OpenGL 1.1 Reference

for HP-UX 11.x

Manufacturing Part Number: B2355-IE001

Edition E0302

Contents

1 A

glAccum

glAccum: operate on the accumulation buffer.

C Specification

```
void glAccum(
    GLenum op,
    GLfloat value)
```

Parameters

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

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 anti-aliasing (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 values. The number of bits per component in the accumulation buffer depends on the implementation. You can examine this number by calling glGetIntegerv four times, with arguments:

GL_ACCUM_RED_BITS,
GL_ACCUM_GREEN_BITS,
GL_ACCUM_BLUE_BITS, and
GL_ACCUM_ALPHA_BITS.

Regardless of the number of bits per component, the range of values stored by each component is [- 1, 1]. The accumulation buffer pixels are mapped one-to-one with frame buffer pixels.

glAccum operates on the accumulation buffer. The first argument, *op*, is a symbolic constant that selects an accumulation buffer operation. The second argument, *value*, is a floating-point value to be used in that operation. Five operations are specified: GL_ACCUM, GL_LOAD, GL_ADD, GL_MULT, and GL_RETURN.

All accumulation buffer operations are limited to the area of the current scissor box and applied identically to the red, green, blue, and alpha components of each pixel. If a glAccum operation results in a value outside the range [- 1, 1], the contents of an accumulation buffer pixel component are undefined.

The operations are as follows:

GL_ACCUM	Obtains R, G, B, and A values from the buffer currently selected for reading (see glReadBuffer).

	Each component value is divided by 2^n -1, where *n* is the number of bits allocated to each color component in the currently selected buffer. The result is a floating-point value in the range [0, 1], which is multiplied by *value* and added to the corresponding pixel component in the accumulation buffer, thereby updating the accumulation buffer.
GL_LOAD	Similar to GL_ACCUM, except that the current value in the accumulation buffer is not used in the calculation of the new value. That is, the R, G, B, and A values from the currently selected buffer are divided by 2^n - 1, multiplied by *value*, and then stored in the corresponding accumulation buffer cell, overwriting the current value.
GL_ADD	Adds *value* to each R, G, B, and A in the accumulation buffer.
GL_MULT	Multiplies each R, G, B, and A in the accumulation buffer by *value* and returns the scaled component to its corresponding accumulation buffer location.
GL_RETURN	Transfers accumulation buffer values to the color buffer or buffers currently selected for writing. Each R, G, B, and A component is multiplied by *value*, then multiplied by 2^n - 1, clamped to the range [0, 2^n - 1], and stored in the corresponding display buffer cell. The only fragment operations that are applied to this transfer are pixel ownership, scissor, dithering, and color writemasks.

To clear the accumulation buffer, call glClearAccum with R, G, B, and A values to set it to, then call glClear with the accumulation buffer enabled.

Notes

Only pixels within the current scissor box are updated by a glAccum operation.

Errors

- GL_INVALID_ENUM is generated if *op* is not an accepted value.
- GL_INVALID_OPERATION is generated if there is no accumulation buffer.
- GL_INVALID_OPERATION is generated if glAccum is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

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

See Also

glBlendFunc,
glClear,
glClearAccum,
glCopyPixels,
glGet,

glLogicOp,
glPixelStore,
glPixelTransfer,
glReadBuffer,
glReadPixels,
glScissor,
glStencilOp

glAlphaFunc

`glAlphaFunc`: specify the alpha test function.

C Specification

```
void glAlphaFunc(
    GLenum func,
    GLclampf ref)
```

Parameters

func	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 initial value is GL_ALWAYS.
ref	Specifies the reference value that incoming alpha values are compared to. This value is clamped to the range 0 through 1, where 0 represents the lowest possible alpha value and 1 the highest possible value. The initial reference value is 0.

Description

The alpha test discards fragments depending on the outcome of a comparison between an incoming fragment's alpha value and a constant reference value.

glAlphaFunc specifies the reference value and the comparison function. The comparison is performed only if alpha testing is enabled. By default, it is not enabled. (See glEnable and glDisable of GL_ALPHA_TEST.)

func and *ref* specify the conditions under which the pixel is drawn. The incoming alpha value is compared to ref using the function specified by *func*. If the value passes the comparison, the incoming fragment is drawn if it also passes subsequent stencil and depth buffer tests. If the value fails the comparison, no change is made to the frame buffer at that pixel location. The comparison functions are as follows:

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 (initial value).

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

Notes

Alpha testing is performed only in RGBA mode.

Errors

- GL_INVALID_ENUM is generated if *func* is not an accepted value.
- GL_INVALID_OPERATION is generated if glAlphaFunc is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_ALPHA_TEST_FUNC
glGet with argument GL_ALPHA_TEST_REF
glIsEnabled with argument GL_ALPHA_TEST

See Also

glBlendFunc,
glClear,
glDepthFunc,
glEnable,
glStencilFunc

glAreaTexturesResident

`glAreaTexturesResident`: determine if textures are loaded in texture memory.

C Specification

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

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.

Description

GL establishes a “working set” of textures that are resident in texture memory. These textures can be bound to a texture target much more efficiently than textures that are not resident.

glAreTexturesResident 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 *p_name* argument set to GL_TEXTURE_RESIDENT. This is the only way that the residence status of a default texture can be queried.

Notes

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

glAreTexturesResident 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 reside in virtual memory (there is no texture memory), they are considered always resident.

Some implementations may not load a texture until the first use of that texture.

Errors

- GL_INVALID_VALUE is generated if *n* is negative.
- GL_INVALID_VALUE is generated if any element in *textures* is 0 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.

See Also

glBindTexture,
glGetTexParameter,
glPrioritizeTextures,
glTexImage1D,
glTexImage2D,
glTexParameter

glArrayElement

`glArrayElement:` render a vertex using the specified vertex array element.

C Specification

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

Parameters

i	Specifies an index into the enabled vertex data arrays.

Description

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 corresponding 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.

Notes

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

glArrayElement 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.

See Also

glColorPointer,
glDrawArrays,
glEdgeFlagPointer,
glGetPointer,
glIndexPointer,
glInterleavedArrays,

glNormalPointer,
glTexCoordPointer,
glVertexPointer

2 B

glBegin

`glBegin, glEnd:` delimit the vertices of a primitive or a group of like primitives.

C Specification

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

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.

Description

glBegin and glEnd delimit the vertices that define a primitive or a group of like primitives. glBegin accepts a single argument that specifies in which of ten ways the vertices are interpreted. Taking *n* as an integer count starting at one, and *n* as the total number of vertices specified, the interpretations are as follows:

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. *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*/21 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 commands can be used between glBegin and glEnd. The commands are glVertex, glColor, glIndex, glNormal, glTexCoord, glEvalCoord, glEvalPoint, glArrayElement, glMaterial, and glEdgeFlag. Also, it is acceptable to use glCallList or glCallLists to execute display lists that include only the preceding commands. If any other GL command is executed between glBegin and glEnd, the error flag is set and the command is ignored.

Regardless of the value chosen for *mode*, there is no limit to the number of vertices that can be defined between glBegin and glEnd. Lines, triangles, quadrilaterals, and polygons that are incompletely specified are not drawn. Incomplete specification results when either too few vertices are provided to specify even a single primitive or when an incorrect multiple of vertices is specified. The incomplete primitive is ignored; the 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).

Errors

- GL_INVALID_ENUM is generated if mode is set to an unaccepted value.
- GL_INVALID_OPERATION is generated if glBegin is executed between a glBegin and the corresponding execution of glEnd.
- GL_INVALID_OPERATION is generated if glEnd is executed without being preceded by a glBegin.
- GL_INVALID_OPERATION is generated if a command other than glVertex, glColor, glIndex, glNormal, glTexCoord, glEvalCoord, glEvalPoint, glArrayElement, glMaterial, glEdgeFlag, glCallList, or glCallLists is executed between the execution of glBegin and the corresponding execution glEnd.

Execution of glEnableClientState, glDisableClientState, glEdgeFlagPointer, glTexCoordPointer, glColorPointer, glIndexPointer, glNormalPointer, glVertexPointer, glInterleavedArrays, or glPixelStore is not allowed after a call to glBegin and before the corresponding call to glEnd, but an error may or may not be generated.

See Also

glArrayElement,
glCallList,
glCallLists,
glColor,
glEdgeFlag,
glEvalCoord,
glEvalPoint,
glIndex,
glMaterial,
glNormal,
glTexCoord,
glVertex

gluBeginCurve

`gluBeginCurve, gluEndCurve`: delimit a NURBS curve definition.

C Specification

```
void gluBeginCurve(
    GLUnurbs* nurb)
void gluEndCurve(
    GLUnurbs* nurb)
```

Parameters

nurb	Specifies the NURBS object (created with gluNewNurbsRenderer).

Description

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

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

See Also

gluBeginSurface,
gluBeginTrim,
gluNewNurbsRenderer,
gluNurbsCurve,
glPopAttrib,
glPushAttrib

gluBeginPolygon

`gluBeginPolygon, gluEndPolygon`: delimit a polygon description.

C Specification

```
void gluBeginPolygon(
    GLUtesselator* tess)
void gluEndPolygon(
    GLUtesselator* tess)
```

Parameters

tess	Specifies the tessellation object (created with gluNewTess).

Description

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

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

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.

See Also

gluNewTess,
gluNextContour,
gluTessCallback,
gluTessVertex,
gluTessBeginPolygon,
gluTessBeginContour

gluBeginSurface

`gluBeginSurface, gluEndSurface:` delimit a NURBS surface definition.

C Specification

```
void gluBeginSurface(
    GLUnurbs* nurb)
void gluEndSurface(
    GLUnurbs* nurb)
```

Parameters

nurb	Specifies the NURBS object (created with gluNewNurbsRenderer).

Description

Use gluBeginSurface to mark the beginning of a NURBS surface definition. After calling gluBeginSurface, make one or more calls to gluNurbsSurface to define the attributes of the surface. Exactly one of these calls to gluNurbsSurface must have a surface type of GL_MAP2_VERTEX_3 or GL_MAP2_VERTEX_4. To mark the end of the NURBS surface definition, call gluEndSurface.

Trimming of NURBS surfaces is supported with gluBeginTrim, gluPwlCurve, gluNurbsCurve, andgluEndTrim. See the gluBeginTrim reference page for details.

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

See Also

gluBeginCurve,
gluBeginTrim,
gluNewNurbsRenderer,
gluNurbsCurve,
gluNurbsSurface,
gluPwlCurve

gluBeginTrim

`gluBeginTrim, gluEndTrim`: delimit a NURBS trimming loop definition.

C Specification

```
void gluBeginTrim(
    GLUnurbs* nurb)
void gluEndTrim(
    GLUnurbs* nurb)
```

Parameters

nurb	Specifies the NURBS object (created with gluNewNurbsRenderer).

Description

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

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

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

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

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

If a trimming loop definition contains multiple curves, the direction of the curves must be consistent (that is, the inside must be to the left of all of the curves). Nested trimming loops are legal as long as the curve orientations alternate correctly. If trimming curves are self-intersecting, or intersect one another, an error results.

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

See Also

gluBeginSurface,
gluNewNurbsRenderer,
gluNurbsCallback,
gluNurbsCurve,
gluPwlCurve

glBindTexture

`glBindTexture`: bind a named texture to a texture target.

C Specification

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

Parameters

target	Specifies the target to which the texture is bound. Must be either GL_TEXTURE_1D or GL_TEXTURE_2D. Initially, both GL_TEXTURE_1D and GL_TEXTURE_2D are bound to texture 0.
texture	Specifies the name of a texture.

Description

glBindTexture binds the texture named *texture* to the specified *target*. If the name does not exist, it is created. *target* must be either GL_TEXTURE_1D or GL_TEXTURE_2D. When a texture is bound to a target, the previous binding for that target is broken.

Texture names are unsigned integers. The value 0 is reserved to represent the default texture for each texture target. glGenTextures may be used 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 and a texture first bound to GL_TEXTURE_2D becomes two-dimensional. The state of a one-dimensional texture immediately after it is first bound is equivalent to the state of the default GL_TEXTURE_1D at GL initialization, and similarly for two-dimensional 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 “0” 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. When a bound texture is deleted, the default texture is bound to that target.

Once created, a named texture may be re-bound 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.

Notes

glBindTexture is available only if the GL version is 1.1 or greater.

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 that 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

See Also

glAreTexturesResident,
glDeleteTextures,
glGenTextures,
glGet,
glGetTexParameter,
glIsTexture,
glPrioritizeTextures,
glTexImage1D,
glTexImage2D,
glTexParameter

glBitMap

`glBitmap`: draw a bitmap.

C Specification

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

Parameters

width, height	Specify the pixel width and height of the bitmap image.
xorig, yorig	Specify the location of the origin in the bitmap image. The origin is measured from the lower left corner of the bitmap, with right and up being the positive axes.
xmove, ymove	Specify the *x* and *y* offsets to be added to the current raster position after the bitmap is drawn.
bitmap	Specifies the address of the bitmap image.

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 1s in the bitmap are written using the current raster color or index. Frame buffer pixels corresponding to 0s in the bitmap are not modified.

glBitmap takes seven arguments. The first pair specifies the width and height of the bitmap image. The second pair specifies the location of the bitmap origin relative to the lower left corner of the bitmap image. The third pair of arguments specifies *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 command, with *width* and *height* corresponding to the width and height arguments of that command, and with *type* set to GL_BITMAP and *format* set to GL_COLOR_INDEX.

Modes specified using glPixelStore affect the interpretation of bitmap image data; modes specified using glPixelTransfer do not.

If the current raster position is invalid, glBitmap is ignored. Otherwise, the lower left corner of the bitmap image is positioned at the window coordinates

$$x_w = \lfloor x_r - x_o \rfloor$$
$$y_w = \lfloor y_r - y_o \rfloor$$

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 (one) 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, texture coordinates, or index.

Notes

To set a valid raster position outside the viewport, first set a valid raster position inside the viewport, then call glBitmap with NULL as the *bitmap* parameter and with *xmove* and *ymove* set to the offsets of the new raster position. This technique is useful when panning an image around the viewport.

Errors

- GL_INVALID_VALUE is generated if *width* or *height* is negative.
- GL_INVALID_OPERATION is generated if glBitmap is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

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

See Also

glDrawPixels,
glPixelStore,
glPixelTransfer,
glRasterPos

glBlendColorEXT

`glBlendColorEXT`: set the blend color.

C Specification

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

Parameters

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

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).

Notes

glBlendColorEXT is part of the EXT_blend_color extension, not part of the core GL command set. If GL_EXT_blend_color is included in the string returned by glGetString, when called with argument GL_EXTENSIONS, extension EXT_blend_color is supported by the connection.

Errors

- GL_INVALID_OPERATION is generated if glBlendColorEXT is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with an argument of GL_BLEND_COLOR_EXT.

See Also

glBlendFunc,
glGetString

glBlendFunc

glBlendFunc: specify pixel arithmetic.

C Specification

```
void glBlendFunc(
    GLenum sfactor,
    GLenum dfactor)
```

Parameters

sfactor	Specifies how the red, green, blue, and alpha source blending factors are computed. Nine symbolic constants are accepted: GL_ZERO, GL_ONE, GL_DST_COLOR, GL_ONE_MINUS_DST_COLOR, GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA, and GL_SRC_ALPHA_SATURATE. The initial value is GL_ONE.
dfactor	Specifies how the red, green, blue, and alpha destination blending factors are computed. Eight symbolic constants are accepted: GL_ZERO, GL_ONE, GL_SRC_COLOR, GL_ONE_MINUS_SRC_COLOR, GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_DST_ALPHA, and GL_ONE_MINUS_DST_ALPHA. The initial value is GL_ZERO.

Description

In RGBA mode, pixels can be drawn using a function that blends the incoming (source) RGBA values with the RGBA values that are already in the frame buffer (the destination values). Blending is initially disabled. Use glEnable and glDisable with argument GL_BLEND to enable and disable blending.

glBlendFunc defines the operation of blending when it is enabled. sfactor specifies which of nine methods is used to scale the source color components. dfactor specifies which of eight methods is used to scale the destination color components. The eleven possible methods are described in the following table. Each method defines four scale factors, one each for red, green, blue, and alpha.

In the table and in subsequent equations, source and destination color components are referred to as (R_s, G_s, B_s, A_s) and (R_d, G_d, B_d, A_d). They are understood to have integer values between 0 and (k_R, k_G, k_B, k_A), where

$$k_c = 2^{m}c - 1$$

and (m_R, m_G, m_B, m_A) is the number of red, green, blue, and alpha bitplanes.

Source and destination scale factors are referred to as (s_R, s_G, s_B, s_A) and (d_R, d_G, d_B, d_A). The scale factors described in the table, denoted (f_R, f_G, f_B, f_A), represent either source or destination factors. All scale factors have range [0,1].

Parameters

Table 2-1

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_SRC_ALPHA_SATURATE	(*i*, *i*, *i*,1)

In the table,

$i = \min(A_s, k_A - A_d) / k_A$

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

$R_d = \min(k_R, R_s\ s_R + R_d\ d_R)$

$G_d = \min(k_G, G_s\ s_G + G_d\ d_G)$

$B_d = \min(k_B, B_s\ s_B + B_d\ d_B)$

$A_d = \min(k_A, A_s\ s_A + A_d\ d_A)$

Despite the apparent precision of the above equations, blending arithmetic is not exactly specified, because blending operates with imprecise integer color values. However, a blend factor that should be equal to 1 is guaranteed not to modify its multiplicand, and a blend factor equal to 0 reduces its multiplicand to 0. For example, when sfactor is GL_SRC_ALPHA, dfactor is GL_ONE_MINUS_SRC_ALPHA, and A_s is equal to k_A, the equations reduce to simple replacement:

$R_d = R_s$

$G_d = G_s$

$B_d = B_s$

$A_d = A_s$

Examples

Blend function (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA) is also useful for rendering anti-aliased points and lines in arbitrary order.

Polygon anti-aliasing is optimized using blend function (GL_SRC_ALPHA_SATURATE, GL_ONE) with polygons sorted from nearest to farthest. (See the glEnable, glDisable reference page and the GL_POLYGON_SMOOTH argument for information on polygon anti-aliasing.) Destination alpha bitplanes, which must be present for this blend function to operate correctly, store the accumulated coverage.

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, the GL performs blending separately for each enabled buffer, using the contents of that buffer for destination color. (See glDrawBuffer.)

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

Errors

- GL_INVALID_ENUM is generated if either *sfactor* or *dfactor* is not an accepted value.
- GL_INVALID_OPERATION is generated if glBlendFunc is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_BLEND_SRC
glGet with argument GL_BLEND_DST
glIsEnabled with argument GL_BLEND

See Also

glAlphaFunc,
glClear,
glDrawBuffer,
glEnable,
glLogicOp,
glStencilFunc

gluBuild1DMipmaps

`gluBuild1DMipmaps`: create 1D mipmaps.

C Specification

```
GLint gluBuild1DMipmaps(
    GLenum target,
    GLint component,
    GLsizei width,
    GLenum format,
    GLenum type,
const void *data)
```

Parameters

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

Description

gluBuild1DMipmaps builds a series of pre-filtered 1D texture maps of decreasing resolution. Mipmaps can be used so that textures don't appear aliased.

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

gluBuild1DMipmaps first checks whether the *width* of data is a power of 2. If not, it scales a copy of *data* (up or down) to the nearest power of two. This copy is used as the base for subsequent mipmapping operations. For example, if *width* is 57, a copy of *data* scales up to 64 before mipmapping takes place. (If *width* is exactly between powers of 2, the copy of *data* is scaled upward.)

If the GL version is 1.1 or greater, gluBuild1DMipmaps uses proxy textures (see glTexImage1D) to determine if the implementation can store the requested texture in texture memory. If there isn't enough room, *width* is halved (and halved again) until it fits.

Next, gluBuild1DMipmaps builds a series of mipmap levels; it halves a copy of *data* (or a scaled version of *data*, if necessary) until size 1 is reached. At each level, each texel in the halved image is an average of the corresponding two texels in the larger image.

glTexImage1D is called to load each of these images by level. If *width* is a power of 2 which fits in the implementation, level 0 is a copy of *data*, and the highest level is log2 *width*. For example, if *width* is 64, the following images are built: 64x1, 32x1, 16x1, 8x1, 4x1, 2x1 and 1x1. These correspond to levels 0 through 6, respectively.

See the glTexImage1D reference page for a description of the acceptable values for *type*. See the glDrawPixels reference page for a description of the acceptable values for *data*.

Notes

While you can't query the maximum level directly, you can derive it indirectly by calling glGetTexLevelParameter. First, query for the width actually used at level 0. (The width may be unequal to width since gluBuild1DMipmaps might have shrunk or expanded width if *width* isn't a power of 2 or if the implementation only supports smaller textures. The maximum level can then be derived using the formula $\log_2$ *width*.

Errors

- GLU_INVALID_VALUE is returned if *width* is negative.
- GLU_INVALID_ENUM is returned if *format* or *type* is not one of the accepted values.

Bugs

Passing GL_STENCIL_INDEX or GL_DEPTH_COMPONENT as format will incorrectly return 0 and set the error code to GL_INVALID_ENUM. It should return GLU_INVALID_ENUM and not set an error code.

See Also

glTexImage1D,
gluBuild2DMipmaps,
gluErrorString,
gluScaleImage

gluBuild2DMipmaps

`gluBuild2DMipmaps`: create 2D mipmaps.

C Specification

```
GLint gluBuild2DMipmaps(
    GLenum target,
    GLint component,
    GLsizei width,
    GLsizei height,
    GLenum format,
    GLenum type,
const void *data)
```

Parameters

target	Specifies the target texture. Must be GL_TEXTURE_2D.
component	Specifies the number of color components in the texture. Must be 1, 2, 3, or 4.
target	Specifies the target texture. Must be GL_TEXTURE_2D.
width, height	Specifies the width and height, respectively, of the texture image.
format	Specifies the format of the pixel data. Must be one of: GL_COLOR_INDEX, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_LUMINANCE, and GL_LUMINANCE_ALPHA.
type	Specifies the data type for data. Must be one of: GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT or GL_FLOAT.
data	Specifies a pointer to the image data in memory.

Description

gluBuild2DMipmaps builds a series of pre-filtered 2D texture maps of decreasing resolution. Mipmaps can be used so that textures don't appear aliased.

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

gluBuild2DMipmaps first check whether *width* and *height* of *data* are both powers of 2. If not, gluBuild2DMipmaps scales a copy of *data* up or down to the nearest power of 2. This copy is then used as the base for subsequent mipmapping operations. For example, if *width* is 57 and *height* is 23, then a copy of *data* scales up to 64 and down to 16, respectively, before mipmapping takes place. (If *width* or *height* is exactly between powers of 2, the copy of *data* is scaled upward.)

If the GL version is 1.1 or greater, gluBuild2DMipmaps then uses proxy textures (see glTexImage1D) to determine whether there's enough room for the requested texture in the implementation. If not, *width* is halved (and halved again) until it fits.

gluBuild2DMipmaps then uses proxy textures (see glTexImage2D) to determine if the implementation can store the requested texture in texture memory. If not, both dimensions are continually halved until it fits.

Next, gluBuild2DMipmaps builds a series of images; it halves a copy of *type* (or a scaled version of *type*, if necessary) along both dimensions until size 11 is reached. At each level, each texel in the halved mipmap is an average of the corresponding four texels in the larger mipmap. (In the case of rectangular images, halving the images repeatedly eventually results in an n 1 or 1n configuration. Here, two texels are averaged instead.)

glTexImage2D is called to load each of these images by level. If *width* and *height* are both powers of 2 which fit in the implementation, level 0 is a copy of data, and the highest level is $\log_2$(max(*width*, *height*)). For example, if width is 64 and height is 16, the following mipmaps are built: 64×16, 32×8, 16×4, 8×2, 4×1, 2×1 and 1×1. These correspond to levels 0 through 6, respectively.

See the glTexImage1D reference page for a description of the acceptable values for *format*. See the glDrawPixels reference page for a description of the acceptable values for *type*.

Notes

While you can't query the maximum level directly, you can derive it indirectly by calling glGetTexLevelParameter. First, query for the width and height actually used at level 0. (The width and height may be unequal to *width* and *height* since proxy textures might have shrunk or expanded them if *width* or *height* are not powers of 2 or if the implementation only supports smaller textures.) The maximum level can then be derived using the formula $\log_2$(max(*width*, *height*)).

Errors

- GLU_INVALID_VALUE is returned if *width* or *height* are negative.
- GLU_INVALID_ENUM is returned if *format* or *type* is not one of the accepted values.

See Also

glDrawPixels,
glTexImage1D,
glTexImage2D,
gluBuild1DMipmaps,
gluErrorString,
gluScaleImage

3 C

glCallList

`glCallList`: execute a display list.

C Specification

```
void glCallList(
    GLuint list)
```

Parameters

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

Description

glCallList causes the named display list to be executed. The commands 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.

glCallList can appear inside a display list. To avoid the possibility of infinite recursion resulting from display lists calling one another, a limit is placed on the nesting level of display lists during display-list execution. This limit is at least 64, and it depends on the implementation.

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 remain after execution of the display list is completed. Use glPushAttrib, glPopAttrib, glPushMatrix, and glPopMatrix to preserve GL state across glCallList calls.

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

glGet with argument GL_MAX_LIST_NESTING glIsList

See Also

glCallLists,
glDeleteLists,
glGenLists,
glNewList,
glPushAttrib,
glPushMatrix

glCallLists

`glCallLists:` execute a list of display lists.

C Specification

```
void glCallLists(
    GLsizei n,
    GLenum type,
const GLvoid *lists)
```

Parameters

n	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*.

Description

glCallLists 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.

glCallLists provides an efficient means for executing more than one display list. *type* allows lists with various name formats to be accepted. The formats are as follows:

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 two-byte integers, each in the range - 32768 through 32767.
GL_UNSIGNED_SHORT	*lists* is treated as an array of unsigned two-byte integers, each in the range 0 through 65535.
GL_INT	*lists* is treated as an array of signed four-byte integers.
GL_UNSIGNED_INT	*lists* is treated as an array of unsigned four-byte integers.
GL_FLOAT	*lists* is treated as an array of four-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 65536 times the unsigned value of the first byte, plus 256 times the unsigned value of the second byte, plus the unsigned value of the third byte.
GL_4_BYTES	*lists* is treated as an array of signed bytes, each in the range 128 through 127.

The list of display-list names is not null-terminated. Rather, *n* specifies how many names are to be taken from *lists*.

An additional level of indirection is made available with the glListBase command, which specifies an unsigned offset that is added to each display-list name specified in *lists* before that display list is executed.

glCallLists can appear inside a display list. To avoid the possibility of infinite recursion resulting from display list scaling one another, a limit is placed on the nesting level of display lists during display-list execution. This limit must be at least 64, and it depends on the implementation.

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 glPushAttrib, glPopAttrib, glPushMatrix, and glPopMatrix to preserve GL state across glCallLists calls.

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.

Errors

- GL_INVALID_VALUE is generated if *n* is negative.
- GL_INVALID_ENUM is generated if *type* is not one of GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, GL_2_BYTES, GL_3_BYTES, GL_4_BYTES.

Associated Gets

glGet with argument GL_LIST_BASE
glGet with argument GL_MAX_LIST_NESTING
glIsList

See Also

glCallList,
glDeleteLists,
glGenLists,
glListBase,
glNewList,
glPushAttrib,
glPushMatrix

glXChooseVisual

`glXChooseVisual`: return a visual that matches specified attributes.

C Specification

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

Parameters

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

Description

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

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

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

GLX_USE_GL

Ignored. Only visuals that can be rendered with GLX are considered.

GLX_BUFFER_SIZE

Must be followed by a non-negative integer that indicates the desired color index buffer size. The smallest index buffer of at least the specified size is preferred. Ignored if GLX_RGBA is asserted.

GLX_LEVEL

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

GLX_RGBA

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

GLX_DOUBLEBUFFER

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

GLX_STEREO

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

GLX_AUX_BUFFERS

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

GLX_RED_SIZE

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

GLX_GREEN_SIZE

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

GLX_BLUE_SIZE

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

GLX_ALPHA_SIZE

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

GLX_DEPTH_SIZE

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

GLX_STENCIL_SIZE

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

GLX_ACCUM_RED_SIZE

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

GLX_ACCUM_GREEN_SIZE

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

GLX_ACCUM_BLUE_SIZE

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

GLX_ACCUM_ALPHA_SIZE

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

Examples

```
attribList =
GLX_RGBA,
GLX_RED_SIZE, 4,
GLX_GREEN_SIZE, 4,
GLX_BLUE_SIZE, 4,
None};
```

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

Notes

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

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.

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

Errors

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

See Also

glXCreateContext,
glXGetConfig

glClear

`glClear`: clear buffers to preset values.

C Specification

```
void glClear(
    GLbitfield mask)
```

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.

Description

glClear sets the bitplane area of the window 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 depth-buffering are ignored by glClear.

glClear takes a single argument that is the bitwise or of several values indicating which buffer is to be cleared. The values are as follows:

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.

Notes

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

Errors

- GL_INVALID_VALUE is generated if any bit other than the four defined bits is set in *mask*.
- GL_INVALID_OPERATION is generated if glClear is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated gets

glGet with argument GL_ACCUM_CLEAR_VALUE

glGet with argument GL_DEPTH_CLEAR_VALUE

glGet with argument GL_INDEX_CLEAR_VALUE

glGet with argument GL_COLOR_CLEAR_VALUE

glGet with argument GL_STENCIL_CLEAR_VALUE

See Also

glClearAccum,
glClearColor,
glClearDepth,
glClearIndex,
glClearStencil,
glDrawBuffer,
glScissor

glClearAccum

`glClearAccum`: specify clear values for the accumulation buffer.

C Specification

```
void glClearAccum(
    GLfloat red,
    GLfloat green,
    GLfloat blue,
    GLfloat alpha)
```

Parameters

red, green, blue, alpha	Specify the red, green, blue, and alpha values used when the accumulation buffer is cleared. The initial values are all 0.

Description

glClearAccum specifies the red, green, blue, and alpha values used by glClear to clear the accumulation buffer.

Values specified by glClearAccum are clamped to the range [-1, 1].

Errors

- GL_INVALID_OPERATION is generated if glClearAccum is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_ACCUM_CLEAR_VALUE

See Also

glClear

glClearColor

`glClearColor`: specify clear values for the color buffers.

C Specification

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

Parameters

red, green, blue, alpha
Specify the red, green, blue, and alpha values used when the color buffers are cleared. The initial values are all 0.

Description

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

Errors

- GL_INVALID_OPERATION is generated if glClearColor is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_COLOR_CLEAR_VALUE

See Also

glClear

glClearDepth

`glClearDepth`: specify the clear value for the depth buffer.

C Specification

```
void glClearDepth(
    GLclampd depth)
```

Parameters

depth	Specifies the depth value used when the depth buffer is cleared. The initial value is 1.

Description

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

Errors

- GL_INVALID_OPERATION is generated if glClearDepth is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_DEPTH_CLEAR_VALUE

See Also

glClear

glClearindex

`glClearIndex`: specify the clear value for the color index buffers.

C Specification

```
void glClearIndex(
    GLfloat c)
```

Parameters

c	Specifies the index used when the color index buffers are cleared. The initial value is 0.

Description

glClearIndex specifies the index used by glClear to clear the color index buffers. *c* is not clamped. Rather, *c* is converted to a fixed-point value with unspecified precision to the right of the binary point. The integer part of this value is then masked with $2^m - 1$, where *m* is the number of bits in a color index stored in the frame buffer.

Errors

- GL_INVALID_OPERATION is generated if glClearIndex is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_INDEX_CLEAR_VALUE
glGet with argument GL_INDEX_BITS

See Also

glClear

glClearStencil

`glClearStencil`: specify the clear value for the stencil buffer.

C Specification

```
void glClearStencil(
    GLint s)
```

Parameters

s	Specifies the index used when the stencil buffer is cleared. The initial value is 0.

Description

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

Errors

- GL_INVALID_OPERATION is generated if glClearStencil is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_STENCIL_CLEAR_VALUE
glGet with argument GL_STENCIL_BITS

See Also

glClear

glClipPlane

glClipPlane: specify a plane against which all geometry is clipped.

C Specification

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

Parameters

plane	Specifies which clipping plane is being positioned. Symbolic names of the form GL_CLIP_PLANEi, 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*. glClipPlane allows the specification of additional planes, not necessarily perpendicular to the *x*, *y*, or *Z* axis, against which all geometry is clipped. To determine the maximum number of additional clipping planes, call glGetIntegerv with argument GL_MAX_CLIP_PLANES. All implementations support at least six such clipping planes. Because the resulting clipping region is the intersection of the defined half-spaces, it is always convex.

glClipPlane 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 zero, the vertex is \f2in\f1 with respect to that clipping plane. Otherwise, it is *out*.

To enable and disable clipping planes, call glEnable and glDisable with the argument GL_CLIP_PLANEi, where *i* is the plane number.

All clipping planes are initially defined as (0, 0, 0, 0) in eye coordinates and are disabled.

Notes

It is always the case that GL_CLIP_PLANEi = GL_CLIP_PLANE0 + i.

Errors

- GL_INVALID_ENUM is generated if *plane* is not an accepted value.
- GL_INVALID_OPERATION is generated if glClipPlane is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated gets

glGetClipPlane
glIsEnabled with argument GL_CLIP_PLANE\f2i\fP

See Also

glEnable

glColor

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.

C Specification

```
void glColor3b(
    GLbyte red,
    GLbyte green,
    GLbyte blue)
void glColor3d(
    GLdouble red,
    GLdouble green,
    GLdouble blue)
void glColor3f(
    GLfloat red,
    GLfloat green,
    GLfloat blue)
void glColor3i(
    GLint red,
    GLint green,
    GLint blue)
void glColor3s(
    GLshort red,
    GLshort green,
    GLshort blue)
void glColor3ub(
    GLubyte red,
    GLubyte green,
    GLubyte blue)
void glColor3ui(
    GLuint red,
    GLuint green,
    GLuint blue)
void glColor3us(
    GLushort red,
    GLushort green,
    GLushort blue)
void glColor4b(
    GLbyte red,
    GLbyte green,
    GLbyte blue,
    GLbyte alpha)
void glColor4d(
    GLdouble red,
    GLdouble green,
    GLdouble blue,
    GLdouble alpha)
void glColor4f(
    GLfloat red,
    GLfloat green,
    GLfloat blue,
    GLfloat alpha)
void glColor4i(
    GLint red,
    GLint green,
    GLint blue,
    GLint alpha)
void glColor4s(
    GLshort red,
    GLshort green,
    GLshort blue,
    GLshort alpha)
void glColor4ub(
    GLubyte red,
    GLubyte green,
    GLubyte blue,
    GLubyte alpha)
void glColor4ui(
    GLuint red,
    GLuint green,
    GLuint blue,
    GLuint alpha)
void glColor4us(
    GLushort red,
    GLushort green,
    GLushort blue,
   GLushort alpha)
void glColor3bv(
    const GLbyte *v)
void glColor3dv(
    const GLdouble *v)
void glColor3fv(
    const GLfloat *v)
void glColor3iv(
    const GLint *v)
void glColor3sv(
    const GLshort *v)
void glColor3ubv(
    const GLubyte *v)
void glColor3uiv(
    const GLuint *v)
void glColor3usv(
    const GLushort *v)
void glColor4bv(
    const GLbyte *v)
void glColor4dv(
    const GLdouble *v)
```

```
void glColor4fv(
    const GLfloat *v)
void glColor4iv(
    const GLint *v)
void glColor4sv(
    const GLshort *v)
void glColor4ubv(
    const GLubyte *v)
void glColor4uiv(
    const GLuint *v)
void glColor4usv(
    const GLushort *v)
```

Parameters

red, green, blue	Specify new red, green, and blue values for the current color.
alpha	Specifies a new alpha value for the current color. Included only in the four-argument glColor4 commands.
v	Specifies a pointer to an array that contains red, green, blue, and (sometimes) alpha values.

Description

The GL stores both a current single-valued color index and a current four-valued RGBA color. glColor sets a new four-valued RGBA color. glColor 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 (full intensity) implicitly. glColor4 variants specify all four color components explicitly.

glColor3b, glColor4b, glColor3s, glColor4s, glColor3i, and glColor4i take three or four signed byte, short, or long integers as arguments. When v is appended to the name, the color commands 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 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. (Note that this mapping does not convert 0 precisely to 0.0.) Floating-point values are mapped directly.

Neither floating-point nor signed integer values are clamped to the range [0, 1] before the current color is updated. However, color components are clamped to this range before they are interpolated or written into a color buffer.

Notes

The initial value for the current color is (1, 1, 1, 1).

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

glGet with argument GL_CURRENT_COLOR
glGet with argument GL_RGBA_MODE

See Also

glIndex

glColorMask

`glColorMask`: enable and disable writing of frame buffer color components.

C Specification

```
void glColorMask(
    GLboolean red,
    GLboolean green,
    GLboolean blue,
    GLboolean alpha)
```

Parameters

red, green, blue, alpha	Specify whether red, green, blue, and alpha can or cannot be written into the frame buffer. The initial values are all GL_TRUE, indicating that the color components can be written.

Description

glColorMask specifies whether the individual color components in the frame buffer can or cannot be written. If *red* is GL_FALSE, for example, no change is made to the red component of any pixel in any of the color buffers, regardless of the drawing operation attempted.

Changes to individual bits of components cannot be controlled. Rather, changes are either enabled or disabled for entire color components.

Errors

- GL_INVALID_OPERATION is generated if glColorMask is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_COLOR_WRITEMASK
glGet with argument GL_RGBA_MODE

See Also

glColor,
glColorPointer,
glDepthMask,
glIndex,
glIndexPointer,
glIndexMask,
glStencilMask

glColorMaterial

`glColorMaterial`: cause a material color to track the current color.

C Specification

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

Parameters

face	Specifies whether front, back, or both front and back material parameters should track the current color. Accepted values are GL_FRONT, GL_BACK, and GL_FRONT_AND_BACK. The initial value is GL_FRONT_AND_BACK.
mode	Specifies which of several material parameters track the current color. Accepted values are GL_EMISSION, GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, and GL_AMBIENT_AND_DIFFUSE. The initial value is GL_AMBIENT_AND_DIFFUSE.

Description

glColorMaterial 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.

To enable and disable GL_COLOR_MATERIAL, call glEnable and glDisable with argument GL_COLOR_MATERIAL. GL_COLOR_MATERIAL is initially disabled.

Notes

glColorMaterial makes it possible to change a subset of material parameters for each vertex using only the glColor command, without calling glMaterial. If only such a subset of parameters is to be specified for each vertex, calling glColorMaterial is preferable to calling glMaterial.

Call glColorMaterial before enabling GL_COLOR_MATERIAL.

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.

If the GL version is 1.1 or greater, and GL_COLOR_MATERIAL is enabled, evaluated color values affect the results of the lighting equation as if the current color were being modified, but no change is made to the tracking lighting parameter of the current color.

Errors

- GL_INVALID_ENUM is generated if *face* or *mode* is not an accepted value.

- GL_INVALID_OPERATION is generated if glColorMaterial is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glIsEnabled with argument GL_COLOR_MATERIAL
glGet with argument GL_COLOR_MATERIAL_PARAMETER
glGet with argument GL_COLOR_MATERIAL_FACE

See Also

glColor,
glColorPointer,
glDrawElements,
glEnable,
glLight,
glLightModel,
glMaterial

glColorPointer

`glColorPointer`: define an array of colors.

C Specification

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

Parameters

size	Specifies the number of components per color. 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, and GL_DOUBLE are accepted.
stride	Specifies the byte offset between consecutive colors. If *stride* is 0, (the initial value), the colors are understood to be tightly packed in the array.
pointer	Specifies a pointer to the first component of the first color element in the array.

Description

glColorPointer 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. *type* specifies the data type of each color component, and *stride* specifies 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; 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.

Notes

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

The color array is initially disabled and isn't accessed when glArrayElement, glDrawArrays, or glDrawElements is called.

Execution of glColorPointer is not allowed between the execution of glBegin and the corresponding execution of glEnd, but an error may or may not be generated. If no error is generated, the operation is undefined.

glColorPointer is typically implemented on the client side.

Color array parameters are client-side state and are therefore not saved or restored by glPushAttrib and glPopAttrib. Use glPushClientAttrib and glPopClientAttrib instead.

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* 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

See Also

glArrayElement,
glDrawArrays,
glDrawElements,
glEdgeFlagPointer,
glEnable,
glGetPointer,
glIndexPointer,
glInterleavedArrays,
glNormalPointer,
glPopClientAttrib,
glPushClientAttrib,
glTexCoordPointer,
glVertexPointer

glXCopyContext

`glXCopyContext`: copy state from one rendering context to another.

C Specification

```
void glXCopyContext(
    Display *dpy,
    GLXContext src,
    GLXContext dst,
    unsigned long mask)
```

Parameters

dpy	Specifies the connection to the X server.
src	Specifies the source context.
dst	Specifies the destination context.
mask	Specifies which portions of src state are to be copied to dst.

Description

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

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

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

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

Notes

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

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

Errors

- *BadMatch* is generated if rendering contexts *src* and *dst* do not share an address space or were not created with respect to the same screen.
- *BadAccess* is generated if *dst* is current to any thread (including the calling thread) at the time glXCopyContext is called.
- GLXBadCurrentWindow is generated if *src* is the current context and the current drawable is a window that is no longer valid.
- GLXBadContext is generated if either *src* or *dst* is not a valid GLX context.

See Also

glPushAttrib,
glXCreateContext,
glXIsDirect

glCopyPixels

`glCopyPixels`: copy pixels in the frame buffer.

C Specification

```
void glCopyPixels(
    GLint x,
    GLint y,
    GLsizei width,
    GLsizei height,
    GLenum type)
```

Parameters

x, y	Specify the window coordinates of the lower left corner of the rectangular region of pixels to be copied.
width, height	Specify the dimensions of the rectangular region of pixels to be copied. Both must be nonnegative.
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.

Description

glCopyPixels 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.

x and *y* specify the window coordinates of the lower left corner of the rectangular region to be copied. *width* and *height* specify the dimensions of the rectangular region to be copied. Both *width* and *height* must not be negative.

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

glCopyPixels copies values from each pixel with the lower left-hand corner at (x + i, y + j) for $0 \geq i < width$ and $0 \geq j < height$. 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.

type 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 RGBA colors are read from the buffer currently specified as the read source buffer (see glReadBuffer). 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 toGL_INDEX_OFFSET. If GL_INDEX_SHIFT is negative, the shift is to the right. In either case, zero bits fill otherwise unspecified bit locations in the result. If GL_MAP_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 2^b - 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 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.

The GL then converts the resulting indices or RGBA colors 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 GL then converts the resulting depth components 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, zero bits fill otherwise unspecified bit locations in the result. If GL_MAP_STENCIL is true, the index is replaced with the value that it references in lookup table GL_PIXEL_MAP_S_TO_S. Whether the lookup replacement of the index is done or not, the integer part of the index is then ANDed with 2^b - 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 ith location of the jth 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 write operations.

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, then fragments are generated for pixels whose centers are in the rectangle with corners at

$(x_r + zoom_x i, y_r + zoom_y j)$

and

$(x_r + zoom_x(i+1), y_r + zoom_y (j+1))$

where $zoom_x$ is the value of GL_ZOOM_X and $zoom_y$ is the value of GL_ZOOM_Y.

Examples

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

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

Notes

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

Errors

- GL_INVALID_ENUM is generated if *type* is not an accepted value.
- GL_INVALID_VALUE is generated if either *width* or *height* is negative.
- GL_INVALID_OPERATION is generated if *type* is GL_DEPTH and there is no depth buffer.
- GL_INVALID_OPERATION is generated if *type* is GL_STENCIL and there is no stencil buffer.
- GL_INVALID_OPERATION is generated if glCopyPixels is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID

See Also

glDepthFunc,
glDrawBuffer,
glDrawPixels,
glPixelMap,
glPixelTransfer,
glPixelZoom,
glRasterPos,

glReadBuffer,
glReadPixels,
glStencilFunc

glCopyTexImage1D

`glCopyTexImage1D`: copy pixels into a 1D texture image.

C Specification

```
void glCopyTexImage1D(
    GLenum target,
    GLint level,
    GLenum internalFormat,
    GLint x,
    GLint y,
    GLsizei width,
    GLint border)
```

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.
internalFormat	Specifies the internal format of the texture. Must be one of the following symbolic constants: 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. Additionally, internalFormat maybe one of the symbolic constants GL_DEPTH_COMPONENT, GL_DEPTH_COMPONENT16_EXT, GL_DEPTH_COMPONENT24_EXT, or GL_DEPTH_COMPONENT32_EXT.
x, y	Specify the window coordinates of the left corner of the row of pixels to be copied.
width	Specifies the width of the texture image. Must be 0 or $2^n + 2 \times$ 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.

Description

glCopyTexImage1D 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 $width+2 \times$ 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 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.

Notes

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

Texturing has no effect in color index mode.

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

The GL_DEPTH_COMPONENT, GL_DEPTH_COMPONENT16_EXT, GL_DEPTH_COMPONENT24_EXT, and GL_DEPTH_COMPONENT32_EXT values of *internalFormat* may be used.

An image with 0 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 0.
- GL_INVALID_VALUE may be generated if *level* is greater than $\log_2$ *max*, where *max* is the returned value of GL_MAX_TEXTURE_SIZE.
- GL_INVALID_VALUE is generated if *internalFormat* is not an allowable value.
- GL_INVALID_VALUE is generated if *width* is less than 0 or greater than 2 + GL_MAX_TEXTURE_SIZE, or if it cannot be represented as $2^n + 2 \times$ 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 glCopyTexImage1D is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGetTexImage

glIsEnabled with argument GL_TEXTURE_1D

See Also

glCopyPixels,
glCopyTexImage2D,
glCopyTexSubImage1D,
glCopyTexSubImage2D,
glPixelStore
glPixelTransfer,
glTexEnv,
glTexGen,
glTexImage1D,
glTexImage2D,
glTexSubImage1D,
glTexSubImage2D,
glTexParameter

glCopyTexImage2D

glCopyTexImage2D: copy pixels into a 2D texture image.

C Specification

C Specification

```
void glCopyTexImage2D(
    GLenum target,
    GLint level,
    GLenum internalFormat,
    GLint x,
    GLint y,
    GLsizei width,
    GLsizei height,
    GLint border)
```

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.
internalFormat	Specifies the internal format of the texture. Must be one of the following symbolic constants: 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. Additionally, *internalFormat* may be one of the symbolic constants GL_DEPTH_COMPONENT, GL_DEPTH_COMPONENT16_EXT, GL_DEPTH_COMPONENT24_EXT, or GL_DEPTH_COMPONENT32_EXT.
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 image. Must be 0 or $2^n + 2 \times border$ for some integer *n*.
height	Specifies the height of the texture image. Must be 0 or $2^m + 2 \times border$ for some integer *m*.
border	Specifies the width of the border. Must be either 0 or 1.

Description

glCopyTexImage2D 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 a height of *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, 1] 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.

Notes

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

Texturing has no effect in color index mode.

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

The GL_DEPTH_COMPONENT, GL_DEPTH_COMPONENT16_EXT, GL_DEPTH_COMPONENT24_EXT, and GL_DEPTH_COMPONENT32_EXT values of *internalFormat* may be used.

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 0.
- GL_INVALID_VALUE may be generated if *level* is greater than $\log_2$ *max*, where *max* is the returned value of GL_MAX_TEXTURE_SIZE.
- GL_INVALID_VALUE is generated if *width* or *height* is less than 0, greater than 2 + GL_MAX_TEXTURE_SIZE, or if *width* or *height* cannot be represented as $2^k + 2 \times$ 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

See Also

glCopyPixels,
glCopyTexImage1D,
glCopyTexSubImage1D,
glCopyTexSubImage2D,
glPixelStore,
glPixelTransfer,
glTexEnv,
glTexGen,
glTexImage1D,
glTexImage2D,
glTexSubImage1D,
glTexSubImage2D,
glTexParameter

glCopyTexSubImage1D

`glCopyTexSubImage1D`: copy a one-dimensional texture sub image.

