k*C)

GL_BLEND_EQUATION_EXT	*Binary Operation*
GL_FUNC_SUBTRACT_EXT	*max(C*s *x s*C-*C*d x *d*C*,0)*
GL_FUNC_REVERSE_SUBTRACT_EXT	*max*(*C*d x *d*C-*C*s *x s*C*,0*)
GL_LOGIC_OP	*C*s Lop *C*d
GL_MIN_EXT	*min*(*C*s*, C*d)
GL_MAX_EXT	*max*(*C*s*, C*d)

where *C* is the relevant color component (R, G, B, or A), *C*s and *C*d are the source and destination color components, respectively, *s*C and *s*D are the source and destination scale factors, respectively, and *Lop* is one of 16 bitwise operators specified by **glLogicOp**.

Despite the apparent precision of the preceding equations, blending arithmetic is not exactly specified, because blending operates with imprecise integer color values. However, a blend factor that should be equal to 1 is guaranteed not to modify its multiplicand, and a blend factor equal to 0 reduces its multiplicand to 0. Thus, for example, when *SourceFactor* is **GL_SRC_ALPHA**, *DestinationFactor* is **GL_ONE_MINUS_SRC_ALPHA**, and *A*s is equal to *k*A, the equations reduce to simple replacement:

```
Rd = Rs
Gd = Gs
Rd = Bs
Ad = As
```

Parameters

SourceFactor
: Specifies how the RGBA source-blending factors are computed. Thirteen 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**, **GL_CONSTANT_COLOR**, **GL_CONSTANT_COLOR_EXT**, **GL_ONE_MINUS_CONSTANT_COLOR**, **GL_ONE_MINUS_CONSTANT_COLOR_EXT**, **GL_CONSTANT_ALPHA**, **GL_CONSTANT_ALPHA_EXT**, **GL_ONE_MINUS_CONSTANT_ALPHA**, **GL_ONE_MINUS_CONSTANT_ALPHA_EXT**, and **GL_SRC_ALPHA_SATURATE**. These symbolic constants are defined in the Description section. The initial value is **GL_ONE**.

DestinationFactor
: Specifies how the RGBA destination-blending factors are computed. Twelve 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** , **GL_ONE_MINUS_DST_ALPHA**, **GL_CONSTANT_COLOR**, **GL_CONSTANT_COLOR_EXT**, **GL_ONE_MINUS_CONSTANT_COLOR**, **GL_ONE_MINUS_CONSTANT_COLOR_EXT**, **GL_CONSTANT_ALPHA**, **GL_CONSTANT_ALPHA_EXT**, **GL_ONE_MINUS_CONSTANT_ALPHA**, and **GL_ONE_MINUS_CONSTANT_ALPHA_EXT**. These symbolic constants are defined in the Description section. The initial value is **GL_ZERO**.

Notes

Incoming (source) alpha is correctly thought of as a material opacity, ranging from 1.0 (*K*A), representing complete opacity, to 0.0 (0), representing complete transparency.

When more than one color buffer is enabled for drawing, blending is done separately for each enabled buffer, using for destination color the contents of that buffer. (See the **glDrawBuffer** subroutine.)

Blending affects only RGB rendering. It is ignored by color index renderers.

The Source and destination factors **GL_CONSTANT_COLOR**, **GL_ONE_MINUS_CONSTANT_COLOR**, **GL_CONSTANT_ALPHA**, **GL_ONE_MINUS_CONSTANT_ALPHA**, and their **_EXT** versions are only valid if the ARB imaging subset is supported and/or the Blend Color extension.

Errors

GL_INVALID_ENUM	Either *SourceFactor* or *DestinationFactor* is set to an unaccepted value.
GL_INVALID_OPERATION	The **glBlendFunc** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glBlendFunc** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_BLEND_SRC**, **GL_BLEND_DST**, **GL_LOGIC_OP_MODE**, or **GL_BLEND_EQUATION_EXT**.

glIsEnabled with argument **GL_BLEND**

Examples

Transparency is best implemented using a blend function (**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 bit planes in the frame buffer.

The blend function operation (**GL_SRC_ALPHA**, **GL_ONE_MINUS_SRC_ALPHA**) is also useful for rendering antialiased points and lines in arbitrary order.

Polygon antialiasing is optimized using a blend function (**GL_SRC_ALPHA_SATURATE**, **GL_ONE**) with polygons sorted from nearest to farthest. (See the **glEnable** or **glDisable** subroutine and the **GL_POLYGON_SMOOTH** argument for information on polygon antialiasing.) Destination alpha bit planes, which must be present for this blend function to operate correctly, store the accumulated coverage.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glAlphaFunc** subroutine, **glBegin** or **glEnd** subroutine, **glClear** subroutine, **glDrawBuffer** subroutine, **glEnable** or **Disable** ubroutine, **glLogicOp** subroutine, **glStencilFunc** subroutine.

glBlendFuncSeparateEXT Subroutine

Purpose

Specifies separate RGB and Alpha blend factors.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
void glBlendFuncSeparateEXT(enum  sfactorRGB,
                            enum  dfactorRGB,
                            enum  sfactorAlpha,
                            enum  dfactorAlpha)
```

Description

Blending capability is extended by this function. It allows independent specification of the RGB and alpha blend factors for blend operations that require source and destination blend factors. It is not always desired that the blending used for RGB is also applied to alpha.

The accepted values for *sfactorRGB* and *sfactorAlpha* are:

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

GL_CONSTANT_COLOR (_EXT)

GL_ONE_MINUS_CONSTANT_COLOR (_EXT)

GL_CONSTANT_ALPHA (_EXT)

GL_ONE_MINUS_CONSTANT_ALPHA (_EXT)

GL_SRC_ALPHA_SATURATE

The accepted values for *sfactorRGB* and *sfactorAlpha* are:

GL_ZERO

GL_ONE

GL_SRC_COLOR

GL_ONE_MINUS_SRC_COLOR

GL_SRC_ALPHA

GL_ONE_MINUS_SRC_ALPHA

GL_DST_ALPHA

GL_ONE_MINUS_DST_ALPHA

GL_CONSTANT_COLOR (_EXT)

GL_ONE_MINUS_CONSTANT_COLOR (_EXT)

GL_CONSTANT_ALPHA (_EXT)

GL_ONE_MINUS_CONSTANT_ALPHA (_EXT)

GL_SRC_ALPHA_SATURATE

For further information on the mathematical function of each of these accepted values, see **glBlendFunc**.

Parameters

sfactorRGB	is the source blend factor for the RGB components.
sfactorAlpha	is the source blend factor for the Alpha component.
dfactorRGB	is the destination blend factor for the RGB components.
dfactorAlpha	is the destination blend factor for the Alpha component.

Notes

This subroutine is only valid if the **EXT_blend_func_separate** extension is defined.

GL_CONSTANT_COLOR (_EXT), **GL_ONE_MINUS_CONTANT_COLOR (_EXT)**, **GL_CONSTANT_ALPHA (_EXT)**, and **GL_ONE_MINUS_CONSTANT_ALPHA (_EXT)** are only valid if the **GL_EXT_blend_color** extension is defined.

The (_EXT) at the end of these values above indicates that the enum can be specified with or without the _EXT suffix, and behaves identically in both cases.

Error Codes

GL_INVALID_ENUM	is generated if any of **sfactorRGB**, **dfactorRGB**, **sfactorAlpha**, or **dfactorAlpha** are not accepted values.
GL_INVALID_OPERATION	is generated if **glBlendFuncSeparateEXT** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBlendFunc** sunbroutine.

glCallList Subroutine

Purpose

Executes a display list.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glCallList(GLuint  List)
```

Description

The **glCallList** subroutine causes the named display list to be executed. The subroutines saved in the display list are executed in order, just as if they were called without using a display list. If *List* has not been defined as a display list, **glCallList** is ignored.

The **glCallList** subroutine may appear inside a display list. To avoid the possibility of infinite recursion resulting from display lists calling one another, an implementation-dependent limit is placed on the the nesting level of display lists during display list execution. This limit is at least 64.

GL state is not saved and restored across a call to **glCallList**. Thus, changes made to GL state during the execution of a display list will remain after execution of the display list is completed. Use the **glPushAttrib**, **glPopAttrib**, **PushMatrix**, and **glPopMatrix** subroutines to preserve GL state across **glCallList** calls.

Parameters

List Specifies the integer name of the display list to be executed.

Notes

Display lists can be executed between a call to **glBegin** and the corresponding call to **glEnd**, as long as the display list includes only commands that are allowed in this interval.

Associated Gets

The associated get for the **glCallList** subroutine is as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_MAX_LIST_NESTING**

glIsList.

Files

/usr/include/GL/gl.h Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glCallLists** subroutine, **glDeleteLists** subroutine, **glGenLists** subroutine, **glNewList** subroutine, **glPushAttrib** or **glPopAttrib** subroutine, **glPushMatrix** or **glPopMatrix** subroutine.

---

glCallLists Subroutine

Purpose

Executes a list of display lists.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glCallLists(GLsizei  Number,
      GLenum  Type,
      const GLvoid * Lists)
```

Description

The **glCallLists** subroutine causes each display list in the list of names passed as lists to be executed. As a result, the commands 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** subroutine provides an efficient means for executing display lists. The *Number* parameter allows lists with various name formats to be accepted. The formats are:

GL_BYTE	*Lists* is treated as an array of signed bytes, each in the range -128 through 127.
GL_UNSIGNED_BYTE	*Lists* is treated as an array of unsigned bytes, each in the range 0 through 255.
GL_SHORT	*Lists* is treated as an array of signed 2-byte integers, each in the range -32,768 through 32,767.
GL_UNSIGNED_SHORT	*Lists* is treated as an array of unsigned 2-byte integers, each in the range 0 through 65,535.
GL_INT	*Lists* is treated as an array of signed 4-byte integers.
GL_UNSIGNED_INT	*Lists* is treated as an array of unsigned 4-byte integers.
GL_FLOAT	*Lists* is treated as an array of 4-byte floating-point values.
GL_2_BYTES	*Lists* 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	*Lists* 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 65,536 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	*Lists* 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 16,777,216 times the unsigned value of the first byte, plus 65,536 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.

The list of display list names is not null-terminated. Rather, the *Number* parameter specifies how many names are to be taken from *Lists*.

An additional level of indirection is made available with the **glListBase** subroutine, which specifies a signed offset that is added to each display list name specified in *Lists* before that display list is executed.

The **glCallLists** subroutine can appear inside a display list. To avoid the possibility of infinite recursion resulting from display lists calling one another, an implementation-dependent limit is placed on the the nesting level of display lists during display list execution. This limit must be at least 64.

GL state is not saved and restored across a call to **glCallLists**. Thus, changes made to GL state during the execution of the display lists remain after execution is completed. Use the **glPushAttrib**, **glPopAttrib**, **glPushMatrix**, and **glPopMatrix** subroutines to preserve GL state across **glCallLists** calls.

Parameters

Number	Specifies the number of display lists to be executed.
Type	Specifies the type of values in lists. Symbolic constants **GL_BYTE**, **GL_UNSIGNED_BYTE**, **GL_SHORT**, **GL_UNSIGNED_SHORT**, **GL_INT**, **GL_UNSIGNED_INT**, **GL_FLOAT**, **GL_2_BYTES**, **GL_3_BYTES**, and **GL_4_BYTES** are accepted.
Lists	Specifies 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*.

Notes

Display lists can be executed between a call to **glBegin** and the corresponding call to **glEnd**, as long as the display list includes only commands that are allowed in this interval.

Associated Gets

Associated gets for the **glCallLists** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_LIST_BASE**

glGet with argument **GL_MAX_LIST_NESTING**

glIsList.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glCallList** subroutine, **glDeleteLists** subroutine, **glGenLists** subroutine, **glListBase** subroutine, **glNewList** subroutine, **glPushAttrib** or **glPopAttrib** subroutine, **glPushMatrix** or **glPopMatrix** subroutine.

glClear Subroutine

Purpose

Clears buffers to preset values.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glClear(GLbitfield Mask)
```

Description

The **glClear** subroutine sets the bit plane area of the viewport to values previously selected by **glClearColor**, **glClearIndex**, **glClearDepth**, **glClearStencil** and **glClearAccum**. Multiple color buffers can be cleared 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. Alpha function, blend function, logical operation, stenciling, texture mapping, and z-buffering are ignored by **glClear**.

The **glClear** subroutine takes a single argument that is the bitwise OR of several values indicating which buffer is to be cleared.

The values are:

GL_COLOR_BUFFER_BIT	Indicates the buffers currently enabled for color writing.
GL_DEPTH_BUFFER_BIT	Indicates the depth buffer.

GL_ACCUM_BUFFER_BIT	Indicates the accumulation buffer.
GL_STENCIL_BUFFER_BIT	Indicates the stencil buffer.

The value to which each buffer is cleared depends on the setting of the clear value for that buffer.

glGet with argument **GL_COLOR_CLEAR_VALUE**

glGet with argument **GL_STENCIL_CLEAR_VALUE**.

Parameters

Mask Bitwise OR of masks that indicate the buffers to be cleared. The four masks are **GL_COLOR_BUFFER_BIT**, **GL_DEPTH_BUFFER_BIT**, **GL_ACCUM_BUFFER_BIT**, and **GL_STENCIL_BUFFER_BIT**.

Notes

If a buffer is not present, then a **glClear** directed at that buffer has no effect.

Errors

GL_INVALID_VALUE	A bit other than the four defined bits is set in *Mask*.
GL_INVALID_OPERATION	The **glClear** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glClear** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_ACCUM_CLEAR_VALUE**

glGet with argument **GL_DEPTH_CLEAR_VALUE**

glGet with argument **GL_INDEX_CLEAR_VALUE**

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glClearAccum** subroutine, **glClearColor** subroutine, **glClearDepth** subroutine, **glClearIndex** subroutine, **glClearStencil** subroutine, **glDrawBuffer** subroutine, **glScissor** subroutine.

glClearAccum Subroutine

Purpose

Specifies clear values for the accumulation buffer.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glClearAccum(GLfloat  Red,
     GLfloat  Green,
     GLfloat  Blue,
     GLfloat  Alpha)
```

Description

The **glClearAccum** subroutine specifies the red, green, blue, and alpha values used by the **glClear** subroutine to clear the accumulation buffer. Values specified by **glClearAccum** are clamped to the range [-1,1].

Parameters

Red	Specifies the red value used when the accumulation buffer is cleared. The default value is 0 (zero).
Green	Specifies the green value used when the accumulation buffer is cleared. The default value is 0.
Blue	Specifies the blue value used when the accumulation buffer is cleared. The default value is 0.
Alpha	Specifies the alpha value used when the accumulation buffer is cleared. The default value is 0.

Errors

GL_INVALID_OPERATION	The **glClearAccum** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glClearAccum** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_ACCUM_CLEAR_VALUE**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glClear** subroutine.

glClearColor Subroutine

Purpose

Specifies clear values for the color buffers.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glClearColor(GLclampf  Red,
     GLclampf  Green,
     GLclampf   Blue,
     GLclampf   Alpha)
```

Description

The **glClearColor** subroutine specifies the red, green, blue, and alpha values used by the **glClear** subroutine to clear the color buffers. Values specified by **glClearColor** are clamped to the range [0,1].

Parameters

Red	Specifies the red value used when the color buffer is cleared. The default value is 0 (zero).
Green	Specifies the green value used when the color buffer is cleared. The default value is 0.
Blue	Specifies the blue value used when the color buffer is cleared. The default value is 0.
Alpha	Specifies the alpha value used when the color buffer is cleared. The default value is 0.

Errors

GL_INVALID_OPERATION	The **glClearColor** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glClearColor** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_COLOR_CLEAR_VALUE**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glClear** subroutine.

glClearDepth Subroutine

Purpose

Specifies the clear value for the depth buffer.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glClearDepth(GLclampd  Depth)
```

Description

The **glClearDepth** subroutine specifies the depth value used by the **glClear** subroutine to clear the depth buffer. Values specified by **glClearDepth** are clamped to the range [0,1].

Parameters

Depth	Specifies the depth value used when the depth buffer is cleared. The default value is 0 (zero).

Errors

GL_INVALID_OPERATION	The **glClearDepth** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glClearDepth** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_DEPTH_CLEAR_VALUE**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glClear** subroutine.

---

glClearIndex Subroutine

Purpose

Specifies the clear value for the color index buffers.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glClearIndex(GLfloat  Clear)
```

Description

The **glClearIndex** subroutine specifies the index used by **glClear** to clear the color index buffers. The *Clear* parameter is not clamped. Rather, *Clear* 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 2m -1, where *m* is the number of bits in a color index stored in the frame buffer.

Parameters

Clear	Specifies the index used when the color index buffers are cleared. The default value is 0 (zero).

Errors

GL_INVALID_OPERATION The **glClearIndex** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glClearIndex** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_INDEX_CLEAR_VALUE**

glGet with argument **GL_INDEX_BITS**.

Files

/usr/include/GL/gl.h Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glClear** subroutine.

---

glClearStencil Subroutine

Purpose

Specifies the clear value for the stencil buffer.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glClearStencil(GLint  Stencil)
```

Description

The **glClearStencil** subroutine specifies the index used by **glClear** to clear the stencil buffer. The *Stencil* parameter is masked with 2m - 1, where *m* is the number of bits in the stencil buffer.

Parameters

Stencil Specifies the index used when the stencil buffer is cleared. The default value is 0 (zero).

Errors

GL_INVALID_OPERATION Indicates that **glClearStencil** is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glClearStencil** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_STENCIL_CLEAR_VALUE**

glGet with argument **GL_STENCIL_BITS**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glClear** subroutine.

glClientActiveTextureARB Subroutine

Purpose

Specify which texture unit is active.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
void glClientActiveTextureARB(GLenum  texture)
```

Description

glClientActiveTextureARB selects which texture unit's client state parameters will be modified by **glTexCoordPointer**, and enabled or disabled with **glEnableClientState** or **glDisableClientState**, respectively, when called with a parameter of **GL_TEXTURE_COORD_ARRAY**. The number of texture units an implementation supports is implementation dependent, but must be at least two. The texture parameter must be one of **GL_TEXTUREi_ARB**, where 0 <= i < **GL_MAX_TEXTURE_UNITS_ARB**. The initial value is **GL_TEXTURE0_ARB**.

Parameters

texture	specifies which texture unit to make active.

Notes

If the **GL_ARB_multitexture** extension is NOT present, then the number of texture units supported by the implementation is one, not two, as described above.

The following OpenGL subroutines will be routed to different texture units based on this call:
- **glEnableClientState (GL_TEXTURE_COORD_ARRAY)**
- **glDisableClientState (GL_TEXTURE_COORD_ARRAY)**
- **glInterleavedArrays**
- **glTexCoordPointer**
- **glTexCoordPointerEXT**
- **glTexCoordPointerListIBM**

Subroutine **glClientActiveTextureARB** is supported only if **GL_ARB_multitexture** is included in the string returned by **glGetString** when called with the argument **GL_EXTENSIONS**.

Error Codes

GL_INVALID_OPERATION is generated if *texture* is not one of the accepted values.

Files

/usr/include/GL/gl.h Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glActiveTextureARB** subroutine, the **glEnableClientState** or **glDisableClientState** subroutine, the **glMultiTexCoordARB** subroutine, the **glTexCoordPointer** subroutine.

glClipBoundingBoxIBM or glClipBoundingSphereIBM or glClipBoundingVerticesIBM Subroutine

Purpose

Determine whether the specified object is trivially accepted, trivially rejected, or clipped by the current set of clipping planes.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
GLenum glClipBoundingBoxIBM (GLfloat xmin,
                             GLfloat ymin,
                             GLfloat zmin,
                             GLfloat xmax,
                             GLfloat ymax,
                             GLfloat zmax)

GLenum glClipBoundingSphereIBM (GLfloat x,
                                GLfloat y,
                                GLfloat z,
                                GLfloat radius)

GLenum glClipBoundingVerticesIBM (GLint size,
                                  GLenum type,
                                  GLsizei stride,
                                  GLsizei count,
                                  GLvoid *data)
```

Description

These three new functions can be used by applications to determine if a complex object is fully outside, inside, or both outside and inside the clip volume (ie, view volume plus any enabled clipping planes). The complex object is generally defined by a simplified representation of the object. This extension provides for 3 different simplified object variants - a bounding box, a bounding sphere, and a set of bounding vertices.

These functions can not be inserted within a display list. If called while a display list is open, they are executed immediately.

An enable is also provided so that applications can directly update the clip volume hint without having to make a separate OpenGL function call.

See **GL_UPDATE_CLIP_VOLUME_HINT** under **glEnable**.

All functions return the results of the clip check. These results include:

GL_REJECT_IBM	Indicates that the bounding object is trivially rejected. Rendering the object will result in nothing being rendered.
GL_ACCEPT_IBM	Indicates that the bounding object is trivially accepted. Rendering the object should be entirely within the viewport and can be rendering without clipping.
GL_CLIP_IBM	Indicates that the bounding object is not trivially accepted or rejected. Implementations that don't support clip checking for all rendering enviroments can return CLIP_IBM for those unsupported environments.

Parameters

xmin,ymin,zmin	Specifies the minimum x,y and z modeling coordinates of the bounding box.
xmax,ymax,zmax	Specifies the maximum x,y and z modeling coordinates of the bounding box.
x,y,z	Specifies the center of the bounding sphere in modeling coordinates.
radius	Specifies the radius of the bounding sphere in modeling coordinates.
size	Specifies the number of coordinate components per vertex; must be 2, 3 or 4.
type	Specifies the data type for the data parameter. Symbolic constants **GL_SHORT**, **GL_INT**, **GL_FLOAT**, and **GL_DOUBLE** are accepted.
stride	Specifies the byte offset between consecutive vertexes. If stride is 0, the vertices are understood to be tightly packed in the array.
count	Specifies the number of vertices pointed to by the data parameter.
data	Specifies a pointer to the first coordinate of the vertex list.

Notes

These three functions are only available if the **GL_IBM_clip_check** extension is present.

Error Codes

GL_INVALID_value	is generated if size is not 2, 3, or 4.
GL_INVALID_ENUM	is generated if type is not one of the acceptable values.
GL_INVALID_value	is generated if count is negative.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

glClipPlane Subroutine

Purpose

Specifies a plane against which all geometry is clipped.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glClipPlane(GLenum  Plane,
     const GLdouble * Equation)
```

By default, all clipping planes are defined as (0,0,0,0) in eye coordinates and are disabled.

Parameters

Plane	Specifies which clipping plane 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	Specifies the address of an array of four double-precision floating-point values. These values are interpreted as a plane equation.

Description

Geometry is always clipped against the boundaries of a six-plane frustum in *x*, *y*, and *z*. The **glClipPlane** subroutine allows the specification of additional planes, not necessarily perpendicular to the *x*, *y*, or *z* axes, against which all geometry is clipped. Up to **GL_MAX_CLIP_PLANES** planes can be specified, where **GL_MAX_CLIP_PLANES** is at least 6 in all implementations. Because the resulting clipping region is the intersection of the defined half-spaces, it is always convex.

The **glClipPlane** subroutine specifies a half-space using a four-component plane equation. When **glClipPlane** is called, *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 0 (zero), the vertex is *in* with respect to that clipping plane. Otherwise it is *out*.

Clipping planes are enabled and disabled with **glEnable** and **glDisable**, called with the argument **GL_CLIP_PLANE***i*, where *i* is the plane number.

Notes

It is always the case that **GL_CLIP_PLANE***i* = **GL_CLIP_PLANE**0 + *i*.

Errors

GL_INVALID_ENUM	*Plane* is set to an unaccepted value.
GL_INVALID_OPERATION	The **glClipPlane** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glClipPlane** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGetClipPlane	
glIsEnabled	Enabled with argument **GL_CLIP_PLANE***i*.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL

Related Information

The **glBegin** or **glEnd** subroutine, **glEnable** or **glDisable** subroutine.

glColor Subroutine

Purpose

Sets the current color.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
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
-set the current color
void glColor3b

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(GLshort  Red,
     GLshort  Green,
     GLshort  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(GLshort  Red,
     GLshort  Green,
     GLshort  Blue,
     GLshort  Alpha)

void glColor3bv(const GLbyte * Variable)

void glColor3dv(const GLdouble * Variable)
```

```
void glColor3fv(const GLfloat * Variable)

void glColor3iv(const GLint * Variable)

void glColor3sv(const GLshort * Variable)

void glColor3ubv(const GLubyte * Variable)

void glColor3uiv(const GLuint * Variable)

void glColor3usv(const GLushort * Variable)

void glColor4bv(const GLbyte * Variable)

void glColor4dv(const GLdouble * Variable)

void glColor4fv(const GLfloat * Variable)

void glColor4iv(const GLint * Variable)

void glColor4sv(const GLshort * Variable)

void glColor4ubv(const GLubyte * Variable)

void glColor4uiv(const GLuint * Variable)

void glColorusv(const GLushort * Variable)
```

Description

The Graphics Library stores both a current single-valued color index and a current four-valued red, green, blue, alpha (RGBA) color. The **glColor** subroutine sets a new four-valued RGBA color. The **glColor** subroutine has two major variants: **glColor3** and **glColor4**. **glColor3** variants specify new red, green, and blue values explicitly, and set the current alpha value to 1.0 implicitly. **glColor4** variants specify all four color components explicitly.

glColor3b, **glColor4b**, **glColor3s**, **glColor4s**, **glColor3i**, and **glColor4i** take 3 or 4 unsigned byte, short, or long integers as arguments. When **v** is appended to the name, the color subroutines 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 0 (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 specified 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.

Parameters

Red	Specifies a red value for the current color. The initial value is 1 (one).
Green	Specifies a green value for the current color. The initial value is 1 (one).
Blue	Specifies a blue value for the current color. The initial value is 1 (one).

Alpha	Specifies a new alpha value for the current color. Included only in the four-argument **glColor** subroutine. The initial value is 1 (one).
Variable	Specifies a pointer to an array that contains red, green, blue, and (sometimes) alpha values.

Notes

The current color can be updated at any time. In particular, **glColor** can be called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glColor** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_CURRENT_COLOR**.

glGet with argument **GL_RGBA_MODE**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** subroutine, **glColorPointer** subroutine, **glColorPointerEXT** subroutine, **glEnd** subroutine, **glIndex** subroutine.

glColorMask Subroutine

Purpose

Enables and disables the writing of frame buffer color components.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glColorMask(GLboolean  Red,
     GLboolean  Green,
     GLboolean  Blue,
     GLboolean  Alpha)
```

Description

The **glColorMask** subroutine specifies whether the individual color components in the frame buffer can or cannot be written. If the *Red* parameter 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.

Parameters

Red	Specifies whether red can or cannot be written into the frame buffer. The default value is True, indicating that the red color component can be written.
Green	Specifies whether green can or cannot be written into the frame buffer. The default value is True, indicating that the green color component can be written.
Blue	Specifies whether blue can or cannot be written into the frame buffer. The default value is True, indicating that the blue color component can be written.
Alpha	Specifies whether alpha can or cannot be written into the frame buffer. The default value is True, indicating that the alpha color component can be written.

Errors

GL_INVALID_OPERATION	The **glColorMask** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glColorMask** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_COLOR_WRITEMASK**

glGet with argument **GL_RGBA_MODE**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glColor** subroutine, **glDepthMask** subroutine, **glIndex** subroutine, **glIndexMask** subroutine, **glStencilMask** subroutine.

glColorMaterial Subroutine

Purpose

Causes a material color to track the current color.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glColorMaterial(GLenum  face,
     GLenum  mode)
```

Description

The **glColorMaterial** subroutine specifies which material parameters track the current color. When **GL_COLOR_MATERIAL** is enabled, the material parameter or parameters specified by *mode*, of the material or materials specified by *face*, track the current color at all times. **GL_COLOR_MATERIAL** is enabled and disabled using the subroutines **glEnable** and **glDisable**, called with **GL_COLOR_MATERIAL** as their argument. By default it is disabled.

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 will 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**.

Notes

The **glColorMaterial** subroutine allows a subset of material parameters to be changed for each vertex using only the **glColor** subroutine, without calling **glMaterial**. If only such a subset of parameters is to be specified for each vertex, the use of the **glColorMaterial** subroutine is preferred over calling **glMaterial**.

Calling **glDrawElements** may leave the current color indeterminate. If **glColorMaterial** is enabled while the current color is indeterminate, the lighting material state specified by *face* and *mode* is also indeterminate.

Errors

GL_INVALID_ENUM	*face* or *mode* is set to an unaccepted value.
GL_INVALID_OPERATION	The **glColorMaterial** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glColorMaterial** subroutine are as follows. (See the **glGet** subroutine for more information.)

glIsEnabled with argument **GL_COLOR_MATERIAL**

glGet with argument **GL_COLOR_MATERIAL_PARAMETER**

glGet with argument **GL_COLOR_MATERIAL_FACE**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glColor** subroutine, **glEnable** or **glDisable** subroutine, **glLight** subroutine, **glLightModel** subroutine, **glMaterial** subroutine.

glColorNormalVertexSUN Subroutine

Purpose

Specifies a color, a normal and a vertex in one call.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
void glColor4fNormal3fVertex3fSUN (GLfloat  r,
                                   GLfloat  g,
                                   GLfloat  b,
                                   GLfloat  a,
                                   GLfloat  nx,
                                   GLfloat  ny,
                                   GLfloat  nz,
                                   GLfloat  x,
                                   GLfloat  y,
                                   GLfloat  z)
void glColor4fNormal3fVertex3fvSUN (const GLfloat  *c,
                                    const GLfloat  *n,
                                    const GLfloat  *v)
```

Description

This subroutine can be used as a replacement for the following calls:

```
glColor();
glNormal();
glVertex();
```

For example, **glColor4fNormal3fVertex3fvSUN** replaces the following calls:

```
glColor4f();
glNormal3f();
glVertex3fv();
```

The only reason for using this call is that it reduces the use of bus bandwidth.

Parameters

r, g, b, a	specifies *r*, *g*, *b*, and *a* components of the color for this vertex.
c	specifies a pointer to an array of the four components *r*, *g*, *b*, and *a*.
nx, ny, nz	specifies *x*, *y*, and *z* coordinates of the normal vector for this vertex.
n	specifies a pointer to an array of the three elements *nx*, *ny* and *nz*.
x, y, z	specifies the *x*, *y*, and *z* coordinates of a vertex. Not all parameters are present in all forms of the command.
v	specifies a pointer to an array of the three elements *x*, *y*, and *z*.

Notes

Calling **glColorNormalVertexSUN** outside of a **glBegin**/**glEnd** subroutine pair results in undefined behavior.

This subroutine is only valid if the **GL_SUN_vertex** extension is defined.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, the **glColor** subroutine, the **glNormal** subroutine, the **glTexCoord** subroutine, the **glVertex** subroutine.

---

glColorPointer Subroutine

Purpose

Defines an array of colors.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glColorPointer( GLint  size,
   GLenum  type,
   GLsizei  stride,
   const GLvoid * pointer)
```

Description

The **glColorPointer** subroutine specifies the location and data format of an array of color components to use when rendering. The *size* parameter specifies the number of components per color, and must be 3 or 4. The *type* parameter specifies the data type of each color component and *stride* gives the byte stride from one color to the next allowing vertices and attributes to be packed into a single array or stored in separate arrays. (Single-array storage may be more efficient on some implementations; see **glInterleavedArrays**).

When a color array is specified, *size*, *type*, *stride*, and *pointer* are saved as client side state.

To enable and disable the color array, call **glEnableClientState** and **glDisableClientState** with the argument **GL_COLOR_ARRAY**. If enabled, the color array is used when **glDrawArrays**, **glDrawElements** or **glArrayElement** is called.

Use **glDrawArrays**, **glMultiDrawArraysEXT**, or **glMultiModeDrawArraysIBM** to construct a sequence of primitives from prespecified vertex and vertex attribute arrays. Use **glArrayElement** to specify primitives by indexing vertices and vertex attributes. Use **glDrawElements**, **glMultiDrawElementsEXT**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

If enabled, the Color array is used when **glDrawArrays**, **glDrawElements**, **glArrayElements**, **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** is called.

Parameters

size	Specifies the number of components per color. It must be 3 or 4. The initial value is 4.
type	Specifies the data type of each color component in the array. Symbolic constants **GL_BYTE**, **GL_UNSIGNED_BYTE**, **GL_SHORT**, **GL_UNSIGNED_SHORT**, **GL_INT**, **GL_UNSIGNED_INT**, **GL_FLOAT**, or **GL_DOUBLE** are accepted. The initial value is GL_FLOAT.
stride	Specifies the byte offset between consecutive colors. If *stride* is zero (the initial value), the colors are understood to be tightly packed in the array. The initial value is 0.
pointer	Specifies a pointer to the first component of the first color element in the array. The initial value is 0 (NULL pointer).

Notes

The **glColorPointer** subroutine is available only if the GL version is 1.1 or greater.

The color array is initially disabled and it won't be accessed when **glArrayElement**, **glDrawElements**, or **glDrawArrays** is called.

Execution of **glColorPointer** is not allowed between **glBegin** and the corresponding **glEnd**, but an error may or may not be generated. If an error is not generated, the operation is undefined.

The **glColorPointer** subroutine is typically implemented on the client side with no protocol.

Since the color array parameters are client side state, they are not saved or restored by **glPushAttrib** and **glPopAttrib**. Use **glPushClientAttrib** and **glPopClientAttrib** instead.

The **glColorPointer** commands are not included in display lists.

Error Codes

GL_INVALID_VALUE is generated if size is not 3 or 4.

GL_INVALID_ENUM is generated if type is not an accepted value.

GL_INVALID_VALUE is generated if stride is negative.

Associated Gets

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**.

glGetPointerv with argument **GL_COLOR_ARRAY_POINTER**.

Related Information

The **glArrayElement** subroutine, **glColorPointerListIBM** subroutine, **glDrawArrays** subroutine, **glDrawElements** subroutine, **glEdgeFlagPointer** subroutine, **glEnable** subroutine, **glGetPointerv** subroutine, **glIndexPointer** subroutine, **glInterleavedArrays** subroutine, **glNormalPointer** subroutine, **glPopClientAttrib** subroutine, **glPushClientAttrib** subroutine, **glTexCoordPointer** subroutine, **glVertexPointer** subroutine.

glColorPointerEXT Subroutine

Purpose

Defines an array of colors.

Library

OpenGL and OpenGL C bindings library: **libGL.a**

C Syntax

```
void glColorPointerEXT(GLint  size,
     GLenum  type,
```

```
GLsizei stride,
GLsizei count,
const GLvoid *pointer)
```

Description

The **glColorPointerEXT** subroutine specifies the location and data format of an array of color components to use when rendering. *size* specifies the number of components per color, and must be 3 or 4. The *type* parameter specifies the data type of each color component and *stride* gives the byte stride from one color to the next allowing vertexes and attributes to be packed into a single array or stored in separate arrays. (Single-array storage may be more efficient on some implementations). The *count* parameter indicates the number of array elements (counting from the first) that are static. Static elements may be modified by the application, but once they are modified, the application must explicitly respecify the array before using it for any rendering. When a color array is specified, *size*, *type*, *stride*, *count* and *pointer* are saved as client-side state, and static array elements may be cached by the implementation.

The color array is enabled and disabled using **glEnable** and **glDisable** with the argument **GL_COLOR_ARRAY_EXT**. If enabled, the color array is used when **glDrawArraysEXT** or **glArrayElementEXT** is called.

Use **glDrawArrays**, **glMultiDrawArraysEXT**, or **glMultiModeDrawArraysIBM** to construct a sequence of primitives from prespecified vertex and vertex attribute arrays. Use **glArrayElement** to specify primitives by indexing vertices and vertex attributes. Use **glDrawElements**, **glMultiDrawElementsEXT**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

If enabled, the Color array is used when **glDrawArrays**, **glDrawElements**, **glArrayElements**, **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** is called.

Parameters

size	Specifies the number of components per color. It must be 3 or 4.
type	Specifies the data type of each color component in the array. Symbolic constants **GL_BYTE**, **GL_UNSIGNED_BYTE**, **GL_SHORT**, **GL_UNSIGNED_SHORT**, **GL_INT**, **GL_UNSIGNED_INT**, **GL_FLOAT**, or **GL_DOUBLE_EXT**, are accepted.
stride	Specifies the byte offset between consecutive colors. If *stride* is zero the colors are understood to be tightly packed in the array.
count	Specifies the number of colors, counting from the first, that are static.
pointer	Specifies a pointer to the first component of the first color element in the array.

Notes

Non-static array elements are not accessed until **glArrayElementEXT** or **glDrawArraysEXT** is executed.

By default the color array is disabled and it won't be accessed when **glArrayElementEXT** or **glDrawArraysEXT** is called.

Although, it is not an error to call **glColorPointerEXT** between the execution of **glBegin** and the corresponding execution of **glEnd**, the results are undefined.

glColorPointerEXT will typically be implemented on the client side with no protocol.

Since the color array parameters are client side state, they are not saved or restored by **glPushAttrib** and **glPopAttrib**.

glColorPointerEXT commands are not entered into display lists.

glColorPointerEXT is part of the _extname(EXT_vertex_array) extension, not part of the core GL command set. If _extstring(EXT_vertex_array) is included in the string returned by **glGetString**, when called with argument **GL_EXTENSIONS**, extension _extname(EXT_vertex_array) is supported.

Errors

GL_INVALID_VALUE is generated if *size* is not 3 or 4.

GL_INVALID_ENUM is generated if *type* is not an accepted value.

GL_INVALID_VALUE is generated if *stride* or *count* is negative.

Associated Gets

glIsEnabled with argument **GL_COLOR_ARRAY_EXT**.

glGet with argument **GL_COLOR_ARRAY_SIZE_EXT**.

glGet with argument **GL_COLOR_ARRAY_TYPE_EXT**.

glGet with argument **GL_COLOR_ARRAY_STRIDE_EXT**.

glGet with argument **GL_COLOR_ARRAY_COUNT_EXT**.

glGetPointervEXT with argument **GL_COLOR_ARRAY_POINTER_EXT**.

File

/usr/include/GL/glext.h	Contains extensions to C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glArrayElement** subroutine, **glDrawArraysEXT** subroutine, **glEdgeFlagPointerEXT** subroutine, **glGetPointervEXT** subroutine, **glIndexPointerEXT** subroutine, **glNormalPointerEXT** subroutine, **glTexCoordPointerEXT** subroutine, **glVertexPointerEXT** subroutine.

glColorPointerListIBM Subroutine

Purpose

Defines a list of color arrays.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glColorPointerListIBM ( GLint  size,
   GLenum  type,
   GLint  stride,
   const GLvoid ** pointer,
   GLint  ptrstride)
```

Description

The **glColorPointerListIBM** subroutine specifies the location and data format of a list of arrays of color components to use when rendering. The *size* parameter specifies the number of components per color, and must be 3 or 4. The *type* parameter specifies the data type of each color component. The *stride* parameter gives the byte stride from one color to the next allowing vertices and attributes to be packed into a single array or stored in separate arrays. (Single-array storage may be more efficient on some implementations; see **glInterleavedArrays**). The *ptrstride* parameter specifies the byte stride from one pointer to the next in the *pointer* array.

When a color array is specified, *size*, *type*, *stride*, *pointer* and *ptrstride* are saved as client side state.

A *stride* value of 0 does not specify a ″tightly packed″ array as it does in **glColorPointer**. Instead, it causes the first array element of each array to be used for each vertex. Also, a negative value can be used for stride, which allows the user to move through each array in reverse order.

To enable and disable the color arrays, call **glEnableClientState** and **glDisableClientState** with the argument **GL_COLOR_ARRAY**. The color array is initially disabled. When enabled, the color arrays are used when **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, **glDrawArrays**, **glDrawElements** or **glArrayElement** is called. The last three calls in this list will only use the first array (the one pointed at by *pointer*[0]). See the descriptions of these routines for more information on their use.

Use **glDrawArrays**, **glMultiDrawArraysEXT**, or **glMultiModeDrawArraysIBM** to construct a sequence of primitives from prespecified vertex and vertex attribute arrays. Use **glArrayElement** to specify primitives by indexing vertices and vertex attributes. Use **glDrawElements**, **glMultiDrawElementsEXT**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

If enabled, the Color array is used when **glDrawArrays**, **glDrawElements**, **glArrayElements**, **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** is called.

Parameters

size	Specifies the number of components per color. It must be 3 or 4. The initial value is 4.
type	Specifies the data type of each color component in the array. Symbolic constants **GL_BYTE**, **GL_UNSIGNED_BYTE**, **GL_SHORT**, **GL_UNSIGNED_SHORT**, **GL_INT**, **GL_UNSIGNED_INT**, **GL_FLOAT**, or **GL_DOUBLE** are accepted. The initial value is GL_FLOAT.
stride	Specifies the byte offset between consecutive colors. The initial value is 0.
pointer	Specifies a list of color arrays. The initial value is 0 (NULL pointer).
ptrstride	Specifies the byte stride between successive pointers in the *pointer* array. The initial value is 0.

Notes

The **glColorPointerListIBM** subroutine is available only if the GL_IBM_vertex_array_lists extension is supported.

Execution of **glColorPointerListIBM** is not allowed between **glBegin** and the corresponding **glEnd**, but an error may or may not be generated. If an error is not generated, the operation is undefined.

The **glColorPointerListIBM** subroutine is typically implemented on the client side.

Since the color array parameters are client side state, they are not saved or restored by **glPushAttrib** and **glPopAttrib**. Use **glPushClientAttrib** and **glPopClientAttrib** instead.

When a **glColorPointerListIBM** call is encountered while compiling a display list, the information it contains does NOT contribute to the display list, but is used to update the immediate context instead.

The **glColorPointer** call and the **glColorPointerListIBM** call share the same state variables. A **glColorPointer** call will reset the color list state to indicate that there is only one color list, so that any and all lists specified by a previous **glColorPointerListIBM** call will be lost, not just the first list that it specified.

Error Codes

GL_INVALID_VALUE is generated if size is not 3 or 4.

GL_INVALID_ENUM is generated if type is not an accepted value.

Associated Gets

glIsEnabled with argument **GL_COLOR_ARRAY**.

glGetPointerv with argument **GL_COLOR_ARRAY_LIST_IBM**.

glGet with argument **GL_COLOR_ARRAY_LIST_STRIDE_IBM**.

glGet with argument **GL_COLOR_ARRAY_SIZE**.

glGet with argument **GL_COLOR_ARRAY_STRIDE**.

glGet with argument **GL_COLOR_ARRAY_TYPE**.

Related Information

The **glArrayElement** subroutine, **glColorPointer** subroutine, **glDrawArrays** subroutine, **glDrawElements** subroutine, **glEdgeFlagPointer** subroutine, **glEnable** subroutine, **glGetPointerv** subroutine, **glIndexPointer** subroutine, **glInterleavedArrays** subroutine, **glMultiDrawArraysEXT** subroutine, **glMultiDrawElementsEXT** subroutine, **glMultiModeDrawArraysIBM** subroutine, **glMultiModeDrawElementsIBM** subroutine, **glNormalPointer** subroutine, **glPopClientAttrib** subroutine, **glPushClientAttrib** subroutine, **glTexCoordPointer** subroutine, **glVertexPointer** subroutine.

glColorSubTable Subroutine

Purpose

Define a contiguous subset of a color lookup table.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
void glColorTable(GLenum  target,
                  GLsizei  start,
                  GLsizei  count,
                  GLenum  format,
                  GLenum  type,
                  const  GLvoid *data)
void glColorTableEXT(GLenum  target,
                     GLsizei  start,
                     GLsizei  count,
```

```
GLenum  format,
GLenum  type,
const GLvoid  *data)
```

Description

glColorSubTable is used to respecify a contiguous portion of a color table previously defined using **glColorTable**. The pixels reference by *data* replace the portion of the existing table from indices *start* to *start* + *count* - 1, inclusive. This region may not include any entries outside the range of the color table as it was originally specified. It is not an error to specify a subtable with width of 0, but such a specification has no effect.

Parameters

target	must be **GL_TEXTURE_COLOR_TABLE_EXT**.
start	is the starting index of the portion of the color table to be replaced.
count	is the number of table entries to replace.
format	is the format of the pixel data in *data*. The allowable values are **GL_RED**, **GL_GREEN**, **GL_BLUE**, **GL_ALPHA**, **GL_LUMINANCE**, **GL_LUMINANCE_ALPHA**, **GL_RGB**, **GL_BGR**, **GL_RGBA** and **GL_BGRA**.
type	is the type of the pixel data in *table*. The allowable values are **GL_UNSIGNED_BYTE, GL_BYTE, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, GL_FLOAT, GL_UNSIGNED_BYTE_3_3_2, GL_UNSIGNED_BYTE_2_3_3_REV, GL_UNSIGNED_SHORT_5_6_5, GL_UNSIGNED_SHORT_5_6_5_REV, GL_UNSIGNED_SHORT_4_4_4_4, GL_UNSIGNED_SHORT_4_4_4_4_REV, GL_UNSIGNED_SHORT_5_5_5_1, GL_UNSIGNED_SHORT_1_5_5_5_REV, GL_UNSIGNED_INT_8_8_8_8, GL_UNSIGNED_INT_8_8_8_8_REV, GL_UNSIGNED_INT_10_10_10_2,** and **GL_UNSIGNED_INT_2_10_10_10_REV**.
data	is a pointer to a one-dimensional array of pixel data that is processed to replace the specified region of the color table.

Notes

GL_TEXTURE_COLOR_TABLE_SGI is an alias for **GL_TEXTURE_COLOR_TABLE_EXT**, and these tokens may be used interchangeably. **GL_PROXY_TEXTURE_COLOR_TABLE_SGI** is an alias for **GL_PROXY_TEXTURE_COLOR_TABLE_EXT**, and these tokens may be used interchangeably.

Error Codes

GL_INVALID_ENUM	is generated if *target* is not one of the allowable values.
GL_INVALID_VALUE	is generated if *start* + *count* > *width*, where *width* is the width of the previously defined color table.
GL_INVALID_ENUM	is generated if *format* is not one of the allowable values.
GL_INVALID_ENUM	is generated if *type* is not one of the allowable values.
GL_INVALID_OPERATION	is generated if **glColorSubTable** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

Associated gets for the **glColorSubTable** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with arguement **glGetColorTableParameter**.

glGet with arguement **glGetColorTable**.

Files

/usr/include/GL/gl.h Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glColorTable** subroutine, the **glColorTableParameter** subroutine, the **glCopyColorTable** subroutine, the **glCopyColorSubTable** subroutine, the **glGetColorTable** subroutine.

glColorTable Subroutine

Purpose

Define a color lookup table.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
void glColorTable(GLenum  target,
                  GLenum  internalformat,
                  GLsizei  width,
                  GLenum  format,
                  GLenum  type,
                  const GLvoid  *table)
void glColorTableSGI(GLenum  target,
                     GLenum  internalformat,
                     GLsizei  width,
                     GLenum  format,
                     GLenum  type,
                     const GLvoid  *table)
```

Description

glColorTable may be used in two ways: to test the actual size and color resolution of a lookup table given a particular set of parameters, or to load the contents of a color lookup table. Use the targets **GL_PROXY_*** for the first case and the other targets for the second case.

If *target* is **GL_TEXTURE_COLOR_TABLE_EXT**, **glColorTable** builds a color lookup table from an array of pixels. The pixel array specified by *width*, *format*, *type*, and *table* is extracted from memory and processed just as if **glDrawPixels** were called, but processing stops after the final expansion to RGBA is completed.

The four scale parameters and the four bias parameters that are defined for the table are then used to scale and bias the R, G, B, and A components of each pixel. (Use **glColorTableParameter** to set these scale and bias parameters).

Next, the R, G, B, and A values are clamped to the range [0, 1]. Each pixel is then converted to the internal format specified by *internalformat*. This conversion simply maps the component values of the pixel (R, G, B, and A) to the values included in the internal format (red, green, blue, alpha, and intensity). The mapping is as follows:

Internal Format	Red	Green	Blue	Alpha	Luminance	Intensity
GL_ALPHA				A		

GL_LUMINANCE					R	
GL_LUMINANCE_ALPHA				A	R	
GL_INTENSITY						R
GL_RGB	R	G	B			
GL_RGBA	R	G	B	A		

Finally, the red, green, blue, alpha, luminance, and/or intensity components of the resulting pixels are stored in the color table. They form a one-dimensional table with indices in the range [0, *width*-1].

If *target* is **GL_PROXY_TEXTURE_COLOR_TABLE_EXT**, **glColorTable** recomputes and stores the values of the proxy color table's state variables **GL_COLOR_TABLE_FORMAT**, **GL_COLOR_TABLE_WIDTH**, **GL_COLOR_TABLE_RED_SIZE**, **GL_COLOR_TABLE_GREEN_SIZE**, **GL_COLOR_TABLE_BLUE_SIZE**, **GL_COLOR_TABLE_ALPHA_SIZE**, **GL_COLOR_TABLE_LUMINANCE_SIZE**, and **GL_COLOR_TABLE_INTENSITY_SIZE**. There is no effect on the image or state of any actual color table. If the specified color table is too large to be supported, then all the proxy state variables listed above are set to zero. Otherwise, the color table could be supported by **glColorTable** using the corresponding non-proxy target, and the proxy state variable are set as if that target were being defined.

The proxy state variables can be retrieved by calling **glGetColorTableParameter** with a target of **GL_PROXY_***. This allows the application to decide what the resulting color table attributes would be.

If a color table is enabled, and its width is non-zero, then its contents are used to replace a subset of the components of each RGBA pixel group, based on the internal format of the table.

Each pixel group has color components (R, G, B, A) that are in the range [0.0, 1.0]. The color components are rescaled to the size of the color lookup table to form an index. Then a subset of the components based on the internal format of the table are replaced by the table entry specified by that index. If the color components and contents of the table are represented as follows:

Representation	Meaning
r	Table index computed from R
g	Table index computed from G
b	Table index computed from B
a	Table index computed from A
L[i]	Luminance value at table index i
I[i]	Intensity value at table index i
R[i]	Red value at table index i
G[i]	Green value at table index i
B[i]	Blue value at table index i
A[i]	Alpha value at table index i

then the result of color table lookup is as follows:

	Resulting Color Components			
Table Internal Format	**R**	**G**	**B**	**A**
GL_ALPHA	R	G	B	A[a]
GL_LUMINANCE	L[r]	L[g]	L[b]	A
GL_LUMINANCE_ALPHA	L[r]	L[g]	L[b]	A[a]
GL_INTENSITY	I[r]	I[g]	I[b]	I[a]
GL_RGB	R[r]	G[g]	B[b]	A
GL_RGBA	R[r]	G[g]	B[b]	A[a]

Parameters

target must be **GL_TEXTURE_COLOR_TABLE_EXT** or **GL_PROXY_TEXTURE_COLOR_TABLE_EXT**.

internalformat	is the internal format of the color table. The allowable values are **GL_ABGR_EXT**, **GL_ALPHA**, **GL_ALPHA4**, **GL_ALPHA8**, **GL_ALPHA12**, **GL_ALPHA16**, **GL_LUMINANCE**, **GL_LUMINANCE4**, **GL_LUMINANCE8**, **GL_LUMINANCE12**, **GL_LUMINANCE16**, **GL_LUMINANCE_ALPHA**, **GL_LUMINANCE4_ALPHA4**, **GL_LUMINANCE6_ALPHA2**, **GL_LUMINANCE8_ALPHA8**, **GL_LUMINANCE12_ALPHA4**, **GL_LUMINANCE12_ALPHA12**, **GL_LUMINANCE16_ALPHA16**, **GL_INTENSITY**, **GL_INTENSITY4**, **GL_INTENSITY8**, **GL_INTENSITY12**, **GL_INTENSITY16**, **GL_R3_G3_B2**, **GL_RGB**, **GL_RGB4**, **GL_RGB5**, **GL_RGB8**, **GL_RGB10**, **GL_RGB12**, **GL_RGB16**, **GL_RGBA**, **GL_RGBA2**, **GL_RGBA4**, **GL_RGB5_A1**, **GL_RGB8**, **GL_RGB10_A2**, **GL_RGBA12,** and **GL_RGB16**.
width	is the number of entries in the color lookup table specified by *table*.
format	is the format of the pixel data in *table*. The allowable values are **GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_RGB, GL_BGR, GL_RGBA**, **GL_BGRA**, **GL_422_EXT**, **GL_422_REV_EXT**, **GL_422_AVERAGE_EXT**, and **GL_422_REV_AVERAGE_EXT**.
type	is the type of the pixel data in *table*. The allowable values are **GL_UNSIGNED_BYTE, GL_BYTE, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, GL_FLOAT, GL_UNSIGNED_BYTE_3_3_2, GL_UNSIGNED_BYTE_2_3_3_REV, GL_UNSIGNED_SHORT_5_6_5, GL_UNSIGNED_SHORT_5_6_5_REV, GL_UNSIGNED_SHORT_4_4_4_4, GL_UNSIGNED_SHORT_4_4_4_4_REV, GL_UNSIGNED_SHORT_5_5_5_1, GL_UNSIGNED_SHORT_1_5_5_5_REV, GL_UNSIGNED_INT_8_8_8_8, GL_UNSIGNED_INT_8_8_8_8_REV, GL_UNSIGNED_INT_10_10_10_2,** and **GL_UNSIGNED_INT_2_10_10_10_REV**.
table	is pointer to a one-dimensional array of pixel data that is processed to build the color table.

Notes

GL_ABGR_EXT is only valid if the **GL_EXT_abgr** extension is defined.

GL_TEXTURE_COLOR_TABLE_SGI is an alias for **GL_TEXTURE_COLOR_TABLE_EXT**, and these tokens may be used interchangeably. **GL_PROXY_TEXTURE_COLOR_TABLE_SGI** is an alias for **GL_PROXY_TEXTURE_COLOR_TABLE_EXT**, and these tokens may be used interchangeably.

Error Codes

GL_INVALID_ENUM	is generated if *target* is not one of the allowable values.
GL_INVALID_ENUM	is generated if *internalformat* is not one of the allowable values.
GL_INVALID_VALUE	is generated if *width* is less than zero.
GL_INVALID_VALUE	is generated if *target* is set to **GL_TEXTURE_COLOR_TABLE_EXT** and *width* is not a power of two.
GL_INVALID_ENUM	is generated if *format* is not one of the allowable values.
GL_INVALID_ENUM	is generated if *type* is not one of the allowable values.
GL_TABLE_TOO_LARGE	is generated if the requested color table is too large to be supported by the implementation, and *target* is not a **GL_PROXY_*** target.
GL_INVALID_OPERATION	is generated if **glColorTable** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

Associated gets for the **glColorTable** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with arguement **glGetColorTableParameter**.

glGet with arguement **glGetColorTable**.

Files

/usr/include/GL/gl.h Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glColorSubTable** subroutine, the **glColorTableParameter** subroutine, the **glCopyColorTable** subroutine, the **glCopyColorSubTable** subroutine, the **glGetColorTable** subroutine.

glColorTableParameter Subroutine

Purpose

Specify attributes to be used when loading a color table.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
void glColorTableParameterfv(GLenum  target,
                     GLenum  pname,
                     const GLfloat  *params)

void glColorTableParameteriv(GLenum  target,
                     GLenum  pname,
                     const GLint  *params)

void glColorTableParameterfvSGI(GLenum  target,
                        GLenum  pname,
                        const GLfloat  *params)

void glColorTableParameterivSGI(GLenum  target,
                        GLenum  pname,
                        const GLint  *params)
```

Description

glColorTableParameter is used to specify the scale factors and bias terms applied to color components when they are loaded into a color table. *target* indicates which color table the scale or bias terms apply to.

If *pname* is set to **GL_COLOR_TABLE_SCALE**, then the four values pointed to by *params* will be stored as the red, green, blue and alpha scale factors, in that order.

If *pname* is set to **GL_COLOR_TABLE_BIAS**, then the four values pointed to by *params* will be stored as the red, green, blue and alpha bias terms, in that order.

Parameters

target is the target color table and must be **GL_TEXTURE_COLOR_TABLE_EXT**.

pname is the symbolic name of a texture color lookup table parameter. Must be **GL_COLOR_TABLE_SCALE** or **GL_COLOR_TABLE_BIAS**.

params is a pointer to an array where the values of the paramaters are stored.

Notes

GL_TEXTURE_COLOR_TABLE_SGI is an alias for **GL_TEXTURE_COLOR_TABLE_EXT**, and these tokens may be used interchangeably.

GL_PROXY_TEXTURE_COLOR_TABLE_SGI is an alias for **GL_PROXY_TEXTURE_COLOR_TABLE_EXT**, and these tokens may be used interchangeably.

Error Codes

GL_INVALID_ENUM is generated if *target* is not one of the allowable values.

GL_INVALID_ENUM is generated if *pname* is not one of the allowable values.

GL_INVALID_OPERATION is generated if **glColorTable** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

Associated gets for the **glColorTableParameter** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with arguement **glGetColorTableParameter**.

Files

/usr/include/GL/gl.h Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glPixelTransfer** subroutine, the **glColorTable** subroutine.

---

glColorVertexSUN Subroutine

Purpose

Specifies a color and a vertex in one call.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
void glColor3fVertex3fSUN (GLfloat  r,
                           GLfloat  g,
                           GLfloat  b,
                           GLfloat  x,
                           GLfloat  y,
                           GLfloat  z)
void glColor3fVertex3fvSUN (const GLfloat  *c,
                            const GLfloat  *v)
void glColor4ubVertex2fSUN (GLubyte  r,
```

```
                            GLubyte  g,
                            GLubyte  b,
                            GLubyte  a,
                            GLfloat  x,
                            GLfloat  y)
void glColor4ubVertex2fvSUN (const GLubyte  *c,
                             const GLfloat  *v)
void glColor4ubVertex3fSUN (GLubyte  r,
                            GLubyte  g,
                            GLubyte  b,
                            GLubyte  a,
                            GLfloat  x,
                            GLfloat  y,
                            GLfloat  z)
void glColor4ubVertex3fvSUN (const GLubyte  *c,
                             const GLfloat  *v)
```

Description

This subroutine can be used as a replacement for the following calls:

```
glColor();
glVertex();
```

For example, **glColor4ubVertex3fvSUN** replaces the following calls:

```
glColor4ub();
glVertex3fv();
```

The only reason for using this call is that it reduces the use of bus bandwidth.

Parameters

x, y, z	Specifies the *x*, *y*, and *z* coordinates of a vertex. Not all parameters are present in all forms of the command.
v	Specifies a pointer to an array of two, or three elements. The elements of a two-element array are *x* and *y*. The elements of a three-element array are *x*, *y*, and *z*.
r, g, b, a	Specifies the red, green, blue, and alpha components of a color. Not all parameters are present in all forms of the command.
c	Specifies a pointer to an array of three or four elements. The elements of a three-element array are *r*, *g*, and *b*. The elements of a four-element array are *r*, *g*, *b*, and *a*.

Notes

Calling **glColorVertexSUN** outside of a **glBegin**/**glEnd** subroutine pair results in undefined behavior.

This subroutine is only valid if the **GL_SUN_vertex** extension is defined.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, the **glColor** subroutine, the **glNormal** subroutine, the **glTexCoord** subroutine, the **glVertex** subroutine.

glCopyColorSubTable Subroutine

Purpose

Load a subset of a color lookup table from the current **GL_READ_BUFFER**.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
void glCopyColorSubTable(GLenum target,
                         GLsizei start,
                         GLint x,
                         GLint y,
                         GLsizei width)
void glCopyColorSubTableSGI(GLenum target,
                            GLsizei start,
                            GLint x,
                            GLint y,
                            GLsizei width)
```

Description

glCopyColorSubTable is used to respecify a contiguous portion of a color table previously defined using **glColorTable**. The pixels copied from the framebuffer replace the portion of the existing table from indices *start* to *start* + *x* - 1, inclusive. This region may not include any entries outside the range of the color table as it was originally specified. It is not an error to specify a subtexture with width of 0, but such a specification has no effect.

Parameters

target	Must be **GL_TEXTURE_COLOR_TABLE_EXT**.
start	is the starting index of the portion of the color table to be replaced.
x, y	is the window coordinates of the left end of the row of pixels to be copied.
width	is the width of the pixel rectangle.

Notes

GL_TEXTURE_COLOR_TABLE_SGI is an alias for **GL_TEXTURE_COLOR_TABLE_EXT**, and these tokens may be used interchangeably. **GL_PROXY_TEXTURE_COLOR_TABLE_SGI** is an alias for **GL_PROXY_TEXTURE_COLOR_TABLE_EXT**, and these tokens may be used interchangeably.

Error Codes

GL_INVALID_ENUM	is generated if *target* is not one of the allowable values.
GL_INVALID_VALUE	is generated if *width* is less than zero.

GL_INVALID_OPERATION is generated if **glCopyColorSubTable** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

Associated gets for the **glColorTable** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with arguement **glGetColorTableParameter**.

glGet with arguement **glGetColorTable**.

Files

/usr/include/GL/gl.h Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glColorSubTable** subroutine, the **glColorTableParameter** subroutine, the **glCopyColorTable** subroutine, the **glGetColorTable** subroutine.

glCopyColorTable Subroutine

Purpose

Load a color lookup table from the current **GL_READ_BUFFER**.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
void glCopyColorTable(GLenum  target,
                      GLenum  internalformat,
                      GLint  x,
                      GLint  y,
                      GLsizei  width)
void glCopyColorTableSGI(GLenum  target,
                         GLenum  internalformat,
                         GLint  x,
                         GLint  y,
                         GLsizei  width)
```

Description

glCopyColorTable loads a color table with pixels from the current **GL_READ_BUFFER** (rather than from main memory, as is the case for **glColorTable**).

The screen-aligned pixel rectangle with lower-left corner at (*x*, *y*) having width *width* and height 1 is loaded into the color table. If any pixels within this region are outside the window that is associated with the GL context, the values obtained for those pixels are undefined

The pixels in the rectangle are processed just as if **glReadPixels** were called, with *internalformat* set to RGBA, but processing stops after the final conversion to RGBA.

The four scale parameters and the four bias parameters that are defined for the table are then used to scale and bias the R, G, B, and A components of each pixel. (Use **glColorTableParameter** to set these scale and bias parameters).

Next, the R, G, B, and A values are clamped to the range [0, 1]. Each pixel is then converted to the internal format specified by *internalformat*. This conversion simply maps the component values of the pixel (R, G, B, and A) to the values included in the internal format (red, green, blue, alpha, and intensity). The mapping is as follows:

Internal Format	Red	Green	Blue	Alpha	Luminance	Intensity
GL_ALPHA				A		
GL_LUMINANCE					R	
GL_LUMINANCE_ALPHA				A	R	
GL_INTENSITY						R
GL_RGB	R	G	B			
GL_RGBA	R	G	B	A		

Finally, the red, green, blue, alpha, luminance, and/or intensity components of the resulting pixels are stored in the color table. They form a one-dimensional table with indices in the range [0, *width*-1].

Parameters

target
: Must be **GL_TEXTURE_COLOR_TABLE_EXT** or **GL_PROXY_TEXTURE_COLOR_TABLE_EXT**.

internalformat
: is the internal format of the color table. The allowable values are: **GL_ABGR_EXT**, **GL_ALPHA**, **GL_ALPHA4**, **GL_ALPHA8**, **GL_ALPHA12**, **GL_ALPHA16**, **GL_LUMINANCE**, **GL_LUMINANCE4**, **GL_LUMINANCE8**, **GL_LUMINANCE12**, **GL_LUMINANCE16**, **GL_LUMINANCE_ALPHA**, **GL_LUMINANCE4_ALPHA4**, **GL_LUMINANCE6_ALPHA2**, **GL_LUMINANCE8_ALPHA8**, **GL_LUMINANCE12_ALPHA4**, **GL_LUMINANCE12_ALPHA12**, **GL_LUMINANCE16_ALPHA16**, **GL_INTENSITY**, **GL_INTENSITY4**, **GL_INTENSITY12**, **GL_INTENSITY16**, **GL_R3_G3_B2**, **GL_RGB**, **GL_RGB4**, **GL_RGB5**, **GL_RGB8**, **GL_RGB10**, **GL_RGB12**, **GL_RGB16**, **GL_RGBA**, **GL_RGBA2**, **GL_RGBA4**, **GL_RGB5_A1**, **GL_RGB8**, **GL_RGB10_A2**, **GL_RGBA12,** and **GL_RGB16**.

width
: The width of the pixel rectangle.

x
: is the x coordinate of the lower-left corner of the pixel rectangle to be transferred to the color table.

y
: is the y coordinate of the lower-left corner of the pixel rectangle to be transferred to the color table.

table
: is a pointer to a one-dimensional array of pixel data that is processed to build the color table.

Notes

GL_ABGR_EXT is only valid if the **GL_EXT_abgr** extension is defined.

GL_TEXTURE_COLOR_TABLE_SGI is an alias for **GL_TEXTURE_COLOR_TABLE_EXT**, and these tokens may be used interchangeably. **GL_PROXY_TEXTURE_COLOR_TABLE_SGI** is an alias for **GL_PROXY_TEXTURE_COLOR_TABLE_EXT**, and these tokens may be used interchangeably.

Error Codes

GL_INVALID_ENUM	is generated if *target* is not one of the allowable values.
GL_INVALID_ENUM	is generated if *internalformat* is not one of the allowable values.
GL_INVALID_VALUE	is generated if *width* is less than zero.
GL_TABLE_TOO_LARGE	is generated if the requested color table is too large to be supported by the implementation.
GL_INVALID_OPERATION	is generated if **glCopyColorTable** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

Associated gets for the **glColorTable** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with arguement **glGetColorTableParameter**.

glGet with arguement **glGetColorTable**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glColorTable** subroutine, the **glColorTableParameter** subroutine, the **glCopyColorSubTable** subroutine, the **glGetColorTable** subroutine.

glCopyPixels Subroutine

Purpose

Copies pixels in the frame buffer.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glCopyPixels(GLint  xCoordinate,
   GLint  yCoordinate,
   GLsizei  Width,
   GLsizei  Height,
   GLenum  Type)
```

Description

The **glCopyPixels** subroutine 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 numbers.

Several parameters control the processing of the pixel data while it is being copied. These parameters are set with three subroutines: **glPixelTransfer**, **glPixelMap**, and **glPixelZoom**. This article describes the effects on **glCopyPixels** of most, but not all, of the parameters specified by these three subroutines.

The **glCopyPixels** subroutine copies values from each pixel with lower left corner at (*x* + *i*, *y* + *j*) for 0 <= *i* <*Width* and 0 <= *j* <*Height*. This pixel is said to be the *i*th pixel in the *j*th 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 details of the transfer for each data type are as follows.

GL_COLOR
: Indices or red, green, blue, alpha (RGBA) colors are read from the buffer currently specified as the read source buffer. (See the **glReadBuffer** subroutine.) If the GL is in color index mode, 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. Each 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, 0 (zero) bits fill otherwise unspecified bit locations in the result. 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**. Whether the lookup replacement of the index is done or not, the integer part of the index is then ANDed with 2b -1, where *b* is the number of bits in a color index buffer.

  If the GL is in RGBA mode, the red, green, blue, and alpha components of each pixel that is read are converted to an internal floating-point format with unspecified precision. The conversion maps the largest representable component value to 1.0, and component value 0 to 0.0. 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. The results are clamped to the range [0,1]. If **GL_MAP_COLOR** is True, each color component is scaled by the size of the lookup table **GL_PIXEL_MAP_*c*_TO_*c***, then replaced by the value that it references in that table. *c* is **R**, **G**, **B**, or **A**, respectively.

  The resulting indices or RGBA colors are then converted to fragments by attaching the current raster position *z* coordinate 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 *i*th pixel in the *j*th 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 *i*th pixel in the *j*th 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 indices 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, 0 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 the 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 ANDed with 2b -1, where *b* is the number of bits in the stencil buffer. The resulting stencil indices are then written to the stencil buffer such that the index read from the *i*th location of the *j*th 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.

The rasterization described thus far assumes pixel zoom factors of 1.0. If **glPixelZoom** is used 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*th location in the *j*th row of the source pixel rectangle, fragments are generated for pixels whose centers are in the rectangle with corners at

Unmapped format: variant of paragraph
(*x*r + *zoom*x *i*, *y*r + *zoom*y *j*)

Unmapped format: variant of paragraph
and
Unmapped format: variant of paragraph
(*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**.

Parameters

xCoordinate Specifies the *x* window coordinate of the lower left corner of the rectangular region of pixels to be copied.

yCoordinate Specifies the *y* window coordinate of the lower left corner of the rectangular region of pixels to be copied.

Width Specifies the width of the rectangular region of pixels to be copied. This parameter does not accept a negative value.

Height Specifies the height of the rectangular region of pixels to be copied. This parameter does not accept a negative value.

Type Specifies whether color values, depth values, or stencil values are to be copied. Symbolic constants **GL_COLOR**, **GL_DEPTH**, and **GL_STENCIL** are accepted.

Notes

Modes specified by **glPixelStore** have no effect on the operation of **glCopyPixels**.

Errors

GL_INVALID_ENUM	*Type* is not an accepted value.
GL_INVALID_VALUE	Either *Width* or *Height* is negative.
GL_INVALID_OPERATION	*Type* is **GL_DEPTH** and there is no depth buffer.
GL_INVALID_OPERATION	*Type* is **GL_STENCIL** and there is no stencil buffer.
GL_INVALID_OPERATION	The **glCopyPixels** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glCopyPixels** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_CURRENT_RASTER_POSITION**.

glGet with argument **GL_CURRENT_RASTER_POSITION_VALID**.

Examples

To copy the color pixel in the lower left corner of the window to the current raster position, enter the following:

```
glCopyPixels(0, 0, 1, 1, GL_COLOR);
```

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glDepthFunc** subroutine, **glDrawBuffer** subroutine, **glDrawPixels** subroutine, **glPixelMap** subroutine, **glPixelTransfer** subroutine, **glPixelZoom** subroutine, **glRasterPos** subroutine, **glReadBuffer** subroutine, **glReadPixels** subroutine, **glStencilFunc** subroutine.

---

glCopyTexImage1D Subroutine

Purpose

Defines a one-dimensional (1D) texture image.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glCopyTexImage1D(GLenum  target,
     GLint  level,
     GLenum  internalFormat,
     GLint  xCoordinate,
     GLint  yCoordinate,
     GLsizei  width,
     GLint  border)
```

Description

The **glCopyTexImage1D** subroutine defines a one dimensional texture image with pixels from the current **GL_READ_BUFFER**.

The screen aligned pixel row with left corner at (x,y) and with a length of width + 2 * border defines the texture array at the mipmap level specified by level. IlternalFormat specifies the internal format of the texture array.

The pixels in the row are processed exactly as if **glCopyPixels** had been called, but the process stops just before final conversion. At this point all pixel component values are clamped to the range [0, 1] and then converted to the texture's internal format for storage in the texel array.

Pixel ordering is such that lower x screen coordinates correspond to lower texture coordinates.

If any of the pixels within the specified row of the current **GL_READ_BUFFER** are outside the window associated with the current rendering context, then the values obtained for those pixels are undefined.

Parameters

target	Specifies the target texture. Must be **GL_TEXTURE_1D**.
level	Specifies the level-of-detail number. Level 0 is the base image level. Level n is the nth mipmap reduction image.
internalFormat	Specifies the internal format of the texture. Must be one of the following symbolic constants: **GL_ABGR_EXT**, **GL_ALPHA**, **GL_ALPHA4**, **GL_ALPHA8**, **GL_ALPHA12**, **GL_ALPHA16**, **GL_LUMINANCE**, **GL_LUMINANCE4**, **GL_LUMINANCE8**, **GL_LUMINANCE12**, **GL_LUMINANCE16**, **GL_LUMINANCE_ALPHA**, **GL_LUMINANCE4_ALPHA4**, **GL_LUMINANCE6_ALPHA2**, **GL_LUMINANCE8_ALPHA8**, **GL_LUMINANCE12_ALPHA4**, **GL_LUMINANCE12_ALPHA12**, **GL_LUMINANCE16_ALPHA16**, **GL_INTENSITY**, **GL_INTENSITY4**, **GL_INTENSITY8**, **GL_INTENSITY12**, **GL_INTENSITY16**, **GL_RGB**, **GL_R3_G3_B2**, **GL_RGB4**, **GL_RGB5**, **GL_RGB8**, **GL_RGB10**, **GL_RGB12**, **GL_RGB16**, **GL_RGBA**, **GL_RGBA2**, **GL_RGBA4**, **GL_RGB5_A1**, **GL_RGBA8**, **GL_RGB10_A2**, **GL_RGBA12**, or **GL_RGBA16**.
xCoordinate	Specifies the *x* window coordinate of the lower left corner of the row of pixels to be copied.
yCoordinate	Specifies the *y* window coordinate of the lower left corner of the row of pixels to be copied.
width	Specifies the width of the texture image. Must be 0 or 2**n + 2*border for some integer n. The height of the texture image is 1.
border	Specifies the width of the border. Must be either 0 or 1.

Notes

GL_ABGR_EXT is only valid if the **GL_EXT_abgr** extension is defined.

The **glCopyTexImage1D** subroutine is available only if the GL version is 1.1 or greater.

1, 2, 3, or 4 are not accepted values for **internalFormat**.

An image with zero width indicates a null texture.

Errors

GL_INVALID_ENUM is generated if target is not one of the allowable values.

GL_INVALID_VALUE is generated if level is less than zero.

GL_INVALID_VALUE may be generated if level is greater than log2max, where max is the returned value of **GL_MAX_TEXTURE_SIZE**.

GL_INVALID_VALUE is generated if border is not 0 or 1.

GL_INVALID_VALUE is generated if width is less than zero, greater than 2 + **GL_MAX_TEXTURE_SIZE**, or if width cannot be represented as 2**k+ 2 * border for some integer k.

GL_INVALID_VALUE is generated if width is less than zero or greater than 2 + **GL_MAX_TEXTURE_SIZE**, or if it cannot be represented as 2**n + 2 * border for some integer value of n.

GL_INVALID_OPERATION is generated if **glCopyTexImage1D** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glGetTexImage

glIsEnabled with argument **GL_TEXTURE_1D**

Related Information

The **glCopyTexImage2D** subroutine, **glDrawPixels** subroutine, **glFog** subroutine, **glPixelStore** subroutine, **glPixelTransfer** subroutine, **glTexEnv** subroutine, **glTexGen** subroutine, **glTexImage1D** subroutine, **glTexParameter** subroutine.

---

glCopyTexImage2D Subroutine

Purpose

Defines a two-dimensional (2D) texture image.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glCopyTexImage2D(GLenum  target,
     GLint   level,
     GLenum   internalFormat,
     GLint   xCoordinate,
     GLint   yCoordinate,
     GLsizei   width,
     GLsizei   height,
     GLint   border)
```

Description

The **glCopyTexImage2D** subroutine defines a two-dimensional texture image with pixels from the current **GL_READ_BUFFER**.

The screen aligned pixel rectangle with lower left corner at (x, y) and with a width of width + 2 * border and height height + 2 * border defines the texture array at the mipmap level specified by level. **internalFormat** specifies the internal format of the texture array.

The pixels in the rectangle are processed exactly as if **glCopyPixels** had been called, but the process stops just before final conversion. At this point all pixel component values are clamped to the range [0.0,1.0] and then converted to the texture's internal format for storage in the texel array.

Pixel ordering is such that lower x and y screen coordinates correspond to lower s and t texture coordinates.

If any of the pixels within the specified rectangle of the current **GL_READ_BUFFER** are outside the window associated with the current rendering context, then the values obtained for those pixels are undefined.

Parameters

target	Specifies the target texture. Must be **GL_TEXTURE_2D**.
level	Specifies the level-of-detail number. Level 0 is the base image level. Level n is the nth mipmap reduction image.
internalFormat	Specifies the internal format of the texture. Must be one of the following symbolic constants: **GL_ABGR_EXT**, **GL_ALPHA**, **GL_ALPHA4**, **GL_ALPHA8**, **GL_ALPHA12**, **GL_ALPHA16**, **GL_LUMINANCE**, **GL_LUMINANCE4**, **GL_LUMINANCE8**, **GL_LUMINANCE12**, **GL_LUMINANCE16**, **GL_LUMINANCE_ALPHA**, **GL_LUMINANCE4_ALPHA4**, **GL_LUMINANCE6_ALPHA2**, **GL_LUMINANCE8_ALPHA8**, **GL_LUMINANCE12_ALPHA4**, **GL_LUMINANCE12_ALPHA12**, **GL_LUMINANCE16_ALPHA16**, **GL_INTENSITY**, **GL_INTENSITY4**, **GL_INTENSITY8**, **GL_INTENSITY12**, **GL_INTENSITY16**, **GL_RGB**, **GL_R3_G3_B2**, **GL_RGB4**, **GL_RGB5**, **GL_RGB8**, **GL_RGB10**, **GL_RGB12**, **GL_RGB16**, **GL_RGBA**, **GL_RGBA2**, **GL_RGBA4**, **GL_RGB5_A1**, **GL_RGBA8**, **GL_RGB10_A2**, **GL_RGBA12**, or **GL_RGBA16**.
xCoordinate	Specifies the *x* window coordinate of the lower left corner of the row of pixels to be copied.
yCoordinate	Specifies the *y* window coordinate of the lower left corner of the row of pixels to be copied.
width	Specifies the width of the texture image. Must be 0 or 2**n + 2*border for some integer n. The height of the texture image is 1.
height	Specifies the height of the texture image. Must be 0 or 2**m + 2*border for some integer m.
border	Specifies the width of the border. Must be either 0 or 1.

Notes

GL_ABGR_EXT is only valid if the **GL_EXT_abgr** extension is defined.

The **glCopyTexImage2D** subroutine is available only if the GL version is 1.1 or greater.

1, 2, 3, or 4 are not accepted values for **internalFormat**.

An image with height or width of 0 indicates a NULL texture.

Errors

GL_INVALID_ENUM is generated if target is not **GL_TEXTURE_2D**.

GL_INVALID_VALUE is generated if level is less than zero.

GL_INVALID_VALUE may be generated if level is greater than log2max, where max is the returned value of **GL_MAX_TEXTURE_SIZE**.

GL_INVALID_VALUE is generated if width or height is less than zero, greater than 2 + **GL_MAX_TEXTURE_SIZE**, or if width or height cannot be represented as 2**k + 2 * border for some integer k.

GL_INVALID_VALUE is generated if border is not 0 or 1.

GL_INVALID_VALUE is generated if internalFormat is not one of the allowable values.

GL_INVALID_OPERATION is generated if **glCopyTexImage2D** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glGetTexImage.

glIsEnabled with argument **GL_TEXTURE_2D**.

Related Information

The **glCopyPixels** subroutine, **glCopyTexImage1D** subroutine, **glPixelStore** subroutine, **glPixelTransfer** subroutine, **glTexEnv** subroutine, **glTexImage2D** subroutine, **glTexParameter** subroutine.

glCopyTexSubImage1D Subroutine

Purpose

Copies a one-dimensional (1D) texture subimage.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glCopyTexSubImage1D(GLenum  target,
     GLint  level,
     GLint  xoffset,
     GLint  xCoordinate,
     GLint  yCoordinate,
     GLsizei  width)
```

Description

The **glCopyTexSubImage1D** subroutine replaces a portion of a one dimensional texture image with pixels from the current **GL_READ_BUFFER** (rather than from main memory, as is the case for **glTexSubImage1D**).

The screen aligned pixel row with left corner at (x, y), and with length width replaces the portion of the texture array with x indices *xoffset* through *xoffset* + width - 1, inclusive. The destination in the texture array may not include any texels outside the texture array as it was originally specified.

The pixels in the row are processed exactly as if **glCopyPixels** had been called, but the process stops just before final conversion. At this point all pixel component values are clamped to the range [0, 1] and then converted to the texture's internal format for storage in the texel array.

It is not an error to specify a subtexture with zero width, but such a specification has no effect. If any of the pixels within the specified row of the current **GL_READ_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*, or *border* parameters of the specified texture array or to texel values outside the specified subregion.

Parameters

target	Specifies the target texture. Must be **GL_TEXTURE_1D**.
level	Specifies the level-of-detail number. Level 0 is the base image level. Level n is the nth mipmap reduction image.
xoffset	Specifies the texel offset within the texture array.
xCoordinate	Specifies the *x* window coordinate of the lower left corner of the row of pixels to be copied.

yCoordinate	Specifies the *y* window coordinate of the lower left corner of the row of pixels to be copied.
width	Specifies the width of the texture image subimage.

Notes

The **glCopyTexSubImage1D** subroutine is available only if the GL version is 1.1 or greater.

Texturing has no effect in color index mode.

The **glPixelTransfer** mode affects texture images in exactly the way they affect **glDrawPixels**.

Errors

GL_INVALID_ENUM is generated if target is not **GL_TEXTURE_1D**.

GL_INVALID_OPERATION is generated if the texture array has not been defined by a previous **glTexImage1D** operation.

GL_INVALID_VALUE is generated if *width* is less than zero.

GL_INVALID_VALUE may be generated if *level*>log2 max, where max is the returned value of **GL_MAX_TEXTURE_SIZE**.

GL_INVALID_VALUE is generated if *y* < -b or if *width* < -b, where b is the border width of the texture array.

GL_INVALID_VALUE is generated if *xoffset* < -b, or (*xoffset* + *width*) > (w-b). Where w is the **GL_TEXTURE_WIDTH**, and b is the **GL_TEXTURE_BORDER** of the texture image being modified. Note that w includes twice the border width.

Associated Gets

glGetTexImage

glIsEnabled with argument **GL_TEXTURE_1D**.

Related Information

The **glCopyTexSubImage2D** subroutine, **glDrawPixels** subroutine, **glFog** subroutine, **glPixelStore** subroutine, **glPixelTransfer** subroutine, **glTexEnv** subroutine, **glTexGen** subroutine, **glTexImage1D** subroutine, **glTexSubImage1D** subroutine, **glTexParameter** subroutine,

---

glCopyTexSubImage2D Subroutine

Purpose

Copies a two-dimensional (2D) texture subimage.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glCopyTexSubImage2D(GLenum  target,
     GLint  level,
     GLint  xoffset,
     GLint  yoffset,
```

```
GLint xCoordinate,
GLint yCoordinate,
GLsizei width,
GLsizei height)
```

Description

The **glCopyTexSubImage2D** subroutine replaces a portion of a two dimensional texture image with pixels from the current **GL_READ_BUFFER** (rather than from main memory, as is the case for **glTexSubImage2D**).

The screen aligned pixel rectangle with lower left corner at (x, y) and with width width and height height replaces the portion of the texture array with x indices *xoffset* through *xoffset* + width - 1, inclusive, and y indices *yoffset* through *yoffset* + height - 1, inclusive, at the mipmap level specified by level.

The pixels in the rectangle are processed exactly as if **glCopyPixels** had been called, but the process stops just before final conversion. At this point all pixel component values are clamped to the range [0, 1] and then converted to the texture's internal format for storage in the texel array.

The destination rectangle in the texture array may not include any texels outside the texture array as it was originally specified. It is not an error to specify a subtexture with zero width or height, but such a specification has no effect.

If any of the pixels within the specified rectangle of the current **GL_READ_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 parameters of the specified texture array or to texel values outside the specified subregion.

Parameters

target	Specifies the target texture. Must be **GL_TEXTURE_2D**.
level	Specifies the level-of-detail number. Level 0 is the base image level. Level n is the nth mipmap reduction image.
xoffset	Specifies a texel offset in the x direction within the texture array.
yoffset	Specifies a texel offset in the y direction within the texture array.
xCoordinate	Specifies the *x* window coordinate of the lower left corner of the row of pixels to be copied.
yCoordinate	Specifies the *y* window coordinate of the lower left corner of the row of pixels to be copied.
width	Specifies the width of the texture image subimage.
height	Specifies the height of the texture subimage.

Notes

The **glCopyTexSubImage2D** subroutine is available only if the GL version is 1.1 or greater.

Texturing has no effect in color index mode.

The **glPixelTransfer** mode affects texture images in exactly the way they affect **glDrawPixels**.

Errors

GL_INVALID_ENUM is generated if target is not **GL_TEXTURE_2D**.

GL_INVALID_OPERATION is generated if the texture array has not been defined by a previous **glTexImage2D** operation.

GL_INVALID_VALUE is generated if *level* is less than zero.

GL_INVALID_VALUE may be generated if *level* is greater than log2max, where max is the returned value of **GL_MAX_TEXTURE_SIZE**.

GL_INVALID_VALUE is generated if *x* < -b or if *y* < -b, where b is the border width of the texture array.

GL_INVALID_VALUE is generated if *xoffset* < -b, (*xoffset* + *width*) > (w - b), *yoffset* < -b, or (*yoffset* + *height*) > (h - b). Where w is the **GL_TEXTURE_WIDTH**, h is the **GL_TEXTURE_HEIGHT**, and b is the **GL_TEXTURE_BORDER** of the texture image being modified. Note that w and h include twice the border width.

GL_INVALID_OPERATION is generated if **glCopyTexSubImage2D** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glGetTexImage

glIsEnabled with argument **GL_TEXTURE_2D**

Related Information

The **glCopyTexImage2D** subroutine, **glDrawPixels** subroutine, **glFog** subroutine, **glPixelStore** subroutine, **glPixelTransfer** subroutine, **glTexEnv** subroutine, **glTexGen** subroutine, **glTexImage2D** subroutine, **glTexParameter** subroutine.

---

glCopyTexSubImage3D Subroutine

Purpose

Copies a three-dimensional (3D) texture subimage. This subroutine is only supported on OpenGL 1.2 and later.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glCopyTexSubImage3D (GLenum  target,
                          GLint  level,
                          GLint  xoffset,
                          GLint  yoffset,
                          GLint  zoffset,
                          GLint  x,
                          GLint  y,
                          GLsizei  width,
                          GLsizei  height)
```

Description

The **glCopyTexSubImage3D** subroutine replaces a rectangular portion of a three-dimensional texture image with pixels from the current **GL_READ_BUFFER** (rather than from main memory, as is the case for **glTexSubImage3D**).

The screen-aligned pixel rectangle with lower-left corner at (*x*, *y*) and with width *width* and height *height* replaces the portion of the texture array with x indices *xoffset* through *xoffset* + *width* - 1, inclusive, and *y* indices *yoffset* through *yoffset* + *height* - 1, inclusive, at the mipmap level specified by level.

The pixels in the rectangle are processed exactly as if **glCopyPixels** had been called, but the process stops just before final conversion. At this point all pixel component values are clamped to the range [0, 1] and then converted to the texture's internal format for storage in the texel array.

The destination rectangle in the texture array may not include any texels outside the texture array as it was originally specified. It is not an error to specify a subtexture with zero width or height, but such a specification has no effect.

If any of the pixels within the specified rectangle of the current **GL_READ_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*, *heigh*t, *depth*, or *border* parameters of the specified texture array or to texel values outside the specified subregion.

Parameters

target	Specifies the target texture. Must be **GL_TEXTURE_3D**.
level	Specifies the level-of-detail number. Level 0 is the base image level. Level n is the nth mipmap reduction image.
xoffset	Specifies a texel offset in the x direction within the texture array.
yoffset	Specifies a texel offset in the y direction within the texture array.
zoffset	Specifies a texel offset in the z direction within the texture array.
x, *y*	Specify the window coordinates of the lower left corner of the rectangular region of pixels to be copied.
width	Specifies the width of the texture subimage.
*heigh*t	Specifies the height of the texture subimage.

Notes

Texturing has no effect in color index mode.

The **glPixelTransfer** mode affects texture images in exactly the way they affect **glDrawPixels**.

Errors

GL_INVALID_ENUM is generated if *target* is not **GL_TEXTURE_3D**.

GL_INVALID_OPERATION is generated if texture array has not been defined by a previous **glTexImage3D** operation.

GL_INVALID_VALUE is generated if *level* is less than zero.

GL_INVALID_VALUE may be generated if *level* is greater than log2max, where max is the returned value of **GL_MAX_3D_TEXTURE_SIZE**.

GL_INVALID_VALUE is generated if *x* < -b or if *y* < -b, where b is the border width of the texture array.

GL_INVALID_VALUE is generated if *xoffset* < -b, (*xoffset* + *width*) > (w - b), *yoffset* < -b, (*yoffset* + *height*) > (h - b), zoffset < -b, or (*zoffset* + *depth*) > (d - b). Where w is the **GL_TEXTURE_WIDTH**, h is the **GL_TEXTURE_HEIGHT**, d is the **GL_TEXTURE_DEPTH**, and b is the **GL_TEXTURE_BORDER** of the texture image being modified. Note that w, h, and d include twice the border width.

GL_INVALID_OPERATION is generated if **glCopyTexSubImage3D** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glGetTexImage

glIsEnabled with argument **GL_TEXTURE_3D**.

Related Information

The **glDrawPixels** subroutine, **glFog** subroutine, **glPixelStore** subroutine, **glPixelTransfer** subroutine, **glTexEnv** subroutine, **glTexGen** subroutine, **glTexImage3D** subroutine, **glTexParameter** subroutine.

glCopyTexSubImage3DEXT Subroutine

Purpose

Copies a three-dimensional (3D) texture subimage.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glCopyTexSubImage3DEXT(GLenum  target,
     GLint  level,
     GLint  xoffset,
     GLint  yoffset,
     GLint  zoffset,
     GLint  x,
     GLint  y,
     GLsizei  width,
     GLsizei  height)
```

Description

The **glCopyTexSubImage3DEXT** subroutine replaces a rectangular portion of a three-dimensional texture image with pixels from the current **GL_READ_BUFFER** (rather than from main memory, as is the case for **glTexSubImage3DEXT**).

The screen-aligned pixel rectangle with lower-left corner at (*x*, *y*) and with width *width* and height *height* replaces the portion of the texture array with x indices *xoffset* through *xoffset* + *width* - 1, inclusive, and *y* indices *yoffset* through *yoffset* + *height* - 1, inclusive, at the mipmap level specified by level.

The pixels in the rectangle are processed exactly as if **glCopyPixels** had been called, but the process stops just before final conversion. At this point all pixel component values are clamped to the range [0, 1] and then converted to the texture's internal format for storage in the texel array.

The destination rectangle in the texture array may not include any texels outside the texture array as it was originally specified. It is not an error to specify a subtexture with zero width or height, but such a specification has no effect.

If any of the pixels within the specified rectangle of the current **GL_READ_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*, *heigh*t, *depth*, or *border* parameters of the specified texture array or to texel values outside the specified subregion.

Parameters

target	Specifies the target texture. Must be **GL_TEXTURE_3D_EXT**.
level	Specifies the level-of-detail number. Level 0 is the base image level. Level n is the nth mipmap reduction image.
xoffset	Specifies a texel offset in the x direction within the texture array.
yoffset	Specifies a texel offset in the y direction within the texture array.
zoffset	Specifies a texel offset in the z direction within the texture array.
x, *y*	Specify the window coordinates of the lower left corner of the rectangular region of pixels to be copied.
width	Specifies the width of the texture subimage.
height	Specifies the height of the texture subimage.

Notes

The **glCopyTexSubImage3DEXT** subroutine is available only if the **EXT_texture_3d** extension is supported.

Texturing has no effect in color index mode.

The **glPixelTransfer** mode affects texture images in exactly the way they affect **glDrawPixels**.

Errors

GL_INVALID_ENUM is generated if *target* is not **GL_TEXTURE_3D_EXT**.

GL_INVALID_OPERATION is generated if texture array has not been defined by a previous **glTexImage3D** operation.

GL_INVALID_VALUE is generated if *level* is less than zero.

GL_INVALID_VALUE may be generated if *level* is greater than log2max, where max is the returned value of **GL_MAX_3D_TEXTURE_SIZE_EXT**.

GL_INVALID_VALUE is generated if *x* < -b or if *y* < -b, where b is the border width of the texture array.

GL_INVALID_VALUE is generated if *xoffset* < -b, (*xoffset* + *width*) > (w - b), *yoffset* < -b, (*yoffset* + *height*) > (h - b), zoffset < -b, or (*zoffset* + *depth*) > (d - b). Where w is the **GL_TEXTURE_WIDTH**, h is the **GL_TEXTURE_HEIGHT**, d is the **GL_TEXTURE_DEPTH_EXT**, and b is the **GL_TEXTURE_BORDER** of the texture image being modified. Note that w, h, and d include twice the border width.

GL_INVALID_OPERATION is generated if **glCopyTexSubImage3DEXT** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glGetTexImage

glIsEnabled with argument **GL_TEXTURE_3D_EXT**.

Related Information

The **glDrawPixels** subroutine, **glFog** subroutine, **glPixelStore** subroutine, **glPixelTransfer** subroutine, **glTexEnv** subroutine, **glTexGen** subroutine, **glTexImage3DEXT** subroutine, **glTexParameter** subroutine.

glCullFace Subroutine

Purpose

Specifies whether frontfacing or backfacing facets may be culled.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glCullFace(GLenum  mode)
```

Parameters

mode	Specifies whether frontfacint or backfacing facets are candidates for culling. Symbolic constants **GL_FRONT**, **GL_BACK**, and **GL_FRONT_AND_BACK** are accepted. The initial value is **GL_BACK**.

Description

The **glCullFace** subroutine specifies whether frontfacing or backfacing facets are culled (as specified by the *mode* parameter) when facet culling is enabled. Facet culling is enabled and disabled using the **glEnable** and **glDisable** subroutines with the argument **GL_CULL_FACE**. Facets include triangles, quadrilaterals, polygons, and rectangles.

The **glFrontFace** subroutine specifies which of the clockwise and counterclockwise facets are frontfacing and backfacing.

Notes

If *mode* is **GL_FRONT_AND_BACK**, no facets are drawn, but other primitives such as points and lines are drawn.

Errors

GL_INVALID_ENUM is generated if *mode* is not an accepted value.

GL_INVALID_OPERATION is generated if **glCullFace** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glIsEnabled with argument **GL_CULL_FACE**.

glGet with argument **GL_CULL_FACE_MODE**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glEnable** or **glDisable** subroutine, **glFrontFace** subroutine.

glDeleteLists Subroutine

Purpose

Deletes a contiguous group of display lists.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glDeleteLists(GLunint  List,
     GLsizei  Range)
```

Description

The **glDeleteLists** subroutine 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 the *Range* parameter is the number of display lists to be deleted. All display lists *d* with *List* <= *d* <= *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 0 (zero), nothing happens.

Parameters

List	Specifies the integer name of the first display list to delete.
Range	Specifies the number of display lists to delete.

Errors

GL_INVALID_VALUE	*Range* is negative.
GL_INVALID_OPERATION	The **glDeleteLists** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glCallList** subroutine, **glCallLists** subroutine, **glGenLists** subroutine, **glIsList** subroutine, **glNewList** subroutine.

glDeleteTextures Subroutine

Purpose

Deletes named textures.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glDeleteTextures(GLsizei  n,
     const GLuint  *textures)
```

Parameters

n	Specifies the number of textures to be deleted
textures	Specifies an array of textures to be deleted.

Description

The **glDeleteTextures** subroutine 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**). If a texture that is currently bound is deleted, the binding reverts to 0 (the default texture).

The **glDeleteTextures** subroutine silently ignores zeros and names that do not correspond to existing textures.

Notes

The **glDeleteTextures** subroutine is available only if the GL version is 1.1 or greater.

The **glDeleteTextures** subroutine is not included in display lists.

Errors

GL_INVALID_VALUE is generated if n is negative.

GL_INVALID_OPERATION is generated if **glDeleteTextures** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glIsTexture

Related Information

The **glAreTexturesResident** subroutine, **glBindTexture** subroutine, **glGenTextures** subroutine, **glGet** subroutine, **glGetTexParameter** subroutine, **glPrioritizeTextures** subroutine, **glTexImage1D** subroutine, **glTexImage2D** subroutine, **glTexParameter** subroutine.

glDeleteTexturesEXT Subroutine

Purpose

Deletes named textures.

Library

OpenGL and OpenGL C bindings library: **libGL.a**

C Syntax

```
void glDeleteTexturesEXT(GLsizei  n,
     const GLuint  *textures)
```

Description

glDeleteTexturesEXT 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 (by **glGenTexturesEXT**, for example). If a texture that is currently bound is deleted, the binding reverts to zero (the default texture).

glDeleteTexturesEXT silently ignores zeros and names that do not correspond to existing textures.

glDeleteTexturesEXT is not included in display lists.

Parameters

n	The number of textures to be deleted.
textures	An array in which each element is the name of a texture to be deleted.

Notes

glDeleteTexturesEXT is part of the **EXT_texture_object** extension, not part of the core GL command set. If **GL_EXT_texture_object** is included in the string returned by **glGetString**, when called with argument **GL_EXTENSIONS**, extension **EXT_texture_object** is supported by the connection.

Errors

GL_INVALID_VALUE is generated if *n* is negative.

GL_INVALID_OPERATION is generated if **glDeleteTexturesEXT** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glIsTextureEXT

File

/usr/include/GL/glext.h	Contains extensions to C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBindTextureEXT** subroutine, **glGenTexturesEXT** subroutine, **glGet** subroutine, **glGetTexParameter** subroutine, **glTexParameter** subroutine, **glTexSubImage1D** subroutine, **glTexSubImage2D** subroutine, **glTexSubImage3DEXT** subroutine.

glDepthFunc Subroutine

Purpose

Specifies the function used for depth buffer comparisons.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glDepthFunc(GLenum  function)
```

Description

The **glDepthFunc** subroutine 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** and **glDisable** of **GL_DEPTH_TEST**.)

The *function* parameter specifies the conditions under which the pixel will be drawn. The comparison functions are as follows:

GL_NEVER	Never passes.
GL_LESS	Passes if the incoming *z* value is less than the stored *z* 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.

The default value of *function* is **GL_LESS**. Initially, depth testing is disabled.

Parameters

function	Specifies the depth comparison function. Symbolic constants **GL_NEVER**, **GL_LESS**, **GL_EQUAL**, **GL_LEQUAL**, **GL_GREATER**, **GL_NOTEQUAL**, **GL_GEQUAL**, and **GL_ALWAYS** are accepted. The default function is **GL_LESS**.

Errors

GL_INVALID_ENUM	*function* is not an accepted value.
GL_INVALID_OPERATION	The **glDepthFunc** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glDepthFunc** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_DEPTH_FUNC**

glIsEnabled with argument **GL_DEPTH_TEST**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glDepthRange** subroutine, **glEnable** or **glDisable** subroutine, **glGet** subroutine, **glPolygonOffset** subroutine, **glPolygonOffsetEXT** subroutine.

glDepthMask Subroutine

Purpose

Enables or disables writing into the depth buffer.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glDepthMask(GLboolean  Flag)
```

Description

The **glDepthMask** subroutine specifies whether the depth buffer is enabled for writing. If the *Flag* parameter is zero (0), depth buffer writing is disabled. Otherwise, it is enabled. Initially, depth buffer writing is enabled.

Parameters

Flag	Specifies whether the depth buffer is enabled for writing. If *Flag* is 0, depth buffer writing is disabled. Otherwise, it is enabled. Initially, depth buffer writing is enabled.

Errors

GL_INVALID_OPERATION	The **glDepthMask** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glDepthMask** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_DEPTH_WRITEMASK**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glColorMask** subroutine, **glDepthFunc** subroutine, **glDepthRange** subroutine, **glIndexMask** subroutine, **glStencilMask** subroutine.

glDepthRange Subroutine

Purpose

Specifies the mapping of *z* values from normalized device coordinates to window coordinates.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glDepthRange(GLclampd  near,
     GLclampd  far)
```

Description

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** subroutine 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 (zero) and the far plane to 1 (one). With this mapping, the depth buffer range is fully utilized.

Parameters

near	Specifies the mapping of the near clipping plane to window coordinates. The default value is 0.
far	Specifies the mapping of the far clipping plane to window coordinates. The default value is 1.

Notes

It is not necessary that *near* be less than *far*. Reverse mappings such as 1,0 are acceptable.

Errors

GL_INVALID_OPERATION	The **glDepthRange** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glDepthRange** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_DEPTH_RANGE**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glDepthFunc** subroutine, **glPolygonOffset** subroutine, **glPolygonOffsetEXT** subroutine, **glViewport** subroutine.

glDrawArrays Subroutine

Purpose

Renders primitives from array data.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glDrawArrays(GLenum  mode,
     GLint  first,
     GLsizei  count)
```

Description

The **glDrawArrays** subroutine lets you specify multiple geometric primitives with very few subroutine calls. Instead of calling a GL procedure to pass each individual vertex, normal, texture coordinate, edge flage, or color, you can prespecify separate arrays of vertexes, normals, and colors and use them to construct a sequence of primitives with a single call to **glDrawArrays**.

When **glDrawArrays** is called, it uses *count* sequential elements from each enabled array to construct a sequence of geometric primitives, beginning with element *first*. The *mode* parameter specifies what kind of primitives are constructed, and how the array elements construct these primitives. If **GL_VERTEX_ARRAY** is not enabled, no geometric primitives are generated.

Vertex attributes that are modified by **glDrawArrays** have an unspecified value after **glDrawArrays** returns. For example, if **GL_COLOR_ARRAY** is enabled, the value of the current color is undefined after **glDrawArrays** executes. Attributes that are not modified remain well defined.

Parameters

mode	Specifies what kind of primitives to render. Symbolic constants **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** are accepted.
first	Specifies the starting index in the enabled arrays.
count	Specifies the number of indices to be rendered.

Notes

The **glDrawArrays** subroutine is available only if the GL version is 1.1 or greater.

The **glDrawArrays** subroutine is included in display lists. If **glDrawArrays** is entered into 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, their values affect display lists when the lists are created, not when the lists are executed.

Errors

GL_INVALID_ENUM is generated if mode is not an accepted value.

GL_INVALID_VALUE is generated if count is negative.

GL_INVALID_OPERATION is generated if **glDrawArrays** is executed between the execution of **glBegin** and the corresponding **glEnd**.

Related Information

The **glArrayElement** subroutine, **glColorPointer** subroutine, **glDrawElements** subroutine, **glEdgeFlagPointer** subroutine, **glGetPointerv** subroutine, **glIndexPointer** subroutine, **glInterleavedArrays** subroutine, **glNormalPointer** subroutine, **glTexCoordPointer** subroutine, **glVertexPointer** subroutine.

glDrawArraysEXT Subroutine

Purpose

Renders primitives from array data.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glDrawArraysEXT(GLenum  mode,
     GLint  first,
     GLsizei  count)
```

Description

glDrawArraysEXT makes it possible to specify multiple geometric primitives with very few subroutine calls. Instead of calling an OpenGL procedure to pass each individual vertex, normal, or color, separate arrays of vertexes, normals, and colors can be prespecified, and used to define a sequence of primitives (all of the same type) with a single call to **glDrawArraysEXT**.

When **glDrawArraysEXT** is called, *count* sequential elements from each enabled array are used to construct a sequence of geometric primitives, beginning with element *first*. *mode* specifies what kind of primitives are constructed, and how the array elements are used to construct these primitives. If **GL_VERTEX_ARRAY_EXT** is not enabled, no geometric primitives are generated.

Vertex attributes that are modified by **glDrawArraysEXT** have an unspecified value after **glDrawArraysEXT** returns. For example, if **GL_COLOR_ARRAY_EXT** is enabled, the value of the current color is undefined after **glDrawArraysEXT** executes. Attributes that aren't modified remain well defined.

Operation of **glDrawArraysEXT** is atomic with respect to error generation. If an error is generated, no other operations take place.

Parameters

mode	Specifies what kind of primitives to render. Symbolic constants **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** are accepted.
first	Specifies the starting index in the enabled arrays.
count	Specifies the number of indices which should be rendered.

Notes

glDrawArraysEXT may be included in display lists. If **glDrawArraysEXT** is entered into 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, their values affect display lists when the lists are created, not when the lists are executed.

Static array data may be read and cached by the implementation at any time. If static array elements are modified and the arrays are not respecified, the results of any subsequent calls to **glDrawArraysEXT** are undefined.

Although it is not an error to respecify an array between the execution of **glBegin** and the corresponding execution of **glEnd**, the result of such respecification is undefined.

glDrawArraysEXT is part of the _extname(EXT_vertex_array) extension, not part of the core GL command set. If _extstring(EXT_vertex_array) is included in the string returned by **glGetString**, when called with argument **GL_EXTENSIONS**, extension _extname(EXT_vertex_array) is supported.

Errors

GL_INVALID_ENUM is generated if *mode* is not an accepted value.

GL_INVALID_VALUE is generated if *count* is negative.

GL_INVALID_OPERATION is generated if **glDrawArraysEXT** is called between the execution of **glBegin** and the corresponding execution of **glEnd**.

File

/usr/include/GL/glext.h	Contains extensions to C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glArrayElementEXT** subroutine, **glColorPointerEXT** subroutine, **glEdgeFlagPointerEXT** subroutine, **glGetPointervEXT** subroutine, **glIndexPointerEXT** subroutine, **glNormalPointerEXT** subroutine, **glTexCoordPointerEXT** subroutine, **glVertexPointerEXT** subroutine.

---

glDrawBuffer Subroutine

Purpose

Specifies which color buffers are to be used for drawing.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glDrawBuffer(GLenum  Mode)
```

Description

When colors are written to the frame buffer, they are written into the color buffers specified by the **glDrawBuffer** subroutine. The specifications are:

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 the 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 auxiliary color buffer *i* is written.

If more than one color buffer is selected for drawing, blending or logical operations are computed and applied independently for each color buffer and may produce different results in each buffer.

Monoscopic contexts include only left buffers, while stereoscopic contexts include both left and right buffers. Likewise, single-buffered contexts include only front buffers, while double-buffered contexts include both front and back buffers. The context is selected at GL initialization.

Parameters

Mode Specifies up to four color buffers to be drawn into. Symbolic constants **GL_NONE**, **GL_FRONT_LEFT**, **GL_FRONT_RIGHT**, **GL_BACK_LEFT**, **GL_BACK_RIGHT**, **GL_FRONT**, **GL_BACK**, **GL_LEFT**, **GL_RIGHT**, **GL_FRONT_AND_BACK**, and **GL_AUX***i*, where *i* is between 0 and **GL_AUX_BUFFERS** - 1, are accepted. (**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.

Notes

It is always the case that **GL_AUX***i* = **GL_AUX**0 + *i*.

Errors

GL_INVALID_ENUM	*Mode* is not an accepted value.
GL_INVALID_OPERATION	None of the buffers indicated by *Mode* exists.
GL_INVALID_OPERATION	The **glDrawBuffer** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glDrawBuffer** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_DRAW_BUFFER**

glGet with argument **GL_AUX_BUFFERS**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glBlendFunc** subroutine, **glColorMask** subroutine, **glIndexMask** subroutine, **glLogicOp** subroutine, **glReadBuffer** subroutine.

glDrawElements Subroutine

Purpose

Renders primitives from array data.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glDrawElements (GLenum  mode,
     GLsizei  count,
     GLenum  type,
     const GLvoid  *indices)
```

Description

The **glDrawElements** subroutine lets you specify multiple geometric primitives with very few subroutine calls. Instead of calling a GL function to pass each individual vertex, normal, texture coordinate, edge flage, or color, you can prespecify separate arrays of vertexes, normals, and so on and use them to construct a sequence of primitives with a single call to **glDrawElements**.

When **glDrawElements** is called, it uses *count* sequential elements from an enabled array, starting at *indices* to construct a sequence of geometric primitives. *mode* specifies what kind of primitives are constructed and how the array elements 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 are not modified maintain their previous values.

Notes

The **glDrawElements** subroutine is available only if the GL version is 1.1 or greater.

The **glDrawElements** subroutine is included in display lists. If **glDrawElements** is entered into 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, their values affect display lists when the lists are created, not when the lists are executed.

Parameters

mode	Specifies what kind of primitives to render. Symbolic constants **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** are accepted.
count	Specifies the number of elements to be rendered.
type	Specifies the type of the values in indices. Must be one of **GL_UNSIGNED_BYTE**, **GL_UNSIGNED_SHORT**, or **GL_UNSIGNED_INT**.
indices	Specifies a pointer to the location where the indices are stored.

Errors

GL_INVALID_ENUM is generated ifmode is not an accepted value.

GL_INVALID_VALUE is generated if count is negative.

GL_INVALID_OPERATION is generated if **glDrawElements** is executed between the execution of **glBegin** and the corresponding **glEnd**.

Related Information

The **glArrayElement** subroutine, **glColorPointer** subroutine, **glDrawArrays** subroutine, **glEdgeFlagPointer** subroutine, **glGetPointerv** subroutine, **glIndexPointer** subroutine, **glInterleavedArrays** subroutine, **glNormalPointer** subroutine, **glTexCoordPointer** subroutine, **glVertexPointer** subroutine.

glDrawPixels Subroutine

Purpose

Writes a block of pixels to the frame buffer.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glDrawPixels(GLsizei  Width,
                  GLsizei  Height,
                  GLenum  Format,
                  GLenum  Type,
                  const GLvoid * Pixels)
```

Description

The **glDrawPixels** subroutine 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.

A number of 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 subroutines: **glPixelStore**, **glPixelTransfer**, **glPixelMap**, and **glPixelZoom**. This article describes the effects on **glDrawPixels** of many, but not all, of the parameters specified by these four subroutines.

Data is read from the *Pixels* parameter 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*. Indices are always treated individually. Color components are treated as groups of one, two, three, or four values, again based on *Format*. Both individual indices 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**.)

Width multiplied by *Height* pixels are read from memory, starting at location *Pixels*. By default these pixels are taken from adjacent memory locations, except that after every *Width* pixels are read, the read pointer is advanced to the next 4-byte boundary. The 4-byte row alignment is specified by **glPixelStore** with argument **GL_UNPACK_ALIGNMENT**, and it can be set 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. Refer to the **glPixelStore** subroutine for details on these options.

The *Width* multiplied by *Height* pixels that are read from memory are each operated on 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 will be drawn, are specific to the format of the pixels, as specified by *Format*. *Format* can assume one of the following 18 symbolic values:

GL_COLOR_INDEX
: Each pixel is a single value, a color index. It is converted to fixed point, 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. Signed and unsigned integer data is converted with all fraction bits set to 0 (zero). Bitmap data converts to either 0.0 or 1.0.

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, 0 bits fill otherwise unspecified bit locations in the result.

If the GL is in red, green, blue, alpha (RGBA) mode, the resulting index is converted 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. If the GL is in color index mode and **GL_MAP_COLOR** is True, the index is replaced with the value that it references in the lookup table **GL_PIXEL_MAP_I_TO_I**. Whether the lookup replacement of the index is done or not, the integer part of the index is then ANDed with 2b -1, where *b* is the number of bits in a color index buffer.

The resulting indices or RGBA colors are then converted 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* mod *Width* and *y*n = *y*r + [*n/Width*], where (*x*r, *y*r) is the current raster position. 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_INDEX
: Each pixel is a single value, a stencil index. It is converted to fixed point, 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. Signed and unsigned integer data is converted with all fraction bits set to 0. Bitmap data converts to either 0.0 or 1.0.

