OpenGL® ES 1.0 Reference Manual
Version 1.0

Editor: Claude Knaus

December 19, 2003

Contents

Chapter 1

Preface

Style Conventions

The following style conventions apply to this document:

Bold	Command and function names
Italics	Variables and parameters
Regular	Ordinary text
`Monospace`	Constants and source code

Chapter 2

Summary of Commands and Routines

OpenGL ES Commands

Primitives

Specify polygon offset:

glPolygonOffset

Vertex Arrays

Specify vertex arrays:

glColorPointer

glNormalPointer

glVertexPointer

glTexCoordPointer

Select the active texture coordinate array:

glClientActiveTexture

Draw vertex arrays:

- **glEnableClientState**
- **glDisableClientState**
- **glDrawArrays**
- **glDrawElements**

Coordinate Transformation

Transform the current matrix:

- **glRotate**
- **glScale**
- **glTranslate**
- **glMultMatrix**
- **glFrustum**
- **glOrtho**

Replace the current matrix:

- **glLoadMatrix**
- **glLoadIdentity**

Manipulate the matrix stack:

- **glMatrixMode**
- **glPushMatrix**
- **glPopMatrix**

Specify the viewport:

- **glDepthRange**
- **glViewport**

Color and Lighting

Set the current color and normal vector:

glColor

glNormal

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

glLight

glMaterial

glLightModel

Choose a shading model:

glShadeModel

Specify which polygon orientation is front-facing:

glFrontFace

Rasterization

Specify the dimensions of points or lines:

glPointSize

glLineWidth

Choose when polygons are rasterized:

glCullFace

Pixel Operations

Read pixels:

glReadPixels

Specify how pixels are encoded:

glPixelStore

Textures

Control how a texture is applied to a fragment:

glTexParameter

glTexEnv

Set the current texture coordinates:

glMultiTexCoord

Specify a two-dimensional texture image or texture subimage:

glTexImage2D

glTexSubImage2D

Copy a texture or part of it:

glCopyTexImage2D

glCopyTexSubImage2D

Specify a two-dimensional compressed texture image or texture subimage:

glCompressedTexImage2D

glCompressedTexSubImage2D

Create a named texture:

glBindTexture

glDeleteTextures

glGenTextures

Select the active texture unit:

glActiveTexture

Fog

Set fog parameter:

- **glFog**

Frame Buffer Operations

Control per-fragment testing:

- **glScissor**
- **glAlphaFunc**
- **glStencilFunc**
- **glStencilOp**
- **glDepthFunc**

Combine fragment and frame buffer values:

- **glBlendFunc**
- **glLogicOp**

Clear buffers:

- **glClear**
- **glClearColor**
- **glClearDepth**
- **glClearStencil**

Control buffers enabled for writing:

- **glColorMask**
- **glDepthMask**
- **glStencilMask**

Control multisampling:

- **glSampleCoverage**

Modes and Execution

Enable and disable modes:

glEnable
glDisable

Wait until all GL commands have executed completely:

glFinish

Force all issued GL commands to be executed:

glFlush

Specify hints for GL operation:

glHint

State Queries

Obtain information about an error or the current GL connection:

glGetError
glGetString

Query static state variables:

glGetInteger

OES Extensions

Query Matrix

Query the current matrix:

glQueryMatrix

EGL Functions

Manage or query display connections:

- **eglGetDisplay**
- **eglInitialize**
- **eglQueryString**
- **eglTerminate**

Request GL and EGL extension functions:

- **eglGetProcAddress**

Query errors:

- **eglGetError**

Request or query frame buffer configurations:

- **eglGetConfigs**
- **eglChooseConfig**
- **eglGetConfigAttrib**

Manage or query EGL rendering contexts:

- **eglCreateContext**
- **eglDestroyContext**
- **eglQueryContext**
- **eglMakeCurrent**
- **eglGetCurrentContext**
- **eglGetCurrentSurface**
- **eglGetCurrentDisplay**

Manage or query EGL surfaces:

- **eglCreateWindowSurface**

> **eglCreatePixmapSurface**
>
> **eglCreatePbufferSurface**
>
> **eglDestroySurface**
>
> **eglQuerySurface**

Synchronize execution:

> **eglWaitGL**
>
> **eglWaitNative**

Post and copy buffers:

> **eglCopyBuffers**
>
> **eglSwapBuffers**

Chapter 3

OpenGL ES Reference Pages

Name

glActiveTexture – select server-side active texture unit

C Specification

`void` **glActiveTexture**`(GLenum` *texture*`)`

Parameters

texture	Specifies which texture unit to make active. The number of texture units is implementation dependent, but must be at least one. *texture* must be one of `GL_TEXTURE`i, where $0 \leq i <$ `GL_MAX_TEXTURE_UNITS`, which is an implementation-dependent value. The intial value is `GL_TEXTURE0`.

Description

glActiveTexture selects which texture unit subsequent texture state calls will affect. The number of texture units an implementation supports is implementation dependent, it must be at least 1.

Errors

`GL_INVALID_ENUM` is generated if *texture* is not one of `GL_TEXTURE`i, where $0 \leq i <$ `GL_MAX_TEXTURE_UNITS`.

Notes

It is always the case that `GL_TEXTURE`i = `GL_TEXTURE0` $+ i$.

A texture unit consists of the texture enable state, texture matrix stack, texture environment and currently bound texture. Modifying any of these states has an effect only on the active texture unit.

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

Associated Gets

glGetInteger with argument `GL_MAX_TEXTURE_UNITS`

See Also

glBindTexture, **glClientActiveTexture**, **glEnable**, **glGetInteger**, **glMatrix-Mode**, **glMultiTexCoord**, **glTexEnv**

Name

glAlphaFunc, **glAlphaFuncx** – specify the alpha test function

C Specification

`void` **glAlphaFunc**`(GLenum` *func*`, GLclampf` *ref*`)`
`void` **glAlphaFuncx**`(GLenum` *func*`, GLclampx` *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, 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. To enable and disable alpha testing, call **glEnable** and **glDisable** with argument `GL_ALPHA_TEST`. Alpha testing is initially disabled.

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, and polygons. **glAlphaFunc** does not affect **glClear**.

Errors

`GL_INVALID_ENUM` is generated if *func* is not an accepted value.

See Also

glBlendFunc, **glClear**, **glDepthFunc**, **glEnable**, **glStencilFunc**

Name

glBindTexture – bind a named texture to a texturing target

C Specification

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

Parameters

target	Specifies the target to which the texture is bound. Must be `GL_TEXTURE_2D`.
texture	Specifies the name of a texture.

Description

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

Texture names are unsigned integers. The value 0 is reserved to represent the default texture for each texture target. Texture names and the corresponding texture contents are local to the shared texture-object space (see **eglCreateContext**) of the current GL rendering context.

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

While a texture is bound, GL operations on the target to which it is bound affect 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**.

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 **glTexImage2D**.

Errors

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

See Also

eglCreateContext, **glActiveTexture**, **glCompressedTexImage2D**, **glCopyTexImage2D**, **glDeleteTextures**, **glGenTextures**, **glGetInteger**, **glTexImage2D**, **glTexParameter**

Name

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. The following 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

Pixels can be drawn using a function that blends the incoming (source) values with the values that are already in the color buffer (the destination values). Use **glEnable** and **glDisable** with argument `GL_BLEND` to enable and disable blending. Blending is initially disabled.

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].

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,

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

To determine the blended values of a pixel, the system uses the following equations:

$$\begin{aligned}
R_d &= \min\left(k_R, R_s s_R + R_d d_R\right) \\
G_d &= \min\left(k_G, G_s s_G + G_d d_G\right) \\
B_d &= \min\left(k_B, B_s s_B + B_d d_B\right) \\
A_d &= \min\left(k_A, A_s s_A + A_d d_A\right)
\end{aligned}$$

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:

$$\begin{aligned} R_d &= R_s \\ G_d &= G_s \\ B_d &= B_s \\ A_d &= A_s \end{aligned}$$

glBlendFunc operates on all pixel write operations, including the scan conversion of points, lines, and polygons. **glBlendFunc** does not affect **glClear**.

Examples

Transparency is best implemented using **glBlendFunc**(`GL_SRC_ALPHA`, `GL_ONE_MINUS_SRC_ALPHA`) with primitives sorted from farthest to nearest. Note that this transparency calculation does not require the presence of alpha bitplanes in the color buffer.

glBlendFunc(`GL_SRC_ALPHA`, `GL_ONE_MINUS_SRC_ALPHA`) is also useful for rendering antialiased points and lines.

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.

Errors

`GL_INVALID_ENUM` is generated if either *sfactor* or *dfactor* is not an accepted value.

See Also

glAlphaFunc, **glClear**, **glDepthFunc**, **glEnable**, **glLogicOp**, **glStencilFunc**

Name

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. Valid masks are `GL_COLOR_BUFFER_BIT`, `GL_DEPTH_BUFFER_BIT`, and `GL_STENCIL_BUFFER_BIT`.

Description

glClear sets the bitplane area of the window to values previously selected by **glClearColor**, **glClearDepth**, and **glClearStencil**.

The pixel ownership test, the scissor test, dithering, and the color buffer masks 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 color buffer.

`GL_DEPTH_BUFFER_BIT`
    Indicates the depth 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 defined bits is set in *mask*.

See Also

glClearColor, **glClearDepth**, **glClearStencil**, **glColorMask**, **glEnable**, **glScissor**,

Name

glClearColor, **glClearColorx** – specify clear values for the color buffer

C Specification

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

```
void glClearColorx(GLclampx red,
                   GLclampx green,
                   GLclampx blue,
                   GLclampx alpha)
```

Parameters

red, *green*, *blue*, *alpha*

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

Description

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

See Also

glClear, **glClearDepth**, **glClearStencil**, **glColorMask**

Name

glClearDepthf, **glClearDepthx** – specify the clear value for the depth buffer

C Specification

void **glClearDepthf**(GLclampf *depth*)
void **glClearDepthx**(GLclampx *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].

See Also

glClear, **glClearColor**, **glClearStencil**, **glDepthFunc**, **glDepthMask**

Name

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.

Associated Gets

glGetInteger with argument `GL_STENCIL_BITS`

See Also

glClear, **glClearColor**, **glClearDepth**, **glGetInteger**, **glStencilFunc**, **glStencilOp**, **glStencilMask**

Name

glClientActiveTexture – select client-side active texture unit

C Specification

`void` **glClientActiveTexture**`(GLenum` *texture*`)`

Parameters

texture	Specifies which texture unit to make active. The number of texture units is implementation dependent, but must be at least one. *texture* must be one of `GL_TEXTURE`i, $0 \leq i <$ `GL_MAX_TEXTURE_UNITS`, which is an implementation-dependent value. The initial value is `GL_TEXTURE0`.

Description

glClientActiveTexture selects the vertex array client state parameters to be modified by **glTexCoordPointer**, and enabled or disabled with **glEnableClientState** or **glDisableClientState**, respectively, when called with a parameter of `GL_TEXTURE_COORD_ARRAY`.

Errors

`GL_INVALID_ENUM` is generated if *texture* is not one of `GL_TEXTURE`i, where $0 \leq i <$ `GL_MAX_TEXTURE_UNITS`.

Notes

It is always the case that `GL_TEXTURE`i = `GL_TEXTURE0` + i.

Associated Gets

glGetInteger with argument `GL_MAX_TEXTURE_UNITS`

See Also

glActiveTexture, **glEnableClientState**, **glGetInteger**, **glMultiTexCoord**, **gl-TexCoordPointer**

Name

glColor4f, **glColor4x** – set the current color

C Specification

```
void glColor4f(GLfloat red,
               GLfloat green,
               GLfloat blue,
               GLfloat alpha)

void glColor4x(GLfixed red,
               GLfixed green,
               GLfixed blue,
               GLfixed alpha)
```

Parameters

red, *green*, *blue*, *alpha*
Specify new red, green, blue, and alpha values for the current color. The initial value is (1, 1, 1, 1).

Description

The GL stores a current four-valued RGBA color. **glColor** sets a new four-valued RGBA color.

Current color values are stored in fixed-point or floating-point. In case the values are stored in floating-point, the mantissa and exponent sizes are unspecified.

Neither fixed-point nor floating-point 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 the color buffer.

See Also

glColorPointer, **glNormal**, **glMultiTexCoord**

Name

glColorMask – enable and disable writing of color buffer 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 color buffer. The initial values are all `GL_TRUE`, indicating that all color components can be written.

Description

glColorMask specifies whether the individual components in the color 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 the color buffer, regardless of the drawing operation attempted, including **glClear**.

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

See Also

glClear, **glColor**, **glColorPointer**, **glDepthMask**, **glStencilMask**

Name

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 4. The initial value is 4.
type	Specifies the data type of each color component in the array. Symbolic constants `GL_UNSIGNED_BYTE` and `GL_FIXED` are accepted. However, the initial value is `GL_FLOAT`. The common profile accepts the symbolic constant `GL_FLOAT` as well.
stride	Specifies the byte offset between consecutive colors. If *stride* is 0, the colors are understood to be tightly packed in the array. The initial value is 0.
pointer	Specifies a pointer to the first component of the first color element in the array.

Description

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

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

If the color array is enabled, it is used when **glDrawArrays**, or **glDrawElements** is called. To enable and disable the color array, call **glEnableClientState** and **glDisableClientState** with the argument `GL_COLOR_ARRAY`. The color array is initially disabled and isn't accessed when **glDrawArrays** or **glDrawElements** is called.

Use **glDrawArrays** to construct a sequence of primitives (all of the same type) from prespecified vertex and vertex attribute arrays. Use **glDrawElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

Notes

glColorPointer is typically implemented on the client side.

Errors

`GL_INVALID_VALUE` is generated if *size* is not 4.

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

`GL_INVALID_VALUE` is generated if *stride* is negative.

See Also

glColor, **glDrawArrays**, **glDrawElements**, **glEnableClientState**, **glNormalPointer**, **glTexCoordPointer**, **glVertexPointer**

Name

glCompressedTexImage2D – specify a two-dimensional compressed texture image

C Specification

```
void glCompressedTexImage2D(GLenum target,
                            GLint level,
                            GLint internalformat,
                            GLsizei width,
                            GLsizei height,
                            GLint border,
                            GLsizei imageSize,
                            const GLvoid * data)
```

Parameters

target Specifies the target texture. Must be `GL_TEXTURE_2D`.

level Specifies the level-of-detail number. Must be less than or equal to 0. Level 0 indicates a single mip-level. Negative values indicate how many mip-levels are described by *data*.

internalformat
Specifies the color components in the texture. The following symbolic constants are accepted: `GL_PALETTE4_RGB8_OES`, `GL_PALETTE4_RGBA8_OES`, `GL_PALETTE4_R5_G6_B5_OES`, `GL_PALETTE4_RGBA4_OES`, `GL_PALETTE4_RGB5_A1_OES`, `GL_PALETTE8_RGB8_OES`, `GL_PALETTE8_RGBA8_OES`, `GL_PALETTE8_R5_G6_B5_OES`, `GL_PALETTE8_RGBA4_OES`, and `GL_PALETTE8_RGB5_A1_OES`.

width Specifies the width of the texture image. Must be $2^n + 2border$ 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 $2^m + 2border$ 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 0.

imageSize
Specifies the size of the compressed image data in bytes.

data Specifies a pointer to the compressed image data in memory.

Description

glCompressedTexImage2D defines a two-dimensional texture image in compressed format.

The supported compressed formats are paletted textures. The layout of the compressed image is a palette followed by multiple mip-levels of texture indices used for lookup into the palette. The palette format can be one of `R5_G6_B5`, `RGBA4`, `RGB5_A1`, `RGB8`, or `RGBA8`. The texture indices can have a resolution of 4 or 8 bits. As a result, the number of palette entries is either 16 or 256. If *level* is 0, only one mip-level of texture indices is described in *data*. Otherwise, the negative value of *level* specifies up to which mip-level the texture indices are described. A possibly remaining pad nibble for the lowest resolution mip-level is ignored.

Notes

glPixelStore has no effect on compressed texture images.

glCompressedTexImage2D specifies the two-dimensional texture for the currently bound texture, specified with **glBindTexture**, and the current texture unit, specified with **glActiveTexture**.

Errors

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

GL_INVALID_VALUE may be generated if if *level* is greater than 0 or the absolute value of *level* is greater than $\log_2 max$, where *max* is the returned value of GL_MAX_TEXTURE_SIZE.

GL_INVALID_VALUEis generated if *internalformat* is not one of the accepted 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 $2^k + 2border$ for some integer k.

GL_INVALID_VALUEis generated if *border* is not 0.

GL_INVALID_VALUEis generated if *imageSize* is not consistent with format, dimentions, and contents of the compressed image.

See Also

glActiveTexture, **glBindTexture**, **glCompressedTexSubImage2D**, **glCopyTexImage2D**, **glCopyTexSubImage2D**, **glPixelStore**, **glTexEnv**, **glTexImage2D**, **glTexParameter**

Name

glCompressedTexSubImage2D – specify a two-dimensional compressed texture subimage

C Specification

```
void glCompressedTexSubImage2D(GLenum target,
                               GLint level,
                               GLint xoffset,
                               GLint yoffset,
                               GLsizei width,
                               GLsizei height,
                               GLenum format,
                               GLsizei imageSize,
                               const GLvoid * data)
```

Parameters

target	Specifies the target texture. Must be `GL_TEXTURE_2D`.
level	Specifies the level-of-detail number.
xoffset	Specifies a texel offset in the x direction within the texture array.
yoffset	Specifies a texel offset in the y direction within the texture array.
width	Specifies the width of the texture subimage.
height	Specifies the height of the texture subimage.
format	Specifies the format of the pixel data. Currently, there is no supported format.
imageSize	Specifies the size of the compressed pixel data in bytes.
data	Specifies a pointer to the compressed image data in memory.

Description

glCompressedTexSubImage2D redefines a contiguous subregion of an existing two-dimensional compressed 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.

Currently, there is no supported compressed format for this function.

Notes

glPixelStore has no effect on compressed texture images.

glCompressedTexSubImage2D specifies the two-dimensional sub texture for the currently bound texture, specified with **glBindTexture**, and the current texture unit, specified with **glActiveTexture**.

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 **glCompressedTexImage2D** 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 texture width, h is the texture height, and b is the border 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 *type* is not a type constant.

