Guile-OpenGL

version 0.1.0, updated 23 March 2014

This manual is for Guile-OpenGL (version 0.1.0, updated 23 March 2014)

Short Contents

1 Introduction

Guile-OpenGL is Guile's interface to OpenGL.

In addition to the OpenGL API, Guile also provides access to related libraries and toolkits such as GLU, GLX, and GLUT. The following chapters discuss the parts of OpenGL and how Guile binds them.

But before that, some notes on the binding as a whole.

1.1 About

Guile-OpenGL uses the dynamic *foreign function interface* provided by Guile 2.0, providing access to OpenGL without any C code at all. In fact, much of Guile-OpenGL (and this manual) is automatically generated from upstream API specifications and documentation.

We have tried to do a very complete job at wrapping OpenGL, and additionally have tried to provide a nice Scheme interface as well. Our strategy has been to separate the binding into low-level and high-level pieces.

The low-level bindings correspond exactly with the OpenGL specification, and are well-documented. However, these interfaces are not so nice to use from Scheme; output arguments have to be allocated by the caller, and there is only the most basic level of type checking, and no sanity checking at all. For example, you can pass a bytevector of image data to the low-level `glTexImage2D` procedure, but no check is made that the dimensions you specify actually correspond to the size of the bytevector. This function could end up reading past the end of the bytevector. Worse things can happen with procedures that write to arrays, like `glGetTexImage`.

The high-level bindings are currently a work in progress, and are being manually written. They intend to be a complete interface to the OpenGL API, without the need to use the low-level bindings. However, the low-level bindings will always be available for you to use if needed, and have the advantage that their behavior is better documented and specified by OpenGL itself.

Low-level bindings are accessed by loading the `(`*`module`* `low-level)`, for example via:

```
(use-modules (gl low-level))
```

The high-level modules are named like `(`*`module`*`)`, for example `(gl)`.

2 API Conventions

FIXME: A very rough draft. Bindings and text are not fully synced until more work is done here.

This chapter documents the general conventions used by the low-level and high-level bindings. Any conventions specific to a particular module are documented in the relevent section.

As Guile-OpenGL is in very early stages of development these conventions are subject to change. Feedback is certainly welcome, and nothing is set in stone.

2.1 Enumerations

The OpenGL API defines many *symbolic constants*, most of which are collected together as named *enumerations* or *bitfields*. Access to these constants is the same for the low-level bindings and high-level interface.

For each OpenGL enumeration type, there is a similarly named Scheme type whose constructor takes an unquoted Scheme symbol naming one of the values. Guile-OpenGL translates the names to a more common Scheme style:

- any API prefix is removed (for example, GL_); and
- all names are lowercase, with underscores and CamelCase replaced by hyphens.

For example, the OpenGL API defines an enumeration with symbolic constants whose C names are GL_POINTS, GL_LINES, GL_TRIANGLES, and so on. Collectively they form the BeginMode enumeration type. To access these constants in Guile, apply the constant name to the enumeration type: `(begin-mode triangles)`.

Bitfields are similar, though the constructor accepts multiple symbols and produces an appropriate mask. In the GLUT API there is the DisplayMode bitfield, with symbolic constants GLUT_RGB, GLUT_INDEX, GLUT_SINGLE, and so on. To create a mask representing a double-buffered, rgb display-mode with a depth buffer: `(display-mode double rgb depth)`.

Enumeration and bitfield values, once constructed, can be compared using `eqv?`. For example, to determine if `modelview` is the current matrix mode use `(eqv? (gl-matrix-mode) (matrix-mode modelview))`.

2.2 Functions

The low-level bindings currently use names identical to their C API counterparts.

High-level bindings adopt names that are closer to natural language, and a more common style for Scheme:

- the API prefix is always removed;
- abbreviations are avoided; and
- names are all lowercase with words separated by hyphens.

Some function names are altered in additional ways, to make clear which object is being operated on. Functions that mutate objects or state will have their name prefixed with `set-`, such as `set-matrix-mode`.

FIXME: This choice may be too unnatural for GL users.

Where the C API specifies multiple functions that perform a similar task on varying number and types of arguments, the high-level bindings provide a single function that takes optional arguments, and, where appropriate, using only the most natural type. Consider the group of C API functions including `glVertex2f`, `glVertex3f`, and so on; the high-level GL interface provides only a single function `glVertex` with optional arguments.

The high-level interfaces may differ in other ways, and it is important to refer to the specific documentation.

It is generally fine to intermix functions from corresponding low-level and high-level bindings. This can be useful if you know the specific type of data you are working with and want to avoid the overhead of dynamic dispatch at runtime. Any cases where such intermixing causes problems will be noted in the documentation for the high-level bindings.

3 GL

3.1 About OpenGL

The OpenGL API is a standard interface for drawing three-dimensional graphics. From its origin in Silicon Graphics's workstations the early 1990s, today it has become ubiquitous, with implementations on mobile phones, televisions, tablets, desktops, and even web browsers.

OpenGL has been able to achieve such widespread adoption not just because it co-evolved with powerful graphics hardware, but also because it was conceived of as an interface specification and not a piece of source code. In fact, these days it is a family of APIs, available in several flavors and versions:

OpenGL 1.x
: This series of specifications started with the original releases in 1992, and ended with OpenGL 1.5 in 2003. This era corresponds to a time when graphics cards were less powerful and more special-purpose, with dedicated hardware to handle such details as fog and lighting. As such the OpenGL 1.x API reflects the capabilities of these special units.

OpenGL 2.x
: By the early 2000s, graphics hardware had become much more general-purpose and needed a more general-purpose API. The so-called *fixed-function rendering pipeline* of the earlier years was replaced with a *programmable rendering pipeline*, in which effects that would have required special hardware were instead performed by custom programs running on the graphics card. OpenGL added support for allocating *buffer objects* on the graphics card, and for *shader programs*, which did the actual rendering. In time, this buffer-focused API came to be the preferred form of talking to the GL.

OpenGL ES
: OpenGL ES was a "cut-down" version of OpenGL 2.x, designed to be small enough to appeal to embedded device vendors. OpenGL ES 1.x removed some of the legacy functionality from OpenGL, while adding interfaces to use fixed-point math, for devices without floating-point units. OpenGL ES 2.x went farther still, removing the fixed-function pipeline entirely. OpenGL ES 2.x is common on current smart phone platforms.

OpenGL 3.x and above
: The OpenGL 3.x series followed the lead of OpenGL ES, first deprecating (in 3.0) and then removing (in 3.1) the fixed-function pipeline. OpenGL 3.0 was released in 2008, but the free Mesa impementation only began supporting it in 2012, so it is currently (23 March 2014) less common.

Guile wraps the OpenGL 2.1 API. It's a ubiquitous subset of the OpenGL implementations that are actually deployed in the wild; its legacy API looks back to OpenGL 1.x, while the buffer-oriented API is compatible with OpenGL ES.

The full OpenGL 2.1 specification is available at `http://www.opengl.org/registry/doc/glspec21.20061201.pdf`.

3.2 GL Contexts

All this talk about drawing is very well and good, but how do you actually get a canvas? Interestingly enough, this is outside the purview of the OpenGL specification. There are specific ways to get an *OpenGL context* for each different windowing system that is out there. OpenGL is all crayons and no paper.

For the X window system, there is a standard API for creating a GL context given a window (or a drawable), *GLX*. See Chapter 5 [GLX], page 460, for more information on its binding in Guile.

Bseides creating contexts from native windows or drawables, each backend also supports functions to make a context *current*. The OpenGL API is stateful; you can think of each call as taking an implicit *current context* parameter, which holds the current state of the GL and is operated on by the function in question. Contexts are thread-specific, and one context should not be active on more than one thread at a time.

All calls to OpenGL functions must be made while a context is active; otherwise the result is undefined. Hopefully while you are getting used to this rule, your driver is nice enough not to crash on you if you call a function outside a GL context, but it's not even required to do that. Backend-specific functions may or may not require a context to be current; for example, Windows requires a context to be current, wheras GLX does not.

There have been a few attempts at abstracting away the need for calling API specific to a given windowing system, notably GLUT and EGL. GLUT is the older of the two, and though it is practically unchanged since the mid-1990s, it is still widely used on desktops. See Chapter 6 [GLUT], page 495, for more on GLUT.

EGL is technically part of OpenGL ES, and was designed with the modern OpenGL API and mobile hardware in mind, though it also works on the desktop. Guile does not yet have an EGL binding.

3.3 Rendering

To draw with OpenGL, you obtain a drawing context (see Section 3.2 [GL Contexts], page 5) and send *the GL* some geometry. (You can think of the GL as a layer over your graphics card.) You can give the GL points, lines, and triangles in three-dimensional space. You configure your GL to render a certain part of space, and it takes your geometry, rasterizes it, and writes it to the screen (when you tell it to).

That's the basic idea. You can customize most parts of this *rendering pipeline*, by specifying attributes of your geometry with the OpenGL API, and by programmatically operating on the geometry and the pixels with programs called *shaders*.

GL is an *immediate-mode* graphics API, which is to say that it doesn't keep around a scene graph of objects. Instead, at every frame you as the OpenGL user have to tell the GL what is in the world, and how to paint it. It's a fairly low-level interface, but a powerful one. See `http://www.opengl.org/wiki/Rendering_Pipeline_Overview`, for more details.

In the old days of OpenGL 1.0, it was common to call a function to paint each individual vertex. You'll still see this style in some old tutorials. This quickly gets expensive if you have a lot of vertexes, though. This style, known as *Legacy OpenGL*, was deprecated and even removed from some versions of OpenGL. See `http://www.opengl.org/wiki/Legacy_OpenGL`, for more on the older APIs.

Instead, the newer thing to do is to send the geometry to the GL in a big array buffer, and have the GL draw geometry from the buffer. The newer functions like `glGenBuffers` allocate buffers, returning an integer that *names* a buffer managed by the GL. You as a user can update the contents of the buffer, but when drawing you reference the buffer by name. This has the advantage of reducing the chatter and data transfer between you and the GL, though it can be less convenient to use.

So which API should you use? Use what you feel like using, if you have a choice. Legacy OpenGL isn't going away any time soon on the desktop. Sometimes you don't have a choice, though; for example, when targeting a device that only supports OpenGL ES 2.x, legacy OpenGL is unavailable.

But if you want some advice, we suggest that you use the newer APIs. Not only will your code be future-proof and more efficient on the GL level, reducing the number of API calls improves performance, and it can reduce the amount of heap allocation in your program. All floating-point numbers are currently allocated on the heap in Guile, and doing less floating-point math in tight loops can only be a good thing.

3.4 GL API

The procedures exported from the `(gl)` module are documented below, organized by their corresponding section in the OpenGL 2.1 specification.

```
(use-modules (gl))
```

See http://www.opengl.org/registry/doc/glspec21.20061201.pdf, for more information.

3.4.1 OpenGL Operation

3.4.1.1 Begin/End Paradigm

`gl-begin` *begin-mode body ...* [Macro]

Begin immediate-mode drawing with *begin-mode*, evaluate the sequence of *body* expressions, and then end drawing (as with `glBegin` and `glEnd`).

The values produced by the last *body* expression are returned to the continuation of the `gl-begin`.

`gl-edge-flag` *boundary?* [Function]

Flag edges as either boundary or nonboundary. Note that the edge mode is only significant if the `polygon-mode` is `line` or `point`.

3.4.1.2 Vertex Specification

`gl-vertex` *x y* [*z=0.0*] [*w=1.0*] [Function]

Draw a vertex at the given coordinates.

The following procedures modify the current per-vertex state. Drawing a vertex captures the current state and associates it with the vertex.

`gl-texture-coordinates` *s* [*t=0.0*] [*r=0.0*] [*q=1.0*] [Function]

Set the current texture coordinate.

`gl-multi-texture-coordinates` *texture s* [*t=0.0*] [*r=0.0*] [*q=1.0*] [Function]
Set the current texture coordinate for a specific texture unit.

`gl-color` *red green blue* [*alpha=1.0*] [Function]
Set the current color.

`gl-vertex-attribute` *index x* [*y=0.0*] [*z=0.0*] [*w=1.0*] [Function]
Set the current value of a generic vertex attribute.

`gl-normal` *x y z* [Function]
Set the current normal vector. By default the normal should have unit length, though setting `(enable-cap rescale-normal)` or `(enable-cap normalize)` can change this.

`gl-fog-coordinate` *coord* [Function]
Set the current fog coordinate.

`gl-secondary-color` *red green blue* [Function]
Set the current secondary color.

`gl-index` *c* [Function]
Set the current color index.

3.4.1.3 Rectangles

`gl-rectangle` *x1 y1 x2 y2* [Function]
Draw a rectangle in immediate-mode with a given pair of corner points.

3.4.1.4 Coordinate Transformation

`gl-depth-range` *near-val far-val* [Function]
Specify the mapping of the near and far clipping planes, respectively, to window coordinates.

`gl-viewport` *x y width height* [Function]
Set the viewport: the pixel position of the lower-left corner of the viewport rectangle, and the width and height of the viewport.

`gl-load-matrix` *m* [*#:transpose=#f*] [Function]
Load a matrix. *m* should be a packed vector in column-major order.

Note that Guile's two-dimensional arrays are stored in row-major order, so you might need to transpose the matrix as it is loaded (via the `#:transpose` keyword argument).

`gl-multiply-matrix` *m* [*#:transpose=#f*] [Function]
Multiply the current matrix by *m*. As with `gl-load-matrix`, you might need to transpose the matrix first.

`set-gl-matrix-mode` *matrix-mode* [Function]
Set the current matrix mode. See the `matrix-mode` enumerator.

`with-gl-push-matrix` *body ...* [Macro]
Save the current matrix, evaluate the sequence of *body* expressions, and restore the saved matrix.

`gl-load-identity` [Function]
Load the identity matrix.

`gl-rotate` *angle x y z* [Function]
Rotate the current matrix about the vector `(x,y,z)`. *angle* should be specified in degrees.

`gl-translate` *x y z* [Function]
Translate the current matrix.

`gl-scale` *x y z* [Function]
Scale the current matrix.

`gl-frustum` *left right bottom top near-val far-val* [Function]
Multiply the current matrix by a perspective matrix. *left*, *right*, *bottom*, and *top* are the coordinates of the corresponding clipping planes. *near-val* and *far-val* specify the distances to the near and far clipping planes.

`gl-ortho` *left right bottom top near-val far-val* [Function]
Multiply the current matrix by a perspective matrix. *left*, *right*, *bottom*, and *top* are the coordinates of the corresponding clipping planes. *near-val* and *far-val* specify the distances to the near and far clipping planes.

`set-gl-active-texture` *texture* [Function]
Set the active texture unit.

`gl-enable` *enable-cap* [Function]
`gl-disable` *enable-cap* [Function]
Enable or disable server-side GL capabilities.

3.4.1.5 Colors and Coloring

`set-gl-shade-model` *mode* [Function]
Select flat or smooth shading.

3.4.2 Rasterization

3.4.3 Per-Fragment Operations

`set-gl-stencil-function` *stencil-function k* [*#:mask*] [*#:face*] [Function]
Set the front and/or back function and the reference value *k* for stencil testing. Without the *face* keyword argument, both functions are set. The default *mask* is all-inclusive.

`set-gl-stencil-operation` *stencil-fail depth-fail depth-pass* [*#:face*] [Function]
Set the front and/or back stencil test actions. Without the *face* keyword argument, both stencil test actions are set. See the `stencil-op` enumeration for possible values for *stencil-fail*, *depth-fail*, and *depth-pass*.

`set-gl-blend-equation` *mode-rgb* [*mode-alpha=mode-rgb*] [Function]
Set the blend equation. With one argument, set the same blend equation for all components. Pass two arguments to specify a separate equation for the alpha component.

`set-gl-blend-function` *src-rgb dest-rgb* [*src-alpha=src-rgb*] [*dest-alpha=dest-rgb*] [Function]

Set the blend function. With two arguments, set the same blend function for all components. Pass an additional two arguments to specify separate functions for the alpha components.

`set-gl-scissor` *x y width height* [Function]

Define the scissor box. The box is defined in window coordinates, with (*x*,*y*) being the lower-left corner of the box.

`set-gl-sample-coverage` *value invert* [Function]

Specify multisample coverage parameters.

`set-gl-alpha-function` *func ref* [Function]

Specify the alpha test function. See the `alpha-function` enumerator.

`set-gl-depth-function` *func* [Function]

Specify the depth test function. See the `depth-function` enumerator.

`set-gl-blend-color` *r g b a* [Function]

Specify the blend color.

`set-gl-logic-operation` *opcode* [Function]

Specify a logical pixel operation for color index rendering.

3.4.3.1 Whole Framebuffer Operations

`set-gl-draw-buffers` *buffers* [Function]

Specify a list of color buffers to be drawn into. *buffers* should be a list of `draw-buffer-mode` enumerated values.

`set-gl-stencil-mask` *mask* [*#:face*] [Function]

Control the writing of individual bits into the front and/or back stencil planes. With one argument, the stencil mask for both states are set.

`set-gl-draw-buffer` *mode* [Function]

Specify the buffer or buffers to draw into.

`set-gl-index-mask` *mask* [Function]

Control the writing of individual bits into the color index buffers.

`set-gl-color-mask` *red? green? blue? alpha?* [Function]

Enable and disable writing of frame buffer color components.

`set-gl-depth-mask` *enable?* [Function]

Enable and disable writing into the depth buffer.

`gl-clear` *mask* [Function]

Clear a set of buffers to pre-set values. Use the `clear-buffer-mask` enumerator to specify which buffers to clear.

`set-gl-clear-color` *r g b a* [Function]
Set the clear color for the color buffers.

`set-gl-clear-index` *c* [Function]
Set the clear index for the color index buffers.

`set-gl-clear-depth` *depth* [Function]
Set the clear value for the depth buffer.

`set-gl-clear-stencil-value` *s* [Function]
Set the clear value for the stencil buffer.

`set-gl-clear-accumulation-color` *r g b a* [Function]
Set the clear color for the accumulation buffer.

`set-gl-accumulation-buffer-operation` *op value* [Function]
Operate on the accumulation buffer. *op* may be one of the `accum-op` enumerated values. The interpretation of *value* depends on *op*.

3.4.3.2 Drawing, Reading and Copying Pixels

`set-gl-read-buffer` *mode* [Function]
Select a color buffer source for pixels. Use `read-buffer-mode` to select a mode.

`gl-copy-pixels` *x y width height type* [Function]
Copy pixels from a screen-aligned rectangle in the frame buffer to a region relative to the current raster position. *type* selects which buffer to copy from.

3.4.4 Special Functions

3.4.5 State and State Requests

3.4.5.1 Querying GL State

`with-gl-push-attrib` *bits body ...* [Macro]
Save part of the current state, evaluation the sequence of *body* expressions, then restore the state. Use `attrib-mask` to specify which parts of the state to save.

3.5 GL Enumerations

The functions from this section may be had by loading the module:

```
(use-modules (gl enums)
```

`attrib-mask` *bit...* [Macro]
Bitfield constructor. The symbolic *bit* arguments are replaced with their corresponding numeric values and combined with `logior` at compile-time. The symbolic arguments known to this bitfield constructor are:

`current`, `point`, `line`, `polygon`, `polygon-stipple`, `pixel-mode`, `lighting`, `fog`, `depth-buffer`, `accum-buffer`, `stencil-buffer`, `viewport`, `transform`, `enable`, `color-buffer`, `hint`, `eval`, `list`, `texture`, `scissor`, `all-attrib`.

`version-1-3` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`multisample-bit`, `multisample`, `sample-alpha-to-coverage`, `sample-alpha-to-one`, `sample-coverage`, `sample-buffers`, `samples`, `sample-coverage-value`, `sample-coverage-invert`, `clamp-to-border`, `texture0`, `texture1`, `texture2`, `texture3`, `texture4`, `texture5`, `texture6`, `texture7`, `texture8`, `texture9`, `texture10`, `texture11`, `texture12`, `texture13`, `texture14`, `texture15`, `texture16`, `texture17`, `texture18`, `texture19`, `texture20`, `texture21`, `texture22`, `texture23`, `texture24`, `texture25`, `texture26`, `texture27`, `texture28`, `texture29`, `texture30`, `texture31`, `active-texture`, `client-active-texture`, `max-texture-units`, `transpose-modelview-matrix`, `transpose-projection-matrix`, `transpose-texture-matrix`, `transpose-color-matrix`, `subtract`, `compressed-alpha`, `compressed-luminance`, `compressed-luminance-alpha`, `compressed-intensity`, `compressed-rgb`, `compressed-rgba`, `texture-compression-hint`, `texture-compressed-image-size`, `texture-compressed`, `num-compressed-texture-formats`, `compressed-texture-formats`, `normal-map`, `reflection-map`, `texture-cube-map`, `texture-binding-cube-map`, `texture-cube-map-positive-x`, `texture-cube-map-negative-x`, `texture-cube-map-positive-y`, `texture-cube-map-negative-y`, `texture-cube-map-positive-z`, `texture-cube-map-negative-z`, `proxy-texture-cube-map`, `max-cube-map-texture-size`, `combine`, `combine-rgb`, `combine-alpha`, `rgb-scale`, `add-signed`, `interpolate`, `constant`, `primary-color`, `previous`, `source0-rgb`, `source1-rgb`, `source2-rgb`, `source0-alpha`, `source1-alpha`, `source2-alpha`, `operand0-rgb`, `operand1-rgb`, `operand2-rgb`, `operand0-alpha`, `operand1-alpha`, `operand2-alpha`, `dot3-rgb`, `dot3-rgba`.

`arb-multisample` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`multisample-bit-arb`, `multisample-arb`, `sample-alpha-to-coverage-arb`, `sample-alpha-to-one-arb`, `sample-coverage-arb`, `sample-buffers-arb`, `samples-arb`, `sample-coverage-value-arb`, `sample-coverage-invert-arb`.

`ext-multisample` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`multisample-bit-ext`, `multisample-ext`, `sample-alpha-to-mask-ext`, `sample-alpha-to-one-ext`, `sample-mask-ext`, `1pass-ext`, `2pass-0-ext`, `2pass-1-ext`, `4pass-0-ext`, `4pass-1-ext`, `4pass-2-ext`, `4pass-3-ext`, `sample-buffers-ext`, `samples-ext`, `sample-mask-value-ext`, `sample-mask-invert-ext`, `sample-pattern-ext`, `multisample-bit-ext`.

`3dfx-multisample` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`multisample-bit-3dfx`, `multisample-3dfx`, `sample-buffers-3dfx`, `samples-3dfx`, `multisample-bit-3dfx`.

`clear-buffer-mask` *bit...* [Macro]

Bitfield constructor. The symbolic *bit* arguments are replaced with their corresponding numeric values and combined with `logior` at compile-time. The symbolic arguments known to this bitfield constructor are:

`depth-buffer`, `accum-buffer`, `stencil-buffer`, `color-buffer`, `coverage-buffer-bit-nv`.

`client-attrib-mask` *bit...* [Macro]

Bitfield constructor. The symbolic *bit* arguments are replaced with their corresponding numeric values and combined with `logior` at compile-time. The symbolic arguments known to this bitfield constructor are:

`client-pixel-store`, `client-vertex-array`, `client-all-attrib`.

`version-3-0` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`map-read-bit`, `map-write-bit`, `map-invalidate-range-bit`, `map-invalidate-buffer-bit`, `map-flush-explicit-bit`, `map-unsynchronized-bit`, `context-flag-forward-compatible-bit`, `invalid-framebuffer-operation`, `half-float`, `clip-distance0`, `clip-distance1`, `clip-distance2`, `clip-distance3`, `clip-distance4`, `clip-distance5`, `clip-distance6`, `clip-distance7`, `framebuffer-attachment-color-encoding`, `framebuffer-attachment-component-type`, `framebuffer-attachment-red-size`, `framebuffer-attachment-green-size`, `framebuffer-attachment-blue-size`, `framebuffer-attachment-alpha-size`, `framebuffer-attachment-depth-size`, `framebuffer-attachment-stencil-size`, `framebuffer-default`, `framebuffer-undefined`, `depth-stencil-attachment`, `major-version`, `minor-version`, `num-extensions`, `context-flags`, `index`, `compressed-red`, `compressed-rg`, `rg`, `rg-integer`, `r8`, `r16`, `rg8`, `rg16`, `r16f`, `r32f`, `rg16f`, `rg32f`, `r8i`, `r8ui`, `r16i`, `r16ui`, `r32i`, `r32ui`, `rg8i`, `rg8ui`, `rg16i`, `rg16ui`, `rg32i`, `rg32ui`, `max-renderbuffer-size`, `depth-stencil`, `unsigned-int-24-8`, `vertex-array-binding`, `rgba32f`, `rgb32f`, `rgba16f`, `rgb16f`, `compare-ref-to-texture`, `depth24-stencil8`, `texture-stencil-size`, `vertex-attrib-array-integer`, `max-array-texture-layers`, `min-program-texel-offset`, `max-program-texel-offset`, `clamp-vertex-color`, `clamp-fragment-color`, `clamp-read-color`, `fixed-only`, `max-varying-components`, `texture-red-type`, `texture-green-type`, `texture-blue-type`, `texture-alpha-type`, `texture-luminance-type`, `texture-intensity-type`, `texture-depth-type`, `unsigned-normalized`, `texture-1d-array`, `proxy-texture-1d-array`, `texture-2d-array`, `proxy-texture-2d-array`,

texture-binding-1d-array, texture-binding-2d-array, r11f-g11f-b10f, unsigned-int-10f-11f-11f-rev, rgb9-e5, unsigned-int-5-9-9-9-rev, texture-shared-size, transform-feedback-varying-max-length, transform-feedback-varying-max-length-ext, back-primary-color-nv, back-secondary-color-nv, texture-coord-nv, clip-distance-nv, vertex-id-nv, primitive-id-nv, generic-attrib-nv, transform-feedback-attribs-nv, transform-feedback-buffer-mode, transform-feedback-buffer-mode-ext, transform-feedback-buffer-mode-nv, max-transform-feedback-separate-components, max-transform-feedback-separate-components-ext, max-transform-feedback-separate-components-nv, active-varyings-nv, active-varying-max-length-nv, transform-feedback-varyings, transform-feedback-varyings-ext, transform-feedback-varyings-nv, transform-feedback-buffer-start, transform-feedback-buffer-start-ext, transform-feedback-buffer-start-nv, transform-feedback-buffer-size, transform-feedback-buffer-size-ext, transform-feedback-buffer-size-nv, transform-feedback-record-nv, primitives-generated, primitives-generated-ext, primitives-generated-nv, transform-feedback-primitives-written, transform-feedback-primitives-written-ext, transform-feedback-primitives-written-nv, rasterizer-discard, rasterizer-discard-ext, rasterizer-discard-nv, max-transform-feedback-interleaved-components, max-transform-feedback-interleaved-components-ext, max-transform-feedback-interleaved-components-nv, max-transform-feedback-separate-attribs, max-transform-feedback-separate-attribs-ext, max-transform-feedback-separate-attribs-nv, interleaved-attribs, interleaved-attribs-ext, interleaved-attribs-nv, separate-attribs, separate-attribs-ext, separate-attribs-nv, transform-feedback-buffer, transform-feedback-buffer-ext, transform-feedback-buffer-nv, transform-feedback-buffer-binding, transform-feedback-buffer-binding-ext, transform-feedback-buffer-binding-nv, framebuffer-binding, draw-framebuffer-binding, renderbuffer-binding, read-framebuffer, draw-framebuffer, read-framebuffer-binding, renderbuffer-samples, depth-component32f, depth32f-stencil8, framebuffer-attachment-object-type, framebuffer-attachment-object-type-ext, framebuffer-attachment-object-name, framebuffer-attachment-object-name-ext, framebuffer-attachment-texture-level, framebuffer-attachment-texture-level-ext, framebuffer-attachment-texture-cube-map-face, framebuffer-attachment-texture-cube-map-face-ext, framebuffer-attachment-texture-layer, framebuffer-attachment-texture-3d-zoffset-ext, framebuffer-complete, framebuffer-complete-ext, framebuffer-incomplete-attachment, framebuffer-incomplete-attachment-ext, framebuffer-incomplete-missing-attachment, framebuffer-incomplete-missing-attachment-ext, framebuffer-incomplete-dimensions-ext, framebuffer-incomplete-formats-ext, framebuffer-incomplete-draw-buffer, framebuffer-incomplete-draw-buffer-ext, framebuffer-incomplete-read-buffer, framebuffer-incomplete-read-buffer-ext, framebuffer-unsupported, framebuffer-unsupported-ext, max-color-attachments, max-color-attachments-ext, color-attachment0, color-attachment0-ext, color-attachment1, color-attachment1-

ext, color-attachment2, color-attachment2-ext, color-attachment3, color-attachment3-ext, color-attachment4, color-attachment4-ext, color-attachment5, color-attachment5-ext, color-attachment6, color-attachment6-ext, color-attachment7, color-attachment7-ext, color-attachment8, color-attachment8-ext, color-attachment9, color-attachment9-ext, color-attachment10, color-attachment10-ext, color-attachment11, color-attachment11-ext, color-attachment12, color-attachment12-ext, color-attachment13, color-attachment13-ext, color-attachment14, color-attachment14-ext, color-attachment15, color-attachment15-ext, depth-attachment, depth-attachment-ext, stencil-attachment, stencil-attachment-ext, framebuffer, framebuffer-ext, renderbuffer, renderbuffer-ext, renderbuffer-width, renderbuffer-width-ext, renderbuffer-height, renderbuffer-height-ext, renderbuffer-internal-format, renderbuffer-internal-format-ext, stencil-index1, stencil-index1-ext, stencil-index4, stencil-index4-ext, stencil-index8, stencil-index8-ext, stencil-index16, stencil-index16-ext, renderbuffer-red-size, renderbuffer-red-size-ext, renderbuffer-green-size, renderbuffer-green-size-ext, renderbuffer-blue-size, renderbuffer-blue-size-ext, renderbuffer-alpha-size, renderbuffer-alpha-size-ext, renderbuffer-depth-size, renderbuffer-depth-size-ext, renderbuffer-stencil-size, renderbuffer-stencil-size-ext, framebuffer-incomplete-multisample, max-samples, rgba32ui, rgba32ui-ext, rgb32ui, rgb32ui-ext, alpha32ui-ext, intensity32ui-ext, luminance32ui-ext, luminance-alpha32ui-ext, rgba16ui, rgba16ui-ext, rgb16ui, rgb16ui-ext, alpha16ui-ext, intensity16ui-ext, luminance16ui-ext, luminance-alpha16ui-ext, rgba8ui, rgba8ui-ext, rgb8ui, rgb8ui-ext, alpha8ui-ext, intensity8ui-ext, luminance8ui-ext, luminance-alpha8ui-ext, rgba32i, rgba32i-ext, rgb32i, rgb32i-ext, alpha32i-ext, intensity32i-ext, luminance32i-ext, luminance-alpha32i-ext, rgba16i, rgba16i-ext, rgb16i, rgb16i-ext, alpha16i-ext, intensity16i-ext, luminance16i-ext, luminance-alpha16i-ext, rgba8i, rgba8i-ext, rgb8i, rgb8i-ext, alpha8i-ext, intensity8i-ext, luminance8i-ext, luminance-alpha8i-ext, red-integer, red-integer-ext, green-integer, green-integer-ext, blue-integer, blue-integer-ext, alpha-integer, alpha-integer-ext, rgb-integer, rgb-integer-ext, rgba-integer, rgba-integer-ext, bgr-integer, bgr-integer-ext, bgra-integer, bgra-integer-ext, luminance-integer-ext, luminance-alpha-integer-ext, rgba-integer-mode-ext, float-32-unsigned-int-24-8-rev, framebuffer-srgb, compressed-red-rgtc1, compressed-signed-red-rgtc1, compressed-rg-rgtc2, compressed-signed-rg-rgtc2, sampler-1d-array, sampler-2d-array, sampler-1d-array-shadow, sampler-2d-array-shadow, sampler-cube-shadow, unsigned-int-vec2, unsigned-int-vec3, unsigned-int-vec4, int-sampler-1d, int-sampler-2d, int-sampler-3d, int-sampler-cube, int-sampler-1d-array, int-sampler-2d-array, unsigned-int-sampler-1d, unsigned-int-sampler-2d, unsigned-int-sampler-3d, unsigned-int-sampler-cube, unsigned-int-sampler-1d-array, unsigned-int-sampler-2d-array, query-wait, query-no-wait,

`query-by-region-wait`, `query-by-region-no-wait`, `buffer-access-flags`, `buffer-map-length`, `buffer-map-offset`.

`arb-map-buffer-range` *bit...* [Macro]
Bitfield constructor. The symbolic *bit* arguments are replaced with their corresponding numeric values and combined with `logior` at compile-time. The symbolic arguments known to this bitfield constructor are:

`map-read`, `map-write`, `map-invalidate-range`, `map-invalidate-buffer`, `map-flush-explicit`, `map-unsynchronized`.

`ext-map-buffer-range` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`map-read-bit-ext`, `map-write-bit-ext`, `map-invalidate-range-bit-ext`, `map-invalidate-buffer-bit-ext`, `map-flush-explicit-bit-ext`, `map-unsynchronized-bit-ext`.

`version-4-3` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`context-flag-debug-bit`, `num-shading-language-versions`, `vertex-attrib-array-long`.

`khr-debug` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`context-flag-debug-bit`, `debug-output-synchronous`, `debug-next-logged-message-length`, `debug-callback-function`, `debug-callback-user-param`, `debug-source-api`, `debug-source-window-system`, `debug-source-shader-compiler`, `debug-source-third-party`, `debug-source-application`, `debug-source-other`, `debug-type-error`, `debug-type-deprecated-behavior`, `debug-type-undefined-behavior`, `debug-type-portability`, `debug-type-performance`, `debug-type-other`, `debug-type-marker`, `debug-type-push-group`, `debug-type-pop-group`, `debug-severity-notification`, `max-debug-group-stack-depth`, `debug-group-stack-depth`, `buffer`, `shader`, `program`, `query`, `program-pipeline`, `sampler`, `display-list`, `max-label-length`, `max-debug-message-length`, `max-debug-logged-messages`, `debug-logged-messages`, `debug-severity-high`, `debug-severity-medium`, `debug-severity-low`, `debug-output`.

`arb-robustness` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`context-flag-robust-access-bit-arb`, `lose-context-on-reset-arb`, `guilty-context-reset-arb`, `innocent-context-reset-arb`, `unknown-context-reset-arb`, `reset-notification-strategy-arb`, `no-reset-notification-arb`.

`arb-separate-shader-objects` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`vertex-shader-bit`, `fragment-shader-bit`, `geometry-shader-bit`, `tess-control-shader-bit`, `tess-evaluation-shader-bit`, `all-shader-bits`, `program-separable`, `active-program`, `program-pipeline-binding`.

`arb-compute-shader` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`compute-shader-bit`, `max-compute-shared-memory-size`, `max-compute-uniform-components`, `max-compute-atomic-counter-buffers`, `max-compute-atomic-counters`, `max-combined-compute-uniform-components`, `compute-local-work-size`, `max-compute-local-invocations`, `uniform-block-referenced-by-compute-shader`, `atomic-counter-buffer-referenced-by-compute-shader`, `dispatch-indirect-buffer`, `dispatch-indirect-buffer-binding`, `compute-shader`, `max-compute-uniform-blocks`, `max-compute-texture-image-units`, `max-compute-image-uniforms`, `max-compute-work-group-count`, `max-compute-work-group-size`.

`ext-separate-shader-objects` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`vertex-shader-bit-ext`, `fragment-shader-bit-ext`, `all-shader-bits-ext`, `program-separable-ext`, `active-program-ext`, `program-pipeline-binding-ext`, `active-program-ext`.

`ext-shader-image-load-store` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`vertex-attrib-array-barrier-bit-ext`, `element-array-barrier-bit-ext`, `uniform-barrier-bit-ext`, `texture-fetch-barrier-bit-ext`, `shader-image-access-barrier-bit-ext`, `command-barrier-bit-ext`, `pixel-buffer-barrier-bit-ext`, `texture-update-barrier-bit-ext`, `buffer-update-barrier-bit-ext`, `framebuffer-barrier-bit-ext`, `transform-feedback-barrier-bit-ext`, `atomic-counter-barrier-bit-ext`, `all-barrier-bits-ext`, `max-image-units-ext`, `max-combined-image-units-and-fragment-outputs-ext`, `image-binding-name-ext`, `image-binding-level-ext`, `image-binding-layered-ext`, `image-binding-layer-ext`, `image-binding-access-ext`, `image-1d-ext`,

image-2d-ext, image-3d-ext, image-2d-rect-ext, image-cube-ext, image-buffer-ext, image-1d-array-ext, image-2d-array-ext, image-cube-map-array-ext, image-2d-multisample-ext, image-2d-multisample-array-ext, int-image-1d-ext, int-image-2d-ext, int-image-3d-ext, int-image-2d-rect-ext, int-image-cube-ext, int-image-buffer-ext, int-image-1d-array-ext, int-image-2d-array-ext, int-image-cube-map-array-ext, int-image-2d-multisample-ext, int-image-2d-multisample-array-ext, unsigned-int-image-1d-ext, unsigned-int-image-2d-ext, unsigned-int-image-3d-ext, unsigned-int-image-2d-rect-ext, unsigned-int-image-cube-ext, unsigned-int-image-buffer-ext, unsigned-int-image-1d-array-ext, unsigned-int-image-2d-array-ext, unsigned-int-image-cube-map-array-ext, unsigned-int-image-2d-multisample-ext, unsigned-int-image-2d-multisample-array-ext, max-image-samples-ext, image-binding-format-ext.

`arb-shader-image-load-store` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

vertex-attrib-array-barrier-bit, element-array-barrier-bit, uniform-barrier-bit, texture-fetch-barrier-bit, shader-image-access-barrier-bit, command-barrier-bit, pixel-buffer-barrier-bit, texture-update-barrier-bit, buffer-update-barrier-bit, framebuffer-barrier-bit, transform-feedback-barrier-bit, atomic-counter-barrier-bit, all-barrier-bits, max-image-units, max-combined-image-units-and-fragment-outputs, image-binding-name, image-binding-level, image-binding-layered, image-binding-layer, image-binding-access, image-1d, image-2d, image-3d, image-2d-rect, image-cube, image-buffer, image-1d-array, image-2d-array, image-cube-map-array, image-2d-multisample, image-2d-multisample-array, int-image-1d, int-image-2d, int-image-3d, int-image-2d-rect, int-image-cube, int-image-buffer, int-image-1d-array, int-image-2d-array, int-image-cube-map-array, int-image-2d-multisample, int-image-2d-multisample-array, unsigned-int-image-1d, unsigned-int-image-2d, unsigned-int-image-3d, unsigned-int-image-2d-rect, unsigned-int-image-cube, unsigned-int-image-buffer, unsigned-int-image-1d-array, unsigned-int-image-2d-array, unsigned-int-image-cube-map-array, unsigned-int-image-2d-multisample, unsigned-int-image-2d-multisample-array, max-image-samples, image-binding-format, image-format-compatibility-type, image-format-compatibility-by-size, image-format-compatibility-by-class, max-vertex-image-uniforms, max-tess-control-image-uniforms, max-tess-evaluation-image-uniforms, max-geometry-image-uniforms, max-fragment-image-uniforms, max-combined-image-uniforms.

`arb-shader-storage-buffer-object` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`shader-storage-barrier-bit`, `shader-storage-buffer`, `shader-storage-buffer-binding`, `shader-storage-buffer-start`, `shader-storage-buffer-size`, `max-vertex-shader-storage-blocks`, `max-geometry-shader-storage-blocks`, `max-tess-control-shader-storage-blocks`, `max-tess-evaluation-shader-storage-blocks`, `max-fragment-shader-storage-blocks`, `max-compute-shader-storage-blocks`, `max-combined-shader-storage-blocks`, `max-shader-storage-buffer-bindings`, `max-shader-storage-block-size`, `shader-storage-buffer-offset-alignment`, `max-combined-shader-output-resources`, `max-combined-image-units-and-fragment-outputs`.

`intel-map-texture` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`layout-default-intel`, `layout-linear-intel`, `layout-linear-cpu-cached-intel`, `texture-memory-layout-intel`.

`boolean` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`false`, `true`.

`begin-mode` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`points`, `lines`, `line-loop`, `line-strip`, `triangles`, `triangle-strip`, `triangle-fan`, `quads`, `quad-strip`, `polygon`.

`version-3-2` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`lines-adjacency`, `line-strip-adjacency`, `triangles-adjacency`, `triangle-strip-adjacency`, `program-point-size`, `depth-clamp`, `texture-cube-map-seamless`, `geometry-vertices-out`, `geometry-input-type`, `geometry-output-type`, `max-geometry-texture-image-units`, `framebuffer-attachment-layered`, `framebuffer-incomplete-layer-targets`, `geometry-shader`, `max-geometry-uniform-components`, `max-geometry-output-vertices`, `max-geometry-total-output-components`, `quads-follow-provoking-vertex-convention`, `first-vertex-convention`, `last-vertex-convention`, `provoking-vertex`, `sample-position`, `sample-mask`, `sample-mask-value`, `max-sample-mask-words`, `texture-2d-multisample`, `proxy-texture-2d-multisample`, `texture-2d-multisample-array`, `proxy-texture-2d-multisample-array`, `texture-binding-2d-multisample`, `texture-binding-2d-multisample-array`, `texture-samples`, `texture-fixed-sample-locations`,

sampler-2d-multisample, int-sampler-2d-multisample, unsigned-int-sampler-2d-multisample, sampler-2d-multisample-array, int-sampler-2d-multisample-array, unsigned-int-sampler-2d-multisample-array, max-color-texture-samples, max-depth-texture-samples, max-integer-samples, max-server-wait-timeout, object-type, sync-condition, sync-status, sync-flags, sync-fence, sync-gpu-commands-complete, unsignaled, signaled, already-signaled, timeout-expired, condition-satisfied, wait-failed, timeout-ignored, sync-flush-commands-bit, timeout-ignored, max-vertex-output-components, max-geometry-input-components, max-geometry-output-components, max-fragment-input-components, context-core-profile-bit, context-compatibility-profile-bit, context-profile-mask.

`arb-geometry-shader-4` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

lines-adjacency-arb, line-strip-adjacency-arb, triangles-adjacency-arb, triangle-strip-adjacency-arb, program-point-size-arb, max-varying-components, max-geometry-texture-image-units-arb, framebuffer-attachment-object-type, framebuffer-attachment-object-type-ext, framebuffer-attachment-object-name, framebuffer-attachment-object-name-ext, framebuffer-attachment-texture-level, framebuffer-attachment-texture-level-ext, framebuffer-attachment-texture-cube-map-face, framebuffer-attachment-texture-cube-map-face-ext, framebuffer-attachment-texture-layer, framebuffer-attachment-texture-3d-zoffset-ext, framebuffer-complete, framebuffer-complete-ext, framebuffer-incomplete-attachment, framebuffer-incomplete-attachment-ext, framebuffer-incomplete-missing-attachment, framebuffer-incomplete-missing-attachment-ext, framebuffer-incomplete-dimensions-ext, framebuffer-incomplete-formats-ext, framebuffer-incomplete-draw-buffer, framebuffer-incomplete-draw-buffer-ext, framebuffer-incomplete-read-buffer, framebuffer-incomplete-read-buffer-ext, framebuffer-unsupported, framebuffer-unsupported-ext, max-color-attachments, max-color-attachments-ext, color-attachment0, color-attachment0-ext, color-attachment1, color-attachment1-ext, color-attachment2, color-attachment2-ext, color-attachment3, color-attachment3-ext, color-attachment4, color-attachment4-ext, color-attachment5, color-attachment5-ext, color-attachment6, color-attachment6-ext, color-attachment7, color-attachment7-ext, color-attachment8, color-attachment8-ext, color-attachment9, color-attachment9-ext, color-attachment10, color-attachment10-ext, color-attachment11, color-attachment11-ext, color-attachment12, color-attachment12-ext, color-attachment13, color-attachment13-ext, color-attachment14, color-attachment14-ext, color-attachment15, color-attachment15-ext, depth-attachment, depth-attachment-ext, stencil-attachment, stencil-attachment-ext, framebuffer, framebuffer-ext, renderbuffer, renderbuffer-ext, renderbuffer-width, renderbuffer-width-ext, renderbuffer-height, renderbuffer-height-ext,

renderbuffer-internal-format, renderbuffer-internal-format-ext, stencil-index1, stencil-index1-ext, stencil-index4, stencil-index4-ext, stencil-index8, stencil-index8-ext, stencil-index16, stencil-index16-ext, renderbuffer-red-size, renderbuffer-red-size-ext, renderbuffer-green-size, renderbuffer-green-size-ext, renderbuffer-blue-size, renderbuffer-blue-size-ext, renderbuffer-alpha-size, renderbuffer-alpha-size-ext, renderbuffer-depth-size, renderbuffer-depth-size-ext, renderbuffer-stencil-size, renderbuffer-stencil-size-ext, framebuffer-attachment-layered-arb, framebuffer-incomplete-layer-targets-arb, framebuffer-incomplete-layer-count-arb, geometry-shader-arb, geometry-vertices-out-arb, geometry-input-type-arb, geometry-output-type-arb, max-geometry-varying-components-arb, max-vertex-varying-components-arb, max-geometry-uniform-components-arb, max-geometry-output-vertices-arb, max-geometry-total-output-components-arb.

`nv-geometry-program-4` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

lines-adjacency-ext, line-strip-adjacency-ext, triangles-adjacency-ext, triangle-strip-adjacency-ext, program-point-size-ext, geometry-program-nv, max-program-output-vertices-nv, max-program-total-output-components-nv, max-geometry-texture-image-units-ext, framebuffer-attachment-texture-layer-ext, framebuffer-attachment-layered-ext, framebuffer-incomplete-layer-targets-ext, framebuffer-incomplete-layer-count-ext, geometry-vertices-out-ext, geometry-input-type-ext, geometry-output-type-ext.

`arb-tessellation-shader` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

patches, uniform-block-referenced-by-tess-control-shader, uniform-block-referenced-by-tess-evaluation-shader, max-tess-control-input-components, max-tess-evaluation-input-components, max-combined-tess-control-uniform-components, max-combined-tess-evaluation-uniform-components, patch-vertices, patch-default-inner-level, patch-default-outer-level, tess-control-output-vertices, tess-gen-mode, tess-gen-spacing, tess-gen-vertex-order, tess-gen-point-mode, isolines, fractional-odd, fractional-even, max-patch-vertices, max-tess-gen-level, max-tess-control-uniform-components, max-tess-evaluation-uniform-components, max-tess-control-texture-image-units, max-tess-evaluation-texture-image-units, max-tess-control-output-components, max-tess-patch-components, max-tess-control-total-output-components, max-tess-evaluation-output-components, tess-evaluation-shader, tess-control-shader, max-tess-control-uniform-blocks, max-tess-evaluation-uniform-blocks.

`nv-gpu-shader-5` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`patches`, `int64-nv`, `unsigned-int64-nv`, `int8-nv`, `int8-vec2-nv`, `int8-vec3-nv`, `int8-vec4-nv`, `int16-nv`, `int16-vec2-nv`, `int16-vec3-nv`, `int16-vec4-nv`, `int64-vec2-nv`, `int64-vec3-nv`, `int64-vec4-nv`, `unsigned-int8-nv`, `unsigned-int8-vec2-nv`, `unsigned-int8-vec3-nv`, `unsigned-int8-vec4-nv`, `unsigned-int16-nv`, `unsigned-int16-vec2-nv`, `unsigned-int16-vec3-nv`, `unsigned-int16-vec4-nv`, `unsigned-int64-vec2-nv`, `unsigned-int64-vec3-nv`, `unsigned-int64-vec4-nv`, `float16-nv`, `float16-vec2-nv`, `float16-vec3-nv`, `float16-vec4-nv`.

`accum-op` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`accum`, `load`, `return`, `mult`, `add`.

`alpha-function` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`never`, `less`, `equal`, `lequal`, `greater`, `notequal`, `gequal`, `always`.

`blending-factor-dest` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`zero`, `one`, `src-color`, `one-minus-src-color`, `src-alpha`, `one-minus-src-alpha`, `dst-alpha`, `one-minus-dst-alpha`, `constant-color-ext`, `one-minus-constant-color-ext`, `constant-alpha-ext`, `one-minus-constant-alpha-ext`.

`blending-factor-src` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`zero`, `one`, `dst-color`, `one-minus-dst-color`, `src-alpha-saturate`, `src-alpha`, `one-minus-src-alpha`, `dst-alpha`, `one-minus-dst-alpha`, `constant-color-ext`, `one-minus-constant-color-ext`, `constant-alpha-ext`, `one-minus-constant-alpha-ext`.

`blend-equation-mode-ext` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`logic-op`, `func-add-ext`, `min-ext`, `max-ext`, `func-subtract-ext`, `func-reverse-subtract-ext`, `alpha-min-sgix`, `alpha-max-sgix`.

`color-material-face` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`front`, `back`, `front-and-back`.

`color-material-parameter` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`ambient`, `diffuse`, `specular`, `emission`, `ambient-and-diffuse`.

`color-pointer-type` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`byte`, `unsigned-byte`, `short`, `unsigned-short`, `int`, `unsigned-int`, `float`, `double`.

`color-table-parameter-p-name-sgi` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`color-table-scale-sgi`, `color-table-bias-sgi`.

`color-table-target-sgi` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`color-table-sgi`, `post-convolution-color-table-sgi`, `post-color-matrix-color-table-sgi`, `proxy-color-table-sgi`, `proxy-post-convolution-color-table-sgi`, `proxy-post-color-matrix-color-table-sgi`, `texture-color-table-sgi`, `proxy-texture-color-table-sgi`.

`convolution-border-mode-ext` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`reduce-ext`.

`convolution-parameter-ext` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`convolution-border-mode-ext`, `convolution-filter-scale-ext`, `convolution-filter-bias-ext`.

`convolution-target-ext` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`convolution-1d-ext`, `convolution-2d-ext`.

`cull-face-mode` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`front`, `back`, `front-and-back`.

`depth-function` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`never`, `less`, `equal`, `lequal`, `greater`, `notequal`, `gequal`, `always`.

`draw-buffer-mode` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`none`, `front-left`, `front-right`, `back-left`, `back-right`, `front`, `back`, `left`, `right`, `front-and-back`, `aux0`, `aux1`, `aux2`, `aux3`.

`oes-framebuffer-object` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`fog`, `lighting`, `texture-1d`, `texture-2d`, `line-stipple`, `polygon-stipple`, `cull-face`, `alpha-test`, `blend`, `index-logic-op`, `color-logic-op`, `dither`, `stencil-test`, `depth-test`, `clip-plane0`, `clip-plane1`, `clip-plane2`, `clip-plane3`, `clip-plane4`, `clip-plane5`, `light0`, `light1`, `light2`, `light3`, `light4`, `light5`, `light6`, `light7`, `texture-gen-s`, `texture-gen-t`, `texture-gen-r`, `texture-gen-q`, `map1-vertex-3`, `map1-vertex-4`, `map1-color-4`, `map1-index`, `map1-normal`, `map1-texture-coord-1`, `map1-texture-coord-2`, `map1-texture-coord-3`, `map1-texture-coord-4`, `map2-vertex-3`, `map2-vertex-4`, `map2-color-4`, `map2-index`, `map2-normal`, `map2-texture-coord-1`, `map2-texture-coord-2`, `map2-texture-coord-3`, `map2-texture-coord-4`, `point-smooth`, `line-smooth`, `polygon-smooth`, `scissor-test`, `color-material`, `normalize`, `auto-normal`, `polygon-offset-point`, `polygon-offset-line`, `polygon-offset-fill`, `vertex-array`, `normal-array`, `color-array`, `index-array`, `texture-coord-array`, `edge-flag-array`, `convolution-1d-ext`, `convolution-2d-ext`, `separable-2d-ext`, `histogram-ext`, `minmax-ext`, `rescale-normal-ext`, `shared-texture-palette-ext`, `texture-3d-ext`, `multisample-sgis`, `sample-alpha-to-mask-sgis`, `sample-alpha-to-one-sgis`, `sample-mask-sgis`, `texture-4d-sgis`, `async-histogram-sgix`, `async-tex-image-sgix`, `async-draw-pixels-sgix`,

async-read-pixels-sgix, calligraphic-fragment-sgix, fog-offset-sgix, fragment-lighting-sgix, fragment-color-material-sgix, fragment-light0-sgix, fragment-light1-sgix, fragment-light2-sgix, fragment-light3-sgix, fragment-light4-sgix, fragment-light5-sgix, fragment-light6-sgix, fragment-light7-sgix, framezoom-sgix, interlace-sgix, ir-instrument1-sgix, pixel-tex-gen-sgix, pixel-texture-sgis, reference-plane-sgix, sprite-sgix, color-table-sgi, post-convolution-color-table-sgi, post-color-matrix-color-table-sgi, texture-color-table-sgi, invalid-framebuffer-operation-oes, rgba4-oes, rgb5-a1-oes, depth-component16-oes, max-renderbuffer-size-oes, framebuffer-binding-oes, renderbuffer-binding-oes, framebuffer-attachment-object-type-oes, framebuffer-attachment-object-name-oes, framebuffer-attachment-texture-level-oes, framebuffer-attachment-texture-cube-map-face-oes, framebuffer-attachment-texture-3d-zoffset-oes, framebuffer-complete-oes, framebuffer-incomplete-attachment-oes, framebuffer-incomplete-missing-attachment-oes, framebuffer-incomplete-dimensions-oes, framebuffer-incomplete-formats-oes, framebuffer-incomplete-draw-buffer-oes, framebuffer-incomplete-read-buffer-oes, framebuffer-unsupported-oes, color-attachment0-oes, depth-attachment-oes, stencil-attachment-oes, framebuffer-oes, renderbuffer-oes, renderbuffer-width-oes, renderbuffer-height-oes, renderbuffer-internal-format-oes, stencil-index1-oes, stencil-index4-oes, stencil-index8-oes, renderbuffer-red-size-oes, renderbuffer-green-size-oes, renderbuffer-blue-size-oes, renderbuffer-alpha-size-oes, renderbuffer-depth-size-oes, renderbuffer-stencil-size-oes, rgb565-oes.

`enable-cap` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

fog, lighting, texture-1d, texture-2d, line-stipple, polygon-stipple, cull-face, alpha-test, blend, index-logic-op, color-logic-op, dither, stencil-test, depth-test, clip-plane0, clip-plane1, clip-plane2, clip-plane3, clip-plane4, clip-plane5, light0, light1, light2, light3, light4, light5, light6, light7, texture-gen-s, texture-gen-t, texture-gen-r, texture-gen-q, map1-vertex-3, map1-vertex-4, map1-color-4, map1-index, map1-normal, map1-texture-coord-1, map1-texture-coord-2, map1-texture-coord-3, map1-texture-coord-4, map2-vertex-3, map2-vertex-4, map2-color-4, map2-index, map2-normal, map2-texture-coord-1, map2-texture-coord-2, map2-texture-coord-3, map2-texture-coord-4, point-smooth, line-smooth, polygon-smooth, scissor-test, color-material, normalize, auto-normal, polygon-offset-point, polygon-offset-line, polygon-offset-fill, vertex-array, normal-array, color-array, index-array, texture-coord-array, edge-flag-array, convolution-1d-ext, convolution-2d-ext, separable-2d-ext, histogram-ext, minmax-ext, rescale-normal-ext, shared-texture-palette-ext, texture-3d-ext, multisample-sgis, sample-alpha-to-mask-sgis, sample-alpha-to-one-sgis, sample-mask-sgis, texture-4d-sgis,

async-histogram-sgix, async-tex-image-sgix, async-draw-pixels-sgix, async-read-pixels-sgix, calligraphic-fragment-sgix, fog-offset-sgix, fragment-lighting-sgix, fragment-color-material-sgix, fragment-light0-sgix, fragment-light1-sgix, fragment-light2-sgix, fragment-light3-sgix, fragment-light4-sgix, fragment-light5-sgix, fragment-light6-sgix, fragment-light7-sgix, framezoom-sgix, interlace-sgix, ir-instrument1-sgix, pixel-tex-gen-sgix, pixel-texture-sgis, reference-plane-sgix, sprite-sgix, color-table-sgi, post-convolution-color-table-sgi, post-color-matrix-color-table-sgi, texture-color-table-sgi.

`error-code` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

no-error, invalid-enum, invalid-value, invalid-operation, stack-overflow, stack-underflow, out-of-memory, table-too-large-ext, texture-too-large-ext.

`arb-framebuffer-object` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

invalid-framebuffer-operation, framebuffer-attachment-color-encoding, framebuffer-attachment-component-type, framebuffer-attachment-red-size, framebuffer-attachment-green-size, framebuffer-attachment-blue-size, framebuffer-attachment-alpha-size, framebuffer-attachment-depth-size, framebuffer-attachment-stencil-size, framebuffer-default, framebuffer-undefined, depth-stencil-attachment, index, max-renderbuffer-size, depth-stencil, unsigned-int-24-8, depth24-stencil8, texture-stencil-size, texture-red-type, texture-green-type, texture-blue-type, texture-alpha-type, texture-luminance-type, texture-intensity-type, texture-depth-type, unsigned-normalized, framebuffer-binding, draw-framebuffer-binding, renderbuffer-binding, read-framebuffer, draw-framebuffer, read-framebuffer-binding, renderbuffer-samples, framebuffer-attachment-object-type, framebuffer-attachment-object-type-ext, framebuffer-attachment-object-name, framebuffer-attachment-object-name-ext, framebuffer-attachment-texture-level, framebuffer-attachment-texture-level-ext, framebuffer-attachment-texture-cube-map-face, framebuffer-attachment-texture-cube-map-face-ext, framebuffer-attachment-texture-layer, framebuffer-attachment-texture-3d-zoffset-ext, framebuffer-complete, framebuffer-complete-ext, framebuffer-incomplete-attachment, framebuffer-incomplete-attachment-ext, framebuffer-incomplete-missing-attachment, framebuffer-incomplete-missing-attachment-ext, framebuffer-incomplete-dimensions-ext, framebuffer-incomplete-formats-ext, framebuffer-incomplete-draw-buffer, framebuffer-incomplete-draw-buffer-ext, framebuffer-incomplete-read-buffer, framebuffer-incomplete-read-buffer-ext, framebuffer-unsupported, framebuffer-unsupported-ext,

max-color-attachments, max-color-attachments-ext, color-attachment0, color-attachment0-ext, color-attachment1, color-attachment1-ext, color-attachment2, color-attachment2-ext, color-attachment3, color-attachment3-ext, color-attachment4, color-attachment4-ext, color-attachment5, color-attachment5-ext, color-attachment6, color-attachment6-ext, color-attachment7, color-attachment7-ext, color-attachment8, color-attachment8-ext, color-attachment9, color-attachment9-ext, color-attachment10, color-attachment10-ext, color-attachment11, color-attachment11-ext, color-attachment12, color-attachment12-ext, color-attachment13, color-attachment13-ext, color-attachment14, color-attachment14-ext, color-attachment15, color-attachment15-ext, depth-attachment, depth-attachment-ext, stencil-attachment, stencil-attachment-ext, framebuffer, framebuffer-ext, renderbuffer, renderbuffer-ext, renderbuffer-width, renderbuffer-width-ext, renderbuffer-height, renderbuffer-height-ext, renderbuffer-internal-format, renderbuffer-internal-format-ext, stencil-index1, stencil-index1-ext, stencil-index4, stencil-index4-ext, stencil-index8, stencil-index8-ext, stencil-index16, stencil-index16-ext, renderbuffer-red-size, renderbuffer-red-size-ext, renderbuffer-green-size, renderbuffer-green-size-ext, renderbuffer-blue-size, renderbuffer-blue-size-ext, renderbuffer-alpha-size, renderbuffer-alpha-size-ext, renderbuffer-depth-size, renderbuffer-depth-size-ext, renderbuffer-stencil-size, renderbuffer-stencil-size-ext, framebuffer-incomplete-multisample, max-samples.

`ext-framebuffer-object` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

invalid-framebuffer-operation-ext, max-renderbuffer-size-ext, framebuffer-binding-ext, renderbuffer-binding-ext, framebuffer-attachment-object-type, framebuffer-attachment-object-type-ext, framebuffer-attachment-object-name, framebuffer-attachment-object-name-ext, framebuffer-attachment-texture-level, framebuffer-attachment-texture-level-ext, framebuffer-attachment-texture-cube-map-face, framebuffer-attachment-texture-cube-map-face-ext, framebuffer-attachment-texture-layer, framebuffer-attachment-texture-3d-zoffset-ext, framebuffer-complete, framebuffer-complete-ext, framebuffer-incomplete-attachment, framebuffer-incomplete-attachment-ext, framebuffer-incomplete-missing-attachment, framebuffer-incomplete-missing-attachment-ext, framebuffer-incomplete-dimensions-ext, framebuffer-incomplete-formats-ext, framebuffer-incomplete-draw-buffer, framebuffer-incomplete-draw-buffer-ext, framebuffer-incomplete-read-buffer, framebuffer-incomplete-read-buffer-ext, framebuffer-unsupported, framebuffer-unsupported-ext, max-color-attachments, max-color-attachments-ext, color-attachment0, color-attachment0-ext, color-attachment1, color-attachment1-ext, color-attachment2, color-attachment2-ext, color-attachment3, color-attachment3-

ext, color-attachment4, color-attachment4-ext, color-attachment5, color-attachment5-ext, color-attachment6, color-attachment6-ext, color-attachment7, color-attachment7-ext, color-attachment8, color-attachment8-ext, color-attachment9, color-attachment9-ext, color-attachment10, color-attachment10-ext, color-attachment11, color-attachment11-ext, color-attachment12, color-attachment12-ext, color-attachment13, color-attachment13-ext, color-attachment14, color-attachment14-ext, color-attachment15, color-attachment15-ext, depth-attachment, depth-attachment-ext, stencil-attachment, stencil-attachment-ext, framebuffer, framebuffer-ext, renderbuffer, renderbuffer-ext, renderbuffer-width, renderbuffer-width-ext, renderbuffer-height, renderbuffer-height-ext, renderbuffer-internal-format, renderbuffer-internal-format-ext, stencil-index1, stencil-index1-ext, stencil-index4, stencil-index4-ext, stencil-index8, stencil-index8-ext, stencil-index16, stencil-index16-ext, renderbuffer-red-size, renderbuffer-red-size-ext, renderbuffer-green-size, renderbuffer-green-size-ext, renderbuffer-blue-size, renderbuffer-blue-size-ext, renderbuffer-alpha-size, renderbuffer-alpha-size-ext, renderbuffer-depth-size, renderbuffer-depth-size-ext, renderbuffer-stencil-size, renderbuffer-stencil-size-ext.

`feedback-type` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`2d`, `3d`, `3d-color`, `3d-color-texture`, `4d-color-texture`.

`feed-back-token` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`pass-through-token`, `point-token`, `line-token`, `polygon-token`, `bitmap-token`, `draw-pixel-token`, `copy-pixel-token`, `line-reset-token`.

`ffd-mask-sgix` *bit...* [Macro]

Bitfield constructor. The symbolic *bit* arguments are replaced with their corresponding numeric values and combined with `logior` at compile-time. The symbolic arguments known to this bitfield constructor are:

`texture-deformation-bit-sgix`, `geometry-deformation-bit-sgix`.

`ffd-target-sgix` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`geometry-deformation-sgix`, `texture-deformation-sgix`.

`fog-mode` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`linear`, `exp`, `exp2`, `fog-func-sgis`.

`fog-parameter` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`fog-color`, `fog-density`, `fog-end`, `fog-index`, `fog-mode`, `fog-start`, `fog-offset-value-sgix`.

`fragment-light-model-parameter-sgix` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`fragment-light-model-local-viewer-sgix`, `fragment-light-model-two-side-sgix`, `fragment-light-model-ambient-sgix`, `fragment-light-model-normal-interpolation-sgix`.

`front-face-direction` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`cw`, `ccw`.

`get-color-table-parameter-p-name-sgi` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`color-table-scale-sgi`, `color-table-bias-sgi`, `color-table-format-sgi`, `color-table-width-sgi`, `color-table-red-size-sgi`, `color-table-green-size-sgi`, `color-table-blue-size-sgi`, `color-table-alpha-size-sgi`, `color-table-luminance-size-sgi`, `color-table-intensity-size-sgi`.

`get-convolution-parameter` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`convolution-border-mode-ext`, `convolution-filter-scale-ext`, `convolution-filter-bias-ext`, `convolution-format-ext`, `convolution-width-ext`, `convolution-height-ext`, `max-convolution-width-ext`, `max-convolution-height-ext`.

`get-histogram-parameter-p-name-ext` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`histogram-width-ext`, `histogram-format-ext`, `histogram-red-size-ext`, `histogram-green-size-ext`, `histogram-blue-size-ext`, `histogram-alpha-size-ext`, `histogram-luminance-size-ext`, `histogram-sink-ext`.

`get-map-query` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`coeff`, `order`, `domain`.

`get-minmax-parameter-p-name-ext` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`minmax-format-ext`, `minmax-sink-ext`.

`get-pixel-map` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`pixel-map-i-to-i`, `pixel-map-s-to-s`, `pixel-map-i-to-r`, `pixel-map-i-to-g`, `pixel-map-i-to-b`, `pixel-map-i-to-a`, `pixel-map-r-to-r`, `pixel-map-g-to-g`, `pixel-map-b-to-b`, `pixel-map-a-to-a`.

`get-pointerv-p-name` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`vertex-array-pointer`, `normal-array-pointer`, `color-array-pointer`, `index-array-pointer`, `texture-coord-array-pointer`, `edge-flag-array-pointer`, `feedback-buffer-pointer`, `selection-buffer-pointer`, `instrument-buffer-pointer-sgix`.

`get-p-name` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`current-color`, `current-index`, `current-normal`, `current-texture-coords`, `current-raster-color`, `current-raster-index`, `current-raster-texture-coords`, `current-raster-position`, `current-raster-position-valid`, `current-raster-distance`, `point-smooth`, `point-size`, `point-size-range`, `point-size-granularity`, `line-smooth`, `line-width`, `line-width-range`, `line-width-granularity`, `line-stipple`, `line-stipple-pattern`, `line-stipple-repeat`, `smooth-point-size-range`, `smooth-point-size-granularity`, `smooth-line-width-range`, `smooth-line-width-granularity`, `aliased-point-size-range`, `aliased-line-width-range`, `list-mode`, `max-list-nesting`, `list-base`, `list-index`, `polygon-mode`, `polygon-smooth`, `polygon-stipple`,

edge-flag, cull-face, cull-face-mode, front-face, lighting, light-model-local-viewer, light-model-two-side, light-model-ambient, shade-model, color-material-face, color-material-parameter, color-material, fog, fog-index, fog-density, fog-start, fog-end, fog-mode, fog-color, depth-range, depth-test, depth-writemask, depth-clear-value, depth-func, accum-clear-value, stencil-test, stencil-clear-value, stencil-func, stencil-value-mask, stencil-fail, stencil-pass-depth-fail, stencil-pass-depth-pass, stencil-ref, stencil-writemask, matrix-mode, normalize, viewport, modelview-stack-depth, projection-stack-depth, texture-stack-depth, modelview-matrix, projection-matrix, texture-matrix, attrib-stack-depth, client-attrib-stack-depth, alpha-test, alpha-test-func, alpha-test-ref, dither, blend-dst, blend-src, blend, logic-op-mode, index-logic-op, logic-op, color-logic-op, aux-buffers, draw-buffer, read-buffer, scissor-box, scissor-test, index-clear-value, index-writemask, color-clear-value, color-writemask, index-mode, rgba-mode, doublebuffer, stereo, render-mode, perspective-correction-hint, point-smooth-hint, line-smooth-hint, polygon-smooth-hint, fog-hint, texture-gen-s, texture-gen-t, texture-gen-r, texture-gen-q, pixel-map-i-to-i-size, pixel-map-s-to-s-size, pixel-map-i-to-r-size, pixel-map-i-to-g-size, pixel-map-i-to-b-size, pixel-map-i-to-a-size, pixel-map-r-to-r-size, pixel-map-g-to-g-size, pixel-map-b-to-b-size, pixel-map-a-to-a-size, unpack-swap-bytes, unpack-lsb-first, unpack-row-length, unpack-skip-rows, unpack-skip-pixels, unpack-alignment, pack-swap-bytes, pack-lsb-first, pack-row-length, pack-skip-rows, pack-skip-pixels, pack-alignment, map-color, map-stencil, index-shift, index-offset, red-scale, red-bias, zoom-x, zoom-y, green-scale, green-bias, blue-scale, blue-bias, alpha-scale, alpha-bias, depth-scale, depth-bias, max-eval-order, max-lights, max-clip-distances, max-clip-planes, max-texture-size, max-pixel-map-table, max-attrib-stack-depth, max-modelview-stack-depth, max-name-stack-depth, max-projection-stack-depth, max-texture-stack-depth, max-viewport-dims, max-client-attrib-stack-depth, subpixel-bits, index-bits, red-bits, green-bits, blue-bits, alpha-bits, depth-bits, stencil-bits, accum-red-bits, accum-green-bits, accum-blue-bits, accum-alpha-bits, name-stack-depth, auto-normal, map1-color-4, map1-index, map1-normal, map1-texture-coord-1, map1-texture-coord-2, map1-texture-coord-3, map1-texture-coord-4, map1-vertex-3, map1-vertex-4, map2-color-4, map2-index, map2-normal, map2-texture-coord-1, map2-texture-coord-2, map2-texture-coord-3, map2-texture-coord-4, map2-vertex-3, map2-vertex-4, map1-grid-domain, map1-grid-segments, map2-grid-domain, map2-grid-segments, texture-1d, texture-2d, feedback-buffer-size, feedback-buffer-type, selection-buffer-size, polygon-offset-units, polygon-offset-point, polygon-offset-line, polygon-offset-fill, polygon-offset-factor, texture-binding-1d, texture-binding-2d, texture-binding-3d, vertex-array, normal-array, color-array, index-array, texture-coord-array, edge-flag-array, vertex-array-size, vertex-array-type, vertex-array-stride, normal-array-type, normal-array-stride, color-array-size, color-array-type, color-array-stride, index-array-type,

index-array-stride, texture-coord-array-size, texture-coord-array-type, texture-coord-array-stride, edge-flag-array-stride, clip-plane0, clip-plane1, clip-plane2, clip-plane3, clip-plane4, clip-plane5, light0, light1, light2, light3, light4, light5, light6, light7, light-model-color-control, blend-color-ext, blend-equation-ext, pack-cmyk-hint-ext, unpack-cmyk-hint-ext, convolution-1d-ext, convolution-2d-ext, separable-2d-ext, post-convolution-red-scale-ext, post-convolution-green-scale-ext, post-convolution-blue-scale-ext, post-convolution-alpha-scale-ext, post-convolution-red-bias-ext, post-convolution-green-bias-ext, post-convolution-blue-bias-ext, post-convolution-alpha-bias-ext, histogram-ext, minmax-ext, polygon-offset-bias-ext, rescale-normal-ext, shared-texture-palette-ext, texture-3d-binding-ext, pack-skip-images-ext, pack-image-height-ext, unpack-skip-images-ext, unpack-image-height-ext, texture-3d-ext, max-3d-texture-size-ext, vertex-array-count-ext, normal-array-count-ext, color-array-count-ext, index-array-count-ext, texture-coord-array-count-ext, edge-flag-array-count-ext, detail-texture-2d-binding-sgis, fog-func-points-sgis, max-fog-func-points-sgis, generate-mipmap-hint-sgis, multisample-sgis, sample-alpha-to-mask-sgis, sample-alpha-to-one-sgis, sample-mask-sgis, sample-buffers-sgis, samples-sgis, sample-mask-value-sgis, sample-mask-invert-sgis, sample-pattern-sgis, pixel-texture-sgis, point-size-min-sgis, point-size-max-sgis, point-fade-threshold-size-sgis, distance-attenuation-sgis, pack-skip-volumes-sgis, pack-image-depth-sgis, unpack-skip-volumes-sgis, unpack-image-depth-sgis, texture-4d-sgis, max-4d-texture-size-sgis, texture-4d-binding-sgis, async-marker-sgix, async-histogram-sgix, max-async-histogram-sgix, async-tex-image-sgix, async-draw-pixels-sgix, async-read-pixels-sgix, max-async-tex-image-sgix, max-async-draw-pixels-sgix, max-async-read-pixels-sgix, calligraphic-fragment-sgix, max-clipmap-virtual-depth-sgix, max-clipmap-depth-sgix, convolution-hint-sgix, fog-offset-sgix, fog-offset-value-sgix, fragment-lighting-sgix, fragment-color-material-sgix, fragment-color-material-face-sgix, fragment-color-material-parameter-sgix, max-fragment-lights-sgix, max-active-lights-sgix, light-env-mode-sgix, fragment-light-model-local-viewer-sgix, fragment-light-model-two-side-sgix, fragment-light-model-ambient-sgix, fragment-light-model-normal-interpolation-sgix, fragment-light0-sgix, framezoom-sgix, framezoom-factor-sgix, max-framezoom-factor-sgix, instrument-measurements-sgix, interlace-sgix, ir-instrument1-sgix, pixel-tex-gen-sgix, pixel-tex-gen-mode-sgix, pixel-tile-best-alignment-sgix, pixel-tile-cache-increment-sgix, pixel-tile-width-sgix, pixel-tile-height-sgix, pixel-tile-grid-width-sgix, pixel-tile-grid-height-sgix, pixel-tile-grid-depth-sgix, pixel-tile-cache-size-sgix, deformations-mask-sgix, reference-plane-equation-sgix, reference-plane-sgix, sprite-sgix, sprite-mode-sgix, sprite-axis-sgix, sprite-translation-sgix, pack-subsample-rate-sgix, unpack-subsample-rate-sgix, pack-resample-sgix, unpack-resample-sgix, post-texture-filter-bias-range-sgix, post-texture-filter-scale-range-

sgix, vertex-preclip-sgix, vertex-preclip-hint-sgix, color-matrix-sgi, color-matrix-stack-depth-sgi, max-color-matrix-stack-depth-sgi, post-color-matrix-red-scale-sgi, post-color-matrix-green-scale-sgi, post-color-matrix-blue-scale-sgi, post-color-matrix-alpha-scale-sgi, post-color-matrix-red-bias-sgi, post-color-matrix-green-bias-sgi, post-color-matrix-blue-bias-sgi, post-color-matrix-alpha-bias-sgi, color-table-sgi, post-convolution-color-table-sgi, post-color-matrix-color-table-sgi, texture-color-table-sgi.

`qcom-alpha-test` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

alpha-test-qcom, alpha-test-func-qcom, alpha-test-ref-qcom.

`ext-unpack-subimage` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

unpack-row-length, unpack-skip-rows, unpack-skip-pixels.

`ext-multiview-draw-buffers` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

draw-buffer-ext, read-buffer-ext, draw-buffer-ext, read-buffer-ext, color-attachment-ext, multiview-ext, max-multiview-buffers-ext.

`nv-read-buffer` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

read-buffer-nv.

`get-texture-parameter` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

texture-mag-filter, texture-min-filter, texture-wrap-s, texture-wrap-t, texture-width, texture-height, texture-internal-format, texture-components, texture-border-color, texture-border, texture-red-size, texture-green-size, texture-blue-size, texture-alpha-size, texture-luminance-size, texture-intensity-size, texture-priority, texture-resident, texture-depth-ext, texture-wrap-r-ext, detail-texture-level-sgis, detail-texture-mode-sgis, detail-texture-func-points-sgis, generate-mipmap-sgis, sharpen-texture-func-points-sgis, texture-filter4-size-sgis, texture-min-lod-sgis, texture-max-lod-sgis, texture-base-level-sgis, texture-max-level-sgis, dual-texture-select-sgis,

`quad-texture-select-sgis`, `texture-4dsize-sgis`, `texture-wrap-q-sgis`, `texture-clipmap-center-sgix`, `texture-clipmap-frame-sgix`, `texture-clipmap-offset-sgix`, `texture-clipmap-virtual-depth-sgix`, `texture-clipmap-lod-offset-sgix`, `texture-clipmap-depth-sgix`, `texture-compare-sgix`, `texture-compare-operator-sgix`, `texture-lequal-r-sgix`, `texture-gequal-r-sgix`, `shadow-ambient-sgix`, `texture-max-clamp-s-sgix`, `texture-max-clamp-t-sgix`, `texture-max-clamp-r-sgix`, `texture-lod-bias-s-sgix`, `texture-lod-bias-t-sgix`, `texture-lod-bias-r-sgix`, `post-texture-filter-bias-sgix`, `post-texture-filter-scale-sgix`.

`nv-texture-border-clamp` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-border-color-nv`, `clamp-to-border-nv`.

`hint-mode` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`dont-care`, `fastest`, `nicest`.

`hint-target` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`perspective-correction-hint`, `point-smooth-hint`, `line-smooth-hint`, `polygon-smooth-hint`, `fog-hint`, `pack-cmyk-hint-ext`, `unpack-cmyk-hint-ext`, `generate-mipmap-hint-sgis`, `convolution-hint-sgix`, `texture-multi-buffer-hint-sgix`, `vertex-preclip-hint-sgix`.

`histogram-target-ext` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`histogram-ext`, `proxy-histogram-ext`.

`index-pointer-type` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`short`, `int`, `float`, `double`.

`light-env-mode-sgix` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`replace`, `modulate`, `add`.

`light-env-parameter-sgix` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`light-env-mode-sgix`.

`light-model-color-control` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`single-color`, `separate-specular-color`.

`light-model-parameter` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`light-model-ambient`, `light-model-local-viewer`, `light-model-two-side`, `light-model-color-control`.

`light-parameter` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`ambient`, `diffuse`, `specular`, `position`, `spot-direction`, `spot-exponent`, `spot-cutoff`, `constant-attenuation`, `linear-attenuation`, `quadratic-attenuation`.

`list-mode` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`compile`, `compile-and-execute`.

`data-type` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`byte`, `unsigned-byte`, `short`, `unsigned-short`, `int`, `unsigned-int`, `float`, `2-bytes`, `3-bytes`, `4-bytes`, `double`, `double-ext`.

`oes-element-index-uint` *bit...* [Macro]

Bitfield constructor. The symbolic *bit* arguments are replaced with their corresponding numeric values and combined with `logior` at compile-time. The symbolic arguments known to this bitfield constructor are:

.

`oes-texture-float` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`half-float-oes`.

`ext-vertex-attrib-64-bit` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`double-mat2-ext`, `double-mat3-ext`, `double-mat4-ext`, `double-mat-2x-3-ext`, `double-mat-2x-4-ext`, `double-mat-3x-2-ext`, `double-mat-3x-4-ext`, `double-mat-4x-2-ext`, `double-mat-4x-3-ext`, `double-vec2-ext`, `double-vec3-ext`, `double-vec4-ext`.

`arb-half-float-vertex` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`half-float`.

`arb-half-float-pixel` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`half-float-arb`.

`nv-half-float` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`half-float-nv`.

`apple-float-pixels` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`half-apple`, `rgba-float32-apple`, `rgb-float32-apple`, `alpha-float32-apple`, `intensity-float32-apple`, `luminance-float32-apple`, `luminance-alpha-float32-apple`, `rgba-float16-apple`, `rgb-float16-apple`, `alpha-float16-apple`, `intensity-float16-apple`, `luminance-float16-apple`, `luminance-alpha-float16-apple`, `color-float-apple`.

`arb-es2-compatibility` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`fixed`, `implementation-color-read-type`, `implementation-color-read-format`, `rgb565`, `low-float`, `medium-float`, `high-float`, `low-int`, `medium-int`, `high-int`, `shader-binary-formats`, `num-shader-binary-formats`, `shader-compiler`, `max-vertex-uniform-vectors`, `max-varying-vectors`, `max-fragment-uniform-vectors`.

`oes-fixed-point` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`fixed-oes`.

`nv-vertex-attrib-integer-64-bit` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`int64-nv`, `unsigned-int64-nv`.

`list-name-type` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`byte`, `unsigned-byte`, `short`, `unsigned-short`, `int`, `unsigned-int`, `float`, `2-bytes`, `3-bytes`, `4-bytes`.

`list-parameter-name` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`list-priority-sgix`.

`logic-op` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`clear`, `and`, `and-reverse`, `copy`, `and-inverted`, `noop`, `xor`, `or`, `nor`, `equiv`, `invert`, `or-reverse`, `copy-inverted`, `or-inverted`, `nand`, `set`.

`map-target` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`map1-color-4`, `map1-index`, `map1-normal`, `map1-texture-coord-1`, `map1-texture-coord-2`, `map1-texture-coord-3`, `map1-texture-coord-4`, `map1-vertex-3`, `map1-vertex-4`, `map2-color-4`, `map2-index`, `map2-normal`, `map2-texture-coord-1`, `map2-texture-coord-2`, `map2-texture-coord-3`, `map2-texture-coord-4`, `map2-vertex-3`, `map2-vertex-4`, `geometry-deformation-sgix`, `texture-deformation-sgix`.

`material-face` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`front`, `back`, `front-and-back`.

`material-parameter` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`emission`, `shininess`, `ambient-and-diffuse`, `color-indexes`, `ambient`, `diffuse`, `specular`.

`matrix-mode` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`modelview`, `projection`, `texture`.

`mesh-mode-1` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`point`, `line`.

`mesh-mode-2` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`point`, `line`, `fill`.

`minmax-target-ext` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`minmax-ext`.

`normal-pointer-type` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`byte`, `short`, `int`, `float`, `double`.

`pixel-copy-type` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`color`, `depth`, `stencil`.

`ext-discard-framebuffer` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`color-ext`, `depth-ext`, `stencil-ext`.

`pixel-format` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`color-index`, `stencil-index`, `depth-component`, `red`, `green`, `blue`, `alpha`, `rgb`, `rgba`, `luminance`, `luminance-alpha`, `abgr-ext`, `cmyk-ext`, `cmyka-ext`, `ycrcb-422-sgix`, `ycrcb-444-sgix`.

`oes-depth-texture` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`red-ext`.

`ext-texture-rg` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`red-ext`, `rg-ext`, `r8-ext`, `rg8-ext`.

`pixel-map` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`pixel-map-i-to-i`, `pixel-map-s-to-s`, `pixel-map-i-to-r`, `pixel-map-i-to-g`, `pixel-map-i-to-b`, `pixel-map-i-to-a`, `pixel-map-r-to-r`, `pixel-map-g-to-g`, `pixel-map-b-to-b`, `pixel-map-a-to-a`.

`pixel-store-parameter` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`unpack-swap-bytes`, `unpack-lsb-first`, `unpack-row-length`, `unpack-skip-rows`, `unpack-skip-pixels`, `unpack-alignment`, `pack-swap-bytes`, `pack-lsb-first`, `pack-row-length`, `pack-skip-rows`, `pack-skip-pixels`, `pack-alignment`, `pack-skip-images-ext`, `pack-image-height-ext`, `unpack-skip-images-ext`, `unpack-image-height-ext`, `pack-skip-volumes-sgis`, `pack-image-depth-sgis`, `unpack-skip-volumes-sgis`, `unpack-image-depth-sgis`, `pixel-tile-width-sgix`, `pixel-tile-height-sgix`, `pixel-tile-grid-width-sgix`, `pixel-tile-grid-height-sgix`, `pixel-tile-grid-depth-sgix`, `pixel-tile-cache-size-sgix`, `pack-subsample-rate-sgix`, `unpack-subsample-rate-sgix`, `pack-resample-sgix`, `unpack-resample-sgix`.

`pixel-store-resample-mode` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`resample-replicate-sgix`, `resample-zero-fill-sgix`, `resample-decimate-sgix`.

`pixel-store-subsample-rate` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`pixel-subsample-4444-sgix`, `pixel-subsample-2424-sgix`, `pixel-subsample-4242-sgix`.

`pixel-tex-gen-mode` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`none`, `rgb`, `rgba`, `luminance`, `luminance-alpha`, `pixel-tex-gen-alpha-replace-sgix`, `pixel-tex-gen-alpha-no-replace-sgix`, `pixel-tex-gen-alpha-ms-sgix`, `pixel-tex-gen-alpha-ls-sgix`.

`pixel-tex-gen-parameter-name-sgis` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`pixel-fragment-rgb-source-sgis`, `pixel-fragment-alpha-source-sgis`.

`pixel-transfer-parameter` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`map-color`, `map-stencil`, `index-shift`, `index-offset`, `red-scale`, `red-bias`, `green-scale`, `green-bias`, `blue-scale`, `blue-bias`, `alpha-scale`, `alpha-bias`, `depth-scale`, `depth-bias`, `post-convolution-red-scale-ext`, `post-convolution-green-scale-ext`, `post-convolution-blue-scale-ext`, `post-convolution-alpha-scale-ext`, `post-convolution-red-bias-ext`, `post-convolution-green-bias-ext`, `post-convolution-blue-bias-ext`, `post-convolution-alpha-bias-ext`, `post-color-matrix-red-scale-sgi`, `post-color-matrix-green-scale-sgi`, `post-color-matrix-blue-scale-sgi`, `post-color-matrix-alpha-scale-sgi`, `post-color-matrix-red-bias-sgi`, `post-color-matrix-green-bias-sgi`, `post-color-matrix-blue-bias-sgi`, `post-color-matrix-alpha-bias-sgi`.

`pixel-type` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`bitmap`, `byte`, `unsigned-byte`, `short`, `unsigned-short`, `int`, `unsigned-int`, `float`, `unsigned-byte-3-3-2-ext`, `unsigned-short-4-4-4-4-ext`, `unsigned-short-5-5-5-1-ext`, `unsigned-int-8-8-8-8-ext`, `unsigned-int-10-10-10-2-ext`.

`point-parameter-name-sgis` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`point-size-min-sgis`, `point-size-max-sgis`, `point-fade-threshold-size-sgis`, `distance-attenuation-sgis`.

`polygon-mode` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`point`, `line`, `fill`.

`read-buffer-mode` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`front-left`, `front-right`, `back-left`, `back-right`, `front`, `back`, `left`, `right`, `aux0`, `aux1`, `aux2`, `aux3`.

`rendering-mode` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`render`, `feedback`, `select`.

`sample-pattern-sgis` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`1pass-sgis`, `2pass-0-sgis`, `2pass-1-sgis`, `4pass-0-sgis`, `4pass-1-sgis`, `4pass-2-sgis`, `4pass-3-sgis`.

`separable-target-ext` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`separable-2d-ext`.

`shading-model` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`flat`, `smooth`.

`stencil-function` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`never`, `less`, `equal`, `lequal`, `greater`, `notequal`, `gequal`, `always`.

`stencil-op` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`zero`, `keep`, `replace`, `incr`, `decr`, `invert`.

`string-name` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`vendor`, `renderer`, `version`, `extensions`.

`tex-coord-pointer-type` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`short`, `int`, `float`, `double`.

`texture-coord-name` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`s`, `t`, `r`, `q`.

`texture-env-mode` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`modulate`, `decal`, `blend`, `replace-ext`, `add`, `texture-env-bias-sgix`.

`texture-env-parameter` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-env-mode`, `texture-env-color`.

`texture-env-target` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-env`.

`texture-filter-func-sgis` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`filter4-sgis`.

`texture-gen-mode` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`eye-linear`, `object-linear`, `sphere-map`, `eye-distance-to-point-sgis`, `object-distance-to-point-sgis`, `eye-distance-to-line-sgis`, `object-distance-to-line-sgis`.

`texture-gen-parameter` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-gen-mode`, `object-plane`, `eye-plane`, `eye-point-sgis`, `object-point-sgis`, `eye-line-sgis`, `object-line-sgis`.

`oes-texture-cube-map` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-gen-mode`, `normal-map-oes`, `reflection-map-oes`, `texture-cube-map-oes`, `texture-binding-cube-map-oes`, `texture-cube-map-positive-x-oes`, `texture-cube-map-negative-x-oes`, `texture-cube-map-positive-y-oes`, `texture-cube-map-negative-y-oes`, `texture-cube-map-positive-z-oes`, `texture-cube-map-negative-z-oes`, `max-cube-map-texture-size-oes`, `texture-gen-str-oes`.

`texture-mag-filter` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`nearest`, `linear`, `linear-detail-sgis`, `linear-detail-alpha-sgis`, `linear-detail-color-sgis`, `linear-sharpen-sgis`, `linear-sharpen-alpha-sgis`, `linear-sharpen-color-sgis`, `filter4-sgis`, `pixel-tex-gen-q-ceiling-sgix`, `pixel-tex-gen-q-round-sgix`, `pixel-tex-gen-q-floor-sgix`.

`texture-min-filter` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`nearest`, `linear`, `nearest-mipmap-nearest`, `linear-mipmap-nearest`, `nearest-mipmap-linear`, `linear-mipmap-linear`, `filter4-sgis`, `linear-clipmap-linear-sgix`, `nearest-clipmap-nearest-sgix`, `nearest-clipmap-linear-sgix`, `linear-clipmap-nearest-sgix`, `pixel-tex-gen-q-ceiling-sgix`, `pixel-tex-gen-q-round-sgix`, `pixel-tex-gen-q-floor-sgix`.

`texture-parameter-name` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-mag-filter`, `texture-min-filter`, `texture-wrap-s`, `texture-wrap-t`, `texture-border-color`, `texture-priority`, `texture-wrap-r-ext`, `detail-texture-level-sgis`, `detail-texture-mode-sgis`, `generate-mipmap-sgis`, `dual-texture-select-sgis`, `quad-texture-select-sgis`, `texture-wrap-q-sgis`, `texture-clipmap-center-sgix`, `texture-clipmap-frame-sgix`, `texture-clipmap-offset-sgix`, `texture-clipmap-virtual-depth-sgix`, `texture-clipmap-lod-offset-sgix`, `texture-clipmap-depth-sgix`, `texture-compare-sgix`, `texture-compare-operator-sgix`, `shadow-ambient-sgix`, `texture-max-clamp-s-sgix`, `texture-max-clamp-t-sgix`, `texture-max-clamp-r-sgix`, `texture-lod-bias-s-sgix`, `texture-lod-bias-t-sgix`, `texture-lod-bias-r-sgix`, `post-texture-filter-bias-sgix`, `post-texture-filter-scale-sgix`.

`texture-target` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-1d`, `texture-2d`, `proxy-texture-1d`, `proxy-texture-2d`, `texture-3d-ext`, `proxy-texture-3d-ext`, `detail-texture-2d-sgis`, `texture-4d-sgis`, `proxy-texture-4d-sgis`, `texture-min-lod-sgis`, `texture-max-lod-sgis`, `texture-base-level-sgis`, `texture-max-level-sgis`.

`texture-wrap-mode` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`clamp`, `repeat`, `clamp-to-border-sgis`, `clamp-to-edge-sgis`.

`pixel-internal-format` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`r3-g3-b2`, `alpha4`, `alpha8`, `alpha12`, `alpha16`, `luminance4`, `luminance8`, `luminance12`, `luminance16`, `luminance4-alpha4`, `luminance6-alpha2`, `luminance8-alpha8`, `luminance12-alpha4`, `luminance12-alpha12`, `luminance16-alpha16`, `intensity`, `intensity4`, `intensity8`, `intensity12`, `intensity16`, `rgb4`, `rgb5`, `rgb8`, `rgb10`, `rgb12`, `rgb16`, `rgba2`, `rgba4`, `rgb5-a1`, `rgba8`, `rgb10-a2`, `rgba12`, `rgba16`, `rgb2-ext`, `dual-alpha4-sgis`, `dual-alpha8-sgis`, `dual-alpha12-sgis`, `dual-alpha16-sgis`, `dual-luminance4-sgis`, `dual-luminance8-sgis`, `dual-luminance12-sgis`, `dual-luminance16-sgis`, `dual-intensity4-sgis`, `dual-intensity8-sgis`, `dual-intensity12-sgis`, `dual-intensity16-sgis`, `dual-luminance-alpha4-sgis`, `dual-luminance-alpha8-sgis`, `quad-alpha4-sgis`, `quad-alpha8-sgis`, `quad-luminance4-sgis`,

`quad-luminance8-sgis`, `quad-intensity4-sgis`, `quad-intensity8-sgis`, `depth-component16-sgix`, `depth-component24-sgix`, `depth-component32-sgix`.

`oes-rgb-8-rgba-8` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`rgb8`, `rgba8`.

`interleaved-array-format` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`v2f`, `v3f`, `c4ub-v2f`, `c4ub-v3f`, `c3f-v3f`, `n3f-v3f`, `c4f-n3f-v3f`, `t2f-v3f`, `t4f-v4f`, `t2f-c4ub-v3f`, `t2f-c3f-v3f`, `t2f-n3f-v3f`, `t2f-c4f-n3f-v3f`, `t4f-c4f-n3f-v4f`.

`vertex-pointer-type` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`short`, `int`, `float`, `double`.

`clip-plane-name` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`clip-plane0`, `clip-plane1`, `clip-plane2`, `clip-plane3`, `clip-plane4`, `clip-plane5`.

`light-name` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`light0`, `light1`, `light2`, `light3`, `light4`, `light5`, `light6`, `light7`, `fragment-light0-sgix`, `fragment-light1-sgix`, `fragment-light2-sgix`, `fragment-light3-sgix`, `fragment-light4-sgix`, `fragment-light5-sgix`, `fragment-light6-sgix`, `fragment-light7-sgix`.

`ext-abgr` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`abgr-ext`.

`version-1-2` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

constant-color, one-minus-constant-color, constant-alpha, one-minus-constant-alpha, blend-color, func-add, func-add-ext, min, min-ext, max, max-ext, blend-equation, blend-equation-ext, func-subtract, func-subtract-ext, func-reverse-subtract, func-reverse-subtract-ext, convolution-1d, convolution-2d, separable-2d, convolution-border-mode, convolution-filter-scale, convolution-filter-bias, reduce, convolution-format, convolution-width, convolution-height, max-convolution-width, max-convolution-height, post-convolution-red-scale, post-convolution-green-scale, post-convolution-blue-scale, post-convolution-alpha-scale, post-convolution-red-bias, post-convolution-green-bias, post-convolution-blue-bias, post-convolution-alpha-bias, histogram, proxy-histogram, histogram-width, histogram-format, histogram-red-size, histogram-green-size, histogram-blue-size, histogram-alpha-size, histogram-sink, minmax, minmax-format, minmax-sink, table-too-large, unsigned-byte-3-3-2, unsigned-short-4-4-4-4, unsigned-short-5-5-5-1, unsigned-int-8-8-8-8, unsigned-int-10-10-10-2, unsigned-byte-2-3-3-rev, unsigned-short-5-6-5, unsigned-short-5-6-5-rev, unsigned-short-4-4-4-4-rev, unsigned-short-1-5-5-5-rev, unsigned-int-8-8-8-8-rev, unsigned-int-2-10-10-10-rev, rescale-normal, pack-skip-images, pack-image-height, unpack-skip-images, unpack-image-height, texture-3d, proxy-texture-3d, texture-depth, texture-wrap-r, max-3d-texture-size, color-matrix, color-matrix-stack-depth, max-color-matrix-stack-depth, post-color-matrix-red-scale, post-color-matrix-green-scale, post-color-matrix-blue-scale, post-color-matrix-alpha-scale, post-color-matrix-red-bias, post-color-matrix-green-bias, post-color-matrix-blue-bias, post-color-matrix-alpha-bias, color-table, post-convolution-color-table, post-color-matrix-color-table, proxy-color-table, proxy-post-convolution-color-table, proxy-post-color-matrix-color-table, color-table-scale, color-table-bias, color-table-format, color-table-width, color-table-red-size, color-table-green-size, color-table-blue-size, color-table-alpha-size, color-table-luminance-size, color-table-intensity-size, bgr, bgra, max-elements-vertices, max-elements-indices, clamp-to-edge, texture-min-lod, texture-max-lod, texture-base-level, texture-max-level, constant-border, replicate-border, convolution-border-color, light-model-color-control, single-color, separate-specular-color, smooth-point-size-range, smooth-point-size-granularity, smooth-line-width-range, smooth-line-width-granularity, aliased-point-size-range, aliased-line-width-range.

`ext-blend-color` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

constant-color-ext, one-minus-constant-color-ext, constant-alpha-ext, one-minus-constant-alpha-ext, blend-color-ext.

`ext-blend-minmax` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`func-add`, `func-add-ext`, `min`, `min-ext`, `max`, `max-ext`, `blend-equation`, `blend-equation-ext`.

`version-2-0` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`blend-equation-rgb`, `vertex-attrib-array-enabled`, `vertex-attrib-array-size`, `vertex-attrib-array-stride`, `vertex-attrib-array-type`, `current-vertex-attrib`, `vertex-program-point-size`, `vertex-program-two-side`, `vertex-attrib-array-pointer`, `stencil-back-func`, `stencil-back-fail`, `stencil-back-pass-depth-fail`, `stencil-back-pass-depth-pass`, `stencil-back-fail-ati`, `max-draw-buffers`, `draw-buffer0`, `draw-buffer1`, `draw-buffer2`, `draw-buffer3`, `draw-buffer4`, `draw-buffer5`, `draw-buffer6`, `draw-buffer7`, `draw-buffer8`, `draw-buffer9`, `draw-buffer10`, `draw-buffer11`, `draw-buffer12`, `draw-buffer13`, `draw-buffer14`, `draw-buffer15`, `blend-equation-alpha`, `point-sprite`, `coord-replace`, `max-vertex-attribs`, `vertex-attrib-array-normalized`, `max-texture-coords`, `max-texture-image-units`, `fragment-shader`, `fragment-shader-arb`, `vertex-shader`, `vertex-shader-arb`, `program-object-arb`, `shader-object-arb`, `max-fragment-uniform-components`, `max-fragment-uniform-components-arb`, `max-vertex-uniform-components`, `max-vertex-uniform-components-arb`, `max-varying-floats`, `max-varying-floats-arb`, `max-vertex-texture-image-units`, `max-vertex-texture-image-units-arb`, `max-combined-texture-image-units`, `max-combined-texture-image-units-arb`, `object-type-arb`, `shader-type`, `object-subtype-arb`, `float-vec2`, `float-vec2-arb`, `float-vec3`, `float-vec3-arb`, `float-vec4`, `float-vec4-arb`, `int-vec2`, `int-vec2-arb`, `int-vec3`, `int-vec3-arb`, `int-vec4`, `int-vec4-arb`, `bool`, `bool-arb`, `bool-vec2`, `bool-vec2-arb`, `bool-vec3`, `bool-vec3-arb`, `bool-vec4`, `bool-vec4-arb`, `float-mat2`, `float-mat2-arb`, `float-mat3`, `float-mat3-arb`, `float-mat4`, `float-mat4-arb`, `sampler-1d`, `sampler-1d-arb`, `sampler-2d`, `sampler-2d-arb`, `sampler-3d`, `sampler-3d-arb`, `sampler-cube`, `sampler-cube-arb`, `sampler-1d-shadow`, `sampler-1d-shadow-arb`, `sampler-2d-shadow`, `sampler-2d-shadow-arb`, `sampler-2d-rect-arb`, `sampler-2d-rect-shadow-arb`, `float-mat-2x-3`, `float-mat-2x-4`, `float-mat-3x-2`, `float-mat-3x-4`, `float-mat-4x-2`, `float-mat-4x-3`, `delete-status`, `object-delete-status-arb`, `compile-status`, `object-compile-status-arb`, `link-status`, `object-link-status-arb`, `validate-status`, `object-validate-status-arb`, `info-log-length`, `object-info-log-length-arb`, `attached-shaders`, `object-attached-objects-arb`, `active-uniforms`, `object-active-uniforms-arb`, `active-uniform-max-length`, `object-active-uniform-max-length-arb`, `shader-source-length`, `object-shader-source-length-arb`, `active-attributes`, `object-active-attributes-arb`, `active-attribute-max-length`, `object-active-attribute-max-length-arb`,

`fragment-shader-derivative-hint`, `fragment-shader-derivative-hint-arb`, `shading-language-version`, `shading-language-version-arb`, `current-program`, `point-sprite-coord-origin`, `lower-left`, `upper-left`, `stencil-back-ref`, `stencil-back-value-mask`, `stencil-back-writemask`.

`ext-blend-equation-separate` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`blend-equation-rgb-ext`, `blend-equation-alpha-ext`.

`oes-blend-equation-separate` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`blend-equation-rgb-oes`, `blend-equation-alpha-oes`.

`ext-blend-subtract` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`func-subtract`, `func-subtract-ext`, `func-reverse-subtract`, `func-reverse-subtract-ext`.

`oes-blend-subtract` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`func-add-oes`, `blend-equation-oes`, `func-subtract-oes`, `func-reverse-subtract-oes`.

`ext-cmyka` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`cmyk-ext`, `cmyka-ext`, `pack-cmyk-hint-ext`, `unpack-cmyk-hint-ext`.

`ext-convolution` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`convolution-1d-ext`, `convolution-2d-ext`, `separable-2d-ext`, `convolution-border-mode-ext`, `convolution-filter-scale-ext`, `convolution-filter-bias-ext`, `reduce-ext`, `convolution-format-ext`, `convolution-width-ext`, `convolution-height-ext`, `max-convolution-width-ext`, `max-convolution-height-ext`, `post-convolution-red-scale-ext`, `post-convolution-green-scale-ext`, `post-convolution-blue-scale-ext`, `post-convolution-alpha-scale-ext`, `post-convolution-red-bias-ext`, `post-convolution-green-bias-ext`, `post-convolution-blue-bias-ext`, `post-convolution-alpha-bias-ext`.

`ext-histogram` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`histogram-ext`, `proxy-histogram-ext`, `histogram-width-ext`, `histogram-format-ext`, `histogram-red-size-ext`, `histogram-green-size-ext`, `histogram-blue-size-ext`, `histogram-alpha-size-ext`, `histogram-luminance-size`, `histogram-luminance-size-ext`, `histogram-sink-ext`, `minmax-ext`, `minmax-format-ext`, `minmax-sink-ext`, `table-too-large-ext`.

`ext-packed-pixels` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`unsigned-byte-3-3-2-ext`, `unsigned-short-4-4-4-4-ext`, `unsigned-short-5-5-5-1-ext`, `unsigned-int-8-8-8-8-ext`, `unsigned-int-10-10-10-2-ext`, `unsigned-byte-2-3-3-rev-ext`, `unsigned-short-5-6-5-ext`, `unsigned-short-5-6-5-rev-ext`, `unsigned-short-4-4-4-4-rev-ext`, `unsigned-short-1-5-5-5-rev-ext`, `unsigned-int-8-8-8-8-rev-ext`, `unsigned-int-2-10-10-10-rev-ext`.

`ext-texture-type-2-10-10-10-rev` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`polygon-offset-ext`, `polygon-offset-factor-ext`, `polygon-offset-bias-ext`.

`ext-polygon-offset` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`polygon-offset-ext`, `polygon-offset-factor-ext`, `polygon-offset-bias-ext`.

`ext-rescale-normal` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`rescale-normal-ext`.

`ext-texture` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`alpha4-ext`, `alpha8-ext`, `alpha12-ext`, `alpha16-ext`, `luminance4-ext`, `luminance8-ext`, `luminance12-ext`, `luminance16-ext`, `luminance4-alpha4-ext`, `luminance6-alpha2-ext`, `luminance8-alpha8-ext`, `luminance12-alpha4-ext`, `luminance12-alpha12-ext`, `luminance16-alpha16-ext`, `intensity-ext`, `intensity4-ext`, `intensity8-ext`, `intensity12-ext`, `intensity16-ext`,

`rgb2-ext`, `rgb4-ext`, `rgb5-ext`, `rgb8-ext`, `rgb10-ext`, `rgb12-ext`, `rgb16-ext`, `rgba2-ext`, `rgba4-ext`, `rgb5-a1-ext`, `rgba8-ext`, `rgb10-a2-ext`, `rgba12-ext`, `rgba16-ext`, `texture-red-size-ext`, `texture-green-size-ext`, `texture-blue-size-ext`, `texture-alpha-size-ext`, `texture-luminance-size-ext`, `texture-intensity-size-ext`, `replace-ext`, `proxy-texture-1d-ext`, `proxy-texture-2d-ext`, `texture-too-large-ext`.

`ext-texture-object` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-priority-ext`, `texture-resident-ext`, `texture-1d-binding-ext`, `texture-2d-binding-ext`, `texture-3d-binding-ext`.

`ext-texture-3d` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`pack-skip-images-ext`, `pack-image-height-ext`, `unpack-skip-images-ext`, `unpack-image-height-ext`, `texture-3d-ext`, `proxy-texture-3d-ext`, `texture-depth-ext`, `texture-wrap-r-ext`, `max-3d-texture-size-ext`.

`oes-texture-3d` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-3d-binding-oes`, `texture-3d-oes`, `texture-wrap-r-oes`, `max-3d-texture-size-oes`, `sampler-3d-oes`, `framebuffer-attachment-texture-3d-zoffset-oes`.

`ext-vertex-array` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`vertex-array-ext`, `normal-array-ext`, `color-array-ext`, `index-array-ext`, `texture-coord-array-ext`, `edge-flag-array-ext`, `vertex-array-size-ext`, `vertex-array-type-ext`, `vertex-array-stride-ext`, `vertex-array-count-ext`, `normal-array-type-ext`, `normal-array-stride-ext`, `normal-array-count-ext`, `color-array-size-ext`, `color-array-type-ext`, `color-array-stride-ext`, `color-array-count-ext`, `index-array-type-ext`, `index-array-stride-ext`, `index-array-count-ext`, `texture-coord-array-size-ext`, `texture-coord-array-type-ext`, `texture-coord-array-stride-ext`, `texture-coord-array-count-ext`, `edge-flag-array-stride-ext`, `edge-flag-array-count-ext`, `vertex-array-pointer-ext`, `normal-array-pointer-ext`, `color-array-pointer-ext`, `index-array-pointer-ext`, `texture-coord-array-pointer-ext`, `edge-flag-array-pointer-ext`.

`sgix-interlace` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`interlace-sgix`.

`sgis-detail-texture` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`detail-texture-2d-sgis`, `detail-texture-2d-binding-sgis`, `linear-detail-sgis`, `linear-detail-alpha-sgis`, `linear-detail-color-sgis`, `detail-texture-level-sgis`, `detail-texture-mode-sgis`, `detail-texture-func-points-sgis`.

`sgis-multisample` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`multisample-sgis`, `sample-alpha-to-mask-sgis`, `sample-alpha-to-one-sgis`, `sample-mask-sgis`, `1pass-sgis`, `2pass-0-sgis`, `2pass-1-sgis`, `4pass-0-sgis`, `4pass-1-sgis`, `4pass-2-sgis`, `4pass-3-sgis`, `sample-buffers-sgis`, `samples-sgis`, `sample-mask-value-sgis`, `sample-mask-invert-sgis`, `sample-pattern-sgis`.

`nv-multisample-coverage` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`coverage-samples-nv`, `color-samples-nv`.

`sgis-sharpen-texture` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`linear-sharpen-sgis`, `linear-sharpen-alpha-sgis`, `linear-sharpen-color-sgis`, `sharpen-texture-func-points-sgis`.

`sgi-color-matrix` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`color-matrix-sgi`, `color-matrix-stack-depth-sgi`, `max-color-matrix-stack-depth-sgi`, `post-color-matrix-red-scale-sgi`, `post-color-matrix-green-scale-sgi`, `post-color-matrix-blue-scale-sgi`, `post-color-matrix-alpha-scale-sgi`, `post-color-matrix-red-bias-sgi`, `post-color-matrix-green-bias-sgi`, `post-color-matrix-blue-bias-sgi`, `post-color-matrix-alpha-bias-sgi`.

`sgi-texture-color-table` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-color-table-sgi`, `proxy-texture-color-table-sgi`.

`sgix-texture-add-env` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-env-bias-sgix`.

`sgix-shadow-ambient` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`shadow-ambient-sgix`.

`version-1-4` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`blend-dst-rgb`, `blend-src-rgb`, `blend-dst-alpha`, `blend-src-alpha`, `point-size-min`, `point-size-max`, `point-fade-threshold-size`, `point-distance-attenuation`, `generate-mipmap`, `generate-mipmap-hint`, `depth-component16`, `depth-component24`, `depth-component32`, `mirrored-repeat`, `fog-coordinate-source`, `fog-coordinate`, `fragment-depth`, `current-fog-coordinate`, `fog-coordinate-array-type`, `fog-coordinate-array-stride`, `fog-coordinate-array-pointer`, `fog-coordinate-array`, `color-sum`, `current-secondary-color`, `secondary-color-array-size`, `secondary-color-array-type`, `secondary-color-array-stride`, `secondary-color-array-pointer`, `secondary-color-array`, `max-texture-lod-bias`, `texture-filter-control`, `texture-lod-bias`, `incr-wrap`, `decr-wrap`, `texture-depth-size`, `depth-texture-mode`, `texture-compare-mode`, `texture-compare-func`, `compare-r-to-texture`.

`ext-blend-func-separate` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`blend-dst-rgb-ext`, `blend-src-rgb-ext`, `blend-dst-alpha-ext`, `blend-src-alpha-ext`.

`oes-blend-func-separate` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`blend-dst-rgb-oes`, `blend-src-rgb-oes`, `blend-dst-alpha-oes`, `blend-src-alpha-oes`.

`ext-422-pixels` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`422-ext`, `422-rev-ext`, `422-average-ext`, `422-rev-average-ext`.

`sgi-color-table` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`color-table-sgi`, `post-convolution-color-table-sgi`, `post-color-matrix-color-table-sgi`, `proxy-color-table-sgi`, `proxy-post-convolution-color-table-sgi`, `proxy-post-color-matrix-color-table-sgi`, `color-table-scale-sgi`, `color-table-bias-sgi`, `color-table-format-sgi`, `color-table-width-sgi`, `color-table-red-size-sgi`, `color-table-green-size-sgi`, `color-table-blue-size-sgi`, `color-table-alpha-size-sgi`, `color-table-luminance-size-sgi`, `color-table-intensity-size-sgi`.

`arb-vertex-array-bgra` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`bgr-ext`, `bgra-ext`.

`ext-bgra` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`bgr-ext`, `bgra-ext`.

`sgis-texture-select` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`dual-alpha4-sgis`, `dual-alpha8-sgis`, `dual-alpha12-sgis`, `dual-alpha16-sgis`, `dual-luminance4-sgis`, `dual-luminance8-sgis`, `dual-luminance12-sgis`, `dual-luminance16-sgis`, `dual-intensity4-sgis`, `dual-intensity8-sgis`, `dual-intensity12-sgis`, `dual-intensity16-sgis`, `dual-luminance-alpha4-sgis`, `dual-luminance-alpha8-sgis`, `quad-alpha4-sgis`, `quad-alpha8-sgis`, `quad-luminance4-sgis`, `quad-luminance8-sgis`, `quad-intensity4-sgis`, `quad-intensity8-sgis`, `dual-texture-select-sgis`, `quad-texture-select-sgis`.

`arb-point-parameters` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`point-size-min-arb`, `point-size-max-arb`, `point-fade-threshold-size-arb`, `point-distance-attenuation-arb`.

`ext-point-parameters` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`point-size-min-ext`, `point-size-max-ext`, `point-fade-threshold-size-ext`, `distance-attenuation-ext`.

`sgis-point-parameters` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`point-size-min-sgis`, `point-size-max-sgis`, `point-fade-threshold-size-sgis`, `distance-attenuation-sgis`.

`sgis-fog-function` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`fog-func-sgis`, `fog-func-points-sgis`, `max-fog-func-points-sgis`.

`arb-texture-border-clamp` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`clamp-to-border-arb`.

`sgis-texture-border-clamp` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`clamp-to-border-sgis`.

`sgix-texture-multi-buffer` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-multi-buffer-hint-sgix`.

`sgis-texture-edge-clamp` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`clamp-to-edge-sgis`.

`sgis-texture-4d` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`pack-skip-volumes-sgis`, `pack-image-depth-sgis`, `unpack-skip-volumes-sgis`, `unpack-image-depth-sgis`, `texture-4d-sgis`, `proxy-texture-4d-sgis`, `texture-4dsize-sgis`, `texture-wrap-q-sgis`, `max-4d-texture-size-sgis`, `texture-4d-binding-sgis`.

`sgix-pixel-texture` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`pixel-tex-gen-sgix`, `pixel-tex-gen-mode-sgix`.

`sgis-texture-lod` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-min-lod-sgis`, `texture-max-lod-sgis`, `texture-base-level-sgis`, `texture-max-level-sgis`.

`sgix-pixel-tiles` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`pixel-tile-best-alignment-sgix`, `pixel-tile-cache-increment-sgix`, `pixel-tile-width-sgix`, `pixel-tile-height-sgix`, `pixel-tile-grid-width-sgix`, `pixel-tile-grid-height-sgix`, `pixel-tile-grid-depth-sgix`, `pixel-tile-cache-size-sgix`.

`sgis-texture-filter-4` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`filter4-sgis`, `texture-filter4-size-sgis`.

`sgix-sprite` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`sprite-sgix`, `sprite-mode-sgix`, `sprite-axis-sgix`, `sprite-translation-sgix`, `sprite-axial-sgix`, `sprite-object-aligned-sgix`, `sprite-eye-aligned-sgix`.

`hp-convolution-border-modes` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`ignore-border-hp`, `constant-border-hp`, `replicate-border-hp`, `convolution-border-color-hp`.

`sgix-clipmap` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`linear-clipmap-linear-sgix`, `texture-clipmap-center-sgix`, `texture-clipmap-frame-sgix`, `texture-clipmap-offset-sgix`, `texture-clipmap-virtual-depth-sgix`, `texture-clipmap-lod-offset-sgix`, `texture-clipmap-depth-sgix`, `max-clipmap-depth-sgix`, `max-clipmap-virtual-depth-sgix`, `nearest-clipmap-nearest-sgix`, `nearest-clipmap-linear-sgix`, `linear-clipmap-nearest-sgix`.

`sgix-texture-scale-bias` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`post-texture-filter-bias-sgix`, `post-texture-filter-scale-sgix`, `post-texture-filter-bias-range-sgix`, `post-texture-filter-scale-range-sgix`.

`sgix-reference-plane` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`reference-plane-sgix`, `reference-plane-equation-sgix`.

`sgix-ir-instrument-1` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`ir-instrument1-sgix`.

`sgix-instruments` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`instrument-buffer-pointer-sgix`, `instrument-measurements-sgix`.

`sgix-list-priority` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`list-priority-sgix`.

`sgix-calligraphic-fragment` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`calligraphic-fragment-sgix`.

`sgix-impact-pixel-texture` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`pixel-tex-gen-q-ceiling-sgix`, `pixel-tex-gen-q-round-sgix`, `pixel-tex-gen-q-floor-sgix`, `pixel-tex-gen-alpha-replace-sgix`, `pixel-tex-gen-alpha-no-replace-sgix`, `pixel-tex-gen-alpha-ls-sgix`, `pixel-tex-gen-alpha-ms-sgix`.

`sgix-framezoom` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`framezoom-sgix`, `framezoom-factor-sgix`, `max-framezoom-factor-sgix`.

`sgix-texture-lod-bias` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-lod-bias-s-sgix`, `texture-lod-bias-t-sgix`, `texture-lod-bias-r-sgix`.

`sgis-generate-mipmap` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`generate-mipmap-sgis`, `generate-mipmap-hint-sgis`.

`sgix-polynomial-ffd` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`geometry-deformation-sgix`, `texture-deformation-sgix`, `deformations-mask-sgix`, `max-deformation-order-sgix`.

`sgix-fog-offset` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`fog-offset-sgix`, `fog-offset-value-sgix`.

`sgix-shadow` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-compare-sgix`, `texture-compare-operator-sgix`, `texture-lequal-r-sgix`, `texture-gequal-r-sgix`.

`arb-depth-texture` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`depth-component16-arb`, `depth-component24-arb`, `depth-component32-arb`, `texture-depth-size-arb`, `depth-texture-mode-arb`.

`sgix-depth-texture` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`depth-component16-sgix`, `depth-component24-sgix`, `depth-component32-sgix`.

`oes-depth-24` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`depth-component24-oes`.

`oes-depth-32` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`depth-component32-oes`.

`ext-compiled-vertex-array` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`array-element-lock-first-ext`, `array-element-lock-count-ext`.

`ext-cull-vertex` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`cull-vertex-ext`, `cull-vertex-eye-position-ext`, `cull-vertex-object-position-ext`.

`ext-index-array-formats` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`iui-v2f-ext`, `iui-v3f-ext`, `iui-n3f-v2f-ext`, `iui-n3f-v3f-ext`, `t2f-iui-v2f-ext`, `t2f-iui-v3f-ext`, `t2f-iui-n3f-v2f-ext`, `t2f-iui-n3f-v3f-ext`.

`ext-index-func` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`index-test-ext`, `index-test-func-ext`, `index-test-ref-ext`.

`ext-index-material` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`index-material-ext`, `index-material-parameter-ext`, `index-material-face-ext`.

`sgix-ycrcb` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`ycrcb-422-sgix`, `ycrcb-444-sgix`.

`sunx-general-triangle-list` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`restart-sun`, `replace-middle-sun`, `replace-oldest-sun`, `wrap-border-sun`, `triangle-list-sun`, `replacement-code-sun`, `replacement-code-array-sun`, `replacement-code-array-type-sun`, `replacement-code-array-stride-sun`, `replacement-code-array-pointer-sun`, `r1ui-v3f-sun`, `r1ui-c4ub-v3f-sun`, `r1ui-c3f-v3f-sun`, `r1ui-n3f-v3f-sun`, `r1ui-c4f-n3f-v3f-sun`, `r1ui-t2f-v3f-sun`, `r1ui-t2f-n3f-v3f-sun`, `r1ui-t2f-c4f-n3f-v3f-sun`.

`sunx-constant-data` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`unpack-constant-data-sunx`, `texture-constant-data-sunx`.

`sun-global-alpha` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`global-alpha-sun`, `global-alpha-factor-sun`.

`sgis-texture-color-mask` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-color-writemask-sgis`.

`sgis-point-line-texgen` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`eye-distance-to-point-sgis`, `object-distance-to-point-sgis`, `eye-distance-to-line-sgis`, `object-distance-to-line-sgis`, `eye-point-sgis`, `object-point-sgis`, `eye-line-sgis`, `object-line-sgis`.

`ext-separate-specular-color` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`light-model-color-control-ext`, `single-color-ext`, `separate-specular-color-ext`.

`ext-shared-texture-palette` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`shared-texture-palette-ext`.

`ati-text-fragment-shader` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`text-fragment-shader-ati`.

`ext-color-buffer-half-float` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`framebuffer-attachment-component-type-ext`, `r16f-ext`, `rg16f-ext`, `rgba16f-ext`, `rgb16f-ext`, `unsigned-normalized-ext`.

`oes-surfaceless-context` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`framebuffer-undefined-oes`.

`arb-texture-rg` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`rg`, `rg-integer`, `r8`, `r16`, `rg8`, `rg16`, `r16f`, `r32f`, `rg16f`, `rg32f`, `r8i`, `r8ui`, `r16i`, `r16ui`, `r32i`, `r32ui`, `rg8i`, `rg8ui`, `rg16i`, `rg16ui`, `rg32i`, `rg32ui`.

`arb-cl-event` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`sync-cl-event-arb`, `sync-cl-event-complete-arb`.

`arb-debug-output` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`debug-output-synchronous-arb`, `debug-next-logged-message-length-arb`, `debug-callback-function-arb`, `debug-callback-user-param-arb`, `debug-source-api-arb`, `debug-source-window-system-arb`, `debug-source-shader-compiler-arb`, `debug-source-third-party-arb`, `debug-source-application-arb`, `debug-source-other-arb`, `debug-type-error-arb`, `debug-type-deprecated-behavior-arb`, `debug-type-undefined-behavior-arb`, `debug-type-portability-arb`, `debug-type-performance-arb`, `debug-type-other-arb`, `max-debug-message-length-arb`, `max-debug-logged-messages-arb`, `debug-logged-messages-arb`, `debug-severity-high-arb`, `debug-severity-medium-arb`, `debug-severity-low-arb`.

`arb-get-program-binary` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`program-binary-retrievable-hint`, `program-binary-length`, `num-program-binary-formats`, `program-binary-formats`.

`arb-viewport-array` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`max-viewports`, `viewport-subpixel-bits`, `viewport-bounds-range`, `layer-provoking-vertex`, `viewport-index-provoking-vertex`, `undefined-vertex`, `first-vertex-convention`, `last-vertex-convention`, `provoking-vertex`.

`arb-explicit-uniform-location` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`max-uniform-locations`.

`arb-internalformat-query-2` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`internalformat-supported`, `internalformat-preferred`, `internalformat-red-size`, `internalformat-green-size`, `internalformat-blue-size`, `internalformat-alpha-size`, `internalformat-depth-size`, `internalformat-stencil-size`, `internalformat-shared-size`, `internalformat-red-type`, `internalformat-green-type`, `internalformat-blue-type`, `internalformat-alpha-type`, `internalformat-depth-type`, `internalformat-stencil-type`, `max-width`, `max-height`, `max-depth`, `max-layers`, `max-combined-dimensions`,

color-components, depth-components, stencil-components, color-renderable, depth-renderable, stencil-renderable, framebuffer-renderable, framebuffer-renderable-layered, framebuffer-blend, read-pixels, read-pixels-format, read-pixels-type, texture-image-format, texture-image-type, get-texture-image-format, get-texture-image-type, mipmap, manual-generate-mipmap, auto-generate-mipmap, color-encoding, srgb-read, srgb-write, srgb-decode-arb, filter, vertex-texture, tess-control-texture, tess-evaluation-texture, geometry-texture, fragment-texture, compute-texture, texture-shadow, texture-gather, texture-gather-shadow, shader-image-load, shader-image-store, shader-image-atomic, image-texel-size, image-compatibility-class, image-pixel-format, image-pixel-type, simultaneous-texture-and-depth-test, simultaneous-texture-and-stencil-test, simultaneous-texture-and-depth-write, simultaneous-texture-and-stencil-write, texture-compressed-block-width, texture-compressed-block-height, texture-compressed-block-size, clear-buffer, texture-view, view-compatibility-class, full-support, caveat-support, image-class-4-x-32, image-class-2-x-32, image-class-1-x-32, image-class-4-x-16, image-class-2-x-16, image-class-1-x-16, image-class-4-x-8, image-class-2-x-8, image-class-1-x-8, image-class-11-11-10, image-class-10-10-10-2, view-class-128-bits, view-class-96-bits, view-class-64-bits, view-class-48-bits, view-class-32-bits, view-class-24-bits, view-class-16-bits, view-class-8-bits, view-class-s3tc-dxt1-rgb, view-class-s3tc-dxt1-rgba, view-class-s3tc-dxt3-rgba, view-class-s3tc-dxt5-rgba, view-class-rgtc1-red, view-class-rgtc2-rg, view-class-bptc-unorm, view-class-bptc-float.

`arb-vertex-attrib-binding` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

vertex-attrib-binding, vertex-attrib-relative-offset, vertex-binding-divisor, vertex-binding-offset, vertex-binding-stride, max-vertex-attrib-relative-offset, max-vertex-attrib-bindings.

`arb-texture-view` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

texture-view-min-level, texture-view-num-levels, texture-view-min-layer, texture-view-num-layers, texture-immutable-levels.

`sgix-depth-pass-instrument` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

depth-pass-instrument-sgix, depth-pass-instrument-counters-sgix, depth-pass-instrument-max-sgix.

`sgix-fragments-instrument` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`fragments-instrument-sgix`, `fragments-instrument-counters-sgix`, `fragments-instrument-max-sgix`.

`sgix-convolution-accuracy` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`convolution-hint-sgix`.

`sgix-ycrcba` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`ycrcb-sgix`, `ycrcba-sgix`.

`sgix-slim` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`unpack-compressed-size-sgix`, `pack-max-compressed-size-sgix`, `pack-compressed-size-sgix`, `slim8u-sgix`, `slim10u-sgix`, `slim12s-sgix`.

`sgix-blend-alpha-minmax` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`alpha-min-sgix`, `alpha-max-sgix`.

`sgix-scalebias-hint` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`scalebias-hint-sgix`.

`sgix-async` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`async-marker-sgix`.

`sgix-async-histogram` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`async-histogram-sgix`, `max-async-histogram-sgix`.

`ext-pixel-transform` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`pixel-transform-2d-ext`, `pixel-mag-filter-ext`, `pixel-min-filter-ext`, `pixel-cubic-weight-ext`, `cubic-ext`, `average-ext`, `pixel-transform-2d-stack-depth-ext`, `max-pixel-transform-2d-stack-depth-ext`, `pixel-transform-2d-matrix-ext`.