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, 0 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 the 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 ANDed with 2b -1, where *b* is the number of bits in the stencil buffer. The resulting stencil indices are then written to the stencil buffer such that the *n*th index is written to location *x*n = *x*r + *n* mod *Width* and *y*n = *y*r + *[n/Width]*, where (*x*r, *y*r) is the current raster position. Only the pixel ownership test, the scissor test, and the stencil writemask affect these write operations.

GL_DEPTH_COMPONENT

Each pixel is a single depth component. Floating-point data is converted 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 0 maps to 0.0. 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 *x* and *y* window coordinates to the *n*th fragment such that $x_n = x_r + n \bmod Width$ and $y_n = y_r + \lfloor n/Width \rfloor$, where ($x_r$, y_r) is the current raster position. 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_RGBA

Each pixel is a four-component group, red first, followed by green, followed by blue, followed by alpha. Floating-point values are converted 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 are mapped similarly: the largest integer value maps to 1.0, and 0 maps to 0.0. 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. The results are clamped to the range [0,1].

If **GL_MAP_COLOR** is True, each color component is scaled by the size of the lookup table **GL_PIXEL_MAP_*c*_TO_*c***, then replaced by the value that it references in that table. *c* is **R**, **G**, **B**, or **A**, respectively.

The resulting RGBA colors are then converted 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 $x_n = x_r + n \bmod Width$ and $y_n = y_r + \lfloor n/Width \rfloor$, where ($x_r$, y_r) is the current raster position. 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_BGRA

Each pixel is a four-component group, blue first, followed by green, followed by red, followed by alpha. Floating-point values are converted 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 are mapped similarly: the largest integer value maps to 1.0, and 0 maps to 0.0. The resulting floating-point color values are then multiplied by **GL_*c*_SCALE** and added to **GL_*c*_BIAS**, where *c* is **BLUE**, **GREEN**, **RED**, and **ALPHA** for the respective color components. The results are clamped to the range [0,1].

If **GL_MAP_COLOR** is True, each color component is scaled by the size of the lookup table **GL_PIXEL_MAP_*c*_TO_*c***, then replaced by the value that it references in that table. *c* is **B**, **G**, **R**, or **A**, respectively.

The resulting BGRA colors are then converted 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 $x_n = x_r + n \bmod Width$ and $y_n = y_r + [n/Width]$, where (x_r, y_r) is the current raster position. 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_ABGR_EXT

Each pixel is a four-component group: for **GL_RGBA**, the red component is first, followed by green, followed by blue, followed by alpha: for **GL_BGRA**, the blue component is first, followed by green, followed by red, followed by alpha: for **GL_ABGR_EXT** the order is alpha, blue, green, and then red. Floating-point values are converted directly to an internal floatingpoint 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. 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. The results are clamped to the range [0,1].

If **GL_MAP_COLOR** is true, each color component is scaled by the size of lookup table **GL_PIXEL_MAP_c_TO_c**, then replaced by the value that it references in that table. *c* is **R**, **G**, **B**, or **A**, respectively.

The resulting RGBA colors are then converted 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. 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_RED

Each pixel is a single red component. This component is converted to the internal floating-point format in the same way as the red component of an RGBA pixel is, then it is converted 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 sent in as an RGBA pixel.

GL_GREEN Each pixel is a single green component. This component is converted to the internal floating-point format in the same way as the green component of an RGBA pixel is, then it is converted 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 sent in as an RGBA pixel.

GL_BLUE Each pixel is a single blue component. This component is converted to the internal floating-point format in the same way as the blue component of an RGBA pixel is, then it is converted 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 sent in as an RGBA pixel.

GL_ALPHA Each pixel is a single alpha component. This component is converted to the internal floating-point format in the same way as the alpha component of an RGBA pixel is, then it is converted 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 sent in as an RGBA pixel.

GL_RGB Each pixel is a three-component group, red first, followed by green, followed by blue. Each component is converted to the internal floating-point format in the same way as 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 sent in as an RGBA pixel.

GL_BGR Each pixel is a three-component group, blue first, followed by green, followed by red. Each component is converted to the internal floating-point format in the same way as the blue, green, and red components of an BGRA pixel are. The color triple is converted to an BGRA pixel with alpha set to 1.0. After this conversion, the pixel is treated just as if it had been sent in as an BGRA pixel.

GL_LUMINANCE Each pixel is a single luminance component. This component is converted to the internal floating-point format in the same way as the red component of an RGBA pixel is, then it is converted 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 sent in as an RGBA pixel.

GL_LUMINANCE_ALPHA Each pixel is a two-component group, luminance first, followed by alpha. The two components are converted to the internal floating-point format in the same way as the red component of an RGBA pixel is, then they are converted 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 sent in as an RGBA pixel.

GL_422_EXT This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Y. The second component is Cb in the even pixels and Cr in the odd pixels. The Cb for each even pixel is used as the Cb value for that pixel and its neighbor to the right. The Cr in each odd pixel is used as the Cr value for that pixel and its neighbor to the left. (If the width of the image is odd, then the colors will be undefined in the rightmost column.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this conversion, the pixel is treated just as if it had been sent in as an RGB pixel.

GL_422_REV_EXT	This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Cb in the even pixels and Cr in the odd pixels. The second component is Y. The Cb for each even pixel is used as the Cb value for that pixel and its neighbor to the right. The Cr in each odd pixel is used as the Cr value for that pixel and its neighbor to the left. (If the width of the image is odd, then the colors will be undefined in the rightmost column.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this conversion, the pixel is treated just as if it had been sent in as an RGB pixel.
GL_422_AVERAGE_EXT	This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Y. The second component is Cb in the even pixels and Cr in the odd pixels. Each even pixel gets its Cb from itself, and its Cr from its neighbor to the right. Each odd pixel gets its Cb from the average of its left and right neighbor, and its Cr from the average of itself and its neighbor two to the right. (If the width of the image is odd, then the colors will be undefined in the rightmost column. If the neighbors to the right are not present for a given fragment, we use GL_422_EXT to compute that fragment.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this conversion, the pixel is treated just as if it had been sent in as an RGB pixel.
GL_422_REV_AVERAGE_EXT	This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Cb in the even pixels and Cr in the odd pixels. The second component is Y. Each even pixel gets its Cb from itself, and its Cr from its neighbor to the right. Each odd pixel gets its Cb from the average of its left and right neighbor, and its Cr from the average of itself and its neighbor two to the right. (If the width of the image is odd, then the colors will be undefined in the rightmost column. If the neighbors to the right are not present for a given fragment, we use GL_422_REV_EXT to compute that fragment.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this convers ion, the pixel is treated just as if it had been sent in as an RGB pixel.

The following table summarizes the meaning of the valid constants for the *Type* parameter:

Type	**Corresponding Type**
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
GL_UNSIGNED_BYTE_3_3_2	Unsigned 8-bit integer
GL_UNSIGNED_BYTE_2_3_3_REV	Unsigned 8-bit integer
GL_UNSIGNED_SHORT_5_6_5	Unsigned 16-bit integer

GL_UNSIGNED_SHORT_5_6_5_REV	Unsigned 16-bit integer
GL_UNSIGNED_SHORT_4_4_4_4	Unsigned 16-bit integer
GL_UNSIGNED_SHORT_4_4_4_4_REV	Unsigned 16-bit integer
GL_UNSIGNED_SHORT_5_5_5_1	Unsigned 16-bit integer
GL_UNSIGNED_SHORT_1_5_5_5_REV	Unsigned 16-bit integer
GL_UNSIGNED_INT_8_8_8_8	Unsigned 32-bit integer
GL_UNSIGNED_INT_8_8_8_8_REV	Unsigned 32-bit integer
GL_UNSIGNED_INT_10_10_10_2	Unsigned 32-bit integer
GL_UNSIGNED_INT_2_10_10_10_REV	Unsigned 32-bit integer

The rasterization described thus far assumed pixel zoom factors of 1.0. If **glPixelZoom** is used 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*th column and *m*th row of the pixel rectangle, fragments are generated for pixels whose centers are in the rectangle with corners at (*x*r + *zoom*x *n*, *y*r + *zoom*y *m*) and (*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**.

Parameters

Width	Specifies the width of the pixel rectangle that will be written into the frame buffer.
Height	Specifies the height of the pixel rectangle that will be written into the frame buffer.
Format	Specifies the format of the pixel data. Symbolic constants **GL_COLOR_INDEX**, **GL_STENCIL_INDEX**, **GL_DEPTH_COMPONENT**, **GL_RED**, **GL_GREEN**, **GL_BLUE**, **GL_ALPHA**, **GL_RGB**, **GL_RGBA**, **GL_BGR**, **GL_BGRA**, **GL_ABGR_EXT**, **GL_LUMINANCE**, **GL_LUMINANCE_ALPHA**, **GL_422_EXT**, **GL_422_REV_EXT**, **GL_422_AVERAGE_EXT**, and **GL_422_REV_AVERAGE_EXT** are accepted.
Type	Specifies the data type for *Pixels*. Symbolic constants **GL_UNSIGNED_BYTE**, **GL_BYTE**, **GL_BITMAP**, **GL_UNSIGNED_SHORT**, **GL_SHORT**, **GL_UNSIGNED_INT**, **GL_INT**, **GL_FLOAT**, **GL_UNSIGNED_BYTE_3_3_2**, **GL_UNSIGNED_BYTE_2_3_3_REV**, **GL_UNSIGNED_SHORT_5_6_5**, **GL_UNSIGNED_SHORT_5_6_5_REV**, **GL_UNSIGNED_SHORT_4_4_4_4**, **GL_UNSIGNED_SHORT_4_4_4_4_REV**, **GL_UNSIGNED_SHORT_5_5_5_1**, **GL_UNSIGNED_SHORT_1_5_5_5_REV**, **GL_UNSIGNED_INT_8_8_8_8**, **GL_UNSIGNED_INT_8_8_8_8_REV**, **GL_UNSIGNED_INT_10_10_10_2**, and **GL_UNSIGNED_INT_2_10_10_10_REV**, are accepted.
Pixels	Specifies a pointer to the pixel data.

Notes

Format of **GL_ABGR_EXT** is part of the _extname (EXT_abgr) extension, not part of the core GL command set.

Packed pixel types and BGR/BGRA formats are only supported in OpenGL 1.2 or later.

Errors

GL_INVALID_VALUE	Either *Width* or *Height* is negative.
GL_INVALID_ENUM	*Format* or *Type* is not one of the accepted values.
GL_INVALID_OPERATION	*Format* is **GL_RED, GL_GREEN**, **GL_BLUE**, **GL_ALPHA**, **GL_RGB**, **GL_RGBA**, **GL_AGBR_EXT**, **GL_LUMINANCE**, **GL_LUMINANCE_ALPHA**, and the GL is in color index mode.
GL_INVALID_ENUM	*Type* is **GL_BITMAP** and *Format* is not either **GL_COLOR_INDEX** or **GL_STENCIL_INDEX**.
GL_INVALID_OPERATION	*Format* is **GL_STENCIL_INDEX** and there is no stencil buffer.
GL_INVALID_OPERATION	The **glDrawPixels** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glDrawPixels** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_CURRENT_RASTER_POSITION**

glGet with argument **GL_CURRENT_RASTER_POSITION_VALID**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glAlphaFunc** subroutine, **glBegin** or **glEnd** subroutine, **glBlendFunc** subroutine, **glCopyPixels** subroutine, **glDepthFunc** subroutine, **glLogicOp** subroutine, **glPixelMap** subroutine, **glPixelStore** subroutine, **glPixelTransfer** subroutine, **glPixelZoom** subroutine, **glRasterPos** subroutine, **glReadPixels** subroutine, **glScissor** subroutine, **glStencilFunc** subroutine.

glDrawRangeElements Subroutine

Purpose

Renders primitives from array data. This subrotuine is only supported on OpenGL 1.2 and later.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glDrawRangeElements(GLenum  mode,
                         GLuint  start,
                         GLuint  end,
                         GLsizei  count,
                         GLenum  type,
                         const GLvoid  *indices)
```

Description

The **glDrawRangeElements** subroutine lets you specify multiple geometric primitives with very few subroutine calls. Instead of calling a GL function to pass each individual vertex, normal, texture coordinate, edge flage, or color, you can prespecify separate arrays of vertexes, normals, and so on and use them to construct a sequence of primitives with a single call to **glDrawRangeElements**.

When **glDrawRangeElements** is called, it uses count sequential elements from indices to construct a sequence of geometric primitives. **GLuint** *start* and **GLuint** *end* specify the values between which all values in the array indices must lie. **GLenum** *mode* specifies what kind of primitives are constructed and how the array elements construct these primitives. If **GL_VERTEX_ARRAY** is not enabled, no geometric primitives are generated.

The recommended maximum amounts of vertex and index data can be determined by calling **GetIntegerv** with the symbolic constants **MAX_ELEMENTS_VERTICES** and **MAX_ELEMENTS_INDICES**. If *end*-*start*+1 is greater than the value of **MAX_ELEMENTS_VERTICES**,or if *count* is greater than the value of **MAX_ELEMENTS_INDICES**, then the call may operate at reduced performance. There is no requirement that all vertices in the range [*start*,*end*] be referenced. However, the implementation may partially process unused vertices, reducing performance from what could be achieved with an optimal index set.

Vertex attributes that are modified by **glDrawRangeElements** have an unspecified value after **glDrawRangeElements** returns. For example, if **GL_COLOR_ARRAY** is enabled, the value of the current color is undefined after **glDrawRangeElements** executes. Attributes that are not modified remain well defined.

Parameters

mode	Specifies what kind of primitives to render. Symbolic constants **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** are accepted.
start	Specifies the start value in indices. Must be less than the end value in indices.
end	Specifies the end value in indices. Must be greater than the start value in indices.
count	Specifies the number of elements to be rendered.
type	Specifies the type of the values in indices. Must be one of **GL_UNSIGNED_BYTE**, **GL_UNSIGNED_SHORT**, or **GL_UNSIGNED_INT**.
indices	Specifies a pointer to the location where the indices are stored.

Notes

The **glDrawRangeElements** subroutine is available only if the GL version is 1.1 or greater.

The **glDrawRangeElements** subroutine is included in display lists. If **glDrawRangeElements** is entered into 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, their values affect display lists when the lists are created, not when the lists are executed.

Errors

GL_INVALID_ENUM is generated if mode is not an accepted value.

GL_INVALID_VALUE is generated if count is negative or the end value is less than the start value.

GL_INVALID_OPERATION is generated if **glDrawRangeElements** is executed between the execution of **glBegin** and the corresponding **glEnd**.

Related Information

The **glArrayElement** subroutine, **glColorPointer** subroutine, **glDrawArrays** subroutine, **glEdgeFlagPointer** subroutine, **glGetPointerv** subroutine, **glIndexPointer** subroutine, **glNormalPointer** subroutine, **glTexCoordPointer** subroutine, **glVertexPointer** subroutine.

glEdgeFlag Subroutine

Purpose

Marks edges as either boundary or nonboundary.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glEdgeFlag(GLboolean  Flag)

void glEdgeFlagv(const GLboolean  *Flagv)
```

Description

Each vertex of a polygon, separate triangle, or separate quadrilateral specified between **glBegin** and **glEnd** 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. Otherwise, the vertex is marked as the start of a nonboundary edge. **glEdgeFlag** sets the edge flag to True if the *Flag* parameter is nonzero; otherwise, the edge flag is set to False.

The vertices of connected triangles and connected quadrilaterals are always marked as a 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.

Parameters

Flag	Specifies the current edge flag value, either True or False.
Flagv	Specifies a pointer to an array that contains a single Boolean element (either True or False). Replaces the current edge flag value.

Notes

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**.

Associated Gets

Associated gets for the **glEdgeFlag** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_EDGE_FLAG**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** subroutine, **glEdgeFlagPointer** subroutine, **glEdgeFlagPointerEXT** subroutine, **glEnd** subroutine, **glPolygonMode** subroutine.

glEdgeFlagPointer Subroutine

Purpose

Defines an array of edge flags.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glEdgeFlagPointer( GLsizei  stride,
   const GLvoid * pointer)
```

Description

The **glEdgeFlagPointer** subroutine specifies the location and data format of an array of Boolean edge flags to use when rendering. The *stride* parameter gives the byte stride from one edge flag to the next allowing vertices and attributes to be packed into a single array or stored in separate arrays. (Single array storage may be more efficient on some implementations; see **glInterleavedArrays**.)

When an edge flag array is specified, *stride* and *pointer* are saved as client side state.

To enable and disable the edge flag array, call **glEnableClientState** and **glDisableClientState** with the argument **GL_EDGE_FLAG_ARRAY**. If enabled, the edge flag array is used when **glDrawArrays**, **glDrawElements** or **glArrayElement** is called.

Use **glDrawArrays**, **glMultiDrawArraysEXT**, or **glMultiModeDrawArraysIBM** to construct a sequence of primitives from prespecified vertex and vertex attribute arrays. Use **glArrayElement** to specify primitives by indexing vertices and vertex attributes. Use **glDrawElements**, **glMultiDrawElementsEXT**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

If enabled, the Edge Flag array is used when **glDrawArrays**, **glDrawElements**, **glArrayElements**, **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** is called.

Parameters

stride	Specifies the byte offset between consecutive edge flags. If *stride* is zero (the initial value), the edge flags are understood to be tightly packed in the array. The initial value is 0.
pointer	Specifies a pointer to the first edge flag in the array. The initial value is 0 (NULL pointer).

Notes

The **glEdgeFlagPointer** subroutine is available only if the GL version is 1.1 or greater.

The edge flag array is initially disabled and it won't be accessed when **glArrayElement**, **glDrawElements**, or **glDrawArrays** is called.

Execution of **glEdgeFlagPointer** is not allowed between **glBegin** and the corresponding **glEnd**, but an error may or may not be generated. If an error is not generated, the operation is undefined.

The **glEdgeFlagPointer** subroutine is typically implemented on the client side with no protocol.

Since the edge flag array parameters are client side state, they are not saved or restored by **glPushAttrib** and **glPopAttrib**. Use **glPushClientAttrib** and **glPopClientAttrib** instead.

The **glEdgeFlagPointer** subroutine is not included in display lists.

Error Codes

GL_INVALID_ENUM is generated if stride is negative.

Associated Gets

glIsEnabled with argument **GL_EDGE_FLAG_ARRAY**

glGet with argument **GL_EDGE_FLAG_ARRAY_STRIDE**

glGetPointerv with argument **GL_EDGE_FLAG_ARRAY_POINTER**

Related Information

The **glArrayElement** subroutine, **glColorPointer** subroutine, **glDrawArrays** subroutine, **glDrawElements** subroutine, **glEdgeFlagPointerListIBM** subroutine, **glEnable** subroutine, **glGetPointerv** subroutine, **glIndexPointer** subroutine, **glNormalPointer** subroutine, **glPopClientAttrib** subroutine, **glPushClientAttrib** subroutine, **glTexCoordPointer** subroutine, **glVertexPointer** subroutine.

glEdgeFlagPointerEXT Subroutine

Purpose

Defines an array of edge flags.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glEdgeFlagPointerEXT(GLsizei  stride,
     GLsizei  count,
     const GLboolean  *pointer)
```

Description

glEdgeFlagPointerEXT specifies the location and data format of an array of boolean edge flags to use when rendering. *stride* gives the byte stride from one edge flag to the next allowing vertexes and attributes to be packed into a single array or stored in separate arrays. (Single-array storage may be more efficient on some implementations.) *count* indicates the number of array elements (counting from the first) that are static. Static elements may be modified by the application, but once they are modified, the application must explicitly respecify the array before using it for any rendering. When an edge flag array is specified, *stride*, *count* and *pointer* are saved as client-side state, and static array elements may be cached by the implementation.

The edge flag array is enabled and disabled using **glEnable** and **glDisable** with the argument **GL_EDGE_FLAG_ARRAY_EXT**. If enabled, the edge flag array is used when **glDrawArraysEXT** or **glArrayElementEXT** is called.

Use **glDrawArrays**, **glMultiDrawArraysEXT**, or **glMultiModeDrawArraysIBM** to construct a sequence of primitives from prespecified vertex and vertex attribute arrays. Use **glArrayElement** to specify primitives by indexing vertices and vertex attributes. Use **glDrawElements**, **glMultiDrawElementsEXT**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

If enabled, the Edge Flag array is used when **glDrawArrays**, **glDrawElements**, **glArrayElements**, **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** is called.

Parameters

stride	Specifies the byte offset between consecutive edge flags. If *stride* is zero the edge flags are understood to be tightly packed in the array.
count	Specifies the number of edge flags, counting from the first, that are static.
pointer	Specifies a pointer to the first edge flag in the array.

Notes

Non-static array elements are not accessed until **glArrayElementEXT** or **glDrawArraysEXT** is executed.

By default the edge flag array is disabled and it won't be accessed when **glArrayElementEXT** or **glDrawArraysEXT** is called.

Although, it is not an error to call **glEdgeFlagPointerEXT** between the execution of **glBegin** and the corresponding execution of **glEnd**, the results are undefined.

glEdgeFlagPointerEXT will typically be implemented on the client side with no protocol.

Since the edge flag array parameters are client side state, they are not saved or restored by **glPushAttrib** and **glPopAttrib**.

glEdgeFlagPointerEXT commands are not entered into display lists.

glEdgeFlagPointerEXT is part of the _extname(EXT_vertex_array) extension, not part of the core GL command set. If _extstring(EXT_vertex_array) is included in the string returned by **glGetString**, when called with argument **GL_EXTENSIONS**, extension _extname(EXT_vertex_array) is supported.

Errors

GL_INVALID_ENUM is generated if *stride* or *count* is negative.

Associated Gets

glIsEnabled with argument **GL_EDGE_FLAG_ARRAY_EXT** .

glGet with argument **GL_EDGE_FLAG_ARRAY_STRIDE_EXT**.

glGet with argument **GL_EDGE_FLAG_ARRAY_COUNT_EXT**.

glGetPointervEXT with argument **GL_EDGE_FLAG_ARRAY_POINTER_EXT**.

File

/usr/include/GL/glext.h	Contains extensions to C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glArrayElementEXT** subroutine, **glColorPointerEXT** subroutine, **glDrawArraysEXT** subroutine, **glGetPointervEXT** subroutine, **glIndexPointerEXT** subroutine, **glNormalPointerEXT** subroutine, **glTexCoordPointerEXT** subroutine, **glVertexPointerEXT** subroutine.

glEdgeFlagPointerListIBM Subroutine

Purpose

Defines a list of edge flag arrays.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glEdgeFlagPointerListIBM ( GLint  stride,
   const GLboolean ** pointer,
   GLint  ptrstride)
```

Description

The **glEdgeFlagPointerListIBM** subroutine specifies the location and data format of a list of arrays of edge flags to use when rendering. The *stride* parameter gives the byte stride from one edge flag to the next allowing vertices and attributes to be packed into a single array or stored in separate arrays. (Single-array storage may be more efficient on some implementations; see **glInterleavedArrays**). The *ptrstride* parameter specifies the byte stride from one pointer to the next in the *pointer* array.

When an edge flag array is specified, *stride*, *pointer* and *ptrstride* are saved as client side state.

A *stride* value of 0 does not specify a ″tightly packed″ array as it does in **glEdgeFlagPointer**. Instead, it causes the first array element of each array to be used for each vertex. Also, a negative value can be used for stride, which allows the user to move through each array in reverse order.

To enable and disable the edge flag arrays, call **glEnableClientState** and **glDisableClientState** with the argument **GL_EDGE_FLAG_ARRAY**. The edge flag array is initially disabled. When enabled, the edge flag arrays are used when **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, **glDrawArrays**, **glDrawElements** or **glArrayElement** is called. The last three calls in this list will only use the first array (the one pointed at by *pointer*[0]). See the descriptions of these routines for more information on their use.

Use **glDrawArrays**, **glMultiDrawArraysEXT**, or **glMultiModeDrawArraysIBM** to construct a sequence of primitives from prespecified vertex and vertex attribute arrays. Use **glArrayElement** to specify primitives by indexing vertices and vertex attributes. Use **glDrawElements**, **glMultiDrawElementsEXT**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

If enabled, the Edge Flag array is used when **glDrawArrays**, **glDrawElements**, **glArrayElements**, **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** is called.

Parameters

stride	Specifies the byte offset between consecutive edge flags. The initial value is 0.
pointer	Specifies a list of edge flag arrays. The initial value is 0 (NULL pointer).
ptrstride	Specifies the byte stride between successive pointers in the *pointer* array. The initial value is 0.

Notes

The **glEdgeFlagPointerListIBM** subroutine is available only if the GL_IBM_vertex_array_lists extension is supported.

Execution of **glEdgeFlagPointerListIBM** is not allowed between **glBegin** and the corresponding **glEnd**, but an error may or may not be generated. If an error is not generated, the operation is undefined.

The **glEdgeFlagPointerListIBM** subroutine is typically implemented on the client side.

Since the edge flag array parameters are client side state, they are not saved or restored by **glPushAttrib** and **glPopAttrib**. Use **glPushClientAttrib** and **glPopClientAttrib** instead.

When a **glEdgeFlagPointerListIBM** call is encountered while compiling a display list, the information it contains does NOT contribute to the display list, but is used to update the immediate context instead.

The **glEdgeFlagPointer** call and the **glEdgeFlagPointerListIBM** call share the same state variables. A **glEdgeFlagPointer** call will reset the edge flag list state to indicate that there is only one edge flag list, so that any and all lists specified by a previous **glEdgeFlagPointerListIBM** call will be lost, not just the first list that it specified.

Error Codes

None.

Associated Gets

glIsEnabled with argument **GL_EDGE_FLAG_ARRAY**.

glGetPointerv with argument **GL_EDGE_FLAG_ARRAY_LIST_IBM**.

glGet with argument **GL_EDGE_FLAG_ARRAY_LIST_STRIDE_IBM**.

glGet with argument **GL_EDGE_FLAG_ARRAY_STRIDE**.

Related Information

The **glArrayElement** subroutine, **glEdgeFlagPointer** subroutine, **glDrawArrays** subroutine, **glDrawElements** subroutine, **glEdgeFlagPointer** subroutine, **glEnable** subroutine, **glGetPointerv** subroutine, **glIndexPointer** subroutine, **glInterleavedArrays** subroutine, **glMultiDrawArraysEXT** subroutine, **glMultiDrawElementsEXT** subroutine, **glMultiModeDrawArraysIBM** subroutine, **glMultiModeDrawElementsIBM** subroutine, **glNormalPointer** subroutine, **glPopClientAttrib** subroutine, **glPushClientAttrib** subroutine, **glTexCoordPointer** subroutine, **glVertexPointer** subroutine.

---

glEnable or glDisable Subroutine

Purpose

Enables or disables a GL capability.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glEnable(GLenum  capability)
void glDisable(GLenum  capability)
```

Description

glEnable and **glDisable** enable and disable various capabilities. Use **glIsEnable** or **glGet** to determine the current setting of any capability. Both **glEnable** and **glDisable** take a single argument, *capability*, which may 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** is used to generate vertices. (See **glMap2**.)
GL_BLEND	If enabled, blend the incoming red, green, blue, alpha (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_ARRAY_EXT	If enabled, colors are taken from the color array when **glArrayElementEXT** or **glDrawArraysEXT** is called. (See **glColorPointerEXT**, **glArrayElementEXT** and **glDrawArraysEXT**.)
GL_COLOR_LOGIC_OP	If enabled, apply the currently selected logical operation to the incoming color and color buffer values. The initial value is **GL_FALSE**. (See **glLogicOp**.)
GL_COLOR_MATERIAL	If enabled, have one or more material parameters track the current color. (See **glColorMaterial**.)
GL_COLOR_SUM_EXT	If enabled, user may specify the RGB components of the secondary color used in the Color Sum stage, instead of using the default (0,0,0,0) color. This applies only in RGBA mode and when LIGHTING is disabled. (See **glSecondaryColorEXT**.)
GL_CULL_FACE	If enabled, cull polygons based on their winding in window coordinates. (See **glCullFace**.)
GL_CULL_VERTEX_IBM	If enabled, cull polygons based on their vertex normals. When vertex culling is enabled, vertices are classified as front or back facing according to the sign of the dot product between the normal at the vertex and an eye direction vector from the vertex toward the eye position. When (normal dot eye_direction) <= 0 the vertex is classified as back facing. When (normal dot eye_direction) > 0 the vertex is classified as front facing. Vertices are culled when the face orientation determined by the dot product is the same as the face specified by CullFace. When all of the vertices of a polygon are culled, then the polygon may be culled. Unlike **GL_CULL_VERTEX_EXT**, vertex culling using **GL_CULL_VERTEX_IBM** does not necessarily result in polygons being culled even if all of the vertices of the polygon are culled. The eye direction is determined by transforming the column vector (0, 0, 1) by the upper leftmost 3x3 matrix taken from the inverse of the modelview matrix. The eye direction is undefined if the modelview matrix is singular or nearly singular. This operation in effect projects the z axis in eye coordinates back into object space. If the projection matrix or DepthRange settings cause the z axis in window coordinates to be misaligned with the z axis in eye coordinates, this extension should not be used. Vertex culling is performed independently of face culling. Polygons on the silhouettes of objects may have both front and back facing vertices. Since polygons are culled only if all of their vertices are culled and are not necessarily culled by **GL_CULL_VERTEX_IBM** even in that case, face culling may have to be used in addition to vertex culling in order to correctly cull silhouette polygons.
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 indices before they are written to the color buffer.
GL_EDGE_FLAG_ARRAY_EXT	If enabled, edge flags are taken from the edge flags array when **glArrayElementEXT** or **glDrawArraysEXT** is called. (See **glEdgeFlagPointerEXT**, **glArrayElementEXT** and **glDrawArraysEXT**.)

GL_FOG	If enabled, blend a fog color into the post-texturing color. (See **glFog**.)
GL_INDEX_ARRAY_EXT	If enabled, color indexes are taken from the color index array when **glArrayElementEXT** or **glDrawArraysEXT** is called. (See **glIndexPointerEXT**, **glArrayElementEXT** and **glDrawArraysEXT**.)
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. Otherwise, simply associate the current color or index with each vertex. (See **glMaterial**, **glLightModel**, and **glLight**.)
GL_LINE_SMOOTH	If enabled, draw lines with correct filtering. Otherwise, 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 indices. (See **glLogicOp**.)
GL_MAP1_COLOR_4	If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** will generate RGBA values. (See **glMap1**.)
GL_MAP1_INDEX	If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** will generate color indices. (See **glMap1**.)
GL_MAP1_NORMAL	If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** will generate normals. (See **glMap1**.)
GL_MAP1_TEXTURE_COORD_1	If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** will generate *s* texture coordinates. (See **glMap1**.)
GL_MAP1_TEXTURE_COORD_2	If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** will generate *s* and *t* texture coordinates. (See **glMap1**.)
GL_MAP1_TEXTURE_COORD_3	If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** will generate *s*, *t*, and *r* texture coordinates. (See **glMap1**.)
GL_MAP1_TEXTURE_COORD_4	If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** will generate *s*, *t*, *r*, and *q* texture coordinates. (See **glMap1**.)
GL_MAP1_VERTEX_3	If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** will generate will generate *x*, *y*, and *z* vertex coordinates. (See **glMap1**.)
GL_MAP1_VERTEX_4	If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** will generate homogeneous *x*, *y*, *z*, and *w* vertex coordinates. (See **glMap1**.)
GL_MAP2_COLOR_4	If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** will generate RGBA values. (See **glMap2**.)
GL_MAP2_INDEX	If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** will generate color indices. (See **glMap2**.)
GL_MAP2_NORMAL	If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** will generate normals. (See **glMap2**.)
GL_MAP2_TEXTURE_COORD_1	If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** will generate *s* texture coordinates. (See **glMap2**.)
GL_MAP2_TEXTURE_COORD_2	If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** will generate *s* and *t* texture coordinates. (See **glMap2**.)
GL_MAP2_TEXTURE_COORD_3	If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** will generate *s*, *t*, and *r* texture coordinates. (See **glMap2**.)
GL_MAP2_TEXTURE_COORD_4	If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** will generate *s*, *t*, *r*, and *q* texture coordinates. (See **glMap2**.)

GL_MAP2_VERTEX_3	If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** will generate will generate *x*, *y*, and *z* vertex coordinates. (See **glMap2**.)
GL_MAP2_VERTEX_4	If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** will generate homogeneous *x*, *y*, *z*, and *w* vertex coordinates. (See **glMap2**.)
GL_NORMAL_ARRAY_EXT	If enabled, normals are taken from the normal array when **glArrayElementEXT** or **glDrawArraysEXT** is called. (See **glNormalPointerEXT**, **glArrayElementEXT** and **glDrawArraysEXT**.)
GL_NORMALIZE	If enabled, normal vectors specified with **glNormal** are scaled to unit length after transformation. (See **glNormal**.)
GL_OCCLUSION_CULLING_HP	If enabled, the occlusion testing described within extension **HP_occlusion_test** is performed. This extension allows an application to render some geometry and, at the completion of the rendering, to determine if any of the geometry could or did modify the depth buffer (in other words, a depth buffer test succeeded). (See **glGet** with parameter **GL_OCCLUSION_TEST_RESULT_HP**). Occlusion culling operates independently of the current rendering state (in other words, when occlusion culling is enabled, fragments are generated and the depth and/or color buffer may be updated). To prevent updating the depth/color buffers, the application must disable updates to these buffers. As a side effect of calling **glGet** with parameter **GL_OCCLUSION_TEST_RESULT_HP**, the internal result state is cleared, and it is reset for a new bounding box test.
GL_POLYGON_OFFSET_EXT	If enabled, an offset is added to *z* values of a polygon's fragments before the depth comparison is performed. (See **glPolygonOffsetEXT**.)
GL_POLYGON_OFFSET_FILL	If enabled, and if the polygon is rendered in **GL_FILL** mode, an offset is added to z values of a polygon's fragments before the depth comparison is performed. The initial value is **GL_FALSE**. (See **glPolygonOffset**.)
GL_POLYGON_OFFSET_LINE	If enabled, and if the polygon is rendered in **GL_LINE** mode, an offset is added to z values of a polygon's fragments before the depth comparison is performed. The initial value is **GL_FALSE**. (See **glPolygonOffset**.)
GL_POLYGON_OFFSET_POINT	If enabled, an offset is added to z values of a polygon's fragments before the depth comparison is performed, if the polygon is rendered in **GL_POINT** mode. The initial value is **GL_FALSE**. (See **glPolygonOffset**.)
GL_POINT_SMOOTH	If enabled, draw points with proper filtering. Otherwise, draw aliased points. (See **glPointSize**.)
GL_POLYGON_SMOOTH	If enabled, draw polygons with proper filtering. Otherwise, draw aliased polygons. (See **glPolygonMode**.)
GL_POLYGON_STIPPLE	If enabled, use the current polygon stipple pattern when rendering polygons. (See **glPolygonStipple**.)
GL_RESCALE_NORMAL	If normal rescaling is enabled, a new operation is added to the transformation of the normal vector into eye coordinates. The normal vector is rescaled after it is multiplied by the inverse modelview matrix and before it is normalized.
GL_RESCALE_NORMAL_EXT	If normal rescaling is enabled, a new operation is added to the transformation of the normal vector into eye coordinates. The normal vector is rescaled after it is multiplied by the inverse modelview matrix and before it is normalized.
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_3D	If enabled, three-dimensional texturing is performed. (See **glTexImage3D**.)
GL_TEXTURE_3D_EXT	If enabled, three-dimensional texture mapping is performed. (See **glTexImage3DEXT**.)
GL_TEXTURE_COLOR_TABLE_EXT	If enabled, a color lookup table is added to the texture mechanism. (See **glColorTable**.)
GL_TEXTURE_COORD_ARRAY_EXT	If enabled, texture coordinates are taken from the texture coordinates array when **glArrayElementEXT** or **glDrawArraysEXT** is called. (See **glTexCoordPointerEXT**, **glArrayElementEXT** and **glDrawArraysEXT**.)
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. (See **glTexGen**.)
GL_TEXTURE_GEN_R	If enabled, the *r* texture coordinate is computed using the texture generation function defined with **glTexGen**. Otherwise the current *r* texture coordinate is used. (See **glTexGen**.)
GL_TEXTURE_GEN_S	If enabled, the *s* texture coordinate is computed using the texture generation function defined with **glTexGen**. Otherwise the current *s* texture coordinate is used. (See **glTexGen**.)
GL_TEXTURE_GEN_T	If enabled, the *t* texture coordinate is computed using the texture generation function defined with **glTexGen**. Otherwise, the current *t* texture coordinate is used. (See **glTexGen**.)
GL_UPDATE_CLIP_VOLUME_HINT	If enabled, calls to **ClipBoundingBoxIBM**, **ClipBoundingSphereIBM**, and **ClipBoundingVerticesIBM** will result in updates to the VOLUME_CLIPPING_HINT_EXT state. A result of REJECT_IBM causes the hint to be set to DONT_CARE. A result of CLIP_IBM causes the hint to be set to NICEST. A result of ACCEPT_IBM causes the hint to be set to FASTEST. If the EXT_clip_volume_hint extension is not supported, then the UPDATE_CLIP_VOLUME_HINT enable state has no effect. (See **glClipBoundingBoxIBM**, **glClipBoundingSphereIBM**, or **glClipBoundingVerticesIBM**,)
GL_VERTEX_ARRAY_EXT	If enabled, vertexes are taken from the vertex array when **glArrayElementEXT** or **glDrawArraysEXT** is called. (See **glVertexPointerEXT**, **glArrayElementEXT** and **glDrawArraysEXT**.)

Parameters

capability	Specifies a symbolic constant indicating a GL capability. Initially, all are disabled except **GL_DITHER**.

Errors

GL_INVALID_ENUM	*capability* is not an accepted value.
GL_INVALID_OPERATION	The **glEnable** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glAlphaFunc** subroutine, **glArrayElementEXT** subroutine, **glBegin** or **glEnd** subroutine, **glBlendFunc** subroutine, **glClipPlane** subroutine, **glColorMaterial** subroutine, **glColorPointerEXT** subroutine, **glCullFace** subroutine, **glDepthFunc** subroutine, **glDepthRange** subroutine, **glDrawArraysEXT** subroutine, **glEdgeFlagPointerEXT** subroutine, **glFog** subroutine, **glIndexPointerEXT** subroutine, **glIsEnabled** subroutine, **glLight** subroutine, **glLightModel** subroutine, **glLineStipple** subroutine, **glLineWidth** subroutine, and the **glLogicOp** subroutine.

The **glMap1** subroutine, **glMap2** subroutine, **glMaterial** subroutine, **glNormal** subroutine, **glNormalPointerEXT** subroutine, **glPointSize** subroutine, **glPolygonMode** subroutine, **glPolygonOffset** subroutine, **glPolygonOffsetEXT** subroutine, **glPolygonStipple** subroutine, **glScissor** subroutine, **glTexCoordPointerEXT** subroutine, **glTexGen** subroutine, **glTexImage1D** subroutine, **glTexImage2D** subroutine, and the **glTexImage3DEXT** subroutine.

glEnableClientState or glDisableClientState Subroutine

Purpose

Enables or disables an array.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glEnableClientState(GLenum  array)

void glDisableClientState(GLenum  array)
```

Description

The **glEnableClientState** subroutine lets you enable individual arrays, and **glDisableClientState** lets you disable individual arrays.

Parameters

array	Specifies the array to enable or disable. Symbolic constraints **GL_EDGE_FLAG_ARRAY**, **GL_TEXTURE_COORD_ARRAY**, **GL_COLOR_ARRAY**, **GL_INDEX_ARRAY**, **GL_NORMAL_ARRAY**, **GL_VERTEX_ARRAY**, **GL_FOG_COORDINATE_ARRAY_EXT**, and **GL_SECONDARY_COLOR_ARRAY_EXT** are accepted (for **glEnableClientState**).

Notes

The **glEnableClientState** and **glDisableClientState** subroutines are available only if the GL version is 1.1 or greater.

Errors

GL_INVALID_ENUM is generated if *array* is not an accepted value.

The **glEnableClientState** subroutine is not allowed between the execution of **glBegin** and the corresponding **glEnd**, but an error may or may not be generated. If no error is generated then the behavior is undefined.

Related Information

The **glArrayElement** subroutine, **glColorPointer** subroutine, **glDrawArrays** subroutine, **glDrawElements** subroutine, **glEdgeFlagPointer** subroutine, **glEnable** subroutine, **glGetPointerv** subroutine, **glFogCoordEXT** subroutine, **glIndexPointer** subroutine, **glInterleavedArrays** subroutine, **glNormalPointer** subroutine, **glSecondaryColorEXT** subroutine, **glTexCoordPointer** subroutine, **glVertexPointer** subroutine.

---

glEvalCoord Subroutine

Purpose

Evaluates enabled one-dimensional (1D) and two-dimensional (2D) maps.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

glEvalCoord1d

```
void glEvalCoord1d(GLdouble  u)

void glEvalCoord1f(GLfloat  u)

void glEvalCoord2d(GLdouble  u,
      GLdouble   v)

void glEvalCoord2f(GLfloat  u,
      GLfloat   v)
```

glEvalCoord1dv

```
void glEvalCoord1dv(const GLdouble * u)

void glEvalCoord1fv(const GLfloat * u)

void glEvalCoord2dv(const GLdouble * u)

void glEvalCoord2fv(const GLfloat * u)
```

Description

The **glEvalCoord1** subroutine evaluates enabled 1D maps at argument *u*. The **glEvalCoord2** subroutine does the same for 2D maps using two domain values, *u* and *v*. Maps are defined with **glMap1** and **glMap2**, and enabled and disabled with **glEnable** and **glDisable**.

When one of the **glEvalCoord** subroutines is issued, all currently enabled maps of the indicated dimension are evaluated. Then, for each enabled map, it is as if the corresponding GL subroutine was issued with the computed value. That is, if **GL_MAP1_INDEX** or **GL_MAP2_INDEX** is enabled, a **glIndex** subroutine is simulated. If **GL_MAP1_COLOR_4** or **GL_MAP2_COLOR_4** is enabled, a **glColor** subroutine 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**, or **GL_MAP2_TEXTURE_COORD_4** is enabled, an appropriate **glTexCoord** subroutine is simulated.

The GL 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** subroutines are interspersed with **glEvalCoord** subroutines, the color, normal, and texture coordinates associated with the **glVertex** subroutines will not be affected by the values generated by the **glEvalCoord** subroutines, but rather only by the most recent **glColor**, **glIndex**, **glNormal**, and **glTexCoord** subroutines.

No subroutines 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**), 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-component nor a four-component vertex map is enabled for the specified dimension, the **glEvalCoord** subroutine is ignored.

If automatic normal generation is enabled by calling **glEnable** with argument **GL_AUTO_NORMAL**, **glEvalCoord2** generates surface normals analytically, regardless of the contents or enabling of the **GL_MAP2_NORMAL** map. Let:

```
m = (delta p / delta u) (delta p / delta v)
```

Then the generated normal **n** is

```
n = m/||m||
```

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** subroutines.

Parameters

glEvalCoord1d

u Specifies a value that is the domain coordinate *u* to the basis function defined in a previous **glMap1** or **glMap2** subroutine.

v Specifies a value that is the domain coordinate *v* to the basis function defined in a previous **glMap2** subroutine. This argument is not present in an **glEvalCoord1** subroutine.

glEvalCoord1dv

u Specifies a pointer to an array containing either one or two domain coordinates. The first coordinate is *u*. The second coordinate is *v*, and is present only in **glEvalCoord2** versions.

Associated Gets

Associated gets for the **glEvalCoord** subroutine are as follows. (See the **glGet** subroutine for more information.)

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.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin or glEnd** subroutine, **glColor** subroutine, **glEnable** or **Disable** subroutine, **glEvalMesh** subroutine, **glEvalPoint** subroutine, **glIndex** subroutine, **glMap1** subroutine, **glMap2** subroutine, **glMapGrid** subroutine, **glNormal** subroutine, **glTexCoord** subroutine, **glVertex** subroutine.

---

glEvalMesh Subroutine

Purpose

Computes a one-dimensional (1D) or two-dimensional (2D) grid of points or lines.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glEvalMesh1(GLenum  Mode,
     GLint  i1,
     GLint  i2)

void glEvalMesh2(GLenum  Mode,
     GLint  i1,
     GLint  i2,
     GLint  j1,
     GLint  j2)
```

Description

The **glMapGrid** and **glEvalMesh** subroutines are used in tandem to efficiently generate and evaluate a series of evenly spaced map domain values. The **glEvalMesh** subroutine steps through the integer domain of a 1D or 2D 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 1D case, **glEvalMesh1**, the mesh is generated as if the following code fragment was executed:

```
glBegin(Type);
for (i = i1; i <= i2; i += 1)
        glEvalCoord1(i (DELTA u) + u1)
glEnd();
```

where DELTA $u = (u2 - u1)/n$ and n, $u1$, and $u2$ are the arguments to the most recent **glMapGrid1** subroutine. *Type* is **GL_POINTS** if *Mode* is **GL_POINT**, or **GL_LINES** if *Mode* is **GL_LINE**. The one absolute numeric requirement is that if $i = n$, the value computed from i (DELTA u) + $u1$ is exactly $u2$.

In the 2D case, **glEvalMesh2**, DELTA $u = (u2 - u1)/n$ and DELTA $v = (v2 - v1)/m$, where n, $u1$, $u2$, m, $v1$, and $v2$ are the arguments to the most recent **glMapGrid2** subroutine. Then, if *Mode* is **GL_FILL**, the **glEvalMesh2** subroutine is equivalent to:

```
for (j = j1; j < j2; j += 1) {
        glBegin(GL_QUAD_STRIP)
        for (i= i1; i <= i2; i += 1) {
                glEvalCoord2(i (DELTA u) + u1, j (DELTA v) + v1);
                glEvalCoord2(i (DELTA u) + u1, (j+1) (DELTA v) +
                        v1);
        }
        glEnd();
}
```

If *Mode* is **GL_LINE**, 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 DELTA u + u1, j (DELTA v) + v1);
        glEnd();
}
for (i = i1; i <= i2; i += 1) {
        glBegin(GL_LINE_STRIP);
        for (j = j1; j <= j1; j += 1)
                glEvalCoord2(i (DELTA u + u1, j (DELTA v) + v1);
        glEnd();
}
```

And finally, if *Mode* is **GL_POINT**, 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 (DELTA u) + u1, j (DELTA v) + v1):
        }
}
glEnd();
```

In all three cases, the only absolute numeric requirements are that if $i = n$, the value computed from i (DELTA u) + $u1$ is exactly $u2$, and if $j = m$, the value computed from j (DELTA v) + $v1$ is exactly $v2$.

Parameters

glEvalMesh1

Mode Specifies whether to compute a 1D mesh of points or lines. Symbolic constants **GL_POINT** and **GL_LINE** are accepted.
i1 Specifies the first integer values for grid domain variable *i*.
i2 Specifies the last integer values for grid domain variable *i*.

glEvalMesh2

Mode Specifies whether to compute a 2D mesh of points, lines, or polygons. Symbolic constants **GL_POINT**, **GL_LINE**, and **GL_FILL** are accepted.
i1 Specifies the first integer values for grid domain variable *i*.
i2 Specifies the last integer values for grid domain variable *i*.
j1 Specifies the first integer values for grid domain variable *j*.
j2 Specifies the last integer values for grid domain variable *j*.

Errors

GL_INVALID_ENUM Indicates that *Mode* is not an accepted value.
GL_INVALID_OPERATION Indicates that **glEvalMesh** is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glEvalMesh** subroutine are as follows. (See the **glGet** subroutine for more information.)

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**.

Files

/usr/include/GL/gl.h Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glEvalCoord** subroutine, **glEvalPoint** subroutine, **glMap1** subroutine, **glMap2** subroutine, **glMapGrid** subroutine.

glEvalPoint Subroutine

Purpose

Generates and evaluates a single point in a mesh.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glEvalPoint1(GLint  i)

void glEvalPoint2(GLint  i,
   GLint  j)
```

Description

The **glMapGrid** and **glEvalMesh** subroutines are used in tandem to efficiently generate and evaluate a series of evenly spaced map domain values. **glEvalPoint** can be used to evaluate a single grid point in the same grid space that is traversed by **glEvalMesh**. Calling **glEvalPoint1** is equivalent to calling

```
EvalCoord1(i (DELTA u) + u1);
```

where DELTA *u* = (*u*2 - *u*1)/*n* and *n*, *u*1, and *u*2 are the arguments to the most recent **glMapGrid1** subroutine. The one absolute numeric requirement is that if *i* = *n*, the value computed from *i* (DELTA *u*) + *u*1 is exactly *u*2.

In the two-dimensional case, **glEvalPoint2**, let

```
DELTA u = (u2 - u1 )/n
DELTA v = (v2 - v1 )/m
```

where *n*, *u*1, *u*2, *m*, *v*1, and *v*2 are the arguments to the most recent **glMapGrid2** subroutine. Then the **glEvalPoint2** subroutine is equivalent to calling:

```
EvalCoord2(i (DELTA u) + u1,
j (DELTA v) + v1)
```

The only absolute numeric requirements are that if *i* = *n*, the value computed from *i* (DELTA *u*) + *u*1 is exactly *u*2, and if *j* = *m*, the value computed from *j* (DELTA *v*) + *v*1 is exactly *v*2.

Parameters

i	Specifies the integer value for grid domain variable *i*.
j	Specifies the integer value for grid domain variable *j*. (This parameter applies to **glEvalPoint2** only.)

Associated Gets

Associated gets for the **glEvalPoint** subroutine are as follows. (See the **glGet** subroutine for more information.)

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**.

Files

/usr/include/GL/gl.h Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glEvalCoord** subroutine, **glEvalMesh** subroutine, **glMap1** subroutine, **glMap2** subroutine, **glMapGrid** subroutine.

glFeedbackBuffer Subroutine

Purpose

Controls the feedback mode.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glFeedbackBuffer(GLsizei  Size,
     GLenum  Type,
     GLfloat * Buffer)
```

Description

The **glFeedbackBuffer** subroutine controls feedback. Feedback, like selection, is a GL mode. The mode is selected by calling **glRenderMode** with **GL_FEEDBACK**. When the GL 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 the GL.

The **glFeedbackBuffer** subroutine 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.

The **glFeedbackBuffer** subroutine must be issued before feedback mode is enabled (by calling **glRenderMode** with argument **GL_FEEDBACK**). Setting **GL_FEEDBACK** without establishing the feedback buffer, or calling **glFeedbackBuffer** while the GL is in feedback mode, results in an error.

The GL is taken out of feedback mode by calling **glRenderMode** with a parameter value other than **GL_FEEDBACK**. When this is done while the GL 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 requires more room than is 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 **glPolyMode** 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 GL implementation renders polygons by performing this decomposition.

The **glPassThrough** subroutine can be used to insert a marker into the feedback buffer. (See **glPassThrough**.)

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 red, green, blue, alpha (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**

where *value* is a floating-point number, and *n* is a floating-point integer giving the number of vertices in the polygon. **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** are symbolic floating-point constants. **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. The variable *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.

Parameters

Size	Specifies the maximum number of values that can be written into *Buffer*.
Type	Specifies a symbolic constant that describes the information that is returned for each vertex. **GL_2D**, **GL_3D**, **GL_3D_COLOR**, **GL_3D_COLOR_TEXTURE**, and **GL_4D_COLOR_TEXTURE** are accepted.
Buffer	Returns the feedback data.

Notes

The **glFeedbackBuffer** subroutine, when used in a display list, is not compiled into the display list but rather is executed immediately.

Errors

GL_INVALID_ENUM	*Type* is not an accepted value.
GL_INVALID_VALUE	*Size* is negative.
GL_INVALID_OPERATION	The **glFeedbackBuffer** subroutine is called while the render mode is **GL_FEEDBACK**, or **glRenderMode** is called with argument **GL_FEEDBACK** before **glFeedbackBuffer** is called at least once.
GL_INVALID_OPERATION	The **glFeedbackBuffer** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glFeedbackBuffer** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_RENDER_MODE**.

glGetPointerv with argument **GL_FEEDBACK_BUFFER_POINTER**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glGetPointerv** subroutine, **glLineStipple** subroutine, **glPassThrough** subroutine, **glPolygonMode** subroutine, **glRenderMode** subroutine, **glSelectBuffer** subroutine.

glFinish Subroutine

Purpose

Blocks until all GL execution is complete.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glFinish( void )
```

Description

The **glFinish** subroutine does not return until the effects of all previously called GL subroutines are complete. Such effects include all changes to the GL state, all changes to the connection state, and all changes to the frame buffer contents.

Notes

The **glFinish** subroutine requires a round-trip to the server.

Errors

GL_INVALID_OPERATION	The **glFinish** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glFlush** subroutine, **glWaitGL** subroutine, **glWaitX** subroutine.

glFlush Subroutine

Purpose

Forces the running of GL subroutines in finite time.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glFlush( void )
```

Description

Different GL implementations buffer subroutines in several different locations, including network buffers and the graphics accelerator itself. The **glFlush** subroutine empties all of these buffers, causing all issued subroutines to be executed as quickly as they are accepted by the actual rendering engine. Though this execution cannot be completed in any particular time period, it does complete in finite time.

Because any GL program might be executed over a network, or on an accelerator that buffers subroutines, all programs should call **glFlush** whenever they must have all of their previously issued subroutines completed. For example, call **glFlush** before waiting for user input that depends on the generated image.

Notes

The **glFlush** subroutine can return at any time. It does not wait until the execution of all previously issued OpenGL commands is complete.

Errors

GL_INVALID_OPERATION The **glFlush** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Files

/usr/include/GL/gl.h Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glFinish** subroutine.

glFog Subroutine

Purpose

Specifies fog parameters.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glFogf(GLenum  ParameterName,
     GLfloat  ParameterValue)

void glFogi(GLenum  ParameterName,
     GLint  ParameterValue)

void glFogfv(GLenum  ParameterName,
     const GLfloat * ParameterValues)

void glFogiv(GLenum  ParameterName,
     const GLint * ParameterValues)
```

Description

The **glFog** subroutine is enabled and disabled 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** subroutine assigns the value or values in *ParameterValues* to the fog parameter specified by *ParameterName*. The accepted values for *ParameterName* are:

GL_FOG_MODE *ParameterValues* 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 sections. The default fog mode is **GL_EXP**.

GL_FOG_DENSITY *ParameterValues* 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	*ParameterValues* 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	*ParameterValues* 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	*ParameterValues* is a single integer or floating-point value that specifies f_i, the fog color index. The default fog index is 0.0.
GL_FOG_COLOR	*ParameterValues* 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).
GL_FOG_COORDINATE_SOURCE_EXT	*ParameterValues* is a single integer or floating point value that specifies the source for the fog coordinates. Two symbolic constants are accepted: **GL_FOG_COORDINATE_EXT** and **GL_FRAGMENT_DEPTH_EXT**. Their use is described below. The default fog coordinate source is **GL_FRAGMENT_DEPTH_EXT**.

Fog blends a fog color with each rasterized pixel fragment's post-texturing color using a blending factor *f*. Factor *f* is computed in one of three ways, depending on the fog mode, using one of two values, depending on the fog coordinate source. If the fog coordinate source is **GL_FOG_COORDINATE_EXT** then *z* in the equations below comes from the current fog coordinate. Otherwise, it comes from the fragment's distance from the origin in eye coordinates.

The equation for **GL_LINEAR** fog is:

$$f = \frac{end - z}{end - start}$$

Figure 1. Equation for GL_LINEAR Fog. This figure shows that f is equal to end–z / end–start.

The equation for **GL_EXP** fog is:

$$f = e^{(-density \cdot z)}$$

*Figure 2. Equation for GL_EXP Fog. This figure shows that f is equal to e(-density*z).*

The equation for **GL_EXP2** fog is:

$$f = e^{(-density \cdot z)^2}$$

*Figure 3. Equation for GL_EXP2 Fog. This figure shows that f is equal to e(-density*z) to the power of two.*

Regardless of the fog mode, *f* is clamped to the range [0,1] after it is computed. Then, if the GL is in red, green, blue, alpha (RGBA) color mode, the fragment's color, *C*r, is replaced by the following:

```
Cr prime = fCr + (1 - f) Cf
```

In color index mode, the fragment's color index, *i*r, is replaced by the following:

```
ir prime = ir + (1 - f) if
```

Parameters

glFogf and glFogi

ParameterName	Specifies a single-valued fog parameter. **GL_FOG_DENSITY**, **GL_FOG_END**, **GL_FOG_INDEX**, **GL_FOG_MODE**, **GL_FOG_START**, and **GL_FOG_COORDINATE_SOURCE_EXT** are accepted.
ParameterValue	Specifies the value to which *ParameterName* is set.

glFogfv and glFogiv

ParameterName	Specifies a fog parameter. **GL_FOG_COLOR**, **GL_FOG_DENSITY**, **GL_FOG_END**, **GL_FOG_INDEX**, **GL_FOG_MODE**, **GL_FOG_START**, and **GL_FOG_COORDINATE_SOURCE_EXT** are accepted.
ParameterValues	Specifies the value or values to be assigned to *ParameterName*. **GL_FOG_COLOR** requires an array of four values. All other parameters accept an array containing only a single value.

Errors

GL_INVALID_ENUM	*ParameterName* is not an accepted value.
GL_INVALID_OPERATION	The **glFog** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.
GL_INVALID_VALUE	*ParameterName* is **GL_FOG_DENSITY** and *ParameterValues* is negative.
GL_INVALID_ENUM	*ParameterName* is **GL_FOG_COORDINATE_SOURCE_EXT** and *ParameterValues* is not one of the two permitted values.

Associated Gets

Associated gets for the **glFog** subroutine are as follows. (See the **glGet** subroutine for more information.)

glIsEnabled with argument **GL_FOG**.

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**.

glGet with argument **GL_CURRENT_FOG_COORDINATE_EXT**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glEnable** or **glDisable** subroutine.

---

glFogCoordEXT Subroutine

Purpose

Specifies a Fog Coordinate.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
void glFogCoordfEXT(GLfloat  coord)
void glFogCoorddEXT(GLdouble  coord)
void glFogCoordfvEXT(GLfloat  *Variable)
void glFogCoorddvEXT(GLdouble  *Variable)
```

Description

This extension allows specifying an explicit per-vertex fog coord to be used in fog computations, rather than using a fragment depth-based fog equation.

Parameters

coord	specifies the fog coordinate, which is used in computing the fogging effect, as described in **glFog**. This coordinate is used in place of the distance in eye coordinates from the origin to the fragment being fogged.
Variable	specifies a pointer to a one-element array containing a fog coordinate.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glFog** subroutine, the **glFogCoordPointerEXT** subroutine, the **glFogCoordPointerListIBM** subroutine.

glFogCoordPointerEXT Subroutine

Purpose

Specifies an array of fog coordinates.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
void glFogCoordPointerEXT(GLenum  type,
                          GLsizei  stride,
                          const GLvoid  *pointer)
```

Description

The **glFogCoordPointerEXT** extension specifies the location and data format of an array of fog coordinates to use when rendering. The type parameter specifies the data type of each fog coordinate, and stride gives the byte stride from one coordinate to the next allowing vertices and attributes to be packed into a single array or stored in separate arrays. (Single-array storage may be more efficient on some implementations; see **glInterleavedArrays**).

When a fog coordinate array is specified, type, stride, and pointer are saved as client side state.

To enable and disable the fog coordinate array, call **glEnableClientState** and **glDisableClientState** with the argument **GL_FOG_COORDINATE_ARRAY_EXT**. If enabled, the fog coordinate array is used when **glDrawArrays**, **glDrawElements** or **glArrayElement** is called.

Use **glDrawArrays**, **glMultiDrawArraysEXT**, or **glMultiModeDrawArraysIBM** to construct a sequence of primitives from prespecified vertex and vertex attribute arrays. Use **glArrayElement** to specify primitives by indexing vertices and vertex attributes. Use **glDrawElements**, **glMultiDrawElementsEXT**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

If enabled, the Fog Coord array is used when **glDrawArrays**, **glDrawElements**, **glArrayElements**, **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** is called.

Parameters

type	specifies the data type of each fog coordinate in the array. Symbolic constants **GL_FLOAT**, or **GL_DOUBLE** are accepted. The initial value is **GL_FLOAT**.
stride	specifies the byte offset between consecutive fog coordinates. If *stride* is zero (the initial value), the coordinates are understood to be tightly packed in the array. The initial value is 0.
pointer	specifies a pointer to the first component of the first fog coordinate in the array. The initial value is 0 (NULL pointer).

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glArrayElement** subroutine, the **glColorPointer** subroutine, the **glDrawArrays** subroutine, the **glDrawElements** subroutine, the **glEdgeFlagPointer** subroutine, the **glEnable** subroutine, the **glFogCoordPointerListIBM** subroutine, the **glGetPointerv** subroutine, the **glIndexPointer** subroutine, the **glInterleavedArrays** subroutine, the **glNormalPointer** subroutine, the **glPushClientAttrib** or **glPopClientAttrib** subroutine, the **glTexCoordPointer** subroutine, the **glVertexPointer** subroutine.

glFogCoordPointerListIBM Subroutine

Purpose

Defines a list of arrays of fog coordinates.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
void glFogCoordPointerListIBM ( GLenum  type,
                                GLint  stride,
                                const GLvoid  **pointer,
                                GLint  ptrstride)
```

Description

The **glFogCoordPointerListIBM** subroutine specifies the location and data format of a list of arrays of fog coordinates to use when rendering. The type parameter specifies the data type of each fog coordinate. The *stride* parameter gives the byte stride from one coordinate to the next, allowing vertices and attributes to be packed into a single array or stored in separate arrays. (Single-array storage may be more efficient on some implementations; see **glInterleavedArrays**). The *ptrstride* parameter specifies the byte stride from one pointer to the next in the pointer array.

When a fog coordinate array is specified, *type*, *stride*, *pointer* and *ptrstride* are saved as client side state.

A *stride* value of *0* does not specify a ″tightly packed″ array as it does in **glFogCoordPointer**. Instead, it causes the first array element of each array to be used for each vertex. Also, a negative value can be used for stride, which allows the user to move through each array in reverse order.

To enable and disable the fog coordinate arrays, call **glEnableClientState** and **glDisableClientState** with the argument **GL_COLOR_ARRAY**. The fog coordinate array is initially disabled. When enabled, the fog coordinate arrays are used when **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, **glDrawArrays**, **glDrawElements** or **glArrayElement** is called. The last three calls in this list will only use the first array (the one pointed at by *pointer*[0]). See the descriptions of these routines for more information on their use.

Use **glDrawArrays**, **glMultiDrawArraysEXT**, or **glMultiModeDrawArraysIBM** to construct a sequence of primitives from prespecified vertex and vertex attribute arrays. Use **glArrayElement** to specify primitives by indexing vertices and vertex attributes. Use **glDrawElements**, **glMultiDrawElementsEXT**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

If enabled, the Fog Coord array is used when **glDrawArrays**, **glDrawElements**, **glArrayElements**, **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** is called.

Parameters

type	specifies the data type of each fog coordinate in the arrays. Symbolic constants **GL_FLOAT** or **GL_DOUBLE** are accepted. The initial value is **GL_FLOAT**.
stride	specifies the byte offset between consecutive fog coordinates. The initial value is 0.
pointer	specifies a list of fog coordinate arrays. The initial value is 0 (NULL pointer).
ptrstride	specifies the byte stride between successive pointers in the pointer array. The initial value is 0.

Notes

The **glFogCoordPointerListIBM** subroutine is available only if the **GL_IBM_vertex_array_lists** extension is supported.

Execution of **glFogCoordPointerListIBM** is not allowed between **glBegin** and the corresponding **glEnd**, but an error may or may not be generated. If an error is not generated, the operation is undefined.

The **glFogCoordPointerListIBM** subroutine is typically implemented on the client side.

Since the fog coordinate array parameters are client side state, they are not saved or restored by **glPushAttrib** and **glPopAttrib**. Use **glPushClientAttrib** and **glPopClientAttrib** instead.

When a **glFogCoordPointerListIBM** call is encountered while compiling a display list, the information it contains does NOT contribute to the display list, but is used to update the immediate context instead.

The **glFogCoordPointerEXT** call and the **glFogCoordPointerListIBM** call share the same state variables. A **glFogCoordPointerEXT** call will reset the fog coordinate list state to indicate that there is only one fog coordinate list, so that any and all lists specified by a previous **glFogCoordPointerListIBM** call will be lost, not just the first list that it specified.

Error Codes

GL_INVALID_ENUM	is generated if *type* is not an accepted value.

Associated Gets

Associated gets for the **glFogCoordPointerListIBM** subroutine are as follows. (See the **glGet** subroutine for more information.)

glIsEnabled with argument **GL_FOG_COORDINATE_ARRAY_EXT**.

glGetPointerv with argument **GL_FOG_COORDINATE_ARRAY_POINTER_EXT**.

glGet with arguement **GL_CURRENT_FOG_COORDINATE**.

glGet with arguement **GL_FOG_COORDINATE_ARRAY_TYPE_EXT**.

glGet with arguement **GL_FOG_COORDINATE_ARRAY_STRIDE_EXT**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glArrayElement** subroutine, the **glColorPointer** subroutine, the **glDrawArrays** subroutine, the **glDrawElements** subroutine, the **glEdgeFlagPointer** subroutine, the **glEnable** subroutine, the **glFogCoordPointerEXT** subroutine, the **glGetPointerv** subroutine, the **glIndexPointer** subroutine, the **glInterleavedArrays** subroutine, the **glMultiDrawArraysEXT** subroutine, the **glMultiDrawElementsEXT** subroutine, the **glMultiModeDrawArraysIBM** subroutine, the **glMultiModeDrawElementsIBM** subroutine, the **glNormalPointer** subroutine, the **glPushClientAttrib** or **glPopClientAttrib** subroutine, the **glTexCoordPointer** subroutine, the **glVertexPointer** subroutine.

glFrontFace Subroutine

Purpose

Defines frontfacing and backfacing polygons.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

void glFrontFace(GLenum *Mode***)**

Description

In a scene composed entirely of opaque closed surfaces, backfacing polygons are never visible. Eliminating these invisible polygons speeds up the rendering of the image. Backface elimination is enabled and disabled 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** subroutine specifies whether polygons with clockwise winding in window coordinates, or counterclockwise winding in window coordinates, are taken to be frontfacing. Passing **GL_CCW** to the *Mode* parameter selects counterclockwise polygons as frontfacing; **GL_CW** selects clockwise polygons as frontfacing. By default, counterclockwise polygons are taken to be frontfacing.

Parameters

Mode	Specifies the orientation of frontfacing polygons. **GL_CW** and **GL_CCW** are accepted. The default value is **GL_CCW**.

Errors

GL_INVALID_ENUM	*Mode* is not an accepted value.
GL_INVALID_OPERATION	The **glFrontFace** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glFrontFace** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_FRONT_FACE**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glCullFace** subroutine, **glLightModel** subroutine.

---

glFrustum Subroutine

Purpose

Multiplies the current matrix by a perspective matrix.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glFrustum(GLdouble  Left,
      GLdouble  Right,
      GLdouble  Bottom,
      GLdouble  Top,
      GLdouble  Near,
      GLdouble  Far)
```

Description

The **glFrustum** subroutine describes a perspective matrix that produces a perspective projection.

The parameters (*Left*, *Bottom*, *-Near)* and (*Right*, *Top*, *-Near*) 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). *-Far* specifies the location of the far clipping plane. Both *Near* and *Far* must be positive.

The corresponding matrix is:

$$\begin{pmatrix} \frac{2\ Near}{Right-Left} & 0 & A & 0 \\ 0 & \frac{2\ Near}{Top-Bottom} & B & 0 \\ 0 & 0 & C & D \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Figure 4. Perspective Projection Perspective Matrix. This diagram shows a matrix enclosed in brackets. The matrix consists of four lines containing four characters each. The first line contains the following (from left to right): 2Near / Right-Left, zero, A, zero. The second line contains the following (from left to right): zero, 2Near / Top-Bottom, B, zero. The third line contains the following (from left to right): zero, zero, C, D. The fourth line contains the following (from left to right): zero, zero, -1, zero.

where the following statements apply:

$$A = \frac{Right+Left}{Right-Left}$$

$$B = \frac{Top+Bottom}{Top-Bottom}$$

$$C = \frac{Far+Near}{Far-Near}$$

$$D = \frac{2\ Far\ Near}{Far-Near}$$

Figure 5. Statements. This figure shows the equations used to find the values of A, B, C, and D in the matrix above. In the first equation, A equals Right+Left / Right-Left. In the second equation, B equals Top+Bottom / Top-Bottom. In the third equation, C equals Far+Near / Far-Near. In the fourth equation, D equals 2FarNear / Far-Near.

The current matrix is multiplied 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, *M* is replaced with *MF*.

Use **glPushMatrix** and **glPopMatrix** to save and restore the current matrix stack.

Parameters

Left	Specifies a point on the left side of the clipping plane
Right	Specifies a point on the right side of the clipping plane.
Bottom	Specifies a point on the bottom of the clipping plane.
Top	Specifies a point on the top of the clipping plane.
Near	Specifies the location of the near clipping plane. This must be a positive value.
Far	Specifies the location of the far clipping plane. This must be a positive value.

Notes

Depth buffer precision is affected by the values specified for *Near* and *Far*. The greater the ratio of *Far* to *Near* is, the less effective the depth buffer will be at distinguishing between surfaces that are near each other. If *r* = *Far* / *Near*, roughly log2*r* bits of depth buffer precision are lost. Because *r* approaches infinity as *Near* approaches 0 (zero), *Near* must never be set to 0.

Errors

GL_INVALID_VALUE	Either *Near* or *Far* is not positive.

GL_INVALID_OPERATION	The **glFrustum** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glFrustum** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_MATRIX_MODE**.

glGet with argument **GL_MODELVIEW_MATRIX**.

glGet with argument **GL_PROJECTION_MATRIX**.

glGet with argument **GL_TEXTURE_MATRIX**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glMatrixMode** subroutine, **glMultMatrix** subroutine, **glOrtho** subroutine, **glPushMatrix** or **glPopMatrix** subroutine, **glViewport** subroutine.

glGenLists Subroutine

Purpose

Generates a contiguous set of empty display lists.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLuint glGenLists(GLsizei  Range)
```

Description

The **glGenLists** subroutine 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 0 (zero), if there is no group of *Range* contiguous names available, or if any error is generated, no display lists are generated, and 0 is returned.

Parameters

Range	Specifies the number of contiguous empty display lists to be generated.

Errors

GL_INVALID_VALUE	*Range* is negative.

GL_INVALID_OPERATION	The **glGenLists** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glGenLists** subroutine are as follows. (See the **glGet** subroutine for more information.)

glIsList.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glCallList** subroutine, **glCallLists** subroutine, **glDeleteLists** subroutine, **glNewList** subroutine.

glGenTextures Subroutine

Purpose

Generate texture names.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glGenTextures(GLsizei  n,
     GLuint  *textures)
```

Parameters

n	Specifies the number of texture names to be generated.
textures	Specifies an array in which the generated texture names are stored.

Description

The **glGenTextures** subroutine returns n texture names in textures. There is no guarantee that the names form a contiguous set of integers; however, it is guaranteed that none of the returned names was in use immediately before the call to **glGenTextures**.

The generated textures have no dimensionality; they assume the dimensionality of the texture target to which they are first bound (see **glBindTexture**).

Texture names returned by a call to **glGenTextures** are not returned by subsequent calls, unless they are first deleted with **glDeleteTextures**.

The **glGenTextures** subroutine is not included in display lists.

Notes

The **glGenTextures** subroutine is available only if the GL version is 1.1 or greater.

Errors

GL_INVALID_VALUE is generated if *n* is negative.

GL_INVALID_OPERATION is generated if **glGenTextures** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glIsTexture

Related Information

The **glBindTexture** subroutine, **glDeleteTextures** subroutine, **glGet** subroutine, **glGetTexParameter** subroutine, **glIsTexture** subroutine, **glTexImage1D** subroutine, **glTexImage2D** subroutine, **glTexImage3DEXT** subroutine, **glTexParameter** subroutine.

---

glGenTexturesEXT Subroutine

Purpose

Generates texture names.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glGenTexturesEXT(GLsizei  n,
     GLuint * textures)
```

Description

glGenTexturesEXT returns *n* texture names in *textures*. There is no guarantee that the names form a contiguous set of integers; however, it is guaranteed that none of the returned names was in use immediately before the call to **glGenTexturesEXT**.

The generated textures have no dimensionality; they assume the dimensionality of the texture target to which they are first bound (see **glBindTextureEXT**).

Texture names returned by a call to **glGenTexturesEXT** will not be returned by subsequent calls, unless they are first deleted with **glDeleteTexturesEXT**.

glGenTexturesEXT is not included in display lists.

Parameters

n	The number of texture names to be generated.
textures	An array in which the generated texture names are stored.

Notes

glGenTexturesEXT is part of the **EXT_texture_object** extension, not part of the core GL command set. If **GL_EXT_texture_object** is included in the string returned by **glGetString**, when called with argument **GL_EXTENSIONS**, extension **EXT_texture_object** is supported by the connection.

Errors

GL_INVALID_VALUE is generated if *n* is negative.

GL_INVALID_OPERATION is generated if **glGenTexturesEXT** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glIsTextureEXT.

File

/usr/include/GL/glext.h	Contains extensions to C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBindTextureEXT** subroutine, **glDeleteTexturesEXT** subroutine, **glGet** subroutine, **glGetTexParameter** subroutine, **glTexImage1D** subroutine, **glTexImage2D** subroutine, **glTexImage3DEXT** subroutine, **glTexParameter** subroutine.

---

glGet Subroutine

Purpose

Returns the value or values of a selected parameter.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glGetBooleanv( GLenum  ParameterName,
   GLBoolean * ParameterValues)

void glGetDoublev(GLenum  ParameterName,
   GLdouble * ParameterValues )

void glGetFloatv(GLenum  ParameterName,
   GLfloat * ParameterValues)

void glGetIntegerv(GLenum  ParameterName,
   GLint  ParameterValues)
```

Description

The four commands, **glGetBooleanv**, **glGetDoublev**, **glGetFloatv**, and **glGetIntegerv**, return values for simple-state variables in GL. *ParameterName* is a symbolic constant indicating the state variable to be returned, and *ParameterValues* is a pointer to an array of the indicated type in which to place the returned data.

Type conversion is performed if *ParameterValues* has a different type than the state variable value being requested. If **glGetBooleanv** is called, a floating-point or integer value is converted to **GL_FALSE** if and only if it is 0 (zero). Otherwise, it is converted to **GL_TRUE**. If **glGetIntegerv** is called, 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 either **glGetFloatv** or **glGetDoublev** is called, Boolean values are returned as **GL_TRUE** or **GL_FALSE**, and integer values are converted to floating-point values.

The following symbolic constants are accepted by *ParameterName*:

GL_ACCUM_ALPHA_BITS
ParameterValues returns one value, the number of alpha bit planes in the accumulation buffer.

GL_ACCUM_BLUE_BITS
ParameterValues returns one value, the number of blue bit planes in the accumulation buffer.

GL_ACCUM_CLEAR_VALUE
ParameterValues returns four values: the red, green, blue, and alpha (RGBA) 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
ParameterValues returns one value, the number of green bit planes in the accumulation buffer.

GL_ACCUM_RED_BITS
ParameterValues returns one value, the number of red bit planes in the accumulation buffer.

GL_ALIASED_LINE_WIDTH_RANGE
ParameterValues returns two values: the smallest and largest supported widths for aliased lines. (See **glLineWidth**.)

GL_ALIASED_POINT_SIZE_RANGE
ParameterValues returns two values: the smallest and largest supported sizes for aliased points. (See **glPointSize**.)

GL_ALPHA_BIAS
ParameterValues returns one value, the alpha bias factor used during pixel transfers. (See **glPixelTransfer**.)

GL_ALPHA_BITS
ParmeterValues returns one value, the number of alpha bit planes in each color buffer.

GL_ALPHA_SCALE
ParameterValues returns one value, the alpha scale factor used during pixel transfers. (See **glPixelTransfer**.)

GL_ALPHA_TEST
ParameterValues returns a single Boolean value indicating whether alpha testing of fragments is enabled. (See **glAlphaFunc**.)

GL_ALPHA_TEST_FUNC
ParameterValues returns one value, the symbolic name of the alpha test function. (See **glAlphaFunc**.)

GL_ALPHA_TEST_REF	*ParameterValues* 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_ARRAY_ELEMENT_LOCK_FIRST_EXT	*ParameterValues* returns one value, the first element in the locked range. (See **glLockArraysEXT**.) Requires extension **EXT_compiled_vertex_array**.
GL_ARRAY_ELEMENT_LOCK_COUNT_EXT	*ParameterValues* returns one value, the count of elements in the locked range. (See **glLockArraysEXT**.) Requires extension **EXT_compiled_vertex_array**.
GL_ATTRIB_STACK_DEPTH	*ParameterValues* returns one value, the depth of the attribute stack. If the stack is empty, 0 is returned. (See **glPushAttrib**.)
GL_AUTO_NORMAL	*ParameterValues* returns a single Boolean value indicating whether two-dimensional (2D) map evaluation automatically generates surface normals. (See **glMap2**.)
GL_AUX_BUFFERS	*ParameterValues* returns one value, the number of auxiliary color buffers.
GL_BLEND	*ParameterValues* returns a single Boolean value indicating whether blending is enabled. (See **glBlendFunc**.)
GL_BLEND_DST	*ParameterValues* returns one value, the symbolic constant identifying the destination blend function. (See **glBlendFunc**.)
GL_BLEND_DST_ALPHA_EXT	*ParameterValues* returns one value, the symbolic constant identifying the destination alpha separate blend function. (See **glBlendFuncSeparate**.)
GL_BLEND_DST_RGB_EXT	*ParameterValues* returns one value, the symbolic constant identifying the destination RGB separate blend function. (See **glBlendFuncSeparate**.)
GL_BLEND_EQUATION_EXT	*ParameterValues* returns one value, a symbolic constant indicating the blend equation. (See **glBlendEquationEXT**.) Requires at least one of the following extensions: **EXT_blend_minmax**, **EXT_blend_color**, **EXT_blend_subtract**, **EXT_blend_logic_op**.
GL_BLEND_SRC	*ParameterValues* returns one value, the symbolic constant identifying the source blend function. (See **glBlendFunc**.)
GL_BLEND_SRC_ALPHA_EXT	*ParameterValues* returns one value, the symbolic constant identifying the source alpha separate blend function. (See **glBlendFuncSeparate**.)
GL_BLEND_SRC_RGB_EXT	*ParameterValues* returns one value, the symbolic constant identifying the source RGB separate blend function. (See **glBlendFuncSeparate**.)

GL_BLUE_BIAS	*ParameterValues* returns one value, the blue bias factor used during pixel transfers. (See **glPixelTransfer**.)
GL_BLUE_BITS	*ParameterValues* returns one value, the number of blue bit planes in each color buffer.
GL_CLIENT_ATTRIB_STACK_DEPTH	*ParameterValues* returns one value indicating the depth of the attribute stack. The initial value is 0. (See **glPushClientAttrib**.)
GL_BLUE_SCALE	*ParameterValues* returns one value, the blue scale factor used during pixel transfers. (See **glPixelTransfer**.)
GL_CLIP_PLANE	*ParameterValues* returns a single Boolean value indicating whether the specified clipping plane is enabled. (See **glClipPlane**.)
GL_COLOR_ARRAY	*ParameterValues* returns a single Boolean value indicating whether the color array is enabled. The initial value is **GL_FALSE**. (See **glColorPointer**.)
GL_COLOR_ARRAY_COUNT_EXT	*ParameterName* returns one value, the number of colors in the color array, counting from the first, that are static. (See **glColorPointerEXT**.) Requires extension **EXT_vertex_array**.
GL_COLOR_ARRAY_EXT	*ParameterValues* returns a single boolean value, indicating whether the color array is enabled. (See **glColorPointerEXT**.) Requires extension **EXT_vertex_array**.
GL_COLOR_ARRAY_LIST_STRIDE_IBM	*ParameterValues* returns one value, the byte stride between successive pointers to color lists. The initial value is 0. (See **glColorPointerListIBM**.) Requires extension **IBM_vertex_array_lists**.
GL_COLOR_ARRAY_SIZE	*ParameterValues* returns one value, the number of components per color in the color array. The initial value is 4. (See **glColorPointer**.)
GL_COLOR_ARRAY_SIZE_EXT	*ParameterValues* returns one value, the number of components per color in the color array. (See **glColorPointerEXT**.) Requires extension **EXT_vertex_array**.
GL_COLOR_ARRAY_STRIDE	ParameterValues returns one value, the byte offset between consecutive colors in the color array. The initial value is 0. (See **glColorPointer**.)
GL_COLOR_ARRAY_STRIDE_EXT	*ParameterName* returns one value, the byte offset between consecutive colors in the color array. (See **glColorPointerEXT**.) Requires extension **EXT_vertex_array**.
GL_COLOR_ARRAY_TYPE	ParameterValues returns one value, the data type of each component in the color array. The initial value is **GL_FLOAT**. (See **glColorPointer**.)
GL_COLOR_ARRAY_TYPE_EXT	*ParameterValues* returns one value, the data type of each component in the color array. (See **glColorPointerEXT**.) Requires extension **EXT_vertex_array**.

GL_COLOR_CLEAR_VALUE	*ParameterValues* returns four values: the RGBA 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_LOGIC_OP	*ParameterValues* returns a single Boolean value indicating whether a fragment's color values are merged into the framebuffer using a logical operation. The initial value is **GL_FALSE**. (See **glLogicOp**.)
GL_COLOR_MATERIAL	ParameterValues returns a single Boolean value indicating whether one or more material parameters are tracking the current color. (See **glColorMaterial**.)
GL_COLOR_MATERIAL_FACE	ParameterValues returns one value, a symbolic constant indicating which materials have a parameter that is tracking the current color. (See **glColorMaterial**.)
GL_COLOR_MATERIAL_PARAMETER	*ParameterValues* returns one value, a symbolic constant indicating which material parameters are tracking the current color. (See **glColorMaterial**.)
GL_COLOR_MATRIX	*ParameterValues* returns 16 values: the color matrix. (See **glLoadNamedMatrixIBM**.)
GL_COLOR_SUM_EXT	*ParameterValues* returns a single Boolean value indicating whether the color sum stage and secondary color handling is enabled. (See **glSecondaryColorEXT**.)
GL_COLOR_WRITEMASK	*ParameterValues* returns four Boolean values: the RGBA write enables for the color buffers. (See **glColorMask**.)
GL_CULL_FACE	*ParameterValues* returns a single Boolean value indicating whether polygon culling is enabled. (See **glCullFace**.)
GL_CULL_FACE_MODE	*ParameterValues* returns one value, a symbolic constant indicating which polygon faces are to be culled. (See **glCullFace**.)
GL_CURRENT_COLOR	*ParameterValues* returns four values: the RGBA 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_FOG_COORDINATE_EXT	*ParameterValues* returns one value, the current fog coordinate. (See **glFogCoordEXT**.)
GL_CURRENT_INDEX	*ParameterValues* returns one value, the current color index. (See **glIndex**.)

GL_CURRENT_NORMAL	*ParameterValues* 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	*ParameterValues* returns four values: the RGBA 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	*ParameterValues* returns one value, the distance from the eye to the current raster position. The initial value is 0. (See **glRasterPos**.)
GL_CURRENT_RASTER_INDEX	*ParameterValues* returns one value, the color index of the current raster position. (See **glRasterPos**.)
GL_CURRENT_RASTER_POSITION	*ParameterValues* returns four values: the *x*, *y*, *z*, and *w* components of the current raster position. *x*, *y*, and *z* are in window coordinates, *w* is in clip coordinates. (See **glRasterPos**.)
GL_CURRENT_RASTER_TEXTURE_COORDS	*ParameterValues* returns four values: the *s*, *t*, *r*, and *q* current raster texture coordinates. (See **glRasterPos** and **glTexCoord**.)
GL_CURRENT_RASTER_POSITION_VALID	*ParameterValues* returns a single Boolean value indicating whether the current raster position is valid. (See **glRasterPos**.)
GL_CURRENT_SECONDARY_COLOR	*ParameterValues* returns a four values: the RGBA values of the secondary color. (See **glSecondaryColorEXT**.)
GL_CURRENT_TEXTURE_COORDS	*ParameterValues* returns four values: the *s*, *t*, *r*, and *q* current texture coordinates. (See **glTexCoord**.)
GL_DEPTH_BIAS	*ParameterValues* returns one value, the depth bias factor used during pixel transfers. (See **glPixelTransfer**.)
GL_DEPTH_BITS	*ParameterValues* returns one value, the number of bit planes in the depth buffer.
GL_DEPTH_CLEAR_VALUE	*ParameterValues* 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	*ParameterValues* returns one value, the symbolic constant that indicates the depth comparison function. (See **glDepthFunc**.)

GL_DEPTH_RANGE	*ParameterValues* 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	*ParameterValues* returns one value, the depth scale factor used during pixel transfers. (See **glPixelTransfer**.)
GL_DEPTH_TEST	*ParameterValues* returns a single Boolean value indicating whether depth testing of fragments is enabled. (See **glDepthFunc** and **glDepthRange**.)
GL_DEPTH_WRITEMASK	*ParameterValues* returns a single Boolean value indicating if the depth buffer is enabled for writing. (See **glDepthMask**.)
GL_DITHER	*ParameterValues* returns a single Boolean value indicating whether dithering of fragment colors and indices is enabled.
GL_DOUBLEBUFFER	*ParameterValues* returns a single Boolean value indicating whether double buffering is supported.
GL_DRAW_BUFFER	*ParameterValues* returns one value, a symbolic constant indicating which buffers are being drawn to. (See **glDrawBuffer**.)
GL_EDGE_FLAG	*ParameterValues* returns a single Boolean value indicating whether the current edge flag is True or False. (See **glEdgeFlag**.)
GL_EDGE_FLAG_ARRAY	*ParameterValues* returns a single Boolean value indicating whether the edge flag array is enabled. The initial value is GL_FALSE. (See **glEdgeFlagPointer**.)
GL_EDGE_FLAG_ARRAY_COUNT_EXT	*ParameterValues* returns one value, the number of edge flags in the edge flag array, counting from the first, that are static. (See **glEdgeFlagPointerEXT**.) Requires extension **EXT_vertex_array**.
GL_EDGE_FLAG_ARRAY_EXT	*ParameterValues* returns a single boolean value, indicating whether the edge flag array is enabled. (See **glEdgeFlagPointerEXT**.) Requires extension **EXT_vertex_array**.
GL_EDGE_FLAG_LIST_STRIDE_IBM	*ParameterValues* returns one value, the byte stride between successive pointers to edge flag lists. The initial value is 0. (See **glEdgeFlagPointerListIBM**.) Requires extension **IBM_XXX**.
GL_EDGE_FLAG_ARRAY_STRIDE	*ParameterValues* returns one value, the byte offset between consecutive edge flags in the edge flag array. The initial value is 0. (See **glEdgeFlagPointer**.)
GL_EDGE_FLAG_ARRAY_STRIDE_EXT	*ParameterValues* returns one value, the byte offset between consecutive edge flags in the edge flag array. (See **glEdgeFlagPointerEXT**.) Requires extension **EXT_vertex_array**.

GL_FOG	*ParameterValues* returns a single Boolean value indicating whether fogging is enabled. (See **glFog**.)
GL_FOG_COLOR	*ParameterValues* returns four values: the RGBA 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_COORDINATE_ARRAY_TYPE_EXT	*ParameterValues* returns one value, the fog coordinate array type. (See **glFogCoordPointerEXT**.)
GL_FOG_COORDINATE_ARRAY_STRIDE_EXT	*ParameterValues* returns one value, the fog coordinate array stride. (See **glFogCoordPointerEXT**.)
GL_FOG_DENSITY	*ParameterValues* returns one value, the fog density parameter. (See **glFog**.)
GL_FOG_END	*ParameterValues* returns one value, the end factor for the linear fog equation. (See **glFog**.)
GL_FOG_HINT	*ParameterValues* returns one value, a symbolic constant indicating the mode of the fog hint. (See **glHint**.)
GL_FOG_INDEX	*ParameterValues* returns one value, the fog color index. (See **glFog**.)
GL_FOG_MODE	*ParameterValues* returns one value, a symbolic constant indicating which fog equation is selected. (See **glFog**.)
GL_FOG_START	*ParameterValues* returns one value, the start factor for the linear fog equation. (See **glFog**.)
GL_FRONT_FACE	*ParameterValues* returns one value, a symbolic constant indicating whether clockwise or counterclockwise polygon winding is treated as frontfacing. (See **glFrontFace**.)
GL_GREEN_BIAS	*ParameterValues* returns one value, the green bias factor used during pixel transfers.
GL_GREEN_BITS	*ParameterValues* returns one value, the number of green bit planes in each color buffer.
GL_GREEN_SCALE	*ParameterValues* returns one value, the green scale factor used during pixel transfers. (See **glPixelTransfer**.)
GL_INDEX_ARRAY	*ParameterValues* returns a single Boolean value indicating whether the color index array is enabled. The initial value is GL_FALSE. (See **glIndexPointer**.)
GL_INDEX_ARRAY_COUNT_EXT	*ParameterValues* returns one value, the number of color indexes in the color index array, counting from the first, that are static. (See **glIndexPointerEXT**.) Requires extension **EXT_vertex_array**.
GL_INDEX_ARRAY_EXT	*ParameterValues* returns a single boolean value, indicating whether the color index array is enabled. (See **glIndexPointerEXT**.) Requires extension **EXT_vertex_array**.

GL_INDEX_ARRAY_LIST_STRIDE_IBM	*ParameterValues* returns one value, the byte stride between successive pointers to index lists. The initial value is 0. (See **glIndexPointerListIBM**.) Requires extension **IBM_vertex_array_lists**.
GL_INDEX_ARRAY_STRIDE	*ParameterValues* returns one value, the byte offset between consecutive color indexes in the color index array. The initial value is 0. (See **glIndexPointer**.)
GL_INDEX_ARRAY_STRIDE_EXT	*ParameterValues* returns one value, the byte offset between consecutive color indexes in the color index array. (See **glIndexPointerEXT**.) Requires extension **EXT_vertex_array**.
GL_INDEX_ARRAY_TYPE	*ParameterValues* returns one value, the data type of indexes in the color index array. The initial value is GL_FLOAT. (See **glIndexPointer**.)
GL_INDEX_ARRAY_TYPE_EXT	*ParameterValues* returns one value, the data type of indexes in the color index array. (See **glIndexPointerEXT**.) Requires extension **EXT_vertex_array**.
GL_INDEX_BITS	*ParameterValues* returns one value, the number of bit planes in each color index buffer.
GL_INDEX_CLEAR_VALUE	*ParameterValues* returns one value, the color index used to clear the color index buffers. (See **glClearIndex**.)
GL_INDEX_MODE	*ParameterValues* returns a single Boolean value indicating whether the GL is in color index mode (True) or RGBA mode (False).
GL_INDEX_OFFSET	*ParameterValues* returns one value, the offset added to color and stencil indices during pixel transfers. (See **glPixelTransfer**.)
GL_INDEX_SHIFT	*ParameterValues* returns one value, the amount that color and stencil indices are shifted during pixel transfers. (See **glPixelTransfer**.)
GL_INDEX_WRITEMASK	*ParameterValues* returns one value, a mask indicating which bit planes of each color index buffer can be written. (See **glIndexMask**.)
GL_LIGHT# (where ’#’ is 0...GL_MAXLIGHTS-1)	*ParameterValues* returns a single Boolean value indicating whether the specified light is enabled. (See **glLight** and **glLightModel**.)
GL_LIGHTING	*ParameterValues* returns a single Boolean value indicating whether lighting is enabled. (See **glLightModel**.)
GL_LIGHT_MODEL_AMBIENT	*ParameterValues* returns four values: the RGBA 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_COLOR_CONTROL	*ParameterValues* can be **GL_SINGLE_COLOR** or **GL_SPECULAR_COLOR**. **GL_SINGLE_COLOR** is the default value. Depending upon the *ParameterValues*, the lighting equations compute the two colors differently. All computations are carried out in eye coordinates. (See **glLightModel**.)
GL_LIGHT_MODEL_LOCAL_VIEWER	*ParameterValues* 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	*ParameterValues* returns a single Boolean value indicating whether separate materials are used to compute lighting for frontfacing and backfacing polygons. (See **glLightModel**.)
GL_LINE_SMOOTH	*ParameterValues* returns a single Boolean value indicating whether antialiasing of lines is enabled. (See **glLineWidth**.)
GL_LINE_SMOOTH_HINT	*ParameterValues* returns one value, a symbolic constant indicating the mode of the line antialiasing hint. (See **glHint**).
GL_LINE_STIPPLE	*ParameterValues* returns a single Boolean value indicating whether stippling of lines is enabled. (See **glLineStipple**.)
GL_LINE_STIPPLE_PATTERN	*ParameterValues* returns one value, the 16-bit line stipple pattern. (See **glLineStipple**.)
GL_LINE_STIPPLE_REPEAT	*ParameterValues* returns one value, the line stipple repeat factor. (See **glLineStipple**.)
GL_LINE_WIDTH	*ParameterValues* returns one value, the line width as specified with **glLineWidth**.
GL_LINE_WIDTH_GRANULARITY	*ParameterValues* returns one value, the width difference between adjacent supported widths for antialiased lines. (See **glLineWidth**.)
GL_LINE_WIDTH_RANGE	*ParameterValues* returns two values: the smallest and largest supported widths for antialiased lines. (See **glLineWidth**.)
GL_LIST_BASE	*ParameterValues* returns one value, the base offset added to all names in arrays presented to **glCallLists**. (See **glListBase**.)
GL_LIST_INDEX	*ParameterValues* returns one value, the name of the display list currently under construction. If no display list is currently under construction, 0 is returned. (See **glNewList**.)
GL_LIST_MODE	*ParameterValues* returns one value, a symbolic constant indicating the construction mode of the display list currently being constructed. (See **glNewList**.)
GL_LOGIC_OP	*ParameterValues* 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	*ParameterValues* returns one value, a symbolic constant indicating the selected logic operational mode. (See **glLogicOp**.)
GL_MAP1_COLOR_4	*ParameterValues* returns a single Boolean value indicating whether 1D evaluation generates colors. (See **glMap1**.)

GL_MAP1_GRID_DOMAIN	*ParameterValues* returns two values: the endpoints of the one-dimensional (1D) map's grid domain. (See **glMapGrid**.)
GL_MAP1_GRID_SEGMENTS	*ParameterValues* returns one value, the number of partitions in the 1D map's grid domain. (See **glMapGrid**.)
GL_MAP1_INDEX	*ParameterValues* returns a single Boolean value indicating whether 1D evaluation generates color indices. (See **glMap1**.)
GL_MAP1_NORMAL	*ParameterValues* returns a single Boolean value indicating whether 1D evaluation generates normals. (See **glMap1**.)
GL_MAP1_TEXTURE_COORD_1	*ParameterValues* returns a single Boolean value indicating whether 1D evaluation generates 1D texture coordinates. (See **glMap1**.)
GL_MAP1_TEXTURE_COORD_2	*ParameterValues* returns a single Boolean value indicating whether 1D evaluation generates 2D texture coordinates. (See **glMap1**.)
GL_MAP1_TEXTURE_COORD_3	*ParameterValues* returns a single Boolean value indicating whether 1D evaluation generates 3D texture coordinates. (See **glMap1**.)
GL_MAP1_TEXTURE_COORD_4	*ParameterValues* returns a single Boolean value indicating whether 1D evaluation generates 4D texture coordinates. (See **glMap1**.)
GL_MAP1_VERTEX_3	*ParameterValues* returns a single Boolean value indicating whether 1D evaluation generates 3D vertex coordinates. (See **glMap1**.)
GL_MAP1_VERTEX_4	*ParameterValues* returns a single Boolean value indicating whether 1D evaluation generates 4D vertex coordinates. (See **glMap1**.)
GL_MAP2_COLOR_4	*ParameterValues* returns a single Boolean value indicating whether 2D evaluation generates colors. (See **glMap2**.)
GL_MAP2_GRID_DOMAIN	*ParameterValues* returns four values: the endpoints of the 2D map's *i* and *j* grid domains. (See **glMapGrid**.)
GL_MAP2_GRID_SEGMENTS	*ParameterValues* returns two values: the number of partitions in the 2D map's *i* and *j* grid domains. (See **glMapGrid**.)
GL_MAP2_INDEX	*ParameterValues* returns a single Boolean value indicating whether 2D evaluation generates color indices. (See **glMap2**.)
GL_MAP2_NORMAL	*ParameterValues* returns a single Boolean value indicating whether 2D evaluation generates normals. (See **glMap2**.)
GL_MAP2_TEXTURE_COORD_1	*ParameterValues* returns a single Boolean value indicating whether 2D evaluation generates 1D texture coordinates. (See **glMap2**.)
GL_MAP2_TEXTURE_COORD_2	*ParameterValues* returns a single Boolean value indicating whether 2D evaluation generates 2D texture coordinates. (See **glMap2**.)

GL_MAP2_TEXTURE_COORD_3	*ParameterValues* returns a single Boolean value indicating whether 2D evaluation generates 3D texture coordinates. (See **glMap2**.)
GL_MAP2_TEXTURE_COORD_4	*ParameterValues* returns a single Boolean value indicating whether 2D evaluation generates 4D texture coordinates. (See **glMap2**.)
GL_MAP2_VERTEX_3	*ParameterValues* returns a single Boolean value indicating whether 2D evaluation generates 3D vertex coordinates. (See **glMap2**.)
GL_MAP2_VERTEX_4	*ParameterValues* returns a single Boolean value indicating whether 2D evaluation generates 4D vertex coordinates. (See **glMap2**.)
GL_MAP_COLOR	*ParameterValues* returns a single Boolean value indicating if colors and color indices are to be replaced by table lookup during pixel transfers. (See **glPixelTransfer**.)
GL_MAP_STENCIL	*ParameterValues* returns a single Boolean value indicating if stencil indices are to be replaced by table lookup during pixel transfers. (See **glPixelTransfer**.)
GL_MATRIX_MODE	*ParameterValues* returns one value, a symbolic constant indicating which matrix stack is currently the target of all matrix operations. (See **glMatrixMode**.)
GL_MAX_3D_TEXTURE_SIZE	*ParameterValues* returns one value, the maximum width, height, or depth of any 3D texture image (without borders). (See **glTexImage3DEXT**.)
GL_MAX_3D_TEXTURE_SIZE_EXT	*ParameterValues* returns one value, a rough estimate of the largest 3D texture that the GL can handle. If the GL version is 1.2 or greater, use **GL_PROXY_TEXTURE_3D**″ to determine if a texture is too large. (See **glTexImage3DEXT**.) Requires extension **EXT_texture3D**.
GL_MAX_ATTRIB_STACK_DEPTH	*ParameterValues* returns one value, the maximum supported depth of the attribute stack. (See **glPushAttrib**.)
GL_MAX_CLIENT_ATTRIB_STACK_DEPTH	*ParameterValues* returns one value indicating the maximum supported depth of the client attribute stack. (See **glPushClientAttrib**.)
GL_MAX_CLIP_PLANES	*ParameterValues* returns one value, the maximum number of application-defined clipping planes. (See **glClipPlane**.)
GL_MAX_ELEMENTS_INDICES	*ParameterValues* returns one value: the maximum number of **DrawRangeElements** vertices.
GL_MAX_ELEMENTS_VERTICES	*ParameterValues* returns one value: the maximum number of **DrawRangeElements** vertices.
GL_MAX_EVAL_ORDER	*ParameterValues* returns one value, the maximum equation order supported by 1D and 2D evaluators. (See **glMap1** and **glMap2**.)
GL_MAX_LIGHTS	*ParameterValues* returns one value, the maximum number of lights. (See **glLight**.)

GL_MAX_LIST_NESTING	*ParameterValues* returns one value, the maximum recursion depth allowed during display list traversal. (See **glCallList**.)
GL_MAX_MODELVIEW_STACK_DEPTH	*ParameterValues* returns one value, the maximum supported depth of the modelview matrix stack. (See **glPushMatrix**.)
GL_MAX_NAME_STACK_DEPTH	*ParameterValues* returns one value, the maximum supported depth of the selection name stack. (See **glPushName**.)
GL_MAX_PIXEL_MAP_TABLE	*ParameterValues* returns one value, the maximum supported size of a **glPixelMap** lookup table. (See **glPixelMap**.)
GL_MAX_PROJECTION_STACK_DEPTH	*ParameterValues* returns one value, the maximum supported depth of the projection matrix stack. (See **glPushMatrix**.)
GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT	*ParameterValues* returns one value, the maximum level of texture anisotropy supported by this implementation. (See **glGetTexParameter**.) Requires extension **EXT_texture_filter_anisotropic**.
GL_MAX_TEXTURE_SIZE	*ParameterValues* returns one value, the maximum width or height of any texture image (without borders). (See **glTexImage1D** and **glTexImage2D**.)
GL_MAX_TEXTURE_STACK_DEPTH	*ParameterValues* returns one value, the maximum supported depth of the texture matrix stack. (See **glPushMatrix**.)
GL_MAX_VIEWPORT_DIMS	*ParameterValues* returns two values: the maximum supported width and height of the viewport. (See **glViewport**.)
GL_MAX_VISIBILITY_THRESHOLD_IBM	*ParameterValues* returns one value: the maximum permitted number of visible fragments that will be discarded prior to registering a visibility hit. (See **glVisibilityBufferIBM**.) Requires extension **IBM_occlusion_cull**.
GL_MODELVIEW_MATRIX	*ParameterValues* returns 16 values: the modelview matrix on the top of the modelview matrix stack. (See **glMatrixMode**.)
GL_MODELVIEW_STACK_DEPTH	*ParameterValues* returns one value, the number of matrices on the modelview matrix stack. (See **glPushMatrix**.)
GL_NAME_STACK_DEPTH	*ParameterValues* returns one value, the number of names on the selection name stack. (See **glPushMatrix**.)
GL_NORMAL_ARRAY	*ParameterValues* returns a single Boolean value, indicating whether the normal array is enabled. The initial value is **GL_FALSE**. (See **glNormalPointer**.)
GL_NORMAL_ARRAY_COUNT_EXT	*ParameterValues* returns one value, the number of normals in the normal array, counting from the first, that are static. (See **glNormalPointerEXT**.) Requires extension **EXT_vertex_array**.
GL_NORMAL_ARRAY_EXT	*ParameterValues* returns a single boolean value, indicating whether the normal array is enabled. (See **glNormalPointerEXT**.) Requires extension **EXT_vertex_array**.

GL_NORMAL_ARRAY_LIST_STRIDE_IBM	*ParameterValues* returns one value, the byte stride between successive pointers to normal lists. The initial value is 0. (See **glNormalPointerListIBM**.) Requires extension **IBM_vertex_array_lists**.
GL_NORMAL_ARRAY_STRIDE	*ParameterValues* returns one value, the byte offset between consecutive normals in the normal array. The initial value is 0. (See **glNormalPointer**.)
GL_NORMAL_ARRAY_STRIDE_EXT	*ParameterValues* returns one value, the byte offset between consecutive normals in the normal array. (See **glNormalPointerEXT**.) Requires extension **EXT_vertex_array**.
GL_NORMAL_ARRAY_TYPE	*ParameterValues* returns one value, the data type of each coordinate in the normal array. The initial value is **GL_FLOAT**. (See **glNormalPointer**.)
GL_NORMAL_ARRAY_TYPE_EXT	*ParameterValues* returns one value, the data type of each coordinate in the normal array. (See **glNormalPointerEXT**.) Requires extension **EXT_vertex_array**.
GL_NORMALIZE	*ParameterValues* 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_OCCLUSION_TEST_HP	*ParameterValues* returns a single Boolean value indicating whether the occlusion test **HP_OCCLUSION_TEST** is enabled. (See **glEnable**.)
GL_OCCLUSION_TEST_RESULT_HP	*ParameterValues* returns a single Boolean value indicating whether the occlusion test **HP_OCCLUSION_TEST** noted any fragments successfully passing the depth test. (See **glEnable**.)
GL_PACK_ALIGNMENT	*ParameterValues* returns one value, the byte alignment used for writing pixel data to memory. (See **glPixelStore**.)
GL_PACK_IMAGE_HEIGHT	*ParameterValues* returns one value, the number of image rows used for writing 3D pixel data to memory. (See **glPixelStore**.)
GL_PACK_IMAGE_HEIGHT_EXT	*ParameterValues* returns one value, the number of image rows used for writing 3D pixel data to memory. (See **glPixelStore**.) Requires extension **EXT_texture3D**.
GL_PACK_LSB_FIRST	*ParameterValues* 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	*ParameterValues* returns one value, the row length used for writing pixel data to memory. (See **glPixelStore**.)
GL_PACK_SKIP_IMAGES	*ParameterValues* returns one value, the number of 2D images skipped before the first pixel of a 3D image is written into memory. (See **glPixelStore**.)

GL_PACK_SKIP_IMAGES_EXT	*ParameterValues* returns one value, the number of 2D images skipped before the first pixel of a 3D image is written into memory. (See **glPixelStore**.) Requires extension **EXT_texture3D**.
GL_PACK_SKIP_PIXELS	*ParameterValues* returns one value, the number of pixel locations skipped before the first pixel is written into memory. (See **glPixelStore**.)
GL_PACK_SKIP_ROWS	*ParameterValues* 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	*ParameterValues* returns a single Boolean value indicating whether the bytes of 2-byte and 4-byte pixel indices and components are swapped before being written to memory. (See **glPixelStore**.)
GL_PERSPECTIVE_CORRECTION_HINT	*ParameterValues* returns one value, a symbolic constant indicating the mode of the perspective correction hint. (See **glHint**.)
GL_PIXEL_MAP_A_TO_A_SIZE	*ParameterValues* returns one value, the size of the alpha-to-alpha pixel translation table. (See **glPixelMap**.)
GL_PIXEL_MAP_B_TO_B_SIZE	*ParameterValues* returns one value, the size of the blue-to-blue pixel translation table. (See **glPixelMap**.)
GL_PIXEL_MAP_G_TO_G_SIZE	*ParameterValues* returns one value, the size of the green-to-green pixel translation table. (See **glPixelMap**.)
GL_PIXEL_MAP_I_TO_A_SIZE	*ParameterValues* returns one value, the size of the index-to-alpha pixel translation table. (See **glPixelMap**.)
GL_PIXEL_MAP_I_TO_B_SIZE	*ParameterValues* returns one value, the size of the index-to-blue pixel translation table. (See **glPixelMap**.)
GL_PIXEL_MAP_I_TO_G_SIZE	*ParameterValues* returns one value, the size of the index-to-green pixel translation table. (See **glPixelMap**.)
GL_PIXEL_MAP_I_TO_I_SIZE	*ParameterValues* returns one value, the size of the index-to-index pixel translation table. (See **glPixelMap**.)
GL_PIXEL_MAP_I_TO_R_SIZE	*ParameterValues* returns one value, the size of the index-to-red pixel translation table. (See **glPixelMap**.)
GL_PIXEL_MAP_R_TO_R_SIZE	*ParameterValues* returns one value, the size of the red-to-red pixel translation table. (See **glPixelMap**.)
GL_PIXEL_MAP_S_TO_S_SIZE	*ParameterValues* returns one value, the size of the stencil-to-stencil pixel translation table. (See **glPixelMap**.)
GL_POINT_SIZE	*ParameterValues* returns one value, the point size as specified by **glPointSize**.
GL_POINT_SIZE_GRANULARITY	*ParameterValues* returns one value, the size difference between adjacent supported sizes for antialiased points. (See **glPointSize**.)
GL_POINT_SIZE_RANGE	*ParameterValues* returns two values: the smallest and largest supported sizes for antialiased points. (See **glPointSize**.)

GL_POINT_SMOOTH	*ParameterValues* returns a single Boolean value indicating whether antialiasing of points is enabled. (See **glPointSize**.)
GL_POINT_SMOOTH_HINT	*ParameterValues* returns one value, a symbolic constant indicating the mode of the point antialiasing hint. (See **glHint**.)
GL_POLYGON_MODE	*ParameterValues* returns two values: symbolic constants indicating whether frontfacing and backfacing polygons are rasterized as points, lines, or filled polygons. (See **glPolygonMode**.)
GL_POLYGON_OFFSET_BIAS_EXT	*ParameterValues* returns one value, the constant which is added to the *z* value of each fragment generated when a polygon is rasterized. (See **glPolygonOffsetEXT**.) Requires extension **EXT_polygon_offset**.
GL_POLYGON_OFFSET_EXT	*ParameterValues* returns a single Boolean value indicating whether polygon offest is enabled. (See **glPolygonOffsetEXT**.) Requires extension **EXT_polygon_offset**.
GL_POLYGON_OFFSET_FACTOR	*ParameterValues* returns one value, the scaling factor used to determine the variable offset which is added to the depth value of each fragment generated when a polygon is rasterized. The initial value is 0.0. (See **glPolygonOffset**.)
GL_POLYGON_OFFSET_FACTOR_EXT	*ParameterValues* returns one value, the scaling factor used to determine the variable offset which is added to the *z* value of each fragment generated when a polygon is rasterized. (See **glPolygonOffsetEXT**.) Requires extension **EXT_polygon_offset**.
GL_POLYGON_OFFSET_FILL	*ParameterValues* returns a single Boolean value indicating whether polygon offset is enabled for polygons in fill mode. The initial value is **GL_FALSE**. (See **glPolygonOffset**.)
GL_POLYGON_OFFSET_LINE	*ParameterValues* returns a single Boolean value indicating whether polygon offset is enabled for polygons in line mode. The initial value is **GL_FALSE**. (See **glPolygonOffset**.)
GL_POLYGON_OFFSET_POINT	*ParameterValues* returns a single Boolean value indicating whether polygon offset is enabled for polygons in point mode. The initial value is **GL_FALSE**. (See **glPolygonOffset**.)
GL_POLYGON_OFFSET_UNITS	*ParameterValues* returns one value, this value is multiplied by an implementation-specific value and then added to the z value of each fragment generated when a polygon is rasterized. The initial value is 0.0. (See **glPolygonOffset**.)
GL_POLYGON_SMOOTH	*ParameterValues* returns a single Boolean value indicating whether antialiasing of polygons is enabled. (See **glPolygonMode**.)
GL_POLYGON_SMOOTH_HINT	*ParameterValues* returns one value, a symbolic constant indicating the mode of the polygon antialiasing hint. (See **glHint**.)
GL_POLYGON_STIPPLE	*ParameterValues* returns a single Boolean value indicating whether stippling of polygons is enabled. (See **glPolygonStipple**.)

GL_PROJECTION_MATRIX	*ParameterValues* returns 16 values: the projection matrix on the top of the projection matrix stack. (See **glMatrixMode**.)
GL_PROJECTION_STACK_DEPTH	*ParameterValues* returns one value, the number of matrices on the projection matrix stack. (See **glPushMatrix**.)
GL_READ_BUFFER	*ParameterValues* returns one value, a symbolic constant indicating which color buffer is selected for reading. (See **glReadPixels** and **glAccum**.)
GL_RED_BIAS	*ParameterValues* returns one value, the red bias factor used during pixel transfers.
GL_RED_BITS	*ParameterValues* returns one value, the number of red bit planes in each color buffer.
GL_RED_SCALE	*ParameterValues* returns one value, the red scale factor used during pixel transfers. (See **glPixelTransfer**.)
GL_RENDER_MODE	*ParameterValues* returns one value, a symbolic constant indicating whether the GL is in render, select, or feedback mode. (See **glRenderMode**.)
GL_RGBA_MODE	*ParameterValues* returns a single Boolean value indicating whether the GL is in RGBA mode (True) or color index mode (False). (See **glColor**.)
GL_SCISSOR_BOX	*ParameterValues* returns four values: the *x* and *y* window coordinates of the scissor box, followed by its width and height. (See **glScissor**.)
GL_SCISSOR_TEST	*ParameterValues* returns a single Boolean value indicating whether scissoring is enabled. (See **glScissor**.)
GL_SHADE_MODEL	*ParameterValues* returns one value, a symbolic constant indicating whether the shading mode is flat or smooth. (See **glShadeModel**.)
GL_SECONDARY_COLOR_ARRAY_SIZE_EXT	*ParameterValues* returns one value, the number of components in each entry of the secondary color array, which will be either 3 or 4. (See **glSecondaryColorPointerEXT**.)
GL_SECONDARY_COLOR_ARRAY_STRIDE_EXT	*ParameterValues* returns one value, the byte offset between consecutive entries in the secondary color array. (See **glSecondaryColorPointerEXT**.)
GL_SECONDARY_COLOR_ARRAY_TYPE_EXT	*ParameterValues* returns one value, the data type of each component in the secondary color array. (See **glSecondaryColorPointerEXT**.)
GL_SMOOTH_LINE_WIDTH_GRANULARITY	*ParameterValues* returns one value, the width difference between adjacent supported widths for antialiased lines. (See **glLineWidth**.)
GL_SMOOTH_LINE_WIDTH_RANGE	*ParameterValues* returns two values: the smallest and largest supported widths for antialiased lines. (See **glLineWidth**.)
GL_SMOOTH_POINT_SIZE_GRANULARITY	*ParameterValues* returns one value, the size difference between adjacent supported sizes for antialiased points. (See **glPointSize**.)

GL_SMOOTH_POINT_SIZE_RANGE	*ParameterValues* returns two values: the smallest and largest supported sizes for antialiased points. (See **glPointSize**.)
GL_STENCIL_BITS	*ParameterValues* returns one value, the number of bit planes in the stencil buffer.
GL_STENCIL_CLEAR_VALUE	*ParameterValues* returns one value, the index to which the stencil bit planes are cleared. (See **glClearStencil**.)
GL_STENCIL_FAIL	*ParameterValues* returns one value, a symbolic constant indicating what action is taken when the stencil test fails. (See **glStencilOp**.)
GL_STENCIL_FUNC	*ParameterValues* 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	*ParameterValues* 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	*ParameterValues* 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	*ParameterValues* returns one value, the reference value that is compared with the contents of the stencil buffer. (See **glStencilFunc**.)
GL_STENCIL_TEST	*ParameterValues* returns a single Boolean value indicating whether stencil testing of fragments is enabled. (See **glStencilFunc** and **glStencilOp**.)
GL_STENCIL_VALUE_MASK	*ParameterValues* 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	*ParameterValues* returns one value, the mask that controls writing of the stencil bit planes. (See **glStencilMask**.)
GL_STEREO	*ParameterValues* returns a single Boolean value indicating whether stereo buffers (left and right) are supported.
GL_SUBPIXEL_BITS	*ParameterValues* 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	*ParameterValues* returns a single Boolean value indicating whether 1D texture mapping is enabled. (See **glTexImage1D**.)
GL_TEXTURE_2D	*ParameterValues* returns a single Boolean value indicating whether 2D texture mapping is enabled. (See **glTexImage2D**.)
GL_TEXTURE_1D_BINDING	*ParameterValues* returns a single value, the name of the texture currently bound to the target GL_TEXTURE_1D. The initial value is 0. (See **glBindTexture**.)

GL_TEXTURE_1D_BINDING_EXT	*ParameterValues* returns a single value, the name of the texture currently bound to the target **GL_TEXTURE_1D**. (See **glBindTextureEXT**.) Requires extension **EXT_texture_object**.
GL_TEXTURE_2D_BINDING	*ParameterValues* returns a single value, the name of the texture currently bound to the target **GL_TEXTURE_2D**. The initial value is 0. (See **glBindTexture**.)
GL_TEXTURE_2D_BINDING_EXT	*ParameterValues* returns a single value, the name of the texture currently bound to the target **GL_TEXTURE_2D**. (See **glBindTextureEXT**.) Requires extension **EXT_texture_object**.
GL_TEXTURE_3D_BINDING_EXT	*ParameterValues* returns a single value, the name of the texture currently bound to the target **GL_TEXTURE_3D_EXT**. (See **glBindTexture**.) Requires extension **EXT_texture_object**.
GL_TEXTURE_3D_EXT	*ParameterValues* returns a single Boolean value indicating whether 3D texture mapping is enabled. (See **glTexImage3DEXT**.) Requires extension **EXT_texture3D**.
GL_TEXTURE_COLOR_TABLE_EXT	*ParameterValues* returns a single Boolean value indicating whether the texture color table is enabled. The initial value is **GL_FALSE**. (See **glColorTable**.)
GL_TEXTURE_COORD_ARRAY	*ParameterValues* returns a single Boolean value indicating whether the texture coordinate array is enabled. The initial value is **GL_FALSE**. (See **glTexCoordPointer**.)
GL_TEXTURE_COORD_ARRAY_COUNT_EXT	*ParameterValues* returns one value, the number of elements in the texture coordinate array, counting from the first, that are static. (See **glTexCoordPointerEXT**.) Requires extension **EXT_vertex_array**.
GL_TEXTURE_COORD_ARRAY_EXT	*ParameterValues* returns a single boolean value, indicating whether the texture coordinate array is enabled. (See **glTexCoordPointerEXT**.) Requires extension **EXT_vertex_array**.
GL_TEXTURE_COORD_ARRAY_LIST_STRIDE_IBM	*ParameterValues* returns one value, the byte stride between successive pointers to texture coord lists. The initial value is 0. (See **glTexCoordPointerListIBM**.) Requires extension **IBM_vertex_array_lists**.
GL_TEXTURE_COORD_ARRAY_SIZE	*ParameterValues* returns one value, the number of coordinates per element in the texture coordinate array. The initial value is 4. (See **glTexCoordPointer**.)
GL_TEXTURE_COORD_ARRAY_SIZE_EXT	*ParameterValues* returns one value, the number of coordinates per element in the texture coordinate array. (See **glTexCoordPointerEXT**.) Requires extension **EXT_vertex_array**.
GL_TEXTURE_COORD_ARRAY_STRIDE	*ParameterValues* returns one value, the byte offset between consecutive elements in the texture coordinate array. The initial value is 0. (See **glTexCoordPointer**.)

GL_TEXTURE_COORD_ARRAY_STRIDE_EXT	*ParameterValues* returns one value, the byte offset between consecutive elements in the texture coordinate array. (See **glTexCoordPointerEXT**.) Requires extension **EXT_vertex_array**.
GL_TEXTURE_COORD_ARRAY_TYPE	*ParameterValues* returns one value, the data type of the coordinates in the texture coordinate array. The initial value is GL_FLOAT. (See **glTexCoordPointer**.)
GL_TEXTURE_COORD_ARRAY_TYPE_EXT	*ParameterValues* returns one value, the data type of the coordinates in the texture coordinate array. (See **glTexCoordPointerEXT**.) Requires extension **EXT_vertex_array**.
GL_TEXTURE_ENV_COLOR	*ParameterValues* returns four values: the RGBA 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	*ParameterValues* returns one value, a symbolic constant indicating what texture environment function is currently selected. (See **glTexEnv**.)
GL_TEXTURE_GEN_S	*ParameterValues* returns a single Boolean value indicating whether automatic generation of the S texture coordinate is enabled. (See **glTexGen**.)
GL_TEXTURE_GEN_T	*ParameterValues* returns a single Boolean value indicating whether automatic generation of the T texture coordinate is enabled. (See **glTexGen**.)
GL_TEXTURE_GEN_R	*ParameterValues* returns a single Boolean value indicating whether automatic generation of the R texture coordinate is enabled. (See **glTexGen**.)
GL_TEXTURE_GEN_Q	*ParameterValues* returns a single Boolean value indicating whether automatic generation of the Q texture coordinate is enabled. (See **glTexGen**.)
GL_TEXTURE_MATRIX	*ParameterValues* returns 16 values: the texture matrix on the top of the texture matrix stack. (See **glMatrixMode**.)
GL_TEXTURE_STACK_DEPTH	*ParameterValues* returns one value, the number of matrices on the texture matrix stack . (See **glPushMatrix**.)
GL_TRANSPOSE_COLOR_MATRIX_ARB	*ParameterValues* returns 16 values: the transpose of the color matrix. (See **glLoadNamedMatrixIBM**.) Requires extension **ARB_transpose_matrix**.
GL_TRANSPOSE_MODELVIEW_MATRIX_ARB	*ParameterValues* returns 16 values: the transpose of the modelview matrix on the top of the modelview matrix stack. (See **glMatrixMode**.) Requires extension **ARB_transpose_matrix**.

GL_TRANSPOSE_PROJECTION_MATRIX_ARB	*ParameterValues* returns 16 values: the transpose of the projection matrix on the top of the projection matrix stack. (See **glMatrixMode**.) Requires extension **ARB_transpose_matrix**.
GL_TRANSPOSE_TEXTURE_MATRIX_ARB	*ParameterValues* returns 16 values: the transpose of the texture matrix on the top of the texture matrix stack. (See **glMatrixMode**.) Requires extension **ARB_transpose_matrix**.
GL_UNPACK_ALIGNMENT	*ParameterValues* returns one value, the byte alignment used for reading pixel data from memory. (See **glPixelStore**.)
GL_UNPACK_IMAGE_HEIGHT	*ParameterValues* returns one value, the number of image rows used for reading 3D pixel data from memory. (See **glPixelStore**.)
GL_UNPACK_IMAGE_HEIGHT_EXT	*ParameterValues* returns one value, the number of image rows used for reading 3D pixel data from memory. (See **glPixelStore**.) Requires extension **EXT_texture3D**.
GL_UNPACK_LSB_FIRST	*ParameterValues* 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	*ParameterValues* returns one value, the row length used for reading pixel data from memory. (See **glPixelStore**.)
GL_UNPACK_SKIP_IMAGES	*ParameterValues* returns one value, the number of 2D images skipped before the first pixel of a 3D image is read from memory. (See **glPixelStore**.)
GL_UNPACK_SKIP_IMAGES_EXT	*ParameterValues* returns one value, the number of 2D images skipped before the first pixel of a 3D image is read from memory. (See **glPixelStore**.) Requires extension **EXT_texture3D**.
GL_UNPACK_SKIP_PIXELS	*ParameterValues* returns one value, the number of pixel locations skipped before the first pixel is read from memory. (See **glPixelStore**.)
GL_UNPACK_SKIP_ROWS	*ParameterValues* 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	*ParameterValues* returns a single Boolean value indicating whether the bytes of 2-byte and 4-byte pixel indices and components are swapped after being read from memory. (See **glPixelStore**.)
GL_UPDATE_CLIP_VOLUME_HINT	*ParameterValues* returns a single Boolean value indicating whether the automatic updating of the Clip Volume Hint (through calls to **glClipBoundingBoxIBM**, **glClipBoundingSphereIBM** or **glClipBoundingVerticesIBM**) is enabled. (See **glHint**.) Requires extension **IBM_clip_check**.

GL_VERTEX_ARRAY	*ParameterValues* returns a single Boolean value indicating whether the vertex array is enabled. The initial value is **GL_FALSE**. (See **glVertexPointer**.)
GL_VERTEX_ARRAY_COUNT_EXT	*ParameterValues* returns one value, the number of vertices in the vertex array, counting from the first, that are static. (See **glVertexPointerEXT**.) Requires extension **EXT_vertex_array**.
GL_VERTEX_ARRAY_EXT	*ParameterValues* returns a single boolean value, indicating whether the vertex array is enabled. (See **glVertexPointerEXT**.)
GL_VERTEX_ARRAY_LIST_STRIDE_IBM	*ParameterValues* returns one value, the byte stride between successive pointers to vertex lists. The initial value is 0. (See **glVertexPointerListIBM**.) Requires extension **IBM_vertex_array_lists**.
GL_VERTEX_ARRAY_SIZE_EXT	*ParameterValues* returns one value, the number of coordinates per vertex in the vertex array. (See **glVertexPointerEXT**.) Requires extension **EXT_vertex_array**.
GL_VERTEX_ARRAY_STRIDE	*ParameterValues* returns one value, the byte offset between consecutive vertices in the vertex array. The initial value is 0. (See **glVertexPointer**.)
GL_VERTEX_ARRAY_STRIDE_EXT	*ParameterValues* returns one value, the byte offset between consecutive vertices in the vertex array. (See **glVertexPointerEXT**.) Requires extension **EXT_vertex_array**.
GL_VERTEX_ARRAY_TYPE	*ParameterValues* returns one value, the data type of each coordinate in the vertex array. The initial value is **GL_FLOAT**. (See **glVertexPointer**.)
GL_VERTEX_ARRAY_TYPE_EXT	*ParameterValues* returns one value, the data type of each coordinate in the vertex array. (See **glVertexPointerEXT**.) Requires extension **EXT_vertex_array**.
GL_VIEWPORT	*ParameterValues* returns four values: the *x* and *y* window coordinates of the viewport, followed by its width and height. (See **glViewPort**.)
GL_VISIBILITY_BUFFER_SIZE_IBM	*ParameterValues* returns one value: the maximum number of values that can be stored in the visibility array. (See **glVisibilityBufferIBM**.) Requires extension **IBM_occlusion_cull**.
GL_VISIBILITY_THRESHOLD_IBM	*ParameterValues* returns one value: the number of visible fragments that will be discarded prior to registering a visibility hit. (See **glVisibilityThresholdIBM**.) Requires extension **IBM_occlusion_cull**.
GL_ZOOM_X	*ParameterValues* returns one value, the *x* pixel zoom factor. (See **glPixelZoom**.)
GL_ZOOM_Y	*ParameterValues* returns one value, the *y* pixel zoom factor. (See **glPixelZoom**.)

Many of the Boolean parameters can also be queried more easily using **glIsEnabled**.

Parameters

ParameterName	Specifies the parameter value to be returned. The symbolic constants listed in the Description section are accepted.
ParameterValues	Returns the value or values of the specified parameter.

Error Codes

GL_INVALID_ENUM	*ParameterName* is not an accepted value.
GL_INVALID_OPERATION	The **glGet** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glGetClipPlane** subroutine, **glGetError** subroutine, **glGetLight** subroutine, **glGetMap** subroutine, **glGetMaterial** subroutine, **glGetPixelMap** subroutine, **glGetPointerv** subroutine, **glGetPointervEXT** subroutine, **glGetPolygonStipple** subroutine, **glGetString** subroutine, **glGetTexEnv** subroutine, **glGetTexGen** subroutine, **glGetTexImage** subroutine, **glGetTexLevelParameter** subroutine, **glGetTexParameter** subroutine, **glIsEnabled** subroutine.

glGetClipPlane Subroutine

Purpose

Returns the coefficients of the clipping plane.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glGetClipPlane(GLenum  Plane,
        GLdouble * Equation)
```

Description

The **glGetClipPlane** subroutine returns in *Equation* the four coefficients of the plane equation for *Plane*.

