Contents

- OpenGL
  - Legal Information
  - Introduction to OpenGL
    - Primitives and Commands
    - OpenGL Graphic Control
    - Execution Model
    - Basic OpenGL Operation
    - OpenGL Function Names
    - OpenGL Processing Pipeline
      - Vertices
        - Input Data
        - Matrix Transformations
        - Setting Lighting and Coloring
        - Generating Texture Coordinates
        - Assembling Primitives
        - Vertices Reference
      - Primitives
        - Clipping
        - Transforming to Window Coordinates
        - Rasterizing
        - Primitives Reference
      - Fragments
        - Pixel Ownership Test
        - Scissor Test
        - Alpha Test
        - Stencil Test
        - Depth-buffer Test
        - Blending
        - Dithering

- Logical Operations
- Fragments Reference
- Pixels
  - Framebuffer Operations
  - Reading or Copying Pixels
  - Pixels Reference
- Using Evaluators
  - Evaluators Reference
- Performing Selection and Feedback
  - Selection
  - Feedback
  - Selection and Feedback Reference
- Using Display Lists
  - Display Lists Reference
- Managing Modes and Execution
  - Modes and Execution Reference
- Obtaining State Information
  - Using the Query Functions
  - Error Handling
  - OpenGL Error Codes
  - Saving and Restoring Sets of State Variables
  - OpenGL State Variables
  - State Information Reference
- OpenGL Utility Library
  - Initializing
  - Manipulating Images for Use in Texturing
  - Transforming Coordinates
  - Tessellating Polygons
  - Using Callback Functions
  - Using Tessellation Objects
  - Specifying Callbacks
  - Specifying the Polygon to Be Tessellated

- Rendering Simple Surfaces
- Using NURBS Curves and Surfaces
- Handling Errors

Windows Extensions to OpenGL

- OpenGL on Windows
  - Guide To Documentation
  - Components
  - Generic Implementation and Hardware Implementations
    - Limitations
  - Rendering Contexts
    - Rendering Context Functions
  - Pixel Formats
    - Pixel Format Functions
  - Front, Back, and Other Buffers
    - Buffer Functions
  - Fonts and Text
    - Font and Text Functions
  - OpenGL Color Modes and Windows Palette Management
    - Palettes and the Palette Manager
    - Palette Awareness
    - Reading Color Values from the Framebuffer
    - Choosing Between RGBA and Color-Index Mode
    - RGBA Mode and Windows Palette Management
    - Color-Index Mode and Windows Palette Management
  - Overlay, Underlay, and Main Planes
  - Sharing Display Lists
  - Extending OpenGL Functions
  - GLX and WGL/Windows
  - Using OpenGL on Windows
    - Header Files
    - Pixel Format Tasks
      - Choosing and Setting a Best-Match Pixel Format

Examining a Device Context's Current Pixel Format
Examining a Device's Supported Pixel Formats
Rendering Context Tasks
Creating a Rendering Context and Making It Current
Making a Rendering Context Not Current
Deleting a Rendering Context
Drawing with Double Buffers
Drawing Text in a Double-Buffered OpenGL Window
Printing an OpenGL Image
Copying an OpenGL Image to the Clipboard
Multithread OpenGL Drawing Strategies
Using the Auxiliary Library
Reference for Windows Extensions to OpenGL
WGL Functions
Windows Functions
Structures
WGL and Windows Reference
ChoosePixelFormat
DescribePixelFormat
GetEnhMetaFilePixelFormat
GetPixelFormat
SetPixelFormat
SwapBuffers
wglCreateContext
wglCreateLayerContext
wglCopyContext
wglDeleteContext
wglDescribeLayerPlane
wglGetCurrentContext
wglGetCurrentDC
wglGetLayerPaletteEntries
wglGetProcAddress

- wglMakeCurrent
- wglRealizeLayerPalette
- wglSetLayerPaletteEntries
- wglShareLists
- wglSwapLayerBuffers
- wglUseFontBitmaps
- wglUseFontOutlines
- GLYPHMETRICSFLOAT
- LAYERPLANEDESCRIPTOR
- PIXELFORMATDESCRIPTOR
- POINTFLOAT

Programming Tips

- OpenGL Correctness Tips
- OpenGL Performance Tips
- Security Considerations: OpenGL

OpenGL Reference

- State Variables
  - Attribute Groups
  - State Variables for Current Values and Associated Data
  - Transformation State Variables
  - Coloring State Variables
  - Lighting State Variables
  - Rasterization State Variables
  - Texturing State Variables
  - Pixel Operations
  - Framebuffer Control State Variables
  - Pixel State Variables
  - Evaluator State Variables
  - Hint State Variables
  - Implementation-Dependent State Variables
  - Implementation-Dependent Pixel-Depth State Variables
  - Miscellaneous State Variables

GL Functions
- glAccum
- glAddSwapHintRectWIN
- glAlphaFunc
- glAreTexturesResident
- glArrayElement
- glBegin
- glBindTexture
- glBitmap
- glBlendFunc
- glCallList
- glCallLists
- glClear
- glClearAccum
- glClearColor
- glClearDepth
- glClearIndex
- glClearStencil
- glClipPlane
- glColor Functions
  - glColor3b
  - glColor3bv
  - glColor3d
  - glColor3dv
  - glColor3f
  - glColor3fv
  - glColor3i
  - glColor3iv
  - glColor3s
  - glColor3sv
  - glColor3ub
  - glColor3ubv

glColor3ui
glColor3uiv
glColor3us
glColor3usv
glColor4b
glColor4bv
glColor4d
glColor4dv
glColor4f
glColor4fv
glColor4i
glColor4iv
glColor4s
glColor4sv
glColor4ub
glColor4ubv
glColor4ui
glColor4uiv
glColor4us
glColor4usv
glColorMask
glColorMaterial
glColorPointer
glColorTableEXT
glColorSubTableEXT
glCopyPixels
glCopyTexImage1D
glCopyTexImage2D
glCopyTexSubImage1D
glCopyTexSubImage2D
glCullFace
glDeleteLists

glDeleteTextures
glDepthFunc
glDepthMask
glDepthRange
glDisable
glDisableClientState
glDrawArrays
glDrawBuffer
glDrawElements
glDrawPixels
glEdgeFlag Functions
  glEdgeFlag
  glEdgeFlagv
glEdgeFlagPointer
glEnable
glEnableClientState
glEnd
glEndList
glEvalCoord Functions
  glEvalCoord1d
  glEvalCoord1dv
  glEvalCoord1f
  glEvalCoord1fv
  glEvalCoord2d
  glEvalCoord2dv
  glEvalCoord2f
  glEvalCoord2fv
glEvalMesh Functions
  glEvalMesh1
  glEvalMesh2
glEvalPoint Functions
  glEvalPoint1

- glEvalPoint2
- glFeedbackBuffer
- glFinish
- glFlush
- glFog Functions
  - glFogf
  - glFogfv
  - glFogi
  - glFogiv
- glFrontFace
- glFrustum
- glGenLists
- glGenTextures
- glGet
  - glGetBooleanv
  - glGetDoublev
  - glGetFloatv
  - glGetIntegerv
- glGetClipPlane
- glGetColorTableEXT
- glGetColorTableParameterfvEXT
- glGetColorTableParameterivEXT
- glGetError
- glGetLight Functions
  - glGetLightfv
  - glGetLightiv
- glGetMap Functions
  - glGetMapdv
  - glGetMapfv
  - glGetMapiv
- glGetMaterial Functions
  - glGetMaterialfv

glGetMaterialiv
glGetPixelMap Functions
glGetPixelMapfv
glGetPixelMapuiv
glGetPixelMapusv
glGetPointerv
glGetPolygonStipple
glGetString
glGetTexEnv Functions
glGetTexEnvfv
glGetTexEnviv
glGetTexGen Functions
glGetTexGendv
glGetTexGenfv
glGetTexGeniv
glGetTexImage
glGetTexLevelParameter Functions
glGetTexLevelParameterfv
glGetTexLevelParameteriv
glGetTexParameter Functions
glGetTexParameterfv
glGetTexParameteriv
glHint
glIndex Functions
glIndexd
glIndexdv
glIndexf
glIndexfv
glIndexi
glIndexiv
glIndexs
glIndexsv

glIndexMask
glIndexPointer
glInitNames
glInterleavedArrays
glIsEnabled
glIsList
glIsTexture
glLight Functions
  glLightf
  glLightfv
  glLighti
  glLightiv
glLightModel Functions
  glLightModelf
  glLightModelfv
  glLightModeli
  glLightModeliv
glLineStipple
glLineWidth
glListBase
glLoadIdentity
glLoadMatrix Functions
  glLoadMatrixd
  glLoadMatrixf
glLoadName
glLogicOp
glMap1 Functions
  glMap1d
  glMap1f
glMap2 Functions
  glMap2d
  glMap2f

glMapGrid Functions
  glMapGrid1d
  glMapGrid1f
  glMapGrid2d
  glMapGrid2f
glMaterial Functions
  glMaterialf
  glMaterialfv
  glMateriali
  glMaterialiv
glMatrixMode
glMultMatrix Functions
  glMultMatrixd
  glMultMatrixf
glNewList
glNormal Functions
  glNormal3b
  glNormal3bv
  glNormal3d
  glNormal3dv
  glNormal3f
  glNormal3fv
  glNormal3i
  glNormal3iv
  glNormal3s
  glNormal3sv
glNormalPointer
glOrtho
glPassThrough
glPixelMap Functions
  glPixelMapfv
  glPixelMapuiv

glPixelMapusv
glPixelStore Functions
glPixelStoref
glPixelStorei
glPixelTransfer Functions
glPixelTransferf
glPixelTransferi
glPixelZoom
glPointSize
glPolygonMode
glPolygonOffset
glPolygonStipple
glPopAttrib
glPopClientAttrib
glPopMatrix
glPopName
glPrioritizeTextures
glPushAttrib
glPushClientAttrib
glPushMatrix
glPushName
glRasterPos Functions
glRasterPos2d
glRasterPos2dv
glRasterPos2f
glRasterPos2fv
glRasterPos2i
glRasterPos2iv
glRasterPos2s
glRasterPos2sv
glRasterPos3d
glRasterPos3dv

glRasterPos3f
glRasterPos3fv
glRasterPos3i
glRasterPos3iv
glRasterPos3s
glRasterPos3sv
glRasterPos4d
glRasterPos4dv
glRasterPos4f
glRasterPos4fv
glRasterPos4i
glRasterPos4iv
glRasterPos4s
glRasterPos4sv
glReadBuffer
glReadPixels
glRect Functions
glRectd
glRectdv
glRectf
glRectfv
glRecti
glRectiv
glRects
glRectsv
glRenderMode
glRotate Functions
glRotated
glRotatef
glScale Functions
glScaled
glScalef

glScissor
glSelectBuffer
glShadeModel
glStencilFunc
glStencilMask
glStencilOp
glTexCoord Functions
  glTexCoord1d
  glTexCoord1dv
  glTexCoord1f
  glTexCoord1fv
  glTexCoord1i
  glTexCoord1iv
  glTexCoord1s
  glTexCoord1sv
  glTexCoord2d
  glTexCoord2dv
  glTexCoord2fv
  glTexCoord2i
  glTexCoord2iv
  glTexCoord2s
  glTexCoord2sv
  glTexCoord3d
  glTexCoord3dv
  glTexCoord3f
  glTexCoord3fv
  glTexCoord3i
  glTexCoord3iv
  glTexCoord3s
  glTexCoord3sv
  glTexCoord4d
  glTexCoord4dv

glTexCoord4f
glTexCoord4fv
glTexCoord4i
glTexCoord4iv
glTexCoord4s
glTexCoord4sv
glTexCoordPointer
glTexEnv Functions
glTexEnvf
glTexEnvfv
glTexEnvi
glTexEnviv
glTexGen Functions
glTexGend
glTexGendv
glTexGenf
glTexGenfv
glTexGeni
glTexGeniv
glTexImage1D
glTexImage2D
glTexParameter Functions
glTexParameterf
glTexParameterfv
glTexParameteri
glTexParameteriv
glTexSubImage1D
glTexSubImage2D
glTranslate Functions
glTranslated
glTranslatef
glVertex Functions

glVertex2d
glVertex2dv
glVertex2f
glVertex2fv
glVertex2i
glVertex2iv
glVertex2s
glVertex2sv
glVertex3d
glVertex3dv
glVertex3f
glVertex3fv
glVertex3i
glVertex3iv
glVertex3s
glVertex3sv
glVertex4d
glVertex4dv
glVertex4f
glVertex4fv
glVertex4i
glVertex4iv
glVertex4s
glVertex4sv
glVertexPointer
glViewport
GLU Functions
gluBeginCurve
gluBeginPolygon
gluBeginSurface
gluBeginTrim
gluBuild1DMipmaps

gluBuild2DMipmaps

gluCylinder

gluDeleteNurbsRenderer

gluDeleteQuadric

gluDeleteTess

gluDisk

gluEndCurve

gluEndPolygon

gluEndSurface

gluEndTrim

gluErrorString

gluGetNurbsProperty

gluGetString

gluGetTessProperty

gluLoadSamplingMatrices

gluLookAt

gluNewNurbsRenderer

gluNewQuadric

gluNewTess

gluNextContour

gluNurbsCallback

gluNurbsCurve

gluNurbsProperty

gluNurbsSurface

gluOrtho2D

gluPartialDisk

gluPerspective

gluPickMatrix

gluProject

gluPwlCurve

gluQuadricCallback

gluQuadricDrawStyle

gluQuadricNormals
gluQuadricOrientation
gluQuadricTexture
gluScaleImage
gluSphere
gluTessBeginContour
gluTessBeginPolygon
gluTessCallback
gluTessEndContour
gluTessEndPolygon
gluTessNormal
gluTessProperty
gluTessVertex
gluUnProject
Porting to OpenGL
Introduction to Porting to OpenGL for Windows
Porting X Window System Applications
Translating the GLX library
Porting Device Contexts and Pixel Formats
GLX Pixel Format Code Sample
Windows Pixel Format Code Sample
Porting Rendering Contexts
GLX Rendering Context Code Sample
Windows Rendering Context Code Sample
Porting GLX Pixmap Code
Porting Other GLX Code
A Porting Sample
An X Window System OpenGL Program
The Program Ported to Windows
Porting Applications from IRIS GL
Special IRIS GL Porting Issues
Porting greset

Porting IRIS GL "Get" Functions
Port code that needs a current graphics position
  Porting Screen and Buffer Clearing Commands
  Porting Matrix and Transformation Functions
  Porting MSINGLE Mode Code
  Porting Functions that Get Matrices and Transformations
  Porting Viewports, Screenmasks, and Scrboxes
  Porting Clipping Planes
Porting Drawing Functions
  The IRIS GL Sphere Library
  Porting v Functions
  Porting bgn/end Commands
  Porting Points
  Porting Lines
  Porting Polygons and Quadrilaterals
  Porting Triangles
  Porting Arcs and Circles
  Porting Spheres
Porting Color, Shading, and Writemask Code
  Porting Color Calls
  Porting Shading Models
Porting Pixel Operations
Porting Depth Cueing and Fog Commands
Porting Curve and Surface Functions
  Porting NURBS Objects
  Porting NURBS Curves
  Porting Trimming Curves
  Porting NURBS Surfaces
Porting Antialiasing Functions
  Porting Blending Code
  Porting afunction Test Functions
  Using Antialiasing Functions

- Porting Accumulation Buffer Calls
- Porting Stencil Plane Calls
- Porting Display Lists
  - Porting the bbox2 Function
  - Porting Edited Display Lists
  - A Sample Port of a Display List
- Porting Defs, Binds, and Sets
- Porting Lighting and Materials Functions
- Porting Texture Functions
  - Translating tevdef
  - Translating texdef
  - Translating texgen
- Porting Picking Functions
- Porting Feedback Functions

OpenGL Functions and Their IRIS GL Equivalents

IRIS GL and OpenGL Differences

OpenGL Glossary

A

B

C

D

E

F

G

H

I

J K

L

M

N

O

P

Q
R
S
T
U V
W
X Y Z
Appendix

OpenGL

2/22/2020 • 2 minutes to read • Edit Online

Purpose

As a software interface for graphics hardware, OpenGL renders multidimensional objects into a framebuffer. The Microsoft implementation of OpenGL for the Windows operating system is industry-standard graphics software with which programmers can create high-quality still and animated three-dimensional color images. The version of OpenGL described in this section is 1.1.

For information about OpenGL ES running on Windows, see ANGLE for Windows Store.

Where applicable

OpenGL is built for compatibility across hardware and operating systems. This architecture makes it easy to port OpenGL programs from one system to another. While each operating system has unique requirements, the OpenGL code in many programs can be used as is.

Developer audience

Designed for use by C/C++ programmers, OpenGL requires familiarity with the Windows graphical user interface as well as message-driven architecture.

Run-time requirements

For more information on which operating systems are required for a particular function, see the Requirements section of the documentation for the function.

In this section

TOPIC	DESCRIPTION
Overview	General information about how OpenGL works.
Reference	Documentation of functions, structures, and other programming elements.
Samples	Examples of OpenGL code.

Related topics

D
i
r
e
c
t
X
G

r
a
p
h
i
c
s
a
n
d
G
a
m
i
n
g

S
t
i
l
l
l
m
a
g
e

W
i
n
d
o
w
s
C
o
l
o
r
S
y
s
t
e
m
(
W
C
S
)

W

i
n
d
o
w
s
G
D
I

W
i
n
d
o
w
s
I
m
a
g
e
A
c
q
u
i
s
i
t
i
o
n

W
i
n
d
o
w
s
M
u
l
t
i
m
e
d
i
a

W

i
n
d
o
w
s
A
P
I

Legal Information

2/22/2020 • 2 minutes to read • Edit Online

OpenGL Programmer's Reference

Information in this document, including URL and other Internet Web site references, is subject to change without notice. Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Microsoft, MS-DOS, Windows, Windows NT, and Win32 are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

TrueType is a registered trademark of Apple Computer, Inc.

UNIX is a registered trademark of the Open Group.

Portions Silicon Graphics, Inc. from the *OpenGL Programming Guide* and the *OpenGL Reference Manual*. Reprinted with permission of Silicon Graphics, Inc.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Introduction to OpenGL

2/22/2020 • 2 minutes to read • Edit Online

As a software interface for graphics hardware, the main purpose of OpenGL is to render two- and three-dimensional objects into a framebuffer. These objects are described as sequences of vertices (that define geometric objects) or pixels (that define images). OpenGL performs several processes on this data to convert it to pixels to form the final desired image in the framebuffer.

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

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

Related topics

OpenGL Processing Pipeline

Using

g
E
v
a
l
u
a
t
o
r
s

P
e
r
f
o
r
m
i
n
g
S
e
l
e
c
t
i
o
n
a
n
d
F
e
e
d
b
a
c
k

U
s
i
n
g
D
i
s
p

I
a
y
L
i
s
t
s

M
a
n
a
g
i
n
g
M
o
d
e
s
a
n
d
E
x
e
c
u
t
i
o
n

O
b
t
a
i
n
i
n
g
S
t
a
t
e
I
n
f

o
r
m
a
t
i
o
n

O
p
e
n
G
L
U
t
i
l
i
t
y
L
i
b
r
a
r
y

Primitives and Commands

2/22/2020 • 2 minutes to read • Edit Online

OpenGL draws *primitives* points, line segments, or polygonssubject to several selectable modes. You can control modes independently of one another. That is, setting one mode doesn't affect whether other modes are set (although many modes may interact to determine what eventually ends up in the framebuffer). To specify primitives, set modes, and perform other OpenGL operations, you issue commands in the form of function calls.

Primitives are defined by a group of one or more *vertices*. A vertex defines a point, an endpoint of a line, or a corner of a polygon where two edges meet. Data (consisting of vertex coordinates, colors, normals, texture coordinates, and edge flags) is associated with a vertex, and each vertex and its associated data are processed independently, in order, and in the same way. The only exceptions to this rule are cases in which the group of vertices must be clipped so that a particular primitive fits within a specified region. In this case, vertex data may be modified and new vertices created. The type of clipping depends on which primitive the group of vertices represents.

Commands are always processed in the order in which they are received, although there may be an indeterminate delay before a command takes effect. This means that each primitive is drawn completely before any subsequent command takes effect. It also means that state-querying commands return data that is consistent with complete execution of all previously issued OpenGL commands.

OpenGL Graphic Control

2/22/2020 • 2 minutes to read • Edit Online

OpenGL provides you with fairly direct control over the fundamental operations of two- and three-dimensional graphics. This includes specification of such parameters as transformation matrices, lighting equation coefficients, antialiasing methods, and pixel-update operators. However, it doesn't provide you with a means for describing or modeling complex geometric objects. Thus, the OpenGL commands you issue specify how a certain result should be produced (what procedure should be followed) rather than what exactly that result should look like. That is, OpenGL is fundamentally procedural rather than descriptive. To fully understand how to use OpenGL, it helps to know the order in which it carries out its operations.

Execution Model

2/22/2020 • 2 minutes to read • Edit Online

The model for interpretation of OpenGL commands is client/server. Application code (the client) issues commands, which are interpreted and processed by OpenGL (the server). The server may or may not operate on the same computer as the client. In this sense, OpenGL is network-transparent. A server can maintain several OpenGL contexts, each of which is an encapsulated OpenGL state. A client can connect to any one of these contexts. The required network protocol can be implemented by augmenting an already existing protocol (such as that of the X Window System) or by using an independent protocol. No OpenGL commands are provided for obtaining user input.

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

- Determines which portions of the framebuffer OpenGL may access at any given time.
- Communicates to OpenGL how those portions are structured.

Therefore, there are no OpenGL commands to configure the framebuffer or initialize OpenGL. Frame buffer configuration is done outside of OpenGL in conjunction with the window system; OpenGL initialization takes place when the window system allocates a window for OpenGL rendering.

Basic OpenGL Operation

2/22/2020 • 2 minutes to read • Edit Online

The following diagram illustrates how OpenGL processes data. As shown, commands enter from the left and proceed through a processing pipeline. Some commands specify geometric objects to be drawn, and others control how the objects are handled during various processing stages.

The processing stages in basic OpenGL operation are as follows:

- **Display list** Rather than having all commands proceed immediately through the pipeline, you can choose to accumulate some of them in a display list for processing later.
- **Evaluator** The evaluator stage of processing provides an efficient way to approximate curve and surface geometry by evaluating polynomial commands of input values.
- **Per-vertex operations and primitive assembly** OpenGL processes geometric primitivespoints, line segments, and polygonsall of which are described by vertices. Vertices are transformed and lit, and primitives are clipped to the viewport in preparation for rasterization.
- **Rasterization** The rasterization stage produces a series of frame-buffer addresses and associated values using a two-dimensional description of a point, line segment, or polygon. Each fragment so produced is fed into the last stage, per-fragment operations.
- **Per-fragment operations** These are the final operations performed on the data before it is stored as pixels in the framebuffer.

  Per-fragment operations include conditional updates to the framebuffer based on incoming and previously stored z values (for z buffering) and blending of incoming pixel colors with stored colors, as well as masking and other logical operations on pixel values.

Data can be input in the form of pixels rather than vertices. Data in the form of pixels, such as might describe an image for use in texture mapping, skips the first stage of processing described above and instead is processed as pixels, in the pixel operations stage. Following pixel operations, the pixel data is either:

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

OpenGL Function Names

2/22/2020 • 2 minutes to read • Edit Online

Many OpenGL functions are variations of each other, differing mostly in the data types of their arguments. Some functions differ in the number of related arguments and whether those arguments can be specified as a vector or must be specified separately in a list. For example, if you use the **glVertex2f** function, you need to supply x- and y-coordinates as 32-bit floating-point numbers; with **glVertex3sv**, you must supply an array of three short (16-bit) integer values for x, y, and z. Only the base name of the function is used in the topics that follow. An asterisk indicates that there may be more to the actual function name than is shown. For example, glVertex* stands for all the variations of the function you use to specify vertices: **glVertex2d**, **glVertex2f**, **glVertex2i**, and so on.

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

OpenGL Processing Pipeline

2/22/2020 • 2 minutes to read • Edit Online

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

The following diagram details the OpenGL processing pipeline. For most of the pipeline, you can see three vertical arrows between the major stages. These arrows represent vertices and the two primary types of data that can be associated with vertices: color values and texture coordinates. Also note that vertices are assembled into primitives, then into fragments, and finally into pixels in the framebuffer. This progression is discussed in more detail in Vertices, Primitives, Fragments, and Pixels.

Vertices

2/22/2020 • 2 minutes to read • Edit Online

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

- Input Data
- Matrix Transformations
- Setting Lighting and Coloring
- Generating Texture Coordinates
- Assembling Primitives
- Vertices Reference

Input Data

2/22/2020 • 2 minutes to read • Edit Online

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

- **Vertices**. Vertices describe the shape of the desired geometric object. To specify vertices, use **glVertex*** functions in conjunction with **glBegin** and **glEnd** to create a point, line, or polygon. You can also use **glRect** to describe an entire rectangle at one time.
- **Edge flag**. By default, all edges of polygons are boundary edges. Use **glEdgeFlag*** to explicitly set the edge flag.
- **Current raster position**. Specified with **glRasterPos***, the current raster position is used to determine raster coordinates for pixel- and bitmap-drawing operations.
- **Current normal**. A normal vector associated with a particular vertex determines how a surface at that vertex is oriented in three-dimensional space; this in turn affects how much light that particular vertex receives. Use **glNormal*** to specify a normal vector.
- **Current color**. The color of a vertex, together with the lighting conditions, determine the final, lit color. Color is specified with **glColor*** if in RGBA mode, or with **glIndex*** if in color-index mode.
- **Current texture coordinates**. Specified with **glTexCoord***, texture coordinates determine the location in a texture map to associate with a vertex of an object.

> **NOTE**
>
> When **glVertex*** is called, the resulting vertex inherits the current edge flag, normal, color, and texture coordinates. Therefore, **glEdgeFlag***, **glNormal***, **glColor***, and **glTexCoord*** must be called before **glVertex***, if they are to affect the resulting vertex.

Matrix Transformations

2/22/2020 • 2 minutes to read • Edit Online

Vertices and normals are transformed by the modelview and projection matrices before they're used to produce an image in the framebuffer. Use functions such as **glMatrixMode**, **glMultMatrix***, **glRotate***, **glTranslate***, and **glScale*** to compose the desired transformations. Or specify matrices directly with **glLoadMatrix*** and **glLoadIdentity**. Use **glPushMatrix** and **glPopMatrix** to save and restore modelview and projection matrices on their respective stacks.

Setting Lighting and Coloring

2/22/2020 • 2 minutes to read • Edit Online

In addition to specifying colors and normal vectors, you can define the desired lighting conditions with **glLight***
and **glLightModel***, and the desired material properties with **glMaterial***. Related functions for controlling how
lighting calculations are performed include **glShadeModel**, **glFrontFace**, and **glColorMaterial**.

Generating Texture Coordinates

2/22/2020 • 2 minutes to read • Edit Online

Rather than explicitly supplying texture coordinates, you can have OpenGL generate them as a function of other vertex data using glTexGen*. After the texture coordinates have been specified or generated, they are transformed by the texture matrix. This matrix is controlled with the same functions that are used for matrix transformations (see Matrix Transformations).

Assembling Primitives

2/22/2020 • 2 minutes to read • Edit Online

Once all necessary calculations have been performed, vertices are assembled into primitives points, line segments, or polygons together with the relevant edge flag, color, and texture information for each vertex.

Vertices Reference

2/22/2020 • 2 minutes to read • Edit Online

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

Primitives

2/22/2020 • 2 minutes to read • Edit Online

During the next stage of processing, primitives are converted to pixel fragments in the following steps:

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

For more information, see the following topics:

- Clipping
- Transforming to Window Coordinates
- Rasterizing
- Primitives Reference

Clipping

2/22/2020 • 2 minutes to read • Edit Online

Clipping occurs in two steps:

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

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

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

Transforming to Window Coordinates

2/22/2020 • 2 minutes to read • Edit Online

Before clip coordinates are converted to window coordinates, they are divided by the value of *w* to yield normalized device coordinates. The viewport transformation applied to these normalized coordinates produces window coordinates. You control the viewport, which determines the area of the on-screen window that displays an image, with **glDepthRange** and **glViewport**.

Rasterizing

2/22/2020 • 2 minutes to read • Edit Online

Rasterizing is the process by which a primitive is converted to a two-dimensional image. Each point of this image contains such information as color, depth, and texture data. A point and its associated information are called a *fragment*. The current raster position, as specified with **glRasterPos***, is used in various ways during this stage for drawing pixels and bitmaps. Different issues arise when rasterizing points, line segments, and polygons. In addition, pixel rectangles and bitmaps need to be rasterized.

With OpenGL you control rasterizing by using the following functions:

- Primitives. Control how primitives are rasterized using functions that determine dimensions and stipple patterns: **glPointSize**, **glLineWidth**, **glLineStipple**, and **glPolygonStipple**. Control how the front and back faces of polygons are rasterized with **glCullFace**, **glFrontFace**, and **glPolygonMode**.
- Pixels. Several functions control pixel storage and transfer modes. The function **glPixelStore*** controls the encoding of pixels in client memory, and **glPixelTransfer*** and **glPixelMap*** control how pixels are processed before being placed in the framebuffer. Specify a pixel rectangle with **glDrawPixels**; control its rasterization with **glPixelZoom**.
- Bitmaps. Bitmaps are rectangles of zeros and ones specifying a particular pattern of fragments to be produced. Each of these fragments has the same associated data. The **glBitmap** function specifies a bitmap.
- Texture Memory. When texturing is enabled, texturing maps a portion of a specified texture image onto each primitive. This mapping is accomplished by using the color of the texture image at the location indicated by a fragment's texture coordinates to modify the fragment's RGBA color. Specify a texture image with **glTexImage2D** or **glTexImage1D**. The **glTexParameter*** and **glTexEnv*** functions control how texture values are interpreted and applied to a fragment.
- Fog. To blend a fog color with a rasterized fragment's post-texturing color, use a blending factor that depends on the distance between the eyepoint and the fragment. Use **glFog*** to specify the fog color and blending factor.

Primitives Reference

2/22/2020 • 2 minutes to read • Edit Online

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

Fragments

2/22/2020 • 2 minutes to read • Edit Online

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

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

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

- Blending
- Dithering
- Logical operations

Pixel Ownership Test

2/22/2020 • 2 minutes to read • Edit Online

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

Scissor Test

2/22/2020 • 2 minutes to read • Edit Online

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

Alpha Test

2/22/2020 • 2 minutes to read • Edit Online

The alpha test (performed only in RGBA mode) discards a fragment, depending on the outcome of a comparison between the fragment's alpha value and a constant reference value. You specify the comparison function and reference value with **glAlphaFunc**.

Stencil Test

2/22/2020 • 2 minutes to read • Edit Online

The stencil test conditionally discards a fragment based on the outcome of a comparison between the value in the stencil buffer and a reference value. The **glStencilFunc** function specifies the comparison function and the reference value. Whether the fragment passes or fails the stencil test, the value in the stencil buffer is modified according to the instructions specified with **glStencilOp**.

Depth-buffer Test

2/22/2020 • 2 minutes to read • Edit Online

The depth-buffer test discards a fragment if a depth comparison fails; **glDepthFunc** specifies the comparison function. If stenciling is enabled, the result of the depth comparison also affects the stencil buffer update value.

Blending

2/22/2020 • 2 minutes to read • Edit Online

Blending combines a fragment's R, G, B, and A values with those stored in the framebuffer at the corresponding location. The blending, which is performed only in RGBA mode, depends on the alpha value of the fragment and that of the corresponding currently stored pixel; it may also depend on the RGB values. You control blending with **glBlendFunc**, with which you indicate the source and destination blending factors.

Dithering

2/22/2020 • 2 minutes to read • Edit Online

If dithering is enabled, a dithering algorithm is applied to the fragment's color or color-index value. This algorithm depends only on the fragment's value and its x and y window coordinates.

Logical Operations

2/22/2020 • 2 minutes to read • Edit Online

A logical operation can be applied between the fragment and the value stored at the corresponding location in the framebuffer; the result replaces the current framebuffer value. You choose the desired logical operation with **glLogicOp**. Logical operations are performed only on color indexes, never on RGBA values.

Fragments Reference

2/22/2020 • 2 minutes to read • Edit Online

- glAlphaFunc
- glBlendFunc
- glDepthFunc
- glLogicOp
- glScissor
- glStencilFunc
- glStencilOp

Pixels

2/22/2020 • 2 minutes to read • Edit Online

Fragments are converted to pixels in the framebuffer. The framebuffer is organized into a set of logical buffers the color, depth, stencil, and accumulation buffers. The color buffer itself consists of a front left, front right, back left, back right, and some number of auxiliary buffers. You can issue functions to control these buffers, and directly read or copy pixels from them. (Note that the particular OpenGL context you're using may not provide all these buffers.)

- Framebuffer Operations
- Reading or Copying Pixels
- Pixels Reference

Framebuffer Operations

2/22/2020 • 2 minutes to read • Edit Online

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

- glIndexMask
- glColorMask
- glDepthMask
- glStencilMask

The glAccum function controls the operation of the accumulation buffer. Finally, glClear sets every pixel in a specified subset of the buffers to the value specified with glClearColor, glClearIndex, glClearDepth, glClearStencil, or glClearAccum.

Reading or Copying Pixels

2/22/2020 • 2 minutes to read • Edit Online

You can read pixels from the framebuffer into memory, encode them in various ways, and store the encoded result in memory with **glReadPixels**. In addition, you can copy a rectangle of pixel values from one region of the framebuffer to another with **glCopyPixels**. The function **glReadBuffer** controls which color buffer the pixels are read or copied from.

Pixels Reference

2/22/2020 • 2 minutes to read • Edit Online

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

Using Evaluators

2/22/2020 • 2 minutes to read • Edit Online

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

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

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

Related topics

Evaluators Reference

Evaluators Reference

2/22/2020 • 2 minutes to read • Edit Online

- glEvalCoord
- glEvalMesh
- glEvalPoint
- glMap1
- glMap2
- glMapGrid

Performing Selection and Feedback

2/22/2020 • 2 minutes to read • Edit Online

Selection, feedback, and rendering are mutually exclusive modes of operation. Rendering is the normal, default mode during which fragments are produced by rasterization.

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

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

- Selection
- Feedback
- Selection and Feedback Reference

Selection

2/22/2020 • 2 minutes to read • Edit Online

Selection returns the current contents of the name stack, which is an array of names with integer values. You assign the names and build the name stack within the modeling code that specifies the geometry of objects you want to draw. Then, in selection mode, whenever a primitive intersects the clip volume, a selection hit occurs. The hit record, which is written into the selection array you've supplied with **glSelectBuffer**, contains information about the contents of the name stack at the time of the hit.

> **NOTE**
>
> Call **glSelectBuffer** before you put OpenGL into selection mode with **glRenderMode**. The entire contents of the name stack aren't guaranteed to be returned until you call **glRenderMode** to take OpenGL out of selection mode.

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

Feedback

2/22/2020 • 2 minutes to read • Edit Online

In feedback mode, each primitive to be rasterized generates a block of values that is copied into the feedback array. Supply this array with **glFeedbackBuffer**, which you must call before putting OpenGL into feedback mode. Each block of values begins with a code indicating the primitive type, followed by values that describe the primitive's vertices and associated data. Entries are also written for bitmaps and pixel rectangles. Values are not guaranteed to be written into the feedback array until you call **glRenderMode** to take OpenGL out of feedback mode. You can use **glPassThrough** to supply a marker that is returned in feedback mode as if it were a primitive.

Selection and Feedback Reference

2/22/2020 • 2 minutes to read • Edit Online

Selection and Feedback

- glRenderMode

Selection

- glInitNames
- glLoadName
- glPopName
- glPushName
- glSelectBuffer
- gluPickMatrix

Feedback

- glFeedbackBuffer
- glPassThrough

Using Display Lists

2/22/2020 • 2 minutes to read • Edit Online

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

Related topics

D
i
s
p
l
a
y
L
i
s
t
s
R
e
f
e
r
e
n
c
e

Display Lists Reference

2/22/2020 • 2 minutes to read • Edit Online

- glCallList
- glCallLists
- glDeleteLists
- glEndList
- glGenLists
- glIsList
- glListBase
- glNewList

Managing Modes and Execution

2/22/2020 • 2 minutes to read • Edit Online

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

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

In a particular implementation of OpenGL, you may be able to control certain behaviors with hints by using **glHint**. Such behaviors are the quality of color and texture coordinate interpolation; the accuracy of fog calculations; and the sampling quality of antialiased points, lines, or polygons.

Related topics

Modes and Execution Reference

Modes and Execution Reference

2/22/2020 • 2 minutes to read • Edit Online

- glDisable
- glEnable
- glFinish
- glFlush
- glHint
- glIsEnabled

Obtaining State Information

2/22/2020 • 2 minutes to read • Edit Online

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

glGetClipPlane	glGetPixelMap	glGetTexImage
glGetLight	glGetPolygonStipple	glGetTexLevelParameter
glGetMap	glGetTexEnv	glGetTexParameter
glGetMaterial	glGetTexGen	

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

- Using the Query Functions
- Error Handling
- OpenGL Error Codes
- Saving and Restoring Sets of State Variables
- OpenGL State Variables
- State Information Reference

Using the Query Functions

2/22/2020 • 2 minutes to read • Edit Online

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

- **glGetBooleanv**
- **glGetIntegerv**
- **glGetFloatv**
- **glGetDoublev**
- **glIsEnabled**

The prototypes for the query functions are:

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

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

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

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

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

The prototype for **glIsEnabled** is:

GLboolean glIsEnabled(GLenum *cap*);

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

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

The functions that return specific state variables are:

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

- glGetTexParameter

Error Handling

2/22/2020 • 2 minutes to read • Edit Online

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

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

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

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

OpenGL Error Codes

2/22/2020 • 2 minutes to read • Edit Online

OpenGL includes the following error codes.

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

Saving and Restoring Sets of State Variables

2/22/2020 • 2 minutes to read • Edit Online

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

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

The prototype of **glPushAttrib** is:

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

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

OpenGL State Variables

2/22/2020 • 2 minutes to read • Edit Online

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

- State Variables for Current Values and Associated Data
- Transformation State Variables
- Coloring State Variables
- Lighting State Variables
- Rasterization State Variables
- Texturing State Variables
- Pixel Operations
- Framebuffer Control State Variables
- Pixel State Variables
- Evaluator State Variables
- Hint State Variables
- Implementation-Dependent State Variables
- Implementation-Dependent Pixel-Depth State Variables
- Miscellaneous State Variables

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

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

OpenGL Reference Manual.

State Information Reference

2/22/2020 • 2 minutes to read • Edit Online

The following list contains the functions that get state information.

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

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

OpenGL Utility Library

2/22/2020 • 2 minutes to read • Edit Online

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

> **NOTE**
>
> The prefix for Utility library functions is "glu" rather than "gl."

Many of these functions are described in preceding sections of the documenation as their topics arise. These functions are listed here for completeness. GLU functions that aren't discussed in preceding sections are described here. For more detailed descriptions of these functions, see the *OpenGL Reference Manual*. This section groups related GLU functions, as follows:

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

Initializing

2/22/2020 • 2 minutes to read • Edit Online

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

The prototype of **gluGetString** is:

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

Manipulating Images for Use in Texturing

2/22/2020 • 2 minutes to read • Edit Online

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

Transforming Coordinates

2/22/2020 • 2 minutes to read • Edit Online

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

The **gluPickMatrix** function simplifies selection of a picking matrix by creating a matrix that restricts drawing to a small region of the viewport. If you re-render the scene in selection mode after this matrix has been applied, all objects that would be drawn near the cursor will be selected, and information about them will be stored in the selection buffer. For more information about selection mode, see "Performing Selection and Feedback" Performing Selection and Feedback.

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

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

Tessellating Polygons

2/22/2020 • 2 minutes to read • Edit Online

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

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

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

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

To use polygon tessellation

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

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

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

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

Using Callback Functions

2/22/2020 • 2 minutes to read • Edit Online

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

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

Using Tessellation Objects

2/22/2020 • 2 minutes to read • Edit Online

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

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

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

Specifying Callbacks

2/22/2020 • 2 minutes to read • Edit Online

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

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

CALLBACK FUNCTION	PROTOTYPE
GLU_BEGIN	**void begin**(**GLenum** *type*);
GLU_EDGE_FLAG	**void edgeFlag**(**GLboolean** *flag*);
GLU_VERTEX	**void vertex**(**void ***data*);
GLU_END	**void end**(*void*);
GLU_ERROR	**void error**(**GLenum** *errno*);

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

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

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

- GL_TRIANGLE_FAN
- GL_TRIANGLE_STRIP
- GL_TRIANGLES

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

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

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

Specifying the Polygon to Be Tessellated

2/22/2020 • 2 minutes to read • Edit Online

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

- **gluBeginPolygon**
- **gluTessVertex**
- **gluNextContour**
- **gluEndPolygon**

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

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

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

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

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

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

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

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

Rendering Simple Surfaces

2/22/2020 • 2 minutes to read • Edit Online

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

To render simple surfaces

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

   To destroy this object when you're finished with it, use **gluDeleteQuadric**.

2. Specify the desired rendering style, as listed below, with the appropriate function (unless you're satisfied with the default values):

   - Whether surface normals should be generated, and if so, whether there should be one normal per vertex or one normal per face: **gluQuadricNormals**
   - Whether texture coordinates should be generated: **gluQuadricTexture**
   - Which side of the quadric should be considered the outside and which the inside: **gluQuadricOrientation**
   - Whether the quadric should be drawn as a set of polygons, lines, or points: **gluQuadricDrawStyle**

3. After specifying the rendering style, invoke the rendering function for the desired type of quadric object: **gluSphere**, **gluCylinder**, **gluDisk**, or **gluPartialDisk**.

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

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

Using NURBS Curves and Surfaces

2/22/2020 • 2 minutes to read • Edit Online

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

To manage a NURBS object, use:

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

To specify the desired curves, use:

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

To specify the desired surfaces, use:

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

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

To specify the trimming region, use:

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

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

- Whether to discard a curve or surface whose control polyhedron lies outside the current viewport.
- The maximum length (in pixels) of edges of polygons used to render curves and surfaces.
- Whether you will take the projection matrix, modelview matrix, and viewport from the OpenGL server or supply them explictly with **gluLoadSamplingMatrices**.

Use **gluNurbsProperty** to set these properties, or use the default values. You can query a NURBS object about its rendering style with **gluGetNurbsProperty**.

Handling Errors

2/22/2020 • 2 minutes to read • Edit Online

The **gluErrorString** function retrieves error strings that correspond to OpenGL or GLU error codes. The currently defined OpenGL error codes are described in **glGetError**. The GLU error codes are listed in **gluErrorString**, *gluTessCallback*, *gluQuadricCallback*, and *gluNurbsCallback*.

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

Windows Extensions to OpenGL

2/22/2020 • 2 minutes to read • Edit Online

The following sections describe extentions to OpenGL:

- OpenGL on Windows
- WGL and Windows Reference

OpenGL on Windows

2/22/2020 • 2 minutes to read • Edit Online

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

For information about OpenGL ES running on Windows, see MSOpenTech ANGLE.

- Guide To Documentation
- Components
- Generic Implementation and Hardware Implementations
- Rendering Contexts
- Pixel Formats
- Front, Back, and Other Buffers
- Fonts and Text
- OpenGL Color Modes and Windows Palette Management
- Overlay, Underlay, and Main Planes
- Sharing Display Lists
- Extending OpenGL Functions
- GLX and WGL/Windows
- Using OpenGL on Windows
- Reference for Windows Extensions to OpenGL

Guide To Documentation

2/22/2020 • 2 minutes to read • Edit Online

The documentation set for OpenGL in Windows includes five elements.

- The *OpenGL Reference Manual* includes an overview of how OpenGL works and a set of detailed reference pages. The reference pages cover all the 115 distinct OpenGL functions, as well as the 43 functions in the OpenGL Utility (GLU) library.
- The *OpenGL Programming Guide* explains how to create graphics programs using OpenGL. It includes discussions of the following major topics:
  - Drawing geometric shapes
  - Pixels, bitmaps, fonts, and images
  - Viewing and matrix transformations
  - Texture mapping
  - Display lists
  - Advanced composite techniques
  - Color
  - Evaluators and NURBS
  - Lighting
  - Selection and feedback
  - Blending, antialiasing, and fog
  - Advanced techniques

  In addition, the *OpenGL Programming Guide* contains appendixes that discuss the OpenGL Utility library and the OpenGL Programming Guide Auxiliary library.
- The third documentation element is this overview. It describes the Windows implementation of OpenGL and provides an overview of its components. It discusses the concepts of rendering contexts, pixel formats, and buffers; the WGL functions that connect OpenGL to the Windows windowing systems; and the Windows functions that support per-window pixel formats and double buffering of windows for OpenGL graphics windows. The WGL functions and the functions are specific to the Windows implementation of OpenGL.
- The fourth documentation element is the set of reference pages for the WGL functions, the Windows functions just mentioned, and the PIXELFORMATDESCRIPTOR data structure.
- The fifth documentation element is the porting guide. It discusses moving existing OpenGL code from other environments to Windows.

The first two elements are the official OpenGL books; they are not included in the Microsoft Windows SDK.

Components

2/22/2020 • 2 minutes to read • Edit Online

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

- The full set of current OpenGL commands

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

  Many of the OpenGL commands have several variants, which differ in the number and type of their parameters. Counting all the variants, there are more than 300 OpenGL commands.

- The OpenGL Utility (GLU) library

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

- The OpenGL Programming Guide Auxiliary library

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

  Do not use them, however, in a production application. Here are some reasons for this warning:

  - The message loop is in the library code.
  - There is no way to add handlers for additional WM* messages.
  - There is very little support for logical palettes.

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

- The WGL functions

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

- New Windows functions for pixel formats and double buffering

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

Generic Implementation and Hardware Implementations

2/22/2020 • 2 minutes to read • Edit Online

This overview discusses the current generic implementation of OpenGL in Windows. Hardware manufacturers may enhance parts of OpenGL in their drivers and may support some features not supported by the generic implementation.

- Limitations

Limitations

2/22/2020 • 2 minutes to read • Edit Online

The generic implementation has the following limitations:

- Printing.

  You can print an OpenGL image directly to a printer using metafiles only. For more information, see Printing an OpenGL Image.

- OpenGL and GDI graphics cannot be mixed in a double-buffered window.

  An application can directly draw both OpenGL graphics and GDI graphics into a single-buffered window, but not into a double-buffered window.

- There are no per-window hardware color palettes.

  Windows has a single system hardware color palette, which applies to the whole screen. An OpenGL window cannot have its own hardware palette, but can have its own logical palette. To do so, it must become a palette-aware application. For more information, see OpenGL Color Modes and Windows Palette Management.

- There is no direct support for the Clipboard, dynamic data exchange (DDE), or OLE.

  A window with OpenGL graphics does not directly support these Windows capabilities. There are workarounds, however, for using the Clipboard. For more information, see Copying an OpenGL Image to the Clipboard.

- The Inventor 2.0 C++ class library is not included.

  The Inventor class library, built on top of OpenGL, provides higher-level constructs for programming 3-D graphics. It is not included in the current version of Microsoft's implementation of OpenGL for Windows.

- There is no support for the following pixel format features: stereoscopic images, alpha bitplanes, and auxiliary buffers.

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

Rendering Contexts

2/22/2020 • 2 minutes to read • Edit Online

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

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

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

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

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

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

Rendering Context Functions

2/22/2020 • 3 minutes to read • Edit Online

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

WGL FUNCTION	DESCRIPTION
wglCreateContext	Creates a new rendering context.
WglMakeCurrent	Sets a thread's current rendering context.
WglGetCurrentContext	Obtains a handle to a thread's current rendering context.
WglGetCurrentDC	Obtains a handle to the device context associated with a thread's current rendering context.
WglDeleteContext	Deletes a rendering context.

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

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

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

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

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

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

You can break the association between a current rendering context and a thread by calling wglMakeCurrent with

either of two handles:

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

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

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

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

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

Pixel Formats

2/22/2020 • 2 minutes to read • Edit Online

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

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

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

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

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

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

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

Related topics

P
i
x
e
l
F
o
r
m
a
t
F
u

n
c
t
i
o
n
s

Pixel Format Functions

2/22/2020 • 2 minutes to read • Edit Online

The following Windows functions manage pixel formats.

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

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

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

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

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

The DescribePixelFormat function takes the following as parameters:

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

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

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

Front, Back, and Other Buffers

2/22/2020 • 2 minutes to read • Edit Online

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

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

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

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

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

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

Related topics

B
u
f
f
e
r
F
u
n
c
t
i
o
n
s

Buffer Functions

2/22/2020 • 2 minutes to read • Edit Online

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

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

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

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

The following functions also affect buffers:

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

Fonts and Text

2/22/2020 • 2 minutes to read • Edit Online

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

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

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

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

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

Related topics

Fonts and Text Functions

Font and Text Functions

2/22/2020 • 2 minutes to read • Edit Online

The following functions can be used to manage fonts and text.

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

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

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

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

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

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

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

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

OpenGL Color Modes and Windows Palette Management

2/22/2020 • 2 minutes to read • Edit Online

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

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

Palettes and the Palette Manager

2/22/2020 • 2 minutes to read • Edit Online

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

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

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

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

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

Palette Awareness

2/22/2020 • 2 minutes to read • Edit Online

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

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

Reading Color Values from the Framebuffer

2/22/2020 • 2 minutes to read • Edit Online

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

On a true-color device:

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

On a palette-based device:

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

Choosing Between RGBA and Color-Index Mode

2/22/2020 • 2 minutes to read • Edit Online

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

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

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

RGBA Mode and Windows Palette Management

2/22/2020 • 6 minutes to read • Edit Online

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

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

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

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

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

RGBA Mode Palette Sample

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

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

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

#define DEFAULT_GAMMA 1.4F

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

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

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

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

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

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

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

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

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

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

    { 192, 220, 192 },
    { 166, 202, 240 },
    { 255, 251, 240 },
    { 160, 160, 164 },

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

struct rampentry rampmap[256];

void
gammacorrect(double gamma)
```

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

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

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

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

    gammacorrect(gamma);

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

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

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

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

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

        /* Loop through RGB ramp and calculate least square error */
        /* if an entry has already been used, skip it */
```

```
    /* If an entry has already been used, skip it */
    for(j=0; j<256; j++) {
        if (rampmap[j].flags != 0)      /* Already used */
            continue;

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

/* First print out the color cube */

printf("Standard 8-bit RGB color cube with gamma %.2f:\n", gamma);
for (i=0; i<256; i++) {
    pc = &rampmap[i].color;
    printf("%3ld: (%3-D, %3-D, %3-D)\n", i, pc->red,
            pc->green, pc->blue);

}
printf("\n");

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

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

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

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

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

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

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

```
    printf("    0x300,\n");
    printf("    256,\n");

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

        printf("    %3-D, %3-D, %3-D,   0,    /* %ld",
                pc->red, pc->green, pc->blue, i);
        if (rampmap[i].flags == EXACTMATCH)
            printf(" - Exact match with default %d",
                        rampmap[i].defaultindex);
        if (rampmap[i].flags == CHANGED)
            printf(" - Changed to match default %d",
                        rampmap[i].defaultindex);
        printf(" */\n");
    }

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

    return 0;
}
```

Color-Index Mode and Windows Palette Management

2/22/2020 • 2 minutes to read • Edit Online

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

Color-Index Mode Palette Sample

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

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

    ppfd = &pfd;

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

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

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

    pixelformat = ChoosePixelFormat(hdc, ppfd);

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

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

    return TRUE;
}
```

Overlay, Underlay, and Main Planes

2/22/2020 • 2 minutes to read • Edit Online

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

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

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

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

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

The following functions are used with layer planes:

- wglCopyContext
- wglCreateLayerContext
- wglDescribeLayerPlane
- wglGetLayerPaletteEntries
- wglRealizeLayerPalette
- wglSetLayerPaletteEntries
- wglSwapLayerBuffers

Sharing Display Lists

2/22/2020 • 2 minutes to read • Edit Online

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

Extending OpenGL Functions

2/22/2020 • 2 minutes to read • Edit Online

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

GLX and WGL/Windows

2/22/2020 • 2 minutes to read • Edit Online

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

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

GLX FUNCTIONS	WGL/WINDOWS FUNCTIONS
	SetPixelFormat
	wglGetProcAddress
	wglShareLists

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

Using OpenGL on Windows

2/22/2020 • 2 minutes to read • Edit Online

The following topics explain how to use several features specific to Windows:

- Header Files
- Pixel Format Tasks
- Rendering Context Tasks
- Drawing with Double Buffers
- Drawing Text in a Double-Buffered OpenGL Window
- Printing an OpenGL Image
- Copying an OpenGL Image to the Clipboard
- Multithread OpenGL Drawing Strategies
- Using the Auxiliary Library

For information about OpenGL ES running on Windows, see MSOpenTech ANGLE.

Header Files

2/22/2020 • 2 minutes to read • Edit Online

Applications that use:

- The core OpenGL functions must include the header file <GL\gl.h>.
- The OpenGL Utility library must include the header file <GL\glu.h>.
- The OpenGL Programming Guide auxiliary library must include the header file <GL\glaux.h>.
- The WGL functions must include the header file windows.h.
- The Windows functions that support Microsoft's implementation of OpenGL in Windows must include the header file Windows.h.

Pixel Format Tasks

2/22/2020 • 2 minutes to read • Edit Online

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

- Choosing and Setting a Best-Match Pixel Format
- Examining a Device Context's Current Pixel Format
- Examining a Device's Supported Pixel Format

Choosing and Setting a Best-Match Pixel Format

2/22/2020 • 2 minutes to read • Edit Online

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

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

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

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

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

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

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

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

Examining a Device Contexts Current Pixel Format

2/22/2020 • 2 minutes to read • Edit Online

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

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

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

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

Examining a Devices Supported Pixel Formats

2/22/2020 • 2 minutes to read • Edit Online

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

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

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

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

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

        // return indicating failure
        return(FALSE);

    // we have successfully obtained pixel format data

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

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

Rendering Context Tasks

2/22/2020 • 2 minutes to read • Edit Online

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

- Creating a Rendering Context and Making It Current
- Making a Rendering Context Not Current
- Deleting a Rendering Context

Creating a Rendering Context and Making It Current

2/22/2020 • 2 minutes to read • Edit Online

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

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

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

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

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

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

        }

    // perform miscellaneous other WM_CREATE chores ...

    }
    break;
```

Making a Rendering Context Not Current

2/22/2020 • 2 minutes to read • Edit Online

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

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

Deleting a Rendering Context

2/22/2020 • 2 minutes to read • Edit Online

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

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

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

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

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

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

        // delete the rendering context
        wglDeleteContext(hglrc);

        }
```

Drawing with Double Buffers

2/22/2020 • 2 minutes to read • Edit Online

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

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

    // draw our objects
    myDrawAllObjects(GL_FALSE);

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

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

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

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

Drawing Text in a Double-Buffered OpenGL Window

2/22/2020 • 2 minutes to read • Edit Online

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

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

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

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

// make the shading smooth
glShadeModel(GL_SMOOTH);

// clear the color buffers
glClear(GL_COLOR_BUFFER_BIT);

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

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

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

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

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

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

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

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

// release the device context
ReleaseDC(hdc) ;

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

Printing an OpenGL Image

2/22/2020 • 2 minutes to read • Edit Online

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

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

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

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

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

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

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

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

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

Copying an OpenGL Image to the Clipboard

2/22/2020 • 2 minutes to read • Edit Online

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

To copy an OpenGL image to the Clipboard

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

Multithread OpenGL Drawing Strategies

2/22/2020 • 2 minutes to read • Edit Online

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

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

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

Using the Auxiliary Library

2/22/2020 • 2 minutes to read • Edit Online

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

Reference for Windows Extensions to OpenGL

2/22/2020 • 2 minutes to read • Edit Online

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

- WGL Functions
- Windows Functions
- Structures

WGL Functions

2/22/2020 • 2 minutes to read • Edit Online

Following are the WGL functions associated with OpenGL:

- wglCopyContext
- wglCreateContext
- wglCreateLayerContext
- wglDeleteContext
- wglDescribeLayerPlane
- wglGetCurrentContext
- wglGetCurrentDC
- wglGetLayerPaletteEntries
- wglGetProcAddress
- wglMakeCurrent
- wglRealizeLayerPalette
- wglSetLayerPaletteEntries
- wglShareLists
- wglSwapLayerBuffers
- wglUseFontBitmaps
- wglUseFontOutlines

Windows Functions

2/22/2020 • 2 minutes to read • Edit Online

Following are the Windows functions associated with OpenGL:

- ChoosePixelFormat
- DescribePixelFormat
- GetEnhMetaFilePixelFormat
- GetPixelFormat
- SetPixelFormat
- SwapBuffers

OpenGL structures

2/22/2020 • 2 minutes to read • Edit Online

Following are the structures associated with OpenGL:

- GLYPHMETRICSFLOAT
- LAYERPLANEDESCRIPTOR
- PIXELFORMATDESCRIPTOR
- POINTFLOAT

WGL and Windows Reference

2/22/2020 • 2 minutes to read • Edit Online

The following are the functions:

- ChoosePixelFormat
- DescribePixelFormat
- GetEnhMetaFilePixelFormat
- GetPixelFormat
- SetPixelFormat
- SwapBuffers
- wglCreateContext
- wglCreateLayerContext
- wglCopyContext
- wglDeleteContext
- wglDescribeLayerPlane
- wglGetCurrentContext
- wglGetCurrentDC
- wglGetLayerPaletteEntries
- wglGetProcAddress
- wglMakeCurrent
- wglRealizeLayerPalette
- wglSetLayerPaletteEntries
- wglShareLists
- wglSwapLayerBuffers
- wglUseFontBitmaps
- wglUseFontOutlines

The following are the structures:

- GLYPHMETRICSFLOAT
- LAYERPLANEDESCRIPTOR
- PIXELFORMATDESCRIPTOR
- POINTFLOAT

Programming Tips

2/22/2020 • 2 minutes to read • Edit Online

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

- OpenGL Correctness Tips
- OpenGL Performance Tips
- Security Considerations: OpenGL

OpenGL Correctness Tips

2/22/2020 • 2 minutes to read • Edit Online

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

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

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

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

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

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

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

OpenGL Performance Tips

2/22/2020 • 2 minutes to read • Edit Online

These programming practices optimize your application's performance:

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

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

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

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

Security Considerations: OpenGL

2/22/2020 • 2 minutes to read • Edit Online

This document provides a list of topics that contain security information. These topics do not provide all you need to know about security issues—instead, use them as a starting point and reference for this technology area.

Related Topics

- MSDN Security Home Page
- TechNet Security Resources

OpenGL Reference

2/22/2020 • 2 minutes to read • Edit Online

The API elements that OpenGL provides fall into the following three sections:

- State Variables
- GL Functions
- GLU Functions

State Variables

2/22/2020 • 2 minutes to read • Edit Online

This section lists variables that can be queried, their default values, and the commands for obtaining the values of these variables. State variables are listed in the following categories:

- Transformation
- Coloring
- Lighting
- Rasterization
- Texturing
- Framebuffer control
- Pixel
- Evaluator
- Hint
- Implementation-dependent
- Implementation-dependent pixel depth
- Miscellaneous

Attribute Groups

2/22/2020 • 2 minutes to read • Edit Online

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

State Variables for Current Values and Associated Data

2/22/2020 • 2 minutes to read • Edit Online

GL_CURRENT_COLOR

Description:	
Attribute group:	
Initial value:	

Get command:	[illegible]

GL_CURRENT_INDEX

Description:	[illegible]

Attribute group:	
Initial value:	
Get command:	

GL_CURRENT_TEXTURE_COORDS

Description:	
Attribute group:	
Initial value:	

Get command:	

GL_CURRENT_NORMAL

Description:	
Attribute group:	
Initial value:	

Get command:	[illegible]

GL_CURRENT_RASTER POSITION

Description:	[illegible]
Attribute group:	[illegible]

Initial value:	
Get command:	

GL_CURRENT_RASTER_DISTANCE

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_CURRENT_RASTER_COLOR

Description:	[illegible]
Attribute group:	[illegible]

Initial value:	
Get command:	

GL_CURRENT_RASTER_INDEX

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_CURRENT_RASTER_TEXTURE_COORDS

Description:

Attribute group:

Initial value:	
Get command:	glGetCommandHeadersNV glGetStagesNV glGetTokenizerStrokeStyleNV glGetVertexAttribdvNV glGetVertexAttribfvNV glGetVertexAttribiv glGetVertexAttribPointerNV glGetVertexAttribPointerv glGetVertexAttribIiv glGetVertexAttribIivEXT glGetVertexAttribIivNV glGetVertexAttribIuiv glGetVertexAttribIuivEXT glGetVertexAttribIuivNV glGetVertexAttribLdv glGetVertexAttribLdvEXT glGetVertexAttribLdvNV glGetVertexAttribLi64vNV glGetVertexAttribLui64vNV glGetVertexAttribLui64vNV

GL_CURRENT_RASTER_POSITION_VALID

Description:	Indicates if the raster position is valid.