`ext-light-texture` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`fragment-material-ext`, `fragment-normal-ext`, `fragment-color-ext`, `attenuation-ext`, `shadow-attenuation-ext`, `texture-application-mode-ext`, `texture-light-ext`, `texture-material-face-ext`, `texture-material-parameter-ext`, `fragment-depth-ext`.

`sgis-pixel-texture` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`pixel-texture-sgis`, `pixel-fragment-rgb-source-sgis`, `pixel-fragment-alpha-source-sgis`, `pixel-group-color-sgis`.

`sgix-line-quality-hint` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`line-quality-hint-sgix`.

`sgix-async-pixel` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`async-tex-image-sgix`, `async-draw-pixels-sgix`, `async-read-pixels-sgix`, `max-async-tex-image-sgix`, `max-async-draw-pixels-sgix`, `max-async-read-pixels-sgix`.

`sgix-texture-coordinate-clamp` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-max-clamp-s-sgix`, `texture-max-clamp-t-sgix`, `texture-max-clamp-r-sgix`, `fog-factor-to-alpha-sgix`.

`arb-texture-mirrored-repeat` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`mirrored-repeat-arb`.

`ibm-texture-mirrored-repeat` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`mirrored-repeat-ibm`.

`oes-texture-mirrored-repeat` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`mirrored-repeat-oes`.

`s3-s-3-tc` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`rgb-s3tc`, `rgb4-s3tc`, `rgba-s3tc`, `rgba4-s3tc`, `rgba-dxt5-s3tc`, `rgba4-dxt5-s3tc`.

`sgix-vertex-preclip` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`vertex-preclip-sgix`, `vertex-preclip-hint-sgix`.

`ext-texture-compression-s-3-tc` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`compressed-rgb-s3tc-dxt1-ext`, `compressed-rgba-s3tc-dxt1-ext`, `compressed-rgba-s3tc-dxt3-ext`, `compressed-rgba-s3tc-dxt5-ext`.

`angle-texture-compression-dxt-3` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`compressed-rgba-s3tc-dxt3-angle`.

`angle-texture-compression-dxt-5` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`compressed-rgba-s3tc-dxt5-angle`.

`intel-parallel-arrays` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`parallel-arrays-intel`, `vertex-array-parallel-pointers-intel`, `normal-array-parallel-pointers-intel`, `color-array-parallel-pointers-intel`, `texture-coord-array-parallel-pointers-intel`.

`sgix-fragment-lighting` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`fragment-lighting-sgix`, `fragment-color-material-sgix`, `fragment-color-material-face-sgix`, `fragment-color-material-parameter-sgix`, `max-fragment-lights-sgix`, `max-active-lights-sgix`, `current-raster-normal-sgix`, `light-env-mode-sgix`, `fragment-light-model-local-viewer-sgix`, `fragment-light-model-two-side-sgix`, `fragment-light-model-ambient-sgix`, `fragment-light-model-normal-interpolation-sgix`, `fragment-light0-sgix`, `fragment-light1-sgix`, `fragment-light2-sgix`, `fragment-light3-sgix`, `fragment-light4-sgix`, `fragment-light5-sgix`, `fragment-light6-sgix`, `fragment-light7-sgix`.

`sgix-resample` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`pack-resample-sgix`, `unpack-resample-sgix`, `resample-replicate-sgix`, `resample-zero-fill-sgix`, `resample-decimate-sgix`.

`version-1-5` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`fog-coord-src`, `fog-coord`, `current-fog-coord`, `fog-coord-array-type`, `fog-coord-array-stride`, `fog-coord-array-pointer`, `fog-coord-array`, `src0-rgb`, `src1-rgb`, `src2-rgb`, `src0-alpha`, `src1-alpha`, `src2-alpha`, `buffer-size`, `buffer-usage`, `query-counter-bits`, `current-query`, `query-result`, `query-result-available`, `array-buffer`, `element-array-buffer`, `array-buffer-binding`, `element-array-buffer-binding`, `vertex-array-buffer-binding`, `normal-array-buffer-binding`, `color-array-buffer-binding`, `index-array-buffer-binding`, `texture-coord-array-buffer-binding`, `edge-flag-array-buffer-binding`, `secondary-color-array-buffer-binding`, `fog-coord-array-buffer-binding`, `fog-coordinate-array-buffer-binding`, `weight-array-buffer-binding`, `vertex-attrib-array-buffer-binding`, `read-only`, `write-only`, `read-write`, `buffer-access`, `buffer-mapped`, `buffer-map-pointer`, `stream-draw`, `stream-read`, `stream-copy`, `static-draw`, `static-read`, `static-copy`, `dynamic-draw`, `dynamic-read`, `dynamic-copy`, `samples-passed`.

`ext-fog-coord` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`fog-coordinate-source-ext`, `fog-coordinate-ext`, `fragment-depth-ext`, `current-fog-coordinate-ext`, `fog-coordinate-array-type-ext`, `fog-coordinate-array-stride-ext`, `fog-coordinate-array-pointer-ext`, `fog-coordinate-array-ext`.

`ext-secondary-color` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`color-sum-ext`, `current-secondary-color-ext`, `secondary-color-array-size-ext`, `secondary-color-array-type-ext`, `secondary-color-array-stride-ext`, `secondary-color-array-pointer-ext`, `secondary-color-array-ext`.

`arb-vertex-program` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`color-sum-arb`, `vertex-program-arb`, `vertex-attrib-array-enabled-arb`, `vertex-attrib-array-size-arb`, `vertex-attrib-array-stride-arb`, `vertex-attrib-array-type-arb`, `current-vertex-attrib-arb`, `program-length-arb`, `program-string-arb`, `max-program-matrix-stack-depth-arb`, `max-program-matrices-arb`, `current-matrix-stack-depth-arb`, `current-matrix-arb`, `vertex-program-point-size-arb`, `vertex-program-two-side-arb`, `vertex-attrib-array-pointer-arb`, `program-error-position-arb`, `program-binding-arb`, `max-vertex-attribs-arb`, `vertex-attrib-array-normalized-arb`, `max-texture-coords-arb`, `max-texture-image-units-arb`, `program-error-string-arb`, `program-format-ascii-arb`, `program-format-arb`, `program-instructions-arb`, `max-program-instructions-arb`, `program-native-instructions-arb`, `max-program-native-instructions-arb`, `program-temporaries-arb`, `max-program-temporaries-arb`, `program-native-temporaries-arb`, `max-program-native-temporaries-arb`, `program-parameters-arb`, `max-program-parameters-arb`, `program-native-parameters-arb`, `max-program-native-parameters-arb`, `program-attribs-arb`, `max-program-attribs-arb`, `program-native-attribs-arb`, `max-program-native-attribs-arb`, `program-address-registers-arb`, `max-program-address-registers-arb`, `program-native-address-registers-arb`, `max-program-native-address-registers-arb`, `max-program-local-parameters-arb`, `max-program-env-parameters-arb`, `program-under-native-limits-arb`, `transpose-current-matrix-arb`, `matrix0-arb`, `matrix1-arb`, `matrix2-arb`, `matrix3-arb`, `matrix4-arb`, `matrix5-arb`, `matrix6-arb`, `matrix7-arb`, `matrix8-arb`, `matrix9-arb`, `matrix10-arb`, `matrix11-arb`, `matrix12-arb`, `matrix13-arb`, `matrix14-arb`, `matrix15-arb`, `matrix16-arb`, `matrix17-arb`, `matrix18-arb`, `matrix19-arb`, `matrix20-arb`, `matrix21-arb`, `matrix22-arb`, `matrix23-arb`,

matrix24-arb, matrix25-arb, matrix26-arb, matrix27-arb, matrix28-arb, matrix29-arb, matrix30-arb, matrix31-arb.

`version-2-1` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

current-raster-secondary-color, pixel-pack-buffer, pixel-unpack-buffer, pixel-pack-buffer-binding, pixel-unpack-buffer-binding, srgb, srgb8, srgb-alpha, srgb8-alpha8, sluminance-alpha, sluminance8-alpha8, sluminance, sluminance8, compressed-srgb, compressed-srgb-alpha, compressed-sluminance, compressed-sluminance-alpha.

`sgix-icc-texture` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

smooth-point-size-range, smooth-point-size-granularity, smooth-line-width-range, smooth-line-width-granularity, aliased-point-size-range, aliased-line-width-range.

`rend-screen-coordinates` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

screen-coordinates-rend, inverted-screen-w-rend.

`arb-multitexture` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

texture0-arb, texture1-arb, texture2-arb, texture3-arb, texture4-arb, texture5-arb, texture6-arb, texture7-arb, texture8-arb, texture9-arb, texture10-arb, texture11-arb, texture12-arb, texture13-arb, texture14-arb, texture15-arb, texture16-arb, texture17-arb, texture18-arb, texture19-arb, texture20-arb, texture21-arb, texture22-arb, texture23-arb, texture24-arb, texture25-arb, texture26-arb, texture27-arb, texture28-arb, texture29-arb, texture30-arb, texture31-arb, active-texture-arb, client-active-texture-arb, max-texture-units-arb.

`oes-texture-env-crossbar` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

texture0, texture1, texture2, texture3, texture4, texture5, texture6, texture7, texture8, texture9, texture10, texture11, texture12, texture13, texture14, texture15, texture16, texture17, texture18, texture19, texture20,

texture21, texture22, texture23, texture24, texture25, texture26, texture27, texture28, texture29, texture30, texture31.

`arb-transpose-matrix` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

transpose-modelview-matrix-arb, transpose-projection-matrix-arb, transpose-texture-matrix-arb, transpose-color-matrix-arb.

`arb-texture-env-combine` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

subtract-arb.

`arb-texture-compression` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

compressed-alpha-arb, compressed-luminance-arb, compressed-luminance-alpha-arb, compressed-intensity-arb, compressed-rgb-arb, compressed-rgba-arb, texture-compression-hint-arb, texture-compressed-image-size-arb, texture-compressed-arb, num-compressed-texture-formats-arb, compressed-texture-formats-arb.

`nv-fence` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

all-completed-nv, fence-status-nv, fence-condition-nv.

`version-3-1` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

texture-rectangle, texture-binding-rectangle, proxy-texture-rectangle, max-rectangle-texture-size, uniform-buffer, uniform-buffer-binding, uniform-buffer-start, uniform-buffer-size, max-vertex-uniform-blocks, max-geometry-uniform-blocks, max-fragment-uniform-blocks, max-combined-uniform-blocks, max-uniform-buffer-bindings, max-uniform-block-size, max-combined-vertex-uniform-components, max-combined-geometry-uniform-components, max-combined-fragment-uniform-components, uniform-buffer-offset-alignment, active-uniform-block-max-name-length, active-uniform-blocks, uniform-type, uniform-size, uniform-name-length, uniform-block-index, uniform-offset, uniform-array-stride, uniform-matrix-stride, uniform-is-row-major, uniform-block-binding,

`uniform-block-data-size`, `uniform-block-name-length`, `uniform-block-active-uniforms`, `uniform-block-active-uniform-indices`, `uniform-block-referenced-by-vertex-shader`, `uniform-block-referenced-by-geometry-shader`, `uniform-block-referenced-by-fragment-shader`, `invalid-index`, `sampler-2d-rect`, `sampler-2d-rect-shadow`, `texture-buffer`, `max-texture-buffer-size`, `texture-binding-buffer`, `texture-buffer-data-store-binding`, `sampler-buffer`, `int-sampler-2d-rect`, `int-sampler-buffer`, `unsigned-int-sampler-2d-rect`, `unsigned-int-sampler-buffer`, `copy-read-buffer`, `copy-write-buffer`, `red-snorm`, `rg-snorm`, `rgb-snorm`, `rgba-snorm`, `r8-snorm`, `rg8-snorm`, `rgb8-snorm`, `rgba8-snorm`, `r16-snorm`, `rg16-snorm`, `rgb16-snorm`, `rgba16-snorm`, `signed-normalized`, `primitive-restart`, `primitive-restart-index`.

`arb-texture-rectangle` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-rectangle-arb`, `texture-binding-rectangle-arb`, `proxy-texture-rectangle-arb`, `max-rectangle-texture-size-arb`.

`nv-texture-rectangle` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-rectangle-nv`, `texture-binding-rectangle-nv`, `proxy-texture-rectangle-nv`, `max-rectangle-texture-size-nv`.

`ext-packed-depth-stencil` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`depth-stencil-ext`, `unsigned-int-24-8-ext`, `depth24-stencil8-ext`, `texture-stencil-size-ext`.

`nv-packed-depth-stencil` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`depth-stencil-nv`, `unsigned-int-24-8-nv`.

`oes-packed-depth-stencil` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`depth-stencil-oes`, `unsigned-int-24-8-oes`, `depth24-stencil8-oes`.

`ext-texture-lod-bias` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`max-texture-lod-bias-ext`, `texture-filter-control-ext`, `texture-lod-bias-ext`.

`ext-texture-filter-anisotropic` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-max-anisotropy-ext`, `max-texture-max-anisotropy-ext`.

`ext-vertex-weighting` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`modelview1-stack-depth-ext`, `modelview-matrix1-ext`, `vertex-weighting-ext`, `modelview1-ext`, `current-vertex-weight-ext`, `vertex-weight-array-ext`, `vertex-weight-array-size-ext`, `vertex-weight-array-type-ext`, `vertex-weight-array-stride-ext`, `vertex-weight-array-pointer-ext`.

`nv-light-max-exponent` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`max-shininess-nv`, `max-spot-exponent-nv`.

`ext-stencil-wrap` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`incr-wrap-ext`, `decr-wrap-ext`.

`oes-stencil-wrap` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`incr-wrap-oes`, `decr-wrap-oes`.

`ext-texture-cube-map` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`normal-map-ext`, `reflection-map-ext`, `texture-cube-map-ext`, `texture-binding-cube-map-ext`, `texture-cube-map-positive-x-ext`, `texture-cube-map-negative-x-ext`, `texture-cube-map-positive-y-ext`, `texture-cube-map-

`negative-y-ext`, `texture-cube-map-positive-z-ext`, `texture-cube-map-negative-z-ext`, `proxy-texture-cube-map-ext`, `max-cube-map-texture-size-ext`.

`nv-texgen-reflection` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`normal-map`, `reflection-map`.

`arb-texture-cube-map` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`normal-map-arb`, `reflection-map-arb`, `texture-cube-map-arb`, `texture-binding-cube-map-arb`, `texture-cube-map-positive-x-arb`, `texture-cube-map-negative-x-arb`, `texture-cube-map-positive-y-arb`, `texture-cube-map-negative-y-arb`, `texture-cube-map-positive-z-arb`, `texture-cube-map-negative-z-arb`, `proxy-texture-cube-map-arb`, `max-cube-map-texture-size-arb`.

`nv-vertex-array-range` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`vertex-array-range-nv`, `vertex-array-range-length-nv`, `vertex-array-range-valid-nv`, `max-vertex-array-range-element-nv`, `vertex-array-range-pointer-nv`.

`apple-vertex-array-range` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`vertex-array-range-apple`, `vertex-array-range-length-apple`, `vertex-array-storage-hint-apple`, `vertex-array-range-pointer-apple`, `storage-client-apple`, `storage-cached-apple`, `storage-shared-apple`.

`nv-register-combiners` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`register-combiners-nv`, `variable-a-nv`, `variable-b-nv`, `variable-c-nv`, `variable-d-nv`, `variable-e-nv`, `variable-f-nv`, `variable-g-nv`, `constant-color0-nv`, `constant-color1-nv`, `primary-color-nv`, `secondary-color-nv`, `spare0-nv`, `spare1-nv`, `discard-nv`, `e-times-f-nv`, `spare0-plus-secondary-color-nv`, `vertex-array-range-without-flush-nv`, `multisample-filter-hint-nv`, `unsigned-identity-nv`, `unsigned-invert-nv`,

`expand-normal-nv`, `expand-negate-nv`, `half-bias-normal-nv`, `half-bias-negate-nv`, `signed-identity-nv`, `unsigned-negate-nv`, `scale-by-two-nv`, `scale-by-four-nv`, `scale-by-one-half-nv`, `bias-by-negative-one-half-nv`, `combiner-input-nv`, `combiner-mapping-nv`, `combiner-component-usage-nv`, `combiner-ab-dot-product-nv`, `combiner-cd-dot-product-nv`, `combiner-mux-sum-nv`, `combiner-scale-nv`, `combiner-bias-nv`, `combiner-ab-output-nv`, `combiner-cd-output-nv`, `combiner-sum-output-nv`, `max-general-combiners-nv`, `num-general-combiners-nv`, `color-sum-clamp-nv`, `combiner0-nv`, `combiner1-nv`, `combiner2-nv`, `combiner3-nv`, `combiner4-nv`, `combiner5-nv`, `combiner6-nv`, `combiner7-nv`.

`nv-register-combiners-2` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`per-stage-constants-nv`.

`nv-primitive-restart` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`primitive-restart-nv`, `primitive-restart-index-nv`.

`nv-fog-distance` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`fog-gen-mode-nv`, `eye-radial-nv`, `eye-plane-absolute-nv`.

`nv-texgen-emboss` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`emboss-light-nv`, `emboss-constant-nv`, `emboss-map-nv`.

`ingr-color-clamp` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`red-min-clamp-ingr`, `green-min-clamp-ingr`, `blue-min-clamp-ingr`, `alpha-min-clamp-ingr`, `red-max-clamp-ingr`, `green-max-clamp-ingr`, `blue-max-clamp-ingr`, `alpha-max-clamp-ingr`.

`ingr-interlace-read` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`interlace-read-ingr`.

`ext-texture-env-combine` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`combine-ext`, `combine-rgb-ext`, `combine-alpha-ext`, `rgb-scale-ext`, `add-signed-ext`, `interpolate-ext`, `constant-ext`, `primary-color-ext`, `previous-ext`, `source0-rgb-ext`, `source1-rgb-ext`, `source2-rgb-ext`, `source0-alpha-ext`, `source1-alpha-ext`, `source2-alpha-ext`, `operand0-rgb-ext`, `operand1-rgb-ext`, `operand2-rgb-ext`, `operand0-alpha-ext`, `operand1-alpha-ext`, `operand2-alpha-ext`.

`nv-texture-env-combine-4` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`combine4-nv`, `source3-rgb-nv`, `source3-alpha-nv`, `operand3-rgb-nv`, `operand3-alpha-nv`.

`sgix-subsample` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`pack-subsample-rate-sgix`, `unpack-subsample-rate-sgix`, `pixel-subsample-4444-sgix`, `pixel-subsample-2424-sgix`, `pixel-subsample-4242-sgix`.

`ext-texture-perturb-normal` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`perturb-ext`, `texture-normal-ext`.

`apple-specular-vector` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`light-model-specular-vector-apple`.

`apple-transform-hint` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`transform-hint-apple`.

`apple-client-storage` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`unpack-client-storage-apple`.

`apple-object-purgeable` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`buffer-object-apple`, `released-apple`, `volatile-apple`, `retained-apple`, `undefined-apple`, `purgeable-apple`.

`arb-vertex-array-object` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`vertex-array-binding`.

`apple-vertex-array-object` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`vertex-array-binding-apple`.

`apple-texture-range` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-range-length-apple`, `texture-range-pointer-apple`, `texture-storage-hint-apple`, `storage-private-apple`, `storage-cached-apple`, `storage-shared-apple`.

`apple-ycbcr-422` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`ycbcr-422-apple`, `unsigned-short-8-8-apple`, `unsigned-short-8-8-rev-apple`.

`mesa-ycbcr-texture` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`unsigned-short-8-8-mesa`, `unsigned-short-8-8-rev-mesa`, `ycbcr-mesa`.

`sun-slice-accum` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`slice-accum-sun`.

`sun-mesh-array` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`quad-mesh-sun`, `triangle-mesh-sun`.

`nv-vertex-program` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`vertex-program-nv`, `vertex-state-program-nv`, `attrib-array-size-nv`, `attrib-array-stride-nv`, `attrib-array-type-nv`, `current-attrib-nv`, `program-length-nv`, `program-string-nv`, `modelview-projection-nv`, `identity-nv`, `inverse-nv`, `transpose-nv`, `inverse-transpose-nv`, `max-track-matrix-stack-depth-nv`, `max-track-matrices-nv`, `matrix0-nv`, `matrix1-nv`, `matrix2-nv`, `matrix3-nv`, `matrix4-nv`, `matrix5-nv`, `matrix6-nv`, `matrix7-nv`, `current-matrix-stack-depth-nv`, `current-matrix-nv`, `vertex-program-point-size-nv`, `vertex-program-two-side-nv`, `program-parameter-nv`, `attrib-array-pointer-nv`, `program-target-nv`, `program-resident-nv`, `track-matrix-nv`, `track-matrix-transform-nv`, `vertex-program-binding-nv`, `program-error-position-nv`, `vertex-attrib-array0-nv`, `vertex-attrib-array1-nv`, `vertex-attrib-array2-nv`, `vertex-attrib-array3-nv`, `vertex-attrib-array4-nv`, `vertex-attrib-array5-nv`, `vertex-attrib-array6-nv`, `vertex-attrib-array7-nv`, `vertex-attrib-array8-nv`, `vertex-attrib-array9-nv`, `vertex-attrib-array10-nv`, `vertex-attrib-array11-nv`, `vertex-attrib-array12-nv`, `vertex-attrib-array13-nv`, `vertex-attrib-array14-nv`, `vertex-attrib-array15-nv`, `map1-vertex-attrib0-4-nv`, `map1-vertex-attrib1-4-nv`, `map1-vertex-attrib2-4-nv`, `map1-vertex-attrib3-4-nv`, `map1-vertex-attrib4-4-nv`, `map1-vertex-attrib5-4-nv`, `map1-vertex-attrib6-4-nv`, `map1-vertex-attrib7-4-nv`, `map1-vertex-attrib8-4-nv`, `map1-vertex-attrib9-4-nv`, `map1-vertex-attrib10-4-nv`, `map1-vertex-attrib11-4-nv`, `map1-vertex-attrib12-4-nv`, `map1-vertex-attrib13-4-nv`, `map1-vertex-attrib14-4-nv`, `map1-vertex-attrib15-4-nv`, `map2-vertex-attrib0-4-nv`, `map2-vertex-attrib1-4-nv`, `map2-vertex-attrib2-4-nv`, `map2-vertex-attrib3-4-nv`, `map2-vertex-attrib4-4-nv`, `map2-vertex-attrib5-4-nv`, `map2-vertex-attrib6-4-nv`, `map2-vertex-attrib7-4-nv`, `map2-vertex-attrib8-4-nv`, `map2-vertex-attrib9-4-nv`, `map2-vertex-attrib10-4-nv`, `map2-vertex-attrib11-4-nv`, `map2-vertex-attrib12-4-nv`, `map2-vertex-attrib13-4-nv`, `map2-vertex-attrib14-4-nv`, `map2-vertex-attrib15-4-nv`.

`arb-depth-clamp` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`depth-clamp`.

`nv-depth-clamp` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`depth-clamp-nv`.

`arb-fragment-program` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`vertex-program-arb`, `vertex-attrib-array-enabled-arb`, `vertex-attrib-array-size-arb`, `vertex-attrib-array-stride-arb`, `vertex-attrib-array-type-arb`, `current-vertex-attrib-arb`, `program-length-arb`, `program-string-arb`, `max-program-matrix-stack-depth-arb`, `max-program-matrices-arb`, `current-matrix-stack-depth-arb`, `current-matrix-arb`, `vertex-program-point-size-arb`, `vertex-program-two-side-arb`, `vertex-attrib-array-pointer-arb`, `program-error-position-arb`, `program-binding-arb`, `fragment-program-arb`, `program-alu-instructions-arb`, `program-tex-instructions-arb`, `program-tex-indirections-arb`, `program-native-alu-instructions-arb`, `program-native-tex-instructions-arb`, `program-native-tex-indirections-arb`, `max-program-alu-instructions-arb`, `max-program-tex-instructions-arb`, `max-program-tex-indirections-arb`, `max-program-native-alu-instructions-arb`, `max-program-native-tex-instructions-arb`, `max-program-native-tex-indirections-arb`, `max-texture-coords-arb`, `max-texture-image-units-arb`, `program-error-string-arb`, `program-format-ascii-arb`, `program-format-arb`, `program-instructions-arb`, `max-program-instructions-arb`, `program-native-instructions-arb`, `max-program-native-instructions-arb`, `program-temporaries-arb`, `max-program-temporaries-arb`, `program-native-temporaries-arb`, `max-program-native-temporaries-arb`, `program-parameters-arb`, `max-program-parameters-arb`, `program-native-parameters-arb`, `max-program-native-parameters-arb`, `program-attribs-arb`, `max-program-attribs-arb`, `program-native-attribs-arb`, `max-program-native-attribs-arb`, `program-address-registers-arb`, `max-program-address-registers-arb`, `program-native-address-registers-arb`, `max-program-native-address-registers-arb`, `max-program-local-parameters-arb`, `max-program-env-parameters-arb`, `program-under-native-limits-arb`, `transpose-current-matrix-arb`, `matrix0-arb`, `matrix1-arb`, `matrix2-arb`, `matrix3-arb`, `matrix4-arb`, `matrix5-arb`, `matrix6-arb`, `matrix7-arb`, `matrix8-arb`, `matrix9-arb`, `matrix10-arb`, `matrix11-arb`, `matrix12-arb`, `matrix13-arb`, `matrix14-arb`, `matrix15-arb`, `matrix16-arb`, `matrix17-arb`, `matrix18-arb`, `matrix19-arb`, `matrix20-arb`, `matrix21-arb`, `matrix22-arb`, `matrix23-arb`, `matrix24-arb`, `matrix25-arb`, `matrix26-arb`, `matrix27-arb`, `matrix28-arb`, `matrix29-arb`, `matrix30-arb`, `matrix31-arb`.

`arb-vertex-blend` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`max-vertex-units-arb`, `active-vertex-units-arb`, `weight-sum-unity-arb`, `vertex-blend-arb`, `current-weight-arb`, `weight-array-type-arb`, `weight-array-stride-arb`, `weight-array-size-arb`, `weight-array-pointer-arb`, `weight-array-arb`, `modelview0-arb`, `modelview1-arb`, `modelview2-arb`, `modelview3-arb`, `modelview4-arb`, `modelview5-arb`, `modelview6-arb`,

`modelview7-arb`, `modelview8-arb`, `modelview9-arb`, `modelview10-arb`, `modelview11-arb`, `modelview12-arb`, `modelview13-arb`, `modelview14-arb`, `modelview15-arb`, `modelview16-arb`, `modelview17-arb`, `modelview18-arb`, `modelview19-arb`, `modelview20-arb`, `modelview21-arb`, `modelview22-arb`, `modelview23-arb`, `modelview24-arb`, `modelview25-arb`, `modelview26-arb`, `modelview27-arb`, `modelview28-arb`, `modelview29-arb`, `modelview30-arb`, `modelview31-arb`.

`oes-matrix-palette` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`max-vertex-units-oes`, `weight-array-oes`, `weight-array-type-oes`, `weight-array-stride-oes`, `weight-array-size-oes`, `weight-array-pointer-oes`, `matrix-palette-oes`, `max-palette-matrices-oes`, `current-palette-matrix-oes`, `matrix-index-array-oes`, `matrix-index-array-size-oes`, `matrix-index-array-type-oes`, `matrix-index-array-stride-oes`, `matrix-index-array-pointer-oes`, `weight-array-buffer-binding-oes`, `matrix-index-array-buffer-binding-oes`.

`arb-texture-env-dot-3` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`dot3-rgb-arb`, `dot3-rgba-arb`.

`img-texture-env-enhanced-fixed-function` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`dot3-rgba-img`, `modulate-color-img`, `recip-add-signed-alpha-img`, `texture-alpha-modulate-img`, `factor-alpha-modulate-img`, `fragment-alpha-modulate-img`, `add-blend-img`.

`3dfx-texture-compression-fxt1` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`compressed-rgb-fxt1-3dfx`, `compressed-rgba-fxt1-3dfx`.

`nv-evaluators` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`eval-2d-nv`, `eval-triangular-2d-nv`, `map-tessellation-nv`, `map-attrib-u-order-nv`, `map-attrib-v-order-nv`, `eval-fractional-tessellation-nv`, `eval-vertex-atrrib0-nv`, `eval-vertex-atrrib1-nv`, `eval-vertex-atrrib2-nv`,

`eval-vertex-atrrib3-nv`, `eval-vertex-atrrib4-nv`, `eval-vertex-atrrib5-nv`, `eval-vertex-atrrib6-nv`, `eval-vertex-atrrib7-nv`, `eval-vertex-atrrib8-nv`, `eval-vertex-atrrib9-nv`, `eval-vertex-atrrib10-nv`, `eval-vertex-atrrib11-nv`, `eval-vertex-atrrib12-nv`, `eval-vertex-atrrib13-nv`, `eval-vertex-atrrib14-nv`, `eval-vertex-atrrib15-nv`, `max-map-tessellation-nv`, `max-rational-eval-order-nv`.

`nv-tessellation-program-5` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`max-program-patch-attribs-nv`, `tess-control-program-nv`, `tess-evaluation-program-nv`, `tess-control-program-parameter-buffer-nv`, `tess-evaluation-program-parameter-buffer-nv`.

`nv-texture-shader` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`offset-texture-rectangle-nv`, `offset-texture-rectangle-scale-nv`, `dot-product-texture-rectangle-nv`, `rgba-unsigned-dot-product-mapping-nv`, `unsigned-int-s8-s8-8-8-nv`, `unsigned-int-8-8-s8-s8-rev-nv`, `dsdt-mag-intensity-nv`, `shader-consistent-nv`, `texture-shader-nv`, `shader-operation-nv`, `cull-modes-nv`, `offset-texture-matrix-nv`, `offset-texture-scale-nv`, `offset-texture-bias-nv`, `offset-texture-2d-matrix-nv`, `offset-texture-2d-scale-nv`, `offset-texture-2d-bias-nv`, `previous-texture-input-nv`, `const-eye-nv`, `pass-through-nv`, `cull-fragment-nv`, `offset-texture-2d-nv`, `dependent-ar-texture-2d-nv`, `dependent-gb-texture-2d-nv`, `dot-product-nv`, `dot-product-depth-replace-nv`, `dot-product-texture-2d-nv`, `dot-product-texture-cube-map-nv`, `dot-product-diffuse-cube-map-nv`, `dot-product-reflect-cube-map-nv`, `dot-product-const-eye-reflect-cube-map-nv`, `hilo-nv`, `dsdt-nv`, `dsdt-mag-nv`, `dsdt-mag-vib-nv`, `hilo16-nv`, `signed-hilo-nv`, `signed-hilo16-nv`, `signed-rgba-nv`, `signed-rgba8-nv`, `signed-rgb-nv`, `signed-rgb8-nv`, `signed-luminance-nv`, `signed-luminance8-nv`, `signed-luminance-alpha-nv`, `signed-luminance8-alpha8-nv`, `signed-alpha-nv`, `signed-alpha8-nv`, `signed-intensity-nv`, `signed-intensity8-nv`, `dsdt8-nv`, `dsdt8-mag8-nv`, `dsdt8-mag8-intensity8-nv`, `signed-rgb-unsigned-alpha-nv`, `signed-rgb8-unsigned-alpha8-nv`, `hi-scale-nv`, `lo-scale-nv`, `ds-scale-nv`, `dt-scale-nv`, `magnitude-scale-nv`, `vibrance-scale-nv`, `hi-bias-nv`, `lo-bias-nv`, `ds-bias-nv`, `dt-bias-nv`, `magnitude-bias-nv`, `vibrance-bias-nv`, `texture-border-values-nv`, `texture-hi-size-nv`, `texture-lo-size-nv`, `texture-ds-size-nv`, `texture-dt-size-nv`, `texture-mag-size-nv`.

`nv-vdpau-interop` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`surface-state-nv`, `surface-registered-nv`, `surface-mapped-nv`, `write-discard-nv`.

`nv-texture-shader-2` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`dot-product-texture-3d-nv`.

`ext-texture-env-dot-3` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`dot3-rgb-ext`, `dot3-rgba-ext`.

`amd-program-binary-z400` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`z400-binary-amd`.

`oes-get-program-binary` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`program-binary-length-oes`, `num-program-binary-formats-oes`, `program-binary-formats-oes`.

`ati-texture-mirror-once` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`mirror-clamp-ati`, `mirror-clamp-to-edge-ati`.

`ext-texture-mirror-clamp` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`mirror-clamp-ext`, `mirror-clamp-to-edge-ext`, `mirror-clamp-to-border-ext`.

`ati-texture-env-combine-3` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`modulate-add-ati`, `modulate-signed-add-ati`, `modulate-subtract-ati`.

`amd-stencil-operation-extended` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`set-amd`, `replace-value-amd`, `stencil-op-value-amd`, `stencil-back-op-value-amd`.

`mesa-packed-depth-stencil` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`depth-stencil-mesa`, `unsigned-int-24-8-mesa`, `unsigned-int-8-24-rev-mesa`, `unsigned-short-15-1-mesa`, `unsigned-short-1-15-rev-mesa`.

`mesa-trace` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`trace-all-bits-mesa`, `trace-operations-bit-mesa`, `trace-primitives-bit-mesa`, `trace-arrays-bit-mesa`, `trace-textures-bit-mesa`, `trace-pixels-bit-mesa`, `trace-errors-bit-mesa`, `trace-mask-mesa`, `trace-name-mesa`.

`mesa-pack-invert` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`pack-invert-mesa`.

`mesax-texture-stack` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-1d-stack-mesax`, `texture-2d-stack-mesax`, `proxy-texture-1d-stack-mesax`, `proxy-texture-2d-stack-mesax`, `texture-1d-stack-binding-mesax`, `texture-2d-stack-binding-mesax`.

`mesa-shader-debug` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`debug-object-mesa`, `debug-print-mesa`, `debug-assert-mesa`.

`ati-vertex-array-object` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`static-ati`, `dynamic-ati`, `preserve-ati`, `discard-ati`, `object-buffer-size-ati`, `object-buffer-usage-ati`, `array-object-buffer-ati`, `array-object-offset-ati`.

`arb-vertex-buffer-object` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`buffer-size-arb`, `buffer-usage-arb`, `array-buffer-arb`, `element-array-buffer-arb`, `array-buffer-binding-arb`, `element-array-buffer-binding-arb`, `vertex-array-buffer-binding-arb`, `normal-array-buffer-binding-arb`, `color-array-buffer-binding-arb`, `index-array-buffer-binding-arb`, `texture-coord-array-buffer-binding-arb`, `edge-flag-array-buffer-binding-arb`, `secondary-color-array-buffer-binding-arb`, `fog-coordinate-array-buffer-binding-arb`, `weight-array-buffer-binding-arb`, `vertex-attrib-array-buffer-binding-arb`, `read-only-arb`, `write-only-arb`, `read-write-arb`, `buffer-access-arb`, `buffer-mapped-arb`, `buffer-map-pointer-arb`, `stream-draw-arb`, `stream-read-arb`, `stream-copy-arb`, `static-draw-arb`, `static-read-arb`, `static-copy-arb`, `dynamic-draw-arb`, `dynamic-read-arb`, `dynamic-copy-arb`.

`ati-element-array` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`element-array-ati`, `element-array-type-ati`, `element-array-pointer-ati`.

`ati-vertex-streams` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`max-vertex-streams-ati`, `vertex-stream0-ati`, `vertex-stream1-ati`, `vertex-stream2-ati`, `vertex-stream3-ati`, `vertex-stream4-ati`, `vertex-stream5-ati`, `vertex-stream6-ati`, `vertex-stream7-ati`, `vertex-source-ati`.

`ati-envmap-bumpmap` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`bump-rot-matrix-ati`, `bump-rot-matrix-size-ati`, `bump-num-tex-units-ati`, `bump-tex-units-ati`, `dudv-ati`, `du8dv8-ati`, `bump-envmap-ati`, `bump-target-ati`.

`ext-vertex-shader` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`vertex-shader-ext`, `vertex-shader-binding-ext`, `op-index-ext`, `op-negate-ext`, `op-dot3-ext`, `op-dot4-ext`, `op-mul-ext`, `op-add-ext`, `op-madd-ext`,

`op-frac-ext`, `op-max-ext`, `op-min-ext`, `op-set-ge-ext`, `op-set-lt-ext`, `op-clamp-ext`, `op-floor-ext`, `op-round-ext`, `op-exp-base-2-ext`, `op-log-base-2-ext`, `op-power-ext`, `op-recip-ext`, `op-recip-sqrt-ext`, `op-sub-ext`, `op-cross-product-ext`, `op-multiply-matrix-ext`, `op-mov-ext`, `output-vertex-ext`, `output-color0-ext`, `output-color1-ext`, `output-texture-coord0-ext`, `output-texture-coord1-ext`, `output-texture-coord2-ext`, `output-texture-coord3-ext`, `output-texture-coord4-ext`, `output-texture-coord5-ext`, `output-texture-coord6-ext`, `output-texture-coord7-ext`, `output-texture-coord8-ext`, `output-texture-coord9-ext`, `output-texture-coord10-ext`, `output-texture-coord11-ext`, `output-texture-coord12-ext`, `output-texture-coord13-ext`, `output-texture-coord14-ext`, `output-texture-coord15-ext`, `output-texture-coord16-ext`, `output-texture-coord17-ext`, `output-texture-coord18-ext`, `output-texture-coord19-ext`, `output-texture-coord20-ext`, `output-texture-coord21-ext`, `output-texture-coord22-ext`, `output-texture-coord23-ext`, `output-texture-coord24-ext`, `output-texture-coord25-ext`, `output-texture-coord26-ext`, `output-texture-coord27-ext`, `output-texture-coord28-ext`, `output-texture-coord29-ext`, `output-texture-coord30-ext`, `output-texture-coord31-ext`, `output-fog-ext`, `scalar-ext`, `vector-ext`, `matrix-ext`, `variant-ext`, `invariant-ext`, `local-constant-ext`, `local-ext`, `max-vertex-shader-instructions-ext`, `max-vertex-shader-variants-ext`, `max-vertex-shader-invariants-ext`, `max-vertex-shader-local-constants-ext`, `max-vertex-shader-locals-ext`, `max-optimized-vertex-shader-instructions-ext`, `max-optimized-vertex-shader-variants-ext`, `max-optimized-vertex-shader-local-constants-ext`, `max-optimized-vertex-shader-invariants-ext`, `max-optimized-vertex-shader-locals-ext`, `vertex-shader-instructions-ext`, `vertex-shader-variants-ext`, `vertex-shader-invariants-ext`, `vertex-shader-local-constants-ext`, `vertex-shader-locals-ext`, `vertex-shader-optimized-ext`, `x-ext`, `y-ext`, `z-ext`, `w-ext`, `negative-x-ext`, `negative-y-ext`, `negative-z-ext`, `negative-w-ext`, `zero-ext`, `one-ext`, `negative-one-ext`, `normalized-range-ext`, `full-range-ext`, `current-vertex-ext`, `mvp-matrix-ext`, `variant-value-ext`, `variant-datatype-ext`, `variant-array-stride-ext`, `variant-array-type-ext`, `variant-array-ext`, `variant-array-pointer-ext`, `invariant-value-ext`, `invariant-datatype-ext`, `local-constant-value-ext`, `local-constant-datatype-ext`.

`amd-compressed-atc-texture` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`atc-rgba-interpolated-alpha-amd`, `atc-rgb-amd`, `atc-rgba-explicit-alpha-amd`.

`ati-pn-triangles` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`pn-triangles-ati`, `max-pn-triangles-tesselation-level-ati`, `pn-triangles-point-mode-ati`, `pn-triangles-normal-mode-ati`, `pn-triangles-tesselation-level-ati`, `pn-triangles-point-mode-linear-ati`, `pn-triangles-point-mode-cubic-ati`, `pn-triangles-normal-mode-linear-ati`, `pn-triangles-normal-mode-quadratic-ati`.

`amd-compressed-3dc-texture` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`3dc-x-amd`, `3dc-xy-amd`.

`ati-meminfo` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`vbo-free-memory-ati`, `texture-free-memory-ati`, `renderbuffer-free-memory-ati`.

`ati-separate-stencil` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`stencil-back-func-ati`, `stencil-back-pass-depth-fail-ati`, `stencil-back-pass-depth-pass-ati`.

`arb-texture-float` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`rgba32f-arb`, `rgb32f-arb`, `alpha32f-arb`, `intensity32f-arb`, `luminance32f-arb`, `luminance-alpha32f-arb`, `rgba16f-arb`, `rgb16f-arb`, `alpha16f-arb`, `intensity16f-arb`, `luminance16f-arb`, `luminance-alpha16f-arb`, `texture-red-type-arb`, `texture-green-type-arb`, `texture-blue-type-arb`, `texture-alpha-type-arb`, `texture-luminance-type-arb`, `texture-intensity-type-arb`, `texture-depth-type-arb`, `unsigned-normalized-arb`.

`ati-texture-float` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`rgba-float32-ati`, `rgb-float32-ati`, `alpha-float32-ati`, `intensity-float32-ati`, `luminance-float32-ati`, `luminance-alpha-float32-ati`, `rgba-float16-ati`, `rgb-float16-ati`, `alpha-float16-ati`, `intensity-float16-ati`, `luminance-float16-ati`, `luminance-alpha-float16-ati`.

`arb-color-buffer-float` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`rgba-float-mode-arb`, `clamp-vertex-color-arb`, `clamp-fragment-color-arb`, `clamp-read-color-arb`, `fixed-only-arb`.

`ati-pixel-format-float` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`type-rgba-float-ati`, `color-clear-unclamped-value-ati`.

`qcom-writeonly-rendering` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`writeonly-rendering-qcom`.

`arb-draw-buffers` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`max-draw-buffers-arb`, `draw-buffer0-arb`, `draw-buffer1-arb`, `draw-buffer2-arb`, `draw-buffer3-arb`, `draw-buffer4-arb`, `draw-buffer5-arb`, `draw-buffer6-arb`, `draw-buffer7-arb`, `draw-buffer8-arb`, `draw-buffer9-arb`, `draw-buffer10-arb`, `draw-buffer11-arb`, `draw-buffer12-arb`, `draw-buffer13-arb`, `draw-buffer14-arb`, `draw-buffer15-arb`.

`ati-draw-buffers` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`max-draw-buffers-ati`, `draw-buffer0-ati`, `draw-buffer1-ati`, `draw-buffer2-ati`, `draw-buffer3-ati`, `draw-buffer4-ati`, `draw-buffer5-ati`, `draw-buffer6-ati`, `draw-buffer7-ati`, `draw-buffer8-ati`, `draw-buffer9-ati`, `draw-buffer10-ati`, `draw-buffer11-ati`, `draw-buffer12-ati`, `draw-buffer13-ati`, `draw-buffer14-ati`, `draw-buffer15-ati`.

`nv-draw-buffers` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`max-draw-buffers-nv`, `draw-buffer0-nv`, `draw-buffer1-nv`, `draw-buffer2-nv`, `draw-buffer3-nv`, `draw-buffer4-nv`, `draw-buffer5-nv`, `draw-buffer6-nv`, `draw-buffer7-nv`, `draw-buffer8-nv`, `draw-buffer9-nv`, `draw-buffer10-nv`, `draw-buffer11-nv`, `draw-buffer12-nv`, `draw-buffer13-nv`, `draw-buffer14-nv`, `draw-buffer15-nv`, `color-attachment0-nv`, `color-attachment1-nv`,

`color-attachment2-nv`, `color-attachment3-nv`, `color-attachment4-nv`, `color-attachment5-nv`, `color-attachment6-nv`, `color-attachment7-nv`, `color-attachment8-nv`, `color-attachment9-nv`, `color-attachment10-nv`, `color-attachment11-nv`, `color-attachment12-nv`, `color-attachment13-nv`, `color-attachment14-nv`, `color-attachment15-nv`.

`amd-sample-positions` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`subsample-distance-amd`.

`arb-matrix-palette` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`matrix-palette-arb`, `max-matrix-palette-stack-depth-arb`, `max-palette-matrices-arb`, `current-palette-matrix-arb`, `matrix-index-array-arb`, `current-matrix-index-arb`, `matrix-index-array-size-arb`, `matrix-index-array-type-arb`, `matrix-index-array-stride-arb`, `matrix-index-array-pointer-arb`.

`arb-shadow` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-compare-mode-arb`, `texture-compare-func-arb`, `compare-r-to-texture-arb`.

`ext-shadow-samplers` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-compare-mode-ext`, `texture-compare-func-ext`, `compare-ref-to-texture-ext`, `sampler-2d-shadow-ext`.

`ext-texture-array` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`compare-ref-depth-to-texture-ext`, `max-array-texture-layers-ext`, `texture-1d-array-ext`, `proxy-texture-1d-array-ext`, `texture-2d-array-ext`, `proxy-texture-2d-array-ext`, `texture-binding-1d-array-ext`, `texture-binding-2d-array-ext`.

`arb-seamless-cube-map` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-cube-map-seamless`.

`nv-texture-shader-3` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`offset-projective-texture-2d-nv`, `offset-projective-texture-2d-scale-nv`, `offset-projective-texture-rectangle-nv`, `offset-projective-texture-rectangle-scale-nv`, `offset-hilo-texture-2d-nv`, `offset-hilo-texture-rectangle-nv`, `offset-hilo-projective-texture-2d-nv`, `offset-hilo-projective-texture-rectangle-nv`, `dependent-hilo-texture-2d-nv`, `dependent-rgb-texture-3d-nv`, `dependent-rgb-texture-cube-map-nv`, `dot-product-pass-through-nv`, `dot-product-texture-1d-nv`, `dot-product-affine-depth-replace-nv`, `hilo8-nv`, `signed-hilo8-nv`, `force-blue-to-one-nv`.

`arb-point-sprite` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`point-sprite-arb`, `coord-replace-arb`.

`nv-point-sprite` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`point-sprite-nv`, `coord-replace-nv`, `point-sprite-r-mode-nv`.

`oes-point-sprite` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`point-sprite-arb`, `coord-replace-arb`.

`arb-occlusion-query` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`query-counter-bits-arb`, `current-query-arb`, `query-result-arb`, `query-result-available-arb`, `samples-passed-arb`.

`nv-occlusion-query` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`pixel-counter-bits-nv`, `current-occlusion-query-id-nv`, `pixel-count-nv`, `pixel-count-available-nv`.

`ext-occlusion-query-boolean` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`current-query-ext`, `query-result-ext`, `query-result-available-ext`, `any-samples-passed-ext`, `any-samples-passed-conservative-ext`.

`nv-fragment-program` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`max-fragment-program-local-parameters-nv`, `fragment-program-nv`, `max-texture-coords-nv`, `max-texture-image-units-nv`, `fragment-program-binding-nv`, `program-error-string-nv`.

`nv-copy-depth-to-color` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`depth-stencil-to-rgba-nv`, `depth-stencil-to-bgra-nv`.

`nv-pixel-data-range` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`write-pixel-data-range-nv`, `read-pixel-data-range-nv`, `write-pixel-data-range-length-nv`, `read-pixel-data-range-length-nv`, `write-pixel-data-range-pointer-nv`, `read-pixel-data-range-pointer-nv`.

`arb-gpu-shader-5` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`geometry-shader-invocations`, `max-geometry-shader-invocations`, `min-fragment-interpolation-offset`, `max-fragment-interpolation-offset`, `fragment-interpolation-offset-bits`.

`nv-float-buffer` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`float-r-nv`, `float-rg-nv`, `float-rgb-nv`, `float-rgba-nv`, `float-r16-nv`, `float-r32-nv`, `float-rg16-nv`, `float-rg32-nv`, `float-rgb16-nv`, `float-rgb32-nv`, `float-rgba16-nv`, `float-rgba32-nv`, `texture-float-components-nv`, `float-clear-color-value-nv`, `float-rgba-mode-nv`.

`nv-texture-expand-normal` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-unsigned-remap-mode-nv`.

`ext-depth-bounds-test` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`depth-bounds-test-ext`, `depth-bounds-ext`.

`oes-mapbuffer` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`write-only-oes`, `buffer-access-oes`, `buffer-mapped-oes`, `buffer-map-pointer-oes`.

`nv-shader-buffer-store` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`read-write`, `write-only`, `shader-global-access-barrier-bit-nv`.

`arb-timer-query` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`time-elapsed`, `timestamp`.

`ext-timer-query` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`time-elapsed-ext`.

`arb-pixel-buffer-object` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`pixel-pack-buffer-arb`, `pixel-unpack-buffer-arb`, `pixel-pack-buffer-binding-arb`, `pixel-unpack-buffer-binding-arb`.

`ext-pixel-buffer-object` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`pixel-pack-buffer-ext`, `pixel-unpack-buffer-ext`, `pixel-pack-buffer-binding-ext`, `pixel-unpack-buffer-binding-ext`.

`nv-s-rgb-formats` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`etc1-srgb8-nv`, `srgb8-nv`, `sluminance-alpha-nv`, `sluminance8-alpha8-nv`, `sluminance-nv`, `sluminance8-nv`, `compressed-srgb-s3tc-dxt1-nv`, `compressed-srgb-alpha-s3tc-dxt1-nv`, `compressed-srgb-alpha-s3tc-dxt3-nv`, `compressed-srgb-alpha-s3tc-dxt5-nv`.

`ext-stencil-clear-tag` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`stencil-tag-bits-ext`, `stencil-clear-tag-value-ext`.

`nv-vertex-program-2-option` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`max-program-exec-instructions-nv`, `max-program-call-depth-nv`.

`nv-fragment-program-2` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`max-program-exec-instructions-nv`, `max-program-call-depth-nv`, `max-program-if-depth-nv`, `max-program-loop-depth-nv`, `max-program-loop-count-nv`.

`arb-blend-func-extended` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`src1-color`, `one-minus-src1-color`, `one-minus-src1-alpha`, `max-dual-source-draw-buffers`.

`nv-vertex-program-4` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`vertex-attrib-array-integer-nv`.

`version-3-3` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`vertex-attrib-array-divisor`.

`arb-instanced-arrays` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`vertex-attrib-array-divisor-arb`.

`angle-instanced-arrays` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`vertex-attrib-array-divisor-angle`.

`nv-instanced-arrays` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`vertex-attrib-array-divisor-nv`.

`nv-gpu-program-4` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`min-program-texel-offset-nv`, `max-program-texel-offset-nv`, `program-attrib-components-nv`, `program-result-components-nv`, `max-program-attrib-components-nv`, `max-program-result-components-nv`, `max-program-generic-attribs-nv`, `max-program-generic-results-nv`.

`ext-stencil-two-side` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`stencil-test-two-side-ext`, `active-stencil-face-ext`.

`arb-sampler-objects` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`sampler-binding`.

`ati-fragment-shader` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`fragment-shader-ati`, `reg-0-ati`, `reg-1-ati`, `reg-2-ati`, `reg-3-ati`, `reg-4-ati`, `reg-5-ati`, `reg-6-ati`, `reg-7-ati`, `reg-8-ati`, `reg-9-ati`, `reg-10-ati`, `reg-11-ati`, `reg-12-ati`, `reg-13-ati`, `reg-14-ati`, `reg-15-ati`, `reg-16-ati`,

reg-17-ati, reg-18-ati, reg-19-ati, reg-20-ati, reg-21-ati, reg-22-ati, reg-23-ati, reg-24-ati, reg-25-ati, reg-26-ati, reg-27-ati, reg-28-ati, reg-29-ati, reg-30-ati, reg-31-ati, con-0-ati, con-1-ati, con-2-ati, con-3-ati, con-4-ati, con-5-ati, con-6-ati, con-7-ati, con-8-ati, con-9-ati, con-10-ati, con-11-ati, con-12-ati, con-13-ati, con-14-ati, con-15-ati, con-16-ati, con-17-ati, con-18-ati, con-19-ati, con-20-ati, con-21-ati, con-22-ati, con-23-ati, con-24-ati, con-25-ati, con-26-ati, con-27-ati, con-28-ati, con-29-ati, con-30-ati, con-31-ati, mov-ati, add-ati, mul-ati, sub-ati, dot3-ati, dot4-ati, mad-ati, lerp-ati, cnd-ati, cnd0-ati, dot2-add-ati, secondary-interpolator-ati, num-fragment-registers-ati, num-fragment-constants-ati, num-passes-ati, num-instructions-per-pass-ati, num-instructions-total-ati, num-input-interpolator-components-ati, num-loopback-components-ati, color-alpha-pairing-ati, swizzle-str-ati, swizzle-stq-ati, swizzle-str-dr-ati, swizzle-stq-dq-ati, swizzle-strq-ati, swizzle-strq-dq-ati, red-bit-ati, green-bit-ati, blue-bit-ati, 2x-bit-ati, 4x-bit-ati, 8x-bit-ati, half-bit-ati, quarter-bit-ati, eighth-bit-ati, saturate-bit-ati, 2x-bit-ati, comp-bit-ati, negate-bit-ati, bias-bit-ati.

`oml-interlace` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`interlace-oml`, `interlace-read-oml`.

`oml-subsample` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`format-subsample-24-24-oml`, `format-subsample-244-244-oml`.

`oml-resample` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`pack-resample-oml`, `unpack-resample-oml`, `resample-replicate-oml`, `resample-zero-fill-oml`, `resample-average-oml`, `resample-decimate-oml`.

`oes-point-size-array` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`point-size-array-type-oes`, `point-size-array-stride-oes`, `point-size-array-pointer-oes`, `point-size-array-oes`, `point-size-array-buffer-binding-oes`.

`oes-matrix-get` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`modelview-matrix-float-as-int-bits-oes`, `projection-matrix-float-as-int-bits-oes`, `texture-matrix-float-as-int-bits-oes`.

`apple-vertex-program-evaluators` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`vertex-attrib-map1-apple`, `vertex-attrib-map2-apple`, `vertex-attrib-map1-size-apple`, `vertex-attrib-map1-coeff-apple`, `vertex-attrib-map1-order-apple`, `vertex-attrib-map1-domain-apple`, `vertex-attrib-map2-size-apple`, `vertex-attrib-map2-coeff-apple`, `vertex-attrib-map2-order-apple`, `vertex-attrib-map2-domain-apple`.

`apple-fence` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`draw-pixels-apple`, `fence-apple`.

`apple-element-array` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`element-array-apple`, `element-array-type-apple`, `element-array-pointer-apple`.

`arb-uniform-buffer-object` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`uniform-buffer`, `uniform-buffer-binding`, `uniform-buffer-start`, `uniform-buffer-size`, `max-vertex-uniform-blocks`, `max-geometry-uniform-blocks`, `max-fragment-uniform-blocks`, `max-combined-uniform-blocks`, `max-uniform-buffer-bindings`, `max-uniform-block-size`, `max-combined-vertex-uniform-components`, `max-combined-geometry-uniform-components`, `max-combined-fragment-uniform-components`, `uniform-buffer-offset-alignment`, `active-uniform-block-max-name-length`, `active-uniform-blocks`, `uniform-type`, `uniform-size`, `uniform-name-length`, `uniform-block-index`, `uniform-offset`, `uniform-array-stride`, `uniform-matrix-stride`, `uniform-is-row-major`, `uniform-block-binding`, `uniform-block-data-size`, `uniform-block-name-length`, `uniform-block-active-uniforms`, `uniform-block-active-uniform-indices`, `uniform-block-referenced-by-vertex-shader`, `uniform-block-referenced-by-geometry-shader`, `uniform-block-referenced-by-fragment-shader`, `invalid-index`.

`apple-flush-buffer-range` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`buffer-serialized-modify-apple`, `buffer-flushing-unmap-apple`.

`apple-aux-depth-stencil` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`aux-depth-stencil-apple`.

`apple-row-bytes` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`pack-row-bytes-apple`, `unpack-row-bytes-apple`.

`apple-rgb-422` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`rgb-422-apple`, `unsigned-short-8-8-apple`, `unsigned-short-8-8-rev-apple`.

`ext-texture-s-rgb-decode` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-srgb-decode-ext`, `decode-ext`, `skip-decode-ext`.

`ext-debug-label` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`program-pipeline-object-ext`, `program-object-ext`, `shader-object-ext`, `buffer-object-ext`, `query-object-ext`, `vertex-array-object-ext`.

`ext-shader-framebuffer-fetch` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`fragment-shader-discards-samples-ext`.

`apple-sync` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

sync-object-apple, max-server-wait-timeout-apple, object-type-apple, sync-condition-apple, sync-status-apple, sync-flags-apple, sync-fence-apple, sync-gpu-commands-complete-apple, unsignaled-apple, signaled-apple, already-signaled-apple, timeout-expired-apple, condition-satisfied-apple, wait-failed-apple, sync-flush-commands-bit-apple, timeout-ignored-apple.

`arb-shader-objects` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

fragment-shader, fragment-shader-arb, vertex-shader, vertex-shader-arb, program-object-arb, shader-object-arb, max-fragment-uniform-components, max-fragment-uniform-components-arb, max-vertex-uniform-components, max-vertex-uniform-components-arb, max-varying-floats, max-varying-floats-arb, max-vertex-texture-image-units, max-vertex-texture-image-units-arb, max-combined-texture-image-units, max-combined-texture-image-units-arb, object-type-arb, shader-type, object-subtype-arb, float-vec2, float-vec2-arb, float-vec3, float-vec3-arb, float-vec4, float-vec4-arb, int-vec2, int-vec2-arb, int-vec3, int-vec3-arb, int-vec4, int-vec4-arb, bool, bool-arb, bool-vec2, bool-vec2-arb, bool-vec3, bool-vec3-arb, bool-vec4, bool-vec4-arb, float-mat2, float-mat2-arb, float-mat3, float-mat3-arb, float-mat4, float-mat4-arb, sampler-1d, sampler-1d-arb, sampler-2d, sampler-2d-arb, sampler-3d, sampler-3d-arb, sampler-cube, sampler-cube-arb, sampler-1d-shadow, sampler-1d-shadow-arb, sampler-2d-shadow, sampler-2d-shadow-arb, sampler-2d-rect-arb, sampler-2d-rect-shadow-arb, float-mat-2x-3, float-mat-2x-4, float-mat-3x-2, float-mat-3x-4, float-mat-4x-2, float-mat-4x-3, delete-status, object-delete-status-arb, compile-status, object-compile-status-arb, link-status, object-link-status-arb, validate-status, object-validate-status-arb, info-log-length, object-info-log-length-arb, attached-shaders, object-attached-objects-arb, active-uniforms, object-active-uniforms-arb, active-uniform-max-length, object-active-uniform-max-length-arb, shader-source-length, object-shader-source-length-arb, active-attributes, object-active-attributes-arb, active-attribute-max-length, object-active-attribute-max-length-arb, fragment-shader-derivative-hint, fragment-shader-derivative-hint-arb, shading-language-version, shading-language-version-arb.

`arb-vertex-shader` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

fragment-shader, fragment-shader-arb, vertex-shader, vertex-shader-arb, program-object-arb, shader-object-arb, max-fragment-uniform-components, max-fragment-uniform-components-arb, max-vertex-uniform-components, max-vertex-uniform-components-arb, max-varying-floats, max-varying-

floats-arb, max-vertex-texture-image-units, max-vertex-texture-image-units-arb, max-combined-texture-image-units, max-combined-texture-image-units-arb, object-type-arb, shader-type, object-subtype-arb, float-vec2, float-vec2-arb, float-vec3, float-vec3-arb, float-vec4, float-vec4-arb, int-vec2, int-vec2-arb, int-vec3, int-vec3-arb, int-vec4, int-vec4-arb, bool, bool-arb, bool-vec2, bool-vec2-arb, bool-vec3, bool-vec3-arb, bool-vec4, bool-vec4-arb, float-mat2, float-mat2-arb, float-mat3, float-mat3-arb, float-mat4, float-mat4-arb, sampler-1d, sampler-1d-arb, sampler-2d, sampler-2d-arb, sampler-3d, sampler-3d-arb, sampler-cube, sampler-cube-arb, sampler-1d-shadow, sampler-1d-shadow-arb, sampler-2d-shadow, sampler-2d-shadow-arb, sampler-2d-rect-arb, sampler-2d-rect-shadow-arb, float-mat-2x-3, float-mat-2x-4, float-mat-3x-2, float-mat-3x-4, float-mat-4x-2, float-mat-4x-3, delete-status, object-delete-status-arb, compile-status, object-compile-status-arb, link-status, object-link-status-arb, validate-status, object-validate-status-arb, info-log-length, object-info-log-length-arb, attached-shaders, object-attached-objects-arb, active-uniforms, object-active-uniforms-arb, active-uniform-max-length, object-active-uniform-max-length-arb, shader-source-length, object-shader-source-length-arb, active-attributes, object-active-attributes-arb, active-attribute-max-length, object-active-attribute-max-length-arb, fragment-shader-derivative-hint, fragment-shader-derivative-hint-arb, shading-language-version, shading-language-version-arb.

`arb-fragment-shader` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

fragment-shader, fragment-shader-arb, vertex-shader, vertex-shader-arb, program-object-arb, shader-object-arb, max-fragment-uniform-components, max-fragment-uniform-components-arb, max-vertex-uniform-components, max-vertex-uniform-components-arb, max-varying-floats, max-varying-floats-arb, max-vertex-texture-image-units, max-vertex-texture-image-units-arb, max-combined-texture-image-units, max-combined-texture-image-units-arb, object-type-arb, shader-type, object-subtype-arb, float-vec2, float-vec2-arb, float-vec3, float-vec3-arb, float-vec4, float-vec4-arb, int-vec2, int-vec2-arb, int-vec3, int-vec3-arb, int-vec4, int-vec4-arb, bool, bool-arb, bool-vec2, bool-vec2-arb, bool-vec3, bool-vec3-arb, bool-vec4, bool-vec4-arb, float-mat2, float-mat2-arb, float-mat3, float-mat3-arb, float-mat4, float-mat4-arb, sampler-1d, sampler-1d-arb, sampler-2d, sampler-2d-arb, sampler-3d, sampler-3d-arb, sampler-cube, sampler-cube-arb, sampler-1d-shadow, sampler-1d-shadow-arb, sampler-2d-shadow, sampler-2d-shadow-arb, sampler-2d-rect-arb, sampler-2d-rect-shadow-arb, float-mat-2x-3, float-mat-2x-4, float-mat-3x-2, float-mat-3x-4, float-mat-4x-2, float-mat-4x-3, delete-status, object-delete-status-arb, compile-status, object-compile-status-arb, link-status, object-link-status-arb, validate-status,

`object-validate-status-arb`, `info-log-length`, `object-info-log-length-arb`, `attached-shaders`, `object-attached-objects-arb`, `active-uniforms`, `object-active-uniforms-arb`, `active-uniform-max-length`, `object-active-uniform-max-length-arb`, `shader-source-length`, `object-shader-source-length-arb`, `active-attributes`, `object-active-attributes-arb`, `active-attribute-max-length`, `object-active-attribute-max-length-arb`, `fragment-shader-derivative-hint`, `fragment-shader-derivative-hint-arb`, `shading-language-version`, `shading-language-version-arb`.

`nv-vertex-program-3` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`fragment-shader`, `fragment-shader-arb`, `vertex-shader`, `vertex-shader-arb`, `program-object-arb`, `shader-object-arb`, `max-fragment-uniform-components`, `max-fragment-uniform-components-arb`, `max-vertex-uniform-components`, `max-vertex-uniform-components-arb`, `max-varying-floats`, `max-varying-floats-arb`, `max-vertex-texture-image-units`, `max-vertex-texture-image-units-arb`, `max-combined-texture-image-units`, `max-combined-texture-image-units-arb`, `object-type-arb`, `shader-type`, `object-subtype-arb`, `float-vec2`, `float-vec2-arb`, `float-vec3`, `float-vec3-arb`, `float-vec4`, `float-vec4-arb`, `int-vec2`, `int-vec2-arb`, `int-vec3`, `int-vec3-arb`, `int-vec4`, `int-vec4-arb`, `bool`, `bool-arb`, `bool-vec2`, `bool-vec2-arb`, `bool-vec3`, `bool-vec3-arb`, `bool-vec4`, `bool-vec4-arb`, `float-mat2`, `float-mat2-arb`, `float-mat3`, `float-mat3-arb`, `float-mat4`, `float-mat4-arb`, `sampler-1d`, `sampler-1d-arb`, `sampler-2d`, `sampler-2d-arb`, `sampler-3d`, `sampler-3d-arb`, `sampler-cube`, `sampler-cube-arb`, `sampler-1d-shadow`, `sampler-1d-shadow-arb`, `sampler-2d-shadow`, `sampler-2d-shadow-arb`, `sampler-2d-rect-arb`, `sampler-2d-rect-shadow-arb`, `float-mat-2x-3`, `float-mat-2x-4`, `float-mat-3x-2`, `float-mat-3x-4`, `float-mat-4x-2`, `float-mat-4x-3`, `delete-status`, `object-delete-status-arb`, `compile-status`, `object-compile-status-arb`, `link-status`, `object-link-status-arb`, `validate-status`, `object-validate-status-arb`, `info-log-length`, `object-info-log-length-arb`, `attached-shaders`, `object-attached-objects-arb`, `active-uniforms`, `object-active-uniforms-arb`, `active-uniform-max-length`, `object-active-uniform-max-length-arb`, `shader-source-length`, `object-shader-source-length-arb`, `active-attributes`, `object-active-attributes-arb`, `active-attribute-max-length`, `object-active-attribute-max-length-arb`, `fragment-shader-derivative-hint`, `fragment-shader-derivative-hint-arb`, `shading-language-version`, `shading-language-version-arb`.

`oes-standard-derivatives` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`fragment-shader-derivative-hint-oes`.

`ext-geometry-shader-4` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`max-varying-components-ext`, `geometry-shader-ext`, `max-geometry-varying-components-ext`, `max-vertex-varying-components-ext`, `max-geometry-uniform-components-ext`, `max-geometry-output-vertices-ext`, `max-geometry-total-output-components-ext`.

`oes-compressed-paletted-texture` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`palette4-rgb8-oes`, `palette4-rgba8-oes`, `palette4-r5-g6-b5-oes`, `palette4-rgba4-oes`, `palette4-rgb5-a1-oes`, `palette8-rgb8-oes`, `palette8-rgba8-oes`, `palette8-r5-g6-b5-oes`, `palette8-rgba4-oes`, `palette8-rgb5-a1-oes`.

`oes-read-format` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`implementation-color-read-type-oes`, `implementation-color-read-format-oes`.

`oes-draw-texture` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-crop-rect-oes`.

`mesa-program-debug` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`fragment-program-position-mesa`, `fragment-program-callback-mesa`, `fragment-program-callback-func-mesa`, `fragment-program-callback-data-mesa`, `vertex-program-callback-mesa`, `vertex-program-position-mesa`, `vertex-program-callback-func-mesa`, `vertex-program-callback-data-mesa`.

`amd-performance-monitor` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`counter-type-amd`, `counter-range-amd`, `unsigned-int64-amd`, `percentage-amd`, `perfmon-result-available-amd`, `perfmon-result-size-amd`, `perfmon-result-amd`.

`qcom-extended-get` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-width-qcom`, `texture-height-qcom`, `texture-depth-qcom`, `texture-internal-format-qcom`, `texture-format-qcom`, `texture-type-qcom`, `texture-image-valid-qcom`, `texture-num-levels-qcom`, `texture-target-qcom`, `texture-object-valid-qcom`, `state-restore`.

`img-texture-compression-pvrtc` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`compressed-rgb-pvrtc-4bppv1-img`, `compressed-rgb-pvrtc-2bppv1-img`, `compressed-rgba-pvrtc-4bppv1-img`, `compressed-rgba-pvrtc-2bppv1-img`.

`img-shader-binary` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`sgx-binary-img`.

`arb-texture-buffer-object` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-buffer-arb`, `max-texture-buffer-size-arb`, `texture-binding-buffer-arb`, `texture-buffer-data-store-binding-arb`, `texture-buffer-format-arb`.

`ext-texture-buffer-object` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-buffer-ext`, `max-texture-buffer-size-ext`, `texture-binding-buffer-ext`, `texture-buffer-data-store-binding-ext`, `texture-buffer-format-ext`.

`arb-occlusion-query-2` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`any-samples-passed`.

`arb-sample-shading` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`sample-shading-arb`, `min-sample-shading-value-arb`.

`ext-packed-float` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`r11f-g11f-b10f-ext`, `unsigned-int-10f-11f-11f-rev-ext`, `rgba-signed-components-ext`.

`ext-texture-shared-exponent` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`rgb9-e5-ext`, `unsigned-int-5-9-9-9-rev-ext`, `texture-shared-size-ext`.

`ext-texture-s-rgb` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`srgb-ext`, `srgb8-ext`, `srgb-alpha-ext`, `srgb8-alpha8-ext`, `sluminance-alpha-ext`, `sluminance8-alpha8-ext`, `sluminance-ext`, `sluminance8-ext`, `compressed-srgb-ext`, `compressed-srgb-alpha-ext`, `compressed-sluminance-ext`, `compressed-sluminance-alpha-ext`, `compressed-srgb-s3tc-dxt1-ext`, `compressed-srgb-alpha-s3tc-dxt1-ext`, `compressed-srgb-alpha-s3tc-dxt3-ext`, `compressed-srgb-alpha-s3tc-dxt5-ext`.

`ext-texture-compression-latc` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`compressed-luminance-latc1-ext`, `compressed-signed-luminance-latc1-ext`, `compressed-luminance-alpha-latc2-ext`, `compressed-signed-luminance-alpha-latc2-ext`.

`ext-transform-feedback` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`transform-feedback-varying-max-length`, `transform-feedback-varying-max-length-ext`, `back-primary-color-nv`, `back-secondary-color-nv`, `texture-coord-nv`, `clip-distance-nv`, `vertex-id-nv`, `primitive-id-nv`, `generic-attrib-nv`, `transform-feedback-attribs-nv`, `transform-feedback-buffer-mode`, `transform-feedback-buffer-mode-ext`, `transform-feedback-buffer-mode-nv`, `max-transform-feedback-separate-components`, `max-transform-feedback-separate-components-ext`, `max-transform-feedback-separate-components-nv`, `active-varyings-nv`, `active-varying-max-length-nv`, `transform-feedback-varyings`, `transform-feedback-varyings-ext`, `transform-feedback-varyings-nv`, `transform-feedback-buffer-start`,

transform-feedback-buffer-start-ext, transform-feedback-buffer-start-nv, transform-feedback-buffer-size, transform-feedback-buffer-size-ext, transform-feedback-buffer-size-nv, transform-feedback-record-nv, primitives-generated, primitives-generated-ext, primitives-generated-nv, transform-feedback-primitives-written, transform-feedback-primitives-written-ext, transform-feedback-primitives-written-nv, rasterizer-discard, rasterizer-discard-ext, rasterizer-discard-nv, max-transform-feedback-interleaved-components, max-transform-feedback-interleaved-components-ext, max-transform-feedback-interleaved-components-nv, max-transform-feedback-separate-attribs, max-transform-feedback-separate-attribs-ext, max-transform-feedback-separate-attribs-nv, interleaved-attribs, interleaved-attribs-ext, interleaved-attribs-nv, separate-attribs, separate-attribs-ext, separate-attribs-nv, transform-feedback-buffer, transform-feedback-buffer-ext, transform-feedback-buffer-nv, transform-feedback-buffer-binding, transform-feedback-buffer-binding-ext, transform-feedback-buffer-binding-nv.

`nv-transform-feedback` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

transform-feedback-varying-max-length, transform-feedback-varying-max-length-ext, back-primary-color-nv, back-secondary-color-nv, texture-coord-nv, clip-distance-nv, vertex-id-nv, primitive-id-nv, generic-attrib-nv, transform-feedback-attribs-nv, transform-feedback-buffer-mode, transform-feedback-buffer-mode-ext, transform-feedback-buffer-mode-nv, max-transform-feedback-separate-components, max-transform-feedback-separate-components-ext, max-transform-feedback-separate-components-nv, active-varyings-nv, active-varying-max-length-nv, transform-feedback-varyings, transform-feedback-varyings-ext, transform-feedback-varyings-nv, transform-feedback-buffer-start, transform-feedback-buffer-start-ext, transform-feedback-buffer-start-nv, transform-feedback-buffer-size, transform-feedback-buffer-size-ext, transform-feedback-buffer-size-nv, transform-feedback-record-nv, primitives-generated, primitives-generated-ext, primitives-generated-nv, transform-feedback-primitives-written, transform-feedback-primitives-written-ext, transform-feedback-primitives-written-nv, rasterizer-discard, rasterizer-discard-ext, rasterizer-discard-nv, max-transform-feedback-interleaved-components, max-transform-feedback-interleaved-components-ext, max-transform-feedback-interleaved-components-nv, max-transform-feedback-separate-attribs, max-transform-feedback-separate-attribs-ext, max-transform-feedback-separate-attribs-nv, interleaved-attribs, interleaved-attribs-ext, interleaved-attribs-nv, separate-attribs, separate-attribs-ext, separate-attribs-nv, transform-feedback-buffer, transform-feedback-buffer-ext, transform-feedback-buffer-nv, transform-feedback-buffer-

`binding`, `transform-feedback-buffer-binding-ext`, `transform-feedback-buffer-binding-nv`, `layer-nv`, `next-buffer-nv`, `skip-components4-nv`, `skip-components3-nv`, `skip-components2-nv`, `skip-components1-nv`.

`ext-framebuffer-blit` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`draw-framebuffer-binding-ext`, `read-framebuffer-ext`, `draw-framebuffer-ext`, `draw-framebuffer-binding-ext`, `read-framebuffer-binding-ext`.

`angle-framebuffer-blit` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`framebuffer-binding-angle`, `renderbuffer-binding-angle`, `read-framebuffer-angle`, `draw-framebuffer-angle`.

`nv-framebuffer-blit` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`read-framebuffer-nv`, `draw-framebuffer-nv`, `draw-framebuffer-binding-nv`, `read-framebuffer-binding-nv`.

`angle-framebuffer-multisample` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`renderbuffer-samples-angle`, `framebuffer-incomplete-multisample-angle`, `max-samples-angle`.

`ext-framebuffer-multisample` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`renderbuffer-samples-ext`, `framebuffer-incomplete-multisample-ext`, `max-samples-ext`.

`nv-framebuffer-multisample` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`renderbuffer-samples-nv`, `framebuffer-incomplete-multisample-nv`, `max-samples-nv`.

`nv-framebuffer-multisample-coverage` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`renderbuffer-coverage-samples-nv`, `renderbuffer-color-samples-nv`, `max-multisample-coverage-modes-nv`, `multisample-coverage-modes-nv`.

`arb-depth-buffer-float` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`depth-component32f`, `depth32f-stencil8`, `float-32-unsigned-int-24-8-rev`.

`nv-fbo-color-attachments` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`max-color-attachments-nv`.

`oes-stencil-1` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`stencil-index1-oes`.

`oes-stencil-4` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`stencil-index4-oes`.

`oes-stencil-8` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`stencil-index8-oes`.

`oes-vertex-half-float` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`half-float-oes`.

`version-4-1` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`rgb565`.

`oes-compressed-etc1-rgb8-texture` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`etc1-rgb8-oes`.

`oes-egl-image-external` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-external-oes`, `sampler-external-oes`, `texture-binding-external-oes`, `required-texture-image-units-oes`.

`arb-es3-compatibility` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`primitive-restart-fixed-index`, `any-samples-passed-conservative`, `max-element-index`, `compressed-r11-eac`, `compressed-signed-r11-eac`, `compressed-rg11-eac`, `compressed-signed-rg11-eac`, `compressed-rgb8-etc2`, `compressed-srgb8-etc2`, `compressed-rgb8-punchthrough-alpha1-etc2`, `compressed-srgb8-punchthrough-alpha1-etc2`, `compressed-rgba8-etc2-eac`, `compressed-srgb8-alpha8-etc2-eac`.

`ext-multisampled-render-to-texture` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`framebuffer-attachment-texture-samples-ext`.

`ext-texture-integer` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`rgba32ui`, `rgba32ui-ext`, `rgb32ui`, `rgb32ui-ext`, `alpha32ui-ext`, `intensity32ui-ext`, `luminance32ui-ext`, `luminance-alpha32ui-ext`, `rgba16ui`, `rgba16ui-ext`, `rgb16ui`, `rgb16ui-ext`, `alpha16ui-ext`, `intensity16ui-ext`, `luminance16ui-ext`, `luminance-alpha16ui-ext`, `rgba8ui`, `rgba8ui-ext`, `rgb8ui`, `rgb8ui-ext`, `alpha8ui-ext`, `intensity8ui-ext`, `luminance8ui-ext`, `luminance-alpha8ui-ext`, `rgba32i`, `rgba32i-ext`, `rgb32i`, `rgb32i-ext`, `alpha32i-ext`, `intensity32i-ext`, `luminance32i-ext`, `luminance-alpha32i-ext`, `rgba16i`, `rgba16i-ext`, `rgb16i`, `rgb16i-ext`, `alpha16i-ext`, `intensity16i-ext`, `luminance16i-ext`, `luminance-alpha16i-ext`, `rgba8i`, `rgba8i-ext`, `rgb8i`, `rgb8i-ext`, `alpha8i-ext`, `intensity8i-ext`, `luminance8i-ext`, `luminance-alpha8i-ext`, `red-integer`, `red-integer-ext`, `green-integer`, `green-integer-ext`, `blue-integer`, `blue-integer-ext`, `alpha-integer`, `alpha-integer-ext`, `rgb-integer`, `rgb-integer-ext`, `rgba-integer`,

`rgba-integer-ext`, `bgr-integer`, `bgr-integer-ext`, `bgra-integer`, `bgra-integer-ext`, `luminance-integer-ext`, `luminance-alpha-integer-ext`, `rgba-integer-mode-ext`.

`arb-vertex-type-2-10-10-10-rev` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`int-2-10-10-10-rev`.

`nv-parameter-buffer-object` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`max-program-parameter-buffer-bindings-nv`, `max-program-parameter-buffer-size-nv`, `vertex-program-parameter-buffer-nv`, `geometry-program-parameter-buffer-nv`, `fragment-program-parameter-buffer-nv`.

`nv-depth-buffer-float` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`depth-component32f-nv`, `depth32f-stencil8-nv`, `float-32-unsigned-int-24-8-rev-nv`, `depth-buffer-float-mode-nv`.

`arb-shading-language-include` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`shader-include-arb`, `named-string-length-arb`, `named-string-type-arb`.

`arb-framebuffer-s-rgb` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`framebuffer-srgb`.

`ext-framebuffer-s-rgb` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`framebuffer-srgb-ext`, `framebuffer-srgb-capable-ext`.

`arb-texture-compression-rgtc` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`compressed-red-rgtc1`, `compressed-signed-red-rgtc1`, `compressed-rg-rgtc2`, `compressed-signed-rg-rgtc2`.

`ext-texture-compression-rgtc` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`compressed-red-rgtc1-ext`, `compressed-signed-red-rgtc1-ext`, `compressed-red-green-rgtc2-ext`, `compressed-signed-red-green-rgtc2-ext`.

`ext-gpu-shader-4` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`sampler-1d-array-ext`, `sampler-2d-array-ext`, `sampler-buffer-ext`, `sampler-1d-array-shadow-ext`, `sampler-2d-array-shadow-ext`, `sampler-cube-shadow-ext`, `unsigned-int-vec2-ext`, `unsigned-int-vec3-ext`, `unsigned-int-vec4-ext`, `int-sampler-1d-ext`, `int-sampler-2d-ext`, `int-sampler-3d-ext`, `int-sampler-cube-ext`, `int-sampler-2d-rect-ext`, `int-sampler-1d-array-ext`, `int-sampler-2d-array-ext`, `int-sampler-buffer-ext`, `unsigned-int-sampler-1d-ext`, `unsigned-int-sampler-2d-ext`, `unsigned-int-sampler-3d-ext`, `unsigned-int-sampler-cube-ext`, `unsigned-int-sampler-2d-rect-ext`, `unsigned-int-sampler-1d-array-ext`, `unsigned-int-sampler-2d-array-ext`, `unsigned-int-sampler-buffer-ext`.

`nv-shadow-samplers-array` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`sampler-2d-array-shadow-nv`.

`nv-shadow-samplers-cube` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`sampler-cube-shadow-nv`.

`ext-bindable-uniform` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`max-vertex-bindable-uniforms-ext`, `max-fragment-bindable-uniforms-ext`, `max-geometry-bindable-uniforms-ext`, `max-bindable-uniform-size-ext`, `uniform-buffer-ext`, `uniform-buffer-binding-ext`.

`arb-shader-subroutine` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`active-subroutines`, `active-subroutine-uniforms`, `max-subroutines`, `max-subroutine-uniform-locations`, `active-subroutine-uniform-locations`,

`active-subroutine-max-length`, `active-subroutine-uniform-max-length`, `num-compatible-subroutines`, `compatible-subroutines`.

`oes-vertex-type-10-10-10-2` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`unsigned-int-10-10-10-2-oes`, `int-10-10-10-2-oes`.

`nv-conditional-render` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`query-wait-nv`, `query-no-wait-nv`, `query-by-region-wait-nv`, `query-by-region-no-wait-nv`.

`arb-transform-feedback-2` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`transform-feedback`, `transform-feedback-paused`, `transform-feedback-buffer-paused`, `transform-feedback-active`, `transform-feedback-buffer-active`, `transform-feedback-binding`.

`nv-transform-feedback-2` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`transform-feedback-nv`, `transform-feedback-buffer-paused-nv`, `transform-feedback-buffer-active-nv`, `transform-feedback-binding-nv`.

`nv-present-video` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`frame-nv`, `fields-nv`, `current-time-nv`, `num-fill-streams-nv`, `present-time-nv`, `present-duration-nv`.

`nv-depth-nonlinear` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`depth-component16-nonlinear-nv`.

`ext-direct-state-access` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`program-matrix-ext`, `transpose-program-matrix-ext`, `program-matrix-stack-depth-ext`.

`arb-texture-swizzle` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-swizzle-r`, `texture-swizzle-g`, `texture-swizzle-b`, `texture-swizzle-a`, `texture-swizzle-rgba`.

`ext-texture-swizzle` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-swizzle-r-ext`, `texture-swizzle-g-ext`, `texture-swizzle-b-ext`, `texture-swizzle-a-ext`, `texture-swizzle-rgba-ext`.

`arb-provoking-vertex` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`quads-follow-provoking-vertex-convention`, `first-vertex-convention`, `last-vertex-convention`, `provoking-vertex`.

`ext-provoking-vertex` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`quads-follow-provoking-vertex-convention-ext`, `first-vertex-convention-ext`, `last-vertex-convention-ext`, `provoking-vertex-ext`.

`arb-texture-multisample` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`sample-position`, `sample-mask`, `sample-mask-value`, `max-sample-mask-words`, `texture-2d-multisample`, `proxy-texture-2d-multisample`, `texture-2d-multisample-array`, `proxy-texture-2d-multisample-array`, `texture-binding-2d-multisample`, `texture-binding-2d-multisample-array`, `texture-samples`, `texture-fixed-sample-locations`, `sampler-2d-multisample`, `int-sampler-2d-multisample`, `unsigned-int-sampler-2d-multisample`, `sampler-2d-multisample-array`, `int-sampler-2d-multisample-array`, `unsigned-int-sampler-2d-multisample-array`, `max-color-texture-samples`, `max-depth-texture-samples`, `max-integer-samples`.

`nv-explicit-multisample` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`sample-position-nv`, `sample-mask-nv`, `sample-mask-value-nv`, `texture-binding-renderbuffer-nv`, `texture-renderbuffer-data-store-binding-nv`, `texture-renderbuffer-nv`, `sampler-renderbuffer-nv`, `int-sampler-renderbuffer-nv`, `unsigned-int-sampler-renderbuffer-nv`, `max-sample-mask-words-nv`.

`nv-gpu-program-5` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`max-geometry-program-invocations-nv`, `min-fragment-interpolation-offset-nv`, `max-fragment-interpolation-offset-nv`, `fragment-program-interpolation-offset-bits-nv`, `min-program-texture-gather-offset-nv`, `max-program-texture-gather-offset-nv`, `max-program-subroutine-parameters-nv`, `max-program-subroutine-num-nv`.

`arb-texture-gather` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`min-program-texture-gather-offset`, `max-program-texture-gather-offset`, `max-program-texture-gather-components-arb`, `max-program-texture-gather-components`.

`arb-transform-feedback-3` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`max-transform-feedback-buffers`, `max-vertex-streams`.

`arb-texture-compression-bptc` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`compressed-rgba-bptc-unorm-arb`, `compressed-srgb-alpha-bptc-unorm-arb`, `compressed-rgb-bptc-signed-float-arb`, `compressed-rgb-bptc-unsigned-float-arb`.

`nv-coverage-sample` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`coverage-component-nv`, `coverage-component4-nv`, `coverage-attachment-nv`, `coverage-buffers-nv`, `coverage-samples-nv`, `coverage-all-fragments-nv`, `coverage-edge-fragments-nv`, `coverage-automatic-nv`, `coverage-buffer-bit-nv`.

`nv-shader-buffer-load` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`buffer-gpu-address-nv`, `gpu-address-nv`, `max-shader-buffer-address-nv`.

`nv-vertex-buffer-unified-memory` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`vertex-attrib-array-unified-nv`, `element-array-unified-nv`, `vertex-attrib-array-address-nv`, `vertex-array-address-nv`, `normal-array-address-nv`, `color-array-address-nv`, `index-array-address-nv`, `texture-coord-array-address-nv`, `edge-flag-array-address-nv`, `secondary-color-array-address-nv`, `fog-coord-array-address-nv`, `element-array-address-nv`, `vertex-attrib-array-length-nv`, `vertex-array-length-nv`, `normal-array-length-nv`, `color-array-length-nv`, `index-array-length-nv`, `texture-coord-array-length-nv`, `edge-flag-array-length-nv`, `secondary-color-array-length-nv`, `fog-coord-array-length-nv`, `element-array-length-nv`, `draw-indirect-unified-nv`, `draw-indirect-address-nv`, `draw-indirect-length-nv`.

`arb-copy-buffer` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`copy-read-buffer-binding`, `copy-read-buffer`, `copy-write-buffer-binding`, `copy-write-buffer`.

`arb-draw-indirect` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`draw-indirect-buffer`, `draw-indirect-buffer-binding`.

`arb-gpu-shader-fp-64` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`double-mat2`, `double-mat3`, `double-mat4`, `double-mat-2x-3`, `double-mat-2x-4`, `double-mat-3x-2`, `double-mat-3x-4`, `double-mat-4x-2`, `double-mat-4x-3`, `double-vec2`, `double-vec3`, `double-vec4`.

`arm-mali-shader-binary` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`mali-shader-binary-arm`.

`qcom-driver-control` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`perfmon-global-mode-qcom`.

`qcom-binning-control` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`binning-control-hint-qcom`, `cpu-optimized-qcom`, `gpu-optimized-qcom`, `render-direct-to-framebuffer-qcom`.

`viv-shader-binary` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`shader-binary-viv`.

`amd-vertex-shader-tesselator` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`sampler-buffer-amd`, `int-sampler-buffer-amd`, `unsigned-int-sampler-buffer-amd`, `tessellation-mode-amd`, `tessellation-factor-amd`, `discrete-amd`, `continuous-amd`.

`arb-texture-cube-map-array` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-cube-map-array`, `texture-binding-cube-map-array`, `proxy-texture-cube-map-array`, `sampler-cube-map-array`, `sampler-cube-map-array-shadow`, `int-sampler-cube-map-array`, `unsigned-int-sampler-cube-map-array`.

`ext-texture-snorm` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`alpha-snorm`, `luminance-snorm`, `luminance-alpha-snorm`, `intensity-snorm`, `alpha8-snorm`, `luminance8-snorm`, `luminance8-alpha8-snorm`, `intensity8-snorm`, `alpha16-snorm`, `luminance16-snorm`, `luminance16-alpha16-snorm`, `intensity16-snorm`.

`amd-blend-minmax-factor` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`factor-min-amd`, `factor-max-amd`.

`amd-depth-clamp-separate` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`depth-clamp-near-amd`, `depth-clamp-far-amd`.

`nv-video-capture` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`video-buffer-nv`, `video-buffer-binding-nv`, `field-upper-nv`, `field-lower-nv`, `num-video-capture-streams-nv`, `next-video-capture-buffer-status-nv`, `video-capture-to-422-supported-nv`, `last-video-capture-status-nv`, `video-buffer-pitch-nv`, `video-color-conversion-matrix-nv`, `video-color-conversion-max-nv`, `video-color-conversion-min-nv`, `video-color-conversion-offset-nv`, `video-buffer-internal-format-nv`, `partial-success-nv`, `success-nv`, `failure-nv`, `ycbycr8-422-nv`, `ycbaycr8a-4224-nv`, `z6y10z6cb10z6y10z6cr10-422-nv`, `z6y10z6cb10z6a10z6y10z6cr10z6a10-4224-nv`, `z4y12z4cb12z4y12z4cr12-422-nv`, `z4y12z4cb12z4a12z4y12z4cr12z4a12-4224-nv`, `z4y12z4cb12z4cr12-444-nv`, `video-capture-frame-width-nv`, `video-capture-frame-height-nv`, `video-capture-field-upper-height-nv`, `video-capture-field-lower-height-nv`, `video-capture-surface-origin-nv`.

`nv-texture-multisample` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-coverage-samples-nv`, `texture-color-samples-nv`.

`arb-texture-rgb-10-a-2-ui` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`rgb10-a2ui`.

`nv-path-rendering` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`path-format-svg-nv`, `path-format-ps-nv`, `standard-font-name-nv`, `system-font-name-nv`, `file-name-nv`, `path-stroke-width-nv`, `path-end-caps-nv`, `path-initial-end-cap-nv`, `path-terminal-end-cap-nv`, `path-join-style-nv`, `path-miter-limit-nv`, `path-dash-caps-nv`, `path-initial-dash-cap-nv`, `path-terminal-dash-cap-nv`, `path-dash-offset-nv`, `path-client-length-nv`, `path-fill-mode-nv`, `path-fill-mask-nv`, `path-fill-cover-mode-nv`, `path-stroke-cover-mode-nv`, `path-stroke-mask-nv`, `count-up-nv`, `count-down-nv`, `path-object-bounding-box-nv`, `convex-hull-nv`, `bounding-box-nv`,

translate-x-nv, translate-y-nv, translate-2d-nv, translate-3d-nv, affine-2d-nv, affine-3d-nv, transpose-affine-2d-nv, transpose-affine-3d-nv, utf8-nv, utf16-nv, bounding-box-of-bounding-boxes-nv, path-command-count-nv, path-coord-count-nv, path-dash-array-count-nv, path-computed-length-nv, path-fill-bounding-box-nv, path-stroke-bounding-box-nv, square-nv, round-nv, triangular-nv, bevel-nv, miter-revert-nv, miter-truncate-nv, skip-missing-glyph-nv, use-missing-glyph-nv, path-error-position-nv, path-fog-gen-mode-nv, accum-adjacent-pairs-nv, adjacent-pairs-nv, first-to-rest-nv, path-gen-mode-nv, path-gen-coeff-nv, path-gen-color-format-nv, path-gen-components-nv, path-dash-offset-reset-nv, move-to-resets-nv, move-to-continues-nv, path-stencil-func-nv, path-stencil-ref-nv, path-stencil-value-mask-nv, close-path-nv, move-to-nv, relative-move-to-nv, line-to-nv, relative-line-to-nv, horizontal-line-to-nv, relative-horizontal-line-to-nv, vertical-line-to-nv, relative-vertical-line-to-nv, quadratic-curve-to-nv, relative-quadratic-curve-to-nv, cubic-curve-to-nv, relative-cubic-curve-to-nv, smooth-quadratic-curve-to-nv, relative-smooth-quadratic-curve-to-nv, smooth-cubic-curve-to-nv, relative-smooth-cubic-curve-to-nv, small-ccw-arc-to-nv, relative-small-ccw-arc-to-nv, small-cw-arc-to-nv, relative-small-cw-arc-to-nv, large-ccw-arc-to-nv, relative-large-ccw-arc-to-nv, large-cw-arc-to-nv, relative-large-cw-arc-to-nv, restart-path-nv, dup-first-cubic-curve-to-nv, dup-last-cubic-curve-to-nv, rect-nv, circular-ccw-arc-to-nv, circular-cw-arc-to-nv, circular-tangent-arc-to-nv, arc-to-nv, relative-arc-to-nv, bold-bit-nv, italic-bit-nv, glyph-width-bit-nv, glyph-height-bit-nv, glyph-horizontal-bearing-x-bit-nv, glyph-horizontal-bearing-y-bit-nv, glyph-horizontal-bearing-advance-bit-nv, glyph-vertical-bearing-x-bit-nv, glyph-vertical-bearing-y-bit-nv, glyph-vertical-bearing-advance-bit-nv, glyph-has-kerning-bit-nv, font-x-min-bounds-bit-nv, font-y-min-bounds-bit-nv, font-x-max-bounds-bit-nv, font-y-max-bounds-bit-nv, font-units-per-em-bit-nv, font-ascender-bit-nv, font-descender-bit-nv, font-height-bit-nv, font-max-advance-width-bit-nv, font-max-advance-height-bit-nv, font-underline-position-bit-nv, font-underline-thickness-bit-nv, font-has-kerning-bit-nv, path-stencil-depth-offset-factor-nv, path-stencil-depth-offset-units-nv, path-cover-depth-func-nv.

`ext-framebuffer-multisample-blit-scaled` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`scaled-resolve-fastest-ext`, `scaled-resolve-nicest-ext`.

`arb-map-buffer-alignment` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`min-map-buffer-alignment`.

`nv-deep-texture-3d` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`max-deep-3d-texture-width-height-nv`, `max-deep-3d-texture-depth-nv`.

`ext-x-11-sync-object` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`sync-x11-fence-ext`.

`arb-stencil-texturing` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`depth-stencil-texture-mode`.

`nv-compute-program-5` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`compute-program-nv`, `compute-program-parameter-buffer-nv`.

`arb-sync` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`max-server-wait-timeout`, `object-type`, `sync-condition`, `sync-status`, `sync-flags`, `sync-fence`, `sync-gpu-commands-complete`, `unsignaled`, `signaled`, `already-signaled`, `timeout-expired`, `condition-satisfied`, `wait-failed`, `sync-flush-commands-bit`, `timeout-ignored`.

`arb-compressed-texture-pixel-storage` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`unpack-compressed-block-width`, `unpack-compressed-block-height`, `unpack-compressed-block-depth`, `unpack-compressed-block-size`, `pack-compressed-block-width`, `pack-compressed-block-height`, `pack-compressed-block-depth`, `pack-compressed-block-size`.

`arb-texture-storage` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-immutable-format`.

`img-program-binary` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`sgx-program-binary-img`.

`img-multisampled-render-to-texture` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`renderbuffer-samples-img`, `framebuffer-incomplete-multisample-img`, `max-samples-img`, `texture-samples-img`.

`img-texture-compression-pvrtc-2` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`compressed-rgba-pvrtc-2bppv2-img`, `compressed-rgba-pvrtc-4bppv2-img`.

`amd-debug-output` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`max-debug-message-length-amd`, `max-debug-logged-messages-amd`, `debug-logged-messages-amd`, `debug-severity-high-amd`, `debug-severity-medium-amd`, `debug-severity-low-amd`, `debug-category-api-error-amd`, `debug-category-window-system-amd`, `debug-category-deprecation-amd`, `debug-category-undefined-behavior-amd`, `debug-category-performance-amd`, `debug-category-shader-compiler-amd`, `debug-category-application-amd`, `debug-category-other-amd`.

`amd-name-gen-delete` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`data-buffer-amd`, `performance-monitor-amd`, `query-object-amd`, `vertex-array-object-amd`, `sampler-object-amd`.

`amd-pinned-memory` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`external-virtual-memory-buffer-amd`.

`amd-query-buffer-object` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`query-buffer-amd`, `query-buffer-binding-amd`, `query-result-no-wait-amd`.

`amd-sparse-texture` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`virtual-page-size-x-amd`, `virtual-page-size-y-amd`, `virtual-page-size-z-amd`, `max-sparse-texture-size-amd`, `max-sparse-3d-texture-size-amd`, `max-sparse-array-texture-layers`, `min-sparse-level-amd`, `min-lod-warning-amd`, `texture-storage-sparse-bit-amd`.

`arb-texture-buffer-range` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-buffer-offset`, `texture-buffer-size`, `texture-buffer-offset-alignment`.

`dmp-shader-binary` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`shader-binary-dmp`.

`fj-shader-binary-gccso` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`gccso-shader-binary-fj`.

`arb-shader-atomic-counters` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`atomic-counter-buffer`, `atomic-counter-buffer-binding`, `atomic-counter-buffer-start`, `atomic-counter-buffer-size`, `atomic-counter-buffer-data-size`, `atomic-counter-buffer-active-atomic-counters`, `atomic-counter-buffer-active-atomic-counter-indices`, `atomic-counter-buffer-referenced-by-vertex-shader`, `atomic-counter-buffer-referenced-by-tess-control-shader`, `atomic-counter-buffer-referenced-by-tess-evaluation-shader`, `atomic-counter-buffer-referenced-by-geometry-shader`, `atomic-counter-buffer-referenced-by-fragment-shader`, `max-vertex-atomic-counter-buffers`, `max-tess-control-atomic-counter-buffers`, `max-tess-evaluation-atomic-counter-buffers`, `max-geometry-atomic-counter-buffers`, `max-fragment-atomic-counter-buffers`, `max-combined-atomic-counter-buffers`, `max-vertex-atomic-counters`, `max-tess-control-atomic-counters`, `max-tess-evaluation-atomic-counters`, `max-geometry-atomic-counters`, `max-fragment-atomic-counters`, `max-combined-atomic-counters`, `max-atomic-counter-buffer-size`, `max-atomic-counter-buffer-bindings`,

`active-atomic-counter-buffers`, `uniform-atomic-counter-buffer-index`, `unsigned-int-atomic-counter`.

`arb-program-interface-query` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`uniform`, `uniform-block`, `program-input`, `program-output`, `buffer-variable`, `shader-storage-block`, `is-per-patch`, `vertex-subroutine`, `tess-control-subroutine`, `tess-evaluation-subroutine`, `geometry-subroutine`, `fragment-subroutine`, `compute-subroutine`, `vertex-subroutine-uniform`, `tess-control-subroutine-uniform`, `tess-evaluation-subroutine-uniform`, `geometry-subroutine-uniform`, `fragment-subroutine-uniform`, `compute-subroutine-uniform`, `transform-feedback-varying`, `active-resources`, `max-name-length`, `max-num-active-variables`, `max-num-compatible-subroutines`, `name-length`, `type`, `array-size`, `offset`, `block-index`, `array-stride`, `matrix-stride`, `is-row-major`, `atomic-counter-buffer-index`, `buffer-binding`, `buffer-data-size`, `num-active-variables`, `active-variables`, `referenced-by-vertex-shader`, `referenced-by-tess-control-shader`, `referenced-by-tess-evaluation-shader`, `referenced-by-geometry-shader`, `referenced-by-fragment-shader`, `referenced-by-compute-shader`, `top-level-array-size`, `top-level-array-stride`, `location`, `location-index`.

`arb-framebuffer-no-attachments` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`framebuffer-default-width`, `framebuffer-default-height`, `framebuffer-default-layers`, `framebuffer-default-samples`, `framebuffer-default-fixed-sample-locations`, `max-framebuffer-width`, `max-framebuffer-height`, `max-framebuffer-layers`, `max-framebuffer-samples`.

`arb-internalformat-query` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`num-sample-counts`.

`angle-translated-shader-source` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`translated-shader-source-length-angle`.

`angle-texture-usage` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-usage-angle`, `framebuffer-attachment-angle`, `none`.

`angle-pack-reverse-row-order` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`pack-reverse-row-order-angle`.

`angle-depth-texture` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`program-binary-angle`.

`gl-khr-texture-compression-astc-ldr` *enum* [Macro]
Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`compressed-rgba-astc-4x4-khr`, `compressed-rgba-astc-5x4-khr`, `compressed-rgba-astc-5x5-khr`, `compressed-rgba-astc-6x5-khr`, `compressed-rgba-astc-6x6-khr`, `compressed-rgba-astc-8x5-khr`, `compressed-rgba-astc-8x6-khr`, `compressed-rgba-astc-8x8-khr`, `compressed-rgba-astc-10x5-khr`, `compressed-rgba-astc-10x6-khr`, `compressed-rgba-astc-10x8-khr`, `compressed-rgba-astc-10x10-khr`, `compressed-rgba-astc-12x10-khr`, `compressed-rgba-astc-12x12-khr`, `compressed-srgb8-alpha8-astc-4x4-khr`, `compressed-srgb8-alpha8-astc-5x4-khr`, `compressed-srgb8-alpha8-astc-5x5-khr`, `compressed-srgb8-alpha8-astc-6x5-khr`, `compressed-srgb8-alpha8-astc-6x6-khr`, `compressed-srgb8-alpha8-astc-8x5-khr`, `compressed-srgb8-alpha8-astc-8x6-khr`, `compressed-srgb8-alpha8-astc-8x8-khr`, `compressed-srgb8-alpha8-astc-10x5-khr`, `compressed-srgb8-alpha8-astc-10x6-khr`, `compressed-srgb8-alpha8-astc-10x8-khr`, `compressed-srgb8-alpha8-astc-10x10-khr`, `compressed-srgb8-alpha8-astc-12x10-khr`, `compressed-srgb8-alpha8-astc-12x12-khr`.

3.6 Low-Level GL

The functions from this section may be had by loading the module:

```
(use-modules (gl low-level)
```

This section of the manual was derived from the upstream OpenGL documentation. Each function's documentation has its own copyright statement; for full details, see the upstream documentation. The copyright notices and licenses present in this section are as follows.

`void glAccum` *op value* [Function]

Operate on the accumulation buffer.

op Specifies the accumulation buffer operation. Symbolic constants `GL_ACCUM`, `GL_LOAD`, `GL_ADD`, `GL_MULT`, and `GL_RETURN` are accepted.

value Specifies a floating-point value used in the accumulation buffer operation. *op* determines how *value* is used.

The accumulation buffer is an extended-range color buffer. Images are not rendered into it. Rather, images rendered into one of the color buffers are added to the contents of the accumulation buffer after rendering. Effects such as antialiasing (of points, lines, and polygons), motion blur, and depth of field can be created by accumulating images generated with different transformation matrices.

Each pixel in the accumulation buffer consists of red, green, blue, and alpha values. The number of bits per component in the accumulation buffer depends on the implementation. You can examine this number by calling `glGetIntegerv` four times, with arguments `GL_ACCUM_RED_BITS`, `GL_ACCUM_GREEN_BITS`, `GL_ACCUM_BLUE_BITS`, and `GL_ACCUM_ALPHA_BITS`. Regardless of the number of bits per component, the range of values stored by each component is [-1,1]. The accumulation buffer pixels are mapped one-to-one with frame buffer pixels.

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

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

The operations are as follows:

`GL_ACCUM` Obtains R, G, B, and A values from the buffer currently selected for reading (see `glReadBuffer`). Each component value is divided by 2^*n*-1, where *n* is the number of bits allocated to each color component in the currently selected buffer. The result is a floating-point value in the range [0,1], which is multiplied by *value* and added to the corresponding pixel component in the accumulation buffer, thereby updating the accumulation buffer.

`GL_LOAD` Similar to `GL_ACCUM`, except that the current value in the accumulation buffer is not used in the calculation of the new value. That is, the R, G, B, and A values from the currently selected buffer are divided by 2^*n*-1,

multiplied by *value*, and then stored in the corresponding accumulation buffer cell, overwriting the current value.

`GL_ADD` Adds *value* to each R, G, B, and A in the accumulation buffer.

`GL_MULT` Multiplies each R, G, B, and A in the accumulation buffer by *value* and returns the scaled component to its corresponding accumulation buffer location.

`GL_RETURN`
Transfers accumulation buffer values to the color buffer or buffers currently selected for writing. Each R, G, B, and A component is multiplied by *value*, then multiplied by 2^*n*-1, clamped to the range [0,2^*n*-1], and stored in the corresponding display buffer cell. The only fragment operations that are applied to this transfer are pixel ownership, scissor, dithering, and color writemasks.

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

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

`GL_INVALID_OPERATION` is generated if there is no accumulation buffer.

`GL_INVALID_OPERATION` is generated if `glAccum` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glActiveTexture` *texture* [Function]
Select active texture unit.

texture Specifies which texture unit to make active. The number of texture units is implementation dependent, but must be at least two. *texture* must be one of `GL_TEXTURE`*i*, where i ranges from 0 to the larger of (`GL_MAX_TEXTURE_COORDS` - 1) and (`GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS` - 1). The initial value is `GL_TEXTURE0`.

`glActiveTexture` selects which texture unit subsequent texture state calls will affect. The number of texture units an implementation supports is implementation dependent, but must be at least 2.

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

`GL_INVALID_ENUM` is generated if *texture* is not one of `GL_TEXTURE`*i*, where i ranges from 0 to the larger of (`GL_MAX_TEXTURE_COORDS` - 1) and (`GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS` - 1).

`void glAlphaFunc` *func ref* [Function]
Specify the alpha test function.

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.

The alpha test discards fragments depending on the outcome of a comparison between an incoming fragment's alpha value and a constant reference value. `glAlphaFunc` specifies the reference value and the comparison function. The comparison is performed only if alpha testing is enabled. By default, it is not enabled. (See `glEnable` and `glDisable` of `GL_ALPHA_TEST`.)

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

`GL_NEVER` Never passes.

`GL_LESS` Passes if the incoming alpha value is less than the reference value.

`GL_EQUAL` Passes if the incoming alpha value is equal to the reference value.

`GL_LEQUAL`
Passes if the incoming alpha value is less than or equal to the reference value.

`GL_GREATER`
Passes if the incoming alpha value is greater than the reference value.

`GL_NOTEQUAL`
Passes if the incoming alpha value is not equal to the reference value.

`GL_GEQUAL`
Passes if the incoming alpha value is greater than or equal to the reference value.

`GL_ALWAYS`
Always passes (initial value).

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

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

`GL_INVALID_OPERATION` is generated if `glAlphaFunc` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`GLboolean glAreTexturesResident` *n textures residences* [Function]
Determine if textures are loaded in texture memory.

n Specifies the number of textures to be queried.

textures Specifies an array containing the names of the textures to be queried.

residences Specifies an array in which the texture residence status is returned. The residence status of a texture named by an element of *textures* is returned in the corresponding element of *residences*.

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

`glAreTexturesResident` queries the texture residence status of the *n* textures named by the elements of *textures*. If all the named textures are resident, `glAreTexturesResident` returns `GL_TRUE`, and the contents of *residences* are undisturbed. If not all the named textures are resident, `glAreTexturesResident` returns `GL_FALSE`, and detailed status is returned in the *n* elements of *residences*. If an element of *residences* is `GL_TRUE`, then the texture named by the corresponding element of *textures* is resident.

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

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

`GL_INVALID_VALUE` is generated if any element in *textures* is 0 or does not name a texture. In that case, the function returns `GL_FALSE` and the contents of *residences* is indeterminate.

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

`void glArrayElement` *i* [Function]

Render a vertex using the specified vertex array element.

i Specifies an index into the enabled vertex data arrays.

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

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

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

`GL_INVALID_VALUE` may be generated if *i* is negative.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to an enabled array and the buffer object's data store is currently mapped.

`void glAttachShader` *program shader* [Function]

Attaches a shader object to a program object.

program Specifies the program object to which a shader object will be attached.

shader Specifies the shader object that is to be attached.

In order to create an executable, there must be a way to specify the list of things that will be linked together. Program objects provide this mechanism. Shaders that are to be linked together in a program object must first be attached to that program object. `glAttachShader` attaches the shader object specified by *shader* to the program object specified by *program*. This indicates that *shader* will be included in link operations that will be performed on *program*.

All operations that can be performed on a shader object are valid whether or not the shader object is attached to a program object. It is permissible to attach a shader object to a program object before source code has been loaded into the shader object or before the shader object has been compiled. It is permissible to attach multiple shader objects of the same type because each may contain a portion of the complete shader. It is also permissible to attach a shader object to more than one program object. If a shader object is deleted while it is attached to a program object, it will be flagged for deletion, and deletion will not occur until `glDetachShader` is called to detach it from all program objects to which it is attached.

`GL_INVALID_VALUE` is generated if either *program* or *shader* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *program* is not a program object.

`GL_INVALID_OPERATION` is generated if *shader* is not a shader object.

`GL_INVALID_OPERATION` is generated if *shader* is already attached to *program*.

`GL_INVALID_OPERATION` is generated if `glAttachShader` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glBeginQuery` *target id* [Function]
`void glEndQuery` *target* [Function]

Delimit the boundaries of a query object.

target Specifies the target type of query object established between `glBeginQuery` and the subsequent `glEndQuery`. The symbolic constant must be `GL_SAMPLES_PASSED`.

id Specifies the name of a query object.

`glBeginQuery` and `glEndQuery` delimit the boundaries of a query object. If a query object with name *id* does not yet exist it is created.

When `glBeginQuery` is executed, the query object's samples-passed counter is reset to 0. Subsequent rendering will increment the counter once for every sample that passes the depth test. When `glEndQuery` is executed, the samples-passed counter is assigned to the query object's result value. This value can be queried by calling `glGetQueryObject` with *pname*`GL_QUERY_RESULT`.

Querying the `GL_QUERY_RESULT` implicitly flushes the GL pipeline until the rendering delimited by the query object has completed and the result is available. `GL_QUERY_RESULT_AVAILABLE` can be queried to determine if the result is immediately available or if the rendering is not yet complete.

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

`GL_INVALID_OPERATION` is generated if `glBeginQuery` is executed while a query object of the same *target* is already active.

`GL_INVALID_OPERATION` is generated if `glEndQuery` is executed when a query object of the same *target* is not active.

`GL_INVALID_OPERATION` is generated if *id* is 0.

`GL_INVALID_OPERATION` is generated if *id* is the name of an already active query object.

`GL_INVALID_OPERATION` is generated if `glBeginQuery` or `glEndQuery` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glBegin` *mode* [Function]
`void glEnd` [Function]

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

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

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

`GL_POINTS`
Treats each vertex as a single point. Vertex n defines point n. N points are drawn.

`GL_LINES` Treats each pair of vertices as an independent line segment. Vertices $2n$-1 and $2n$ define line n. $N/2$ lines are drawn.

`GL_LINE_STRIP`
Draws a connected group of line segments from the first vertex to the last. Vertices n and n+1 define line n. N-1 lines are drawn.

`GL_LINE_LOOP`
Draws a connected group of line segments from the first vertex to the last, then back to the first. Vertices n and n+1 define line n. The last line, however, is defined by vertices N and 1. N lines are drawn.

`GL_TRIANGLES`
Treats each triplet of vertices as an independent triangle. Vertices $3n$-2, $3n$-1, and $3n$ define triangle n. $N/3$ triangles are drawn.

`GL_TRIANGLE_STRIP`
Draws a connected group of triangles. One triangle is defined for each vertex presented after the first two vertices. For odd n, vertices n, n+1, and n+2 define triangle n. For even n, vertices n+1, n, and n+2 define triangle n. N-2 triangles are drawn.

`GL_TRIANGLE_FAN`
: Draws a connected group of triangles. One triangle is defined for each vertex presented after the first two vertices. Vertices 1, *n*+1, and *n*+2 define triangle *n*. *N*-2 triangles are drawn.

`GL_QUADS`
: Treats each group of four vertices as an independent quadrilateral. Vertices 4*n*-3, 4*n*-2, 4*n*-1, and 4*n* define quadrilateral *n*. $N/4$ quadrilaterals are drawn.

`GL_QUAD_STRIP`
: Draws a connected group of quadrilaterals. One quadrilateral is defined for each pair of vertices presented after the first pair. Vertices 2*n*-1, 2*n*, 2*n*+2, and 2*n*+1 define quadrilateral *n*. $N/2$-1 quadrilaterals are drawn. Note that the order in which vertices are used to construct a quadrilateral from strip data is different from that used with independent data.

`GL_POLYGON`
: Draws a single, convex polygon. Vertices 1 through *N* define this polygon.

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

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

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

`GL_INVALID_ENUM` is generated if *mode* is set to an unaccepted value.

`GL_INVALID_OPERATION` is generated if `glBegin` is executed between a `glBegin` and the corresponding execution of `glEnd`.

`GL_INVALID_OPERATION` is generated if `glEnd` is executed without being preceded by a `glBegin`.

`GL_INVALID_OPERATION` is generated if a command other than `glVertex`, `glColor`, `glSecondaryColor`, `glIndex`, `glNormal`, `glFogCoord`, `glTexCoord`, `glMultiTexCoord`, `glVertexAttrib`, `glEvalCoord`, `glEvalPoint`, `glArrayElement`, `glMaterial`, `glEdgeFlag`, `glCallList`, or `glCallLists` is executed between the execution of `glBegin` and the corresponding execution `glEnd`.

Execution of `glEnableClientState`, `glDisableClientState`, `glEdgeFlagPointer`, `glFogCoordPointer`, `glTexCoordPointer`, `glColorPointer`, `glSecondaryColorPointer`,

`glIndexPointer`, `glNormalPointer`, `glVertexPointer`, `glVertexAttribPointer`, `glInterleavedArrays`, or `glPixelStore` is not allowed after a call to `glBegin` and before the corresponding call to `glEnd`, but an error may or may not be generated.

`void glBindAttribLocation` *program index name* [Function]

Associates a generic vertex attribute index with a named attribute variable.

program Specifies the handle of the program object in which the association is to be made.

index Specifies the index of the generic vertex attribute to be bound.

name Specifies a null terminated string containing the name of the vertex shader attribute variable to which *index* is to be bound.

`glBindAttribLocation` is used to associate a user-defined attribute variable in the program object specified by *program* with a generic vertex attribute index. The name of the user-defined attribute variable is passed as a null terminated string in *name*. The generic vertex attribute index to be bound to this variable is specified by *index*. When *program* is made part of current state, values provided via the generic vertex attribute *index* will modify the value of the user-defined attribute variable specified by *name*.

If *name* refers to a matrix attribute variable, *index* refers to the first column of the matrix. Other matrix columns are then automatically bound to locations *index+1* for a matrix of type mat2; *index+1* and *index+2* for a matrix of type mat3; and *index+1*, *index+2*, and *index+3* for a matrix of type mat4.

This command makes it possible for vertex shaders to use descriptive names for attribute variables rather than generic variables that are numbered from 0 to `GL_MAX_VERTEX_ATTRIBS` -1. The values sent to each generic attribute index are part of current state, just like standard vertex attributes such as color, normal, and vertex position. If a different program object is made current by calling `glUseProgram`, the generic vertex attributes are tracked in such a way that the same values will be observed by attributes in the new program object that are also bound to *index*.

Attribute variable name-to-generic attribute index bindings for a program object can be explicitly assigned at any time by calling `glBindAttribLocation`. Attribute bindings do not go into effect until `glLinkProgram` is called. After a program object has been linked successfully, the index values for generic attributes remain fixed (and their values can be queried) until the next link command occurs.

Applications are not allowed to bind any of the standard OpenGL vertex attributes using this command, as they are bound automatically when needed. Any attribute binding that occurs after the program object has been linked will not take effect until the next time the program object is linked.

`GL_INVALID_VALUE` is generated if *index* is greater than or equal to `GL_MAX_VERTEX_ATTRIBS`.

`GL_INVALID_OPERATION` is generated if *name* starts with the reserved prefix "gl_".

`GL_INVALID_VALUE` is generated if *program* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *program* is not a program object.

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

`void glBindBuffer` *target buffer* [Function]

Bind a named buffer object.

target Specifies the target to which the buffer object is bound. The symbolic constant must be GL_ARRAY_BUFFER, GL_ELEMENT_ARRAY_BUFFER, GL_PIXEL_PACK_BUFFER, or GL_PIXEL_UNPACK_BUFFER.

buffer Specifies the name of a buffer object.

glBindBuffer lets you create or use a named buffer object. Calling glBindBuffer with *target* set to GL_ARRAY_BUFFER, GL_ELEMENT_ARRAY_BUFFER, GL_PIXEL_PACK_BUFFER or GL_PIXEL_UNPACK_BUFFER and *buffer* set to the name of the new buffer object binds the buffer object name to the target. When a buffer object is bound to a target, the previous binding for that target is automatically broken.

Buffer object names are unsigned integers. The value zero is reserved, but there is no default buffer object for each buffer object target. Instead, *buffer* set to zero effectively unbinds any buffer object previously bound, and restores client memory usage for that buffer object target. Buffer object names and the corresponding buffer object contents are local to the shared display-list space (see glXCreateContext) of the current GL rendering context; two rendering contexts share buffer object names only if they also share display lists.

You may use glGenBuffers to generate a set of new buffer object names.

The state of a buffer object immediately after it is first bound is an unmapped zero-sized memory buffer with GL_READ_WRITE access and GL_STATIC_DRAW usage.

While a non-zero buffer object name is bound, GL operations on the target to which it is bound affect the bound buffer object, and queries of the target to which it is bound return state from the bound buffer object. While buffer object name zero is bound, as in the initial state, attempts to modify or query state on the target to which it is bound generates an GL_INVALID_OPERATION error.

When vertex array pointer state is changed, for example by a call to glNormalPointer, the current buffer object binding (GL_ARRAY_BUFFER_BINDING) is copied into the corresponding client state for the vertex array type being changed, for example GL_NORMAL_ARRAY_BUFFER_BINDING. While a non-zero buffer object is bound to the GL_ARRAY_BUFFER target, the vertex array pointer parameter that is traditionally interpreted as a pointer to client-side memory is instead interpreted as an offset within the buffer object measured in basic machine units.

While a non-zero buffer object is bound to the GL_ELEMENT_ARRAY_BUFFER target, the indices parameter of glDrawElements, glDrawRangeElements, or glMultiDrawElements that is traditionally interpreted as a pointer to client-side memory is instead interpreted as an offset within the buffer object measured in basic machine units.

While a non-zero buffer object is bound to the GL_PIXEL_PACK_BUFFER target, the following commands are affected: glGetCompressedTexImage, glGetConvolutionFilter, glGetHistogram, glGetMinmax, glGetPixelMap,

`glGetPolygonStipple`, `glGetSeparableFilter`, `glGetTexImage`, and `glReadPixels`. The pointer parameter that is traditionally interpreted as a pointer to client-side memory where the pixels are to be packed is instead interpreted as an offset within the buffer object measured in basic machine units.

While a non-zero buffer object is bound to the `GL_PIXEL_UNPACK_BUFFER` target, the following commands are affected: `glBitmap`, `glColorSubTable`, `glColorTable`, `glCompressedTexImage1D`, `glCompressedTexImage2D`, `glCompressedTexImage3D`, `glCompressedTexSubImage1D`, `glCompressedTexSubImage2D`, `glCompressedTexSubImage3D`, `glConvolutionFilter1D`, `glConvolutionFilter2D`, `glDrawPixels`, `glPixelMap`, `glPolygonStipple`, `glSeparableFilter2D`, `glTexImage1D`, `glTexImage2D`, `glTexImage3D`, `glTexSubImage1D`, `glTexSubImage2D`, and `glTexSubImage3D`. The pointer parameter that is traditionally interpreted as a pointer to client-side memory from which the pixels are to be unpacked is instead interpreted as an offset within the buffer object measured in basic machine units.

A buffer object binding created with `glBindBuffer` remains active until a different buffer object name is bound to the same target, or until the bound buffer object is deleted with `glDeleteBuffers`.

Once created, a named buffer object may be re-bound to any target as often as needed. However, the GL implementation may make choices about how to optimize the storage of a buffer object based on its initial binding target.

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

`GL_INVALID_OPERATION` is generated if `glBindBuffer` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glBindTexture` *target texture* [Function]

Bind a named texture to a texturing target.

target Specifies the target to which the texture is bound. Must be either `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, `GL_TEXTURE_3D`, or `GL_TEXTURE_CUBE_MAP`.

texture Specifies the name of a texture.

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

Texture names are unsigned integers. The value zero is reserved to represent the default texture for each texture target. Texture names and the corresponding texture contents are local to the shared display-list space (see `glXCreateContext`) of the current GL rendering context; two rendering contexts share texture names only if they also share display lists.

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

When a texture is first bound, it assumes the specified target: A texture first bound to `GL_TEXTURE_1D` becomes one-dimensional texture, a texture first bound to `GL_TEXTURE_2D` becomes two-dimensional texture, a texture first bound to `GL_TEXTURE_`

3D becomes three-dimensional texture, and a texture first bound to GL_TEXTURE_CUBE_MAP becomes a cube-mapped texture. The state of a one-dimensional texture immediately after it is first bound is equivalent to the state of the default GL_TEXTURE_1D at GL initialization, and similarly for two- and three-dimensional textures and cube-mapped textures.

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

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

Once created, a named texture may be re-bound to its same original target as often as needed. It is usually much faster to use glBindTexture to bind an existing named texture to one of the texture targets than it is to reload the texture image using glTexImage1D, glTexImage2D, or glTexImage3D. For additional control over performance, use glPrioritizeTextures.

glBindTexture is included in display lists.

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

GL_INVALID_OPERATION is generated if *texture* was previously created with a target that doesn't match that of *target*.

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

`void glBitmap` *width height xorig yorig xmove ymove bitmap* [Function]

Draw a bitmap.

width
height Specify the pixel width and height of the bitmap image.

xorig
yorig Specify the location of the origin in the bitmap image. The origin is measured from the lower left corner of the bitmap, with right and up being the positive axes.

xmove
ymove Specify the x and y offsets to be added to the current raster position after the bitmap is drawn.

bitmap Specifies the address of the bitmap image.

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

glBitmap takes seven arguments. The first pair specifies the width and height of the bitmap image. The second pair specifies the location of the bitmap origin relative

to the lower left corner of the bitmap image. The third pair of arguments specifies x and y offsets to be added to the current raster position after the bitmap has been drawn. The final argument is a pointer to the bitmap image itself.

If a non-zero named buffer object is bound to the `GL_PIXEL_UNPACK_BUFFER` target (see `glBindBuffer`) while a bitmap image is specified, *bitmap* is treated as a byte offset into the buffer object's data store.

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

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

$x\_w=x\_r-x\_o$,

$y\_w=y\_r-y\_o$,

where (x_r,y_r) is the raster position and (x_o,y_o) is the bitmap origin. Fragments are then generated for each pixel corresponding to a 1 (one) in the bitmap image. These fragments are generated using the current raster z coordinate, color or color index, and current raster texture coordinates. They are then treated just as if they had been generated by a point, line, or polygon, including texture mapping, fogging, and all per-fragment operations such as alpha and depth testing.

After the bitmap has been drawn, the x and y coordinates of the current raster position are offset by *xmove* and *ymove*. No change is made to the z coordinate of the current raster position, or to the current raster color, texture coordinates, or index.

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

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the buffer object's data store is currently mapped.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

`GL_INVALID_OPERATION` is generated if `glBitmap` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glBlendColor` *red green blue alpha* [Function]

Set the blend color.

red
green
blue
alpha specify the components of `GL_BLEND_COLOR`

The `GL_BLEND_COLOR` may be used to calculate the source and destination blending factors. The color components are clamped to the range [0,1] before being stored.

See `glBlendFunc` for a complete description of the blending operations. Initially the `GL_BLEND_COLOR` is set to (0, 0, 0, 0).

`GL_INVALID_OPERATION` is generated if `glBlendColor` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glBlendEquationSeparate` *modeRGB modeAlpha* [Function]

Set the RGB blend equation and the alpha blend equation separately.

modeRGB
: specifies the RGB blend equation, how the red, green, and blue components of the source and destination colors are combined. It must be `GL_FUNC_ADD`, `GL_FUNC_SUBTRACT`, `GL_FUNC_REVERSE_SUBTRACT`, `GL_MIN`, `GL_MAX`.

modeAlpha
: specifies the alpha blend equation, how the alpha component of the source and destination colors are combined. It must be `GL_FUNC_ADD`, `GL_FUNC_SUBTRACT`, `GL_FUNC_REVERSE_SUBTRACT`, `GL_MIN`, `GL_MAX`.

The blend equations determines how a new pixel (the "source" color) is combined with a pixel already in the framebuffer (the "destination" color). This function specifies one blend equation for the RGB-color components and one blend equation for the alpha component.

The blend equations use the source and destination blend factors specified by either `glBlendFunc` or `glBlendFuncSeparate`. See `glBlendFunc` or `glBlendFuncSeparate` for a description of the various blend factors.