`GL_INVALID_OPERATION` is generated if *type* is `GL_UNSIGNED_SHORT_5_6_5` and *format* is not `GL_RGB`.

`GL_INVALID_OPERATION` is generated if *type* is one of `GL_UNSIGNED_SHORT_4_4_4_4`, or `GL_UNSIGNED_SHORT_5_5_5_1` and *format* is not `GL_RGBA`.

`GL_INVALID_OPERATION` is generated if none of the above error conditions apply.

Associated Gets

glGetInteger with argument `GL_MAX_TEXTURE_SIZE`

See Also

glActiveTexture, **glBindTexture**, **glCompressedTexImage2D**, **glCopyTexSubImage2D**, **glGetInteger**, **glPixelStore**, **glTexEnv**, **glTexParameter**

Name

glCopyTexImage2D – specify a two-dimensional texture image with pixels from the color buffer

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 nth mipmap reduction image.

internalformat
Specifies the color components of the texture. Must be one of the following symbolic constants: GL_ALPHA, GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_RGB, or GL_RGBA.

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 $2^n + 2border$ for some integer n.

height Specifies the height of the texture image. Must be 0 or $2^m + 2border$ for some integer m.

border Specifies the width of the border. Must be 0.

Description

glCopyTexImage2D defines a two-dimensional texture image with pixels from the color buffer.

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

The red, green, blue, and alpha components of each pixel that is read are converted to an internal fixed-point or 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 values are then converted to the texture's internal format for storage in the texel array.

internalformat must be chosen such that color buffer components can be dropped during conversion to the internal format, but new components cannot be added. For example, an `RGB` color buffer can be used to create `LUMINANCE` or `RGB` textures, but not `ALPHA`, `LUMINANCE_ALPHA` or `RGBA` textures.

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 color buffer are outside the window associated with the current rendering context, then the values obtained for those pixels are undefined.

Notes

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_OPERATION` is generated if *internalformat* is not compatible with the color buffer format.

`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 `GL_MAX_TEXTURE_SIZE`, or if *width* or *height* cannot be represented as $2^k + 2border$ for some integer *k*.

`GL_INVALID_VALUE` is generated if *border* is not 0.

`GL_INVALID_VALUE` is generated if *internalformat* is not an accepted constant.

Associated Gets

glGetInteger with argument `GL_MAX_TEXTURE_SIZE`

See Also

glCompressedTexImage2D, **glCopyTexSubImage2D**, **glGetInteger**, **glTexEnv**, **glTexImage2D**, **glTexSubImage2D**, **glTexParameter**

Name

glCopyTexSubImage2D – specify a two-dimensional texture subimage with pixels from the color buffer

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 nth mipmap reduction image.
xoffset	Specifies a texel offset in the x direction within the texture array.
yoffset	Specifies a texel offset in the y direction within the texture array.
x, *y*	Specify the window coordinates of the lower left corner of the rectangular region of pixels to be copied.
width	Specifies the width of the texture subimage.
height	Specifies the height of the texture subimage.

Description

glCopyTexSubImage2D replaces a rectangular portion of a two-dimensional texture image with pixels from the color buffer.

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 the same way as with **glCopyTexImage2D**.

glCopyTexSubImage2D requires that the internal format of the currently bound texture is such that color buffer components can be dropped during conversion to the internal format, but new components cannot be added. For example, an `RGB` color buffer can be used to create `LUMINANCE` or `RGB` textures, but not `ALPHA`, `LUMINANCE_ALPHA` or `RGBA` textures.

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 color 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.

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 or if the internal format of the currently bound texture is not compatible with the color buffer format.

`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 $y < -b$, where b is the border of the texture being modified.

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

Associated Gets

glGetInteger with argument `GL_MAX_TEXTURE_SIZE`

See Also

glCompressedTexSubImage2D, **glCopyTexImage2D**, **glGetInteger**, **glTexEnv**, **glTexImage2D**, **glTexParameter**, **glTexSubImage2D**

Name

glCullFace – specify whether front- or back-facing polygons are culled

C Specification

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

Parameters

mode	Specifies whether front- or back-facing polygons are culled. 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 polygons are culled (as specified by *mode*) when culling is enabled. To enable and disable culling, call **glEnable** and **glDisable** with argument `GL_CULL_FACE`. Culling is initially disabled.

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

Notes

If *mode* is `GL_FRONT_AND_BACK`, no polygons 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.

See Also

glEnable, **glFrontFace**

Name

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 reverts to 0 (the default texture).

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

Errors

`GL_INVALID_VALUE` is generated if *n* is negative.

See Also

glBindTexture, **glGenTextures**

Name

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. To enable and disable depth testing, call **glEnable** and **glDisable** with argument `GL_DEPTH_TEST`. Depth testing is initially disabled.

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. 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.

See Also

glDepthRange, **glEnable**, **glPolygonOffset**

Name

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. The initial value is `GL_TRUE`.

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.

Notes

glDepthMask does not affect **glClear**.

See Also

glClear, **glColorMask**, **glDepthFunc**, **glDepthRange**, **glStencilMask**

Name

glDepthRangef, **glDepthRangex** – specify mapping of depth values from normalized device coordinates to window coordinates

C Specification

void **glDepthRangef**(GLclampf *near*, GLclampf *far*)
void **glDepthRangex**(GLclampx *near*, GLclampx *far*)

Parameters

near	Specifies the mapping of the near clipping plane to window coordinates. The initial value is 0.
far	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 *near* be less than *far*. Reverse mappings such as $near = 1$, and $far = 0$ are acceptable.

See Also

glDepthFunc, **glPolygonOffset**, **glViewport**

Name

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`, and `GL_TRIANGLES` 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. You can prespecify separate arrays of vertices, normals, colors, and texture coordinates 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.

Errors

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

`GL_INVALID_VALUE` is generated if *count* is negative.

See Also

glClientActiveTexture, **glColorPointer**, **glDrawElements**, **glNormalPointer**, **glTexCoordPointer**, **glVertexPointer**

Name

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`, and `GL_TRIANGLES` are accepted.
count	Specifies the number of elements to be rendered.
type	Specifies the type of the values in *indices*. Must be either `GL_UNSIGNED_BYTE` or `GL_UNSIGNED_SHORT`.
indices	Specifies a pointer to the location where the indices are stored.

Description

glDrawElements specifies multiple geometric primitives with very few subroutine calls. You can prespecify separate arrays of vertices, normals, colors, and texture coordinates and use them to construct a sequence of primitives with a single call to **glDrawElements**.

When **glDrawElements** is called, it uses *count* sequential indices from *indices* to lookup elements in enabled arrays to construct a sequence of geometric primitives. *mode* specifies what kind of primitives are constructed, and how the array elements construct these primitives. If `GL_VERTEX_ARRAY` is not enabled, no geometric primitives are 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 maintain their previous values.

Errors

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

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

`GL_INVALID_VALUE` is generated if *count* is negative.

See Also

glClientActiveTexture, **glColorPointer**, **glDrawArrays**, **glNormalPointer**, **glTexCoordPointer**, **glVertexPointer**

Name

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. The initial value for each capability with the exception of `GL_DITHER` and `GL_MULTISAMPLE` is `GL_FALSE`. The initial value for `GL_DITHER` and `GL_MULTISAMPLE` 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_BLEND`
: If enabled, blend the incoming color values with the values in the color buffers. See **glBlendFunc**.

`GL_COLOR_LOGIC_OP`
: If enabled, apply the currently selected logical operation to the incoming color and color buffer values. See **glLogicOp**.

`GL_COLOR_MATERIAL`
: If enabled, have ambient and diffuse material parameters track the current color.

`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**, **glDepthMask**, 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 posttexturing color. See **glFog**.

`GL_LIGHT`*i*
: If enabled, include light i in the evaluation of the lighting equation. See **glLightModel** and **glLight**.

`GL_LIGHTING`
: If enabled, use the current lighting parameters to compute the vertex color. Otherwise, simply associate the current color 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_MULTISAMPLE`
: If enabled, perform multisampling of fragments for single-pass antialiasing and other effects. See **glSampleCoverage**.

`GL_NORMALIZE`
: If enabled, normal vectors are scaled to unit length after transformation. See **glNormal** and **glNormalPointer**.

`GL_POINT_SMOOTH`
: If enabled, draw points with proper filtering. Otherwise, draw aliased points. See **glPointSize**.

`GL_POLYGON_OFFSET_FILL`
: If enabled, an offset is added to depth values of a polygon's fragments before the depth comparison is performed. See **glPolygonOffset**.

`GL_RESCALE_NORMAL`
: If enabled, normal vectors are scaled by a factor derived from the modelview matrix. See **glNormal** and **glNormalPointer**.

`GL_SAMPLE_ALPHA_TO_MASK`
: If enabled, convert fragment alpha values to multisample coverage modification masks. See **glSampleCoverage**.

`GL_SAMPLE_ALPHA_TO_ONE`
: If enabled, set fragment alpha to the maximum permissible value after computing multisample coverage modification masks. See **glSampleCoverage**.

`GL_SAMPLE_MASK`
: If enabled, apply a mask to modify fragment coverage during multisampling. See **glSampleCoverage**.

`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**, **glStencilMask**, and **glStencilOp**.

`GL_TEXTURE_2D`
: If enabled, two-dimensional texturing is performed for the active texture unit. See **glActiveTexture**, **glTexImage2D**, **glCompressedTexImage2D**, and **glCopyTexImage2D**.

Errors

`GL_INVALID_ENUM` is generated if *cap* is not one of the values listed previously.

See Also

glActiveTexture, **glAlphaFunc**, **glBlendFunc**, **glCompressedTexImage2D**, **glCopyTexImage2D**, **glCullFace**, **glDepthFunc**, **glDepthRange**, **glEnableClientState**, **glFog**, **glLight**, **glLightModel**, **glLineWidth**, **glLogicOp**,

glMaterial, **glNormal**, **glPointSize**, **glPolygonOffset**, **glSampleCoverage**, **glScissor**, **glStencilFunc**, **glStencilOp**, **glTexImage2D**

Name

glEnableClientState, **glDisableClientState** – enable or disable client-side capability

C Specification

`void` **glEnableClientState**`(GLenum` *array*`)`
`void` **glDisableClientState**`(GLenum` *array*`)`

Parameters

array Specifies the capability to enable or disable. Symbolic constants `GL_COLOR_ARRAY`, `GL_NORMAL_ARRAY`, `GL_TEXTURE_COORD_ARRAY`, and `GL_VERTEX_ARRAY` are accepted.

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, *array*, 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_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 during 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

Enabling and disabling `GL_TEXTURE_COORD_ARRAY` affects the active client texture unit. The active client texture unit is controlled with **glClientActiveTexture**.

Errors

`GL_INVALID_ENUM` is generated if *array* is not an accepted value.

See Also

glClientActiveTexture, **glColorPointer**, **glDrawArrays**, **glDrawElements**, **glEnable**, **glNormalPointer**, **glTexCoordPointer**, **glVertexPointer**

Name

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.

See Also

glFlush

Name

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.

See Also

glFinish

Name

glFogf, **glFogx**, **glFogfv**, **glFogxv** – specify fog parameters

C Specification

`void` **glFogf**`(GLenum` *pname*`, GLfloat` *param*`)`
`void` **glFogx**`(GLenum` *pname*`, GLfixed` *param*`)`

Parameters

pname	Specifies a single-valued fog parameter. `GL_FOG_MODE`, `GL_FOG_DENSITY`, `GL_FOG_START`, and `GL_FOG_END` are accepted.
param	Specifies the value that *pname* will be set to.

C Specification

`void` **glFogfv**`(GLenum` *pname*`, const GLfloat *` *params*`)`
`void` **glFogxv**`(GLenum` *pname*`, const GLfixed *` *params*`)`

Parameters

pname	Specifies a fog parameter. `GL_FOG_MODE`, `GL_FOG_DENSITY`, `GL_FOG_START`, `GL_FOG_END`, 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

If fog is 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`. Fog is initially disabled.

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

`GL_FOG_COLOR`
: *params* contains four fixed-point or floating-point values that specify C_f, the fog color. Both fixed-point and 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 posttexturing color using a blending factor *f*. Factor *f* is computed in one of three ways, depending on the fog mode. Let *z* be the distance in eye coordinates from the origin to the fragment being fogged. The equation for `GL_LINEAR` fog is

$$f = \frac{end - z}{end - start}$$

The equation for `GL_EXP` fog is

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

The equation for `GL_EXP2` fog is

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

Regardless of the fog mode, f is clamped to the range [0, 1] after it is computed. Then, the fragment's red, green, and blue colors, represented by C_r, are replaced by

$$C_r' = fC_r + (1 - f)C_f$$

Fog does not affect a fragment's alpha component.

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.

See Also

glEnable

Name

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 (culling) these invisible polygons has the obvious benefit of speeding up the rendering of the image. To enable and disable culling, call **glEnable** and **glDisable** with argument `GL_CULL_FACE`. Culling is initially disabled.

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.

See Also

glCullFace, **glEnable**, **glLightModel**

Name

glFrustumf, **glFrustumx** – multiply the current matrix by a perspective matrix

C Specification

```
void glFrustumf(GLfloat left,
                GLfloat right,
                GLfloat bottom,
                GLfloat top,
                GLfloat near,
                GLfloat far)
```

```
void glFrustumx(GLfixed left,
                GLfixed right,
                GLfixed bottom,
                GLfixed top,
                GLfixed near,
                GLfixed far)
```

Parameters

left, *right*
: Specify the coordinates for the left and right vertical clipping planes.

bottom, *top*
: Specify the coordinates for the bottom and top horizontal clipping planes.

near, *far*
: Specify the distances to the 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:

$$\begin{pmatrix} \frac{2}{right-left} & 0 & A & 0 \\ 0 & \frac{2}{top-bottom} & B & 0 \\ 0 & 0 & C & D \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

where

$$\begin{aligned} A &= -\frac{right+left}{right-left} \\ B &= -\frac{top+bottom}{top-bottom} \\ C &= -\frac{far+near}{far-near} \\ D &= -\frac{2farnear}{far-near} \end{aligned}$$

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

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

Notes

Depth buffer precision is affected by the values specified for *near* and *far*. The greater the ratio of *far* to *near* is, the less effective the depth buffer will be at distinguishing between surfaces that are near each other. If

$$r = \frac{far}{near}$$

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

Errors

`GL_INVALID_VALUE` is generated if *near* or *far* is not positive, or if $left = right$, or $bottom = top$.

See Also

glOrtho, **glMatrixMode**, **glMultMatrix**, **glPushMatrix**, **glViewport**

Name

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**.

Errors

`GL_INVALID_VALUE` is generated if *n* is negative.

See Also

glBindTexture, **glCopyTexImage2D**, **glDeleteTextures**, **glTexImage2D**, **glTexParameter**

Name

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.

Name

glGetIntegerv – return the value or values of a selected parameter

C Specification

`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

glGetIntegerv returns values for static state variables in GL. *pname* is a symbolic constant indicating the static state variable to be returned, and *params* is a pointer to an array of integer in which to place the returned data.

The following symbolic constants are accepted by *pname*:

`GL_ALIASED_POINT_SIZE_RANGE`
: *params* returns two values, the smallest and largest supported sizes for aliased points. The range must include 1. See **glPointSize**.

`GL_ALIASED_LINE_WIDTH_RANGE`
: *params* returns two values, the smallest and largest supported widths for aliased lines. The range must include 1. See **glLineWidth**.

`GL_ALPHA_BITS`
: *params* returns one value, the number of alpha bitplanes in the color buffer.

`GL_BLUE_BITS`
: *params* returns one value, the number of blue bitplanes in the color buffer.

`GL_COMPRESSED_TEXTURE_FORMATS`
: *params* returns `GL_NUM_COMPRESSED_TEXTURE_FORMATS` values, the supported compressed texture formats. See **glCompressedTexImage2D** and **glCompressedTexSubImage2D**.

`GL_DEPTH_BITS`
: *params* returns one value, the number of bitplanes in the depth buffer.

`GL_GREEN_BITS`
: *params* returns one value, the number of green bitplanes in the color buffer.

`GL_IMPLEMENTATION_COLOR_READ_FORMAT_OES`
: *params* returns one value, the preferred format for pixel read back. See **glReadPixels**.

`GL_IMPLEMENTATION_COLOR_READ_TYPE_OES`
: *params* returns one value, the preferred type for pixel read back. See **glReadPixels**.

`GL_MAX_ELEMENTS_INDICES`
: *params* returns one value, the recommended maximum number of vertex array indices. See **glDrawElements**.

`GL_MAX_ELEMENTS_VERTICES`
: *params* returns one value, the recommended maximum number of vertex array vertices. See **glDrawArrays** and **glDrawElements**.

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

`GL_MAX_MODELVIEW_STACK_DEPTH`
: *params* returns one value, the maximum supported depth of the modelview matrix stack. The value must be at least 16. See **glPushMatrix**.

`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. The value must be at least 64. See **glTexImage2D**, **glCompressedTexImage2D**, and **glCopyTexImage2D**.

`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_TEXTURE_UNITS`
: *params* returns a single value indicating the number of texture units supported. The value must be at least 1. See **glActiveTexture**, **glClientActiveTexture** and **glMultiTexCoord**.

`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_NUM_COMPRESSED_TEXTURE_FORMATS`
: *params* returns one value, the number of supportex compressed texture formats. The value must be at least 10. See **glCompressedTexImage2D** and **glCompressedTexSubImage2D**.

`GL_RED_BITS`
: *params* returns one value, the number of red bitplanes in each color buffer.

`GL_SMOOTH_LINE_WIDTH_RANGE`
: *params* returns two values, the smallest and largest supported widths for antialiased lines. The range must include 1. See **glLineWidth**.

`GL_SMOOTH_POINT_SIZE_RANGE`
: *params* returns two values, the smallest and largest supported widths for antialiased points. The range must include 1. See **glPointSize**.

`GL_STENCIL_BITS`
: *params* returns one value, the number of bitplanes in the stencil buffer.

`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 value must be at least 4.

Errors

`GL_INVALID_ENUM` is generated if *pname* is not an accepted value.

See Also

glGetError, **glGetString**

Name

glGetString – return a string describing the current GL connection

C Specification

`const GLubyte *` **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 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 `NULL`.