Attribute group:	
Initial value:	
Get command:	

GL_EDGE_FLAG

Description:	

Attribute group:	
Initial value:	
Get command:	

Transformation State Variables

2/22/2020 • 2 minutes to read • Edit Online

GL_MODELVIEW_MATRIX

Description:	
Attribute group:	
Initial value:	

Get command:	[illegible]

GL_PROJECTION_MATRIX

Description:	[illegible]
Attribute group:	

Initial value:	
Get command:	glGetTexLevelParameter

GL_TEXTURE_MATRIX

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_VIEWPORT

Description:	
Attribute group:	
Initial value:	

Get command:	[illegible]

GL_DEPTH_RANGE

Description:	[illegible]
Attribute group:	[illegible]

Initial value:	
Get command:	

GL_MODELVIEW_STACK_DEPTH

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_PROJECTION_STACK_DEPTH

Description:	
Attribute group:	
Initial value:	

Get command:	

GL_TEXTURE_STACK_DEPTH

Description:	
Attribute group:	

Initial value:	
Get command:	glGetIntegerv

GL_MATRIX_MODE

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_NORMALIZE

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_CLIP_PLANE
¡

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_CLIP_PLANE

¡

Description:	
Attribute group:	
Initial value:	

Get command:	

Coloring State Variables

2/22/2020 • 2 minutes to read • Edit Online

GL_FOG_COLOR

Description:	
Attribute group:	
Initial value:	

Get command:	glGetCommandHeadersNV

GL_FOG_INDEX

Description:	
Attribute group:	
Initial value:	0

Get command:	[illegible]

GL_FOG_DENSITY

Description:	[illegible]
Attribute group:	[illegible]

Initial value:	
Get command:	

GL_FOG_START

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_FOG_END

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_FOG_MODE

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_FOG

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_SHADE_MODEL

Description:	
Attribute group:	
Initial value:	

Get command:	[illegible]

Lighting State Variables

2/22/2020 • 2 minutes to read • Edit Online

GL_LIGHTING

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_COLOR_MATERIAL

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_COLOR_MATERIAL_PARAMETER

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_COLOR_MATERIAL_FACE

Description:	[illegible]
Attribute group:	[illegible]

Initial value:	
Get command:	

GL_AMBIENT

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_DIFFUSE

Description:	
Attribute group:	
Initial value:	

Get command:	

GL_SPECULAR

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_EMISSION

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_SHININESS

Description:	
Attribute group:	
Initial value:	

Get command:	

GL_LIGHT_MODEL_AMBIENT

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_LIGHT_MODEL_LOCAL_VIEWER

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_LIGHT_MODEL_TWO_SIDE

Description:	
Attribute group:	
Initial value:	

Get command:	

GL_AMBIENT

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_DIFFUSE

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_SPECULAR

Description:	
Attribute group:	
Initial value:	

Get command:	glcommand

GL_POSITION

Description:	Describition
Attribute group:	Attribute group:

Initial value:	
Get command:	

GL_CONSTANT_ATTENUATION

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_LINEAR_ATTENUATION

Description:	
Attribute group:	
Initial value:	

Get command:	glGetMaterialfv glGetMaterialiv glGetMaterialxfv glGetMaterialxiv

GL_QUADRATIC_ATTENUATION

Description:	Determines the quadratic attenuation factor.
Attribute group:	Light

Initial value:	
Get command:	

GL_SPOT_DIRECTION

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_SPOT_EXPONENT

Description:	
Attribute group:	
Initial value:	

Get command:	

GL_SPOT_CUTOFF

Description:	
Attribute group:	

Initial value:	
Get command:	glGetLightfv glGetLightiv

GL_LIGHT
¡

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_COLOR_INDEXES

Description:	
Attribute group:	

Initial value:	
Get command:	

Rasterization State Variables

2/22/2020 • 2 minutes to read • Edit Online

GL_POINT_SIZE

Description:	
Attribute group:	
Initial value:	

Get command:	glEnable(GL_POINT_SMOOTH);

GL_POINT_SMOOTH

Description:	Draw points with antialiasing.
Attribute group:	Point/Antialias

Initial value:	
Get command:	

GL_LINE_WIDTH

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_LINE_SMOOTH

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_LINE_STIPPLE_PATTERN

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_LINE_STIPPLE_REPEAT

Description:	
Attribute group:	
Initial value:	
Get command:	

GL_LINE_STIPPLE

Description:	
Attribute group:	
Initial value:	

Get command:	

GL_CULL_FACE

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_CULL_FACE_MODE

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_FRONT_FACE

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_POLYGON_SMOOTH

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_POLYGON_MODE

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_POLYGON_STIPPLE

Description:	

Attribute group:	
Initial value:	
Get command:	

Description:	
Attribute group:	
Initial value:	

Get command:

Texturing State Variables

2/22/2020 • 2 minutes to read • Edit Online

GL_TEXTURE_*x*

Description:

Attribute group:	
Initial value:	
Get command:	

GL_TEXTURE

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_TEXTURE_WIDTH

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_TEXTURE_HEIGHT

Description:	
Attribute group:	
Initial value:	

Get command:	[illegible]

GL_TEXTURE_BORDER

Description:	[illegible]

Attribute group:	
Initial value:	
Get command:	

GL_TEXTURE_COMPONENTS

Description:	[illegible]
Attribute group:	
Initial value:	

Get command:	[illegible]

GL_TEXTURE_BORDER_COLOR

Description:	[illegible]
Attribute group:	[illegible]

Initial value:	
Get command:	

GL_TEXTURE_MIN_FILTER

Description:	[illegible]
Attribute group:	[illegible]

Initial value:	
Get command:	

GL_TEXTURE_MAG_FILTER

Description:	
Attribute group:	
Initial value:	

Get command:

GL_TEXTURE_WRAP_*x*

Description:

Attribute group:	
Initial value:	
Get command:	

GL_TEXTURE_ENV_MODE

Description:	
Attribute group:	
Initial value:	

Get command:	

GL_TEXTURE_ENV_COLOR

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_TEXTURE_GEN_
x

Description:

Attribute group:

Initial value:

Get command:	

GL_EYE_PLANE

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_OBJECT_PLANE

Description:	
Attribute group:	
Initial value:	

Get command:	

GL_TEXTURE_GEN_MODE

Description:	
Attribute group:	

Initial value:	
Get command:	

Pixel Operations

2/22/2020 • 2 minutes to read • Edit Online

GL_SCISSOR_TEST

Description:	
Attribute group:	
Initial value:	

Get command:	glscissor

GL_SCISSOR_BOX

Description:	Sets the scissor box.
Attribute group:	Scissor
Initial value:	

Get command:	

GL_STENCIL_TEST

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_STENCIL_FUNC

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_STENCIL_VALUE_MASK

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_STENCIL_REF

Description:	
Attribute group:	
Initial value:	

Get command:	

GL_STENCIL_FAIL

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_STENCIL_PASS_DEPTH_FAIL

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_STENCIL_PASS_DEPTH_PASS

Description:	
Attribute group:	
Initial value:	

Get command:	

GL_ALPHA_TEST

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_ALPHA_TEST_FUNC

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_ALPHA_TEST_REF

Description:	A l p a r t s t o n e c o n t r o l v a l v e v a l u e
Attribute group:	c o n t r o l l e r t y p e
Initial value:	c

Get command:	[illegible]

GL_DEPTH_TEST

Description:	[illegible]
Attribute group:	[illegible]

Initial value:	
Get command:	

GL_DEPTH_FUNC

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_BLEND

Description:	
Attribute group:	
Initial value:	
Get command:	

GL_BLENC_SRC

Description:	
Attribute group:	
Initial value:	

Get command:	

GL_BLEND_DST

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_DITHER

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_LOGIC_OP

Description:	
Attribute group:	
Initial value:	

Get command:	

GL_LOGIC_OP_MODE

Description:	
Attribute group:	

Initial value:	
Get command:	

Framebuffer Control State Variables

2/22/2020 • 2 minutes to read • Edit Online

GL_DRAW_BUFFER

Description:

Attribute group:

Initial value:	
Get command:	

GL_INDEX_WRITEMASK

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_COLOR_WRITEMASK

Description:	
Attribute group:	
Initial value:	

Get command:	

GL_DEPTH_WRITEMASK

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_STENCIL_WRITEMASK

Description:	
Attribute group:	
Initial value:	

Get command:	glGetCommandHeadersNV glGetCommandHeaderNV glGetStageIndexNV

GL_COLOR_CLEAR_VALUE

Description:	glClearColor glClearBufferfv glClearBufferfi glClearBufferiv glClearBufferuiv glClearNamedBufferfv glClearNamedBufferfi glClearNamedBufferiv glClearNamedBufferuiv glClearTexImage glClearTexSubImage glGetFloatv glGetDoublev glGetIntegerv glGetBooleanv glGetInteger64v glGetIntegerui64vNV glGetIntegerui64i_vNV glGetIntegerindexedvEXT glGetIntegerindexedvOES glGetBooleanindexedvEXT glGetBooleanindexedvOES glGetIntegerv glGetIntegerv glGetIntegerv

Attribute group:	
Initial value:	
Get command:	

GL_INDEX_CLEAR_VALUE

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_DEPTH_CLEAR_VALUE

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_STENCIL_CLEAR_VALUE

Description:	
Attribute group:	
Initial value:	

Get command:	

GL_ACCUM_CLEAR_VALUE

Description:	

Attribute group:	
Initial value:	
Get command:	

Pixel State Variables

2/22/2020 • 2 minutes to read • Edit Online

GL_UNPACK_SWAP_BYTES

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_UNPACK_LSB_FIRST

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_UNPACK_ROW_LENGTH

Description:	
Attribute group:	
Initial value:	

Get command:	

GL_UNPACK_SKIP_ROWS

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_UNPACK_SKIP_PIXELS

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_UNPACK_ALIGNMENT

Description:	
Attribute group:	
Initial value:	

Get command:

GL_PACK_SWAP_BYTES

Description:

Attribute group:

Initial value:	
Get command:	

GL_PACK_LSB_FIRST

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_PACK_ROW_LENGTH

Description:	[illegible]
Attribute group:	
Initial value:	[illegible]

Get command:	

GL_PACK_SKIP_ROWS

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_PACK_SKIP_PIXELS

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_PACK_ALIGNMENT

Description:	
Attribute group:	
Initial value:	

Get command:	[illegible]

GL_MAP_COLOR

Description:	[illegible]
Attribute group:	[illegible]

Initial value:	
Get command:	

GL_MAP_STENCIL

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_INDEX_SHIFT

Description:	
Attribute group:	
Initial value:	

Get command:	

GL_INDEX_OFFSET

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_*x*_SCALE

Description:

Attribute group:	
Initial value:	
Get command:	

GL_*x*_BIAS

Description:

Attribute group:	
Initial value:	
Get command:	

GL_ZOOM_X

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_ZOOM_Y

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_*x*

Description:	
Attribute group:	
Initial value:	

Get command:	

GL_*x*_SIZE

Description:	
Attribute group:	
Initial value:	

Get command:	

GL_READ_BUFFER

Description:	
Attribute group:	

Initial value:	
Get command:	g l c c t l m t c g c l v

Evaluator State Variables

2/22/2020 • 2 minutes to read • Edit Online

GL_ORDER

Description:	
Attribute group:	
Initial value:	

Get command:	

GL_ORDER

Description:	
Attribute group:	
Initial value:	

Get command:	

GL_COEFF

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_COEFF

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_DOMAIN

Description:	
Attribute group:	
Initial value:	
Get command:	

GL_DOMAIN

Description:	
Attribute group:	
Initial value:	

Get command:	

GL_MAP1_x

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_MAP2_x

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_MAP1_GRID_DOMAIN

Description:	
Attribute group:	
Initial value:	
Get command:	

GL_MAP2_GRID_DOMAIN

Description:

Attribute group:

Initial value:

Get command:

GL_MAP1_GRID_SEGMENTS

Description:

Attribute group:

Initial value:

Get command:

GL_MAP1_GRID_SEGMENTS

Description:	
Attribute group:	
Initial value:	
Get command:	

GL_AUTO_NORMAL

Description:	[illegible]
Attribute group:	[illegible]

Initial value:	
Get command:	

Hint State Variables

2/22/2020 • 2 minutes to read • Edit Online

GL_PERSPECTIVE_CORRECTION_HINT

Description:

Attribute group:

Initial value:	
Get command:	

GL_POINT_SMOOTH_HINT

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_LINE_SMOOTH_HINT

Description:	
Attribute group:	
Initial value:	
Get command:	

GL_POLYGON_SMOOTH_HINT

Description:

Attribute group:

Initial value:

Get command:

GL_FOG_HINT

Description:	
Attribute group:	
Initial value:	
Get command:	

Implementation-Dependent State Variables

2/22/2020 • 2 minutes to read • Edit Online

GL_MAX_LIGHTS

Description:	
Attribute group:	
Initial value:	

Get command:	[illegible]

GL_MAX_CLIP_PLANES

Description:	[illegible]

Attribute group:	
Initial value:	
Get command:	

GL_MAX_MODELVIEW_STACK_DEPTH

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_MAX_PROJECTION_STACK_DEPTH

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_MAX_TEXTURE_STACK_DEPTH

Description:	[illegible]
Attribute group:	

Initial value:	
Get command:	

GL_SUBPIXEL_BITS

Description:

Attribute group:

Initial value:	4
Get command:	glGetIntegerv

GL_MAX_TEXTURE_SIZE

Description:

Attribute group:	
Initial value:	
Get command:	

GL_MAX_PIXEL_MAP_TABLE

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_MAX_NAME_STACK_DEPTH

Description:	
Attribute group:	
Initial value:	

Get command:	glGetCommandHeaderNV glGetStageIndexNV

GL_MAX_LIST_NESTING

Description:	Data is a single integer indicating the maximum recursion depth for display list call instructions. The value must be at least 1.

Attribute group:	
Initial value:	
Get command:	

GL_MAX_EVAL_ORDER

Description:	
Attribute group:	
Initial value:	

Get command:	

GL_MAX_VIEWPORT_DIMS

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_MAX_ATTRIB_STACK_DEPTH

Description:	
Attribute group:	
Initial value:	

Get command:	[illegible]

GL_AUX_BUFFERS

Description:	[illegible]
Attribute group:	

Initial value:	
Get command:	

GL_RGBA_MODE

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_INDEX_MODE

Description:	[illegible]
Attribute group:	
Initial value:	

Get command:	

GL_DOUBLEBUFFER

Description:	

Attribute group:	
Initial value:	
Get command:	glGetBooleanv glGetDoublev glGetFloatv glGetIntegerv

GL_STEREO

Description:	[illegible]
Attribute group:	
Initial value:	

Get command:	[illegible]

GL_POINT_SIZE_RANGE

Description:	[illegible]

Attribute group:	
Initial value:	
Get command:	

GL_POINT_SIZE_GRANULARITY

Description:	A [illegible]
Attribute group:	
Initial value:	

Get command:	

GL_LINE_WIDTH_RANGE

Description:	

Attribute group:	
Initial value:	
Get command:	

GL_LINE_WIDTH_GRANULARITY

Description:	A
Attribute group:	

Initial value:	
Get command:	g l c c t l c a t v

Implementation-Dependent Pixel-Depth State Variables

2/22/2020 • 2 minutes to read • Edit Online

GL_RED_BITS

Description:

Attribute group:	
Initial value:	
Get command:	

GL_GREEN_BITS

Description:	
Attribute group:	

Get command:	glGetMultiTexGenfv glGetMultiTexGeniv glGetMultiTexGenfv glGetMultiTexGeniv
Initial value:	

GL_BLUE_BITS

Description:	
Attribute group:	

Initial value:	
Get command:	[illegible]

GL_ALPHA_BITS

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_INDEX_BITS

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_DEPTH_BITS

Description:	

Attribute group:	
Initial value:	
Get command:	
GL_STENCIL_BITS	

Description:	
Attribute group:	
Initial value:	

Get command:	

GL_ACCUM_RED_BITS

Description:

Attribute group:	
Initial value:	
Get command:	

GL_ACCUM_GREEN_BITS

Description:

Attribute group:	
Initial value:	
Get command:	

GL_ACCUM_BLUE_BITS

Description:

Attribute group:	
Initial value:	
Get command:	glGetIntegerv

GL_ACCUM_ALPHA_BITS

Description:

Attribute group:	
Initial value:	
Get command:	

Miscellaneous State Variables

2/22/2020 • 2 minutes to read • Edit Online

GL_LIST_BASE

Description:	
Attribute group:	
Initial value:	

Get command:	

GL_LIST_INDEX

Description:	
Attribute group:	

Initial value:	
Get command:	

GL_LIST_MODE

Description:

Attribute group:	
Initial value:	
Get command:	

GL_ATTRIB_STACK_DEPTH

Description:	A
Attribute group:	
Initial value:	C

Get command:	[illegible]

GL_NAME_STACK_DEPTH

Description:	[illegible]
Attribute group:	

Initial value:	
Get command:	

GL_RENDER_MODE

Description:	

Attribute group:	
Initial value:	
Get command:	

Description:	
Attribute group:	
Initial value:	
Get command:	

GL Functions

2/22/2020 • 5 minutes to read • Edit Online

The OpenGL commands, in alphabetical order are presented here. Each reference page describes one or more function. See also GLU Functions.

FUNCTION	DESCRIPTION
glAccum	Operates on the accumulation buffer.
glAddSwapHintRectWIN	Specifies a set of rectangles that are to be copied by SwapBuffers.
glAlphaFunc	Enables your application to set the alpha test function.
glAreTexturesResident	Determines whether specified texture objects are resident in texture memory.
glArrayElement	Specifies the array elements used to render a vertex.
glBegin, glEnd	Delimit the vertices of a primitive or a group of like primitives.
glBindTexture	Enables the creation of a named texture that is bound to a texture target.
glBitmap	Draws a bitmap.
glBlendFunc	Specifies pixel arithmetic.
glCallList	Executes a display list.
glCallLists	Executes a list of display lists.
glClear	Clears buffers to preset values.
glClearAccum	Specifies the clear values for the accumulation buffer.
glClearColor	Specifies clear values for the color buffers.
glClearDepth	Specifies the clear value for the depth buffer.
glClearIndex	Specifies the clear value for the color-index buffers.
glClearStencil	Specifies the clear value for the stencil buffer.
glClipPlane	Specifies a plane against which all geometry is clipped.
glColor functions	Set the current color.

FUNCTION	DESCRIPTION
glColorMask	Enables and disables writing of frame-buffer color components.
glColorMaterial	Causes a material color to track the current color.
glColorPointer	Defines an array of colors.
glColorTableEXT	Specifies the format and size of a palette for targeted paletted textures.
glColorSubTableEXT	Specifies a portion of the targeted texture's palette to be replaced.
glCopyPixels	Copies pixels in the framebuffer.
glCopyTexImage1D	Copies pixels from the framebuffer into a one-dimensional texture image.
glCopyTexImage2D	Copies pixels from the framebuffer into a two-dimensional texture image.
glCopyTexSubImage1D	Copies a sub-image of a one-dimensional texture image from the framebuffer.
glCopyTexSubImage2D	Copies a sub-image of a two-dimensional texture image from the framebuffer.
glCullFace	Specifies whether front- or back-facing facets can be culled.
glDeleteLists	Deletes a contiguous group of display lists.
glDeleteTextures	Deletes named textures.
glDepthFunc	Specifies the value used for depth-buffer comparisons.
glDepthMask	Enables or disables writing into the depth buffer.
glDepthRange	Specifies the mapping of *z* values from normalized device coordinates to window coordinates.
glDrawArrays	Specifies multiple primitives to render.
glDrawBuffer	Specifies which color buffers are to be drawn into.
glDrawElements	Renders primitives from array data.
glDrawPixels	Writes a block of pixels to the framebuffer.
glEdgeFlag functions	Define an array of edge flags.
glEdgeFlagPointer	Defines an array of edge flags.

FUNCTION	DESCRIPTION
glEnable, glDisable	Enables or disables OpenGL capabilities.
glEnableClientState, glDisableClientState	Enables and disables arrays, respectively.
glEvalCoord functions	Evaluate enabled one- and two-dimensional maps.
glEvalMesh functions	Compute a one- or two-dimensional grid of points or lines.
glEvalPoint functions	Generate and evaluate a single point in a mesh.
glFeedbackBuffer	Controls feedback mode.
glFinish	Blocks until all OpenGL execution is complete.
glFlush	Forces execution of OpenGL functions in finite time.
glFog functions	Specify fog parameters.
glFrontFace	Defines front- and back-facing polygons.
glFrustum	Multiplies the current matrix by a perspective matrix.
glGenLists	Generates a contiguous set of empty display lists.
glGenTextures	Generates texture names.
glGetBooleanv	Retrieves the value or values of a selected Boolean parameter.
glGetClipPlane	Retrieves the coefficients of the specified clipping plane.
glGetColorTableEXT	Retrieves the color table data of the current targeted texture palette.
glGetColorTableParameterfvEXT, glGetColorTableParameterivEXT	Retrieves palette parameters from color tables.
glGetDoublev	Retrieves the value or values of a selected double parameter.
glGetError	Retrieves error information.
glGetFloatv	Retrieves the value or values of a selected float parameter.
glGetIntegerv	Retrieves the value or values of a selected int parameter.
glGetLight functions	Retrieve light source parameter values.
glGetMap functions	Retrieve evaluator parameters.
glGetMaterial functions	Retrieve material parameters.

FUNCTION	DESCRIPTION
glGetPixelMap functions	Retrieve the specified pixel map.
glGetPointerv	Retrieves the address of a vertex data array.
glGetPolygonStipple	Retrieves the polygon stipple pattern.
glGetString	Retrieves a string describing the current OpenGL connection.
glGetTexEnv functions	Retrieve texture environment parameters.
glGetTexGen functions	Retrieve texture coordinate generation parameters.
glGetTexImage	Retrieves a texture image.
glGetTexLevelParameter functions	Retrieve texture parameter values for a specific level of detail.
glGetTexParameter functions	Retrieve texture parameter values.
glHint	Specifies implementation-specific hints.
glIndex functions	Set the current color index.
glIndexMask	Controls the writing of individual bits in the color-index buffers.
glIndexPointer	Defines an array of color indexes.
glInitNames	Initializes the name stack.
glInterleavedArrays	Simultaneously specifies and enables several interleaved arrays in a larger aggregate array.
glIsEnabled	Tests whether a capability is enabled.
glIsList	Tests for display list existence.
glIsTexture	Determines whether a name corresponds to a texture.
glLight functions	Set light-source parameters.
glLightModel functions	Set the lighting model parameters.
glLineStipple	Specifies the line stipple pattern.
glLineWidth	Specifies the width of rasterized lines.
glListBase	Sets the display list base for **glCallLists**.
glLoadIdentity	Replaces the current matrix with the identity matrix.

FUNCTION	DESCRIPTION
glLoadMatrix functions	Replace the current matrix with an arbitrary matrix.
glLoadName	Loads a name onto the name stack.
glLogicOp	Specifies a logical pixel operation for color index rendering.
glMap1 functions	Define a one-dimensional evaluator.
glMap2 functions	Define a two-dimensional evaluator.
glMapGrid functions	Define a one- or two-dimensional mesh.
glMaterial functions	Specify material parameters for the lighting model.
glMatrixMode	Specifies which matrix is the current matrix.
glMultMatrix functions	Multiply the current matrix by an arbitrary matrix.
glNewList, glEndList	Create or replace a display list.
glNormal functions	Set the current normal vector.
glNormalPointer	Defines an array of normals.
glOrtho	Multiplies the current matrix by an orthographic matrix.
glPassThrough	Places a marker in the feedback buffer.
glPixelMap functions	Set up pixel transfer maps.
glPixelStore functions	Set pixel storage modes.
glPixelTransfer functions	Set pixel transfer modes.
glPixelZoom	Specifies the pixel zoom factors.
glPointSize	Specifies the diameter of rasterized points.
glPolygonMode	Selects a polygon rasterization mode.
glPolygonOffset	Sets the scale and units OpenGL uses to calculate depth values.
glPolygonStipple	Sets the polygon stippling pattern.
glPrioritizeTextures	Sets the residence priority of textures.
glPushAttrib, glPopAttrib	Push and pop the attribute stack.

FUNCTION	DESCRIPTION
glPushClientAttrib, glPopClientAttrib	Save and restore groups of client-state variables on the client-attribute stack.
glPushMatrix, glPopMatrix	Push and pop the current matrix stack, respectively.
glPushName, glPopName	Push and pop the name stack, respectively.
glRasterPos functions	Specify the raster position for pixel operations.
glReadBuffer	Slects a color buffer source for pixels.
glReadPixels	Reads a block of pixels from the framebuffer.
glRect functions	Draw a rectangle.
glRenderMode	Sets the rasterization mode.
glRotate functions	Multiply the current matrix by a rotation matrix.
glScale functions	Multiply the current matrix by a general scaling matrix.
glScissor	Defines the scissor box.
glSelectBuffer	Establishes a buffer for selection mode values.
glShadeModel	Selects flat or smooth shading.
glStencilFunc	Sets the function and reference value for stencil testing.
glStencilMask	Controls the writing of individual bits in the stencil planes.
glStencilOp	Sets the stencil test actions.
glTexCoord functions	Set the current texture coordinates.
glTexCoordPointer	Defines an array of texture coordinates.
glTexEnv functions	Set texture environment parameters.
glTexGen functions	Control the generation of texture coordinates.
glTexImage1D	Specifies a one-dimensional texture image.
glTexImage2D	Specifies a two-dimensional texture image.
glTexParameter functions	Set texture parameters.
glTexSubImage1D	Specifies a portion of an existing one-dimensional texture image. You cannot define a new texture with this function.

FUNCTION	DESCRIPTION
glTexSubImage2D	Specifies a portion of an existing two-dimensional texture image. You cannot define a new texture with this function.
glTranslate functions	Multiply the current matrix by a translation matrix.
glVertex	These functions specify a vertex.
glVertexPointer	Defines an array of vertex data.
glViewport	Sets the viewport.

glAccum function

2/22/2020 • 3 minutes to read • Edit Online

The **glAccum** function operates on the accumulation buffer.

Syntax

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

Parameters

op

The accumulation buffer operation. The accepted symbolic constants are as follows.

VALUE	MEANING
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.

VALUE	MEANING
GL_RETURN	Transfers accumulation buffer values to the color buffer or buffers currently selected for writing. Each R, G, B, and A component is multiplied by *value*, then multiplied by $2^n - 1$, clamped to the range $[0, 2^n - 1]$, and stored in the corresponding display buffer cell. The only fragment operations that are applied to this transfer are pixel ownership, scissor, dithering, and color writemasks.

value

A floating-point value used in the accumulation buffer operation. The *op* parameter determines how *value* is used.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*op* was not an accepted value.
GL_INVALID_OPERATION	There was no accumulation buffer or the function **glAccum** was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*op* was not an accepted value.
GL_INVALID_OPERATION	There was no accumulation buffer or the function **glAccum** was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The accumulation buffer is an extended-range color buffer. Images are not rendered into it. Rather, images rendered into one of the color buffers are added to the contents of the accumulation buffer after rendering. You can create effects such as antialiasing (of points, lines, and polygons), motion blur, and depth of field by accumulating images generated with different transformation matrices.

Each pixel in the accumulation buffer consists of red, green, blue, and alpha values. The number of bits per component in the accumulation buffer depends on the implementation. You can examine this number by calling **glGetIntegerv** four times, with the arguments GL_ACCUM_RED_BITS, GL_ACCUM_GREEN_BITS, GL_ACCUM_BLUE_BITS, and GL_ACCUM_ALPHA_BITS, respectively. Regardless of the number of bits per component, however, the range of values stored by each component is [1,?1]. The accumulation buffer pixels are mapped one-to-one with framebuffer pixels.

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

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

To clear the accumulation buffer, use the **glClearAccum** function to specify R, G, B, and A values to set it to, and issue a **glClear** function with the accumulation buffer enabled.

Only those pixels within the current scissor box are updated by any **glAccum** operation.

The following functions retrieve information related to the **glAccum** function:

glGet with argument GL_ACCUM_RED_BITS

glGet with argument GL_ACCUM_GREEN_BITS

glGet with argument GL_ACCUM_BLUE_BITS

glGet with argument GL_ACCUM_ALPHA_BITS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glBlendFunc

glClear

gl

C
l
e
a
r
A
c
c
u
m

g
l
C
o
p
y
P
i
x
e
l
s

g
l
E
n
d

g
l
G
e
t

g
l
L
o
g
i
c
O
p

g
l
P
i
x
e
l

S
t
o
r
e

g
l
P
i
x
e
l
T
r
a
n
s
f
e
r

g
l
R
e
a
d
B
u
f
f
e
r

g
l
R
e
a
d
P
i
x
e
l
s

g
l
S
c
i

s
s
o
r

g
l
S
t
e
n
c
i
l
O
p

glAddSwapHintRectWIN function

2/22/2020 • 2 minutes to read • Edit Online

The **glAddSwapHintRectWIN** callback function specifies a set of rectangles that are to be copied by **SwapBuffers**.

Syntax

```
void WINAPI glAddSwapHintRectWIN(
   GLint   x,
   GLint   y,
   GLsizei width,
   GLsizei height
);
```

Parameters

x	The *x*-coordinate (in window coordinates) of the lower-left corner of the hint region rectangle.
y	The *y*-coordinate (in window coordinates) of the lower-left corner of the hint region rectangle.
width	The width of the hint region rectangle.
height	The height of the hint region rectangle.

Return value

This function does not return a value.

Remarks

The **glAddSwapHintRectWIN** function speeds up animation by reducing the amount of repainting between frames. With **glAddSwapHintRectWIN**, you specify a set of rectangular areas that you want copied when you call **SwapBuffers**. When you do not specify any rectangles with **glAddSwapHintRectWIN** before calling **SwapBuffers**, the entire framebuffer is swapped. Using **glAddSwapHintRectWIN** to copy only changed parts of the buffer can significantly increase the performance of **SwapBuffers**, especially when **SwapBuffers** is implemented in software.

The **glAddSwapHintRectWIN** function adds a rectangle to the hint region. When the PFD_SWAP_COPY flag of the **PIXELFORMATDESCRIPTOR** pixel format structure is set, **SwapBuffers** uses this region to clip the copying of the

back buffer to the front buffer. You don't specify PFD_SWAP_COPY; it is set by capable hardware. The hint region is cleared after each call to **SwapBuffers**. With some hardware configurations, **SwapBuffers** can ignore the hint region and exchange the entire buffer. **SwapBuffers** is implemented by the system, not by the application.

OpenGL maintains a separate hint region for each window. When you call **glAddSwapHintRectWIN** on any rendering contexts associated with a window, the hint rectangles are combined into a single region.

Call **glAddSwapHintRectWIN** with a bounding rectangle for each object drawn for a frame and for each rectangle cleared to erase previous frame objects.

> **NOTE**
>
> The **glAddSwapHintRectWIN** function is an extension function that is not part of the standard OpenGL library but is part of the GL_WIN_swap_hint extension. To check whether your implementation of OpenGL supports **glAddSwapHintRectWIN**, call **glGetString**(GL_EXTENSIONS). If it returns GL_WIN_swap_hint, **glAddSwapHintRectWIN** is supported. To obtain the address of an extension function, call **wglGetProcAddress**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

See also

glGetString

PIXELFORMAT

D
E
S
C
R
I
P
T
O
R

S
w
a
p
B
u
f
f
e
r
s

w
g
l
G
e
t
P
r
o
c
A
d
d
r
e
s
s

glAlphaFunc function

2/22/2020 • 2 minutes to read • Edit Online

The **glAlphaFunc** function enables your application to set the alpha test function.

Syntax

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

Parameters

f
u
n
c

The alpha comparison function. The following are the accepted symbolic constants and their meanings.

VALUE	MEANING
GL_NEVER	Never passes.
GL_LESS	Passes if the incoming alpha value is less than the reference value.
GL_EQUAL	Passes if the incoming alpha value is equal to the reference value.
GL_LEQUAL	Passes if the incoming alpha value is less than or equal to the reference value.
GL_GREATER	Passes if the incoming alpha value is greater than the reference value.

VALUE	MEANING
GL_NOTEQUAL	Passes if the incoming alpha value is not equal to the reference value.
GL_GEQUAL	Passes if the incoming alpha value is greater than or equal to the reference value.
GL_ALWAYS	Always passes. This is the default.

ref

The reference value to which incoming alpha values are compared. This value is clamped to the range 0 through 1, where 0 represents the lowest possible alpha value and 1 the highest possible value. The default reference is 0.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*func* was not an accepted value.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*func* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The alpha test discards fragments depending on the outcome of a comparison between the incoming fragments' alpha values and a constant reference value. The **glAlphaFunc** function specifies the reference and comparison function. The comparison is performed only if alpha testing is enabled. (For more information on GL_ALPHA_TEST, see **glEnable**.)

The *func* and *ref* parameters specify the conditions under which the pixel is drawn. The incoming alpha value is compared to *ref* using the function specified by *func*. If the comparison passes, the incoming fragment is drawn, conditional on subsequent stencil and depth-buffer tests. If the comparison fails, no change is made to the framebuffer at that pixel location.

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

Alpha testing is done only in RGBA mode.

The following functions retrieve information related to the **glAlphaFunc** function:

glGet with argument GL_ALPHA_TEST_FUNC

glGet with argument GL_ALPHA_TEST_REF

glIsEnabled with argument GL_ALPHA_TEST

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glBlendFunc

glClea

r

glDepthFunc

glEnable

glEnd

glGet

glIsEnabled

glStenc

i
l
F
u
n
c

glAreTexturesResident function

2/22/2020 • 2 minutes to read • Edit Online

The **glAreTexturesResident** function determines whether specified texture objects are resident in texture memory.

Syntax

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

Parameters

n

The number of textures to be queried.

textures

The address of an array containing the names of the textures to be queried.

residences

The address of an array in which the texture residence status is returned. The residence status of a texture named by an element of *textures* is returned in the corresponding element of *residences*.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	*n* was a negative value, an element in *textures* was zero, or an element in *textures* did not contain a texture identifier.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	*n* was a negative value, an element in *textures* was zero, or an element in *textures* did not contain a texture identifier.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

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

The **glAreTexturesResident** function queries the texture residence status of the *n* textures named by the elements of *textures*. If all the named textures are resident, **glAreTexturesResident** returns GL_TRUE, and the contents of *residences* are undisturbed. If any of the named textures are not resident, **glAreTexturesResident** returns GL_FALSE, and detailed status is returned in the *n* elements of *residences*.

If an element of *residences* is GL_TRUE, then the texture named by the corresponding element of *textures* is

resident in texture memory.

To query the residence status of a single bound texture, call **glGetTexParameter** with the *target* parameter set to the target texture to which the target is bound and set the *pname* parameter to GL_TEXTURE_RESIDENT. You must use this method to query the resident status of a default texture.

You cannot include **glAreTexturesResident** in display lists.

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

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

> **NOTE**
> The **glAreTexturesResident** function is only available in OpenGL version 1.1 or later.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glBind

T
e
x
t
u
r
e

g
l
E
n
d

g
l
G
e
t
T
e
x
P
a
r
a
m
e
t
e
r

g
l
P
r
i
o
r
i
t
i
z
e
T
e
x
t
u
r
e
s

glTexImage1D

glTexImage2D

glArrayElement function

2/22/2020 • 2 minutes to read • Edit Online

The **glArrayElement** function specifies the array elements used to render a vertex.

Syntax

```
void WINAPI glArrayElement(
   GLint index
);
```

Parameters

index	An index in the enabled arrays.

Return value

This function does not return a value.

Remarks

Use the **glArrayElement** function within **glBegin** and **glEnd** pairs to specify vertex and attribute data for point, line, and polygon primitives. The **glArrayElement** function specifies the data for a single vertex using vertex and attribute data located at the *index* of the enabled vertex arrays.

You can use **glArrayElement** to construct primitives by indexing vertex data, rather than by streaming through arrays of data in first-to-last order. Because **glArrayElement** specifies a single vertex only, you can explicitly specify attributes for individual primitives. For example, you can set a single normal for each individual triangle.

When you include calls to **glArrayElement** in display lists, the necessary array data, determined by the array pointers and enable values, is entered in the display list also. Array pointer and enable values are determined when display lists are created, not when display lists are executed.

You can read and cache static array data at any time with **glArrayElement**. When you modify the elements of a static array without specifying the array again, the results of any subsequent calls to **glArrayElement** are undefined.

When you call **glArrayElement** without first calling **glEnableClientState**(GL_VERTEX_ARRAY), no drawing occurs, but the attributes corresponding to enabled arrays are modified. Although no error is generated when you specify an array within **glBegin** and **glEnd** pairs, the results are undefined.

> **NOTE**
>
> The **glArrayElement** function is only available in OpenGL version 1.1 or later.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColorPointer

glDrawA

r
r
a
y
s

g
l
E
d
g
e
F
l
a
g
P
o
i
n
t
e
r

g
l
E
n
a
b
l
e
C
l
i
e
n
t
S
t
a
t
e

g
l
E
n
d

g
l
G

e
t
P
o
i
n
t
e
r
v

g
l
G
e
t
S
t
r
i
n
g

g
l
I
n
d
e
x
P
o
i
n
t
e
r

g
l
N
o
r
m
a
l
P
o
i
n
t
e
r

glTexCoordPointer

glVertexPointer

glBegin function

2/22/2020 • 4 minutes to read • Edit Online

The **glBegin** and **glend** functions delimit the vertices of a primitive or a group of like primitives.

Syntax

```
void WINAPI glBegin(
   GLenum mode
);
```

Parameters

mode

The primitive or primitives that will be created from vertices presented between **glBegin** and the subsequent **glend**. The following are accepted symbolic constants and their meanings:

VALUE	MEANING

VALUE	MEANING
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.

VALUE	MEANING
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.

VALUE	MEANING
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*, *n + 2* define triangle *n*. *N - 2* triangles are drawn.

VALUE	MEANING
GL_QUADS	Treats each group of four vertices as an independent quadrilateral. Vertices *4n - 3*, *4n - 2*, *4n - 1*, and *4n* define quadrilateral *n*. *N/4* quadrilaterals are drawn.

VALUE	MEANING
GL_QUAD_STRIP	Draws a connected group of quadrilaterals. One quadrilateral is defined for each pair of vertices presented after the first pair. Vertices *2n - 1*, *2n*, *2n + 2*, and *2n + 1* define quadrilateral *n*. *N/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.

VALUE	MEANING
GL_POLYGON	Draws a single, convex polygon. Vertices *1* through *N* define this polygon.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*mode* was set to an unacceptd value.

NAME	MEANING
GL_INVALID_OPERATION	A function other than glVertex, glColor, glIndex, glNormal, glTexCoord, glEvalCoord, glEvalPoint, glMaterial, glEdgeFlag, **glCallList**, or **glCallLists** was called between **glBegin** and the corresponding **glend**. The function **glend** was called before the corresponding **glBegin** was called, or **glBegin** was called within a **glBegin/glend** sequence.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*mode* was set to an unacceptd value.

NAME	MEANING
GL_INVALID_OPERATION	A function other than glVertex, glColor, glIndex, glNormal, glTexCoord, glEvalCoord, glEvalPoint, glMaterial, glEdgeFlag, **glCallList**, or **glCallLists** was called between **glBegin** and the corresponding **glend**. The function **glend** was called before the corresponding **glBegin** was called, or **glBegin** was called within a **glBegin/glend** sequence.

Remarks

The **glBegin** and **glend** functions delimit the vertices that define a primitive or a group of like primitives. The **glBegin** function accepts a single argument that specifies which of ten primitives the vertices compose. Taking *n* as an integer count starting at one, and *N* as the total number of vertices specified, the interpretations are as follows:

- You can use only

a subset of OpenGL functions between **glBegin** and **glend**. The functions you can use are:

glVertex

glColor

glIndex

glNormal

glTexCoord

glEvalCoord

glEvalPoint

glMaterial

glEdgeFlag

You can also use **glCallList** or **glCallLists** to execute display lists that include only the preceding functions. If any other OpenGL function is called between **glBegin** and **glend**, the error flag is set and the function is ignored.

- Regardless of the value chosen for *mode* in **glBegin**, there is no limit to the number of vertices you can define between **glBegin** and **glend**. Lines, triangles, quadrilaterals, and polygons that are incompletely specified are not drawn. Incomplete

specification results when either too few vertices are provided to specify even a single primitive or when an incorrect multiple of vertices is specified. The incomplete primitive is ignored; the complete primitives are drawn.

- The minimum specification of vertices for each primitive is:

MINIMUM NUMBER OF VERTICES	TYPE OF PRIMITIVE
1	point
2	line
3	triangle
4	quadrilateral
3	polygon

- Modes that require a certain multiple of vertices are GL_LINES (2), GL_TRIANGLES (3), GL_QUADS (4), and GL_QUAD_STRIP (2).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glCallList

glCallLists

glColo

r

glEdgeFlag

glEnd

glEvalCoord

glEvalPoint

glIndex

glMate

r
i
a
l

g
l
N
o
r
m
a
l

g
l
T
e
x
C
o
o
r
d

g
l
V
e
r
t
e
x

glBindTexture function

2/22/2020 • 2 minutes to read • Edit Online

The **glBindTexture** function enables the creation of a named texture that is bound to a texture target.

Syntax

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

Parameters

target	The target to which the texture is bound. Must have the value GL_TEXTURE_1D or GL_TEXTURE_2D.
texture	The name of a texture; the texture name cannot currently be in use.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	The parameter *target* was not an accepted value.

NAME	MEANING
GL_INVALID_OPERATION	The parameter *texture* did not have the same dimensionality as *target*, or the function was called between a call to glBegin and the corresponding call to glEnd.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	The parameter *target* was not an accepted value.
GL_INVALID_OPERATION	The parameter *texture* did not have the same dimensionality as *target*, or the function was called between a call to glBegin and the corresponding call to glEnd.

Remarks

The **glBindTexture** function enables you to create a named texture. Calling **glBindTexture** with *target* set to GL_TEXTURE_1D or GL_TEXTURE_2D, and *texture* set to the name of the new texture you have created binds the texture name to the appropriate texture target. When a texture is bound to a target, the previous binding for that target is no longer in effect.

Texture names are unsigned integers with the value zero reserved to represent the default texture for each texture target. Texture names and the corresponding texture contents are local to the shared display-list space of the current OpenGL rendering context; two rendering contexts share texture names only if they also share display lists. You can generate a set of new texture names using glGenTextures.

When a texture is first bound, it assumes the dimensionality of its texture target; a texture bound to GL_TEXTURE_1D becomes one-dimensional and a texture bound to GL_TEXTURE_2D becomes two-dimensional. Operations you perform on a texture target also affect a texture bound to the target. When you query a texture target, the return value is the state of the texture bound to it. Texture targets become aliases for textures currently bound to them.

When you bind a texture with **glBindTexture**, the binding remains active until a different texture is bound to the same target or you delete the bound texture with the glDeleteTextures function. Once you create a named

texture you can bind it to a texture target that has the same dimensionality as often as needed.

It is usually much faster to use **glBindTexture** to bind an existing named texture to one of the texture targets than it is to reload the texture image using glTexImage1D or glTexImage2D. For additional control of texturing performance, use glPrioritizeTextures.

You can include calls to **glBindTexture** in display lists.

> **NOTE**
> The **glBindTexture** function is only available in OpenGL version 1.1 or later.

The following functions retrieve information related to **glBindTexture**:

- glGet with argument GL_TEXTURE_1D_BINDING

glGet with argument GL_TEXTURE_2D_BINDING

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glAreTextures

R
e
s
i
d
e
n
t

g
l
D
e
l
e
t
e
T
e
x
t
u
r
e
s

g
l
G
e
n
T
e
x
t
u
r
e
s

g
l
G
e
t

g
l
G
e
t
T
e

x
P
a
r
a
m
e
t
e
r

g
l
I
s
T
e
x
t
u
r
e

g
l
P
r
i
o
r
i
t
i
z
e
T
e
x
t
u
r
e
s

g
l
T
e
x
I
m
a
g

e
1
D

g
l
T
e
x
I
m
a
g
e
2
D

g
l
T
e
x
P
a
r
a
m
e
t
e
r

glBitmap function

2/22/2020 • 2 minutes to read • Edit Online

The **glBitmap** function draws a bitmap.

Syntax

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

Parameters

width The pixel width of the bitmap image.

height The pixel height of the bitmap image.

xorig

The *x* location of the origin in the bitmap image. The origin is measured from the lower-left corner of the bitmap, with right and up directions being the positive axes.

yorig

The *y* location of the origin in the bitmap image. The origin is measured from the lower-left corner of the bitmap, with right and up directions being the positive axes.

x The *x* offset to be added to the current raster position after the bitmap is drawn.

m
o
v
e

y
m
o
v
e

The *y* offset to be added to the current raster position after the bitmap is drawn.

b
i
t
m
a
p

The address of the bitmap image.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_ INV ALI D_V AL UE	*width* or *height* is negative.
GL_ INV ALI D_ OP ERA TIO N	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	*width* or *height* is negative.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

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

The bitmap image is interpreted like image data for the **glDrawPixels** function, with *width* and *height* corresponding to the width and height arguments of that function, and with *type* set to GL_BITMAP and *format* set to GL_COLOR_INDEX. Modes you specify using **glPixelStore** affect the interpretation of bitmap image data; modes you specify using **glPixelTransfer** do not.

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

$x_w = x_r\ x?$

$y_w = y_r\ y?$

In these coordinates, (x_r, y_r) is the raster position, and $(x?, y?)$ is the bitmap origin. Fragments are then generated for each pixel corresponding to a 1 in the bitmap image. These fragments are generated using the current raster *z*-coordinate, color or color index, and current raster texture coordinates. They are then treated just as if they had been generated by a point, line, or polygon, including texture mapping, fogging, and all per-fragment operations such as alpha and depth testing.

After the bitmap has been drawn, the *x* and *y* coordinates of the current raster position are offset by *xmove* and *ymove*. No change is made to the *z*-coordinate of the current raster position, or to the current raster color, index, or texture coordinates.

The following functions retrieve information related to the **glBitmap** function:

glGet with argument GL_CURRENT_RASTER_POSITION

glGet with argument GL_CURRENT_RASTER_COLOR

glGet with argument GL_CURRENT_RASTER_INDEX

glGet with argument GL_CURRENT_RASTER_TEXTURE_COORDS

glGet with argument GL_CURRENT_RASTER_POSITION_VALID

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glDrawPixels

glEnd

glP

i
x
e
l
S
t
o
r
e

g
l
P
i
x
e
l
T
r
a
n
s
f
e
r

g
l
R
a
s
t
e
r
P
o
s

glBlendFunc function

2/22/2020 • 4 minutes to read • Edit Online

The **glBlendFunc** function specifies pixel arithmetic.

Syntax

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

Parameters

s
f
a
c
t
o
r

Specifies how the red, green, blue, and alpha source-blending factors are computed. Nine symbolic constants are accepted: GL_ZERO, GL_ONE, GL_DST_COLOR, GL_ONE_MINUS_DST_COLOR, GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA, and GL_SRC_ALPHA_SATURATE.

d
f
a
c
t
o
r

Specifies how the red, green, blue, and alpha destination-blending factors are computed. Eight symbolic constants are accepted: GL_ZERO, GL_ONE, GL_SRC_COLOR, GL_ONE_MINUS_SRC_COLOR, GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_DST_ALPHA, and GL_ONE_MINUS_DST_ALPHA.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	Either *sfactor* or *dfactor* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	Either *sfactor* or *dfactor* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

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

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

In the table and in subsequent equations, source and destination color components are referred to as (R? , G? , B? , A?) and (R_d , G_d , B_d , A_d). They are understood to have integer values between zero and (k_R , k_G , k_R , k_A),

where

$k_R = 2^m R - 1$

$k_G = 2^m G - 1$

$k_B = 2^m B - 1$

$k_A = 2^m A - 1$

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

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

PARAMETER	(F_R , F_G , F_B , F_A)
GL_ZERO	(0,0,0,0)
GL_ONE	(1,1,1,1)
GL_SRC_COLOR	($R? / k_R$, $G? / k_G$, $B? / k_B$, $A? / k_A$)
GL_ONE_MINUS_SRC_COLOR	(1,1,1,1) - ($R? / k_R$, $G? / k_G$, $B? / k_B$, $A? / k_A$)
GL_DST_COLOR	(R_d / k_R , G_d / k_G , B_d / k_B , A_d / k_A)
GL_ONE_MINUS_DST_COLOR	(1,1,1,1) - (R_d / k_R , G_d / k_G , B_d / k_B , A_d / k_A)
GL_SRC_ALPHA	($A? / k_A$, $A? / k_A$, $A? / k_A$, $A? / k_A$)
GL_ONE_MINUS_SRC_ALPHA	(1,1,1,1) - ($A? / k_A$, $A? / k_A$, $A? / k_A$, $A? / k_A$)
GL_DST_ALPHA	(A_d / k_A , A_d / k_A , A_d / k_A , A_d / k_A)
GL_ONE_MINUS_DST_ALPHA	(1,1,1,1) - (A_d / k_A , A_d / k_A , A_d / k_A , A_d / k_A)
GL_SRC_ALPHA_SATURATE	($i,i,i,$ 1)

In the table,

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

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

$R(d) = \min(k_R , R? s_R + R_d d_R)$

$G(d) = \min(k_G , G? s_G + G_d d_G)$

$B(d) = \min(k_B , B? s_B + B_d d_B)$

$A(d) = \min(k_A , A? s_A + A_d d_A)$

Despite the apparent precision of the above equations, blending arithmetic is not exactly specified, because blending operates with imprecise integer color values. However, a blend factor that should be equal to one is guaranteed not to modify its multiplicand, and a blend factor equal to zero reduces its multiplicand to zero.

Thus, for example, when *sfactor* is GL_SRC_ALPHA, *dfactor* is GL_ONE_MINUS_SRC_ALPHA, and *A*? is equal to k_A, the equations reduce to simple replacement:

$R_d = R?$

$G_d = G?$

$B_d = B?$

$A_d = A?$

Examples

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

You can also use **glBlendFunc**(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA) for rendering antialiased points and lines in arbitrary order.

To optimize polygon antialiasing, use **glBlendFunc**(GL_SRC_ALPHA_SATURATE, GL_ONE) with polygons sorted from nearest to farthest. (See the GL_POLYGON_SMOOTH argument in **glEnable** for information on polygon antialiasing.) Destination alpha bitplanes, which must be present for this blend function to operate correctly, store the accumulated coverage.

Incoming (source) alpha is a material opacity, ranging from 1.0 (K_A), representing complete opacity, to 0.0 (0), representing complete transparency.

When you enable more than one color buffer for drawing, each enabled buffer is blended separately, and the contents of the buffer is used for destination color. (See **glDrawBuffer**.)

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

The following functions retrieve information related to **glBlendFunc**:

glGet with argument GL_BLEND_SRC

glGet with argument GL_BLEND_DST

glIsEnabled with argument GL_BLEND

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glAlphaFunc

glBegin

glClear

glDisable

glDraw

B
u
f
f
e
r

g
l
E
n
a
b
l
e

g
l
G
e
t

g
l
I
s
E
n
a
b
l
e
d

g
l
L
o
g
i
c
O
p

g
l
S
t
e
n
c
i
l

F
u
n
c

glCallList function

2/22/2020 • 2 minutes to read • Edit Online

The **glCallList** function executes a display list.

Syntax

```
void WINAPI glCallList(
   GLuint list
);
```

Parameters

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

Return value

This function does not return a value.

Remarks

Invoking the **glCallList** function begins execution of the named display list. The functions saved in the display list are executed in order, just as if you called them without using a display list. If *list* has not been defined as a display list, **glCallList** is ignored.

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

The OpenGL state is not saved and restored across a call to **glCallList**. Thus, changes made to the OpenGL state during the execution of a display list remain after execution of the display list is completed. To preserve the OpenGL state across **glCallList** calls, use **glPushAttrib**, **glPopAttrib**, **glPushMatrix**, and **glPopMatrix**.

You can execute display lists between a call to **glBegin** and the corresponding call to **glEnd**, as long as the display list includes only functions that are allowed in this interval.

The following functions retrieve information related to **glCallList**:

glGet with argument GL_MAX_LIST_NESTING

glIsList

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib
DLL	Ope ngl3 2.dll

See also

glBegin

glCallLists

glDeleteList

s

glEnd

glGenLists

glGet

glIsList

glNewList

glPopAttri

b

g
l
P
o
p
M
a
t
r
i
x

g
l
P
u
s
h
A
t
t
r
i
b

g
l
P
u
s
h
M
a
t
r
i
x

glCallLists function

2/22/2020 • 3 minutes to read • Edit Online

The **glCallLists** function executes a list of display lists.

Syntax

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

Parameters

n

The number of display lists to be executed.

type

The type of values in *lists*. The following symbolic constants are accepted.

VALUE	MEANING
GL_BYTE	The *lists* parameter is treated as an array of signed bytes, each in the range -128 through 127.
GL_UNSIGNED_BYTE	The *lists* parameter is treated as an array of unsigned bytes, each in the range 0 through 255.
GL_SHORT	The *lists* parameter is treated as an array of signed 2-byte integers, each in the range -32768 through 32767.
GL_UNSIGNED_SHORT	The *lists* parameter is treated as an array of unsigned 2-byte integers, each in the range 0 through 65535.

VALUE	MEANING
GL_INT	The *lists* parameter is treated as an array of signed 4-byte integers.
GL_UNSIGNED_INT	The *lists* parameter is treated as an array of unsigned 4-byte integers.
GL_FLOAT	The *lists* parameter is treated as an array of 4-byte, floating-point values.
GL_2_BYTES	The *lists* parameter is treated as an array of unsigned bytes. Each pair of bytes specifies a single display-list name. The value of the pair is computed as 256 times the unsigned value of the first byte plus the unsigned value of the second byte.
GL_3_BYTES	The *lists* parameter is treated as an array of unsigned bytes. Each triplet of bytes specifies a single display list name. The value of the triplet is computed as 65536 times the unsigned value of the first byte, plus 256 times the unsigned value of the second byte, plus the unsigned value of the third byte.
GL_4_BYTES	The *lists* parameter is treated as an array of unsigned bytes. Each quadruplet of bytes specifies a single display list name. The value of the quadruplet is computed as 16777216 times the unsigned value of the first byte, plus 65536 times the unsigned value of the second byte, plus 256 times the unsigned value of the third byte, plus the unsigned value of the fourth byte.

lists

The address of an array of name offsets in the display list. The pointer type is void because the offsets can be bytes, shorts, ints, or floats, depending on the value of *type*.

Return value

This function does not return a value.

Remarks

The **glCallLists** function causes each display list in the list of names passed as *lists* to be executed. As a result, the functions saved in each display list are executed in order, just as if they were called without using a display list. Names of display lists that have not been defined are ignored.

The **glCallLists** function provides an efficient means for executing display lists. The *n* parameter specifies the number of lists with various name formats (specified by the *type* parameter) **glCallLists** executes.

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

The glListBase function makes an additional level of indirection available. The **glListBase** function specifies an unsigned offset that is added to each display list name specified in *lists* before that display list is executed.

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

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

You can execute display lists between a call to glBegin and the corresponding call to glEnd, as long as the display list includes only functions that are allowed in this interval.

The following functions retrieve information related to the **glCallLists** function:

glGet with argument GL_LIST_BASE

glGet with argument GL_MAX_LIST_NESTING

glIsList

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glBegin

glCallList

glDeleteLists

glEnd

glGen

L
i
s
t
s

g
l
G
e
t

g
l
I
s
L
i
s
t

g
l
L
i
s
t
B
a
s
e

g
l
N
e
w
L
i
s
t

g
l
P
o
p
A
t
t
r
i
b

glPopMatrix

glPushAttrib

glPushMatrix

glClear function

2/22/2020 • 2 minutes to read • Edit Online

The **glClear** function clears buffers to preset values.

Syntax

```
void WINAPI glClear(
   GLbitfield mask
);
```

Parameters

mask Bitwise OR operators of masks that indicate the buffers to be cleared. The four masks are as follows.

VALUE	MEANING
GL_COLOR_BUFFER_BIT	The buffers currently enabled for color writing.
GL_DEPTH_BUFFER_BIT	The depth buffer.
GL_ACCUM_BUFFER_BIT	The accumulation buffer.

VALUE	MEANING
GL_ STE NCI L_B UFF ER_ BIT	The stencil buffer.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_ INV ALI D_V AL UE	Any bit other than the four defined bits was set in *mask*.
GL_ INV ALI D_ OP ERA TIO N	The function was called between a call to glBegin and the corresponding call to glEnd.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_ INV ALI D_V AL UE	Any bit other than the four defined bits was set in *mask*.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glClear** function sets the bitplane area of the window to values previously selected by **glClearColor**, **glClearIndex**, **glClearDepth**, **glClearStencil**, and **glClearAccum**. You can clear multiple color buffers simultaneously by selecting more than one buffer at a time using **glDrawBuffer**.

The pixel-ownership test, the scissor test, dithering, and the buffer writemasks affect the operation of **glClear**. The scissor box bounds the cleared region. The **glClear** function ignores the alpha function, blend function, logical operation, stenciling, texture mapping, and *z*-buffering.

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

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

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

The following functions retrieve information related to **glClear**:

glGet with argument GL_ACCUM_CLEAR_VALUE

glGet with argument GL_DEPTH_CLEAR_VALUE

glGet with argument GL_INDEX_CLEAR_VALUE

glGet with argument GL_COLOR_CLEAR_VALUE

glGet with argument GL_STENCIL_CLEAR_VALUE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib

DLL	Opengl32.dll

See also

glClearAccum

glClearColor

glClearDepth

glCle

a
r
I
n
d
e
x

g
l
C
l
e
a
r
S
t
e
n
c
i
l

g
l
D
r
a
w
B
u
f
f
e
r

g
l
G
e
t

g
l
S
c
i
s
s
o
r

glClearAccum function

2/22/2020 • 2 minutes to read • Edit Online

The **glClearAccum** function specifies the clear values for the accumulation buffer.

Syntax

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

Parameters

red	The red value used when the accumulation buffer is cleared. The default value is zero.
green	The green value used when the accumulation buffer is cleared. The default value is zero.
blue	The blue value used when the accumulation buffer is cleared. The default value is zero.
alpha	The alpha value used when the accumulation buffer is cleared. The default value is zero.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_ INV ALI D_ OP ERA TIO N	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_ INV ALI D_ OP ERA TIO N	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

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

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

The following function retrieves information related to **glClearAccum**:

glGet with argument GL_ACCUM_CLEAR_VALUE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glBegin

glClear

glEnd

glGet

glClearColor function

2/22/2020 • 2 minutes to read • Edit Online

The **glClearColor** function specifies clear values for the color buffers.

Syntax

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

Parameters

red	The red value that **glClear** uses to clear the color buffers. The default value is zero.
green	The green value that **glClear** uses to clear the color buffers. The default value is zero.
blue	The blue value that **glClear** uses to clear the color buffers. The default value is zero.
alpha	The alpha value that **glClear** uses to clear the color buffers. The default value is zero.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_ INV ALI D_ OP ERA TIO N	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_ INV ALI D_ OP ERA TIO N	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

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

The following functions retrieve information related to the **glClearColor** function:

glGet with argument GL_ACCUM_CLEAR_VALUE

glGet with argument GL_COLOR_CLEAR_VALUE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glBegin

glClear

glEnd

glGet

glClearDepth function

2/22/2020 • 2 minutes to read • Edit Online

The **glClearDepth** function specifies the clear value for the depth buffer.

Syntax

```
void WINAPI glClearDepth(
   GLclampd depth
);
```

Parameters

d *e* *p* *t* *h*	The depth value used when the depth buffer is cleared.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Remarks

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

The following function retrieves information related to the **glClearDepth** function:

glGet with argument GL_DEPTH_CLEAR_VALUE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

gl

C
l
e
a
r

g
l
E
n
d

g
l
G
e
t

glClearIndex function

2/22/2020 • 2 minutes to read • Edit Online

The **glClearIndex** function specifies the clear value for the color-index buffers.

Syntax

```
void WINAPI glClearIndex(
   GLfloat c
);
```

Parameters

c	The index used when the color-index buffers are cleared. The default value is zero.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Remarks

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

The following functions retrieve information related to **glClearIndex**:

glGet with argument GL_INDEX_CLEAR_VALUE

glGet with argument GL_INDEX_BITS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glClear

glE

n
d

g
l
G
e
t

glClearStencil function

2/22/2020 • 2 minutes to read • Edit Online

The **glClearStencil** function specifies the clear value for the stencil buffer.

Syntax

```
void WINAPI glClearStencil(
   GLint s
);
```

Parameters

s	The index used when the stencil buffer is cleared. The default value is zero.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

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

The following functions retrieve information related to the **glClearStencil** function:

glGet with argument GL_STENCIL_CLEAR_VALUE

glGet with argument GL_STENCIL_BITS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glClear

glEnd

g
l
G
e
t

glClipPlane function

2/22/2020 • 2 minutes to read • Edit Online

The **glClipPlane** function specifies a plane against which all geometry is clipped.

Syntax

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

Parameters

plane

The clipping plane that is being positioned. Symbolic names of the form GL_CLIP_PLANE*i*, where *i* is an integer between 0 and GL_MAX_CLIP_PLANES - 1, are accepted.

equation

The address of an array of four double-precision floating-point values. These values are interpreted as a plane equation.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*plane* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*plane* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

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

The **glClipPlane** function specifies a half-space using a four-component plane equation. When you call **glClipPlane**,*equation* is transformed by the inverse of the modelview matrix and stored in the resulting eye coordinates. Subsequent changes to the modelview matrix have no effect on the stored plane-equation components. If the dot product of the eye coordinates of a vertex with the stored plane equation components is

positive or zero, the vertex is in with respect to that clipping plane. Otherwise, it is out.

Use the **glEnable** and **glDisable** functions to enable and disable clipping planes. Call clipping planes with the argument GL_CLIP_PLANE*i*, where *i* is the plane number.

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

It is always the case that GL_CLIP_PLANE*i* = GL_CLIP_PLANE0 + *i*.

The following functions retrieve information related to **glClipPlane**:

glGetClipPlane

glIsEnabled with argument GL_CLIP_PLANE *i*

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glDisable

glEnable

glEnd

glGetClipPlane

glIsEnabled

glColor Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions set the current color:

- glColor3b
- glColor3d
- glColor3f
- glColor3i
- glColor3s
- glColor3ub
- glColor3ui
- glColor3us
- glColor4b
- glColor4d
- glColor4f
- glColor4i
- glColor4s
- glColor4ub
- glColor4ui
- glColor4us
- glColor3bv
- glColor3dv
- glColor3fv
- glColor3iv
- glColor3sv
- glColor3ubv
- glColor3uiv
- glColor3usv
- glColor4bv
- glColor4dv
- glColor4fv
- glColor4iv
- glColor4sv
- glColor4ubv
- glColor4uiv
- glColor4usv

glColor3b function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color.

Syntax

```
void WINAPI glColor3b(
   GLbyte red,
   GLbyte green,
   GLbyte blue
);
```

Parameters

red	The new red value for the current color.
green	The new green value for the current color.
blue	The new blue value for the current color.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGetBoolean

n
v
,
g
l
G
e
t
D
o
u
b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
I
n
d
e
x

glColor3bv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color from an already existing array of color values.

Syntax

```
void WINAPI glColor3bv(
   const GLbyte *v
);
```

Parameters

v	A pointer to an array that contains red, green, and blue values.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGetBoolean v, glGetDou

b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
I
n
d
e
x

glColor3d function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color.

Syntax

```
void WINAPI glColor3d(
   GLdouble red,
   GLdouble green,
   GLdouble blue
);
```

Parameters

red	The new red value for the current color.
green	The new green value for the current color.
blue	The new blue value for the current color.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGetBoolean

n
v
,
g
l
G
e
t
D
o
u
b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
I
n
d
e
x

glColor3dv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color from an already existing array of color values.

Syntax

```
void WINAPI glColor3dv(
   const GLdouble *v
);
```

Parameters

v	A pointer to an array that contains red, green, and blue values.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGetBoolean v, glGetDou

b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
I
n
d
e
x

glColor3f function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color.

Syntax

```
void WINAPI glColor3f(
   GLfloat red,
   GLfloat green,
   GLfloat blue
);
```

Parameters

red	The new red value for the current color.
green	The new green value for the current color.
blue	The new blue value for the current color.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGetBoolean

n
v
,
g
l
G
e
t
D
o
u
b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
I
n
d
e
x

glColor3fv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color from an already existing array of color values.

Syntax

```
void WINAPI glColor3fv(
   const GLfloat *v
);
```

Parameters

v	A pointer to an array that contains red, green, and blue values.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGetBoolean v, glGetDou

b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
I
n
d
e
x

glColor3i function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color.

Syntax

```
void WINAPI glColor3i(
   GLint red,
   GLint green,
   GLint blue
);
```

Parameters

red	The new red value for the current color.
green	The new green value for the current color.
blue	The new blue value for the current color.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGetBoolean

n
v
,
g
l
G
e
t
D
o
u
b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
I
n
d
e
x

glColor3iv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color from an already existing array of color values.

Syntax

```
void WINAPI glColor3iv(
   const GLint *v
);
```

Parameters

v	A pointer to an array that contains red, green, and blue values.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGetBoolean v, glGetDou

b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
I
n
d
e
x

glColor3s function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color.

Syntax

```
void WINAPI glColor3s(
   GLshort red,
   GLshort green,
   GLshort blue
);
```

Parameters

red	The new red value for the current color.
green	The new green value for the current color.
blue	The new blue value for the current color.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGetBoolean

n
v
,
g
l
G
e
t
D
o
u
b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
I
n
d
e
x

glColor3sv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color from an already existing array of color values.

Syntax

```
void WINAPI glColor3sv(
   const GLshort *v
);
```

Parameters

v	A pointer to an array that contains red, green, and blue values.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGetBoolean v, glGetDou

b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
I
n
d
e
x

glColor3ub function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color.

Syntax

```
void WINAPI glColor3ub(
   GLubyte red,
   GLubyte green,
   GLubyte blue
);
```

Parameters

red	The new red value for the current color.
green	The new green value for the current color.
blue	The new blue value for the current color.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGetBoolean

n
v
,
g
l
G
e
t
D
o
u
b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
I
n
d
e
x

glColor3ubv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color from an already existing array of color values.

Syntax

```
void WINAPI glColor3ubv(
   const GLubyte *v
);
```

Parameters

v	A pointer to an array that contains red, green, and blue values.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGetBoolean v, glGetDou

b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
I
n
d
e
x

glColor3ui function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color.

Syntax

```
void WINAPI glColor3ui(
   GLuint red,
   GLuint green,
   GLuint blue
);
```

Parameters

red	The new red value for the current color.
green	The new green value for the current color.
blue	The new blue value for the current color.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGetBoolean

n
v
,
g
l
G
e
t
D
o
u
b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
I
n
d
e
x

glColor3uiv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color from an already existing array of color values.

Syntax

```
void WINAPI glColor3uiv(
   const GLuint *v
);
```

Parameters

v	A pointer to an array that contains red, green, and blue values.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGetBoolean v, glGetDou

b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
I
n
d
e
x

glColor3us function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color.

Syntax

```
void WINAPI glColor3us(
   GLushort red,
   GLushort green,
   GLushort blue
);
```

Parameters

red	The new red value for the current color.
green	The new green value for the current color.
blue	The new blue value for the current color.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGetBoolean

n
v
,
g
l
G
e
t
D
o
u
b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
I
n
d
e
x

glColor3usv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color from an already existing array of color values.

Syntax

```
void WINAPI glColor3usv(
   const GLushort *v
);
```

Parameters

v	A pointer to an array that contains red, green, and blue values.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGetBoolean v, glGetDou

b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
I
n
d
e
x

glColor4b function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color.

Syntax

```
void WINAPI glColor4b(
   GLbyte red,
   GLbyte green,
   GLbyte blue,
   GLbyte alpha
);
```

Parameters

red	The new red value for the current color.
green	The new green value for the current color.
blue	The new blue value for the current color.
alpha	The new alpha value for the current color.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGe

t
B
o
o
l
e
a
n
v
,
g
l
G
e
t
D
o
u
b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
l

n
d
e
x

glColor4bv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color from an already existing array of color values.