C Specification

```
void glCopyTexSubImage1D(
    GLenum target,
    GLint level,
    GLint xoffset,
    GLint x,
    GLint y,
    GLsizei width)
```

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 the texel offset within the texture array.
x, *y*	Specify the window coordinates of the left corner of the row of pixels to be copied.
width	Specifies the width of the texture sub image.

Description

glCopyTexSubImage1D 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.

Notes

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

Texturing has no effect in color index mode.

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_1D.
- GL_INVALID_OPERATION is generated if the texture array has not been defined by a previous glTexImage1D or glCopyTexImage1D operation.
- GL_INVALID_VALUE is generated if *level* is less than 0.
- GL_INVALID_VALUE may be generated if $level > \log_2 max$, 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

See Also

glCopyPixels,
glCopyTexImage1D,
glCopyTexImage2D,
glCopyTexSubImage2D,
glPixelStore,
glPixelTransfer,
glTexEnv,
glTexGen,
glTexImage1D,
glTexImage2D,
glTexParameter,
glTexSubImage1D,
glTexSubImage2D

glCopyTexSubImage2D

`glCopyTexSubImage2D`: copy a two-dimensional texture sub image.

C Specification

```
void glCopyTexSubImage2D(
    GLenum target,
    GLint level,
    GLint xoffset,
    GLint yoffset,
    GLint x,
    GLint y,
    GLsizei width,
    GLsizei height)
```

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.
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 sub image.
height	Specifies the height of the texture sub image.

Description

glCopyTexSubImage2D replaces a rectangular 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.

Notes

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

Texturing has no effect in color index mode.

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_2D.
- GL_INVALID_OPERATION is generated if the texture array has not been defined by a previous glTexImage2D or glCopyTexImage2D operation.
- GL_INVALID_VALUE is generated if *level* is less than 0.
- GL_INVALID_VALUE may be generated if *level* is greater than log_2 *max*, 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

See Also

glCopyPixels,
glCopyTexImage1D,
glCopyTexImage2D,
glCopyTexSubImage1D,
glPixelStore,
glPixelTransfer,
glTexEnv,
glTexGen,
glTexImage1D,
glTexImage2D,

glTexParameter,
glTexSubImage1D,
glTexSubImage2D

glCopyTexSubImage3DEXT

`glCopyTexSubImage3DEXT`: copy pixels into a 3D texture sub image.

C Specification

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

Parameters

target	The target texture. Must be GL_TEXTURE_3D_EXT.
level	The level-of-detail number. Level 0 is the base image level, and level *n* is the *n*th mipmap reduction image.
xoffset	Texel offset in the X direction within the texture array.
yoffset	Texel offset in the Y direction within the texture array.
zoffset	Texel offset in the Z direction within the texture array.
x	The X coordinate of the lower-left corner of the pixel rectangle to be transferred to the texture array.
y	The Y coordinate of the lower-left corner of the pixel rectangle to be transferred to the texture array.
width	The width of the texture sub image.
height	The height of the texture sub image.

Description

glCopyTexSubImage3DEXT 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*) having width *width* and height *height* replaces the rectangular area of the S-T slice located at *zoffset* with X indices *xoffset* through *xoffset* + *width* - 1, inclusive, and Y indices *yoffset* through *yoffset* + *height* - 1, inclusive.

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.

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.

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.

Notes

glCopyTexSubImage3DEXT is part of the EXT_copy_texture extension.

Errors

- GL_INVALID_ENUM is generated when *target* is not one of the allowable values.
- GL_INVALID_VALUE is generated if *level* is less than zero or greater than log_2max, where *max* is the returned value of GL_MAX_TEXTURE_SIZE.
- GL_INVALID_VALUE is generated if *xoffset* <- TEXTURE_BORDER, (*xoffset* + *width*) > (TEXTURE_WIDTH - TEXTURE_BORDER), *yoffset* < - TEXTURE_BORDER, or if *zoffset* < - TEXTURE_BORDER, where TEXTURE_WIDTH, TEXTURE_HEIGHT, and TEXTURE_BORDER are the state values of the texture image being modified, and interlace is 1 if GL_INTERLACE_SGIX is disabled, and 2 otherwise. Note that TEXTURE_WIDTH and TEXTURE_HEIGHT include twice the border width.
- GL_INVALID_VALUE is generated if *width* or *height* is negative.
- GL_INVALID_OPERATION is generated when the texture array has not been defined by a previous glTexImage3D (or equivalent) operation.
- GL_INVALID_OPERATION is generated if glCopyTexSubImage3DEXT is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGetTexImage

See Also

glTexImage3D,
glTexSubImage3DEXT,
glCopyPixels.

glXCreateContext

`glXCreateContext`: create a new GLX rendering context.

C Specification

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

Parameters

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

Description

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

If *direct* is True, then a direct rendering context is created if the implementation supports direct rendering, if the connection is to an X server that is local, and if a direct rendering context is available. (An implementation may return an indirect context when *direct* is True). If *direct* is False, then a rendering context that renders through the X server is always created. Direct rendering provides a performance advantage in some implementations. However, direct rendering contexts cannot be shared outside a single process, and they may be unable to render to GLX pixmaps.

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

If the GL version is 1.1 or greater, then all texture objects except object 0, are shared by any contexts that share display lists.

Notes

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

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

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

It may not be possible to render to a GLX pixmap with a direct rendering context.

Errors

- NULL is returned if execution fails on the client side.
- *BadMatch* is generated if the context to be created would not share the address space or the screen of the context specified by *shareList*.
- *BadValue* is generated if *vis* is not a valid visual (for example, if a particular GLX implementation does not support it).
- GLXBadContext is generated if *shareList* is not a GLX context and is not NULL.
- *BadAlloc* is generated if the server does not have enough resources to allocate the new context.

See Also

glXDestroyContext,
glXGetConfig,
glXIsDirect,
glXMakeCurrent

glXCreateGLXPixmap

`glXCreateGLXPixmap`: create an off-screen GLX rendering area.

C Specification

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

Parameters

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

Description

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

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

Some implementations may not support GLX pixmaps with direct rendering contexts.

Notes

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

Errors

- BadMatch is generated if the depth of *pixmap* does not match the depth value reported by core X11 for *vis*, or if *pixmap* was not created with respect to the same screen as *vis*.
- BadValue is generated if *vis* is not a valid XVisualInfo pointer (for example, if a particular GLX implementation does not support this visual).
- BadPixmap is generated if *pixmap* is not a valid pixmap.

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

See Also

glXCreateContext,
glXIsDirect,
glXMakeCurrent

glCullFace

`glCullFace:` specify whether front- or back-facing facets can be culled.

C Specification

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

Parameters

mode	Specifies whether front- or back-facing 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

glCullFace specifies whether front- or back-facing facets are culled (as specified by *mode*) when facet culling is enabled. Facet culling is initially disabled. To enable and disable facet culling, call the glEnable and glDisable commands with the argument GL_CULL_FACE. Facets include triangles, quadrilaterals, polygons, and rectangles.

glFrontFace specifies which of the clockwise and counterclockwise facets are front-facing and back-facing. See glFrontFace.

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

See Also

glEnable,
glFrontFace

gluCylinder

`gluCylinder`: draw a cylinder.

C Specification

```
void gluCylinder(
    GLUquadric* quad,
    GLdouble base,
    GLdouble top,
    GLdouble height,
    GLint slices,
GLint stacks)
```

Parameters

quad	Specifies the quadrics object (created with gluNewQuadric).
base	Specifies the radius of the cylinder at z = 0.
top	Specifies the radius of the cylinder at z = *height*.
height	Specifies the height of the cylinder.
slices	Specifies the number of subdivisions around the Z axis.
stacks	Specifies the number of subdivisions along the Z axis.

Description

gluCylinder draws a cylinder oriented along the Z axis. The base of the cylinder is placed at Z = 0, and the top at Z = *height*. Like a sphere, a cylinder is subdivided around the Z axis into slices, and along the Z axis into stacks.

Note that if *top* is set to 0.0, this routine generates a cone.

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

If texturing is turned on (with gluQuadricTexture), then texture coordinates are generated so that t ranges linearly from 0.0 at Z = 0 to 1.0 at Z = *height*, and *s* ranges from 0.0 at the +Y axis, to 0.25 at the +X axis, to0.5 at the - Y axis, to 0.75 at the - X axis, and back to 1.0 at the +Y axis.

See Also

gluDisk,
gluNewQuadric,
gluPartialDisk,
gluQuadricTexture,
gluSphere

4 D

glDeleteLists

`glDeleteLists`: delete a contiguous group of display lists.

C Specification

```
void glDeleteLists(
    GLuint list,
    GLsizei range)
```

Parameters

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

Description

glDeleteLists causes a contiguous group of display lists to be deleted. *list* is the name of the first display list to be deleted, and *range* is the number of display lists to delete. All display lists d with $list \geq d \geq list + range - 1$ 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, nothing happens.

Errors

- GL_INVALID_VALUE is generated if *range* is negative.
- GL_INVALID_OPERATION is generated if glDeleteLists is executed between the execution of glBegin and the corresponding execution of glEnd.

See Also

glCallList,
glCallLists,
glGenLists,
glIsList,
glNewList

gluDeleteNurbsRenderer

`gluDeleteNurbsRenderer`: destroy a NURBS object.

C Specification

```
void gluDeleteNurbsRenderer(
    GLUnurbs* nurb)
```

Parameters

nurb	Specifies the NURBS object to be destroyed.

Description

gluDeleteNurbsRenderer destroys the NURBS object (which was created with gluNewNurbsRenderer) and frees any memory it uses. Once gluDeleteNurbsRenderer has been called, *nurb* cannot be used again.

See Also

gluNewNurbsRenderer

gluDeleteQuadric

`gluDeleteQuadric:` destroy a quadrics object.

C Specification

```
void gluDeleteQuadric(
    GLUquadric* quad)
```

Parameters

quad	Specifies the quadrics object to be destroyed.

Description

gluDeleteQuadric destroys the quadrics object (created with gluNewQuadric) and frees any memory it uses. Once gluDeleteQuadric has been called, *quad* cannot be used again.

See Also

gluNewQuadric

gluDeleteTess

`gluDeleteTess`: destroy a tessellation object.

C Specification

```
void gluDeleteTess(
    GLUtesselator* tess)
```

Parameters

tess	Specifies the tessellation object to destroy.

Description

gluDeleteTess destroys the indicated tessellation object (which was created with gluNewTess) and frees any memory that it used.

See Also

gluBeginPolygon,
gluNewTess,
gluTessCallback

glDeleteTextures

`glDeleteTextures`: delete named textures.

C Specification

```
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

glDeleteTextures 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 everts to 0 (the default texture).

glDeleteTextures silently ignores 0s and names that do not correspond to existing textures.

Notes

glDeleteTextures 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 glDeleteTextures is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glIsTexture

See Also

glAreTexturesResident,
glBindTexture,
glCopyTexImage1D,
glCopyTexImage2D,
glGenTextures,
glGet,
glGetTexParameter,
glPrioritizeTextures,

glTexImage1D,
glTexImage2D,
glTexParameter

glDepthFunc

`glDepthFunc`: specify the value used for depth buffer comparisons.

C Specification

```
void glDepthFunc(
    GLenum func)
```

Parameters

func	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 initial value is GL_LESS.

Description

glDepthFunc specifies the function used to compare each incoming pixel depth value with the depth value present in the depth buffer. The comparison is performed only if depth testing is enabled. (See glEnable and glDisable of GL_DEPTH_TEST.)

func 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 depth value is less than the stored depth value.

GL_EQUAL

Passes if the incoming depth value is equal to the stored depth value.

GL_LEQUAL

Passes if the incoming depth value is less than or equal to the stored depth value.

GL_GREATER

Passes if the incoming depth value is greater than the stored depth value.

GL_NOTEQUAL

Passes if the incoming depth value is not equal to the stored depth value.

GL_GEQUAL

Passes if the incoming depth value is greater than or equal to the stored depth value.

GL_ALWAYS

Always passes.

The initial value of *func* is GL_LESS. Initially, depth testing is disabled.

Notes

Even if the depth buffer exists and the depth mask is non-zero, the depth buffer is not updated if the depth test is disabled.

Errors

- GL_INVALID_ENUM is generated if *func* is not an accepted value.
- GL_INVALID_OPERATION is generated if glDepthFunc is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_DEPTH_FUNC
glIsEnabled with argument GL_DEPTH_TEST

See Also

glDepthRange,
glEnable,
glPolygonOffset

glDepthMask

`glDepthMask`: enable or disable writing into the depth buffer.

C Specification

```
void glDepthMask(
    GLboolean flag)
```

Parameters

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

Description

glDepthMask specifies whether the depth buffer is enabled for writing. If *flag* is GL_FALSE, depth buffer writing is disabled. Otherwise, it is enabled. Initially, depth buffer writing is enabled.

Errors

- GL_INVALID_OPERATION is generated if glDepthMask is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_DEPTH_WRITEMASK

See Also

glColorMask,
glDepthFunc,
glDepthRange,
glIndexMask,
glStencilMask

glDepthRange

`glDepthRange`: specify mapping of depth values from normalized device coordinates to window coordinates.

C Specification

```
void glDepthRange(
    GLclampd zNear,
    GLclampd zFar)
```

Parameters

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

Description

After clipping and division by w, depth coordinates range from - 1 to 1, corresponding to the near and far clipping planes. glDepthRange specifies a linear mapping of the normalized depth coordinates in this range to window depth coordinates. Regardless of the actual depth buffer implementation, window coordinate depth values are treated as though they range from 0 through 1 (like color components). Thus, the values accepted by glDepthRange are both clamped to this range before they are accepted.

The setting of (0, 1) maps the near plane to 0 and the far plane to 1. With this mapping, the depth buffer range is fully utilized.

Notes

It is not necessary that *zNear* be less than *zFar*.
Reverse mappings such as *zNear* = 1, and *zFar* = 0 are acceptable.

Errors

- GL_INVALID_OPERATION is generated if glDepthRange is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_DEPTH_RANGE

See Also

glDepthFunc,
glPolygonOffset,
glViewport

glXDestroyContext

`glXDestroyContext`: destroy a GLX context.

C Specification

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

Parameters

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

Description

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

Errors

- GLXBadContext is generated if *ctx* is not a valid GLX context.

See Also

glXCreateContext,
glXMakeCurrent

glXDestroyGLXPixmap

`glXDestroyGLXPixmap`: destroy a GLX pixmap.

C Specification

```
void glXDestroyGLXPixmap(
    Display *dpy,
    GLXPixmap pix)
```

Parameters

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

Description

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

Errors

- GLXBadPixmap is generated if *pix* is not a valid GLX pixmap.

See Also

glXCreateGLXPixmap,
glXMakeCurrent

glDisable

`glEnable, glDisable`: enable or disable server-side GL capabilities.

C Specification

```
void glEnable(
    GLenum cap)
void glDisable(
    GLenum cap)
```

Parameters

cap	Specifies a symbolic constant indicating a GL capability.

Description

glEnable and glDisable enable and disable various capabilities. Use glIsEnabled or glGet to determine the current setting of any capability. The initial value for each capability with the exception of GL_DITHER is GL_FALSE. The initial value for GL_DITHER is GL_TRUE.

Both glEnable and glDisable take a single argument, *cap*, which can assume one of the following values:

GL_ALPHA_TEST

If enabled, do alpha testing. See glAlphaFunc.

GL_AUTO_NORMAL

If enabled, generate normal vectors 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 RGBA color values with the values in the color buffers. See glBlendFunc.

GL_CLIP_PLANEi

If enabled, clip geometry against user-defined clipping plane i. See glClipPlane.

GL_COLOR_LOGIC_OP

If enabled, apply the currently selected logical operation to the incoming RGBA color and color buffer values. See glLogicOp.

GL_COLOR_MATERIAL

If enabled, have one or more material parameters track the current color. See glColorMaterial.

GL_CULL_FACE

If enabled, cull polygons based on their winding in window coordinates. See glCullFace.

GL_DEPTH_TEST

If enabled, do depth comparisons and update the depth buffer. Note that even if the depth buffer exists and the depth mask is non-zero, the depth buffer is not updated if the depth test is disabled. See glDepthFunc and glDepthRange.

GL_DITHER

If enabled, dither color components or indices before they are written to the color buffer.

GL_FOG

If enabled, blend a fog color into the post texturing color. See glFog.

GL_INDEX_LOGIC_OP

If enabled, apply the currently selected logical operation to the incoming index and color buffer indices. See glLogicOp.

GL_LIGHTi

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_MAP1_COLOR_4

If enabled, calls to glEvalCoord1, glEvalMesh1, and glEvalPoint1 generate RGBA values. See glMap1.

GL_MAP1_INDEX

If enabled, calls to glEvalCoord1, glEvalMesh1, and glEvalPoint1 generate color indices. See glMap1.

GL_MAP1_NORMAL

If enabled, calls to glEvalCoord1, glEvalMesh1, and glEvalPoint1 generate normals. See glMap1.

GL_MAP1_TEXTURE_COORD_1

If enabled, calls to glEvalCoord1, glEvalMesh1, and glEvalPoint1 generate *s* texture coordinates. See glMap1.

GL_MAP1_TEXTURE_COORD_2

If enabled, calls to glEvalCoord1, glEvalMesh1, and glEvalPoint1 generate *s* and *t* texture coordinates. See glMap1.

GL_MAP1_TEXTURE_COORD_3

If enabled, calls to glEvalCoord1, glEvalMesh1, and glEvalPoint1 generate *s*, *t*, and *r* texture coordinates. See glMap1.

GL_MAP1_TEXTURE_COORD_4

If enabled, calls to glEvalCoord1, glEvalMesh1, and glEvalPoint1 generate *s*, *t*, *r*, and *q* texture coordinates. See glMap1.

GL_MAP1_VERTEX_3

If enabled, calls to glEvalCoord1, glEvalMesh1, and glEvalPoint1 generate *x*, *y*, and *z* vertex coordinates. See glMap1.

GL_MAP1_VERTEX_4

If enabled, calls to glEvalCoord1, glEvalMesh1, and glEvalPoint1 generate homogeneous *x*, *y*, *z*, and *w* vertex coordinates. See glMap1.

GL_MAP2_COLOR_4

If enabled, calls to glEvalCoord2, glEvalMesh2, and glEvalPoint2 generate RGBA values. See glMap2.

GL_MAP2_INDEX

If enabled, calls to glEvalCoord2, glEvalMesh2, and glEvalPoint2 generate color indices. See glMap2.

GL_MAP2_NORMAL

If enabled, calls to glEvalCoord2, glEvalMesh2, and glEvalPoint2 generate normals. See glMap2.

GL_MAP2_TEXTURE_COORD_1

If enabled, calls to glEvalCoord2, glEvalMesh2, and glEvalPoint2 generate *s*

GL_MAP2_TEXTURE_COORD_2

If enabled, calls to glEvalCoord2, glEvalMesh2, and glEvalPoint2 generate *s* and *t* texture coordinates. See glMap2.

GL_MAP2_TEXTURE_COORD_3

If enabled, calls to glEvalCoord2, glEvalMesh2, and glEvalPoint2 generate *s*, *t*, and *r* texture coordinates. See glMap2.

GL_MAP2_TEXTURE_COORD_4

If enabled, calls to glEvalCoord2, glEvalMesh2, and glEvalPoint2 generate *s*, *t*, *r*, and *q* texture coordinates. See glMap2.

GL_MAP2_VERTEX_3

If enabled, calls to glEvalCoord2, glEvalMesh2, and glEvalPoint2 generate *x*, *y*, and *z* vertex coordinates. See glMap2.

GL_MAP2_VERTEX_4

If enabled, calls to glEvalCoord2, glEvalMesh2, and glEvalPoint2 generate homogeneous *x*, *y*, *z*, and *w* vertex coordinates. See glMap2.

GL_NORMALIZE

If enabled, normal vectors specified with glNormal are scaled to unit length after transformation. See glNormal.

GL_OCCLUSION_TEST_hp

This token enables HP's occlusion-testing extension. If any geometry is rendered while occlusion culling is enabled, and if that geometry would be visible (i.e., rendering it would affect the Z-buffer), the occlusion test state bit is set, indicating that the rendered object is visible. This is typically done to increase performance: if every pixel of a bounding box would be "behind" the current Z-buffer values for those pixels (i.e., the bounding box is entirely occluded), anything you would draw *within* that bounding box would also be behind the current Z values, and therefore you can cull it (i.e., avoid processing that geometry through the pipeline). Note that this enable has no effect on the current render mode, or any other OpenGL state.

GL_POINT_SMOOTH

If enabled, draw points with proper filtering. Otherwise, draw aliased points. See glPointSize.

GL_POLYGON_OFFSET_FILL

If enabled, and if the polygon is rendered in GL_FILL mode, an offset is added to depth values of a polygon's fragments before the depth comparison is performed. See glPolygonOffset.

GL_POLYGON_OFFSET_LINE

If enabled, and if the polygon is rendered in GL_LINE mode, an offset is added to depth values of a polygon's fragments before the depth comparison is performed. See glPolygonOffset.

GL_POLYGON_OFFSET_POINT

If enabled, an offset is added to depth values of a polygon's fragments before the depth comparison is performed, if the polygon is rendered in GL_POINT mode. See glPolygonOffset.

GL_POLYGON_SMOOTH

If enabled, draw polygons with proper filtering. Otherwise, draw aliased polygons. For correct anti-aliased polygons, an alpha buffer is needed and the polygons must be sorted front to back.

GL_POLYGON_STIPPLE

If enabled, use the current polygon stipple pattern when rendering polygons. See glPolygonStipple.

GL_RESCALE_NORMAL_EXT

When 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. The rescale factor is chosen so that in many cases normal vectors with unit length in object coordinates will not need to be normalized as they are transformed into eye coordinates.

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_EXT

If supported and enabled, three-dimensional texturing is performed. See glTexImage3DEXT.

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.

Notes

GL_POLYGON_OFFSET_FILL, GL_POLYGON_OFFSET_LINE, GL_POLYGON_OFFSET_POINT, GL_COLOR_LOGIC_OP, and GL_INDEX_LOGIC_OP are only available if the GL version is 1.1 or greater.

Errors

- GL_INVALID_ENUM is generated if *cap* is not one of the values listed previously.
- GL_INVALID_OPERATION is generated if glEnable or glDisable is executed between the execution of glBegin and the corresponding execution of glEnd.

See Also

glAlphaFunc,
glBlendFunc,
glClipPlane,
glColorMaterial,
glCullFace,
glDepthFunc,
glDepthRange,

glEnableClientState,
glFog,
glGet,
glIsEnabled,
glLight,
glLightModel,
glLineWidth,
glLineStipple,
glLogicOp,
glMap1,
glMap2,
glMaterial,
glNormal,
glPointSize,
glPolygonMode,
glPolygonOffset,
glPolygonStipple,
glScissor,
glStencilFunc,
glStencilOp,
glTexGen,
glTexImage1D,
glTexImage2D

gluDisk

gluDisk: draw a disk.

C Specification

```
void gluDisk(
    GLUquadric* quad,
    GLdouble inner,
    GLdouble outer,
    GLint slices,
    GLint loops)
```

Parameters

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

Description

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

With respect to orientation, the +*z* side of the disk is considered to be “outside” (see gluQuadricOrientation).

This means that if the orientation is set to GLU_OUTSIDE, then any normals generated point along the +Z axis. Otherwise, they point along the - Z axis.

If texturing has been turned on (with gluQuadricTexture), texture coordinates are generated linearly such that where $r = outer$, the value at $(r, 0, 0)$ is (1, 0.5), at $(0, r, 0)$ it is (0.5, 1), at $(-r, 0, 0)$ it is (0, 0.5), and at $(0, -r, 0)$ it is (0.5, 0).

See Also

gluCylinder,
gluNewQuadric,
gluPartialDisk,
gluQuadricOrientation,
gluQuadricTexture,
gluSphere

glDrawArrays

`glDrawArrays`: render primitives from array data.

C Specification

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

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.

Description

glDrawArrays specifies 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 pre-specify 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. mode* specifies what kind of primitives are constructed, and how the array elements construct those 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 aren't modified remain well defined.

Notes

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

glDrawArrays 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.

See Also

glArrayElement,
glColorPointer,
glDrawElements,
glEdgeFlagPointer,
glGetPointer,
glIndexPointer,
glInterleavedArrays,
glNormalPointer,
glTexCoordPointer,
glVertexPointer

glDrawArraysSethp

`glDrawArraySethp`: render multiple primitives from array data.

C Specification

```
void glDrawArraySethp(
    GLenum mode,
    const GLint* list,
    GLsizei count)
```

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.
list	Points to an array of starting indices in the enabled arrays.
count	Specifies the number of groups of primitives to be rendered.

Description

glDrawArraySethp specifies multiple geometric primitives of the same type with a single subroutine call. Instead of calling a GL procedure to pass each individual vertex, normal, texture coordinate, edge flag, or color, you can use the vertex array calls to pre-specify arrays of vertices, normals, and colors and use them to construct a sequence of primitives with a single call to glDrawArraySethp.

When glDrawArraySethp is called, it iterates over *count*+1 array indices from the list. For 0 ≥ i<*count*, glDrawArraySethp uses *list*[*i*+1] - *list*[*i*] sequential elements from each enabled array to construct a sequence of geometric primitives, beginning with element *list*[*i*]. *mode* specifies what kind of primitives are constructed, and how the array elements construct those primitives. If GL_VERTEX_ARRAY is not enabled, no geometric primitives are generated.

Vertex attributes that are modified by glDrawArraySethp have an unspecified value after glDrawArraySethp returns. For example, if GL_C4UB_V3F is enabled, the value of the current color is undefined after glDrawArraySethp executes. Attributes that aren't modified remain well defined.

glDrawArraySethp(*mode, list, count*) is functionally equivalent to:

```
for (i = 0; i < count; i++)
    glDrawArrays(mode, list[i], list[i+1] - list[i]);
```

The behavior is undefined if *list*[*i*+1] is less than *list*[*i*] for any *i* in the range *i* ≥ 0 and *i*<*count*.

Notes

glDrawArraySethp is a Hewlett-Packard GL version 1.1 extension.

glDrawArraySethp is included in display lists. If glDrawArraySethp 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 glDrawArraySethp is executed between the execution of glBegin and the corresponding glEnd.

See Also

glArrayElement,
glColorPointer,
glDrawArrays,
glDrawElements,
glEdgeFlagPointer,
glGetPointer,
glIndexPointer,
glInterleavedArrays,
glNormalPointer,
glTexCoordPointer,
glVertexPointer

glDrawBuffer

`glDrawBuffer`: specify which color buffers are to be drawn into.

C Specification

```
void glDrawBuffer(
    GLenum mode)
```

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 initial value is GL_FRONT for single-buffered contexts, and GL_BACK for double-buffered contexts.

Description

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

GL_NONE

No color buffers are written.

GL_FRONT_LEFT

Only the front left color buffer is written.

GL_FRONT_RIGHT

Only the front right color buffer is written.

GL_BACK_LEFT

Only the back left color buffer is written.

GL_BACK_RIGHT

Only the back right color buffer is written.

GL_FRONT

Only the front left and front right color buffers are written. If there is no front right color buffer, only the front left color buffer is written.

GL_BACK

Only the back left and back right color buffers are written. If there is no back right color buffer, only the back left color buffer is written.

GL_LEFT

Only the front left and back left color buffers are written. If there is no back left color buffer, only the front left color buffer is written.

GL_RIGHT

Only the front right and back right color buffers are written. If there is no back right color buffer, only the front right color buffer is written.

GL_FRONT_AND_BACK

All the front and back color buffers (front left, front right, back left, back right) are written. If there are no back color buffers, only the front left and front right color buffers are written. If there are no right color buffers, only the front left and back left color buffers are written. If there are no right or back color buffers, only the front left color buffer is written. GL_AUXi Only auxiliary color buffer *i* is written.

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

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

Notes

It is always the case that GL_AUX*i* = GL_AUX0 + *i*.

Errors

- GL_INVALID_ENUM is generated if *mode* is not an accepted value.
- GL_INVALID_OPERATION is generated if none of the buffers indicated by *mode* exists.
- GL_INVALID_OPERATION is generated if glDrawBuffer is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_DRAW_BUFFER
glGet with argument GL_AUX_BUFFERS

See Also

glBlendFunc,
glColorMask,
glIndexMask,
glLogicOp,
glReadBuffer

glDrawElements

`glDrawElements`: render primitives from array data.

C Specification

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

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.

Description

glDrawElements specifies 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 pre-specify 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 more than one array is enabled, each is used. If GL_VERTEX_ARRAY is not enabled, no geometric primitives are constructed.

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

Notes

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

glDrawElements 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.

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 ifglDrawElements is executed between the execution of glBegin and the corresponding glEnd.

See Also

glArrayElement,
glColorPointer,
glDrawArrays,
glEdgeFlagPointer,
glGetPointer,
glIndexPointer,
glInterleavedArrays,
glNormalPointer,
glTexCoordPointer,
glVertexPointer

glDrawPixels

`glDrawPixels`: write a block of pixels to the frame buffer.

C Specification

```
void glDrawPixels(
    GLsizei      width,
    GLsizei      height,
    GLenum       format,
    GLenum       type,
    const GLvoid *pixels)
```

Parameters

width, height	Specify the dimensions of the pixel rectangle to 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_RGBA, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_LUMINANCE, 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, and GL_FLOAT are accepted.
pixels	Specifies a pointer to the pixel data.

Description

glDrawPixels 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; use glGet with argument GL_CURRENT_RASTER_POSITION to query the raster position.

Several parameters define the encoding of pixel data in memory and control the processing of the pixel data before it is placed in the frame buffer. These parameters are set with four commands: glPixelStore, glPixelTransfer, glPixelMap, and glPixelZoom. This reference page describes the effects on glDrawPixels of many, but not all, of the parameters specified by these four commands.

Data is read from *pixels* as a sequence of signed or unsigned bytes, signed or unsigned shorts, signed or unsigned integers, or single-precision floating-point values, depending on *type*. Each of these bytes, shorts, integers, or floating-point values is interpreted as one color or depth component, or one index, depending on *format*. 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 × *height* pixels are read from memory, starting at location *pixels*. By default, these pixels are taken from adjacent memory locations, except that after all *width* pixels are read, the read pointer is advanced to the next four-byte boundary. The four-byte row alignment is specified by glPixelStore with argument GL_UNPACK_ALIGNMENT, and it can be set to one, two, four, or eight bytes. Other pixel store parameters specify different read pointer advancements, both before the first pixel is read and after all *width* pixels are read. See the glPixelStore reference page for details on these options.

The *width* × *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 are drawn, are specific to the format of the pixels, as specified by *format*. *format* can assume one of eleven symbolic values:

GL_COLOR_INDEX

Each pixel is a single value, a color index. It is converted to fixed-point format, with an unspecified number of bits to the right of the binary point, regardless of the memory data type. Floating-point values convert to true fixed-point values. Signed and unsigned integer data is converted with all fraction bits set to 0. Bitmap data convert to either 0 or 1.

Each fixed-point index is then shifted left by GL_INDEX_SHIFT bits and added to GL_INDEX_OFFSET. If GL_INDEX_SHIFT is negative, the shift is to the right. In either case, zero bits fill otherwise unspecified bit locations in the result.

If the GL is in RGBA mode, the resulting index is converted to an RGBA pixel with the help of 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 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 $2^b - 1$, where *b* is the number of bits in a color index buffer.

The GL then converts the resulting indices or RGBA colors to fragments by attaching the current raster position *z* coordinate and texture coordinates to each pixel, then assigning *x* and *y* window coordinates to the *n*th fragment such that

$x_n = x_r + n \bmod width$

$y_n = y_r + \lfloor n/width \rfloor$

where (x_r, y_r) is the current raster position.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 format, with an unspecified number of bits to the right of the binary point, regardless of the memory data type. Floating-point values convert to true fixed-point values. Signed and unsigned integer data is converted with all fraction bits set to 0. Bitmap data convert to either 0 or 1.

Each fixed-point index is then shifted left by GL_INDEX_SHIFT bits, and added to GL_INDEX_OFFSET. If GL_INDEX_SHIFT is negative, the shift is to the right. In either case, zero bits fill otherwise unspecified bit locations in the result. If

GL_MAP_STENCIL is true, the index is replaced with the value that it references in lookup table GL_PIXEL_MAP_S_TO_S. Whether the lookup replacement of the index is done or not, the integer part of the index is then ANDed with 2^b - 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 nth index is written to location

$x_n = x_r + n \bmod width$

$y_n = y_r + \lfloor n/width \rfloor$

where (x_r, y_r) is the current raster position. Only the pixel ownership test, the scissor test, and the stencil writemask affect these 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 GL then converts the resulting depth components 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 nth fragment such that

$x_n = x_r + n \bmod width$

$y_n = y_r + \lfloor n/width \rfloor$

where (x_r, y_r) is the current raster position. 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: for GL_RGBA, the red component is 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. (Note that this mapping does not convert 0 precisely to 0.0.) Unsigned integer data is mapped similarly: the largest integer value maps to 1.0, and 0 maps to 0.0. The resulting floating-point 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 GL then converts the resulting RGBA colors to fragments by attaching the current raster position Z coordinate and texture coordinates to each pixel, then assigning *x* and *y* window coordinates to the nth fragment such that

$x_n = x_r + n \bmod width$

$y_n = y_r + \lfloor n/width \rfloor$

where (x_r, y_r) is the current raster position.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 the red component of an RGBA pixel is. It is then converted to an RGBA pixel with green and blue set to 0, and alpha set to 1. After this conversion, the pixel is treated 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 the green component of an RGBA pixel is. It is then converted to an RGBA pixel with red and blue set to 0, and alpha set to 1. After this conversion, the pixel is treated 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 the blue component of an RGBA pixel is. It is then converted to an RGBA pixel with red and green set to 0, and alpha set to 1. After this conversion, the pixel is treated 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 the alpha component of an RGBA pixel is. It is then converted to an RGBA pixel with red, green, and blue set to 0. After this conversion, the pixel is treated 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 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. After this conversion, the pixel is treated as if it had been read as an RGBA pixel.

GL_LUMINANCE

Each pixel is a single luminance component. This component is converted to the internal floating-point format in the same way the red component of an RGBA pixel is. It is then converted to an RGBA pixel with red, green, and blue set to the converted luminance value, and alpha set to 1. After this conversion, the pixel is treated 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 the red component of an RGBA pixel is. They are then 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 as if it had been read as an RGBA pixel.

The following table summarizes the meaning of the valid constants for the *type* parameter:

Table 4-1

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 integers
GL_SHORT	Signed 16-bit integers
GL_UNSIGNED_INT	Unsigned 32-bit integers
GL_INT	32-bit integer
GL_FLOAT	Single-precision floating-point

The rasterization described so far assumes pixel zoom factors of 1. 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, then fragments are generated for pixels whose centers are in the rectangle with corners at

$(x_r + zoom_x n, y_r + zoom_y m)$

$(x_r + zoom_x (n + 1), y_r + zoom_y (m + 1))$

where $zoom_x$ is the value of GL_ZOOM_X and $zoom_y$ is the value of GL_ZOOM_Y.

Errors

- GL_INVALID_VALUE is generated if either *width* or *height* is negative.
- GL_INVALID_ENUM is generated if *format* or *type* is not one of the accepted values.
- GL_INVALID_OPERATION is generated if *format* is GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_LUMINANCE, or GL_LUMINANCE_ALPHA, and the GL is in color index mode.
- GL_INVALID_ENUM is generated if *type* is GL_BITMAP and *format* is not either GL_COLOR_INDEX or GL_STENCIL_INDEX.
- GL_INVALID_OPERATION is generated if *format* is GL_STENCIL_INDEX and there is no stencil buffer.
- GL_INVALID_OPERATION is generated if glDrawPixels is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID

See Also

glAlphaFunc,
glBlendFunc,
glCopyPixels,
glDepthFunc,
glLogicOp,
glPixelMap,
glPixelStore,
glPixelTransfer,
glPixelZoom,
glRasterPos,
glReadPixels,
glScissor,
glStencilFunc

5 E

glEdgeFlag

`glEdgeFlag, glEdgeFlagv`: flag edges as either boundary or non-boundary.

C Specification

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

Parameters

flag	Specifies the current edge flag value, either GL_TRUE or GL_FALSE. The initial value is GL_TRUE.
flag	Specifies a pointer to an array that contains a single boolean element, which replaces the current edge flag value.

Description

Each vertex of a polygon, separate triangle, or separate quadrilateral specified between a glBegin/glEnd pair is marked as the start of either a boundary or non-boundary 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 non-boundary edge. glEdgeFlag sets the edge flag bit to GL_TRUE if *flag* is GL_TRUE, and to GL_FALSE otherwise.

The vertices of connected triangles and connected quadrilaterals are always marked as boundary, regardless of the value of the edge flag.

Boundary and non-boundary edge flags on vertices are significant only if GL_POLYGON_MODE is set toGL_POINT or GL_LINE. See glPolygonMode.

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

glGet with argument GL_EDGE_FLAG

See Also

glBegin,
glEdgeFlagPointer,
glPolygonMode

glEdgeFlagPointer

`glEdgeFlagPointer`: define an array of edge flags.

C Specification

```
void glEdgeFlagPointer(
    GLsizei stride
    const GLvoid *pointer)
```

Parameters

stride	Specifies the byte offset between consecutive edge flags. If *stride* is 0 (the initial value), the edge flags are understood to be tightly packed in the array.
pointer	Specifies a pointer to the first edge flag in the array.

Description

glEdgeFlagPointer specifies the location and data format of an array of boolean edge flags to use when rendering. *stride* specifies 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; 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 to construct a sequence of primitives (all of the same type) from pre-specified vertex and vertex attribute arrays. Use glArrayElement to specify primitives by indexing vertexes and vertex attributes and glDrawElements to construct a sequence of primitives by indexing vertexes and vertex attributes.

Notes

glEdgeFlagPointer 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 the execution of glBegin and the corresponding execution of glEnd, but an error may or may not be generated. If no error is generated, the operation is undefined.

glEdgeFlagPointer is typically implemented on the client side.

Edge flag array parameters are client-side state and are therefore not saved or restored by glPushAttrib and glPopAttrib. Use glPushClientAttrib and glPopClientAttrib instead.

Errors

- 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
glGetPointer with argument GL_EDGE_FLAG_ARRAY_POINTER

See Also

glArrayElement,
glColorPointer,
glDrawArrays,
glDrawElements,
glEnable,
glGetPointer,
glIndexPointer,
glNormalPointer,
glPopClientAttrib,
glPushClientAttrib,
glTexCoordPointer,
glVertexPointer

glEnable

`glEnable, glDisable`: enable or disable server-side GL capabilities.

C Specification

```
void glEnable(
    GLenum cap)
void glDisable(
    GLenum cap)
```

Parameters

cap	Specifies a symbolic constant indicating a GL capability.

Description

glEnable and glDisable enable and disable various capabilities. Use glIsEnabled or glGet to determine the current setting of any capability. The initial value for each capability with the exception of GL_DITHER is GL_FALSE. The initial value for GL_DITHER is GL_TRUE.

Both glEnable and glDisable take a single argument, cap, which can assume one of the following values:

GL_ALPHA_TEST

If enabled, do alpha testing. See glAlphaFunc.

GL_AUTO_NORMAL

If enabled, generate normal vectors 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 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_LOGIC_OP

If enabled, apply the currently selected logical operation to the incoming RGBA color and color buffer values. See glLogicOp.

GL_COLOR_MATERIAL

If enabled, have one or more material parameters track the current color. See glColorMaterial.

GL_CULL_FACE

If enabled, cull polygons based on their winding in window coordinates. See glCullFace.

GL_DEPTH_TEST

If enabled, do depth comparisons and update the depth buffer. Note that even if the depth buffer exists and the depth mask is non-zero, the depth buffer is not updated if the depth test is disabled. See glDepthFunc and glDepthRange.

GL_DITHER

If enabled, dither color components or indices before they are written to the color buffer.

GL_FOG

If enabled, blend a fog color into the post texturing color. See glFog.

GL_INDEX_LOGIC_OP

If enabled, apply the currently selected logical operation to the incoming index and color buffer indices. See glLogicOp.

GL_LIGHTi

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_MAP1_COLOR_4

If enabled, calls to glEvalCoord1, glEvalMesh1, and glEvalPoint1 generate RGBA values. See glMap1.

GL_MAP1_INDEX

If enabled, calls to glEvalCoord1, glEvalMesh1, and glEvalPoint1 generate color indices. See glMap1.

GL_MAP1_NORMAL

If enabled, calls to glEvalCoord1, glEvalMesh1, and glEvalPoint1 generate normals. See glMap1.

GL_MAP1_TEXTURE_COORD_1

If enabled, calls to glEvalCoord1, glEvalMesh1, and glEvalPoint1 generate *s* texture coordinates. See glMap1.

GL_MAP1_TEXTURE_COORD_2

If enabled, calls to glEvalCoord1, glEvalMesh1, and glEvalPoint1 generate *s* and *t* texture coordinates. See glMap1.

GL_MAP1_TEXTURE_COORD_3

If enabled, calls to glEvalCoord1, glEvalMesh1, and glEvalPoint1 generate *s*, *t*, and *r* texture coordinates. See glMap1.

GL_MAP1_TEXTURE_COORD_4

If enabled, calls to glEvalCoord1, glEvalMesh1, and glEvalPoint1 generate *s*, *t*, *r*, and *q* texture coordinates. See glMap1.

GL_MAP1_VERTEX_3

If enabled, calls to glEvalCoord1, glEvalMesh1, and glEvalPoint1 generate *x*, *y*, and *z* vertex coordinates. See glMap1.

GL_MAP1_VERTEX_4

If enabled, calls to glEvalCoord1, glEvalMesh1, and glEvalPoint1 generate homogeneous *x*, *y*, *z*, and *w* vertex coordinates. See glMap1.

GL_MAP2_COLOR_4

If enabled, calls to glEvalCoord2, glEvalMesh2, and glEvalPoint2 generate RGBA values. See glMap2.

GL_MAP2_INDEX

If enabled, calls to glEvalCoord2, glEvalMesh2, and glEvalPoint2 generate color indices. See glMap2.

GL_MAP2_NORMAL

If enabled, calls to glEvalCoord2, glEvalMesh2, and glEvalPoint2 generate normals. See glMap2.

GL_MAP2_TEXTURE_COORD_1

If enabled, calls to glEvalCoord2, glEvalMesh2, and glEvalPoint2 generate *s*

GL_MAP2_TEXTURE_COORD_2

If enabled, calls to glEvalCoord2, glEvalMesh2, and glEvalPoint2 generate *s* and *t* texture coordinates. See glMap2.

GL_MAP2_TEXTURE_COORD_3

If enabled, calls to glEvalCoord2, glEvalMesh2, and glEvalPoint2 generate *s*, *t*, and *r* texture coordinates. See glMap2.

GL_MAP2_TEXTURE_COORD_4

If enabled, calls to glEvalCoord2, glEvalMesh2, and glEvalPoint2 generate *s*, *t*, *r*, and *q* texture coordinates. See glMap2.

GL_MAP2_VERTEX_3

If enabled, calls to glEvalCoord2, glEvalMesh2, and glEvalPoint2 generate *x*, *y*, and *z* vertex coordinates. See glMap2.

GL_MAP2_VERTEX_4

If enabled, calls to glEvalCoord2, glEvalMesh2, and glEvalPoint2 generate homogeneous *x*, *y*, *z*, and *w* vertex coordinates. See glMap2.

GL_NORMALIZE

If enabled, normal vectors specified with glNormal are scaled to unit length after transformation. See glNormal.

GL_OCCLUSION_TEST_hp

This token enables HP's occlusion-testing extension. If any geometry is rendered while occlusion culling is enabled, and if that geometry would be visible (i.e., rendering it would affect the Z-buffer), the occlusion test state bit is set, indicating that the rendered object is visible. This is typically done to increase performance: if every pixel of a bounding box would be "behind" the current Z-buffer values for those pixels (i.e., the bounding box is entirely occluded), anything you would draw *within* that bounding box would also be behind the current Z values, and therefore you can cull it (i.e., avoid processing that geometry through the pipeline). Note that this enable has no effect on the current render mode, or any other OpenGL state.

GL_POINT_SMOOTH

f enabled, draw points with proper filtering. Otherwise, draw aliased points. See glPointSize.

GL_POLYGON_OFFSET_FILL

If enabled, and if the polygon is rendered in GL_FILL mode, an offset is added to depth values of a polygon's fragments before the depth comparison is performed. See glPolygonOffset.

GL_POLYGON_OFFSET_LINE

If enabled, and if the polygon is rendered in GL_LINE mode, an offset is added to depth values of a polygon's fragments before the depth comparison is performed. See glPolygonOffset.

GL_POLYGON_OFFSET_POINT

If enabled, an offset is added to depth values of a polygon's fragments before the depth comparison is performed, if the polygon is rendered in GL_POINT mode. See glPolygonOffset.

GL_POLYGON_SMOOTH

If enabled, draw polygons with proper filtering. Otherwise, draw aliased polygons. For correct anti-aliased polygons, an alpha buffer is needed and the polygons must be sorted front to back.

GL_POLYGON_STIPPLE

If enabled, use the current polygon stipple pattern when rendering polygons. See glPolygonStipple.

GL_RESCALE_NORMAL_EXT

When 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. The rescale factor is chosen so that in many cases normal vectors with unit length in object coordinates will not need to be normalized as they are transformed into eye coordinates.

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_EXT

If supported and enabled, three-dimensional texturing is performed. See glTexImage3DEXT.

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.

Notes

GL_POLYGON_OFFSET_FILL, GL_POLYGON_OFFSET_LINE, GL_POLYGON_OFFSET_POINT, GL_COLOR_LOGIC_OP, and GL_INDEX_LOGIC_OP are only available if the GL version is 1.1 or greater.

Errors

- GL_INVALID_ENUM is generated if *cap* is not one of the values listed previously.
- GL_INVALID_OPERATION is generated if glEnable or glDisable is executed between the execution of glBegin and the corresponding execution of glEnd.

See Also

glAlphaFunc,
glBlendFunc,
glClipPlane,
glColorMaterial,
glCullFace,
glDepthFunc,
glDepthRange,
glEnableClientState,

glFog,
glGet,
glIsEnabled,
glLight,
glLightModel,
glLineWidth,
glLineStipple,
glLogicOp,
glMap1,
glMap2,
glMaterial,
glNormal,
glPointSize,
glPolygonMode,
glPolygonOffset,
glPolygonStipple,
glScissor,
glStencilFunc,
glStencilOp,
glTexGen,
glTexImage1D,
glTexImage2D

glEnableClientState

`glEnableClientState, glDisableClientState`: enable or disable client-side capability.

C Specification