In the equations that follow, 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$), respectively. The result color is referred to as (R_r,$G_r$$B_r$$A_r$). The source and destination blend factors are denoted (s_R,$s_G$$s_B$$s_A$) and ($d_R$,$d_G$$d_B$$d_A$), respectively. For these equations all color components are understood to have values in the range [0,1].

Mode **RGB Components**, **Alpha Component**

`GL_FUNC_ADD`
: $R_r = R_s * R_d$ $G_r = G_s * G_d$ $B_r = B_s * B_d$,
$A_r = A_s * A_d$

`GL_FUNC_SUBTRACT`
: $R_r = R_s * R_d$ $G_r = G_s * G_d$ $B_r = B_s * B_d$,
$A_r = A_s * A_d$

`GL_FUNC_REVERSE_SUBTRACT`
: $R_r = R_d * R_s$ $G_r = G_d * G_s$ $B_r = B_d * B_s$,
$A_r = A_d * A_s$

`GL_MIN` $R_r = min(R_s,R_d)$ $G_r = min(G_s,G_d)$ $B_r = min(B_s,B_d)$,
$A_r = min(A_s,A_d)$

`GL_MAX` $R_r = max(R_s,R_d)$ $G_r = max(G_s,G_d)$ $B_r = max(B_s,B_d)$,
$A_r = max(A_s,A_d)$

The results of these equations are clamped to the range [0,1].

The `GL_MIN` and `GL_MAX` equations are useful for applications that analyze image data (image thresholding against a constant color, for example). The `GL_FUNC_ADD` equation is useful for antialiasing and transparency, among other things.

Initially, both the RGB blend equation and the alpha blend equation are set to `GL_FUNC_ADD`.

`GL_INVALID_ENUM` is generated if either *modeRGB* or *modeAlpha* is not one of `GL_FUNC_ADD`, `GL_FUNC_SUBTRACT`, `GL_FUNC_REVERSE_SUBTRACT`, `GL_MAX`, or `GL_MIN`.

`GL_INVALID_OPERATION` is generated if `glBlendEquationSeparate` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glBlendEquation` *mode* [Function]

Specify the equation used for both the RGB blend equation and the Alpha blend equation.

mode specifies how source and destination colors are combined. It must be `GL_FUNC_ADD`, `GL_FUNC_SUBTRACT`, `GL_FUNC_REVERSE_SUBTRACT`, `GL_MIN`, `GL_MAX`.

The blend equations determine how a new pixel (the "source" color) is combined with a pixel already in the framebuffer (the "destination" color). This function sets both the RGB blend equation and the alpha blend equation to a single equation.

These equations use the source and destination blend factors specified by either `glBlendFunc` or `glBlendFuncSeparate`. See `glBlendFunc` or `glBlendFuncSeparate` for a description of the various blend factors.

In the equations that follow, 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$), respectively. The result color is referred to as (R_r,$G_r$$B_r$$A_r$). The source and destination blend factors are denoted (s_R,$s_G$$s_B$$s_A$) and ($d_R$,$d_G$$d_B$$d_A$), respectively. For these equations all color components are understood to have values in the range [0,1].

Mode **RGB Components**, **Alpha Component**

`GL_FUNC_ADD`
: $Rr = R_s R + R_d R$ $Gr = G_s G + G_d G$ $Br = B_s B + B_d B$,
$Ar = A_s A + A_d A$

`GL_FUNC_SUBTRACT`
: $Rr = R_s R - R_d R$ $Gr = G_s G - G_d G$ $Br = B_s B - B_d B$,
$Ar = A_s A - A_d A$

`GL_FUNC_REVERSE_SUBTRACT`
: $Rr = R_d R - R_s R$ $Gr = G_d G - G_s G$ $Br = B_d B - B_s B$,
$Ar = A_d A - A_s A$

`GL_MIN` $Rr = min(R_s,R_d)$ $Gr = min(G_s,G_d)$ $Br = min(B_s,B_d)$,
$Ar = min(A_s,A_d)$

`GL_MAX` $Rr = max(R_s,R_d)$ $Gr = max(G_s,G_d)$ $Br = max(B_s,B_d)$,
$Ar = max(A_s,A_d)$

The results of these equations are clamped to the range [0,1].

The `GL_MIN` and `GL_MAX` equations are useful for applications that analyze image data (image thresholding against a constant color, for example). The `GL_FUNC_ADD` equation is useful for antialiasing and transparency, among other things.

Initially, both the RGB blend equation and the alpha blend equation are set to `GL_FUNC_ADD`.

`GL_INVALID_ENUM` is generated if *mode* is not one of `GL_FUNC_ADD`, `GL_FUNC_SUBTRACT`, `GL_FUNC_REVERSE_SUBTRACT`, `GL_MAX`, or `GL_MIN`.

`GL_INVALID_OPERATION` is generated if `glBlendEquation` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glBlendFuncSeparate` *srcRGB dstRGB srcAlpha dstAlpha* [Function]

Specify pixel arithmetic for RGB and alpha components separately.

srcRGB Specifies how the red, green, and blue blending factors are computed. The following symbolic constants are accepted: `GL_ZERO`, `GL_ONE`, `GL_SRC_COLOR`, `GL_ONE_MINUS_SRC_COLOR`, `GL_DST_COLOR`, `GL_ONE_MINUS_DST_COLOR`, `GL_SRC_ALPHA`, `GL_ONE_MINUS_SRC_ALPHA`, `GL_DST_ALPHA`, `GL_ONE_MINUS_DST_ALPHA`, `GL_CONSTANT_COLOR`, `GL_ONE_MINUS_CONSTANT_COLOR`, `GL_CONSTANT_ALPHA`, `GL_ONE_MINUS_CONSTANT_ALPHA`, and `GL_SRC_ALPHA_SATURATE`. The initial value is `GL_ONE`.

dstRGB Specifies how the red, green, and blue destination blending factors are computed. The following symbolic constants are accepted: `GL_ZERO`, `GL_ONE`, `GL_SRC_COLOR`, `GL_ONE_MINUS_SRC_COLOR`, `GL_DST_COLOR`, `GL_ONE_MINUS_DST_COLOR`, `GL_SRC_ALPHA`, `GL_ONE_MINUS_SRC_ALPHA`, `GL_DST_ALPHA`, `GL_ONE_MINUS_DST_ALPHA`. `GL_CONSTANT_COLOR`, `GL_ONE_MINUS_CONSTANT_COLOR`, `GL_CONSTANT_ALPHA`, and `GL_ONE_MINUS_CONSTANT_ALPHA`. The initial value is `GL_ZERO`.

srcAlpha Specified how the alpha source blending factor is computed. The same symbolic constants are accepted as for *srcRGB*. The initial value is `GL_ONE`.

dstAlpha Specified how the alpha destination blending factor is computed. The same symbolic constants are accepted as for *dstRGB*. The initial value is `GL_ZERO`.

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

`glBlendFuncSeparate` defines the operation of blending when it is enabled. *srcRGB* specifies which method is used to scale the source RGB-color components. *dstRGB* specifies which method is used to scale the destination RGB-color components. Likewise, *srcAlpha* specifies which method is used to scale the source alpha color component, and *dstAlpha* specifies which method is used to scale the destination alpha component. The 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,$G\_sB\_sA\_s$) and (R_d,$G\_dB\_dA\_d$). The color specified by `glBlendColor` is referred to as (R_c,$G\_cB\_cA\_c$). They are understood to have integer values between 0 and (k_R,$k\_Gk\_Bk\_A$), where

k_c=2^m_c,-1

and (m_R,$m\_Gm\_Bm\_A$) is the number of red, green, blue, and alpha bitplanes.

Source and destination scale factors are referred to as (s_R,$s\_Gs\_Bs\_A$) and (d_R,$d\_Gd\_Bd\_A$). All scale factors have range [0,1].

Parameter
: **RGB Factor**, **Alpha Factor**

`GL_ZERO`
: (0,00), 0

`GL_ONE`
: (1,11), 1

`GL_SRC_COLOR`
: (R_s/k_R,$G\_s/k\_GB\_s/k\_B$), A_s/k_A

`GL_ONE_MINUS_SRC_COLOR`
: (1,111)-(R_s/k_R,$G\_s/k\_GB\_s/k\_B$), 1-A_s/k_A

`GL_DST_COLOR`
: (R_d/k_R,$G\_d/k\_GB\_d/k\_B$), A_d/k_A

`GL_ONE_MINUS_DST_COLOR`
: (1,11)-(R_d/k_R,$G\_d/k\_GB\_d/k\_B$), 1-A_d/k_A

`GL_SRC_ALPHA`
: (A_s/k_A,$A\_s/k\_AA\_s/k\_A$), A_s/k_A

`GL_ONE_MINUS_SRC_ALPHA`
: (1,11)-(A_s/k_A,$A\_s/k\_AA\_s/k\_A$), 1-A_s/k_A

`GL_DST_ALPHA`
: (A_d/k_A,$A\_d/k\_AA\_d/k\_A$), A_d/k_A

`GL_ONE_MINUS_DST_ALPHA`
: (1,11)-(A_d/k_A,$A\_d/k\_AA\_d/k\_A$), 1-A_d/k_A

`GL_CONSTANT_COLOR`
: (R_c,G_cB_c), A_c

`GL_ONE_MINUS_CONSTANT_COLOR`
: (1,11)-(R_c,G_cB_c), 1-A_c

`GL_CONSTANT_ALPHA`
: (A_c,A_cA_c), A_c

`GL_ONE_MINUS_CONSTANT_ALPHA`
: (1,11)-(A_c,A_cA_c), 1-A_c

`GL_SRC_ALPHA_SATURATE`
: (i,ii), 1

In the table,

$i=min(A\_s,1\text{-}A\_d,)$

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

$R\_d=min(k\_R,R\_ss\_R+R\_dd\_R)G\_d=min(k\_G,G\_ss\_G+G\_dd\_G)B\_d=min(k\_B,B\_ss\_B+B\_dd\_B)A\_d$

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 *srcRGB* is `GL_SRC_ALPHA`, *dstRGB* is `GL_ONE_MINUS_SRC_ALPHA`, and A_s is equal to k_A, the equations reduce to simple replacement:

$R\_d=R\_sG\_d=G\_sB\_d=B\_sA\_d=A\_s$

`GL_INVALID_ENUM` is generated if either *srcRGB* or *dstRGB* is not an accepted value.

`GL_INVALID_OPERATION` is generated if `glBlendFuncSeparate` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glBlendFunc` *sfactor dfactor* [Function]

Specify pixel arithmetic.

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_SRC_COLOR`, `GL_ONE_MINUS_SRC_COLOR`, `GL_DST_COLOR`, `GL_ONE_MINUS_DST_COLOR`, `GL_SRC_ALPHA`, `GL_ONE_MINUS_SRC_ALPHA`, `GL_DST_ALPHA`, `GL_ONE_MINUS_DST_ALPHA`, `GL_CONSTANT_COLOR`, `GL_ONE_MINUS_CONSTANT_COLOR`, `GL_CONSTANT_ALPHA`, `GL_ONE_MINUS_CONSTANT_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. The following symbolic constants are accepted: `GL_ZERO`, `GL_ONE`, `GL_SRC_COLOR`, `GL_ONE_MINUS_SRC_COLOR`, `GL_DST_COLOR`, `GL_ONE_MINUS_DST_COLOR`, `GL_SRC_ALPHA`, `GL_ONE_MINUS_SRC_ALPHA`, `GL_DST_ALPHA`, `GL_ONE_MINUS_DST_ALPHA`. `GL_CONSTANT_COLOR`, `GL_ONE_MINUS_CONSTANT_COLOR`, `GL_CONSTANT_ALPHA`, and `GL_ONE_MINUS_CONSTANT_ALPHA`. The initial value is `GL_ZERO`.

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

`glBlendFunc` defines the operation of blending when it is enabled. *sfactor* specifies which method is used to scale the source color components. *dfactor* specifies which method is used to scale the destination color components. The 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,$G\_sB\_sA\_s$) and (R_d,$G\_dB\_dA\_d$).

The color specified by `glBlendColor` is referred to as (R_c,$G\_cB\_cA\_c$). They are understood to have integer values between 0 and (k_R,$k\_Gk\_Bk\_A$), where

k_c=2^m_c,-1

and (m_R,$m\_Gm\_Bm\_A$) is the number of red, green, blue, and alpha bitplanes.

Source and destination scale factors are referred to as (s_R,$s\_Gs\_Bs\_A$) and (d_R,$d\_Gd\_Bd\_A$). The scale factors described in the table, denoted (f_R,$f\_Gf\_Bf\_A$), represent either source or destination factors. All scale factors have range [0,1].

Parameter

(f_R,$f\_Gf\_Bf\_A$)

`GL_ZERO` (0,000)

`GL_ONE` (1,111)

`GL_SRC_COLOR`
(R_s/k_R,$G\_s/k\_GB\_s/k\_BA\_s/k\_A$)

`GL_ONE_MINUS_SRC_COLOR`
(1,111)-(R_s/k_R,$G\_s/k\_GB\_s/k\_BA\_s/k\_A$)

`GL_DST_COLOR`
(R_d/k_R,$G\_d/k\_GB\_d/k\_BA\_d/k\_A$)

`GL_ONE_MINUS_DST_COLOR`
(1,111)-(R_d/k_R,$G\_d/k\_GB\_d/k\_BA\_d/k\_A$)

`GL_SRC_ALPHA`
(A_s/k_A,$A\_s/k\_AA\_s/k\_AA\_s/k\_A$)

`GL_ONE_MINUS_SRC_ALPHA`
(1,111)-(A_s/k_A,$A\_s/k\_AA\_s/k\_AA\_s/k\_A$)

`GL_DST_ALPHA`
(A_d/k_A,$A\_d/k\_AA\_d/k\_AA\_d/k\_A$)

`GL_ONE_MINUS_DST_ALPHA`
(1,111)-(A_d/k_A,$A\_d/k\_AA\_d/k\_AA\_d/k\_A$)

`GL_CONSTANT_COLOR`
(R_c,$G\_cB\_cA\_c$)

`GL_ONE_MINUS_CONSTANT_COLOR`
(1,111)-(R_c,$G\_cB\_cA\_c$)

`GL_CONSTANT_ALPHA`
(A_c,$A\_cA\_cA\_c$)

`GL_ONE_MINUS_CONSTANT_ALPHA`
(1,111)-(A_c,$A\_cA\_cA\_c$)

`GL_SRC_ALPHA_SATURATE`
(i,ii1)

In the table,

$i=min(A\_s,k\_A\text{-}A\_d)/k\_A$

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

$R\_d=min(k\_R,R\_ss\_R+R\_dd\_R)G\_d=min(k\_G,G\_ss\_G+G\_dd\_G)B\_d=min(k\_B,B\_ss\_B+B\_dd\_B)A\_d$

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

$R\_d=R\_sG\_d=G\_sB\_d=B\_sA\_d=A\_s$

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

`GL_INVALID_OPERATION` is generated if `glBlendFunc` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glBufferData` *target size data usage* [Function]

Creates and initializes a buffer object's data store.

target Specifies the target buffer object. The symbolic constant must be `GL_ARRAY_BUFFER`, `GL_ELEMENT_ARRAY_BUFFER`, `GL_PIXEL_PACK_BUFFER`, or `GL_PIXEL_UNPACK_BUFFER`.

size Specifies the size in bytes of the buffer object's new data store.

data Specifies a pointer to data that will be copied into the data store for initialization, or `NULL` if no data is to be copied.

usage Specifies the expected usage pattern of the data store. The symbolic constant must be `GL_STREAM_DRAW`, `GL_STREAM_READ`, `GL_STREAM_COPY`, `GL_STATIC_DRAW`, `GL_STATIC_READ`, `GL_STATIC_COPY`, `GL_DYNAMIC_DRAW`, `GL_DYNAMIC_READ`, or `GL_DYNAMIC_COPY`.

`glBufferData` creates a new data store for the buffer object currently bound to *target*. Any pre-existing data store is deleted. The new data store is created with the specified *size* in bytes and *usage*. If *data* is not `NULL`, the data store is initialized with data from this pointer. In its initial state, the new data store is not mapped, it has a `NULL` mapped pointer, and its mapped access is `GL_READ_WRITE`.

usage is a hint to the GL implementation as to how a buffer object's data store will be accessed. This enables the GL implementation to make more intelligent decisions that may significantly impact buffer object performance. It does not, however, constrain the actual usage of the data store. *usage* can be broken down into two parts: first, the frequency of access (modification and usage), and second, the nature of that access. The frequency of access may be one of these:

STREAM The data store contents will be modified once and used at most a few times.

STATIC The data store contents will be modified once and used many times.

DYNAMIC
: The data store contents will be modified repeatedly and used many times.

The nature of access may be one of these:

DRAW
: The data store contents are modified by the application, and used as the source for GL drawing and image specification commands.

READ
: The data store contents are modified by reading data from the GL, and used to return that data when queried by the application.

COPY
: The data store contents are modified by reading data from the GL, and used as the source for GL drawing and image specification commands.

`GL_INVALID_ENUM` is generated if *target* is not `GL_ARRAY_BUFFER`, `GL_ELEMENT_ARRAY_BUFFER`, `GL_PIXEL_PACK_BUFFER`, or `GL_PIXEL_UNPACK_BUFFER`.

`GL_INVALID_ENUM` is generated if *usage* is not `GL_STREAM_DRAW`, `GL_STREAM_READ`, `GL_STREAM_COPY`, `GL_STATIC_DRAW`, `GL_STATIC_READ`, `GL_STATIC_COPY`, `GL_DYNAMIC_DRAW`, `GL_DYNAMIC_READ`, or `GL_DYNAMIC_COPY`.

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

`GL_INVALID_OPERATION` is generated if the reserved buffer object name 0 is bound to *target*.

`GL_OUT_OF_MEMORY` is generated if the GL is unable to create a data store with the specified *size*.

`GL_INVALID_OPERATION` is generated if `glBufferData` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glBufferSubData` *target offset size data* [Function]

Updates a subset of a buffer object's data store.

target
: Specifies the target buffer object. The symbolic constant must be `GL_ARRAY_BUFFER`, `GL_ELEMENT_ARRAY_BUFFER`, `GL_PIXEL_PACK_BUFFER`, or `GL_PIXEL_UNPACK_BUFFER`.

offset
: Specifies the offset into the buffer object's data store where data replacement will begin, measured in bytes.

size
: Specifies the size in bytes of the data store region being replaced.

data
: Specifies a pointer to the new data that will be copied into the data store.

`glBufferSubData` redefines some or all of the data store for the buffer object currently bound to *target*. Data starting at byte offset *offset* and extending for *size* bytes is copied to the data store from the memory pointed to by *data*. An error is thrown if *offset* and *size* together define a range beyond the bounds of the buffer object's data store.

`GL_INVALID_ENUM` is generated if *target* is not `GL_ARRAY_BUFFER`, `GL_ELEMENT_ARRAY_BUFFER`, `GL_PIXEL_PACK_BUFFER`, or `GL_PIXEL_UNPACK_BUFFER`.

`GL_INVALID_VALUE` is generated if *offset* or *size* is negative, or if together they define a region of memory that extends beyond the buffer object's allocated data store.

`GL_INVALID_OPERATION` is generated if the reserved buffer object name 0 is bound to *target*.

`GL_INVALID_OPERATION` is generated if the buffer object being updated is mapped.

`GL_INVALID_OPERATION` is generated if `glBufferSubData` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glCallLists` *n type lists* [Function]

Execute a list of display lists.

n Specifies the number of display lists to be executed.

type Specifies the type of values in *lists*. Symbolic constants `GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_UNSIGNED_SHORT`, `GL_INT`, `GL_UNSIGNED_INT`, `GL_FLOAT`, `GL_2_BYTES`, `GL_3_BYTES`, and `GL_4_BYTES` are accepted.

lists Specifies the address of an array of name offsets in the display list. The pointer type is void because the offsets can be bytes, shorts, ints, or floats, depending on the value of *type*.

`glCallLists` causes each display list in the list of names passed as *lists* to be executed. As a result, the commands saved in each display list are executed in order, just as if they were called without using a display list. Names of display lists that have not been defined are ignored.

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

`GL_BYTE` *lists* is treated as an array of signed bytes, each in the range -128 through 127.

`GL_UNSIGNED_BYTE`
lists is treated as an array of unsigned bytes, each in the range 0 through 255.

`GL_SHORT` *lists* is treated as an array of signed two-byte integers, each in the range -32768 through 32767.

`GL_UNSIGNED_SHORT`
lists is treated as an array of unsigned two-byte integers, each in the range 0 through 65535.

`GL_INT` *lists* is treated as an array of signed four-byte integers.

`GL_UNSIGNED_INT`
lists is treated as an array of unsigned four-byte integers.

`GL_FLOAT` *lists* is treated as an array of four-byte floating-point values.

`GL_2_BYTES`
lists is treated as an array of unsigned bytes. Each pair of bytes specifies a single display-list name. The value of the pair is computed as 256 times the unsigned value of the first byte plus the unsigned value of the second byte.

`GL_3_BYTES`
: *lists* is treated as an array of unsigned bytes. Each triplet of bytes specifies a single display-list name. The value of the triplet is computed as 65536 times the unsigned value of the first byte, plus 256 times the unsigned value of the second byte, plus the unsigned value of the third byte.

`GL_4_BYTES`
: *lists* is treated as an array of unsigned bytes. Each quadruplet of bytes specifies a single display-list name. The value of the quadruplet is computed as 16777216 times the unsigned value of the first byte, plus 65536 times the unsigned value of the second byte, plus 256 times the unsigned value of the third byte, plus the unsigned value of the fourth byte.

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

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

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

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

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

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

`void glCallList` *list* [Function]

Execute a display list.

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

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

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

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

`void glClearAccum` *red green blue alpha* [Function]

Specify clear values for the accumulation buffer.

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

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

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

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

`void glClearColor` *red green blue alpha* [Function]

Specify clear values for the color buffers.

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

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

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

`void glClearDepth` *depth* [Function]

Specify the clear value for the depth buffer.

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

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

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

`void glClearIndex` *c* [Function]

Specify the clear value for the color index buffers.

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

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

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

`void glClearStencil` *s* [Function]

Specify the clear value for the stencil buffer.

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

`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.

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

`void glClear` *mask* [Function]

Clear buffers to preset values.

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

`glClear` sets the bitplane area of the window to values previously selected by `glClearColor`, `glClearIndex`, `glClearDepth`, `glClearStencil`, and `glClearAccum`. Multiple color buffers can be cleared simultaneously by selecting more than one buffer at a time using `glDrawBuffer`.

The pixel ownership test, the scissor test, dithering, and the buffer writemasks affect the operation of `glClear`. The scissor box bounds the cleared region. Alpha function, blend function, logical operation, stenciling, texture mapping, and depth-buffering are ignored by `glClear`.

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

The values are as follows:

`GL_COLOR_BUFFER_BIT`
Indicates the buffers currently enabled for color writing.

`GL_DEPTH_BUFFER_BIT`
Indicates the depth buffer.

`GL_ACCUM_BUFFER_BIT`
Indicates the accumulation buffer.

`GL_STENCIL_BUFFER_BIT`
Indicates the stencil buffer.

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

`GL_INVALID_VALUE` is generated if any bit other than the four defined bits is set in *mask*.

`GL_INVALID_OPERATION` is generated if `glClear` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glClientActiveTexture` *texture* [Function]

Select active texture unit.

texture Specifies which texture unit to make active. The number of texture units is implementation dependent, but must be at least two. *texture* must be one of `GL_TEXTURE`*i*, where i ranges from 0 to the value of `GL_MAX_TEXTURE_COORDS` - 1, which is an implementation-dependent value. The initial value is `GL_TEXTURE0`.

`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`.

`GL_INVALID_ENUM` is generated if *texture* is not one of `GL_TEXTURE`*i*, where i ranges from 0 to the value of `GL_MAX_TEXTURE_COORDS` - 1.

`void glClipPlane` *plane equation* [Function]

Specify a plane against which all geometry is clipped.

plane Specifies which clipping plane is being positioned. Symbolic names of the form `GL_CLIP_PLANE`*i*, where *i* is an integer between 0 and `GL_MAX_CLIP_PLANES`-1, are accepted.

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

Geometry is always clipped against the boundaries of a six-plane frustum in *x*, *y*, and *z*. `glClipPlane` allows the specification of additional planes, not necessarily perpendicular to the *x*, *y*, or *z* axis, against which all geometry is clipped. To determine the maximum number of additional clipping planes, call `glGetIntegerv` with argument `GL_MAX_CLIP_PLANES`. All implementations support at least six such clipping planes. Because the resulting clipping region is the intersection of the defined half-spaces, it is always convex.

`glClipPlane` specifies a half-space using a four-component plane equation. When `glClipPlane` is called, *equation* is transformed by the inverse of the modelview matrix and stored in the resulting eye coordinates. Subsequent changes to the modelview matrix have no effect on the stored plane-equation components. If the dot product of the eye coordinates of a vertex with the stored plane equation components is positive or zero, the vertex is *in* with respect to that clipping plane. Otherwise, it is *out*.

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

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

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

`GL_INVALID_OPERATION` is generated if `glClipPlane` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glColorMask` *red green blue alpha* [Function]

Enable and disable writing of frame buffer color components.

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

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

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

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

`void glColorMaterial` *face mode* [Function]
Cause a material color to track the current color.

face Specifies whether front, back, or both front and back material parameters should track the current color. Accepted values are `GL_FRONT`, `GL_BACK`, and `GL_FRONT_AND_BACK`. The initial value is `GL_FRONT_AND_BACK`.

mode Specifies which of several material parameters track the current color. Accepted values are `GL_EMISSION`, `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`, and `GL_AMBIENT_AND_DIFFUSE`. The initial value is `GL_AMBIENT_AND_DIFFUSE`.

`glColorMaterial` specifies which material parameters track the current color. When `GL_COLOR_MATERIAL` is enabled, the material parameter or parameters specified by *mode*, of the material or materials specified by *face*, track the current color at all times.

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

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

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

`void glColorPointer` *size type stride pointer* [Function]
Define an array of colors.

size Specifies the number of components per color. Must be 3 or 4. The initial value is 4.

type Specifies the data type of each color component in the array. Symbolic constants `GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_UNSIGNED_SHORT`, `GL_INT`, `GL_UNSIGNED_INT`, `GL_FLOAT`, and `GL_DOUBLE` are accepted. The initial value is `GL_FLOAT`.

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. The initial value is 0.

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

If a non-zero named buffer object is bound to the `GL_ARRAY_BUFFER` target (see `glBindBuffer`) while a color array is specified, *pointer* is treated as a byte offset into the buffer object's data store. Also, the buffer object binding (`GL_ARRAY_BUFFER_BINDING`) is saved as color vertex array client-side state (`GL_COLOR_ARRAY_BUFFER_BINDING`).

When a color array is specified, *size*, *type*, *stride*, and *pointer* are saved as client-side state, in addition to the current vertex array buffer object binding.

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

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

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

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

`void glColorSubTable` *target start count format type data* [Function]

Respecify a portion of a color table.

target Must be one of `GL_COLOR_TABLE`, `GL_POST_CONVOLUTION_COLOR_TABLE`, or `GL_POST_COLOR_MATRIX_COLOR_TABLE`.

start The starting index of the portion of the color table to be replaced.

count The number of table entries to replace.

format The format of the pixel data in *data*. The allowable values are `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`, `GL_LUMINANCE`, `GL_LUMINANCE_ALPHA`, `GL_RGB`, `GL_BGR`, `GL_RGBA`, and `GL_BGRA`.

type The type of the pixel data in *data*. The allowable values are `GL_UNSIGNED_BYTE`, `GL_BYTE`, `GL_UNSIGNED_SHORT`, `GL_SHORT`, `GL_UNSIGNED_INT`, `GL_INT`, `GL_FLOAT`, `GL_UNSIGNED_BYTE_3_3_2`, `GL_UNSIGNED_BYTE_2_3_3_REV`, `GL_UNSIGNED_SHORT_5_6_5`, `GL_UNSIGNED_SHORT_5_6_5_REV`, `GL_UNSIGNED_SHORT_4_4_4_4`, `GL_UNSIGNED_SHORT_4_4_4_4_REV`, `GL_UNSIGNED_SHORT_5_5_5_1`, `GL_UNSIGNED_SHORT_1_5_5_5_REV`, `GL_UNSIGNED_INT_8_8_8_8`, `GL_UNSIGNED_INT_8_8_8_8_REV`, `GL_UNSIGNED_INT_10_10_10_2`, and `GL_UNSIGNED_INT_2_10_10_10_REV`.

data Pointer to a one-dimensional array of pixel data that is processed to replace the specified region of the color table.

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

If a non-zero named buffer object is bound to the `GL_PIXEL_UNPACK_BUFFER` target (see `glBindBuffer`) while a portion of a color table is respecified, *data* is treated as a byte offset into the buffer object's data store.

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

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

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

`GL_INVALID_VALUE` is generated if *start*+*count*>*width*.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the buffer object's data store is currently mapped.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and *data* is not evenly divisible into the number of bytes needed to store in memory a datum indicated by *type*.

`GL_INVALID_OPERATION` is generated if `glColorSubTable` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glColorTableParameterfv` *target pname params* [Function]
`void glColorTableParameteriv` *target pname params* [Function]

Set color lookup table parameters.

target The target color table. Must be `GL_COLOR_TABLE`, `GL_POST_CONVOLUTION_COLOR_TABLE`, or `GL_POST_COLOR_MATRIX_COLOR_TABLE`.

pname The symbolic name of a texture color lookup table parameter. Must be one of `GL_COLOR_TABLE_SCALE` or `GL_COLOR_TABLE_BIAS`.

params A pointer to an array where the values of the parameters are stored.

`glColorTableParameter` is used to specify the scale factors and bias terms applied to color components when they are loaded into a color table. *target* indicates which color table the scale and bias terms apply to; it must be set to `GL_COLOR_TABLE`, `GL_POST_CONVOLUTION_COLOR_TABLE`, or `GL_POST_COLOR_MATRIX_COLOR_TABLE`.

pname must be `GL_COLOR_TABLE_SCALE` to set the scale factors. In this case, *params* points to an array of four values, which are the scale factors for red, green, blue, and alpha, in that order.

pname must be `GL_COLOR_TABLE_BIAS` to set the bias terms. In this case, *params* points to an array of four values, which are the bias terms for red, green, blue, and alpha, in that order.

The color tables themselves are specified by calling `glColorTable`.

`GL_INVALID_ENUM` is generated if *target* or *pname* is not an acceptable value.

`GL_INVALID_OPERATION` is generated if `glColorTableParameter` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glColorTable` *target internalformat width format type data* [Function]

Define a color lookup table.

target Must be one of `GL_COLOR_TABLE`, `GL_POST_CONVOLUTION_COLOR_TABLE`, `GL_POST_COLOR_MATRIX_COLOR_TABLE`, `GL_PROXY_COLOR_TABLE`, `GL_PROXY_POST_CONVOLUTION_COLOR_TABLE`, or `GL_PROXY_POST_COLOR_MATRIX_COLOR_TABLE`.

internalformat
The internal format of the color table. The allowable values are `GL_ALPHA`, `GL_ALPHA4`, `GL_ALPHA8`, `GL_ALPHA12`, `GL_ALPHA16`, `GL_LUMINANCE`, `GL_LUMINANCE4`, `GL_LUMINANCE8`, `GL_LUMINANCE12`, `GL_LUMINANCE16`, `GL_LUMINANCE_ALPHA`, `GL_LUMINANCE4_ALPHA4`, `GL_LUMINANCE6_ALPHA2`, `GL_LUMINANCE8_ALPHA8`, `GL_LUMINANCE12_ALPHA4`, `GL_LUMINANCE12_ALPHA12`, `GL_LUMINANCE16_ALPHA16`, `GL_INTENSITY`, `GL_INTENSITY4`, `GL_INTENSITY8`, `GL_INTENSITY12`, `GL_INTENSITY16`, `GL_R3_G3_B2`, `GL_RGB`, `GL_RGB4`, `GL_RGB5`, `GL_RGB8`, `GL_RGB10`, `GL_RGB12`, `GL_RGB16`, `GL_RGBA`, `GL_RGBA2`, `GL_RGBA4`, `GL_RGB5_A1`, `GL_RGBA8`, `GL_RGB10_A2`, `GL_RGBA12`, and `GL_RGBA16`.

width The number of entries in the color lookup table specified by *data*.

format The format of the pixel data in *data*. The allowable values are `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`, `GL_LUMINANCE`, `GL_LUMINANCE_ALPHA`, `GL_RGB`, `GL_BGR`, `GL_RGBA`, and `GL_BGRA`.

type The type of the pixel data in *data*. The allowable values are `GL_UNSIGNED_BYTE`, `GL_BYTE`, `GL_UNSIGNED_SHORT`, `GL_SHORT`, `GL_UNSIGNED_INT`, `GL_INT`, `GL_FLOAT`, `GL_UNSIGNED_BYTE_3_3_2`, `GL_UNSIGNED_BYTE_2_3_3_REV`, `GL_UNSIGNED_SHORT_5_6_5`, `GL_UNSIGNED_SHORT_5_6_5_REV`, `GL_UNSIGNED_SHORT_4_4_4_4`, `GL_UNSIGNED_SHORT_4_4_4_4_REV`, `GL_UNSIGNED_SHORT_5_5_5_1`, `GL_UNSIGNED_SHORT_1_5_5_5_REV`, `GL_UNSIGNED_INT_8_8_8_8`, `GL_UNSIGNED_INT_8_8_8_8_REV`, `GL_UNSIGNED_INT_10_10_10_2`, and `GL_UNSIGNED_INT_2_10_10_10_REV`.

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

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

If a non-zero named buffer object is bound to the GL_PIXEL_UNPACK_BUFFER target (see glBindBuffer) while a color table is specified, *data* is treated as a byte offset into the buffer object's data store.

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

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

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

Internal Format **Red**, **Green**, **Blue**, **Alpha**, **Luminance**, **Intensity**

GL_ALPHA , , , A , ,

GL_LUMINANCE , , , , R ,

GL_LUMINANCE_ALPHA , , , A , R ,

GL_INTENSITY , , , , , R

GL_RGB R , G , B , , ,

GL_RGBA R , G , B , A , ,

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

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

The proxy state variables can be retrieved by calling glGetColorTableParameter with a target of GL_PROXY_*. This allows the application to decide if a particular glColorTable command would succeed, and to determine what the resulting color table attributes would be.

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

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

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

then the result of color table lookup is as follows:

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

When `GL_COLOR_TABLE` is enabled, the colors resulting from the pixel map operation (if it is enabled) are mapped by the color lookup table before being passed to the convolution operation. The colors resulting from the convolution operation are modified by the post convolution color lookup table when `GL_POST_CONVOLUTION_COLOR_TABLE` is enabled. These modified colors are then sent to the color matrix

operation. Finally, if `GL_POST_COLOR_MATRIX_COLOR_TABLE` is enabled, the colors resulting from the color matrix operation are mapped by the post color matrix color lookup table before being used by the histogram operation.

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

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

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

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

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

`GL_TABLE_TOO_LARGE` is generated if the requested color table is too large to be supported by the implementation, and *target* is not a `GL_PROXY_*` target.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the buffer object's data store is currently mapped.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and *data* is not evenly divisible into the number of bytes needed to store in memory a datum indicated by *type*.

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

`void glColor3b` *red green blue* [Function]
`void glColor3s` *red green blue* [Function]
`void glColor3i` *red green blue* [Function]
`void glColor3f` *red green blue* [Function]
`void glColor3d` *red green blue* [Function]
`void glColor3ub` *red green blue* [Function]
`void glColor3us` *red green blue* [Function]
`void glColor3ui` *red green blue* [Function]
`void glColor4b` *red green blue alpha* [Function]
`void glColor4s` *red green blue alpha* [Function]
`void glColor4i` *red green blue alpha* [Function]
`void glColor4f` *red green blue alpha* [Function]
`void glColor4d` *red green blue alpha* [Function]
`void glColor4ub` *red green blue alpha* [Function]
`void glColor4us` *red green blue alpha* [Function]
`void glColor4ui` *red green blue alpha* [Function]
`void glColor3bv` *v* [Function]
`void glColor3sv` *v* [Function]
`void glColor3iv` *v* [Function]
`void glColor3fv` *v* [Function]
`void glColor3dv` *v* [Function]
`void glColor3ubv` *v* [Function]
`void glColor3usv` *v* [Function]

`void glColor3uiv` *v* [Function]
`void glColor4bv` *v* [Function]
`void glColor4sv` *v* [Function]
`void glColor4iv` *v* [Function]
`void glColor4fv` *v* [Function]
`void glColor4dv` *v* [Function]
`void glColor4ubv` *v* [Function]
`void glColor4usv` *v* [Function]
`void glColor4uiv` *v* [Function]

Set the current color.

red
green
blue Specify new red, green, and blue values for the current color.

alpha Specifies a new alpha value for the current color. Included only in the four-argument `glColor4` commands.

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

`glColor3b`, `glColor4b`, `glColor3s`, `glColor4s`, `glColor3i`, and `glColor4i` take three or four signed byte, short, or long integers as arguments. When `v` is appended to the name, the color commands can take a pointer to an array of such values.

Current color values are stored in floating-point format, with unspecified mantissa and exponent sizes. Unsigned integer color components, when specified, are linearly mapped to floating-point values such that the largest representable value maps to 1.0 (full intensity), and 0 maps to 0.0 (zero intensity). Signed integer color components, when specified, are linearly mapped to floating-point values such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. (Note that this mapping does not convert 0 precisely to 0.0.) Floating-point values are mapped directly.

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

`void glCompileShader` *shader* [Function]

Compiles a shader object.

shader Specifies the shader object to be compiled.

`glCompileShader` compiles the source code strings that have been stored in the shader object specified by *shader*.

The compilation status will be stored as part of the shader object's state. This value will be set to `GL_TRUE` if the shader was compiled without errors and is ready for use, and `GL_FALSE` otherwise. It can be queried by calling `glGetShader` with arguments *shader* and `GL_COMPILE_STATUS`.

Compilation of a shader can fail for a number of reasons as specified by the OpenGL Shading Language Specification. Whether or not the compilation was successful, information about the compilation can be obtained from the shader object's information log by calling `glGetShaderInfoLog`.

`GL_INVALID_VALUE` is generated if *shader* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *shader* is not a shader object.

`GL_INVALID_OPERATION` is generated if `glCompileShader` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glCompressedTexImage1D` *target level internalformat width border imageSize data* [Function]

Specify a one-dimensional texture image in a compressed format.

target Specifies the target texture. Must be `GL_TEXTURE_1D` or `GL_PROXY_TEXTURE_1D`.

level Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.

internalformat
Specifies the format of the compressed image data stored at address *data*.

width Specifies the width of the texture image including the border if any. If the GL version does not support non-power-of-two sizes, this value must be 2^*n*+2(*border*,) for some integer *n*. All implementations support texture images that are at least 64 texels wide. The height of the 1D texture image is 1.

border Specifies the width of the border. Must be either 0 or 1.

imageSize Specifies the number of unsigned bytes of image data starting at the address specified by *data*.

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

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. To enable and disable one-dimensional texturing, call `glEnable` and `glDisable` with argument `GL_TEXTURE_1D`.

`glCompressedTexImage1D` loads a previously defined, and retrieved, compressed one-dimensional texture image if *target* is `GL_TEXTURE_1D` (see `glTexImage1D`).

If *target* is `GL_PROXY_TEXTURE_1D`, no data is read from *data*, but all of the texture image state is recalculated, checked for consistency, and checked against the implementation's capabilities. If the implementation cannot handle a texture of the requested texture size, it sets all of the image state to 0, but does not generate an error (see `glGetError`). To query for an entire mipmap array, use an image array level greater than or equal to 1.

internalformat must be extension-specified compressed-texture format. When a texture is loaded with `glTexImage1D` using a generic compressed texture format (e.g., `GL_COMPRESSED_RGB`) the GL selects from one of its extensions supporting compressed textures. In order to load the compressed texture image using

`glCompressedTexImage1D`, query the compressed texture image's size and format using `glGetTexLevelParameter`.

If a non-zero named buffer object is bound to the `GL_PIXEL_UNPACK_BUFFER` target (see `glBindBuffer`) while a texture image is specified, *data* is treated as a byte offset into the buffer object's data store.

`GL_INVALID_ENUM` is generated if *internalformat* is one of the generic compressed internal formats: `GL_COMPRESSED_ALPHA`, `GL_COMPRESSED_LUMINANCE`, `GL_COMPRESSED_LUMINANCE_ALPHA`, `GL_COMPRESSED_INTENSITY`, `GL_COMPRESSED_RGB`, or `GL_COMPRESSED_RGBA`.

`GL_INVALID_VALUE` is generated if *imageSize* is not consistent with the format, dimensions, and contents of the specified compressed image data.

`GL_INVALID_OPERATION` is generated if parameter combinations are not supported by the specific compressed internal format as specified in the specific texture compression extension.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the buffer object's data store is currently mapped.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

`GL_INVALID_OPERATION` is generated if `glCompressedTexImage1D` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

Undefined results, including abnormal program termination, are generated if *data* is not encoded in a manner consistent with the extension specification defining the internal compression format.

`void glCompressedTexImage2D` *target level internalformat width height border imageSize data* [Function]

Specify a two-dimensional texture image in a compressed format.

target Specifies the target texture. Must be `GL_TEXTURE_2D`, `GL_PROXY_TEXTURE_2D`, `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Z`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`, or `GL_PROXY_TEXTURE_CUBE_MAP`.

level Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.

internalformat
Specifies the format of the compressed image data stored at address *data*.

width Specifies the width of the texture image including the border if any. If the GL version does not support non-power-of-two sizes, this value must be 2^*n*+2(*border*,) for some integer *n*. All implementations support 2D texture images that are at least 64 texels wide and cube-mapped texture images that are at least 16 texels wide.

height Specifies the height of the texture image including the border if any. If the GL version does not support non-power-of-two sizes, this value must be Must be 2^*n*+2(*border*,) for some integer *n*. All implementations support 2D texture images that are at least 64 texels high and cube-mapped texture images that are at least 16 texels high.

border Specifies the width of the border. Must be either 0 or 1.

imageSize Specifies the number of unsigned bytes of image data starting at the address specified by *data*.

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

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. To enable and disable two-dimensional texturing, call `glEnable` and `glDisable` with argument `GL_TEXTURE_2D`. To enable and disable texturing using cube-mapped textures, call `glEnable` and `glDisable` with argument `GL_TEXTURE_CUBE_MAP`.

`glCompressedTexImage2D` loads a previously defined, and retrieved, compressed two-dimensional texture image if *target* is `GL_TEXTURE_2D` (see `glTexImage2D`).

If *target* is `GL_PROXY_TEXTURE_2D`, no data is read from *data*, but all of the texture image state is recalculated, checked for consistency, and checked against the implementation's capabilities. If the implementation cannot handle a texture of the requested texture size, it sets all of the image state to 0, but does not generate an error (see `glGetError`). To query for an entire mipmap array, use an image array level greater than or equal to 1.

internalformat must be an extension-specified compressed-texture format. When a texture is loaded with `glTexImage2D` using a generic compressed texture format (e.g., `GL_COMPRESSED_RGB`), the GL selects from one of its extensions supporting compressed textures. In order to load the compressed texture image using `glCompressedTexImage2D`, query the compressed texture image's size and format using `glGetTexLevelParameter`.

If a non-zero named buffer object is bound to the `GL_PIXEL_UNPACK_BUFFER` target (see `glBindBuffer`) while a texture image is specified, *data* is treated as a byte offset into the buffer object's data store.

`GL_INVALID_ENUM` is generated if *internalformat* is one of the generic compressed internal formats: `GL_COMPRESSED_ALPHA`, `GL_COMPRESSED_LUMINANCE`, `GL_COMPRESSED_LUMINANCE_ALPHA`, `GL_COMPRESSED_INTENSITY`, `GL_COMPRESSED_RGB`, or `GL_COMPRESSED_RGBA`.

`GL_INVALID_VALUE` is generated if *imageSize* is not consistent with the format, dimensions, and contents of the specified compressed image data.

`GL_INVALID_OPERATION` is generated if parameter combinations are not supported by the specific compressed internal format as specified in the specific texture compression extension.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the buffer object's data store is currently mapped.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

`GL_INVALID_OPERATION` is generated if `glCompressedTexImage2D` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

Undefined results, including abnormal program termination, are generated if *data* is not encoded in a manner consistent with the extension specification defining the internal compression format.

`void glCompressedTexImage3D` *target level internalformat width height depth border imageSize data* [Function]

Specify a three-dimensional texture image in a compressed format.

target Specifies the target texture. Must be `GL_TEXTURE_3D` or `GL_PROXY_TEXTURE_3D`.

level Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.

internalformat
Specifies the format of the compressed image data stored at address *data*.

width Specifies the width of the texture image including the border if any. If the GL version does not support non-power-of-two sizes, this value must be 2^*n*+2(*border*,) for some integer *n*. All implementations support 3D texture images that are at least 16 texels wide.

height Specifies the height of the texture image including the border if any. If the GL version does not support non-power-of-two sizes, this value must be 2^*n*+2(*border*,) for some integer *n*. All implementations support 3D texture images that are at least 16 texels high.

depth Specifies the depth of the texture image including the border if any. If the GL version does not support non-power-of-two sizes, this value must be 2^*n*+2(*border*,) for some integer *n*. All implementations support 3D texture images that are at least 16 texels deep.

border Specifies the width of the border. Must be either 0 or 1.

imageSize Specifies the number of unsigned bytes of image data starting at the address specified by *data*.

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

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. To enable and disable three-dimensional texturing, call `glEnable` and `glDisable` with argument `GL_TEXTURE_3D`.

`glCompressedTexImage3D` loads a previously defined, and retrieved, compressed three-dimensional texture image if *target* is `GL_TEXTURE_3D` (see `glTexImage3D`).

If *target* is `GL_PROXY_TEXTURE_3D`, no data is read from *data*, but all of the texture image state is recalculated, checked for consistency, and checked against the implementation's capabilities. If the implementation cannot handle a texture of the

requested texture size, it sets all of the image state to 0, but does not generate an error (see `glGetError`). To query for an entire mipmap array, use an image array level greater than or equal to 1.

internalformat must be an extension-specified compressed-texture format. When a texture is loaded with `glTexImage2D` using a generic compressed texture format (e.g., `GL_COMPRESSED_RGB`), the GL selects from one of its extensions supporting compressed textures. In order to load the compressed texture image using `glCompressedTexImage3D`, query the compressed texture image's size and format using `glGetTexLevelParameter`.

If a non-zero named buffer object is bound to the `GL_PIXEL_UNPACK_BUFFER` target (see `glBindBuffer`) while a texture image is specified, *data* is treated as a byte offset into the buffer object's data store.

`GL_INVALID_ENUM` is generated if *internalformat* is one of the generic compressed internal formats: `GL_COMPRESSED_ALPHA`, `GL_COMPRESSED_LUMINANCE`, `GL_COMPRESSED_LUMINANCE_ALPHA`, `GL_COMPRESSED_INTENSITY`, `GL_COMPRESSED_RGB`, or `GL_COMPRESSED_RGBA`.

`GL_INVALID_VALUE` is generated if *imageSize* is not consistent with the format, dimensions, and contents of the specified compressed image data.

`GL_INVALID_OPERATION` is generated if parameter combinations are not supported by the specific compressed internal format as specified in the specific texture compression extension.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the buffer object's data store is currently mapped.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

`GL_INVALID_OPERATION` is generated if `glCompressedTexImage3D` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

Undefined results, including abnormal program termination, are generated if *data* is not encoded in a manner consistent with the extension specification defining the internal compression format.

`void glCompressedTexSubImage1D` *target level xoffset width format imageSize data* [Function]

Specify a one-dimensional texture subimage in a compressed format.

target Specifies the target texture. Must be `GL_TEXTURE_1D`.

level Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.

xoffset Specifies a texel offset in the x direction within the texture array.

width Specifies the width of the texture subimage.

format Specifies the format of the compressed image data stored at address *data*.

imageSize Specifies the number of unsigned bytes of image data starting at the address specified by *data*.

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

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. To enable and disable one-dimensional texturing, call `glEnable` and `glDisable` with argument `GL_TEXTURE_1D`.

`glCompressedTexSubImage1D` redefines a contiguous subregion of an existing one-dimensional texture image. The texels referenced by *data* replace the portion of the existing texture array with x indices *xoffset* and *xoffset*+*width*-1, inclusive. This region may not include any texels outside the range of the texture array as it was originally specified. It is not an error to specify a subtexture with width of 0, but such a specification has no effect.

format must be an extension-specified compressed-texture format. The *format* of the compressed texture image is selected by the GL implementation that compressed it (see `glTexImage1D`), and should be queried at the time the texture was compressed with `glGetTexLevelParameter`.

If a non-zero named buffer object is bound to the `GL_PIXEL_UNPACK_BUFFER` target (see `glBindBuffer`) while a texture image is specified, *data* is treated as a byte offset into the buffer object's data store.

`GL_INVALID_ENUM` is generated if *format* is one of these generic compressed internal formats: `GL_COMPRESSED_ALPHA`, `GL_COMPRESSED_LUMINANCE`, `GL_COMPRESSED_LUMINANCE_ALPHA`, `GL_COMPRESSED_INTENSITY`, `GL_COMPRESSED_RGB`, `GL_COMPRESSED_RGBA`, `GL_COMPRESSED_SLUMINANCE`, `GL_COMPRESSED_SLUMINANCE_ALPHA`, `GL_COMPRESSED_SRGB`, `GL_COMPRESSED_SRGBA`, or `GL_COMPRESSED_SRGB_ALPHA`.

`GL_INVALID_VALUE` is generated if *imageSize* is not consistent with the format, dimensions, and contents of the specified compressed image data.

`GL_INVALID_OPERATION` is generated if parameter combinations are not supported by the specific compressed internal format as specified in the specific texture compression extension.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the buffer object's data store is currently mapped.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

`GL_INVALID_OPERATION` is generated if `glCompressedTexSubImage1D` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

Undefined results, including abnormal program termination, are generated if *data* is not encoded in a manner consistent with the extension specification defining the internal compression format.

`void glCompressedTexSubImage2D` *target level xoffset yoffset width height format imageSize data* [Function]

Specify a two-dimensional texture subimage in a compressed format.

target Specifies the target texture. Must be `GL_TEXTURE_2D`, `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Z`, or `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`.

level Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.

xoffset Specifies a texel offset in the x direction within the texture array.

yoffset Specifies a texel offset in the y direction within the texture array.

width Specifies the width of the texture subimage.

height Specifies the height of the texture subimage.

format Specifies the format of the compressed image data stored at address *data*.

imageSize Specifies the number of unsigned bytes of image data starting at the address specified by *data*.

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

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. To enable and disable two-dimensional texturing, call `glEnable` and `glDisable` with argument `GL_TEXTURE_2D`. To enable and disable texturing using cube-mapped texture, call `glEnable` and `glDisable` with argument `GL_TEXTURE_CUBE_MAP`.

`glCompressedTexSubImage2D` redefines a contiguous subregion of an existing two-dimensional texture image. The texels referenced by *data* replace the portion of the existing texture array with x indices *xoffset* and *xoffset*+*width*-1, and the 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 width of 0, but such a specification has no effect.

format must be an extension-specified compressed-texture format. The *format* of the compressed texture image is selected by the GL implementation that compressed it (see `glTexImage2D`) and should be queried at the time the texture was compressed with `glGetTexLevelParameter`.

If a non-zero named buffer object is bound to the `GL_PIXEL_UNPACK_BUFFER` target (see `glBindBuffer`) while a texture image is specified, *data* is treated as a byte offset into the buffer object's data store.

`GL_INVALID_ENUM` is generated if *format* is one of these generic compressed internal formats: `GL_COMPRESSED_ALPHA`, `GL_COMPRESSED_LUMINANCE`, `GL_COMPRESSED_LUMINANCE_ALPHA`, `GL_COMPRESSED_INTENSITY`, `GL_COMPRESSED_RGB`, `GL_COMPRESSED_RGBA`, `GL_COMPRESSED_SLUMINANCE`, `GL_COMPRESSED_SLUMINANCE_ALPHA`, `GL_COMPRESSED_SRGB`, `GL_COMPRESSED_SRGBA`, or `GL_COMPRESSED_SRGB_ALPHA`.

`GL_INVALID_VALUE` is generated if *imageSize* is not consistent with the format, dimensions, and contents of the specified compressed image data.

`GL_INVALID_OPERATION` is generated if parameter combinations are not supported by the specific compressed internal format as specified in the specific texture compression extension.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the buffer object's data store is currently mapped.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

`GL_INVALID_OPERATION` is generated if `glCompressedTexSubImage2D` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

Undefined results, including abnormal program termination, are generated if *data* is not encoded in a manner consistent with the extension specification defining the internal compression format.

`void glCompressedTexSubImage3D` *target level xoffset yoffset zoffset width height depth format imageSize data* [Function]

Specify a three-dimensional texture subimage in a compressed format.

target Specifies the target texture. Must be `GL_TEXTURE_3D`.

level Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *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.

depth Specifies the depth of the texture subimage.

format Specifies the format of the compressed image data stored at address *data*.

imageSize Specifies the number of unsigned bytes of image data starting at the address specified by *data*.

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

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. To enable and disable three-dimensional texturing, call `glEnable` and `glDisable` with argument `GL_TEXTURE_3D`.

`glCompressedTexSubImage3D` redefines a contiguous subregion of an existing three-dimensional texture image. The texels referenced by *data* replace the portion of the existing texture array with x indices *xoffset* and *xoffset*+*width*-1, and the y indices *yoffset* and *yoffset*+*height*-1, and the z indices *zoffset* and *zoffset*+*depth*-1, inclusive. This region may not include any texels outside the range of the texture array as it was originally specified. It is not an error to specify a subtexture with width of 0, but such a specification has no effect.

format must be an extension-specified compressed-texture format. The *format* of the compressed texture image is selected by the GL implementation that compressed it (see `glTexImage3D`) and should be queried at the time the texture was compressed with `glGetTexLevelParameter`.

If a non-zero named buffer object is bound to the `GL_PIXEL_UNPACK_BUFFER` target (see `glBindBuffer`) while a texture image is specified, *data* is treated as a byte offset into the buffer object's data store.

`GL_INVALID_ENUM` is generated if *format* is one of these generic compressed internal formats: `GL_COMPRESSED_ALPHA`, `GL_COMPRESSED_LUMINANCE`, `GL_COMPRESSED_LUMINANCE_ALPHA`, `GL_COMPRESSED_INTENSITY`, `GL_COMPRESSED_RGB`, `GL_COMPRESSED_RGBA`, `GL_COMPRESSED_SLUMINANCE`, `GL_COMPRESSED_SLUMINANCE_ALPHA`, `GL_COMPRESSED_SRGB`, `GL_COMPRESSED_SRGBA`, or `GL_COMPRESSED_SRGB_ALPHA`.

`GL_INVALID_VALUE` is generated if *imageSize* is not consistent with the format, dimensions, and contents of the specified compressed image data.

`GL_INVALID_OPERATION` is generated if parameter combinations are not supported by the specific compressed internal format as specified in the specific texture compression extension.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the buffer object's data store is currently mapped.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

`GL_INVALID_OPERATION` is generated if `glCompressedTexSubImage3D` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

Undefined results, including abnormal program termination, are generated if *data* is not encoded in a manner consistent with the extension specification defining the internal compression format.

`void glConvolutionFilter1D` *target internalformat width format type data* [Function]

Define a one-dimensional convolution filter.

target Must be `GL_CONVOLUTION_1D`.

internalformat

The internal format of the convolution filter kernel. The allowable values are `GL_ALPHA`, `GL_ALPHA4`, `GL_ALPHA8`, `GL_ALPHA12`, `GL_ALPHA16`, `GL_LUMINANCE`, `GL_LUMINANCE4`, `GL_LUMINANCE8`, `GL_LUMINANCE12`, `GL_LUMINANCE16`, `GL_LUMINANCE_ALPHA`, `GL_LUMINANCE4_ALPHA4`, `GL_LUMINANCE6_ALPHA2`, `GL_LUMINANCE8_ALPHA8`, `GL_LUMINANCE12_ALPHA4`, `GL_LUMINANCE12_ALPHA12`, `GL_LUMINANCE16_ALPHA16`, `GL_INTENSITY`, `GL_INTENSITY4`, `GL_INTENSITY8`, `GL_INTENSITY12`, `GL_INTENSITY16`, `GL_R3_G3_B2`, `GL_RGB`, `GL_RGB4`, `GL_RGB5`, `GL_RGB8`, `GL_RGB10`, `GL_RGB12`, `GL_RGB16`, `GL_RGBA`, `GL_RGBA2`, `GL_RGBA4`, `GL_RGB5_A1`, `GL_RGBA8`, `GL_RGB10_A2`, `GL_RGBA12`, or `GL_RGBA16`.

width The width of the pixel array referenced by *data*.

format The format of the pixel data in *data*. The allowable values are `GL_ALPHA`, `GL_LUMINANCE`, `GL_LUMINANCE_ALPHA`, `GL_INTENSITY`, `GL_RGB`, and `GL_RGBA`.

type The type of the pixel data in *data*. Symbolic constants `GL_UNSIGNED_BYTE`, `GL_BYTE`, `GL_BITMAP`, `GL_UNSIGNED_SHORT`, `GL_SHORT`, `GL_UNSIGNED_INT`, `GL_INT`, `GL_FLOAT`, `GL_UNSIGNED_BYTE_3_3_2`, `GL_UNSIGNED_BYTE_2_3_3_REV`, `GL_UNSIGNED_SHORT_5_6_5`, `GL_UNSIGNED_SHORT_5_6_5_REV`, `GL_UNSIGNED_SHORT_4_4_4_4`, `GL_UNSIGNED_SHORT_4_4_4_4_REV`, `GL_UNSIGNED_SHORT_5_5_5_1`, `GL_UNSIGNED_SHORT_1_5_5_5_REV`, `GL_UNSIGNED_INT_8_8_8_8`, `GL_UNSIGNED_INT_8_8_8_8_REV`, `GL_UNSIGNED_INT_10_10_10_2`, and `GL_UNSIGNED_INT_2_10_10_10_REV` are accepted.

data Pointer to a one-dimensional array of pixel data that is processed to build the convolution filter kernel.

`glConvolutionFilter1D` builds a one-dimensional convolution filter kernel from an array of pixels.

The pixel array specified by *width*, *format*, *type*, and *data* is extracted from memory and processed just as if `glDrawPixels` were called, but processing stops after the final expansion to RGBA is completed.

If a non-zero named buffer object is bound to the `GL_PIXEL_UNPACK_BUFFER` target (see `glBindBuffer`) while a convolution filter is specified, *data* is treated as a byte offset into the buffer object's data store.

The R, G, B, and A components of each pixel are next scaled by the four 1D `GL_CONVOLUTION_FILTER_SCALE` parameters and biased by the four 1D `GL_CONVOLUTION_FILTER_BIAS` parameters. (The scale and bias parameters are set by `glConvolutionParameter` using the `GL_CONVOLUTION_1D` target and the names `GL_CONVOLUTION_FILTER_SCALE` and `GL_CONVOLUTION_FILTER_BIAS`. The parameters themselves are vectors of four values that are applied to red, green, blue, and alpha, in that order.) The R, G, B, and A values are not clamped to [0,1] at any time during this process.

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

Internal Format
Red, **Green**, **Blue**, **Alpha**, **Luminance**, **Intensity**

`GL_ALPHA` , , , A , ,

`GL_LUMINANCE`
, , , , R ,

`GL_LUMINANCE_ALPHA`
, , , A , R ,

`GL_INTENSITY`
: , , , , , R

`GL_RGB`
: R , G , B , , ,

`GL_RGBA`
: R , G , B , A , ,

The red, green, blue, alpha, luminance, and/or intensity components of the resulting pixels are stored in floating-point rather than integer format. They form a one-dimensional filter kernel image indexed with coordinate *i* such that *i* starts at 0 and increases from left to right. Kernel location *i* is derived from the *i*th pixel, counting from 0.

Note that after a convolution is performed, the resulting color components are also scaled by their corresponding `GL_POST_CONVOLUTION_c_SCALE` parameters and biased by their corresponding `GL_POST_CONVOLUTION_c_BIAS` parameters (where *c* takes on the values **RED**, **GREEN**, **BLUE**, and **ALPHA**). These parameters are set by `glPixelTransfer`.

`GL_INVALID_ENUM` is generated if *target* is not `GL_CONVOLUTION_1D`.

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

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

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

`GL_INVALID_VALUE` is generated if *width* is less than zero or greater than the maximum supported value. This value may be queried with `glGetConvolutionParameter` using target `GL_CONVOLUTION_1D` and name `GL_MAX_CONVOLUTION_WIDTH`.

`GL_INVALID_OPERATION` is generated if *format* is one of `GL_UNSIGNED_BYTE_3_3_2`, `GL_UNSIGNED_BYTE_2_3_3_REV`, `GL_UNSIGNED_SHORT_5_6_5`, or `GL_UNSIGNED_SHORT_5_6_5_REV` and *type* is not `GL_RGB`.

`GL_INVALID_OPERATION` is generated if *format* is one of `GL_UNSIGNED_SHORT_4_4_4_4`, `GL_UNSIGNED_SHORT_4_4_4_4_REV`, `GL_UNSIGNED_SHORT_5_5_5_1`, `GL_UNSIGNED_SHORT_1_5_5_5_REV`, `GL_UNSIGNED_INT_8_8_8_8`, `GL_UNSIGNED_INT_8_8_8_8_REV`, `GL_UNSIGNED_INT_10_10_10_2`, or `GL_UNSIGNED_INT_2_10_10_10_REV` and *type* is neither `GL_RGBA` nor `GL_BGRA`.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the buffer object's data store is currently mapped.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and *data* is not evenly divisible into the number of bytes needed to store in memory a datum indicated by *type*.

`GL_INVALID_OPERATION` is generated if `glConvolutionFilter1D` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glConvolutionFilter2D` *target internalformat width height format type data* [Function]

Define a two-dimensional convolution filter.

target Must be GL_CONVOLUTION_2D.

internalformat
The internal format of the convolution filter kernel. The allowable values are GL_ALPHA, GL_ALPHA4, GL_ALPHA8, GL_ALPHA12, GL_ALPHA16, GL_LUMINANCE, GL_LUMINANCE4, GL_LUMINANCE8, GL_LUMINANCE12, GL_LUMINANCE16, GL_LUMINANCE_ALPHA, GL_LUMINANCE4_ALPHA4, GL_LUMINANCE6_ALPHA2, GL_LUMINANCE8_ALPHA8, GL_LUMINANCE12_ALPHA4, GL_LUMINANCE12_ALPHA12, GL_LUMINANCE16_ALPHA16, GL_INTENSITY, GL_INTENSITY4, GL_INTENSITY8, GL_INTENSITY12, GL_INTENSITY16, GL_R3_G3_B2, GL_RGB, GL_RGB4, GL_RGB5, GL_RGB8, GL_RGB10, GL_RGB12, GL_RGB16, GL_RGBA, GL_RGBA2, GL_RGBA4, GL_RGB5_A1, GL_RGBA8, GL_RGB10_A2, GL_RGBA12, or GL_RGBA16.

width The width of the pixel array referenced by *data*.

height The height of the pixel array referenced by *data*.

format The format of the pixel data in *data*. The allowable values are GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_BGR, GL_RGBA, GL_BGRA, GL_LUMINANCE, and GL_LUMINANCE_ALPHA.

type The type of the pixel data in *data*. Symbolic constants GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, GL_FLOAT, GL_UNSIGNED_BYTE_3_3_2, GL_UNSIGNED_BYTE_2_3_3_REV, GL_UNSIGNED_SHORT_5_6_5, GL_UNSIGNED_SHORT_5_6_5_REV, GL_UNSIGNED_SHORT_4_4_4_4, GL_UNSIGNED_SHORT_4_4_4_4_REV, GL_UNSIGNED_SHORT_5_5_5_1, GL_UNSIGNED_SHORT_1_5_5_5_REV, GL_UNSIGNED_INT_8_8_8_8, GL_UNSIGNED_INT_8_8_8_8_REV, GL_UNSIGNED_INT_10_10_10_2, and GL_UNSIGNED_INT_2_10_10_10_REV are accepted.

data Pointer to a two-dimensional array of pixel data that is processed to build the convolution filter kernel.

glConvolutionFilter2D builds a two-dimensional convolution filter kernel from an array of pixels.

The pixel array specified by *width*, *height*, *format*, *type*, and *data* is extracted from memory and processed just as if glDrawPixels were called, but processing stops after the final expansion to RGBA is completed.

If a non-zero named buffer object is bound to the GL_PIXEL_UNPACK_BUFFER target (see glBindBuffer) while a convolution filter is specified, *data* is treated as a byte offset into the buffer object's data store.

The R, G, B, and A components of each pixel are next scaled by the four 2D GL_CONVOLUTION_FILTER_SCALE parameters and biased by the four 2D GL_CONVOLUTION_FILTER_BIAS parameters. (The scale and bias parameters are set by glConvolutionParameter using the GL_CONVOLUTION_2D target and the names GL_CONVOLUTION_FILTER_SCALE and GL_CONVOLUTION_FILTER_BIAS. The parameters themselves are vectors of four values that are applied to red, green, blue, and alpha, in that order.) The R, G, B, and A values are not clamped to [0,1] at any time during this process.

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

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

The red, green, blue, alpha, luminance, and/or intensity components of the resulting pixels are stored in floating-point rather than integer format. They form a two-dimensional filter kernel image indexed with coordinates *i* and *j* such that *i* starts at zero and increases from left to right, and *j* starts at zero and increases from bottom to top. Kernel location *i*,*j* is derived from the *N*th pixel, where *N* is *i*+*j***width*.

Note that after a convolution is performed, the resulting color components are also scaled by their corresponding GL_POST_CONVOLUTION_*c*_SCALE parameters and biased by their corresponding GL_POST_CONVOLUTION_*c*_BIAS parameters (where *c* takes on the values **RED**, **GREEN**, **BLUE**, and **ALPHA**). These parameters are set by glPixelTransfer.

GL_INVALID_ENUM is generated if *target* is not GL_CONVOLUTION_2D.

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

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

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

GL_INVALID_VALUE is generated if *width* is less than zero or greater than the maximum supported value. This value may be queried with glGetConvolutionParameter using target GL_CONVOLUTION_2D and name GL_MAX_CONVOLUTION_WIDTH.

GL_INVALID_VALUE is generated if *height* is less than zero or greater than the maximum supported value. This value may be queried with glGetConvolutionParameter using target GL_CONVOLUTION_2D and name GL_MAX_CONVOLUTION_HEIGHT.

GL_INVALID_OPERATION is generated if *height* is one of GL_UNSIGNED_BYTE_3_3_2, GL_UNSIGNED_BYTE_2_3_3_REV, GL_UNSIGNED_SHORT_5_6_5, or GL_UNSIGNED_SHORT_5_6_5_REV and *format* is not GL_RGB.

GL_INVALID_OPERATION is generated if *height* is one of GL_UNSIGNED_SHORT_4_4_4_4, GL_UNSIGNED_SHORT_4_4_4_4_REV, GL_UNSIGNED_SHORT_5_5_5_1, GL_UNSIGNED_SHORT_1_5_5_5_REV, GL_UNSIGNED_INT_8_8_8_8, GL_UNSIGNED_INT_8_8_8_8_REV,

GL_UNSIGNED_INT_10_10_10_2, or GL_UNSIGNED_INT_2_10_10_10_REV and *format* is neither GL_RGBA nor GL_BGRA.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and the buffer object's data store is currently mapped.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and *data* is not evenly divisible into the number of bytes needed to store in memory a datum indicated by *type*.

GL_INVALID_OPERATION is generated if glConvolutionFilter2D is executed between the execution of glBegin and the corresponding execution of glEnd.

`void glConvolutionParameterf` *target pname params* [Function]
`void glConvolutionParameteri` *target pname params* [Function]
`void glConvolutionParameterfv` *target pname params* [Function]
`void glConvolutionParameteriv` *target pname params* [Function]

Set convolution parameters.

target The target for the convolution parameter. Must be one of GL_CONVOLUTION_1D, GL_CONVOLUTION_2D, or GL_SEPARABLE_2D.

pname The parameter to be set. Must be GL_CONVOLUTION_BORDER_MODE.

params The parameter value. Must be one of GL_REDUCE, GL_CONSTANT_BORDER, GL_REPLICATE_BORDER.

glConvolutionParameter sets the value of a convolution parameter.

target selects the convolution filter to be affected: GL_CONVOLUTION_1D, GL_CONVOLUTION_2D, or GL_SEPARABLE_2D for the 1D, 2D, or separable 2D filter, respectively.

pname selects the parameter to be changed. GL_CONVOLUTION_FILTER_SCALE and GL_CONVOLUTION_FILTER_BIAS affect the definition of the convolution filter kernel; see glConvolutionFilter1D, glConvolutionFilter2D, and glSeparableFilter2D for details. In these cases, *params*v is an array of four values to be applied to red, green, blue, and alpha values, respectively. The initial value for GL_CONVOLUTION_FILTER_SCALE is (1, 1, 1, 1), and the initial value for GL_CONVOLUTION_FILTER_BIAS is (0, 0, 0, 0).

A *pname* value of GL_CONVOLUTION_BORDER_MODE controls the convolution border mode. The accepted modes are:

GL_REDUCE

The image resulting from convolution is smaller than the source image. If the filter width is *Wf* and height is *Hf*, and the source image width is *Ws* and height is *Hs*, then the convolved image width will be *Ws*-*Wf*+1 and height will be *Hs*-*Hf*+1. (If this reduction would generate an image with zero or negative width and/or height, the output is simply null,

with no error generated.) The coordinates of the image resulting from convolution are zero through *Ws-Wf* in width and zero through *Hs-Hf* in height.

`GL_CONSTANT_BORDER`
: The image resulting from convolution is the same size as the source image, and processed as if the source image were surrounded by pixels with their color specified by the `GL_CONVOLUTION_BORDER_COLOR`.

`GL_REPLICATE_BORDER`
: The image resulting from convolution is the same size as the source image, and processed as if the outermost pixel on the border of the source image were replicated.

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

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

`GL_INVALID_ENUM` is generated if *pname* is `GL_CONVOLUTION_BORDER_MODE` and *params* is not one of `GL_REDUCE`, `GL_CONSTANT_BORDER`, or `GL_REPLICATE_BORDER`.

`GL_INVALID_OPERATION` is generated if `glConvolutionParameter` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glCopyColorSubTable` *target start x y width* [Function]

Respecify a portion of a color table.

target Must be one of `GL_COLOR_TABLE`, `GL_POST_CONVOLUTION_COLOR_TABLE`, or `GL_POST_COLOR_MATRIX_COLOR_TABLE`.

start The starting index of the portion of the color table to be replaced.

x
y The window coordinates of the left corner of the row of pixels to be copied.

width The number of table entries to replace.

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

`GL_INVALID_VALUE` is generated if *target* is not a previously defined color table.

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

`GL_INVALID_VALUE` is generated if *start*+*x*>*width*.

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

`void glCopyColorTable` *target internalformat x y width* [Function]

Copy pixels into a color table.

target The color table target. Must be `GL_COLOR_TABLE`, `GL_POST_CONVOLUTION_COLOR_TABLE`, or `GL_POST_COLOR_MATRIX_COLOR_TABLE`.

internalformat
: The internal storage format of the texture image. Must be one of the following symbolic constants: `GL_ALPHA`, `GL_ALPHA4`, `GL_ALPHA8`, `GL_ALPHA12`, `GL_ALPHA16`, `GL_LUMINANCE`, `GL_LUMINANCE4`, `GL_LUMINANCE8`, `GL_LUMINANCE12`, `GL_LUMINANCE16`, `GL_LUMINANCE_ALPHA`, `GL_LUMINANCE4_ALPHA4`, `GL_LUMINANCE6_ALPHA2`, `GL_LUMINANCE8_ALPHA8`, `GL_LUMINANCE12_ALPHA4`, `GL_LUMINANCE12_ALPHA12`, `GL_LUMINANCE16_ALPHA16`, `GL_INTENSITY`, `GL_INTENSITY4`, `GL_INTENSITY8`, `GL_INTENSITY12`, `GL_INTENSITY16`, `GL_R3_G3_B2`, `GL_RGB`, `GL_RGB4`, `GL_RGB5`, `GL_RGB8`, `GL_RGB10`, `GL_RGB12`, `GL_RGB16`, `GL_RGBA`, `GL_RGBA2`, `GL_RGBA4`, `GL_RGB5_A1`, `GL_RGBA8`, `GL_RGB10_A2`, `GL_RGBA12`, or `GL_RGBA16`.

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

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

width
: The width of the pixel rectangle.

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

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

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

The four scale parameters and the four bias parameters that are defined for the table are then used to scale and bias the R, G, B, and A components of each pixel. The scale and bias parameters are set by calling `glColorTableParameter`.

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

Internal Format
: **Red**, **Green**, **Blue**, **Alpha**, **Luminance**, **Intensity**

`GL_ALPHA` , , , A , ,

`GL_LUMINANCE`
: , , , , R ,

`GL_LUMINANCE_ALPHA`
: , , , A , R ,

`GL_INTENSITY`
: , , , , , R

`GL_RGB` R , G , B , , ,

`GL_RGBA` R , G , B , A , ,

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

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

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

`GL_INVALID_VALUE` is generated if *internalformat* is not one of the allowable values.

`GL_TABLE_TOO_LARGE` is generated if the requested color table is too large to be supported by the implementation.

`GL_INVALID_OPERATION` is generated if `glCopyColorTable` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glCopyConvolutionFilter1D` *target internalformat x y width* [Function]

Copy pixels into a one-dimensional convolution filter.

target Must be `GL_CONVOLUTION_1D`.

internalformat
The internal format of the convolution filter kernel. The allowable values are `GL_ALPHA`, `GL_ALPHA4`, `GL_ALPHA8`, `GL_ALPHA12`, `GL_ALPHA16`, `GL_LUMINANCE`, `GL_LUMINANCE4`, `GL_LUMINANCE8`, `GL_LUMINANCE12`, `GL_LUMINANCE16`, `GL_LUMINANCE_ALPHA`, `GL_LUMINANCE4_ALPHA4`, `GL_LUMINANCE6_ALPHA2`, `GL_LUMINANCE8_ALPHA8`, `GL_LUMINANCE12_ALPHA4`, `GL_LUMINANCE12_ALPHA12`, `GL_LUMINANCE16_ALPHA16`, `GL_INTENSITY`, `GL_INTENSITY4`, `GL_INTENSITY8`, `GL_INTENSITY12`, `GL_INTENSITY16`, `GL_R3_G3_B2`, `GL_RGB`, `GL_RGB4`, `GL_RGB5`, `GL_RGB8`, `GL_RGB10`, `GL_RGB12`, `GL_RGB16`, `GL_RGBA`, `GL_RGBA2`, `GL_RGBA4`, `GL_RGB5_A1`, `GL_RGBA8`, `GL_RGB10_A2`, `GL_RGBA12`, or `GL_RGBA16`.

x
y The window space coordinates of the lower-left coordinate of the pixel array to copy.

width The width of the pixel array to copy.

`glCopyConvolutionFilter1D` defines a one-dimensional convolution filter kernel with pixels from the current `GL_READ_BUFFER` (rather than from main memory, as is the case for `glConvolutionFilter1D`).

The screen-aligned pixel rectangle with lower-left corner at (x,\ y), width *width* and height 1 is used to define the convolution filter. If any pixels within this region are outside the window that is associated with the GL context, the values obtained for those pixels are undefined.

The pixels in the rectangle are processed exactly as if `glReadPixels` had been called with *format* set to RGBA, but the process stops just before final conversion. The R, G, B, and A components of each pixel are next scaled by the four 1D `GL_CONVOLUTION_FILTER_SCALE` parameters and biased by the four 1D `GL_CONVOLUTION_FILTER_BIAS`

parameters. (The scale and bias parameters are set by `glConvolutionParameter` using the `GL_CONVOLUTION_1D` target and the names `GL_CONVOLUTION_FILTER_SCALE` and `GL_CONVOLUTION_FILTER_BIAS`. The parameters themselves are vectors of four values that are applied to red, green, blue, and alpha, in that order.) The R, G, B, and A values are not clamped to [0,1] at any time during this process.

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

Internal Format
: **Red**, **Green**, **Blue**, **Alpha**, **Luminance**, **Intensity**

`GL_ALPHA` , , , A , ,

`GL_LUMINANCE`
: , , , , R ,

`GL_LUMINANCE_ALPHA`
: , , , A , R ,

`GL_INTENSITY`
: , , , , , R

`GL_RGB` R , G , B , , ,

`GL_RGBA` R , G , B , A , ,

The red, green, blue, alpha, luminance, and/or intensity components of the resulting pixels are stored in floating-point rather than integer format.

Pixel ordering is such that lower x screen coordinates correspond to lower *i* filter image coordinates.

Note that after a convolution is performed, the resulting color components are also scaled by their corresponding `GL_POST_CONVOLUTION_c_SCALE` parameters and biased by their corresponding `GL_POST_CONVOLUTION_c_BIAS` parameters (where *c* takes on the values **RED**, **GREEN**, **BLUE**, and **ALPHA**). These parameters are set by `glPixelTransfer`.

`GL_INVALID_ENUM` is generated if *target* is not `GL_CONVOLUTION_1D`.

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

`GL_INVALID_VALUE` is generated if *width* is less than zero or greater than the maximum supported value. This value may be queried with `glGetConvolutionParameter` using target `GL_CONVOLUTION_1D` and name `GL_MAX_CONVOLUTION_WIDTH`.

`GL_INVALID_OPERATION` is generated if `glCopyConvolutionFilter1D` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glCopyConvolutionFilter2D` *target internalformat x y width height* [Function]

Copy pixels into a two-dimensional convolution filter.

target Must be `GL_CONVOLUTION_2D`.

internalformat
: The internal format of the convolution filter kernel. The allowable values are `GL_ALPHA`, `GL_ALPHA4`, `GL_ALPHA8`, `GL_ALPHA12`, `GL_ALPHA16`, `GL_LUMINANCE`, `GL_LUMINANCE4`, `GL_LUMINANCE8`, `GL_LUMINANCE12`, `GL_LUMINANCE16`, `GL_LUMINANCE_ALPHA`, `GL_LUMINANCE4_ALPHA4`, `GL_LUMINANCE6_ALPHA2`, `GL_LUMINANCE8_ALPHA8`, `GL_LUMINANCE12_ALPHA4`, `GL_LUMINANCE12_ALPHA12`, `GL_LUMINANCE16_ALPHA16`, `GL_INTENSITY`, `GL_INTENSITY4`, `GL_INTENSITY8`, `GL_INTENSITY12`, `GL_INTENSITY16`, `GL_R3_G3_B2`, `GL_RGB`, `GL_RGB4`, `GL_RGB5`, `GL_RGB8`, `GL_RGB10`, `GL_RGB12`, `GL_RGB16`, `GL_RGBA`, `GL_RGBA2`, `GL_RGBA4`, `GL_RGB5_A1`, `GL_RGBA8`, `GL_RGB10_A2`, `GL_RGBA12`, or `GL_RGBA16`.

x
y
: The window space coordinates of the lower-left coordinate of the pixel array to copy.

width
: The width of the pixel array to copy.

height
: The height of the pixel array to copy.

`glCopyConvolutionFilter2D` defines a two-dimensional convolution filter kernel with pixels from the current `GL_READ_BUFFER` (rather than from main memory, as is the case for `glConvolutionFilter2D`).

The screen-aligned pixel rectangle with lower-left corner at (x,\ y), width *width* and height *height* is used to define the convolution filter. If any pixels within this region are outside the window that is associated with the GL context, the values obtained for those pixels are undefined.

The pixels in the rectangle are processed exactly as if `glReadPixels` had been called with *format* set to RGBA, but the process stops just before final conversion. The R, G, B, and A components of each pixel are next scaled by the four 2D `GL_CONVOLUTION_FILTER_SCALE` parameters and biased by the four 2D `GL_CONVOLUTION_FILTER_BIAS` parameters. (The scale and bias parameters are set by `glConvolutionParameter` using the `GL_CONVOLUTION_2D` target and the names `GL_CONVOLUTION_FILTER_SCALE` and `GL_CONVOLUTION_FILTER_BIAS`. The parameters themselves are vectors of four values that are applied to red, green, blue, and alpha, in that order.) The R, G, B, and A values are not clamped to [0,1] at any time during this process.

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

Internal Format
: **Red**, **Green**, **Blue**, **Alpha**, **Luminance**, **Intensity**

`GL_ALPHA` , , , A , ,

`GL_LUMINANCE`
: , , , , R ,

`GL_LUMINANCE_ALPHA`
: , , , A , R ,

`GL_INTENSITY`
: , , , , , R

`GL_RGB`
: R , G , B , , ,

`GL_RGBA`
: R , G , B , A , ,

The red, green, blue, alpha, luminance, and/or intensity components of the resulting pixels are stored in floating-point rather than integer format.

Pixel ordering is such that lower x screen coordinates correspond to lower *i* filter image coordinates, and lower y screen coordinates correspond to lower *j* filter image coordinates.

Note that after a convolution is performed, the resulting color components are also scaled by their corresponding `GL_POST_CONVOLUTION_c_SCALE` parameters and biased by their corresponding `GL_POST_CONVOLUTION_c_BIAS` parameters (where *c* takes on the values **RED**, **GREEN**, **BLUE**, and **ALPHA**). These parameters are set by `glPixelTransfer`.

`GL_INVALID_ENUM` is generated if *target* is not `GL_CONVOLUTION_2D`.

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

`GL_INVALID_VALUE` is generated if *width* is less than zero or greater than the maximum supported value. This value may be queried with `glGetConvolutionParameter` using target `GL_CONVOLUTION_2D` and name `GL_MAX_CONVOLUTION_WIDTH`.

`GL_INVALID_VALUE` is generated if *height* is less than zero or greater than the maximum supported value. This value may be queried with `glGetConvolutionParameter` using target `GL_CONVOLUTION_2D` and name `GL_MAX_CONVOLUTION_HEIGHT`.

`GL_INVALID_OPERATION` is generated if `glCopyConvolutionFilter2D` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glCopyPixels` *x y width height type* [Function]

Copy pixels in the frame buffer.

x
y
: Specify the window coordinates of the lower left corner of the rectangular region of pixels to be copied.

width
height
: Specify the dimensions of the rectangular region of pixels to be copied. Both must be nonnegative.

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

`glCopyPixels` copies a screen-aligned rectangle of pixels from the specified frame buffer location to a region relative to the current raster position. Its operation is well defined only if the entire pixel source region is within the exposed portion of the window. Results of copies from outside the window, or from regions of the window that are not exposed, are hardware dependent and undefined.

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

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

`glCopyPixels` copies values from each pixel with the lower left-hand corner at (*x*+*i*,*y*+*j*) for 0<=*i*<*width* and 0<=*j*<*height*. This pixel is said to be the *i*th pixel in the *j*th row. Pixels are copied in row order from the lowest to the highest row, left to right in each row.

type specifies whether color, depth, or stencil data is to be copied. The details of the transfer for each data type are as follows:

`GL_COLOR` Indices or RGBA colors are read from the buffer currently specified as the read source buffer (see `glReadBuffer`). If the GL is in color index mode, each index that is read from this buffer is converted to a fixed-point format with an unspecified number of bits to the right of the binary point. Each index is then shifted left by `GL_INDEX_SHIFT` bits, and added to `GL_INDEX_OFFSET`. If `GL_INDEX_SHIFT` is negative, the shift is to the right. In either case, zero bits fill otherwise unspecified bit locations in the result. If `GL_MAP_COLOR` is true, the index is replaced with the value that it references in lookup table `GL_PIXEL_MAP_I_TO_I`. Whether the lookup replacement of the index is done or not, the integer part of the index is then ANDed with 2^*b*-1, where *b* is the number of bits in a color index buffer.

If the GL is in RGBA mode, the red, green, blue, and alpha components of each pixel that is read are converted to an internal floating-point format with unspecified precision. The conversion maps the largest representable component value to 1.0, and component value 0 to 0.0. The resulting floating-point color values are then multiplied by `GL_c_SCALE` and added to `GL_c_BIAS`, where *c* is RED, GREEN, BLUE, and ALPHA for the respective color components. The results are clamped to the range [0,1]. If `GL_MAP_COLOR` is true, each color component is scaled by the size of lookup table `GL_PIXEL_MAP_c_TO_c`, then replaced by the value that it references in that table. *c* is R, G, B, or A.

If the `ARB_imaging` extension is supported, the color values may be additionally processed by color-table lookups, color-matrix transformations, and convolution filters.

The GL then converts the resulting indices or RGBA colors to fragments by attaching the current raster position *z* coordinate and texture coordinates to each pixel, then assigning window coordinates (*x_r*+*i*,*y_r*+*j*), where (*x_r*,*y_r*) is the current raster position, and the pixel was the *i*th pixel in the *j*th row. These pixel fragments are then treated just like the fragments generated by rasterizing points, lines, or polygons. Tex-

ture mapping, fog, and all the fragment operations are applied before the fragments are written to the frame buffer.

GL_DEPTH Depth values are read from the depth buffer and converted directly to an internal floating-point format with unspecified precision. The resulting floating-point depth value is then multiplied by GL_DEPTH_SCALE and added to GL_DEPTH_BIAS. The result is clamped to the range [0,1].

The GL then converts the resulting depth components to fragments by attaching the current raster position color or color index and texture coordinates to each pixel, then assigning window coordinates (x_r+i,y_r+j), where (x_r,y_r) is the current raster position, and the pixel was the ith pixel in the jth row. These pixel fragments are then treated just like the fragments generated by rasterizing points, lines, or polygons. Texture mapping, fog, and all the fragment operations are applied before the fragments are written to the frame buffer.

GL_STENCIL

Stencil indices are read from the stencil buffer and converted to an internal fixed-point format with an unspecified number of bits to the right of the binary point. Each fixed-point index is then shifted left by GL_INDEX_SHIFT bits, and added to GL_INDEX_OFFSET. If GL_INDEX_SHIFT is negative, the shift is to the right. In either case, zero bits fill otherwise unspecified bit locations in the result. If GL_MAP_STENCIL is true, the index is replaced with the value that it references in lookup table GL_PIXEL_MAP_S_TO_S. Whether the lookup replacement of the index is done or not, the integer part of the index is then ANDed with 2^b-1, where b is the number of bits in the stencil buffer. The resulting stencil indices are then written to the stencil buffer such that the index read from the ith location of the jth row is written to location (x_r+i,y_r+j), where (x_r,y_r) is the current raster position. Only the pixel ownership test, the scissor test, and the stencil writemask affect these write operations.

The rasterization described thus far assumes pixel zoom factors of 1.0. If glPixelZoom is used to change the x and y pixel zoom factors, pixels are converted to fragments as follows. If (x_r,y_r) is the current raster position, and a given pixel is in the ith location in the jth row of the source pixel rectangle, then fragments are generated for pixels whose centers are in the rectangle with corners at

(x_r+$zoom\_x$,i,y_r+$zoom\_y$,j)

and

(x_r+$zoom\_x$,(i+1,),y_r+$zoom\_y$,(j+1,))

where $zoom\_x$ is the value of GL_ZOOM_X and $zoom\_y$ is the value of GL_ZOOM_Y.

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

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

GL_INVALID_OPERATION is generated if *type* is GL_DEPTH and there is no depth buffer.

GL_INVALID_OPERATION is generated if *type* is GL_STENCIL and there is no stencil buffer.

`GL_INVALID_OPERATION` is generated if `glCopyPixels` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glCopyTexImage1D` *target level internalformat x y width border* [Function]
Copy pixels into a 1D texture image.

target Specifies the target texture. Must be `GL_TEXTURE_1D`.

level Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.

internalformat
Specifies the internal format of the texture. Must be one of the following symbolic constants: `GL_ALPHA`, `GL_ALPHA4`, `GL_ALPHA8`, `GL_ALPHA12`, `GL_ALPHA16`, `GL_COMPRESSED_ALPHA`, `GL_COMPRESSED_LUMINANCE`, `GL_COMPRESSED_LUMINANCE_ALPHA`, `GL_COMPRESSED_INTENSITY`, `GL_COMPRESSED_RGB`, `GL_COMPRESSED_RGBA`, `GL_DEPTH_COMPONENT`, `GL_DEPTH_COMPONENT16`, `GL_DEPTH_COMPONENT24`, `GL_DEPTH_COMPONENT32`, `GL_LUMINANCE`, `GL_LUMINANCE4`, `GL_LUMINANCE8`, `GL_LUMINANCE12`, `GL_LUMINANCE16`, `GL_LUMINANCE_ALPHA`, `GL_LUMINANCE4_ALPHA4`, `GL_LUMINANCE6_ALPHA2`, `GL_LUMINANCE8_ALPHA8`, `GL_LUMINANCE12_ALPHA4`, `GL_LUMINANCE12_ALPHA12`, `GL_LUMINANCE16_ALPHA16`, `GL_INTENSITY`, `GL_INTENSITY4`, `GL_INTENSITY8`, `GL_INTENSITY12`, `GL_INTENSITY16`, `GL_RGB`, `GL_R3_G3_B2`, `GL_RGB4`, `GL_RGB5`, `GL_RGB8`, `GL_RGB10`, `GL_RGB12`, `GL_RGB16`, `GL_RGBA`, `GL_RGBA2`, `GL_RGBA4`, `GL_RGB5_A1`, `GL_RGBA8`, `GL_RGB10_A2`, `GL_RGBA12`, `GL_RGBA16`, `GL_SLUMINANCE`, `GL_SLUMINANCE8`, `GL_SLUMINANCE_ALPHA`, `GL_SLUMINANCE8_ALPHA8`, `GL_SRGB`, `GL_SRGB8`, `GL_SRGB_ALPHA`, or `GL_SRGB8_ALPHA8`.

x
y Specify the window coordinates of the left corner of the row of pixels to be copied.

width Specifies the width of the texture image. Must be 0 or 2^*n*+2(*border*,) for some integer *n*. The height of the texture image is 1.

border Specifies the width of the border. Must be either 0 or 1.

`glCopyTexImage1D` defines a one-dimensional texture image with pixels from the current `GL_READ_BUFFER`.

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

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

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

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

glCopyTexImage1D defines a one-dimensional texture image with pixels from the current GL_READ_BUFFER.

When *internalformat* is one of the sRGB types, the GL does not automatically convert the source pixels to the sRGB color space. In this case, the glPixelMap function can be used to accomplish the conversion.

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

GL_INVALID_VALUE is generated if *level* is less than 0.

GL_INVALID_VALUE may be generated if *level* is greater than *log_2max*, where *max* is the returned value of GL_MAX_TEXTURE_SIZE.

GL_INVALID_VALUE is generated if *internalformat* is not an allowable value.

GL_INVALID_VALUE is generated if *width* is less than 0 or greater than 2 + GL_MAX_TEXTURE_SIZE.

GL_INVALID_VALUE is generated if non-power-of-two textures are not supported and the *width* cannot be represented as 2^*n*+2(*border*,) for some integer value of *n*.

GL_INVALID_VALUE is generated if *border* is not 0 or 1.

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

GL_INVALID_OPERATION is generated if *internalformat* is GL_DEPTH_COMPONENT, GL_DEPTH_COMPONENT16, GL_DEPTH_COMPONENT24, or GL_DEPTH_COMPONENT32 and there is no depth buffer.

`void glCopyTexImage2D` *target level internalformat x y width height border* [Function]

Copy pixels into a 2D texture image.

target Specifies the target texture. Must be GL_TEXTURE_2D, GL_TEXTURE_CUBE_MAP_POSITIVE_X, GL_TEXTURE_CUBE_MAP_NEGATIVE_X, GL_TEXTURE_CUBE_MAP_POSITIVE_Y, GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, GL_TEXTURE_CUBE_MAP_POSITIVE_Z, or GL_TEXTURE_CUBE_MAP_NEGATIVE_Z.

level Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.

internalformat
Specifies the internal format of the texture. Must be one of the following symbolic constants: GL_ALPHA, GL_ALPHA4, GL_ALPHA8, GL_ALPHA12, GL_ALPHA16, GL_COMPRESSED_ALPHA, GL_COMPRESSED_LUMINANCE, GL_COMPRESSED_LUMINANCE_ALPHA, GL_COMPRESSED_INTENSITY, GL_COMPRESSED_RGB, GL_COMPRESSED_RGBA, GL_DEPTH_COMPONENT, GL_DEPTH_COMPONENT16, GL_DEPTH_COMPONENT24, GL_DEPTH_COMPONENT32, GL_LUMINANCE, GL_LUMINANCE4, GL_LUMINANCE8, GL_LUMINANCE12, GL_LUMINANCE16, GL_LUMINANCE_ALPHA, GL_LUMINANCE4_ALPHA4,

GL_LUMINANCE6_ALPHA2, GL_LUMINANCE8_ALPHA8, GL_LUMINANCE12_ALPHA4, GL_LUMINANCE12_ALPHA12, GL_LUMINANCE16_ALPHA16, GL_INTENSITY, GL_INTENSITY4, GL_INTENSITY8, GL_INTENSITY12, GL_INTENSITY16, GL_RGB, GL_R3_G3_B2, GL_RGB4, GL_RGB5, GL_RGB8, GL_RGB10, GL_RGB12, GL_RGB16, GL_RGBA, GL_RGBA2, GL_RGBA4, GL_RGB5_A1, GL_RGBA8, GL_RGB10_A2, GL_RGBA12, GL_RGBA16, GL_SLUMINANCE, GL_SLUMINANCE8, GL_SLUMINANCE_ALPHA, GL_SLUMINANCE8_ALPHA8, GL_SRGB, GL_SRGB8, GL_SRGB_ALPHA, or GL_SRGB8_ALPHA8.

x
y Specify the window coordinates of the lower left corner of the rectangular region of pixels to be copied.

width Specifies the width of the texture image. Must be 0 or 2^*n*+2(*border*,) for some integer *n*.

height Specifies the height of the texture image. Must be 0 or 2^*m*+2(*border*,) for some integer *m*.

border Specifies the width of the border. Must be either 0 or 1.

glCopyTexImage2D defines a two-dimensional texture image, or cube-map texture image with pixels from the current GL_READ_BUFFER.

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

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

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

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

When *internalformat* is one of the sRGB types, the GL does not automatically convert the source pixels to the sRGB color space. In this case, the glPixelMap function can be used to accomplish the conversion.

GL_INVALID_ENUM is generated if *target* is not GL_TEXTURE_2D, GL_TEXTURE_CUBE_MAP_POSITIVE_X, GL_TEXTURE_CUBE_MAP_NEGATIVE_X, GL_TEXTURE_CUBE_MAP_POSITIVE_Y, GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, GL_TEXTURE_CUBE_MAP_POSITIVE_Z, or GL_TEXTURE_CUBE_MAP_NEGATIVE_Z.

GL_INVALID_VALUE is generated if *level* is less than 0.

GL_INVALID_VALUE may be generated if *level* is greater than *log_2max*, where *max* is the returned value of GL_MAX_TEXTURE_SIZE.

`GL_INVALID_VALUE` is generated if *width* is less than 0 or greater than 2 + `GL_MAX_TEXTURE_SIZE`.

`GL_INVALID_VALUE` is generated if non-power-of-two textures are not supported and the *width* or *depth* cannot be represented as 2^*k*+2(*border*,) for some integer *k*.

`GL_INVALID_VALUE` is generated if *border* is not 0 or 1.

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

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

`GL_INVALID_OPERATION` is generated if *internalformat* is `GL_DEPTH_COMPONENT`, `GL_DEPTH_COMPONENT16`, `GL_DEPTH_COMPONENT24`, or `GL_DEPTH_COMPONENT32` and there is no depth buffer.

`void glCopyTexSubImage1D` *target level xoffset x y width* [Function]

Copy a one-dimensional texture subimage.

target Specifies the target texture. Must be `GL_TEXTURE_1D`.

level Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.

xoffset Specifies the texel offset within the texture array.

x
y Specify the window coordinates of the left corner of the row of pixels to be copied.

width Specifies the width of the texture subimage.

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

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

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

It is not an error to specify a subtexture with zero width, but such a specification has no effect. If any of the pixels within the specified row of the current `GL_READ_BUFFER` are outside the read window associated with the current rendering context, then the values obtained for those pixels are undefined.

No change is made to the *internalformat*, *width*, or *border* parameters of the specified texture array or to texel values outside the specified subregion.

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

`GL_INVALID_OPERATION` is generated if the texture array has not been defined by a previous `glTexImage1D` or `glCopyTexImage1D` operation.

`GL_INVALID_VALUE` is generated if *level* is less than 0.

`GL_INVALID_VALUE` may be generated if *level*>*log_2*(*max*,), where *max* is the returned value of `GL_MAX_TEXTURE_SIZE`.

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

`void glCopyTexSubImage2D` *target level xoffset yoffset x y width height* [Function]
Copy a two-dimensional texture subimage.

target Specifies the target texture. Must be `GL_TEXTURE_2D`, `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Z`, or `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`.

level Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.

xoffset Specifies a texel offset in the x direction within the texture array.

yoffset Specifies a texel offset in the y direction within the texture array.

x
y Specify the window coordinates of the lower left corner of the rectangular region of pixels to be copied.

width Specifies the width of the texture subimage.

height Specifies the height of the texture subimage.

`glCopyTexSubImage2D` replaces a rectangular portion of a two-dimensional texture image or cube-map texture image with pixels from the current `GL_READ_BUFFER` (rather than from main memory, as is the case for `glTexSubImage2D`).

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

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

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

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

No change is made to the *internalformat*, *width*, *height*, or *border* parameters of the specified texture array or to texel values outside the specified subregion.

`GL_INVALID_ENUM` is generated if *target* is not `GL_TEXTURE_2D`, `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Z`, or `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`.

`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*>*log_2*(*max*,), where *max* is the returned value of `GL_MAX_TEXTURE_SIZE`.

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

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

`void glCopyTexSubImage3D` *target level xoffset yoffset zoffset x y width height* [Function]

Copy a three-dimensional texture subimage.

target Specifies the target texture. Must be `GL_TEXTURE_3D`

level Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.

xoffset Specifies a texel offset in the x direction within the texture array.

yoffset Specifies a texel offset in the y direction within the texture array.

zoffset Specifies a texel offset in the z direction within the texture array.

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.

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

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

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

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

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

No change is made to the *internalformat*, *width*, *height*, *depth*, or *border* parameters of the specified texture array or to texel values outside the specified subregion.

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

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

`GL_INVALID_VALUE` is generated if *level* is less than 0.

`GL_INVALID_VALUE` may be generated if *level*>*log_2*(*max*,), where *max* is the returned value of `GL_MAX_3D_TEXTURE_SIZE`.

`GL_INVALID_VALUE` is generated if *xoffset*<-*b*, (*xoffset*+*width*,)>(*w*-*b*,), *yoffset*<-*b*, (*yoffset*+*height*,)>(*h*-*b*,), *zoffset*<-*b*, or (*zoffset*+1,)>(*d*-*b*,), where *w* is the `GL_TEXTURE_WIDTH`, *h* is the `GL_TEXTURE_HEIGHT`, *d* is the `GL_TEXTURE_DEPTH`, and *b* is the `GL_TEXTURE_BORDER` of the texture image being modified. Note that *w*, *h*, and *d* include twice the border width.

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

`GLuint glCreateProgram` [Function]

Creates a program object.

`glCreateProgram` creates an empty program object and returns a non-zero value by which it can be referenced. A program object is an object to which shader objects can be attached. This provides a mechanism to specify the shader objects that will be linked to create a program. It also provides a means for checking the compatibility of the shaders that will be used to create a program (for instance, checking the compatibility between a vertex shader and a fragment shader). When no longer needed as part of a program object, shader objects can be detached.

One or more executables are created in a program object by successfully attaching shader objects to it with `glAttachShader`, successfully compiling the shader objects with `glCompileShader`, and successfully linking the program object with `glLinkProgram`. These executables are made part of current state when `glUseProgram` is called. Program objects can be deleted by calling `glDeleteProgram`. The memory associated with the program object will be deleted when it is no longer part of current rendering state for any context.

This function returns 0 if an error occurs creating the program object.

`GL_INVALID_OPERATION` is generated if `glCreateProgram` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`GLuint glCreateShader` *shaderType* [Function]

Creates a shader object.

shaderType
: Specifies the type of shader to be created. Must be either `GL_VERTEX_SHADER` or `GL_FRAGMENT_SHADER`.

`glCreateShader` creates an empty shader object and returns a non-zero value by which it can be referenced. A shader object is used to maintain the source code strings that define a shader. *shaderType* indicates the type of shader to be created. Two types of shaders are supported. A shader of type `GL_VERTEX_SHADER` is a shader that is intended to run on the programmable vertex processor and replace the fixed functionality vertex processing in OpenGL. A shader of type `GL_FRAGMENT_SHADER` is a shader that is intended to run on the programmable fragment processor and replace the fixed functionality fragment processing in OpenGL.

When created, a shader object's `GL_SHADER_TYPE` parameter is set to either `GL_VERTEX_SHADER` or `GL_FRAGMENT_SHADER`, depending on the value of *shaderType*.

This function returns 0 if an error occurs creating the shader object.

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

`GL_INVALID_OPERATION` is generated if `glCreateShader` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glCullFace` *mode* [Function]

Specify whether front- or back-facing facets can be culled.

mode
: Specifies whether front- or back-facing facets are candidates for culling. Symbolic constants `GL_FRONT`, `GL_BACK`, and `GL_FRONT_AND_BACK` are accepted. The initial value is `GL_BACK`.

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

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

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

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

`void glDeleteBuffers` *n buffers* [Function]

Delete named buffer objects.

n
: Specifies the number of buffer objects to be deleted.

buffers
: Specifies an array of buffer objects to be deleted.

`glDeleteBuffers` deletes *n* buffer objects named by the elements of the array *buffers*. After a buffer object is deleted, it has no contents, and its name is free for reuse (for example by `glGenBuffers`). If a buffer object that is currently bound is deleted, the binding reverts to 0 (the absence of any buffer object, which reverts to client memory usage).

`glDeleteBuffers` silently ignores 0's and names that do not correspond to existing buffer objects.

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

`GL_INVALID_OPERATION` is generated if `glDeleteBuffers` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glDeleteLists` *list range* [Function]

Delete a contiguous group of display lists.

list Specifies the integer name of the first display list to delete.

range Specifies the number of display lists to delete.

`glDeleteLists` causes a contiguous group of display lists to be deleted. *list* is the name of the first display list to be deleted, and *range* is the number of display lists to delete. All display lists *d* with *list*<=*d*<=*list*+*range*-1 are deleted.

All storage locations allocated to the specified display lists are freed, and the names are available for reuse at a later time. Names within the range that do not have an associated display list are ignored. If *range* is 0, nothing happens.

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

`GL_INVALID_OPERATION` is generated if `glDeleteLists` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glDeleteProgram` *program* [Function]

Deletes a program object.

program Specifies the program object to be deleted.

`glDeleteProgram` frees the memory and invalidates the name associated with the program object specified by *program*. This command effectively undoes the effects of a call to `glCreateProgram`.

If a program object is in use as part of current rendering state, it will be flagged for deletion, but it will not be deleted until it is no longer part of current state for any rendering context. If a program object to be deleted has shader objects attached to it, those shader objects will be automatically detached but not deleted unless they have already been flagged for deletion by a previous call to `glDeleteShader`. A value of 0 for *program* will be silently ignored.

To determine whether a program object has been flagged for deletion, call `glGetProgram` with arguments *program* and `GL_DELETE_STATUS`.

`GL_INVALID_VALUE` is generated if *program* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if `glDeleteProgram` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glDeleteQueries` *n ids* [Function]

Delete named query objects.

n Specifies the number of query objects to be deleted.

ids Specifies an array of query objects to be deleted.

`glDeleteQueries` deletes *n* query objects named by the elements of the array *ids*. After a query object is deleted, it has no contents, and its name is free for reuse (for example by `glGenQueries`).

`glDeleteQueries` silently ignores 0's and names that do not correspond to existing query objects.

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

`GL_INVALID_OPERATION` is generated if `glDeleteQueries` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glDeleteShader` *shader* [Function]

Deletes a shader object.

shader Specifies the shader object to be deleted.

`glDeleteShader` frees the memory and invalidates the name associated with the shader object specified by *shader*. This command effectively undoes the effects of a call to `glCreateShader`.

If a shader object to be deleted is attached to a program object, it will be flagged for deletion, but it will not be deleted until it is no longer attached to any program object, for any rendering context (i.e., it must be detached from wherever it was attached before it will be deleted). A value of 0 for *shader* will be silently ignored.

To determine whether an object has been flagged for deletion, call `glGetShader` with arguments *shader* and `GL_DELETE_STATUS`.

`GL_INVALID_VALUE` is generated if *shader* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if `glDeleteShader` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glDeleteTextures` *n textures* [Function]

Delete named textures.

n Specifies the number of textures to be deleted.

textures Specifies an array of textures to be deleted.

`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.

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

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

`void glDepthFunc` *func* [Function]

Specify the value used for depth buffer comparisons.

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`.

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

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

`GL_NEVER` Never passes.

`GL_LESS` Passes if the incoming depth value is less than the stored depth value.

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

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

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

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

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

`GL_ALWAYS`
Always passes.

The initial value of *func* is `GL_LESS`. Initially, depth testing is disabled. If depth testing is disabled or if no depth buffer exists, it is as if the depth test always passes.

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

`GL_INVALID_OPERATION` is generated if `glDepthFunc` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glDepthMask` *flag* [Function]

Enable or disable writing into the depth buffer.

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

`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.

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

`void glDepthRange` *nearVal farVal* [Function]

Specify mapping of depth values from normalized device coordinates to window coordinates.

nearVal Specifies the mapping of the near clipping plane to window coordinates. The initial value is 0.

farVal Specifies the mapping of the far clipping plane to window coordinates. The initial value is 1.

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.

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

`void glDetachShader` *program shader* [Function]

Detaches a shader object from a program object to which it is attached.

program Specifies the program object from which to detach the shader object.

shader Specifies the shader object to be detached.

`glDetachShader` detaches the shader object specified by *shader* from the program object specified by *program*. This command can be used to undo the effect of the command `glAttachShader`.

If *shader* has already been flagged for deletion by a call to `glDeleteShader` and it is not attached to any other program object, it will be deleted after it has been detached.

`GL_INVALID_VALUE` is generated if either *program* or *shader* is a value that was not generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *program* is not a program object.

`GL_INVALID_OPERATION` is generated if *shader* is not a shader object.

`GL_INVALID_OPERATION` is generated if *shader* is not attached to *program*.

`GL_INVALID_OPERATION` is generated if `glDetachShader` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glDrawArrays` *mode first count* [Function]

Render primitives from array data.

mode Specifies what kind of primitives to render. Symbolic constants `GL_POINTS`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_LINES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_TRIANGLES`, `GL_QUAD_STRIP`, `GL_QUADS`, and `GL_POLYGON` are accepted.

first Specifies the starting index in the enabled arrays.

count Specifies the number of indices to be rendered.

`glDrawArrays` specifies multiple geometric primitives with very few subroutine calls. Instead of calling a GL procedure to pass each individual vertex, normal, texture coordinate, edge flag, or color, you can prespecify separate arrays of vertices, normals,

and colors and use them to construct a sequence of primitives with a single call to `glDrawArrays`.

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

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

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

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

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to an enabled array and the buffer object's data store is currently mapped.

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

`void glDrawBuffers` *n bufs* [Function]

Specifies a list of color buffers to be drawn into.

n Specifies the number of buffers in *bufs*.

bufs Points to an array of symbolic constants specifying the buffers into which fragment colors or data values will be written.

`glDrawBuffers` defines an array of buffers into which fragment color values or fragment data will be written. If no fragment shader is active, rendering operations will generate only one fragment color per fragment and it will be written into each of the buffers specified by *bufs*. If a fragment shader is active and it writes a value to the output variable `gl_FragColor`, then that value will be written into each of the buffers specified by *bufs*. If a fragment shader is active and it writes a value to one or more elements of the output array variable `gl_FragData[]`, then the value of `gl_FragData[0]` will be written into the first buffer specified by *bufs*, the value of `gl_FragData[1]` will be written into the second buffer specified by *bufs*, and so on up to `gl_FragData[n-1]`. The draw buffer used for `gl_FragData[n]` and beyond is implicitly set to be `GL_NONE`.

The symbolic constants contained in *bufs* may be any of the following:

`GL_NONE` The fragment color/data value is not written into any color buffer.

`GL_FRONT_LEFT`
The fragment color/data value is written into the front left color buffer.

`GL_FRONT_RIGHT`
The fragment color/data value is written into the front right color buffer.

`GL_BACK_LEFT`
The fragment color/data value is written into the back left color buffer.

`GL_BACK_RIGHT`
: The fragment color/data value is written into the back right color buffer.

`GL_AUXi`
: The fragment color/data value is written into auxiliary buffer `i`.

Except for `GL_NONE`, the preceding symbolic constants may not appear more than once in *bufs*. The maximum number of draw buffers supported is implementation dependent and can be queried by calling `glGet` with the argument `GL_MAX_DRAW_BUFFERS`. The number of auxiliary buffers can be queried by calling `glGet` with the argument `GL_AUX_BUFFERS`.

`GL_INVALID_ENUM` is generated if one of the values in *bufs* is not an accepted value.

`GL_INVALID_ENUM` is generated if *n* is less than 0.

`GL_INVALID_OPERATION` is generated if a symbolic constant other than `GL_NONE` appears more than once in *bufs*.

`GL_INVALID_OPERATION` is generated if any of the entries in *bufs* (other than `GL_NONE`) indicates a color buffer that does not exist in the current GL context.

`GL_INVALID_VALUE` is generated if *n* is greater than `GL_MAX_DRAW_BUFFERS`.

`GL_INVALID_OPERATION` is generated if `glDrawBuffers` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glDrawBuffer` *mode* [Function]

Specify which color buffers are to be drawn into.

mode
: Specifies up to four color buffers to be drawn into. Symbolic constants `GL_NONE`, `GL_FRONT_LEFT`, `GL_FRONT_RIGHT`, `GL_BACK_LEFT`, `GL_BACK_RIGHT`, `GL_FRONT`, `GL_BACK`, `GL_LEFT`, `GL_RIGHT`, `GL_FRONT_AND_BACK`, and `GL_AUX`*i*, where *i* is between 0 and the value of `GL_AUX_BUFFERS` minus 1, are accepted. (`GL_AUX_BUFFERS` is not the upper limit; use `glGet` to query the number of available aux buffers.) The initial value is `GL_FRONT` for single-buffered contexts, and `GL_BACK` for double-buffered contexts.

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

`GL_NONE`
: No color buffers are written.

`GL_FRONT_LEFT`
: Only the front left color buffer is written.

`GL_FRONT_RIGHT`
: Only the front right color buffer is written.

`GL_BACK_LEFT`
: Only the back left color buffer is written.

`GL_BACK_RIGHT`
: Only the back right color buffer is written.

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

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

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

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

`GL_FRONT_AND_BACK`
All the front and back color buffers (front left, front right, back left, back right) are written. If there are no back color buffers, only the front left and front right color buffers are written. If there are no right color buffers, only the front left and back left color buffers are written. If there are no right or back color buffers, only the front left color buffer is written.

`GL_AUX`*i* Only auxiliary color buffer *i* is written.

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

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

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

`GL_INVALID_OPERATION` is generated if none of the buffers indicated by *mode* exists.

`GL_INVALID_OPERATION` is generated if `glDrawBuffer` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glDrawElements` *mode count type indices* [Function]
Render primitives from array data.

mode Specifies what kind of primitives to render. Symbolic constants `GL_POINTS`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_LINES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_TRIANGLES`, `GL_QUAD_STRIP`, `GL_QUADS`, and `GL_POLYGON` are accepted.

count Specifies the number of elements to be rendered.

type Specifies the type of the values in *indices*. Must be one of `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT`, or `GL_UNSIGNED_INT`.

indices Specifies a pointer to the location where the indices are stored.

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

When `glDrawElements` is called, it uses *count* sequential elements from an enabled array, starting at *indices* to construct a sequence of geometric primitives. *mode*

specifies what kind of primitives are constructed and how the array elements construct these primitives. If more than one array is enabled, each is used. If `GL_VERTEX_ARRAY` is not enabled, no geometric primitives are constructed.

Vertex attributes that are modified by `glDrawElements` have an unspecified value after `glDrawElements` returns. For example, if `GL_COLOR_ARRAY` is enabled, the value of the current color is undefined after `glDrawElements` executes. Attributes that aren't modified maintain their previous values.

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

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

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to an enabled array or the element array and the buffer object's data store is currently mapped.

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

`void glDrawPixels` *width height format type data* [Function]

Write a block of pixels to the frame buffer.

width
height Specify the dimensions of the pixel rectangle to be written into the frame buffer.

format Specifies the format of the pixel data. Symbolic constants `GL_COLOR_INDEX`, `GL_STENCIL_INDEX`, `GL_DEPTH_COMPONENT`, `GL_RGB`, `GL_BGR`, `GL_RGBA`, `GL_BGRA`, `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`, `GL_LUMINANCE`, and `GL_LUMINANCE_ALPHA` are accepted.

type Specifies the data type for *data*. Symbolic constants `GL_UNSIGNED_BYTE`, `GL_BYTE`, `GL_BITMAP`, `GL_UNSIGNED_SHORT`, `GL_SHORT`, `GL_UNSIGNED_INT`, `GL_INT`, `GL_FLOAT`, `GL_UNSIGNED_BYTE_3_3_2`, `GL_UNSIGNED_BYTE_2_3_3_REV`, `GL_UNSIGNED_SHORT_5_6_5`, `GL_UNSIGNED_SHORT_5_6_5_REV`, `GL_UNSIGNED_SHORT_4_4_4_4`, `GL_UNSIGNED_SHORT_4_4_4_4_REV`, `GL_UNSIGNED_SHORT_5_5_5_1`, `GL_UNSIGNED_SHORT_1_5_5_5_REV`, `GL_UNSIGNED_INT_8_8_8_8`, `GL_UNSIGNED_INT_8_8_8_8_REV`, `GL_UNSIGNED_INT_10_10_10_2`, and `GL_UNSIGNED_INT_2_10_10_10_REV` are accepted.

data Specifies a pointer to the pixel data.

`glDrawPixels` reads pixel data from memory and writes it into the frame buffer relative to the current raster position, provided that the raster position is valid. Use `glRasterPos` or `glWindowPos` to set the current raster position; use `glGet` with argument `GL_CURRENT_RASTER_POSITION_VALID` to determine if the specified raster position is valid, and `glGet` with argument `GL_CURRENT_RASTER_POSITION` to query the raster position.

Several parameters define the encoding of pixel data in memory and control the processing of the pixel data before it is placed in the frame buffer. These parameters are set with four commands: `glPixelStore`, `glPixelTransfer`, `glPixelMap`, and

glPixelZoom. This reference page describes the effects on glDrawPixels of many, but not all, of the parameters specified by these four commands.

Data is read from *data* as a sequence of signed or unsigned bytes, signed or unsigned shorts, signed or unsigned integers, or single-precision floating-point values, depending on *type*. When *type* is one of GL_UNSIGNED_BYTE, GL_BYTE, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, or GL_FLOAT each of these bytes, shorts, integers, or floating-point values is interpreted as one color or depth component, or one index, depending on *format*. When *type* is one of GL_UNSIGNED_BYTE_3_3_2, GL_UNSIGNED_SHORT_5_6_5, GL_UNSIGNED_SHORT_4_4_4_4, GL_UNSIGNED_SHORT_5_5_5_1, GL_UNSIGNED_INT_8_8_8_8, or GL_UNSIGNED_INT_10_10_10_2, each unsigned value is interpreted as containing all the components for a single pixel, with the color components arranged according to *format*. When *type* is one of GL_UNSIGNED_BYTE_2_3_3_REV, GL_UNSIGNED_SHORT_5_6_5_REV, GL_UNSIGNED_SHORT_4_4_4_4_REV, GL_UNSIGNED_SHORT_1_5_5_5_REV, GL_UNSIGNED_INT_8_8_8_8_REV, or GL_UNSIGNED_INT_2_10_10_10_REV, each unsigned value is interpreted as containing all color components, specified by *format*, for a single pixel in a reversed order. Indices are always treated individually. Color components are treated as groups of one, two, three, or four values, again based on *format*. Both individual indices and groups of components are referred to as pixels. If *type* is GL_BITMAP, the data must be unsigned bytes, and *format* must be either GL_COLOR_INDEX or GL_STENCIL_INDEX. Each unsigned byte is treated as eight 1-bit pixels, with bit ordering determined by GL_UNPACK_LSB_FIRST (see glPixelStore).

widthheight pixels are read from memory, starting at location *data*. By default, these pixels are taken from adjacent memory locations, except that after all *width* pixels are read, the read pointer is advanced to the next four-byte boundary. The four-byte row alignment is specified by glPixelStore with argument GL_UNPACK_ALIGNMENT, and it can be set to one, two, four, or eight bytes. Other pixel store parameters specify different read pointer advancements, both before the first pixel is read and after all *width* pixels are read. See the glPixelStore reference page for details on these options.

If a non-zero named buffer object is bound to the GL_PIXEL_UNPACK_BUFFER target (see glBindBuffer) while a block of pixels is specified, *data* is treated as a byte offset into the buffer object's data store.

The *widthheight* pixels that are read from memory are each operated on in the same way, based on the values of several parameters specified by glPixelTransfer and glPixelMap. The details of these operations, as well as the target buffer into which the pixels are drawn, are specific to the format of the pixels, as specified by *format*. *format* can assume one of 13 symbolic values:

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

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

If the GL is in RGBA mode, the resulting index is converted to an RGBA pixel with the help of the GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, and GL_PIXEL_MAP_I_TO_A tables. If the GL is in color index mode, and if GL_MAP_COLOR is true, the index is replaced with the value that it references in lookup table GL_PIXEL_MAP_I_TO_I. Whether the lookup replacement of the index is done or not, the integer part of the index is then ANDed with 2^*b*-1, where *b* is the number of bits in a color index buffer.

The GL then converts the resulting indices or RGBA colors to fragments by attaching the current raster position *z* coordinate and texture coordinates to each pixel, then assigning *x* and *y* window coordinates to the *n*th fragment such that *x_n*=*x_r*+*n*%*width*_*y_n*=*y_r*+*n*/*width*,

where (*x_r*,*y_r*) is the current raster position. These pixel fragments are then treated just like the fragments generated by rasterizing points, lines, or polygons. Texture mapping, fog, and all the fragment operations are applied before the fragments are written to the frame buffer.

GL_STENCIL_INDEX

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

Each fixed-point index is then shifted left by GL_INDEX_SHIFT bits, and added to GL_INDEX_OFFSET. If GL_INDEX_SHIFT is negative, the shift is to the right. In either case, zero bits fill otherwise unspecified bit locations in the result. If GL_MAP_STENCIL is true, the index is replaced with the value that it references in lookup table GL_PIXEL_MAP_S_TO_S. Whether the lookup replacement of the index is done or not, the integer part of the index is then ANDed with 2^*b*-1, where *b* is the number of bits in the stencil buffer. The resulting stencil indices are then written to the stencil buffer such that the *n*th index is written to location

x_n=*x_r*+*n*%*width*_*y_n*=*y_r*+*n*/*width*,

where (*x_r*,*y_r*) is the current raster position. Only the pixel ownership test, the scissor test, and the stencil writemask affect these write operations.

GL_DEPTH_COMPONENT

Each pixel is a single-depth component. Floating-point data is converted directly to an internal floating-point format with unspecified precision. Signed integer data is mapped linearly to the internal floating-point format such that the most positive representable integer value maps to 1.0, and the most negative representable value maps to -1.0. Unsigned integer

data is mapped similarly: the largest integer value maps to 1.0, and 0 maps to 0.0. The resulting floating-point depth value is then multiplied by GL_DEPTH_SCALE and added to GL_DEPTH_BIAS. The result is clamped to the range [0,1].

The GL then converts the resulting depth components to fragments by attaching the current raster position color or color index and texture coordinates to each pixel, then assigning x and y window coordinates to the nth fragment such that

$x\_n = x\_r + n\%width$
$y\_n = y\_r + n/width$,

where (x_r, y_r) is the current raster position. These pixel fragments are then treated just like the fragments generated by rasterizing points, lines, or polygons. Texture mapping, fog, and all the fragment operations are applied before the fragments are written to the frame buffer.

GL_RGBA

GL_BGRA Each pixel is a four-component group: For GL_RGBA, the red component is first, followed by green, followed by blue, followed by alpha; for GL_BGRA the order is blue, green, red and then alpha. Floating-point values are converted directly to an internal floating-point format with unspecified precision. Signed integer values are mapped linearly to the internal floating-point format such that the most positive representable integer value maps to 1.0, and the most negative representable value maps to -1.0. (Note that this mapping does not convert 0 precisely to 0.0.) Unsigned integer data is mapped similarly: The largest integer value maps to 1.0, and 0 maps to 0.0. The resulting floating-point color values are then multiplied by GL_c_SCALE and added to GL_c_BIAS, where c is RED, GREEN, BLUE, and ALPHA for the respective color components. The results are clamped to the range [0,1].

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

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

$x\_n = x\_r + n\%width$
$y\_n = y\_r + n/width$,

where (x_r, y_r) is the current raster position. These pixel fragments are then treated just like the fragments generated by rasterizing points, lines, or polygons. Texture mapping, fog, and all the fragment operations are applied before the fragments are written to the frame buffer.

GL_RED Each pixel is a single red component. This component is converted to the internal floating-point format in the same way the red component of an RGBA pixel is. It is then converted to an RGBA pixel with green and blue set to 0, and alpha set to 1. After this conversion, the pixel is treated as if it had been read as an RGBA pixel.

`GL_GREEN` Each pixel is a single green component. This component is converted to the internal floating-point format in the same way the green component of an RGBA pixel is. It is then converted to an RGBA pixel with red and blue set to 0, and alpha set to 1. After this conversion, the pixel is treated as if it had been read as an RGBA pixel.

`GL_BLUE` Each pixel is a single blue component. This component is converted to the internal floating-point format in the same way the blue component of an RGBA pixel is. It is then converted to an RGBA pixel with red and green set to 0, and alpha set to 1. After this conversion, the pixel is treated as if it had been read as an RGBA pixel.

`GL_ALPHA` Each pixel is a single alpha component. This component is converted to the internal floating-point format in the same way the alpha component of an RGBA pixel is. It is then converted to an RGBA pixel with red, green, and blue set to 0. After this conversion, the pixel is treated as if it had been read as an RGBA pixel.

`GL_RGB`

`GL_BGR` Each pixel is a three-component group: red first, followed by green, followed by blue; for `GL_BGR`, the first component is blue, followed by green and then red. Each component is converted to the internal floating-point format in the same way the red, green, and blue components of an RGBA pixel are. The color triple is converted to an RGBA pixel with alpha set to 1. After this conversion, the pixel is treated as if it had been read as an RGBA pixel.

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

`GL_LUMINANCE_ALPHA`
Each pixel is a two-component group: luminance first, followed by alpha. The two components are converted to the internal floating-point format in the same way the red component of an RGBA pixel is. They are then converted to an RGBA pixel with red, green, and blue set to the converted luminance value, and alpha set to the converted alpha value. After this conversion, the pixel is treated as if it had been read as an RGBA pixel.