Syntax

```
void WINAPI glColor4bv(
   const GLbyte *v
);
```

Parameters

v	A pointer to an array that contains red, green, blue, and alpha values.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGetBoolean v, glGetDou

b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
I
n
d
e
x

glColor4d function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color.

Syntax

```
void WINAPI glColor4d(
   GLdouble red,
   GLdouble green,
   GLdouble blue,
   GLdouble alpha
);
```

Parameters

red	The new red value for the current color.
green	The new green value for the current color.
blue	The new blue value for the current color.
alpha	The new alpha value for the current color.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGe

t
B
o
o
l
e
a
n
v
,
g
l
G
e
t
D
o
u
b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
l

n
d
e
x

glColor4dv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color from an already existing array of color values.

Syntax

```
void WINAPI glColor3dv(
   const GLdouble *v
);
```

Parameters

v	A pointer to an array that contains red, green, blue, and alpha values.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGetBoolean v, glGetDou

b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
I
n
d
e
x

glColor4f function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color.

Syntax

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

Parameters

red	The new red value for the current color.
green	The new green value for the current color.
blue	The new blue value for the current color.
alpha	The new alpha value for the current color.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGe

t
B
o
o
l
e
a
n
v
,
g
l
G
e
t
D
o
u
b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
l

n
d
e
x

glColor4fv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color from an already existing array of color values.

Syntax

```
void WINAPI glColor4fv(
   const GLfloat *v
);
```

Parameters

v	A pointer to an array that contains red, green, blue, and alpha values.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGetBoolean v, glGetDou

b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
I
n
d
e
x

glColor4i function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color.

Syntax

```
void WINAPI glColor4i(
   GLint red,
   GLint green,
   GLint blue,
   GLint alpha
);
```

Parameters

red	The new red value for the current color.
green	The new green value for the current color.
blue	The new blue value for the current color.
alpha	The new alpha value for the current color.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGe

t
B
o
o
l
e
a
n
v
,
g
l
G
e
t
D
o
u
b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
l

n
d
e
x

glColor4iv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color from an already existing array of color values.

Syntax

```
void WINAPI glColor4iv(
   const GLint *v
);
```

Parameters

v	A pointer to an array that contains red, green, blue, and alpha values.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGetBoolean v, glGetDou

b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
I
n
d
e
x

glColor4s function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color.

Syntax

```
void WINAPI glColor4s(
   GLshort red,
   GLshort green,
   GLshort blue,
   GLshort alpha
);
```

Parameters

red	The new red value for the current color.
green	The new green value for the current color.
blue	The new blue value for the current color.
alpha	The new alpha value for the current color.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGe

t
B
o
o
l
e
a
n
v
,
g
l
G
e
t
D
o
u
b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
l

n
d
e
x

glColor4sv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color from an already existing array of color values.

Syntax

```
void WINAPI glColor4sv(
   const GLshort *v
);
```

Parameters

v	A pointer to an array that contains red, green, blue, and alpha values.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGetBoolean v, glGetDou

b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
I
n
d
e
x

glColor4ub function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color.

Syntax

```
void WINAPI glColor4ub(
   GLubyte red,
   GLubyte green,
   GLubyte blue,
   GLubyte alpha
);
```

Parameters

red	The new red value for the current color.
green	The new green value for the current color.
blue	The new blue value for the current color.
alpha	The new alpha value for the current color.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGe

t
B
o
o
l
e
a
n
v
,
g
l
G
e
t
D
o
u
b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
l

n
d
e
x

glColor4ubv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color from an already existing array of color values.

Syntax

```
void WINAPI glColor4ubv(
   const GLubyte *v
);
```

Parameters

v	A pointer to an array that contains red, green, blue, and alpha values.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGetBoolean v, glGetDou

b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
I
n
d
e
x

glColor4ui function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color.

Syntax

```
void WINAPI glColor4ui(
   GLuint red,
   GLuint green,
   GLuint blue,
   GLuint alpha
);
```

Parameters

red	The new red value for the current color.
green	The new green value for the current color.
blue	The new blue value for the current color.
alpha	The new alpha value for the current color.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGe

t
B
o
o
l
e
a
n
v
,
g
l
G
e
t
D
o
u
b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
l

n
d
e
x

glColor4uiv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color from an already existing array of color values.

Syntax

```
void WINAPI glColor4uiv(
   const GLuint *v
);
```

Parameters

v	A pointer to an array that contains red, green, blue, and alpha values.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGetBoolean v, glGetDou

b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
I
n
d
e
x

glColor4us function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color.

Syntax

```
void WINAPI glColor4us(
   GLushort red,
   GLushort green,
   GLushort blue,
   GLushort alpha
);
```

Parameters

red	The new red value for the current color.
green	The new green value for the current color.
blue	The new blue value for the current color.
alpha	The new alpha value for the current color.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGe

t
B
o
o
l
e
a
n
v
,
g
l
G
e
t
D
o
u
b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
l

n
d
e
x

glColor4usv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current color from an already existing array of color values.

Syntax

```
void WINAPI glColor4usv(
   const GLushort *v
);
```

Parameters

v	A pointer to an array that contains red, green, blue, and alpha values.

Return value

This function does not return a value.

Remarks

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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGetBoolean v, glGetDou

b
l
e
v
,
g
l
G
e
t
F
l
o
a
t
v
,
g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
I
n
d
e
x

glColorMask function

2/22/2020 • 2 minutes to read • Edit Online

The **glColorMask** function enables and disables writing of frame-buffer color components.

Syntax

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

Parameters

r
e
d

Specify whether red can or cannot be written into the framebuffer. The default values is GL_TRUE, indicating that the color component can be written.

g
r
e
e
n

Specify whether green can or cannot be written into the framebuffer. The default value is GL_TRUE, indicating that the color component can be written.

b
l
u
e

Specify whether blue can or cannot be written into the framebuffer. The default value is GL_TRUE, indicating that the color component can be written.

a
l
p
h
a

Specify whether alpha can or cannot be written into the framebuffer. The default value is GL_TRUE, indicating that the color component can be written.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_ INV ALI D_ OP ERA TIO N	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_ INV ALI D_ OP ERA TIO N	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

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

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

The following functions retrieve information related to **glColorMask**:

glGet with argument GL_COLOR_WRITEMASK

glGet with argument GL_RGBA_MODE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glDepthMask

glEnd

glGet

glIndex

glIndexMask

glStencilMask

glColorMaterial function

2/22/2020 • 2 minutes to read • Edit Online

The **glColorMaterial** function causes a material color to track the current color.

Syntax

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

Parameters

face

Specifies whether front, back, or both front and back material parameters should track the current color. Accepted values are GL_FRONT, GL_BACK, and GL_FRONT_AND_BACK. The default value is GL_FRONT_AND_BACK.

mode

Specifies which of several material parameters track the current color. Accepted values are GL_EMISSION, GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, and GL_AMBIENT_AND_DIFFUSE. The default value is GL_AMBIENT_AND_DIFFUSE.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*face* or *mode* was not an accepted value.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*face* or *mode* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glColorMaterial** function specifies which material parameters track the current color. When you enable GL_COLOR_MATERIAL, for each of the material or materials specified by *face*, the material parameter or parameters specified by *mode* track the current color at all times. Enable and disable GL_COLOR_MATERIAL with the functions **glEnable** and **glDisable**, which you call with GL_COLOR_MATERIAL as their argument. By default, GL_COLOR_MATERIAL is disabled.

With **glColorMaterial**, you can change a subset of material parameters for each vertex using only the **glColor** function, without calling **glMaterial**. If you are going to specify only such a subset of parameters for each vertex, it is better to do so with **glColorMaterial** than with **glMaterial**.

The following functions retrieve information related to **glColorMaterial**:

glGet with argument GL_COLOR_MATERIAL_PARAMETER

glGet with argument GL_COLOR_MATERIAL_FACE

glIsEnabled with argument GL_COLOR_MATERIAL

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glDisable

glEnab

l
e

g
l
E
n
d

g
l
G
e
t

g
l
I
s
E
n
a
b
l
e
d

g
l
L
i
g
h
t

g
l
L
i
g
h
t
M
o
d
e
l

g
l
M
a
t
e

r
i
a
l

glColorPointer function

2/22/2020 • 2 minutes to read • Edit Online

The **glColorPointer** function defines an array of colors.

Syntax

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

Parameters

size

The number of components per color. The value must be either 3 or 4.

type

The data type of each color component in a color array. Acceptable data types are specified with the following constants: GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, or GL_DOUBLE.

stride

The byte offset between consecutive colors. When *stride* is zero, the colors are tightly packed in the array.

pointer

A pointer to the first component of the first color element in a color array.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	*size* was not 3 or 4.
GL_INVALID_ENUM	*type* was not an accepted value.
GL_INVALID_VALUE	*stride* or *count* was negative.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	*size* was not 3 or 4.
GL_INVALID_ENUM	*type* was not an accepted value.

NAME	MEANING
GL_ INV ALI D_V AL UE	*stride* or *count* was negative.

Remarks

The **glColorPointer** function specifies the location and data format of an array of color components to use when rendering. The *stride* parameter determines the byte offset from one color to the next, enabling the packing of vertex attributes in a single array or storage in separate arrays. In some implementations, storing vertex attributes in a single array can be more efficient than the use of separate arrays.

Enabled the color array by specifying the GL_COLOR_ARRAY constant with glEnableClientState. Calling glArrayElement, glDrawElements, or glDrawArrays uses the color array that is thus enabled. By default, the color array is disabled. The **glColorPointer** calls cannot by entered in display lists.

When you specify a color array using **glColorPointer**, the values of all the function's color array parameters are saved in a client-side state, and you can cache static array elements. Because the color array parameters are in a client-side state, glPushAttrib and glPopAttrib do not save or restore the parameters' values.

Although specifying the color array within glBegin and glend pairs does not generate an error, the results are undefined.

The following functions retrieve information related to the **glColorPointer** function:

glIsEnabled with argument GL_COLOR_ARRAY

glGet with argument GL_COLOR_ARRAY_SIZE

glGet with argument GL_COLOR_ARRAY_TYPE

glGet with argument GL_COLOR_ARRAY_STRIDE

glGet with argument GL_COLOR_ARRAY_COUNT

glGetPointerv with argument GL_COLOR_ARRAY_POINTER

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glArrayElement

glBegin

glDrawArrays

glEdgeFl

a
g
P
o
i
n
t
e
r

g
l
E
n
a
b
l
e
C
l
i
e
n
t
S
t
a
t
e

g
l
E
n
d

g
l
G
e
t

g
l
G
e
t
S
t
r
i
n
g

glGetPointerv

glIndexPointer

glIsEnabled

glNormalPoi

n
t
e
r

g
l
P
o
p
A
t
t
r
i
b

g
l
P
u
s
h
A
t
t
r
i
b

g
l
T
e
x
C
o
o
r
d
P
o
i
n
t
e
r

g
l
V
e
r

t
e
x
P
o
i
n
t
e
r

glColorTableEXT function

2/22/2020 • 7 minutes to read • Edit Online

The **glColorTableEXT** function specifies the format and size of a palette for targeted paletted textures.

Syntax

```
void WINAPI glColorTableEXT(
         GLenum  target,
         GLenum  internalFormat,
         GLsizei width,
         GLenum  format,
         GLenum  type,
   const GLvoid  *data
);
```

Parameters

target

The target texture that is to have its palette changed. Must be TEXTURE_1D, TEXTURE_2D, PROXY_TEXTURE_1D, or PROXY_TEXTURE_2D.

internalFormat

The internal format and resolution of the palette. This parameter can assume one of the following symbolic values.

CONSTANT	BASE FORMAT	R BITS	G BITS	B BITS	A BITS
GL_R3_G3_B2	GL_RGB	3	3	2	

CONSTANT	BASE FORMAT	R BITS	G BITS	B BITS	A BITS
GL_RGB4	GL_RGB	4	4	4	
GL_RGB5	GL_RGB	5	5	5	
GL_RGB8	GL_RGB	8	8	8	
GL_RGB10	GL_RGB	10	10	10	
GL_RGB12	GL_RGB	12	12	12	
GL_RGB16	GL_RGB	16	16	16	
GL_RGBA2	GL_RGBA	2	2	2	2
GL_RGBA4	GL_RGBA	4	4	4	4
GL_RGB5_A1	GL_RGBA	5	5	5	1
GL_RGBA8	GL_RGBA	8	8	8	8
GL_RG10_A2	GL_RGBA	10	10	10	2
GL_RGB12	GL_RGBA	12	12	12	12
GL_RGBA16	GL_RGBA	16	16	16	16

width

The size of the palette. Must be 2n = 1 for some integer *n*.

format

The format of the pixel data. The following symbolic constants are accepted.

VALUE	MEANING

VALUE	MEANING
GL_ RG BA	Each pixel is a group of four components in this order: red, green, blue, alpha. The RGBA format is determined in this way: 1. The **glColorTableEXT** function converts floating-point values directly to an internal format with unspecified precision. Signed integer values are mapped linearly to the internal format such that the most positive representable integer value maps to 1.0, and the most negative representable integer value maps to -1.0. Unsigned integer data is mapped similarly: the largest integer value maps to 1.0, and zero maps to 0.0. 2. The **glColorTableEXT** function multiplies the resulting color values by GL_c_SCALE and adds them to GL_c_BIAS, where *c* is RED, GREEN, BLUE, and ALPHA for the respective color components. The results are clamped to the range [0,1]. 3. If GL_MAP_COLOR is **TRUE**, **glColorTableEXT** scales each color component by the size of lookup table GL_PIXEL_MAP_c_TO_c, then replaces the component by the value that it references in that table; *c* is R, G, B, or A, respectively. 4. The **glColorTableEXT** function converts the resulting RGBA colors to fragments by attaching the current raster position *z*-coordinate and texture coordinates to each pixel, then assigning *x* and *y* window coordinates to the *n*th fragment such that $x? = x_r + n \bmod width$ $y? = y_r + n / width$ where (x_r , y_r) is the current raster position. 5. These pixel fragments are then treated just like the fragments generated by rasterizing points, lines, or polygons. The **glColorTableEXT** function applies texture mapping, fog, and all the fragment operations before writing the fragments to the framebuffer.

VALUE	MEANING
GL_RED	Each pixel is a single red component. The **glColorTableEXT** function converts this component to the internal format in the same way that the red component of an RGBA pixel is, then converts it to an RGBA pixel with green and blue set to 0.0, and alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.
GL_GREEN	Each pixel is a single green component. The **glColorTableEXT** function converts this component to the internal format in the same way that the green component of an RGBA pixel is, and then converts it to an RGBA pixel with red and blue set to 0.0, and alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.
GL_BLUE	Each pixel is a single blue component. The **glColorTableEXT** function converts this component to the internal format in the same way that the blue component of an RGBA pixel is, and then converts it to an RGBA pixel with red and green set to 0.0, and alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.
GL_ALPHA	Each pixel is a single alpha component. The **glColorTableEXT** function converts this component to the internal format in the same way that the alpha component of an RGBA pixel is, and then converts it to an RGBA pixel with red, green, and blue set to 0.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.
GL_RGB	Each pixel is a group of three components in this order: red, green, blue. The **glColorTableEXT** function converts each component to the internal format in the same way that the red, green, and blue components of an RGBA pixel are. The color triple is converted to an RGBA pixel with alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.

VALUE	MEANING
GL_BGR_EXT	Each pixel is a group of three components in this order: blue, green, red. GL_BGR_EXT provides a format that matches the memory layout of Windows device-independent bitmaps (DIBs). Thus your applications can use the same data with Windows function calls and OpenGL pixel function calls.
GL_BGRA_EXT	Each pixel is a group of four components in this order: blue, green, red, alpha. GL_BGRA_EXT provides a format that matches the memory layout of Windows device-independent bitmaps (DIBs). Thus your applications can use the same data with Windows function calls and OpenGL pixel function calls.

type

The data type for *data*. The following symbolic constants are accepted: GL_UNSIGNED_BYTE, GL_BYTE, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, and GL_FLOAT.

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

VALUE	MEANING
GL_UNSIGNED_BYTE	Unsigned 8-bit integer
GL_BYTE	Signed 8-bit integer
GL_UNSIGNED_SHORT	Unsigned 16-bit integer

VALUE	MEANING
GL_SHORT	Signed 16-bit integer
GL_UNSIGNED_INT	Unsigned 32-bit integer
GL_INT	32-bit integer
GL_FLOAT	Single-precision floating-point value

data

A pointer to the paletted texture data. The data is treated as single pixels of a 1-D texture palette entry for a palette entry.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_VALUE	*width* was an invalid integer.

NAME	MEANING
GL_INVALID_ENUM	*target*, *internalFormat*, *format*, or *type* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	*width* was an invalid integer.
GL_INVALID_ENUM	*target*, *internalFormat*, *format*, or *type* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

Paletted textures are defined with a palette of colors and a set of image data that is composed of indexes to color entries of a palette (a color table).

The **glColorTableEXT** function specifies the texture palette of a targeted texture. It takes the *data* from memory and converts the data as if each palette entry is a single pixel of a 1-D texture. The **glColorTableEXT** function unpacks and converts the data and translates it into an internal format that matches the given *format* as closely as possible.

If a palette's *width* is greater than the range of the color indexes in the texture data, some of the palette entries are unused. If a palette's *width* is less than the range of the color indexes in the texture data, the most significant bits of the texture data are ignored and only the appropriate number of bits in the index are used when accessing the palette. When you specify a proxy *target* using PROXY_TEXTURE_1D or PROXY_TEXTURE_2D, the palette of the proxy texture is resized and its parameters are set but no data is transferred or accessed.

When the *target* parameter is GL_PROXY_TEXTURE_1D or GL_PROXY_TEXTURE_2D, and the implementation does not support the values specified for either *format* or *width*, **glColorTableEXT** can fail to create the requested color table. In this case, the color table is empty and all parameters retrieved will be zero. You can determine whether OpenGL supports a particular color table format and size by calling **glColorTableEXT** with a proxy target, and then calling glGetColorTableParameterivEXT or glGetColorTableParameterfvEXT to determine whether the width parameter matches that set by **glColorTableEXT**. If the retrieved width is zero, the color table request by **glColorTable** failed. If the retrieved width is not zero, you can call **glColorTable** with the real target with TEXTURE_1D or TEXTURE_2D to set the color table.

> **NOTE**
>
> The **glColorTableEXT** function is an extension function that is not part of the standard OpenGL library but is part of the GL_EXT_paletted_texture extension. To check whether your implementation of OpenGL supports **glColorTableEXT**, call glGetString(GL_EXTENSIONS). If it returns GL_EXT_paletted_texture, **glColorTableEXT** is supported. To obtain the function address of an extension function, call wglGetProcAddress.

To retrieve the actual color table data specified by the **glColorTableEXT** function, call glGetColorTableEXT. To retrieve the parameters, such as *width* and *format*, of the color table specified by the **glColorTableEXT** function, call the **glGetColorTableParameterivEXT** or **glGetColorTableParameterfvEXT** function.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

See also

g
l
B
e
g
i
n

g

lColorSubTableEXT

glEnd

glGetColorTableEXT

glGetColor

T
a
b
l
e
P
a
r
a
m
e
t
e
r
f
v
E
X
T

g
l
G
e
t
C
o
l
o
r
T
a
b
l
e
P
a
r
a
m
e
t
e
r
i
v
E
X
T

w
g

lGetProcAddress

glColorSubTableEXT function

2/22/2020 • 5 minutes to read • Edit Online

The **glColorSubTableEXT** function specifies a portion of the targeted texture's palette to be replaced.

Syntax

```
void WINAPI glColorSubTableEXT(
         GLenum  target,
         GLsizei start,
         GLsizei count,
         GLenum  format,
         GLenum  type,
   const GLvoid  *data
);
```

Parameters

target The target paletted texture that is to have its palette changed. Must be TEXTURE_1D or TEXTURE_2D.

start The starting palette index entry of the palette to be changed.

count The number of palette index entries of the palette to be changed beginning at *start*. The *count* parameter determines the range of palette index entries that are changed.

format The format of the pixel data. The following symbolic constants are accepted.

VALUE	MEANING

VALUE	MEANING
GL_ RG BA	Each pixel is a group of four components in the following order: red, green, blue, alpha. The RGBA format is determined in this way: 1. The **glColorSubTableEXT** function converts floating-point values directly to an internal format with unspecified precision. Signed integer values are mapped linearly to the internal format such that the most positive representable integer value maps to 1.0, and the most negative representable value maps to -1.0. Unsigned integer data is mapped similarly: the largest integer value maps to 1.0, and zero maps to 0.0. 2. The **glColorSubTableEXT** function multiplies the resulting color values by GL_c_SCALE and adds them to GL_c_BIAS, where *c* is RED, GREEN, BLUE, and ALPHA for the respective color components. The results are clamped to the range [0,1]. 3. If GL_MAP_COLOR is **TRUE**, **glColorSubTableEXT** scales each color component by the size of lookup table GL_PIXEL_MAP_c_TO_c, then replaces the component by the value that it references in that table; *c* is R, G, B, or A, respectively. 4. The **glColorSubTableEXT** function converts the resulting RGBA colors to fragments by attaching the current raster position *z*-coordinate and texture coordinates to each pixel, then assigning *x* and *y* window coordinates to the *n*th fragment such that $x? = x_r + n \bmod width$ $y? = y_r + n / width$ where (x_r , y_r) is the current raster position. 5. These pixel fragments are then treated just like the fragments generated by rasterizing points, lines, or polygons. The **glColorSubTableEXT** function applies texture mapping, fog, and all the fragment operations before writing the fragments to the framebuffer.

VALUE	MEANING
GL_RED	Each pixel is a single red component. The **glColorSubTableEXT** function converts this component to the internal format in the same way that the red component of an RGBA pixel is, then converts it to an RGBA pixel with green and blue set to 0.0, and alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.
GL_GREEN	Each pixel is a single green component. The **glColorSubTableEXT** function converts this component to the internal format in the same way that the green component of an RGBA pixel is, and then converts it to an RGBA pixel with red and blue set to 0.0, and alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.
GL_BLUE	Each pixel is a single blue component. The **glColorSubTableEXT** function converts this component to the internal format in the same way that the blue component of an RGBA pixel is, and then converts it to an RGBA pixel with red and green set to 0.0, and alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.
GL_ALPHA	Each pixel is a single alpha component. The **glColorSubTableEXT** function converts this component to the internal format in the same way that the alpha component of an RGBA pixel is, and then converts it to an RGBA pixel with red, green, and blue set to 0.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.
GL_RGB	Each pixel is a group of three components in this order: red, green, blue. The **glColorSubTableEXT** function converts each component to the internal format in the same way that the red, green, and blue components of an RGBA pixel are. The color triple is converted to an RGBA pixel with alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.

VALUE	MEANING
GL_BGR_EXT	Each pixel is a group of three components in this order: blue, green, red. GL_BGR_EXT provides a format that matches the memory layout of Windows device-independent bitmaps (DIBs). Thus your applications can use the same data with Windows function calls and OpenGL pixel function calls.
GL_BGRA_EXT	Each pixel is a group of four components in this order: blue, green, red, alpha. GL_BGRA_EXT provides a format that matches the memory layout of Windows device-independent bitmaps (DIBs). Thus your applications can use the same data with Windows function calls and OpenGL pixel function calls.

type

The data type for *data*. The following symbolic constants are accepted: GL_UNSIGNED_BYTE, GL_BYTE, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, and GL_FLOAT.

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

VALUE	MEANING
GL_UNSIGNED_BYTE	Unsigned 8-bit integer
GL_BYTE	Signed 8-bit integer
GL_UNSIGNED_SHORT	Unsigned 16-bit integer

VALUE	MEANING
GL_SHORT	Signed 16-bit integer
GL_UNSIGNED_INT	Unsigned 32-bit integer
GL_INT	32-bit integer
GL_FLOAT	Single-precision floating-point value

data

A pointer to the paletted texture data. The data is treated as single pixels of a 1-D texture palette entry for a palette entry.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	*start* or *count* was an invalid integer.

NAME	MEANING
GL_INVALID_ENUM	*target*, *format*, or *type* was not an accepted value.
GL_INVALID_VALUE	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	*start* or *count* was an invalid integer.
GL_INVALID_ENUM	*target*, *format*, or *type* was not an accepted value.
GL_INVALID_VALUE	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glColorSubTableEXT** function specifies portions of the current targeted texture's palette to be replaced. Unlike **glColorTableEXT**, you cannot specify the *target* parameter to be a proxy texture palette.

> **NOTE**
>
> The **glColorSubTableEXT** function is an extension function that is not part of the standard OpenGL library but is part of the GL_EXT_paletted_texture extension. To check whether your implementation of OpenGL supports **glColorSubTableEXT**, call **glGetString**(GL_EXTENSIONS). If it returns GL_EXT_paletted_texture, **glColorSubTableEXT** is supported. To obtain the function address of an extension function, call **wglGetProcAddress**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

See also

glBegin

glColorTableEXT

glEnd

gl

G
e
t
C
o
l
o
r
T
a
b
l
e
E
X
T

g
l
G
e
t
C
o
l
o
r
T
a
b
l
e
P
a
r
a
m
e
t
e
r
f
v
E
X
T

g
l
G
e
t

C
o
l
o
r
T
a
b
l
e
P
a
r
a
m
e
t
e
r
i
v
E
X
T

g
l
G
e
t
S
t
r
i
n
g

w
g
l
G
e
t
P
r
o
c
A
d
d
r
e

s

s

glCopyPixels function

2/22/2020 • 6 minutes to read • Edit Online

The **glCopyPixels** function copies pixels in the framebuffer.

Syntax

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

Parameters

x

The window x-plane coordinate of the lower-left corner of the rectangular region of pixels to be copied.

y

The window y-plane coordinate of the lower-left corner of the rectangular region of pixels to be copied.

width

The width dimension of the rectangular region of pixels to be copied. Must be nonnegative.

height

The height dimension of the rectangular region of pixels to be copied. Must be nonnegative.

type

Specifies whether **glCopyPixels** is to copy color values, depth values, or stencil values. The acceptable symbolic constants are.

VALUE	MEANING
GL_ CO LOR	The **glCopyPixels** function reads indexes or RGBA colors from the buffer currently specified as the read source buffer (see **glReadBuffer**). If OpenGL is in color-index mode: 1. Each index that is read from this buffer is converted to a fixed-point format with an unspecified number of bits to the right of the binary point. 2. Each index is shifted left by GL_INDEX_SHIFT bits, and added to GL_INDEX_OFFSET.If GL_INDEX_SHIFT is negative, the shift is to the right. In either case, zero bits fill otherwise unspecified bit locations in the result. 3. If GL_MAP_COLOR is true, the index is replaced with the value that it references in lookup table GL_PIXEL_MAP_I_TO_I. 4. Whether the lookup replacement of the index is done or not, the integer part of the index is then **AND**ed with 2^b 1, where *b* is the number of bits in a color-index buffer. If OpenGL is in RGBA mode: 1. The red, green, blue, and alpha components of each pixel that is read are converted to an internal floating-point format with unspecified precision. 2. The conversion maps the largest representable component value to 1.0, and component value zero to 0.0. 3. The resulting floating-point color values are then multiplied by GL_c_SCALE and added to GL_c_BIAS, where *c* is RED, GREEN, BLUE, and ALPHA for the respective color components. 4. The results are clamped to the range [0,1]. 5. If GL_MAP_COLOR is true, each color component is scaled by the size of lookup table GL_PIXEL_MAP_c_TO_c, and then replaced by the value that it references in that table; *c* is R, G, B, or A, respectively. The resulting indexes or RGBA colors are then converted to fragments by attaching the current raster position *z*-coordinate and texture coordinates to each pixel, and then assigning window coordinates (x_r + i, y_r + *j*), where (x_r , y_r) is the current raster position, and the pixel was the pixel in the *i* position in the *j* row. These pixel fragments are then treated just like the fragments generated by rasterizing points, lines, or polygons. Texture mapping, fog, and all the fragment operations are applied before the fragments are written to the framebuffer.

VALUE	MEANING
GL_DEPTH	Depth values are read from the depth buffer and converted directly to an internal floating-point format with unspecified precision. The resulting floating-point depth value is then multiplied by GL_DEPTH_SCALE and added to GL_DEPTH_BIAS. The result is clamped to the range [0,1]. The resulting depth components are then converted to fragments by attaching the current raster position color or color index and texture coordinates to each pixel, then assigning window coordinates (x_r + i, y_r + j), where (x_r , y_r) is the current raster position, and the pixel was the pixel in the *i* position in the *j* row. These pixel fragments are then treated just like the fragments generated by rasterizing points, lines, or polygons. Texture mapping, fog, and all the fragment operations are applied before the fragments are written to the framebuffer.
GL_STENCIL	Stencil indexes are read from the stencil buffer and converted to an internal fixed-point format with an unspecified number of bits to the right of the binary point. Each fixed-point index is then shifted left by GL_INDEX_SHIFT bits, and added to GL_INDEX_OFFSET. If GL_INDEX_SHIFT is negative, the shift is to the right. In either case, zero bits fill otherwise unspecified bit locations in the result. If GL_MAP_STENCIL is true, the index is replaced with the value that it references in lookup table GL_PIXEL_MAP_S_TO_S. Whether the lookup replacement of the index is done or not, the integer part of the index is then **AND**ed with 2^b - 1, where *b* is the number of bits in the stencil buffer. The resulting stencil indexes are then written to the stencil buffer such that the index read from the *i* location of the *j* row is written to location (x_r + *i*, y_r + j), where (x_r , y_r) is the current raster position. Only the pixel-ownership test, the scissor test, and the stencil writemask affect these writes.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*type* was not an accepted value.

NAME	MEANING
GL_INVALID_VALUE	Either *width* or height was negative.
GL_INVALID_OPERATION	*type* was GL_DEPTH and there was no depth buffer.
GL_INVALID_OPERATION	*type* was GL_STENCIL and there was no stencil buffer.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*type* was not an accepted value.

NAME	MEANING
GL_INVALID_VALUE	Either *width* or height was negative.
GL_INVALID_OPERATION	*type* was GL_DEPTH and there was no depth buffer.
GL_INVALID_OPERATION	*type* was GL_STENCIL and there was no stencil buffer.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glCopyPixels** function copies a screen-aligned rectangle of pixels from the specified framebuffer location to a region relative to the current raster position. Its operation is well defined only if the entire pixel source region is within the exposed portion of the window. Results of copies from outside the window, or from regions of the window that are not exposed, are hardware dependent and undefined.

The *x* and *y* parameters specify the window coordinates of the lower-left corner of the rectangular region to be copied. The *width* and *height* parameters specify the dimensions of the rectangular region to be copied. Both *width* and *height* must be nonnegative.

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

The **glCopyPixels** function copies values from each pixel with the lower-left corner at $(x + i, y + j)$ for $0 = i < width$ and $0 = j < height$. This pixel is said to be the *i* pixel in the *j* row. Pixels are copied in row order from the lowest to the highest row, left to right in each row.

The *type* parameter specifies whether color, depth, or stencil data is to be copied.

The rasterization described thus far assumes pixel zoom factors of 1.0. If you use **glPixelZoom** to change the *x* and *y* pixel zoom factors, pixels are converted to fragments as follows. If (x_r, y_r) is the current raster position, and a given pixel is in the *i* location in the *j* row of the source pixel rectangle, then fragments are generated for pixels whose centers are in the rectangle with corners at

$(x_r + zoom? \, i, y_r + zoom_y \, j)$

and

$(x_r + zoom? \, (i + 1), y_r + zoom_y \, (j + 1))$

where *zoom*? is the value of GL_ZOOM_X and $zoom_y$ is the value of GL_ZOOM_Y.

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

The following functions retrieve information related to **glCopyPixels**:

glGet with argument GL_CURRENT_RASTER_POSITION

glGet with argument GL_CURRENT_RASTER_POSITION_VALID

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

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

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glDepthFunc

glDrawBuffer

glDrawPixels

glEnd

glGet

glP

i
x
e
l
M
a
p

g
l
P
i
x
e
l
S
t
o
r
e

g
l
P
i
x
e
l
T
r
a
n
s
f
e
r

g
l
P
i
x
e
l
Z
o
o
m

g
l
R
a

s
t
e
r
P
o
s

g
l
R
e
a
d
B
u
f
f
e
r

g
l
R
e
a
d
P
i
x
e
l
s

g
l
S
t
e
n
c
i
l
F
u
n
c

glCopyTexImage1D function

2/22/2020 • 3 minutes to read • Edit Online

The **glCopyTexImage1D** function copies pixels from the framebuffer into a one-dimensional texture image.

Syntax

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

Parameters

target

The target for which the image data will be changed. Must have the value GL_TEXTURE_1D.

level

The level-of-detail number. Level 0 is the base image. Level *n* is the *n*th mipmap reduction image.

internalFormat

The internal format and resolution of the texture data. This parameter must be one of the following symbolic values.

CONSTANT	R BITS	G BITS	B BITS	A BITS	L BITS	I BITS
GL_ALPHA						
GL_ALPHA4				4		
GL_ALPHA8				8		
GL_ALPHA12				12		
GL_ALPHA16				16		
GL_LUMINANCE						
GL_LUMINANCE4					4	
GL_LUMINANCE8					8	
GL_LUMINANCE12					12	
GL_LUMINANCE16					16	
GL_LUMINANCE_ALPHA						
GL_LUMINANCE4_ALPHA4				4	4	
GL_LUMINANCE6_ALPHA2				2	6	
GL_LUMINANCE8_ALPHA8				8	8	
GL_LUMINANCE12_ALPHA4				4	12	
GL_LUMINANCE12_ALPHA12				12	12	
GL_LUMINANCE16_ALPHA16				16	16	

CONSTANT	R BITS	G BITS	B BITS	A BITS	L BITS	I BITS
GL_INTENSITY						
GL_INTENSITY4						4
GL_INTENSITY8						8
GL_INTENSITY12						12
GL_INTENSITY16						16
GL_RGB						
GL_R3_G3_B2	3	3	2			
GL_RGB4	4	4	4			
GL_RGB5	5	5	5			
GL_RGB8	8	8	8			
GL_RGB10	10	10	10			
GL_RGB12	12	12	12			
GL_RGB16	16	16	16			
GL_RGBA						
GL_RGBA2	2	2	2	2		
GL_RGBA4	4	4	4	4		
GL_RGB5_A1	5	5	5	1		
GL_RGBA8	8	8	8	8		
GL_RGB10_A2	10	10	10	2		
GL_RGBA12	12	12	12	12		
GL_RGBA16	16	16	16	16		

x The window x-plane coordinate of the lower-left corner of the row of pixels to be copied.

y The window y-plane coordinate of the lower-left corner of the row of pixels to be copied.

width

The width of the texture image. Must be zero or 2n + 2(*border*) for some integer *n*. The height of the texture image is 1.

border

The width of the border. Must be either zero or 1.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* was not an accepted value.
GL_INVALID_VALUE	*level* was less than zero or greater than log2 *max*, where *max* is the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_VALUE	*border* was not zero or 1.

NAME	MEANING
GL_INVALID_VALUE	*width* was less than zero, greater than 2 + GL_MAX_TEXTURE_SIZE, or *width* cannot be represented as 2n +(*border*) for some integer *n*.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* was not an accepted value.
GL_INVALID_VALUE	*level* was less than zero or greater than log2 *max*, where *max* is the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_VALUE	*border* was not zero or 1.

NAME	MEANING
GL_INVALID_VALUE	*width* was less than zero, greater than 2 + GL_MAX_TEXTURE_SIZE, or *width* cannot be represented as 2n +(*border*) for some integer *n*.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glCopyTexImage1D** function defines a one-dimensional texture image using pixels from the current framebuffer, rather than from main memory as is the case for **glTexImage1D**.

Using the mipmap level specified with *level*, texture arrays are defined as a pixel row aligned with the lower-left corner of the window at the coordinates specified by *x* and *y*, with a length equal to *width* + 2 * *border*. The internal format of the texture array is specified with the *internalFormat* parameter.

The **glCopyTexImage1D** function processes the pixels in a row in the same way as **glCopyPixels**, except that before the final conversion of the pixels, all pixel component values are clamped to the range [0,1] and converted to the texture's internal format for storage in the texture array. Pixel ordering is determined with lower *x* coordinates corresponding to lower texture coordinates. If any of the pixels within a specified row of the current framebuffer are outside the window associated with the current rendering context, then their values are undefined.

You cannot include calls to **glCopyTexImage1D** in display lists.

> **NOTE**
>
> The **glCopyTexImage1D** function is only available in OpenGL version 1.1 or later.

Texturing has no effect in color-index mode. The **glPixelStore** and **glPixelTransfer** functions affect texture images in exactly the way they affect **glDrawPixels**.

The following function retrieves information related to **glCopyTexImage1D**:

glIsEnabled with argument GL_TEXTURE_1D

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glCopyPixels

glCopyTexImage2D

glDraw

Pixels

glFog

glPixelStore

glPixelTransfer

glTexEnv

gl

T
e
x
G
e
n

g
l
T
e
x
I
m
a
g
e
1
D

g
l
T
e
x
I
m
a
g
e
2
D

g
l
T
e
x
P
a
r
a
m
e
t
e
r

glCopyTexImage2D function

2/22/2020 • 3 minutes to read • Edit Online

The **glCopyTexImage2D** function copies pixels from the framebuffer into a two-dimensional texture image.

Syntax

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

Parameters

target

The target to which the image data will be changed. Must have the value GL_TEXTURE_2D.

level

The level-of-detail number. Level 0 is the base image. Level *n* is the *n*th mipmap reduction image.

internalFormat

The internal format and resolution of the texture data. The values 1, 2, 3, and 4 are not accepted for *internalFormat*. The parameter can assume one of the following symbolic values.

CONSTANT	R BITS	G BITS	B BITS	A BITS	L BITS	I BITS
GL_ALPHA						
GL_ALPHA4				4		
GL_ALPHA8				8		
GL_ALPHA12				12		
GL_ALPHA16				16		
GL_LUMINANCE						
GL_LUMINANCE4					4	
GL_LUMINANCE8					8	
GL_LUMINANCE12					12	
GL_LUMINANCE16					16	
GL_LUMINANCE_ALPHA						
GL_LUMINANCE4_ALPHA4				4	4	
GL_LUMINANCE6_ALPHA2				2	6	
GL_LUMINANCE8_ALPHA8				8	8	
GL_LUMINANCE12_ALPHA4				4	12	
GL_LUMINANCE12_ALPHA12				12	12	
GL_LUMINANCE16_ALPHA16				16	16	

CONSTANT	R BITS	G BITS	B BITS	A BITS	L BITS	I BITS
GL_INTENSITY						
GL_INTENSITY4						4
GL_INTENSITY8						8
GL_INTENSITY12						12
GL_INTENSITY16						16
GL_RGB						
GL_R3_G3_B2	3	3	2			
GL_RGB4	4	4	4			
GL_RGB5	5	5	5			
GL_RGB8	8	8	8			
GL_RGB10	10	10	10			
GL_RGB12	12	12	12			
GL_RGB16	16	16	16			
GL_RGBA						
GL_RGBA2	2	2	2	2		
GL_RGBA4	4	4	4	4		
GL_RGB5_A1	5	5	5	1		
GL_RGBA8	8	8	8	8		
GL_RGB10_A2	10	10	10	2		
GL_RGBA12	12	12	12	12		
GL_RGBA16	16	16	16	16		

x

The window x-plane coordinate of the lower-left corner of the rectangular region of pixels to be copied.

y

The window y-plane coordinate of the lower-left corner of the rectangular region of pixels to be copied.

width

The width of the texture image. Must be 2n + 2 * *border* for some integer *n*.

height

The height of the texture image. Must be 2n + 2 * *border* for some integer *n*.

border

The width of the border. Must be either zero or 1.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*target* was not an accepted value.
GL_INVALID_VALUE	*level* was less than zero or greater than log2 *max*, where *max* is the returned value of GL_MAX_TEXTURE_SIZE.

NAME	MEANING
GL_INVALID_VALUE	*border* was not zero or 1.
GL_INVALID_VALUE	*width* was less than zero, greater than 2 + GL_MAX_TEXTURE_SIZE, or *width* cannot be represented as 2n + 2 * *border* for some integer *n*.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* was not an accepted value.
GL_INVALID_VALUE	*level* was less than zero or greater than log2 *max*, where *max* is the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_VALUE	*border* was not zero or 1.

NAME	MEANING
GL_INVALID_VALUE	*width* was less than zero, greater than 2 + GL_MAX_TEXTURE_SIZE, or *width* cannot be represented as 2n + 2 * *border* for some integer *n*.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glCopyTexImage2D** function defines a two-dimensional texture image using pixels from the current framebuffer, rather than from main memory as is the case for **glTexImage2D**.

Using the mipmap level specified with *level*, texture arrays are defined as a rectangle of pixels with the lower-left corner located at the coordinates *x* and *y*, width equal to *width* + (2 * *border*), and a height equal to *height* + (2 * *border*). The internal format of the texture array is specified with the *internalFormat* parameter.

The **glCopyTexImage2D** function processes the pixels in a row in the same way as **glCopyPixels** except that before the final conversion of the pixels, all pixel component values are clamped to the range [0,1] and converted to the texture's internal format for storage in the texture array. Pixel ordering is determined with lower *x* and *y* coordinates corresponding to lower *s* and *t* texture coordinates. If any of the pixels within a specified row of the current framebuffer are outside the window associated with the current rendering context, then their values are undefined.

You cannot include calls to **glCopyTexImage2D** in display lists.

> **NOTE**
>
> The **glCopyTexImage2D** function is only available in OpenGL version 1.1 or later.

Texturing has no effect in color-index mode. The **glPixelStore** and **glPixelTransfer** functions affect texture images in exactly the way they affect **glDrawPixels**.

The following function retrieves information related to **glCopyTexImage2D**:

glIsEnabled with argument GL_TEXTURE_2D

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glCopyTexImage1D

glDra

wPixels

glEnd

glFog

glPixelStore

glPixelTransfer

glTe

xEnv

glTexGen

glTexImage1D

glTexImage2D

glTexParameter

r

glCopyTexSubImage1D function

2/22/2020 • 3 minutes to read • Edit Online

The **glCopyTexSubImage1D** function copies a sub-image of a one-dimensional texture image from the framebuffer.

Syntax

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

Parameters

target The target to which the image data will be changed. Must have the value GL_TEXTURE_1D.

level The level-of-detail number. Level 0 is the base image. Level *n* is the *n*th mipmap reduction image.

xoffset The texel offset within the texture array.

x The window x-plane coordinate of the lower-left corner of the row of pixels to be copied.

y The window y-plane coordinate of the lower-left corner of the row of pixels to be copied.

width

The width of the sub-image of the texture image. Specifying a texture sub-image with zero width has no effect.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*target* was not an accepted value.
GL_INVALID_VALUE	*level* was less than zero or *level* is greater than *log*2(*max*), where *max* is the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_VALUE	*xoffset* was less than *border* or (*xoffset* + *width*)was greater than (*w* + *border*), where *w* is GL_TEXTURE_WIDTH and *border* is GL_TEXTURE_BORDER. Note that *w* includes twice the *border* width.
GL_INVALID_VALUE	*width* was less than *border* or *y* was less than *border*, where *border* is the border width of the texture array.
GL_INVALID_OPERATION	The texture array was not defined by a previous glTexImage1D operation.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* was not an accepted value.
GL_INVALID_VALUE	*level* was less than zero or *level* is greater than *log2*(*max*), where *max* is the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_VALUE	*xoffset* was less than *border* or (*xoffset* + *width*)was greater than (*w* + *border*), where *w* is GL_TEXTURE_WIDTH and *border* is GL_TEXTURE_BORDER. Note that *w* includes twice the *border* width.
GL_INVALID_VALUE	*width* was less than *border* or *y* was less than *border*, where *border* is the border width of the texture array.

NAME	MEANING
GL_INVALID_OPERATION	The texture array was not defined by a previous glTexImage1D operation.
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Remarks

The **glCopyTexSubImage1D** function replaces a portion of a one-dimensional texture image using pixels from the current framebuffer, rather than from main memory as is the case for glTexSubImage1D.

A row of pixels beginning with the window coordinates specified by *x* and *y* and with the length *width* replaces the portion of the texture array with the indexes *xoffset* through *xoffset* + (*width* - 1). The destination in the texture array cannot include any texels outside the originally specified texture array.

The **glCopyTexSubImage1D** function processes the pixels in a row in the same way as glCopyPixels except that before the final conversion of the pixels, all pixel component values are clamped to the range [0,1] and converted to the texture's internal format for storage in the texture array. Pixel ordering is determined with lower *x* coordinates corresponding to lower texture coordinates. If any of the pixels within a specified row of the current framebuffer are outside the window associated with the current rendering context, then their values are undefined.

No change is made to the *internalFormat*, *width*, or *border* parameter of the specified texture array or to texel values outside the specified texture sub-image.

You cannot include calls to **glCopyTexSubImage1D** in display lists.

> **NOTE**
> The **glCopyTexSubImage1D** function is only available in OpenGL version 1.1 or later.

Texturing has no effect in color-index mode. The glPixelStore and glPixelTransfer functions affect texture images in exactly the way they affect the way pixels are drawn using glDrawPixels.

The following functions retrieve information related to **glCopyTexSubImage1D**:

glGetTexImage

glIsEnabled with argument GL_TEXTURE_1D

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glCopyTexSubImage2D

gl

D
r
a
w
P
i
x
e
l
s

g
l
E
n
d

g
l
F
o
g

g
l
P
i
x
e
l
S
t
o
r
e

g
l
P
i
x
e
l
T
r
a
n
s
f
e
r

g

lTexEnv

glTexGen

glTexImage1D

glTexImage2D

glTexSubIm

a
g
e
1
D

g
l
T
e
x
S
u
b
I
m
a
g
e
2
D

g
l
T
e
x
P
a
r
a
m
e
t
e
r

glCopyTexSubImage2D function

2/22/2020 • 4 minutes to read • Edit Online

The **glCopyTexSubImage2D** function copies a sub-image of a two-dimensional texture image from the framebuffer.

Syntax

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

Parameters

target The target to which the image data will be changed. Must have the value GL_TEXTURE_2D.

level The level-of-detail number. Level 0 is the base image. Level *n* is the *n*th mipmap reduction image.

xoffset The texel offset in the *x* direction within the texture array.

yoffset The texel offset in the *y* direction within the texture array.

x The window x-plane coordinates of the lower-left corner of the row of pixels to be copied.

y The window y-plane coordinates of the lower-left corner of the row of pixels to be copied.

width

The width of the sub-image of the texture image. Specifying a texture sub-image with zero width has no effect.

height

The height of the sub-image of the texture image. Specifying a texture sub-image with zero width has no effect.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*target* was not an accepted value.
GL_INVALID_VALUE	*level* was less than zero or greater than *log2* (*max*), where *max* is the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_VALUE	*xoffset* was less than *border* or (*xoffset* + *width*)was greater than (*w* + *border*), *yoffset* was less than *border*, or (*yoffset* + *height*) was greater than (*h* + *border*), where *w* is GL_TEXTURE_WIDTH and *border* is GL_TEXTURE_BORDER. Note that *w* includes twice the *border* width.

NAME	MEANING
GL_INVALID_VALUE	*width* was less than *border* or *y* was less than *border*, where *border* is the border width of the texture array.
GL_INVALID_OPERATION	The texture array was not defined by a previous glTexImage1D operation.
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*target* was not an accepted value.

NAME	MEANING
GL_INVALID_VALUE	*level* was less than zero or greater than *log2* (*max*), where *max* is the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_VALUE	*xoffset* was less than *border* or (*xoffset* + *width*)was greater than (*w* + *border*), *yoffset* was less than *border*, or (*yoffset* + *height*) was greater than (*h* + *border*), where *w* is GL_TEXTURE_WIDTH and *border* is GL_TEXTURE_BORDER. Note that *w* includes twice the *border* width.
GL_INVALID_VALUE	*width* was less than *border* or *y* was less than *border*, where *border* is the border width of the texture array.
GL_INVALID_OPERATION	The texture array was not defined by a previous **glTexImage1D** operation.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glCopyTexSubImage2D** function replaces a rectangular portion of a two-dimensional texture image with pixels from the current framebuffer, rather than from main memory as is the case for **glTexSubImage2D**.

A rectangle of pixels beginning with the *x* and *y* window coordinates and with the dimensions *width* and *height* replaces the portion of the texture array with the indexes *xoffset* through *xoffset* + (*width* - 1), with the indexes *yoffset* through *yoffset* + (*width* - 1) at the mipmap level specified by *level*. The destination rectangle in the texture array cannot include any texels outside the originally specified texture array.

The **glCopyTexSubImage2D** function processes the pixels in a row in the same way as **glCopyPixels**, except that before the final conversion of the pixels, all pixel component values are clamped to the range [0,1] and

converted to the texture's internal format for storage in the texture array. Pixel ordering is determined with lower *x* coordinates corresponding to lower texture coordinates. If any of the pixels within a specified row of the current framebuffer are outside the window associated with the current rendering context, then their values are undefined.

If any of the pixels within the specified rectangle of the current framebuffer are outside the read window associated with the current rendering context, then the values obtained for those pixels are undefined. No change is made to the *internalFormat*, *width*, *height*, or *border* parameter of the specified texture array or to texel values outside the specified texture sub-image.

You cannot include calls to **glCopyTexSubImage2D** in display lists.

> **NOTE**
>
> The **glCopyTexSubImage2D** function is only available in OpenGL version 1.1 or later.

Texturing has no effect in color-index mode. The **glPixelStore** and **glPixelTransfer** functions affect texture images in exactly the way they affect the way pixels are drawn using **glDrawPixels**.

The following functions retrieve information related to **glCopyTexSubImage2D**:

glGetTexImage

glIsEnabled with argument GL_TEXTURE_2D

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glCopyPixels

glCopyTexSubImage1D

glDrawPixels

glEnd

glFog

glPixelStore

glPixelTransfer

glTexEnv

glTexGen

glTexImage2D

glTexSubImage2D

glTexParameter

glCullFace function

2/22/2020 • 2 minutes to read • Edit Online

The **glCullFace** function specifies whether front-facing or back-facing facets can be culled.

Syntax

```
void WINAPI glCullFace(
   GLenum mode
);
```

Parameters

m
o
d
e

Specifies whether front-facing or back-facing facets are candidates for culling. The symbolic constants GL_FRONT, GL_BACK, and GL_FRONT_AND_BACK are accepted. The default value is GL_BACK.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*mode* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*mode* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

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

The **glFrontFace** function specifies which of the clockwise and counterclockwise facets are front-facing and back-facing.

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

The following functions retrieve information related to **glCullFace**:

glGet with argument GL_CULL_FACE_MODE

glIsEnabled with argument GL_CULL_FACE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib

DLL	Opengl32.dll

See also

glBegin

glDisable

glEnable

glEnd

glFrontFace

glGet

glIsEnabled

glDeleteLists function

2/22/2020 • 2 minutes to read • Edit Online

The **glDeleteLists** function deletes a contiguous group of display lists.

Syntax

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

Parameters

list — The integer name of the first display list to delete.

range — The number of display lists to delete.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	*range* was negative.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	*range* was negative.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glDeleteLists** function causes a contiguous group of display lists to be deleted. The *list* parameter is the name of the first display list to be deleted, and *range* is the number of display lists to delete. All display lists *d* with *list* = *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 zero, nothing happens.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glCallList

glCallLists

glEnd

glGenLists

glIsList

glNewList

glDeleteTextures function

2/22/2020 • 2 minutes to read • Edit Online

The **glDeleteTextures** function deletes named textures.

Syntax

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

Parameters

n The number of textures to be deleted.

textures An array of textures to be deleted.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INV ALI D_V AL UE	*n* was a negative value.

NAME	MEANING
GL_ INV ALI D_ OP ERA TIO N	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_ INV ALI D_V AL UE	*n* was a negative value.
GL_ INV ALI D_ OP ERA TIO N	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glDeleteTextures** function deletes *n* textures named by the elements of the array *textures*. After a texture is deleted, it has no contents or dimensionality, and its name is free for reuse (for example, by **glGenTextures**). The **glDeleteTextures** function ignores zeros and names that do not correspond to existing textures.

If a texture that is currently bound is deleted, the binding reverts to zero (the default texture).

You cannot include calls to **glDeleteTextures** in display lists.

> **NOTE**
>
> The **glDeleteTextures** function is only available in OpenGL version 1.1 or later.

The following function retrieves information related to **glDeleteTextures**:

- **glIsTexture**

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glAreTexturesResident

glBegin

gl

BindTexture

glEnd

glGenTextures

glGet

glGetTexParamet

e
r

g
l
I
s
T
e
x
t
u
r
e

g
l
P
r
i
o
r
i
t
i
z
e
T
e
x
t
u
r
e
s

g
l
T
e
x
G
e
n

g
l
T
e
x
I
m
a

g
e
1
D

g
l
T
e
x
I
m
a
g
e
2
D

g
l
T
e
x
P
a
r
a
m
e
t
e
r

glDepthFunc function

2/22/2020 • 2 minutes to read • Edit Online

The **glDepthFunc** function specifies the value used for depth-buffer comparisons.

Syntax

```
void WINAPI glDepthFunc(
   GLenum func
);
```

Parameters

func

Specifies the depth-comparison function. The following symbolic constants are accepted.

VALUE	MEANING
GL_NEVER	Never passes.
GL_LESS	Passes if the incoming *z* value is less than the stored *z* value. This is the default value.
GL_LEQUAL	Passes if the incoming z value is less than or equal to the stored z value.
GL_EQUAL	Passes if the incoming z value is equal to the stored z value.
GL_GREATER	Passes if the incoming z value is greater than the stored z value.

VALUE	MEANING
GL_ NO TEQ UA L	Passes if the incoming z value is not equal to the stored z value.
GL_ GE QU AL	Passes if the incoming z value is greater than or equal to the stored z value.
GL_ AL WA YS	Always passes.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the glGetError function.

NAME	MEANING
GL_ INV ALI D_ OP ERA TIO N	The function was called between a call to glBegin and the corresponding call to glEnd.

Error codes

The following error code can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

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

Initially, depth testing is disabled.

The following functions retrieve information related to **glDepthFunc**:

glGet with argument GL_DEPTH_FUNC

glIsEnabled with argument GL_DEPTH_TEST

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

glDepthRange

glEnable

glEnd

glGet

glIsEnabled

glDepthMask function

2/22/2020 • 2 minutes to read • Edit Online

The **glDepthMask** function enables or disables writing into the depth buffer.

Syntax

```
void WINAPI glDepthMask(
   GLboolean flag
);
```

Parameters

f
l
a
g

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

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The following function retrieves information related to **glDepthMask**:

glGet with argument GL_DEPTH_WRITEMASK

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

o
r
M
a
s
k

g
l
D
e
p
t
h
F
u
n
c

g
l
D
e
p
t
h
R
a
n
g
e

g
l
E
n
d

g
l
G
e
t

g
l
I
n
d
e
x
M
a

s
k

g
l
S
t
e
n
c
i
l
M
a
s
k

glDepthRange function

2/22/2020 • 2 minutes to read • Edit Online

The **glDepthRange** function specifies the mapping of *z* values from normalized device coordinates to window coordinates.

Syntax

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

Parameters

zNear The mapping of the near clipping plane to window coordinates. The default value is zero.

zFar The mapping of the far clipping plane to window coordinates. The default value is 1.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

After clipping and division by *w*, *z* -coordinates range from 0.0 to 1.0, corresponding to the near and far clipping planes. The **glDepthRange** function specifies a linear mapping of the normalized *z*-coordinates in this range to window *z*-coordinates. Regardless of the actual depth buffer implementation, window coordinate depth values are treated as though they range from 0.0 through 1.0 (like color components). Thus, the values accepted by **glDepthRange** are both clamped to this range before they are accepted.

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

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

The following function retrieves information related to **glDepthRange**:

glGet with argument GL_DEPTH_RANGE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glB

e
g
i
n

g
l
D
e
p
t
h
F
u
n
c

g
l
E
n
d

g
l
G
e
t

g
l
V
i
e
w
p
o
r
t

glDisable function

2/22/2020 • 5 minutes to read • Edit Online

The **glEnable** and **glDisable** functions enable or disable OpenGL capabilities.

Syntax

```
void WINAPI glDisable(
   GLenum cap
);
```

Parameters

cap	A symbolic constant indicating an OpenGL capability. For discussion of the values *cap* can take, see the following Remarks section.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*cap* was not one of the values listed in the preceding Remarks section.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*cap* was not one of the values listed in the preceding Remarks section.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glEnable** and **glDisable** functions enable and disable various OpenGL graphics capabilities. Use **glIsEnabled** or **glGet** to determine the current setting of any capability.

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

VALUE	MEANING
GL_ALPHA_TEST	If enabled, do alpha testing. See **glAlphaFunc**.
GL_AUTO_NORMAL	If enabled, compute surface normal vectors analytically when either GL_MAP2_VERTEX_3 or GL_MAP2_VERTEX_4 has generated vertices. See **glMap2**.
GL_BLEND	If enabled, blend the incoming RGBA color values with the values in the color buffers. See **glBlendFunc**.
GL_CLIP_PLANE*i*	If enabled, clip geometry against user-defined clipping plane *i*. See **glClipPlane**.
GL_COLOR_LOGIC_OP	If enabled, apply the current logical operation to the incoming RGBA color and color buffer values. See **glLogicOp**.
GL_COLOR_MATERIAL	If enabled, have one or more material parameters track the current color. See **glColorMaterial**.
GL_CULL_FACE	If enabled, cull polygons based on their winding in window coordinates. See **glCullFace**.
GL_DEPTH_TEST	If enabled, do depth comparisons and update the depth buffer. See **glDepthFunc** and **glDepthRange**.

VALUE	MEANING
GL_DITHER	If enabled, dither color components or indexes before they are written to the color buffer.
GL_FOG	If enabled, blend a fog color into the post-texturing color. See **glFog**.
GL_INDEX_LOGIC_OP	If enabled, apply the current 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, use the current lighting parameters to compute the vertex color or index. If disabled, associate the current color or index with each vertex. See **glMaterial**, **glLightModel**, and **glLight**.
GL_LINE_SMOOTH	If enabled, draw lines with correct filtering. If disabled, draw aliased lines. See **glLineWidth**.
GL_LINE_STIPPLE	If enabled, use the current line stipple pattern when drawing lines. See **glLineStipple**.
GL_LOGIC_OP	If enabled, apply the currently selected logical operation to the incoming and color-buffer indexes. See **glLogicOp**.
GL_MAP1_COLOR_4	If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** generate RGBA values. See also **glMap1**.
GL_MAP1_INDEX	If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** generate color indexes. See also **glMap1**.
GL_MAP1_NORMAL	If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** generate normals. See also **glMap1**.
GL_MAP1_TEXTURE_COORD_1	If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** generate *s* texture coordinates. See also **glMap1**.
GL_MAP1_TEXTURE_COORD_2	If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** generate *s* and *t* texture coordinates. See also **glMap1**.
GL_MAP1_TEXTURE_COORD_3	If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** generate *s*, *t*, and *r* texture coordinates. See also **glMap1**.
GL_MAP1_TEXTURE_COORD_4	If enabled, calls to glEvalCoord1, glEvalMesh1, and **glEvalPoint1** generate *s*, *t*, *r*, and *q* texture coordinates. See also **glMap1**.
GL_MAP1_VERTEX_3	If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** generate *x*, *y*, and *z* vertex coordinates. See also **glMap1**.

VALUE	MEANING
GL_MAP1_VERTEX_4	If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** generate homogeneous *x*, *y*, *z*, and *w* vertex coordinates. See also **glMap1**.
GL_MAP2_COLOR_4	If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** generate RGBA values. See also **glMap2**.
GL_MAP2_INDEX	If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** generate color indexes. See also **glMap2**.
GL_MAP2_NORMAL	If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** generate normals. See also **glMap2**.
GL_MAP2_TEXTURE_COORD_1	If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** generate *s* texture coordinates. See also **glMap2**.
GL_MAP2_TEXTURE_COORD_2	If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** generate *s* and *t* texture coordinates. See also **glMap2**.
GL_MAP2_TEXTURE_COORD_3	If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** generate *s*, *t*, and *r* texture coordinates. See also **glMap2**.
GL_MAP2_TEXTURE_COORD_4	If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** generate *s*, *t*, *r*, and *q* texture coordinates. See also **glMap2**.
GL_MAP2_VERTEX_3	If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** generate *x*, *y*, and *z* vertex coordinates. See also **glMap2**.
GL_MAP2_VERTEX_4	If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** generate homogeneous *x*, *y*, *z*, and *w* vertex coordinates. See also **glMap2**.
GL_NORMALIZE	If enabled, normal vectors specified with **glNormal** are scaled to unit length after transformation. See **glNormal**.
GL_POINT_SMOOTH	If enabled, draw points with proper filtering. If disabled, draw aliased points. See **glPointSize**.
GL_POLYGON_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**.

VALUE	MEANING
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. If disabled, draw aliased polygons. See **glPolygonMode**.
GL_POLYGON_STIPPLE	If enabled, use the current polygon stipple pattern when rendering polygons. See **glPolygonStipple**.
GL_SCISSOR_TEST	If enabled, discard fragments that are outside the scissor rectangle. See **glScissor**.
GL_STENCIL_TEST	If enabled, do stencil testing and update the stencil buffer. See **glStencilFunc** and **glStencilOp**.
GL_TEXTURE_1D	If enabled, one-dimensional texturing is performed (unless two-dimensional texturing is also enabled). See **glTexImage1D**.
GL_TEXTURE_2D	If enabled, two-dimensional texturing is performed. See **glTexImage2D**.
GL_TEXTURE_GEN_Q	If enabled, the *q* texture coordinate is computed using the texture-generation function defined with **glTexGen**. Otherwise, the current *q* texture coordinate is used.
GL_TEXTURE_GEN_R	If enabled, the *r* texture coordinate is computed using the texture generation function defined with **glTexGen**. If disabled, the current *r* texture coordinate is used.
GL_TEXTURE_GEN_S	If enabled, the *s* texture coordinate is computed using the texture generation function defined with **glTexGen**. If disabled, the current *s* texture coordinate is used.
GL_TEXTURE_GEN_T	If enabled, the *t* texture coordinate is computed using the texture generation function defined with **glTexGen**. If disabled, the current *t* texture coordinate is used.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glAlphaFunc

glArrayElement

glBegin

g

lBlendFunc

glClipPlane

glColorMaterial

glColorPointer

glCullFace

glDepthFunc

glDepthRange

glDrawArrays

glE

d
g
e
F
l
a
g
P
o
i
n
t
e
r

g
l
E
n
a
b
l
e

g
l
E
n
d

g
l
E
v
a
l
C
o
o
r
d
1

g
l
E
v
a
l
M
e
s
h

1

glEvalPoint1

glFog

glGet

glIndexPointer

glIsEnabled

glLight

glLightModel

glLineWidth

glLineStipple

glLogi

c
O
p

g
l
M
a
p
1

g
l
M
a
p
2

g
l
M
a
t
e
r
i
a
l

g
l
N
o
r
m
a
l

g
l
N
o
r
m
a
l
P
o
i
n
t
e
r

glPointSize

glPolygonMode

glPolygonStipple

glScissor

glStencilFunc

glStencilOp

glTexCoordPointer

glTexGen

glTexImage1D

glTexImage2D

glDisableClientState function

2/22/2020 • 2 minutes to read • Edit Online

The **glEnableClientState** and **glDisableClientState** functions enable and disable arrays respectively.

Syntax

```
void WINAPI glDisableClientState(
   GLenum array
);
```

Parameters

array

A symbolic constant for the array you want to enable or disable. This parameter can assume one of the following values.

VALUE	MEANING
GL_COLOR_ARRAY	If enabled, use color arrays with calls to **glArrayElement**, **glDrawElements**, or **glDrawArrays**. See also **glColorPointer**.
GL_EDGE_FLAG_ARRAY	If enabled, use edge flag arrays with calls to **glArrayElement**, **glDrawElements**, or **glDrawArrays**. See also **glEdgeFlagPointer**.
GL_INDEX_ARRAY	If enabled, use index arrays with calls to **glArrayElement**, **glDrawElements**, or **glDrawArrays**. See also **glIndexPointer**.

VALUE	MEANING
GL_NORMAL_ARRAY	If enabled, use normal arrays with calls to glArrayElement, glDrawElements, or glDrawArrays. See also glNormalPointer.
GL_TEXTURE_COORD_ARRAY	If enabled, use texture coordinate arrays with calls to glArrayElement, glDrawElements, or glDrawArrays. See also glTexCoordPointer.
GL_VERTEX_ARRAY	If enabled, use vertex arrays with calls to glArrayElement, glDrawElements, or glDrawArrays. See also glVertexPointer.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*array* was not an accepted value.

Error codes

The following error code can be retrieved by the glGetError function.