```
void glEnableClientState(
    GLenum cap)
void glDisableClientState(
    GLenum cap)
```

Parameters

cap	Specifies the capability to enable. Symbolic constants GL_COLOR_ARRAY, GL_EDGE_FLAG_ARRAY, GL_INDEX_ARRAY, GL_NORMAL_ARRAY, GL_TEXTURE_COORD_ARRAY, and GL_VERTEX_ARRAY are accepted.
cap	Specifies the capability to disable.

Description

glEnableClientState and glDisableClientState enable or disable individual client-side capabilities. By default, all client-side capabilities are disabled. Both glEnableClientState and glDisableClientState take a single argument, *cap*, which can assume one of the following values:

GL_COLOR_ARRAY

If enabled, the color array is enabled for writing and used during rendering when glDrawArrays or glDrawElements is called. See glColorPointer.

GL_EDGE_FLAG_ARRAY

If enabled, the edge flag array is enabled for writing and used during rendering when glDrawArrays or glDrawElements is called. See glEdgeFlagPointer.

GL_INDEX_ARRAY

If enabled, the index array is enabled for writing and used during rendering when glDrawArrays or glDrawElements is called. See glIndexPointer.

GL_NORMAL_ARRAY

If enabled, the normal array is enabled for writing and used during rendering when glDrawArrays or glDrawElements is called. See glNormalPointer.

GL_TEXTURE_COORD_ARRAY

If enabled, the texture coordinate array is enabled for writing and used for rendering when glDrawArrays or glDrawElements is called. See glTexCoordPointer.

GL_VERTEX_ARRAY

If enabled, the vertex array is enabled for writing and used during rendering when glDrawArrays or glDrawElements is called. See glVertexPointer.

Notes

glEnableClientState is available only if the GL version is 1.1 or greater.

Errors

- GL_INVALID_ENUM is generated if *cap* is not an accepted value. glEnableClientState 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, the behavior is undefined.

See Also

glArrayElement,
glColorPointer,
glDrawArrays,
glDrawElements,
glEdgeFlagPointer,
glEnable,
glGetPointer,
glIndexPointer,
glInterleavedArrays,
glNormalPointer,
glTexCoordPointer,
glVertexPointer

glErrorString

`gluErrorString`: produce an error string from a GL or GLU error code.

C Specification

```
constGLubyte *gluErrorString(
    GLenum error)
```

Parameters

error	Specifies a GL or GLU error code.

Description

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

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

See Also

glGetError,
gluNurbsCallback,
luQuadricCallback,
gluTessCallback

glEvalCoord

`glEvalCoord1d, glEvalCoord1f, glEvalCoord2d, glEvalCoord2f, glEvalCoord1dv, glEvalCoord1fv, glEvalCoord2dv, glEvalCoord2fv`: evaluate enabled one- and two-dimensional maps.

C Specification

```
void glEvalCoord1d(
    GLdouble u)
void glEvalCoord1f(
    GLfloat u)
void glEvalCoord2d(
    GLdouble u,
    GLdouble v)
void glEvalCoord2f(
    GLfloat u,
    GLfloat v)
void glEvalCoord1dv(
    const GLdouble *u)
void glEvalCoord1fv(
    const GLfloat *u)
void glEvalCoord2dv(
    const GLdouble *u)
void glEvalCoord2fv(
    const GLfloat *u)
```

Parameters

u	Specifies a value that is the domain coordinate *u* to the basis function defined in a previous glMap1 or glMap2 command.
v	Specifies a value that is the domain coordinate *v* to the basis function defined in a previous glMap2 command. This argument is not present in a glEvalCoord1 command.
u	Specifies a pointer to an array containing either one or two domain coordinates. The first coordinate is *u*. The second coordinate is *v*, which is present only in glEvalCoord2 versions.

Description

glEvalCoord1 evaluates enabled one-dimensional maps at argument *u*. glEvalCoord2 does the same for two-dimensional maps using two domain values, *u* and *v*. To define a map, call glMap1 and glMap2; to enable and disable it, call glEnable and glDisable.

When one of the glEvalCoord commands is issued, all currently enabled maps of the indicated dimension are evaluated. Then, for each enabled map, it is as if the corresponding GL command had been issued with the computed value. That is, if GL_MAP1_INDEX or GL_MAP2_INDEX is enabled, a glIndex command is simulated. If GL_MAP1_COLOR_4 or GL_MAP2_COLOR_4 is enabled, a glColor command 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, then an appropriate glTexCoord command is simulated.

For color, color index, normal, and texture coordinates the GL uses evaluated values instead of current values for those evaluations that are enabled, and current values otherwise, However, the evaluated values do not update the current values. Thus, if glVertex commands are interspersed with glEvalCoord commands, the color, normal, and texture coordinates associated with the glVertex commands are not affected by the values generated by the glEvalCoord commands, but only by the most recent glColor, glIndex, glNormal, and glTexCoord commands.

No commands are issued for maps that are not enabled. If more than one texture evaluation is enabled for a particular dimension (for example, GL_MAP2_TEXTURE_COORD_1 and GL_MAP2_TEXTURE_COORD_2), then only the evaluation of the map that produces the larger number of coordinates (in this case, GL_MAP2_TEXTURE_COORD_2) is carried out. GL_MAP1_VERTEX_4 overrides GL_MAP1_VERTEX_3, and GL_MAP2_VERTEX_4 overrides GL_MAP2_VERTEX_3, in the same manner. If neither a three- nor a four-component vertex map is enabled for the specified dimension, the glEvalCoord command is ignored.

If you have enabled automatic normal generation, 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

$$\mathbf{m} = \frac{\partial \mathbf{p}}{\partial \mathbf{u}} \times \frac{\partial \mathbf{p}}{\partial \mathbf{v}}$$

Then the generated normal *n* is

$$\mathbf{n} = \frac{\mathbf{m}}{\|\mathbf{m}\|}$$

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 commands.

Associated Gets

glIsEnabled with argument GL_MAP1_VERTEX_3
glIsEnabled with argument GL_MAP1_VERTEX_4
glIsEnabled with argument GL_MAP1_INDEX
glIsEnabled with argument GL_MAP1_COLOR_4
glIsEnabled with argument GL_MAP1_NORMAL
glIsEnabled with argument GL_MAP1_TEXTURE_COORD_1
glIsEnabled with argument GL_MAP1_TEXTURE_COORD_2
glIsEnabled with argument GL_MAP1_TEXTURE_COORD_3
glIsEnabled with argument GL_MAP1_TEXTURE_COORD_4
glIsEnabled with argument GL_MAP2_VERTEX_3
glIsEnabled with argument GL_MAP2_VERTEX_4

glIsEnabled with argument GL_MAP2_INDEX
glIsEnabled with argument GL_MAP2_COLOR_4
glIsEnabled with argument GL_MAP2_NORMAL
glIsEnabled with argument GL_MAP2_TEXTURE_COORD_1
glIsEnabled with argument GL_MAP2_TEXTURE_COORD_2
glIsEnabled with argument GL_MAP2_TEXTURE_COORD_3
glIsEnabled with argument GL_MAP2_TEXTURE_COORD_4
glIsEnabled with argument GL_AUTO_NORMAL
glGetMap

See Also

glBegin,
glColor,
glEnable,
glEvalMesh,
glEvalPoint,
glIndex,
glMap1,
glMap2,
glMapGrid,
glNormal,
glTexCoord,
glVertex

glEvalMesh

`glEvalMesh1, glEvalMesh2`: compute a one- or two-dimensional grid of points or lines.

C Specification

```
void glEvalMesh1(
     GLenum mode,
     GLint i1,
     GLint i2)
void glEvalMesh2(
     GLenum mode,
     GLint i1,
     GLint i2,
     GLint j1,
     GLint j2)
```

Parameters

mode	In glEvalMesh1, specifies whether to compute a one-dimensional mesh of points or lines. Symbolic constants GL_POINT and GL_LINE are accepted.
i1, i2	Specify the first and last integer values for grid domain variable *i*.
mode	In glEvalMesh2, specifies whether to compute a two-dimensional mesh of points, lines, or polygons. Symbolic constants GL_POINT, GL_LINE, and GL_FILL are accepted.
i1, i2	Specify the first and last integer values for grid domain variable *i*.
j1, j2	Specify the first and last integer values for grid domain variable *j*.

Description

glMapGrid and glEvalMesh are used in tandem to efficiently generate and evaluate a series of evenly-spaced map domain values. glEvalMesh steps through the integer domain of a one- or two-dimensional grid, whose range is the domain of the evaluation maps specified by glMap1 and glMap2. mode determines whether the resulting vertices are connected as points, lines, or filled polygons.

In the one-dimensional case, glEvalMesh1, the mesh is generated as if the following code fragment were executed:

```
glBegin (type);
for (i = i1; i ≤ i2; i += 1)
   glEvalCoord1(i·$\Delta u + u_1$)
glEnd();
```

where

$\Delta u = (u_2 - u_1) / n$

and *n*, u_1, and u_2 are the arguments to the most recent glMapGrid1 command. *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, then the value computed from $i–\Delta u + u_1$ is exactly u_2.

In the two-dimensional case, glEvalMesh2, let

$\Delta u = (u_2 – u_1) / n$
$\Delta v = (v_2 – v_1) / m$,

where *n*, u_1, u_2, *m*, v_1, and v_2 are the arguments to the most recent glMapGrid2 command. Then, if *mode* is GL_FILL, the glEvalMesh2 command is equivalent to:

```
for (j = j1;  j < j2; j += 1) {
    glBegin (GL_QUAD_STRIP);
for (i = i1; i ≤ i2; i += 1) {
    glEvalCoord2(i – $\Delta u + u_1$, j–$\Delta v + v_1$);
     glEvalCoord2(i – $\Delta u + u_1$, (j + 1) – $\Delta v + v_1$);
   }
   glEnd();

}
```

If *mode* is GL_LINE, then a call to glEvalMesh2 is equivalent to:

```
for (j = j1; j ≤ j2; j += 1) {
     glBegin(GL_LINE_STRIP);
      for (i = i1; i ≤ i2; i += 1)
           glEvalCoord2(i–$\Delta u + u_1$, j–$\Delta v + v_1$);
   glEnd();
}
for (i = i1;  i ≤i2; i += 1) {
    glBegin(GL_LINE_STRIP);
     for (j = j1; j ≤ j1; j += 1)
         glEvalCoord2(i – $\Delta u + u_1$, j–$\Delta v + v_1$);
     glEnd();
}
```

And finally, if *mode* is GL_POINT, then a call to glEvalMesh2 is equivalent to:

```
glBegin(GL_POINTS);
for (j = j1; j ≤ j2; j += 1) {
   for (i = i1; i ≤ i2; i += 1) {
     glEvalCoord2(i – $\Delta u + u_1$,  j – $\Delta v + v_1$);
   }
}
glEnd();
```

In all three cases, the only absolute numeric requirements are that if i=n, then the value computed from $i – \Delta u + u_1$ is exactly u_2, and if j=m, then the value computed from $j – \Delta v + v_1$ is exactly v_2.

Errors

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

- GL_INVALID_OPERATION is generated if glEvalMesh is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_MAP1_GRID_DOMAIN
glGet with argument GL_MAP2_GRID_DOMAIN
glGet with argument GL_MAP1_GRID_SEGMENTS
glGet with argument GL_MAP2_GRID_SEGMENTS

See Also

glBegin,
glEvalCoord,
glEvalPoint,
glMap1,
glMap2,
glMapGrid

glEvalPoint

`glEvalPoint1, glEvalPoint2`: generate and evaluate a single point in a mesh.

C Specification

```
void glEvalPoint1(
    GLint i)
void glEvalPoint2(
    GLint i,
    GLint j)
```

Parameters

i	Specifies the integer value for grid domain variable *i*.
j	Specifies the integer value for grid domain variable *j* (glEvalPoint2 only).

Description

glMapGrid and glEvalMesh 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

glEvalCoord1($i - \Delta u + u_1$);

where

$\Delta u = (u_2 - u_1) / n$

and n, u_1, and u_2 are the arguments to the most recentglMapGrid1 command. The one absolute numeric requirement is that if i=n, then the value computed from $i - \Delta u + u_1$ is exactly u_2.

In the two-dimensional case, glEvalPoint2, let

$\Delta u = (u_2 - u_1) / n$

$\Delta v = (v_2 - v_1) / m$

where n, u_1, u_2, m, v_1, and v_2 are the arguments to the most recent glMapGrid2 command. Then the glEvalPoint2 command is equivalent to calling

glEvalCoord2($i - \Delta u + u_1$, $j - \Delta v + v_1$);

The only absolute numeric requirements are that if i=n, then the value computed from $i - \Delta u + u_1$ is exactly u_2, and if j=m, then the value computed from $j - \Delta v + v_1$ is exactly v_2.

Associated Gets

glGet with argument GL_MAP1_GRID_DOMAIN
glGet with argument GL_MAP2_GRID_DOMAIN
glGet with argument GL_MAP1_GRID_SEGMENTS
glGet with argument GL_MAP2_GRID_SEGMENTS

See Also

glEvalCoord,
glEvalMesh,
glMap1,
glMap2,
glMapGrid

6 F

glFeedbackBuffer

`glFeedbackBuffer`: controls feedback mode.

C Specification

```
void glFeedbackBuffer(
    GLsizei size,
    GLenum type,
    GLfloat *buffer)
```

Parameters

size	Specifies the maximum number of values that can be written into *buffer.*
type	Specifies a symbolic constant that describes the information that will be returned for each vertex. GL_2D, GL_3D, GL_3D_COLOR, GL_3D_COLOR_TEXTURE, and GL_4D_COLOR_TEXTURE are accepted.
size	Returns the feedback data.

Description

The glFeedbackBuffer function 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.

glFeedbackBuffer 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. glFeedbackBuffer 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, is an error.

When glRenderMode is called while in feedback mode, it returns the number of entries placed in the feedback array, and resets the feedback array pointer to the base of the feedback buffer. The returned value never exceeds size. If the feedback data required more room than was available in buffer, glRenderMode returns a negative value. To take the GL out of feedback mode, call glRenderMode with a parameter value other than GL_FEEDBACK.

While in feedback mode, each primitive, bitmap, or pixel rectangle that would be rasterized generates a block of values that are copied into the feedback array. If doing so would cause the number of entries to exceed the maximum, the block is partially written so as to fill the array (if there is any room left at all), and an overflow flag is set. Each block begins with a code indicating the primitive type, followed by values that describe the primitive's vertices and associated data. Entries are also written for bitmaps and pixel rectangles. Feedback occurs after polygon culling and glPolygonMode interpretation of polygons has taken place, so polygons that are culled are not returned in the feedback buffer. It can also occur after polygons with more than three edges are broken up into triangles, if the GL implementation renders polygons by performing this decomposition.

The glPassThrough command 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 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

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. *k* is 1 in color index mode and 4 in RGBA mode.

Table 6-1

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.

Notes

glFeedbackBuffer, when used in a display list, is not compiled into the display list but is executed immediately.

Errors

- GL_INVALID_ENUM is generated if *type* is not an accepted value.
- GL_INVALID_VALUE is generated if *size* is negative.
- GL_INVALID_OPERATION is generated if glFeedbackBuffer is called while the render mode is GL_FEEDBACK, or if glRenderMode is called with argument GL_FEEDBACK before glFeedbackBuffer is called at least once.
- GL_INVALID_OPERATION is generated if glFeedbackBuffer is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_RENDER_MODE

See Also

glBegin,
glLineStipple,
glPassThrough,
glPolygonMode,
glRenderMode,
glSelectBuffer

glFinish

`glFinish`: block until all GL execution is complete.

C Specification

```
void glFinish(void)
```

Description

glFinish does not return until the effects of all previously called GL commands are complete. Such effects include all changes to GL state, all changes to connection state, and all changes to the frame buffer contents.

Notes

glFinish requires a round trip to the server.

Errors

- GL_INVALID_OPERATION is generated if glFinish is executed between the execution of glBegin and the corresponding execution of glEnd.

See Also

glFlush

glFlush

`glFlush`: force execution of GL commands in finite time.

C Specification

```
void glFlush(void)
```

Description

Different GL implementations buffer commands in several different locations, including network buffers and the graphics accelerator itself. glFlush empties all of these buffers, causing all issued commands to be executed as quickly as they are accepted by the actual rendering engine. Though this execution may not be completed in any particular time period, it does complete in finite time.

Because any GL program might be executed over a network, or on an accelerator that buffers commands, all programs should call glFlush whenever they count on having all of their previously issued commands completed. For example, call glFlush before waiting for user input that depends on the generated image.

Notes

glFlush can return at any time. It does not wait until the execution of all previously issued GL commands is complete.

Errors

- GL_INVALID_OPERATION is generated if glFlush is executed between the execution of glBegin and the corresponding execution of glEnd.

See Also

glFinish

glFog

`glFogf, glFogi, glFogfv, glFogiv`: specify fog parameters.

C Specification

```
void glFogf(
    GLenum pname,
    GLfloat param)
void glFogi(
    GLenum pname,
    GLint param)
void glFogfv(
    GLenum pname,
    const GLfloat *params)
void glFogiv(
    GLenum pname,
    const GLint *params)
```

Parameters

pname	Specifies a single-valued fog parameter. GL_FOG_MODE, GL_FOG_DENSITY, GL_FOG_START, GL_FOG_END, and GL_FOG_INDEX are accepted.
param	Specifies the value that *pname* will be set to.
pname	Specifies a fog parameter. GL_FOG_MODE, GL_FOG_DENSITY, GL_FOG_START, GL_FOG_END, GL_FOG_INDEX, and GL_FOG_COLOR are accepted.
params	Specifies the value or values to be assigned to pname. GL_FOG_COLOR requires an array of four values. All other parameters accept an array containing only a single value.

Description

Fog is initially disabled. While enabled, fog affects rasterized geometry, bitmaps, and pixel blocks, but not buffer clear operations. To enable and disable fog, call glEnable and glDisable with argument GL_FOG.

glFog assigns the value or values in *params* to the fog parameter specified by *pname*. The following values are accepted for *pname*:

GL_FOG_MODE

params 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 below. The initial fog mode is GL_EXP.

GL_FOG_DENSITY

params 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 initial fog density is 1.

GL_FOG_START

params is a single integer or floating-point value that specifies *start*, the near distance used in the linear fog equation. The initial near distance is 0.

GL_FOG_END

params is a single integer or floating-point value that specifies *end*, the far distance used in the linear fog equation. The initial far distance is 1.

GL_FOG_INDEX

params is a single integer or floating-point value that specifies i_f, the fog color index. The initial fog index is 0.

GL_FOG_COLOR

params 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 initial fog color is (0, 0, 0, 0).

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. Let *z* be the distance in eye coordinates from the origin to the fragment being fogged. The equation for GL_LINEAR fog is

$$f = (end - z) / (end - start)$$

The equation for GL_EXP fog is

$$f = e^{-(\text{density} \cdot z)}$$

The equation for GL_EXP2 fog is

$$f = e^{-(\text{density} \cdot z)2}$$

Regardless of the fog mode, *f* is clamped to the range [0, 1] after it is computed. Then, if the GL is in RGBA color mode, the fragment's color C_r is replaced by

$$C_{r'} = f\,C_r + (1- f)\,C_f$$

In color index mode, the fragment's color index i_r is replaced by

$$i_{r'} = i_r + (1 - f)\,i_f$$

Errors

- GL_INVALID_ENUM is generated if *pname* is not an accepted value, or if *pname* is GL_FOG_MODE and *params* is not an accepted value.
- GL_INVALID_VALUE is generated if *pname* is GL_FOG_DENSITY, and *params* is negative.
- GL_INVALID_OPERATION is generated if glFog is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

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

See Also

glEnable

glFrontFace

`glFrontFace`: define front- and back-facing polygons.

C Specification

```
void glFrontFace(
    GLenum mode)
```

Parameters

mode	Specifies the orientation of front-facing polygons. GL_CW and GL_CCW are accepted. The initial value is GL_CCW.

Description

In a scene composed entirely of opaque closed surfaces, back-facing polygons are never visible. Eliminating these invisible polygons has the obvious benefit of speeding up the rendering of the image. To enable and disable elimination of back-facing polygons, call glEnable and glDisable with 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. glFrontFace specifies whether polygons with clockwise winding in window coordinates, or counterclockwise winding in window coordinates, are taken to be front-facing. Passing GL_CCW to *mode* selects counterclockwise polygons as front-facing; GL_CW selects clockwise polygons as front-facing. By default, counterclockwise polygons are taken to be front-facing.

Errors

- GL_INVALID_ENUM is generated if *mode* is not an accepted value.
- GL_INVALID_OPERATION is generated if glFrontFace is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_FRONT_FACE

See Also

glCullFace,
glLightModel

glFrustum

`glFrustum`: multiply the current matrix by a perspective matrix.

C Specification

```
void glFrustum(
    GLdouble left,
    GLdouble right,
    GLdouble bottom,
    GLdouble top,
    GLdouble zNear,
    GLdouble zFar)
```

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.
zNear, zFar	Specify the distances to the near and far depth clipping planes. Both distances must be positive.

Description

glFrustum describes a perspective matrix that produces a perspective projection. The current matrix (see glMatrixMode) is multiplied by this matrix and the result replaces the current matrix, as if glMultMatrix were called with the following matrix as its argument:

$$\begin{matrix} E & 0 & A & 0 \\ 0 & F & B & 0 \\ 0 & 0 & C & D \\ 0 & 0 & -1 & 0 \end{matrix}$$

where:

A = (right + left) / (right - left)

B = (top + bottom) / (top - bottom)

C = (far + near) / (far- near)

D = (2×far×near) / (far- near)

E = (2×near) / (right - left)

F = (2×near) / (top - bottom)

Typically, the matrix mode is GL_PROJECTION, and (*left*, *bottom*, -*zNear*) and (*right*, *top*, -*zNear*) specify the points on the near clipping plane that are mapped to the lower left and upper right corners of the window, assuming that the eye is located at (0, 0, 0). -*zFar* specifies the location of the far clipping plane. Both *zNear* and *zFar* must be positive.

Use glPushMatrix and glPopMatrix to save and restore the current matrix stack.

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 $\log_2 r$ bits of depth-buffer precision are lost. Because *r* approaches infinity as *zNear* approaches 0, *zNear* must never be set to 0.

Errors

- GL_INVALID_VALUE is generated if *zNear* or *zFar* is not positive.
- GL_INVALID_OPERATION is generated if glFrustum is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_MATRIX_MODE
glGet with argument GL_MODELVIEW_MATRIX
glGet with argument GL_PROJECTION_MATRIX
glGet with argument GL_TEXTURE_MATRIX

See Also

glOrtho,
glMatrixMode,
glMultMatrix,
glPushMatrix
glViewport

7 G

glGenLists

`glGenLists`: generate a contiguous set of empty display lists.

C Specification

```
GLuint glGenLists(
    GLsizei range)
```

Parameters

range	Specifies the number of contiguous empty display lists to be generated.

Description

glGenLists 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, 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.

Errors

- GL_INVALID_VALUE is generated if *range* is negative.
- GL_INVALID_OPERATION is generated if glGenLists is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glIsList

See Also

glCallList,
glCallLists,
glDeleteLists,
glNewList

glGenTextures

`glGenTextures`: generate texture names.

C Specification

```
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

glGenTextures 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.

Notes

glGenTextures 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

See Also

glBindTexture,
glCopyTexImage1D,
glCopyTexImage2D,
glDeleteTextures,
glGet,
glGetTexParameter,

glTexImage1D,
glTexImage2D,
glTexParameter

glGet

`glGetBooleanv, glGetDoublev, glGetFloatv, glGetIntegerv`: return the value or values of a selected parameter.

C Specification

```
void glGetBooleanv(
   GLenum pname,
   GLboolean *params)
void glGetDoublev(
   GLenum pname,
   GLdouble *params)
void glGetFloatv(
   GLenum pname,
   GLfloat *params)
void glGetIntegerv(
   GLenum pname,
   GLint *params)
```

Parameters

pname	Specifies the parameter value to be returned. The symbolic constants in the list below are accepted.
params	Returns the value or values of the specified parameter.

Description

These four commands return values for simple state variables in GL. *pname* is a symbolic constant indicating the state variable to be returned, and *params* is a pointer to an array of the indicated type in which to place the returned data.

Type conversion is performed if *params* has a different type 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 0.0 (or 0). 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 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 *pname*:

GL_ACCUM_ALPHA_BITS

params returns one value, the number of alpha bitplanes in the accumulation buffer.

GL_ACCUM_BLUE_BITS

params returns one value, the number of blue bitplanes in the accumulation buffer.

GL_ACCUM_CLEAR_VALUE

params returns four values: the red, green, blue, and alpha values used to clear the accumulation buffer. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and - 1.0 returns the most negative representable integer value. The initial value is (0, 0, 0, 0). See glClearAccum.

GL_ACCUM_GREEN_BITS

params returns one value, the number of green bitplanes in the accumulation buffer.

GL_ACCUM_RED_BITS

params returns one value, the number of red bitplanes in the accumulation buffer.

GL_ALPHA_BIAS

params returns one value, the alpha bias factor used during pixel transfers. The initial value is 0.See glPixelTransfer.

GL_ALPHA_BITS

params returns one value, the number of alpha bitplanes in each color buffer.

GL_ALPHA_SCALE

params returns one value, the alpha scale factor used during pixel transfers. The initial value is 1. See glPixelTransfer.

GL_ALPHA_TEST

params returns a single boolean value indicating whether alpha testing of fragments is enabled. The initial value is GL_FALSE. See glAlphaFunc.

GL_ALPHA_TEST_FUNC

params returns one value, the symbolic name of the alpha test function. The initial value is GL_ALWAYS. See glAlphaFunc.

GL_ALPHA_TEST_REF

params returns one value, the reference value for the alpha test. The initial value is 0. See glAlphaFunc. An integer value, if requested, is linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and - 1.0 returns the most negative representable integer value.

GL_ATTRIB_STACK_DEPTH

params returns one value, the depth of the attribute stack. If the stack is empty, 0 is returned. The initial value is 0. See glPushAttrib.

GL_AUTO_NORMAL

params returns a single boolean value indicating whether 2D map evaluation automatically generates surface normals. The initial value is GL_FALSE. See glMap2.

GL_AUX_BUFFERS

params returns one value, the number of auxiliary color buffers. The initial value is 0.

GL_BLEND

params returns a single boolean value indicating whether blending is enabled. The initial value is GL_FALSE. See glBlendFunc.

GL_BLEND_DST

params returns one value, the symbolic constant identifying the destination blend function. The initial value is GL_ZERO. See glBlendFunc.

GL_BLEND_SRC

params returns one value, the symbolic constant identifying the source blend function. The initial value is GL_ONE. See glBlendFunc.

GL_BLUE_BIAS

params returns one value, the blue bias factor used during pixel transfers. The initial value is 0. See glPixelTransfer.

GL_BLUE_BITS

params returns one value, the number of blue bitplanes in each color buffer.

GL_BLUE_SCALE

params returns one value, the blue scale factor used during pixel transfers. The initial value is 1. See glPixelTransfer.

GL_BUFFER_SWAP_MODE_HINT_hp

params returns one value, a symbolic constant indicating the mode of the buffer-swap mode hint. The initial value is GL_FASTEST. See glHint.

GL_CLIENT_ATTRIB_STACK_DEPTH

params returns one value indicating the depth of the attribute stack. The initial value is 0. See glPushClientAttrib. GL_CLIP_PLANE*i* *params* returns a single boolean value indicating whether the specified clipping plane is enabled. The initial value is GL_FALSE. See glClipPlane.

GL_COLOR_ARRAY

params returns a single boolean value indicating whether the color array is enabled. The initial value is GL_FALSE. See glColorPointer.

GL_COLOR_ARRAY_SIZE

params returns one value, the number of components per color in the color array. The initial value is 4. See glColorPointer.

GL_COLOR_ARRAY_STRIDE

params returns one value, the byte offset between consecutive colors in the color array. The initial value is 0. See glColorPointer.

GL_COLOR_ARRAY_TYPE

params returns one value, the data type of each component in the color array. The initial value is GL_FLOAT. See glColorPointer.

GL_COLOR_CLEAR_VALUE

params returns four values: the red, green, blue, and alpha values used to clear the color buffers. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. The initial value is (0, 0, 0, 0). See glClearColor.

GL_COLOR_LOGIC_OP

params returns a single boolean value indicating whether a fragment's RGBA color values are merged into the frame buffer using a logical operation. The initial value is GL_FALSE. See glLogicOp.

GL_COLOR_MATERIAL

params returns a single boolean value indicating whether one or more material parameters are tracking the current color. The initial value is GL_FALSE. See glColorMaterial.

GL_COLOR_MATERIAL_FACE

params returns one value, a symbolic constant indicating which materials have a parameter that is tracking the current color. The initial value is GL_FRONT_AND_BACK. See glColorMaterial.

GL_COLOR_MATERIAL_PARAMETER

params returns one value, a symbolic constant indicating which material parameters are tracking the current color. The initial value is GL_AMBIENT_AND_DIFFUSE. See glColorMaterial.

GL_COLOR_WRITEMASK

params returns four boolean values: the red, green, blue, and alpha write enables for the color buffers. The initial value is (GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE). See glColorMask.

GL_CULL_FACE

params returns a single boolean value indicating whether polygon culling is enabled. The initial value is GL_FALSE. See glCullFace.

GL_CULL_FACE_MODE

params returns one value, a symbolic constant indicating which polygon faces are to be culled. The initial value is GL_BACK. See glCullFace.

GL_CURRENT_COLOR

params returns four values: the red, green, blue, and alpha values of the current color. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See glColor. The initial value is (1, 1, 1, 1).

GL_CURRENT_INDEX

params returns one value, the current color index. The initial value is 1. See glIndex.

GL_CURRENT_NORMAL

params 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. The initial value is (0, 0, 1). See glNormal.

GL_CURRENT_RASTER_COLOR

params returns four values: the red, green, blue, and alpha values of the current raster position. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and - 1.0 returns the most negative representable integer value. The initial value is (1, 1, 1, 1). See glRasterPos.

GL_CURRENT_RASTER_DISTANCE

params returns one value, the distance from the eye to the current raster position. The initial value is 0. See glRasterPos.

GL_CURRENT_RASTER_INDEX

params returns one value, the color index of the current raster position. The initial value is 1. See glRasterPos.

GL_CURRENT_RASTER_POSITION

params returns four values: the *x*, *y*, *z*, and *w* components of the current raster position. *x*, *y*, and *z* are in window coordinates, and *w* is in clip coordinates. The initial value is (0, 0, 0, 1). See glRasterPos.

GL_CURRENT_RASTER_POSITION_VALID

params returns a single boolean value indicating whether the current raster position is valid. The initial value is GL_TRUE. See glRasterPos.

GL_CURRENT_RASTER_TEXTURE_COORDS

params returns four values: the *s*, *t*, *r*, and *q* current raster texture coordinates. The initial value is (0, 0, 0, 1). See glRasterPos and glTexCoord.

GL_CURRENT_TEXTURE_COORDS

params returns four values: the *s*, *t*, *r*, and *q* current texture coordinates. The initial value is (0, 0, 0, 1). See glTexCoord.

GL_DEPTH_BIAS

params returns one value, the depth bias factor used during pixel transfers. The initial value is 0. See glPixelTransfer.

GL_DEPTH_BITS

params returns one value, the number of bitplanes in the depth buffer.

GL_DEPTH_CLEAR_VALUE

params 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. The initial value is 1. See glClearDepth.

GL_DEPTH_FUNC

params returns one value, the symbolic constant that indicates the depth comparison function. The initial value is GL_LESS. See glDepthFunc.

GL_DEPTH_RANGE

params 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. The initial value is (0, 1). See glDepthRange.

GL_DEPTH_SCALE

params returns one value, the depth scale factor used during pixel transfers. The initial value is 1. See glPixelTransfer.

GL_DEPTH_TEST

params returns a single boolean value indicating whether depth testing of fragments is enabled. The initial value is GL_FALSE. See glDepthFunc and glDepthRange.

GL_DEPTH_WRITEMASK

params returns a single boolean value indicating if the depth buffer is enabled for writing. The initial value is GL_TRUE. See glDepthMask.

GL_DITHER

params returns a single boolean value indicating whether dithering of fragment colors and indices is enabled. The initial value is GL_TRUE.

GL_DOUBLEBUFFER

params returns a single boolean value indicating whether double buffering is supported.

GL_DRAW_BUFFER

params returns one value, a symbolic constant indicating which buffers are being drawn to. See glDrawBuffer. The initial value is GL_BACK if there are back buffers, otherwise it is GL_FRONT.

GL_EDGE_FLAG

params returns a single boolean value indicating whether the current edge flag is GL_TRUE or GL_FALSE. The initial value is GL_TRUE. See glEdgeFlag.

GL_EDGE_FLAG_ARRAY

params 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_STRIDE

params returns one value, the byte offset between consecutive edge flags in the edge flag array. The initial value is 0. See glEdgeFlagPointer.

GL_EXT_DEPTH_TEXTURE

params returns a single boolean value indicating whether texture-depth is enabled. The initial value is GL_FALSE. See glTexImage3DEXT.

GL_EXT_SHADOW

params returns a single boolean value indicating whether shadowing is enabled. The initial value is GL_FALSE. See glTexImage3DEXT.

GL_EXT_TEXTURE_BORDER_CLAMP

params returns a single boolean value indicating whether border clamping is enabled. The initial value is GL_FALSE. See glTexImage3DEXT.

GL_EXT_TEXTURE3D

params returns a single boolean value indicating whether 3D texturing is enabled. The initial value is GL_FALSE. See glTexImage3DEXT.

GL_EXT_TEXTURE_EDGE_CLAMP

params returns a single boolean value indicating whether edge clamping is enabled. The initial value is GL_FALSE. See glTexImage3DEXT.

GL_FOG

params returns a single boolean value indicating whether fogging is enabled. The initial value is GL_FALSE. See glFog.

GL_FOG_COLOR

params returns four values: the red, green, blue and alpha components of the fog color. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. The initial value is (0, 0, 0, 0). See glFog.

GL_FOG_DENSITY

params returns one value, the fog density parameter. The initial value is 1. See glFog.

GL_FOG_END

params returns one value, the end factor for the linear fog equation. The initial value is 1. See glFog.

GL_FOG_HINT

params returns one value, a symbolic constant indicating the mode of the fog hint. The initial value is GL_DONT_CARE. See glHint.

GL_FOG_INDEX

params returns one value, the fog color index. The initial value is 0. See glFog.

GL_FOG_MODE

params returns one value, a symbolic constant indicating which fog equation is selected. The initial value is GL_EXP. See glFog.

GL_FOG_START

params returns one value, the start factor for the linear fog equation. The initial value is 0. See glFog.

GL_FRONT_FACE

params returns one value, a symbolic constant indicating whether clockwise or counterclockwise polygon winding is treated as front-facing. The initial value is GL_CCW. See glFrontFace.

GL_GREEN_BIAS

params returns one value, the green bias factor used during pixel transfers. The initial value is 0.

GL_GREEN_BITS

params returns one value, the number of green bitplanes in each color buffer.

GL_GREEN_SCALE

params returns one value, the green scale factor used during pixel transfers. The initial value is 1. See glPixelTransfer.

GL_hp_TEXTURING_LIGHTING

params returns a single boolean value indicating whether texture lighting is enabled. The initial value is GL_FALSE. See glTexImage3DEXT.

GL_INDEX_ARRAY

params returns a single boolean value indicating whether the color index array is enabled. The initial value is GL_FALSE. See glIndexPointer.

GL_INDEX_ARRAY_STRIDE

params 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_TYPE

params returns one value, the data type of indexes in the color index array. The initial value is GL_FLOAT. See glIndexPointer.

GL_INDEX_BITS

params returns one value, the number of bitplanes in each color index buffer.

GL_INDEX_CLEAR_VALUE

params returns one value, the color index used to clear the color index buffers. The initial value is 0. See glClearIndex.

GL_INDEX_LOGIC_OP

params returns a single boolean value indicating whether a fragment's index values are merged into the frame buffer using a logical operation. The initial value is GL_FALSE. See glLogicOp.

GL_INDEX_MODE

params returns a single boolean value indicating whether the GL is in color index mode (GL_TRUE) or RGBA mode (GL_FALSE).

GL_INDEX_OFFSET

params returns one value, the offset added to color and stencil indices during pixel transfers. The initial value is 0. See glPixelTransfer.

GL_INDEX_SHIFT

params returns one value, the amount that color and stencil indices are shifted during pixel transfers. The initial value is 0. See glPixelTransfer.

GL_INDEX_WRITEMASK

params returns one value, a mask indicating which bitplanes of each color index buffer can be written. The initial value is all 1s. See glIndexMask. GL_LIGHT*i params* returns a single boolean value indicating whether the specified light is enabled. The initial value is GL_FALSE. See glLight and glLightModel.

GL_LIGHTING

params returns a single boolean value indicating whether lighting is enabled. The initial value is GL_FALSE. See glLightModel.

GL_LIGHT_MODEL_AMBIENT

params returns four values: the red, green, blue, and alpha components of the ambient intensity of the entire scene. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and - 1.0 returns the most negative representable integer value. The initial value is (0.2, 0.2, 0.2, 1.0). See glLightModel.

GL_LIGHT_MODEL_LOCAL_VIEWER

params returns a single boolean value indicating whether specular reflection calculations treat the viewer as being local to the scene. The initial value is GL_FALSE. See glLightModel.

GL_LIGHT_MODEL_TWO_SIDE

params returns a single boolean value indicating whether separate materials are used to compute lighting for front- and back-facing polygons. The initial value is GL_FALSE. See glLightModel.

GL_LINE_SMOOTH

params returns a single boolean value indicating whether anti-aliasing of lines is enabled. The initial value is GL_FALSE. See glLineWidth.

GL_LINE_SMOOTH_HINT

params returns one value, a symbolic constant indicating the mode of the line anti-aliasing hint. The initial value is GL_DONT_CARE. See glHint.

GL_LINE_STIPPLE

params returns a single boolean value indicating whether stippling of lines is enabled. The initial value is GL_FALSE. See glLineStipple.

GL_LINE_STIPPLE_PATTERN

params returns one value, the 16-bit line stipple pattern. The initial value is all 1s. See glLineStipple.

GL_LINE_STIPPLE_REPEAT

params returns one value, the line stipple repeat factor. The initial value is 1. See glLineStipple.

GL_LINE_WIDTH

params returns one value, the line width as specified with glLineWidth. The initial value is 1.

GL_LINE_WIDTH_GRANULARITY

params returns one value, the width difference between adjacent supported widths for anti-aliased lines. See glLineWidth.

GL_LINE_WIDTH_RANGE

params returns two values: the smallest and largest supported widths for anti-aliased lines. See glLineWidth.

GL_LIST_BASE

params returns one value, the base offset added to all names in arrays presented to glCallLists. The initial value is 0. See glListBase.

GL_LIST_INDEX

params returns one value, the name of the display list currently under construction. 0 is returned if no display list is currently under construction. The initial value is 0. See glNewList.

GL_LIST_MODE

params returns one value, a symbolic constant indicating the construction mode of the display list currently under construction. The initial value is 0. See glNewList.

GL_LOGIC_OP_MODE

params returns one value, a symbolic constant indicating the selected logic operation mode. The initial value is GL_COPY. See glLogicOp.

GL_MAP1_COLOR_4

params returns a single boolean value indicating whether 1D evaluation generates colors. The initial value is GL_FALSE. See glMap1.

GL_MAP1_GRID_DOMAIN

params returns two values: the endpoints of the 1D map's grid domain. The initial value is (0, 1). See glMapGrid.

GL_MAP1_GRID_SEGMENTS

params returns one value, the number of partitions in the 1D map's grid domain. The initial value is 1. See glMapGrid.

GL_MAP1_INDEX

params returns a single boolean value indicating whether 1D evaluation generates color indices. The initial value is GL_FALSE. See glMap1.

GL_MAP1_NORMAL

params returns a single boolean value indicating whether 1D evaluation generates normals. The initial value is GL_FALSE. See glMap1.

GL_MAP1_TEXTURE_COORD_1

params returns a single boolean value indicating whether 1D evaluation generates 1D texture coordinates. The initial value is GL_FALSE. See glMap1.

GL_MAP1_TEXTURE_COORD_2

params returns a single boolean value indicating whether 1D evaluation generates 2D texture coordinates. The initial value is GL_FALSE. See glMap1.

GL_MAP1_TEXTURE_COORD_3

params returns a single boolean value indicating whether 1D evaluation generates 3D texture coordinates. The initial value is GL_FALSE. See glMap1.

GL_MAP1_TEXTURE_COORD_4

params returns a single boolean value indicating whether 1D evaluation generates 4D texture coordinates. The initial value is GL_FALSE. See glMap1.

GL_MAP1_VERTEX_3

params returns a single boolean value indicating whether 1D evaluation generates 3D vertex coordinates. The initial value is GL_FALSE. See glMap1.

GL_MAP1_VERTEX_4

params returns a single boolean value indicating whether 1D evaluation generates 4D vertex coordinates. The initial value is GL_FALSE. See glMap1. GL_MAP2_COLOR_4

params returns a single boolean value indicating whether 2D evaluation generates colors. The initial value is GL_FALSE. See glMap2.

GL_MAP2_GRID_DOMAIN

params returns four values: the endpoints of the 2D map's *i* and *j* grid domains. The initial value is (0, 1; 0, 1). See glMapGrid.

GL_MAP2_GRID_SEGMENTS

params returns two values: the number of partitions in the 2D map's *i* and *j* grid domains. The initial value is (1, 1). See glMapGrid.

GL_MAP2_INDEX

params returns a single boolean value indicating whether 2D evaluation generates color indices. The initial value is GL_FALSE. See glMap2.

GL_MAP2_NORMAL

params returns a single boolean value indicating whether 2D evaluation generates normals. The initial value is GL_FALSE. See glMap2.

GL_MAP2_TEXTURE_COORD_1

params returns a single boolean value indicating whether 2D evaluation generates 1D texture coordinates. The initial value is GL_FALSE. See glMap2.

GL_MAP2_TEXTURE_COORD_2

params returns a single boolean value indicating whether 2D evaluation generates 2D texture coordinates. The initial value is GL_FALSE. See glMap2.

GL_MAP2_TEXTURE_COORD_3

params returns a single boolean value indicating whether 2D evaluation generates 3D texture coordinates. The initial value is GL_FALSE. See glMap2.

GL_MAP2_TEXTURE_COORD_4

params returns a single boolean value indicating whether 2D evaluation generates 4D texture coordinates. The initial value is GL_FALSE. See glMap2.

GL_MAP2_VERTEX_3

params returns a single boolean value indicating whether 2D evaluation generates 3D vertex coordinates. The initial value is GL_FALSE. See glMap2.

GL_MAP2_VERTEX_4

params returns a single boolean value indicating whether 2D evaluation generates 4D vertex coordinates. The initial value is GL_FALSE. See glMap2.

GL_MAP_COLOR

params returns a single boolean value indicating if colors and color indices are to be replaced by table lookup during pixel transfers. The initial value is GL_FALSE. See glPixelTransfer.

GL_MAP_STENCIL

params returns a single boolean value indicating if stencil indices are to be replaced by table lookup during pixel transfers. The initial value is GL_FALSE. See glPixelTransfer.

GL_MATRIX_MODE

params returns one value, a symbolic constant indicating which matrix stack is currently the target of all matrix operations. The initial value is GL_MODELVIEW. See glMatrixMode.

GL_MAX_CLIENT_ATTRIB_STACK_DEPTH

params returns one value indicating the maximum supported depth of the client attribute stack. See glPushClientAttrib.

GL_MAX_ATTRIB_STACK_DEPTH

params returns one value, the maximum supported depth of the attribute stack. The value must be at least 16. See glPushAttrib.

GL_MAX_CLIP_PLANES

params returns one value, the maximum number of application-defined clipping planes. The value must be at least 6. See glClipPlane.

GL_MAX_EVAL_ORDER

params returns one value, the maximum equation order supported by 1D and 2D evaluators. The value must be at least 8. See glMap1 and glMap2.

GL_MAX_LIGHTS

params returns one value, the maximum number of lights. The value must be at least 8. See glLight.

GL_MAX_LIST_NESTING

params returns one value, the maximum recursion depth allowed during display-list traversal. The value must be at least 64. See glCallList.

GL_MAX_MODELVIEW_STACK_DEPTH

params returns one value, the maximum supported depth of the model view matrix stack. The value must be at least 32. See glPushMatrix.

GL_MAX_NAME_STACK_DEPTH

params returns one value, the maximum supported depth of the selection name stack. he value must be at least 64. See glPushName.

GL_MAX_PIXEL_MAP_TABLE

params returns one value, the maximum supported size of a glPixelMap lookup table. The value must be at least 32. See glPixelMap.

GL_MAX_PROJECTION_STACK_DEPTH

params returns one value, the maximum supported depth of the projection matrix stack. The value must be at least 2. See glPushMatrix.

GL_MAX_TEXTURE_SIZE

params returns one value. The value gives a rough estimate of the largest texture that the GL can handle. If the GL version is 1.1 or greater, use GL_PROXY_TEXTURE_1D or GL_PROXY_TEXTURE_2D to determine if a texture is too large. See glTexImage1D and glTexImage2D.

GL_MAX_TEXTURE_STACK_DEPTH

params returns one value, the maximum supported depth of the texture matrix stack. The value must be at least 2. See glPushMatrix.

GL_MAX_VIEWPORT_DIMS

params returns two values: the maximum supported width and height of the viewport. These must be at least as large as the visible dimensions of the display being rendered to. See glViewport.

GL_MODELVIEW_MATRIX

params returns sixteen values: the modelview matrix on the top of the modelview matrix stack. Initially this matrix is the identity matrix. See glPushMatrix.

GL_MODELVIEW_STACK_DEPTH

params returns one value, the number of matrices on the modelview matrix stack. The initial value is 1. See glPushMatrix.

GL_NAME_STACK_DEPTH

params returns one value, the number of names on the selection name stack. The initial value is 0. See glPushName.

GL_NORMAL_ARRAY

params returns a single boolean value, indicating whether the normal array is enabled. The initial value is GL_FALSE. See glNormalPointer.

GL_NORMAL_ARRAY_STRIDE

params returns one value, the byte offset between consecutive normals in the normal array. The initial value is 0. See glNormalPointer.

GL_NORMAL_ARRAY_TYPE

params returns one value, the data type of each coordinate in the normal array. The initial value is GL_FLOAT. See glNormalPointer.

GL_NORMALIZE

params returns a single boolean value indicating whether normals are automatically scaled to unit length after they have been transformed to eye coordinates. The initial value is GL_FALSE. See glNormal.

GL_OCCLUSION_TEST_hp

params returns a single boolean value indicating whether HP's occlusion-testing extension is currently activated. See glEnable.

GL_OCCLUSION_TEST_RESULT_hp

params returns a single boolean value indicating whether the previously-rendered geometry was entirely occluded. A side-effect of getting this value is that the flag is cleared (in preparation for the next occlusion test). See glEnable.

GL_PACK_ALIGNMENT

params returns one value, the byte alignment used for writing pixel data to memory. The initial value is 4. See glPixelStore.

GL_PACK_LSB_FIRST

params 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. The initial value is GL_FALSE. See glPixelStore.

GL_PACK_ROW_LENGTH

params returns one value, the row length used for writing pixel data to memory. The initial value is 0. See glPixelStore.

GL_PACK_SKIP_PIXELS

params returns one value, the number of pixel locations skipped before the first pixel is written into memory. The initial value is 0. See glPixelStore.

GL_PACK_SKIP_ROWS

params returns one value, the number of rows of pixel locations skipped before the first pixel is written into memory. The initial value is 0. See glPixelStore.

GL_PACK_SWAP_BYTES

params returns a single boolean value indicating whether the bytes of two-byte and four-byte pixel indices and components are swapped before being written to memory. The initial value is GL_FALSE. See glPixelStore.

GL_PERSPECTIVE_CORRECTION_HINT

params returns one value, a symbolic constant indicating the mode of the perspective correction hint. The initial value is GL_DONT_CARE. See glHint.

GL_PIXEL_MAP_A_TO_A_SIZE

params returns one value, the size of the alpha-to-alpha pixel translation table. The initial value is 1. See glPixelMap.

GL_PIXEL_MAP_B_TO_B_SIZE

params returns one value, the size of the blue-to-blue pixel translation table. The initial value is 1. See glPixelMap.

GL_PIXEL_MAP_G_TO_G_SIZE

params returns one value, the size of the green-to-green pixel translation table. The initial value is 1. See glPixelMap.

GL_PIXEL_MAP_I_TO_A_SIZE

params returns one value, the size of the index-to-alpha pixel translation table. The initial value is 1. See glPixelMap.

GL_PIXEL_MAP_I_TO_B_SIZE

params returns one value, the size of the index-to-blue pixel translation table. The initial value is 1. See glPixelMap.