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

Type	**Corresponding Type**
`GL_UNSIGNED_BYTE`	unsigned 8-bit integer
`GL_BYTE`	signed 8-bit integer

`GL_BITMAP`
: single bits in unsigned 8-bit integers

`GL_UNSIGNED_SHORT`
: unsigned 16-bit integer

`GL_SHORT` signed 16-bit integer

`GL_UNSIGNED_INT`
: unsigned 32-bit integer

`GL_INT` 32-bit integer

`GL_FLOAT` single-precision floating-point

`GL_UNSIGNED_BYTE_3_3_2`
: unsigned 8-bit integer

`GL_UNSIGNED_BYTE_2_3_3_REV`
: unsigned 8-bit integer with reversed component ordering

`GL_UNSIGNED_SHORT_5_6_5`
: unsigned 16-bit integer

`GL_UNSIGNED_SHORT_5_6_5_REV`
: unsigned 16-bit integer with reversed component ordering

`GL_UNSIGNED_SHORT_4_4_4_4`
: unsigned 16-bit integer

`GL_UNSIGNED_SHORT_4_4_4_4_REV`
: unsigned 16-bit integer with reversed component ordering

`GL_UNSIGNED_SHORT_5_5_5_1`
: unsigned 16-bit integer

`GL_UNSIGNED_SHORT_1_5_5_5_REV`
: unsigned 16-bit integer with reversed component ordering

`GL_UNSIGNED_INT_8_8_8_8`
: unsigned 32-bit integer

`GL_UNSIGNED_INT_8_8_8_8_REV`
: unsigned 32-bit integer with reversed component ordering

`GL_UNSIGNED_INT_10_10_10_2`
: unsigned 32-bit integer

`GL_UNSIGNED_INT_2_10_10_10_REV`
: unsigned 32-bit integer with reversed component ordering

The rasterization described so far assumes pixel zoom factors of 1. If `glPixelZoom` is used to change the x and y pixel zoom factors, pixels are converted to fragments as follows. If (x_r,y_r) is the current raster position, and a given pixel is in the nth column and mth row of the pixel rectangle, then fragments are generated for pixels whose centers are in the rectangle with corners at

$(x\_r+zoom\_x,n,y\_r+zoom\_y,m)(x\_r+zoom\_x,(n+1,),y\_r+zoom\_y,(m+1,))$

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

GL_INVALID_ENUM is generated if *format* or *type* is not one of the accepted values.

GL_INVALID_ENUM is generated if *type* is GL_BITMAP and *format* is not either GL_COLOR_INDEX or GL_STENCIL_INDEX.

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

GL_INVALID_OPERATION is generated if *format* is GL_STENCIL_INDEX and there is no stencil buffer.

GL_INVALID_OPERATION is generated if *format* is GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_BGR, GL_BGRA, GL_LUMINANCE, or GL_LUMINANCE_ALPHA, and the GL is in color index mode.

GL_INVALID_OPERATION is generated if *format* is one of GL_UNSIGNED_BYTE_3_3_2, GL_UNSIGNED_BYTE_2_3_3_REV, GL_UNSIGNED_SHORT_5_6_5, or GL_UNSIGNED_SHORT_5_6_5_REV and *format* is not GL_RGB.

GL_INVALID_OPERATION is generated if *format* is one of GL_UNSIGNED_SHORT_4_4_4_4, GL_UNSIGNED_SHORT_4_4_4_4_REV, GL_UNSIGNED_SHORT_5_5_5_1, GL_UNSIGNED_SHORT_1_5_5_5_REV, GL_UNSIGNED_INT_8_8_8_8, GL_UNSIGNED_INT_8_8_8_8_REV, GL_UNSIGNED_INT_10_10_10_2, or GL_UNSIGNED_INT_2_10_10_10_REV and *format* is neither GL_RGBA nor GL_BGRA.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and the buffer object's data store is currently mapped.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and *data* is not evenly divisible into the number of bytes needed to store in memory a datum indicated by *type*.

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

`void glDrawRangeElements` *mode start end count type indices* [Function]

Render primitives from array data.

mode Specifies what kind of primitives to render. Symbolic constants GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES, GL_QUAD_STRIP, GL_QUADS, and GL_POLYGON are accepted.

start Specifies the minimum array index contained in *indices*.

end Specifies the maximum array index contained in *indices*.

count Specifies the number of elements to be rendered.

type Specifies the type of the values in *indices*. Must be one of GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, or GL_UNSIGNED_INT.

indices Specifies a pointer to the location where the indices are stored.

`glDrawRangeElements` is a restricted form of `glDrawElements`. *mode*, *start*, *end*, and *count* match the corresponding arguments to `glDrawElements`, with the additional constraint that all values in the arrays *count* must lie between *start* and *end*, inclusive.

Implementations denote recommended maximum amounts of vertex and index data, which may be queried by calling `glGet` with argument `GL_MAX_ELEMENTS_VERTICES` and `GL_MAX_ELEMENTS_INDICES`. If *end*-*start*+1 is greater than the value of `GL_MAX_ELEMENTS_VERTICES`, or if *count* is greater than the value of `GL_MAX_ELEMENTS_INDICES`, then the call may operate at reduced performance. There is no requirement that all vertices in the range [*start*,*end*] be referenced. However, the implementation may partially process unused vertices, reducing performance from what could be achieved with an optimal index set.

When `glDrawRangeElements` is called, it uses *count* sequential elements from an enabled array, starting at *start* to construct a sequence of geometric primitives. *mode* specifies what kind of primitives are constructed, and how the array elements construct these primitives. If more than one array is enabled, each is used. If `GL_VERTEX_ARRAY` is not enabled, no geometric primitives are constructed.

Vertex attributes that are modified by `glDrawRangeElements` have an unspecified value after `glDrawRangeElements` returns. For example, if `GL_COLOR_ARRAY` is enabled, the value of the current color is undefined after `glDrawRangeElements` executes. Attributes that aren't modified maintain their previous values.

It is an error for indices to lie outside the range [*start*,*end*], but implementations may not check for this situation. Such indices cause implementation-dependent behavior.

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

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

`GL_INVALID_VALUE` is generated if *end*<*start*.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to an enabled array or the element array and the buffer object's data store is currently mapped.

`GL_INVALID_OPERATION` is generated if `glDrawRangeElements` is executed between the execution of `glBegin` and the corresponding `glEnd`.

`void glEdgeFlagPointer` *stride pointer* [Function]

Define an array of edge flags.

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

pointer Specifies a pointer to the first edge flag in the array. The initial value is 0.

`glEdgeFlagPointer` specifies the location and data format of an array of boolean edge flags to use when rendering. *stride* specifies the byte stride from one edge flag to the next, allowing vertices and attributes to be packed into a single array or stored in separate arrays.

If a non-zero named buffer object is bound to the `GL_ARRAY_BUFFER` target (see `glBindBuffer`) while an edge flag array is specified, *pointer* is treated as a byte

offset into the buffer object's data store. Also, the buffer object binding (GL_ARRAY_BUFFER_BINDING) is saved as edge flag vertex array client-side state (GL_EDGE_FLAG_ARRAY_BUFFER_BINDING).

When an edge flag array is specified, *stride* and *pointer* are saved as client-side state, in addition to the current vertex array buffer object binding.

To enable and disable the edge flag array, call glEnableClientState and glDisableClientState with the argument GL_EDGE_FLAG_ARRAY. If enabled, the edge flag array is used when glDrawArrays, glMultiDrawArrays, glDrawElements, glMultiDrawElements, glDrawRangeElements, or glArrayElement is called.

GL_INVALID_ENUM is generated if *stride* is negative.

void glEdgeFlag *flag* [Function]
void glEdgeFlagv *flag* [Function]

Flag edges as either boundary or nonboundary.

flag Specifies the current edge flag value, either GL_TRUE or GL_FALSE. The initial value is GL_TRUE.

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

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

Boundary and nonboundary edge flags on vertices are significant only if GL_POLYGON_MODE is set to GL_POINT or GL_LINE. See glPolygonMode.

void glEnableClientState *cap* [Function]
void glDisableClientState *cap* [Function]

Enable or disable client-side capability.

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

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

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

`GL_EDGE_FLAG_ARRAY`
: If enabled, the edge flag array is enabled for writing and used during rendering when `glArrayElement`, `glDrawArrays`, `glDrawElements`, `glDrawRangeElementsglMultiDrawArrays`, or `glMultiDrawElements` is called. See `glEdgeFlagPointer`.

`GL_FOG_COORD_ARRAY`
: If enabled, the fog coordinate array is enabled for writing and used during rendering when `glArrayElement`, `glDrawArrays`, `glDrawElements`, `glDrawRangeElementsglMultiDrawArrays`, or `glMultiDrawElements` is called. See `glFogCoordPointer`.

`GL_INDEX_ARRAY`
: If enabled, the index array is enabled for writing and used during rendering when `glArrayElement`, `glDrawArrays`, `glDrawElements`, `glDrawRangeElementsglMultiDrawArrays`, or `glMultiDrawElements` is called. See `glIndexPointer`.

`GL_NORMAL_ARRAY`
: If enabled, the normal array is enabled for writing and used during rendering when `glArrayElement`, `glDrawArrays`, `glDrawElements`, `glDrawRangeElementsglMultiDrawArrays`, or `glMultiDrawElements` is called. See `glNormalPointer`.

`GL_SECONDARY_COLOR_ARRAY`
: If enabled, the secondary color array is enabled for writing and used during rendering when `glArrayElement`, `glDrawArrays`, `glDrawElements`, `glDrawRangeElementsglMultiDrawArrays`, or `glMultiDrawElements` is called. See `glColorPointer`.

`GL_TEXTURE_COORD_ARRAY`
: If enabled, the texture coordinate array is enabled for writing and used during rendering when `glArrayElement`, `glDrawArrays`, `glDrawElements`, `glDrawRangeElementsglMultiDrawArrays`, or `glMultiDrawElements` is called. See `glTexCoordPointer`.

`GL_VERTEX_ARRAY`
: If enabled, the vertex array is enabled for writing and used during rendering when `glArrayElement`, `glDrawArrays`, `glDrawElements`, `glDrawRangeElementsglMultiDrawArrays`, or `glMultiDrawElements` is called. See `glVertexPointer`.

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

`glEnableClientState` is not allowed between the execution of `glBegin` and the corresponding `glEnd`, but an error may or may not be generated. If no error is generated, the behavior is undefined.

`void glEnableVertexAttribArray` *index* [Function]
`void glDisableVertexAttribArray` *index* [Function]
Enable or disable a generic vertex attribute array.

index Specifies the index of the generic vertex attribute to be enabled or disabled.

`glEnableVertexAttribArray` enables the generic vertex attribute array specified by *index*. `glDisableVertexAttribArray` disables the generic vertex attribute array specified by *index*. By default, all client-side capabilities are disabled, including all generic vertex attribute arrays. If enabled, the values in the generic vertex attribute array will be accessed and used for rendering when calls are made to vertex array commands such as `glDrawArrays`, `glDrawElements`, `glDrawRangeElements`, `glArrayElement`, `glMultiDrawElements`, or `glMultiDrawArrays`.

`GL_INVALID_VALUE` is generated if *index* is greater than or equal to `GL_MAX_VERTEX_ATTRIBS`.

`GL_INVALID_OPERATION` is generated if either `glEnableVertexAttribArray` or `glDisableVertexAttribArray` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glEnable` *cap* [Function]
`void glDisable` *cap* [Function]

Enable or disable server-side GL capabilities.

cap Specifies a symbolic constant indicating a GL capability.

`glEnable` and `glDisable` enable and disable various capabilities. Use `glIsEnabled` or `glGet` to determine the current setting of any capability. The initial value for each capability with the exception of `GL_DITHER` 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_AUTO_NORMAL`
If enabled, generate normal vectors when either `GL_MAP2_VERTEX_3` or `GL_MAP2_VERTEX_4` is used to generate vertices. See `glMap2`.

`GL_BLEND`
If enabled, blend the computed fragment color values with the values in the color buffers. See `glBlendFunc`.

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

`GL_COLOR_LOGIC_OP`
If enabled, apply the currently selected logical operation to the computed fragment color and color buffer values. See `glLogicOp`.

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

`GL_COLOR_SUM`
: If enabled and no fragment shader is active, add the secondary color value to the computed fragment color. See `glSecondaryColor`.

`GL_COLOR_TABLE`
: If enabled, perform a color table lookup on the incoming RGBA color values. See `glColorTable`.

`GL_CONVOLUTION_1D`
: If enabled, perform a 1D convolution operation on incoming RGBA color values. See `glConvolutionFilter1D`.

`GL_CONVOLUTION_2D`
: If enabled, perform a 2D convolution operation on incoming RGBA color values. See `glConvolutionFilter2D`.

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

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

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

`GL_FOG`
: If enabled and no fragment shader is active, blend a fog color into the post-texturing color. See `glFog`.

`GL_HISTOGRAM`
: If enabled, histogram incoming RGBA color values. See `glHistogram`.

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

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

`GL_LIGHTING`
: If enabled and no vertex shader is active, use the current lighting parameters to compute the vertex color or index. Otherwise, simply associate the current color or index with each vertex. See `glMaterial`, `glLightModel`, and `glLight`.

`GL_LINE_SMOOTH`
: If enabled, draw lines with correct filtering. Otherwise, draw aliased lines. See `glLineWidth`.

`GL_LINE_STIPPLE`
: If enabled, use the current line stipple pattern when drawing lines. See `glLineStipple`.

`GL_MAP1_COLOR_4`
: If enabled, calls to `glEvalCoord1`, `glEvalMesh1`, and `glEvalPoint1` generate RGBA values. See `glMap1`.

`GL_MAP1_INDEX`
: If enabled, calls to `glEvalCoord1`, `glEvalMesh1`, and `glEvalPoint1` generate color indices. See `glMap1`.

`GL_MAP1_NORMAL`
: If enabled, calls to `glEvalCoord1`, `glEvalMesh1`, and `glEvalPoint1` generate normals. See `glMap1`.

`GL_MAP1_TEXTURE_COORD_1`
: If enabled, calls to `glEvalCoord1`, `glEvalMesh1`, and `glEvalPoint1` generate s texture coordinates. See `glMap1`.

`GL_MAP1_TEXTURE_COORD_2`
: If enabled, calls to `glEvalCoord1`, `glEvalMesh1`, and `glEvalPoint1` generate s and t texture coordinates. See `glMap1`.

`GL_MAP1_TEXTURE_COORD_3`
: If enabled, calls to `glEvalCoord1`, `glEvalMesh1`, and `glEvalPoint1` generate s, t, and r texture coordinates. See `glMap1`.

`GL_MAP1_TEXTURE_COORD_4`
: If enabled, calls to `glEvalCoord1`, `glEvalMesh1`, and `glEvalPoint1` generate s, t, r, and q texture coordinates. See `glMap1`.

`GL_MAP1_VERTEX_3`
: If enabled, calls to `glEvalCoord1`, `glEvalMesh1`, and `glEvalPoint1` generate x, y, and z vertex coordinates. See `glMap1`.

`GL_MAP1_VERTEX_4`
: If enabled, calls to `glEvalCoord1`, `glEvalMesh1`, and `glEvalPoint1` generate homogeneous x, y, z, and w vertex coordinates. See `glMap1`.

`GL_MAP2_COLOR_4`
: If enabled, calls to `glEvalCoord2`, `glEvalMesh2`, and `glEvalPoint2` generate RGBA values. See `glMap2`.

`GL_MAP2_INDEX`
: If enabled, calls to `glEvalCoord2`, `glEvalMesh2`, and `glEvalPoint2` generate color indices. See `glMap2`.

`GL_MAP2_NORMAL`
: If enabled, calls to `glEvalCoord2`, `glEvalMesh2`, and `glEvalPoint2` generate normals. See `glMap2`.

`GL_MAP2_TEXTURE_COORD_1`
: If enabled, calls to `glEvalCoord2`, `glEvalMesh2`, and `glEvalPoint2` generate s texture coordinates. See `glMap2`.

`GL_MAP2_TEXTURE_COORD_2`
: If enabled, calls to `glEvalCoord2`, `glEvalMesh2`, and `glEvalPoint2` generate s and t texture coordinates. See `glMap2`.

`GL_MAP2_TEXTURE_COORD_3`
: If enabled, calls to `glEvalCoord2`, `glEvalMesh2`, and `glEvalPoint2` generate s, t, and r texture coordinates. See `glMap2`.

`GL_MAP2_TEXTURE_COORD_4`
: If enabled, calls to `glEvalCoord2`, `glEvalMesh2`, and `glEvalPoint2` generate s, t, r, and q texture coordinates. See `glMap2`.

`GL_MAP2_VERTEX_3`
: If enabled, calls to `glEvalCoord2`, `glEvalMesh2`, and `glEvalPoint2` generate x, y, and z vertex coordinates. See `glMap2`.

`GL_MAP2_VERTEX_4`
: If enabled, calls to `glEvalCoord2`, `glEvalMesh2`, and `glEvalPoint2` generate homogeneous x, y, z, and w vertex coordinates. See `glMap2`.

`GL_MINMAX`
: If enabled, compute the minimum and maximum values of incoming RGBA color values. See `glMinmax`.

`GL_MULTISAMPLE`
: If enabled, use multiple fragment samples in computing the final color of a pixel. See `glSampleCoverage`.

`GL_NORMALIZE`
: If enabled and no vertex shader is active, normal vectors are normalized to unit length after transformation and before lighting. This method is generally less efficient than `GL_RESCALE_NORMAL`. See `glNormal` and `glNormalPointer`.

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

`GL_POINT_SPRITE`
: If enabled, calculate texture coordinates for points based on texture environment and point parameter settings. Otherwise texture coordinates are constant across points.

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

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

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

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

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

`GL_POST_COLOR_MATRIX_COLOR_TABLE`
: If enabled, perform a color table lookup on RGBA color values after color matrix transformation. See `glColorTable`.

`GL_POST_CONVOLUTION_COLOR_TABLE`
: If enabled, perform a color table lookup on RGBA color values after convolution. See `glColorTable`.

`GL_RESCALE_NORMAL`
: If enabled and no vertex shader is active, normal vectors are scaled after transformation and before lighting by a factor computed from the modelview matrix. If the modelview matrix scales space uniformly, this has the effect of restoring the transformed normal to unit length. This method is generally more efficient than `GL_NORMALIZE`. See `glNormal` and `glNormalPointer`.

`GL_SAMPLE_ALPHA_TO_COVERAGE`
: If enabled, compute a temporary coverage value where each bit is determined by the alpha value at the corresponding sample location. The temporary coverage value is then ANDed with the fragment coverage value.

`GL_SAMPLE_ALPHA_TO_ONE`
: If enabled, each sample alpha value is replaced by the maximum representable alpha value.

`GL_SAMPLE_COVERAGE`
: If enabled, the fragment's coverage is ANDed with the temporary coverage value. If `GL_SAMPLE_COVERAGE_INVERT` is set to `GL_TRUE`, invert the coverage value. See `glSampleCoverage`.

`GL_SEPARABLE_2D`
: If enabled, perform a two-dimensional convolution operation using a separable convolution filter on incoming RGBA color values. See `glSeparableFilter2D`.

`GL_SCISSOR_TEST`
: If enabled, discard fragments that are outside the scissor rectangle. See `glScissor`.

`GL_STENCIL_TEST`
: If enabled, do stencil testing and update the stencil buffer. See `glStencilFunc` and `glStencilOp`.

`GL_TEXTURE_1D`
: If enabled and no fragment shader is active, one-dimensional texturing is performed (unless two- or three-dimensional or cube-mapped texturing is also enabled). See `glTexImage1D`.

`GL_TEXTURE_2D`
: If enabled and no fragment shader is active, two-dimensional texturing is performed (unless three-dimensional or cube-mapped texturing is also enabled). See `glTexImage2D`.

`GL_TEXTURE_3D`
: If enabled and no fragment shader is active, three-dimensional texturing is performed (unless cube-mapped texturing is also enabled). See `glTexImage3D`.

`GL_TEXTURE_CUBE_MAP`
: If enabled and no fragment shader is active, cube-mapped texturing is performed. See `glTexImage2D`.

`GL_TEXTURE_GEN_Q`
: If enabled and no vertex shader is active, the q texture coordinate is computed using the texture generation function defined with `glTexGen`. Otherwise, the current q texture coordinate is used. See `glTexGen`.

`GL_TEXTURE_GEN_R`
: If enabled and no vertex shader is active, the r texture coordinate is computed using the texture generation function defined with `glTexGen`. Otherwise, the current r texture coordinate is used. See `glTexGen`.

`GL_TEXTURE_GEN_S`
: If enabled and no vertex shader is active, the s texture coordinate is computed using the texture generation function defined with `glTexGen`. Otherwise, the current s texture coordinate is used. See `glTexGen`.

`GL_TEXTURE_GEN_T`
: If enabled and no vertex shader is active, the t texture coordinate is computed using the texture generation function defined with `glTexGen`. Otherwise, the current t texture coordinate is used. See `glTexGen`.

`GL_VERTEX_PROGRAM_POINT_SIZE`
: If enabled and a vertex shader is active, then the derived point size is taken from the (potentially clipped) shader builtin `gl_PointSize` and clamped to the implementation-dependent point size range.

`GL_VERTEX_PROGRAM_TWO_SIDE`
: If enabled and a vertex shader is active, it specifies that the GL will choose between front and back colors based on the polygon's face direction of which the vertex being shaded is a part. It has no effect on points or lines.

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

`GL_INVALID_OPERATION` is generated if `glEnable` or `glDisable` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glEvalCoord1f` *u* [Function]
`void glEvalCoord1d` *u* [Function]
`void glEvalCoord2f` *u v* [Function]
`void glEvalCoord2d` *u v* [Function]
`void glEvalCoord1fv` *u* [Function]
`void glEvalCoord1dv` *u* [Function]
`void glEvalCoord2fv` *u* [Function]
`void glEvalCoord2dv` *u* [Function]

Evaluate enabled one- and two-dimensional maps.

u Specifies a value that is the domain coordinate *u* to the basis function defined in a previous `glMap1` or `glMap2` command.

v Specifies a value that is the domain coordinate *v* to the basis function defined in a previous `glMap2` command. This argument is not present in a `glEvalCoord1` command.

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

When one of the `glEvalCoord` commands is issued, all currently enabled maps of the indicated dimension are evaluated. Then, for each enabled map, it is as if the corresponding GL command had been issued with the computed value. That is, if `GL_MAP1_INDEX` or `GL_MAP2_INDEX` is enabled, a `glIndex` command is simulated. If `GL_MAP1_COLOR_4` or `GL_MAP2_COLOR_4` is enabled, a `glColor` command is simulated. If `GL_MAP1_NORMAL` or `GL_MAP2_NORMAL` is enabled, a normal vector is produced, and if any of `GL_MAP1_TEXTURE_COORD_1`, `GL_MAP1_TEXTURE_COORD_2`, `GL_MAP1_TEXTURE_COORD_3`, `GL_MAP1_TEXTURE_COORD_4`, `GL_MAP2_TEXTURE_COORD_1`, `GL_MAP2_TEXTURE_COORD_2`, `GL_MAP2_TEXTURE_COORD_3`, or `GL_MAP2_TEXTURE_COORD_4` is enabled, then an appropriate `glTexCoord` command is simulated.

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

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

four-component vertex map is enabled for the specified dimension, the `glEvalCoord` command is ignored.

If you have enabled automatic normal generation, by calling `glEnable` with argument `GL_AUTO_NORMAL`, `glEvalCoord2` generates surface normals analytically, regardless of the contents or enabling of the `GL_MAP2_NORMAL` map. Let

`m`=`p`,/*u*,,`p`,/*v*,,

Then the generated normal `n` is `n`=`m`/`m`,,

If automatic normal generation is disabled, the corresponding normal map `GL_MAP2_NORMAL`, if enabled, is used to produce a normal. If neither automatic normal generation nor a normal map is enabled, no normal is generated for `glEvalCoord2` commands.

`void glEvalMesh1` *mode i1 i2* [Function]
`void glEvalMesh2` *mode i1 i2 j1 j2* [Function]

Compute a one- or two-dimensional grid of points or lines.

mode In `glEvalMesh1`, specifies whether to compute a one-dimensional mesh of points or lines. Symbolic constants `GL_POINT` and `GL_LINE` are accepted.

i1
i2 Specify the first and last integer values for grid domain variable *i*.

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

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

where

```
glBegin( type );
for ( i = i1; i <= i2; i += 1 )
   glEvalCoord1( iu+u_1 );
glEnd();
```

u=(*u*_2-*u*_1,)/*n*

and *n*, *u*_1, and *u*_2 are the arguments to the most recent `glMapGrid1` command. *type* is `GL_POINTS` if *mode* is `GL_POINT`, or `GL_LINES` if *mode* is `GL_LINE`.

The one absolute numeric requirement is that if *i*=*n*, then the value computed from *iu*+*u*_1 is exactly *u*_2.

In the two-dimensional case, `glEvalMesh2`, let .cp *u*=(*u*_2-*u*_1,)/*n*

v=(*v*_2-*v*_1,)/*m*

where *n*, *u*_1, *u*_2, *m*, *v*_1, and *v*_2 are the arguments to the most recent `glMapGrid2` command. Then, if *mode* is `GL_FILL`, the `glEvalMesh2` command is equivalent to:

```
for ( j = j1; j < j2; j += 1 ) {
   glBegin( GL_QUAD_STRIP );
   for ( i = i1; i <= i2; i += 1 ) {
      glEvalCoord2( iu+u_1,jv+v_1 );
      glEvalCoord2( iu+u_1,(j+1,)v+v_1 );
   }
   glEnd();
}
```

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

```
for ( j = j1; j <= j2; j += 1 ) {
   glBegin( GL_LINE_STRIP );
   for ( i = i1; i <= i2; i += 1 )
      glEvalCoord2( iu+u_1,jv+v_1 );
   glEnd();
}

for ( i = i1;  i <= i2; i += 1 ) {
   glBegin( GL_LINE_STRIP );
   for ( j = j1; j <= j1; j += 1 )
      glEvalCoord2( iu+u_1,jv+v_1 );
   glEnd();
}
```

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

```
glBegin( GL_POINTS );
for ( j = j1; j <= j2; j += 1 )
   for ( i = i1; i <= i2; i += 1 )
      glEvalCoord2( iu+u_1,jv+v_1 );
glEnd();
```

In all three cases, the only absolute numeric requirements are that if i=n, then the value computed from iu+u_1 is exactly u_2, and if j=m, then the value computed from jv+v_1 is exactly v_2.

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

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

`void glEvalPoint1` *i* [Function]
`void glEvalPoint2` *i j* [Function]

Generate and evaluate a single point in a mesh.

i Specifies the integer value for grid domain variable *i*.

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

`glMapGrid` and `glEvalMesh` are used in tandem to efficiently generate and evaluate a series of evenly spaced map domain values. `glEvalPoint` can be used to evaluate

a single grid point in the same gridspace that is traversed by `glEvalMesh`. Calling `glEvalPoint1` is equivalent to calling where u=(u_2-u_1,)/n

```
glEvalCoord1( iu+u_1 );
```

and n, u_1, and u_2 are the arguments to the most recent `glMapGrid1` command. The one absolute numeric requirement is that if i=n, then the value computed from iu+u_1 is exactly u_2.

In the two-dimensional case, `glEvalPoint2`, let

u=(u_2-u_1,)/nv=(v_2-v_1,)/m

where n, u_1, u_2, m, v_1, and v_2 are the arguments to the most recent `glMapGrid2` command. Then the `glEvalPoint2` command is equivalent to calling The only absolute numeric requirements are that if i=n, then the value computed from iu+u_1 is exactly u_2, and if j=m, then the value computed from jv+v_1 is exactly v_2.

```
glEvalCoord2( iu+u_1,jv+v_1 );
```

`void glFeedbackBuffer` *size type buffer* [Function]

Controls feedback mode.

size Specifies the maximum number of values that can be written into *buffer*.

type Specifies a symbolic constant that describes the information that will be returned for each vertex. `GL_2D`, `GL_3D`, `GL_3D_COLOR`, `GL_3D_COLOR_TEXTURE`, and `GL_4D_COLOR_TEXTURE` are accepted.

buffer Returns the feedback data.

The `glFeedbackBuffer` function controls feedback. Feedback, like selection, is a GL mode. The mode is selected by calling `glRenderMode` with `GL_FEEDBACK`. When the GL is in feedback mode, no pixels are produced by rasterization. Instead, information about primitives that would have been rasterized is fed back to the application using the GL.

`glFeedbackBuffer` has three arguments: *buffer* is a pointer to an array of floating-point values into which feedback information is placed. *size* indicates the size of the array. *type* is a symbolic constant describing the information that is fed back for each vertex. `glFeedbackBuffer` must be issued before feedback mode is enabled (by calling `glRenderMode` with argument `GL_FEEDBACK`). Setting `GL_FEEDBACK` without establishing the feedback buffer, or calling `glFeedbackBuffer` while the GL is in feedback mode, is an error.

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

While in feedback mode, each primitive, bitmap, or pixel rectangle that would be rasterized generates a block of values that are copied into the feedback array. If doing

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

The `glPassThrough` command can be used to insert a marker into the feedback buffer. See `glPassThrough`.

Following is the grammar for the blocks of values written into the feedback buffer. Each primitive is indicated with a unique identifying value followed by some number of vertices. Polygon entries include an integer value indicating how many vertices follow. A vertex is fed back as some number of floating-point values, as determined by *type*. Colors are fed back as four values in RGBA mode and one value in color index mode.

feedbackList feedbackItem feedbackList | feedbackItem feedbackItem point | lineSegment | polygon | bitmap | pixelRectangle | passThru point `GL_POINT_TOKEN` vertex lineSegment `GL_LINE_TOKEN` vertex vertex | `GL_LINE_RESET_TOKEN` vertex vertex polygon `GL_POLYGON_TOKEN` n polySpec polySpec polySpec vertex | vertex vertex vertex bitmap `GL_BITMAP_TOKEN` vertex pixelRectangle `GL_DRAW_PIXEL_TOKEN` vertex | `GL_COPY_PIXEL_TOKEN` vertex passThru `GL_PASS_THROUGH_TOKEN` value vertex 2d | 3d | 3dColor | 3dColorTexture | 4dColorTexture 2d value value 3d value value value 3dColor value value value color 3dColorTexture value value value color tex 4dColorTexture value value value value color tex color rgba | index rgba value value value value index value tex value value value value

value is a floating-point number, and *n* is a floating-point integer giving the number of vertices in the polygon. `GL_POINT_TOKEN`, `GL_LINE_TOKEN`, `GL_LINE_RESET_TOKEN`, `GL_POLYGON_TOKEN`, `GL_BITMAP_TOKEN`, `GL_DRAW_PIXEL_TOKEN`, `GL_COPY_PIXEL_TOKEN` and `GL_PASS_THROUGH_TOKEN` are symbolic floating-point constants. `GL_LINE_RESET_TOKEN` is returned whenever the line stipple pattern is reset. The data returned as a vertex depends on the feedback *type*.

The following table gives the correspondence between *type* and the number of values per vertex. *k* is 1 in color index mode and 4 in RGBA mode.

Type **Coordinates**, **Color**, **Texture**, **Total Number of Values**

`GL_2D` *x*, *y*, , , 2

`GL_3D` *x*, *y*, *z*, , , 3

`GL_3D_COLOR`
x, *y*, *z*, *k*, , 3+*k*

`GL_3D_COLOR_TEXTURE`
x, *y*, *z*, *k*, 4 , 7+*k*

`GL_4D_COLOR_TEXTURE`
x, *y*, *z*, *w*, *k*, 4 , 8+*k*

Feedback vertex coordinates are in window coordinates, except *w*, which is in clip coordinates. Feedback colors are lighted, if lighting is enabled. Feedback texture coordinates are generated, if texture coordinate generation is enabled. They are always transformed by the texture matrix.

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

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

`GL_INVALID_OPERATION` is generated if `glFeedbackBuffer` is called while the render mode is `GL_FEEDBACK`, or if `glRenderMode` is called with argument `GL_FEEDBACK` before `glFeedbackBuffer` is called at least once.

`GL_INVALID_OPERATION` is generated if `glFeedbackBuffer` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glFinish` [Function]

Block until all GL execution is complete.

`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.

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

`void glFlush` [Function]

Force execution of GL commands in finite time.

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.

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

`void glFogCoordPointer` *type stride pointer* [Function]

Define an array of fog coordinates.

type Specifies the data type of each fog coordinate. Symbolic constants `GL_FLOAT`, or `GL_DOUBLE` are accepted. The initial value is `GL_FLOAT`.

stride Specifies the byte offset between consecutive fog coordinates. If *stride* is 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 fog coordinate in the array. The initial value is 0.

`glFogCoordPointer` specifies the location and data format of an array of fog coordinates to use when rendering. *type* specifies the data type of each fog coordinate, and *stride* specifies the byte stride from one fog coordinate to the next, allowing vertices and attributes to be packed into a single array or stored in separate arrays.

If a non-zero named buffer object is bound to the `GL_ARRAY_BUFFER` target (see `glBindBuffer`) while a fog coordinate array is specified, *pointer* is treated as a byte offset into the buffer object's data store. Also, the buffer object binding (`GL_ARRAY_BUFFER_BINDING`) is saved as fog coordinate vertex array client-side state (`GL_FOG_COORD_ARRAY_BUFFER_BINDING`).

When a fog coordinate array is specified, *type*, *stride*, and *pointer* are saved as client-side state, in addition to the current vertex array buffer object binding.

To enable and disable the fog coordinate array, call `glEnableClientState` and `glDisableClientState` with the argument `GL_FOG_COORD_ARRAY`. If enabled, the fog coordinate array is used when `glDrawArrays`, `glMultiDrawArrays`, `glDrawElements`, `glMultiDrawElements`, `glDrawRangeElements`, or `glArrayElement` is called.

`GL_INVALID_ENUM` is generated if *type* is not either `GL_FLOAT` or `GL_DOUBLE`.

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

`void glFogCoordd` *coord* [Function]
`void glFogCoordf` *coord* [Function]
`void glFogCoorddv` *coord* [Function]
`void glFogCoordfv` *coord* [Function]

Set the current fog coordinates.

coord Specify the fog distance.

`glFogCoord` specifies the fog coordinate that is associated with each vertex and the current raster position. The value specified is interpolated and used in computing the fog color (see `glFog`).

`void glFogf` *pname param* [Function]
`void glFogi` *pname param* [Function]
`void glFogfv` *pname params* [Function]
`void glFogiv` *pname params* [Function]

Specify fog parameters.

pname Specifies a single-valued fog parameter. `GL_FOG_MODE`, `GL_FOG_DENSITY`, `GL_FOG_START`, `GL_FOG_END`, `GL_FOG_INDEX`, and `GL_FOG_COORD_SRC` are accepted.

param Specifies the value that *pname* will be set to.

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

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

`GL_FOG_MODE`

params is a single integer or floating-point value that specifies the equation to be used to compute the fog blend factor, f. Three symbolic constants are accepted: `GL_LINEAR`, `GL_EXP`, and `GL_EXP2`. The equations corresponding to these symbolic constants are defined below. The initial fog mode is `GL_EXP`.

`GL_FOG_DENSITY`

params is a single integer or floating-point value that specifies *density*, the fog density used in both exponential fog equations. Only nonnegative densities are accepted. The initial fog density is 1.

`GL_FOG_START`

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

`GL_FOG_END`

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

`GL_FOG_INDEX`

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

`GL_FOG_COLOR`

params contains four integer or floating-point values that specify C_f, the fog color. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. After conversion, all color components are clamped to the range [0,1]. The initial fog color is (0, 0, 0, 0).

`GL_FOG_COORD_SRC`

params contains either of the following symbolic constants: `GL_FOG_COORD` or `GL_FRAGMENT_DEPTH`. `GL_FOG_COORD` specifies that the current fog coordinate should be used as distance value in the fog color computation. `GL_FRAGMENT_DEPTH` specifies that the current fragment depth should be used as distance value in the fog computation.

Fog blends a fog color with each rasterized pixel fragment's post-texturing color using a blending factor f. Factor f is computed in one of three ways, depending on the fog mode. Let c be either the distance in eye coordinate from the origin (in the case that the `GL_FOG_COORD_SRC` is `GL_FRAGMENT_DEPTH`) or the current fog coordinate (in the case that `GL_FOG_COORD_SRC` is `GL_FOG_COORD`). The equation for `GL_LINEAR` fog is f=*end*-*c*,/*end*-*start*,

The equation for `GL_EXP` fog is f=*e*^-(*density**c*,),

The equation for `GL_EXP2` fog is f=*e*^-(*density**c*,),^2

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

$C\_r,\hat{}=f C\_r+(1-f,)C\_f$

Fog does not affect a fragment's alpha component.

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

$i\_r,\hat{}=i\_r+(1-f,)i\_f$

`GL_INVALID_ENUM` is generated if *pname* is not an accepted value, or if *pname* is `GL_FOG_MODE` and *params* is not an accepted value.

`GL_INVALID_VALUE` is generated if *pname* is `GL_FOG_DENSITY` and *params* is negative.

`GL_INVALID_OPERATION` is generated if `glFog` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glFrontFace` *mode* [Function]

Define front- and back-facing polygons.

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

In a scene composed entirely of opaque closed surfaces, back-facing polygons are never visible. Eliminating these invisible polygons has the obvious benefit of speeding up the rendering of the image. To enable and disable elimination of back-facing polygons, call `glEnable` and `glDisable` with argument `GL_CULL_FACE`.

The projection of a polygon to window coordinates is said to have clockwise winding if an imaginary object following the path from its first vertex, its second vertex, and so on, to its last vertex, and finally back to its first vertex, moves in a clockwise direction about the interior of the polygon. The polygon's winding is said to be counterclockwise if the imaginary object following the same path moves in a counterclockwise direction about the interior of the polygon. `glFrontFace` specifies whether polygons with clockwise winding in window coordinates, or counterclockwise winding in window coordinates, are taken to be front-facing. Passing `GL_CCW` to *mode* selects counterclockwise polygons as front-facing; `GL_CW` selects clockwise polygons as front-facing. By default, counterclockwise polygons are taken to be front-facing.

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

`GL_INVALID_OPERATION` is generated if `glFrontFace` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glFrustum` *left right bottom top nearVal farVal* [Function]

Multiply the current matrix by a perspective matrix.

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.

nearVal
farVal Specify the distances to the near and far depth clipping planes. Both distances must be positive.

`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:

[(2*nearVal*/*right-left*,, 0 *A* 0), (0 2*nearVal*/*top-bottom*,, *B* 0), (0 0 *C* *D*), (0 0 -1 0),]

A=*right*+*left*,/*right-left*,

B=*top*+*bottom*,/*top-bottom*,

C=-*farVal*+*nearVal*,/*farVal-nearVal*,,

D=-2*farValnearVal*,/*farVal-nearVal*,,

Typically, the matrix mode is `GL_PROJECTION`, and (*left*,*bottom-nearVal*) and (*right*,*top-nearVal*) 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). -*farVal* specifies the location of the far clipping plane. Both *nearVal* and *farVal* must be positive.

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

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

`GL_INVALID_OPERATION` is generated if `glFrustum` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGenBuffers` *n buffers* [Function]

Generate buffer object names.

n Specifies the number of buffer object names to be generated.

buffers Specifies an array in which the generated buffer object names are stored.

`glGenBuffers` returns *n* buffer object names in *buffers*. 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 `glGenBuffers`.

Buffer object names returned by a call to `glGenBuffers` are not returned by subsequent calls, unless they are first deleted with `glDeleteBuffers`.

No buffer objects are associated with the returned buffer object names until they are first bound by calling `glBindBuffer`.

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

`GL_INVALID_OPERATION` is generated if `glGenBuffers` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`GLuint glGenLists` *range* [Function]

Generate a contiguous set of empty display lists.

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

`glGenLists` has one argument, *range*. It returns an integer *n* such that *range* contiguous empty display lists, named *n*, *n*+1, ..., *n*+*range*-1, are created. If *range* is 0, if there is no group of *range* contiguous names available, or if any error is generated, no display lists are generated, and 0 is returned.

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

`GL_INVALID_OPERATION` is generated if `glGenLists` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGenQueries` *n ids* [Function]

Generate query object names.

n Specifies the number of query object names to be generated.

ids Specifies an array in which the generated query object names are stored.

`glGenQueries` returns *n* query object names in *ids*. 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 `glGenQueries`.

Query object names returned by a call to `glGenQueries` are not returned by subsequent calls, unless they are first deleted with `glDeleteQueries`.

No query objects are associated with the returned query object names until they are first used by calling `glBeginQuery`.

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

`GL_INVALID_OPERATION` is generated if `glGenQueries` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGenTextures` *n textures* [Function]

Generate texture names.

n Specifies the number of texture names to be generated.

textures Specifies an array in which the generated texture names are stored.

`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`.

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

`GL_INVALID_OPERATION` is generated if `glGenTextures` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGetActiveAttrib` *program index bufSize length size type name* [Function]

Returns information about an active attribute variable for the specified program object.

program Specifies the program object to be queried.

index Specifies the index of the attribute variable to be queried.

bufSize Specifies the maximum number of characters OpenGL is allowed to write in the character buffer indicated by *name*.

length Returns the number of characters actually written by OpenGL in the string indicated by *name* (excluding the null terminator) if a value other than `NULL` is passed.

size Returns the size of the attribute variable.

type Returns the data type of the attribute variable.

name Returns a null terminated string containing the name of the attribute variable.

`glGetActiveAttrib` returns information about an active attribute variable in the program object specified by *program*. The number of active attributes can be obtained by calling `glGetProgram` with the value `GL_ACTIVE_ATTRIBUTES`. A value of 0 for *index* selects the first active attribute variable. Permissible values for *index* range from 0 to the number of active attribute variables minus 1.

A vertex shader may use either built-in attribute variables, user-defined attribute variables, or both. Built-in attribute variables have a prefix of "gl_" and reference conventional OpenGL vertex attribtes (e.g., *gl_Vertex*, *gl_Normal*, etc., see the OpenGL Shading Language specification for a complete list.) User-defined attribute variables have arbitrary names and obtain their values through numbered generic vertex attributes. An attribute variable (either built-in or user-defined) is considered active if it is determined during the link operation that it may be accessed during program execution. Therefore, *program* should have previously been the target of a call to `glLinkProgram`, but it is not necessary for it to have been linked successfully.

The size of the character buffer required to store the longest attribute variable name in *program* can be obtained by calling `glGetProgram` with the value `GL_ACTIVE_ATTRIBUTE_MAX_LENGTH`. This value should be used to allocate a buffer of sufficient size to store the returned attribute name. The size of this character buffer is passed in *bufSize*, and a pointer to this character buffer is passed in *name*.

`glGetActiveAttrib` returns the name of the attribute variable indicated by *index*, storing it in the character buffer specified by *name*. The string returned will be null terminated. The actual number of characters written into this buffer is returned in *length*, and this count does not include the null termination character. If the length of the returned string is not required, a value of `NULL` can be passed in the *length* argument.

The *type* argument will return a pointer to the attribute variable's data type. The symbolic constants `GL_FLOAT`, `GL_FLOAT_VEC2`, `GL_FLOAT_VEC3`, `GL_FLOAT_VEC4`, `GL_FLOAT_MAT2`, `GL_FLOAT_MAT3`, `GL_FLOAT_MAT4`, `GL_FLOAT_MAT2x3`, `GL_FLOAT_MAT2x4`, `GL_FLOAT_MAT3x2`, `GL_FLOAT_MAT3x4`, `GL_FLOAT_MAT4x2`, or `GL_FLOAT_MAT4x3` may be returned. The *size* argument will return the size of the attribute, in units of the type returned in *type*.

The list of active attribute variables may include both built-in attribute variables (which begin with the prefix "gl_") as well as user-defined attribute variable names.

This function will return as much information as it can about the specified active attribute variable. If no information is available, *length* will be 0, and *name* will be an empty string. This situation could occur if this function is called after a link

operation that failed. If an error occurs, the return values *length*, *size*, *type*, and *name* will be unmodified.

`GL_INVALID_VALUE` is generated if *program* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *program* is not a program object.

`GL_INVALID_VALUE` is generated if *index* is greater than or equal to the number of active attribute variables in *program*.

`GL_INVALID_OPERATION` is generated if `glGetActiveAttrib` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`GL_INVALID_VALUE` is generated if *bufSize* is less than 0.

`void glGetActiveUniform` *program index bufSize length size type name* [Function]

Returns information about an active uniform variable for the specified program object.

program Specifies the program object to be queried.

index Specifies the index of the uniform variable to be queried.

bufSize Specifies the maximum number of characters OpenGL is allowed to write in the character buffer indicated by *name*.

length Returns the number of characters actually written by OpenGL in the string indicated by *name* (excluding the null terminator) if a value other than `NULL` is passed.

size Returns the size of the uniform variable.

type Returns the data type of the uniform variable.

name Returns a null terminated string containing the name of the uniform variable.

`glGetActiveUniform` returns information about an active uniform variable in the program object specified by *program*. The number of active uniform variables can be obtained by calling `glGetProgram` with the value `GL_ACTIVE_UNIFORMS`. A value of 0 for *index* selects the first active uniform variable. Permissible values for *index* range from 0 to the number of active uniform variables minus 1.

Shaders may use either built-in uniform variables, user-defined uniform variables, or both. Built-in uniform variables have a prefix of `"gl_"` and reference existing OpenGL state or values derived from such state (e.g., *gl_Fog*, *gl_ModelViewMatrix*, etc., see the OpenGL Shading Language specification for a complete list.) User-defined uniform variables have arbitrary names and obtain their values from the application through calls to `glUniform`. A uniform variable (either built-in or user-defined) is considered active if it is determined during the link operation that it may be accessed during program execution. Therefore, *program* should have previously been the target of a call to `glLinkProgram`, but it is not necessary for it to have been linked successfully.

The size of the character buffer required to store the longest uniform variable name in *program* can be obtained by calling `glGetProgram` with the value `GL_ACTIVE_UNIFORM_MAX_LENGTH`. This value should be used to allocate a buffer of sufficient size to store the returned uniform variable name. The size of this character buffer is passed in *bufSize*, and a pointer to this character buffer is passed in *name*.

`glGetActiveUniform` returns the name of the uniform variable indicated by *index*, storing it in the character buffer specified by *name*. The string returned will be null terminated. The actual number of characters written into this buffer is returned in *length*, and this count does not include the null termination character. If the length of the returned string is not required, a value of `NULL` can be passed in the *length* argument.

The *type* argument will return a pointer to the uniform variable's data type. The symbolic constants `GL_FLOAT`, `GL_FLOAT_VEC2`, `GL_FLOAT_VEC3`, `GL_FLOAT_VEC4`, `GL_INT`, `GL_INT_VEC2`, `GL_INT_VEC3`, `GL_INT_VEC4`, `GL_BOOL`, `GL_BOOL_VEC2`, `GL_BOOL_VEC3`, `GL_BOOL_VEC4`, `GL_FLOAT_MAT2`, `GL_FLOAT_MAT3`, `GL_FLOAT_MAT4`, `GL_FLOAT_MAT2x3`, `GL_FLOAT_MAT2x4`, `GL_FLOAT_MAT3x2`, `GL_FLOAT_MAT3x4`, `GL_FLOAT_MAT4x2`, `GL_FLOAT_MAT4x3`, `GL_SAMPLER_1D`, `GL_SAMPLER_2D`, `GL_SAMPLER_3D`, `GL_SAMPLER_CUBE`, `GL_SAMPLER_1D_SHADOW`, or `GL_SAMPLER_2D_SHADOW` may be returned.

If one or more elements of an array are active, the name of the array is returned in *name*, the type is returned in *type*, and the *size* parameter returns the highest array element index used, plus one, as determined by the compiler and/or linker. Only one active uniform variable will be reported for a uniform array.

Uniform variables that are declared as structures or arrays of structures will not be returned directly by this function. Instead, each of these uniform variables will be reduced to its fundamental components containing the "." and "[]" operators such that each of the names is valid as an argument to `glGetUniformLocation`. Each of these reduced uniform variables is counted as one active uniform variable and is assigned an index. A valid name cannot be a structure, an array of structures, or a subcomponent of a vector or matrix.

The size of the uniform variable will be returned in *size*. Uniform variables other than arrays will have a size of 1. Structures and arrays of structures will be reduced as described earlier, such that each of the names returned will be a data type in the earlier list. If this reduction results in an array, the size returned will be as described for uniform arrays; otherwise, the size returned will be 1.

The list of active uniform variables may include both built-in uniform variables (which begin with the prefix "gl_") as well as user-defined uniform variable names.

This function will return as much information as it can about the specified active uniform variable. If no information is available, *length* will be 0, and *name* will be an empty string. This situation could occur if this function is called after a link operation that failed. If an error occurs, the return values *length*, *size*, *type*, and *name* will be unmodified.

`GL_INVALID_VALUE` is generated if *program* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *program* is not a program object.

`GL_INVALID_VALUE` is generated if *index* is greater than or equal to the number of active uniform variables in *program*.

`GL_INVALID_OPERATION` is generated if `glGetActiveUniform` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`GL_INVALID_VALUE` is generated if *bufSize* is less than 0.

`void glGetAttachedShaders` *program maxCount count shaders* [Function]

Returns the handles of the shader objects attached to a program object.

program Specifies the program object to be queried.

maxCount Specifies the size of the array for storing the returned object names.

count Returns the number of names actually returned in *objects*.

shaders Specifies an array that is used to return the names of attached shader objects.

`glGetAttachedShaders` returns the names of the shader objects attached to *program*. The names of shader objects that are attached to *program* will be returned in *shaders*. The actual number of shader names written into *shaders* is returned in *count*. If no shader objects are attached to *program*, *count* is set to 0. The maximum number of shader names that may be returned in *shaders* is specified by *maxCount*.

If the number of names actually returned is not required (for instance, if it has just been obtained by calling `glGetProgram`), a value of `NULL` may be passed for count. If no shader objects are attached to *program*, a value of 0 will be returned in *count*. The actual number of attached shaders can be obtained by calling `glGetProgram` with the value `GL_ATTACHED_SHADERS`.

`GL_INVALID_VALUE` is generated if *program* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *program* is not a program object.

`GL_INVALID_VALUE` is generated if *maxCount* is less than 0.

`GL_INVALID_OPERATION` is generated if `glGetAttachedShaders` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`GLint glGetAttribLocation` *program name* [Function]

Returns the location of an attribute variable.

program Specifies the program object to be queried.

name Points to a null terminated string containing the name of the attribute variable whose location is to be queried.

`glGetAttribLocation` queries the previously linked program object specified by *program* for the attribute variable specified by *name* and returns the index of the generic vertex attribute that is bound to that attribute variable. If *name* is a matrix attribute variable, the index of the first column of the matrix is returned. If the named attribute variable is not an active attribute in the specified program object or if *name* starts with the reserved prefix "gl_", a value of -1 is returned.

The association between an attribute variable name and a generic attribute index can be specified at any time by calling `glBindAttribLocation`. Attribute bindings do not go into effect until `glLinkProgram` is called. After a program object has been linked successfully, the index values for attribute variables remain fixed until the next link command occurs. The attribute values can only be queried after a link if the link was successful. `glGetAttribLocation` returns the binding that actually went into effect the last time `glLinkProgram` was called for the specified program object. Attribute bindings that have been specified since the last link operation are not returned by `glGetAttribLocation`.

`GL_INVALID_OPERATION` is generated if *program* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *program* is not a program object.

`GL_INVALID_OPERATION` is generated if *program* has not been successfully linked.

`GL_INVALID_OPERATION` is generated if `glGetAttribLocation` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGetBufferParameteriv` *target value data* [Function]

Return parameters of a buffer object.

target Specifies the target buffer object. The symbolic constant must be `GL_ARRAY_BUFFER`, `GL_ELEMENT_ARRAY_BUFFER`, `GL_PIXEL_PACK_BUFFER`, or `GL_PIXEL_UNPACK_BUFFER`.

value Specifies the symbolic name of a buffer object parameter. Accepted values are `GL_BUFFER_ACCESS`, `GL_BUFFER_MAPPED`, `GL_BUFFER_SIZE`, or `GL_BUFFER_USAGE`.

data Returns the requested parameter.

`glGetBufferParameteriv` returns in *data* a selected parameter of the buffer object specified by *target*.

value names a specific buffer object parameter, as follows:

`GL_BUFFER_ACCESS`
: *params* returns the access policy set while mapping the buffer object. The initial value is `GL_READ_WRITE`.

`GL_BUFFER_MAPPED`
: *params* returns a flag indicating whether the buffer object is currently mapped. The initial value is `GL_FALSE`.

`GL_BUFFER_SIZE`
: *params* returns the size of the buffer object, measured in bytes. The initial value is 0.

`GL_BUFFER_USAGE`
: *params* returns the buffer object's usage pattern. The initial value is `GL_STATIC_DRAW`.

`GL_INVALID_ENUM` is generated if *target* or *value* is not an accepted value.

`GL_INVALID_OPERATION` is generated if the reserved buffer object name 0 is bound to *target*.

`GL_INVALID_OPERATION` is generated if `glGetBufferParameteriv` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGetBufferPointerv` *target pname params* [Function]

Return the pointer to a mapped buffer object's data store.

target Specifies the target buffer object. The symbolic constant must be `GL_ARRAY_BUFFER`, `GL_ELEMENT_ARRAY_BUFFER`, `GL_PIXEL_PACK_BUFFER`, or `GL_PIXEL_UNPACK_BUFFER`.

pname Specifies the pointer to be returned. The symbolic constant must be `GL_BUFFER_MAP_POINTER`.

params Returns the pointer value specified by *pname*.

`glGetBufferPointerv` returns pointer information. *pname* is a symbolic constant indicating the pointer to be returned, which must be `GL_BUFFER_MAP_POINTER`, the pointer to which the buffer object's data store is mapped. If the data store is not currently mapped, `NULL` is returned. *params* is a pointer to a location in which to place the returned pointer value.

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

`GL_INVALID_OPERATION` is generated if the reserved buffer object name 0 is bound to *target*.

`GL_INVALID_OPERATION` is generated if `glGetBufferPointerv` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGetBufferSubData` *target offset size data* [Function]

Returns a subset of a buffer object's data store.

target Specifies the target buffer object. The symbolic constant must be `GL_ARRAY_BUFFER`, `GL_ELEMENT_ARRAY_BUFFER`, `GL_PIXEL_PACK_BUFFER`, or `GL_PIXEL_UNPACK_BUFFER`.

offset Specifies the offset into the buffer object's data store from which data will be returned, measured in bytes.

size Specifies the size in bytes of the data store region being returned.

data Specifies a pointer to the location where buffer object data is returned.

`glGetBufferSubData` returns some or all of the data from the buffer object currently bound to *target*. Data starting at byte offset *offset* and extending for *size* bytes is copied from the data store to the memory pointed to by *data*. An error is thrown if the buffer object is currently mapped, or if *offset* and *size* together define a range beyond the bounds of the buffer object's data store.

`GL_INVALID_ENUM` is generated if *target* is not `GL_ARRAY_BUFFER`, `GL_ELEMENT_ARRAY_BUFFER`, `GL_PIXEL_PACK_BUFFER`, or `GL_PIXEL_UNPACK_BUFFER`.

`GL_INVALID_VALUE` is generated if *offset* or *size* is negative, or if together they define a region of memory that extends beyond the buffer object's allocated data store.

`GL_INVALID_OPERATION` is generated if the reserved buffer object name 0 is bound to *target*.

`GL_INVALID_OPERATION` is generated if the buffer object being queried is mapped.

`GL_INVALID_OPERATION` is generated if `glGetBufferSubData` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGetClipPlane` *plane equation* [Function]

Return the coefficients of the specified clipping plane.

plane Specifies a clipping plane. The number of clipping planes depends on the implementation, but at least six clipping planes are supported. They are

identified by symbolic names of the form GL_CLIP_PLANE*i* where i ranges from 0 to the value of GL_MAX_CLIP_PLANES - 1.

equation Returns four double-precision values that are the coefficients of the plane equation of *plane* in eye coordinates. The initial value is (0, 0, 0, 0).

glGetClipPlane returns in *equation* the four coefficients of the plane equation for *plane*.

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

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

`void glGetColorTableParameterfv` *target pname params* [Function]
`void glGetColorTableParameteriv` *target pname params* [Function]

Get color lookup table parameters.

target The target color table. Must be GL_COLOR_TABLE, GL_POST_CONVOLUTION_COLOR_TABLE, GL_POST_COLOR_MATRIX_COLOR_TABLE, GL_PROXY_COLOR_TABLE, GL_PROXY_POST_CONVOLUTION_COLOR_TABLE, or GL_PROXY_POST_COLOR_MATRIX_COLOR_TABLE.

pname The symbolic name of a color lookup table parameter. Must be one of GL_COLOR_TABLE_BIAS, GL_COLOR_TABLE_SCALE, GL_COLOR_TABLE_FORMAT, GL_COLOR_TABLE_WIDTH, GL_COLOR_TABLE_RED_SIZE, GL_COLOR_TABLE_GREEN_SIZE, GL_COLOR_TABLE_BLUE_SIZE, GL_COLOR_TABLE_ALPHA_SIZE, GL_COLOR_TABLE_LUMINANCE_SIZE, or GL_COLOR_TABLE_INTENSITY_SIZE.

params A pointer to an array where the values of the parameter will be stored.

Returns parameters specific to color table *target*.

When *pname* is set to GL_COLOR_TABLE_SCALE or GL_COLOR_TABLE_BIAS, glGetColorTableParameter returns the color table scale or bias parameters for the table specified by *target*. For these queries, *target* must be set to GL_COLOR_TABLE, GL_POST_CONVOLUTION_COLOR_TABLE, or GL_POST_COLOR_MATRIX_COLOR_TABLE and *params* points to an array of four elements, which receive the scale or bias factors for red, green, blue, and alpha, in that order.

glGetColorTableParameter can also be used to retrieve the format and size parameters for a color table. For these queries, set *target* to either the color table target or the proxy color table target. The format and size parameters are set by glColorTable.

The following table lists the format and size parameters that may be queried. For each symbolic constant listed below for *pname*, *params* must point to an array of the given length and receive the values indicated.

Parameter

N, **Meaning**

GL_COLOR_TABLE_FORMAT

1 , Internal format (e.g., GL_RGBA)

GL_COLOR_TABLE_WIDTH

1 , Number of elements in table

`GL_COLOR_TABLE_RED_SIZE`
1 , Size of red component, in bits

`GL_COLOR_TABLE_GREEN_SIZE`
1 , Size of green component

`GL_COLOR_TABLE_BLUE_SIZE`
1 , Size of blue component

`GL_COLOR_TABLE_ALPHA_SIZE`
1 , Size of alpha component

`GL_COLOR_TABLE_LUMINANCE_SIZE`
1 , Size of luminance component

`GL_COLOR_TABLE_INTENSITY_SIZE`
1 , Size of intensity component

`GL_INVALID_ENUM` is generated if *target* or *pname* is not an acceptable value.

`GL_INVALID_OPERATION` is generated if `glGetColorTableParameter` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGetColorTable` *target format type table* [Function]
Retrieve contents of a color lookup table.

target Must be `GL_COLOR_TABLE`, `GL_POST_CONVOLUTION_COLOR_TABLE`, or `GL_POST_COLOR_MATRIX_COLOR_TABLE`.

format The format of the pixel data in *table*. The possible values are `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`, `GL_LUMINANCE`, `GL_LUMINANCE_ALPHA`, `GL_RGB`, `GL_BGR`, `GL_RGBA`, and `GL_BGRA`.

type The type of the pixel data in *table*. Symbolic constants `GL_UNSIGNED_BYTE`, `GL_BYTE`, `GL_BITMAP`, `GL_UNSIGNED_SHORT`, `GL_SHORT`, `GL_UNSIGNED_INT`, `GL_INT`, `GL_FLOAT`, `GL_UNSIGNED_BYTE_3_3_2`, `GL_UNSIGNED_BYTE_2_3_3_REV`, `GL_UNSIGNED_SHORT_5_6_5`, `GL_UNSIGNED_SHORT_5_6_5_REV`, `GL_UNSIGNED_SHORT_4_4_4_4`, `GL_UNSIGNED_SHORT_4_4_4_4_REV`, `GL_UNSIGNED_SHORT_5_5_5_1`, `GL_UNSIGNED_SHORT_1_5_5_5_REV`, `GL_UNSIGNED_INT_8_8_8_8`, `GL_UNSIGNED_INT_8_8_8_8_REV`, `GL_UNSIGNED_INT_10_10_10_2`, and `GL_UNSIGNED_INT_2_10_10_10_REV` are accepted.

table Pointer to a one-dimensional array of pixel data containing the contents of the color table.

`glGetColorTable` returns in *table* the contents of the color table specified by *target*. No pixel transfer operations are performed, but pixel storage modes that are applicable to `glReadPixels` are performed.

If a non-zero named buffer object is bound to the `GL_PIXEL_PACK_BUFFER` target (see `glBindBuffer`) while a histogram table is requested, *table* is treated as a byte offset into the buffer object's data store.

Color components that are requested in the specified *format*, but which are not included in the internal format of the color lookup table, are returned as zero. The assignments of internal color components to the components requested by *format* are

Internal Component	Resulting Component
Red	Red
Green	Green
Blue	Blue
Alpha	Alpha
Luminance	Red
Intensity	Red

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

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

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

`GL_INVALID_OPERATION` is generated if *type* is one of `GL_UNSIGNED_BYTE_3_3_2`, `GL_UNSIGNED_BYTE_2_3_3_REV`, `GL_UNSIGNED_SHORT_5_6_5`, or `GL_UNSIGNED_SHORT_5_6_5_REV` and *format* is not `GL_RGB`.

`GL_INVALID_OPERATION` is generated if *type* is one of `GL_UNSIGNED_SHORT_4_4_4_4`, `GL_UNSIGNED_SHORT_4_4_4_4_REV`, `GL_UNSIGNED_SHORT_5_5_5_1`, `GL_UNSIGNED_SHORT_1_5_5_5_REV`, `GL_UNSIGNED_INT_8_8_8_8`, `GL_UNSIGNED_INT_8_8_8_8_REV`, `GL_UNSIGNED_INT_10_10_10_2`, or `GL_UNSIGNED_INT_2_10_10_10_REV` and *format* is neither `GL_RGBA` nor `GL_BGRA`.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_PACK_BUFFER` target and the buffer object's data store is currently mapped.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_PACK_BUFFER` target and the data would be packed to the buffer object such that the memory writes required would exceed the data store size.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_PACK_BUFFER` target and *table* is not evenly divisible into the number of bytes needed to store in memory a datum indicated by *type*.

`GL_INVALID_OPERATION` is generated if `glGetColorTable` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGetCompressedTexImage` *target lod img* [Function]

Return a compressed texture image.

target Specifies which texture is to be obtained. `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, and `GL_TEXTURE_3DGL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Z`, and `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z` are accepted.

lod Specifies the level-of-detail number of the desired image. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.

img Returns the compressed texture image.

`glGetCompressedTexImage` returns the compressed texture image associated with *target* and *lod* into *img*. *img* should be an array of `GL_TEXTURE_COMPRESSED_IMAGE_SIZE` bytes. *target* specifies whether the desired texture image was one specified by `glTexImage1D` (`GL_TEXTURE_1D`), `glTexImage2D` (`GL_TEXTURE_2D` or any of `GL_TEXTURE_CUBE_MAP_*`), or `glTexImage3D` (`GL_TEXTURE_3D`). *lod* specifies the level-of-detail number of the desired image.

If a non-zero named buffer object is bound to the `GL_PIXEL_PACK_BUFFER` target (see `glBindBuffer`) while a texture image is requested, *img* is treated as a byte offset into the buffer object's data store.

To minimize errors, first verify that the texture is compressed by calling `glGetTexLevelParameter` with argument `GL_TEXTURE_COMPRESSED`. If the texture is compressed, then determine the amount of memory required to store the compressed texture by calling `glGetTexLevelParameter` with argument `GL_TEXTURE_COMPRESSED_IMAGE_SIZE`. Finally, retrieve the internal format of the texture by calling `glGetTexLevelParameter` with argument `GL_TEXTURE_INTERNAL_FORMAT`. To store the texture for later use, associate the internal format and size with the retrieved texture image. These data can be used by the respective texture or subtexture loading routine used for loading *target* textures.

`GL_INVALID_VALUE` is generated if *lod* is less than zero or greater than the maximum number of LODs permitted by the implementation.

`GL_INVALID_OPERATION` is generated if `glGetCompressedTexImage` is used to retrieve a texture that is in an uncompressed internal format.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_PACK_BUFFER` target and the buffer object's data store is currently mapped.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_PACK_BUFFER` target and the data would be packed to the buffer object such that the memory writes required would exceed the data store size.

`GL_INVALID_OPERATION` is generated if `glGetCompressedTexImage` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGetConvolutionFilter` *target format type image* [Function]

Get current 1D or 2D convolution filter kernel.

target The filter to be retrieved. Must be one of `GL_CONVOLUTION_1D` or `GL_CONVOLUTION_2D`.

format Format of the output image. Must be one of `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`, `GL_RGB`, `GL_BGR`, `GL_RGBA`, `GL_BGRA`, `GL_LUMINANCE`, or `GL_LUMINANCE_ALPHA`.

type Data type of components in the output image. Symbolic constants `GL_UNSIGNED_BYTE`, `GL_BYTE`, `GL_BITMAP`, `GL_UNSIGNED_SHORT`, `GL_SHORT`, `GL_UNSIGNED_INT`, `GL_INT`, `GL_FLOAT`, `GL_UNSIGNED_BYTE_3_3_2`, `GL_UNSIGNED_BYTE_2_3_3_REV`, `GL_UNSIGNED_SHORT_5_6_5`, `GL_UNSIGNED_SHORT_5_6_5_REV`, `GL_UNSIGNED_SHORT_4_4_4_4`, `GL_UNSIGNED_SHORT_4_4_4_4_REV`, `GL_UNSIGNED_SHORT_5_5_5_1`,

GL_UNSIGNED_SHORT_1_5_5_5_REV, GL_UNSIGNED_INT_8_8_8_8, GL_UNSIGNED_INT_8_8_8_8_REV, GL_UNSIGNED_INT_10_10_10_2, and GL_UNSIGNED_INT_2_10_10_10_REV are accepted.

image Pointer to storage for the output image.

glGetConvolutionFilter returns the current 1D or 2D convolution filter kernel as an image. The one- or two-dimensional image is placed in *image* according to the specifications in *format* and *type*. No pixel transfer operations are performed on this image, but the relevant pixel storage modes are applied.

If a non-zero named buffer object is bound to the GL_PIXEL_PACK_BUFFER target (see glBindBuffer) while a convolution filter is requested, *image* is treated as a byte offset into the buffer object's data store.

Color components that are present in *format* but not included in the internal format of the filter are returned as zero. The assignments of internal color components to the components of *format* are as follows.

Internal Component	**Resulting Component**
Red	Red
Green	Green
Blue	Blue
Alpha	Alpha
Luminance	Red
Intensity	Red

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

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

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

GL_INVALID_OPERATION is generated if *type* is one of GL_UNSIGNED_BYTE_3_3_2, GL_UNSIGNED_BYTE_2_3_3_REV, GL_UNSIGNED_SHORT_5_6_5, or GL_UNSIGNED_SHORT_5_6_5_REV and *format* is not GL_RGB.

GL_INVALID_OPERATION is generated if *type* is one of GL_UNSIGNED_SHORT_4_4_4_4, GL_UNSIGNED_SHORT_4_4_4_4_REV, GL_UNSIGNED_SHORT_5_5_5_1, GL_UNSIGNED_SHORT_1_5_5_5_REV, GL_UNSIGNED_INT_8_8_8_8, GL_UNSIGNED_INT_8_8_8_8_REV, GL_UNSIGNED_INT_10_10_10_2, or GL_UNSIGNED_INT_2_10_10_10_REV and *format* is neither GL_RGBA nor GL_BGRA.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_PACK_BUFFER target and the buffer object's data store is currently mapped.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_PACK_BUFFER target and the data would be packed to the buffer object such that the memory writes required would exceed the data store size.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_PACK_BUFFER` target and *image* is not evenly divisible into the number of bytes needed to store in memory a datum indicated by *type*.

`GL_INVALID_OPERATION` is generated if `glGetConvolutionFilter` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGetConvolutionParameterfv` *target pname params* [Function]
`void glGetConvolutionParameteriv` *target pname params* [Function]
Get convolution parameters.

target The filter whose parameters are to be retrieved. Must be one of `GL_CONVOLUTION_1D`, `GL_CONVOLUTION_2D`, or `GL_SEPARABLE_2D`.

pname The parameter to be retrieved. Must be one of `GL_CONVOLUTION_BORDER_MODE`, `GL_CONVOLUTION_BORDER_COLOR`, `GL_CONVOLUTION_FILTER_SCALE`, `GL_CONVOLUTION_FILTER_BIAS`, `GL_CONVOLUTION_FORMAT`, `GL_CONVOLUTION_WIDTH`, `GL_CONVOLUTION_HEIGHT`, `GL_MAX_CONVOLUTION_WIDTH`, or `GL_MAX_CONVOLUTION_HEIGHT`.

params Pointer to storage for the parameters to be retrieved.

`glGetConvolutionParameter` retrieves convolution parameters. *target* determines which convolution filter is queried. *pname* determines which parameter is returned:

`GL_CONVOLUTION_BORDER_MODE`
The convolution border mode. See `glConvolutionParameter` for a list of border modes.

`GL_CONVOLUTION_BORDER_COLOR`
The current convolution border color. *params* must be a pointer to an array of four elements, which will receive the red, green, blue, and alpha border colors.

`GL_CONVOLUTION_FILTER_SCALE`
The current filter scale factors. *params* must be a pointer to an array of four elements, which will receive the red, green, blue, and alpha filter scale factors in that order.

`GL_CONVOLUTION_FILTER_BIAS`
The current filter bias factors. *params* must be a pointer to an array of four elements, which will receive the red, green, blue, and alpha filter bias terms in that order.

`GL_CONVOLUTION_FORMAT`
The current internal format. See `glConvolutionFilter1D`, `glConvolutionFilter2D`, and `glSeparableFilter2D` for lists of allowable formats.

`GL_CONVOLUTION_WIDTH`
The current filter image width.

`GL_CONVOLUTION_HEIGHT`
The current filter image height.

`GL_MAX_CONVOLUTION_WIDTH`
: The maximum acceptable filter image width.

`GL_MAX_CONVOLUTION_HEIGHT`
: The maximum acceptable filter image height.

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

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

`GL_INVALID_ENUM` is generated if *target* is `GL_CONVOLUTION_1D` and *pname* is `GL_CONVOLUTION_HEIGHT` or `GL_MAX_CONVOLUTION_HEIGHT`.

`GL_INVALID_OPERATION` is generated if `glGetConvolutionParameter` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`GLenum glGetError` [Function]

Return error information.

`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.

`GL_TABLE_TOO_LARGE`
: The specified table exceeds the implementation's maximum supported table size. The offending command is ignored and has no other side effect than to set the error flag.

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.

`GL_INVALID_OPERATION` is generated if `glGetError` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`. In this case, `glGetError` returns 0.

`void glGetHistogramParameterfv` *target pname params* [Function]
`void glGetHistogramParameteriv` *target pname params* [Function]

Get histogram parameters.

target Must be one of `GL_HISTOGRAM` or `GL_PROXY_HISTOGRAM`.

pname The name of the parameter to be retrieved. Must be one of `GL_HISTOGRAM_WIDTH`, `GL_HISTOGRAM_FORMAT`, `GL_HISTOGRAM_RED_SIZE`, `GL_HISTOGRAM_GREEN_SIZE`, `GL_HISTOGRAM_BLUE_SIZE`, `GL_HISTOGRAM_ALPHA_SIZE`, `GL_HISTOGRAM_LUMINANCE_SIZE`, or `GL_HISTOGRAM_SINK`.

params Pointer to storage for the returned values.

`glGetHistogramParameter` is used to query parameter values for the current histogram or for a proxy. The histogram state information may be queried by calling `glGetHistogramParameter` with a *target* of `GL_HISTOGRAM` (to obtain information for the current histogram table) or `GL_PROXY_HISTOGRAM` (to obtain information from the most recent proxy request) and one of the following values for the *pname* argument:

Parameter
: **Description**

`GL_HISTOGRAM_WIDTH`
: Histogram table width

`GL_HISTOGRAM_FORMAT`
: Internal format

`GL_HISTOGRAM_RED_SIZE`
: Red component counter size, in bits

`GL_HISTOGRAM_GREEN_SIZE`
: Green component counter size, in bits

`GL_HISTOGRAM_BLUE_SIZE`
: Blue component counter size, in bits

`GL_HISTOGRAM_ALPHA_SIZE`
: Alpha component counter size, in bits

`GL_HISTOGRAM_LUMINANCE_SIZE`
: Luminance component counter size, in bits

`GL_HISTOGRAM_SINK`
: Value of the *sink* parameter

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

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

`GL_INVALID_OPERATION` is generated if `glGetHistogramParameter` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGetHistogram` *target reset format type values* [Function]

Get histogram table.

target
: Must be `GL_HISTOGRAM`.

reset
: If `GL_TRUE`, each component counter that is actually returned is reset to zero. (Other counters are unaffected.) If `GL_FALSE`, none of the counters in the histogram table is modified.

format
: The format of values to be returned in *values*. Must be one of `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`, `GL_RGB`, `GL_BGR`, `GL_RGBA`, `GL_BGRA`, `GL_LUMINANCE`, or `GL_LUMINANCE_ALPHA`.

type
: The type of values to be returned in *values*. Symbolic constants `GL_UNSIGNED_BYTE`, `GL_BYTE`, `GL_BITMAP`, `GL_UNSIGNED_SHORT`, `GL_SHORT`, `GL_UNSIGNED_INT`, `GL_INT`, `GL_FLOAT`, `GL_UNSIGNED_BYTE_3_3_2`, `GL_UNSIGNED_BYTE_2_3_3_REV`, `GL_UNSIGNED_SHORT_5_6_5`, `GL_UNSIGNED_SHORT_5_6_5_REV`, `GL_UNSIGNED_SHORT_4_4_4_4`, `GL_UNSIGNED_SHORT_4_4_4_4_REV`, `GL_UNSIGNED_SHORT_5_5_5_1`, `GL_UNSIGNED_SHORT_1_5_5_5_REV`, `GL_UNSIGNED_INT_8_8_8_8`, `GL_UNSIGNED_INT_8_8_8_8_REV`, `GL_UNSIGNED_INT_10_10_10_2`, and `GL_UNSIGNED_INT_2_10_10_10_REV` are accepted.

values
: A pointer to storage for the returned histogram table.

`glGetHistogram` returns the current histogram table as a one-dimensional image with the same width as the histogram. No pixel transfer operations are performed on this image, but pixel storage modes that are applicable to 1D images are honored.

If a non-zero named buffer object is bound to the `GL_PIXEL_PACK_BUFFER` target (see `glBindBuffer`) while a histogram table is requested, *values* is treated as a byte offset into the buffer object's data store.

Color components that are requested in the specified *format*, but which are not included in the internal format of the histogram, are returned as zero. The assignments of internal color components to the components requested by *format* are:

Internal Component	Resulting Component
Red	Red
Green	Green
Blue	Blue
Alpha	Alpha
Luminance	Red

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

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

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

`GL_INVALID_OPERATION` is generated if *type* is one of `GL_UNSIGNED_BYTE_3_3_2`, `GL_UNSIGNED_BYTE_2_3_3_REV`, `GL_UNSIGNED_SHORT_5_6_5`, or `GL_UNSIGNED_SHORT_5_6_5_REV` and *format* is not `GL_RGB`.

`GL_INVALID_OPERATION` is generated if *type* is one of `GL_UNSIGNED_SHORT_4_4_4_4`, `GL_UNSIGNED_SHORT_4_4_4_4_REV`, `GL_UNSIGNED_SHORT_5_5_5_1`, `GL_UNSIGNED_SHORT_1_5_5_5_REV`, `GL_UNSIGNED_INT_8_8_8_8`, `GL_UNSIGNED_INT_8_8_8_8_REV`, `GL_UNSIGNED_INT_10_10_10_2`, or `GL_UNSIGNED_INT_2_10_10_10_REV` and *format* is neither `GL_RGBA` nor `GL_BGRA`.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_PACK_BUFFER` target and the buffer object's data store is currently mapped.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_PACK_BUFFER` target and the data would be packed to the buffer object such that the memory writes required would exceed the data store size.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_PACK_BUFFER` target and *values* is not evenly divisible into the number of bytes needed to store in memory a datum indicated by *type*.

`GL_INVALID_OPERATION` is generated if `glGetHistogram` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGetLightfv` *light pname params* [Function]
`void glGetLightiv` *light pname params* [Function]

Return light source parameter values.

light Specifies a light source. The number of possible lights depends on the implementation, but at least eight lights are supported. They are identified by symbolic names of the form `GL_LIGHT`*i* where *i* ranges from 0 to the value of `GL_MAX_LIGHTS` - 1.

pname Specifies a light source parameter for *light*. Accepted symbolic names are `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`, `GL_POSITION`, `GL_SPOT_DIRECTION`, `GL_SPOT_EXPONENT`, `GL_SPOT_CUTOFF`, `GL_CONSTANT_ATTENUATION`, `GL_LINEAR_ATTENUATION`, and `GL_QUADRATIC_ATTENUATION`.

params Returns the requested data.

`glGetLight` returns in *params* the value or values of a light source parameter. *light* names the light and is a symbolic name of the form `GL_LIGHT`*i* where i ranges from 0 to the value of `GL_MAX_LIGHTS` - 1. `GL_MAX_LIGHTS` is an implementation dependent constant that is greater than or equal to eight. *pname* specifies one of ten light source parameters, again by symbolic name.

The following parameters are defined:

`GL_AMBIENT`
: *params* returns four integer or floating-point values representing the ambient intensity of the light source. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined. The initial value is (0, 0, 0, 1).

`GL_DIFFUSE`
: *params* returns four integer or floating-point values representing the diffuse intensity of the light source. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined. The initial value for `GL_LIGHT0` is (1, 1, 1, 1); for other lights, the initial value is (0, 0, 0, 0).

`GL_SPECULAR`
: *params* returns four integer or floating-point values representing the specular intensity of the light source. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined. The initial value for `GL_LIGHT0` is (1, 1, 1, 1); for other lights, the initial value is (0, 0, 0, 0).

`GL_POSITION`
: *params* returns four integer or floating-point values representing the position of the light source. Integer values, when requested, are computed by rounding the internal floating-point values to the nearest integer value. The returned values are those maintained in eye coordinates. They will not be equal to the values specified using `glLight`, unless the modelview matrix was identity at the time `glLight` was called. The initial value is (0, 0, 1, 0).

`GL_SPOT_DIRECTION`
: *params* returns three integer or floating-point values representing the direction of the light source. Integer values, when requested, are computed

by rounding the internal floating-point values to the nearest integer value. The returned values are those maintained in eye coordinates. They will not be equal to the values specified using `glLight`, unless the modelview matrix was identity at the time `glLight` was called. Although spot direction is normalized before being used in the lighting equation, the returned values are the transformed versions of the specified values prior to normalization. The initial value is (0,0-1).

`GL_SPOT_EXPONENT`
: *params* returns a single integer or floating-point value representing the spot exponent of the light. An integer value, when requested, is computed by rounding the internal floating-point representation to the nearest integer. The initial value is 0.

`GL_SPOT_CUTOFF`
: *params* returns a single integer or floating-point value representing the spot cutoff angle of the light. An integer value, when requested, is computed by rounding the internal floating-point representation to the nearest integer. The initial value is 180.

`GL_CONSTANT_ATTENUATION`
: *params* returns a single integer or floating-point value representing the constant (not distance-related) attenuation of the light. An integer value, when requested, is computed by rounding the internal floating-point representation to the nearest integer. The initial value is 1.

`GL_LINEAR_ATTENUATION`
: *params* returns a single integer or floating-point value representing the linear attenuation of the light. An integer value, when requested, is computed by rounding the internal floating-point representation to the nearest integer. The initial value is 0.

`GL_QUADRATIC_ATTENUATION`
: *params* returns a single integer or floating-point value representing the quadratic attenuation of the light. An integer value, when requested, is computed by rounding the internal floating-point representation to the nearest integer. The initial value is 0.

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

`GL_INVALID_OPERATION` is generated if `glGetLight` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGetMapdv` *target query v* [Function]
`void glGetMapfv` *target query v* [Function]
`void glGetMapiv` *target query v* [Function]

Return evaluator parameters.

target Specifies the symbolic name of a map. Accepted values are `GL_MAP1_COLOR_4`, `GL_MAP1_INDEX`, `GL_MAP1_NORMAL`, `GL_MAP1_TEXTURE_COORD_1`, `GL_MAP1_TEXTURE_COORD_2`, `GL_MAP1_TEXTURE_COORD_3`,

GL_MAP1_TEXTURE_COORD_4, GL_MAP1_VERTEX_3, GL_MAP1_VERTEX_4, GL_MAP2_COLOR_4, GL_MAP2_INDEX, GL_MAP2_NORMAL, GL_MAP2_TEXTURE_COORD_1, GL_MAP2_TEXTURE_COORD_2, GL_MAP2_TEXTURE_COORD_3, GL_MAP2_TEXTURE_COORD_4, GL_MAP2_VERTEX_3, and GL_MAP2_VERTEX_4.

query Specifies which parameter to return. Symbolic names GL_COEFF, GL_ORDER, and GL_DOMAIN are accepted.

v Returns the requested data.

glMap1 and glMap2 define evaluators. glGetMap returns evaluator parameters. *target* chooses a map, *query* selects a specific parameter, and *v* points to storage where the values will be returned.

The acceptable values for the *target* parameter are described in the glMap1 and glMap2 reference pages.

query can assume the following values:

GL_COEFF *v* returns the control points for the evaluator function. One-dimensional evaluators return *order* control points, and two-dimensional evaluators return *uorder*×*vorder* control points. Each control point consists of one, two, three, or four integer, single-precision floating-point, or double-precision floating-point values, depending on the type of the evaluator. The GL returns two-dimensional control points in row-major order, incrementing the *uorder* index quickly and the *vorder* index after each row. Integer values, when requested, are computed by rounding the internal floating-point values to the nearest integer values.

GL_ORDER *v* returns the order of the evaluator function. One-dimensional evaluators return a single value, *order*. The initial value is 1. Two-dimensional evaluators return two values, *uorder* and *vorder*. The initial value is 1,1.

GL_DOMAIN

v returns the linear *u* and *v* mapping parameters. One-dimensional evaluators return two values, *u1* and *u2*, as specified by glMap1. Two-dimensional evaluators return four values (*u1*, *u2*, *v1*, and *v2*) as specified by glMap2. Integer values, when requested, are computed by rounding the internal floating-point values to the nearest integer values.

GL_INVALID_ENUM is generated if either *target* or *query* is not an accepted value.

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

`void glGetMaterialfv` *face pname params* [Function]
`void glGetMaterialiv` *face pname params* [Function]

Return material parameters.

face Specifies which of the two materials is being queried. GL_FRONT or GL_BACK are accepted, representing the front and back materials, respectively.

pname Specifies the material parameter to return. GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_EMISSION, GL_SHININESS, and GL_COLOR_INDEXES are accepted.

params Returns the requested data.

`glGetMaterial` returns in *params* the value or values of parameter *pname* of material *face*. Six parameters are defined:

`GL_AMBIENT`
: *params* returns four integer or floating-point values representing the ambient reflectance of the material. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined. The initial value is (0.2, 0.2, 0.2, 1.0)

`GL_DIFFUSE`
: *params* returns four integer or floating-point values representing the diffuse reflectance of the material. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined. The initial value is (0.8, 0.8, 0.8, 1.0).

`GL_SPECULAR`
: *params* returns four integer or floating-point values representing the specular reflectance of the material. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined. The initial value is (0, 0, 0, 1).

`GL_EMISSION`
: *params* returns four integer or floating-point values representing the emitted light intensity of the material. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined. The initial value is (0, 0, 0, 1).

`GL_SHININESS`
: *params* returns one integer or floating-point value representing the specular exponent of the material. Integer values, when requested, are computed by rounding the internal floating-point value to the nearest integer value. The initial value is 0.

`GL_COLOR_INDEXES`
: *params* returns three integer or floating-point values representing the ambient, diffuse, and specular indices of the material. These indices are used only for color index lighting. (All the other parameters are used only for

RGBA lighting.) Integer values, when requested, are computed by rounding the internal floating-point values to the nearest integer values.

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

`GL_INVALID_OPERATION` is generated if `glGetMaterial` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGetMinmaxParameterfv` *target pname params* [Function]
`void glGetMinmaxParameteriv` *target pname params* [Function]
Get minmax parameters.

target Must be `GL_MINMAX`.

pname The parameter to be retrieved. Must be one of `GL_MINMAX_FORMAT` or `GL_MINMAX_SINK`.

params A pointer to storage for the retrieved parameters.

`glGetMinmaxParameter` retrieves parameters for the current minmax table by setting *pname* to one of the following values:

Parameter
Description

`GL_MINMAX_FORMAT`
Internal format of minmax table

`GL_MINMAX_SINK`
Value of the *sink* parameter

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

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

`GL_INVALID_OPERATION` is generated if `glGetMinmaxParameter` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGetMinmax` *target reset format types values* [Function]
Get minimum and maximum pixel values.

target Must be `GL_MINMAX`.

reset If `GL_TRUE`, all entries in the minmax table that are actually returned are reset to their initial values. (Other entries are unaltered.) If `GL_FALSE`, the minmax table is unaltered.

format The format of the data to be returned in *values*. Must be one of `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`, `GL_RGB`, `GL_BGR`, `GL_RGBA`, `GL_BGRA`, `GL_LUMINANCE`, or `GL_LUMINANCE_ALPHA`.

types The type of the data to be returned in *values*. Symbolic constants `GL_UNSIGNED_BYTE`, `GL_BYTE`, `GL_BITMAP`, `GL_UNSIGNED_SHORT`, `GL_SHORT`, `GL_UNSIGNED_INT`, `GL_INT`, `GL_FLOAT`, `GL_UNSIGNED_BYTE_3_3_2`, `GL_UNSIGNED_BYTE_2_3_3_REV`, `GL_UNSIGNED_SHORT_5_6_5`, `GL_UNSIGNED_SHORT_5_6_5_REV`, `GL_UNSIGNED_SHORT_4_4_4_4`, `GL_UNSIGNED_SHORT_4_4_4_4_REV`, `GL_UNSIGNED_SHORT_5_5_5_1`,

GL_UNSIGNED_SHORT_1_5_5_5_REV, GL_UNSIGNED_INT_8_8_8_8, GL_UNSIGNED_INT_8_8_8_8_REV, GL_UNSIGNED_INT_10_10_10_2, and GL_UNSIGNED_INT_2_10_10_10_REV are accepted.

values A pointer to storage for the returned values.

glGetMinmax returns the accumulated minimum and maximum pixel values (computed on a per-component basis) in a one-dimensional image of width 2. The first set of return values are the minima, and the second set of return values are the maxima. The format of the return values is determined by *format*, and their type is determined by *types*.

If a non-zero named buffer object is bound to the GL_PIXEL_PACK_BUFFER target (see glBindBuffer) while minimum and maximum pixel values are requested, *values* is treated as a byte offset into the buffer object's data store.

No pixel transfer operations are performed on the return values, but pixel storage modes that are applicable to one-dimensional images are performed. Color components that are requested in the specified *format*, but that are not included in the internal format of the minmax table, are returned as zero. The assignment of internal color components to the components requested by *format* are as follows:

Internal Component **Resulting Component**

Red Red

Green Green

Blue Blue

Alpha Alpha

Luminance Red

If *reset* is GL_TRUE, the minmax table entries corresponding to the return values are reset to their initial values. Minimum and maximum values that are not returned are not modified, even if *reset* is GL_TRUE.

GL_INVALID_ENUM is generated if *target* is not GL_MINMAX.

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

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

GL_INVALID_OPERATION is generated if *types* is one of GL_UNSIGNED_BYTE_3_3_2, GL_UNSIGNED_BYTE_2_3_3_REV, GL_UNSIGNED_SHORT_5_6_5, or GL_UNSIGNED_SHORT_5_6_5_REV and *format* is not GL_RGB.

GL_INVALID_OPERATION is generated if *types* is one of GL_UNSIGNED_SHORT_4_4_4_4, GL_UNSIGNED_SHORT_4_4_4_4_REV, GL_UNSIGNED_SHORT_5_5_5_1, GL_UNSIGNED_SHORT_1_5_5_5_REV, GL_UNSIGNED_INT_8_8_8_8, GL_UNSIGNED_INT_8_8_8_8_REV, GL_UNSIGNED_INT_10_10_10_2, or GL_UNSIGNED_INT_2_10_10_10_REV and *format* is neither GL_RGBA nor GL_BGRA.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_PACK_BUFFER target and the buffer object's data store is currently mapped.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_PACK_BUFFER target and the data would be packed to the buffer object such that the memory writes required would exceed the data store size.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_PACK_BUFFER target and *values* is not evenly divisible into the number of bytes needed to store in memory a datum indicated by *type*.

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

`void glGetPixelMapfv` *map data* [Function]
`void glGetPixelMapuiv` *map data* [Function]
`void glGetPixelMapusv` *map data* [Function]

Return the specified pixel map.

map Specifies the name of the pixel map to return. Accepted values are GL_PIXEL_MAP_I_TO_I, GL_PIXEL_MAP_S_TO_S, GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, GL_PIXEL_MAP_I_TO_A, GL_PIXEL_MAP_R_TO_R, GL_PIXEL_MAP_G_TO_G, GL_PIXEL_MAP_B_TO_B, and GL_PIXEL_MAP_A_TO_A.

data Returns the pixel map contents.

See the glPixelMap reference page for a description of the acceptable values for the *map* parameter. glGetPixelMap returns in *data* the contents of the pixel map specified in *map*. Pixel maps are used during the execution of glReadPixels, glDrawPixels, glCopyPixels, glTexImage1D, glTexImage2D, glTexImage3D, glTexSubImage1D, glTexSubImage2D, glTexSubImage3D, glCopyTexImage1D, glCopyTexImage2D, glCopyTexSubImage1D, glCopyTexSubImage2D, and glCopyTexSubImage3D. to map color indices, stencil indices, color components, and depth components to other values.

If a non-zero named buffer object is bound to the GL_PIXEL_PACK_BUFFER target (see glBindBuffer) while a pixel map is requested, *data* is treated as a byte offset into the buffer object's data store.

Unsigned integer values, if requested, are linearly mapped from the internal fixed or floating-point representation such that 1.0 maps to the largest representable integer value, and 0.0 maps to 0. Return unsigned integer values are undefined if the map value was not in the range [0,1].

To determine the required size of *map*, call glGet with the appropriate symbolic constant.

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

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_PACK_BUFFER target and the buffer object's data store is currently mapped.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_PACK_BUFFER target and the data would be packed to the buffer object such that the memory writes required would exceed the data store size.

GL_INVALID_OPERATION is generated by glGetPixelMapfv if a non-zero buffer object name is bound to the GL_PIXEL_PACK_BUFFER target and *data* is not evenly divisible into the number of bytes needed to store in memory a GLfloat datum.

GL_INVALID_OPERATION is generated by glGetPixelMapuiv if a non-zero buffer object name is bound to the GL_PIXEL_PACK_BUFFER target and *data* is not evenly divisible into the number of bytes needed to store in memory a GLuint datum.

GL_INVALID_OPERATION is generated by glGetPixelMapusv if a non-zero buffer object name is bound to the GL_PIXEL_PACK_BUFFER target and *data* is not evenly divisible into the number of bytes needed to store in memory a GLushort datum.

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

`void glGetPointerv` *pname params* [Function]

Return the address of the specified pointer.

pname Specifies the array or buffer pointer to be returned. Symbolic constants GL_COLOR_ARRAY_POINTER, GL_EDGE_FLAG_ARRAY_POINTER, GL_FOG_COORD_ARRAY_POINTER, GL_FEEDBACK_BUFFER_POINTER, GL_INDEX_ARRAY_POINTER, GL_NORMAL_ARRAY_POINTER, GL_SECONDARY_COLOR_ARRAY_POINTER, GL_SELECTION_BUFFER_POINTER, GL_TEXTURE_COORD_ARRAY_POINTER, or GL_VERTEX_ARRAY_POINTER are accepted.

params Returns the pointer value specified by *pname*.

glGetPointerv returns pointer information. *pname* is a symbolic constant indicating the pointer to be returned, and *params* is a pointer to a location in which to place the returned data.

For all *pname* arguments except GL_FEEDBACK_BUFFER_POINTER and GL_SELECTION_BUFFER_POINTER, if a non-zero named buffer object was bound to the GL_ARRAY_BUFFER target (see glBindBuffer) when the desired pointer was previously specified, the pointer returned is a byte offset into the buffer object's data store. Buffer objects are only available in OpenGL versions 1.5 and greater.

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

`void glGetPolygonStipple` *pattern* [Function]

Return the polygon stipple pattern.

pattern Returns the stipple pattern. The initial value is all 1's.

glGetPolygonStipple returns to *pattern* a 3232 polygon stipple pattern. The pattern is packed into memory as if glReadPixels with both *height* and *width* of 32, *type* of GL_BITMAP, and *format* of GL_COLOR_INDEX were called, and the stipple pattern were stored in an internal 3232 color index buffer. Unlike glReadPixels, however, pixel transfer operations (shift, offset, pixel map) are not applied to the returned stipple image.

If a non-zero named buffer object is bound to the GL_PIXEL_PACK_BUFFER target (see glBindBuffer) while a polygon stipple pattern is requested, *pattern* is treated as a byte offset into the buffer object's data store.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_PACK_BUFFER target and the buffer object's data store is currently mapped.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_PACK_BUFFER` target and the data would be packed to the buffer object such that the memory writes required would exceed the data store size.

`GL_INVALID_OPERATION` is generated if `glGetPolygonStipple` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGetProgramInfoLog` *program maxLength length infoLog* [Function]
Returns the information log for a program object.

program Specifies the program object whose information log is to be queried.

maxLength
Specifies the size of the character buffer for storing the returned information log.

length Returns the length of the string returned in *infoLog* (excluding the null terminator).

infoLog Specifies an array of characters that is used to return the information log.

`glGetProgramInfoLog` returns the information log for the specified program object. The information log for a program object is modified when the program object is linked or validated. The string that is returned will be null terminated.

`glGetProgramInfoLog` returns in *infoLog* as much of the information log as it can, up to a maximum of *maxLength* characters. The number of characters actually returned, excluding the null termination character, is specified by *length*. If the length of the returned string is not required, a value of `NULL` can be passed in the *length* argument. The size of the buffer required to store the returned information log can be obtained by calling `glGetProgram` with the value `GL_INFO_LOG_LENGTH`.

The information log for a program object is either an empty string, or a string containing information about the last link operation, or a string containing information about the last validation operation. It may contain diagnostic messages, warning messages, and other information. When a program object is created, its information log will be a string of length 0.

`GL_INVALID_VALUE` is generated if *program* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *program* is not a program object.

`GL_INVALID_VALUE` is generated if *maxLength* is less than 0.

`GL_INVALID_OPERATION` is generated if `glGetProgramInfoLog` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGetProgramiv` *program pname params* [Function]
Returns a parameter from a program object.

program Specifies the program object to be queried.

pname Specifies the object parameter. Accepted symbolic names are `GL_DELETE_STATUS`, `GL_LINK_STATUS`, `GL_VALIDATE_STATUS`, `GL_INFO_LOG_LENGTH`, `GL_ATTACHED_SHADERS`, `GL_ACTIVE_ATTRIBUTES`, `GL_ACTIVE_ATTRIBUTE_MAX_LENGTH`, `GL_ACTIVE_UNIFORMS`, `GL_ACTIVE_UNIFORM_MAX_LENGTH`.

params Returns the requested object parameter.

`glGetProgram` returns in *params* the value of a parameter for a specific program object. The following parameters are defined:

`GL_DELETE_STATUS`
: *params* returns `GL_TRUE` if *program* is currently flagged for deletion, and `GL_FALSE` otherwise.

`GL_LINK_STATUS`
: *params* returns `GL_TRUE` if the last link operation on *program* was successful, and `GL_FALSE` otherwise.

`GL_VALIDATE_STATUS`
: *params* returns `GL_TRUE` or if the last validation operation on *program* was successful, and `GL_FALSE` otherwise.

`GL_INFO_LOG_LENGTH`
: *params* returns the number of characters in the information log for *program* including the null termination character (i.e., the size of the character buffer required to store the information log). If *program* has no information log, a value of 0 is returned.

`GL_ATTACHED_SHADERS`
: *params* returns the number of shader objects attached to *program*.

`GL_ACTIVE_ATTRIBUTES`
: *params* returns the number of active attribute variables for *program*.

`GL_ACTIVE_ATTRIBUTE_MAX_LENGTH`
: *params* returns the length of the longest active attribute name for *program*, including the null termination character (i.e., the size of the character buffer required to store the longest attribute name). If no active attributes exist, 0 is returned.

`GL_ACTIVE_UNIFORMS`
: *params* returns the number of active uniform variables for *program*.

`GL_ACTIVE_UNIFORM_MAX_LENGTH`
: *params* returns the length of the longest active uniform variable name for *program*, including the null termination character (i.e., the size of the character buffer required to store the longest uniform variable name). If no active uniform variables exist, 0 is returned.

`GL_INVALID_VALUE` is generated if *program* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *program* does not refer to a program object.

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

`GL_INVALID_OPERATION` is generated if `glGetProgram` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGetQueryiv` *target pname params* [Function]

Return parameters of a query object target.

target Specifies a query object target. Must be `GL_SAMPLES_PASSED`.

pname Specifies the symbolic name of a query object target parameter. Accepted values are `GL_CURRENT_QUERY` or `GL_QUERY_COUNTER_BITS`.

params Returns the requested data.

`glGetQueryiv` returns in *params* a selected parameter of the query object target specified by *target*.

pname names a specific query object target parameter. When *target* is `GL_SAMPLES_PASSED`, *pname* can be as follows:

`GL_CURRENT_QUERY`
: *params* returns the name of the currently active occlusion query object. If no occlusion query is active, 0 is returned. The initial value is 0.

`GL_QUERY_COUNTER_BITS`
: *params* returns the number of bits in the query counter used to accumulate passing samples. If the number of bits returned is 0, the implementation does not support a query counter, and the results obtained from `glGetQueryObject` are useless.

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

`GL_INVALID_OPERATION` is generated if `glGetQueryiv` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGetQueryObjectiv` *id pname params* [Function]
`void glGetQueryObjectuiv` *id pname params* [Function]

Return parameters of a query object.

id Specifies the name of a query object.

pname Specifies the symbolic name of a query object parameter. Accepted values are `GL_QUERY_RESULT` or `GL_QUERY_RESULT_AVAILABLE`.

params Returns the requested data.

`glGetQueryObject` returns in *params* a selected parameter of the query object specified by *id*.

pname names a specific query object parameter. *pname* can be as follows:

`GL_QUERY_RESULT`
: *params* returns the value of the query object's passed samples counter. The initial value is 0.

`GL_QUERY_RESULT_AVAILABLE`
: *params* returns whether the passed samples counter is immediately available. If a delay would occur waiting for the query result, `GL_FALSE` is returned. Otherwise, `GL_TRUE` is returned, which also indicates that the results of all previous queries are available as well.

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

`GL_INVALID_OPERATION` is generated if *id* is not the name of a query object.

`GL_INVALID_OPERATION` is generated if *id* is the name of a currently active query object.

`GL_INVALID_OPERATION` is generated if `glGetQueryObject` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGetSeparableFilter` *target format type row column span* [Function]

Get separable convolution filter kernel images.

target The separable filter to be retrieved. Must be `GL_SEPARABLE_2D`.

format Format of the output images. Must be one of `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`, `GL_RGB`, `GL_BGRGL_RGBA`, `GL_BGRA`, `GL_LUMINANCE`, or `GL_LUMINANCE_ALPHA`.

type Data type of components in the output images. Symbolic constants `GL_UNSIGNED_BYTE`, `GL_BYTE`, `GL_BITMAP`, `GL_UNSIGNED_SHORT`, `GL_SHORT`, `GL_UNSIGNED_INT`, `GL_INT`, `GL_FLOAT`, `GL_UNSIGNED_BYTE_3_3_2`, `GL_UNSIGNED_BYTE_2_3_3_REV`, `GL_UNSIGNED_SHORT_5_6_5`, `GL_UNSIGNED_SHORT_5_6_5_REV`, `GL_UNSIGNED_SHORT_4_4_4_4`, `GL_UNSIGNED_SHORT_4_4_4_4_REV`, `GL_UNSIGNED_SHORT_5_5_5_1`, `GL_UNSIGNED_SHORT_1_5_5_5_REV`, `GL_UNSIGNED_INT_8_8_8_8`, `GL_UNSIGNED_INT_8_8_8_8_REV`, `GL_UNSIGNED_INT_10_10_10_2`, and `GL_UNSIGNED_INT_2_10_10_10_REV` are accepted.

row Pointer to storage for the row filter image.

column Pointer to storage for the column filter image.

span Pointer to storage for the span filter image (currently unused).

`glGetSeparableFilter` returns the two one-dimensional filter kernel images for the current separable 2D convolution filter. The row image is placed in *row* and the column image is placed in *column* according to the specifications in *format* and *type*. (In the current implementation, *span* is not affected in any way.) No pixel transfer operations are performed on the images, but the relevant pixel storage modes are applied.

If a non-zero named buffer object is bound to the `GL_PIXEL_PACK_BUFFER` target (see `glBindBuffer`) while a separable convolution filter is requested, *row*, *column*, and *span* are treated as a byte offset into the buffer object's data store.

Color components that are present in *format* but not included in the internal format of the filters are returned as zero. The assignments of internal color components to the components of *format* are as follows:

Internal Component	**Resulting Component**
Red	Red
Green	Green
Blue	Blue
Alpha	Alpha

Luminance
: Red

Intensity
: Red

`GL_INVALID_ENUM` is generated if *target* is not `GL_SEPARABLE_2D`.

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

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

`GL_INVALID_OPERATION` is generated if *type* is one of `GL_UNSIGNED_BYTE_3_3_2`, `GL_UNSIGNED_BYTE_2_3_3_REV`, `GL_UNSIGNED_SHORT_5_6_5`, or `GL_UNSIGNED_SHORT_5_6_5_REV` and *format* is not `GL_RGB`.

`GL_INVALID_OPERATION` is generated if *type* is one of `GL_UNSIGNED_SHORT_4_4_4_4`, `GL_UNSIGNED_SHORT_4_4_4_4_REV`, `GL_UNSIGNED_SHORT_5_5_5_1`, `GL_UNSIGNED_SHORT_1_5_5_5_REV`, `GL_UNSIGNED_INT_8_8_8_8`, `GL_UNSIGNED_INT_8_8_8_8_REV`, `GL_UNSIGNED_INT_10_10_10_2`, or `GL_UNSIGNED_INT_2_10_10_10_REV` and *format* is neither `GL_RGBA` nor `GL_BGRA`.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_PACK_BUFFER` target and the buffer object's data store is currently mapped.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_PACK_BUFFER` target and the data would be packed to the buffer object such that the memory writes required would exceed the data store size.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_PACK_BUFFER` target and *row* or *column* is not evenly divisible into the number of bytes needed to store in memory a datum indicated by *type*.

`GL_INVALID_OPERATION` is generated if `glGetSeparableFilter` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGetShaderInfoLog` *shader maxLength length infoLog* [Function]

Returns the information log for a shader object.

shader
: Specifies the shader object whose information log is to be queried.

maxLength
: Specifies the size of the character buffer for storing the returned information log.

length
: Returns the length of the string returned in *infoLog* (excluding the null terminator).

infoLog
: Specifies an array of characters that is used to return the information log.

`glGetShaderInfoLog` returns the information log for the specified shader object. The information log for a shader object is modified when the shader is compiled. The string that is returned will be null terminated.

`glGetShaderInfoLog` returns in *infoLog* as much of the information log as it can, up to a maximum of *maxLength* characters. The number of characters actually returned, excluding the null termination character, is specified by *length*. If the length of the returned string is not required, a value of `NULL` can be passed in the *length* argument.

The size of the buffer required to store the returned information log can be obtained by calling `glGetShader` with the value `GL_INFO_LOG_LENGTH`.

The information log for a shader object is a string that may contain diagnostic messages, warning messages, and other information about the last compile operation. When a shader object is created, its information log will be a string of length 0.

`GL_INVALID_VALUE` is generated if *shader* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *shader* is not a shader object.

`GL_INVALID_VALUE` is generated if *maxLength* is less than 0.

`GL_INVALID_OPERATION` is generated if `glGetShaderInfoLog` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGetShaderSource` *shader bufSize length source* [Function]

Returns the source code string from a shader object.

shader Specifies the shader object to be queried.

bufSize Specifies the size of the character buffer for storing the returned source code string.

length Returns the length of the string returned in *source* (excluding the null terminator).

source Specifies an array of characters that is used to return the source code string.

`glGetShaderSource` returns the concatenation of the source code strings from the shader object specified by *shader*. The source code strings for a shader object are the result of a previous call to `glShaderSource`. The string returned by the function will be null terminated.

`glGetShaderSource` returns in *source* as much of the source code string as it can, up to a maximum of *bufSize* characters. The number of characters actually returned, excluding the null termination character, is specified by *length*. If the length of the returned string is not required, a value of `NULL` can be passed in the *length* argument. The size of the buffer required to store the returned source code string can be obtained by calling `glGetShader` with the value `GL_SHADER_SOURCE_LENGTH`.

`GL_INVALID_VALUE` is generated if *shader* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *shader* is not a shader object.

`GL_INVALID_VALUE` is generated if *bufSize* is less than 0.

`GL_INVALID_OPERATION` is generated if `glGetShaderSource` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGetShaderiv` *shader pname params* [Function]

Returns a parameter from a shader object.

shader Specifies the shader object to be queried.

pname Specifies the object parameter. Accepted symbolic names are `GL_SHADER_TYPE`, `GL_DELETE_STATUS`, `GL_COMPILE_STATUS`, `GL_INFO_LOG_LENGTH`, `GL_SHADER_SOURCE_LENGTH`.

params Returns the requested object parameter.

`glGetShader` returns in *params* the value of a parameter for a specific shader object. The following parameters are defined:

`GL_SHADER_TYPE`
: *params* returns `GL_VERTEX_SHADER` if *shader* is a vertex shader object, and `GL_FRAGMENT_SHADER` if *shader* is a fragment shader object.

`GL_DELETE_STATUS`
: *params* returns `GL_TRUE` if *shader* is currently flagged for deletion, and `GL_FALSE` otherwise.

`GL_COMPILE_STATUS`
: *params* returns `GL_TRUE` if the last compile operation on *shader* was successful, and `GL_FALSE` otherwise.

`GL_INFO_LOG_LENGTH`
: *params* returns the number of characters in the information log for *shader* including the null termination character (i.e., the size of the character buffer required to store the information log). If *shader* has no information log, a value of 0 is returned.

`GL_SHADER_SOURCE_LENGTH`
: *params* returns the length of the concatenation of the source strings that make up the shader source for the *shader*, including the null termination character. (i.e., the size of the character buffer required to store the shader source). If no source code exists, 0 is returned.

`GL_INVALID_VALUE` is generated if *shader* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *shader* does not refer to a shader object.

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

`GL_INVALID_OPERATION` is generated if `glGetShader` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`const-GLubyte* glGetString` *name* [Function]

Return a string describing the current GL connection.

name Specifies a symbolic constant, one of `GL_VENDOR`, `GL_RENDERER`, `GL_VERSION`, `GL_SHADING_LANGUAGE_VERSION`, or `GL_EXTENSIONS`.

`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_SHADING_LANGUAGE_VERSION`
: Returns a version or release number for the shading language.

`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` and `GL_SHADING_LANGUAGE_VERSION` strings begin with a version number. The version number uses one of these forms:

major_number.minor_numbermajor_number.minor_number.release_number

Vendor-specific information may follow the version number. Its format depends on the implementation, but a space always separates the version number and the vendor-specific information.

All strings are null-terminated.

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

`GL_INVALID_OPERATION` is generated if `glGetString` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGetTexEnvfv` *target pname params* [Function]
`void glGetTexEnviv` *target pname params* [Function]
Return texture environment parameters.

target Specifies a texture environment. May be `GL_TEXTURE_ENV`, `GL_TEXTURE_FILTER_CONTROL`, or `GL_POINT_SPRITE`.

pname Specifies the symbolic name of a texture environment parameter. Accepted values are `GL_TEXTURE_ENV_MODE`, `GL_TEXTURE_ENV_COLOR`, `GL_TEXTURE_LOD_BIAS`, `GL_COMBINE_RGB`, `GL_COMBINE_ALPHA`, `GL_SRC0_RGB`, `GL_SRC1_RGB`, `GL_SRC2_RGB`, `GL_SRC0_ALPHA`, `GL_SRC1_ALPHA`, `GL_SRC2_ALPHA`, `GL_OPERAND0_RGB`, `GL_OPERAND1_RGB`, `GL_OPERAND2_RGB`, `GL_OPERAND0_ALPHA`, `GL_OPERAND1_ALPHA`, `GL_OPERAND2_ALPHA`, `GL_RGB_SCALE`, `GL_ALPHA_SCALE`, or `GL_COORD_REPLACE`.

params Returns the requested data.

`glGetTexEnv` returns in *params* selected values of a texture environment that was specified with `glTexEnv`. *target* specifies a texture environment.

When *target* is GL_TEXTURE_FILTER_CONTROL, *pname* must be GL_TEXTURE_LOD_BIAS. When *target* is GL_POINT_SPRITE, *pname* must be GL_COORD_REPLACE. When *target* is GL_TEXTURE_ENV, *pname* can be GL_TEXTURE_ENV_MODE, GL_TEXTURE_ENV_COLOR, GL_COMBINE_RGB, GL_COMBINE_ALPHA, GL_RGB_SCALE, GL_ALPHA_SCALE, GL_SRC0_RGB, GL_SRC1_RGB, GL_SRC2_RGB, GL_SRC0_ALPHA, GL_SRC1_ALPHA, or GL_SRC2_ALPHA.

pname names a specific texture environment parameter, as follows:

GL_TEXTURE_ENV_MODE
: *params* returns the single-valued texture environment mode, a symbolic constant. The initial value is GL_MODULATE.

GL_TEXTURE_ENV_COLOR
: *params* returns four integer or floating-point values that are the texture environment color. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer, and -1.0 maps to the most negative representable integer. The initial value is (0, 0, 0, 0).

GL_TEXTURE_LOD_BIAS
: *params* returns a single floating-point value that is the texture level-of-detail bias. The initial value is 0.

GL_COMBINE_RGB
: *params* returns a single symbolic constant value representing the current RGB combine mode. The initial value is GL_MODULATE.

GL_COMBINE_ALPHA
: *params* returns a single symbolic constant value representing the current alpha combine mode. The initial value is GL_MODULATE.

GL_SRC0_RGB
: *params* returns a single symbolic constant value representing the texture combiner zero's RGB source. The initial value is GL_TEXTURE.

GL_SRC1_RGB
: *params* returns a single symbolic constant value representing the texture combiner one's RGB source. The initial value is GL_PREVIOUS.

GL_SRC2_RGB
: *params* returns a single symbolic constant value representing the texture combiner two's RGB source. The initial value is GL_CONSTANT.

GL_SRC0_ALPHA
: *params* returns a single symbolic constant value representing the texture combiner zero's alpha source. The initial value is GL_TEXTURE.

GL_SRC1_ALPHA
: *params* returns a single symbolic constant value representing the texture combiner one's alpha source. The initial value is GL_PREVIOUS.

GL_SRC2_ALPHA
: *params* returns a single symbolic constant value representing the texture combiner two's alpha source. The initial value is GL_CONSTANT.

`GL_OPERAND0_RGB`
: *params* returns a single symbolic constant value representing the texture combiner zero's RGB operand. The initial value is `GL_SRC_COLOR`.

`GL_OPERAND1_RGB`
: *params* returns a single symbolic constant value representing the texture combiner one's RGB operand. The initial value is `GL_SRC_COLOR`.

`GL_OPERAND2_RGB`
: *params* returns a single symbolic constant value representing the texture combiner two's RGB operand. The initial value is `GL_SRC_ALPHA`.

`GL_OPERAND0_ALPHA`
: *params* returns a single symbolic constant value representing the texture combiner zero's alpha operand. The initial value is `GL_SRC_ALPHA`.

`GL_OPERAND1_ALPHA`
: *params* returns a single symbolic constant value representing the texture combiner one's alpha operand. The initial value is `GL_SRC_ALPHA`.

`GL_OPERAND2_ALPHA`
: *params* returns a single symbolic constant value representing the texture combiner two's alpha operand. The initial value is `GL_SRC_ALPHA`.

`GL_RGB_SCALE`
: *params* returns a single floating-point value representing the current RGB texture combiner scaling factor. The initial value is 1.0.

`GL_ALPHA_SCALE`
: *params* returns a single floating-point value representing the current alpha texture combiner scaling factor. The initial value is 1.0.

`GL_COORD_REPLACE`
: *params* returns a single boolean value representing the current point sprite texture coordinate replacement enable state. The initial value is `GL_FALSE`.

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

`GL_INVALID_OPERATION` is generated if `glGetTexEnv` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGetTexGendv` *coord pname params* [Function]
`void glGetTexGenfv` *coord pname params* [Function]
`void glGetTexGeniv` *coord pname params* [Function]

Return texture coordinate generation parameters.

coord Specifies a texture coordinate. Must be `GL_S`, `GL_T`, `GL_R`, or `GL_Q`.

pname Specifies the symbolic name of the value(s) to be returned. Must be either `GL_TEXTURE_GEN_MODE` or the name of one of the texture generation plane equations: `GL_OBJECT_PLANE` or `GL_EYE_PLANE`.

params Returns the requested data.

`glGetTexGen` returns in *params* selected parameters of a texture coordinate generation function that was specified using `glTexGen`. *coord* names one of the (s, t, r, q) texture coordinates, using the symbolic constant `GL_S`, `GL_T`, `GL_R`, or `GL_Q`.

pname specifies one of three symbolic names:

`GL_TEXTURE_GEN_MODE`
: *params* returns the single-valued texture generation function, a symbolic constant. The initial value is `GL_EYE_LINEAR`.

`GL_OBJECT_PLANE`
: *params* returns the four plane equation coefficients that specify object linear-coordinate generation. Integer values, when requested, are mapped directly from the internal floating-point representation.

`GL_EYE_PLANE`
: *params* returns the four plane equation coefficients that specify eye linear-coordinate generation. Integer values, when requested, are mapped directly from the internal floating-point representation. The returned values are those maintained in eye coordinates. They are not equal to the values specified using `glTexGen`, unless the modelview matrix was identity when `glTexGen` was called.

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

`GL_INVALID_OPERATION` is generated if `glGetTexGen` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGetTexImage` *target level format type img* [Function]

Return a texture image.

target
: Specifies which texture is to be obtained. `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, `GL_TEXTURE_3D`, `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Z`, and `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z` are accepted.

level
: Specifies the level-of-detail number of the desired image. Level 0 is the base image level. Level n is the nth mipmap reduction image.

format
: Specifies a pixel format for the returned data. The supported formats are `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`, `GL_RGB`, `GL_BGR`, `GL_RGBA`, `GL_BGRA`, `GL_LUMINANCE`, and `GL_LUMINANCE_ALPHA`.

type
: Specifies a pixel type for the returned data. The supported types are `GL_UNSIGNED_BYTE`, `GL_BYTE`, `GL_UNSIGNED_SHORT`, `GL_SHORT`, `GL_UNSIGNED_INT`, `GL_INT`, `GL_FLOAT`, `GL_UNSIGNED_BYTE_3_3_2`, `GL_UNSIGNED_BYTE_2_3_3_REV`, `GL_UNSIGNED_SHORT_5_6_5`, `GL_UNSIGNED_SHORT_5_6_5_REV`, `GL_UNSIGNED_SHORT_4_4_4_4`, `GL_UNSIGNED_SHORT_4_4_4_4_REV`, `GL_UNSIGNED_SHORT_5_5_5_1`, `GL_UNSIGNED_SHORT_1_5_5_5_REV`, `GL_UNSIGNED_INT_8_8_8_8`, `GL_UNSIGNED_INT_8_8_8_8_REV`, `GL_UNSIGNED_INT_10_10_10_2`, and `GL_UNSIGNED_INT_2_10_10_10_REV`.

img Returns the texture image. Should be a pointer to an array of the type specified by *type*.

`glGetTexImage` returns a texture image into *img*. *target* specifies whether the desired texture image is one specified by `glTexImage1D` (`GL_TEXTURE_1D`), `glTexImage2D` (`GL_TEXTURE_2D` or any of `GL_TEXTURE_CUBE_MAP_*`), or `glTexImage3D` (`GL_TEXTURE_3D`). *level* specifies the level-of-detail number of the desired image. *format* and *type* specify the format and type of the desired image array. See the reference pages `glTexImage1D` and `glDrawPixels` for a description of the acceptable values for the *format* and *type* parameters, respectively.

If a non-zero named buffer object is bound to the `GL_PIXEL_PACK_BUFFER` target (see `glBindBuffer`) while a texture image is requested, *img* is treated as a byte offset into the buffer object's data store.

To understand the operation of `glGetTexImage`, consider the selected internal four-component texture image to be an RGBA color buffer the size of the image. The semantics of `glGetTexImage` are then identical to those of `glReadPixels`, with the exception that no pixel transfer operations are performed, when called with the same *format* and *type*, with *x* and *y* set to 0, *width* set to the width of the texture image (including border if one was specified), and *height* set to 1 for 1D images, or to the height of the texture image (including border if one was specified) for 2D images. Because the internal texture image is an RGBA image, pixel formats `GL_COLOR_INDEX`, `GL_STENCIL_INDEX`, and `GL_DEPTH_COMPONENT` are not accepted, and pixel type `GL_BITMAP` is not accepted.

If the selected texture image does not contain four components, the following mappings are applied. Single-component textures are treated as RGBA buffers with red set to the single-component value, green set to 0, blue set to 0, and alpha set to 1. Two-component textures are treated as RGBA buffers with red set to the value of component zero, alpha set to the value of component one, and green and blue set to 0. Finally, three-component textures are treated as RGBA buffers with red set to component zero, green set to component one, blue set to component two, and alpha set to 1.

To determine the required size of *img*, use `glGetTexLevelParameter` to determine the dimensions of the internal texture image, then scale the required number of pixels by the storage required for each pixel, based on *format* and *type*. Be sure to take the pixel storage parameters into account, especially `GL_PACK_ALIGNMENT`.

`GL_INVALID_ENUM` is generated if *target*, *format*, or *type* is not an accepted value.

`GL_INVALID_VALUE` is generated if *level* is less than 0.

`GL_INVALID_VALUE` may be generated if *level* is greater than *log_2(max,)*, where *max* is the returned value of `GL_MAX_TEXTURE_SIZE`.

`GL_INVALID_OPERATION` is returned if *type* is one of `GL_UNSIGNED_BYTE_3_3_2`, `GL_UNSIGNED_BYTE_2_3_3_REV`, `GL_UNSIGNED_SHORT_5_6_5`, or `GL_UNSIGNED_SHORT_5_6_5_REV` and *format* is not `GL_RGB`.

`GL_INVALID_OPERATION` is returned if *type* is one of `GL_UNSIGNED_SHORT_4_4_4_4`, `GL_UNSIGNED_SHORT_4_4_4_4_REV`, `GL_UNSIGNED_SHORT_5_5_5_1`, `GL_UNSIGNED_SHORT_1_5_5_5_REV`, `GL_UNSIGNED_INT_8_8_8_8`, `GL_UNSIGNED_INT_8_8_8_8_REV`,

GL_UNSIGNED_INT_10_10_10_2, or GL_UNSIGNED_INT_2_10_10_10_REV, and *format* is neither GL_RGBA or GL_BGRA.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_PACK_BUFFER target and the buffer object's data store is currently mapped.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_PACK_BUFFER target and the data would be packed to the buffer object such that the memory writes required would exceed the data store size.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_PACK_BUFFER target and *img* is not evenly divisible into the number of bytes needed to store in memory a datum indicated by *type*.

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

`void glGetTexLevelParameterfv` *target level pname params* [Function]
`void glGetTexLevelParameteriv` *target level pname params* [Function]

Return texture parameter values for a specific level of detail.

target Specifies the symbolic name of the target texture, either GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D, GL_PROXY_TEXTURE_1D, GL_PROXY_TEXTURE_2D, GL_PROXY_TEXTURE_3D, GL_TEXTURE_CUBE_MAP_POSITIVE_X, GL_TEXTURE_CUBE_MAP_NEGATIVE_X, GL_TEXTURE_CUBE_MAP_POSITIVE_Y, GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, GL_TEXTURE_CUBE_MAP_POSITIVE_Z, GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, or GL_PROXY_TEXTURE_CUBE_MAP.

level Specifies the level-of-detail number of the desired image. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.

pname Specifies the symbolic name of a texture parameter. GL_TEXTURE_WIDTH, GL_TEXTURE_HEIGHT, GL_TEXTURE_DEPTH, GL_TEXTURE_INTERNAL_FORMAT, GL_TEXTURE_BORDER, GL_TEXTURE_RED_SIZE, GL_TEXTURE_GREEN_SIZE, GL_TEXTURE_BLUE_SIZE, GL_TEXTURE_ALPHA_SIZE, GL_TEXTURE_LUMINANCE_SIZE, GL_TEXTURE_INTENSITY_SIZE, GL_TEXTURE_DEPTH_SIZE, GL_TEXTURE_COMPRESSED, and GL_TEXTURE_COMPRESSED_IMAGE_SIZE are accepted.

params Returns the requested data.

glGetTexLevelParameter returns in *params* texture parameter values for a specific level-of-detail value, specified as *level*. *target* defines the target texture, either GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D, GL_PROXY_TEXTURE_1D, GL_PROXY_TEXTURE_2D, GL_PROXY_TEXTURE_3D, GL_TEXTURE_CUBE_MAP_POSITIVE_X, GL_TEXTURE_CUBE_MAP_NEGATIVE_X, GL_TEXTURE_CUBE_MAP_POSITIVE_Y, GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, GL_TEXTURE_CUBE_MAP_POSITIVE_Z, GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, or GL_PROXY_TEXTURE_CUBE_MAP.

GL_MAX_TEXTURE_SIZE, and GL_MAX_3D_TEXTURE_SIZE are not really descriptive enough. It has to report the largest square texture image that can be accommodated with mipmaps and borders, but a long skinny texture, or a texture without mipmaps and borders, may easily fit in texture memory. The proxy targets allow the user

to more accurately query whether the GL can accommodate a texture of a given configuration. If the texture cannot be accommodated, the texture state variables, which may be queried with `glGetTexLevelParameter`, are set to 0. If the texture can be accommodated, the texture state values will be set as they would be set for a non-proxy target.

pname specifies the texture parameter whose value or values will be returned.

The accepted parameter names are as follows:

`GL_TEXTURE_WIDTH`
: *params* returns a single value, the width of the texture image. This value includes the border of the texture image. The initial value is 0.

`GL_TEXTURE_HEIGHT`
: *params* returns a single value, the height of the texture image. This value includes the border of the texture image. The initial value is 0.

`GL_TEXTURE_DEPTH`
: *params* returns a single value, the depth of the texture image. This value includes the border of the texture image. The initial value is 0.

`GL_TEXTURE_INTERNAL_FORMAT`
: *params* returns a single value, the internal format of the texture image.

`GL_TEXTURE_BORDER`
: *params* returns a single value, the width in pixels of the border of the texture image. The initial value is 0.

`GL_TEXTURE_RED_SIZE`,
`GL_TEXTURE_GREEN_SIZE`,
`GL_TEXTURE_BLUE_SIZE`,
`GL_TEXTURE_ALPHA_SIZE`,
`GL_TEXTURE_LUMINANCE_SIZE`,
`GL_TEXTURE_INTENSITY_SIZE`,
`GL_TEXTURE_DEPTH_SIZE`
: The internal storage resolution of an individual component. The resolution chosen by the GL will be a close match for the resolution requested by the user with the component argument of `glTexImage1D`, `glTexImage2D`, `glTexImage3D`, `glCopyTexImage1D`, and `glCopyTexImage2D`. The initial value is 0.

`GL_TEXTURE_COMPRESSED`
: *params* returns a single boolean value indicating if the texture image is stored in a compressed internal format. The initiali value is `GL_FALSE`.

`GL_TEXTURE_COMPRESSED_IMAGE_SIZE`
: *params* returns a single integer value, the number of unsigned bytes of the compressed texture image that would be returned from `glGetCompressedTexImage`.

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

`GL_INVALID_VALUE` is generated if *level* is less than 0.

GL_INVALID_VALUE may be generated if *level* is greater than log_2max, where *max* is the returned value of GL_MAX_TEXTURE_SIZE.

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

GL_INVALID_OPERATION is generated if GL_TEXTURE_COMPRESSED_IMAGE_SIZE is queried on texture images with an uncompressed internal format or on proxy targets.