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.

Name

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`, and `GL_POINT_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 antialiased 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 antialiased points. If a larger filter function is applied, hinting `GL_NICEST` can result in more pixel fragments being generated during rasterization,

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.

Name

glIntro – introduction to OpenGL ES

Overview

OpenGL ES (GL) is a 3D-oriented renderer for embedded systems.

See **eglIntro** for a short example program.

OpenGL Extensions

The following extensions currently exist. Note that the set of supported extensions depends on the implementation.

`OES_query_matrix`
: provides a way to query the values of the current matrix. Each matrix value is returned as a mantissa/exponent pair. If the implementation supports tracking of invalid values, they are indicated in the returned status value.

Using OpenGL ES Extensions

Function names and tokens for OpenGL ES extensions are suffixed with OES or with a vendor-specfic acronym. OES is used for extensions that have been reviewed by the Khronos Group and might be supported by more than one OpenGL ES vendor.

All supported extensions have an associated macro definition in `gl.h` and a corresponding token in the extensions string returned by **glGetString**. For example, if the `OES_query_matrix` extension is supported, then the token `GL_OES_query_matrix` will be defined in `gl.h` and `GL_OES_query_matrix` will appear in the extensions string returned by **glGetString**.

The definitions in `gl.h` can be used at compile time to determine if an extension's tokens and procedures exist in the OpenGL ES library. However, the tokens returned by **glGetString** must be consulted at runtime to determine whether the extension is supported on the particular context in use at that moment.

Files

`GLES/gl.h`

OpenGL ES header file

See Also

eglIntro, **glGetString**

Name

glLightf, **glLightx**, **glLightfv**, **glLightxv** – set light source parameters

C Specification

`void` **glLightf**`(GLenum` *light*`, GLenum` *pname*`, GLfloat` *param*`)`
`void` **glLightx**`(GLenum` *light*`, GLenum` *pname*`, GLfixed` *param*`)`

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 $0 \leq i <$ `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.

C Specification

`void` **glLightfv**`(GLenum` *light*`, GLenum` *pname*`, const GLfloat *` *params*`)`
`void` **glLightxv**`(GLenum` *light*`, GLenum` *pname*`, const GLfixed *` *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 $0 \leq i <$ `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 $0 \le i <$ 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 fixed-point or floating-point values that specify the ambient RGBA intensity of the light. Both fixed-point and floating-point values are mapped directly. Neither fixed-point nor floating-point values are clamped. The initial ambient light intensity is (0, 0, 0, 1).

GL_DIFFUSE

params contains four fixed-point or floating-point values that specify the diffuse RGBA intensity of the light. Both fixed-point and floating-point values are mapped directly. Neither fixed-point 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 fixed-point or floating-point values that specify the specular RGBA intensity of the light. Both fixed-point and floating-point values are mapped directly. Neither fixed-point 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 fixed-point or floating-point values that specify the position of the light in homogeneous object coordinates. Both fixed-point and floating-point values are mapped directly. Neither fixed-point 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 fixed-point or floating-point values that specify the direction of the light in homogeneous object coordinates. Both fixed-point and floating-point values are mapped directly. Neither fixed-point 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 fixed-point or floating-point value that specifies the intensity distribution of the light. Fixed-point 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 fixed-point or floating-point value that specifies the maximum spread angle of a light source. Fixed-point 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 fixed-point or floating-point value that specifies one of the three light attenuation factors. Fixed-point 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.

See Also

glEnable, **glLightModel**, **glMaterial**

Name

glLightModelf, **glLightModelx**, **glLightModelfv**, **glLightModelxv** – set the lighting model parameters

C Specification

void **glLightModelf**(GLenum *pname*, GLfloat *param*)
void **glLightModelx**(GLenum *pname*, GLfixed *param*)

Parameters

pname	Specifies a single-valued lighting model parameter. Must be `GL_LIGHT_MODEL_TWO_SIDE`.
param	Specifies the value that *param* will be set to.

C Specification

void **glLightModelfv**(GLenum *pname*, const GLfloat * *params*)
void **glLightModelxv**(GLenum *pname*, const GLfixed * *params*)

Parameters

pname	Specifies a lighting model parameter. `GL_LIGHT_MODEL_AMBIENT` 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 two lighting model parameters:

`GL_LIGHT_MODEL_AMBIENT`
: *params* contains four fixed-point or floating-point values that specify the ambient intensity of the entire scene. The values are not clamped. The initial value is (0.2, 0.2, 0.2, 1.0).

`GL_LIGHT_MODEL_TWO_SIDE`
: *params* is a single fixed-point 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, 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.

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.

Errors

`GL_INVALID_ENUM` is generated if *pname* is not an accepted value.

See Also

glLight, **glMaterial**

Name

glLineWidth, **glLineWidthx** – specify the width of rasterized lines

C Specification

`void` **glLineWidth**`(GLfloat` *width*`)`
`void` **glLineWidthx**`(GLfixed` *width*`)`

Parameters

width Specifies the width of rasterized lines. The initial value is 1.

Description

glLineWidth specifies the rasterized width of both aliased and antialiased lines. Using a line width other than 1 has different effects, depending on whether line antialiasing is enabled. To enable and disable line antialiasing, call **glEnable** and **glDisable** with argument `GL_LINE_SMOOTH`. Line antialiasing is initially disabled.

If line antialiasing is disabled, the actual width is determined by rounding the supplied width to the nearest integer. (If the rounding results in the value 0, it is as if the line width were 1.) If $|\Delta x| \geq |\Delta y|$, i pixels are filled in each column that is rasterized, where i is the rounded value of *width*. Otherwise, i pixels are filled in each row that is rasterized.

If antialiasing is enabled, line rasterization produces a fragment for each pixel square that intersects the region lying within the rectangle having width equal to the current line width, length equal to the actual length of the line, and centered on the mathematical line segment. The coverage value for each fragment is the window coordinate area of the intersection of the rectangular region with the corresponding pixel square. This value is saved and used in the final rasterization step.

Not all widths can be supported when line antialiasing is enabled. If an unsupported width is requested, the nearest supported width is used. Only width 1 is guaranteed to be supported; others depend on the implementation. Likewise, there is a range for aliased line widths as well. To query the range of supported widths and the size difference between supported widths within the range, call **glGetInteger** with arguments `GL_ALIASED_LINE_WIDTH_RANGE`, `GL_SMOOTH_LINE_WIDTH_RANGE`, `GL_SMOOTH_LINE_WIDTH_GRANULARITY`.

Notes

Nonantialiased line width may be clamped to an implementation-dependent maximum. Call **glGetInteger** with `GL_ALIASED_LINE_WIDTH_RANGE` to determine the maximum width.

Errors

`GL_INVALID_VALUE` is generated if *width* is less than or equal to 0.

Associated Gets

glGetInteger with argument `GL_ALIASED_LINE_WIDTH_RANGE`

glGetInteger with argument `GL_SMOOTH_LINE_WIDTH_RANGE`

See Also

glEnable, **glGetInteger**

Name

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

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

but in some cases it is more efficient.

See Also

glLoadMatrix, **glMatrixMode**, **glMultMatrix**, **glPushMatrix**

Name

glLoadMatrixf, **glLoadMatrixx** – replace the current matrix with the specified matrix

C Specification

void **glLoadMatrixf**(const GLfloat * *m*)
void **glLoadMatrixx**(const GLfixed * *m*)

Parameters

m	Specifies a pointer to 16 consecutive values, which are used as the elements of a 4×4 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 fixed-point or single-precision floating-point values $m\,[0]\,, m\,[1]\,, \ldots m\,[15]$, then the modelview transformation $M(v)$ does the following:

$$M(v) = \begin{pmatrix} m\,[0] & m\,[4] & m\,[8] & m\,[12] \\ m\,[1] & m\,[5] & m\,[9] & m\,[13] \\ m\,[2] & m\,[6] & m\,[10] & m\,[14] \\ m\,[3] & m\,[7] & m\,[11] & m\,[15] \end{pmatrix} \times \begin{pmatrix} v\,[0] \\ v\,[1] \\ v\,[2] \\ v\,[3] \end{pmatrix}$$

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.

See Also

glLoadIdentity, **glMatrixMode**, **glMultMatrix**, **glPushMatrix**

Name

glLogicOp – specify a logical pixel operation

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 and the color at the corresponding location in the frame buffer. To enable or disable the logical operation, call **glEnable** and **glDisable** with argument `GL_COLOR_LOGIC_OP`. Logical operation is initially disabled.

Opcode	**Resulting Operation**
`GL_CLEAR`	0
`GL_SET`	1
`GL_COPY`	s
`GL_COPY_INVERTED`	$\tilde{}s$
`GL_NOOP`	d
`GL_INVERT`	$\tilde{}d$
`GL_AND`	s & d
`GL_NAND`	$\tilde{}(s$ & $d)$
`GL_OR`	$s \mid d$
`GL_NOR`	$\tilde{}(s \mid d)$
`GL_XOR`	s ^ d
`GL_EQUIV`	$\tilde{}(s$ ^ $d)$
`GL_AND_REVERSE`	s & $\tilde{}d$
`GL_AND_INVERTED`	$\tilde{}s$ & d
`GL_OR_REVERSE`	$s \mid \tilde{}d$
`GL_OR_INVERTED`	$\tilde{}s \mid d$

opcode is a symbolic constant chosen from the list above. In the explanation of the logical operations, *s* represents the incoming color and *d* represents the color 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.

Errors

`GL_INVALID_ENUM` is generated if *opcode* is not an accepted value.

See Also

glAlphaFunc, **glBlendFunc**, **glEnable**, **glStencilOp**

Name

glMaterialf, **glMaterialx**, **glMaterialfv**, **glMaterialxv** – specify material parameters for the lighting model

C Specification

```
void glMaterialf(GLenum face, GLenum pname, GLfloat param)
void glMaterialx(GLenum face, GLenum pname, GLfixed param)
```

Parameters

face Specifies which face or faces are being updated. Must be `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.

C Specification

```
void glMaterialfv(GLenum face,
                  GLenum pname,
                  const GLfloat * params)

void glMaterialxv(GLenum face,
                  GLenum pname,
                  const GLfixed * params)
```

Parameters

face Specifies which face or faces are being updated. Must be `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`, or `GL_AMBIENT_AND_DIFFUSE`.

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, 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*, must be `GL_FRONT_AND_BACK` and specifies that both 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 fixed-point or floating-point values that specify the ambient RGBA reflectance of the material. The values are not clamped. The initial ambient reflectance is (0.2, 0.2, 0.2, 1.0).

`GL_DIFFUSE`
params contains four fixed-point or floating-point values that specify the diffuse RGBA reflectance of the material. The values are not clamped. The initial diffuse reflectance is (0.8, 0.8, 0.8, 1.0).

`GL_SPECULAR`
params contains four fixed-point or floating-point values that specify the specular RGBA reflectance of the material. The values are not clamped. The initial specular reflectance is (0, 0, 0, 1).

`GL_EMISSION`
: *params* contains four fixed-point or floating-point values that specify the RGBA emitted light intensity of the material. The values are not clamped. The initial emission intensity is (0, 0, 0, 1).

`GL_SHININESS`
: *params* is a single fixed-point or floating-point value that specifies the RGBA specular exponent of the material. Only values in the range [0, 128] are accepted. The initial specular exponent 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`.

Notes

To change the diffuse and ambient material per vertex, color material can be used. To enable and disable `GL_COLOR_MATERIAL`, call **glEnable** and **glDisable** with argument `GL_COLOR_MATERIAL`. Color material is initially disabled.

While the ambient, diffuse, specular and emission material parameters all have alpha components, only the diffuse alpha component is used in the lighting computation.

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.

See Also

glEnable, **glLight**, **glLightModel**

Name

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 four 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.

Errors

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

See Also

glLoadMatrix, **glPushMatrix**

Name

glMultMatrixf, **glMultMatrixx** – multiply the current matrix with the specified matrix

C Specification

`void` **glMultMatrixf**`(const GLfloat *` *m*`)`
`void` **glMultMatrixx**`(const GLfixed *` *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

$$\begin{pmatrix} c[0] & c[4] & c[8] & c[12] \\ c[1] & c[5] & c[9] & c[13] \\ c[2] & c[6] & c[10] & c[14] \\ c[3] & c[7] & c[11] & c[15] \end{pmatrix} \times \begin{pmatrix} v[0] \\ v[1] \\ v[2] \\ v[3] \end{pmatrix}$$

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

$$\begin{pmatrix} c[0] & c[4] & c[8] & c[12] \\ c[1] & c[5] & c[9] & c[13] \\ c[2] & c[6] & c[10] & c[14] \\ c[3] & c[7] & c[11] & c[15] \end{pmatrix} \times \begin{pmatrix} m[0] & m[4] & m[8] & m[12] \\ m[1] & m[5] & m[9] & m[13] \\ m[2] & m[6] & m[10] & m[14] \\ m[3] & m[7] & m[11] & m[15] \end{pmatrix} \times \begin{pmatrix} v[0] \\ v[1] \\ v[2] \\ v[3] \end{pmatrix}$$

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.

See Also

glLoadIdentity, **glLoadMatrix**, **glMatrixMode**, **glPushMatrix**

Name

glMultiTexCoord4f, **glMultiTexCoord4x** – set the current texture coordinates

C Specification

```
void glMultiTexCoord4f(GLenum target,
                       GLfloat s,
                       GLfloat t,
                       GLfloat r,
                       GLfloat q)
```

```
void glMultiTexCoord4x(GLenum target,
                       GLfixed s,
                       GLfixed t,
                       GLfixed r,
                       GLfixed q)
```

Parameters

target	Specifies texture unit whose coordinates should be modified. The number of texture units is implementation dependent, but must be at least one. Must be one of `GL_TEXTURE`i, where $0 \leq i <$ `GL_MAX_TEXTURE_UNITS`, which is an implementation-dependent value.
s, t, r, q	Specify s, t, r, and q texture coordinates for *target* texture unit. The initial value is (0, 0, 0, 1).

Description

glMultiTexCoord specifies the four texture coordinates as $(s,\ t,\ r,\ q)$.

The current texture coordinates are part of the data that is associated with each vertex.

Notes

It is always the case that `GL_TEXTURE`i = `GL_TEXTURE0` + i.

Associated Gets

glGetInteger with argument `GL_MAX_TEXTURE_UNITS`

See Also

glActiveTexture, **glClientActiveTexture**, **glColor**, **glGetInteger**, **glNormal**, **glTexCoordPointer**

Name

glNormal3f, **glNormal3x** – set the current normal vector

C Specification

`void` **glNormal3f**`(GLfloat` *nx*`, GLfloat` *ny*`, GLfloat` *nz*`)`
`void` **glNormal3x**`(GLfixed` *nx*`, GLfixed` *ny*`, GLfixed` *nz*`)`

Parameters

nx, *ny*, *nz*

Specify the x, y, and z coordinates of the new current normal. The initial value is (0, 0, 1).

Description

The current normal is set to the given coordinates whenever **glNormal** is issued. Byte, short, or integer arguments are converted to floating-point 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 `GL_NORMALIZE` is enabled, then normals of any length specified with **glNormal** are normalized after transformation. If `GL_RESCALE_NORMAL` is enabled, normals are scaled by a scaling factor derived from the modelview matrix. `GL_RESCALE_NORMAL` requires that the originally specified normals were of unit length, and that the modelview matrix contain only uniform scales for proper results. To enable and disable normalization, call **glEnable** and **glDisable** with either `GL_NORMALIZE` or `GL_RESCALE_NORMAL`. Normalization is initially disabled.

See Also

glColor, **glEnable**, **glMultiTexCoord**, **glNormalPointer**

Name

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`, and `GL_FIXED` are accepted. However, the initial value is `GL_FLOAT`.

The common profile accepts the symbolic constant `GL_FLOAT` as well.

stride Specifies the byte offset between consecutive normals. If *stride* is 0, the normals 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 normal in the array. The initial value is 0.

Description

glNormalPointer specifies the location and data 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 vertices and attributes to be packed into a single array or stored in separate arrays. (Single-array storage may be more efficient on some implementations.) When a normal array is specified, *type* , *stride* , and *pointer* are saved as client-side state.

If the normal array is enabled, it is used when **glDrawArrays** or **glDrawElements** is called. To enable and disable the normal array, call **glEnableClientState** and **glDisableClientState** with the argument `GL_NORMAL_ARRAY`. The normal array is initially disabled and isn't accessed when **glDrawArrays** or **glDrawElements** is called.

Use **glDrawArrays** to construct a sequence of primitives (all of the same type) from prespecified vertex and vertex attribute arrays. Use **glDrawElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

Notes

glNormalPointer is typically implemented on the client side.

Errors

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

`GL_INVALID_VALUE` is generated if *stride* is negative.

See Also

glColorPointer, **glDrawArrays**, **glDrawElements**, **glEnable**, **glTexCoordPointer**, **glVertexPointer**

Name

glOrthof, **glOrthox** – multiply the current matrix with an orthographic matrix

C Specification

```
void glOrthof(GLfloat left,
              GLfloat right,
              GLfloat bottom,
              GLfloat top,
              GLfloat near,
              GLfloat far)
```

```
void glOrthox(GLfixed left,
              GLfixed right,
              GLfixed bottom,
              GLfixed top,
              GLfixed near,
              GLfixed far)
```

Parameters

left, *right*
: Specify the coordinates for the left and right vertical clipping planes.

bottom, *top*
: Specify the coordinates for the bottom and top horizontal clipping planes.

near, *far*
: Specify the distances to the nearer and farther depth clipping planes. These 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:

$$\begin{pmatrix} \frac{2}{right-left} & 0 & 0 & t_x \\ 0 & \frac{2}{top-bottom} & 0 & t_y \\ 0 & 0 & \frac{-2}{far-near} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where

$$\begin{aligned} t_x &= -\frac{right+left}{right-left} \\ t_y &= -\frac{top+bottom}{top-bottom} \\ t_z &= -\frac{far+near}{far-near} \end{aligned}$$

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

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

See Also

glFrustum, **glMatrixMode**, **glMultMatrix**, **glPushMatrix**, **glViewport**

Name

glPixelStorei – set pixel storage modes

C Specification

`void` **glPixelStorei**`(GLenum` *pname*`, GLint` *param*`)`

Parameters

pname Specifies the symbolic name of the parameter to be set. `GL_PACK_ALIGNMENT` affects the packing of pixel data into memory. `GL_UNPACK_ALIGNMENT` affects the unpacking of pixel data *from* memory.

param Specifies the value that *pname* is set to.