GL_PIXEL_MAP_I_TO_G_SIZE

params returns one value, the size of the index-to-green pixel translation table. The initial value is 1. See glPixelMap.

GL_PIXEL_MAP_I_TO_I_SIZE

params returns one value, the size of the index-to-index pixel translation table. The initial value is 1. See glPixelMap.

GL_PIXEL_MAP_I_TO_R_SIZE

params returns one value, the size of the index-to-red pixel translation table. The initial value is 1. See glPixelMap.

GL_PIXEL_MAP_R_TO_R_SIZE

params returns one value, the size of the red-to-red pixel translation table. The initial value is 1. See glPixelMap.

GL_PIXEL_MAP_S_TO_S_SIZE

params returns one value, the size of the stencil-to-stencil pixel translation table. The initial value is 1. See glPixelMap.

GL_POINT_SIZE

params returns one value, the point size as specified by glPointSize. The initial value is 1.

GL_POINT_SIZE_GRANULARITY

params returns one value, the size difference between adjacent supported sizes for anti-aliased points. See glPointSize.

GL_POINT_SIZE_RANGE

params returns two values: the smallest and largest supported sizes for anti-aliased points. The smallest size must be at most 1, and the largest size must be at least 1. See glPointSize.

GL_POINT_SMOOTH

params returns a single boolean value indicating whether anti-aliasing of points is enabled. The initial value is GL_FALSE. See glPointSize.

GL_POINT_SMOOTH_HINT

params returns one value, a symbolic constant indicating the mode of the point anti-aliasing hint. The initial value is GL_DONT_CARE. See glHint.

GL_POLYGON_MODE

params returns two values: symbolic constants indicating whether front-facing and back-facing polygons are rasterized as points, lines, or filled polygons. The initial value is GL_FILL. See glPolygonMode.

GL_POLYGON_OFFSET_FACTOR

params returns one value, the scaling factor used to determine the variable offset that is added to the depth value of each fragment generated when a polygon is rasterized. The initial value is 0. See glPolygonOffset.

GL_POLYGON_OFFSET_UNITS

params returns one value. This value is multiplied by an implementation-specific value and then added to the depth value of each fragment generated when a polygon is rasterized. The initial value is 0. See glPolygonOffset.

GL_POLYGON_OFFSET_FILL

params 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

params 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

params 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_SMOOTH

params returns a single boolean value indicating whether anti-aliasing of polygons is enabled. The initial value is GL_FALSE. See glPolygonMode.

GL_POLYGON_SMOOTH_HINT

params returns one value, a symbolic constant indicating the mode of the polygon anti-aliasing hint. The initial value is GL_DONT_CARE. See glHint.

GL_POLYGON_STIPPLE

params returns a single boolean value indicating whether polygon stippling is enabled. The initial value is GL_FALSE. See glPolygonStipple.

GL_PROJECTION_MATRIX

params returns sixteen values: the projection matrix on the top of the projection matrix stack. Initially this matrix is the identity matrix. See glPushMatrix.

GL_PROJECTION_STACK_DEPTH

params returns one value, the number of matrices on the projection matrix stack. The initial value is 1. See glPushMatrix.

GL_READ_BUFFER

params returns one value, a symbolic constant indicating which color buffer is selected for reading. The initial value is GL_BACK if there is a back buffer, otherwise it is GL_FRONT. See glReadPixels and glAccum.

GL_RED_BIAS

params returns one value, the red bias factor used during pixel transfers. The initial value is 0.

GL_RED_BITS

params returns one value, the number of red bitplanes in each color buffer.

GL_RED_SCALE

params returns one value, the red scale factor used during pixel transfers. The initial value is 1. See glPixelTransfer.

GL_RENDER_MODE

params returns one value, a symbolic constant indicating whether the GL is in render, select, or feedback mode. The initial value is GL_RENDER. See glRenderMode.

GL_RGBA_MODE

params returns a single boolean value indicating whether the GL is in RGBA mode (true) or color index mode (false). See glColor.

GL_SCISSOR_BOX

params returns four values: the x and y window coordinates of the scissor box, followed by its width and height. Initially the x and y window coordinates are both 0 and the width and height are set to the size of the window. See glScissor.

GL_SCISSOR_TEST

params returns a single boolean value indicating whether scissoring is enabled. The initial value is GL_FALSE. See glScissor.

GL_SHADE_MODEL

params returns one value, a symbolic constant indicating whether the shading mode is flat or smooth. The initial value is GL_SMOOTH. See glShadeModel.

GL_STENCIL_BITS

params returns one value, the number of bitplanes in the stencil buffer.

GL_STENCIL_CLEAR_VALUE

params returns one value, the index to which the stencil bitplanes are cleared. The initial value is 0. See glClearStencil.

GL_STENCIL_FAIL

params returns one value, a symbolic constant indicating what action is taken when the stencil test fails. The initial value is GL_KEEP. See glStencilOp.

GL_STENCIL_FUNC

params returns one value, a symbolic constant indicating what function is used to compare the stencil reference value with the stencil buffer value. The initial value is GL_ALWAYS. See glStencilFunc.

GL_STENCIL_PASS_DEPTH_FAIL

params returns one value, a symbolic constant indicating what action is taken when the stencil test passes, but the depth test fails. The initial value is GL_KEEP. See glStencilOp.

GL_STENCIL_PASS_DEPTH_PASS

params returns one value, a symbolic constant indicating what action is taken when the stencil test passes and the depth test passes. The initial value is GL_KEEP. See glStencilOp.

GL_STENCIL_REF

params returns one value, the reference value that is compared with the contents of the stencil buffer. The initial value is 0. See glStencilFunc.

GL_STENCIL_TEST

params returns a single boolean value indicating whether stencil testing of fragments is enabled. The initial value is GL_FALSE. See glStencilFunc and glStencilOp.

GL_STENCIL_VALUE_MASK

params returns one value, the mask that is used to mask both the stencil reference value and the stencil buffer value before they are compared. The initial value is all 1s. See glStencilFunc.

GL_STENCIL_WRITEMASK

params returns one value, the mask that controls writing of the stencil bitplanes. The initial value is all 1s. See glStencilMask.

GL_STEREO

params returns a single boolean value indicating whether stereo buffers (left and right) are supported.

GL_SUBPIXEL_BITS

params returns one value, an estimate of the number of bits of subpixel resolution that are used to position rasterized geometry in window coordinates. The initial value is 4.

GL_TEXTURE_1D

params returns a single boolean value indicating whether 1D texture mapping is enabled. The initial value is GL_FALSE. See glTexImage1D.

GL_TEXTURE_1D_BINDING

params 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_2D

params returns a single boolean value indicating whether 2D texture mapping is enabled. The initial value is GL_FALSE. See glTexImage2D.

GL_TEXTURE_2D_BINDING

params returns a single value, the name of the texture currently bound to the targetGL_TEXTURE_2D. The initial value is 0. See glBindTexture.

GL_TEXTURE_COORD_ARRAY

params 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_SIZE

params 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_STRIDE

params 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_TYPE

params returns one value, the data type of the coordinates in the texture coordinate array. The initial value is GL_FLOAT. See glTexCoordPointer.

GL_TEXTURE_GEN_Q

params returns a single boolean value indicating whether automatic generation of the q texture coordinate is enabled. The initial value is GL_FALSE. See glTexGen.

GL_TEXTURE_GEN_R

params returns a single boolean value indicating whether automatic generation of the r texture coordinate is enabled. The initial value is GL_FALSE. See glTexGen.

GL_TEXTURE_GEN_S

params returns a single boolean value indicating whether automatic generation of the S texture coordinate is enabled. The initial value is GL_FALSE. See glTexGen.

GL_TEXTURE_GEN_T

params returns a single boolean value indicating whether automatic generation of the T texture coordinate is enabled. The initial value is GL_FALSE. See glTexGen.

GL_TEXTURE_MATRIX

params returns sixteen values: the texture matrix on the top of the texture matrix stack. Initially this matrix is the identity matrix. See glPushMatrix.

GL_TEXTURE_STACK_DEPTH

params returns one value, the number of matrices on the texture matrix stack. The initial value is 1. See glPushMatrix.

GL_UNPACK_ALIGNMENT

params returns one value, the byte alignment used for reading pixel data from memory. The initial value is 4. See glPixelStore.

GL_UNPACK_LSB_FIRST

params 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. The initial value is GL_FALSE. See glPixelStore.

GL_UNPACK_ROW_LENGTH

params returns one value, the row length used for reading pixel data from memory. The initial value is 0. See glPixelStore.

GL_UNPACK_SKIP_PIXELS

params returns one value, the number of pixel locations skipped before the first pixel is read from memory. The initial value is 0. See glPixelStore.

GL_UNPACK_SKIP_ROWS

params returns one value, the number of rows of pixel locations skipped before the first pixel is read from memory. The initial value is 0. See glPixelStore.

GL_UNPACK_SWAP_BYTES

params returns a single boolean value indicating whether the bytes of two-byte and four-byte pixel indices and components are swapped after being read from memory. The initial value is GL_FALSE. See glPixelStore.

GL_VERTEX_ARRAY

params returns a single boolean value indicating whether the vertex array is enabled. The initial value is GL_FALSE. See glVertexPointer.

GL_VERTEX_ARRAY_SIZE

params returns one value, the number of coordinates per vertex in the vertex array. The initial value is 4. See glVertexPointer.

GL_VERTEX_ARRAY_STRIDE

params returns one value, the byte offset between consecutive vertexes in the vertex array. The initial value is 0. See glVertexPointer.

GL_VERTEX_ARRAY_TYPE

params returns one value, the data type of each coordinate in the vertex array. The initial value is GL_FLOAT. See glVertexPointer.

GL_VIEWPORT

params returns four values: the *x* and *y* window coordinates of the viewport, followed by its width and height. Initially the *x* and *y* window coordinates are both set to 0, and the width and height are set to the width and height of the window into which the GL will do its rendering. See glViewport.

GL_ZOOM_X

params returns one value, the x pixel zoom factor. The initial value is 1. See glPixelZoom.

GL_ZOOM_Y

params returns one value, the y pixel zoom factor. The initial value is 1. See glPixelZoom.

Many of the boolean parameters can also be queried more easily using glIsEnabled.

Notes

GL_COLOR_LOGIC_OP, GL_COLOR_ARRAY, GL_COLOR_ARRAY_SIZE, GL_COLOR_ARRAY_STRIDE, GL_COLOR_ARRAY_TYPE, GL_EDGE_FLAG_ARRAY, GL_EDGE_FLAG_ARRAY_STRIDE, GL_INDEX_ARRAY, GL_INDEX_ARRAY_STRIDE, GL_INDEX_ARRAY_TYPE, GL_INDEX_LOGIC_OP, GL_NORMAL_ARRAY, GL_NORMAL_ARRAY_STRIDE, GL_NORMAL_ARRAY_TYPE, GL_POLYGON_OFFSET_UNITS, GL_POLYGON_OFFSET_FACTOR, GL_POLYGON_OFFSET_FILL, GL_POLYGON_OFFSET_LINE, GL_POLYGON_OFFSET_POINT, GL_TEXTURE_COORD_ARRAY, GL_TEXTURE_COORD_ARRAY_SIZE, GL_TEXTURE_COORD_ARRAY_STRIDE, GL_TEXTURE_COORD_ARRAY_TYPE, GL_VERTEX_ARRAY, GL_VERTEX_ARRAY_SIZE, GL_VERTEX_ARRAY_STRIDE, and GL_VERTEX_ARRAY_TYPE are available only if the GL version is 1.1 or greater.

Errors

- GL_INVALID_ENUM is generated if *pname* is not an accepted value.
- GL_INVALID_OPERATION is generated if glGet is executed between the execution of glBegin and the corresponding execution of glEnd.

See Also

glGetClipPlane,
glGetError,
glGetLight,
glGetMap,
glGetMaterial,
glGetPixelMap,
glGetPointer,
glGetPolygonStipple,
glGetString,
glGetTexEnv,

glGetTexGen,
glGetTexImage,
glGetTexLevelParameter,
glGetTexParameter,
glIsEnabled

glXGetClientString

`glXGetClientString`: return a string describing the client.

C Specification

```
constchar *glXGetClientString(
   Display *dpy,
   int name)
```

Parameters

dpy	Specifies the connection to the X server.
name	Specifies which string is returned. One of GLX_VENDOR, GLX_VERSION, or GLX_EXTENSIONS.

Description

glXGetClientString 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.

Notes

glXGetClientString 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.

glXGetClientString only returns information about GLX extensions supported by the client. Call glGetString to get a list of GL extensions supported by the server.

See Also

glXQueryVersion,
glXQueryExtensionsString,
glXQueryServerString

glXGetClipPlane

`glGetClipPlane`: return the coefficients of the specified clipping plane.

C Specification

```
void glGetClipPlane(
    GLenum plane,
    GLdouble *equation)
```

Parameters

plane	Specifies a clipping plane. The number of clipping planes depends on the implementation, but at least six clipping planes are supported. They are identified by symbolic names of the form GL_CLIP_PLANE*i* where $0 \geq i <$ GL_MAX_CLIP_PLANES.
equation	Returns four double-precision values that are the coefficients of the plane equation of *plane* in eye coordinates. The initial value is (0, 0, 0, 0).

Description

glGetClipPlane returns in *equation* the four coefficients of the plane equation for *plane*.

Notes

It is always the case that GL_CLIP_PLANE*i* = GL_CLIP_PLANE0 + *i*.

If an error is generated, no change is made to the contents of *equation*.

Errors

- GL_INVALID_ENUM is generated if *plane* is not an accepted value.
- GL_INVALID_OPERATION is generated if glGetClipPlane is executed between the execution of glBegin and the corresponding execution of glEnd.

See Also

glClipPlane

glXGetConfig

`glXGetConfig`: return information about GLX visuals.

C Specification

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

Parameters

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

Description

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

attrib is one of the following:

GLX_USE_GL

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

GLX_BUFFER_SIZE

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

GLX_LEVEL

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

GLX_RGBA

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

GLX_DOUBLEBUFFER

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

GLX_STEREO

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

GLX_AUX_BUFFERS

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

GLX_RED_SIZE

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

GLX_GREEN_SIZE

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

GLX_BLUE_SIZE

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

GLX_ALPHA_SIZE

Number of bits of alpha stored in each color buffer. Undefined if GLX_RGBA is False.

GLX_DEPTH_SIZE

Number of bits in the depth buffer.

GLX_STENCIL_SIZE

Number of bits in the stencil buffer.

GLX_ACCUM_RED_SIZE

Number of bits of red stored in the accumulation buffer.

GLX_ACCUM_GREEN_SIZE

Number of bits of green stored in the accumulation buffer.

GLX_ACCUM_BLUE_SIZE

Number of bits of blue stored in the accumulation buffer.

GLX_ACCUM_ALPHA_SIZE

Number of bits of alpha stored in the accumulation buffer.

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

Although a GLX implementation can export many visuals that support GL rendering, it must support at least one RGBA visual. This visual must have at least one color buffer, a stencil buffer of at least 1 bit, a depth buffer of at least 12 bits, and an accumulation buffer. Alpha bitplanes are optional in this visual. However, its color buffer size must be as great as that of the deepest TrueColor, DirectColor, PseudoColor, or StaticColor visual supported on level zero, and it must itself be made available on level zero.

In addition, if the X server exports a PseudoColor or StaticColor visual on frame buffer level 0, a color index visual is also required on that level. It must have at least one color buffer, a stencil buffer of at least 1 bit, and 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.

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

Notes

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

Errors

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

See Also

glXChooseVisual,
glXCreateContext

glXGetCurrentContext

`glXGetCurrentContext`: return the current context.

C Specification

```
GLXContext glXGetCurrentContext(void)
```

Description

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

See Also

glXCreateContext,
glXMakeCurrent

glXGetCurrentDisplay

`glXGetCurrentDisplay`: get display for current context.

C Specification

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

Description

glXGetCurrentDisplay returns the display for the current context. If no context is current, NULL is returned.

glXGetCurrentDisplay returns client-side information. It does not make a round trip to the server, and therefore does not flush any pending events.

Notes

glXGetCurrentDisplay is only supported if the GLX version is 1.2 or greater.

See Also

glXQueryVersion,
glXQueryExtensionsString

glXGetCurrentDrawable

`glXGetCurrentDrawable`: return the current drawable.

C Specification

```
GLXDrawable glXGetCurrentDrawable(void)
```

Description

glXGetCurrentDrawable returns the current drawable, as specified by glXMakeCurrent. If there is no current drawable, None is returned.

glXGetCurrentDrawable returns client-side information. It does not make a round trip to the server.

See Also

glXCreateGLXPixmap,
glXMakeCurrent

glGetError

`glGetError`: return error information.

C Specification

```
GLenum glGetError(void)
```

Description

glGetError 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. Thus, 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 following errors are currently defined:

GL_NO_ERROR

No error has been recorded. The value of this symbolic constant is guaranteed to be 0.

GL_INVALID_ENUM

An unacceptable value is specified for an enumerated argument. The offending command is ignored, and has no other side effect than to set the error flag.

GL_INVALID_VALUE

A numeric argument is out of range. The offending command is ignored, and has no other side effect than to set the error flag.

GL_INVALID_OPERATION

The specified operation is not allowed in the current state. The offending command is ignored, and has no other side effect than to set the error flag.

GL_STACK_OVERFLOW

This command would cause a stack overflow. The offending command is ignored, and has no other side effect than to set the error flag.

GL_STACK_UNDERFLOW

This command would cause a stack underflow. The offending command is ignored, and has no other side effect 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.

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. If the generating command returns a value, it returns 0. If glGetError itself generates an error, it returns 0.

Errors

- GL_INVALID_OPERATION is generated if glGetError is executed between the execution of glBegin and the corresponding execution of glEnd. In this case glGetError returns 0.

glGetLight

`glGetLightfv, glGetLightiv`: return light source parameter values.

C Specification

```
void glGetLightfv(
    GLenum light,
    GLenum pname,
    GLfloat *params)
void glGetLightiv(
    GLenum light,
    GLenum pname,
    GLint *params)
```

Parameters

light	Specifies a light source. The number of possible lights depends on the implementation, but at least eight lights are supported. They are identified by symbolic names of the form GL_LIGHT*i* where $0 \geq i <$ GL_MAX_LIGHTS.
pname	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.
params	Returns the requested data.

Description

glGetLight returns in *params* 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 $0 \geq i <$ GL_MAX_LIGHTS, where GL_MAX_LIGHTS is an implementation-dependent constant that is greater than or equal to eight. *pname* specifies one of ten light source parameters, again by symbolic name.

The following parameters are defined:

GL_AMBIENT

params 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. The initial value is (0, 0, 0, 1).

GL_DIFFUSE

params 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. The initial value for GL_LIGHT0 is (1, 1, 1, 1); for other lights, the initial value is (0, 0, 0, 0).

GL_SPECULAR

params 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. The initial value for GL_LIGHT0 is (1, 1, 1, 1); for other lights, the initial value is (0, 0, 0, 0).

GL_POSITION

params 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 identity at the time glLight was called. The initial value is (0, 0, 1, 0).

GL_SPOT_DIRECTION

params 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. The initial value is (0, 0, - 1).

GL_SPOT_EXPONENT

params 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. The initial value is 0.

GL_SPOT_CUTOFF

params 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. The initial value is 180.

GL_CONSTANT_ATTENUATION

params 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. The initial value is 1.

GL_LINEAR_ATTENUATION

params 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. The initial value is 0.

GL_QUADRATIC_ATTENUATION

params 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. The initial value is 0.

Notes

It is always the case that GL_LIGHT*i* = GL_LIGHT0 + *i*.

If an error is generated, no change is made to the contents of *params*.

Errors

- GL_INVALID_ENUM is generated if *light* or *pname* is not an accepted value.
- GL_INVALID_OPERATION is generated if glGetLight is executed between the execution of glBegin and the corresponding execution of glEnd.

See Also

glLight

glGetMap

glGetMapdv, glGetMapfv, glGetMapiv: return evaluator parameters.

C Specification

```
void glGetMapdv(
    GLenum target,
    GLenum query,
    GLdouble *v)
void glGet
    GLenum target,
    GLenum query,
    GLfloat *v)
void glGetMapiv(
    GLenum target,
    GLenum query,
    GLint *v)
```

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.

Description

glMap1 and glMap2 define evaluators. glGetMap returns evaluator parameters. *target* chooses a map, *query* selects a specific parameter, and *v* points to storage where the values will be returned.

The acceptable values for the *target* parameter are described in the glMap1 and glMap2 reference pages.

query can assume the following values:

GL_COEFF

v returns the control points for the evaluator function. One-dimensional evaluators return *order* control points, and two-dimensional evaluators return *uorder* times *vorder* control points. Each control point consists of one, two, three, or four integer, single-precision floating-point, or double-precision floating-point values, depending on the type of the evaluator. The GL returns two-dimensional control points 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*. The initial value is 1. Two-dimensional evaluators return two values, *uorder* and *vorder*. The initial value is 1,1.

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.

Notes

If an error is generated, no change is made to the contents of *v*.

Errors

- GL_INVALID_ENUM is generated if either *target* or *query* is not an accepted value.
- GL_INVALID_OPERATION is generated if glGetMap is executed between the execution of glBegin and the corresponding execution of glEnd.

See Also

glEvalCoord,
glMap1,
glMap2

glGetMaterial

`glGetMaterialfv, glGetMaterialiv`: return material parameters.

C Specification

```
void glGetMaterialfv(
    GLenum face,
    GLenum pname,
    GLfloat *params)
void glGetMaterialiv(
    GLenum face,
    GLenum pname,
    GLint *params)
```

Parameters

face	Specifies which of the two materials is being queried. GL_FRONT or GL_BACK are accepted, representing the front and back materials, respectively.
pname	Specifies the material parameter to return. GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_EMISSION, GL_SHININESS, and GL_COLOR_INDEXES are accepted.
params	Returns the requested data.

Description

glGetMaterial returns in *params* the value or values of parameter *pname* of material *face*. Six parameters are defined:

GL_AMBIENT

params 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. The initial value is (0.2, 0.2, 0.2, 1.0).

GL_DIFFUSE

params 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. The initial value is (0.8, 0.8, 0.8, 1.0).

GL_SPECULAR

params 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. The initial value is (0, 0, 0, 1).

GL_EMISSION

params 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, 1.0], the corresponding integer return value is undefined. The initial value is (0, 0, 0, 1).

GL_SHININESS

params 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. The initial value is 0.

GL_COLOR_INDEXES

params 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. (All the other parameters are used only for RGBA lighting.) Integer values, when requested, are computed by rounding the internal floating-point values to the nearest integer values.

Notes

If an error is generated, no change is made to the contents of params.

Errors

- GL_INVALID_ENUM is generated if *face* or *pname* is not an accepted value.
- GL_INVALID_OPERATION is generated if glGetMaterial is executed between the execution of glBegin and the corresponding execution of glEnd.

See Also

glMaterial

gluGetNurbsProperty

`gluGetNurbsProperty`: get a NURBS property.

C Specification

```
void gluGetNurbsProperty(
    GLUnurbs* nurb,
    GLenum property,
    GLfloat* data)
```

Parameters

nurb	Specifies the NURBS object (created with gluNewNurbsRenderer).
property	Specifies the property whose value is to be fetched. Valid values are GLU_CULLING, GLU_SAMPLING_TOLERANCE, GLU_DISPLAY_MODE, GLU_AUTO_LOAD_MATRIX, GLU_PARAMETRIC_TOLERANCE, GLU_SAMPLING_METHOD, GLU_U_STEP, and GLU_V_STEP.
data	Specifies a pointer to the location into which the value of the named property is written.

Description

gluGetNurbsProperty retrieves properties stored in a NURBS object. These properties affect the way that NURBS curves and surfaces are rendered. See the gluNurbsProperty reference page for information about what the properties are and what they do.

See Also

gluNewNurbsRenderer,
gluNurbsProperty

glGetPixelMap

`glGetPixelMapfv, glGetPixelMapuiv, glGetPixelMapusv`: return the specified pixel map.

C Specification

```
void glGetPixelMapfv
    GLenum map
    GLfloat *values
void glGetPixelMapuiv
    GLenum map
    GLuint *values
void glGetPixelMapusv
    GLenum map
    GLushort *values)
```

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.

Description

See the glPixelMap reference page for a description of the acceptable values for the *map* parameter. glGetPixelMap returns in *values* the contents of the pixel map specified in *map*. 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. Return unsigned integer values are undefined if the map value was not in the range [0, 1].

To determine the required size of *map*, call glGet with the appropriate symbolic constant.

Notes

If an error is generated, no change is made to the contents of *values*.

Errors

- GL_INVALID_ENUM is generated if *map* is not an accepted value.

- GL_INVALID_OPERATION is generated if glGetPixelMap is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

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

See Also

glCopyPixels,
glDrawPixels,
glPixelMap,
glPixelTransfer,
glReadPixels,
glTexImage1D,
glTexImage2D

glGetPointer

`glGetPointer:` return the address of the specified pointer.

C Specification

```
void glGetPointerv(
    GLenum pname,
    GLvoid* *params)
```

Parameters

pname	Specifies the array or buffer pointer to be returned. Symbolic constants GL_COLOR_ARRAY_POINTER, GL_EDGE_FLAG_ARRAY_POINTER, GL_FEEDBACK_BUFFER_POINTER, GL_INDEX_ARRAY_POINTER, GL_NORMAL_ARRAY_POINTER, GL_TEXTURE_COORD_ARRAY_POINTER, GL_SELECTION_BUFFER_POINTER, and GL_VERTEX_ARRAY_POINTER are accepted.
params	Returns the pointer value specified by *pname*.

Description

glGetPointerv returns pointer information. *pname* 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.

Notes

glGetPointerv is available only if the GL version is 1.1 or greater.

The pointers are all client-side state.

The initial value for each pointer is 0.

Errors

- GL_INVALID_ENUM is generated if *pname* is not an accepted value.

See Also

glArrayElement,
glColorPointer,
glDrawArrays,
glEdgeFlagPointer,
glFeedbackBuffer,
glIndexPointer,
glInterleavedArrays,
glNormalPointer,

glSelectBuffer,
glTexCoordPointer,
glVertexPointer

glGetPolygonStipple

`glGetPolygonStipple`: return the polygon stipple pattern.

C Specification

```
void glGetPolygonStipple
    GLubyte *mask)
```

Parameters

mask	Returns the stipple pattern. The initial value is all 1’s.

Description

glGetPolygonStipple returns to *mask* a 32×32 polygon stipple pattern. The pattern is packed into memory as if glReadPixels with both *height* and *width* of 32, *type* of GL_BITMAP, and *format* of GL_COLOR_INDEX were called, and the stipple pattern were stored in an internal 32×32 color index buffer. Unlike glReadPixels, however, pixel transfer operations (shift, offset, pixel map) are not applied to the returned stipple image.

Notes

If an error is generated, no change is made to the contents of *mask*.

Errors

- GL_INVALID_OPERATION is generated if glGetPolygonStipple is executed between the execution of glBegin and the corresponding execution of glEnd.

See Also

glPixelStore,
glPixelTransfer,
glPolygonStipple,
glReadPixels

glGetString

`glGetString`: return a string describing the current GL connection.

C Specification

```
constGLubyte *glGetString(
    GLenum name)
```

Parameters

name	Specifies a symbolic constant, one of GL_VENDOR, GL_RENDERER, GL_VERSION, or GL_EXTENSIONS.

Description

glGetString returns a pointer to a static string describing some aspect of the current GL connection. *name* can be one of the following:

GL_VENDOR

Returns 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 the GL does not include queries for the performance characteristics of an implementation, some applications are written to recognize known platforms and modify their GL usage based on known performance characteristics of these platforms. Strings GL_VENDOR and GL_RENDERER together uniquely specify a platform. They do not change from release to release and should be used by platform-recognition algorithms.

Some applications want to make use of features that are not part of the standard GL. These features may be implemented as extensions to the standard GL. The GL_EXTENSIONS string is a space-separated list of supported GL extensions. (Extension names never contain a space character.)

The GL_VERSION string begins with a version number. The version number uses one of these forms:

<major_number>.<minor_number>
<major_number>.<minor_number>.<release_number>

Vendor-specific information may follow the version number. Its format depends on the implementation, but a space always separates the version number and the vendor-specific information.

All strings are null-terminated.

Notes

If an error is generated, glGetString returns 0. The client and server may support different versions or extensions. glGetString always returns a compatible version number or list of extensions. The release number always describes the server.

Errors

- GL_INVALID_ENUM is generated if *name* is not an accepted value.
- GL_INVALID_OPERATION is generated if glGetString is executed between the execution of glBegin and the corresponding execution of glEnd.

gluGetString

`gluGetString`: return a string describing the GLU version or GLU extensions.

C Specification

constGLubyte *gluGetString(GLenum name)

Parameters

name	Specifies a symbolic constant, one of GLU_VERSION, or GLU_EXTENSIONS.

Description

gluGetString 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.

Notes

gluGetString only returns information about GLU extensions. Call glGetString to get a list of GL extensions.

gluGetString is an initialization routine. Calling it after a glNewList results in undefined behavior.

Errors

- NULL is returned if *name* is not GLU_VERSION or GLU_EXTENSIONS.

See Also

glGetString

gluGetTessProperty

`gluGetTessProperty`: get a tessellation object property.

C Specification

```
void gluGetTessProperty(
    GLUtesselator* tess,
    GLenum which,
    GLdouble* data)
```

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, and GLU_TESS_TOLERANCE.
data	Specifies a pointer to the location into which the value of the named property is written.

Description

gluGetTessProperty 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.

See Also

gluNewTess,
gluTessProperty

glGetTexEnv

`glGetTexEnvfv, glGetTexEnviv`: return texture environment parameters.

C Specification

```
void glGetTexEnvfv(
    GLenum target,
    GLenum pname,
    GLfloat *params)
void glGetTexEnviv(
    GLenum target,
    GLenum pname,
    GLint *params)
```

Parameters

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 and GL_TEXTURE_LIGHTING_MODE_hp.
params	Returns the requested data.

Description

glGetTexEnv returns in *params* selected values of a texture environment that was specified with glTexEnv. *target* specifies a texture environment. Currently, only one texture environment is defined and supported: GL_TEXTURE_ENV.

pname names a specific texture environment parameter, as follows:

GL_TEXTURE_ENV_MODE

params returns the single-valued texture environment mode, a symbolic constant. The initial value is GL_MODULATE.

GL_TEXTURE_ENV_COLOR

params 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. The initial value is (0, 0, 0, 0).

GL_TEXTURE_LIGHTING_MODE_hp

params returns the single-valued texture lighting mode, a symbolic constant.

Notes

If an error is generated, no change is made to the contents of *params*.

GL_TEXTURE_LIGHTING_MODE_hp is only supported if the GL_hp_texture_lighting extension is supported.

Errors

- GL_INVALID_ENUM is generated if *target* or *pname* is not an accepted value.
- GL_INVALID_OPERATION is generated if glGetTexEnv is executed between the execution of glBegin and the corresponding execution of glEnd.

See Also

glTexEnv

glGetTexGen

glGetTexGendv, glGetTexGenfv, glGetTexGeniv: return texture coordinate generation parameters.

C Specification

```
void glGetTexGendv(
    GLenum coord,
    GLenum pname,
    GLdouble *params)
void glGetTexGenfv(
    GLenum coord,
    GLenum pname,
    GLfloat *params)
void glGetTexGeniv(
    GLenum coord,
    GLenum pname,
    GLint *params)
```

Parameters

coord	Specifies a texture coordinate. Must be GL_S, GL_T, GL_R, or GL_Q.
pname	Specifies the symbolic name of the value(s) to be returned. Must be either GL_TEXTURE_GEN_MODE or the name of one of the texture generation plane equations: GL_OBJECT_PLANE or GL_EYE_PLANE.
params	Returns the requested data.

Description

glGetTexGen returns in *params* selected parameters of a texture coordinate generation function that was specified using glTexGen. *coord* names one of the (*s*, *t*, *r*, *q*) texture coordinates, using the symbolic constant GL_S, GL_T, GL_R, or GL_Q.

pname specifies one of three symbolic names:

GL_TEXTURE_GEN_MODE

params returns the single-valued texture generation function, a symbolic constant. The initial value is GL_EYE_LINEAR.

GL_OBJECT_PLANE

params 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

params 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 identity when glTexGen was called.

Notes

If an error is generated, no change is made to the contents of *params*.

Errors

- GL_INVALID_ENUM is generated if *coord* or *pname* is not an accepted value.
- GL_INVALID_OPERATION is generated if glGetTexGen is executed between the execution of glBegin and the corresponding execution of glEnd.

See Also

glTexGen

glGetTexImage

`glGetTexImage`: return a texture image.

C Specification

```
void glGetTexImage(
    GLenum target,
    GLint level,
    GLenum format,
    GLenum type,
    GLvoid *pixels)
```

Parameters

target	Specifies which texture is to be obtained. GL_TEXTURE_1D and GL_TEXTURE_2D are accepted.
level	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_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 *type*.

Description

glGetTexImage returns a texture image into *pixels*. *target* specifies whether the desired texture image is one specified by glTexImage1D (GL_TEXTURE_1D) or by glTexImage2D (GL_TEXTURE_2D). *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 reference pages glTexImage1D and glDrawPixels for a description of the acceptable values for the *format* and *type* parameters, respectively.

To understand the operation of glGetTexImage, consider the selected internal four-component texture image to be an RGBA color buffer the size of the image. The semantics of glGetTexImage are then identical to those of glReadPixels called with the same *format* and *type*, with *x* and *y* set to 0, *width* set to the width of the texture image (including border if one was specified), and *height* set to 1 for 1D images, or to the height of the texture image (including border if one was specified) for 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, green set to 0, blue set to 0, and alpha set to 1. Two-component textures are treated as RGBA buffers with red set to the value of component zero, alpha set to the value of component one, and green and blue set to 0. Finally, three-component textures are treated as RGBA buffers with red set to component zero, green set to component one, blue set to component two, and alpha set to 1.

To determine the required size of *pixels*, use glGetTexLevelParameter to determine 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 take the pixel storage parameters into account, especially GL_PACK_ALIGNMENT.

Notes

If an error is generated, no change is made to the contents of *pixels*.

Errors

- GL_INVALID_ENUM is generated if *target*, *format*, or *type* is not an accepted value.
- GL_INVALID_VALUE is generated if *level* is less than 0.
- 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 glGetTexImage is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

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

See Also

glDrawPixels,
glReadPixels,
glTexEnv,
glTexGen,
glTexImage1D,
glTexImage2D,
glTexSubImage1D,
glTexSubImage2D,
glTexParameter

glGetTexLevelParameter

`glGetTexLevelParameterfv, glGetTexLevelParameteriv`: return texture parameter values for a specific level of detail.

C Specification

```
void glGetTexLevelParameterfv(
    GLenum target,
    GLint level,
    GLenum pname
    GLfloat *params)
void glGetTexLevelParameteriv(
    GLenum target,
    GLint level,
    GLenum pname,
    GLint *params)
```

Parameters

target	Specifies the symbolic name of the target texture, either GL_TEXTURE_1D, GL_TEXTURE_2D, GL_PROXY_TEXTURE_1D, or GL_PROXY_TEXTURE_2D.
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.
pname	Specifies the symbolic name of a texture parameter. 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.

Description

glGetTexLevelParameter returns in *params* texture parameter values for a specific level-of-detail value, specified as *level. target* defines the target texture, either GL_TEXTURE_1D, GL_TEXTURE_2D, GL_PROXY_TEXTURE_1D, or GL_PROXY_TEXTURE_2D.

GL_MAX_TEXTURE_SIZE is not really descriptive enough. It has to report the largest square texture image that can be accommodated 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 accommodate a texture of a given configuration. If the texture cannot be accommodated, the texture state variables, which may be queried with glGetTexLevelParameter, are set to 0. If the texture can be accommodated, the texture state values will be set as they would be set for a non-proxy target.

pname specifies the texture parameter whose value or values will be returned.

The accepted parameter names are as follows:

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.

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_INTERNAL_FORMAT

params returns a single value, the internal format of the texture image.

GL_TEXTURE_BORDER

params returns a single value, the width in pixels of the border of the texture image. The initial value is 0.

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

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.

Notes

If an error is generated, no change is made to the contents of *params*.

GL_TEXTURE_INTERNAL_FORMAT is only available if the GL version is 1.1 or greater. In version 1.0, use GL_TEXTURE_COMPONENTS instead.

GL_PROXY_TEXTURE_1D and GL_PROXY_TEXTURE_2D are only available if the GL version is 1.1 or greater.

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 0.
- 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.

See Also

glGetTexParameter,
glCopyTexImage1D,
glCopyTexImage2D,
glCopyTexSubImage1D,
glCopyTexSubImage2D,

glTexEnv,
glTexGen,
glTexImage1D,
glTexImage2D,
glTexSubImage1D,
glTexSubImage2D,
glTexParameter

glGetTexParameter

`glGetTexParameterfv, glGetTexParameteriv`: return texture parameter values.

C Specification

```
void glGetTexParameterfv(
    GLenum target,
    GLenum pname,
    GLfloat *params)
void glGetTexParameteriv(
    GLenum target,
    GLenum pname,
    GLint *params)
```

Parameters

target	Specifies the symbolic name of the target texture. GL_TEXTURE_1D, GL_TEXTURE_2D, and GL_TEXTURE_3D_EXT are accepted.
pname	Specifies the symbolic name of a texture parameter. GL_TEXTURE_MAG_FILTER, GL_TEXTURE_MIN_FILTER, GL_TEXTURE_WRAP_R_EXT, GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T, GL_TEXTURE_BORDER_COLOR, GL_TEXTURE_PRIORITY, GL_GENERATE_MIPMAP_EXT, GL_TEXTURE_COMPARE_EXT, GL_TEXTURE_COMPARE_OPERATOR_EXT, and GL_TEXTURE_RESIDENT are accepted.
params	Returns the texture parameters.

Description

glGetTexParameter returns in *params* the value or values of the texture parameter specified as *pname*. *target* defines the target texture, either GL_TEXTURE_1D or GL_TEXTURE_2D, to specify one- or two-dimensional texturing. *pname* accepts the same symbols as glTexParameter, with the same interpretations:

GL_TEXTURE_MAG_FILTER

Returns the single-valued texture magnification filter, a symbolic constant. The initial value is GL_LINEAR.

GL_TEXTURE_MIN_FILTER

Returns the single-valued texture minification filter, a symbolic constant. The initial value is GL_NEAREST_MIPMAP_LINEAR.

GL_TEXTURE_WRAP_R_EXT

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. The initial value is GL_REPEAT.

GL_TEXTURE_WRAP_T

Returns the single-valued wrapping function for texture coordinate *t*, a symbolic constant. The initial value is GL_REPEAT.

GL_TEXTURE_BORDER_COLOR

Returns four integer or floating-point numbers that comprise the RGBA color of the texture border. Floating-point values are returned in the range [0, 1]. Integer values are returned as a linear mapping of the internal floating-point representation such that 1.0 maps to the most positive representable integer and - 1.0 maps to the most negative representable integer. The initial value is (0, 0, 0, 0).

GL_TEXTURE_PRIORITY

Returns the residence priority of the target texture (or the named texture bound to it). The initial value is 1. See glPrioritizeTextures.

GL_GENERATE_MIPMAP_EXT

Returns the single-valued flag which determines whether automatic mip level generation is in effect, either GL_TRUE or GL_FALSE.

GL_TEXTURE_COMPARE_EXT

Returns the single-valued flag which determines whether depth texture comparison is enabled, either GL_TRUE or GL_FALSE.

GL_TEXTURE_COMPARE_OPERATOR_EXT

Returns the single-valued depth texture comparison operator, a symbolic constant.

GL_TEXTURE_RESIDENT

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.

Notes

GL_TEXTURE_PRIORITY and GL_TEXTURE_RESIDENT are only available if the GL version is 1.1 or greater.

GL_TEXTURE_WRAP_R_EXT and the *target* GL_TEXTURE_3D_EXT are only supported if the extension GL_EXT_texture3D is supported.

GL_GENERATE_MIPMAP_EXT is only supported if the extension GL_EXT_generate_mipmap is supported.

GL_TEXTURE_COMPARE_EXT and GL_TEXTURE_COMPARE_OPERATOR_EXT are only supported if the extension GL_EXT_shadow is supported.

If an error is generated, no change is made to the contents of *params*.

Errors

- GL_INVALID_ENUM is generated if *target* or *pname* is not an accepted value.
- GL_INVALID_OPERATION is generated ifglGetTexParameter is executed between the execution of glBegin and the corresponding execution of glEnd.

See Also

glAreTexturesResident,
glPrioritizeTextures,
glTexParameter

8 H

glHint

`glHint`: specify implementation-specific hints.

C Specification

```
void glHint(
    GLenum target,
    GLenum mode)
```

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, and GL_POLYGON_SMOOTH_HINT are accepted.
mode	Specifies a symbolic constant indicating the desired behavior. GL_FASTEST, GL_NICEST, and GL_DONT_CARE are accepted.

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. The initial value for each *target* is GL_DONT_CARE. *mode* can be one of the following:

GL_FASTEST

The most efficient option should be chosen.

GL_NICEST

The most correct, or highest quality, option should be chosen.

GL_DONT_CARE

No preference.

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 as follows:

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 anti-aliased lines. If a larger filter function is applied, hinting GL_NICEST can result in more pixel fragments being generated during rasterization, 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/or texture coordinates.

GL_POINT_SMOOTH_HINT

Indicates the sampling quality of anti-aliased points. If a larger filter function is applied, hinting GL_NICEST can result in more pixel fragments being generated during rasterization,

GL_POLYGON_SMOOTH_HINT

Indicates the sampling quality of anti-aliased polygons. Hinting GL_NICEST can result in more pixel fragments being generated during rasterization, if a larger filter function is applied.

GL_BUFFER_SWAP_MODE_HINT_hp

GL_FASTEST switches to the faster double-buffering method, and GL_NICEST switches to the slower double-buffering method.

Notes

The interpretation of hints depends on the implementation. Some implementations ignore glHint settings.

Errors

- GL_INVALID_ENUM is generated if either *target* or *mode* is not an accepted value.
- GL_INVALID_OPERATION is generated if glHint is executed between the execution of glBegin and the corresponding execution of glEnd.

9 I

glIndex

`glIndexd, glIndexf, glIndexi, glIndexs, glIndexub, glIndexdv, glIndexfv, glIndexiv, glIndexsv, glIndexubv`: set the current color index.

C Specification

```
void glIndexd(
    GLdouble c)
void glIndexf(
    GLfloat c)
void glIndexi(
    GLint c)
void glIndexs(
    GLshort c)
void glIndexub(
    GLubyte c)
void glIndexdv(
    const GLdouble *c)
void glIndexfv(
    const GLfloat *c)
void glIndexiv(
    const GLint *c)
void glIndexsv(
    const GLshort *c)
void glIndexubv(
    const GLubyte *c)
```

Parameters

c	Specifies the new value for the current color index.
c	Specifies a pointer to a one-element array that contains the new value for the current color index.

Description

glIndex 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. The initial value is 1.

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.

Notes

glIndexub and glIndexubv are available only if the GL version is 1.1 or greater.

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

glGet with argument GL_CURRENT_INDEX

See Also

glColor,
glIndexPointer

glIndexMask

`glIndexMask`: control the writing of individual bits in the color index buffers.

C Specification

```
void glIndexMask(
    GLuint mask)
```

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.

Description

glIndexMask controls the writing of individual bits in the color index buffers. The least significant *n* bits of *mask*, where *n* is the number of bits in a color index buffer, specify a mask. Where a 1 (one) appears in the mask, it's possible to write to the corresponding bit in the color index buffer (or buffers). Where a 0 (zero) appears, the corresponding 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.

Errors

- GL_INVALID_OPERATION is generated if glIndexMask is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_INDEX_WRITEMASK

See Also

glColorMask,
glDepthMask,
glDrawBuffer,
glIndex,
glIndexPointer,
glStencilMask

glIndexPointer

`glIndexPointer`: define an array of color indexes.

C Specification

```
void glIndexPointer(
    GLenum type,
    GLsizei stride,
    const GLvoid *pointer)
```

Parameters

type	Specifies the data type of each color index in the
array	Symbolic constants GL_UNSIGNED_BYTE, GL_SHORT, GL_INT, GL_FLOAT, and GL_DOUBLE are accepted.
stride	Specifies the byte offset between consecutive color indexes. If *stride* is 0 (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.

Description

glIndexPointer 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; see glInterleavedArrays.)

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 to construct a sequence of primitives (all of the same type) from pre-specified vertex and vertex attribute arrays. Use glArrayElement to specify primitives by indexing vertexes and vertex attributes and glDrawElements to construct a sequence of primitives by indexing vertexes and vertex attributes.

Notes

glIndexPointer is available only if the GL version is 1.1 or greater.

The color index array is initially disabled, and it isn't 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.

glIndexPointer 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.

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

See Also

glArrayElement,
glColorPointer,
glDrawArrays,
glDrawElements,
glEdgeFlagPointer,
glEnable,
glGetPointer,
glInterleavedArrays,
glNormalPointer,
glPopClientAttrib,
glPushClientAttrib,
glTexCoordPointer,
glVertexPointer

glInitNames

`glInitNames`: initialize the name stack.

C Specification

```
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. glInitNames 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 glInitNames while the render mode is not GL_SELECT are ignored.

Errors

- GL_INVALID_OPERATION is generated if glInitNames is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_NAME_STACK_DEPTH
glGet with argument GL_MAX_NAME_STACK_DEPTH

See Also

glLoadName,
glPushName,
glRenderMode,
glSelectBuffer

glInterleavedArrays

`glInterleavedArrays`: simultaneously specify and enable several interleaved arrays.

C Specification

```
void glInterleavedArrays(
    GLenum format,
    GLsizei stride,
    const GLvoid *pointer)
```

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, and GL_T4F_C4F_N3F_V4F are accepted.
stride	Specifies the offset in bytes between each aggregate array element.

Description

glInterleavedArrays 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 separately.

If *stride* is 0, the aggregate elements are stored consecutively. Otherwise, *stride* bytes occur between the beginning of one aggregate array element and the beginning of the next aggregate array element.

format 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 four unsigned bytes if 4UB follows the C. If a color is extracted as four unsigned bytes, the vertex array element which follows is located at the first possible floating-point aligned address.

Notes

glInterleavedArrays 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 the execution of glBegin and the corresponding execution of glEnd, but an error may or may not be generated. If no error is generated, the operation is undefined.

glInterleavedArrays is typically implemented on the client side.

Vertex array parameters are client-side state and are therefore 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.

See Also

glArrayElement,
glColorPointer,
glDrawArrays,
glDrawElements,
glEdgeFlagPointer,
glEnableClientState,
glGetPointer,
glIndexPointer,
glNormalPointer,
glTexCoordPointer,
glVertexPointer

glXIntro

`glXIntro`: Introduction to OpenGL in the X Window system.

Overview

OpenGL (called GL in other pages) is a high-performance 3D-oriented renderer. It is available in the X window system through the GLX extension. To determine whether the GLX extension is supported by an X server, and if so, what version is supported, call glXQueryExtension and glXQueryVersion.

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 and with the renderer of any other X extension that is compatible with all core X visuals.

GLX extends drawables with several buffers other than the standard color buffer. These buffers include back and auxiliary color buffers, a depth buffer, a stencil buffer, and a color accumulation buffer. Some or all are included in each X visual that supports OpenGL.

To render using OpenGL into an X drawable, you must first choose a visual that defines the required OpenGL buffers. glXChooseVisual can be used to simplify selecting a compatible visual. If more control of the selection process is required, use XGetVisualInfo and glXGetConfig to select among all the available visuals.

Use the selected visual to create both a GLX context and an X drawable. GLX contexts are created with glXCreateContext, and drawables are created with either XCreateWindow or glXCreateGLXPixmap. Finally, bind the context and the drawable together using glXMakeCurrent. This context/drawable pair becomes the current context and current drawable, and it is used by all OpenGL commands until glXMakeCurrent is called with different arguments.

Both core X and OpenGL commands can be used to operate on the current drawable. The X and OpenGL command streams are not synchronized, however, except at explicitly created boundaries generated by calling glXWaitGL, glXWaitX, XSync, and glFlush.

Examples

Below is the minimum code required to create an RGBA-format, X window that's compatible with OpenGL and to clear it to yellow. The code is correct, but it does not include any error checking. Return values *dpy*, *vi*, *cx*, *cmap*, and *win* should all be tested.