`void glGetTexParameterfv` *target pname params* [Function]
`void glGetTexParameteriv` *target pname params* [Function]

Return texture parameter values.

target Specifies the symbolic name of the target texture. GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D, and GL_TEXTURE_CUBE_MAP are accepted.

pname Specifies the symbolic name of a texture parameter. GL_TEXTURE_MAG_FILTER, GL_TEXTURE_MIN_FILTER, GL_TEXTURE_MIN_LOD, GL_TEXTURE_MAX_LOD, GL_TEXTURE_BASE_LEVEL, GL_TEXTURE_MAX_LEVEL, GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T, GL_TEXTURE_WRAP_R, GL_TEXTURE_BORDER_COLOR, GL_TEXTURE_PRIORITY, GL_TEXTURE_RESIDENT, GL_TEXTURE_COMPARE_MODE, GL_TEXTURE_COMPARE_FUNC, GL_DEPTH_TEXTURE_MODE, and GL_GENERATE_MIPMAP are accepted.

params Returns the texture parameters.

glGetTexParameter returns in *params* the value or values of the texture parameter specified as *pname*. *target* defines the target texture, either GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D, or GL_TEXTURE_CUBE_MAP, to specify one-, two-, or three-dimensional or cube-mapped texturing. *pname* accepts the same symbols as glTexParameter, with the same interpretations:

GL_TEXTURE_MAG_FILTER
: Returns the single-valued texture magnification filter, a symbolic constant. The initial value is GL_LINEAR.

GL_TEXTURE_MIN_FILTER
: Returns the single-valued texture minification filter, a symbolic constant. The initial value is GL_NEAREST_MIPMAP_LINEAR.

GL_TEXTURE_MIN_LOD
: Returns the single-valued texture minimum level-of-detail value. The initial value is -1000.

GL_TEXTURE_MAX_LOD
: Returns the single-valued texture maximum level-of-detail value. The initial value is 1000.

GL_TEXTURE_BASE_LEVEL
: Returns the single-valued base texture mipmap level. The initial value is 0.

GL_TEXTURE_MAX_LEVEL
: Returns the single-valued maximum texture mipmap array level. The initial value is 1000.

`GL_TEXTURE_WRAP_S`
: Returns the single-valued wrapping function for texture coordinate s, a symbolic constant. The initial value is `GL_REPEAT`.

`GL_TEXTURE_WRAP_T`
: Returns the single-valued wrapping function for texture coordinate t, a symbolic constant. The initial value is `GL_REPEAT`.

`GL_TEXTURE_WRAP_R`
: Returns the single-valued wrapping function for texture coordinate r, a symbolic constant. The initial value is `GL_REPEAT`.

`GL_TEXTURE_BORDER_COLOR`
: Returns four integer or floating-point numbers that comprise the RGBA color of the texture border. Floating-point values are returned in the range [0,1]. Integer values are returned as a linear mapping of the internal floating-point representation such that 1.0 maps to the most positive representable integer and -1.0 maps to the most negative representable integer. The initial value is (0, 0, 0, 0).

`GL_TEXTURE_PRIORITY`
: Returns the residence priority of the target texture (or the named texture bound to it). The initial value is 1. See `glPrioritizeTextures`.

`GL_TEXTURE_RESIDENT`
: Returns the residence status of the target texture. If the value returned in *params* is `GL_TRUE`, the texture is resident in texture memory. See `glAreTexturesResident`.

`GL_TEXTURE_COMPARE_MODE`
: Returns a single-valued texture comparison mode, a symbolic constant. The initial value is `GL_NONE`. See `glTexParameter`.

`GL_TEXTURE_COMPARE_FUNC`
: Returns a single-valued texture comparison function, a symbolic constant. The initial value is `GL_LEQUAL`. See `glTexParameter`.

`GL_DEPTH_TEXTURE_MODE`
: Returns a single-valued texture format indicating how the depth values should be converted into color components. The initial value is `GL_LUMINANCE`. See `glTexParameter`.

`GL_GENERATE_MIPMAP`
: Returns a single boolean value indicating if automatic mipmap level updates are enabled. See `glTexParameter`.

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

`GL_INVALID_OPERATION` is generated if `glGetTexParameter` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`GLint glGetUniformLocation` *program name* [Function]

Returns the location of a uniform variable.

program Specifies the program object to be queried.

name Points to a null terminated string containing the name of the uniform variable whose location is to be queried.

`glGetUniformLocation` returns an integer that represents the location of a specific uniform variable within a program object. *name* must be a null terminated string that contains no white space. *name* must be an active uniform variable name in *program* that is not a structure, an array of structures, or a subcomponent of a vector or a matrix. This function returns -1 if *name* does not correspond to an active uniform variable in *program* or if *name* starts with the reserved prefix "gl_".

Uniform variables that are structures or arrays of structures may be queried by calling `glGetUniformLocation` for each field within the structure. The array element operator "[]" and the structure field operator "." may be used in *name* in order to select elements within an array or fields within a structure. The result of using these operators is not allowed to be another structure, an array of structures, or a subcomponent of a vector or a matrix. Except if the last part of *name* indicates a uniform variable array, the location of the first element of an array can be retrieved by using the name of the array, or by using the name appended by "[0]".

The actual locations assigned to uniform variables are not known until the program object is linked successfully. After linking has occurred, the command `glGetUniformLocation` can be used to obtain the location of a uniform variable. This location value can then be passed to `glUniform` to set the value of the uniform variable or to `glGetUniform` in order to query the current value of the uniform variable. After a program object has been linked successfully, the index values for uniform variables remain fixed until the next link command occurs. Uniform variable locations and values can only be queried after a link if the link was successful.

`GL_INVALID_VALUE` is generated if *program* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *program* is not a program object.

`GL_INVALID_OPERATION` is generated if *program* has not been successfully linked.

`GL_INVALID_OPERATION` is generated if `glGetUniformLocation` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGetUniformfv` *program location params* [Function]
`void glGetUniformiv` *program location params* [Function]

Returns the value of a uniform variable.

program Specifies the program object to be queried.

location Specifies the location of the uniform variable to be queried.

params Returns the value of the specified uniform variable.

`glGetUniform` returns in *params* the value(s) of the specified uniform variable. The type of the uniform variable specified by *location* determines the number of values returned. If the uniform variable is defined in the shader as a boolean, int, or float, a single value will be returned. If it is defined as a vec2, ivec2, or bvec2, two values will be returned. If it is defined as a vec3, ivec3, or bvec3, three values will be returned, and so on. To query values stored in uniform variables declared as arrays,

call `glGetUniform` for each element of the array. To query values stored in uniform variables declared as structures, call `glGetUniform` for each field in the structure. The values for uniform variables declared as a matrix will be returned in column major order.

The locations assigned to uniform variables are not known until the program object is linked. After linking has occurred, the command `glGetUniformLocation` can be used to obtain the location of a uniform variable. This location value can then be passed to `glGetUniform` in order to query the current value of the uniform variable. After a program object has been linked successfully, the index values for uniform variables remain fixed until the next link command occurs. The uniform variable values can only be queried after a link if the link was successful.

`GL_INVALID_VALUE` is generated if *program* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *program* is not a program object.

`GL_INVALID_OPERATION` is generated if *program* has not been successfully linked.

`GL_INVALID_OPERATION` is generated if *location* does not correspond to a valid uniform variable location for the specified program object.

`GL_INVALID_OPERATION` is generated if `glGetUniform` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glGetVertexAttribPointerv` *index pname pointer* [Function]

Return the address of the specified generic vertex attribute pointer.

index Specifies the generic vertex attribute parameter to be returned.

pname Specifies the symbolic name of the generic vertex attribute parameter to be returned. Must be `GL_VERTEX_ATTRIB_ARRAY_POINTER`.

pointer Returns the pointer value.

`glGetVertexAttribPointerv` returns pointer information. *index* is the generic vertex attribute to be queried, *pname* is a symbolic constant indicating the pointer to be returned, and *params* is a pointer to a location in which to place the returned data.

If a non-zero named buffer object was bound to the `GL_ARRAY_BUFFER` target (see `glBindBuffer`) when the desired pointer was previously specified, the *pointer* returned is a byte offset into the buffer object's data store.

`GL_INVALID_VALUE` is generated if *index* is greater than or equal to `GL_MAX_VERTEX_ATTRIBS`.

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

`void glGetVertexAttribdv` *index pname params* [Function]
`void glGetVertexAttribfv` *index pname params* [Function]
`void glGetVertexAttribiv` *index pname params* [Function]

Return a generic vertex attribute parameter.

index Specifies the generic vertex attribute parameter to be queried.

pname Specifies the symbolic name of the vertex attribute parameter to be queried. Accepted values are `GL_VERTEX_ATTRIB_ARRAY_BUFFER_BINDING`, `GL_VERTEX_ATTRIB_ARRAY_ENABLED`,

GL_VERTEX_ATTRIB_ARRAY_SIZE, GL_VERTEX_ATTRIB_ARRAY_STRIDE, GL_VERTEX_ATTRIB_ARRAY_TYPE, GL_VERTEX_ATTRIB_ARRAY_NORMALIZED, or GL_CURRENT_VERTEX_ATTRIB.

params Returns the requested data.

glGetVertexAttrib returns in *params* the value of a generic vertex attribute parameter. The generic vertex attribute to be queried is specified by *index*, and the parameter to be queried is specified by *pname*.

The accepted parameter names are as follows:

GL_VERTEX_ATTRIB_ARRAY_BUFFER_BINDING
: *params* returns a single value, the name of the buffer object currently bound to the binding point corresponding to generic vertex attribute array *index*. If no buffer object is bound, 0 is returned. The initial value is 0.

GL_VERTEX_ATTRIB_ARRAY_ENABLED
: *params* returns a single value that is non-zero (true) if the vertex attribute array for *index* is enabled and 0 (false) if it is disabled. The initial value is GL_FALSE.

GL_VERTEX_ATTRIB_ARRAY_SIZE
: *params* returns a single value, the size of the vertex attribute array for *index*. The size is the number of values for each element of the vertex attribute array, and it will be 1, 2, 3, or 4. The initial value is 4.

GL_VERTEX_ATTRIB_ARRAY_STRIDE
: *params* returns a single value, the array stride for (number of bytes between successive elements in) the vertex attribute array for *index*. A value of 0 indicates that the array elements are stored sequentially in memory. The initial value is 0.

GL_VERTEX_ATTRIB_ARRAY_TYPE
: *params* returns a single value, a symbolic constant indicating the array type for the vertex attribute array for *index*. Possible values are GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, and GL_DOUBLE. The initial value is GL_FLOAT.

GL_VERTEX_ATTRIB_ARRAY_NORMALIZED
: *params* returns a single value that is non-zero (true) if fixed-point data types for the vertex attribute array indicated by *index* are normalized when they are converted to floating point, and 0 (false) otherwise. The initial value is GL_FALSE.

GL_CURRENT_VERTEX_ATTRIB
: *params* returns four values that represent the current value for the generic vertex attribute specified by index. Generic vertex attribute 0 is unique in that it has no current state, so an error will be generated if *index* is 0. The initial value for all other generic vertex attributes is (0,0,0,1).

All of the parameters except GL_CURRENT_VERTEX_ATTRIB represent client-side state.

`GL_INVALID_VALUE` is generated if *index* is greater than or equal to `GL_MAX_VERTEX_ATTRIBS`.

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

`GL_INVALID_OPERATION` is generated if *index* is 0 and *pname* is `GL_CURRENT_VERTEX_ATTRIB`.

`void glGetBooleanv` *pname params* [Function]
`void glGetDoublev` *pname params* [Function]
`void glGetFloatv` *pname params* [Function]
`void glGetIntegerv` *pname params* [Function]

Return the value or values of a selected parameter.

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.

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

Type conversion is performed if *params* has a different type than the state variable value being requested. If `glGetBooleanv` is called, a floating-point (or integer) value is converted to `GL_FALSE` if and only if it is 0.0 (or 0). Otherwise, it is converted to `GL_TRUE`. If `glGetIntegerv` is called, boolean values are returned as `GL_TRUE` or `GL_FALSE`, and most floating-point values are rounded to the nearest integer value. Floating-point colors and normals, however, are returned with a linear mapping that maps 1.0 to the most positive representable integer value and -1.0 to the most negative representable integer value. If `glGetFloatv` or `glGetDoublev` is called, boolean values are returned as `GL_TRUE` or `GL_FALSE`, and integer values are converted to floating-point values.

The following symbolic constants are accepted by *pname*:

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

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

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

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

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

GL_ACTIVE_TEXTURE
: *params* returns a single value indicating the active multitexture unit. The initial value is GL_TEXTURE0. See glActiveTexture.

GL_ALIASED_POINT_SIZE_RANGE
: *params* returns two values, the smallest and largest supported sizes for aliased points.

GL_ALIASED_LINE_WIDTH_RANGE
: *params* returns two values, the smallest and largest supported widths for aliased lines.

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

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

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

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

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

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

GL_ARRAY_BUFFER_BINDING
: *params* returns a single value, the name of the buffer object currently bound to the target GL_ARRAY_BUFFER. If no buffer object is bound to this target, 0 is returned. The initial value is 0. See glBindBuffer.

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

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

GL_AUX_BUFFERS
: *params* returns one value, the number of auxiliary color buffers available.

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

GL_BLEND_COLOR
: *params* returns four values, the red, green, blue, and alpha values which are the components of the blend color. See glBlendColor.

GL_BLEND_DST_ALPHA
: *params* returns one value, the symbolic constant identifying the alpha destination blend function. The initial value is GL_ZERO. See glBlendFunc and glBlendFuncSeparate.

GL_BLEND_DST_RGB
: *params* returns one value, the symbolic constant identifying the RGB destination blend function. The initial value is GL_ZERO. See glBlendFunc and glBlendFuncSeparate.

GL_BLEND_EQUATION_RGB
: *params* returns one value, a symbolic constant indicating whether the RGB blend equation is GL_FUNC_ADD, GL_FUNC_SUBTRACT, GL_FUNC_REVERSE_SUBTRACT, GL_MIN or GL_MAX. See glBlendEquationSeparate.

GL_BLEND_EQUATION_ALPHA
: *params* returns one value, a symbolic constant indicating whether the Alpha blend equation is GL_FUNC_ADD, GL_FUNC_SUBTRACT, GL_FUNC_REVERSE_SUBTRACT, GL_MIN or GL_MAX. See glBlendEquationSeparate.

GL_BLEND_SRC_ALPHA
: *params* returns one value, the symbolic constant identifying the alpha source blend function. The initial value is GL_ONE. See glBlendFunc and glBlendFuncSeparate.

GL_BLEND_SRC_RGB
: *params* returns one value, the symbolic constant identifying the RGB source blend function. The initial value is GL_ONE. See glBlendFunc and glBlendFuncSeparate.

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

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

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

GL_CLIENT_ACTIVE_TEXTURE
: *params* returns a single integer value indicating the current client active multitexture unit. The initial value is GL_TEXTURE0. See glClientActiveTexture.

GL_CLIENT_ATTRIB_STACK_DEPTH
: *params* returns one value indicating the depth of the attribute stack. The initial value is 0. See glPushClientAttrib.

GL_CLIP_PLANE*i*
: *params* returns a single boolean value indicating whether the specified clipping plane is enabled. The initial value is GL_FALSE. See glClipPlane.

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

GL_COLOR_ARRAY_BUFFER_BINDING
: *params* returns a single value, the name of the buffer object associated with the color array. This buffer object would have been bound to the target GL_ARRAY_BUFFER at the time of the most recent call to glColorPointer. If no buffer object was bound to this target, 0 is returned. The initial value is 0. See glBindBuffer.

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

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

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

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

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

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

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

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

GL_COLOR_MATRIX
: *params* returns sixteen values: the color matrix on the top of the color matrix stack. Initially this matrix is the identity matrix. See glPushMatrix.

GL_COLOR_MATRIX_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_COLOR_SUM
: *params* returns a single boolean value indicating whether primary and secondary color sum is enabled. See glSecondaryColor.

GL_COLOR_TABLE
: *params* returns a single boolean value indicating whether the color table lookup is enabled. See glColorTable.

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

GL_COMPRESSED_TEXTURE_FORMATS
: *params* returns a list of symbolic constants of length GL_NUM_COMPRESSED_TEXTURE_FORMATS indicating which compressed texture formats are available. See glCompressedTexImage2D.

GL_CONVOLUTION_1D
: *params* returns a single boolean value indicating whether 1D convolution is enabled. The initial value is GL_FALSE. See glConvolutionFilter1D.

GL_CONVOLUTION_2D
: *params* returns a single boolean value indicating whether 2D convolution is enabled. The initial value is GL_FALSE. See glConvolutionFilter2D.

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

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

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

`GL_CURRENT_FOG_COORD`
: *params* returns one value, the current fog coordinate. The initial value is 0. See `glFogCoord`.

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

`GL_CURRENT_NORMAL`
: *params* returns three values: the *x*, *y*, and *z* values of the current normal. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. The initial value is (0, 0, 1). See `glNormal`.

`GL_CURRENT_PROGRAM`
: *params* returns one value, the name of the program object that is currently active, or 0 if no program object is active. See `glUseProgram`.

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

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

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

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

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

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

`GL_CURRENT_RASTER_TEXTURE_COORDS`
params returns four values: the s, t, r, and q texture coordinates of the current raster position. The initial value is (0, 0, 0, 1). See `glRasterPos` and `glMultiTexCoord`.

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

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

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

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

`GL_DEPTH_CLEAR_VALUE`
params returns one value, the value that is used to clear the depth buffer. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. The initial value is 1. See `glClearDepth`.

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

`GL_DEPTH_RANGE`
params returns two values: the near and far mapping limits for the depth buffer. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. The initial value is (0, 1). See `glDepthRange`.

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

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

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

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

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

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

GL_DRAW_BUFFER*i*
: *params* returns one value, a symbolic constant indicating which buffers are being drawn to by the corresponding output color. See glDrawBuffers. The initial value of GL_DRAW_BUFFER0 is GL_BACK if there are back buffers, otherwise it is GL_FRONT. The initial values of draw buffers for all other output colors is GL_NONE.

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

GL_EDGE_FLAG_ARRAY
: *params* returns a single boolean value indicating whether the edge flag array is enabled. The initial value is GL_FALSE. See glEdgeFlagPointer.

GL_EDGE_FLAG_ARRAY_BUFFER_BINDING
: *params* returns a single value, the name of the buffer object associated with the edge flag array. This buffer object would have been bound to the target GL_ARRAY_BUFFER at the time of the most recent call to glEdgeFlagPointer. If no buffer object was bound to this target, 0 is returned. The initial value is 0. See glBindBuffer.

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

GL_ELEMENT_ARRAY_BUFFER_BINDING
: *params* returns a single value, the name of the buffer object currently bound to the target GL_ELEMENT_ARRAY_BUFFER. If no buffer object is bound to this target, 0 is returned. The initial value is 0. See glBindBuffer.

GL_FEEDBACK_BUFFER_SIZE
: *params* returns one value, the size of the feedback buffer. See glFeedbackBuffer.

GL_FEEDBACK_BUFFER_TYPE
: *params* returns one value, the type of the feedback buffer. See glFeedbackBuffer.

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

GL_FOG_COORD_ARRAY
: *params* returns a single boolean value indicating whether the fog coordinate array is enabled. The initial value is GL_FALSE. See glFogCoordPointer.

GL_FOG_COORD_ARRAY_BUFFER_BINDING
: *params* returns a single value, the name of the buffer object associated with the fog coordinate array. This buffer object would have been bound to the target GL_ARRAY_BUFFER at the time of the most recent call to glFogCoordPointer. If no buffer object was bound to this target, 0 is returned. The initial value is 0. See glBindBuffer.

GL_FOG_COORD_ARRAY_STRIDE
: *params* returns one value, the byte offset between consecutive fog coordinates in the fog coordinate array. The initial value is 0. See glFogCoordPointer.

GL_FOG_COORD_ARRAY_TYPE
: *params* returns one value, the type of the fog coordinate array. The initial value is GL_FLOAT. See glFogCoordPointer.

GL_FOG_COORD_SRC
: *params* returns one value, a symbolic constant indicating the source of the fog coordinate. The initial value is GL_FRAGMENT_DEPTH. See glFog.

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

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

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

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

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

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

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

`GL_FRAGMENT_SHADER_DERIVATIVE_HINT`
: *params* returns one value, a symbolic constant indicating the mode of the derivative accuracy hint for fragment shaders. The initial value is `GL_DONT_CARE`. See `glHint`.

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

`GL_GENERATE_MIPMAP_HINT`
: *params* returns one value, a symbolic constant indicating the mode of the mipmap generation filtering hint. The initial value is `GL_DONT_CARE`. See `glHint`.

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

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

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

`GL_HISTOGRAM`
: *params* returns a single boolean value indicating whether histogram is enabled. The initial value is `GL_FALSE`. See `glHistogram`.

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

`GL_INDEX_ARRAY_BUFFER_BINDING`
: *params* returns a single value, the name of the buffer object associated with the color index array. This buffer object would have been bound to the target `GL_ARRAY_BUFFER` at the time of the most recent call to `glIndexPointer`. If no buffer object was bound to this target, 0 is returned. The initial value is 0. See `glBindBuffer`.

`GL_INDEX_ARRAY_STRIDE`
: *params* returns one value, the byte offset between consecutive color indexes in the color index array. The initial value is 0. See `glIndexPointer`.

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

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

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

`GL_INDEX_LOGIC_OP`
: *params* returns a single boolean value indicating whether a fragment's index values are merged into the framebuffer using a logical operation. The initial value is `GL_FALSE`. See `glLogicOp`.

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

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

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

`GL_INDEX_WRITEMASK`
: *params* returns one value, a mask indicating which bitplanes of each color index buffer can be written. The initial value is all 1's. See `glIndexMask`.

`GL_LIGHT`*i*
: *params* returns a single boolean value indicating whether the specified light is enabled. The initial value is `GL_FALSE`. See `glLight` and `glLightModel`.

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

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

`GL_LIGHT_MODEL_COLOR_CONTROL`
: *params* returns single enumerated value indicating whether specular reflection calculations are separated from normal lighting computations. The initial value is `GL_SINGLE_COLOR`.

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

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

`GL_LINE_SMOOTH`
: *params* returns a single boolean value indicating whether antialiasing of lines is enabled. The initial value is `GL_FALSE`. See `glLineWidth`.

`GL_LINE_SMOOTH_HINT`
: *params* returns one value, a symbolic constant indicating the mode of the line antialiasing hint. The initial value is `GL_DONT_CARE`. See `glHint`.

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

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

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

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

`GL_LINE_WIDTH_GRANULARITY`
: *params* returns one value, the width difference between adjacent supported widths for antialiased lines. See `glLineWidth`.

`GL_LINE_WIDTH_RANGE`
: *params* returns two values: the smallest and largest supported widths for antialiased lines. See `glLineWidth`.

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

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

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

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

`GL_MAP1_COLOR_4`
: *params* returns a single boolean value indicating whether 1D evaluation generates colors. The initial value is `GL_FALSE`. See `glMap1`.

`GL_MAP1_GRID_DOMAIN`
: *params* returns two values: the endpoints of the 1D map's grid domain. The initial value is (0, 1). See `glMapGrid`.

`GL_MAP1_GRID_SEGMENTS`
: *params* returns one value, the number of partitions in the 1D map's grid domain. The initial value is 1. See `glMapGrid`.

`GL_MAP1_INDEX`
: *params* returns a single boolean value indicating whether 1D evaluation generates color indices. The initial value is `GL_FALSE`. See `glMap1`.

`GL_MAP1_NORMAL`
: *params* returns a single boolean value indicating whether 1D evaluation generates normals. The initial value is `GL_FALSE`. See `glMap1`.

`GL_MAP1_TEXTURE_COORD_1`
: *params* returns a single boolean value indicating whether 1D evaluation generates 1D texture coordinates. The initial value is `GL_FALSE`. See `glMap1`.

`GL_MAP1_TEXTURE_COORD_2`
: *params* returns a single boolean value indicating whether 1D evaluation generates 2D texture coordinates. The initial value is `GL_FALSE`. See `glMap1`.

`GL_MAP1_TEXTURE_COORD_3`
: *params* returns a single boolean value indicating whether 1D evaluation generates 3D texture coordinates. The initial value is `GL_FALSE`. See `glMap1`.

`GL_MAP1_TEXTURE_COORD_4`
: *params* returns a single boolean value indicating whether 1D evaluation generates 4D texture coordinates. The initial value is `GL_FALSE`. See `glMap1`.

`GL_MAP1_VERTEX_3`
: *params* returns a single boolean value indicating whether 1D evaluation generates 3D vertex coordinates. The initial value is `GL_FALSE`. See `glMap1`.

`GL_MAP1_VERTEX_4`
: *params* returns a single boolean value indicating whether 1D evaluation generates 4D vertex coordinates. The initial value is `GL_FALSE`. See `glMap1`.

`GL_MAP2_COLOR_4`
: *params* returns a single boolean value indicating whether 2D evaluation generates colors. The initial value is `GL_FALSE`. See `glMap2`.

`GL_MAP2_GRID_DOMAIN`
: *params* returns four values: the endpoints of the 2D map's *i* and *j* grid domains. The initial value is (0,1; 0,1). See `glMapGrid`.

`GL_MAP2_GRID_SEGMENTS`
: *params* returns two values: the number of partitions in the 2D map's *i* and *j* grid domains. The initial value is (1,1). See `glMapGrid`.

`GL_MAP2_INDEX`
: *params* returns a single boolean value indicating whether 2D evaluation generates color indices. The initial value is `GL_FALSE`. See `glMap2`.

`GL_MAP2_NORMAL`
: *params* returns a single boolean value indicating whether 2D evaluation generates normals. The initial value is `GL_FALSE`. See `glMap2`.

`GL_MAP2_TEXTURE_COORD_1`
: *params* returns a single boolean value indicating whether 2D evaluation generates 1D texture coordinates. The initial value is `GL_FALSE`. See `glMap2`.

`GL_MAP2_TEXTURE_COORD_2`
: *params* returns a single boolean value indicating whether 2D evaluation generates 2D texture coordinates. The initial value is `GL_FALSE`. See `glMap2`.

`GL_MAP2_TEXTURE_COORD_3`
: *params* returns a single boolean value indicating whether 2D evaluation generates 3D texture coordinates. The initial value is `GL_FALSE`. See `glMap2`.

`GL_MAP2_TEXTURE_COORD_4`
: *params* returns a single boolean value indicating whether 2D evaluation generates 4D texture coordinates. The initial value is `GL_FALSE`. See `glMap2`.

`GL_MAP2_VERTEX_3`
: *params* returns a single boolean value indicating whether 2D evaluation generates 3D vertex coordinates. The initial value is `GL_FALSE`. See `glMap2`.

`GL_MAP2_VERTEX_4`
: *params* returns a single boolean value indicating whether 2D evaluation generates 4D vertex coordinates. The initial value is `GL_FALSE`. See `glMap2`.

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

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

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

`GL_MAX_3D_TEXTURE_SIZE`
: *params* returns one value, a rough estimate of the largest 3D texture that the GL can handle. The value must be at least 16. If the GL version is 1.2 or greater, use `GL_PROXY_TEXTURE_3D` to determine if a texture is too large. See `glTexImage3D`.

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

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

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

`GL_MAX_COLOR_MATRIX_STACK_DEPTH`
: *params* returns one value, the maximum supported depth of the color matrix stack. The value must be at least 2. See `glPushMatrix`.

`GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS`
: *params* returns one value, the maximum supported texture image units that can be used to access texture maps from the vertex shader and the fragment processor combined. If both the vertex shader and the fragment processing stage access the same texture image unit, then that counts as

using two texture image units against this limit. The value must be at least 2. See `glActiveTexture`.

`GL_MAX_CUBE_MAP_TEXTURE_SIZE`
params returns one value. The value gives a rough estimate of the largest cube-map texture that the GL can handle. The value must be at least 16. If the GL version is 1.3 or greater, use `GL_PROXY_TEXTURE_CUBE_MAP` to determine if a texture is too large. See `glTexImage2D`.

`GL_MAX_DRAW_BUFFERS`
params returns one value, the maximum number of simultaneous output colors allowed from a fragment shader using the `gl_FragData` built-in array. The value must be at least 1. See `glDrawBuffers`.

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

`GL_MAX_ELEMENTS_VERTICES`
params returns one value, the recommended maximum number of vertex array vertices. See `glDrawRangeElements`.

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

`GL_MAX_FRAGMENT_UNIFORM_COMPONENTS`
params returns one value, the maximum number of individual floating-point, integer, or boolean values that can be held in uniform variable storage for a fragment shader. The value must be at least 64. See `glUniform`.

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

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

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

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

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

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

`GL_MAX_TEXTURE_COORDS`
: *params* returns one value, the maximum number of texture coordinate sets available to vertex and fragment shaders. The value must be at least 2. See `glActiveTexture` and `glClientActiveTexture`.

`GL_MAX_TEXTURE_IMAGE_UNITS`
: *params* returns one value, the maximum supported texture image units that can be used to access texture maps from the fragment shader. The value must be at least 2. See `glActiveTexture`.

`GL_MAX_TEXTURE_LOD_BIAS`
: *params* returns one value, the maximum, absolute value of the texture level-of-detail bias. The value must be at least 4.

`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. If the GL version is 1.1 or greater, use `GL_PROXY_TEXTURE_1D` or `GL_PROXY_TEXTURE_2D` to determine if a texture is too large. See `glTexImage1D` and `glTexImage2D`.

`GL_MAX_TEXTURE_STACK_DEPTH`
: *params* returns one value, the maximum supported depth of the texture matrix stack. The value must be at least 2. See `glPushMatrix`.

`GL_MAX_TEXTURE_UNITS`
: *params* returns a single value indicating the number of conventional texture units supported. Each conventional texture unit includes both a texture coordinate set and a texture image unit. Conventional texture units may be used for fixed-function (non-shader) rendering. The value must be at least 2. Additional texture coordinate sets and texture image units may be accessed from vertex and fragment shaders. See `glActiveTexture` and `glClientActiveTexture`.

`GL_MAX_VARYING_FLOATS`
: *params* returns one value, the maximum number of interpolators available for processing varying variables used by vertex and fragment shaders. This value represents the number of individual floating-point values that can be interpolated; varying variables declared as vectors, matrices, and arrays will all consume multiple interpolators. The value must be at least 32.

`GL_MAX_VERTEX_ATTRIBS`
: *params* returns one value, the maximum number of 4-component generic vertex attributes accessible to a vertex shader. The value must be at least 16. See `glVertexAttrib`.

`GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS`
: *params* returns one value, the maximum supported texture image units that can be used to access texture maps from the vertex shader. The value may be 0. See `glActiveTexture`.

GL_MAX_VERTEX_UNIFORM_COMPONENTS
: *params* returns one value, the maximum number of individual floating-point, integer, or boolean values that can be held in uniform variable storage for a vertex shader. The value must be at least 512. See glUniform.

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_MINMAX
: *params* returns a single boolean value indicating whether pixel minmax values are computed. The initial value is GL_FALSE. See glMinmax.

GL_MODELVIEW_MATRIX
: *params* returns sixteen values: the modelview matrix on the top of the modelview matrix stack. Initially this matrix is the identity matrix. See glPushMatrix.

GL_MODELVIEW_STACK_DEPTH
: *params* returns one value, the number of matrices on the modelview matrix stack. The initial value is 1. See glPushMatrix.

GL_NAME_STACK_DEPTH
: *params* returns one value, the number of names on the selection name stack. The initial value is 0. See glPushName.

GL_NORMAL_ARRAY
: *params* returns a single boolean value, indicating whether the normal array is enabled. The initial value is GL_FALSE. See glNormalPointer.

GL_NORMAL_ARRAY_BUFFER_BINDING
: *params* returns a single value, the name of the buffer object associated with the normal array. This buffer object would have been bound to the target GL_ARRAY_BUFFER at the time of the most recent call to glNormalPointer. If no buffer object was bound to this target, 0 is returned. The initial value is 0. See glBindBuffer.

GL_NORMAL_ARRAY_STRIDE
: *params* returns one value, the byte offset between consecutive normals in the normal array. The initial value is 0. See glNormalPointer.

GL_NORMAL_ARRAY_TYPE
: *params* returns one value, the data type of each coordinate in the normal array. The initial value is GL_FLOAT. See glNormalPointer.

GL_NORMALIZE
: *params* returns a single boolean value indicating whether normals are automatically scaled to unit length after they have been transformed to eye coordinates. The initial value is GL_FALSE. See glNormal.

`GL_NUM_COMPRESSED_TEXTURE_FORMATS`
: *params* returns a single integer value indicating the number of available compressed texture formats. The minimum value is 0. See `glCompressedTexImage2D`.

`GL_PACK_ALIGNMENT`
: *params* returns one value, the byte alignment used for writing pixel data to memory. The initial value is 4. See `glPixelStore`.

`GL_PACK_IMAGE_HEIGHT`
: *params* returns one value, the image height used for writing pixel data to memory. The initial value is 0. See `glPixelStore`.

`GL_PACK_LSB_FIRST`
: *params* returns a single boolean value indicating whether single-bit pixels being written to memory are written first to the least significant bit of each unsigned byte. The initial value is `GL_FALSE`. See `glPixelStore`.

`GL_PACK_ROW_LENGTH`
: *params* returns one value, the row length used for writing pixel data to memory. The initial value is 0. See `glPixelStore`.

`GL_PACK_SKIP_IMAGES`
: *params* returns one value, the number of pixel images skipped before the first pixel is written into memory. The initial value is 0. See `glPixelStore`.

`GL_PACK_SKIP_PIXELS`
: *params* returns one value, the number of pixel locations skipped before the first pixel is written into memory. The initial value is 0. See `glPixelStore`.

`GL_PACK_SKIP_ROWS`
: *params* returns one value, the number of rows of pixel locations skipped before the first pixel is written into memory. The initial value is 0. See `glPixelStore`.

`GL_PACK_SWAP_BYTES`
: *params* returns a single boolean value indicating whether the bytes of two-byte and four-byte pixel indices and components are swapped before being written to memory. The initial value is `GL_FALSE`. See `glPixelStore`.

`GL_PERSPECTIVE_CORRECTION_HINT`
: *params* returns one value, a symbolic constant indicating the mode of the perspective correction hint. The initial value is `GL_DONT_CARE`. See `glHint`.

`GL_PIXEL_MAP_A_TO_A_SIZE`
: *params* returns one value, the size of the alpha-to-alpha pixel translation table. The initial value is 1. See `glPixelMap`.

`GL_PIXEL_MAP_B_TO_B_SIZE`
: *params* returns one value, the size of the blue-to-blue pixel translation table. The initial value is 1. See `glPixelMap`.

`GL_PIXEL_MAP_G_TO_G_SIZE`
: *params* returns one value, the size of the green-to-green pixel translation table. The initial value is 1. See `glPixelMap`.

`GL_PIXEL_MAP_I_TO_A_SIZE`
: *params* returns one value, the size of the index-to-alpha pixel translation table. The initial value is 1. See `glPixelMap`.

`GL_PIXEL_MAP_I_TO_B_SIZE`
: *params* returns one value, the size of the index-to-blue pixel translation table. The initial value is 1. See `glPixelMap`.

`GL_PIXEL_MAP_I_TO_G_SIZE`
: *params* returns one value, the size of the index-to-green pixel translation table. The initial value is 1. See `glPixelMap`.

`GL_PIXEL_MAP_I_TO_I_SIZE`
: *params* returns one value, the size of the index-to-index pixel translation table. The initial value is 1. See `glPixelMap`.

`GL_PIXEL_MAP_I_TO_R_SIZE`
: *params* returns one value, the size of the index-to-red pixel translation table. The initial value is 1. See `glPixelMap`.

`GL_PIXEL_MAP_R_TO_R_SIZE`
: *params* returns one value, the size of the red-to-red pixel translation table. The initial value is 1. See `glPixelMap`.

`GL_PIXEL_MAP_S_TO_S_SIZE`
: *params* returns one value, the size of the stencil-to-stencil pixel translation table. The initial value is 1. See `glPixelMap`.

`GL_PIXEL_PACK_BUFFER_BINDING`
: *params* returns a single value, the name of the buffer object currently bound to the target `GL_PIXEL_PACK_BUFFER`. If no buffer object is bound to this target, 0 is returned. The initial value is 0. See `glBindBuffer`.

`GL_PIXEL_UNPACK_BUFFER_BINDING`
: *params* returns a single value, the name of the buffer object currently bound to the target `GL_PIXEL_UNPACK_BUFFER`. If no buffer object is bound to this target, 0 is returned. The initial value is 0. See `glBindBuffer`.

`GL_POINT_DISTANCE_ATTENUATION`
: *params* returns three values, the coefficients for computing the attenuation value for points. See `glPointParameter`.

`GL_POINT_FADE_THRESHOLD_SIZE`
: *params* returns one value, the point size threshold for determining the point size. See `glPointParameter`.

`GL_POINT_SIZE`
: *params* returns one value, the point size as specified by `glPointSize`. The initial value is 1.

`GL_POINT_SIZE_GRANULARITY`
: *params* returns one value, the size difference between adjacent supported sizes for antialiased points. See `glPointSize`.

`GL_POINT_SIZE_MAX`
: *params* returns one value, the upper bound for the attenuated point sizes. The initial value is 0.0. See `glPointParameter`.

`GL_POINT_SIZE_MIN`
: *params* returns one value, the lower bound for the attenuated point sizes. The initial value is 1.0. See `glPointParameter`.

`GL_POINT_SIZE_RANGE`
: *params* returns two values: the smallest and largest supported sizes for antialiased points. The smallest size must be at most 1, and the largest size must be at least 1. See `glPointSize`.

`GL_POINT_SMOOTH`
: *params* returns a single boolean value indicating whether antialiasing of points is enabled. The initial value is `GL_FALSE`. See `glPointSize`.

`GL_POINT_SMOOTH_HINT`
: *params* returns one value, a symbolic constant indicating the mode of the point antialiasing hint. The initial value is `GL_DONT_CARE`. See `glHint`.

`GL_POINT_SPRITE`
: *params* returns a single boolean value indicating whether point sprite is enabled. The initial value is `GL_FALSE`.

`GL_POLYGON_MODE`
: *params* returns two values: symbolic constants indicating whether front-facing and back-facing polygons are rasterized as points, lines, or filled polygons. The initial value is `GL_FILL`. See `glPolygonMode`.

`GL_POLYGON_OFFSET_FACTOR`
: *params* returns one value, the scaling factor used to determine the variable offset that is added to the depth value of each fragment generated when a polygon is rasterized. The initial value is 0. See `glPolygonOffset`.

`GL_POLYGON_OFFSET_UNITS`
: *params* returns one value. This value is multiplied by an implementation-specific value and then added to the depth value of each fragment generated when a polygon is rasterized. The initial value is 0. See `glPolygonOffset`.

`GL_POLYGON_OFFSET_FILL`
: *params* returns a single boolean value indicating whether polygon offset is enabled for polygons in fill mode. The initial value is `GL_FALSE`. See `glPolygonOffset`.

GL_POLYGON_OFFSET_LINE
: *params* returns a single boolean value indicating whether polygon offset is enabled for polygons in line mode. The initial value is GL_FALSE. See glPolygonOffset.

GL_POLYGON_OFFSET_POINT
: *params* returns a single boolean value indicating whether polygon offset is enabled for polygons in point mode. The initial value is GL_FALSE. See glPolygonOffset.

GL_POLYGON_SMOOTH
: *params* returns a single boolean value indicating whether antialiasing of polygons is enabled. The initial value is GL_FALSE. See glPolygonMode.

GL_POLYGON_SMOOTH_HINT
: *params* returns one value, a symbolic constant indicating the mode of the polygon antialiasing hint. The initial value is GL_DONT_CARE. See glHint.

GL_POLYGON_STIPPLE
: *params* returns a single boolean value indicating whether polygon stippling is enabled. The initial value is GL_FALSE. See glPolygonStipple.

GL_POST_COLOR_MATRIX_COLOR_TABLE
: *params* returns a single boolean value indicating whether post color matrix transformation lookup is enabled. The initial value is GL_FALSE. See glColorTable.

GL_POST_COLOR_MATRIX_RED_BIAS
: *params* returns one value, the red bias factor applied to RGBA fragments after color matrix transformations. The initial value is 0. See glPixelTransfer.

GL_POST_COLOR_MATRIX_GREEN_BIAS
: *params* returns one value, the green bias factor applied to RGBA fragments after color matrix transformations. The initial value is 0. See glPixelTransfer

GL_POST_COLOR_MATRIX_BLUE_BIAS
: *params* returns one value, the blue bias factor applied to RGBA fragments after color matrix transformations. The initial value is 0. See glPixelTransfer.

GL_POST_COLOR_MATRIX_ALPHA_BIAS
: *params* returns one value, the alpha bias factor applied to RGBA fragments after color matrix transformations. The initial value is 0. See glPixelTransfer.

GL_POST_COLOR_MATRIX_RED_SCALE
: *params* returns one value, the red scale factor applied to RGBA fragments after color matrix transformations. The initial value is 1. See glPixelTransfer.

`GL_POST_COLOR_MATRIX_GREEN_SCALE`
: *params* returns one value, the green scale factor applied to RGBA fragments after color matrix transformations. The initial value is 1. See `glPixelTransfer`.

`GL_POST_COLOR_MATRIX_BLUE_SCALE`
: *params* returns one value, the blue scale factor applied to RGBA fragments after color matrix transformations. The initial value is 1. See `glPixelTransfer`.

`GL_POST_COLOR_MATRIX_ALPHA_SCALE`
: *params* returns one value, the alpha scale factor applied to RGBA fragments after color matrix transformations. The initial value is 1. See `glPixelTransfer`.

`GL_POST_CONVOLUTION_COLOR_TABLE`
: *params* returns a single boolean value indicating whether post convolution lookup is enabled. The initial value is `GL_FALSE`. See `glColorTable`.

`GL_POST_CONVOLUTION_RED_BIAS`
: *params* returns one value, the red bias factor applied to RGBA fragments after convolution. The initial value is 0. See `glPixelTransfer`.

`GL_POST_CONVOLUTION_GREEN_BIAS`
: *params* returns one value, the green bias factor applied to RGBA fragments after convolution. The initial value is 0. See `glPixelTransfer`.

`GL_POST_CONVOLUTION_BLUE_BIAS`
: *params* returns one value, the blue bias factor applied to RGBA fragments after convolution. The initial value is 0. See `glPixelTransfer`.

`GL_POST_CONVOLUTION_ALPHA_BIAS`
: *params* returns one value, the alpha bias factor applied to RGBA fragments after convolution. The initial value is 0. See `glPixelTransfer`.

`GL_POST_CONVOLUTION_RED_SCALE`
: *params* returns one value, the red scale factor applied to RGBA fragments after convolution. The initial value is 1. See `glPixelTransfer`.

`GL_POST_CONVOLUTION_GREEN_SCALE`
: *params* returns one value, the green scale factor applied to RGBA fragments after convolution. The initial value is 1. See `glPixelTransfer`.

`GL_POST_CONVOLUTION_BLUE_SCALE`
: *params* returns one value, the blue scale factor applied to RGBA fragments after convolution. The initial value is 1. See `glPixelTransfer`.

`GL_POST_CONVOLUTION_ALPHA_SCALE`
: *params* returns one value, the alpha scale factor applied to RGBA fragments after convolution. The initial value is 1. See `glPixelTransfer`.

`GL_PROJECTION_MATRIX`
: *params* returns sixteen values: the projection matrix on the top of the projection matrix stack. Initially this matrix is the identity matrix. See `glPushMatrix`.

`GL_PROJECTION_STACK_DEPTH`
: *params* returns one value, the number of matrices on the projection matrix stack. The initial value is 1. See `glPushMatrix`.

`GL_READ_BUFFER`
: *params* returns one value, a symbolic constant indicating which color buffer is selected for reading. The initial value is `GL_BACK` if there is a back buffer, otherwise it is `GL_FRONT`. See `glReadPixels` and `glAccum`.

`GL_RED_BIAS`
: *params* returns one value, the red bias factor used during pixel transfers. The initial value is 0.

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

`GL_RED_SCALE`
: *params* returns one value, the red scale factor used during pixel transfers. The initial value is 1. See `glPixelTransfer`.

`GL_RENDER_MODE`
: *params* returns one value, a symbolic constant indicating whether the GL is in render, select, or feedback mode. The initial value is `GL_RENDER`. See `glRenderMode`.

`GL_RESCALE_NORMAL`
: *params* returns single boolean value indicating whether normal rescaling is enabled. See `glEnable`.

`GL_RGBA_MODE`
: *params* returns a single boolean value indicating whether the GL is in RGBA mode (true) or color index mode (false). See `glColor`.

`GL_SAMPLE_BUFFERS`
: *params* returns a single integer value indicating the number of sample buffers associated with the framebuffer. See `glSampleCoverage`.

`GL_SAMPLE_COVERAGE_VALUE`
: *params* returns a single positive floating-point value indicating the current sample coverage value. See `glSampleCoverage`.

`GL_SAMPLE_COVERAGE_INVERT`
: *params* returns a single boolean value indicating if the temporary coverage value should be inverted. See `glSampleCoverage`.

`GL_SAMPLES`
: *params* returns a single integer value indicating the coverage mask size. See `glSampleCoverage`.

`GL_SCISSOR_BOX`
: *params* returns four values: the x and y window coordinates of the scissor box, followed by its width and height. Initially the x and y window coordinates are both 0 and the width and height are set to the size of the window. See `glScissor`.

GL_SCISSOR_TEST
: *params* returns a single boolean value indicating whether scissoring is enabled. The initial value is GL_FALSE. See glScissor.

GL_SECONDARY_COLOR_ARRAY
: *params* returns a single boolean value indicating whether the secondary color array is enabled. The initial value is GL_FALSE. See glSecondaryColorPointer.

GL_SECONDARY_COLOR_ARRAY_BUFFER_BINDING
: *params* returns a single value, the name of the buffer object associated with the secondary color array. This buffer object would have been bound to the target GL_ARRAY_BUFFER at the time of the most recent call to glSecondaryColorPointer. If no buffer object was bound to this target, 0 is returned. The initial value is 0. See glBindBuffer.

GL_SECONDARY_COLOR_ARRAY_SIZE
: *params* returns one value, the number of components per color in the secondary color array. The initial value is 3. See glSecondaryColorPointer.

GL_SECONDARY_COLOR_ARRAY_STRIDE
: *params* returns one value, the byte offset between consecutive colors in the secondary color array. The initial value is 0. See glSecondaryColorPointer.

GL_SECONDARY_COLOR_ARRAY_TYPE
: *params* returns one value, the data type of each component in the secondary color array. The initial value is GL_FLOAT. See glSecondaryColorPointer.

GL_SELECTION_BUFFER_SIZE
: *params* return one value, the size of the selection buffer. See glSelectBuffer.

GL_SEPARABLE_2D
: *params* returns a single boolean value indicating whether 2D separable convolution is enabled. The initial value is GL_FALSE. See glSeparableFilter2D.

GL_SHADE_MODEL
: *params* returns one value, a symbolic constant indicating whether the shading mode is flat or smooth. The initial value is GL_SMOOTH. See glShadeModel.

GL_SMOOTH_LINE_WIDTH_RANGE
: *params* returns two values, the smallest and largest supported widths for antialiased lines. See glLineWidth.

GL_SMOOTH_LINE_WIDTH_GRANULARITY
: *params* returns one value, the granularity of widths for antialiased lines. See glLineWidth.

`GL_SMOOTH_POINT_SIZE_RANGE`
params returns two values, the smallest and largest supported widths for antialiased points. See `glPointSize`.

`GL_SMOOTH_POINT_SIZE_GRANULARITY`
params returns one value, the granularity of sizes for antialiased points. See `glPointSize`.

`GL_STENCIL_BACK_FAIL`
params returns one value, a symbolic constant indicating what action is taken for back-facing polygons when the stencil test fails. The initial value is `GL_KEEP`. See `glStencilOpSeparate`.

`GL_STENCIL_BACK_FUNC`
params returns one value, a symbolic constant indicating what function is used for back-facing polygons to compare the stencil reference value with the stencil buffer value. The initial value is `GL_ALWAYS`. See `glStencilFuncSeparate`.

`GL_STENCIL_BACK_PASS_DEPTH_FAIL`
params returns one value, a symbolic constant indicating what action is taken for back-facing polygons when the stencil test passes, but the depth test fails. The initial value is `GL_KEEP`. See `glStencilOpSeparate`.

`GL_STENCIL_BACK_PASS_DEPTH_PASS`
params returns one value, a symbolic constant indicating what action is taken for back-facing polygons when the stencil test passes and the depth test passes. The initial value is `GL_KEEP`. See `glStencilOpSeparate`.

`GL_STENCIL_BACK_REF`
params returns one value, the reference value that is compared with the contents of the stencil buffer for back-facing polygons. The initial value is 0. See `glStencilFuncSeparate`.

`GL_STENCIL_BACK_VALUE_MASK`
params returns one value, the mask that is used for back-facing polygons to mask both the stencil reference value and the stencil buffer value before they are compared. The initial value is all 1's. See `glStencilFuncSeparate`.

`GL_STENCIL_BACK_WRITEMASK`
params returns one value, the mask that controls writing of the stencil bitplanes for back-facing polygons. The initial value is all 1's. See `glStencilMaskSeparate`.

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

`GL_STENCIL_CLEAR_VALUE`
params returns one value, the index to which the stencil bitplanes are cleared. The initial value is 0. See `glClearStencil`.

GL_STENCIL_FAIL
: *params* returns one value, a symbolic constant indicating what action is taken when the stencil test fails. The initial value is GL_KEEP. See glStencilOp. If the GL version is 2.0 or greater, this stencil state only affects non-polygons and front-facing polygons. Back-facing polygons use separate stencil state. See glStencilOpSeparate.

GL_STENCIL_FUNC
: *params* returns one value, a symbolic constant indicating what function is used to compare the stencil reference value with the stencil buffer value. The initial value is GL_ALWAYS. See glStencilFunc. If the GL version is 2.0 or greater, this stencil state only affects non-polygons and front-facing polygons. Back-facing polygons use separate stencil state. See glStencilFuncSeparate.

GL_STENCIL_PASS_DEPTH_FAIL
: *params* returns one value, a symbolic constant indicating what action is taken when the stencil test passes, but the depth test fails. The initial value is GL_KEEP. See glStencilOp. If the GL version is 2.0 or greater, this stencil state only affects non-polygons and front-facing polygons. Back-facing polygons use separate stencil state. See glStencilOpSeparate.

GL_STENCIL_PASS_DEPTH_PASS
: *params* returns one value, a symbolic constant indicating what action is taken when the stencil test passes and the depth test passes. The initial value is GL_KEEP. See glStencilOp. If the GL version is 2.0 or greater, this stencil state only affects non-polygons and front-facing polygons. Back-facing polygons use separate stencil state. See glStencilOpSeparate.

GL_STENCIL_REF
: *params* returns one value, the reference value that is compared with the contents of the stencil buffer. The initial value is 0. See glStencilFunc. If the GL version is 2.0 or greater, this stencil state only affects non-polygons and front-facing polygons. Back-facing polygons use separate stencil state. See glStencilFuncSeparate.

GL_STENCIL_TEST
: *params* returns a single boolean value indicating whether stencil testing of fragments is enabled. The initial value is GL_FALSE. See glStencilFunc and glStencilOp.

GL_STENCIL_VALUE_MASK
: *params* returns one value, the mask that is used to mask both the stencil reference value and the stencil buffer value before they are compared. The initial value is all 1's. See glStencilFunc. If the GL version is 2.0 or greater, this stencil state only affects non-polygons and front-facing polygons. Back-facing polygons use separate stencil state. See glStencilFuncSeparate.

GL_STENCIL_WRITEMASK
: *params* returns one value, the mask that controls writing of the stencil bitplanes. The initial value is all 1's. See glStencilMask. If the GL version is 2.0 or greater, this stencil state only affects non-polygons and front-facing polygons. Back-facing polygons use separate stencil state. See glStencilMaskSeparate.

GL_STEREO
: *params* returns a single boolean value indicating whether stereo buffers (left and right) are supported.

GL_SUBPIXEL_BITS
: *params* returns one value, an estimate of the number of bits of subpixel resolution that are used to position rasterized geometry in window coordinates. The value must be at least 4.

GL_TEXTURE_1D
: *params* returns a single boolean value indicating whether 1D texture mapping is enabled. The initial value is GL_FALSE. See glTexImage1D.

GL_TEXTURE_BINDING_1D
: *params* returns a single value, the name of the texture currently bound to the target GL_TEXTURE_1D. The initial value is 0. See glBindTexture.

GL_TEXTURE_2D
: *params* returns a single boolean value indicating whether 2D texture mapping is enabled. The initial value is GL_FALSE. See glTexImage2D.

GL_TEXTURE_BINDING_2D
: *params* returns a single value, the name of the texture currently bound to the target GL_TEXTURE_2D. The initial value is 0. See glBindTexture.

GL_TEXTURE_3D
: *params* returns a single boolean value indicating whether 3D texture mapping is enabled. The initial value is GL_FALSE. See glTexImage3D.

GL_TEXTURE_BINDING_3D
: *params* returns a single value, the name of the texture currently bound to the target GL_TEXTURE_3D. The initial value is 0. See glBindTexture.

GL_TEXTURE_BINDING_CUBE_MAP
: *params* returns a single value, the name of the texture currently bound to the target GL_TEXTURE_CUBE_MAP. The initial value is 0. See glBindTexture.

GL_TEXTURE_COMPRESSION_HINT
: *params* returns a single value indicating the mode of the texture compression hint. The initial value is GL_DONT_CARE.

GL_TEXTURE_COORD_ARRAY
: *params* returns a single boolean value indicating whether the texture coordinate array is enabled. The initial value is GL_FALSE. See glTexCoordPointer.

GL_TEXTURE_COORD_ARRAY_BUFFER_BINDING
: *params* returns a single value, the name of the buffer object associated with the texture coordinate array. This buffer object would have been bound to the target GL_ARRAY_BUFFER at the time of the most recent call to glTexCoordPointer. If no buffer object was bound to this target, 0 is returned. The initial value is 0. See glBindBuffer.

GL_TEXTURE_COORD_ARRAY_SIZE
: *params* returns one value, the number of coordinates per element in the texture coordinate array. The initial value is 4. See glTexCoordPointer.

GL_TEXTURE_COORD_ARRAY_STRIDE
: *params* returns one value, the byte offset between consecutive elements in the texture coordinate array. The initial value is 0. See glTexCoordPointer.

GL_TEXTURE_COORD_ARRAY_TYPE
: *params* returns one value, the data type of the coordinates in the texture coordinate array. The initial value is GL_FLOAT. See glTexCoordPointer.

GL_TEXTURE_CUBE_MAP
: *params* returns a single boolean value indicating whether cube-mapped texture mapping is enabled. The initial value is GL_FALSE. See glTexImage2D.

GL_TEXTURE_GEN_Q
: *params* returns a single boolean value indicating whether automatic generation of the q texture coordinate is enabled. The initial value is GL_FALSE. See glTexGen.

GL_TEXTURE_GEN_R
: *params* returns a single boolean value indicating whether automatic generation of the r texture coordinate is enabled. The initial value is GL_FALSE. See glTexGen.

GL_TEXTURE_GEN_S
: *params* returns a single boolean value indicating whether automatic generation of the S texture coordinate is enabled. The initial value is GL_FALSE. See glTexGen.

GL_TEXTURE_GEN_T
: *params* returns a single boolean value indicating whether automatic generation of the T texture coordinate is enabled. The initial value is GL_FALSE. See glTexGen.

GL_TEXTURE_MATRIX
: *params* returns sixteen values: the texture matrix on the top of the texture matrix stack. Initially this matrix is the identity matrix. See glPushMatrix.

GL_TEXTURE_STACK_DEPTH
: *params* returns one value, the number of matrices on the texture matrix stack. The initial value is 1. See glPushMatrix.

`GL_TRANSPOSE_COLOR_MATRIX`
params returns 16 values, the elements of the color matrix in row-major order. See `glLoadTransposeMatrix`.

`GL_TRANSPOSE_MODELVIEW_MATRIX`
params returns 16 values, the elements of the modelview matrix in row-major order. See `glLoadTransposeMatrix`.

`GL_TRANSPOSE_PROJECTION_MATRIX`
params returns 16 values, the elements of the projection matrix in row-major order. See `glLoadTransposeMatrix`.

`GL_TRANSPOSE_TEXTURE_MATRIX`
params returns 16 values, the elements of the texture matrix in row-major order. See `glLoadTransposeMatrix`.

`GL_UNPACK_ALIGNMENT`
params returns one value, the byte alignment used for reading pixel data from memory. The initial value is 4. See `glPixelStore`.

`GL_UNPACK_IMAGE_HEIGHT`
params returns one value, the image height used for reading pixel data from memory. The initial is 0. See `glPixelStore`.

`GL_UNPACK_LSB_FIRST`
params returns a single boolean value indicating whether single-bit pixels being read from memory are read first from the least significant bit of each unsigned byte. The initial value is `GL_FALSE`. See `glPixelStore`.

`GL_UNPACK_ROW_LENGTH`
params returns one value, the row length used for reading pixel data from memory. The initial value is 0. See `glPixelStore`.

`GL_UNPACK_SKIP_IMAGES`
params returns one value, the number of pixel images skipped before the first pixel is read from memory. The initial value is 0. See `glPixelStore`.

`GL_UNPACK_SKIP_PIXELS`
params returns one value, the number of pixel locations skipped before the first pixel is read from memory. The initial value is 0. See `glPixelStore`.

`GL_UNPACK_SKIP_ROWS`
params returns one value, the number of rows of pixel locations skipped before the first pixel is read from memory. The initial value is 0. See `glPixelStore`.

`GL_UNPACK_SWAP_BYTES`
params returns a single boolean value indicating whether the bytes of two-byte and four-byte pixel indices and components are swapped after being read from memory. The initial value is `GL_FALSE`. See `glPixelStore`.

`GL_VERTEX_ARRAY`
params returns a single boolean value indicating whether the vertex array is enabled. The initial value is `GL_FALSE`. See `glVertexPointer`.

`GL_VERTEX_ARRAY_BUFFER_BINDING`
: *params* returns a single value, the name of the buffer object associated with the vertex array. This buffer object would have been bound to the target `GL_ARRAY_BUFFER` at the time of the most recent call to `glVertexPointer`. If no buffer object was bound to this target, 0 is returned. The initial value is 0. See `glBindBuffer`.

`GL_VERTEX_ARRAY_SIZE`
: *params* returns one value, the number of coordinates per vertex in the vertex array. The initial value is 4. See `glVertexPointer`.

`GL_VERTEX_ARRAY_STRIDE`
: *params* returns one value, the byte offset between consecutive vertices in the vertex array. The initial value is 0. See `glVertexPointer`.

`GL_VERTEX_ARRAY_TYPE`
: *params* returns one value, the data type of each coordinate in the vertex array. The initial value is `GL_FLOAT`. See `glVertexPointer`.

`GL_VERTEX_PROGRAM_POINT_SIZE`
: *params* returns a single boolean value indicating whether vertex program point size mode is enabled. If enabled, and a vertex shader is active, then the point size is taken from the shader built-in `gl_PointSize`. If disabled, and a vertex shader is active, then the point size is taken from the point state as specified by `glPointSize`. The initial value is `GL_FALSE`.

`GL_VERTEX_PROGRAM_TWO_SIDE`
: *params* returns a single boolean value indicating whether vertex program two-sided color mode is enabled. If enabled, and a vertex shader is active, then the GL chooses the back color output for back-facing polygons, and the front color output for non-polygons and front-facing polygons. If disabled, and a vertex shader is active, then the front color output is always selected. The initial value is `GL_FALSE`.

`GL_VIEWPORT`
: *params* returns four values: the x and y window coordinates of the viewport, followed by its width and height. Initially the x and y window coordinates are both set to 0, and the width and height are set to the width and height of the window into which the GL will do its rendering. See `glViewport`.

`GL_ZOOM_X`
: *params* returns one value, the x pixel zoom factor. The initial value is 1. See `glPixelZoom`.

`GL_ZOOM_Y`
: *params* returns one value, the y pixel zoom factor. The initial value is 1. See `glPixelZoom`.

Many of the boolean parameters can also be queried more easily using `glIsEnabled`.

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

`GL_INVALID_OPERATION` is generated if `glGet` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glHint` *target mode* [Function]

Specify implementation-specific hints.

target Specifies a symbolic constant indicating the behavior to be controlled. `GL_FOG_HINT`, `GL_GENERATE_MIPMAP_HINT`, `GL_LINE_SMOOTH_HINT`, `GL_PERSPECTIVE_CORRECTION_HINT`, `GL_POINT_SMOOTH_HINT`, `GL_POLYGON_SMOOTH_HINT`, `GL_TEXTURE_COMPRESSION_HINT`, and `GL_FRAGMENT_SHADER_DERIVATIVE_HINT` are accepted.

mode Specifies a symbolic constant indicating the desired behavior. `GL_FASTEST`, `GL_NICEST`, and `GL_DONT_CARE` are accepted.

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_FRAGMENT_SHADER_DERIVATIVE_HINT`
: Indicates the accuracy of the derivative calculation for the GL shading language fragment processing built-in functions: `dFdx`, `dFdy`, and `fwidth`.

`GL_GENERATE_MIPMAP_HINT`
: Indicates the quality of filtering when generating mipmap images.

`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, texture coordinate, and fog 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.

GL_POLYGON_SMOOTH_HINT
: Indicates the sampling quality of antialiased polygons. Hinting GL_NICEST can result in more pixel fragments being generated during rasterization, if a larger filter function is applied.

GL_TEXTURE_COMPRESSION_HINT
: Indicates the quality and performance of the compressing texture images. Hinting GL_FASTEST indicates that texture images should be compressed as quickly as possible, while GL_NICEST indicates that texture images should be compressed with as little image quality loss as possible. GL_NICEST should be selected if the texture is to be retrieved by glGetCompressedTexImage for reuse.

GL_INVALID_ENUM is generated if either *target* or *mode* is not an accepted value.

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

`void glHistogram` *target width internalformat sink* [Function]

Define histogram table.

target The histogram whose parameters are to be set. Must be one of GL_HISTOGRAM or GL_PROXY_HISTOGRAM.

width The number of entries in the histogram table. Must be a power of 2.

internalformat
: The format of entries in the histogram table. Must be one of GL_ALPHA, GL_ALPHA4, GL_ALPHA8, GL_ALPHA12, GL_ALPHA16, GL_LUMINANCE, GL_LUMINANCE4, GL_LUMINANCE8, GL_LUMINANCE12, GL_LUMINANCE16, GL_LUMINANCE_ALPHA, GL_LUMINANCE4_ALPHA4, GL_LUMINANCE6_ALPHA2, GL_LUMINANCE8_ALPHA8, GL_LUMINANCE12_ALPHA4, GL_LUMINANCE12_ALPHA12, GL_LUMINANCE16_ALPHA16, GL_R3_G3_B2, GL_RGB, GL_RGB4, GL_RGB5, GL_RGB8, GL_RGB10, GL_RGB12, GL_RGB16, GL_RGBA, GL_RGBA2, GL_RGBA4, GL_RGB5_A1, GL_RGBA8, GL_RGB10_A2, GL_RGBA12, or GL_RGBA16.

sink If GL_TRUE, pixels will be consumed by the histogramming process and no drawing or texture loading will take place. If GL_FALSE, pixels will proceed to the minmax process after histogramming.

When GL_HISTOGRAM is enabled, RGBA color components are converted to histogram table indices by clamping to the range [0,1], multiplying by the width of the histogram table, and rounding to the nearest integer. The table entries selected by the RGBA indices are then incremented. (If the internal format of the histogram table includes

luminance, then the index derived from the R color component determines the luminance table entry to be incremented.) If a histogram table entry is incremented beyond its maximum value, then its value becomes undefined. (This is not an error.)

Histogramming is performed only for RGBA pixels (though these may be specified originally as color indices and converted to RGBA by index table lookup). Histogramming is enabled with `glEnable` and disabled with `glDisable`.

When *target* is `GL_HISTOGRAM`, `glHistogram` redefines the current histogram table to have *width* entries of the format specified by *internalformat*. The entries are indexed 0 through *width*-1, and all entries are initialized to zero. The values in the previous histogram table, if any, are lost. If *sink* is `GL_TRUE`, then pixels are discarded after histogramming; no further processing of the pixels takes place, and no drawing, texture loading, or pixel readback will result.

When *target* is `GL_PROXY_HISTOGRAM`, `glHistogram` computes all state information as if the histogram table were to be redefined, but does not actually define the new table. If the requested histogram table is too large to be supported, then the state information will be set to zero. This provides a way to determine if a histogram table with the given parameters can be supported.

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

`GL_INVALID_VALUE` is generated if *width* is less than zero or is not a power of 2.

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

`GL_TABLE_TOO_LARGE` is generated if *target* is `GL_HISTOGRAM` and the histogram table specified is too large for the implementation.

`GL_INVALID_OPERATION` is generated if `glHistogram` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glIndexMask` *mask* [Function]

Control the writing of individual bits in the color index buffers.

mask Specifies a bit mask to enable and disable the writing of individual bits in the color index buffers. Initially, the mask is all 1's.

`glIndexMask` controls the writing of individual bits in the color index buffers. The least significant *n* bits of *mask*, where *n* is the number of bits in a color index buffer, specify a mask. Where a 1 (one) appears in the mask, it's possible to write to the corresponding bit in the color index buffer (or buffers). Where a 0 (zero) appears, the corresponding bit is write-protected.

This mask is used only in color index mode, and it affects only the buffers currently selected for writing (see `glDrawBuffer`). Initially, all bits are enabled for writing.

`GL_INVALID_OPERATION` is generated if `glIndexMask` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glIndexPointer` *type stride pointer* [Function]

Define an array of color indexes.

type Specifies the data type of each color index in the array. Symbolic constants `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_INT`, `GL_FLOAT`, and `GL_DOUBLE` are accepted. The initial value is `GL_FLOAT`.

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

pointer Specifies a pointer to the first index in the array. The initial value is 0.

`glIndexPointer` specifies the location and data format of an array of color indexes to use when rendering. *type* specifies the data type of each color index and *stride* specifies the byte stride from one color index to the next, allowing vertices and attributes to be packed into a single array or stored in separate arrays.

If a non-zero named buffer object is bound to the `GL_ARRAY_BUFFER` target (see `glBindBuffer`) while a color index array is specified, *pointer* is treated as a byte offset into the buffer object's data store. Also, the buffer object binding (`GL_ARRAY_BUFFER_BINDING`) is saved as color index vertex array client-side state (`GL_INDEX_ARRAY_BUFFER_BINDING`).

When a color index array is specified, *type*, *stride*, and *pointer* are saved as client-side state, in addition to the current vertex array buffer object binding.

To enable and disable the color index array, call `glEnableClientState` and `glDisableClientState` with the argument `GL_INDEX_ARRAY`. If enabled, the color index array is used when `glDrawArrays`, `glMultiDrawArrays`, `glDrawElements`, `glMultiDrawElements`, `glDrawRangeElements`, or `glArrayElement` is called.

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

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

`void glIndexs` *c* [Function]
`void glIndexi` *c* [Function]
`void glIndexf` *c* [Function]
`void glIndexd` *c* [Function]
`void glIndexub` *c* [Function]
`void glIndexsv` *c* [Function]
`void glIndexiv` *c* [Function]
`void glIndexfv` *c* [Function]
`void glIndexdv` *c* [Function]
`void glIndexubv` *c* [Function]

Set the current color index.

c Specifies the new value for the current color index.

`glIndex` updates the current (single-valued) color index. It takes one argument, the new value for the current color index.

The current index is stored as a floating-point value. Integer values are converted directly to floating-point values, with no special mapping. The initial value is 1.

Index values outside the representable range of the color index buffer are not clamped. However, before an index is dithered (if enabled) and written to the frame buffer, it is converted to fixed-point format. Any bits in the integer portion of the resulting fixed-point value that do not correspond to bits in the frame buffer are masked out.

`void glInitNames` [Function]

Initialize the name stack.

The name stack is used during selection mode to allow sets of rendering commands to be uniquely identified. It consists of an ordered set of unsigned integers. `glInitNames` causes the name stack to be initialized to its default empty state.

The name stack is always empty while the render mode is not `GL_SELECT`. Calls to `glInitNames` while the render mode is not `GL_SELECT` are ignored.

`GL_INVALID_OPERATION` is generated if `glInitNames` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glInterleavedArrays` *format stride pointer* [Function]

Simultaneously specify and enable several interleaved arrays.

format Specifies the type of array to enable. Symbolic constants `GL_V2F`, `GL_V3F`, `GL_C4UB_V2F`, `GL_C4UB_V3F`, `GL_C3F_V3F`, `GL_N3F_V3F`, `GL_C4F_N3F_V3F`, `GL_T2F_V3F`, `GL_T4F_V4F`, `GL_T2F_C4UB_V3F`, `GL_T2F_C3F_V3F`, `GL_T2F_N3F_V3F`, `GL_T2F_C4F_N3F_V3F`, and `GL_T4F_C4F_N3F_V4F` are accepted.

stride Specifies the offset in bytes between each aggregate array element.

`glInterleavedArrays` lets you specify and enable individual color, normal, texture and vertex arrays whose elements are part of a larger aggregate array element. For some implementations, this is more efficient than specifying the arrays separately.

If *stride* is 0, the aggregate elements are stored consecutively. Otherwise, *stride* bytes occur between the beginning of one aggregate array element and the beginning of the next aggregate array element.

format serves as a "key" describing the extraction of individual arrays from the aggregate array. If *format* contains a T, then texture coordinates are extracted from the interleaved array. If C is present, color values are extracted. If N is present, normal coordinates are extracted. Vertex coordinates are always extracted.

The digits 2, 3, and 4 denote how many values are extracted. F indicates that values are extracted as floating-point values. Colors may also be extracted as 4 unsigned bytes if 4UB follows the C. If a color is extracted as 4 unsigned bytes, the vertex array element which follows is located at the first possible floating-point aligned address.

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

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

`GLboolean glIsBuffer` *buffer* [Function]

Determine if a name corresponds to a buffer object.

buffer Specifies a value that may be the name of a buffer object.

`glIsBuffer` returns `GL_TRUE` if *buffer* is currently the name of a buffer object. If *buffer* is zero, or is a non-zero value that is not currently the name of a buffer object, or if an error occurs, `glIsBuffer` returns `GL_FALSE`.

A name returned by `glGenBuffers`, but not yet associated with a buffer object by calling `glBindBuffer`, is not the name of a buffer object.

`GL_INVALID_OPERATION` is generated if `glIsBuffer` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`GLboolean glIsEnabled` *cap* [Function]

Test whether a capability is enabled.

cap Specifies a symbolic constant indicating a GL capability.

`glIsEnabled` returns `GL_TRUE` if *cap* is an enabled capability and returns `GL_FALSE` otherwise. Initially all capabilities except `GL_DITHER` are disabled; `GL_DITHER` is initially enabled.

The following capabilities are accepted for *cap*:

Constant	See
`GL_ALPHA_TEST`	`glAlphaFunc`
`GL_AUTO_NORMAL`	`glEvalCoord`
`GL_BLEND`	`glBlendFunc`, `glLogicOp`
`GL_CLIP_PLANE`*i*	`glClipPlane`
`GL_COLOR_ARRAY`	`glColorPointer`
`GL_COLOR_LOGIC_OP`	`glLogicOp`
`GL_COLOR_MATERIAL`	`glColorMaterial`
`GL_COLOR_SUM`	`glSecondaryColor`
`GL_COLOR_TABLE`	`glColorTable`
`GL_CONVOLUTION_1D`	`glConvolutionFilter1D`
`GL_CONVOLUTION_2D`	`glConvolutionFilter2D`
`GL_CULL_FACE`	`glCullFace`
`GL_DEPTH_TEST`	`glDepthFunc`, `glDepthRange`
`GL_DITHER`	`glEnable`
`GL_EDGE_FLAG_ARRAY`	`glEdgeFlagPointer`
`GL_FOG`	`glFog`

GL_FOG_COORD_ARRAY
: glFogCoordPointer

GL_HISTOGRAM
: glHistogram

GL_INDEX_ARRAY
: glIndexPointer

GL_INDEX_LOGIC_OP
: glLogicOp

GL_LIGHT*i*
: glLightModel, glLight

GL_LIGHTING
: glMaterial, glLightModel, glLight

GL_LINE_SMOOTH
: glLineWidth

GL_LINE_STIPPLE
: glLineStipple

GL_MAP1_COLOR_4
: glMap1

GL_MAP1_INDEX
: glMap1

GL_MAP1_NORMAL
: glMap1

GL_MAP1_TEXTURE_COORD_1
: glMap1

GL_MAP1_TEXTURE_COORD_2
: glMap1

GL_MAP1_TEXTURE_COORD_3
: glMap1

GL_MAP1_TEXTURE_COORD_4
: glMap1

GL_MAP2_COLOR_4
: glMap2

GL_MAP2_INDEX
: glMap2

GL_MAP2_NORMAL
: glMap2

GL_MAP2_TEXTURE_COORD_1
: glMap2

GL_MAP2_TEXTURE_COORD_2
: glMap2

GL_MAP2_TEXTURE_COORD_3
: glMap2

GL_MAP2_TEXTURE_COORD_4
: glMap2

GL_MAP2_VERTEX_3
: glMap2

GL_MAP2_VERTEX_4
: glMap2

GL_MINMAX
: glMinmax

GL_MULTISAMPLE
: glSampleCoverage

GL_NORMAL_ARRAY
: glNormalPointer

GL_NORMALIZE
: glNormal

GL_POINT_SMOOTH
: glPointSize

GL_POINT_SPRITE
: glEnable

GL_POLYGON_SMOOTH
: glPolygonMode

GL_POLYGON_OFFSET_FILL
: glPolygonOffset

GL_POLYGON_OFFSET_LINE
: glPolygonOffset

GL_POLYGON_OFFSET_POINT
: glPolygonOffset

GL_POLYGON_STIPPLE
: glPolygonStipple

GL_POST_COLOR_MATRIX_COLOR_TABLE
: glColorTable

GL_POST_CONVOLUTION_COLOR_TABLE
: glColorTable

GL_RESCALE_NORMAL
: glNormal

GL_SAMPLE_ALPHA_TO_COVERAGE
: glSampleCoverage

`GL_SAMPLE_ALPHA_TO_ONE`
`glSampleCoverage`

`GL_SAMPLE_COVERAGE`
`glSampleCoverage`

`GL_SCISSOR_TEST`
`glScissor`

`GL_SECONDARY_COLOR_ARRAY`
`glSecondaryColorPointer`

`GL_SEPARABLE_2D`
`glSeparableFilter2D`

`GL_STENCIL_TEST`
`glStencilFunc`, `glStencilOp`

`GL_TEXTURE_1D`
`glTexImage1D`

`GL_TEXTURE_2D`
`glTexImage2D`

`GL_TEXTURE_3D`
`glTexImage3D`

`GL_TEXTURE_COORD_ARRAY`
`glTexCoordPointer`

`GL_TEXTURE_CUBE_MAP`
`glTexImage2D`

`GL_TEXTURE_GEN_Q`
`glTexGen`

`GL_TEXTURE_GEN_R`
`glTexGen`

`GL_TEXTURE_GEN_S`
`glTexGen`

`GL_TEXTURE_GEN_T`
`glTexGen`

`GL_VERTEX_ARRAY`
`glVertexPointer`

`GL_VERTEX_PROGRAM_POINT_SIZE`
`glEnable`

`GL_VERTEX_PROGRAM_TWO_SIDE`
`glEnable`

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

`GL_INVALID_OPERATION` is generated if `glIsEnabled` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`GLboolean glIsList` *list* [Function]

Determine if a name corresponds to a display list.

list Specifies a potential display list name.

`glIsList` returns `GL_TRUE` if *list* is the name of a display list and returns `GL_FALSE` if it is not, or if an error occurs.

A name returned by `glGenLists`, but not yet associated with a display list by calling `glNewList`, is not the name of a display list.

`GL_INVALID_OPERATION` is generated if `glIsList` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`GLboolean glIsProgram` *program* [Function]

Determines if a name corresponds to a program object.

program Specifies a potential program object.

`glIsProgram` returns `GL_TRUE` if *program* is the name of a program object previously created with `glCreateProgram` and not yet deleted with `glDeleteProgram`. If *program* is zero or a non-zero value that is not the name of a program object, or if an error occurs, `glIsProgram` returns `GL_FALSE`.

`GL_INVALID_OPERATION` is generated if `glIsProgram` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`GLboolean glIsQuery` *id* [Function]

Determine if a name corresponds to a query object.

id Specifies a value that may be the name of a query object.

`glIsQuery` returns `GL_TRUE` if *id* is currently the name of a query object. If *id* is zero, or is a non-zero value that is not currently the name of a query object, or if an error occurs, `glIsQuery` returns `GL_FALSE`.

A name returned by `glGenQueries`, but not yet associated with a query object by calling `glBeginQuery`, is not the name of a query object.

`GL_INVALID_OPERATION` is generated if `glIsQuery` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`GLboolean glIsShader` *shader* [Function]

Determines if a name corresponds to a shader object.

shader Specifies a potential shader object.

`glIsShader` returns `GL_TRUE` if *shader* is the name of a shader object previously created with `glCreateShader` and not yet deleted with `glDeleteShader`. If *shader* is zero or a non-zero value that is not the name of a shader object, or if an error occurs, `glIsShader` returns `GL_FALSE`.

`GL_INVALID_OPERATION` is generated if `glIsShader` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`GLboolean glIsTexture` *texture* [Function]

Determine if a name corresponds to a texture.

texture Specifies a value that may be the name of a texture.

`glIsTexture` returns `GL_TRUE` if *texture* is currently the name of a texture. If *texture* is zero, or is a non-zero value that is not currently the name of a texture, or if an error occurs, `glIsTexture` returns `GL_FALSE`.

A name returned by `glGenTextures`, but not yet associated with a texture by calling `glBindTexture`, is not the name of a texture.

`GL_INVALID_OPERATION` is generated if `glIsTexture` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glLightModelf` *pname param* [Function]
`void glLightModeli` *pname param* [Function]
`void glLightModelfv` *pname params* [Function]
`void glLightModeliv` *pname params* [Function]
Set the lighting model parameters.

pname Specifies a single-valued lighting model parameter. `GL_LIGHT_MODEL_LOCAL_VIEWER`, `GL_LIGHT_MODEL_COLOR_CONTROL`, and `GL_LIGHT_MODEL_TWO_SIDE` are accepted.

param Specifies the value that *param* will be set to.

`glLightModel` sets the lighting model parameter. *pname* names a parameter and *params* gives the new value. There are three lighting model parameters:

`GL_LIGHT_MODEL_AMBIENT`
params contains four integer or floating-point values that specify the ambient RGBA intensity of the entire scene. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The initial ambient scene intensity is (0.2, 0.2, 0.2, 1.0).

`GL_LIGHT_MODEL_COLOR_CONTROL`
params must be either `GL_SEPARATE_SPECULAR_COLOR` or `GL_SINGLE_COLOR`. `GL_SINGLE_COLOR` specifies that a single color is generated from the lighting computation for a vertex. `GL_SEPARATE_SPECULAR_COLOR` specifies that the specular color computation of lighting be stored separately from the remainder of the lighting computation. The specular color is summed into the generated fragment's color after the application of texture mapping (if enabled). The initial value is `GL_SINGLE_COLOR`.

`GL_LIGHT_MODEL_LOCAL_VIEWER`
params is a single integer or floating-point value that specifies how specular reflection angles are computed. If *params* is 0 (or 0.0), specular reflection angles take the view direction to be parallel to and in the direction of the -*z* axis, regardless of the location of the vertex in eye coordinates. Otherwise, specular reflections are computed from the origin of the eye coordinate system. The initial value is 0.

`GL_LIGHT_MODEL_TWO_SIDE`

params is a single integer or floating-point value that specifies whether one- or two-sided lighting calculations are done for polygons. It has no effect on the lighting calculations for points, lines, or bitmaps. If *params* is 0 (or 0.0), one-sided lighting is specified, and only the *front* material parameters are used in the lighting equation. Otherwise, two-sided lighting is specified. In this case, vertices of back-facing polygons are lighted using the *back* material parameters and have their normals reversed before the lighting equation is evaluated. Vertices of front-facing polygons are always lighted using the *front* material parameters, with no change to their normals. The initial value is 0.

In RGBA mode, the lighted color of a vertex is the sum of the material emission intensity, the product of the material ambient reflectance and the lighting model full-scene ambient intensity, and the contribution of each enabled light source. Each light source contributes the sum of three terms: ambient, diffuse, and specular. The ambient light source contribution is the product of the material ambient reflectance and the light's ambient intensity. The diffuse light source contribution is the product of the material diffuse reflectance, the light's diffuse intensity, and the dot product of the vertex's normal with the normalized vector from the vertex to the light source. The specular light source contribution is the product of the material specular reflectance, the light's specular intensity, and the dot product of the normalized vertex-to-eye and vertex-to-light vectors, raised to the power of the shininess of the material. All three light source contributions are attenuated equally based on the distance from the vertex to the light source and on light source direction, spread exponent, and spread cutoff angle. All dot products are replaced with 0 if they evaluate to a negative value.

The alpha component of the resulting lighted color is set to the alpha value of the material diffuse reflectance.

In color index mode, the value of the lighted index of a vertex ranges from the ambient to the specular values passed to `glMaterial` using `GL_COLOR_INDEXES`. Diffuse and specular coefficients, computed with a (.30, .59, .11) weighting of the lights' colors, the shininess of the material, and the same reflection and attenuation equations as in the RGBA case, determine how much above ambient the resulting index is.

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

`GL_INVALID_ENUM` is generated if *pname* is `GL_LIGHT_MODEL_COLOR_CONTROL` and *params* is not one of `GL_SINGLE_COLOR` or `GL_SEPARATE_SPECULAR_COLOR`.

`GL_INVALID_OPERATION` is generated if `glLightModel` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glLightf` *light pname param* [Function]
`void glLighti` *light pname param* [Function]
`void glLightfv` *light pname params* [Function]
`void glLightiv` *light pname params* [Function]

Set light source 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 i ranges from 0 to the value of GL_MAX_LIGHTS - 1.

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.

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 i ranges from 0 to the value of GL_MAX_LIGHTS - 1. *pname* specifies one of ten light source parameters, again by symbolic name. *params* is either a single value or a pointer to an array that contains the new values.

To enable and disable lighting calculation, call glEnable and glDisable with argument GL_LIGHTING. Lighting is initially disabled. When it is enabled, light sources that are enabled contribute to the lighting calculation. Light source *i* is enabled and disabled using glEnable and glDisable with argument GL_LIGHT*i*.

The ten light parameters are as follows:

GL_AMBIENT

params contains four integer or floating-point values that specify the ambient RGBA intensity of the light. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The initial ambient light intensity is (0, 0, 0, 1).

GL_DIFFUSE

params contains four integer or floating-point values that specify the diffuse RGBA intensity of the light. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The initial value for GL_LIGHT0 is (1, 1, 1, 1); for other lights, the initial value is (0, 0, 0, 1).

GL_SPECULAR

params contains four integer or floating-point values that specify the specular RGBA intensity of the light. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The initial value for GL_LIGHT0 is (1, 1, 1, 1); for other lights, the initial value is (0, 0, 0, 1).

GL_POSITION

params contains four integer or floating-point values that specify the position of the light in homogeneous object coordinates. Both integer and

floating-point values are mapped directly. Neither integer nor floating-point values are clamped.

The position is transformed by the modelview matrix when `glLight` is called (just as if it were a point), and it is stored in eye coordinates. If the *w* component of the position is 0, the light is treated as a directional source. Diffuse and specular lighting calculations take the light's direction, but not its actual position, into account, and attenuation is disabled. Otherwise, diffuse and specular lighting calculations are based on the actual location of the light in eye coordinates, and attenuation is enabled. The initial position is (0, 0, 1, 0); thus, the initial light source is directional, parallel to, and in the direction of the -*z* axis.

`GL_SPOT_DIRECTION`

params contains three integer or floating-point values that specify the direction of the light in homogeneous object coordinates. Both integer and floating-point values are mapped directly. Neither integer nor floating-point values are clamped.

The spot direction is transformed by the upper 3x3 of the modelview matrix when `glLight` is called, and it is stored in eye coordinates. It is significant only when `GL_SPOT_CUTOFF` is not 180, which it is initially. The initial direction is (0,0-1).

`GL_SPOT_EXPONENT`

params is a single integer or floating-point value that specifies the intensity distribution of the light. Integer and floating-point values are mapped directly. Only values in the range [0,128] are accepted.

Effective light intensity is attenuated by the cosine of the angle between the direction of the light and the direction from the light to the vertex being lighted, raised to the power of the spot exponent. Thus, higher spot exponents result in a more focused light source, regardless of the spot cutoff angle (see `GL_SPOT_CUTOFF`, next paragraph). The initial spot exponent is 0, resulting in uniform light distribution.

`GL_SPOT_CUTOFF`

params is a single integer or floating-point value that specifies the maximum spread angle of a light source. Integer and floating-point values are mapped directly. Only values in the range [0,90] and the special value 180 are accepted. If the angle between the direction of the light and the direction from the light to the vertex being lighted is greater than the spot cutoff angle, the light is completely masked. Otherwise, its intensity is controlled by the spot exponent and the attenuation factors. The initial spot cutoff is 180, resulting in uniform light distribution.

`GL_CONSTANT_ATTENUATION`
`GL_LINEAR_ATTENUATION`
`GL_QUADRATIC_ATTENUATION`

params is a single integer or floating-point value that specifies one of the three light attenuation factors. Integer and floating-point values are mapped directly. Only nonnegative values are accepted. If the light

is positional, rather than directional, its intensity is attenuated by the reciprocal of the sum of the constant factor, the linear factor times the distance between the light and the vertex being lighted, and the quadratic factor times the square of the same distance. The initial attenuation factors are (1, 0, 0), resulting in no attenuation.

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

`GL_INVALID_VALUE` is generated if a spot exponent value is specified outside the range [0,128], or if spot cutoff is specified outside the range [0,90] (except for the special value 180), or if a negative attenuation factor is specified.

`GL_INVALID_OPERATION` is generated if `glLight` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glLineStipple` *factor pattern* [Function]

Specify the line stipple pattern.

factor Specifies a multiplier for each bit in the line stipple pattern. If *factor* is 3, for example, each bit in the pattern is used three times before the next bit in the pattern is used. *factor* is clamped to the range [1, 256] and defaults to 1.

pattern Specifies a 16-bit integer whose bit pattern determines which fragments of a line will be drawn when the line is rasterized. Bit zero is used first; the default pattern is all 1's.

Line stippling masks out certain fragments produced by rasterization; those fragments will not be drawn. The masking is achieved by using three parameters: the 16-bit line stipple pattern *pattern*, the repeat count *factor*, and an integer stipple counter *s*.

Counter *s* is reset to 0 whenever `glBegin` is called and before each line segment of a `glBegin(GL_LINES)/glEnd` sequence is generated. It is incremented after each fragment of a unit width aliased line segment is generated or after each *i* fragments of an *i* width line segment are generated. The *i* fragments associated with count *s* are masked out if

pattern bit (*s*/*factor*,)%16

is 0, otherwise these fragments are sent to the frame buffer. Bit zero of *pattern* is the least significant bit.

Antialiased lines are treated as a sequence of 1*width* rectangles for purposes of stippling. Whether rectangle *s* is rasterized or not depends on the fragment rule described for aliased lines, counting rectangles rather than groups of fragments.

To enable and disable line stippling, call `glEnable` and `glDisable` with argument `GL_LINE_STIPPLE`. When enabled, the line stipple pattern is applied as described above. When disabled, it is as if the pattern were all 1's. Initially, line stippling is disabled.

`GL_INVALID_OPERATION` is generated if `glLineStipple` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glLineWidth` *width* [Function]

Specify the width of rasterized lines.

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

`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 $x,>=y,$, 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 `glGet` with arguments `GL_ALIASED_LINE_WIDTH_RANGE`, `GL_SMOOTH_LINE_WIDTH_RANGE`, and `GL_SMOOTH_LINE_WIDTH_GRANULARITY`.

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

`GL_INVALID_OPERATION` is generated if `glLineWidth` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glLinkProgram` *program* [Function]

Links a program object.

program Specifies the handle of the program object to be linked.

`glLinkProgram` links the program object specified by *program*. If any shader objects of type `GL_VERTEX_SHADER` are attached to *program*, they will be used to create an executable that will run on the programmable vertex processor. If any shader objects of type `GL_FRAGMENT_SHADER` are attached to *program*, they will be used to create an executable that will run on the programmable fragment processor.

The status of the link operation will be stored as part of the program object's state. This value will be set to `GL_TRUE` if the program object was linked without errors and is ready for use, and `GL_FALSE` otherwise. It can be queried by calling `glGetProgram` with arguments *program* and `GL_LINK_STATUS`.

As a result of a successful link operation, all active user-defined uniform variables belonging to *program* will be initialized to 0, and each of the program object's active uniform variables will be assigned a location that can be queried by calling `glGetUniformLocation`. Also, any active user-defined attribute variables that have not been bound to a generic vertex attribute index will be bound to one at this time.

Linking of a program object can fail for a number of reasons as specified in the *OpenGL Shading Language Specification*. The following lists some of the conditions that will cause a link error.

- The number of active attribute variables supported by the implementation has been exceeded.
- The storage limit for uniform variables has been exceeded.
- The number of active uniform variables supported by the implementation has been exceeded.
- The `main` function is missing for the vertex shader or the fragment shader.
- A varying variable actually used in the fragment shader is not declared in the same way (or is not declared at all) in the vertex shader.
- A reference to a function or variable name is unresolved.
- A shared global is declared with two different types or two different initial values.
- One or more of the attached shader objects has not been successfully compiled.
- Binding a generic attribute matrix caused some rows of the matrix to fall outside the allowed maximum of `GL_MAX_VERTEX_ATTRIBS`.
- Not enough contiguous vertex attribute slots could be found to bind attribute matrices.

When a program object has been successfully linked, the program object can be made part of current state by calling `glUseProgram`. Whether or not the link operation was successful, the program object's information log will be overwritten. The information log can be retrieved by calling `glGetProgramInfoLog`.

`glLinkProgram` will also install the generated executables as part of the current rendering state if the link operation was successful and the specified program object is already currently in use as a result of a previous call to `glUseProgram`. If the program object currently in use is relinked unsuccessfully, its link status will be set to `GL_FALSE` , but the executables and associated state will remain part of the current state until a subsequent call to `glUseProgram` removes it from use. After it is removed from use, it cannot be made part of current state until it has been successfully relinked.

If *program* contains shader objects of type `GL_VERTEX_SHADER` but does not contain shader objects of type `GL_FRAGMENT_SHADER`, the vertex shader will be linked against the implicit interface for fixed functionality fragment processing. Similarly, if *program* contains shader objects of type `GL_FRAGMENT_SHADER` but it does not contain shader objects of type `GL_VERTEX_SHADER`, the fragment shader will be linked against the implicit interface for fixed functionality vertex processing.

The program object's information log is updated and the program is generated at the time of the link operation. After the link operation, applications are free to modify attached shader objects, compile attached shader objects, detach shader objects, delete shader objects, and attach additional shader objects. None of these operations affects the information log or the program that is part of the program object.

`GL_INVALID_VALUE` is generated if *program* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *program* is not a program object.

`GL_INVALID_OPERATION` is generated if `glLinkProgram` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glListBase` *base* [Function]

Set the display-list base for .

base Specifies an integer offset that will be added to `glCallLists` offsets to generate display-list names. The initial value is 0.

`glCallLists` specifies an array of offsets. Display-list names are generated by adding *base* to each offset. Names that reference valid display lists are executed; the others are ignored.

`GL_INVALID_OPERATION` is generated if `glListBase` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glLoadIdentity` [Function]

Replace the current matrix with the identity matrix.

`glLoadIdentity` replaces the current matrix with the identity matrix. It is semantically equivalent to calling `glLoadMatrix` with the identity matrix

((1 0 0 0), (0 1 0 0), (0 0 1 0), (0 0 0 1),,)

but in some cases it is more efficient.

`GL_INVALID_OPERATION` is generated if `glLoadIdentity` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glLoadMatrixd` *m* [Function]
`void glLoadMatrixf` *m* [Function]

Replace the current matrix with the specified matrix.

m Specifies a pointer to 16 consecutive values, which are used as the elements of a 44 column-major matrix.

`glLoadMatrix` replaces the current matrix with the one whose elements are specified by *m*. The current matrix is the projection matrix, modelview matrix, or texture matrix, depending on the current matrix mode (see `glMatrixMode`).

The current matrix, M, defines a transformation of coordinates. For instance, assume M refers to the modelview matrix. If v=(v[0,],v[1,]v[2,]v[3,]) is the set of object coordinates of a vertex, and *m* points to an array of 16 single- or double-precision floating-point values m={m[0,],m[1,]...m[15,]}, then the modelview transformation $M(v,)$ does the following:

$M(v,)$=((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,]),)((v[0,]), (v[1,]), (v[2,]), (v[3,]),)

Projection and texture transformations are similarly defined.

`GL_INVALID_OPERATION` is generated if `glLoadMatrix` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glLoadName` *name* [Function]

Load a name onto the name stack.

name Specifies a name that will replace the top value on the name stack.

The name stack is used during selection mode to allow sets of rendering commands to be uniquely identified. It consists of an ordered set of unsigned integers and is initially empty.

`glLoadName` causes *name* to replace the value on the top of the name stack.

The name stack is always empty while the render mode is not `GL_SELECT`. Calls to `glLoadName` while the render mode is not `GL_SELECT` are ignored.

`GL_INVALID_OPERATION` is generated if `glLoadName` is called while the name stack is empty.

`GL_INVALID_OPERATION` is generated if `glLoadName` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glLoadTransposeMatrixd` *m* [Function]
`void glLoadTransposeMatrixf` *m* [Function]

Replace the current matrix with the specified row-major ordered matrix.

m Specifies a pointer to 16 consecutive values, which are used as the elements of a 44 row-major matrix.

`glLoadTransposeMatrix` replaces the current matrix with the one whose elements are specified by *m*. The current matrix is the projection matrix, modelview matrix, or texture matrix, depending on the current matrix mode (see `glMatrixMode`).

The current matrix, M, defines a transformation of coordinates. For instance, assume M refers to the modelview matrix. If v=(v[0,],v[1,]v[2,]v[3,]) is the set of object coordinates of a vertex, and *m* points to an array of 16 single- or double-precision floating-point values m={m[0,],m[1,]...m[15,]}, then the modelview transformation M(v,) does the following:

M(v,)=((m[0,] m[1,] m[2,] m[3,]), (m[4,] m[5,] m[6,] m[7,]), (m[8,] m[9,] m[10,] m[11,]), (m[12,] m[13,] m[14,] m[15,]),)((v[0,]), (v[1,]), (v[2,]), (v[3,]),)

Projection and texture transformations are similarly defined.

Calling `glLoadTransposeMatrix` with matrix M is identical in operation to `glLoadMatrix` with M^T, where T represents the transpose.

`GL_INVALID_OPERATION` is generated if `glLoadTransposeMatrix` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glLogicOp` *opcode* [Function]

Specify a logical pixel operation for color index rendering.

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`.

`glLogicOp` specifies a logical operation that, when enabled, is applied between the incoming color index or RGBA color and the color index or RGBA color at the corresponding location in the frame buffer. To enable or disable the logical operation, call `glEnable` and `glDisable` using the symbolic constant `GL_COLOR_LOGIC_OP` for RGBA mode or `GL_INDEX_LOGIC_OP` for color index mode. The initial value is disabled for both operations.

Opcode	Resulting Operation
GL_CLEAR	0
GL_SET	1
GL_COPY	s
GL_COPY_INVERTED	~s
GL_NOOP	d
GL_INVERT	~d
GL_AND	s & d
GL_NAND	~(s & d)
GL_OR	s | d
GL_NOR	~(s | d)
GL_XOR	s ^ d
GL_EQUIV	~(s ^ d)
GL_AND_REVERSE	s & ~d
GL_AND_INVERTED	~s & d
GL_OR_REVERSE	s | ~d
GL_OR_INVERTED	~s | d

opcode is a symbolic constant chosen from the list above. In the explanation of the logical operations, *s* represents the incoming color index and *d* represents the index in the frame buffer. Standard C-language operators are used. As these bitwise operators suggest, the logical operation is applied independently to each bit pair of the source and destination indices or colors.

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

`GL_INVALID_OPERATION` is generated if `glLogicOp` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glMap1f` *target u1 u2 stride order points* [Function]
`void glMap1d` *target u1 u2 stride order points* [Function]

Define a one-dimensional evaluator.

target Specifies the kind of values that are generated by the evaluator. Symbolic constants `GL_MAP1_VERTEX_3`, `GL_MAP1_VERTEX_4`, `GL_MAP1_INDEX`, `GL_MAP1_COLOR_4`, `GL_MAP1_NORMAL`, `GL_MAP1_TEXTURE_COORD_1`, `GL_MAP1_TEXTURE_COORD_2`, `GL_MAP1_TEXTURE_COORD_3`, and `GL_MAP1_TEXTURE_COORD_4` are accepted.

u1
u2 Specify a linear mapping of *u*, as presented to `glEvalCoord1`, to *u^*, the variable that is evaluated by the equations specified by this command.

stride Specifies the number of floats or doubles between the beginning of one control point and the beginning of the next one in the data structure referenced in *points*. This allows control points to be embedded in arbitrary data structures. The only constraint is that the values for a particular control point must occupy contiguous memory locations.

order Specifies the number of control points. Must be positive.

points Specifies a pointer to the array of control points.

Evaluators provide a way to use polynomial or rational polynomial mapping to produce vertices, normals, texture coordinates, and colors. The values produced by an evaluator are sent to further stages of GL processing just as if they had been presented using `glVertex`, `glNormal`, `glTexCoord`, and `glColor` commands, except that the generated values do not update the current normal, texture coordinates, or color.

All polynomial or rational polynomial splines of any degree (up to the maximum degree supported by the GL implementation) can be described using evaluators. These include almost all splines used in computer graphics: B-splines, Bezier curves, Hermite splines, and so on.

Evaluators define curves based on Bernstein polynomials. Define $p(u\hat{},)$ as

$p(u\hat{},)=i=0nB\_i,\hat{}n(u\hat{},)R\_i$

where R_i is a control point and $B\_i,\hat{}n(u\hat{},)$ is the ith Bernstein polynomial of degree n ($order = n+1$):

$B\_i,\hat{}n(u\hat{},)=((n), (i),,)u\hat{},\hat{}i(1-u\hat{},)\hat{}n-i,,$

Recall that

$0\hat{}0==1$ and $((n), (0),,)==1$

`glMap1` is used to define the basis and to specify what kind of values are produced. Once defined, a map can be enabled and disabled by calling `glEnable` and `glDisable` with the map name, one of the nine predefined values for *target* described below. `glEvalCoord1` evaluates the one-dimensional maps that are enabled. When `glEvalCoord1` presents a value *u*, the Bernstein functions are evaluated using *u^*, where $u\hat{}=u-u1,/u2-u1,$

target is a symbolic constant that indicates what kind of control points are provided in *points*, and what output is generated when the map is evaluated. It can assume one of nine predefined values:

`GL_MAP1_VERTEX_3`
Each control point is three floating-point values representing *x*, *y*, and *z*. Internal `glVertex3` commands are generated when the map is evaluated.

`GL_MAP1_VERTEX_4`
Each control point is four floating-point values representing *x*, *y*, *z*, and *w*. Internal `glVertex4` commands are generated when the map is evaluated.

`GL_MAP1_INDEX`
: Each control point is a single floating-point value representing a color index. Internal `glIndex` commands are generated when the map is evaluated but the current index is not updated with the value of these `glIndex` commands.

`GL_MAP1_COLOR_4`
: Each control point is four floating-point values representing red, green, blue, and alpha. Internal `glColor4` commands are generated when the map is evaluated but the current color is not updated with the value of these `glColor4` commands.

`GL_MAP1_NORMAL`
: Each control point is three floating-point values representing the x, y, and z components of a normal vector. Internal `glNormal` commands are generated when the map is evaluated but the current normal is not updated with the value of these `glNormal` commands.

`GL_MAP1_TEXTURE_COORD_1`
: Each control point is a single floating-point value representing the s texture coordinate. Internal `glTexCoord1` commands are generated when the map is evaluated but the current texture coordinates are not updated with the value of these `glTexCoord` commands.

`GL_MAP1_TEXTURE_COORD_2`
: Each control point is two floating-point values representing the s and t texture coordinates. Internal `glTexCoord2` commands are generated when the map is evaluated but the current texture coordinates are not updated with the value of these `glTexCoord` commands.

`GL_MAP1_TEXTURE_COORD_3`
: Each control point is three floating-point values representing the s, t, and r texture coordinates. Internal `glTexCoord3` commands are generated when the map is evaluated but the current texture coordinates are not updated with the value of these `glTexCoord` commands.

`GL_MAP1_TEXTURE_COORD_4`
: Each control point is four floating-point values representing the s, t, r, and q texture coordinates. Internal `glTexCoord4` commands are generated when the map is evaluated but the current texture coordinates are not updated with the value of these `glTexCoord` commands.

stride, *order*, and *points* define the array addressing for accessing the control points. *points* is the location of the first control point, which occupies one, two, three, or four contiguous memory locations, depending on which map is being defined. *order* is the number of control points in the array. *stride* specifies how many float or double locations to advance the internal memory pointer to reach the next control point.

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

`GL_INVALID_VALUE` is generated if *u1* is equal to *u2*.

`GL_INVALID_VALUE` is generated if *stride* is less than the number of values in a control point.

`GL_INVALID_VALUE` is generated if *order* is less than 1 or greater than the return value of `GL_MAX_EVAL_ORDER`.

`GL_INVALID_OPERATION` is generated if `glMap1` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`GL_INVALID_OPERATION` is generated if `glMap1` is called and the value of `GL_ACTIVE_TEXTURE` is not `GL_TEXTURE0`.

`void glMap2f` *target u1 u2 ustride uorder v1 v2 vstride vorder points* [Function]
`void glMap2d` *target u1 u2 ustride uorder v1 v2 vstride vorder points* [Function]
Define a two-dimensional evaluator.

target Specifies the kind of values that are generated by the evaluator. Symbolic constants `GL_MAP2_VERTEX_3`, `GL_MAP2_VERTEX_4`, `GL_MAP2_INDEX`, `GL_MAP2_COLOR_4`, `GL_MAP2_NORMAL`, `GL_MAP2_TEXTURE_COORD_1`, `GL_MAP2_TEXTURE_COORD_2`, `GL_MAP2_TEXTURE_COORD_3`, and `GL_MAP2_TEXTURE_COORD_4` are accepted.

u1
u2 Specify a linear mapping of u, as presented to `glEvalCoord2`, to $u\hat{}$, one of the two variables that are evaluated by the equations specified by this command. Initially, *u1* is 0 and *u2* is 1.

ustride Specifies the number of floats or doubles between the beginning of control point *R_ij* and the beginning of control point *R_*(i+1,)j,, where i and j are the u and v control point indices, respectively. This allows control points to be embedded in arbitrary data structures. The only constraint is that the values for a particular control point must occupy contiguous memory locations. The initial value of *ustride* is 0.

uorder Specifies the dimension of the control point array in the u axis. Must be positive. The initial value is 1.

v1
v2 Specify a linear mapping of v, as presented to `glEvalCoord2`, to $v\hat{}$, one of the two variables that are evaluated by the equations specified by this command. Initially, *v1* is 0 and *v2* is 1.

vstride Specifies the number of floats or doubles between the beginning of control point *R_ij* and the beginning of control point *R_i*(j+1,),, where i and j are the u and v control point indices, respectively. This allows control points to be embedded in arbitrary data structures. The only constraint is that the values for a particular control point must occupy contiguous memory locations. The initial value of *vstride* is 0.

vorder Specifies the dimension of the control point array in the v axis. Must be positive. The initial value is 1.

points Specifies a pointer to the array of control points.

Evaluators provide a way to use polynomial or rational polynomial mapping to produce vertices, normals, texture coordinates, and colors. The values produced by an evaluator are sent on to further stages of GL processing just as if they had been

presented using `glVertex`, `glNormal`, `glTexCoord`, and `glColor` commands, except that the generated values do not update the current normal, texture coordinates, or color.

All polynomial or rational polynomial splines of any degree (up to the maximum degree supported by the GL implementation) can be described using evaluators. These include almost all surfaces used in computer graphics, including B-spline surfaces, NURBS surfaces, Bezier surfaces, and so on.

Evaluators define surfaces based on bivariate Bernstein polynomials. Define $p(u\hat{},v\hat{})$ as

$p(u\hat{},v\hat{})=i=0nj=0mB\_i,\hat{}n(u\hat{},)B\_j,\hat{}m(v\hat{},)R\_ij$

where R_ij is a control point, $B\_i,\hat{}n(u\hat{},)$ is the ith Bernstein polynomial of degree n ($uorder = n+1$)

$B\_i,\hat{}n(u\hat{},)=((n),\ (i),,)u\hat{},\hat{}i(1\text{-}u\hat{},)\hat{}n\text{-}i,,$

and $B\_j,\hat{}m(v\hat{},)$ is the jth Bernstein polynomial of degree m ($vorder = m+1$)

$B\_j,\hat{}m(v\hat{},)=((m),\ (j),,)v\hat{},\hat{}j(1\text{-}v\hat{},)\hat{}m\text{-}j,,$

Recall that $0\hat{}0==1$ and $((n),\ (0),,)==1$

`glMap2` is used to define the basis and to specify what kind of values are produced. Once defined, a map can be enabled and disabled by calling `glEnable` and `glDisable` with the map name, one of the nine predefined values for *target*, described below. When `glEvalCoord2` presents values u and v, the bivariate Bernstein polynomials are evaluated using $u\hat{}$ and $v\hat{}$, where

$u\hat{}=u\text{-}u1,/u2\text{-}u1,$

$v\hat{}=v\text{-}v1,/v2\text{-}v1,$

target is a symbolic constant that indicates what kind of control points are provided in *points*, and what output is generated when the map is evaluated. It can assume one of nine predefined values:

`GL_MAP2_VERTEX_3`
: Each control point is three floating-point values representing x, y, and z. Internal `glVertex3` commands are generated when the map is evaluated.

`GL_MAP2_VERTEX_4`
: Each control point is four floating-point values representing x, y, z, and w. Internal `glVertex4` commands are generated when the map is evaluated.

`GL_MAP2_INDEX`
: Each control point is a single floating-point value representing a color index. Internal `glIndex` commands are generated when the map is evaluated but the current index is not updated with the value of these `glIndex` commands.

`GL_MAP2_COLOR_4`
: Each control point is four floating-point values representing red, green, blue, and alpha. Internal `glColor4` commands are generated when the map is evaluated but the current color is not updated with the value of these `glColor4` commands.

`GL_MAP2_NORMAL`
: Each control point is three floating-point values representing the x, y, and z components of a normal vector. Internal `glNormal` commands are generated when the map is evaluated but the current normal is not updated with the value of these `glNormal` commands.

`GL_MAP2_TEXTURE_COORD_1`
: Each control point is a single floating-point value representing the s texture coordinate. Internal `glTexCoord1` commands are generated when the map is evaluated but the current texture coordinates are not updated with the value of these `glTexCoord` commands.

`GL_MAP2_TEXTURE_COORD_2`
: Each control point is two floating-point values representing the s and t texture coordinates. Internal `glTexCoord2` commands are generated when the map is evaluated but the current texture coordinates are not updated with the value of these `glTexCoord` commands.

`GL_MAP2_TEXTURE_COORD_3`
: Each control point is three floating-point values representing the s, t, and r texture coordinates. Internal `glTexCoord3` commands are generated when the map is evaluated but the current texture coordinates are not updated with the value of these `glTexCoord` commands.

`GL_MAP2_TEXTURE_COORD_4`
: Each control point is four floating-point values representing the s, t, r, and q texture coordinates. Internal `glTexCoord4` commands are generated when the map is evaluated but the current texture coordinates are not updated with the value of these `glTexCoord` commands.

ustride, *uorder*, *vstride*, *vorder*, and *points* define the array addressing for accessing the control points. *points* is the location of the first control point, which occupies one, two, three, or four contiguous memory locations, depending on which map is being defined. There are *uorder*×*vorder* control points in the array. *ustride* specifies how many float or double locations are skipped to advance the internal memory pointer from control point R_ij, to control point $R\_(i+1,)j,$. *vstride* specifies how many float or double locations are skipped to advance the internal memory pointer from control point R_ij, to control point $R\_i(j+1,),$.

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

`GL_INVALID_VALUE` is generated if *u1* is equal to *u2*, or if *v1* is equal to *v2*.

`GL_INVALID_VALUE` is generated if either *ustride* or *vstride* is less than the number of values in a control point.

`GL_INVALID_VALUE` is generated if either *uorder* or *vorder* is less than 1 or greater than the return value of `GL_MAX_EVAL_ORDER`.

`GL_INVALID_OPERATION` is generated if `glMap2` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`GL_INVALID_OPERATION` is generated if `glMap2` is called and the value of `GL_ACTIVE_TEXTURE` is not `GL_TEXTURE0`.

`void-* glMapBuffer` *target access* [Function]
`GLboolean glUnmapBuffer` *target* [Function]

Map a buffer object's data store.

target Specifies the target buffer object being mapped. The symbolic constant must be `GL_ARRAY_BUFFER`, `GL_ELEMENT_ARRAY_BUFFER`, `GL_PIXEL_PACK_BUFFER`, or `GL_PIXEL_UNPACK_BUFFER`.

access Specifies the access policy, indicating whether it will be possible to read from, write to, or both read from and write to the buffer object's mapped data store. The symbolic constant must be `GL_READ_ONLY`, `GL_WRITE_ONLY`, or `GL_READ_WRITE`.

`glMapBuffer` maps to the client's address space the entire data store of the buffer object currently bound to *target*. The data can then be directly read and/or written relative to the returned pointer, depending on the specified *access* policy. If the GL is unable to map the buffer object's data store, `glMapBuffer` generates an error and returns `NULL`. This may occur for system-specific reasons, such as low virtual memory availability.

If a mapped data store is accessed in a way inconsistent with the specified *access* policy, no error is generated, but performance may be negatively impacted and system errors, including program termination, may result. Unlike the *usage* parameter of `glBufferData`, *access* is not a hint, and does in fact constrain the usage of the mapped data store on some GL implementations. In order to achieve the highest performance available, a buffer object's data store should be used in ways consistent with both its specified *usage* and *access* parameters.

A mapped data store must be unmapped with `glUnmapBuffer` before its buffer object is used. Otherwise an error will be generated by any GL command that attempts to dereference the buffer object's data store. When a data store is unmapped, the pointer to its data store becomes invalid. `glUnmapBuffer` returns `GL_TRUE` unless the data store contents have become corrupt during the time the data store was mapped. This can occur for system-specific reasons that affect the availability of graphics memory, such as screen mode changes. In such situations, `GL_FALSE` is returned and the data store contents are undefined. An application must detect this rare condition and reinitialize the data store.

A buffer object's mapped data store is automatically unmapped when the buffer object is deleted or its data store is recreated with `glBufferData`.

`GL_INVALID_ENUM` is generated if *target* is not `GL_ARRAY_BUFFER`, `GL_ELEMENT_ARRAY_BUFFER`, `GL_PIXEL_PACK_BUFFER`, or `GL_PIXEL_UNPACK_BUFFER`.

`GL_INVALID_ENUM` is generated if *access* is not `GL_READ_ONLY`, `GL_WRITE_ONLY`, or `GL_READ_WRITE`.

`GL_OUT_OF_MEMORY` is generated when `glMapBuffer` is executed if the GL is unable to map the buffer object's data store. This may occur for a variety of system-specific reasons, such as the absence of sufficient remaining virtual memory.

`GL_INVALID_OPERATION` is generated if the reserved buffer object name 0 is bound to *target*.

`GL_INVALID_OPERATION` is generated if `glMapBuffer` is executed for a buffer object whose data store is already mapped.

`GL_INVALID_OPERATION` is generated if `glUnmapBuffer` is executed for a buffer object whose data store is not currently mapped.

`GL_INVALID_OPERATION` is generated if `glMapBuffer` or `glUnmapBuffer` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glMapGrid1d` *un u1 u2* [Function]
`void glMapGrid1f` *un u1 u2* [Function]
`void glMapGrid2d` *un u1 u2 vn v1 v2* [Function]
`void glMapGrid2f` *un u1 u2 vn v1 v2* [Function]

Define a one- or two-dimensional mesh.

un Specifies the number of partitions in the grid range interval [*u1*, *u2*]. Must be positive.

u1
u2 Specify the mappings for integer grid domain values i=0 and i=*un*.

vn Specifies the number of partitions in the grid range interval [*v1*, *v2*] (`glMapGrid2` only).

v1
v2 Specify the mappings for integer grid domain values j=0 and j=*vn* (`glMapGrid2` only).

`glMapGrid` and `glEvalMesh` are used together to efficiently generate and evaluate a series of evenly-spaced map domain values. `glEvalMesh` steps through the integer domain of a one- or two-dimensional grid, whose range is the domain of the evaluation maps specified by `glMap1` and `glMap2`.

`glMapGrid1` and `glMapGrid2` specify the linear grid mappings between the i (or i and j) integer grid coordinates, to the u (or u and v) floating-point evaluation map coordinates. See `glMap1` and `glMap2` for details of how u and v coordinates are evaluated.

`glMapGrid1` specifies a single linear mapping such that integer grid coordinate 0 maps exactly to *u1*, and integer grid coordinate *un* maps exactly to *u2*. All other integer grid coordinates i are mapped so that

$u = i(u2-u1,)/un+u1$

`glMapGrid2` specifies two such linear mappings. One maps integer grid coordinate i=0 exactly to *u1*, and integer grid coordinate i=*un* exactly to *u2*. The other maps integer grid coordinate j=0 exactly to *v1*, and integer grid coordinate j=*vn* exactly to *v2*. Other integer grid coordinates i and j are mapped such that

$u = i(u2-u1,)/un+u1$

$v = j(v2-v1,)/vn+v1$

The mappings specified by `glMapGrid` are used identically by `glEvalMesh` and `glEvalPoint`.

`GL_INVALID_VALUE` is generated if either *un* or *vn* is not positive.

`GL_INVALID_OPERATION` is generated if `glMapGrid` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glMaterialf` *face pname param* [Function]
`void glMaterialiali` *face pname param* [Function]
`void glMaterialfv` *face pname params* [Function]
`void glMaterialiv` *face pname params* [Function]

Specify material parameters for the lighting model.

face Specifies which face or faces are being updated. Must be one of `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`.

pname Specifies the single-valued material parameter of the face or faces that is being updated. Must be `GL_SHININESS`.

param Specifies the value that parameter `GL_SHININESS` will be set to.

`glMaterial` assigns values to material parameters. There are two matched sets of material parameters. One, the *front-facing* set, is used to shade points, lines, bitmaps, and all polygons (when two-sided lighting is disabled), or just front-facing polygons (when two-sided lighting is enabled). The other set, *back-facing*, is used to shade back-facing polygons only when two-sided lighting is enabled. Refer to the `glLightModel` reference page for details concerning one- and two-sided lighting calculations.

`glMaterial` takes three arguments. The first, *face*, specifies whether the `GL_FRONT` materials, the `GL_BACK` materials, or both `GL_FRONT_AND_BACK` materials will be modified. The second, *pname*, specifies which of several parameters in one or both sets will be modified. The third, *params*, specifies what value or values will be assigned to the specified parameter.

Material parameters are used in the lighting equation that is optionally applied to each vertex. The equation is discussed in the `glLightModel` reference page. The parameters that can be specified using `glMaterial`, and their interpretations by the lighting equation, are as follows:

`GL_AMBIENT`
params contains four integer or floating-point values that specify the ambient RGBA reflectance of the material. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The initial ambient reflectance for both front- and back-facing materials is (0.2, 0.2, 0.2, 1.0).

`GL_DIFFUSE`
params contains four integer or floating-point values that specify the diffuse RGBA reflectance of the material. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The initial diffuse reflectance for both front- and back-facing materials is (0.8, 0.8, 0.8, 1.0).

`GL_SPECULAR`
params contains four integer or floating-point values that specify the specular RGBA reflectance of the material. Integer values are mapped lin-

early such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The initial specular reflectance for both front- and back-facing materials is (0, 0, 0, 1).

`GL_EMISSION`
: *params* contains four integer or floating-point values that specify the RGBA emitted light intensity of the material. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The initial emission intensity for both front- and back-facing materials is (0, 0, 0, 1).

`GL_SHININESS`
: *params* is a single integer or floating-point value that specifies the RGBA specular exponent of the material. Integer and floating-point values are mapped directly. Only values in the range [0,128] are accepted. The initial specular exponent for both front- and back-facing materials is 0.

`GL_AMBIENT_AND_DIFFUSE`
: Equivalent to calling `glMaterial` twice with the same parameter values, once with `GL_AMBIENT` and once with `GL_DIFFUSE`.

`GL_COLOR_INDEXES`
: *params* contains three integer or floating-point values specifying the color indices for ambient, diffuse, and specular lighting. These three values, and `GL_SHININESS`, are the only material values used by the color index mode lighting equation. Refer to the `glLightModel` reference page for a discussion of color index lighting.

`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.

`void glMatrixMode` *mode* [Function]

Specify which matrix is the current matrix.

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`. Additionally, if the `ARB_imaging` extension is supported, `GL_COLOR` is also accepted.

`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.

`GL_COLOR`
: Applies subsequent matrix operations to the color matrix stack.

To find out which matrix stack is currently the target of all matrix operations, call `glGet` with argument `GL_MATRIX_MODE`. The initial value is `GL_MODELVIEW`.

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

`GL_INVALID_OPERATION` is generated if `glMatrixMode` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glMinmax` *target internalformat sink* [Function]

Define minmax table.

target
: The minmax table whose parameters are to be set. Must be `GL_MINMAX`.

internalformat
: The format of entries in the minmax table. Must be one of `GL_ALPHA`, `GL_ALPHA4`, `GL_ALPHA8`, `GL_ALPHA12`, `GL_ALPHA16`, `GL_LUMINANCE`, `GL_LUMINANCE4`, `GL_LUMINANCE8`, `GL_LUMINANCE12`, `GL_LUMINANCE16`, `GL_LUMINANCE_ALPHA`, `GL_LUMINANCE4_ALPHA4`, `GL_LUMINANCE6_ALPHA2`, `GL_LUMINANCE8_ALPHA8`, `GL_LUMINANCE12_ALPHA4`, `GL_LUMINANCE12_ALPHA12`, `GL_LUMINANCE16_ALPHA16`, `GL_R3_G3_B2`, `GL_RGB`, `GL_RGB4`, `GL_RGB5`, `GL_RGB8`, `GL_RGB10`, `GL_RGB12`, `GL_RGB16`, `GL_RGBA`, `GL_RGBA2`, `GL_RGBA4`, `GL_RGB5_A1`, `GL_RGBA8`, `GL_RGB10_A2`, `GL_RGBA12`, or `GL_RGBA16`.

sink
: If `GL_TRUE`, pixels will be consumed by the minmax process and no drawing or texture loading will take place. If `GL_FALSE`, pixels will proceed to the final conversion process after minmax.

When `GL_MINMAX` is enabled, the RGBA components of incoming pixels are compared to the minimum and maximum values for each component, which are stored in the two-element minmax table. (The first element stores the minima, and the second element stores the maxima.) If a pixel component is greater than the corresponding component in the maximum element, then the maximum element is updated with the pixel component value. If a pixel component is less than the corresponding component in the minimum element, then the minimum element is updated with the pixel component value. (In both cases, if the internal format of the minmax table includes luminance, then the R color component of incoming pixels is used for comparison.) The contents of the minmax table may be retrieved at a later time by calling `glGetMinmax`. The minmax operation is enabled or disabled by calling `glEnable` or `glDisable`, respectively, with an argument of `GL_MINMAX`.

`glMinmax` redefines the current minmax table to have entries of the format specified by *internalformat*. The maximum element is initialized with the smallest possible component values, and the minimum element is initialized with the largest possible component values. The values in the previous minmax table, if any, are lost. If *sink* is `GL_TRUE`, then pixels are discarded after minmax; no further processing of the pixels takes place, and no drawing, texture loading, or pixel readback will result.

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

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

`GL_INVALID_OPERATION` is generated if `glMinmax` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glMultiDrawArrays` *mode first count primcount* [Function]
Render multiple sets of primitives from array data.

mode Specifies what kind of primitives to render. Symbolic constants `GL_POINTS`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_LINES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_TRIANGLES`, `GL_QUAD_STRIP`, `GL_QUADS`, and `GL_POLYGON` are accepted.

first Points to an array of starting indices in the enabled arrays.

count Points to an array of the number of indices to be rendered.

primcount Specifies the size of the first and count

`glMultiDrawArrays` specifies multiple sets of geometric primitives with very few subroutine calls. Instead of calling a GL procedure to pass each individual vertex, normal, texture coordinate, edge flag, or color, you can prespecify separate arrays of vertices, normals, and colors and use them to construct a sequence of primitives with a single call to `glMultiDrawArrays`.

`glMultiDrawArrays` behaves identically to `glDrawArrays` except that *primcount* separate ranges of elements are specified instead.

When `glMultiDrawArrays` 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 `glMultiDrawArrays` have an unspecified value after `glMultiDrawArrays` returns. For example, if `GL_COLOR_ARRAY` is enabled, the value of the current color is undefined after `glMultiDrawArrays` executes. Attributes that aren't modified remain well defined.

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

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

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to an enabled array and the buffer object's data store is currently mapped.

`GL_INVALID_OPERATION` is generated if `glMultiDrawArrays` is executed between the execution of `glBegin` and the corresponding `glEnd`.

`void glMultiDrawElements` *mode count type indices primcount* [Function]
Render multiple sets of primitives by specifying indices of array data elements.

mode Specifies what kind of primitives to render. Symbolic constants `GL_POINTS`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_LINES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_TRIANGLES`, `GL_QUAD_STRIP`, `GL_QUADS`, and `GL_POLYGON` are accepted.

count Points to an array of the elements counts.

type Specifies the type of the values in *indices*. Must be one of `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT`, or `GL_UNSIGNED_INT`.

indices Specifies a pointer to the location where the indices are stored.

primcount Specifies the size of the *count* array.

`glMultiDrawElements` specifies multiple sets of geometric primitives with very few subroutine calls. Instead of calling a GL function to pass each individual vertex, normal, texture coordinate, edge flag, or color, you can prespecify separate arrays of vertices, normals, and so on, and use them to construct a sequence of primitives with a single call to `glMultiDrawElements`.

`glMultiDrawElements` is identical in operation to `glDrawElements` except that *primcount* separate lists of elements are specified.

Vertex attributes that are modified by `glMultiDrawElements` have an unspecified value after `glMultiDrawElements` returns. For example, if `GL_COLOR_ARRAY` is enabled, the value of the current color is undefined after `glMultiDrawElements` executes. Attributes that aren't modified maintain their previous values.

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

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

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to an enabled array or the element array and the buffer object's data store is currently mapped.

`GL_INVALID_OPERATION` is generated if `glMultiDrawElements` is executed between the execution of `glBegin` and the corresponding `glEnd`.

`void glMultiTexCoord1s` *target s* [Function]
`void glMultiTexCoord1i` *target s* [Function]
`void glMultiTexCoord1f` *target s* [Function]
`void glMultiTexCoord1d` *target s* [Function]
`void glMultiTexCoord2s` *target s t* [Function]
`void glMultiTexCoord2i` *target s t* [Function]
`void glMultiTexCoord2f` *target s t* [Function]
`void glMultiTexCoord2d` *target s t* [Function]
`void glMultiTexCoord3s` *target s t r* [Function]
`void glMultiTexCoord3i` *target s t r* [Function]
`void glMultiTexCoord3f` *target s t r* [Function]
`void glMultiTexCoord3d` *target s t r* [Function]
`void glMultiTexCoord4s` *target s t r q* [Function]
`void glMultiTexCoord4i` *target s t r q* [Function]
`void glMultiTexCoord4f` *target s t r q* [Function]
`void glMultiTexCoord4d` *target s t r q* [Function]
`void glMultiTexCoord1sv` *target v* [Function]
`void glMultiTexCoord1iv` *target v* [Function]
`void glMultiTexCoord1fv` *target v* [Function]
`void glMultiTexCoord1dv` *target v* [Function]
`void glMultiTexCoord2sv` *target v* [Function]
`void glMultiTexCoord2iv` *target v* [Function]

`void glMultiTexCoord2fv` *target v* [Function]
`void glMultiTexCoord2dv` *target v* [Function]
`void glMultiTexCoord3sv` *target v* [Function]
`void glMultiTexCoord3iv` *target v* [Function]
`void glMultiTexCoord3fv` *target v* [Function]
`void glMultiTexCoord3dv` *target v* [Function]
`void glMultiTexCoord4sv` *target v* [Function]
`void glMultiTexCoord4iv` *target v* [Function]
`void glMultiTexCoord4fv` *target v* [Function]
`void glMultiTexCoord4dv` *target v* [Function]

Set the current texture coordinates.

target Specifies the texture unit whose coordinates should be modified. The number of texture units is implementation dependent, but must be at least two. Symbolic constant must be one of `GL_TEXTURE`*i*, where i ranges from 0 to `GL_MAX_TEXTURE_COORDS` - 1, which is an implementation-dependent value.

s
t
r
q Specify *s*, *t*, *r*, and *q* texture coordinates for *target* texture unit. Not all parameters are present in all forms of the command.