NAME	MEANING
GL_ INV ALI D_E NU M	*array* was not an accepted value.

Remarks

The **glEnableClientState** and **glDisableClientState** functions enable and disable various individual arrays. Use **glIsEnabled** or **glGet** to determine the current setting of any capability.

Calling **glEnableClientState** and **glDisableClientState** between calls to **glBegin** and the corresponding call to **glEnd** can cause an error. If no error is generated, the behavior is undefined.

> **NOTE**
>
> The **glEnableClientState** and **glDisableClientState** functions are only available in OpenGL version 1.1 or later.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib
DLL	Ope ngl3 2.dll

See also

g
l
A
r
r
a
y

E
l
e
m
e
n
t

g
l
B
e
g
i
n

g
l
C
o
l
o
r
P
o
i
n
t
e
r

g
l
D
r
a
w
A
r
r
a
y
s

g
l
D
r
a
w
E
l
e

m
e
n
t
s

g
l
E
d
g
e
F
l
a
g
P
o
i
n
t
e
r

g
l
E
n
a
b
l
e

g
l
E
n
a
b
l
e
C
l
i
e
n
t
S
t
a
t
e

glEnd

glGetPointerv

glIndexPointer

glInterleavedArrays

s

glNormalPointer

glTexCoordPointer

glVertexPointer

glDrawArrays function

2/22/2020 • 2 minutes to read • Edit Online

The **glDrawArrays** function specifies multiple primitives to render.

Syntax

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

Parameters

mode

The kind of primitives to render. The following constants specify acceptable types of primitives: GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES, GL_QUAD_STRIP, GL_QUADS, and GL_POLYGON.

first

The starting index in the enabled arrays.

count

The number of indexes to render.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_ INV ALI D_V AL UE	*count* was negative.
GL_ INV ALI D_E NU M	*mode* was not an accepted value.
GL_ INV ALI D_ OP ERA TIO N	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_ INV ALI D_V AL UE	*count* was negative.
GL_ INV ALI D_E NU M	*mode* was not an accepted value.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

With **glDrawArrays**, you can specify multiple geometric primitives to render. Instead of calling separate OpenGL functions to pass each individual vertex, normal, or color, you can specify separate arrays of vertices, normals, and colors to define a sequence of primitives (all the same kind) with a single call to **glDrawArrays**.

When you call **glDrawArrays**, *count* sequential elements from each enabled array are used to construct a sequence of geometric primitives, beginning with the *first* element. The *mode* parameter specifies what kind of primitive to construct and how to use the array elements to construct the primitives.

After **glDrawArrays** returns, the values of vertex attributes that are modified by **glDrawArrays** are undefined. For example, if GL_COLOR_ARRAY is enabled, the value of the current color is undefined after **glDrawArrays** returns. Attributes not modified by **glDrawArrays** remain defined. When GL_VERTEX_ARRAY is not enabled, no geometric primitives are generated but the attributes corresponding to enabled arrays are modified.

You can include **glDrawArrays** in display lists. When you include **glDrawArrays** in a display list, the necessary array data, determined by the array pointers and the enables, are generated and entered in the display list. The values of array pointers and enables are determined during the creation of display lists.

You can read static array data at any time. If any static array elements are modified and the array is not specified again, the results of any subsequent calls to **glDrawArrays** are undefined.

Although no error is generated when you specify an array more than once within **glBegin** and **glend** pairs, the results are undefined.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib

DLL	Opengl32.dll

See also

glArrayElement

glBegin

glColorPointer

glEdge

F
l
a
g
P
o
i
n
t
e
r

g
l
E
n
d

g
l
G
e
t
P
o
i
n
t
e
r
v

g
l
G
e
t
S
t
r
i
n
g

g
l
I
n
d
e
x
P
o

i
n
t
e
r

g
l
N
o
r
m
a
l
P
o
i
n
t
e
r

g
l
T
e
x
C
o
o
r
d
P
o
i
n
t
e
r

g
l
V
e
r
t
e
x
P
o
i
n
t

e

r

glDrawBuffer function

2/22/2020 • 2 minutes to read • Edit Online

The **glDrawBuffer** function specifies which color buffers are to be drawn into.

Syntax

```
void WINAPI glDrawBuffer(
   GLenum mode
);
```

Parameters

mode

Specifies up to four color buffers to be drawn into with the following acceptable symbolic constants.

VALUE	MEANING
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.

VALUE	MEANING
GL_BACK_RIGHT	Only the back-right color buffer is written.
GL_FRONT	Only the front-left and front-right color buffers are written. If there is no front-right color buffer, only the front left-color buffer is written.
GL_BACK	Only the back-left and back-right color buffers are written. If there is no back-right color buffer, only the back-left color buffer is written.
GL_LEFT	Only the front-left and back-left color buffers are written. If there is no back-left color buffer, only the front-left color buffer is written.
GL_RIGHT	Only the front-right and back-right color buffers are written. If there is no back-right color buffer, only the front-right color buffer is written.
GL_FRONT_AND_BACK	All the front and back color buffers (front-left, front-right, back-left, back-right) are written. If there are no back color buffers, only the front-left and front-right color buffers are written. If there are no right color buffers, only the front-left and back-left color buffers are written. If there are no right or back color buffers, only the front-left color buffer is written.
GL_AUXi	Only the auxiliary color buffer *i* is written; *i* is between 0 and GL_AUX_BUFFERS - 1. (GL_AUX_BUFFERS is not the upper limit; use **glGet** to query the number of available auxiliary buffers.)

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

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*mode* was not an accepted value.
GL_INVALID_OPERATION	None of the buffers indicated by *mode* existed.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*mode* was not an accepted value.
GL_INVALID_OPERATION	None of the buffers indicated by *mode* existed.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

When colors are written to the framebuffer, they are written into the color buffers specified by **glDrawBuffer**.

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

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

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

The following functions retrieve information related to the **glDrawBuffer** function:

glGet with argument GL_DRAW_BUFFER

glGet with argument GL_AUX_BUFFERS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

BeginglBlendFunc

glColorMask

glEnd

glGet

glIndexMask

glLogicOp

glReadBuffer

glDrawElements function

2/22/2020 • 2 minutes to read • Edit Online

The **glDrawElements** function renders primitives from array data.

Syntax

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

Parameters

mode

The kind of primitives to render. It can assume one of the following symbolic values: GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES, GL_QUAD_STRIP, GL_QUADS, and GL_POLYGON.

count

The number of elements to be rendered.

type

The type of the values in indices. Must be one of GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, or GL_UNSIGNED_INT.

indices

A pointer to the location where the indices are stored.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*mode* was not an accepted value.
GL_INVALID_VALUE	*count* was a negative value.
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*mode* was not an accepted value.
GL_INVALID_VALUE	*count* was a negative value.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glDrawElements** function enables you to specify multiple geometric primitives with very few function calls. Instead of calling an OpenGL function to pass each individual vertex, normal, or color, you can specify separate arrays of vertices, normals, and colors beforehand and use them to define a sequence of primitives (all of the same type) with a single call to **glDrawElements**.

When you call the **glDrawElements** function, it uses *count* sequential elements from *indices* to construct a sequence of geometric primitives. The *mode* parameter specifies what kind of primitives are constructed, and how the array elements are used to construct these primitives. If GL_VERTEX_ARRAY is not enabled, no geometric primitives are generated.

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

You can include the **glDrawElements** function in display lists. When **glDrawElements** is included in a display list, the necessary array data (determined by the array pointers and enables) is also entered into the display list. Because the array pointers and enables are client-side state variables, their values affect display lists when the lists are created, not when the lists are executed.

> **NOTE**
> The **glDrawElements** function is only available in OpenGL version 1.1 or later.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib

DLL	Opengl32.dll

See also

glArrayElement

glBegin

glColorPointer

glDraw

Arrays

glEdgeFlagPointer

glEnd

glGetPointerv

glIndexP

o
i
n
t
e
r

g
l
N
o
r
m
a
l
P
o
i
n
t
e
r

g
l
T
e
x
C
o
o
r
d
P
o
i
n
t
e
r

g
l
V
e
r
t
e
x
P
o
i
n

t
e
r

glDrawPixels function

2/22/2020 • 12 minutes to read • Edit Online

The **glDrawPixels** function writes a block of pixels to the framebuffer.

Syntax

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

Parameters

width The width dimension of the pixel rectangle that will be written into the framebuffer.

height The height dimension of the pixel rectangle that will be written into the framebuffer.

format The format of the pixel data. Acceptable symbolic constants are as follows.

VALUE	MEANING
GL_COLOR_INDEX	Each pixel is a single value, a color index. 1. The **glDrawPixels** function converts each pixel to fixed-point format, with an unspecified number of bits to the right of the binary point, regardless of the memory data type. Floating-point values convert to true fixed-point values. The **glDrawPixels** function converts signed and unsigned integer data with all fraction bits set to zero. The function converts bitmap data to either 0.0 or 1.0. 2. The **glDrawPixels** function shifts each fixed-point index left by GL_INDEX_SHIFT bits and

VALUE	MEANING
	adds it to GL_INDEX_OFFSET. If GL_INDEX_SHIFT is negative, the shift is to the right. In either case, zero bits fill otherwise unspecified bit locations in the result. 3. When in RGBA mode, **glDrawPixels** converts the resulting index to an RGBA pixel using the GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, and GL_PIXEL_MAP_I_TO_A tables. When in the color-index mode and GL_MAP_COLOR is true, the index is replaced with the value that **glDrawPixels** references in lookup table GL_PIXEL_MAP_I_TO_I. 4. Whether the lookup replacement of the index is done or not, the integer part of the index is **AND**ed with $2^b - 1$, where *b* is the number of bits in a color-index buffer. 5. The resulting indexes or RGBA colors are then converted to fragments by attaching the current raster position *z*-coordinate and texture coordinates to each pixel, and then assigning *x* and *y* window coordinates to the *n*th fragment such that $x? = x_r + n \bmod width$ $y? = y_r + n/width$ where (x_r , y_r) is the current raster position. 6. The **glDrawPixels** function treats these pixel fragments just like the fragments generated by rasterizing points, lines, or polygons. It applies texture mapping, fog, and all the fragment operations before writing the fragments to the framebuffer.

VALUE	MEANING
GL_STENCIL_INDEX	Each pixel is a single value, a stencil index. 1. The **glDrawPixels** function converts it to fixed-point format, with an unspecified number of bits to the right of the binary point, regardless of the memory data type. Floating-point values convert to true fixed-point values. The **glDrawPixels** function converts signed and unsigned integer data with all fraction bits set to zero. Bitmap data converts to either 0.0 or 1.0. 2. The **glDrawPixels** function shifts each fixed-point index left by GL_INDEX_SHIFT bits, and adds it to GL_INDEX_OFFSET. If GL_INDEX_SHIFT is negative, the shift is to the right. In either case, zero bits fill otherwise unspecified bit locations in the result. 3. If GL_MAP_STENCIL is true, the index is replaced with the value that **glDrawPixels** references in lookup table GL_PIXEL_MAP_S_TO_S. 4. Whether the lookup replacement of the index is done or not, the integer part of the index is then **AND**ed with $2^b - 1$, where *b* is the number of bits in the stencil buffer. The resulting stencil indexes are then written to the stencil buffer such that the *n*th index is written to location $x? = x_r + n \bmod width$ $y? = y_r + n/width$ where (x_r, y_r) is the current raster position. Only the pixel ownership test, the scissor test, and the stencil writemask affect these writes.

VALUE	MEANING
GL_DEPTH_COMPONENT	Each pixel is a single-depth component. 1. The **glDrawPixels** function converts floating-point data directly to an internal floating-point format with unspecified precision. Signed integer data is mapped linearly to the internal floating-point format such that the most positive representable integer value maps to 1.0, and the most negative representable value maps to -1.0. Unsigned integer data is mapped similarly: the largest integer value maps to 1.0, and zero maps to 0.0. 2. The **glDrawPixels** function multiplies the resulting floating-point depth value by GL_DEPTH_SCALE and adds it to GL_DEPTH_BIAS. The result is clamped to the range [0,1]. 3. The **glDrawPixels** function converts the resulting depth components to fragments by attaching the current raster position color or color index and texture coordinates to each pixel, and then assigning *x* and *y* window coordinates to the *n* th fragment such that $x? = x_r + n \bmod width$ $y? = y_r + n/width$ where (x_r, y_r) is the current raster position. 4. These pixel fragments are then treated just like the fragments generated by rasterizing points, lines, or polygons. The **glDrawPixels** function applies texture mapping, fog, and all the fragment operations before writing the fragments to the framebuffer.

VALUE	MEANING
GL_ RG BA	Each pixel is a four-component group in this order: red, green, blue, alpha. 1. The **glDrawPixels** function converts floating-point values directly to an internal floating-point format with unspecified precision. Signed integer values are mapped linearly to the internal floating-point format such that the most positive representable integer value maps to 1.0, and the most negative representable value maps to -1.0. Unsigned integer data is mapped similarly: the largest integer value maps to 1.0, and zero maps to 0.0. 2. The **glDrawPixels** function multiplies the resulting floating-point color values by GL_c_SCALE and adds them to GL_c_BIAS, where *c* is RED, GREEN, BLUE, and ALPHA for the respective color components. The results are clamped to the range [0,1]. 3. If GL_MAP_COLOR is true, **glDrawPixels** scales each color component by the size of lookup table GL_PIXEL_MAP_c_TO_c, and then replaces the component by the value that it references in that table; *c* is R, G, B, or A, respectively. 4. The **glDrawPixels** function converts the resulting RGBA colors to fragments by attaching the current raster position *z*-coordinate and texture coordinates to each pixel, then assigning *x* and *y* window coordinates to the *n*th fragment such that $x? = x_r + n \bmod width$ $y? = y_r + n / width$ where (x_r, y_r) is the current raster position. 5. These pixel fragments are then treated just like the fragments generated by rasterizing points, lines, or polygons. The **glDrawPixels** function applies texture mapping, fog, and all the fragment operations before writing the fragments to the framebuffer.

VALUE	MEANING
GL_RED	Each pixel is a single red component. The **glDrawPixels** function converts this component to the internal floating-point format in the same way that the red component of an RGBA pixel is, and then converts it to an RGBA pixel with green and blue set to 0.0, and alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.
GL_GREEN	Each pixel is a single green component. The **glDrawPixels** function converts this component to the internal floating-point format in the same way that the green component of an RGBA pixel is, and then converts it to an RGBA pixel with red and blue set to 0.0, and alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.
GL_BLUE	Each pixel is a single blue component. The **glDrawPixels** function converts this component to the internal floating-point format in the same way that the blue component of an RGBA pixel is, and then converts it to an RGBA pixel with red and green set to 0.0, and alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.
GL_ALPHA	Each pixel is a single alpha component. The **glDrawPixels** function converts this component to the internal floating-point format in the same way that the alpha component of an RGBA pixel is, and then converts it to an RGBA pixel with red, green, and blue set to 0.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.
GL_RGB	Each pixel is a group of three components in this order: red, green, blue. The **glDrawPixels** function converts each component to the internal floating-point format in the same way that the red, green, and blue components of an RGBA pixel are. The color triple is converted to an RGBA pixel with alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.

VALUE	MEANING
GL_LUMINANCE	Each pixel is a single luminance component. The **glDrawPixels** function converts this component to the internal floating-point format in the same way that the red component of an RGBA pixel is, and then converts it to an RGBA pixel with red, green, and blue set to the converted luminance value, and alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.
GL_LUMINANCE_ALPHA	Each pixel is a group of two components in this order: luminance, alpha. The **glDrawPixels** function converts the two components to the internal floating-point format in the same way that the red component of an RGBA pixel is, and then converts them to an RGBA pixel with red, green, and blue set to the converted luminance value, and alpha set to the converted alpha value. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.
GL_BGR_EXT	Each pixel is a group of three components in this order: blue, green, red. GL_BGR_EXT provides a format that matches the memory layout of Windows device-independent bitmaps (DIBs). Thus, your applications can use the same data with Windows function calls and OpenGL pixel function calls.
GL_BGRA_EXT	Each pixel is a group of four components in this order: blue, green, red, alpha. GL_BGRA_EXT provides a format that matches the memory layout of Windows device-independent bitmaps (DIBs). Thus, your applications can use the same data with Windows function calls and OpenGL pixel function calls.

type

The data type for *pixels*. The following are the accepted symbolic constants and their meanings.

VALUE	MEANING
GL_UNSIGNED_BYTE	Unsigned 8-bit integer
GL_BYTE	Signed 8-bit integer
GL_BITMAP	Single bits in unsigned 8-bit integers
GL_UNSIGNED_SHORT	Unsigned 16-bit integer
GL_SHORT	Signed 16-bit integer
GL_UNSIGNED_INT	Unsigned 32-bit integer
GL_INT	32-bit integer
GL_FLOAT	Single-precision floating-point

pix A pointer to the pixel data.

e
l
s

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	Either *width* or *height* was negative.
GL_INVALID_ENUM	Either *format* or *type* was not an accepted value.
GL_INVALID_OPERATION	*format* was GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_BGR_EXT, GL_BGRA_EXT, GL_LUMINANCE, or GL_LUMINANCE_ALPHA, and OpenGL was in color-index mode.
GL_INVALID_ENUM	*type* was GL_BITMAP and *format* was not either GL_COLOR_INDEX or GL_STENCIL_INDEX.
GL_INVALID_OPERATION	*format* was GL_STENCIL_INDEX and there was no stencil buffer.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	Either *width* or *height* was negative.
GL_INVALID_ENUM	Either *format* or *type* was not an accepted value.
GL_INVALID_OPERATION	*format* was GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_BGR_EXT, GL_BGRA_EXT, GL_LUMINANCE, or GL_LUMINANCE_ALPHA, and OpenGL was in color-index mode.
GL_INVALID_ENUM	*type* was GL_BITMAP and *format* was not either GL_COLOR_INDEX or GL_STENCIL_INDEX.

NAME	MEANING
GL_INVALID_OPERATION	*format* was GL_STENCIL_INDEX and there was no stencil buffer.
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Remarks

The **glDrawPixels** function reads pixel data from memory and writes it into the framebuffer relative to the current raster position. Use glRasterPos to set the current raster position, and use glGet with argument GL_CURRENT_RASTER_POSITION to query the raster position.

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

Data is read from *pixels* as a sequence of signed or unsigned bytes, signed or unsigned shorts, signed or unsigned integers, or single-precision floating-point values, depending on *type*. Each of these bytes, shorts, integers, or floating-point values is interpreted as one color or depth component, or one index, depending on *format*. Indexes are always treated individually. Color components are treated as groups of one, two, three, or four values, again based on *format*. Both individual indexes and groups of components are referred to as pixels. If *type* is GL_BITMAP, the data must be unsigned bytes, and *format* must be either GL_COLOR_INDEX or GL_STENCIL_INDEX. Each unsigned byte is treated as eight 1-bit pixels, with bit ordering determined by GL_UNPACK_LSB_FIRST (see glPixelStore).

The *width* by *height* pixels are read from memory, starting at location *pixels*. By default, these pixels are taken from adjacent memory locations, except that after all *width* pixels are read, the read pointer is advanced to the next 4-byte boundary. The **glPixelStore** function specifies the 4-byte row alignment with argument GL_UNPACK_ALIGNMENT, and you can set it to 1, 2, 4, or 8 bytes. Other pixel store parameters specify different read pointer advancements, both before the first pixel is read, and after all *width* pixels are read. The **glPixelStore** function operates on each of the *width-by-height* pixels that it reads from memory in the same way, based on the values of several parameters specified by glPixelTransfer and glPixelMap. The details of these operations, as well as the target buffer into which the pixels are drawn, are specific to the format of the pixels, as specified by *format*.

The rasterization described thus far assumes pixel zoom factors of 1.0. If you use glPixelZoom to change the *x* and *y* pixel zoom factors, pixels are converted to fragments as follows. If (*xr*,*yr*) is the

current raster position, and a given pixel is in the *n*th column and *m*th row of the pixel rectangle, then fragments are generated for pixels whose centers are in the rectangle with corners at

$(x_r + zoom? \ n, y_r + zoom_y \ m)$

$(x_r + zoom? \ (n + 1), y_r + zoom_y \ (m + 1))$

where *zoom*? is the value of GL_ZOOM_X and $zoom_y$ is the value of GL_ZOOM_Y.

The following functions retrieve information related to **glDrawPixels**:

glGet with argument GL_CURRENT_RASTER_POSITION

glGet with argument GL_CURRENT_RASTER_POSITION_VALID

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glAlphaFunc

glBeg

i
n

g
l
B
l
e
n
d
F
u
n
c

g
l
C
o
p
y
P
i
x
e
l
s

g
l
D
e
p
t
h
F
u
n
c

g
l
E
n
d

g
l
G
e
t

g
l

LogicOp

glPixelMap

glPixelStore

glPixelTransfer

glPix

e
l
Z
o
o
m

g
l
R
a
s
t
e
r
P
o
s

g
l
R
e
a
d
P
i
x
e
l
s

g
l
S
c
i
s
s
o
r

g
l
S
t
e
n
c
i
l
F
u

n

c

glEdgeFlag Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions flag edges as either boundary or nonboundary:

- **glEdgeFlag**
- **glEdgeFlagv**

glEdgeFlag function

2/22/2020 • 2 minutes to read • Edit Online

Flags edges as either boundary or nonboundary.

Syntax

```
void WINAPI glEdgeFlag(
   GLboolean flag
);
```

Parameters

flag	Specifies the current edge flag value, either **TRUE** or **FALSE**.

Return value

This function does not return a value.

Remarks

Each vertex of a polygon, separate triangle, or separate quadrilateral specified between a **glBegin**/**glEnd** pair is marked as the start of either a boundary or nonboundary edge. If the current edge flag is **TRUE** when the vertex is specified, the vertex is marked as the start of a boundary edge. If the current edge flag is **FALSE**, the vertex is marked as the start of a nonboundary edge. The **glEdgeFlag** function sets the edge flag to **TRUE** if flag is nonzero, **FALSE** otherwise.

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

Boundary and nonboundary edge flags on vertices are significant only if GL_POLYGON_MODE is set to GL_POINT or GL_LINE. See **glPolygonMode**.

Initially, the edge flag bit is **TRUE**.

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

The following functions retrieve information related to **glEdgeFlag**:

glGet with argument GL_EDGE_FLAG

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib
DLL	Ope ngl3 2.dll

glEdgeFlagv function

2/22/2020 • 2 minutes to read • Edit Online

Flags edges as either boundary or nonboundary.

Syntax

```
void WINAPI glEdgeFlagv(
   const GLboolean *flag
);
```

Parameters

f
l
a
g

Specifies a pointer to an array that contains a single Boolean element, which replaces the current edge flag value.

Return value

This function does not return a value.

Remarks

Each vertex of a polygon, separate triangle, or separate quadrilateral specified between a **glBegin**/**glEnd** pair is marked as the start of either a boundary or nonboundary edge. If the current edge flag is **TRUE** when the vertex is specified, the vertex is marked as the start of a boundary edge. If the current edge flag is **FALSE**, the vertex is marked as the start of a nonboundary edge. The **glEdgeFlagv** function sets the edge flag to **TRUE** if flag is nonzero, **FALSE** otherwise.

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

Boundary and nonboundary edge flags on vertices are significant only if GL_POLYGON_MODE is set to GL_POINT or GL_LINE. See **glPolygonMode**.

Initially, the edge flag bit is **TRUE**.

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

The following functions retrieve information related to **glEdgeFlagv**:

glGet with argument GL_EDGE_FLAG

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib
DLL	Ope ngl3 2.dll

glEdgeFlagPointer function

2/22/2020 • 2 minutes to read • Edit Online

The **glEdgeFlagPointer** function defines an array of edge flags.

Syntax

```
void WINAPI glEdgeFlagPointer(
         GLsizei stride,
   const GLvoid  *pointer
);
```

Parameters

stride

The byte offset between consecutive edge flags. When *stride* is zero, the edge flags are tightly packed in the array.

pointer

A pointer to the first edge flag in the array.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*stride* or *count* was negative.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_ INV ALI D_E NU M	*stride* or *count* was negative.

Remarks

The **glEdgeFlagPointer** function specifies the location and data of an array of Boolean edge flags to use when rendering. The *stride* parameter determines the byte offset from one edge flag to the next, which enables the packing of vertices and attributes in a single array or storage in separate arrays. In some implementations, storing the vertices and attributes in a single array can be more efficient than using separate arrays.

An edge-flag array is enabled when you specify the GL_EDGE_FLAG_ARRAY constant with **glEnableClientState**. When enabled, **glDrawArrays** or **glArrayElement** uses the edge-flag array. By default the edge-flag array is disabled.

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

You cannot include **glEdgeFlagPointer** in display lists.

When you specify an edge-flag array using **glEdgeFlagPointer**, the values of all the function's edge-flag array parameters are saved in a client-side state and static array elements can be cached. Because the edge-flag array parameters are in a client-side state, **glPushAttrib** and **glPopAttrib** do not save or restore their values.

Although calling **glEdgeFlagPointer** within a **glBegin**/**glend** pair does not generate an error, the results are undefined.

The following functions retrieve information related to the **glEdgeFlagPointer** function:

glGet with argument GL_EDGE_FLAG_ARRAY_STRIDE

glGet with argument GL_EDGE_FLAG_ARRAY_COUNT

glGetPointerv with argument GL_EDGE_FLAG_ARRAY_POINTER

glIsEnabled with argument GL_EDGE_FLAG_ARRAY

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glArrayElement

glBegin

glColorPointer

r

glDrawArrays

glEnableClientState

glEnd

glGet

glGetP

o
i
n
t
e
r
v

g
l
G
e
t
S
t
r
i
n
g

g
l
I
n
d
e
x
P
o
i
n
t
e
r

g
l
I
s
E
n
a
b
l
e
d

g
l
N
o
r
m

a
l
P
o
i
n
t
e
r

g
l
P
o
p
A
t
t
r
i
b

g
l
P
u
s
h
A
t
t
r
i
b

g
l
T
e
x
C
o
o
r
d
P
o
i
n
t
e
r

glVertexPointer

glEnable function

2/22/2020 • 5 minutes to read • Edit Online

The **glEnable** and **glDisable** functions enable or disable OpenGL capabilities.

Syntax

```
void WINAPI glEnable(
   GLenum cap
);
```

Parameters

cap	A symbolic constant indicating an OpenGL capability. For discussion of the values *cap* can take, see the following Remarks section.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*cap* was not one of the values listed in the preceding Remarks section.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*cap* was not one of the values listed in the preceding Remarks section.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glEnable** and **glDisable** functions enable and disable various OpenGL graphics capabilities. Use **glIsEnabled** or **glGet** to determine the current setting of any capability.

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

VALUE	MEANING
GL_ALPHA_TEST	If enabled, do alpha testing. See **glAlphaFunc**.
GL_AUTO_NORMAL	If enabled, compute surface normal vectors analytically when either GL_MAP2_VERTEX_3 or GL_MAP2_VERTEX_4 has generated vertices. See **glMap2**.
GL_BLEND	If enabled, blend the incoming RGBA color values with the values in the color buffers. See **glBlendFunc**.
GL_CLIP_PLANE*i*	If enabled, clip geometry against user-defined clipping plane *i*. See **glClipPlane**.
GL_COLOR_LOGIC_OP	If enabled, apply the current logical operation to the incoming RGBA color and color buffer values. See **glLogicOp**.
GL_COLOR_MATERIAL	If enabled, have one or more material parameters track the current color. See **glColorMaterial**.
GL_CULL_FACE	If enabled, cull polygons based on their winding in window coordinates. See **glCullFace**.

VALUE	MEANING
GL_DEPTH_TEST	If enabled, do depth comparisons and update the depth buffer. See glDepthFunc and glDepthRange.
GL_DITHER	If enabled, dither color components or indexes before they are written to the color buffer.
GL_FOG	If enabled, blend a fog color into the post-texturing color. See glFog.
GL_INDEX_LOGIC_OP	If enabled, apply the current 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, use the current lighting parameters to compute the vertex color or index. If disabled, associate the current color or index with each vertex. See glMaterial, glLightModel, and glLight.
GL_LINE_SMOOTH	If enabled, draw lines with correct filtering. If disabled, draw aliased lines. See glLineWidth.
GL_LINE_STIPPLE	If enabled, use the current line stipple pattern when drawing lines. See glLineStipple.
GL_LOGIC_OP	If enabled, apply the currently selected logical operation to the incoming and color-buffer indexes. See glLogicOp.
GL_MAP1_COLOR_4	If enabled, calls to glEvalCoord1, glEvalMesh1, and glEvalPoint1 generate RGBA values. See also glMap1.
GL_MAP1_INDEX	If enabled, calls to glEvalCoord1, glEvalMesh1, and glEvalPoint1 generate color indexes. See also glMap1.
GL_MAP1_NORMAL	If enabled, calls to glEvalCoord1, glEvalMesh1, and glEvalPoint1 generate normals. See also glMap1.
GL_MAP1_TEXTURE_COORD_1	If enabled, calls to glEvalCoord1, glEvalMesh1, and glEvalPoint1 generate *s* texture coordinates. See also glMap1.
GL_MAP1_TEXTURE_COORD_2	If enabled, calls to glEvalCoord1, glEvalMesh1, and glEvalPoint1 generate *s* and *t* texture coordinates. See also glMap1.

VALUE	MEANING
GL_MAP1_TEXTURE_COORD_3	If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** generate *s*, *t*, and *r* texture coordinates. See also **glMap1**.
GL_MAP1_TEXTURE_COORD_4	If enabled, calls to glEvalCoord1, glEvalMesh1, and **glEvalPoint1** generate *s*, *t*, *r*, and *q* texture coordinates. See also **glMap1**.
GL_MAP1_VERTEX_3	If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** generate *x*, *y*, and *z* vertex coordinates. See also **glMap1**.
GL_MAP1_VERTEX_4	If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** generate homogeneous *x*, *y*, *z*, and *w* vertex coordinates. See also **glMap1**.
GL_MAP2_COLOR_4	If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** generate RGBA values. See also **glMap2**.
GL_MAP2_INDEX	If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** generate color indexes. See also **glMap2**.
GL_MAP2_NORMAL	If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** generate normals. See also **glMap2**.
GL_MAP2_TEXTURE_COORD_1	If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** generate *s* texture coordinates. See also **glMap2**.
GL_MAP2_TEXTURE_COORD_2	If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** generate *s* and *t* texture coordinates. See also **glMap2**.
GL_MAP2_TEXTURE_COORD_3	If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** generate *s*, *t*, and *r* texture coordinates. See also **glMap2**.
GL_MAP2_TEXTURE_COORD_4	If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** generate *s*, *t*, *r*, and *q* texture coordinates. See also **glMap2**.
GL_MAP2_VERTEX_3	If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** generate *x*, *y*, and *z* vertex coordinates. See also **glMap2**.
GL_MAP2_VERTEX_4	If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** generate homogeneous *x*, *y*, *z*, and *w* vertex coordinates. See also **glMap2**.
GL_NORMALIZE	If enabled, normal vectors specified with **glNormal** are scaled to unit length after transformation. See **glNormal**.

VALUE	MEANING
GL_POINT_SMOOTH	If enabled, draw points with proper filtering. If disabled, draw aliased points. See **glPointSize**.
GL_POLYGON_OFFSET_FILL	If enabled, and if the polygon is rendered in GL_FILL mode, an offset is added to depth values of a polygon's fragments before the depth comparison is performed. See **glPolygonOffset**.
GL_POLYGON_OFFSET_LINE	If enabled, and if the polygon is rendered in GL_LINE mode, an offset is added to depth values of a polygon's fragments before the depth comparison is performed. See **glPolygonOffset**.
GL_POLYGON_OFFSET_POINT	If enabled, an offset is added to depth values of a polygon's fragments before the depth comparison is performed, if the polygon is rendered in GL_POINT mode. See **glPolygonOffset**.
GL_POLYGON_SMOOTH	If enabled, draw polygons with proper filtering. If disabled, draw aliased polygons. See **glPolygonMode**.
GL_POLYGON_STIPPLE	If enabled, use the current polygon stipple pattern when rendering polygons. See **glPolygonStipple**.
GL_SCISSOR_TEST	If enabled, discard fragments that are outside the scissor rectangle. See **glScissor**.
GL_STENCIL_TEST	If enabled, do stencil testing and update the stencil buffer. See **glStencilFunc** and **glStencilOp**.
GL_TEXTURE_1D	If enabled, one-dimensional texturing is performed (unless two-dimensional texturing is also enabled). See **glTexImage1D**.
GL_TEXTURE_2D	If enabled, two-dimensional texturing is performed. See **glTexImage2D**.
GL_TEXTURE_GEN_Q	If enabled, the *q* texture coordinate is computed using the texture-generation function defined with **glTexGen**. Otherwise, the current *q* texture coordinate is used.
GL_TEXTURE_GEN_R	If enabled, the *r* texture coordinate is computed using the texture generation function defined with **glTexGen**. If disabled, the current *r* texture coordinate is used.
GL_TEXTURE_GEN_S	If enabled, the *s* texture coordinate is computed using the texture generation function defined with **glTexGen**. If disabled, the current *s* texture coordinate is used.

VALUE	MEANING
GL_TEXTURE_GEN_T	If enabled, the *t* texture coordinate is computed using the texture generation function defined with **glTexGen**. If disabled, the current *t* texture coordinate is used.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glAlphaFunc

glArrayElem

e
n
t

g
l
B
e
g
i
n

g
l
B
l
e
n
d
F
u
n
c

g
l
C
l
i
p
P
l
a
n
e

g
l
C
o
l
o
r
M
a
t
e
r
i
a
l

g

l
C
o
l
o
r
P
o
i
n
t
e
r

g
l
C
u
l
l
F
a
c
e

g
l
D
e
p
t
h
F
u
n
c

g
l
D
e
p
t
h
R
a
n
g
e

g
l
D

i
s
a
b
l
e

g
l
D
r
a
w
A
r
r
a
y
s

g
l
E
d
g
e
F
l
a
g
P
o
i
n
t
e
r

g
l
E
n
d

g
l
E
v
a
l
C
o
o

r
d
1

g
l
E
v
a
l
M
e
s
h
1

g
l
E
v
a
l
P
o
i
n
t
1

g
l
F
o
g

g
l
G
e
t

g
l
I
n
d
e
x
P
o
i
n
t

e
r

g
l
l
s
E
n
a
b
l
e
d

g
l
L
i
g
h
t

g
l
L
i
g
h
t
M
o
d
e
l

g
l
L
i
n
e
W
i
d
t
h

g
l
L
i
n

e
S
t
i
p
p
l
e

g
l
L
o
g
i
c
O
p

g
l
M
a
p
1

g
l
M
a
p
2

g
l
M
a
t
e
r
i
a
l

g
l
N
o
r
m
a
l

g

l
N
o
r
m
a
l
P
o
i
n
t
e
r

g
l
P
o
i
n
t
S
i
z
e

g
l
P
o
l
y
g
o
n
M
o
d
e

g
l
P
o
l
y
g
o
n
S
t
i

p
p
l
e

g
l
S
c
i
s
s
o
r

g
l
S
t
e
n
c
i
l
F
u
n
c

g
l
S
t
e
n
c
i
l
O
p

g
l
T
e
x
C
o
o
r
d
P
o

i
n
t
e
r

g
l
T
e
x
G
e
n

g
l
T
e
x
I
m
a
g
e
1
D

g
l
T
e
x
I
m
a
g
e
2
D

glEnableClientState function

2/22/2020 • 2 minutes to read • Edit Online

The **glEnableClientState** and **glDisableClientState** functions enable and disable arrays respectively.

Syntax

```
void WINAPI glEnableClientState(
   GLenum array
);
```

Parameters

array

A symbolic constant for the array you want to enable or disable. This parameter can assume one of the following values.

VALUE	MEANING
GL_COLOR_ARRAY	If enabled, use color arrays with calls to **glArrayElement**, **glDrawElements**, or **glDrawArrays**. See also **glColorPointer**.
GL_EDGE_FLAG_ARRAY	If enabled, use edge flag arrays with calls to **glArrayElement**, **glDrawElements**, or **glDrawArrays**. See also **glEdgeFlagPointer**.
GL_INDEX_ARRAY	If enabled, use index arrays with calls to **glArrayElement**, **glDrawElements**, or **glDrawArrays**. See also **glIndexPointer**.

VALUE	MEANING
GL_NORMAL_ARRAY	If enabled, use normal arrays with calls to glArrayElement, glDrawElements, or glDrawArrays. See also glNormalPointer.
GL_TEXTURE_COORD_ARRAY	If enabled, use texture coordinate arrays with calls to glArrayElement, glDrawElements, or glDrawArrays. See also glTexCoordPointer.
GL_VERTEX_ARRAY	If enabled, use vertex arrays with calls to glArrayElement, glDrawElements, or glDrawArrays. See also glVertexPointer.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*array* was not an accepted value.

Error codes

The following error code can be retrieved by the glGetError function.

NAME	MEANING
GL_ INV ALI D_E NU M	*array* was not an accepted value.

Remarks

The **glEnableClientState** and **glDisableClientState** functions enable and disable various individual arrays. Use **glIsEnabled** or **glGet** to determine the current setting of any capability.

Calling **glEnableClientState** and **glDisableClientState** between calls to **glBegin** and the corresponding call to **glEnd** can cause an error. If no error is generated, the behavior is undefined.

> **NOTE**
>
> The **glEnableClientState** and **glDisableClientState** functions are only available in OpenGL version 1.1 or later.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib
DLL	Ope ngl3 2.dll

See also

g
l
A
r
r
a
y

E
l
e
m
e
n
t

g
l
B
e
g
i
n

g
l
C
o
l
o
r
P
o
i
n
t
e
r

g
l
D
i
s
a
b
l
e
C
l
i
e
n
t
S
t
a
t
e

g

lDrawArrays

glDrawElements

glEdgeFlagPointer

glEnable

glEnd

glGetPointerv

glIndexPointer

glInterleavedArra

y
s

g
l
N
o
r
m
a
l
P
o
i
n
t
e
r

g
l
T
e
x
C
o
o
r
d
P
o
i
n
t
e
r

g
l
V
e
r
t
e
x
P
o
i
n
t
e
r

glEnd function

2/22/2020 • 2 minutes to read • Edit Online

The glBegin and glend functions delimit the vertices of a primitive or a group of like primitives.

Syntax

```
void WINAPI glEnd(void);
```

Parameters

This function has no parameters.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_OPERATION	A function other than glVertex, glColor, glIndex, glNormal, glTexCoord, glEvalCoord, glEvalPoint, glMaterial, glEdgeFlag, glCallList, or glCallLists was called between glBegin and the corresponding glEnd. The function glEnd was called before the corresponding glBegin was called, or glBegin was called within a glBegin/glEnd sequence.

Error codes

The following error code can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_OPERATION	A function other than glVertex, glColor, glIndex, glNormal, glTexCoord, glEvalCoord, glEvalPoint, glMaterial, glEdgeFlag, glCallList, or glCallLists was called between glBegin and the corresponding glEnd. The function glEnd was called before the corresponding glBegin was called, or glBegin was called within a glBegin/glEnd sequence.

Remarks

The glBegin and glend functions delimit the vertices that define a primitive or a group of like primitives. The glBegin function accepts a single argument that specifies which of ten primitives the vertices compose. Taking *n* as an integer count starting at one, and *N* as the total number of vertices specified, the interpretations are as follows:

- You can use only a subset of OpenGL functions between glBegin and glEnd. The functions you can use are:
  - glVertex
  - glColor
  - glIndex
  - glNormal
  - glTexCoord
  - glEvalCoord
  - glEvalPoint
  - glMaterial
  - glEdgeFlag

  You can also use glCallList or glCallLists to execute display lists that include only the preceding functions. If any other OpenGL function is called between glBegin and glEnd, the error flag is set and the function is ignored.
- Regardless of the value chosen for *mode* in glBegin, there is no limit to the number of vertices you can define between glBegin and glEnd. Lines, triangles, quadrilaterals, and polygons that are incompletely specified are not drawn. Incomplete specification

results when either too few vertices are provided to specify even a single primitive or when an incorrect multiple of vertices is specified. The incomplete primitive is ignored; the complete primitives are drawn.

- The minimum specification of vertices for each primitive is:

MINIMUM NUMBER OF VERTICES	TYPE OF PRIMITIVE
1	point
2	line
3	triangle
4	quadrilateral
3	polygon

- Modes that require a certain multiple of vertices are GL_LINES (2), GL_TRIANGLES (3), GL_QUADS (4), and GL_QUAD_STRIP (2).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glCallLists

glColor

glEdgeFlag

g
l
E
v
a
l
C
o
o
r
d

g
l
E
v
a
l
P
o
i
n
t

g
l
I
n
d
e
x

g
l
M
a
t
e
r
i
a
l

g
l
N
o
r
m
a
l

g
l
T
e
x
C
o
o
r
d

g
l
V
e
r
t
e
x

glEndList function

2/22/2020 • 2 minutes to read • Edit Online

The **glNewList** and **glEndList** functions create or replace a display list.

Syntax

```
void WINAPI glEndList(void);
```

Parameters

This function has no parameters.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	**glEndList** was called without a preceding **glNewList**, or if **glnewlist** was called while a display list was being defined.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	**glEndList** was called without a preceding **glNewList**, or if **glnewlist** was called while a display list was being defined.

Remarks

Display lists are groups of OpenGL commands that have been stored for subsequent execution. The display lists are created with **glNewList**. All subsequent commands are placed in the display list, in the order issued, until **glEndList** is called.

The **glNewList** function has two parameters. The first parameter, *list*, is a positive integer that becomes the unique name for the display list. Names can be created and reserved with **glGenLists** and tested for uniqueness with **glIsList**. The second parameter, *mode*, is a symbolic constant that can assume one of the two preceding values.

Certain commands are not compiled into the display list, but are executed immediately, regardless of the display list mode. These commands are **glColorPointer**, **glDeleteLists**, **glDisableClientState**, **glEdgeFlagPointer**, **glEnableClientState**, **glFeedbackBuffer**, **glFinish**, **glFlush**, **glGenLists**, **glIndexPointer**, **glInterleavedArrays**, **glIsEnabled**, **glIsList**, **glNormalPointer**, **glPopClientAttrib**, **glPixelStore**, **glPushClientAttrib**, **glReadPixels**, **glRenderMode**, **glSelectBuffer**, **glTexCoordPointer**, **glVertexPointer**, and all of the **glGet** routines.

Similarly, **glTexImage2D** and **glTexImage1D** are executed immediately and not compiled into the display list when their first argument is GL_PROXY_TEXTURE_2D or GL_PROXY_TEXTURE_1D, respectively.

When the **glEndList** function is encountered, the display list definition is completed by associating the list with the unique name *list* (specified in the **glNewList** command). If a display list with name *list* already exists, it is replaced only when **glEndList** is called.

The **glCallList** and **glCallLists** functions can be entered into display lists. The commands in the display list or lists executed by **glCallList** or **glCallLists** are not included in the display list being created, even if the list creation mode is GL_COMPILE_AND_EXECUTE.

The following function retrieves information related to **glNewList**:

glGet with argument GL_MATRIX_MODE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

g
l

B
e
g
i
n

g
l
C
a
l
l
L
i
s
t

g
l
C
a
l
l
L
i
s
t
s

g
l
D
e
l
e
t
e
L
i
s
t
s

g
l
E
n
d

g
l
G
e

n
L
i
s
t
s

g
l
l
s
L
i
s
t

g
l
N
e
w
L
i
s
t

glEvalCoord Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions evaluate enabled one-dimensional or two-dimensional maps:

- glEvalCoord1d
- glEvalCoord1f
- glEvalCoord2d
- glEvalCoord2f
- glEvalCoord1dv
- glEvalCoord2dv
- glEvalCoord1fv
- glEvalCoord2fv

glEvalCoord1d function

2/22/2020 • 2 minutes to read • Edit Online

The **glEvalCoord1d** function evaluates enabled one-dimensional maps.

Syntax

```
void WINAPI glEvalCoord1d(
   GLdouble u
);
```

Parameters

u	A value that is the domain coordinate *u* to the basis function defined in a previous **glMap1** function.

Return value

This function does not return a value.

Remarks

The **glEvalCoord1d** function evaluates enabled one-dimensional maps at argument *u*. Define maps with **glMap1**. Enable or disable them with **glEnable** and **glDisable**.

When one of the **glEvalCoord** functions is issued, all currently enabled maps of the indicated dimension are evaluated. Then, for each enabled map, it is as if the corresponding OpenGL function were issued with the computed value. That is, if GL_MAP1_INDEX or GL_MAP2_INDEX is enabled, a **glIndex** function is simulated. If GL_MAP1_COLOR_4 or GL_MAP2_COLOR_4 is enabled, a **glcolor** function is simulated. If GL_MAP1_NORMAL or GL_MAP2_NORMAL is enabled, a normal vector is produced, and if any of GL_MAP1_TEXTURE_COORD_1, GL_MAP1_TEXTURE_COORD_2, GL_MAP1_TEXTURE_COORD_3, GL_MAP1_TEXTURE_COORD_4, GL_MAP2_TEXTURE_COORD_1, GL_MAP2_TEXTURE_COORD_2, GL_MAP2_TEXTURE_COORD_3, and GL_MAP2_TEXTURE_COORD_4 is enabled, then an appropriate **glTexCoord** function is simulated.

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

The following functions retrieve information related to the **glEvalCoord1d** function:

glIsEnabled with argument GL_MAP1_VERTEX_3

glIsEnabled with argument GL_MAP1_VERTEX_4

glIsEnabled with argument GL_MAP1_INDEX

glIsEnabled with argument GL_MAP1_COLOR_4

glIsEnabled with argument GL_MAP1_NORMAL

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_1

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_2

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_3

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_4

glIsEnabled with argument GL_MAP2_VERTEX_3

glIsEnabled with argument GL_MAP2_VERTEX_4

glIsEnabled with argument GL_MAP2_INDEX

glIsEnabled with argument GL_MAP2_COLOR_4

glIsEnabled with argument GL_MAP2_NORMAL

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_1

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_2

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_3

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_4

glIsEnabled with argument GL_AUTO_NORMAL

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glDisable

glEnable

glEnd

glEvalMesh

glEvalPoi

n
t

g
l
G
e
t
M
a
p

g
l
I
n
d
e
x

g
l
I
s
E
n
a
b
l
e
d

g
l
M
a
p
1

g
l
M
a
p
2

g
l
M
a
p
G
r
i

d

glNormal

glTexCoord

glVertex

glEvalCoord1dv function

2/22/2020 • 2 minutes to read • Edit Online

The **glEvalCoord1dv** function evaluates enabled one-dimensional maps.

Syntax

```
void WINAPI glEvalCoord1dv(
   const GLdouble *u
);
```

Parameters

u	A pointer to an array containing the domain coordinate *u*.

Return value

This function does not return a value.

Remarks

The **glEvalCoord1dv** function evaluates enabled one-dimensional maps at argument *u*. Define maps with **glMap1**. Enable or disable them with **glEnable** and **glDisable**.

When one of the **glEvalCoord** functions is issued, all currently enabled maps of the indicated dimension are evaluated. Then, for each enabled map, it is as if the corresponding OpenGL function were issued with the computed value. That is, if GL_MAP1_INDEX or GL_MAP2_INDEX is enabled, a **glIndex** function is simulated. If GL_MAP1_COLOR_4 or GL_MAP2_COLOR_4 is enabled, a **glcolor** function is simulated. If GL_MAP1_NORMAL or GL_MAP2_NORMAL is enabled, a normal vector is produced, and if any of GL_MAP1_TEXTURE_COORD_1, GL_MAP1_TEXTURE_COORD_2, GL_MAP1_TEXTURE_COORD_3, GL_MAP1_TEXTURE_COORD_4, GL_MAP2_TEXTURE_COORD_1, GL_MAP2_TEXTURE_COORD_2, GL_MAP2_TEXTURE_COORD_3, and GL_MAP2_TEXTURE_COORD_4 is enabled, then an appropriate **glTexCoord** function is simulated.

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

The following functions retrieve information related to the **glEvalCoord1dv** function:

glIsEnabled with argument GL_MAP1_VERTEX_3

glIsEnabled with argument GL_MAP1_VERTEX_4

glIsEnabled with argument GL_MAP1_INDEX

glIsEnabled with argument GL_MAP1_COLOR_4

glIsEnabled with argument GL_MAP1_NORMAL

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_1

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_2

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_3

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_4

glIsEnabled with argument GL_MAP2_VERTEX_3

glIsEnabled with argument GL_MAP2_VERTEX_4

glIsEnabled with argument GL_MAP2_INDEX

glIsEnabled with argument GL_MAP2_COLOR_4

glIsEnabled with argument GL_MAP2_NORMAL

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_1

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_2

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_3

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_4

glIsEnabled with argument GL_AUTO_NORMAL

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glDisable

glEnable

glEnd

glEvalMesh

glEvalPoi

n
t

g
l
G
e
t
M
a
p

g
l
I
n
d
e
x

g
l
I
s
E
n
a
b
l
e
d

g
l
M
a
p
1

g
l
M
a
p
2

g
l
M
a
p
G
r
i

d

glNormal

glTexCoord

glVertex

glEvalCoord1f function

2/22/2020 • 2 minutes to read • Edit Online

The **glEvalCoord1f** function evaluates enabled one-dimensional maps.

Syntax

```
void WINAPI glEvalCoord1f(
   GLfloat u
);
```

Parameters

u	A value that is the domain coordinate *u* to the basis function defined in a previous **glMap1** function.

Return value

This function does not return a value.

Remarks

The **glEvalCoord1f** function evaluates enabled one-dimensional maps at argument *u*. Define maps with **glMap1**. Enable or disable them with **glEnable** and **glDisable**.

When one of the **glEvalCoord** functions is issued, all currently enabled maps of the indicated dimension are evaluated. Then, for each enabled map, it is as if the corresponding OpenGL function were issued with the computed value. That is, if GL_MAP1_INDEX or GL_MAP2_INDEX is enabled, a **glIndex** function is simulated. If GL_MAP1_COLOR_4 or GL_MAP2_COLOR_4 is enabled, a **glcolor** function is simulated. If GL_MAP1_NORMAL or GL_MAP2_NORMAL is enabled, a normal vector is produced, and if any of GL_MAP1_TEXTURE_COORD_1, GL_MAP1_TEXTURE_COORD_2, GL_MAP1_TEXTURE_COORD_3, GL_MAP1_TEXTURE_COORD_4, GL_MAP2_TEXTURE_COORD_1, GL_MAP2_TEXTURE_COORD_2, GL_MAP2_TEXTURE_COORD_3, and GL_MAP2_TEXTURE_COORD_4 is enabled, then an appropriate **glTexCoord** function is simulated.

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

The following functions retrieve information related to the **glEvalCoord1f** function:

glIsEnabled with argument GL_MAP1_VERTEX_3

glIsEnabled with argument GL_MAP1_VERTEX_4

glIsEnabled with argument GL_MAP1_INDEX

glIsEnabled with argument GL_MAP1_COLOR_4

glIsEnabled with argument GL_MAP1_NORMAL

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_1

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_2

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_3

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_4

glIsEnabled with argument GL_MAP2_VERTEX_3

glIsEnabled with argument GL_MAP2_VERTEX_4

glIsEnabled with argument GL_MAP2_INDEX

glIsEnabled with argument GL_MAP2_COLOR_4

glIsEnabled with argument GL_MAP2_NORMAL

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_1

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_2

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_3

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_4

glIsEnabled with argument GL_AUTO_NORMAL

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glDisable

glEnable

glEnd

glEvalMesh

glEvalPoi

n
t

g
l
G
e
t
M
a
p

g
l
I
n
d
e
x

g
l
I
s
E
n
a
b
l
e
d

g
l
M
a
p
1

g
l
M
a
p
2

g
l
M
a
p
G
r
i

d

glNormal

glTexCoord

glVertex

glEvalCoord1fv function

2/22/2020 • 2 minutes to read • Edit Online

The **glEvalCoord1fv** function evaluates enabled one-dimensional maps.

Syntax

```
void WINAPI glEvalCoord1fv(
   const GLfloat *u
);
```

Parameters

u	A pointer to an array containing the domain coordinate *u*.

Return value

This function does not return a value.

Remarks

The **glEvalCoord1fv** function evaluates enabled one-dimensional maps at argument *u*. Define maps with **glMap1**. Enable or disable them with **glEnable** and **glDisable**.

When one of the **glEvalCoord** functions is issued, all currently enabled maps of the indicated dimension are evaluated. Then, for each enabled map, it is as if the corresponding OpenGL function were issued with the computed value. That is, if GL_MAP1_INDEX or GL_MAP2_INDEX is enabled, a **glIndex** function is simulated. If GL_MAP1_COLOR_4 or GL_MAP2_COLOR_4 is enabled, a **glcolor** function is simulated. If GL_MAP1_NORMAL or GL_MAP2_NORMAL is enabled, a normal vector is produced, and if any of GL_MAP1_TEXTURE_COORD_1, GL_MAP1_TEXTURE_COORD_2, GL_MAP1_TEXTURE_COORD_3, GL_MAP1_TEXTURE_COORD_4, GL_MAP2_TEXTURE_COORD_1, GL_MAP2_TEXTURE_COORD_2, GL_MAP2_TEXTURE_COORD_3, and GL_MAP2_TEXTURE_COORD_4 is enabled, then an appropriate **glTexCoord** function is simulated.

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

The following functions retrieve information related to the **glEvalCoord1fv** function:

glIsEnabled with argument GL_MAP1_VERTEX_3

glIsEnabled with argument GL_MAP1_VERTEX_4

glIsEnabled with argument GL_MAP1_INDEX

glIsEnabled with argument GL_MAP1_COLOR_4

glIsEnabled with argument GL_MAP1_NORMAL

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_1

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_2

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_3

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_4

glIsEnabled with argument GL_MAP2_VERTEX_3

glIsEnabled with argument GL_MAP2_VERTEX_4

glIsEnabled with argument GL_MAP2_INDEX

glIsEnabled with argument GL_MAP2_COLOR_4

glIsEnabled with argument GL_MAP2_NORMAL

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_1

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_2

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_3

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_4

glIsEnabled with argument GL_AUTO_NORMAL

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glDisable

glEnable

glEnd

glEvalMesh

glEvalPoi

n
t

g
l
G
e
t
M
a
p

g
l
I
n
d
e
x

g
l
I
s
E
n
a
b
l
e
d

g
l
M
a
p
1

g
l
M
a
p
2

g
l
M
a
p
G
r
i

d

glNormal

glTexCoord

glVertex

glEvalCoord2d function

2/22/2020 • 2 minutes to read • Edit Online

The **glEvalCoord2d** function evaluates enabled two-dimensional maps.

Syntax

```
void WINAPI glEvalCoord2d(
   GLdouble u,
   GLdouble v
);
```

Parameters

u	A value that is the domain coordinate *u* to the basis function defined in a previous **glMap2** function.
v	A value that is the domain coordinate *v* to the basis function defined in a previous **glMap2** function.

Return value

This function does not return a value.

Remarks

The **glEvalCoord2d** function evaluates enabled two-dimensional maps using two domain values, *u* and *v*. Define maps with **glMap1** and **glMap2**. Enable or disable them with **glEnable** and **glDisable**.

When one of the **glEvalCoord** functions is issued, all currently enabled maps of the indicated dimension are evaluated. Then, for each enabled map, it is as if the corresponding OpenGL function were issued with the computed value. That is, if GL_MAP1_INDEX or GL_MAP2_INDEX is enabled, a **glIndex** function is simulated. If GL_MAP1_COLOR_4 or GL_MAP2_COLOR_4 is enabled, a **glcolor** function is simulated. If GL_MAP1_NORMAL or GL_MAP2_NORMAL is enabled, a normal vector is produced, and if any of GL_MAP1_TEXTURE_COORD_1, GL_MAP1_TEXTURE_COORD_2, GL_MAP1_TEXTURE_COORD_3, GL_MAP1_TEXTURE_COORD_4, GL_MAP2_TEXTURE_COORD_1, GL_MAP2_TEXTURE_COORD_2, GL_MAP2_TEXTURE_COORD_3, and GL_MAP2_TEXTURE_COORD_4 is enabled, then an appropriate **glTexCoord** function is simulated.

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

If automatic normal generation is enabled, **glEvalCoord2d** calls **glEnable** with argument GL_AUTO_NORMAL to generate surface normals analytically, regardless of the contents or enabling of the GL_MAP2_NORMAL map. Let

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

The generated normal **n** is

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

The following functions retrieve information related to the **glEvalCoord2d** function:

glIsEnabled with argument GL_MAP1_VERTEX_3

glIsEnabled with argument GL_MAP1_VERTEX_4

glIsEnabled with argument GL_MAP1_INDEX

glIsEnabled with argument GL_MAP1_COLOR_4

glIsEnabled with argument GL_MAP1_NORMAL

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_1

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_2

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_3

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_4

glIsEnabled with argument GL_MAP2_VERTEX_3

glIsEnabled with argument GL_MAP2_VERTEX_4

glIsEnabled with argument GL_MAP2_INDEX

glIsEnabled with argument GL_MAP2_COLOR_4

glIsEnabled with argument GL_MAP2_NORMAL

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_1

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_2

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_3

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_4

glIsEnabled with argument GL_AUTO_NORMAL

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glBegin

glColor

glDisable

glEnable

glEnd

glE

v
a
l
M
e
s
h

g
l
E
v
a
l
P
o
i
n
t

g
l
G
e
t
M
a
p

g
l
I
n
d
e
x

g
l
I
s
E
n
a
b
l
e
d

g
l
M
a

p
1

g
l
M
a
p
2

g
l
M
a
p
G
r
i
d

g
l
N
o
r
m
a
l

g
l
T
e
x
C
o
o
r
d

g
l
V
e
r
t
e
x

glEvalCoord2dv function

2/22/2020 • 2 minutes to read • Edit Online

The **glEvalCoord2dv** function evaluates enabled two-dimensional maps.

Syntax

```
void WINAPI glEvalCoord2dv(
   const GLdouble *u
);
```

Parameters

u	A pointer to an array containing the domain coordinate *u*.

Return value

This function does not return a value.

Remarks

The **glEvalCoord2dv** function evaluates enabled two-dimensional maps using two domain values, *u* and *v*. Define maps with **glMap1**. Enable or disable them with **glEnable** and **glDisable**.

When one of the **glEvalCoord** functions is issued, all currently enabled maps of the indicated dimension are evaluated. Then, for each enabled map, it is as if the corresponding OpenGL function were issued with the computed value. That is, if GL_MAP1_INDEX or GL_MAP2_INDEX is enabled, a **glIndex** function is simulated. If GL_MAP1_COLOR_4 or GL_MAP2_COLOR_4 is enabled, a **glcolor** function is simulated. If GL_MAP1_NORMAL or GL_MAP2_NORMAL is enabled, a normal vector is produced, and if any of GL_MAP1_TEXTURE_COORD_1, GL_MAP1_TEXTURE_COORD_2, GL_MAP1_TEXTURE_COORD_3, GL_MAP1_TEXTURE_COORD_4, GL_MAP2_TEXTURE_COORD_1, GL_MAP2_TEXTURE_COORD_2, GL_MAP2_TEXTURE_COORD_3, and GL_MAP2_TEXTURE_COORD_4 is enabled, then an appropriate **glTexCoord** function is simulated.

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

If automatic normal generation is enabled, **glEvalCoord2dv** calls **glEnable** with argument GL_AUTO_NORMAL to generate surface normals analytically, regardless of the contents or enabling of the GL_MAP2_NORMAL map. Let

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

The generated normal **n** is

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

The following functions retrieve information related to the **glEvalCoord2dv** function:

glIsEnabled with argument GL_MAP1_VERTEX_3

glIsEnabled with argument GL_MAP1_VERTEX_4

glIsEnabled with argument GL_MAP1_INDEX

glIsEnabled with argument GL_MAP1_COLOR_4

glIsEnabled with argument GL_MAP1_NORMAL

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_1

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_2

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_3

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_4

glIsEnabled with argument GL_MAP2_VERTEX_3

glIsEnabled with argument GL_MAP2_VERTEX_4

glIsEnabled with argument GL_MAP2_INDEX

glIsEnabled with argument GL_MAP2_COLOR_4

glIsEnabled with argument GL_MAP2_NORMAL

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_1

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_2

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_3

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_4

glIsEnabled with argument GL_AUTO_NORMAL

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glDisable

glEnable

glEnd

glEvalMesh

gl

E
v
a
l
P
o
i
n
t

g
l
G
e
t
M
a
p

g
l
I
n
d
e
x

g
l
I
s
E
n
a
b
l
e
d

g
l
M
a
p
1

g
l
M
a
p
2

g

l
M
a
p
G
r
i
d

g
l
N
o
r
m
a
l

g
l
T
e
x
C
o
o
r
d

g
l
V
e
r
t
e
x

glEvalCoord2f function

2/22/2020 • 2 minutes to read • Edit Online

The **glEvalCoord2f** function evaluates enabled two-dimensional maps.

Syntax

```
void WINAPI glEvalCoord2f(
   GLfloat u,
   GLfloat v
);
```

Parameters

u	A value that is the domain coordinate *u* to the basis function defined in a previous **glMap2** function.
v	A value that is the domain coordinate *v* to the basis function defined in a previous **glMap2** function.

Return value

This function does not return a value.

Remarks

The **glEvalCoord2f** function evaluates enabled two-dimensional maps using two domain values, *u* and *v*. Define maps with **glMap2**. Enable or disable them with **glEnable** and **glDisable**.

When one of the **glEvalCoord** functions is issued, all currently enabled maps of the indicated dimension are evaluated. Then, for each enabled map, it is as if the corresponding OpenGL function were issued with the computed value. That is, if GL_MAP1_INDEX or GL_MAP2_INDEX is enabled, a **glIndex** function is simulated. If GL_MAP1_COLOR_4 or GL_MAP2_COLOR_4 is enabled, a **glcolor** function is simulated. If GL_MAP1_NORMAL or GL_MAP2_NORMAL is enabled, a normal vector is produced, and if any of GL_MAP1_TEXTURE_COORD_1, GL_MAP1_TEXTURE_COORD_2, GL_MAP1_TEXTURE_COORD_3, GL_MAP1_TEXTURE_COORD_4, GL_MAP2_TEXTURE_COORD_1, GL_MAP2_TEXTURE_COORD_2, GL_MAP2_TEXTURE_COORD_3, and GL_MAP2_TEXTURE_COORD_4 is enabled, then an appropriate **glTexCoord** function is simulated.

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

If automatic normal generation is enabled, **glEvalCoord2f** calls **glEnable** with argument GL_AUTO_NORMAL to generate surface normals analytically, regardless of the contents or enabling of the GL_MAP2_NORMAL map. Let

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

The generated normal **n** is

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

The following functions retrieve information related to the **glEvalCoord2f** function:

glIsEnabled with argument GL_MAP1_VERTEX_3

glIsEnabled with argument GL_MAP1_VERTEX_4

glIsEnabled with argument GL_MAP1_INDEX

glIsEnabled with argument GL_MAP1_COLOR_4

glIsEnabled with argument GL_MAP1_NORMAL

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_1

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_2

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_3

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_4

glIsEnabled with argument GL_MAP2_VERTEX_3

glIsEnabled with argument GL_MAP2_VERTEX_4

glIsEnabled with argument GL_MAP2_INDEX

glIsEnabled with argument GL_MAP2_COLOR_4

glIsEnabled with argument GL_MAP2_NORMAL

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_1

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_2

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_3

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_4

glIsEnabled with argument GL_AUTO_NORMAL

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glBegin

glColor

glDisable

glEnable

glEnd

glE

v
a
l
M
e
s
h

g
l
E
v
a
l
P
o
i
n
t

g
l
G
e
t
M
a
p

g
l
I
n
d
e
x

g
l
I
s
E
n
a
b
l
e
d

g
l
M
a

p
1

g
l
M
a
p
2

g
l
M
a
p
G
r
i
d

g
l
N
o
r
m
a
l

g
l
T
e
x
C
o
o
r
d

g
l
V
e
r
t
e
x

glEvalCoord2fv function

2/22/2020 • 2 minutes to read • Edit Online

The **glEvalCoord2fv** function evaluates enabled two-dimensional maps.

Syntax

```
void WINAPI glEvalCoord2fv(
   const GLfloat *u
);
```

Parameters

u	A pointer to an array containing the domain coordinate *u*.

Return value

This function does not return a value.

Remarks

The **glEvalCoord2fv** function evaluates enabled two-dimensional maps using two domain values, *u* and *v*. Define maps with **glMap1**. Enable or disable them with **glEnable** and **glDisable**.