```
#include <GL/glx.h>
#include <GL/gl.h>
#include <unistd.h>

static int attributeListSgl[] = {
        GLX_RGBA,
        GLX_RED_SIZE,   1, /*get the deepest buffer with 1 red bit*/
        GLX_GREEN_SIZE, 1,
        GLX_BLUE_SIZE, 1,
           None};

static int attributeListDbl[] = {
        GLX_RGBA,
        GLX_DOUBLE_BUFFER, /*In case single buffering is not supported*/
        GLX_RED_SIZE,   1,
        GLX_GREEN_SIZE, 1,
        GLX_BLUE_SIZE, 1,
          None};

static Bool WaitForNotify(Display *d, XEvent *e, char *arg) {
    return (e->type == MapNotify) && (e->xmap.window == (Window)arg);

}

int main(int argc, char **argv) {
   Display *dpy;
   XVisualInfo *vi;
   Colormap cmap;
   XSetWindowAttributes swa;
    Window win;
    GLXContext cx;
    XEvent event;
      int swap_flag = FALSE;

   /* get a connection */
    dpy = XOpenDisplay(0);

   /* get an appropriate visual */
   vi = glXChooseVisual(dpy, DefaultScreen(dpy), attributeListSgl);
    if (vi == NULL) {
    vi = glXChooseVisual(dpy, DefaultScreen(dpy), attributeListDbl);
      swap_flag = TRUE;
}

   /* create a GLX context */
     cx = glXCreateContext(dpy, vi, 0, GL_TRUE);

   /* create a color map */
    cmap = XCreateColormap(dpy, RootWindow(dpy, vi->screen),
                              vi->visual, AllocNone);

    /* create a window */
    swa.colormap = cmap;
    swa.border_pixel = 0;
    swa.event_mask = StructureNotifyMask;
      win = XCreateWindow(dpy, RootWindow(dpy, vi->screen), 0, 0, 100, 100,
                       0, vi->depth, InputOutput, vi->visual,
                              CWBorderPixel|CWColormap|CWEventMask, &swa);
     XMapWindow(dpy, win);
       XIfEvent(dpy, &event, WaitForNotify, (char*)win);

    /* connect the context to the window */
     glXMakeCurrent(dpy, win, cx);

     /* clear the buffer */
     glClearColor(1,1,0,1);
     glClear(GL_COLOR_BUFFER_BIT);
     glFlush();
       if (swap_flag) glXSwapBuffers(dpy,win);
    /* wait a while */
     sleep(10);
}
```

Notes

A color map must be created and passed to XCreateWindow. See the preceding example code.

A GLX context must be created and attached to an X drawable before OpenGL commands can be executed.

OpenGL commands issued while no context/drawable pair is current result in undefined behavior.

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

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

Using GLX Extensions

All supported GLX extensions will have a corresponding definition in glx.h and a token in the extension string returned by glXQueryExtensionsString. For example, if the EXT_visual_info extension is supported, then this token will be defined in glx.h and EXT_visual_info will appear in the extension string returned by glXQueryExtensionsString. The definitions in glx.h can be used at compile time to determine if procedure calls corresponding to an extension exist in the library.

GLX 1.1 and GLX 1.2

GLX 1.2 is now supported. It is backward compatible with GLX 1.1 and GLX 1.0.

GLX 1.2 corresponds to OpenGL version 1.1 and introduces the following new call: glXGetCurrentDisplay.

GLX 1.1 corresponds to OpenGL version 1.0 and introduces the following new calls: glXQueryExtensionsString, glXQueryServerString, and glXGetClientString.

Call glXQueryVersion to determine at runtime what version of GLX is available. glXQueryVersion returns the version that is supported on the connection. Thus if 1.2 is returned, both the client and server support GLX 1.2. You can also check the GLX version at compile time: GLX_VERSION_1_1 will be defined in glx.h if GLX 1.1 calls are supported and GLX_VERSION_1_2 will be defined if GLX 1.2 calls are supported.

See Also

glFinish,
glFlush,
glXChooseVisual,
glXCopyContext,
glXCreateContext,

glXCreateGLXPixmap,
glXDestroyContext,
glXGetClientString,
glXGetConfig,
glXIsDirect,
glXMakeCurrent,
glXQueryExtension,
glXQueryExtensionsString,
glXQueryServerString,
glXQueryVersion,
glXSwapBuffers,
glXUseXFont,
glXWaitGL,
glXWaitX,
XCreateColormap,
XCreateWindow,
XSync

glXIsDirect

`glXIsDirect`: indicate whether direct rendering is enabled.

C Specification

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

Parameters

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

Description

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

Errors

- GLXBadContext is generated if *ctx* is not a valid GLX context.

See Also

glXCreateContext

glIsEnabled

`glIsEnabled`: test whether a capability is enabled.

C Specification

```
GLboolean glIsEnabled(
    GLenum cap)
```

Parameters

cap	Specifies a symbolic constant indicating a GL capability.

Description

glIsEnabled returns GL_TRUE if *cap* is an enabled capability and returns GL_FALSE otherwise. Initially all capabilities except GL_DITHER are disabled; GL_DITHER is initially enabled.

The following capabilities are accepted for cap:

Constant	**See**
GL_ALPHA_TEST	glAlphaFunc
GL_AUTO_NORMAL	glEvalCoord
GL_BLEND	glBlendFunc, glLogicOp
GL_CLIP_PLANEi	glClipPlane
GL_COLOR_ARRAY	glColorPointer
GL_COLOR_LOGIC_OP	glLogicOp
GL_COLOR_MATERIAL	glColorMaterial
GL_CULL_FACE	glCullFace
GL_DEPTH_TEST	glDepthFunc, glDepthRange
GL_DITHER	glEnable
GL_EDGE_FLAG_ARRAY	glEdgeFlagPointer
GL_FOG	glFog
GL_INDEX_ARRAY	glIndexPointer
GL_INDEX_LOGIC_OP	glLogicOp
GL_LIGHTi	glLightModel, glLight
GL_LIGHTING	glMaterial, glLightModel, glLight
GL_LINE_SMOOTH	glLineWidth
GL_LINE_STIPPLE	glLineStipple

Constant	See
GL_MAP1_COLOR_4	glMap1, glMap2
GL_MAP2_TEXTURE_COORD_2	glMap2
GL_MAP2_TEXTURE_COORD_3	glMap2
GL_MAP2_TEXTURE_COORD_4	glMap2
GL_MAP2_VERTEX_3	glMap2
GL_MAP2_VERTEX_4	glMap2
GL_NORMAL_ARRAY	glNormalPointer
GL_NORMALIZE	glNormal
GL_OCCLUSION_TEST_hp	glEnable
GL_POINT_SMOOTH	glPointSize
GL_POLYGON_SMOOTH	glPolygonMode
GL_POLYGON_OFFSET_FILL	glPolygonOffset
GL_POLYGON_OFFSET_LINE	glPolygonOffset
GL_POLYGON_OFFSET_POINT	glPolygonOffset
GL_POLYGON_STIPPLE	glPolygonStipple
GL_RESCALE_NORMAL_EXT	glEnable
GL_SCISSOR_TEST	glScissor
GL_STENCIL_TEST	glStencilFunc, glStencilOp
GL_TEXTURE_1D	glTexImage1D
GL_TEXTURE_2D	glTexImage2D
GL_TEXTURE_3D_EXT	(if 3D texturing is supported) glTexImage3DEXT
GL_TEXTURE_COORD_ARRAY	glTexCoordPointer
GL_TEXTURE_GEN_Q	glTexGen
GL_TEXTURE_GEN_R	glTexGen
GL_TEXTURE_GEN_S	glTexGen
GL_TEXTURE_GEN_T	glTexGen
GL_VERTEX_ARRAY	glVertexPointer

Notes

If an error is generated, glIsEnabled returns 0.

GL_COLOR_LOGIC_OP, GL_COLOR_ARRAY, GL_EDGE_FLAG_ARRAY, GL_INDEX_ARRAY, GL_INDEX_LOGIC_OP, GL_NORMAL_ARRAY, GL_POLYGON_OFFSET_FILL,GL_POLYGON_OFFSET_LINE, GL_POLYGON_OFFSET_POINT, GL_TEXTURE_COORD_ARRAY, and GL_VERTEX_ARRAY are only available if the GL version is 1.1 or greater

Errors

- GL_INVALID_ENUM is generated if *cap* is not an accepted value.

- GL_INVALID_OPERATION is generated if glIsEnabled is executed between the execution of glBegin and the corresponding execution of glEnd.

See Also

glEnable,
glEnableClientState

glIsList

`glIsList`: determine if a name corresponds to a display-list.

C Specification

```
GLboolean glIsList(
    GLuint list)
```

Parameters

list	Specifies a potential display-list name.

Description

glIsList returns GL_TRUE if *list* is the name of a display list and returns GL_FALSE otherwise.

Errors

- GL_INVALID_OPERATION is generated if glIsList is executed between the execution of glBegin and the corresponding execution of glEnd.

See Also

glCallList,
glCallLists,
glDeleteLists,
glGenLists,
glNewList

glIsTexture

`glIsTexture`: determine if a name corresponds to a texture.

C Specification

```
GLboolean glIsTexture(
   GLuint texture)
```

Parameters

texture	Specifies a value that may be the name of a texture.

Description

glIsTexture 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.

Notes

glIsTexture 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.

See Also

glBindTexture,
glCopyTexImage1D,
glCopyTexImage2D,
glDeleteTextures,
glGenTextures,
glGet,
glGetTexParameter,
glTexImage1D,
glTexImage2D,
glTexParameter

10 L

glLight

`glLightf, glLighti, glLightfv, glLightiv`: set light source parameters.

C Specification

```
void glLightf(
    GLenum light,
    GLenum pname,
    GLfloat param)
void glLighti(
    GLenum light,
    GLenum pname,
    GLint param)
void glLightfv(
    GLenum light,
    GLenum pname,
    const GLfloat *params)
void glLightiv(
    GLenum light,
    GLenum pname,
    const GLint *params)
```

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 ≥ *i* < GL_MAX_LIGHTS.
pname	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.
param	Specifies the value that parameter *pname* of light source *light* will be set to.
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 ≥*i* < GL_MAX_LIGHTS.
pname	Specifies a light source parameter for light. GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_POSITION, GL_SPOT_CUTOFF, GL_SPOT_DIRECTION, GL_SPOT_EXPONENT, GL_CONSTANT_ATTENUATION,GL_LINEAR_ATTENUATION, and GL_QUADRATIC_ATTENUATION are accepted.
params	Specifies a pointer to the value or values that parameter *pname* of light source *light* will be set to.

Description

glLight 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 ≥ *i* < GL_MAX_LIGHTS. *pname* specifies one of ten light source parameters, again by symbolic name. *params* is either a single value or a pointer to an array that contains the new values.

To enable and disable lighting calculation, call glEnable and glDisable with argument GL_LIGHTING. Lighting is initially disabled. When it 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 ten light parameters are as follows:

GL_AMBIENT

params contains four integer or floating-point values that specify the ambient RGBA intensity of the light. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The initial ambient light intensity is (0, 0, 0, 1).

GL_DIFFUSE

params 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 initial value for GL_LIGHT0 is (1, 1, 1, 1); for other lights, the initial value is (0, 0, 0, 0).

GL_SPECULAR

params 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 initial value for GL_LIGHT0 is (1, 1, 1, 1); for other lights, the initial value is (0, 0, 0, 0).

GL_POSITION

params 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, the light is treated as a directional source. Diffuse and specular lighting calculations take the light's direction, but not its actual position, into account, and attenuation is disabled. Otherwise, diffuse and specular lighting calculations are based on the actual location of the light in eye coordinates, and attenuation is enabled. The initial position is (0, 0, 1, 0); thus, the initial light source is directional, parallel to, and in the direction of the Z axis.

GL_SPOT_DIRECTION

params 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 initially. The initial direction is (0, 0, 1).

GL_SPOT_EXPONENT

params 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 GL_SPOT_CUTOFF, next paragraph). The initial spot exponent is 0, resulting in uniform light distribution.

GL_SPOT_CUTOFF

params 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 initial spot cutoff is 180, resulting in uniform light distribution.

GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION, GL_QUADRATIC_ATTENUATION

params 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 initial attenuation factors are (1, 0, 0), resulting in no attenuation.

Notes

It is always the case that GL_LIGHT*i* = GL_LIGHT0 + *i*.

Errors

- GL_INVALID_ENUM is generated if either *light* or *pname* is not an accepted value.
- GL_INVALID_VALUE is generated if a spot exponent value is specified outside the range [0,128], or if spot cutoff is specified outside the range [0,90] (except for the special value 180), or if a negative attenuation factor is specified.
- GL_INVALID_OPERATION is generated if glLight is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGetLight
glIsEnabled with argument GL_LIGHTING

See Also

glColorMaterial,
glLightModel,
glMaterial

glLightModel

`glLightModelf, glLightModeli, glLightModelfv, glLightModeliv`: set the lighting model parameters.

C Specification

```
void glLightModelf(
    GLenum pname,
    GLfloat param)
void glLightModeli(
    GLenum pname,
    GLint param)
void glLightModelfv(
    GLenum pname,
    const GLfloat *params)
void glLightModeliv(
    GLenum pname,
    const GLint *params)
```

Parameters

pname	Specifies a single-valued lighting model parameter. GL_LIGHT_MODEL_LOCAL_VIEWER and GL_LIGHT_MODEL_TWO_SIDE are accepted.
param	Specifies the value that *param* will be set to.
pname	Specifies a lighting model parameter. GL_LIGHT_MODEL_AMBIENT, GL_LIGHT_MODEL_LOCAL_VIEWER, and GL_LIGHT_MODEL_TWO_SIDE are accepted.
params	Specifies a pointer to the value or values that *params* will be set to.

Description

glLightModel sets the lighting model parameter. *pname* names a parameter and *params* gives the new value. There are three lighting model parameters:

GL_LIGHT_MODEL_AMBIENT

params contains four integer or floating-point values that specify the ambient RGBA intensity of the entire scene. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to - 1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The initial ambient scene intensity is (0.2, 0.2, 0.2, 1.0).

GL_LIGHT_MODEL_LOCAL_VIEWER

params is a single integer or floating-point value that specifies how specular reflection angles are computed. If *params* is 0 (or 0.0), specular reflection angles take the view direction to be parallel to and in the direction of the - Z axis, regardless of the location of the vertex in eye coordinates. Otherwise, specular reflections are computed from the origin of the eye coordinate system. The initial value is 0.

GL_LIGHT_MODEL_TWO_SIDE

params is a single integer or floating-point value that specifies whether one- or two-sided lighting calculations are done for polygons. It has no effect on the lighting calculations for points, lines, or bitmaps. If *params* is 0 (or 0.0), one-sided lighting is specified, and only the *front* material parameters are used in the lighting equation. Otherwise, two-sided lighting is specified. In this case, vertices of back-facing polygons are lighted using the *back* material parameters, and have their normals reversed before the lighting equation is evaluated. Vertices of front-facing polygons are always lighted using the *front* material parameters, with no change to their normals. The initial value is 0.

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 if they evaluate to a negative value.

The alpha component of the resulting lighted color is set to the alpha value of the material diffuse reflectance.

In color index mode, the value of the lighted index of a vertex ranges from the ambient to the specular values passed to glMaterial using GL_COLOR_INDEXES. Diffuse and specular coefficients, computed with a (.30, .59, .11) weighting of the lights' colors, the shininess of the material, and the same reflection and attenuation equations as in the RGBA case, determine how much above ambient the resulting index is.

Errors

- GL_INVALID_ENUM is generated if *pname* is not an accepted value.
- GL_INVALID_OPERATION is generated if glLightModel is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

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

See Also

glLight,
glMaterial

glLineStipple

`glLineStipple`: specify the line stipple pattern.

C Specification

```
void glLineStipple(
    GLint factor,
    GLushort pattern)
```

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, 256] and defaults to 1.
pattern	Specifies a 16-bit integer whose bit pattern determines which fragments of a line will be drawn when the line is rasterized. Bit zero is used first; the default pattern is all 1s.

Description

Line stippling masks out certain fragments produced by rasterization; those fragments will not be drawn. The masking is achieved by using three parameters: the 16-bit line stipple pattern *pattern*, the repeat count *factor*, and an integer stipple counter *s*. Counter *s* is reset to 0 whenever glBegin is called, and before each line segment of a glBegin(GL_LINES)/glEnd sequence is generated. It is incremented after each fragment of a unit width aliased line segment is generated, or after each *i* fragments of an *i* width line segment are generated. The *i* fragments associated with count *s* are masked out if

pattern bit (s / factor) mod 16 = 0

otherwise, these fragments are sent to the frame buffer. Bit zero of *pattern* is the least significant bit.

Anti-aliased lines are treated as a sequence of $1 \times width$ rectangles for purposes of stippling. Whether rectangle s is rasterized or not depends on the fragment rule described for aliased lines, counting rectangles rather than groups of fragments.

To enable and disable line stippling, call glEnable and glDisable with argument GL_LINE_STIPPLE. When enabled, the line stipple pattern is applied as described above. When disabled, it is as if the pattern were all 1s. Initially, line stippling is disabled.

Errors

- GL_INVALID_OPERATION is generated if glLineStipple is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_LINE_STIPPLE_PATTERN
glGet with argument GL_LINE_STIPPLE_REPEAT
glIsEnabled with argument GL_LINE_STIPPLE

See Also

glLineWidth,
glPolygonStipple

glLineWidth

`glLineWidth`: specify the width of rasterized lines.

C Specification

```
void glLineWidth(
    GLfloat width)
```

Parameters

width	Specifies the width of rasterized lines. The initial value is 1.

Description

glLineWidth specifies the rasterized width of both aliased and anti-aliased lines. Using a line width other than 1 has different effects, depending on whether line anti-aliasing is enabled. To enable and disable line anti-aliasing, call glEnable and glDisable with argument GL_LINE_SMOOTH. Line anti-aliasing is initially disabled.

If line anti-aliasing is disabled, the actual width is determined by rounding the supplied width to the nearest integer. (If the rounding results in the value 0, it is as if the line width were 1.) If $|\Delta x| \geq |\Delta y|$, *i* pixels are filled in each column that is rasterized, where i is the rounded value of width. Otherwise, *i* pixels are filled in each row that is rasterized.

If anti-aliasing is enabled, line rasterization produces a fragment for each pixel square that intersects the region lying within the rectangle having width equal to the current line width, length equal to the actual length of the line, and centered on the mathematical line segment. The coverage value for each fragment is the window coordinate area of the intersection of the rectangular region with the corresponding pixel square. This value is saved and used in the final rasterization step.

Not all widths can be supported when line anti-aliasing is enabled. If an unsupported width is requested, the nearest supported width is used. Only width 1 is guaranteed to be supported; others depend on the implementation. To query the range of supported widths and the size difference between supported widths within the range, call glGet with arguments GL_LINE_WIDTH_RANGE and GL_LINE_WIDTH_GRANULARITY.

Notes

The line width specified by glLineWidth is always returned when GL_LINE_WIDTH is queried. Clamping and rounding for aliased and anti-aliased lines have no effect on the specified value.

Non anti-aliased 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 anti-aliased lines, rounded to the nearest integer value.

Errors

- GL_INVALID_VALUE is generated if *width* is less than or equal to 0.

- GL_INVALID_OPERATION is generated if glLineWidth is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

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

See Also

glEnable

glListBase

`glListBase`: set the display-list base for glCallLists.

C Specification

```
void glListBase(
    GLuint base)
```

Parameters

base	Specifies an integer offset that will be added to glCallLists offsets to generate display-list names. The initial value is 0.

Description

glCallLists specifies an array of offsets. Display-list names are generated by adding *base* to each offset. Names that reference valid display lists are executed; the others are ignored.

Errors

- GL_INVALID_OPERATION is generated if glListBase is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_LIST_BASE

See Also

glCallLists

glLoadIdentity

`glLoadIdentity`: replace the current matrix with the identity matrix.

C Specification

```
void glLoadIdentity(void)
```

Description

glLoadIdentity replaces the current matrix with the identity matrix. It is semantically equivalent to calling glLoadMatrix with the identity matrix:

1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1

but in some cases it is more efficient.

Errors

- GL_INVALID_OPERATION is generated if glLoadIdentity is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_MATRIX_MODE
glGet with argument GL_MODELVIEW_MATRIX
glGet with argument GL_PROJECTION_MATRIX
glGet with argument GL_TEXTURE_MATRIX

See Also

glLoadMatrix,
glMatrixMode,
glMultMatrix,
glPushMatrix

glLoadMatrix

`glLoadMatrixd, glLoadMatrixf`: replace the current matrix with the specified matrix.

C Specification

```
void glLoadMatrixd(
    const GLdouble *m)
void glLoadMatrixf(
    const GLfloa *m)
```

Parameters

m	Specifies a pointer to 16 consecutive values, which are used as the elements of a 44 column-major matrix.

Description

glLoadMatrix replaces the current matrix with the one whose elements are specified by *m*. The current matrix is the projection matrix, modelview matrix, or texture matrix, depending on the current matrix mode (see glMatrixMode).

The current matrix, M, defines a transformation of coordinates. For instance, assume M refers to the modelview matrix. If v = (v[0], v[1], v[2], v[3]) is the set of object coordinates of a vertex, and *m* points to an array of 16 single- or double-precision floating-point values m[0], m[1],... ,m[15], then the modelview transformation M(v) does the following:

$$Mv = \begin{matrix} m[0] & m[4] & m[8] & m[12] \\ m[1] & m[5] & m[9] & m[13] \\ m[2] & m[6] & m[10] & m[14] \\ m[3] & m[7] & m[11] & m[15] \end{matrix} \times \begin{matrix} v[0] \\ v[1] \\ v[2] \\ v[3] \end{matrix}$$

Where × denotes matrix multiplication.

Projection and texture transformations are similarly defined.

Notes

While the elements of the matrix may be specified with single or double precision, the GL implementation may store or operate on these values in less than single precision.

Errors

- GL_INVALID_OPERATION is generated if glLoadMatrix is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_MATRIX_MODE
glGet with argument GL_MODELVIEW_MATRIX
glGet with argument GL_PROJECTION_MATRIX
glGet with argument GL_TEXTURE_MATRIX

See Also

glLoadIdentity,
glMatrixMode,
glMultMatrix,
glPushMatrix

glLoadName

`glLoadName`: load a name onto the name stack.

C Specification

```
void glLoadName(
    GLuint name)
```

Parameters

name	Specifies a name that will replace the top value on 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. glLoadName causes *name* 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.

Errors

- GL_INVALID_OPERATION is generated if glLoadName is called while the name stack is empty.
- GL_INVALID_OPERATION is generated if glLoadName is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_NAME_STACK_DEPTH
glGet with argument GL_MAX_NAME_STACK_DEPTH

See Also

glInitNames,
glPushName,
glRenderMode,
glSelectBuffer

gluLoadSamplingMatrices

`gluLoadSamplingMatrices`: load NURBS sampling and culling matrices.

C Specification

```
void gluLoadSamplingMatrices(
    GLUnurbs* nurb,
    const GLfloat *model,
    const GLfloat *perspective,
    const GLint *view)
```

Parameters

nurb	Specifies the NURBS object (created with gluNewNurbsRenderer).
model	Specifies a modelview matrix (as from a glGetFloatv call).
perspective	Specifies a projection matrix (as from a glGetFloatv call).
view	Specifies a viewport (as from a glGetIntegerv call).

Description

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

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

See Also

gluGetNurbsProperty,
gluNewNurbsRenderer,
gluNurbsProperty

glLogicOp

`glLogicOp`: specify a logical pixel operation for color index rendering.

C Specification

```
void glLogicOp(
    GLenum opcode)
```

Parameters

opcode	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, and GL_OR_INVERTED. The initial value is GL_COPY.

Description

glLogicOp specifies a logical operation that, when enabled, is applied between the incoming color index or RGBA color and the color index or RGBA color at the corresponding location in the frame buffer. To enable or disable the logical operation, call glEnable and glDisable using the symbolic constant GL_COLOR_LOGIC_OP for RGBA mode or GL_INDEX_LOGIC_OP for color index mode. The initial value is disabled for both operations.

opcode	**Resulting Value**
GL_CLEAR	0
GL_SET	1
GL_COPY	s
GL_COPY_INVERTED	!s
GL_NOOP	!d
GL_INVERT	d
GL_AND	s & d
GL_NAND	! (s & d)
GL_OR	s | d
GL_NOR	! (s | d)
GL_XOR	s ^ d
GL_EQUIV	! (s ^ d)
GL_AND_REVERSE	s & !d
GL_AND_INVERTED	!s & d

opcode	**Resulting Value**
GL_OR_REVERSE	s | !d
GL_OR_INVERTED	!s | d

opcode is a symbolic constant chosen from the list above. In the explanation of the logical operations, *s* represents the incoming color index and *d* represents the index in the frame buffer. Standard C-language operators are used. As these bitwise operators suggest, the logical operation is applied independently to each bit pair of the source and destination indices or colors.

Notes

Color index logical operations are always supported. RGBA logical operations are supported only if the GL version is 1.1 or greater.

When more than one RGBA color or index buffer is enabled for drawing, logical operations are performed separately for each enabled buffer, using for the destination value the contents of that buffer (see glDrawBuffer).

Errors

- GL_INVALID_ENUM is generated if *opcode* is not an accepted value.
- GL_INVALID_OPERATION is generated if glLogicOp is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_LOGIC_OP_MODE.
glIsEnabled with argument GL_COLOR_LOGIC_OP or GL_INDEX_LOGIC_OP.

See Also

glAlphaFunc,
glBlendFunc,
glDrawBuffer,
glEnable,
glStencilOp

gluLookAt

gluLookAt: define a viewing transformation.

C Specification

```
void gluLookAt(
    GLdouble eyeX.
    GLdouble eyeY,
    GLdouble eyeZ,
    GLdouble centerX,
    GLdouble centerY,
    GLdouble centerZ,
    GLdouble upX,
    GLdouble upY,
    GLdouble upZ)
```

Parameters

eyeX, eyeY, eyeZ	Specifies the position of the eye point.
centerX, centerY, centerZ	Specifies the position of the reference point.
upX, upY, upZ	Specifies the direction of the up-vector.

Description

gluLookAt creates a viewing matrix derived from an eye point, a reference point indicating the center of the scene, and an up-vector.

The matrix maps the reference point to the negative Z axis and the eye point to the origin. When a typical projection matrix is used, the center of the scene therefore maps to the center of the viewport. Similarly, the direction described by the up-vector projected onto the viewing plane is mapped to the positive Y axis so that it points upward in the viewport. The up-vector must not be parallel to the line of sight from the eye point to the reference point.

Let

F = (centerx - eyex, centery - eyey, centerz- eyez), let *up* be the vector (upX, upY, upZ), and then normalize as follows:

$$f = \frac{F}{||F||}$$

and

$$up = \frac{up}{||up||}$$

Finally, let s = f × up, and u = s × f.

M is then constructed as follows:

```
          M = s[0]  s[1]  s[2] 0
                 u[0]  u[1] u[2] 0
         - f[0 - f[1]- f[2] 0
                    0       0       0   1
```

and gluLookAt is equivalent to

glMultMatrixf(M);
glTranslated (-eyex, -eyey, -eyez);

See Also

glFrustum,
gluPerspective

11 M

glXMakeCurrent

`glXMakeCurrent`: attach a GLX context to a window or a GLX pixmap.

C Specification

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

Parameters

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

Description

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

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

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

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

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

Notes

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

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

Errors

- BadMatch is generated if *drawable* was not created with the same X screen and visual as *ctx*. It is also generated if *drawable* is None and *ctx* is not NULL.
- BadAccess is generated if *ctx* was current to another *thread* at the time glXMakeCurrent was called.
- GLXBadDrawable is generated if *drawable* is not a valid GLX drawable.
- GLXBadContext is generated if *ctx* is not a valid GLX context.
- GLXBadContextState is generated if glXMakeCurrent is executed between the execution of glBegin and the corresponding execution of glEnd.
- GLXBadContextState is also generated if the rendering context current to the calling thread has GL renderer state GL_FEEDBACK or GL_SELECT.
- GLXBadCurrentWindow is generated if there are pending GL commands for the previous context and the current drawable is a window that is no longer valid.
- BadAlloc may be generated if the server has delayed allocation of ancillary buffers until glXMakeCurrent is called, only to find that it has insufficient resources to complete the allocation.

See Also

glXCreateContext,
glXCreateGLXPixmap

glMap1

`glMap1d, glMap1f`: define a one-dimensional evaluator.

C Specification

```
void glMap1d(
    GLenum target,
    GLdouble u1,
    GLdouble u2,
    GLint stride,
    GLint order,
const GLdouble *points)
void glMap1f(
    GLenum target,
    GLfloat u1,
    GLfloat u2,
    GLint stride,
    GLint order,
const GLfloat *points)
```

Parameters

target	Specifies the kind of values that are generated by the evaluator. Symbolic constants 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, and GL_MAP1_TEXTURE_COORD_4 are accepted.
u1, u2	Specify a linear mapping of *u*, as presented to glEvalCoord1, to $u hat$, the variable that is evaluated by the equations specified by this command.
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.

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 glVertex, glNormal, glTexCoord, and glColor commands, except that the generated values do not update the current normal, texture coordinates, or color.

All polynomial or rational polynomial splines of any degree (up to the maximum degree supported by the GL implementation) can be described using evaluators. These include almost all splines used in compute graphics: B-splines, Bezier curves, Hermite splines, and so on.

Evaluators define curves based on Bernstein polynomials. Define p($\hat{u}$) as

Equation 11-1 $$p(\hat{u}) = \sum_{i=0}^{n} B_i^n(\hat{u}) R_i$$

where R$_i$ is a control point and Bin($\hat{u}$) is the *i*th Bernstein polynomial of degree *n* (order = n + 1):

Equation 11-2 $$B_i^n(\hat{u}) = \binom{n}{i} \hat{u}^i (1 - \hat{u})^{n-i}$$

Recall that:

Equation 11-3 $$0^0 \equiv 1 and \binom{n}{0} \equiv 1$$

glMap1 is used to define the basis and to specify what kind of values are produced. Once defined, a map can be enabled and disabled by calling glEnable and glDisable with the map name, one of the nine predefined values for *target* described below. glEvalCoord1 evaluates the one-dimensional maps that are enabled. When glEvalCoord1 presents a value *u*, the Bernstein functions are evaluated using $\hat{u}$, where

Equation 11-4 $$\hat{u} = \frac{u - u1}{u2 - u1}$$

target is a symbolic constant that indicates what kind of control points are provided in points, and what output is generated when the map is evaluated. It can assume one of nine predefined values:

GL_MAP1_VERTEX_3

Each control point is three floating-point values representing *x*, *y*, and *z*. Internal glVertex3 commands are generated when the map is evaluated.

GL_MAP1_VERTEX_4

Each control point is four floating-point values representing *x*, *y*, *z*, and *w*. Internal glVertex4 commands are generated when the map is evaluated.

GL_MAP1_INDEX

Each control point is a single floating-point value representing a color index. Internal glIndex commands are generated when the map is evaluated but the current index is not updated with the value of these glIndex commands.

GL_MAP1_COLOR_4

Each control point is four floating-point values representing red, green, blue, and alpha. Internal glColor4 commands are generated when the map is evaluated but the current color is not updated with the value of these glColor4 commands.

GL_MAP1_NORMAL

Each control point is three floating-point values representing the *x*, *y*, and *z* components of a normal vector. Internal glNormal commands are generated when the map is evaluated but the current normal is not updated with the value of these glNormal commands.

GL_MAP1_TEXTURE_COORD_1

Each control point is a single floating-point value representing the *s* texture coordinate. Internal glTexCoord1 commands are generated when the map is evaluated but the current texture coordinates are not updated with the value of these glTexCoord commands.

GL_MAP1_TEXTURE_COORD_2

Each control point is two floating-point values representing the *s* and *t* texture coordinates. Internal glTexCoord2 commands are generated when the map is evaluated but the current texture coordinates are not updated with the value of these glTexCoord commands.

GL_MAP1_TEXTURE_COORD_3

Each control point is three floating-point values representing the *s*, *t*, and *r* texture coordinates. Internal glTexCoord3 commands are generated when the map is evaluated but the current texture coordinates are not updated with the value of these glTexCoord commands.

GL_MAP1_TEXTURE_COORD_4

Each control point is four floating-point values representing the *s*, *t*, *r*, and *q* texture coordinates. Internal glTexCoord4 commands are generated when the map is evaluated but the current texture coordinates are not updated with the value of these glTexCoord commands.

stride, *order*, and *points* 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* specifies how many float or double locations to advance the internal memory pointer to reach the next control point.

Notes

As is the case with all GL commands that accept pointers to data, it is as if the contents of *points* were copied by glMap1 before glMap1 returns. Changes to he contents of *points* have no effect after glMap1 is called.

Errors

- GL_INVALID_ENUM is generated if *target* is not an accepted value.
- GL_INVALID_VALUE is generated if *u1* is equal to *u2*.
- GL_INVALID_VALUE is generated if *stride* is less than the number of values in a control point.

- GL_INVALID_VALUE is generated if *order* is less than 1 or greater than the return value of GL_MAX_EVAL_ORDER.
- GL_INVALID_OPERATION is generated if glMap1 is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

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

See Also

glBegin,
glColor,
glEnable,
glEvalCoord,
glEvalMesh,
glEvalPoint,
glMap2,
glMapGrid,
glNormal,
glTexCoord,
glVertex

glMap2

`glMap2d, glMap2f`: define a two-dimensional evaluator.

C Specification

```
void glMap2d(
    GLenum target,
    GLdouble u1,
    GLdouble u2,
    GLint ustride,
    GLint uorder,
    GLdouble v1,
    GLdouble v2,
    GLint vstride,
    GLint vorder,
    const GLdouble *points)
void glMap2f(
    GLenum target,
    GLfloat u1,
    GLfloat u2,
    GLint ustride,
    GLint uorder,
    GLfloat v1,
    GLfloat v2,
    GLint vstride,
    GLint vorder,
    const GLfloat*points)
```

Parameters

target	Specifies the kind of values that are generated by the evaluator. Symbolic constants 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, and GL_MAP2_TEXTURE_COORD_4 are accepted.
u1, u2	Specify a linear mapping of *u*, as presented to glEvalCoord2, to û, one of the two variables that are evaluated by the equations specified by this command. Initially, *u1* is 0 and *u2* is 1.
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 *v* control point indices, 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. The initial value of *ustride* is 0.
uorder	Specifies the dimension of the control point array in the *u* axis. Must be positive. The initial value is 1.

v1, v2	Specify a linear mapping of ˆv, as presented to glEvalCoord2, to one of the two variables that are evaluated by the equations specified by this command. Initially, *v1* is 0 and *v2* is 1.
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 indices, 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. The initial value of *vstride* is 0.
vorder	Specifies the dimension of the control point array in the *v* axis. Must be positive. The initial value is 1.

points

Specifies a pointer to the array of control 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 glVertex, glNormal, glTexCoord, and glColor commands, except that the generated values do not update the current normal, texture coordinates, or color.

All polynomial or rational polynomial splines of any degree (up to the maximum degree supported by the GL implementation) can be described using evaluators. These include almost all surfaces used in computer graphics, including B-spline surfaces, NURBS surfaces, Bezier surfaces, and so on.

Evaluators define surfaces based on bivariate Bernstein polynomials. Define p($\hat{u}$, *v*) as

Equation 11-5

$$p(\hat{u}, \hat{v}) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_i^n(\hat{u}) B_j^m(\hat{v}) R_{ij}$$

where R_{ij} is a control point, $B_i^n(\hat{u})$ is the *i*th Bernstein polynomial of degree *n* (*uorder* = *n* + *1*)

Equation 11-6

$$B_i^n(\hat{u}) = \binom{n}{i} \hat{u}^i (1 - \hat{u})^{n-i}$$

and $B_j^m(v)$ is the *j*th Bernstein polynomial of degree *m* (*vorder* = *m* + *1*)

Equation 11-7

$$B_j^m(\hat{v}) = \binom{m}{j} \hat{v}^j (1 - \hat{v})^{m-j}$$

Recall that

Equation 11-8

$$0^0 \equiv 1 and \binom{n}{0} \equiv 1$$

glMap2 is used to define the basis and to specify what kind of values are produced. Once defined, a map can be enabled and disabled by calling glEnable and glDisable with the map name, one of the nine predefined values for *target*, described below. WhenglEvalCoord2 presents values *u* and *v*, the bivariate Bernstein polynomials are evaluated using $\hat{u}$ and v, where

Equation 11-9

$$\hat{u} = \frac{u - u1}{u2 - u1}$$

$$\hat{v} = \frac{v - v1}{v2 - v1}$$

target is a symbolic constant that indicates what kind of control points are provided in *points*, and what output is generated when the map is evaluated. It can assume one of nine predefined values:

GL_MAP2_VERTEX_3

Each control point is three floating-point values representing *x*, *y*, and *z*. Internal glVertex3 commands are generated when the map is evaluated.

GL_MAP2_VERTEX_4

Each control point is four floating-point values representing *x*, *y*, *z*, and *w*. Internal glVertex4 commands are generated when the map is evaluated.

GL_MAP2_INDEX

Each control point is a single floating-point value representing a color index. Internal glIndex commands are generated when the map is evaluated but the current index is not updated with the value of these glIndex commands.

GL_MAP2_COLOR_4

Each control point is four floating-point values representing red, green, blue, and alpha. Internal glColor4 commands are generated when the map is evaluated but the current color is not updated with the value of these glColor4 commands.

GL_MAP2_NORMAL

Each control point is three floating-point values representing the *x*, *y*, and *z* components of a normal vector. Internal glNormal commands are generated when the map is evaluated but the current normal is not updated with the value of these glNormal commands.

GL_MAP2_TEXTURE_COORD_1

Each control point is a single floating-point value representing the *s* texture coordinate. Internal glTexCoord1 commands are generated when the map is evaluated but the current texture coordinates are not updated with the value of these glTexCoord commands.

GL_MAP2_TEXTURE_COORD_2

Each control point is two floating-point values representing the *s* and *t* texture coordinates. Internal glTexCoord2 commands are generated when the map is evaluated but the current texture coordinates are not updated with the value of these glTexCoord commands.

GL_MAP2_TEXTURE_COORD_3

Each control point is three floating-point values representing the *s*, *t*, and *r* texture coordinates. Internal glTexCoord3 commands are generated when the map is evaluated but the current texture coordinates are not updated with the value of these glTexCoord commands.

GL_MAP2_TEXTURE_COORD_4

Each control point is four floating-point values representing the *s*, *t*, *r*, and *q* texture coordinates. Internal glTexCoord4 commands are generated when the map is evaluated but the current texture coordinates are not updated with the value of these glTexCoord commands.

ustride, *uorder*, *vstride*, *vorder*, and *points* 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. There are $uorder \times vorder$ control points in the array. *ustride* specifies how many float or double locations are skipped to advance the internal memory pointer from control point R_{ij} to control point $R_{(i+1)j}$. *vstride* specifies how many float or double locations are skipped to advance the internal memory pointer from control point R_{ij} to control point $R_{i(j+1)}$.

Notes

As is the case with all GL commands that accept pointers to data, it is as if the contents of *points* were copied by glMap2 before glMap2 returns. Changes to the contents of *points* have no effect after glMap2 is called.

Initially, GL_AUTO_NORMAL is enabled. If GL_AUTO_NORMAL is enabled, normal vectors are generated when either GL_MAP2_VERTEX_3 or GL_MAP2_VERTEX_4 is used to generate vertices.

Errors

- GL_INVALID_ENUM is generated if *target* is not an accepted value.
- GL_INVALID_VALUE is generated if *u1* is equal to *u2*, or if *v1* is equal to *v2*.
- GL_INVALID_VALUE is generated if either *ustride* or *vstride* is less than the number of values in a control point.
- GL_INVALID_VALUE is generated if either *uorder* or *vorder* is less than 1 or greater than the return value of GL_MAX_EVAL_ORDER.
- GL_INVALID_OPERATION is generated if glMap2 is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

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

See Also

glBegin,
glColor,
glEnable,
glEvalCoord,
glEvalMesh,
glEvalPoint,
glMap1,
glMapGrid,
glNormal,
glTexCoord,
glVertex

glMapGrid

`glMapGrid1d, glMapGrid1f, glMapGrid2d, glMapGrid2f`: define a one- or two-dimensional mesh.

C Specification

```
void glMapGrid1d(
    GLint un,
    GLdouble u1,
    GLdouble u2)
void glMapGrid1f(
    GLint un,
    GLfloat u1,
    GLfloat u2)
void glMapGrid2d(
    GLint un,
    GLdouble u1,
    GLdouble u2,
    GLint vn,
    GLdouble v1,
    GLdouble v2)
void glMapGrid2f(
    GLint un,
    GLfloat u1,
    GLfloat u2,
    GLint vn,
    GLfloat v1,
    GLfloat v2)
```

Parameters

un	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).

Description

glMapGrid and glEvalMesh are used together to efficiently generate and evaluate a series of evenly-spaced map domain values. glEvalMesh steps through the integer domain of a one- or two-dimensional grid, whose range is the domain of the evaluation maps specified by glMap1 and glMap2.

glMapGrid1 and glMapGrid2 specify the linear grid mappings between the *i* (or *i* and *j*) integer grid coordinates, to the *u* (or *u* and *v*) floating-point evaluation map coordinates. See glMap1 and glMap2 for details of how *u* and *v* coordinates are evaluated.

glMapGrid1 specifies a single linear mapping such that integer grid coordinate 0 maps exactly to *u1*, and integer grid coordinate *un* maps exactly to *u2*. All other integer grid coordinates *i* are mapped so that

$u = i\,(u2 \;\; u1)\, /\, un + u1$

glMapGrid2 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 used identically by glEvalMesh and glEvalPoint.

Errors

- GL_INVALID_VALUE is generated if either un or vn is not positive.
- GL_INVALID_OPERATION is generated if glMapGrid is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_MAP1_GRID_DOMAIN
glGet with argument GL_MAP2_GRID_DOMAIN
glGet with argument GL_MAP1_GRID_SEGMENTS
glGet with argument GL_MAP2_GRID_SEGMENTS

See Also

glEvalCoord,
glEvalMesh,
glEvalPoint,
glMap1,
glMap2

glMaterials

`glMaterialf, glMateriali, glMaterialfv, glMaterialiv`: specify material parameters for the lighting model.

C Specification

```
void glMaterialf(
    GLenum face,
    GLenum pname,
    GLfloat param)
void glMateriali(
    GLenum face,
    GLenum pname,
    GLint param)
void glMaterialfv(
    GLenum face,
    GLenum pname,
    const GLfloat *params)
void glMaterialiv(
    GLenum face,
    GLenum pname,
    const GLint *params)
```

Parameters

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 single-valued material parameter of the face or faces that is being updated. Must be GL_SHININESS.
param	Specifies the value that parameter GL_SHININESS will be set to.
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 GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_EMISSION, GL_SHININESS, GL_AMBIENT_AND_DIFFUSE, or GL_COLOR_INDEXES.
params	Specifies a pointer to the value or values that *pname* will be set to.

Description

glMaterial assigns values to material parameters. There are two matched sets of material parameters. One, the *front-facing* set, is used to shade points, lines, bitmaps, and all polygons (when two-sided lighting is disabled), or just front-facing polygons (when two-sided lighting is enabled). The other set, *back-facing*, is used to shade back-facing polygons only when two-sided lighting is enabled. Refer to the glLightModel reference page for details concerning one- and two-sided lighting calculations.

glMaterial takes three arguments. The first, *face*, specifies whether the GL_FRONT materials, the GL_BACK materials, or both GL_FRONT_AND_BACK materials will be modified. The second, *pname*, specifies which of several parameters in one or both sets will be modified. The third, *params*, specifies what value or values will be assigned to the specified parameter.

Material parameters are used in the lighting equation that is optionally applied to each vertex. The equation is discussed in the glLightModel reference page. The parameters that can be specified using glMaterial, and their interpretations by the lighting equation, are as follows:

GL_AMBIENT

params contains four integer or floating-point values that specify the ambient RGBA reflectance of the material. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to - 1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The initial ambient reflectance for both front- and back-facing materials is (0.2, 0.2, 0.2, 1.0).

GL_DIFFUSE

params 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 initial diffuse reflectance for both front- and back-facing materials is (0.8, 0.8, 0.8, 1.0).

GL_SPECULAR

params 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 initial specular reflectance for both front- and back-facing materials is (0, 0, 0, 1).

GL_EMISSION

params 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 initial emission intensity for both front- and back-facing materials is (0, 0, 0, 1).

GL_SHININESS

params 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 initial specular exponent for both front- and back-facing materials is 0.

GL_AMBIENT_AND_DIFFUSE

Equivalent to calling glMaterial twice with the same parameter values, once with GL_AMBIENT and once with GL_DIFFUSE.

GL_COLOR_INDEXES

params 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. Refer to the glLightModel reference page for a discussion of color index lighting.

Notes

The material parameters can be updated at any time. In particular, glMaterial can be called between a call to glBegin and the corresponding call to glEnd. If only a single material parameter is to be changed per vertex, however, glColorMaterial is preferred over glMaterial (see glColorMaterial).

Errors

- GL_INVALID_ENUM is generated if either *face* or *pname* is not an accepted value.
- GL_INVALID_VALUE is generated if a specular exponent outside the range [0,128] is specified.

Associated Gets

glGetMaterial

See Also

glColorMaterial,
glLight,
glLightModel

glMatrixMode

`glMatrixMode`: specify which matrix is the current matrix.

C Specification

```
void glMatrixMode(
    GLenum mode)
```

Parameters

mode	Specifies which matrix stack is the target for subsequent matrix operations. Three values are accepted: GL_MODELVIEW, GL_PROJECTION, and GL_TEXTURE. The initial value is GL_MODELVIEW.

Description

glMatrixMode sets the current matrix mode. *mode* can assume one of three values:

GL_MODELVIEW

Applies subsequent matrix operations to the modelview matrix stack.

GL_PROJECTION

Applies subsequent matrix operations to the projection matrix stack.

GL_TEXTURE

Applies subsequent matrix operations to the texture matrix stack.

To find out which matrix stack is currently the target of all matrix operations, call glGet with argument GL_MATRIX_MODE. The initial value is GL_MODELVIEW.

Errors

- GL_INVALID_ENUM is generated if *mode* is not an accepted value.
- GL_INVALID_OPERATION is generated if glMatrixMode is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_MATRIX_MODE

See Also

glLoadMatrix,
glPushMatrix

glMultMatrix

`glMultMatrixd, glMultMatrixf`: multiply the current matrix with the specified matrix.

C Specification