Parameters

Plane	Specifies a clipping plane. The number of clipping planes depends on the implementation; however, 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.

Notes

It is always the case that **GL_CLIP_PLANE***i* = **GL_CLIP_PLANE**0 + *i*.

If an error is generated, no change is made to the contents of *Equation*.

Errors

GL_INVALID_ENUM	*Plane* is not an accepted value.
GL_INVALID_OPERATION	The **glGetClipPlane** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glClipPlane** subroutine.

glGetColorTable Subroutine

Purpose

Return a color lookup table to the user.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
void glGetColorTable(GLenum  target,
                     GLenum  format,
                     GLenum  type,
                     const GLvoid  *table)
void glGetColorTableSGI(GLenum  target,
                        GLenum  format,
                        GLenum  type,
                        const GLvoid  *table)
```

Description

glGetColorTable returns in *table* the contents of the color table specified by *target*. No pixel transfer operations are performed, but pixel storage modes that are applicable to **glReadPixels** are performed.

Color components that are requested to be in the specified *format*, but which are not included in the internal format of the color lookup table, are returned as zero. The assignments of the internal color components to the components requested by *format* are:

Internal Component	Resulting Component
red	red
green	green
blue	blue
alpha	alpha
luminance	red
intensity	red

Parameters

target	Must be **GL_TEXTURE_COLOR_TABLE_EXT**.
format	is the format of the pixel data in *table*. The allowable values are **GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_RGB, GL_BGR, GL_RGBA**, **GL_BGRA**, **GL_422_EXT**, **GL_422_REV_EXT**, **GL_422_AVERAGE_EXT**, and **GL_422_REV_AVERAGE_EXT**.
type	is the type of the pixel data in *table*. The allowable values are **GL_UNSIGNED_BYTE, GL_BYTE, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, GL_FLOAT, GL_UNSIGNED_BYTE_3_3_2, GL_UNSIGNED_BYTE_2_3_3_REV, GL_UNSIGNED_SHORT_5_6_5, GL_UNSIGNED_SHORT_5_6_5_REV, GL_UNSIGNED_SHORT_4_4_4_4, GL_UNSIGNED_SHORT_4_4_4_4_REV, GL_UNSIGNED_SHORT_5_5_5_1, GL_UNSIGNED_SHORT_1_5_5_5_REV, GL_UNSIGNED_INT_8_8_8_8, GL_UNSIGNED_INT_8_8_8_8_REV, GL_UNSIGNED_INT_10_10_10_2**, and **GL_UNSIGNED_INT_2_10_10_10_REV**.
table	is a pointer to a one-dimensional array of pixel data that will be loaded with the contents of the color table.

Notes

GL_TEXTURE_COLOR_TABLE_SGI is an alias for **GL_TEXTURE_COLOR_TABLE_EXT**, and these tokens may be used interchangeably.

GL_PROXY_TEXTURE_COLOR_TABLE_SGI is an alias for **GL_PROXY_TEXTURE_COLOR_TABLE_EXT**, and these tokens may be used interchangeably.

Error Codes

GL_INVALID_ENUM	is generated if *target* is not one of the allowable values.
GL_INVALID_ENUM	is generated if *format* is not one of the allowable values.
GL_INVALID_ENUM	is generated if *type* is not one of the allowable values.
GL_INVALID_OPERATION	is generated if *type* is one of **GL_UNSIGNED_BYTE_3_3_2, GL_UNSIGNED_BYTE_2_3_3_REV, GL_UNSIGNED_SHORT_5_6_5**, or **GL_UNSIGNED_SHORT_5_6_5_REV**, and *format* is not **GL_RGB**.
GL_INVALID_OPERATION	is generated if *type* is one of **GL_UNSIGNED_SHORT_4_4_4_4, GL_UNSIGNED_SHORT_4_4_4_4_REV, GL_UNSIGNED_SHORT_5_5_5_1, GL_UNSIGNED_SHORT_1_5_5_5_REV, GL_UNSIGNED_SHORT_8_8_8_8, GL_UNSIGNED_SHORT_8_8_8_8_REV, GL_UNSIGNED_SHORT_10_10_10_2**, or **GL_UNSIGNED_SHORT_2_10_10_10_REV**, and *format* is neigher **GL_RGBA** nor **GL_BGRA**.

GL_INVALID_OPERATION	is generated if **glColorTable** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glColorSubTable** subroutine, the **glColorTableParameter** subroutine, the **glGetColorTableParameter** subroutine.

glGetColorTableParameter Subroutine

Purpose

Returns attributes used when loading a color table.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
void glGetColorTableParameterfv(GLenum  target,
                                GLenum  pname,
                                const GLfloat  *params)
void glGetColorTableParameteriv(GLenum  target,
                                GLenum  pname,
                                const GLint  *params)
void glGetColorTableParameterfvSGI(GLenum  target,
                                   GLenum  pname,
                                   const GLfloat  *params)
void glGetColorTableParameterivSGI(GLenum  target,
                                   GLenum  pname,
                                   const GLint  *params)
```

Description

This subroutine returns parameters specific to color table *target*.

When *pname* is set to **GL_COLOR_TABLE_SCALE** or **GL_COLOR_TABLE_BIAS**, **glGetColorTableParameter** returns the color table scale or bias parameters for the table specified by *target*. For these queries, *target* must be set to **GL_TEXTURE_COLOR_TABLE_EXT** and *params* points to an array of four elements, which receive the scale or bias factors for red, green, blue, and alpha, in that order.

glGetColorTableParameter can also be used to retrieve the format and size parameters for a color table. For thes queries, set *target* to any of the six targets listed above. The *format* and *size* parameters are set by **glColorTable**.

The following table lists the format and size parameters that may be queried. For each symbolic constant listed below for *pname*, *params* must point to an array of the given length, and will receive the values indicated.

```
Parameter                      N    Meaning

GL_COLOR_TABLE_FORMAT          1    Internal format (e.g. GL_RGBA)
GL_COLOR_TABLE_WIDTH           1    Number of elements in the table
GL_COLOR_TABLE_RED_SIZE        1    Size of red component, in bits
GL_COLOR_TABLE_GREEN_SIZE      1    Size of green component, in bits
GL_COLOR_TABLE_BLUE_SIZE       1    Size of blue component, in bits
GL_COLOR_TABLE_ALPHA_SIZE      1    Size of alpha component, in bits
GL_COLOR_TABLE_LUMINANCE_SIZE  1    Size of luminance component, in bits
GL_COLOR_TABLE_INTENSITY_SIZE  1    Size of intensity component, in bits
```

Parameters

target — is the target color table. Must be **GL_TEXTURE_COLOR_TABLE_EXT**, or **GL_PROXY_TEXTURE_COLOR_TABLE_EXT**.

pname — is the symbolic name of a texture color lookup table parameter. Must be one of **GL_COLOR_TABLE_SCALE, GL_COLOR_TABLE_BIAS, GL_COLOR_TABLE_FORMAT, GL_COLOR_TABLE_WIDTH, GL_COLOR_TABLE_RED_SIZE, GL_COLOR_TABLE_GREEN_SIZE, GL_COLOR_TABLE_BLUE_SIZE, GL_COLOR_TABLE_ALPHA_SIZE, GL_COLOR_TABLE_LUMINANCE_SIZE**, or **GL_COLOR_TABLE_INTENSITY_SIZE**.

params — is a pointer to an array where the values of the paramaters will be stored.

Notes

GL_TEXTURE_COLOR_TABLE_SGI is an alias for **GL_TEXTURE_COLOR_TABLE_EXT**, and these tokens may be used interchangeably. **GL_PROXY_TEXTURE_COLOR_TABLE_SGI** is an alias for **GL_PROXY_TEXTURE_COLOR_TABLE_EXT**, and these tokens may be used interchangeably.

Error Codes

GL_INVALID_ENUM — is generated if *target* is not one of the allowable values.

GL_INVALID_ENUM — is generated if *pname* is not one of the allowable values.

GL_INVALID_OPERATION — is generated if **glColorTableParameter** is executed between the execution of a **glBegin** and the corresponding execution of **glEnd**.

Files

/usr/include/GL/gl.h — Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glColorSubTable** subroutine, the **glColorTableParameter** subroutine.

glGetError Subroutine

Purpose

Returns error information.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLenum glGetError( void )
```

Description

The **glGetError** subroutine 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 the GL 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. Therefore, **glGetError** should always be called in a loop, until it returns **GL_NO_ERROR**, if all error flags are to be reset.

Initially, all error flags are set to **GL_NO_ERROR**.

The currently defined errors are:

GL_NO_ERROR	No error has been recorded. The value of this symbolic constant is guaranteed to be 0 (zero).
GL_INVALID_ENUM	An unacceptable value is specified for an enumerated argument. The offending command 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 command 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 command is ignored, having no side effect other than to set the error flag.
GL_STACK_OVERFLOW	This command would cause a stack overflow. The offending command is ignored, having no side effect other than to set the error flag.
GL_STACK_UNDERFLOW	This command would cause a stack underflow. The offending command 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 command. The state of the GL is undefined, except for the state of the error flags, after this error is recorded.
GL_TABLE_TOO_LARGE	The specified table is too large.

When an error flag is set, results of a GL operation are undefined only if **GL_OUT_OF_MEMORY** has occurred. In all other cases, the command generating the error is ignored and has no effect on the GL state or frame buffer contents.

Errors

GL_INVALID_OPERATION	The **glGetError** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes fo OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine.

glGetLight Subroutine

Purpose

Returns light source parameter values.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glGetLightfv(GLenum  Light,
     GLenum  ParameterName,
     GLfloat * ParameterValues)

void glGetLightiv(GLenum  Light,
     GLenum  ParameterName,
     GLint * ParameterValues)
```

Description

The **glGetLight** subroutine returns in *ParameterValues* the value or values of a light source parameter. *Light* names the light and is a symbolic name of the form **GL_LIGHT***i* for 0 < *i* < **GL_MAX_LIGHTS**, where **GL_MAX_LIGHTS** is an implementation-dependent constant that is greater than or equal to 8. *ParameterName* specifies one of 10 light source parameters, again by symbolic name.

The parameters are:

GL_AMBIENT	*ParameterValues* 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	*ParameterValues* 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	*ParameterValues* 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	*ParameterValues* 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	*ParameterValues* 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 identity 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	*ParameterValues* 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	*ParameterValues* 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	*ParameterValues* 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	*ParameterValues* 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	*ParameterValues* 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.

Parameters

Light	Specifies a light source. The number of possible lights depends on the implementation; however, at least eight lights are supported. They are identified by symbolic names of the form **GL_LIGHT***i* where 0 < *i* < **GL_MAX_LIGHTS**.
ParameterName	Specifies a light source parameter for *Light*. Accepted symbolic names are **GL_AMBIENT**, **GL_DIFFUSE**, **GL_SPECULAR**, **GL_POSITION**, **GL_SPOT_DIRECTION**, **GL_SPOT_EXPONENT**, **GL_SPOT_CUTOFF**, **GL_CONSTANT_ATTENUATION**, **GL_LINEAR_ATTENUATION**, and **GL_QUADRATIC_ATTENUATION**.
ParameterValues	Returns the requested data.

Notes

It is always the case that **GL_LIGHT***i* = **GL_LIGHT**0 + *i*.

If an error is generated, no change is made to the contents of *ParameterValues*.

Errors

GL_INVALID_ENUM	Either *Light* or *ParameterName* is not an accepted value.
GL_INVALID_OPERATION	The **glGetLight** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glLight** subroutine.

---

glGetMap Subroutine

Purpose

Returns evaluator parameters.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glGetMapdv(GLenum  Target,
     GLenum  Query,
     GLdouble * v)

void glGetMapfv(GLenum  Target,
     GLenum  Query,
     GLfloat * v)

void glGetMapiv(GLenum  Target,
     GLenum  Query,
     GLint * v)
```

Description

The **glMap1** and **glMap2** subroutines define evaluators. The **glGetMap** subroutine returns evaluator parameters. *Target* chooses a map, *Query* selects a specific parameter, and *v* points to storage where the values are returned. (See the **glMap1** and **glMap2** subroutines for a description of the acceptable values for the *Target* parameter.)

Query can assume the following values:

GL_COEFF	*v* returns the control points for the evaluator function. One-dimensional (1D) evaluators return *order* control points, and two-dimensional (2D) evaluators return *uorder* x *vorder* control points. Each control point consists of 1, 2, 3, or 4 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	*v* 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	*v* returns the linear *u* and *v* mapping parameters. One-dimensional evaluators return two values, *u1* and *u2*, as specified by **glMap1**. Two-dimensional evaluators return four values, *u1*, *u2*, *v1*, and *v2*, as specified by **glMap2**. Integer values, when requested, are computed by rounding the internal floating-point values to the nearest integer values.

Parameters

Target	Specifies the symbolic name of a map. Accepted values are **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. Symbolic names **GL_COEFF**, **GL_ORDER**, and **GL_DOMAIN** are accepted.
v	Returns the requested data.

Notes

If an error is generated, no change is made to the contents of *v*.

Errors

GL_INVALID_ENUM	Either *Target* or *Query* is not an accepted value.
GL_INVALID_OPERATION	The **glGetMap** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glEvalCoord** subroutine, **glMap1** subroutine, **glMap2** subroutine.

glGetMaterial Subroutine

Purpose

Returns material parameters.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glGetMaterialfv(GLenum  Face,
     GLenum  ParameterName,
     GLfloat * ParameterValues)

void glGetMaterialiv(GLenum  Face,
     GLenum  ParameterName,
     GLint * ParameterValues)
```

Description

The **glGetMaterial** subroutine returns in *ParameterValues* the value or values of parameter *ParameterName* of material *Face*. Six parameters are defined:

GL_AMBIENT
: *ParameterValues* 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
: *ParameterValues* 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
: *ParameterValues* 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
: *ParameterValues* 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
: *ParameterValues* 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
: *ParameterValues* returns three integer or floating-point values representing the ambient, diffuse, and specular indices of the material. These indices are used only for color index lighting. (The other parameters are all used only for red, green, blue, and alpha lighting.) Integer values, when requested, are computed by rounding the internal floating-point values to the nearest integer values.

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.

ParameterName
: Specifies the material parameter to return. **GL_AMBIENT**, **GL_DIFFUSE**, **GL_SPECULAR**, **GL_EMISSION**, **GL_SHININESS**, and **GL_COLOR_INDEXES** are accepted.

ParameterValues	Returns the requested data.

Notes

If an error is generated, no change is made to the contents of *ParameterValues*.

Errors

GL_INVALID_ENUM	Either *Face* or *ParameterName* is not an accepted value.
GL_INVALID_OPERATION	The **glGetMaterial** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glMaterial** subroutine.

---

glGetPixelMap Subroutine

Purpose

Returns the specified pixel map.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glGetPixelMapfv(GLenum  Map,
     GLfloat  *Values)

void glGetPixelMapuiv(GLenum  Map,
     GLuint  *Values)

void glGetPixelMapusv(GLenum  Map,
     GLushort  *Values)
```

Description

The **glGetPixelMap** subroutine returns in the *Values* parameter the contents of the pixel map specified by the *Map* parameter. Pixel maps are used during the execution of **glReadPixels**, **glDrawPixels**, **glCopyPixels**, **glTexImage1D**, and **glTexImage2D** to map color indices, stencil indices, 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 0 (zero). Returned unsigned integer values are undefined if the map value was not in the range [0,1].

To determine the required size of the *Map* parameter, call the **glGet** subroutine with the appropriate symbolic constant.

Parameters

Map	Specifies 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.

Notes

If an error is generated, no change is made to the contents of the *Values* parameter.

Errors

GL_INVALID_ENUM	*Map* is not an accepted value.
GL_INVALID_OPERATION	The **glGetPixelMap** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glGetPixelMap** subroutine are as follows. (See the **glGet** subroutine for more information.)

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**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glCopyPixels** subroutine, **glDrawPixels** subroutine, **glPixelMap** subroutine, **glPixelTransfer** subroutine, **glReadPixels** subroutine, **glTexImage1D** subroutine, **glTexImage2D** subroutine.

glGetPointerv Subroutine

Purpose

Returns the address of the specified pointer.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glGetPointerv(GLenum  pname,
     GLvoid*  *params)
```

Description

The **glGetPointerv** subroutine returns pointer information. The *pname* parameter is a symbolic constant indicating the pointer to be returned, and *params* is a pointer to a location in which to place the returned data.

Use **glDrawArrays**, **glMultiDrawArraysEXT**, or **glMultiModeDrawArraysIBM** to construct a sequence of primitives from prespecified vertex and vertex attribute arrays. Use **glArrayElement** to specify primitives by indexing vertices and vertex attributes. Use **glDrawElements**, **glMultiDrawElementsEXT**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

If enabled, the various vertex arrays are used when **glDrawArrays**, **glDrawElements**, **glArrayElements**, **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** is called.

Parameters

pname Specifies the array or buffer pointer to be returned. The following symbolic constants are accepted:

- **GL_COLOR_ARRAY_LIST_IBM**
- **GL_COLOR_ARRAY_POINTER**
- **GL_EDGE_FLAG_ARRAY_LIST_IBM**
- **GL_EDGE_FLAG_ARRAY_POINTER**
- **GL_FEEDBACK_BUFFER_POINTER**
- **GL_FOG_COORDINATE_ARRAY_LIST_IBM**
- **GL_FOG_COORDINATE_ARRAY_POINTER_EXT**
- **GL_INDEX_ARRAY_LIST_IBM**
- **GL_INDEX_ARRAY_POINTER**
- **GL_NORMAL_ARRAY_LIST_IBM**
- **GL_NORMAL_ARRAY_POINTER**
- **GL_SECONDARY_COLOR_ARRAY_LIST_IBM**
- **GL_SECONDARY_COLOR_ARRAY_POINTER**
- **GL_SELECTION_BUFFER_POINTER**
- **GL_TEXTURE_COORD_ARRAY_LIST_IBM**
- **GL_TEXTURE_COORD_ARRAY_POINTER**
- **GL_VERTEX_ARRAY_LIST_IBM**
- **GL_VERTEX_ARRAY_POINTER**
- **GL_VISIBILITY_BUFFER_POINTER_IBM**

params Returns the pointer value specified by pname.

Notes

The **glGetPointerv** subroutine is available only if the GL version is 1.1 or greater.

The ″***_ARRAY_LIST_IBM**″ symbolic constants are only accepted if the **IBM_vertex_array_list** extension is defined.

The ***_ARRAY_LIST_IBM** symbolic constants are only accepted if the **IBM_vertex_array_list** extension is defined.

The ″**GL_FOG_COORDINATE_***″ symbolic constants are only accepted if the **EXT_fog_coord** extension is defined.

The ″**GL_SECONDARY_COLOR_***″ symbolic constants are only accepted if the **EXT_secondary_color** extension is defined.

The **GL_VISIBILITY_BUFFER_POINTER_IBM** symbolic constant is only accepted if the **IBM_occlusion_cull** extension is supported.

The pointers are all client side state.

The initial value for each pointer is 0.

Error Codes

GL_INVALID_ENUM is generated if *pname* is not an accepted value.

Related Information

The **glArrayElement** subroutine, **glColorPointer** subroutine, **glColorPointerListIBM** subroutine, **glDrawArrays** subroutine, **glEdgeFlagPointer** subroutine, **glEdgeFlagPointerListIBM** subroutine, **glFeedbackBuffer** subroutine, **glIndexPointer** subroutine, **glIndexPointerListIBM** subroutine, **glNormalPointer** subroutine, **glNormalPointerListIBM** subroutine, **glSelectBuffer** subroutine, **glTexCoordPointer** subroutine, **glTexCoordPointerListIBM** subroutine, **glVertexPointer** subroutine, **glVertexPointerListIBM** subroutine, **glVisibilityBufferIBM** subroutine.

glGetPointervEXT Subroutine

Purpose

Returns the address of a vertex data array.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glGetPointervEXT(GLenum  pname,
     GLvoid  **params)
```

Description

glGetPointervEXT returns array pointer information. *pname* is a symbolic constant indicating the array pointer to be returned, and *params* is a pointer to a location in which to place the returned data.

Use **glDrawArrays**, **glMultiDrawArraysEXT**, or **glMultiModeDrawArraysIBM** to construct a sequence of primitives from prespecified vertex and vertex attribute arrays. Use **glArrayElement** to specify primitives by indexing vertices and vertex attributes. Use **glDrawElements**, **glMultiDrawElementsEXT**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

If enabled, the various vertex arrays are used when **glDrawArrays**, **glDrawElements**, **glArrayElements**, **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** is called.

Parameters

pname	Specifies the array pointer to be returned. Symbolic constants **GL_VERTEX_ARRAY_POINTER_EXT**, **GL_NORMAL_ARRAY_POINTER_EXT**, **GL_COLOR_ARRAY_POINTER_EXT**, **GL_INDEX_ARRAY_POINTER_EXT**, **GL_TEXTURE_COORD_ARRAY_POINTER_EXT**, **GL_EDGE_FLAG_ARRAY_POINTER_EXT**, are accepted.
***params*	returns the array pointer value specified by *pname*.

Notes

The array pointers are client side state.

glGetPointervEXT is part of the _extname(EXT_vertex_array) extension, not part of the core GL command set. If _extstring(EXT_vertex_array) is included in the string returned by **glGetString**, when called with argument **GL_EXTENSIONS**, extension _extname(EXT_vertex_array) is supported.

Errors

GL_INVALID_ENUM is generated if *pname* is not an accepted value.

File

/usr/include/GL/glext.h	Contains extensions to C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glArrayElementEXT** subroutine, **glColorPointerEXT** subroutine, **glDrawArraysEXT** subroutine, **glEdgeFlagPointerEXT** subroutine, **glIndexPointerEXT** subroutine, **glNormalPointerEXT** subroutine, **glTexCoordPointerEXT** subroutine, **glVertexPointerEXT** subroutine.

glGetPolygonStipple Subroutine

Purpose

Returns the polygon stipple pattern.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glGetPolygonStipple(GLubyte  *Mask)
```

Description

The **glGetPolygonStipple** subroutine returns to *Mask* a 32 x 32 polygon stipple pattern. The pattern is packed into memory as if the following values were called:

- **glReadPixels** with both *Height* and *Width* equal to 32.
- *Type* is **GL_BITMAP**.
- *Format* is **GL_COLOR_INDEX**.

In addition, the pattern is packed into memory as if the stipple pattern was stored in an internal 32 x 32 color index buffer. Unlike **glReadPixels**, however, pixel transfer operations (shift, offset, pixel map) are not applied to the returned stipple image.

Parameters

Mask	Returns the stipple pattern.

Notes

If an error is generated, no change is made to the contents of the *Mask* parameter.

Errors

GL_INVALID_OPERATION	The **glGetPolygonStipple** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glPixelStore** subroutine, **glPixelTransfer** subroutine, **glPolygonStipple** subroutine, **glReadPixels** subroutine.

---

glGetString Subroutine

Purpose

Returns a string describing the current GL connection.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
const GLubyte * glGetString(GLenum  Parameter1)
```

Description

The **glGetString** subroutine returns a pointer to a static string describing some aspect of the current GL connection. The *Parameter1* parameter can be one of the following values:

GL_VENDOR	Returns the name of the company responsible for this GL 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 GL.

Because GL 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 GL usage based on known performance characteristics of these platforms. Together, strings **GL_VENDOR** and **GL_RENDERER** uniquely specify a platform, and do not change from release to release. These strings 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 do not include space characters and are separated by space characters in the **GL_EXTENSIONS** string, and all strings are null-terminated.

Parameters

Parameter1	Specifies a symbolic constant, one of **GL_VENDOR**, **GL_RENDERER**, **GL_VERSION**, or **GL_EXTENSIONS**.

Notes

If an error is generated, **glGetString** returns 0 (zero).

Errors

GL_INVALID_ENUM	*Parameter1* is not an accepted value.
GL_INVALID_OPERATION	The **glGetString** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine.

glGetTexEnv Subroutine

Purpose

Returns texture environment parameters.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glGetTexEnvfv(GLenum  Target,
     GLenum  ParameterName,
     GLfloat  *ParameterValues)

void glGetTexEnviv(GLenum  Target,
     GLenum  ParameterName,
     GLint  *ParameterValues)
```

Description

The **glGetTexEnv** subroutine returns in the *ParameterValues* parameter selected values of a texture environment that was specified with **glTexEnv**. The *Target* parameter specifies a texture environment. Currently only the **GL_TEXTURE_ENV** texture environment is defined and supported.

ParameterName names a specific texture environment parameter. The parameters are:

GL_TEXTURE_ENV_MODE	*ParameterValues* returns the single-valued texture environment mode, a symbolic constant.
GL_TEXTURE_ENV_COLOR	*ParameterValues* 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.
GL_COMBINE_RGB_EXT	*ParameterValues* returns the currently defined function to be used when blending texture RGB values in ″combine″ mode.
GL_COMBINE_ALPHA_EXT	*ParameterValues* returns the currently defined function to be used when blending texture Alpha values in ″combine″ mode.
GL_SOURCE0_RGB_EXT	*ParameterValues* returns the currently defined value used to determine the source for RGB Operand 0.
GL_SOURCE1_RGB_EXT	*ParameterValues* returns the currently defined value used to determine the source for RGB Operand 1.
GL_SOURCE2_RGB_EXT	*ParameterValues* returns the currently defined value used to determine the source for RGB Operand 2.
GL_SOURCE0_ALPHA_EXT	*ParameterValues* returns the currently defined value used to determine the source for Alpha Operand 0.
GL_SOURCE1_ALPHA_EXT	*ParameterValues* returns the currently defined value used to determine the source for Alpha Operand 1.
GL_SOURCE2_ALPHA_EXT	*ParameterValues* returns the currently defined value used to determine the source for Alpha Operand 2.
GL_OPERAND0_RGB_EXT	*ParameterValues* returns the currently defined RGB Operand 0.
GL_OPERAND1_RGB_EXT	*ParameterValues* returns the currently defined RGB Operand 1.
GL_OPERAND2_RGB_EXT	*ParameterValues* returns the currently defined RGB Operand 2.
GL_OPERAND0_ALPHA_EXT	*ParameterValues* returns the currently defined RGB Alpha 0.
GL_OPERAND1_ALPHA_EXT	*ParameterValues* returns the currently defined RGB Alpha 1.
GL_OPERAND2_ALPHA_EXT	*ParameterValues* returns the currently defined RGB Alpha 2.
GL_RGB_SCALE_EXT	*ParameterValues* returns the floating-point value which is used to do the final scale on the RGB channels.
GL_ALPHA_SCALE	*ParameterValues* returns the floating-point number which is used to do the final scale on the alpha channel.

Parameters

Target	Specifies a texture environment. Must be **GL_TEXTURE_ENV**.

ParameterName Specifies the symbolic name of a texture environment parameter. Accepted values are:

- **GL_TEXTURE_ENV_MODE**
- **GL_TEXTURE_ENV_COLOR**
- **GL_COMBINE_RGB_EXT**
- **GL_COMBINE_ALPHA_EXT**
- **GL_SOURCE0_RGB_EXT**
- **GL_SOURCE1_RGB_EXT**
- **GL_SOURCE2_RGB_EXT**
- **GL_SOURCE0_ALPHA_EXT**
- **GL_SOURCE1_ALPHA_EXT**
- **GL_SOURCE2_ALPHA_EXT**
- **GL_OPERAND0_RGB_EXT**
- **GL_OPERAND1_RGB_EXT**
- **GL_OPERAND2_RGB_EXT**
- **GL_OPERAND0_ALPHA_EXT**
- **GL_OPERAND1_ALPHA_EXT**
- **GL_OPERAND2_ALPHA_EXT**
- **GL_RGB_SCALE_EXT**
- **GL_ALPHA_SCALE**

ParameterValues Returns the requested data.

Notes

If an error is generated, no change is made to the contents of *ParameterValues*.

Errors

GL_INVALID_ENUM	Either *Target* or *ParameterName* is not an accepted value.
GL_INVALID_OPERATION	The **glGetTexEnv** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glTexEnv** subroutine.

glGetTexGen Subroutine

Purpose

Returns texture coordinate generation parameters.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glGetTexGendv(GLenum  Coordinate,
     GLenum  ParameterName,
     GLdouble  *ParameterValues)

void glGetTexGenfv(GLenum  Coordinate,
     GLenum  ParameterName,
     GLfloat  *ParameterValues)

void glGetTexGeniv(GLenum  Coordinate,
     GLenum  ParameterName,
     GLint  *ParameterValues)
```

Description

The **glGetTexGen** subroutine returns in *ParameterValues* selected parameters of a texture coordinate generation function specified with **glTexGen**. *Coordinate* names one of the (*s*, *t*, *r*, *q*) texture coordinates, using the symbolic constant **GL_S**, **GL_T**, **GL_R**, or **GL_Q**.

ParameterName specifies one of three symbolic names:

GL_TEXTURE_GEN_MODE	*ParameterValues* returns the single-valued texture generation function, a symbolic constant.
GL_OBJECT_PLANE	*ParameterValues* 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	*ParameterValues* 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.

Parameters

Coordinate	Specifies a texture coordinate. Must be **GL_S**, **GL_T**, **GL_R**, or **GL_Q**.
ParameterName	Specifies the symbolic name of the values to be returned. Must be either **GL_TEXTURE_GEN_MODE** or the name of one of the texture generation plane equations, either **GL_OBJECT_PLANE** or **GL_EYE_PLANE**.
ParameterValues	Returns the requested data.

Notes

If an error is generated, no change is made to the contents of *ParameterValues*.

Errors

GL_INVALID_ENUM	Either *Coordinate* or *ParameterName* is not an accepted value.
GL_INVALID_OPERATION	The **glGetTexGen** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glTexGen** subroutine.

---

glGetTexImage Subroutine

Purpose

Returns a texture image.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glGetTexImage(GLenum  Target,
                   GLint  Level,
                   GLenum  Format,
                   GLenum  Type,
                   GLvoid  *Pixels)
```

Description

The **glGetTexImage** subroutine returns a texture image and places it in the *Pixels* parameter. *Target* specifies whether the desired texture image is one specified by **glTexImage1D** (**GL_TEXTURE_1D**), **glTexImage2D** (**GL_TEXTURE_2D**), **glTexImage3D** (**GL_TEXTURE_3D**), or by **glTexImage3DEXT** (**GL_TEXTURE_3D_EXT**). *Level* specifies the level-of-detail number of the desired image. *Format* and *Type* specify the format and type of the desired image array. (See the **glTexImage1D** and **glDrawPixels** subroutines for a description of the acceptable values for the *Format* and *Type* parameters, respectively.)

Operation of **glGetTexImage** is best understood by considering the selected internal four-component texture image to be a red, green, blue, alpha (RGBA) color buffer that is 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 0 (zero), *Width* set to the width of the texture image (including the border if one was specified), and *Height* set to 1 (one) for one-dimensional (1D) images, or to the height of the texture image (including the border if one was specified) for two-dimensional (2D) 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 0.
- Two-component textures are treated as RGBA buffers with red set to the value of component 0, alpha set to the value of component 1, and green and blue set to 0.
- Three-component textures are treated as RGBA buffers with red set to component 0, green set to component 1, blue set to component 2, and alpha set to 0.

To determine the required size of *Pixels*, use the **glGetTexLevelParameter** subroutine to ascertain the dimensions of the internal texture image, then scale the required number of pixels by the storage required for each pixel, based on *Format* and *Type*. Be sure to consider the pixel storage parameters, especially **GL_PACK_ALIGNMENT**.

Notes

If an error is generated, no change is made to the contents of *Pixels*.

Format of **GL_ABGR_EXT** is part of the _extname (EXT_abgr) extension, not part of the core GL command set.

Target of **GL_TEXTURE_3D_EXT** is part of the _extname (EXT_texture3D) extension, not part of the core GL command set.

Parameters

Target	Specifies which texture is to be obtained. **GL_TEXTURE_1D**, **GL_TEXTURE_2D**, **GL_TEXTURE_3D**, and **GL_TEXTURE_3D_EXT** are accepted.
Level	Specifies 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	Specifies a pixel format for the returned data. The supported formats are **GL_RED**, **GL_GREEN**, **GL_BLUE**, **GL_ALPHA**, **GL_RGB**, **GL_RGBA**, **GL_ABGR_EXT**, **GL_LUMINANCE**, and **GL_LUMINANCE_ALPHA**.
Type	Specifies 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 the *Type* parameter.

Errors

GL_INVALID_ENUM	Either *Target*, *Format*, or *Type* is not an accepted value.
GL_INVALID_VALUE	*Level* is less than 0 or greater than log2*max*, where *max* is the returned value of **GL_MAX_TEXTURE_SIZE**.
GL_INVALID_OPERATION	The **glGetTexImage** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glGetTexImage** subroutine are as follows. (See the **glGet** subroutine for more information.)

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 arguments **GL_PACK_ALIGNMENT** and others.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glDrawPixels** subroutine, **glReadPixels** subroutine, **glTexImage1D** subroutine, **glTexImage2D** subroutine.

glGetTexLevelParameter Subroutine

Purpose

Returns texture parameter values for a specific level of detail.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glGetTexLevelParameterfv(GLenum  target,
                              GLint  level,
                              GLenum  pname,
                              GLfloat * params)

void glGetTexLevelParameteriv(GLenum  target,
                              GLint  level,
                              GLenum  pname,
                              GLint  *params)
```

Description

The **glGetTexLevelParameter** subroutine 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**, **GL_TEXTURE_2D**, **GL_TEXTURE_3D**, **GL_TEXTURE_3D_EXT**, **GL_PROXY_TEXTURE_1D**, **GL_PROXY_TEXTURE_2D**, **GL_PROXY_TEXTURE_3D**, or **GL_PROXY_TEXTURE_3D_EXT**.

The **GL_MAX_TEXTURE_SIZE** parameter reports the largest square texture image which can be accomodated with mipmaps and borders (but a long skinny texture, or a texture without mipmaps and borders, may easily fit in texture memory). The proxy targets allow the user to more accurately query whether the GL can accomodate a texture of a given configuration. If the texture cannot be accomodated, the texture state variables (which may be queried with **glGetTexLevelParameter**) are set to 0. If the texture can be accomodated the texture state values will be set as they would be set for a non-proxy target.

The *pname* parameter specifies the texture parameter whose value or values will be returned.

The accepted parameter names are as follows:

GL_TEXTURE_ALPHA_SIZE	The internal storage resolution of an individual *alpha* component. The resolution chosen by the GL will be a close match for the resolution requested by the user with the component argument of **glTexImage1D**, **glTexImage2D**, or **glTexImage3DEXT**. The initial value is 0.
GL_TEXTURE_BLUE_SIZE	The internal storage resolution of an individual *blue* component. The resolution chosen by the GL will be a close match for the resolution requested by the user with the component argument of **glTexImage1D**, **glTexImage2D**, or **glTexImage3DEXT**. The initial value is 0.
GL_TEXTURE_BORDER	*params* returns a single value, the width in pixels of the border of the texture image. The inital value is 0.
GL_TEXTURE_DEPTH	*params* returns a single value, the depth of the texture image. This value includes the border of the texture image. The initial value is 0.

GL_TEXTURE_DEPTH_EXT	*params* returns a single value, the depth of the texture image. This value includes the border of the texture image. The initial value is 0.
GL_TEXTURE_GREEN_SIZE	The internal storage resolution of an individual *green* component. The resolution chosen by the GL will be a close match for the resolution requested by the user with the component argument of **glTexImage1D**, **glTexImage2D**, or **glTexImage3DEXT**. The initial value is 0.
GL_TEXTURE_HEIGHT	*params* returns a single value, the height of the texture image. This value includes the border of the texture image. The initial value is 0.
GL_TEXTURE_INTENSITY_SIZE	The internal storage resolution of an individual component. The resolution chosen by the GL will be a close match for the resolution requested by the user with the component argument of **glTexImage1D** or **glTexImage2D**. The initial value is 0.
GL_TEXTURE_INTERNAL_FORMAT	*params* returns a single value, the requested internal format of the texture image.
GL_TEXTURE_LUMINANCE_SIZE	The internal storage resolution of an individual *luminance* component. The resolution chosen by the GL will be a close match for the resolution requested by the user with the component argument of **glTexImage1D**, **glTexImage2D**, or **glTexImage3DEXT**. The initial value is 0.
GL_TEXTURE_RED_SIZE	The internal storage resolution of an individual *red* component. The resolution chosen by the GL will be a close match for the resolution requested by the user with the component argument of **glTexImage1D**, **glTexImage2D**, or **glTexImage3DEXT**. The initial value is 0.
GL_TEXTURE_WIDTH	*params* returns a single value, the width of the texture image. This value includes the border of the texture image. The initial value is 0.

Parameters

target	Specifies the symbolic name of the target texture, either **GL_TEXTURE_1D**, **GL_TEXTURE_2D**, **GL_TEXTURE_3D**, **GL_PROXY_TEXTURE_1D**, or **GL_PROXY_TEXTURE_2D**, **GL_PROXY_TEXTURE_3D**, **GL_PROXY_TEXTURE_3D_EXT**, **GL_TEXTURE_3D_EXT**.
level	Specifies the level-of-detail number of the desired image. Level 0 is the base image level. Level n is the nth mipmap reduction image.
pname	Specifies the symbolic name of a texture parameter. **GL_TEXTURE_DEPTH**, **GL_TEXTURE_DEPTH_EXT**, **GL_TEXTURE_WIDTH**, **GL_TEXTURE_HEIGHT**, **GL_TEXTURE_INTERNAL_FORMAT**, **GL_TEXTURE_BORDER**, **GL_TEXTURE_RED_SIZE**, **GL_TEXTURE_GREEN_SIZE**, **GL_TEXTURE_BLUE_SIZE**, **GL_TEXTURE_ALPHA_SIZE**, **GL_TEXTURE_LUMINANCE_SIZE**, and **GL_TEXTURE_INTENSITY_SIZE** are accepted.
params	Returns the requested data.

Notes

If an error is generated, no change is made to the contents of *params*.

The **GL_TEXTURE_INTERNAL_FORMAT** parameter is only available if the GL version is 1.1 or greater. In version 1.0, use **GL_TEXTURE_COMPONENTS** instead.

Errors

GL_INVALID_ENUM is generated if *target* or *pname* is not an accepted value.

GL_INVALID_VALUE is generated if level is less than zero.

GL_INVALID_VALUE may be generated if *level* is greater than log sub 2 max, where max is the returned value of **GL_MAX_TEXTURE_SIZE**.

GL_INVALID_OPERATION is generated if **glGetTexLevelParameter** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glGetTexParameter** subroutine, **glTexImage1D** subroutine, **glTexImage2D** subroutine, **glTexParameter** subroutine.

glGetTexParameter Subroutine

Purpose

Returns texture parameter values.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glGetTexParameterfv(GLenum  Target,
                         GLenum  ParameterName,
                         GLfloat  *ParameterValues)

void glGetTexParameteriv(GLenum  Target,
                         GLenum  ParameterName,
                         GLint  *ParameterValues)
```

Description

The **glGetTexParameter** subroutine returns in *ParameterValues* the value or values of the texture parameter specified as *ParameterName*. *Target* defines the target texture, either **GL_TEXTURE_1D**, **GL_TEXTURE_2D**, **GL_TEXTURE_3D**, and **GL_TEXTURE_3D_EXT** (if the 3D texture extension is supported). *ParameterName* accepts the same symbols as **glTexParameter**, with the same interpretations:

GL_TEXTURE_BASE_LEVEL	Specifies for the texture the base array level. Any non-negative integer value is permissable. Supported in OpenGL 1.2 and later.
GL_TEXTURE_MAX_LEVEL	Specifies for the texture the maximum array level. Any non-negative integer value is permissable. Supported in OpenGL 1.2 and later.
GL_TEXTURE_BORDER_COLOR	Returns four integer or floating-point numbers that comprise the red, green, blue, alpha (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.
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_MAX_LOD	Specifies for the texture the maximum level of detail of the image array. Any floating-point value is permissable. Supported in OpenGL 1.2 and later.
GL_TEXTURE_MIN_LOD	Specifies for the texture the minimum level of detail of the image array. Any floating-point value is permissable. Supported in OpenGL 1.2 and later.
GL_TEXTURE_PRIORITY (1.1 only)	
GL_TEXTURE_PRIORITY_EXT (EXT_texture_object)	Returns the priority of the target texture (or the named texture bound to it). The initial value is 1. See **glPrioritizeTextures**.
GL_TEXTURE_RESIDENT (1.1 only)	
GL_TEXTURE_RESIDENT_EXT (EXT_texture_object)	Returns the residence status of the target texture. If the value returned in params is GL_TRUE, the texture is resident in texture memory. See **glAreTexturesResident**.
GL_TEXTURE_WRAP_R	Returns the single-valued wrapping function for texture coordinate *r*, a symbolic constant.
GL_TEXTURE_WRAP_R (3D Texture Extension)	Returns the single-valued wrapping function for texture coordinate *r*, 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.

Parameters

Target	Specifies the symbolic name of the target texture. **GL_TEXTURE_1D**, **GL_TEXTURE_2D**, **GL_TEXTURE_3D**, and **GL_TEXTURE_3D_EXT** (EXT_texture_3D) are accepted.
ParameterName	Specifies the symbolic name of a texture parameter. **GL_TEXTURE_BASE_LEVEL**, **GL_TEXTURE_MAX_LEVEL**, **GL_TEXTURE_MAG_FILTER**, **GL_TEXTURE_MIN_FILTER**, **GL_TEXTURE_MAX_LOD**, **GL_TEXTURE_MIN_LOD**, **GL_TEXTURE_PRIORITY**, **GL_TEXTURE_PRIORITY_EXT**, **GL_TEXTURE_RESIDENT**, **GL_TEXTURE_RESIDENT_EXT**, **GL_TEXTURE_WRAP_R**, **GL_TEXTURE_WRAP_R_EXT**, **GL_TEXTURE_WRAP_S**, **GL_TEXTURE_WRAP_T**, and **GL_TEXTURE_BORDER_COLOR** are accepted.
ParameterValues	Returns the texture parameters.

Notes

If an error is generated, no change is made to the contents of *ParameterValues*.

Errors

GL_INVALID_ENUM	Either *Target* or *ParameterName* is not an accepted value.
GL_INVALID_OPERATION	The **glGetTexParameter** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glTexParameter** subroutine.

glHint Subroutine

Purpose

Specifies implementation-specific hints.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glHint( GLenum  Target,
   GLenum  Mode)
```

Description

Certain aspects of GL behavior, when there is room for interpretation, can be controlled with hints. A hint is specified with two arguments. *Target* is a symbolic constant indicating the behavior to be controlled, and *Mode* is another symbolic constant indicating the desired behavior. *Mode* can be one of the following three:

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 does not have a preference. This is the initial setting for all hints.

Though the implementation aspects that can be hinted are well-defined, the interpretation of the hints depends on the implementation. The hint aspects that can be specified with *Target*, along with suggested semantics, are:

GL_FOG_HINT	Indicates the accuracy of fog calculation. If per-pixel fog calculation is not efficiently supported by the GL 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 GL implementation, hinting **GL_DONT_CARE** or **GL_FASTEST** can result in simple linear interpolation of colors and 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.
GL_SUBPIXEL_HINT_IBM	Indicates if primitives are rendered using subpixel sampling techniques. Hinting **GL_NICEST** can result in a greater accuracy of pixels turned on when a primitive is rendered. **GL_FASTEST** and **GL_DONT_CARE** may result in faster, non-subpixel positioned, rendering of some primitives.

GL_CLIP_VOLUME_CLIPPING_HINT_EXT	Indicates whether clip volume clipping is desirable. Hinting **GL_NICEST** can result in all clipping calculations being performed, while **GL_FASTEST** can suppress such clipping. **GL_FASTEST** should only be used when the user is confident that no attempts to render will occur outside the clip volume, for the behavior of the GL library is undefined if any primitive extends beyond the clip volume. If extension **IBM_clip_check** is present and **GL_UPDATE_CLIP_VOLUME_HINT** is enabled, this hint can be automatically updated by calls to **glClipBoundingBoxIBM**, **glClipBoundingVolumeIBM**, or **glClipBoundingVerticesIBM**. See these routines for details. This hint is supported only if the **GL_EXT_clip_volume_hint** extension is supported.
GL_PIXEL_FILTER_HINT_IBM	Indicates desired quality of pixel filtering when rendering pixel images specified by glBitmap, glCopyPixel, and glDrawPixel. Hinting **GL_NICEST** should perform pixel filtering that provides the *best* image quality, regardless of performance. **GL_FASTEST** should perform pixel filtering that provides the fastest possible pixel zoom regardless of the image quality. **GL_DONT_CARE** should perform point-sampled blits in accordance with the OpenGL specification.

Parameters

Target	Specifies a symbolic constant indicating the behavior to be controlled. **GL_FOG_HINT**, **GL_LINE_SMOOTH_HINT**, **GL_PERSPECTIVE_CORRECTION_HINT**, **GL_POINT_SMOOTH_HINT**, **GL_POLYGON_SMOOTH_HINT**, **GL_SUBPIXEL_HINT_IBM**, **GL_CLIP_VOLUME_CLIPPING_HINT_EXT** and **GL_PIXEL_FILTER_HINT_IBM** are accepted.
Mode	Specifies a symbolic constant indicating the desired behavior. **GL_FASTEST**, **GL_NICEST**, and **GL_DONT_CARE** are accepted.

Notes

The interpretation of hints depends on the implementation. The **glHint** subroutine can be ignored.

GL_CLIP_VOLUME_CLIPPING_HINT_EXT is only valid if the **GL_EXT_clip_volume_hint** extension is present.

Errors

GL_INVALID_ENUM	Either *Target* or *Mode* is not an accepted value.
GL_INVALID_OPERATION	The **glHint** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.
GL_INVALID_ENUM	The **GL_PIXEL_FILTER_HINT_IBM** parameter is used in an OpenGL implementation that doesn't support the **GL_EXT_pixel_filter_hint** extension.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine.

glIndex Subroutine

Purpose

Sets the current color index.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glIndexd(GLdouble  Current)
void glIndexf(GLfloat Current)
void glIndexi(GLint Current)
void glIndexs(GLshort Current)
void glIndexdv(const GLdouble *Current)
void glIndexfv(const GLfloat *Current)
void glIndexiv(const GLint *Current)
void glIndexsv(const GLshort *Current)
void glIndexub(GLubyte Current)
void glIndexubv(const GLubyte *Current)
```

Description

The **glIndex** subroutine 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.

Parameters

Current	In the case of **glIndexd**, **glIndexf**, **glIndexi**, **glIndexs**, and **glIndexub** this parameter specifies the new value for the current color index.

In the case of **glIndexdv**, **glIndexfv**, **glIndexiv**, **glIndexsv**, and **glIndexubv** this parameter specifies a pointer to a one-element array that contains the new value for the current color index.

Notes

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**.

Associated Gets

Associated gets for the **glIndex** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_CURRENT_INDEX**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** subroutine, **glColor** subroutine, **glEnd** subroutine, **glIndexPointer** subroutine, **glIndexPointerEXT** subroutine.

glIndexMask Subroutine

Purpose

Controls the writing of individual bits in the color index buffers.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glIndexMask(GLuint  Mask)
```

Description

The **glIndexMask** subroutine controls the writing of individual bits in the color index buffers. The least significant *n* bits of the *Mask* parameter, where *n* is the number of bits in a color index buffer, specify a mask. Wherever a 1 (one) appears in the mask, the corresponding bit in the color index buffer (or buffers) is made writable. Where a 0 (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.

Parameters

Mask	Specifies a bit mask to enable and disable the writing of individual bits in the color index buffers. Initially, the mask is all 1's.

Errors

GL_INVALID_OPERATION	The **glIndexMask** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glIndexMask** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_INDEX_WRITEMASK**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glColorMask** subroutine, **glDepthMask** subroutine, **glDrawBuffer** subroutine, **glIndex** subroutine, **glStencilMask** subroutine.

glIndexPointer Subroutine

Purpose

Defines an array of color indexes.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glIndexPointer( GLenum  type,
   GLsizei  stride,
   const GLvoid * pointer)
```

Description

The **glIndexPointer** subroutine specifies the location and data format of an array of color indexes to use when rendering. The *type* parameter specifies the data type of each color index and *stride* gives the byte stride from one color index to the next allowing vertices and attributes to be packed into a single array or stored in separate arrays. (Single array storage may be more efficient on some implementations; see **glInterleavedArrays**.)

The parameters *type*, *stride*, and *pointer* are saved as client-side state.

The color index array is initially disabled. To enable and disable the array, call **glEnableClientState** and **glDisableClientState** with the argument **GL_INDEX_ARRAY**. If enabled, the color index array is used when **glDrawArrays**, **glDrawElements** or **glArrayElement** is called.

Use **glDrawArrays**, **glMultiDrawArraysEXT**, or **glMultiModeDrawArraysIBM** to construct a sequence of primitives from prespecified vertex and vertex attribute arrays. Use **glArrayElement** to specify primitives by indexing vertices and vertex attributes. Use **glDrawElements**, **glMultiDrawElementsEXT**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

If enabled, the Index array is used when **glDrawArrays**, **glDrawElements**, **glArrayElements**, **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** is called.

Parameters

type	Specifies the data type of each color index in the array. Symbolic constants **GL_UNSIGNED_BYTE**, **GL_SHORT**, **GL_INT**, **GL_FLOAT**, or **GL_DOUBLE** are accepted. The initial value is **GL_FLOAT**.

stride	Specifies the byte offset between consecutive color indexes. If *stride* is zero (the initial value), the color indexes are understood to be tightly packed in the array.
pointer	Specifies a pointer to the first index in the array. The initial value is 0 (NULL pointer).

Notes

The **glIndexPointer** subroutine is available only if the GL version is 1.1 or greater.

The color index array is initially disabled, and it won't be accessed when **glArrayElement**, **glDrawElements** or **glDrawArrays** is called.

Execution of **glIndexPointer** is not allowed between **glBegin** and the corresponding **glEnd**, but an error may or may not be generated. If an error is not generated, the operation is undefined.

The **glIndexPointer** subroutine is typically implemented on the client side with no protocol.

Since the color index array parameters are client side state, they are not saved or restored by **glPushAttrib** and **glPopAttrib**. Use **glPushClientAttrib** and **glPopClientAttrib** instead.

The **glIndexPointer** subroutine is not included in display lists.

Errors

GL_INVALID_ENUM is generated if type is not an accepted value.

GL_INVALID_VALUE is generated if stride is negative.

Associated Gets

glIsEnabled with argument **GL_INDEX_ARRAY**

glGet with argument **GL_INDEX_ARRAY_TYPE**

glGet with argument **GL_INDEX_ARRAY_STRIDE**

glGetPointerv with argument **GL_INDEX_ARRAY_POINTER**

Related Information

The **glArrayElement** subroutine, **glColorPointer** subroutine, **glDrawArrays** subroutine, **glDrawElements** subroutine, **glEdgeFlagPointer** subroutine, **glEnable** subroutine, **glGetPointerv** subroutine, **glIndexPointerListIBM** subroutine, **glNormalPointer** subroutine, **glPopClientAttrib** subroutine, **glPushClientAttrib** subroutine, **glTexCoordPointer** subroutine, **glVertexPointer** subroutine.

glIndexPointerEXT Subroutine

Purpose

Defines an array of color indexes.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glIndexPointerEXT(GLenum  type,
     GLsizei  stride,
     GLsizei  count,
     const GLvoid  *pointer)
```

Description

The **glIndexPointerEXT** subroutine specifies the location and data format of an array of color indexes to use when rendering. *type* specifies the data type of each color index and *stride* gives the byte stride from one color index to the next allowing vertexes and attributes to be packed into a single array or stored in separate arrays. (Single-array storage may be more efficient on some implementations.) *count* indicates the number of array elements (counting from the first) that are static. Static elements may be modified by the application, but once they are modified, the application must explicitly respecify the array before using it for any rendering. When a color index array is specified, *type*, *stride*, *count* and *pointer* are saved as client-side state, and static array elements may be cached by the implementation.

The color index array is enabled and disabled using **glEnable** and **glDisable** with the argument **GL_INDEX_ARRAY_EXT**. If enabled, the color index array is used when **glDrawArraysEXT** or **glArrayElementEXT** is called.

Use **glDrawArrays**, **glMultiDrawArraysEXT**, or **glMultiModeDrawArraysIBM** to construct a sequence of primitives from prespecified vertex and vertex attribute arrays. Use **glArrayElement** to specify primitives by indexing vertices and vertex attributes. Use **glDrawElements**, **glMultiDrawElementsEXT**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

If enabled, the Index array is used when **glDrawArrays**, **glDrawElements**, **glArrayElements**, **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** is called.

Parameters

type	Specifies the data type of each color index in the array. Symbolic constants **GL_SHORT**, **GL_INT**, **GL_FLOAT**, or **GL_DOUBLE_EXT**, are accepted.
stride	Specifies the byte offset between consecutive color indexes. If *stride* is zero the color indexes are understood to be tightly packed in the array.
count	Specifies the number of indexes, counting from the first, that are static.
pointer	Specifies a pointer to the first index in the array.

Notes

Non-static array elements are not accessed until **glArrayElementEXT** or **glDrawArraysEXT** is executed.

By default the color index array is disabled and it won't be accessed when **glArrayElementEXT** or **glDrawArraysEXT** is called.

Although, it is not an error to call **glIndexPointerEXT** between the execution of **glBegin** and the corresponding execution of **glEnd**, the results are undefined.

glIndexPointerEXT will typically be implemented on the client side with no protocol.

Since the color index array parameters are client side state, they are not saved or restored by **glPushAttrib** and **glPopAttrib**.

glIndexPointerEXT commands are not entered into display lists.

glIndexPointerEXT is part of the _extname(EXT_vertex_array) extension, not part of the core GL command set. If _extstring(EXT_vertex_array) is included in the string returned by **glGetString**, when called with argument **GL_EXTENSIONS**, extension _extname(EXT_vertex_array) is supported.

Errors

GL_INVALID_ENUM is generated if *type* is not an accepted value.

GL_INVALID_VALUE is generated if *stride* or *count* is negative.

Associated Gets

glIsEnabled with argument **GL_INDEX_ARRAY_EXT**.

glGet with argument **GL_INDEX_ARRAY_SIZE_EXT**.

glGet with argument **GL_INDEX_ARRAY_TYPE_EXT**.

glGet with argument **GL_INDEX_ARRAY_STRIDE_EXT**.

glGet with argument **GL_INDEX_ARRAY_COUNT_EXT**.

glGetPointervEXT with argument **GL_INDEX_ARRAY_POINTER_EXT**.

File

/usr/include/GL/glext.h	Contains extensions to C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glArrayElementEXT** subroutine, **glColorPointerEXT** subroutine, **glDrawArraysEXT** subroutine, **glEdgeFlagPointerEXT** subroutine, **glGetPointervEXT** subroutine, **glNormalPointerEXT** subroutine, **glTexCoordPointerEXT** subroutine, **glVertexPointerEXT** subroutine.

---

glIndexPointerListIBM Subroutine

Purpose

Defines a list of color index arrays.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glIndexPointerListIBM( GLenum  type,
   GLint  stride,
   const GLvoid  ** pointer,
   GLint  ptrstride)
```

Description

The **glIndexPointerListIBM** subroutine specifies the location and data format of a list of arrays of color indices to use when rendering. The *type* parameter specifies the data type of each color index. The *stride* parameter gives the byte stride from one color index to the next allowing vertices and attributes to be packed into a single array or stored in separate arrays. (Single-array storage may be more efficient on some implementations; see **glInterleavedArrays**). The *ptrstride* parameter specifies the byte stride from one pointer to the next in the *pointer* array.

When a color index array is specified, *type*, *stride*, *pointer* and *ptrstride* are saved as client side state.

A *stride* value of 0 does not specify a ″tightly packed″ array as it does in **glIndexPointer**. Instead, it causes the first array element of each array to be used for each vertex. Also, a negative value can be used for stride, which allows the user to move through each array in reverse order.

To enable and disable the color index arrays, call **glEnableClientState** and **glDisableClientState** with the argument **GL_INDEX_ARRAY**. The color index array is initially disabled. When enabled, the color index arrays are used when **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, **glDrawArrays**, **glDrawElements** or **glArrayElement** is called. The last three calls in this list will only use the first array (the one pointed at by *pointer*[0]). See the descriptions of these routines for more information on their use.

Use **glDrawArrays**, **glMultiDrawArraysEXT**, or **glMultiModeDrawArraysIBM** to construct a sequence of primitives from prespecified vertex and vertex attribute arrays. Use **glArrayElement** to specify primitives by indexing vertices and vertex attributes. Use **glDrawElements**, **glMultiDrawElementsEXT**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

If enabled, the Index array is used when **glDrawArrays**, **glDrawElements**, **glArrayElements**, **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** is called.

Parameters

type	Specifies the data type of each color component in the array. Symbolic constants **GL_BYTE**, **GL_UNSIGNED_BYTE**, **GL_SHORT**, **GL_UNSIGNED_SHORT**, **GL_INT**, **GL_UNSIGNED_INT**, **GL_FLOAT**, or **GL_DOUBLE** are accepted. The initial value is GL_FLOAT.
stride	Specifies the byte offset between consecutive color indices. The initial value is 0.
pointer	Specifies a list of color index arrays. The initial value is 0 (NULL pointer).
ptrstride	Specifies the byte stride between successive pointers in the *pointer* array. The initial value is 0.

Notes

The **glIndexPointerListIBM** subroutine is available only if the GL_IBM_vertex_array_lists extension is supported.

Execution of **glIndexPointerListIBM** is not allowed between **glBegin** and the corresponding **glEnd**, but an error may or may not be generated. If an error is not generated, the operation is undefined.

The **glIndexPointerListIBM** subroutine is typically implemented on the client side.

Since the color index array parameters are client side state, they are not saved or restored by **glPushAttrib** and **glPopAttrib**. Use **glPushClientAttrib** and **glPopClientAttrib** instead.

When a **glIndexPointerListIBM** call is encountered while compiling a display list, the information it contains does NOT contribute to the display list, but is used to update the immediate context instead.

The **glIndexPointer** call and the **glIndexPointerListIBM** call share the same state variables. A **glIndexPointer** call will reset the color index list state to indicate that there is only one color index list, so that any and all lists specified by a previous **glIndexPointerListIBM** call will be lost, not just the first list that it specified.

Error Codes

GL_INVALID_ENUM is generated if type is not an accepted value.

Associated Gets

glIsEnabled with argument **GL_INDEX_ARRAY**.

glGetPointerv with argument **GL_INDEX_ARRAY_LIST_IBM**.

glGet with argument **GL_INDEX_ARRAY_LIST_STRIDE_IBM**.

glGet with argument **GL_INDEX_ARRAY_STRIDE**.

glGet with argument **GL_INDEX_ARRAY_TYPE**.

Related Information

The **glArrayElement** subroutine, **glIndexPointer** subroutine, **glDrawArrays** subroutine, **glDrawElements** subroutine, **glEdgeFlagPointer** subroutine, **glEnable** subroutine, **glGetPointerv** subroutine, **glIndexPointer** subroutine, **glInterleavedArrays** subroutine, **glMultiDrawArraysEXT** subroutine, **glMultiDrawElementsEXT** subroutine, **glMultiModeDrawArraysIBM** subroutine, **glMultiModeDrawElementsIBM** subroutine, **glNormalPointer** subroutine, **glPopClientAttrib** subroutine, **glPushClientAttrib** subroutine, **glTexCoordPointer** subroutine, **glVertexPointer** subroutine.

---

glInitNames Subroutine

Purpose

Initializes the name stack.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glInitNames( void )
```

Description

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 **glInitNames** subroutine causes the name stack to be initialized to its default empty state.

The name stack is always empty while the render mode is not **GL_SELECT**. Calls to the **glInitNames** subroutine while the render mode is not **GL_SELECT** are ignored.

Errors

GL_INVALID_OPERATION	The **glInitNames** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glInitNames** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_NAME_STACK_DEPTH**

glGet with argument **GL_MAX_NAME_STACK_DEPTH**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glLoadName** subroutine, **glPushName** subroutine, **glRenderMode** subroutine, **glSelectBuffer** subroutine.

---

glInterleavedArrays Subroutine

Purpose

Simultaneously specifies and enables several interleaved arrays.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glInterleavedArrays(GLenum  format,
     GLsizei  stride,
     const GLvoid *pointer)
```

Description

The **glInterleavedArrays** subroutine lets you specify and enable individual color, normal, texture and vertex arrays whose elements are part of a larger aggregate array element. For some implementations, this is more efficient than specifying the arrays seperately.

If *stride* is zero then the aggregate element are stored consecutively, otherwise *stride* bytes occur between aggregate array elements.

The *format* enumerant serves as a 'key' describing the extraction of 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. Colors may also be extracted as 4 unsigned bytes if 4UB follows the C. If a color is extracted as 4 unsigned bytes, the vertex array element which follows is located at the first possible floating point aligned address.

Parameters

format	Specifies the type of array to enable. Symbolic constants **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** are accepted.
stride	Specifies the offset in bytes between each aggregate array element.

Notes

The **glInterleavedArrays** subroutine is available only if the GL version is 1.1 or greater.

If **glInterleavedArrays** is called while compiling a display list, it is not compiled into the list, and it is executed immediately.

Execution of **glInterleavedArrays** is not allowed between **glBegin** and the corresponding **glEnd**, but an error may or may not be generated. If an error is not generated, the operation is undefined.

The **glInterleavedArrays** subroutine is typically implemented on the client side with no protocol.

Since the vertex array parameters are client side state, they are not saved or restored by **glPushAttrib** and **glPopAttrib**. Use **glPushClientAttrib** and **glPopClientAttrib** instead.

Errors

GL_INVALID_ENUM is generated if format is not an accepted value.

GL_INVALID_VALUE is generated if stride is negative.

Related Information

The **glArrayElement** subroutine, **glColorPointer** subroutine, **glDrawArrays** subroutine, **glDrawElements** subroutine, **glEdgeFlagPointer** subroutine, **glEnableClientState** subroutine, **glGetPointerv** subroutine, **glIndexPointer** subroutine, **glNormalPointer** subroutine, **PopClientAttrib** subroutine, **glPushClientAttrib** subroutine, **glTexCoordPointer** subroutine, **glVertexPointer** subroutine.

glIsEnabled Subroutine

Purpose

Tests whether a capability is enabled.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLboolean glIsEnabled (GLenum  Capability)
```

Description

The **glIsEnabled** subroutine returns **GL_TRUE** if the *Capability* parameter is an enabled capability and returns **GL_FALSE** otherwise. The following capabilities are accepted for *Capability*:

GL_ALPHA_TEST	See **glAlphaFunc**.
GL_AUTO_NORMAL	See **glEvalCoord**.
GL_BLEND	See **glBlendFunc**.
GL_CLIP_PLANE*i*	See **glClipPlane**.
GL_COLOR_ARRAY	See **glColorPointer**
GL_COLOR_ARRAY_EXT	See **glColorPointerEXT**
GL_COLOR_LOGIC_OP	See **glLogicOp**.
GL_COLOR_MATERIAL	See **glColorMaterial**.
GL_COLOR_SUM_EXT	See **glSecondaryColorEXT**.
GL_CULL_FACE	See **glCullFace**.
GL_CULL_VERTEX_IBM	See **GL_CULL_VERTEX_IBM**.
GL_DEPTH_TEST	See **glDepthFunc** and **glDepthRange**.

GL_DITHER	See **glEnable**.
GL_EDGE_FLAG_ARRAY	See **glEdgeFlagPointer**.
GL_EDGE_FLAG_ARRAY_EXT	See **glEdgeFlagPointerEXT**
GL_FOG	See **glFog**.
GL_INDEX_ARRAY	See **glIndexPointer**.
GL_INDEX_ARRAY_EXT	See **glIndexPointerEXT**
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_NORMAL_ARRAY	See **glNormalPointer**.
GL_NORMAL_ARRAY_EXT	See **glNormalPointerEXT**
GL_NORMALIZE	See **glNormal**.
GL_OCCLUSION_CULLING_HP	See **glEnable**.
GL_POINT_SMOOTH	See **glPointSize**.
GL_POLYGON_SMOOTH	See **glPolygonMode**.
GL_POLYGON_STIPPLE	See **glPolygonStipple**.
GL_POLYGON_OFFSET_EXT	See **glPolygonOffsetEXT**
GL_POLYGON_OFFSET_FILL	See **glPolygonOffset**.
GL_POLYGON_OFFSET_LINE	See **glPolygonOffset**.
GL_POLYGON_OFFSET_POINT	See **glPolygonOffset**.
GL_RESCALE_NORMAL_EXT	See **glEnable**.
GL_SCISSOR_TEST	See **glScissor**.
GL_STENCIL_TEST	See **glStencilFunc** and **glStencilOp**.
GL_TEXTURE_1D	See **glTexImage1D**.
GL_TEXTURE_2D	See **glTexImage2D**.
GL_TEXTURE_3D_EXT	See **glTexImage3DEXT**
GL_TEXTURE_COLOR_TABLE_EXT	See **glColorTable**.
GL_TEXTURE_COORD_ARRAY	See **glTexCoordPointer**.
GL_VERTEX_ARRAY	See **glVertexPointer**.
GL_TEXTURE_COORD_ARRAY_EXT	See **glTexCoordPointerEXT**
GL_TEXTURE_GEN_Q	See **glTexGen**.
GL_TEXTURE_GEN_R	See **glTexGen**.
GL_TEXTURE_GEN_S	See **glTexGen**.
GL_TEXTURE_GEN_T	See **glTexGen**.
GL_UPDATE_CLIP_VOLUME_HINT	See **glHint**.

GL_VERTEX_ARRAY_EXT	See **glVertexPointerEXT**.

Parameters

Capability	Specifies a symbolic constant indicating a GL capability.

Notes

If an error is generated, **glIsEnabled** returns 0 (zero).

Errors

GL_INVALID_ENUM	*Capability* is not an accepted value.
GL_INVALID_OPERATION	The **glIsEnabled** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glEnable** subroutine.

glIsList Subroutine

Purpose

Tests for display list existence.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLboolean glIsList(GLuint  List)
```

Description

The **glIsList** subroutine returns **GL_TRUE** if the *List* parameter is the name of a display list and returns **GL_FALSE** otherwise.

Parameters

List	Specifies a potential display-list name.

Errors

GL_INVALID_OPERATION	The **glIsList** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glCallList** subroutine, **glCallLists** subroutine, **glDeleteLists** subroutine, **glGenLists** subroutine, **glNewList** subroutine.

glIsTexture Subroutine

Purpose

Determines if a name corresponds to a texture.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLboolean glIsTexture(GLuint  texture)
```

Description

The **glIsTexture** subroutine returns **GL_TRUE** if *texture* is currently the name of a texture. If *texture* is zero, or is a non-zero value that is not currently the name of a texture, or if an error occurs, **glIsTexture** returns **GL_FALSE**.

The **glIsTexture** subroutine is not included in display lists.

Parameters

texture	Specifies a value which may be the name of a texture.

Notes

The **glIsTexture** subroutine is available only if the GL version is 1.1 or greater.

Errors

GL_INVALID_OPERATION is generated if **glIsTexture** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Related Information

The **glBindTexture** subroutine, **glDeleteTextures** subroutine, **glGenTextures** subroutine, **glGet** subroutine, **glGetTexParameter** subroutine, **glTexImage1D** subroutine, **glTexImage2D** subroutine, **glTexParamete**r subroutine.

glIsTextureEXT Subroutine

Purpose

Determines if a name corresponds to a texture.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

GLboolean **glIsTextureEXT**(GLuint *texture* **)**

Description

glIsTextureEXT returns **GL_TRUE** if *texture* is currently the name of a texture. If *texture* is zero, or is a non-zero value that is not currently the name of a texture, or if an error occurs, **glIsTextureEXT** returns **GL_FALSE**.

glIsTextureEXT is not included in display lists.

Parameters

texture	A value which might be the name of a texture.

Notes

glIsTextureEXT is part of the **EXT_texture_object** extension, not part of the core GL command set. If **GL_EXT_texture_object** is included in the string returned by **glGetString**, when called with argument **GL_EXTENSIONS**, extension **EXT_texture_object** is supported by the connection.

Errors

GL_INVALID_OPERATION is generated if **glIsTextureEXT** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

File

/usr/include/GL/glext.h	Contains extensions to C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBindTextureEXT** subroutine, **glDeleteTexturesEXT** subroutine, **glGenTexturesEXT** subroutine, **glGet** subroutine, **glGetTexParameter** subroutine, **glTexImage1D** subroutine, **glTexImage2D** subroutine, **glTexParameter** subroutine.

glLight Subroutine

Purpose

Sets light source parameters.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glLightf(GLenum  Light,
   GLenum  ParameterName,
   GLfloat  Parameter)
```

```
void glLighti(GLenum  Light,
   GLenum  ParameterName,
   GLint  Parameter)

void glLightfv(GLenum  Light,
   GLenum  ParameterName,
   const GLfloat * ParameterValues)

void glLightiv(GLenum  Light,
   GLenum  ParameterName,
   const GLint * ParameterValues)
```

Description

The **glLight** subroutine sets the values of individual light source parameters. *Light* names the light and is a symbolic name of the form **GL_LIGHT***i*, where 0 is less than or equal to *i* which is less than **GL_MAX_LIGHTS**. *ParameterName* specifies one of 10 light source parameters, again by symbolic name. *ParameterValues* 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*.

The 10 light parameters are as follows:

GL_AMBIENT
ParameterValues contains four integer or floating-point values that specify the ambient red, green, blue, alpha (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
ParameterValues 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
ParameterValues 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
: ParameterValues 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 consider the light's direction, but not its actual position, 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, as well as parallel to and in the direction of the *-z* axis.

GL_SPOT_DIRECTION
: ParameterValues 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
: ParameterValues 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 **GL_SPOT_CUTOFF** description.) The default spot exponent is 0, resulting in uniform light distribution.

GL_SPOT_CUTOFF
: ParameterValues 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, 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**, or **GL_QUADRATIC_ATTENUATION**
: *ParameterValues* 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 times the distance between the light and the vertex being lighted, and the quadratic factor times the square of the same distance. The default attenuation factors are (1,0,0), resulting in no attenuation.

Parameters

Light	Specifies 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 is less than or equal to *i* which is less than **GL_MAX_LIGHTS**.
ParameterName	For **glLightf**, **glLighti**, and **glLightv**, this parameter specifies a single-valued light source parameter for *Light*. **GL_SPOT_EXPONENT**, **GL_SPOT_CUTOFF**, **GL_CONSTANT_ATTENUATION**, **GL_LINEAR_ATTENUATION**, and **GL_QUADRATIC_ATTENUATION** are accepted. For **glLightfv** and **glLightiv**, this parameter specifies a light source parameter for *Light.* **GL_AMBIENT**, **GL_DIFFUSE**, **GL_SPECULAR**, **GL_POSITION**, **GL_SPOT_DIRECTION**, **GL_SPOT_EXPONENT**, **GL_SPOT_CUTOFF**, **GL_CONSTANT_ATTENUATION**, **GL_LINEAR_ATTENUATION**, and **GL_QUADRATIC_ATTENUATION** are accepted.
Parameter	Specifies the value to which the parameter *ParameterName* of light source *Light* is set.
ParameterValues	Specifies a pointer to the value or values to which the parameter *ParameterName* of light source *Light* is set. This parameter is used only with **glLightfv** and **glLightiv**.

Notes

It is always the case that **GL_LIGHT***i* = **GL_LIGHT**0 + *i*.

Errors

GL_INVALID_ENUM	Either *Light* or *ParameterName* is not an accepted value.
GL_INVALID_VALUE	A spot exponent value is specified outside the range [0,128], or spot cutoff is specified outside the range [0,90] (except for the special value 180), or a negative attenuation factor is specified.
GL_INVALID_OPERATION	The **glLight** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glLight** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGetLight

glIsEnabled with argument **GL_LIGHTING**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glColorMaterial** subroutine, **glEnable** or **glDisable** subroutine, **glLightModel** subroutine, **glMaterial** subroutine.

---

glLightModel Subroutine

Purpose

Sets the lighting model parameters.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glLightModelf(GLenum  ParameterName,
                   GLfloat  Parameter)

void glLightModeli(GLenum  ParameterName,
                   GLint  Parameter)

void glLightModelfv(GLenum  ParameterName,
                    const GLfloat * ParameterValues)

void glLightModeliv(GLenum  ParameterName,
                    const GLint * ParameterValues)
```

Description

The **glLightModel** subroutine sets the lighting model parameters. *ParameterName* names a parameter and *ParameterValues* gives the new value. There are three lighting model parameters:

GL_LIGHT_MODEL_COLOR_CONTROL
Lighting produces two colors at the vertex: a primary color and a secondary color. The values of the two colors depend on the light model color control. *ParameterValues* can be GL_SINGLE_COLOR or GL_SPECULAR_COLOR. GL_SINGLE_COLOR is the default value. Depending upon the *ParameterValues*, the lighting equations compute the two colors differently. All computations are carried out in eye coordinates

GL_LIGHT_MODEL_AMBIENT
ParameterValues contains four integer or floating-point values that specify the ambient red, green, blue, alpha (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
ParameterValues is a single integer or floating-point value that specifies how specular reflection angles are computed. If *ParameterValues* is 0 (or 0.0), specular reflections are computed from the origin of the eye coordinate system. Otherwise, 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. The default is False.

GL_LIGHT_MODEL_TWO_SIDE
ParameterValues is a single integer or floating-point value that specifies whether one-sided or two-sided lighting calculations are done for polygons. It has no effect on the lighting calculations for points, lines, or bitmaps. If *ParameterValues* 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 backfacing polygons are lighted using the *back* material parameters, and have their normals reversed before the lighting equation is evaluated. Vertices of frontfacing polygons are always lighted using the *front* material parameters, with no change to their normals. The default is False.

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 0 (zero) if they are 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**. The extent to which the resulting index is above *ambient* is determined by diffuse and specular coefficients, computed with a weighting of the lights' colors (.30, .59, .11); the shininess of the material; and the same reflection and attenuation equations as in the RGBA case.

Parameters

ParameterName	For **glLightModelf** and **glLightModeli**, this parameter specifies a single-valued lighting model parameter. **GL_LIGHT_MODEL_COLOR_CONTROL**, **GL_LIGHT_MODEL_LOCAL_VIEWER**, and **GL_LIGHT_MODEL_TWO_SIDE** are accepted. For **glLightModelfv** and **glLightModeliv**, this parameter specifies a lighting model parameter. **GL_LIGHT_MODEL_AMBIENT**, **GL_LIGHT_MODEL_COLOR_CONTROL**, **GL_LIGHT_MODEL_LOCAL_VIEWER**, and **GL_LIGHT_MODEL_TWO_SIDE** are accepted.
Parameter	Specifies the value to which *ParameterName* is set. This parameter applies only to **GL_LIGHT_MODEL_COLOR_CONTROL**, **glLightModelf**, and **glLightModeli**.
ParameterValues	Specifies a pointer to the value or values to which *ParameterName* is set. This parameter applies only to **glLightModelfv** and **glLightModeliv**.

Errors

GL_INVALID_ENUM	*ParameterName* is not an accepted value.
GL_INVALID_OPERATION	The **glLightModel** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glLightModel** subroutine are as follows. (See the **glGet** subroutine for more information.)

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**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glEnable** or **glDisable** subroutine, **glLight** subroutine, **glMaterial** subroutine.

glLineStipple Subroutine

Purpose

Specifies the line stipple pattern.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glLineStipple(GLint  Factor,
     GLushort  Pattern)
```

Description

Line stippling masks out certain fragments produced by rasterization; those fragments are not drawn. The masking is achieved by using three parameters: the 16-bit line stipple pattern (the *Pattern* parameter), the repeat count (the *Factor* parameter), and an integer stipple counter *s*.

Counter *s* is reset to 0 (zero) whenever the **glBegin** subroutine 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 of the *i* fragments of an *i* width line segment are generated. The *i* fragments associated with count *s* are masked out if

```
Pattern bit floor (s/Factor) mod 16
```

is 0, otherwise these fragments are sent to the frame buffer. Bit 0 of the *Pattern* parameter is the least significant bit.

Antialiased lines are treated as a sequence of *1 times width* rectangles for purposes of stippling. Rectangle *s* is rasterized or not rasterized, based on the fragment rule described for aliased lines, counting rectangles rather than groups of fragments.

Line stippling is enabled or disabled using the **glEnable** and **glDisable** subroutines with the **GL_LINE_STIPPLE** argument. When enabled, the line stipple pattern is applied as described in the preceding section. When disabled, it is as if the pattern were all 1s. Initially, line stippling is disabled.

Parameters

Factor Specifies a multiplier for each bit in the line stipple pattern. If *Factor* is 3, for example, each bit in the pattern is used three times before the next bit in the pattern is used. *Factor* is clamped to the range [1, 255] and defaults to 1.

Pattern Specifies a 16-bit integer whose bit pattern determines which fragments of a line is drawn when the line is rasterized. Bit 0 is used first, and the default pattern is all 1s.

Errors

GL_INVALID_OPERATION The **glLineStipple** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glLineStipple** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_LINE_STIPPLE_PATTERN**

glGet with argument **GL_LINE_STIPPLE_REPEAT**

glIsEnabled with argument **GL_LINE_STIPPLE**.

Files

/usr/include/GL/gl.h Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** subroutine, **glEnable** or **Disable** subroutine, **glLineWidth** subroutine, **glPolygonStipple** subroutine.

glLineWidth Subroutine

Purpose

Specifies the width of rasterized lines.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glLineWidth(GLfloat  Width)
```

Description

The **glLineWidth** subroutine 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 the **glEnable** and **glDisable** subroutines with the **GL_LINE_SMOOTH** argument.

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 (zero), it is as if the line width were 1 (one).) If | DELTA*x* | > | 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. The fragment has a width equal to the current line width, a length equal to the actual length of the line, and is 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 the **glGet** subroutine with the **GL_LINE_WIDTH_RANGE** and **GL_LINE_WIDTH_GRANULARITY** arguments.

Parameters

Width	Specifies the width of rasterized lines. The default is 1.0.

Notes

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.

Errors

GL_INVALID_VALUE	*Width* is less than or equal to 0.
GL_INVALID_OPERATION	The **glLineWidth** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glLineWidth** subroutine are as follows. (See the **glGet** subroutine for more information.)

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**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glEnable** or **Disable** subroutine.

---

glListBase Subroutine

Purpose

Sets the display-list base for the **glCallLists** subroutine.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glListBase(GLuint  Base)
```

Description

The **glCallLists** subroutine specifies an array of offsets. Display-list names are generated by adding the *Base* parameter to each offset. Names that reference valid display lists are executed; the others are ignored.

Parameters

Base	Specifies an integer offset that is added to **glCallLists** offsets to generate display-list names. Initial value is 0 (zero).

Errors

GL_INVALID_OPERATION	The **glListBase** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glListBase** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_LIST_BASE**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glCallLists** subroutine.

---

glLoadIdentity Subroutine

Purpose

Replaces the current matrix with the identity matrix.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glLoadIdentity( void )
```

Description

The **glLoadIdentity** subroutine replaces the current matrix with the identity matrix. It is semantically equivalent to calling the **glLoadMatrix** subroutine with the following identity matrix:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 6. Identity Matrix. This diagram shows a matrix enclosed in brackets. The matrix consists of four lines containing four characters each. The first line contains the following (from left to right): one, zero, zero, zero. The second line contains the following (from left to right): zero, one, zero, zero. The third line contains the following (from left to right): zero, zero, one, zero. The fourth line contains the following (from left to right): zero, zero, zero, one.

Calling **glLoadIdentity** is in some cases more efficient.

Errors

GL_INVALID_OPERATION	The **glLoadIdentity** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glLoadIdentity** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_MATRIX_MODE**

glGet with argument **GL_MODELVIEW_MATRIX**

glGet with argument **GL_PROJECTION_MATRIX**

glGet with argument **GL_TEXTURE_MATRIX**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glLoadMatrix** subroutine, **glMatrixMode** subroutine, **glMultMatrix** subroutine, **glPushMatrix** subroutine.

glLoadMatrix Subroutine

Purpose

Replaces the current matrix with an arbitrary matrix.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glLoadMatrixd(const GLdouble  *Matrix)

void glLoadMatrixf(const GLfloat  *Matrix)
```

Description

The **glLoadMatrix** subroutine replaces the current matrix with the one specified in the *Matrix* parameter. The current matrix is the projection matrix, model view matrix, or texture matrix, determined by the current matrix mode. (See the **glMatrixMode** subroutine for information on specifiying the current matrix.) The *Matrix* parameter points to a 4 x 4 matrix of single- or double-precision floating-point values stored in column-major order. That is, the matrix is stored as the following:

$$\begin{pmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{pmatrix}$$

Figure 7. Stored Matrix. This diagram shows a matrix enclosed in brackets. The matrix consists of four lines containing four characters each. The first line contains the following (from left to right): a subscript zero, a subscript four, a subscript eight, a subscript twelve. The second line contains the following (from left to right): a subscript one, a subscript five, a subscript nine, a subscript thirteen. The third line contains the following (from left to right): a subscript two, a subscript six, a subscript ten, a subscript fourteen. The fourth line contains the following (from left to right): a subscript three, a subscript seven, a subscript eleven, a subscript fifteen.

Parameters

Matrix	Specifies a pointer to 4 x 4 matrix stored in column-major order as 16 consecutive values.

Errors

GL_INVALID_OPERATION	The **glLoadMatrix** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glLoadMatrix** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_MATRIX_MODE**

glGet with argument **GL_MODELVIEW_MATRIX**

glGet with argument **GL_PROJECTION_MATRIX**

glGet with argument **GL_TEXTURE_MATRIX**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glLoadIdentity** subroutine, **glMatrixMode** subroutine, **glMultMatrix** subroutine, **glPushMatrix** subroutine, **glLoadTransposeMatrixARB** subroutine.

glLoadName Subroutine

Purpose

Loads a name onto the name stack.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void LoadName(GLuint  Name)
```

Description

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 **glLoadName** subroutine causes the *Name* parameter to replace the value on the top of the name stack, which is initially empty.

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.

Parameters

Name	Specifies a name that replaces the top value on the name stack.

Errors

GL_INVALID_OPERATION	The **glLoadName** subroutine is called while the name stack is empty.
GL_INVALID_OPERATION	The **glLoadName** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glLoadName** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_NAME_STACK_DEPTH**.

glGet with argument **GL_MAX_NAME_STACK_DEPTH**.

Files

/usr/include/GL/gl.h Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glInitNames** subroutine, **glPushName** subroutine, **glRenderMode** subroutine, **glSelectBuffer** subroutine.

glLoadNamedMatrixIBM Subroutine

Purpose

Loads a pre-defined matrix into the top of the named matrix stack.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
void glLoadNamedMatrixIBM(GLenum  matrix,
                          GLenum  name)
```

Description

Using this subroutine, a predefined matrix can be loaded into any matrix stack, regardless of the current matrix mode in use.

glLoadNamedMaxtrixIBM(*matrix*, **GL_IDENTITY_MATRIX_IBM**) is functionally equivalent to:

```
PushAttrib(GL_TRANSFORM_BIT);
MatrixMode(matrix);
LoadIdentity();
PopAttrib();
```

This subroutine does NOT change the current matrix mode.

Parameters

matrix specifies which of the matrices to load. Acceptable values are **GL_COLOR**, **GL_TEXTURE**, **GL_MODELVIEW**, and **GL_PROJECTION**.

name specifies the named matrix to load. Acceptable values and their corresponding matrices are:

```
GL_IDENTITY_MATRIX_IBM  1.0 0.0 0.0 0.0
    0.0 1.0 0.0 0.0
    0.0 0.0 1.0 0.0
    0.0 0.0 0.0 1.0

GL_YCRCB_TO_RGB_MATRIX_IBM 1.164 0.000 1.596  -0.874
    1.164  -0.392  -0.813   0.532
    1.164 2.017 0.000 1.000
    0.000 0.000 0.000 1.000

GL_RGB_TO_YCRCB_MATRIX_IBM 0.257 0.504 0.098 0.063
          -0.148  -0.291   0.439 0.502
    0.439  -0.368  -0.071 0.502
        0.000 0.000 0.000 1.000
```

Note that the second and third parameters above are only valid if the **GL_IBM_YCbCr** extension is present.

Notes

This subroutine is only available if the **GL_IBM_load_matrix** extension is present.

Error Codes

GL_INVALID_ENUM is generated if *matrix* is not one of the acceptable values.
GL_INVALID_ENUM is generated if *name* is not one of the acceptable values.

Files

/usr/include/GL/gl.h Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

glLoadTransposeMatrixARB Subroutine

Purpose

Loads a matrix in row-major order, rather than column-major order.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

void glLoadTransposeMatrixfARB(const GLfloat **Matrix***)**
void glLoadTransposeMatrixdARB(const GLdouble **Matrix***)**

Description

The **glLoadTransposeMatrixARB** subroutine replaces the current matrix with the one specified in the *Matrix* parameter. The current matrix is the projection matrix, model view matrix, or texture matrix, determined by the current matrix mode. (See the **glMatrixMode** subroutine for information on specifiying the current matrix.) The *Matrix* parameter points to a 4 x 4 matrix of single- or double-precision floating-point values stored in row-major order. That is, the matrix is stored as the following:

```
/ a0  a1  a2  a3  \
| a4  a5  a6  a7  |
| a8  a9  a10 a11 |
\ a12 a13 a14 a15 /
```

Parameters

Matrix is an array of 16 values, specified in row-major order.

Error Codes

GL_INVALID_OPERATION is generated if **glLoadTransposeMatrixARB** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Files

/usr/include/GL/gl.h Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glLoadMatrix** subroutine, the **glMatrixMode** subroutine.

glLockArraysEXT Subroutine

Purpose

Locks the currently enabled vertex arrays.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glLockArraysEXT (int  first,
     sizei  count)
```

Description

The currently enabled vertex arrays can be locked with the subroutine **glLockArraysEXT**. When the vertex arrays are locked, the GL can compile the array data or the transformed results of array data associated with the currently enabled vertex arrays. The vertex arrays are unlocked by the **glUnlockArraysEXT** subroutine.

Between **glLockArraysEXT** and **glUnlockArraysEXT** the application should ensure that none of the array data in the range of elements specified by *first* and *count* are changed. Changes to the array data between the execution of glLockArraysEXT and **glUnlockArraysEXT** subroutines may affect calls to DrawArrays, ArrayElement, or DrawElements subroutines in non-sequential ways.

While using a compiled vertex array, references to array elements by the subroutines DrawArrays, ArrayElement, or DrawElements which are outside of the range specified by *first* and *count* are undefined.

This extension defines an interface which allows static vertex array data to be cached or pre-compiled for more efficient rendering. This is useful for implementations which can cache the transformed results of array data for reuse by several DrawArrays, ArrayElement, or DrawElements subroutines. It is also useful for implementations which can transfer array data to fast memory for more efficient processing.

For example, rendering an M by N mesh of quadrilaterals can be accomplished by setting up vertex arrays containing all of the vertexes in the mesh and issuing M DrawElements subroutines each of which operate on 2 * N vertexes. Each DrawElements subroutine after the first will share N vertexes with the preceding DrawElements subroutine. If the vertex array data is locked while the DrawElements subroutines are executed, then OpenGL may be able to transform each of these shared vertexes just once.

Parameters

first	The first element in the locked range.
count	The number of elements to be contained in the locked range.

Errors

INVALID_VALUE	*First* is less than or equal to zero.
INVALID_OPERATION	The **glLockArraysEXT** subroutine is called between execution of **glLockArraysEXT** and the corresponding execution of **glUnlockArraysEXT**.
INVALID_OPERATION	The **glLockArraysEXT** subroutine is called between execution of **Begin** and the corresponding execution of **End**.

Related Information

The **glUnlockArraysEXT** subroutine.

---

glLogicOp Subroutine

Purpose

Specifies a logical pixel operation for color index rendering.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void LogicOp(GLenum  OperatorCode)
```

Description

The **glLogicOp** subroutine specifies a logical operation that, when enabled, is applied between the incoming color and the color at the corresponding location in the frame buffer. The logical operation is enabled or disabled with the **glEnable** and **glDisable** subroutines using the **GL_LOGIC_OP** symbolic constant for color index mode or the **GL_COLOR_LOGIC_OP** for RGB mode.

The *OperatorCode* parameter specifies a symbolic constant chosen from the following list. 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 bit-wise operators suggest, the logical operation is applied independently to each bit pair of the source and destination indexes.

Operation	Resulting Value
GL_CLEAR	0
GL_SET	1
GL_COPY	s

Operation	Resulting Value
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

Parameters

OperatorCode Specifies a symbolic constant that selects a logical operation. The following symbols are accepted:

- **GL_CLEAR**
- **GL_SET**
- **GL_COPY**
- **GL_COPY_INVERTED**
- **GL_NOOP**
- **GL_INVERT**
- **GL_AND**
- **GL_NAND**
- **GL_OR**
- **GL_NOR**
- **GL_XOR**
- **GL_EQUIV**
- **GL_AND_REVERSE**
- **GL_AND_INVERTED**
- **GL_OR_REVERSE**
- **GL_OR_INVERTED**

Notes

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 the **glDrawBuffer** subroutine for information about specifying color buffers for drawing.)

The *OperatorCode* parameter must be one of the 16 accepted values. Other values result in an error.

Errors

GL_INVALID_ENUM *OperatorCode* is not an accepted value.

GL_INVALID_OPERATION The **glLogicOp** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glLogicOp** subroutine are as follows. (See the **glGet** subroutine for more information.)

glEnable or **glDisable** with argument **GL_COLOR_LOGIC_OP** for RGB mode or **GL_INDEX_LOGIC_OP** for color index mode.

glGet with argument **GL_LOGIC_OP_MODE**.

glIsEnabled with argument **GL_LOGIC_OP**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glAlphaFunc** subroutine, **glBegin** or **glEnd** subroutine, **glBlendEquationEXT** subroutine, **glBlendFunc** subroutine, **glDrawBuffer** subroutine, **glEnable** or **Disable** subroutine, **glStencilOp** subroutine.

glMap1 Subroutine

Purpose

Defines a 1-dimensional (1D) evaluator.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
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)
```

Description

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 GL processing just as if they had been presented using the **glVertex**, **glNormal**, **glTexCoord**, and **glColor** subroutines, 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 GL implementation) can be described using evaluators. These include almost all splines used in computer graphics, such as B-splines, Bezier curves, and Hermite splines.

Evaluators define curves based on Bernstein polynomials. Define **p***(t)* as the following:

```
Let p(t) = Bn0(t)R0 + Bn1(t)R1 + . . . + Bnn(t)Rn
```

where **R**i is a control point and *Bn*i*(t)* is the *i*th Bernstein polynomial of degree:

```
n (Order = n+1)
```

See the figure:

$$\binom{n}{k} \text{ is the binomial coefficient given by } \binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

Figure 8. Binomial Coefficient Equation. This figure shows that the binomial coefficient with original set of size n and subset of size k is the binomial coefficient given by the following equation: the binomial coefficient with original set of size n and subset of size k is equal to n! / k! (n-k)!.

See the figure:

$$0^0 \equiv 1 \text{ and } \binom{n}{k} \equiv 1$$

Figure 9. Definition. This figure shows that zero to the power of zero is eqivalent to one and the binomial coefficient with original set of size n and subset of size k is also eqivalent to one.

The **glMap1** subroutine 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 the **glEnable** and **glDisable** subroutines with the map name, one of the nine predefined values for the *Target* parameter. The **glEvalCoord1** subroutine evaluates the 1D maps that are enabled. When **glEvalCoord1** presents a value *u*, the Bernstein functions are evaluated using *t*, as in the following figure:

$$t = \frac{u - u1}{u2 - u1}$$

Figure 10. Value of t. This figure shows that t is equal to u-u1 / u2–u1.

The *Target* parameter specifies a symbolic constant that indicates what kind of control points are provided in the *Points* parameter, and what output is generated when the map is evaluated. It can assume one of the following nine predefined values:

GL_MAP1_VERTEX_3	Each control point is three floating-point values representing *x*, *y*, and *z*. Internal **glVertex3** subroutines 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** subroutines 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** subroutines are generated when the map is evaluated. However, the current index is not updated with the value of these **glIndex** subroutines.
GL_MAP1_COLOR_4	Each control point is four floating-point values representing red, green, blue, and alpha (RGBA). Internal **glColor4** subroutines are generated when the map is evaluated. However, the current color is not updated with the value of these **glColor4** subroutines.
GL_MAP1_NORMAL	Each control point is three floating-point values representing the *x*, *y*, and *z* components of a normal vector. Internal **glNormal** subroutines are generated when the map is evaluated. However, the current normal is not updated with the value of these **glNormal** subroutines.
GL_MAP1_TEXTURE_COORD_1	Each control point is a single floating-point value representing the *s* texture coordinate. Internal **glTexCoord1** subroutines are generated when the map is evaluated. However, the current texture coordinates are not updated with the value of these **glTexCoord** subroutines.
GL_MAP1_TEXTURE_COORD_2	Each control point is two floating-point values representing the *s* and *t* texture coordinates. Internal **glTexCoord2** subroutines are generated when the map is evaluated. However, the current texture coordinates are not updated with the value of these **glTexCoord** subroutines.
GL_MAP1_TEXTURE_COORD_3	Each control point is three floating-point values representing the *s*, *t*, and *r* texture coordinates. Internal **glTexCoord3** subroutines are generated when the map is evaluated. However, the current texture coordinates are not updated with the value of these **glTexCoord** subroutines.
GL_MAP1_TEXTURE_COORD_4	Each control point is four floating-point values representing the *s*, *t*, *r* and *q* texture coordinates. Internal **glTexCoord4** subroutines are generated when the map is evaluated. However, the current texture coordinates are not updated with the value of these **glTexCoord** subroutines.

The *Stride*, *Order*, and *Points* parameters define the array addressing for accessing the control points. *Points* 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. *Order* is the number of control points in the array. *Stride* tells how many float or double locations to advance the internal memory pointer to reach the next control point.

Parameters

Target Specifies the values that are generated by the evaluator. The following symbolic constants are accepted:

- **GL_MAP1_VERTEX_3**
- **GL_MAP1_VERTEX_4**
- **GL_MAP1_INDEX**
- **GL_MAP1_COLOR_4**
- **GL_MAP1_NORMAL**
- **GL_MAP1_TEXTURE_COORD_1**
- **GL_MAP1_TEXTURE_COORD_2**
- **GL_MAP1_TEXTURE_COORD_3**
- **GL_MAP1_TEXTURE_COORD_4**

u1, *u2*	Specify a linear mapping of *u*, as presented to **glEvalCoord1**, to *u*1, the variable that is evaluated by the equations specified by this subroutine.
Stride	Specifies 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	Specifies the number of control points. Must be positive.
Points	Specifies a pointer to the array of control points.

Notes

As is the case with all GL subroutines 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.

Errors

GL_INVALID_ENUM	*Target* is not an accepted value.
GL_INVALID_VALUE	*u1* is equal to *u2*.
GL_INVALID_VALUE	*Stride* is less than the number of values in a control point.
GL_INVALID_VALUE	*Order* is less than one or greater than **GL_MAX_EVAL_ORDER**.
GL_INVALID_OPERATION	The **glMap1** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glMap1** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGetMap.

glGet with argument **GL_MAX_EVAL_ORDER**.

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**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glColor** subroutine, **glEnable** or **glDisable** subroutine, **glEvalCoord** subroutine, **glEvalMesh** subroutine, **glEvalPoint** subroutine, **glGetMap** subroutine, **glIndex** subroutine, **glMap2** subroutine, **glMapGrid** subroutine, **glNormal** subroutine, **glTexCoord** subroutine, **glVertex** subroutine.

---

glMap2 Subroutine

Purpose

Defines a 2-dimensional (2D) evaluator.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
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)
```

Description

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 GL processing just as if they had been presented using the **glVertex**, **glNormal**, **glTexCoord**, and **glColor** subroutines, 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 GL implementation) can be described using evaluators. These include almost all surfaces used in computer graphics, such as B-spline surfaces, non-uniform rational B-spline surfaces (NURBS), and Bezier surfaces.