When one of the **glEvalCoord** functions is issued, all currently enabled maps of the indicated dimension are evaluated. Then, for each enabled map, it is as if the corresponding OpenGL function were issued with the computed value. That is, if GL_MAP1_INDEX or GL_MAP2_INDEX is enabled, a **glIndex** function is simulated. If GL_MAP1_COLOR_4 or GL_MAP2_COLOR_4 is enabled, a **glcolor** function is simulated. If GL_MAP1_NORMAL or GL_MAP2_NORMAL is enabled, a normal vector is produced, and if any of GL_MAP1_TEXTURE_COORD_1, GL_MAP1_TEXTURE_COORD_2, GL_MAP1_TEXTURE_COORD_3, GL_MAP1_TEXTURE_COORD_4, GL_MAP2_TEXTURE_COORD_1, GL_MAP2_TEXTURE_COORD_2, GL_MAP2_TEXTURE_COORD_3, and GL_MAP2_TEXTURE_COORD_4 is enabled, then an appropriate **glTexCoord** function is simulated.

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

If automatic normal generation is enabled, **glEvalCoord2fv** calls **glEnable** with argument GL_AUTO_NORMAL to generate surface normals analytically, regardless of the contents or enabling of the GL_MAP2_NORMAL map. Let

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

The generated normal **n** is

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

The following functions retrieve information related to the **glEvalCoord2fv** function:

glIsEnabled with argument GL_MAP1_VERTEX_3

glIsEnabled with argument GL_MAP1_VERTEX_4

glIsEnabled with argument GL_MAP1_INDEX

glIsEnabled with argument GL_MAP1_COLOR_4

glIsEnabled with argument GL_MAP1_NORMAL

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_1

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_2

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_3

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_4

glIsEnabled with argument GL_MAP2_VERTEX_3

glIsEnabled with argument GL_MAP2_VERTEX_4

glIsEnabled with argument GL_MAP2_INDEX

glIsEnabled with argument GL_MAP2_COLOR_4

glIsEnabled with argument GL_MAP2_NORMAL

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_1

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_2

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_3

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_4

glIsEnabled with argument GL_AUTO_NORMAL

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glDisable

glEnable

glEnd

glEvalMesh

gl

E
v
a
l
P
o
i
n
t

g
l
G
e
t
M
a
p

g
l
I
n
d
e
x

g
l
I
s
E
n
a
b
l
e
d

g
l
M
a
p
1

g
l
M
a
p
2

g

l
M
a
p
G
r
i
d

g
l
N
o
r
m
a
l

g
l
T
e
x
C
o
o
r
d

g
l
V
e
r
t
e
x

glEvalMesh Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions compute a one-dimensional or a two-dimensional grid of points or lines:

- glEvalMesh1
- glEvalMesh2

glEvalMesh1 function

2/22/2020 • 2 minutes to read • Edit Online

Computes a one-dimensional grid of points or lines.

Syntax

```
void WINAPI glEvalMesh1(
   GLenum mode,
   GLint  i1,
   GLint  i2
);
```

Parameters

mode

A value that specifies whether to compute a one-dimensional mesh of points or lines. The following symbolic constants are accepted: GL_POINT and GL_LINE.

i1 The first integer value for grid domain variable i.

i2 The last integer value for grid domain variable i.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	Indicates that *mode* is not an accepted value.

NAME	MEANING
GL_ INV ALI D_ OP ERA TIO N	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_ INV ALI D_E NU M	Indicates that *mode* is not an accepted value.
GL_ INV ALI D_ OP ERA TIO N	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

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

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

```
glBegin(type);

for (i = i1; i <= i2; i += 1)

{

glEvalCoord1(i?u + u1)

}

glEnd( );
```

where

?u = (u2 u1) / n

and n, u1, and u2 are the arguments to the most recent **glMapGrid1** function. The *type* parameter is GL_POINTS if mode is GL_POINT, or GL_LINES if mode is GL_LINE. The one absolute numeric requirement is that if i = n, then the value computed from i?u + u1 is exactly u2.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEvalMesh2 function

2/22/2020 • 2 minutes to read • Edit Online

Computes a two-dimensional grid of points or lines.

Syntax

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

Parameters

mode

A value that specifies whether to compute a two-dimensional mesh of points, lines, or polygons. The following symbolic constants are accepted: GL_POINT, GL_LINE, and GL_FILL.

i1 The first integer value for grid domain variable i.

i2 The last integer value for grid domain variable i.

j1 The first integer value for grid domain variable j.

j2 The last integer value for grid domain variable j.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	Indicates that *mode* is not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	Indicates that *mode* is not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

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

In the two-dimensional case, **glEvalMesh2**, let

? u = (u2 u1)/n

? v = (v2 v1)/m,

where n, u1, u2, m, v1, and v2 are the arguments to the most recent **glMapGrid2** function. Then, if *mode* is GL_FILL, **glEvalMesh2** is equivalent to:

```
for (j = j1; j < j2; j += 1)
{
glBegin(GL_QUAD_STRIP);
for (i = i1; i <= i2; i += 1)
{
glEvalCoord2(i? u + u1 ( ) , j ? v + v1);
glEvalCoord2(i? u + u1 ( ) , (j+1) ? v + v1);
}
glEnd( ); }
```

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

```
for (j = j1; j <= j2; j += 1)
{
glBegin(GL_LINE_STRIP);
for (i = i1; i <= i2; i += 1)
{
glEvalCoord2(i? u + u1, j? v + v1);
}
glEnd( );
}
for (i = i1; i <= i2; i += 1)
{
glBegin(GL_LINE_STRIP);
for (j = j1; j <= j1; j += 1)
{
glEvalCoord2(i? u + u1, j? v + v1);
}
glEnd( );
}
```

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

```
glBegin(GL_POINTS);
for (j = j1; j <= j2; j += 1)
{
```

```
for (i = i1; i <= i2; i += 1)
{
glEvalCoord2(i? u + u1, j? v + v1);
}
}
glEnd( );
```

In all three cases, the only absolute numeric requirements are that if i = n, then the value computed from i? u + u1 is exactly u2, and if j = m, then the value computed from j? v + v1 is exactly v2. The following functions retrieve information relating to **glEvalMesh**:

glGet with argument GL_MAP1_GRID_DOMAIN

glGet with argument GL_MAP2_GRID_DOMAIN

glGet with argument GL_MAP1_GRID_SEGMENTS

glGet with argument GL_MAP2_GRID_SEGMENTS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

glEvalPoint Functions

2/22/2020 • 2 minutes to read • Edit Online

Generates and evaluates a single point in a mesh:

- glEvalPoint1
- glEvalPoint2

glEvalPoint1 function

2/22/2020 • 2 minutes to read • Edit Online

The **glEvalPoint1** and **glEvalPoint2** functions generate and evaluate a single point in a mesh.

Syntax

```
void glEvalPoint1(
   GLint i
);
```

Parameters

i	The integer value for grid domain variable *i*.

Return value

This function does not return a value.

Remarks

The **glMapGrid** and **glEvalMesh** functions are used in tandem to efficiently generate and evaluate a series of evenly spaced map domain values. You can use **glEvalPoint** to evaluate a single grid point in the same gridspace that is traversed by **glEvalMesh**. Calling **glEvalPoint1** is equivalent to calling

glEvalCoord1 ($i ?u + u1$);

where

$?u = (u2 \ u1)/n$

and n, $u1$, and $u2$ are the arguments to the most recent **glMapGrid1** function. The one absolute numeric requirement is that if $i = n$, then the value computed from ($i ?u$ + u1) is exactly $u2$.

In the two-dimensional case, **glEvalPoint2**, let

$?u = (u2 \ u1)/n$

$?v = (v2 \ v1)/m$

where n, $u1$, $u2$, m, $v1$, and $v2$ are the arguments to the most recent **glMapGrid2** function. Then the **glEvalPoint2** function is equivalent to calling

glEvalCoord2 ($i ?u + u1$, $j ?v + v1$);

The only absolute numeric requirements are that if $i=n$, then the value computed from ($i ?u + u1$) is exactly u2 , and if $j = m$, then the value computed from ($j ?v + v1$) is exactly $v2$.

The following functions retrieve information relating to **glEvalPoint1** and **glEvalPoint2**:

glGet with argument GL_MAP1_GRID_DOMAIN

glGet with argument GL_MAP2_GRID_DOMAIN

glGet with argument GL_MAP1_GRID_SEGMENTS

glGet with argument GL_MAP2_GRID_SEGMENTS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glEvalCoord

glEvalMesh

glG

e
t

g
l
M
a
p
1

g
l
M
a
p
2

g
l
M
a
p
G
r
i
d

glEvalPoint2 function

2/22/2020 • 2 minutes to read • Edit Online

The **glEvalPoint1** and **glEvalPoint2** functions generate and evaluate a single point in a mesh.

Syntax

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

Parameters

i	The integer value for grid domain variable *i*.
j	The integer value for grid domain variable *j* .

Return value

This function does not return a value.

Remarks

The **glMapGrid** and **glEvalMesh** functions are used in tandem to efficiently generate and evaluate a series of evenly spaced map domain values. You can use **glEvalPoint** to evaluate a single grid point in the same gridspace that is traversed by **glEvalMesh**. Calling **glEvalPoint1** is equivalent to calling

glEvalCoord1 ($i ?u + u1$);

where

$?u = (u2 \ u1)/n$

and n, $u1$, and $u2$ are the arguments to the most recent **glMapGrid1** function. The one absolute numeric requirement is that if $i = n$, then the value computed from ($i ?u$ + u1) is exactly $u2$.

In the two-dimensional case, **glEvalPoint2**, let

$?u = (u2 \ u1)/n$

$?v = (v2 \ v1)/m$

where n, $u1$, $u2$, m, $v1$, and $v2$ are the arguments to the most recent **glMapGrid2** function. Then the **glEvalPoint2** function is equivalent to calling

glEvalCoord2 ($i ?u + u1$, $j ?v + v1$);

The only absolute numeric requirements are that if $i=n$, then the value computed from ($i ?u + u1$) is exactly u2 , and if $j = m$, then the value computed from ($j ?v + v1$) is exactly $v2$.

The following functions retrieve information relating to **glEvalPoint1** and **glEvalPoint2**:

glGet with argument GL_MAP1_GRID_DOMAIN

glGet with argument GL_MAP2_GRID_DOMAIN

glGet with argument GL_MAP1_GRID_SEGMENTS

glGet with argument GL_MAP2_GRID_SEGMENTS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib
DLL	Ope ngl3 2.dll

See also

glEvalCoord

glEvalMesh

glGet

glMap1

glMap2

glMapGrid

glFeedbackBuffer function

2/22/2020 • 4 minutes to read • Edit Online

The **glFeedbackBuffer** function controls feedback mode.

Syntax

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

Parameters

size

The maximum number of values that can be written into *buffer*.

type

A symbolic constant that describes the information that will be returned for each vertex. The following symbolic constants are accepted: GL_2D, GL_3D, GL_3D_COLOR, GL_3D_COLOR_TEXTURE, and GL_4D_COLOR_TEXTURE.

buffer

Returns the feedback data.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*type* was not an accepted value.
GL_INVALID_ENUM	*size* was negative.
GL_INVALID_OPERATION	**glFeedbackBuffer** was called while the render mode was GL_FEEDBACK, or **glRenderMode** was called with argument GL_FEEDBACK before **glFeedbackBuffer** was called at least once.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*type* was not an accepted value.

NAME	MEANING
GL_INVALID_ENUM	*size* was negative.
GL_INVALID_OPERATION	**glFeedbackBuffer** was called while the render mode was GL_FEEDBACK, or **glRenderMode** was called with argument GL_FEEDBACK before **glFeedbackBuffer** was called at least once.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glFeedbackBuffer** function controls feedback. Feedback, like selection, is an OpenGL mode. The mode is selected by calling **glRenderMode** with GL_FEEDBACK. When OpenGL is in feedback mode, no pixels are produced by rasterization. Instead, information about primitives that would have been rasterized is fed back to the application using OpenGL.

The **glFeedbackBuffer** function has three arguments:

- *buffer* is a pointer to an array of floating-point values into which feedback information is placed.
- *size* indicates the size of the array.
- *type* is a symbolic constant describing the information that is fed back for each vertex.

You must issue **glFeedbackBuffer** before feedback mode is enabled (by calling **glRenderMode** with argument GL_FEEDBACK). Setting GL_FEEDBACK without establishing the feedback buffer, or calling **glFeedbackBuffer** while OpenGL is in feedback mode, is an error.

Take OpenGL out of feedback mode by calling **glRenderMode** with a parameter value other than GL_FEEDBACK. When you do this while OpenGL is in feedback mode, **glRenderMode** returns the number of entries placed in the feedback array. The returned value never exceeds *size*. If the feedback data required more room than was available in *buffer*, **glRenderMode** returns a negative value.

While in feedback mode, each primitive that would be rasterized generates a block of values that gets copied into the feedback array. If doing so would cause the number of entries to exceed the maximum, **glFeedbackBuffer** partially writes the block so as to fill the array (if there is any room left at all), and sets an overflow flag. Each block begins with a code indicating the primitive type, followed by values that describe the primitive's vertices and associated data. The **glFeedbackBuffer** function also writes entries for bitmaps and pixel rectangles.

Feedback occurs after polygon culling and **glPolygonMode** interpretation of polygons has taken place, so polygons that are culled are not returned in the feedback buffer. It can also occur after polygons with more than three edges are broken up into triangles, if the OpenGL implementation renders polygons by performing this decomposition.

You can insert a marker into the feedback buffer with **glPassThrough**.

The following is the grammar for the blocks of values written into the feedback buffer. Each primitive is indicated with a unique identifying value followed by some number of vertices. Polygon entries include an integer value indicating how many vertices follow. A vertex is fed back as some number of floating-point values, as determined by *type*. Colors are fed back as four values in RGBA mode and one value in color-index mode.

feedbackList < feedbackItem feedbackList | feedbackItem

feedbackItem < point | lineSegment | polygon | bitmap | pixelRectangle | passThru

point < GL_POINT_TOKEN vertex

lineSegment < GL_LINE_TOKEN vertex vertex | GL_LINE_RESET_TOKEN vertex vertex

polygon < GL_POLYGON_TOKEN n polySpec

polySpec < polySpec vertex | vertex vertex vertex

bitmap < GL_BITMAP_TOKEN vertex

pixelRectangle < GL_DRAW_PIXEL_TOKEN vertex | GL_COPY_PIXEL_TOKEN vertex

passThru < GL_PASS_THROUGH_TOKEN value

vertex < 2d | 3d | 3dColor | 3dColorTexture | 4dColorTexture

2d < value value

3d < value value value

3dColor < value value value color

3dColorTexture < value value value color tex

4dColorTexture < value value value value color tex

color < rgba | index

rgba < value value value value

index < value

tex < value value value value

The *value* parameter is a floating-point number, and *n* is a floating-point integer giving the number of vertices in the polygon. The following are symbolic floating-point constants: GL_POINT_TOKEN, GL_LINE_TOKEN, GL_LINE_RESET_TOKEN, GL_POLYGON_TOKEN, GL_BITMAP_TOKEN, GL_DRAW_PIXEL_TOKEN, GL_COPY_PIXEL_TOKEN, and GL_PASS_THROUGH_TOKEN. GL_LINE_RESET_TOKEN is returned whenever the line stipple pattern is reset. The data returned as a vertex depends on the feedback *type*.

The following table gives the correspondence between *type* and the number of values per vertex; *k* is 1 in color-index mode and 4 in RGBA mode.

TYPE	COORDINATES	COLOR	TEXTURE	TOTAL NUMBER OF VALUES
GL_2D	*x, y*			2
GL_3D	*x, y, z*			3
GL_3D_COLOR	*x, y, z*	*k*		3 + *k*
GL_3D_COLOR_TEXTURE	*x, y, z,*	*k*	4	7 + *k*
GL_4D_COLOR_TEXTURE	*x, y, z, w*	*k*	4	8 + *k*

Feedback vertex coordinates are in window coordinates, except *w*, which is in clip coordinates. Feedback colors are lighted, if lighting is enabled. Feedback texture coordinates are generated, if texture coordinate generation is enabled. They are always transformed by the texture matrix.

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

The following function retrieves information related to **glFeedbackBuffer**:

glGet with argument GL_RENDER_MODE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegi

n

glEnd

glGet

glLineStipple

glPassThrough

glPolygonMo

d
e

g
l
R
e
n
d
e
r
M
o
d
e

g
l
S
e
l
e
c
t
B
u
f
f
e
r

glFinish function

2/22/2020 • 2 minutes to read • Edit Online

The **glFinish** function blocks until all OpenGL execution is complete.

Syntax

```
void WINAPI glFinish(void);
```

Parameters

This function has no parameters.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glFinish** function does not return until the effects of all previously called OpenGL functions are complete. Such effects include all changes to the OpenGL state, all changes to the connection state, and all changes to the framebuffer contents.

The **glFinish** function requires a round trip to the server.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glFlush

glFlush function

2/22/2020 • 2 minutes to read • Edit Online

The **glFlush** function forces execution of OpenGL functions in finite time.

Syntax

```
void WINAPI glFlush(void);
```

Parameters

This function has no parameters.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

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

Because any OpenGL program might be executed over a network, or on an accelerator that buffers commands, be sure to call **glFlush** in any programs requiring that all of their previously issued commands have been completed. For example, call **glFlush** before waiting for user input that depends on the generated image.

The **glFlush** function can return at any time. It does not wait until the execution of all previously issued OpenGL functions is complete.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glFin

i
s
h

glFog Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions specify fog parameters:

- glFogf
- glFogi
- glFogfv
- glFogiv

glFogf function

2/22/2020 • 2 minutes to read • Edit Online

The **glFogf** and function specifies fog parameters.

Syntax

```
void WINAPI glFogf(
   GLenum  pname,
   GLfloat param
);
```

Parameters

pname

Specifies a single-valued fog parameter.

Accepts one of the following values.

VALUE	MEANING
GL_FOG_MODE	The *params* parameter is a single floating-point value that specifies the equation to be used to compute the fog blend factor, *f*. Three symbolic constants are accepted: GL_LINEAR, GL_EXP, and GL_EXP2. The equations corresponding to these symbolic constants are defined in the following Remarks section. The default fog mode is GL_EXP.

VALUE	MEANING
GL_FOG_DENSITY	The *params* parameter is a single floating-point value that specifies *density*, the fog density used in both exponential fog equations. Only nonnegative densities are accepted. The default fog density is 1.0.
GL_FOG_START	The *params* parameter is a single floating-point value that specifies *start*, the near distance used in the linear fog equation. The default near distance is 0.0.
GL_FOG_END	The *params* parameter is a single floating-point value that specifies *end*, the far distance used in the linear fog equation. The default far distance is 1.0.
GL_FOG_INDEX	The *params* parameter is a single floating-point value that specifies i_f , the fog color index. The default fog index is 0.0.

p
a
r
a
m

Specifies the value that *pname* will be set to.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*pname* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*pname* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

You enable and disable fog with **glEnable** and **glDisable**, using the argument GL_FOG. While enabled, fog affects rasterized geometry, bitmaps, and pixel blocks, but not buffer-clear operations.

The **glFogf** function assigns the value or values in *params* to the fog parameter specified by *pname*.

Fog blends a fog color with each rasterized pixel fragment's posttexturing color using a blending factor *f*. Factor *f* is computed in one of three ways, depending on the fog mode. Let *z* be the distance in eye coordinates from the

origin to the fragment being fogged. The equation for GL_LINEAR fog is:

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

The equation for GL_EXP fog is:

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

The equation for GL_EXP2 fog is:

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

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

$$C_r' = f\,C_r + (1-f)\,C_f$$

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

$$i_r' = i_r + (1-f)\,i_f$$

The following functions retrieve information related to the **glFog** functions:

glGet with argument GL_FOG_COLOR

glGet with argument GL_FOG_INDEX

glGet with argument GL_FOG_DENSITY

glGet with argument GL_FOG_START

glGet with argument GL_FOG_END

glGet with argument GL_FOG_MODE

glIsEnabled with argument GL_FOG

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glBegin

glDisable

glEnable

glEnd

glGet

glIsE

n
a
b
l
e
d

glFogfv function

2/22/2020 • 2 minutes to read • Edit Online

The **glFogfv** function specifies fog parameters.

Syntax

```
void WINAPI glFogfv(
         GLenum  pname,
   const GLfloat *params
);
```

Parameters

pname	Specifies a fog parameter. Accepts one of the following values. | VALUE	MEANING
GL_FOG_MODE	The *params* parameter is a floating-point value that specifies the equation to be used to compute the fog blend factor, *f*. Three symbolic constants are accepted: GL_LINEAR, GL_EXP, and GL_EXP2. The equations corresponding to these symbolic constants are defined in the following Remarks section. The default fog mode is GL_EXP.

VALUE	MEANING
GL_FOG_DENSITY	The *params* parameter is a floating-point value that specifies *density*, the fog density used in both exponential fog equations. Only nonnegative densities are accepted. The default fog density is 1.0.
GL_FOG_START	The *params* parameter is a floating-point value that specifies *start*, the near distance used in the linear fog equation. The default near distance is 0.0.
GL_FOG_END	The *params* parameter is a floating-point value that specifies *end*, the far distance used in the linear fog equation. The default far distance is 1.0.
GL_FOG_INDEX	The *params* parameter is a floating-point value that specifies i_f , the fog color index. The default fog index is 0.0.

VALUE	MEANING
GL_FOG_COLOR	The *params* parameter contains four floating-point values that specify C_f , the fog color. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. After conversion, all color components are clamped to the range [0,1]. The default fog color is (0,0,0,0).

params

Specifies the value or values to be assigned to *pname*. GL_FOG_COLOR requires an array of four values. All other parameters accept an array containing only a single value.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*pname* was not an accepted value.

NAME	MEANING
GL_ INV ALI D_ OP ERA TIO N	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_ INV ALI D_E NU M	*pname* was not an accepted value.
GL_ INV ALI D_ OP ERA TIO N	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

You enable and disable fog with **glEnable** and **glDisable**, using the argument GL_FOG. While enabled, fog affects rasterized geometry, bitmaps, and pixel blocks, but not buffer-clear operations.

The **glFogfv** function assigns the value or values in *params* to the fog parameter specified by *pname*.

Fog blends a fog color with each rasterized pixel fragment's posttexturing color using a blending factor *f*. Factor *f* is computed in one of three ways, depending on the fog mode. Let *z* be the distance in eye coordinates from the origin to the fragment being fogged. The equation for GL_LINEAR fog is:

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

The equation for GL_EXP fog is:

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

The equation for GL_EXP2 fog is:

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

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

$$C_r' = f\,C_r + (1-f)\,C_f$$

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

$$i_r' = i_r + (1-f)\,i_f$$

The following functions retrieve information related to the **glFog** functions:

glGet with argument GL_FOG_COLOR

glGet with argument GL_FOG_INDEX

glGet with argument GL_FOG_DENSITY

glGet with argument GL_FOG_START

glGet with argument GL_FOG_END

glGet with argument GL_FOG_MODE

glIsEnabled with argument GL_FOG

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glDisable

glEnable

glEnd

glGet

glIsEnabled

glFogi function

2/22/2020 • 2 minutes to read • Edit Online

The **glFogi** function specifies fog parameters.

Syntax

```
void WINAPI glFogi(
   GLenum pname,
   GLint  param
);
```

Parameters

pname

Specifies a single-valued fog parameter.

Accepts one of the following values.

VALUE	MEANING
GL_FOG_MODE	The *params* parameter is a single integer value that specifies the equation to be used to compute the fog blend factor, *f*. Three symbolic constants are accepted: GL_LINEAR, GL_EXP, and GL_EXP2. The equations corresponding to these symbolic constants are defined in the following Remarks section. The default fog mode is GL_EXP.
GL_FOG_DENSITY	The *params* parameter is a single integer value that specifies *density*, the fog density used in both exponential fog equations. Only nonnegative densities are accepted. The default fog density is 1.0.

VALUE	MEANING
GL_FOG_START	The *params* parameter is a single integer value that specifies *start*, the near distance used in the linear fog equation. The default near distance is 0.0.
GL_FOG_END	The *params* parameter is a single integer value that specifies *end*, the far distance used in the linear fog equation. The default far distance is 1.0.
GL_FOG_INDEX	The *params* parameter is a single integer value that specifies i_f , the fog color index. The default fog index is 0.0.

p
a
r
a
m

Specifies the value that *pname* will be set to.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*pname* was not an accepted value.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*pname* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

You enable and disable fog with **glEnable** and **glDisable**, using the argument GL_FOG. While enabled, fog affects rasterized geometry, bitmaps, and pixel blocks, but not buffer-clear operations.

The **glFogi** function assigns the value or values in *params* to the fog parameter specified by *pname*.

Fog blends a fog color with each rasterized pixel fragment's posttexturing color using a blending factor *f*. Factor *f* is computed in one of three ways, depending on the fog mode. Let *z* be the distance in eye coordinates from the origin to the fragment being fogged. The equation for GL_LINEAR fog is:

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

The equation for GL_EXP fog is:

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

The equation for GL_EXP2 fog is:

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

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

$$C_r' = f\,C_r + (1-f)\,C_f$$

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

$$i_r' = i_r + (1-f)\,i_f$$

The following functions retrieve information related to the **glFog** functions:

glGet with argument GL_FOG_COLOR

glGet with argument GL_FOG_INDEX

glGet with argument GL_FOG_DENSITY

glGet with argument GL_FOG_START

glGet with argument GL_FOG_END

glGet with argument GL_FOG_MODE

glIsEnabled with argument GL_FOG

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glDisable

glEnable

glEnd

glGet

glIsEnabled

glFogiv function

2/22/2020 • 2 minutes to read • Edit Online

The **glFogfv** function specifies fog parameters.

Syntax

```
void WINAPI glFogiv(
         GLenum pname,
   const GLint  *params
);
```

Parameters

pname	Specifies a fog parameter. Accepts one of the following values. **VALUE** / **MEANING** **GL_FOG_MODE** — The *params* parameter is a single integer value that specifies the equation to be used to compute the fog blend factor, *f*. Three symbolic constants are accepted: GL_LINEAR, GL_EXP, and GL_EXP2. The equations corresponding to these symbolic constants are defined in the following Remarks section. The default fog mode is GL_EXP.

VALUE	MEANING
GL_FOG_DENSITY	The *params* parameter is a single integer value that specifies *density*, the fog density used in both exponential fog equations. Only nonnegative densities are accepted. The default fog density is 1.0.
GL_FOG_START	The *params* parameter is a single integer value that specifies *start*, the near distance used in the linear fog equation. The default near distance is 0.0.
GL_FOG_END	The *params* parameter is a single integer value that specifies *end*, the far distance used in the linear fog equation. The default far distance is 1.0.
GL_FOG_INDEX	The *params* parameter is a single integer value that specifies i_f , the fog color index. The default fog index is 0.0.

VALUE	MEANING
GL_FOG_COLOR	The *params* parameter contains four integer or floating-point values that specify C_f , the fog color. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. After conversion, all color components are clamped to the range [0,1]. The default fog color is (0,0,0,0).

params

Specifies the value or values to be assigned to *pname*. GL_FOG_COLOR requires an array of four values. All other parameters accept an array containing only a single value.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*pname* was not an accepted value.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*pname* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

You enable and disable fog with **glEnable** and **glDisable**, using the argument GL_FOG. While enabled, fog affects rasterized geometry, bitmaps, and pixel blocks, but not buffer-clear operations.

The **glFogiv** function assigns the value or values in *params* to the fog parameter specified by *pname*.

Fog blends a fog color with each rasterized pixel fragment's posttexturing color using a blending factor *f*. Factor *f* is computed in one of three ways, depending on the fog mode. Let *z* be the distance in eye coordinates from the origin to the fragment being fogged. The equation for GL_LINEAR fog is:

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

The equation for GL_EXP fog is:

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

The equation for GL_EXP2 fog is:

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

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

$$C_r' = f\,C_r + (1-f)\,C_f$$

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

$$i_r' = i_r + (1-f)\,i_f$$

The following functions retrieve information related to the **glFog** functions:

glGet with argument GL_FOG_COLOR

glGet with argument GL_FOG_INDEX

glGet with argument GL_FOG_DENSITY

glGet with argument GL_FOG_START

glGet with argument GL_FOG_END

glGet with argument GL_FOG_MODE

glIsEnabled with argument GL_FOG

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glDisable

glEnable

glEnd

glGet

glIsEnabled

glFrontFace function

2/22/2020 • 2 minutes to read • Edit Online

The **glFrontFace** function defines front-facing and back-facing polygons.

Syntax

```
void WINAPI glFrontFace(
   GLenum mode
);
```

Parameters

m
o
d
e

The orientation of front-facing polygons. GL_CW and GL_CCW are accepted. The default value is GL_CCW.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*mode* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*mode* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

In a scene composed entirely of opaque closed surfaces, back-facing polygons are never visible. Eliminating these invisible polygons has the obvious benefit of speeding up the rendering of the image. You enable and disable elimination of back-facing polygons with **glEnable** and **glDisable** using argument GL_CULL_FACE.

The projection of a polygon to window coordinates is said to have clockwise winding if an imaginary object following the path from its first vertex, its second vertex, and so on, to its last vertex, and finally back to its first vertex, moves in a clockwise direction about the interior of the polygon. The polygon's winding is said to be counterclockwise if the imaginary object following the same path moves in a counterclockwise direction about the interior of the polygon. The **glFrontFace** function specifies whether polygons with clockwise winding in window coordinates, or counterclockwise winding in window coordinates, are taken to be front-facing. Passing GL_CCW to *mode* selects counterclockwise polygons as front-facing; GL_CW selects clockwise polygons as front-facing. By default, counterclockwise polygons are taken to be front-facing.

The following function retrieves information about **glFrontface**:

glGet with argument GL_FRONT_FACE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib

DLL	Opengl32.dll

See also

glBegin

glCullFace

glDisable

glEnable

glEnd

glGet

glLightModel

glFrustum function

2/22/2020 • 2 minutes to read • Edit Online

The **glFrustum** function multiplies the current matrix by a perspective matrix.

Syntax

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

Parameters

left	The coordinate for the left-vertical clipping plane.
right	The coordinate for the right-vertical clipping plane.
bottom	The coordinate for the bottom-horizontal clipping plane.
top	The coordinate for the bottom-horizontal clipping plane.
zNear	The distances to the near-depth clipping plane. Must be positive.
zFar	The distances to the far-depth clipping planes. Must be positive.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*zNear* or *zFar* was not postitive.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*zNear* or *zFar* was not postitive.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glFrustum** function describes a perspective matrix that produces a perspective projection. The (*left*, *bottom*, *zNear*) and (*right*, *top*, *zNear*) parameters specify the points on the near clipping plane that are mapped to the lower-left and upper-right corners of the window, respectively, assuming that the eye is located at (0,0,0). The *zFar*

parameter specifies the location of the far clipping plane. Both *zNear* and *zFar* must be positive. The corresponding matrix is shown in the following image.

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

$$A = \frac{right+left}{right-left} \qquad C = -\frac{far+near}{far-near}$$

$$B = \frac{top+bottom}{top-bottom} \qquad D = -\frac{2\ far\ near}{far-near}$$

The **glFrustum** function multiplies the current matrix by this matrix, with the result replacing the current matrix. That is, if M is the current matrix and F is the frustum perspective matrix, then **glFrustum** replaces M with M F.

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

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

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

roughly *log*2 (*r*) bits of depth buffer precision are lost. Because *r* approaches infinity as *zNear* approaches zero, you should never set *zNear* to zero.

The following functions retrieve information about **glFrustum**:

glGet with argument GL_MATRIX_MODE

glGet with argument GL_MODELVIEW_MATRIX

glGet with argument GL_PROJECTION_MATRIX

glGet with argument GL_TEXTURE_MATRIX

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGet

glMatrixMode

glMultMatrix

glOrt

h
o

g
l
P
o
p
M
a
t
r
i
x

g
l
P
u
s
h
M
a
t
r
i
x

g
l
V
i
e
w
p
o
r
t

glGenLists function

2/22/2020 • 2 minutes to read • Edit Online

The **glGenLists** function generates a contiguous set of empty display lists.

Syntax

```
GLuint WINAPI glGenLists(
   GLsizei range
);
```

Parameters

range	The number of contiguous empty display lists to be generated.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	*range* was negative.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_ INV ALI D_V AL UE	*range* was negative.
GL_ INV ALI D_ OP ERA TIO N	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glGenLists** function has one argument, *range*. It returns an integer *n* such that *range* contiguous empty display lists, named *n*, *n* + 1, . . ., *n* + (*range* - 1), are created. If *range* is zero, if there is no group of *range* contiguous names available, or if any error is generated, then no display lists are generated and zero is returned.

The following function retrieves information related to **glGenLists**:

glIsList

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib
DLL	Ope ngl3 2.dll

See also

g
l

B
e
g
i
n

g
l
C
a
l
l
L
i
s
t

g
l
C
a
l
l
L
i
s
t
s

g
l
D
e
l
e
t
e
L
i
s
t
s

g
l
E
n
d

g
l
l
s

L
i
s
t

g
l
N
e
w
L
i
s
t

glGenTextures function

2/22/2020 • 2 minutes to read • Edit Online

The **glGenTextures** function generates texture names.

Syntax

```
void WINAPI glGenTextures(
   GLsizei n,
   GLuint  *textures
);
```

Parameters

n The number of texture names to be generated.

t
e
x
t
u
r
e
s A pointer to the first element of an array in which the generated texture names are stored.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INV ALI D_V AL UE	*n* was a negative value.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_VALUE	*n* was a negative value.
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Remarks

The **glGenTextures** function returns *n* texture names in the *textures* parameter. The texture names are not necessarily a contiguous set of integers, however, none of the returned names can have been in use immediately prior to calling the **glGenTextures** function. The generated textures assume the dimensionality of the texture target to which they are first bound with the glBindTexture function. Texture names returned by **glGenTextures** are not returned by subsequent calls to **glGenTextures** unless they are first deleted by calling glDeleteTextures.

You cannot include **glGenTextures** in display lists.

> **NOTE**
>
> The **glGenTextures** function is only available in OpenGL version 1.1 or later.

The following function retrieves information related to **glGenTextures**:

- glIsTexture

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glBindTexture

glDeleteTe

x
t
u
r
e
s

g
l
E
n
d

g
l
G
e
t

g
l
G
e
t
T
e
x
P
a
r
a
m
e
t
e
r

g
l
I
s
T
e
x
t
u
r
e

g
l
T
e

x
l
m
a
g
e
1
D

g
l
T
e
x
l
m
a
g
e
2
D

g
l
T
e
x
P
a
r
a
m
e
t
e
r

glGet

2/22/2020 • 2 minutes to read

• Edit Online

The following functions return the value or values of a selected parameter:

- glGetBooleanv
- glGetDoublev
- glGetFloatv
- glGetIntegerv

glGetBooleanv function

2/22/2020 • 25 minutes to read • Edit Online

The **glGetBooleanv** function returns the value or values of a selected parameter.

Syntax

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

Parameters

pname

The parameter value to be returned. The following symbolic constants are accepted.

VALUE	MEANING
GL_ACCUM_ALPHA_BITS	The *params* parameter returns one value: the number of alpha bitplanes in the accumulation buffer.
GL_ACCUM_BLUE_BITS	The *params* parameter returns one value: the number of blue bitplanes in the accumulation buffer.
GL_ACCUM_CLEAR_VALUE	The *params* parameter returns four values: the red, green, blue, and alpha values used to clear the accumulation buffer. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glClearAccum**.

VALUE	MEANING
GL_ACCUM_GREEN_BITS	The *params* parameter returns one value: the number of green bitplanes in the accumulation buffer.
GL_ACCUM_RED_BITS	The *params* parameter returns one value: the number of red bitplanes in the accumulation buffer.
GL_ALPHA_BIAS	The *params* parameter returns one value: the alpha bias factor used during pixel transfers. See **glPixelTransfer**.
GL_ALPHA_BITS	The *params* parameter returns one value: the number of alpha bitplanes in each color buffer.
GL_ALPHA_SCALE	The *params* parameter returns one value: the alpha scale factor used during pixel transfers. See glPixelTransfer.
GL_ALPHA_TEST	The *params* parameter returns a single Boolean value indicating whether alpha testing of fragments is enabled. See **glAlphaFunc**.
GL_ALPHA_TEST_FUNC	The *params* parameter returns one value: the symbolic name of the alpha test function. See **glAlphaFunc**.

VALUE	MEANING
GL_ALPHA_TEST_REF	The *params* parameter returns one value: the reference value for the alpha test. See glAlphaFunc. An integer value, if requested, is linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value.
GL_ATTRIB_STACK_DEPTH	The *params* parameter returns one value: the depth of the attribute stack. If the stack is empty, zero is returned. See glPushAttrib.
GL_AUTO_NORMAL	The *params* parameter returns a single Boolean value indicating whether 2-D map evaluation automatically generates surface normals. See glMap2.
GL_AUX_BUFFERS	The *params* parameter returns one value: the number of auxiliary color buffers.
GL_BLEND	The *params* parameter returns a single Boolean value indicating whether blending is enabled. See glBlendFunc.
GL_BLEND_DST	The *params* parameter returns one value: the symbolic constant identifying the destination blend function. See glBlendFunc.
GL_BLEND_SRC	The *params* parameter returns one value: the symbolic constant identifying the source blend function. See glBlendFunc.

VALUE	MEANING
GL_BLUE_BIAS	The *params* parameter returns one value: the blue bias factor used during pixel transfers. See **glPixelTransfer**.
GL_BLUE_BITS	The *params* parameter returns one value: the number of blue bitplanes in each color buffer.
GL_BLUE_SCALE	The *params* parameter returns one value: the blue scale factor used during pixel transfers. See glPixelTransfer.
GL_CLIENT_ATTRIB_STACK_DEPTH	The *params* parameter returns one value indicating the depth of the attribute stack. The initial value is zero. See **glPushClientAttrib**.
GL_CLIP_PLANE*i*	The *params* parameter returns a single Boolean value indicating whether the specified clipping plane is enabled. See **glClipPlane**.
GL_COLOR_ARRAY	The *params* parameter returns a single Boolean value indicating whether the specified color array is defined. See **glColorPointer**.
GL_COLOR_ARRAY_SIZE	The *params* parameter returns one value, the number of components per color in the color array. See **glColorPointer**.

VALUE	MEANING
GL_COLOR_ARRAY_STRIDE	The *params* parameter returns one value, the byte offset between consecutive colors in the color array. See glColorPointer.
GL_COLOR_ARRAY_TYPE	The *params* parameter returns one value, the data type of each component in the color array. See glColorPointer.
GL_COLOR_CLEAR_VALUE	The *params* parameter returns four values: the red, green, blue, and alpha values used to clear the color buffers. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See glClearColor.
GL_COLOR_LOGIC_OP	The *params* parameter returns a single Boolean value indicating whether a fragment's RGBA color values are merged into the framebuffer using a logical operation. See glLogicOp.
GL_COLOR_MATERIAL	The *params* parameter returns a single Boolean value indicating whether one or more material parameters are tracking the current color. See glColorMaterial.
GL_COLOR_MATERIAL_FACE	The *params* parameter returns one value: a symbolic constant indicating which materials have a parameter that is tracking the current color. See glColorMaterial.

VALUE	MEANING
GL_COLOR_MATERIAL_PARAMETER	The *params* parameter returns one value: a symbolic constant indicating which material parameters are tracking the current color. See glColorMaterial.
GL_COLOR_WRITEMASK	The *params* parameter returns four Boolean values: the red, green, blue, and alpha write enables for the color buffers. See glColorMask.
GL_CULL_FACE	The *params* parameter returns a single Boolean value indicating whether polygon culling is enabled. See glCullFace.
GL_CULL_FACE_MODE	The *params* parameter returns one value: a symbolic constant indicating which polygon faces are to be culled. See glCullFace.
GL_CURRENT_COLOR	The *params* parameter returns four values: the red, green, blue, and alpha values of the current color. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See glColor.
GL_CURRENT_INDEX	The *params* parameter returns one value: the current color index. See glIndex.

VALUE	MEANING
GL_CURRENT_NORMAL	The *params* parameter returns three values: the *x*, *y*, and *z* values of the current normal. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glNormal**.
GL_CURRENT_RASTER_COLOR	The *params* parameter returns four values: the red, green, blue, and alpha values of the current raster position. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glRasterPos**.
GL_CURRENT_RASTER_DISTANCE	The *params* parameter returns one value: the distance from the eye to the current raster position. See **glRasterPos**.
GL_CURRENT_RASTER_INDEX	The *params* parameter returns one value: the color index of the current raster position. See **glRasterPos**.
GL_CURRENT_RASTER_POSITION	The *params* parameter returns four values: the *x*, *y*, *z*, and *w* components of the current raster position. The *x*, *y*, and *z* components are in window coordinates, and *w* is in clip coordinates. See glRasterPos.

VALUE	MEANING
GL_CURRENT_RASTER_POSITION_VALID	The *params* parameter returns a single Boolean value indicating whether the current raster position is valid. See glRasterPos.
GL_CURRENT_RASTER_TEXTURE_COORDS	The *params* parameter returns four values: the *s*, *t*, *r*, and *q* current raster texture coordinates. See glRasterPos and glTexCoord.
GL_CURRENT_TEXTURE_COORDS	The *params* parameter returns four values: the *s*, *t*, *r*, and *q* current texture coordinates. See glTexCoord.
GL_DEPTH_BIAS	The *params* parameter returns one value: the depth bias factor used during pixel transfers. See glPixelTransfer.
GL_DEPTH_BITS	The *params* parameter returns one value: the number of bitplanes in the depth buffer.

VALUE	MEANING
GL_DEPTH_CLEAR_VALUE	The *params* parameter returns one value: the value that is used to clear the depth buffer. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See glClearDepth.
GL_DEPTH_FUNC	The *params* parameter returns one value: the symbolic constant that indicates the depth comparison function. See glDepthFunc.
GL_DEPTH_RANGE	The *params* parameter returns two values: the near and far mapping limits for the depth buffer. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See glDepthRange.
GL_DEPTH_SCALE	The *params* parameter returns one value: the depth scale factor used during pixel transfers. See glPixelTransfer.
GL_DEPTH_TEST	The *params* parameter returns a single Boolean value indicating whether depth testing of fragments is enabled. See glDepthFunc and glDepthRange.
GL_DEPTH_WRITEMASK	The *params* parameter returns a single Boolean value indicating if the depth buffer is enabled for writing. See glDepthMask.
GL_DITHER	The *params* parameter returns a single Boolean value indicating whether dithering of fragment colors and indexes is enabled.

VALUE	MEANING
GL_DOUBLEBUFFER	The *params* parameter returns a single Boolean value indicating whether double buffering is supported.
GL_DRAW_BUFFER	The *params* parameter returns one value: a symbolic constant indicating which buffers are being drawn to. See **glDrawBuffer**.
GL_EDGE_FLAG	The *params* parameter returns a single Boolean value indicating whether the current edge flag is true or false. See glEdgeFlag.
GL_EDGE_FLAG_ARRAY	The *params* parameter returns a single Boolean value indicating whether the edge flag array is enabled. See **glEdgeFlagPointer**.
GL_EDGE_FLAG_ARRAY_STRIDE	The *params* parameter returns one value, the byte offset between consecutive edge flags in the edge flag array. See **glEdgeFlagPointer**.
GL_FOG	The *params* parameter returns a single Boolean value indicating whether fogging is enabled. See **glFog**.

VALUE	MEANING
GL_FOG_COLOR	The *params* parameter returns four values: the red, green, blue, and alpha components of the fog color. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glFog**.
GL_FOG_DENSITY	The *params* parameter returns one value: the fog density parameter. See glFog.
GL_FOG_END	The *params* parameter returns one value: the end factor for the linear fog equation. See **glFog**.
GL_FOG_HINT	The *params* parameter returns one value: a symbolic constant indicating the mode of the fog hint. See **glHint**.
GL_FOG_INDEX	The *params* parameter returns one value: the fog color index. See **glFog**.
GL_FOG_MODE	The *params* parameter returns one value: a symbolic constant indicating which fog equation is selected. See **glFog**.
GL_FOG_START	The *params* parameter returns one value: the start factor for the linear fog equation. See **glFog**.

VALUE	MEANING
GL_FRONT_FACE	The *params* parameter returns one value: a symbolic constant indicating whether clockwise or counterclockwise polygon winding is treated as front-facing. See glFrontFace.
GL_GREEN_BIAS	The *params* parameter returns one value: the green bias factor used during pixel transfers.
GL_GREEN_BITS	The *params* parameter returns one value: the number of green bitplanes in each color buffer.
GL_GREEN_SCALE	The *params* parameter returns one value: the green scale factor used during pixel transfers. See glPixelTransfer.
GL_INDEX_ARRAY	The *params* parameter returns a single Boolean value indicating whether the color index array is enabled. See glIndexPointer.
GL_INDEX_ARRAY_STRIDE	The *params* parameter returns one value, the byte offset between consecutive color indexes in the color index array. See glIndexPointer.
GL_INDEX_ARRAY_TYPE	The *params* parameter returns one value, the data type of indexes in the color index array. The initial value is GL_FLOAT. See glIndexPointer.

VALUE	MEANING
GL_INDEX_BITS	The *params* parameter returns one value: the number of bitplanes in each color-index buffer.
GL_INDEX_CLEAR_VALUE	The *params* parameter returns one value: the color index used to clear the color-index buffers. See glClearIndex.
GL_INDEX_LOGIC_OP	The *params* parameter returns a single Boolean value indicating whether a fragment's index values are merged into the framebuffer using a logical operation. See glLogicOp.
GL_INDEX_MODE	The *params* parameter returns a single Boolean value indicating whether OpenGL is in color-index mode (TRUE) or RGBA mode (FALSE).
GL_INDEX_OFFSET	The *params* parameter returns one value: the offset added to color and stencil indexes during pixel transfers. See glPixelTransfer.
GL_INDEX_SHIFT	The *params* parameter returns one value: the amount that color and stencil indexes are shifted during pixel transfers. See glPixelTransfer.
GL_INDEX_WRITEMASK	The *params* parameter returns one value: a mask indicating which bitplanes of each color-index buffer can be written. See glIndexMask.

VALUE	MEANING
GL_LIGHT*i*	The *params* parameter returns a single Boolean value indicating whether the specified light is enabled. See glLight and glLightModel.
GL_LIGHTING	The *params* parameter returns a single Boolean value indicating whether lighting is enabled. See glLightModel.
GL_LIGHT_MODEL_AMBIENT	The *params* parameter returns four values: the red, green, blue, and alpha components of the ambient intensity of the entire scene. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See glLightModel.
GL_LIGHT_MODEL_LOCAL_VIEWER	The *params* parameter returns a single Boolean value indicating whether specular reflection calculations treat the viewer as being local to the scene. See glLightModel.
GL_LIGHT_MODEL_TWO_SIDE	The *params* parameter returns a single Boolean value indicating whether separate materials are used to compute lighting for front-facing and back-facing polygons. See glLightModel.
GL_LINE_SMOOTH	The *params* parameter returns a single Boolean value indicating whether antialiasing of lines is enabled. See glLineWidth.

VALUE	MEANING
GL_LINE_SMOOTH_HINT	The *params* parameter returns one value: a symbolic constant indicating the mode of the line antialiasing hint. See glHint.
GL_LINE_STIPPLE	The *params* parameter returns a single Boolean value indicating whether stippling of lines is enabled. See glLineStipple.
GL_LINE_STIPPLE_PATTERN	The *params* parameter returns one value: the 16-bit line stipple pattern. See glLineStipple.
GL_LINE_STIPPLE_REPEAT	The *params* parameter returns one value: the line stipple repeat factor. See glLineStipple.
GL_LINE_WIDTH	The *params* parameter returns one value: the line width as specified with glLineWidth.

VALUE	MEANING
GL_LINE_WIDTH_GRANULARITY	The *params* parameter returns one value: the width difference between adjacent supported widths for antialiased lines. See glLineWidth.
GL_LINE_WIDTH_RANGE	The *params* parameter returns two values: the smallest and largest supported widths for antialiased lines. See glLineWidth.
GL_LIST_BASE	The *params* parameter returns one value: the base offset added to all names in arrays presented to glCallLists. See glListBase.
GL_LIST_INDEX	The *params* parameter returns one value: the name of the display list currently under construction. Zero is returned if no display list is currently under construction. See glNewList.
GL_LIST_MODE	The *params* parameter returns one value: a symbolic constant indicating the construction mode of the display list currently being constructed. See glNewList.
GL_LOGIC_OP	The *params* parameter returns a single Boolean value indicating whether fragment indexes are merged into the framebuffer using a logical operation. See glLogicOp.

VALUE	MEANING
GL_LOGIC_OP_MODE	The *params* parameter returns one value: a symbolic constant indicating the selected logic operational mode. See **glLogicOp**.
GL_MAP1_COLOR_4	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates colors. See **glMap1**.
GL_MAP1_GRID_DOMAIN	The *params* parameter returns two values: the endpoints of the 1-D maps grid domain. See glMapGrid.
GL_MAP1_GRID_SEGMENTS	The *params* parameter returns one value: the number of partitions in the 1-D maps grid domain. See glMapGrid.
GL_MAP1_INDEX	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates color indexes. See **glMap1**.
GL_MAP1_NORMAL	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates normals. See **glMap1**.

VALUE	MEANING
GL_MAP1_TEXTURE_COORD_1	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 1-D texture coordinates. See glMap1.
GL_MAP1_TEXTURE_COORD_2	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 2-D texture coordinates. See glMap1.
GL_MAP1_TEXTURE_COORD_3	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 3-D texture coordinates. See glMap1.
GL_MAP1_TEXTURE_COORD_4	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 4-D texture coordinates. See glMap1.
GL_MAP1_VERTEX_3	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 3-D vertex coordinates. See glMap1.

VALUE	MEANING
GL_MAP1_VERTEX_4	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 4-D vertex coordinates. See **glMap1**.
GL_MAP2_COLOR_4	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates colors. See **glMap2**.
GL_MAP2_GRID_DOMAIN	The *params* parameter returns four values: the endpoints of the 2-D maps *i* and *j* grid domains. See glMapGrid.
GL_MAP2_GRID_SEGMENTS	The *params* parameter returns two values: the number of partitions in the 2-D maps *i* and *j* grid domains. See glMapGrid.
GL_MAP2_INDEX	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates color indexes. See **glMap2**.
GL_MAP2_NORMAL	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates normals. See **glMap2**.

VALUE	MEANING
GL_MAP2_TEXTURE_COORD_1	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 1-D texture coordinates. See glMap2.
GL_MAP2_TEXTURE_COORD_2	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 2-D texture coordinates. See glMap2.
GL_MAP2_TEXTURE_COORD_3	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 3-D texture coordinates. See glMap2.
GL_MAP2_TEXTURE_COORD_4	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 4-D texture coordinates. See glMap2.
GL_MAP2_VERTEX_3	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 3-D vertex coordinates. See glMap2.

VALUE	MEANING
GL_MAP2_VERTEX_4	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 4-D vertex coordinates. See glMap2.
GL_MAP_COLOR	The *params* parameter returns a single Boolean value indicating whether colors and color indexes are to be replaced by table lookup during pixel transfers. See glPixelTransfer.
GL_MAP_STENCIL	The *params* parameter returns a single Boolean value indicating whether stencil indexes are to be replaced by table lookup during pixel transfers. See glPixelTransfer.
GL_MATRIX_MODE	The *params* parameter returns one value: a symbolic constant indicating which matrix stack is currently the target of all matrix operations. See glMatrixMode.
GL_MAX_CLIENT_ATTRIB_STACK_DEPTH	The *params* parameter returns one value indicating the maximum supported depth of the client attribute stack. See glPushClientAttrib.
GL_MAX_ATTRIB_STACK_DEPTH	The *params* parameter returns one value: the maximum supported depth of the attribute stack. See glPushAttrib.

VALUE	MEANING
GL_MAX_CLIP_PLANES	The *params* parameter returns one value: the maximum number of application-defined clipping planes. See **glClipPlane**.
GL_MAX_EVAL_ORDER	The *params* parameter returns one value: the maximum equation order supported by 1-D and 2-D evaluators. See **glMap1** and **glMap2**.
GL_MAX_LIGHTS	The *params* parameter returns one value: the maximum number of lights. See glLight.
GL_MAX_LIST_NESTING	The *params* parameter returns one value: the maximum recursion depth allowed during display-list traversal. See **glCallList**.
GL_MAX_MODELVIEW_STACK_DEPTH	The *params* parameter returns one value: the maximum supported depth of the modelview matrix stack. See **glPushMatrix**.
GL_MAX_NAME_STACK_DEPTH	The *params* parameter returns one value: the maximum supported depth of the selection name stack. See **glPushName**.

VALUE	MEANING
GL_MAX_PIXEL_MAP_TABLE	The *params* parameter returns one value: the maximum supported size of a glPixelMap lookup table.
GL_MAX_PROJECTION_STACK_DEPTH	The *params* parameter returns one value: the maximum supported depth of the projection matrix stack. See glPushMatrix.
GL_MAX_TEXTURE_SIZE	The *params* parameter returns one value: the maximum width or height of any texture image (without borders). See glTexImage1D and glTexImage2D.
GL_MAX_TEXTURE_STACK_DEPTH	The *params* parameter returns one value: the maximum supported depth of the texture matrix stack. See glPushMatrix.
GL_MAX_VIEWPORT_DIMS	The *params* parameter returns two values: the maximum supported width and height of the viewport. See glViewport.

VALUE	MEANING
GL_MODELVIEW_MATRIX	The *params* parameter returns 16 values: the modelview matrix on the top of the modelview matrix stack. See glPushMatrix.
GL_MODELVIEW_STACK_DEPTH	The *params* parameter returns one value: the number of matrices on the modelview matrix stack. See glPushMatrix.
GL_NAME_STACK_DEPTH	The *params* parameter returns one value: the number of names on the selection name stack. See glPushName.
GL_NORMAL_ARRAY	The *params* parameter returns a single Boolean value, indicating whether the normal array is enabled. See glNormalPointer.
GL_NORMAL_ARRAY_STRIDE	The *params* parameter returns one value, the byte offset between consecutive normals in the normal array. See glNormalPointer.

VALUE	MEANING
GL_NORMAL_ARRAY_TYPE	The *params* parameter returns one value, the data type of each coordinate in the normal array. See **glNormalPointer**.
GL_NORMALIZE	The *params* parameter returns a single Boolean value indicating whether normals are automatically scaled to unit length after they have been transformed to eye coordinates. See glNormal.
GL_PACK_ALIGNMENT	The *params* parameter returns one value: the byte alignment used for writing pixel data to memory. See glPixelStore.
GL_PACK_LSB_FIRST	The *params* parameter returns a single Boolean value indicating whether single-bit pixels being written to memory are written first to the least significant bit of each unsigned byte. See glPixelStore.
GL_PACK_ROW_LENGTH	The *params* parameter returns one value: the row length used for writing pixel data to memory. See glPixelStore.
GL_PACK_SKIP_PIXELS	The *params* parameter returns one value: the number of pixel locations skipped before the first pixel is written into memory. See glPixelStore.

VALUE	MEANING
GL_PACK_SKIP_ROWS	The *params* parameter returns one value: the number of rows of pixel locations skipped before the first pixel is written into memory. See glPixelStore.
GL_PACK_SWAP_BYTES	The *params* parameter returns a single Boolean value indicating whether the bytes of 2-byte and 4-byte pixel indexes and components are swapped before being written to memory. See glPixelStore.
GL_PERSPECTIVE_CORRECTION_HINT	The *params* parameter returns one value: a symbolic constant indicating the mode of the perspective correction hint. See **glHint**.
GL_PIXEL_MAP_A_TO_A_SIZE	The *params* parameter returns one value: the size of the alpha-to-alpha pixel-translation table. See **glPixelMap**.
GL_PIXEL_MAP_B_TO_B_SIZE	The *params* parameter returns one value: the size of the blue-to-blue pixel-translation table. See **glPixelMap**.

VALUE	MEANING
GL_PIXEL_MAP_G_TO_G_SIZE	The *params* parameter returns one value: the size of the green-to-green pixel-translation table. See **glPixelMap**.
GL_PIXEL_MAP_I_TO_A_SIZE	The *params* parameter returns one value: the size of the index-to-alpha pixel translation table. See **glPixelMap**.
GL_PIXEL_MAP_I_TO_B_SIZE	The *params* parameter returns one value: the size of the index-to-blue pixel translation table. See **glPixelMap**.
GL_PIXEL_MAP_I_TO_G_SIZE	The *params* parameter returns one value: the size of the index-to-green pixel-translation table. See **glPixelMap**.
GL_PIXEL_MAP_I_TO_I_SIZE	The *params* parameter returns one value: the size of the index-to-index pixel-translation table. See **glPixelMap**.

VALUE	MEANING
GL_PIXEL_MAP_I_TO_R_SIZE	The *params* parameter returns one value: the size of the index-to-red pixel-translation table. See glPixelMap.
GL_PIXEL_MAP_R_TO_R_SIZE	The *params* parameter returns one value: the size of the red-to-red pixel-translation table. See glPixelMap.
GL_PIXEL_MAP_S_TO_S_SIZE	The *params* parameter returns one value: the size of the stencil-to-stencil pixel translation table. See glPixelMap.
GL_POINT_SIZE	The *params* parameter returns one value: the point size as specified by glPointSize.
GL_POINT_SIZE_GRANULARITY	The *params* parameter returns one value: the size difference between adjacent supported sizes for antialiased points. See glPointSize.

VALUE	MEANING
GL_POINT_SIZE_RANGE	The *params* parameter returns two values: the smallest and largest supported sizes for antialiased points. See glPointSize.
GL_POINT_SMOOTH	The *params* parameter returns a single Boolean value indicating whether antialiasing of points is enabled. See glPointSize.
GL_POINT_SMOOTH_HINT	The *params* parameter returns one value: a symbolic constant indicating the mode of the point antialiasing hint. See glHint.
GL_POLYGON_MODE	The *params* parameter returns two values: symbolic constants indicating whether front-facing and back-facing polygons are rasterized as points, lines, or filled polygons. See glPolygonMode.
GL_POLYGON_OFFSET_FACTOR	The *params* parameter 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. See glPolygonOffset.

VALUE	MEANING
GL_POLYGON_OFFSET_UNITS	The *params* parameter 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. See glPolygonOffset.
GL_POLYGON_OFFSET_FILL	The *params* parameter returns a single Boolean value indicating whether polygon offset is enabled for polygons in fill mode. See glPolygonOffset.
GL_POLYGON_OFFSET_LINE	The *params* parameter returns a single Boolean value indicating whether polygon offset is enabled for polygons in line mode. See glPolygonOffset.
GL_POLYGON_OFFSET_POINT	The *params* parameter returns a single Boolean value indicating whether polygon offset is enabled for polygons in point mode. See glPolygonOffset.
GL_POLYGON_SMOOTH	The *params* parameter returns a single Boolean value indicating whether antialiasing of polygons is enabled. See glPolygonMode.

VALUE	MEANING
GL_POLYGON_SMOOTH_HINT	The *params* parameter returns one value: a symbolic constant indicating the mode of the polygon antialiasing hint. See glHint.
GL_POLYGON_STIPPLE	The *params* parameter returns a single Boolean value indicating whether stippling of polygons is enabled. See glPolygonStipple.
GL_PROJECTION_MATRIX	The *params* parameter returns 16 values: the projection matrix on the top of the projection matrix stack. See glPushMatrix.
GL_PROJECTION_STACK_DEPTH	The *params* parameter returns one value: the number of matrices on the projection matrix stack. See glPushMatrix.
GL_READ_BUFFER	The *params* parameter returns one value: a symbolic constant indicating which color buffer is selected for reading. See glReadPixels and glAccum.
GL_RED_BIAS	The *params* parameter returns one value: the red bias factor used during pixel transfers. See glPixelTransfer.

VALUE	MEANING
GL_RED_BITS	The *params* parameter returns one value: the number of red bitplanes in each color buffer.
GL_RED_SCALE	The *params* parameter returns one value: the red scale factor used during pixel transfers. See **glPixelTransfer**.
GL_RENDER_MODE	The *params* parameter returns one value: a symbolic constant indicating whether OpenGL is in render, select, or feedback mode. See **glRenderMode**.
GL_RGBA_MODE	The *params* parameter returns a single Boolean value indicating whether OpenGL is in RGBA mode (TRUE) or color-index mode (FALSE). See glColor.
GL_SCISSOR_BOX	The *params* parameter returns four values: the *x* and *y* window coordinates of the scissor box, followed by its width and height. See **glScissor**.
GL_SCISSOR_TEST	The *params* parameter returns a single Boolean value indicating whether scissoring is enabled. See **glScissor**.
GL_SHADE_MODEL	The *params* parameter returns one value: a symbolic constant indicating whether the shading mode is flat or smooth. See **glShadeModel**.
GL_STENCIL_BITS	The *params* parameter returns one value: the number of bitplanes in the stencil buffer.

VALUE	MEANING
GL_STENCIL_CLEAR_VALUE	The *params* parameter returns one value: the index to which the stencil bitplanes are cleared. See glClearStencil.
GL_STENCIL_FAIL	The *params* parameter returns one value: a symbolic constant indicating what action is taken when the stencil test fails. See glStencilOp.
GL_STENCIL_FUNC	The *params* parameter returns one value: a symbolic constant indicating what function is used to compare the stencil reference value with the stencil buffer value. See glStencilFunc.
GL_STENCIL_PASS_DEPTH_FAIL	The *params* parameter returns one value: a symbolic constant indicating what action is taken when the stencil test passes, but the depth test fails. See glStencilOp.
GL_STENCIL_PASS_DEPTH_PASS	The *params* parameter returns one value: a symbolic constant indicating what action is taken when the stencil test passes and the depth test passes. See glStencilOp.
GL_STENCIL_REF	The *params* parameter returns one value: the reference value that is compared with the contents of the stencil buffer. See glStencilFunc.

VALUE	MEANING
GL_STENCIL_TEST	The *params* parameter returns a single Boolean value indicating whether stencil testing of fragments is enabled. See glStencilFunc and glStencilOp.
GL_STENCIL_VALUE_MASK	The *params* parameter returns one value: the mask that is used to mask both the stencil reference value and the stencil buffer value before they are compared. See glStencilFunc.
GL_STENCIL_WRITEMASK	The *params* parameter returns one value: the mask that controls writing of the stencil bitplanes. See glStencilMask.
GL_STEREO	The *params* parameter returns a single Boolean value indicating whether stereo buffers (left and right) are supported.
GL_SUBPIXEL_BITS	The *params* parameter returns one value: an estimate of the number of bits of subpixel resolution that are used to position rasterized geometry in window coordinates.
GL_TEXTURE_1D	The *params* parameter returns a single Boolean value indicating whether 1-D texture mapping is enabled. See glTexImage1D.
GL_TEXTURE_2D	The *params* parameter returns a single Boolean value indicating whether 2-D texture mapping is enabled. See glTexImage2D.

VALUE	MEANING
GL_TEXTURE_COORD_ARRAY	The *params* parameter returns a single Boolean value indicating whether the texture coordinate array is enabled. See **glTexCoordPointer**.
GL_TEXTURE_COORD_ARRAY_SIZE	The *params* parameter returns one value, the number of coordinates per element in the texture coordinate array. See **glTexCoordPointer**.
GL_TEXTURE_COORD_ARRAY_STRIDE	The *params* parameter returns one value, the byte offset between consecutive elements in the texture coordinate array. See **glTexCoordPointer**.
GL_TEXTURE_COORD_ARRAY_TYPE	The *params* parameter params returns one value, the data type of the coordinates in the texture coordinate array. See **glTexCoordPointer**.
GL_TEXTURE_ENV_COLOR	The *params* parameter returns four values: the red, green, blue, and alpha values of the texture environment color. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and 1.0 returns the most negative representable integer value. See glTexEnv.

VALUE	MEANING
GL_TEXTURE_ENV_MODE	The *params* parameter returns one value: a symbolic constant indicating which texture environment function is currently selected. See glTexEnv.
GL_TEXTURE_GEN_Q	The *params* parameter returns a single Boolean value indicating whether automatic generation of the Q texture coordinate is enabled. See **glTexGen**.
GL_TEXTURE_GEN_R	The *params* parameter returns a single Boolean value indicating whether automatic generation of the R texture coordinate is enabled. See **glTexGen**.
GL_TEXTURE_GEN_S	The *params* parameter returns a single Boolean value indicating whether automatic generation of the S texture coordinate is enabled. See **glTexGen**.
GL_TEXTURE_GEN_T	The *params* parameter returns a single Boolean value indicating whether automatic generation of the T texture coordinate is enabled. See **glTexGen**.
GL_TEXTURE_MATRIX	The *params* parameter returns 16 values: the texture matrix on the top of the texture matrix stack. See **glPushMatrix**.