`glMultiTexCoord` specifies texture coordinates in one, two, three, or four dimensions. `glMultiTexCoord1` sets the current texture coordinates to (*s*,001); a call to `glMultiTexCoord2` sets them to (*s*,*t*01). Similarly, `glMultiTexCoord3` specifies the texture coordinates as (*s*,*tr*1), and `glMultiTexCoord4` defines all four components explicitly as (*s*,*trq*).

The current texture coordinates are part of the data that is associated with each vertex and with the current raster position. Initially, the values for (*s*,*trq*) are (0,001).

`void glMultMatrixd` *m* [Function]
`void glMultMatrixf` *m* [Function]

Multiply the current matrix with the specified matrix.

m Points to 16 consecutive values that are used as the elements of a 44 column-major matrix.

`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.

`GL_INVALID_OPERATION` is generated if `glMultMatrix` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glMultTransposeMatrixd` *m* [Function]
`void glMultTransposeMatrixf` *m* [Function]

Multiply the current matrix with the specified row-major ordered matrix.

m Points to 16 consecutive values that are used as the elements of a 44 row-major matrix.

`glMultTransposeMatrix` 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.

`GL_INVALID_OPERATION` is generated if `glMultTransposeMatrix` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glNewList` *list mode* [Function]
`void glEndList` [Function]

Create or replace a display list.

list Specifies the display-list name.

mode Specifies the compilation mode, which can be `GL_COMPILE` or `GL_COMPILE_AND_EXECUTE`.

Display lists are groups of GL commands that have been stored for subsequent execution. Display lists are created with `glNewList`. All subsequent commands are placed in the display list, in the order issued, until `glEndList` is called.

`glNewList` has two arguments. The first argument, *list*, is a positive integer that becomes the unique name for the display list. Names can be created and reserved with `glGenLists` and tested for uniqueness with `glIsList`. The second argument, *mode*, is a symbolic constant that can assume one of two values:

`GL_COMPILE`
Commands are merely compiled.

`GL_COMPILE_AND_EXECUTE`
Commands are executed as they are compiled into the display list.

Certain commands are not compiled into the display list but are executed immediately, regardless of the display-list mode. These commands are `glAreTexturesResident`, `glColorPointer`, `glDeleteLists`, `glDeleteTextures`, `glDisableClientState`, `glEdgeFlagPointer`, `glEnableClientState`, `glFeedbackBuffer`, `glFinish`, `glFlush`, `glGenLists`, `glGenTextures`, `glIndexPointer`, `glInterleavedArrays`, `glIsEnabled`, `glIsList`, `glIsTexture`, `glNormalPointer`, `glPopClientAttrib`, `glPixelStore`, `glPushClientAttrib`, `glReadPixels`, `glRenderMode`, `glSelectBuffer`, `glTexCoordPointer`, `glVertexPointer`, and all of the `glGet` commands.

Similarly, `glTexImage1D`, `glTexImage2D`, and `glTexImage3D` are executed immediately and not compiled into the display list when their first argument is `GL_PROXY_TEXTURE_1D`, `GL_PROXY_TEXTURE_1D`, or `GL_PROXY_TEXTURE_3D`, respectively.

When the `ARB_imaging` extension is supported, `glHistogram` executes immediately when its argument is `GL_PROXY_HISTOGRAM`. Similarly, `glColorTable` executes immediately when its first argument is `GL_PROXY_COLOR_TABLE`, `GL_PROXY_POST_CONVOLUTION_COLOR_TABLE`, or `GL_PROXY_POST_COLOR_MATRIX_COLOR_TABLE`.

For OpenGL versions 1.3 and greater, or when the `ARB_multitexture` extension is supported, `glClientActiveTexture` is not compiled into display lists, but executed immediately.

When `glEndList` is encountered, the display-list definition is completed by associating the list with the unique name *list* (specified in the `glNewList` command). If a display list with name *list* already exists, it is replaced only when `glEndList` is called.

`GL_INVALID_VALUE` is generated if *list* is 0.

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

`GL_INVALID_OPERATION` is generated if `glEndList` is called without a preceding `glNewList`, or if `glNewList` is called while a display list is being defined.

`GL_INVALID_OPERATION` is generated if `glNewList` or `glEndList` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`GL_OUT_OF_MEMORY` is generated if there is insufficient memory to compile the display list. If the GL version is 1.1 or greater, no change is made to the previous contents of the display list, if any, and no other change is made to the GL state. (It is as if no attempt had been made to create the new display list.)

`void glNormalPointer` *type stride pointer* [Function]

Define an array of normals.

type Specifies the data type of each coordinate in the array. Symbolic constants `GL_BYTE`, `GL_SHORT`, `GL_INT`, `GL_FLOAT`, and `GL_DOUBLE` are accepted. The initial value is `GL_FLOAT`.

stride Specifies the byte offset between consecutive normals. If *stride* is 0, the 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.

`glNormalPointer` specifies the location and data format of an array of normals to use when rendering. *type* specifies the data type of each normal coordinate, and *stride* specifies the byte stride from one normal to the next, allowing vertices and attributes to be packed into a single array or stored in separate arrays. (Single-array storage may be more efficient on some implementations; see `glInterleavedArrays`.)

If a non-zero named buffer object is bound to the `GL_ARRAY_BUFFER` target (see `glBindBuffer`) while a normal array is specified, *pointer* is treated as a byte offset into the buffer object's data store. Also, the buffer object binding (`GL_ARRAY_BUFFER_BINDING`) is saved as normal vertex array client-side state (`GL_NORMAL_ARRAY_BUFFER_BINDING`).

When a normal array is specified, *type*, *stride*, and *pointer* are saved as client-side state, in addition to the current vertex array buffer object binding.

To enable and disable the normal array, call `glEnableClientState` and `glDisableClientState` with the argument `GL_NORMAL_ARRAY`. If enabled, the normal array is used when `glDrawArrays`, `glMultiDrawArrays`, `glDrawElements`, `glMultiDrawElements`, `glDrawRangeElements`, or `glArrayElement` is called.

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

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

`void glNormal3b` *nx ny nz* [Function]
`void glNormal3d` *nx ny nz* [Function]
`void glNormal3f` *nx ny nz* [Function]
`void glNormal3i` *nx ny nz* [Function]
`void glNormal3s` *nx ny nz* [Function]
`void glNormal3bv` *v* [Function]
`void glNormal3dv` *v* [Function]
`void glNormal3fv` *v* [Function]
`void glNormal3iv` *v* [Function]
`void glNormal3sv` *v* [Function]

Set the current normal vector.

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

The current normal is set to the given coordinates whenever `glNormal` is issued. Byte, short, or integer arguments are converted to floating-point format with a linear mapping that maps the most positive representable integer value to 1.0 and the most negative representable integer value to -1.0.

Normals specified with `glNormal` need not have unit length. If `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.

`void glOrtho` *left right bottom top nearVal farVal* [Function]

Multiply the current matrix with an orthographic matrix.

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.

nearVal
farVal Specify the distances to the nearer and farther depth clipping planes. These values are negative if the plane is to be behind the viewer.

`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:

((2/*right-left*,, 0 0 *t_x*,), (0 2/*top-bottom*,, 0 *t_y*,), (0 0 -2/*farVal-nearVal*,, *t_z*,), (0 0 0 1),)

where *t_x*=-*right*+*left*,/*right-left*,,*t_y*=-*top*+*bottom*,/*top-bottom*,,*t_z*=-*farVal*+*nearVal*,/*farVal-nearVal*,,

Typically, the matrix mode is `GL_PROJECTION`, and (*left*,*bottom-nearVal*) and (*right*,*top-nearVal*) 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). -*farVal* specifies the location of the far clipping plane. Both *nearVal* and *farVal* can be either positive or negative.

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

`GL_INVALID_VALUE` is generated if *left* = *right*, or *bottom* = *top*, or *near* = *far*.

`GL_INVALID_OPERATION` is generated if `glOrtho` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glPassThrough` *token* [Function]

Place a marker in the feedback buffer.

token Specifies a marker value to be placed in the feedback buffer following a `GL_PASS_THROUGH_TOKEN`.

Feedback is a GL render mode. The mode is selected by calling `glRenderMode` with `GL_FEEDBACK`. When the GL is in feedback mode, no pixels are produced by rasterization. Instead, information about primitives that would have been rasterized is fed back to the application using the GL. See the `glFeedbackBuffer` reference page for a description of the feedback buffer and the values in it.

`glPassThrough` inserts a user-defined marker in the feedback buffer when it is executed in feedback mode. *token* is returned as if it were a primitive; it is indicated with its own unique identifying value: `GL_PASS_THROUGH_TOKEN`. The order of `glPassThrough` commands with respect to the specification of graphics primitives is maintained.

`GL_INVALID_OPERATION` is generated if `glPassThrough` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glPixelMapfv` *map mapsize values* [Function]
`void glPixelMapuiv` *map mapsize values* [Function]
`void glPixelMapusv` *map mapsize values* [Function]

Set up pixel transfer maps.

map Specifies a symbolic map name. Must be one of the following: `GL_PIXEL_MAP_I_TO_I`, `GL_PIXEL_MAP_S_TO_S`, `GL_PIXEL_MAP_I_TO_R`, `GL_PIXEL_MAP_I_TO_G`, `GL_PIXEL_MAP_I_TO_B`, `GL_PIXEL_MAP_I_TO_A`, `GL_PIXEL_MAP_R_TO_R`, `GL_PIXEL_MAP_G_TO_G`, `GL_PIXEL_MAP_B_TO_B`, or `GL_PIXEL_MAP_A_TO_A`.

mapsize Specifies the size of the map being defined.

values Specifies an array of *mapsize* values.

`glPixelMap` sets up translation tables, or *maps*, used by `glCopyPixels`, `glCopyTexImage1D`, `glCopyTexImage2D`, `glCopyTexSubImage1D`, `glCopyTexSubImage2D`, `glCopyTexSubImage3D`, `glDrawPixels`, `glReadPixels`, `glTexImage1D`, `glTexImage2D`, `glTexImage3D`, `glTexSubImage1D`, `glTexSubImage2D`, and `glTexSubImage3D`. Additionally, if the `ARB_imaging` subset is supported, the routines `glColorTable`, `glColorSubTable`, `glConvolutionFilter1D`, `glConvolutionFilter2D`, `glHistogram`, `glMinmax`, and `glSeparableFilter2D`. Use of these maps is described completely in the `glPixelTransfer` reference page, and partly in the reference pages for the pixel and texture image commands. Only the specification of the maps is described in this reference page.

map is a symbolic map name, indicating one of ten maps to set. *mapsize* specifies the number of entries in the map, and *values* is a pointer to an array of *mapsize* map values.

If a non-zero named buffer object is bound to the `GL_PIXEL_UNPACK_BUFFER` target (see `glBindBuffer`) while a pixel transfer map is specified, *values* is treated as a byte offset into the buffer object's data store.

The ten maps are as follows:

`GL_PIXEL_MAP_I_TO_I`
: Maps color indices to color indices.

`GL_PIXEL_MAP_S_TO_S`
: Maps stencil indices to stencil indices.

`GL_PIXEL_MAP_I_TO_R`
: Maps color indices to red components.

`GL_PIXEL_MAP_I_TO_G`
: Maps color indices to green components.

`GL_PIXEL_MAP_I_TO_B`
: Maps color indices to blue components.

`GL_PIXEL_MAP_I_TO_A`
: Maps color indices to alpha components.

`GL_PIXEL_MAP_R_TO_R`
: Maps red components to red components.

`GL_PIXEL_MAP_G_TO_G`
: Maps green components to green components.

`GL_PIXEL_MAP_B_TO_B`
: Maps blue components to blue components.

`GL_PIXEL_MAP_A_TO_A`
: Maps alpha components to alpha components.

The entries in a map can be specified as single-precision floating-point numbers, unsigned short integers, or unsigned int integers. Maps that store color component values (all but `GL_PIXEL_MAP_I_TO_I` and `GL_PIXEL_MAP_S_TO_S`) retain their values in floating-point format, with unspecified mantissa and exponent sizes. Floating-point

values specified by `glPixelMapfv` are converted directly to the internal floating-point format of these maps, then clamped to the range [0,1]. Unsigned integer values specified by `glPixelMapusv` and `glPixelMapuiv` are converted linearly such that the largest representable integer maps to 1.0, and 0 maps to 0.0.

Maps that store indices, `GL_PIXEL_MAP_I_TO_I` and `GL_PIXEL_MAP_S_TO_S`, retain their values in fixed-point format, with an unspecified number of bits to the right of the binary point. Floating-point values specified by `glPixelMapfv` are converted directly to the internal fixed-point format of these maps. Unsigned integer values specified by `glPixelMapusv` and `glPixelMapuiv` specify integer values, with all 0's to the right of the binary point.

The following table shows the initial sizes and values for each of the maps. Maps that are indexed by either color or stencil indices must have *mapsize* = 2^*n* for some *n* or the results are undefined. The maximum allowable size for each map depends on the implementation and can be determined by calling `glGet` with argument `GL_MAX_PIXEL_MAP_TABLE`. The single maximum applies to all maps; it is at least 32.

map **Lookup Index**, **Lookup Value**, **Initial Size**, **Initial Value**

`GL_PIXEL_MAP_I_TO_I`
color index , color index , 1 , 0

`GL_PIXEL_MAP_S_TO_S`
stencil index , stencil index , 1 , 0

`GL_PIXEL_MAP_I_TO_R`
color index , R , 1 , 0

`GL_PIXEL_MAP_I_TO_G`
color index , G , 1 , 0

`GL_PIXEL_MAP_I_TO_B`
color index , B , 1 , 0

`GL_PIXEL_MAP_I_TO_A`
color index , A , 1 , 0

`GL_PIXEL_MAP_R_TO_R`
R , R , 1 , 0

`GL_PIXEL_MAP_G_TO_G`
G , G , 1 , 0

`GL_PIXEL_MAP_B_TO_B`
B , B , 1 , 0

`GL_PIXEL_MAP_A_TO_A`
A , A , 1 , 0

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

`GL_INVALID_VALUE` is generated if *mapsize* is less than one or larger than `GL_MAX_PIXEL_MAP_TABLE`.

`GL_INVALID_VALUE` is generated if *map* is `GL_PIXEL_MAP_I_TO_I`, `GL_PIXEL_MAP_S_TO_S`, `GL_PIXEL_MAP_I_TO_R`, `GL_PIXEL_MAP_I_TO_G`, `GL_PIXEL_MAP_I_TO_B`, or `GL_PIXEL_MAP_I_TO_A`, and *mapsize* is not a power of two.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and the buffer object's data store is currently mapped.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

GL_INVALID_OPERATION is generated by glPixelMapfv if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and *values* is not evenly divisible into the number of bytes needed to store in memory a GLfloat datum.

GL_INVALID_OPERATION is generated by glPixelMapuiv if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and *values* is not evenly divisible into the number of bytes needed to store in memory a GLuint datum.

GL_INVALID_OPERATION is generated by glPixelMapusv if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and *values* is not evenly divisible into the number of bytes needed to store in memory a GLushort datum.

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

`void glPixelStoref` *pname param* [Function]
`void glPixelStorei` *pname param* [Function]

Set pixel storage modes.

pname Specifies the symbolic name of the parameter to be set. Six values affect the packing of pixel data into memory: GL_PACK_SWAP_BYTES, GL_PACK_LSB_FIRST, GL_PACK_ROW_LENGTH, GL_PACK_IMAGE_HEIGHT, GL_PACK_SKIP_PIXELS, GL_PACK_SKIP_ROWS, GL_PACK_SKIP_IMAGES, and GL_PACK_ALIGNMENT. Six more affect the unpacking of pixel data *from* memory: GL_UNPACK_SWAP_BYTES, GL_UNPACK_LSB_FIRST, GL_UNPACK_ROW_LENGTH, GL_UNPACK_IMAGE_HEIGHT, GL_UNPACK_SKIP_PIXELS, GL_UNPACK_SKIP_ROWS, GL_UNPACK_SKIP_IMAGES, and GL_UNPACK_ALIGNMENT.

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

glPixelStore sets pixel storage modes that affect the operation of subsequent glDrawPixels and glReadPixels as well as the unpacking of polygon stipple patterns (see glPolygonStipple), bitmaps (see glBitmap), texture patterns (see glTexImage1D, glTexImage2D, glTexImage3D, glTexSubImage1D, glTexSubImage2D, glTexSubImage3D). Additionally, if the ARB_imaging extension is supported, pixel storage modes affect convolution filters (see glConvolutionFilter1D, glConvolutionFilter2D, and glSeparableFilter2D, color table (see glColorTable, and glColorSubTable, and unpacking histogram (See glHistogram), and minmax (See glMinmax) data.

pname is a symbolic constant indicating the parameter to be set, and *param* is the new value. Six of the twelve storage parameters affect how pixel data is returned to client memory. They are as follows:

`GL_PACK_SWAP_BYTES`

If true, byte ordering for multibyte color components, depth components, color indices, or stencil indices is reversed. That is, if a four-byte component consists of bytes b_0, b_1, b_2, b_3, it is stored in memory as b_3, b_2, b_1, b_0 if `GL_PACK_SWAP_BYTES` is true. `GL_PACK_SWAP_BYTES` has no effect on the memory order of components within a pixel, only on the order of bytes within components or indices. For example, the three components of a `GL_RGB` format pixel are always stored with red first, green second, and blue third, regardless of the value of `GL_PACK_SWAP_BYTES`.

`GL_PACK_LSB_FIRST`

If true, bits are ordered within a byte from least significant to most significant; otherwise, the first bit in each byte is the most significant one. This parameter is significant for bitmap data only.

`GL_PACK_ROW_LENGTH`

If greater than 0, `GL_PACK_ROW_LENGTH` defines the number of pixels in a row. If the first pixel of a row is placed at location p in memory, then the location of the first pixel of the next row is obtained by skipping

$k=\{(nl),\ (a/s,snl,/a,)(s>=a),\ (s<a),$

components or indices, where n is the number of components or indices in a pixel, l is the number of pixels in a row (`GL_PACK_ROW_LENGTH` if it is greater than 0, the *width* argument to the pixel routine otherwise), a is the value of `GL_PACK_ALIGNMENT`, and s is the size, in bytes, of a single component (if $a<s$, then it is as if $a=s$). In the case of 1-bit values, the location of the next row is obtained by skipping

$k=8anl,/8a,,$

components or indices.

The word *component* in this description refers to the nonindex values red, green, blue, alpha, and depth. Storage format `GL_RGB`, for example, has three components per pixel: first red, then green, and finally blue.

`GL_PACK_IMAGE_HEIGHT`

If greater than 0, `GL_PACK_IMAGE_HEIGHT` defines the number of pixels in an image three-dimensional texture volume, where "image" is defined by all pixels sharing the same third dimension index. If the first pixel of a row is placed at location p in memory, then the location of the first pixel of the next row is obtained by skipping

$k=\{(nlh),\ (a/s,snlh,/a,)(s>=a),\ (s<a),$

components or indices, where n is the number of components or indices in a pixel, l is the number of pixels in a row (`GL_PACK_ROW_LENGTH` if it is greater than 0, the *width* argument to `glTexImage3D` otherwise), h is the number of rows in a pixel image (`GL_PACK_IMAGE_HEIGHT` if it is greater than 0, the *height* argument to the `glTexImage3D` routine otherwise), a is the value of `GL_PACK_ALIGNMENT`, and s is the size, in bytes, of a single component (if $a<s$, then it is as if $a=s$).

The word *component* in this description refers to the nonindex values red, green, blue, alpha, and depth. Storage format `GL_RGB`, for example, has three components per pixel: first red, then green, and finally blue.

`GL_PACK_SKIP_PIXELS`, `GL_PACK_SKIP_ROWS`, and `GL_PACK_SKIP_IMAGES`
: These values are provided as a convenience to the programmer; they provide no functionality that cannot be duplicated simply by incrementing the pointer passed to `glReadPixels`. Setting `GL_PACK_SKIP_PIXELS` to i is equivalent to incrementing the pointer by in components or indices, where n is the number of components or indices in each pixel. Setting `GL_PACK_SKIP_ROWS` to j is equivalent to incrementing the pointer by jm components or indices, where m is the number of components or indices per row, as just computed in the `GL_PACK_ROW_LENGTH` section. Setting `GL_PACK_SKIP_IMAGES` to k is equivalent to incrementing the pointer by kp, where p is the number of components or indices per image, as computed in the `GL_PACK_IMAGE_HEIGHT` section.

`GL_PACK_ALIGNMENT`
: Specifies the alignment requirements for the start of each pixel row in memory. The allowable values are 1 (byte-alignment), 2 (rows aligned to even-numbered bytes), 4 (word-alignment), and 8 (rows start on double-word boundaries).

The other six of the twelve storage parameters affect how pixel data is read from client memory. These values are significant for `glDrawPixels`, `glTexImage1D`, `glTexImage2D`, `glTexImage3D`, `glTexSubImage1D`, `glTexSubImage2D`, `glTexSubImage3D`, `glBitmap`, and `glPolygonStipple`.

Additionally, if the `ARB_imaging` extension is supported, `glColorTable`, `glColorSubTable`, `glConvolutionFilter1D`, `glConvolutionFilter2D`, and `glSeparableFilter2D`. They are as follows:

`GL_UNPACK_SWAP_BYTES`
: If true, byte ordering for multibyte color components, depth components, color indices, or stencil indices is reversed. That is, if a four-byte component consists of bytes b_0, b_1, b_2, b_3, it is taken from memory as b_3, b_2, b_1, b_0 if `GL_UNPACK_SWAP_BYTES` is true. `GL_UNPACK_SWAP_BYTES` has no effect on the memory order of components within a pixel, only on the order of bytes within components or indices. For example, the three components of a `GL_RGB` format pixel are always stored with red first, green second, and blue third, regardless of the value of `GL_UNPACK_SWAP_BYTES`.

`GL_UNPACK_LSB_FIRST`
: If true, bits are ordered within a byte from least significant to most significant; otherwise, the first bit in each byte is the most significant one. This is relevant only for bitmap data.

`GL_UNPACK_ROW_LENGTH`
: If greater than 0, `GL_UNPACK_ROW_LENGTH` defines the number of pixels in a row. If the first pixel of a row is placed at location p in memory, then the location of the first pixel of the next row is obtained by skipping

k={(nl), (a/s,snl,/a,)(s>=a), (s<a),

components or indices, where n is the number of components or indices in a pixel, l is the number of pixels in a row (GL_UNPACK_ROW_LENGTH if it is greater than 0, the *width* argument to the pixel routine otherwise), a is the value of GL_UNPACK_ALIGNMENT, and s is the size, in bytes, of a single component (if a<s, then it is as if a=s). In the case of 1-bit values, the location of the next row is obtained by skipping

k=8anl,/8a,,

components or indices.

The word *component* in this description refers to the nonindex values red, green, blue, alpha, and depth. Storage format GL_RGB, for example, has three components per pixel: first red, then green, and finally blue.

GL_UNPACK_IMAGE_HEIGHT

If greater than 0, GL_UNPACK_IMAGE_HEIGHT defines the number of pixels in an image of a three-dimensional texture volume. Where "image" is defined by all pixel sharing the same third dimension index. If the first pixel of a row is placed at location p in memory, then the location of the first pixel of the next row is obtained by skipping

k={(nlh), (a/s,$snlh$,/a,)(s>=a), (s<a),

components or indices, where n is the number of components or indices in a pixel, l is the number of pixels in a row (GL_UNPACK_ROW_LENGTH if it is greater than 0, the *width* argument to glTexImage3D otherwise), h is the number of rows in an image (GL_UNPACK_IMAGE_HEIGHT if it is greater than 0, the *height* argument to glTexImage3D otherwise), a is the value of GL_UNPACK_ALIGNMENT, and s is the size, in bytes, of a single component (if a<s, then it is as if a=s).

The word *component* in this description refers to the nonindex values red, green, blue, alpha, and depth. Storage format GL_RGB, for example, has three components per pixel: first red, then green, and finally blue.

GL_UNPACK_SKIP_PIXELS and GL_UNPACK_SKIP_ROWS

These values are provided as a convenience to the programmer; they provide no functionality that cannot be duplicated by incrementing the pointer passed to glDrawPixels, glTexImage1D, glTexImage2D, glTexSubImage1D, glTexSubImage2D, glBitmap, or glPolygonStipple. Setting GL_UNPACK_SKIP_PIXELS to i is equivalent to incrementing the pointer by in components or indices, where n is the number of components or indices in each pixel. Setting GL_UNPACK_SKIP_ROWS to j is equivalent to incrementing the pointer by jk components or indices, where k is the number of components or indices per row, as just computed in the GL_UNPACK_ROW_LENGTH section.

GL_UNPACK_ALIGNMENT

Specifies the alignment requirements for the start of each pixel row in memory. The allowable values are 1 (byte-alignment), 2 (rows aligned to even-numbered bytes), 4 (word-alignment), and 8 (rows start on double-word boundaries).

The following table gives the type, initial value, and range of valid values for each storage parameter that can be set with `glPixelStore`.

pname **Type**, **Initial Value**, **Valid Range**

`GL_PACK_SWAP_BYTES`
boolean , false , true or false

`GL_PACK_LSB_FIRST`
boolean , false , true or false

`GL_PACK_ROW_LENGTH`
integer , 0 , [0,)

`GL_PACK_IMAGE_HEIGHT`
integer , 0 , [0,)

`GL_PACK_SKIP_ROWS`
integer , 0 , [0,)

`GL_PACK_SKIP_PIXELS`
integer , 0 , [0,)

`GL_PACK_SKIP_IMAGES`
integer , 0 , [0,)

`GL_PACK_ALIGNMENT`
integer , 4 , 1, 2, 4, or 8

`GL_UNPACK_SWAP_BYTES`
boolean , false , true or false

`GL_UNPACK_LSB_FIRST`
boolean , false , true or false

`GL_UNPACK_ROW_LENGTH`
integer , 0 , [0,)

`GL_UNPACK_IMAGE_HEIGHT`
integer , 0 , [0,)

`GL_UNPACK_SKIP_ROWS`
integer , 0 , [0,)

`GL_UNPACK_SKIP_PIXELS`
integer , 0 , [0,)

`GL_UNPACK_SKIP_IMAGES`
integer , 0 , [0,)

`GL_UNPACK_ALIGNMENT`
integer , 4 , 1, 2, 4, or 8

`glPixelStoref` can be used to set any pixel store parameter. If the parameter type is boolean, then if *param* is 0, the parameter is false; otherwise it is set to true. If *pname* is a integer type parameter, *param* is rounded to the nearest integer.

Likewise, `glPixelStorei` can also be used to set any of the pixel store parameters. Boolean parameters are set to false if *param* is 0 and true otherwise.

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

`GL_INVALID_VALUE` is generated if a negative row length, pixel skip, or row skip value is specified, or if alignment is specified as other than 1, 2, 4, or 8.

`GL_INVALID_OPERATION` is generated if `glPixelStore` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glPixelTransferf` *pname param* [Function]
`void glPixelTransferi` *pname param* [Function]
Set pixel transfer modes.

pname Specifies the symbolic name of the pixel transfer parameter to be set. Must be one of the following: `GL_MAP_COLOR`, `GL_MAP_STENCIL`, `GL_INDEX_SHIFT`, `GL_INDEX_OFFSET`, `GL_RED_SCALE`, `GL_RED_BIAS`, `GL_GREEN_SCALE`, `GL_GREEN_BIAS`, `GL_BLUE_SCALE`, `GL_BLUE_BIAS`, `GL_ALPHA_SCALE`, `GL_ALPHA_BIAS`, `GL_DEPTH_SCALE`, or `GL_DEPTH_BIAS`.

Additionally, if the `ARB_imaging` extension is supported, the following symbolic names are accepted: `GL_POST_COLOR_MATRIX_RED_SCALE`, `GL_POST_COLOR_MATRIX_GREEN_SCALE`, `GL_POST_COLOR_MATRIX_BLUE_SCALE`, `GL_POST_COLOR_MATRIX_ALPHA_SCALE`, `GL_POST_COLOR_MATRIX_RED_BIAS`, `GL_POST_COLOR_MATRIX_GREEN_BIAS`, `GL_POST_COLOR_MATRIX_BLUE_BIAS`, `GL_POST_COLOR_MATRIX_ALPHA_BIAS`, `GL_POST_CONVOLUTION_RED_SCALE`, `GL_POST_CONVOLUTION_GREEN_SCALE`, `GL_POST_CONVOLUTION_BLUE_SCALE`, `GL_POST_CONVOLUTION_ALPHA_SCALE`, `GL_POST_CONVOLUTION_RED_BIAS`, `GL_POST_CONVOLUTION_GREEN_BIAS`, `GL_POST_CONVOLUTION_BLUE_BIAS`, and `GL_POST_CONVOLUTION_ALPHA_BIAS`.

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

`glPixelTransfer` sets pixel transfer modes that affect the operation of subsequent `glCopyPixels`, `glCopyTexImage1D`, `glCopyTexImage2D`, `glCopyTexSubImage1D`, `glCopyTexSubImage2D`, `glCopyTexSubImage3D`, `glDrawPixels`, `glReadPixels`, `glTexImage1D`, `glTexImage2D`, `glTexImage3D`, `glTexSubImage1D`, `glTexSubImage2D`, and `glTexSubImage3D` commands. Additionally, if the `ARB_imaging` subset is supported, the routines `glColorTable`, `glColorSubTable`, `glConvolutionFilter1D`, `glConvolutionFilter2D`, `glHistogram`, `glMinmax`, and `glSeparableFilter2D` are also affected. The algorithms that are specified by pixel transfer modes operate on pixels after they are read from the frame buffer (`glCopyPixelsglCopyTexImage1D`, `glCopyTexImage2D`, `glCopyTexSubImage1D`, `glCopyTexSubImage2D`, `glCopyTexSubImage3D`, and `glReadPixels`), or unpacked from client memory (`glDrawPixels`, `glTexImage1D`, `glTexImage2D`, `glTexImage3D`, `glTexSubImage1D`, `glTexSubImage2D`, and `glTexSubImage3D`). Pixel transfer operations happen in the same order, and in the same manner, regardless of the command that resulted in the pixel operation. Pixel storage modes (see `glPixelStore`) control the unpacking of pixels being read from client memory and the packing of pixels being written back into client memory.

Pixel transfer operations handle four fundamental pixel types: *color*, *color index*, *depth*, and *stencil*. *Color* pixels consist of four floating-point values with unspecified mantissa and exponent sizes, scaled such that 0 represents zero intensity and 1 represents full intensity. *Color indices* comprise a single fixed-point value, with unspecified precision to the right of the binary point. *Depth* pixels comprise a single floating-point value, with unspecified mantissa and exponent sizes, scaled such that 0.0 represents the minimum depth buffer value, and 1.0 represents the maximum depth buffer value. Finally, *stencil* pixels comprise a single fixed-point value, with unspecified precision to the right of the binary point.

The pixel transfer operations performed on the four basic pixel types are as follows:

Color Each of the four color components is multiplied by a scale factor, then added to a bias factor. That is, the red component is multiplied by `GL_RED_SCALE`, then added to `GL_RED_BIAS`; the green component is multiplied by `GL_GREEN_SCALE`, then added to `GL_GREEN_BIAS`; the blue component is multiplied by `GL_BLUE_SCALE`, then added to `GL_BLUE_BIAS`; and the alpha component is multiplied by `GL_ALPHA_SCALE`, then added to `GL_ALPHA_BIAS`. After all four color components are scaled and biased, each is clamped to the range [0,1]. All color, scale, and bias values are specified with `glPixelTransfer`.

If `GL_MAP_COLOR` is true, each color component is scaled by the size of the corresponding color-to-color map, then replaced by the contents of that map indexed by the scaled component. That is, the red component is scaled by `GL_PIXEL_MAP_R_TO_R_SIZE`, then replaced by the contents of `GL_PIXEL_MAP_R_TO_R` indexed by itself. The green component is scaled by `GL_PIXEL_MAP_G_TO_G_SIZE`, then replaced by the contents of `GL_PIXEL_MAP_G_TO_G` indexed by itself. The blue component is scaled by `GL_PIXEL_MAP_B_TO_B_SIZE`, then replaced by the contents of `GL_PIXEL_MAP_B_TO_B` indexed by itself. And the alpha component is scaled by `GL_PIXEL_MAP_A_TO_A_SIZE`, then replaced by the contents of `GL_PIXEL_MAP_A_TO_A` indexed by itself. All components taken from the maps are then clamped to the range [0,1]. `GL_MAP_COLOR` is specified with `glPixelTransfer`. The contents of the various maps are specified with `glPixelMap`.

If the `ARB_imaging` extension is supported, each of the four color components may be scaled and biased after transformation by the color matrix. That is, the red component is multiplied by `GL_POST_COLOR_MATRIX_RED_SCALE`, then added to `GL_POST_COLOR_MATRIX_RED_BIAS`; the green component is multiplied by `GL_POST_COLOR_MATRIX_GREEN_SCALE`, then added to `GL_POST_COLOR_MATRIX_GREEN_BIAS`; the blue component is multiplied by `GL_POST_COLOR_MATRIX_BLUE_SCALE`, then added to `GL_POST_COLOR_MATRIX_BLUE_BIAS`; and the alpha component is multiplied by `GL_POST_COLOR_MATRIX_ALPHA_SCALE`, then added to `GL_POST_COLOR_MATRIX_ALPHA_BIAS`. After all four color components are scaled and biased, each is clamped to the range [0,1].

Similarly, if the `ARB_imaging` extension is supported, each of the four color components may be scaled and biased after processing

by the enabled convolution filter. That is, the red component is multiplied by GL_POST_CONVOLUTION_RED_SCALE, then added to GL_POST_CONVOLUTION_RED_BIAS; the green component is multiplied by GL_POST_CONVOLUTION_GREEN_SCALE, then added to GL_POST_CONVOLUTION_GREEN_BIAS; the blue component is multiplied by GL_POST_CONVOLUTION_BLUE_SCALE, then added to GL_POST_CONVOLUTION_BLUE_BIAS; and the alpha component is multiplied by GL_POST_CONVOLUTION_ALPHA_SCALE, then added to GL_POST_CONVOLUTION_ALPHA_BIAS. After all four color components are scaled and biased, each is clamped to the range [0,1].

Color index

Each color index is shifted left by GL_INDEX_SHIFT bits; any bits beyond the number of fraction bits carried by the fixed-point index are filled with zeros. If GL_INDEX_SHIFT is negative, the shift is to the right, again zero filled. Then GL_INDEX_OFFSET is added to the index. GL_INDEX_SHIFT and GL_INDEX_OFFSET are specified with glPixelTransfer.

From this point, operation diverges depending on the required format of the resulting pixels. If the resulting pixels are to be written to a color index buffer, or if they are being read back to client memory in GL_COLOR_INDEX format, the pixels continue to be treated as indices. If GL_MAP_COLOR is true, each index is masked by 2^*n*-1, where *n* is GL_PIXEL_MAP_I_TO_I_SIZE, then replaced by the contents of GL_PIXEL_MAP_I_TO_I indexed by the masked value. GL_MAP_COLOR is specified with glPixelTransfer. The contents of the index map is specified with glPixelMap.

If the resulting pixels are to be written to an RGBA color buffer, or if they are read back to client memory in a format other than GL_COLOR_INDEX, the pixels are converted from indices to colors by referencing the four maps GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, and GL_PIXEL_MAP_I_TO_A. Before being dereferenced, the index is masked by 2^*n*-1, where *n* is GL_PIXEL_MAP_I_TO_R_SIZE for the red map, GL_PIXEL_MAP_I_TO_G_SIZE for the green map, GL_PIXEL_MAP_I_TO_B_SIZE for the blue map, and GL_PIXEL_MAP_I_TO_A_SIZE for the alpha map. All components taken from the maps are then clamped to the range [0,1]. The contents of the four maps is specified with glPixelMap.

Depth Each depth value is multiplied by GL_DEPTH_SCALE, added to GL_DEPTH_BIAS, then clamped to the range [0,1].

Stencil Each index is shifted GL_INDEX_SHIFT bits just as a color index is, then added to GL_INDEX_OFFSET. If GL_MAP_STENCIL is true, each index is masked by 2^*n*-1, where *n* is GL_PIXEL_MAP_S_TO_S_SIZE, then replaced by the contents of GL_PIXEL_MAP_S_TO_S indexed by the masked value.

The following table gives the type, initial value, and range of valid values for each of the pixel transfer parameters that are set with glPixelTransfer.

pname **Type**, **Initial Value**, **Valid Range**

`GL_MAP_COLOR`
boolean , false , true/false

`GL_MAP_STENCIL`
boolean , false , true/false

`GL_INDEX_SHIFT`
integer , 0 , (-,)

`GL_INDEX_OFFSET`
integer , 0 , (-,)

`GL_RED_SCALE`
float , 1 , (-,)

`GL_GREEN_SCALE`
float , 1 , (-,)

`GL_BLUE_SCALE`
float , 1 , (-,)

`GL_ALPHA_SCALE`
float , 1 , (-,)

`GL_DEPTH_SCALE`
float , 1 , (-,)

`GL_RED_BIAS`
float , 0 , (-,)

`GL_GREEN_BIAS`
float , 0 , (-,)

`GL_BLUE_BIAS`
float , 0 , (-,)

`GL_ALPHA_BIAS`
float , 0 , (-,)

`GL_DEPTH_BIAS`
float , 0 , (-,)

`GL_POST_COLOR_MATRIX_RED_SCALE`
float , 1 , (-,)

`GL_POST_COLOR_MATRIX_GREEN_SCALE`
float , 1 , (-,)

`GL_POST_COLOR_MATRIX_BLUE_SCALE`
float , 1 , (-,)

`GL_POST_COLOR_MATRIX_ALPHA_SCALE`
float , 1 , (-,)

`GL_POST_COLOR_MATRIX_RED_BIAS`
float , 0 , (-,)

`GL_POST_COLOR_MATRIX_GREEN_BIAS`
float , 0 , (-,)

`GL_POST_COLOR_MATRIX_BLUE_BIAS`
float , 0 , (-,)

`GL_POST_COLOR_MATRIX_ALPHA_BIAS`
float , 0 , (-,)

`GL_POST_CONVOLUTION_RED_SCALE`
float , 1 , (-,)

`GL_POST_CONVOLUTION_GREEN_SCALE`
float , 1 , (-,)

`GL_POST_CONVOLUTION_BLUE_SCALE`
float , 1 , (-,)

`GL_POST_CONVOLUTION_ALPHA_SCALE`
float , 1 , (-,)

`GL_POST_CONVOLUTION_RED_BIAS`
float , 0 , (-,)

`GL_POST_CONVOLUTION_GREEN_BIAS`
float , 0 , (-,)

`GL_POST_CONVOLUTION_BLUE_BIAS`
float , 0 , (-,)

`GL_POST_CONVOLUTION_ALPHA_BIAS`
float , 0 , (-,)

`glPixelTransferf` can be used to set any pixel transfer parameter. If the parameter type is boolean, 0 implies false and any other value implies true. If *pname* is an integer parameter, *param* is rounded to the nearest integer.

Likewise, `glPixelTransferi` can be used to set any of the pixel transfer parameters. Boolean parameters are set to false if *param* is 0 and to true otherwise. *param* is converted to floating point before being assigned to real-valued parameters.

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

`GL_INVALID_OPERATION` is generated if `glPixelTransfer` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glPixelZoom` *xfactor yfactor* [Function]

Specify the pixel zoom factors.

xfactor
yfactor Specify the x and y zoom factors for pixel write operations.

`glPixelZoom` specifies values for the x and y zoom factors. During the execution of `glDrawPixels` or `glCopyPixels`, if (xr, yr) is the current raster position, and a given element is in the mth row and nth column of the pixel rectangle, then pixels whose centers are in the rectangle with corners at

$(xr+nxfactor, yr+myfactor)$

(xr+(n+1,)$xfactor$, yr+(m+1,)$yfactor$)

are candidates for replacement. Any pixel whose center lies on the bottom or left edge of this rectangular region is also modified.

Pixel zoom factors are not limited to positive values. Negative zoom factors reflect the resulting image about the current raster position.

`GL_INVALID_OPERATION` is generated if `glPixelZoom` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glPointParameterf` *pname param* [Function]
`void glPointParameteri` *pname param* [Function]
`void glPointParameterfv` *pname params* [Function]
`void glPointParameteriv` *pname params* [Function]

Specify point parameters.

pname Specifies a single-valued point parameter. `GL_POINT_SIZE_MIN`, `GL_POINT_SIZE_MAX`, `GL_POINT_FADE_THRESHOLD_SIZE`, and `GL_POINT_SPRITE_COORD_ORIGIN` are accepted.

param Specifies the value that *pname* will be set to.

The following values are accepted for *pname*:

`GL_POINT_SIZE_MIN`
params is a single floating-point value that specifies the minimum point size. The default value is 0.0.

`GL_POINT_SIZE_MAX`
params is a single floating-point value that specifies the maximum point size. The default value is 1.0.

`GL_POINT_FADE_THRESHOLD_SIZE`
params is a single floating-point value that specifies the threshold value to which point sizes are clamped if they exceed the specified value. The default value is 1.0.

`GL_POINT_DISTANCE_ATTENUATION`
params is an array of three floating-point values that specify the coefficients used for scaling the computed point size. The default values are (1,00).

`GL_POINT_SPRITE_COORD_ORIGIN`
params is a single enum specifying the point sprite texture coordinate origin, either `GL_LOWER_LEFT` or `GL_UPPER_LEFT`. The default value is `GL_UPPER_LEFT`.

`GL_INVALID_VALUE` is generated If the value specified for `GL_POINT_SIZE_MIN`, `GL_POINT_SIZE_MAX`, or `GL_POINT_FADE_THRESHOLD_SIZE` is less than zero.

`GL_INVALID_ENUM` is generated If the value specified for `GL_POINT_SPRITE_COORD_ORIGIN` is not `GL_LOWER_LEFT` or `GL_UPPER_LEFT`.

If the value for `GL_POINT_SIZE_MIN` is greater than `GL_POINT_SIZE_MAX`, the point size after clamping is undefined, but no error is generated.

`void glPointSize` *size* [Function]

Specify the diameter of rasterized points.

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

`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.

The specified point size is multiplied with a distance attenuation factor and clamped to the specified point size range, and further clamped to the implementation-dependent point size range to produce the derived point size using

pointSize=*clamp*(*size*(1/*a*+*bd*+*cd*^2,,),,)

where d is the eye-coordinate distance from the eye to the vertex, and a, b, and c are the distance attenuation coefficients (see `glPointParameter`).

If multisampling is disabled, the computed point size is used as the point's width.

If multisampling is enabled, the point may be faded by modifying the point alpha value (see `glSampleCoverage`) instead of allowing the point width to go below a given threshold (see `glPointParameter`). In this case, the width is further modified in the following manner:

pointWidth={(*pointSize*), (*threshold*)(*pointSize*>=*threshold*), (*otherwise*),

The point alpha value is modified by computing:

pointAlpha={(1), ((*pointSize*/*threshold*,)^2)(*pointSize*>=*threshold*), (*otherwise*),

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

(x_w,+.5,y_w,+.5)

where w subscripts indicate window coordinates. All pixels that lie within the square grid of the rounded size centered at (x, y) make up the fragment. If the size is even, the center point is

(x_w+.5,,y_w+.5,)

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

If antialiasing is enabled, 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 and the size difference between supported sizes within the range, call `glGet` with arguments `GL_SMOOTH_POINT_SIZE_RANGE` and `GL_SMOOTH_POINT_SIZE_GRANULARITY`. For aliased points, query the supported ranges and granularity with `glGet` with arguments `GL_ALIASED_POINT_SIZE_RANGE`.

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

`GL_INVALID_OPERATION` is generated if `glPointSize` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glPolygonMode` *face mode* [Function]

Select a polygon rasterization mode.

face Specifies the polygons that *mode* applies to. Must be `GL_FRONT` for front-facing polygons, `GL_BACK` for back-facing polygons, or `GL_FRONT_AND_BACK` for front- and back-facing polygons.

mode Specifies how polygons will be rasterized. Accepted values are `GL_POINT`, `GL_LINE`, and `GL_FILL`. The initial value is `GL_FILL` for both front- and back-facing polygons.

`glPolygonMode` controls the interpretation of polygons for rasterization. *face* describes which polygons *mode* applies to: front-facing polygons (`GL_FRONT`), back-facing polygons (`GL_BACK`), or both (`GL_FRONT_AND_BACK`). The polygon mode affects only the final rasterization of polygons. In particular, a polygon's vertices are lit and the polygon is clipped and possibly culled before these modes are applied.

Three modes are defined and can be specified in *mode*:

`GL_POINT` Polygon vertices that are marked as the start of a boundary edge are drawn as points. Point attributes such as `GL_POINT_SIZE` and `GL_POINT_SMOOTH` control the rasterization of the points. Polygon rasterization attributes other than `GL_POLYGON_MODE` have no effect.

`GL_LINE` Boundary edges of the polygon are drawn as line segments. They are treated as connected line segments for line stippling; the line stipple counter and pattern are not reset between segments (see `glLineStipple`). Line attributes such as `GL_LINE_WIDTH` and `GL_LINE_SMOOTH` control the rasterization of the lines. Polygon rasterization attributes other than `GL_POLYGON_MODE` have no effect.

`GL_FILL` The interior of the polygon is filled. Polygon attributes such as `GL_POLYGON_STIPPLE` and `GL_POLYGON_SMOOTH` control the rasterization of the polygon.

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

`GL_INVALID_OPERATION` is generated if `glPolygonMode` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glPolygonOffset` *factor units* [Function]

Set the scale and units used to calculate depth values.

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.

When `GL_POLYGON_OFFSET_FILL`, `GL_POLYGON_OFFSET_LINE`, or `GL_POLYGON_OFFSET_POINT` 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 *factorDZ+runits*, where *DZ* is a measurement of the change in depth relative to the screen area of the polygon, and *r* is the smallest value that is guaranteed to produce a resolvable offset for a given implementation. The offset is added before the depth test is performed and before the value is written into the depth buffer.

`glPolygonOffset` is useful for rendering hidden-line images, for applying decals to surfaces, and for rendering solids with highlighted edges.

`GL_INVALID_OPERATION` is generated if `glPolygonOffset` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glPolygonStipple` *pattern* [Function]

Set the polygon stippling pattern.

pattern Specifies a pointer to a 3232 stipple pattern that will be unpacked from memory in the same way that `glDrawPixels` unpacks pixels.

Polygon stippling, like line stippling (see `glLineStipple`), masks out certain fragments produced by rasterization, creating a pattern. Stippling is independent of polygon antialiasing.

pattern is a pointer to a 3232 stipple pattern that is stored in memory just like the pixel data supplied to a `glDrawPixels` call with height and *width* both equal to 32, a pixel format of `GL_COLOR_INDEX`, and data type of `GL_BITMAP`. That is, the stipple pattern is represented as a 3232 array of 1-bit color indices packed in unsigned bytes. `glPixelStore` parameters like `GL_UNPACK_SWAP_BYTES` and `GL_UNPACK_LSB_FIRST` affect the assembling of the bits into a stipple pattern. Pixel transfer operations (shift, offset, pixel map) are not applied to the stipple image, however.

If a non-zero named buffer object is bound to the `GL_PIXEL_UNPACK_BUFFER` target (see `glBindBuffer`) while a stipple pattern is specified, *pattern* is treated as a byte offset into the buffer object's data store.

To enable and disable polygon stippling, call `glEnable` and `glDisable` with argument `GL_POLYGON_STIPPLE`. Polygon stippling is initially disabled. If it's enabled, a rasterized polygon fragment with window coordinates x_w and y_w is sent to the next stage of the GL if and only if the (x_w%32)th bit in the (y_w%32)th row of the stipple pattern is 1 (one). When polygon stippling is disabled, it is as if the stipple pattern consists of all 1's.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the buffer object's data store is currently mapped.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

`GL_INVALID_OPERATION` is generated if `glPolygonStipple` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glPrioritizeTextures` *n textures priorities* [Function]

Set texture residence priority.

n Specifies the number of textures to be prioritized.

textures Specifies an array containing the names of the textures to be prioritized.

priorities Specifies an array containing the texture priorities. A priority given in an element of *priorities* applies to the texture named by the corresponding element of *textures*.

`glPrioritizeTextures` assigns the *n* texture priorities given in *priorities* to the *n* textures named in *textures*.

The GL establishes a "working set" of textures that are resident in texture memory. These textures may be bound to a texture target much more efficiently than textures that are not resident. By specifying a priority for each texture, `glPrioritizeTextures` allows applications to guide the GL implementation in determining which textures should be resident.

The priorities given in *priorities* are clamped to the range [0,1] before they are assigned. 0 indicates the lowest priority; textures with priority 0 are least likely to be resident. 1 indicates the highest priority; textures with priority 1 are most likely to be resident. However, textures are not guaranteed to be resident until they are used.

`glPrioritizeTextures` silently ignores attempts to prioritize texture 0 or any texture name that does not correspond to an existing texture.

`glPrioritizeTextures` does not require that any of the textures named by *textures* be bound to a texture target. `glTexParameter` may also be used to set a texture's priority, but only if the texture is currently bound. This is the only way to set the priority of a default texture.

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

`GL_INVALID_OPERATION` is generated if `glPrioritizeTextures` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glPushAttrib` *mask* [Function]
`void glPopAttrib` [Function]

Push and pop the server attribute stack.

mask Specifies a mask that indicates which attributes to save. Values for *mask* are listed below.

`glPushAttrib` takes one argument, a mask that indicates which groups of state variables to save on the attribute stack. Symbolic constants are used to set bits in the mask. *mask* is typically constructed by specifying the bitwise-or of several of these constants together. The special mask `GL_ALL_ATTRIB_BITS` can be used to save all stackable states.

The symbolic mask constants and their associated GL state are as follows (the second column lists which attributes are saved):

`GL_ACCUM_BUFFER_BIT`
Accumulation buffer clear value

`GL_COLOR_BUFFER_BIT`
- `GL_ALPHA_TEST` enable bit
- Alpha test function and reference value
- `GL_BLEND` enable bit
- Blending source and destination functions
- Constant blend color
- Blending equation
- `GL_DITHER` enable bit
- `GL_DRAW_BUFFER` setting
- `GL_COLOR_LOGIC_OP` enable bit
- `GL_INDEX_LOGIC_OP` enable bit
- Logic op function
- Color mode and index mode clear values
- Color mode and index mode writemasks

`GL_CURRENT_BIT`
- Current RGBA color
- Current color index
- Current normal vector
- Current texture coordinates
- Current raster position
- `GL_CURRENT_RASTER_POSITION_VALID` flag
- RGBA color associated with current raster position
- Color index associated with current raster position
- Texture coordinates associated with current raster position
- `GL_EDGE_FLAG` flag

`GL_DEPTH_BUFFER_BIT`
- `GL_DEPTH_TEST` enable bit
- Depth buffer test function
- Depth buffer clear value
- `GL_DEPTH_WRITEMASK` enable bit

`GL_ENABLE_BIT`
- `GL_ALPHA_TEST` flag
- `GL_AUTO_NORMAL` flag
- `GL_BLEND` flag

. Enable bits for the user-definable clipping planes

. `GL_COLOR_MATERIAL`

. `GL_CULL_FACE` flag

. `GL_DEPTH_TEST` flag

. `GL_DITHER` flag

. `GL_FOG` flag

. `GL_LIGHT`*i* where `0` <= *i* < `GL_MAX_LIGHTS`

. `GL_LIGHTING` flag

. `GL_LINE_SMOOTH` flag

. `GL_LINE_STIPPLE` flag

. `GL_COLOR_LOGIC_OP` flag

. `GL_INDEX_LOGIC_OP` flag

. `GL_MAP1_`*x* where *x* is a map type

. `GL_MAP2_`*x* where *x* is a map type

. `GL_MULTISAMPLE` flag

. `GL_NORMALIZE` flag

. `GL_POINT_SMOOTH` flag

. `GL_POLYGON_OFFSET_LINE` flag

. `GL_POLYGON_OFFSET_FILL` flag

. `GL_POLYGON_OFFSET_POINT` flag

. `GL_POLYGON_SMOOTH` flag

. `GL_POLYGON_STIPPLE` flag

. `GL_SAMPLE_ALPHA_TO_COVERAGE` flag

. `GL_SAMPLE_ALPHA_TO_ONE` flag

. `GL_SAMPLE_COVERAGE` flag

. `GL_SCISSOR_TEST` flag

. `GL_STENCIL_TEST` flag

. `GL_TEXTURE_1D` flag

. `GL_TEXTURE_2D` flag

. `GL_TEXTURE_3D` flag

. Flags `GL_TEXTURE_GEN_`*x* where *x* is S, T, R, or Q

`GL_EVAL_BIT`

`GL_MAP1_`*x* enable bits, where *x* is a map type

. GL_MAP2_*x* enable bits, where *x* is a map type
. 1D grid endpoints and divisions
. 2D grid endpoints and divisions
. GL_AUTO_NORMAL enable bit

GL_FOG_BIT
GL_FOG enable bit
. Fog color
. Fog density
. Linear fog start
. Linear fog end
. Fog index
. GL_FOG_MODE value

GL_HINT_BIT
GL_PERSPECTIVE_CORRECTION_HINT setting
. GL_POINT_SMOOTH_HINT setting
. GL_LINE_SMOOTH_HINT setting
. GL_POLYGON_SMOOTH_HINT setting
. GL_FOG_HINT setting
. GL_GENERATE_MIPMAP_HINT setting
. GL_TEXTURE_COMPRESSION_HINT setting

GL_LIGHTING_BIT
GL_COLOR_MATERIAL enable bit
. GL_COLOR_MATERIAL_FACE value
. Color material parameters that are tracking the current color
. Ambient scene color
. GL_LIGHT_MODEL_LOCAL_VIEWER value
. GL_LIGHT_MODEL_TWO_SIDE setting
. GL_LIGHTING enable bit
. Enable bit for each light
. Ambient, diffuse, and specular intensity for each light
. Direction, position, exponent, and cutoff angle for each light
. Constant, linear, and quadratic attenuation factors for each light
. Ambient, diffuse, specular, and emissive color for each material
. Ambient, diffuse, and specular color indices for each material

. Specular exponent for each material

. GL_SHADE_MODEL setting

GL_LINE_BIT
GL_LINE_SMOOTH flag

. GL_LINE_STIPPLE enable bit

. Line stipple pattern and repeat counter

. Line width

GL_LIST_BIT
GL_LIST_BASE setting

GL_MULTISAMPLE_BIT
GL_MULTISAMPLE flag

. GL_SAMPLE_ALPHA_TO_COVERAGE flag

. GL_SAMPLE_ALPHA_TO_ONE flag

. GL_SAMPLE_COVERAGE flag

. GL_SAMPLE_COVERAGE_VALUE value

. GL_SAMPLE_COVERAGE_INVERT value

GL_PIXEL_MODE_BIT
GL_RED_BIAS and GL_RED_SCALE settings

. GL_GREEN_BIAS and GL_GREEN_SCALE values

. GL_BLUE_BIAS and GL_BLUE_SCALE

. GL_ALPHA_BIAS and GL_ALPHA_SCALE

. GL_DEPTH_BIAS and GL_DEPTH_SCALE

. GL_INDEX_OFFSET and GL_INDEX_SHIFT values

. GL_MAP_COLOR and GL_MAP_STENCIL flags

. GL_ZOOM_X and GL_ZOOM_Y factors

. GL_READ_BUFFER setting

GL_POINT_BIT
GL_POINT_SMOOTH flag

. Point size

GL_POLYGON_BIT
GL_CULL_FACE enable bit

. GL_CULL_FACE_MODE value

. GL_FRONT_FACE indicator

. GL_POLYGON_MODE setting

. GL_POLYGON_SMOOTH flag

. GL_POLYGON_STIPPLE enable bit

. GL_POLYGON_OFFSET_FILL flag

. GL_POLYGON_OFFSET_LINE flag

. GL_POLYGON_OFFSET_POINT flag

. GL_POLYGON_OFFSET_FACTOR

. GL_POLYGON_OFFSET_UNITS

GL_POLYGON_STIPPLE_BIT
Polygon stipple image

GL_SCISSOR_BIT
GL_SCISSOR_TEST flag

. Scissor box

GL_STENCIL_BUFFER_BIT
GL_STENCIL_TEST enable bit

. Stencil function and reference value

. Stencil value mask

. Stencil fail, pass, and depth buffer pass actions

. Stencil buffer clear value

. Stencil buffer writemask

GL_TEXTURE_BIT
Enable bits for the four texture coordinates

. Border color for each texture image

. Minification function for each texture image

. Magnification function for each texture image

. Texture coordinates and wrap mode for each texture image

. Color and mode for each texture environment

. Enable bits GL_TEXTURE_GEN_*x*, *x* is S, T, R, and Q

. GL_TEXTURE_GEN_MODE setting for S, T, R, and Q

. glTexGen plane equations for S, T, R, and Q

. Current texture bindings (for example, GL_TEXTURE_BINDING_2D)

GL_TRANSFORM_BIT
Coefficients of the six clipping planes

. Enable bits for the user-definable clipping planes

. GL_MATRIX_MODE value

. GL_NORMALIZE flag

. `GL_RESCALE_NORMAL` flag

`GL_VIEWPORT_BIT`
Depth range (near and far)

. Viewport origin and extent

`glPopAttrib` restores the values of the state variables saved with the last `glPushAttrib` command. Those not saved are left unchanged.

It is an error to push attributes onto a full stack or to pop attributes off an empty stack. In either case, the error flag is set and no other change is made to GL state.

Initially, the attribute stack is empty.

`GL_STACK_OVERFLOW` is generated if `glPushAttrib` is called while the attribute stack is full.

`GL_STACK_UNDERFLOW` is generated if `glPopAttrib` is called while the attribute stack is empty.

`GL_INVALID_OPERATION` is generated if `glPushAttrib` or `glPopAttrib` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glPushClientAttrib` *mask* [Function]
`void glPopClientAttrib` [Function]
Push and pop the client attribute stack.

mask Specifies a mask that indicates which attributes to save. Values for *mask* are listed below.

`glPushClientAttrib` takes one argument, a mask that indicates which groups of client-state variables to save on the client attribute stack. Symbolic constants are used to set bits in the mask. *mask* is typically constructed by specifying the bitwise-or of several of these constants together. The special mask `GL_CLIENT_ALL_ATTRIB_BITS` can be used to save all stackable client state.

The symbolic mask constants and their associated GL client state are as follows (the second column lists which attributes are saved):

`GL_CLIENT_PIXEL_STORE_BIT` Pixel storage modes `GL_CLIENT_VERTEX_ARRAY_BIT` Vertex arrays (and enables)

`glPopClientAttrib` restores the values of the client-state variables saved with the last `glPushClientAttrib`. Those not saved are left unchanged.

It is an error to push attributes onto a full client attribute stack or to pop attributes off an empty stack. In either case, the error flag is set, and no other change is made to GL state.

Initially, the client attribute stack is empty.

`GL_STACK_OVERFLOW` is generated if `glPushClientAttrib` is called while the attribute stack is full.

`GL_STACK_UNDERFLOW` is generated if `glPopClientAttrib` is called while the attribute stack is empty.

`void glPushMatrix` [Function]
`void glPopMatrix` [Function]

Push and pop the current matrix stack.

There is a stack of matrices for each of the matrix modes. In `GL_MODELVIEW` mode, the stack depth is at least 32. In the other modes, `GL_COLOR`, `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.

`GL_STACK_OVERFLOW` is generated if `glPushMatrix` is called while the current matrix stack is full.

`GL_STACK_UNDERFLOW` is generated if `glPopMatrix` is called while the current matrix stack contains only a single matrix.

`GL_INVALID_OPERATION` is generated if `glPushMatrix` or `glPopMatrix` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glPushName` *name* [Function]
`void glPopName` [Function]

Push and pop the name stack.

name Specifies a name that will be pushed onto the name stack.

The name stack is used during selection mode to allow sets of rendering commands to be uniquely identified. It consists of an ordered set of unsigned integers and is initially empty.

`glPushName` causes *name* to be pushed onto the name stack. `glPopName` pops one name off the top of the stack.

The maximum name stack depth is implementation-dependent; call `GL_MAX_NAME_STACK_DEPTH` to find out the value for a particular implementation. It is an error to push a name onto a full stack or to pop a name off an empty stack. It is also an error to manipulate the name stack between the execution of `glBegin` and the corresponding execution of `glEnd`. In any of these cases, the error flag is set and no other change is made to GL state.

The name stack is always empty while the render mode is not `GL_SELECT`. Calls to `glPushName` or `glPopName` while the render mode is not `GL_SELECT` are ignored.

`GL_STACK_OVERFLOW` is generated if `glPushName` is called while the name stack is full.

`GL_STACK_UNDERFLOW` is generated if `glPopName` is called while the name stack is empty.

`GL_INVALID_OPERATION` is generated if `glPushName` or `glPopName` is executed between a call to `glBegin` and the corresponding call to `glEnd`.

`void glRasterPos2s` *x y* [Function]
`void glRasterPos2i` *x y* [Function]
`void glRasterPos2f` *x y* [Function]
`void glRasterPos2d` *x y* [Function]
`void glRasterPos3s` *x y z* [Function]
`void glRasterPos3i` *x y z* [Function]
`void glRasterPos3f` *x y z* [Function]
`void glRasterPos3d` *x y z* [Function]
`void glRasterPos4s` *x y z w* [Function]
`void glRasterPos4i` *x y z w* [Function]
`void glRasterPos4f` *x y z w* [Function]
`void glRasterPos4d` *x y z w* [Function]
`void glRasterPos2sv` *v* [Function]
`void glRasterPos2iv` *v* [Function]
`void glRasterPos2fv` *v* [Function]
`void glRasterPos2dv` *v* [Function]
`void glRasterPos3sv` *v* [Function]
`void glRasterPos3iv` *v* [Function]
`void glRasterPos3fv` *v* [Function]
`void glRasterPos3dv` *v* [Function]
`void glRasterPos4sv` *v* [Function]
`void glRasterPos4iv` *v* [Function]
`void glRasterPos4fv` *v* [Function]
`void glRasterPos4dv` *v* [Function]

Specify the raster position for pixel operations.

x
y
z
w Specify the *x*, *y*, *z*, and *w* object coordinates (if present) for the raster position.

The GL maintains a 3D position in window coordinates. This position, called the raster position, is used to position pixel and bitmap write operations. It is maintained with subpixel accuracy. See `glBitmap`, `glDrawPixels`, and `glCopyPixels`.

The current raster position consists of three window coordinates (*x*, *y*, *z*), a clip coordinate value (*w*), an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The *w* coordinate is a clip coordinate, because *w* is not projected to window coordinates. `glRasterPos4` specifies object coordinates *x*, *y*, *z*, and *w* explicitly. `glRasterPos3` specifies object coordinate *x*, *y*, and *z* explicitly, while *w* is implicitly set to 1. `glRasterPos2` uses the argument values for *x* and *y* while implicitly setting *z* and *w* to 0 and 1.

The object coordinates presented by `glRasterPos` are treated just like those of a `glVertex` command: They are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, then it is

projected and scaled to window coordinates, which become the new current raster position, and the GL_CURRENT_RASTER_POSITION_VALID flag is set. If the vertex *is* culled, then the valid bit is cleared and the current raster position and associated color and texture coordinates are undefined.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then GL_CURRENT_RASTER_COLOR (in RGBA mode) or GL_CURRENT_RASTER_INDEX (in color index mode) is set to the color produced by the lighting calculation (see glLight, glLightModel, and glShadeModel). If lighting is disabled, current color (in RGBA mode, state variable GL_CURRENT_COLOR) or color index (in color index mode, state variable GL_CURRENT_INDEX) is used to update the current raster color. GL_CURRENT_RASTER_SECONDARY_COLOR (in RGBA mode) is likewise updated.

Likewise, GL_CURRENT_RASTER_TEXTURE_COORDS is updated as a function of GL_CURRENT_TEXTURE_COORDS, based on the texture matrix and the texture generation functions (see glTexGen). Finally, the distance from the origin of the eye coordinate system to the vertex as transformed by only the modelview matrix replaces GL_CURRENT_RASTER_DISTANCE.

Initially, the current raster position is (0, 0, 0, 1), the current raster distance is 0, the valid bit is set, the associated RGBA color is (1, 1, 1, 1), the associated color index is 1, and the associated texture coordinates are (0, 0, 0, 1). In RGBA mode, GL_CURRENT_RASTER_INDEX is always 1; in color index mode, the current raster RGBA color always maintains its initial value.

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

`void glReadBuffer` *mode* [Function]

Select a color buffer source for pixels.

mode Specifies a color buffer. Accepted values are GL_FRONT_LEFT, GL_FRONT_RIGHT, GL_BACK_LEFT, GL_BACK_RIGHT, GL_FRONT, GL_BACK, GL_LEFT, GL_RIGHT, and GL_AUX*i*, where *i* is between 0 and the value of GL_AUX_BUFFERS minus 1.

glReadBuffer specifies a color buffer as the source for subsequent glReadPixels, glCopyTexImage1D, glCopyTexImage2D, glCopyTexSubImage1D, glCopyTexSubImage2D, glCopyTexSubImage3D, and glCopyPixels commands. *mode* accepts one of twelve or more predefined values. (GL_AUX0 through GL_AUX3 are always defined.) In a fully configured system, GL_FRONT, GL_LEFT, and GL_FRONT_LEFT all name the front left buffer, GL_FRONT_RIGHT and GL_RIGHT name the front right buffer, and GL_BACK_LEFT and GL_BACK name the back left buffer.

Nonstereo double-buffered configurations have only a front left and a back left buffer. Single-buffered configurations have a front left and a front right buffer if stereo, and only a front left buffer if nonstereo. It is an error to specify a nonexistent buffer to glReadBuffer.

mode is initially GL_FRONT in single-buffered configurations and GL_BACK in double-buffered configurations.

`GL_INVALID_ENUM` is generated if *mode* is not one of the twelve (or more) accepted values.

`GL_INVALID_OPERATION` is generated if *mode* specifies a buffer that does not exist.

`GL_INVALID_OPERATION` is generated if `glReadBuffer` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glReadPixels` *x y width height format type data* [Function]

Read a block of pixels from the frame buffer.

x
y Specify the window coordinates of the first pixel that is read from the frame buffer. This location is the lower left corner of a rectangular block of pixels.

width
height Specify the dimensions of the pixel rectangle. *width* and *height* of one correspond to a single pixel.

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

type Specifies the data type of the pixel data. Must be one of `GL_UNSIGNED_BYTE`, `GL_BYTE`, `GL_BITMAP`, `GL_UNSIGNED_SHORT`, `GL_SHORT`, `GL_UNSIGNED_INT`, `GL_INT`, `GL_FLOAT`, `GL_UNSIGNED_BYTE_3_3_2`, `GL_UNSIGNED_BYTE_2_3_3_REV`, `GL_UNSIGNED_SHORT_5_6_5`, `GL_UNSIGNED_SHORT_5_6_5_REV`, `GL_UNSIGNED_SHORT_4_4_4_4`, `GL_UNSIGNED_SHORT_4_4_4_4_REV`, `GL_UNSIGNED_SHORT_5_5_5_1`, `GL_UNSIGNED_SHORT_1_5_5_5_REV`, `GL_UNSIGNED_INT_8_8_8_8`, `GL_UNSIGNED_INT_8_8_8_8_REV`, `GL_UNSIGNED_INT_10_10_10_2`, or `GL_UNSIGNED_INT_2_10_10_10_REV`.

data Returns the pixel data.

`glReadPixels` returns pixel data from the frame buffer, starting with the pixel whose lower left corner is at location (x, y), into client memory starting at location *data*. Several parameters control the processing of the pixel data before it is placed into client memory. These parameters are set with three commands: `glPixelStore`, `glPixelTransfer`, and `glPixelMap`. This reference page describes the effects on `glReadPixels` of most, but not all of the parameters specified by these three commands.

If a non-zero named buffer object is bound to the `GL_PIXEL_PACK_BUFFER` target (see `glBindBuffer`) while a block of pixels is requested, *data* is treated as a byte offset into the buffer object's data store rather than a pointer to client memory.

When the `ARB_imaging` extension is supported, the pixel data may be processed by additional operations including color table lookup, color matrix transformations, convolutions, histograms, and minimum and maximum pixel value computations.

`glReadPixels` returns values from each pixel with lower left corner at (x+i,y+j) for 0<=i<*width* and 0<=j<*height*. This pixel is said to be the 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 for the returned pixel values; accepted values are:

`GL_COLOR_INDEX`
: Color indices are read from the color buffer selected by `glReadBuffer`. Each index is converted to fixed point, shifted left or right depending on the value and sign of `GL_INDEX_SHIFT`, and added to `GL_INDEX_OFFSET`. If `GL_MAP_COLOR` is `GL_TRUE`, indices are replaced by their mappings in the table `GL_PIXEL_MAP_I_TO_I`.

`GL_STENCIL_INDEX`
: Stencil values are read from the stencil buffer. Each index is converted to fixed point, shifted left or right depending on the value and sign of `GL_INDEX_SHIFT`, and added to `GL_INDEX_OFFSET`. If `GL_MAP_STENCIL` is `GL_TRUE`, indices are replaced by their mappings in the table `GL_PIXEL_MAP_S_TO_S`.

`GL_DEPTH_COMPONENT`
: Depth values are read from the depth buffer. Each component is converted to floating point such that the minimum depth value maps to 0 and the maximum value maps to 1. Each component is then multiplied by `GL_DEPTH_SCALE`, added to `GL_DEPTH_BIAS`, and finally clamped to the range [0,1].

`GL_RED`

`GL_GREEN`

`GL_BLUE`

`GL_ALPHA`

`GL_RGB`

`GL_BGR`

`GL_RGBA`

`GL_BGRA`

`GL_LUMINANCE`
`GL_LUMINANCE_ALPHA`
: Processing differs depending on whether color buffers store color indices or RGBA color components. If color indices are stored, they are read from the color buffer selected by `glReadBuffer`. Each index is converted to fixed point, shifted left or right depending on the value and sign of `GL_INDEX_SHIFT`, and added to `GL_INDEX_OFFSET`. Indices are then replaced by the red, green, blue, and alpha values obtained by indexing the tables `GL_PIXEL_MAP_I_TO_R`, `GL_PIXEL_MAP_I_TO_G`, `GL_PIXEL_MAP_I_TO_B`, and `GL_PIXEL_MAP_I_TO_A`. Each table must be of size 2^n, but n may be different for different tables. Before an index is used to look up a value in a table of size 2^n, it must be masked against 2^n-1.

If RGBA color components are stored in the color buffers, they are read from the color buffer selected by `glReadBuffer`. Each color component is converted to floating point such that zero intensity maps to 0.0 and full intensity maps to 1.0. Each component is then multiplied by `GL_c_SCALE` and added to `GL_c_BIAS`, where *c* is RED, GREEN, BLUE, or ALPHA. Finally, if `GL_MAP_COLOR` is `GL_TRUE`, each component is clamped to the range [0,1], scaled to the size of its corresponding table, and is then replaced by its mapping in the table `GL_PIXEL_MAP_c_TO_c`, where *c* is R, G, B, or A.

Unneeded data is then discarded. For example, `GL_RED` discards the green, blue, and alpha components, while `GL_RGB` discards only the alpha component. `GL_LUMINANCE` computes a single-component value as the sum of the red, green, and blue components, and `GL_LUMINANCE_ALPHA` does the same, while keeping alpha as a second value. The final values are clamped to the range [0,1].

The shift, scale, bias, and lookup factors just described are all specified by `glPixelTransfer`. The lookup table contents themselves are specified by `glPixelMap`.

Finally, the indices or components are converted to the proper format, as specified by *type*. If *format* is `GL_COLOR_INDEX` or `GL_STENCIL_INDEX` and *type* is not `GL_FLOAT`, each index is masked with the mask value given in the following table. If *type* is `GL_FLOAT`, then each integer index is converted to single-precision floating-point format.

If *format* is `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`, `GL_RGB`, `GL_BGR`, `GL_RGBA`, `GL_BGRA`, `GL_LUMINANCE`, or `GL_LUMINANCE_ALPHA` and *type* is not `GL_FLOAT`, each component is multiplied by the multiplier shown in the following table. If type is `GL_FLOAT`, then each component is passed as is (or converted to the client's single-precision floating-point format if it is different from the one used by the GL).

type	**Index Mask**, **Component Conversion**
`GL_UNSIGNED_BYTE`	2^8-1, (2^8-1,)*c*
`GL_BYTE`	2^7-1, (2^8-1,)*c*-1,/2
`GL_BITMAP`	1, 1
`GL_UNSIGNED_SHORT`	2^16-1, (2^16-1,)*c*
`GL_SHORT`	2^15-1, (2^16-1,)*c*-1,/2
`GL_UNSIGNED_INT`	2^32-1, (2^32-1,)*c*
`GL_INT`	2^31-1, (2^32-1,)*c*-1,/2
`GL_FLOAT`	none , *c*

Return values are placed in memory as follows. If *format* is GL_COLOR_INDEX, GL_STENCIL_INDEX, GL_DEPTH_COMPONENT, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, or GL_LUMINANCE, a single value is returned and the data for the *i*th pixel in the *j*th row is placed in location (*j*,)*width*+*i*. GL_RGB and GL_BGR return three values, GL_RGBA and GL_BGRA return four values, and GL_LUMINANCE_ALPHA returns two values for each pixel, with all values corresponding to a single pixel occupying contiguous space in *data*. Storage parameters set by glPixelStore, such as GL_PACK_LSB_FIRST and GL_PACK_SWAP_BYTES, affect the way that data is written into memory. See glPixelStore for a description.

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

GL_INVALID_ENUM is generated if *type* is GL_BITMAP and *format* is not GL_COLOR_INDEX or GL_STENCIL_INDEX.

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

GL_INVALID_OPERATION is generated if *format* is GL_COLOR_INDEX and the color buffers store RGBA color components.

GL_INVALID_OPERATION is generated if *format* is GL_STENCIL_INDEX and there is no stencil buffer.

GL_INVALID_OPERATION is generated if *format* is GL_DEPTH_COMPONENT and there is no depth buffer.

GL_INVALID_OPERATION is generated if *type* is one of GL_UNSIGNED_BYTE_3_3_2, GL_UNSIGNED_BYTE_2_3_3_REV, GL_UNSIGNED_SHORT_5_6_5, or GL_UNSIGNED_SHORT_5_6_5_REV and *format* is not GL_RGB.

GL_INVALID_OPERATION is generated if *type* is one of GL_UNSIGNED_SHORT_4_4_4_4, GL_UNSIGNED_SHORT_4_4_4_4_REV, GL_UNSIGNED_SHORT_5_5_5_1, GL_UNSIGNED_SHORT_1_5_5_5_REV, GL_UNSIGNED_INT_8_8_8_8, GL_UNSIGNED_INT_8_8_8_8_REV, GL_UNSIGNED_INT_10_10_10_2, or GL_UNSIGNED_INT_2_10_10_10_REV and *format* is neither GL_RGBA nor GL_BGRA.

The formats GL_BGR, and GL_BGRA and types GL_UNSIGNED_BYTE_3_3_2, GL_UNSIGNED_BYTE_2_3_3_REV, GL_UNSIGNED_SHORT_5_6_5, GL_UNSIGNED_SHORT_5_6_5_REV, GL_UNSIGNED_SHORT_4_4_4_4, GL_UNSIGNED_SHORT_4_4_4_4_REV, GL_UNSIGNED_SHORT_5_5_5_1, GL_UNSIGNED_SHORT_1_5_5_5_REV, GL_UNSIGNED_INT_8_8_8_8, GL_UNSIGNED_INT_8_8_8_8_REV, GL_UNSIGNED_INT_10_10_10_2, and GL_UNSIGNED_INT_2_10_10_10_REV are available only if the GL version is 1.2 or greater.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_PACK_BUFFER target and the buffer object's data store is currently mapped.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_PACK_BUFFER target and the data would be packed to the buffer object such that the memory writes required would exceed the data store size.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_PACK_BUFFER target and *data* is not evenly divisible into the number of bytes needed to store in memory a datum indicated by *type*.

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

`void glRectd` *x1 y1 x2 y2* [Function]
`void glRectf` *x1 y1 x2 y2* [Function]
`void glRecti` *x1 y1 x2 y2* [Function]
`void glRects` *x1 y1 x2 y2* [Function]
`void glRectdv` *v1 v2* [Function]
`void glRectfv` *v1 v2* [Function]
`void glRectiv` *v1 v2* [Function]
`void glRectsv` *v1 v2* [Function]

Draw a rectangle.

x1
y1 Specify one vertex of a rectangle.

x2
y2 Specify the opposite vertex of the rectangle.

`glRect` supports efficient specification of rectangles as two corner points. Each rectangle command takes four arguments, organized either as two consecutive pairs of (x,y) coordinates or as two pointers to arrays, each containing an (x,y) pair. The resulting rectangle is defined in the z=0 plane.

`glRect`(*x1*, *y1*, *x2*, *y2*) is exactly equivalent to the following sequence: Note that if the second vertex is above and to the right of the first vertex, the rectangle is constructed with a counterclockwise winding.

```
glBegin(GL_POLYGON);
glVertex2(x1, y1);
glVertex2(x2, y1);
glVertex2(x2, y2);
glVertex2(x1, y2);
glEnd();
```

`GL_INVALID_OPERATION` is generated if `glRect` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`GLint glRenderMode` *mode* [Function]

Set rasterization mode.

mode Specifies the rasterization mode. Three values are accepted: `GL_RENDER`, `GL_SELECT`, and `GL_FEEDBACK`. The initial value is `GL_RENDER`.

`glRenderMode` sets the rasterization mode. It takes one argument, *mode*, which can assume one of three predefined values:

`GL_RENDER`
Render mode. Primitives are rasterized, producing pixel fragments, which are written into the frame buffer. This is the normal mode and also the default mode.

`GL_SELECT`
Selection mode. No pixel fragments are produced, and no change to the frame buffer contents is made. Instead, a record of the names of primitives

that would have been drawn if the render mode had been `GL_RENDER` is returned in a select buffer, which must be created (see `glSelectBuffer`) before selection mode is entered.

`GL_FEEDBACK`
: Feedback mode. No pixel fragments are produced, and no change to the frame buffer contents is made. Instead, the coordinates and attributes of vertices that would have been drawn if the render mode had been `GL_RENDER` is returned in a feedback buffer, which must be created (see `glFeedbackBuffer`) before feedback mode is entered.

The return value of `glRenderMode` is determined by the render mode at the time `glRenderMode` is called, rather than by *mode*. The values returned for the three render modes are as follows:

`GL_RENDER`
: 0.

`GL_SELECT`
: The number of hit records transferred to the select buffer.

`GL_FEEDBACK`
: The number of values (not vertices) transferred to the feedback buffer.

See the `glSelectBuffer` and `glFeedbackBuffer` reference pages for more details concerning selection and feedback operation.

`GL_INVALID_ENUM` is generated if *mode* is not one of the three accepted values.

`GL_INVALID_OPERATION` is generated if `glSelectBuffer` is called while the render mode is `GL_SELECT`, or if `glRenderMode` is called with argument `GL_SELECT` before `glSelectBuffer` is called at least once.

`GL_INVALID_OPERATION` is generated if `glFeedbackBuffer` is called while the render mode is `GL_FEEDBACK`, or if `glRenderMode` is called with argument `GL_FEEDBACK` before `glFeedbackBuffer` is called at least once.

`GL_INVALID_OPERATION` is generated if `glRenderMode` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glResetHistogram` *target* [Function]

Reset histogram table entries to zero.

target Must be `GL_HISTOGRAM`.

`glResetHistogram` resets all the elements of the current histogram table to zero.

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

`GL_INVALID_OPERATION` is generated if `glResetHistogram` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glResetMinmax` *target* [Function]

Reset minmax table entries to initial values.

target Must be `GL_MINMAX`.

`glResetMinmax` resets the elements of the current minmax table to their initial values: the "maximum" element receives the minimum possible component values, and the "minimum" element receives the maximum possible component values.

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

`GL_INVALID_OPERATION` is generated if `glResetMinmax` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glRotated` *angle x y z* [Function]
`void glRotatef` *angle x y z* [Function]

Multiply the current matrix by a rotation matrix.

angle Specifies the angle of rotation, in degrees.

x
y
z Specify the *x*, *y*, and *z* coordinates of a vector, respectively.

`glRotate` produces a rotation of *angle* degrees around the vector (*x*,*yz*). 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:

((x^2(1-c,)+c xy(1-c,)-zs xz(1-c,)+ys 0), (yx(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),)

Where *c*=*cos*(*angle*,), *s*=*sin*(*angle*,), and (*x*,*yz*),=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.

`GL_INVALID_OPERATION` is generated if `glRotate` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glSampleCoverage` *value invert* [Function]

Specify multisample coverage parameters.

value Specify a single floating-point sample coverage value. The value is clamped to the range [0,1]. The initial value is 1.0.

invert Specify a single boolean value representing if the coverage masks should be inverted. `GL_TRUE` and `GL_FALSE` are accepted. The initial value is `GL_FALSE`.

Multisampling samples a pixel multiple times at various implementation-dependent subpixel locations to generate antialiasing effects. Multisampling transparently antialiases points, lines, polygons, bitmaps, and images if it is enabled.

value is used in constructing a temporary mask used in determining which samples will be used in resolving the final fragment color. This mask is bitwise-anded with the coverage mask generated from the multisampling computation. If the *invert* flag is set, the temporary mask is inverted (all bits flipped) and then the bitwise-and is computed.

If an implementation does not have any multisample buffers available, or multisampling is disabled, rasterization occurs with only a single sample computing a pixel's final RGB color.

Provided an implementation supports multisample buffers, and multisampling is enabled, then a pixel's final color is generated by combining several samples per pixel. Each sample contains color, depth, and stencil information, allowing those operations to be performed on each sample.

`GL_INVALID_OPERATION` is generated if `glSampleCoverage` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glScaled` *x y z* [Function]
`void glScalef` *x y z* [Function]

Multiply the current matrix by a general scaling matrix.

x
y
z Specify scale factors along the *x*, *y*, and *z* axes, respectively.

`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 `glMultMatrix` were called with the following matrix as its argument:

((*x* 0 0 0), (0 *y* 0 0), (0 0 *z* 0), (0 0 0 1),)

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.

`GL_INVALID_OPERATION` is generated if `glScale` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glScissor` *x y width height* [Function]

Define the scissor box.

x
y Specify the lower left corner of the scissor box. Initially (0, 0).

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

`glScissor` defines a rectangle, called the scissor box, in window coordinates. The first two arguments, *x* and *y*, specify the lower left corner of the box. *width* and *height* specify the width and height of the box.

To enable and disable the scissor test, call `glEnable` and `glDisable` with argument `GL_SCISSOR_TEST`. The test is initially disabled. While the test is enabled, only pixels that lie within the scissor box can be modified by drawing commands. Window coordinates have integer values at the shared corners of frame buffer pixels.

`glScissor(0,0,1,1)` allows modification of only the lower left pixel in the window, and `glScissor(0,0,0,0)` doesn't allow modification of any pixels in the window.

When the scissor test is disabled, it is as though the scissor box includes the entire window.

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

`GL_INVALID_OPERATION` is generated if `glScissor` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glSecondaryColorPointer` *size type stride pointer* [Function]

Define an array of secondary colors.

size Specifies the number of components per color. Must be 3.

type Specifies the data type of each color component in the array. Symbolic constants `GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_UNSIGNED_SHORT`, `GL_INT`, `GL_UNSIGNED_INT`, `GL_FLOAT`, or `GL_DOUBLE` are accepted. The initial value is `GL_FLOAT`.

stride Specifies the byte offset between consecutive colors. If *stride* is 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. The initial value is 0.

`glSecondaryColorPointer` specifies the location and data format of an array of color components to use when rendering. *size* specifies the number of components per color, and must be 3. *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.

If a non-zero named buffer object is bound to the `GL_ARRAY_BUFFER` target (see `glBindBuffer`) while a secondary color array is specified, *pointer* is treated as a byte offset into the buffer object's data store. Also, the buffer object binding (`GL_ARRAY_BUFFER_BINDING`) is saved as secondary color vertex array client-side state (`GL_SECONDARY_COLOR_ARRAY_BUFFER_BINDING`).

When a secondary color array is specified, *size*, *type*, *stride*, and *pointer* are saved as client-side state, in addition to the current vertex array buffer object binding.

To enable and disable the secondary color array, call `glEnableClientState` and `glDisableClientState` with the argument `GL_SECONDARY_COLOR_ARRAY`. If enabled, the secondary color array is used when `glArrayElement`, `glDrawArrays`, `glMultiDrawArrays`, `glDrawElements`, `glMultiDrawElements`, or `glDrawRangeElements` is called.

`GL_INVALID_VALUE` is generated if *size* is not 3.

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

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

`void glSecondaryColor3b` *red green blue* [Function]
`void glSecondaryColor3s` *red green blue* [Function]

`void glSecondaryColor3i` *red green blue* [Function]
`void glSecondaryColor3f` *red green blue* [Function]
`void glSecondaryColor3d` *red green blue* [Function]
`void glSecondaryColor3ub` *red green blue* [Function]
`void glSecondaryColor3us` *red green blue* [Function]
`void glSecondaryColor3ui` *red green blue* [Function]
`void glSecondaryColor3bv` *v* [Function]
`void glSecondaryColor3sv` *v* [Function]
`void glSecondaryColor3iv` *v* [Function]
`void glSecondaryColor3fv` *v* [Function]
`void glSecondaryColor3dv` *v* [Function]
`void glSecondaryColor3ubv` *v* [Function]
`void glSecondaryColor3usv` *v* [Function]
`void glSecondaryColor3uiv` *v* [Function]

Set the current secondary color.

red
green
blue Specify new red, green, and blue values for the current secondary color.

The GL stores both a primary four-valued RGBA color and a secondary four-valued RGBA color (where alpha is always set to 0.0) that is associated with every vertex.

The secondary color is interpolated and applied to each fragment during rasterization when `GL_COLOR_SUM` is enabled. When lighting is enabled, and `GL_SEPARATE_SPECULAR_COLOR` is specified, the value of the secondary color is assigned the value computed from the specular term of the lighting computation. Both the primary and secondary current colors are applied to each fragment, regardless of the state of `GL_COLOR_SUM`, under such conditions. When `GL_SEPARATE_SPECULAR_COLOR` is specified, the value returned from querying the current secondary color is undefined.

`glSecondaryColor3b`, `glSecondaryColor3s`, and `glSecondaryColor3i` take three signed byte, short, or long integers as arguments. When `v` is appended to the name, the color commands can take a pointer to an array of such values.

Color values are stored in floating-point format, with unspecified mantissa and exponent sizes. Unsigned integer color components, when specified, are linearly mapped to floating-point values such that the largest representable value maps to 1.0 (full intensity), and 0 maps to 0.0 (zero intensity). Signed integer color components, when specified, are linearly mapped to floating-point values such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. (Note that this mapping does not convert 0 precisely to 0.0). Floating-point values are mapped directly.

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

`void glSelectBuffer` *size buffer* [Function]

Establish a buffer for selection mode values.

size Specifies the size of *buffer*.

buffer Returns the selection data.

`glSelectBuffer` has two arguments: *buffer* is a pointer to an array of unsigned integers, and *size* indicates the size of the array. *buffer* returns values from the name stack (see `glInitNames`, `glLoadName`, `glPushName`) when the rendering mode is `GL_SELECT` (see `glRenderMode`). `glSelectBuffer` must be issued before selection mode is enabled, and it must not be issued while the rendering mode is `GL_SELECT`.

A programmer can use selection to determine which primitives are drawn into some region of a window. The region is defined by the current modelview and perspective matrices.

In selection mode, no pixel fragments are produced from rasterization. Instead, if a primitive or a raster position intersects the clipping volume defined by the viewing frustum and the user-defined clipping planes, this primitive causes a selection hit. (With polygons, no hit occurs if the polygon is culled.) When a change is made to the name stack, or when `glRenderMode` is called, a hit record is copied to *buffer* if any hits have occurred since the last such event (name stack change or `glRenderMode` call). The hit record consists of the number of names in the name stack at the time of the event, followed by the minimum and maximum depth values of all vertices that hit since the previous event, followed by the name stack contents, bottom name first.

Depth values (which are in the range [0,1]) are multiplied by 2^32-1, before being placed in the hit record.

An internal index into *buffer* is reset to 0 whenever selection mode is entered. Each time a hit record is copied into *buffer*, the index is incremented to point to the cell just past the end of the block of names\(emthat is, to the next available cell If the hit record is larger than the number of remaining locations in *buffer*, as much data as can fit is copied, and the overflow flag is set. If the name stack is empty when a hit record is copied, that record consists of 0 followed by the minimum and maximum depth values.

To exit selection mode, call `glRenderMode` with an argument other than `GL_SELECT`. Whenever `glRenderMode` is called while the render mode is `GL_SELECT`, it returns the number of hit records copied to *buffer*, resets the overflow flag and the selection buffer pointer, and initializes the name stack to be empty. If the overflow bit was set when `glRenderMode` was called, a negative hit record count is returned.

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

`GL_INVALID_OPERATION` is generated if `glSelectBuffer` is called while the render mode is `GL_SELECT`, or if `glRenderMode` is called with argument `GL_SELECT` before `glSelectBuffer` is called at least once.

`GL_INVALID_OPERATION` is generated if `glSelectBuffer` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glSeparableFilter2D` *target internalformat width height format type row column* [Function]

Define a separable two-dimensional convolution filter.

target Must be `GL_SEPARABLE_2D`.

internalformat
: The internal format of the convolution filter kernel. The allowable values are `GL_ALPHA`, `GL_ALPHA4`, `GL_ALPHA8`, `GL_ALPHA12`, `GL_ALPHA16`, `GL_LUMINANCE`, `GL_LUMINANCE4`, `GL_LUMINANCE8`, `GL_LUMINANCE12`, `GL_LUMINANCE16`, `GL_LUMINANCE_ALPHA`, `GL_LUMINANCE4_ALPHA4`, `GL_LUMINANCE6_ALPHA2`, `GL_LUMINANCE8_ALPHA8`, `GL_LUMINANCE12_ALPHA4`, `GL_LUMINANCE12_ALPHA12`, `GL_LUMINANCE16_ALPHA16`, `GL_INTENSITY`, `GL_INTENSITY4`, `GL_INTENSITY8`, `GL_INTENSITY12`, `GL_INTENSITY16`, `GL_R3_G3_B2`, `GL_RGB`, `GL_RGB4`, `GL_RGB5`, `GL_RGB8`, `GL_RGB10`, `GL_RGB12`, `GL_RGB16`, `GL_RGBA`, `GL_RGBA2`, `GL_RGBA4`, `GL_RGB5_A1`, `GL_RGBA8`, `GL_RGB10_A2`, `GL_RGBA12`, or `GL_RGBA16`.

width
: The number of elements in the pixel array referenced by *row*. (This is the width of the separable filter kernel.)

height
: The number of elements in the pixel array referenced by *column*. (This is the height of the separable filter kernel.)

format
: The format of the pixel data in *row* and *column*. The allowable values are `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`, `GL_RGB`, `GL_BGR`, `GL_RGBA`, `GL_BGRA`, `GL_INTENSITY`, `GL_LUMINANCE`, and `GL_LUMINANCE_ALPHA`.

type
: The type of the pixel data in *row* and *column*. Symbolic constants `GL_UNSIGNED_BYTE`, `GL_BYTE`, `GL_BITMAP`, `GL_UNSIGNED_SHORT`, `GL_SHORT`, `GL_UNSIGNED_INT`, `GL_INT`, `GL_FLOAT`, `GL_UNSIGNED_BYTE_3_3_2`, `GL_UNSIGNED_BYTE_2_3_3_REV`, `GL_UNSIGNED_SHORT_5_6_5`, `GL_UNSIGNED_SHORT_5_6_5_REV`, `GL_UNSIGNED_SHORT_4_4_4_4`, `GL_UNSIGNED_SHORT_4_4_4_4_REV`, `GL_UNSIGNED_SHORT_5_5_5_1`, `GL_UNSIGNED_SHORT_1_5_5_5_REV`, `GL_UNSIGNED_INT_8_8_8_8`, `GL_UNSIGNED_INT_8_8_8_8_REV`, `GL_UNSIGNED_INT_10_10_10_2`, and `GL_UNSIGNED_INT_2_10_10_10_REV` are accepted.

row
: Pointer to a one-dimensional array of pixel data that is processed to build the row filter kernel.

column
: Pointer to a one-dimensional array of pixel data that is processed to build the column filter kernel.

`glSeparableFilter2D` builds a two-dimensional separable convolution filter kernel from two arrays of pixels.

The pixel arrays specified by (*width*, *format*, *type*, *row*) and (*height*, *format*, *type*, *column*) are processed just as if they had been passed to `glDrawPixels`, but processing stops after the final expansion to RGBA is completed.

If a non-zero named buffer object is bound to the `GL_PIXEL_UNPACK_BUFFER` target (see `glBindBuffer`) while a convolution filter is specified, *row* and *column* are treated as byte offsets into the buffer object's data store.

Next, the R, G, B, and A components of all pixels in both arrays are scaled by the four separable 2D `GL_CONVOLUTION_FILTER_SCALE` parameters and biased by the four separable 2D `GL_CONVOLUTION_FILTER_BIAS` parameters. (The scale and bias parameters are set by `glConvolutionParameter` using the `GL_SEPARABLE_2D` target

and the names GL_CONVOLUTION_FILTER_SCALE and GL_CONVOLUTION_FILTER_BIAS. The parameters themselves are vectors of four values that are applied to red, green, blue, and alpha, in that order.) The R, G, B, and A values are not clamped to [0,1] at any time during this process.

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

Internal Format
: **Red**, **Green**, **Blue**, **Alpha**, **Luminance**, **Intensity**

GL_LUMINANCE
: , , , , R ,

GL_LUMINANCE_ALPHA
: , , , A , R ,

GL_INTENSITY
: , , , , , R

GL_RGB
: R , G , B , , ,

GL_RGBA
: R , G , B , A , ,

The red, green, blue, alpha, luminance, and/or intensity components of the resulting pixels are stored in floating-point rather than integer format. They form two one-dimensional filter kernel images. The row image is indexed by coordinate *i* starting at zero and increasing from left to right. Each location in the row image is derived from element *i* of *row*. The column image is indexed by coordinate *j* starting at zero and increasing from bottom to top. Each location in the column image is derived from element *j* of *column*.

Note that after a convolution is performed, the resulting color components are also scaled by their corresponding GL_POST_CONVOLUTION_c_SCALE parameters and biased by their corresponding GL_POST_CONVOLUTION_c_BIAS parameters (where *c* takes on the values **RED**, **GREEN**, **BLUE**, and **ALPHA**). These parameters are set by glPixelTransfer.

GL_INVALID_ENUM is generated if *target* is not GL_SEPARABLE_2D.

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

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

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

GL_INVALID_VALUE is generated if *width* is less than zero or greater than the maximum supported value. This value may be queried with glGetConvolutionParameter using target GL_SEPARABLE_2D and name GL_MAX_CONVOLUTION_WIDTH.

GL_INVALID_VALUE is generated if *height* is less than zero or greater than the maximum supported value. This value may be queried with glGetConvolutionParameter using target GL_SEPARABLE_2D and name GL_MAX_CONVOLUTION_HEIGHT.

GL_INVALID_OPERATION is generated if *height* is one of GL_UNSIGNED_BYTE_3_3_2, GL_UNSIGNED_BYTE_2_3_3_REV, GL_UNSIGNED_SHORT_5_6_5, or GL_UNSIGNED_SHORT_5_6_5_REV and *format* is not GL_RGB.

`GL_INVALID_OPERATION` is generated if *height* is one of `GL_UNSIGNED_SHORT_4_4_4_4`, `GL_UNSIGNED_SHORT_4_4_4_4_REV`, `GL_UNSIGNED_SHORT_5_5_5_1`, `GL_UNSIGNED_SHORT_1_5_5_5_REV`, `GL_UNSIGNED_INT_8_8_8_8`, `GL_UNSIGNED_INT_8_8_8_8_REV`, `GL_UNSIGNED_INT_10_10_10_2`, or `GL_UNSIGNED_INT_2_10_10_10_REV` and *format* is neither `GL_RGBA` nor `GL_BGRA`.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the buffer object's data store is currently mapped.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and *row* or *column* is not evenly divisible into the number of bytes needed to store in memory a datum indicated by *type*.

`GL_INVALID_OPERATION` is generated if `glSeparableFilter2D` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glShadeModel` *mode* [Function]

Select flat or smooth shading.

mode Specifies a symbolic value representing a shading technique. Accepted values are `GL_FLAT` and `GL_SMOOTH`. The initial value is `GL_SMOOTH`.

GL primitives can have either flat or smooth shading. Smooth shading, the default, causes the computed colors of vertices to be interpolated as the primitive is rasterized, typically assigning different colors to each resulting pixel fragment. Flat shading selects the computed color of just one vertex and assigns it to all the pixel fragments generated by rasterizing a single primitive. In either case, the computed color of a vertex is the result of lighting if lighting is enabled, or it is the current color at the time the vertex was specified if lighting is disabled.

Flat and smooth shading are indistinguishable for points. Starting when `glBegin` is issued and counting vertices and primitives from 1, the GL gives each flat-shaded line segment *i* the computed color of vertex *i*+1, its second vertex. Counting similarly from 1, the GL gives each flat-shaded polygon the computed color of the vertex listed in the following table. This is the last vertex to specify the polygon in all cases except single polygons, where the first vertex specifies the flat-shaded color.

Primitive Type of Polygon *i*
Vertex

Single polygon (*i*==1)
1

Triangle strip
i+2

Triangle fan
i+2

Independent triangle
3*i*

Quad strip
: $2i+2$

Independent quad
: $4i$

Flat and smooth shading are specified by `glShadeModel` with *mode* set to `GL_FLAT` and `GL_SMOOTH`, respectively.

`GL_INVALID_ENUM` is generated if *mode* is any value other than `GL_FLAT` or `GL_SMOOTH`.

`GL_INVALID_OPERATION` is generated if `glShadeModel` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glShaderSource` *shader count string length* [Function]

Replaces the source code in a shader object.

shader
: Specifies the handle of the shader object whose source code is to be replaced.

count
: Specifies the number of elements in the *string* and *length* arrays.

string
: Specifies an array of pointers to strings containing the source code to be loaded into the shader.

length
: Specifies an array of string lengths.

`glShaderSource` sets the source code in *shader* to the source code in the array of strings specified by *string*. Any source code previously stored in the shader object is completely replaced. The number of strings in the array is specified by *count*. If *length* is `NULL`, each string is assumed to be null terminated. If *length* is a value other than `NULL`, it points to an array containing a string length for each of the corresponding elements of *string*. Each element in the *length* array may contain the length of the corresponding string (the null character is not counted as part of the string length) or a value less than 0 to indicate that the string is null terminated. The source code strings are not scanned or parsed at this time; they are simply copied into the specified shader object.

`GL_INVALID_VALUE` is generated if *shader* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *shader* is not a shader object.

`GL_INVALID_VALUE` is generated if *count* is less than 0.

`GL_INVALID_OPERATION` is generated if `glShaderSource` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glStencilFuncSeparate` *face func ref mask* [Function]

Set front and/or back function and reference value for stencil testing.

face
: Specifies whether front and/or back stencil state is updated. Three symbolic constants are valid: `GL_FRONT`, `GL_BACK`, and `GL_FRONT_AND_BACK`.

func
: Specifies the test function. Eight symbolic constants 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.

Stenciling, like depth-buffering, enables and disables drawing on a per-pixel basis. You draw into the stencil planes using GL drawing primitives, then render geometry and images, using the stencil planes to mask out portions of the screen. Stenciling is typically used in multipass rendering algorithms to achieve special effects, such as decals, outlining, and constructive solid geometry rendering.

The stencil test conditionally eliminates a pixel based on the outcome of a comparison between the reference value and the value in the stencil buffer. To enable and disable the test, call `glEnable` and `glDisable` with argument `GL_STENCIL_TEST`. To specify actions based on the outcome of the stencil test, call `glStencilOp` or `glStencilOpSeparate`.

There can be two separate sets of *func*, *ref*, and *mask* parameters; one affects back-facing polygons, and the other affects front-facing polygons as well as other non-polygon primitives. `glStencilFunc` sets both front and back stencil state to the same values, as if `glStencilFuncSeparate` were called with *face* set to `GL_FRONT_AND_BACK`.

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*) <= (*stencil* & *mask*).

`GL_GREATER`
Passes if (*ref* & *mask*) > (*stencil* & *mask*).

`GL_GEQUAL`
Passes if (*ref* & *mask*) >= (*stencil* & *mask*).

`GL_EQUAL` Passes if (*ref* & *mask*) = (*stencil* & *mask*).

`GL_NOTEQUAL`
: Passes if (*ref* & *mask*) != (*stencil* & *mask*).

`GL_ALWAYS`
: Always passes.

`GL_INVALID_ENUM` is generated if *func* is not one of the eight accepted values.

`GL_INVALID_OPERATION` is generated if `glStencilFuncSeparate` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glStencilFunc` *func ref mask* [Function]

Set front and back function and reference value for stencil testing.

func
: Specifies the test function. Eight symbolic constants 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.

Stenciling, like depth-buffering, enables and disables drawing on a per-pixel basis. Stencil planes are first drawn into using GL drawing primitives, then geometry and images are rendered using the stencil planes to mask out portions of the screen. Stenciling is typically used in multipass rendering algorithms to achieve special effects, such as decals, outlining, and constructive solid geometry rendering.

The stencil test conditionally eliminates a pixel based on the outcome of a comparison between the reference value and the value in the stencil buffer. To enable and disable the test, call `glEnable` and `glDisable` with argument `GL_STENCIL_TEST`. To specify actions based on the outcome of the stencil test, call `glStencilOp` or `glStencilOpSeparate`.

There can be two separate sets of *func*, *ref*, and *mask* parameters; one affects back-facing polygons, and the other affects front-facing polygons as well as other non-polygon primitives. `glStencilFunc` sets both front and back stencil state to the same values. Use `glStencilFuncSeparate` to set front and back stencil state to different values.

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*) <= (*stencil* & *mask*).

`GL_GREATER`
Passes if (*ref* & *mask*) > (*stencil* & *mask*).

`GL_GEQUAL`
Passes if (*ref* & *mask*) >= (*stencil* & *mask*).

`GL_EQUAL` Passes if (*ref* & *mask*) = (*stencil* & *mask*).

`GL_NOTEQUAL`
Passes if (*ref* & *mask*) != (*stencil* & *mask*).

`GL_ALWAYS`
Always passes.

`GL_INVALID_ENUM` is generated if *func* is not one of the eight accepted values.

`GL_INVALID_OPERATION` is generated if `glStencilFunc` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glStencilMaskSeparate` *face mask* [Function]
Control the front and/or back writing of individual bits in the stencil planes.

face Specifies whether the front and/or back stencil writemask is updated. Three symbolic constants are valid: `GL_FRONT`, `GL_BACK`, and `GL_FRONT_AND_BACK`.

mask Specifies a bit mask to enable and disable writing of individual bits in the stencil planes. Initially, the mask is all 1's.

`glStencilMaskSeparate` 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.

There can be two separate *mask* writemasks; one affects back-facing polygons, and the other affects front-facing polygons as well as other non-polygon primitives. `glStencilMask` sets both front and back stencil writemasks to the same values, as if `glStencilMaskSeparate` were called with *face* set to `GL_FRONT_AND_BACK`.

`GL_INVALID_OPERATION` is generated if `glStencilMaskSeparate` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glStencilMask` *mask* [Function]
Control the front and back writing of individual bits in the stencil planes.

mask Specifies a bit mask to enable and disable writing of individual bits in the stencil planes. Initially, the mask is all 1's.

`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.

There can be two separate *mask* writemasks; one affects back-facing polygons, and the other affects front-facing polygons as well as other non-polygon primitives. `glStencilMask` sets both front and back stencil writemasks to the same values. Use `glStencilMaskSeparate` to set front and back stencil writemasks to different values.

`GL_INVALID_OPERATION` is generated if `glStencilMask` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glStencilOpSeparate` *face sfail dpfail dppass* [Function]

Set front and/or back stencil test actions.

face Specifies whether front and/or back stencil state is updated. Three symbolic constants are valid: `GL_FRONT`, `GL_BACK`, and `GL_FRONT_AND_BACK`.

sfail Specifies the action to take when the stencil test fails. Eight symbolic constants are accepted: `GL_KEEP`, `GL_ZERO`, `GL_REPLACE`, `GL_INCR`, `GL_INCR_WRAP`, `GL_DECR`, `GL_DECR_WRAP`, and `GL_INVERT`. The initial value is `GL_KEEP`.

dpfail Specifies the stencil action when the stencil test passes, but the depth test fails. *dpfail* accepts the same symbolic constants as *sfail*. The initial value is `GL_KEEP`.

dppass 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. *dppass* accepts the same symbolic constants as *sfail*. The initial value is `GL_KEEP`.

Stenciling, like depth-buffering, enables and disables drawing on a per-pixel basis. You draw into the stencil planes using GL drawing primitives, then render geometry and images, using the stencil planes to mask out portions of the screen. Stenciling is typically used in multipass rendering algorithms to achieve special effects, such as decals, outlining, and constructive solid geometry rendering.

The stencil test conditionally eliminates a pixel based on the outcome of a comparison between the value in the stencil buffer and a reference value. To enable and disable the test, call `glEnable` and `glDisable` with argument `GL_STENCIL_TEST`; to control it, call `glStencilFunc` or `glStencilFuncSeparate`.

There can be two separate sets of *sfail*, *dpfail*, and *dppass* parameters; one affects back-facing polygons, and the other affects front-facing polygons as well as other non-polygon primitives. `glStencilOp` sets both front and back stencil state to the same values, as if `glStencilOpSeparate` were called with *face* set to `GL_FRONT_AND_BACK`.

`glStencilOpSeparate` 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 *sfail* specifies what happens to the stencil buffer contents. The following eight 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_INCR_WRAP`
Increments the current stencil buffer value. Wraps stencil buffer value to zero when incrementing the maximum representable unsigned value.

`GL_DECR` Decrements the current stencil buffer value. Clamps to 0.

`GL_DECR_WRAP`
Decrements the current stencil buffer value. Wraps stencil buffer value to the maximum representable unsigned value when decrementing a stencil buffer value of zero.

`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 `glStencilOpSeparate` specify stencil buffer actions that depend on whether subsequent depth buffer tests succeed (*dppass*) or fail (*dpfail*) (see `glDepthFunc`). The actions are specified using the same eight symbolic constants as *sfail*. Note that *dpfail* is ignored when there is no depth buffer, or when the depth buffer is not enabled. In these cases, *sfail* and *dppass* specify stencil action when the stencil test fails and passes, respectively.

`GL_INVALID_ENUM` is generated if *face* is any value other than `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`.

`GL_INVALID_ENUM` is generated if *sfail*, *dpfail*, or *dppass* is any value other than the eight defined constant values.

`GL_INVALID_OPERATION` is generated if `glStencilOpSeparate` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glStencilOp` *sfail dpfail dppass* [Function]
Set front and back stencil test actions.

sfail Specifies the action to take when the stencil test fails. Eight symbolic constants are accepted: `GL_KEEP`, `GL_ZERO`, `GL_REPLACE`, `GL_INCR`, `GL_INCR_WRAP`, `GL_DECR`, `GL_DECR_WRAP`, and `GL_INVERT`. The initial value is `GL_KEEP`.

dpfail Specifies the stencil action when the stencil test passes, but the depth test fails. *dpfail* accepts the same symbolic constants as *sfail*. The initial value is `GL_KEEP`.

dppass 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. *dppass* accepts the same symbolic constants as *sfail*. The initial value is `GL_KEEP`.

Stenciling, like depth-buffering, enables and disables drawing on a per-pixel basis. You draw into the stencil planes using GL drawing primitives, then render geometry and images, using the stencil planes to mask out portions of the screen. Stenciling is typically used in multipass rendering algorithms to achieve special effects, such as decals, outlining, and constructive solid geometry rendering.

The stencil test conditionally eliminates a pixel based on the outcome of a comparison between the value in the stencil buffer and a reference value. To enable and disable the test, call `glEnable` and `glDisable` with argument `GL_STENCIL_TEST`; to control it, call `glStencilFunc` or `glStencilFuncSeparate`.

There can be two separate sets of *sfail*, *dpfail*, and *dppass* parameters; one affects back-facing polygons, and the other affects front-facing polygons as well as other non-polygon primitives. `glStencilOp` sets both front and back stencil state to the same values. Use `glStencilOpSeparate` to set front and back stencil state to different values.

`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 *sfail* specifies what happens to the stencil buffer contents. The following eight 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_INCR_WRAP`
Increments the current stencil buffer value. Wraps stencil buffer value to zero when incrementing the maximum representable unsigned value.

`GL_DECR` Decrements the current stencil buffer value. Clamps to 0.

`GL_DECR_WRAP`
Decrements the current stencil buffer value. Wraps stencil buffer value to the maximum representable unsigned value when decrementing a stencil buffer value of zero.

`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 (*dppass*) or fail (*dpfail*) (see `glDepthFunc`). The actions are specified using the same eight symbolic constants as *sfail*. Note that *dpfail* is ignored when there is no depth buffer, or when the depth buffer is not enabled. In these cases, *sfail* and *dppass* specify stencil action when the stencil test fails and passes, respectively.

`GL_INVALID_ENUM` is generated if *sfail*, *dpfail*, or *dppass* is any value other than the eight defined constant values.

`GL_INVALID_OPERATION` is generated if `glStencilOp` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glTexCoordPointer` *size type stride pointer* [Function]

Define an array of texture coordinates.

size Specifies the number of coordinates per array element. Must be 1, 2, 3, or 4. The initial value is 4.

type Specifies the data type of each texture coordinate. Symbolic constants `GL_SHORT`, `GL_INT`, `GL_FLOAT`, or `GL_DOUBLE` are accepted. The initial value is `GL_FLOAT`.

stride Specifies the byte offset between consecutive texture coordinate sets. 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 texture coordinate set in the array. The initial value is 0.

`glTexCoordPointer` specifies the location and data format of an array of texture coordinates to use when rendering. *size* specifies the number of coordinates per texture coordinate set, and must be 1, 2, 3, or 4. *type* specifies the data type of each texture coordinate, and *stride* specifies the byte stride from one texture coordinate set to the next, allowing vertices and attributes to be packed into a single array or stored in separate arrays. (Single-array storage may be more efficient on some implementations; see `glInterleavedArrays`.)

If a non-zero named buffer object is bound to the `GL_ARRAY_BUFFER` target (see `glBindBuffer`) while a texture coordinate array is specified, *pointer* is treated as a byte offset into the buffer object's data store. Also, the buffer object binding (`GL_ARRAY_BUFFER_BINDING`) is saved as texture coordinate vertex array client-side state (`GL_TEXTURE_COORD_ARRAY_BUFFER_BINDING`).

When a texture coordinate array is specified, *size*, *type*, *stride*, and *pointer* are saved as client-side state, in addition to the current vertex array buffer object binding.

To enable and disable a texture coordinate array, call `glEnableClientState` and `glDisableClientState` with the argument `GL_TEXTURE_COORD_ARRAY`. If enabled, the texture coordinate array is used when `glArrayElement`, `glDrawArrays`, `glMultiDrawArrays`, `glDrawElements`, `glMultiDrawElements`, or `glDrawRangeElements` is called.

`GL_INVALID_VALUE` is generated if *size* is not 1, 2, 3, or 4.

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

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

`void glTexCoord1s` *s* [Function]
`void glTexCoord1i` *s* [Function]
`void glTexCoord1f` *s* [Function]
`void glTexCoord1d` *s* [Function]
`void glTexCoord2s` *s t* [Function]
`void glTexCoord2i` *s t* [Function]
`void glTexCoord2f` *s t* [Function]
`void glTexCoord2d` *s t* [Function]
`void glTexCoord3s` *s t r* [Function]
`void glTexCoord3i` *s t r* [Function]
`void glTexCoord3f` *s t r* [Function]
`void glTexCoord3d` *s t r* [Function]
`void glTexCoord4s` *s t r q* [Function]
`void glTexCoord4i` *s t r q* [Function]
`void glTexCoord4f` *s t r q* [Function]
`void glTexCoord4d` *s t r q* [Function]
`void glTexCoord1sv` *v* [Function]
`void glTexCoord1iv` *v* [Function]
`void glTexCoord1fv` *v* [Function]
`void glTexCoord1dv` *v* [Function]
`void glTexCoord2sv` *v* [Function]
`void glTexCoord2iv` *v* [Function]
`void glTexCoord2fv` *v* [Function]
`void glTexCoord2dv` *v* [Function]
`void glTexCoord3sv` *v* [Function]
`void glTexCoord3iv` *v* [Function]
`void glTexCoord3fv` *v* [Function]
`void glTexCoord3dv` *v* [Function]
`void glTexCoord4sv` *v* [Function]
`void glTexCoord4iv` *v* [Function]
`void glTexCoord4fv` *v* [Function]
`void glTexCoord4dv` *v* [Function]

Set the current texture coordinates.

s
t
r
q Specify *s*, *t*, *r*, and *q* texture coordinates. Not all parameters are present in all forms of the command.

`glTexCoord` specifies texture coordinates in one, two, three, or four dimensions. `glTexCoord1` sets the current texture coordinates to (*s*,001); a call to `glTexCoord2` sets them to (*s*,*t*01). Similarly, `glTexCoord3` specifies the texture coordinates as (*s*,*tr*1), and `glTexCoord4` defines all four components explicitly as (*s*,*trq*).

The current texture coordinates are part of the data that is associated with each vertex and with the current raster position. Initially, the values for s, t, r, and q are (0, 0, 0, 1).

`void glTexEnvf` *target pname param* [Function]
`void glTexEnvi` *target pname param* [Function]
`void glTexEnvfv` *target pname params* [Function]
`void glTexEnviv` *target pname params* [Function]

Set texture environment parameters.

target Specifies a texture environment. May be `GL_TEXTURE_ENV`, `GL_TEXTURE_FILTER_CONTROL` or `GL_POINT_SPRITE`.

pname Specifies the symbolic name of a single-valued texture environment parameter. May be either `GL_TEXTURE_ENV_MODE`, `GL_TEXTURE_LOD_BIAS`, `GL_COMBINE_RGB`, `GL_COMBINE_ALPHA`, `GL_SRC0_RGB`, `GL_SRC1_RGB`, `GL_SRC2_RGB`, `GL_SRC0_ALPHA`, `GL_SRC1_ALPHA`, `GL_SRC2_ALPHA`, `GL_OPERAND0_RGB`, `GL_OPERAND1_RGB`, `GL_OPERAND2_RGB`, `GL_OPERAND0_ALPHA`, `GL_OPERAND1_ALPHA`, `GL_OPERAND2_ALPHA`, `GL_RGB_SCALE`, `GL_ALPHA_SCALE`, or `GL_COORD_REPLACE`.

param Specifies a single symbolic constant, one of `GL_ADD`, `GL_ADD_SIGNED`, `GL_INTERPOLATE`, `GL_MODULATE`, `GL_DECAL`, `GL_BLEND`, `GL_REPLACE`, `GL_SUBTRACT`, `GL_COMBINE`, `GL_TEXTURE`, `GL_CONSTANT`, `GL_PRIMARY_COLOR`, `GL_PREVIOUS`, `GL_SRC_COLOR`, `GL_ONE_MINUS_SRC_COLOR`, `GL_SRC_ALPHA`, `GL_ONE_MINUS_SRC_ALPHA`, a single boolean value for the point sprite texture coordinate replacement, a single floating-point value for the texture level-of-detail bias, or 1.0, 2.0, or 4.0 when specifying the `GL_RGB_SCALE` or `GL_ALPHA_SCALE`.

A texture environment specifies how texture values are interpreted when a fragment is textured. When *target* is `GL_TEXTURE_FILTER_CONTROL`, *pname* must be `GL_TEXTURE_LOD_BIAS`. When *target* is `GL_TEXTURE_ENV`, *pname* can be `GL_TEXTURE_ENV_MODE`, `GL_TEXTURE_ENV_COLOR`, `GL_COMBINE_RGB`, `GL_COMBINE_ALPHA`, `GL_RGB_SCALE`, `GL_ALPHA_SCALE`, `GL_SRC0_RGB`, `GL_SRC1_RGB`, `GL_SRC2_RGB`, `GL_SRC0_ALPHA`, `GL_SRC1_ALPHA`, or `GL_SRC2_ALPHA`.

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

The following table shows the correspondence of filtered texture values R_t, G_t, B_t, A_t, L_t, I_t to texture source components. C_s and A_s are used by the texture functions described below.

Texture Base Internal Format
: `C_`s, `A_`s

`GL_ALPHA` : (0, 0, 0) , A_t

`GL_LUMINANCE`
: (L_t, L_t, L_t) , 1

GL_LUMINANCE_ALPHA
(L_t, L_t, L_t) , A_t

GL_INTENSITY
(I_t, I_t, I_t) , I_t

GL_RGB (R_t, G_t, B_t) , 1

GL_RGBA (R_t, G_t, B_t) , A_t

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 first five 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 p refers to the color computed from the previous texture stage (or the incoming fragment if processing texture stage 0), the subscript s to the texture source color, the subscript c to the texture environment color, and the subscript v indicates a value produced by the texture function.

Texture Base Internal Format
Value, GL_REPLACE Function , GL_MODULATE Function , GL_DECAL Function , GL_BLEND Function , GL_ADD Function

GL_ALPHA $C\_v=$, C_p, C_p, undefined , C_p, C_p

. $A\_v=$, A_s, A_pA_s, , $A\_v=A\_pA\_s$, A_pA_s

GL_LUMINANCE
$C\_v=$, C_s, C_pC_s, undefined , C_p(1-C_s,)+C_cC_s, C_p+C_s

(or 1) $A\_v=$, A_p, A_p, , A_p, A_p

GL_LUMINANCE_ALPHA
$C\_v=$, C_s, C_pC_s, undefined , C_p(1-C_s,)+C_cC_s, C_p+C_s

(or 2) $A\_v=$, A_s, A_pA_s, , A_pA_s, A_pA_s

GL_INTENSITY
$C\_v=$, C_s, C_pC_s, undefined , C_p(1-C_s,)+C_cC_s, C_p+C_s

. $A\_v=$, A_s, A_pA_s, , A_p(1-A_s,)+A_cA_s, A_p+A_s

GL_RGB $C\_v=$, C_s, C_pC_s, C_s, C_p(1-C_s,)+C_cC_s, C_p+C_s

(or 3) $A\_v=$, A_p, A_p, A_p, A_p, A_p

GL_RGBA $C\_v=$, C_s, C_pC_s, C_p(1-A_s,)+C_sA_s, C_p(1-C_s,)+C_cC_s, C_p+C_s

(or 4) $A\_v=$, A_s, A_pA_s, A_p, A_pA_s, A_pA_s

If *pname* is GL_TEXTURE_ENV_MODE, and *params* is GL_COMBINE, the form of the texture function depends on the values of GL_COMBINE_RGB and GL_COMBINE_ALPHA.

The following describes how the texture sources, as specified by GL_SRC0_RGB, GL_SRC1_RGB, GL_SRC2_RGB, GL_SRC0_ALPHA, GL_SRC1_ALPHA, and GL_SRC2_ALPHA, are combined to produce a final texture color. In the following tables, GL_SRC0_c is

represented by *Arg0*, `GL_SRC1_c` is represented by *Arg1*, and `GL_SRC2_c` is represented by *Arg2*.

`GL_COMBINE_RGB` accepts any of `GL_REPLACE`, `GL_MODULATE`, `GL_ADD`, `GL_ADD_SIGNED`, `GL_INTERPOLATE`, `GL_SUBTRACT`, `GL_DOT3_RGB`, or `GL_DOT3_RGBA`.

`GL_COMBINE_RGB`
Texture Function

`GL_REPLACE`
Arg0

`GL_MODULATE`
*Arg0***Arg1*

`GL_ADD` *Arg0*+*Arg1*

`GL_ADD_SIGNED`
Arg0+*Arg1*-0.5

`GL_INTERPOLATE`
*Arg0***Arg2*+*Arg1*(1-*Arg2*,)

`GL_SUBTRACT`
Arg0-*Arg1*

`GL_DOT3_RGB` or `GL_DOT3_RGBA`
4(((*Arg0*_*r*,-0.5,)(*Arg1*_*r*,-0.5,),)+((*Arg0*_*g*,-0.5,)(*Arg1*_*g*,-0.5,),)+((*Arg0*_*b*,-0.5,)(*Arg1*_*b*,-0.5,),),)

The scalar results for `GL_DOT3_RGB` and `GL_DOT3_RGBA` are placed into each of the 3 (RGB) or 4 (RGBA) components on output.

Likewise, `GL_COMBINE_ALPHA` accepts any of `GL_REPLACE`, `GL_MODULATE`, `GL_ADD`, `GL_ADD_SIGNED`, `GL_INTERPOLATE`, or `GL_SUBTRACT`. The following table describes how alpha values are combined:

`GL_COMBINE_ALPHA`
Texture Function

`GL_REPLACE`
Arg0

`GL_MODULATE`
*Arg0***Arg1*

`GL_ADD` *Arg0*+*Arg1*

`GL_ADD_SIGNED`
Arg0+*Arg1*-0.5

`GL_INTERPOLATE`
*Arg0***Arg2*+*Arg1*(1-*Arg2*,)

`GL_SUBTRACT`
Arg0-*Arg1*

In the following tables, the value C_s represents the color sampled from the currently bound texture, C_c represents the constant texture-environment color, C_f represents the primary color of the incoming fragment, and C_p represents the color computed from the previous texture stage or C_f if processing texture stage 0. Likewise, A_s, A_c, A_f, and A_p represent the respective alpha values.

The following table describes the values assigned to *Arg0*, *Arg1*, and *Arg2* based upon the RGB sources and operands:

`GL_SRCn_RGB`
: `GL_OPERANDn_RGB`, **Argument Value**

`GL_TEXTURE`
: `GL_SRC_COLOR`, C_s,

. `GL_ONE_MINUS_SRC_COLOR`, 1-C_s,

. `GL_SRC_ALPHA`, A_s,

. `GL_ONE_MINUS_SRC_ALPHA`, 1-A_s,

`GL_TEXTUREn`
: `GL_SRC_COLOR`, C_s,

. `GL_ONE_MINUS_SRC_COLOR`, 1-C_s,

. `GL_SRC_ALPHA`, A_s,

. `GL_ONE_MINUS_SRC_ALPHA`, 1-A_s,

`GL_CONSTANT`
: `GL_SRC_COLOR`, C_c,

. `GL_ONE_MINUS_SRC_COLOR`, 1-C_c,

. `GL_SRC_ALPHA`, A_c,

. `GL_ONE_MINUS_SRC_ALPHA`, 1-A_c,

`GL_PRIMARY_COLOR`
: `GL_SRC_COLOR`, C_f,

. `GL_ONE_MINUS_SRC_COLOR`, 1-C_f,

. `GL_SRC_ALPHA`, A_f,

. `GL_ONE_MINUS_SRC_ALPHA`, 1-A_f,

`GL_PREVIOUS`
: `GL_SRC_COLOR`, C_p,

. `GL_ONE_MINUS_SRC_COLOR`, 1-C_p,

. `GL_SRC_ALPHA`, A_p,

. `GL_ONE_MINUS_SRC_ALPHA`, 1-A_p,

For `GL_TEXTUREn` sources, C_s and A_s represent the color and alpha, respectively, produced from texture stage n.