Evaluators define surfaces based on bivariate Bernstein polynomials. Define **p***(s, t)* as follows:

```
Let p(s, t) = Bn0Bm0R00 + Bn1Bm0R01 + . . . + BnnBm0Rn0
   +Bn0Bm1R01 + . . . + BnnBm1Rn1
    .
    .
    .
   +Bn0BmmR0m + . . . + BnnBmmRnm
```

where **R**ij is a control point, *Bn*i *(s)* is the *i*th Bernstein polynomial of degree:

n (*uOrder* = *n* +1)

See the following figure:

$$Bn_i\,(s) = \begin{pmatrix} n \\ i \end{pmatrix} s^i\,(1 - s)^{n-i}$$

Figure 11. Value of Bni (s). This figure shows that Bni (s) is equal to [the binomial coefficient with original set of size n and subset of size i] s to the power of i (1–s) to the power of n-i.

and *Bm*j (*t*) is the *j*th Bernstein polynomial of degree:

m (*vOrder* = *m* + 1)

See the following figure:

$$Bm_j\;(t) = \begin{pmatrix} m \\ j \end{pmatrix} t^j\,(1 - t)^{m-j}$$

Figure 12. Value of Bmj (t). This figure shows that Bmj (t) is equal to [the binomial coefficient with original set of size m and subset of size j] t to the power of j (1–t) to the power of m-j.

See the following figure:

$$0^0 \equiv 1 \text{ and } \begin{pmatrix} n \\ 0 \end{pmatrix} \equiv 1$$

Figure 13. Definition. This figure shows that zero to the power of zero is eqivalent to one and the binomial coefficient with original set of size n and subset of size zero is also eqivalent to one.

The **glMap2** subroutine 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 the **glEnable** and **glDisable** subroutines with the map name, which is one of the nine predefined values for the *Target* parameter. When the **glEvalCoord2** subroutine presents values *u* and *v*, the bivariate Bernstein polynomials are evaluated using *s* and *t*, as in the following figure:

$$s = \frac{u - u1}{u2 - u1}$$

$$t = \frac{v - v1}{v2 - v1}$$

Figure 14. Value of s and t. This figure shows two equations. The first equation shows that s is equal to u–u1 / u2–u1. The second equation shows that t is equal to v–v1 / v2–v1.

The *Target* parameter specifies a symbolic constant that indicates what kind of control points are provided in the *Points* parameter, and what output is generated when the map is evaluated. It can assume one of the following nine predefined values:

GL_MAP2_VERTEX_3
: Each control point is three floating-point values representing *x*, *y*, and *z*. Internal **glVertex3** subroutines 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** subroutines 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** subroutines are generated when the map is evaluated. However, the current index is not updated with the value of these **glIndex** subroutines.

GL_MAP2_COLOR_4
: Each control point is four floating-point values representing red, green, blue, and alpha. Internal **glColor4** subroutines are generated when the map is evaluated. However, the current color is not updated with the value of these **glColor4** subroutines.

GL_MAP2_NORMAL
: Each control point is three floating-point values representing the *x*, *y*, and *z* components of a normal vector. Internal **glNormal** subroutines are generated when the map is evaluated. However, the current normal is not updated with the value of these **glNormal** subroutines.

GL_MAP2_TEXTURE_COORD_1
: Each control point is a single floating-point value representing the *s* texture coordinate. Internal **glTexCoord1** subroutines are generated when the map is evaluated. However, the current texture coordinates are not updated with the value of these **glTexCoord** subroutines.

GL_MAP2_TEXTURE_COORD_2
: Each control point is two floating-point values representing the *s* and *t* texture coordinates. Internal **glTexCoord2** subroutines are generated when the map is evaluated. However, the current texture coordinates are not updated with the value of these **glTexCoord** subroutines.

GL_MAP2_TEXTURE_COORD_3
: Each control point is three floating-point values representing the *s*, *t*, and *r* texture coordinates. Internal **glTexCoord3** subroutines are generated when the map is evaluated. However, the current texture coordinates are not updated with the value of these **glTexCoord** subroutines.

GL_MAP2_TEXTURE_COORD_4
: Each control point is four floating-point values representing the *s*, *t*, *r*, and *q* texture coordinates. Internal **glTexCoord4** subroutines are generated when the map is evaluated. However, the current texture coordinates are not updated with the value of these **glTexCoord** subroutines.

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 times 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).

Parameters

Target	Specifies the kind of values that are generated by the evaluator. The following symbolic constants are accepted: • **GL_MAP2_VERTEX_3** • **GL_MAP2_VERTEX_4** • **GL_MAP2_INDEX** • **GL_MAP2_COLOR_4** • **GL_MAP2_NORMAL** • **GL_MAP2_TEXTURE_COORD_1** • **GL_MAP2_TEXTURE_COORD_2** • **GL_MAP2_TEXTURE_COORD_3** • **GL_MAP2_TEXTURE_COORD_4**
u1, *u2*	Specify a linear mapping of *u*, as presented to **glEvalCoord2**, to *u*1, one of the two variables that is evaluated by the equations specified by this subroutine.
uStride	Specifies the number of floats or doubles between the beginning of control point **R**ij and the beginning of control point **R**(i+1)j, where *i* and *j* are the *u* and *y* 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	Specifies the dimension of the control point array in the *u* axis. Must be positive.
v1, *v2*	Specify a linear mapping of *v*, as presented to **glEvalCoord2**, to *v*1, one of the two variables that is evaluated by the equations specified by this subroutine.
vStride	Specifies the number of floats or doubles between the beginning of control point **R**ij and the beginning of control point **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	Specifies the dimension of the control point array in the *v* axis. Must be positive.
Points	Specifies a pointer to the array of control points.

Notes

For all GL subroutines 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.

Errors

GL_INVALID_ENUM	*Target* is not an accepted value.
GL_INVALID_VALUE	*u1* is equal to *u2*, or if *v1* is equal to *v2*.
GL_INVALID_VALUE	*uStride* or *vStride* is less than the number of values in a control point.
GL_INVALID_VALUE	*uOrder* or *vOrder* is less than one or greater than **GL_MAX_EVAL_ORDER**.
GL_INVALID_OPERATION	The **glMap2** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glMap2** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGetMap

glGet with argument **GL_MAX_EVAL_ORDER**

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**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glColor** subroutine, **glEnable** or **Disable** subroutine, **glEvalCoord** subroutine, **glEvalMesh** subroutine, **glEvalPoint** subroutine, **glGetMap** subroutine, **glIndex** subroutine, **glMap1** subroutine, **glMapGrid** subroutine, **glNormal** subroutine, **glTexCoord** subroutine, **glVertex** subroutine.

glMapGrid Subroutine

Purpose

Defines a 1-dimensional (1D) or 2-dimensional (2D) mesh.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
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)
```

Description

The **glMapGrid** and **glEvalMesh** subroutines are used in tandem to efficiently generate and evaluate a series of evenly spaced map domain values. The **glEvalMesh** subroutine steps through the integer domain of a 1D or 2D grid, whose range is the domain of the evaluation maps specified by the **glMap1** and **glMap2** subroutines.

The **glMapGrid1** and **glMapGrid2** subroutines 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 the **glMap1** subroutine and the **glMap2** subroutine for details of how *u* and *v* coordinates are evaluated.

The **glMapGrid1** subroutine specifies a single linear mapping such that integer grid coordinate 0 (zero) maps exactly to *u1*, and integer grid coordinate *un* maps exactly to *u2*. All other integer grid coordinates *i* are mapped such that

$u = i(u2 - u1)/un + u1$

The **glMapGrid2** subroutine 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

$u = i(u2 - u1)/un + u1$

$v = j(v2 - v1)/vn + v1$

The mappings specified by **glMapGrid** are identically used by **glEvalMesh** and **glEvalPoint**.

Parameters

un	Specifies the number of partitions in the grid range interval [*u1*, *u2*]. Must be positive.
u1, *u2*	Specify the mappings for integer grid domain values *i=0* and *i=un*.
vn	Specifies the number of partitions in the grid range interval [*v1*, *v2*] (**glMapGrid2** only).
v1, *v2*	Specify the mappings for integer grid domain values *j*=0 and *j=vn* (**glMapGrid2** only).

Errors

GL_INVALID_VALUE	*un* or *vn* is not positive.
GL_INVALID_OPERATION	The **glMapGrid** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glMapGrid** subroutine are as follows. (See the **glGet** subroutine for more information.)

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**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glEvalCoord** subroutine, **glEvalMesh** subroutine, **glEvalPoint** subroutine, **glMap1** subroutine, **glMap2** subroutine.

glMaterial Subroutine

Purpose

Specifies material parameters for the lighting model.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glMaterialf(GLenum  Face,
      GLenum  pName,
      GLfloat  Parameter)

void glMateriali(GLenum  Face,
      GLenum  pName,
      GLint  Parameter)

void glMaterialfv(GLenum  Face,
        GLenum  pName,
        const GLfloat * Parameters)

void glMaterialiv(GLenum  Face,
        GLenum  pName,
        const GLint * Parameters)
```

Description

The **glMaterial** subroutine assigns values to material parameters. There are two matched sets of material parameters. One, the *frontfacing* set, is used to shade points, lines, bitmaps, and all polygons (when two-sided lighting is disabled), or just frontfacing polygons (when two-sided lighting is enabled). The other set, *backfacing*, is used to shade backfacing polygons only when two-sided lighting is enabled. See the **glLightModel** subroutine for details concerning one- and two-sided lighting calculations.

The **glMaterial** subroutine takes three arguments:

- The *Face* parameter specifies whether the **GL_FRONT** materials, the **GL_BACK** materials, or both **GL_FRONT_AND_BACK** materials are modified.
- The *pName* parameter specifies which of several parameters in one or both sets are modified.
- The *Parameters* parameter specifies what value or values are assigned to the specified parameter.

Material parameters are used in the lighting equation that is optionally applied to each vertex. See the **glLightModel** subroutine for details about the lighting equation. The following parameters and their interpretations by the lighting equation can be specified using **glMaterial**:

GL_AMBIENT	*Parameters* contains four integer or floating-point values that specify the ambient red, green, blue, alpha (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 backfacing materials is (0.2, 0.2, 0.2, 1.0).
GL_DIFFUSE	*Parameters* 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 backfacing materials is (0.8, 0.8, 0.8, 1.0).
GL_SPECULAR	*Parameters* 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 backfacing materials is (0.0, 0.0, 0.0, 1.0).
GL_EMISSION	*Parameters* 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 backfacing materials is (0.0, 0.0, 0.0, 1.0).
GL_SHININESS	*Parameters* 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 frontfacing and backfacing 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	*Parameters* contains three integer or floating-point values specifying the color indices 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. See the **glLightModel** subroutine for a discussion of color index lighting.

Parameters

materialf and materiali

Face	Specifies which face or faces are being updated. The *Face* parameter must be one of **GL_FRONT**, **GL_BACK** or **GL_FRONT_AND_BACK**.
pName	Specifies the single-valued material parameter of the face or faces that is being updated. Must be **GL_SHININESS**.
Parameter	Specifies the value to which **GL_SHININESS** is set.

materialfv and materialiv

Face	Specifies which face or faces are being updated. Must be one of **GL_FRONT**, **GL_BACK**, or **GL_FRONT_AND_BACK**.

pName	Specifies the material parameter of the face or faces that is being updated. Must be one of the following: • **GL_AMBIENT** • **GL_DIFFUSE** • **GL_SPECULAR** • **GL_EMISSION** • **GL_SHININESS** • **GL_AMBIENT_AND_DIFFUSE** • **GL_COLOR_INDEXES**
Parameters	Specifies a pointer to the value or values to which the *pName* parameter is set.

Notes

The material parameters can be updated at any time. In particular, **glMaterial** can be called between a call to the **glBegin** subroutine and the corresponding call to the **glEnd** subroutine. If only a single material parameter is to be changed per vertex, however, **glColorMaterial** is preferred over **glMaterial**. (See the **glColorMaterial** subroutine for information on tracking the current color with the material color.)

Errors

GL_INVALID_ENUM	*Face* or *pName* is not an accepted value.
GL_INVALID_VALUE	A specular exponent outside the range [0,128] is specified.

Associated Gets

Associated get for the **glMaterial** subroutine is as follows. (See the **glGet** subroutine for more information.)

glGetMaterial.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glColorMaterial** subroutine, **glGetMaterial** subroutine, **glLight** subroutine, **glLightModel** subroutine.

glMatrixMode Subroutine

Purpose

Specifies the current matrix.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glMatrixMode(GLenum  Mode)
```

Description

The **glMatrixMode** subroutine sets the current matrix mode. The *Mode* parameter can assume one of the following three values:

GL_MODELVIEW	Applies subsequent matrix operations to the model view matrix stack.
GL_PROJECTION	Applies subsequent matrix operations to the projection matrix stack.
GL_TEXTURE	Applies subsequent matrix operations to the texture matrix stack.

Parameters

Mode Specifies which matrix stack is the target for subsequent matrix operations. The following three values are accepted:

- **GL_MODELVIEW**
- **GL_PROJECTION**
- **GL_TEXTURE**

Associated Gets

Associated gets for the **glMatrixMode** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_MATRIX_MODE**.

Errors

GL_INVALID_ENUM	*Mode* is not an accepted value.
GL_INVALID_OPERATION	The **glMatrixMode** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glLoadMatrix** subroutine, **glPushMatrix** subroutine.

glMultiDrawArraysEXT Subroutine

Purpose

Renders multiple primitives from array data.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glMultiDrawArraysEXT(GLenum  mode,
                          GLint  *first,
                          GLsizei  *count,
                          GLsizei  primcount)
```

Description

The **glMultiDrawArraysEXT** subroutine lets you specify multiple geometric primitives with very few subroutine calls. Instead of calling a GL procedure to pass each individual vertex, normal, texture coordinate, edge flag, or color, you can prespecify separate arrays of vertexes, normals, and colors and use them to construct a sequence of primitives with a single call to **glMultiDrawArraysEXT**.

When **glMultiDrawArraysEXT** is called, it uses count sequential elements from each enabled array to construct a sequence of geometric primitives, beginning with element first. The *mode* parameter specifies what kind of primitives are constructed, and how the array elements construct these primitives. If **GL_VERTEX_ARRAY** is not enabled, no geometric primitives are generated.

Vertex attributes that are modified by **glMultiDrawArraysEXT** have an unspecified value after **glMultiDrawArraysEXT** returns. For example, if **GL_COLOR_ARRAY** is enabled, the value of the current color is undefined after **glMultiDrawArraysEXT** executes. Attributes that are not modified remain well defined.

Behaves identically to DrawArrays except that a list of arrays is specified instead. The number of lists is specified in the primcount parameter. It has the same effect as:

```
for(i=0; i<primcount; i++) {
   if (*(count+i)>0) DrawArrays(mode, *(first+i), *(count+i));
}
```

Parameters

mode	Specifies what kind of primitives to render. Symbolic constants **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** are accepted.
first	Points to an array of the starting indeces in the enabled arrays.
count	Points to an array of the number of indices to be rendered.
primcount	Specifies the size of *first* and *count*.

Notes

The **glMultiDrawArraysEXT** subroutine is included in display lists. If **glMultiDrawArraysEXT** is entered into 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, their values affect display lists when the lists are created, not when the lists are executed.

Errors

GL_INVALID_ENUM is generated if *mode* is not an accepted value.

GL_INVALID_VALUE is generated if *count* is negative.

GL_INVALID_OPERATION is generated if **glMultiDrawArraysEXT** is executed between the execution of **glBegin** and the corresponding **glEnd**.

Related Information

The **glArrayElement** subroutine, **glColorPointer** subroutine, **glMultiDrawElementsEXT** subroutine, **glEdgeFlagPointer** subroutine, **glGetPointerv** subroutine, **glIndexPointer** subroutine, **glNormalPointer** subroutine, **glTexCoordPointer** subroutine, **glVertexPointer** subroutine.

glMultiDrawElementsEXT Subroutine

Purpose

Renders multiple primitives from array data.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glMultiDrawElementsEXT(GLenum  mode,
                            GLsizei  *count,
                            GLenum  type,
                            const GLvoid  **indices,
                            GLsizei  primcount)
```

Description

The **glMultiDrawElementsEXT** subroutine lets you specify multiple geometric primitives with very few subroutine calls. Instead of calling a GL function to pass each individual vertex, normal, texture coordinate, edge flag, or color, you can prespecify separate arrays of vertexes, normals, and so on and use them to construct a sequence of primitives with a single call to **glMultiDrawElementsEXT**.

When **glMultiDrawElementsEXT** is called, it uses count sequential elements from indices to construct a sequence of geometric primitives. **GLenum** *mode* specifies what kind of primitives are constructed and how the array elements construct these primitives. If **GL_VERTEX_ARRAY** is not enabled, no geometric primitives are generated.

Vertex attributes that are modified by **glMultiDrawElementsEXT** have an unspecified value after **glMultiDrawElementsEXT** returns. For example, if **GL_COLOR_ARRAY** is enabled, the value of the current color is undefined after **glMultiDrawElementsEXT** executes. Attributes that are not modified remain well defined.

Parameters

mode	Specifies what kind of primitives to render. Symbolic constants **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** are accepted.
count	Points to an array of the element counts.
type	Specifies the type of the values in indices. Must be one of **GL_UNSIGNED_BYTE**, **GL_UNSIGNED_SHORT**, or **GL_UNSIGNED_INT**.
indices	Specifies a pointer to the location where the indices are stored.
primcount	Specifies the size of the count array.

Notes

The **glMultiDrawElementsEXT** subroutine is included in display lists. If **glMultiDrawElementsEXT** is entered into 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, their values affect display lists when the lists are created, not when the lists are executed.

glMultiDrawElementsEXT is part of the _extname(EXT_multi_draw_arrays) extension, not part of the core GL command set. If _extstring(EXT_multi_draw_arrays) is included in the string returned by **glGetString**, when called with argument **GL_EXTENSIONS**, extension _extname(EXT_multi_draw_arrays) is supported.

Errors

GL_INVALID_ENUM is generated if *mode* is not an accepted value.

GL_INVALID_VALUE is generated if *count* is negative.

GL_INVALID_OPERATION is generated if **glMultiDrawElementsEXT** is executed between the execution of **glBegin** and the corresponding **glEnd**.

Associated Gets

glGetTexImage, **glIsEnabled** with argument **GL_TEXTURE_1D**.

Related Information

The **glArrayElement** subroutine, **glColorPointer** subroutine, **glMultiDrawArraysEXT** subroutine, **glEdgeFlagPointer** subroutine, **glGetPointerv** subroutine, **glIndexPointer** subroutine, **glNormalPointer** subroutine, **glTexCoordPointer** subroutine, **glVertexPointer** subroutine.

glMultiModeDrawArraysIBM Subroutine

Purpose

Renders primitives of multiple primitive types from array data.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glMultiModeDrawArraysIBM( GLenum * mode,
   GLint   * first,
   GLsizei * count,
   GLsizei  primcount,
   GLint    modestride)
```

Description

The **glMultiModeDrawArraysIBM** subroutine behaves identically to **glDrawArrays** except that a list of arrays and a list of primitive modes is specified instead. The number of lists is specified in the *primcount* parameter. It has the same effect as:

```
for(i=0; i < primcount; i++) {
   if (*(count+i) > 0)
      glDrawArrays(*((GLenum *)((char *)mode+i*modestride)),
                 *(first+i),
                 *(count+i));
}
```

Parameters

mode	Points to an array of primitive modes. Symbolic constants **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** are accepted.
first	Points to an array of the starting indices in the enabled arrays.
count	Points to an array of the number of indices to be rendered for each primitive.
primcount	Specifies the word size of the *mode*, *first* and *count* arrays.
modestride	Specifies how to stride through the *mode* array. Typical values are 0 (single primitive mode for all primitives) and *sizeof(GLenum)* (separate primitive mode for each primitive).

Notes

The **glMultiModeDrawArraysIBM** subroutine is available only if the IBM_multi_mode_draw_arrays extension is supported.

The **glMultiModeDrawArraysIBM** subroutine is included in display lists. If **glMultiModeDrawArraysIBM** is entered into 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, their values affect display lists when the lists are created, not when the lists are executed.

Error Codes

- **GL_INVALID_ENUM** is generated if any of the primitive modes in the *mode* array is not an accepted value.
- **GL_INVALID_OPERATION** is generated if **glMultiModeDrawArraysIBM** is executed between the execution of **glBegin** and the corresponding **glEnd**.

Related Information

The **glArrayElement** subroutine, **glColorPointer** subroutine, **glColorPointerListIBM** subroutine, **glDrawElements** subroutine, **glEdgeFlagPointer** subroutine, **glEdgeFlagPointerListIBM** subroutine, **glGetPointerv** subroutine, **glIndexPointer** subroutine, **glIndexPointerListIBM** subroutine, **glInterleavedArrays** subroutine, **glMultiModeDrawElementsIBM** subroutine, **glNormalPointer** subroutine, **glNormalPointerListIBM** subroutine, **glTexCoordPointer** subroutine, **glTexCoordPointerListIBM** subroutine, **glVertexPointer** subroutine, **glVertexPointerListIBM** subroutine.

glMultiModeDrawElementsIBM Subroutine

Purpose

Renders primitives of multiple primitive types from array data.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glMultiModeDrawElementsIBM(GLenum   *mode,
      GLsizei   *count,
      GLenum   type,
      const GLvoid   **indices,
      GLsizei   primcount,
      GLint   modestride)
```

Description

glMultiModeDrawElementsIBM behaves identically to glDrawElements except that a list of arrays and a list of primitive modes is specified instead. The number of lists is specified in the *primcount* parameter. It has the same effect as:

```
for(i=0; i < primcount; i++) {
    if (*(count+i) > 0)
        glDrawElements(*((GLenum *)((char *)mode+i*modestride)),
                    *(count+i),
                    type,
                    *(indices+i));
}
```

Parameters

mode	Points to an array of primitive modes, Specifying what kind of primitives to render. Symbolic constants **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** are accepted.
count	Points to an array of the element counts. Each count specifies the number of elements to be rendered for that primitive.
type	Specifies the type of the values in *indices*. Must be one of **GL_UNSIGNED_BYTE**, **GL_UNSIGNED_SHORT**, or **GL_UNSIGNED_INT**.
indices	Specifies a pointer to the list of index arrays.
primcount	Specifies the number of elements to be read from the *mode* array and from the *count* array (and how many arrays there are in the *indices* list). Each such (*mode*,*count*,*indices*[]) triple tells us how many vertices of the indicated mode are to be rendered, and the location of their array of *indices*.
modestride	Specifies how to stride through the *mode* array. Typical values are 0 (single primitive mode for all primitives)and *sizeof(GLenum)* (separate primitive mode for each primitive)

Notes

The **glMultiModeDrawElementsIBM** subroutine is available only if the IBM_multimode_draw_arrays extension is supported.

The **glMultiModeDrawElementsIBM** subroutine is included in display lists. If **glMultiModeDrawElementsIBM** is entered into 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, their values affect display lists when the lists are created, not when the lists are executed.

Errors

GL_INVALID_ENUM is generated if *mode* is not an accepted value.

GL_INVALID_OPERATION is generated if **glMultiModeDrawElementsIBM** is executed between the execution of **glBegin** and the corresponding **glEnd**.

Related Information

The **glArrayElement** subroutine, **glColorPointer** subroutine, **glColorPointerListIBM** subroutine, **glDrawArrays** subroutine, **glEdgeFlagPointer** subroutine, **glEdgeFlagPointerListIBM** subroutine, **glGetPointerv** subroutine, **glIndexPointer** subroutine, **glIndexPointerListIBM** subroutine, **glMultiModeDrawArraysIBM** subroutine, **glNormalPointer** subroutine, **glNormalPointerListIBM** subroutine, **glTexCoordPointer** subroutine, **glTexCoordPointerListIBM** subroutine, **glVertexPointer** subroutine, **glVertexPointerListIBM** subroutine.

glMultiTexCoordARB Subroutine

Purpose

Sets the current texture coordinates.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
void glMultiTexCoord1dARB( GLenum  target,
                           GLdouble  s )
void glMultiTexCoord1fARB( GLenum  target,
                           GLfloat  s )
void glMultiTexCoord1iARB( GLenum  target,
                           GLint  s )
void glMultiTexCoord1sARB( GLenum  target,
                           GLshort  s )
void glMultiTexCoord2dARB( GLenum  target,
                           GLdouble  s,
                           GLdouble  t )
void glMultiTexCoord2fARB( GLenum  target,
                           GLfloat  s,
                           GLfloat  t )
void glMultiTexCoord2iARB( GLenum  target,
                           GLint  s,
                           GLint  t )
void glMultiTexCoord2sARB( GLenum  target,
                           GLshort  s,
                           GLshort  t )
void glMultiTexCoord3dARB( GLenum  target,
                           GLdouble  s,
                           GLdouble  t,
                           GLdouble  r )
void glMultiTexCoord3fARB( GLenum  target,
                           GLfloat  s,
                           GLfloat  t,
                           GLfloat  r )
void glMultiTexCoord3iARB( GLenum  target,
                           GLint  s,
                           GLint  t,
                           GLint  r )
void glMultiTexCoord3sARB( GLenum  target,
                           GLshort  s,
                           GLshort  t,
                           GLshort  r )
void glMultiTexCoord4dARB( GLenum  target,
                           GLdouble  s,
                           GLdouble  t,
                           GLdouble  r,
                           GLdouble  q )
void glMultiTexCoord4fARB( GLenum  target,
                           GLfloat  s,
                           GLfloat  t,
                           GLfloat  r,
                           GLfloat  q )
void glMultiTexCoord4iARB( GLenum  target,
                           GLint  s,
                           GLint  t,
                           GLint  r,
                           GLint  q )
void glMultiTexCoord4sARB( GLenum  target,
                           GLshort  s,
                           GLshort  t,
```

```
                            GLshort  r,
                            GLshort  q )
void glMultiTexCoord1dvARB( GLenum  target,
                            GLdouble  *v )
void glMultiTexCoord1fvARB( GLenum  target,
                            GLfloat  *v )
void glMultiTexCoord1ivARB( GLenum  target,
                            GLint  *v )
void glMultiTexCoord1svARB( GLenum  target,
                            GLshort  *v )
void glMultiTexCoord2dvARB( GLenum  target,
                            GLdouble  *v )
void glMultiTexCoord2fvARB( GLenum  target,
                            GLfloat  *v )
void glMultiTexCoord2ivARB( GLenum  target,
                            GLint  *v )
void glMultiTexCoord2svARB( GLenum  target,
                            GLshort  *v )
void glMultiTexCoord3dvARB( GLenum  target,
                            GLdouble  *v )
void glMultiTexCoord3fvARB( GLenum  target,
                            GLfloat  *v )
void glMultiTexCoord3ivARB( GLenum  target,
                            GLint  *v )
void glMultiTexCoord3svARB( GLenum  target,
                            GLshort  *v )
void glMultiTexCoord4dvARB( GLenum  target,
                            GLdouble  *v )
void glMultiTexCoord4fvARB( GLenum  target,
                            GLfloat  *v )
void glMultiTexCoord4ivARB( GLenum  target,
                            GLint  *v )
void glMultiTexCoord4svARB( GLenum  target,
                            GLshort  *v )
```

Description

glMultiTexCoordARB specifies texture coordinates in one, two, three or four dimensions. If *t* is not specified it is taken to be 0. If *r* is not specified it is taken to be 0. If *q* is not specified, it is taken to be 1. The current texture coordinates are part of the data that is associated with each vertex and with the current raster position. Initially, the values for *s*, *t*, *r* and *q* are (0, 0, 0, 1).

Parameters

target	specifies texture unit whose coordinates should be modified. The number of texture units is implementation dependent, but must be at least two. Must be one of **GL_TEXTUREi_ARB**, where 0 <= i < the implementation-dependent value of **GL_MAX_TEXTURE_UNITS_ARB**.
s, t, r, q	specifies the *s*, *t*, *r*, and *q* texture coordinates for target texture unit. Not all parameters are present in all forms of the command.
v	specifies a pointer to an array of one, two, three or four elements, which in turn specify the *s*, *t*, *r*, and *q* texture coordinates.

Notes

glMultiTexCoordARB is only supported if **GL_ARB_multitexture** is included in the string returned by **glGetString** when called with the argument **GL_EXTENSIONS**.

The current texture coordinates can be updated at any time. In particular, **glMultiTexCoordARB** can be called between a call to **glBegin** and the corresponding call to **glEnd**.

It is always the case that **GL_TEXTUREi_ARB** = **GL_TEXTURE0_ARB + i**.

Associated Gets

Associated gets for the **glMultiTexCoordARB** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet GL_CURRENT_TEXTURE_COORDS with appropriate texture unit selected.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glActiveTextureARB** subroutine, the **glClientActiveTextureARB** subroutine, the **glTexCoord** subroutine, the **glTexCoordPointer** subroutine.

---

glMultMatrix Subroutine

Purpose

Multiplies the current matrix by an arbitrary matrix.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glMultMatrixd(const GLdouble  *Matrix)

void glMultMatrixf(const GLfloat  *Matrix)
```

Description

The **glMultMatrix** subroutine multiplies the current matrix with the one specified in the *Matrix* parameter. For example, if M is the current matrix and T is the matrix passed to **glMultMatrix**, M is replaced with MT.

The current matrix is the projection matrix, model view matrix, or texture matrix, determined by the current matrix mode. (See the **glMatrixMode** subroutine for information on specifying the current matrix.)

The *Matrix* parameter points to a 4 x 4 matrix of single- or double-precision floating-point values stored in column-major order. That is, the matrix is stored as in the following figure:

$$\begin{pmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{pmatrix}$$

Figure 15. Stored Matrix. This diagram shows a matrix enclosed in brackets. The matrix consists of four lines containing four characters each. The first line contains the following (from left to right): a subscript zero, a subscript four, a subscript eight, a subscript twelve. The second line contains the following (from left to right): a subscript one, a subscript five, a subscript nine, a subscript thirteen. The third line contains the following (from left to right): a subscript two, a subscript six, a subscript ten, a subscript fourteen. The fourth line contains the following (from left to right): a subscript three, a subscript seven, a subscript eleven, a subscript fifteen.

Parameters

Matrix	Specifies a pointer to 4 x 4 matrix stored in column-major order as 16 consecutive values.

Errors

GL_INVALID_OPERATION	The **glMultMatrix** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glMultMatrix** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_MATRIX_MODE**.

glGet with argument **GL_MODELVIEW_MATRIX**.

glGet with argument **GL_PROJECTION_MATRIX**.

glGet with argument **GL_TEXTURE_MATRIX**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glLoadIdentity** subroutine, **glLoadMatrix** subroutine, **glMatrixMode** subroutine, **glPushMatrix** subroutine, **glMultTransposeMatrixARB** subroutine.

glMultTransposeMatrixARB Subroutine

Purpose

Multiplies the current matrix by a matrix specified in row-major order, rather than column-major order.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
void glMultTransposeMatrixfARB(const GLfloat *Matrix)
void glMultTransposeMatrixdARB(const GLdouble *Matrix)
```

Description

The **glMultTransposeMatrixARB** subroutine replaces the current matrix with the product of the current matrix and the one specified in the *Matrix* parameter. The current matrix is the projection matrix, model view matrix, or texture matrix, determined by the current matrix mode. (See the **glMatrixMode** subroutine for information on specifiying the current matrix.) *The Matrix* parameter points to a 4 x 4 matrix of single- or double-precision floating-point values stored in row-major order. That is, the matrix is stored as the following:

```
/ a0  a1  a2  a3  \
| a4  a5  a6  a7  |
| a8  a9  a10 a11 |
\ a12 a13 a14 a15 /
```

The effect on an input vertex is as if it is first multiplied by the matrix specified in this call, and then subsequently multiplied by the previous ″current″ matrix.

Parameters

Matrix	is an array of 16 values, specified in row-major order.

Error Codes

GL_INVALID_OPERATION	is generated if **glMultTransposeMatrixARB** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glMultMatrix** subroutine, the **glMatrixMode** subroutine.

glNewList or glEndList Subroutine

Purpose

Creates or replaces a display list.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glNewList(GLuint  List,
     GLenum  Mode)
```

Description

Display lists are groups of GL commands that have been stored for subsequent execution. The display lists are created with the **glNewList** subroutine. All subsequent commands are placed in the display list, in the order issued, until the **glEndList** subroutine is called.

The **glNewList** subroutine has two arguments. The first argument, *List*, is a positive integer that becomes the unique name for the display list. Names can be created and reserved with the **glGenLists** subroutine and tested for uniqueness with the **glIsList** subroutine. The second argument, *Mode*, is a symbolic constant that can assume one of two values:

GL_COMPILE	Commands are compiled only.
GL_COMPILE_AND_EXECUTE	Commands are performed as they are compiled into the display list.

The following subroutines are not compiled into the display list, but are performed immediately, regardless of the display-list mode:

- **glIsList**
- **glGenLists**
- **glDeleteLists**
- **glFeedbackBuffer**
- **glSelectBuffer**
- **glRenderMode**
- **glReadPixels**
- **glPixelStore**
- **glFlush**
- **glFinish**
- **glIsEnabled**
- All **glGet** subroutines

When **glEndList** is encountered, the display-list definition is completed by associating the list with the unique name *List* (specified in **glNewList**). If a display list with the name *List* already exists, it is replaced only when **glEndList** is called.

Parameters

List	Specifies the display list name.
Mode	Specifies the compilation mode, which can be **GL_COMPILE** or **GL_COMPILE_AND_EXECUTE**.

Notes

The **glCallList** and **glCallLists** subroutines can be entered into display lists. The commands in the display list or lists run by **glCallList** or **glCallLists** are not included in the display list being created, even if the list creation mode is **GL_COMPILE_AND_EXECUTE**.

Error Codes

GL_INVALID_VALUE	*List* is 0 (zero).
GL_INVALID_ENUM	*Mode* is not an accepted value.

GL_INVALID_OPERATION	The **glEndList** subroutine is called without a preceding **glNewList**. OR The **glNewList** subroutine is called while a display list is being defined.
GL_INVALID_OPERATION	The **glNewList** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glNewList** and **glEndList** subroutines are as follows. (See the **glGet** subroutine for more information.)

glIsList.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glCallList** subroutine, **glCallLists** subroutine, **glDeleteLists** subroutine, **glGenLists** subroutine.

glNormal Subroutine

Purpose

Set the current normal vector; for use in lighting calculations.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
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)
```

```
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)
```

Description

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. To enable and disable normalization, call **glEnable** and **glDisable** with the argument **GL_NORMALIZE**. Normalization is initially disabled.

Parameters

nx, *ny*, *nz*	Specify the *x*, *y*, and *z* coordinates of the new current normal. The initial value of the current normal is the unit vector, (0, 0, 1).
v	Specifies a pointer to an array of three elements: the *x*, *y*, and *z* coordinates of the new current normal.

Notes

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**.

Associated Gets

glGet with argument **GL_CURRENT_NORMAL**

glIsEnabled with argument **GL_NORMALIZE**

Related Information

The **glBegin** subroutine, **glColor** subroutine, **glIndex** subroutine, **glNormalPointer** subroutine, **glTexCoord** subroutine, and the **glVertex** subroutine.

glNormalPointer Subroutine

Purpose

Defines an array of normals.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glNormalPointer( GLenum  type,
   GLsizei  stride,
   const GLvoid * pointer)
```

Description

The **glNormalPointer** subroutine specifies the location and data format of an array of normals to use when rendering. The *type* parameter specifies the data type of the normal coordinates and *stride* gives the byte stride from one normal to the next allowing vertices and attributes to be packed into a single array or stored in separate arrays. (Single array storage may be more efficient on some implementations; see **glInterleavedArrays**). When a normal array is specified, *type*, *stride*, and *pointer* are saved as client side state.

To enable and disable the normal array, call **glEnableClientState** and **glDisableClientState** with the argument **GL_NORMAL_ARRAY**. If enabled, the normal array is used when **glDrawArrays**, **glDrawElements** or **glArrayElement** is called.

Use **glDrawArrays**, **glMultiDrawArraysEXT**, or **glMultiModeDrawArraysIBM** to construct a sequence of primitives from prespecified vertex and vertex attribute arrays. Use **glArrayElement** to specify primitives by indexing vertices and vertex attributes. Use **glDrawElements**, **glMultiDrawElementsEXT**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

If enabled, the Normal array is used when **glDrawArrays**, **glDrawElements**, **glArrayElements**, **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** is called.

Parameters

type	Specifies the the data type of each coordinate in the array. Symbolic constants **GL_BYTE**, **GL_SHORT**, **GL_INT**, **GL_FLOAT**, or **GL_DOUBLE** are accepted. The initial value is **GL_FLOAT**.
stride	Specifies the byte offset between consecutive normals. The initial value is 0.
pointer	Specifies a pointer to the first coordinate of the first normal in the array. The initial value is 0 (NULL pointer).

Notes

The **glNormalPointer** subroutine is available only if the GL version is 1.1 or greater.

The normal array is initially disabled and it won't be accessed when **glArrayElement**, **glDrawElements** or **glDrawArrays** is called.

Execution of **glNormalPointer** is not allowed between **glBegin** and the corresponding **glEnd**, but an error may or may not be generated. If an error is not generated, the operation is undefined.

The **glNormalPointer** subroutine is typically implemented on the client side with no protocol.

Since the normal array parameters are client side state, they are not saved or restored by **glPushAttrib** and **glPopAttrib**. Use **glPushClientAttrib** and **glPopClientAttrib** instead.

The **glNormalPointer** subroutine is not included in display lists.

Errors

- **GL_INVALID_ENUM** is generated if type is not an accepted value.
- **GL_INVALID_VALUE** is generated if stride is negative.

Associated Gets

- **glIsEnabled** with argument **GL_NORMAL_ARRAY**
- **glGet** with argument **GL_NORMAL_ARRAY_TYPE**
- **glGet** with argument **GL_NORMAL_ARRAY_STRIDE**
- **glGetPointerv** with argument **GL_NORMAL_ARRAY_POINTER**

Related Information

The **glArrayElement** subroutine, **glColorPointer** subroutine, **glDrawArrays** subroutine, **glDrawElements** subroutine, **glEdgeFlagPointer** subroutine, **glEnable** subroutine, **glGetPointerv** subroutine, **glIndexPointer** subroutine, **glNormalPointerListIBM** subroutine, **glPopClientAttrib** subroutine, **glPushClientAttrib** subroutine, **glTexCoordPointer** subroutine, **glVertexPointer** subroutine.

glNormalPointerEXT Subroutine

Purpose

Defines an array of normals.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glNormalPointerEXT(GLenum  type,
     GLsizei  stride,
     GLsizei  count,
     const GLvoid  *pointer)
```

Description

glNormalPointerEXT specifies the location and data format of an array of normals to use when rendering. *type* specifies the data type of the normal coordinates and *stride* gives the byte stride from one normal to the next allowing vertexes and attributes to be packed into a single array or stored in separate arrays. (Single-array storage may be more efficient on some implementations.) *count* indicates the number of array elements (counting from the first) that are static. Static elements may be modified by the application, but once they are modified, the application must explicitly respecify the array before using it for any rendering. When a normal array is specified, *type*, *stride*, *count* and *pointer* are saved as client-side state, and static array elements may be cached by the implementation.

The normal array is enabled and disabled using **glEnable** and **glDisable** with the argument **GL_NORMAL_ARRAY_EXT**. If enabled, the normal array is used when **glDrawArraysEXT** or **glArrayElementEXT** is called.

Use **glDrawArraysEXT** to define a sequence of primitives (all of the same type) from pre-specified vertex and vertex attribute arrays. Use **glArrayElementEXT** to specify primitives by indexing vertexes and vertex attributes.

Use **glDrawArrays**, **glMultiDrawArraysEXT**, or **glMultiModeDrawArraysIBM** to construct a sequence of primitives from prespecified vertex and vertex attribute arrays. Use **glArrayElement** to specify primitives by indexing vertices and vertex attributes. Use **glDrawElements**, **glMultiDrawElementsEXT**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

If enabled, the Normal array is used when **glDrawArrays**, **glDrawElements**, **glArrayElements**, **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** is called.

Parameters

type	Specifies the the data type of each coordinate in the array. Symbolic constants **GL_BYTE**, **GL_SHORT**, **GL_INT**, **GL_FLOAT**, or **GL_DOUBLE_EXT** are accepted.
stride	Specifies the byte offset between consecutive normals.
count	Specifies the number of normals, counting from the first, that are static.
pointer	Specifies a pointer to the first coordinate of the first normal in the array.

Notes

Non-static array elements are not accessed until **glArrayElementEXT** or **glDrawArraysEXT** is executed.

By default the normal array is disabled and it won't be accessed when **glArrayElementEXT** or **glDrawArraysEXT** is called.

Although, it is not an error to call **glNormalPointerEXT** between the execution of **glBegin** and the corresponding execution of **glEnd**, the results are undefined.

glNormalPointerEXT will typically be implemented on the client side with no protocol.

Since the normal array parameters are client side state, they are not saved or restored by **glPushAttrib** and **glPopAttrib**.

glNormalPointerEXT commands are not entered into display lists.

glNormalPointerEXT is part of the _extname(EXT_vertex_array) extension, not part of the core GL command set. If _extstring(EXT_vertex_array) is included in the string returned by **glGetString**, when called with argument **GL_EXTENSIONS**, extension _extname(EXT_vertex_array) is supported.

Errors

GL_INVALID_ENUM is generated if *type* is not an accepted value.

GL_INVALID_VALUE is generated if *stride* or *count* is negative.

Associated Gets

glIsEnabled with argument **GL_NORMAL_ARRAY_EXT** .

glGet with argument **GL_NORMAL_ARRAY_TYPE_EXT**.

glGet with argument **GL_NORMAL_ARRAY_STRIDE_EXT**.

glGet with argument **GL_NORMAL_ARRAY_COUNT_EXT**.

glGetPointervEXT with argument **GL_NORMAL_ARRAY_POINTER_EXT**.

File

/usr/include/GL/glext.h	Contains extensions to C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glArrayElementEXT** subroutine, **glColorPointerEXT** subroutine, **glDrawArraysEXT** subroutine, **glEdgeFlagPointerEXT** subroutine, **glGetPointervEXT** subroutine, **glIndexPointerEXT** subroutine, **glTexCoordPointerEXT** subroutine, **glVertexPointerEXT** subroutine.

glNormalPointerListIBM Subroutine

Purpose

Defines a list of normal arrays.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glNormalPointerListIBM(GLenum  type,
     GLint  stride,
     const GLvoid ** pointer,
     GLint  ptrstride)
```

Description

The **glNormalPointerListIBM** subroutine specifies the location and data format of a list of arrays of normal components to use when rendering. The *type* parameter specifies the data type of each normal component. The *stride* parameter gives the byte stride from one normal to the next allowing vertices and attributes to be packed into a single array or stored in separate arrays. (Single-array storage may be more efficient on some implementations; see **glInterleavedArrays**). The *ptrstride* parameter specifies the byte stride from one pointer to the next in the *pointer* array.

When a normal array is specified, *type*, *stride*, *pointer* and *ptrstride* are saved as client side state.

A *stride* value of 0 does not specify a ″tightly packed″ array as it does in **glNormalPointer**. Instead, it causes the first array element of each array to be used for each vertex. Also, a negative value can be used for stride, which allows the user to move through each array in reverse order.

To enable and disable the normal arrays, call **glEnableClientState** and **glDisableClientState** with the argument **GL_NORMAL_ARRAY**. The normal array is initially disabled. When enabled, the normal arrays are used when **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, **glDrawArrays**, **glDrawElements** or **glArrayElement** is called. The last three calls in this list will only use the first array (the one pointed at by *pointer*[0]). See the descriptions of these routines for more information on their use.

Use **glDrawArrays**, **glMultiDrawArraysEXT**, or **glMultiModeDrawArraysIBM** to construct a sequence of primitives from prespecified vertex and vertex attribute arrays. Use **glArrayElement** to specify primitives by indexing vertices and vertex attributes. Use **glDrawElements**, **glMultiDrawElementsEXT**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

If enabled, the Normal array is used when **glDrawArrays**, **glDrawElements**, **glArrayElements**, **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** is called.

Parameters

type	Specifies the data type of each normal component in the array. Symbolic constants **GL_BYTE**, **GL_UNSIGNED_BYTE**, **GL_SHORT**, **GL_UNSIGNED_SHORT**, **GL_INT**, **GL_UNSIGNED_INT**, **GL_FLOAT**, or **GL_DOUBLE** are accepted. The initial value is GL_FLOAT.
stride	Specifies the byte offset between consecutive normal. The initial value is 0.
pointer	Specifies a list of normal arrays. The initial value is 0 (NULL pointer).
ptrstride	Specifies the byte stride between successive pointers in the *pointer* array. The initial value is 0.

Notes

The **glNormalPointerListIBM** subroutine is available only if the GL_IBM_vertex_array_lists extension is supported.

Execution of **glNormalPointerListIBM** is not allowed between **glBegin** and the corresponding **glEnd**, but an error may or may not be generated. If an error is not generated, the operation is undefined.

The **glNormalPointerListIBM** subroutine is typically implemented on the client side.

Since the normal array parameters are client side state, they are not saved or restored by **glPushAttrib** and **glPopAttrib**. Use **glPushClientAttrib** and **glPopClientAttrib** instead.

When a **glNormalPointerListIBM** call is encountered while compiling a display list, the information it contains does NOT contribute to the display list, but is used to update the immediate context instead.

The **glNormalPointer** call and the **glNormalPointerListIBM** call share the same state variables. A **glNormalPointer** call will reset the normal list state to indicate that there is only one normal list, so that any and all lists specified by a previous **glNormalPointerListIBM** call will be lost, not just the first list that it specified.

Error Codes

GL_INVALID_ENUM is generated if type is not an accepted value.

Associated Gets

glIsEnabled with argument **GL_NORMAL_ARRAY**

glGetPointerv with argument **GL_NORMAL_ARRAY_LIST_IBM**

glGet with argument **GL_NORMAL_ARRAY_LIST_STRIDE_IBM**

glGet with argument **GL_NORMAL_ARRAY_STRIDE**

glGet with argument **GL_NORMAL_ARRAY_TYPE**

Related Information

The **glArrayElement** subroutine, **glDrawArrays** subroutine, **glDrawElements** subroutine, **glEdgeFlagPointer** subroutine, **glEnable** subroutine, **glGetPointerv** subroutine, **glIndexPointer** subroutine, **glInterleavedArrays** subroutine, **glMultiDrawArraysEXT** subroutine, **glMultiDrawElementsEXT** subroutine, **glMultiModeDrawArraysIBM** subroutine,

glMultiModeDrawElementsIBM subroutine, **glNormalPointer** subroutine, **glPopClientAttrib** subroutine, **glPushClientAttrib** subroutine, **glTexCoordPointer** subroutine, **glVertexPointer** subroutine.

glNormalVertexSUN Subroutine

Purpose

Specifies a normal and a vertex in one call.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
void glNormal3fVertex3fSUN (GLfloat  nx,
                            GLfloat  ny,
                            GLfloat  nz,
                            GLfloat  x,
                            GLfloat  y,
                            GLfloat  z)
void glNormal3fVertex3fvSUN (const GLfloat  *n,
                             const GLfloat  *v)
```

Description

This subroutine can be used as a replacement for the following calls:

```
glNormal();
glVertex();
```

For example, **glNormal3fVertex3fvSUN** replaces the following calls:

```
glNormal3f();
glVertex3fv();
```

The only reason for using this call is that it reduces the use of bus bandwidth.

Parameters

x, y, z	Specifies the *x*, *y*, and *z* coordinates of a vertex. Not all parameters are present in all forms of the command.
v	Specifies a pointer to an array of the three elements *x*, *y*, and *z*.
nx, ny, nz	Specify *x*, *y*, and *z* coordinates of the normal vector for this vertex.
n	Specifies a pointer to an array of the three elements *nx*, *ny* and *nz*.

Notes

Calling glNormalVertexSUN outside of a **glBegin**/**glEnd** subroutine pair results in undefined behavior.

This subroutine is only valid if the **GL_SUN_vertex** extension is defined.

Files

/usr/include/GL/gl.h Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, the **glColor** subroutine, the **glNormal** subroutine, the **glTexCoord** subroutine, the **glVertex** subroutine.

glOrtho Subroutine

Purpose

Multiplies the current matrix by an orthographic matrix.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glOrtho(GLdouble  Left,
     GLdouble  Right,
     GLdouble  Bottom,
     GLdouble  Top,
     GLdouble  Near,
     GLdouble  Far)
```

Description

The **glOrtho** subroutine describes a perspective matrix that produces a parallel projection. (*Left*, *Bottom*, -*Near*) and (*Right*, *Top*, -*Near*) 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). -*Far* specifies the location of the far clipping plane. Both *Near* and *Far* can be either positive or negative. The corresponding matrix is as follows:

$$\begin{pmatrix} \frac{2}{Right-Left} & 0 & 0 & t_x \\ 0 & \frac{2}{Top-Bottom} & 0 & t_y \\ 0 & 0 & \frac{-2}{Far-Near} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 16. Parallel Projection Perspective Matrix. This diagram shows a matrix enclosed in brackets. The matrix consists of four lines containing four characters each. The first line contains the following (from left to right): 2 / Right-Left, zero, zero, t subscript x. The second line contains the following (from left to right): zero, 2 / Top-Bottom, zero, t subscript y. The third line contains the following (from left to right): zero, zero, -2 / Far-Near, t subscript z. The fourth line contains the following (from left to right): zero, zero, zero, one.

where the following statements apply:

$$t_x = -\frac{Right+Left}{Right-Left}$$

$$t_y = -\frac{Top+Bottom}{Top-Bottom}$$

$$t_z = -\frac{Far+Near}{Far-Near}$$

Figure 17. Statements. This figure shows three equations. The first equation shows that t subscript x (from the above matrix) is equal to negative Right+Left / Right–Left. The second equation shows that t subscript y (from the above matrix) is equal to negative Top+Bottom / Top–Bottom. The third equation shows that t subscript z (from the above matrix) is equal to negative Far+Near / Far–Near.

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, M is replaced with MO.

Use the **glPushMatrix** and **glPopMatrix** subroutines to save and restore the current matrix stack.

Parameters

Left, *Right*	Specify the coordinates for the left and right vertical clipping planes.
Bottom, *Top*	Specify the coordinates for the bottom and top horizontal clipping planes.
Near, *Far*	Specify the distances to the nearer and farther depth clipping planes. These distances are negative if the plane is to be behind the viewer.

Errors

GL_INVALID_OPERATION	The **glOrtho** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glOrtho** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_MATRIX_MODE**

glGet with argument **GL_MODELVIEW_MATRIX**

glGet with argument **GL_PROJECTION_MATRIX**

glGet with argument **GL_TEXTURE_MATRIX**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glFrustum** subroutine, **glMatrixMode** subroutine, **glMultMatrix** subroutine, **glPushMatrix** subroutine, **glViewport** subroutine.

glPassThrough Subroutine

Purpose

Places a marker in the feedback buffer.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glPassThrough(GLfloat Token)
```

Description

Feedback is a GL render mode. The mode is selected by calling the **glRenderMode** subroutine with **GL_FEEDBACK**. When the GL 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 the GL. See the **glFeedbackBuffer** subroutine for a description of the feedback buffer and the values in the feedback buffer.

The **glPassThrough** subroutine 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; it is indicated with its own unique identifying value: **GL_PASS_THROUGH_TOKEN**. The order of **glPassThrough** commands with respect to the specification of graphics primitives is maintained.

Parameters

Token	Specifies a marker value to be placed in the feedback buffer following a **GL_PASS_THROUGH_TOKEN** value.

Notes

The **glPassThrough** subroutine is ignored if the GL is not in feedback mode.

Errors

GL_INVALID_OPERATION	The **glPassThrough** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glPassThrough** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_RENDER_MODE**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glFeedbackBuffer** subroutine, **glRenderMode** subroutine.

glPixelMap Subroutine

Purpose

Sets up pixel transfer maps.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
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)
```

Description

The **glPixelMap** subroutine sets up translation tables, or *maps*, used by the **glDrawPixels**, **glReadPixels**, **glCopyPixels**, **glTexImage1D**, and **glTexImage2D** subroutines. See the **glPixelTransfer** subroutine for a complete description on using these maps. Use of these maps is also described in part in the pixel and texture image subroutines. Only the specification of the maps is described here.

The *Map* parameter is a symbolic map name, indicating one of 10 maps to set. The *MapSize* parameter specifies the number of entries in the map, and the *Values* parameter is a pointer to an array of *MapSize* map values.

The 10 maps are:

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.

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 the **GL_PIXEL_MAP_I_TO_I** and **GL_PIXEL_MAP_S_TO_S** maps) 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, 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 0 (zero) maps to 0.0.

Maps that store indices, **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 0s 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* = 2n for some *n* or results are undefined. The maximum allowable size for each map depends on the implementation and can be determined by calling the **glGet** subroutine with argument **GL_MAX_PIXEL_MAP_TABLE**. The single maximum applies to all maps, and it is at least 32.

Map	**Lookup Index**	**Lookup Value**	**Initial Size**	**Initial Value**
GL_PIXEL_MAP_I_TO_I	color index	color index	1	0.0
GL_PIXEL_MAP_S_TO_S	stencil index	stencil index	1	0
GL_PIXEL_MAP_I_TO_R	color index	R	1	0.0
GL_PIXEL_MAP_I_TO_G	color index	G	1	0.0
GL_PIXEL_MAP_I_TO_B	color index	B	1	0.0
GL_PIXEL_MAP_I_TO_A	color index	A	1	0.0
GL_PIXEL_MAP_R_TO_R	R	R	1	0.0
GL_PIXEL_MAP_G_TO_G	G	G	1	0.0
GL_PIXEL_MAP_B_TO_B	B	B	1	0.0
GL_PIXEL_MAP_A_TO_A	A	A	1	0.0

Parameters

Map Specifies a symbolic map name. *Map* must be one of the following:

- **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**
- **GL_PIXEL_MAP_A_TO_A**

MapSize Specifies the size of the map being defined.

Values Specifies an array of *MapSize* values.

Errors

GL_INVALID_ENUM *Map* is not an accepted value.

GL_INVALID_VALUE *MapSize* is negative or larger than **GL_MAX_PIXEL_MAP_TABLE**.

GL_INVALID_VALUE *Map* is **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* is not a power of two.

GL_INVALID_OPERATION	The **glPixelMap** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glPixelMap** subroutine are as follows. (See the **glGet** subroutine for more information.)

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**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glCopyPixels** subroutine, **glDrawPixels** subroutine, **glGetPixelMap** subroutine, **glPixelStore** subroutine, **glPixelTransfer** subroutine, **glReadPixels** subroutine, **glTexImage1D** subroutine, **glTexImage2D** subroutine.

glPixelStore Subroutine

Purpose

Sets pixel storage modes.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glPixelStoref(GLenum  pName,
     GLfloat Parameter)

void glPixelStorei(GLenum  pName,
     GLint Parameter)
```

Description

The **glPixelStore** subroutine sets pixel storage modes that affect the operation of subsequent **glDrawPixels** and **glReadPixels** subroutines as well as the unpacking of polygon stipple patterns (see the **glPolygonStipple** subroutine), bitmaps (see the **glBitmap** subroutine), and texture patterns (see the **glTexImage1D** subroutine, **glTexImage2D** subroutine, **glTexImage3DEXT** subroutine, **glTexSubImage1D** subroutine, **glTexSubImage2D** subroutine, and the **glTexSubImage3DEXT** subroutine).

The *pName* parameter is a symbolic constant indicating the parameter to be set, and the *Parameter* parameter is the new value. The following 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 4-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 0 (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, the location of the first pixel of the next row is obtained by skipping the result of the equation in Figure 18.

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 0; otherwise, the *width* argument to the pixel routine), *a* is the value of **GL_PACK_ALIGNMENT**, and *s* is the size, in bytes, of a single component (if *a* < *s*, it is as if *a* = *s*). In the case of 1-bit values, the location of the next row is obtained by skipping the result of the equation in Figure 19 on page 257.

The word *component* in this description refers 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.

$$k = \begin{cases} nl & s \geq a \\ \dfrac{a}{s}\left\lceil \dfrac{snl}{a} \right\rceil & s < a \end{cases} \quad \text{components or indexes}$$

Figure 18. GL_PACK_ROW_LENGTH Equation. This figure shows an equation where k is equal to the following two lines preceded by a single curly brace: nl s greater than or equal to a. Below the first line is the second line as follows: a / s [snl / a] s less than a components or indexes.

$$k = 8a \left\lceil \frac{nl}{8a} \right\rceil \text{ components or indexes}$$

Figure 19. GL_PACK_ROW_LENGTH 1–bit Values Equation. This figure shows an equation where k is equal to 8a[nl / 8a] components or indexes.

GL_PACK_IMAGE_HEIGHT
If greater than 0 (zero), **GL_PACK_IMAGE_HEIGHT** defines the number of rows in a 3D image, otherwise the number of rows is defined to be the height of the 3D image.

GL_PACK_SKIP_PIXELS, GL_PACK_SKIP_ROWS, and **GL_PACK_SKIP_IMAGES**
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 the **glReadPixels** subroutine. Setting **GL_PACK_SKIP_PIXELS** to *i* is equivalent to incrementing the pointer by *in* 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 *jk* components or indexes, where *k* is the number of components or indexes per row, as computed in the **GL_PACK_ROW_LENGTH** section. Setting the **GL_PACK_SKIP_IMAGES** to l is equivalent to incrementing the pointer by *lmk* components or indexes, where *m* is the number of rows per image as specified by **GL_PACK_IMAGE_HEIGHT**.

GL_PACK_ALIGNMENT
Specifies the alignment requirements for the start of each pixel row in memory. The allowable values are 1 (one) (byte alignment), 2 (rows aligned to even-numbered bytes), 4 (word alignment), and 8 (rows start on double-word boundaries).

The remaining parameters affect how pixel data is read from client memory. These values are significant for the **glDrawPixels**, **glTexImage1D**, **glTexImage2D**, **glBitmap**, and **glPolygonStipple** subroutines. They are as follows:

GL_UNPACK_SWAP_BYTES
If True, byte ordering for a multibyte color components, depth components, color indexes, or stencil indexes is reversed. That is, if a 4-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 0, **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 the result of the equation in Figure 20.

Where *n* is the number of components or indexes in a pixel, *i* is the number of pixels in a row (**GL_UNPACK_ROW_LENGTH** if it is greater than 0; otherwise, the *width* argument to the pixel routine), *a* is the value of **GL_UNPACK_ALIGNMENT**, and *s* is the size, in bytes, of a single component (if $a < s$, it is as if $a = s$). In the case of 1-bit values, the location of the next row is obtained by skipping the result of the equation in Figure 21.

The word *component* in this description refers 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.

$$k = \begin{cases} nl & s \geq a \\ \dfrac{a}{s}\left\lceil \dfrac{snl}{a} \right\rceil & s < a \end{cases} \quad \text{components or indexes}$$

Figure 20. GL_UNPACK_ROW_LENGTH Equation. This figure shows an equation where k is equal to the following two lines preceded by a single curly brace: nl s greater than or equal to a. Below the first line is the second line as follows: a / s [snl / a] s less than a components or indexes.

$$k = 8a\left\lceil \frac{nl}{8a} \right\rceil \quad \text{components or indexes}$$

Figure 21. GL_UNPACK_ROW_LENGTH 1–bit Values Equation. This figure shows an equation where k is equal to 8a[nl / 8a] components or indexes.

GL_UNPACK_IMAGE_HEIGHT

If greater than 0 (zero), **GL_UNPACK_IMAGE_HEIGHT** defines the number of rows in a 3D image, otherwise the number of rows is defined to be the height of the 3D image.

GL_UNPACK_SKIP_PIXELS, GL_UNPACK_SKIP_ROWS, and **GL_UNPACK_SKIP_IMAGES**

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**, **glTexImage3DEXT**, **glBitmap**, or **glPolygonStipple**. Setting **GL_UNPACK_SKIP_PIXELS** to *i* is equivalent to incrementing the pointer by *in* 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 *jk* components or indexes, where *k* is the number of components or indexes per row, as computed in the **GL_UNPACK_ROW_LENGTH** section. Setting the **GL_UNPACK_SKIP_IMAGES** to *l* is equivalent to incrementing the pointer by *lmk* components or indexes, where *m* is the number of rows per image as specified by **GL_UNPACK_IMAGE_HEIGHT**.

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).

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_LSB_FIRST	Boolean	False	True or False
GL_PACK_ROW_LENGTH	integer	0	[0,+infinity)
GL_PACK_SKIP_ROWS	integer	0	[0,+infinity)
GL_PACK_SKIP_PIXELS	integer	0	[0,+infinity)
GL_PACK_ALIGNMENT	integer	4	1, 2, 4, or 8
GL_PACK_IMAGE_HEIGTH	integer	0	[0,+infinity)
GL_PACK_SKIP_IMAGES	integer	0	[0,+infinity)
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,+infinity)
GL_UNPACK_SKIP_ROWS	integer	0	[0,+infinity)
GL_UNPACK_SKIP_PIXELS	integer	0	[0,+infinity)
GL_UNPACK_ALIGNMENT	integer	4	1, 2, 4, or 8
GL_UNPACK_IMAGE_HEIGTH	integer	0	[0,+infinity)
GL_UNPACK_SKIP_IMAGES	integer	0	[0,+infinity)

The **glPixelStoref** subroutine can be used to set any pixel store parameter. If the parameter type is Boolean, and if *Parameter* is 0.0, the parameter is False; otherwise it is set to True. If *pName* is an integer type parameter, *Parameter* is rounded to the nearest integer.

Likewise, **glPixelStorei** can also be used to set any of the pixel store parameters. Boolean parameters are set to False if *Parameter* is 0 and True otherwise. *Parameter* is converted to floating-point format before being assigned to real-valued parameters.

Parameters

pName Specifies the symbolic name of the parameter to be set. The following values affect the packing of pixel data into memory:

- **GL_PACK_SWAP_BYTES**
- **GL_PACK_LSB_FIRST**
- **GL_PACK_ROW_LENGTH**
- **GL_PACK_SKIP_PIXELS**
- **GL_PACK_SKIP_ROWS**
- **GL_PACK_ALIGNMENT**
- **GL_PACK_IMAGE_HEIGHT**
- **GL_PACK_SKIP_IMAGES**

The following values affect the unpacking of pixel data *from* memory:

- **GL_UNPACK_SWAP_BYTES**
- **GL_UNPACK_LSB_FIRST**
- **GL_UNPACK_ROW_LENGTH**
- **GL_UNPACK_SKIP_PIXELS**
- **GL_UNPACK_SKIP_ROWS**
- **GL_UNPACK_ALIGNMENT**
- **GL_UNPACK_IMAGE_HEIGHT**
- **GL_UNPACK_SKIP_IMAGES**

Notes

The pixel storage modes in effect when **glDrawPixels**, **glReadPixels**, **glTexImage**, **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.

Errors

GL_INVALID_ENUM	*pName* is not an accepted value.
GL_INVALID_VALUE	A negative row length, pixel skip, or row skip value is specified, or alignment is specified as other than 1, 2, 4, or 8.

Associated Gets

Associated gets for the **glPixelStore** subroutine are as follows. (See the **glGet** subroutine for more information.)

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_PACK_IMAGE_HEIGTH**

glGet with argument **GL_PACK_SKIP_IMAGES**

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**.

glGet with argument **GL_UNPACK_IMAGE_HEIGTH**

glGet with argument **GL_UNPACK_SKIP_IMAGES**

GL_INVALID_OPERATION	The **glPixelStore** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glBitmap** subroutine, **glDrawPixels** subroutine, **glPixelMap** subroutine, **glPixelTransfer** subroutine, **glPixelZoom** subroutine, **glPolygonStipple** subroutine, **glReadPixels** subroutine, **glTexImage1D** subroutine, **glTexImage2D** subroutine, **glTexImage3DEXT** subroutine, **glTexSubImage1D** subroutine, **glTexSubImage2D** subroutine, **glTexSubImage3DEXT** subroutine.

glPixelTransfer Subroutine

Purpose

Sets pixel transfer modes.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glPixelTransferf(GLenum  pName,
        GLfloat  Parameter)

void glPixelTransferi(GLenum  pName,
        GLint  Parameter)
```

Description

The **glPixelTransfer** subroutine sets pixel transfer modes that affect the operation of subsequent **glDrawPixels**, **glReadPixels**, **glCopyPixels**, **glCopyTexImage1D**, **glCopyTexImage2D**, **glCopyTexSubImage1D**, **glCopyTexSubImage2D**, **glCopyTexSubImage3DEXT**, **glTexImage1D**,

glTexImage2D, **glTexImage3DEXT**, **glTexSubImage1D**, **glTexSubImage2D**, and **glTexSubImage3DEXT** subroutines. 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**, **glReadPixels**, **glCopyPixels**, **glCopyTexImage1D**, **glCopyTexImage2D**, **glCopyTexSubImage1D**, **glCopyTexSubImage2D**, **glCopyTexSubImage3DEXT**, **glTexImage1D**, **glTexImage2D**, **glTexImage3DEXT**, **glTexSubImage1D**, **glTexSubImage2D**, and **glTexSubImage3DEXT** subroutines). 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 control the unpacking of pixels being read from client memory and the packing of pixels being written back into client memory. (See the **glPixelStore** subroutine for information on setting pixel storage modes.)

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 0 (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, then added to a bias factor. That is, the red component is multiplied by **GL_RED_SCALE**, then added to **GL_RED_BIAS**; the green component is multiplied by **GL_GREEN_SCALE**, then added to **GL_GREEN_BIAS**; the blue component is multiplied by **GL_BLUE_SCALE**, then added to **GL_BLUE_BIAS**; and the alpha component is multiplied by **GL_ALPHA_SCALE**, 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, 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**, 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**, 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**, 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**, 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 the **glPixelMap** subroutine.

color index	Each color index is shifted left by **GL_INDEX_SHIFT** bits, and any bits beyond the number of fraction bits carried by the fixed-point index are filled with 0s. If **GL_INDEX_SHIFT** is negative, the shift is to the right, again 0 filled. Then **GL_INDEX_OFFSET** is 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, each index is masked by 2n-1, where *n* is **GL_PIXEL_MAP_I_TO_I_SIZE**, 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 the **glPixelMap** subroutine. If the resulting pixels are to be written to a red, green, blue, alpha (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 2n-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 the **glPixelMap** subroutine.
depth	Each depth value is multiplied by **GL_DEPTH_SCALE**, added to **GL_DEPTH_BIAS**, then clamped to the range [0,1].
stencil	Each index is shifted **GL_INDEX_SHIFT** bits just as a color index is, then added to **GL_INDEX_OFFSET**. If **GL_MAP_STENCIL** is True, each index is masked by 2n-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 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 or False
GL_MAP_STENCIL	Boolean	False	True or False
GL_INDEX_SHIFT	integer	0	(-infinity,+infinity)
GL_INDEX_OFFSET	integer	0	(-infinity,+infinity)
GL_RED_SCALE	float	1.0	(-infinity,+infinity)
GL_GREEN_SCALE	float	1.0	(-infinity,+infinity)
GL_BLUE_SCALE	float	1.0	(-infinity,+infinity)
GL_ALPHA_SCALE	float	1.0	(-infinity,+infinity)
GL_DEPTH_SCALE	float	1.0	(-infinity,+infinity)
GL_RED_BIAS	float	0.0	(-infinity,+infinity)
GL_GREEN_BIAS	float	0.0	(-infinity,+infinity)
GL_BLUE_BIAS	float	0.0	(-infinity,+infinity)
GL_ALPHA_BIAS	float	0.0	(-infinity,+infinity)
GL_DEPTH_BIAS	float	0.0	(-infinity,+infinity)

The **glPixelTransferf** subroutine 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, *Parameter* is rounded to the nearest integer.

Likewise, **glPixelTransferi** can be used to set any of the pixel transfer parameters. Boolean parameters are set to False if *Parameter* is 0 and True otherwise. *Parameter* is converted to floating-point format before being assigned to real-valued parameters.

Parameters

pName	Specifies the symbolic name of the pixel transfer parameter to be set. Must be one of the following: • **GL_MAP_COLOR** • **GL_MAP_STENCIL** • **GL_INDEX_SHIFT** • **GL_INDEX_OFFSET** • **GL_RED_SCALE** • **GL_RED_BIAS** • **GL_GREEN_SCALE** • **GL_GREEN_BIAS** • **GL_BLUE_SCALE** • **GL_BLUE_BIAS** • **GL_ALPHA_SCALE** • **GL_ALPHA_BIAS** • **GL_DEPTH_SCALE** • **GL_DEPTH_BIAS**
Parameter	Specifies the value to which *pName* is set.

Notes

If a **glDrawPixels**, **glReadPixels**, **glCopyPixels**, **glTexImage1D**, or **glTexImage2D** subroutine is placed in a display list (see the **glNewList** subroutine and the **glCallList** subroutine for information about display lists), 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.

Errors

GL_INVALID_ENUM	*pName* is not an accepted value.
GL_INVALID_OPERATION	The **glPixelTransfer** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glPixelTransfer** subroutine are as follows. (See the **glGet** subroutine for more information.)

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**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glCallList** subroutine, **glCopyPixels** subroutine, **glCopyTexImage1D** subroutine, **glCopyTexImage2D** subroutine, **glCopyTexSubImage1D** subroutine, **glCopyTexSubImage2D** subroutine, **glCopyTexSubImage3DEXT** subroutine, **glDrawPixels** subroutine, **glNewList** subroutine, **glPixelMap** subroutine, **glPixelStore** subroutine, **glPixelZoom** subroutine, **glReadPixels** subroutine, **glTexImage1D** subroutine, **glTexImage2D** subroutine, **glTexImage3DEXT** subroutine, **glTexSubImage1D** subroutine, **glTexSubImage2D** subroutine, **glTexSubImage3DEXT** subroutine, .

---

glPixelZoom Subroutine

Purpose

Specifies the pixel zoom factors.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glPixelZoom(GLfloat xFactor,
    GLfloat yFactor)
```

Parameters

xFactor and *yFactor*	Specify the *x* and *y* zoom factors for pixel write operations.

Description

The **glPixelZoom** subroutine specifies values for the *x* and *y* zoom factors. During the execution of the **glDrawPixels** or **glCopyPixels** subroutines, if (*xr*, *yr*) 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

(*xr* + *n* **x** *xFactor*, *yr* + *m* **x** *yFactor*) and

(xr + (n+1) **x** $xFactor$, yr + (m+1) **x** $yFactor$)

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.

Errors

GL_INVALID_OPERATION	The **glPixelZoom** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glPixelZoom** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_ZOOM_X**.

glGet with argument **GL_ZOOM_Y**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glCopyPixels** subroutine, **glDrawPixels** subroutine.

glPointSize Subroutine

Purpose

Specifies the diameter of rasterized points.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glPointSize(GLfloat  Size)
```

Description

The **glPointSize** subroutine 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 the **glEnable** and **glDisable** subroutines 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 (zero), it is as if the point size were 1 (one).) If the rounded size is odd, the center point (x, y) of the pixel fragment that represents the point is computed as

```
(floor(xw) + 0.5, floor(yw) + 0.5)
```

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

```
(floor(xw + 0.5), floor(yw + 0.5))
```

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, 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 point's (*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 are dependent on the implementation. The range of supported sizes and the size difference between supported sizes within the range can be queried by calling the **glGet** subroutine with the **GL_POINT_SIZE_RANGE** and **GL_POINT_SIZE_GRANULARITY** arguments.

Notes

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.

Nonantialiased point size may be clamped to a maximum that depends on the implementation. Although this maximum cannot be queried, it must be no less than the maximum value for antialiased points, rounded to the nearest integer value.

Parameters

Size Specifies the diameter of rasterized points. The default is 1.0.

Errors

GL_INVALID_VALUE	*Size* is less than or equal to 0.
GL_INVALID_OPERATION	The **glPointSize** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glPointSize** subroutine are as follows. (See the **glGet** subroutine for more information.)

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**.

Files

/usr/include/GL/gl.h Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glEnable** or **Disable** subroutine.

glPolygonMode Subroutine

Purpose

Selects a polygon rasterization mode.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glPolygonMode(GLenum  Face,
     GLenum  Mode)
```

Description

The **glPolygonMode** subroutine controls the interpretation of polygons for rasterization. The *Face* parameter describes which polygons the *Mode* parameters applies to: frontfacing polygons (**GL_FRONT**), backfacing 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.

Three modes are defined and can be specified in the *Mode* parameter:

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 the **glLineStipple** subroutine for information on specifying the line stipple pattern.) 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.

Parameters

Face Specifies the polygons to which *Mode* applies. Must be **GL_FRONT** for frontfacing polygons, **GL_BACK** for backfacing polygons, or **GL_FRONT_AND_BACK** for frontfacing and backfacing polygons.

Mode Specifies the way polygons are rasterized. Accepted values are **GL_POINT**, **GL_LINE**, and **GL_FILL**. The default is **GL_FILL** for both frontfacing and backfacing polygons.

Notes

Vertices are marked as boundary or nonboundary with an edge flag. Edge flags are generated internally by the GL when it decomposes polygons, and they can be set explicitly with the **glEdgeFlag** subroutine.

Errors

GL_INVALID_ENUM	*Face* or *Mode* is not an accepted value.
GL_INVALID_OPERATION	The **glPolygonMode** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glPolygonMode** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_POLYGON_MODE**.

Examples

To draw a surface with filled backfacing polygons and outlined frontfacing polygons, enter the following:

```
glPolygonMode(GL_FRONT, GL_LINE);
```

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glEdgeFlag** subroutine, **glLineStipple** subroutine, **glLineWidth** subroutine, **glPointSize** subroutine, **glPolygonStipple** subroutine.

glPolygonOffset Subroutine

Purpose

Sets the scale and bias used to calculate depth values.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glPolygonOffset(GLfloat  factor,
     GLfloat  units)
```

Description

When **GL_POLYGON_OFFSET** is enabled, each fragment's depth value will be offset after it is interpolated from the depth values of the appropriate vertices. The value of the offset is factor * DZ + r * units, where DZ is a measurement of the change in depth relative to the screen area of the polygon, and r is the smallest value which is guaranteed to produce a resolveable offset for a given implementation. The offset is added before the depth test is performed and before the value is written into the depth buffer.

This is useful for rendering hidden line images, for applying decals to surfaces, and for rendering solids with highlighted edges.

Parameters

factor	Specifies a scale factor which is used to create a variable depth offset for each polygon. The initial value is 0.
units	Is multiplied by an implementation specific value to create a constant depth offset. The initial value is 0.

Notes

The **glPolygonOffset** subroutine is available only if the GL version is 1.1 or greater.

Errors

GL_INVALID_OPERATION is generated if **glPolygonOffset** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glIsEnabled with argument **GL_POLYGON_OFFSET_FILL**, **GL_POLYGON_OFFSET_LINE**, or **GL_POLYGON_OFFSET_POINT**.

glGet with argument **GL_POLYGON_OFFSET_FACTOR** or **GL_POLYGON_OFFSET_UNITS**.

Related Information

The **glDepthFunc** subroutine, **glDisable** subroutine, **glEnable** subroutine, **glGet** subroutine, **glIsEnabled** subroutine, **glLineWidth** subroutine, **glStencilOp** subroutine, **glTexEnv** subroutine.

glPolygonOffsetEXT Subroutine

Purpose

Sets the scale and bias used to calculate z values.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glPolygonOffsetEXT(GLfloat  factor,
     GLfloat  bias)
```

Description

When **GL_POLYGON_OFFSET_EXT** is enabled, each fragment's *z* value will be offset after it is interpolated from the *z* values of the appropriate vertices. The value of the offset is *factor* * DZ + *bias*, where DZ is a measurement of the change in *z* relative to the screen area of the polygon. The offset is added before the Depth Test is performed and before the value is written into the Depth Buffer.

Initially **GL_POLYGON_OFFSET_FACTOR_EXT** and **GL_POLYGON_OFFSET_BIAS_EXT** are both set to 0.0.

This is useful for rendering hidden line images, for applying decals to surfaces, and for rendering solids with highlighted edges.

Parameters

factor	specifies a scale factor which is used to create a offset for each polygon.

bias	specifies a constant which is added to each polygon's *z* offset.

Notes

glPolygonOffsetEXT is part of the **EXT_polygon_offset** extension, not part of the core GL command set. If **GL_EXT_polygon_offset** is included in the string returned by **glGetString**, when called with argument **GL_EXTENSIONS**, extension **EXT_polygon_offset** is supported by the connection.

Errors

GL_INVALID_OPERATION is generated if **glPolygonOffsetEXT** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glIsEnabled with argument **GL_POLYGON_OFFSET_EXT**.

glGet with argument **GL_POLYGON_OFFSET_FACTOR_EXT** or **GL_POLYGON_OFFSET_BIAS_EXT** .

File

/usr/include/GL/glext.h	Contains extensions to C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glDepthFunc** subroutine, **glEnable** or**glDisable** subroutine, **glGet** subroutine, **glIsEnabled** subroutine, **glLineWidth** subroutine, **glStencilOp** subroutine, **glTexEnv** subroutine.

glPolygonStipple Subroutine

Purpose

Sets the polygon stippling pattern.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glPolygonStipple(const GLubyte * Mask)
```

Description

Polygon stippling, like line stippling, masks out certain fragments produced by rasterization, creating a pattern. (See the **glLineStipple** subroutine .) Stippling is independent of polygon antialiasing.

The *Mask* parameter is a pointer to a *32 x 32* stipple pattern that is stored in memory just like the pixel data supplied to a **glDrawPixels** subroutine 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 32 x 32 array of 1-bit color indexes packed in unsigned bytes. The **glPixelStore** subroutine 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, pixel map) are not applied to the stipple image, however.

Polygon stippling is enabled and disabled with the **glEnable**/**glDisable** subroutine pair, 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 the GL if and only if the (*x*w mod 32)th bit in the (*y*w mod 32)th row of the stipple pattern is 1 (one). When polygon stippling is disabled, it is as if the stipple pattern were all 1s.

Parameters

Mask	Specifies a pointer to a 32 x 32 stipple pattern that is unpacked from memory in the same way that the **glDrawPixels** subroutine unpacks pixels.

Associated Gets

Associated gets for the **glPolygonStipple** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGetPolygonStipple

glIsEnabled with argument **GL_POLYGON_STIPPLE**.

Error Codes

GL_INVALID_OPERATION	The **glPolygonStipple** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glDrawPixels** subroutine, **glEnable** or **glDisable** subroutine, **glGetPolygonStipple** subroutine, **glLineStipple** subroutine, **glPixelStore** subroutine, **glPixelTransfer** subroutine.

glPrioritizeTextures Subroutine

Purpose

Sets texture residence priority.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glPrioritizeTextures(GLsizei n,
                          const GLuint *textures,
                          const GLclampf *priorities)
```

Parameters

n	Specifies the number of textures to be prioritized.
textures	Specifies an array containing the names of the textures to be prioritized.

priorities	Specifies an array containing the texture priorities. A priority given in an element of priorities applies to the texture named by the corresponding element of textures.

Description

The **glPrioritizeTextures** subroutine assigns the *n* texture priorities given in priorities to the *n* textures named in *textures*.

On machines with a limited amount of texture memory, GL establishes a ``working set'' of textures that are resident in texture memory. These textures may be bound to a texture target much more efficiently than textures that are not resident. By specifying a priority for each texture, **glPrioritizeTextures** allows applications to guide the GL implementation in determining which *textures* should be resident.

The priorities given in *priorities* are clamped to the range [0.0, 1.0] before being assigned. Zero indicates the lowest priority; *textures* with priority zero are least likely to be resident. One indicates the highest priority; *textures* with priority one are most likely to be resident. However, *textures* are not guaranteed to be resident until they are bound.

The **glPrioritizeTextures** subroutine silently ignores attempts to prioritize texture zero, or any texture name that does not correspond to an existing texture.

The **glPrioritizeTextures** subroutine does not require that any of the textures named by *textures* be bound to a texture target. It can also be used to set the priority of a texture, but only if the texture is currently bound. This is the only way to set the priority of a default texture.

The **glPrioritizeTextures** subroutine is included in display lists.

Notes

The **glPrioritizeTextures** subroutine is available only if the GL version is 1.1 or greater.

Errors

GL_INVALID_VALUE is generated if n is negative.

GL_INVALID_OPERATION is generated if **glPrioritizeTextures** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glGetTexParameter with parameter name **GL_TEXTURE_PRIORITY** retrieves the priority of a currently bound texture.

Related Information

The **glAreTexturesResident** subroutine, **glBindTexture** subroutine, **glTexImage1D** subroutine, **glTexImage2D** subroutine, **glTexImage3DEXT** subroutine, **glTexParameter** subroutine.

glPrioritizeTexturesEXT Subroutine

Purpose

Sets texture residence priority.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glPrioritizeTexturesEXT(GLsizei n,
                    const GLuint *textures,
                    const GLclampf *priorities)
```

Parameters

n	The number of textures to be prioritized.
textures	An array containing the names of the textures to be prioritized.
priorities	An array containing the texture priorities. A priority given in an element of *priorities* applies to the texture named by the corresponding element of *textures*.

Description

glPrioritizeTexturesEXT assigns the *n* texture priorities given in *priorities* to the *n* textures named in *textures*.

On machines with a limited amount of texture memory, OpenGL establishes a ``working set'' of textures that are resident in texture memory. These textures may be bound to a texture target much more efficiently than textures that are not resident. By specifying a priority for each texture, **glPrioritizeTexturesEXT** allows applications to guide the OpenGL implementation in determining which textures should be resident.

The priorities given in *priorities* are clamped to the range [0.0, 1.0] before being assigned. Zero indicates the lowest priority, and hence textures with priority zero are least likely to be resident. One indicates the highest priority, and hence textures with priority one are most likely to be resident. However, textures are not guaranteed to be resident until they are bound.

glPrioritizeTexturesEXT silently ignores attempts to prioritize texture zero, or any texture name that does not correspond to an existing texture.

glPrioritizeTexturesEXT does not require that any of the textures named by *textures* be bound to a texture target. **glTexParameter** may also be used to set a texture's priority, but only if the texture is currently bound. This is the only way to set the priority of a default texture.

glPrioritizeTexturesEXT is included in display lists.

Notes

glPrioritizeTexturesEXT is part of the **EXT_texture_object** extension, not part of the core GL command set. If **GL_EXT_texture_object** is included in the string returned by **glGetString**, when called with argument **GL_EXTENSIONS**, extension **EXT_texture_object** is supported by the connection.

Errors

GL_INVALID_VALUE is generated if *n* is negative.

GL_INVALID_OPERATION is generated if **glPrioritizeTexturesEXT** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glGetTexParameter with parameter name **GL_TEXTURE_PRIORITY_EXT** retrieves the priority of a currently-bound texture.

File

/usr/include/GL/glext.h	Contains extensions to C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glAreTexturesResidentEXT** subroutine, **glBindTextureEXT** subroutine, **glTexImage1D** subroutine, **glTexImage2D** subroutine, **glTexImage3DEXT** subroutine, **glTexParameter** subroutine.

glPushAttrib or glPopAttrib Subroutine

Purpose

Pushes and pops the attribute stack.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glPushAttrib(GLbitfield mask)
void glPopAttrib(void)
```

Parameters

mask	Specifies a mask that indicates which attributes to save. Values for *Mask* are provided in the preceding list.

Description

The **glPushAttrib** subroutine 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 ORing several of these constants together. The **GL_ALL_ATTRIB_BITS** special mask can be used to save all stackable states.

The symbolic mask constants and their associated GL states are in the following list.

Mask	Attributes 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_COLOR_LOGIC_OP** enable bit **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 write masks **GL_BLEND_EQUATION_EXT** setting
GL_CURRENT_BIT	Current red, green, blue, alpha (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_LOGIC_OP flag
	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_OFFSET_EXT flag
	GL_POLYGON_OFFSET_FILL flag
	GL_POLYGON_OFFSET_LINE flag
	GL_POLYGON_OFFSET_POINT 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
	GL_TEXTURE_3D_EXT 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-dimensional (1D) grid endpoints and divisions
	2-dimensional (2D) 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_SUBPIXEL_HINT_IBM 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 indices 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_OFFSET_BIAS_EXT setting
	GL_POLYGON_OFFSET_EXT flag
	GL_POLYGON_OFFSET_FACTOR setting
	GL_POLYGON_OFFSET_FACTOR_EXT setting
	GL_POLYGON_OFFSET_FILL flag
	GL_POLYGON_OFFSET_LINE flag
	GL_POLYGON_OFFSET_POINT flag
	GL_POLYGON_OFFSET_UNITS setting
	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* Enables for 1D, 2D, and 3D_EXT testures
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

The **glPopAttrib** subroutine restores the values of the state variables saved with the last **glPushAttrib** subroutine. 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 GL state.

Initially, the attribute stack is empty.

Notes

Not all values for the GL 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 is dependent on the implementation, but it must be at least 16.

Errors

GL_STACK_OVERFLOW	The **glPushAttrib** subroutine is called while the attribute stack is full.
GL_STACK_UNDERFLOW	The **glPopAttrib** subroutine is called while the attribute stack is empty.
GL_INVALID_OPERATION	The **glPushAttrib** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

glGet with argument **GL_ATTRIB_STACK_DEPTH**

glGet with argument **GL_MAX_ATTRIB_STACK_DEPTH**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** subroutine, **glEnd** subroutine, **glGet** subroutine, **glGetClipPlane** subroutine, **glGetError** subroutine, **glGetLight** subroutine, **glGetMap** subroutine, **glGetMaterial** subroutine, **glGetPixelMap** subroutine, **glGetPolygonStipple** subroutine, **glGetString** subroutine, **glGetTexEnv** subroutine, **glGetTexGen** subroutine, **glGetTexImage** subroutine, **glGetTexLevelParameter** subroutine, **glGetTexParameter** subroutine, **glIsEnabled** subroutine, **glPushClientAttrib** or **PopClientAttrib** subroutine.

glPushClientAttrib or glPopClientAttrib Subroutine

Purpose

Pushes and pops the attribute stack.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glPushClientAttrib(GLbitfield mask)
void glPopClientAttrib(void)
```

Parameters

mask Specifies a mask that indicates which attributes to save. Values for *mask* are listed below.

Description

The **glPushClientAttrib** subroutine takes one argument, a mask that indicates which groups of client state variables to save on the client 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_CLIENT_ALL_ATTRIB_BITS** can be used to save all stackable client state.

The symbolic mask constants and their associated GL client state are as follows (the second column lists which attributes are saved):

GL_CLIENT_PIXEL_STORE_BIT	Pixel storage modes
GL_CLIENT_VERTEX_ARRAY_BIT	Vertex arrays (and enables)

The **glPopClientAttrib** subroutine restores the values of the client state variables saved with the last **glPushClientAttrib**. Those not * saved are left unchanged.

It is an error to push attributes onto a full client attribute stack, or to pop attributes off an empty stack. In either case, the error flag is set, and no other change is made to GL state.

Initially, the client attribute stack is empty.

Notes

The **glPushClientAttrib** subroutine is available only if the GL version is 1.1 or greater.

Not all values for GL client state can be saved on the attribute stack. For example, 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 **glPushClientAttrib** and **glPopClientAttrib** subroutines are not compiled

into display lists, but are executed immediately.

Use **glPushAttrib** and **glPopAttrib** to push and restore state which is kept on the server. Only pixel storage modes and vertex array state may be pushed and popped with **glPushClientAttrib** and **glPopClientAttrib**.

Errors

GL_STACK_OVERFLOW is generated if **glPushClientAttrib** is called while the attribute *stack* is full.

GL_STACK_UNDERFLOW is generated if **glPopClientAttrib** is called while the attribute *stack* is empty.

Associated Gets

glGet with argument **GL_ATTRIB_STACK_DEPTH**

glGet with argument **GL_MAX_CLIENT_ATTRIB_STACK_DEPTH**

Related Information

The **glColorPointer** subroutine, **glDisableClientState** subroutine, **glEdgeFlagPointer** subroutine, **glEnableClientState** subroutine, **glGet** subroutine, **glGetError** subroutine, **glIndexPointer** subroutine, **glNewList** subroutine, **glNormalPointer** subroutine, **glPixelStore** subroutine, **glPushAttrib** subroutine, **glTexCoordPointer** subroutine, **glVertexPointer** subroutine.

---

glPushMatrix or glPopMatrix Subroutine

Purpose

Pushes and pops the current matrix stack.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glPushMatrix(void)
void glPopMatrix(void)
```

Description

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** subroutine 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 **glPopMatrix** subroutine 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.

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 GL state.

Error Codes

GL_STACK_OVERFLOW	The **glPushMatrix** subroutine is called while the current matrix stack is full.
GL_STACK_UNDERFLOW	The **glPopMatrix** subroutine is called while the current matrix stack contains only a single matrix.
GL_INVALID_OPERATION	The **glPushMatrix** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glPushMatrix** or **glPopMatrix** subroutine are as follows. (See the **glGet** subroutine for more information.)

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**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glFrustum** subroutine, **glLoadIdentity** subroutine, **glLoadMatrix** subroutine, **glMatrixMode** subroutine, **glMultMatrix** subroutine, **glOrtho** subroutine, **glRotate** subroutine, **glScale** subroutine, **glTranslate** subroutine, **glViewport** subroutine.

glPushName or glPopName Subroutine

Purpose

Pushes and pops the name stack.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glPushName(GLuint Name)
void glPopName(void)
```

Parameters

Name Specifies a name that will be pushed onto the name stack.

Description

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 **glPushName** subroutine causes the *Name* parameter to be pushed onto the name stack, which is initially empty. The **glPopName** subroutine pops one name off the top of the stack.

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 the **glBegin** subroutine and the corresponding call to the **glEnd** subroutine. In any of these cases, the error flag is set and no other change is made to GL state.

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.

Associated Gets

Associated gets for the **glPushName** or **glPopName** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_NAME_STACK_DEPTH**

glGet with argument **GL_MAX_NAME_STACK_DEPTH**.

Error Codes

GL_STACK_OVERFLOW	The **glPushName** subroutine is called while the name stack is full.
GL_STACK_UNDERFLOW	The **glPopName** subroutine is called while the name stack is empty.
GL_INVALID_OPERATION	The **glPushName** or **glPopName** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glInitNames** subroutine, **glLoadName** subroutine, **glRenderMode** subroutine, **glSelectBuffer** subroutine.

---

glRasterPos Subroutine

Purpose

Specifies the raster position for pixel operations.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
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)

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

X, *Y*, *Z*, *W*	Specify the *x*, *y*, *z*, and *w* object coordinates (if present) for the raster position.
V	Specifies a pointer to an array of two, three, or four elements, specifying *x*, *y*, *z*, and *w* coordinates, respectively.

Description

The GL maintains a 3-dimensional (3D) 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 the **glBitmap** subroutine for information on drawing bitmaps; the **glCopyPixels** subroutine for information on copying pixels to the frame buffer; and the **glDrawPixels** subroutine for information on writing a block of pixels to the frame buffer.)

The current raster position consists of four window coordinates (*X*, *Y*, *Z*, *W*), a valid bit, and associated color data and texture coordinates. The *W* coordinate is actually a clip coordinate, because *W* is not projected to window coordinates. The **glRasterPos4** subroutine specifies object coordinates *X*, *Y*, *Z*, and *W* explicitly. The **glRasterPos3** subroutine specifies object coordinates *X*, *Y*, and *Z* explicitly, while *W* is implicitly set to 1 (one). The **glRasterPos2** subroutine uses the argument values for *X* and *Y* while implicitly setting *Z* and *W* to 0 (zero) and 1.

The object coordinates presented by **glRasterPos** are treated just like those of a **glVertex** subroutine: they are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, 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, 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, **GL_CURRENT_RASTER_COLOR** in red, green, blue, alpha (RGBA) mode or the **GL_CURRENT_RASTER_INDEX** in color index mode is set to the color produced by the lighting calculation. (See the **glLight** subroutine for information on setting light source parameters; the **glLightModel** subroutine for information on setting lighting model parameters; and the **glShadeModel** subroutine for information on selecting flat or smooth shading.) 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, the **GL_CURRENT_RASTER_TEXTURE_COORDS** is updated as a function of the **GL_CURRENT_TEXTURE_COORDS**, based on the texture matrix and the texture generation functions. (See the **glTexGen** subroutine for information on generating texture coordinates.)

Initially, the current raster position is (0,0,0,1), 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.

Notes

The raster position is modified both by **glRasterPos** and by **glBitmap**.

When the raster position coordinates are not valid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to GL state).

Errors

GL_INVALID_OPERATION	The **glRasterPos** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glRasterPos** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_CURRENT_RASTER_POSITION**

glGet with argument **GL_CURRENT_RASTER_POSITION_VALID**

glGet with argument **GL_CURRENT_RASTER_COLOR**

glGet with argument **GL_CURRENT_RASTER_INDEX**

glGet with argument **GL_CURRENT_RASTER_TEXTURE_COORDS**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glBitmap** subroutine, **glCopyPixels** subroutine, **glDrawPixels** subroutine, **glLight** subroutine, **glLightModel** subroutine, **glShadeModel** subroutine, **glTexCoord** subroutine, **glTexGen** subroutine, **glVertex** subroutine.

glReadBuffer Subroutine

Purpose

Selects a color buffer source for pixels.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glReadBuffer(GLenum Mode)
```

Parameters

Mode Specifies a color buffer. Accepted values are as follows:

- **GL_FRONT_LEFT**
- **GL_FRONT_RIGHT**
- **GL_BACK_LEFT**
- **GL_BACK_RIGHT**
- **GL_FRONT**, **GL_BACK**
- **GL_LEFT**
- **GL_RIGHT**
- **GL_AUX***i*, where *i* is between 0 (zero) and **GL_AUX_BUFFERS** - 1

Description

The **glReadBuffer** subroutine specifies a color buffer as the source for subsequent **glReadPixels** and **glCopyPixels** subroutines. 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 configurations have only a left buffer, or a front left and a back left buffer if double-buffered. Single-buffered configurations have only a front buffer, or a front left and a front right buffer if stereo. It is an error to specify a nonexistent buffer to **glReadBuffer**.

By default, the *Mode* parameter is **GL_FRONT** in single-buffered configurations and **GL_BACK** in double-buffered configurations.

Error Codes

GL_INVALID_ENUM	*Mode* is not one of the twelve (or more) accepted values.
GL_INVALID_OPERATION	*Mode* specifies a buffer that does not exist.
GL_INVALID_OPERATION	The **glReadBuffer** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glReadBuffer** subroutine are as follows. (See the **glGet** subroutine.)

glGet with argument **GL_READ_BUFFER**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glCopyPixels** subroutine, **glDrawBuffer** subroutine, **glReadPixels** subroutine.

glReadPixels Subroutine

Purpose

Reads a block of pixels from the frame buffer.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glReadPixels(GLint X,
                  GLint Y,
                  GLsizei Width,
                  GLsizei Height,
                  GLenum Format,
                  GLenum Type,
                  GLvoid *Pixels)
```

Parameters

X, *Y*	Specify 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*	Specify the dimensions of the pixel rectangle. *Width* and *Height* of 1 (one) correspond to a single pixel.
Format	Specifies the format of the pixel data. Symbolic constants **GL_COLOR_INDEX**, **GL_STENCIL_INDEX**, **GL_DEPTH_COMPONENT**, **GL_RED**, **GL_GREEN**, **GL_BLUE**, **GL_ALPHA**, **GL_RGB**, **GL_RGBA**, **GL_BGR**, **GL_BGRA**, **GL_ABGR_EXT**, **GL_LUMINANCE**, **GL_LUMINANCE_ALPHA**, **GL_422_EXT**, **GL_422_REV_EXT**, **GL_422_AVERAGE_EXT** and **GL_422_REV_AVERAGE_EXT** are accepted.
Type	Specifies the data type for *Pixels*. Sybolic constants **GL_UNSIGNED_BYTE**, **GL_BYTE**, **GL_BITMAP**, **GL_UNSIGNED_SHORT**, **GL_SHORT**, **GL_UNSIGNED_INT**, **GL_INT**, **GL_FLOAT**, **GL_UNSIGNED_BYTE_3_3_2**, **GL_UNSIGNED_BYTE_2_3_3_REV**, **GL_UNSIGNED_SHORT_5_6_5**, **GL_UNSIGNED_SHORT_5_6_5_REV**, **GL_UNSIGNED_SHORT_4_4_4_4**, **GL_UNSIGNED_SHORT_4_4_4_4_REV**, **GL_UNSIGNED_SHORT_5_5_5_1**, **GL_UNSIGNED_SHORT_1_5_5_5_REV**, **GL_UNSIGNED_INT_8_8_8_8**, **GL_UNSIGNED_INT_8_8_8_8_REV**, **GL_UNSIGNED_INT_10_10_10_2**, **GL_UNSIGNED_INT_2_10_10_10_REV** are accepted.
Pixels	Returns the pixel data.

Description

The **glReadPixels** subroutine returns pixel data from the frame buffer, starting with the pixel whose lower left corner is at location (*X*, *Y*), and puts it into client memory starting at the location specified by the *Pixels* parameter. Several parameters control the processing of the pixel data before it is placed into client memory. These parameters are set with three subroutines: **glPixelStore**, **glPixelTransfer**, and **glPixelMap**. The effects on **glReadPixels** of most, but not all, of the parameters specified by these three subroutines are described here.

The **glReadPixels** subroutine returns values from each pixel with the lower left-hand corner at ($x + i$, $y + j$) for $0 < i < Width$ and $0 < j < Height$. 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 *Format* parameter specifies the format for the returned pixel values. Accepted values for *Format* are as follows:

GL_COLOR_INDEX	Color indexes are read from the color buffer selected by the **glReadBuffer** subroutine. Each index is converted to fixed-point format, 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 format, 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 format 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_ABGR_EXT	Each pixel is a four-component group: for **GL_RGBA**, the red component is first, followed by green, followed by blue, followed by alpha; for **GL_BGRA**, the blue component is first, followed by green, followed by red, followed by alpha; for **GL_ABGR_EXT** the order is alpha, blue, green, and then red. Floating-point values are converted directly to an internal floatingpoint 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. 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. The results are clamped to the range [0,1].
GL_RED	Each pixel is a single red component. This component is converted to the internal floating-point format in the same way as the red component of an RGBA pixel is, then it is converted 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. This component is converted to the internal floating-point format in the same way as the green component of an RGBA pixel is, then it is converted 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. This component is converted to the internal floating-point format in the same way as the blue component of an RGBA pixel is, then it is converted 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. This component is converted to the internal floating-point format in the same way as the alpha component of an RGBA pixel is, then it is converted 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 three-component group, red first, followed by green, followed by blue. Each component is converted to the internal floating-point format in the same way as 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_RGBA
Each pixel is a four-component group, red first, followed by green, followed by blue, followed by alpha. Floating-point values are converted 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 are mapped similarly: the largest integer value maps to 1.0, and 0 maps to 0.0. 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. The results are clamped to the range [0,1].

If **GL_MAP_COLOR** is True, each color component is scaled by the size of the lookup table **GL_PIXEL_MAP_*c*_TO_*c***, then replaced by the value that it references in that table. *c* is **R**, **G**, **B**, or **A**, respectively.

The resulting RGBA colors are then converted 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* mod *Width* and *y*n = *y*r + [*n/Width*], where (*x*r, *y*r) is the current raster position. 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_BGR
Each pixel is a three-component group, blue first, followed by green, followed by red. Each component is converted to the internal floating-point format in the same way as the blue, green, and red components of an BGRA pixel are. The color triple is converted to an BGRA pixel with alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an BGRA pixel.

GL_BGRA
Each pixel is a four-component group, blue first, followed by green, followed by red, followed by alpha. Floating-point values are converted 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 are mapped similarly: the largest integer value maps to 1.0, and 0 maps to 0.0. The resulting floating-point color values are then multiplied by **GL_*c*_SCALE** and added to **GL_*c*_BIAS**, where *c* is **BLUE**, **GREEN**, **RED**, and **ALPHA** for the respective color components. The results are clamped to the range [0,1].

If **GL_MAP_COLOR** is True, each color component is scaled by the size of the lookup table **GL_PIXEL_MAP_*c*_TO_*c***, then replaced by the value that it references in that table. *c* is **B**, **G**, **R**, or **A**, respectively.

The resulting BGRA colors are then converted 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* mod *Width* and *y*n = *y*r + [*n/Width*], where (*x*r, *y*r) is the current raster position. 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_LUMINANCE
Each pixel is a single luminance component. This component is converted to the internal floating-point format in the same way as the red component of an RGBA pixel is, then it is converted 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

Processing differs depending on whether color buffers store color indexes or red, green, blue, alpha (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 format, 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 RGBA 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 format 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 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.

GL_422_EXT

This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_RGB_TO_YCBCR_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glReadPixels is called with this parameter. The internal RGB values are sent through the RGB_to_YCbCr matrix to create Y, Cb, and Cr values. Each returned pixel is a two-component group. The first component is Y. The second component is Cb in the even pixels and Cr in the odd pixels. The Cb for each even pixel comes from the Cb value for that pixel. The Cr in each odd pixel comes from the Cr value of its neighbor to the left. (If the width of the image is odd, then the colors will be undefined in the rightmost column.).

GL_422_REV_EXT

This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_RGB_TO_YCBCR_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glReadPixels is called with this parameter. The internal RGB values are sent through the RGB_to_YCbCr matrix to create Y, Cb, and Cr values. Each returned pixel is a two-component group. The first component is Cb in the even pixels and Cr in the odd pixels. The second component is Y. The Cb for each even pixel comes from the Cb value for that pixel. The Cr in each odd pixel comes from its neighbor to the left. (If the width of the image is odd, then the colors will be undefined in the rightmost column.).

GL_422_AVERAGE_EXT — This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_RGB_TO_YCBCR_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glReadPixels is called with this parameter. The internal RGB values are sent through the RGB_to_YCbCr matrix to create Y, Cb, and Cr values. Each returned pixel is a two-component group. The first component is Y. The second component is Cb in the even pixels and Cr in the odd pixels. Each even pixel gets its Cb from itself, and its Cr from its neighbor to the right. Each odd pixel gets its Cb from the average of its own Cb and that of its left neighbor, and gets its Cr from the average of its own Cr and that of its left neighbor. (If the width of the image is odd, then the colors will be undefined in the rightmost column. If the neighbors to the right are not present for a given fragment, we use GL_422_EXT to compute that fragment.).

GL_422_REV_AVERAGE_EXT — This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_RGB_TO_YCBCR_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glReadPixels is called with this parameter. The internal RGB values are sent through the RGB_to_YCbCr matrix to create Y, Cb, and Cr values. Each returned pixel is a two-component group. The first component is Cb in the even pixels and Cr in the odd pixels. The second component is Y. Each even pixel gets its Cb from itself, and its Cr from its neighbor to the right. Each odd pixel gets its Cb from the average of its own Cb and that of its left neighbor, and gets its Cr from the average of its own Cr and that of its left neighbor. (If the width of the image is odd, then the colors will be undefined in the rightmost column. If the neighbors to the right are not present for a given fragment, we use GL_422_EXT to compute that fragment.).

The shift, scale, bias, and lookup factors described in the preceding section are all specified by **glPixelTransfer**. The lookup table contents themselves are specified by the **glPixelMap** subroutine.

The final step involves converting the indexes or components to the proper format, as specified by the *Type* parameter. If the *Format* parameter 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 the *Type* parameter is **GL_FLOAT**, each integer index is converted to single-precision floating-point format.

If the *Format* parameter is any legal value other than **GL_COLOR_INDEX**, **GL_STENCIL_INDEX**, or **GL_DEPTH_COMPONENT**, and the *Type* parameter is not **GL_FLOAT**, each component is multiplied by the multiplier shown in the following table. If *Type* is **GL_FLOAT**, 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 the GL).

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
GL_UNSIGNED_BYTE_3_3_2	2^8 -1	$(2N -1)c$
GL_UNSIGNED_BYTE_2_3_3_REV	2^8 -1	$(2N -1)c$

Type	**Index Mask**	**Component Conversion**
GL_UNSIGNED_SHORT_5_6_5	2^{16} -1	$(2N -1)c$
GL_UNSIGNED_SHORT_5_6_5_REV	2^{16} -1	$(2N -1)c$
GL_UNSIGNED_SHORT_4_4_4_4	2^{16} -1	$(2N -1)c$
GL_UNSIGNED_SHORT_4_4_4_4_REV	2^{16} -1	$(2N -1)c$
GL_UNSIGNED_SHORT_5_5_5_1	2^{16} -1	$(2N -1)c$
GL_UNSIGNED_SHORT_1_5_5_5_REV	2^{16} -1	$(2N -1)c$
GL_UNSIGNED_INT_8_8_8_8	2^{32} -1	$(2N -1)c$
GL_UNSIGNED_INT_8_8_8_8_REV	2^{32} -1	$(2N -1)c$
GL_UNSIGNED_INT_10_10_10_2	2^{32} -1	$(2N -1)c$
GL_UNSIGNED_INT_2_10_10_10_REV	2^{32} -1	$(2N -1)c$

Equations with *N* as the exponent are performed for each bitfield of the packed data type, with *N* set to the number of bits in the bitfield.

Return values are placed in memory as follows. If the *Format* parameter 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** return three values, **GL_RGBA**, **GL_BGRA**, and **GL_ABGR_EXT** return four values, and **GL_LUMINANCE_ALPHA**, **GL_422_EXT**, **GL_422_REV_EXT**, **GL_422_AVERAGE_EXT** and **GL_422_REV_AVERAGE_EXT** return 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 the **glPixelStore** subroutine for a description.

Notes

Values for pixels that lie outside the window connected to the current GL context are undefined. If an error is generated, no change is made to the contents of *Pixels*.

Format of **GL_ABGR_EXT** is part of the _extname (EXT_abgr) extension, not part of the core GL command set.

Packed pixel types and BGR/BGRA formats are only supported in OpenGL 1.2 and later.

Error Codes

GL_INVALID_ENUM	*Format* or *Type* is not an accepted value.
GL_INVALID_VALUE	*Width* or *Height* is negative.
GL_INVALID_OPERATION	*Format* is **GL_COLOR_INDEX** and the color buffers store RGBA color components.
GL_INVALID_OPERATION	*Format* is **GL_STENCIL_INDEX** and there is no stencil buffer.
GL_INVALID_OPERATION	*Format* is **GL_DEPTH_COMPONENT** and there is no depth buffer.
GL_INVALID_OPERATION	The **glReadPixels** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glReadPixels** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_INDEX_MODE**.

Files

/usr/include/GL/gl.h Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glCopyPixels** subroutine, **glDrawPixels** subroutine, **glPixelMap** subroutine, **glPixelStore** subroutine, **glPixelTransfer** subroutine, **glReadBuffer** subroutine.

glRect Subroutine

Purpose

Draws a rectangle.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
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)

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

X1, *Y1*	Specify one vertex of a rectangle.
X2, *Y2*	Specify the opposite vertex of the rectangle.
V1	Specifies a pointer to one vertex of a rectangle.
V2	Specifies a pointer to the opposite vertex of the rectangle.

Description

The **glRect** subroutine 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.

`glRect(X1, Y1, X2, Y2)` is equivalent to the following sequence:

```
glBegin(GL_POLYGON);
glVertex2(X1, Y1);
glVertex2(X2, Y1);
glVertex2(X2, Y2);
glVertex2(X1, Y2);
glEnd();
```

> **Note:** If the second vertex is above and to the right of the first vertex, the rectangle is constructed with a counterclockwise winding.

Errors

GL_INVALID_OPERATION	The **glRect** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glVertex** subroutine.

---

glRenderMode Subroutine

Purpose

Sets rasterization mode.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

`GLint glRenderMode(GLenum` *`Mode`***`)`**

Parameters

Mode	Specifies the rasterization mode. Four values are accepted: **GL_RENDER**, **GL_SELECT**, **GL_FEEDBACK**, and **GL_VISIBILITY_IBM**. The default value is **GL_RENDER**.

Description

The **glRenderMode** subroutine sets the rasterization mode. It takes one argument, the *Mode* parameter, which can assume one of four predefined values:

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 before selection mode is entered. (See the **glSelectBuffer** subroutine for information about establishing a buffer for selection mode values.)
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 before feedback mode is entered. (See the **glFeedbackBuffer** subroutine for information about controlling the feedback mode.)
GL_VISIBILITY_IBM	*Visibility* RenderMode is identical to *render* RenderMode, except whenever a fragment passes all tests (in other words, depth, stencil, alpha, scissor and window-ownership) then a visibility hit results. Whenever a name stack manipulation command is executed or RenderMode is called, and there is a hit since the last time the stack was manipulated or RenderMode was called, then a hit record is written into the visibility array. The hit record consists of the number of names in the name stack at the time of the event, followed by the name stack contents (bottom name first). (See the **glVisibilityBufferIBM** subroutine for information about controlling the visibility mode.)

The return value of **glRenderMode** is determined by the render mode at the time **glRenderMode** is called, rather than by the *Mode* parameter.

Refer to **glSelectBuffer**, **glFeedbackBuffer** and **glVisibilityBufferIBM** for more details concerning selection, feedback and visibility operation.

Notes

If an error is generated, **glRenderMode** returns 0 (zero) regardless of the current render mode.

Errors

GL_INVALID_ENUM	*Mode* is not one of the four accepted values.
GL_INVALID_OPERATION	The **glSelectBuffer** subroutine is called while the render mode is **GL_SELECT**, or **glRenderMode** is called with the **GL_SELECT** argument before **glSelectBuffer** is called at least once.
GL_INVALID_OPERATION	The **glFeedbackBuffer** subroutine is called while the render mode is **GL_FEEDBACK**, or **glRenderMode** is called with the **GL_FEEDBACK** argument before **glFeedbackBuffer** is called at least once.
GL_INVALID_OPERATION	The **glRenderMode** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.
GL_INVALID_OPERATION	The **glVisibilityBufferIBM** subroutine is called while the render mode is **GL_VISIBILITY_IBM**, or **glRenderMode** is called with the **GL_VISIBILITY_IBM** argument before **glVisibilityBufferIBM** is called at least once.

Associated Gets

Associated gets for the **glRenderMode** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_RENDER_MODE**.

Return Values

GL_RENDER	0.
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.
GL_VISIBILITY_IBM	The number of hit records transferred to the visibility buffer.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glFeedbackBuffer** subroutine, **glVisibilityBufferIBM** subroutine, **glInitNames** subroutine, **glLoadName** subroutine, **glPassThrough** subroutine, **glPushName** subroutine, **glSelectBuffer** subroutine.

glRotate Subroutine

Purpose

Multiplies the current matrix by a rotation matrix.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glRotated(GLdouble Angle,
    GLdouble X,
    GLdouble Y,
    GLdouble Z)
```

```
void glRotatef(GLfloat Angle,
   GLfloat X,
   GLfloat Y,
   GLfloat Z)
```

Parameters

Angle	Specifies the angle of rotation, in degrees.
X, *Y*, *Z*	Specify the *X*, *Y*, and *Z* coordinates of a vector, respectively.