VALUE	MEANING
GL_TEXTURE_STACK_DEPTH	The *params* parameter returns one value: the number of matrices on the texture matrix stack. See **glPushMatrix**.
GL_UNPACK_ALIGNMENT	The *params* parameter returns one value: the byte alignment used for reading pixel data from memory. See glPixelStore.
GL_UNPACK_LSB_FIRST	The *params* parameter returns a single Boolean value indicating whether single-bit pixels being read from memory are read first from the least significant bit of each unsigned byte. See glPixelStore.
GL_UNPACK_ROW_LENGTH	The *params* parameter returns one value: the row length used for reading pixel data from memory. See glPixelStore.
GL_UNPACK_SKIP_PIXELS	The *params* parameter returns one value: the number of pixel locations skipped before the first pixel is read from memory. See glPixelStore.
GL_UNPACK_SKIP_ROWS	The *params* parameter returns one value: the number of rows of pixel locations skipped before the first pixel is read from memory. See glPixelStore.

VALUE	MEANING
GL_UNPACK_SWAP_BYTES	The *params* parameter returns a single Boolean value indicating whether the bytes of 2-byte and 4-byte pixel indexes and components are swapped after being read from memory. See glPixelStore.
GL_VERTEX_ARRAY	The *params* parameter returns a single Boolean value indicating whether the vertex array is enabled. See **glVertexPointer**.
GL_VERTEX_ARRAY_SIZE	The *params* parameter returns one value, the number of coordinates per vertex in the vertex array. See **glVertexPointer**.
GL_VERTEX_ARRAY_STRIDE	The *params* parameter returns one value, the byte offset between consecutive vertexes in the vertex array. See **glVertexPointer**.
GL_VERTEX_ARRAY_TYPE	The *params* parameter returns one value, the data type of each coordinate in the vertex array. See **glVertexPointer**.
GL_VIEWPORT	The *params* parameter returns four values: the *x* and *y* window coordinates of the viewport, followed by its width and height. See **glViewport**.

VALUE	MEANING
GL_ZOOM_X	The *params* parameter returns one value: the *x* pixel zoom factor. See **glPixelZoom**.
GL_ZOOM_Y	The *params* parameter returns one value: the *y* pixel zoom factor. See **glPixelZoom**.

p a r a m s Returns the value or values of the specified parameter.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*pname* was not an accepted value.
GL_INVALID_OPERATION	**glGetBoolean** was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*pname* was not an accepted value.
GL_INVALID_OPERATION	**glGetBoolean** was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

This function returns values for simple state variables in OpenGL. The *pname* parameter is a symbolic constant indicating the state variable to be returned, and *params* is a pointer to an array of the indicated type in which to place the returned data.

Type conversion is performed if *params* has a different type from the state variable value being requested. If you call **glGetBooleanv**, a floating-point or integer value is converted to GL_FALSE if and only if it is zero. Otherwise, it is converted to GL_TRUE.

If you call **glGetIntegerv**, Boolean values are returned as GL_TRUE or GL_FALSE, and most floating-point values are rounded to the nearest integer value. Floating-point colors and normals, however, are returned with a linear mapping that maps 1.0 to the most positive representable integer value and 1.0 to the most negative representable integer value.

If you call **glGetFloatv** or **glGetDoublev**, Boolean values are returned as GL_TRUE or GL_FALSE, and integer values are converted to floating-point values.

You can query many of the Boolean parameters more easily with **glIsEnabled**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib

DLL	Opengl32.dll

See also

glAccum

glAlphaFunc

glBegin

glBlendFunc

glCa

l
l
L
i
s
t

g
l
C
l
e
a
r
A
c
c
u
m

g
l
C
l
e
a
r
C
o
l
o
r

g
l
C
l
e
a
r
D
e
p
t
h

g
l
C
l
e
a
r

Index

glClearStencil

glClipPlane

glColor

glColorMask

glColorMaterial

glCullFace

glDepthFunc

glDepthMask

gl

D
e
p
t
h
R
a
n
g
e

g
l
D
r
a
w
B
u
f
f
e
r

g
l
E
d
g
e
F
l
a
g

g
l
E
n
d

g
l
F
o
g

g
l
F
r
o
n

t
F
a
c
e

g
l
G
e
t
C
l
i
p
P
l
a
n
e

g
l
G
e
t
E
r
r
o
r

g
l
G
e
t
L
i
g
h
t

g
l
G
e
t
M
a
p

g

lGetMaterialg

lGetPixelMapg

lGetPolygonStipple

glGetS

t
r
i
n
g

g
l
G
e
t
T
e
x
E
n
v

g
l
G
e
t
T
e
x
G
e
n

g
l
G
e
t
T
e
x
I
m
a
g
e

g
l
G
e
t
T
e
x
L

e
v
e
l
P
a
r
a
m
e
t
e
r

g
l
G
e
t
T
e
x
P
a
r
a
m
e
t
e
r

g
l
H
i
n
t

g
l
I
n
d
e
x

g
l
I
n
d
e

x
M
a
s
k

g
l
I
s
E
n
a
b
l
e
d

g
l
L
i
g
h
t

g
l
L
i
g
h
t
M
o
d
e
l

g
l
L
i
n
e
S
t
i
p
p
l
e

glLineWidth

glListBase

glLogicOp

glMap1

glMap2

glMapG

r
i
d

g
l
M
a
t
r
i
x
M
o
d
e

g
l
N
e
w
L
i
s
t

g
l
N
o
r
m
a
l

g
l
P
i
x
e
l
M
a
p

g
l
P
i
x
e

lStore

glPixelTransfer

glPixelZoom

glPointSize

glPoly

g
o
n
M
o
d
e

g
l
P
o
l
y
g
o
n
S
t
i
p
p
l
e

g
l
P
u
s
h
A
t
t
r
i
b

g
l
P
u
s
h
M
a
t
r
i
x

g
l

PushName

glRasterPos

glReadPixels

glScissor

glShadeMo

d
e
l

g
l
S
t
e
n
c
i
l
F
u
n
c

g
l
S
t
e
n
c
i
l
M
a
s
k

g
l
S
t
e
n
c
i
l
O
p

g
l
T
e
x
C
o
o
r

d

glTexEnv

glTexGen

glTexImage1D

glTexImage2D

glViewp

o
r
t

glGetDoublev function

2/22/2020 • 25 minutes to read • Edit Online

The **glGetDoublev** function returns the value or values of a selected parameter.

Syntax

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

Parameters

pname

The parameter value to be returned. The following symbolic constants are accepted.

VALUE	MEANING
GL_ACCUM_ALPHA_BITS	The *params* parameter returns one value: the number of alpha bitplanes in the accumulation buffer.
GL_ACCUM_BLUE_BITS	The *params* parameter returns one value: the number of blue bitplanes in the accumulation buffer.
GL_ACCUM_CLEAR_VALUE	The *params* parameter returns four values: the red, green, blue, and alpha values used to clear the accumulation buffer. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glClearAccum**.

VALUE	MEANING
GL_ACCUM_GREEN_BITS	The *params* parameter returns one value: the number of green bitplanes in the accumulation buffer.
GL_ACCUM_RED_BITS	The *params* parameter returns one value: the number of red bitplanes in the accumulation buffer.
GL_ALPHA_BIAS	The *params* parameter returns one value: the alpha bias factor used during pixel transfers. See **glPixelTransfer**.
GL_ALPHA_BITS	The *params* parameter returns one value: the number of alpha bitplanes in each color buffer.
GL_ALPHA_SCALE	The *params* parameter returns one value: the alpha scale factor used during pixel transfers. See glPixelTransfer.
GL_ALPHA_TEST	The *params* parameter returns a single Boolean value indicating whether alpha testing of fragments is enabled. See **glAlphaFunc**.
GL_ALPHA_TEST_FUNC	The *params* parameter returns one value: the symbolic name of the alpha test function. See **glAlphaFunc**.

VALUE	MEANING
GL_ALPHA_TEST_REF	The *params* parameter returns one value: the reference value for the alpha test. See glAlphaFunc. An integer value, if requested, is linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value.
GL_ATTRIB_STACK_DEPTH	The *params* parameter returns one value: the depth of the attribute stack. If the stack is empty, zero is returned. See glPushAttrib.
GL_AUTO_NORMAL	The *params* parameter returns a single Boolean value indicating whether 2-D map evaluation automatically generates surface normals. See glMap2.
GL_AUX_BUFFERS	The *params* parameter returns one value: the number of auxiliary color buffers.
GL_BLEND	The *params* parameter returns a single Boolean value indicating whether blending is enabled. See glBlendFunc.
GL_BLEND_DST	The *params* parameter returns one value: the symbolic constant identifying the destination blend function. See glBlendFunc.
GL_BLEND_SRC	The *params* parameter returns one value: the symbolic constant identifying the source blend function. See glBlendFunc.

VALUE	MEANING
GL_BLUE_BIAS	The *params* parameter returns one value: the blue bias factor used during pixel transfers. See **glPixelTransfer**.
GL_BLUE_BITS	The *params* parameter returns one value: the number of blue bitplanes in each color buffer.
GL_BLUE_SCALE	The *params* parameter returns one value: the blue scale factor used during pixel transfers. See glPixelTransfer.
GL_CLIENT_ATTRIB_STACK_DEPTH	The *params* parameter returns one value indicating the depth of the attribute stack. The initial value is zero. See **glPushClientAttrib**.
GL_CLIP_PLANE*i*	The *params* parameter returns a single Boolean value indicating whether the specified clipping plane is enabled. See **glClipPlane**.
GL_COLOR_ARRAY	The *params* parameter returns a single Boolean value indicating whether the specified color array is defined. See **glColorPointer**.
GL_COLOR_ARRAY_SIZE	The *params* parameter returns one value, the number of components per color in the color array. See **glColorPointer**.

VALUE	MEANING
GL_COLOR_ARRAY_STRIDE	The *params* parameter returns one value, the byte offset between consecutive colors in the color array. See glColorPointer.
GL_COLOR_ARRAY_TYPE	The *params* parameter returns one value, the data type of each component in the color array. See glColorPointer.
GL_COLOR_CLEAR_VALUE	The *params* parameter returns four values: the red, green, blue, and alpha values used to clear the color buffers. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See glClearColor.
GL_COLOR_LOGIC_OP	The *params* parameter returns a single Boolean value indicating whether a fragment's RGBA color values are merged into the framebuffer using a logical operation. See glLogicOp.
GL_COLOR_MATERIAL	The *params* parameter returns a single Boolean value indicating whether one or more material parameters are tracking the current color. See glColorMaterial.
GL_COLOR_MATERIAL_FACE	The *params* parameter returns one value: a symbolic constant indicating which materials have a parameter that is tracking the current color. See glColorMaterial.

VALUE	MEANING
GL_COLOR_MATERIAL_PARAMETER	The *params* parameter returns one value: a symbolic constant indicating which material parameters are tracking the current color. See glColorMaterial.
GL_COLOR_WRITEMASK	The *params* parameter returns four Boolean values: the red, green, blue, and alpha write enables for the color buffers. See glColorMask.
GL_CULL_FACE	The *params* parameter returns a single Boolean value indicating whether polygon culling is enabled. See glCullFace.
GL_CULL_FACE_MODE	The *params* parameter returns one value: a symbolic constant indicating which polygon faces are to be culled. See glCullFace.
GL_CURRENT_COLOR	The *params* parameter returns four values: the red, green, blue, and alpha values of the current color. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See glColor.
GL_CURRENT_INDEX	The *params* parameter returns one value: the current color index. See glIndex.

VALUE	MEANING
GL_CURRENT_NORMAL	The *params* parameter returns three values: the *x*, *y*, and *z* values of the current normal. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glNormal**.
GL_CURRENT_RASTER_COLOR	The *params* parameter returns four values: the red, green, blue, and alpha values of the current raster position. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glRasterPos**.
GL_CURRENT_RASTER_DISTANCE	The *params* parameter returns one value: the distance from the eye to the current raster position. See **glRasterPos**.
GL_CURRENT_RASTER_INDEX	The *params* parameter returns one value: the color index of the current raster position. See **glRasterPos**.
GL_CURRENT_RASTER_POSITION	The *params* parameter returns four values: the *x*, *y*, *z*, and *w* components of the current raster position. The *x*, *y*, and *z* components are in window coordinates, and *w* is in clip coordinates. See glRasterPos.

VALUE	MEANING
GL_CURRENT_RASTER_POSITION_VALID	The *params* parameter returns a single Boolean value indicating whether the current raster position is valid. See glRasterPos.
GL_CURRENT_RASTER_TEXTURE_COORDS	The *params* parameter returns four values: the *s*, *t*, *r*, and *q* current raster texture coordinates. See glRasterPos and glTexCoord.
GL_CURRENT_TEXTURE_COORDS	The *params* parameter returns four values: the *s*, *t*, *r*, and *q* current texture coordinates. See glTexCoord.
GL_DEPTH_BIAS	The *params* parameter returns one value: the depth bias factor used during pixel transfers. See glPixelTransfer.
GL_DEPTH_BITS	The *params* parameter returns one value: the number of bitplanes in the depth buffer.

VALUE	MEANING
GL_DEPTH_CLEAR_VALUE	The *params* parameter returns one value: the value that is used to clear the depth buffer. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See glClearDepth.
GL_DEPTH_FUNC	The *params* parameter returns one value: the symbolic constant that indicates the depth comparison function. See glDepthFunc.
GL_DEPTH_RANGE	The *params* parameter returns two values: the near and far mapping limits for the depth buffer. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See glDepthRange.
GL_DEPTH_SCALE	The *params* parameter returns one value: the depth scale factor used during pixel transfers. See glPixelTransfer.
GL_DEPTH_TEST	The *params* parameter returns a single Boolean value indicating whether depth testing of fragments is enabled. See glDepthFunc and glDepthRange.
GL_DEPTH_WRITEMASK	The *params* parameter returns a single Boolean value indicating if the depth buffer is enabled for writing. See glDepthMask.
GL_DITHER	The *params* parameter returns a single Boolean value indicating whether dithering of fragment colors and indexes is enabled.

VALUE	MEANING
GL_DOUBLEBUFFER	The *params* parameter returns a single Boolean value indicating whether double buffering is supported.
GL_DRAW_BUFFER	The *params* parameter returns one value: a symbolic constant indicating which buffers are being drawn to. See **glDrawBuffer**.
GL_EDGE_FLAG	The *params* parameter returns a single Boolean value indicating whether the current edge flag is true or false. See glEdgeFlag.
GL_EDGE_FLAG_ARRAY	The *params* parameter returns a single Boolean value indicating whether the edge flag array is enabled. See **glEdgeFlagPointer**.
GL_EDGE_FLAG_ARRAY_STRIDE	The *params* parameter returns one value, the byte offset between consecutive edge flags in the edge flag array. See **glEdgeFlagPointer**.
GL_FOG	The *params* parameter returns a single Boolean value indicating whether fogging is enabled. See **glFog**.

VALUE	MEANING
GL_FOG_COLOR	The *params* parameter returns four values: the red, green, blue, and alpha components of the fog color. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glFog**.
GL_FOG_DENSITY	The *params* parameter returns one value: the fog density parameter. See glFog.
GL_FOG_END	The *params* parameter returns one value: the end factor for the linear fog equation. See **glFog**.
GL_FOG_HINT	The *params* parameter returns one value: a symbolic constant indicating the mode of the fog hint. See **glHint**.
GL_FOG_INDEX	The *params* parameter returns one value: the fog color index. See **glFog**.
GL_FOG_MODE	The *params* parameter returns one value: a symbolic constant indicating which fog equation is selected. See **glFog**.
GL_FOG_START	The *params* parameter returns one value: the start factor for the linear fog equation. See **glFog**.

VALUE	MEANING
GL_FRONT_FACE	The *params* parameter returns one value: a symbolic constant indicating whether clockwise or counterclockwise polygon winding is treated as front-facing. See **glFrontFace**.
GL_GREEN_BIAS	The *params* parameter returns one value: the green bias factor used during pixel transfers.
GL_GREEN_BITS	The *params* parameter returns one value: the number of green bitplanes in each color buffer.
GL_GREEN_SCALE	The *params* parameter returns one value: the green scale factor used during pixel transfers. See **glPixelTransfer**.
GL_INDEX_ARRAY	The *params* parameter returns a single Boolean value indicating whether the color index array is enabled. See **glIndexPointer**.
GL_INDEX_ARRAY_STRIDE	The *params* parameter returns one value, the byte offset between consecutive color indexes in the color index array. See **glIndexPointer**.
GL_INDEX_ARRAY_TYPE	The *params* parameter returns one value, the data type of indexes in the color index array. The initial value is GL_FLOAT. See **glIndexPointer**.

VALUE	MEANING
GL_INDEX_BITS	The *params* parameter returns one value: the number of bitplanes in each color-index buffer.
GL_INDEX_CLEAR_VALUE	The *params* parameter returns one value: the color index used to clear the color-index buffers. See **glClearIndex**.
GL_INDEX_LOGIC_OP	The *params* parameter returns a single Boolean value indicating whether a fragment's index values are merged into the framebuffer using a logical operation. See **glLogicOp**.
GL_INDEX_MODE	The *params* parameter returns a single Boolean value indicating whether OpenGL is in color-index mode (TRUE) or RGBA mode (FALSE).
GL_INDEX_OFFSET	The *params* parameter returns one value: the offset added to color and stencil indexes during pixel transfers. See **glPixelTransfer**.
GL_INDEX_SHIFT	The *params* parameter returns one value: the amount that color and stencil indexes are shifted during pixel transfers. See **glPixelTransfer**.
GL_INDEX_WRITEMASK	The *params* parameter returns one value: a mask indicating which bitplanes of each color-index buffer can be written. See **glIndexMask**.

VALUE	MEANING
GL_LIGHT*i*	The *params* parameter returns a single Boolean value indicating whether the specified light is enabled. See **glLight** and **glLightModel**.
GL_LIGHTING	The *params* parameter returns a single Boolean value indicating whether lighting is enabled. See **glLightModel**.
GL_LIGHT_MODEL_AMBIENT	The *params* parameter returns four values: the red, green, blue, and alpha components of the ambient intensity of the entire scene. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glLightModel**.
GL_LIGHT_MODEL_LOCAL_VIEWER	The *params* parameter returns a single Boolean value indicating whether specular reflection calculations treat the viewer as being local to the scene. See **glLightModel**.
GL_LIGHT_MODEL_TWO_SIDE	The *params* parameter returns a single Boolean value indicating whether separate materials are used to compute lighting for front-facing and back-facing polygons. See **glLightModel**.
GL_LINE_SMOOTH	The *params* parameter returns a single Boolean value indicating whether antialiasing of lines is enabled. See **glLineWidth**.

VALUE	MEANING
GL_LINE_SMOOTH_HINT	The *params* parameter returns one value: a symbolic constant indicating the mode of the line antialiasing hint. See glHint.
GL_LINE_STIPPLE	The *params* parameter returns a single Boolean value indicating whether stippling of lines is enabled. See glLineStipple.
GL_LINE_STIPPLE_PATTERN	The *params* parameter returns one value: the 16-bit line stipple pattern. See glLineStipple.
GL_LINE_STIPPLE_REPEAT	The *params* parameter returns one value: the line stipple repeat factor. See glLineStipple.
GL_LINE_WIDTH	The *params* parameter returns one value: the line width as specified with glLineWidth.

VALUE	MEANING
GL_LINE_WIDTH_GRANULARITY	The *params* parameter returns one value: the width difference between adjacent supported widths for antialiased lines. See glLineWidth.
GL_LINE_WIDTH_RANGE	The *params* parameter returns two values: the smallest and largest supported widths for antialiased lines. See glLineWidth.
GL_LIST_BASE	The *params* parameter returns one value: the base offset added to all names in arrays presented to glCallLists. See glListBase.
GL_LIST_INDEX	The *params* parameter returns one value: the name of the display list currently under construction. Zero is returned if no display list is currently under construction. See glNewList.
GL_LIST_MODE	The *params* parameter returns one value: a symbolic constant indicating the construction mode of the display list currently being constructed. See glNewList.
GL_LOGIC_OP	The *params* parameter returns a single Boolean value indicating whether fragment indexes are merged into the framebuffer using a logical operation. See glLogicOp.

VALUE	MEANING
GL_LOGIC_OP_MODE	The *params* parameter returns one value: a symbolic constant indicating the selected logic operational mode. See **glLogicOp**.
GL_MAP1_COLOR_4	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates colors. See **glMap1**.
GL_MAP1_GRID_DOMAIN	The *params* parameter returns two values: the endpoints of the 1-D maps grid domain. See glMapGrid.
GL_MAP1_GRID_SEGMENTS	The *params* parameter returns one value: the number of partitions in the 1-D maps grid domain. See glMapGrid.
GL_MAP1_INDEX	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates color indexes. See **glMap1**.
GL_MAP1_NORMAL	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates normals. See **glMap1**.

VALUE	MEANING
GL_MAP1_TEXTURE_COORD_1	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 1-D texture coordinates. See **glMap1**.
GL_MAP1_TEXTURE_COORD_2	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 2-D texture coordinates. See **glMap1**.
GL_MAP1_TEXTURE_COORD_3	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 3-D texture coordinates. See **glMap1**.
GL_MAP1_TEXTURE_COORD_4	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 4-D texture coordinates. See **glMap1**.
GL_MAP1_VERTEX_3	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 3-D vertex coordinates. See **glMap1**.

VALUE	MEANING
GL_MAP1_VERTEX_4	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 4-D vertex coordinates. See **glMap1**.
GL_MAP2_COLOR_4	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates colors. See **glMap2**.
GL_MAP2_GRID_DOMAIN	The *params* parameter returns four values: the endpoints of the 2-D maps *i* and *j* grid domains. See glMapGrid.
GL_MAP2_GRID_SEGMENTS	The *params* parameter returns two values: the number of partitions in the 2-D maps *i* and *j* grid domains. See glMapGrid.
GL_MAP2_INDEX	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates color indexes. See **glMap2**.
GL_MAP2_NORMAL	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates normals. See **glMap2**.

VALUE	MEANING
GL_MAP2_TEXTURE_COORD_1	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 1-D texture coordinates. See glMap2.
GL_MAP2_TEXTURE_COORD_2	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 2-D texture coordinates. See glMap2.
GL_MAP2_TEXTURE_COORD_3	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 3-D texture coordinates. See glMap2.
GL_MAP2_TEXTURE_COORD_4	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 4-D texture coordinates. See glMap2.
GL_MAP2_VERTEX_3	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 3-D vertex coordinates. See glMap2.

VALUE	MEANING
GL_MAP2_VERTEX_4	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 4-D vertex coordinates. See glMap2.
GL_MAP_COLOR	The *params* parameter returns a single Boolean value indicating whether colors and color indexes are to be replaced by table lookup during pixel transfers. See glPixelTransfer.
GL_MAP_STENCIL	The *params* parameter returns a single Boolean value indicating whether stencil indexes are to be replaced by table lookup during pixel transfers. See glPixelTransfer.
GL_MATRIX_MODE	The *params* parameter returns one value: a symbolic constant indicating which matrix stack is currently the target of all matrix operations. See glMatrixMode.
GL_MAX_CLIENT_ATTRIB_STACK_DEPTH	The *params* parameter returns one value indicating the maximum supported depth of the client attribute stack. See glPushClientAttrib.
GL_MAX_ATTRIB_STACK_DEPTH	The *params* parameter returns one value: the maximum supported depth of the attribute stack. See glPushAttrib.

VALUE	MEANING
GL_MAX_CLIP_PLANES	The *params* parameter returns one value: the maximum number of application-defined clipping planes. See **glClipPlane**.
GL_MAX_EVAL_ORDER	The *params* parameter returns one value: the maximum equation order supported by 1-D and 2-D evaluators. See **glMap1** and **glMap2**.
GL_MAX_LIGHTS	The *params* parameter returns one value: the maximum number of lights. See glLight.
GL_MAX_LIST_NESTING	The *params* parameter returns one value: the maximum recursion depth allowed during display-list traversal. See **glCallList**.
GL_MAX_MODELVIEW_STACK_DEPTH	The *params* parameter returns one value: the maximum supported depth of the modelview matrix stack. See **glPushMatrix**.
GL_MAX_NAME_STACK_DEPTH	The *params* parameter returns one value: the maximum supported depth of the selection name stack. See **glPushName**.

VALUE	MEANING
GL_MAX_PIXEL_MAP_TABLE	The *params* parameter returns one value: the maximum supported size of a glPixelMap lookup table.
GL_MAX_PROJECTION_STACK_DEPTH	The *params* parameter returns one value: the maximum supported depth of the projection matrix stack. See glPushMatrix.
GL_MAX_TEXTURE_SIZE	The *params* parameter returns one value: the maximum width or height of any texture image (without borders). See glTexImage1D and glTexImage2D.
GL_MAX_TEXTURE_STACK_DEPTH	The *params* parameter returns one value: the maximum supported depth of the texture matrix stack. See glPushMatrix.
GL_MAX_VIEWPORT_DIMS	The *params* parameter returns two values: the maximum supported width and height of the viewport. See glViewport.

VALUE	MEANING
GL_MODELVIEW_MATRIX	The *params* parameter returns 16 values: the modelview matrix on the top of the modelview matrix stack. See glPushMatrix.
GL_MODELVIEW_STACK_DEPTH	The *params* parameter returns one value: the number of matrices on the modelview matrix stack. See glPushMatrix.
GL_NAME_STACK_DEPTH	The *params* parameter returns one value: the number of names on the selection name stack. See glPushName.
GL_NORMAL_ARRAY	The *params* parameter returns a single Boolean value, indicating whether the normal array is enabled. See glNormalPointer.
GL_NORMAL_ARRAY_STRIDE	The *params* parameter returns one value, the byte offset between consecutive normals in the normal array. See glNormalPointer.

VALUE	MEANING
GL_NORMAL_ARRAY_TYPE	The *params* parameter returns one value, the data type of each coordinate in the normal array. See **glNormalPointer**.
GL_NORMALIZE	The *params* parameter returns a single Boolean value indicating whether normals are automatically scaled to unit length after they have been transformed to eye coordinates. See glNormal.
GL_PACK_ALIGNMENT	The *params* parameter returns one value: the byte alignment used for writing pixel data to memory. See glPixelStore.
GL_PACK_LSB_FIRST	The *params* parameter returns a single Boolean value indicating whether single-bit pixels being written to memory are written first to the least significant bit of each unsigned byte. See glPixelStore.
GL_PACK_ROW_LENGTH	The *params* parameter returns one value: the row length used for writing pixel data to memory. See glPixelStore.
GL_PACK_SKIP_PIXELS	The *params* parameter returns one value: the number of pixel locations skipped before the first pixel is written into memory. See glPixelStore.

VALUE	MEANING
GL_PACK_SKIP_ROWS	The *params* parameter returns one value: the number of rows of pixel locations skipped before the first pixel is written into memory. See glPixelStore.
GL_PACK_SWAP_BYTES	The *params* parameter returns a single Boolean value indicating whether the bytes of 2-byte and 4-byte pixel indexes and components are swapped before being written to memory. See glPixelStore.
GL_PERSPECTIVE_CORRECTION_HINT	The *params* parameter returns one value: a symbolic constant indicating the mode of the perspective correction hint. See **glHint**.
GL_PIXEL_MAP_A_TO_A_SIZE	The *params* parameter returns one value: the size of the alpha-to-alpha pixel-translation table. See **glPixelMap**.
GL_PIXEL_MAP_B_TO_B_SIZE	The *params* parameter returns one value: the size of the blue-to-blue pixel-translation table. See **glPixelMap**.

VALUE	MEANING
GL_PIXEL_MAP_G_TO_G_SIZE	The *params* parameter returns one value: the size of the green-to-green pixel-translation table. See **glPixelMap**.
GL_PIXEL_MAP_I_TO_A_SIZE	The *params* parameter returns one value: the size of the index-to-alpha pixel translation table. See **glPixelMap**.
GL_PIXEL_MAP_I_TO_B_SIZE	The *params* parameter returns one value: the size of the index-to-blue pixel translation table. See **glPixelMap**.
GL_PIXEL_MAP_I_TO_G_SIZE	The *params* parameter returns one value: the size of the index-to-green pixel-translation table. See **glPixelMap**.
GL_PIXEL_MAP_I_TO_I_SIZE	The *params* parameter returns one value: the size of the index-to-index pixel-translation table. See **glPixelMap**.

VALUE	MEANING
GL_PIXEL_MAP_I_TO_R_SIZE	The *params* parameter returns one value: the size of the index-to-red pixel-translation table. See glPixelMap.
GL_PIXEL_MAP_R_TO_R_SIZE	The *params* parameter returns one value: the size of the red-to-red pixel-translation table. See glPixelMap.
GL_PIXEL_MAP_S_TO_S_SIZE	The *params* parameter returns one value: the size of the stencil-to-stencil pixel translation table. See glPixelMap.
GL_POINT_SIZE	The *params* parameter returns one value: the point size as specified by glPointSize.
GL_POINT_SIZE_GRANULARITY	The *params* parameter returns one value: the size difference between adjacent supported sizes for antialiased points. See glPointSize.

VALUE	MEANING
GL_POINT_SIZE_RANGE	The *params* parameter returns two values: the smallest and largest supported sizes for antialiased points. See glPointSize.
GL_POINT_SMOOTH	The *params* parameter returns a single Boolean value indicating whether antialiasing of points is enabled. See glPointSize.
GL_POINT_SMOOTH_HINT	The *params* parameter returns one value: a symbolic constant indicating the mode of the point antialiasing hint. See glHint.
GL_POLYGON_MODE	The *params* parameter returns two values: symbolic constants indicating whether front-facing and back-facing polygons are rasterized as points, lines, or filled polygons. See glPolygonMode.
GL_POLYGON_OFFSET_FACTOR	The *params* parameter 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. See glPolygonOffset.

VALUE	MEANING
GL_POLYGON_OFFSET_UNITS	The *params* parameter 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. See glPolygonOffset.
GL_POLYGON_OFFSET_FILL	The *params* parameter returns a single Boolean value indicating whether polygon offset is enabled for polygons in fill mode. See glPolygonOffset.
GL_POLYGON_OFFSET_LINE	The *params* parameter returns a single Boolean value indicating whether polygon offset is enabled for polygons in line mode. See glPolygonOffset.
GL_POLYGON_OFFSET_POINT	The *params* parameter returns a single Boolean value indicating whether polygon offset is enabled for polygons in point mode. See glPolygonOffset.
GL_POLYGON_SMOOTH	The *params* parameter returns a single Boolean value indicating whether antialiasing of polygons is enabled. See glPolygonMode.

VALUE	MEANING
GL_POLYGON_SMOOTH_HINT	The *params* parameter returns one value: a symbolic constant indicating the mode of the polygon antialiasing hint. See glHint.
GL_POLYGON_STIPPLE	The *params* parameter returns a single Boolean value indicating whether stippling of polygons is enabled. See glPolygonStipple.
GL_PROJECTION_MATRIX	The *params* parameter returns 16 values: the projection matrix on the top of the projection matrix stack. See glPushMatrix.
GL_PROJECTION_STACK_DEPTH	The *params* parameter returns one value: the number of matrices on the projection matrix stack. See glPushMatrix.
GL_READ_BUFFER	The *params* parameter returns one value: a symbolic constant indicating which color buffer is selected for reading. See glReadPixels and glAccum.
GL_RED_BIAS	The *params* parameter returns one value: the red bias factor used during pixel transfers. See glPixelTransfer.

VALUE	MEANING
GL_RED_BITS	The *params* parameter returns one value: the number of red bitplanes in each color buffer.
GL_RED_SCALE	The *params* parameter returns one value: the red scale factor used during pixel transfers. See **glPixelTransfer**.
GL_RENDER_MODE	The *params* parameter returns one value: a symbolic constant indicating whether OpenGL is in render, select, or feedback mode. See **glRenderMode**.
GL_RGBA_MODE	The *params* parameter returns a single Boolean value indicating whether OpenGL is in RGBA mode (TRUE) or color-index mode (FALSE). See glColor.
GL_SCISSOR_BOX	The *params* parameter returns four values: the *x* and *y* window coordinates of the scissor box, followed by its width and height. See **glScissor**.
GL_SCISSOR_TEST	The *params* parameter returns a single Boolean value indicating whether scissoring is enabled. See **glScissor**.
GL_SHADE_MODEL	The *params* parameter returns one value: a symbolic constant indicating whether the shading mode is flat or smooth. See **glShadeModel**.
GL_STENCIL_BITS	The *params* parameter returns one value: the number of bitplanes in the stencil buffer.

VALUE	MEANING
GL_STENCIL_CLEAR_VALUE	The *params* parameter returns one value: the index to which the stencil bitplanes are cleared. See glClearStencil.
GL_STENCIL_FAIL	The *params* parameter returns one value: a symbolic constant indicating what action is taken when the stencil test fails. See glStencilOp.
GL_STENCIL_FUNC	The *params* parameter returns one value: a symbolic constant indicating what function is used to compare the stencil reference value with the stencil buffer value. See glStencilFunc.
GL_STENCIL_PASS_DEPTH_FAIL	The *params* parameter returns one value: a symbolic constant indicating what action is taken when the stencil test passes, but the depth test fails. See glStencilOp.
GL_STENCIL_PASS_DEPTH_PASS	The *params* parameter returns one value: a symbolic constant indicating what action is taken when the stencil test passes and the depth test passes. See glStencilOp.
GL_STENCIL_REF	The *params* parameter returns one value: the reference value that is compared with the contents of the stencil buffer. See glStencilFunc.

VALUE	MEANING
GL_STENCIL_TEST	The *params* parameter returns a single Boolean value indicating whether stencil testing of fragments is enabled. See glStencilFunc and glStencilOp.
GL_STENCIL_VALUE_MASK	The *params* parameter returns one value: the mask that is used to mask both the stencil reference value and the stencil buffer value before they are compared. See glStencilFunc.
GL_STENCIL_WRITEMASK	The *params* parameter returns one value: the mask that controls writing of the stencil bitplanes. See glStencilMask.
GL_STEREO	The *params* parameter returns a single Boolean value indicating whether stereo buffers (left and right) are supported.
GL_SUBPIXEL_BITS	The *params* parameter returns one value: an estimate of the number of bits of subpixel resolution that are used to position rasterized geometry in window coordinates.
GL_TEXTURE_1D	The *params* parameter returns a single Boolean value indicating whether 1-D texture mapping is enabled. See glTexImage1D.
GL_TEXTURE_2D	The *params* parameter returns a single Boolean value indicating whether 2-D texture mapping is enabled. See glTexImage2D.

VALUE	MEANING
GL_TEXTURE_COORD_ARRAY	The *params* parameter returns a single Boolean value indicating whether the texture coordinate array is enabled. See **glTexCoordPointer**.
GL_TEXTURE_COORD_ARRAY_SIZE	The *params* parameter returns one value, the number of coordinates per element in the texture coordinate array. See **glTexCoordPointer**.
GL_TEXTURE_COORD_ARRAY_STRIDE	The *params* parameter returns one value, the byte offset between consecutive elements in the texture coordinate array. See **glTexCoordPointer**.
GL_TEXTURE_COORD_ARRAY_TYPE	The *params* parameter params returns one value, the data type of the coordinates in the texture coordinate array. See **glTexCoordPointer**.
GL_TEXTURE_ENV_COLOR	The *params* parameter returns four values: the red, green, blue, and alpha values of the texture environment color. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and 1.0 returns the most negative representable integer value. See glTexEnv.

VALUE	MEANING
GL_TEXTURE_ENV_MODE	The *params* parameter returns one value: a symbolic constant indicating which texture environment function is currently selected. See glTexEnv.
GL_TEXTURE_GEN_Q	The *params* parameter returns a single Boolean value indicating whether automatic generation of the Q texture coordinate is enabled. See **glTexGen**.
GL_TEXTURE_GEN_R	The *params* parameter returns a single Boolean value indicating whether automatic generation of the R texture coordinate is enabled. See **glTexGen**.
GL_TEXTURE_GEN_S	The *params* parameter returns a single Boolean value indicating whether automatic generation of the S texture coordinate is enabled. See **glTexGen**.
GL_TEXTURE_GEN_T	The *params* parameter returns a single Boolean value indicating whether automatic generation of the T texture coordinate is enabled. See **glTexGen**.
GL_TEXTURE_MATRIX	The *params* parameter returns 16 values: the texture matrix on the top of the texture matrix stack. See **glPushMatrix**.

VALUE	MEANING
GL_TEXTURE_STACK_DEPTH	The *params* parameter returns one value: the number of matrices on the texture matrix stack. See **glPushMatrix**.
GL_UNPACK_ALIGNMENT	The *params* parameter returns one value: the byte alignment used for reading pixel data from memory. See glPixelStore.
GL_UNPACK_LSB_FIRST	The *params* parameter returns a single Boolean value indicating whether single-bit pixels being read from memory are read first from the least significant bit of each unsigned byte. See glPixelStore.
GL_UNPACK_ROW_LENGTH	The *params* parameter returns one value: the row length used for reading pixel data from memory. See glPixelStore.
GL_UNPACK_SKIP_PIXELS	The *params* parameter returns one value: the number of pixel locations skipped before the first pixel is read from memory. See glPixelStore.
GL_UNPACK_SKIP_ROWS	The *params* parameter returns one value: the number of rows of pixel locations skipped before the first pixel is read from memory. See glPixelStore.

VALUE	MEANING
GL_UNPACK_SWAP_BYTES	The *params* parameter returns a single Boolean value indicating whether the bytes of 2-byte and 4-byte pixel indexes and components are swapped after being read from memory. See glPixelStore.
GL_VERTEX_ARRAY	The *params* parameter returns a single Boolean value indicating whether the vertex array is enabled. See **glVertexPointer**.
GL_VERTEX_ARRAY_SIZE	The *params* parameter returns one value, the number of coordinates per vertex in the vertex array. See **glVertexPointer**.
GL_VERTEX_ARRAY_STRIDE	The *params* parameter returns one value, the byte offset between consecutive vertexes in the vertex array. See **glVertexPointer**.
GL_VERTEX_ARRAY_TYPE	The *params* parameter returns one value, the data type of each coordinate in the vertex array. See **glVertexPointer**.
GL_VIEWPORT	The *params* parameter returns four values: the *x* and *y* window coordinates of the viewport, followed by its width and height. See **glViewport**.

VALUE	MEANING
GL_ZOOM_X	The *params* parameter returns one value: the *x* pixel zoom factor. See glPixelZoom.
GL_ZOOM_Y	The *params* parameter returns one value: the *y* pixel zoom factor. See glPixelZoom.

p a r a m s Returns the value or values of the specified parameter.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*pname* was not an accepted value.
GL_INVALID_OPERATION	glGetDoublev was called between a call toglBegin and the corresponding call to glEnd.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*pname* was not an accepted value.
GL_INVALID_OPERATION	**glGetDoublev** was called between a call to**glBegin** and the corresponding call to **glEnd**.

Remarks

This function returns values for simple state variables in OpenGL. The *pname* parameter is a symbolic constant indicating the state variable to be returned, and *params* is a pointer to an array of the indicated type in which to place the returned data.

Type conversion is performed if *params* has a different type from the state variable value being requested. If you call **glGetBooleanv**, a floating-point or integer value is converted to GL_FALSE if and only if it is zero. Otherwise, it is converted to GL_TRUE.

If you call **glGetIntegerv**, Boolean values are returned as GL_TRUE or GL_FALSE, and most floating-point values are rounded to the nearest integer value. Floating-point colors and normals, however, are returned with a linear mapping that maps 1.0 to the most positive representable integer value and 1.0 to the most negative representable integer value.

If you call **glGetFloatv** or **glGetDoublev**, Boolean values are returned as GL_TRUE or GL_FALSE, and integer values are converted to floating-point values.

You can query many of the Boolean parameters more easily with **glIsEnabled**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib

DLL	Opengl32.dll

See also

glAccum

glAlphaFunc

glBegin

glBlendFunc

glCa

l
l
L
i
s
t

g
l
C
l
e
a
r
A
c
c
u
m

g
l
C
l
e
a
r
C
o
l
o
r

g
l
C
l
e
a
r
D
e
p
t
h

g
l
C
l
e
a
r

Index

glClearStencil

glClipPlane

glColor

glColorMask

glColorMaterial

glCullFace

glDepthFunc

glDepthMask

gl

D
e
p
t
h
R
a
n
g
e

g
l
D
r
a
w
B
u
f
f
e
r

g
l
E
d
g
e
F
l
a
g

g
l
E
n
d

g
l
F
o
g

g
l
F
r
o
n

t
F
a
c
e

g
l
G
e
t
C
l
i
p
P
l
a
n
e

g
l
G
e
t
E
r
r
o
r

g
l
G
e
t
L
i
g
h
t

g
l
G
e
t
M
a
p

g

lGetMaterial

glGetPixelMap

glGetPolygonStipple

glGetS

t
r
i
n
g

g
l
G
e
t
T
e
x
E
n
v

g
l
G
e
t
T
e
x
G
e
n

g
l
G
e
t
T
e
x
I
m
a
g
e

g
l
G
e
t
T
e
x
L

e
v
e
l
P
a
r
a
m
e
t
e
r

g
l
G
e
t
T
e
x
P
a
r
a
m
e
t
e
r

g
l
H
i
n
t

g
l
I
n
d
e
x

g
l
I
n
d
e

x
M
a
s
k

g
l
I
s
E
n
a
b
l
e
d

g
l
L
i
g
h
t

g
l
L
i
g
h
t
M
o
d
e
l

g
l
L
i
n
e
S
t
i
p
p
l
e

glLineWidth

glListBase

glLogicOp

glMap1

glMap2

glMapG

r
i
d

g
l
M
a
t
r
i
x
M
o
d
e

g
l
N
e
w
L
i
s
t

g
l
N
o
r
m
a
l

g
l
P
i
x
e
l
M
a
p

g
l
P
i
x
e

lStore

glPixelTransfer

glPixelZoom

glPointSize

glPoly

g
o
n
M
o
d
e

g
l
P
o
l
y
g
o
n
S
t
i
p
p
l
e

g
l
P
u
s
h
A
t
t
r
i
b

g
l
P
u
s
h
M
a
t
r
i
x

g
l

P
u
s
h
N
a
m
e

g
l
R
a
s
t
e
r
P
o
s

g
l
R
e
a
d
P
i
x
e
l
s

g
l
S
c
i
s
s
o
r

g
l
S
h
a
d
e
M
o

d
e
l

g
l
S
t
e
n
c
i
l
F
u
n
c

g
l
S
t
e
n
c
i
l
M
a
s
k

g
l
S
t
e
n
c
i
l
O
p

g
l
T
e
x
C
o
o
r

d

glTexEnv

glTexGen

glTexImage1D

glTexImage2D

glViewp

o
r
t

glGetFloatv function

2/22/2020 • 25 minutes to read • Edit Online

The **glGetFloatv** function returns the value or values of a selected parameter.

Syntax

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

Parameters

pname

The parameter value to be returned. The following symbolic constants are accepted.

VALUE	MEANING
GL_ACCUM_ALPHA_BITS	The *params* parameter returns one value: the number of alpha bitplanes in the accumulation buffer.
GL_ACCUM_BLUE_BITS	The *params* parameter returns one value: the number of blue bitplanes in the accumulation buffer.
GL_ACCUM_CLEAR_VALUE	The *params* parameter returns four values: the red, green, blue, and alpha values used to clear the accumulation buffer. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See glClearAccum.

VALUE	MEANING
GL_ACCUM_GREEN_BITS	The *params* parameter returns one value: the number of green bitplanes in the accumulation buffer.
GL_ACCUM_RED_BITS	The *params* parameter returns one value: the number of red bitplanes in the accumulation buffer.
GL_ALPHA_BIAS	The *params* parameter returns one value: the alpha bias factor used during pixel transfers. See **glPixelTransfer**.
GL_ALPHA_BITS	The *params* parameter returns one value: the number of alpha bitplanes in each color buffer.
GL_ALPHA_SCALE	The *params* parameter returns one value: the alpha scale factor used during pixel transfers. See glPixelTransfer.
GL_ALPHA_TEST	The *params* parameter returns a single Boolean value indicating whether alpha testing of fragments is enabled. See **glAlphaFunc**.
GL_ALPHA_TEST_FUNC	The *params* parameter returns one value: the symbolic name of the alpha test function. See **glAlphaFunc**.

VALUE	MEANING
GL_ALPHA_TEST_REF	The *params* parameter returns one value: the reference value for the alpha test. See glAlphaFunc. An integer value, if requested, is linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value.
GL_ATTRIB_STACK_DEPTH	The *params* parameter returns one value: the depth of the attribute stack. If the stack is empty, zero is returned. See glPushAttrib.
GL_AUTO_NORMAL	The *params* parameter returns a single Boolean value indicating whether 2-D map evaluation automatically generates surface normals. See glMap2.
GL_AUX_BUFFERS	The *params* parameter returns one value: the number of auxiliary color buffers.
GL_BLEND	The *params* parameter returns a single Boolean value indicating whether blending is enabled. See glBlendFunc.
GL_BLEND_DST	The *params* parameter returns one value: the symbolic constant identifying the destination blend function. See glBlendFunc.
GL_BLEND_SRC	The *params* parameter returns one value: the symbolic constant identifying the source blend function. See glBlendFunc.

VALUE	MEANING
GL_BLUE_BIAS	The *params* parameter returns one value: the blue bias factor used during pixel transfers. See **glPixelTransfer**.
GL_BLUE_BITS	The *params* parameter returns one value: the number of blue bitplanes in each color buffer.
GL_BLUE_SCALE	The *params* parameter returns one value: the blue scale factor used during pixel transfers. See glPixelTransfer.
GL_CLIENT_ATTRIB_STACK_DEPTH	The *params* parameter returns one value indicating the depth of the attribute stack. The initial value is zero. See **glPushClientAttrib**.
GL_CLIP_PLANE*i*	The *params* parameter returns a single Boolean value indicating whether the specified clipping plane is enabled. See **glClipPlane**.
GL_COLOR_ARRAY	The *params* parameter returns a single Boolean value indicating whether the specified color array is defined. See **glColorPointer**.
GL_COLOR_ARRAY_SIZE	The *params* parameter returns one value, the number of components per color in the color array. See **glColorPointer**.

VALUE	MEANING
GL_COLOR_ARRAY_STRIDE	The *params* parameter returns one value, the byte offset between consecutive colors in the color array. See glColorPointer.
GL_COLOR_ARRAY_TYPE	The *params* parameter returns one value, the data type of each component in the color array. See glColorPointer.
GL_COLOR_CLEAR_VALUE	The *params* parameter returns four values: the red, green, blue, and alpha values used to clear the color buffers. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See glClearColor.
GL_COLOR_LOGIC_OP	The *params* parameter returns a single Boolean value indicating whether a fragment's RGBA color values are merged into the framebuffer using a logical operation. See glLogicOp.
GL_COLOR_MATERIAL	The *params* parameter returns a single Boolean value indicating whether one or more material parameters are tracking the current color. See glColorMaterial.
GL_COLOR_MATERIAL_FACE	The *params* parameter returns one value: a symbolic constant indicating which materials have a parameter that is tracking the current color. See glColorMaterial.

VALUE	MEANING
GL_COLOR_MATERIAL_PARAMETER	The *params* parameter returns one value: a symbolic constant indicating which material parameters are tracking the current color. See glColorMaterial.
GL_COLOR_WRITEMASK	The *params* parameter returns four Boolean values: the red, green, blue, and alpha write enables for the color buffers. See glColorMask.
GL_CULL_FACE	The *params* parameter returns a single Boolean value indicating whether polygon culling is enabled. See glCullFace.
GL_CULL_FACE_MODE	The *params* parameter returns one value: a symbolic constant indicating which polygon faces are to be culled. See glCullFace.
GL_CURRENT_COLOR	The *params* parameter returns four values: the red, green, blue, and alpha values of the current color. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See glColor.
GL_CURRENT_INDEX	The *params* parameter returns one value: the current color index. See glIndex.

VALUE	MEANING
GL_CURRENT_NORMAL	The *params* parameter returns three values: the *x*, *y*, and *z* values of the current normal. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glNormal**.
GL_CURRENT_RASTER_COLOR	The *params* parameter returns four values: the red, green, blue, and alpha values of the current raster position. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glRasterPos**.
GL_CURRENT_RASTER_DISTANCE	The *params* parameter returns one value: the distance from the eye to the current raster position. See **glRasterPos**.
GL_CURRENT_RASTER_INDEX	The *params* parameter returns one value: the color index of the current raster position. See **glRasterPos**.
GL_CURRENT_RASTER_POSITION	The *params* parameter returns four values: the *x*, *y*, *z*, and *w* components of the current raster position. The *x*, *y*, and *z* components are in window coordinates, and *w* is in clip coordinates. See glRasterPos.

VALUE	MEANING
GL_CURRENT_RASTER_POSITION_VALID	The *params* parameter returns a single Boolean value indicating whether the current raster position is valid. See glRasterPos.
GL_CURRENT_RASTER_TEXTURE_COORDS	The *params* parameter returns four values: the *s*, *t*, *r*, and *q* current raster texture coordinates. See glRasterPos and glTexCoord.
GL_CURRENT_TEXTURE_COORDS	The *params* parameter returns four values: the *s*, *t*, *r*, and *q* current texture coordinates. See glTexCoord.
GL_DEPTH_BIAS	The *params* parameter returns one value: the depth bias factor used during pixel transfers. See glPixelTransfer.
GL_DEPTH_BITS	The *params* parameter returns one value: the number of bitplanes in the depth buffer.

VALUE	MEANING
GL_DEPTH_CLEAR_VALUE	The *params* parameter returns one value: the value that is used to clear the depth buffer. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See glClearDepth.
GL_DEPTH_FUNC	The *params* parameter returns one value: the symbolic constant that indicates the depth comparison function. See glDepthFunc.
GL_DEPTH_RANGE	The *params* parameter returns two values: the near and far mapping limits for the depth buffer. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See glDepthRange.
GL_DEPTH_SCALE	The *params* parameter returns one value: the depth scale factor used during pixel transfers. See glPixelTransfer.
GL_DEPTH_TEST	The *params* parameter returns a single Boolean value indicating whether depth testing of fragments is enabled. See glDepthFunc and glDepthRange.
GL_DEPTH_WRITEMASK	The *params* parameter returns a single Boolean value indicating if the depth buffer is enabled for writing. See glDepthMask.
GL_DITHER	The *params* parameter returns a single Boolean value indicating whether dithering of fragment colors and indexes is enabled.

VALUE	MEANING
GL_DOUBLEBUFFER	The *params* parameter returns a single Boolean value indicating whether double buffering is supported.
GL_DRAW_BUFFER	The *params* parameter returns one value: a symbolic constant indicating which buffers are being drawn to. See **glDrawBuffer**.
GL_EDGE_FLAG	The *params* parameter returns a single Boolean value indicating whether the current edge flag is true or false. See glEdgeFlag.
GL_EDGE_FLAG_ARRAY	The *params* parameter returns a single Boolean value indicating whether the edge flag array is enabled. See **glEdgeFlagPointer**.
GL_EDGE_FLAG_ARRAY_STRIDE	The *params* parameter returns one value, the byte offset between consecutive edge flags in the edge flag array. See **glEdgeFlagPointer**.
GL_FOG	The *params* parameter returns a single Boolean value indicating whether fogging is enabled. See **glFog**.

VALUE	MEANING
GL_FOG_COLOR	The *params* parameter returns four values: the red, green, blue, and alpha components of the fog color. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glFog**.
GL_FOG_DENSITY	The *params* parameter returns one value: the fog density parameter. See glFog.
GL_FOG_END	The *params* parameter returns one value: the end factor for the linear fog equation. See **glFog**.
GL_FOG_HINT	The *params* parameter returns one value: a symbolic constant indicating the mode of the fog hint. See **glHint**.
GL_FOG_INDEX	The *params* parameter returns one value: the fog color index. See **glFog**.
GL_FOG_MODE	The *params* parameter returns one value: a symbolic constant indicating which fog equation is selected. See **glFog**.
GL_FOG_START	The *params* parameter returns one value: the start factor for the linear fog equation. See **glFog**.

VALUE	MEANING
GL_FRONT_FACE	The *params* parameter returns one value: a symbolic constant indicating whether clockwise or counterclockwise polygon winding is treated as front-facing. See glFrontFace.
GL_GREEN_BIAS	The *params* parameter returns one value: the green bias factor used during pixel transfers.
GL_GREEN_BITS	The *params* parameter returns one value: the number of green bitplanes in each color buffer.
GL_GREEN_SCALE	The *params* parameter returns one value: the green scale factor used during pixel transfers. See glPixelTransfer.
GL_INDEX_ARRAY	The *params* parameter returns a single Boolean value indicating whether the color index array is enabled. See glIndexPointer.
GL_INDEX_ARRAY_STRIDE	The *params* parameter returns one value, the byte offset between consecutive color indexes in the color index array. See glIndexPointer.
GL_INDEX_ARRAY_TYPE	The *params* parameter returns one value, the data type of indexes in the color index array. The initial value is GL_FLOAT. See glIndexPointer.

VALUE	MEANING
GL_INDEX_BITS	The *params* parameter returns one value: the number of bitplanes in each color-index buffer.
GL_INDEX_CLEAR_VALUE	The *params* parameter returns one value: the color index used to clear the color-index buffers. See glClearIndex.
GL_INDEX_LOGIC_OP	The *params* parameter returns a single Boolean value indicating whether a fragment's index values are merged into the framebuffer using a logical operation. See glLogicOp.
GL_INDEX_MODE	The *params* parameter returns a single Boolean value indicating whether OpenGL is in color-index mode (TRUE) or RGBA mode (FALSE).
GL_INDEX_OFFSET	The *params* parameter returns one value: the offset added to color and stencil indexes during pixel transfers. See glPixelTransfer.
GL_INDEX_SHIFT	The *params* parameter returns one value: the amount that color and stencil indexes are shifted during pixel transfers. See glPixelTransfer.
GL_INDEX_WRITEMASK	The *params* parameter returns one value: a mask indicating which bitplanes of each color-index buffer can be written. See glIndexMask.

VALUE	MEANING
GL_LIGHT*i*	The *params* parameter returns a single Boolean value indicating whether the specified light is enabled. See glLight and glLightModel.
GL_LIGHTING	The *params* parameter returns a single Boolean value indicating whether lighting is enabled. See glLightModel.
GL_LIGHT_MODEL_AMBIENT	The *params* parameter returns four values: the red, green, blue, and alpha components of the ambient intensity of the entire scene. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See glLightModel.
GL_LIGHT_MODEL_LOCAL_VIEWER	The *params* parameter returns a single Boolean value indicating whether specular reflection calculations treat the viewer as being local to the scene. See glLightModel.
GL_LIGHT_MODEL_TWO_SIDE	The *params* parameter returns a single Boolean value indicating whether separate materials are used to compute lighting for front-facing and back-facing polygons. See glLightModel.
GL_LINE_SMOOTH	The *params* parameter returns a single Boolean value indicating whether antialiasing of lines is enabled. See glLineWidth.

VALUE	MEANING
GL_LINE_SMOOTH_HINT	The *params* parameter returns one value: a symbolic constant indicating the mode of the line antialiasing hint. See glHint.
GL_LINE_STIPPLE	The *params* parameter returns a single Boolean value indicating whether stippling of lines is enabled. See glLineStipple.
GL_LINE_STIPPLE_PATTERN	The *params* parameter returns one value: the 16-bit line stipple pattern. See glLineStipple.
GL_LINE_STIPPLE_REPEAT	The *params* parameter returns one value: the line stipple repeat factor. See glLineStipple.
GL_LINE_WIDTH	The *params* parameter returns one value: the line width as specified with glLineWidth.

VALUE	MEANING
GL_LINE_WIDTH_GRANULARITY	The *params* parameter returns one value: the width difference between adjacent supported widths for antialiased lines. See glLineWidth.
GL_LINE_WIDTH_RANGE	The *params* parameter returns two values: the smallest and largest supported widths for antialiased lines. See glLineWidth.
GL_LIST_BASE	The *params* parameter returns one value: the base offset added to all names in arrays presented to glCallLists. See glListBase.
GL_LIST_INDEX	The *params* parameter returns one value: the name of the display list currently under construction. Zero is returned if no display list is currently under construction. See glNewList.
GL_LIST_MODE	The *params* parameter returns one value: a symbolic constant indicating the construction mode of the display list currently being constructed. See glNewList.
GL_LOGIC_OP	The *params* parameter returns a single Boolean value indicating whether fragment indexes are merged into the framebuffer using a logical operation. See glLogicOp.

VALUE	MEANING
GL_LOGIC_OP_MODE	The *params* parameter returns one value: a symbolic constant indicating the selected logic operational mode. See **glLogicOp**.
GL_MAP1_COLOR_4	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates colors. See **glMap1**.
GL_MAP1_GRID_DOMAIN	The *params* parameter returns two values: the endpoints of the 1-D maps grid domain. See glMapGrid.
GL_MAP1_GRID_SEGMENTS	The *params* parameter returns one value: the number of partitions in the 1-D maps grid domain. See glMapGrid.
GL_MAP1_INDEX	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates color indexes. See **glMap1**.
GL_MAP1_NORMAL	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates normals. See **glMap1**.

VALUE	MEANING
GL_MAP1_TEXTURE_COORD_1	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 1-D texture coordinates. See glMap1.
GL_MAP1_TEXTURE_COORD_2	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 2-D texture coordinates. See glMap1.
GL_MAP1_TEXTURE_COORD_3	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 3-D texture coordinates. See glMap1.
GL_MAP1_TEXTURE_COORD_4	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 4-D texture coordinates. See glMap1.
GL_MAP1_VERTEX_3	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 3-D vertex coordinates. See glMap1.

VALUE	MEANING
GL_MAP1_VERTEX_4	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 4-D vertex coordinates. See **glMap1**.
GL_MAP2_COLOR_4	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates colors. See **glMap2**.
GL_MAP2_GRID_DOMAIN	The *params* parameter returns four values: the endpoints of the 2-D maps *i* and *j* grid domains. See glMapGrid.
GL_MAP2_GRID_SEGMENTS	The *params* parameter returns two values: the number of partitions in the 2-D maps *i* and *j* grid domains. See glMapGrid.
GL_MAP2_INDEX	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates color indexes. See **glMap2**.
GL_MAP2_NORMAL	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates normals. See **glMap2**.

VALUE	MEANING
GL_MAP2_TEXTURE_COORD_1	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 1-D texture coordinates. See glMap2.
GL_MAP2_TEXTURE_COORD_2	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 2-D texture coordinates. See glMap2.
GL_MAP2_TEXTURE_COORD_3	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 3-D texture coordinates. See glMap2.
GL_MAP2_TEXTURE_COORD_4	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 4-D texture coordinates. See glMap2.
GL_MAP2_VERTEX_3	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 3-D vertex coordinates. See glMap2.

VALUE	MEANING
GL_MAP2_VERTEX_4	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 4-D vertex coordinates. See glMap2.
GL_MAP_COLOR	The *params* parameter returns a single Boolean value indicating whether colors and color indexes are to be replaced by table lookup during pixel transfers. See glPixelTransfer.
GL_MAP_STENCIL	The *params* parameter returns a single Boolean value indicating whether stencil indexes are to be replaced by table lookup during pixel transfers. See glPixelTransfer.
GL_MATRIX_MODE	The *params* parameter returns one value: a symbolic constant indicating which matrix stack is currently the target of all matrix operations. See glMatrixMode.
GL_MAX_CLIENT_ATTRIB_STACK_DEPTH	The *params* parameter returns one value indicating the maximum supported depth of the client attribute stack. See glPushClientAttrib.
GL_MAX_ATTRIB_STACK_DEPTH	The *params* parameter returns one value: the maximum supported depth of the attribute stack. See glPushAttrib.

VALUE	MEANING
GL_MAX_CLIP_PLANES	The *params* parameter returns one value: the maximum number of application-defined clipping planes. See **glClipPlane**.
GL_MAX_EVAL_ORDER	The *params* parameter returns one value: the maximum equation order supported by 1-D and 2-D evaluators. See **glMap1** and **glMap2**.
GL_MAX_LIGHTS	The *params* parameter returns one value: the maximum number of lights. See glLight.
GL_MAX_LIST_NESTING	The *params* parameter returns one value: the maximum recursion depth allowed during display-list traversal. See **glCallList**.
GL_MAX_MODELVIEW_STACK_DEPTH	The *params* parameter returns one value: the maximum supported depth of the modelview matrix stack. See **glPushMatrix**.
GL_MAX_NAME_STACK_DEPTH	The *params* parameter returns one value: the maximum supported depth of the selection name stack. See **glPushName**.

VALUE	MEANING
GL_MAX_PIXEL_MAP_TABLE	The *params* parameter returns one value: the maximum supported size of a glPixelMap lookup table.
GL_MAX_PROJECTION_STACK_DEPTH	The *params* parameter returns one value: the maximum supported depth of the projection matrix stack. See glPushMatrix.
GL_MAX_TEXTURE_SIZE	The *params* parameter returns one value: the maximum width or height of any texture image (without borders). See glTexImage1D and glTexImage2D.
GL_MAX_TEXTURE_STACK_DEPTH	The *params* parameter returns one value: the maximum supported depth of the texture matrix stack. See glPushMatrix.
GL_MAX_VIEWPORT_DIMS	The *params* parameter returns two values: the maximum supported width and height of the viewport. See glViewport.

VALUE	MEANING
GL_MODELVIEW_MATRIX	The *params* parameter returns 16 values: the modelview matrix on the top of the modelview matrix stack. See glPushMatrix.
GL_MODELVIEW_STACK_DEPTH	The *params* parameter returns one value: the number of matrices on the modelview matrix stack. See glPushMatrix.
GL_NAME_STACK_DEPTH	The *params* parameter returns one value: the number of names on the selection name stack. See glPushName.
GL_NORMAL_ARRAY	The *params* parameter returns a single Boolean value, indicating whether the normal array is enabled. See glNormalPointer.
GL_NORMAL_ARRAY_STRIDE	The *params* parameter returns one value, the byte offset between consecutive normals in the normal array. See glNormalPointer.

VALUE	MEANING
GL_NORMAL_ARRAY_TYPE	The *params* parameter returns one value, the data type of each coordinate in the normal array. See **glNormalPointer**.
GL_NORMALIZE	The *params* parameter returns a single Boolean value indicating whether normals are automatically scaled to unit length after they have been transformed to eye coordinates. See glNormal.
GL_PACK_ALIGNMENT	The *params* parameter returns one value: the byte alignment used for writing pixel data to memory. See glPixelStore.
GL_PACK_LSB_FIRST	The *params* parameter returns a single Boolean value indicating whether single-bit pixels being written to memory are written first to the least significant bit of each unsigned byte. See glPixelStore.
GL_PACK_ROW_LENGTH	The *params* parameter returns one value: the row length used for writing pixel data to memory. See glPixelStore.
GL_PACK_SKIP_PIXELS	The *params* parameter returns one value: the number of pixel locations skipped before the first pixel is written into memory. See glPixelStore.

VALUE	MEANING
GL_PACK_SKIP_ROWS	The *params* parameter returns one value: the number of rows of pixel locations skipped before the first pixel is written into memory. See glPixelStore.
GL_PACK_SWAP_BYTES	The *params* parameter returns a single Boolean value indicating whether the bytes of 2-byte and 4-byte pixel indexes and components are swapped before being written to memory. See glPixelStore.
GL_PERSPECTIVE_CORRECTION_HINT	The *params* parameter returns one value: a symbolic constant indicating the mode of the perspective correction hint. See **glHint**.
GL_PIXEL_MAP_A_TO_A_SIZE	The *params* parameter returns one value: the size of the alpha-to-alpha pixel-translation table. See **glPixelMap**.
GL_PIXEL_MAP_B_TO_B_SIZE	The *params* parameter returns one value: the size of the blue-to-blue pixel-translation table. See **glPixelMap**.

VALUE	MEANING
GL_PIXEL_MAP_G_TO_G_SIZE	The *params* parameter returns one value: the size of the green-to-green pixel-translation table. See **glPixelMap**.
GL_PIXEL_MAP_I_TO_A_SIZE	The *params* parameter returns one value: the size of the index-to-alpha pixel translation table. See **glPixelMap**.
GL_PIXEL_MAP_I_TO_B_SIZE	The *params* parameter returns one value: the size of the index-to-blue pixel translation table. See **glPixelMap**.
GL_PIXEL_MAP_I_TO_G_SIZE	The *params* parameter returns one value: the size of the index-to-green pixel-translation table. See **glPixelMap**.
GL_PIXEL_MAP_I_TO_I_SIZE	The *params* parameter returns one value: the size of the index-to-index pixel-translation table. See **glPixelMap**.