The follow table describes the values assigned to *Arg0*, *Arg1*, and *Arg2* based upon the alpha sources and operands:

GL_SRCn_ALPHA
GL_OPERANDn_ALPHA, **Argument Value**

GL_TEXTURE
GL_SRC_ALPHA, A_s,

. GL_ONE_MINUS_SRC_ALPHA, 1-A_s,

GL_TEXTUREn
GL_SRC_ALPHA, A_s,

. GL_ONE_MINUS_SRC_ALPHA, 1-A_s,

GL_CONSTANT
GL_SRC_ALPHA, A_c,

. GL_ONE_MINUS_SRC_ALPHA, 1-A_c,

GL_PRIMARY_COLOR
GL_SRC_ALPHA, A_f,

. GL_ONE_MINUS_SRC_ALPHA, 1-A_f,

GL_PREVIOUS
GL_SRC_ALPHA, A_p,

. GL_ONE_MINUS_SRC_ALPHA, 1-A_p,

The RGB and alpha results of the texture function are multipled by the values of GL_RGB_SCALE and GL_ALPHA_SCALE, respectively, and clamped to the range [0,1].

If *pname* is GL_TEXTURE_ENV_COLOR, *params* is a pointer to an array that holds an RGBA color consisting of four values. Integer color components are interpreted linearly such that the most positive integer maps to 1.0, and the most negative integer maps to -1.0. The values are clamped to the range [0,1] when they are specified. C_c takes these four values.

If *pname* is GL_TEXTURE_LOD_BIAS, the value specified is added to the texture level-of-detail parameter, that selects which mipmap, or mipmaps depending upon the selected GL_TEXTURE_MIN_FILTER, will be sampled.

GL_TEXTURE_ENV_MODE defaults to GL_MODULATE and GL_TEXTURE_ENV_COLOR defaults to (0, 0, 0, 0).

If *target* is GL_POINT_SPRITE and *pname* is GL_COORD_REPLACE, the boolean value specified is used to either enable or disable point sprite texture coordinate replacement. The default value is GL_FALSE.

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

GL_INVALID_VALUE is generated if the *params* value for GL_RGB_SCALE or GL_ALPHA_SCALE are not one of 1.0, 2.0, or 4.0.

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

`void glTexGeni` *coord pname param* [Function]
`void glTexGenf` *coord pname param* [Function]
`void glTexGend` *coord pname param* [Function]
`void glTexGeniv` *coord pname params* [Function]
`void glTexGenfv` *coord pname params* [Function]
`void glTexGendv` *coord pname params* [Function]

Control the generation of texture coordinates.

coord Specifies a texture coordinate. Must be one of `GL_S`, `GL_T`, `GL_R`, or `GL_Q`.

pname Specifies the symbolic name of the texture-coordinate generation function. Must be `GL_TEXTURE_GEN_MODE`.

param Specifies a single-valued texture generation parameter, one of `GL_OBJECT_LINEAR`, `GL_EYE_LINEAR`, `GL_SPHERE_MAP`, `GL_NORMAL_MAP`, or `GL_REFLECTION_MAP`.

`glTexGen` selects a texture-coordinate generation function or supplies coefficients for one of the functions. *coord* names one of the (s, t, r, q) texture coordinates; it must be one of the symbols `GL_S`, `GL_T`, `GL_R`, or `GL_Q`. *pname* must be one of three symbolic constants: `GL_TEXTURE_GEN_MODE`, `GL_OBJECT_PLANE`, or `GL_EYE_PLANE`. If *pname* is `GL_TEXTURE_GEN_MODE`, then *params* chooses a mode, one of `GL_OBJECT_LINEAR`, `GL_EYE_LINEAR`, `GL_SPHERE_MAP`, `GL_NORMAL_MAP`, or `GL_REFLECTION_MAP`. If *pname* is either `GL_OBJECT_PLANE` or `GL_EYE_PLANE`, *params* contains coefficients for the corresponding texture generation function.

If the texture generation function is `GL_OBJECT_LINEAR`, the function

$g = p\_1 x\_o + p\_2 y\_o + p\_3 z\_o + p\_4 w\_o$

is used, where g is the value computed for the coordinate named in *coord*, p_1, p_2, p_3, and p_4 are the four values supplied in *params*, and x_o, y_o, z_o, and w_o are the object coordinates of the vertex. This function can be used, for example, to texture-map terrain using sea level as a reference plane (defined by p_1, p_2, p_3, and p_4). The altitude of a terrain vertex is computed by the `GL_OBJECT_LINEAR` coordinate generation function as its distance from sea level; that altitude can then be used to index the texture image to map white snow onto peaks and green grass onto foothills.

If the texture generation function is `GL_EYE_LINEAR`, the function

$g = p\_1 \hat{} x\_e + p\_2 \hat{} y\_e + p\_3 \hat{} z\_e + p\_4 \hat{} w\_e$

is used, where

$(p\_1 \hat{} p\_2 \hat{} p\_3 \hat{} p\_4 \hat{}) = (p\_1 p\_2 p\_3 p\_4) M \hat{} -1$

and x_e, y_e, z_e, and w_e are the eye coordinates of the vertex, p_1, p_2, p_3, and p_4 are the values supplied in *params*, and M is the modelview matrix when `glTexGen` is invoked. If M is poorly conditioned or singular, texture coordinates generated by the resulting function may be inaccurate or undefined.

Note that the values in *params* define a reference plane in eye coordinates. The modelview matrix that is applied to them may not be the same one in effect when the polygon vertices are transformed. This function establishes a field of texture coordinates that can produce dynamic contour lines on moving objects.

If the texture generation function is GL_SPHERE_MAP and *coord* is either GL_S or GL_T, *s* and *t* texture coordinates are generated as follows. Let *u* be the unit vector pointing from the origin to the polygon vertex (in eye coordinates). Let *n* sup prime be the current normal, after transformation to eye coordinates. Let

f=($f\_xf\_yf\_z$,)^T be the reflection vector such that

f=u-2n^n^,^Tu

Finally, let m=2(f_x,^2+f_y,^2+(f_z+1,)^2,). Then the values assigned to the *s* and *t* texture coordinates are

s=f_x/m+1/2

t=f_y/m+1/2

To enable or disable a texture-coordinate generation function, call glEnable or glDisable with one of the symbolic texture-coordinate names (GL_TEXTURE_GEN_S, GL_TEXTURE_GEN_T, GL_TEXTURE_GEN_R, or GL_TEXTURE_GEN_Q) as the argument. When enabled, the specified texture coordinate is computed according to the generating function associated with that coordinate. When disabled, subsequent vertices take the specified texture coordinate from the current set of texture coordinates. Initially, all texture generation functions are set to GL_EYE_LINEAR and are disabled. Both *s* plane equations are (1, 0, 0, 0), both *t* plane equations are (0, 1, 0, 0), and all *r* and *q* plane equations are (0, 0, 0, 0).

When the ARB_multitexture extension is supported, glTexGen sets the texture generation parameters for the currently active texture unit, selected with glActiveTexture.

GL_INVALID_ENUM is generated when *coord* or *pname* is not an accepted defined value, or when *pname* is GL_TEXTURE_GEN_MODE and *params* is not an accepted defined value.

GL_INVALID_ENUM is generated when *pname* is GL_TEXTURE_GEN_MODE, *params* is GL_SPHERE_MAP, and *coord* is either GL_R or GL_Q.

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

`void glTexImage1D` *target level internalFormat width border format type data* [Function]

Specify a one-dimensional texture image.

target Specifies the target texture. Must be GL_TEXTURE_1D or GL_PROXY_TEXTURE_1D.

level Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.

internalFormat
Specifies the number of color components in the texture. Must be 1, 2, 3, or 4, or one of the following symbolic constants: GL_ALPHA, GL_ALPHA4, GL_ALPHA8, GL_ALPHA12, GL_ALPHA16, GL_COMPRESSED_ALPHA, GL_COMPRESSED_LUMINANCE, GL_COMPRESSED_LUMINANCE_ALPHA, GL_COMPRESSED_INTENSITY, GL_COMPRESSED_RGB, GL_COMPRESSED_RGBA, GL_DEPTH_COMPONENT, GL_DEPTH_COMPONENT16, GL_DEPTH_COMPONENT24, GL_DEPTH_COMPONENT32, GL_LUMINANCE,

GL_LUMINANCE4, GL_LUMINANCE8, GL_LUMINANCE12, GL_LUMINANCE16, GL_LUMINANCE_ALPHA, GL_LUMINANCE4_ALPHA4, GL_LUMINANCE6_ALPHA2, GL_LUMINANCE8_ALPHA8, GL_LUMINANCE12_ALPHA4, GL_LUMINANCE12_ALPHA12, GL_LUMINANCE16_ALPHA16, GL_INTENSITY, GL_INTENSITY4, GL_INTENSITY8, GL_INTENSITY12, GL_INTENSITY16, GL_R3_G3_B2, GL_RGB, GL_RGB4, GL_RGB5, GL_RGB8, GL_RGB10, GL_RGB12, GL_RGB16, GL_RGBA, GL_RGBA2, GL_RGBA4, GL_RGB5_A1, GL_RGBA8, GL_RGB10_A2, GL_RGBA12, GL_RGBA16, GL_SLUMINANCE, GL_SLUMINANCE8, GL_SLUMINANCE_ALPHA, GL_SLUMINANCE8_ALPHA8, GL_SRGB, GL_SRGB8, GL_SRGB_ALPHA, or GL_SRGB8_ALPHA8.

width Specifies the width of the texture image including the border if any. If the GL version does not support non-power-of-two sizes, this value must be 2^*n*+2(*border*,) for some integer *n*. All implementations support texture images that are at least 64 texels wide. The height of the 1D texture image is 1.

border Specifies the width of the border. Must be either 0 or 1.

format Specifies the format of the pixel data. The following symbolic values are accepted: GL_COLOR_INDEX, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_BGR, GL_RGBA, GL_BGRA, 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_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, GL_FLOAT, GL_UNSIGNED_BYTE_3_3_2, GL_UNSIGNED_BYTE_2_3_3_REV, GL_UNSIGNED_SHORT_5_6_5, GL_UNSIGNED_SHORT_5_6_5_REV, GL_UNSIGNED_SHORT_4_4_4_4, GL_UNSIGNED_SHORT_4_4_4_4_REV, GL_UNSIGNED_SHORT_5_5_5_1, GL_UNSIGNED_SHORT_1_5_5_5_REV, GL_UNSIGNED_INT_8_8_8_8, GL_UNSIGNED_INT_8_8_8_8_REV, GL_UNSIGNED_INT_10_10_10_2, and GL_UNSIGNED_INT_2_10_10_10_REV.

data Specifies a pointer to the image data in memory.

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. To enable and disable one-dimensional texturing, call glEnable and glDisable with argument GL_TEXTURE_1D.

Texture images are defined with glTexImage1D. The arguments describe the parameters of the texture image, such as width, width of the border, level-of-detail number (see glTexParameter), and the internal resolution and format used to store the image. The last three arguments describe how the image is represented in memory; they are identical to the pixel formats used for glDrawPixels.

If *target* is GL_PROXY_TEXTURE_1D, no data is read from *data*, but all of the texture image state is recalculated, checked for consistency, and checked against the implementation's capabilities. If the implementation cannot handle a texture of the requested texture size, it sets all of the image state to 0, but does not generate an error (see glGetError). To query for an entire mipmap array, use an image array level greater than or equal to 1.

If *target* is GL_TEXTURE_1D, data is read from *data* as a sequence of signed or unsigned bytes, shorts, or longs, or single-precision floating-point values, depending on *type*. These values are grouped into sets of one, two, three, or four values, depending on *format*, to form elements. If *type* is GL_BITMAP, the data is considered as a string of unsigned bytes (and *format* must be GL_COLOR_INDEX). Each data byte is treated as eight 1-bit elements, with bit ordering determined by GL_UNPACK_LSB_FIRST (see glPixelStore).

If a non-zero named buffer object is bound to the GL_PIXEL_UNPACK_BUFFER target (see glBindBuffer) while a texture image is specified, *data* is treated as a byte offset into the buffer object's data store.

The first element corresponds to the left end of the texture array. Subsequent elements progress left-to-right through the remaining texels in the texture array. The final element corresponds to the right end of the texture array.

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

GL_COLOR_INDEX
: Each element is a single value, a color index. The GL converts it to fixed point (with an unspecified number of zero bits to the right of the binary point), shifted left or right depending on the value and sign of GL_INDEX_SHIFT, and added to GL_INDEX_OFFSET (see glPixelTransfer). The resulting index is converted to a set of color components using the GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, and GL_PIXEL_MAP_I_TO_A tables, and clamped to the range [0,1].

GL_RED
: Each element is a single red component. The GL converts it to floating point and assembles it into an RGBA element by attaching 0 for green and blue, and 1 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see glPixelTransfer).

GL_GREEN
: Each element is a single green component. The GL converts it to floating point and assembles it into an RGBA element by attaching 0 for red and blue, and 1 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see glPixelTransfer).

GL_BLUE
: Each element is a single blue component. The GL converts it to floating point and assembles it into an RGBA element by attaching 0 for red and green, and 1 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see glPixelTransfer).

GL_ALPHA
: Each element is a single alpha component. The GL converts it to floating point and assembles it into an RGBA element by attaching 0 for red, green, and blue. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see glPixelTransfer).

GL_INTENSITY
: Each element is a single intensity value. The GL converts it to floating point, then assembles it into an RGBA element by replicating the intensity value three times for red, green, blue, and alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see glPixelTransfer).

GL_RGB

GL_BGR
: Each element is an RGB triple. The GL converts it to floating point and assembles it into an RGBA element by attaching 1 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see glPixelTransfer).

GL_RGBA

GL_BGRA
: Each element contains all four components. Each component is multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see glPixelTransfer).

GL_LUMINANCE
: Each element is a single luminance value. The GL converts it to floating point, then assembles it into an RGBA element by replicating the luminance value three times for red, green, and blue and attaching 1 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see glPixelTransfer).

GL_LUMINANCE_ALPHA
: Each element is a luminance/alpha pair. The GL converts it to floating point, then assembles it into an RGBA element by replicating the luminance value three times for red, green, and blue. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see glPixelTransfer).

GL_DEPTH_COMPONENT
: Each element is a single depth value. The GL converts it to floating point, multiplies by the signed scale factor GL_DEPTH_SCALE, adds the signed bias GL_DEPTH_BIAS, and clamps to the range [0,1] (see glPixelTransfer).

Refer to the glDrawPixels reference page for a description of the acceptable values for the *type* parameter.

If an application wants to store the texture at a certain resolution or in a certain format, it can request the resolution and format with *internalFormat*. The GL will choose an internal representation that closely approximates that requested by *internalFormat*, but it may not match exactly. (The representations specified by GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_RGB, and GL_RGBA must match exactly. The numeric values 1, 2, 3, and 4 may also be used to specify the above representations.)

If the *internalFormat* parameter is one of the generic compressed formats, GL_COMPRESSED_ALPHA, GL_COMPRESSED_INTENSITY, GL_COMPRESSED_LUMINANCE, GL_COMPRESSED_LUMINANCE_ALPHA, GL_COMPRESSED_RGB, or GL_COMPRESSED_RGBA, the GL will replace the internal format with the symbolic constant for a specific internal format and compress the texture before storage. If no corresponding internal format is available, or the GL can not compress that image for any reason, the internal format is instead replaced with a corresponding base internal format.

If the *internalFormat* parameter is GL_SRGB, GL_SRGB8, GL_SRGB_ALPHA, GL_SRGB8_ALPHA8, GL_SLUMINANCE, GL_SLUMINANCE8, GL_SLUMINANCE_ALPHA, or GL_SLUMINANCE8_ALPHA8, the texture is treated as if the red, green, blue, or luminance components are encoded in the sRGB color space. Any alpha component is left unchanged. The conversion from the sRGB encoded component c_s to a linear component c_l is:

c_l={(c_s/12.92 if c_s0.04045), ((c_s+0.055/1.055)^2.4 if c_s>0.04045)

Assume c_s is the sRGB component in the range [0,1].

Use the GL_PROXY_TEXTURE_1D target to try out a resolution and format. The implementation will update and recompute its best match for the requested storage resolution and format. To then query this state, call glGetTexLevelParameter. If the texture cannot be accommodated, texture state is set to 0.

A one-component texture image uses only the red component of the RGBA color from *data*. A two-component image uses the R and A values. A three-component image uses the R, G, and B values. A four-component image uses all of the RGBA components.

Depth textures can be treated as LUMINANCE, INTENSITY or ALPHA textures during texture filtering and application. Image-based shadowingcanbe enabledby-comparing texture r coordinates to depth texture values to generate a boolean result. See glTexParameter for details on texture comparison.

GL_INVALID_ENUM is generated if *target* is not GL_TEXTURE_1D or GL_PROXY_TEXTURE_1D.

GL_INVALID_ENUM is generated if *format* is not an accepted format constant. Format constants other than GL_STENCIL_INDEX are accepted.

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

GL_INVALID_ENUM is generated if *type* is GL_BITMAP and *format* is not GL_COLOR_INDEX.

GL_INVALID_VALUE is generated if *level* is less than 0.

GL_INVALID_VALUE may be generated if *level* is greater than $log\_2(max,)$, where *max* is the returned value of GL_MAX_TEXTURE_SIZE.

GL_INVALID_VALUE is generated if *internalFormat* is not 1, 2, 3, 4, or one of the accepted resolution and format symbolic constants.

GL_INVALID_VALUE is generated if *width* is less than 0 or greater than 2 + GL_MAX_TEXTURE_SIZE.

GL_INVALID_VALUE is generated if non-power-of-two textures are not supported and the *width* cannot be represented as $2^n+2(border,)$ for some integer value of n.

GL_INVALID_VALUE is generated if *border* is not 0 or 1.

GL_INVALID_OPERATION is generated if *type* is one of GL_UNSIGNED_BYTE_3_3_2, GL_UNSIGNED_BYTE_2_3_3_REV, GL_UNSIGNED_SHORT_5_6_5, or GL_UNSIGNED_SHORT_5_6_5_REV and *format* is not GL_RGB.

GL_INVALID_OPERATION is generated if *type* is one of GL_UNSIGNED_SHORT_4_4_4_4, GL_UNSIGNED_SHORT_4_4_4_4_REV, GL_UNSIGNED_SHORT_5_5_5_1, GL_UNSIGNED_SHORT_1_5_5_5_REV, GL_UNSIGNED_INT_8_8_8_8, GL_UNSIGNED_INT_8_8_8_8_REV, GL_UNSIGNED_INT_10_10_10_2, or GL_UNSIGNED_INT_2_10_10_10_REV and *format* is neither GL_RGBA nor GL_BGRA.

GL_INVALID_OPERATION is generated if *format* is GL_DEPTH_COMPONENT and *internalFormat* is not GL_DEPTH_COMPONENT, GL_DEPTH_COMPONENT16, GL_DEPTH_COMPONENT24, or GL_DEPTH_COMPONENT32.

GL_INVALID_OPERATION is generated if *internalFormat* is GL_DEPTH_COMPONENT, GL_DEPTH_COMPONENT16, GL_DEPTH_COMPONENT24, or GL_DEPTH_COMPONENT32, and *format* is not GL_DEPTH_COMPONENT.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and the buffer object's data store is currently mapped.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and *data* is not evenly divisible into the number of bytes needed to store in memory a datum indicated by *type*.

GL_INVALID_OPERATION is generated if glTexImage1D is executed between the execution of glBegin and the corresponding execution of glEnd.

`void glTexImage2D` *target level internalFormat width height border format type data* [Function]

Specify a two-dimensional texture image.

target Specifies the target texture. Must be GL_TEXTURE_2D, GL_PROXY_TEXTURE_2D, GL_TEXTURE_CUBE_MAP_POSITIVE_X, GL_TEXTURE_CUBE_MAP_NEGATIVE_X, GL_TEXTURE_CUBE_MAP_POSITIVE_Y, GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, GL_TEXTURE_CUBE_MAP_POSITIVE_Z, GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, or GL_PROXY_TEXTURE_CUBE_MAP.

level Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.

internalFormat
Specifies the number of color components in the texture. Must be 1, 2, 3, or 4, or one of the following symbolic constants: GL_ALPHA, GL_ALPHA4, GL_ALPHA8, GL_ALPHA12, GL_ALPHA16, GL_COMPRESSED_ALPHA, GL_COMPRESSED_LUMINANCE, GL_COMPRESSED_LUMINANCE_ALPHA, GL_COMPRESSED_INTENSITY, GL_COMPRESSED_RGB, GL_COMPRESSED_RGBA, GL_DEPTH_COMPONENT, GL_DEPTH_COMPONENT16,

GL_DEPTH_COMPONENT24, GL_DEPTH_COMPONENT32, GL_LUMINANCE, GL_LUMINANCE4, GL_LUMINANCE8, GL_LUMINANCE12, GL_LUMINANCE16, GL_LUMINANCE_ALPHA, GL_LUMINANCE4_ALPHA4, GL_LUMINANCE6_ALPHA2, GL_LUMINANCE8_ALPHA8, GL_LUMINANCE12_ALPHA4, GL_LUMINANCE12_ALPHA12, GL_LUMINANCE16_ALPHA16, GL_INTENSITY, GL_INTENSITY4, GL_INTENSITY8, GL_INTENSITY12, GL_INTENSITY16, GL_R3_G3_B2, GL_RGB, GL_RGB4, GL_RGB5, GL_RGB8, GL_RGB10, GL_RGB12, GL_RGB16, GL_RGBA, GL_RGBA2, GL_RGBA4, GL_RGB5_A1, GL_RGBA8, GL_RGB10_A2, GL_RGBA12, GL_RGBA16, GL_SLUMINANCE, GL_SLUMINANCE8, GL_SLUMINANCE_ALPHA, GL_SLUMINANCE8_ALPHA8, GL_SRGB, GL_SRGB8, GL_SRGB_ALPHA, or GL_SRGB8_ALPHA8.

width Specifies the width of the texture image including the border if any. If the GL version does not support non-power-of-two sizes, this value must be 2^*n*+2(*border*,) for some integer *n*. All implementations support texture images that are at least 64 texels wide.

height Specifies the height of the texture image including the border if any. If the GL version does not support non-power-of-two sizes, this value must be 2^*m*+2(*border*,) for some integer *m*. All implementations support texture images that are at least 64 texels high.

border Specifies the width of the border. Must be either 0 or 1.

format Specifies the format of the pixel data. The following symbolic values are accepted: GL_COLOR_INDEX, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_BGR, GL_RGBA, GL_BGRA, 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_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, GL_FLOAT, GL_UNSIGNED_BYTE_3_3_2, GL_UNSIGNED_BYTE_2_3_3_REV, GL_UNSIGNED_SHORT_5_6_5, GL_UNSIGNED_SHORT_5_6_5_REV, GL_UNSIGNED_SHORT_4_4_4_4, GL_UNSIGNED_SHORT_4_4_4_4_REV, GL_UNSIGNED_SHORT_5_5_5_1, GL_UNSIGNED_SHORT_1_5_5_5_REV, GL_UNSIGNED_INT_8_8_8_8, GL_UNSIGNED_INT_8_8_8_8_REV, GL_UNSIGNED_INT_10_10_10_2, and GL_UNSIGNED_INT_2_10_10_10_REV.

data Specifies a pointer to the image data in memory.

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. To enable and disable two-dimensional texturing, call glEnable and glDisable with argument GL_TEXTURE_2D. To enable and disable texturing using cube-mapped texture, call glEnable and glDisable with argument GL_TEXTURE_CUBE_MAP.

To define texture images, call glTexImage2D. The arguments describe the parameters of the texture image, such as height, width, width of the border, level-of-detail number (see glTexParameter), and number of color components provided. The last three arguments describe how the image is represented in memory; they are identical to the pixel formats used for glDrawPixels.

If *target* is GL_PROXY_TEXTURE_2D or GL_PROXY_TEXTURE_CUBE_MAP, no data is read from *data*, but all of the texture image state is recalculated, checked for consistency, and checked against the implementation's capabilities. If the implementation cannot handle a texture of the requested texture size, it sets all of the image state to 0, but does not generate an error (see glGetError). To query for an entire mipmap array, use an image array level greater than or equal to 1.

If *target* is GL_TEXTURE_2D, or one of the GL_TEXTURE_CUBE_MAP targets, data is read from *data* as a sequence of signed or unsigned bytes, shorts, or longs, or single-precision floating-point values, depending on *type*. These values are grouped into sets of one, two, three, or four values, depending on *format*, to form elements. If *type* is GL_BITMAP, the data is considered as a string of unsigned bytes (and *format* must be GL_COLOR_INDEX). Each data byte is treated as eight 1-bit elements, with bit ordering determined by GL_UNPACK_LSB_FIRST (see glPixelStore).

If a non-zero named buffer object is bound to the GL_PIXEL_UNPACK_BUFFER target (see glBindBuffer) while a texture image is specified, *data* is treated as a byte offset into the buffer object's data store.

The first element corresponds to the lower left corner of the texture image. Subsequent elements progress left-to-right through the remaining texels in the lowest row of the texture image, and then in successively higher rows of the texture image. The final element corresponds to the upper right corner of the texture image.

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

GL_COLOR_INDEX
: Each element is a single value, a color index. The GL converts it to fixed point (with an unspecified number of zero bits to the right of the binary point), shifted left or right depending on the value and sign of GL_INDEX_SHIFT, and added to GL_INDEX_OFFSET (see glPixelTransfer). The resulting index is converted to a set of color components using the GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, and GL_PIXEL_MAP_I_TO_A tables, and clamped to the range [0,1].

GL_RED
: Each element is a single red component. The GL converts it to floating point and assembles it into an RGBA element by attaching 0 for green and blue, and 1 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see glPixelTransfer).

GL_GREEN
: Each element is a single green component. The GL converts it to floating point and assembles it into an RGBA element by attaching 0 for red and blue, and 1 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see glPixelTransfer).

GL_BLUE
: Each element is a single blue component. The GL converts it to floating point and assembles it into an RGBA element by attaching 0 for red and green, and 1 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see glPixelTransfer).

GL_ALPHA Each element is a single alpha component. The GL converts it to floating point and assembles it into an RGBA element by attaching 0 for red, green, and blue. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see glPixelTransfer).

GL_INTENSITY
Each element is a single intensity value. The GL converts it to floating point, then assembles it into an RGBA element by replicating the intensity value three times for red, green, blue, and alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see glPixelTransfer).

GL_RGB

GL_BGR Each element is an RGB triple. The GL converts it to floating point and assembles it into an RGBA element by attaching 1 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see glPixelTransfer).

GL_RGBA

GL_BGRA Each element contains all four components. Each component is multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see glPixelTransfer).

GL_LUMINANCE
Each element is a single luminance value. The GL converts it to floating point, then assembles it into an RGBA element by replicating the luminance value three times for red, green, and blue and attaching 1 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see glPixelTransfer).

GL_LUMINANCE_ALPHA
Each element is a luminance/alpha pair. The GL converts it to floating point, then assembles it into an RGBA element by replicating the luminance value three times for red, green, and blue. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see glPixelTransfer).

GL_DEPTH_COMPONENT
Each element is a single depth value. The GL converts it to floating point, multiplies by the signed scale factor GL_DEPTH_SCALE, adds the signed bias GL_DEPTH_BIAS, and clamps to the range [0,1] (see glPixelTransfer).

Refer to the glDrawPixels reference page for a description of the acceptable values for the *type* parameter.

If an application wants to store the texture at a certain resolution or in a certain format, it can request the resolution and format with *internalFormat*. The GL will choose an internal representation that closely approximates that requested by *internalFormat*, but it may not match exactly. (The representations specified by GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_RGB, and GL_RGBA must match exactly. The numeric values 1, 2, 3, and 4 may also be used to specify the above representations.)

If the *internalFormat* parameter is one of the generic compressed formats, GL_COMPRESSED_ALPHA, GL_COMPRESSED_INTENSITY, GL_COMPRESSED_LUMINANCE, GL_COMPRESSED_LUMINANCE_ALPHA, GL_COMPRESSED_RGB, or GL_COMPRESSED_RGBA, the GL will replace the internal format with the symbolic constant for a specific internal format and compress the texture before storage. If no corresponding internal format is available, or the GL can not compress that image for any reason, the internal format is instead replaced with a corresponding base internal format.

If the *internalFormat* parameter is GL_SRGB, GL_SRGB8, GL_SRGB_ALPHA, GL_SRGB8_ALPHA8, GL_SLUMINANCE, GL_SLUMINANCE8, GL_SLUMINANCE_ALPHA, or GL_SLUMINANCE8_ALPHA8, the texture is treated as if the red, green, blue, or luminance components are encoded in the sRGB color space. Any alpha component is left unchanged. The conversion from the sRGB encoded component c_s to a linear component c_l is:

c_l={(c_s/12.92 if c_s0.04045), ((c_s+0.055/1.055)^2.4 if c_s>0.04045)

Assume c_s is the sRGB component in the range [0,1].

Use the GL_PROXY_TEXTURE_2D or GL_PROXY_TEXTURE_CUBE_MAP target to try out a resolution and format. The implementation will update and recompute its best match for the requested storage resolution and format. To then query this state, call glGetTexLevelParameter. If the texture cannot be accommodated, texture state is set to 0.

A one-component texture image uses only the red component of the RGBA color extracted from *data*. A two-component image uses the R and A values. A three-component image uses the R, G, and B values. A four-component image uses all of the RGBA components.

Depth textures can be treated as LUMINANCE, INTENSITY or ALPHA textures during texture filtering and application. Image-based shadowingcanbe enabledby-comparing texture r coordinates to depth texture values to generate a boolean result. See glTexParameter for details on texture comparison.

GL_INVALID_ENUM is generated if *target* is not GL_TEXTURE_2D, GL_PROXY_TEXTURE_2D, GL_PROXY_TEXTURE_CUBE_MAP, GL_TEXTURE_CUBE_MAP_POSITIVE_X, GL_TEXTURE_CUBE_MAP_NEGATIVE_X, GL_TEXTURE_CUBE_MAP_POSITIVE_Y, GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, GL_TEXTURE_CUBE_MAP_POSITIVE_Z, or GL_TEXTURE_CUBE_MAP_NEGATIVE_Z.

GL_INVALID_ENUM is generated if *target* is one of the six cube map 2D image targets and the width and height parameters are not equal.

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

GL_INVALID_ENUM is generated if *type* is GL_BITMAP and *format* is not GL_COLOR_INDEX.

GL_INVALID_VALUE is generated if *width* or *height* is less than 0 or greater than 2 + GL_MAX_TEXTURE_SIZE.

GL_INVALID_VALUE is generated if *level* is less than 0.

GL_INVALID_VALUE may be generated if *level* is greater than *log_2(max,)*, where *max* is the returned value of GL_MAX_TEXTURE_SIZE.

GL_INVALID_VALUE is generated if *internalFormat* is not 1, 2, 3, 4, or one of the accepted resolution and format symbolic constants.

GL_INVALID_VALUE is generated if *width* or *height* is less than 0 or greater than 2 + GL_MAX_TEXTURE_SIZE.

GL_INVALID_VALUE is generated if non-power-of-two textures are not supported and the *width* or *height* cannot be represented as 2^*k*+2(*border*,) for some integer value of *k*.

GL_INVALID_VALUE is generated if *border* is not 0 or 1.

GL_INVALID_OPERATION is generated if *type* is one of GL_UNSIGNED_BYTE_3_3_2, GL_UNSIGNED_BYTE_2_3_3_REV, GL_UNSIGNED_SHORT_5_6_5, or GL_UNSIGNED_SHORT_5_6_5_REV and *format* is not GL_RGB.

GL_INVALID_OPERATION is generated if *type* is one of GL_UNSIGNED_SHORT_4_4_4_4, GL_UNSIGNED_SHORT_4_4_4_4_REV, GL_UNSIGNED_SHORT_5_5_5_1, GL_UNSIGNED_SHORT_1_5_5_5_REV, GL_UNSIGNED_INT_8_8_8_8, GL_UNSIGNED_INT_8_8_8_8_REV, GL_UNSIGNED_INT_10_10_10_2, or GL_UNSIGNED_INT_2_10_10_10_REV and *format* is neither GL_RGBA nor GL_BGRA.

GL_INVALID_OPERATION is generated if *target* is not GL_TEXTURE_2D or GL_PROXY_TEXTURE_2D and *internalFormat* is GL_DEPTH_COMPONENT, GL_DEPTH_COMPONENT16, GL_DEPTH_COMPONENT24, or GL_DEPTH_COMPONENT32.

GL_INVALID_OPERATION is generated if *format* is GL_DEPTH_COMPONENT and *internalFormat* is not GL_DEPTH_COMPONENT, GL_DEPTH_COMPONENT16, GL_DEPTH_COMPONENT24, or GL_DEPTH_COMPONENT32.

GL_INVALID_OPERATION is generated if *internalFormat* is GL_DEPTH_COMPONENT, GL_DEPTH_COMPONENT16, GL_DEPTH_COMPONENT24, or GL_DEPTH_COMPONENT32, and *format* is not GL_DEPTH_COMPONENT.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and the buffer object's data store is currently mapped.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and *data* is not evenly divisible into the number of bytes needed to store in memory a datum indicated by *type*.

GL_INVALID_OPERATION is generated if glTexImage2D is executed between the execution of glBegin and the corresponding execution of glEnd.

`void glTexImage3D` *target level internalFormat width height depth border format type data* [Function]

Specify a three-dimensional texture image.

target Specifies the target texture. Must be GL_TEXTURE_3D or GL_PROXY_TEXTURE_3D.

level Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n^th* mipmap reduction image.

internalFormat
Specifies the number of color components in the texture. Must be 1, 2, 3, or 4, or one of the following symbolic constants: GL_ALPHA, GL_ALPHA4, GL_ALPHA8, GL_ALPHA12, GL_ALPHA16, GL_COMPRESSED_ALPHA, GL_COMPRESSED_LUMINANCE, GL_COMPRESSED_LUMINANCE_ALPHA, GL_COMPRESSED_INTENSITY, GL_COMPRESSED_RGB, GL_COMPRESSED_RGBA, GL_LUMINANCE, GL_LUMINANCE4, GL_LUMINANCE8, GL_LUMINANCE12, GL_LUMINANCE16, GL_LUMINANCE_ALPHA, GL_LUMINANCE4_ALPHA4, GL_LUMINANCE6_ALPHA2, GL_LUMINANCE8_ALPHA8, GL_LUMINANCE12_ALPHA4, GL_LUMINANCE12_ALPHA12, GL_LUMINANCE16_ALPHA16, GL_INTENSITY, GL_INTENSITY4, GL_INTENSITY8, GL_INTENSITY12, GL_INTENSITY16, GL_R3_G3_B2, GL_RGB, GL_RGB4, GL_RGB5, GL_RGB8, GL_RGB10, GL_RGB12, GL_RGB16, GL_RGBA, GL_RGBA2, GL_RGBA4, GL_RGB5_A1, GL_RGBA8, GL_RGB10_A2, GL_RGBA12, GL_RGBA16, GL_SLUMINANCE, GL_SLUMINANCE8, GL_SLUMINANCE_ALPHA, GL_SLUMINANCE8_ALPHA8, GL_SRGB, GL_SRGB8, GL_SRGB_ALPHA, or GL_SRGB8_ALPHA8.

width Specifies the width of the texture image including the border if any. If the GL version does not support non-power-of-two sizes, this value must be 2^*n*+2(*border*,) for some integer *n*. All implementations support 3D texture images that are at least 16 texels wide.

height Specifies the height of the texture image including the border if any. If the GL version does not support non-power-of-two sizes, this value must be 2^*m*+2(*border*,) for some integer *m*. All implementations support 3D texture images that are at least 16 texels high.

depth Specifies the depth of the texture image including the border if any. If the GL version does not support non-power-of-two sizes, this value must be 2^*k*+2(*border*,) for some integer *k*. All implementations support 3D texture images that are at least 16 texels deep.

border Specifies the width of the border. Must be either 0 or 1.

format Specifies the format of the pixel data. The following symbolic values are accepted: GL_COLOR_INDEX, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_BGR, GL_RGBA, GL_BGRA, 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_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, GL_FLOAT, GL_UNSIGNED_BYTE_3_3_2, GL_UNSIGNED_BYTE_2_3_3_REV, GL_UNSIGNED_SHORT_5_6_5, GL_UNSIGNED_SHORT_5_6_5_REV,

GL_UNSIGNED_SHORT_4_4_4_4, GL_UNSIGNED_SHORT_4_4_4_4_REV, GL_UNSIGNED_SHORT_5_5_5_1, GL_UNSIGNED_SHORT_1_5_5_5_REV, GL_UNSIGNED_INT_8_8_8_8, GL_UNSIGNED_INT_8_8_8_8_REV, GL_UNSIGNED_INT_10_10_10_2, and GL_UNSIGNED_INT_2_10_10_10_REV.

data Specifies a pointer to the image data in memory.

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. To enable and disable three-dimensional texturing, call glEnable and glDisable with argument GL_TEXTURE_3D.

To define texture images, call glTexImage3D. The arguments describe the parameters of the texture image, such as height, width, depth, width of the border, level-of-detail number (see glTexParameter), and number of color components provided. The last three arguments describe how the image is represented in memory; they are identical to the pixel formats used for glDrawPixels.

If *target* is GL_PROXY_TEXTURE_3D, no data is read from *data*, but all of the texture image state is recalculated, checked for consistency, and checked against the implementation's capabilities. If the implementation cannot handle a texture of the requested texture size, it sets all of the image state to 0, but does not generate an error (see glGetError). To query for an entire mipmap array, use an image array level greater than or equal to 1.

If *target* is GL_TEXTURE_3D, data is read from *data* as a sequence of signed or unsigned bytes, shorts, or longs, or single-precision floating-point values, depending on *type*. These values are grouped into sets of one, two, three, or four values, depending on *format*, to form elements. If *type* is GL_BITMAP, the data is considered as a string of unsigned bytes (and *format* must be GL_COLOR_INDEX). Each data byte is treated as eight 1-bit elements, with bit ordering determined by GL_UNPACK_LSB_FIRST (see glPixelStore).

If a non-zero named buffer object is bound to the GL_PIXEL_UNPACK_BUFFER target (see glBindBuffer) while a texture image is specified, *data* is treated as a byte offset into the buffer object's data store.

The first element corresponds to the lower left corner of the texture image. Subsequent elements progress left-to-right through the remaining texels in the lowest row of the texture image, and then in successively higher rows of the texture image. The final element corresponds to the upper right corner of the texture image.

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

GL_COLOR_INDEX
: Each element is a single value, a color index. The GL converts it to fixed point (with an unspecified number of zero bits to the right of the binary point), shifted left or right depending on the value and sign of GL_INDEX_SHIFT, and added to GL_INDEX_OFFSET (see glPixelTransfer). The resulting index is converted to a set of color components using the GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, and GL_PIXEL_MAP_I_TO_A tables, and clamped to the range [0,1].

`GL_RED` Each element is a single red component. The GL converts it to floating point and assembles it into an RGBA element by attaching 0 for green and blue, and 1 for alpha. Each component is then multiplied by the signed scale factor `GL_c_SCALE`, added to the signed bias `GL_c_BIAS`, and clamped to the range [0,1] (see `glPixelTransfer`).

`GL_GREEN` Each element is a single green component. The GL converts it to floating point and assembles it into an RGBA element by attaching 0 for red and blue, and 1 for alpha. Each component is then multiplied by the signed scale factor `GL_c_SCALE`, added to the signed bias `GL_c_BIAS`, and clamped to the range [0,1] (see `glPixelTransfer`).

`GL_BLUE` Each element is a single blue component. The GL converts it to floating point and assembles it into an RGBA element by attaching 0 for red and green, and 1 for alpha. Each component is then multiplied by the signed scale factor `GL_c_SCALE`, added to the signed bias `GL_c_BIAS`, and clamped to the range [0,1] (see `glPixelTransfer`).

`GL_ALPHA` Each element is a single alpha component. The GL converts it to floating point and assembles it into an RGBA element by attaching 0 for red, green, and blue. Each component is then multiplied by the signed scale factor `GL_c_SCALE`, added to the signed bias `GL_c_BIAS`, and clamped to the range [0,1] (see `glPixelTransfer`).

`GL_INTENSITY`
Each element is a single intensity value. The GL converts it to floating point, then assembles it into an RGBA element by replicating the intensity value three times for red, green, blue, and alpha. Each component is then multiplied by the signed scale factor `GL_c_SCALE`, added to the signed bias `GL_c_BIAS`, and clamped to the range [0,1] (see `glPixelTransfer`).

`GL_RGB`

`GL_BGR` Each element is an RGB triple. The GL converts it to floating point and assembles it into an RGBA element by attaching 1 for alpha. Each component is then multiplied by the signed scale factor `GL_c_SCALE`, added to the signed bias `GL_c_BIAS`, and clamped to the range [0,1] (see `glPixelTransfer`).

`GL_RGBA`

`GL_BGRA` Each element contains all four components. Each component is multiplied by the signed scale factor `GL_c_SCALE`, added to the signed bias `GL_c_BIAS`, and clamped to the range [0,1] (see `glPixelTransfer`).

`GL_LUMINANCE`
Each element is a single luminance value. The GL converts it to floating point, then assembles it into an RGBA element by replicating the luminance value three times for red, green, and blue and attaching 1 for alpha. Each component is then multiplied by the signed scale factor `GL_c_SCALE`, added to the signed bias `GL_c_BIAS`, and clamped to the range [0,1] (see `glPixelTransfer`).

GL_LUMINANCE_ALPHA
: Each element is a luminance/alpha pair. The GL converts it to floating point, then assembles it into an RGBA element by replicating the luminance value three times for red, green, and blue. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see glPixelTransfer).

Refer to the glDrawPixels reference page for a description of the acceptable values for the *type* parameter.

If an application wants to store the texture at a certain resolution or in a certain format, it can request the resolution and format with *internalFormat*. The GL will choose an internal representation that closely approximates that requested by *internalFormat*, but it may not match exactly. (The representations specified by GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_RGB, and GL_RGBA must match exactly. The numeric values 1, 2, 3, and 4 may also be used to specify the above representations.)

If the *internalFormat* parameter is one of the generic compressed formats, GL_COMPRESSED_ALPHA, GL_COMPRESSED_INTENSITY, GL_COMPRESSED_LUMINANCE, GL_COMPRESSED_LUMINANCE_ALPHA, GL_COMPRESSED_RGB, or GL_COMPRESSED_RGBA, the GL will replace the internal format with the symbolic constant for a specific internal format and compress the texture before storage. If no corresponding internal format is available, or the GL can not compress that image for any reason, the internal format is instead replaced with a corresponding base internal format.

If the *internalFormat* parameter is GL_SRGB, GL_SRGB8, GL_SRGB_ALPHA, GL_SRGB8_ALPHA8, GL_SLUMINANCE, GL_SLUMINANCE8, GL_SLUMINANCE_ALPHA, or GL_SLUMINANCE8_ALPHA8, the texture is treated as if the red, green, blue, or luminance components are encoded in the sRGB color space. Any alpha component is left unchanged. The conversion from the sRGB encoded component c_s to a linear component c_l is:

c_l={(c_s/12.92 if c_s0.04045), ((c_s+0.055/1.055)^2.4 if c_s>0.04045)

Assume c_s is the sRGB component in the range [0,1].

Use the GL_PROXY_TEXTURE_3D target to try out a resolution and format. The implementation will update and recompute its best match for the requested storage resolution and format. To then query this state, call glGetTexLevelParameter. If the texture cannot be accommodated, texture state is set to 0.

A one-component texture image uses only the red component of the RGBA color extracted from *data*. A two-component image uses the R and A values. A three-component image uses the R, G, and B values. A four-component image uses all of the RGBA components.

GL_INVALID_ENUM is generated if *target* is not GL_TEXTURE_3D or GL_PROXY_TEXTURE_3D.

GL_INVALID_ENUM is generated if *format* is not an accepted format constant. Format constants other than GL_STENCIL_INDEX and GL_DEPTH_COMPONENT are accepted.

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

GL_INVALID_ENUM is generated if *type* is GL_BITMAP and *format* is not GL_COLOR_INDEX.

GL_INVALID_VALUE is generated if *level* is less than 0.

GL_INVALID_VALUE may be generated if *level* is greater than *log_2(max,)*, where *max* is the returned value of GL_MAX_TEXTURE_SIZE.

GL_INVALID_VALUE is generated if *internalFormat* is not 1, 2, 3, 4, or one of the accepted resolution and format symbolic constants.

GL_INVALID_VALUE is generated if *width*, *height*, or *depth* is less than 0 or greater than 2 + GL_MAX_TEXTURE_SIZE.

GL_INVALID_VALUE is generated if non-power-of-two textures are not supported and the *width*, *height*, or *depth* cannot be represented as *2^k+2(border,)* for some integer value of *k*.

GL_INVALID_VALUE is generated if *border* is not 0 or 1.

GL_INVALID_OPERATION is generated if *type* is one of GL_UNSIGNED_BYTE_3_3_2, GL_UNSIGNED_BYTE_2_3_3_REV, GL_UNSIGNED_SHORT_5_6_5, or GL_UNSIGNED_SHORT_5_6_5_REV and *format* is not GL_RGB.

GL_INVALID_OPERATION is generated if *type* is one of GL_UNSIGNED_SHORT_4_4_4_4, GL_UNSIGNED_SHORT_4_4_4_4_REV, GL_UNSIGNED_SHORT_5_5_5_1, GL_UNSIGNED_SHORT_1_5_5_5_REV, GL_UNSIGNED_INT_8_8_8_8, GL_UNSIGNED_INT_8_8_8_8_REV, GL_UNSIGNED_INT_10_10_10_2, or GL_UNSIGNED_INT_2_10_10_10_REV and *format* is neither GL_RGBA nor GL_BGRA.

GL_INVALID_OPERATION is generated if *format* or *internalFormat* is GL_DEPTH_COMPONENT, GL_DEPTH_COMPONENT16, GL_DEPTH_COMPONENT24, or GL_DEPTH_COMPONENT32.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and the buffer object's data store is currently mapped.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and *data* is not evenly divisible into the number of bytes needed to store in memory a datum indicated by *type*.

GL_INVALID_OPERATION is generated if glTexImage3D is executed between the execution of glBegin and the corresponding execution of glEnd.

`void glTexParameterf` *target pname param* [Function]
`void glTexParameteri` *target pname param* [Function]
`void glTexParameterfv` *target pname params* [Function]
`void glTexParameteriv` *target pname params* [Function]

Set texture parameters.

target Specifies the target texture, which must be either GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D, or GL_TEXTURE_CUBE_MAP.

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_MIN_LOD, GL_TEXTURE_MAX_LOD,

GL_TEXTURE_BASE_LEVEL, GL_TEXTURE_MAX_LEVEL, GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T, GL_TEXTURE_WRAP_R, GL_TEXTURE_PRIORITY, GL_TEXTURE_COMPARE_MODE, GL_TEXTURE_COMPARE_FUNC, GL_DEPTH_TEXTURE_MODE, or GL_GENERATE_MIPMAP.

param Specifies the value of *pname*.

Texture mapping is a technique that applies an image onto an object's surface as if the image were a decal or cellophane shrink-wrap. The image is created in texture space, with an (s, t) coordinate system. A texture is a one- or two-dimensional image and a set of parameters that determine how samples are derived from the image.

glTexParameter assigns the value or values in *params* to the texture parameter specified as *pname*. *target* defines the target texture, either GL_TEXTURE_1D, GL_TEXTURE_2D, or GL_TEXTURE_3D. 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 $2^n 2^m$, there are $max(n,m)+1$ mipmaps. The first mipmap is the original texture, with dimensions $2^n 2^m$. Each subsequent mipmap has dimensions 2^k-1,2^l-1,, where $2^k 2^l$ are the dimensions of the previous mipmap, until either k=0 or l=0. At that point, subsequent mipmaps have dimension 12^l-1, or 2^k-1,1 until the final mipmap, which has dimension 11. To define the mipmaps, call glTexImage1D, glTexImage2D, glTexImage3D, glCopyTexImage1D, 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 11 mipmap.

params supplies a function for minifying the texture as one of the following:

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_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.

`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_MIN_LOD`
: Sets the minimum level-of-detail parameter. This floating-point value limits the selection of highest resolution mipmap (lowest mipmap level). The initial value is -1000.

`GL_TEXTURE_MAX_LOD`
: Sets the maximum level-of-detail parameter. This floating-point value limits the selection of the lowest resolution mipmap (highest mipmap level). The initial value is 1000.

`GL_TEXTURE_BASE_LEVEL`
: Specifies the index of the lowest defined mipmap level. This is an integer value. The initial value is 0.

`GL_TEXTURE_MAX_LEVEL`
: Sets the index of the highest defined mipmap level. This is an integer value. The initial value is 1000.

`GL_TEXTURE_WRAP_S`
: Sets the wrap parameter for texture coordinate *s* to either `GL_CLAMP`, `GL_CLAMP_TO_BORDER`, `GL_CLAMP_TO_EDGE`, `GL_MIRRORED_REPEAT`, 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_BORDER` causes the *s* coordinate to be clamped to the range [-1/2*N*,,1+1/2*N*,], where *N* is the size of the texture in the direction of clamping.`GL_CLAMP_TO_EDGE` causes *s* coordinates to be clamped to the range [1/2*N*,,1-1/2*N*,], 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. `GL_MIRRORED_REPEAT` causes the *s* coordinate to be set to the fractional part of the texture coordinate if the integer part of *s* is even; if the integer part of *s* is odd, then the *s* texture coordinate is set to 1-*frac*(*s*,), where *frac*(*s*,) represents the fractional part of *s*. Border texture elements are accessed only if wrapping is set to `GL_CLAMP` or `GL_CLAMP_TO_BORDER`. 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_BORDER`, `GL_CLAMP_TO_EDGE`, `GL_MIRRORED_REPEAT`, or `GL_REPEAT`. See the discussion under `GL_TEXTURE_WRAP_S`. Initially, `GL_TEXTURE_WRAP_T` is set to `GL_REPEAT`.

`GL_TEXTURE_WRAP_R`
: Sets the wrap parameter for texture coordinate *r* to either `GL_CLAMP`, `GL_CLAMP_TO_BORDER`, `GL_CLAMP_TO_EDGE`, `GL_MIRRORED_REPEAT`, or `GL_REPEAT`. See the discussion under `GL_TEXTURE_WRAP_S`. Initially, `GL_TEXTURE_WRAP_R` is set to `GL_REPEAT`.

`GL_TEXTURE_BORDER_COLOR`
: Sets a border color. *params* contains four values that comprise the RGBA color of the texture border. Integer color components are interpreted linearly such that the most positive integer maps to 1.0, and the most negative integer maps to -1.0. The values are clamped to the range [0,1] when they are specified. Initially, the border color is (0, 0, 0, 0).

`GL_TEXTURE_PRIORITY`
: Specifies the texture residence priority of the currently bound texture. Permissible values are in the range [0,1]. See `glPrioritizeTextures` and `glBindTexture` for more information.

GL_TEXTURE_COMPARE_MODE
: Specifies the texture comparison mode for currently bound depth textures. That is, a texture whose internal format is GL_DEPTH_COMPONENT_*; see glTexImage2D) Permissible values are:

GL_TEXTURE_COMPARE_FUNC
: Specifies the comparison operator used when GL_TEXTURE_COMPARE_MODE is set to GL_COMPARE_R_TO_TEXTURE. Permissible values are: where r is the current interpolated texture coordinate, and D_t is the depth texture value sampled from the currently bound depth texture. *result* is assigned to the either the luminance, intensity, or alpha (as specified by GL_DEPTH_TEXTURE_MODE.)

GL_DEPTH_TEXTURE_MODE
: Specifies a single symbolic constant indicating how depth values should be treated during filtering and texture application. Accepted values are GL_LUMINANCE, GL_INTENSITY, and GL_ALPHA. The initial value is GL_LUMINANCE.

GL_GENERATE_MIPMAP
: Specifies a boolean value that indicates if all levels of a mipmap array should be automatically updated when any modification to the base level mipmap is done. The initial value is GL_FALSE.

GL_COMPARE_R_TO_TEXTURE
: Specifies that the interpolated and clamped r texture coordinate should be compared to the value in the currently bound depth texture. See the discussion of GL_TEXTURE_COMPARE_FUNC for details of how the comparison is evaluated. The result of the comparison is assigned to luminance, intensity, or alpha (as specified by GL_DEPTH_TEXTURE_MODE).

GL_NONE
: Specifies that the luminance, intensity, or alpha (as specified by GL_DEPTH_TEXTURE_MODE) should be assigned the appropriate value from the currently bound depth texture.

Texture Comparison Function
: **Computed result**

GL_LEQUAL
: *result*={(1.0), (0.0)(r<=D_t,), (r>D_t,),

GL_GEQUAL
: *result*={(1.0), (0.0)(r>=D_t,), (r<D_t,),

GL_LESS
: *result*={(1.0), (0.0)(r<D_t,), (r>=D_t,),

GL_GREATER
: *result*={(1.0), (0.0)(r>D_t,), (r<=D_t,),

GL_EQUAL
: *result*={(1.0), (0.0)(r=D_t,), ($r$$D_t$,),

GL_NOTEQUAL
: *result*={(1.0), (0.0)($r$$D_t$,), ($r$=$D_t$,),

`GL_ALWAYS`
: *result*=`1.0`

`GL_NEVER` *result*=`0.0`

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

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

`GL_INVALID_OPERATION` is generated if `glTexParameter` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glTexSubImage1D` *target level xoffset width format type data* [Function]

Specify a one-dimensional texture subimage.

target
: Specifies the target texture. Must be `GL_TEXTURE_1D`.

level
: Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.

xoffset
: Specifies a texel offset in the x direction within the texture array.

width
: Specifies the width of the texture subimage.

format
: Specifies the format of the pixel data. The following symbolic values are accepted: `GL_COLOR_INDEX`, `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`, `GL_RGB`, `GL_BGR`, `GL_RGBA`, `GL_BGRA`, `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_BYTE`, `GL_BITMAP`, `GL_UNSIGNED_SHORT`, `GL_SHORT`, `GL_UNSIGNED_INT`, `GL_INT`, `GL_FLOAT`, `GL_UNSIGNED_BYTE_3_3_2`, `GL_UNSIGNED_BYTE_2_3_3_REV`, `GL_UNSIGNED_SHORT_5_6_5`, `GL_UNSIGNED_SHORT_5_6_5_REV`, `GL_UNSIGNED_SHORT_4_4_4_4`, `GL_UNSIGNED_SHORT_4_4_4_4_REV`, `GL_UNSIGNED_SHORT_5_5_5_1`, `GL_UNSIGNED_SHORT_1_5_5_5_REV`, `GL_UNSIGNED_INT_8_8_8_8`, `GL_UNSIGNED_INT_8_8_8_8_REV`, `GL_UNSIGNED_INT_10_10_10_2`, and `GL_UNSIGNED_INT_2_10_10_10_REV`.

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

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. To enable or disable one-dimensional texturing, call `glEnable` and `glDisable` with argument `GL_TEXTURE_1D`.

`glTexSubImage1D` redefines a contiguous subregion of an existing one-dimensional texture image. The texels referenced by *data* replace the portion of the existing texture array with x indices *xoffset* and *xoffset*+*width*-1, inclusive. This region may not include any texels outside the range of the texture array as it was originally specified. It is not an error to specify a subtexture with width of 0, but such a specification has no effect.

If a non-zero named buffer object is bound to the `GL_PIXEL_UNPACK_BUFFER` target (see `glBindBuffer`) while a texture image is specified, *data* is treated as a byte offset into the buffer object's data store.

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

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

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

GL_INVALID_ENUM is generated if *type* is GL_BITMAP and *format* is not GL_COLOR_INDEX.

GL_INVALID_VALUE is generated if *level* is less than 0.

GL_INVALID_VALUE may be generated if *level* is greater than log_2max, where max is the returned value of GL_MAX_TEXTURE_SIZE.

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

GL_INVALID_VALUE is generated if *width* is less than 0.

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

GL_INVALID_OPERATION is generated if *type* is one of GL_UNSIGNED_BYTE_3_3_2, GL_UNSIGNED_BYTE_2_3_3_REV, GL_UNSIGNED_SHORT_5_6_5, or GL_UNSIGNED_SHORT_5_6_5_REV and *format* is not GL_RGB.

GL_INVALID_OPERATION is generated if *type* is one of GL_UNSIGNED_SHORT_4_4_4_4, GL_UNSIGNED_SHORT_4_4_4_4_REV, GL_UNSIGNED_SHORT_5_5_5_1, GL_UNSIGNED_SHORT_1_5_5_5_REV, GL_UNSIGNED_INT_8_8_8_8, GL_UNSIGNED_INT_8_8_8_8_REV, GL_UNSIGNED_INT_10_10_10_2, or GL_UNSIGNED_INT_2_10_10_10_REV and *format* is neither GL_RGBA nor GL_BGRA.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and the buffer object's data store is currently mapped.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and *data* is not evenly divisible into the number of bytes needed to store in memory a datum indicated by *type*.

GL_INVALID_OPERATION is generated if glTexSubImage1D is executed between the execution of glBegin and the corresponding execution of glEnd.

`void glTexSubImage2D` *target level xoffset yoffset width height format type data* [Function]

Specify a two-dimensional texture subimage.

target Specifies the target texture. Must be GL_TEXTURE_2D, GL_TEXTURE_CUBE_MAP_POSITIVE_X, GL_TEXTURE_CUBE_MAP_NEGATIVE_X, GL_TEXTURE_CUBE_MAP_POSITIVE_Y, GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, GL_TEXTURE_CUBE_MAP_POSITIVE_Z, or GL_TEXTURE_CUBE_MAP_NEGATIVE_Z.

level Specifies the level-of-detail number. Level 0 is the base image level. Level n is the nth mipmap reduction image.

xoffset Specifies a texel offset in the x direction within the texture array.

yoffset Specifies a texel offset in the y direction within the texture array.

width Specifies the width of the texture subimage.

height Specifies the height of the texture subimage.

format Specifies the format of the pixel data. The following symbolic values are accepted: `GL_COLOR_INDEX`, `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`, `GL_RGB`, `GL_BGR`, `GL_RGBA`, `GL_BGRA`, `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_BYTE`, `GL_BITMAP`, `GL_UNSIGNED_SHORT`, `GL_SHORT`, `GL_UNSIGNED_INT`, `GL_INT`, `GL_FLOAT`, `GL_UNSIGNED_BYTE_3_3_2`, `GL_UNSIGNED_BYTE_2_3_3_REV`, `GL_UNSIGNED_SHORT_5_6_5`, `GL_UNSIGNED_SHORT_5_6_5_REV`, `GL_UNSIGNED_SHORT_4_4_4_4`, `GL_UNSIGNED_SHORT_4_4_4_4_REV`, `GL_UNSIGNED_SHORT_5_5_5_1`, `GL_UNSIGNED_SHORT_1_5_5_5_REV`, `GL_UNSIGNED_INT_8_8_8_8`, `GL_UNSIGNED_INT_8_8_8_8_REV`, `GL_UNSIGNED_INT_10_10_10_2`, and `GL_UNSIGNED_INT_2_10_10_10_REV`.

data Specifies a pointer to the image data in memory.

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. To enable and disable two-dimensional texturing, call `glEnable` and `glDisable` with argument `GL_TEXTURE_2D`.

`glTexSubImage2D` redefines a contiguous subregion of an existing two-dimensional texture image. The texels referenced by *data* 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.

If a non-zero named buffer object is bound to the `GL_PIXEL_UNPACK_BUFFER` target (see `glBindBuffer`) while a texture image is specified, *data* is treated as a byte offset into the buffer object's data store.

`GL_INVALID_ENUM` is generated if *target* is not `GL_TEXTURE_2D`, `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Z`, or `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`.

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

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

`GL_INVALID_ENUM` is generated if *type* is `GL_BITMAP` and *format* is not `GL_COLOR_INDEX`.

`GL_INVALID_VALUE` is generated if *level* is less than 0.

`GL_INVALID_VALUE` may be generated if *level* is greater than *log_2max*, where *max* is the returned value of `GL_MAX_TEXTURE_SIZE`.

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

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

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

`GL_INVALID_OPERATION` is generated if *type* is one of `GL_UNSIGNED_BYTE_3_3_2`, `GL_UNSIGNED_BYTE_2_3_3_REV`, `GL_UNSIGNED_SHORT_5_6_5`, or `GL_UNSIGNED_SHORT_5_6_5_REV` and *format* is not `GL_RGB`.

`GL_INVALID_OPERATION` is generated if *type* is one of `GL_UNSIGNED_SHORT_4_4_4_4`, `GL_UNSIGNED_SHORT_4_4_4_4_REV`, `GL_UNSIGNED_SHORT_5_5_5_1`, `GL_UNSIGNED_SHORT_1_5_5_5_REV`, `GL_UNSIGNED_INT_8_8_8_8`, `GL_UNSIGNED_INT_8_8_8_8_REV`, `GL_UNSIGNED_INT_10_10_10_2`, or `GL_UNSIGNED_INT_2_10_10_10_REV` and *format* is neither `GL_RGBA` nor `GL_BGRA`.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the buffer object's data store is currently mapped.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

`GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to the `GL_PIXEL_UNPACK_BUFFER` target and *data* is not evenly divisible into the number of bytes needed to store in memory a datum indicated by *type*.

`GL_INVALID_OPERATION` is generated if `glTexSubImage2D` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glTexSubImage3D` *target level xoffset yoffset zoffset width height depth format type data* [Function]

Specify a three-dimensional texture subimage.

target Specifies the target texture. Must be `GL_TEXTURE_3D`.

level Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.

xoffset Specifies a texel offset in the x direction within the texture array.

yoffset Specifies a texel offset in the y direction within the texture array.

zoffset Specifies a texel offset in the z direction within the texture array.

width Specifies the width of the texture subimage.

height Specifies the height of the texture subimage.

depth Specifies the depth of the texture subimage.

format Specifies the format of the pixel data. The following symbolic values are accepted: `GL_COLOR_INDEX`, `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`, `GL_RGB`, `GL_BGR`, `GL_RGBA`, `GL_BGRA`, `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_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, GL_FLOAT, GL_UNSIGNED_BYTE_3_3_2, GL_UNSIGNED_BYTE_2_3_3_REV, GL_UNSIGNED_SHORT_5_6_5, GL_UNSIGNED_SHORT_5_6_5_REV, GL_UNSIGNED_SHORT_4_4_4_4, GL_UNSIGNED_SHORT_4_4_4_4_REV, GL_UNSIGNED_SHORT_5_5_5_1, GL_UNSIGNED_SHORT_1_5_5_5_REV, GL_UNSIGNED_INT_8_8_8_8, GL_UNSIGNED_INT_8_8_8_8_REV, GL_UNSIGNED_INT_10_10_10_2, and GL_UNSIGNED_INT_2_10_10_10_REV.

data Specifies a pointer to the image data in memory.

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. To enable and disable three-dimensional texturing, call glEnable and glDisable with argument GL_TEXTURE_3D.

glTexSubImage3D redefines a contiguous subregion of an existing three-dimensional texture image. The texels referenced by *data* replace the portion of the existing texture array with x indices *xoffset* and *xoffset*+*width*-1, inclusive, y indices *yoffset* and *yoffset*+*height*-1, inclusive, and z indices *zoffset* and *zoffset*+*depth*-1, inclusive. This region may not include any texels outside the range of the texture array as it was originally specified. It is not an error to specify a subtexture with zero width, height, or depth but such a specification has no effect.

If a non-zero named buffer object is bound to the GL_PIXEL_UNPACK_BUFFER target (see glBindBuffer) while a texture image is specified, *data* is treated as a byte offset into the buffer object's data store.

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

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

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

GL_INVALID_ENUM is generated if *type* is GL_BITMAP and *format* is not GL_COLOR_INDEX.

GL_INVALID_VALUE is generated if *level* is less than 0.

GL_INVALID_VALUE may be generated if *level* is greater than *log_2max*, 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*,), or *zoffset*<-*b*, or (*zoffset*+*depth*,)>(*d*-*b*,), where *w* is the GL_TEXTURE_WIDTH, *h* is the GL_TEXTURE_HEIGHT, *d* is the GL_TEXTURE_DEPTH and *b* is the border width of the texture image being modified. Note that *w*, *h*, and *d* include twice the border width.

GL_INVALID_VALUE is generated if *width*, *height*, or *depth* is less than 0.

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

GL_INVALID_OPERATION is generated if *type* is one of GL_UNSIGNED_BYTE_3_3_2, GL_UNSIGNED_BYTE_2_3_3_REV, GL_UNSIGNED_SHORT_5_6_5, or GL_UNSIGNED_SHORT_5_6_5_REV and *format* is not GL_RGB.

GL_INVALID_OPERATION is generated if *type* is one of GL_UNSIGNED_SHORT_4_4_4_4, GL_UNSIGNED_SHORT_4_4_4_4_REV, GL_UNSIGNED_SHORT_5_5_5_1, GL_UNSIGNED_SHORT_1_5_5_5_REV, GL_UNSIGNED_INT_8_8_8_8, GL_UNSIGNED_INT_8_8_8_8_REV, GL_UNSIGNED_INT_10_10_10_2, or GL_UNSIGNED_INT_2_10_10_10_REV and *format* is neither GL_RGBA nor GL_BGRA.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and the buffer object's data store is currently mapped.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and the data would be unpacked from the buffer object such that the memory reads required would exceed the data store size.

GL_INVALID_OPERATION is generated if a non-zero buffer object name is bound to the GL_PIXEL_UNPACK_BUFFER target and *data* is not evenly divisible into the number of bytes needed to store in memory a datum indicated by *type*.

GL_INVALID_OPERATION is generated if glTexSubImage3D is executed between the execution of glBegin and the corresponding execution of glEnd.

`void glTranslated` *x y z* [Function]
`void glTranslatef` *x y z* [Function]

Multiply the current matrix by a translation matrix.

x
y
z Specify the *x*, *y*, and *z* coordinates of a translation vector.

glTranslate produces a translation by (*x*,*yz*). 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:

((1 0 0 *x*), (0 1 0 *y*), (0 0 1 *z*), (0 0 0 1),)

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.

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

`void glUniform1f` *location v0* [Function]
`void glUniform2f` *location v0 v1* [Function]
`void glUniform3f` *location v0 v1 v2* [Function]
`void glUniform4f` *location v0 v1 v2 v3* [Function]
`void glUniform1i` *location v0* [Function]
`void glUniform2i` *location v0 v1* [Function]
`void glUniform3i` *location v0 v1 v2* [Function]
`void glUniform4i` *location v0 v1 v2 v3* [Function]
`void glUniform1fv` *location count value* [Function]
`void glUniform2fv` *location count value* [Function]

`void glUniform3fv` *location count value* [Function]
`void glUniform4fv` *location count value* [Function]
`void glUniform1iv` *location count value* [Function]
`void glUniform2iv` *location count value* [Function]
`void glUniform3iv` *location count value* [Function]
`void glUniform4iv` *location count value* [Function]
`void glUniformMatrix2fv` *location count transpose value* [Function]
`void glUniformMatrix3fv` *location count transpose value* [Function]
`void glUniformMatrix4fv` *location count transpose value* [Function]
`void glUniformMatrix2x3fv` *location count transpose value* [Function]
`void glUniformMatrix3x2fv` *location count transpose value* [Function]
`void glUniformMatrix2x4fv` *location count transpose value* [Function]
`void glUniformMatrix4x2fv` *location count transpose value* [Function]
`void glUniformMatrix3x4fv` *location count transpose value* [Function]
`void glUniformMatrix4x3fv` *location count transpose value* [Function]

Specify the value of a uniform variable for the current program object.

location Specifies the location of the uniform variable to be modified.

v0, *v1*, *v2*, *v3*
Specifies the new values to be used for the specified uniform variable.

`glUniform` modifies the value of a uniform variable or a uniform variable array. The location of the uniform variable to be modified is specified by *location*, which should be a value returned by `glGetUniformLocation`. `glUniform` operates on the program object that was made part of current state by calling `glUseProgram`.

The commands `glUniform{1|2|3|4}{f|i}` are used to change the value of the uniform variable specified by *location* using the values passed as arguments. The number specified in the command should match the number of components in the data type of the specified uniform variable (e.g., `1` for float, int, bool; `2` for vec2, ivec2, bvec2, etc.). The suffix `f` indicates that floating-point values are being passed; the suffix `i` indicates that integer values are being passed, and this type should also match the data type of the specified uniform variable. The `i` variants of this function should be used to provide values for uniform variables defined as int, ivec2, ivec3, ivec4, or arrays of these. The `f` variants should be used to provide values for uniform variables of type float, vec2, vec3, vec4, or arrays of these. Either the `i` or the `f` variants may be used to provide values for uniform variables of type bool, bvec2, bvec3, bvec4, or arrays of these. The uniform variable will be set to false if the input value is 0 or 0.0f, and it will be set to true otherwise.

All active uniform variables defined in a program object are initialized to 0 when the program object is linked successfully. They retain the values assigned to them by a call to `glUniform` until the next successful link operation occurs on the program object, when they are once again initialized to 0.

The commands `glUniform{1|2|3|4}{f|i}v` can be used to modify a single uniform variable or a uniform variable array. These commands pass a count and a pointer to the values to be loaded into a uniform variable or a uniform variable array. A count of 1 should be used if modifying the value of a single uniform variable, and a count of 1 or greater can be used to modify an entire array or part of an array. When loading

n elements starting at an arbitrary position *m* in a uniform variable array, elements *m* + *n* - 1 in the array will be replaced with the new values. If *m* + *n* - 1 is larger than the size of the uniform variable array, values for all array elements beyond the end of the array will be ignored. The number specified in the name of the command indicates the number of components for each element in *value*, and it should match the number of components in the data type of the specified uniform variable (e.g., `1` for float, int, bool; `2` for vec2, ivec2, bvec2, etc.). The data type specified in the name of the command must match the data type for the specified uniform variable as described previously for `glUniform{1|2|3|4}{f|i}`.

For uniform variable arrays, each element of the array is considered to be of the type indicated in the name of the command (e.g., `glUniform3f` or `glUniform3fv` can be used to load a uniform variable array of type vec3). The number of elements of the uniform variable array to be modified is specified by *count*

The commands `glUniformMatrix{2|3|4|2x3|3x2|2x4|4x2|3x4|4x3}fv` are used to modify a matrix or an array of matrices. The numbers in the command name are interpreted as the dimensionality of the matrix. The number `2` indicates a 2 2 matrix (i.e., 4 values), the number `3` indicates a 3 3 matrix (i.e., 9 values), and the number `4` indicates a 4 4 matrix (i.e., 16 values). Non-square matrix dimensionality is explicit, with the first number representing the number of columns and the second number representing the number of rows. For example, `2x4` indicates a 2 4 matrix with 2 columns and 4 rows (i.e., 8 values). If *transpose* is `GL_FALSE`, each matrix is assumed to be supplied in column major order. If *transpose* is `GL_TRUE`, each matrix is assumed to be supplied in row major order. The *count* argument indicates the number of matrices to be passed. A count of 1 should be used if modifying the value of a single matrix, and a count greater than 1 can be used to modify an array of matrices.

`GL_INVALID_OPERATION` is generated if there is no current program object.

`GL_INVALID_OPERATION` is generated if the size of the uniform variable declared in the shader does not match the size indicated by the `glUniform` command.

`GL_INVALID_OPERATION` is generated if one of the integer variants of this function is used to load a uniform variable of type float, vec2, vec3, vec4, or an array of these, or if one of the floating-point variants of this function is used to load a uniform variable of type int, ivec2, ivec3, or ivec4, or an array of these.

`GL_INVALID_OPERATION` is generated if *location* is an invalid uniform location for the current program object and *location* is not equal to -1.

`GL_INVALID_VALUE` is generated if *count* is less than 0.

`GL_INVALID_OPERATION` is generated if *count* is greater than 1 and the indicated uniform variable is not an array variable.

`GL_INVALID_OPERATION` is generated if a sampler is loaded using a command other than `glUniform1i` and `glUniform1iv`.

`GL_INVALID_OPERATION` is generated if `glUniform` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glUseProgram` *program* [Function]

Installs a program object as part of current rendering state.

program Specifies the handle of the program object whose executables are to be used as part of current rendering state.

`glUseProgram` installs the program object specified by *program* as part of current rendering state. One or more executables are created in a program object by successfully attaching shader objects to it with `glAttachShader`, successfully compiling the shader objects with `glCompileShader`, and successfully linking the program object with `glLinkProgram`.

A program object will contain an executable that will run on the vertex processor if it contains one or more shader objects of type `GL_VERTEX_SHADER` that have been successfully compiled and linked. Similarly, a program object will contain an executable that will run on the fragment processor if it contains one or more shader objects of type `GL_FRAGMENT_SHADER` that have been successfully compiled and linked.

Successfully installing an executable on a programmable processor will cause the corresponding fixed functionality of OpenGL to be disabled. Specifically, if an executable is installed on the vertex processor, the OpenGL fixed functionality will be disabled as follows.

- The modelview matrix is not applied to vertex coordinates.
- The projection matrix is not applied to vertex coordinates.
- The texture matrices are not applied to texture coordinates.
- Normals are not transformed to eye coordinates.
- Normals are not rescaled or normalized.
- Normalization of `GL_AUTO_NORMAL` evaluated normals is not performed.
- Texture coordinates are not generated automatically.
- Per-vertex lighting is not performed.
- Color material computations are not performed.
- Color index lighting is not performed.
- This list also applies when setting the current raster position.

The executable that is installed on the vertex processor is expected to implement any or all of the desired functionality from the preceding list. Similarly, if an executable is installed on the fragment processor, the OpenGL fixed functionality will be disabled as follows.

- Texture environment and texture functions are not applied.
- Texture application is not applied.
- Color sum is not applied.
- Fog is not applied.

Again, the fragment shader that is installed is expected to implement any or all of the desired functionality from the preceding list.

While a program object is in use, applications are free to modify attached shader objects, compile attached shader objects, attach additional shader objects, and detach or delete shader objects. None of these operations will affect the executables that are part of the current state. However, relinking the program object that is currently in

use will install the program object as part of the current rendering state if the link operation was successful (see `glLinkProgram`). If the program object currently in use is relinked unsuccessfully, its link status will be set to `GL_FALSE`, but the executables and associated state will remain part of the current state until a subsequent call to `glUseProgram` removes it from use. After it is removed from use, it cannot be made part of current state until it has been successfully relinked.

If *program* contains shader objects of type `GL_VERTEX_SHADER` but it does not contain shader objects of type `GL_FRAGMENT_SHADER`, an executable will be installed on the vertex processor, but fixed functionality will be used for fragment processing. Similarly, if *program* contains shader objects of type `GL_FRAGMENT_SHADER` but it does not contain shader objects of type `GL_VERTEX_SHADER`, an executable will be installed on the fragment processor, but fixed functionality will be used for vertex processing. If *program* is 0, the programmable processors will be disabled, and fixed functionality will be used for both vertex and fragment processing.

`GL_INVALID_VALUE` is generated if *program* is neither 0 nor a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *program* is not a program object.

`GL_INVALID_OPERATION` is generated if *program* could not be made part of current state.

`GL_INVALID_OPERATION` is generated if `glUseProgram` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glValidateProgram` *program* [Function]

Validates a program object.

program Specifies the handle of the program object to be validated.

`glValidateProgram` checks to see whether the executables contained in *program* can execute given the current OpenGL state. The information generated by the validation process will be stored in *program*'s information log. The validation information may consist of an empty string, or it may be a string containing information about how the current program object interacts with the rest of current OpenGL state. This provides a way for OpenGL implementers to convey more information about why the current program is inefficient, suboptimal, failing to execute, and so on.

The status of the validation operation will be stored as part of the program object's state. This value will be set to `GL_TRUE` if the validation succeeded, and `GL_FALSE` otherwise. It can be queried by calling `glGetProgram` with arguments *program* and `GL_VALIDATE_STATUS`. If validation is successful, *program* is guaranteed to execute given the current state. Otherwise, *program* is guaranteed to not execute.

This function is typically useful only during application development. The informational string stored in the information log is completely implementation dependent; therefore, an application should not expect different OpenGL implementations to produce identical information strings.

`GL_INVALID_VALUE` is generated if *program* is not a value generated by OpenGL.

`GL_INVALID_OPERATION` is generated if *program* is not a program object.

`GL_INVALID_OPERATION` is generated if `glValidateProgram` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glVertexAttribPointer` *index size type normalized stride pointer* [Function]

Define an array of generic vertex attribute data.

index Specifies the index of the generic vertex attribute to be modified.

size Specifies the number of components per generic vertex attribute. Must be 1, 2, 3, or 4. The initial value is 4.

type Specifies the data type of each component in the array. Symbolic constants `GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_UNSIGNED_SHORT`, `GL_INT`, `GL_UNSIGNED_INT`, `GL_FLOAT`, or `GL_DOUBLE` are accepted. The initial value is `GL_FLOAT`.

normalized
Specifies whether fixed-point data values should be normalized (`GL_TRUE`) or converted directly as fixed-point values (`GL_FALSE`) when they are accessed.

stride Specifies the byte offset between consecutive generic vertex attributes. If *stride* is 0, the generic vertex attributes 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 generic vertex attribute in the array. The initial value is 0.

`glVertexAttribPointer` specifies the location and data format of the array of generic vertex attributes at index *index* to use when rendering. *size* specifies the number of components per attribute and must be 1, 2, 3, or 4. *type* specifies the data type of each component, and *stride* specifies the byte stride from one attribute to the next, allowing vertices and attributes to be packed into a single array or stored in separate arrays. If set to `GL_TRUE`, *normalized* indicates that values stored in an integer format are to be mapped to the range [-1,1] (for signed values) or [0,1] (for unsigned values) when they are accessed and converted to floating point. Otherwise, values will be converted to floats directly without normalization.

If a non-zero named buffer object is bound to the `GL_ARRAY_BUFFER` target (see `glBindBuffer`) while a generic vertex attribute array is specified, *pointer* is treated as a byte offset into the buffer object's data store. Also, the buffer object binding (`GL_ARRAY_BUFFER_BINDING`) is saved as generic vertex attribute array client-side state (`GL_VERTEX_ATTRIB_ARRAY_BUFFER_BINDING`) for index *index*.

When a generic vertex attribute array is specified, *size*, *type*, *normalized*, *stride*, and *pointer* are saved as client-side state, in addition to the current vertex array buffer object binding.

To enable and disable a generic vertex attribute array, call `glEnableVertexAttribArray` and `glDisableVertexAttribArray` with *index*. If enabled, the generic vertex attribute array is used when `glArrayElement`, `glDrawArrays`, `glMultiDrawArrays`, `glDrawElements`, `glMultiDrawElements`, or `glDrawRangeElements` is called.

`GL_INVALID_VALUE` is generated if *index* is greater than or equal to `GL_MAX_VERTEX_ATTRIBS`.

`GL_INVALID_VALUE` is generated if *size* is not 1, 2, 3, or 4.

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

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

`void glVertexAttrib1f` *index v0* [Function]
`void glVertexAttrib1s` *index v0* [Function]
`void glVertexAttrib1d` *index v0* [Function]
`void glVertexAttrib2f` *index v0 v1* [Function]
`void glVertexAttrib2s` *index v0 v1* [Function]
`void glVertexAttrib2d` *index v0 v1* [Function]
`void glVertexAttrib3f` *index v0 v1 v2* [Function]
`void glVertexAttrib3s` *index v0 v1 v2* [Function]
`void glVertexAttrib3d` *index v0 v1 v2* [Function]
`void glVertexAttrib4f` *index v0 v1 v2 v3* [Function]
`void glVertexAttrib4s` *index v0 v1 v2 v3* [Function]
`void glVertexAttrib4d` *index v0 v1 v2 v3* [Function]
`void glVertexAttrib4Nub` *index v0 v1 v2 v3* [Function]
`void glVertexAttrib1fv` *index v* [Function]
`void glVertexAttrib1sv` *index v* [Function]
`void glVertexAttrib1dv` *index v* [Function]
`void glVertexAttrib2fv` *index v* [Function]
`void glVertexAttrib2sv` *index v* [Function]
`void glVertexAttrib2dv` *index v* [Function]
`void glVertexAttrib3fv` *index v* [Function]
`void glVertexAttrib3sv` *index v* [Function]
`void glVertexAttrib3dv` *index v* [Function]
`void glVertexAttrib4fv` *index v* [Function]
`void glVertexAttrib4sv` *index v* [Function]
`void glVertexAttrib4dv` *index v* [Function]
`void glVertexAttrib4iv` *index v* [Function]
`void glVertexAttrib4bv` *index v* [Function]
`void glVertexAttrib4ubv` *index v* [Function]
`void glVertexAttrib4usv` *index v* [Function]
`void glVertexAttrib4uiv` *index v* [Function]
`void glVertexAttrib4Nbv` *index v* [Function]
`void glVertexAttrib4Nsv` *index v* [Function]
`void glVertexAttrib4Niv` *index v* [Function]
`void glVertexAttrib4Nubv` *index v* [Function]
`void glVertexAttrib4Nusv` *index v* [Function]
`void glVertexAttrib4Nuiv` *index v* [Function]

Specifies the value of a generic vertex attribute.

index Specifies the index of the generic vertex attribute to be modified.

v0, *v1*, *v2*, *v3*
Specifies the new values to be used for the specified vertex attribute.

OpenGL defines a number of standard vertex attributes that applications can modify with standard API entry points (color, normal, texture coordinates, etc.). The

`glVertexAttrib` family of entry points allows an application to pass generic vertex attributes in numbered locations.

Generic attributes are defined as four-component values that are organized into an array. The first entry of this array is numbered 0, and the size of the array is specified by the implementation-dependent constant `GL_MAX_VERTEX_ATTRIBS`. Individual elements of this array can be modified with a `glVertexAttrib` call that specifies the index of the element to be modified and a value for that element.

These commands can be used to specify one, two, three, or all four components of the generic vertex attribute specified by *index*. A `1` in the name of the command indicates that only one value is passed, and it will be used to modify the first component of the generic vertex attribute. The second and third components will be set to 0, and the fourth component will be set to 1. Similarly, a `2` in the name of the command indicates that values are provided for the first two components, the third component will be set to 0, and the fourth component will be set to 1. A `3` in the name of the command indicates that values are provided for the first three components and the fourth component will be set to 1, whereas a `4` in the name indicates that values are provided for all four components.

The letters `s`, `f`, `i`, `d`, `ub`, `us`, and `ui` indicate whether the arguments are of type short, float, int, double, unsigned byte, unsigned short, or unsigned int. When `v` is appended to the name, the commands can take a pointer to an array of such values. The commands containing `N` indicate that the arguments will be passed as fixed-point values that are scaled to a normalized range according to the component conversion rules defined by the OpenGL specification. Signed values are understood to represent fixed-point values in the range [-1,1], and unsigned values are understood to represent fixed-point values in the range [0,1].

OpenGL Shading Language attribute variables are allowed to be of type mat2, mat3, or mat4. Attributes of these types may be loaded using the `glVertexAttrib` entry points. Matrices must be loaded into successive generic attribute slots in column major order, with one column of the matrix in each generic attribute slot.

A user-defined attribute variable declared in a vertex shader can be bound to a generic attribute index by calling `glBindAttribLocation`. This allows an application to use more descriptive variable names in a vertex shader. A subsequent change to the specified generic vertex attribute will be immediately reflected as a change to the corresponding attribute variable in the vertex shader.

The binding between a generic vertex attribute index and a user-defined attribute variable in a vertex shader is part of the state of a program object, but the current value of the generic vertex attribute is not. The value of each generic vertex attribute is part of current state, just like standard vertex attributes, and it is maintained even if a different program object is used.

An application may freely modify generic vertex attributes that are not bound to a named vertex shader attribute variable. These values are simply maintained as part of current state and will not be accessed by the vertex shader. If a generic vertex attribute bound to an attribute variable in a vertex shader is not updated while the vertex shader is executing, the vertex shader will repeatedly use the current value for the generic vertex attribute.

The generic vertex attribute with index 0 is the same as the vertex position attribute previously defined by OpenGL. A `glVertex2`, `glVertex3`, or `glVertex4` command is completely equivalent to the corresponding `glVertexAttrib` command with an index argument of 0. A vertex shader can access generic vertex attribute 0 by using the built-in attribute variable *gl_Vertex*. There are no current values for generic vertex attribute 0. This is the only generic vertex attribute with this property; calls to set other standard vertex attributes can be freely mixed with calls to set any of the other generic vertex attributes.

`GL_INVALID_VALUE` is generated if *index* is greater than or equal to `GL_MAX_VERTEX_ATTRIBS`.

`void glVertexPointer` *size type stride pointer* [Function]

Define an array of vertex data.

size Specifies the number of coordinates per vertex. Must be 2, 3, or 4. The initial value is 4.

type Specifies the data type of each coordinate in the array. Symbolic constants `GL_SHORT`, `GL_INT`, `GL_FLOAT`, or `GL_DOUBLE` are accepted. The initial value is `GL_FLOAT`.

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

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

`glVertexPointer` specifies the location and data format of an array of vertex coordinates to use when rendering. *size* specifies the number of coordinates per vertex, and must be 2, 3, or 4. *type* specifies the data type of each coordinate, and *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; see `glInterleavedArrays`.)

If a non-zero named buffer object is bound to the `GL_ARRAY_BUFFER` target (see `glBindBuffer`) while a vertex array is specified, *pointer* is treated as a byte offset into the buffer object's data store. Also, the buffer object binding (`GL_ARRAY_BUFFER_BINDING`) is saved as vertex array client-side state (`GL_VERTEX_ARRAY_BUFFER_BINDING`).

When a vertex array is specified, *size*, *type*, *stride*, and *pointer* are saved as client-side state, in addition to the current vertex array buffer object binding.

To enable and disable the vertex array, call `glEnableClientState` and `glDisableClientState` with the argument `GL_VERTEX_ARRAY`. If enabled, the vertex array is used when `glArrayElement`, `glDrawArrays`, `glMultiDrawArrays`, `glDrawElements`, `glMultiDrawElements`, or `glDrawRangeElements` is called.

`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.

`void glVertex2s` *x y* [Function]
`void glVertex2i` *x y* [Function]
`void glVertex2f` *x y* [Function]
`void glVertex2d` *x y* [Function]
`void glVertex3s` *x y z* [Function]
`void glVertex3i` *x y z* [Function]
`void glVertex3f` *x y z* [Function]
`void glVertex3d` *x y z* [Function]
`void glVertex4s` *x y z w* [Function]
`void glVertex4i` *x y z w* [Function]
`void glVertex4f` *x y z w* [Function]
`void glVertex4d` *x y z w* [Function]
`void glVertex2sv` *v* [Function]
`void glVertex2iv` *v* [Function]
`void glVertex2fv` *v* [Function]
`void glVertex2dv` *v* [Function]
`void glVertex3sv` *v* [Function]
`void glVertex3iv` *v* [Function]
`void glVertex3fv` *v* [Function]
`void glVertex3dv` *v* [Function]
`void glVertex4sv` *v* [Function]
`void glVertex4iv` *v* [Function]
`void glVertex4fv` *v* [Function]
`void glVertex4dv` *v* [Function]

Specify a vertex.

x
y
z
w Specify *x*, *y*, *z*, and *w* coordinates of a vertex. Not all parameters are present in all forms of the command.

`glVertex` commands are used within `glBegin`/`glEnd` pairs to specify point, line, and polygon vertices. The current color, normal, texture coordinates, and fog coordinate are associated with the vertex when `glVertex` is called.

When only *x* and *y* are specified, *z* defaults to 0 and *w* defaults to 1. When *x*, *y*, and *z* are specified, *w* defaults to 1.

`void glViewport` *x y width height* [Function]

Set the viewport.

x
y Specify the lower left corner of the viewport rectangle, in pixels. The initial value is (0,0).

width
height Specify the width and height of the viewport. When a GL context is first attached to a window, *width* and *height* are set to the dimensions of that window.

`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:

$x\_w=(x\_nd+1,)(width/2,)+x$

$y\_w=(y\_nd+1,)(height/2,)+y$

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

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

`GL_INVALID_OPERATION` is generated if `glViewport` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`void glWindowPos2s` *x y* [Function]
`void glWindowPos2i` *x y* [Function]
`void glWindowPos2f` *x y* [Function]
`void glWindowPos2d` *x y* [Function]
`void glWindowPos3s` *x y z* [Function]
`void glWindowPos3i` *x y z* [Function]
`void glWindowPos3f` *x y z* [Function]
`void glWindowPos3d` *x y z* [Function]
`void glWindowPos2sv` *v* [Function]
`void glWindowPos2iv` *v* [Function]
`void glWindowPos2fv` *v* [Function]
`void glWindowPos2dv` *v* [Function]
`void glWindowPos3sv` *v* [Function]
`void glWindowPos3iv` *v* [Function]
`void glWindowPos3fv` *v* [Function]
`void glWindowPos3dv` *v* [Function]

Specify the raster position in window coordinates for pixel operations.

x
y
z Specify the x, y, z coordinates for the raster position.

The GL maintains a 3D position in window coordinates. This position, called the raster position, is used to position pixel and bitmap write operations. It is maintained with subpixel accuracy. See `glBitmap`, `glDrawPixels`, and `glCopyPixels`.

`glWindowPos2` specifies the x and y coordinates, while z is implicitly set to 0. `glWindowPos3` specifies all three coordinates. The w coordinate of the current raster position is always set to 1.0.

`glWindowPos` directly updates the x and y coordinates of the current raster position with the values specified. That is, the values are neither transformed by the current modelview and projection matrices, nor by the viewport-to-window transform. The z coordinate of the current raster position is updated in the following manner:

$z=\{(n), (f), (n+z(f-n,),)(if\,z<=0), (if\,z>=1), (\texttt{otherwise},),$

where n is `GL_DEPTH_RANGE`'s near value, and f is `GL_DEPTH_RANGE`'s far value. See `glDepthRange`.

The specified coordinates are not clip-tested, causing the raster position to always be valid.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then `GL_CURRENT_RASTER_COLOR` (in RGBA mode) or `GL_CURRENT_RASTER_INDEX` (in color index mode) is set to the color produced by the lighting calculation (see `glLight`, `glLightModel`, and `glShadeModel`). If lighting is disabled, current color (in RGBA mode, state variable `GL_CURRENT_COLOR`) or color index (in color index mode, state variable `GL_CURRENT_INDEX`) is used to update the current raster color. `GL_CURRENT_RASTER_SECONDARY_COLOR` (in RGBA mode) is likewise updated.

Likewise, `GL_CURRENT_RASTER_TEXTURE_COORDS` is updated as a function of `GL_CURRENT_TEXTURE_COORDS`, based on the texture matrix and the texture generation functions (see `glTexGen`). The `GL_CURRENT_RASTER_DISTANCE` is set to the `GL_CURRENT_FOG_COORD`.

`GL_INVALID_OPERATION` is generated if `glWindowPos` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

3.7 GL Extensions

The future is already here – it's just not very evenly distributed.

– William Gibson

Before interfaces end up in the core OpenGL API, the are usually present as vendor-specific or candidate extensions. Indeed, the making of an OpenGL standard these days seems to be a matter of simply collecting a set of mature extensions and making them coherent.

Guile doesn't currently provide specific interfaces for GL extensions. Perhaps it should, but that's a lot of work that we haven't had time to do. Contributions are welcome.

In the meantime, if you know enough about GL to know that you need an extension, you can define one yourself – after all, this library is all a bunch of Scheme code anyway.

For example, let's say you decide that you need to render to a framebuffer object. You go to http://www.opengl.org/registry/ and pick out an extension, say http://www.opengl.org/registry/specs/ARB/framebuffer_object.txt.

This extension defines a procedure, `GLboolean glIsRenderBuffer(GLuint)`. So you define it:

```
(use-modules (gl runtime) (gl types))
(define-gl-procedure (glIsRenderBuffer (buf GLuint) -> GLboolean)
  "Render buffer predicate.  Other docs here.")
```

And that's that. It's a low-level binding, but what did you expect?

Note that you'll still need to check for the availability of this extension at runtime with `(glGetString GL_EXTENSIONS)`.

4 GLU

4.1 GLU API

Import the GLU module to have access to these procedures:

```
(use-modules (glu))
```

The GLU specification is available at http://www.opengl.org/registry/doc/glu1.3.pdf.

4.1.1 Initialization

4.1.2 Mipmapping

4.1.3 Matrix Manipulation

`glu-perspective` *fov-y aspect z-near z-far* [Function]

Set up a perspective projection matrix.

fov-y is the field of view angle, in degrees, in the Y direction. *aspect* is the ratio of width to height. *z-near* and *z-far* are the distances from the viewer to the near and far clipping planes, respectively.

The resulting matrix is multiplied against the current matrix.

4.1.4 Polygon Tesselation

4.1.5 Quadrics

4.1.6 NURBS

4.1.7 Errors

4.2 Low-Level GLU

The functions from this section may be had by loading the module:

```
(use-modules (glu low-level)
```

This section of the manual was derived from the upstream OpenGL documentation. Each function's documentation has its own copyright statement; for full details, see the upstream documentation. The copyright notices and licenses present in this section are as follows.

`void gluBeginCurve` *nurb* [Function]
`void gluEndCurve` *nurb* [Function]

Delimit a NURBS curve definition.

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

Use `gluBeginCurve` to mark the beginning of a NURBS curve definition. After calling `gluBeginCurve`, make one or more calls to `gluNurbsCurve` to define the attributes of the curve. Exactly one of the calls to `gluNurbsCurve` must have a curve type of `GLU_MAP1_VERTEX_3` or `GLU_MAP1_VERTEX_4`. To mark the end of the NURBS curve definition, call `gluEndCurve`.

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

`void gluBeginPolygon` *tess* [Function]
`void gluEndPolygon` *tess* [Function]

Delimit a polygon description.

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

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

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

`void gluBeginSurface` *nurb* [Function]
`void gluEndSurface` *nurb* [Function]

Delimit a NURBS surface definition.

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

Use `gluBeginSurface` to mark the beginning of a NURBS surface definition. After calling `gluBeginSurface`, make one or more calls to `gluNurbsSurface` to define the attributes of the surface. Exactly one of these calls to `gluNurbsSurface` must have a surface type of `GLU_MAP2_VERTEX_3` or `GLU_MAP2_VERTEX_4`. To mark the end of the NURBS surface definition, call `gluEndSurface`.

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

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

`void gluBeginTrim` *nurb* [Function]
`void gluEndTrim` *nurb* [Function]

Delimit a NURBS trimming loop definition.

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

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

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

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

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

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

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

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

`GLint gluBuild1DMipmapLevels` *target internalFormat width format type level base max data* [Function]

Builds a subset of one-dimensional mipmap levels.

target Specifies the target texture. Must be `GLU_TEXTURE_1D`.

internalFormat

Requests the internal storage format of the texture image. The most current version of the SGI implementation of GLU does not check this value for validity before passing it on to the underlying OpenGL implementation. A value that is not accepted by the OpenGL implementation will lead to an OpenGL error. The benefit of not checking this value at the GLU level is that OpenGL extensions can add new internal texture

formats without requiring a revision of the GLU implementation. Older implementations of GLU check this value and raise a GLU error if it is not 1, 2, 3, or 4 or one of the following symbolic constants: GLU_ALPHA, GLU_ALPHA4, GLU_ALPHA8, GLU_ALPHA12, GLU_ALPHA16, GLU_LUMINANCE, GLU_LUMINANCE4, GLU_LUMINANCE8, GLU_LUMINANCE12, GLU_LUMINANCE16, GLU_LUMINANCE_ALPHA, GLU_LUMINANCE4_ALPHA4, GLU_LUMINANCE6_ALPHA2, GLU_LUMINANCE8_ALPHA8, GLU_LUMINANCE12_ALPHA4, GLU_LUMINANCE12_ALPHA12, GLU_LUMINANCE16_ALPHA16, GLU_INTENSITY, GLU_INTENSITY4, GLU_INTENSITY8, GLU_INTENSITY12, GLU_INTENSITY16, GLU_RGB, GLU_R3_G3_B2, GLU_RGB4, GLU_RGB5, GLU_RGB8, GLU_RGB10, GLU_RGB12, GLU_RGB16, GLU_RGBA, GLU_RGBA2, GLU_RGBA4, GLU_RGB5_A1, GLU_RGBA8, GLU_RGB10_A2, GLU_RGBA12, or GLU_RGBA16.

width Specifies the width in pixels of the texture image. This should be a power of 2.

format Specifies the format of the pixel data. Must be one of: GLU_COLOR_INDEX, GLU_DEPTH_COMPONENT, GLU_RED, GLU_GREEN, GLU_BLUE, GLU_ALPHA, GLU_RGB, GLU_RGBA, GLU_BGR, GLU_BGRA, GLU_LUMINANCE, or GLU_LUMINANCE_ALPHA.

type Specifies the data type for *data*. Must be one of: GLU_UNSIGNED_BYTE, GLU_BYTE, GLU_BITMAP, GLU_UNSIGNED_SHORT, GLU_SHORT, GLU_UNSIGNED_INT, GLU_INT, GLU_FLOAT, GLU_UNSIGNED_BYTE_3_3_2, GLU_UNSIGNED_BYTE_2_3_3_REV, GLU_UNSIGNED_SHORT_5_6_5, GLU_UNSIGNED_SHORT_5_6_5_REV, GLU_UNSIGNED_SHORT_4_4_4_4, GLU_UNSIGNED_SHORT_4_4_4_4_REV, GLU_UNSIGNED_SHORT_5_5_5_1, GLU_UNSIGNED_SHORT_1_5_5_5_REV, GLU_UNSIGNED_INT_8_8_8_8, GLU_UNSIGNED_INT_8_8_8_8_REV, GLU_UNSIGNED_INT_10_10_10_2, or GLU_UNSIGNED_INT_2_10_10_10_REV.

level Specifies the mipmap level of the image data.

base Specifies the minimum mipmap level to pass to glTexImage1D.

max Specifies the maximum mipmap level to pass to glTexImage1D.

data Specifies a pointer to the image data in memory.

gluBuild1DMipmapLevels builds a subset of prefiltered one-dimensional texture maps of decreasing resolutions called a mipmap. This is used for the antialiasing of texture mapped primitives.

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

A series of mipmap levels from *base* to *max* is built by decimating *data* in half until size 11 is reached. At each level, each texel in the halved mipmap level is an average of the corresponding two texels in the larger mipmap level. glTexImage1D is called to load these mipmap levels from *base* to *max*. If *max* is larger than the highest mipmap level for the texture of the specified size, then a GLU error code is returned (see gluErrorString) and nothing is loaded.

For example, if *level* is 2 and *width* is 16, the following levels are possible: 161, 81, 41, 21, 11. These correspond to levels 2 through 6 respectively. If *base* is 3 and *max* is 5, then only mipmap levels 81, 41 and 21 are loaded. However, if *max* is 7, then an error is returned and nothing is loaded since *max* is larger than the highest mipmap level which is, in this case, 6.

The highest mipmap level can be derived from the formula *log_2(width2^level,)*.

See the `glTexImage1D` reference page for a description of the acceptable values for *type* parameter. See the `glDrawPixels` reference page for a description of the acceptable values for *level* parameter.

`GLU_INVALID_VALUE` is returned if *level* > *base*, *base* < 0, *max* < *base* or *max* is > the highest mipmap level for *data*.

`GLU_INVALID_VALUE` is returned if *width* is < 1.

`GLU_INVALID_ENUM` is returned if *internalFormat*, *format*, or *type* are not legal.

`GLU_INVALID_OPERATION` is returned if *type* is `GLU_UNSIGNED_BYTE_3_3_2` or `GLU_UNSIGNED_BYTE_2_3_3_REV` and *format* is not `GLU_RGB`.

`GLU_INVALID_OPERATION` is returned if *type* is `GLU_UNSIGNED_SHORT_5_6_5` or `GLU_UNSIGNED_SHORT_5_6_5_REV` and *format* is not `GLU_RGB`.

`GLU_INVALID_OPERATION` is returned if *type* is `GLU_UNSIGNED_SHORT_4_4_4_4` or `GLU_UNSIGNED_SHORT_4_4_4_4_REV` and *format* is neither `GLU_RGBA` nor `GLU_BGRA`.

`GLU_INVALID_OPERATION` is returned if *type* is `GLU_UNSIGNED_SHORT_5_5_5_1` or `GLU_UNSIGNED_SHORT_1_5_5_5_REV` and *format* is neither `GLU_RGBA` nor `GLU_BGRA`.

`GLU_INVALID_OPERATION` is returned if *type* is `GLU_UNSIGNED_INT_8_8_8_8` or `GLU_UNSIGNED_INT_8_8_8_8_REV` and *format* is neither `GLU_RGBA` nor `GLU_BGRA`.

`GLU_INVALID_OPERATION` is returned if *type* is `GLU_UNSIGNED_INT_10_10_10_2` or `GLU_UNSIGNED_INT_2_10_10_10_REV` and *format* is neither `GLU_RGBA` nor `GLU_BGRA`.

`GLint gluBuild1DMipmaps` *target internalFormat width format type data* [Function]

Builds a one-dimensional mipmap.

target Specifies the target texture. Must be `GLU_TEXTURE_1D`.

internalFormat

Requests the internal storage format of the texture image. The most current version of the SGI implementation of GLU does not check this value for validity before passing it on to the underlying OpenGL implementation. A value that is not accepted by the OpenGL implementation will lead to an OpenGL error. The benefit of not checking this value at the GLU level is that OpenGL extensions can add new internal texture formats without requiring a revision of the GLU implementation. Older implementations of GLU check this value and raise a GLU error if it is not 1, 2, 3, or 4 or one of the following symbolic constants: `GLU_ALPHA`, `GLU_ALPHA4`, `GLU_ALPHA8`, `GLU_ALPHA12`, `GLU_ALPHA16`, `GLU_LUMINANCE`, `GLU_LUMINANCE4`, `GLU_LUMINANCE8`, `GLU_LUMINANCE12`, `GLU_LUMINANCE16`, `GLU_LUMINANCE_ALPHA`, `GLU_LUMINANCE4_ALPHA4`,

GLU_LUMINANCE6_ALPHA2, GLU_LUMINANCE8_ALPHA8, GLU_LUMINANCE12_ALPHA4, GLU_LUMINANCE12_ALPHA12, GLU_LUMINANCE16_ALPHA16, GLU_INTENSITY, GLU_INTENSITY4, GLU_INTENSITY8, GLU_INTENSITY12, GLU_INTENSITY16, GLU_RGB, GLU_R3_G3_B2, GLU_RGB4, GLU_RGB5, GLU_RGB8, GLU_RGB10, GLU_RGB12, GLU_RGB16, GLU_RGBA, GLU_RGBA2, GLU_RGBA4, GLU_RGB5_A1, GLU_RGBA8, GLU_RGB10_A2, GLU_RGBA12, or GLU_RGBA16.

width Specifies the width, in pixels, of the texture image.

format Specifies the format of the pixel data. Must be one of GLU_COLOR_INDEX, GLU_DEPTH_COMPONENT, GLU_RED, GLU_GREEN, GLU_BLUE, GLU_ALPHA, GLU_RGB, GLU_RGBA, GLU_BGR, GLU_BGRA, GLU_LUMINANCE, or GLU_LUMINANCE_ALPHA.

type Specifies the data type for *data*. Must be one of GLU_UNSIGNED_BYTE, GLU_BYTE, GLU_BITMAP, GLU_UNSIGNED_SHORT, GLU_SHORT, GLU_UNSIGNED_INT, GLU_INT, GLU_FLOAT, GLU_UNSIGNED_BYTE_3_3_2, GLU_UNSIGNED_BYTE_2_3_3_REV, GLU_UNSIGNED_SHORT_5_6_5, GLU_UNSIGNED_SHORT_5_6_5_REV, GLU_UNSIGNED_SHORT_4_4_4_4, GLU_UNSIGNED_SHORT_4_4_4_4_REV, GLU_UNSIGNED_SHORT_5_5_5_1, GLU_UNSIGNED_SHORT_1_5_5_5_REV, GLU_UNSIGNED_INT_8_8_8_8, GLU_UNSIGNED_INT_8_8_8_8_REV, GLU_UNSIGNED_INT_10_10_10_2, or GLU_UNSIGNED_INT_2_10_10_10_REV.

data Specifies a pointer to the image data in memory.

gluBuild1DMipmaps builds a series of prefiltered one-dimensional texture maps of decreasing resolutions called a mipmap. This is used for the antialiasing of texture mapped primitives.

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

Initially, the *width* of *data* is checked to see if it is a power of 2. If not, a copy of *data* is scaled up or down to the nearest power of 2. (If *width* is exactly between powers of 2, then the copy of *data* will scale upwards.) This copy will be used for subsequent mipmapping operations described below. For example, if *width* is 57, then a copy of *data* will scale up to 64 before mipmapping takes place.

Then, proxy textures (see glTexImage1D) are used to determine if the implementation can fit the requested texture. If not, *width* is continually halved until it fits.

Next, a series of mipmap levels is built by decimating a copy of *data* in half until size 11 is reached. At each level, each texel in the halved mipmap level is an average of the corresponding two texels in the larger mipmap level.

glTexImage1D is called to load each of these mipmap levels. Level 0 is a copy of *data*. The highest level is *log_2,(width,)*. For example, if *width* is 64 and the implementation can store a texture of this size, the following mipmap levels are built: 641, 321, 161, 81, 41, 21, and 11. These correspond to levels 0 through 6, respectively.

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

GLU_INVALID_VALUE is returned if *width* is < 1.

GLU_INVALID_ENUM is returned if *format* or *type* are not legal.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_BYTE_3_3_2 or GLU_UNSIGNED_BYTE_2_3_3_REV and *format* is not GLU_RGB.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_SHORT_5_6_5 or GLU_UNSIGNED_SHORT_5_6_5_REV and *format* is not GLU_RGB.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_SHORT_4_4_4_4 or GLU_UNSIGNED_SHORT_4_4_4_4_REV and *format* is neither GLU_RGBA nor GLU_BGRA.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_SHORT_5_5_5_1 or GLU_UNSIGNED_SHORT_1_5_5_5_REV and *format* is neither GLU_RGBA nor GLU_BGRA.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_INT_8_8_8_8 or GLU_UNSIGNED_INT_8_8_8_8_REV and *format* is neither GLU_RGBA nor GLU_BGRA.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_INT_10_10_10_2 or GLU_UNSIGNED_INT_2_10_10_10_REV and *format* is neither GLU_RGBA nor GLU_BGRA.

GLint gluBuild2DMipmapLevels *target internalFormat width height format type level base max data* [Function]

Builds a subset of two-dimensional mipmap levels.

target Specifies the target texture. Must be GLU_TEXTURE_2D.

internalFormat
Requests the internal storage format of the texture image. The most current version of the SGI implementation of GLU does not check this value for validity before passing it on to the underlying OpenGL implementation. A value that is not accepted by the OpenGL implementation will lead to an OpenGL error. The benefit of not checking this value at the GLU level is that OpenGL extensions can add new internal texture formats without requiring a revision of the GLU implementation. Older implementations of GLU check this value and raise a GLU error if it is not 1, 2, 3, or 4 or one of the following symbolic constants: GLU_ALPHA, GLU_ALPHA4, GLU_ALPHA8, GLU_ALPHA12, GLU_ALPHA16, GLU_LUMINANCE, GLU_LUMINANCE4, GLU_LUMINANCE8, GLU_LUMINANCE12, GLU_LUMINANCE16, GLU_LUMINANCE_ALPHA, GLU_LUMINANCE4_ALPHA4, GLU_LUMINANCE6_ALPHA2, GLU_LUMINANCE8_ALPHA8, GLU_LUMINANCE12_ALPHA4, GLU_LUMINANCE12_ALPHA12, GLU_LUMINANCE16_ALPHA16, GLU_INTENSITY, GLU_INTENSITY4, GLU_INTENSITY8, GLU_INTENSITY12, GLU_INTENSITY16, GLU_RGB, GLU_R3_G3_B2, GLU_RGB4, GLU_RGB5, GLU_RGB8, GLU_RGB10, GLU_RGB12, GLU_RGB16, GLU_RGBA, GLU_RGBA2, GLU_RGBA4, GLU_RGB5_A1, GLU_RGBA8, GLU_RGB10_A2, GLU_RGBA12, or GLU_RGBA16.

width
height Specifies the width and height, respectively, in pixels of the texture image. These should be a power of 2.

format Specifies the format of the pixel data. Must be one of GLU_COLOR_INDEX, GLU_DEPTH_COMPONENT, GLU_RED, GLU_GREEN, GLU_BLUE, GLU_ALPHA,

GLU_RGB, GLU_RGBA, GLU_BGR, GLU_BGRA, GLU_LUMINANCE, or GLU_LUMINANCE_ALPHA.

type Specifies the data type for *data*. Must be one of GLU_UNSIGNED_BYTE, GLU_BYTE, GLU_BITMAP, GLU_UNSIGNED_SHORT, GLU_SHORT, GLU_UNSIGNED_INT, GLU_INT, GLU_FLOAT, GLU_UNSIGNED_BYTE_3_3_2, GLU_UNSIGNED_BYTE_2_3_3_REV, GLU_UNSIGNED_SHORT_5_6_5, GLU_UNSIGNED_SHORT_5_6_5_REV, GLU_UNSIGNED_SHORT_4_4_4_4, GLU_UNSIGNED_SHORT_4_4_4_4_REV, GLU_UNSIGNED_SHORT_5_5_5_1, GLU_UNSIGNED_SHORT_1_5_5_5_REV, GLU_UNSIGNED_INT_8_8_8_8, GLU_UNSIGNED_INT_8_8_8_8_REV, GLU_UNSIGNED_INT_10_10_10_2, or GLU_UNSIGNED_INT_2_10_10_10_REV.

level Specifies the mipmap level of the image data.

base Specifies the minimum mipmap level to pass to glTexImage2D.

max Specifies the maximum mipmap level to pass to glTexImage2D.

data Specifies a pointer to the image data in memory.

gluBuild2DMipmapLevels builds a subset of prefiltered two-dimensional texture maps of decreasing resolutions called a mipmap. This is used for the antialiasing of texture mapped primitives.

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

A series of mipmap levels from *base* to *max* is built by decimating *data* in half along both dimensions until size 11 is reached. At each level, each texel in the halved mipmap level is an average of the corresponding four texels in the larger mipmap level. (In the case of rectangular images, the decimation will ultimately reach an $N1$ or $1N$ configuration. Here, two texels are averaged instead.) glTexImage2D is called to load these mipmap levels from *base* to *max*. If *max* is larger than the highest mipmap level for the texture of the specified size, then a GLU error code is returned (see gluErrorString) and nothing is loaded.

For example, if *level* is 2 and *width* is 16 and *height* is 8, the following levels are possible: 168, 84, 42, 21, 11. These correspond to levels 2 through 6 respectively. If *base* is 3 and *max* is 5, then only mipmap levels 84, 42, and 21 are loaded. However, if *max* is 7, then an error is returned and nothing is loaded since *max* is larger than the highest mipmap level which is, in this case, 6.

The highest mipmap level can be derived from the formula *log_2(max(width,height)2^level,)*.

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

GLU_INVALID_VALUE is returned if *level* > *base*, *base* < 0, *max* < *base*, or *max* is > the highest mipmap level for *data*.

GLU_INVALID_VALUE is returned if *width* or *height* is < 1.

GLU_INVALID_ENUM is returned if *internalFormat*, *format*, or *type* is not legal.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_BYTE_3_3_2 or GLU_UNSIGNED_BYTE_2_3_3_REV and *format* is not GLU_RGB.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_SHORT_5_6_5 or GLU_UNSIGNED_SHORT_5_6_5_REV and *format* is not GLU_RGB.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_SHORT_4_4_4_4 or GLU_UNSIGNED_SHORT_4_4_4_4_REV and *format* is neither GLU_RGBA nor GLU_BGRA.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_SHORT_5_5_5_1 or GLU_UNSIGNED_SHORT_1_5_5_5_REV and *format* is neither GLU_RGBA nor GLU_BGRA.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_INT_8_8_8_8 or GLU_UNSIGNED_INT_8_8_8_8_REV and *format* is neither GLU_RGBA nor GLU_BGRA.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_INT_10_10_10_2 or GLU_UNSIGNED_INT_2_10_10_10_REV and *format* is neither GLU_RGBA nor GLU_BGRA.

GLint gluBuild2DMipmaps *target internalFormat width height format type data* [Function]

Builds a two-dimensional mipmap.

target Specifies the target texture. Must be GLU_TEXTURE_2D.

internalFormat
Requests the internal storage format of the texture image. The most current version of the SGI implementation of GLU does not check this value for validity before passing it on to the underlying OpenGL implementation. A value that is not accepted by the OpenGL implementation will lead to an OpenGL error. The benefit of not checking this value at the GLU level is that OpenGL extensions can add new internal texture formats without requiring a revision of the GLU implementation. Older implementations of GLU check this value and raise a GLU error if it is not 1, 2, 3, or 4 or one of the following symbolic constants: GLU_ALPHA, GLU_ALPHA4, GLU_ALPHA8, GLU_ALPHA12, GLU_ALPHA16, GLU_LUMINANCE, GLU_LUMINANCE4, GLU_LUMINANCE8, GLU_LUMINANCE12, GLU_LUMINANCE16, GLU_LUMINANCE_ALPHA, GLU_LUMINANCE4_ALPHA4, GLU_LUMINANCE6_ALPHA2, GLU_LUMINANCE8_ALPHA8, GLU_LUMINANCE12_ALPHA4, GLU_LUMINANCE12_ALPHA12, GLU_LUMINANCE16_ALPHA16, GLU_INTENSITY, GLU_INTENSITY4, GLU_INTENSITY8, GLU_INTENSITY12, GLU_INTENSITY16, GLU_RGB, GLU_R3_G3_B2, GLU_RGB4, GLU_RGB5, GLU_RGB8, GLU_RGB10, GLU_RGB12, GLU_RGB16, GLU_RGBA, GLU_RGBA2, GLU_RGBA4, GLU_RGB5_A1, GLU_RGBA8, GLU_RGB10_A2, GLU_RGBA12, or GLU_RGBA16.

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

format Specifies the format of the pixel data. Must be one of GLU_COLOR_INDEX, GLU_DEPTH_COMPONENT, GLU_RED, GLU_GREEN, GLU_BLUE, GLU_ALPHA, GLU_RGB, GLU_RGBA, GLU_BGR, GLU_BGRA, GLU_LUMINANCE, or GLU_LUMINANCE_ALPHA.

type Specifies the data type for *data*. Must be one of GLU_UNSIGNED_BYTE, GLU_BYTE, GLU_BITMAP, GLU_UNSIGNED_SHORT, GLU_SHORT, GLU_UNSIGNED_INT, GLU_INT, GLU_FLOAT, GLU_UNSIGNED_BYTE_3_3_2,

GLU_UNSIGNED_BYTE_2_3_3_REV, GLU_UNSIGNED_SHORT_5_6_5, GLU_UNSIGNED_SHORT_5_6_5_REV, GLU_UNSIGNED_SHORT_4_4_4_4, GLU_UNSIGNED_SHORT_4_4_4_4_REV, GLU_UNSIGNED_SHORT_5_5_5_1, GLU_UNSIGNED_SHORT_1_5_5_5_REV, GLU_UNSIGNED_INT_8_8_8_8, GLU_UNSIGNED_INT_8_8_8_8_REV, GLU_UNSIGNED_INT_10_10_10_2, or GLU_UNSIGNED_INT_2_10_10_10_REV.

data Specifies a pointer to the image data in memory.

gluBuild2DMipmaps builds a series of prefiltered two-dimensional texture maps of decreasing resolutions called a mipmap. This is used for the antialiasing of texture-mapped primitives.

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

Initially, the *width* and *height* of *data* are checked to see if they are a power of 2. If not, a copy of *data* (not *data*), is scaled up or down to the nearest power of 2. This copy will be used for subsequent mipmapping operations described below. (If *width* or *height* is exactly between powers of 2, then the copy of *data* will scale upwards.) For example, if *width* is 57 and *height* is 23, then a copy of *data* will scale up to 64 in *width* and down to 16 in depth, before mipmapping takes place.

Then, proxy textures (see glTexImage2D) are used to determine if the implementation can fit the requested texture. If not, both dimensions are continually halved until it fits. (If the OpenGL version is \(<= 1.0, both maximum texture dimensions are clamped to the value returned by glGetIntegerv with the argument GLU_MAX_TEXTURE_SIZE.)

Next, a series of mipmap levels is built by decimating a copy of *data* in half along both dimensions until size 11 is reached. At each level, each texel in the halved mipmap level is an average of the corresponding four texels in the larger mipmap level. (In the case of rectangular images, the decimation will ultimately reach an $N1$ or $1N$ configuration. Here, two texels are averaged instead.)

glTexImage2D is called to load each of these mipmap levels. Level 0 is a copy of *data*. The highest level is *log_2,(max(width,height),)*. For example, if *width* is 64 and *height* is 16 and the implementation can store a texture of this size, the following mipmap levels are built: 6416, 328, 164, 82, 41, 21, and 11 These correspond to levels 0 through 6, respectively.

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

GLU_INVALID_VALUE is returned if *width* or *height* is < 1.

GLU_INVALID_ENUM is returned if *internalFormat*, *format*, or *type* is not legal.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_BYTE_3_3_2 or GLU_UNSIGNED_BYTE_2_3_3_REV and *format* is not GLU_RGB.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_SHORT_5_6_5 or GLU_UNSIGNED_SHORT_5_6_5_REV and *format* is not GLU_RGB.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_SHORT_4_4_4_4 or GLU_UNSIGNED_SHORT_4_4_4_4_REV and *format* is neither GLU_RGBA nor GLU_BGRA.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_SHORT_5_5_5_1 or GLU_UNSIGNED_SHORT_1_5_5_5_REV and *format* is neither GLU_RGBA nor GLU_BGRA.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_INT_8_8_8_8 or GLU_UNSIGNED_INT_8_8_8_8_REV and *format* is neither GLU_RGBA nor GLU_BGRA.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_INT_10_10_10_2 or GLU_UNSIGNED_INT_2_10_10_10_REV and *format* is neither GLU_RGBA nor GLU_BGRA.

GLint gluBuild3DMipmapLevels *target internalFormat width height depth format type level base max data* [Function]

Builds a subset of three-dimensional mipmap levels.

target Specifies the target texture. Must be GLU_TEXTURE_3D.

internalFormat
Requests the internal storage format of the texture image. The most current version of the SGI implementation of GLU does not check this value for validity before passing it on to the underlying OpenGL implementation. A value that is not accepted by the OpenGL implementation will lead to an OpenGL error. The benefit of not checking this value at the GLU level is that OpenGL extensions can add new internal texture formats without requiring a revision of the GLU implementation. Older implementations of GLU check this value and raise a GLU error if it is not 1, 2, 3, or 4 or one of the following symbolic constants: GLU_ALPHA, GLU_ALPHA4, GLU_ALPHA8, GLU_ALPHA12, GLU_ALPHA16, GLU_LUMINANCE, GLU_LUMINANCE4, GLU_LUMINANCE8, GLU_LUMINANCE12, GLU_LUMINANCE16, GLU_LUMINANCE_ALPHA, GLU_LUMINANCE4_ALPHA4, GLU_LUMINANCE6_ALPHA2, GLU_LUMINANCE8_ALPHA8, GLU_LUMINANCE12_ALPHA4, GLU_LUMINANCE12_ALPHA12, GLU_LUMINANCE16_ALPHA16, GLU_INTENSITY, GLU_INTENSITY4, GLU_INTENSITY8, GLU_INTENSITY12, GLU_INTENSITY16, GLU_RGB, GLU_R3_G3_B2, GLU_RGB4, GLU_RGB5, GLU_RGB8, GLU_RGB10, GLU_RGB12, GLU_RGB16, GLU_RGBA, GLU_RGBA2, GLU_RGBA4, GLU_RGB5_A1, GLU_RGBA8, GLU_RGB10_A2, GLU_RGBA12, or GLU_RGBA16.

width
height
depth Specifies in pixels the width, height and depth respectively, of the texture image. These should be a power of 2.

format Specifies the format of the pixel data. Must be one of GLU_COLOR_INDEX, GLU_DEPTH_COMPONENT, GLU_RED, GLU_GREEN, GLU_BLUE, GLU_ALPHA, GLU_RGB, GLU_RGBA, GLU_BGR, GLU_BGRA, GLU_LUMINANCE, or GLU_LUMINANCE_ALPHA.

type Specifies the data type for *data*. Must be one of GLU_UNSIGNED_BYTE, GLU_BYTE, GLU_BITMAP, GLU_UNSIGNED_SHORT, GLU_SHORT, GLU_UNSIGNED_INT, GLU_INT, GLU_FLOAT, GLU_UNSIGNED_BYTE_3_3_2, GLU_UNSIGNED_BYTE_2_3_3_REV, GLU_UNSIGNED_SHORT_5_6_5, GLU_UNSIGNED_SHORT_5_6_5_REV, GLU_UNSIGNED_SHORT_4_4_4_4, GLU_UNSIGNED_SHORT_4_4_4_4_REV, GLU_UNSIGNED_SHORT_5_5_5_1,

GLU_UNSIGNED_SHORT_1_5_5_5_REV, GLU_UNSIGNED_INT_8_8_8_8, GLU_UNSIGNED_INT_8_8_8_8_REV, GLU_UNSIGNED_INT_10_10_10_2, or GLU_UNSIGNED_INT_2_10_10_10_REV.

level Specifies the mipmap level of the image data.

base Specifies the minimum mipmap level to pass to glTexImage3D.

max Specifies the maximum mipmap level to pass to glTexImage3D.

data Specifies a pointer to the image data in memory.

gluBuild3DMipmapLevels builds a subset of prefiltered three-dimensional texture maps of decreasing resolutions called a mipmap. This is used for the antialiasing of texture mapped primitives.

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

A series of mipmap levels from *base* to *max* is built by decimating *data* in half along both dimensions until size 111 is reached. At each level, each texel in the halved mipmap level is an average of the corresponding eight texels in the larger mipmap level. (If exactly one of the dimensions is 1, four texels are averaged. If exactly two of the dimensions are 1, two texels are averaged.) glTexImage3D is called to load these mipmap levels from *base* to *max*. If *max* is larger than the highest mipmap level for the texture of the specified size, then a GLU error code is returned (see gluErrorString) and nothing is loaded.

For example, if *level* is 2 and *width* is 16, *height* is 8 and *depth* is 4, the following levels are possible: 1684, 842, 421, 211, 111. These correspond to levels 2 through 6 respectively. If *base* is 3 and *max* is 5, then only mipmap levels 842, 421, and 211 are loaded. However, if *max* is 7, then an error is returned and nothing is loaded, since *max* is larger than the highest mipmap level which is, in this case, 6.

The highest mipmap level can be derived from the formula *log_2(max(width,heightdepth)2^level,)*.

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

GLU_INVALID_VALUE is returned if *level* > *base*, *base* < 0, *max* < *base*, or *max* is > the highest mipmap level for *data*.

GLU_INVALID_VALUE is returned if *width*, *height*, or *depth* is < 1.

GLU_INVALID_ENUM is returned if *internalFormat*, *format*, or *type* is not legal.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_BYTE_3_3_2 or GLU_UNSIGNED_BYTE_2_3_3_REV and *format* is not GLU_RGB.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_SHORT_5_6_5 or GLU_UNSIGNED_SHORT_5_6_5_REV and *format* is not GLU_RGB.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_SHORT_4_4_4_4 or GLU_UNSIGNED_SHORT_4_4_4_4_REV and *format* is neither GLU_RGBA nor GLU_BGRA.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_SHORT_5_5_5_1 or GLU_UNSIGNED_SHORT_1_5_5_5_REV and *format* is neither GLU_RGBA nor GLU_BGRA.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_INT_8_8_8_8 or GLU_UNSIGNED_INT_8_8_8_8_REV and *format* is neither GLU_RGBA nor GLU_BGRA.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_INT_10_10_10_2 or GLU_UNSIGNED_INT_2_10_10_10_REV and *format* is neither GLU_RGBA nor GLU_BGRA.