Description

glPixelStore sets pixel storage modes that affect the operation of subsequent **glReadPixels** as well as the unpacking of **glTexImage2D**, and **glTexSubImage2D**.

pname is a symbolic constant indicating the parameter to be set, and *param* is the new value. The following storage parameter affects how pixel data is returned to client memory. This value is significant for **glReadPixels**:

`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 initial value is 4.

The following storage parameter affects how pixel data is read from client memory. This value is significant for **glTexImage2D** and **glTexSubImage2D**:

`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 initial value is 4.

Notes

Pixel storage modes are client states.

glCompressedTexImage2D and **glCompressedTexSubImage2D** are not affected by **glPixelStore**.

Errors

`GL_INVALID_ENUM` is generated if *pname* is not an accepted value.

`GL_INVALID_VALUE` is generated if alignment is specified as other than 1, 2, 4, or 8.

See Also

glReadPixels, **glCompressedTexImage2D**, **glCompressedTexSubImage2D**, **glTexImage2D**, **glTexSubImage2D**

Name

glPointSize, **glPointSizex** – specify the diameter of rasterized points

C Specification

`void` **glPointSize**`(GLfloat` *size*`)`
`void` **glPointSizex**`(GLfixed` *size*`)`

Parameters

size	Specifies the diameter of rasterized points. The initial value is 1.

Description

glPointSize specifies the rasterized diameter of both aliased and antialiased points. Using a point size other than 1 has different effects, depending on whether point antialiasing is enabled. To enable and disable point antialiasing, call **glEnable** and **glDisable** with argument `GL_POINT_SMOOTH`. Point antialiasing is initially disabled.

If point antialiasing is disabled, the actual size is determined by rounding the supplied size to the nearest integer. (If the rounding results in the value 0, it is as if the point size were 1.) If the rounded size is odd, then the center point (x, y) of the pixel fragment that represents the point is computed as

$$(\lfloor x_w \rfloor + \frac{1}{2}, \lfloor y_w \rfloor + \frac{1}{2})$$

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

$$(\lfloor x_w + \frac{1}{2} \rfloor, \lfloor y_w + \frac{1}{2} \rfloor)$$

and the rasterized fragment's centers are the half-integer window coordinates within the square of the rounded size centered at (x, y). All pixel fragments produced in rasterizing a nonantialiased point are assigned the same associated data, that of the vertex corresponding to the point.

If antialiasing is enabled, then point rasterization produces a fragment for each pixel square that intersects the region lying within the circle having diameter equal to the current point size and centered at the point's (x_w, y_w). The coverage value for each fragment is the window coordinate area of the intersection of the circular region with the corresponding pixel square. This value is saved and used in the final rasterization step. The data associated with each fragment is the data associated with the point being rasterized.

Not all sizes are supported when point antialiasing is enabled. If an unsupported size is requested, the nearest supported size is used. Only size 1 is guaranteed to be supported; others depend on the implementation. To query the range of supported sizes, call **glGetInteger** with the argument `GL_SMOOTH_POINT_SIZE_RANGE`. For aliased points, query the supported ranges **glGetInteger** with the argument `GL_ALIASED_POINT_SIZE_RANGE`.

Notes

A non-antialiased point size may be clamped to an implementation-dependent maximum. Although this maximum cannot be queried, it must be no less than the maximum value for antialiased points, rounded to the nearest integer value.

Errors

`GL_INVALID_VALUE` is generated if *size* is less than or equal to 0.

Associated Gets

glGetInteger with argument `GL_ALIASED_POINT_SIZE_RANGE`

glGetInteger with argument `GL_SMOOTH_POINT_SIZE_RANGE`

See Also

glEnable, **glGetInteger**

Name

glPolygonOffset, **glPolygonOffsetx** – set the scale and units used to calculate depth values

C Specification

`void` **glPolygonOffset**`(GLfloat` *factor*`, GLfloat` *units*`)`
`void` **glPolygonOffsetx**`(GLfixed` *factor*`, GLfixed` *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_FILL` 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 $m * factor + r * units$, where m 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 for applying decals to surfaces.

See Also

glDepthFunc, **glEnable**

Name

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 16. In the other 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.

Notes

Each texture unit has its own texture matrix stack. Use **glActiveTexture** to select the desired texture matrix stack.

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.

Associated Gets

glGetInteger with argument `GL_MAX_MODELVIEW_STACK_DEPTH`

glGetInteger with argument `GL_MAX_PROJECTION_STACK_DEPTH`

glGetInteger with argument `GL_MAX_TEXTURE_STACK_DEPTH`

glGetInteger with argument `GL_MAX_TEXTURE_UNITS`

See Also

glActiveTexture, **glFrustum**, **glGetInteger**, **glLoadIdentity**, **glLoadMatrix**, **glMatrixMode**, **glMultMatrix**, **glOrtho**, **glRotate**, **glScale**, **glTranslate**, **glViewport**

Name

glQueryMatrixxOES – return the values of the current matrix

C Specification

```
GLbitfield glQueryMatrixxOES(GLfixed * mantissa,
                             GLint * exponent)
```

Parameters

mantissa	Returns the mantissi of the current matrix.
exponent	Returns the exponents of the current matrix.

Description

glQueryMatrixxOES returns the values of the current matrix. *mantissa* returns the 16 mantissa values of the current matrix, and *exponent* returns the correspnding 16 exponent values. The matrix value i is then close to $mantissa[i] * 2^{exponent[i]}$.

Use **glMatrixMode** and **glActiveTexture**, to select the desired matrix to return.

If all are valid (not NaN or Inf), **glQueryMatrixxOES** returns the status value 0. Otherwise, for every component i which is not valid, the ith bit is set.

Notes

glQueryMatrixxOES is available only if the `GL_OES_query_matrix` extension is supported by your implementation.

The implementation is not required to keep track of overflows. If overflows are not tracked, the returned status value is always 0.