```
void glMultMatrixd(
    const GLdouble *m)
void glMultMatrixf(
    const GLfloat *m)
```

Parameters

m	Points to 16 consecutive values that are used as the elements of a 4×4 column-major matrix.

Description

glMultMatrix multiplies the current matrix with the one specified using *m*, and replaces the current matrix with the product.

The current matrix is determined by the current matrix mode (see glMatrixMode). It is either the projection matrix, modelview matrix, or the texture matrix.

Examples

If the current matrix is *C*, and the coordinates to be transformed are, *v = (v[0], v[1], v[2], v[3]*. Then the current transformation is $C \times v$, or

$$\begin{matrix} C[0] & C[4] & C[8] & C[12] \\ C[1] & C[5] & C[9] & C[13] \\ C[2] & C[6] & C[10] & C[14] \\ C[3] & C[7] & C[11] & C[15] \end{matrix} \times \begin{matrix} v[0] \\ v[1] \\ v[2] \\ v[3] \end{matrix}$$

Calling glMultMatrix with an argument of $m = m[0], m[1], \ldots, m[15]$ replaces the current transformation with $(C \times m) \times v$, or

$$\begin{matrix} C[0] & C[4] & C[8] & C[12] \\ C[1] & C[5] & C[9] & C[13] \\ C[2] & C[6] & C[10] & C[14] \\ C[3] & C[7] & C[11] & C[15] \end{matrix} \times \begin{matrix} m[0] & m[4] & m[8] & m[12] \\ m[1] & m[5] & m[5] & m[13] \\ m[2] & m[6] & m[10] & m[14] \\ m[3] & m[7] & m[11] & m[15] \end{matrix} \times \begin{matrix} v[0] \\ v[1] \\ v[2] \\ v[3] \end{matrix}$$

Where $\times$ denotes matrix multiplication, and *v* is represented as a 4×1 matrix.

Notes

While the elements of the matrix may be specified with single or double precision, the GL may store or operate on these values in less than single precision.

In many computer languages, 4×4 arrays are represented in row-major order. The transformations just described represent these matrices in column-major order. The order of the multiplication is important. For example, if the current transformation is a rotation, and glMultMatrix is called with a translation matrix, the translation is done directly on the coordinates to be transformed, while the rotation is done on the results of that translation.

Errors

- GL_INVALID_OPERATION is generated if glMultMatrix is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_MATRIX_MODE
glGet with argument GL_MODELVIEW_MATRIX
glGet with argument GL_PROJECTION_MATRIX
glGet with argument GL_TEXTURE_MATRIX

See Also

glLoadIdentity,
glLoadMatrix,
glMatrixMode,
glPushMatrix

12 N

glNewList

`glNewList, glEndList`: create or replace a display list.

C Specification

```
void glNewList(
    GLuint list,
    GLenum mode)
void glEndList(void)
```

Parameters

list	Specifies the display-list name.
mode	Specifies the compilation mode, which can be GL_COMPILE or GL_COMPILE_AND_EXECUTE.

Description

Display lists are groups of GL commands that have been stored for subsequent execution. Display lists are created with glNewList. All subsequent commands are placed in the display list, in the order issued, until glEndList is called.

glNewList 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 glGenLists and tested for uniqueness with glIsList. The second argument, *mode*, is a symbolic constant that can assume one of two values:

GL_COMPILE

Commands are merely compiled.

GL_COMPILE_AND_EXECUTE

Commands are executed as they are compiled into the display list.

Certain commands are not compiled into the display list but are executed immediately, regardless of the display-list mode. These commands are glColorPointer, glDeleteLists, glDisableClientState, glEdgeFlagPointer, glEnableClientState, glFeedbackBuffer, glFinish, glFlush, glGenLists, glIndexPointer, glInterleavedArrays, glIsEnabled, glIsList, glNormalPointer, glPopClientAttrib, glPixelStore, glPushClientAttrib, glReadPixels, glRenderMode, glSelectBuffer, glTexCoordPointer, glVertexPointer, and all of the glGet commands.

Similarly, glTexImage2D and glTexImage1D are executed immediately and not compiled into the display list when their first argument is GL_PROXY_TEXTURE_2D or GL_PROXY_TEXTURE_1D, respectively.

When glEndList is encountered, the display-list definition is completed by associating the list with the unique name *list* (specified in the glNewList command). If a display list with name *list* already exists, it is replaced only when glEndList is called.

Notes

glCallList and glCallLists can be entered into display lists. Commands in the display list or lists executed by glCallList or glCallLists are not included in the display list being created, even if the list creation mode is

GL_COMPILE_AND_EXECUTE.

A display list is just a group of commands and arguments, so errors generated by commands in a display list must be generated when the list is executed. If the list is created in GL_COMPILE mode, errors are not generated until the list is executed.

Errors

- GL_INVALID_VALUE is generated if *list* is 0.
- GL_INVALID_ENUM is generated if *mode* is not an accepted value.
- GL_INVALID_OPERATION is generated if glEndList is called without a preceding glNewList, or if glNewList is called while a display list is being defined.
- GL_INVALID_OPERATION is generated if glNewList or glEndList is executed between the execution of glBegin and the corresponding execution of glEnd.
- GL_OUT_OF_MEMORY is generated if there is insufficient memory to compile the display list. If the GL version is 1.1 or greater, no change is made to the previous contents of the display list, if any, and no other change is made to the GL state. (It is as if no attempt had been made to create the new display list.)

Associated Gets

glIsList
glGet with argument GL_LIST_INDEX
glGet with argument GL_LIST_MODE

See Also

glCallList,
glCallLists,
glDeleteLists,
glGenLists

glNextVisibilityTesthp

`glNextVisibilityTesthp` - end the current visibility test and begin the next.

C Specification

```
void glNextVisibilityTesthp(void)
```

Parameters

None

Description

glNextVisibilityTesthp is used in conjunction with glVisibilityBufferhp to test the visibility of primitives against the current contents of the depth buffer.

With glVisibilityBufferhp, the programmer specifies a boolean array in which to store the results of the visibility tests; a call to glNextVisibilityTesthp finishes the current test (eventually placing the results of the test in the current position of the array specified by glVisiblityBufferhp), and beginning the next test, whose results will be placed in the subsequent entry of the array.

Visibility Testing is enabled by a call to glEnable(VISIBILITY_TEST_hp). Making this call to glEnable (if VISIBILITY_TEST_hp has not already been enabled) causes Visibility Testing to begin, and sets the result destination to be the first entry in the buffer specified by glVisibilityBufferhp.

Notes

None

Errors

GL_INVALID_OPERATION is generated if glNextVisibilityTesthp is called and VISIBILITY_TEST_hp has not been enabled via glEnable().

GL_INVALID_OPERATION is generated if glVisibiliyBufferhp is called between a call to glBegin and the corresponding call to glEnd.

Associated Gets

None

See Also

glVisibilityBufferhp

gluNewNurbsRenderer

`gluNewNurbsRenderer`: create a NURBS object.

C Specification

```
GLUnurbs* gluNewNurbsRenderer(void)
```

Description

gluNewNurbsRenderer creates and returns a pointer to a new NURBS object. This object must be referred to when calling NURBS rendering and control functions. A return value of 0 means that there is not enough memory to allocate the object.

See Also

gluBeginCurve,
gluBeginSurface,
gluBeginTrim,
gluDeleteNurbsRenderer,
gluNurbsCallback,
gluNurbsProperty

gluNewQuadric

`gluNewQuadric`: create a quadrics object.

C Specification

```
GLUquadric* gluNewQuadric(void)
```

Description

gluNewQuadric creates and returns a pointer to a new quadrics object. This object must be referred to when calling quadrics rendering and control functions. A return value of 0 means that there is not enough memory to allocate the object.

See Also

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

gluNewTess

`gluNewTess`: create a tessellation object.

C Specification

```
GLUtesselator* gluNewTess(void)
```

Description

gluNewTess creates and returns a pointer to a new tessellation object. This object must be referred to when calling tessellation functions. A return value of 0 means that there is not enough memory to allocate the object.

See Also

gluTessBeginPolygon,
gluDeleteTess,
gluTessCallback

gluNextContour

`gluNextContour`: mark the beginning of another contour.

C Specification

```
void gluNextContour(
GLUtesselator* tess,
GLenum type)
```

Parameters

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

Description

gluNextContour is used in describing polygons with multiple contours. After the first contour has been described through a series of gluTessVertex calls, a gluNextContour call indicates that the previous contour is complete and that the next contour is about to begin.

Another series of gluTessVertex calls is then used to describe the new contour. This process can be repeated until all contours have been described.

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

GLU_EXTERIOR

An exterior contour defines an exterior boundary of the polygon.

GLU_INTERIOR

An interior contour defines an interior boundary of the polygon (such as a hole).

GLU_UNKNOWN

An unknown contour is analyzed by the library to determine if it is interior or exterior.

GLU_CCW, GLU_CW

The first GLU_CCW or GLU_CW contour defined is considered to be exterior. All other contours are considered to be exterior if they are oriented in the same direction (clockwise or counterclockwise) as the first contour, and interior if they are not.

If one contour is of type GLU_CCW or GLU_CW, then all contours must be of the same type (if they are not, then all GLU_CCW and GLU_CW contours will be changed to GLU_UNKNOWN).

Note that there is no real difference between the GLU_CCW and GLU_CW contour types.

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

This command is obsolete and is provided for backward compatibility only. Calls to gluNextContour are mapped to gluTessEndContour followed by gluTessBeginContour.

See Also

gluBeginPolygon,
gluNewTess,
gluTessCallback,
gluTessVertex,
gluTessBeginContour

glNormal

glNormal3b, glNormal3d, glNormal3f, glNormal3i, glNormal3s, glNormal3bv, glNormal3dv, glNormal3fv, glNormal3iv, glNormal3sv: set the current normal vector.

C Specification

```
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)
```

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.

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.

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

See Also

glBegin
glColor,
glIndex,
glNormalPointer,
glTexCoord,
glVertex

glNormalPointer

`glNormalPointer`: define an array of normals.

C Specification

```
void glNormalPointer(
    GLenum type,
    GLsizei stride,
    const GLvoid *pointer)
```

Parameters

type	Specifies the data type of each coordinate in the array. Symbolic constants GL_BYTE, GL_SHORT, GL_INT, GL_FLOAT, and GL_DOUBLE are accepted. The initial value is GL_FLOAT.
stride	Specifies the byte offset between consecutive normals. If *stride* is 0--the initial value--the normals are understood to be tightly packed in the array.
pointer	Specifies a pointer to the first coordinate of the first normal in the array.

Description

glNormalPointer 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; 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 to construct a sequence of primitives (all of the same type) from pre-specified vertex and vertex attribute arrays. Use glArrayElement to specify primitives by indexing vertexes and vertex attributes and glDrawElements to construct a sequence of primitives by indexing vertexes and vertex attributes.

Notes

glNormalPointer is available only if the GL version is 1.1 or greater.

The normal array is initially disabled and isn't 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.

glNormalPointer 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.

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

See Also

glArrayElement,
glColorPointer,
glDrawArrays,
glDrawElements,
glEdgeFlagPointer,
glEnable,
glGetPointerv,
glIndexPointer,
glInterleavedArrays,
glPopClientAttrib,
glPushClientAttrib,
glTexCoordPointer,
glVertexPointer

gluNurbsCallback

`gluNurbsCallback`: define a callback for a NURBS object.

C Specification

```
void gluNurbsCallback(
    GLUnurbs* nurb,
    GLenum which,
    GLvoid (*CallBackFunc)()
```

Parameters

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

Description

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

The one legal callback is GLU_ERROR:

GLU_ERROR

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

See Also

gluErrorString,
gluNewNurbsRenderer

gluNurbsCurve

`gluNurbsCurve`: define the shape of a NURBS curve.

C Specification

```
void gluNurbsCurve(
    GLUnurbs* nurb,
    GLint knotCount,
    GLfloat *knots,
    GLint stride,
    GLfloat *control,
    GLint order,
    GLenum type)
```

Parameters

nurb	Specifies the NURBS object (created with gluNewNurbsRenderer).
knotCount	Specifies the number of knots in *knots*. *knotCount* equals the number of control points plus the order.
knots	Specifies an array of *knotCount* non-decreasing knot values.
stride	Specifies 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. The coordinates must agree with *type*, specified below.
order	Specifies the order of the NURBS curve. *order* equals *degree + 1*, hence a cubic curve has an order of 4.
type	Specifies the type of the curve. If this curve is defined within a gluBeginCurve/gluEndCurve pair, then the type can be any of the valid one-dimensional evaluator types (such as GL_MAP1_VERTEX_3 or GL_MAP1_COLOR_4). Between a gluBeginCurve/gluEndCurve pair, the only valid types are GLU_MAP1_TRIM_2 and GLU_MAP1_TRIM_3.

Description

Use gluNurbsCurve to describe a NURBS curve.

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

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

Notes

To define trim curves that stitch well, use gluPwlCurve.

See Also

gluBeginCurve,
gluBeginTrim,
gluNewNurbsRenderer,
gluPwlCurve

gluNurbsPrperty

`gluNurbsProperty`: set a NURBS property.

C Specification

```
void gluNurbsProperty(
    GLUnurbs* nurb,
    GLenum property,
    GLfloat value)
```

Parameters

nurb	Specifies the NURBS object (created with gluNewNurbsRenderer).
property	Specifies the property to be set. Valid values are GLU_SAMPLING_TOLERANCE, GLU_DISPLAY_MODE, GLU_CULLING, GLU_AUTO_LOAD_MATRIX, GLU_PARAMETRIC_TOLERANCE, GLU_SAMPLING_METHOD, GLU_U_STEP, or GLU_V_STEP.
value	Specifies the value of the indicated property. It may be a numeric value, or one of GLU_PATH_LENGTH, GLU_PARAMETRIC_ERROR, or GLU_DOMAIN_DISTANCE.

Description

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

GLU_SAMPLING_METHOD

Specifies how a NURBS surface should be tessellated. *value* may be one of GLU_PATH_LENGTH, GLU_PARAMETRIC_ERROR, or GLU_DOMAIN_DISTANCE. 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. The initial value of GLU_SAMPLING_METHOD is GLU_PATH_LENGTH.

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_SAMPLING_TOLERANCE

Specifies the maximum length, in pixels to use when the sampling method is set to GLU_PATH_LENGTH. The NURBS code is conservative when rendering a curve or surface, so the actual length can be somewhat shorter. The initial value is 50.0 pixels.

GLU_PARAMETRIC_TOLERANCE

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

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

Specifies the number of sample points per unit length taken along the *v* axis in parametric coordinate. It is needed when GLU_SAMPLING_METHOD is set to GLU_DOMAIN_DISTANCE. The initial value is 100.

GLU_DISPLAY_MODE

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

GLU_CULLING

value is a boolean value that, when set to GL_TRUE, indicates that a NURBS curve should be discarded prior to tessellation if its control points lie outside the current viewport. The initial value is GL_FALSE.

GLU_AUTO_LOAD_MATRIX

value is a boolean value. When set to GL_TRUE, the NURBS code downloads the projection matrix, the modelview matrix, and the viewport from the GL server to compute sampling and culling matrices for each NURBS curve that is rendered. Sampling and culling matrices are required to determine the tesselation of a NURBS surface into line segments or polygons and to cull a NURBS surface if it lies outside the viewport.

If this mode is set to GL_FALSE, then the program needs to provide a projection matrix, a modelview matrix, and a viewport for the NURBS renderer to use to construct sampling and culling matrices. This can be done with the gluLoadSamplingMatrices function. This mode is initially set to GL_TRUE. Changing it from GL_TRUE to GL_FALSE does not affect the sampling and culling matrices until gluLoadSamplingMatrices is called.

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.

gluGetString can be used to determine the GLU version.

See Also

gluGetNurbsProperty,
gluLoadSamplingMatrices,
gluNewNurbsRenderer,
gluGetString

gluNurbsSurface

`gluNurbsSurface`: define the shape of a NURBS surface.

C Specification

```
void gluNurbsSurface(
    GLUnurbs* nurb,
    GLint sKnotCount,
    GLfloat* sKnots,
    GLint tKnotCount,
    GLfloat* tKnots,
    GLint sStride,
    GLint tStride,
    GLfloat* control,
    GLint sOrder,
    GLint tOrder,
    GLenum type)
```

Parameters

nurb	Specifies the NURBS object (created with gluNewNurbsRenderer).
sKnotCount	Specifies the number of knots in the parametric u direction.
sKnots	Specifies an array of *sKnotCount* non-decreasing knot values in the parametric *u* direction.
tKnotCount	Specifies the number of knots in the parametric v direction.
tKnots	Specifies an array of *tKnotCount* non-decreasing knot 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, hence a surface that is cubic in *u* has a *u* order of 4.
tOrder	Specifies the order of the NURBS surface in the parametric *v* direction. The order is one more than the degree, hence a surface that is cubic in *v* has a *v* order of 4.
type	Specifies type of the surface. type can be any of the valid two-dimensional evaluator types (such as GL_MAP2_VERTEX_3 or GL_MAP2_COLOR_4).

Description

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

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

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

Note that a gluNurbsSurface with *sKnotCount* knots in the *u* direction and *tKnotCount* knots in the *v* direction with orders *sOrder* and *tOrder* must have (*sKnotCount* - *sOrder*) $\times$ (*tKnotCount* - *tOrder*) control points.

See Also

gluBeginSurface,
gluBeginTrim,
gluNewNurbsRenderer,
gluNurbsCurve,
gluPwlCurve

13 O

glOrtho

`glOrtho:` multiply the current matrix with an orthographic matrix.

C Specification

```
void glOrtho(
    GLdouble left,
    GLdouble right,
    GLdouble bottom,
    GLdouble top,
    GLdouble zNear,
    GLdouble zFar)
```

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.
zNear, zFar	Specify the distances to the nearer and farther depth clipping planes. These values are negative if the plane is to be behind the viewer.

Description

glOrtho describes a transformation that produces a parallel projection. The current matrix (see glMatrixMode) is multiplied by this matrix and the result replaces the current matrix, as if glMultMatrix were called with the following matrix as its argument:

$$\begin{matrix} A & 0 & 0 & t_x \\ 0 & B & 0 & t_y \\ 0 & 0 & C & t_z \\ 0 & 0 & 0 & 1 \end{matrix}$$

where

A = 2 / (right left)

B = 2 / (top bottom)

C = 2 / (far near)

t_x = (right + left) / (right left)

t_y = (top + bottom) / (top bottom)

t_z = (zFar + zNear) / (zFar zNear)

Typically, the matrix mode is GL_PROJECTION, and *(left, bottom, zNear*) and (*right, top, zNear*) 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). *zFar* specifies the location of the far clipping plane. Both *zNear* and *zFar* can be either positive or negative.

Use glPushMatrix and glPopMatrix to save and restore the current matrix stack.

Errors

- GL_INVALID_OPERATION is generated if glOrtho is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_MATRIX_MODE
glGet with argument GL_MODELVIEW_MATRIX
glGet with argument GL_PROJECTION_MATRIX
glGet with argument GL_TEXTURE_MATRIX

See Also

glFrustum,
glMatrixMode,
glMultMatrix,
glPushMatrix,
glViewport

gluOrtho2D

gluOrtho2D: define a 2D orthographic projection matrix.

C Specification

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

Parameters

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

Description

gluOrtho2D sets up a two-dimensional orthographic viewing region. This is equivalent to calling glOrtho with *near* = 1 and *far* = 1.

See Also

glOrtho,
gluPerspective

14 P

gluPartialDisk

`gluPartialDisk`: draw an arc of a disk.

C Specification

```
void gluPartialDisk(
    GLUquadric* quad,
    GLdouble inner,
    GLdouble outer,
    GLint slices,
    GLint loops,
    GLdouble start,
    GLdouble sweep)
```

Parameters

quad	Specifies a quadrics object (created with gluNewQuadric).
inner	Specifies the inner radius of the partial disk (can be 0).
outer	Specifies the outer radius of the partial disk.
slices	Specifies the number of subdivisions around the Z axis.
loops	Specifies the number of concentric rings about the origin into which the partial disk is subdivided.
start	Specifies the starting angle, in degrees, of the disk portion.
sweep	Specifies the sweep angle, in degrees, of the disk portion.

Description

gluPartialDisk renders a partial disk on the Z = 0 plane. A partial disk is similar to a full disk, except that only the subset of the disk from *start* through *start* + *sweep* is included (where 0 degrees is along the +Y axis, 90 degrees along the +X axis, 180 along the Y axis, and 270 along the X axis).

The partial disk has a radius of *outer*, and contains a concentric circular hole with a radius of *inner*. If inner is 0, then no hole is generated. The partial disk is subdivided around the Z axis into slices (like pizza slices), and also about the Z axis into rings (as specified by *slices* and *loops*, respectively).

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

If texturing is turned on (with gluQuadricTexture), texture coordinates are generated linearly such that where *r* = *outer*, the value at (*r*, 0, 0) is (1.0, 0.5), at 0, *r*, 0) it is (0.5, 1.0), at (- *r*, 0, 0) it is (0.0, 0.5), and at (0, - *r*, 0) it is (0.5, 0.0).

See Also

gluCylinder,
gluDisk,
gluNewQuadric,
gluQuadricOrientation,
gluQuadricTexture,
gluSphere

glPassThrough

glPassThrough: place a marker in the feedback buffer.

C Specification

```
void glPassThrough(
    GLfloat token)
```

Parameters

token	Specifies a marker value to be placed in the feedback buffer following a GL_PASS_THROUGH_TOKEN.

Description

Feedback is a GL render 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. See the glFeedbackBuffer reference page for a description of the feedback buffer and the values in it.

glPassThrough inserts a user-defined marker in the feedback buffer when it is executed in feedback mode. *token* 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.

Notes

glPassThrough is ignored if the GL is not in feedback mode.

Errors

- GL_INVALID_OPERATION is generated if glPassThrough is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_RENDER_MODE

See Also

glFeedbackBuffer,
glRenderMode

gluPerspective

`gluPerspective`: set up a perspective projection matrix.

C Specification

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

Parameters

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

Description

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

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

Given f defined as cotangent(*fovy* / 2), the generated matrix is

$$\begin{matrix} A & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & B & C \\ 0 & 0 & -1 & 0 \end{matrix}$$

where:

$A = f \,/\, aspect$

$B = (zFar + zNear) \,/\, (zNear - zFar)$

$C = (2 \times zFar \times zNear) \,/\, (zNear \times zFar)$

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 $\log_2 r$ bits of depth buffer precision are lost. Because r approaches infinity as *zNear* approaches 0, *zNear* must never be set to 0.

See Also

glFrustum,
glLoadIdentity,
glMultMatrix,
gluOrtho2D

gluPickMatrix

`gluPickMatrix`: define a picking region.

C Specification

```
void gluPickMatrix(
    GLdouble x,
    GLdouble y,
    GLdouble delX,
    GLdouble delY,
    GLint *viewport)
```

Parameters

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

Description

gluPickMatrix creates a projection matrix that can be used to restrict drawing to a small region of the viewport. This is typically useful to determine what objects are being drawn near the cursor. Use gluPickMatrix to restrict drawing to a small region around the cursor. Then, enter selection mode (with glRenderMode) and 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 gluPickMatrix is multiplied by the current matrix just as if glMultMatrix is called with the generated matrix. To effectively use the generated pick matrix for picking, first call glLoadIdentity to load an identity matrix onto the perspective matrix stack. Then call gluPickMatrix, and finally, call a command (such as gluPerspective) to multiply the perspective matrix by the pick matrix.

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

See Also

glGet,
glLoadIdentity,
glMultMatrix,
glRenderMode,
gluPerspective

glPixelMap

glPixelMapfv, glPixelMapuiv, glPixelMapusv: set up pixel transfer maps.

C Specification

```
void glPixelMapfv(
    GLenum map,
    GLsizei mapsize,
    const GLfloat *values)
void glPixelMapuiv(
    GLenum map,
    GLsizei mapsize,
    const GLuint *values)
void glPixelMapusv(
    GLenum map,
    GLsizei mapsize,const GLushort *values)
```

Parameters

map	Specifies a symbolic map name. 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, or GL_PIXEL_MAP_A_TO_A.
mapsize	Specifies the size of the map being defined.
values	Specifies an array of *mapsize* values.

Description

glPixelMap sets up translation tables, or *maps*, used by glCopyPixels, glCopyTexImage1D, glCopyTexImage2D, glCopyTexSubImage1D, glCopyTexSubImage2D, glDrawPixels, glReadPixels, glTexImage1D, glTexImage2D, glTexSubImage1D, and glTexSubImage2D. Use of these maps is described completely in the glPixelTransfer reference page, and partly in the reference pages for the pixel and texture image commands. Only the specification of the maps is described in this reference page.

map is a symbolic map name, indicating one of ten maps to set.

mapsize specifies the number of entries in the map, and values is a pointer to an array of *mapsize* map values.

The ten maps are as follows:

GL_PIXEL_MAP_I_TO_I

Maps color indices to color indices.

GL_PIXEL_MAP_S_TO_S

Maps stencil indices to stencil indices.

GL_PIXEL_MAP_I_TO_R

Maps color indices to red components.

GL_PIXEL_MAP_I_TO_G

Maps color indices to green components.

GL_PIXEL_MAP_I_TO_B

Maps color indices to blue components.

GL_PIXEL_MAP_I_TO_A

Maps color indices 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 GL_PIXEL_MAP_I_TO_I and GL_PIXEL_MAP_S_TO_S) retain their values in floating-point format, with unspecified mantissa and exponent sizes. Floating-point values specified by glPixelMapfv are converted directly to the internal floating-point format of these maps, 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 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 indices must have $mapsize = 2^n$ for some n or the results are undefined. The maximum allowable size for each map depends on the implementation and can be determined by calling glGet with argument GL_MAX_PIXEL_MAP_TABLE. The single maximum applies to all maps; 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
GL_PIXEL_MAP_S_TO_S	stencil index	stencil index	1	0
GL_PIXEL_MAP_I_TO_R	color index	R	1	0
GL_PIXEL_MAP_I_TO_G	color index	G	1	0
GL_PIXEL_MAP_I_TO_B	color index	B	1	0
GL_PIXEL_MAP_I_TO_A	color index	A	1	0
GL_PIXEL_MAP_R_TO_R	R	R	1	0
GL_PIXEL_MAP_G_TO_G	G	G	1	0
GL_PIXEL_MAP_B_TO_B	B	B	1	0
GL_PIXEL_MAP_A_TO_A	A	A	1	0

Errors

- GL_INVALID_ENUM is generated if *map* is not an accepted value.
- GL_INVALID_VALUE is generated if *mapsize* is less than one or larger than
- GL_MAX_PIXEL_MAP_TABLE. GL_INVALID_VALUE is generated if *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 is generated if glPixelMap is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

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

See Also

glCopyPixels,
glCopyTexImage1D,
glCopyTexImage2D,
glCopyTexSubImage1D,
glCopyTexSubImage2D,
glDrawPixels,
glPixelStore,
glPixelTransfer,
glReadPixels,
glTexImage1D,
glTexImage2D,
glTexSubImage1D,
glTexSubImage2D

glPixelStore

`glPixelStoref, glPixelStorei`: set pixel storage modes.

C Specification

```
void glPixelStoref(
    GLenum pname,
    GLfloat param)
void glPixelStorei(
    GLenum pname,
    GLint param)
```

Parameters

pname	Specifies the symbolic name of the parameter to be set. Six 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, and GL_PACK_ALIGNMENT. Six more 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, and GL_UNPACK_ALIGNMENT.
param	Specifies the value that *pname* is set to.

Description

glPixelStore sets pixel storage modes that affect the operation of subsequent glDrawPixels and glReadPixels as well as the unpacking of polygon stipple patterns (see glPolygonStipple), bitmaps (see glBitmap), and texture patterns (see glTexImage1D, glTexImage2D, glTexSubImage1D, and glTexSubImage2D).

pname is a symbolic constant indicating the parameter to be set, and *param* is the new value. Six of the twelve storage parameters affect how pixel data is returned to client memory, and are therefore significant only for glReadPixels commands.

They are as follows:

GL_PACK_SWAP_BYTES

If true, byte ordering for multibyte color components, depth components, color indices, or stencil indices is reversed. That is, if a four-byte component consists 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 indices. 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, GL_PACK_ROW_LENGTH defines the number of pixels in a row. If the first pixel of a row is placed at location p in memory, then the location of the first pixel of the next row is obtained by skipping

$$k = \begin{cases} nl \\ \frac{a}{s}\left\lceil \frac{snl}{a} \right\rceil \end{cases} if \frac{s \geq a}{s < a}$$

components or indices, where n is the number of components or indices in a pixel, l is the number of pixels in a row (GL_PACK_ROW_LENGTH if it is greater than 0, the *width* argument to the pixel routine otherwise), a is the value of GL_PACK_ALIGNMENT, and s is the size, in bytes, of a single component (if a<s, then it is as if a=s). In the case of 1-bit values, the location of the next row is obtained by skipping

$$k = 8a\left\lceil \frac{nl}{8a} \right\rceil$$

components or indices.

The word *component* in this description refers to the non-index values red, green, blue, alpha, and depth. Storage format GL_RGB, for example, has three components per pixel: first red, then green, and finally blue.

GL_PACK_SKIP_PIXELS and GL_PACK_SKIP_ROWS

These values are provided as a convenience to the programmer; they provide no functionality that cannot be duplicated simply by incrementing the pointer passed to glReadPixels. Setting GL_PACK_SKIP_PIXELS to i is equivalent to incrementing the pointer by in components or indices, where n is the number of components or indices in each pixel. Setting GL_PACK_SKIP_ROWS to j is equivalent to incrementing the pointer by jk components or indices, where k is the number of components or indices per row, as just computed in the GL_PACK_ROW_LENGTH section.

GL_PACK_ALIGNMENT

Specifies the alignment requirements for the start of each pixel row in memory. The allowable values are 1 (byte-alignment), 2 (rows aligned to even-numbered bytes), 4 (word-alignment), and 8 (rows start on double-word boundaries).

The other six of the twelve storage parameters affect how pixel data is read from client memory. These values are significant for glDrawPixels, glTexImage1D, glTexImage2D, glTexSubImage1D, glTexSubImage2D, glBitmap, and glPolygonStipple. They are as follows:

GL_UNPACK_SWAP_BYTES

If true, byte ordering for multibyte color components, depth components, color indices, or stencil indices is reversed. That is, if a four-byte component consists 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 indices. 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 relevant only for bitmap data.

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

$$k = \begin{cases} nl & \text{if } s \geq a \\ \frac{a}{s}\left\lceil \frac{snl}{a} \right\rceil & s < a \end{cases}$$

components or indices, where *n* is the number of components or indices in a pixel, l is the number of pixels in a row (GL_UNPACK_ROW_LENGTH if it is greater than 0, the *width* argument to the pixel routine otherwise), *a* is the value of GL_UNPACK_ALIGNMENT, and *s* is the size, in bytes, of a single component (if $a<s$, then it is as if $a=s$). In the case of 1-bit values, the location of the next row is obtained by skipping

$$k = \begin{cases} nl & \text{if } s \geq a \\ \frac{a}{s}\left\lceil \frac{snl}{a} \right\rceil & s < a \end{cases}$$

components or indices.

The word *component* in this description refers to the non-index values red, green, blue, alpha, and depth. Storage format GL_RGB, for example, has three components per pixel: first red, then green, and finally blue.

GL_UNPACK_SKIP_PIXELS and GL_UNPACK_SKIP_ROWS

These values are provided as a convenience to the programmer; they provide no functionality that cannot be duplicated by incrementing the pointer passed to glDrawPixels, glTexImage1D, glTexImage2D, glTexSubImage1D, glTexSubImage2D, glBitmap, or glPolygonStipple. Setting GL_UNPACK_SKIP_PIXELS to *i* is equivalent to incrementing the pointer by *in* components or indices, where *n* is the number of components or indices in each pixel. Setting GL_UNPACK_SKIP_ROWS to *j* is equivalent to incrementing the pointer by *jk* components or indices, where *k* is the number of components or indices per row, as just computed in the GL_UNPACK_ROW_LENGTH section.

GL_UNPACK_ALIGNMENT

Specifies the alignment requirements for the start of each pixel row in memory. The allowable values are 1 (byte-alignment), 2 (rows aligned to even-numbered bytes), 4 (word-alignment), and 8 (rows start on double-word boundaries).

The following table gives the type, initial value, and range of valid values for each storage parameter 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, ∞]
GL_PACK_SKIP_ROWS	integer	0	[0, ∞]
GL_PACK_SKIP_PIXELS	integer	0	[0, ∞]
GL_PACK_ALIGNMENT	integer	4	1, 2, 4 or 8
GL_UNPACK_SWAP_BYTES	boolean	false	true or false
GL_UNPACK_LSB_FIRST	boolean	false	true or false
GL_UNPACK_ROW_LENGTH	integer	0	[0, ∞]
GL_UNPACK_SKIP_ROWS	integer	0	[0, ∞]
GL_UNPACK_SKIP_PIXELS	integer	0	[0, ∞]
GL_UNPACK_ALIGNMENT	integer	4	1, 2, 4 or 8

glPixelStoref can be used to set any pixel store parameter. If the parameter type is boolean, then if *param* is 0, the parameter is false; otherwise it is set to true. If *pname* is a integer type parameter, *param* 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 *param* is 0 and true otherwise.

Notes

The pixel storage modes in effect when glDrawPixels, glReadPixels, glTexImage1D, glTexImage2D, glTexSubImage1D, glTexSubImage2D, 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.

Pixel storage modes are client state and must be pushed and restored using glPushClientAttrib and glPopClientAttrib.

Errors

- GL_INVALID_ENUM is generated if *pname* is not an accepted value.
- GL_INVALID_VALUE is generated if a negative row length, pixel skip, or row skip value is specified, or if alignment is specified as other than 1, 2, 4, or 8.
- GL_INVALID_OPERATION is generated if glPixelStore is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_PACK_SWAP_BYTES
glGet with argument GL_PACK_LSB_FIRST
glGet with argument GL_PACK_ROW_LENGTH
glGet with argument GL_PACK_SKIP_ROWS
glGet with argument GL_PACK_SKIP_PIXELS
glGet with argument GL_PACK_ALIGNMENT
glGet with argument GL_UNPACK_SWAP_BYTES
glGet with argument GL_UNPACK_LSB_FIRST
glGet with argument GL_UNPACK_ROW_LENGTH
glGet with argument GL_UNPACK_SKIP_ROWS
glGet with argument GL_UNPACK_SKIP_PIXELS
glGet with argument GL_UNPACK_ALIGNMENT

See Also

glBitmap,
glDrawPixels,
glPixelMap,
glPixelTransfer,
glPixelZoom,
glPolygonStipple,
glPushClientAttrib,
glReadPixels,
glTexImage1D,
glTexImage2D,
glTexSubImage1D,
glTexSubImage2D

glPixelTransfer

`glPixelTransferf, glPixelTransferi`: set pixel transfer modes.

C Specification

```
void glPixelTransferf(
    GLenum pname,
    GLfloat param)
void glPixelTransferi(
    GLenum pname,
    GLint param)
```

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, or GL_DEPTH_BIAS.
param	Specifies the value that *pname* is set to.

Description

glPixelTransfer sets pixel transfer modes that affect the operation of subsequent glCopyPixels, glCopyTexImage1D, glCopyTexImage2D, glCopyTexSubImage1D, glCopyTexSubImage2D, glDrawPixels, glReadPixels, glTexImage1D, glTexImage2D, glTexSubImage1D, and glTexSubImage2D commands. The algorithms that are specified by pixel transfer modes operate on pixels after they are read from the frame buffer (glCopyPixels glCopyTexImage1D, glCopyTexImage2D, glCopyTexSubImage1D, glCopyTexSubImage2D, and glReadPixels), or unpacked from client memory (glDrawPixels, glTexImage1D, glTexImage2D, glTexSubImage1D, and glTexSubImage2D). 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 (see glPixelStore) control the unpacking of pixels being read from client memory, and the packing of pixels being written back into client memory.

Pixel transfer operations handle four fundamental pixel types: *color, color index, depth*, and *stencil*. *Color* pixels consist of four floating-point values with unspecified mantissa and exponent sizes, scaled such that 0 represents zero intensity and 1 represents full intensity. *Color indices* 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. And 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 glPixelMap.

Color index

Each color index is shifted left by GL_INDEX_SHIFT bits; any bits beyond the number of fraction bits carried by the fixed-point index are filled with zeros. If GL_INDEX_SHIFT is negative, the shift is to the right, again zero 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 indices. If GL_MAP_COLOR is true, each index is masked by 2^n -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 is specified with glPixelMap.

If the resulting pixels are to be written to an RGBA color buffer, or if they are read back to client memory in a format other than GL_COLOR_INDEX, the pixels are converted from indices 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 de-referenced, the index is masked by 2^n - 1, where n is GL_PIXEL_MAP_I_TO_R_SIZE for the red map, GL_PIXEL_MAP_I_TO_G_SIZE for the green map, GL_PIXEL_MAP_I_TO_B_SIZE for the blue map, and GL_PIXEL_MAP_I_TO_A_SIZE for the alpha map. All components taken from the maps are then clamped to the range [0, 1]. The contents of the four maps is specified with glPixelMap.

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 $2^n - 1$, where *n* is GL_PIXEL_MAP_S_TO_S_SIZE, then replaced by the contents of GL_PIXEL_MAP_S_TO_S indexed by the masked value.

The following table gives the type, initial value, and range of valid values for each of the pixel transfer parameters that are set with glPixelTransfer.

pname	**Type**	**Initial Value**	**Valid Range**
GL_MAP_COLOR	boolean	false	true/false
GL_MAP_STENCIL	boolean	false	true/false
GL_INDEX_SHIFT	integer	0	$(-\infty,\infty)$
GL_INDEX_OFFSET	integer	0	$(-\infty,\infty)$
GL_RED_SCALE	float	1	$(-\infty,\infty)$
GL_GREEN_SCALE	float	1	$(-\infty,\infty)$
GL_BLUE_SCALE	float	1	$(-\infty,\infty)$
GL_ALPHA_SCALE	float	1	$(-\infty,\infty)$
GL_DEPTH_SCALE	float	1	$(-\infty,\infty)$
GL_RED_BIAS	float	0	$(-\infty,\infty)$
GL_GREEN_BIAS	float	0	$(-\infty,\infty)$
GL_BLUE_BIAS	float	0	$(-\infty,\infty)$
GL_ALPHA_BIAS	float	0	$(-\infty,\infty)$
GL_DEPTH_BIAS	float	0	$(-\infty,\infty)$

glPixelTransferf can be used to set any pixel transfer parameter. If the parameter type is boolean, 0 implies false and any other value implies true. If pname is an integer parameter, *param* is rounded to the nearest integer.

Likewise, glPixelTransferi can be used to set any of the pixel transfer parameters. Boolean parameters are set to false if *param* is 0 and to true otherwise. *param* is converted to floating point before being assigned to real-valued parameters.

Notes

If a glCopyPixels, glCopyTexImage1D,glCopyTexImage2D, glCopyTexSubImage1D, glCopyTexSubImage2D, glDrawPixels, glReadPixels, glTexImage1D, glTexImage2D, glTexSubImage1D, or glTexSubImage2D command is placed in a display list (see glNewList and glCallList), the pixel transfer mode settings in effect when the display list is executed are the ones that are used. They may be different from the settings when the command was compiled into the display list.

Errors

- GL_INVALID_ENUM is generated if *pname* is not an accepted value.
- GL_INVALID_OPERATION is generated if glPixelTransfer is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

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

See Also

glCallList,
glCopyPixels,
glCopyTexImage1D,
glCopyTexImage2D,
glCopyTexSubImage1D,
glCopyTexSubImage2D,
glDrawPixels,
glNewList,
glPixelMap,
glPixelStore,
glPixelZoom,
glReadPixels,
glTexImage1D,
glTexImage2D,
glTexSubImage1D,
glTexSubImage2D

glPixelZoom

`glPixelZoom`: specify the pixel zoom factors.

C Specification

```
void glPixelZoom(
    GLfloat xfactor,
    GLfloat yfactor)
```

Parameters

xfactor, yfactor	Specify the x and y zoom factors for pixel write operations.

Description

glPixelZoom specifies values for the x and y zoom factors. During the execution of glDrawPixels or glCopyPixels, if (x_r, y_r) is the current raster position, and a given element is in the mth row and nth column of the pixel rectangle, then pixels whose centers are in the rectangle with corners at

$(x_r + n \cdot xfactor, y_r + m \cdot yfactor)$

$(x_r + (n+1) \cdot xfactor, y_r + (m+1) \cdot yfactor)$

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 is generated if glPixelZoom is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_ZOOM_X
glGet with argument GL_ZOOM_Y

See Also

glCopyPixels,
glDrawPixels

glPointSize

`glPointSize`: specify the diameter of rasterized points.

C Specification

```
void glPointSize(
    GLfloat size)
```

Parameters

size	Specifies the diameter of rasterized points. The initial value is 1.

Description

glPointSize specifies the rasterized diameter of both aliased and anti-aliased points. Using a point size other than 1 has different effects, depending on whether point anti-aliasing is enabled. To enable and disable point anti-aliasing, call glEnable and glDisable with argument GL_POINT_SMOOTH. Point anti-aliasing is initially disabled.

If point anti-aliasing is disabled, the actual size is determined by rounding the supplied size to the nearest integer. (If the rounding results in the value 0, it is as if the point size were 1.) If the rounded size is odd, then the center point *(x, y*) of the pixel fragment that represents the point is computed as

$$(\lfloor x_w \rfloor + .5, \lfloor y_w \rfloor + .5)$$

where *w* subscripts indicate window coordinates. All pixels that lie within the square grid of the rounded size centered at *(x, y*) make up the fragment. If the size is even, the center point is

$$(\lfloor x_w + .5 \rfloor, \lfloor y_w + .5 \rfloor)$$

and the rasterized fragment's centers are the half-integer window coordinates within the square of the rounded size centered at (*x, y*). All pixel fragments produced in rasterizing a non anti-aliased point are assigned the same associated data, that of the vertex corresponding to the point.

If anti-aliasing is enabled, then point rasterization produces a fragment for each pixel square that intersects the region lying within the circle having diameter equal to the current point size and centered at the 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 anti-aliasing is enabled. If an unsupported size is requested, the nearest supported size is used. Only size 1 is guaranteed to be supported; others depend on the implementation. To query the range of supported sizes and the size difference between supported sizes within the range, call glGet with arguments GL_POINT_SIZE_RANGE and GL_POINT_SIZE_GRANULARITY.

Notes

The point size specified by glPointSize is always returned when GL_POINT_SIZE is queried. Clamping and rounding for aliased and anti-aliased points have no effect on the specified value.

A non-anti-aliased point size may be clamped to an implementation-dependent maximum. Although this maximum cannot be queried, it must be no less than the maximum value for anti-aliased points, rounded to the nearest integer value.

Errors

- GL_INVALID_VALUE is generated if *size* is less than or equal to 0.
- GL_INVALID_OPERATION is generated if glPointSize is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

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

See Also

glEnable

glPolygonMode

`glPolygonMode`: select a polygon rasterization mode.

C Specification

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

Parameters

face	Specifies the polygons that *mode* applies to. Must be GL_FRONT for front-facing polygons, GL_BACK for back-facing polygons, or GL_FRONT_AND_BACK for front- and back-facing polygons.
mode	Specifies how polygons will be rasterized. Accepted values are GL_POINT, GL_LINE, and GL_FILL. The initial value is GL_FILL for both front- and back-facing polygons.

Description

glPolygonMode controls the interpretation of polygons for rasterization. *face* describes which polygons *mode* applies to: front-facing polygons (GL_FRONT), back-facing polygons (GL_BACK), or both (GL_FRONT_AND_BACK). The polygon mode affects only the final rasterization of polygons. In particular, a polygon's vertices are lit and the polygon is clipped and possibly culled before these modes are applied.

Three modes are defined and can be specified in mode:

GL_POINT

Polygon vertices that are marked as the start of a boundary edge are drawn as points. Point attributes such as GL_POINT_SIZE and GL_POINT_SMOOTH control the rasterization of the points. Polygon rasterization attributes other than GL_POLYGON_MODE have no effect.

GL_LINE

Boundary edges of the polygon are drawn as line segments. They are treated as connected line segments for line stippling; the line stipple counter and pattern are not reset between segments (see glLineStipple). Line attributes such as GL_LINE_WIDTH and GL_LINE_SMOOTH control the rasterization of the lines. Polygon rasterization attributes other than GL_POLYGON_MODE have no effect.

GL_FILL

The interior of the polygon is filled. Polygon attributes such as GL_POLYGON_STIPPLE and GL_POLYGON_SMOOTH control the rasterization of the polygon.

Examples

To draw a surface with filled back-facing polygons and outlined front-facing polygons, call

```
glPolygonMode(GL_FRONT, GL_LINE);
```

Notes

Vertices are marked as boundary or non-boundary with an edge flag. Edge flags are generated internally by the GL when it decomposes polygons; they can be set explicitly using glEdgeFlag.

Errors

- GL_INVALID_ENUM is generated if either *face* or *mode* is not an accepted value.
- GL_INVALID_OPERATION is generated if glPolygonMode is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_POLYGON_MODE

See Also

glBegin,
glEdgeFlag,
glLineStipple,
glLineWidth,
glPointSize,
glPolygonStipple

glPolygonOffset

`glPolygonOffset`: set the scale and bias used to calculate depth values.

C Specification

```
void glPolygonOffset(
    GLfloat factor,
    GLfloat units)
```

Parameters

factor	Specifies a scale factor that 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.

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 $factor \times \Delta z + r \times units$, where z is a measurement of the change in depth relative to the screen area of the polygon, and r is the smallest value that is guaranteed to produce a resolvable 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.

glPolygonOffset is useful for rendering hidden-line images, for applying decals to surfaces, and for rendering solids with highlighted edges.

Notes

glPolygonOffset is available only if the GL version is 1.1 or greater.

glPolygonOffset has no effect on depth coordinates placed in the feedback buffer.

glPolygonOffset has no effect on selection.

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.

See Also

glDepthFunc,
glDisable,
glEnable,
glGet,
glIsEnabled,
glLineWidth,
glStencilOp,
glTexEnv

glPolygonStipple

`glPolygonStipple`: set the polygon stippling pattern.

C Specification

```
void glPolygonStipple(
const GLubyte *mask)
```

Parameters

mask	Specifies a pointer to a 32×32 stipple pattern that will be unpacked from memory in the same way that glDrawPixels unpacks pixels.

Description

Polygon stippling, like line stippling (see glLineStipple), masks out certain fragments produced by rasterization, creating a pattern. Stippling is independent of polygon anti-aliasing.

**mask* is a pointer to a 32×32 stipple pattern that is stored in memory just like the pixel data supplied to a glDrawPixels call 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 32×32 array of 1-bit color indices packed in unsigned bytes. glPixelStore parameters like 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.

To enable and disable polygon stippling, call glEnable and glDisable with argument GL_POLYGON_STIPPLE. Polygon stippling is initially disabled. If it's 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 consists of all 1s.

Errors

- GL_INVALID_OPERATION is generated if glPolygonStipple is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGetPolygonStipple
glIsEnabled with argument GL_POLYGON_STIPPLE

See Also

glDrawPixels,
glLineStipple,
glPixelStore,
glPixelTransfer

glPrioritizeTextures

`glPrioritizeTextures`: set texture residence priority.

C Specification

```
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

glPrioritizeTextures assigns the *n* texture priorities given in *priorities* to the *n* textures named in *textures*.

The 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, 1] before they are assigned. 0 indicates the lowest priority; textures with priority 0 are least likely to be resident. 1 indicates the highest priority; textures with priority 1 are most likely to be resident. However, textures are not guaranteed to be resident until they are used.

glPrioritizeTextures silently ignores attempts to prioritize texture 0, or any texture name that does not correspond to an existing texture.

glPrioritizeTextures 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.

Notes

glPrioritizeTextures 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.

See Also

glAreTexturesResident,
glBindTexture,
glCopyTexImage1D,
glCopyTexImage2D,
glTexImage1D,
glTexImage2D,
glTexParameter

gluProject

gluProject: map object coordinates to window coordinates.

C Specification

```
GLint gluProject(
    GLdouble objX,
    GLdouble objY,
    GLdouble objZ,
    const GLdouble *model,
    const GLdouble *proj,
    const GLint *view,
    GLdouble* winX,
    GLdouble* winY,
    GLdouble* winZ)
```

Parameters

objA, 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	Return the computed window coordinates.