Description

The **glRotate** subroutine 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 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, M is replaced with MR. (See the **glMatrixMode** subroutine for information on specifying the current matrix.)

If the matrix mode is either **GL_MODELVIEW** or **GL_PROJECTION**, all objects drawn after **glRotate** is called are rotated. Use the **glPushMatrix** and **glPopMatrix** subroutines to save and restore the unrotated coordinate system.

Associated Gets

Associated gets for the **glRotate** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_MATRIX_MODE**

glGet with argument **GL_MODELVIEW_MATRIX**

glGet with argument **GL_PROJECTION_MATRIX**

glGet with argument **GL_TEXTURE_MATRIX**.

Errors

GL_INVALID_OPERATION	The **glRotate** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glMatrixMode** subroutine, **glMultMatrix** subroutine, **glPushMatrix** subroutine, **glScale** subroutine, **glTranslate** subroutine.

glScale Subroutine

Purpose

Multiplies the current matrix by a general scaling matrix.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glScaled(GLdouble X,
       GLdouble Y,
       GLdouble Z)
```

```
void glScalef(GLfloat X,
      GLfloat Y,
      GLfloat Z)
```

Parameters

X, *Y*, *Z*	Specify scale factors along the *X*, *Y*, and *Z* axes, respectively.

Description

The **glScale** subroutine 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 as follows:

$$\begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 22. Resulting Matrix. This diagram shows a matrix enclosed in brackets. The matrix consists of four lines containing four characters each. The first line contains the following (from left to right): x, zero, zero, zero. The second line contains the following (from left to right): zero, y, zero, zero. The third line contains the following (from left to right): zero, zero, z, zero. The fourth line contains the following (from left to right): zero, zero, zero, one.

The current matrix 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, M is replaced with MS. (See the **glMatrixMode** subroutine for information on specifying the current matrix.)

If the matrix mode is either **GL_MODELVIEW** or **GL_PROJECTION**, all objects drawn after **glScale** is called are scaled. Use the **glPushMatrix** and **glPopMatrix** subroutines to save and restore the unscaled coordinate system.

Notes

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. (Use the **glEnable** and **glDisable** subroutines with the **GL_NORMALIZE** argument.)

Errors

GL_INVALID_OPERATION	The **glScale** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glScale** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_MATRIX_MODE**

glGet with argument **GL_MODELVIEW_MATRIX**

glGet with argument **GL_PROJECTION_MATRIX**

glGet with argument **GL_TEXTURE_MATRIX**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glEnable** subroutine, **glMatrixMode** subroutine, **glMultMatrix** subroutine, **glPushMatrix** subroutine, **glRotate** subroutine, **glTranslate** subroutine.

glScissor Subroutine

Purpose

Defines the scissor box.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glScissor(GLint X,
    GLint Y,
    GLsizei Width,
    GLsizei Height)
```

When the scissor test is disabled, it is as though the scissor box includes the entire window.

Parameters

X, *Y*	Specify the lower left corner of the scissor box. Initially (0,0).
Width, *Height*	Specify the width and height of the scissor box. When a GL context is *first* attached to a window, *Width* and *Height* are set to the dimensions of that window.

Description

The **glScissor** subroutine defines a rectangle, called the scissor box, in window coordinates. The first two arguments, *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 with the **glEnable** and **glDisable** subroutines with the **GL_SCISSOR_TEST** argument. 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.

Errors

GL_INVALID_VALUE	*Width* or *Height* is negative.
GL_INVALID_OPERATION	The **glScissor** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glScissor** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_SCISSOR_BOX**

glIsEnabled with argument **GL_SCISSOR_TEST**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glEnable** or **glDisable** subroutine, **glViewport** subroutine.

glSecondaryColorEXT Subroutine

Purpose

Specifies an RGB color used by the Color Sum stage.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
void glSecondaryColorbEXT(GLbyte  Red,
                          GLbyte  Green,
                          GLbyte  Blue)
void glSecondaryColorsEXT(GLshort  Red,
                          GLshort  Green,
                          GLshort  Blue)
void glSecondaryColoriEXT(GLint  Red,
                          GLint  Green,
                          GLint  Blue)
void glSecondaryColorfEXT(GLfloat  Red,
                          GLfloat  Green,
                          GLfloat  Blue)
void glSecondaryColordEXT(GLdouble  Red,
                          GLdouble  Green,
                          GLdouble  Blue)
void glSecondaryColorubEXT(GLubyte  Red,
                           GLubyte  Green,
                           GLubyte  Blue)
void glSecondaryColorusEXT(GLushort  Red,
                           GLushort  Green,
                           GLushort  Blue)
void glSecondaryColoruiEXT(GLuint  Red,
                           GLuint  Green,
                           GLuint  Blue)
void glSecondaryColorbvEXT(GLbyte  *Variable)
void glSecondaryColorsvEXT(GLshort  *Variable)
void glSecondaryColorivEXT(GLint  *Variable)
```

```
void glSecondaryColorfvEXT(GLfloat  *Variable)
void glSecondaryColordvEXT(GLdouble  *Variable)
void glSecondaryColorubvEXT(GLubyte  *Variable)
void glSecondaryColorusvEXT(GLushort  *Variable)
void glSecondaryColoruivEXT(GLuint  *Variable)
```

Description

This extension allows specifying the RGB components of the secondary color used in the Color Sum stage, instead of using the default (0,0,0,0) color. It applies only in RGBA mode and when LIGHTING is disabled.

Secondary alpha is always implicitly set to 0.0.

After texturing, a fragment has two RGBA colors: a primary color c_pri (which texturing, if enabled, may have modified) and a secondary color c_sec.

If color sum is enabled, the components of these two colors are summed to produce a single post-texturing RGBA color c (the A component of the secondary color is always 0). The components of c are then clamped to the range [0,1]. If color sum is disabled, then c_pri is assigned to the post texturing color. Color sum is enabled or disabled using the generic Enable and Disable commands, respectively, with the symbolic constant **GL_COLOR_SUM_EXT**.

Parameters

Red,Green,Blue	Specify the red, green and blue values of the Secondary color.
Variable	Specifies a pointer to an array of three values. These are interpreted, respectively, as the *red*, *green* and *blue* values of the Secondary color.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

glSecondaryColorPointerEXT Subroutine

Purpose

Specifies an array of secondary colors.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
void glSecondaryColorPointerEXT(GLint  size,
                                GLenum  type,
                                GLsizei  stride,
                                const GLvoid  *pointer)
```

Description

The **glSecondaryColorPointerEXT** extension specifies the location and data format of an array of secondary color components to use when rendering. The *size* parameter specifies the number of components per color, and must be 3 or 4. The *type* parameter specifies the data type of each color component and *stride* gives the byte stride from one color to the next allowing vertices and attributes to be packed into a single array or stored in separate arrays. (Single-array storage may be more efficient on some implementations; see **glInterleavedArrays**).

When a secondary color array is specified, *size*, *type*, *stride*, and *pointer* are saved as client side state.

To enable and disable the secondary color array, call **glEnableClientState** and **glDisableClientState** with the argument **GL_SECONDARY_COLOR_ARRAY**. If enabled, the secondary color array is used when **glDrawArrays**, **glDrawElements** or **glArrayElement** is called.

Use **glDrawArrays**, **glMultiDrawArraysEXT**, or **glMultiModeDrawArraysIBM** to construct a sequence of primitives from prespecified vertex and vertex attribute arrays. Use **glArrayElement** to specify primitives by indexing vertices and vertex attributes. Use **glDrawElements**, **glMultiDrawElementsEXT**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

If enabled, the Secondary Color array is used when **glDrawArrays**, **glDrawElements**, **glArrayElements**, **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** is called.

Parameters

size	specifies the number of components per color. It must be 3 or 4. The initial value is 4.
type	specifies the data type of each color component in the array. Symbolic constants **GL_BYTE**, **GL_UNSIGNED_BYTE**, **GL_SHORT**, **GL_UNSIGNED_SHORT**, **GL_INT**, **GL_UNSIGNED_INT**, **GL_FLOAT**, or **GL_DOUBLE** are accepted. The initial value is **GL_FLOAT**.
stride	specifies the byte offset between consecutive colors. If *stride* is zero (the initial value), the colors are understood to be tightly packed in the array. The initial value is 0.
pointer	specifies a pointer to the first component of the first color element in the array. The initial value is 0 (NULL pointer).

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glArrayElement** subroutine, the **glColorPointer** subroutine, the **glDrawArrays** subroutine, the **glDrawElements** subroutine, the **glEdgeFlagPointer** subroutine, the **glEnable** subroutine, the **glGetPointerv** subroutine, the **glIndexPointer** subroutine, the **glInterleavedArrays** subroutine, the **glNormalPointer** subroutine, the **glPushClientAttrib** or **glPopClientAttrib** subroutine, the **glSecondaryColorPointerListIBM** subroutine, the **glTexCoordPointer** subroutine, the **glVertexPointer** subroutine.

glSecondaryColorPointerListIBM Subroutine

Purpose

Defines a list of arrays of secondary colors.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
void glColorPointerListIBM (GLint  size,
                            GLenum  type,
                            GLint  stride,
                            const GLvoid  **pointer,
                            GLint  ptrstride)
```

Description

The **glSecondaryColorPointerListIBM** subroutine specifies the location and data format of a list of arrays of color components to use when rendering. The *size* parameter specifies the number of components per color, and must be 3 or 4. The *type* parameter specifies the data type of each color component. The *stride* parameter gives the byte stride from one color to the next allowing vertices and attributes to be packed into a single array or stored in separate arrays. (Single-array storage may be more efficient on some implementations; see **glInterleavedArrays**). The *ptrstride* parameter specifies the byte stride from one pointer to the next in the pointer array.

When a secondary color array is specified, *size*, *type*, *stride*, *pointer* and *ptrstride* are saved as client side state.

A *stride* value of 0 does not specify a ″tightly packed″ array as it does in **glSecondaryColorPointer**. Instead, it causes the first array element of each array to be used for each vertex. Also, a negative value can be used for *stride*, which allows the user to move through each array in reverse order.

To enable and disable the secondary color arrays, call **glEnableClientState** and **glDisableClientState** with the argument **GL_COLOR_ARRAY**. The secondary color array is initially disabled. When enabled, the secondary color arrays are used when **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, **glDrawArrays**, **glDrawElements** or **glArrayElement** is called. The last three calls in this list will only use the first array (the one pointed at by pointer[0]). See the descriptions of these routines for more information on their use.

Use **glDrawArrays**, **glMultiDrawArraysEXT**, or **glMultiModeDrawArraysIBM** to construct a sequence of primitives from prespecified vertex and vertex attribute arrays. Use **glArrayElement** to specify primitives by indexing vertices and vertex attributes. Use **glDrawElements**, **glMultiDrawElementsEXT**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

If enabled, the Secondary Color array is used when **glDrawArrays**, **glDrawElements**, **glArrayElements**, **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** is called.

Parameters

size — specifies the number of components per secondary color. This must be 3 or 4. The initial value is 4.

type	specifies the data type of each secondary color component in the array. Symbolic constants **GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT,** or **GL_DOUBLE** are accepted. The initial value is **GL_FLOAT**.
stride	specifies the byte offset between consecutive secondary colors. The initial value is 0.
pointer	specifies a list of secondary color arrays. The initial value is 0 (NULL pointer).
ptrstride	specifies the byte stride between successive pointers in the pointer array. The initial value is 0.

Notes

The **glSecondaryColorPointerListIBM** subroutine is available only if the **GL_IBM_vertex_array_lists** extension is supported.

Execution of **glSecondaryColorPointerListIBM** is not allowed between **glBegin** and the corresponding **glEnd**, but an error may or may not be generated. If an error is not generated, the operation is undefined.

The **glSecondaryColorPointerListIBM** subroutine is typically implemented on the client side.

Since the secondary color array parameters are client side state, they are not saved or restored by **glPushAttrib** and **glPopAttrib**. Use **glPushClientAttrib** and **glPopClientAttrib** instead.

When a **glSecondaryColorPointerListIBM** call is encountered while compiling a display list, the information it contains does NOT contribute to the display list, but is used to update the immediate context instead.

The **glSecondaryColorPointer** call and the **glSecondaryColorPointerListIBM** call share the same state variables. A **glSecondaryColorPointer** call will reset the secondary color list state to indicate that there is only one secondary color list, so that any and all lists specified by a previous **glSecondaryColorPointerListIBM** call will be lost, not just the first list that it specified.

Error Codes

GL_INVALID_VALUE	is generated if *size* is not 3 or 4.
GL_INVALID_ENUM	is generated if *type* is not an accepted value.

Associated gets for the **glSecondaryColorPointerListIBM** subroutine are as follows. (See the **glGet** subroutine for more information.)

glIsEnabled with argument **GL_COLOR_ARRAY.**.

glGetPointerv with argument **GL_COLOR_ARRAY_LIST_IBM**.

glGet with argument **GL_COLOR_ARRAY_LIST_STRIDE_IBM**.

glGet with argument **GL_COLOR_ARRAY_SIZE**.

glGet with argument **GL_COLOR_ARRAY_STRIDE**.

glGet with argument **GL_COLOR_ARRAY_TYPE**.

Files

/usr/include/GL/gl.h Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glArrayElement** subroutine, the **glColorPointer** subroutine, the **glDrawArrays** subroutine, the **glDrawElements** subroutine, the **glEdgeFlagPointer** subroutine, the **glEnable** subroutine, the **glGetPointerv** subroutine, the **glIndexPointer** subroutine, the **glInterleavedArrays** subroutine, the **glMultiDrawArraysEXT** subroutine, the **glMultiDrawElementsEXT** subroutine, the **glMultiModeDrawArraysIBM** subroutine, the **glMultiModeDrawElementsIBM** subroutine, the **glNormalPointer** subroutine, the **glPushClientAttrib** or **glPopClientAttrib** subroutine, the **glTexCoordPointer** subroutine, the **glVertexPointer** subroutine.

glSelectBuffer Subroutine

Purpose

Establishes a buffer for selection mode values.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glSelectBuffer(GLsizei Size,
         GLuint *Buffer)
```

Parameters

Size Specifies the size of *Buffer*.
Buffer Returns the selection data.

Description

The **glSelectBuffer** subroutine has two arguments: the *Buffer* parameter is a pointer to an array of unsigned integers, and the *Size* parameter indicates the size of the array. *Buffer* returns values from the name stack when the rendering mode is **GL_SELECT**. (See the **glInitNames** subroutine for information on initializing the name stack; the **glLoadName** subroutine for information on loading names onto the name stack; the **glPushName** subroutine for pushing and popping the name stack; and the **glRenderMode** subroutine for information on setting the rasterization mode.) The **glSelectBuffer** subroutine 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 clipping 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 the **glRenderMode** subroutine is called, a hit record is copied to *Buffer* if any hits have occurred since the last such event (name stack change or **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 0 (zero) corresponds to window coordinate depth 0.0.

An internal index into *Buffer* is reset to 0 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 0 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.

Notes

The contents of *Buffer* are undefined until **glRenderMode** is called with an argument other than **GL_SELECT**.

The **glBegin**/**glEnd** subroutine primitives and calls to **glRasterPos** can result in hits.

Errors

GL_INVALID_VALUE	*Size* is negative.
GL_INVALID_OPERATION	The **glSelectBuffer** subroutine is called while the render mode is **GL_SELECT**, or **glRenderMode** is called with the **GL_SELECT** argument before **glSelectBuffer** is called at least once.
GL_INVALID_OPERATION	The **glSelectBuffer** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glSelectBuffer** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_NAME_STACK_DEPTH**.

glGetPointerv with argument **GL_SELECTION_BUFFER_POINTER**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glFeedbackBuffer** subroutine, **glGetPointerv** subroutine, **glInitNames** subroutine, **glLoadName** subroutine, **glPushName** subroutine, **glRenderMode** subroutine.

glShadeModel Subroutine

Purpose

Selects flat or smooth shading.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glShadeModel(GLenum Mode)
```

Parameters

Mode Specifies a symbolic value representing a shading technique. Accepted values are **GL_FLAT** and **GL_SMOOTH**. The default is **GL_SMOOTH**.

Description

GL 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 1 (one) starting when the **glBegin** subroutine is issued, each flat-shaded line segment *i* is given the computed color of vertex $i + 1$, its second vertex. Counting similarly from 1, each flat-shaded polygon is given the computed color of the vertex in the following list. 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 the *Mode* parameter set to **GL_FLAT** and **GL_SMOOTH**, respectively.

Errors

GL_INVALID_ENUM	*Mode* is any value other than **GL_FLAT** or **GL_SMOOTH**.
GL_INVALID_OPERATION	The **glShadeModel** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glShadeModel** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_SHADE_MODEL**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glColor** subroutine, **glLight** subroutine, **glLightModel** subroutine.

glStencilFunc Subroutine

Purpose

Sets function and reference values for stencil testing.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glStencilFunc(GLenum Function,
     GLint Reference,
     GLuint Mask)
```

Parameters

Function
Specifies the test function. Eight tokens are valid:

- **GL_NEVER**
- **GL_LESS**
- **GL_LEQUAL**
- **GL_GREATER**
- **GL_GEQUAL**
- **GL_EQUAL**
- **GL_NOTEQUAL**
- **GL_ALWAYS**

Reference
Specifies the reference value for the stencil test. *Reference* is clamped to the range [0,2n-1], where *n* is the number of bit planes in the stencil buffer.

Mask
Specifies a mask that is ANDed with both the reference value and the stored stencil value when the test is done.

Description

Stenciling, like z-buffering, enables and disables drawing on a per-pixel basis. You draw into the stencil planes using GL drawing primitives, and 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 the **glEnable** and **glDisable** subroutines with the **GL_STENCIL** argument. Actions taken based on the outcome of the stencil test are specified with the **glStencilOp** subroutine.

The *Function* parameter is a symbolic constant that determines the stencil comparison function. It accepts one of the eight following values. The *Reference* parameter is an integer reference value that is used in the stencil comparison. It is clamped to the range [0,2n-1], where *n* is the number of bit planes in the stencil buffer. The *Mask* parameter is bitwise ANDed with both the reference value and the stored stencil value, with the ANDed values participating in the comparison.

If *stencil* represents the value stored in the corresponding stencil buffer location, the following list shows the effect of each comparison function that can be specified by the *Function* parameter. Only if the comparison succeeds is the pixel passed through to the next stage in the rasterization process. (See the **glStencilOp** subroutine for information on setting stencil test actions.) All tests treat *stencil* values as unsigned integers in the range [0,2n-1], where *n* is the number of bit planes in the stencil buffer.

The following values are accepted by the *Function* parameter:

GL_NEVER	Always fails.
GL_LESS	Passes if (*Reference* & *Mask*) is less than (*stencil* & *Mask*).
GL_LEQUAL	Passes if (*Reference* & *Mask*) is less than or equal to (*stencil* & *Mask*).
GL_GREATER	Passes if (*Reference* & *Mask*) is greater than (*stencil* & *Mask*).
GL_GEQUAL	Passes if (*Reference* & *Mask*) is greater than or equal to (*stencil* & *Mask*).
GL_EQUAL	Passes if (*Reference* & *Mask*) is equal to (*stencil* & *Mask*).
GL_NOTEQUAL	Passes if (*Reference* & *Mask*) is not equal to (*stencil* & *Mask*).
GL_ALWAYS	Always passes.

Notes

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.

Errors

GL_INVALID_ENUM	*Function* is not one of the eight accepted values.
GL_INVALID_OPERATION	The **glStencilFunc** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glStencilFunc** subroutine are as follows. (See the **glGet** subroutine.)

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**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glAlphaFunc** subroutine, **glBegin** or **glEnd** subroutine, **glBlendFunc** subroutine, **glDepthFunc** subroutine, **glEnable** or **glDisable** subroutine, **glLogicOp** subroutine, **glStencilOp** subroutine.

glStencilMask Subroutine

Purpose

Controls the writing of individual bits in the stencil planes.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glStencilMask(GLuint Mask)
```

Parameters

Mask	Specifies a bit mask to enable and disable writing of individual bits in the stencil planes. Initially, the mask is all 1s.

Description

The **glStencilMask** subroutine controls the writing of individual bits in the stencil planes. The least significant *n* bits of the *Mask* parameter, where *n* is the number of bits in the stencil buffer, specify a mask. Wherever a 1 (one) appears in the mask, the corresponding bit in the stencil buffer is made writable. Where a 0 (zero) appears, the bit is write-protected. Initially, all bits are enabled for writing.

Errors

GL_INVALID_OPERATION	The **glStencilMask** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glStencilMask** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_STENCIL_WRITEMASK**

glGet with argument **GL_STENCIL_BITS**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glColorMask** subroutine, **glDepthMask** subroutine, **glIndexMask** subroutine, **glStencilFunc** subroutine, **glStencilOp** subroutine.

glStencilOp Subroutine

Purpose

Sets stencil test actions.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glStencilOp(GLenum Fail,
       GLenum zFail,
       GLenum zPass)
```

Parameters

Fail Specifies the action to take when the stencil test fails. Six symbolic constants are accepted:

- **GL_KEEP**
- **GL_ZERO**
- **GL_REPLACE**
- **GL_INCR**
- **GL_DECR**
- **GL_INCR_WRAP_EXT**
- **GL_DECR_WRAP_EXT**
- **GL_INVERT**

zFail Specifies stencil action when the stencil test passes but the depth test fails. *zFail* accepts the same symbolic constants as *Fail*.

zPass Specifies 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. *zPass* accepts the same symbolic constants as *Fail*.

Description

Stenciling, like z-buffering, enables and disables drawing on a per-pixel basis. You draw into the stencil planes using GL drawing primitives, and 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 the **glEnable** and **glDisable** subroutine calls with the **GL_STENCIL** argument, and controlled with the **glStencilFunc** subroutine.

The **glStencilOp** subroutine 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 the *Fail* parameter specifies what happens to the stencil buffer contents. The eight possible actions are as follows:

GL_KEEP Keeps the current value.

GL_ZERO Sets the stencil buffer value to 0 (zero).

GL_REPLACE Sets the stencil buffer value to the *Reference* parameter, as specified by the **glStencilFunc** subroutine.

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 0.

GL_INCR_WRAP_EXT	Increments the current stencil buffer value. A **GL_INCR_WRAP_EXT** on the maximum representable unsigned value yields a 0 value.
GL_DECR_WRAP_EXT	Decrements the current stencil buffer value. A **GL_DECR_WRAP_EXT** on 0 yields the maximum representable unsigned value.
GL_INVERT	Bitwise inverts the current stencil buffer value.

Stencil buffer values are treated as unsigned integers. The maximum representable value is *2n-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 (the *zPass* parameter) or fail (the *zFail* parameter). (See the **glDepthFunc** for information about specifying the function used for depth buffer comparisons.) They are specified using the same eight symbolic constants as the *Fail* parameter. Note that the *zFail* parameter is ignored when there is no depth buffer, or when the depth buffer is not enabled. In these cases, the *Fail* and *zPass* parameters specify stencil action when the stencil test fails and passes, respectively.

Notes

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 the **glStencilOp** subroutine.

The **GL_INCR_WRAP_EXT** and **GL_DECR_WRAP_EXT** stencil actions are only supported if the **GL_EXT_stencil_wrap** extension is supported.

Errors

GL_INVALID_ENUM	*Fail*, *zFail*, or *zPass* is any value other than the eight defined constant values.
GL_INVALID_OPERATION	The **glStencilOp** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glStencilOp** subroutine are as follows. (See the **glGet** subroutine for more information.)

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**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glAlphaFunc** subroutine, **glBegin** or **glEnd** subroutine, **glBlendFunc** subroutine, **glDepthFunc** subroutine, **glEnable** or **glDisable** subroutine, **glLogicOp** subroutine, **glStencilFunc** subroutine.

glTexCoord Subroutine

Purpose

Sets the current texture coordinates.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
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)

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

S, *T*, *R*, *Q*	Specify *S*, *T*, *R*, and *A* texture coordinates. Not all parameters are present in all forms of the command.
V	Specifies a pointer to an array of one, two, three, or four elements, which in turn specify the *S*, *T*, *R*, and *Q* texture coordinates.

Description

The **glTexCoord** subroutine specifies texture coordinates in one, two, three, or four dimensions. The **glTexCoord1** subroutine 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 are part of the data that is associated with each vertex and with the current raster position. Initially, the values for *S*, *T*, *R*, and *Q* are (0, 0, 0, 1).

Notes

The current texture coordinates can be updated at any time. In particular, the **glTexCoord** subroutine can be called between a call to **glBegin** and the corresponding call to **glEnd**.

If the **GL_ARB_multitexture** extension is present, then there will be multiple texture units present. This call will only affect the current textrue coordinate on Texture Unit 0. Use **glMultiTexCoord*ARB** to affect texture coordinates on other Texture Units.

Associated Gets

Associated gets for the **glTexCoord** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_CURRENT_TEXTURE_COORDS**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glTexCoordPointer** subroutine, **glTexCoordPointerEXT** subroutine, **glVertex** subroutine.

---

glTexCoordColorNormalVertexSUN Subroutine

Purpose

Specifies a texture coordinate, a color, a normal and a vertex in one call.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
void glTexCoord2fColor4fNormal3fVertex3fSUN (GLfloat  s,
                                              GLfloat  t,
                                              GLfloat  r,
                                              GLfloat  g,
                                              GLfloat  b,
                                              GLfloat  a,
                                              GLfloat  nx,
                                              GLfloat  ny,
                                              GLfloat  nz,
                                              GLfloat  x,
                                              GLfloat  y,
                                              GLfloat  z)
void glTexCoord2fColor4fNormal3fVertex3fvSUN (const GLfloat  *tc,
                                               const GLfloat  *c,
                                               const GLfloat  *n,
                                               const GLfloat  *v)
```

```
void glTexCoord4fColor4fNormal3fVertex4fSUN (GLfloat  s,
                                             GLfloat  t,
                                             GLfloat  p,
                                             GLfloat  q,
                                             GLfloat  r,
                                             GLfloat  g,
                                             GLfloat  b,
                                             GLfloat  a,
                                             GLfloat  nx,
                                             GLfloat  ny,
                                             GLfloat  nz,
                                             GLfloat  x,
                                             GLfloat  y,
                                             GLfloat  z,
                                             GLfloat  w)
void glTexCoord4fColor4fNormal3fVertex4fvSUN (const GLfloat  *tc,
                                              const GLfloat  *c,
                                              const GLfloat  *n,
                                              const GLfloat  *v)
```

Description

This subroutine can be used as a replacement for the following calls:

```
              glTexCoord();
glColor();
glNormal();
glVertex();
```

For example:

glTexCoord4fColor4fNormal3fVertex4fvSUN replaces the following calls:

```
glTexCoord4f();
glColor4f();
glNormal3f();
glVertex4fv();
```

The only reason for using this call is that it reduces the use of bus bandwidth.

Parameters

s, t, p, q	Specifies the *s*, *t*, *p*, and *q* components of the texture coordinate for this vertex. Not all parameters are present in all forms of the command.
tc	Specifies a pointer to an array of texture coordinate values. The elements of a two-element array are *s* and *t*. The elements of a four-element array are *s*, *t*, *p*, and *q*.
r, g, b, a	Specifies the *r*, *g*, *b*, and *a* components of the color for this vertex.
c	Specifies a pointer to an array of the four components *r*, *g*, *b*, and *a*.
nx, ny, nz	Specifies the *x*, *y*, and *z* coordinates of the normal vector for this vertex.
n	Specifies a pointer to an array of the three elements *nx*, *ny* and *nz*.
x, y, z, w	Specifies the *x*, *y*, *z*, and *w* coordinates of a vertex. Not all parameters are present in all forms of the command.

v	Specifies a pointer to an array of vertex coordinates. The elements of a three-element array are *x*, *y*, and *z*. The elements of a four-element array are *x*, *y*, *z*, and *w*.

Notes

Calling **glTexCoordColorNormalVertexSUN** outside of a **glBegin**/**glEnd** subroutine pair results in undefined behavior.

This subroutine is only valid if the **GL_SUN_vertex** extension is defined.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, the **glColor** subroutine, the **glNormal** subroutine, the **glTexCoord** subroutine, the **glVertex** subroutine.

glTexCoordColorVertexSUN Subroutine

Purpose

Specifies a texture coordinate, a color, and a vertex in one call.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
extern void glTexCoord2fColor4ubVertex3fSUN (GLfloat  s,
                                             GLfloat  t,
                                             GLubyte  r,
                                             GLubyte  g,
                                             GLubyte  b,
                                             GLubyte  a,
                                             GLfloat  x,
                                             GLfloat  y,
                                             GLfloat  z)
extern void glTexCoord2fColor4ubVertex3fvSUN (const GLfloat  *tc,
                                              const GLubyte  *c,
                                              const GLfloat  *v)
extern void glTexCoord2fColor3fVertex3fSUN (GLfloat  s,
                                            GLfloat  t,
                                            GLfloat  r,
                                            GLfloat  g,
                                            GLfloat  b,
                                            GLfloat  x,
                                            GLfloat  y,
                                            GLfloat  z)
extern void glTexCoord2fColor3fVertex3fvSUN (const GLfloat  *tc,
                                             const GLfloat  *c,
                                             const GLfloat  *v)
```

Description

This subroutine can be used as a replacement for the following calls:

```
        glTexCoord();
glColor();
glVertex();
```

For example, **glTexCoord2fColor3fVertex3fvSUN** replaces the following calls:

```
glTexCoord2f();
glColor3f();
glVertex3fv();
```

The only reason for using this call is that it reduces the use of bus bandwidth.

Parameters

s, t	Specifies the *s* and *t* components of the texture coordinate for this vertex.
tc	Specifies a pointer to an array of texture coordinate values. The elements of a two-element array are *s* and *t*. The elements of a four-element array are *s*, *t*, *p*, and *q*.
r, g, b, a	Specifies the red, green, blue, and alpha components of a color. Not all parameters are present in all forms of the command.
c	Specifies a pointer to an array of three or four elements. The elements of a three-element array are *r*, *g*, and *b*. The elements of a four-element array are *r*, *g*, *b*, and *a*.
x, y, z	Specifies the *x*, *y*, and *z* coordinates of a vertex.
v	Specifies a pointer to an array of the three elements *x*, *y*, and *z*.

Notes

Calling **glTexCoordColorVertexSUN** outside of a **glBegin**/**glEnd** subroutine pair results in undefined behavior.

This subroutine is only valid if the **GL_SUN_vertex** extension is defined.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, the **glColor** subroutine, the **glNormal** subroutine, the **glTexCoord** subroutine, the **glVertex** subroutine.

glTexCoordNormalVertexSUN Subroutine

Purpose

Specifies a texture coordinate, a normal and a vertex in one call.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
void glTexCoord2fNormal3fVertex3fSUN (GLfloat  s,
                                      GLfloat  t,
                                      GLfloat  nx,
                                      GLfloat  ny,
                                      GLfloat  nz,
                                      GLfloat  x,
                                      GLfloat  y,
                                      GLfloat  z)
void glTexCoord2fNormal3fVertex3fvSUN (const GLfloat  *tc,
                                       const GLfloat  *n,
                                       const GLfloat  *v)
```

Description

This subroutine can be used as a replacement for the following calls:

```
        glTexCoord();
glNormal();
glVertex();
```

For example, **glTexCoord2fNormal3fVertex3fvSUN** replaces the following calls:

```
glTexCoord2f();
glNormal3f();
glVertex3fv();
```

The only reason for using this call is that it reduces the use of bus bandwidth.

Parameters

s, t	Specifies the texture coordinate *s* and *t* values.
tc	Specifies a pointer to an array of the two texture coordinate values *s* and *t*.
x, y, z	Specifies the *x*, *y*, and *z* coordinates of a vertex.
v	Specifies a pointer to an array of the three elements *x*, *y*, and *z*.
nx, ny, nz	Specifies the *x*, *y*, and *z* coordinates of the normal vector for this vertex.
n	Specifies a pointer to an array of the three elements *nx*, *ny* and *nz*.

Notes

Calling **glTexCoordNormalVertexSUN** outside of a **glBegin**/**glEnd** subroutine pair results in undefined behavior.

This subroutine is only valid if the **GL_SUN_vertex** extension is defined.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, the **glColor** subroutine, the **glNormal** subroutine, the **glTexCoord** subroutine, the **glVertex** subroutine.

glTexCoordPointer Subroutine

Purpose

Defines an array of texture coordinates.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glTexCoordPointer( GLint  size,
   GLenum  type,
   GLsizei  stride,
   const GLvoid * pointer)
```

Description

The **glTexCoordPointer** subroutine specifies the location and data format of an array of texture coordinates to use when rendering. The *size* parameter specifies the number of coordinates per element, and must be 1, 2, 3, or 4. The *type* parameter specifies the data type of each texture coordinate and *stride* gives the byte stride from one array element to the next allowing vertices and attributes to be packed into a single array or stored in separate arrays. (Single array storage may be more efficient on some implementations; see **glInterleavedArrays**). When a texture coordinate array is specified, *size*, *type*, *stride*, and *pointer* are saved client side state.

To enable and disable the texture coordinate array, call **glEnableClientState** and **glDisableClientState** with the argument **GL_TEXTURE_COORD_ARRAY**. If enabled, the texture coordinate array is used when **glDrawArrays**, **glDrawElements** or **glArrayElement** is called.

Use **glDrawArrays**, **glMultiDrawArraysEXT**, or **glMultiModeDrawArraysIBM** to construct a sequence of primitives from prespecified vertex and vertex attribute arrays. Use **glArrayElement** to specify primitives by indexing vertices and vertex attributes. Use **glDrawElements**, **glMultiDrawElementsEXT**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

If enabled, the Tex Coord array is used when **glDrawArrays**, **glDrawElements**, **glArrayElements**, **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** is called.

Parameters

size	Specifies the number of coordinates per array element. Must be 1, 2, 3 or 4. The initial value is 4.
type	Specifies the data type of each texture coordinate. Symbolic constants **GL_SHORT**, **GL_INT**, **GL_FLOAT**, or **GL_DOUBLE** are accepted. The initial value is **GL_FLOAT**.
stride	Specifies the byte offset between consecutive array elements. If *stride* is 0, the array elements are understood to be tightly packed. The initial value is 0.
pointer	Specifies a pointer to the first coordinate of the first element in the array. The initial value is 0 (NULL pointer).

Notes

The **glTexCoordPointer** subroutine is available only if the GL version is 1.1 or greater.

The texture coordinate array is initially disabled and it won't be accessed when **glArrayElement**, **glDrawElements** or **glDrawArrays** is called.

Execution of **glTexCoordPointer** is not allowed between **glBegin** and the corresponding **glEnd**, but an error may or may not be generated. If an error is not generated, the operation is undefined.

The **glTexCoordPointer** subroutine is typically implemented on the client side with no protocol.

Since the texture coordinate array parameters are client side state, they are not saved or restored by **glPushAttrib** and **glPopAttrib**. Use **glPushClientAttrib** and **glPopClientAttrib** instead.

The **glTexCoordPointer** subroutine is not included in display lists.

Errors

- **GL_INVALID_VALUE** is generated if size is not 1, 2, 3, or 4.
- **GL_INVALID_ENUM** is generated if type is not an accepted value.
- **GL_INVALID_VALUE** is generated if stride is negative.

Associated Gets

- **glIsEnabled** with argument **GL_TEXTURE_COORD_ARRAY**
- **glGet** with argument **GL_TEXTURE_COORD_ARRAY_SIZE**
- **glGet** with argument **GL_TEXTURE_COORD_ARRAY_TYPE**
- **glGetPointerv** with argument **GL_TEXTURE_COORD_ARRAY_POINTER**

Related Information

The **glArrayElement** subroutine, **glClientActiveTextureARB** subroutine, **glColorPointer** subroutine, **glDrawArrays** subroutine, **glDrawElements** subroutine, **glDrawRangeElements** subroutine, **glEdgeFlagPointer** subroutine, **glEnable** subroutine, **glGetPointerv** subroutine, **glIndexPointer** subroutine, **glNormalPointer** subroutine, **glPopClientAttrib** subroutine, **glPushClientAttrib** subroutine, **glTexCoord** subroutine, **glTexCoordPointerListIBM** subroutine, **glVertexPointer** subroutine.

glTexCoordPointerEXT Subroutine

Purpose

Defines an array of texture coordinates.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glTexCoordPointerEXT(GLint size,
                          GLenum type,
                          GLsizei stride,
                          GLsizei count,
                          const GLvoid *pointer)
```

Parameters

size	Specifies the number of coordinates per array element. It must be 1, 2, 3 or 4.
type	Specifies the data type of each texture coordinate. Symbolic constants **GL_SHORT**, **GL_INT**, **GL_FLOAT**, or **GL_DOUBLE_EXT**, are accepted.
stride	Specifies the byte offset between consecutive array elements. If *stride* is zero the array elements are understood to be tightly packed.
count	Specifies the number of array elements, counting from the first, that are static.

pointer Specifies a pointer to the first coordinate of the first element in the array.

Description

glTexCoordPointerEXT specifies the location and data format of an array of texture coordinates to use when rendering. *size* specifies the number of coordinates per element, and must be 1, 2, 3, or 4. *type* specifies the data type of each texture coordinate and *stride* gives the byte stride from one array element to the next allowing vertexes and attributes to be packed into a single array or stored in separate arrays. (Single-array storage may be more efficient on some implementations.) *count* indicates the number of array elements (counting from the first) that are static. Static elements may be modified by the application, but once they are modified, the application must explicitly respecify the array before using it for any rendering. When a texture coordinate array is specified, *size*, *type*, *stride*, *count*, and *pointer* are saved as client-side state, and static array elements may be cached by the implementation.

The texture coordinate array is enabled and disabled using **glEnable** and **glDisable** with the argument **GL_TEXTURE_COORD_ARRAY_EXT**. If enabled, the texture coordinate array is used when **glDrawArraysEXT** or **glArrayElementEXT** is called.

Notes

Non-static array elements are not accessed until **glArrayElementEXT** or **glDrawArraysEXT** is executed.

By default the texture coordinate array is disabled and it won't be accessed when **glArrayElementEXT** or **glDrawArraysEXT** is called.

Although, it is not an error to call **glTexCoordPointerEXT** between the execution of **glBegin** and the corresponding execution of **glEnd**, the results are undefined.

glTexCoordPointerEXT will typically be implemented on the client side with no protocol.

Since the texture coordinate array parameters are client side state, they are not saved or restored by **glPushAttrib** and **glPopAttrib**.

glTexCoordPointerEXT commands are not entered into display lists.

glTexCoordPointerEXT is part of the _extname(EXT_vertex_array) extension, not part of the core GL command set. If _extstring(EXT_vertex_array) is included in the string returned by **glGetString**, when called with argument **GL_EXTENSIONS**, extension _extname(EXT_vertex_array) is supported.

Use **glDrawArrays**, **glMultiDrawArraysEXT**, or **glMultiModeDrawArraysIBM** to construct a sequence of primitives from prespecified vertex and vertex attribute arrays. Use **glArrayElement** to specify primitives by indexing vertices and vertex attributes. Use **glDrawElements**, **glMultiDrawElementsEXT**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

If enabled, the Tex Coord array is used when **glDrawArrays**, **glDrawElements**, **glArrayElements**, **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** is called.

Errors

GL_INVALID_VALUE is generated if *size* is not 1, 2, 3, or 4.

GL_INVALID_ENUM is generated if *type* is not an accepted value.

GL_INVALID_VALUE is generated if *stride* or *count* is negative

Associated Gets

glIsEnabled with argument **GL_TEXTURE_COORD_ARRAY_EXT**

glGet with argument **GL_TEXTURE_COORD_ARRAY_SIZE_EXT**

glGet with argument **GL_TEXTURE_COORD_ARRAY_TYPE_EXT**

glGet with argument **GL_TEXTURE_COORD_ARRAY_STRIDE_EXT**

glGet with argument **GL_TEXTURE_COORD_ARRAY_COUNT_EXT**

glGetPointervEXT with argument **GL_TEXTURE_COORD_ARRAY_POINTER_EXT**

File

/usr/include/GL/glext.h	Contains extensions to C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glArrayElementEXT** subroutine, **glColorPointerEXT** subroutine, **glDrawArraysEXT** subroutine, **glEdgeFlagPointerEXT** subroutine, **glGetPointervEXT** subroutine, **glIndexPointerEXT** subroutine, **glNormalPointerEXT** subroutine, **glVertexPointerEXT** subroutine.

---

glTexCoordPointerListIBM Subroutine

Purpose

Defines a list of texture coordinate arrays.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glTexCoordPointerListIBM(GLint  size,
   GLenum  type,
   GLint  stride,
   const GLvoid ** pointer,
   GLint  ptrstride)
```

Description

The **glTexCoordPointerListIBM** subroutine specifies the location and data format of a list of arrays of texture coordinate components to use when rendering. The *size* parameter specifies the number of components per texture coordinate, and must be 1, 2, 3 or 4. The *type* parameter specifies the data type of each texture coordinate component. The *stride* parameter gives the byte stride from one texture coordinate to the next allowing vertices and attributes to be packed into a single array or stored in separate arrays. (Single-array storage may be more efficient on some implementations; see **glInterleavedArrays**). The *ptrstride* parameter specifies the byte stride from one pointer to the next in the *pointer* array.

When a texture coordinate array is specified, *size*, *type*, *stride*, *pointer* and *ptrstride* are saved as client side state.

A *stride* value of 0 does not specify a “tightly packed” array as it does in **glTexCoordPointer**. Instead, it causes the first array element of each array to be used for each vertex. Also, a negative value can be used for stride, which allows the user to move through each array in reverse order.

To enable and disable the texture coordinate arrays, call **glEnableClientState** and **glDisableClientState** with the argument **GL_TEXTURE_COORD_ARRAY**. The texture coordinate array is initially disabled. When enabled, the texture coordinate arrays are used when **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, **glDrawArrays**, **glDrawElements** or **glArrayElement** is called. The last three calls in this list will only use the first array (the one pointed at by *pointer*[0]). See the descriptions of these routines for more information on their use.

Use **glDrawArrays**, **glMultiDrawArraysEXT**, or **glMultiModeDrawArraysIBM** to construct a sequence of primitives from prespecified vertex and vertex attribute arrays. Use **glArrayElement** to specify primitives by indexing vertices and vertex attributes. Use **glDrawElements**, **glMultiDrawElementsEXT**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

If enabled, the Tex Coord array is used when **glDrawArrays**, **glDrawElements**, **glArrayElements**, **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** is called.

Parameters

size	Specifies the number of components per texture coordinate. It must be 1, 2, 3 or 4. The initial value is 4.
type	Specifies the data type of each texture coordinate component in the array. Symbolic constants **GL_BYTE**, **GL_UNSIGNED_BYTE**, **GL_SHORT**, **GL_UNSIGNED_SHORT**, **GL_INT**, **GL_UNSIGNED_INT**, **GL_FLOAT**, or **GL_DOUBLE** are accepted. The initial value is GL_FLOAT.
stride	Specifies the byte offset between consecutive texture coordinates. The initial value is 0.
pointer	Specifies a list of texture coordinate arrays. The initial value is 0 (NULL pointer).
ptrstride	Specifies the byte stride between successive pointers in the *pointer* array. The initial value is 0.

Notes

The **glTexCoordPointerListIBM** subroutine is available only if the GL_IBM_vertex_array_lists extension is supported.

Execution of **glTexCoordPointerListIBM** is not allowed between **glBegin** and the corresponding **glEnd**, but an error may or may not be generated. If an error is not generated, the operation is undefined.

The **glTexCoordPointerListIBM** subroutine is typically implemented on the client side.

Since the texture coordinate array parameters are client side state, they are not saved or restored by **glPushAttrib** and **glPopAttrib**. Use **glPushClientAttrib** and **glPopClientAttrib** instead.

When a **glTexCoordPointerListIBM** call is encountered while compiling a display list, the information it contains does NOT contribute to the display list, but is used to update the immediate context instead.

The **glTexCoordPointer** call and the **glTexCoordPointerListIBM** call share the same state variables. A **glTexCoordPointer** call will reset the texture coordinate list state to indicate that there is only one texture coordinate list, so that any and all lists specified by a previous **glTexCoordPointerListIBM** call will be lost, not just the first list that it specified.

Error Codes

- **GL_INVALID_VALUE** is generated if size is not 1, 2, 3 or 4.
- **GL_INVALID_ENUM** is generated if type is not an accepted value.

Associated Gets

- **glIsEnabled** with argument **GL_TEXTURE_COORD_ARRAY**
- **glGetPointerv** with argument **GL_TEXTURE_COORD_ARRAY_LIST_IBM**
- **glGet** with argument **GL_TEXTURE_COORD_ARRAY_LIST_STRIDE_IBM**
- **glGet** with argument **GL_TEXTURE_COORD_ARRAY_SIZE**
- **glGet** with argument **GL_TEXTURE_COORD_ARRAY_STRIDE**
- **glGet** with argument **GL_TEXTURE_COORD_ARRAY_TYPE**

Related Information

The **glArrayElement** subroutine, **glClientActiveTextureARB** subroutine, **glDrawArrays** subroutine, **glDrawElements** subroutine, **glDrawRangeElements** subroutine, **glEdgeFlagPointer** subroutine, **glEnable** subroutine, **glGetPointerv** subroutine, **glIndexPointer** subroutine, **glInterleavedArrays** subroutine, **glMultiDrawArraysEXT** subroutine, **glMultiDrawElementsEXT** subroutine, **glMultiModeDrawArraysIBM** subroutine, **glMultiModeDrawElementsIBM** subroutine, **glNormalPointer** subroutine, **glPopClientAttrib** subroutine, **glPushClientAttrib** subroutine, **glTexCoordPointer** subroutine, **glVertexPointer** subroutine.

---

glTexCoordVertexSUN Subroutine

Purpose

Specifies a texture coordinate and a vertex in one call.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
void glTexCoord2fVertex3fSUN (GLfloat  s,
                              GLfloat  t,
                              GLfloat  x,
                              GLfloat  y,
                              GLfloat  z)
void glTexCoord2fVertex3fvSUN (const GLfloat  *tc,
                               const GLfloat  *v)
void glTexCoord4fVertex4fSUN (GLfloat  s,
                              GLfloat  t,
                              GLfloat  p,
                              GLfloat  q,
                              GLfloat  x,
                              GLfloat  y,
                              GLfloat  z,
                              GLfloat  w)
void glTexCoord4fVertex4fvSUN (const GLfloat  *tc,
                               const GLfloat  *v)
```

Description

This subroutine can be used as a replacement for the following calls:

```
                glTexCoord();
  glVertex();
```

For example, **glTexCoord4fVertex4fvSUN** replaces the following calls:

```
  glTexCoord4f();
  glVertex4fv();
```

The only reason for using this call is that it reduces the use of bus bandwidth.

Parameters

s, t, p, q	Specifies the *s*, *t*, *p*, and *q* components of the texture coordinate for this vertex. Not all parameters are present in all forms of the command.
tc	Specifies a pointer to an array of texture coordinate values. The elements of a two-element array are *s* and *t*. The elements of a four-element array are *s*, *t*, *p*, and *q*.
x, y, z, w	Specifies the *x*, *y*, *z*, and *w* coordinates of a vertex. Not all parameters are present in all forms of the command.
v	Specifies a pointer to an array of vertex coordinates. The elements of a three-element array are *x*, *y*, and *z*. The elements of a four-element array are *x*, *y*, *z*, and *w*.

Notes

Calling **glTexCoordVertexSUN** outside of a **glBegin**/**glEnd** subroutine pair results in undefined behavior.

This subroutine is only valid if the **GL_SUN_vertex** extension is defined.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, the **glColor** subroutine, the **glNormal** subroutine, the **glTexCoord** subroutine, the **glVertex** subroutine.

---

glTexEnv Subroutine

Purpose

Sets texture environment parameters.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glTexEnvf(GLenum  Target,
   GLenum  pName,
   GLfloat  Parameter)
```

```
void glTexEnvi(GLenum Target,
   GLenum pName,
   GLint Parameter)

void glTexEnvfv(GLenum Target,
     GLenum pName,
     const GLfloat * Parameters)

void glTexEnviv(GLenum Target,
    GLenum pName,
    const GLint * Parameters)
```

Parameters

glTexEnvf or glTexEnvi

Target
Specifies a texture environment. Must be **GL_TEXTURE_ENV**.

pName
Specifies the symbolic name of a single-valued texture environment parameter. Accepted values are:

- **GL_TEXTURE_ENV_MODE**
- **GL_COMBINE_RGB_EXT**
- **GL_COMBINE_ALPHA_EXT**
- **GL_SOURCE0_RGB_EXT**
- **GL_SOURCE1_RGB_EXT**
- **GL_SOURCE2_RGB_EXT**
- **GL_SOURCE0_ALPHA_EXT**
- **GL_SOURCE1_ALPHA_EXT**
- **GL_SOURCE2_ALPHA_EXT**
- **GL_OPERAND0_RGB_EXT**
- **GL_OPERAND1_RGB_EXT**
- **GL_OPERAND2_RGB_EXT**
- **GL_OPERAND0_ALPHA_EXT**
- **GL_OPERAND1_ALPHA_EXT**
- **GL_OPERAND2_ALPHA_EXT**
- **GL_RGB_SCALE_EXT**, or
- **GL_ALPHA_SCALE**

Parameter
Specifies a single symbolic constant, one of **GL_MODULATE**, **GL_DECAL**, **GL_BLEND**, **GL_COMBINE_EXT**, **GL_ADD**, **GL_REPLACE**, **GL_ADD_SIGNED_EXT** or **GL_INTERPOLATE_EXT**.

glTexEnvfv or glTexEnviv

Target Specifies a texture environment. Must be **GL_TEXTURE_ENV**.

pName	Specifies the symbolic name of a texture environment parameter. Accepted values are: • **GL_TEXTURE_ENV_MODE** • **GL_TEXTURE_ENV_COLOR** • **GL_COMBINE_RGB_EXT** • **GL_COMBINE_ALPHA_EXT** • **GL_SOURCE0_RGB_EXT** • **GL_SOURCE1_RGB_EXT** • **GL_SOURCE2_RGB_EXT** • **GL_SOURCE0_ALPHA_EXT** • **GL_SOURCE1_ALPHA_EXT** • **GL_SOURCE2_ALPHA_EXT** • **GL_OPERAND0_RGB_EXT** • **GL_OPERAND1_RGB_EXT** • **GL_OPERAND2_RGB_EXT** • **GL_OPERAND0_ALPHA_EXT** • **GL_OPERAND1_ALPHA_EXT** • **GL_OPERAND2_ALPHA_EXT** • **GL_RGB_SCALE_EXT** • **GL_ALPHA_SCALE**
Parameters	Specifies a pointer to an array of parameters: either a single symbolic constant or an RGBA color.

Description

A *texture environment* specifies how texture values are interpreted when a fragment is textured.

If the *pName* parameter is **GL_TEXTURE_ENV_MODE**, the *Parameter(s)* parameter is (or points to) the symbolic name of a texture function. Six texture functions are defined: **GL_MODULATE**, **GL_DECAL**, **GL_BLEND**, **GL_REPLACE**, **GL_ADD** or **GL_COMBINE**. **GL_TEXTURE_ENV_MODE** defaults to **GL_MODULATE**

If the *pName* parameter is **GL_TEXTURE_ENV_COLOR**, the *Parameters* parameter 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. *Cc* (see tables below) takes these four values. **GL_TEXTURE_ENV_COLOR** defaults to (0,0,0,0).

If the *pName* parameter is **GL_COMBINE_RGB_EXT** or **GL_COMBINE_ALPHA_EXT**, the *Parameter(s)* parameter is (or points to) the symbolic name of a texture function. Five texture functions are defined: **GL_MODULATE**, **GL_REPLACE**, **GL_ADD**, **GL_ADD_SIGNED_EXT** or **GL_INTERPOLATE_EXT**. The default value for these *pNames* is **GL_MODULATE**.

If the *pName* parameter is **GL_SOURCE0_RGB_EXT**, **GL_SOURCE1_RGB_EXT**, **GL_SOURCE2_RGB_EXT**, **GL_SOURCE0_ALPHA_EXT**, **GL_SOURCE1_ALPHA_EXT**, or **GL_SOURCE2_ALPHA_EXT**, the *Parameter(s)* parameter is (or points to) the symbolic name of a texture operator. Four texture operators are defined: **GL_TEXTURE**, **GL_CONSTANT_EXT**, **GL_PRIMARY_COLOR_EXT**, or **GL_PREVIOUS_EXT**. The default value for these *pNames* are shown in the following table:

Parameter	Default value
GL_SOURCE0_RGB_EXT	GL_TEXTURE
GL_SOURCE1_RGB_EXT	GL_PREVIOUS_EXT

Parameter	Default value
GL_SOURCE2_RGB_EXT	**GL_CONSTANT_EXT**
GL_SOURCE0_ALPHA_EXT	**GL_TEXTURE**
GL_SOURCE1_ALPHA_EXT	**GL_PREVIOUS_EXT**
GL_SOURCE2_ALPHA_EXT	**GL_CONSTANT_EXT**

If the *pName* parameter is **GL_OPERAND0_RGB_EXT**, or **GL_OPERAND1_RGB_EXT**, the *Parameter(s)* parameter is (or points to) the symbolic name of a texture operand. Four texture operands are defined: **GL_SRC_COLOR**, **GL_ONE_MINUS_SRC_COLOR**, **GL_SRC_ALPHA**, or **GL_ONE_MINUS_SRC_ALPHA**. The default value for these *pNames* is **GL_SRC_COLOR**.

If the *pName* parameter is **GL_OPERAND0_ALPHA_EXT**, or **GL_OPERAND1_ALPHA_EXT**, the *Parameter(s)* parameter is (or points to) the symbolic name of a texture operand. Two texture operands are defined: **GL_SRC_ALPHA**, or **GL_ONE_MINUS_SRC_ALPHA**. The default value for these *pNames* is **GL_SRC_ALPHA**.

If the *pName* parameter is **GL_OPERAND2_RGB_EXT**, or **GL_OPERAND2_ALPHA_EXT**, the *Parameter(s)* parameter is (or points to) the symbolic name of a texture operand. One texture operand is defined: **GL_SRC_ALPHA**.

If the *pName* parameter is **GL_RGB_SCALE_EXT**, or **GL_ALPHA_SCALE**, the *Parameter(s)* parameter is (or points to) a floating-point scale factor. Only three such scale factors are valid: 1.0, 2.0, and 4.0. The default value is 1.0.

A *texture function* acts on the fragment to be textured using the texture image value that applies to the fragment and produces a red, green, blue, alpha (RGBA) color for that fragment. (See the **glTexParameter** subroutine for details on setting texture parameters.)

A *texture image* can have up to four components per texture element. (See the **glTexImage1D** subroutine, the **glTexImage2D** subroutine, and the **glTexImage3DEXT** subroutine.) 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.

The following table shows how the RGBA color is produced when the **GL_TEXTURE_ENV_MODE** is *NOT* **GL_COMBINE_EXT**. *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.

Note: In the following table, ″It″ equals the texture intensity.

	Texture Functions				
Internal Formats	**GL_ MODULATE**	**GL_ DECAL**	**GL_ BLEND**	**GL_ REPLACE**	**GL_ ADD**
GL_ LUMINANCE or 1	*Cv=LtCf Av=Af*	undefined	*Cv=(1- <I>Lt) Cf+LtCc Av=Af*	*Cv=Lt Av=Af*	*Cv=Cf+Lt Av=Af*
GL_ LUMINANCE_ ALPHA or 2	*Cv=LtCf Av=AtAf*	undefined	*Cv*=(1- *Lt*) *Cf+LtCc Av=AtAf*	*Cv=Lt Av=At*	*Cv=Cf+Lt Av=AfAt*
GL_ RGB or 3	*Cv=CtCf Av=Af*	*Cv=Ct Av=Af*	*Cv*=(1- *Ct*) *Cf+CtCc Av=Af*	*Cv=Ct Av=Af*	*Cv=Cf+Ct Av=Af*

	Texture Functions				
Internal Formats	**GL_ MODULATE**	**GL_ DECAL**	**GL_ BLEND**	**GL_ REPLACE**	**GL_ ADD**
GL_ RGBA or 4	*Cv=CtCf Av=AtAf*	*Cv*=(1-*At*) *Cf*+*AtCt* *Av=Af*	*Cv*=(1-*Ct*) *Cf*+*CtCc* *Av=AtAf*	*Cv=Ct Av=At*	*Cv=Cf+Ct Av=AfAt*
GL_ INTENSITY	*Cv=ItCf Av=ItAf*	undefined	*Cv*=(1-*It*) *Cf*+*ItCc* *Av*=(1-*It*) *Af*+*ItAc*	*Cv=It Av=It*	*Cv=Cf+It Av=Af+It*
GL_ ALPHA	*Cv=Cf Av=AtAf*	undefined	*Cv=Cf Av=AtAf*	*Cv=Cf Av=At*	*Cv=Cf Av=AfAt*

If the value of **GL_TEXTURE_ENV_MODE** is **GL_COMBINE_EXT**, the form of the texture function depends on the values of **GL_COMBINE_RGB_EXT** and **GL_COMBINE_ALPHA_EXT**, according to the following table:

Combine Function	**Texture Function**
GL_REPLACE	Arg0
GL_MODULATE	Arg0 * Arg1
GL_ADD	Arg0 + Arg1
GL_ADD_SIGNED_EXT	Arg0 + Arg1 - 0.5
GL_INTERPOLATE_EXT	Arg0 * (Arg2) + Arg1 * (1-Arg2)

The RGB and ALPHA results of the texture function are then multiplied by the values of **GL_RGB_SCALE_EXT** and **GL_ALPHA_SCALE**, respectively. The results are clamped to [0,1].

The arguments Arg0, Arg1 and Arg2 are determined by the values of **GL_SOURCE(n)_RGB_EXT**, **GL_SOURCE(n)_ALPHA_EXT**, **GL_OPERAND(n)_RGB_EXT** and **GL_OPERAND(n)_ALPHA_EXT**. In the following two tables, *Ct* and *At* are the filtered texture RGB and alpha values; *Cc* and *Ac* are the texture environment RGB and alpha values; *Cf* and *Af* are the RGB and alpha of the primary color of the incoming fragment; and *Cp* and *Ap* are the RGB and alpha values resulting from the previous texture environment. On texture unit *0*, *Cp* and *Ap* are identical to *Cf* and *Af*, respectively. The relationship is described in the following two tables:

GL_ SOURCE(n)_ RGB_ EXT	**GL_ SRC_ COLOR**	**GL_ ONE_ MINUS_ SRC_ COLOR**	**GL_ SRC_ ALPHA**	**GL_ ONE_ MINUS_ SRC_ ALPHA**
GL_ TEXTURE	Ct	(1-Ct)	At	(1-At)
GL_ CONSTANT_ EXT	Cc	(1-Cc)	Ac	(1-Ac)
GL_ PRIMARY_ COLOR_ EXT	Cf	(1-Cf)	Af	(1-Af)
GL_ PREVIOUS_ EXT	Cp	(1-Cp)	Ap	(1-Ap)

GL_SOURCE(n)_ALPHA_EXT	**GL_SRC_ALPHA**	**GL_ONE_MINUS_SRC_ALPHA**
GL_TEXTURE	At	(1-At)
GL_CONSTANT_EXT	Ac	(1-Ac)
GL_PRIMARY_COLOR_EXT	Af	(1-Af)
GL_PREVIOUS_EXT	Ap	(1-Ap)

The mapping of texture components to source components is summarized in the following table, where *At*, *Lt*, *It*, *Rt*, *Gt* and *Bt* are the filtered texel values.

Base Internal Format>	RGB Values	Alpha Value
GL_ALPHA	0, 0, 0	At
GL_LUMINANCE	Lt, Lt, Lt	1
GL_LUMINANCE_ALPHA	Lt, Lt, Lt	At
GL_INTENSITY	It, It, It	It
GL_RGB	Rt, Gt, Bt	1
GL_RGBA	Rt, Gt, Bt	At

Notes

GL_ADD is only valid if the **GL_EXT_texture_env_add** extension is present.

GL_COMBINE_EXT, **GL_ADD_SIGNED_EXT**, **GL_INTERPOLATE_EXT**, **GL_COMBINE_RGB_EXT**, **GL_COMBINE_ALPHA_EXT**,**GL_SOURCEn_RGB_EXT**, **GL_SOURCEn_ALPHA_EXT**, **GL_OPERANDn_RGB_EXT**, **GL_OPERANDn_ALPHA_EXT**, **GL_RGB_SCALE_EXT**, and **GL_ALPHA_SCALE** are only valid if the **GL_EXT_texture_env_combine** extension is present.

Error Codes

GL_INVALID_ENUM	*Target* or *pName* is not one of the accepted defined values, or *Parameters* should have a defined constant value (based on the value of *pName*) and does not.
INVALID_ENUM	The *Parameter(s)* value for **GL_COMBINE_RGB_EXT** or **GL_COMBINE_ALPHA_EXT** is not one of **GL_REPLACE**, **GL_MODULATE**, **GL_ADD**, **GL_ADD_SIGNED_EXT**, or **GL_INTERPOLATE_EXT**.
INVALID_ENUM	The *Parameter(s)* value for **GL_SOURCE0_RGB_EXT**, **GL_SOURCE1_RGB_EXT**, **GL_SOURCE2_RGB_EXT**, **GL_SOURCE0_ALPHA_EXT**, **GL_SOURCE1_ALPHA_EXT** or **GL_SOURCE2_ALPHA_EXT** is not one of **GL_TEXTURE**, **GL_CONSTANT_EXT**, **GL_PRIMARY_COLOR_EXT** or **GL_PREVIOUS_EXT**.
INVALID_ENUM	The *Parameter(s)* value for **GL_OPERAND0_RGB_EXT** or **GL_OPERAND1_RGB_EXT** is not one of **GL_SRC_COLOR**, **GL_ONE_MINUS_SRC_COLOR**, **GL_SRC_ALPHA** or **GL_ONE_MINUS_SRC_ALPHA**.
INVALID_ENUM	The *Parameter(s)* value for **GL_OPERAND0_ALPHA_EXT** or **GL_OPERAND1_ALPHA_EXT** is not one of **GL_SRC_ALPHA** or **GL_ONE_MINUS_SRC_ALPHA**.
INVALID_ENUM	The *Parameter(s)* value for **GL_OPERAND2_RGB_EXT** or **GL_OPERAND2_ALPHA_EXT** is not **GL_SRC_ALPHA**.
INVALID_VALUE	The *Parameter(s)* value for **RGB_SCALE_EXT** or **ALPHA_SCALE** is not one of 1.0, 2.0, or 4.0.
GL_INVALID_OPERATION	The **glTexEnv** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glTexEnv** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGetTexEnv.

Files

/usr/include/GL/gl.h Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glGetTexEnv** subroutine, **glTexImage1D** subroutine, **glTexImage2D** subroutine, **glTexParameter** subroutine.

glTexGen Subroutine

Purpose

Controls the generation of texture coordinates.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glTexGend(GLenum  Coordinate,
     GLenum  pName,
     GLdouble  Parameter)

void glTexGenf(GLenum  Coordinate,
    GLenum  pName,
    GLfloat  Parameter)

void glTexGeni(GLenum  Coordinate,
    GLenum  pName,
    GLint  Parameter)

void glTexGendv(GLenum  Coordinate,
       GLenum  pName,
       const GLdouble * Parameters)

void glTexGenfv(GLenum  Coordinate,
      GLenum  pName,
      const GLfloat * Parameters)

void glTexGeniv(GLenum  Coordinate,
      GLenum  pName,
      const GLint * Parameters)
```

Parameters

glTexGend, glTexGenf or glTexGeni

Coordinate Specifies a texture coordinate. Must be one of the following:

- **GL_S**
- **GL_T**
- **GL_R**
- **GL_Q**

pName	Specifies the symbolic name of the texture-coordinate generation function. Must be **GL_TEXTURE_GEN_MODE**.
Parameter	Specifies a single-valued texture generation parameter, one of **GL_OBJECT_LINEAR**, **GL_EYE_LINEAR**, or **GL_SPHERE_MAP**.

glTexGendv, glTexGenfv or glTexGeniv

Coordinate	Specifies a texture coordinate. Must be one of the following: • **GL_S** • **GL_T** • **GL_R** • **GL_Q**
pName	Specifies 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**.
Parameters	Specifies a pointer to an array of texture generation parameters. If *pName* is **GL_TEXTURE_GEN_MODE**, the array must contain a single symbolic constant, one of **GL_OBJECT_LINEAR**, **GL_EYE_LINEAR**, or **GL_SPHERE_MAP**. Otherwise, *Parameters* holds the coefficients for the texture-coordinate generation function specified by *pName*.

Description

The **glTexGen** subroutine selects a texture-coordinate generation function or supplies coefficients for one of the functions. The *Coordinate* 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**, the *Parameters* parameter 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**, the *Parameters* parameter contains coefficients for the corresponding texture generation function.

If the texture generation function is **GL_OBJECT_LINEAR**, the following function is used:

$$g=p_1x_0+p_2y_0+p_3z_0+p_4w_0$$

Figure 23. GL_OBJECT_LINEAR Function. This figure shows that g is equal to p subscript one x subscript zero + p subscript two y subscript zero + p subscript three z subscript zero + p subscript four w subscript zero.

where *g* is the value computed for the coordinate named in the *Coordinate* parameter, *p*1, *p*2, *p*3, and *p*4 are the four values supplied in the *Parameters* parameter, and *x*0, *y*0, *z*0, *w*0 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 following function is used:

$$g=p_{1'}x_e+p_{2'}y_e+p_{3'}z_e+p_{4'}w_e$$

Figure 24. GL_EYE_LINEAR Function. This figure shows that g is equal to p subscript one′ x subscript e + p subscript two′ y subscript e + p subscript three′ z subscript e + p subscript four′ w subscript e.

where:

$$(p_{1'}\ p_{2'}\ p_{3'}\ p_{4'})=(p_1\ p_2\ p_3\ p_4)M^{-1}$$

Figure 25. GL_EYE_LINEAR Function Definition. This figure shows that (p subscript one′ p subscript two′ p subscript three′ p subscript four′) equals (p subscript one p subscript two p subscript three p subscript four)M to the power of -1.

and *x*e*,* *y*e*,* *z*e*,* and *w*e are the eye coordinates of the vertex, *p*1*,* *p*2*,* *p*3*,* *p*4 are the values supplied in *Parameters*, 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 the *Parameters* parameter 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 the *pName* parameter is **GL_SPHERE_MAP** and the *Coordinate* parameter is either **GL_R** or **GL_Q**, *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 **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

f=**u**-2**n'n'**T**u**

Finally, let *m*=2(square root (*f*x2*+f*y2*+*(*f*z*+1*)2)). Then the values assigned to the *s* and *t* texture coordinates are the following:

$$s=\frac{f_x}{m}+1/2$$

$$t=\frac{f_y}{m}+1/2$$

Figure 26. s and t Values. This figure shows two equations, one for each texture coordinate. The first equation shows that texture coordinate s is equal to f subscript x / m + 1/2. The second equation shows that texture coordinate t is equal to f subscript y / m + 1/2.

A texture-coordinate generation function is enabled or disabled using the **glEnable** or **glDisable** subroutines 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).

Error Codes

GL_INVALID_ENUM	*Coordinate* or *pName* is not an accepted defined value, or *pName* is **GL_TEXTURE_GEN_MODE** and *Parameters* is not an accepted defined value.
GL_INVALID_ENUM	*pName* is **GL_TEXTURE_GEN_MODE**, *Parameters* is **GL_SPHERE_MAP**, and *Coordinate* is either **GL_R** or **GL_Q**.
GL_INVALID_OPERATION	The **glTexGen** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glTexGen** subroutine are as follows. (See the **glGet** subroutine for more information.)

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**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glEnable** or **glDisable** subroutine, **glGetTexGen** subroutine, **glTexEnv** subroutine, **glTexImage1D** subroutine, **glTexImage2D** subroutine, **glTexImage3DEXT** subroutine, **glTexParameter** subroutine.

glTexImage1D Subroutine

Purpose

Specifies a one-dimensional (1D) texture image.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glTexImage1D(GLenum  target,
                  GLint  level,
                  GLint  internalformat,
                  GLsizei  width,
```

```
GLint border,
GLenum format,
GLenum type,
const GLvoid * pixels)
```

Parameters

target	Specifies the target texture. Must be **GL_TEXTURE_1D** of **GL_PROXY_TEXTURE_1D**.
level	Specifies the level-of-detail number. Level 0 is the base image level. Level n is the nth mipmap reduction image.
internalformat	Specifies the number of color components in the texture. Must be 1, 2, 3, or 4, or one of the following symbolic constants: **GL_ABGR_EXT, GL_ALPHA**, **GL_ALPHA4**, **GL_ALPHA8**, **GL_ALPHA12**, **GL_ALPHA16**, **GL_LUMINANCE**, **GL_LUMINANCE4**, **GL_LUMINANCE8**, **GL_LUMINANCE12**, **GL_LUMINANCE16**, **GL_LUMINANCE_ALPHA**, **GL_LUMINANCE4_ALPHA4**, **GL_LUMINANCE6_ALPHA2**, **GL_LUMINANCE8_ALPHA8**, **GL_LUMINANCE12_ALPHA4**, **GL_LUMINANCE12_ALPHA12**, **GL_LUMINANCE16_ALPHA16**, **GL_INTENSITY**, **GL_INTENSITY4**, **GL_INTENSITY8**, **GL_INTENSITY12**, **GL_INTENSITY16**, **GL_RGB**, **GL_R3_G3_B2**, **GL_RGB4**, **GL_RGB5**, **GL_RGB8**, **GL_RGB10**, **GL_RGB12**, **GL_RGB16**, **GL_RGBA**, **GL_RGBA2**, **GL_RGBA4**, **GL_RGB5_A1**,**GL_RGBA8**, **GL_RGB10_A2**, **GL_RGBA12**, **or GL_RGBA16**.
width	Specifies the width of the texture image. Must be 2n + 2 **x** *border* for some integer n. All implementations support texture images that are at least 64 texels wide. The height of the 1D texture image is 1.
border	Specifies the width of the border. Must be either 0 or 1.
format	Specifies the format of the pixel data. Symbolic constants **GL_COLOR_INDEX**, **GL_RED**, **GL_GREEN**, **GL_BLUE**, **GL_ALPHA**, **GL_RGB**, **GL_RGBA**, **GL_BGR**, **GL_BGRA**, **GL_ABGR_EXT**, **GL_LUMINANCE**, **GL_422_EXT**, **GL_422_REV_EXT**, **GL_422_AVERAGE_EXT**, **GL_422_REV_AVERAGE_EXT**, and **GL_LUMINANCE_ALPHA** are accepted.
type	Specifies the data type for *Pixels*. Symbolic constants **GL_UNSIGNED_BYTE**, **GL_BYTE**, **GL_BITMAP**, **GL_UNSIGNED_SHORT**, **GL_SHORT**, **GL_UNSIGNED_INT**, **GL_INT**, **GL_FLOAT**, **GL_UNSIGNED_BYTE_3_3_2**, **GL_UNSIGNED_BYTE_2_3_3_REV**, **GL_UNSIGNED_SHORT_5_6_5**, **GL_UNSIGNED_SHORT_5_6_5_REV**, **GL_UNSIGNED_SHORT_4_4_4_4**, **GL_UNSIGNED_SHORT_4_4_4_4_REV**, **GL_UNSIGNED_SHORT_5_5_5_1**, **GL_UNSIGNED_SHORT_1_5_5_5_REV**, **GL_UNSIGNED_INT_8_8_8_8**, **GL_UNSIGNED_INT_8_8_8_8_REV**, **GL_UNSIGNED_INT_10_10_10_2**, and **GL_UNSIGNED_INT_2_10_10_10_REV** are accepted.
pixels	Specifies a pointer to the image data in memory.

Description

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. To enable and disable one-dimensional texturing, call **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 the internal resolution and format used to store the image. The last three arguments describe how the image is represented in memory; they are identical to the pixel formats used for **glDrawPixels**.

If *target* is **GL_PROXY_TEXTURE_1D** no data is read from *pixels*, but all of the texture image state is recalculated, checked for consistency, and checked against the

implementation's capabilities. If the implementation cannot handle a texture of the requested texture size, it sets all of the image state to 0, but does not generate an error (See **glGetError**). To query for an entire mipmap array, use an image array level greater than or equal to 1.

If *target* is **GL_TEXTURE_1D**, 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**).

The first element corresponds to the left end of the texture array. Subsequent elements progress left-to-right through the remaining texels in the texture array. The final element corresponds to the right end of the texture array.

The *format* parameter determines the composition of each element in *pixels*. It can assume one of 16 symbolic values:

GL_COLOR_INDEX	Each element is a single value, a color index. The GL converts it to fixed point (with an unspecified number of zero 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. The GL converts it to floating point and assembles it 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. The GL converts it to floating point and assembles it 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. The GL converts it to floating point and assembles it 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. The GL converts it to floating point and assembles it 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. The GL converts it to floating point and assembles it into an RGBA element by attaching 1.0 for alpha. Eachcomponent 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 contains all four components. 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	Each pixel is a three-component group, blue first, followed by green, followed by red. Each component is converted to the internal floating-point format in the same way as the blue, green, and red components of an BGRA pixel are. The color triple is converted to an BGRA pixel with alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an BGRA pixel.

GL_BGRA

Each pixel is a four-component group, blue first, followed by green, followed by red, followed by alpha. Floating-point values are converted 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 are mapped similarly: the largest integer value maps to 1.0, and 0 maps to 0.0. The resulting floating-point color values are then multiplied by **GL_*c*_SCALE** and added to **GL_*c*_BIAS**, where *c* is **BLUE**, **GREEN**, **RED**, and **ALPHA** for the respective color components. The results are clamped to the range [0,1].

If **GL_MAP_COLOR** is True, each color component is scaled by the size of the lookup table **GL_PIXEL_MAP_*c*_TO_*c***, then replaced by the value that it references in that table. *c* is **B**, **G**, **R**, or **A**, respectively.

The resulting BGRA colors are then converted 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 $x_n = x_r + n \bmod Width$ and $y_n = y_r + [n/Width]$, where (x_r, y_r) is the current raster position. 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_ABGR_EXT

Each pixel is a four-component group: for **GL_RGBA**, the red component is first, followed by green, followed by blue, followed by alpha; for **GL_BGRA**, the blue component is first, followed by green, followed by red, followed by alpha; for **GL_ABGR_EXT** the order is alpha, blue, green, and then red. Floating-point values are converted directly to an internal floatingpoint 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. 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. The results are clamped to the range [0,1].

If **GL_MAP_COLOR** is true, each color component is scaled by the size of lookup table **GL_PIXEL_MAP_c_TO _c**, then replaced by the value that it references in that table. *c* is **R**, **G**, **B**, or **A**, respectively.

The resulting RGBA colors are then converted 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. 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_LUMINANCE

Each element is a single luminance value. The GL converts it to floating point, then assembles it 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. The GL converts it to floating point, then assembles it 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**).
GL_422_EXT	This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Y. The second component is Cb in the even pixels and Cr in the odd pixels. The Cb for each even pixel is used as the Cb value for that pixel and its neighbor to the right. The Cr in each odd pixel is used as the Cr value for that pixel and its neighbor to the left. (If the width of the image is odd, then the colors will be undefined in the rightmost column.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this conversion, the pixel is treated just as if it had been sent in as an RGB pixel.
GL_422_REV_EXT	This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Cb in the even pixels and Cr in the odd pixels. The second component is Y. The Cb for each even pixel is used as the Cb value for that pixel and its neighbor to the right. The Cr in each odd pixel is used as the Cr value for that pixel and its neighbor to the left. (If the width of the image is odd, then the colors will be undefined in the rightmost column.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this conversion, the pixel is treated just as if it ha d been sent in as an RGB pixel.
GL_422_AVERAGE_EXT	This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Y. The second component is Cb in the even pixels and Cr in the odd pixels. Each even pixel gets its Cb from itself, and its Cr from its neighbor to the right. Each odd pixel gets its Cb from the average of its left and right neighbor, and its Cr from the average of itself and its neighbor two to the right. (If the width of the image is odd, then the colors will be undefined in the rightmost column. If the neighbors to the right are not present for a given fragment, we use GL_422_EXT to compute that fragment.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this conversion, the pixel is treated just as if it had been sent in as an RGB pixel.

GL_422_REV_AVERAGE_EXT	This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Cb in the even pixels and Cr in the odd pixels. The second component is Y. Each even pixel gets its Cb from itself, and its Cr from its neighbor to the right. Each odd pixel gets its Cb from the average of its left and right neighbor, and its Cr from the average of itself and its neighbor two to the right. (If the width of the image is odd, then the colors will be undefined in the rightmost column. If the neighbors to the right are not present for a given fragment, we use GL_422_REV_EXT to compute that fragment.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this conversi on, the pixel is treated just as if it had been sent in as an RGB pixel.

For applications that store the texture at a certain resolution or in a certain format, request the resolution and format with *internalformat*. The GL will choose an internal representation that closely approximates that requested by *internalformat*, but it may not match exactly. (The representations specified by **GL_LUMINANCE**, **GL_LUMINANCE_ALPHA**, **GL_RGB**, and **GL_RGBA** must match exactly. The numeric values 1, 2, 3, and 4 may also be used to specify the above representations.)

Use the **GL_PROXY_TEXTURE_1D** target to try out a resolution and format. The implementation will update and recompute its best match for the requested storage resolution and format. To query this state, call **glGetTexLevelParameter**. If the texture cannot be accomodated, texture state is set to 0.

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.

Notes

GL_ABGR_EXT is only valid if the **GL_EXT_abgr** extension is defined.

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**.

GL_PROXY_TEXTURE_1D can only be used if the GL version is 1.1 or greater.

Internal formats other than 1, 2, 3, or 4 can only be used if the GL version is 1.1 or greater.

In GL version 1.1 or greater, *pixels* may be a null pointer. In this case texture memory is allocated to accomodate a texture of width *width*. You can then download subtextures to initialize the texture memory. The image is undefined if the user tries to apply an uninitialized portion of the texture image to a primitive.

Format of **GL_ABGR_EXT** is part of the _extname (EXT_abgr) extension, not part of the core GL command set.

Errors

GL_INVALID_ENUM is generated if *target* is not **GL_TEXTURE_1D** or **GL_PROXY_TEXTURE_1D**.

GL_INVALID_ENUM is generated if *format* is not an accepted format constant. Format constants other than **GL_STENCIL_INDEX** and **GL_DEPTH_COMPONENT** are accepted.

GL_INVALID_ENUM is generated if *type* is not a type constant.

GL_INVALID_ENUM is generated if *type* is **GL_BITMAP** and *format* is not **GL_COLOR_INDEX**.

GL_INVALID_VALUE is generated if *level* is less than zero.

GL_INVALID_VALUE may be generated if *level* is greater than log2max, where max is the returned value of **GL_MAX_TEXTURE_SIZE**.

GL_INVALID_VALUE is generated if *internalformat* is not 1, 2, 3, 4, or one of the accepted resolution and format symbolic constants.

GL_INVALID_VALUE is generated if *width* is less than zero or greater than 2 + **GL_MAX_TEXTURE_SIZE**, or if it cannot be represented as 2n + 2 **x** *border* for some integer value of n.

GL_INVALID_VALUE is generated if *border* is not 0 or 1.

GL_INVALID_OPERATION is generated if **glTexImage1D** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glGetTexImage

glIsEnabled with argument **GL_TEXTURE_1D**.

Related Information

The **glCopyTexImage1D** subroutine, **glDrawPixels** subroutine, **glFog** subroutine, **glPixelStore** subroutine, **glPixelTransfer** subroutine, **glTexEnv** subroutine, **glTexGen** subroutine, **glTexImage2D** subroutine, **glTexParameter** subroutine, **glTexSubImage1D** subroutine.

glTexImage2D Subroutine

Purpose

Specifies a two-dimensional (2D) texture image.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

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

Parameters

target	Specifies the target texture. Must be **GL_TEXTURE_2D** or **GL_PROXY_TEXTURE_2D**.
level	Specifies the level-of-detail number. Level 0 is the base image level. Level n is the nth mipmap reduction image.
internalformat	Specifies the number of color components in the texture. Must be 1, 2, 3, or 4, or one of the following symbolic constants: **GL_ABGR_EXT, GL_ALPHA**,**GL_ALPHA4**, **GL_ALPHA8**, **GL_ALPHA12**, **GL_ALPHA16**,**GL_LUMINANCE**, **GL_LUMINANCE4**, **GL_LUMINANCE8**,**GL_LUMINANCE12**, **GL_LUMINANCE16**, **GL_LUMINANCE_ALPHA**,**GL_LUMINANCE4_ALPHA4**, **GL_LUMINANCE6_ALPHA2**,**GL_LUMINANCE8_ALPHA8**, **GL_LUMINANCE12_ALPHA4**,**GL_LUMINANCE12_ALPHA12**, **GL_LUMINANCE16_ALPHA16**,**GL_INTENSITY**, **GL_INTENSITY4**, **GL_INTENSITY8**,**GL_INTENSITY12**, **GL_INTENSITY16**, **GL_R3_G3_B2**, **GL_RGB**,**GL_RGB4**, **GL_RGB5**, **GL_RGB8**, **GL_RGB10**,**GL_RGB12**, **GL_RGB16**, **GL_RGBA**, **GL_RGBA2**,**GL_RGBA4**, **GL_RGB5_A1**, **GL_RGBA8**, **GL_RGB10_A2**,**GL_RGBA12**, or **GL_RGBA16**.
width	Specifies the width of the texture image. Must be 2n + 2 x *border* for some integer n. All implementations support texture images that are at least 64 texels wide.
height	Specifies the height of the texture image. Must be 2m + 2 x *border* for some integer m. All implementations support texture images that are at least 64 texels high.
border	Specifies the width of the border. Must be either 0 or 1.
format	Specifies the format of the pixel data. Symbolic constants **GL_COLOR_INDEX**, **GL_RED**, **GL_GREEN**, **GL_BLUE**, **GL_ALPHA**, **GL_RGB**, **GL_RGBA**, **GL_BGR**, **GL_BGRA**, **GL_ABGR_EXT**, **GL_LUMINANCE**, **GL_422_EXT**, **GL_422_REV_EXT**, **GL_422_AVERAGE_EXT**, **GL_422_REV_AVERAGE_EXT**, and **GL_LUMINANCE_ALPHA** are accepted.
type	Specifies the data type for *Pixels*. Symbolic constants **GL_UNSIGNED_BYTE**, **GL_BYTE**, **GL_BITMAP**, **GL_UNSIGNED_SHORT**, **GL_SHORT**, **GL_UNSIGNED_INT**, **GL_INT**, **GL_FLOAT**, **GL_UNSIGNED_BYTE_3_3_2**, **GL_UNSIGNED_BYTE_2_3_3_REV**, **GL_UNSIGNED_SHORT_5_6_5**, **GL_UNSIGNED_SHORT_5_6_5_REV**, **GL_UNSIGNED_SHORT_4_4_4_4**, **GL_UNSIGNED_SHORT_4_4_4_4_REV**, **GL_UNSIGNED_SHORT_5_5_5_1**, **GL_UNSIGNED_SHORT_1_5_5_5_REV**, **GL_UNSIGNED_INT_8_8_8_8**, **GL_UNSIGNED_INT_8_8_8_8_REV**, **GL_UNSIGNED_INT_10_10_10_2**, and **GL_UNSIGNED_INT_2_10_10_10_REV**, are accepted.
pixels	Specifies a pointer to the image data in memory.

Description

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. To enable and disable two-dimensional texturing, call **glEnable** and **glDisable** with argument **GL_TEXTURE_2D**.

To define texture images, call **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 how the image is represented in memory. They are identical to the pixel formats used for **glDrawPixels**.

If *target* is **GL_PROXY_TEXTURE_2D** no data is read from pixels, but all of the texture image state is recalculated, checked for consistency, and checked against the implementation's capabilities. If the implementation cannot handle a texture of the requested texture size, it sets all of the image state to 0, but does not generate an error (see **glGetError**). To query for an entire mipmap array, use an image array level greater than or equal to 1.

If *target* is **GL_TEXTURE_2D**, 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**).

The first element corresponds to the lower-left corner of the texture image. Subsequent elements progress left-to-right through the remaining texels in the lowest row of the texture image, and then in successively higher rows of the texture image. The final element corresponds to the upper-right corner of the texture image.

The *format* parameter determines the composition of each element in pixels. It can assume one of 16 symbolic values:

GL_COLOR_INDEX
Each element is a single value, a color index. The GL converts it to fixed point (with an unspecified number of zero 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. The GL converts it to floating point and assembles it 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. The GL converts it to floating point and assembles it 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. The GL converts it to floating point and assembles it 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. The GL converts it to floating point and assembles it 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. The GL converts it to floating point and assembles it 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 contains all four components. Each *component is 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
Each pixel is a three-component group, blue first, followed by green, followed by red. Each component is converted to the internal floating-point format in the same way as the blue, green, and red components of an BGRA pixel are. The color triple is converted to an BGRA pixel with alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an BGRA pixel.

GL_BGRA
: Each pixel is a four-component group, blue first, followed by green, followed by red, followed by alpha. Floating-point values are converted 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 are mapped similarly: the largest integer value maps to 1.0, and 0 maps to 0.0. The resulting floating-point color values are then multiplied by **GL_*c*_SCALE** and added to **GL_*c*_BIAS**, where *c* is **BLUE**, **GREEN**, **RED**, and **ALPHA** for the respective color components. The results are clamped to the range [0,1].

  If **GL_MAP_COLOR** is True, each color component is scaled by the size of the lookup table **GL_PIXEL_MAP_*c*_TO_*c***, then replaced by the value that it references in that table. *c* is **B**, **G**, **R**, or **A**, respectively.

  The resulting BGRA colors are then converted 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 $x_n = x_r + n \bmod Width$ and $y_n = y_r + [n/Width]$, where (x_r, y_r) is the current raster position. 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_ABGR_EXT
: Each pixel is a four-component group: for **GL_RGBA**, the red component is first, followed by green, followed by blue, followed by alpha; for **GL_BGRA**, the blue component is first, followed by green, followed by red, followed by alpha; for **GL_ABGR_EXT** the order is alpha, blue, green, and then red. Floating-point values are converted directly to an internal floatingpoint 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. 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. The results are clamped to the range [0,1].

  If **GL_MAP_COLOR** is true, each color component is scaled by the size of lookup table **GL_PIXEL_MAP_c_TO _c**, then replaced by the value that it references in that table. *c* is **R**, **G**, **B**, or **A**, respectively.

  The resulting RGBA colors are then converted 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. 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_LUMINANCE
: Each element is a single luminance value. The GL converts it to floating point, then assembles it 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. The GL converts it to floating point, then assembles it 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**).
GL_422_EXT	This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Y. The second component is Cb in the even pixels and Cr in the odd pixels. The Cb for each even pixel is used as the Cb value for that pixel and its neighbor to the right. The Cr in each odd pixel is used as the Cr value for that pixel and its neighbor to the left. (If the width of the image is odd, then the colors will be undefined in the rightmost column.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this conversion, the pixel is treated just as if it had been sent in as an RGB pixel.
GL_422_AVERAGE_EXT	This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Y. The second component is Cb in the even pixels and Cr in the odd pixels. Each even pixel gets its Cb from itself, and its Cr from its neighbor to the right. Each odd pixel gets its Cb from the average of its left and right neighbor, and its Cr from the average of itself and its neighbor two to the right. (If the width of the image is odd, then the colors will be undefined in the rightmost column. If the neighbors to the right are not present for a given fragment, we use GL_422_EXT to compute that fragment.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this conversion, the pixel is treated just as if it had been sent in as an RGB pixel.
GL_422_REV_AVERAGE_EXT	This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Cb in the even pixels and Cr in the odd pixels. The second component is Y. Each even pixel gets its Cb from itself, and its Cr from its neighbor to the right. Each odd pixel gets its Cb from the average of its left and right neighbor, and its Cr from the average of itself and its neighbor two to the right. (If the width of the image is odd, then the colors will be undefined in the rightmost column. If the neighbors to the right are not present for a given fragment, we use GL_422_REV_EXT to compute that fragment.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this convers ion, the pixel is treated just as if it had been sent in as an RGB pixel.

Refer to the **glDrawPixels** subroutine for a description of the acceptable values for the *type* parameter. For applications that store the texture at a certain resolution or in a certain format, request the resolution and format with *internalformat*. The GL will choose an internal representation that closely approximates that requested by *internalformat*, but it may not match exactly. (The representations specified by **GL_LUMINANCE**, **GL_LUMINANCE_ALPHA**, **GL_RGB**, and **GL_RGBA** must match exactly. The numeric values 1, 2, 3, and 4 may also be used to specify the above representations.)

Use the **GL_PROXY_TEXTURE_2D** target to try out a resolution and format. The implementation will update and recompute its best match for the requested storage resolution and format. To then query this state, call **glGetTexLevelParameter**. If the texture cannot be accomodated, texture state is set to 0.

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.

Notes

GL_ABGR_EXT is only valid if the **GL_EXT_abgr** extension is defined.

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**.

The **GL_PROXY_TEXTURE_2D** target are only available if the GL version is 1.1 or greater.

Internal formats other than 1, 2, 3, or 4 may only be used if the GL version is 1.1 or greater.

In GL version 1.1 or greater, pixels may be a null pointer. In this case texture memory is allocated to accomodate a texture of width *width* and height *height*. You can then download subtextures to initialize this texture memory. The image is undefined if the user tries to apply an uninitialized portion of the texture image to a primitive.

Format of **GL_ABGR_EXT** is part of the _extname (EXT_abgr) extension, not part of the core GL command set.

Errors

GL_INVALID_ENUM is generated if *target* is not **GL_TEXTURE_2D** or **GL_PROXY_TEXTURE_2D**.

GL_INVALID_ENUM is generated if *format* is not an accepted format constant. Format constants other than **GL_STENCIL_INDEX** and **GL_DEPTH_COMPONENT** are accepted.

GL_INVALID_ENUM is generated if *type* is not a type constant.

GL_INVALID_ENUM is generated if *type* is **GL_BITMAP** and *format* is not **GL_COLOR_INDEX**.

GL_INVALID_VALUE is generated if l*evel* is less than zero.

GL_INVALID_VALUE may be generated if *level* is greater than log2(max), where max is the returned value of **GL_MAX_TEXTURE_SIZE**.

GL_INVALID_VALUE is generated if *internalformat* is not 1, 2, 3, 4, or one of the accepted resolution and format symbolic constants.

GL_INVALID_VALUE is generated if *width* or *height* is less than zero or greater than 2 + **GL_MAX_TEXTURE_SIZE**, or if either cannot be represented as 2k + 2 x *border* for some integer value of k.

GL_INVALID_VALUE is generated if *border* is not 0 or 1.

GL_INVALID_OPERATION is generated if **glTexImage2D** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glGetTexImage

glIsEnabled with argument **GL_TEXTURE_2D**.

Related Information

The **glCopyTexImage2D** subroutine, **glDrawPixels** subroutine, **glFog** subroutine, **glPixelStore** subroutine, **glPixelTransfer** subroutine, **glTexEnv** subroutine, **glTexGen** subroutine, **glTexImage1D** subroutine, **glTexParameter** subroutine, **glTexSubImage2D** subroutine.

---

glTexImage3D Subroutine

Purpose

Specifies a three-dimensional (3D) texture subimage.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glTexImage3D  (GLenum  target,
                    GLint  level,
                    GLint  internalformat,
                    GLsizei  width,
                    GLsizei  height,
                    GLsizei  depth,
                    GLint  border,
                    GLenum  format,
                    GLenum  type,
                    const GLvoid * pixels)
```

Parameters

target	Specifies the target texture. Must be **GL_TEXTURE_3D** or **GL_PROXY_TEXTURE_3D**.
level	Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.
internalformat	Specifies the number of color components in the texture. Must be 1, 2, 3, or 4, or one of the following symbolic constants: **GL_ABGR_EXT, GL_ALPHA**, **GL_ALPHA4**, **GL_ALPHA8**, **GL_ALPHA12**, **GL_ALPHA16**, **GL_LUMINANCE**, **GL_LUMINANCE4**, **GL_LUMINANCE8**, **GL_LUMINANCE12**, **GL_LUMINANCE16**, **GL_LUMINANCE_ALPHA**, **GL_LUMINANCE4_ALPHA4**, **GL_LUMINANCE6_ALPHA2**, **GL_LUMINANCE8_ALPHA8**, **GL_LUMINANCE12_ALPHA4**, **GL_LUMINANCE12_ALPHA12**, **GL_LUMINANCE16_ALPHA16**, **GL_INTENSITY**, **GL_INTENSITY4**, **GL_INTENSITY8**, **GL_INTENSITY12**, **GL_INTENSITY16**, **GL_R3_G3_B2**, **GL_RGB**, **GL_RGB4**, **GL_RGB5**, **GL_RGB8**, **GL_RGB10**, **GL_RGB12**, **GL_RGB16**, **GL_RGBA**, **GL_RGBA2**, **GL_RGBA4**, **GL_RGB5_A1**, **GL_RGBA8**, **GL_RGB10_A2**, **GL_RGBA12**, or **GL_RGBA16**.
width	Specifies the width of the texture image. Must be 2n + 2 **x** *border* for some integer n.
height	Specifies the height of the texture image. Must be 2m + 2 **x** *border* for some integer m.
depth	Specifies the depth of the texture image. Must be 2l + 2 **x** *border* for some integer l.
border	Specifies the width of the border. Must be either 0 or 1.

format	Specifies the format of the pixel data. Symbolic constants **GL_COLOR_INDEX**, **GL_RED**, **GL_GREEN**, **GL_BLUE**, **GL_ALPHA**, **GL_RGB**, **GL_RGBA**, **GL_BGR**, **GL_BGRA**, **GL_ABGR_EXT**, **GL_LUMINANCE**, **GL_422_EXT**, **GL_422_REV_EXT**, **GL_422_AVERAGE_EXT**, **GL_422_REV_AVERAGE_EXT**, and accepted.
type	Specifies the data type for *Pixels*. Symbolic constants **GL_UNSIGNED_BYTE**, **GL_BYTE**, **GL_BITMAP**, **GL_UNSIGNED_SHORT**, **GL_SHORT**, **GL_UNSIGNED_INT**, **GL_INT**, **GL_FLOAT**, **GL_UNSIGNED_BYTE_3_3_2**, **GL_UNSIGNED_BYTE_2_3_3_REV**, **GL_UNSIGNED_SHORT_5_6_5**, **GL_UNSIGNED_SHORT_5_6_5_REV**, **GL_UNSIGNED_SHORT_4_4_4_4**, **GL_UNSIGNED_SHORT_4_4_4_4_REV**, **GL_UNSIGNED_SHORT_5_5_5_1**, **GL_UNSIGNED_SHORT_1_5_5_5_REV**, **GL_UNSIGNED_INT_8_8_8_8**, **GL_UNSIGNED_INT_8_8_8_8_REV**, **GL_UNSIGNED_INT_10_10_10_2**, and **GL_UNSIGNED_INT_2_10_10_10_REV** are accepted.
pixels	Specifies a pointer to the image data in memory.

Description

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. To enable and disable three-dimensional texturing, call **glEnable** and **glDisable** with argument **GL_TEXTURE_3D**.

To define 3D texture images, call **glTexImage3D**. The arguments describe the parameters of the texture image, such as height, width, depth, width of the border, level-of-detail number (see **glTexParameter**), and number of color components provided. The last three arguments describe how the image is represented in memory; they are identical to the pixel formats used for **glDrawPixels**.

If *target* is **GL_PROXY_TEXTURE_3D** no data is read from *pixels*, but all of the texture image state is recalculated, checked for consistency, and checked against the implementation's capabilities. If the implementation cannot handle a texture of the requested texture size, it sets all of the image state to 0, but does not generate an error (see **glGetError**). To query for an entire mipmap array, use an image array level greater than or equal to 1.

If *target* is **GL_TEXTURE_3D**, 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**).

The first element corresponds to the lower-left corner of the texture image. Subsequent elements progress left-to-right through the remaining texels in the lowest row of the texture image, and then in successively higher rows of the texture image. The final element corresponds to the upper-right corner of the texture image.

The *format* parameter determines the composition of each element in *pixels*. It can assume one of 16 symbolic values:

GL_COLOR_INDEX	Each element is a single value, a color index. The GL converts it to fixed point (with an unspecified number of zero 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. The GL converts it to floating point and assembles it 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. The GL converts it to floating point and assembles it 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. The GL converts it to floating point and assembles it 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. The GL converts it to floating point and assembles it 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. The GL converts it to floating point and assembles it 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 contains all four components. Each *component is 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 Each pixel is a three-component group, blue first, followed by green, followed by red. Each component is converted to the internal floating-point format in the same way as the blue, green, and red components of an BGRA pixel are. The color triple is converted to an BGRA pixel with alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an BGRA pixel.

GL_BGRA Each pixel is a four-component group, blue first, followed by green, followed by red, followed by alpha. Floating-point values are converted 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 are mapped similarly: the largest integer value maps to 1.0, and 0 maps to 0.0. The resulting floating-point color values are then multiplied by **GL_*c*_SCALE** and added to **GL_*c*_BIAS**, where *c* is **BLUE**, **GREEN**, **RED**, and **ALPHA** for the respective color components. The results are clamped to the range [0,1].

If **GL_MAP_COLOR** is True, each color component is scaled by the size of the lookup table **GL_PIXEL_MAP_*c*_TO_*c***, then replaced by the value that it references in that table. *c* is **B**, **G**, **R**, or **A**, respectively.

The resulting BGRA colors are then converted 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 $x_n = x_r + n \bmod Width$ and $y_n = y_r + [n/Width]$, where (x_r, y_r) is the current raster position. 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_ABGR_EXT

Each pixel is a four-component group: for **GL_RGBA**, the red component is first, followed by green, followed by blue, followed by alpha; for **GL_BGRA**, the blue component is first, followed by green, followed by red, followed by alpha; for **GL_ABGR_EXT** the order is alpha, blue, green, and then red. Floating-point values are converted directly to an internal floatingpoint 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. 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. The results are clamped to the range [0,1].

If **GL_MAP_COLOR** is true, each color component is scaled by the size of lookup table **GL_PIXEL_MAP_c_TO _c**, then replaced by the value that it references in that table. *c* is **R**, **G**, **B**, or **A**, respectively.

The resulting RGBA colors are then converted 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

$$x_n = x_r + n \bmod width$$

$$y_n = y_r + \left\lfloor \frac{n}{width} \right\rfloor$$

where (x_r,y_r) is the current raster position. 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_LUMINANCE

Each element is a single luminance value. The GL converts it to floating point, then assembles it 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. The GL converts it to floating point, then assembles it 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**).

GL_422_EXT

This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Y. The second component is Cb in the even pixels and Cr in the odd pixels. The Cb for each even pixel is used as the Cb value for that pixel and its neighbor to the right. The Cr in each odd pixel is used as the Cr value for that pixel and its neighbor to the left. (If the width of the image is odd, then the colors will be undefined in the rightmost column.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this conversion, the pixel is treated just as if it had been sent in as an RGB pixel.

GL_422_REV_EXT	This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Cb in the even pixels and Cr in the odd pixels. The second component is Y. The Cb for each even pixel is used as the Cb value for that pixel and its neighbor to the right. The Cr in each odd pixel is used as the Cr value for that pixel and its neighbor to the left. (If the width of the image is odd, then the colors will be undefined in the rightmost column.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this conversion, the pixel is treated just as if it had been sent in as an RGB pixel.
GL_422_AVERAGE_EXT	This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Y. The second component is Cb in the even pixels and Cr in the odd pixels. Each even pixel gets its Cb from itself, and its Cr from its neighbor to the right. Each odd pixel gets its Cb from the average of its left and right neighbor, and its Cr from the average of itself and its neighbor two to the right. (If the width of the image is odd, then the colors will be undefined in the rightmost column. If the neighbors to the right are not present for a given fragment, we use GL_422_EXT to compute that fragment.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this conversion, the pixel is treated just as if it had been sent in as an RGB pixel.
GL_422_REV_AVERAGE_EXT	This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Cb in the even pixels and Cr in the odd pixels. The second component is Y. Each even pixel gets its Cb from itself, and its Cr from its neighbor to the right. Each odd pixel gets its Cb from the average of its left and right neighbor, and its Cr from the average of itself and its neighbor two to the right. (If the width of the image is odd, then the colors will be undefined in the rightmost column. If the neighbors to the right are not present for a given fragment, we use GL_422_REV_EXT to compute that fragment.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this conversion, the pixel is treated just as if it had been sent in as an RGB pixel.

Refer to the **glDrawPixels** reference page for a description of the acceptable values for the type parameter. If an application must store the texture at a certain resolution or in a certain format, use *internalformat* to request the resolution and format. The GL will choose an internal representation that closely approximates that requested by *internalformat*, but it may not match exactly. (The representations specified by **GL_LUMINANCE**, **GL_LUMINANCE_ALPHA**, **GL_RGB**, and **GL_RGBA** must match exactly. The numeric values 1, 2, 3, and 4 may also be used to specify the above representations.)

Use the **GL_PROXY_TEXTURE_3D** target to try out a resolution and format. The implementation will update and recompute its best match for the requested storage resolution and format. To then query this state, call **glGetTexLevelParameter**. If the texture cannot be accommodated, texture state is set to 0.

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.

Notes

GL_ABGR_EXT is only valid if the **GL_EXT_abgr** extension is defined.

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**.

Internal formats other than 1, 2, 3, or 4 may only be used if the GL version is 1.2 or greater.

In GL version 1.2 or greater, *pixels* may be a null pointer. In this case texture memory is allocated to accomodate a texture of width *width* and height *height*. You can then download subtextures to initialize this texture memory. The image is undefined if the user tries to apply an uninitialized portion of the texture image to a primitive.

Errors

GL_INVALID_ENUM is generated if *target* is not **GL_TEXTURE_3D** or **GL_PROXY_TEXTURE_3D**.

GL_INVALID_ENUM is generated if *format* is not an accepted format constant. Format constants other than **GL_STENCIL_INDEX** and **GL_DEPTH_COMPONENT** are accepted.

GL_INVALID_ENUM is generated if *type* is not a type constant.

GL_INVALID_ENUM is generated if *type* is **GL_BITMAP** and *format* is not **GL_COLOR_INDEX**.

GL_INVALID_VALUE is generated if *level* is less than zero.

GL_INVALID_VALUE may be generated if *level* is greater than log2(max), where max is the returned value of **GL_MAX_3D_TEXTURE_SIZE**.

GL_INVALID_VALUE is generated if *internalformat* is not 1, 2, 3, 4, or one of the accepted resolution and format symbolic constants.

GL_INVALID_VALUE is generated if *width*, *height*, or *depth* is less than zero or greater than 2 + **GL_MAX_3D_TEXTURE_SIZE**, or if either cannot be represented as 2k + 2 **x** *border* for some integer value of k.

GL_INVALID_VALUE is generated if *border* is not 0 or 1.

GL_INVALID_OPERATION is generated if **glTexImage3D** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glGetTexImage

glIsEnabled with argument **GL_TEXTURE_3D**

Related Information

The **glCopyTexSubImage3D** subroutine, **glDrawPixels** subroutine, **glFog** subroutine, **glPixelStore** subroutine, **glPixelTransfe**r subroutine, **glTexEnv** subroutine, **glTexGen** subroutine, **glTexImage1D** subroutine, **glTexParameter** subroutine, **glTexImage2D** subroutine.

glTexImage3DEXT Subroutine

Purpose

Specifies a three-dimensional (3D) texture subimage.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glTexImage3DEXT(GLenum  target,
         GLint  level,
         GLint  internalformat,
         GLsizei  width,
         GLsizei  height,
         GLsizei  depth,
         GLint  border,
         GLenum  format,
         GLenum  type,
         const GLvoid * pixels)
```

Parameters

target	Specifies the target texture. Must be **GL_TEXTURE_3D_EXT** or **GL_PROXY_TEXTURE_3D_EXT**.
level	Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.
internalformat	Specifies the number of color components in the texture. Must be 1, 2, 3, or 4, or one of the following symbolic constants: **GL_ABGR_EXT, GL_ALPHA**, **GL_ALPHA4**, **GL_ALPHA8**, **GL_ALPHA12**, **GL_ALPHA16**, **GL_LUMINANCE**, **GL_LUMINANCE4**, **GL_LUMINANCE8**, **GL_LUMINANCE12**, **GL_LUMINANCE16**, **GL_LUMINANCE_ALPHA**, **GL_LUMINANCE4_ALPHA4**, **GL_LUMINANCE6_ALPHA2**, **GL_LUMINANCE8_ALPHA8**, **GL_LUMINANCE12_ALPHA4**, **GL_LUMINANCE12_ALPHA12**, **GL_LUMINANCE16_ALPHA16**, **GL_INTENSITY**, **GL_INTENSITY4**, **GL_INTENSITY8**, **GL_INTENSITY12**, **GL_INTENSITY16**, **GL_R3_G3_B2**, **GL_RGB**, **GL_RGB4**, **GL_RGB5**, **GL_RGB8**, **GL_RGB10**, **GL_RGB12**, **GL_RGB16**, **GL_RGBA**, **GL_RGBA2**, **GL_RGBA4**, **GL_RGB5_A1**, **GL_RGBA8**, **GL_RGB10_A2**, **GL_RGBA12**, or **GL_RGBA16**.
width	Specifies the width of the texture image. Must be 2n + 2 **x** *border* for some integer n.
height	Specifies the height of the texture image. Must be 2m + 2 **x** *border* for some integer m.
depth	Specifies the depth of the texture image. Must be 2l + 2 **x** *border* for some integer l.
border	Specifies the width of the border. Must be either 0 or 1.
format	Specifies the format of the pixel data. The following symbolic values are accepted: **GL_COLOR_INDEX**, **GL_RED, GL_GREEN**, **GL_BLUE**, **GL_ALPHA**, **GL_RGB, GL_RGBA**, **GL_LUMINANCE**, **GL_422_EXT**, **GL_422_REV_EXT**, **GL_422_AVERAGE_EXT**, **GL_422_REV_AVERAGE_EXT**, and **GL_LUMINANCE_ALPHA**.
type	Specifies 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	Specifies a pointer to the image data in memory.

Description

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. To enable and disable three-dimensional texturing, call **glEnable** and **glDisable** with argument **GL_TEXTURE_3D_EXT**.

To define 3D texture images, call **glTexImage3DEXT**. The arguments describe the parameters of the texture image, such as height, width, depth, width of the border, level-of-detail number (see **glTexParameter**), and number of color components provided. The last three arguments describe how the image is represented in memory; they are identical to the pixel formats used for **glDrawPixels**.

If *target* is **GL_PROXY_TEXTURE_3D_EXT** no data is read from *pixels*, but all of the texture image state is recalculated, checked for consistency, and checked against the implementation's capabilities. If the implementation cannot handle a texture of the requested texture size, it sets all of the image state to 0, but does not generate an error (see **glGetError**). To query for an entire mipmap array, use an image array level greater than or equal to 1.

If *target* is **GL_TEXTURE_3D_EXT**, 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**).

The first element corresponds to the lower-left corner of the texture image. Subsequent elements progress left-to-right through the remaining texels in the lowest row of the texture image, and then in successively higher rows of the texture image. The final element corresponds to the upper-right corner of the texture image.

The *format* parameter determines the composition of each element in *pixels*. It can assume one of 16 symbolic values:

GL_COLOR_INDEX
Each element is a single value, a color index. The GL converts it to fixed point (with an unspecified number of zero 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. The GL converts it to floating point and assembles it 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. The GL converts it to floating point and assembles it 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. The GL converts it to floating point and assembles it 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. The GL converts it to floating point and assembles it 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. The GL converts it to floating point and assembles it 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 contains all four components. Each *component is 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. The GL converts it to floating point, then assembles it 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. The GL converts it to floating point, then assembles it 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**).
GL_422_EXT	This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Y. The second component is Cb in the even pixels and Cr in the odd pixels. The Cb for each even pixel is used as the Cb value for that pixel and its neighbor to the right. The Cr in each odd pixel is used as the Cr value for that pixel and its neighbor to the left. (If the width of the image is odd, then the colors will be undefined in the rightmost column.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this conversion, the pixel is treated just as if it had been sent in as an RGB pixel.
GL_422_REV_EXT	This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Cb in the even pixels and Cr in the odd pixels. The second component is Y. The Cb for each even pixel is used as the Cb value for that pixel and its neighbor to the right. The Cr in each odd pixel is used as the Cr value for that pixel and its neighbor to the left. (If the width of the image is odd, then the colors will be undefined in the rightmost column.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this conversion, the pixel is treated just as if it had been sent in as an RGB pixel.

GL_422_AVERAGE_EXT	This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Y. The second component is Cb in the even pixels and Cr in the odd pixels. Each even pixel gets its Cb from itself, and its Cr from its neighbor to the right. Each odd pixel gets its Cb from the average of its left and right neighbor, and its Cr from the average of itself and its neighbor two to the right. (If the width of the image is odd, then the colors will be undefined in the rightmost column. If the neighbors to the right are not present for a given fragment, we use GL_422_EXT to compute that fragment.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this conversion, the pixel is treated just as if it had been sent in as an RGB pixel.
GL_422_REV_AVERAGE_EXT	This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Cb in the even pixels and Cr in the odd pixels. The second component is Y. Each even pixel gets its Cb from itself, and its Cr from its neighbor to the right. Each odd pixel gets its Cb from the average of its left and right neighbor, and its Cr from the average of itself and its neighbor two to the right. (If the width of the image is odd, then the colors will be undefined in the rightmost column. If the neighbors to the right are not present for a given fragment, we use GL_422_REV_EXT to compute that fragment.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this convers ion, the pixel is treated just as if it had been sent in as an RGB pixel.