Associated Gets

glGetString with argument `GL_EXTENSIONS`

See Also

glActiveTexture, **glGetString**, **glMatrixMode**

Name

glReadPixels – read a block of pixels from the color buffer

C Specification

```
void glReadPixels(GLint y,
                  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 color 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. Must be either `GL_RGBA` or the value of `GL_IMPLEMENTATION_COLOR_READ_FORMAT_OES`.

type Specifies the data type of the pixel data. Must be either `GL_UNSIGNED_BYTE` or the value of `GL_IMPLEMENTATION_COLOR_READ_TYPE_OES`.

pixels Returns the pixel data.

Description

glReadPixels returns pixel data from the color buffer, starting with the pixel whose lower left corner is at location (x, y), into client memory starting at location *pixels*. The processing of the pixel data before it is placed into client memory can be controlled with **glPixelStore**.

glReadPixels returns values from each pixel with lower left corner at $(x+i, y+j)$ for $0 \leq i < width$ and $0 \leq j < height$. This pixel is said to be the ith pixel in the jth row. Pixels are returned in row order from the lowest to the highest row, left to right in each row.

format specifies the format of the returned pixel values. `GL_RGBA` is always accepted, the value of `GL_IMPLEMENTATION_COLOR_READ_FORMAT_OES` may allow another format:

`GL_RGBA` Each color component is converted to floating point such that zero intensity maps to 0 and full intensity maps to 1.

`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.

`GL_LUMINANCE`
Each element is a single luminance value. The GL converts it to floating point and assembles it into an RGBA element by replicating the luminance value three times for red, green and blue and attaching 1 for alpha.

`GL_LUMINANCE_ALPHA`
Each element is a luminance/alpha pair. The GL converts it to floating point and assembles it into an RGBA element by replicating the luminance value three times for red, green and blue.

`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.

Unneeded data is then discarded. For example, `GL_ALPHA` discards the red, green, and blue 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].

Finally, the components are converted to the proper, as specified by *type* where each component is multiplied by $2^n - 1$, where n is the number of bits per component.

Return values are placed in memory as follows. If *format* is `GL_ALPHA`, or `GL_LUMINANCE`, a single value is returned and the data for the ith pixel in the jth row is placed in location $j * width + i$. 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 parameter GL_PACK_ALIGNMENT set by **glPixelStore**, affects 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* is not GL_RGBA or the value of GL_IMPLEMENTATION_COLOR_READ_FORMAT_OES.

GL_INVALID_ENUM is generated if *type* is not GL_UNSIGNED_BYTE or the value of GL_IMPLEMENTATION_COLOR_READ_TYPE_OES.

GL_INVALID_VALUE is generated if either *width* or *height* is negative.

GL_INVALID_OPERATION is generated if *format* and *type* are neither (GL_RGBA, GL_UNSIGNED_BYTE) nor the values of (GL_IMPLEMENTATION_COLOR_READ_FORMAT_OES, GL_IMPLEMENTATION_COLOR_READ_TYPE_OES).

Associated Gets

glGetInteger with argument GL_IMPLEMENTATION_COLOR_READ_FORMAT_OES

glGetInteger with argument GL_IMPLEMENTATION_COLOR_READ_TYPE_OES

See Also

glGetInteger, **glPixelStore**

Name

glRotatef, **glRotatex** – multiply the current matrix by a rotation matrix

C Specification

void **glRotatef**(GLfloat *angle*, GLfloat *x*, GLfloat *y*, GLfloat *z*)
void **glRotatex**(GLfixed *angle*, GLfixed *x*, GLfixed *y*, GLfixed *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:

$$\begin{pmatrix} x^2(1-c)+c & xy(1-c)-zs & xz(1-c)+ys & 0 \\ xy(1-c)+zs & y^2(1-c)+c & yz(1-c)-xs & 0 \\ xz(1-c)-ys & yz(1-c)+xs & z^2(1-c)+c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

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.

See Also

glMatrixMode, **glMultMatrix**, **glPushMatrix**, **glScale**, **glTranslate**

Name

glSampleCoverage, **glSampleCoveragex** – specify mask to modify multisampled pixel fragments

C Specification

`void` **glSampleCoverage**`(GLclampf` *value*`, GLboolean` *invert*`)`
`void` **glSampleCoveragex**`(GLclampx` *value*`, GLboolean` *invert*`)`

Parameters

value	Specifies the coverage of the modification mask. The value is clamped to the range [0, 1], where 0 represents no coverage and 1 full coverage. The initial value is 1.
invert	Specifies whether the modification mask implied by *value* is inverted or not. The initial value is `GL_FALSE`.

Description

glSampleCoverage defines a mask to modify the coverage of multisampled pixel fragments. This capability is used for antialiased screen-door transparency and smooth transitions between two renderings of an object (often for level-of-detail management in simulation systems).

When multisampling is enabled (see **glEnable** with argument `GL_MULTISAMPLE`) a “fragment mask” is computed for each fragment generated by a primitive. This mask reflects the amount of the pixel covered by the fragment, and determines the frame buffer samples that may be affected by the fragment.

If conversion of alpha values to masks is enabled (**glEnable** with argument `GL_SAMPLE_ALPHA_TO_MASK`), the fragment alpha value is used to generate a temporary modification mask which is then ANDed with the fragment mask. One way to interpret this is as a form of dithering: a multivalued alpha (coverage or opacity) for the whole fragment is converted to simple binary values of coverage at many locations (the samples).

After conversion of alpha values to masks, if replacement of alpha values is enabled (**glEnable** with argument `GL_SAMPLE_ALPHA_TO_ONE`), the fragment's alpha is set to the maximum allowable value.

Finally, if fragment mask modification is enabled (**glEnable** with argument `GL_SAMPLE_MASK`), **glSampleCoverage** defines an additional modification mask. value is used to generate a modification mask in much the same way alpha was used above. If invert is `GL_TRUE`, then the modification mask specified by value will be inverted. The final modification mask will then be ANDed with the fragment mask resulting from the previous steps. This can be viewed as an "override" control that selectively fades the effects of multisampled fragments.

Note that **glSampleCoverage**(value, `GL_TRUE`) is not necessarily equivalent to **glSampleCoverage**(1.0 - value, `GL_FALSE`); due to round-off and other issues, complementing the coverage will not necessarily yield an inverted modification mask.

See Also

glEnable

Name

glScalef, **glScalex** – multiply the current matrix by a general scaling matrix

C Specification

`void` **glScalef**`(GLfloat` *x*`, GLfloat` *y*`, GLfloat` *z*`)`
`void` **glScalex**`(GLfixed` *x*`, GLfixed` *y*`, GLfixed` *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:

$$\begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

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`.

See Also

glEnable, **glMatrixMode**, **glMultMatrix**, **glPushMatrix**, **glRotate**, **glTranslate**

Name

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, in pixels. The initial value is (0, 0).

width, *height*
Specify the width and height of the scissor box. When a GL context is first attached to a surface (e.g. window), *width* and *height* are set to the dimensions of that surface.

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 scissor test is initially disabled. While scissor test is enabled, only pixels that lie within the scissor box can be modified by drawing commands. Window coordinates have integer values at the shared corners of frame buffer pixels. **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.

See Also

glEnable, **glViewport**

Name

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 at the beginning of the vertex array 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 vertex $i + 2$, which is the last vertex to specify the polygon.

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`.

See Also

glColor, **glColorPointer**, **glLight**, **glLightModel**, **glMaterial**

Name

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 1's.

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 stencil test, call **glEnable** and **glDisable** with argument `GL_STENCIL_TEST`. Stencil test is initially disabled. 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 ANDed values participating in the comparison.

If *stencil* represents the value stored in the corresponding stencil buffer location, the following list shows the effect of each comparison function that can be specified by *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 $(ref \ \& \ mask) \leq (stencil \ \& \ mask)$.

`GL_GREATER`
: Passes if $(ref \ \& \ mask) > (stencil \ \& \ mask)$.

`GL_GEQUAL`
: Passes if $(ref \ \& \ mask) \geq (stencil \ \& \ mask)$.

`GL_EQUAL`
: Passes if $(ref \ \& \ mask) = (stencil \ \& \ mask)$.

`GL_NOTEQUAL`
: Passes if $(ref \ \& \ mask) \neq (stencil \ \& \ mask)$.

`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.

Associated Gets

glGetInteger with argument `GL_STENCIL_BITS`

See Also

glAlphaFunc, **glBlendFunc**, **glDepthFunc**, **glEnable**, **glGetInteger**, **glLogicOp**, **glStencilOp**

Name

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. The initial value is all 1's.

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.

Associated Gets

glGetInteger with argument `GL_STENCIL_BITS`

See Also

glClear, **glColorMask**, **glDepthMask**, **glGetInteger**, **glStencilFunc**, **glStencilOp**

Name

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 stencil test, call **glEnable** and **glDisable** with argument `GL_STENCIL_TEST`. To control it, call **glStencilFunc**. Stenciling is initially disabled.

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

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.

Associated Gets

glGetInteger with argument `GL_STENCIL_BITS`

See Also

glAlphaFunc, **glBlendFunc**, **glDepthFunc**, **glEnable**, **glGetInteger**, **glLogicOp**, **glStencilFunc**

Name

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 2, 3 or 4. The initial value is 4.

type Specifies the data type of each texture coordinate. Symbolic constants `GL_BYTE`, `GL_SHORT`, and `GL_FIXED` are accepted. However, the initial value is `GL_FLOAT`.

The common profile accepts the symbolic constant `GL_FLOAT` as well.

stride Specifies the byte offset between consecutive array elements. If *stride* is 0, the array elements are understood to be tightly packed. The initial value is 0.

pointer Specifies a pointer to the first coordinate of the first element in the array. The initial value is 0.

Description

glTexCoordPointer specifies the location and data of an array of texture coordinates to use when rendering. *size* specifies the number of coordinates per element, and must be 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 vertices and attributes to be packed into a single array or stored in separate arrays. (Single-array storage may be more efficient on some implementations.)

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

If the texture coordinate array is enabled, it is used when **glDrawArrays**, or **glDrawElements** is called. To enable and disable the texture coordinate array for the client-side active texture unit, call **glEnableClientState** and **glDisableClientState** with the argument `GL_TEXTURE_COORD_ARRAY`. The texture coordinate array is initially disabled for all client-side active texture units and isn't accessed when **glDrawArrays** or **glDrawElements** is called.

Use **glDrawArrays** to construct a sequence of primitives (all of the same type) from prespecified vertex and vertex attribute arrays. Use **glDrawElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

Notes

glTexCoordPointer is typically implemented on the client side.

glTexCoordPointer updates the texture coordinate array state of the client-side active texture unit, specified with **glClientActiveTexture**.

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.

See Also

glClientActiveTexture, **glColorPointer**, **glDrawArrays**, **glDrawElements**, **glEnable**, **glMultiTexCoord**, **glNormalPointer**, **glVertexPointer**

Name

glTexEnvf, **glTexEnvx**, **glTexEnvfv**, **glTexEnvxv** – set texture environment parameters

C Specification

`void` **glTexEnvf**`(GLenum` *pname*`, GLfloat` *param*`)`
`void` **glTexEnvx**`(GLenum` *pname*`, GLfixed` *param*`)`

Parameters

target	Specifies a texture environment. Must be `GL_TEXTURE_ENV`.
pname	Specifies the symbolic name of a single-valued texture environment parameter. Must be `GL_TEXTURE_ENV_MODE`.
param	Specifies a single symbolic constant, one of `GL_MODULATE`, `GL_DECAL`, `GL_BLEND`, or `GL_REPLACE`.

C Specification

`void` **glTexEnvfv**`(GLenum` *pname*`, const GLfloat *` *params*`)`
`void` **glTexEnvxv**`(GLenum` *pname*`, const GLfixed *` *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` and `GL_TEXTURE_ENV_COLOR`.
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` or `GL_TEXTURE_ENV_COLOR`.

If *pname* is `GL_TEXTURE_ENV_MODE`, then *params* is (or points to) the symbolic name of a texture function. Four texture functions may be specified: `GL_MODULATE`, `GL_DECAL`, `GL_BLEND`, and `GL_REPLACE`.

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 **glTexImage2D**, 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	**Texture functions**	
format	`GL_MODULATE`	`GL_DECAL`
`GL_ALPHA`	$C_v = C_f$ $A_v = A_t A_f$	undefined
`GL_LUMINANCE`	$C_v = L_t C_f$ $A_v = A_f$	undefined
`GL_LUMINANCE_ALPHA`	$C_v = L_t C_f$ $A_v = A_t A_f$	undefined
`GL_RGB`	$C_v = C_t C_f$ $A_v = A_f$	$C_v = C_t$ $A_v = A_f$
`GL_RGBA`	$C_v = C_t C_f$ $A_v = A_t A_f$	$C_v = (1 - A_t)C_f + A_t C_t$ $A_v = A_f$

Base internal format	**Texture functions** GL_BLEND	GL_REPLACE
GL_ALPHA	$C_v = C_f$ $A_v = A_t A_f$	$C_v = C_f$ $A_v = A_t$
GL_LUMINANCE	$C_v = (1 - L_t)C_f + L_t C_c$ $A_v = A_f$	$C_v = L_t$ $A_v = A_f$
GL_LUMINANCE_ALPHA	$C_v = (1 - L_t)C_f + L_t C_c$ $A_v = A_t A_f$	$C_v = L_t$ $A_v = A_t$
GL_RGB	$C_v = (1 - C_t)C_f + C_t C_c$ $A_v = A_f$	$C_v = C_t$ $A_v = A_f$
GL_RGBA	$C_v = (1 - C_t)C_f + C_t C_c$ $A_v = A_t A_f$	$C_v = C_t$ $A_v = A_t$

If *pname* is `GL_TEXTURE_ENV_COLOR`, *params* is a pointer to an array that holds an RGBA color consisting of four values. The values are clamped to the range [0, 1] when they are specified. C_c takes these four values.

The initial value of `GL_TEXTURE_ENV_MODE` is `GL_MODULATE`. The initial value of `GL_TEXTURE_ENV_COLOR` is (0, 0, 0, 0).

Errors

`GL_INVALID_ENUM` is generated when *target* or *pname* is not one of the accepted values, or when *params* should have a defined constant value (based on the value of *pname*) and does not.

See Also

glActiveTexture, **glCompressedTexImage2D**, **glCompressedTexSubImage2D**, **glCopyTexImage2D**, **glCopyTexSubImage2D**, **glTexImage2D**, **glTexParameter**, **glTexSubImage2D**

Name

glTexImage2D – specify a two-dimensional texture image

C Specification

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

Parameters

target Specifies the target texture. Must be `GL_TEXTURE_2D`.

level Specifies the level-of-detail number. Level 0 is the base image level. Level n is the nth mipmap reduction image. Must be greater or equal 0.

internalformat
Specifies the color components in the texture. Must be same as *format*. The following symbolic values are accepted: `GL_ALPHA`, `GL_RGB`, `GL_RGBA`, `GL_LUMINANCE`, or `GL_LUMINANCE_ALPHA`.

width Specifies the width of the texture image. Must be $2^n + 2border$ 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 $2^m + 2border$ 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 0.

format	Specifies the format of the pixel data. Must be same as *internalformat*. The following symbolic values are accepted: `GL_ALPHA`, `GL_RGB`, `GL_RGBA`, `GL_LUMINANCE`, and `GL_LUMINANCE_ALPHA`.
type	Specifies the data type of the pixel data. The following symbolic values are accepted: `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT_5_6_5`, `GL_UNSIGNED_SHORT_4_4_4_4`, and `GL_UNSIGNED_SHORT_5_5_5_1`.
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`. Two-dimensional texturing is initially disabled.

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.

Data is read from *pixels* as a sequence of unsigned bytes or shorts, depending on *type*. These values are grouped into sets of one, two, three, or four values, depending on *format*, to form elements.

When *type* is `GL_UNSIGNED_BYTE`, each of these bytes is interpreted as one color component, depending on *format*. When *type* is one of `GL_UNSIGNED_SHORT_5_6_5`, `GL_UNSIGNED_SHORT_4_4_4_4`, `GL_UNSIGNED_SHORT_5_5_5_1`, each unsigned value is interpreted as containing all the components for a single pixel, with the color components arranged according to format.

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.

By default, adjacent 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.

format determines the composition of each element in *pixels*. It can assume one of the following symbolic values:

`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.

`GL_RGB`
: Each element is an RGB triple. The GL converts it to fixed-point or floating-point and assembles it into an RGBA element by attaching 1 for alpha.

`GL_RGBA`
: Each element contains all four components. The GL converts it to fixed-point or floating-point.

`GL_LUMINANCE`
: Each element is a single luminance value. The GL converts it to fixed-point or 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.

`GL_LUMINANCE_ALPHA`
: Each element is a luminance/alpha pair. The GL converts it to fixed-point or floating point, then assembles it into an RGBA element by replicating the luminance value three times for red, green, and blue.

Notes

pixels may be `NULL`. 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.

glTexImage2D specifies the two-dimensional texture for the currently bound texture specified with **glBindTexture**, and the current texture unit, specified with **glActiveTexture**.

Errors

GL_INVALID_ENUM is generated if *target* is not GL_TEXTURE_2D.

GL_INVALID_ENUM is generated if *format* is not an accepted constant.

GL_INVALID_ENUM is generated if *type* is not a type constant.

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 accepted constant.

GL_INVALID_VALUE is generated if *width* or *height* is less than 0 or greater than GL_MAX_TEXTURE_SIZE, or if either cannot be represented as $2^k + 2border$ for some integer k.

GL_INVALID_VALUE is generated if *border* is not 0.

GL_INVALID_OPERATION is generated if *internalformat* and *format* are not the same.

GL_INVALID_OPERATION is generated if *type* is GL_UNSIGNED_SHORT_5_6_5 and *format* is not GL_RGB.

GL_INVALID_OPERATION is generated if *type*is one of GL_UNSIGNED_SHORT_4_4_4_4, or GL_UNSIGNED_SHORT_5_5_5_1 and *format*is not GL_RGBA.

Associated Gets

glGetInteger with argument GL_MAX_TEXTURE_SIZE

See Also

glActiveTexture, **glBindTexture**, **glCopyTexImage2D**, **glCopyTex-SubImage2D**, **glGetInteger**, **glMatrixMode**, **glPixelStore**, **glTexEnv**, **glTexSubImage2D**, **glTexParameter**

Name

glTexParameterf, **glTexParameterx** – set texture parameters

C Specification

`void` **glTexParameterf**`(GLenum` *target*`, GLenum` *pname*`, GLfloat` *param*`)`
`void` **glTexParameterx**`(GLenum` *target*`, GLenum` *pname*`, GLfixed` *param*`)`

Parameters

target Specifies the target texture, which must be `GL_TEXTURE_2D`.

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`, or `GL_TEXTURE_WRAP_T`.

param Specifies the value of *pname*.

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 *param*to the texture parameter specified as *pname*. *target* defines the target texture, which must be `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. If the texture has dimensions $2^n \times 2^m$, there are $\max(n, m) + 1$ mipmaps. The first mipmap is the original texture, with dimensions $2^n \times 2^m$. Each subsequent mipmap has dimensions $2^{k-1} \times 2^{l-1}$, where $2^k \times 2^l$ 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 **glTexImage2D** or **glCopyTexImage2D** with the *level* argument indicating the order of the mipmaps. Level 0 is the original texture. Level $\max(n, m)$ is the final 1×1 mipmap.

param 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 either GL_CLAMP, GL_CLAMP_TO_EDGE, or GL_REPEAT. GL_CLAMP causes s coordinates to be clamped to the range [0, 1] and is useful for preventing wrapping artifacts when mapping a single image onto an object. GL_CLAMP_TO_EDGE causes s coordinates to be clamped to the range $\left[\frac{1}{2N}, 1 - \frac{1}{2N}\right]$, where N is the size of the texture in the direction of clamping. `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. Border texture elements are accessed only if wrapping is set to `GL_CLAMP`. Initially, `GL_TEXTURE_WRAP_S` is set to `GL_REPEAT`.

`GL_TEXTURE_WRAP_T`

Sets the wrap parameter for texture coordinate t to either `GL_CLAMP`, `GL_CLAMP_TO_EDGE`, or `GL_REPEAT`. See the discussion under `GL_TEXTURE_WRAP_S`. Initially, `GL_TEXTURE_WRAP_T` is set to `GL_REPEAT`.

Notes

Suppose that a program has enabled texturing (by calling **glEnable** with argument `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 **glTexImage2D**, 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.

glTexParameter specifies the texture parameters for the active texture unit, specified by calling **glActiveTexture**.

Errors

`GL_INVALID_ENUM` is generated if *target* or *pname* is not one of the accepted defined values.

`GL_INVALID_ENUM` is generated if *param* should have a defined constant value (based on the value of *pname*) and does not.

See Also

glActiveTexture, **glBindTexture**, **glCopyTexImage2D**, **glCopyTexSubImage2D**, **glEnable**, **glPixelStore**, **glTexEnv**, **glTexImage2D**,

glTexSubImage2D

Name

glTexSubImage2D – specify a two-dimensional texture subimage

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 subimage.
height	Specifies the height of the texture subimage.
format	Specifies the of the pixel data. The following symbolic values are accepted: `GL_ALPHA`, `GL_RGB`, `GL_RGBA`, `GL_LUMINANCE`, and `GL_LUMINANCE_ALPHA`.
type	Specifies the data type of the pixel data. The following symbolic values are accepted: `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT_5_6_5`, `GL_UNSIGNED_SHORT_4_4_4_4`, and `GL_UNSIGNED_SHORT_5_5_5_1`.

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`. Two-dimensional texturing is initially disabled.

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

glPixelStore affects texture images in exactly the way it affects **glTexImage2D**.

glTexSubImage2D specifies a two-dimensional sub texture for the currently bound texture, specified with **glBindTexture** and current texture unit, specified with **glActiveTexture**.

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 $xoffset < -b$, $xoffset + width > (w - b)$, $yoffset < -b$, or $yoffset + height > (h - b)$, where w is the texture width, h is the texture height, and *b* is the border 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 constant.

`GL_INVALID_ENUM` is generated if *type* is not a type constant.

`GL_INVALID_OPERATION` is generated if *type* is `GL_UNSIGNED_SHORT_5_6_5` and *format* is not `GL_RGB`.

`GL_INVALID_OPERATION` is generated if *type* is one of `GL_UNSIGNED_SHORT_4_4_4_4`, or `GL_UNSIGNED_SHORT_5_5_5_1` and *format* is not `GL_RGBA`.

Associated Gets

glGetInteger with argument `GL_MAX_TEXTURE_SIZE`

See Also

glActiveTexture, **glBindTexture**, **glCompressedTexSubImage2D**, **glCopyTexSubImage2D**, **glGetInteger**, **glPixelStore**, **glTexEnv**, **glTexImage2D**, **glTexParameter**

Name

glTranslatef, **glTranslatex** – multiply the current matrix by a translation matrix

C Specification

void **glTranslatef**(GLfloat *x*, GLfloat *y*, GLfloat *z*)
void **glTranslatex**(GLfixed *x*, GLfixed *y*, GLfixed *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:

$$\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

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 untranslated coordinate system.

See Also

glMatrixMode, **glMultMatrix**, **glPushMatrix**, **glRotate**, **glScale**

Name

glVertexPointer – define an array of vertex coordinates

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 vertex coordinate in the array. Symbolic constants `GL_BYTE`, `GL_SHORT`, and `GL_FIXED`, are accepted. However, the initial value is `GL_FLOAT`. The common profile accepts the symbolic constant `GL_FLOAT` as well.
stride	Specifies the byte offset between consecutive vertices. If *stride* is 0, the vertices are understood to be tightly packed in the array. The initial value is 0.
pointer	Specifies a pointer to the first coordinate of the first vertex in the array. The initial value is 0.

Description

glVertexPointer specifies the location and data 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 vertices and attributes to be packed into a single array or stored in separate arrays. (Single-array storage may be more efficient on some implementations.) When a vertex array is specified, *size*, *type*, *stride*, and *pointer* are saved as client-side state.

If the vertex array is enabled, it is used when **glDrawArrays**, or **glDrawElements** is called. To enable and disable the vertex array, call **glEnableClientState** and **glDisableClientState** with the argument `GL_VERTEX_ARRAY`. The vertex array is initially disabled and isn't accessed when **glDrawArrays** or **glDrawElements** is called.

Use **glDrawArrays** to construct a sequence of primitives (all of the same type) from prespecified vertex and vertex attribute arrays. Use **glDrawElements** to construct a sequence of primitives by indexing vertices and vertex attributes.

Notes

glVertexPointer is typically implemented on the client side.

Errors

`GL_INVALID_VALUE` is generated if *size* is not 2, 3, or 4.

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

`GL_INVALID_VALUE` is generated if *stride* is negative.

See Also

glColorPointer, **glDrawArrays**, **glDrawElements**, **glEnable**, **glNormalPointer**, **glTexCoordPointer**

Name

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 surface (e.g. window), *width* and *height* are set to the dimensions of that surface.

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:

$$x_w = (x_{nd} + 1)\,\frac{width}{2} + x$$
$$y_w = (y_{nd} + 1)\,\frac{height}{2} + y$$

Viewport width and height are silently clamped to a range that depends on the implementation. To query this range, call **glGetInteger** with argument `GL_MAX_VIEWPORT_DIMS`.

Errors

`GL_INVALID_VALUE` is generated if either *width* or *height* is negative.

Associated Gets

glGetInteger with argument `GL_MAX_VIEWPORT_DIMS`

See Also

glDepthRange, **glGetInteger**

Chapter 4

EGL Reference Pages

Name

eglChooseConfig – return a list of EGL frame buffer configurations that match specified attributes

C Specification

```
EGLBoolean eglChooseConfig(EGLDisplay display,
                           EGLint const * attrib_list,
                           EGLConfig * configs,
                           EGLint config_size,
                           EGLint * num_config)
```

Parameters

display Specifies the EGL display connection.

attrib_list
Specifies attributes required to match by configs.

configs Returns an array of frame buffer configurations.

config_size
Specifies the size of the array of frame buffer configurations.

num_config
Returns the number of frame buffer configurations returned.

Description

eglChooseConfig returns a list of all EGL frame buffer configurations that match the attributes specified in *attrib_list*. The items in the list can be used in any EGL function that requires an EGL frame buffer configuration.

configs does not return values, if it is specified as `NULL`. This is useful for querying just the number of matching frame buffer configurations.

All attributes in *attrib_list*, including boolean attributes, are immediately followed by the corresponding desired value. The list is terminated with `EGL_NONE`. If an attribute is not specified in *attrib_list* then the default value (see below) is used (and the attribute is said to be specified implicitly). For example, if `EGL_DEPTH_SIZE` is not specified then it is assumed to be 0. For some attributes, the default is `EGL_DONT_CARE` meaning that any value is OK for this attribute, so the attribute will not be checked.

Attributes are matched in an attribute-specific manner. Some of the attributes, such as `EGL_LEVEL`, must match the specified value exactly. Others, such as, `EGL_RED_SIZE` must meet or exceed the specified minimum values. If more than one EGL frame buffer configuration is found, then a list of configurations, sorted according to the "best" match criteria, is returned. The match criteria for each attribute and the exact sorting order is defined below.

The interpretations of the various EGL frame buffer configuration attributes are as follows:

`EGL_BUFFER_SIZE`
: Must be followed by a nonnegative integer that indicates the desired color buffer size. The smallest color buffer of at least the specified size is preferred. The default value is 0.

`EGL_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. The default value is 0.

`EGL_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. The default value is 0.

`EGL_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. The default value is 0.

`EGL_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. The default value is 0.

`EGL_CONFIG_CAVEAT`
: Must be followed by one of `EGL_DONT_CARE`, `EGL_NONE`, `EGL_SLOW_CONFIG`, `EGL_NON_CONFORMANT_CONFIG`. If `EGL_NONE` is specified, then only frame buffer configurations with no caveats will be considered. If `EGL_SLOW_CONFIG` is specified, then only slow frame buffer configurations will be considered. If `EGL_NON_CONFORMANT_CONFIG` is specified, then only non-conformant frame buffer configurations will be considered. The default value is `EGL_DONT_CARE`.

`EGL_CONFIG_ID`
: Must be followed by a valid ID that indicates the desired EGL frame buffer configuration. When a `EGL_CONFIG_ID` is specified, all attributes are ignored. The default value is `EGL_DONT_CARE`.

`EGL_DEPTH_SIZE`
: Must be followed by a nonnegative integer that indicates the desired depth buffer size. The smallest available depth buffer of at least the minimum size is preferred. If the desired value is zero, frame buffer configurations with no depth buffer are preferred. The default value is 0.

`EGL_LEVEL`
: Must be followed by an integer buffer-level specification. This specification is honored exactly. Buffer level 0 corresponds to the default frame buffer of the display. Buffer level 1 is the first overlay frame buffer, level two the second overlay frame buffer, and so on. Negative buffer levels correspond to underlay frame buffers. The default value is 0.

`EGL_NATIVE_RENDERABLE`
: Must be followed by `EGL_DONT_CARE`, `EGL_TRUE`, or `EGL_FALSE`. If `EGL_TRUE` is specified, then only frame buffer configurations that allow native rendering into the surface will be considered. The default value is `EGL_DONT_CARE`.

EGL_NATIVE_VISUAL_TYPE
: Must be followed by a platform dependent value or EGL_DONT_CARE. The default value is EGL_DONT_CARE.

EGL_SAMPLE_BUFFERS
: Must be followed by the minimum acceptable number of multisample buffers. Configurations with the smallest number of multisample buffers that meet or exceed this minimum number are preferred. Currently operation with more than one multisample buffer is undefined, so only values of zero or one will produce a match. The default value is 0.

EGL_SAMPLES
: Must be followed by the minimum number of samples required in multisample buffers. Configurations with the smallest number of samples that meet or exceed the specified minimum number are preferred. Note that it is possible for color samples in the multisample buffer to have fewer bits than colors in the main color buffers. However, multisampled colors maintain at least as much color resolution in aggregate as the main color buffers.

EGL_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, frame buffer configurations with no stencil buffer are preferred. The default value is 0.

EGL_SURFACE_TYPE
: Must be followed by a mask indicating which EGL surface types the frame buffer configuration must support. Valid bits are EGL_WINDOW_BIT, EGL_PBUFFER_BIT, and EGL_PIXMAP_BIT. For example, if mask is set to EGL_WINDOW_BIT | EGL_PIXMAP_BIT, only frame buffer configurations that support both windows and pixmaps will be considered. The default value is EGL_WINDOW_BIT.

EGL_TRANSPARENT_TYPE
: Must be followed by one of EGL_NONE or EGL_TRANSPARENT_RGB. If EGL_NONE is specified, then only opaque frame buffer configurations will be considered. If EGL_TRANSPARENT_RGB is specified, then only transparent frame buffer configurations will be considered. The default value is EGL_NONE.

`EGL_TRANSPARENT_RED_VALUE`
: Must be followed by an integer value indicating the transparent red value. The value must be between 0 and the maximum color buffer value for red. Only frame buffer configurations that use the specified transparent red value will be considered. The default value is `EGL_DONT_CARE`.

  This attribute is ignored unless `EGL_TRANSPARENT_TYPE` is included in *attrib_list* and specified as `EGL_TRANSPARENT_RGB`.

`EGL_TRANSPARENT_GREEN_VALUE`
: Must be followed by an integer value indicating the transparent green value. The value must be between 0 and the maximum color buffer value for red. Only frame buffer configurations that use the specified transparent green value will be considered. The default value is `EGL_DONT_CARE`.

  This attribute is ignored unless `EGL_TRANSPARENT_TYPE` is included in *attrib_list* and specified as `EGL_TRANSPARENT_RGB`.

`EGL_TRANSPARENT_BLUE_VALUE`
: Must be followed by an integer value indicating the transparent blue value. The value must be between 0 and the maximum color buffer value for red. Only frame buffer configurations that use the specified transparent blue value will be considered. The default value is `EGL_DONT_CARE`.

  This attribute is ignored unless `EGL_TRANSPARENT_TYPE` is included in *attrib_list* and specified as `EGL_TRANSPARENT_RGB`.

When more than one EGL frame buffer configuration matches the specified attributes, a list of matching configurations is returned. The list is sorted according to the following precedence rules, which are applied in ascending order (i.e., configurations that are considered equal by a lower numbered rule are sorted by the higher numbered rule):

1. By `EGL_CONFIG_CAVEAT`, where the precedence is `EGL_NONE`, `EGL_SLOW_CONFIG`, and `EGL_NON_CONFORMANT_CONFIG`.

2. Larger total number of color components (`EGL_RED_SIZE`, `EGL_GREEN_SIZE`, `EGL_BLUE_SIZE`, and `EGL_ALPHA_SIZE`) that have higher number of bits. If the requested number of bits in *attrib_list* is zero or `EGL_DONT_CARE` for a particular color component, then the number of bits for that component is not considered.

3. Smaller `EGL_BUFFER_SIZE`.
4. Smaller `EGL_SAMPLE_BUFFERS`.
5. Smaller `EGL_SAMPLES`.
6. Smaller `EGL_DEPTH_SIZE`.
7. Smaller `EGL_STENCIL_SIZE`.
8. By `EGL_NATIVE_VISUAL_TYPE`, where the precedence order is platform dependent.
9. Smaller `EGL_CONFIG_ID`.

Examples

The following example specifies a frame buffer configuration in the normal frame buffer (not an overlay or underlay). The returned frame buffer configuration supports at least 4 bits each of red, green and blue and possible no alpha bits. The code shown in the example may or may not have a depth buffer, or a stencil buffer.