Description

gluProject transforms the specified object coordinates into window coordinates using *model*, *proj*, and *view*. The result is stored in *winX*, *winY*, and *winZ*. A return value of GL_TRUE indicates success, a return value of 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:

$$v = P \times M \times v$$

where P is the current projection matrix proj, M is the current modelview matrix *model* (both represented as 4 4 matrices in column-major order) and "×" represents matrix multiplication.

The window coordinates are then computed as follows:

$$winX = view(0) + view(2)\ \ (v'(0) + 1)/2$$
$$winY = view(1) + view(3)\ \ (v'(1) + 1)/2$$
$$winZ = (v(2) + 1)/\ 2$$

See Also

glGet,
gluUnProject

glPushAttrib

`glPushAttrib, glPopAttrib`: push and pop the server attribute stack.

C Specification

```
void glPushAttrib(
    GLbitfield   mask)
void glPopAttrib(void)
```

Parameters

mask	Specifies a mask that indicates which attributes to save. Values for mask are listed below.

Description

glPushAttrib 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. *mask* is typically constructed by ORing several of these constants together. The special mask GL_ALL_ATTRIB_BITS can be used to save all stackable states.

The symbolic mask constants and their associated GL state are as follows (the second column lists which attributes are saved):

GL_ACCUM_BUFFER_BIT	Accumulation buffer clear value
GL_COLOR_BUFFER_BIT	GL_ALPHA_TEST enable bit Alpha test function and reference value GL_BLEND enable bit Blending source and destination functions Constant blend color Blending equation GL_DITHER enable bit GL_DRAW_BUFFER setting GL_COLOR_LOGIC_OP enable bit GL_INDEX_LOGIC_OP enable bit Logic op function Color mode and index mode clear values Color mode and index mode writemasks

GL_CURRENT_BIT	Current RGBA color Current color index Current normal vector Current texture coordinates Current raster position GL_CURRENT_RASTER_POSITION_VALID flag RGBA color associated with current raster position Color index associated with current raster position Texture coordinates associated with current raster position GL_EDGE_FLAG flag
GL_DEPTH_BUFFER_BIT	GL_DEPTH_TEST enable bit Depth buffer test function Depth buffer clear value GL_DEPTH_WRITEMASK enable bit
GL_ENABLE_BIT	GL_ALPHA_TEST flag GL_AUTO_NORMAL flag GL_BLEND flag Enable bits for the user-definable clipping planes GL_COLOR_MATERIAL GL_CULL_FACE flag GL_DEPTH_TEST flag GL_DITHER flag GL_FOG flag GL_LIGHTi where 0 i < GL_MAX_LIGHTS GL_LIGHTING flag GL_LINE_SMOOTH flag GL_LINE_STIPPLE flag GL_COLOR_LOGIC_OP flag GL_INDEX_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_LINE flag GL_POLYGON_OFFSET_FILL 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 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 1D grid endpoints and divisions 2D grid endpoints and divisions GL_AUTO_NORMAL enable bit
GL_FOG_BIT	GL_FOG enable bit Fog color Fog density Linear fog start Linear fog end Fog index GL_FOG_MODE value
GL_HINT_BIT	GL_PERSPECTIVE_CORRECTION_HINT setting GL_POINT_SMOOTH_HINT setting GL_LINE_SMOOTH_HINT setting GL_POLYGON_SMOOTH_HINT setting GL_FOG_HINT setting
GL_LIGHTING_BIT	GL_COLOR_MATERIAL enable bit GL_COLOR_MATERIAL_FACE value Color material parameters that are tracking the current color Ambient scene color GL_LIGHT_MODEL_LOCAL_VIEWER value GL_LIGHT_MODEL_TWO_SIDE setting GL_LIGHTING enable bit Enable bit for each light Ambient, diffuse, and specular intensity for each light Direction, position, exponent, and cutoff angle for each light Constant, linear, and quadratic attenuation factors for each light Ambient, diffuse, specular, and emissive color for each material Ambient, diffuse, and specular color 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_MODE setting GL_POLYGON_SMOOTH flag GL_POLYGON_STIPPLE enable bit GL_POLYGON_OFFSET_FILL flag GL_POLYGON_OFFSET_LINE flag GL_POLYGON_OFFSET_POINT flag GL_POLYGON_OFFSET_FACTOR GL_POLYGON_OFFSET_UNITS
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 Current texture bindings (for example, GL_TEXTURE_2D_BINDING)
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

glPopAttrib restores the values of the state variables saved with the last glPushAttrib command. Those not saved are left unchanged.

It is an error to push attributes onto a full stack, or to pop attributes off an empty

stack. In either case, the error flag is set and no other change is made to GL state.

Initially, the attribute stack is empty.

Notes

Not all values for GL state can be saved on the attribute stack. For example, render mode state, and select and feedback state cannot be saved. Client state must be saved with glPushClientAttrib.

The depth of the attribute stack depends on the implementation, but it must be at least 16.

Errors

- GL_STACK_OVERFLOW is generated if glPushAttrib is called while the attribute stack is full.
- GL_STACK_UNDERFLOW is generated if glPopAttrib is called while the attribute stack is empty.
- GL_INVALID_OPERATION is generated if glPushAttrib or glPopAttrib is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_ATTRIB_STACK_DEPTH
glGet with argument GL_MAX_ATTRIB_STACK_DEPTH

See Also

glGet,
glGetClipPlane,
glGetError,
glGetLight,
glGetMap,
glGetMaterial,
glGetPixelMap,
glGetPolygonStipple,
glGetString,
glGetTexEnv,
glGetTexGen,
glGetTexImage,
glGetTexLevelParameter,
glGetTexParameter,
glIsEnabled,
glPushClientAttrib

glPushClientAttrib

`glPushClientAttrib, glPopClientAttrib`: push and pop the client attribute stack.

C Specification

```
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

glPushClientAttrib 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. *mask* is typically constructed by ORing 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)

glPopClientAttrib 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

glPushClientAttrib 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.

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

See Also

glColorPointer,
glDisableClientState,
glEdgeFlagPointer,
glEnableClientState,
glGet,
glGetError,
glIndexPointer,
glNormalPointer,
glNewList,
glPixelStore,
glPushAttrib,
glTexCoordPointer,
glVertexPointer

glPushMatrix

`glPushMatrix, glPopMatrix`: push and pop the current matrix stack.

C Specification

```
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.

glPushMatrix pushes the current matrix stack down by one, duplicating the current matrix. That is, after a glPushMatrix call, the matrix on top of the stack is identical to the one below it.

glPopMatrix 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.

Errors

- GL_STACK_OVERFLOW is generated if glPushMatrix is called while the current matrix stack is full.
- GL_STACK_UNDERFLOW is generated if glPopMatrix is called while the current matrix stack contains only a single matrix.
- GL_INVALID_OPERATION is generated if glPushMatrix or glPopMatrix is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

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

See Also

glFrustum,
glLoadIdentity,
glLoadMatrix,
glMatrixMode,
glMultMatrix,
glOrtho,
glRotate,
glScale,
glTranslate,
glViewport

glPushName

`glPushName, glPopName`: push and pop the name stack.

C Specification

```
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 and is initially empty.

glPushName causes *name* to be pushed onto the name stack. glPopName pops one name off the top of the stack.

The maximum name stack depth is implementation-dependent; call GL_MAX_NAME_STACK_DEPTH to find out the value for a particular implementation. 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 the execution of glBegin and the corresponding execution of glEnd. 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.

Errors

- GL_STACK_OVERFLOW is generated if glPushName is called while the name stack is full.
- GL_STACK_UNDERFLOW is generated if glPopName is called while the name stack is empty.
- GL_INVALID_OPERATION is generated if glPushName or glPopName is executed between a call to glBegin and the corresponding call to glEnd.

Associated Gets

glGet with argument GL_NAME_STACK_DEPTH
glGet with argument GL_MAX_NAME_STACK_DEPTH

See Also

glInitNames,
glLoadName,
glRenderMode,
glSelectBuffer

gluPwlCurve

`gluPwlCurve`: describe a piece-wise linear NURBS trimming curve.

C Specification

```
void gluPwlCurve(
    GLUnurbs* nurb,
    GLint count,
    GLfloat* data,
    GLint stride,
    GLenum type)
```

Parameters

nurb	Specifies the NURBS object (created with gluNewNurbsRenderer).
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 type of curve. Must be either GLU_MAP1_TRIM_2 or GLU_MAP1_TRIM_3.

Description

gluPwlCurve describes a piece-wise linear trimming curve for a NURBS surface. A piece-wise linear curve consists of a list of coordinates of points in the parameter space for the NURBS surface to be trimmed. These points are connected with line segments to form a curve. If the curve is an approximation to a curve that is not piece-wise linear, the points should be close enough in parameter space that the resulting path appears curved at the resolution used in the application.

If *type* is GLU_MAP1_TRIM_2, then it describes a curve in two-dimensional (*u* and *v*) parameter space. If it is GLU_MAP1_TRIM_3, then it describes a curve in two-dimensional homogeneous (*u*, *v*, and *w*) parameter space. See the gluBeginTrim reference page for more information about trimming curves.

Notes

To describe a trim curve that closely follows the contours of a NURBS surface, call gluNurbsCurve.

See Also

gluBeginCurve,
gluBeginTrim,
gluNewNurbsRenderer,
gluNurbsCurve

15 Q

gluQuadricCallback

`gluQuadricCallback`: define a callback for a quadrics object.

C Specification

```
void gluQuadricCallback(
    GLUquadric* quad,
    GLenum which,
    GLvoid (*CallBackFunc)()
```

Parameters

quad	Specifies the quadrics object (created with gluNewQuadric).
which	Specifies the callback being defined. The only valid value is GLU_ERROR.
CallBackFunc	Specifies the function to be called.

Description

gluQuadricCallback is used to define a new callback to be used by a quadrics object. If the specified callback is already defined, then it is replaced. If *CallBackFunc* is NULL, then any existing callback is erased.

The one legal callback is GLU_ERROR:

GLU_ERROR

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

See Also

gluErrorString,
gluNewQuadric

gluQuadricDrawStyle

`gluQuadricDrawStyle`: specify the draw style desired for quadrics.

C Specification

```
void gluQuadricDrawStyle(
    GLUquadric* quad,
    GLenum draw)
```

Parameters

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

Description

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

GLU_FILL

Quadrics are rendered with polygon primitives. The polygons are drawn in a counterclockwise fashion with respect to their normals (as defined with gluQuadricOrientation).

GLU_LINE

Quadrics are rendered as a set of lines.

GLU_SILHOUETTE

Quadrics are rendered as a set of lines, except that edges separating coplanar faces will not be drawn.

GLU_POINT

Quadrics are rendered as a set of points.

See Also

gluNewQuadric,
gluQuadricNormals,
gluQuadricOrientation,
gluQuadricTexture

gluQuadricNormals

`gluQuadricNormals:` specify what kind of normals are desired for quadrics.

C Specification

```
void gluQuadricNormals(
    GLUquadric* quad,
    GLenum normal)
```

Parameters

quad	Specifies the quadrics object (created with gluNewQuadric).
normal	Specifies the desired type of normals. Valid values are GLU_NONE, GLU_FLAT, and GLU_SMOOTH.

Description

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

GLU_NONE

No normals are generated.

GLU_FLAT

One normal is generated for every facet of a quadric.

GLU_SMOOTH

One normal is generated for every vertex of a quadric. This is the initial value.

See Also

gluNewQuadric,
gluQuadricDrawStyle,
gluQuadricOrientation,
gluQuadricTexture

gluQuadricOrientation

`gluQuadricOrientation`: specify inside/outside orientation for quadrics.

C Specification

```
void gluQuadricOrientation(
    GLUquadric* quad,
    GLenum orientation)
```

Parameters

quad	Specifies the quadrics object (created with gluNewQuadric).
orientation	Specifies the desired orientation. Valid values are GLU_OUTSIDE and GLU_INSIDE.

Description

gluQuadricOrientation specifies what kind of orientation is desired for quadrics rendered with *quad*. The *orientation* values are as follows:

GLU_OUTSIDE

Quadrics are drawn with normals pointing outward (the initial value).

GLU_INSIDE

Quadrics are drawn with normals pointing inward.

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

See Also

gluNewQuadric,
gluQuadricDrawStyle,
gluQuadricNormals,
gluQuadricTexture

gluQuadricTexture

`gluQuadricTexture`: specify if texturing is desired for quadrics.

C Specification

```
void gluQuadricTexture(
    GLUquadric* quad,
    GLboolean texture)
```

Parameters

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

Description

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

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

See Also

gluNewQuadric,
gluQuadricDrawStyle,
gluQuadricNormals,
gluQuadricOrientation

glXQueryExtension

`glXQueryExtension`: indicate whether the GLX extension is supported.

C Specification

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

Parameters

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

Description

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

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

Notes

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

See Also

glXQueryVersion

glXQueryExtensionsString

`glXQueryExtensionsString`: return list of supported extensions.

C Specification

```
constchar *glXQueryExtensionsString(
    Display *dpy,
    int screen)
```

Parameters

dpy	Specifies the connection to the X server.
screen	Specifies the screen.

Description

glXQueryExtensionsString 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.

Notes

glXQueryExtensionsString is available only if the GLX version is 1.1 or greater.

glXQueryExtensionsString only returns information about GLX extensions. Call glGetString to get a list of GL extensions.

See Also

glGetString,
glXQueryVersion,
glXQueryServerString,
glXGetClientString

glXQueryServerString

`glXQueryServerString`: return string describing the server.

C Specification

```
constchar *glXQueryServerString(
    Display *dpy,
    int screen,
    int name)
```

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.

Description

glXQueryServerString 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.

Notes

glXQueryServerString 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.

glXQueryServerString 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.

See Also

glXQueryVersion,
glXGetClientString,
glXQueryExtensionsString

glXQueryVersion

16 R

glRasterPos

`glRasterPos2d, glRasterPos2f, glRasterPos2i, glRasterPos2s, glRasterPos3d, glRasterPos3f, glRasterPos3i, glRasterPos3s, glRasterPos4d, glRasterPos4f, glRasterPos4i, glRasterPos4s, glRasterPos2dv, glRasterPos2fv, glRasterPos2iv, glRasterPos2sv, glRasterPos3dv, glRasterPos3fv, glRasterPos3iv, glRasterPos3sv, glRasterPos4dv, glRasterPos4fv, glRasterPos4iv, glRasterPos4sv`: specify the raster position for pixel operations.

C Specification

```
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 3D position in window coordinates. This position, called the raster position, is used to position pixel and bitmap write operations. It is maintained with subpixel accuracy. See glBitmap, glDrawPixels, and glCopyPixels.

The current raster position consists of three window coordinates (*x*, *y*, *z*), a clip coordinate value (*w*), an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The *w* coordinate is a clip coordinate, because *w* is not projected to window coordinates. glRasterPos4 specifies object coordinates *x*, *y*, *z*, and *w* explicitly.

glRasterPos3 specifies object coordinate *x*, *y*, and *z* explicitly, while *w* is implicitly set to 1. glRasterPos2 uses the argument values for *x* and *y* while implicitly setting *z* and *w* to 0 and 1.

The object coordinates presented by glRasterPos are treated just like those of a glVertex command: They are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, then it is projected and scaled to window coordinates, which become the new current raster position, and the GL_CURRENT_RASTER_POSITION_VALID flag is set. If the vertex *is* culled, then the valid bit is cleared and the current raster position and associated color and texture coordinates are undefined.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then GL_CURRENT_RASTER_COLOR (in RGBA mode) or GL_CURRENT_RASTER_INDEX (in color index mode) is set to the color produced by the lighting calculation (see glLight, glLightModel, and glShadeModel). If lighting is disabled, current color (in RGBA mode, state variable GL_CURRENT_COLOR) or color index (in color index mode, state variable GL_CURRENT_INDEX) is used to update the current raster color.

Likewise, GL_CURRENT_RASTER_TEXTURE_COORDS is updated as a function of GL_CURRENT_TEXTURE_COORDS, based on the texture matrix and the texture generation functions (see glTexGen). Finally, the distance from the origin of the eye coordinate system to the vertex as transformed by only the modelview matrix replaces GL_CURRENT_RASTER_DISTANCE.

Initially, the current raster position is (0, 0, 0, 1), the current raster distance is 0, the valid bit is set, the associated RGBA color is (1, 1, 1, 1), the associated color index is 1, and the associated texture coordinates are (0, 0, 0, 1). In RGBA mode, GL_CURRENT_RASTER_INDEX is always 1; in color index mode, the current raster RGBA color always maintains its initial value.

Notes

The raster position is modified both by glRasterPos and by glBitmap.

When the raster position coordinates are invalid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to GL state).

Calling glDrawElements may leave the current color or index indeterminate. If glRasterPos is executed while the current color or index is indeterminate, the current raster color or current raster index remains indeterminate.

To set a valid raster position outside the viewport, first set a valid raster position, then call glBitmap with NULL as the *bitmap* parameter.

Errors

- GL_INVALID_OPERATION is generated if glRasterPos is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID
glGet with argument GL_CURRENT_RASTER_DISTANCE

glGet with argument GL_CURRENT_RASTER_COLOR
glGet with argument GL_CURRENT_RASTER_INDEX
glGet with argument GL_CURRENT_RASTER_TEXTURE_COORDS

See Also

glBitmap,
glCopyPixels,
glDrawElements,
glDrawPixels,
glLight,
glLightModel,
glShadeModel,
glTexCoord,
glTexGen,
glVertex

glReadBuffer

`glReadBuffer`: select a color buffer source for pixels.

C Specification

```
void glReadBuffer(
    GLenum mode)
```

Parameters

mode	Specifies a color buffer. Accepted values are GL_FRONT_LEFT, GL_FRONT_RIGHT, GL_BACK_LEFT, GL_BACK_RIGHT, GL_FRONT, GL_BACK, GL_LEFT, GL_RIGHT, and GL_AUXi, where *i* is between 0 and GL_AUX_BUFFERS - 1.

Description

glReadBuffer specifies a color buffer as the source for subsequent glReadPixels, glCopyTexImage1D, glCopyTexImage2D, glCopyTexSubImage1D, glCopyTexSubImage2D, and glCopyPixels commands. *mode* 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.

Non-stereo double-buffered configurations have only a front left and a back left buffer. Single-buffered configurations have a front left and a front right buffer if stereo, and only a front left buffer if non-stereo. It is an error to specify a nonexistent buffer to glReadBuffer.

mode is initially GL_FRONT in single-buffered configurations, and GL_BACK in double-buffered configurations.

Errors

- GL_INVALID_ENUM is generated if *mode* is not one of the twelve (or more) accepted values.
- GL_INVALID_OPERATION is generated if *mode* specifies a buffer that does not exist.
- GL_INVALID_OPERATION is generated if glReadBuffer is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_READ_BUFFER

See Also

glCopyPixels,
glCopyTexImage1D,
glCopyTexImage2D,
glCopyTexSubImage1D,
glCopyTexSubImage2D,
glDrawBuffer,
glReadPixels

glReadPixels

`glReadPixels`: read a block of pixels from the frame buffer.

C Specification

```
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 one correspond to a single pixel.
format	Specifies the format of the pixel data. The following symbolic values are accepted: GL_COLOR_INDEX, GL_STENCIL_INDEX, GL_DEPTH_COMPONENT, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_LUMINANCE, and GL_LUMINANCE_ALPHA.
type	Specifies the data type of the pixel data. Must be one of GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, or GL_FLOAT.
pixels	Returns the pixel data.

Description

glReadPixels returns pixel data from the frame buffer, starting with the pixel whose lower left corner is at location (*x, y*), into client memory starting at location pixels. Several parameters control the processing of the pixel data before it is placed into client memory. These parameters are set with three commands: glPixelStore, glPixelTransfer, and glPixelMap. This reference page describes the effects on glReadPixels of most, but not all of the parameters specified by these three commands.

glReadPixels returns values from each pixel with lower left corner at $(x + i, y + j)$ for $0 \geq i < width$ and $0 \geq j < height$. 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.

format specifies the format for the returned pixel values; accepted values are:

GL_COLOR_INDEX

Color indices are read from the color buffer selected by glReadBuffer. Each index is converted to fixed point, shifted left or right depending on the value and sign of GL_INDEX_SHIFT, and added to GL_INDEX_OFFSET. If GL_MAP_COLOR is GL_TRUE, indices are replaced by their mappings in the table GL_PIXEL_MAP_I_TO_I. GL_STENCIL_INDEX

Stencil values are read from the stencil buffer.

Each index is converted to fixed point, shifted left or right depending on the value and sign of GL_INDEX_SHIFT, and added to GL_INDEX_OFFSET. If GL_MAP_STENCIL is GL_TRUE, indices are replaced by their mappings in the table GL_PIXEL_MAP_S_TO_S.

GL_DEPTH_COMPONENT

Depth values are read from the depth buffer. Each component is converted to floating point such that the minimum depth value maps to 0 and the maximum value maps to 1. Each component is then multiplied by GL_DEPTH_SCALE, added to GL_DEPTH_BIAS, and finally clamped to the range [0,1].

GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_LUMINANCE, and GL_LUMINANCE_ALPHA

Processing differs depending on whether color buffers store color indices or RGBA color components. If color indices are stored, they are read from the color buffer selected by glReadBuffer. Each index is converted to fixed point, shifted left or right depending on the value and sign of GL_INDEX_SHIFT, and added to GL_INDEX_OFFSET. Indices are then replaced by the red, green, blue, and alpha values obtained by indexing the tables GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, Each table must be of size 2^n, but n may be different for different tables.

Before an index is used to look up a value in a table of size 2^n, it must be masked against 2^n - 1.

If RGBA color components are stored in the color buffers, they are read from the color buffer selected by glReadBuffer. Each color component is converted to floating point such that zero intensity maps to 0.0 and full intensity maps to 1.0. Each component is then multiplied by GL_*c*_SCALE and added to GL_*c*_BIAS, where c is RED, GREEN, BLUE, or ALPHA. Finally, if GL_MAP_COLOR is GL_TRUE, each component is clamped to the range [0, 1], scaled to the size of its corresponding table, and is then replaced by its mapping in the table GL_PIXEL_MAP_*c*_TO_*c*, where *c* is R, G, B, or A.

Unneeded data is then 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. The final values are clamped to the range [0, 1].

The shift, scale, bias, and lookup factors just described are all specified by glPixelTransfer. The lookup table contents themselves are specified by glPixelMap.

Finally, the indices or components are converted to the proper format, as specified by type. If *format* is GL_COLOR_INDEX or GL_STENCIL_INDEX and *type* is not GL_FLOAT, each index is masked with the mask value given in the following table. If type is GL_FLOAT, then each integer index is converted to single-precision floating-point format.

If format is GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_LUMINANCE, or GL_LUMINANCE_ALPHA and type is not GL_FLOAT, each component is multiplied by the multiplier shown in the following table. If type is GL_FLOAT, then each component is passed as is (or converted to the client's single-precision floating-point format if it is different from the one used by the GL).

Type	**Index Mask**	**Component Conversion**
GL_UNSIGNED_BYTE	$2^8 - 1$	$(2^8 - 1)c$
GL_BYTE	$2^7 - 1$	$[(2^8 - 1)c - 1] / 2$
GL_BITMAP	1	1
GL_UNSIGNED_SHORT	$2^{16} - 1$	$(2^{16}- 1)c$
GL_SHORT	$2^{15} - 1$	$[(2^{16} - 1)c - 1] / 2$
GL_UNSIGNED_INT	$2^{32} - 1$	$(2^{32} - 1)c$
GL_INT	$2^{31} - 1$	$[(2^{32} - 1)c - 1] / 2$
GL_FLOAT	none	c

Return values are placed in memory as follows. If *format* is GL_COLOR_INDEX, GL_STENCIL_INDEX, GL_DEPTH_COMPONENT, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, or GL_LUMINANCE, a single value is returned and the data for the *i*th pixel in the *j*th row is placed in location $j \cdot width + i$. GL_RGB returns three values, GL_RGBA returns four values, and GL_LUMINANCE_ALPHA returns two values for each pixel, with all values corresponding to a single pixel occupying contiguous space in *pixels*. Storage parameters set by glPixelStore, such as GL_PACK_LSB_FIRST and GL_PACK_SWAP_BYTES, affect the way that data is written into memory. See glPixelStore 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*.

Errors

- GL_INVALID_ENUM is generated if *format* or *type* is not an accepted value.
- GL_INVALID_VALUE is generated if either *width* or *height* is negative.
- GL_INVALID_OPERATION is generated if *format* is GL_COLOR_INDEX and the color buffers store RGBA color components.
- GL_INVALID_OPERATION is generated if *format* is GL_STENCIL_INDEX and there is no stencil buffer.
- GL_INVALID_OPERATION is generated if *format* is GL_DEPTH_COMPONENT and there is no depth buffer.

- GL_INVALID_OPERATION is generated if glReadPixels is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_INDEX_MODE

See Also

glCopyPixels,
glDrawPixels,
glPixelMap,
glPixelStore,
glPixelTransfer,
glReadBuffer

glRect

glRectd, glRectf, glRecti, glRects, glRectdv, glRectfv, glRectiv, glRectsv: draw a rectangle.

C Specification

```
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,
    GLshortx2,
    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

glRect 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 exactly equivalent to the following sequence:

```
glBegin(GL_POLYGON);
glVertex2(x1, y1);
glVertex2(x2, y1);
glVertex2(x2, y2);
glVertex2(x1, y2);
glEnd();
```

Note that if the second vertex is above and to the right of the first vertex, the rectangle is constructed with a counter-clockwise winding.

Errors

- GL_INVALID_OPERATION is generated if glRect is executed between the execution of glBegin and the corresponding execution of glEnd.

See Also

glBegin,
glVertex

glRenderMode

`glRenderMode`: set rasterization mode.

C Specification

```
GLint glRenderMode(
GLenum mode)
```

Parameters

mode	Specifies the rasterization mode. Three values are accepted: GL_RENDER, GL_SELECT, and GL_FEEDBACK. The initial value is GL_RENDER.

Description

glRenderMode sets the rasterization mode. It takes one argument, *mode*, which can assume one of three 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 had been GL_RENDER is returned in a select buffer, which must be created (see glSelectBuffer) before selection mode is entered.

GL_FEEDBACK

Feedback mode. No pixel fragments are produced, and no change to the frame buffer contents is made. Instead, the coordinates and attributes of vertices that would have been drawn if the render mode had been GL_RENDER is returned in a feedback buffer, which must be created (see glFeedbackBuffer) before feedback mode is entered.

The return value of glRenderMode is determined by the render mode at the time glRenderMode is called, rather than by *mode*. The values returned for the three render modes are as follows:

GL_RENDER

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.

See the glSelectBuffer and glFeedbackBuffer reference pages for more details concerning selection and feedback operation.

Notes

If an error is generated, glRenderMode returns 0 regardless of the current render mode.

Errors

- GL_INVALID_ENUM is generated if mode is not one of the three accepted values.
- GL_INVALID_OPERATION is generated if glSelectBuffer is called while the render mode is GL_SELECT, or if glRenderMode is called with argument GL_SELECT before glSelectBuffer is called at least once.
- GL_INVALID_OPERATION is generated if glFeedbackBuffer is called while the render mode is GL_FEEDBACK, or if glRenderMode is called with argument GL_FEEDBACK before glFeedbackBuffer is called at least once.
- GL_INVALID_OPERATION is generated if glRenderMode is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_RENDER_MODE

See Also

glFeedbackBuffer,
glInitNames,
glLoadName,
glPassThrough,
glPushName,
glSelectBuffer

glRotate

`glRotated, glRotatef`: multiply the current matrix by a rotation matrix.

C Specification

```
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

glRotate produces a rotation of angle degrees around the vector (*x, y, z*). The current matrix (see glMatrixMode) is multiplied by a rotation matrix with the product replacing the current matrix, as if glMultMatrix were called with the following matrix as its argument:

$$\begin{matrix} xx(1-c)+c & xy(1-c)-zs & xz(1-c)+ys & 0 \\ yx(1-c)+zs & yy(1-c)+c & yz(1-c)\ xs & 0 \\ zx(1-c)-ys & zy(1-c)+xs & zz(1-c)+c & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

Where $c = \cos(angle)$, $s = \sin(angle)$, and $||(x, y, z)|| = 1$ (if not, the GL will normalize this vector).

If the matrix mode is either GL_MODELVIEW or GL_PROJECTION, all objects drawn after glRotate is called are rotated. Use glPushMatrix and glPopMatrix to save and restore the unrotated coordinate system.

Notes

This rotation follows the right-hand rule, so if the vector (*x, y, z*) points toward the user, the rotation will be counterclockwise.

Errors

- GL_INVALID_OPERATION is generated if glRotate is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_MATRIX_MODE
glGet with argument GL_MODELVIEW_MATRIX
glGet with argument GL_PROJECTION_MATRIX
glGet with argument GL_TEXTURE_MATRIX

See Also

glMatrixMode,
glMultMatrix,
glPushMatrix,
glScale,
glTranslate

17 S

glScale

`glScaled`, `glScalef`: multiply the current matrix by a general scaling matrix.

C Specification

```
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

glScale produces a nonuniform scaling along the *x*, *y*, and *z* axes. The three parameters indicate the desired scale factor along each of the three axes.

The current matrix (see glMatrixMode) is multiplied by this scale matrix, and the product replaces the current matrix as if glScale were called with the following matrix as its argument:

$$\begin{matrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

If the matrix mode is either GL_MODELVIEW or GL_PROJECTION, all objects drawn after glScale is called are scaled.

Use glPushMatrix and glPopMatrix to save and restore the unscaled coordinate system.

Notes

If scale factors other than 1 are applied to the modelview matrix and lighting is enabled, lighting often appears wrong. In that case, enable automatic normalization of normals by calling glEnable with the argument GL_NORMALIZE.

Errors

- GL_INVALID_OPERATION is generated if glScale is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_MATRIX_MODE
glGet with argument GL_MODELVIEW_MATRIX
glGet with argument GL_PROJECTION_MATRIX
glGet with argument GL_TEXTURE_MATRIX

See Also

glMatrixMode,
glMultMatrix,
glPushMatrix,
glRotate,
glTranslate

gluScaleImage

`gluScaleImage`: scale an image to an arbitrary size.

C Specification

```
GLint gluScaleImage(
    GLenum format,
    GLsizei wIn,
    GLsizei hIn,
    GLenum typeIn,
const void *dataIn,
    GLsizei wOut,
    GLsizei hOut,
    GLenum typeOut,
    GLvoid* dataOut)
```

Parameters

format	Specifies the format of the pixel data. The following symbolic values are valid: GL_COLOR_INDEX, GL_STENCIL_INDEX, GL_DEPTH_COMPONENT, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_LUMINANCE, and GL_LUMINANCE_ALPHA.
wIn, hIn	Specify the width and height, respectively, of the source image that is scaled.
typeIn	Specifies the data type for *dataIn*. Must be one of GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, or GL_FLOAT.
dataIn	Specifies a pointer to the source image.
wOut, hOut	Specify the width and height, respectively, of the destination image.
typeOut	Specifies the data type for *dataOut*. Must be one of GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, or GL_FLOAT.
dataOut	Specifies a pointer to the destination image.

Description

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

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

A return value of 0 indicates success. Otherwise gluScaleImage returns a GLU error code that indicates what the problem is (see gluErrorString).

See the glReadPixels reference page for a description of the acceptable values for *format, typeIn,* and *typeOut.*

Errors

- GLU_INVALID_VALUE is returned if wIn, hIn, wOut, r hOut is negative.
- GLU_INVALID_ENUM is returned if format, typeIn, or typeOut is not one of the accepted values.

See Also

glDrawPixels,
glReadPixels,
gluBuild1DMipmaps,
gluBuild2DMipmaps,
gluErrorString

glScissor

`glScissor`: define the scissor box.

C Specification

```
void glScissor(
    GLint x,
    GLint y,
    GLsizei width,
    GLsizei height)
```

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

glScissor 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. *width* and *height* specify the width and height of the box.

To enable and disable the scissor test, call glEnable and glDisable with argument GL_SCISSOR_TEST. The test is initially disabled. While the 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. glScissor(0, 0, 1, 1) allows modification of only the lower left pixel in the window, and glScissor(0, 0, 0, 0) doesn't allow modification of any pixels in the window.

When the scissor test is disabled, it is as though the scissor box includes the entire window.

Errors

- GL_INVALID_VALUE is generated if either *width* or *height* is negative.
- GL_INVALID_OPERATION is generated if glScissor is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_SCISSOR_BOX
glIsEnabled with argument GL_SCISSOR_TEST

See Also

glEnable,
glViewport

glSelectBuffer

`glSelectBuffer`: establish a buffer for selection mode values.

C Specification

```
void glSelectBuffer(
    GLsizei size,
    GLuint *buffer)
```

Parameters

size	Specifies the size of *buffer*.
buffer	Returns the selection data.

Description

glSelectBuffer has two arguments: *buffer* is a pointer to an array of unsigned integers, and *size* indicates the size of the array. *buffer* returns values from the name stack (see glInitNames, glLoadName, glPushName) when the rendering mode is GL_SELECT (see glRenderMode). glSelectBuffer must be issued before selection mode is enabled, and it must not be issued while the rendering mode is GL_SELECT.

A programmer can use selection 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 or a raster position 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 glRenderMode 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.

Depth values (which are in the range [0, 1]) are multiplied by $2^{32} - 1$ before being placed in the hit record.

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.

To exit selection mode, call 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* is undefined until glRenderModeis called with an argument other than GL_SELECT.

glBegin/glEnd primitives and calls to glRasterPos can result in hits.

Errors

- GL_INVALID_VALUE is generated if *size* is negative.
- GL_INVALID_OPERATION is generated if glSelectBuffer is called while the render mode is GL_SELECT, or if glRenderMode is called with argument GL_SELECT before glSelectBuffer is called at least once.
- GL_INVALID_OPERATION is generated if glSelectBuffer is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_NAME_STACK_DEPTH

See Also

glFeedbackBuffer,
glInitNames,
glLoadName,
glPushName,
glRenderMode

glShadeModel

`glShadeModel`: select flat or smooth shading.

C Specification

```
void glShadeModel(
    GLenum mode)
```

Parameters

mode	Specifies a symbolic value representing a shading technique. Accepted values are GL_FLAT and GL_SMOOTH. The initial value 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. Starting when glBegin is issued and counting vertices and primitives from 1, the GL gives each flat-shaded line segment *i* the computed color of vertex *i* + 1, its second vertex. Counting similarly from 1, the GL gives each flat-shaded polygon the computed color of the vertex listed in the following table. This is the last vertex to specify the polygon in all cases except single polygons, where the first vertex specifies the flat-shaded color.

Primitive Type of Polygon *i*	**Vertex**
Single polygon ($i \equiv 1$)	1
Triangle strip	*i* + 2
Triangle fan	*i* + 2
Independent triangle	3*i*
Quad strip	2*i* + 2
Independent quad	4*i*

Flat and smooth shading are specified by glShadeModel with *mode* set to GL_FLAT and GL_SMOOTH, respectively.

Errors

- GL_INVALID_ENUM is generated if *mode* is any value other than GL_FLAT or GL_SMOOTH.
- GL_INVALID_OPERATION is generated if glShadeModel is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_SHADE_MODEL

See Also

glBegin,
glColor,
glLight,
glLightModel

gluSphere

`gluSphere`: draw a sphere.

C Specification

```
void gluSphere(
    GLUquadric* quad,
    GLdouble radius,
    GLint slices,
    GLint stacks)
```

Parameters

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

Description

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

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

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

See Also

gluCylinder,
gluDisk,
gluNewQuadric,
gluPartialDisk,
gluQuadricOrientation,
gluQuadricTexture

glStencilFunc

`glStencilFunc`: set function and reference value for stencil testing.

C Specification

```
void glStencilFunc(
    GLenum func,
    GLint ref,
    GLuint mask)
```

Parameters

func	Specifies the test function. Eight tokens are valid: GL_NEVER, GL_LESS, GL_LEQUAL, GL_GREATER, GL_GEQUAL, GL_EQUAL, GL_NOTEQUAL, and GL_ALWAYS. The initial value is GL_ALWAYS.
ref	Specifies the reference value for the stencil test. *ref* is clamped to the range [0, 2^n - 1], where *n* is the number of bitplanes in the stencil buffer. The initial value is 0.
mask	Specifies a mask that is ANDed with both the reference value and the stored stencil value when the test is done. The initial value is all 1s.

Description

Stenciling, like depth-buffering, enables and disables drawing on a per-pixel basis. You draw into the stencil planes using GL drawing primitives, then render geometry and images, using the stencil planes to mask out portions of the screen. Stenciling is typically used in multipass rendering algorithms to achieve special effects, such as decals, outlining, and constructive solid geometry rendering.

The stencil test conditionally eliminates a pixel based on the outcome of a comparison between the reference value and the value in the stencil buffer. To enable and disable the test, call glEnable and glDisable with argument GL_STENCIL_TEST. To specify actions based on the outcome of the stencil test, call glStencilOp.

func is a symbolic constant that determines the stencil comparison function. It accepts one of eight values, shown in the following list. *ref* is an integer reference value that is used in the stencil comparison. It is clamped to the range [0, 2^n - 1], where *n* is the number of bitplanes in the stencil buffer. *mask* is bitwise anded with both the reference value and the stored stencil value, with the *and*ed values participating in the comparison.

If *stencil* represents the value stored in the corresponding stencil buffer location, the following list shows the effect of each comparison function that can be specified by *func*. Only if the comparison succeeds is the pixel passed through to the next stage in the rasterization process (see glStencilOp). All tests treat *stencil* values as unsigned integers in the range [0, 2^n - 1], where *n* is the number of bitplanes in the stencil buffer.

The following values are accepted by *func:*

GL_NEVER
Always fails.

GL_LESS
Passes if (*ref* & *mask*) < (*stencil* & *mask*).

GL_LEQUAL
Passes if (*ref* & *mask*) ≤(*stencil* & *mask*).

GL_GREATER
Passes if (*ref* & *mask*) > (*stencil* & *mask*).

GL_GEQUAL
Passes if (*ref* & *mask*) ≥ (*stencil* & *mask*).

GL_EQUAL
Passes if (*ref* & *mask*) = (*stencil* & *mask*).

GL_NOTEQUAL
Passes if (*ref* & *mask*) ≠ (*stencil* & *mask*).

GL_ALWAYS
Always passes.

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 is generated if *func* is not one of the eight accepted values.
- GL_INVALID_OPERATION is generated if glStencilFunc is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

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

See Also

glAlphaFunc,
glBlendFunc,
glDepthFunc,
glEnable,
glIsEnabled,
glLogicOp,
glStencilOp

glStencilMask

`glStencilMask`: control the writing of individual bits in the stencil planes.

C Specification

```
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

glStencilMask controls the writing of individual bits in the stencil planes. The least significant *n* bits of *mask*, where *n* is the number of bits in the stencil buffer, specify a mask. Where a 1 appears in the mask, it's possible to write to the corresponding bit in the stencil buffer. Where a 0 appears, the corresponding bit is write-protected. Initially, all bits are enabled for writing.

Errors

- GL_INVALID_OPERATION is generated if glStencilMask is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_STENCIL_WRITEMASK
glGet with argument GL_STENCIL_BITS

See Also

glColorMask,
glDepthMask,
glIndexMask,
glStencilFunc,
glStencilOp

glStencilOp

`glStencilOp`: set stencil test actions.

C Specification

```
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, and GL_INVERT. The initial value is GL_KEEP.
zfail	Specifies the stencil action when the stencil test passes, but the depth test fails. *zfail* accepts the same symbolic constants as *fail*. The initial value is GL_KEEP.
zpass	Specifies the 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*. The initial value is GL_KEEP.

Description

Stenciling, like depth-buffering, enables and disables drawing on a per-pixel basis. You draw into the stencil planes using GL drawing primitives, then render geometry and images, using the stencil planes to mask out portions of the screen. Stenciling is typically used in multipass rendering algorithms to achieve special effects, such as decals, outlining, and constructive solid geometry rendering.

The stencil test conditionally eliminates a pixel based on the outcome of a comparison between the value in the stencil buffer and a reference value. To enable and disable the test, call glEnable and glDisable with argument GL_STENCIL_TEST; to control it, call glStencilFunc.

glStencilOp takes three arguments that indicate what happens to the stored stencil value while stenciling is enabled. If the stencil test fails, no change is made to the pixel's color or depth buffers, and *fail* specifies what happens to the stencil buffer contents. The following six actions are possible.

GL_KEEP
Keeps the current value.

GL_ZERO
Sets the stencil buffer value to 0.

GL_REPLACE
Sets the stencil buffer value to *ref*, as specified by glStencilFunc.

GL_INCR
Increments the current stencil buffer value. Clamps to the maximum representable unsigned value.

GL_DECR
Decrements the current stencil buffer value. Clamps to 0.

GL_INVERT
Bitwise inverts the current stencil buffer value.

Stencil buffer values are treated as unsigned integers. When incremented and decremented, values are clamped to 0 and $2^n - 1$, where *n* is the value returned by querying GL_STENCIL_BITS.

The other two arguments to glStencilOp specify stencil buffer actions that depend on whether subsequent depth buffer tests succeed (*zpass*) or fail (*zfail*) (see glDepthFunc). The actions are specified using the same six symbolic constants as *fail*. Note that *zfail* is ignored when there is no depth buffer, or when the depth buffer is not enabled. In these cases, *fail* and *zpass* specify stencil action when the stencil test fails and passes, respectively.

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 glStencilOp.

Errors

- GL_INVALID_ENUM is generated if *fail*, *zfail*, or *zpass* is any value other than the six defined constant values.
- GL_INVALID_OPERATION is generated if glStencilOp is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

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

See Also

glAlphaFunc,
glBlendFunc,
glDepthFunc,
glEnable,
glLogicOp,
glStencilFunc

glXSwapBuffers

`glXSwapBuffers`: make back buffer visible.

C Specification

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

Parameters

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

Description

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

glXSwapBuffers performs an implicit glFlush before it returns. Subsequent GL commands can be issued immediately after calling glXSwapBuffers, but are not executed until the buffer exchange is completed.

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

Notes

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

The X double buffer extension (DBE) has information on which buffer is the currently displayed buffer. This information is shared with GLX.

Errors

- GLXBadDrawable is generated if *drawable* is not a valid GLX drawable.
- GLXBadCurrentWindow is generated if *dpy* and *drawable* are respectively the display and drawable associated with the current context of the calling thread, and *drawable* identifies a window that is no longer valid.

See Also

glFlush

18 T

gluTessBeginContour

`gluTessBeginContour, gluTessEndContour`: delimit a contour description.

C Specification

```
void gluTessBeginContour(
    GLUtesselator* tess)
void gluTessEndContour(
    GLUtesselator* tess)
```

Parameters

tess	Specifies the tessellation object (created with gluNewTess).

Description

gluTessBeginContour and gluTessEndContour 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.

See Also

gluNewTess,
gluTessBeginPolygon,
gluTessVertex,
gluTessCallback,
gluTessProperty,
gluTessNormal,
gluTessEndPolygon

gluTessBeginPolygon

`gluTessBeginPolygon`: delimit a polygon description.

C Specification

```
void gluTessBeginPolygon(
    GLUtesselator* tess,
    GLvoid* data)
```

Parameters

tess	Specifies the tessellation object (created with gluNewTess).
data	Specifies a pointer to user polygon data.

Description

gluTessBeginPolygon and gluTessEndPolygon delimit the definition of a non-convex 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.

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.

See Also

gluNewTess,
gluTessBeginContour,
gluTessVertex,
gluTessCallback,
gluTessProperty,
gluTessNormal gluTessEndPolygon

gluTessCallback

`gluTessCallback`: define a callback for a tessellation object.

C Specification

```
void gluTessCallback(
    GLUtesselator* tess,
    GLenum which,
    GLvoid (*CallBackFunc)()
```

Parameters

tess	Specifies the tessellation object (created with gluNewTess).
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, and GLU_TESS_ERROR_DATA.
CallBackFunc	Specifies the function to be called.

Description

gluTessCallback is used to indicate a callback to be used by a tessellation object. If the specified callback is already defined, then it is replaced. If *CallBackFunc* is NULL, then 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 that 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. Note that the *polygon_data* parameter used by some of the functions is a copy of the pointer that was specified when gluTessBeginPolygon was called. The legal 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

The edge flag callback is similar to glEdgeFlag. The function takes a single boolean flag that indicates which edges lie on the polygon boundary. If the flag is GL_TRUE, then each vertex that follows begins an edge that lies on the polygon boundary, that is, an edge that separates an interior region from an exterior one. If the flag is GL_FALSE, then each vertex that follows begins an edge that lies in the polygon interior. The edge flag callback (if defined) is invoked before the first vertex callback.

Since triangle fans and triangle strips do not support edge flags, the begin callback is not called with GL_TRIANGLE_FAN or GL_TRIANGLE_STRIP if a non-NULL edge flag callback is provided. (If the callback is initialized to NULL, there is no impact on performance). Instead, the fans and strips are converted to 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

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

The end callback serves the same purpose as glEnd. It indicates the end of a primitive and 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. (This is the only error that can occur during tessellation and rendering.)

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

The error callback is called when an error is encountered. The one argument is of type GLenum; it indicates the specific error that occurred and will be set to one of 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, GLU_TESS_NEED_COMBINE_CALLBACK.

Character strings describing these errors can be retrieved with the gluErrorString call. The function prototype for this callback is:

```
void error(GLenum errno);
```

The GLU library will recover from the first four errors by inserting the missing call(s). 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_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);
```

See Also

glBegin,
glEdgeFlag,
glVertex,
gluNewTess,
gluErrorString,
gluTessVertex,
gluTessBeginPolygon,
gluTessBeginContour,
gluTessProperty,
gluTessNormal

gluTessEndPolygon

`gluTessEndPolygon`: delimit a polygon description.

C Specification

```
void gluTessEndPolygon(
    GLUtesselator* tess)
```

Parameters

tess	Specifies the tessellation object (created with gluNewTess).

Description

gluTessBeginPolygon and gluTessEndPolygon delimit the definition of a nonconvex 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.

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

See Also

gluNewTess,
gluTessBeginContour,
gluTessVertex,
gluTessCallback,
gluTessProperty,
gluTessNormal,
gluTessBeginPolygon

gluTessNormal

`gluTessNormal`: specify a normal for a polygon.

C Specification

```
void gluTessNormal(
    GLUtesselator* tess,
    GLdouble valueX,
    GLdouble valueY,
    GLdouble valueZ)
```

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.

Description

gluTessNormal 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.

See Also

gluTessBeginPolygon,
gluTessEndPolygon

gluTessProperty

`gluTessProperty`: set a tessellation object property.

C Specification

```
void gluTessProperty(
    GLUtesselator* tess,
    GLenum which,
    GLdouble data)
```

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.

Description

gluTessProperty 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, or GLU_TESS_WINDING_NEGATIVE, or 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.

See Also

gluGetTessProperty

gluTessVertex

`gluTessVertex`: specify a vertex on a polygon.

C Specification

```
void gluTessVertex(
    GLUtesselator* tess,
    GLdouble *location,
    GLvoid* data)
```

Parameters

tess	Specifies the tessellation object (created with gluNewTess).
location	Specifies the location of the vertex.
data	Specifies an opaque pointer passed back to the program with the vertex callback (as specified by gluTessCallback).

Description

gluTessVertex describes a vertex on a polygon that the program defines. Successive gluTessVertex calls describe a closed contour. For example, to describe a quadrilateral gluTessVertex should be called four times. gluTessVertex can only be called between gluTessBeginContour and gluTessEndContour.

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

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 de-referenced 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*).

See Also

gluTessBeginPolygon,
gluNewTess,
gluTessBeginContour,
gluTessCallback,
gluTessProperty,
gluTessNormal,
gluTessEndPolygon

glTexCoord

`glTexCoord1d, glTexCoord1f, glTexCoord1i, glTexCoord1s, glTexCoord2d, glTexCoord2f, glTexCoord2i, glTexCoord2s, glTexCoord3d, glTexCoord3f, glTexCoord3i, glTexCoord3s, glTexCoord4d, glTexCoord4f, glTexCoord4i, glTexCoord4s, glTexCoord1dv, glTexCoord1fv, glTexCoord1iv, glTexCoord1sv, glTexCoord2dv, glTexCoord2fv, glTexCoord2iv, glTexCoord2sv, glTexCoord3dv, glTexCoord3fv, glTexCoord3iv, glTexCoord3sv, glTexCoord4dv, glTexCoord4fv, glTexCoord4iv, glTexCoord4sv`: set the current texture coordinates.

C Specification

```
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 *q* 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

glTexCoord specifies texture coordinates in one, two, three, or four dimensions. glTexCoord1 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, glTexCoord can be called between a call to glBegin and the corresponding call to glEnd.