Refer to the **glDrawPixels** reference page for a description of the acceptable values for the type parameter. If an application must store the texture at a certain resolution or in a certain format, use *internalformat* to request the resolution and format. The GL will choose an internal representation that closely approximates that requested by *internalformat*, but it may not match exactly. (The representations specified by **GL_LUMINANCE**, **GL_LUMINANCE_ALPHA**, **GL_RGB**, and **GL_RGBA** must match exactly. The numeric values 1, 2, 3, and 4 may also be used to specify the above representations.)

Use the **GL_PROXY_TEXTURE_3D_EXT** target to try out a resolution and format. The implementation will update and recompute its best match for the requested storage resolution and format. To then query this state, call **glGetTexLevelParameter**. If the texture cannot be accomodated, texture state is set to 0.

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.

Notes

GL_ABGR_EXT is only valid if the **GL_EXT_abgr** extension is defined.

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**.

The **glTexImage3DEXT** subroutine and **GL_PROXY_TEXTURE_3D_EXT** are available only if the **EXT_texture3D** extension is supported.

Internal formats other than 1, 2, 3, or 4 may only be used if the GL version is 1.1 or greater.

In GL version 1.1 or greater, *pixels* may be a null pointer. In this case texture memory is allocated to accomodate a texture of width *width* and height *height*. You can then download subtextures to initialize this texture memory. The image is undefined if the user tries to apply an uninitialized portion of the texture image to a primitive.

Format of **GL_ABGR_EXT** is part of the _extname (EXT_abgr) extension, not part of the core GL command set.

Errors

GL_INVALID_ENUM is generated if *target* is not **GL_TEXTURE_3D_EXT** or **GL_PROXY_TEXTURE_3D_EXT**.

GL_INVALID_ENUM is generated if *format* is not an accepted format constant. Format constants other than **GL_STENCIL_INDEX** and **GL_DEPTH_COMPONENT** are accepted.

GL_INVALID_ENUM is generated if *type* is not a type constant.

GL_INVALID_ENUM is generated if *type* is **GL_BITMAP** and *format* is not **GL_COLOR_INDEX**.

GL_INVALID_VALUE is generated if *level* is less than zero.

GL_INVALID_VALUE may be generated if *level* is greater than log2(max), where max is the returned value of **GL_MAX_3D_TEXTURE_SIZE_EXT**.

GL_INVALID_VALUE is generated if *internalformat* is not 1, 2, 3, 4, or one of the accepted resolution and format symbolic constants.

GL_INVALID_VALUE is generated if *width*, *height*, or *depth* is less than zero or greater than 2 + **GL_MAX_3D_TEXTURE_SIZE_EXT**, or if either cannot be represented as 2k + 2 **x** *border* for some integer value of k.

GL_INVALID_VALUE is generated if *border* is not 0 or 1.

GL_INVALID_OPERATION is generated if **glTexImage3DEXT** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glGetTexImage

glIsEnabled with argument **GL_TEXTURE_3D_EXT**

Related Information

The **glCopyTexSubImage3DEXT** subroutine, **glDrawPixels** subroutine, **glFog** subroutine, **glPixelStore** subroutine, **glPixelTransfer** subroutine, **glTexEnv** subroutine, **glTexGen** subroutine, **glTexImage1D** subroutine, **glTexParameter** subroutine, **glTexImage2D** subroutine.

glTexParameter Subroutine

Purpose

Sets texture parameters.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glTexParameterf(GLenum  target,
                     GLenum  pname,
                     GLfloat  param)

void glTexParameteri(GLenum  target,
                     GLenum  pname,
                     GLint  param)

void glTexParameterfv(GLenum  target,
                      GLenum  pname,
                      const GLfloat * params)

void glTexParameteriv(GLenum  target,
                      GLenum  pname,
                      const GLint * params)
```

Parameters

glTexParameterf or glTexParameteri

target Specifies the target texture, which must be either **GL_TEXTURE_1D**, **GL_TEXTURE_2D**, **GL_TEXTURE_3D**, or **GL_TEXTURE_3D_EXT.**

pname Specifies the symbolic name of a single-valued texture parameter. The *pname* parameter can be one of the following: **GL_TEXTURE_MIN_FILTER**, **GL_TEXTURE_MAG_FILTER**, **GL_TEXTURE_WRAP_S**, **GL_TEXTURE_WRAP_T**, **GL_TEXTURE_WRAP_R**, **GL_TEXTURE_PRIORITY**, **GL_TEXTURE_MIN_LOD**, **GL_TEXTURE_MAX_LOD**, **GL_TEXTURE_BASE_LEVEL**, **GL_TEXTURE_MAX_LEVEL**, **GL_TEXTURE_MAX_ANISOTROPY_EXT**.

param Specifies the value of *pname*.

glTexParameterfv or glTexParameteriv

target Specifies the target texture, which must be either **GL_TEXTURE_1D**, **GL_TEXTURE_2D**, **GL_TEXTURE_3D**, or **GL_TEXTURE_3D_EXT**.

pname Specifies the symbolic name of a texture parameter. The *pname* parameter can be one of the following: **GL_TEXTURE_MIN_FILTER**, **GL_TEXTURE_MAG_FILTER**, **GL_TEXTURE_WRAP_S**, **GL_TEXTURE_WRAP_T**, **GL_TEXTURE_WRAP_R**, **GL_TEXTURE_BORDER_COLOR**, **GL_TEXTURE_PRIORITY**, **GL_TEXTURE_MIN_LOD**, **GL_TEXTURE_MAX_LOD**, **GL_TEXTURE_BASE_LEVEL**, **GL_TEXTURE_MAX_LEVEL**, **GL_TEXTURE_MAX_ANISOTROPY_EXT** .

params Specifies a pointer to an array where the value or values of *pname* are stored.

Description

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-dimensional (1D) or two-dimensional (2D) image and a set of parameters that determine how samples are derived from the image.

The **glTexParameter** subroutine 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**, **GL_TEXTURE_2D**, **GL_TEXTURE_3D**, or **GL_TEXTURE_3D_EXT**. The following symbols are accepted in *pname*:

GL_TEXTURE_BORDER_COLOR
Sets a border color. The *params* parameter contains four values that comprise the red, green, blue, alpha (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).

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 2n x 2m there are **max**(*n,m*)+1 mipmaps. The first mipmap is the original texture, with dimensions 2n x 2m. Each subsequent mipmap has dimensions 2k-1 x 2l-1 where 2k x 2l are the dimensions of the previous mipmap, until either *k*=0 or *l*=0. At that point, subsequent mipmaps have the dimension 1 x 2l-1 or 2k-1 x 1 until the final mipmap, which has the dimension 1 x 1. Mipmaps are defined using the **glTexImage1D**, **glTexImage2D**, or **glTexImage3DEXT** subroutines with the level-of-detail argument indicating the order of the mipmaps. Level 0 is the original texture; level **max**(*n,m*) is the final 1 x 1 mipmap.

The *paramrs* 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** and **GL_TEXTURE_WRAP_T**, and on the exact mapping.

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.

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**.

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_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 initial value of **GL_TEXTURE_MAG_FILTER** is **GL_LINEAR**.

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** and **GL_TEXTURE_WRAP_T**, and on the exact mapping.

GL_NEAREST is generally faster than **GL_LINEAR**, but 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_TEXTURE_PRIORITY
Specifies the texture residence priority of the currently bound texture. Permissible values are in the range [0.0, 1.0]. See **glPrioritizeTextures** and **glBindTexture** for more information.

GL_TEXTURE_MAX_LOD
Specifies for the texture the maximum level of detail of the image array. Any floating-point value is permissable. Supported in OpenGL 1.2 and later.

GL_TEXTURE_MIN_LOD
Specifies for the texture the minimum level of detail of the image array. Any floating-point value is permissable. Supported in OpenGL 1.2 and later.

GL_TEXTURE_BASE_LEVEL
Specifies for the texture the base array level. Any non-negative integer value is permissable. Supported in OpenGL 1.2 and later.

GL_TEXTURE_MAX_LEVEL
Specifies for the texture the maximum array level. Any non-negative integer value is permissable. Supported in OpenGL 1.2 and later.

GL_TEXTURE_WRAP_R
Sets the wrap parameter for texture coordinate *r* to either **GL_CLAMP**, **GL_CLAMP_NODRAW_IBM**, **GL_CLAMP_TO_EDGE**, or **GL_REPEAT**. See the discussion under **GL_TEXTURE_WRAP_S**. Initially, **GL_TEXTURE_WRAP_R** is set to **GL_REPEAT**.

GL_TEXTURE_WRAP_S
Sets the wrap parameter for texture coordinate *s* to either **GL_CLAMP**, **GL_CLAMP_NODRAW_IBM**, **GL_CLAMP_TO_EDGE**, or **GL_REPEAT**. **GL_CLAMP** causes *s*, *t*, or *r* 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_CLAMP_NODRAW_IBM** clamps texture coordinates at all mipmap levels such that any pixels whose corresponding texture coordinate falls outside the specified texture map are not drawn at all. **GL_CLAMP_TO_EDGE** clamps texture coordinates at all mipmap levels such that the texture filter never samples a border texel. The color returned when clamping is derived only from texels at the edge of the texture image. **GL_REPEAT** causes the integer part of the *s*, *t*, or *r* coordinates to be ignored; the GL 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**,

GL_CLAMP_NODRAW_IBM, **GL_CLAMP_TO_EDGE**, or **GL_REPEAT**. See the discussion under **GL_TEXTURE_WRAP_S**. Initially, **GL_TEXTURE_WRAP_T** is set to **GL_REPEAT**.

GL_TEXTURE_MAX_ANISOTROPIC_EXT
Sets the maximum degree of anisotropy for this texture map. Initially, **GL_TEXTURE_MAX_ANISOTROPIC_EXT** is set to 1.0.

Notes

Suppose that a program has enabled texturing (by calling **glEnable** with argument **GL_TEXTURE_1D**, **GL_TEXTURE_2D**, or **GL_TEXTURE_3D**) and has set **GL_TEXTURE_MIN_FILTER** to one of the functions that requires a mipmap. If either the dimensions of the texture images currently defined (with previous calls to **glTexImage1D**, **glTexImage2D**, or **glTexImage3D**) do not follow the proper sequence for mipmaps (described above) 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 2D textures. In 1D textures, linear filtering accesses the two nearest texture elements.

GL_TEXTURE_3D is supported in OpenGL 1.2 and later.

GL_TEXTURE_3D_EXT requires the 3D texture extension.

GL_TEXTURE_MAX_ANISOTROPY_EXT requires the **EXT_texture_filter_anisotropic** extension.

Errors

GL_INVALID_ENUM is generated if *target* or *pname* is not one of the accepted defined values.

GL_INVALID_ENUM is generated if *params* should have a defined constant value (based on the value of *pname*) and does not.

GL_INVALID_OPERATION is generated if **glTexParameter** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glGetTexParameter

glGetTexLevelParameter

Related Information

The **glBindTexture** subroutine, **glPrioritizeTextures** subroutine, **glTexEnv** subroutine, **glTexGen** subroutine, **glTexImage1D** subroutine, **glTexImage2D** subroutine, **glTexImage3D** subroutine, **glTexImage3DEXT** subroutine.

glTexSubImage1D Subroutine

Purpose

Specifies a one-dimensional (1D) texture subimage.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glTexSubImage1D(GLenum  target,
                     GLint  level,
                     GLint  xoffset,
                     GLsizei  width,
                     GLenum  format,
                     GLenum  type,
                     const GLvoid * pixels)
```

Parameters

target Specifies the target texture. Must be **GL_TEXTURE_1D**.

level Specifies the level-of-detail number. Level 0 is the base image level. Level n is the nth mipmap reduction image.

xoffset Specifies a texel offset in the x direction within the texture array.

width Specifies the width of the texture subimage.

format Specifies the format of the pixel data. Symbolic constants **GL_COLOR_INDEX**, **GL_RED**, **GL_GREEN**, **GL_BLUE**, **GL_ALPHA**, **GL_RGB**, **GL_RGBA**, **GL_BGR**, **GL_BGRA**, **GL_ABGR_EXT**, **GL_LUMINANCE**, **GL_422_EXT**, **GL_422_REV_EXT**, **GL_422_AVERAGE_EXT**, **GL_422_REV_AVERAGE_EXT**, and **GL_LUMINANCE_ALPHA** are accepted.

type Specifies the data type for *Pixels*. Symbolic constants **GL_UNSIGNED_BYTE**, **GL_BYTE**, **GL_BITMAP**, **GL_UNSIGNED_SHORT**, **GL_SHORT**, **GL_UNSIGNED_INT**, **GL_INT**, **GL_FLOAT**, **GL_UNSIGNED_BYTE_3_3_2**, **GL_UNSIGNED_BYTE_2_3_3_REV**, **GL_UNSIGNED_SHORT_5_6_5**, **GL_UNSIGNED_SHORT_5_6_5_REV**, **GL_UNSIGNED_SHORT_4_4_4_4**, **GL_UNSIGNED_SHORT_4_4_4_4_REV**, **GL_UNSIGNED_SHORT_5_5_5_1**, **GL_UNSIGNED_SHORT_1_5_5_5_REV**, **GL_UNSIGNED_INT_8_8_8_8**, **GL_UNSIGNED_INT_8_8_8_8_REV**, **GL_UNSIGNED_INT_10_10_10_2**, and **GL_UNSIGNED_INT_2_10_10_10_REV** are accepted.

pixels Specifies a pointer to the image data in memory.

Description

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. To enable or disable one-dimensional texturing, call **glEnable** and **glDisable** with argument **GL_TEXTURE_1D**.

The **glTexSubImage1D** subroutine redefines a contiguous subregion of an existing one-dimensional texture image. The texels referenced by pixels replace the portion of the existing texture array with x indices xoffset and *xoffset* + *width* - 1, inclusive. This region may not include any texels outside the range of the texture array as it was originally specified. It is not an error to specify a subtexture with zero width, but such a specification has no effect.

GL_COLOR_INDEX

Each pixel is a single value, a color index. It is converted to fixed point, 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. Signed and unsigned integer data is converted with all fraction bits set to 0 (zero). Bitmap data converts to either 0.0 or 1.0.

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, 0 bits fill otherwise unspecified bit locations in the result.

If the GL is in red, green, blue, alpha (RGBA) mode, the resulting index is converted 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. If the GL is in color index mode and **GL_MAP_COLOR** is True, the index is replaced with the value that it references in the lookup table **GL_PIXEL_MAP_I_TO_I**. Whether the lookup replacement of the index is done or not, the integer part of the index is then ANDed with 2b -1, where *b* is the number of bits in a color index buffer.

The resulting indices or RGBA colors are then converted 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 $x_n = x_r + n \bmod Width$ and $y_n = y_r + [n/Width]$, where (x_r, y_r) is the current raster position. 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_RED

Each pixel is a single red component. This component is converted to the internal floating-point format in the same way as the red component of an RGBA pixel is, then it is converted 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. This component is converted to the internal floating-point format in the same way as the green component of an RGBA pixel is, then it is converted 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. This component is converted to the internal floating-point format in the same way as the blue component of an RGBA pixel is, then it is converted 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. This component is converted to the internal floating-point format in the same way as the alpha component of an RGBA pixel is, then it is converted 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 three-component group, red first, followed by green, followed by blue. Each component is converted to the internal floating-point format in the same way as 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_RGBA
: Each pixel is a four-component group, red first, followed by green, followed by blue, followed by alpha. Floating-point values are converted 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 are mapped similarly: the largest integer value maps to 1.0, and 0 maps to 0.0. 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. The results are clamped to the range [0,1].

If **GL_MAP_COLOR** is True, each color component is scaled by the size of the lookup table **GL_PIXEL_MAP_*c*_TO_*c***, then replaced by the value that it references in that table. *c* is **R**, **G**, **B**, or **A**, respectively.

The resulting RGBA colors are then converted 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 $x_n = x_r + n \bmod Width$ and $y_n = y_r + [n/Width]$, where (x_r, y_r) is the current raster position. 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_BGR
: Each pixel is a three-component group, blue first, followed by green, followed by red. Each component is converted to the internal floating-point format in the same way as the blue, green, and red components of an BGRA pixel are. The color triple is converted to an BGRA pixel with alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an BGRA pixel.

GL_BGRA
: Each pixel is a four-component group, blue first, followed by green, followed by red, followed by alpha. Floating-point values are converted 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 are mapped similarly: the largest integer value maps to 1.0, and 0 maps to 0.0. The resulting floating-point color values are then multiplied by **GL_*c*_SCALE** and added to **GL_*c*_BIAS**, where *c* is **BLUE**, **GREEN**, **RED**, and **ALPHA** for the respective color components. The results are clamped to the range [0,1].

If **GL_MAP_COLOR** is True, each color component is scaled by the size of the lookup table **GL_PIXEL_MAP_*c*_TO_*c***, then replaced by the value that it references in that table. *c* is **B**, **G**, **R**, or **A**, respectively.

The resulting BGRA colors are then converted 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 $x_n = x_r + n \bmod Width$ and $y_n = y_r + [n/Width]$, where (x_r, y_r) is the current raster position. 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_ABGR_EXT

Each pixel is a four-component group: for **GL_RGBA**, the red component is first, followed by green, followed by blue, followed by alpha; for **GL_BGRA**, the blue component is first, followed by green, followed by red, followed by alpha; for **GL_ABGR_EXT** the order is alpha, blue, green, and then red. Floating-point values are converted directly to an internal floatingpoint 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. 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. The results are clamped to the range [0,1].

If **GL_MAP_COLOR** is true, each color component is scaled by the size of lookup table **GL_PIXEL_MAP_c_TO _c**, then replaced by the value that it references in that table. *c* is **R**, **G**, **B**, or **A**, respectively.

The resulting RGBA colors are then converted 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

$$x_n = x_r + n \bmod width$$

$$y_n = y_r + \left\lfloor \frac{n}{width} \right\rfloor$$

where (x_r,y_r) is the current raster position. 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_LUMINANCE

Each pixel is a single luminance component. This component is converted to the internal floating-point format in the same way as the red component of an RGBA pixel is, then it is converted 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 two-component group, luminance first, followed by alpha. The two components are converted to the internal floating-point format in the same way as the red component of an RGBA pixel is, then they are converted 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_422_EXT

This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Y. The second component is Cb in the even pixels and Cr in the odd pixels. The Cb for each even pixel is used as the Cb value for that pixel and its neighbor to the right. The Cr in each odd pixel is used as the Cr value for that pixel and its neighbor to the left. (If the width of the image is odd, then the colors will be undefined in the rightmost column.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this conversion, the pixel is treated just as if it had been sent in as an RGB pixel.

GL_422_REV_EXT	This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Cb in the even pixels and Cr in the odd pixels. The second component is Y. The Cb for each even pixel is used as the Cb value for that pixel and its neighbor to the right. The Cr in each odd pixel is used as the Cr value for that pixel and its neighbor to the left. (If the width of the image is odd, then the colors will be undefined in the rightmost column.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this conversion, the pixel is treated just as if it had been sent in as an RGB pixel.
GL_422_AVERAGE_EXT	This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Y. The second component is Cb in the even pixels and Cr in the odd pixels. Each even pixel gets its Cb from itself, and its Cr from its neighbor to the right. Each odd pixel gets its Cb from the average of its left and right neighbor, and its Cr from the average of itself and its neighbor two to the right. (If the width of the image is odd, then the colors will be undefined in the rightmost column. If the neighbors to the right are not present for a given fragment, we use GL_422_EXT to compute that fragment.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this conversion, the pixel is treated just as if it had been sent in as an RGB pixel.
GL_422_REV_AVERAGE_EXT	This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Cb in the even pixels and Cr in the odd pixels. The second component is Y. Each even pixel gets its Cb from itself, and its Cr from its neighbor to the right. Each odd pixel gets its Cb from the average of its left and right neighbor, and its Cr from the average of itself and its neighbor two to the right. (If the width of the image is odd, then the colors will be undefined in the rightmost column. If the neighbors to the right are not present for a given fragment, we use GL_422_REV_EXT to compute that fragment.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this conversion, the pixel is treated just as if it had been sent in as an RGB pixel.

Notes

Texturing has no effect in color index mode.

The **glPixelStore** and **glPixelTransfer** modes affect texture images in exactly the way they affect **glDrawPixels**.

Format of **GL_ABGR_EXT** is part of the _extname (EXT_abgr) extension, not part of the core GL command set.

Errors

GL_INVALID_ENUM is generated if *target* is not one of the allowable values.

GL_INVALID_OPERATION is generated if the texture array has not been defined by a previous **glTexImage1D** operation.

GL_INVALID_VALUE is generated if *level* is less than zero.

GL_INVALID_VALUE may be generated if *level* is greater than log2(max), where max is the returned value of **GL_MAX_TEXTURE_SIZE**.

GL_INVALID_VALUE is generated if *width* < -b, where b is the border width of the texture array.

GL_INVALID_VALUE is generated if *xoffset* < -b, or if (*xoffset* + *width*) > (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_ENUM is generated if *format* is not an accepted format constant.

GL_INVALID_ENUM is generated if *type* is not a type constant.

GL_INVALID_ENUM is generated if *type* is **GL_BITMAP** and *format* is not **GL_COLOR_INDEX**.

GL_INVALID_OPERATION is generated if **glTexSubImage1D** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glGetTexImage

glIsEnabled with argument **GL_TEXTURE_1D**

Related Information

The **glDrawPixels** subroutine, **glFog** subroutine, **glPixelStore** subroutine, **glPixelTransfer** subroutine, **glTexEnv** subroutine, **glTexGen** subroutine, **glTexImage1D** subroutine, **glTexParameter** subroutine.

---

glTexSubImage1DEXT Subroutine

Purpose

Specifies a one-dimensional texture subimage.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glTexSubImage1DEXT(GLenum target,
                 GLint level,
                 GLint xoffset,
                 GLsizei width,
                 GLenum format,
                 GLenum type,
                 const GLvoid *pixels)
```

Parameters

target	Specifies the target texture. Must be **GL_TEXTURE_1D**

level	Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.
xoffset	Specifies a texel offset in the x direction within the texture array.
width	Specifies the width of the texture subimage.
format	Specifies the format of the pixel data. The following symbolic values are accepted: **GL_COLOR_INDEX**, **GL_RED**, **GL_GREEN**, **GL_BLUE**, **GL_ALPHA**, **GL_RGB**, **GL_RGBA**, **GL_ABGR_EXT**, **GL_LUMINANCE**, **GL_LUMINANCE_ALPHA**, **GL_422_EXT**, **GL_422_REV_EXT**, **GL_422_AVERAGE_EXT**, and **GL_422_REV_AVERAGE_EXT**.
type	Specifies 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	Specifies a pointer to the image data in memory.

Description

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**.

glTexSubImage1DEXT redefines a contiguous subregion of an existing one-dimensional texture image. The texels referenced by *pixels* replace the portion of the existing texture array with x indices *xoffset* and *xoffset*+*width*-1, inclusive. This region may not include any texels outside the range of the texture array as it was originally specified. It is not an error to specify a subtexture with zero width, but such a specification has no effect.

Notes

Texturing has no effect in color index mode.

glPixelStore and **glPixelTransfer** modes affect texture images in exactly the way they affect **glDrawPixels**.

Format of **GL_ABGR_EXT** is part of the _extname (EXT_abgr) extension, not part of the core GL command set.

Errors

GL_INVALID_ENUM is generated when *target* is not one of the allowable values.

GL_INVALID_OPERATION is generated when the texture array has not been defined by a previous **glTexImage1D** operation.

GL_INVALID_VALUE is generated if *level* is less than zero or greater than log2(*max*), where *max* is the returned value of **GL_MAX_TEXTURE_SIZE**.

GL_INVALID_VALUE is generated if *width* <-TEXTURE_BORDER, where TEXTURE_BORDER is the border width of the texture array.

GL_INVALID_VALUE is generated if *xoffset* <-TEXTURE_BORDER, (*xoffset*+*width*) > (TEXTURE_WIDTH- TEXTURE_BORDER). Where TEXTURE_WIDTH and TEXTURE_BORDER are the state values of the texture image being modified. Note that TEXTURE_WIDTH includes twice the border width.

GL_INVALID_ENUM is generated when *format* is not an accepted *format* constant.

GL_INVALID_ENUM is generated when *type* is not a *type* constant.

GL_INVALID_ENUM is generated if *type* is **GL_BITMAP** and *format* is not **GL_COLOR_INDEX**.

GL_INVALID_OPERATION is generated if **glTexSubImage1DEXT** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glGetTexImage

glIsEnabled with argument **GL_TEXTURE_1D**

File

/usr/include/GL/glext.h	Contains extensions to C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glDrawPixels** subroutine, **glFog** subroutine, **glPixelStore** subroutine, **glPixelTransfer** subroutine, **glTexEnv** subroutine, **glTexGen** subroutine, **glTexImage1D** subroutine, **glTexParameter** subroutine.

glTexSubImage2D Subroutine

Purpose

Specifies a two-dimensional (2D) texture subimage.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glTexSubImage2D(GLenum  target,
                     GLint  level,
                     GLint  xoffset,
                     GLint  yoffset,
                     GLsizei  width,
                     GLsizei  height,
                     GLenum  format,
                     GLenum  type,
                     const GLvoid * pixels)
```

Parameters

target Specifies the target texture. Must be **GL_TEXTURE_2D**.

level	Specifies the level-of-detail number. Level 0 is the base image level. Level n is the nth mipmap reduction image.
xoffset	Specifies a texel offset in the x direction within the texture array.
yoffset	Specifies a texel offset in the y direction within the texture array.
width	Specifies the width of the texture subimage.
height	Specifies the height of the texture subimage.
format	Specifies the format of the pixel data. Symbolic constants **GL_COLOR_INDEX**, **GL_RED**, **GL_GREEN**, **GL_BLUE**, **GL_ALPHA**, **GL_RGB**, **GL_RGBA**, **GL_BGR**, **GL_BGRA**, **GL_ABGR_EXT**, **GL_LUMINANCE**, **GL_422_EXT**, **GL_422_REV_EXT**, **GL_422_AVERAGE_EXT**, **GL_422_REV_AVERAGE_EXT**, and **GL_LUMINANCE_ALPHA** are accepted.
type	Specifies the data type for *Pixels*. Symbolic constants **GL_UNSIGNED_BYTE**, **GL_BYTE**, **GL_BITMAP**, **GL_UNSIGNED_SHORT**, **GL_SHORT**, **GL_UNSIGNED_INT**, **GL_INT**, **GL_FLOAT**, **GL_UNSIGNED_BYTE_3_3_2**, **GL_UNSIGNED_BYTE_2_3_3_REV**, **GL_UNSIGNED_SHORT_5_6_5**, **GL_UNSIGNED_SHORT_5_6_5_REV**, **GL_UNSIGNED_SHORT_4_4_4_4**, **GL_UNSIGNED_SHORT_4_4_4_4_REV**, **GL_UNSIGNED_SHORT_5_5_5_1**, **GL_UNSIGNED_SHORT_1_5_5_5_REV**, **GL_UNSIGNED_INT_8_8_8_8**, **GL_UNSIGNED_INT_8_8_8_8_REV**, **GL_UNSIGNED_INT_10_10_10_2**, and **GL_UNSIGNED_INT_2_10_10_10_REV** are accepted.
pixels	Specifies 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**.

Description

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. To enable and disable two-dimensional texturing, call **glEnable** and **glDisable** with argument **GL_TEXTURE_2D**.

The **glTexSubImage2D** subroutine redefines a contiguous subregion of an existing two-dimensional texture image. The texels referenced by pixels replace the portion of the existing texture array with x indices *xoffset* and *xoffset* + *width* - 1, inclusive, and y indices *yoffset* and *yoffset* + *height* - 1, inclusive. This region may not include any texels outside the range of the texture array as it was originally specified. It is not an error to specify a subtexture with zero width or height, but such a specification has no effect.

GL_COLOR_INDEX

Each pixel is a single value, a color index. It is converted to fixed point, 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. Signed and unsigned integer data is converted with all fraction bits set to 0 (zero). Bitmap data converts to either 0.0 or 1.0.

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, 0 bits fill otherwise unspecified bit locations in the result.

If the GL is in red, green, blue, alpha (RGBA) mode, the resulting index is converted 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. If the GL is in color index mode and **GL_MAP_COLOR** is True, the index is replaced with the value that it references in the lookup table **GL_PIXEL_MAP_I_TO_I**. Whether the lookup replacement of the index is done or not, the integer part of the index is then ANDed with 2b -1, where *b* is the number of bits in a color index buffer.

The resulting indices or RGBA colors are then converted 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* mod *Width* and *y*n = *y*r + [*n/Width*], where (*x*r, *y*r) is the current raster position. 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_RED

Each pixel is a single red component. This component is converted to the internal floating-point format in the same way as the red component of an RGBA pixel is, then it is converted 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. This component is converted to the internal floating-point format in the same way as the green component of an RGBA pixel is, then it is converted 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. This component is converted to the internal floating-point format in the same way as the blue component of an RGBA pixel is, then it is converted 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. This component is converted to the internal floating-point format in the same way as the alpha component of an RGBA pixel is, then it is converted 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 three-component group, red first, followed by green, followed by blue. Each component is converted to the internal floating-point format in the same way as 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_RGBA
Each pixel is a four-component group, red first, followed by green, followed by blue, followed by alpha. Floating-point values are converted 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 are mapped similarly: the largest integer value maps to 1.0, and 0 maps to 0.0. 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. The results are clamped to the range [0,1].

If **GL_MAP_COLOR** is True, each color component is scaled by the size of the lookup table **GL_PIXEL_MAP_*c*_TO_*c***, then replaced by the value that it references in that table. *c* is **R**, **G**, **B**, or **A**, respectively.

The resulting RGBA colors are then converted 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 $x_n = x_r + n \bmod Width$ and $y_n = y_r + [n/Width]$, where (x_r, y_r) is the current raster position. 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_BGR
Each pixel is a three-component group, blue first, followed by green, followed by red. Each component is converted to the internal floating-point format in the same way as the blue, green, and red components of an BGRA pixel are. The color triple is converted to an BGRA pixel with alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an BGRA pixel.

GL_BGRA
Each pixel is a four-component group, blue first, followed by green, followed by red, followed by alpha. Floating-point values are converted 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 are mapped similarly: the largest integer value maps to 1.0, and 0 maps to 0.0. The resulting floating-point color values are then multiplied by **GL_*c*_SCALE** and added to **GL_*c*_BIAS**, where *c* is **BLUE**, **GREEN**, **RED**, and **ALPHA** for the respective color components. The results are clamped to the range [0,1].

If **GL_MAP_COLOR** is True, each color component is scaled by the size of the lookup table **GL_PIXEL_MAP_*c*_TO_*c***, then replaced by the value that it references in that table. *c* is **B**, **G**, **R**, or **A**, respectively.

The resulting BGRA colors are then converted 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 $x_n = x_r + n \bmod Width$ and $y_n = y_r + [n/Width]$, where (x_r, y_r) is the current raster position. 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_ABGR_EXT

Each pixel is a four-component group: for **GL_RGBA**, the red component is first, followed by green, followed by blue, followed by alpha; for **GL_BGRA**, the blue component is first, followed by green, followed by red, followed by alpha; for **GL_ABGR_EXT** the order is alpha, blue, green, and then red. Floating-point values are converted directly to an internal floatingpoint 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. 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. The results are clamped to the range [0,1].

If **GL_MAP_COLOR**is true, each color component is scaled by the size of lookup table **GL_PIXEL_MAP_c_TO _c**, then replaced by the value that it references in that table. *c* is **R**, **G**, **B**, or **A**, respectively.

The resulting RGBA colors are then converted 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. 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_LUMINANCE

Each pixel is a single luminance component. This component is converted to the internal floating-point format in the same way as the red component of an RGBA pixel is, then it is converted 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 two-component group, luminance first, followed by alpha. The two components are converted to the internal floating-point format in the same way as the red component of an RGBA pixel is, then they are converted 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_422_EXT

This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Y. The second component is Cb in the even pixels and Cr in the odd pixels. The Cb for each even pixel is used as the Cb value for that pixel and its neighbor to the right. The Cr in each odd pixel is used as the Cr value for that pixel and its neighbor to the left. (If the width of the image is odd, then the colors will be undefined in the rightmost column.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this conversion, the pixel is treated just as if it hadbeen sent in as an RGB pixel.

GL_422_REV_EXT This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Cb in the even pixels and Cr in the odd pixels. The second component is Y. The Cb for each even pixel is used as the Cb value for that pixel and its neighbor to the right. The Cr in each odd pixel is used as the Cr value for that pixel and its neighbor to the left. (If the width of the image is odd, then the colors will be undefined in the rightmost column.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this conversion, the pixel is treated just as if it had been sent in as an RGB pixel.

GL_422_AVERAGE_EXT This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Y. The second component is Cb in the even pixels and Cr in the odd pixels. Each even pixel gets its Cb from itself, and its Cr from its neighbor to the right. Each odd pixel gets its Cb from the average of its left and right neighbor, and its Cr from the average of itself and its neighbor two to the right. (If the width of the image is odd, then the colors will be undefined in the rightmost column. If the neighbors to the right are not present for a given fragment, we use GL_422_EXT to compute that fragment.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this conversion, the pixel is treated just as if it had been sent in as an RGB pixel.

GL_422_REV_AVERAGE_EXT This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Cb in the even pixels and Cr in the odd pixels. The second component is Y. Each even pixel gets its Cb from itself, and its Cr from its neighbor to the right. Each odd pixel gets its Cb from the average of its left and right neighbor, and its Cr from the average of itself and its neighbor two to the right. (If the width of the image is odd, then the colors will be undefined in the rightmost column. If the neighbors to the right are not present for a given fragment, we use GL_422_REV_EXT to compute that fragment.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this convers ion, the pixel is treated just as if it had been sent in as an RGB pixel.

Notes

Texturing has no effect in color index mode.

The **glPixelStore** and **glPixelTransfer** modes affect texture images in exactly the way they affect **glDrawPixels**.

Format of **GL_ABGR_EXT** is part of the _extname (EXT_abgr) extension, not part of the core GL command set.

Errors

GL_INVALID_ENUM is generated if target is not **GL_TEXTURE_2D**.

GL_INVALID_OPERATION is generated if the texture array has not been defined by a previous **glTexImage2D** operation.

GL_INVALID_VALUE is generated if *level* is less than zero.

GL_INVALID_VALUE may be generated if *level* is greater than log2(max), where max is the returned value of **GL_MAX_TEXTURE_SIZE**.

GL_INVALID_VALUE is generated if *width* < -b or if *height* < -b, where b is the border width of the texture array.

GL_INVALID_VALUE is generated if *xoffset* < -b, (*xoffset* + *width*) > (w - b), *yoffset* < -b, or (*yoffset* + *height*) > (h - b). Where w is the **GL_TEXTURE_WIDTH**, h is the **GL_TEXTURE_HEIGHT**, and b is the border width of the texture image being modified. Note that w and h include twice the border width.

GL_INVALID_ENUM is generated if *format* is not an accepted format constant.

GL_INVALID_ENUM is generated if *type* is not a type constant.

GL_INVALID_ENUM is generated if *type* is **GL_BITMAP** and *format* is not **GL_COLOR_INDEX**.

GL_INVALID_OPERATION is generated if **glTexSubImage2D** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glGetTexImage

glIsEnabled with argument **GL_TEXTURE_2D**

Related Information

The **glDrawPixels** subroutine, **glFog** subroutine, **glPixelStore** subroutine, **glPixelTransfer** subroutine, **glTexEnv** subroutine, **glTexGen** subroutine, **glTexImage2D** subroutine, **glTexParameter** subroutine.

---

glTexSubImage2DEXT Subroutine

Purpose

Specifies a two-dimensional texture subimage.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glTexSubImage2DEXT( GLenum target,
                  GLint level,
                  GLint xoffset,
                  GLint yoffset,
                  GLsizei width,
                  GLsizei height,
                  GLenum format,
                  GLenum type,
                  const GLvoid *pixels)
```

Parameters

target	Specifies the target texture. Must be **GL_TEXTURE_2D**
level	Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.
xoffset	Specifies a texel offset in the x direction within the texture array.
yoffset	Specifies a texel offset in the y direction within the texture array.
width	Specifies the width of the texture subimage.
height	Specifies the height of the texture subimage.
format	Specifies the format of the pixel data. The following symbolic values are accepted: **GL_COLOR_INDEX**, **GL_RED**, **GL_GREEN**, **GL_BLUE**, **GL_ALPHA**, **GL_RGB**, **GL_RGBA**, **GL_ABGR_EXT**, **GL_LUMINANCE**, **GL_LUMINANCE_ALPHA**, **GL_422_EXT**, **GL_422_REV_EXT**, **GL_422_AVERAGE_EXT**, and **GL_422_REV_AVERAGE_EXT**.
type	Specifies 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	Specifies a pointer to the image data in memory.

Description

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**.

glTexSubImage2DEXT redefines a contiguous subregion of an existing two-dimensional texture image. The texels referenced by *pixels* replace the portion of the existing texture array with x indices *xoffset* and *xoffset*+*width*-1, inclusive, and y indices *yoffset* and *yoffset*+*height*-1, inclusive. This region may not include any texels outside the range of the texture array as it was originally specified. It is not an error to specify a subtexture with zero width or height, but such a specification has no effect.

Notes

Texturing has no effect in color index mode.

glPixelStore and **glPixelTransfer** modes affect texture images in exactly the way they affect **glDrawPixels**.

GL_ABGR_EXT is only valid if the **GL_EXT_abgr** extension is present.

Errors

GL_INVALID_ENUM is generated when *target* is not one of the allowable values.

GL_INVALID_OPERATION is generated when the texture array has not been defined by a previous **glTexImage2D** operation.

GL_INVALID_VALUE is generated if *level* is less than zero or greater than log2(*max*), where *max* is the returned value of **GL_MAX_TEXTURE_SIZE**.

GL_INVALID_VALUE is generated if *width* height

GL_INVALID_VALUE is generated if *xoffset* <-TEXTURE_BORDER, (*xoffset*+*width*) > (TEXTURE_WIDTH - TEXTURE_BORDER), *yoffset* <-TEXTURE_BORDER, or (*yoffset*+*height*)> (TEXTURE_HEIGHT - TEXTURE_BORDER), where TEXTURE_WIDTH, TEXTURE_HEIGHT, and TEXTURE_BORDER are the state values of the texture image being modified. Note that TEXTURE_WIDTH and TEXTURE_HEIGHT include twice the border width.

GL_INVALID_ENUM is generated when *format* is not an accepted *format* constant.

GL_INVALID_ENUM is generated when *type* is not a *type* constant.

GL_INVALID_ENUM is generated if *type* is **GL_BITMAP** and *format* is not **GL_COLOR_INDEX**.

GL_INVALID_OPERATION is generated if **glTexSubImage2DEXT** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glGetTexImage

glIsEnabled with argument **GL_TEXTURE_2D**

File

/usr/include/GL/glext.h	Contains extensions to C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glDrawPixels** subroutine, **glFog** subroutine, **glPixelStore** subroutine, **glPixelTransfer** subroutine, **glTexEnv** subroutine, **glTexGen** subroutine, **glTexImage2D** subroutine, **glTexParameter** subroutine.

---

glTexSubImage3D Subroutine

Purpose

Specifies a three-dimensional (3D) texture subimage.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glTexSubImage3D(GLenum  target,
                     GLint  level,
                     GLint  xoffset,
                     GLint  yoffset,
                     GLint  zoffset,
                     GLsizei  width,
                     GLsizei  height,
                     GLsizei  depth,
                     GLenum  format,
```

GLenum *type*,

const GLvoid * *pixels***)**

Parameters

target	Specifies the target texture. Must be **GL_TEXTURE_3D**.
level	Specifies the level-of-detail number. Level 0 is the base image level. Level n is the nth mipmap reduction image.
xoffset	Specifies a texel offset in the x direction within the texture array.
yoffset	Specifies a texel offset in the y direction within the texture array.
zoffset	Specifies a texel offset in the z direction within the texture array.
width	Specifies the width of the texture subimage.
height	Specifies the height of the texture subimage.
depth	Specifies the depth of the texture subimage.
format	Specifies the format of the pixel data. Symbolic constants **GL_COLOR_INDEX**, **GL_RED**, **GL_GREEN**, **GL_BLUE**, **GL_ALPHA**, **GL_RGB**, **GL_RGBA**, **GL_BGR**, **GL_BGRA**, **GL_ABGR_EXT**, **GL_LUMINANCE**, **GL_422_EXT**, **GL_422_REV_EXT**, **GL_422_AVERAGE_EXT**, **GL_422_REV_AVERAGE_EXT**, and **GL_LUMINANCE_ALPHA** are accepted.
type	Specifies the data type for *Pixels*. Symbolic constants **GL_UNSIGNED_BYTE**, **GL_BYTE**, **GL_BITMAP**, **GL_UNSIGNED_SHORT**, **GL_SHORT**, **GL_UNSIGNED_INT**, **GL_INT**, **GL_FLOAT**, **GL_UNSIGNED_BYTE_3_3_2**, **GL_UNSIGNED_BYTE_2_3_3_REV**, **GL_UNSIGNED_SHORT_5_6_5**, **GL_UNSIGNED_SHORT_5_6_5_REV**, **GL_UNSIGNED_SHORT_4_4_4_4**, **GL_UNSIGNED_SHORT_4_4_4_4_REV**, **GL_UNSIGNED_SHORT_5_5_5_1**, **GL_UNSIGNED_SHORT_1_5_5_5_REV**, **GL_UNSIGNED_INT_8_8_8_8**, **GL_UNSIGNED_INT_8_8_8_8_REV**, **GL_UNSIGNED_INT_10_10_10_2**, and **GL_UNSIGNED_INT_2_10_10_10_REV** are accepted.
pixels	Specifies a pointer to the image data in memory.

Description

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. To enable and disable three-dimensional texturing, call **glEnable** and **glDisable** with argument **GL_TEXTURE_3D**.

The **glTexSubImage3D** subroutine redefines a contiguous subregion of an existing three-dimensional texture image. The texels referenced by *pixels* replace the portion of the existing texture array with x indices *xoffset* and *xoffset* + *width* - 1, inclusive, y indices *yoffset* and *yoffset* + *height* - 1, inclusive, z indices *zoffset* and *zoffset* + *depth* - 1, inclusize. This region may not include any texels outside the range of the texture array as it was originally specified. It is not an error to specify a subtexture with zero *width*, *height* or *depth*, but such a specification has no effect.

GL_COLOR_INDEX

Each pixel is a single value, a color index. It is converted to fixed point, 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. Signed and unsigned integer data is converted with all fraction bits set to 0 (zero). Bitmap data converts to either 0.0 or 1.0.

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, 0 bits fill otherwise unspecified bit locations in the result.

If the GL is in red, green, blue, alpha (RGBA) mode, the resulting index is converted 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. If the GL is in color index mode and **GL_MAP_COLOR** is True, the index is replaced with the value that it references in the lookup table **GL_PIXEL_MAP_I_TO_I**. Whether the lookup replacement of the index is done or not, the integer part of the index is then ANDed with 2b -1, where *b* is the number of bits in a color index buffer.

The resulting indices or RGBA colors are then converted 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 $xn = xr + n$ mod *Width* and $yn = yr + [n/Width]$, where (xr, yr) is the current raster position. 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_RED

Each pixel is a single red component. This component is converted to the internal floating-point format in the same way as the red component of an RGBA pixel is, then it is converted 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. This component is converted to the internal floating-point format in the same way as the green component of an RGBA pixel is, then it is converted 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. This component is converted to the internal floating-point format in the same way as the blue component of an RGBA pixel is, then it is converted 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. This component is converted to the internal floating-point format in the same way as the alpha component of an RGBA pixel is, then it is converted 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 three-component group, red first, followed by green, followed by blue. Each component is converted to the internal floating-point format in the same way as 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_RGBA

Each pixel is a four-component group, red first, followed by green, followed by blue, followed by alpha. Floating-point values are converted 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 are mapped similarly: the largest integer value maps to 1.0, and 0 maps to 0.0. 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. The results are clamped to the range [0,1].

If **GL_MAP_COLOR** is True, each color component is scaled by the size of the lookup table **GL_PIXEL_MAP_*c*_TO_*c***, then replaced by the value that it references in that table. *c* is **R**, **G**, **B**, or **A**, respectively.

The resulting RGBA colors are then converted 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 $x_n = x_r + n \bmod Width$ and $y_n = y_r + [n/Width]$, where (x_r, y_r) is the current raster position. 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_BGR

Each pixel is a three-component group, blue first, followed by green, followed by red. Each component is converted to the internal floating-point format in the same way as the blue, green, and red components of an BGRA pixel are. The color triple is converted to an BGRA pixel with alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an BGRA pixel.

GL_BGRA

Each pixel is a four-component group, blue first, followed by green, followed by red, followed by alpha. Floating-point values are converted 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 are mapped similarly: the largest integer value maps to 1.0, and 0 maps to 0.0. The resulting floating-point color values are then multiplied by **GL_*c*_SCALE** and added to **GL_*c*_BIAS**, where *c* is **BLUE**, **GREEN**, **RED**, and **ALPHA** for the respective color components. The results are clamped to the range [0,1].

If **GL_MAP_COLOR** is True, each color component is scaled by the size of the lookup table **GL_PIXEL_MAP_*c*_TO_*c***, then replaced by the value that it references in that table. *c* is **B**, **G**, **R**, or **A**, respectively.

The resulting BGRA colors are then converted 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 $x_n = x_r + n \bmod Width$ and $y_n = y_r + [n/Width]$, where (x_r, y_r) is the current raster position. 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_ABGR_EXT

Each pixel is a four-component group: for **GL_RGBA**, the red component is first, followed by green, followed by blue, followed by alpha; for **GL_BGRA**, the blue component is first, followed by green, followed by red, followed by alpha; for **GL_ABGR_EXT** the order is alpha, blue, green, and then red. Floating-point values are converted directly to an internal floatingpoint 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. 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. The results are clamped to the range [0,1].

If **GL_MAP_COLOR**is true, each color component is scaled by the size of lookup table **GL_PIXEL_MAP_c_TO _c**, then replaced by the value that it references in that table. *c* is **R**, **G**, **B**, or **A**, respectively.

The resulting RGBA colors are then converted 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. 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_LUMINANCE

Each pixel is a single luminance component. This component is converted to the internal floating-point format in the same way as the red component of an RGBA pixel is, then it is converted 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 two-component group, luminance first, followed by alpha. The two components are converted to the internal floating-point format in the same way as the red component of an RGBA pixel is, then they are converted 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_422_EXT

This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Y. The second component is Cb in the even pixels and Cr in the odd pixels. The Cb for each even pixel is used as the Cb value for that pixel and its neighbor to the right. The Cr in each odd pixel is used as the Cr value for that pixel and its neighbor to the left. (If the width of the image is odd, then the colors will be undefined in the rightmost column.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this conversion, the pixel is treated just as if it had been sent in as an RGB pixel.

GL_422_REV_EXT	This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Cb in the even pixels and Cr in the odd pixels. The second component is Y. The Cb for each even pixel is used as the Cb value for that pixel and its neighbor to the right. The Cr in each odd pixel is used as the Cr value for that pixel and its neighbor to the left. (If the width of the image is odd, then the colors will be undefined in the rightmost column.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this conversion, the pixel is treated just as if it had been sent in as an RGB pixel.
GL_422_AVERAGE_EXT	This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Y. The second component is Cb in the even pixels and Cr in the odd pixels. Each even pixel gets its Cb from itself, and its Cr from its neighbor to the right. Each odd pixel gets its Cb from the average of its left and right neighbor, and its Cr from the average of itself and its neighbor two to the right. (If the width of the image is odd, then the colors will be undefined in the rightmost column. If the neighbors to the right are not present for a given fragment, we use GL_422_EXT to compute that fragment.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this conversion , the pixel is treated just as if it had been sent in as an RGB pixel.
GL_422_REV_AVERAGE_EXT	This extension is for use with the ″YCbCr″ color space, and should only be used in systems that have the **IBM_YCbCr** extension. The **GL_YCBCR_TO_RGB_MATRIX_IBM** matrix should be loaded using **glLoadNamedMatrixIBM** before glDrawPixels is called with this parameter. Each pixel is a two-component group. The first component is Cb in the even pixels and Cr in the odd pixels. The second component is Y. Each even pixel gets its Cb from itself, and its Cr from its neighbor to the right. Each odd pixel gets its Cb from the average of its left and right neighbor, and its Cr from the average of itself and its neighbor two to the right. (If the width of the image is odd, then the colors will be undefined in the rightmost column. If the neighbors to the right are not present for a given fragment, we use GL_422_REV_EXT to compute that fragment.) Through the use of the color matrix, Y then assumes the role of red, Cb becomes green and Cr becomes blue. After this convers ion, the pixel is treated just as if it had been sent in as an RGB pixel.

Notes

Texturing has no effect in color index mode.

The **glPixelStore** and **glPixelTransfer** modes affect texture images in exactly the way they affect **glDrawPixels**.

Errors

GL_INVALID_ENUM is generated if *target* is not **GL_TEXTURE_3D**.

GL_INVALID_OPERATION is generated if the texture array has not been defined by a previous **glTexImage3D** operation.

GL_INVALID_VALUE is generated if *level* is less than zero.

GL_INVALID_VALUE may be generated if *level* is greater than log2(max), where max is the returned value of **GL_MAX_3D_TEXTURE_SIZE**.

GL_INVALID_VALUE is generated if *width* < -b or if *height* < -b, or if *depth* < -b where b is the border width of the texture array.

GL_INVALID_VALUE is generated if *xoffset* < -b, (*xoffset* + *width*) > (w - b), *yoffset* < -b, (*yoffset* + *height*) > (h - b), *zoffset* < -b, (*zoffset* + *depth*) > (d -b). Where w is the **GL_TEXTURE_WIDTH**, h is the **GL_TEXTURE_HEIGHT**, d is the **GL_TEXTURE_DEPTH**, and b is the border width of the texture image being modified. Note that w, h, and d include twice the border width.

GL_INVALID_ENUM is generated if *format* is not an accepted format constant.

GL_INVALID_ENUM is generated if *type* is not a type constant.

GL_INVALID_ENUM is generated if *type* is **GL_BITMAP** and *format* is not **GL_COLOR_INDEX**.

GL_INVALID_OPERATION is generated if **glTexSubImage3D** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glGetTexImage

glIsEnabled with argument GL_TEXTURE_3D

Related Information

The **glDrawPixels** subroutine, **glFog** subroutine, **glPixelStore** subroutine, **glPixelTransfer** subroutine, **glTexEnv** subroutine, **glTexGen** subroutine, **glTexImage3D** subroutine, **glTexParameter** subroutine.

glTexSubImage3DEXT Subroutine

Purpose

Specifies a three-dimensional (3D) texture subimage.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glTexSubImage3DEXT(GLenum target,
                        GLint level,
                        GLint xoffset,
                        GLint yoffset,
                        GLint zoffset,
                        GLsizei width,
                        GLsizei height,
                        GLsizei depth,
                        GLenum format,
                        GLenum type,
                        const GLvoid *pixels)
```

Parameters

target	Specifies the target texture. Must be **GL_TEXTURE_3D_EXT**.

level	Specifies the level-of-detail number. Level 0 is the base image level. Level n is the nth mipmap reduction image.
xoffset	Specifies a texel offset in the x direction within the texture array.
yoffset	Specifies a texel offset in the y direction within the texture array.
zoffset	Specifies a texel offset in the z direction within the texture array.
width	Specifies the width of the texture subimage.
height	Specifies the height of the texture subimage.
depth	Specifies the depth of the texture subimage.
format	Specifies the format of the pixel data. The following symbolic values are accepted: **GL_COLOR_INDEX**, **GL_RED**, **GL_GREEN**, **GL_BLUE**, **GL_ALPHA**, **GL_RGB**, **GL_RGBA**, **GL_ABGR_EXT**, **GL_LUMINANCE**, **GL_LUMINANCE_ALPHA**, **GL_422_EXT**, **GL_422_REV_EXT**, **GL_422_AVERAGE_EXT**, and **GL_422_REV_AVERAGE_EXT**.
type	Specifies 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	Specifies a pointer to the image data in memory.

Description

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. To enable and disable three-dimensional texturing, call **glEnable** and **glDisable** with argument **GL_TEXTURE_3D_EXT**.

The **glTexSubImage3DEXT** subroutine redefines a contiguous subregion of an existing three-dimensional texture image. The texels referenced by *pixels* replace the portion of the existing texture array with x indices *xoffset* and *xoffset* + *width* - 1, inclusive, y indices *yoffset* and *yoffset* + *height* - 1, inclusive, z indices *zoffset* and *zoffset* + *depth* - 1, inclusize. This region may not include any texels outside the range of the texture array as it was originally specified. It is not an error to specify a subtexture with zero *width*, *height* or *depth*, but such a specification has no effect.

Notes

Texturing has no effect in color index mode.

The **glPixelStore** and **glPixelTransfer** modes affect texture images in exactly the way they affect **glDrawPixels**.

Format of **GL_ABGR_EXT** is part of the _extstring(EXT_abgr) extension, not part of the core GL command set.

Errors

GL_INVALID_ENUM is generated if *target* is not **GL_TEXTURE_3D_EXT**.

GL_INVALID_OPERATION is generated if the texture array has not been defined by a previous **glTexImage3D** operation.

GL_INVALID_VALUE is generated if *level* is less than zero.

GL_INVALID_VALUE may be generated if *level* is greater than log2(max), where max is the returned value of **GL_MAX_3D_TEXTURE_SIZE_EXT**.

GL_INVALID_VALUE is generated if *width* < -b or if *height* < -b, or if *depth* < -b where b is the border width of the texture array.

GL_INVALID_VALUE is generated if *xoffset* < -b, (*xoffset* + *width*) > (w - b), *yoffset* < -b, (*yoffset* + *height*) > (h - b), *zoffset* < -b, (*zoffset* + *depth*) > (d -b). Where w is the **GL_TEXTURE_WIDTH**, h is the

GL_TEXTURE_HEIGHT, d is the **GL_TEXTURE_DEPTH_EXT**, and b is the border width of the texture image being modified. Note that w, h, and d include twice the border width.

GL_INVALID_ENUM is generated if *format* is not an accepted format constant.

GL_INVALID_ENUM is generated if *type* is not a type constant.

GL_INVALID_ENUM is generated if *type* is **GL_BITMAP** and *format* is not **GL_COLOR_INDEX**.

GL_INVALID_OPERATION is generated if **glTexSubImage3DEXT** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

Associated Gets

glGetTexImage

glIsEnabled with argument GL_TEXTURE_3D_EXT

Related Information

The **glDrawPixels** subroutine, **glFog** subroutine, **glPixelStore** subroutine, **glPixelTransfer** subroutine, **glTexEnv** subroutine, **glTexGen** subroutine, **glTexImage3DEXT** subroutine, **glTexParameter** subroutine.

glTranslate Subroutine

Purpose

Multiplies the current matrix by a translation matrix.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glTranslated(GLdouble X,
   GLdouble Y,
   GLdouble Z)
void glTranslatef(GLfloat X,
   GLfloat Y,
   GLfloat Z)
```

Parameters

X, *Y*, *Z*	Specify the *X*, *Y*, and *Z* coordinates of a translation vector.

Description

The **glTranslate** subroutine moves the coordinate system origin to the point specified by (*X*,*Y*,*Z*). The translation vector is used to compute a 4 x 4 translation matrix as follows:

$$\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 27. Translation Matrix. This diagram shows a matrix in brackets. The matrix consists of four lines containing four characters each. The first line contains the following (from left to right): one, zero, zero, x. The second line contains the following (from left to right): zero, one, zero, y. The third line contains the following (from left to right): zero, zero, one, z. The fourth line contains the following (from left to right): zero, zero, zero, one.

The current matrix (see the **glMatrixMode** subroutine for information on specifying the current matrix) 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, M is replaced with MT.

If the matrix mode is either **GL_MODELVIEW** or **GL_PROJECTION**, all objects drawn after **glTranslate** is called are translated. Use the **glPushMatrix** and **glPopMatrix** subroutines to save and restore the untranslated coordinate system.

Errors

GL_INVALID_OPERATION	The **glTranslate** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glTranslate** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_MATRIX_MODE**

glGet with argument **GL_MODELVIEW_MATRIX**

glGet with argument **GL_PROJECTION_MATRIX**

glGet with argument **GL_TEXTURE_MATRIX**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glMatrixMode** subroutine, **glMultMatrix** subroutine, **glPushMatrix** subroutine, **glRotate** subroutine, **glScale** subroutine.

glUnLockArraysEXT Subroutine

Purpose

Unlocks the currently enabled vertex arrays.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glUnlockArraysEXT (void)
```

Description

The **glUnlockArraysEXT** subroutine unlocks vertex arrays locked by the **glLockArraysEXT** subroutine.

Errors

INVALID_OPERATION	The **glUnlockArraysEXT** subroutine is called without a corresponding previous execution of **glLockArraysEXT**.
INVALID_OPERATION	The **glUnlockArraysEXT** subroutine is called between execution of **Begin** and the corresponding execution of **End**.

Related Information

The **glLockArraysEXT** subroutine.

glVertex Subroutine

Purpose

Specifies a vertex.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
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)

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

X, *Y*, *Z*, *W*	Specify *X*, *Y*, *Z*, and *W* coordinates of a vertex. Not all parameters are present in all forms of the command.
V	Specifies a pointer to an array of two, three, or four elements. The elements of a two-element array are *X* and *Y*. The elements of a three-element array are X, *Y*, and *Z*. The elements of a four-element array are *X*, *Y*, *Z*, and *W*.

Description

The **glVertex** subroutines are used within the **glBegin** and **glEnd** subroutine pairs to specify point, line, and polygon vertices. The current color, normal, texture coordinate, edge flag, secondary color, fog coordinate and color index 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.

Notes

Calling **glVertex** outside of a **glBegin**/**glEnd** subroutine pair results in undefined behavior.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glCallList** subroutine, **glColor** subroutine, **glEdgeFlag** subroutine, **glEvalCoord** subroutine, **glIndex** subroutine, **glMaterial** subroutine, **glNormal** subroutine, **glRect** subroutine, **glTexCoord** subroutine.

glVertexPointer Subroutine

Purpose

Defines an array of vertex data.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glVertexPointer(GLint  size,
   GLenum  type,
   GLsizei  stride,
   const GLvoid * pointer)
```

Description

The **glVertexPointer** subroutine specifies the location and data format of an array of vertex coordinates to use when rendering. The *size* parameter specifies the number of coordinates per vertex and *type* the data type of the coordinates. The *stride* parameter specifies the byte stride from one vertex to the next allowing vertices and attributes to be packed into a single array or stored in separate arrays. (Single array storage may be more efficient on some implementations; see **glInterleavedArrays**). When a vertex array is specified, *size*, *type*, *stride*, and *pointer* are saved as client side state.

To enable and disable the vertex array, call **glEnableClientState** and **glDisableClientState** with the argument **GL_VERTEX_ARRAY**. If enabled, the vertex array is used when **glDrawArrays**, **glDrawElements**, or **glArrayElement** is called.

Use **glDrawArrays** to construct a sequence of primitives (all of the same type) from prespecified vertex and vertex attribute arrays. Use **glArrayElement** to specify primitives by indexing vertices and vertex attributes and **glDrawElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

Use **glDrawArrays**, **glMultiDrawArraysEXT**, or **glMultiModeDrawArraysIBM** to construct a sequence of primitives from prespecified vertex and vertex attribute arrays. Use **glArrayElement** to specify primitives by indexing vertices and vertex attributes. Use **glDrawElements**, **glMultiDrawElementsEXT**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

If enabled, the Vertex array is used when **glDrawArrays**, **glDrawElements**, **glArrayElements**, **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** is called.

Parameters

size	Specifies the number of coordinates per vertex; must be 2, 3, or 4. The initial value is 4.
type	Specifies the data type of each coordinate in the array. Symbolic constants **GL_SHORT**, **GL_INT**, **GL_FLOAT**, or **GL_DOUBLE** are accepted. The initial value is **GL_FLOAT**.
stride	Specifies the byte offset between consecutive vertices. If stride is 0, the vertices are understood to be tightly packed in the array. The initial value is 0.
pointer	Specifies a pointer to the first coordinate of the first vertex in the array. The initial value is 0 (NULL pointer).

Notes

The **glVertexPointer** subroutine is available only if the GL version is 1.1 or greater.

The vertex array is initially disabled and it won't be accessed when **glArrayElement**, **glDrawElements** or **glDrawArrays** is called.

Execution of **glVertexPointer** is not allowed between **glBegin** and the corresponding **glEnd**, but an error may or may not be generated. If an error is not generated, the operation is undefined.

The **glVertexPointer** subroutine is typically implemented on the client side with no protocol.

Since the vertex array parameters are client side state, they are not saved or restored by **glPushAttrib** and **glPopAttrib**. Use **glPushClientAttrib** and **glPopClientAttrib** instead.

The **glVertexPointer** subroutine is not included in display lists.

Errors

- **GL_INVALID_VALUE** is generated if size is not 2, 3, or 4.
- **GL_INVALID_ENUM** is generated if type is is not an accepted value.
- **GL_INVALID_VALUE** is generated if stride is negative.

Associated Gets

- **glIsEnabled** with argument **GL_VERTEX_ARRAY**
- **glGet** with argument **GL_VERTEX_ARRAY_SIZE**
- **glGet** with argument **GL_VERTEX_ARRAY_TYPE**
- **glGet** with argument **GL_VERTEX_ARRAY_STRIDE**
- **glGetPointerv** with argument **GL_VERTEX_ARRAY_POINTER**

Related Information

The **glArrayElement** subroutine, **glColorPointer** subroutine, **glDrawArrays** subroutine, **glDrawElements** subroutine, **glEdgeFlagPointer** subroutine, **glEnable** subroutine, **glGetPointerv** subroutine,

glIndexPointer subroutine, **glNormalPointer** subroutine, **glPopClientAttrib** subroutine, **glPushClientAttrib** subroutine, **glTexCoordPointer** subroutine. **glVertexPointerListIBM** subroutine.

glVertexPointerEXT Subroutine

Purpose

Defines an array of vertex data.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glVertexPointerEXT(GLint size,
            GLenum type,
            GLsizei stride,
            GLsizei count,
            const GLvoid *pointer)
```

Parameters

size	Specifies the number of coordinates per vertex, must be 2,3, or 4.
type	Specifies the data type of each coordinate in the array. Symbolic constants **GL_SHORT**, **GL_INT**, **GL_FLOAT**, or **GL_DOUBLE_EXT** are accepted.
stride	Specifies the byte offset between consecutive vertexes. If *stride* is 0 the vertexes are understood to be tightly packed in the array.
count	Specifies the number of vertexes, counting from the first, that are static.
pointer	Specifies a pointer to the first coordinate of the first vertex in the array.

Description

The **glVertexPointerEXT** subroutine specifies the location and data format of an array of vertex coordinates to use when rendering. *size* specifies the number of coordinates per vertex and *type* the data type of the coordinates. *stride* gives the byte stride from one vertex to the next allowing vertexes and attributes to be packed into a single array or stored in separate arrays. (Single-array storage may be more efficient on some implementations.) *count* indicates the number of array elements (counting from the first) that are static. Static elements may be modified by the application, but once they are modified, the application must explicitly respecify the array before using it for any rendering. When a vertex array is specified, *size*, *type*, *stride*, *count*, and *pointer* are saved as client-side state, and static array elements may be cached by the implementation.

The vertex array is enabled and disabled using **glEnable** and **glDisable** with the argument **GL_VERTEX_ARRAY_EXT**. If enabled, the vertex array is used when **glDrawArraysEXT** or **glArrayElementEXT** is called.

Notes

Non-static array elements are not accessed until **glArrayElementEXT** or **glDrawArraysEXT** is executed.

By default the vertex array is disabled and it won't be accessed when **glArrayElementEXT** or **glDrawArraysEXT** is called.

Although, it is not an error to call **glVertexPointerEXT** between the execution of **glBegin** and the corresponding execution of **glEnd**, the results are undefined.

The **glVertexPointerEXT** subroutine will typically be implemented on the client side with no protocol.

Since the vertex array parameters are client side state, they are not saved or restored by **glPushAttrib** and **glPopAttrib**.

The **glVertexPointerEXT** commands are not entered into display lists.

The **glVertexPointerEXT** subroutine is part of the _extname(EXT_vertex_array) extension, not part of the core GL command set. If _extstring(EXT_vertex_array) is included in the string returned by **glGetString**, when called with argument **GL_EXTENSIONS**, extension _extname(EXT_vertex_array) is supported.

Use **glDrawArrays**, **glMultiDrawArraysEXT**, or **glMultiModeDrawArraysIBM** to construct a sequence of primitives from prespecified vertex and vertex attribute arrays. Use **glArrayElement** to specify primitives by indexing vertices and vertex attributes. Use **glDrawElements**, **glMultiDrawElementsEXT**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

If enabled, the Vertex array is used when **glDrawArrays**, **glDrawElements**, **glArrayElements**, **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** is called.

Errors

GL_INVALID_VALUE is generated if *size* is not 2, 3, or 4.

GL_INVALID_ENUM is generated if *type* is is not an accepted value.

GL_INVALID_VALUE is generated if *stride* or *count* is negative.

Associated Gets

glIsEnabled with argument **GL_VERTEX_ARRAY_EXT**

glGet with argument **GL_VERTEX_ARRAY_SIZE_EXT**

glGet with argument **GL_VERTEX_ARRAY_TYPE_EXT**

glGet with argument **GL_VERTEX_ARRAY_STRIDE_EXT**

glGet with argument **GL_VERTEX_ARRAY_COUNT_EXT**

glGetPointervEXT with argument **GL_VERTEX_ARRAY_POINTER_EXT**

File

/usr/include/GL/glext.h	Contains extensions to C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glArrayElementEXT** subroutine, **glColorPointerEXT** subroutine, **glDrawArraysEXT** subroutine, **glEdgeFlagPointerEXT** subroutine, **glGetPointervEXT** subroutine, **glIndexPointerEXT**subroutine, **glNormalPointerEXT** subroutine, **glTexCoordPointerEXT** subroutine.

glVertexPointerListIBM Subroutine

Purpose

Defines a list of vertex arrays.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glVertexPointerListIBM( GLint  size,
   GLenum  type,
   GLint  stride,
   const GLvoid ** pointer,
   GLint  ptrstride)
```

Description

The **glVertexPointerListIBM** subroutine specifies the location and data format of a list of arrays of vertex components to use when rendering. The *size* parameter specifies the number of components per vertex, and must be 2, 3 or 4. The *type* parameter specifies the data type of each vertex component. The *stride* parameter gives the byte stride from one vertex to the next allowing vertices and attributes to be packed into a single array or stored in separate arrays. (Single-array storage may be more efficient on some implementations; see **glInterleavedArrays**). The *ptrstride* parameter specifies the byte stride from one pointer to the next in the *pointer* array.

When a vertex array is specified, *size*, *type*, *stride*, *pointer* and *ptrstride* are saved as client side state.

A *stride* value of 0 does not specify a “tightly packed” array as it does in **glVertexPointer**. Instead, it causes the first array element of each array to be used for each vertex. Also, a negative value can be used for stride, which allows the user to move through each array in reverse order.

To enable and disable the vertex arrays, call **glEnableClientState** and **glDisableClientState** with the argument **GL_VERTEX_ARRAY**. The vertex array is initially disabled. When enabled, the vertex arrays are used when **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, **glDrawArrays**, **glDrawElements** or **glArrayElement** is called. The last three calls in this list will only use the first array (the one pointed at by *pointer*[0]). See the descriptions of these routines for more information on their use.

Use **glDrawArrays**, **glMultiDrawArraysEXT**, or **glMultiModeDrawArraysIBM** to construct a sequence of primitives from prespecified vertex and vertex attribute arrays. Use **glArrayElement** to specify primitives by indexing vertices and vertex attributes. Use **glDrawElements**, **glMultiDrawElementsEXT**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

If enabled, the Vertex array is used when **glDrawArrays**, **glDrawElements**, **glArrayElements**, **glMultiDrawArraysEXT**, **glMultiDrawElementsEXT**, **glMultiModeDrawArraysIBM**, **glMultiModeDrawElementsIBM**, or **glDrawRangeElements** is called.

Parameters

size	Specifies the number of components per vertex. It must be 2, 3 or 4. The initial value is 4.
type	Specifies the data type of each vertex component in the array. Symbolic constants **GL_BYTE**, **GL_UNSIGNED_BYTE**, **GL_SHORT**, **GL_UNSIGNED_SHORT**, **GL_INT**, **GL_UNSIGNED_INT**, **GL_FLOAT**, or **GL_DOUBLE** are accepted. The initial value is GL_FLOAT.

stride	Specifies the byte offset between consecutive vertices. The initial value is 0.
pointer	Specifies a list of vertex arrays. The initial value is 0 (NULL pointer).
ptrstride	Specifies the byte stride between successive pointers in the *pointer* array. The initial value is 0.

Notes

The **glVertexPointerListIBM** subroutine is available only if the GL_IBM_vertex_array_lists extension is supported.

Execution of **glVertexPointerListIBM** is not allowed between **glBegin** and the corresponding **glEnd**, but an error may or may not be generated. If an error is not generated, the operation is undefined.

The **glVertexPointerListIBM** subroutine is typically implemented on the client side.

Since the vertex array parameters are client side state, they are not saved or restored by **glPushAttrib** and **glPopAttrib**. Use **glPushClientAttrib** and **glPopClientAttrib** instead.

When a **glVertexPointerListIBM** call is encountered while compiling a display list, the information it contains does NOT contribute to the display list, but is used to update the immediate context instead.

The **glVertexPointer** call and the **glVertexPointerListIBM** call share the same state variables. A **glVertexPointer** call will reset the vertex list state to indicate that there is only one vertex list, so that any and all lists specified by a previous **glVertexPointerListIBM** call will be lost, not just the first list that it specified.

Error Codes

- **GL_INVALID_VALUE** is generated if size is not 2, 3 or 4.
- **GL_INVALID_ENUM** is generated if type is not an accepted value.

Associated Gets

- **glIsEnabled** with argument **GL_VERTEX_ARRAY**
- **glGetPointerv** with argument **GL_VERTEX_ARRAY_LIST_IBM**
- **glGet** with argument **GL_VERTEX_ARRAY_LIST_STRIDE_IBM**
- **glGet** with argument **GL_VERTEX_ARRAY_SIZE**
- **glGet** with argument **GL_VERTEX_ARRAY_STRIDE**
- **glGet** with argument **GL_VERTEX_ARRAY_TYPE**

Related Information

The **glArrayElement** subroutine, **glVertexPointer** subroutine, **glDrawArrays** subroutine, **glDrawElements** subroutine, **glEdgeFlagPointer** subroutine, **glEnable** subroutine, **glGetPointerv** subroutine, **glIndexPointer** subroutine, **glInterleavedArrays** subroutine, **glMultiDrawArraysEXT** subroutine, **glMultiDrawElementsEXT** subroutine, **glMultiModeDrawArraysIBM** subroutine, **glMultiModeDrawElementsIBM** subroutine, **glNormalPointer** subroutine, **glPopClientAttrib** subroutine, **glPushClientAttrib** subroutine, **glTexCoordPointer** subroutine, **glVertexPointer** subroutine.

glViewport Subroutine

Purpose

Sets the viewport.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glViewport(GLint X,
   GLint Y,
   GLsizei Width,
   GLsizei Height)
```

Parameters

X, *Y*	Specify the lower left corner of the viewport rectangle in pixels. The default is (0,0).
Width, Height	Specify the width and height, respectively, of the viewport. When a GL context is *first* attached to a window, *Width* and *Height* are set to the dimensions of that window.

Description

The **glViewport** subroutine specifies the affine transformation of *X* and *Y* from normalized device coordinates to window coordinates. Let (*X*nd, *Y*nd) be normalized device coordinates. Then the window coordinates (*X*w, *Y*w) are computed as follows:

Viewport width and height are silently clamped to a range that depends on the implementation. This range is queried by calling the **glGet** subroutine with the **GL_MAX_VIEWPORT_DIMS** argument.

Errors

GL_INVALID_VALUE	*Width* or *Height* is negative.
GL_INVALID_OPERATION	The **glViewport** subroutine is called between a call to **glBegin** and the corresponding call to **glEnd**.

Associated Gets

Associated gets for the **glViewport** subroutine are as follows. (See the **glGet** subroutine for more information.)

glGet with argument **GL_VIEWPORT**

glGet with argument **GL_MAX_VIEWPORT_DIMS**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** or **glEnd** subroutine, **glDepthRange** subroutine.

---

glVisibilityBufferIBM Subroutine

Purpose

Specifies the array in which visibility calculation results are stored.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
void glVisibilityBufferIBM(GLsizei  size,
                           GLuint  *buffer)
```

Description

This call helps implement an extension providing a mechanism similar to selection and feedback that can be used to perform occlusion culling. The basic algorithm is as follows:

1. Render the occluders (objects most likely to occlude other objects) into the frame buffer.
2. Specify the visibility buffer (the buffer in which non-occluded names are returned, **glVisibiltyBufferIBM(len, ptr)**).
3. Disable z-buffer, stencil-buffer, and color-buffer updates.
4. Select **GL_VISIBILITY_IBM** rendering mode (**glRenderMode(GL_VISIBILITY_IBM)**).
5. For each possible occludee: a) identify its name using the glLoadName command. b) render a simplified representation of the occludee
6. Restore the render mode to **GL_RENDER** (**glRenderMode(GL_RENDER)**). The return value from glRenderMode in this case is the number of visible (picked) objects.
7. Restore z-buffer, stencil-buffer, and color-buffer updates.
8. Render all objects that are found to be non-occluded (those appearing in the visibility buffer).

GL_VISIBILITY render mode is identical to **GL_RENDER** render mode except whenever a fragment passes all tests (ie, depth, stencil, alpha, scissor and window-ownership) then a visibility hit results. Whenever a name stack manipulation command is executed or **glRenderMode** is called and there is a hit since the last time the stack was manipulated or **glRenderMode** was called, then a hit record is written into the visibility array. The hit record consists of the number of names in the name stack at the time of the event followed by the name stack contents (bottom name first).

Besides occlusion culling, this extension can also be used to refine selection (picking) to include visiblity. The basic algorithm is a follows:

1. Application renders a scene in which the user wishes to pick a object in the scene.
2. Application uses the base OpenGL select feature to obtain a list of pick candidates.
3. Disable z-buffer, stencil-buffer, and color-buffer updates.
4. Change depth test to GL_EQUAL
5. Set the Scissor region to match the Pick aperture.
6. Select **GL_VISIBILITY_IBM** rendering mode (**glRenderMode(GL_VISIBILITY_IBM)**).
7. Render each pick candidate with name identifiers.
8. Restore the render mode to RENDER (**glRenderMode(GL_RENDER)**). The return value from **glRenderMode** in this case is the number of visible (picked) objects.
9. Restore the depth test.
10. Restore z-buffer, stencil-buffer, and color-buffer updates.

Parameters

size is an integer indicating the maximum number of values that can be stored in the visibility array.

buffer is a pointer to an array of unsigned integers (called the visibility array) to be filled with names.

Notes

This subroutine is only valid if the **GL_IBM_occlusion_cull** extension is defined.

Error Codes

GL_INVALID_VALUE	is generated if *size* is negative.
GL_INVALID_OPERATION	is generated if **glVisibilityBufferIBM** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.
GL_INVALID_OPERATION	is generated if **glVisibilityBufferIBM** is executed while the **glRenderMode** is **GL_VISIBILITY_IBM**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glRenderMode** subroutine, the **glVisibilityThresholdIBM** subroutine.

glVisibilityThresholdIBM Subroutine

Purpose

Specifies the number of visible fragments rendered before a visibility hit is registered.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
void glVisibilityThresholdIBM(GLsizei  threshold)
```

Description

glVisibilityThresholdIBM specifies the number of visible fragments rendered before a visibility hit is registered. A value of 0 results in a visibility hit on the first visible fragment; a value of 1 results in a visilibility hit on the second visible fragment. The *threshold* parameter is silently clamped to an implementation dependent range **0 - GL_MAX_VISIBILITY_THRESHOLD_IBM**.

Parameters

threshold	is an integer indicating the number visible fragments prior to registering a visibility hit.

Error Codes

GL_INVALID_OPERATION is generated if one of the following conditions exists:

- **glVisibilityThresholdIBM** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.
- **glVisibilityThresholdIBM** is executed while **RenderMode** is **GL_VISIBILITY_IBM**.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glRenderMode** subroutine, the **glVisibilityBufferIBM** subroutine.

Chapter 2. OpenGL Utility (GLU) Library

Following is a list of the subroutines available in the OpenGL utility library and the purpose of each subroutine.

Select the subroutine about which you want to read.

B	
gluBeginCurve	Delimits the beginning or end of a non-uniform rational B-spline (NURBS) curve definition.
gluBeginPolygon	Delimits the beginning or end of a polygon description.
gluBeginSurface	Delimits the beginning or end of a non-uniform rational B-spline (NURBS) surface definition.
gluBeginTrim	Delimits the beginning or end of a non-uniform rational B-spline (NURBS) trimming loop definition.
gluBuild1DMipmapLevels	Builds a subset of 1D mipmap levels.
gluBuild1DMipmaps	Creates 1-dimensional (1D) mipmaps.
gluBuild2DMipmapLevels	Builds a subset of 2D mipmap levels.
gluBuild2DMipmaps	Creates 2-dimensional (2D) mipmaps.
gluBuild3DMipmapLevels	Builds a subset of 3D mipmap levels.
gluBuild3DMipmaps	Builds a 3-dimensional (3D) mipmap.
C	
gluCheckExtension	Determines if an extension name is supported.
gluCylinder	Draws a cylinder.
D	
gluDeleteNurbsRenderer	Destroys a non-uniform rational B-spline (NURBS) object.
gluDeleteQuadric	Destroys a quadrics object.
gluDeleteTess	Destroys a tessellation object.
gluDisk	Draws a disk.
E	
gluErrorString	Produces an error string from an OpenGL or GLU error code.
G	
gluGetNurbsProperty	Gets a non-uniform rational B-spline (NURBS) property.
gluGetString	Returns a pointer to a static string describing the GLU version or the GLU extensions that are supported.
gluGetTessProperty	Gets a tessellation object property.
L	
gluLoadSamplingMatrices	Loads non-uniform rational B-spline (NURBS) sampling and culling matrices.
gluLookAt	Defines a viewing transformation.
N	
gluNewNurbsRenderer	Creates a non-uniform rational B-spline (NURBS) object.
gluNewQuadric	Creates a quadrics object.
gluNewTess	Creates a tessellation object.
gluNextContour	Marks the beginning of another contour.
gluNurbsCallback	Defines a callback for a non-uniform rational B-spline (NURBS) object.
gluNurbsCallbackData	Sets a user data pointer.
gluNurbsCallbackDataEXT	Sets a user data pointer.
gluNurbsCurve	Defines the shape of a non-uniform rational B-spline (NURBS) curve.

© Copyright IBM Corp. 1994, 2002

gluNurbsProperty	Sets a non-uniform rational B-spline (NURBS) property.
gluNurbsSurface	Defines the shape of a non-uniform rational B-spline (NURBS) surface.
O	
gluOrtho2D	Defines a 2-dimensional (2D) orthographic projection matrix.
P	
gluPartialDisk	Draws an arc of a disk.
gluPerspective	Sets up a perspective projection matrix.
gluPickMatrix	Defines a picking region.
gluProject	Maps object coordinates to window coordinates.
gluPwlCurve	Defines a piecewise linear non-uniform rational B-spline (NURBS) trimming curve.
Q	
gluQuadricCallback	Defines a callback for a quadrics object.
gluQuadricDrawStyle	Specifies the desired quadric drawing style.
gluQuadricNormals	Specifies the desired normals for quadrics.
gluQuadricOrientation	Specifies the desired inside/outside orientation for quadrics.
gluQuadricTexture	Specifies if texturing is desired for quadrics.
S	
gluScaleImage	Scales an image to an arbitrary size.
gluSphere	Draws a sphere.
T	
gluTessBeginContour	Delimits a contour description.
gluTessBeginPolygon	Delimits a polygon description.
gluTessCallback	Defines a callback for a tessellation object.
gluTessEndPolygon	Delimits a polygon description.
gluTessNormal	Specifies a normal for a polygon.
gluTessProperty	Sets a tessellation object property.
gluTessVertex	Specifies a vertex on a polygon.
U	
gluUnProject	Projects world space coordinates to object space.
gluUnProject4	Maps window and clip coordinates to object coordinates.

gluBeginCurve or gluEndCurve Subroutine

Purpose

Delimits the beginning or end of a non-uniform rational B-spline (NURBS) curve definition.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

void gluBeginCurve(GLUnurbs* *nurb***)**

void gluEndCurve(GLUnurbs* *nurb***)**

Description

Use the **gluBeginCurve** subroutine to mark the beginning of a NURBS curve definition. After calling the **gluBeginCurve** subroutine, make one or more calls to the **gluNurbsCurve** subroutine to define the attributes of the curve. One (and only one) of these calls must have a curve type of **GL_MAP1_VERTEX_3** or **GL_MAP1_VERTEX_4**.

Use the **gluEndCurve** subroutine to mark the end of the NURBS curve definition.

OpenGL evaluators render the NURBS curve as a series of line segments. Evaluator state is preserved during rendering with the **glPushAttrib**(**GL_EVAL_BIT**) and **glPopAttrib** attributes. (See the **glPushAttrib** subroutine for details on what state these calls preserve.)

Parameters

nurb	Specifies the NURBS object created with the **gluNewNurbsRenderer** subroutine.

Examples

The following commands render a textured NURBS curve with normals. Texture coordinates and normals are also specified as NURBS curves.

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

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glPushAttrib** or **glPopAttrib** subroutine, **gluBeginSurface** subroutine, **gluBeginTrim** subroutine, **gluNewNurbsRenderer** subroutine, **gluNurbsCurve** subroutine.

gluBeginPolygon or gluEndPolygon Subroutine

Purpose

Delimits the beginning or end of a polygon description.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

void gluBeginPolygon(GLUtesselator* *tess***)**

void gluEndPolygon(GLUtesselator* *tess***)**

Description

The **gluBeginPolygon** and **gluEndPolygon** subroutines delimit the definition of a nonconvex polygon. To define a nonconvex polygon, first call the **gluBeginPolygon** subroutine. Then, call the **gluTessVertex** subroutine to define the contours of the polygon for each vertex and the **gluNextContour** subroutine to start each new contour. (See the **gluTessVertex** subroutine for details about defining a polygon vertex; and the **gluNextContour** subroutine for details about describing polygons with multiple contours.) Finally, call the **gluEndPolygon** subroutine to signal the end of the definition.

Once the **gluEndPolygon** subroutine is called, the polygon is tessellated and the resulting triangles are described through the callbacks. (See the **gluTessCallback** subroutine for a list of definitions for the callback routines.)

Parameters

tess	Specifies the tessellation object created with the **gluNewTess** subroutine.

Notes

This command is obsolete and is provided for backward compatibility only. Calls to **gluBeginPolygon** are mapped to **gluTessBeginPolygon** followed by **gluTessBeginContour**. Calls to **gluEndPolygon** are mapped to **gluTessEndContour** followed by **gluTessEndPolygon**.

Examples

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

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

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **gluNewTess** subroutine, **gluNextContour** subroutine, **gluTessBeginContour** subroutine, **gluTessBeginPolygon** subroutine, **gluTessCallback** subroutine, **gluTessVertex** subroutine.

gluBeginSurface or gluEndSurface Subroutine

Purpose

Delimits the beginning or end of a non-uniform rational B-spline (NURBS) surface definition.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

void gluBeginSurface(GLUnurbs* *nurb***)**

void gluEndSurface(GLUnurbs* *nurb***)**

Description

Use the **gluBeginSurface** subroutine to mark the beginning of a NURBS surface definition. After calling the **gluBeginSurface** subroutine, make one or more calls to the **gluNurbsSurface** subroutine to define the attributes of the surface. One (and only one) of these calls must have a surface type of **GL_MAP2_VERTEX_3** or **GL_MAP2_VERTEX_4**.

Use the **gluEndSurface** subroutine to mark the end of the NURBS surface definition.

Trimming of NURBS surfaces is supported with the **gluBeginTrim, gluPwlCurve**, **gluNurbsCurve**, and **gluEndTrim** subroutines. (See the **gluBeginTrim** subroutine for details about delimiting a NURBS trimming loop.)

OpenGL evaluators render the NURBS surface as a series of polygons. Evaluator state is preserved during rendering with the **glPushAttrib** (**GL_EVAL_BIT**) and **glPopAttrib**( ) attributes. (See the **glPushAttrib** for details on what state these calls preserve.)

Parameters

nurb Specifies the NURBS object created with the **gluNewNurbsRenderer** subroutine.

Examples

The following commands render a textured NURBS surface with normals. Texture coordinates and normals are also specified as NURBS surfaces.

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

Files

/usr/include/GL/gl.h Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glPushAttrib** subroutine, **gluBeginCurve** subroutine, **gluBeginTrim** subroutine, **gluNewNurbsRenderer** subroutine, **gluNurbsCurve** subroutine, **gluNurbsSurface** subroutine, **gluPwlCurve** subroutine.

gluBeginTrim or gluEndTrim Subroutine

Purpose

Delimits the beginning or end of a non-uniform rational B-spline (NURBS) trimming loop definition.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

void gluBeginTrim(GLUnurbs* *nurb***)**

void gluEndTrim(GLUnurbs* *nurb***)**

Description

Use the **gluBeginTrim** subroutine to mark the beginning of a NURBS trimming loop. A *trimming loop* is a set of oriented curve segments (forming a closed curve) that define boundaries of a NURBS surface. Trimming loops are included in a NURBS surface definition between calls to the **gluBeginSurface** and **gluEndSurface** subroutine pair.

Use the **gluEndTrim** subroutine to mark the end of a trimming loop.

The definition for a NURBS surface can contain multiple trimming loops. For example, if a NURBS surface definition resembles a rectangle with a hole through it, the definition contains two trimming loops. One trimming loop defines the outer edge of the rectangle and the other defines the hole in the rectangle. Definitions for each of these trimming loops are bracketed by a **gluBeginTrim** and **gluEndTrim** subroutine pair.

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

The region of the NURBS surface displayed is in the domain to the left of the trimming curve as the curve parameter increases. Therefore, the retained region of the NURBS surface is inside a counterclockwise trimming loop and outside a clockwise trimming loop. Using the rectangle with the hole mentioned in the preceding example, the trimming loop for the outer edge of the rectangle runs counterclockwise; the trimming loop for the hole runs clockwise.

If you use more than one curve to define a single trimming loop, the curve segments must form a closed loop. That is, the endpoint of each curve must be the starting point of the next curve and the endpoint of the final curve must be the starting point of the first curve. If the endpoints of these curves are sufficiently close together but not precisely coincident, they are forced to meet. If the endpoints are not sufficiently close, an error is generated. (See **gluNurbsCallback** for details on defining a NURBS object callback.)

If a trimming loop definition contains multiple curves, the direction of the curves must be consistent. (The inside must be to the left of the curves.) Nested trimming loops are acceptable as long as curve orientations alternate correctly. Trimming curves cannot be self-intersecting; nor can they intersect each other.

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

Parameters

nurb Specifies the NURBS object created with the **gluNewNurbsRenderer** subroutine.

Examples

This code fragment defines a trimming loop that consists of one piecewise linear curve and two NURBS curves:

```
gluBeginTrim(nobj);
   gluPwlCurve(..., GL_MAP1_TRIM_2);
   gluNurbsCurve(..., GL_MAP1_TRIM_2);
   gluNurbsCurve(..., GL_MAP1_TRIM_3);
gluEndTrim(nobj);
```

Files

/usr/include/GL/gl.h Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **gluBeginSurface** subroutine, **gluNewNurbsRenderer** subroutine, **gluNurbsCallback** subroutine, **gluNurbsCurve** subroutine, **gluPwlCurve** subroutine.

gluBuild1DMipmapLevels Subroutine

Purpose

Builds a subset of 1D mipmap levels.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLint gluBuild1DMipmapLevels( GLenum  target,
   GLint  internalFormat,
   GLsizei  width,
   GLenum  format,
   GLenum  type,
   GLint  level,
   GLint  base,
   GLint  max,
   const void * data )
```

Description

gluBuild1DMipmapLevels builds a subset of prefiltered 1D texture maps of decreasing resolutions called a mipmap. This is used for the antialiasing of texture mapped primitives.

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

A series of mipmap levels from *base* to *max* is built by decimating *data* in half until size 1x1 is reached. At each level, each texel in the halved mipmap level is an average of the corresponding two texels in the larger mipmap level. **glTexImage1D** is called to load these mipmap levels from *base* to *max*. If *max* is larger than the highest mipmap level for the texture of the specified size, then a GLU error code is returned (see **gluErrorString**) and nothing is loaded.

For example, if *level* is 2 and *width* is 16, the following levels are possible: 16x1, 8x1, 4x1, 2x1, 1x1. These correspond to levels 2 through 6 respectively. If *base* is 3 and *max* is 5, then only mipmap levels 8x1, 4x1 and 2x1 are loaded. However, if *max* is 7 then an error is returned and nothing is loaded since *max* is larger than the highest mipmap level which is, in this case, 6.

The highest mipmap level can be derived from the formula log2(*width*)*(2^*level*)). See the **glTexImage1D** reference page for a description of the acceptable values for *type* parameter. See the **glDrawPixels** reference page for a description of the acceptable values for *level* parameter.

Parameters

target Specifies the target texture. Must be **GL_TEXTURE_1D**.

internalFormat

Requests the internal storage format of the texture image. Must be 1, 2, 3, or 4 or one of the following symbolic constants:

- **GL_ABGR_EXT**
- **GL_ALPHA**
- **GL_ALPHA4**
- **GL_ALPHA8**
- **GL_ALPHA12**
- **GL_ALPHA16**
- **GL_LUMINANCE**
- **GL_LUMINANCE4**
- **GL_LUMINANCE8**
- **GL_LUMINANCE12**
- **GL_LUMINANCE16**
- **GL_LUMINANCE_ALPHA**
- **GL_LUMINANCE4_ALPHA4**
- **GL_LUMINANCE6_ALPHA2**
- **GL_LUMINANCE8_ALPHA8**
- **GL_LUMINANCE12_ALPHA4**
- **GL_LUMINANCE12_ALPHA12**
- **GL_LUMINANCE16_ALPHA16**
- **GL_INTENSITY**
- **GL_INTENSITY4**
- **GL_INTENSITY8**
- **GL_INTENSITY12**
- **GL_INTENSITY16**
- **GL_RGB**
- **GL_R3_G3_B2**
- **GL_RGB4**
- **GL_RGB5**
- **GL_RGB8**
- **GL_RGB10**
- **GL_RGB12**
- **GL_RGB16**
- **GL_RGBA**
- **GL_RGBA2**
- **GL_RGBA4**
- **GL_RGB5_A1**
- **GL_RGBA8**
- **GL_RGB10_A2**
- **GL_RGBA12**
- **GL_RGBA16**

width

Specifies the width in pixels of the texture image. This should be a power of 2.

format	Specifies the format of the pixel data. Must be one of: • **GL_COLOR_INDEX** • **GL_DEPTH_COMPONENT** • **GL_RED** • **GL_GREEN** • **GL_BLUE** • **GL_ALPHA** • **GL_RGB** • **GL_RGBA** • **GL_BGRA** • **GL_LUMINANCE** • **GL_LUMINANCE_ALPHA**
type	Specifies the data type for *data*. Must be one of: • **GL_UNSIGNED_BYTE** • **GL_BYTE** • **GL_BITMAP** • **GL_UNSIGNED_SHORT** • **GL_SHORT** • **GL_UNSIGNED_INT** • **GL_INT** • **GL_FLOAT** • **GL_UNSIGNED_BYTE_3_3_2** • **GL_UNSIGNED_BYTE_2_3_3_REV** • **GL_UNSIGNED_SHORT_5_6_5** • **GL_UNSIGNED_SHORT_5_6_5_REV** • **GL_UNSIGNED_SHORT_4_4_4_4** • **GL_UNSIGNED_SHORT_4_4_4_4_REV** • **GL_UNSIGNED_SHORT_5_5_5_1** • **GL_UNSIGNED_SHORT_1_5_5_5_REV** • **GL_UNSIGNED_INT_8_8_8_8** • **GL_UNSIGNED_INT_8_8_8_8_REV** • **GL_UNSIGNED_INT_10_10_10_2** • **GL_UNSIGNED_INT_2_10_10_10_REV**
level	Specifies the mipmap level of the image data.
base	Specifies the minimum mipmap level to pass to **glTexImage1D**.
max	Specifies the maximum mipmap level to pass to **glTexImage1D**.
data	Specifies a pointer to the image data in memory.

Notes

GL_ABGR_EXT is only valid if the **GL_EXT_abgr** extension is defined.

Error Codes

- **GLU_INVALID_VALUE** is returned if *level* > *base*, *base* < 0, *max*< *base* or *max* is > the highest mipmap level for *data*.
- **GLU_INVALID_VALUE** is returned if *width* is < 1.
- **GLU_INVALID_ENUM** is returned if *internalFormat*, *format* or *type* are not legal.

- **GLU_INVALID_OPERATION** is returned if *level* is **GL_UNSIGNED_BYTE_3_3_2** or **GL_UNSIGNED_BYTE_2_3_3_REV** and *type* is not **GL_RGB**.
- **GLU_INVALID_OPERATION** is returned if *level* is **GL_UNSIGNED_SHORT_5_6_5** or **GL_UNSIGNED_SHORT_5_6_5_REV** and *type* is not **GL_RGB**.
- **GLU_INVALID_OPERATION** is returned if *level* is **GL_UNSIGNED_SHORT_4_4_4_4** or **GL_UNSIGNED_SHORT_4_4_4_4_REV** and *type* is neither **GL_RGBA** nor **GL_BGRA**.
- **GLU_INVALID_OPERATION** is returned if *level* is **GL_UNSIGNED_SHORT_5_5_5_1** or **GL_UNSIGNED_SHORT_1_5_5_5_REV** and *type* is neither **GL_RGBA** nor **GL_BGRA**.
- **GLU_INVALID_OPERATION** is returned if *level* is **GL_UNSIGNED_INT_8_8_8_8** or **GL_UNSIGNED_INT_8_8_8_8_REV** and *type* is neither **GL_RGBA** nor **GL_BGRA**.
- **GLU_INVALID_OPERATION** is returned if *level* is **GL_UNSIGNED_INT_10_10_10_2** or **GL_UNSIGNED_INT_2_10_10_10_REV** and *type* is neither **GL_RGBA** nor **GL_BGRA**.

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glDrawPixels** subroutine, **glTexImage1D** subroutine, **glTexImage2D** subroutine, **glTexImage3D** subroutine, **gluBuild1DMipmaps** subroutine, **gluBuild2DMipmaps** subroutine, **gluBuild3DMipmaps** subroutine, **gluErrorString** subroutine, **glGetTexImage** subroutine, **glGetTexLevelParameter** subroutine, **gluBuild2DMipmapLevels** subroutine and **gluBuild3DMipmapLevels** subroutine.

gluBuild1DMipmaps Subroutine

Purpose

Creates 1-dimensional (1D) mipmaps.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLint gluBuild1DMipmaps(GLenum  target,
   GLint  internalFormat,
   GLsizei  width,
   GLenum  format,
   GLenum  type,
   const void * data)
```

Description

The **gluBuild1DMipmaps** subroutine builds a series of prefiltered 1D texture maps of decreasing resolutions called a mipmap. This is used for the antialiasing of texture mapped primitives.

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

Initially, the *width* of *data* is checked to see if it is a power of two. If not, a copy of *data* is scaled up or down to the nearest power of two. This copy will be used for subsequent mipmapping operations described below. (If *width* is exactly between powers of 2, then the copy of *data* will scale upwards.) For example, if *width* is 57 then a copy of *data* will scale up to 64 before mipmapping takes place.

Then, proxy textures (see **glTexImage1D**) are used to determine if the implementation can fit the requested texture. If not, *width* is continually halved until it fits.

Next, a series of mipmap levels is built by decimating a copy of *data* in half until size 1x1 is reached. At each level, each texel in the halved mipmap level is an average of the corresponding two texels in the larger mipmap level.

glTexImage1D is called to load each of these mipmap levels. Level 0 is a copy of *data*. The highest level is log2(width). For example, if width is 64 and the implementation can store a texture of this size, the following mipmap levels are built: 64x1, 32x1, 16x1, 8x1, 4x1, 2x1 and 1x1. These correspond to levels 0 through 6, respectively.

See the **glTexImage1D** subroutine for a description of the acceptable values for the *format* parameter. See the **glDrawPixels** subroutine for acceptable values for the *type* parameter.

Parameters

target	Specifies the target texture. This value must be **GL_TEXTURE_1D**.

internalFormat Specifies the number of color components in the texture. Values must be 1, 2, 3, or 4 or one of the following symbolic constants:

- **GL_ABGR_EXT**
- **GL_ALPHA**
- **GL_ALPHA4**
- **GL_ALPHA8**
- **GL_ALPHA12**
- **GL_ALPHA16**
- **GL_LUMINANCE**
- **GL_LUMINANCE4**
- **GL_LUMINANCE8**
- **GL_LUMINANCE12**
- **GL_LUMINANCE16**
- **GL_LUMINANCE_ALPHA**
- **GL_LUMINANCE4_ALPHA4**
- **GL_LUMINANCE6_ALPHA2**
- **GL_LUMINANCE8_ALPHA8**
- **GL_LUMINANCE12_ALPHA4**
- **GL_LUMINANCE12_ALPHA12**
- **GL_LUMINANCE16_ALPHA16**
- **GL_INTENSITY**
- **GL_INTENSITY4**
- **GL_INTENSITY8**
- **GL_INTENSITY12**
- **GL_INTENSITY16**
- **GL_RGB**
- **GL_R3_G3_B2**
- **GL_RGB4**
- **GL_RGB5**
- **GL_RGB8**
- **GL_RGB10**
- **GL_RGB12**
- **GL_RGB16**
- **GL_RGBA**
- **GL_RGBA2**
- **GL_RGBA4**
- **GL_RGB5_A1**
- **GL_RGBA8**
- **GL_RGB10_A2**
- **GL_RGBA12**
- **GL_RGBA16**

width Specifies the width, in pixels, of the texture image.

format	Specifies the format of the pixel data. The following symbolic values are valid: • **GL_COLOR_INDEX** • **GL_DEPTH_COMPONENT** • **GL_RED** • **GL_GREEN** • **GL_BLUE** • **GL_ALPHA** • **GL_RGB** • **GL_RGBA** • **GL_BGRA** • **GL_LUMINANCE** • **GL_LUMINANCE_ALPHA** (See the **glTexImage1D** subroutine for a description of the acceptable values for the *format* parameter.)
type	Specifies the data type. The following data types for *data* are valid: • **GL_UNSIGNED_BYTE** • **GL_BYTE** • **GL_BITMAP** • **GL_UNSIGNED_SHORT** • **GL_SHORT** • **GL_UNSIGNED_INT** • **GL_INT** • **GL_FLOAT** • **GL_UNSIGNED_BYTE_3_3_2** • **GL_UNSIGNED_BYTE_2_3_3_REV** • **GL_UNSIGNED_SHORT_5_6_5** • **GL_UNSIGNED_SHORT_5_6_5_REV** • **GL_UNSIGNED_SHORT_4_4_4_4** • **GL_UNSIGNED_SHORT_4_4_4_4_REV** • **GL_UNSIGNED_SHORT_5_5_5_1** • **GL_UNSIGNED_SHORT_1_5_5_5_REV** • **GL_UNSIGNED_INT_8_8_8_8** • **GL_UNSIGNED_INT_8_8_8_8_REV** • **GL_UNSIGNED_INT_10_10_10_2** • **GL_UNSIGNED_INT_2_10_10_10_REV** (See the **glDrawPixels** subroutine for acceptable values for the *type* parameter.)
data	Specifies a pointer to the image data in memory.

NOTES

GL_ABGR_EXT is only valid if the **GL_EXT_abgr** extension is defined.

Note that there is no direct way of querying the maximum level. This can be derived indirectly via **glGetTexLevelParameter**. First, query for the width actually used at level 0. (The width may not be equal to *width* since proxy textures might have scaled it to fit the implementation.) Then the maximum level can be derived from the formula log2(width).

ERRORS

- **GLU_INVALID_VALUE** is returned if *width* is < 1.
- **GLU_INVALID_ENUM** is returned if *format* or *type* are not legal.
- **GLU_INVALID_OPERATION** is returned if *type* is **GL_UNSIGNED_BYTE_3_3_2** or **GL_UNSIGNED_BYTE_2_3_3_REV** and *format* is not **GL_RGB**.
- **GLU_INVALID_OPERATION** is returned if *type* is **GL_UNSIGNED_SHORT_5_6_5** or **GL_UNSIGNED_SHORT_5_6_5_REV** and *format* is not **GL_RGB**.
- **GLU_INVALID_OPERATION** is returned if *type* is **GL_UNSIGNED_SHORT_4_4_4_4** or **GL_UNSIGNED_SHORT_4_4_4_4_REV** and *format* is neither **GL_RGBA**, nor **GL_BGRA**.
- **GLU_INVALID_OPERATION** is returned if *type* is **GL_UNSIGNED_SHORT_5_5_5_1** or **GL_UNSIGNED_SHORT_1_5_5_5_REV** and *format* is neither **GL_RGBA** nor **GL_BGRA**.
- **GLU_INVALID_OPERATION** is returned if *type* is **GL_UNSIGNED_INT_8_8_8_8** or **GL_UNSIGNED_INT_8_8_8_8_REV** and *format* is neither **GL_RGBA** nor **GL_BGRA**.
- **GLU_INVALID_OPERATION** is returned if *type* is **GL_UNSIGNED_INT_10_10_10_2** or **GL_UNSIGNED_INT_2_10_10_10_REV** and *format* is neither **GL_RGBA** nor **GL_BGRA**.

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **gluBuild1DMipmaps** subroutine, **gluBuild2DMipmaps** subroutine, **gluBuild3DMipmaps** subroutine, **gluBuild1DMipmapLevels** subroutine, **gluBuild2DMipmapLevels** subroutine, **gluBuild3DMipmapLevels** subroutine, **glDrawPixels** subroutine, **glGetTexLevelParameter** subroutine, **glGetTexImage** subroutine, **glTexImage1D** subroutine, **glTexImage2D** subroutine, **glTexImage3D** subroutine .

gluBuild2DMipmapLevels Subroutine

Purpose

Builds a subset of 2D mipmap levels.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLint gluBuild2DMipmapLevels( GLenum  target,
   GLint  internalFormat,
   GLsizei  width,
   GLsizei  height,
   GLenum  format,
   GLenum  type,
   GLint  level,
   GLint  base,
   GLint  max,
   const void * data )
```

Description

gluBuild2DMipmapLevels builds a subset of prefiltered 2D texture maps of decreasing resolutions called a mipmap. This is used for the antialiasing of texture mapped primitives.

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

A series of mipmap levels from *base* to *max* is built by decimating *data* in half along both dimensions until size 1x1 is reached. At each level, each texel in the halved mipmap level is an average of the corresponding four texels in the larger mipmap level. (In the case of rectangular images, the decimation will ultimately reach an N x 1 or 1 x N configuration. Here, two texels are averaged instead.) **glTexImage2D** is called to load these mipmap levels from *base* to *max*. If *max* is larger than the highest mipmap level for the texture of the specified size, then a GLU error code is returned (see **gluErrorString**) and nothing is loaded.

For example, if *level* is 2 and *width* is 16 and *height* is 8, the following levels are possible: 16x8, 8x4, 4x2, 2x1, 1x1. These correspond to levels 2 through 6 respectively. If *base* is 3 and *max* is 5, then only mipmap levels 8x4, 4x2 and 2x1 are loaded. However, if *max* is 7 then an error is returned and nothing is loaded since *max* is larger than the highest mipmap level which is, in this case, 6.

The highest mipmap level can be derived from the formula log2(max(*width*,*height*)*(2^*level*)).

See the **glTexImage1D** subroutine for a description of the acceptable values for *format* parameter. See the **glDrawPixels** subroutine for a description of the acceptable values for *type* parameter.

Parameters

target	Specifies the target texture. Must be **GL_TEXTURE_2D**.

internalFormat

Requests the internal storage format of the texture image. Must be 1, 2, 3, or 4 or one of the following symbolic constants:

- **GL_ABGR_EXT**
- **GL_ALPHA**
- **GL_ALPHA4**
- **GL_ALPHA8**
- **GL_ALPHA12**
- **GL_ALPHA16**
- **GL_LUMINANCE**
- **GL_LUMINANCE4**
- **GL_LUMINANCE8**
- **GL_LUMINANCE12**
- **GL_LUMINANCE16**
- **GL_LUMINANCE_ALPHA**
- **GL_LUMINANCE4_ALPHA4**
- **GL_LUMINANCE6_ALPHA2**
- **GL_LUMINANCE8_ALPHA8**
- **GL_LUMINANCE12_ALPHA4**
- **GL_LUMINANCE12_ALPHA12**
- **GL_LUMINANCE16_ALPHA16**
- **GL_INTENSITY**
- **GL_INTENSITY4**
- **GL_INTENSITY8**
- **GL_INTENSITY12**
- **GL_INTENSITY16**
- **GL_RGB**
- **GL_R3_G3_B2**
- **GL_RGB4**
- **GL_RGB5**
- **GL_RGB8**
- **GL_RGB10**
- **GL_RGB12**
- **GL_RGB16**
- **GL_RGBA**
- **GL_RGBA2**
- **GL_RGBA4**
- **GL_RGB5_A1**
- **GL_RGBA8**
- **GL_RGB10_A2**
- **GL_RGBA12**
- **GL_RGBA16**

width, *height*

Specifies the width and height, respectively, in pixels of the texture image. These should be a power of 2.

format	Specifies the format of the pixel data. Must be one of: • **GL_COLOR_INDEX** • **GL_DEPTH_COMPONENT** • **GL_RED** • **GL_GREEN** • **GL_BLUE** • **GL_ALPHA** • **GL_RGB** • **GL_RGBA** • **GL_BGRA** • **GL_LUMINANCE** • **GL_LUMINANCE_ALPHA**
type	Specifies the data type for *data*. Must be one of: • **GL_UNSIGNED_BYTE** • **GL_BYTE** • **GL_BITMAP** • **GL_UNSIGNED_SHORT** • **GL_SHORT** • **GL_UNSIGNED_INT** • **GL_INT** • **GL_FLOAT** • **GL_UNSIGNED_BYTE_3_3_2** • **GL_UNSIGNED_BYTE_2_3_3_REV** • **GL_UNSIGNED_SHORT_5_6_5** • **GL_UNSIGNED_SHORT_5_6_5_REV** • **GL_UNSIGNED_SHORT_4_4_4_4** • **GL_UNSIGNED_SHORT_4_4_4_4_REV** • **GL_UNSIGNED_SHORT_5_5_5_1** • **GL_UNSIGNED_SHORT_1_5_5_5_REV** • **GL_UNSIGNED_INT_8_8_8_8** • **GL_UNSIGNED_INT_8_8_8_8_REV** • **GL_UNSIGNED_INT_10_10_10_2** • **GL_UNSIGNED_INT_2_10_10_10_REV**
level	Specifies the mipmap level of the image data.
base	Specifies the minimum mipmap level to pass to **glTexImage2D**.
max	Specifies the maximum mipmap level to pass to **glTexImage2D**.
data	Specifies a pointer to the image data in memory.

Notes

GL_ABGR_EXT is only valid if the **GL_EXT_abgr** extension is defined.

Error Codes

GLU_INVALID_VALUE is returned if *level* > *base*, *base* < 0, *max* < *base* or *max* is > the highest mipmap level for *data*.

GLU_INVALID_VALUE is returned if *width* or *height* are < 1.

GLU_INVALID_ENUM is returned if *internalFormat*, *format* or *type* are not legal.

GLU_INVALID_OPERATION is returned if *type* is **GL_UNSIGNED_BYTE_3_3_2** or **GL_UNSIGNED_BYTE_2_3_3_REV** and *format* is not **GL_RGB**.

GLU_INVALID_OPERATION is returned if *type* is **GL_UNSIGNED_SHORT_5_6_5** or **GL_UNSIGNED_SHORT_5_6_5_REV** and *format* is not **GL_RGB**.

GLU_INVALID_OPERATION is returned if *type* is **GL_UNSIGNED_SHORT_4_4_4_4** or **GL_UNSIGNED_SHORT_4_4_4_4_REV** and *format* is neither **GL_RGBA** nor **GL_BGRA**.

GLU_INVALID_OPERATION is returned if *type* is **GL_UNSIGNED_SHORT_5_5_5_1** or **GL_UNSIGNED_SHORT_1_5_5_5_REV** and *format* is neither **GL_RGBA** nor **GL_BGRA**.

GLU_INVALID_OPERATION is returned if *type* is **GL_UNSIGNED_INT_8_8_8_8** or **GL_UNSIGNED_INT_8_8_8_8_REV** and *format* is neither **GL_RGBA** nor **GL_BGRA**.

GLU_INVALID_OPERATION is returned if *type* is **GL_UNSIGNED_INT_10_10_10_2** or **GL_UNSIGNED_INT_2_10_10_10_REV** and *format* is neither **GL_RGBA** nor **GL_BGRA**.