```
EGLint const attrib_list[] = {
    EGL_RED_SIZE, 4,
    EGL_GREEN_SIZE, 4,
    EGL_BLUE_SIZE, 4,
    EGL_NONE
};
```

Notes

eglGetConfigs and **eglGetConfigAttrib** can be used to implement selection algorithms other than the generic one implemented by **eglChooseConfig**. Call **eglGetConfigs** to retrieve all the frame buffer configurations, or alternatively, all the frame buffer configurations with a particular set of attributes. Next call **eglGetConfigAttrib** to retrieve additional attributes for the frame buffer configurations and then select between them.

EGL implementors are strongly discouraged, but not proscribed, from changing the selection algorithm used by **eglChooseConfig**. Therefore, selections may change from release to release of the client-side library.

Errors

`EGL_FALSE` is returned on failure, `EGL_TRUE` otherwise. *configs* and *num_config* are not modified when `EGL_FALSE` is returned.

`EGL_BAD_DISPLAY` is generated if *display* is not an EGL display connection.

`EGL_BAD_ATTRIBUTE` is generated if *attribute_list* contains an invalid frame buffer configuration attribute or an attribute value that is unrecognized or out of range.

`EGL_NOT_INITIALIZED` is generated if *display* has not been initialized.

`EGL_BAD_PARAMETER` is generated if *num_config* is `NULL`.

See Also

eglCreateContext, **eglCreatePbufferSurface**, **eglCreatePixmapSurface**, **eglCreateWindowSurface**, **eglGetConfigs**, **eglGetConfigAttrib**

Name

eglCopyBuffers – copy EGL surface color buffer to a native pixmap

C Specification

```
EGLBoolean eglCopyBuffers(EGLDisplay display,
                          EGLSurface surface,
                          NativePixmapType native_pixmap)
```

Parameters

display	Specifies the EGL display connection.
surface	Specifies the EGL surface whose color buffer is to be copied.
native_pixmap	Specifies the native pixmap as target of the copy.

Description

eglCopyBuffers copies the color buffer of *surface* to *native_pixmap*.

eglCopyBuffers performs an implicit **glFlush** before it returns. Subsequent GL commands may be issued immediately after calling **eglCopyBuffers**, but are not executed until copying of the color buffer is completed.

Notes

The color buffer of *surface* is left unchanged after calling **eglCopyBuffers**.

Errors

`EGL_FALSE` is returned if swapping of the surface buffers fails, `EGL_TRUE` otherwise.

`EGL_BAD_DISPLAY` is generated if *display* is not an EGL display connection.

`EGL_NOT_INITIALIZED` is generated if *display* has not been initialized.

`EGL_BAD_SURFACE` is generated if *surface* is not an EGL drawing surface.

`EGL_BAD_NATIVE_PIXMAP` is generated if the implementation does not support native pixmaps.

`EGL_BAD_NATIVE_PIXMAP` may be generated if *native_pixmap* is not a valid native pixmap.

`EGL_BAD_MATCH` is generated if the format of *native_pixmap* is not compatible with the color buffer of *surface*.

See Also

glFlush, **eglSwapBuffers**

Name

eglCreateContext – create a new EGL rendering context

C Specification

```
EGLContext eglCreateContext(EGLDisplay display,
                            EGLConfig config,
                            EGLContext share_context,
                            EGLint const * attrib_list)
```

Parameters

display	Specifies the EGL display connection.
config	Specifies the EGL frame buffer configuration that defines the frame buffer resource available to the rendering context.
share_context	Specifies the EGL rendering context with which to share texture objects. `EGL_NO_CONTEXT` indicates that no sharing is to take place.
attrib_list	Specifies attributes.

Description

eglCreateContext creates an EGL rendering context and returns its handle. This context can be used to render into an EGL drawing surface. If **eglCreateContext** fails to create a rendering context, `EGL_NO_CONTEXT` is returned.

If *share_context* is not `EGL_NO_CONTEXT`, then all texture objects except object 0, are shared by context *share_context* and by the newly created context. An arbitrary number of rendering contexts can share a single texture object space. However, all rendering contexts that share a single texture object space must themselves exist in the same address space. Two rendering contexts share an address space if both are owned by a single process.

Notes

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

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

Errors

`EGL_NO_CONTEXT` is returned if creation of the context fails.

`EGL_BAD_DISPLAY` is generated if *display* is not an EGL display connection.

`EGL_NOT_INITIALIZED` is generated if *display* has not been initialized.

`EGL_BAD_CONFIG` is generated if *config* is not an EGL frame buffer configuration.

`EGL_BAD_CONTEXT` is generated if *share_context* is not an EGL rendering context and is not `EGL_NO_CONTEXT`.

`EGL_BAD_ATTRIBUTE` is generated if *attrib_list* contains an invalid context attribute or if an attribute is not recognized or out of range.

`EGL_BAD_ALLOC` is generated if there are not enough resources to allocate the new context.

See Also

eglDestroyContext, **eglChooseConfig**, **eglGetConfigs**, **eglMakeCurrent**

Name

eglCreatePbufferSurface – create a new EGL pixel buffer surface

C Specification

```
EGLSurface eglCreatePbufferSurface(EGLDisplay display,
                                   EGLConfig config,
                                   EGLint const * attrib_list)
```

Parameters

display	Specifies the EGL display connection.
config	Specifies the EGL frame buffer configuration that defines the frame buffer resource available to the surface.
attrib_list	Specifies the pixel buffer surface attributes. May be `NULL` or empty (first attribute is `EGL_NONE`). Accepted attributes are `EGL_WIDTH`, `EGL_HEIGHT`, and `EGL_LARGEST_PBUFFER`.

Description

eglCreatePbufferSurface creates an off-screen pixel buffer surface and returns its handle. If **eglCreatePbufferSurface** fails to create a pixel buffer surface, `EGL_NO_SURFACE` is returned.

Any EGL rendering context that was created with respect to *config* can be used to render into the surface. Use **eglMakeCurrent** to attach an EGL rendering context to the surface.

Use **eglQuerySurface** to retrieve the dimensions of the allocated pixel buffer surface or the ID of *config*.

Use **eglDestroySurface** to destroy the surface.

The pixel buffer surface attributes are specified as a list of attribute value pairs, terminated with `EGL_NONE`. The accepted attributes for an EGL pixel buffer surface are:

`EGL_WIDTH`
: Requests a pixel buffer surface with the specified width. The default value is 0.

`EGL_HEIGHT`
: Requests a pixel buffer surface with the specified height. The default value is 0.

`EGL_LARGEST_PBUFFER`
: Requests a pixel buffer surface with the largest width and height. Use **eglQuerySurface** to retrieve the dimensions of the allocated pixel buffer. Default value is `EGL_FALSE`.

Errors

`EGL_NO_SURFACE` is returned if creation of the context fails.

`EGL_BAD_DISPLAY` is generated if *display* is not an EGL display connection.

`EGL_NOT_INITIALIZED` is generated if *display* has not been initialized.

`EGL_BAD_CONFIG` is generated if *config* is not an EGL frame buffer configuration.

`EGL_BAD_ATTRIBUTE` is generated if *attrib_list* contains an invalid pixel buffer attribute or if an attribute value is not recognized or out of range.

`EGL_BAD_ALLOC` is generated if there are not enough resources to allocate the new surface.

`EGL_BAD_MATCH` is generated if *config* does not support rendering to pixel buffers (the `EGL_SURFACE_TYPE` attribute does not contain `EGL_PBUFFER_BIT`).

See Also

eglDestroySurface, **eglChooseConfig**, **eglGetConfigs**, **eglMakeCurrent**, **egl-QuerySurface**

Name

eglCreatePixmapSurface – create a new EGL pixmap surface

C Specification

```
EGLSurface eglCreatePixmapSurface(EGLDisplay display,
                                  EGLConfig config,
                                  NativePixmapType native_pixmap,
                                  EGLint const * attrib_list)
```

Parameters

display	Specifies the EGL display connection.
config	Specifies the EGL frame buffer configuration that defines the frame buffer resource available to the surface.
native_pixmap	Specifies the native pixmap.
attrib_list	Specifies pixmap surface attributes. Must be `NULL` or empty (first attribute is `EGL_NONE`).

Description

eglCreatePixmapSurface creates an off-screen EGL pixmap surface and returns its handle. If **eglCreatePixmapSurface** fails to create a pixmap surface, `EGL_NO_SURFACE` is returned.

Any EGL rendering context that was created with respect to *config* can be used to render into the surface. Use **eglMakeCurrent** to attach an EGL rendering context to the surface.

Use **eglQuerySurface** to retrieve the ID of *config*.

Use **eglDestroySurface** to destroy the surface.

Errors

`EGL_NO_SURFACE` is returned if creation of the context fails.

`EGL_BAD_DISPLAY` is generated if *display* is not an EGL display connection.

`EGL_NOT_INITIALIZED` is generated if *display* has not been initialized.

`EGL_BAD_CONFIG` is generated if *config* is not an EGL config.

`EGL_BAD_NATIVE_PIXMAP` may be generated if *native_pixmap* is not a valid native pixmap.

`EGL_BAD_ATTRIBUTE` is generated if *attrib_list* contains an invalid pixmap attribute or if an attribute value is not recognized or out of range.

`EGL_BAD_ALLOC` is generated if there are not enough resources to allocate the new surface.

`EGL_BAD_MATCH` is generated if the attributes of *native_pixmap* do not correspond to *config* or if *config* does not support rendering to pixmaps (the `EGL_SURFACE_TYPE` attribute does not contain `EGL_PIXMAP_BIT`).

See Also

eglDestroySurface, **eglChooseConfig**, **eglGetConfigs**, **eglMakeCurrent**, **egl-QuerySurface**

Name

eglCreateWindowSurface – create a new EGL window surface

C Specification

```
EGLSurface eglCreateWindowSurface(EGLDisplay display,
                                  EGLConfig config,
                                  NativeWindowType native_window,
                                  EGLint const * attrib_list)
```

Parameters

display	Specifies the EGL display connection.
config	Specifies the EGL frame buffer configuration that defines the frame buffer resource available to the surface.
native_window	Specifies the native window.
attrib_list	Specifies window surface attributes. Must be `NULL` or empty (first attribute is `EGL_NONE`).

Description

eglCreateWindowSurface creates an EGL window surface and returns its handle. If **eglCreateWindowSurface** fails to create a window surface, `EGL_NO_SURFACE` is returned.

Any EGL rendering context that was created with respect to *config* can be used to render into the surface. Use **eglMakeCurrent** to attach an EGL rendering context to the surface.

Use **eglQuerySurface** to retrieve the ID of *config*.

Use **eglDestroySurface** to destroy the surface.

Errors

EGL_NO_SURFACE is returned if creation of the context fails.

EGL_BAD_DISPLAY is generated if *display* is not an EGL display connection.

EGL_NOT_INITIALIZED is generated if *display* has not been initialized.

EGL_BAD_CONFIG is generated if *config* is not an EGL frame buffer configuration.

EGL_BAD_NATIVE_WINDOW may be generated if *native_window* is not a valid native window.

EGL_BAD_ATTRIBUTE is generated if *attrib_list* contains an invalid window attribute or if an attribute value is not recognized or is out of range.

EGL_BAD_ALLOC is generated if there are not enough resources to allocate the new surface.

EGL_BAD_MATCH is generated if the attributes of *native_window* do not correspond to *config* or if *config* does not support rendering to windows (the EGL_SURFACE_TYPE attribute does not contain EGL_WINDOW_BIT).

See Also

eglDestroySurface, **eglChooseConfig**, **eglGetConfigs**, **eglMakeCurrent**, **egl-QuerySurface**

Name

eglDestroyContext – destroy an EGL rendering context

C Specification

```
EGLBoolean eglDestroyContext(EGLDisplay display,
                             EGLContext context)
```

Parameters

display	Specifies the EGL display connection.
context	Specifies the EGL rendering context to be destroyed.

Description

If the EGL rendering context *context* is not current to any thread, **eglDestroyContext** destroys it immediately. Otherwise, *context* is destroyed when it becomes not current to any thread.

Errors

`EGL_FALSE` is returned if destruction of the context fails, `EGL_TRUE` otherwise.

`EGL_BAD_DISPLAY` is generated if *display* is not an EGL display connection.

`EGL_NOT_INITIALIZED` is generated if *display* has not been initialized.

`EGL_BAD_CONTEXT` is generated if *context* is not an EGL rendering context.

See Also

eglCreateContext, **eglMakeCurrent**

Name

eglDestroySurface – destroy an EGL surface

C Specification

```
EGLBoolean eglDestroySurface(EGLDisplay display,
                             EGLSurface surface)
```

Parameters

display	Specifies the EGL display connection.
surface	Specifies the EGL surface to be destroyed.

Description

If the EGL surface *surface* is not current to any thread, **eglDestroySurface** destroys it immediately. Otherwise, *surface* is destroyed when it becomes not current to any thread.

Errors

`EGL_FALSE` is returned if destruction of the surface fails, `EGL_TRUE` otherwise.

`EGL_BAD_DISPLAY` is generated if *display* is not an EGL display connection.

`EGL_NOT_INITIALIZED` is generated if *display* has not been initialized.

`EGL_BAD_SURFACE` is generated if *surface* is not an EGL surface.

See Also

eglCreatePbufferSurface, **eglCreatePixmapSurface**, **eglCreateWindowSurface**, **eglMakeCurrent**

Name

eglGetConfigAttrib – return information about an EGL frame buffer configuration

C Specification