Associated Gets

glGet with argument GL_CURRENT_TEXTURE_COORDS

See Also

glTexCoordPointer,
glVertex

glTexCoordPointer

`glTexCoordPointer`: define an array of texture coordinates.

C Specification

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

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.

Description

glTexCoordPointer 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* specifies 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; 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 to construct a sequence of primitives (all of the same type) from pre-specified vertex and vertex attribute arrays. Use glArrayElement to specify primitives by indexing vertexes and vertex attributes and glDrawElements to construct a sequence of primitives by indexing vertexes and vertex attributes.

Notes

glTexCoordPointer 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 the execution of glBegin and the corresponding execution of glEnd, but an error may or may not be generated. If no error is generated, the operation is undefined.

glTexCoordPointer is typically implemented on the client side with no protocol.

The texture coordinate array parameters are client-side state and are therefore not saved or restored by glPushAttrib and glPopAttrib. Use glPushClientAttrib and glPopClientAttrib instead.

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

See Also

glArrayElement,
glColorPointer,
glDrawArrays,
glDrawElements,
glEdgeFlagPointer,
glEnable,
glGetPointerv,
glIndexPointer,
glNormalPointer,
glPopClientAttrib,
glPushClientAttrib,
glTexCoord,
glVertexPointer

glTexEnv

`glTexEnvf, glTexEnvi, glTexEnvfv, glTexEnviv`: set texture environment parameters.

C Specification

```
void glTexEnvf(
    GLenum target,
    GLenum pname,
    GLfloat param)
void glTexEnvi(
    GLenum target,
    GLenum pname,
    GLint param)
void glTexEnvfv(
    GLenum target,
    GLenum pname,
    const GLfloat *params)
void glTexEnviv(
    GLenum target,
    GLenum pname,
    const GLint   *params)
```

Parameters

target	Specifies a texture environment. Must be GL_TEXTURE_ENV.
pname (glTexEnvf and glTexEnvi)	Specifies the symbolic name of a single-valued texture environment parameter. Must be GL_TEXTURE_ENV_MODE.
pname (for glTexEnvfv and glTexEnviv)	Specifies the symbolic name of a texture environment parameter. Accepted values are GL_TEXTURE_ENV_MODE and GL_TEXTURE_ENV_COLOR and GL_TEXTURE_LIGHTING_MODE_hp.
param	Specifies a single symbolic constant, one of GL_MODULATE, GL_DECAL, GL_BLEND, or GL_REPLACE. If *pname* is GL_TEXTURE_LIGHTING_MODE_hp, specifies a single symbolic constant, either GL_TEXTURE_POST_SPECULAR_hp or GL_TEXTURE_PRE_SPECULAR_hp.
params	Specifies a pointer to a parameter array that contains either a single symbolic constant or an RGBA color.

Description

A texture environment specifies how texture values are interpreted when a fragment is textured. *target* must be GL_TEXTURE_ENV. *pname* can be either GL_TEXTURE_ENV_MODE, GL_TEXTURE_ENV_COLOR or GL_TEXTURE_LIGHTING_MODE_hp (if the extension GL_hp_texture_lighting is supported).

If *pname* is GL_TEXTURE_ENV_MODE, then *params* is (or points to) the symbolic name of a texture function. Four texture functions may be specified: GL_MODULATE, GL_DECAL, GL_BLEND, and GL_REPLACE. If *pname* is GL_TEXTURE_LIGHTING_MODE_hp, two possible values for *param* may be specified: either GL_TEXTURE_PRE_SPECULAR_hp or GL_TEXTURE_POST_SPECULAR_hp.

A texture function acts on the fragment to be textured using the texture image value that applies to the fragment (see glTexParameter) and produces an RGBA color for that fragment. The following table shows how the RGBA color is produced for each of the three texture functions that can be chosen. *C* is a triple of color values (RGB) and *A* is the associated alpha value. RGBA values extracted from a texture image are in the range [0, 1]. The subscript f refers to the incoming fragment, the subscript *t* to the texture image, the subscript *c* to the texture environment color, and subscript *v* indicates a value produced by the texture function.

A texture image can have up to four components per texture element (see glTexImage1D, glTexImage2D, glCopyTexImage1D, and glCopyTexImage2D). 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.

Base Internal Format	**Texture Functions**			
	GL_MODULATE	**GL_DECAL**	**GL_BLEND**	**GL_REPLACE**
GL_ALPHA	$C_v = C_f$ $A_v = A_f A_t$	(undefined)	$C_v = C_f$ $A_v = A_f$	$C_v = C_f$ $A_v = A_t$
GL_LUMINANCE 1	$C_v = L_t C_f$ $A_v = A_f$	(undefined)	$C_v = (1 - L_t)$ $C_f + L_t C_c$ $A_v = A_f$	$C_v = L_t$ $A_v = A_f$
GL_LUMINANCE_ALPHA 2	$C_v = L_t C_f$ $A_v = A_t A_f$	(undefined)	$C_v = (1 - L_t)$ $C_f + L_t C_c$ $A_v = A_t A_f$	$C_v = L_t$ $A_v = A_t$
GL_INTENSITY	$C_v = C_f I_t$ $A_v = A_f I_t$	(undefined)	$C_v = (1 - I_t)$ $C_f + I_t C_c$ $A_v = (1 - I_t)$ $A_f + I_t A_c$	$C_v = I_t$ $A_v = I_t$

Base Internal Format	Texture Functions			
	GL_MODULATE	GL_DECAL	GL_BLEND	GL_REPLACE
GL_RGB 3	$C_v = C_t\ C_f$ $A_v = A_f$	$C_v = C_t$ $A_v = A_f$	$C_v = (1 - C_t\ )$ $C_f + C_t\ C_c$ $A_v = A_f$	$C_v = C_t$ $A_v = A_f$
GL_RGBA 4	$C_v = C_t\ C_f$ $A_v = A_t\ A_f$	$C_v = (1 - A_t\ )$ C_f +At C_t $A_v = A_f$	$C_v = (1 - C_t\ )$ $C_f + C_t\ C_c$ $A_v = A_t\ A_f$	$C_v = C_t$ $A_v = A_t$

If *pname* is GL_TEXTURE_ENV_COLOR, *params* is a pointer to an array that holds an RGBA color consisting of four values. Integer color components are interpreted linearly such that the most positive integer maps to 1.0, and the most negative integer maps to -1.0. The values are clamped to the range [0, 1] when they are specified. C_c takes these four values.

GL_TEXTURE_ENV_MODE defaults to GL_MODULATE and GL_TEXTURE_ENV_COLOR defaults to (0, 0, 0, 0). GL_TEXTURE_LIGHTING_hp defaults to GL_TEXTURE_POST_SPECULAR_hp.

Notes

GL_REPLACE may only be used if the GL version is 1.1 or greater.

GL_TEXTURE_LIGHTING_MODE_hp may only be used if the GL_hp_texture_lighting 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.

Errors

- GL_INVALID_ENUM is generated when *target* or *pname* is not one of the accepted defined values, or when *params* should have a defined constant value (based on the value of *pname*) and does not.
- GL_INVALID_OPERATION is generated if glTexEnv is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGetTexEnv

See Also

glCopyPixels,
glCopyTexImage1D,
glCopyTexImage2D,
glCopyTexSubImage1D,

glCopyTexSubImage2D,
glTexImage1D,
glTexImage2D,
glTexParameter,
glTexSubImage1D,
glTexSubImage2D

glTexGen

`glTexGend, glTexGenf, glTexGeni, glTexGendv, glTexGenfv, glTexGeniv`: control the generation of texture coordinates.

C Specification

```
void glTexGend(
    GLenum coord,
    GLenum pname,
    GLdouble param)
void glTexGenf(
    GLenum coord,
    GLenum pname,
    GLfloat param)
void glTexGeni(
    GLenum coord,
    GLenum pname,
    GLint param)
void glTexGendv(
    GLenum coord,
    GLenum pname,
    const GLdouble *params)
void glTexGenfv(
    GLenum coord,
    GLenum pname,
    const GLfloat  *params)
void glTexGeniv(
    GLenum coord,
    GLenum pname,
    const GLint *params)
```

Parameters

coord	Specifies a texture coordinate. Must be one of GL_S, GL_T, GL_R, or GL_Q.
pname	Specifies the symbolic name of the texture-coordinate generation function. Must be GL_TEXTURE_GEN_MODE.
param	Specifies a single-valued texture generation parameter, one of GL_OBJECT_LINEAR, GL_EYE_LINEAR, or GL_SPHERE_MAP.
coord	Specifies a texture coordinate. Must be one of GL_S, GL_T, GL_R, or 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.

params Specifies a pointer to an array of texture generation parameters. If *pname* is GL_TEXTURE_GEN_MODE, then the array must contain a single symbolic constant, one of GL_OBJECT_LINEAR, GL_EYE_LINEAR, or GL_SPHERE_MAP. Otherwise, *params* holds the coefficients for the texture-coordinate generation function specified by *pname*.

Description

glTexGen selects a texture-coordinate generation function or supplies coefficients for one of the functions. coord names one of the (*s*, *t*, *r*, *q*) texture coordinates; it must be one of the symbols GL_S, GL_T, GL_R, or GL_Q. pname must be one of three symbolic constants: GL_TEXTURE_GEN_MODE, GL_OBJECT_PLANE, or GL_EYE_PLANE. If *pname* is GL_TEXTURE_GEN_MODE, then *params* chooses a mode, one of GL_OBJECT_LINEAR, GL_EYE_LINEAR, or GL_SPHERE_MAP. If *pname* is either GL_OBJECT_PLANE or GL_EYE_PLANE, *params* contains coefficients for the corresponding texture generation function.

If the texture generation function is GL_OBJECT_LINEAR, the function

$$g = p_1 x_o + p_2 y_o + p_3 z_o + p_4 w_o$$

is used, where *g* is the value computed for the coordinate named in *coord*, p_1, p_2, p_3, and p_4 are the four values supplied in params, and x_o, y_o, z_o, and w_o are the object coordinates of the vertex. This function can be used, for example, 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 can then be used to index the texture image to map white snow onto peaks and green grass onto foothills.

If the texture generation function is GL_EYE_LINEAR, the function

$$g = p_{1'}\ x_e + p_{2'}\ y_e + p_{3'}\ z_e + p_{4'}\ w_e$$

is used, where

$$(\ p_{1'}\ p_{2'}\ p_{3'}\ p_{4'}\ ) = (\ p_1\ p_2\ p_3\ p_4)\ M^{-1}$$

and x_e, y_e, z_e, and w_e are the eye coordinates of the vertex, p_1, p_2, p_3, and p_4 are the values supplied in *params*, and *M* is the modelview matrix when glTexGen is invoked. If *M* is poorly conditioned or singular, texture coordinates generated by the resulting function may be inaccurate or undefined.

Note that the values in *params* define a reference plane in eye coordinates. The modelview matrix that is applied to them may not be the same one in effect when the polygon vertices are transformed. This function establishes a field of texture coordinates that can produce dynamic contour lines on moving objects.

If *pname* is GL_SPHERE_MAP and *coord* is either GL_S or GL_T, *s* and *t* texture coordinates are generated as follows. Let *u* be the unit vector pointing from the origin to the polygon vertex (in eye coordinates). Let ***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

$$\mathbf{f} = \mathbf{u}\ \ 2\mathbf{n}'\ \mathbf{n}'^T u$$

Finally, let *m* = 2 sqrt($f_x^2 + f_y^2 + (f_z + 1)^2$). Then the values assigned to the *s* and *t* texture coordinates are

$s = f_x/m + 1/2$

$t = f_y/m + 1/2$

To enable or disable a texture-coordinate generation function, call glEnable or glDisable with one of the symbolic texture-coordinate names (GL_TEXTURE_GEN_S, GL_TEXTURE_GEN_T, GL_TEXTURE_GEN_R, or GL_TEXTURE_GEN_Q) as the argument. When enabled, the specified texture coordinate is computed according to the generating function associated with that coordinate. When disabled, subsequent vertices take the specified texture coordinate from the current set of texture coordinates. Initially, all texture generation functions are set to GL_EYE_LINEAR and are disabled. Both *s* plane equations are (1, 0, 0, 0), both *t* plane equations are (0, 1, 0, 0), and all *r* and *q* plane equations are (0, 0, 0, 0).

Errors

- GL_INVALID_ENUM is generated when *coord* or *pname* is not an accepted defined value, or when *pname* is GL_TEXTURE_GEN_MODE and *params* is not an accepted defined value.
- GL_INVALID_ENUM is generated when *pname* is GL_TEXTURE_GEN_MODE, *params* is GL_SPHERE_MAP, and c*oord* is either GL_R or GL_Q.
- GL_INVALID_OPERATION is generated if glTexGen is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

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

See Also

glCopyPixels,
glCopyTexImage2D,
glCopyTexSubImage1D,
glCopyTexSubImage2D,
glTexEnv,
glTexImage1D,
glTexImage2D,
glTexParameter,
glTexSubImage1D,
glTexSubImage2D

glTexImage1D

`glTexImage1D`: specify a one-dimensional texture image.

C Specification

```
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 or 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_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. Additionally, if the extension GL_EXT_shadow is supported, may be one of the symbolic constants GL_DEPTH_COMPONENT, GL_DEPTH_COMPONENT16_EXT, GL_DEPTH_COMPONENT24_EXT, or GL_DEPTH_COMPONENT32_EXT.
width	Specifies the width of the texture image. Must be $2^n + 2 \times$ 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. The following symbolic values are accepted: GL_COLOR_INDEX, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_LUMINANCE, and

	GL_LUMINANCE_ALPHA. If the extension GL_EXT_shadow is supported, the symbolic value GL_DEPTH_COMPONENT is also accepted.
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 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.

format determines the composition of each element in *pixels*. It can assume one of nine 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 for green and blue, and 1 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 for red and blue, and 1 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 for red and green, and 1 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 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 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 then multiplied by the signed scale factor GL_*c*_SCALE, added to the signed bias GL_*c*_BIAS, and clamped to the range [0, 1] (see glPixelTransfer).

GL_LUMINANCE

Each element is a single luminance value. 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 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_DEPTH_COMPONENT

Each element is a single depth component. It is converted to floating-point, then multiplied by the signed scale factor GL_DEPTH_SCALE, added to the signed bias GL_DEPTH_BIAS, and clamped to the range [0, 1] (see glPixelTransfer).

If an application wants to store the texture at a certain resolution or in a certain format, it can 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 preceding 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 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

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. glPixelStore and glPixelTransfer modes affect texture images in exactly the way they affect glDrawPixels.

The *format* value GL_DEPTH_COMPONENT and *internalformat* values GL_DEPTH_COMPONENT16_EXT, GL_DEPTH_COMPONENT24_EXT, and GL_DEPTH_COMPONENT32_EXT may only be used if the GL_EXT_shadow extension is supported.

GL_PROXY_TEXTURE_1D may only be used 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 accommodate a texture of width *width*. You can then download subtextures to initialize the texture memory. The image is undefined if the program 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_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 0.

- GL_INVALID_VALUE may be generated if *level* is greater than $\log_2$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* is less than 0 or greater than 2 + GL_MAX_TEXTURE_SIZE, or if it cannot be represented as $2^n + 2 \times$ 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

See Also

glCopyPixels,
glCopyTexImage1D,
glCopyTexImage2D,
glCopyTexSubImage1D,
glCopyTexSubImage2D,
glDrawPixels,
glPixelStore,
glPixelTransfer,
glTexEnv,
glTexGen,
glTexImage2D,
glTexSubImage1D,
glTexSubImage2D,
glTexParameter

glTexImage2D

`glTexImage2D`: specify a two-dimensional texture image.

C Specification

```
void glTexImage2D(
    GLenum target,
    GLint level,
    GLint internalformat,
    GLsizei width,
    GLsizeiheight,
    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 *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_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. Additionally, if the extension GL_EXT_shadow is supported, may be one of the symbolic constants GL_DEPTH_COMPONENT, GL_DEPTH_COMPONENT16_EXT, GL_DEPTH_COMPONENT24_EXT, or GL_DEPTH_COMPONENT32_EXT.
width	Specifies the width of the texture image. Must be $2^n + 2 \times$ 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 $2^m + 2 \times$ 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. The following symbolic values are accepted: GL_COLOR_INDEX, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_LUMINANCE, and GL_LUMINANCE_ALPHA. If the extension GL_EXT_shadow is supported, the symbolic value GL_DEPTH_COMPONENT is also accepted.
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 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.

format determines the composition of each element in *pixels*. It can assume one of nine 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 for green and blue, and 1 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 for red and blue, and 1 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 for red and green, and 1 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 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 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 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_DEPTH_COMPONENT

Each element is a single depth component. It is converted to floating-point, then multiplied by the signed scale factor GL_DEPTH_SCALE, added to the signed bias GL_DEPTH_BIAS, and clamped to the range [0, 1] (see glPixelTransfer).

Refer to the glDrawPixels reference page for a description of the acceptable values for the *type* parameter.

If an application wants to store the texture at a certain resolution or in a certain format, it can 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 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

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. glPixelStore and glPixelTransfer modes affect texture images in exactly the way they affect glDrawPixels.

The *format* value GL_DEPTH_COMPONENT and *internalformat* values GL_DEPTH_COMPONENT16_EXT, GL_DEPTH_COMPONENT24_EXT, and GL_DEPTH_COMPONENT32_EXT may only be used if the GL_EXT_shadow extension is supported.

glTexImage2D and GL_PROXY_TEXTURE_2D 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 accommodate 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_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 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 0.
- GL_INVALID_VALUE may be generated if *level* is greater than $\log_2$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 0 or greater than 2 + GL_MAX_TEXTURE_SIZE, or if either cannot be represented as $2k + 2 \times$ 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

See Also

glCopyPixels,
glCopyTexImage1D,
glCopyTexImage2D,
glCopyTexSubImage1D,
glCopyTexSubImage2D,
glDrawPixels,
glPixelStore,
glPixelTransfer,
glTexEnv,
glTexGen,
glTexImage1D,
glTexSubImage1D,
glTexSubImage2D,
glTexParameter

glTexImage3DEXT

`glTexImage3DEXT`: Specify a three-dimensional texture image.

C Specification

```
void glTexImage3DEXT(
    GLenum target,
    GLint level,
    GLenum 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.
	Specifies the number of color components in the texture. Must be 1, 2, 3, or 4, or one of the following symbolic constants: 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. Additionally, if the extension GL_EXT_shadow is supported, may be one of the symbolic constants GL_DEPTH_COMPONENT, GL_DEPTH_COMPONENT16_EXT, GL_DEPTH_COMPONENT24_EXT, or GL_DEPTH_COMPONENT32_EXT.
width	Specifies the width of the texture image. Must be $2^n + 2 \times$ border for some integer *n*.
height	Specifies the height of the texture image. Must be $2^m + 2 \times$ border for some integer *m*.

depth	Specifies the depth of the texture image. Must be $2^l + 2 \times$ 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, and GL_LUMINANCE_ALPHA. If the extension GL_EXT_shadow is supported, the symbolic value GL_DEPTH_COMPONENT is also accepted.
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. Three-dimensional texturing is enabled and disabled using glEnable and glDisable with argument GL_TEXTURE_3D_EXT.

Texture images are defined with 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 the internal resolution and format used to store the image. The last three arguments describe the way the image is represented in memory, and 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 will set all of the texture image state to 0 (GL_TEXTURE_WIDTH, GL_TEXTURE_HEIGHT, GL_TEXTURE_BORDER, GL_TEXTURE_COMPONENTS), but no error will be generated.

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.

The first element corresponds to the lower-left-rear corner of the texture volume. Subsequent elements progress left-to-right through the remaining texels in the lowest-rear row of the texture volume, then in successively higher rows of the rear 2D slice of the texture volume, then in successively closer 2D slices of the texture volume. The final element corresponds to the upper-right-front corner of the texture volume.

Each element of pixels is converted to an RGBA element according to

GL_COLOR_INDEX

Each element is a single value, a color index. It is converted 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. It is converted to floating-point and assembled into an RGBA element by attaching 0.0 for green and blue, and 1.0 for alpha.

GL_GREEN

Each element is a single green component. It is converted to floating-point and assembled into an RGBA element by attaching 0.0 for red and blue, and 1.0 for alpha.

GL_BLUE

Each element is a single blue component. It is converted to floating-point and assembled into an RGBA element by attaching 0.0 for red and green, and 1.0 for alpha.

GL_ALPHA

Each element is a single alpha component. It is converted to floating-point and assembled into an RGBA element by attaching 0.0 for red, green, and blue.

GL_RGB

Each element is an RGB triple. It is converted to floating-point and assembled into an RGBA element by attaching 1.0 for alpha (see glPixelTransfer).

GL_RGBA, GL_ABGR_EXT

Each element contains all four components; for GL_RGBA, the red component is first, followed by green, then blue, and then alpha; for GL_ABGR_EXT the order is alpha, blue, green, and then red.

GL_LUMINANCE

Each element is a single luminance value. It is converted to floating-point, then assembled into an RGBA element by replicating the luminance value three times for red, green, and blue and attaching 1.0 for alpha.

GL_LUMINANCE_ALPHA

Each element is a luminance/alpha pair. It is converted to floating-point, then assembled into an RGBA element by replicating the luminance value three times for red, green, and blue.

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

An application may desire that the texture be stored at a certain resolution, or that it be stored in a certain format. This resolution and format can be requested by *internalformat*, but the implementation may not support that resolution (the formats of GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_RGB, and GL_RGBA must be supported). When a resolution and storage format is specified, the implementation will update the texture state to provide the best match to the requested resolution. The GL_PROXY_TEXTURE_3D_EXT target can be used to try a resolution and format. The implementation will compute its best match for the requested storage resolution and format; this state can then be queried using glGetTexLevelParameter.

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

Texturing has no effect in color index mode.

The texture image can be represented by the same data formats and types as the pixels in a glDrawPixels command, except that formats GL_STENCIL_INDEX and GL_DEPTH_COMPONENT cannot be used, and type GL_BITMAP cannot be used. glPixelStore and glPixelTransfer modes affect texture images in exactly the way they affect glDrawPixels.

A texture image with zero height, width, or depth indicates the null texture. If the null texture is specified for level-of-detail 0, it is as if texturing were disabled.

glTexImage3DEXT is part of the EXT_texture3d extension.

Errors

- GL_INVALID_ENUM is generated when *target* is not an accepted value.
- GL_INVALID_ENUM is generated when *format* is not an accepted value.
- GL_INVALID_ENUM is generated when *type* is not an accepted value.
- GL_INVALID_VALUE is generated if *level* is less than zero or greater than $\log_2$max, where *max* is the returned value of GL_MAX_3D_TEXTURE_SIZE_EXT.
- GL_INVALID_VALUE is generated if *internalformat* is not an accepted value.
- GL_INVALID_VALUE is generated if *width*, *height*, or *depth* is less than zero or greater than GL_MAX_3D_TEXTURE_SIZE_EXT, when *width*, *height*, or *depth* cannot be represented as $2^k + 2 \times$ border for some integer *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.
- GL_TEXTURE_TOO_LARGE_EXT is generated if the implementation cannot accommodate a texture of the size requested.

Associated Gets

glGetTexImage
glIsEnabled with argument GL_TEXTURE_3D_EXT

See Also

glDrawPixels,
glFog,
glPixelStore,
glPixelTransfer,
glTexEnv,

glTexGen,
glTexImage1D,
glTexImage2D,
glTexParameter.

glTexParameter

`glTexParameterf`, `glTexParameteri`, `glTexParameterfv`, `glTexParameteriv`: set texture parameters.

C Specification

```
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

target	Specifies the target texture, which must be either GL_TEXTURE_1D, GL_TEXTURE_2D or GL_TEXTURE_3D_EXT.
pname	Specifies the symbolic name of a single-valued texture parameter. *pname* can be one of the following: GL_TEXTURE_MIN_FILTER, GL_TEXTURE_MAG_FILTER, GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T, GL_TEXTURE_PRIORITY, GL_TEXTURE_COMPARE_EXT, or GL_TEXTURE_COMPARE_OPERATOR_EXT.
param	Specifies the value of *pname*.
target	Specifies the target texture, which must be either GL_TEXTURE_1D or GL_TEXTURE_2D.
pname	Specifies the symbolic name of a texture parameter. *pname* can be one of the following: GL_TEXTURE_MIN_FILTER, GL_TEXTURE_MAG_FILTER, GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T, GL_TEXTURE_BORDER_COLOR, or GL_TEXTURE_PRIORITY.
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- or two-dimensional image and a set of parameters that determine how samples are derived from the image.

glTexParameter assigns the value or values in *params* to the texture parameter specified as *pname*. *target* defines the target texture, either GL_TEXTURE_1D or GL_TEXTURE_2D. The following symbols are accepted in *pname*:

GL_TEXTURE_MIN_FILTER

The texture minifying function is used whenever the pixel being textured maps to an area greater than one texture element. There are six defined minifying functions. Two of them use the nearest one or nearest four texture elements to compute the texture value. The other four use mipmaps.

A mipmap is an ordered set of arrays representing the same image at progressively lower resolutions: 2^a for 1D mipmaps, $2^a \times 2^b$ for 2D mipmaps, and $2^a \times 2^b \times 2^c$ for 3D mipmaps.

For example, if a 2D texture has dimensions $2m \times 2n$, there are max(*m*, *n*) + 1 mipmaps. The first mipmap is the original texture, with dimensions $2m \times 2n$. Each subsequent mipmap has dimensions $2^{k-1} \times 2^{l-1}$, where $2k \times 2l$ are the dimensions of the previous mipmap, until either *k*=0 or *l*=0. At that point, subsequent mipmaps have dimension $1 \times 2^{l-1}$ or $2^{k-1} \times 1$ until the final mipmap, which has dimension 1×1. To define the mipmaps, call glTexImage1D, glTexImage2D, glCopyTexImage1D, glCopyTexImage2D, or glCopyTexImage3DEXT with the level argument indicating the order of the mipmaps. Level 0 is the original texture; level max*(m, n)* is the final 1×1 mipmap. *params* 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 initial 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 (see below). 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_TEXTURE_WRAP_S

Sets the wrap parameter for texture coordinate *s* to GL_CLAMP, GL_REPEAT, GL_CLAMP_TO_BORDER_EXT, or GL_CLAMP_TO_EDGE_EXT. GL_CLAMP causes s coordinates to be clamped to the range [0, 1] and is useful for preventing wrapping artifacts when mapping a single image onto an object. GL_REPEAT causes the integer part of the s coordinate to be ignored; the GL uses only the fractional part, thereby creating a repeating pattern. GL_CLAMP_TO_BORDER_EXT causes s coordinates to be clamped to a range 1/2 texel outside [0, 1]; this prevents the “half border, half edge” color artifact. GL_CLAMP_TO_EDGE_EXT causes s coordinates to be clamped to a range 1/2 texel inside [0, 1]; this prevents any border colors from showing up in the image. Border texture elements are accessed only if wrapping is set to GL_CLAMP or GL_CLAMP_TO_BORDER_EXT. Initially, GL_TEXTURE_WRAP_S is set to GL_REPEAT.

GL_TEXTURE_WRAP_T

Sets the wrap parameter for texture coordinate *t* to GL_CLAMP, GL_REPEAT, GL_CLAMP_TO_BORDER_EXT, or GL_CLAMP_TO_EDGE_EXT. See the discussion under GL_TEXTURE_WRAP_S. Initially, GL_TEXTURE_WRAP_T is set to GL_REPEAT.

GL_TEXTURE_WRAP_R_EXT

Sets the wrap parameter for texture coordinate *r* to GL_CLAMP, GL_REPEAT, GL_CLAMP_TO_BORDER_EXT, or GL_CLAMP_TO_EDGE_EXT. See the discussion under GL_TEXTURE_WRAP_S. Initially, GL_TEXTURE_WRAP_R_EXT is set to GL_REPEAT.

GL_TEXTURE_BORDER_COLOR

Sets a border color. *params* contains four values that comprise the RGBA color of the texture border. Integer color components are interpreted linearly such that the most positive integer maps to 1.0, and the most negative integer maps to - 1.0. The values are clamped to the range [0, 1] when they are specified. Initially, the border color is (0, 0, 0, 0).

GL_TEXTURE_PRIORITY

Specifies the texture residence priority of the currently bound texture. Permissible values are in the range [0, 1]. See glPrioritizeTextures and glBindTexture for more information.

GL_GENERATE_MIPMAP_EXT

Specifies whether MIP levels should be automatically filtered when the base level (level 0) of a texture map is modified with glTexImage1D, glTexImage2D, glTexImage3DEXT, glTexSubImage1D. glTexSubImage2D, glTexSubImage3DEXT, glCopyTexImage1D, glCopyTexImage2D, glCopyTexSubImage1D, glCopyTexSubImage2D, or glCopyTexSubImage3DEXT. The default is GL_FALSE (no automatic generation of MIP levels).

GL_TEXTURE_COMPARE_EXT

Specifies whether the depth texture comparison operator is in effect. *param* is either GL_TRUE or GL_FALSE. The default is GL_FALSE, meaning that the depth texture comparison operator is not in effect (see GL_TEXTURE_COMPARE_OPERATOR_EXT, below).

GL_TEXTURE_COMPARE_OPERATOR_EXT

Specifies the comparison operator to be used when the depth texture comparison operator is in effect and the texture format is one of the GL_DEPTH formats. *param* is one of GL_TEXTURE_LEQUAL_R_EXT or GL_TEXTURE_GEQUAL_R_EXT. When the depth texture comparison operator is enabled, the r coordinate is interpolated over the primitive and compared with the depth value found at the interpolated *s* and *t* coordinate location in the texture map. This comparison is either less-than-or-equal-to (GL_TEXTURE_LEQUAL_R_EXT) or greater-than-or-equal-to (GL_TEXTURE_GEQUAL_R_EXT). The result of the comparison is 0.0 if it fails, or 1.0 if it passes. If texture filtering is enabled, this comparison is performed for all of the texels involved in the filtering operation, and the resulting values interpolated (note that only GL_LINEAR and GL_NEAREST minification and magnification filters are supported for depth texture comparison). This result is passed down as the alpha component of the texture color to subsequent texture application; the red, green, and blue components are set to 0.0. The depth comparison operator is typically used to produce shadow effects.

Notes

Suppose that a program has enabled texturing (by calling glEnable with argument GL_TEXTURE_1D or GL_TEXTURE_2D) 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, glCopyTexImage1D, or glCopyTexImage2D) 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. In 3D textures, linear filtering accesses the eight nearest texture elements.

The GL_CLAMP_TO_BORDER_EXT *param* to GL_WRAP_S, GL_WRAP_T, and GL_WRAP_R_EXT is only supported if the extension GL_EXT_texture_border_clamp is supported.

The GL_CLAMP_TO_EDGE_EXT *param* to GL_WRAP_S, GL_WRAP_T, and GL_WRAP_R_EXT is only supported if the extension GL_EXT_texture_edge_clamp is supported.

GL_TEXTURE_WRAP_R_EXT and the *target* GL_TEXTURE_3D_EXT are only supported if the extension GL_EXT_texture3D is supported.

GL_GENERATE_MIPMAP_EXTis only supported if the extension GL_EXT_generate_mipmap is supported.

GL_TEXTURE_COMPARE_EXT and GL_TEXTURE_COMPARE_OPERATOR_EXT are only supported if the extension GL_EXT_shadow is supported.

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

See Also

glBindTexture,
glCopyPixels,
glCopyTexImage1D,
glCopyTexImage2D,
glCopyTexSubImage1D,
glCopyTexSubImage2D,

glDrawPixels,
glPixelStore,
glPixelTransfer,
glPrioritizeTextures,
glTexEnv,
glTexGen,
glTexImage1D,
glTexImage2D,
glTexSubImage1D,
glTexSubImage2D

glTexSubImage1D

`glTexSubImage1D`: specify a two-dimensional texture sub-image.

C Specification

```
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 *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 sub-image.
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, and GL_LUMINANCE_ALPHA. If the extension GL_EXT_shadow is supported, then the symbolic value GL_DEPTH_COMPONENT is also accepted.
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 or disable one-dimensional texturing, call glEnable and glDisable with argument GL_TEXTURE_1D.

glTexSubImage1D 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 width of 0, but such a specification has no effect.

Notes

glTexSubImage1D is available only if the GL version is 1.1 or greater.

Texturing has no effect in color index mode.

The *format* GL_DEPTH_COMPONENT may only be used if the GL_EXT_shadow extension is supported.

glPixelStore and glPixelTransfer modes affect texture images in exactly the way they affect glDrawPixels.

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 0.
- GL_INVALID_VALUE may be generated if *level* is greater than $\log_2$max, where *max* is the returned value of GL_MAX_TEXTURE_SIZE.
- 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_VALUE is generated if *width* is less than 0.
- 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

See Also

glCopyTexImage1D,
glCopyTexImage2D,
glCopyTexSubImage1D,
glCopyTexSubImage2D,
glDrawPixels,
glPixelStore,
glPixelTransfer,
glTexEnv,
glTexGen,
glTexImage1D,

glTexImage2D,
glTexParameter,
glTexSubImage2D

glTexSubImage2D

`glTexSubImage2D`: specify a two-dimensional texture sub-image.

C Specification

```
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 *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 sub-image.
height	Specifies the height of the texture sub-image.
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, and GL_LUMINANCE_ALPHA. If the extension GL_EXT_shadow is supported, then the symbolic value GL_DEPTH_COMPONENT is also accepted.
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 two-dimensional texturing, call glEnable and glDisable with argument GL_TEXTURE_2D.

glTexSubImage2D 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

glTexSubImage2D is available only if the GL version is 1.1 or greater.

Texturing has no effect in color index mode.

The *format* GL_DEPTH_COMPONENT may only be used if the GL_EXT_shadow extension is supported.

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_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 0.
- GL_INVALID_VALUE may be generated if *level* is greater than $\log_2$ max, where *max* is the returned value of GL_MAX_TEXTURE_SIZE.
- 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_VALUE is generated if *width* or *height* is less than 0.
- 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

See Also

glCopyTexImage1D,
glCopyTexImage2D,
glCopyTexSubImage1D,
glCopyTexSubImage2D,
glDrawPixels,

glPixelStore,
glPixelTransfer,
glTexEnv,
glTexGen,
glTexImage1D,
glTexImage2D,
glTexSubImage1D,
glTexParameter

glTexSubImage3DEXT

glTexSubImage3DEXT: specify a three-dimensional texture sub-image.

C Specification

```
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 *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.
zoffset	Specifies a texel offset in the Z direction within the texture array.
width	Specifies the width of the texture sub-image.
height	Specifies the height of the texture sub-image.
depth	Specifies the depth of the texture sub-image.
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, and GL_LUMINANCE_ALPHA. If the extension GL_EXT_shadow is supported, the symbolic value GL_DEPTH_COMPONENT is also accepted.
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. Three-dimensional texturing is enabled and disabled using glEnable and glDisable with argument GL_TEXTURE_3D_EXT.

glTexSubImage3DEXT 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, and Z indices *zoffset* and *zoffset* + *depth* - 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, height or depth, 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.

Errors

- GL_INVALID_ENUM is generated when *target* is not GL_TEXTURE_3D_EXT.
- GL_INVALID_OPERATION is generated when the texture array has not been defined by a previous glTexImage3D operation.
- GL_INVALID_VALUE is generated if *level* is less than zero or greater than $\log_2$max, where *max* is the returned value of GL_MAX_3D_TEXTURE_SIZE_EXT.
- GL_INVALID_VALUE is generated if *xoffset* <TEXTURE_BORDER, (*xoffset* + *width*) > (TEXTURE_WIDTH TEXTURE_BORDER), *yoffset* < TEXTURE_BORDER, *zoffset* < TEXTURE_BORDER, or (*zoffset* + *depth*) > (TEXTURE_DEPTH_EXT TEXTURE_BORDER), where TEXTURE_WIDTH, TEXTURE_HEIGHT, TEXTURE_DEPTH_EXT and TEXTURE_BORDER are the state values of the texture image being modified. Note that TEXTURE_WIDTH, TEXTURE_HEIGHT and TEXTURE_DEPTH_EXT 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 glTexSubImage3DEXT is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGetTexImage
glIsEnabled with argument GL_TEXTURE_3D_EXT

See Also

glDrawPixels,
glFog,
glPixelStore,
glPixelTransfer,
glTexEnv,
glTexGen,
glTexImage3D,
glTexParameter.

glTranslate

`glTranslated, glTranslatef`: multiply the current matrix by a translation matrix.

C Specification

```
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

glTranslate produces a translation by (*x, y, z*). The current matrix (see glMatrixMode) is multiplied by this translation matrix, with the product replacing the current matrix, as if glMultMatrix were called with the following matrix for its argument:

$$\begin{matrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{matrix}$$

If the matrix mode is either GL_MODELVIEW or GL_PROJECTION, all objects drawn after a call to glTranslate are translated.

Use glPushMatrix and glPopMatrix to save and restore the un-translated coordinate system.

Errors

- GL_INVALID_OPERATION is generated if glTranslate is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_MATRIX_MODE
glGet with argument GL_MODELVIEW_MATRIX
glGet with argument GL_PROJECTION_MATRIX
glGet with argument GL_TEXTURE_MATRIX

See Also

glMatrixMode,
glMultMatrix,
glPushMatrix,
glScale

19 U

gluUnProject

`gluUnProject`: map window coordinates to object coordinates.

C Specification

```
GLint gluUnProject(
    GLdouble winX,
    GLdouble winY,
    GLdouble winZ,
    const GLdouble *model,
    const GLdouble *proj,
    const GLint *view,
    GLdouble* objX,
    GLdouble* objY,
    GLdouble* objZ)
```

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.

Description

gluUnProject maps the specified window coordinates into object coordinates using *model*, *proj*, and *view*. The result is stored in *objX*, *objY*, and *objZ*. A return value of GL_TRUE indicates success; a return value of GL_FALSE indicates failure.

To compute the coordinates (*objX*, *objY*, *objZ*), gluUnProject multiplies the normalized device coordinates by the inverse of *model* × *proj* as follows:

$$\begin{pmatrix} \mathrm{objX} \\ \mathrm{objY} \\ \mathrm{objZ} \\ \mathrm{W} \end{pmatrix} = \mathrm{INV(PM)} \begin{pmatrix} 2\,(\mathrm{winX} - \mathrm{view}[0]) / \mathrm{view}[2] - 1 \\ 2\,(\mathrm{winY} - \mathrm{view}[1]) / \mathrm{view}[3] - 1 \\ 2\,\mathrm{winZ} - 1 \\ 1 \end{pmatrix}$$

"INV()" denotes matrix inversion. *W* is an unused variable, included for consistent matrix notation.

See Also

glGet,
gluProject

glXUseXFont

`glXUseXFont`: create bitmap display lists from an X font.

C Specification

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

Parameters

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

Description

glXUseXFont generates *count* display lists, named *listBase* through *listBase* + *count* - *1*, each containing a single glBitmap command. The parameters of the glBitmap command of display list *listBase* + *i* are derived from glyph *first* + *i*. Bitmap parameters *xorig, yorig, width*, and *height* are computed from font metrics as *descent* - 1, *lbearing, rbearing* - *lbearing*, and *ascent* + *descent*, respectively. *xmove* is taken from the glyph's *width* metric, and *ymove* is set to zero. Finally, the glyph's image is converted to the appropriate format for glBitmap.

Using glXUseXFont may be more efficient than accessing the X font and generating the display lists explicitly, both because the display lists are created on the server without requiring a round trip of the glyph data, and because the server may choose to delay the creation of each bitmap until it is accessed.

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

Errors

- BadFont is generated if *font* is not a valid font.
- 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.

See Also

glBitmap,
glXMakeCurrent

20 V

glVertex

`glVertex2d, glVertex2f, glVertex2i, glVertex2s, glVertex3d, glVertex3f, glVertex3i, glVertex3s, glVertex4d, glVertex4f, glVertex4i, glVertex4s,glVertex2dv, glVertex2fv, glVertex2iv, glVertex2sv, glVertex3dv, glVertex3fv, glVertex3iv, glVertex3sv, glVertex4dv, glVertex4fv, glVertex4iv, glVertex4sv`: specify a vertex.

C Specification

```
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*; of a three-element array, *x, y,* and *z*; and of a four-element array, *x, y, z,* and *w.*

Description

glVertex commands are used within glBegin/glEnd pairs to specify point, line, and polygon vertices. The current color, normal, and texture coordinates are associated with the vertex when glVertex is called.

When only *x* and *y* are specified, *z* defaults to 0 and *w* defaults to 1. When *x, y,* and *z* are specified, *w* defaults to 1.

Notes

Invoking glVertex outside of a glBegin/glEnd pair results in undefined behavior.

See Also

glBegin,
glCallList,
glColor,
glEdgeFlag,
glEvalCoord,
glIndex,
glMaterial,
glNormal,
glRect,
glTexCoord,
glVertexPointer

glVertexPointer

`glVertexPointer`: define an array of vertex data.

C Specification

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

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, and GL_DOUBLE are accepted. The initial value is GL_FLOAT.
stride	Specifies the byte offset between consecutive vertexes. If *stride* is 0, the vertexes 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.

Description

glVertexPointer 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* specifies 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; 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 pre-specified vertex and vertex attribute arrays. Use glArrayElement to specify primitives by indexing vertexes and vertex attributes and glDrawElements to construct a sequence of primitives by indexing vertexes and vertex attributes.

Notes

glVertexPointer is available only if the GL version is 1.1 or greater.

The vertex array is initially disabled and isn't accessed when glArrayElement, glDrawElements or glDrawArrays is called.

Execution of glVertexPointer is not allowed between the execution of glBegin and the corresponding execution of glEnd, but an error may or may not be generated. If no error is generated, the operation is undefined.

glVertexPointer is typically implemented on the client side.

Vertex array parameters are client-side state and are therefore not saved or restored by glPushAttrib and glPopAttrib. Use glPushClientAttrib and glPopClientAttrib instead.

Errors

- 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.
- 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

See Also

glArrayElement,
glColorPointer,
glDrawArrays,
glDrawElements,
glEdgeFlagPointer,
glEnable,
glGetPointerv,
glIndexPointer,
glInterleavedArrays,
glNormalPointer,
glPopClientAttrib,
glPushClientAttrib,
glTexCoordPointer

glViewport

`glViewport`: set the viewport.

C Specification

```
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 initial value is (0, 0).
width, height	Specify the width and height 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

glViewport 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:

$x_w = (x_{nd} + 1) \cdot (\text{width} / 2) + x$

$y_w = (y_{nd} + 1) \cdot (\text{height} / 2) + y$

Viewport width and height are silently clamped to a range that depends on the implementation. To query this range, call glGet with argumentGL_MAX_VIEWPORT_DIMS.

Errors

- GL_INVALID_VALUE is generated if either *width* or *height* is negative.
- GL_INVALID_OPERATION is generated if glViewport is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGet with argument GL_VIEWPORT
glGet with argument GL_MAX_VIEWPORT_DIMS

See Also

glDepthRange

glVisibilityBufferhp

`glVisibilityBufferhp` - establish a buffer for Visibility Test results

C Specification

```
void glVisibilityBufferhp(GLSizei size,
    GLboolean *buffer,
    GLboolean wait_on_get)
```

Parameters

size	Specifies the size of buffer (in bytes)
buffer	Returns the Visibility Test results
wait_on_get	Specifies whether a glGet of VISIBILITY_TEST_hp waits for all Visibility Test data to be returned from the hardware before returning to the calling program.

Description

glVisibilityBuffer has three arguments: 'buffer' is a pointer to an array of Boolean, and 'size' indicates the size of the array. 'buffer' returns values of GL_TRUE or GL_FALSE for each Visibility Test issued. A Visibility Test begins when a call is made to glEnable(GL_VISIBILITY_TEST_hp), and ends when a call is made to either glDisable(GL_VISIBILITY_TEST_hp), or glNextVisibilityTesthp(). When all Visibility Tests have been performed, glDisable(GL_VISIBILITY_TEST_hp) should be called, followed by a call to glGet of GL_VISIBILITY_TEST_hp. Calling glGet causes the contents of 'buffer' to be up to date (based on the value of the 'wait_on_get' parameter, discussed below). A Visibility Test result of GL_TRUE indicates that some portion of the primitive(s) rendered during that test were visible. A value of GL_FALSE indicates that no portion of the primitive(s) rendered during the test was visible.

The third parameter, 'wait_on_get', is intended to allow for potential optimizations during Visibility Tests. When Visibility Tests are made, there is often some amount of latency between the time the request is made and when the answer is available. Once a glGet of GL_VISIBILITY_TEST_hp is called after glDisable(GL_VISIBILITY_TEST_hp), there will usually be a short delay (perhaps as much as 50 microseconds) before the answer is available. If the application would prefer to wait until the Visibility Test results are available before proceeding, then a value of GL_TRUE should be specified for the 'wait_on_get' parameter. If, on the other hand, the application has some useful work to do during that time, a value of GL_FALSE can be specified for the 'wait_on_get' parameter, and glGet will return immediately. However, 'buffer' is not guaranteed to be defined until some time later. To check to see when the Visibility Test results are complete, the programmer can look at the data returned by a glGet of GL_VISIBILITY_TEST_hp. Two Boolean values are returned; The first indicates whether or not GL_VISIBILITY_TEST_hp is currently enabled. The second indicates if there are any test results pending. If there are no test results pending, the programmer may conclude that 'buffer' is up to date.

Visibility Tests can be used by a programmer to determine whether or not a primitive or set of primitives are visible against the current contents of the depth buffer; “Visibile” in this context means that at least one pixel passes the depth test. This differs from previous Occlusion Culling extensions by allowing the programmer to provide memory for many different Visibility Test results to be returned at the same time.

Notes

The contents of ’buffer’ are undefined before a glGet of GL_VISIBILITY_TEST_hp is called. See Description section above for more information on when the contents of ’buffer’ become valid.

The programmer is responsible for allocating the memory for ’buffer’. If an incorrect ’size’ is specified, memory corruption may occur.

Errors

- GL_INVALID_VALUE is generated if ’size’ is negative.
- GL_INVALID_OPERATION is generated if glVisibilityBufferhp is called while GL_VISIBILITY_TEST_hp is enabled (via glEnable()), or if glEnable(GL_VISIBILITY_TEST_hp) is called before glVisibilityBufferhp is called at least once.
- GL_INVALID_OPERATION is generated if glVisibilityBufferhp is called between a call to glBegin and the corresponding call to glEnd.

Associated Gets

glGet with argument GL_VISIBILITY_TEST_hp

See Also

glNextVisibilityTesthp
glIsEnabled with argument

21 W

glXWaitGL

`glXWaitGL`: complete GL execution prior to subsequent X calls.

C Specification

```
void glXWaitGL(void)
```

Description

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

glXWaitGL is ignored if there is no current GLX context.

Notes

glXWaitGL may or may not flush the X stream.

Errors

- 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.

See Also

glFinish,
glFlush,
glXWaitX,
XSync

glXWaitX

`glXWaitX`: complete X execution prior to subsequent GL calls.

C Specification

```
void glXWaitX(void)
```

Description

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

glXWaitX is ignored if there is no current GLX context.

Note

glXWaitX may or may not flush the GL stream.

Errors

- 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.

See Also

glFinish,
glFlush,
glXWaitGL,
XSync