GLint gluBuild3DMipmaps *target internalFormat width height depth format type data* [Function]

Builds a three-dimensional mipmap.

target Specifies the target texture. Must be GLU_TEXTURE_3D.

internalFormat
Requests the internal storage format of the texture image. The most current version of the SGI implementation of GLU does not check this value for validity before passing it on to the underlying OpenGL implementation. A value that is not accepted by the OpenGL implementation will lead to an OpenGL error. The benefit of not checking this value at the GLU level is that OpenGL extensions can add new internal texture formats without requiring a revision of the GLU implementation. Older implementations of GLU check this value and raise a GLU error if it is not 1, 2, 3, or 4 or one of the following symbolic constants: GLU_ALPHA, GLU_ALPHA4, GLU_ALPHA8, GLU_ALPHA12, GLU_ALPHA16, GLU_LUMINANCE, GLU_LUMINANCE4, GLU_LUMINANCE8, GLU_LUMINANCE12, GLU_LUMINANCE16, GLU_LUMINANCE_ALPHA, GLU_LUMINANCE4_ALPHA4, GLU_LUMINANCE6_ALPHA2, GLU_LUMINANCE8_ALPHA8, GLU_LUMINANCE12_ALPHA4, GLU_LUMINANCE12_ALPHA12, GLU_LUMINANCE16_ALPHA16, GLU_INTENSITY, GLU_INTENSITY4, GLU_INTENSITY8, GLU_INTENSITY12, GLU_INTENSITY16, GLU_RGB, GLU_R3_G3_B2, GLU_RGB4, GLU_RGB5, GLU_RGB8, GLU_RGB10, GLU_RGB12, GLU_RGB16, GLU_RGBA, GLU_RGBA2, GLU_RGBA4, GLU_RGB5_A1, GLU_RGBA8, GLU_RGB10_A2, GLU_RGBA12, or GLU_RGBA16.

width
height
depth Specifies in pixels the width, height and depth respectively, in pixels of the texture image.

format Specifies the format of the pixel data. Must be one of GLU_COLOR_INDEX, GLU_DEPTH_COMPONENT, GLU_RED, GLU_GREEN, GLU_BLUE, GLU_ALPHA, GLU_RGB, GLU_RGBA, GLU_BGR, GLU_BGRA, GLU_LUMINANCE, or GLU_LUMINANCE_ALPHA.

type Specifies the data type for *data*. Must be one of: GLU_UNSIGNED_BYTE, GLU_BYTE, GLU_BITMAP, GLU_UNSIGNED_SHORT, GLU_SHORT, GLU_UNSIGNED_INT, GLU_INT, GLU_FLOAT, GLU_UNSIGNED_BYTE_3_3_2, GLU_UNSIGNED_BYTE_2_3_3_REV, GLU_UNSIGNED_SHORT_5_6_5, GLU_UNSIGNED_SHORT_5_6_5_REV, GLU_UNSIGNED_SHORT_4_4_4_4, GLU_UNSIGNED_SHORT_4_4_4_4_REV, GLU_UNSIGNED_SHORT_5_5_5_1, GLU_UNSIGNED_SHORT_1_5_5_5_REV, GLU_UNSIGNED_INT_8_8_8_8,

GLU_UNSIGNED_INT_8_8_8_8_REV, GLU_UNSIGNED_INT_10_10_10_2, or GLU_UNSIGNED_INT_2_10_10_10_REV.

data Specifies a pointer to the image data in memory.

gluBuild3DMipmaps builds a series of prefiltered three-dimensional texture maps of decreasing resolutions called a mipmap. This is used for the antialiasing of texture-mapped primitives.

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

Initially, the *width*, *height* and *depth* of *data* are checked to see if they are a power of 2. If not, a copy of *data* is made and scaled up or down to the nearest power of 2. (If *width*, *height*, or *depth* is exactly between powers of 2, then the copy of *data* will scale upwards.) This copy will be used for subsequent mipmapping operations described below. For example, if *width* is 57, *height* is 23, and *depth* is 24, then a copy of *data* will scale up to 64 in width, down to 16 in height, and up to 32 in depth before mipmapping takes place.

Then, proxy textures (see glTexImage3D) are used to determine if the implementation can fit the requested texture. If not, all three dimensions are continually halved until it fits.

Next, a series of mipmap levels is built by decimating a copy of *data* in half along all three dimensions until size 111 is reached. At each level, each texel in the halved mipmap level is an average of the corresponding eight texels in the larger mipmap level. (If exactly one of the dimensions is 1, four texels are averaged. If exactly two of the dimensions are 1, two texels are averaged.)

glTexImage3D is called to load each of these mipmap levels. Level 0 is a copy of *data*. The highest level is $log\_2,(max(width,heightdepth),)$. For example, if *width* is 64, *height* is 16, and *depth* is 32, and the implementation can store a texture of this size, the following mipmap levels are built: 641632, 32816, 1648, 824, 412, 211, and 111. These correspond to levels 0 through 6, respectively.

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

GLU_INVALID_VALUE is returned if *width*, *height*, or *depth* is < 1.

GLU_INVALID_ENUM is returned if *internalFormat*, *format*, or *type* is not legal.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_BYTE_3_3_2 or GLU_UNSIGNED_BYTE_2_3_3_REV and *format* is not GLU_RGB.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_SHORT_5_6_5 or GLU_UNSIGNED_SHORT_5_6_5_REV and *format* is not GLU_RGB.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_SHORT_4_4_4_4 or GLU_UNSIGNED_SHORT_4_4_4_4_REV and *format* is neither GLU_RGBA nor GLU_BGRA.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_SHORT_5_5_5_1 or GLU_UNSIGNED_SHORT_1_5_5_5_REV and *format* is neither GLU_RGBA nor GLU_BGRA.

GLU_INVALID_OPERATION is returned if *type* is GLU_UNSIGNED_INT_8_8_8_8 or GLU_UNSIGNED_INT_8_8_8_8_REV and *format* is neither GLU_RGBA nor GLU_BGRA.

`GLU_INVALID_OPERATION` is returned if *type* is `GLU_UNSIGNED_INT_10_10_10_2` or `GLU_UNSIGNED_INT_2_10_10_10_REV` and *format* is neither `GLU_RGBA` nor `GLU_BGRA`.

`GLboolean gluCheckExtension` *extName extString* [Function]

Determines if an extension name is supported.

extName Specifies an extension name.

extString Specifies a space-separated list of extension names supported.

`gluCheckExtension` returns `GLU_TRUE` if *extName* is supported otherwise `GLU_FALSE` is returned.

This is used to check for the presence for OpenGL, GLU, or GLX extension names by passing the extension strings returned by `glGetString`, `gluGetString`, `glXGetClientString`, `glXQueryExtensionsString`, or `glXQueryServerString`, respectively, as *extString*.

`void gluCylinder` *quad base top height slices stacks* [Function]

Draw a cylinder.

quad Specifies the quadrics object (created with `gluNewQuadric`).

base Specifies the radius of the cylinder at $z = 0$.

top Specifies the radius of the cylinder at $z =$ *height*.

height Specifies the height of the cylinder.

slices Specifies the number of subdivisions around the z axis.

stacks Specifies the number of subdivisions along the z axis.

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

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

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

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

`void gluDeleteNurbsRenderer` *nurb* [Function]

Destroy a NURBS object.

nurb Specifies the NURBS object to be destroyed.

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

`void gluDeleteQuadric` *quad* [Function]

Destroy a quadrics object.

quad Specifies the quadrics object to be destroyed.

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

`void gluDeleteTess` *tess* [Function]

Destroy a tessellation object.

tess Specifies the tessellation object to destroy.

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

`void gluDisk` *quad inner outer slices loops* [Function]

Draw a disk.

quad Specifies the quadrics object (created with `gluNewQuadric`).

inner Specifies the inner radius of the disk (may be 0).

outer Specifies the outer radius of the disk.

slices Specifies the number of subdivisions around the z axis.

loops Specifies the number of concentric rings about the origin into which the disk is subdivided.

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

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

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

`const-GLubyte-* gluErrorString` *error* [Function]

Produce an error string from a GL or GLU error code.

error Specifies a GL or GLU error code.

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

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

`NULL` is returned if *error* is not a valid GL or GLU error code.

`void gluGetNurbsProperty` *nurb property data* [Function]
Get a NURBS property.

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

property Specifies the property whose value is to be fetched. Valid values are `GLU_CULLING`, `GLU_SAMPLING_TOLERANCE`, `GLU_DISPLAY_MODE`, `GLU_AUTO_LOAD_MATRIX`, `GLU_PARAMETRIC_TOLERANCE`, `GLU_SAMPLING_METHOD`, `GLU_U_STEP`, `GLU_V_STEP`, and `GLU_NURBS_MODE`.

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

`gluGetNurbsProperty` retrieves properties stored in a NURBS object. These properties affect the way that NURBS curves and surfaces are rendered. See the `gluNurbsProperty` reference page for information about what the properties are and what they do.

`const-GLubyte-* gluGetString` *name* [Function]
Return a string describing the GLU version or GLU extensions .

name Specifies a symbolic constant, one of `GLU_VERSION`, or `GLU_EXTENSIONS`.

`gluGetString` returns a pointer to a static string describing the GLU version or the GLU extensions that are supported.

The version number is one of the following forms:

major_number.minor_numbermajor_number.minor_number.release_number.

The version string is of the following form:

version number<space>vendor-specific information

Vendor-specific information is optional. Its format and contents depend on the implementation.

The standard GLU contains a basic set of features and capabilities. If a company or group of companies wish to support other features, these may be included as extensions to the GLU. If *name* is `GLU_EXTENSIONS`, then `gluGetString` returns a space-separated list of names of supported GLU extensions. (Extension names never contain spaces.)

All strings are null-terminated.

NULL is returned if *name* is not `GLU_VERSION` or `GLU_EXTENSIONS`.

`void gluGetTessProperty` *tess which data* [Function]
Get a tessellation object property.

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

which Specifies the property whose value is to be fetched. Valid values are `GLU_TESS_WINDING_RULE`, `GLU_TESS_BOUNDARY_ONLY`, and `GLU_TESS_TOLERANCE`.

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

`gluGetTessProperty` retrieves properties stored in a tessellation object. These properties affect the way that tessellation objects are interpreted and rendered. See the `gluTessProperty` reference page for information about the properties and what they do.

`void gluLoadSamplingMatrices` *nurb model perspective view* [Function]

Load NURBS sampling and culling matrices.

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

model Specifies a modelview matrix (as from a `glGetFloatv` call).

perspective
Specifies a projection matrix (as from a `glGetFloatv` call).

view Specifies a viewport (as from a `glGetIntegerv` call).

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

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

`void gluLookAt` *eyeX eyeY eyeZ centerX centerY centerZ upX upY upZ* [Function]

Define a viewing transformation.

eyeX
eyeY
eyeZ Specifies the position of the eye point.

centerX
centerY
centerZ Specifies the position of the reference point.

upX
upY
upZ Specifies the direction of the *up* vector.

`gluLookAt` creates a viewing matrix derived from an eye point, a reference point indicating the center of the scene, and an *UP* vector.

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

Let

F=((*centerX*-*eyeX*), (*centerY*-*eyeY*), (*centerZ*-*eyeZ*),)

Let UP be the vector (upX,upY upZ).

Then normalize as follows: $f=F/F$,,

$UP\hat{}=UP/UP$,,

Finally, let $s=fUP\hat{}$, and $u=sf$.

M is then constructed as follows: M=((s[0,] s[1,] s[2,] 0), (u[0,] u[1,] u[2,] 0), (-f[0,] -f[1,] -f[2,] 0), (0 0 0 1),)

and `gluLookAt` is equivalent to

```
glMultMatrixf(M);
glTranslated(-eyex, -eyey, -eyez);
```

`GLUnurbs* gluNewNurbsRenderer` [Function]

Create a NURBS object.

`gluNewNurbsRenderer` creates and returns a pointer to a new NURBS object. This object must be referred to when calling NURBS rendering and control functions. A return value of 0 means that there is not enough memory to allocate the object.

`GLUquadric* gluNewQuadric` [Function]

Create a quadrics object.

`gluNewQuadric` creates and returns a pointer to a new quadrics object. This object must be referred to when calling quadrics rendering and control functions. A return value of 0 means that there is not enough memory to allocate the object.

`GLUtesselator* gluNewTess` [Function]

Create a tessellation object.

`gluNewTess` creates and returns a pointer to a new tessellation object. This object must be referred to when calling tessellation functions. A return value of 0 means that there is not enough memory to allocate the object.

`void gluNextContour` *tess type* [Function]

Mark the beginning of another contour.

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

type Specifies the type of the contour being defined. Valid values are `GLU_EXTERIOR`, `GLU_INTERIOR`, `GLU_UNKNOWN`, `GLU_CCW`, and `GLU_CW`.

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

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

`GLU_EXTERIOR`
: An exterior contour defines an exterior boundary of the polygon.

`GLU_INTERIOR`
: An interior contour defines an interior boundary of the polygon (such as a hole).

`GLU_UNKNOWN`
: An unknown contour is analyzed by the library to determine if it is interior or exterior.

`GLU_CCW`,

`GLU_CW`
: The first `GLU_CCW` or `GLU_CW` contour defined is considered to be exterior. All other contours are considered to be exterior if they are oriented in the same direction (clockwise or counterclockwise) as the first contour, and interior if they are not.

If one contour is of type `GLU_CCW` or `GLU_CW`, then all contours must be of the same type (if they are not, then all `GLU_CCW` and `GLU_CW` contours will be changed to `GLU_UNKNOWN`).

Note that there is no real difference between the `GLU_CCW` and `GLU_CW` contour types.

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

This command is obsolete and is provided for backward compatibility only. Calls to `gluNextContour` are mapped to `gluTessEndContour` followed by `gluTessBeginContour`.

`void gluNurbsCallbackDataEXT` *nurb userData* [Function]

Set a user data pointer.

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

userData Specifies a pointer to the user's data.

`gluNurbsCallbackDataEXT` is used to pass a pointer to the application's data to NURBS tessellator. A copy of this pointer will be passed by the tessellator in the NURBS callback functions (set by `gluNurbsCallback`).

`void gluNurbsCallbackData` *nurb userData* [Function]

Set a user data pointer.

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

userData Specifies a pointer to the user's data.

`gluNurbsCallbackData` is used to pass a pointer to the application's data to NURBS tessellator. A copy of this pointer will be passed by the tessellator in the NURBS callback functions (set by `gluNurbsCallback`).

`void gluNurbsCallback` *nurb which CallBackFunc* [Function]

Define a callback for a NURBS object.

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

which Specifies the callback being defined. Valid values are `GLU_NURBS_BEGIN`, `GLU_NURBS_VERTEX`, `GLU_NURBS_NORMAL`, `GLU_NURBS_COLOR`, `GLU_NURBS_TEXTURE_COORD`, `GLU_NURBS_END`, `GLU_NURBS_BEGIN_DATA`, `GLU_NURBS_VERTEX_DATA`, `GLU_NURBS_NORMAL_DATA`, `GLU_NURBS_COLOR_DATA`, `GLU_NURBS_TEXTURE_COORD_DATA`, `GLU_NURBS_END_DATA`, and `GLU_NURBS_ERROR`.

CallBackFunc
Specifies the function that the callback calls.

`gluNurbsCallback` is used to define a callback to be used by a NURBS object. If the specified callback is already defined, then it is replaced. If *CallBackFunc* is NULL, then this callback will not get invoked and the related data, if any, will be lost.

Except the error callback, these callbacks are used by NURBS tessellator (when `GLU_NURBS_MODE` is set to be `GLU_NURBS_TESSELLATOR`) to return back the OpenGL polygon primitives resulting from the tessellation. Note that there are two versions of each callback: one with a user data pointer and one without. If both versions for a particular callback are specified then the callback with the user data pointer will be used. Note that "userData" is a copy of the pointer that was specified at the last call to `gluNurbsCallbackData`.

The error callback function is effective no matter which value that `GLU_NURBS_MODE` is set to. All other callback functions are effective only when `GLU_NURBS_MODE` is set to `GLU_NURBS_TESSELLATOR`.

The legal callbacks are as follows:

`GLU_NURBS_BEGIN`
The begin callback indicates the start of a primitive. The function takes a single argument of type GLenum, which can be one of `GLU_LINES`, `GLU_LINE_STRIP`, `GLU_TRIANGLE_FAN`, `GLU_TRIANGLE_STRIP`, `GLU_TRIANGLES`, or `GLU_QUAD_STRIP`. The default begin callback function is NULL. The function prototype for this callback looks like:

`GLU_NURBS_BEGIN_DATA`
The same as the `GLU_NURBS_BEGIN` callback except that it takes an additional pointer argument. This pointer is a copy of the pointer that was specified at the last call to `gluNurbsCallbackData`. The default callback function is NULL. The function prototype for this callback function looks like:

`GLU_NURBS_VERTEX`
The vertex callback indicates a vertex of the primitive. The coordinates of the vertex are stored in the parameter "vertex". All the generated vertices have dimension 3; that is, homogeneous coordinates have been transformed into affine coordinates. The default vertex callback function is NULL. The function prototype for this callback function looks like:

`GLU_NURBS_VERTEX_DATA`
This is the same as the `GLU_NURBS_VERTEX` callback, except that it takes an additional pointer argument. This pointer is a copy of the pointer

that was specified at the last call to `gluNurbsCallbackData`. The default callback function is NULL. The function prototype for this callback function looks like:

`GLU_NURBS_NORMAL`
: The normal callback is invoked as the vertex normal is generated. The components of the normal are stored in the parameter "normal." In the case of a NURBS curve, the callback function is effective only when the user provides a normal map (`GLU_MAP1_NORMAL`). In the case of a NURBS surface, if a normal map (`GLU_MAP2_NORMAL`) is provided, then the generated normal is computed from the normal map. If a normal map is not provided, then a surface normal is computed in a manner similar to that described for evaluators when `GLU_AUTO_NORMAL` is enabled. The default normal callback function is NULL. The function prototype for this callback function looks like:

`GLU_NURBS_NORMAL_DATA`
: The same as the `GLU_NURBS_NORMAL` callback except that it takes an additional pointer argument. This pointer is a copy of the pointer that was specified at the last call to `gluNurbsCallbackData`. The default callback function is NULL. The function prototype for this callback function looks like:

`GLU_NURBS_COLOR`
: The color callback is invoked as the color of a vertex is generated. The components of the color are stored in the parameter "color." This callback is effective only when the user provides a color map (`GLU_MAP1_COLOR_4` or `GLU_MAP2_COLOR_4`). "color" contains four components: R, G, B, A. The default color callback function is NULL. The prototype for this callback function looks like:

`GLU_NURBS_COLOR_DATA`
: The same as the `GLU_NURBS_COLOR` callback except that it takes an additional pointer argument. This pointer is a copy of the pointer that was specified at the last call to `gluNurbsCallbackData`. The default callback function is NULL. The function prototype for this callback function looks like:

`GLU_NURBS_TEXTURE_COORD`
: The texture callback is invoked as the texture coordinates of a vertex are generated. These coordinates are stored in the parameter "texCoord." The number of texture coordinates can be 1, 2, 3, or 4 depending on which type of texture map is specified (`GLU_MAP1_TEXTURE_COORD_1`, `GLU_MAP1_TEXTURE_COORD_2`, `GLU_MAP1_TEXTURE_COORD_3`, `GLU_MAP1_TEXTURE_COORD_4`, `GLU_MAP2_TEXTURE_COORD_1`, `GLU_MAP2_TEXTURE_COORD_2`, `GLU_MAP2_TEXTURE_COORD_3`, `GLU_MAP2_TEXTURE_COORD_4`). If no texture map is specified, this callback function will not be called. The default texture callback function is NULL. The function prototype for this callback function looks like:

`GLU_NURBS_TEXTURE_COORD_DATA`
: This is the same as the `GLU_NURBS_TEXTURE_COORD` callback, except that it takes an additional pointer argument. This pointer is a copy of the pointer that was specified at the last call to `gluNurbsCallbackData`. The default callback function is NULL. The function prototype for this callback function looks like:

`GLU_NURBS_END`
: The end callback is invoked at the end of a primitive. The default end callback function is NULL. The function prototype for this callback function looks like:

`GLU_NURBS_END_DATA`
: This is the same as the `GLU_NURBS_END` callback, except that it takes an additional pointer argument. This pointer is a copy of the pointer that was specified at the last call to `gluNurbsCallbackData`. The default callback function is NULL. The function prototype for this callback function looks like:

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

```
void begin( GLenum type );

void beginData(GLenum type, void *userData);

void vertex( GLfloat *vertex );

void vertexData( GLfloat *vertex, void *userData );

void normal( GLfloat *normal );

void normalData( GLfloat *normal, void *userData );

void color( GLfloat *color );

void colorData( GLfloat *color, void *userData );

void texCoord( GLfloat *texCoord );

void texCoordData( GLfloat *texCoord, void *userData );

void end( void );
```

```
void endData( void  *userData );
```

`void gluNurbsCurve` *nurb knotCount knots stride control order type* [Function]

Define the shape of a NURBS curve.

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

knotCount
Specifies the number of knots in *knots*. *knotCount* equals the number of control points plus the order.

knots Specifies an array of *knotCount* nondecreasing knot values.

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

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

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

type Specifies the type of the curve. If this curve is defined within a `gluBeginCurve`/`gluEndCurve` pair, then the type can be any of the valid one-dimensional evaluator types (such as `GLU_MAP1_VERTEX_3` or `GLU_MAP1_COLOR_4`). Between a `gluBeginTrim`/`gluEndTrim` pair, the only valid types are `GLU_MAP1_TRIM_2` and `GLU_MAP1_TRIM_3`.

Use `gluNurbsCurve` to describe a NURBS curve.

When `gluNurbsCurve` appears between a `gluBeginCurve`/`gluEndCurve` pair, it is used to describe a curve to be rendered. Positional, texture, and color coordinates are associated by presenting each as a separate `gluNurbsCurve` between a `gluBeginCurve`/`gluEndCurve` pair. No more than one call to `gluNurbsCurve` for each of color, position, and texture data can be made within a single `gluBeginCurve`/`gluEndCurve` pair. Exactly one call must be made to describe the position of the curve (a *type* of `GLU_MAP1_VERTEX_3` or `GLU_MAP1_VERTEX_4`).

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

`void gluNurbsProperty` *nurb property value* [Function]

Set a NURBS property.

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

property Specifies the property to be set. Valid values are `GLU_SAMPLING_TOLERANCE`, `GLU_DISPLAY_MODE`, `GLU_CULLING`, `GLU_AUTO_LOAD_MATRIX`, `GLU_PARAMETRIC_TOLERANCE`, `GLU_SAMPLING_METHOD`, `GLU_U_STEP`, `GLU_V_STEP`, or `GLU_NURBS_MODE`.

value Specifies the value of the indicated property. It may be a numeric value or one of GLU_OUTLINE_POLYGON, GLU_FILL, GLU_OUTLINE_PATCH, GLU_TRUE, GLU_FALSE, GLU_PATH_LENGTH, GLU_PARAMETRIC_ERROR, GLU_DOMAIN_DISTANCE, GLU_NURBS_RENDERER, or GLU_NURBS_TESSELLATOR.

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

GLU_NURBS_MODE

value should be set to be either GLU_NURBS_RENDERER or GLU_NURBS_TESSELLATOR. When set to GLU_NURBS_RENDERER, NURBS objects are tessellated into OpenGL primitives and sent to the pipeline for rendering. When set to GLU_NURBS_TESSELLATOR, NURBS objects are tessellated into OpenGL primitives but the vertices, normals, colors, and/or textures are retrieved back through a callback interface (see gluNurbsCallback). This allows the user to cache the tessellated results for further processing. The initial value is GLU_NURBS_RENDERER.

GLU_SAMPLING_METHOD

Specifies how a NURBS surface should be tessellated. *value* may be one of GLU_PATH_LENGTH, GLU_PARAMETRIC_ERROR, GLU_DOMAIN_DISTANCE, GLU_OBJECT_PATH_LENGTH, or GLU_OBJECT_PARAMETRIC_ERROR. When set to GLU_PATH_LENGTH, the surface is rendered so that the maximum length, in pixels, of the edges of the tessellation polygons is no greater than what is specified by GLU_SAMPLING_TOLERANCE.

GLU_PARAMETRIC_ERROR specifies that the surface is rendered in such a way that the value specified by GLU_PARAMETRIC_TOLERANCE describes the maximum distance, in pixels, between the tessellation polygons and the surfaces they approximate.

GLU_DOMAIN_DISTANCE allows users to specify, in parametric coordinates, how many sample points per unit length are taken in u, v direction.

GLU_OBJECT_PATH_LENGTH is similar to GLU_PATH_LENGTH except that it is view independent; that is, the surface is rendered so that the maximum length, in object space, of edges of the tessellation polygons is no greater than what is specified by GLU_SAMPLING_TOLERANCE.

GLU_OBJECT_PARAMETRIC_ERROR is similar to GLU_PARAMETRIC_ERROR except that it is view independent; that is, the surface is rendered in such a way that the value specified by GLU_PARAMETRIC_TOLERANCE describes the maximum distance, in object space, between the tessellation polygons and the surfaces they approximate.

The initial value of GLU_SAMPLING_METHOD is GLU_PATH_LENGTH.

GLU_SAMPLING_TOLERANCE

Specifies the maximum length, in pixels or in object space length unit, to use when the sampling method is set to GLU_PATH_LENGTH or GLU_OBJECT_PATH_LENGTH. The NURBS code is conservative when rendering a curve or surface, so the actual length can be somewhat shorter. The initial value is 50.0 pixels.

GLU_PARAMETRIC_TOLERANCE
: Specifies the maximum distance, in pixels or in object space length unit, to use when the sampling method is GLU_PARAMETRIC_ERROR or GLU_OBJECT_PARAMETRIC_ERROR. The initial value is 0.5.

GLU_U_STEP
: Specifies the number of sample points per unit length taken along the *u* axis in parametric coordinates. It is needed when GLU_SAMPLING_METHOD is set to GLU_DOMAIN_DISTANCE. The initial value is 100.

GLU_V_STEP
: Specifies the number of sample points per unit length taken along the *v* axis in parametric coordinate. It is needed when GLU_SAMPLING_METHOD is set to GLU_DOMAIN_DISTANCE. The initial value is 100.

GLU_DISPLAY_MODE
: *value* can be set to GLU_OUTLINE_POLYGON, GLU_FILL, or GLU_OUTLINE_PATCH. When GLU_NURBS_MODE is set to be GLU_NURBS_RENDERER, *value* defines how a NURBS surface should be rendered. When *value* is set to GLU_FILL, the surface is rendered as a set of polygons. When *value* is set to GLU_OUTLINE_POLYGON, the NURBS library draws only the outlines of the polygons created by tessellation. When *value* is set to GLU_OUTLINE_PATCH just the outlines of patches and trim curves defined by the user are drawn.

  When GLU_NURBS_MODE is set to be GLU_NURBS_TESSELLATOR, *value* defines how a NURBS surface should be tessellated. When GLU_DISPLAY_MODE is set to GLU_FILL or GLU_OUTLINE_POLYGON, the NURBS surface is tessellated into OpenGL triangle primitives that can be retrieved back through callback functions. If GLU_DISPLAY_MODE is set to GLU_OUTLINE_PATCH, only the outlines of the patches and trim curves are generated as a sequence of line strips that can be retrieved back through callback functions.

  The initial value is GLU_FILL.

GLU_CULLING
: *value* is a boolean value that, when set to GLU_TRUE, indicates that a NURBS curve should be discarded prior to tessellation if its control points lie outside the current viewport. The initial value is GLU_FALSE.

GLU_AUTO_LOAD_MATRIX
: *value* is a boolean value. When set to GLU_TRUE, the NURBS code downloads the projection matrix, the modelview matrix, and the viewport from the GL server to compute sampling and culling matrices for each NURBS curve that is rendered. Sampling and culling matrices are required to determine the tessellation of a NURBS surface into line segments or polygons and to cull a NURBS surface if it lies outside the viewport.

  If this mode is set to GLU_FALSE, then the program needs to provide a projection matrix, a modelview matrix, and a viewport for the NURBS renderer to use to construct sampling and culling matrices. This can be

done with the `gluLoadSamplingMatrices` function. This mode is initially set to `GLU_TRUE`. Changing it from `GLU_TRUE` to `GLU_FALSE` does not affect the sampling and culling matrices until `gluLoadSamplingMatrices` is called.

`void gluNurbsSurface` *nurb sKnotCount sKnots tKnotCount tKnots sStride tStride control sOrder tOrder type* [Function]

Define the shape of a NURBS surface.

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

sKnotCount
Specifies the number of knots in the parametric *u* direction.

sKnots Specifies an array of *sKnotCount* nondecreasing knot values in the parametric *u* direction.

tKnotCount
Specifies the number of knots in the parametric *v* direction.

tKnots Specifies an array of *tKnotCount* nondecreasing knot values in the parametric *v* direction.

sStride Specifies the offset (as a number of single-precision floating-point values) between successive control points in the parametric *u* direction in *control*.

tStride Specifies the offset (in single-precision floating-point values) between successive control points in the parametric *v* direction in *control*.

control Specifies an array containing control points for the NURBS surface. The offsets between successive control points in the parametric *u* and *v* directions are given by *sStride* and *tStride*.

sOrder Specifies the order of the NURBS surface in the parametric *u* direction. The order is one more than the degree, hence a surface that is cubic in *u* has a *u* order of 4.

tOrder Specifies the order of the NURBS surface in the parametric *v* direction. The order is one more than the degree, hence a surface that is cubic in *v* has a *v* order of 4.

type Specifies type of the surface. *type* can be any of the valid two-dimensional evaluator types (such as `GLU_MAP2_VERTEX_3` or `GLU_MAP2_COLOR_4`).

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

Positional, texture, and color coordinates are associated with a surface by presenting each as a separate `gluNurbsSurface` between a `gluBeginSurface`/`gluEndSurface` pair. No more than one call to `gluNurbsSurface` for each of color, position, and texture data can be made within a single `gluBeginSurface`/`gluEndSurface` pair.

Exactly one call must be made to describe the position of the surface (a *type* of `GLU_MAP2_VERTEX_3` or `GLU_MAP2_VERTEX_4`).

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

Note that a `gluNurbsSurface` with *sKnotCount* knots in the *u* direction and *tKnotCount* knots in the *v* direction with orders *sOrder* and *tOrder* must have (*sKnotCount* - *sOrder*) *times* (*tKnotCount* - *tOrder*) control points.

`void gluOrtho2D` *left right bottom top* [Function]

Define a 2D orthographic projection matrix.

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.

`gluOrtho2D` sets up a two-dimensional orthographic viewing region. This is equivalent to calling `glOrtho` with *near*=-1 and *far*=1.

`void gluPartialDisk` *quad inner outer slices loops start sweep* [Function]

Draw an arc of a disk.

quad Specifies a quadrics object (created with `gluNewQuadric`).

inner Specifies the inner radius of the partial disk (can be 0).

outer Specifies the outer radius of the partial disk.

slices Specifies the number of subdivisions around the *z* axis.

loops Specifies the number of concentric rings about the origin into which the partial disk is subdivided.

start Specifies the starting angle, in degrees, of the disk portion.

sweep Specifies the sweep angle, in degrees, of the disk portion.

`gluPartialDisk` renders a partial disk on the *z*=0 plane. A partial disk is similar to a full disk, except that only the subset of the disk from *start* through *start* + *sweep* is included (where 0 degrees is along the +\f2y\f axis, 90 degrees along the +*x* axis, 180 degrees along the \-*y* axis, and 270 degrees along the \-*x* axis).

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

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

If texturing is turned on (with `gluQuadricTexture`), texture coordinates are generated linearly such that where *r*=*outer*, the value at (*r*, 0, 0) is (1.0, 0.5), at (0, *r*, 0) it is (0.5, 1.0), at (\-*r*, 0, 0) it is (0.0, 0.5), and at (0, \-*r*, 0) it is (0.5, 0.0).

`void gluPerspective` *fovy aspect zNear zFar* [Function]

Set up a perspective projection matrix.

fovy Specifies the field of view angle, in degrees, in the *y* direction.

aspect Specifies the aspect ratio that determines the field of view in the *x* direction. The aspect ratio is the ratio of *x* (width) to *y* (height).

zNear Specifies the distance from the viewer to the near clipping plane (always positive).

zFar Specifies the distance from the viewer to the far clipping plane (always positive).

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

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

Given *f* defined as follows:

f=*cotangent*(*fovy*/2,) The generated matrix is

((*f*/*aspect* 0 0 0), (0 *f* 0 0), (0 0 *zFar*+*zNear*,/*zNear*-*zFar*, 2*zFarzNear*,/*zNear*-*zFar*,), (0 0 -1 0),)

`void gluPickMatrix` *x y delX delY viewport* [Function]

Define a picking region.

x
y Specify the center of a picking region in window coordinates.

delX
delY Specify the width and height, respectively, of the picking region in window coordinates.

viewport Specifies the current viewport (as from a `glGetIntegerv` call).

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

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

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

`GLint gluProject` *objX objY objZ model proj view winX winY winZ* [Function]

Map object coordinates to window coordinates.

objX
objY
objZ Specify the object coordinates.

model Specifies the current modelview matrix (as from a `glGetDoublev` call).

proj Specifies the current projection matrix (as from a `glGetDoublev` call).

view Specifies the current viewport (as from a `glGetIntegerv` call).

winX
winY
winZ Return the computed window coordinates.

`gluProject` transforms the specified object coordinates into window coordinates using *model*, *proj*, and *view*. The result is stored in *winX*, *winY*, and *winZ*. A return value of `GLU_TRUE` indicates success, a return value of `GLU_FALSE` indicates failure.

To compute the coordinates, let v=(*objX*,*objY objZ*1.0) represented as a matrix with 4 rows and 1 column. Then `gluProject` computes $v\hat{}$ as follows:

$v\hat{}=PMv$

where P is the current projection matrix *proj* and M is the current modelview matrix *model* (both represented as 44 matrices in column-major order).

The window coordinates are then computed as follows:

winX=*view*(0,)+*view*(2,)($v\hat{}$(0,)+1,)/2*winY*=*view*(1,)+*view*(3,)($v\hat{}$(1,)+1,)/2
winZ=($v\hat{}$(2,)+1,)/2

`void gluPwlCurve` *nurb count data stride type* [Function]

Describe a piecewise linear NURBS trimming curve.

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

count Specifies the number of points on the curve.

data Specifies an array containing the curve points.

stride Specifies the offset (a number of single-precision floating-point values) between points on the curve.

type Specifies the type of curve. Must be either `GLU_MAP1_TRIM_2` or `GLU_MAP1_TRIM_3`.

`gluPwlCurve` describes a piecewise linear trimming curve for a NURBS surface. A piecewise linear curve consists of a list of coordinates of points in the parameter space for the NURBS surface to be trimmed. These points are connected with line segments to form a curve. If the curve is an approximation to a curve that is not piecewise

linear, the points should be close enough in parameter space that the resulting path appears curved at the resolution used in the application.

If *type* is `GLU_MAP1_TRIM_2`, then it describes a curve in two-dimensional (u and v) parameter space. If it is `GLU_MAP1_TRIM_3`, then it describes a curve in two-dimensional homogeneous (u, v, and w) parameter space. See the `gluBeginTrim` reference page for more information about trimming curves.

`void gluQuadricCallback` *quad which CallBackFunc* [Function]

Define a callback for a quadrics object.

quad Specifies the quadrics object (created with `gluNewQuadric`).

which Specifies the callback being defined. The only valid value is `GLU_ERROR`.

CallBackFunc
Specifies the function to be called.

`gluQuadricCallback` is used to define a new callback to be used by a quadrics object. If the specified callback is already defined, then it is replaced. If *CallBackFunc* is NULL, then any existing callback is erased.

The one legal callback is `GLU_ERROR`:

`GLU_ERROR`
The function is called when an error is encountered. Its single argument is of type GLenum, and it indicates the specific error that occurred. Character strings describing these errors can be retrieved with the `gluErrorString` call.

`void gluQuadricDrawStyle` *quad draw* [Function]

Specify the draw style desired for quadrics.

quad Specifies the quadrics object (created with `gluNewQuadric`).

draw Specifies the desired draw style. Valid values are `GLU_FILL`, `GLU_LINE`, `GLU_SILHOUETTE`, and `GLU_POINT`.

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

`GLU_FILL` Quadrics are rendered with polygon primitives. The polygons are drawn in a counterclockwise fashion with respect to their normals (as defined with `gluQuadricOrientation`).

`GLU_LINE` Quadrics are rendered as a set of lines.

`GLU_SILHOUETTE`
Quadrics are rendered as a set of lines, except that edges separating coplanar faces will not be drawn.

`GLU_POINT`
Quadrics are rendered as a set of points.

`void gluQuadricNormals` *quad normal* [Function]

Specify what kind of normals are desired for quadrics.

quad Specifies the quadrics object (created with `gluNewQuadric`).

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

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

`GLU_NONE` No normals are generated.

`GLU_FLAT` One normal is generated for every facet of a quadric.

`GLU_SMOOTH`
One normal is generated for every vertex of a quadric. This is the initial value.

`void gluQuadricOrientation` *quad orientation* [Function]
Specify inside/outside orientation for quadrics.

quad Specifies the quadrics object (created with `gluNewQuadric`).

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

`gluQuadricOrientation` specifies what kind of orientation is desired for quadrics rendered with *quad*. The *orientation* values are as follows:

`GLU_OUTSIDE`
Quadrics are drawn with normals pointing outward (the initial value).

`GLU_INSIDE`
Quadrics are drawn with normals pointing inward.

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

`void gluQuadricTexture` *quad texture* [Function]
Specify if texturing is desired for quadrics.

quad Specifies the quadrics object (created with `gluNewQuadric`).

texture Specifies a flag indicating if texture coordinates should be generated.

`gluQuadricTexture` specifies if texture coordinates should be generated for quadrics rendered with *quad*. If the value of *texture* is `GLU_TRUE`, then texture coordinates are generated, and if *texture* is `GLU_FALSE`, they are not. The initial value is `GLU_FALSE`.

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

`GLint gluScaleImage` *format wIn hIn typeIn dataIn wOut hOut typeOut dataOut* [Function]
Scale an image to an arbitrary size.

format Specifies the format of the pixel data. The following symbolic values are valid: `GLU_COLOR_INDEX`, `GLU_STENCIL_INDEX`, `GLU_DEPTH_COMPONENT`, `GLU_RED`, `GLU_GREEN`, `GLU_BLUE`, `GLU_ALPHA`, `GLU_RGB`, `GLU_RGBA`, `GLU_BGR`, `GLU_BGRA`, `GLU_LUMINANCE`, and `GLU_LUMINANCE_ALPHA`.

wIn
hIn Specify in pixels the width and height, respectively, of the source image.

typeIn Specifies the data type for *dataIn*. Must be one of `GLU_UNSIGNED_BYTE`, `GLU_BYTE`, `GLU_BITMAP`, `GLU_UNSIGNED_SHORT`, `GLU_SHORT`, `GLU_UNSIGNED_INT`, `GLU_INT`, `GLU_FLOAT`, `GLU_UNSIGNED_BYTE_3_3_2`, `GLU_UNSIGNED_BYTE_2_3_3_REV`, `GLU_UNSIGNED_SHORT_5_6_5`, `GLU_UNSIGNED_SHORT_5_6_5_REV`, `GLU_UNSIGNED_SHORT_4_4_4_4`, `GLU_UNSIGNED_SHORT_4_4_4_4_REV`, `GLU_UNSIGNED_SHORT_5_5_5_1`, `GLU_UNSIGNED_SHORT_1_5_5_5_REV`, `GLU_UNSIGNED_INT_8_8_8_8`, `GLU_UNSIGNED_INT_8_8_8_8_REV`, `GLU_UNSIGNED_INT_10_10_10_2`, or `GLU_UNSIGNED_INT_2_10_10_10_REV`.

dataIn Specifies a pointer to the source image.

wOut
hOut Specify the width and height, respectively, in pixels of the destination image.

typeOut Specifies the data type for *dataOut*. Must be one of `GLU_UNSIGNED_BYTE`, `GLU_BYTE`, `GLU_BITMAP`, `GLU_UNSIGNED_SHORT`, `GLU_SHORT`, `GLU_UNSIGNED_INT`, `GLU_INT`, `GLU_FLOAT`, `GLU_UNSIGNED_BYTE_3_3_2`, `GLU_UNSIGNED_BYTE_2_3_3_REV`, `GLU_UNSIGNED_SHORT_5_6_5`, `GLU_UNSIGNED_SHORT_5_6_5_REV`, `GLU_UNSIGNED_SHORT_4_4_4_4`, `GLU_UNSIGNED_SHORT_4_4_4_4_REV`, `GLU_UNSIGNED_SHORT_5_5_5_1`, `GLU_UNSIGNED_SHORT_1_5_5_5_REV`, `GLU_UNSIGNED_INT_8_8_8_8`, `GLU_UNSIGNED_INT_8_8_8_8_REV`, `GLU_UNSIGNED_INT_10_10_10_2`, or `GLU_UNSIGNED_INT_2_10_10_10_REV`.

dataOut Specifies a pointer to the destination image.

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

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

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

See the `glReadPixels` reference page for a description of the acceptable values for the *format*, *typeIn*, and *typeOut* parameters.

`GLU_INVALID_VALUE` is returned if *wIn*, *hIn*, *wOut*, or *hOut* is negative.

`GLU_INVALID_ENUM` is returned if *format*, *typeIn*, or *typeOut* is not legal.

`GLU_INVALID_OPERATION` is returned if *typeIn* or *typeOut* is `GLU_UNSIGNED_BYTE_3_3_2` or `GLU_UNSIGNED_BYTE_2_3_3_REV` and *format* is not `GLU_RGB`.

`GLU_INVALID_OPERATION` is returned if *typeIn* or *typeOut* is `GLU_UNSIGNED_SHORT_5_6_5` or `GLU_UNSIGNED_SHORT_5_6_5_REV` and *format* is not `GLU_RGB`.

`GLU_INVALID_OPERATION` is returned if *typeIn* or *typeOut* is `GLU_UNSIGNED_SHORT_4_4_4_4` or `GLU_UNSIGNED_SHORT_4_4_4_4_REV` and *format* is neither `GLU_RGBA` nor `GLU_BGRA`.

`GLU_INVALID_OPERATION` is returned if *typeIn* or *typeOut* is `GLU_UNSIGNED_SHORT_5_5_5_1` or `GLU_UNSIGNED_SHORT_1_5_5_5_REV` and *format* is neither `GLU_RGBA` nor `GLU_BGRA`.

`GLU_INVALID_OPERATION` is returned if *typeIn* or *typeOut* is `GLU_UNSIGNED_INT_8_8_8_8` or `GLU_UNSIGNED_INT_8_8_8_8_REV` and *format* is neither `GLU_RGBA` nor `GLU_BGRA`.

`GLU_INVALID_OPERATION` is returned if *typeIn* or *typeOut* is `GLU_UNSIGNED_INT_10_10_10_2` or `GLU_UNSIGNED_INT_2_10_10_10_REV` and *format* is neither `GLU_RGBA` nor `GLU_BGRA`.

`void gluSphere` *quad radius slices stacks* [Function]

Draw a sphere.

quad Specifies the quadrics object (created with `gluNewQuadric`).

radius Specifies the radius of the sphere.

slices Specifies the number of subdivisions around the z axis (similar to lines of longitude).

stacks Specifies the number of subdivisions along the z axis (similar to lines of latitude).

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

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

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

`void gluTessBeginContour` *tess* [Function]
`void gluTessEndContour` *tess* [Function]

Delimit a contour description.

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

`gluTessBeginContour` and `gluTessEndContour` delimit the definition of a polygon contour. Within each `gluTessBeginContour`/`gluTessEndContour` pair, there can be zero or more calls to `gluTessVertex`. The vertices specify a closed contour (the last vertex of each contour is automatically linked to the first). See the `gluTessVertex` reference page for more details. `gluTessBeginContour` can only be called between `gluTessBeginPolygon` and `gluTessEndPolygon`.

`void gluTessBeginPolygon` *tess data* [Function]

Delimit a polygon description.

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

data Specifies a pointer to user polygon data.

`gluTessBeginPolygon` and `gluTessEndPolygon` delimit the definition of a convex, concave or self-intersecting polygon. Within each `gluTessBeginPolygon`/`gluTessEndPolygon` pair, there must be one or more calls to `gluTessBeginContour`/`gluTessEndContour`. Within each contour, there are zero or more calls to `gluTessVertex`. The vertices specify a closed contour (the last vertex of each contour is automatically linked to the first). See the `gluTessVertex`, `gluTessBeginContour`, and `gluTessEndContour` reference pages for more details.

data is a pointer to a user-defined data structure. If the appropriate callback(s) are specified (see `gluTessCallback`), then this pointer is returned to the callback function(s). Thus, it is a convenient way to store per-polygon information.

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

`void gluTessCallback` *tess which CallBackFunc* [Function]

Define a callback for a tessellation object.

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

which Specifies the callback being defined. The following values are valid: `GLU_TESS_BEGIN`, `GLU_TESS_BEGIN_DATA`, `GLU_TESS_EDGE_FLAG`, `GLU_TESS_EDGE_FLAG_DATA`, `GLU_TESS_VERTEX`, `GLU_TESS_VERTEX_DATA`, `GLU_TESS_END`, `GLU_TESS_END_DATA`, `GLU_TESS_COMBINE`, `GLU_TESS_COMBINE_DATA`, `GLU_TESS_ERROR`, and `GLU_TESS_ERROR_DATA`.

CallBackFunc
Specifies the function to be called.

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

These callbacks are used by the tessellation object to describe how a polygon specified by the user is broken into triangles. Note that there are two versions of each callback: one with user-specified polygon data and one without. If both versions of a particular callback are specified, then the callback with user-specified polygon data will be used. Note that the *polygon_data* parameter used by some of the functions is a copy of the pointer that was specified when `gluTessBeginPolygon` was called. The legal callbacks are as follows:

`GLU_TESS_BEGIN`
The begin callback is invoked like `glBegin` to indicate the start of a (triangle) primitive. The function takes a single argument of type GLenum. If the `GLU_TESS_BOUNDARY_ONLY` property is set to `GLU_FALSE`, then the

argument is set to either GLU_TRIANGLE_FAN, GLU_TRIANGLE_STRIP, or GLU_TRIANGLES. If the GLU_TESS_BOUNDARY_ONLY property is set to GLU_TRUE, then the argument will be set to GLU_LINE_LOOP. The function prototype for this callback is:

GLU_TESS_BEGIN_DATA

The same as the GLU_TESS_BEGIN callback except that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when gluTessBeginPolygon was called. The function prototype for this callback is:

GLU_TESS_EDGE_FLAG

The edge flag callback is similar to glEdgeFlag. The function takes a single boolean flag that indicates which edges lie on the polygon boundary. If the flag is GLU_TRUE, then each vertex that follows begins an edge that lies on the polygon boundary, that is, an edge that separates an interior region from an exterior one. If the flag is GLU_FALSE, then each vertex that follows begins an edge that lies in the polygon interior. The edge flag callback (if defined) is invoked before the first vertex callback.

Since triangle fans and triangle strips do not support edge flags, the begin callback is not called with GLU_TRIANGLE_FAN or GLU_TRIANGLE_STRIP if a non-NULL edge flag callback is provided. (If the callback is initialized to NULL, there is no impact on performance). Instead, the fans and strips are converted to independent triangles. The function prototype for this callback is:

GLU_TESS_EDGE_FLAG_DATA

The same as the GLU_TESS_EDGE_FLAG callback except that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when gluTessBeginPolygon was called. The function prototype for this callback is:

GLU_TESS_VERTEX

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

GLU_TESS_VERTEX_DATA

The same as the GLU_TESS_VERTEX callback except that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when gluTessBeginPolygon was called. The function prototype for this callback is:

GLU_TESS_END

The end callback serves the same purpose as glEnd. It indicates the end of a primitive and it takes no arguments. The function prototype for this callback is:

`GLU_TESS_END_DATA`

The same as the `GLU_TESS_END` callback except that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when `gluTessBeginPolygon` was called. The function prototype for this callback is:

`GLU_TESS_COMBINE`

The combine callback is called to create a new vertex when the tessellation detects an intersection or wishes to merge features. The function takes four arguments: an array of three elements each of type GLdouble, an array of four pointers, an array of four elements each of type GLfloat, and a pointer to a pointer. The prototype is:

The vertex is defined as a linear combination of up to four existing vertices, stored in *vertex_data*. The coefficients of the linear combination are given by *weight*; these weights always add up to 1. All vertex pointers are valid even when some of the weights are 0. *coords* gives the location of the new vertex.

The user must allocate another vertex, interpolate parameters using *vertex_data* and *weight*, and return the new vertex pointer in *outData*. This handle is supplied during rendering callbacks. The user is responsible for freeing the memory some time after `gluTessEndPolygon` is called.

For example, if the polygon lies in an arbitrary plane in 3-space, and a color is associated with each vertex, the `GLU_TESS_COMBINE` callback might look like this:

If the tessellation detects an intersection, then the `GLU_TESS_COMBINE` or `GLU_TESS_COMBINE_DATA` callback (see below) must be defined, and it must write a non-NULL pointer into *dataOut*. Otherwise the `GLU_TESS_NEED_COMBINE_CALLBACK` error occurs, and no output is generated.

`GLU_TESS_COMBINE_DATA`

The same as the `GLU_TESS_COMBINE` callback except that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when `gluTessBeginPolygon` was called. The function prototype for this callback is:

`GLU_TESS_ERROR`

The error callback is called when an error is encountered. The one argument is of type GLenum; it indicates the specific error that occurred and will be set to one of `GLU_TESS_MISSING_BEGIN_POLYGON`, `GLU_TESS_MISSING_END_POLYGON`, `GLU_TESS_MISSING_BEGIN_CONTOUR`, `GLU_TESS_MISSING_END_CONTOUR`, `GLU_TESS_COORD_TOO_LARGE`, `GLU_TESS_NEED_COMBINE_CALLBACK`, or `GLU_OUT_OF_MEMORY`. Character strings describing these errors can be retrieved with the `gluErrorString` call. The function prototype for this callback is:

The GLU library will recover from the first four errors by inserting the missing call(s). `GLU_TESS_COORD_TOO_LARGE` indicates that some vertex coordinate exceeded the predefined constant `GLU_TESS_MAX_COORD` in absolute value, and that the value has been clamped. (Coordinate values

must be small enough so that two can be multiplied together without overflow.) GLU_TESS_NEED_COMBINE_CALLBACK indicates that the tessellation detected an intersection between two edges in the input data, and the GLU_TESS_COMBINE or GLU_TESS_COMBINE_DATA callback was not provided. No output is generated. GLU_OUT_OF_MEMORY indicates that there is not enough memory so no output is generated.

GLU_TESS_ERROR_DATA

The same as the GLU_TESS_ERROR callback except that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when gluTessBeginPolygon was called. The function prototype for this callback is:

```
void begin( GLenum type );

void beginData( GLenum type, void *polygon_data );

void edgeFlag( GLboolean flag );

void edgeFlagData( GLboolean flag, void *polygon_data );

void vertex( void *vertex_data );

void vertexData( void *vertex_data, void *polygon_data );

void end( void );

void endData( void *polygon_data );

void combine( GLdouble coords[3], void *vertex_data[4],
              GLfloat weight[4], void **outData );

void myCombine( GLdouble coords[3], VERTEX *d[4],
                GLfloat w[4], VERTEX **dataOut )
{
   VERTEX *new = new_vertex();

   new->x = coords[0];
   new->y = coords[1];
   new->z = coords[2];
   new->r = w[0]*d[0]->r + w[1]*d[1]->r + w[2]*d[2]->r + w[3]*d[3]->r;
   new->g = w[0]*d[0]->g + w[1]*d[1]->g + w[2]*d[2]->g + w[3]*d[3]->g;
   new->b = w[0]*d[0]->b + w[1]*d[1]->b + w[2]*d[2]->b + w[3]*d[3]->b;
   new->a = w[0]*d[0]->a + w[1]*d[1]->a + w[2]*d[2]->a + w[3]*d[3]->a;
   *dataOut = new;
```

```
}

void combineData( GLdouble coords[3], void *vertex_data[4],
                  GLfloat weight[4], void **outData,
                  void *polygon_data );

void error( GLenum errno );

void errorData( GLenum errno, void *polygon_data );
```

`void gluTessEndPolygon` *tess* [Function]

Delimit a polygon description.

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

`gluTessBeginPolygon` and `gluTessEndPolygon` delimit the definition of a convex, concave, or self-intersecting polygon. Within each `gluTessBeginPolygon`/`gluTessEndPolygon` pair, there must be one or more calls to `gluTessBeginContour`/`gluTessEndContour`. Within each contour, there are zero or more calls to `gluTessVertex`. The vertices specify a closed contour (the last vertex of each contour is automatically linked to the first). See the `gluTessVertex`, `gluTessBeginContour`, and `gluTessEndContour` reference pages for more details.

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

`void gluTessNormal` *tess valueX valueY valueZ* [Function]

Specify a normal for a polygon.

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

valueX Specifies the first component of the normal.

valueY Specifies the second component of the normal.

valueZ Specifies the third component of the normal.

`gluTessNormal` describes a normal for a polygon that the program is defining. All input data will be projected onto a plane perpendicular to one of the three coordinate axes before tessellation and all output triangles will be oriented CCW with respect to the normal (CW orientation can be obtained by reversing the sign of the supplied normal). For example, if you know that all polygons lie in the x-y plane, call `gluTessNormal`(tess, 0.0, 0.0, 1.0) before rendering any polygons.

If the supplied normal is (0.0, 0.0, 0.0) (the initial value), the normal is determined as follows. The direction of the normal, up to its sign, is found by fitting a plane to the vertices, without regard to how the vertices are connected. It is expected that the input data lies approximately in the plane; otherwise, projection perpendicular to one of the three coordinate axes may substantially change the geometry. The sign of the normal is chosen so that the sum of the signed areas of all input contours is nonnegative (where a CCW contour has positive area).

The supplied normal persists until it is changed by another call to `gluTessNormal`.

`void gluTessProperty` *tess which data* [Function]

Set a tessellation object property.

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

which Specifies the property to be set. Valid values are `GLU_TESS_WINDING_RULE`, `GLU_TESS_BOUNDARY_ONLY`, and `GLU_TESS_TOLERANCE`.

data Specifies the value of the indicated property.

`gluTessProperty` is used to control properties stored in a tessellation object. These properties affect the way that the polygons are interpreted and rendered. The legal values for *which* are as follows:

`GLU_TESS_WINDING_RULE`

Determines which parts of the polygon are on the "interior". *data* may be set to one of `GLU_TESS_WINDING_ODD`, `GLU_TESS_WINDING_NONZERO`, `GLU_TESS_WINDING_POSITIVE`, `GLU_TESS_WINDING_NEGATIVE`, or `GLU_TESS_WINDING_ABS_GEQ_TWO`.

To understand how the winding rule works, consider that the input contours partition the plane into regions. The winding rule determines which of these regions are inside the polygon.

For a single contour C, the winding number of a point x is simply the signed number of revolutions we make around x as we travel once around C (where CCW is positive). When there are several contours, the individual winding numbers are summed. This procedure associates a signed integer value with each point x in the plane. Note that the winding number is the same for all points in a single region.

The winding rule classifies a region as "inside" if its winding number belongs to the chosen category (odd, nonzero, positive, negative, or absolute value of at least two). The previous GLU tessellator (prior to GLU 1.2) used the "odd" rule. The "nonzero" rule is another common way to define the interior. The other three rules are useful for polygon CSG operations.

`GLU_TESS_BOUNDARY_ONLY`

Is a boolean value ("value" should be set to GL_TRUE or GL_FALSE). When set to GL_TRUE, a set of closed contours separating the polygon interior and exterior are returned instead of a tessellation. Exterior contours are oriented CCW with respect to the normal; interior contours are oriented CW. The `GLU_TESS_BEGIN` and `GLU_TESS_BEGIN_DATA` callbacks use the type GL_LINE_LOOP for each contour.

`GLU_TESS_TOLERANCE`

Specifies a tolerance for merging features to reduce the size of the output. For example, two vertices that are very close to each other might be replaced by a single vertex. The tolerance is multiplied by the largest coordinate magnitude of any input vertex; this specifies the maximum distance that any feature can move as the result of a single merge operation. If a single feature takes part in several merge operations, the total distance moved could be larger.

Feature merging is completely optional; the tolerance is only a hint. The implementation is free to merge in some cases and not in others, or to never merge features at all. The initial tolerance is 0.

The current implementation merges vertices only if they are exactly coincident, regardless of the current tolerance. A vertex is spliced into an edge only if the implementation is unable to distinguish which side of the edge the vertex lies on. Two edges are merged only when both endpoints are identical.

`void gluTessVertex` *tess location data* [Function]

Specify a vertex on a polygon.

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

location Specifies the location of the vertex.

data Specifies an opaque pointer passed back to the program with the vertex callback (as specified by `gluTessCallback`).

`gluTessVertex` describes a vertex on a polygon that the program defines. Successive `gluTessVertex` calls describe a closed contour. For example, to describe a quadrilateral, `gluTessVertex` should be called four times. `gluTessVertex` can only be called between `gluTessBeginContour` and `gluTessEndContour`.

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

`GLint gluUnProject4` *winX winY winZ clipW model proj view nearVal farVal objX objY objZ objW* [Function]

Map window and clip coordinates to object coordinates.

winX
winY
winZ Specify the window coordinates to be mapped.

clipW Specify the clip w coordinate to be mapped.

model Specifies the modelview matrix (as from a `glGetDoublev` call).

proj Specifies the projection matrix (as from a `glGetDoublev` call).

view Specifies the viewport (as from a `glGetIntegerv` call).

nearVal
farVal Specifies the near and far planes (as from a `glGetDoublev` call).

objX
objY
objZ
objW Returns the computed object coordinates.

`gluUnProject4` maps the specified window coordinatesi: *winX*, *winY*, and *winZ* and its clip w coordinate *clipW* into object coordinates (*objX*,*objY objZ objW*) using

model, *proj*, and *view*. *clipW* can be other than 1 as for vertices in `glFeedbackBuffer` when data type `GLU_4D_COLOR_TEXTURE` is returned. This also handles the case where the *nearVal* and *farVal* planes are different from the default, 0 and 1, respectively. A return value of `GLU_TRUE` indicates success; a return value of `GLU_FALSE` indicates failure.

To compute the coordinates (*objX*,*objY objZ objW*), `gluUnProject4` multiplies the normalized device coordinates by the inverse of *model* * *proj* as follows:

((*objX*), (*objY*), (*objZ*), (*objW*),)=*INV*(*PM*,)((2(*winX*-*view*[0,],),/*view*[2,],-1), (2(*winY*-*view*[1,],),/*view*[3,],-1), (2(*winZ*-*nearVal*,),/(*farVal*-*nearVal*,),-1), (*clipW*),)

INV denotes matrix inversion.

`gluUnProject4` is equivalent to `gluUnProject` when *clipW* is 1, *nearVal* is 0, and *farVal* is 1.

`GLint gluUnProject` *winX winY winZ model proj view objX objY objZ* [Function]

Map window coordinates to object coordinates.

winX
winY
winZ Specify the window coordinates to be mapped.

model Specifies the modelview matrix (as from a `glGetDoublev` call).

proj Specifies the projection matrix (as from a `glGetDoublev` call).

view Specifies the viewport (as from a `glGetIntegerv` call).

objX
objY
objZ Returns the computed object coordinates.

`gluUnProject` maps the specified window coordinates into object coordinates using *model*, *proj*, and *view*. The result is stored in *objX*, *objY*, and *objZ*. A return value of `GLU_TRUE` indicates success; a return value of `GLU_FALSE` indicates failure.

To compute the coordinates (*objX*,*objY objZ*), `gluUnProject` multiplies the normalized device coordinates by the inverse of *model* * *proj* as follows:

((*objX*), (*objY*), (*objZ*), (*W*),)=*INV*(*PM*,)((2(*winX*-*view*[0,],),/*view*[2,],-1), (2(*winY*-*view*[1,],),/*view*[3,],-1), (2(*winZ*,)-1), (1),)*INV* denotes matrix inversion. W is an unused variable, included for consistent matrix notation.

5 GLX

5.1 GLX API

Import the GLX module to have access to these procedures:

```
(use-modules (glx))
```

The GLX specification is available at http://www.opengl.org/registry/doc/glx1.3.pdf.

5.2 GLX Enumerations

The functions from this section may be had by loading the module:

```
(use-modules (glx enums)
```

`glx-string-name` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`vendor`, `version`, `extensions`.

`glx-error-code` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`bad-screen`, `bad-attribute`, `no-extension`, `bad-visual`, `bad-context`, `bad-value`, `bad-enum`, `bad-hyperpipe-config-sgix`, `bad-hyperpipe-sgix`.

`glx-drawable-type-mask` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`window-bit`, `pixmap-bit`, `pbuffer-bit`, `window-bit-sgix`, `pixmap-bit-sgix`, `pbuffer-bit-sgix`.

`glx-render-type-mask` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`rgba-bit`, `color-index-bit`, `rgba-bit-sgix`, `color-index-bit-sgix`, `rgba-float-bit-arb`, `rgba-unsigned-float-bit-ext`.

`glx-sync-type` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`sync-frame-sgix`, `sync-swap-sgix`.

`glx-event-mask` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`pbuffer-clobber-mask`, `buffer-clobber-mask-sgix`, `buffer-swap-complete-intel-mask`.

`glx-pbuffer-clobber-mask` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`front-left-buffer-bit`, `front-right-buffer-bit`, `back-left-buffer-bit`, `back-right-buffer-bit`, `aux-buffers-bit`, `depth-buffer-bit`, `stencil-buffer-bit`, `accum-buffer-bit`, `front-left-buffer-bit-sgix`, `front-right-buffer-bit-sgix`, `back-left-buffer-bit-sgix`, `back-right-buffer-bit-sgix`, `aux-buffers-bit-sgix`, `depth-buffer-bit-sgix`, `stencil-buffer-bit-sgix`, `accum-buffer-bit-sgix`, `sample-buffers-bit-sgix`.

`glx-hyperpipe-type-mask` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`hyperpipe-display-pipe-sgix`, `hyperpipe-render-pipe-sgix`.

`glx-hyperpipe-attrib` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`pipe-rect-sgix`, `pipe-rect-limits-sgix`, `hyperpipe-stereo-sgix`, `hyperpipe-pixel-average-sgix`.

`glx-hyperpipe-misc` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`hyperpipe-pipe-name-length-sgix`.

`glx-bind-to-texture-target-mask` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`texture-1d-bit-ext`, `texture-2d-bit-ext`, `texture-rectangle-bit-ext`.

`glx-context-flags` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`context-debug-bit-arb`, `context-forward-compatible-bit-arb`, `context-robust-access-bit-arb`, `context-reset-isolation-bit-arb`.

`glx-context-profile-mask` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`context-core-profile-bit-arb`, `context-compatibility-profile-bit-arb`, `context-es-profile-bit-ext`, `context-es2-profile-bit-ext`.

`glx-attribute` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`use-gl`, `buffer-size`, `level`, `rgba`, `doublebuffer`, `stereo`, `aux-buffers`, `red-size`, `green-size`, `blue-size`, `alpha-size`, `depth-size`, `stencil-size`, `accum-red-size`, `accum-green-size`, `accum-blue-size`, `accum-alpha-size`, `config-caveat`, `x-visual-type`, `transparent-type`, `transparent-index-value`, `transparent-red-value`, `transparent-green-value`, `transparent-blue-value`, `transparent-alpha-value`, `dont-care`, `none`, `slow-config`, `true-color`, `direct-color`, `pseudo-color`, `static-color`, `gray-scale`, `static-gray`, `transparent-rgb`, `transparent-index`, `visual-id`, `screen`, `non-conformant-config`, `drawable-type`, `render-type`, `x-renderable`, `fbconfig-id`, `rgba-type`, `color-index-type`, `max-pbuffer-width`, `max-pbuffer-height`, `max-pbuffer-pixels`, `preserved-contents`, `largest-pbuffer`, `width`, `height`, `event-mask`, `damaged`, `saved`, `window`, `pbuffer`, `pbuffer-height`, `pbuffer-width`, `visual-caveat-ext`, `x-visual-type-ext`, `transparent-type-ext`, `transparent-index-value-ext`, `transparent-red-value-ext`, `transparent-green-value-ext`, `transparent-blue-value-ext`, `transparent-alpha-value-ext`, `none-ext`, `slow-visual-ext`, `true-color-ext`, `direct-color-ext`, `pseudo-color-ext`, `static-color-ext`, `gray-scale-ext`, `static-gray-ext`, `transparent-rgb-ext`, `transparent-index-ext`, `share-context-ext`, `visual-id-ext`, `screen-ext`, `non-conformant-visual-ext`, `drawable-type-sgix`, `render-type-sgix`, `x-renderable-sgix`, `fbconfig-id-sgix`, `rgba-type-sgix`, `color-index-type-sgix`, `max-pbuffer-width-sgix`, `max-pbuffer-height-sgix`, `max-pbuffer-pixels-sgix`, `optimal-pbuffer-width-sgix`, `optimal-pbuffer-height-sgix`, `preserved-contents-sgix`, `largest-pbuffer-sgix`, `width-sgix`, `height-sgix`, `event-mask-sgix`, `damaged-sgix`, `saved-sgix`, `window-sgix`, `pbuffer-sgix`, `digital-media-pbuffer-sgix`, `blended-rgba-sgis`, `multisample-sub-rect-width-sgis`, `multisample-sub-rect-height-sgis`, `visual-select-group-sgix`, `hyperpipe-id-sgix`, `sample-buffers-sgis`, `samples-sgis`, `sample-buffers-arb`, `samples-arb`, `sample-buffers`, `samples`, `coverage-samples-nv`, `context-major-version-arb`, `context-minor-version-arb`, `context-flags-arb`, `context-allow-buffer-byte-order-mismatch-arb`, `float-components-nv`, `rgba-unsigned-float-type-ext`, `framebuffer-srgb-capable-arb`, `framebuffer-srgb-capable-ext`, `color-samples-nv`, `rgba-float-type-arb`, `video-out-color-nv`, `video-out-alpha-nv`, `video-out-depth-nv`, `video-out-color-and-alpha-nv`, `video-out-color-and-depth-nv`, `video-out-frame-nv`, `video-out-field-1-nv`, `video-out-field-2-nv`, `video-out-stacked-fields-

`1-2-nv`, `video-out-stacked-fields-2-1-nv`, `device-id-nv`, `unique-id-nv`, `num-video-capture-slots-nv`, `bind-to-texture-rgb-ext`, `bind-to-texture-rgba-ext`, `bind-to-mipmap-texture-ext`, `bind-to-texture-targets-ext`, `y-inverted-ext`, `texture-format-ext`, `texture-target-ext`, `mipmap-texture-ext`, `texture-format-none-ext`, `texture-format-rgb-ext`, `texture-format-rgba-ext`, `texture-1d-ext`, `texture-2d-ext`, `texture-rectangle-ext`, `front-left-ext`, `front-right-ext`, `back-left-ext`, `back-right-ext`, `front-ext`, `back-ext`, `aux0-ext`, `aux1-ext`, `aux2-ext`, `aux3-ext`, `aux4-ext`, `aux5-ext`, `aux6-ext`, `aux7-ext`, `aux8-ext`, `aux9-ext`.

`nv-present-video` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`num-video-slots-nv`.

`ext-swap-control` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`swap-interval-ext`, `max-swap-interval-ext`.

`ext-swap-control-tear` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`late-swaps-tear-ext`.

`ext-buffer-age` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`back-buffer-age-ext`.

`glx-amd-gpu-association` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`gpu-vendor-amd`, `gpu-renderer-string-amd`, `gpu-opengl-version-string-amd`, `gpu-fastest-target-gpus-amd`, `gpu-ram-amd`, `gpu-clock-amd`, `gpu-num-pipes-amd`, `gpu-num-simd-amd`, `gpu-num-rb-amd`, `gpu-num-spi-amd`.

`glx-arb-create-context-robustness` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`lose-context-on-reset-arb`, `context-reset-notification-strategy-arb`, `no-reset-notification-arb`.

`arb-create-context-profile` *enum* [Macro]

Enumerated value. The symbolic *enum* argument is replaced with its corresponding numeric value at compile-time. The symbolic arguments known to this enumerated value form are:

`context-profile-mask-arb`.

5.3 Low-Level GLX

The functions from this section may be had by loading the module:

```
(use-modules (glx low-level)
```

This section of the manual was derived from the upstream OpenGL documentation. Each function's documentation has its own copyright statement; for full details, see the upstream documentation. The copyright notices and licenses present in this section are as follows.

`GLXFBConfig-*` `glXChooseFBConfig` *dpy screen attrib_list nelements* [Function]

Return a list of GLX frame buffer configurations that match the specified attributes.

dpy Specifies the connection to the X server.

screen Specifies the screen number.

attrib_list Specifies a list of attribute/value pairs. The last attribute must be `None`.

nelements Returns the number of elements in the list returned by `glXChooseFBConfig`.

`glXChooseFBConfig` returns GLX frame buffer configurations that match the attributes specified in *attrib_list*, or `NULL` if no matches are found. If *attrib_list* is `NULL`, then `glXChooseFBConfig` returns an array of GLX frame buffer configurations that are available on the specified screen. If an error occurs, no frame buffer configurations exist on the specified screen, or if no frame buffer configurations match the specified attributes, then `NULL` is returned. Use `XFree` to free the memory returned by `glXChooseFBConfig`.

All attributes in *attrib_list*, including boolean attributes, are immediately followed by the corresponding desired value. The list is terminated with `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 `GLX_STEREO` is not specified, then it is assumed to be `False`. For some attributes, the default is `GLX_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 `GLX_LEVEL`, must match the specified value exactly; others, such as, `GLX_RED_SIZE` must meet or exceed the specified minimum values. If more than one GLX 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 GLX visual attributes are as follows:

GLX_FBCONFIG_ID
: Must be followed by a valid XID that indicates the desired GLX frame buffer configuration. When a GLX_FBCONFIG_ID is specified, all attributes are ignored. The default value is GLX_DONT_CARE.

GLX_BUFFER_SIZE
: Must be followed by a nonnegative integer that indicates the desired color index buffer size. The smallest index buffer of at least the specified size is preferred. This attribute is ignored if GLX_COLOR_INDEX_BIT is not set in GLX_RENDER_TYPE. The default value is 0.

GLX_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.

GLX_DOUBLEBUFFER
: Must be followed by True or False. If True is specified, then only double-buffered frame buffer configurations are considered; if False is specified, then only single-buffered frame buffer configurations are considered. The default value is GLX_DONT_CARE.

GLX_STEREO
: Must be followed by True or False. If True is specified, then only stereo frame buffer configurations are considered; if False is specified, then only monoscopic frame buffer configurations are considered. The default value is False.

GLX_AUX_BUFFERS
: Must be followed by a nonnegative integer that indicates the desired number of auxiliary buffers. Configurations with the smallest number of auxiliary buffers that meet or exceed the specified number are preferred. The default value is 0.

GLX_RED_SIZE, GLX_GREEN_SIZE, GLX_BLUE_SIZE, GLX_ALPHA_SIZE
: Each attribute, if present, must be followed by a nonnegative minimum size specification or GLX_DONT_CARE. The largest available total RGBA color buffer size (sum of GLX_RED_SIZE, GLX_GREEN_SIZE, GLX_BLUE_SIZE, and GLX_ALPHA_SIZE) of at least the minimum size specified for each color component is preferred. If the requested number of bits for a color component is 0 or GLX_DONT_CARE, it is not considered. The default value for each color component is 0.

GLX_DEPTH_SIZE
: Must be followed by a nonnegative minimum size specification. If this value is zero, frame buffer configurations with no depth buffer are preferred. Otherwise, the largest available depth buffer of at least the minimum size is preferred. The default value is 0.

`GLX_STENCIL_SIZE`
: Must be followed by a nonnegative integer that indicates the desired number of stencil bitplanes. The smallest stencil buffer of at least the specified size is preferred. If the desired value is zero, frame buffer configurations with no stencil buffer are preferred. The default value is 0.

`GLX_ACCUM_RED_SIZE`
: Must be followed by a nonnegative minimum size specification. If this value is zero, frame buffer configurations with no red accumulation buffer are preferred. Otherwise, the largest possible red accumulation buffer of at least the minimum size is preferred. The default value is 0.

`GLX_ACCUM_GREEN_SIZE`
: Must be followed by a nonnegative minimum size specification. If this value is zero, frame buffer configurations with no green accumulation buffer are preferred. Otherwise, the largest possible green accumulation buffer of at least the minimum size is preferred. The default value is 0.

`GLX_ACCUM_BLUE_SIZE`
: Must be followed by a nonnegative minimum size specification. If this value is zero, frame buffer configurations with no blue accumulation buffer are preferred. Otherwise, the largest possible blue accumulation buffer of at least the minimum size is preferred. The default value is 0.

`GLX_ACCUM_ALPHA_SIZE`
: Must be followed by a nonnegative minimum size specification. If this value is zero, frame buffer configurations with no alpha accumulation buffer are preferred. Otherwise, the largest possible alpha accumulation buffer of at least the minimum size is preferred. The default value is 0.

`GLX_RENDER_TYPE`
: Must be followed by a mask indicating which OpenGL rendering modes the frame buffer configuration must support. Valid bits are `GLX_RGBA_BIT` and `GLX_COLOR_INDEX_BIT`. If the mask is set to `GLX_RGBA_BIT | GLX_COLOR_INDEX_BIT`, then only frame buffer configurations that can be bound to both RGBA contexts and color index contexts will be considered. The default value is `GLX_RGBA_BIT`.

`GLX_DRAWABLE_TYPE`
: Must be followed by a mask indicating which GLX drawable types the frame buffer configuration must support. Valid bits are `GLX_WINDOW_BIT`, `GLX_PIXMAP_BIT`, and `GLX_PBUFFER_BIT`. For example, if mask is set to `GLX_WINDOW_BIT | GLX_PIXMAP_BIT`, only frame buffer configurations that support both windows and GLX pixmaps will be considered. The default value is `GLX_WINDOW_BIT`.

`GLX_X_RENDERABLE`
: Must be followed by `True` or `False`. If `True` is specified, then only frame buffer configurations that have associated X visuals (and can be used to render to Windows and/or GLX pixmaps) will be considered. The default value is `GLX_DONT_CARE`.

GLX_X_VISUAL_TYPE
: Must be followed by one of GLX_TRUE_COLOR, GLX_DIRECT_COLOR, GLX_PSEUDO_COLOR, GLX_STATIC_COLOR, GLX_GRAY_SCALE, or GLX_STATIC_GRAY, indicating the desired X visual type. Not all frame buffer configurations have an associated X visual. If GLX_DRAWABLE_TYPE is specified in *attrib_list* and the mask that follows does not have GLX_WINDOW_BIT set, then this value is ignored. It is also ignored if GLX_X_RENDERABLE is specified as False. RGBA rendering may be supported for visuals of type GLX_TRUE_COLOR, GLX_DIRECT_COLOR, GLX_PSEUDO_COLOR, or GLX_STATIC_COLOR, but color index rendering is only supported for visuals of type GLX_PSEUDO_COLOR or GLX_STATIC_COLOR (i.e., single-channel visuals). The tokens GLX_GRAY_SCALE and GLX_STATIC_GRAY will not match current OpenGL enabled visuals, but are included for future use. The default value for GLX_X_VISUAL_TYPE is GLX_DONT_CARE.

GLX_CONFIG_CAVEAT
: Must be followed by one of GLX_NONE, GLX_SLOW_CONFIG, GLX_NON_CONFORMANT_CONFIG. If GLX_NONE is specified, then only frame buffer configurations with no caveats will be considered; if GLX_SLOW_CONFIG is specified, then only slow frame buffer configurations will be considered; if GLX_NON_CONFORMANT_CONFIG is specified, then only nonconformant frame buffer configurations will be considered. The default value is GLX_DONT_CARE.

GLX_TRANSPARENT_TYPE
: Must be followed by one of GLX_NONE, GLX_TRANSPARENT_RGB, GLX_TRANSPARENT_INDEX. If GLX_NONE is specified, then only opaque frame buffer configurations will be considered; if GLX_TRANSPARENT_RGB is specified, then only transparent frame buffer configurations that support RGBA rendering will be considered; if GLX_TRANSPARENT_INDEX is specified, then only transparent frame buffer configurations that support color index rendering will be considered. The default value is GLX_NONE.

GLX_TRANSPARENT_INDEX_VALUE
: Must be followed by an integer value indicating the transparent index value; the value must be between 0 and the maximum frame buffer value for indices. Only frame buffer configurations that use the specified transparent index value will be considered. The default value is GLX_DONT_CARE. This attribute is ignored unless GLX_TRANSPARENT_TYPE is included in *attrib_list* and specified as GLX_TRANSPARENT_INDEX.

GLX_TRANSPARENT_RED_VALUE
: Must be followed by an integer value indicating the transparent red value; the value must be between 0 and the maximum frame buffer value for red. Only frame buffer configurations that use the specified transparent red value will be considered. The default value is GLX_DONT_CARE. This attribute is ignored unless GLX_TRANSPARENT_TYPE is included in *attrib_list* and specified as GLX_TRANSPARENT_RGB.

GLX_TRANSPARENT_GREEN_VALUE
: Must be followed by an integer value indicating the transparent green value; the value must be between 0 and the maximum frame buffer value for green. Only frame buffer configurations that use the specified transparent green value will be considered. The default value is GLX_DONT_CARE. This attribute is ignored unless GLX_TRANSPARENT_TYPE is included in *attrib_list* and specified as GLX_TRANSPARENT_RGB.

GLX_TRANSPARENT_BLUE_VALUE
: Must be followed by an integer value indicating the transparent blue value; the value must be between 0 and the maximum frame buffer value for blue. Only frame buffer configurations that use the specified transparent blue value will be considered. The default value is GLX_DONT_CARE. This attribute is ignored unless GLX_TRANSPARENT_TYPE is included in *attrib_list* and specified as GLX_TRANSPARENT_RGB.

GLX_TRANSPARENT_ALPHA_VALUE
: Must be followed by an integer value indicating the transparent alpha value; the value must be between 0 and the maximum frame buffer value for alpha. Only frame buffer configurations that use the specified transparent alpha value will be considered. The default value is GLX_DONT_CARE.

When more than one GLX 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 GLX_CONFIG_CAVEAT where the precedence is GLX_NONE, GLX_SLOW_CONFIG, and GLX_NON_CONFORMANT_CONFIG.

2. Larger total number of RGBA color components (GLX_RED_SIZE, GLX_GREEN_SIZE, GLX_BLUE_SIZE, plus GLX_ALPHA_SIZE) that have higher number of bits. If the requested number of bits in *attrib_list* is zero or GLX_DONT_CARE for a particular color component, then the number of bits for that component is not considered.

3. Smaller GLX_BUFFER_SIZE.

4. Single buffered configuration (GLX_DOUBLEBUFFER being `False` precedes a double buffered one.

5. Smaller GLX_AUX_BUFFERS.

6. Larger GLX_DEPTH_SIZE.

7. Smaller GLX_STENCIL_SIZE.

8. Larger total number of accumulation buffer color components (GLX_ACCUM_RED_SIZE, GLX_ACCUM_GREEN_SIZE, GLX_ACCUM_BLUE_SIZE, plus GLX_ACCUM_ALPHA_SIZE) that have higher number of bits. If the requested number of bits in *attrib_list* is zero or GLX_DONT_CARE for a

particular color component, then the number of bits for that component is not considered.

9. By `GLX_X_VISUAL_TYPE` where the precedence order is `GLX_TRUE_COLOR`, `GLX_DIRECT_COLOR`, `GLX_PSEUDO_COLOR`, `GLX_STATIC_COLOR`, `GLX_GRAY_SCALE`, `GLX_STATIC_GRAY`.

`NULL` is returned if an undefined GLX attribute is encountered in *attrib_list*, if *screen* is invalid, or if *dpy* does not support the GLX extension.

`XVisualInfo* glXChooseVisual` *dpy screen attribList* [Function]

Return a visual that matches specified attributes.

dpy Specifies the connection to the X server.

screen Specifies the screen number.

attribList Specifies a list of boolean attributes and integer attribute/value pairs. The last attribute must be `None`.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`GLX_ACCUM_GREEN_SIZE`
: Must be followed by a nonnegative minimum size specification. If this value is zero, visuals with no green accumulation buffer are preferred.

Otherwise, the largest possible green accumulation buffer of at least the minimum size is preferred.

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

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

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

`void glXCopyContext` *dpy src dst mask* [Function]

Copy state from one rendering context to another.

dpy Specifies the connection to the X server.

src Specifies the source context.

dst Specifies the destination context.

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

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

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

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

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

`BadMatch` is generated if rendering contexts *src* and *dst* do not share an address space or were not created with respect to the same screen.

`BadAccess` is generated if *dst* is current to any thread (including the calling thread) at the time `glXCopyContext` is called.

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

`GLXBadContext` is generated if either *src* or *dst* is not a valid GLX context.

`GLXContext glXCreateContext` *dpy vis shareList direct* [Function]

Create a new GLX rendering context.

dpy Specifies the connection to the X server.

vis Specifies the visual that defines the frame buffer resources available to the rendering context. It is a pointer to an `XVisualInfo` structure, not a visual ID or a pointer to a `Visual`.

shareList Specifies the context with which to share display lists. `NULL` indicates that no sharing is to take place.

direct Specifies whether rendering is to be done with a direct connection to the graphics system if possible (`True`) or through the X server (`False`).

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

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

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

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

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

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

`BadValue` is generated if *vis* is not a valid visual (for example, if a particular GLX implementation does not support it).

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

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

`GLXPixmap glXCreateGLXPixmap` *dpy vis pixmap* [Function]

Create an off-screen GLX rendering area.

dpy Specifies the connection to the X server.

vis Specifies the visual that defines the structure of the rendering area. It is a pointer to an `XVisualInfo` structure, not a visual ID or a pointer to a `Visual`.

pixmap Specifies the X pixmap that will be used as the front left color buffer of the off-screen rendering area.

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

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

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

`BadMatch` is generated if the depth of *pixmap* does not match the depth value reported by core X11 for *vis*, or if *pixmap* was not created with respect to the same screen as *vis*.

`BadValue` is generated if *vis* is not a valid XVisualInfo pointer (for example, if a particular GLX implementation does not support this visual).

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

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

`GLXContext glXCreateNewContext` *dpy config render_type share_list direct* [Function]

Create a new GLX rendering context.

dpy Specifies the connection to the X server.

config Specifies the GLXFBConfig structure with the desired attributes for the context.

render_type
Specifies the type of the context to be created. Must be one of `GLX_RGBA_TYPE` or `GLX_COLOR_INDEX_TYPE`.

share_list Specifies the context with which to share display lists. `NULL` indicates that no sharing is to take place.

share_list Specifies whether rendering is to be done with a direct connection to the graphics system if possible (`True`) or through the X server (`False`).

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

If *render_type* is `GLX_RGBA_TYPE`, then a context that supports RGBA rendering is created. If *config* is `GLX_COLOR_INDEX_TYPE`, then context supporting color-index rendering is created.

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

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

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

`GLXBadContext` is generated if *render_type* is not a GLX context and is not `NULL`.

`GLXBadFBConfig` is generated if *config* is not a valid GLXFBConfig.

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

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

`BadValue` is generated if *config* is not a valid visual (for example, if a particular GLX implementation does not support it).

`GLXPbuffer glXCreatePbuffer` *dpy config attrib_list* [Function]

Create an off-screen rendering area.

dpy Specifies the connection to the X server.

config Specifies a GLXFBConfig structure with the desired attributes for the window.

attrib_list Specifies a list of attribute value pairs, which must be terminated with `None` or `NULL`. Accepted attributes are `GLX_PBUFFER_WIDTH`, `GLX_PBUFFER_HEIGHT`, `GLX_PRESERVED_CONTENTS`, and `GLX_LARGEST_PBUFFER`.

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

The accepted attributes for a GLXPbuffer are:

`GLX_PBUFFER_WIDTH`

Specify the pixel width of the requested GLXPbuffer. The default value is 0.

`GLX_PBUFFER_HEIGHT`
: Specify the pixel height of the requested GLXPbuffer. The default value is 0.

`GLX_LARGEST_PBUFFER`
: Specify to obtain the largest available pixel buffer, if the requested allocation would have failed. The width and height of the allocated pixel buffer will never exceed the specified `GLX_PBUFFER_WIDTH` or `GLX_PBUFFER_HEIGHT`, respectively. Use `glXQueryDrawable` to retrieve the dimensions of the allocated pixel buffer. The default value is `False`.

`GLX_PRESERVED_CONTENTS`
: Specify if the contents of the pixel buffer should be preserved when a resource conflict occurs. If set to `False`, the contents of the pixel buffer may be lost at any time. If set to `True`, or not specified in *attrib_list*, then the contents of the pixel buffer will be preserved (most likely by copying the contents into main system memory from the frame buffer). In either case, the client can register (using `glXSelectEvent`, to receive pixel buffer clobber events that are generated when the pbuffer contents have been preserved or damaged.

GLXPbuffers contain the color and ancillary buffers specified by *config*. It is possible to create a pixel buffer with back buffers and to swap those buffers using `glXSwapBuffers`.

`BadAlloc` is generated if there are insufficient resources to allocate the requested GLXPbuffer.

`GLXBadFBConfig` is generated if *config* is not a valid GLXFBConfig.

`BadMatch` is generated if *config* does not support rendering to pixel buffers (e.g., `GLX_DRAWABLE_TYPE` does not contain `GLX_PBUFFER_BIT`).

`GLXPixmap glXCreatePixmap` *dpy config pixmap attrib_list* [Function]

Create an off-screen rendering area.

dpy Specifies the connection to the X server.

config Specifies a GLXFBConfig structure with the desired attributes for the window.

pixmap Specifies the X pixmap to be used as the rendering area.

attrib_list Currently unused. This must be set to `NULL` or be an empty list (i.e., one in which the first element is `None`).

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

`BadMatch` is generated if *pixmap* was not created with a visual that corresponds to *config*.

`BadMatch` is generated if *config* does not support rendering to windows (e.g., `GLX_DRAWABLE_TYPE` does not contain `GLX_WINDOW_BIT`).

`BadWindow` is generated if *pixmap* is not a valid window XID. `BadAlloc` is generated if there is already a GLXFBConfig associated with *pixmap*.

`BadAlloc` is generated if the X server cannot allocate a new GLX window.

`GLXBadFBConfig` is generated if *config* is not a valid GLXFBConfig.

`GLXWindow glXCreateWindow` *dpy config win attrib_list* [Function]

Create an on-screen rendering area.

dpy Specifies the connection to the X server.

config Specifies a GLXFBConfig structure with the desired attributes for the window.

win Specifies the X window to be used as the rendering area.

attrib_list Currently unused. This must be set to `NULL` or be an empty list (i.e., one in which the first element is `None`).

`glXCreateWindow` creates an on-screen rendering area from an existing X window that was created with a visual matching *config*. The XID of the GLXWindow is returned. Any GLX rendering context that was created with respect to *config* can be used to render into this window. Use `glXMakeContextCurrent` to associate the rendering area with a GLX rendering context.

`BadMatch` is generated if *win* was not created with a visual that corresponds to *config*.

`BadMatch` is generated if *config* does not support rendering to windows (i.e., `GLX_DRAWABLE_TYPE` does not contain `GLX_WINDOW_BIT`).

`BadWindow` is generated if *win* is not a valid pixmap XID.

`BadAlloc` is generated if there is already a GLXFBConfig associated with *win*.

`BadAlloc` is generated if the X server cannot allocate a new GLX window.

`GLXBadFBConfig` is generated if *config* is not a valid GLXFBConfig.

`void glXDestroyContext` *dpy ctx* [Function]

Destroy a GLX context.

dpy Specifies the connection to the X server.

ctx Specifies the GLX context to be destroyed.

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

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

`void glXDestroyGLXPixmap` *dpy pix* [Function]

Destroy a GLX pixmap.

dpy Specifies the connection to the X server.

pix Specifies the GLX pixmap to be destroyed.

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

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

`void glXDestroyPbuffer` *dpy pbuf* [Function]
Destroy an off-screen rendering area.

dpy Specifies the connection to the X server.

pbuf Specifies the GLXPbuffer to be destroyed.

`glXDestroyPbuffer` destroys a GLXPbuffer created by `glXCreatePbuffer`.

`GLXBadPbuffer` is generated if *pbuf* is not a valid GLXPbuffer.

`void glXDestroyPixmap` *dpy pixmap* [Function]
Destroy an off-screen rendering area.

dpy Specifies the connection to the X server.

pixmap Specifies the GLXPixmap to be destroyed.

`glXDestroyPixmap` destroys a GLXPixmap created by `glXCreatePixmap`.

`GLXBadPixmap` is generated if *pixmap* is not a valid GLXPixmap.

`void glXDestroyWindow` *dpy win* [Function]
Destroy an on-screen rendering area.

dpy Specifies the connection to the X server.

win Specifies the GLXWindow to be destroyed.

`glXDestroyWindow` destroys a GLXWindow created by `glXCreateWindow`.

`GLXBadWindow` is generated if *win* is not a valid GLXPixmap.

`void glXFreeContextEXT` *dpy ctx* [Function]
Free client-side memory for imported context.

dpy Specifies the connection to the X server.

ctx Specifies a GLX rendering context.

`glXFreeContextEXT` frees the client-side part of a GLXContext that was created with `glXImportContextEXT`. `glXFreeContextEXT` does not free the server-side context information or the XID associated with the server-side context.

`glXFreeContextEXT` is part of the `EXT_import_context` extension, not part of the core GLX command set. If _glxextstring(EXT_import_context) is included in the string returned by `glXQueryExtensionsString`, when called with argument `GLX_EXTENSIONS`, extension `EXT_vertex_array` is supported.

`GLXBadContext` is generated if *ctx* does not refer to a valid context.

`const-char-* glXGetClientString` *dpy name* [Function]
Return a string describing the client.

dpy Specifies the connection to the X server.

name Specifies which string is returned. The symbolic constants `GLX_VENDOR`, `GLX_VERSION`, and `GLX_EXTENSIONS` are accepted.

`glXGetClientString` returns a string describing some aspect of the client library. The possible values for *name* are `GLX_VENDOR`, `GLX_VERSION`, and `GLX_EXTENSIONS`. If *name* is not set to one of these values, `glXGetClientString` returns `NULL`. The format and contents of the vendor string is implementation dependent.

The extensions string is null-terminated and contains a space-separated list of extension names. (The extension names never contain spaces.) If there are no extensions to GLX, then the empty string is returned.

The version string is laid out as follows:

<major_version.minor_version><space><vendor-specific info>

Both the major and minor portions of the version number are of arbitrary length. The vendor-specific information is optional. However, if it is present, the format and contents are implementation specific.

`int glXGetConfig` *dpy vis attrib value* [Function]

Return information about GLX visuals.

dpy Specifies the connection to the X server.

vis Specifies the visual to be queried. It is a pointer to an `XVisualInfo` structure, not a visual ID or a pointer to a `Visual`.

attrib Specifies the visual attribute to be returned.

value Returns the requested value.

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

attrib is one of the following:

`GLX_USE_GL`
: `True` if OpenGL rendering is supported by this visual, `False` otherwise.

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

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

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

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

`GLX_STEREO`
: `True` if color buffers exist in left/right pairs, `False` otherwise.

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

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

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

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

`GLX_ALPHA_SIZE`
: Number of bits of alpha stored in each color buffer. Undefined if `GLX_RGBA` is `False`.

`GLX_DEPTH_SIZE`
: Number of bits in the depth buffer.

`GLX_STENCIL_SIZE`
: Number of bits in the stencil buffer.

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

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

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

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

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

Although a GLX implementation can export many visuals that support GL rendering, it must support at least one RGBA visual. This visual must have at least one color buffer, a stencil buffer of at least 1 bit, a depth buffer of at least 12 bits, and an accumulation buffer. Alpha bitplanes are optional in this visual. However, its color buffer size must be as great as that of the deepest `TrueColor`, `DirectColor`, `PseudoColor`, or `StaticColor` visual supported on level zero, and it must itself be made available on level zero.

In addition, if the X server exports a `PseudoColor` or `StaticColor` visual on framebuffer level 0, a color index visual is also required on that level. It must have at least

one color buffer, a stencil buffer of at least 1 bit, and a depth buffer of at least 12 bits. This visual must have as many color bitplanes as the deepest `PseudoColor` or `StaticColor` visual supported on level 0.

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

`GLX_NO_EXTENSION` is returned if *dpy* does not support the GLX extension.

`GLX_BAD_SCREEN` is returned if the screen of *vis* does not correspond to a screen.

`GLX_BAD_ATTRIBUTE` is returned if *attrib* is not a valid GLX attribute.

`GLX_BAD_VISUAL` is returned if *vis* doesn't support GLX and an attribute other than `GLX_USE_GL` is requested.

`GLXContextID glXGetContextIDEXT` *ctx* [Function]

Get the XID for a context..

ctx Specifies a GLX rendering context.

`glXGetContextIDEXT` returns the XID associated with a GLXContext.

No round trip is forced to the server; unlike most X calls that return a value, `glXGetContextIDEXT` does not flush any pending events.

`glXGetContextIDEXT` is part of the `EXT_import_context` extension, not part of the core GLX command set. If _glxextstring(EXT_import_context) is included in the string returned by `glXQueryExtensionsString`, when called with argument `GLX_EXTENSIONS`, extension `EXT_import_context` is supported.

`GLXBadContext` is generated if *ctx* does not refer to a valid context.

`GLXContext glXGetCurrentContext` [Function]

Return the current context.

`glXGetCurrentContext` returns the current context, as specified by `glXMakeCurrent`. If there is no current context, `NULL` is returned.

`glXGetCurrentContext` returns client-side information. It does not make a round trip to the server.

`Display-* glXGetCurrentDisplay` [Function]

Get display for current context.

`glXGetCurrentDisplay` returns the display for the current context. If no context is current, `NULL` is returned.

`glXGetCurrentDisplay` returns client-side information. It does not make a round-trip to the server, and therefore does not flush any pending events.

`GLXDrawable glXGetCurrentDrawable` [Function]

Return the current drawable.

`glXGetCurrentDrawable` returns the current drawable, as specified by `glXMakeCurrent`. If there is no current drawable, `None` is returned.

`glXGetCurrentDrawable` returns client-side information. It does not make a round trip to the server.

`GLXDrawable glXGetCurrentReadDrawable` [Function]

Return the current drawable.

`glXGetCurrentReadDrawable` returns the current read drawable, as specified by `read` parameter of `glXMakeContextCurrent`. If there is no current drawable, `None` is returned.

`glXGetCurrentReadDrawable` returns client-side information. It does not make a round-trip to the server.

`int glXGetFBConfigAttrib` *dpy config attribute value* [Function]

Return information about a GLX frame buffer configuration.

dpy Specifies the connection to the X server.

config Specifies the GLX frame buffer configuration to be queried.

attribute Specifies the attribute to be returned.

value Returns the requested value.

`glXGetFBConfigAttrib` sets *value* to the *attribute* value of GLX drawables created with respect to *config*. `glXGetFBConfigAttrib` returns an error code if it fails for any reason. Otherwise, `Success` is returned.

attribute is one of the following:

`GLX_FBCONFIG_ID`
: XID of the given GLXFBConfig.

`GLX_BUFFER_SIZE`
: Number of bits per color buffer. If the frame buffer configuration supports RGBA contexts, then `GLX_BUFFER_SIZE` is the sum of `GLX_RED_SIZE`, `GLX_GREEN_SIZE`, `GLX_BLUE_SIZE`, and `GLX_ALPHA_SIZE`. If the frame buffer configuration supports only color index contexts, `GLX_BUFFER_SIZE` is the size of the color indexes.

`GLX_LEVEL`
: Frame buffer level of the configuration. 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 underlie the default buffer.

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

`GLX_STEREO`
: `True` if color buffers exist in left/right pairs, `False` otherwise.

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

`GLX_RED_SIZE`
: Number of bits of red stored in each color buffer. Undefined if RGBA contexts are not supported by the frame buffer configuration.

`GLX_GREEN_SIZE`
: Number of bits of green stored in each color buffer. Undefined if RGBA contexts are not supported by the frame buffer configuration.

`GLX_BLUE_SIZE`
: Number of bits of blue stored in each color buffer. Undefined if RGBA contexts are not supported by the frame buffer configuration.

`GLX_ALPHA_SIZE`
: Number of bits of alpha stored in each color buffer. Undefined if RGBA contexts are not supported by the frame buffer configuration.

`GLX_DEPTH_SIZE`
: Number of bits in the depth buffer.

`GLX_STENCIL_SIZE`
: Number of bits in the stencil buffer.

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

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

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

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

`GLX_RENDER_TYPE`
: Mask indicating what type of GLX contexts can be made current to the frame buffer configuration. Valid bits are `GLX_RGBA_BIT` and `GLX_COLOR_INDEX_BIT`.

`GLX_DRAWABLE_TYPE`
: Mask indicating what drawable types the frame buffer configuration supports. Valid bits are `GLX_WINDOW_BIT`, `GLX_PIXMAP_BIT`, and `GLX_PBUFFER_BIT`.

`GLX_X_RENDERABLE`
: `True` if drawables created with the frame buffer configuration can be rendered to by X.

`GLX_VISUAL_ID`
: XID of the corresponding visual, or zero if there is no associated visual (i.e., if `GLX_X_RENDERABLE` is `False` or `GLX_DRAWABLE_TYPE` does not have the `GLX_WINDOW_BIT` bit set).

`GLX_X_VISUAL_TYPE`
: Visual type of associated visual. The returned value will be one of: `GLX_TRUE_COLOR`, `GLX_DIRECT_COLOR`, `GLX_PSEUDO_COLOR`, `GLX_STATIC_COLOR`, `GLX_GRAY_SCALE`, `GLX_STATIC_GRAY`, or `GLX_NONE`, if there is no associated visual (i.e., if `GLX_X_RENDERABLE` is `False` or `GLX_DRAWABLE_TYPE` does not have the `GLX_WINDOW_BIT` bit set).

`GLX_CONFIG_CAVEAT`
: One of `GLX_NONE`, `GLX_SLOW_CONFIG`, or `GLX_NON_CONFORMANT_CONFIG`, indicating that the frame buffer configuration has no caveats, some aspect of the frame buffer configuration runs slower than other frame buffer configurations, or some aspect of the frame buffer configuration is non-conformant, respectively.

`GLX_TRANSPARENT_TYPE`
: One of `GLX_NONE`, `GLX_TRANSPARENT_RGB`, `GLX_TRANSPARENT_INDEX`, indicating that the frame buffer configuration is opaque, is transparent for particular values of red, green, and blue, or is transparent for particular index values, respectively.

`GLX_TRANSPARENT_INDEX_VALUE`
: Integer value between 0 and the maximum frame buffer value for indices, indicating the transparent index value for the frame buffer configuration. Undefined if `GLX_TRANSPARENT_TYPE` is not `GLX_TRANSPARENT_INDEX`.

`GLX_TRANSPARENT_RED_VALUE`
: Integer value between 0 and the maximum frame buffer value for red, indicating the transparent red value for the frame buffer configuration. Undefined if `GLX_TRANSPARENT_TYPE` is not `GLX_TRANSPARENT_RGB`.

`GLX_TRANSPARENT_GREEN_VALUE`
: Integer value between 0 and the maximum frame buffer value for green, indicating the transparent green value for the frame buffer configuration. Undefined if `GLX_TRANSPARENT_TYPE` is not `GLX_TRANSPARENT_RGB`.

`GLX_TRANSPARENT_BLUE_VALUE`
: Integer value between 0 and the maximum frame buffer value for blue, indicating the transparent blue value for the frame buffer configuration. Undefined if `GLX_TRANSPARENT_TYPE` is not `GLX_TRANSPARENT_RGB`.

`GLX_TRANSPARENT_ALPHA_VALUE`
: Integer value between 0 and the maximum frame buffer value for alpha, indicating the transparent blue value for the frame buffer configuration. Undefined if `GLX_TRANSPARENT_TYPE` is not `GLX_TRANSPARENT_RGB`.

`GLX_MAX_PBUFFER_WIDTH`
: The maximum width that can be specified to `glXCreatePbuffer`.

`GLX_MAX_PBUFFER_HEIGHT`
: The maximum height that can be specified to `glXCreatePbuffer`.

`GLX_MAX_PBUFFER_PIXELS`
: The maximum number of pixels (width times height) for a pixel buffer. Note that this value may be less than `GLX_MAX_PBUFFER_WIDTH` times `GLX_MAX_PBUFFER_HEIGHT`. Also, this value is static and assumes that no other pixel buffers or X resources are contending for the frame buffer memory. As a result, it may not be possible to allocate a pixel buffer of the size given by `GLX_MAX_PBUFFER_PIXELS`.

Applications should choose the frame buffer configuration that most closely meets their requirements. Creating windows, GLX pixmaps, or GLX pixel buffers with unnecessary buffers can result in reduced rendering performance as well as poor resource allocation.

`GLX_NO_EXTENSION` is returned if *dpy* does not support the GLX extension. `GLX_BAD_ATTRIBUTE` is returned if *attribute* is not a valid GLX attribute.

`GLXFBConfig-* glXGetFBConfigs` *dpy screen nelements* [Function]

List all GLX frame buffer configurations for a given screen.

dpy Specifies the connection to the X server.

screen Specifies the screen number.

nelements Returns the number of GLXFBConfigs returned.

`glXGetFBConfigs` returns a list of all GLXFBConfigs available on the screen specified by *screen*. Use `glXGetFBConfigAttrib` to obtain attribute values from a specific GLXFBConfig.

`void(*)() glXGetProcAddress` *procName* [Function]

Obtain a pointer to an OpenGL or GLX function.

procName Specifies the name of the OpenGL or GLX function whose address is to be returned.

`glXGetProcAddress` returns the address of the function specified in *procName*. This is necessary in environments where the OpenGL link library exports a different set of functions than the runtime library.

`void glXGetSelectedEvent` *dpy draw event_mask* [Function]

Returns GLX events that are selected for a window or a GLX pixel buffer.

dpy Specifies the connection to the X server.

draw Specifies a GLX drawable. Must be a GLX pixel buffer or a window.

event_mask
Returns the events that are selected for *draw*.

`glXGetSelectedEvent` returns in *event_mask* the events selected for *draw*.

`GLXBadDrawable` is generated if *draw* is not a valid window or a valid GLX pixel buffer.

`XVisualInfo-* glXGetVisualFromFBConfig` *dpy config* [Function]

Return visual that is associated with the frame buffer configuration.

dpy Specifies the connection to the X server.

config Specifies the GLX frame buffer configuration.

If *config* is a valid GLX frame buffer configuration and it has an associated X Visual, then information describing that visual is returned; otherwise `NULL` is returned. Use `XFree` to free the data returned.

Returns `NULL` if *config* is not a valid GLXFBConfig.

`GLXContext glXImportContextEXT` *dpy contextID* [Function]

Import another process's indirect rendering context..

dpy Specifies the connection to the X server.

contextID Specifies a GLX rendering context.

`glXImportContextEXT` creates a GLXContext given the XID of an existing GLXContext. It may be used in place of `glXCreateContext`, to share another process's indirect rendering context.

Only the server-side context information can be shared between X clients; client-side state, such as pixel storage modes, cannot be shared. Thus, `glXImportContextEXT` must allocate memory to store client-side information. This memory is freed by calling `glXFreeContextEXT`.

This call does not create a new XID. It merely makes an existing object available to the importing client (Display *). Like any XID, it goes away when the creating client drops its connection or the ID is explicitly deleted. Note that this is when the XID goes away. The object goes away when the XID goes away AND the context is not current to any thread.

If *contextID* refers to a direct rendering context then no error is generated but `glXImportContextEXT` returns NULL.

`glXImportContextEXT` is part of the `EXT_import_context` extension, not part of the core GLX command set. If _glxextstring(EXT_import_context) is included in the string returned by `glXQueryExtensionsString`, when called with argument `GLX_EXTENSIONS`, extension `EXT_import_context` is supported.

`GLXBadContext` is generated if *contextID* does not refer to a valid context.

`Bool glXIsDirect` *dpy ctx* [Function]

Indicate whether direct rendering is enabled.

dpy Specifies the connection to the X server.

ctx Specifies the GLX context that is being queried.

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

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

`Bool glXMakeContextCurrent` *display draw read ctx* [Function]

Attach a GLX context to a GLX drawable.

display Specifies the connection to the X server.

draw Specifies a GLX drawable to render into. Must be an XID representing a GLXWindow, GLXPixmap, or GLXPbuffer.

read Specifies a GLX drawable to read from. Must be an XID representing a GLXWindow, GLXPixmap, or GLXPbuffer.

ctx Specifies the GLX context to be bound to *read* and *ctx*.

`glXMakeContextCurrent` binds *ctx* to the current rendering thread and to the *draw* and *read* GLX drawables. *draw* and *read* may be the same.

draw is used for all OpenGL operations except:

Any pixel data that are read based on the value of `GLX_READ_BUFFER`. Note that accumulation operations use the value of `GLX_READ_BUFFER`, but are not allowed unless *draw* is identical to *read*.

Any depth values that are retrieved by `glReadPixels` or `glCopyPixels`.

Any stencil values that are retrieved by `glReadPixels` or `glCopyPixels`.

Frame buffer values are taken from *draw*.

If the current rendering thread has a current rendering context, that context is flushed and replaced by *ctx*.

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

To release the current context without assigning a new one, call `glXMakeContextCurrent` with *draw* and *read* set to `None` and *ctx* set to `NULL`.

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

`BadMatch` is generated if *draw* and *read* are not compatible.

`BadAccess` is generated if *ctx* is current to some other thread.

`GLXContextState` is generated if there is a current rendering context and its render mode is either `GLX_FEEDBACK` or `GLX_SELECT`.

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

`GLXBadDrawable` is generated if *draw* or *read* is not a valid GLX drawable.

`GLXBadWindow` is generated if the underlying X window for either *draw* or *read* is no longer valid.

`GLXBadCurrentDrawable` is generated if the previous context of the calling thread has unflushed commands and the previous drawable is no longer valid.

`BadAlloc` is generated if the X server does not have enough resources to allocate the buffers.

`BadMatch` is generated if:

draw and *read* cannot fit into frame buffer memory simultaneously.

draw or *read* is a GLXPixmap and *ctx* is a direct-rendering context.

draw or *read* is a GLXPixmap and *ctx* was previously bound to a GLXWindow or GLXPbuffer.

draw or *read* is a GLXWindow or GLXPbuffer and *ctx* was previously bound to a GLXPixmap.

`Bool glXMakeCurrent` *dpy drawable ctx* [Function]

Attach a GLX context to a window or a GLX pixmap.

dpy Specifies the connection to the X server.

drawable Specifies a GLX drawable. Must be either an X window ID or a GLX pixmap ID.

ctx Specifies a GLX rendering context that is to be attached to *drawable*.

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

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

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

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

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

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

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

`GLXBadDrawable` is generated if *drawable* is not a valid GLX drawable.

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

`GLXBadContextState` is generated if `glXMakeCurrent` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

`GLXBadContextState` is also generated if the rendering context current to the calling thread has GL renderer state `GLX_FEEDBACK` or `GLX_SELECT`.

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

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

`int glXQueryContextInfoEXT` *dpy ctx attribute value* [Function]

Query context information.

dpy Specifies the connection to the X server.

ctx Specifies a GLX rendering context.

attribute Specifies that a context parameter should be retrieved. Must be one of `GLX_SHARED_CONTEXT_EXT`, `GLX_VISUAL_ID_EXT`, or `GLX_SCREEN_EXT`.

value Contains the return value for *attribute*.

`glXQueryContextInfoEXT` sets *value* to the value of *attribute* with respect to *ctx*. `glXQueryContextInfoEXT` returns an error code if it fails for any reason. Otherwise, `Success` is returned.

attribute may be one of the following:

`GLX_SHARED_CONTEXT_EXT`
: Returns the XID of the share list context associated with *ctx* at its creation.

`GLX_VISUAL_ID_EXT`
: Returns the XID of the GLX Visual associated with *ctx*.

`GLX_SCREEN_EXT`
: Returns the screen number associated with *ctx*.

This call may cause a round-trip to the server.

`glXQueryContextInfoEXT` is part of the `EXT_import_context` extension, not part of the core GLX command set. If _glxextstring(EXT_import_context) is included in the string returned by `glXQueryExtensionsString`, when called with argument `GLX_EXTENSIONS`, extension `EXT_import_context` is supported.

`GLXBadContext` is generated if *ctx* does not refer to a valid context.

`GLX_BAD_ATTRIBUTE` is returned if *attribute* is not a valid GLX context attribute.

fred `GLX_BAD_CONTEXT` is returned if *attribute* is not a valid context.

`int glXQueryContext` *dpy ctx attribute value* [Function]

Query context information.

dpy Specifies the connection to the X server.

ctx Specifies a GLX rendering context.

attribute Specifies that a context parameter should be retrieved. Must be one of `GLX_FBCONFIG_ID`, `GLX_RENDER_TYPE`, or `GLX_SCREEN`.

value Contains the return value for *attribute*.

`glXQueryContext` sets *value* to the value of *attribute* with respect to *ctx*. *attribute* may be one of the following:

`GLX_FBCONFIG_ID`
: Returns the XID of the GLXFBConfig associated with *ctx*.

`GLX_RENDER_TYPE`
: Returns the rendering type supported by *ctx*.

`GLX_SCREEN`
: Returns the screen number associated with *ctx*.

`Success` is returned unless *attribute* is not a valid GLX context attribute, in which case `GLX_BAD_ATTRIBUTE` is returned.

This call may cause a round-trip to the server.

`GLXBadContext` is generated if *ctx* does not refer to a valid context.

`int glXQueryDrawable` *dpy draw attribute value* [Function]

Returns an attribute assoicated with a GLX drawable.

dpy Specifies the connection to the X server.

draw Specifies the GLX drawable to be queried.

attribute Specifies the attribute to be returned. Must be one of `GLX_WIDTH`, `GLX_HEIGHT`, `GLX_PRESERVED_CONTENTS`, `GLX_LARGEST_PBUFFER`, or `GLX_FBCONFIG_ID`.

value Contains the return value for *attribute*.

`glXQueryDrawable` sets *value* to the value of *attribute* with respect to the GLXDrawable *draw*.

attribute may be one of the following:

`GLX_WIDTH`
Returns the width of *ctx*.

`GLX_HEIGHT`
Returns the height of *ctx*.

`GLX_PRESERVED_CONTENTS`
Returns `True` if the contents of a GLXPbuffer are preserved when a resource conflict occurs; `False` otherwise.

`GLX_LARGEST_PBUFFER`
Returns the value set when `glXCreatePbuffer` was called to create the GLXPbuffer. If `False` is returned, then the call to `glXCreatePbuffer` will fail to create a GLXPbuffer if the requested size is larger than the implementation maximum or available resources. If `True` is returned, a GLXPbuffer of the maximum availble size (if less than the requested width and height) is created.

`GLX_FBCONFIG_ID`
Returns the XID for *draw*.

If *draw* is a GLXWindow or GLXPixmap and *attribute* is set to `GLX_PRESERVED_CONTENTS` or `GLX_LARGETST_PBUFFER`, the contents of *value* are undefined. If *attribute* is not one of the attributes listed above, the contents of *value* are unedfined.

A `GLXBadDrawable` is generated if *draw* is not a valid GLXDrawable.

`const-char-* glXQueryExtensionsString` *dpy screen* [Function]

Return list of supported extensions.

dpy Specifies the connection to the X server.

screen Specifies the screen number.

`glXQueryExtensionsString` returns a pointer to a string describing which GLX extensions are supported on the connection. The string is null-terminated and contains a space-separated list of extension names. (The extension names themselves never contain spaces.) If there are no extensions to GLX, then the empty string is returned.

`Bool glXQueryExtension` *dpy errorBase eventBase* [Function]
Indicate whether the GLX extension is supported.

dpy Specifies the connection to the X server.

errorBase Returns the base error code of the GLX server extension.

eventBase Returns the base event code of the GLX server extension.

`glXQueryExtension` returns `True` if the X server of connection *dpy* supports the GLX extension, `False` otherwise. If `True` is returned, then *errorBase* and *eventBase* return the error base and event base of the GLX extension. These values should be added to the constant error and event values to determine the actual event or error values. Otherwise, *errorBase* and *eventBase* are unchanged.

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

`const-char-* glXQueryServerString` *dpy screen name* [Function]
Return string describing the server.

dpy Specifies the connection to the X server.

screen Specifies the screen number.

name Specifies which string is returned: one of `GLX_VENDOR`, `GLX_VERSION`, or `GLX_EXTENSIONS`.

`glXQueryServerString` returns a pointer to a static, null-terminated string describing some aspect of the server's GLX extension. The possible values for *name* and the format of the strings is the same as for `glXGetClientString`. If *name* is not set to a recognized value, `NULL` is returned.

`Bool glXQueryVersion` *dpy major minor* [Function]
Return the version numbers of the GLX extension.

dpy Specifies the connection to the X server.

major Returns the major version number of the GLX server extension.

minor Returns the minor version number of the GLX server extension.

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

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

`glXQueryVersion` returns `False` if it fails, `True` otherwise.

major and *minor* are not updated when `False` is returned.

`void glXSelectEvent` *dpy draw event_mask* [Function]
Select GLX events for a window or a GLX pixel buffer.

dpy Specifies the connection to the X server.

draw Specifies a GLX drawable. Must be a GLX pixel buffer or a window.

event_mask
Specifies the events to be returned for *draw*.

`glXSelectEvent` sets the GLX event mask for a GLX pixel buffer or a window. Calling `glXSelectEvent` overrides any previous event mask that was set by the client for *draw*. Note that it does not affect the event masks that other clients may have specified for *draw* since each client rendering to *draw* has a separate event mask for it.

Currently, only one GLX event, `GLX_PBUFFER_CLOBBER_MASK`, can be selected. The following data is returned to the client when a `GLX_PBUFFER_CLOBBER_MASK` event occurs:

typedef struct {

int *event_type*;
/* GLX_DAMAGED or GLX_SAVED */

int *draw_type*;
/* GLX_WINDOW or GLX_PBUFFER */

unsigned long *serial*;
/* # of last request processed by server */

Bool *send_event*;
/* true if this came for SendEvent request */

Display **display*;
/* display the event was read from */

GLXDrawable *drawable*;
/* i.d. of Drawable */

unsigned int *buffer_mask*;
/* mask indicating affected buffers */

int *x, y*;

int *width, height*;
int *count*; /* if nonzero, at least this many more */

} GLXPbufferClobberEvent; The valid bit masks used in *buffer_mask* are:

Bitmask **Corresponding Buffer**

`GLX_FRONT_LEFT_BUFFER_BIT`
Front left color buffer

`GLX_FRONT_RIGHT_BUFFER_BIT`
Front right color buffer

`GLX_BACK_LEFT_BUFFER_BIT`
Back left color buffer

`GLX_BACK_RIGHT_BUFFER_BIT`
Back right color buffer

`GLX_AUX_BUFFERS_BIT`
: Auxiliary buffer

`GLX_DEPTH_BUFFER_BIT`
: Depth buffer

`GLX_STENCIL_BUFFER_BIT`
: Stencil buffer

`GLX_ACCUM_BUFFER_BIT`
: Accumulation buffer

A single X server operation can cause several buffer clobber events to be sent. (e.g., a single GLX pixel buffer may be damaged and cause multiple buffer clobber events to be generated). Each event specifies one region of the GLX drawable that was affected by the X Server operation. The *buffer_mask* field indicates which color buffers and ancillary buffers were affected. All the buffer clobber events generated by a single X server action are guaranteed to be contiguous in the event queue. The conditions under which this event is generated and the *event_type* varies, depending on the type of the GLX drawable.

When the `GLX_AUX_BUFFERS_BIT` is set in *buffer_mask*, then *aux_buffer* is set to indicate which buffer was affected. If more than one aux buffer was affected, then additional events are generated as part of the same contiguous event group. Each additional event will have only the `GLX_AUX_BUFFERS_BIT` set in *buffer_mask*, and the *aux_buffer* field will be set appropriately. For nonstereo drawables, `GLX_FRONT_LEFT_BUFFER_BIT` and `GLX_BACK_LEFT_BUFFER_BIT` are used to specify the front and back color buffers.

For preserved GLX pixel buffers, a buffer clobber event with type `GLX_SAVED` is generated whenever the contents of the GLX pixel buffer is moved out of offscreen memory. The event(s) describes which portions of the GLX pixel buffer were affected. Clients who receive many buffer clobber events, referring to different save actions, should consider freeing the GLX pixel buffer resource in order to prevent the system from thrashing due to insufficient resources.

For an unpreserved GLXPbuffer, a buffer clobber event, with type `GLX_DAMAGED`, is generated whenever a portion of the GLX pixel buffer becomes invalid. The client may wish to regenerate the invalid portions of the GLX pixel buffer.

For Windows, buffer clobber events, with type `GLX_SAVED`, occur whenever an ancillary buffer, associated with the window, gets clobbered or moved out of off-screen memory. The event contains information indicating which color buffers and ancillary buffers\(emand which portions of those buffers\(emwere affected.

`GLXBadDrawable` is generated if *draw* is not a valid window or a valid GLX pixel buffer.

`void glXSwapBuffers` *dpy drawable* [Function]

Exchange front and back buffers.

dpy Specifies the connection to the X server.

drawable Specifies the drawable whose buffers are to be swapped.

`glXSwapBuffers` promotes the contents of the back buffer of *drawable* to become the contents of the front buffer of *drawable*. The contents of the back buffer then become undefined. The update typically takes place during the vertical retrace of the monitor, rather than immediately after `glXSwapBuffers` is called.

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

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

`GLXBadDrawable` is generated if *drawable* is not a valid GLX drawable.

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

`void glXUseXFont` *font first count listBase* [Function]

Create bitmap display lists from an X font.

font Specifies the font from which character glyphs are to be taken.

first Specifies the index of the first glyph to be taken.

count Specifies the number of glyphs to be taken.

listBase Specifies the index of the first display list to be generated.

`glXUseXFont` generates *count* display lists, named *listBase* through *listBase*+*count*-1, each containing a single `glBitmap` command. The parameters of the `glBitmap` command of display list *listBase*+*i* are derived from glyph *first*+*i*. Bitmap parameters *xorig*, *yorig*, *width*, and *height* are computed from font metrics as *descent*-1, -*lbearing*, *rbearing*-*lbearing*, and *ascent*+*descent*, respectively. *xmove* is taken from the glyph's *width* metric, and *ymove* is set to zero. Finally, the glyph's image is converted to the appropriate format for `glBitmap`.

Using `glXUseXFont` may be more efficient than accessing the X font and generating the display lists explicitly, both because the display lists are created on the server without requiring a round trip of the glyph data, and because the server may choose to delay the creation of each bitmap until it is accessed.

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

`BadFont` is generated if *font* is not a valid font.

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

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

`void glXWaitGL` [Function]

Complete GL execution prior to subsequent X calls.

GL rendering calls made prior to `glXWaitGL` are guaranteed to be executed before X rendering calls made after `glXWaitGL`. Although this same result can be achieved

using `glFinish`, `glXWaitGL` does not require a round trip to the server, and it is therefore more efficient in cases where client and server are on separate machines.

`glXWaitGL` is ignored if there is no current GLX context.

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

`void glXWaitX` [Function]

Complete X execution prior to subsequent GL calls.

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

`glXWaitX` is ignored if there is no current GLX context.

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

6 GLUT

Import the GLUT module to have access to these procedures:

```
(use-modules (glut))
```

The GLUT specification is available at http://www.opengl.org/resources/libraries/glut/glut-3.spec.pdf.

6.1 GLUT Initialization

`set-initial-display-mode` *mode* [Function]

`set-initial-window-position` *x y* [Function]

`set-initial-window-size` *width height* [Function]

`initialize-glut` [*args*] [*#:window-position*] [*#:window-size*] [*#:display-mode*] [Function]

6.2 Beginning Event Processing

`glut-main-loop` [Function]

6.3 Window Management

`window-id` [Function]

`window-live?` [Function]

`window?` [Function]

`set-window-cursor!` *window cursor* [Function]

`set-window-icon-title!` *window str* [Function]

`set-window-title!` *window str* [Function]

`show-window` [*window*] [Function]

`sub-window?` *window* [Function]

`swap-buffers` [*window*] [Function]

`top-level-window?` *window* [Function]

`with-window` *window body1 body2 ...* [Macro]

`with-window*` _ _ [Function]

`make-sub-window` *window x y width height* [Function]

`make-window` *str* [Function]

`pop-window` [Function]

`position-window` *window x y* [Function]

`post-redisplay` [*window*] [Function]

`push-window` [Function]

`reshape-window` *window width height* [Function]

`current-window` [Function]

`destroy-window` *window* [Function]

`full-screen` *window full-screen?* [Function]

`hide-window` [*window*] [Function]

`iconify-window` [*window*] [Function]

6.4 Overlay Management

6.5 Menu Management

6.6 Callback Registration

`set-button-box-callback` *func* [Function]

`set-current-window` *window* [Function]

`set-dials-callback` *func* [Function]

`set-display-callback` *func* [Function]

`set-entry-callback` *func* [Function]

`set-idle-callback` *func* [Function]

`set-keyboard-callback` *func* [Function]

`set-menu-status-callback` *func* [Function]

`set-motion-callback` *func* [Function]

`set-mouse-callback` *func* [Function]

`set-overlay-display-callback` *func* [Function]

`set-passive-motion-callback` *func* [Function]

`set-reshape-callback` *func* [Function]

`set-spaceball-button-callback` *func* [Function]

`set-spaceball-motion-callback` *func* [Function]

`set-spaceball-rotate-callback` *func* [Function]

`set-special-callback` *func* [Function]

`set-tablet-button-callback` *func* [Function]

`set-tablet-motion-callback` *func* [Function]

`set-visibility-callback` *func* [Function]

`add-timer-callback` *msecs func value* [Function]

6.7 Color Index Colormap Management

6.8 State Retrieval

`window-alpha-size` *window* [Function]

`window-blue-size` *window* [Function]

`window-color-buffer-size` *window* [Function]

`window-colormap-size` *window* [Function]

`window-depth-buffer-size` *window* [Function]

`window-double-buffered?` *window* [Function]

`window-green-size` *window* [Function]

`window-height` *width* [Function]

`window-number-of-children` *window* [Function]

`window-number-of-samples` *window* [Function]

`window-parent` *window* [Function]

`window-position` *window* [Function]

`window-red-size` *window* [Function]

`window-size` *window* [Function]

`window-stencil-buffer-size` *window* [Function]

`window-stereo?` *window* [Function]

`window-rgba` *window* [Function]

`window-width` *width* [Function]

`window-x` *width* [Function]

`window-y` *width* [Function]

`screen-height` [Function]

`screen-height-mm` [Function]

`screen-size` [Function]

`screen-size-mm` [Function]

`screen-width` [Function]

`screen-width-mm` [Function]

`display-mode-possible?` [Function]

`initial-display-mode` [Function]

`initial-window-height` [Function]

`initial-window-position` [Function]

`initial-window-size` [Function]

`initial-window-width` [Function]

`initial-window-x` [Function]

`initial-window-y` [Function]

`elapsed-time` [Function]

6.9 Font Rendering

6.10 Geometric Object Rendering

Appendix A GNU General Public License

Version 3, 29 June 2007

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program—to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

   To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

   An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

   Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   a. The work must carry prominent notices stating that you modified it, and giving a relevant date.

   b. The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

   c. You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

   d. If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

   A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

   You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c. Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d. Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e. Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source.

The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a. Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b. Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c. Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d. Limiting the use for publicity purposes of names of licensors or authors of the material; or

e. Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f. Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance.

However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so

available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) year name of author

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or (at
your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see http://www.gnu.org/licenses/.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
program Copyright (C) year name of author
This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see http://www.gnu.org/licenses/.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read http://www.gnu.org/philosophy/why-not-lgpl.html.

Appendix B GNU Lesser General Public License

Version 3, 29 June 2007

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

0. Additional Definitions.

   As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

   "The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

   An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

   A "Combined Work" is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version".

   The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

   The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL.

   You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions.

   If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

   a. under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or

   b. under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

   The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

   a. Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License.

   b. Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

   You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

   a. Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.

   b. Accompany the Combined Work with a copy of the GNU GPL and this license document.

   c. For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.

   d. Do one of the following:

      0. Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.

      1. Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.

   e. Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

   You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

   a. Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.

   b. Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

   The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

   If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.

Function Index

3

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

V

W