VALUE	MEANING
GL_PIXEL_MAP_I_TO_R_SIZE	The *params* parameter returns one value: the size of the index-to-red pixel-translation table. See glPixelMap.
GL_PIXEL_MAP_R_TO_R_SIZE	The *params* parameter returns one value: the size of the red-to-red pixel-translation table. See glPixelMap.
GL_PIXEL_MAP_S_TO_S_SIZE	The *params* parameter returns one value: the size of the stencil-to-stencil pixel translation table. See glPixelMap.
GL_POINT_SIZE	The *params* parameter returns one value: the point size as specified by glPointSize.
GL_POINT_SIZE_GRANULARITY	The *params* parameter returns one value: the size difference between adjacent supported sizes for antialiased points. See glPointSize.

VALUE	MEANING
GL_POINT_SIZE_RANGE	The *params* parameter returns two values: the smallest and largest supported sizes for antialiased points. See glPointSize.
GL_POINT_SMOOTH	The *params* parameter returns a single Boolean value indicating whether antialiasing of points is enabled. See glPointSize.
GL_POINT_SMOOTH_HINT	The *params* parameter returns one value: a symbolic constant indicating the mode of the point antialiasing hint. See glHint.
GL_POLYGON_MODE	The *params* parameter returns two values: symbolic constants indicating whether front-facing and back-facing polygons are rasterized as points, lines, or filled polygons. See glPolygonMode.
GL_POLYGON_OFFSET_FACTOR	The *params* parameter 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. See glPolygonOffset.

VALUE	MEANING
GL_POLYGON_OFFSET_UNITS	The *params* parameter 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. See glPolygonOffset.
GL_POLYGON_OFFSET_FILL	The *params* parameter returns a single Boolean value indicating whether polygon offset is enabled for polygons in fill mode. See glPolygonOffset.
GL_POLYGON_OFFSET_LINE	The *params* parameter returns a single Boolean value indicating whether polygon offset is enabled for polygons in line mode. See glPolygonOffset.
GL_POLYGON_OFFSET_POINT	The *params* parameter returns a single Boolean value indicating whether polygon offset is enabled for polygons in point mode. See glPolygonOffset.
GL_POLYGON_SMOOTH	The *params* parameter returns a single Boolean value indicating whether antialiasing of polygons is enabled. See glPolygonMode.

VALUE	MEANING
GL_POLYGON_SMOOTH_HINT	The *params* parameter returns one value: a symbolic constant indicating the mode of the polygon antialiasing hint. See glHint.
GL_POLYGON_STIPPLE	The *params* parameter returns a single Boolean value indicating whether stippling of polygons is enabled. See glPolygonStipple.
GL_PROJECTION_MATRIX	The *params* parameter returns 16 values: the projection matrix on the top of the projection matrix stack. See glPushMatrix.
GL_PROJECTION_STACK_DEPTH	The *params* parameter returns one value: the number of matrices on the projection matrix stack. See glPushMatrix.
GL_READ_BUFFER	The *params* parameter returns one value: a symbolic constant indicating which color buffer is selected for reading. See glReadPixels and glAccum.
GL_RED_BIAS	The *params* parameter returns one value: the red bias factor used during pixel transfers. See glPixelTransfer.

VALUE	MEANING
GL_RED_BITS	The *params* parameter returns one value: the number of red bitplanes in each color buffer.
GL_RED_SCALE	The *params* parameter returns one value: the red scale factor used during pixel transfers. See **glPixelTransfer**.
GL_RENDER_MODE	The *params* parameter returns one value: a symbolic constant indicating whether OpenGL is in render, select, or feedback mode. See **glRenderMode**.
GL_RGBA_MODE	The *params* parameter returns a single Boolean value indicating whether OpenGL is in RGBA mode (TRUE) or color-index mode (FALSE). See glColor.
GL_SCISSOR_BOX	The *params* parameter returns four values: the *x* and *y* window coordinates of the scissor box, followed by its width and height. See **glScissor**.
GL_SCISSOR_TEST	The *params* parameter returns a single Boolean value indicating whether scissoring is enabled. See **glScissor**.
GL_SHADE_MODEL	The *params* parameter returns one value: a symbolic constant indicating whether the shading mode is flat or smooth. See **glShadeModel**.
GL_STENCIL_BITS	The *params* parameter returns one value: the number of bitplanes in the stencil buffer.

VALUE	MEANING
GL_STENCIL_CLEAR_VALUE	The *params* parameter returns one value: the index to which the stencil bitplanes are cleared. See glClearStencil.
GL_STENCIL_FAIL	The *params* parameter returns one value: a symbolic constant indicating what action is taken when the stencil test fails. See glStencilOp.
GL_STENCIL_FUNC	The *params* parameter returns one value: a symbolic constant indicating what function is used to compare the stencil reference value with the stencil buffer value. See glStencilFunc.
GL_STENCIL_PASS_DEPTH_FAIL	The *params* parameter returns one value: a symbolic constant indicating what action is taken when the stencil test passes, but the depth test fails. See glStencilOp.
GL_STENCIL_PASS_DEPTH_PASS	The *params* parameter returns one value: a symbolic constant indicating what action is taken when the stencil test passes and the depth test passes. See glStencilOp.
GL_STENCIL_REF	The *params* parameter returns one value: the reference value that is compared with the contents of the stencil buffer. See glStencilFunc.

VALUE	MEANING
GL_STENCIL_TEST	The *params* parameter returns a single Boolean value indicating whether stencil testing of fragments is enabled. See glStencilFunc and glStencilOp.
GL_STENCIL_VALUE_MASK	The *params* parameter returns one value: the mask that is used to mask both the stencil reference value and the stencil buffer value before they are compared. See glStencilFunc.
GL_STENCIL_WRITEMASK	The *params* parameter returns one value: the mask that controls writing of the stencil bitplanes. See glStencilMask.
GL_STEREO	The *params* parameter returns a single Boolean value indicating whether stereo buffers (left and right) are supported.
GL_SUBPIXEL_BITS	The *params* parameter returns one value: an estimate of the number of bits of subpixel resolution that are used to position rasterized geometry in window coordinates.
GL_TEXTURE_1D	The *params* parameter returns a single Boolean value indicating whether 1-D texture mapping is enabled. See glTexImage1D.
GL_TEXTURE_2D	The *params* parameter returns a single Boolean value indicating whether 2-D texture mapping is enabled. See glTexImage2D.

VALUE	MEANING
GL_TEXTURE_COORD_ARRAY	The *params* parameter returns a single Boolean value indicating whether the texture coordinate array is enabled. See **glTexCoordPointer**.
GL_TEXTURE_COORD_ARRAY_SIZE	The *params* parameter returns one value, the number of coordinates per element in the texture coordinate array. See **glTexCoordPointer**.
GL_TEXTURE_COORD_ARRAY_STRIDE	The *params* parameter returns one value, the byte offset between consecutive elements in the texture coordinate array. See **glTexCoordPointer**.
GL_TEXTURE_COORD_ARRAY_TYPE	The *params* parameter params returns one value, the data type of the coordinates in the texture coordinate array. See **glTexCoordPointer**.
GL_TEXTURE_ENV_COLOR	The *params* parameter returns four values: the red, green, blue, and alpha values of the texture environment color. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and 1.0 returns the most negative representable integer value. See glTexEnv.

VALUE	MEANING
GL_TEXTURE_ENV_MODE	The *params* parameter returns one value: a symbolic constant indicating which texture environment function is currently selected. See glTexEnv.
GL_TEXTURE_GEN_Q	The *params* parameter returns a single Boolean value indicating whether automatic generation of the Q texture coordinate is enabled. See **glTexGen**.
GL_TEXTURE_GEN_R	The *params* parameter returns a single Boolean value indicating whether automatic generation of the R texture coordinate is enabled. See **glTexGen**.
GL_TEXTURE_GEN_S	The *params* parameter returns a single Boolean value indicating whether automatic generation of the S texture coordinate is enabled. See **glTexGen**.
GL_TEXTURE_GEN_T	The *params* parameter returns a single Boolean value indicating whether automatic generation of the T texture coordinate is enabled. See **glTexGen**.
GL_TEXTURE_MATRIX	The *params* parameter returns 16 values: the texture matrix on the top of the texture matrix stack. See **glPushMatrix**.

VALUE	MEANING
GL_TEXTURE_STACK_DEPTH	The *params* parameter returns one value: the number of matrices on the texture matrix stack. See **glPushMatrix**.
GL_UNPACK_ALIGNMENT	The *params* parameter returns one value: the byte alignment used for reading pixel data from memory. See glPixelStore.
GL_UNPACK_LSB_FIRST	The *params* parameter returns a single Boolean value indicating whether single-bit pixels being read from memory are read first from the least significant bit of each unsigned byte. See glPixelStore.
GL_UNPACK_ROW_LENGTH	The *params* parameter returns one value: the row length used for reading pixel data from memory. See glPixelStore.
GL_UNPACK_SKIP_PIXELS	The *params* parameter returns one value: the number of pixel locations skipped before the first pixel is read from memory. See glPixelStore.
GL_UNPACK_SKIP_ROWS	The *params* parameter returns one value: the number of rows of pixel locations skipped before the first pixel is read from memory. See glPixelStore.

VALUE	MEANING
GL_UNPACK_SWAP_BYTES	The *params* parameter returns a single Boolean value indicating whether the bytes of 2-byte and 4-byte pixel indexes and components are swapped after being read from memory. See glPixelStore.
GL_VERTEX_ARRAY	The *params* parameter returns a single Boolean value indicating whether the vertex array is enabled. See **glVertexPointer**.
GL_VERTEX_ARRAY_SIZE	The *params* parameter returns one value, the number of coordinates per vertex in the vertex array. See **glVertexPointer**.
GL_VERTEX_ARRAY_STRIDE	The *params* parameter returns one value, the byte offset between consecutive vertexes in the vertex array. See **glVertexPointer**.
GL_VERTEX_ARRAY_TYPE	The *params* parameter returns one value, the data type of each coordinate in the vertex array. See **glVertexPointer**.
GL_VIEWPORT	The *params* parameter returns four values: the *x* and *y* window coordinates of the viewport, followed by its width and height. See **glViewport**.

VALUE	MEANING
GL_ZOOM_X	The *params* parameter returns one value: the *x* pixel zoom factor. See **glPixelZoom**.
GL_ZOOM_Y	The *params* parameter returns one value: the *y* pixel zoom factor. See **glPixelZoom**.

p a r a m s Returns the value or values of the specified parameter.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*pname* was not an accepted value.
GL_INVALID_OPERATION	**glGetFloatv** was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*pname* was not an accepted value.
GL_INVALID_OPERATION	**glGetFloatv** was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

This function returns values for simple state variables in OpenGL. The *pname* parameter is a symbolic constant indicating the state variable to be returned, and *params* is a pointer to an array of the indicated type in which to place the returned data.

Type conversion is performed if *params* has a different type from the state variable value being requested. If you call **glGetBooleanv**, a floating-point or integer value is converted to GL_FALSE if and only if it is zero. Otherwise, it is converted to GL_TRUE.

If you call **glGetIntegerv**, Boolean values are returned as GL_TRUE or GL_FALSE, and most floating-point values are rounded to the nearest integer value. Floating-point colors and normals, however, are returned with a linear mapping that maps 1.0 to the most positive representable integer value and 1.0 to the most negative representable integer value.

If you call **glGetFloatv** or **glGetDoublev**, Boolean values are returned as GL_TRUE or GL_FALSE, and integer values are converted to floating-point values.

You can query many of the Boolean parameters more easily with **glIsEnabled**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib

DLL	Opengl32.dll

See also

glAccum

glAlphaFunc

glBegin

glBlendFunc

glCa

l
l
L
i
s
t

g
l
C
l
e
a
r
A
c
c
u
m

g
l
C
l
e
a
r
C
o
l
o
r

g
l
C
l
e
a
r
D
e
p
t
h

g
l
C
l
e
a
r

Index

glClearStencil

glClipPlane

glColor

glColorMask

glColorMaterial

glCullFace

glDepthFunc

glDepthMask

gl

D
e
p
t
h
R
a
n
g
e

g
l
D
r
a
w
B
u
f
f
e
r

g
l
E
d
g
e
F
l
a
g

g
l
E
n
d

g
l
F
o
g

g
l
F
r
o
n

t
F
a
c
e

g
l
G
e
t
C
l
i
p
P
l
a
n
e

g
l
G
e
t
E
r
r
o
r

g
l
G
e
t
L
i
g
h
t

g
l
G
e
t
M
a
p

g

glGetMaterialf

glGetPixelMap

glGetPolygonStipple

glGetS

t
r
i
n
g

g
l
G
e
t
T
e
x
E
n
v

g
l
G
e
t
T
e
x
G
e
n

g
l
G
e
t
T
e
x
I
m
a
g
e

g
l
G
e
t
T
e
x
L

e
v
e
l
P
a
r
a
m
e
t
e
r

g
l
G
e
t
T
e
x
P
a
r
a
m
e
t
e
r

g
l
H
i
n
t

g
l
I
n
d
e
x

g
l
I
n
d
e

x
M
a
s
k

g
l
l
s
E
n
a
b
l
e
d

g
l
L
i
g
h
t

g
l
L
i
g
h
t
M
o
d
e
l

g
l
L
i
n
e
S
t
i
p
p
l
e

glLineWidth

glListBase

glLogicOp

glMap1

glMap2

glMapG

r
i
d

g
l
M
a
t
r
i
x
M
o
d
e

g
l
N
e
w
L
i
s
t

g
l
N
o
r
m
a
l

g
l
P
i
x
e
l
M
a
p

g
l
P
i
x
e

lStore

glPixelTransfer

glPixelZoom

glPointSize

glPolly

g
o
n
M
o
d
e

g
l
P
o
l
y
g
o
n
S
t
i
p
p
l
e

g
l
P
u
s
h
A
t
t
r
i
b

g
l
P
u
s
h
M
a
t
r
i
x

g
l

PushName

glRasterPos

glReadPixels

glScissor

glShadeMo

d
e
l

g
l
S
t
e
n
c
i
l
F
u
n
c

g
l
S
t
e
n
c
i
l
M
a
s
k

g
l
S
t
e
n
c
i
l
O
p

g
l
T
e
x
C
o
o
r

d

g
l
T
e
x
E
n
v

g
l
T
e
x
G
e
n

g
l
T
e
x
I
m
a
g
e
1
D

g
l
T
e
x
I
m
a
g
e
2
D

g
l
V
i
e
w
p

o
r
t

glGetIntegerv function

2/22/2020 • 25 minutes to read • Edit Online

The **glGetIntegerv** function returns the value or values of a selected parameter.

Syntax

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

Parameters

pname

The parameter value to be returned. The following symbolic constants are accepted.

VALUE	MEANING
GL_ACCUM_ALPHA_BITS	The *params* parameter returns one value: the number of alpha bitplanes in the accumulation buffer.
GL_ACCUM_BLUE_BITS	The *params* parameter returns one value: the number of blue bitplanes in the accumulation buffer.
GL_ACCUM_CLEAR_VALUE	The *params* parameter returns four values: the red, green, blue, and alpha values used to clear the accumulation buffer. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glClearAccum**.

VALUE	MEANING
GL_ACCUM_GREEN_BITS	The *params* parameter returns one value: the number of green bitplanes in the accumulation buffer.
GL_ACCUM_RED_BITS	The *params* parameter returns one value: the number of red bitplanes in the accumulation buffer.
GL_ALPHA_BIAS	The *params* parameter returns one value: the alpha bias factor used during pixel transfers. See **glPixelTransfer**.
GL_ALPHA_BITS	The *params* parameter returns one value: the number of alpha bitplanes in each color buffer.
GL_ALPHA_SCALE	The *params* parameter returns one value: the alpha scale factor used during pixel transfers. See glPixelTransfer.
GL_ALPHA_TEST	The *params* parameter returns a single Boolean value indicating whether alpha testing of fragments is enabled. See **glAlphaFunc**.
GL_ALPHA_TEST_FUNC	The *params* parameter returns one value: the symbolic name of the alpha test function. See **glAlphaFunc**.

VALUE	MEANING
GL_ALPHA_TEST_REF	The *params* parameter returns one value: the reference value for the alpha test. See glAlphaFunc. An integer value, if requested, is linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value.
GL_ATTRIB_STACK_DEPTH	The *params* parameter returns one value: the depth of the attribute stack. If the stack is empty, zero is returned. See glPushAttrib.
GL_AUTO_NORMAL	The *params* parameter returns a single Boolean value indicating whether 2-D map evaluation automatically generates surface normals. See glMap2.
GL_AUX_BUFFERS	The *params* parameter returns one value: the number of auxiliary color buffers.
GL_BLEND	The *params* parameter returns a single Boolean value indicating whether blending is enabled. See glBlendFunc.
GL_BLEND_DST	The *params* parameter returns one value: the symbolic constant identifying the destination blend function. See glBlendFunc.
GL_BLEND_SRC	The *params* parameter returns one value: the symbolic constant identifying the source blend function. See glBlendFunc.

VALUE	MEANING
GL_BLUE_BIAS	The *params* parameter returns one value: the blue bias factor used during pixel transfers. See **glPixelTransfer**.
GL_BLUE_BITS	The *params* parameter returns one value: the number of blue bitplanes in each color buffer.
GL_BLUE_SCALE	The *params* parameter returns one value: the blue scale factor used during pixel transfers. See glPixelTransfer.
GL_CLIENT_ATTRIB_STACK_DEPTH	The *params* parameter returns one value indicating the depth of the attribute stack. The initial value is zero. See **glPushClientAttrib**.
GL_CLIP_PLANE*i*	The *params* parameter returns a single Boolean value indicating whether the specified clipping plane is enabled. See **glClipPlane**.
GL_COLOR_ARRAY	The *params* parameter returns a single Boolean value indicating whether the specified color array is defined. See **glColorPointer**.
GL_COLOR_ARRAY_SIZE	The *params* parameter returns one value, the number of components per color in the color array. See **glColorPointer**.

VALUE	MEANING
GL_COLOR_ARRAY_STRIDE	The *params* parameter returns one value, the byte offset between consecutive colors in the color array. See glColorPointer.
GL_COLOR_ARRAY_TYPE	The *params* parameter returns one value, the data type of each component in the color array. See glColorPointer.
GL_COLOR_CLEAR_VALUE	The *params* parameter returns four values: the red, green, blue, and alpha values used to clear the color buffers. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See glClearColor.
GL_COLOR_LOGIC_OP	The *params* parameter returns a single Boolean value indicating whether a fragment's RGBA color values are merged into the framebuffer using a logical operation. See glLogicOp.
GL_COLOR_MATERIAL	The *params* parameter returns a single Boolean value indicating whether one or more material parameters are tracking the current color. See glColorMaterial.
GL_COLOR_MATERIAL_FACE	The *params* parameter returns one value: a symbolic constant indicating which materials have a parameter that is tracking the current color. See glColorMaterial.

VALUE	MEANING
GL_COLOR_MATERIAL_PARAMETER	The *params* parameter returns one value: a symbolic constant indicating which material parameters are tracking the current color. See glColorMaterial.
GL_COLOR_WRITEMASK	The *params* parameter returns four Boolean values: the red, green, blue, and alpha write enables for the color buffers. See glColorMask.
GL_CULL_FACE	The *params* parameter returns a single Boolean value indicating whether polygon culling is enabled. See glCullFace.
GL_CULL_FACE_MODE	The *params* parameter returns one value: a symbolic constant indicating which polygon faces are to be culled. See glCullFace.
GL_CURRENT_COLOR	The *params* parameter returns four values: the red, green, blue, and alpha values of the current color. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See glColor.
GL_CURRENT_INDEX	The *params* parameter returns one value: the current color index. See glIndex.

VALUE	MEANING
GL_CURRENT_NORMAL	The *params* parameter returns three values: the *x*, *y*, and *z* values of the current normal. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glNormal**.
GL_CURRENT_RASTER_COLOR	The *params* parameter returns four values: the red, green, blue, and alpha values of the current raster position. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glRasterPos**.
GL_CURRENT_RASTER_DISTANCE	The *params* parameter returns one value: the distance from the eye to the current raster position. See **glRasterPos**.
GL_CURRENT_RASTER_INDEX	The *params* parameter returns one value: the color index of the current raster position. See **glRasterPos**.
GL_CURRENT_RASTER_POSITION	The *params* parameter returns four values: the *x*, *y*, *z*, and *w* components of the current raster position. The *x*, *y*, and *z* components are in window coordinates, and *w* is in clip coordinates. See glRasterPos.

VALUE	MEANING
GL_CURRENT_RASTER_POSITION_VALID	The *params* parameter returns a single Boolean value indicating whether the current raster position is valid. See glRasterPos.
GL_CURRENT_RASTER_TEXTURE_COORDS	The *params* parameter returns four values: the *s*, *t*, *r*, and *q* current raster texture coordinates. See glRasterPos and glTexCoord.
GL_CURRENT_TEXTURE_COORDS	The *params* parameter returns four values: the *s*, *t*, *r*, and *q* current texture coordinates. See glTexCoord.
GL_DEPTH_BIAS	The *params* parameter returns one value: the depth bias factor used during pixel transfers. See glPixelTransfer.
GL_DEPTH_BITS	The *params* parameter returns one value: the number of bitplanes in the depth buffer.

VALUE	MEANING
GL_DEPTH_CLEAR_VALUE	The *params* parameter returns one value: the value that is used to clear the depth buffer. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See glClearDepth.
GL_DEPTH_FUNC	The *params* parameter returns one value: the symbolic constant that indicates the depth comparison function. See glDepthFunc.
GL_DEPTH_RANGE	The *params* parameter returns two values: the near and far mapping limits for the depth buffer. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See glDepthRange.
GL_DEPTH_SCALE	The *params* parameter returns one value: the depth scale factor used during pixel transfers. See glPixelTransfer.
GL_DEPTH_TEST	The *params* parameter returns a single Boolean value indicating whether depth testing of fragments is enabled. See glDepthFunc and glDepthRange.
GL_DEPTH_WRITEMASK	The *params* parameter returns a single Boolean value indicating if the depth buffer is enabled for writing. See glDepthMask.
GL_DITHER	The *params* parameter returns a single Boolean value indicating whether dithering of fragment colors and indexes is enabled.

VALUE	MEANING
GL_DOUBLEBUFFER	The *params* parameter returns a single Boolean value indicating whether double buffering is supported.
GL_DRAW_BUFFER	The *params* parameter returns one value: a symbolic constant indicating which buffers are being drawn to. See **glDrawBuffer**.
GL_EDGE_FLAG	The *params* parameter returns a single Boolean value indicating whether the current edge flag is true or false. See glEdgeFlag.
GL_EDGE_FLAG_ARRAY	The *params* parameter returns a single Boolean value indicating whether the edge flag array is enabled. See **glEdgeFlagPointer**.
GL_EDGE_FLAG_ARRAY_STRIDE	The *params* parameter returns one value, the byte offset between consecutive edge flags in the edge flag array. See **glEdgeFlagPointer**.
GL_FOG	The *params* parameter returns a single Boolean value indicating whether fogging is enabled. See **glFog**.

VALUE	MEANING
GL_FOG_COLOR	The *params* parameter returns four values: the red, green, blue, and alpha components of the fog color. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glFog**.
GL_FOG_DENSITY	The *params* parameter returns one value: the fog density parameter. See glFog.
GL_FOG_END	The *params* parameter returns one value: the end factor for the linear fog equation. See **glFog**.
GL_FOG_HINT	The *params* parameter returns one value: a symbolic constant indicating the mode of the fog hint. See **glHint**.
GL_FOG_INDEX	The *params* parameter returns one value: the fog color index. See **glFog**.
GL_FOG_MODE	The *params* parameter returns one value: a symbolic constant indicating which fog equation is selected. See **glFog**.
GL_FOG_START	The *params* parameter returns one value: the start factor for the linear fog equation. See **glFog**.

VALUE	MEANING
GL_FRONT_FACE	The *params* parameter returns one value: a symbolic constant indicating whether clockwise or counterclockwise polygon winding is treated as front-facing. See **glFrontFace**.
GL_GREEN_BIAS	The *params* parameter returns one value: the green bias factor used during pixel transfers.
GL_GREEN_BITS	The *params* parameter returns one value: the number of green bitplanes in each color buffer.
GL_GREEN_SCALE	The *params* parameter returns one value: the green scale factor used during pixel transfers. See **glPixelTransfer**.
GL_INDEX_ARRAY	The *params* parameter returns a single Boolean value indicating whether the color index array is enabled. See **glIndexPointer**.
GL_INDEX_ARRAY_STRIDE	The *params* parameter returns one value, the byte offset between consecutive color indexes in the color index array. See **glIndexPointer**.
GL_INDEX_ARRAY_TYPE	The *params* parameter returns one value, the data type of indexes in the color index array. The initial value is GL_FLOAT. See **glIndexPointer**.

VALUE	MEANING
GL_INDEX_BITS	The *params* parameter returns one value: the number of bitplanes in each color-index buffer.
GL_INDEX_CLEAR_VALUE	The *params* parameter returns one value: the color index used to clear the color-index buffers. See **glClearIndex**.
GL_INDEX_LOGIC_OP	The *params* parameter returns a single Boolean value indicating whether a fragment's index values are merged into the framebuffer using a logical operation. See **glLogicOp**.
GL_INDEX_MODE	The *params* parameter returns a single Boolean value indicating whether OpenGL is in color-index mode (TRUE) or RGBA mode (FALSE).
GL_INDEX_OFFSET	The *params* parameter returns one value: the offset added to color and stencil indexes during pixel transfers. See **glPixelTransfer**.
GL_INDEX_SHIFT	The *params* parameter returns one value: the amount that color and stencil indexes are shifted during pixel transfers. See **glPixelTransfer**.
GL_INDEX_WRITEMASK	The *params* parameter returns one value: a mask indicating which bitplanes of each color-index buffer can be written. See **glIndexMask**.

VALUE	MEANING
GL_LIGHT*i*	The *params* parameter returns a single Boolean value indicating whether the specified light is enabled. See **glLight** and **glLightModel**.
GL_LIGHTING	The *params* parameter returns a single Boolean value indicating whether lighting is enabled. See **glLightModel**.
GL_LIGHT_MODEL_AMBIENT	The *params* parameter returns four values: the red, green, blue, and alpha components of the ambient intensity of the entire scene. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glLightModel**.
GL_LIGHT_MODEL_LOCAL_VIEWER	The *params* parameter returns a single Boolean value indicating whether specular reflection calculations treat the viewer as being local to the scene. See **glLightModel**.
GL_LIGHT_MODEL_TWO_SIDE	The *params* parameter returns a single Boolean value indicating whether separate materials are used to compute lighting for front-facing and back-facing polygons. See **glLightModel**.
GL_LINE_SMOOTH	The *params* parameter returns a single Boolean value indicating whether antialiasing of lines is enabled. See **glLineWidth**.

VALUE	MEANING
GL_LINE_SMOOTH_HINT	The *params* parameter returns one value: a symbolic constant indicating the mode of the line antialiasing hint. See glHint.
GL_LINE_STIPPLE	The *params* parameter returns a single Boolean value indicating whether stippling of lines is enabled. See glLineStipple.
GL_LINE_STIPPLE_PATTERN	The *params* parameter returns one value: the 16-bit line stipple pattern. See glLineStipple.
GL_LINE_STIPPLE_REPEAT	The *params* parameter returns one value: the line stipple repeat factor. See glLineStipple.
GL_LINE_WIDTH	The *params* parameter returns one value: the line width as specified with glLineWidth.

VALUE	MEANING
GL_LINE_WIDTH_GRANULARITY	The *params* parameter returns one value: the width difference between adjacent supported widths for antialiased lines. See glLineWidth.
GL_LINE_WIDTH_RANGE	The *params* parameter returns two values: the smallest and largest supported widths for antialiased lines. See glLineWidth.
GL_LIST_BASE	The *params* parameter returns one value: the base offset added to all names in arrays presented to glCallLists. See glListBase.
GL_LIST_INDEX	The *params* parameter returns one value: the name of the display list currently under construction. Zero is returned if no display list is currently under construction. See glNewList.
GL_LIST_MODE	The *params* parameter returns one value: a symbolic constant indicating the construction mode of the display list currently being constructed. See glNewList.
GL_LOGIC_OP	The *params* parameter returns a single Boolean value indicating whether fragment indexes are merged into the framebuffer using a logical operation. See glLogicOp.

VALUE	MEANING
GL_LOGIC_OP_MODE	The *params* parameter returns one value: a symbolic constant indicating the selected logic operational mode. See **glLogicOp**.
GL_MAP1_COLOR_4	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates colors. See **glMap1**.
GL_MAP1_GRID_DOMAIN	The *params* parameter returns two values: the endpoints of the 1-D maps grid domain. See glMapGrid.
GL_MAP1_GRID_SEGMENTS	The *params* parameter returns one value: the number of partitions in the 1-D maps grid domain. See glMapGrid.
GL_MAP1_INDEX	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates color indexes. See **glMap1**.
GL_MAP1_NORMAL	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates normals. See **glMap1**.

VALUE	MEANING
GL_MAP1_TEXTURE_COORD_1	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 1-D texture coordinates. See **glMap1**.
GL_MAP1_TEXTURE_COORD_2	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 2-D texture coordinates. See **glMap1**.
GL_MAP1_TEXTURE_COORD_3	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 3-D texture coordinates. See **glMap1**.
GL_MAP1_TEXTURE_COORD_4	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 4-D texture coordinates. See **glMap1**.
GL_MAP1_VERTEX_3	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 3-D vertex coordinates. See **glMap1**.

VALUE	MEANING
GL_MAP1_VERTEX_4	The *params* parameter returns a single Boolean value indicating whether 1-D evaluation generates 4-D vertex coordinates. See **glMap1**.
GL_MAP2_COLOR_4	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates colors. See **glMap2**.
GL_MAP2_GRID_DOMAIN	The *params* parameter returns four values: the endpoints of the 2-D maps *i* and *j* grid domains. See glMapGrid.
GL_MAP2_GRID_SEGMENTS	The *params* parameter returns two values: the number of partitions in the 2-D maps *i* and *j* grid domains. See glMapGrid.
GL_MAP2_INDEX	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates color indexes. See **glMap2**.
GL_MAP2_NORMAL	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates normals. See **glMap2**.

VALUE	MEANING
GL_MAP2_TEXTURE_COORD_1	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 1-D texture coordinates. See glMap2.
GL_MAP2_TEXTURE_COORD_2	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 2-D texture coordinates. See glMap2.
GL_MAP2_TEXTURE_COORD_3	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 3-D texture coordinates. See glMap2.
GL_MAP2_TEXTURE_COORD_4	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 4-D texture coordinates. See glMap2.
GL_MAP2_VERTEX_3	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 3-D vertex coordinates. See glMap2.

VALUE	MEANING
GL_MAP2_VERTEX_4	The *params* parameter returns a single Boolean value indicating whether 2-D evaluation generates 4-D vertex coordinates. See glMap2.
GL_MAP_COLOR	The *params* parameter returns a single Boolean value indicating whether colors and color indexes are to be replaced by table lookup during pixel transfers. See glPixelTransfer.
GL_MAP_STENCIL	The *params* parameter returns a single Boolean value indicating whether stencil indexes are to be replaced by table lookup during pixel transfers. See glPixelTransfer.
GL_MATRIX_MODE	The *params* parameter returns one value: a symbolic constant indicating which matrix stack is currently the target of all matrix operations. See glMatrixMode.
GL_MAX_CLIENT_ATTRIB_STACK_DEPTH	The *params* parameter returns one value indicating the maximum supported depth of the client attribute stack. See glPushClientAttrib.
GL_MAX_ATTRIB_STACK_DEPTH	The *params* parameter returns one value: the maximum supported depth of the attribute stack. See glPushAttrib.

VALUE	MEANING
GL_MAX_CLIP_PLANES	The *params* parameter returns one value: the maximum number of application-defined clipping planes. See **glClipPlane**.
GL_MAX_EVAL_ORDER	The *params* parameter returns one value: the maximum equation order supported by 1-D and 2-D evaluators. See **glMap1** and **glMap2**.
GL_MAX_LIGHTS	The *params* parameter returns one value: the maximum number of lights. See glLight.
GL_MAX_LIST_NESTING	The *params* parameter returns one value: the maximum recursion depth allowed during display-list traversal. See **glCallList**.
GL_MAX_MODELVIEW_STACK_DEPTH	The *params* parameter returns one value: the maximum supported depth of the modelview matrix stack. See **glPushMatrix**.
GL_MAX_NAME_STACK_DEPTH	The *params* parameter returns one value: the maximum supported depth of the selection name stack. See **glPushName**.

VALUE	MEANING
GL_MAX_PIXEL_MAP_TABLE	The *params* parameter returns one value: the maximum supported size of a glPixelMap lookup table.
GL_MAX_PROJECTION_STACK_DEPTH	The *params* parameter returns one value: the maximum supported depth of the projection matrix stack. See glPushMatrix.
GL_MAX_TEXTURE_SIZE	The *params* parameter returns one value: the maximum width or height of any texture image (without borders). See glTexImage1D and glTexImage2D.
GL_MAX_TEXTURE_STACK_DEPTH	The *params* parameter returns one value: the maximum supported depth of the texture matrix stack. See glPushMatrix.
GL_MAX_VIEWPORT_DIMS	The *params* parameter returns two values: the maximum supported width and height of the viewport. See glViewport.

VALUE	MEANING
GL_MODELVIEW_MATRIX	The *params* parameter returns 16 values: the modelview matrix on the top of the modelview matrix stack. See glPushMatrix.
GL_MODELVIEW_STACK_DEPTH	The *params* parameter returns one value: the number of matrices on the modelview matrix stack. See glPushMatrix.
GL_NAME_STACK_DEPTH	The *params* parameter returns one value: the number of names on the selection name stack. See glPushName.
GL_NORMAL_ARRAY	The *params* parameter returns a single Boolean value, indicating whether the normal array is enabled. See glNormalPointer.
GL_NORMAL_ARRAY_STRIDE	The *params* parameter returns one value, the byte offset between consecutive normals in the normal array. See glNormalPointer.

VALUE	MEANING
GL_NORMAL_ARRAY_TYPE	The *params* parameter returns one value, the data type of each coordinate in the normal array. See **glNormalPointer**.
GL_NORMALIZE	The *params* parameter returns a single Boolean value indicating whether normals are automatically scaled to unit length after they have been transformed to eye coordinates. See glNormal.
GL_PACK_ALIGNMENT	The *params* parameter returns one value: the byte alignment used for writing pixel data to memory. See glPixelStore.
GL_PACK_LSB_FIRST	The *params* parameter returns a single Boolean value indicating whether single-bit pixels being written to memory are written first to the least significant bit of each unsigned byte. See glPixelStore.
GL_PACK_ROW_LENGTH	The *params* parameter returns one value: the row length used for writing pixel data to memory. See glPixelStore.
GL_PACK_SKIP_PIXELS	The *params* parameter returns one value: the number of pixel locations skipped before the first pixel is written into memory. See glPixelStore.

VALUE	MEANING
GL_PACK_SKIP_ROWS	The *params* parameter returns one value: the number of rows of pixel locations skipped before the first pixel is written into memory. See glPixelStore.
GL_PACK_SWAP_BYTES	The *params* parameter returns a single Boolean value indicating whether the bytes of 2-byte and 4-byte pixel indexes and components are swapped before being written to memory. See glPixelStore.
GL_PERSPECTIVE_CORRECTION_HINT	The *params* parameter returns one value: a symbolic constant indicating the mode of the perspective correction hint. See **glHint**.
GL_PIXEL_MAP_A_TO_A_SIZE	The *params* parameter returns one value: the size of the alpha-to-alpha pixel-translation table. See **glPixelMap**.
GL_PIXEL_MAP_B_TO_B_SIZE	The *params* parameter returns one value: the size of the blue-to-blue pixel-translation table. See **glPixelMap**.

VALUE	MEANING
GL_PIXEL_MAP_G_TO_G_SIZE	The *params* parameter returns one value: the size of the green-to-green pixel-translation table. See **glPixelMap**.
GL_PIXEL_MAP_I_TO_A_SIZE	The *params* parameter returns one value: the size of the index-to-alpha pixel translation table. See **glPixelMap**.
GL_PIXEL_MAP_I_TO_B_SIZE	The *params* parameter returns one value: the size of the index-to-blue pixel translation table. See **glPixelMap**.
GL_PIXEL_MAP_I_TO_G_SIZE	The *params* parameter returns one value: the size of the index-to-green pixel-translation table. See **glPixelMap**.
GL_PIXEL_MAP_I_TO_I_SIZE	The *params* parameter returns one value: the size of the index-to-index pixel-translation table. See **glPixelMap**.

VALUE	MEANING
GL_PIXEL_MAP_I_TO_R_SIZE	The *params* parameter returns one value: the size of the index-to-red pixel-translation table. See glPixelMap.
GL_PIXEL_MAP_R_TO_R_SIZE	The *params* parameter returns one value: the size of the red-to-red pixel-translation table. See glPixelMap.
GL_PIXEL_MAP_S_TO_S_SIZE	The *params* parameter returns one value: the size of the stencil-to-stencil pixel translation table. See glPixelMap.
GL_POINT_SIZE	The *params* parameter returns one value: the point size as specified by glPointSize.
GL_POINT_SIZE_GRANULARITY	The *params* parameter returns one value: the size difference between adjacent supported sizes for antialiased points. See glPointSize.

VALUE	MEANING
GL_POINT_SIZE_RANGE	The *params* parameter returns two values: the smallest and largest supported sizes for antialiased points. See glPointSize.
GL_POINT_SMOOTH	The *params* parameter returns a single Boolean value indicating whether antialiasing of points is enabled. See glPointSize.
GL_POINT_SMOOTH_HINT	The *params* parameter returns one value: a symbolic constant indicating the mode of the point antialiasing hint. See glHint.
GL_POLYGON_MODE	The *params* parameter returns two values: symbolic constants indicating whether front-facing and back-facing polygons are rasterized as points, lines, or filled polygons. See glPolygonMode.
GL_POLYGON_OFFSET_FACTOR	The *params* parameter 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. See glPolygonOffset.

VALUE	MEANING
GL_POLYGON_OFFSET_UNITS	The *params* parameter 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. See glPolygonOffset.
GL_POLYGON_OFFSET_FILL	The *params* parameter returns a single Boolean value indicating whether polygon offset is enabled for polygons in fill mode. See glPolygonOffset.
GL_POLYGON_OFFSET_LINE	The *params* parameter returns a single Boolean value indicating whether polygon offset is enabled for polygons in line mode. See glPolygonOffset.
GL_POLYGON_OFFSET_POINT	The *params* parameter returns a single Boolean value indicating whether polygon offset is enabled for polygons in point mode. See glPolygonOffset.
GL_POLYGON_SMOOTH	The *params* parameter returns a single Boolean value indicating whether antialiasing of polygons is enabled. See glPolygonMode.

VALUE	MEANING
GL_POLYGON_SMOOTH_HINT	The *params* parameter returns one value: a symbolic constant indicating the mode of the polygon antialiasing hint. See glHint.
GL_POLYGON_STIPPLE	The *params* parameter returns a single Boolean value indicating whether stippling of polygons is enabled. See glPolygonStipple.
GL_PROJECTION_MATRIX	The *params* parameter returns 16 values: the projection matrix on the top of the projection matrix stack. See glPushMatrix.
GL_PROJECTION_STACK_DEPTH	The *params* parameter returns one value: the number of matrices on the projection matrix stack. See glPushMatrix.
GL_READ_BUFFER	The *params* parameter returns one value: a symbolic constant indicating which color buffer is selected for reading. See glReadPixels and glAccum.
GL_RED_BIAS	The *params* parameter returns one value: the red bias factor used during pixel transfers. See glPixelTransfer.

VALUE	MEANING
GL_RED_BITS	The *params* parameter returns one value: the number of red bitplanes in each color buffer.
GL_RED_SCALE	The *params* parameter returns one value: the red scale factor used during pixel transfers. See **glPixelTransfer**.
GL_RENDER_MODE	The *params* parameter returns one value: a symbolic constant indicating whether OpenGL is in render, select, or feedback mode. See **glRenderMode**.
GL_RGBA_MODE	The *params* parameter returns a single Boolean value indicating whether OpenGL is in RGBA mode (TRUE) or color-index mode (FALSE). See glColor.
GL_SCISSOR_BOX	The *params* parameter returns four values: the *x* and *y* window coordinates of the scissor box, followed by its width and height. See **glScissor**.
GL_SCISSOR_TEST	The *params* parameter returns a single Boolean value indicating whether scissoring is enabled. See **glScissor**.
GL_SHADE_MODEL	The *params* parameter returns one value: a symbolic constant indicating whether the shading mode is flat or smooth. See **glShadeModel**.
GL_STENCIL_BITS	The *params* parameter returns one value: the number of bitplanes in the stencil buffer.

VALUE	MEANING
GL_STENCIL_CLEAR_VALUE	The *params* parameter returns one value: the index to which the stencil bitplanes are cleared. See glClearStencil.
GL_STENCIL_FAIL	The *params* parameter returns one value: a symbolic constant indicating what action is taken when the stencil test fails. See glStencilOp.
GL_STENCIL_FUNC	The *params* parameter returns one value: a symbolic constant indicating what function is used to compare the stencil reference value with the stencil buffer value. See glStencilFunc.
GL_STENCIL_PASS_DEPTH_FAIL	The *params* parameter returns one value: a symbolic constant indicating what action is taken when the stencil test passes, but the depth test fails. See glStencilOp.
GL_STENCIL_PASS_DEPTH_PASS	The *params* parameter returns one value: a symbolic constant indicating what action is taken when the stencil test passes and the depth test passes. See glStencilOp.
GL_STENCIL_REF	The *params* parameter returns one value: the reference value that is compared with the contents of the stencil buffer. See glStencilFunc.

VALUE	MEANING
GL_STENCIL_TEST	The *params* parameter returns a single Boolean value indicating whether stencil testing of fragments is enabled. See glStencilFunc and glStencilOp.
GL_STENCIL_VALUE_MASK	The *params* parameter returns one value: the mask that is used to mask both the stencil reference value and the stencil buffer value before they are compared. See glStencilFunc.
GL_STENCIL_WRITEMASK	The *params* parameter returns one value: the mask that controls writing of the stencil bitplanes. See glStencilMask.
GL_STEREO	The *params* parameter returns a single Boolean value indicating whether stereo buffers (left and right) are supported.
GL_SUBPIXEL_BITS	The *params* parameter returns one value: an estimate of the number of bits of subpixel resolution that are used to position rasterized geometry in window coordinates.
GL_TEXTURE_1D	The *params* parameter returns a single Boolean value indicating whether 1-D texture mapping is enabled. See glTexImage1D.
GL_TEXTURE_2D	The *params* parameter returns a single Boolean value indicating whether 2-D texture mapping is enabled. See glTexImage2D.

VALUE	MEANING
GL_TEXTURE_COORD_ARRAY	The *params* parameter returns a single Boolean value indicating whether the texture coordinate array is enabled. See **glTexCoordPointer**.
GL_TEXTURE_COORD_ARRAY_SIZE	The *params* parameter returns one value, the number of coordinates per element in the texture coordinate array. See **glTexCoordPointer**.
GL_TEXTURE_COORD_ARRAY_STRIDE	The *params* parameter returns one value, the byte offset between consecutive elements in the texture coordinate array. See **glTexCoordPointer**.
GL_TEXTURE_COORD_ARRAY_TYPE	The *params* parameter params returns one value, the data type of the coordinates in the texture coordinate array. See **glTexCoordPointer**.
GL_TEXTURE_ENV_COLOR	The *params* parameter returns four values: the red, green, blue, and alpha values of the texture environment color. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and 1.0 returns the most negative representable integer value. See glTexEnv.

VALUE	MEANING
GL_TEXTURE_ENV_MODE	The *params* parameter returns one value: a symbolic constant indicating which texture environment function is currently selected. See glTexEnv.
GL_TEXTURE_GEN_Q	The *params* parameter returns a single Boolean value indicating whether automatic generation of the Q texture coordinate is enabled. See **glTexGen**.
GL_TEXTURE_GEN_R	The *params* parameter returns a single Boolean value indicating whether automatic generation of the R texture coordinate is enabled. See **glTexGen**.
GL_TEXTURE_GEN_S	The *params* parameter returns a single Boolean value indicating whether automatic generation of the S texture coordinate is enabled. See **glTexGen**.
GL_TEXTURE_GEN_T	The *params* parameter returns a single Boolean value indicating whether automatic generation of the T texture coordinate is enabled. See **glTexGen**.
GL_TEXTURE_MATRIX	The *params* parameter returns 16 values: the texture matrix on the top of the texture matrix stack. See **glPushMatrix**.

VALUE	MEANING
GL_TEXTURE_STACK_DEPTH	The *params* parameter returns one value: the number of matrices on the texture matrix stack. See **glPushMatrix**.
GL_UNPACK_ALIGNMENT	The *params* parameter returns one value: the byte alignment used for reading pixel data from memory. See glPixelStore.
GL_UNPACK_LSB_FIRST	The *params* parameter returns a single Boolean value indicating whether single-bit pixels being read from memory are read first from the least significant bit of each unsigned byte. See glPixelStore.
GL_UNPACK_ROW_LENGTH	The *params* parameter returns one value: the row length used for reading pixel data from memory. See glPixelStore.
GL_UNPACK_SKIP_PIXELS	The *params* parameter returns one value: the number of pixel locations skipped before the first pixel is read from memory. See glPixelStore.
GL_UNPACK_SKIP_ROWS	The *params* parameter returns one value: the number of rows of pixel locations skipped before the first pixel is read from memory. See glPixelStore.

VALUE	MEANING
GL_UNPACK_SWAP_BYTES	The *params* parameter returns a single Boolean value indicating whether the bytes of 2-byte and 4-byte pixel indexes and components are swapped after being read from memory. See glPixelStore.
GL_VERTEX_ARRAY	The *params* parameter returns a single Boolean value indicating whether the vertex array is enabled. See **glVertexPointer**.
GL_VERTEX_ARRAY_SIZE	The *params* parameter returns one value, the number of coordinates per vertex in the vertex array. See **glVertexPointer**.
GL_VERTEX_ARRAY_STRIDE	The *params* parameter returns one value, the byte offset between consecutive vertexes in the vertex array. See **glVertexPointer**.
GL_VERTEX_ARRAY_TYPE	The *params* parameter returns one value, the data type of each coordinate in the vertex array. See **glVertexPointer**.
GL_VIEWPORT	The *params* parameter returns four values: the *x* and *y* window coordinates of the viewport, followed by its width and height. See **glViewport**.

VALUE	MEANING
GL_ZOOM_X	The *params* parameter returns one value: the *x* pixel zoom factor. See **glPixelZoom**.
GL_ZOOM_Y	The *params* parameter returns one value: the *y* pixel zoom factor. See **glPixelZoom**.

p a r a m s Returns the value or values of the specified parameter.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*pname* was not an accepted value.
GL_INVALID_OPERATION	**glGetIntegerv** was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*pname* was not an accepted value.
GL_INVALID_OPERATION	**glGetIntegerv** was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

This function returns values for simple state variables in OpenGL. The *pname* parameter is a symbolic constant indicating the state variable to be returned, and *params* is a pointer to an array of the indicated type in which to place the returned data.

Type conversion is performed if *params* has a different type from the state variable value being requested. If you call **glGetBooleanv**, a floating-point or integer value is converted to GL_FALSE if and only if it is zero. Otherwise, it is converted to GL_TRUE.

If you call **glGetIntegerv**, Boolean values are returned as GL_TRUE or GL_FALSE, and most floating-point values are rounded to the nearest integer value. Floating-point colors and normals, however, are returned with a linear mapping that maps 1.0 to the most positive representable integer value and 1.0 to the most negative representable integer value.

If you call **glGetFloatv** or **glGetDoublev**, Boolean values are returned as GL_TRUE or GL_FALSE, and integer values are converted to floating-point values.

You can query many of the Boolean parameters more easily with **glIsEnabled**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib

DLL	Opengl32.dll

See also

glAccum

glAlphaFunc

glBegin

glBlendFunc

glCa

l
l
L
i
s
t

g
l
C
l
e
a
r
A
c
c
u
m

g
l
C
l
e
a
r
C
o
l
o
r

g
l
C
l
e
a
r
D
e
p
t
h

g
l
C
l
e
a
r

Index

glClearStencil

glClipPlane

glColor

glColorMask

glColorMaterial

glCullFace

glDepthFunc

glDepthMask

gl

D
e
p
t
h
R
a
n
g
e

g
l
D
r
a
w
B
u
f
f
e
r

g
l
E
d
g
e
F
l
a
g

g
l
E
n
d

g
l
F
o
g

g
l
F
r
o
n

t
F
a
c
e

g
l
G
e
t
C
l
i
p
P
l
a
n
e

g
l
G
e
t
E
r
r
o
r

g
l
G
e
t
L
i
g
h
t

g
l
G
e
t
M
a
p

g

lGetMaterial

glGetPixelMap

glGetPolygonStipple

glGetS

t
r
i
n
g

g
l
G
e
t
T
e
x
E
n
v

g
l
G
e
t
T
e
x
G
e
n

g
l
G
e
t
T
e
x
I
m
a
g
e

g
l
G
e
t
T
e
x
L

e
v
e
l
P
a
r
a
m
e
t
e
r

g
l
G
e
t
T
e
x
P
a
r
a
m
e
t
e
r

g
l
H
i
n
t

g
l
I
n
d
e
x

g
l
I
n
d
e

x
M
a
s
k

g
l
I
s
E
n
a
b
l
e
d

g
l
L
i
g
h
t

g
l
L
i
g
h
t
M
o
d
e
l

g
l
L
i
n
e
S
t
i
p
p
l
e

glLineWidth

glListBase

glLogicOp

glMap1

glMap2

glMapG

r
i
d

g
l
M
a
t
r
i
x
M
o
d
e

g
l
N
e
w
L
i
s
t

g
l
N
o
r
m
a
l

g
l
P
i
x
e
l
M
a
p

g
l
P
i
x
e

lStore

glPixelTransfer

glPixelZoom

glPointSize

glPoly

g
o
n
M
o
d
e

g
l
P
o
l
y
g
o
n
S
t
i
p
p
l
e

g
l
P
u
s
h
A
t
t
r
i
b

g
l
P
u
s
h
M
a
t
r
i
x

g
l

P
u
s
h
N
a
m
e

g
l
R
a
s
t
e
r
P
o
s

g
l
R
e
a
d
P
i
x
e
l
s

g
l
S
c
i
s
s
o
r

g
l
S
h
a
d
e
M
o

d
e
l

g
l
S
t
e
n
c
i
l
F
u
n
c

g
l
S
t
e
n
c
i
l
M
a
s
k

g
l
S
t
e
n
c
i
l
O
p

g
l
T
e
x
C
o
o
r

d

glTexEnv

glTexGen

glTexImage1D

glTexImage2D

glViewp

o
r
t

glGetClipPlane function

2/22/2020 • 2 minutes to read • Edit Online

The **glGetClipPlane** function returns the coefficients of the specified clipping plane.

Syntax

```
void WINAPI glGetClipPlane(
   GLenum   plane,
   GLdouble *equation
);
```

Parameters

plane

A clipping plane. The number of clipping planes depends on the implementation, but at least six clipping planes are supported. They are identified by symbolic names of the form GL_CLIP_PLANE *i* where 0 = *i* < GL_MAX_CLIP_PLANES.

equation

Returns four double-precision values that are the coefficients of the plane equation of *plane* in eye coordinates.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*plane* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*plane* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glGetClipPlane** function returns in *equation* the four coefficients of the plane equation for *plane*.

It is always the case that GL_CLIP_PLANE*i* = GL_CLIP_PLANE0 + *i*.

If an error is generated, no change is made to the contents of *equation*.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glClipPlane

glEnd

glGetColorTableEXT function

2/22/2020 • 5 minutes to read • Edit Online

The **glGetColorTableEXT** function gets the color table data of the current targeted texture palette.

Syntax

```
void WINAPI glGetColorTableEXT(
         GLenum target,
         GLenum format,
         GLenum type,
   const GLvoid *data
);
```

Parameters

target

The target texture that is to have its palette changed. Must be TEXTURE_1D or TEXTURE_2D.

format

The format of the pixel data. The following symbolic constants are accepted.

VALUE	MEANING

VALUE	MEANING
GL_ RG BA	Each pixel is a group of four components in the following order: red, green, blue, alpha. The RGBA format is determined in this way: 1. The **glGetColorTableEXT** function converts floating-point values directly to an internal format with unspecified precision. Signed integer values are mapped linearly to the internal format such that the most positive representable integer value maps to 1.0, and the most negative representable integer value maps to -1.0. Unsigned integer data is mapped similarly: the largest integer value maps to 1.0, and zero maps to 0.0. 2. The **glGetColorTableEXT** function multiplies the resulting color values by GL_c_SCALE and adds them to GL_c_BIAS, where *c* is RED, GREEN, BLUE, and ALPHA for the respective color components. The results are clamped to the range [0,1]. 3. If GL_MAP_COLOR is **TRUE**, **glGetColorTableEXT** scales each color component by the size of lookup table GL_PIXEL_MAP_c_TO_c, then replaces the component by the value that it references in that table; *c* is R, G, B, or A, respectively. 4. The **glGetColorTableEXT** function converts the resulting RGBA colors to fragments by attaching the current raster position z-coordinate and texture coordinates to each pixel, then assigning *x* and *y* window coordinates to the *n*th fragment, such that $x? = x_r + n \bmod width$ $y? = y_r + n/width$ where (x_r , y_r) is the current raster position. 5. These pixel fragments are then treated just like the fragments generated by rasterizing points, lines, or polygons. The **glGetColorTableEXT** function applies texture mapping, fog, and all the fragment operations before writing the fragments to the framebuffer.

VALUE	MEANING
GL_RED	Each pixel is a single red component. The **glGetColorTableEXT** function converts this component to the internal format in the same way that the red component of an RGBA pixel is, then converts it to an RGBA pixel with green and blue set to 0.0, and alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.
GL_GREEN	Each pixel is a single green component. The **glGetColorTableEXT** function converts this component to the internal format in the same way that the green component of an RGBA pixel is, and then converts it to an RGBA pixel with red and blue set to 0.0, and alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.
GL_BLUE	Each pixel is a single blue component. The **glGetColorTableEXT** function converts this component to the internal format in the same way that the blue component of an RGBA pixel is, and then converts it to an RGBA pixel with red and green set to 0.0, and alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.
GL_ALPHA	Each pixel is a single alpha component. The **glGetColorTableEXT** function converts this component to the internal format in the same way that the alpha component of an RGBA pixel is, and then converts it to an RGBA pixel with red, green, and blue set to 0.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.
GL_RGB	Each pixel is a group of three components in this order: red, green, blue. The **glGetColorTableEXT** function converts each component to the internal format in the same way that the red, green, and blue components of an RGBA pixel are. The color triple is converted to an RGBA pixel with alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.

VALUE	MEANING
GL_BGR_EXT	Each pixel is a group of three components in this order: blue, green, red. GL_BGR_EXT provides a format that matches the memory layout of Microsoft Windows device-independent bitmaps (DIBs). Thus, your applications can use the same data with Windows function calls and OpenGL pixel function calls.
GL_BGRA_EXT	Each pixel is a group of four components in this order: blue, green, red, alpha. GL_BGRA_EXT provides a format that matches the memory layout of Windows device-independent bitmaps (DIBs). Thus your applications can use the same data with Windows function calls and OpenGL pixel function calls.

type

The data type for *data*. The following are the accepted symbolic constants and their meanings.

VALUE	MEANING
GL_UNSIGNED_BYTE	Unsigned 8-bit integer
GL_BYTE	Signed 8-bit integer
GL_UNSIGNED_SHORT	Unsigned 16-bit integer
GL_SHORT	Signed 16-bit integer

VALUE	MEANING
GL_UNSIGNED_INT	Unsigned 32-bit integer
GL_INT	32-bit integer
GL_FLOAT	Single-precision floating-point value

data

Points to the location where returned color table information is to be stored. Each color table entry is stored as if it was a single pixel of a 1-D texture. Because all textures have a default palette, **glGetColorTableEXT** always returns palette information even if the texture data is not in a paletted format.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target*, *format*, or *type* was not an accepted value.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target*, *format*, or *type* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glGetColorTableEXT** function gets the actual color table data specified by **glColorTableEXT** and **glColorSubTableEXT**.

The **glGetColorTableEXT** function is an extension function that is not part of the standard OpenGL library but is part of the GL_EXT_paletted_texture extension. To check whether your implementation of OpenGL supports **glGetColorTableEXT**, call **glGetString**(GL_EXTENSIONS). If it returns GL_EXT_paletted_texture, **glGetColorTableEXT** is supported. To obtain the function address of an extension function, call **wglGetProcAddress**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Gl.h

See also

glColorSubTableEXT

glColorTableEXT

glGetColor

T
a
b
l
e
P
a
r
a
m
e
t
e
r
f
v
E
X
T

g
l
G
e
t
C
o
l
o
r
T
a
b
l
e
P
a
r
a
m
e
t
e
r
i
v
E
X
T

w
g

l
G
e
t
P
r
o
c
A
d
d
r
e
s
s

glGetColorTableParameterfvEXT function

2/22/2020 • 2 minutes to read • Edit Online

The **glGetColorTableParameterfvEXT** and **glGetColorTableParameterivEXT** functions get palette parameters from color tables.

Syntax

```
void WINAPI glGetColorTableParameterfvEXT(
   GLenum  target,
   GLenum  pname,
   GLfloat *params
);
```

Parameters

target

The target texture of the palette for which you want parameter data. Must be TEXTURE_1D, TEXTURE_2D, PROXY_TEXTURE_1D, or PROXY_TEXTURE_2D.

pname

A symbolic constant for the type of palette parameter data pointed to by *params*.

The following are the accepted symbolic constants and their meanings.

VALUE	MEANING
GL_COLOR_TABLE_FORMAT_EXT	Return the internal format specified by the most recent call to **glColorTableEXT** or the default value.

VALUE	MEANING
GL_COLOR_TABLE_WIDTH_EXT	Return the width of the current palette.
GL_COLOR_TABLE_RED_SIZE_EXT	Return the actual size used internally to store the red component of the palette data.
GL_COLOR_TABLE_GREEN_SIZE_EXT	Return the actual size used internally to store the green component of the palette data.
GL_COLOR_TABLE_BLUE_SIZE_EXT	Return the actual size used internally to store the blue component of the palette data.

VALUE	MEANING
GL_COLOR_TABLE_ALPHA_SIZE_EXT	Return the actual size used internally to store the alpha component of the palette data.

p a r a m s

Points to the color table parameter data specified by the *pname* parameter.

Return value

This function does not return a value.

Remarks

You use the **glGetColorTableParameterivEXT** and **glGetColorTableParameterfvEXT** functions to retrieve specific parameter data from color tables set with glColorTableEXT for targeted texture palettes. Also you can use these functions to determine the number of color table entries that **glGetColorTableEXT** returns.

When the *target* parameter is GL_PROXY_TEXTURE_1D or GL_PROXY_TEXTURE_2D, and the implementation does not support the values specified for either *format* or *width*, **glColorTableEXT** can fail to create the requested color table. In this case, the color table is empty and all parameters retrieved will be zero. You can determine whether OpenGL supports a particular color table format and size by calling **glColorTableEXT** with a proxy target, and then calling **glGetColorTableParameterivEXT** or **glGetColorTableParameterfvEXT** to determine whether the width parameter matches that set by **glColorTableEXT**. If the retrieved width is zero, the color table request by **glColorTable** failed. If the retrieved width is not zero, you can call **glColorTable** with the real target with TEXTURE_1D or TEXTURE_2D to set the color table.

The **glGetColorTableParameterivEXT** and **glGetColorTableParameterfvEXT** functions are extension functions that are not part of the standard OpenGL library but are part of the GL_EXT_paletted_texture extension. To check whether your implementation of OpenGL supports **glGetColorTableParameterivEXT** and **glGetColorTableParameterfvEXT**, call glGetString(GL_EXTENSIONS). If it returns GL_EXT_paletted_texture, **glGetColorTableParameterivEXT** and **glGetColorTableParameterfvEXT** are supported. To obtain the function address of an extension function, call wglGetProcAddress.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

See also

glColorSubTableEXT

glColorTableEXT

glGetCol

o
r
T
a
b
l
e
E
X
T

g
l
G
e
t
C
o
l
o
r
T
a
b
l
e
P
a
r
a
m
e
t
e
r
i
v
E
X
T

w
g
l
G
e
t
P
r
o
c
A

d
d
r
e
s
s

glGetColorTableParameterivEXT function

2/22/2020 • 2 minutes to read • Edit Online

The **glGetColorTableParameterfvEXT** and **glGetColorTableParameterivEXT** functions get palette parameters from color tables.

Syntax

```
void WINAPI glGetColorTableParameterivEXT(
   GLenum target,
   GLenum pname,
   GLint  *params
);
```

Parameters

target

The target texture of the palette for which you want parameter data. Must be TEXTURE_1D, TEXTURE_2D, PROXY_TEXTURE_1D, or PROXY_TEXTURE_2D.

pname

A symbolic constant for the type of palette parameter data pointed to by *params*.

The following are the accepted symbolic constants and their meanings.

VALUE	MEANING
GL_COLOR_TABLE_FORMAT_EXT	Return the internal format specified by the most recent call to **glColorTableEXT** or the default value.

VALUE	MEANING
GL_COLOR_TABLE_WIDTH_EXT	Return the width of the current palette.
GL_COLOR_TABLE_RED_SIZE_EXT	Return the actual size used internally to store the red component of the palette data.
GL_COLOR_TABLE_GREEN_SIZE_EXT	Return the actual size used internally to store the green component of the palette data.
GL_COLOR_TABLE_BLUE_SIZE_EXT	Return the actual size used internally to store the blue component of the palette data.

VALUE	MEANING
GL_COLOR_TABLE_ALPHA_SIZE_EXT	Return the actual size used internally to store the alpha component of the palette data.

p a r a m s

Points to the color table parameter data specified by the *pname* parameter.

Return value

This function does not return a value.

Remarks

You use the **glGetColorTableParameterivEXT** and **glGetColorTableParameterfvEXT** functions to retrieve specific parameter data from color tables set with **glColorTableEXT** for targeted texture palettes. Also you can use these functions to determine the number of color table entries that **glGetColorTableEXT** returns.

When the *target* parameter is GL_PROXY_TEXTURE_1D or GL_PROXY_TEXTURE_2D, and the implementation does not support the values specified for either *format* or *width*, **glColorTableEXT** can fail to create the requested color table. In this case, the color table is empty and all parameters retrieved will be zero. You can determine whether OpenGL supports a particular color table format and size by calling **glColorTableEXT** with a proxy target, and then calling **glGetColorTableParameterivEXT** or **glGetColorTableParameterfvEXT** to determine whether the width parameter matches that set by **glColorTableEXT**. If the retrieved width is zero, the color table request by **glColorTable** failed. If the retrieved width is not zero, you can call **glColorTable** with the real target with TEXTURE_1D or TEXTURE_2D to set the color table.

The **glGetColorTableParameterivEXT** and **glGetColorTableParameterfvEXT** functions are extension functions that are not part of the standard OpenGL library but are part of the GL_EXT_paletted_texture extension. To check whether your implementation of OpenGL supports **glGetColorTableParameterivEXT** and **glGetColorTableParameterfvEXT**, call **glGetString**(GL_EXTENSIONS). If it returns GL_EXT_paletted_texture, **glGetColorTableParameterivEXT** and **glGetColorTableParameterfvEXT** are supported. To obtain the function address of an extension function, call **wglGetProcAddress**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

See also

glColorSubTableEXT

glColorTableEXT

glGetCol

o
r
T
a
b
l
e
E
X
T

g
l
G
e
t
C
o
l
o
r
T
a
b
l
e
P
a
r
a
m
e
t
e
r
f
v
E
X
T

w
g
l
G
e
t
P
r
o
c
A

d
d
r
e
s
s

glGetError function

2/22/2020 • 2 minutes to read • Edit Online

The **glGetError** function returns error information.

Syntax

```
GLenum WINAPI glGetError(void);
```

Parameters

This function has no parameters.

Return value

The **glGetError** function returns one of the following error codes.

RETURN CODE	DESCRIPTION
GL_INVALID_ENUM	An unacceptable value is specified for an enumerated argument. The offending function is ignored, having no side effect other than to set the error flag.
GL_INVALID_VALUE	A numeric argument is out of range. The offending function is ignored, having no side effect other than to set the error flag.
GL_INVALID_OPERATION	The specified operation is not allowed in the current state. The offending function is ignored, having no side effect other than to set the error flag.