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glDrawPixels** subroutine, **glTexImage1D** subroutine, **glTexImage2D** subroutine, **glTexImage3D** subroutine, **gluBuild1DMipmaps** subroutine, **gluBuild2DMipmaps** subroutine, **gluBuild3DMipmaps** subroutine, **gluErrorString** subroutine, **glGetTexImage** subroutine, **glGetTexLevelParameter** subroutine, **gluBuild1DMipmapLevels** subroutine, **gluBuild3DMipmapLevels** subroutine.

gluBuild2DMipmaps Subroutine

Purpose

Creates 2-dimensional (2D) mipmaps.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLint gluBuild2DMipmaps(GLenum  target,
   GLint  internalFormat,
   GLsizei width,
   GLsizei  height,
   GLenum  format,
   GLenum  type,
   const void * data)
```

Description

The **gluBuild2DMipmaps** subroutine builds a series of prefiltered 2-D texture maps of decreasing resolutions called a mipmap. This is used for the antialiasing of texture mapped primitives.

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

Initially, the *width* and *height* of *data* are checked to see if they are a power of two. If not, a copy of *data* (not *data*), is scaled up or down to the nearest power of two. This copy will be used for subsequent mipmapping operations described below. (If *width* or *height* is exactly between powers of 2, then the copy of *data* will scale upwards.) For example, if *width* is 57 and *height* is 23 then a copy of *data* will scale up to 64 in width and down to 16 in depth, before mipmapping takes place.

Then, proxy textures (see **glTexImage2D**) are used to determine if the implementation can fit the requested texture. If not, both dimensions are continually halved until it fits. (If the OpenGL version is <= 1.0, both maximum texture dimensions are clamped to the value returned by **glGetIntegerv** with the argument **GL_MAX_TEXTURE_SIZE**.)

Next, a series of mipmap levels is built by decimating a copy of *data* in half along both dimensions until size 1x1 is reached. At each level, each texel in the halved mipmap level is an average of the corresponding four texels in the larger mipmap level. (In the case of rectangular images, the decimation will ultimately reach an N x 1 or 1 x N configuration. Here, two texels are averaged instead.)

glTexImage2D is called to load each of these mipmap levels. Level 0 is a copy of *data*. The highest level is log2(max(width,height)). For example, if width is 64 and height is 16 and the implementation can store a texture of this size, the following mipmap levels are built: 64x16, 32x8, 16x4, 8x2, 4x1, 2x1 and 1x1. These correspond to levels 0 through 6, respectively.

See the **glTexImage1D** subroutine for a description of the acceptable values for *format* parameter. See the **glDrawPixels** subroutine for a description of the acceptable values for *type* parameter.

Parameters

target	Specifies the target texture. This value must be **GL_TEXTURE_2D**.

internalFormat	Specifies the number of color components in the texture. Values must be 1, 2, 3, or 4 or one of the following symbolic constants: • **GL_ABGR_EXT** • **GL_ALPHA** • **GL_ALPHA4** • **GL_ALPHA8** • **GL_ALPHA12** • **GL_ALPHA16** • **GL_LUMINANCE** • **GL_LUMINANCE4** • **GL_LUMINANCE8** • **GL_LUMINANCE12** • **GL_LUMINANCE16** • **GL_LUMINANCE_ALPHA** • **GL_LUMINANCE4_ALPHA4** • **GL_LUMINANCE6_ALPHA2** • **GL_LUMINANCE8_ALPHA8** • **GL_LUMINANCE12_ALPHA4** • **GL_LUMINANCE12_ALPHA12** • **GL_LUMINANCE16_ALPHA16** • **GL_INTENSITY** • **GL_INTENSITY4** • **GL_INTENSITY8** • **GL_INTENSITY12** • **GL_INTENSITY16** • **GL_RGB** • **GL_R3_G3_B2** • **GL_RGB4** • **GL_RGB5** • **GL_RGB8** • **GL_RGB10** • **GL_RGB12** • **GL_RGB16** • **GL_RGBA** • **GL_RGBA2** • **GL_RGBA4** • **GL_RGB5_A1** • **GL_RGBA8** • **GL_RGB10_A2** • **GL_RGBA12** • **GL_RGBA16**
width	Specifies the width, in pixels, of the texture image.
height	Specifies the height, in pixels, of the texture image.

format Specifies the format of the pixel data. The following symbolic values are valid:

- **GL_COLOR_INDEX**
- **GL_DEPTH_COMPONENT**
- **GL_RED**
- **GL_GREEN**
- **GL_BLUE**
- **GL_ALPHA**
- **GL_RGB**
- **GL_RGBA**
- **GL_BGRA**
- **GL_LUMINANCE**
- **GL_LUMINANCE_ALPHA**

(See the **glTexImage1D** subroutine for acceptable values for the *Format* parameter.)

type Specifies the data type. The following data types are valid for *data*:

- **GL_UNSIGNED_BYTE**
- **GL_BYTE**
- **GL_BITMAP**
- **GL_UNSIGNED_SHORT**
- **GL_SHORT**
- **GL_UNSIGNED_INT**
- **GL_INT**
- **GL_FLOAT**
- **GL_UNSIGNED_BYTE_3_3_2**
- **GL_UNSIGNED_BYTE_2_3_3_REV**
- **GL_UNSIGNED_SHORT_5_6_5**
- **GL_UNSIGNED_SHORT_5_6_5_REV**
- **GL_UNSIGNED_SHORT_4_4_4_4**
- **GL_UNSIGNED_SHORT_4_4_4_4_REV**
- **GL_UNSIGNED_SHORT_5_5_5_1**
- **GL_UNSIGNED_SHORT_1_5_5_5_REV**
- **GL_UNSIGNED_INT_8_8_8_8**
- **GL_UNSIGNED_INT_8_8_8_8_REV**
- **GL_UNSIGNED_INT_10_10_10_2**
- **GL_UNSIGNED_INT_2_10_10_10_REV**

(See the **glDrawPixels** subroutine for acceptable values for the *Type* parameter.)

data Specifies a pointer to the image data in memory.

Notes

GL_ABGR_EXT is only valid if the **GL_EXT_abgr** extension is defined.

There is no direct way of querying the maximum level. This can be derived indirectly via **glGetTexLevelParameter**. First, query for the width & height actually used at level 0. (The width & height may not be equal to *width* & *height* respectively since proxy textures might have scaled them to fit the implementation.) Then the maximum level can be derived from the formula log2(max(width,height)).

Error Codes

- **GLU_INVALID_VALUE** is returned if *width* or *height* are < 1.
- **GLU_INVALID_ENUM** is returned if *internalFormat*, *format* or *type* are not legal.

- **GLU_INVALID_OPERATION** is returned if *type* is **GL_UNSIGNED_BYTE_3_3_2** or **GL_UNSIGNED_BYTE_2_3_3_REV** and *format* is not **GL_RGB**.
- **GLU_INVALID_OPERATION** is returned if *type* is **GL_UNSIGNED_SHORT_5_6_5** or **GL_UNSIGNED_SHORT_5_6_5_REV** and *format* is not **GL_RGB**.
- **GLU_INVALID_OPERATION** is returned if *type* is **GL_UNSIGNED_SHORT_4_4_4_4** or **GL_UNSIGNED_SHORT_4_4_4_4_REV** and *format* is neither **GL_RGBA** nor **GL_BGRA**.
- **GLU_INVALID_OPERATION** is returned if *type* is **GL_UNSIGNED_SHORT_5_5_5_1** or **GL_UNSIGNED_SHORT_1_5_5_5_REV** and *format* is neither **GL_RGBA** nor **GL_BGRA**.
- **GLU_INVALID_OPERATION** is returned if *type* is **GL_UNSIGNED_INT_8_8_8_8** or **GL_UNSIGNED_INT_8_8_8_8_REV** and *format* is neither **GL_RGBA** nor **GL_BGRA**.
- **GLU_INVALID_OPERATION** is returned if *type* is **GL_UNSIGNED_INT_10_10_10_2** or **GL_UNSIGNED_INT_2_10_10_10_REV** and *format* is neither **GL_RGBA** nor **GL_BGRA**.

Files

/usr/include/GL/gl.h Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glDrawPixels** subroutine, **glTexImage1D** subroutine, **glTexImage2D** subroutine, **glTexImage3D** subroutine, **gluBuild1DMipmaps** subroutine, **gluBuild3DMipmaps** subroutine, **gluErrorString** subroutine, **glGetTexImage** subroutine, **glGetTexLevelParameter** subroutine, **gluBuild1DMipmapLevels** subroutine, **gluBuild2DMipmapLevels** subroutine, **gluBuild3DMipmapLevels** subroutine .

gluBuild3DMipmapLevels Subroutine

Purpose

Builds a subset of 3D mipmap levels.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLint gluBuild3DMipmapLevels( GLenum  target,
   GLint  internalFormat,
   GLsizei  width,
   GLsizei  height,
   GLsizei  depth,
   GLenum  format,
   GLenum  type,
   GLint  level,
   GLint  base,
   GLint  max,
   const void * data )
```

Description

gluBuild3DMipmapLevels builds a subset of prefiltered 3D texture maps of decreasing resolutions called a mipmap. This is used for the antialiasing of texture mapped primitives.

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

A series of mipmap levels from *base* to *max* is built by decimating *data* in half along both dimensions until size 1x1x1 is reached. At each level, each texel in the halved mipmap level is an average of the corresponding eight texels in the larger mipmap level. (If exactly one of the dimensions is 1, four texels are averaged. If exactly two of the dimensions are 1, two texels are averaged.) **glTexImage3D** is called to load these mipmap levels from *base* to *max*. If *max* is larger than the highest mipmap level for the texture of the specified size, then a GLU error code is returned (see **gluErrorString**) and nothing is loaded.

For example, if *level* is 2 and *width* is 16, *height* is 8 and *depth* is 4, the following levels are possible: 16x8x4, 8x4x2, 4x2x1, 2x1x1, 1x1x1. These correspond to levels 2 through 6 respectively. If *base* is 3 and *max* is 5, then only mipmap levels 8x4x2, 4x2x1 and 2x1x1 are loaded. However, if *max* is 7 then an error is returned and nothing is loaded since *max* is larger than the highest mipmap level which is, in this case, 6.

The highest mipmap level can be derived from the formula log2(max(*width*,*height*,*depth*)*(2^*level*)).

See the **glTexImage1D** subroutine for a description of the acceptable values for *format* parameter. See the **glDrawPixels** subroutine for a description of the acceptable values for *type* parameter.

Parameters

target	Specifies the target texture. Must be **GL_TEXTURE_3D**.

internalFormat	Requests the internal storage format of the texture image. Must be 1, 2, 3, or 4 or one of the following symbolic constants: • **GL_ABGR_EXT** • **GL_ALPHA** • **GL_ALPHA4** • **GL_ALPHA8** • **GL_ALPHA12** • **GL_ALPHA16** • **GL_LUMINANCE** • **GL_LUMINANCE4** • **GL_LUMINANCE8** • **GL_LUMINANCE12** • **GL_LUMINANCE16** • **GL_LUMINANCE_ALPHA** • **GL_LUMINANCE4_ALPHA4** • **GL_LUMINANCE6_ALPHA2** • **GL_LUMINANCE8_ALPHA8** • **GL_LUMINANCE12_ALPHA4** • **GL_LUMINANCE12_ALPHA12** • **GL_LUMINANCE16_ALPHA16** • **GL_INTENSITY** • **GL_INTENSITY4** • **GL_INTENSITY8** • **GL_INTENSITY12** • **GL_INTENSITY16** • **GL_RGB** • **GL_R3_G3_B2** • **GL_RGB4** • **GL_RGB5** • **GL_RGB8** • **GL_RGB10** • **GL_RGB12** • **GL_RGB16** • **GL_RGBA** • **GL_RGBA2** • **GL_RGBA4** • **GL_RGB5_A1** • **GL_RGBA8** • **GL_RGB10_A2** • **GL_RGBA12** • **GL_RGBA16**
width	Specifies the width, in pixels, of the texture image. These should be a power of 2.
height	Specifies the height, in pixels, of the texture image. These should be a power of 2.
depth	Specifies the depth, in pixels, of the texture image. These should be a power of 2.

format
Specifies the format of the pixel data. Must be one of:

- **GL_COLOR_INDEX**
- **GL_DEPTH_COMPONENT**
- **GL_RED**
- **GL_GREEN**
- **GL_BLUE**
- **GL_ALPHA**
- **GL_RGB**
- **GL_RGBA**
- **GL_BGRA**
- **GL_LUMINANCE**
- **GL_LUMINANCE_ALPHA**

type
Specifies the data type for *data*. Must be one of:

- **GL_UNSIGNED_BYTE**
- **GL_BYTE**
- **GL_BITMAP**
- **GL_UNSIGNED_SHORT**
- **GL_SHORT**
- **GL_UNSIGNED_INT**
- **GL_INT**
- **GL_FLOAT**
- **GL_UNSIGNED_BYTE_3_3_2**
- **GL_UNSIGNED_BYTE_2_3_3_REV**
- **GL_UNSIGNED_SHORT_5_6_5**
- **GL_UNSIGNED_SHORT_5_6_5_REV**
- **GL_UNSIGNED_SHORT_4_4_4_4**
- **GL_UNSIGNED_SHORT_4_4_4_4_REV**
- **GL_UNSIGNED_SHORT_5_5_5_1**
- **GL_UNSIGNED_SHORT_1_5_5_5_REV**
- **GL_UNSIGNED_INT_8_8_8_8**
- **GL_UNSIGNED_INT_8_8_8_8_REV**
- **GL_UNSIGNED_INT_10_10_10_2**
- **GL_UNSIGNED_INT_2_10_10_10_REV**

level
Specifies the mipmap level of the image data.

base
Specifies the minimum mipmap level to pass to **glTexImage3D**.

max
Specifies the maximum mipmap level to pass to **glTexImage3D**.

data
Specifies a pointer to the image data in memory.

Notes

GL_ABGR_EXT is only valid if the **GL_EXT_abgr** extension is defined.

Error Codes

GLU_INVALID_VALUE is returned if *level* > *base*, *base* < 0, *max* < *base* or *max* is > the highest mipmap level for *data*.

GLU_INVALID_VALUE is returned if *width*, *height* or *depth* are < 1.

GLU_INVALID_ENUM is returned if *internalFormat*, *format* or *type* are not legal.

GLU_INVALID_OPERATION is returned if *type* is **GL_UNSIGNED_BYTE_3_3_2** or **GL_UNSIGNED_BYTE_2_3_3_REV** and *format* is not **GL_RGB**.

GLU_INVALID_OPERATION is returned if *type* is **GL_UNSIGNED_SHORT_5_6_5** or **GL_UNSIGNED_SHORT_5_6_5_REV** and *format* is not **GL_RGB**.

GLU_INVALID_OPERATION is returned if *type* is **GL_UNSIGNED_SHORT_4_4_4_4** or **GL_UNSIGNED_SHORT_4_4_4_4_REV** and *format* is neither **GL_RGBA** nor **GL_BGRA**.

GLU_INVALID_OPERATION is returned if *type* is **GL_UNSIGNED_SHORT_5_5_5_1** or **GL_UNSIGNED_SHORT_1_5_5_5_REV** and *format* is neither **GL_RGBA** nor **GL_BGRA**.

GLU_INVALID_OPERATION is returned if *type* is **GL_UNSIGNED_INT_8_8_8_8** or **GL_UNSIGNED_INT_8_8_8_8_REV** and *format* is neither **GL_RGBA** nor **GL_BGRA**.

GLU_INVALID_OPERATION is returned if *type* is **GL_UNSIGNED_INT_10_10_10_2** or **GL_UNSIGNED_INT_2_10_10_10_REV** and *format* is neither **GL_RGBA** nor **GL_BGRA**.

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

glDrawPixels subroutine, **glTexImage1D** subroutine, **glTexImage2D** subroutine, **glTexImage3D** subroutine, **gluBuild1DMipmaps** subroutine, **gluBuild2DMipmaps** subroutine, **gluBuild3DMipmaps** subroutine, **gluErrorString** subroutine, **glGetTexImage** subroutine, **glGetTexLevelParameter** subroutine, **gluBuild1DMipmapLevels** subroutine, **gluBuild2DMipmapLevels** subroutine.

gluBuild3DMipmaps Subroutine

Purpose

Builds a 3-D mipmap.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLint gluBuild3DMipmaps( GLenum  target,
   GLint  internalFormat,
   GLsizei  width,
   GLsizei  height,
   GLsizei  depth,
   GLenum  format,
   GLenum  type,
   const void * data )
```

Description

gluBuild3DMipmaps builds a series of prefiltered 3D texture maps of decreasing resolutions called a mipmap. This is used for the antialiasing of texture mapped primitives.

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

Initially, the *width*, *height* and *depth* of *data* are checked to see if they are a power of two. If not, a copy of *data* (not *data*), is scaled up or down to the nearest power of two. This copy will be used for subsequent mipmapping operations described below. (If *width*, *height* or *depth* is exactly between powers of 2, then the copy of *data* will scale upwards.) For example, if *width* is 57, *height* is 23 and *depth* is 24 then a copy of *data* will scale up to 64 in width, down to 16 in height and up to 32 in depth, before mipmapping takes place.

Then, proxy textures (see **glTexImage3D**) are used to determine if the implementation can fit the requested texture. If not, all three dimensions are continually halved until it fits.

Next, a series of mipmap levels is built by decimating a copy of *data* in half along all three dimensions until size 1x1x1 is reached. At each level, each texel in the halved mipmap level is an average of the corresponding eight texels in the larger mipmap level. (If exactly one of the dimensions is 1, four texels are averaged. If exactly two of the dimensions are 1, two texels are averaged.)

glTexImage3D is called to load each of these mipmap levels. Level 0 is a copy of *data*. The highest level is log2(max(width,height,depth)). For example, if width is 64, height is 16 and depth is 32, and the implementation can store a texture of this size, the following mipmap levels are built: 64x16x32, 32x8x16, 16x4x8, 8x2x4, 4x1x2, 2x1x1 and 1x1x1. These correspond to levels 0 through 6, respectively.

See the **glTexImage1D** subroutine for a description of the acceptable values for *format* parameter. See the **glDrawPixels** subroutine for a description of the acceptable values for *type* parameter.

Parameters

target	Specifies the target texture. Must be **GL_TEXTURE_3D**.

internalFormat
Requests the internal storage format of the texture image. Must be 1, 2, 3, or 4 or one of the following symbolic constants:

- **GL_ABGR_EXT**
- **GL_ALPHA**
- **GL_ALPHA4**
- **GL_ALPHA8**
- **GL_ALPHA12**
- **GL_ALPHA16**
- **GL_LUMINANCE**
- **GL_LUMINANCE4**
- **GL_LUMINANCE8**
- **GL_LUMINANCE12**
- **GL_LUMINANCE16**
- **GL_LUMINANCE_ALPHA**
- **GL_LUMINANCE4_ALPHA4**
- **GL_LUMINANCE6_ALPHA2**
- **GL_LUMINANCE8_ALPHA8**
- **GL_LUMINANCE12_ALPHA4**
- **GL_LUMINANCE12_ALPHA12**
- **GL_LUMINANCE16_ALPHA16**
- **GL_INTENSITY**
- **GL_INTENSITY4**
- **GL_INTENSITY8**
- **GL_INTENSITY12**
- **GL_INTENSITY16**
- **GL_RGB**
- **GL_R3_G3_B2**
- **GL_RGB4**
- **GL_RGB5**
- **GL_RGB8**
- **GL_RGB10**
- **GL_RGB12**
- **GL_RGB16**
- **GL_RGBA**
- **GL_RGBA2**
- **GL_RGBA4**
- **GL_RGB5_A1**
- **GL_RGBA8**
- **GL_RGB10_A2**
- **GL_RGBA12**
- **GL_RGBA16**

width
Specifies the width, in pixels, of the texture image.

height
Specifies the height, in pixels, of the texture image.

depth
Specifies the depth, in pixels, of the texture image.

format
Specifies the format of the pixel data. Must be one of:
- **GL_COLOR_INDEX**
- **GL_DEPTH_COMPONENT**
- **GL_RED**
- **GL_GREEN**
- **GL_BLUE**
- **GL_ALPHA**
- **GL_RGB**
- **GL_RGBA**
- **GL_BGRA**
- **GL_LUMINANCE**
- **GL_LUMINANCE_ALPHA**

type
Specifies the data type for *data*. Must be one of:
- **GL_UNSIGNED_BYTE**
- **GL_BYTE**
- **GL_BITMAP**
- **GL_UNSIGNED_SHORT**
- **GL_SHORT**
- **GL_UNSIGNED_INT**
- **GL_INT**
- **GL_FLOAT**
- **GL_UNSIGNED_BYTE_3_3_2**
- **GL_UNSIGNED_BYTE_2_3_3_REV**
- **GL_UNSIGNED_SHORT_5_6_5**
- **GL_UNSIGNED_SHORT_5_6_5_REV**
- **GL_UNSIGNED_SHORT_4_4_4_4**
- **GL_UNSIGNED_SHORT_4_4_4_4_REV**
- **GL_UNSIGNED_SHORT_5_5_5_1**
- **GL_UNSIGNED_SHORT_1_5_5_5_REV**
- **GL_UNSIGNED_INT_8_8_8_8**
- **GL_UNSIGNED_INT_8_8_8_8_REV**
- **GL_UNSIGNED_INT_10_10_10_2**
- **GL_UNSIGNED_INT_2_10_10_10_REV**

data
Specifies a pointer to the image data in memory.

Notes

GL_ABGR_EXT is only valid if the **GL_EXT_abgr** extension is defined.

There is no direct way of querying the maximum level. This can be derived indirectly via **glGetTexLevelParameter**. First, query for the width, height and depth actually used at level 0. (The width, height and depth may not be equal to *width*, *height* and *depth* respectively since proxy textures might have scaled them to fit the implementation.) Then the maximum level can be derived from the formula log2(max(width,height,depth)).

Error Codes

GLU_INVALID_VALUE is returned if *width*, *height* or *depth* are < 1.

GLU_INVALID_ENUM is returned if *internalFormat*, *format* or *type* are not legal.

GLU_INVALID_OPERATION is returned if *type* is **GL_UNSIGNED_BYTE_3_3_2** or **GL_UNSIGNED_BYTE_2_3_3_REV** and *format* is not **GL_RGB**.

GLU_INVALID_OPERATION is returned if *type* is **GL_UNSIGNED_SHORT_5_6_5** or **GL_UNSIGNED_SHORT_5_6_5_REV** and *format* is not **GL_RGB**.

GLU_INVALID_OPERATION is returned if *type* is **GL_UNSIGNED_SHORT_4_4_4_4** or **GL_UNSIGNED_SHORT_4_4_4_4_REV** and *format* is neither **GL_RGBA** nor **GL_BGRA**.

GLU_INVALID_OPERATION is returned if *type* is **GL_UNSIGNED_SHORT_5_5_5_1** or **GL_UNSIGNED_SHORT_1_5_5_5_REV** and *format* is neither **GL_RGBA** nor **GL_BGRA**.

GLU_INVALID_OPERATION is returned if *type* is **GL_UNSIGNED_INT_8_8_8_8** or **GL_UNSIGNED_INT_8_8_8_8_REV** and *format* is neither **GL_RGBA** nor **GL_BGRA**.

GLU_INVALID_OPERATION is returned if *type* is **GL_UNSIGNED_INT_10_10_10_2** or **GL_UNSIGNED_INT_2_10_10_10_REV** and *format* is neither **GL_RGBA** nor **GL_BGRA**.

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

- **glDrawPixels**
- **glTexImage1D**
- **glTexImage2D**
- **glTexImage3D**
- **gluBuild1DMipmaps**
- **gluBuild3DMipmaps**
- **gluErrorString**
- **glGetTexImage**
- **glGetTexLevelParameter**
- **gluBuild1DMipmapLevels**
- **gluBuild2DMipmapLevels**
- **gluBuild3DMipmapLevels**

gluCheckExtension Subroutine

Purpose

Determines if an extension name is supported.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLboolean gluCheckExtension( const GLubyte * extName,
   const GLubyte * extString )
```

Description

gluCheckExtension returns **GL_TRUE** if *extName* is supported otherwise **GL_FALSE** is returned.

This is used to check for the presence for OpenGL, GLU or GLX extension names by passing the extensions strings returned by **glGetString**, **gluGetString**, or **glXGetClientString**, respectively, as *extString*.

Parameters

extName	Specifies an extension name.
extString	Specifies a space-separated list of extension names supported.

Notes

Cases where one extension name is a substring of another are correctly handled.

There may or may not be leading or trailing blanks in *extString*.

Extension names should not contain embedded spaces.

All strings are null-terminated.

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glGetString** subroutine, **gluGetString** subroutine, **glXGetClientString** subroutine.

---

gluCylinder Subroutine

Purpose

Draws a cylinder.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void gluCylinder(GLUquadric* quad,
   GLdouble  base,
   GLdouble  top,
   GLdouble  height,
   GLint  slices,
   GLint  stacks)
```

Description

The **gluCylinder** subroutine draws a cylinder that is oriented along the z axis. The base of the cylinder is placed at z = 0; the top of the cylinder is placed at z=*height*. Like a sphere, the cylinder is subdivided around the z axis into slices and along the z axis into stacks.

Note: If the *top* parameter is set to zero, this subroutine will generate a cone.

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

If texturing is turned on using the **gluQuadricTexture** subroutine, texture coordinates are generated so that *t* ranges linearly from 0.0 at z=0 to 1.0 at z=*height*, and *s* ranges from 0.0 at the +y axis to 0.25 at the +x axis, as well as up to 0.5 at the -y axis and 0.75 at the -x axis, then back to 1.0 at the +y axis.

Parameters

quad	Specifies the quadrics object created with the **gluNewQuadric** subroutine.
base	Specifies the radius of the cylinder at z=0.
top	Specifies the radius of the cylinder at z=*Height*. If *top* is set to 0, this subroutine generates a cone.
height	Specifies the height of the cylinder.
slices	Specifies the number of subdivisions around the z axis.
stacks	Specifies the number of subdivisions along the z axis.

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **gluDisk** subroutine, **gluNewQuadric** subroutine, **gluPartialDisk** subroutine, **gluQuadricOrientation** subroutine, **gluQuadricTexture** subroutine, **gluSphere** subroutine.

---

gluDeleteNurbsRenderer Subroutine

Purpose

Destroys a non-uniform rational B-spline (NURBS) object.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

void gluDeleteNurbsRenderer(GLUnurbs* *nurb***)**

Description

The **gluDeleteNurbsRenderer** subroutine destroys the NURBS object and frees any memory used by that object. Once this **gluDeleteNurbsRenderer** subroutine is called, the previously defined value for the *nobj* parameter cannot be used.

Parameters

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

Files

/usr/include/GL/gl.h Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **gluNewNurbsRenderer** subroutine.

gluDeleteQuadric Subroutine

Purpose

Destroys a quadrics object.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

void gluDeleteQuadric(GLUquadric* *quad***)**

Description

The **gluDeleteQuadric** subroutine destroys the quadrics object and frees any memory used by that object. Once the **gluDeleteQuadric** subroutine has been called, the *quad* parameter cannot be used again.

Parameters

quad Specifies the quadrics object (created with the **gluNewQuadric** subroutine) to be destroyed.

Files

/usr/include/GL/gl.h Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **gluNewQuadric** subroutine.

gluDeleteTess Subroutine

Purpose

Destroys a tessellation object.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

void gluDeleteTess(GLUtesselator* *tess***)**

Description

The **gluDeleteTess** subroutine destroys the tessellation object and frees any memory used by that object. Once this subroutine has been called, the value previously defined for the *tess* parameter cannot be used again.

Parameters

tess	Specifies the tessellation object (created with the **gluNewTess** subroutine) to be destroyed.

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **gluBeginPolygon** subroutine, **gluNewTess** subroutine, **gluTessCallback** subroutine.

gluDisk Subroutine

Purpose

Draws a disk.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void gluDisk(GLquadric*  quad,
   GLdouble  inner,
   GLdouble  outer,
   GLint  slices,
   GLint  loops)
```

Description

The **gluDisk** subroutine renders a disk on the z=0 plane. The disk has a radius defined by the *outer* parameter and contains a concentric circular hole with a radius defined by the *inner* parameter. If the value of *inner* is 0, no hole is generated. The disk is subdivided around the z axis into slices and rings (as specified by the *slices* and *loops* parameters, respectively).

With regard to orientation, the +z side of the disk is considered to be *outside*. (See the **gluQuadricOrientation** subroutine for details on specifying quadrics orientation.) If orientation is set to **GLU_OUTSIDE**, any normals generated point along the +z axis. Otherwise, they point along the -z axis.

If texturing is turned on with the **gluQuadricTexture** subroutine, texture coordinates are generated linearly, consistent with the following table:

XYZ Coordinates	(u, v) Texture Coordinates
(outer, 0.0, 0.0)	(1.0, 0.5)
(0.0, outer, 0.0)	(0.5, 1.0)
(-outer, 0.0, 0.0)	(0.5, 0.0)

The formulae are:

texture coordinate u = 0.5 + 0.5 * (x/outer)
texture coordinate v = 0.5 + 0.5 * (y/outer)

Parameters

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

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **gluCylinder** subroutine, **gluNewQuadric** subroutine, **gluPartialDisk** subroutine, **gluQuadricOrientation** subroutine, **gluQuadricTexture** subroutine, **gluSphere** subroutine.

gluErrorString Subroutine

Purpose

Produces an error string from an OpenGL or GLU error code.

Library

`OpenGL C bindings library: libGL.a`

C Syntax

```
const Glubyte * gluErrorString( GLenum  error)
```

Description

The **gluErrorString** subroutine produces an error string from an OpenGL or GLU error code. The format of the string is **ISO_Latin_1**. For example, the code line

```
gluErrorString(GL_OUT_OF_MEMORY)
```

returns the `out of memory` string.

The standard GLU error codes are **GLU_INVALID_ENUM**, **GLU_INVALID_VALID**, and **GLU_OUT_OF_MEMORY**. Certain other GLU functions can return specialized error codes through callbacks. See the **glGetError** subroutine for a list of OpenGL error codes.

Parameters

error	Specifies an OpenGL or GLU error code.

Error Codes

The standard GLU error codes are:

- **GLU_INVALID_ENUM**
- **GLU_INVALID_VALUE**
- **GLU_OUT_OF_MEMORY**

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glGetError** subroutine, **gluNurbsCallback** subroutine, **gluQuadricCallback** subroutine, **gluTessCallback** subroutine.

---

gluGetNurbsProperty Subroutine

Purpose

Retrieves a non-uniform rational B-spline (NURBS) property.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void gluGetNurbsProperty(GLUnurbs*  nurb,
   GLenum  property,
   GLfloat* data )
```

Description

The **gluGetNurbsProperty** subroutine retrieves properties stored in a NURBS object. These properties affect the way that NURBS curves and surfaces are rendered. See the **gluNurbsProperty** subroutine for more information on these properties.

Parameters

nurb Specifies the NURBS object created with the **gluNewNurbsRenderer** subroutine.

property Specifies the property whose value is to be fetched. Valid values for this parameter are:

- **GLU_CULLING**
- **GLU_SAMPLING_TOLERANCE**
- **GLU_DISPLAY_MODE**
- **GLU_AUTO_LOAD_MATRIX**
- **GLU_PARAMETRIC_TOLERANCE**
- **GLU_SAMPLING_METHOD**
- **GLU_U_STEP**
- **GLU_V_STEP**
- **GLU_NURBS_MODE**

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

Files

/usr/include/GL/gl.h — Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **gluNewNurbsRenderer** subroutine, **gluNurbsProperty** subroutine.

gluGetString Subroutine

Purpose

Returns a string describing the GLU version or GLU extensions.

C Syntax

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

Description

The **gluGetString** subroutine returns a pointer to a static string describing the GLU version or the GLU extensions that are supported.

The version number is one of the following forms:

major_number.minor_number
major_number.minor_number.release_number

The version string is of the following form:

version number<space>vendor-specific information

Vendor-specific information is optional. Its format and contents depend on the implementation.

The standard GLU contains a basic set of features and capabilities. If a company or group of companies wish to support other features, these may be included as extensions to the GLU. If *name* is **GLU_EXTENSIONS**, then **gluGetString** returns a space-separated list of names of supported GLU extensions. (Extension names never contain spaces.)

All strings are null-terminated.

Parameters

name — Specifies a symbolic constant, one of **GLU_VERSION**, or **GLU_EXTENSIONS**.

Notes

The **gluGetString** subroutine only returns information about GLU extensions. Call **glGetString** to get a list of GL extensions.

The **gluGetString** subroutine is an initialization routine. Calling it after a **glNewList** results in undefined behavior.

Error Codes

`NULL` is returned if *name* is not **GLU_VERSION** or **GLU_EXTENSIONS**.

Related Information

The **glGetString** subroutine.

---

gluGetTessProperty

Purpose

Gets a tessellation object property.

Library

C bindings library: **libGL.a**

C Syntax

```
void gluGetTessProperty( GLUtesselator*  tess,
   GLenum  which,
   GLdouble*  data)
```

Description

The **gluGetTessProperty** subroutine retrieves properties stored in a tessellation object. These properties affect the way that tessellation objects are interpreted and rendered. See the **gluTessProperty** reference page for information about the properties and what they do.

Parameters

tess	Specifies the tessellation object (created with **gluNewTess**).
which	Specifies the property whose value is to be fetched. Valid values are: • **GLU_TESS_WINDING_RULE** • **GLU_TESS_BOUNDARY_ONLY** • **GLU_TESS_TOLERANCE**
data	Specifies a pointer to the location into which the value of the named property is written.

Related Information

The **gluNewTess** subroutine, **gluTessProperty** subroutine.

---

gluLoadSamplingMatrices Subroutine

Purpose

Loads non-uniform rational B-spline (NURBS) sampling and culling matrices.

Library

`OpenGL C bindings library:` **`libGL.a`**

C Syntax

```
void gluLoadSamplingMatrices(GLUnurbs*  nurb,
   const GLfloat  *model,
   const GLfloat  *perspective,
   const GLint  *view)
```

Description

The **gluLoadSamplingMatrices** subroutine uses the *model*, *perspective*, and *view* parameters to recompute the sampling and culling matrices stored in the *nurb* parameter. The sampling matrix determines how finely a NURBS surface or curve must be tessellated to satisfy the sampling tolerance (as determined by the **GLU_SAMPLING_TOLERANCE** property). The culling matrix determines whether a NURBS curve or surface should be culled before rendering (when the **GLU_CULLING** property is turned on).

Use of the **gluLoadSamplingMatrices** subroutine is necessary only if the **GLU_AUTO_LOAD_MATRIX** property is turned off. (See the **gluNurbsProperty** subroutine for information on adjusting properties in a NURBS object.) Leaving the **GLU_AUTO_LOAD_MATRIX** property turned on causes performance slowdown since it necessitates a round-trip to the OpenGL server to fetch the current values of the modelview matrix, projection matrix, and viewport.

Parameters

nurb	Specifies the NURBS object created with the **gluNewNurbsRenderer** subroutine.
model	Specifies a modelview matrix, such as from a **glGetFloatv** call.
perspective	Specifies a projection matrix, such as from a **glGetFloatv** call.
view	Specifies a viewport, such as from a **glGetIntegerv** call.

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **gluGetNurbsProperty** subroutine, **gluNewNurbsRenderer** subroutine, **gluNurbsProperty** subroutine.

gluLookAt Subroutine

Purpose

Defines a viewing transformation.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void gluLookAt(GLdouble  eyeX,
   GLdouble  eyeY,
   GLdouble  eyeZ,
   GLdouble  centerX,
   GLdouble  centerY,
   GLdouble  centerZ,
```

```
GLdouble  upX,
GLdouble  upY,
GLdouble  upZ)
```

Description

The gluLookAt subroutine multiplies the top matrix of the current matrix stack with a matrix M (computed below), whose effect is to place the eye point at the origin, the center point along the negative *z* axis, and the up vector somewhere in the YZ plane, above the z axis. This is done through pure rotation and translation, preserving all distance metrics.

The matrix M generated by the OpenGL could be computed as follows:

```
Let E be the 3d column vector (eyeX, eyeY, eyeZ).
Let C be the 3d column vector (centerX, centerY, centerZ).
Let U be the 3d column vector (upX, upY, upZ).
Compute L = C - E.
Normalize L.
Compute S = L x U.
Normalize S.
Compute U' = S x L.
```

M is the matrix whose columns are, in order:

```
(S, 0), (U', 0), (-L, 0), (-E, 1)  (all column vectors)
```

Note: This matrix is defined for use in systems where the the modelling coordinate vector is a column vector and is multiplied on the left by the matrices. If you prefer a row vector which gets multiplied by matrices to its right, then use the transpose of this matrix M.

Note: It is necessary that the UP vector NOT be parallel to the line connecting the center point with the eye point.

Parameters

eyeX, *eyeY*, *eyeZ*	Specifies the position of the eye point.
centerX, *centerY*, *centerZ*	Specifies the position of the reference point.
upX, *upY*, *upZ*	Specifies the direction of the up vector.

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glFrustum** subroutine, **gluPerspective** subroutine.

gluNewNurbsRenderer Subroutine

Purpose

Creates a non-uniform rational B-spline (NURBS) object.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLUnurbs* gluNewNurbsRenderer(void)
```

Description

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

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **gluBeginCurve** subroutine, **gluBeginSurface** subroutine, **gluBeginTrim** subroutine, **gluDeleteNurbsRenderer** subroutine, **gluNurbsCallback** subroutine, **gluNurbsProperty** subroutine.

gluNewQuadric Subroutine

Purpose

Creates a quadrics object.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLUquadric* gluNewQuadric( void )
```

Description

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

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **gluCylinder** subroutine, **gluDeleteQuadric** subroutine, **gluDisk** subroutine, **gluPartialDisk** subroutine, **gluQuadricCallback** subroutine, **gluQuadricDrawStyle** subroutine, **gluQuadricNormals** subroutine, **gluQuadricOrientation** subroutine, **gluQuadricTexture** subroutine, **gluSphere** subroutine.

gluNewTess Subroutine

Purpose

Creates a tessellation object.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLUtesselator* gluNewTess( void )
```

Description

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

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **gluBeginPolygon** subroutine, **gluDeleteTess** subroutine, **gluTessCallback** subroutine.

gluNextContour Subroutine

Purpose

Marks the beginning of another contour.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void gluNextContour(GLUtesselator*  tess,
         GLenum  type )
```

Description

The **gluNextContour** subroutine is used in describing polygons with multiple contours. After describing the first contour through a series of **gluTessVertex** calls, a **gluNextContour** call indicates that the previous contour is complete and the next contour is about to begin. Then, another series of **gluTessVertex** calls is used to describe the new contour. This process can be repeated until all contours are described.

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

The type of contour that follows the **gluNextContour** subroutine is determined by the *type* parameter. Acceptable contour types are as follows:

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

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

This command is obsolete and is provided for backward compatibility only. Calls to **gluNextContour** are mapped to **gluTessEndContour** followed by **gluTessBeginContour**.

Parameters

tess Specifies the tessellation object created with the **gluNewTess** subroutine.

type Specifies the contour type. Valid values are:

- **GLU_EXTERIOR**
- **GLU_INTERIOR**
- **GLU_UNKNOWN**
- **GLU_CCW**
- **GLU_CW**

Examples

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

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

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **gluBeginPolygon** subroutine, **gluNewTess** subroutine, **gluTessBeginContour** subroutine, **gluTessCallback** subroutine, **gluTessVertex** subroutine.

gluNurbsCallback Subroutine

Purpose

Defines a callback for a non-uniform rational B-spline (NURBS) object.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void gluNurbsCallback(GLUnurbs*  nurb,
   GLenum  which,
   GLvoid (* CallBackFunc)())
```

Description

The **gluNurbsCallback** subroutine is used to define a callback to be used by a NURBS object. If the specified callback is already defined, the existing definition is replaced. If the *CallBackFunc* parameter is null, then this callback will not get involked and the replaced data, if any, will be lost.

Except the error callback, these callbacks are used by NURBS tessellator (when **GLU_NURBS_MODE** is set to be **GLU_NURBS_TESSELLATOR**) to return back the OpenGL polygon primitives resulted from the tessellation. Note that there are two versions of each callback: one with a user data pointer and one without. If both versions for a particular callback are specified then the callback with the user data pointer will be used. Note that ″userData″ is a copy of the pointer that was specified at the last call to **gluNurbsCallbackData**.

The error callback function is effective no matter which value that **GLU_NURBS_MODE** is set to. All other callback functions are effective only when **GLU_NURBS_MODE** is set to **GLU_NURBS_TESSELLATOR**.

The legal callbacks are as follows:

GLU_NURBS_BEGIN
The begin callback indicates the start of a primitive. The function takes a single argument of type Glenum which can be one of **GL_LINES**, **GL_LINE_STRIPS**, **GL_TRIANGLE_FAN**, **GL_TRIANGLE_STRIP**, **GL_TRIANGLES**, or **GL_QUAD_STRIP**. The default begin callback function is null. The function prototype for this callback looks like:
void begin (GLenum type);

GLU_NURBS_BEGIN_DATA
The same as the **GLU_NURBS_BEGIN** callback except that it takes an additional pointer argument. This pointer is a copy of the pointer that was specified at the last call to **gluNurbsCallbackData**. The default callback function is null. The function prototype for this callback function looks like:
void beginData (GLenum type, void *userData);

GLU_NURBS_VERTEX
The vertex callback indicates a vertex of the primitive. The coordinates of the vertex are stored in the parameter ″vertex″. All the generated vertices have dimension 3, that is, Homogeneous coordinates have been transformed into affine coordinates. The default vertex callback function is null. The function prototype for this callback function looks like:
void vertex (GLfloat *vertex);

GLU_NURBS_VERTEX_DATA
The same as the **GLU_NURBS_VERTEX** callback except that it takes an additional pointer argument. This pointer is a copy of the pointer that was specified at the last call to **gluNurbsCallbackData**. The default callback function is null. The function prototype for this callback function looks like:
void vertexData (GLfloat *vertex, void *userData);

GLU_NURBS_NORMAL
The normal callback is invoked as the vertex normal is generated. The components of the normal are stored in the parameter ″normal″. In the case of a NURBS curve, the callback function is effective only when the user provides a normal map (**GL_MAP1_NORMAL**). In the case of a NURBS surface, if a normal map (**GL_MAP2_NORMAL**) is provided, then the generated normal is computed from the normal map. If a normal map is not provided then a surface normal is computed in a manner similar to that described for evaluators when **GL_AUTO_NORMAL** is enabled. The default normal callback function is null. The function prototype for this callback function looks like:
void normal (GLfloat *normal);

GLU_NURBS_NORMAL_DATA
The same as the **GLU_NURBS_NORMAL** callback except that it takes an additional pointer argument. This pointer is a copy of the pointer that was specified at the last call to **gluNurbsCallbackData**. The default callback function is null. The function prototype for this callback function looks like:
void normalData (GLfloat *normal, void *userData);

GLU_NURBS_COLOR
The color callback is invoked as the color of a vertex is generated. The components of the color are stored in the parameter ″color″. This callback is effective only when the user provides a color map (**GL_MAP1_COLOR_4** or **GL_MAP2_COLOR_4**). ″color″ contains four components: R,G,B,A. The default color callback function is null. The prototype for this callback function looks like:
void color (GLfloat *color);

GLU_NURBS_COLOR_DATA
The same as the **GLU_NURBS_COLOR** callback except that it takes an additional pointer argument. This pointer is a copy of the pointer that was specified at the last call to **gluNurbsCallbackData**. The default callback function is null. The function prototype for this callback function looks like:
void colorData (GLfloat *color, void *userData);

GLU_NURBS_TEX_COORD
The texture callback is invoked as the texture coordinates of a vertex are generated. These coordinates are stored in the parameter ″texCoord″. The number of texture coordinates can be 1, 2, 3, or 4 depending on which type of texture map is specified (**GL_MAP*_TEXTURE_COORD_1**, **GL_MAP*_TEXTURE_COORD_2**, **GL_MAP*_TEXTURE_COORD_3**, **GL_MAP*_TEXTURE_COORD_4** where * can be either 1 or 2). If no texture map is specified, this callback function will not be called. The default texture callback function is null. The function prototype for this callback function looks like:
void texCoord (GLfloat *texCoord);

GLU_NURBS_TEXTURE_COORD_DATA
The same as the **GLU_NURBS_TEX_COORD** callback except that it takes an additional pointer argument. This pointer is a copy of the pointer that was specified at the last call to **gluNurbsCallbackData**. The default callback function is null. The function prototype for this callback function looks like:
void texCoordData (GLfloat *texCoord, void *userData);

GLU_NURBS_END	The end callback is invoked at the end of a primitive. The default end callback function is null. The function prototype for this callback function looks like: void end (void);
GLU_NURBS_END_DATA	The same as the **GLU_NURBS_TEX_COORD** callback except that it takes an additional pointer argument. This pointer is a copy of the pointer that was specified at the last call to **gluNurbsCallbackData**. The default callback function is null. The function prototype for this callback function looks like: void endData (void *userData);
GLU_NURBS_ERROR	The error function is called when an error is encountered. Its single argument is of type GLenum, and it indicates the specific error that occurred. There are 37 errors unique to NURBS named **GLU_NURBS_ERROR1** through **GLU_NURBS_ERROR37**. Character strings describing these errors can be retrieved with **gluErrorString**.

Parameters

nurb	Specifies the NURBS object created with the **gluNewNurbsRenderer** subroutine.
which	Specifies the callback being defined. Valid values are: • **GLU_NURBS_BEGIN** • **GLU_NURBS_VERTEX** • **GLU_NURBS_NORMAL** • **GLU_NURBS_COLOR** • **GLU_NURBS_TEX_COORD** • **GLU_NURBS_END** • **GLU_NURBS_BEGIN_DATA** • **GLU_NURBS_VERTEX_DATA** • **GLU_NURBS_NORMAL_DATA** • **GLU_NURBS_COLOR_DATA** • **GLU_NURBS_TEXTURE_COORD_DATA** • **GLU_NURBS_END_DATA** • **GLU_NURBS_ERROR**
CallBackFunc	Specifies the function that the callback calls.

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **gluErrorString** subroutine, **gluNewNurbsRenderer** subroutine **gluNurbsCallbackData** subroutine **gluNurbsProperty** subroutine .

gluNurbsCallbackData Subroutine

Purpose

Sets a user data pointer.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void gluNurbsCallbackData( GLUnurbs* nurb,
   GLvoid* userData )
```

Description

The same as the GLU_NURBS_END callback, the gluNurbsCallbackData is used to pass a pointer to the application's data to NURBS tessellator. A copy of this pointer will be passed by the tessellator in the NURBS callback functions (set by **gluNurbsCallback**).

Parameters

nurb	Specifies the NURBS object (created with **gluNewNurbsRenderer**).
userData	Specifies a pointer to the user's data.

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **gluNewNurbsRenderer**subroutine and **gluNurbsCallback** subroutine.

---

gluNurbsCallbackDataEXT Subroutine

Purpose

Sets a user data pointer.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void gluNurbsCallbackDataEXT( GLUnurbs* nurb,
   GLvoid* userData )
```

Description

gluNurbsCallbackDataEXT is used to pass a pointer to the application's data to NURBS tessellator. A copy of this pointer will be passed by the tessellator in the NURBS callback functions (set by **gluNurbsCallback**).

Parameters

nurb	Specifies the NURBS object (created with **gluNewNurbsRenderer**).
userData	Specifies a pointer to the user's data.

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **gluNurbsCallback** subroutine.

gluNurbsCurve Subroutine

Purpose

Defines the shape of a non-uniform rational B-spline (NURBS) curve.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void gluNurbsCurve(GLUnurbs*  nurb,
   GLint  knotCount,
   GLfloat * knots,
   GLint  stride,
   GLfloat * control,
   GLint  order,
   GLenum  type)
```

Description

Use the **gluNurbsCurve** subroutine to describe a NURBS curve. When this subroutine is displayed between a **gluBeginCurve** and **gluEndCurve** pair, it describes a curve to be rendered. Positional, texture, and color coordinates are established by presenting each as a separate **gluNurbsCurve** statement between **gluBeginCurve** and **gluEndCurve** pairs. No more than one call to **gluNurbsCurve** for each color, position, or texture data can be made within a single **gluBeginCurve** and **gluEndCurve** pair. Exactly one call must be made to describe the position of the curve (a type of **GL_MAP1_VERTEX_3** or **GL_MAP1_VERTEX_4** description).

When a **gluNurbsCurve** subroutine is displayed between a **gluBeginTrim** and **gluEndTrim** pair, it describes a trimming curve on a NURBS surface. If the *Type* parameter is **GLU_MAP1_TRIM_2**, it describes a curve in 2-dimensional (2D) (*u* and *v*) parameter space. If the *type* parameter is **GLU_MAP1_TRIM_3**, it describes a curve in 2D homogeneous (*u*, *v*, and *w*) parameter space. (See the **gluBeginTrim** subroutine for more information about trimming curves.)

Parameters

nurb	Specifies the NURBS object created with the **gluNewNurbsRenderer** subroutine.
knotCount	Specifies the number of knots defined in the *knot* parameter. *knotCount* should equal the number of control points plus the order.
knots	Specifies an array of nondecreasing knot values. The length of this array is contained in the *knotCount* parameter.
stride	Defines the offset (as a number of single-precision floating-point values) between successive curve control points.
control	Specifies a pointer to an array of control points. These coordinates must agree with the *type* parameter specified below.

order	Specifies the order of the NURBS curve. The *order* parameter equals degree + 1, meaning that a cubic curve has an order of 4.
type	Indicates the type of the curve. If the curve is defined within a **gluBeginCurve**/**gluEndCurve** pair, the type may be any of the valid 1-dimensional evaluator type (such as **GL_MAP1_VERTEX_3** or **GL_MAP1_COLOR_4**). If it is between a **gluBeginTrim**/**gluEndTrim** pair, the only valid types are **GLU_MAP1_TRIM_2** or **GLU_MAP1_TRIM_3**.

Examples

The following commands render a textured NURBS curve with normals:

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

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Note: To define trim curves which stitch well, use **gluPwlCurve**.

Related Information

The **gluBeginCurve** subroutine, **gluBeginTrim** subroutine, **gluNewNurbsRenderer** subroutine, **gluPwlCurve** subroutine.

gluNurbsProperty Subroutine

Purpose

Sets a non-uniform rational B-spline (NURBS) property.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void gluNurbsProperty(GLUnurbs* nurb,
   GLenum property,
   GLfloat value)
```

Description

The **gluNurbsProperty** subroutine is used to control properties stored in a NURBS object. These properties affect the way that a NURBS curves is rendered. The following values are valid for the *property* parameter:

GLU_NURBS_MODE
value should be set to be either **GLU_NURBS_RENDERER** or **GLU_NURBS_TESSELLATOR**. When set to **GLU_NURBS_RENDERER**, NURBS objects are tessellated into OpenGL primitives and sent to the pipeline for rendering. When set to **GLU_NURBS_TESSELLATOR**, NURBS objects are tessellated into OpenGL primitives but the vertices, normals, colors, and/or textures are retrieved back through a callback interface (see **gluNurbsCallback**). This allows the user to cache the tessellated results for further processing.

GLU_SAMPLING_METHOD *Value*
Specifies how a NURBS surface should be tessellated. The *value* parameter may be one of **GLU_PATH_LENGTH**, **GLU_PARAMETRIC_ERROR**, or **GLU_DOMAIN_DISTANCE**, **GLU_OBJECT_PATH_LENGTH**, or **GLU_OBJECT_PARAMETRIC_ERROR**. When set to **GLU_PATH_LENGTH**, the surface is rendered so that the maximum length, in pixels, of the edges of the tessellation polygons is no greater than what is specified by **GLU_SAMPLING_TOLERANCE**.

GLU_PARAMETRIC_ERROR specifies that the surface is rendered in such a way that the *value* specified by **GLU_PARAMETRIC_TOLERANCE** describes the maximum distance, in pixels, between the tessellation polygons and the surfaces they approximate.

GLU_DOMAIN_DISTANCE allows users to specify, in parametric coordinates, how many sample points per unit length are taken in *u*, *v* direction.

GLU_OBJECT_PATH_LENGTH is similar to **GLU_PATH_LENGTH** except that it is view independent, that is, the surface is rendered so that the maximum length, in object space, of edges of the tessellation polygons is no greater than what is specified by **GLU_SAMPLING_TOLERANCE**.

GLU_OBJECT_PARAMETRIC_ERROR is similar to **GLU_PARAMETRIC_ERROR** except that it is view independent, that is, the surface is rendered in such a way that the value specified by **GLU_PARAMETRIC_TOLERANCE** describes the maximum distance, in object space, between the tessellation polygons and the surfaces they approximate.

The initial value of **GLU_SAMPLING_METHOD** is **GLU_PATH_LENGTH**.

GLU_SAMPLING_TOLERANCE *Value*
Specifies the maximum length, in pixels or in object space length unit, to use when the sampling methos is set to **GLU_PATH_LENGTH** OR **GLU_OBJECT_PATH_LENGTH**. The NURBS code is conservative when rendering a curve or surface, so the actual length can be somewhat shorter. The default value is 50.0 pixels.

GLU_PARAMETRIC_TOLERANCE *Value*
The *Value* parameter specifies the maximum distance, in pixels, to use when the sampling method is **GLU_PARAMETRIC_ERROR**. The initial value is 0.5.

GLU_U_STEP *Value*
: The *Value* parameter specifies the number of sample points per unit length taken along the *u* axis in parametric coordinates. It is needed when **GLU_SAMPLING_METHOD** is set to **GLU_DOMAIN_DISTANCE**. The initial value is 100.

GLU_V_STEP *Value*
: The *Value* parameter specifies the number of sample points per unit length taken along the *v* axis in parametric coordinates. It is needed when **GLU_SAMPLING_METHOD** is set to **GLU_DOMAIN_DISTANCE**. The initial value is 100

GLU_DISPLAY_MODE *Value*
: The *Value* parameter defines how a NURBS surface should be rendered. The *Value* parameter can be set to the following:

- **GLU_FILL**
- **GLU_OUTLINE_POLYGON**
- **GLU_OUTLINE_PATCH**

Only one of these values can be used. **GLU_FILL** causes the surface to be rendered as a set of polygons. **GLU_OUTLINE_POLYGON** instructs the NURBS library to draw only the outlines of the polygons created by tessellation. **GLU_OUTLINE_PATCH** causes just the outlines of patches and trim curves defined by the user to be drawn.

When **GLU_NURBS_MODE** is set to be **GLU_NURBS_TESSELLATOR**, *value* defines how a NURBS surface should be tessellated. When **GLU_DISPLAY_MODE** is set to **GLU_FILL** or **GLU_OUTLINE_POLY**, the NURBS surface is tessellated into OpenGL triangle primitives which can be retrieved back through callback functions. If **GLU_DISPLAY_MODE** is set to **GLU_OUTLINE_PATCH**, only the outlines of the patches and trim curves are generated as a sequence of line strips which can be retrieved back through callback functions.

The default value is **GLU_FILL**.

GLU_CULLING *Value*
: The *Value* parameter is a boolean value that, when set to **GL_TRUE**, indicates that a NURBS curve should be discarded prior to tessellation if its control points lie outside the current viewport. The default is **GL_FALSE**.

GLU_AUTO_LOAD_MATRIX *Value*

The *Value* parameter is a Boolean value that, when set to **GL_TRUE**, causes the NURBS code to download the projection matrix, the modelview matrix, and the viewport from the OpenGL server to compute sampling and culling matrices for each NURBS curve that is rendered. Sampling and culling matrices are required to determine the tessellation of a NURBS surface into line segments or polygons and to cull a NURBS surface if it lies outside of the viewport.

If this mode is set to **GL_FALSE**, the user must provide a projection matrix, a modelview matrix, and a viewport for the NURBS renderer to use to construct sampling and culling matrices. This can be done with the **gluLoadSamplingMatrices** subroutine. The default for this mode is **GL_TRUE**. Changing this mode from **GL_TRUE** to **GL_FALSE** does not affect the sampling and culling matrices until **gluLoadSamplingMatrices** is called.

Parameters

nobj Specifies the NURBS object created with the **gluNewNurbsRenderer** subroutine.

Property Specifies the name of the property to be set. The following values are valid:

- **GLU_SAMPLING_TOLERANCE**
- **GLU_DISPLAY_MODE**
- **GLU_CULLING**
- **GLU_AUTO_LOAD_MATRIX**
- **GLU_PARAMETRIC_TOLERANCE**
- **GLU_SAMPLING_METHOD**
- **GLU_U_STEP**
- **GLU_V_STEP**

Value Specifies the value to which the indicated property is set. *Value* may be a numeric value or one of the following:

- **GLU_PATH_LENGTH**
- **GLU_PARAMETRIC_ERROR**
- **GLU_DOMAIN_DISTANCE**

Notes

If **GLU_AUTO_LOAD_MATRIX** is true, sampling and culling may be executed incorrectly if NURBS routines are compiled into a display list.

A *property* of **GLU_PARAMETRIC_TOLERANCE**, **GLU_SAMPLING_METHOD**, **GLU_U_STEP**, or **GLU_V_STEP**, or a *value* of **GLU_PATH_LENGTH**, **GLU_PARAMETRIC_ERROR**, **GLU_DOMAIN_DISTANCE** are only available if the GLU version is 1.1 or greater. They are not valid parameters in GLU 1.0.

Use the **gluGetString** subroutine to determine the GLU version.

Files

/usr/include/GL/gl.h Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **gluGetNurbsProperty** subroutine, **gluLoadSamplingMatrices** subroutine, **gluNewNurbsRenderer** subroutine, **gluGetString** subroutine, **gluNurbsCallback** subroutine .

gluNurbsSurface Subroutine

Purpose

Defines the shape of a non-uniform rational B-spline (NURBS) surface.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void gluNurbsSurface(GLUnurbs*  nurb,
   GLint  sKnotCount,
   GLfloat * sKnots,
   GLint  tKnotCount,
   GLfloat * tKnots,
   GLint  sStride,
   GLint  tStride,
   GLfloat * control,
   GLint  sOrder,
   GLint  tOrder,
   GLenum  type)
```

Description

The **gluNurbsSurface** subroutine is used within a NURBS surface definition to describe the shape of a NURBS surface before trimming. To mark the beginning and end of a NURBS surface definition, use the **gluBeginSurface** and **gluEndSurface** commands.

Note: Call the **gluNurbsSurface** subroutine within a NURBS surface definition only.

Positional, texture, and color coordinates are associated with a surface by presenting each as a separate **gluNurbsSurface** statement between a **gluBeginSurface** and **gluEndSurface** pair. Each **gluBeginSurface** and **gluEndSurface** pair can contain no more than one call to **gluNurbsSurface** for each color, position, and texture data. One (and only one) call must be made to describe the position of the surface. (The *Type* parameter must be either **GL_MAP2_VERTEX_3** or **GL_MAP2_VERTEX_4**.)

A NURBS surface can be trimmed using the **gluNurbsCurve** and **gluPwlCurve** subroutines within calls to **gluBeginTrim** and **gluEndTrim**.

Note: A **gluNurbsSurface** with *sKnotCount* knots in the *u* direction and *tKnotCount* knots in the *v* direction with the *sOrder* and *tOrder* orders must have control points equal to (*sKnotCount* - *sOrder*) x (*tKnotCount* - *tOrder*).

Parameters

nurb	Specifies the NURBS object created with the **gluNewNurbsRenderer** subroutine.
sKnotCount	Specifies the number of knots in the parametric *U* direction.
sKnots	Specifies an array of non-decreasing *sKnotCount* values in the parametric *U* direction
tKnotCount	Specifies the number of knots in the parametric *V* direction.
tKnots	Specifies an array of non-decreasing *tKnotCount* values in the parametric *V* direction.

sStride	Specifies the offset (as a number of single-precision floating-point values) between successive control points in the parametric *U* direction in *control*.
tStride	Specifies the offset (in single-precision floating-point values) between successive control points in the parametric *V* direction in *control*.
control	Specifies an array containing control points for the NURBS surface. The offsets between successive control points in the parametric *u* and *v* directions are given by *sStride* and *tStride*.
sOrder	Specifies the order of the NURBS surface in the parametric *u* direction. The order is one more than the degree; therefore, a surface that is cubic in *u* has a *u* order of 4.
tOrder	Specifies the order of the NURBS surface in the parametric *v* direction. The order is one more than the degree; therefore, a surface that is cubic in *v* has a *v* order of 4.
type	Specifies the surface type. This value must be one of the valid 2-dimensional evaluators (such as **GL_MAP2_VERTEX_3** or **GL_MAP2_COLOR_4**).

Examples

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

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

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **gluBeginSurface** subroutine, **gluBeginTrim** subroutine, **gluNewNurbsRenderer** subroutine, **gluNurbsCurve** subroutine, **gluPwlCurve** subroutine.

gluOrtho2D Subroutine

Purpose

Defines a 2-dimensional (2D) orthographic projection matrix.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

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

Description

The **gluOrtho2D** subroutine sets up a 2D orthographic viewing region. Use of this subroutine is equivalent to calling **glOrtho** with values of *Near* = -1 and *Far* = 1.

Parameters

left	Specifies the coordinates for the left vertical clipping planes.
right	Specifies the coordinates for the right vertical clipping planes.
bottom	Specifies the coordinates for the bottom horizontal clipping planes.
top	Specifies the coordinates for the top horizontal clipping planes.

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glOrtho** subroutine, **gluPerspective** subroutine.

---

gluPartialDisk Subroutine

Purpose

Draws an arc of a disk.

Library

`OpenGL C bindings library: libGL.a`

C Syntax

```
void gluPartialDisk(GLUquadricObj * quad,
   GLdouble  inner,
   GLdouble  outer,
   GLint  slices,
   GLint  loops,
   GLdouble  start,
   GLdouble  sweep)
```

Description

The **gluPartialDisk** subroutine renders a partial disk on the z=0 (zero) plane. A partial disk is similar to a full disk, except that only a subset of the disk consisting of *start* through *start* plus *sweep* is included. For the purposes of this subroutine, 0 degrees is along the +y axis, 90 degrees is along the +x axis, 180 degrees is along the -y axis, and 270 degrees is along the -x axis.

The partial disk has a radius defined by the *outer* parameter and contains a concentric circular hole with a radius defined by the *inner* parameter. If the value of *InnerRadius* is 0, no hole is generated. The partial disk is subdivided around the z axis into slices and rings (as specified by the *slices* and *loops* parameters, respectively).

With regard to orientation, the +z side of the partial disk is considered to be *outside*. (See the **gluQuadricOrientation** subroutine for details on specifying quadrics orientation.) This means that if the orientation is set to **GLU_OUTSIDE**, any normals generated point along the +z axis. Otherwise, they point along the -z axis.

If texturing is turned on with the **gluQuadricTexture** subroutine, texture coordinates are generated linearly such that the value at (*r*, 0, 0) (where *r*=*outer*) is (1, 0.5).

Under the same definition, the following values also apply:

Coordinates	Value
(0, *r*, 0)	(0.5, 1)
(-*r*, 0, 0)	(0, 0.5)
(0, -*r*, 0)	(0.5, 0)

Parameters

quad	Specifies a quadrics object created with the **gluNewQuadric** subroutine.
inner	Specifies the inner radius of the partial disk. (This value can be 0.)
outer	Specifies the outer radius of the partial disk.
slices	Specifies the number of desired subdivisions around the z axis.
loops	Specifies the number of concentric rings around the origin into which the partial disk is subdivided.
start	Specifies the start angle (in degrees) of the disk portion.
sweep	Specifies the sweep angle (in degrees) of the disk portion.

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **gluCylinder** subroutine, **gluDisk** subroutine, **gluNewQuadric** subroutine, **gluQuadricOrientation** subroutine, **gluQuadricTexture** subroutine, **gluSphere** subroutine.

---

gluPerspective Subroutine

Purpose

Sets up a perspective projection matrix.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

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

Description

The **gluPerspective** subroutine specifies a viewing frustum into the world coordinate system. Generally, the *aspect* used with this subroutine should match that of its associated viewport. For example, an aspect ratio value of *aspect*=2.0 means the viewer's angle of view is twice as wide in *x* as it is in *y*. If the viewport is twice as wide as it is tall, it displays the image without distortion.

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

Given *f* defined as follows:

f = cotangent(fovy/2)

The generated matrix is

$$\begin{pmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{zFar+zNear}{zNear-zFar} & \frac{2*zFar*zNear}{zNear-zFar} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Parameters

fovy	Specifies the field of view angle (in degrees) in the y direction.
aspect	Indicates the aspect ratio. This value determines the field of view in the *x* direction and is the ratio of *x* (width) to *y* (height).
zNear	Specifies the distance from the viewer to the closest clipping plane. This value must be positive.
zFar	Specifies the distance from the viewer to the farthest clipping plane. This value must be positive.

Notes

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

r = zFar/zNear

roughly log2(*r*) bits of depth buffer precision are lost. Because *r* approaches infinity as *zNear* approaches 0, *zNear* must never be set to 0.

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glFrustum** subroutine, **glLoadIdentity** subroutine, **glMultMatrix** subroutine, **gluOrtho2D** subroutine.

gluPickMatrix Subroutine

Purpose

Defines a picking region.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void gluPickMatrix( GLdouble  x,
   GLdouble  y,
   GLdouble  delX,
   GLdouble  delY,
   GLint  viewport[4])
```

Description

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

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

When using **gluPickMatrix** to pick non-uniform rational B-splines (NURBS), be careful to turn off the NURBS **GLU_AUTO_LOAD_MATRIX** property. If **GLU_AUTO_LOAD_MATRIX** is not turned off, any NURBS surface rendered is subdivided differently with the pick matrix than the way it was subdividied without the pick matrix.

Parameters

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

Examples

When rendering a scene as in the following example:

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

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

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

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glGet** subroutine, **glLoadIdentity** subroutine, **glMultMatrix** subroutine, **glRenderMode** subroutine, **gluPerspective** subroutine.

---

gluProject Subroutine

Purpose

Maps object coordinates to window coordinates.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
int gluProject(GLdouble  objX,
   GLdouble  objY,
   GLdouble  objZ,
   const GLdouble  model,
   const GLdouble  proj,
   const GLint  view,
   GLdouble * winX,
   GLdouble * winY,
   GLdouble * winZ )
```

Description

The **gluProject** transforms the specified object space coordinates into window coordinates using the *model*, *proj*, and *view* values provided. The results are stored in *winX*, *winY*, and *winZ*. A return value of **GL_TRUE** indicates success, and **GL_FALSE** indicates failure.

To compute the coordinates, let v=(objX,objY,objZ,1.0) represented as a matrix with 4 rows and 1 column. Then **gluProject** computes v' as follows:

v' = P x M x v

where P is the current projection matrix *proj*, M is the current modelview matrix *model* (both represented as 4x4 matrices in column-major order) and 'x' represents matrix multiplication.

The window coordinates are then computed as follows:

```
winX = view(0) + view(2) * (v'(0) + 1) / 2

winY = view(1) + view(3) * (v'(1) + 1) / 2

winZ = (v'(2) + 1) / 2
```

Parameters

objX, *objY*, *objZ*	Specify the object coordinates.
model	Specifies the current modelview matrix (as from a **glGetDoublev** call).
proj	Specifies the current projection matrix (as from a **glGetDoublev** call).
view	Specifies the current viewport (as from a **glGetIntegerv** call).
winX, *winY*, *winZ*	Returns the computed window coordinates.

Return Values

GL_TRUE	Indicates the conversion succeeded.
GL_FALSE	Indicates the conversion failed.

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glGet** subroutine, **gluUnProject** subroutine.

---

gluPwlCurve Subroutine

Purpose

Defines a piecewise linear non-uniform rational B-spline (NURBS) trimming curve.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void gluPwlCurve(GLUnurbs *  nurb,
   GLint  count,
   GLfloat*  data,
   GLint  stride,
   GLenum  type)
```

Description

The **gluPwlCurve** subroutine describes a piecewise linear trimming curve for a NURBS surface. A piecewise linear curve consists of a list of coordinates of points in the parameter space for the NURBS surface to be trimmed. These points are connected with line segments to form a curve. If the curve is an approximation of a curve that is not piecewise linear, the points should be close enough in parameter space that the resulting path appears curved at the resolution used in the application.

A value of **GLU_MAP1_TRIM_2** assigned for the *type* parameter describes a curve in 2-dimensional (2D) (*u* and *v*) parameter space; **GLU_MAP1_TRIM_3** describes a curve in 2D homogeneous (*u*, *v*, and *w*) parameter space. (See the **gluBeginTrim** subroutine for more information on trimming curves.)

> **Note:** to describe a trim curve that closely follows the contours of a NURBS surface, call **gluNurbsCurb**.

Parameters

nurb	Specifies the NURBS object created with the **gluNewNurbsRenderer** subroutine.
count	Specifies the number of points on the curve.
data	Specifies an array containing the curve points.
stride	Specifies the offset (a number of single-precision floating-point values) between points on the curve.
type	Specifies the curve type. The valid types are **GLU_MAP1_TRIM_2** and **GLU_MAP1_TRIM_3**.

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **gluBeginCurve** subroutine, **gluBeginTrim** subroutine, **gluNewNurbsRenderer** subroutine, **gluNurbsCurve** subroutine.

gluQuadricCallback Subroutine

Purpose

Defines a callback for a quadrics object.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void gluQuadricCallback( GLUquadric* quad,
   GLenum  which,
   void * CallBackFunc( ) )
```

Description

The **gluQuadricCallback** subroutine defines a new callback for use by a quadrics object. If the specified callback is already defined, the existing definition for that callback is replaced. If the *CallBackFunc* parameter is set to null, any existing callback is erased.

GLU_ERROR is the only legal callback for this subroutine. The function is called when an error is encountered. Its only argument is of type **GLenum**, which indicates the specific error. Character strings describing these errors can be retrieved with the **gluErrorString** call.

Parameters

quad	Specifies the quadrics object created with the **gluNewQuadric** subroutine.
which	Specifies the callback being defined. The only valid value for this parameter is **GLU_ERROR**.
CallBackFunc	Specifies the function being called.

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **gluErrorString** subroutine, **gluNewQuadric** subroutine.

gluQuadricDrawStyle Subroutine

Purpose

Specifies the desired quadric drawing style.

Library

`OpenGL C bindings library: libGL.a`

C Syntax

```
void gluQuadricDrawStyle( GLUquadric*  qobj,
   GLenum  draw )
```

Description

The **gluQuadricDrawStyle** subroutine specifies the draw style for quadrics rendered with the *quad* parameter. Legal values are as follows:

GLU_FILL	Quadrics are rendered with polygon primitives. The polygons are drawn counterclockwise of their normals (as defined with the **gluQuadricOrientation** subroutine).
GLU_LINE	Quadrics are rendered as a set of lines.
GLU_SILHOUETTE	Quadrics are rendered as a set of lines; however, edges separating coplanar faces are not drawn.
GLU_POINT	Quadrics are rendered as a set of points.

Parameters

quad Specifies the quadrics object created with the **gluNewQuadric** subroutine.

draw Specifies the desired draw style. Valid values are as follows:

- **GLU_FILL**
- **GLU_LINE**
- **GLU_SILHOUETTE**
- **GLU_POINT**

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **gluNewQuadric** subroutine, **gluQuadricNormals** subroutine, **gluQuadricOrientation** subroutine, **gluQuadricTexture** subroutine.

---

gluQuadricNormals Subroutine

Purpose

Specifies the desired normals for quadrics.

Library

`OpenGL C bindings library: libGL.a`

C Syntax

```
void gluQuadricNormals( GLUquadric*  quad,
   GLenum  normal)
```

Description

The **gluQuadricNormals** subroutine specifies the desired normals for quadric objects rendered with the *quad* parameter option. Legal parameter values are as follows:

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

Parameters

quad Specifies the quadrics object created with the **gluNewQuadric** subroutine.

normal Specifies the desired type of normals. Valid values are as follows:

- **GLU_NONE**
- **GLU_FLAT**
- **GLU_SMOOTH**

Files

/usr/include/GL/gl.h Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **gluNewQuadric** subroutine, **gluQuadricDrawStyle** subroutine, **gluQuadricOrientation** subroutine, **gluQuadricTexture** subroutine.

gluQuadricOrientation Subroutine

Purpose

Specifies inside and outside orientation for quadrics.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void gluQuadricOrientation( GLUquadric*  quad,
   GLenum  orientation)
```

Description

The **gluQuadricOrientation** subroutine specifies the desired orientation for quadrics rendered with the *quad* parameter option. Acceptable values for the *orientation* parameter are as follows:

GLU_OUTSIDE	Quadrics are drawn with normals pointing outward. This value is the default.
GLU_INSIDE	Quadrics are drawn with normals pointing inward. The default value is **GLU_OUTSIDE**.

Note: Outward and inward orientations are defined relative to the quadric being drawn.

Parameters

quad	Specifies the quadrics object created with the **gluNewQuadric** subroutine.
orientation	Specifies the desired orientation. Valid values are **GLU_OUTSIDE** and **GLU_INSIDE**.

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **gluNewQuadric** subroutine, **gluQuadricDrawStyle** subroutine, **gluQuadricNormals** subroutine, **gluQuadricTexture** subroutine.

---

gluQuadricTexture Subroutine

Purpose

Specifies if texturing is desired for quadrics.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void gluQuadricTexture( GLUquadric* quad,
   GLboolean  texture )
```

Description

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

> **Note:** The manner in which texture coordinates are generated depends upon the specific quadric rendered.

Parameters

quad	Specifies the quadrics object created with the **gluNewQuadric** subroutine.
texture	Specifies a flag indicating if texture coordinates should be generated. Valid values are **GL_TRUE** and **GL_FALSE**.

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **gluNewQuadric** subroutine, **gluQuadricDrawStyle** subroutine, **gluQuadricNormals** subroutine, **gluQuadricOrientation** subroutine.

gluScaleImage Subroutine

Purpose

Scales an image to an arbitrary size.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLint gluScaleImage( GLenum  format,
   GLsizei  wIn,
   GLsizei  hIn,
   GLenum  typeIn,
   const void * dataIn,
   GLsizei  wOut,
   GLsizei  hOut,
   GLenum  typeOut,
   void*  dataOut)
```

Description

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

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

A return value of zero indicates success; otherwise, a GLU error code is returned. (See the **gluErrorString** subroutine for standard GLU error codes.)

See the **glReadPixels** subroutine for a description of valid values for the *format*, *typeIn*, and *typeOut* parameters.

Parameters

format Specifies the format of the pixel data. The following symbolic values are valid:

- **GL_COLOR_INDEX**
- **GL_STENCIL_INDEX**
- **GL_DEPTH_COMPONENT**
- **GL_RED**
- **GL_GREEN**
- **GL_BLUE**
- **GL_ALPHA**
- **GL_RGB**
- **GL_RGBA**
- **GL_BGRA**
- **GL_LUMINANCE**
- **GL_LUMINANCE_ALPHA**

wIn, *hIn* Specify the width and height, respectively, of the source image that is scaled. Values are listed in map pixels.

typeIn Specifies the data type for the *dataIn* parameter. The following symbolic values are valid:

- **GL_UNSIGNED_BYTE**
- **GL_BYTE**
- **GL_BITMAP**
- **GL_UNSIGNED_SHORT**
- **GL_SHORT**
- **GL_UNSIGNED_INT**
- **GL_INT**
- **GL_FLOAT**
- **GL_UNSIGNED_BYTE_3_3_2**
- **GL_UNSIGNED_BYTE_2_3_3_REV**
- **GL_UNSIGNED_SHORT_5_6_5**
- **GL_UNSIGNED_SHORT_5_6_5_REV**
- **GL_UNSIGNED_SHORT_4_4_4_4**
- **GL_UNSIGNED_SHORT_4_4_4_4_REV**
- **GL_UNSIGNED_SHORT_5_5_5_1**
- **GL_UNSIGNED_SHORT_1_5_5_5_REV**
- **GL_UNSIGNED_INT_8_8_8_8**
- **GL_UNSIGNED_INT_8_8_8_8_REV**
- **GL_UNSIGNED_INT_10_10_10_2**
- **GL_UNSIGNED_INT_2_10_10_10_REV**

dataIn Specifies a pointer to the source image.

wOut, *hOut* Specify the width and height, respectively, in pixels, of the destination image.

typeOut Specifies the data type for the *dataOut* parameter. The following symbolic values are valid:

- **GL_UNSIGNED_BYTE**
- **GL_BYTE**
- **GL_BITMAP**
- **GL_UNSIGNED_SHORT**
- **GL_SHORT**
- **GL_UNSIGNED_INT**
- **GL_INT**
- **GL_FLOAT**
- **GL_UNSIGNED_BYTE_3_3_2**
- **GL_UNSIGNED_BYTE_2_3_3_REV**
- **GL_UNSIGNED_SHORT_5_6_5**
- **GL_UNSIGNED_SHORT_5_6_5_REV**
- **GL_UNSIGNED_SHORT_4_4_4_4**
- **GL_UNSIGNED_SHORT_4_4_4_4_REV**
- **GL_UNSIGNED_SHORT_5_5_5_1**
- **GL_UNSIGNED_SHORT_1_5_5_5_REV**
- **GL_UNSIGNED_INT_8_8_8_8**
- **GL_UNSIGNED_INT_8_8_8_8_REV**
- **GL_UNSIGNED_INT_10_10_10_2**
- **GL_UNSIGNED_INT_2_10_10_10_REV**

dataOut Specifies a pointer to the destination image.

See the **glReadPixels** subroutine for a description of valid values for the *Format*, *TypeIn*, and *TypeOut* parameters.

Return Values

0 Indicates the scaling succeeded. If the subroutine fails, a GLU error code is returned. (See the **gluErrorString** subroutine for standard GLU error codes.)

Error Codes

- **GLU_INVALID_VALUE** is returned if *wIn*, *hIn*, *wOut* or *hOut* is Negative.
- **GLU_INVALID_ENUM** is returned if *format*, *typeIn* or *typeOut* are not legal.
- **GLU_INVALID_OPERATION** is returned if *typeIn* or *typeOut* is **GL_UNSIGNED_BYTE_3_3_2** or **GL_UNSIGNED_BYTE_2_3_3_REV** and *format* is not **GL_RGB**.
- **GLU_INVALID_OPERATION** is returned if *typeIn* or *typeOut* is **GL_UNSIGNED_SHORT_5_6_5** or **GL_UNSIGNED_SHORT_5_6_5_REV** and *format* is not **GL_RGB**.
- **GLU_INVALID_OPERATION** is returned if *typeIn* or *typeOut* is **GL_UNSIGNED_SHORT_4_4_4_4** or **GL_UNSIGNED_SHORT_4_4_4_4_REV** and *format* is neither **GL_RGBA** nor **GL_BGRA**.
- **GLU_INVALID_OPERATION** is returned if *typeIn* or *typeOut* is **GL_UNSIGNED_SHORT_5_5_5_1** or **GL_UNSIGNED_SHORT_1_5_5_5_REV** and *format* is neither **GL_RGBA** nor **GL_BGRA**.
- **GLU_INVALID_OPERATION** is returned if *typeIn* or *typeOut* is **GL_UNSIGNED_INT_8_8_8_8** or **GL_UNSIGNED_INT_8_8_8_8_REV** and *format* is neither **GL_RGBA** nor **GL_BGRA**.
- **GLU_INVALID_OPERATION** is returned if *typeIn* or *typeOut* is **GL_UNSIGNED_INT_10_10_10_2** or **GL_UNSIGNED_INT_2_10_10_10_REV** and *format* is neither **GL_RGBA** nor **GL_BGRA**.

Files

/usr/include/GL/gl.h Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glDrawPixels** subroutine, **glReadPixels** subroutine, **gluBuild1DMipmaps** subroutine, **gluBuild2DMipmaps** subroutine, **gluBuild3DMipmaps** subroutine, **gluErrorString** subroutine.

gluSphere Subroutine

Purpose

Draws a sphere.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void gluSphere( GLUquadric* quad,
   GLdouble radius,
   GLint slices,
   GLint stacks)
```

Description

The **gluSphere** subroutine draws a sphere of the supplied *radius* around the centerpoint of the origin. The sphere is subdivided around the z axis into slices (longitude) and along the z axis into stacks (latitude).

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

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

Parameters

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

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **gluCylinder** subroutine, **gluDisk** subroutine, **gluNewQuadric** subroutine, **gluPartialDisk** subroutine, **gluQuadricOrientation** subroutine, **gluQuadricTexture** subroutine.

gluTessBeginContour, gluTessEndContour

Purpose

Delimits a contour description.

Library

C bindings library: **libGL.a**

C Syntax

```
void gluTessBeginContour( GLUtesselator*  tess)
void gluTessEndContour( GLUtesselator*  tess)
```

Description

The **gluTessBeginContour** subroutine and **gluTessEndContour** subroutine delimit the definition of a polygon contour. Within each **gluTessBeginContour**/**gluTessEndContour** pair, there can be zero or more calls to **gluTessVertex**. The vertices specify a closed contour (the last vertex of each contour is automatically linked to the first). See the **gluTessVertex** reference page for more details. **gluTessBeginContour** can only be called between **gluTessBeginPolygon** and **gluTessEndPolygon**.

Parameters

tess	Specifies the tessellation object (created with **gluNewTess**).

Related Information

The **gluNewTess** subroutine, **gluTessBeginPolygon** subroutine, **gluTessVertex** subroutine, **gluTessCallback** subroutine, **gluTessProperty** subroutine, **gluTessNormal** subroutine, **gluTessEndPolygon** subroutine.

gluTessBeginPolygon Subroutine

Purpose

Delimits a polygon description.

Library

C bindings library: **libGL.a**

C Syntax

```
void gluTessBeginPolygon( GLUtesselator*  tess,
   GLvoid*  data )
```

Description

The **gluTessBeginPolygon** and **gluTessEndPolygon** routines delimit the definition of a convex, concave or self-intersecting polygon. Within each **gluTessBeginPolygon**/**gluTessEndPolygon** pair, there must be one or more calls to **gluTessBeginContour**/ **gluTessEndContour**. Within each contour, there are zero or more calls to **gluTessVertex**. The vertices specify a closed contour (the last vertex of each contour is automatically linked to the first). See the **gluTessVertex**, **gluTessBeginContour**, and **gluTessEndContour** reference pages for more details.

The parameter *data* is a pointer to a user-defined data structure. If the appropriate callback(s) are specified (see **gluTessCallback**), then this pointer is returned to the callback function(s). Thus, it is a convenient way to store per-polygon information.

Once **gluTessEndPolygon** is called, the polygon is tessellated, and the resulting triangles are described through callbacks. See **gluTessCallback** for descriptions of the callback functions.

Parameters

tess	Specifies the tessellation object (created with **gluNewTess**).
data	Specifies a pointer to user polygon data.

Examples

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

```
gluTessBeginPolygon(tobj, NULL);
 gluTessBeginContour(tobj);
   gluTessVertex(tobj, v1, v1);
   gluTessVertex(tobj, v2, v2);
   gluTessVertex(tobj, v3, v3);
   gluTessVertex(tobj, v4, v4);
 gluTessEndContour(tobj);
 gluTessBeginContour(tobj);
   gluTessVertex(tobj, v5, v5);
   gluTessVertex(tobj, v6, v6);
   gluTessVertex(tobj, v7, v7);
 gluTessEndContour(tobj);
gluTessEndPolygon(tobj);
```

Related Information

gluNewTess subroutine, **gluTessBeginContour** subroutine, **gluTessVertex** subroutine, **gluTessCallback** subroutine, **gluTessProperty** subroutine, **gluTessNormal** subroutine, **gluTessEndPolygon** subroutine.

gluTessCallback Subroutine

Purpose

Defines a callback for a tessellation object.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void gluTessCallback( GLUtesselator*  tess,
   GLenum  which,
   void (* CallBackFunc)())
```

Description

The **gluTessCallback** subroutine defines a new callback for use by a tessellation object. If the specified callback is already defined, it is replaced. If the *CallBackFunc* parameter is set to null, the existing callback becomes undefined.

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

> **Note:** There are two versions of each callback: one with user-specified polygon data and one without. If both versions of a particular callback are specified, then the callback with user-specified polygon data will be used. The *polygon_data* parameter used by some of the functions is a copy of the pointer that was specified when **gluTessBeginPolygon** was called.

The acceptable callbacks are as follows:

GLU_TESS_BEGIN
: The begin callback is invoked like **glBegin** to indicate the start of a (triangle) primitive. The function takes a single argument of type **GLenum**. If the **GLU_TESS_BOUNDARY_ONLY** property is set to **GL_FALSE**, then the argument is set to either **GL_TRIANGLE_FAN**, **GL_TRIANGLE_STRIP**, or **GL_TRIANGLES**. If the **GLU_TESS_BOUNDARY_ONLY** property is set to **GL_TRUE**, then the argument will be set to **GL_LINE_LOOP**. The function prototype for this callback is:

```
void begin(GLenum type);
```

GLU_TESS_BEGIN_DATA
: The same as the **GLU_TESS_BEGIN** callback except that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when **gluTessBeginPolygon** was called. The function prototype for this callback is:

```
void beginData ( GLenum type, void *polygon_data );
```

GLU_TESS_EDGE_FLAG

Similar to the **glEdgeFlag** subroutine. It takes a single Boolean flag that indicates which edges lie on the polygon boundary. If the flag is **GL_TRUE**, each vertex that follows begins an edge that lies on the polygon boundary, that is, an edge that seperates an interior region from an exterior one. If the flag is **GL_FALSE**, each vertex that follows begins an edge that kies in the polygon interior. To avoid confusion with the first few edges, the edge flag callback is started before the first vertex callback is made.

Because triangle fans and strips do not support edge flags, the begin callback cannot be called with **GL_TRIANGLE_FAN** or **GL_TRIANGLE_STRIP** if the **GLU_EDGE_FLAG** If a non-null edge flag callback is provided. (If the callback is initialized to null, there is no impact on performance.) Instead, fans and strips are converted into independent triangles. The function prototype for this callback is:

```
void edgeFlag(GLboolean flag);
```

GLU_TESS_EDGE_FLAG_DATA

The same as the **GLU_TESS_EDGE_FLAG** callback except that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when **gluTessBeginPolygon** was called. The function prototype for this callback is:

```
void edgeFlagData(GLboolean flag, void *polygon_data );
```

GLU_TESS_VERTEX

Started between the begin and end callbacks. It is similar to the **glVertex** subroutine and defines the vertices of the triangles created by the tessellation process. The function takes a pointer as its only argument. This pointer is identical to the opaque pointer provided by the user when the vertex was described. (See the **gluTessVertex** subroutine for details on specifying a polygon vertex.) The function prototype for this callback is:

```
void vertex (void *vertex_data);
```

GLU_TESS_VERTEX_DATA

The same as the **GLU_TESS_VERTEX** callback except that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when **gluTessBeginPolygon** was called. The function prototype for this callback is:

```
void vertexData ( void *vertex_data,
                  void *polygon_data );
```

GLU_TESS_END

Serves the same purpose as the **glEnd** subroutine and indicates the end of a primitive. It takes no arguments. The function prototype for this callback is:

```
void end(void);
```

GLU_TESS_END_DATA

The same as the **GLU_TESS_END** callback except that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when **gluTessBeginPolygon** was called. The function prototype for this callback is:

```
void endData ( void *polygon_data);
```

GLU_TESS_COMBINE

The combine callback is called to create a new vertex when the tessellation detects an intersection, or wishes to merge features. The function takes four arguments: an array of three elements each of type GLdouble, an array of four pointers, an array of four elements each of type GLfloat, and a pointer to a pointer. The prototype Is:

```
void combine( GLdouble coords[3],
              void *vertex_data[4],
              GLfloat weight[4],
              void **outData );
```

The vertex is defined as a linear combination of up to four existing vertices, stored in *vertex_data*. The coefficients of the linear combination are given by *weight*; these weights always add up to 1. All vertex pointers are valid even when some of the weights are 0. *coords* gives the location of the new vertex.

The user must allocate another vertex, interpolate parameters using *vertex_data* and *weight*, and return the new vertex pointer in *outData*. This handle is supplied during rendering callbacks. The user is responsible for freeing the memory some time after **gluTessEndPolygon** is called.

For example, if the polygon lies in an arbitrary plane in 3-space, and a color is associated with each vertex, the **GLU_TESS_COMBINE** callback might look like this:

```
void myCombine( GLdouble coords[3], VERTEX *d[4],
                GLfloat w[4], VERTEX **dataOut )
{VERTEX *new = new_vertex();

  new->x = coords[0];
  new->y = coords[1];
  new->z = coords[2];
  new->r = w[0]*d[0]->r + w[1]*d[1]->r +
w[2]*d[2]->r + w[3]*d[3]->r;
  new->g = w[0]*d[0]->g + w[1]*d[1]->g +
w[2]*d[2]->g + w[3]*d[3]->g;
  new->b = w[0]*d[0]->b + w[1]*d[1]->b +
w[2]*d[2]->b + w[3]*d[3]->b;
  new->a = w[0]*d[0]->a + w[1]*d[1]->a +
w[2]*d[2]->a + w[3]*d[3]->a;
 *dataOut = new; }
```

If the tessellation detects an intersection, then the **GLU_TESS_COMBINE** or **GLU_TESS_COMBINE_DATA** callback (see below) must be defined, and it must write a non-NULL pointer into *dataOut*. Otherwise the **GLU_TESS_NEED_COMBINE_CALLBACK** error occurs, and no output is generated.

GLU_TESS_COMBINE_DATA

The same as the **GLU_TESS_COMBINE** callback except that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when **gluTessBeginPolygon** was called. The function prototype for this callback is:

```
void combineData ( GLdouble coords[3],
                   void *vertex_data[4],
                   GLfloat weight[4],
                   void **outData,
                   void *polygon_data );
```

GLU_TESS_ERROR

Called when an error is encountered. Character strings describing these errors can be retrieved with the **gluErrorString** subroutine. The **GLenum** type is the only argument and indicates the specific error that occurred. It will be set to one of the following:

- **GLU_TESS_MISSING_BEGIN_POLYGON**
- **GLU_TESS_MISSING_END_POLYGON**
- **GLU_TESS_MISSING_BEGIN_CONTOUR**
- **GLU_TESS_MISSING_END_CONTOUR**
- **GLU_TESS_COORD_TOO_LARGE:** Indicates that some vertex coordinate exceeded the predefined constant **GLU_TESS_MAX_COORD** in absolute value, and that the value has been clamped. (Coordinate values must be small enough so that two can be multiplied together without overflow.)
- **GLU_TESS_NEED_COMBINE_CALLBACK:** Indicates that the tessellation detected an intersection between two edges in the input data, and the **GLU_TESS_COMBINE** or **GLU_TESS_COMBINE_DATA** callback was not provided. No output is generated.
- **GLU_OUT_OF_MEMORY:** Indicates that there is not enough memory so no output is generated.

Note: The GLU library will recover from the first four errors by inserting the missing call(s).

The function prototype for this callback is:

```
void error(GLenum errno);
```

GLU_TESS_ERROR_DATA

The same as the **GLU_TESS_ERROR** callback except that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when **gluTessBeginPolygon** was called. The function prototype for this callback is:

```
void errorData ( GLenum errno, void *polygon_data );
```

Parameters

tess Specifies the tessellation object created with the **gluNewTess** subroutine.

which Specifies the callback being defined. The following values are valid:

- **GLU_TESS_BEGIN**
- **GLU_TESS_BEGIN_DATA**
- **GLU_TESS_EDGE_FLAG**
- **GLU_TESS_EDGE_FLAG_DATA**
- **GLU_TESS_VERTEX**
- **GLU_TESS_VERTEX_DATA**
- **GLU_TESS_END**
- **GLU_TESS_END_DATA**
- **GLU_TESS_COMBINE**
- **GLU_TESS_COMBINE_DATA**
- **GLU_TESS_ERROR**
- **GLU_TESS_ERROR_DATA**

CallBackFunc Specifies the new callback.

Examples

Tessellated polygons can be rendered directly as in the following example:

```
gluTessCallback(tobj, GLU_TESS_BEGIN, glBegin);
gluTessCallback(tobj, GLU_TESS_VERTEX, glVertex3dv);
gluTessCallback(tobj, GLU_TESS_END, glEnd);
gluTessCallback(tobj, GLU_TESS_COMBINE, myCombine);
gluTessBeginPolygon(tobj, NULL);
  gluTessBeginContour(tobj);
    gluTessVertex(tobj, v, v);
    ...
  gluTessEndContour(tobj);
gluTessEndPolygon(tobj);
```

Typically, the tessellated polygon should be stored in a display list so that is does need to be retessellated every time it is rendered.

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glBegin** subroutine, **glEdgeFlag** subroutine, **glVertex** subroutine, **gluNewTess** subroutine, **gluTessVertex** subroutine, **gluErrorString** subroutine, **gluTessBeginPolygon** subroutine, **gluTessBeginContour** subroutine, **gluTessProperty** subroutine, **gluTessNormal** subroutine.

gluTessEndPolygon Subroutine

Purpose

Delimit a polygon description.

Library

C bindings library: **libGL.a**

C Syntax

void gluTessEndPolygon(GLUtesselator* *tess*)

Description

The **gluTessBeginPolygon** and **gluTessEndPolygon** routines delimit the definition of a convex, concave or self-intersecting polygon. Within each **gluTessBeginPolygon**/**gluTessEndPolygon** pair, there must be one or more calls to **gluTessBeginContour**/ **gluTessEndContour**. Within each contour, there are zero or more calls to **gluTessVertex**. The vertices specify a closed contour (the last vertex of each contour is automatically linked to the first). See the **gluTessVertex** subroutine, **gluTessBeginContour** subroutine and **gluTessEndContour** subroutine for more details.

Once **gluTessEndPolygon** is called, the polygon is tessellated, and the resulting triangles are described through callbacks. See **gluTessCallback** for descriptions of the callback functions.

Parameters

tess Specifies the tessellation object (created with **gluNewTess**).

Examples

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

```
gluTessBeginPolygon(tobj, NULL);
 gluTessBeginContour(tobj);
   gluTessVertex(tobj, v1, v1);
   gluTessVertex(tobj, v2, v2);
   gluTessVertex(tobj, v3, v3);
   gluTessVertex(tobj, v4, v4);
 gluTessEndContour(tobj);
 gluTessBeginContour(tobj);
   gluTessVertex(tobj, v5, v5);
   gluTessVertex(tobj, v6, v6);
   gluTessVertex(tobj, v7, v7);
 gluTessEndContour(tobj);
       gluTessEndPolygon(tobj);
```

In the above example the pointers, v1 through v7, should point to different addresses, since the values stored at these addresses will not be read by the tesselator until **gluTessEndPolygon** is called.

Related Information

The **gluNewTess** subroutine, **gluTessBeginContour** subroutine, **gluTessVertex** subroutine, **gluTessCallback** subroutine, **gluTessProperty** subroutine, **gluTessNormal** subroutine, and **gluTessBeginPolygon** subroutine.

gluTessNormal Subroutine

Purpose

Specifies a normal for a polygon.

Library

C bindings library: **libGL.a**

C Syntax

```
void gluTessNormal( GLUtesselator*  tess,
   GLdouble  valueX,
   GLdouble  valueY,
   GLdouble  valueZ)
```

Description

The **gluTessNormal** subroutine describes a normal for a polygon that the program is defining. All input data will be projected onto a plane perpendicular to one of the three coordinate axes before tessellation and all output triangles will be oriented CCW with respect to the normal (CW orientation can be obtained by reversing the sign of the supplied normal). For example, if you know that all polygons lie in the XY plane, call **gluTessNormal**(*tess*, 0.0, 0.0, 1.0) before rendering any polygons.

If the supplied normal is (0.0, 0.0, 0.0) (the initial value), the normal is determined as follows. The direction of the normal, up to its sign, is found by fitting a plane to the vertices, without regard to how the vertices are connected. It is expected that the input data lies approximately in the plane; otherwise, projection perpendicular to one of the three coordinate axes may substantially change the geometry. The sign of the normal is chosen so that the sum of the signed areas of all input contours is nonnegative (where a CCW contour has positive area).

The supplied normal persists until it is changed by another call to **gluTessNormal**.

Parameters

tess	Specifies the tessellation object (created with **gluNewTess**).
valueX	Specifies the first component of the normal.
valueY	Specifies the second component of the normal.
valueZ	Specifies the third component of the normal.

Related Information

The **gluTessBeginPolygon** subroutine and the **gluTessEndPolygon** subroutine.

gluTessProperty Subroutine

Purpose

Sets a tessellation object property.

Library

C bindings library: **libGL.a**

C Syntax

```
void gluTessProperty( GLUtesselator*  tess,
   GLenum   which,
   GLdouble  data)
```

Description

The **gluTessProperty** subroutine is used to control properties stored in a tessellation object. These properties affect the way that the polygons are interpreted and rendered. The legal values for *which* are as follows:

GLU_TESS_WINDING_RULE

Determines which parts of the polygon are on the ″interior″. *data* may be set to one of:

- **GLU_TESS_WINDING_ODD**
- **GLU_TESS_WINDING_NONZERO**
- **GLU_TESS_WINDING_POSITIVE**
- **GLU_TESS_WINDING_NEGATIVE**
- **GLU_TESS_WINDING_ABS_GEQ_TWO**

To understand how the winding rule works, consider that the input contours partition the plane into regions. The winding rule determines which of these regions are inside the polygon.

For a single contour C, the winding number of a point x is simply the signed number of revolutions we make around x as we travel once around C (where CCW is positive). When there are several contours, the individual winding numbers are summed. This procedure associates a signed integer value with each point x in the plane. Note that the winding number is the same for all points in a single region.

The winding rule classifies a region as ″inside″ if its winding number belongs to the chosen category (odd, nonzero, positive, negative, or absolute value of at least two). The previous GLU tessellator (prior to GLU 1.2) used the ″odd″ rule. The ″nonzero″ rule is another common way to define the interior. The other three rules are useful for polygon CSG Operations.

GLU_TESS_BOUNDARY_ONLY

Is a boolean value (″value″ should be set to **GL_TRUE** or **GL_FALSE**). When set to **GL_TRUE**, a set of closed contours separating the polygon interior and exterior are returned instead of a tessellation. Exterior contours are oriented CCW with respect to the normal; interior contours are oriented CW. The **GLU_TESS_BEGIN** and **GLU_TESS_BEGIN_DATA** callbacks use the type **GL_LINE_LOOP** for each contour.

GLU_TESS_TOLERANCE

Specifies a tolerance for merging features to reduce the size of the output. For example, two vertices that are very close to each other might be replaced by a single vertex. The tolerance is multiplied by the largest coordinate magnitude of any input vertex; this specifies the maximum distance that any feature can move as the result of a single merge operation. If a single feature takes part in several merge operations, the total distance moved could be larger.

Feature merging is completely optional; the tolerance is only a hint. The implementation is free to merge in some cases and not in others, or to never merge features at all. The initial tolerance is 0.

The current implementation merges vertices only if they are exactly coincident, regardless of the current tolerance. A vertex is spliced into an edge only if the implementation is unable to distinguish which side of the edge the vertex lies on. Two edges are merged only when both endpoints are identical.

Parameters

tess	Specifies the tessellation object (created with **gluNewTess**).
which	Specifies the property to be set. Valid values are: • **GLU_TESS_WINDING_RULE** • **GLU_TESS_BOUNDARY_ONLY** • **GLU_TESS_TOLERANCE**
data	Specifies the value of the indicated property.

Related Information

The gluGetTessProperty subroutine.

gluTessVertex Subroutine

Purpose

Specifies a vertex on a polygon.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void gluTessVertex(GLUtesselator*  tess,
   GLdouble * location,
   GLvoid*  data)
```

Description

The **gluTessVertex** subroutine describes a vertex on a polygon that the user is defining. Successive **gluTessVertex** calls describe a closed contour. For example, to describe a quadrilateral, the **gluTessVertex** subroutine must be called four times.

This subroutine can only be called between **gluBeginContour** and **gluEndContour**.

The *data* parameter normally points to a structure containing the vertex location, as well as other vertex-specific attributes (such as color and normal). This pointer is passed back to the user through the **GLU_TESS_VERTEX** or **GLU_TESS_VERTEX_DATA** callback after tessellation. (See the **gluTessCallback** subroutine for details on defining callbacks for a tessellation object.)

Parameters

tess	Specifies the tessellation object created with the **gluNewTess** subroutine.
location	Specifies the location of the vertex.
data	Specifies an opaque pointer that is passed back to the user with the vertex callback (as specified by the **gluTessCallback** subroutine).

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Examples

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

```
gluBeginPolygon(tobj, NULL);
 gluTessBeginContour(tobj);
   gluTessVertex(tobj, v1, v1);
   gluTessVertex(tobj, v2, v2);
   gluTessVertex(tobj, v3, v3);
   gluTessVertex(tobj, v4, v4);
 gluTessEndContour(tobj);
 gluTessBeginContour(tobj);
   gluTessVertex(tobj, v5, v5);
   gluTessVertex(tobj, v6, v6);
   gluTessVertex(tobj, v7, v7);
 gluTessEndContour(tobj);
gluTessEndPolygon(tobj);
```

Notes

It is a common error to use a local variable for *location* or *data* and store values into it as part of a loop. For example: for (i = 0; i < NVERTICES; ++i) {

```
   GLdouble data[3];
   data[0] = vertex[i][0];
   data[1] = vertex[i][1];
   data[2] = vertex[i][2];
   gluTessVertex(tobj, data, data);
   }
```

This doesn't work. Because the pointers specified by *location* and *data* might not be dereferenced until **gluTessEndPolygon** is executed, all the vertex coordinates but the very last set could be overwritten before tessellation begins.

Two common symptoms of this problem are consists of a single point (when a local variable is used for *data*) and a **GLU_TESS_NEED_COMBINE_CALLBACK** error (when a local variable is used for *location*.

Related Information

The **gluBeginPolygon** subroutine, **gluNewTess** subroutine, **gluTessBeginContour** subroutine, **gluTessCallback** subroutine, **gluTessProperty** subroutine, **gluTessNormal** subroutine, **gluTessEndPolygon** subroutine.

gluUnProject Subroutine

Purpose

Maps window coordinates to object coordinates.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
int gluUnProject( GLdouble  winX,
   GLdouble  winY,
   GLdouble  winZ,
   const GLdouble * model,
   const GLdouble * proj,
   const GLint * view,
```

```
GLdouble* objX,
GLdouble* objY,
GLdouble* objZ)
```

Description

The **gluUnProject** subroutine maps the specified window coordinates into object space coordinates using the *model*, *proj*, and *view* parameter values provided. Results are stored in *objX*, *objY*, and *objZ*. A return value of **GL_TRUE** indicates success, and **GL_FALSE** indicates failure.

To compute the coordinates (*objX*, *objY*, and *objZ*), **gluUnProject** multiplies the normalized device coordinates by the inverse of *model *proj* as follows:

```
                     (  2(winX - view[0])      )
                     |  ----------------- - 1  |
                     |      view[2]            |
(      )             |                         |
| objX |             |  2(winY - view[1])      |
| objY | = INV(PM)   |  ----------------- - 1  |
| objZ |             |      view[3]            |
(  W   )             |                         |
                     |      2(winZ) - 1        |
                     |                         |
                     (          1              )
```

INV( ) denotes matrix inversion. W is an unused variable, included for consistent matrix notation.

Parameters

winX, *winY*, *winZ*	Specify the window coordinates to be mapped.
model	Specifies the modelview matrix (as from a **glGetDoublev** call).
proj	Specifies the projection matrix (as from a **glGetDoublev** call).
view	Specifies the viewport (as from a **glGetIntegerv** call).
objX, *objY*, *objZ*	Returns the computed object coordinates.

Return Values

GL_TRUE	Indicates the projection succeeded.
GL_FALSE	Indicates the projection failed.

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glGet** subroutine, **gluProject** subroutine.

---

gluUnProject4 Subroutine

Purpose

Maps window and clip coordinates to object coordinates.

Library

`OpenGL C bindings library: libGL.a`

C Syntax

```
GLint gluUnProject4( GLdouble  winX,
   GLdouble  winY,
   GLdouble  winZ,
   GLdouble  clipW,
   const GLdouble * model,
   const GLdouble * proj,
   const GLint * view,
   GLdouble  near,
   GLdouble  far,
   GLdouble*  objX,
   GLdouble*  objY,
   GLdouble*  objZ,
   GLdouble*  objW )
```

Description

gluUnProject4 maps the specified window coordinates *winX*, *winY* and *winZ* and its clip w coordinate *clipW* into object coordinates (*objX*, *objY*, *objZ*, *objW*) using *model*, *proj* and *view*. *clipW* can be other than 1 as for vertices in **glFeedbackBuffer** when data type **GL_4D_COLOR_TEXTURE** is returned. This also handles the case where the *near* and *far* planes are different from the default, 0 and 1 respectively. A return value of **GL_TRUE** indicates success; a return value of **GL_FALSE** indicates failure. To compute the coordinates (*objX*, *objY*, *objZ* and *objW*), **gluUnProject4** multiplies the normalized device coordinates by the inverse of *model***proj* as follows:

```
                           ( 2(winX - view[0]      )
                           | ---------------- - 1  |
                           |     view[2]           |
                           |                       |
                           | 2(winY - view[1]      |
 ( objX  )                 | ---------------- - 1  |
 | objY  | = INV(PM) *     |     view[3]           |
 | objZ  |                 |                       |
 ( objW  )                 |  2(winZ - near)       |
                           |  -------------- - 1   |
                           |    far - near         |
                           |                       |
                           (       clipW           )
```

INV( ) denotes matrix inversion.

gluUnProject4 is equivalent to **gluUnProject** when *clipW* is 1, *near* is 0 and *far* is 1.

Parameters

winX, *winY*, *winZ*	Specify the window coordinates to be mapped.
clipW	Specify the clip w coordinate to be mapped.
model	Specifies the modelview matrix (as from a **glGetDoublev** call).
proj	Specifies the projection matrix (as from a **glGetDoublev** call).
view	Specifies the viewport (as from a **glGetIntegerv** call).
near, *far*	Specifies the near and far planes (as from a **glGetDoublev** call).
objX, *objY*, *objZ*, *objW*	Returns the computed object coordinates.

Files

/usr/include/GL/gl.h Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glGet** subroutine, **glFeedbackBuffer** subroutine, **gluProject** subroutine, **gluUnProject** subroutine.

Chapter 3. OpenGL in the AIXwindows (GLX) Environment

OpenGL is a high-performance, 3D-oriented renderer available in the AIXwindows system through the GLX extension. Use the **glXQueryExtension** and **glXQueryVersion** subroutines to determine whether the GLX extension is supported by an X server. If the GLX extension is supported, the **glXQueryVersion** subroutine will return the GLX version supported by the X server and client and the **glXQueryServerString** subroutine will return the GLX version supported by a particular screen (because different screens might support different GLX versions).

GLX-extended servers make a subset of their visuals available for OpenGL rendering. Drawables created with these visuals can also be rendered using the core X renderer, or any other X extension compatible with all core X visuals. In GLX 1.3, these visuals are represented via frame buffer configuration structures (FBConfigs).

GLX extends drawables with several buffers other than the standard color buffer. These buffers include back and auxiliary color buffers, a depth buffer, a stencil buffer, and a color accumulation buffer. Some or all are included in each FBConfig or X visual that supports OpenGL.

Both core X and OpenGL commands can be used to operate on the current read and write drawables, if the drawables are windows or pixmaps. However, the X and OpenGL command streams are not synchronized (except at explicitly created boundaries generated by calling the **glXWaitGL**, **glXWaitX**, **XSync**, or **glFlush** subroutines).

Related Information

OpenGL AIXwindows (GLX) Subroutines lists the GLX subroutines.

How to Render into an X Drawable gives steps on how to create an OpenGL-compatible X window.

XCreateColormap subroutine, **XCreateWindow** subroutine, **glFinish** subroutine, **glFlush** subroutine.

How to Render into an X Drawable

Procedure

To render into an X drawable:

Notes:

1. Decide which version of GLX can be used.

   The **glXQueryVersion** and **glXQueryServerString** subroutines can be used to see if the GLX 1.3 subroutines are available to be used or not.
2. If GLX 1.3 subroutines can be used:
   - Choose a FBConfig that defines the required OpenGL buffers.

     The **glXChooseFBConfig** subroutine can be used to simplify selection of a compatible FBConfig. If more control of the selection process is required, use the **glXGetFBConfigs** and **glXGetFBConfigAttrib** subroutines to select among the available FBConfigs.
   - Use the selected FBConfig to create a GLX context, a GLX drawable and an X drawable.

     GLX contexts are created with the **glXCreateNewContext** subroutine. GLX drawables are created using either the **glXCreateWindow**, **glXCreatePixmap** or **glXCreatePbuffer** subroutines. The **glXCreateWindow** subroutines requires an X window to be associated with the GLX window drawable. Therefore, the **glXGetVisualFromFBConfig** subroutine can be used to get the XVisualInfo structure for the X visual that is associated with the selected FBConfig and the **XCreateWindow** subroutine can be used to create the X window.

© Copyright IBM Corp. 1994, 2002

- Bind the context and the drawable together using the **glXMakeContextCurrent** subroutine.

  This context/drawable pair becomes the current context and current read and write drawable, and it is used by all OpenGL commands until the **glXMakeContextCurrent** or **glXMakeCurrent** subroutine is called with different arguments.

3. If GLX 1.3 subroutines can not be used:
   - Choose a visual that defines the required OpenGL buffers.

     The **glXChooseVisual** subroutine can be used to simplify selection of a compatible visual. If more control of the selection process is required, use the **XGetVisualInfo** and **glXGetConfig** subroutines to select among the available visuals.
   - Use the selected visual to create both a GLX context and an X drawable.

     GLX contexts are created with the **glXCreateContext** subroutine; drawables are created with either the **XCreateWindow** or **glXCreateGLXPixmap** subroutines.
   - Bind the context and the drawable together using the **glXMakeCurrent** subroutine.

     This context/drawable pair becomes the current context and current drawable, and it is used by all OpenGL commands until the **glXMakeCurrent** subroutine is called with different arguments.

Example

Following is an example of the minimum code required to create a red, green, blue, alpha (RGBA) format, OpenGL-compatible X window. In this example, the X window is cleared to yellow. Note that although the code is valid, no error checking is included. Under normal conditions, all return values should be tested.