```
EGLBoolean eglGetConfigAttrib(EGLDisplay display,
                              EGLConfig config,
                              EGLint attribute,
                              EGLint * value)
```

Parameters

display	Specifies the EGL display connection.
config	Specifies the EGL frame buffer configuration to be queried.
attribute	Specifies the EGL rendering context attribute to be returned.
value	Returns the requested value.

Description

eglGetConfigAttrib returns in *value* the value of *attribute* for *config*. *attribute* can be one of the following:

`EGL_BUFFER_SIZE`
: Returns the depth of the color buffer. It is the sum of `EGL_RED_SIZE`, `EGL_GREEN_SIZE`, `EGL_BLUE_SIZE`, and `EGL_ALPHA_SIZE`.

`EGL_RED_SIZE`
: Returns the number of bits of red stored in the color buffer.

`EGL_GREEN_SIZE`
: Returns the number of bits of green stored in the color buffer.

`EGL_BLUE_SIZE`
: Returns the number of bits of blue stored in the color buffer.

`EGL_ALPHA_SIZE`
: Returns the number of bits of alpha stored in the color buffer.

`EGL_CONFIG_CAVEAT`
: Returns the caveats for the frame buffer configuration. Possible caveat values are `EGL_NONE`, `EGL_SLOW_CONFIG`, and `EGL_NON_CONFORMANT`.

`EGL_CONFIG_ID`
: Returns the ID of the frame buffer configuration.

`EGL_DEPTH_SIZE`
: Returns the number of bits in the depth buffer.

`EGL_LEVEL`
: Returns the frame buffer level. 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.

`EGL_MAX_PBUFFER_WIDTH`
: Returns the maximum width of a pixel buffer surface in pixels.

`EGL_MAX_PBUFFER_HEIGHT`
: Returns the maximum height of a pixel buffer surface in pixels.

`EGL_MAX_PBUFFER_PIXELS`
: Returns the maximum size of a pixel buffer surface in pixels.

`EGL_NATIVE_RENDERABLE`
: Returns `EGL_TRUE` if native rendering APIs can render into the surface, `EGL_FALSE` otherwise.

`EGL_NATIVE_VISUAL_ID`
: Returns the ID of the associated native visual.

`EGL_NATIVE_VISUAL_TYPE`
: Returns the type of the associated native visual.

`EGL_PRESERVED_RESOURCES`
: Returns `EGL_TRUE` if resources are preserved across power management events, `EGL_FALSE` otherwise.

`EGL_SAMPLE_BUFFERS`
: Returns the number of multisample buffers.

`EGL_SAMPLES`
: Returns the number of samples per pixel.

`EGL_STENCIL_BITS`
: Returns the number of bits in the stencil buffer.

`EGL_SURFACE_TYPE`
: Returns the types of supported EGL surfaces.

`EGL_TRANSPARENT_TYPE`
: Returns the type of supported transparency. Possible transparency values are: `EGL_NONE`, and `EGL_TRANSPARENT_RGB`.

`EGL_TRANSPARENT_RED`
: Returns the transparent red value.

`EGL_TRANSPARENT_GREEN`
: Returns the transparent green value.

`EGL_TRANSPARENT_BLUE`
: Returns the transparent blue value.

Errors

`EGL_FALSE` is returned on failure, `EGL_TRUE` otherwise. *value* is not modified when `EGL_FALSE` is returned.

`EGL_BAD_DISPLAY` is generated if *display* is not an EGL display connection.

`EGL_NOT_INITIALIZED` is generated if *display* has not been initialized.

`EGL_BAD_CONFIG` is generated if *config* is not an EGL frame buffer configuration.

`EGL_BAD_ATTRIBUTE` is generated if *attribute* is not a valid frame buffer configuration attribute.

See Also

eglChooseConfig, **eglGetConfigs**

Name

eglGetConfigs – return a list of all EGL frame buffer configurations for a display

C Specification

```
EGLBoolean eglGetConfigs(EGLDisplay display,
                         EGLConfig * configs,
                         EGLint config_size,
                         EGLint * num_config)
```

Parameters

display	Specifies the EGL display connection.
configs	Returns a list of configs.
config_size	Specifies the size of the list of configs.
num_config	Returns the number of configs returned.

Description

eglGetConfigs returns a list of all EGL frame buffer configurations that are available for the specified display. The items in the list can be used in any EGL function that requires an EGL frame buffer configuration.

configs does not return values, if it is specified as `NULL`. This is useful for querying just the number of all frame buffer configurations.

Use **eglGetConfigAttrib** to retrieve individual attribute values of a frame buffer configuration.

Errors

`EGL_FALSE` is returned on failure, `EGL_TRUE` otherwise. *configs* and *num_config* are not modified when `EGL_FALSE` is returned.

`EGL_BAD_DISPLAY` is generated if *display* is not an EGL display connection.

`EGL_NOT_INITIALIZED` is generated if *display* has not been initialized.

`EGL_BAD_PARAMETER` is generated if *num_config* is `NULL`.

See Also

eglCreateContext, **eglCreatePbufferSurface**, **eglCreatePixmapSurface**, **egl-CreateWindowSurface**, **eglChooseConfig**, **eglGetConfigAttrib**

Name

eglGetCurrentContext – return the current EGL rendering context

C Specification

`EGLContext` **eglGetCurrentContext**`(void)`

Description

eglGetCurrentContext returns the current EGL rendering context, as specified by **eglMakeCurrent**. If no context is current, `EGL_NO_CONTEXT` is returned.

See Also

eglCreateContext, **eglMakeCurrent**

Name

eglGetCurrentDisplay – return the display for the current EGL rendering context

C Specification

`EGLDisplay` **eglGetCurrentDisplay**`(void)`

Description

eglGetCurrentDisplay returns the current EGL display connection for the current EGL rendering context, as specified by **eglMakeCurrent**. If no context is current, `EGL_NO_DISPLAY` is returned.

See Also

eglGetDisplay, **eglInitialize**, **eglMakeCurrent**

Name

eglGetCurrentSurface – return the read or draw surface for the current EGL rendering context

C Specification

`EGLSurface` **eglGetCurrentSurface**`(EGLint` *readdraw*`)`

Parameters

readdraw Specifies whether the EGL read or draw surface is to be returned.

Description

eglGetCurrentSurface returns the read or draw surface attached to the current EGL rendering context, as specified by **eglMakeCurrent**. If no context is current, `EGL_NO_SURFACE` is returned.

See Also

eglCreatePbufferSurface, **eglCreatePixmapSurface**, **eglCreateWindowSurface**, **eglMakeCurrent**

Name

eglGetDisplay – return an EGL display connection

C Specification

`EGLDisplay` **eglGetDisplay**`(NativeDisplayType` *native_display*)

Parameters

native_display
: Specifies the display to connect to. `EGL_DEFAULT_DISPLAY` indicates the default display.

Description

eglGetDisplay obtains the EGL display connection for the native display *native_display*.

If *display_id* is `EGL_DEFAULT_DISPLAY`, a default display connection is returned.

If no display connection matching *native_display* is available, `EGL_NO_DISPLAY` is returned. No error is generated.

Use **eglInitialize** to initialize the display connection.

See Also

eglInitialize

Name

eglGetError – return error information

C Specification

`EGLint` **eglGetError**`(void)`

Description

eglGetError returns the error of the last called EGL function in the current thread. Initially, the error is set to `EGL_SUCCESS`.

The following errors are currently defined:

`EGL_SUCCESS`
: The last function succeeded without error.

`EGL_NOT_INITIALIZED`
: EGL is not initialized, or could not be initialized, for the specified EGL display connection.

`EGL_BAD_ACCESS`
: EGL cannot access a requested resource (for example a context is bound in another thread).

`EGL_BAD_ALLOC`
: EGL failed to allocate resources for the requested operation.

`EGL_BAD_ATTRIBUTE`
: An unrecognized attribute or attribute value was passed in the attribute list.

`EGL_BAD_CONTEXT`
: An `EGLContext` argument does not name a valid EGL rendering context.

`EGL_BAD_CONFIG`
: An `EGLConfig` argument does not name a valid EGL frame buffer configuration.

`EGL_BAD_CURRENT_SURFACE`
: The current surface of the calling thread is a window, pixel buffer or pixmap that is no longer valid.

`EGL_BAD_DISPLAY`
: An `EGLDisplay` argument does not name a valid EGL display connection.

`EGL_BAD_SURFACE`
: An `EGLSurface` argument does not name a valid surface (window, pixel buffer or pixmap) configured for GL rendering.

`EGL_BAD_MATCH`
: Arguments are inconsistent (for example, a valid context requires buffers not supplied by a valid surface).

`EGL_BAD_PARAMETER`
: One or more argument values are invalid.

`EGL_BAD_NATIVE_PIXMAP`
: A `NativePixmapType` argument does not refer to a valid native pixmap.

`EGL_BAD_NATIVE_WINDOW`
: A `NativeWindowType` argument does not refer to a valid native window.

Errors

A call to **eglGetError** sets the error to `EGL_SUCCESS`.

Name

eglGetProcAddress – return a GL or an EGL extension function

C Specification

`void (* `**eglGetProcAddress**`(char const * `*procname*`))()`

Parameters

procname

Specifies the name of the function to return.

Description

eglGetProcAddress returns the address of the extension function named by *procname*. *procname* must be a null-terminated string. The pointer returned should be cast to a function pointer type matching the extension function's definition in that extension specification. A return value of `NULL` indicates that the specific function does not exist for the EGL implementation.

A non-`NULL` return value does not guarantee that an extension function is actually supported at runtime. The client must also query **glGetString**(`GL_EXTENSIONS`) or **eglQueryString**(*display*, `EGL_EXTENSIONS`) to determine if an extension is supported by a particular context or display.

Function pointers returned by **eglGetProcAddress** are independent of the display and the currently bound context and may be used by any context which supports the extension.

eglGetProcAddress may be queried for all GL and EGL extension functions.

See Also

glGetString, **eglQueryString**

Name

eglInitialize – initialize an EGL display connection

C Specification

```
EGLBoolean eglInitialize(EGLDisplay display,
                         EGLint * major,
                         EGLint * minor)
```

Parameters

display	Specifies the EGL display connection to initialize.
major	Returns the major version number of the EGL implementation. May be `NULL`.
minor	Returns the minor version number of the EGL implementation. May be `NULL`.

Description

eglInitialize initialized the EGL display connection obtained with **eglGetDisplay**. Initializing an already initialized EGL display connection has no effect besides returning the version numbers.

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

Use **eglTerminate** to release resources associated with an EGL display connection.

Errors

`EGL_FALSE` is returned if **eglInitialize** fails, `EGL_TRUE` otherwise. *major* and *minor* are not modified when `EGL_FALSE` is returned.

`EGL_BAD_DISPLAY` is generated if *display* is not an EGL display connection.

`EGL_NOT_INITIALIZED` is generated if *display* cannot be initialized.

See Also

eglGetDisplay, **eglTerminate**

Name

eglIntro – introduction to OpenGL ES in an EGL window system

Overview

OpenGL ES (GL) is a 3D-oriented renderer for embedded systems. It is made available through the *OpenGL ES Native Platform Graphics Interface* (EGL). Depending on its implementation EGL might be more or less tightly integrated into the native window system. Most EGL functions require an EGL display connection, which can be obtained by calling **eglGetDisplay** and passing in a native display handler or `EGL_DEFAULT_DISPLAY`. To initialize and query what EGL version is supported on the display connection, call **eglInitialize**.

Native window systems supporting EGL make a subset of their visuals available for GL rendering. Windows and pixmaps created with these visuals may also be rendered into using the native window system API.

EGL extends a native window or pixmap with additional buffers. These buffers include a color buffer, a depth buffer, and a stencil buffer. Some or all of the buffers listed are included in each EGL frame buffer configuration.

EGL supports rendering into three types of surfaces: windows, pixmaps and pixel buffers (pbuffers). EGL window and pixmap surfaces are associated with corresponding resources of the native window system. EGL pixel buffers are EGL only resources, and might not accept rendering through the native window system.

To render using OpenGL ES into an EGL surface, you must determine the appropriate EGL frame buffer configuration, which supports the rendering features the application requires. **eglChooseConfig** returns an `EGLConfig` matching the required attributes, if any. A complete list of EGL frame buffer configurations can be obtained by calling **eglGetConfigs**. Attributes of a particular EGL frame buffer configuration can be queried by calling **eglGetConfigAttrib**.

For EGL window and pixmap surfaces, a suitable native window or pixmap with a matching native visual must be created first. For a given EGL frame buffer configuration, the native visual type and ID can be retrieved with a call to **eglGetConfigAttrib**. For pixel buffers, no underlying native resource is required.

To create an EGL window surface from a native window, call **eglCreateWindowSurface**. Likewise, to create an EGL pixmap surface, call **eglCreatePixmap-**

Surface. Pixel buffers are created by calling **eglCreatePbufferSurface** Use **eglDestroySurface** to release previously allocated resources.

An EGL rendering context is required to bind OpenGL ES rendering to an EGL surface. An EGL surface and an EGL rendering context must have compatible EGL frame buffer configurations. To create an EGL rendering context, call **eglCreateContext** An EGL rendering context may be bound to one or two EGL surfaces by calling **eglMakeCurrent**. This context/surfaces association becomes the current context and current surfaces, and is used by all GL rendering commands until **eglMakeCurrent** is called with different arguments.

Both native and GL commands may be used to operate on certain surfaces, however, the two command streams are not synchronized. Synchronization can be explicitly specified using by calling **eglWaitGL**, **eglWaitNative**, and possibly by calling other native window system commands.

Examples

Below is a minimal example of creating an RGBA-format window that allows rendering with OpenGL ES. The window is cleared to yellow when the program runs. For simplicity, the program does not check for any errors.

```
#include <stdlib.h>
#include <unistd.h>
#include <GLES/egl.h>
#include <GLES/gl.h>

typedef ... NativeWindowType;
extern NativeWindowType createNativeWindow(void);

static EGLint const attribute_list[] = {
    EGL_RED_SIZE, 1,
    EGL_GREEN_SIZE, 1,
    EGL_BLUE_SIZE, 1,
    EGL_NONE
};
```

```
int main(int argc, char ** argv)
{
    EGLDisplay display;
    EGLConfig config;
    EGLContext context;
    EGLSurface surface;
    NativeWindowType native_window;
    EGLint num_config;

    /* get an EGL display connection */
    display = eglGetDisplay(EGL_DEFAULT_DISPLAY);

    /* initialize the EGL display connection */
    eglInitialize(display, NULL, NULL);

    /* get an appropriate EGL frame buffer configuration */
    eglChooseConfig(display, attribute_list, &config, 1, &num_config);

    /* create an EGL rendering context */
    context = eglCreateContext(display, config, EGL_NO_CONTEXT, NULL);

    /* create a native window */
    native_window = createNativeWindow();

    /* create an EGL window surface */
    surface = eglCreateWindowSurface(display, config, native_window, NULL);

    /* connect the context to the surface */
    eglMakeCurrent(display, surface, surface, context);

    /* clear the color buffer */
    glClearColor(1.0, 1.0, 0.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glFlush();
    eglSwapBuffers(display, surface);

    sleep(10);

    return EXIT_SUCCESS;
}
```

Using EGL Extensions

All supported EGL extensions will have a corresponding definition in `egl.h` and a token in the extensions string returned by **eglQueryString**. For extensions in OpenGL ES, refer to **glIntro**.

Future EGL Versions

eglInitialize and **eglQueryString** can be used to determine at run-time what version of EGL is available. To check the EGL version at compile-time, test whether `EGL_VERSION_`*x*`_`*y* is defined, where *x* and *y* are the major and minor version numbers.

Files

`GLES/egl.h`
EGL header file

See Also

glIntro, **glFinish**, **glFlush**, **eglChooseConfig**, **eglCreateContext**, **eglCreatePbufferSurface**, **eglCreatePixmapSurface**, **eglCreateWindowSurface**, **eglDestroyContext**, **eglDestroySurface**, **eglGetConfigs**, **eglGetDisplay**, **eglInitialize**, **eglMakeCurrent**, **eglQueryString**, **eglSwapBuffers**, **eglTerminate**, **eglWaitGL**, **eglWaitNative**

Name

eglMakeCurrent – attach an EGL rendering context to EGL surfaces

C Specification

```
EGLBoolean eglMakeCurrent(EGLDisplay display,
                          EGLSurface draw,
                          EGLSurface read,
                          EGLContext context)
```

Parameters

display	Specifies the EGL display connection.
draw	Specifies the EGL draw surface.
read	Specifies the EGL read surface.
context	Specifies the EGL rendering context to be attached to the surfaces.

Description

eglMakeCurrent binds *context* to the current rendering thread and to the *draw* and *read* surfaces. *draw* is used for all GL operations except for any pixel data read back (**glReadPixels**, **glCopyTexImage2D**, and **glCopyTexSubImage2D**), which is taken from the frame buffer values of *read*.

If the calling thread has already a current rendering context, that context is flushed and marked as no longer current.

The first time that *context* is made current, the viewport and scissor dimensions are set to the size of the *draw* surface. The viewport and scissor are not modified when *context* is subsequently made current.

To release the current context without assigning a new one, call **eglMakeCurrent** with *draw* and *read* set to `EGL_NO_SURFACE` and *context* set to `EGL_NO_CONTEXT`.

Use **eglGetCurrentContext**, **eglGetCurrentDisplay**, and **eglGetCurrentSurface** to query the current rendering context and associated display connection and surfaces.

Errors

`EGL_FALSE` is returned on failure, `EGL_TRUE` otherwise. If `EGL_FALSE` is returned, the previously current rendering context and surfaces (if any) remain unchanged.

`EGL_BAD_DISPLAY` is generated if *display* is not an EGL display connection.

`EGL_NOT_INITIALIZED` is generated if *display* has not been initialized.

`EGL_BAD_SURFACE` is generated if *draw* or *read* is not an EGL surface.

`EGL_BAD_CONTEXT` is generated if *context* is not an EGL rendering context.

`EGL_BAD_MATCH` is generated if *draw* or *read* are not compatible with *context*, or if *context* is set to `EGL_NO_CONTEXT` and *draw* or *read* are not set to `EGL_NO_SURFACE`, or if *draw* or *read* are set to `EGL_NO_SURFACE` and *context* is not set to `EGL_NO_CONTEXT`.

`EGL_BAD_ACCESS` is generated if *context* is current to some other thread.

`EGL_BAD_NATIVE_PIXMAP` may be generated if a native pixmap underlying either *draw* or *read* is no longer valid.

`EGL_BAD_NATIVE_WINDOW` may be generated if a native window underlying either *draw* or *read* is no longer valid.

`EGL_BAD_CURRENT_SURFACE` is generated if the previous context has unflushed commands and the previous surface is no longer valid.

`EGL_BAD_ALLOC` may be generated if allocation of ancillary buffers for *draw* or *read* were delayed until **eglMakeCurrent** is called, and there are not enough resources to allocate them.

See Also

glReadPixels, **glCopyTexImage2D**, **glCopyTexSubImage2D**, **eglCreateContext**, **eglCreatePbufferSurface**, **eglCreatePixmapSurface**, **eglCreateWindowSurface**, **eglGetCurrentContext**, **eglGetCurrentDisplay**, **eglGetCurrentSurface**, **eglGetDisplay**, **eglInitialize**

Name

eglQueryContext – return EGL rendering context information

C Specification