```
#include <GL/glx.h>
#include <GL/gl.h>
#include <string.h>

static int AttributeList[] = { GLX_RGBA, None };

static Bool WaitForNotify(Display *d, XEvent *e, char *arg) {
   return (e->type == MapNotify) && (e->xmap.window == (Window)arg);
}

void setup_glx12(Display *dpy) {
   XVisualInfo *vi;
   Colormap cmap;
   XSetWindowAttributes swa;
   Window win;
   GLXContext cx;
   XEvent event;

   /* Get an appropriate visual */
   vi = glXChooseVisual(dpy, DefaultScreen(dpy), AttributeList);

   /* Create a GLX context */
   cx = glXCreateContext(dpy, vi, 0, GL_FALSE);

   /* Create a colormap */
   cmap = XCreateColormap(dpy, RootWindow(dpy, vi->screen),vi->visual, AllocNone);

   /* Create a window */
   swa.colormap = cmap;
   swa.border_pixel = 0;
   swa.event_mask = StructureNotifyMask;
   win = XCreateWindow(dpy, RootWindow(dpy, vi->screen), 0, 0, 100, 100, 0, vi->depth, InputOutput,
                       vi->visual, CWBorderPixel|CWColormap|CWEventMask, &swa);
   XMapWindow(dpy, win);
   XIfEvent(dpy, &event, WaitForNotify, (Char*)win);
```

```
   /* Connect the context to the window */
   glXMakeCurrent(dpy, win, cx);
}

void setup_glx13(Display *dpy) {
   GLXFBConfig *fbc;
   XVisualInfo *vi;
   Colormap cmap;
   XSetWindowAttributes swa;
   Window win;
   GLXContext cx;
   GLXWindow gwin;
   XEvent event;
   int nelements;

   /* Find a FBConfig that uses RGBA.  Note that no attribute list is */
   /* needed since GLX_RGBA_BIT is a default attribute.               */
   fbc = glXChooseFBConfig(dpy, DefaultScreen(dpy), 0, &nelements);
   vi = glXGetVisualFromFBConfig(dpy, fbc[0]);

   /* Create a GLX context using the first FBConfig in the list. */
   cx = glXCreateNewContext(dpy, fbc[0], GLX_RGBA_TYPE, 0, GL_FALSE);

   /* Create a colormap */
   cmap = XCreateColormap(dpy, RootWindow(dpy, vi->screen),vi->visual, AllocNone);

   /* Create a window */
   swa.colormap = cmap;
   swa.border_pixel = 0;
   swa.event_mask = StructureNotifyMask;
   win = XCreateWindow(dpy, RootWindow(dpy, vi->screen), 0, 0, 100, 100, 0, vi->depth, InputOutput,
                       vi->visual, CWBorderPixel|CWColormap|CWEventMask, &swa);
   XMapWindow(dpy, win);
   XIfEvent(dpy, &event, WaitForNotify, (Char*)win);

   /* Create a GLX window using the same FBConfig that we used for the */
   /* the GLX context.                                                 */
   gwin = glXCreateWindow(dpy, fbc[0], win, 0);

   /* Connect the context to the window for read and write */
   glXMakeContextCurrent(dpy, gwin, gwin, cx);
}

int main(int argc, char **argv) {
   Display *dpy;
   GLXContext cx;
   XEvent event;
   int major, minor;
   char *string_data;

   /* get a connection */
   dpy = XOpenDisplay(0);

   /*  */
   if (glXQueryVersion(dpy, &major, &minor)) {
     if (major == 1) {
       if (minor < 3) setup_glx12(dpy);
       else {
         string_data = glXGetServerString(dpy, DefaultScreen(dpy), GLX_VERSION);
         if (strchr(string_data,"1.3")) setup_glx13(dpy);
         else setup_glx12(dpy);
       }
       /* clear the buffer */
```

```
        glClearColor(1,1,0,1);
        glClear(GL_COLOR_BUFFER_BIT);
        glFlush();

        /* wait a while */
        sleep(10);
      }
    }
}
```

Special Considerations

When creating an X window, keep the following in mind:

Notes:

1. A color map must be created and passed to the **XCreateWindow** subroutine.
2. A GLX context must be created and attached to one or more X drawable or GLX drawable before OpenGL commands are processed. OpenGL commands issued while no context/drawable pair is current are ignored.
3. Exposure events indicate that *all* buffers associated with the specified window may be damaged and should be repainted. Although some visual buffers on certain systems may never require repainting (the depth buffer, for example), this is not the rule. Do not create code based on the assumption that these buffers cannot be damaged.
4. GLX subroutines (at the GLX 1.2 level and earlier) manipulate XVisualInfo structures, not pointers to visuals or visual IDs. XVisualInfo structures contain visual, visual ID, screen, and depth parameters, as well as other X-specific information. GLX 1.3 subroutines use GLXFBConfig structures instead of the XVisualInfo structures.
5. The search methods used by the **glXChooseFBConfig** and **glXChooseVisual** subroutines are different and can return dissimilar results. Using the two subroutines in the same program to create GLX contexts and drawables can lead to **BadMatch X** protocol errors being generated in subsequent calls to GLX subroutines.

Related Information

OpenGL in the AIXwindows (GLX) Environment.

OpenGL in the AIXwindows environment (GLX) Subroutines

Following is a list of the GLX subroutines and the purpose of each.

Select the subroutine about which you want to read:

Subroutine	Description
glXChooseFBConfig	Returns a list of FBConfigs matching the attributes specified. (GLX 1.3 only)
glXChooseVisual	Returns a visual matching the attributes specified.
glXCopyContext	Copies state variables from one rendering context to another.
glXCreateContext	Creates a new GLX rendering context using a visual.
glXCreateGLXPixmap	Creates an off-screen GLX rendering area, using a visual.
glXCreateNewContext	Creates a new GLX rendering context using a FBConfig. (GLX 1.3 only)
glXCreatePbuffer	Creates an off-screen, hardware GLX rendering area, using a FBConfig. (GLX 1.3 only)
glXCreatePixmap	Creates an off-screen GLX rendering area, using a FBConfig. (GLX 1.3 only)

Subroutine	Description
glXCreateWindow	Creates a GLX window drawable, using a FBConfig. (GLX 1.3 only)
glXDestroyContext	Destroys a GLX context.
glXDestroyGLXPixmap	Destroys a GLX pixmap.
glXDestroyPbuffer	Destroys a GLX pbuffer. (GLX 1.3 only)
glXDestroyPixmap	Destroys a GLX pixmap. (GLX 1.3 only)
glXDestroyWindow	Destroys a GLX window. (GLX 1.3 only)
glXFreeContextEXT	Frees client-side memory for imported context.
glXGetClientString	Returns a string describing the client.
glXGetConfig	Returns information about GLX visuals.
glXGetContextIDEXT	Gets the XID for a context.
glXGetCurrentContext	Returns the current context.
glXGetCurrentDisplay	Gets display for current context.
glXGetCurrentDrawable	Returns the current rendering drawable.
glXGetCurrentReadDrawable	Returns the current read drawable. (GLX 1.3 only)
glXGetFBConfigAttrib	Returns information about a specified FBConfig. (GLX 1.3 only)
glXGetFBConfigs	Returns a list of all FBConfigs for a specified screen. (GLX 1.3 only)
glXGetSelectedEvent	Returns the GLX Events that were selected for a specified GLX window or pbuffer. (GLX 1.3 only)
glXGetVisualFromFBConfig	Returns the XVisualInfo information for the visual that corresponds to a specified FBConfig. (GLX 1.3 only)
glXImportContextEXT	Imports another process's indirect rendering context.
glXIsDirect	Indicates whether direct rendering is enabled.
glXMakeContextCurrent	Attaches a GLX context to one or two GLX windows, GLX pbuffers or GLX pixmaps. (GLX 1.3 only)
glXMakeCurrent	Attaches a GLX context to a window or GLX pixmap.
glXQueryContext	Queries context information. (GLX 1.3 only)
glXQueryContextInfoEXT	Queries context information.
glXQueryDrawable	Queries GLX drawable information. (GLX 1.3 only)
glXQueryExtension	Indicates whether the GLX extension is supported.
glXQueryExtensionsString	Returns list of supported extensions.
glXQueryServerString	Returns string describing the server.
glXQueryVersion	Returns the version numbers of the GLX extension.
glXSelectEvent	Turns on GLX events for a specified GLX drawable. (GLX 1.3 only)
glXSwapBuffers	Exchanges front and back buffers.
glXUseXFont	Creates bitmap display lists from an X font.
glXWaitGL	Completes GL processing prior to subsequent X calls.
glXWaitX	Completes X processing prior to subsequent OpenGL calls.

glXChooseFBConfig Subroutine

Purpose

Returns a list of GLX FBConfigs that match the attributes specified.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLXFBConfig *glXChooseFBConfig(Display * dpy,
                               int   screen,
                               int * AttributeList,
                               int * nelements)
```

Description

The **glXChooseFBConfig** subroutine returns a pointer to a list of GLX FBConfig structures that match a specified list of attributes. The GLX attributes of the returned GLX FBConfigs match or exceed the the specified values, based on the table, below. To free the data returned by this function, use the **XFree** subroutine.

If an attribute is not specified in *AttributeList* then the default value will be used instead. If the default value is **GLX_DONT_CARE** and the attribute is not in *AttributeList* then the attribute will not be checked. **GLX_DONT_CARE** may be specified for all attributes except **GLX_LEVEL**. If **GLX_DONT_CARE** is specified for an attribute then the attribute will not be checked. If *AttributeList* is **NULL** or empty (that is, the first attribute is **None** (or 0)), then the selection and sorting of the GLXFBConfigs is done according to the default values.

To retrieve a GLX FBConfig given its XID, use the **GLX_FBCONFIG_ID** attribute. When **GLX_FBCONFIG_ID** is specified, all other attributes are ignored and only the GLX FBConfig with the given XID is returned (**NULL** (or 0) is returned if it does not exist).

The following attributes can be specified in *AttributeList* but they will be ignored:

- **GLX_MAX_PBUFFER_WIDTH**
- **GLX_MAX_PBUFFER_HEIGHT**
- **GLX_MAX_PBUFFER_PIXELS**
- **GLX_VISUAL_ID**

If **GLX_TRANSPARENT_TYPE** is set to **GLX_NONE** in *AttributeList*, then the following attributes can be included in *AttributeList* but they will be ignored:

- **GLX_TRANSPARENT_INDEX_VALUE**
- **GLX_TRANSPARENT_RED_VALUE**
- **GLX_TRANSPARENT_GREEN_VALUE**
- **GLX_TRANSPARENT_BLUE_VALUE**
- **GLX_TRANSPARENT_ALPHA_VALUE**

Attribute[1]	**Default Value**	**Selection Criteria**[2]	**Sorting Criteria**[3]	**Sort Priority**[3]
GLX_ FBCONFIG_ ID	GLX_ DONT_ CARE	*Exact*		
GLX_ BUFFER_ SIZE	0	*Larger*	*Smaller*	3
GLX_ LEVEL	0	*Exact*		

Attribute[1]	Default Value	Selection Criteria[2]	Sorting Criteria[3]	Sort Priority[3]
GLX_ DOUBLEBUFFER	GLX_ DONT_ CARE	*Exact*	*Exact*	4
GLX_ STEREO	False	*Exact*		
GLX_ AUX_ BUFFERS	0	*Larger*	*Smaller*	5
GLX_ RED_ SIZE	0	*Larger*	*Larger*	2
GLX_ GREEN_ SIZE	0	*Larger*	*Larger*	2
GLX_ BLUE_ SIZE	0	*Larger*	*Larger*	2
GLX_ ALPHA_ SIZE	0	*Larger*	*Larger*	2
GLX_ DEPTH_ SIZE	0	*Larger*	*Larger*	6
GLX_ STENCIL_ SIZE	0	*Larger*	*Larger*	7
GLX_ ACCUM_ RED_ SIZE	0	*Larger*	*Larger*	8
GLX_ ACCUM_ GREEN_ SIZE	0	*Larger*	*Larger*	8
GLX_ ACCUM_ BLUE_ SIZE	0	*Larger*	*Larger*	8
GLX_ ACCUM_ ALPHA_ SIZE	0	*Larger*	*Larger*	8
GLX_ RENDER_ TYPE	GLX_ RGBA_ BIT	*Mask*		
GLX_ DRAWABLE_ TYPE	GLX_ WINDOW_ BIT	*Mask*		
GLX_ X_ RENDERABLE	GLX_ DONT_ CARE	*Exact*		
GLX_ X_ VISUAL_ TYPE	GLX_ DONT_ CARE	*Exact*	*Exact*	9
GLX_ CONFIG_ CAVEAT	GLX_ DONT_ CARE	*Exact*	*Exact*	1
GLX_ TRANSPARENT_ TYPE	GLX_ NONE	*Exact*		
GLX_ TRANSPARENT_ INDEX_ VALUE	GLX_ DONT_ CARE	*Exact*		
GLX_ TRANSPARENT_ RED_ VALUE	GLX_ DONT_ CARE	*Exact*		
GLX_ TRANSPARENT_ GREEN_ VALUE	GLX_ DONT_ CARE	*Exact*		
GLX_ TRANSPARENT_ BLUE_ VALUE	GLX_ DONT_ CARE	*Exact*		
GLX_ TRANSPARENT_ ALPHA_ VALUE	GLX_ DONT_ CARE	*Exact*		

Table Notes:

1. See the **glXGetFBConfigAttrib** subroutine for the definition of each of the GLX FBConfig attributes.
2. The values in the **Selection criteria** column have the following meaning:

   Larger GLX FBConfigs with an attribute value that meets or exceeds the specified value are returned

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

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

3. If more than one matching GLX FBConfig is found, then a list of GLX FBConfigs, sorted according to the *best* match criteria, is returned. The list is sorted according to the following precedence rules that are applied in ascending order:
   a. By **GLX_CONFIG_CAVEAT** where the precedence is:
      - **GLX_NONE**
      - **GLX_SLOW_CONFIG**
      - **GLX_NON_CONFORMANT_CONFIG**

b. Larger total number of RGBA color bits (**GLX_RED_SIZE**, **GLX_GREEN_SIZE**, **GLX_BLUE_SIZE**, plus **GLX_ALPHA_SIZE**). If the requested number of bits in *AttributeList* for a particular color component is 0 or **GLX_DONT_CARE**, then the number of bits for that component is not considered.
c. Smaller **GLX_BUFFER_SIZE**.
d. Single-buffered configuration (**GLX_DOUBLE_BUFFER** is **False**) precedes a double-buffered configuration.
e. Smaller **GLX_AUX_BUFFERS**.
f. Larger **GLX_DEPTH_SIZE**.
g. Smaller **GLX_STENCIL_BITS**.
h. Larger total number of accumulation buffer color bits (**GLX_ACCUM_RED_SIZE**, **GLX_ACCUM_GREEN_SIZE**, **GLX_ACCUM_BLUE_SIZE**, plus **GLX_ACCUM_ALPHA_SIZE**). If the requested number of bits in *AttributeList* for a particular color component is 0 or **GLX_DONT_CARE**, then the number of bits for that color component is not considered.
i. By **GLX_X_VISUAL_TYPE** where the precedence is:
   - **GLX_TRUE_COLOR**
   - **GLX_DIRECT_COLOR**
   - **GLX_PSEUDO_COLOR**
   - **GLX_STATIC_COLOR**
   - **GLX_GRAY_SCALE**
   - **GLX_STATIC_GRAY**

Parameters

dpy	Specifies the connection to the X server.
screen	Specifies the screen number.
AttributeList	Specifies a list of attribute/value pairs. The last attribute must be **None** (or 0). **Note:** The format of this list is **not** the same as found in **glXChooseVisual**. All attributes (including the boolean attributes) must be paired with a corresponding value in this list.
nelements	Returns the number of FBConfigs that are in the returned list.

Notes

This subroutine requires GLX 1.3 support on both the GLX system on the client and on the specified screen on the X server.

The search methods used by the **glXChooseFBConfig** and **glXChooseVisual** subroutines are different and can return dissimilar results. Using the two subroutines in the same program to create GLX contexts and drawables can lead to **BadMatch X** protocol errors being generated in subsequent calls to GLX subroutines.

Return Values

Null	Indicates that either an undefined GLX attribute is encountered in the specified *AttributeList*, that no FBConfig matches the specified values for the GLX attributes or that *screen* is invalid.

Examples

The following example specifies a single-buffered RGB FBConfig in the normal frame buffer (not an overlay or underlay). The returned visual supports at least 4 bits each of red, green, and blue and possibly no alpha bits. It does not support color-index mode, double-buffering, stereo display or transparency. The code shown in the example may or may not have one or more auxiliary color buffers, a depth buffer, a stencil buffer, or an accumulation buffer.

```
AttributeList = { GLX_DOUBLE_BUFFER, False, GLX_RED_SIZE, 4, GLX_GREEN_SIZE, 4, GLX_BLUE_SIZE,
4, None};
```

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXCreateNewContext** subroutine, **glXGetFBConfigAttrib** subroutine, **glXGetFBConfigs** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXChooseVisual Subroutine

Purpose

Returns a visual matching the attributes specified.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

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

Description

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

All Boolean GLX attributes default to False, except for **GLX_USE_GL**. The **GLX_USE_GL** attribute defaults to True. All integer GLX attributes default to 0 (zero). Default specifications are superseded by attributes included in *AttributeList* specified. Boolean attributes included in the specified *AttributeList* are understood to be True. Integer attributes are followed immediately by the corresponding specified (or minimum) value. The *AttributeList* must be terminated with the **None** attribute.

The GLX visual attributes are defined as follows:

Attribute	Definition
GLX_USE_GL	This attribute is ignored. Only visuals that can be rendered with GLX are considered.
GLX_BUFFER_SIZE	This attribute must be followed by a nonnegative integer indicating the desired color index buffer size. The smallest index buffer of at least the specified size is preferred. This attribute is ignored if the **GLX_RGBA** attribute is asserted.
GLX_LEVEL	This attribute must be followed by an integer buffer-level specification. This specification is honored exactly. Buffer level 0 (zero) corresponds to the default frame buffer of the display. Buffer level 1 (one) is the first overlay frame buffer, level 2 the second overlay frame buffer, and so on. Negative buffer levels correspond to underlay frame buffers.
GLX_RGBA	This attribute specifies that if present, only TrueColor and DirectColor visuals are considered. Otherwise, only PseudoColor and StaticColor visuals are considered.
GLX_DOUBLEBUFFER	This attribute specifies that only double-buffered visuals are considered. Otherwise, only single-buffered visuals are considered.
GLX_STEREO	This attribute specifies that only stereo visuals are to be considered. Otherwise, only monoscopic visuals are considered.
GLX_AUX_BUFFERS	This attribute must be followed by a nonnegative integer indicating the desired number of auxiliary buffers (preferably visuals with the smallest number of auxiliary buffers that meets or exceeds the specified number).
GLX_RED_SIZE	This attribute must be followed by a nonnegative minimum size specification. If 0, the smallest available red buffer is preferred. Otherwise, the largest available red buffer of at least the minimum size is preferred.
GLX_GREEN_SIZE	This attribute must be followed by a nonnegative minimum size specification. If 0, the smallest available green buffer is preferred. Otherwise, the largest available green buffer of at least the minimum size is preferred.
GLX_BLUE_SIZE	This attribute must be followed by a nonnegative minimum size specification. If 0, the smallest available blue buffer is preferred. Otherwise, the largest available blue buffer of at least the minimum size is preferred.
GLX_ALPHA_SIZE	This attribute must be followed by a nonnegative minimum size specification. If 0, the smallest available alpha buffer is preferred. Otherwise, the largest available alpha buffer of at least the minimum size is preferred.
GLX_DEPTH_SIZE	This attribute must be followed by a nonnegative minimum size specification. If 0, visuals with no depth buffer are preferred. Otherwise, the largest available depth buffer of at least the minimum size is preferred.
GLX_STENCIL_SIZE	This attribute must be followed by a nonnegative integer indicating the desired number of stencil bitplanes. The smallest stencil buffer of at least the specified size is preferred. If the desired value is 0, visuals with no stencil buffer are preferred.

Attribute	Definition
GLX_ACCUM_RED_SIZE	This attribute must be followed by a nonnegative minimum size specification. If 0, visuals with no red accumulation buffer are preferred. Otherwise, the largest possible red accumulation buffer of at least the minimum size is preferred.
GLX_ACCUM_GREEN_SIZE	This attribute must be followed by a nonnegative minimum size specification. If 0, visuals with no green accumulation buffer are preferred. Otherwise, the largest possible green accumulation buffer of at least the minimum size is preferred.
GLX_ACCUM_BLUE_SIZE	This attribute must be followed by a nonnegative minimum size specification. If 0, visuals with no blue accumulation buffer are preferred. Otherwise, the largest possible blue accumulation buffer of at least the minimum size is preferred.
GLX_ACCUM_ALPHA_SIZE	This attribute must be followed by a nonnegative minimum size specification. If 0, visuals with no alpha accumulation buffer are preferred. Otherwise, the largest possible alpha accumulation buffer of at least the minimum size is preferred.
GLX_TRANSPARENT_TYPE_EXT	This attribute defines the type of transparency (if any) in the visual. It must be one of the following: **GLX_NONE_EXT** no transparency **GLX_TRANSPARENT_INDEX_EXT** PseudoColor transparency **GLX_TRANSPARENT_RGB_EXT** RGB Transparency
GLX_TRANSPARENT_RED_VALUE_EXT	This attribute must be followed by the red value of the RGB transparent pixel.
GLX_TRANSPARENT_BLUE_VALUE_EXT	This attribute must be followed by the blue value of the RGB transparent pixel.
GLX_TRANSPARENT_GREEN_VALUE_EXT	This attribute must be followed by the green value of the RGB transparent pixel.
GLX_TRANSPARENT_ALPHA_VALUE_EXT	This attribute must be followed by the alpha value of the RGB transparent pixel.
GLX_TRANSPARENT_INDEX_VALUE_EXT	This attribute must be followed by the INDEX transparent pixel.
GLX_X_VISUAL_TYPE_EXT	This attribute must be followed by the visual type: **GLX_TRUE_COLOR_EXT** TrueColor colormap **GLX_DIRECT_COLOR_EXT** DirectColor colormap **GLX_PSEUDO_COLOR_EXT** PseudoColor colormap **GLX_STATIC_COLOR_EXT** StaticColor colormap **GLX_GRAY_SCALE_EXT** Grayscale colormap **GLX_STATIC_GRAY_EXT** StaticGray colormap
GLX_VISUAL_CAVEAT_EXT	This attribute must be followed by:

Attribute	Definition
	0 or GLX_NONE_EXT no rating **GLX_SLOW_VISUAL_EXT** not an optimal visual

Parameters

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

Notes

XVisualInfo is defined in the **Xutil.h** file. It is a structure that includes *Visual*, *VisualID*, *Screen*, and *Depth* elements.

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

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

The search methods used by the **glXChooseFBConfig** and **glXChooseVisual** subroutines are different and can return dissimilar results. Using the two subroutines in the same program to create GLX contexts and drawables can lead to **BadMatch X** protocol errors being generated in subsequent calls to GLX subroutines.

There is no direct filter for picking only visuals that support GLX pixmaps. GLX pixmaps are supported for visuals whose **GLX_BUFFER_SIZE** is one of the pixmap depths supported by the X server.

Return Values

Null	Indicates that an undefined GLX attribute is encountered in the specified *AttributeList*.

Examples

The following example specifies a single-buffered RGB visual in the normal frame buffer (not an overlay or underlay). The returned visual supports at least 4 bits each of red, green, and blue and possibly no alpha bits. It does not support color-index mode, double-buffering, or stereo display. The code shown in the example may or may not have one or more auxiliary color buffers, a depth buffer, a stencil buffer, or an accumulation buffer.

```
AttributeList = {GLX_RGBA, GLX_RED_SIZE, 4, GLX_GREEN_SIZE, 4, GLX_BLUE_SIZE, 4, None};
```

GLX implementers are strongly discouraged from changing the selection algorithm used by the **glXChooseVisual** subroutine. Selections may change from between releases of the client-side library.

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXCreateContext** subroutine, **glXGetConfig** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

---

glXCopyContext Subroutine

Purpose

Copies state variables from one rendering context to another.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glXCopyContext(Display * dpy,
                    GLXContext  Source,
                    GLXContext  Destination,
                    GLuint  Mask)
```

Description

The **glXCopyContext** subroutine copies selected groups of state variables from the specified *Source* to the specified *Destination*. The *Mask* parameter identifies the state variable groups to be copied. The *Mask* parameter contains the bitwise OR of the same symbolic names that are passed to the **glPushAttrib** subroutine. The **GL_ALL_ATTRIB_BITS** single symbolic constant can be used to copy the maximum possible portion of the rendering state.

This subroutine is successful only if the renderers named by the *Source* and *Destination* parameters share an address space.

If both rendering contexts are nondirect, it is not necessary for the calling threads to share an address space; however, their related rendering contexts must share an address space.

If *Source* is not the current context for the thread issuing the request, the state of the *Source* is undefined.

Not all values for OpenGL state can be copied. For example, pixel pack and unpack state, render mode state, and select and feedback state cannot be copied using this subroutine. The state that is manipulated by the **glPushAttrib** subroutine is the only one that can be copied.

Parameters

dpy	Specifies the connection to the X server.
Source	Specifies the source context.
Destination	Specifies the destination context.
Mask	Specifies which portions of the *Source* state are to be copied to the *Destination*.

Notes

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

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

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

Error Codes

BadMatch	Is generated if *Source* and *Destination* either do not share an address space or were not created with respect to the same screen.
BadAccess	Is generated if *Destination* is current to any thread (including the calling thread) at the time the **glXCopyContext** subroutine is called, or *Source* is current to any thread other than the calling thread.
BadValue	Is generated if undefined *Mask* bits are specified.
GLXBadContext	Is generated if either *Source* or *Destination* is not a valid GLX context.
GLXBadCurrentWindow	Is generated if *Source* is the current context and the current drawable is a window that is no longer valid.

Files

/usr/include/GL/gl.h	Contains C language constraints, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glPushAttrib** or **glPopAttrib** subroutine, **glXCreateContext** subroutine, **glXCreateNewContext** subroutine, **glXIsDirect** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXCreateContext Subroutine

Purpose

Creates a new GLX rendering context.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLXContext glXCreateContext(Display * dpy
                            XVisualInfo * Visual
                            GLXContext  ShareList
                            Bool  Direct)
```

Description

The **glXCreateContext** subroutine creates a GLX rendering context and returns its handle. This context can be used to render into both windows and GLX pixmaps. If the **glXCreateContext** subroutine fails to create a rendering context, Null is returned.

If *Direct* is set to True, a direct rendering context is created if the implementation supports direct rendering and the connection is to a local X server. If *Direct* is set to False, a rendering context that renders through the X server is created. Direct rendering provides a performance advantage in some implementations. However, direct rendering contexts cannot be shared outside a single process or used to render to GLX pixmaps.

If *ShareList* is not Null, all display-list indexes and definitions are shared by both the *ShareList* context and the newly created context. An arbitrary number of contexts can share a single display-list space. However, all rendering contexts that share a single display-list space must exist in the same address space. Two rendering contexts share an address space if both are nondirect and use the same server, or if both are direct and owned by a single process.

If both rendering contexts are nondirect, it is not necessary for the calling threads to share an address space; however, their related rendering contexts must share the address space.

Parameters

dpy	Specifies the connection to the X server.
Visual	Specifies the visual that defines the frame buffer resources available to the rendering context. It is a pointer to an **XVisualInfo** structure, not a visual ID or a pointer to a **Visual** structure.
ShareList	Specifies the context with which to share display lists. Null indicates no sharing.
Direct	A value of True specifies that rendering be done through a direct connection to the graphics system if possible; a value of False specifies rendering through the X server.

Notes

XVisualInfo is defined in the **Xutil.h** file. It is a structure that includes *Visual*, *VisualID*, *Screen*, and *Depth* elements.

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

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

Error Codes

Null	Is returned if the **glXCreateContext** subroutine fails to create a rendering context on the client side.
BadAlloc	Is generated if the server does not have enough resources to allocate the new context.
BadMatch	Is generated if the context to be created cannot share the address space or the screen of the context specified by *ShareList*, or the specified visual is not available.
GLXBadContext	Is generated if *ShareList* is not a GLX context and is not Null.
BadValue	Is generated if the *Visual* parameter specifies an invalid visual (for example, if the GLX implementation does not support it).

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXCreateNewContext** subroutine, **glXDestroyContext** subroutine, **glXGetConfig** subroutine, **glXIsDirect** subroutine, **glXMakeContextCurrent** subroutine, **glXMakeCurrent** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXCreateGLXPixmap Subroutine

Purpose

Creates an off-screen GLX rendering area.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLXPixmap glXCreateGLXPixmap(Display * dpy,
                             XVisualInfo * Visual,
                             Pixmap  Pixmap)
```

Description

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

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

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

Parameters

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

Notes

XVisualInfo is defined in the **Xutil.h** file. It is a structure that includes *Visual*, *VisualID*, *Screen*, and *Depth* elements.

Error Codes

BadAlloc	Is generated if the server cannot allocate the GLX pixmap.
BadMatch	Is generated if one or more of the following is detected: the depth of *Pixmap* does not match the **GLX_BUFFER_SIZE** value of *Visual*, *Pixmap* was not created with respect to the same screen as *Visual*.
BadPixmap	Is generated if *Pixmap* is not a valid pixmap.
BadValue	Is generated if *Visual* is not a valid XVisualInfo pointer (for example, if the GLX implementation does not support this visual).

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.

Related Information

The **glXCreateContext** subroutine, **glXIsDirect** subroutine, **glXMakeCurrent** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

---

glXCreateNewContext Subroutine

Purpose

Creates a new GLX rendering context.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLXContext glXCreateNewContext(Display * dpy
                               GLXFBConfig  config
                               int  renderType
                               GLXContext  ShareList
                               Bool  Direct)
```

Description

The **glXCreateNewContext** subroutine creates a GLX rendering context and returns its handle. This context can be used to render into GLX windows, GLX pixmaps and GLX pbuffers. If the **glXCreateNewContext** subroutine fails to create a rendering context, Null is returned.

If *Direct* is set to True, a direct rendering context is created if the implementation supports direct rendering and the connection is to a local X server. If *Direct* is set to False, a rendering context that renders through the X server is created. Direct rendering provides a performance advantage in some implementations. However, direct rendering contexts cannot be shared outside a single process or used to render to GLX pixmaps. If a direct rendering context cannot be created, then an attempt to create an indirect context instead.

If *ShareList* is not Null, all display-list indexes and definitions are shared by both the *ShareList* context and the newly created context. An arbitrary number of contexts can share a single display-list space. However, all rendering contexts that share a single display-list space must exist in the same address space. Two rendering contexts share an address space if both are nondirect and use the same server, or if both are direct and owned by a single process.

If both rendering contexts are nondirect, it is not necessary for the calling threads to share an address space; however, their related rendering contexts must share the address space.

Parameters

dpy	Specifies the connection to the X server.
config	Specifies the GLX FBConfig that defines the frame buffer resources available to the rendering context.
renderType	Specifies the type of rendering that the context will support. One of the following values can be used: **GLX_RGBA_TYPE** This is used if the context is to support RGBA rendering. **GLX_COLOR_INDEX_TYPE** This is used if the context is to support color index rendering.
ShareList	Specifies the context with which to share display lists. Null indicates no sharing.
Direct	A value of True specifies that rendering be done through a direct connection to the graphics system if possible; a value of False specifies rendering through the X server.

Notes

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

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

Error Codes

Null	Is returned if the **glXCreateNewContext** subroutine fails to create a rendering context on the client side.
BadAlloc	Is generated if the server does not have enough resources to allocate the new context.
BadMatch	Is generated if the context to be created cannot share the address space or the screen of the context specified by *ShareList* is not available.
BadValue	Is generated if the *renderType* parameter specifies an invalid rendering type.
GLXBadContext	Is generated if *ShareList* is not a GLX context and is not Null.
GLXBadFBConfig	Is generated if *config* is not a valid GLX FBConfig.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

glXDestroyContext subroutine, **glXGetFBConfigAttrib** subroutine, **glXIsDirect** subroutine, **glXMakeContextCurrent** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXCreatePbuffer Subroutine

Purpose

Creates an off-screen GLX rendering area in a non-visible framebuffer area.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLXPbuffer glXCreatePbuffer(Display * dpy,
                            GLXFBConfig  config,
                            const int * AttributeList)
```

Description

The **glXCreatePbuffer** subroutine creates an off-screen rendering area in a non-visible area of the framebuffer and returns its XID. Any GLX rendering context that was created with respect to the *config* parameter can be used to render into this off-screen area. Use the **glXMakeContextCurrent** subroutine to associate the rendering area with a GLX rendering context.

The resulting pbuffer will contain color buffers and ancillary buffers as specified by the *config* parameter, GLX pbuffers with double-buffering are supported. The **glXSwapBuffers** subroutine can be called to swap the front and back buffers.

Parameters

dpy	Specifies the connection to the X server.
config	Specifies the GLX FBConfig that defines the structure of the rendering area.

AttributeList Specifies a list of GLX attribute/value pairs that help define the GLX Pbuffer. The list has the same structure as described for the **glXChooseFBConfig** subroutine. The following attributes can be used in the attribute list:

Attributes
Description

GLX_PBUFFER_WIDTH
Specifies the pixel width of the rectangular pbuffer. This defaults to 0.

GLX_PBUFFER_HEIGHT
Specifies the pixel height of the rectangular pbuffer. This defaults to 0.

GLX_LARGEST_PBUFFER
A boolean value that specifies that the largest available pbuffer should be gotten if the allocation of the pbuffer would otherwise fail. The width and height of the allocated pbuffer will never exceed the values of **GLX_PBUFFER_WIDTH** and **GLX_PBUFFER_HEIGHT**, respectively. Use **glXQueryDrawable** to retrieve the dimensions of the allocated pbuffer. By default, **GLX_LARGEST_PBUFFER** is set to False.

GLX_PRESERVED_CONTENTS
A boolean value. If it is specified as False, then an *unpreserved* pbuffer is created and the contents of the pbuffer may be lost at any time. Once the contents of an unpreserved pbuffer have been lost, it is considered to be in a *damaged* state. It is not an error to render to a pbuffer that is in this state but the effect of rendering to it is the same as if the pbuffer were destroyed: the context state will be updated but the frame buffer state becomes undefined. It is also not an error to query the pixel contents of such a pbuffer, but the values of the returned pixels are undefined.

If it is specified as True (the default value), then when a resource conflict occurs the contents of the pbuffer will be *preserved*. In either case, the application can register to receive a pbuffer clobber event, which is generated when the pbuffer contents have been preserved or have been damaged. (See **glXSelectEvent** for more information).

Since the contents of an unpreserved pbuffer can be lost at any time with only asynchronous notification (via the pbuffer clobber event), the only way an application can guarantee that valid pixels are read back with **glReadPixels** is by grabbing the X server. Applications that don't wish to do this can check if the data returned by **glReadPixels** is valid by calling **XSync** and then checking the event queue for pbuffer clobber events (assuming that these events had been pulled off of the queue prior to the call to **glReadPixels**.

Error Codes

BadAlloc	Is generated if the server cannot allocate the GLX pbuffer.
BadFBConfig	Is generated if *config* is not a valid GLX FBConfig.
BadMatch	Is generated if *config* does not support pbuffer rendering.
BadValue	Is generated if the value for **GLX_PBUFFER_WIDTH** or **GLX_PBUFFER_HEIGHT** is zero or less (IBM X server only). Note that, by default, the values of these attributes are zero.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXCreateNewContext** subroutine, **glXMakeContextCurrent** subroutine, **glXChooseFBConfig** subroutine, **glXSelectEvent** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

---

glXCreatePixmap Subroutine

Purpose

Creates an off-screen GLX rendering area.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLXPixmap glXCreatePixmap(Display * dpy,
                          GLXFBConfig  config,
                          Pixmap  Pixmap,
                          const int * AttributeList)
```

Description

The **glXCreatePixmap** subroutine creates an off-screen rendering area and returns its XID. Any GLX rendering context that was created with respect to the *config* parameter can be used to render into this off-screen area. Use the **glXMakeContextCurrent** subroutine to associate the rendering area with a GLX rendering context.

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

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

Parameters

dpy	Specifies the connection to the X server.
config	Specifies the GLX FBConfig that defines the structure of the rendering area.
Pixmap	Specifies the X pixmap that is used as the front left color buffer of the off-screen rendering area.
AttributeList	Specifies a list of GLX attribute/value pairs that help define the GLX pixmap. Currently, there are no attributes that affect GLX pixmaps so list parameter must either be NULL or an empty list (first attribute of 0).

Error Codes

BadAlloc	Is generated if the server cannot allocate the GLX pixmap.
BadMatch	Is generated if one or more of the following is detected: the depth of *Pixmap* does not match the **GLX_BUFFER_SIZE** value of *config* or *Pixmap* was not created with respect to the same screen as *config*.
BadPixmap	Is generated if *Pixmap* is not a valid pixmap.
BadFBConfig	Is generated if *config*is not a valid GLX FBConfig or if the GLX FBConfig does not support pixmap rendering.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXCreateNewContext** subroutine, **glXIsDirect** subroutine, **glXMakeContextCurrent** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXCreateWindow Subroutine

Purpose

Creates an on-screen GLX rendering area.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLXWindow glXCreateWindow(Display * dpy,
                          GLXFBConfig  config,
                          Window  window,
                          const int * AttributeList)
```

Description

The **glXCreateWindow** subroutine creates an on-screen rendering area and returns its XID. Any GLX rendering context that was created with respect to the *config* parameter can be used to render into this on-screen area. Use the **glXMakeContextCurrent** subroutine to associate the rendering area with a GLX rendering context.

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

Parameters

dpy	Specifies the connection to the X server.
config	Specifies the GLX FBConfig that defines the structure of the rendering area.
window	Specifies the X window that is used as the on-screen rendering area.
AttributeList	Specifies a list of GLX attribute/value pairs that help define the GLX window. Currently, there are no attributes that affect GLX windows so list parameter must either be NULL or an empty list (first attribute of 0).

Error Codes

BadAlloc	Is generated if the server cannot allocate the GLX window or if *window* is already associated with another GLX FBConfig (as a result of a previous invocation of glXCreateWindow).

BadMatch	Is generated if one or more of the following is detected: *window* was not created with the visual that corresponds to *config*, if *config* does not support rendering to GLX windows.
BadWindow	Is generated if *window* is not a valid window.
GLXBadFBConfig	Is generated if *config* is not a valid GLX FBConfig.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXCreateNewContext** subroutine, **glXMakeContextCurrent** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXDestroyContext Subroutine

Purpose

Destroys a GLX context.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glXDestroyContext(Display * dpy,
                       GLXContext  Context)
```

Description

If the *Context* parameter is not current to any thread, the **glXDestroyContext** subroutine destroys it immediately. If *Context* is current, the **glXDestroyContext** subroutine destroys it when it becomes not current to any thread. In either case, the resource ID referenced by *Context* is freed immediately.

Parameters

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

Error Codes

GLXBadContext	Is generated if *Context* is not a valid GLX context.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXCreateContext** subroutine, **glXCreateNewContext** subroutine, **glXMakeContextCurrent** subroutine, **glXMakeCurrent** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXDestroyGLXPixmap Subroutine

Purpose

Destroys a GLX pixmap.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glXDestroyGLXPixmap(Display * dpy,
                         GLXPixmap  Pixmap)
```

Description

If GLX pixmap *Pixmap* is not current to any client, the **glXDestroyGLXPixmap** subroutine destroys it immediately. Otherwise, *Pixmap* is destroyed when it is no longer current to any client. In either case, the resource ID is freed immediately.

Parameters

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

Error Codes

GLXBadPixmap	Is generated if *Pixmap* is not a valid GLX pixmap.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXCreateGLXPixmap** subroutine, **glXMakeContextCurrent** subroutine, **glXMakeCurrent** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXDestroyPbuffer Subroutine

Purpose

Destroys a GLX pbuffer.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glXDestroyPbuffer(Display * dpy,
                       GLXPbuffer  Pbuffer)
```

Description

If GLX pbuffer *Pbuffer* is not current to any client, the **glXDestroyPbuffer** subroutine destroys it immediately. Otherwise, *Pbuffer* is destroyed when it is no longer current to any client. In either case, the resource ID is freed immediately.

Parameters

dpy	Specifies the connection to the X server.
Pbuffer	Specifies the GLX pbuffer to be destroyed.

Error Codes

GLXBadPbuffer	Is generated if *Pbuffer* is not a valid GLX pbuffer.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXCreatePbuffer** subroutine, **glXMakeContextCurrent** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXDestroyPixmap Subroutine

Purpose

Destroys a GLX pixmap.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glXDestroyPixmap(Display * dpy,
                      GLXPixmap  Pixmap)
```

Description

If GLX pixmap *Pixmap* is not current to any client, the **glXDestroyPixmap** subroutine destroys it immediately. Otherwise, *Pixmap* is destroyed when it is no longer current to any client. In either case, the resource ID is freed immediately.

Parameters

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

Error Codes

GLXBadPixmap	Is generated if *Pixmap* is not a valid GLX pixmap.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXCreatePixmap** subroutine, **glXMakeContextCurrent** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXDestroyWindow Subroutine

Purpose

Destroys a GLX window.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glXDestroyWindow(Display * dpy,
                      GLXWindow  Window)
```

Description

If GLX window *Window* is not current to any client, the **glXDestroyWindow** subroutine destroys it immediately. Otherwise, *Window* is destroyed when it is no longer current to any client. In either case, the resource ID is freed immediately.

Parameters

dpy	Specifies the connection to the X server.
Window	Specifies the GLX window to be destroyed.

Error Codes

GLXBadWindow	Is generated if *Window* is not a valid GLX window.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXCreateWindow** subroutine, **glXMakeContextCurrent** subroutine.

glXFreeContextEXT Subroutine

Purpose

Frees client-side memory for imported context.

Library

C bindings library: **libGL.a**

C Syntax

```
void glXFreeContextEXT(Display  *dpy,
                       GLXContext  ctx)
```

Description

The **glXFreeContextEXT** subroutine frees the client-side part of a GLXContext that was created with **glXImportContextEXT**. The **glXFreeContextEXT** subroutine does not free the server-side context information or the XID associated with the server-side context.

The **glXFreeContextEXT** subroutine is part of the **EXT_import_context** extension, not part of the core GLX command set. If **GLX_EXT_import_context** is included in the string returned by **glXQueryExtensionsString**, when called with argument **GLX_EXTENSIONS**, extension **EXT_vertex_array** is supported.

Parameters

dpy	Specifies the connection to the X server.
ctx	Specifies a GLX rendering context.

Errors

GLXBadContext is generated if *ctx* does not refer to a valid context.

Related Information

The **glXCreateContext** subroutine, **glXCreateNewContext** subroutine, **glXQueryVersion** subroutine, **glXQueryExtensionsString** subroutine, **glXImportContextEXT** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXGetClientString Subroutine

Purpose

Returns a string describing the client

Library

C bindings library: **libGL.a**

C Syntax

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

Description

The **glXGetClientString** subroutine returns a string describing some aspect of the client library. The possible values for *name* are **GLX_VENDOR**, **GLX_VERSION**, and **GLX_EXTENSIONS**. If *name* is not set to one of these values, **glXGetClientString** returns NULL. The format and contents of the vendor string is implementation dependent.

The extensions string is null-terminated and contains a space-separated list of extension names. (The extension names never contain spaces.) If there are no extensions to GLX, then the empty string is returned.

The version string is laid out as follows:

<*major_version . minor_version*> <*space*> <*vendor-specific_info*>

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

Parameters

dpy	Specifies the connection to the X server.
name	Specifies which string is returned. One of **GLX_VENDOR**, **GLX_VERSION**, or **GLX_EXTENSIONS**.

Notes

The **glXGetClientString** subroutine is available only if the GLX version is 1.1 or greater.

If the GLX version is 1.1 or 1.0, the GL version must be 1.0. If the GLX version is 1.2, then the GL version must be 1.1.

The **glXGetClientString** subroutine only returns information about GLX extensions supported by the client. Call **glGetString** to get a list of GL extensions supported by the server.

Related Information

The **glXQueryVersion** subroutine, **glXQueryExtensionsString** subroutine, **glXQueryServerString** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXGetConfig Subroutine

Purpose

Returns information about GLX visuals.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
int glXGetConfig(Display * dpy
                 XVisualInfo * Visual
                 int  Attribute
                 int * Value)
```

Description

The **glXGetConfig** subroutine sets the *Value* provided to the *Attribute* value of the windows or GLX pixmaps created with respect to the *Visual* parameter. The **glXGetConfig** subroutine returns an error code if for any reason it is unsuccessful. If it is successful, 0 (zero) is returned.

The *Attribute* parameter can be one of the following:

Attribute	Definition
GLX_USE_GL	This attribute is True if OpenGL rendering is supported by this visual. Otherwise, it is False.
GLX_BUFFER_SIZE	This attribute defines the number of bits per color buffer. For red, green, blue, and alpha (RGBA) visuals, it is the sum of **GLX_RED_SIZE**, **GLX_GREEN_SIZE**, **GLX_BLUE_SIZE**, and **GLX_ALPHA_SIZE**. For color index visuals, this attribute is the size of the color indexes.
GLX_LEVEL	This attribute defines the frame buffer level of the visual. Level 0 is the default frame buffer. Positive levels correspond to frame buffers that overlay the default buffer; negative levels correspond to frame buffers that underlay the default buffer.
GLX_RGBA	This attribute is True if color buffers store RGBA values. It is False if they store color indexes.
GLX_DOUBLEBUFFER	This attribute is True if color buffers exist in front/back pairs that can be swapped. Otherwise, it is False.
GLX_STEREO	This attribute is True if color buffers exist in left/right pairs. Otherwise, it is False.
GLX_AUX_BUFFERS	This attribute defines the number of auxiliary color buffers available. Zero indicates that no auxiliary color buffers exist.
GLX_RED_SIZE	This attribute defines the number of red bits stored in each color buffer. If **GLX_RGBA** is False, the **GLX_RED_SIZE** attribute is undefined.

Attribute	Definition
GLX_GREEN_SIZE	This attribute defines the number of green bits stored in each color buffer. If **GLX_RGBA** is False, the **GLX_GREEN_SIZE** attribute is undefined.
GLX_BLUE_SIZE	This attribute defines the number of blue bits stored in each color buffer. If **GLX_RGBA** is False, the **GLX_BLUE_SIZE** attribute is undefined.
GLX_ALPHA_SIZE	This attribute defines the number of alpha bits stored in each color buffer. If **GLX_RGBA** is False, the **GLX_ALPHA_SIZE** attribute is undefined.
GLX_DEPTH_SIZE	This attribute defines the number of bits in the depth buffer.
GLX_STENCIL_SIZE	This attribute defines the number of bits in the stencil buffer.
GLX_ACCUM_RED_SIZE	This attribute defines the number of red bits stored in the accumulation buffer.
GLX_ACCUM_GREEN_SIZE	This attribute defines the number of green bits stored in the accumulation buffer.
GLX_ACCUM_BLUE_SIZE	This attribute defines the number of blue bits stored in the accumulation buffer.
GLX_ACCUM_ALPHA_SIZE	This attribute defines the number of alpha bits stored in the accumulation buffer.
GLX_TRANSPARENT_TYPE_EXT	This attribute defines the type of transparency (if any) in the visual. Return values are: **GLX_NONE_EXT** no transparency **GLX_TRANSPARENT_INDEX_EXT** PseudoColor transparency **GLX_TRANSPARENT_RGB_EXT** RGB Transparency
GLX_TRANSPARENT_RED_VALUE_EXT	This attribute returns the red value of the transparent pixel when the transparency type is **GLX_TRANSPARENT_RGB_EXT**.
GLX_TRANSPARENT_GREEN_VALUE_EXT	This attribute returns the green value of the transparent pixel when the transparency type is **GLX_TRANSPARENT_RGB_EXT**
GLX_TRANSPARENT_BLUE_VALUE_EXT	This attribute returns the blue value of the transparent pixel when the transparency type is **GLX_TRANSPARENT_RGB_EXT**.
GLX_TRANSPARENT_ALPHA_VALUE_EXT	This attribute returns the alpha value of the transparent pixel when the transparency type is **GLX_TRANSPARENT_RGB_EXT**.
GLX_TRANSPARENT_INDEX_VALUE_EXT	This attribute returns the index value of the transparent pixel when the transparency type is **GLX_TRANSPARENT_INDEX_EXT.**
GLX_X_VISUAL_TYPE_EXT	This attribute returns the visual type:

Attribute	Definition
	GLX_TRUE_COLOR_EXT TrueColor colormap **GLX_DIRECT_COLOR_EXT** DirectColor colormap **GLX_PSEUDO_COLOR_EXT** PseudoColor colormap **GLX_STATIC_COLOR_EXT** StaticColor colormap **GLX_GRAY_SCALE_EXT** Grayscale colormap **GLX_STATIC_GRAY_EXT** StaticGray colormap
GLX_VISUAL_CAVEAT_EXT	This attribute returns the visual rating: **0 or GLX_NONE_EXT** no rating **GLX_SLOW_VISUAL_EXT** not an optimal visual

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

Although a GLX implementation can export many visuals that support OpenGL rendering, it *must* support at least two. The first required visual must be an RGBA visual with at least one color buffer, a stencil buffer of at least 1 bit, a depth buffer of at least 12 bits, and an accumulation buffer. Alpha bitplanes are optional in this required visual. However, the color buffer size of this visual must be as great as the deepest **TrueColor**, **DirectColor**, **PseudoColor**, or **StaticColor** visual supported on level 0. The visual itself must also be available on level 0.

The other required visual is a color index one with at least one color buffer, a stencil buffer of at least 1 bit, and a depth buffer of at least 12 bits. This visual must have as many color bitplanes as the deepest **PseudoColor** or **StaticColor** visual supported on level 0. The visual itself must also be available on level 0.

An application is most effective when written to select the visual most closely meeting its requirements. Creating windows or GLX pixmaps with unnecessary buffers can result in reduced rendering performance and poor resource allocation.

Parameters

dpy	Specifies the connection to the X server.
Visual	Specifies the visual to be queried. *Visual* is a pointer to an **XVisualInfo** structure, not a visual ID or a pointer to a **Visual** structure.
Attribute	Specifies the visual attribute to be returned.
Value	Returns the requested value.

Notes

XVisualInfo is defined in the **Xutil.h** file. It is a structure that includes *Visual*, *VisualID*, *Screen*, and *Depth* elements.

Return Values

GLXNoExtension	*dpy* does not support the GLX extension.
GLXBadScreen	The *Visual* screen does not correspond to a valid screen.
GLXBadAttrib	*Attribute* is not a valid GLX attribute.
GLXBadVisual	*Visual* does not support GLX and an attribute other than **GLX_USE_GL** is requested.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXChooseVisual** subroutine, **glXCreateContext** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXGetContextIDEXT Subroutine

Purpose

Gets the XID for a context

Library

C bindings library: **libGL.a**

C Syntax

```
GLXContextID glXGetContextIDEXT(const GLXContext  ctx)
```

Description

The **glXGetContextIDEXT** subroutine returns the XID associated with a GLXContext. No round trip is forced to the server; unlike most X calls that return a value, **glXGetContextIDEXT** does not flush any pending events.

The **glXGetContextIDEXT** subroutine is part of the **EXT_import_context** extension, not part of the core GLX command set. If **GLX_EXT_import_context** is included in the string returned by **glXQueryExtensionsString**, when called with argument **GLX_EXTENSIONS**, extension **EXT_import_context** is supported.

Parameters

ctx	Specifies a GLX rendering context.

Errors

GLXBadContext is generated if *ctx* does not refer to a valid context.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXCreateContext** subroutine, **glXCreateNewContext** subroutine, **glXQueryVersion** subroutine, **glXQueryExtensionsString** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXGetCurrentContext Subroutine

Purpose

Returns the current context.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLXContext glXGetCurrentContext( void )
```

Description

The **glXGetCurrentContext** subroutine returns the current context, as specified by the **glXMakeCurrent** subroutine. If there is no current context, Null is returned.

This subroutine returns client-side information only. It does not make a round-trip to the server.

Return Values

Null	Returned if there is no current context.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXCreateContext** subroutine, **glXCreateNewContext** subroutine, **glXMakeContextCurrent** subroutine, **glXMakeCurrent** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXGetCurrentDisplay Subroutine

Purpose

Gets display for current context

Library

C bindings library: **libGL.a**

C Syntax

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

Description

The **glXGetCurrentDisplay** subroutine returns the display for the current context. If no context is current, NULL is returned.

The **glXGetCurrentDisplay** subroutine returns client-side information. It does not make a round trip to the server, and therefore does not flush any pending events.

Notes

The **glXGetCurrentDisplay** subroutine is only supported if the GLX version is 1.2 or greater.

Related Information

The **glXQueryVersion** subroutine, **glXQueryExtensionsString** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXGetCurrentDrawable Subroutine

Purpose

Returns the current drawable.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLXDrawable glXGetCurrentDrawable( void )
```

Description

The **glXGetCurrentDrawable** subroutine returns the current drawable, as specified by the **glXMakeCurrent** subroutine. If there is no current drawable, None is returned.

This subroutine returns client-side information only. It does not make a round-trip to the server.

Return Values

None	Returned if there is no current drawable.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXCreateGLXPixmap** subroutine, **glXMakeContextCurrent** subroutine, **glXMakeCurrent** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXGetCurrentReadDrawable Subroutine

Purpose

Returns the current read drawable.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLXDrawable glXGetCurrentReadDrawable( void )
```

Description

The **glXGetCurrentReadDrawable** subroutine returns the current read drawable, as specified by the **glXMakeContextCurrent** subroutine. If the **glXMakeCurrent** subroutine is used, then the specified drawable is both the read and write drawable. If there is no current read drawable, None is returned.

This subroutine returns client-side information only. It does not make a round-trip to the server.

Return Values

None	Returned if there is no current read drawable.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXGetCurrentDrawable** subroutine, **glXMakeContextCurrent** subroutine, **glXMakeCurrent** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXGetFBConfigAttrib Subroutine

Purpose

Returns information about GLX FBConfigs.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
int glXGetFBConfigAttrib(Display * dpy,
                         GLXFBConfig  config,
                         int  Attribute,
                         int * Value)
```

Description

The **glXGetFBConfigAttrib** subroutine sets the *Value* provided to the *Attribute* value of the specified GLX FBConfig. The **glXGetFBConfigAttrib** subroutine returns an error code if for any reason it is unsuccessful. If it is successful, 0 (zero) is returned.

The *Attribute* parameter can be one of the following:

Attribute	Definition
GLX_FBCONFIG_ID	This attribute is the XID of the GLX FBConfig.
GLX_VISUAL_ID	This attribute is the XID of the X Visual associated with the GLX FBConfig.
GLX_BUFFER_SIZE	This attribute defines the number of bits per color buffer. For GLX FBConfigs that correspond to a **PseudoColor** or **StaticColor** visual, this is equal to the depth value reported in the X11 visual. For GLX FBConfigs that correspond to **TrueColor** or **DirectColor** visual, this is the sum of **GLX_RED_SIZE**, **GLX_GREEN_SIZE**, **GLX_BLUE_SIZE**, and **GLX_ALPHA_SIZE**.
GLX_LEVEL	This attribute defines the frame buffer level of the visual. Level 0 is the default frame buffer. Positive levels correspond to frame buffers that overlay the default buffer; negative levels correspond to frame buffers that underlay the default buffer.
GLX_DOUBLEBUFFER	This attribute is True if color buffers exist in front/back pairs that can be swapped. Otherwise, it is False.
GLX_STEREO	This attribute is True if color buffers exist in left/right pairs. Otherwise, it is False.
GLX_AUX_BUFFERS	This attribute defines the number of auxiliary color buffers available. Zero indicates that no auxiliary color buffers exist.
GLX_RENDER_TYPE	This attribute indicates what type of GLX Context a drawable created with the corresponding GLX FBConfig can be bound to. The following bit settings can exist: **GLX_RGBA_BIT** RGBA rendering supported. **GLX_COLOR_INDEX_BIT** Color index rendering supported.
GLX_RED_SIZE	This attribute defines the number of red bits stored in each color buffer. If the **GLX_RGBA_BIT** is not set in the **GLX_RENDER_TYPE** attribute, the **GLX_RED_SIZE** attribute is undefined.

Attribute	Definition
GLX_GREEN_SIZE	This attribute defines the number of green bits stored in each color buffer. If the **GLX_RGBA_BIT** is not set in the **GLX_RENDER_TYPE** attribute, the **GLX_GREEN_SIZE** attribute is undefined.
GLX_BLUE_SIZE	This attribute defines the number of blue bits stored in each color buffer. If the **GLX_RGBA_BIT** is not set in the **GLX_RENDER_TYPE** attribute, the **GLX_BLUE_SIZE** attribute is undefined.
GLX_ALPHA_SIZE	This attribute defines the number of alpha bits stored in each color buffer. If the **GLX_RGBA_BIT** is not set in the **GLX_RENDER_TYPE** attribute, the **GLX_ALPHA_SIZE** attribute is undefined.
GLX_DEPTH_SIZE	This attribute defines the number of bits in the depth buffer.
GLX_STENCIL_SIZE	This attribute defines the number of bits in the stencil buffer.
GLX_ACCUM_RED_SIZE	This attribute defines the number of red bits stored in the accumulation buffer.
GLX_ACCUM_GREEN_SIZE	This attribute defines the number of green bits stored in the accumulation buffer.
GLX_ACCUM_BLUE_SIZE	This attribute defines the number of blue bits stored in the accumulation buffer.
GLX_ACCUM_ALPHA_SIZE	This attribute defines the number of alpha bits stored in the accumulation buffer.
GLX_DRAWABLE_TYPE	This attribute defines which GLX drawables are supported by the GLX FBConfig. The following bit settings can exist: **GLX_WINDOW_BIT** GLX Windows are supported. **GLX_PIXMAP_BIT** GLX Pixmaps are supported. **GLX_PBUFFER_BIT** GLX Pbuffers are supported.
GLX_X_RENDERABLE	This attribute indicates whether X can be used to render into a drawable created with the GLX FBConfig. This attribute is True is the GLX FBConfig supports GLX windows and/or pixmaps, otherwise it is False.
GLX_X_VISUAL_TYPE	This attribute defines the X visual type of the X visual associated with the GLX FBConfig. It can have one of the following values:

Attribute	Definition
	Attribute Value Equivalent X Visual Type **GLX_TRUE_COLOR** **TrueColor** **GLX_DIRECT_COLOR** **DirectColor** **GLX_PSEUDO_COLOR** **PseudoColor** **GLX_STATIC_COLOR** **StaticColor** **GLX_GRAY_SCALE** **GrayScale** **GLX_STATIC_GRAY** **StaticGray** **GLX_X_VISUAL_TYPE** No Associated Visual
GLX_CONFIG_CAVEAT	This attribute defines any problems that the GLX FBConfig may have: **GLX_NONE** No caveats **GLX_SLOW_CONFIG** A drawable with this configuration may run at reduced performance. **GLX_NON_CONFORMANT_CONFIG** A drawable with this configuration will not pass the required OpenGL conformance tests.
GLX_TRANSPARENT_TYPE	This attribute defines the type of transparency (if any) supported by the FBConfig. It can have the following values: **GLX_NONE** No transparency supported **GLX_TRANSPARENT_INDEX** Index Color transparency is supported **GLX_TRANSPARENT_RGB** RGB Transparency is supported
GLX_TRANSPARENT_INDEX_VALUE	This attribute defines the index value of the transparent pixel when the transparency type is **GLX_TRANSPARENT_INDEX.**
GLX_TRANSPARENT_RED_VALUE	This attribute defines the red value of the transparent pixel when the transparency type is **GLX_TRANSPARENT_RGB**.
GLX_TRANSPARENT_GREEN_VALUE	This attribute defines the green value of the transparent pixel when the transparency type is **GLX_TRANSPARENT_RGB**
GLX_TRANSPARENT_BLUE_VALUE	This attribute defines the blue value of the transparent pixel when the transparency type is **GLX_TRANSPARENT_RGB**.
GLX_TRANSPARENT_ALPHA_VALUE	This attribute defines the alpha value of the transparent pixel when the transparency type is **GLX_TRANSPARENT_RGB**.
GLX_MAX_PBUFFER_WIDTH	This attribute defines the maximum width value that can be passed into **glXCreatePbuffer**.
GLX_MAX_PBUFFER_HEIGHT	This attribute defines the maximum height value that can be passed into **glXCreatePbuffer**.

Attribute	Definition
GLX_MAX_PBUFFER_PIXELS	This attribute defines the maximum number of pixels (width times height) for a GLX Pbuffer. It can have a value that is less than the maximum width times the maximum height. Also, the value is static and assumes that no other pbuffers or X resources are contending for the framebuffer memory. Therefore, it may not be possible to allocate a pbuffer of the size given by this attribute.

Although a GLX implementation can export many FBConfigs that support OpenGL rendering, it *must* export at least one FBConfig where the **GLX_RENDER_TYPE** attribute has the **GLX_RGBA_BIT** set and the **GLX_CONFIG_CAVEAT** must not be set to **GLX_NON_CONFORMANT_CONFIG**, Also, this FBConfig just have at least one color buffer, a stencil buffer of at least 1 bit, a depth buffer of at least 12 bits and an accumulation buffer. Auxillary buffers are optional and the alpha buffer may have 0 bits. The color buffer size of this FBConfig must be as large as that of the deepest **TrueColor**, **DirectColor**, **PseudoColor**, or **StaticColor** visual supported on framebuffer level 0.

An application is most effective when written to select the GLX FBConfig that most closely meeting its requirements. Creating GLX drawables with unnecessary buffers can result in reduced rendering performance and poor resource allocation.

Parameters

dpy	Specifies the connection to the X server.
config	Specifies the GLX FBConfig to be queried.
Attribute	Specifies the GLX FBConfig attribute to be returned.
Value	Returns the requested value.

Return Values

GLX_NO_EXTENSION	*dpy* does not support the GLX extension.
GLX_BAD_ATTRIBUTE	*Attribute* is not a valid GLX attribute.
GLX_BAD_VALUE	*config* is not a valid GLX FBConfig.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXChooseFBConfig** subroutine, **glXCreateNewContext** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXGetFBConfigs Subroutine

Purpose

Returns a list of all GLX FBConfigs for a specified screen.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
GLXFBConfig *glXGetFBConfigs(Display * dpy
                             int  screen
                             int * nelements)
```

Description

The **glXGetFBConfigs** subroutine returns a list of all GLX FBConfigs that are available for a specified screen. The items in the list can be used in any GLX subroutine that requires a GLX FBConfig. If an error occurs, then a NULL value will be returned.

Parameters

dpy	Specifies the connection to the X server.
screen	Specifies the screen to use to get the list of GLX FBConfigs.
nelements	Returns the number of GLX FBConfigs in the list. If there is an error then it has an undefined value.

Return Values

GLXNoExtension	*dpy* does not support the GLX extension.
GLXBadScreen	The *screen* parameter does not correspond to a valid screen.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXChooseFBConfig** subroutine and the **glXGetFBConfigAttrib** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXGetProcAddressARB Subroutine

Purpose

Returns the address of a GLX or OpenGL subroutine, given the subroutine name.

Library

OpenGL C bindings library: (**libGL.a**)

C Syntax

```
void *glXGetProcAddressARB(const GLubyte  *procName)
```

Description

The **glXGetProcAddressARB** subroutine returns the address of a GLX or OpenGL subroutine, given the subroutine name. This is useful in heterogenous implementations where hardware drivers may implement subroutines not known to the link library.

The pointer returned must be cast exactly as the given subroutine is defined in the **gl.h** or **glx.h** files or the OpenGL specification. Failure to do so can lead to unintended results, if and when the returned value is ever used to try to call the given subroutine.

A return value of NULL indicates that the specified subroutine does not exist for the implementation.

A non-NULL return value for **glXGetProcAddressARB** does not guarantee that a subroutine is actually supported at runtime. The client must also query **glGetString**(**GL_EXTENSIONS**) or **glXQueryExtensionsString** to determine if a subroutine is supported by a particular context.

OpenGL and GLX subroutine pointers returned by **glXGetProcAddressARB** are independent of the currently bound context and may be used by any context which supports that subroutine.

glXGetProcAddressARB may be queried for all of the following subroutines:

- All OpenGL and GLX extension subroutines supported by the implementation (whether those extension subroutines are supported by the current context or not).

- All core (non-extension) subroutines in OpenGL and GLX from version 1.0 up to and including the versions of those specifications supported by the implementation, as determined by **glGetString**(**GL_VERSION**) and **glXQueryVersion** queries.

Parameters

procName	is the name of a GLX or OpenGL subroutine.

Return Values

<address>	the address of the named subroutine.
NULL	notification that the subroutine does not exist in this implementation.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXCreateContext** subroutine, the **glXCreateNewContext** subroutine, the **glGetString** subroutine, the **glXQueryVersion** subroutine, the **glXQueryExtensionsString** subroutine.

OpenGL in the AIX windows (GLX) Environment.

glXGetSelectedEvent Subroutine

Purpose

Returns the GLX events that have been selected for GLX drawables.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glXGetSelectedEvent(display * dpy
                         GLXDrawable  draw
                         unsigned long * eventmask)
```

Description

The **glXGetSelectedEvent** subroutine sets the *eventmask* provided to the GLX events that has been selected for *drawable*. See **glXSelectEvent** for a list of event masks.

Parameters

dpy	Specifies the connection to the X server.
drawable	Specifies a GLX window id or a GLX pbuffer id.
eventmask	Returns which GLX events have been selected for *drawable*.

Errors

GLXBadDrawable is generated *draw* is not a GLX window or a GLX pixmap drawable.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXSelectEvent** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXGetVisualFromFBConfig Subroutine

Purpose

Returns XVisualInfo structure for the X visual associated with a specified GLX FBConfig.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

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

Description

The **glXGetVisualFromFBConfig** subroutine returns an XVisualInfo structure for the X visual associated with a specified GLX FBConfig, if one exists. If the GLX FBConfig does not have an associated X visual or if an error occurs (due to an invalid GLX FBConfig) then NULL is returned.

The data in the returned XVisualInfo structure can be used to create X drawables that will be be needed to create GLX drawables. Use **XFree** to free the returned data.

Parameters

dpy	Specifies the connection to the X server.
config	Specifies the GLX FBConfig to be used.

Notes

XVisualInfo is defined in the **Xutil.h** file. It is a structure that includes *Visual*, *VisualID*, *Screen*, and *Depth* elements.

Return Values

NULL	is returned if *config* does not have an associated X visual or if *config* is not a valid GLX FBConfig.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXChooseFBConfig** subroutine, the **glXGetFBConfigs** subroutine and the **glXGetFBConfigAttrib** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXImportContextEXT Subroutine

Purpose

Imports another process's indirect rendering context.

Library

C bindings library: **libGL.a**

C Syntax

```
GLXContext glXImportContextEXT(Display  *dpy,
                               GLXContextID  contextID)
```

Description

The **glXImportContextEXT** subroutine creates a GLXContext given the XID of an existing GLXContext. It may be used in place of **glXCreateContext**, to share another process's indirect rendering context.

Only the server-side context information can be shared between X clients; client-side state, such as pixel storage modes, cannot be shared. Thus, **glXImportContextEXT** must allocate memory to store client-side information. This memory is freed by calling **glXFreeContextEXT**.

This call does not create a new XID. It merely makes an existing object available to the importing client (`Display *`). Like any XID, it goes away when the creating client drops its connection or the ID is explicitly deleted. Note that this is when the XID goes away. The object goes away when the XID goes away AND the context is not current to any thread.

If *`contextID`* refers to a direct rendering context then no error is generated but **glXImportContextEXT** returns `NULL`.

The **glXImportContextEXT** subroutine is part of the **`EXT_import_context`** extension, not part of the core GLX command set. If **`GLX_EXT_import_context`** is included in the string returned by **glXQueryExtensionsString**, when called with argument **`GLX_EXTENSIONS`**, extension **`EXT_import_context`** is supported.

Parameters

dpy	Specifies the connection to the X server.
contextID	Specifies a GLX rendering context.

Errors

GLXBadContext is generated if *`contextID`* does not refer to a valid context.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXCreateContext** subroutine, **glXCreateNewContext** subroutine, **glXQueryVersion** subroutine, **glXQueryExtensionsString** subroutine, **glXGetContextIDEXT** subroutine, **glXFreeContextEXT** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXIsDirect Subroutine

Purpose

Indicates whether direct rendering is enabled.

Library

`OpenGL C bindings library:` **`libGL.a`**

C Syntax

```
Bool glXIsDirect(Display * dpy
                 GLXContext  Context)
```

Description

The **glXIsDirect** subroutine returns a value of True if the *Context* parameter supplied is a direct rendering context. Otherwise, False is returned. Direct rendering contexts bypass the X server and pass rendering commands directly from the address space of the calling process to the rendering system. Nondirect rendering contexts pass all rendering commands to the X server.

Parameters

dpy	Specifies the connection to the X server.
Context	Specifies the GLX context being queried.

Return Values

True	Returned if *Context* is a direct rendering context.
False	Returned if *Context* is not a direct rendering context.

Error Codes

GLXBadContext	*Context* is not a valid GLX context.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXCreateContext** subroutine, **glXCreateNewContext** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

---

glXMakeContextCurrent Subroutine

Purpose

Attaches a GLX context to one or more GLX drawables.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

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

Description

The **glXMakeContextCurrent** subroutine does two things: (1) it makes the specified *context* parameter the current GLX rendering context of the calling thread, replacing the previously current context if one exists, and (2) it attaches *context* to the GLX drawables *draw* and *read*. As a result of these two actions, subsequent OpenGL rendering calls use *context* as a rendering context to modify the *draw* and *read* GLX drawables. Since the **glXMakeContextCurrent** subroutine always replaces the current rendering context with the specified *context*, there can be only one current context per thread.

draw is used for all OpenGL operations except:

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

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

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

The first time *context* is made current to any thread, its viewport is set to the full size of *draw*. Subsequent calls by any thread to the **glXMakeContextCurrent** subroutine using *context* have no effect on its viewport.

To release the current context without assigning a new one, call the **glXMakeContextCurrent** subroutine with the *draw*, *read* and *context* parameters set to None, None and Null, respectively.

The **glXMakeContextCurrent** subroutine returns True if it is successful, False otherwise. If False is returned, the previously current rendering context and drawables (if any) remain unchanged.

Parameters

dpy	Specifies the connection to the X server.
draw	Specifies a GLX drawable to be used for draw operations. This value must reflect either a GLX window ID, a GLX pixmap ID or a GLX pbuffer ID.
read	Specifies a GLX drawable to be used for read operations. This value must reflect either a GLX window ID, a GLX pixmap ID or a GLX pbuffer ID.
context	Specifies a GLX rendering context to be attached to the specified *draw* and *read* GLX drawables.

Return Values

True	Returned if the **glXMakeContextCurrent** subroutine is successful.
False	Returned if the **glXMakeContextCurrent** subroutine is not successful. The previously current rendering context and drawable (if any) remain unchanged.

Notes

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

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

Error Codes

BadAccess	*context* is current to another thread at the time that the **glXMakeContextCurrent** subroutine is called.
BadAlloc	The server has delayed allocation of ancillary buffers until **glXMakeContextCurrent** is called, only to find that it has insufficient resources to complete the allocation.
BadMatch	It is generated by a number of conditions: • *draw* or *read* is not compatible with *context*. • *draw* or *read* is a GLX Pixmap. and *context* is a direct rendering context. • *draw* and *read* are None and *context* is not None. • This may be generated for implementation-specific reasons.
GLXBadContext	The specified *context* is not a valid GLX context.
GLXBadContextState	The rendering context current to the calling thread has an OpenGL renderer state of **GL_FEEDBACK** or **GL_SELECT**.
GLXBadCurrentDrawable	Pending OpenGL commands exist for the previous context, and the previous *draw* or *read* is no longer valid.
GLXBadDrawable	*draw* or *read* is not a valid GLX drawable.
GLXBadWindow	The X window underlying either *draw* or *read* is no longer valid.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glCopyPixels** subroutine, **glReadPixels** subroutine, **glXCreateNewContext** subroutine, **glXCreatePbuffer** subroutine, **glXCreatePixmap** subroutine, **glXCreateWindow** subroutine, and **glXMakeCurrent** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

---

glXMakeCurrent Subroutine

Purpose

Attaches a GLX context to a window or GLX pixmap.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
Bool glXMakeCurrent(Display * dpy,
                    GLXDrawable  Drawable,
                    GLXContext  Context)
```

Description

The **glXMakeCurrent** subroutine does two things: (1) it makes the specified *Context* parameter the current GLX rendering context of the calling thread, replacing the previously current context if one exists, and (2) it attaches *Context* to a GLX drawable (either a window or GLX pixmap). As a result of these two actions, subsequent OpenGL rendering calls use *Context* as a rendering context to modify the *Drawable* GLX drawable. Since the **glXMakeCurrent** subroutine always replaces the current rendering context with the specified *Context*, there can be only one current context per thread.

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

The first time *Context* is made current to any thread, its viewport is set to the full size of *Drawable*. Subsequent calls by any thread to the **glXMakeCurrent** subroutine using *Context* have no effect on its viewport.

To release the current context without assigning a new one, call the **glXMakeCurrent** subroutine with the *Drawable* and *Context* parameters set to None and Null, respectively.

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

Parameters

dpy	Specifies the connection to the X server.
Drawable	Specifies a GLX drawable. This value must reflect either an X window ID or a GLX pixmap ID.
Context	Specifies a GLX rendering context to be attached to the specified *Drawable*.

Return Values

True	Returned if the **glXMakeCurrent** subroutine is successful.
False	Returned if the **glXMakeCurrent** subroutine is not successful. The previously current rendering context and drawable (if any) remain unchanged.

Notes

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

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

Error Codes

BadMatch	The specified *Drawable* was not created with the same X screen and visual as *Context*. It is also generated if *Drawable* is None and *Context* is not None.
BadAccess	*Context* is current to another thread at the time that the **glXMakeCurrent** subroutine is called.
GLXBadDrawable	The specified *Drawable* is not a valid GLX drawable.
GLXBadContext	The specified *Context* is not a valid GLX context.
GLXBadContextState	The rendering context current to the calling thread has an OpenGL renderer state of **GL_FEEDBACK** or **GL_SELECT**.
GLXBadCurrentWindow	Pending OpenGL commands exist for the previous context, and the current drawable is a window that is no longer valid.
BadAlloc	The server has delayed allocation of ancillary buffers until **glXMakeCurrent** is called, only to find that it has insufficient resources to complete the allocation.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXCreateContext** subroutine, **glXCreateNewContext** subroutine, **glXCreateGLXPixmap** subroutine, **glXMakeContextCurrent** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXQueryContext Subroutine

Purpose

Queries context information

Library

C bindings library: **libGL.a**

C Syntax

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

Description

The **glXQueryContext** subroutine sets the *value* provided to the *attribute* value of the specified GLX Context. The **glXQueryContext** returns an error code if for any reason it is unsuccessful. If it is successful, 0 (zero) is returned.

The *attribute* parameter can be one of the following:

Attribute	Description
GLX_FBCONFIG_ID	This attribute is the XID of the GLX FBConfig associated with the GLX Context.
GLX_RENDER_TYPE	This attribute is the type of rendering supported by the GLX Context.
GLX_SCREEN	This attribute is the number of the screen associated with the GLX Context.

Parameters

dpy	Specifies the connection to the X server.
ctx	Specifies a GLX rendering context.
attribute	Specifies the GLX Context attribute to be returned.
value	Returns the requested value.

Errors

GLX_BAD_ATTRIBUTE	Is generated if *attribute* is not a valid GLX Context attribute.
GLXBadContext	Is generated if *ctx* does not refer to a valid context and a round trip to the X server is involved.

Related Information

The **glXCreateContext** subroutine, **glXCreateNewContext** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

---

glXQueryContextInfoEXT Subroutine

Purpose

Queries context information

Library

C bindings library: **libGL.a**

C Syntax

```
int glXQueryContextInfoEXT(Display  *dpy,
                           GLXContext  ctx,
                           int  attribute,
                           int  *value)
```

Description

The **glXQueryContextInfoEXT** subroutine returns the the visual id, screen number, and share list of *ctx*. This call may cause a round trip to the server.

The **glXQueryContextInfoEXT** subroutine is part of the **EXT_import_context** extension, not part of the core GLX command set. If **GLX_EXT_import_context** is included in the string returned by **glXQueryExtensionsString**, when called with argument **GLX_EXTENSIONS**, extension **EXT_import_context** is supported.

Parameters

dpy	Specifies the connection to the X server.
ctx	Specifies a GLX rendering context.
attribute	The visual ID that the context was created with.
value	The X screen the the context was created for.

Errors

GLXBadContext is generated if *ctx* does not refer to a valid context.

Related Information

The **glXCreateContext** subroutine, **glXCreateNewContext** subroutine, **glXQueryVersion** subroutine, **glXQueryExtensionsString** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXQueryDrawable Subroutine

Purpose

Returns an attribute associated with a GLX drawable.

Library

C bindings library: **libGL.a**

C Syntax

```
int glXQueryDrawable(Display *  dpy,
                     GLXDrawable  drawable,
                     int  attribute,
                     unsigned int * value)
```

Description

The **glXQueryDrawable** subroutine sets the *value* provided to the *attribute* value of the specified GLX drawable.

The *attribute* parameter can be one of the following:

Attribute	Description
GLX_WIDTH	This attribute is the width of the GLX drawable.
GLX_HEIGHT	This attribute is the height of the GLX drawable.
GLX_PRESERVED_CONTENTS	This attribute is a boolean value that shows whether the contents of the GLX pbuffer is to be preserved when a resource conflict occurs.
GLX_LARGEST_PBUFFER	This attribute is a boolean value that shows whether the largest pbuffer allocation was obtained when the allocation of the pbuffer would have failed.
GLX_FBCONFIG_ID	This attribute is the XID of the GLX FBConfig used when *drawable* was created.

The contents of *value* will be undefined if *drawable* is not a GLX pbuffer and *attribute* is set to **GLX_PRESERVED_CONTENTS** or **GLX_LARGEST_PBUFFER**. The contents of *value* will be 0 (zero) if *attribute* is not one of the attributes listed above.

Parameters

dpy	Specifies the connection to the X server.
drawable	Specifies a GLX drawable ID.
attribute	Specifies the GLX drawable attribute to be returned.
value	Returns the requested value.

Errors

GLXBadDrawable	Is generated if *drawable* does not refer to a valid GLX drawable.

Related Information

The **glXCreatePbuffer** subroutine, **glXCreatePixmap** subroutine, or **glXCreateWindow** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

---

glXQueryExtension Subroutine

Purpose

Indicates whether the GLX extension is supported.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
Bool glXQueryExtension(Display * dpy
                       int * ErrorBase
                       int * EventBase)
```

Description

The **glXQueryExtension** subroutine returns True if the X server of connection *dpy* supports the GLX extension. Otherwise, the subroutine returns False. If True is returned, the *ErrorBase* and *EventBase* parameters also return the error base and event base of the GLX extension, respectively. If the **glXQueryExtension** subroutine returns False, *ErrorBase* and *EventBase* are unchanged.

ErrorBase and *EventBase* do not return values if they are specified as Null.

Notes

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

Parameters

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

Return Values

True	Returned if the X server of connection *dpy* supports the GLX extension.
False	Returned if the X server of connection *dpy* does not support the GLX extension.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXQueryVersion** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXQueryExtensionsString Subroutine

Purpose

Returns a list of supported extensions.

Library

C bindings library: **libGL.a**

C Syntax

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

Description

The **glXQueryExtensionsString** subroutine returns a pointer to a string describing which GLX extensions are supported on the connection. The string is null-terminated and contains a space-separated list of extension names. (The extension names themselves never contain spaces.) If there are no extensions to GLX, then the empty string is returned.

Parameters

dpy	Specifies the connection to the X server.
screen	Specifies the screen.

Notes

The **glXQueryExtensionsString** subroutine is available only if the GLX version is 1.1 or greater.

The **glXQueryExtensionsString** subroutines only returns information about GLX extensions. Call **glGetString** to get a list of GL extensions.

Related Information

The **glGetString** subroutine, **glXQueryVersion** subroutine, **glXQueryServerString** subroutine, **glXGetClientString** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXQueryServerString Subroutine

Purpose

Returns string describing the server

Library

C bindings library: **libGL.a**

C Syntax

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

Description

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

Parameters

dpy	Specifies the connection to the X server.
screen	Specifies the screen number.
name	Specifies which string is returned. One of **GLX_VENDOR**, **GLX_VERSION**, or **GLX_EXTENSIONS**.

Notes

The **glXQueryServerString** subroutine is available only if the GLX version is 1.1 or greater.

If the GLX version is 1.1 or 1.0, the GL version must be 1.0. If the GLX version is 1.2, the GL version must be 1.1.

The **glXQueryServerString** subroutine only returns information about GLX extensions supported by the server. Call **glGetString** to get a list of GL extensions. Call **glXGetClientString** to get a list of GLX extensions supported by the client.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXQueryVersion** subroutine, **glXGetClientString** subroutine, **glXQueryExtensionsString** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXQueryVersion Subroutine

Purpose

Returns the version numbers of the GLX extension.

Library

`OpenGL C bindings library:` **`libGL.a`**

C Syntax

```
Bool glXQueryVersion(Display * dpy
                     int * Major
                     int * Minor)
```

Description

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

The *Major* and *Minor* parameters do not return values if they are specified as Null.

Parameters

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

Return Values

True	Returned if the subroutine is successful.
False	Returned if the subroutine fails. If False is returned, *Major* and *Minor* parameter values are not updated.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXQueryExtension** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

---

glXSelectEvent Subroutine

Purpose

Requests that a GLX drawable receive GLX events.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glXSelectEvent(Display * dpy,
                    GLXDrawable  drawable,
                    unsigned long  eventmask)
```

Description

The **glXSelectEvent** subroutine is used to allow the application to receive GLX events for a specified GLX drawable (*drawable*). Calling **glXSelectEvent** overrides any previous event mask that was set by the application for *drawable*.

GLX events are returned in the X11 event stream. The GLX event mask is private to GLX (it is separate from the X11 event mask) and a separate GLX event mask is maintained in the server for each client for each drawable.

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

typedef struct {

int	*event_type*;	/* This will have a value of **GLX_DAMAGED** or **GLX_SAVE** */
int	*draw_type*;	/* This will have a value of **GLX_WINDOW** or **GLX_PBUFFER** */
unsigned long	*serial*;	/* Number of last request processed by X server */
Bool	*send_event*;	/* Whether the event was generated by a SendEvent request */
Display *	*display*;	/* The display that the event was read from */
GLXDrawable	*drawable*;	/* XID of the GLX drawable */
unsigned int	*buffer_mask*;	/* Mask indicating which buffers are affected. */
unsigned int	*aux_buffer*;	/* Mask indicating which aux buffer was affected */
int	*x*, *y*;	/* Location of the area clobbered in the GLX drawable */
int	*width*, *height*;	/* Size of the area clobbered in the GLX drawable */
int	*count*;	/* If non-zero, at least this many more events exist */

} GLX_PbufferClobberEvent;

The masks that represent the clobbered buffers are defined as:

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

A single X server operation can cause several pbuffer clobber events to be sent. Each event specifies one region of the GLX drawable that was affected by the operation. *buffer_mask* indicates which color or ancillary buffers were affected. When the **GLX_AUX_BUFFERS_BIT** is set in *buffer_mask*, then *aux_buffer* is set to indicate which buffer was affected. If more than one aux buffer was affected then additional events are generated. For non-stereo drawables, **GLX_FRONT_LEFT_BUFFER_BIT** and **GLX_BACK_LEFT_BUFFER_BIT** are used to specify the front and back color buffers.

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

For an unpreserved pbuffer, a pbuffer clobber event, that has *event_type* set to **GLX_DAMAGED**, is generated whenever a portion of the pbuffer becomes invalid.

Parameters

dpy	Specifies the connection to the X server.
drawable	Specifies a GLX window or GLX pbuffer.
eventmask	Specifies the GLX events that *drawable* will receive.

Error Codes

GLXBadDrawable	Is generated if *drawable* is not a valid GLX window or GLX pbuffer.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXCreatePbuffer** subroutine, **glXCreateWindow** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXSwapBuffers Subroutine

Purpose

Makes the back buffer visible.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glXSwapBuffers(Display * dpy
                    GLXDrawable  Drawable)
```

Description

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

An implicit **glFlush** subroutine is performed by the **glXSwapBuffers** subroutine before it returns. Subsequent OpenGL commands can be issued immediately after calling **glXSwapBuffers**, but these commands are not executed until the buffer exchange is complete.

If *Drawable* was not created with respect to a double-buffered visual or GLX FBConfig, or if *Drawable* is a GLX pixmap, the **glXSwapBuffers** subroutine has no effect and no error is generated.

Parameters

dpy	Specifies the connection to the X server.
Drawable	Specifies the window whose buffers are to be swapped.

Notes

Synchronization between multiple GLX contexts that render to the same double-buffered window is the responsibility of the client. The X Synchronization Extension can be used to facilitate this cooperation.

Error Codes

GLXBadCurrentDrawable	Is generated if *dpy* and *Drawable* are respectively the display and drawable associated with the current context of the calling thread, and *Drawable* identifies a window that is no longer valid.
GLXBadDrawable	Is generated if *Drawable* is not a valid GLX drawable.
GLXBadWindow	Is generated if the X window underlying *Drawable* is no longer valid.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glFlush** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

---

glXUseXFont Subroutine

Purpose

Creates bitmap display lists from an X font.

Library

`OpenGL C bindings library:` **`libGL.a`**

C Syntax

```
void glXUseXFont(Font  font
                 int  First
                 int  Count
                 int  ListBase)
```

Description

The **glXUseXFont** subroutine generates *Count* display lists, with each containing a single **glBitmap** command. These lists are named *ListBase* through *ListBase*+*Count*-**1**. The parameters of the **glBitmap** command of display list *ListBase*+*i* are derived from glyph *first*+*i*. Bitmap parameters *Xorig*, *Yorig*, *Width*, and *Height* are computed from font metrics as *descent*-1, -*lbearing*, *rbearing*-*lbearing*, and *ascent*+*descent*, respectively. *Xmove* is taken from the glyph's *Width* metric, and *Ymove* is set to 0 (zero). Finally, the glyph's image is converted to the appropriate format for the **glBitmap** command.

Using the **glXUseXFont** subroutine may be more efficient than accessing the X font and generating the display lists explicitly, since display lists are created on the server without requiring the glyph data to make a round-trip. Also, the server may choose to delay the creation of each bitmap until it is accessed.

Empty display lists are created for all glyphs that are requested but not defined in the *Font* parameter.

The **glXUseXFont** subroutine is ignored if there is no current GLX context.

Parameters

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

Error Codes

BadFont	Is generated if *font* is not a valid font.
GLXBadContextState	Is generated if the current GLX context is in display-list construction mode.
GLXBadCurrentWindow	Is generated if the drawable associated with the current context of the calling thread is a window, and that window is no longer valid.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glXMakeContextCurrent** subroutine, **glXMakeCurrent** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

glXWaitGL Subroutine

Purpose

Completes OpenGL processing prior to subsequent X calls.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glXWaitGL( void )
```

Description

The **glXWaitGL** subroutine ensures that OpenGL processing is complete before any subsequent X calls are processed. Any OpenGL rendering calls made prior to the **glXWaitGL** subroutine are completed before any X rendering calls made after **glXWaitGL**. Although this same result can be achieved using **glFinish**, the **glXWaitGL** subroutine does not require a round-trip to the server. Therefore, **glXWaitGL** is more efficient in cases where the client and server are on separate machines.

The **glXWaitGL** subroutine is ignored if there is no current GLX context.

Notes

Using the **glXWaitGL** subroutine may or may not flush the X stream.

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glFinish** subroutine, **glFlush** subroutine, **glXWaitX** subroutine.

The **XSync** function.

OpenGL in the AIXwindows (GLX) Environment.

glXWaitX Subroutine

Purpose

Completes X processing prior to subsequent OpenGL calls.

Library

OpenGL C bindings library: **libGL.a**

C Syntax

```
void glXWaitX( void )
```

Description

The **glXWaitX** subroutine ensures that X processing is complete before any subsequent OpenGL rendering calls are processed. Any X rendering calls made prior to the **glXWaitX** subroutine are completed before any OpenGL rendering calls made after **glXWaitX**. Although this same result can be achieved using **XSync**, the **glXWaitX** subroutine does not require a round-trip to the server. Therefore, **glXWaitX** is more efficient in cases where the client and server are on separate machines.

The **glXWaitX** subroutine is ignored if there is no current GLX context.

Notes

Using the **glXWaitX** subroutine may or may not flush the OpenGL stream.

Error Codes

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

Files

/usr/include/GL/gl.h	Contains C language constants, variable type definitions, and ANSI function prototypes for OpenGL.
/usr/include/GL/glx.h	Contains C language constants, variable type definitions, and ANSI function prototypes for GLX.

Related Information

The **glFinish** subroutine, **glFlush** subroutine, **glXWaitGL** subroutine.

The **XSync** subroutine.

OpenGL in the AIXwindows (GLX) Environment.

Chapter 4. OpenGL Drawing Widgets and Related Functions

Following is a list of the OpenGL widgets and widget-related functions, and the purpose of each. Select the item to receive more information.

GLwCreateMDrawingArea function	Creates an instance of a **GLwMDrawingArea** widget and returns the associated widget ID.
GLwDrawingArea widget	Provides a window with the appropriate visuals and color maps needed for OpenGL drawing. (Xt)
GLwMDrawingArea widget	Provides a window with the appropriate visuals and color maps needed for OpenGL drawing. (Motif)
GLwDrawingAreaMakeCurrent function	Provides a front end to the **glXMakeCurrent** subroutine.
GLwDrawingAreaSwapBuffers function	Provides a front end to the **glXSwapBuffers** subroutine.

GLwCreateMDrawingArea Function

Purpose

Creates an instance of a **GLwMDrawingArea** widget and returns the associated widget ID.

Library

OpenGL C bindings library: **libXGLW.a**

C Syntax

```
#include <X11/GLW/GLwMDraw.h>

Widget GLwCreateMDrawingArea( Parent,  Name,
           ArgumentList,  ArgumentCount)
   Widget    Parent;
   String    Name;
   ArgList    ArgumentList;
   Cardinal    ArgumentCount;
```

Description

The **GLwCreateMDrawingArea** function creates an instance of a **GLwMDrawingArea** widget and returns the associated widget ID. For a complete definition of **GLwMDrawingArea** and its associated resources, see the **GLwMDrawingArea** widget.

Parameters

Parent	Specifies the parent widget ID name.
Name	Specifies the name of the created widget.
ArgumentList	Specifies the argument list.
ArgumentCount	Specifies the number of attribute/value pairs in the *ArgumentList* parameter.

Files

/usr/include/GL/GLwDrawA.h	Contains the **GLwDrawingArea** widget definitions derived from the Xt.

© Copyright IBM Corp. 1994, 2002

/usr/include/GL/GLwDrawAP.h	Contains **GLwDrawingArea** widget private definitions.
/usr/include/GL/GLwMDrawA.h	Contains the **GLwMDrawingArea** widget definitions derived from Motif.
/usr/include/GL/GLwMDrawAP.h	Contains **GLwMDrawingArea** widget private definitions.

Related Information

The **GLwDrawingArea** or **GLwMDrawingArea** widget.

GLwDrawingArea or GLwMDrawingArea Widget

Purpose

OpenGL Draw Widget Class

Library

OpenGLC bindings library: **libXGLW.a**

C Syntax

```
#include </usr/include/GL/GLwDrawA.h>

Widget = XtCreateWidget(Widget, glwDrawingAreaWidgetClass, ...);
ld ... -lXGLW -l<anywidgetlibrary> -lXt -lGL -lX11 ...

#include </usr/include/GL/GLwMDrawA.h>

Widget = XtCreateWidget(Widget, glwMDrawingAreaWidgetClass, ...);
ld ... -lXGLW -lXm -IXt -IGL -IX11 ...
```

Description

GLwDrawingArea and **GLwMDrawingArea** are widgets suitable for OpenGL drawing. Based on supplied parameters, these widgets provide a window with the appropriate visual and color maps needed for OpenGL. The **GLwDrawingArea** and **GLwMDrawingArea** widgets also provide callbacks for redraw, resize, input, and initialization.

The **GLwDrawingArea** widget is not a part of any widget set, but depends only on the Intrinsics Library (Xt). **GLwDrawingArea** can be used with any widget set. The **GLwMDrawingArea** widget is identical to the **GLwDrawingArea** widget except that it is a subclass of the Motif **XmPrimitive** widget class and has resources and defaults suitable for use with Motif. For example, **GLwMDrawingArea** provides the default Motif background and foreground colors for resources, and handles keyboard traversal more efficiently. Although the **GLwDrawingArea** widget can be used in a Motif program, it is recommended that **GLwMDrawingArea** be used instead.

Because both **GLwDrawingArea** and **GLwMDrawingArea** widgets behave almost identically, the remainder of this article refers only to **GLwDrawingArea**, except when the behaviors differ. Unless explicitly stated, all statements about **GLwDrawingArea** also apply to **GLwMDrawingArea**.

Among the information the programmer must provide to create a **GLwDrawingArea** widget is information necessary to determine the visual. The programmer can provide this information through resources by using one of the following methods:

- Supply a specific visualInfo structure. This visualInfo must have been obtained elsewhere; it is the application designer's responsibility to ensure that it is compatible with the OpenGL rendering done by the application.
- Provide an attribute list, which is formatted identically to that used for direct OpenGL programming.
- Specify each attribute as an individual resource. The method is the simplest, and is the only method that works from resource files.

In addition to allocating the visual, the **GLwDrawingArea** widget also allocates the color map unless one is provided by the application.

> **Note:** If the color map is provided by the application, the application writer is responsible for guaranteeing compatibility between the color map and the visual.

If an application creates multiple **GLwDrawingArea** widgets in the same visual, the same color map will be used. However, the color map will not be shared among separate applications.

Creating the widget does not actually create the window until it is realized, and consequently, the application should not perform any OpenGL operations to the window immediately after creation. Instead, the application must wait until after it has realized the window. Alternatively, the **ginit** callback may be used to indicate when the window has been created. Upon receiving this callback, the application can perform all OpenGL initialization for the window, and can subsequently perform other operations on it. The initialization is discussed in more detail in the following sections.

Applications select which **GLwDrawingArea** they are accessing using either the **glXMakeCurrent** subroutine or the convenience function **GLwDrawingAreaMakeCurrent**, which uses a widget instead of a display and window. If there is only one **GLwDrawingArea**, this need only be called once; however, if there is more than one **GLwDrawingArea**, the widget should be called at the start of each callback. Callbacks in this case include not only callbacks provided by the widget itself, but any other callback that leads to Graphics Library (GL) activity, such as a timeout or a workproc.

If an application is using double buffering, it may call **GLwDrawingAreaSwapBuffers** instead of **glXSwapBuffers**. This allows the use of the widget instead of the display and window.

The **GLwDrawingArea** widget class is subclassed from the **Core** class, and inherits behavior and resources from the **Core** class. The **GLwDrawingArea** widget has the following class information:

Class Pointer: **GLwDrawingAreaClass**
Class Name: **GLwDrawingArea**

The **GLwMDrawingArea** widget class is subclassed from the **XmPrimitive** class, and inherits behavior and resources from the **XmPrimitive** and **Core** classes.

Class Pointer: **GLwMDrawingAreaClass**
Class Name: **GLwMDrawingArea**

New Resources

The following table defines a set of widget resources used by the programmer to specify data. The programmer can also set the resource values for the inherited classes to set attributes for this widget. To reference a resource by name or by class in an **.Xdefaults** file, remove the GLwN or GLwC prefix and use the remaining letters. There are two tables included. The following table includes resources that correspond directly to the attributes used by the **glXChooseVisual** subroutine. As with **glXChooseVisual**, all Boolean resources default to False and all integer resources default to 0. These resources can all be set only at creation time, and are used to determine the visual. If either the **GLwNattribList** or **GLwNvisualInfo** resource is set, these resources are ignored. The specific meaning of these resources is discussed in the **glXChooseVisual** subroutine and will not be discussed here.

Name	Class	Type	OpenGL Attribute
GLwNbufferSize	GLwCBufferSize	Integer	GLX_BUFFER_SIZE
GLwNlevel	GLwCLevel	Integer	GLX_LEVEL
GLwNrgba	GLwCRgba	Integer	GLX_RGBA

Name	Class	Type	OpenGL Attribute
GLwNdoublebuffer	GLwCDoublebuffer	Boolean	GLX_DOUBLE- BUFFER
GLwNstereo	GLwCStereo	Boolean	GLX_STEREO
GLwNauxBuffers	GLwCAuxBuffers	Boolean	GLX_AUX _BUFFERS
GLwNredSize	GLwCColorSize	Integer	GLX_RED_SIZE
GLwNgreenSize	GLwCColorSize	Integer	GLX_GREEN_SIZE
GLwNblueSize	GLwCColorSize	Integer	GLX_BLUE_SIZE
GLwNalphaSize	GLwCAlphaSize	Integer	GLX_ALPHA_SIZE
GLwNdepthSize	GLwCDepthSize	Integer	GLX_DEPTH_SIZE
GLwNstencilSize	GLwCStencilSize	Integer	GLX_ STENCIL_SIZE
GLwNaccum- RedSize	GLwCAccum- ColorSize	Integer	GLX_ACCUM _RED_SIZE
GLwNaccum- GreenSize	GLwCAccum- ColorSize	Integer	GLX_ACCUM _GREEN_SIZE
GLwNaccum- BlueSize	GLwCAccum- ColorSize	Integer	GLX_ACCUM _BLUE_SIZE
GLwNaccum- AlphaSize	GLwCAccum- AlphaSize	Integer	GLX_ACCUM _ALPHA_SIZE

The following table lists other resources of the **GLwDrawingArea** widget. Following the table is a description of each resource. The codes in the access column indicate if the given resource can be set at creation time (C), set by using **XtSetValues** (S), retrieved by using **XtGetValues** (G), or is not applicable (N/A).

Name	Class	Type	Default	Access
GLwNallocate- Background	GLwCAllocate- Colors	Boolean	False	CG
GLwNallocate- OtherColors	GLwCAllocate- Colors	Boolean	False	CG
GLwNattribList	GLwCAttribList	Integer *	NULL	CG
GLwNexpose- Callback	GLwCCallback	XtCallbackList	NULL	C
GLwNginit- Callback	GLwCCallback	XtCallbackList	NULL	C
GLwNinput- Callback	GLwCCallback	XtCallbackList	NULL	C
GLwNinstall- Background	GLwCInstall- Background	Boolean	True	CG
GLwNinstall- Colormap	GLwCInstall- Colormap	Boolean	True	CG
GLwNresize- Callback	GLwCCallback	XtCallbackList	NULL	C
GLwNvisual- Info	GLwCVisual- Info	XVisualInfo*	NULL	CG

GLwNallocateBackground
If True, the background pixel and pixmap are allocated (if appropriate) using the newly calculated color map and visual. If False, they retain values calculated using the parent's color map and visual. Applications that wish to have X clear their background for them will usually set this to True. Applications clearing their own background will often set this to False, although they may set this to True if they query the background for their own use. One reason to leave this resource False is that if color index mode is in use, this will avoid using up a pixel from the newly allocated color map. Also, on hardware that supports only one color map, the application may need to do more careful color allocation to avoid flashing between the OpenGL color map and the default X color map.

> **Note:** Because of the way the Intrinsics Library (Xt) works, the background colors are originally calculated using the default color map; if this resource is set they can be recalculated correctly. If a color map was explicitly supplied to the widget rather than being dynamically calculated, these resources are always calculated using that color map.)

GLwNallocateOtherColors
This is similar to **GLwNallocateBackground**, but allocates other colors normally allocated by widgets. Although the **GLwDrawingArea** and **GLwMDrawingArea** widget do not make use of these colors the application may choose to query them. For the non-Motif **GLwDrawingArea** widget there are no other colors allocated, so this resource is a no-op. For the Motif **GLwMDrawingArea** are widget, the **XmPrimitive** resources **XmNforeground**, **XmNhighlightColor**, and **XmNhighlightPixmap** are calculated.

GLwNattribList
Contains the list of attributes suitable for a call to **glXChooseVisual**. If this resource is NULL, it is calculated based on the attribute resources. If it is not NULL, the attribute resources are ignored.

GLwNexposeCallback
Specifies the list of callbacks that is called when the widget receives an exposure event. The callback reason is **GLwCR_EXPOSE**. The callback structure also includes the exposure event. The application will generally want to redraw the scene.

GLwNginitCallback
Specifies the list of callbacks that is called when the widget is first realized. Since no OpenGL operations can be done before the widget is realized, this callback can be used to perform any appropriate OpenGL initialization such as creating a context. The callback reason is **GLwCR_GINIT**.

GLwNinputCallback
Specifies the list of callbacks that is called when the widget receives a keyboard or mouse event. By default, the input callback is called on each key press and key release, on each mouse button press and release, and whenever the mouse is moved while a button is pressed. However this can be changed by providing a different translation table. The callback structure also includes the input event. The callback reason is **GLwCR_INPUT**.

The input callback is provided as a programming convenience, since it provides a convenient way to catch all input events. However, a more modular program can often be obtained by providing specific actions and translations in the application rather than by using a single catch-all callback. Use of explicit translations can also provide greater customizing ability.

GLwNinstallBackground
If set to True, the background is installed on the window. If set to False, the window has no background. This resource has no effect unless **GLwNallocateBackground** is also True.

GLwNinstallColormap	If set to True, the widget will call **XSetWMColormapWindows** to tell the window manager to install the color map when the window's shell has focus. If set to False, this will not be called. For applications with multiple **GLwDrawingArea** widgets sharing a single color map, it is most efficient to set this resource to True for exactly one **GLwDrawingArea** with each color map. If an application needs additional control over the order of color maps, this resource can be set to False, with the application calling **XSetWMColormapWindows** explicitly.
GLwNresizeCallback	Specifies the list of callbacks that is called when the **GLwDrawingArea** is resized. The callback reason is **GLwCR_RESIZE**.
GLwNvisualInfo	Contains a pointer to the window's visual info structure. If Null, the visualInfo is calculated at widget creation time based on the **GLwNattributeList** resource (which is itself calculated from the various resources). If **GLwNvisualInfo** is not Null the **GLwNattributList** and the attribute resources are ignored.

Inherited Resources

Both **GLwDrawingArea** and **GLwMDrawingArea** inherit behavior and resources from the **Core** superclass. Other than the behavior of the color map and background resources described previously, all defaults are the same as for **Core**.

In addition, the Motif version **GLwMDrawingArea** also inherits from **XmPrimitive**. The behavior of the color resources has been described previously. The **TraversalOn** resource is disabled for this widget, but if keyboard input is required it should be enabled. (Also, the application should call **XmProcessTraversal**(widget, **XmTRAVERSE_CURRENT**) whenever mouse button 1 is clicked in the widget. This is similar to the requirements of the Motif Drawing area.) Because Motif gets confused by having multiple visuals in one top level shell, **XmNhighlightOnEnter** has been disabled, and **XmNhighlightThickness** has been set to 0.

Callback Information

A pointer to the following structure is passed to each callback:

```
typedef struct
{
   Integer reason;
   XEvent * event;
   Dimension    width, height;
} GLwDrawingAreaCallbackStruct;
```

Reason	Indicates why the callback was invoked. Appropriate values are stated in the previous resource descriptions. For Motif programmers, the values **GLwCR_EXPOSE**, **GLwCR_RESIZE**, and **GLwCR_INPUT** are equal to **XmCR_EXPOSE**, **XmCR_RESIZE**, and **XmCR_INPUT** respectively. **GLwCR_GINIT** does not have a Motif equivalent.
Event	Points to the XEvent that triggered the callback. This is Null for **GLwNginitCallback** and **GLwNresizeCallback**.
Width	Sets the width of the window.
Height	Sets the height of the window.
1	Adds space before the SS.

Translations

The **GLwDrawingArea** widget has the following translations:

<KeyDown>:	glwInput()
<KeyUp>:	glwInput()
<BtnDown>:	glwInput()

<BtnUp>: glwInput()
<BtnMotion>: glwInput()

The **GLwMDrawingArea** widget has the following additional translation:

<Key>osfHelp: PrimitiveHelp()

An application wishing to catch other events than these defaults can do so by installing a different translation table.

1 Adds space before the SS.

Action Routines

The **GLwDrawingArea** widget has the following action routine:

glwInput(): Called whenever one of the previous translations specifies that input has occurred. Its sole purpose is to call the input callback.

Initialization

When the widget is initially created (for example, through **XtCreateWidget**) the associated window is not actually created. Instead, window creation is delayed until the widget is realized. However, **glXchooseVisual** is called immediately, so information based on its results is available.

Between the time the widget is created and it is realized, the following apply:

- No OpenGL operations can be done to the window.
- No resize callbacks are generated.
- The normal window is available (**XtWindow** returns Null).
- The **GLwDrawingAreaMakeCurrent** function (and **glXMakeCurrent** subroutine) should not be called.

When the widget is realized, the following actions take place:

- The window is created.
- The **ginit** callback is called. The user may use this callback to perform any needed OpenGL initialization to the window.

Notes

When using the input callback to receive keyboard input, the keycode in the event must be converted to a keysym. Use **XLookupKeysym** or **XLookupString** to do the conversion. Keyboard input can also be dealt using translations, in which case no such conversion is required.

Motif programmers should keep in mind that OSF uses virtual bindings and replaces some of the key bindings. As a common example, if the Esc key is to be used to exit the program (as it often is in GL programs), the translation should specify <key>osfCancel instead of <key>Escape.

Motif programmers may also create a **GLwMDrawingArea** widget with the Motif style **GLwCreateMDrawingArea**.

Examples

The following are some code fragments that create a **GLwDrawingArea** widget and manage the appropriate callbacks:

```
#include </usr/include/GL/GLwDrawA.h>
   static GLXContext glx_context;  /* assume only one context */
   . . .
   main()
   {
       Arg args[10];
       int n;
       Widget parent;   /* The parent of the gl widget */
       Widget glw;   /* The GLwDrawingArea widget          */
       . . .
       /* Create the widget using RGB mode.  This can also be set
        * in an X Defaults file
        */
       n = 0;
       XtSetArg(args[n], GLwNrgba, TRUE); n++;
       glw = XtCreateManagedWidget("glw",
      GLwDrawingAreaWidgetClass,
      parent, args, n);
       XtAddCallback(glw, GLwNexposeCallback, exposeCB, 0);
       XtAddCallback(glw, GLwNresizeCallback, resizeCB, 0);
       XtAddCallback(glw, GLwNginitCallback, ginitCB, 0);
       /* Also add input callback if needed */
       . . .
   }
   static void
   exposeCB(Widget w, XtPointer client_data,
             GLwDrawingAreaCallbackStruct call_data)
   {
       GLwDrawingAreaMakeCurrent (w, glx_context);
       /* redraw the display */
   }
   static void
   resizeCB(Widget w, XtPointer client_data,
             GLwDrawingAreaCallbackStruct call_data)
   {
       GLwDrawingAreaMakeCurrent (w, glx_context);
       /* perform any resize actions */
        glViewport (0, 0, call_data->width -1,
         call_data->height -1);
      /* redraw the display */
   }
   static void
   ginitCB(Widget w, XtPointer client_data,
             GLwDrawingAreaCallbackStruct call_data)
   {
       Arg args[1];
       XVisualInfo *vi;
       XtSetArg(args[0], GLwNvisualInfo, &vi);
       XtGetValues(w, args, 1);
       /* create a visual context */
       glx_context = glXCreateContext(XtDisplay(w), vi, 0, GL_FALSE);
       GLwDrawingAreaMakeCurrent (w, glx_context);
       /* Perform any necessary graphics initialization.*
```

The Motif program need only differ by including **GLwMDrawA.h** instead of **GLwDrawA.h** and by creating a widget of type **GLwMDrawingAreaWidgetClass** instead of **GLwDrawingAreaWidgetClass**. As an alternative, the Motif program could use **GLwCreateMDraw** instead.

Notes:

1. If a **GLwDrawingArea** widget is created as a child of an already realized widget, the **GLwDrawingArea** widget will be created immediately, without giving the user an opportunity to add the **ginit** callback. In such a case, initialization should be done immediately after creating the widget rather than by using the callback.

2. If the non-Motif **GLwDrawingArea** widget is used in a Motif program and keyboard traversal is attempted, the behavior is undefined if the user traverses into the **GLwDrawingArea** widget.

Files

/usr/include/GL/GLwDrawA.h	Contains the **GLwDrawingArea** widget definitions derived from the Xt.
/usr/include/GL/GLwDrawAP.h	Contains **GLwDrawingArea** widget private definitions.
/usr/include/GL/GLwMDrawA.h	Contains the **GLwMDrawingArea** widget definitions derived from Motif.
/usr/include/GL/GLwMDrawAP.h	Contains **GLwMDrawingArea** widget private definitions.

Related Information

The **GLwCreateMDrawingArea** function, **GLwDrawingAreaMakeCurrent** function, **GLwDrawingAreaSwapBuffers** function.

The **glXChooseVisual** subroutine, **glXMakeCurrent** subroutine.

GLwDrawingAreaMakeCurrent Function

Purpose

Provides a front end to the **glXMakeCurrent** subroutine.

Library

OpenGL C bindings library: **libXGLW.a**

C Syntax

```
#include <X11/GLW/GLwDraw.h>

void GLwDrawingAreaMakeCurrent( Widget,  Context)
Widget Widget;
GLXContext Context;
```

Description

The **GLwDrawingAreaMakeCurrent** function provides a front end to the **glXMakeCurrent** subroutine by means of a widget (rather than a display or a window).

Parameters

Widget	Specifies the widget created with the **GLwCreateMDrawingArea** function.
Context	Specifies a GLX rendering context created with the **glXCreateContext** subroutine.

Files

/usr/include/GL/GLwDrawA.h	Contains the **GLwDrawingArea** widget definitions derived from the Xt.
/usr/include/GL/GLwDrawAP.h	Contains **GLwDrawingArea** widget private definitions.
/usr/include/GL/GLwMDrawA.h	Contains the **GLwMDrawingArea** widget definitions derived from Motif.
/usr/include/GL/GLwMDrawAP.h	Contains **GLwMDrawingArea** widget private definitions.

Related Information

The **GLwDrawingArea** or **GLwMDrawingArea** widget.

The **glXCreateContext** subroutine, **glXMakeCurrent** subroutine.

GLwDrawingAreaSwapBuffers Function

Purpose

Provides a front end to the **glXSwapBuffers** subroutine.

Library

OpenGL C bindings library: **libXGLW.a**

C Syntax

```
#include <X11/GLW/GLwDraw.h>

void GLwDrawingAreaSwapBuffers( Widget)
Widget Widget;
```

Description

The **GLwDrawingAreaSwapBuffers** function provides a front end to the **glXSwapBuffers** subroutine by means of a widget (rather than a display or a window).

Parameters

Widget	Specifies the widget created with the **GLwCreateMDrawingArea** function.

Files

/usr/include/GL/GLwDrawA.h	Contains the **GLwDrawingArea** widget definitions derived from the Xt.
/usr/include/GL/GLwDrawAP.h	Contains **GLwDrawingArea** widget private definitions.
/usr/include/GL/GLwMDrawA.h	Contains the **GLwMDrawingArea** widget definitions derived from Motif.
/usr/include/GL/GLwMDrawAP.h	Contains **GLwMDrawingArea** widget private definitions.

Related Information

The **GLwCreateMDrawingArea** function.

The **GLwDrawingArea** or **GLwMDrawingArea** widget.

The **glXSwapBuffers** subroutine.

Appendix. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Dept. LRAS/Bldg. 003
11400 Burnet Road
Austin, TX 78758-3498
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

© Copyright IBM Corp. 1994, 2002

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:
(c) (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. (c) Copyright IBM Corp. _enter the year or years_. All rights reserved.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be the trademarks or service marks of others.

Index

© Copyright IBM Corp. 1994, 2002

Readers' Comments — We'd Like to Hear from You

OpenGL 1.2 Reference Manual

Publication No. SR28-5125-01

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.

Submit your comments using one of these channels:
- Send your comments to the address on the reverse side of this form.
- Send your comments via e-mail to: pserinfo@us.ibm.com

If you would like a response from IBM, please fill in the following information:

Name	Address
Company or Organization	
Phone No.	E-mail address

Readers' Comments — We'd Like to Hear from You
SR28-5125-01

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Information Development
Department 04XA-905-6B013
11501 Burnet Road
Austin, TX 78758-3400

Fold and Tape **Please do not staple** Fold and Tape

Cut or Fold
Along Line

Printed in U. S. A.

SR28-5125-01