```
EGLBoolean eglQueryContext(EGLDisplay display,
                           EGLContext context,
                           EGLint attribute,
                           EGLint * value)
```

Parameters

display	Specifies the EGL display connection.
context	Specifies the EGL rendering context to query.
attribute	Specifies the EGL rendering context attribute to be returned.
value	Returns the requested value.

Description

eglQueryContext returns in *value* the value of *attribute* for *context*. *attribute* can be one of the following:

`EGL_CONFIG_ID`
: Returns the ID of the EGL frame buffer configuration with respect to which the context was created.

Errors

`EGL_FALSE` is returned on failure, `EGL_TRUE` otherwise. *value* is not modified when `EGL_FALSE` is returned.

`EGL_BAD_DISPLAY` is generated if *display* is not an EGL display connection.

`EGL_NOT_INITIALIZED` is generated if *display* has not been initialized.

`EGL_BAD_CONTEXT` is generated if *context* is not an EGL rendering context.

`EGL_BAD_ATTRIBUTE` is generated if *attribute* is not a valid context attribute.

See Also

eglCreateContext

Name

eglQueryString – return a string describing an EGL display connection

C Specification

`char const *` **eglQueryString**`(EGLDisplay` *display*, `EGLint` *name*)

Parameters

display	Specifies the EGL display connection.
name	Specifies a symbolic constant, one of `EGL_VENDOR`, `EGL_VERSION`, or `EGL_EXTENSIONS`.

Description

eglQueryString returns a pointer to a static string describing an EGL display connection. *name* can be one of the following:

`EGL_VENDOR`
: Returns the company responsible for this EGL implementation. This name does not change from release to release.

`EGL_VERSION`
: Returns a version or release number. The `EGL_VERSION` string is laid out as follows:

  major_version.minor_version space *vendor_specific_info*

`EGL_EXTENSIONS`
: Returns a space separated list of supported extensions to EGL.

Errors

`NULL` is returned on failure.

`EGL_BAD_DISPLAY` is generated if *display* is not an EGL display connection.

`EGL_NOT_INITIALIZED` is generated if *display* has not been initialized.

`EGL_BAD_PARAMETER` is generated if *name* is not an accepted value.

See Also

eglGetDisplay, **eglInitialize**

Name

eglQuerySurface – return EGL surface information

C Specification

```
EGLBoolean eglQuerySurface(EGLDisplay display,
                           EGLSurface surface,
                           EGLint attribute,
                           EGLint * value)
```

Parameters

display	Specifies the EGL display connection.
surface	Specifies the EGL surface to query.
attribute	Specifies the EGL surface attribute to be returned.
value	Returns the requested value.

Description

eglQuerySurface returns in *value* the value of *attribute* for *surface*. *attribute* can be one of the following:

`EGL_CONFIG_ID`
: Returns the ID of the EGL frame buffer configuration with respect to which the surface was created.

`EGL_WIDTH`
: Returns the width of the surface in pixels.

`EGL_HEIGHT`
: Returns the height of the surface in pixels.

`EGL_LARGEST_PBUFFER`
: Returns the same attribute value specified when the surface was created with **eglCreatePbufferSurface**. For a window or pixmap surface, *value* is not modified.

Errors

`EGL_FALSE` is returned on failure, `EGL_TRUE` otherwise. *value* is not modified when `EGL_FALSE` is returned.

`EGL_BAD_DISPLAY` is generated if *display* is not an EGL display connection.

`EGL_NOT_INITIALIZED` is generated if *display* has not been initialized.

`EGL_BAD_SURFACE` is generated if *surface* is not an EGL surface.

`EGL_BAD_ATTRIBUTE` is generated if *attribute* is not a valid surface attribute.

See Also

eglCreatePbufferSurface, **eglCreatePixmapSurface**, **eglCreateWindowSurface**

Name

eglSwapBuffers – post EGL surface color buffer to a native window

C Specification

```
EGLBoolean eglSwapBuffers(EGLDisplay display,
                          EGLSurface surface)
```

Parameters

display	Specifies the EGL display connection.
surface	Specifies the EGL drawing surface whose buffers are to be swapped.

Description

If *surface* is a window surface, **eglSwapBuffers** posts its color buffer to the associated native window.

eglSwapBuffers performs an implicit **glFlush** before it returns. Subsequent GL commands may be issued immediately after calling **eglSwapBuffers**, but are not executed until the buffer exchange is completed.

If *surface* is a pixel buffer or a pixmap, **eglSwapBuffers** has no effect, and no error is generated.

Notes

The color buffer of *surface* is left undefined after calling **eglSwapBuffers**.

Errors

`EGL_FALSE` is returned if swapping of the surface buffers fails, `EGL_TRUE` otherwise.

`EGL_BAD_DISPLAY` is generated if *display* is not an EGL display connection.

`EGL_NOT_INITIALIZED` is generated if *display* has not been initialized.

`EGL_BAD_SURFACE` is generated if *surface* is not an EGL drawing surface.

See Also

glFlush, **eglCopyBuffers**

Name

eglTerminate – terminate an EGL display connection

C Specification

`EGLBoolean` **eglTerminate**`(EGLDisplay` *display*`)`

Parameters

display	Specifies the EGL display connection to terminate.

Description

eglTerminate releases resources associated with an EGL display connection. Termination marks all EGL resources associated with the EGL display connection for deletion. If contexts or surfaces associated with *display* is current to any thread, they are not released until they are no longer current as a result of **eglMakeCurrent**.

Terminating an already terminated EGL display connection has no effect. A terminated display may be re-initialized by calling **eglInitialize** again.

Errors

`EGL_FALSE` is returned if **eglTerminate** fails, `EGL_TRUE` otherwise.

`EGL_BAD_DISPLAY` is generated if *display* is not an EGL display connection.

See Also

eglInitialize, **eglMakeCurrent**

Name

eglWaitGL – complete GL execution prior to subsequent native rendering calls

C Specification

`EGLBoolean` **eglWaitGL**`(void)`

Description

GL rendering calls made prior to **eglWaitGL** are guaranteed to be executed before native rendering calls made after **eglWaitGL**. The same result can be achieved using **glFinish**.

eglWaitGL is ignored if there is no current EGL rendering context.

Errors

`EGL_FALSE` is returned if **eglWaitGL** fails, `EGL_TRUE` otherwise.

`EGL_BAD_NATIVE_SURFACE` is generated if the surface associated with the current context has a native window or pixmap, and that window or pixmap is no longer valid.

See Also

glFinish, **glFlush**, **eglWaitNative**

Name

eglWaitNative – complete native execution prior to subsequent GL rendering calls

C Specification

`EGLBoolean` **eglWaitNative**`(EGLint` *engine*`)`

Parameters

engine	Specifies a particular marking engine to be waited on. Must be `EGL_EGL_CORE_NATIVE_ENGINE`.

Description

Native rendering calls made prior to **eglWaitNative** are guaranteed to be executed before GL rendering calls made after **eglWaitNative**.

eglWaitNative is ignored if there is no current EGL rendering context.

Errors

`EGL_BAD_PARAMETER` is generated if *engine* is not a recognized marking engine.

`EGL_BAD_NATIVE_SURFACE` is generated if the surface associated with the current context has a native window or pixmap, and that window or pixmap is no longer valid.

See Also

glFinish, **glFlush**, **eglWaitGL**

Appendix A

SGI Free Software License B

Version 1.1 [02/22/2000]

1. **Definitions.**

   1.1. **"Additional Notice Provisions"** means such additional provisions as appear in the Notice in Original Code under the heading "Additional Notice Provisions."

   1.2. **"Covered Code"** means the Original Code or Modifications, or any combination thereof.

   1.3. **"Hardware"** means any physical device that accepts input, processes input, stores the results of processing, and/or provides output.

   1.4. **"Larger Work"** means a work that combines Covered Code or portions thereof with code not governed by the terms of this License.

   1.5. **"Licensable"** means having the right to grant, to the maximum extent possible, whether at the time of the initial grant or subsequently acquired, any and all of the rights conveyed herein.

   1.6. **"License"** means this document.

   1.7. **"Licensed Patents"** means patent claims Licensable by SGI that are infringed by the use or sale of Original Code or any Modifications provided by SGI, or any combination thereof.

1.8. **"Modifications"** means any addition to or deletion from the substance or structure of the Original Code or any previous Modifications. When Covered Code is released as a series of files, a Modification is:

   A. to the contents of a file containing Original Code and/or addition to or deletion from the contents of a file containing previous Modifications.

   B. file that contains any part of the Original Code or previous Modifications.

1.9. **"Notice"** means any notice in Original Code or Covered Code, as required by and in compliance with this License.

1.10. **"Original Code"** means source code of computer software code that is described in the source code Notice required by Exhibit A as Original Code, and updates and error corrections specifically thereto.

1.11. **"Recipient"** means an individual or a legal entity exercising rights under, and complying with all of the terms of, this License or a future version of this License issued under Section 8. For legal entities, "Recipient" includes any entity that controls, is controlled by, or is under common control with Recipient. For purposes of this definition, "control" of an entity means (a) the power, direct or indirect, to direct or manage such entity, or (b) ownership of fifty percent (50%) or more of the outstanding shares or beneficial ownership of such entity.

1.12. **"Recipient Patents"** means patent claims Licensable by a Recipient that are infringed by the use or sale of Original Code or any Modifications provided by SGI, or any combination thereof.

1.13. **"SGI"** means Silicon Graphics, Inc.

1.14. **"SGI Patents"** means patent claims Licensable by SGI other than the Licensed Patents.

2. **License Grant and Restrictions.**

2.1. **SGI License Grant.** Subject to the terms of this License and any third party intellectual property claims, for the duration of intellectual property protections inherent in the Original Code, SGI hereby grants Recipient a worldwide, royalty-free, non-exclusive license, to do the following: (i) under copyrights Licensable by SGI, to reproduce, distribute, create derivative works from, and, to the extent applicable, display and perform the Original Code and/or any Modifications provided by SGI alone and/or as part of a Larger Work; and (ii) under any Licensable Patents, to make, have made, use, sell, offer for sale, import and/or otherwise transfer the Original Code and/or any Modifications provided by SGI. Recipient accepts the terms and conditions of this License by undertaking any of the aforementioned actions. The patent license shall apply to the Covered Code if, at the time any related Modification is added, such addition of the Modification causes such combination to be covered by the Licensed Patents. The patent license in Section 2.1(ii) shall not apply to any other combinations that include the Modification. No patent license is provided under SGI Patents for infringements of SGI Patents by Modifications not provided by SGI or combinations of Original Code and Modifications not provided by SGI.

2.2. **Recipient License Grant.** Subject to the terms of this License and any third party intellectual property claims, Recipient hereby grants SGI and any other Recipients a worldwide, royalty-free, non-exclusive license, under any Recipient Patents, to make, have made, use, sell, offer for sale, import and/or otherwise transfer the Original Code and/or any Modifications provided by SGI.

2.3. **No License For Hardware Implementations.** The licenses granted in Section 2.1 and 2.2 are not applicable to implementation in Hardware of the algorithms embodied in the Original Code or any Modifications provided by SGI .

3. **Redistributions.**

3.1. **Retention of Notice/Copy of License.** The Notice set forth in Exhibit A, below, must be conspicuously retained or included in any and all redistributions of Covered Code. For distributions of the Covered Code in source code form, the Notice must appear in every file that can include a text comments field; in executable form, the Notice and a copy of this License must appear in related documentation or collateral where the Recipient's rights relating to Covered Code are described. Any Additional Notice Provisions which actually appears in the Original Code must also be retained or included in any and all redistributions of Covered Code.

3.2. **Alternative License.** Provided that Recipient is in compliance with the terms of this License, Recipient may, so long as without derogation of any of SGI's rights in and to the Original Code, distribute the source code and/or executable version(s) of Covered Code under (1) this License; (2) a license identical to this License but for only such changes as are necessary in order to clarify Recipient's role as licensor of Modifications; and/or (3) a license of Recipient's choosing, containing terms different from this License, provided that the license terms include this Section 3 and Sections 4, 6, 7, 10, 12, and 13, which terms may not be modified or superseded by any other terms of such license. If Recipient elects to use any license other than this License, Recipient must make it absolutely clear that any of its terms which differ from this License are offered by Recipient alone, and not by SGI. It is emphasized that this License is a limited license, and, regardless of the license form employed by Recipient in accordance with this Section 3.2, Recipient may relicense only such rights, in Original Code and Modifications by SGI, as it has actually been granted by SGI in this License.

3.3. **Indemnity.** Recipient hereby agrees to indemnify SGI for any liability incurred by SGI as a result of any such alternative license terms Recipient offers.

4. **Termination.** This License and the rights granted hereunder will terminate automatically if Recipient breaches any term herein and fails to cure such breach within 30 days thereof. Any sublicense to the Covered Code that is properly granted shall survive any termination of this License, absent termination by the terms of such sublicense. Provisions that, by their nature, must remain in effect beyond the termination of this License, shall survive.

5. **No Trademark Or Other Rights.** This License does not grant any rights to: (i) any software apart from the Covered Code, nor shall any other rights or licenses not expressly granted hereunder arise by implication, estoppel or otherwise with respect to the Covered Code; (ii) any trade name, trademark or service mark whatsoever, including without limitation any related right for purposes of endorsement or promotion of products derived from the Covered

Code, without prior written permission of SGI; or (iii) any title to or ownership of the Original Code, which shall at all times remains with SGI. All rights in the Original Code not expressly granted under this License are reserved.

6. **Compliance with Laws; Non-Infringement.** There are various worldwide laws, regulations, and executive orders applicable to dispositions of Covered Code, including without limitation export, re-export, and import control laws, regulations, and executive orders, of the U.S. government and other countries, and Recipient is reminded it is obliged to obey such laws, regulations, and executive orders. Recipient may not distribute Covered Code that (i) in any way infringes (directly or contributorily) any intellectual property rights of any kind of any other person or entity or (ii) breaches any representation or warranty, express, implied or statutory, to which, under any applicable law, it might be deemed to have been subject.

7. **Claims of Infringement.** If Recipient learns of any third party claim that any disposition of Covered Code and/or functionality wholly or partially infringes the third party's intellectual property rights, Recipient will promptly notify SGI of such claim.

8. **Versions of the License.** SGI may publish revised and/or new versions of the License from time to time, each with a distinguishing version number. Once Covered Code has been published under a particular version of the License, Recipient may, for the duration of the license, continue to use it under the terms of that version, or choose to use such Covered Code under the terms of any subsequent version published by SGI. Subject to the provisions of Sections 3 and 4 of this License, only SGI may modify the terms applicable to Covered Code created under this License.

9. **DISCLAIMER OF WARRANTY.** COVERED CODE IS PROVIDED "AS IS." ALL EXPRESS AND IMPLIED WARRANTIES AND CONDITIONS ARE DISCLAIMED, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES AND CONDITIONS OF MERCHANTABILITY, SATISFACTORY QUALITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. SGI ASSUMES NO RISK AS TO THE QUALITY AND PERFORMANCE OF THE SOFTWARE. SHOULD THE SOFTWARE PROVE DEFECTIVE IN ANY RESPECT, SGI ASSUMES

NO COST OR LIABILITY FOR SERVICING, REPAIR OR CORRECTION. THIS DISCLAIMER OF WARRANTY IS AN ESSENTIAL PART OF THIS LICENSE. NO USE OF ANY COVERED CODE IS AUTHORIZED HEREUNDER EXCEPT SUBJECT TO THIS DISCLAIMER.

10. **LIMITATION OF LIABILITY.** UNDER NO CIRCUMSTANCES NOR LEGAL THEORY, WHETHER TORT (INCLUDING, WITHOUT LIMITATION, NEGLIGENCE OR STRICT LIABILITY), CONTRACT, OR OTHERWISE, SHALL SGI OR ANY SGI LICENSOR BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY CHARACTER INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF GOODWILL, WORK STOPPAGE, LOSS OF DATA, COMPUTER FAILURE OR MALFUNCTION, OR ANY AND ALL OTHER COMMERCIAL DAMAGES OR LOSSES, EVEN IF SUCH PARTY SHALL HAVE BEEN INFORMED OF THE POSSIBILITY OF SUCH DAMAGES. THIS LIMITATION OF LIABILITY SHALL NOT APPLY TO LIABILITY FOR DEATH OR PERSONAL INJURY RESULTING FROM SGI's NEGLIGENCE TO THE EXTENT APPLICABLE LAW PROHIBITS SUCH LIMITATION. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THAT EXCLUSION AND LIMITATION MAY NOT APPLY TO RECIPIENT.

11. **Indemnity.** Recipient shall be solely responsible for damages arising, directly or indirectly, out of its utilization of rights under this License. Recipient will defend, indemnify and hold harmless Silicon Graphics, Inc. from and against any loss, liability, damages, costs or expenses (including the payment of reasonable attorneys fees) arising out of Recipient's use, modification, reproduction and distribution of the Covered Code or out of any representation or warranty made by Recipient.

12. **U.S. Government End Users.** The Covered Code is a "commercial item" consisting of "commercial computer software" as such terms are defined in title 48 of the Code of Federal Regulations and all U.S. Government End Users acquire only the rights set forth in this License and are subject to the terms of this License.

13. **Miscellaneous.** This License represents the complete agreement concerning the its subject matter. If any provision of this License is held to be unenforceable, such provision shall be reformed so as to achieve as nearly as possible the same legal and economic effect as the original provision and the remainder of this License will remain in effect. This License shall be governed by and construed in accordance with the laws of the United States and the State of California as applied to agreements entered into and to be performed entirely within California between California residents. Any litigation relating to this License shall be subject to the exclusive jurisdiction of the Federal Courts of the Northern District of California (or, absent subject matter jurisdiction in such courts, the courts of the State of California), with venue lying exclusively in Santa Clara County, California, with the losing party responsible for costs, including without limitation, court costs and reasonable attorneys fees and expenses. The application of the United Nations Convention on Contracts for the International Sale of Goods is expressly excluded. Any law or regulation that provides that the language of a contract shall be construed against the drafter shall not apply to this License.

Exhibit A

License Applicability. Except to the extent portions of this file are made subject to an alternative license as permitted in the SGI Free Software License B, Version 1.1 (the "License"), the contents of this file are subject only to the provisions of the License. You may not use this file except in compliance with the License. You may obtain a copy of the License at Silicon Graphics, Inc., attn: Legal Services, 1600 Amphitheatre Parkway, Mountain View, CA 94043-1351, or at:

http://oss.sgi.com/projects/FreeB

Note that, as provided in the License, the Software is distributed on an "AS IS" basis, with ALL EXPRESS AND IMPLIED WARRANTIES AND CONDITIONS DISCLAIMED, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES AND CONDITIONS OF MERCHANTABILITY, SATISFACTORY QUALITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT.

SGI Free Software License B