RETURN CODE	DESCRIPTION
GL_NO_ERROR	No error has been recorded. The value of this symbolic constant is guaranteed to be zero.
GL_STACK_OVERFLOW	This function would cause a stack overflow. The offending function is ignored, having no side effect other than to set the error flag.
GL_STACK_UNDERFLOW	This function would cause a stack underflow. The offending function is ignored, having no side effect other than to set the error flag.
GL_OUT_OF_MEMORY	There is not enough memory left to execute the function. The state of OpenGL is undefined, except for the state of the error flags, after this error is recorded.

Note that **glGetError** returns GL_INVALID_OPERATION if it is called between a call to **glBegin** and its corresponding call to **glEnd**.

Remarks

Each detectable error is assigned a numeric code and symbolic name. When an error occurs, the error flag is set to the appropriate error code value. No other errors are recorded until **glGetError** is called, the error code is returned, and the flag is reset to GL_NO_ERROR. If a call to **glGetError** returns GL_NO_ERROR, there has been no detectable error since the last call to **glGetError**, or since OpenGL was initialized.

To allow for distributed implementations, there may be several error flags. If any single error

flag has recorded an error, the value of that flag is returned and that flag is reset to GL_NO_ERROR when **glGetError** is called. If more than one flag has recorded an error, **glGetError** returns and clears an arbitrary error flag value. If all error flags are to be reset, you should always call **glGetError** in a loop until it returns GL_NO_ERROR.

Initially, all error flags are set to GL_NO_ERROR.

When an error flag is set, results of an OpenGL operation are undefined only if GL_OUT_OF_MEMORY has occurred. In all other cases, the function generating the error is ignored and has no effect on the OpenGL state or framebuffer contents.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBeginning

I
E
n
d

glGetLight Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions return light source parameter values:

- **glGetLightfv**
- **glGetLightiv**

glGetLightfv function

2/22/2020 • 3 minutes to read • Edit Online

The **glGetLightfv** and **glGetLightiv** functions return light source parameter values.

Syntax

```
void WINAPI glGetLightfv(
   GLenum  light,
   GLenum  pname,
   GLfloat *params
);
```

Parameters

light

A light source. The number of possible lights depends on the implementation, but at least eight lights are supported. They are identified by symbolic names of the form GL_LIGHT *i* where 0 = *i* < GL_MAX_LIGHTS.

pname

A light source parameter for *light*. The following symbolic names are accepted.

VALUE	MEANING
GL_AMBIENT	The *params* parameter returns four integer or floating-point values representing the ambient intensity of the light source. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined.

VALUE	MEANING
GL_DIFFUSE	The *params* parameter returns four integer or floating-point values representing the diffuse intensity of the light source. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined.
GL_SPECULAR	The *params* parameter returns four integer or floating-point values representing the specular intensity of the light source. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined.
GL_POSITION	The *params* parameter returns four integer or floating-point values representing the position of the light source. Integer values, when requested, are computed by rounding the internal floating-point values to the nearest integer value. The returned values are those maintained in eye coordinates. They will not be equal to the values specified using **glLight**, unless the modelview matrix was identified at the time **glLight** was called.
GL_SPOT_DIRECTION	The *params* parameter returns three integer or floating-point values representing the direction of the light source. Integer values, when requested, are computed by rounding the internal floating-point values to the nearest integer value. The returned values are those maintained in eye coordinates. They will not be equal to the values specified using **glLight**, unless the modelview matrix was identified at the time **glLight** was called. Although spot direction is normalized before being used in the lighting equation, the returned values are the transformed versions of the specified values prior to normalization.

VALUE	MEANING
GL_SPOT_EXPONENT	The *params* parameter returns a single integer or floating-point value representing the spot exponent of the light. An integer value, when requested, is computed by rounding the internal floating-point representation to the nearest integer.
GL_SPOT_CUTOFF	The *params* parameter returns a single integer or floating-point value representing the spot cutoff angle of the light. An integer value, when requested, is computed by rounding the internal floating-point representation to the nearest integer.
GL_CONSTANT_ATTENUATION	The *params* parameter returns a single integer or floating-point value representing the constant (not distance-related) attenuation of the light. An integer value, when requested, is computed by rounding the internal floating-point representation to the nearest integer.
GL_LINEAR_ATTENUATION	The *params* parameter returns a single integer or floating-point value representing the linear attenuation of the light. An integer value, when requested, is computed by rounding the internal floating-point representation to the nearest integer.
GL_QUADRATIC_ATTENUATION	The *params* parameter returns a single integer or floating-point value representing the quadratic attenuation of the light. An integer value, when requested, is computed by rounding the internal floating-point representation to the nearest integer.

params

Returns the requested data.

Return value

This function does not return a value.

Remarks

The **glGetLight** function returns in *params* the value or values of a light source parameter. The *light* parameter names the light and is a symbolic name of the form GL_LIGHT*i* for 0 = *i* < GL_MAX_LIGHTS, where GL_MAX_LIGHTS is an implementation-dependent constant that is greater than or equal to eight. The *pname* parameter specifies one of ten light source parameters, again by symbolic name.

It is always the case that GL_LIGHT*i* = GL_LIGHT0 + *i*.

If an error is generated, no change is made to the contents of *params*.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

gl

L
i
g
h
t

glGetLightiv function

2/22/2020 • 3 minutes to read • Edit Online

The **glGetLightfv** and **glGetLightiv** functions return light source parameter values.

Syntax

```
void WINAPI glGetLightiv(
   GLenum light,
   GLenum pname,
   GLint  *params
);
```

Parameters

light

A light source. The number of possible lights depends on the implementation, but at least eight lights are supported. They are identified by symbolic names of the form GL_LIGHT *i* where 0 = *i* < GL_MAX_LIGHTS.

pname

A light source parameter for *light*. The following symbolic names are accepted.

VALUE	MEANING
GL_AMBIENT	The *params* parameter returns four integer or floating-point values representing the ambient intensity of the light source. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined.

VALUE	MEANING
GL_DIFFUSE	The *params* parameter returns four integer or floating-point values representing the diffuse intensity of the light source. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined.
GL_SPECULAR	The *params* parameter returns four integer or floating-point values representing the specular intensity of the light source. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined.
GL_POSITION	The *params* parameter returns four integer or floating-point values representing the position of the light source. Integer values, when requested, are computed by rounding the internal floating-point values to the nearest integer value. The returned values are those maintained in eye coordinates. They will not be equal to the values specified using **glLight**, unless the modelview matrix was identified at the time **glLight** was called.
GL_SPOT_DIRECTION	The *params* parameter returns three integer or floating-point values representing the direction of the light source. Integer values, when requested, are computed by rounding the internal floating-point values to the nearest integer value. The returned values are those maintained in eye coordinates. They will not be equal to the values specified using **glLight**, unless the modelview matrix was identified at the time **glLight** was called. Although spot direction is normalized before being used in the lighting equation, the returned values are the transformed versions of the specified values prior to normalization.

VALUE	MEANING
GL_SPOT_EXPONENT	The *params* parameter returns a single integer or floating-point value representing the spot exponent of the light. An integer value, when requested, is computed by rounding the internal floating-point representation to the nearest integer.
GL_SPOT_CUTOFF	The *params* parameter returns a single integer or floating-point value representing the spot cutoff angle of the light. An integer value, when requested, is computed by rounding the internal floating-point representation to the nearest integer.
GL_CONSTANT_ATTENUATION	The *params* parameter returns a single integer or floating-point value representing the constant (not distance-related) attenuation of the light. An integer value, when requested, is computed by rounding the internal floating-point representation to the nearest integer.
GL_LINEAR_ATTENUATION	The *params* parameter returns a single integer or floating-point value representing the linear attenuation of the light. An integer value, when requested, is computed by rounding the internal floating-point representation to the nearest integer.
GL_QUADRATIC_ATTENUATION	The *params* parameter returns a single integer or floating-point value representing the quadratic attenuation of the light. An integer value, when requested, is computed by rounding the internal floating-point representation to the nearest integer.

params Returns the requested data.

Return value

This function does not return a value.

Remarks

The **glGetLight** function returns in *params* the value or values of a light source parameter. The *light* parameter names the light and is a symbolic name of the form GL_LIGHT*i* for 0 = *i* < GL_MAX_LIGHTS, where GL_MAX_LIGHTS is an implementation-dependent constant that is greater than or equal to eight. The *pname* parameter specifies one of ten light source parameters, again by symbolic name.

It is always the case that GL_LIGHT*i* = GL_LIGHT0 + *i*.

If an error is generated, no change is made to the contents of *params*.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

gl

L
i
g
h
t

glGetMap Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions return evaluator parameters:

- glGetMapdv
- glGetMapfv
- glGetMapiv

glGetMapdv function

2/22/2020 • 2 minutes to read • Edit Online

The **glGetMapdv**, **glGetMapfv**, and **glGetMapiv** functions return evaluator parameters.

Syntax

```
void WINAPI glGetMapdv(
   GLenum   target,
   GLenum   query,
   GLdouble *v
);
```

Parameters

target

The symbolic name of a map. The following are accepted values: GL_MAP1_COLOR_4, GL_MAP1_INDEX, GL_MAP1_NORMAL, GL_MAP1_TEXTURE_COORD_1, GL_MAP1_TEXTURE_COORD_2, GL_MAP1_TEXTURE_COORD_3, GL_MAP1_TEXTURE_COORD_4, GL_MAP1_VERTEX_3, GL_MAP1_VERTEX_4, GL_MAP2_COLOR_4, GL_MAP2_INDEX, GL_MAP2_NORMAL, GL_MAP2_TEXTURE_COORD_1, GL_MAP2_TEXTURE_COORD_2, GL_MAP2_TEXTURE_COORD_3, GL_MAP2_TEXTURE_COORD_4, GL_MAP2_VERTEX_3, and GL_MAP2_VERTEX_4.

query

Specifies which parameter to return. The following symbolic names are accepted.

VALUE	MEANING
GL_COEFF	The *v* parameter returns the control points for the evaluator function. One-dimensional evaluators return *order* control points, and two-dimensional evaluators return *uorder x vorder* control points. Each control point consists of one, two, three, or four integer, single-precision floating-point, or double-precision floating-point values, depending on the type of the evaluator. Two-dimensional control points are returned in row-major order, incrementing the *uorder* index quickly, and the *vorder* index after each row. Integer values, when requested, are computed by rounding the internal floating-point values to the nearest integer values.

VALUE	MEANING
GL_ORDER	The *v* parameter returns the order of the evaluator function. One-dimensional evaluators return a single value, *order*. Two-dimensional evaluators return two values, *uorder* and *vorder*.
GL_DOMAIN	The *v* parameter returns the linear *u* and *v* mapping parameters. One-dimensional evaluators return two values, *u* 1 and *u* 2, as specified by **glMap1**. Two-dimensional evaluators return four values (*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.

v Returns the requested data.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *query* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *query* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glGetMap** function returns evaluator parameters. (The **glMap1** and **glMap2** functions define evaluators.) The *target* parameter specifies a map, *query* selects a specific parameter, and *v* points to storage where the values will be returned.

The acceptable values for the *target* parameter are described in **glMap1** and **glMap2**.

If an error is generated, no change is made to the contents of *v*.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

gl

B
e
g
i
n

g
l
E
n
d

g
l
E
v
a
l
C
o
o
r
d

g
l
M
a
p
1

g
l
M
a
p
2

glGetMapfv function

2/22/2020 • 2 minutes to read • Edit Online

The **glGetMapdv**, **glGetMapfv**, and **glGetMapiv** functions return evaluator parameters.

Syntax

```
void WINAPI glGetMapfv(
   GLenum  target,
   GLenum  query,
   GLfloat *v
);
```

Parameters

target

The symbolic name of a map. The following are accepted values: GL_MAP1_COLOR_4, GL_MAP1_INDEX, GL_MAP1_NORMAL, GL_MAP1_TEXTURE_COORD_1, GL_MAP1_TEXTURE_COORD_2, GL_MAP1_TEXTURE_COORD_3, GL_MAP1_TEXTURE_COORD_4, GL_MAP1_VERTEX_3, GL_MAP1_VERTEX_4, GL_MAP2_COLOR_4, GL_MAP2_INDEX, GL_MAP2_NORMAL, GL_MAP2_TEXTURE_COORD_1, GL_MAP2_TEXTURE_COORD_2, GL_MAP2_TEXTURE_COORD_3, GL_MAP2_TEXTURE_COORD_4, GL_MAP2_VERTEX_3, and GL_MAP2_VERTEX_4.

query

Specifies which parameter to return. The following symbolic names are accepted.

VALUE	MEANING
GL_COEFF	The *v* parameter returns the control points for the evaluator function. One-dimensional evaluators return *order* control points, and two-dimensional evaluators return *uorder x vorder* control points. Each control point consists of one, two, three, or four integer, single-precision floating-point, or double-precision floating-point values, depending on the type of the evaluator. Two-dimensional control points are returned in row-major order, incrementing the *uorder* index quickly, and the *vorder* index after each row. Integer values, when requested, are computed by rounding the internal floating-point values to the nearest integer values.

VALUE	MEANING
GL_ORDER	The *v* parameter returns the order of the evaluator function. One-dimensional evaluators return a single value, *order*. Two-dimensional evaluators return two values, *uorder* and *vorder*.
GL_DOMAIN	The *v* parameter returns the linear *u* and *v* mapping parameters. One-dimensional evaluators return two values, *u* 1 and *u* 2, as specified by **glMap1**. Two-dimensional evaluators return four values (*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.

v Returns the requested data.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *query* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *query* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glGetMap** function returns evaluator parameters. (The **glMap1** and **glMap2** functions define evaluators.) The *target* parameter specifies a map, *query* selects a specific parameter, and *v* points to storage where the values will be returned.

The acceptable values for the *target* parameter are described in **glMap1** and **glMap2**.

If an error is generated, no change is made to the contents of *v*.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

gl

B
e
g
i
n

g
l
E
n
d

g
l
E
v
a
l
C
o
o
r
d

g
l
M
a
p
1

g
l
M
a
p
2

glGetMapiv function

2/22/2020 • 2 minutes to read • Edit Online

The **glGetMapdv**, **glGetMapfv**, and **glGetMapiv** functions return evaluator parameters.

Syntax

```
void WINAPI glGetMapiv(
   GLenum target,
   GLenum query,
   GLint  *v
);
```

Parameters

target

The symbolic name of a map. The following are accepted values: GL_MAP1_COLOR_4, GL_MAP1_INDEX, GL_MAP1_NORMAL, GL_MAP1_TEXTURE_COORD_1, GL_MAP1_TEXTURE_COORD_2, GL_MAP1_TEXTURE_COORD_3, GL_MAP1_TEXTURE_COORD_4, GL_MAP1_VERTEX_3, GL_MAP1_VERTEX_4, GL_MAP2_COLOR_4, GL_MAP2_INDEX, GL_MAP2_NORMAL, GL_MAP2_TEXTURE_COORD_1, GL_MAP2_TEXTURE_COORD_2, GL_MAP2_TEXTURE_COORD_3, GL_MAP2_TEXTURE_COORD_4, GL_MAP2_VERTEX_3, and GL_MAP2_VERTEX_4.

query

Specifies which parameter to return. The following symbolic names are accepted.

VALUE	MEANING
GL_COEFF	The *v* parameter returns the control points for the evaluator function. One-dimensional evaluators return *order* control points, and two-dimensional evaluators return *uorder x vorder* control points. Each control point consists of one, two, three, or four integer, single-precision floating-point, or double-precision floating-point values, depending on the type of the evaluator. Two-dimensional control points are returned in row-major order, incrementing the *uorder* index quickly, and the *vorder* index after each row. Integer values, when requested, are computed by rounding the internal floating-point values to the nearest integer values.

VALUE	MEANING
GL_ORDER	The *v* parameter returns the order of the evaluator function. One-dimensional evaluators return a single value, *order*. Two-dimensional evaluators return two values, *uorder* and *vorder*.
GL_DOMAIN	The *v* parameter returns the linear *u* and *v* mapping parameters. One-dimensional evaluators return two values, *u* 1 and *u* 2, as specified by **glMap1**. Two-dimensional evaluators return four values (*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.

v Returns the requested data.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *query* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *query* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glGetMap** functions return evaluator parameters. (The **glMap1** and **glMap2** functions define evaluators.) The *target* parameter specifies a map, *query* selects a specific parameter, and *v* points to storage where the values will be returned.

The acceptable values for the *target* parameter are described in **glMap1** and **glMap2**.

If an error is generated, no change is made to the contents of *v*.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

gl

B
e
g
i
n

g
l
E
n
d

g
l
E
v
a
l
C
o
o
r
d

g
l
M
a
p
1

g
l
M
a
p
2

glGetMaterial Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions return material parameters:

- glGetMaterialfv
- glGetMaterialiv

glGetMaterialfv function

2/22/2020 • 2 minutes to read • Edit Online

The **glGetMaterialfv** and **glGetMaterialiv** functions return material parameters.

Syntax

```
void WINAPI glGetMaterialfv(
   GLenum  face,
   GLenum  pname,
   GLfloat *params
);
```

Parameters

face

Specifies which of the two materials is being queried. GL_FRONT or GL_BACK are accepted, representing the front and back materials, respectively.

pname

The material parameter to return. The following values are accepted.

VALUE	MEANING
GL_AMBIENT	The *params* parameter returns four integer or floating-point values representing the ambient reflectance of the material. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined.

VALUE	MEANING
GL_DIFFUSE	The *params* parameter returns four integer or floating-point values representing the diffuse reflectance of the material. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined.
GL_SPECULAR	The *params* parameter returns four integer or floating-point values representing the specular reflectance of the material. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined.
GL_EMISSION	The *params* parameter returns four integer or floating-point values representing the emitted light intensity of the material. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined.
GL_SHININESS	The *params* parameter returns one integer or floating-point value representing the specular exponent of the material. Integer values, when requested, are computed by rounding the internal floating-point value to the nearest integer value.

VALUE	MEANING
GL_COLOR_INDEXES	The *params* parameter returns three integer or floating-point values representing the ambient, diffuse, and specular indexes of the material. Use these indexes only for color-index lighting. (The other parameters are all used only for RGBA lighting.) Integer values, when requested, are computed by rounding the internal floating-point values to the nearest integer values.

p a r a m s

Returns the requested data.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *query* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *query* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glGetMaterial** function returns in *params* the value or values of parameter *pname* of material *face*.

If an error is generated, no change is made to the contents of *params*.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBeg

i
n

g
l
E
n
d

g
l
M
a
t
e
r
i
a
l

glGetMaterialiv function

2/22/2020 • 2 minutes to read • Edit Online

The **glGetMaterialfv** and **glGetMaterialiv** functions return material parameters.

Syntax

```
void WINAPI glGetMaterialiv(
   GLenum face,
   GLenum pname,
   GLint  *params
);
```

Parameters

face

Specifies which of the two materials is being queried. GL_FRONT or GL_BACK are accepted, representing the front and back materials, respectively.

pname

The material parameter to return. The following values are accepted.

VALUE	MEANING
GL_AMBIENT	The *params* parameter returns four integer or floating-point values representing the ambient reflectance of the material. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined.

VALUE	MEANING
GL_DIFFUSE	The *params* parameter returns four integer or floating-point values representing the diffuse reflectance of the material. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined.
GL_SPECULAR	The *params* parameter returns four integer or floating-point values representing the specular reflectance of the material. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined.
GL_EMISSION	The *params* parameter returns four integer or floating-point values representing the emitted light intensity of the material. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined.
GL_SHININESS	The *params* parameter returns one integer or floating-point value representing the specular exponent of the material. Integer values, when requested, are computed by rounding the internal floating-point value to the nearest integer value.

VALUE	MEANING
GL_COLOR_INDEXES	The *params* parameter returns three integer or floating-point values representing the ambient, diffuse, and specular indexes of the material. Use these indexes only for color-index lighting. (The other parameters are all used only for RGBA lighting.) Integer values, when requested, are computed by rounding the internal floating-point values to the nearest integer values.

p
a
r
a
m
s

Returns the requested data.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *query* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_ INV ALI D_E NU M	*target* or *query* was not an accepted value.
GL_ INV ALI D_ OP ERA TIO N	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glGetMaterial** function returns in *params* the value or values of parameter *pname* of material *face*.

If an error is generated, no change is made to the contents of *params*.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib
DLL	Ope ngl3 2.dll

See also

g
l
B
e
g

i
n

g
l
E
n
d

g
l
M
a
t
e
r
i
a
l

glGetPixelMap Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions return the specified pixel map:

- **glGetPixelMapfv**
- **glGetPixelMapuiv**
- **glGetPixelMapusv**

glGetPixelMapfv function

2/22/2020 • 2 minutes to read • Edit Online

The **glGetPixelMapfv**, **glGetPixelMapuiv**, and **glGetPixelMapusv** functions return the specified pixel map.

Syntax

```
void WINAPI glGetPixelMapfv(
   GLenum  map,
   GLfloat *values
);
```

Parameters

map

The name of the pixel map to return. Accepted values are GL_PIXEL_MAP_I_TO_I, GL_PIXEL_MAP_S_TO_S, GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, GL_PIXEL_MAP_I_TO_A, GL_PIXEL_MAP_R_TO_R, GL_PIXEL_MAP_G_TO_G, GL_PIXEL_MAP_B_TO_B, and GL_PIXEL_MAP_A_TO_A.

values

Returns the pixel map contents.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*map* was not an accepted value.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*map* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Remarks

See glPixelMap for a description of the acceptable values for the *map* parameter. The **glGetPixelMap** function returns in *values* the contents of the pixel map specified in *map*. Use pixel maps during the execution of glReadPixels, glDrawPixels, glCopyPixels, glTexImage1D, and glTexImage2D to map color indexes, stencil indexes, color components, and depth components to other values.

Unsigned integer values, if requested, are linearly mapped from the internal fixed or floating-point representation such that 1.0 maps to the largest representable integer value, and 0.0 maps to zero. Return unsigned integer values are undefined if the map value was not in the range [0,1].

To determine the required size of *map*, call glGet with the appropriate symbolic constant.

If an error is generated, no change is made to the contents of *values*.

The following functions retrieve information related to **glGetPixelMap**:

glGet with argument GL_PIXEL_MAP_I_TO_I_SIZE

glGet with argument GL_PIXEL_MAP_S_TO_S_SIZE

glGet with argument GL_PIXEL_MAP_I_TO_R_SIZE

glGet with argument GL_PIXEL_MAP_I_TO_G_SIZE

glGet with argument GL_PIXEL_MAP_I_TO_B_SIZE

glGet with argument GL_PIXEL_MAP_I_TO_A_SIZE

glGet with argument GL_PIXEL_MAP_R_TO_R_SIZE

glGet with argument GL_PIXEL_MAP_G_TO_G_SIZE

glGet with argument GL_PIXEL_MAP_B_TO_B_SIZE

glGet with argument GL_PIXEL_MAP_A_TO_A_SIZE

glGet with argument GL_MAX_PIXEL_MAP_TABLE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glCopyPix

e
l
s

g
l
D
r
a
w
P
i
x
e
l
s

g
l
E
n
d

g
l
G
e
t

g
l
P
i
x
e
l
M
a
p

g
l
P
i
x
e
l
T
r
a
n
s
f

e
r

g
l
R
e
a
d
P
i
x
e
l
s

g
l
T
e
x
I
m
a
g
e
1
D

g
l
T
e
x
I
m
a
g
e
2
D

glGetPixelMapuiv function

2/22/2020 • 2 minutes to read • Edit Online

The **glGetPixelMapfv**, **glGetPixelMapuiv**, and **glGetPixelMapusv** functions return the specified pixel map.

Syntax

```
void WINAPI glGetPixelMapuiv(
   GLenum map,
   GLuint *values
);
```

Parameters

map

The name of the pixel map to return. Accepted values are GL_PIXEL_MAP_I_TO_I, GL_PIXEL_MAP_S_TO_S, GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, GL_PIXEL_MAP_I_TO_A, GL_PIXEL_MAP_R_TO_R, GL_PIXEL_MAP_G_TO_G, GL_PIXEL_MAP_B_TO_B, and GL_PIXEL_MAP_A_TO_A.

values

Returns the pixel map contents.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*map* was not an accepted value.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*map* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

See **glPixelMap** for a description of the acceptable values for the *map* parameter. The **glGetPixelMap** function returns in *values* the contents of the pixel map specified in *map*. Use pixel maps during the execution of **glReadPixels**, **glDrawPixels**, **glCopyPixels**, **glTexImage1D**, and **glTexImage2D** to map color indexes, stencil indexes, color components, and depth components to other values.

Unsigned integer values, if requested, are linearly mapped from the internal fixed or floating-point representation such that 1.0 maps to the largest representable integer value, and 0.0 maps to zero. Return unsigned integer values are undefined if the map value was not in the range [0,1].

To determine the required size of *map*, call **glGet** with the appropriate symbolic constant.

If an error is generated, no change is made to the contents of *values*.

The following functions retrieve information related to **glGetPixelMap**:

glGet with argument GL_PIXEL_MAP_I_TO_I_SIZE

glGet with argument GL_PIXEL_MAP_S_TO_S_SIZE

glGet with argument GL_PIXEL_MAP_I_TO_R_SIZE

glGet with argument GL_PIXEL_MAP_I_TO_G_SIZE

glGet with argument GL_PIXEL_MAP_I_TO_B_SIZE

glGet with argument GL_PIXEL_MAP_I_TO_A_SIZE

glGet with argument GL_PIXEL_MAP_R_TO_R_SIZE

glGet with argument GL_PIXEL_MAP_G_TO_G_SIZE

glGet with argument GL_PIXEL_MAP_B_TO_B_SIZE

glGet with argument GL_PIXEL_MAP_A_TO_A_SIZE

glGet with argument GL_MAX_PIXEL_MAP_TABLE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib
DLL	Ope ngl3 2.dll

See also

glBegin

glCopyPix

e
l
s

g
l
D
r
a
w
P
i
x
e
l
s

g
l
E
n
d

g
l
G
e
t

g
l
P
i
x
e
l
M
a
p

g
l
P
i
x
e
l
T
r
a
n
s
f

e
r

g
l
R
e
a
d
P
i
x
e
l
s

g
l
T
e
x
I
m
a
g
e
1
D

g
l
T
e
x
I
m
a
g
e
2
D

glGetPixelMapusv function

2/22/2020 • 2 minutes to read • Edit Online

The **glGetPixelMapfv**, **glGetPixelMapuiv**, and **glGetPixelMapusv** functions return the specified pixel map.

Syntax

```
void WINAPI glGetPixelMapusv(
   GLenum   map,
   GLushort *values
);
```

Parameters

map

The name of the pixel map to return. Accepted values are GL_PIXEL_MAP_I_TO_I, GL_PIXEL_MAP_S_TO_S, GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, GL_PIXEL_MAP_I_TO_A, GL_PIXEL_MAP_R_TO_R, GL_PIXEL_MAP_G_TO_G, GL_PIXEL_MAP_B_TO_B, and GL_PIXEL_MAP_A_TO_A.

values

Returns the pixel map contents.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*map* was not an accepted value.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*map* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

See **glPixelMap** for a description of the acceptable values for the *map* parameter. The **glGetPixelMap** function returns in *values* the contents of the pixel map specified in *map*. Use pixel maps during the execution of **glReadPixels**, **glDrawPixels**, **glCopyPixels**, **glTexImage1D**, and **glTexImage2D** to map color indexes, stencil indexes, color components, and depth components to other values.

Unsigned integer values, if requested, are linearly mapped from the internal fixed or floating-point representation such that 1.0 maps to the largest representable integer value, and 0.0 maps to zero. Return unsigned integer values are undefined if the map value was not in the range [0,1].

To determine the required size of *map*, call **glGet** with the appropriate symbolic constant.

If an error is generated, no change is made to the contents of *values*.

The following functions retrieve information related to **glGetPixelMap**:

glGet with argument GL_PIXEL_MAP_I_TO_I_SIZE

glGet with argument GL_PIXEL_MAP_S_TO_S_SIZE

glGet with argument GL_PIXEL_MAP_I_TO_R_SIZE

glGet with argument GL_PIXEL_MAP_I_TO_G_SIZE

glGet with argument GL_PIXEL_MAP_I_TO_B_SIZE

glGet with argument GL_PIXEL_MAP_I_TO_A_SIZE

glGet with argument GL_PIXEL_MAP_R_TO_R_SIZE

glGet with argument GL_PIXEL_MAP_G_TO_G_SIZE

glGet with argument GL_PIXEL_MAP_B_TO_B_SIZE

glGet with argument GL_PIXEL_MAP_A_TO_A_SIZE

glGet with argument GL_MAX_PIXEL_MAP_TABLE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glCopyPix

e
l
s

g
l
D
r
a
w
P
i
x
e
l
s

g
l
E
n
d

g
l
G
e
t

g
l
P
i
x
e
l
M
a
p

g
l
P
i
x
e
l
T
r
a
n
s
f

e
r

g
l
R
e
a
d
P
i
x
e
l
s

g
l
T
e
x
I
m
a
g
e
1
D

g
l
T
e
x
I
m
a
g
e
2
D

glGetPointerv function

2/22/2020 • 2 minutes to read • Edit Online

The **glGetPointerv** function returns the address of a vertex data array.

Syntax

```
void WINAPI glGetPointerv(
   GLenum pname,
   GLvoid **params
);
```

Parameters

pname

The type of array pointer to return from the following symbolic constants: GL_COLOR_ARRAY_POINTER, GL_EDGE_FLAG_ARRAY_POINTER, GL_FEEDBACK_BUFFER_POINTER, GL_INDEX_ARRAY_POINTER, GL_NORMAL_ARRAY_POINTER, GL_TEXTURE_COORD_ARRAY_POINTER, GL_SELECTION_BUFFER_POINTER, and GL_VERTEX_ARRAY_POINTER.

params

Returns the value of the array pointer specified by *pname*.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*pname* was not an accepted value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*pname* was not an accepted value.

Remarks

The **glGetPointerv** function returns array pointer information. The *pname* parameter is a symbolic constant specifying the kind of array pointer to return, and *params* is a pointer to a location to place the returned data.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glArrayElem

e
n
t

g
l
C
o
l
o
r
P
o
i
n
t
e
r

g
l
D
r
a
w
A
r
r
a
y
s

g
l
E
d
g
e
F
l
a
g
P
o
i
n
t
e
r

g
l
G

e
t
S
t
r
i
n
g

g
l
I
n
d
e
x
P
o
i
n
t
e
r

g
l
N
o
r
m
a
l
P
o
i
n
t
e
r

g
l
T
e
x
C
o
o
r
d
P
o
i

n
t
e
r

g
l
V
e
r
t
e
x
P
o
i
n
t
e
r

glGetPolygonStipple function

2/22/2020 • 2 minutes to read • Edit Online

The **glGetPolygonStipple** function returns the polygon stipple pattern.

Syntax

```
void WINAPI glGetPolygonStipple(
   GLubyte *mask
);
```

Parameters

m *a* *s* *k*	Returns the stipple pattern.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glGetPolygonStipple** function returns a 32x32 polygon stipple pattern through the *mask* parameter. The pattern is packed into memory as if **glReadPixels** with both *height* and *width* of 32, *type* of GL_BITMAP, and *format* of GL_COLOR_INDEX were called, and the stipple pattern were stored in an internal 32x32 color-index buffer. Unlike **glReadPixels**, however, pixel-transfer operations (shift, offset, and pixel map) are not applied to the returned stipple image.

If an error is generated, no change is made to the contents of *mask*.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glPixelStore

glPixelTransfer

glPolygonStipple

g

I
R
e
a
d
P
i
x
e
l
s

glGetString function

2/22/2020 • 2 minutes to read • Edit Online

The **glGetString** function returns a string describing the current OpenGL connection.

Syntax

```
const GLubyte* WINAPI glGetString(
   GLenum name
);
```

Parameters

name

One of the following symbolic constants.

VALUE	MEANING
GL_VENDOR	Returns the company responsible for this OpenGL implementation. This name does not change from release to release.
GL_RENDERER	Returns the name of the renderer. This name is typically specific to a particular configuration of a hardware platform. It does not change from release to release.
GL_VERSION	Returns a version or release number.
GL_EXTENSIONS	Returns a space-separated list of supported extensions to OpenGL.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*name* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*name* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glGetString** function returns a pointer to a static string describing some aspect of the current OpenGL connection.

Because OpenGL does not include queries for the performance characteristics of an implementation, it is expected that some applications will be written to recognize known platforms and will modify their OpenGL usage based on known performance characteristics of these platforms. The strings GL_VENDOR and GL_RENDERER together uniquely specify a platform, and will not change from release to release. They should be used as such by platform recognition algorithms.

The format and contents of the string that **glGetString** returns depend on the implementation, except that:

- Extension names will not include space characters and will be separated by space characters in the GL_EXTENSIONS string.
- The GL_VERSION string begins with a version number. The version number uses one of these forms:

  major_number.minor_number

  major_number.minor_number.release_number
- Vendor-specific information may follow the version number. Its format depends on the implementation, but a space always separates the version number and the vendor-specific information.
- All strings are null-terminated.

If an error is generated, **glGetString** returns zero.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGetTexEnv Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions return texture environment parameters:

- glGetTexEnvfv
- glGetTexEnviv

glGetTexEnvfv function

2/22/2020 • 2 minutes to read • Edit Online

The **glGetTexEnvfv** and **glGetTexEnviv** functions return texture environment parameters.

Syntax

```
void WINAPI glGetTexEnvfv(
   GLenum  target,
   GLenum  pname,
   GLfloat *params
);
```

Parameters

target

A texture environment. Must be GL_TEXTURE_ENV.

pname

The symbolic name of a texture environment parameter. The following values are accepted.

VALUE	MEANING
GL_TEXTURE_ENV_MODE	The *params* parameter returns the single-valued texture environment mode, a symbolic constant.
GL_TEXTURE_ENV_COLOR	The *params* parameter returns four integer or floating-point values that are the texture environment color. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer, and -1.0 maps to the most negative representable integer.

params

Returns the requested data.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *pname* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *pname* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glGetTexEnv** function returns in *params* selected values of a texture environment that was specified with **glTexEnv**. The *target* parameter specifies a texture environment. Currently, only one texture environment is defined and supported: GL_TEXTURE_ENV.

The *pname* parameter names a specific texture environment parameter.

If an error is generated, no change is made to the contents of *params*.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib
DLL	Ope ngl3 2.dll

See also

glBegin

glEnd

glTexEnv

glGetTexEnviv function

2/22/2020 • 2 minutes to read • Edit Online

The **glGetTexEnvfv** and **glGetTexEnviv** functions return texture environment parameters.

Syntax

```
void WINAPI glGetTexEnviv(
   GLenum target,
   GLenum pname,
   GLint  *params
);
```

Parameters

target

A texture environment. Must be GL_TEXTURE_ENV.

pname

The symbolic name of a texture environment parameter. The following values are accepted.

VALUE	MEANING
GL_TEXTURE_ENV_MODE	The *params* parameter returns the single-valued texture environment mode, a symbolic constant.
GL_TEXTURE_ENV_COLOR	The *params* parameter returns four integer or floating-point values that are the texture environment color. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer, and -1.0 maps to the most negative representable integer.

params

Returns the requested data.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *pname* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *pname* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glGetTexEnv** function returns in *params* selected values of a texture environment that was specified with **glTexEnv**. The *target* parameter specifies a texture environment. Currently, only one texture environment is defined and supported: GL_TEXTURE_ENV.

The *pname* parameter names a specific texture environment parameter.

If an error is generated, no change is made to the contents of *params*.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib
DLL	Ope ngl3 2.dll

See also

glBegin

glEnd

glTexEnv

glGetTexGen Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions return texture coordinate generation parameters:

- **glGetTexGendv**
- **glGetTexGenfv**
- **glGetTexGeniv**

glGetTexGendv function

2/22/2020 • 2 minutes to read • Edit Online

The **glGetTexGendv**, **glGetTexGenfv**, and **glGetTexGeniv** functions return texture coordinate generation parameters.

Syntax

```
void WINAPI glGetTexGendv(
   GLenum   coord,
   GLenum   pname,
   GLdouble *params
);
```

Parameters

coord

A texture coordinate. Must be GL_S, GL_T, GL_R, or GL_Q.

pname

The symbolic name of the value(s) to be returned. Must be either GL_TEXTURE_GEN_MODE or the name of one of the texture generation plane equations: GL_OBJECT_PLANE or GL_EYE_PLANE. These values are as follows.

VALUE	MEANING
GL_TEXTURE_GEN_MODE	The *params* parameter returns the single-valued texture generation function, a symbolic constant.
GL_OBJECT_PLANE	The *params* parameter returns the four plane equation coefficients that specify object linear-coordinate generation. Integer values, when requested, are mapped directly from the internal floating-point representation.

VALUE	MEANING
GL_EYE_PLANE	The *params* parameter returns the four plane equation coefficients that specify eye linear-coordinate generation. Integer values, when requested, are mapped directly from the internal floating-point representation. The returned values are those maintained in eye coordinates. They are not equal to the values specified using **glTexGen**, unless the modelview matrix was identified at the time **glTexGen** was called.

p a r a m s Returns the requested data.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*coord* or *pname* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*coord* or *pname* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glGetTexGen** function returns in *params* selected parameters of a texture-coordinate generation function that you specified with **glTexGen**. The *coord* parameter names one of the (*s*, *t*, *r*, *q*) texture coordinates, using the symbolic constant GL_S, GL_T, GL_R, or GL_Q.

If an error is generated, no change is made to the contents of *params*.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glB

e
g
i
n

g
l
E
n
d

g
l
T
e
x
G
e
n

glGetTexGenfv function

2/22/2020 • 2 minutes to read • Edit Online

The **glGetTexGendv**, **glGetTexGenfv**, and **glGetTexGeniv** functions return texture coordinate generation parameters.

Syntax

```
void WINAPI glGetTexGenfv(
   GLenum  coord,
   GLenum  pname,
   GLfloat *params
);
```

Parameters

coord

A texture coordinate. Must be GL_S, GL_T, GL_R, or GL_Q.

pname

The symbolic name of the value(s) to be returned. Must be either GL_TEXTURE_GEN_MODE or the name of one of the texture generation plane equations: GL_OBJECT_PLANE or GL_EYE_PLANE. These values are as follows.

VALUE	MEANING
GL_TEXTURE_GEN_MODE	The *params* parameter returns the single-valued texture generation function, a symbolic constant.
GL_OBJECT_PLANE	The *params* parameter returns the four plane equation coefficients that specify object linear-coordinate generation. Integer values, when requested, are mapped directly from the internal floating-point representation.

VALUE	MEANING
GL_EYE_PLANE	The *params* parameter returns the four plane equation coefficients that specify eye linear-coordinate generation. Integer values, when requested, are mapped directly from the internal floating-point representation. The returned values are those maintained in eye coordinates. They are not equal to the values specified using **glTexGen**, unless the modelview matrix was identified at the time **glTexGen** was called.

p a r a m s Returns the requested data.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*coord* or *pname* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*coord* or *pname* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glGetTexGen** function returns in *params* selected parameters of a texture-coordinate generation function that you specified with **glTexGen**. The *coord* parameter names one of the (*s*, *t*, *r*, *q*) texture coordinates, using the symbolic constant GL_S, GL_T, GL_R, or GL_Q.

If an error is generated, no change is made to the contents of *params*.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glB

e
g
i
n

g
l
E
n
d

g
l
T
e
x
G
e
n

glGetTexGeniv function

2/22/2020 • 2 minutes to read • Edit Online

The **glGetTexGendv**, **glGetTexGenfv**, and **glGetTexGeniv** functions return texture coordinate generation parameters.

Syntax

```
void WINAPI glGetTexGeniv(
   GLenum coord,
   GLenum pname,
   GLint  *params
);
```

Parameters

coord

A texture coordinate. Must be GL_S, GL_T, GL_R, or GL_Q.

pname

The symbolic name of the value(s) to be returned. Must be either GL_TEXTURE_GEN_MODE or the name of one of the texture generation plane equations: GL_OBJECT_PLANE or GL_EYE_PLANE. These values are as follows.

VALUE	MEANING
GL_TEXTURE_GEN_MODE	The *params* parameter returns the single-valued texture generation function, a symbolic constant.
GL_OBJECT_PLANE	The *params* parameter returns the four plane equation coefficients that specify object linear-coordinate generation. Integer values, when requested, are mapped directly from the internal floating-point representation.

VALUE	MEANING
GL_EYE_PLANE	The *params* parameter returns the four plane equation coefficients that specify eye linear-coordinate generation. Integer values, when requested, are mapped directly from the internal floating-point representation. The returned values are those maintained in eye coordinates. They are not equal to the values specified using **glTexGen**, unless the modelview matrix was identified at the time **glTexGen** was called.

p a r a m s Returns the requested data.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*coord* or *pname* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*coord* or *pname* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glGetTexGen** function returns in *params* selected parameters of a texture-coordinate generation function that you specified with **glTexGen**. The *coord* parameter names one of the (*s*, *t*, *r*, *q*) texture coordinates, using the symbolic constant GL_S, GL_T, GL_R, or GL_Q.

If an error is generated, no change is made to the contents of *params*.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glB

e
g
i
n

g
l
E
n
d

g
l
T
e
x
G
e
n

glGetTexImage function

2/22/2020 • 3 minutes to read • Edit Online

The **glGetTexImage** function returns a texture image.

Syntax

```
void WINAPI glGetTexImage(
   GLenum target,
   GLint  level,
   GLenum format,
   GLenum type,
   GLvoid *pixels
);
```

Parameters

target

Specifies which texture is to be obtained. GL_TEXTURE_1D and GL_TEXTURE_2D are accepted.

level

The level-of-detail number of the desired image. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.

format

A pixel format for the returned data. The supported formats are GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_LUMINANCE, GL_BGR_EXT, GL_BGRA_EXT, and GL_LUMINANCE_ALPHA.

type

A pixel type for the returned data. The supported types are GL_UNSIGNED_BYTE, GL_BYTE,

GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, and GL_FLOAT.

pixels Returns the texture image. Should be a pointer to an array of the type specified by *type*.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target*, *format*, or *type* was not an accepted value.
GL_INVALID_VALUE	*level* is less than zero or greater than *log2* (*max*), where *max* is the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd** .

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target*, *format*, or *type* was not an accepted value.
GL_INVALID_VALUE	*level* is less than zero or greater than *log*2 (*max*), where *max* is the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd** .

Remarks

The **glGetTexImage** function returns a texture image into *pixels*. The *target* parameter specifies whether the desired texture image is one specified by **glTexImage1D**(GL_TEXTURE_1D) or by **glTexImage2D**(GL_TEXTURE_2D). The *level* parameter specifies the level-of-detail number of the desired image. The *format* and *type* parameters specify the format and type of the desired image array. For a description of the acceptable values for the *format* and *type* parameters, respectively, see **glTexImage1D** and **glDrawPixels**.

Operation of **glGetTexImage** is best understood by considering the selected internal four-component texture image to be an RGBA color buffer the size of the image. The semantics of **glGetTexImage** are then identical to those of **glReadPixels** called with the same *format* and *type*, with *x* and *y* set to zero, *width* set to the width of the texture image (including border if one was specified), and *height* set to one for 1-D images, or to the height of the texture image (including border, if one was specified) for 2-D images.

Because the internal texture image is an RGBA image, pixel formats GL_COLOR_INDEX, GL_STENCIL_INDEX, and GL_DEPTH_COMPONENT are not accepted, and pixel type GL_BITMAP is not accepted.

If the selected texture image does not contain four components, the following mappings are applied. Single-component textures are treated as RGBA buffers with red set to the single-component value, and green, blue, and alpha set to zero.

Two-component textures are treated as RGBA buffers, with red set to the value of component zero, alpha set to the value of component one, and green and blue set to zero. Finally, three-component textures are treated as RGBA buffers with red set to component zero, green set to component one, blue set to component two, and alpha set to zero.

To determine the required size of *pixels*, use **glGetTexLevelParameter** to ascertain the dimensions of the internal texture image, and then scale the required number of pixels by the storage required for each pixel,

based on *format* and *type*. Be sure to take the pixel-storage parameters into account, especially GL_PACK_ALIGNMENT.

If an error is generated, no change is made to the contents of *pixels*.

The following functions retrieve information related to **glGetTexImage**:

glGet with argument GL_PACK_ALIGNMENT and others

glGetTexLevelParameter with argument GL_TEXTURE_WIDTH

glGetTexLevelParameter with argument GL_TEXTURE_HEIGHT

glGetTexLevelParameter with argument GL_TEXTURE_BORDER

glGetTexLevelParameter with argument GL_TEXTURE_COMPONENTS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glDrawPi

x
e
l
s

g
l
E
n
d

g
l
G
e
t
T
e
x
L
e
v
e
l
P
a
r
a
m
e
t
e
r

g
l
R
e
a
d
P
i
x
e
l
s

g
l
T
e
x
l

m
a
g
e
1
D

g
l
T
e
x
l
m
a
g
e
2
D

glGetTexLevelParameter Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions return texture parameter values for a specific level of detail:

- **glGetTexLevelParameterfv**
- **glGetTexLevelParameteriv**

glGetTexLevelParameterfv function

2/22/2020 • 3 minutes to read • Edit Online

The **glGetTexLevelParameterfv** and **glGetTexLevelParameteriv** functions return texture parameter values for a specific level of detail.

Syntax

```
void WINAPI glGetTexLevelParameterfv(
   GLenum  target,
   GLint   level,
   GLenum  pname,
   GLfloat *params
);
```

Parameters

target

The symbolic name of the target texture: either GL_TEXTURE_1D, GL_TEXTURE_2D, GL_PROXY_TEXTURE_1D, or GL_PROXY_TEXTURE_2D.

level

The level-of-detail number of the desired image. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.

pname

The symbolic name of a texture parameter. The following parameter names are accepted.

VALUE	MEANING
GL_TEXTURE_WIDTH	The *params* parameter returns a single value containing the width of the texture image. This value includes the border of the texture image.

VALUE	MEANING
GL_TEXTURE_HEIGHT	The *params* parameter returns a single value containing the height of the texture image. This value includes the border of the texture image.
GL_TEXTURE_INTERNAL_FORMAT	The *params* parameter returns a single value which describes the texel format of the texture.
GL_TEXTURE_BORDER	The *params* parameter returns a single value: the width in pixels of the border of the texture image.
GL_TEXTURE_RED_SIZE	The internal storage resolution of the red component of a texel. The resolution chosen by the OpenGL will be a close match for the resolution requested by the user with the component argument of glTexImage1D or glTexImage2D.
GL_TEXTURE_GREEN_SIZE	The internal storage resolution of the green component of a texel. The resolution chosen by the OpenGL will be a close match for the resolution requested by the user with the component argument of glTexImage1D or glTexImage2D.
GL_TEXTURE_BLUE_SIZE	The internal storage resolution of the blue component of a texel. The resolution chosen by the OpenGL will be a close match for the resolution requested by the user with the component argument of glTexImage1D or glTexImage2D.

VALUE	MEANING
GL_TEXTURE_ALPHA_SIZE	The internal storage resolution of the alpha component of a texel. The resolution chosen by the OpenGL will be a close match for the resolution requested by the user with the component argument of glTexImage1D or glTexImage2D.
GL_TEXTURE_LUMINANCE_SIZE	The internal storage resolution of the luminance component of a texel. The resolution chosen by the OpenGL will be a close match for the resolution requested by the user with the component argument of glTexImage1D or glTexImage2D.
GL_TEXTURE_INTENSITY_SIZE	The internal storage resolution of the intensity component of a texel. The resolution chosen by the OpenGL will be a close match for the resolution requested by the user with the component argument of glTexImage1D or glTexImage2D.
GL_TEXTURE_COMPONENTS	The *params* parameter returns a single value: the number of components in the texture image.

p
a
r
a
m
s

Returns the requested data.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *pname* was not an accepted value.
GL_INVALID_VALUE	*level* is less than zero or greater than *log2*(max)*, where *max* is the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *pname* was not an accepted value.
GL_INVALID_VALUE	*level* is less than zero or greater than *log2*(max)*, where *max* is the returned value of GL_MAX_TEXTURE_SIZE.

NAME	MEANING
GL_ INV ALI D_ OP ERA TIO N	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glGetTexLevelParameter** function returns in *params* texture parameter values for a specific level-of-detail value, specified as *level*. The *target* parameter defines the target texture, either GL_TEXTURE_1D, GL_TEXTURE_2D, GL_PROXY_TEXTURE_1D, or GL_PROXY_TEXTURE_2D to specify one-dimensional or two-dimensional texturing. The *pname* parameter specifies the texture parameter whose value or values will be returned.

If an error is generated, no change is made to the contents of *params*.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib
DLL	Ope ngl3 2.dll

See also

g l B e g i n

g l

E
n
d

g
l
G
e
t
T
e
x
P
a
r
a
m
e
t
e
r

g
l
T
e
x
I
m
a
g
e
1
D

g
l
T
e
x
I
m
a
g
e
2
D

g
l
T
e
x

P
a
r
a
m
e
t
e
r

glGetTexLevelParameteriv function

2/22/2020 • 3 minutes to read • Edit Online

The **glGetTexLevelParameterfv** and **glGetTexLevelParameteriv** functions return texture parameter values for a specific level of detail.

Syntax

```
void WINAPI glGetTexLevelParameteriv(
   GLenum target,
   GLint  level,
   GLenum pname,
   GLint  *params
);
```

Parameters

target

The symbolic name of the target texture: either GL_TEXTURE_1D, GL_TEXTURE_2D, GL_PROXY_TEXTURE_1D, or GL_PROXY_TEXTURE_2D.

level

The level-of-detail number of the desired image. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.

pname

The symbolic name of a texture parameter. The following parameter names are accepted.

VALUE	MEANING
GL_TEXTURE_WIDTH	The *params* parameter returns a single value containing the width of the texture image. This value includes the border of the texture image.

VALUE	MEANING
GL_TEXTURE_HEIGHT	The *params* parameter returns a single value containing the height of the texture image. This value includes the border of the texture image.
GL_TEXTURE_INTERNAL_FORMAT	The *params* parameter returns a single value which describes the texel format of the texture.
GL_TEXTURE_BORDER	The *params* parameter returns a single value: the width in pixels of the border of the texture image.
GL_TEXTURE_RED_SIZE	The internal storage resolution of the red component of a texel. The resolution chosen by the OpenGL will be a close match for the resolution requested by the user with the component argument of glTexImage1D or glTexImage2D.
GL_TEXTURE_GREEN_SIZE	The internal storage resolution of the green component of a texel. The resolution chosen by the OpenGL will be a close match for the resolution requested by the user with the component argument of glTexImage1D or glTexImage2D.
GL_TEXTURE_BLUE_SIZE	The internal storage resolution of the blue component of a texel. The resolution chosen by the OpenGL will be a close match for the resolution requested by the user with the component argument of glTexImage1D or glTexImage2D.

VALUE	MEANING
GL_TEXTURE_ALPHA_SIZE	The internal storage resolution of the alpha component of a texel. The resolution chosen by the OpenGL will be a close match for the resolution requested by the user with the component argument of glTexImage1D or glTexImage2D.
GL_TEXTURE_LUMINANCE_SIZE	The internal storage resolution of the luminance component of a texel. The resolution chosen by the OpenGL will be a close match for the resolution requested by the user with the component argument of glTexImage1D or glTexImage2D.
GL_TEXTURE_INTENSITY_SIZE	The internal storage resolution of the intensity component of a texel. The resolution chosen by the OpenGL will be a close match for the resolution requested by the user with the component argument of glTexImage1D or glTexImage2D.
GL_TEXTURE_COMPONENTS	The *params* parameter returns a single value: the number of components in the texture image.

p
a
r
a
m
s

Returns the requested data.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *pname* was not an accepted value.
GL_INVALID_VALUE	*level* is less than zero or greater than *log*2*(max)*, where *max* is the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *pname* was not an accepted value.
GL_INVALID_VALUE	*level* is less than zero or greater than *log*2*(max)*, where *max* is the returned value of GL_MAX_TEXTURE_SIZE.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glGetTexLevelParameter** function returns in *params* texture parameter values for a specific level-of-detail value, specified as *level*. The *target* parameter defines the target texture, either GL_TEXTURE_1D, GL_TEXTURE_2D, GL_PROXY_TEXTURE_1D, or GL_PROXY_TEXTURE_2D to specify one-dimensional or two-dimensional texturing. The *pname* parameter specifies the texture parameter whose value or values will be returned.

If an error is generated, no change is made to the contents of *params*.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

E
n
d

g
l
G
e
t
T
e
x
P
a
r
a
m
e
t
e
r

g
l
T
e
x
I
m
a
g
e
1
D

g
l
T
e
x
I
m
a
g
e
2
D

g
l
T
e
x

P
a
r
a
m
e
t
e
r

glGetTexParameter Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions return texture parameter values:

- glGetTexParameterfv
- glGetTexParameteriv

glGetTexParameterfv function

2/22/2020 • 2 minutes to read • Edit Online

The **glGetTexParameterfv** and **glGetTexParameteriv** functions return texture parameter values.

Syntax

```
void WINAPI glGetTexParameterfv(
   GLenum  target,
   GLenum  pname,
   GLfloat *params
);
```

Parameters

target

The symbolic name of the target texture. GL_TEXTURE_1D and GL_TEXTURE_2D are accepted.

pname

The symbolic name of a texture parameter. The following values are accepted.

VALUE	MEANING
GL_TEXTURE_MAG_FILTER	Returns the single-valued texture magnification filter, a symbolic constant.
GL_TEXTURE_MIN_FILTER	Returns the single-valued texture minification filter, a symbolic constant.

VALUE	MEANING
GL_TEXTURE_WRAP_S	Returns the single-valued wrapping function for texture coordinate *s*, a symbolic constant.
GL_TEXTURE_WRAP_T	Returns the single-valued wrapping function for texture coordinate *t*, a symbolic constant.
GL_TEXTURE_BORDER_COLOR	Returns four integer or floating-point numbers that comprise the RGBA color of the texture border. Floating-point values are returned in the range [0,1]. Integer values are returned as a linear mapping of the internal floating-point representation such that 1.0 maps to the most positive representable integer and -1.0 maps to the most negative representable integer.
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**.

p a r a m s

Returns the texture parameters.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *name* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *name* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glGetTexParameter** function returns in *params* the value or values of the texture parameter specified as *pname*. The *target* parameter defines the target texture, either GL_TEXTURE_1D or GL_TEXTURE_2D, to specify one-

dimensional or two-dimensional texturing. The *pname* parameter accepts the same symbols as **glTexParameter**, with the same interpretations.

If an error is generated, no change is made to the contents of *params*.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glTexParamet

e

r

glGetTexParameteriv function

2/22/2020 • 2 minutes to read • Edit Online

The **glGetTexParameterfv** and **glGetTexParameteriv** functions return texture parameter values.

Syntax

```
void WINAPI glGetTexParameteriv(
   GLenum target,
   GLenum pname,
   GLint  *params
);
```

Parameters

target

The symbolic name of the target texture. GL_TEXTURE_1D and GL_TEXTURE_2D are accepted.

pname

The symbolic name of a texture parameter. The following values are accepted.

VALUE	MEANING
GL_TEXTURE_MAG_FILTER	Returns the single-valued texture magnification filter, a symbolic constant.
GL_TEXTURE_MIN_FILTER	Returns the single-valued texture minification filter, a symbolic constant.

VALUE	MEANING
GL_TEXTURE_WRAP_S	Returns the single-valued wrapping function for texture coordinate *s*, a symbolic constant.
GL_TEXTURE_WRAP_T	Returns the single-valued wrapping function for texture coordinate *t*, a symbolic constant.
GL_TEXTURE_BORDER_COLOR	Returns four integer or floating-point numbers that comprise the RGBA color of the texture border. Floating-point values are returned in the range [0,1]. Integer values are returned as a linear mapping of the internal floating-point representation such that 1.0 maps to the most positive representable integer and -1.0 maps to the most negative representable integer.
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**.

p
a
r
a
m
s

Returns the texture parameters.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *name* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *name* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glGetTexParameter** function returns in *params* the value or values of the texture parameter specified as *pname*. The *target* parameter defines the target texture, either GL_TEXTURE_1D or GL_TEXTURE_2D, to specify one-

dimensional or two-dimensional texturing. The *pname* parameter accepts the same symbols as glTexParameter, with the same interpretations.

If an error is generated, no change is made to the contents of *params*.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glTexParamet

e

r

glHint function

2/22/2020 • 2 minutes to read • Edit Online

The **glHint** function specifies implementation-specific hints.

Syntax

```
void WINAPI glHint(
   GLenum target,
   GLenum mode
);
```

Parameters

target

A symbolic constant indicating the behavior to be controlled. The following symbolic constants, along with suggested semantics, are accepted.

VALUE	MEANING
GL_FOG_HINT	Indicates the accuracy of fog calculation. If per-pixel fog calculation is not efficiently supported by the OpenGL implementation, hinting GL_DONT_CARE or GL_FASTEST can result in per-vertex calculation of fog effects.
GL_LINE_SMOOTH_HINT	Indicates the sampling quality of antialiased lines. Hinting GL_NICEST can result in more pixel fragments being generated during rasterization, if a larger filter function is applied.

VALUE	MEANING
GL_PERSPECTIVE_CORRECTION_HINT	Indicates the quality of color and texture coordinate interpolation. If perspective-corrected parameter interpolation is not efficiently supported by the OpenGL implementation, hinting GL_DONT_CARE or GL_FASTEST can result in simple linear interpolation of colors and/or texture coordinates.
GL_POINT_SMOOTH_HINT	Indicates the sampling quality of antialiased points. Hinting GL_NICEST can result in more pixel fragments being generated during rasterization, if a larger filter function is applied.
GL_POLYGON_SMOOTH_HINT	Indicates the sampling quality of antialiased polygons. Hinting GL_NICEST can result in more pixel fragments being generated during rasterization, if a larger filter function is applied.

mode

A symbolic constant indicating the desired behavior. The following symbolic constants are accepted.

VALUE	MEANING
GL_FASTEST	The most efficient option should be chosen.
GL_NICEST	The most correct, or highest quality, option should be chosen.

VALUE	MEANING
GL_ DO NT_ CA RE	The client doesn't have a preference.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_ INV ALI D_E NU M	*target* or *mode* was not an accepted value.
GL_ INV ALI D_ OP ERA TIO N	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_ INV ALI D_E NU M	*target* or *mode* was not an accepted value.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

When there is room for interpretation, you can control certain aspects of OpenGL behavior with hints. You specify a hint with two arguments. The *target* parameter is a symbolic constant indicating the behavior to be controlled, and *mode* is another symbolic constant indicating the desired behavior.

Though the implementation aspects that can be hinted are well defined, the interpretation of the hints depends on the implementation.

The **glHint** function can be ignored.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

g
l
E
n
d

glIndex Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions set the current color index:

- glIndexd
- glIndexf
- glIndexi
- glIndexs
- glIndexdv
- glIndexfv
- glIndexiv
- glIndexsv

glIndexd function

2/22/2020 • 2 minutes to read • Edit Online

The **glIndexd** function sets the current color index.

Syntax

```
void WINAPI glIndexd(
   GLdouble c
);
```

Parameters

c	The new value for the current color index.

Return value

This function does not return a value.

Remarks

The **glIndexd** function updates the current (single-valued) color index. It takes one argument: the new value for the current color index.

The current index is stored as a floating-point value. Integer values are converted directly to floating-point values, with no special mapping.

Index values outside the representable range of the color-index buffer are not clamped. However, before an index is dithered (if enabled) and written to the framebuffer, 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 framebuffer are masked out.

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

The following function retrieves information related to **glIndexd**:

glGet with argument GL_CURRENT_INDEX

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glEnd

glGet

glIndexdv function

2/22/2020 • 2 minutes to read • Edit Online

The **glIndexdv** function sets the current color index.

Syntax

```
void WINAPI glIndexdv(
   const GLdouble *c
);
```

Parameters

c	A pointer to a one-element array that contains the new value for the current color index.

Return value

This function does not return a value.

Remarks

The **glIndexdv** function updates the current (single-valued) color index. It takes one argument: the new value for the current color index.

The current index is stored as a floating-point value. Integer values are converted directly to floating-point values, with no special mapping.

Index values outside the representable range of the color-index buffer are not clamped. However, before an index is dithered (if enabled) and written to the framebuffer, 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 framebuffer are masked out.

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

The following function retrieves information related to **glIndexdv**:

glGet with argument GL_CURRENT_INDEX

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glEnd

glGet

glIndexf function

2/22/2020 • 2 minutes to read • Edit Online

The **glIndexf** function sets the current color index.

Syntax

```
void WINAPI glIndexf(
   GLfloat c
);
```

Parameters

c	The new value for the current color index.

Return value

This function does not return a value.

Remarks

The **glIndexf** function updates the current (single-valued) color index. It takes one argument: the new value for the current color index.

The current index is stored as a floating-point value. Integer values are converted directly to floating-point values, with no special mapping.

Index values outside the representable range of the color-index buffer are not clamped. However, before an index is dithered (if enabled) and written to the framebuffer, 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 framebuffer are masked out.

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

The following function retrieves information related to **glIndexf**:

glGet with argument GL_CURRENT_INDEX

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glEnd

glGet

glIndexfv function

2/22/2020 • 2 minutes to read • Edit Online

The **glIndexfv** function sets the current color index.

Syntax

```
void WINAPI glIndexfv(
   const GLfloat *c
);
```

Parameters

c	A pointer to a one-element array that contains the new value for the current color index.

Return value

This function does not return a value.

Remarks

The **glIndexfv** function updates the current (single-valued) color index. It takes one argument: the new value for the current color index.

The current index is stored as a floating-point value. Integer values are converted directly to floating-point values, with no special mapping.

Index values outside the representable range of the color-index buffer are not clamped. However, before an index is dithered (if enabled) and written to the framebuffer, 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 framebuffer are masked out.

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

The following function retrieves information related to **glIndexfv**:

glGet with argument GL_CURRENT_INDEX

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glEnd

glGet

glIndexi function

2/22/2020 • 2 minutes to read • Edit Online

The **glIndexi** function sets the current color index.

Syntax

```
void WINAPI glIndexi(
   GLint c
);
```

Parameters

c	The new value for the current color index.

Return value

This function does not return a value.

Remarks

The **glIndexi** function updates the current (single-valued) color index. It takes one argument: the new value for the current color index.

The current index is stored as a floating-point value. Integer values are converted directly to floating-point values, with no special mapping.

Index values outside the representable range of the color-index buffer are not clamped. However, before an index is dithered (if enabled) and written to the framebuffer, 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 framebuffer are masked out.

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

The following function retrieves information related to **glIndexi**:

glGet with argument GL_CURRENT_INDEX

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glEnd

glGet

glIndexiv function

2/22/2020 • 2 minutes to read • Edit Online

The **glIndexiv** function sets the current color index.

Syntax

```
void WINAPI glIndexiv(
   const GLint *c
);
```

Parameters

c	A pointer to a one-element array that contains the new value for the current color index.

Return value

This function does not return a value.

Remarks

The **glIndexiv** function updates the current (single-valued) color index. It takes one argument: the new value for the current color index.

The current index is stored as a floating-point value. Integer values are converted directly to floating-point values, with no special mapping.

Index values outside the representable range of the color-index buffer are not clamped. However, before an index is dithered (if enabled) and written to the framebuffer, 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 framebuffer are masked out.

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

The following function retrieves information related to **glIndexiv**:

glGet with argument GL_CURRENT_INDEX

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glEnd

glGet

glIndexs function

2/22/2020 • 2 minutes to read • Edit Online

The **glIndexs** function sets the current color index.

Syntax

```
void WINAPI glIndexs(
   GLshort c
);
```

Parameters

c	The new value for the current color index.

Return value

This function does not return a value.

Remarks

The **glIndexs** function updates the current (single-valued) color index. It takes one argument: the new value for the current color index.

The current index is stored as a floating-point value. Integer values are converted directly to floating-point values, with no special mapping.

Index values outside the representable range of the color-index buffer are not clamped. However, before an index is dithered (if enabled) and written to the framebuffer, 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 framebuffer are masked out.

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

The following function retrieves information related to **glIndexs**:

glGet with argument GL_CURRENT_INDEX

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glEnd

glGet

glIndexsv function

2/22/2020 • 2 minutes to read • Edit Online

The **glIndexsv** function sets the current color index.

Syntax

```
void WINAPI glIndexsv(
   const GLshort *c
);
```

Parameters

c	A pointer to a one-element array that contains the new value for the current color index.

Return value

This function does not return a value.

Remarks

The **glIndexsv** function updates the current (single-valued) color index. It takes one argument: the new value for the current color index.

The current index is stored as a floating-point value. Integer values are converted directly to floating-point values, with no special mapping.

Index values outside the representable range of the color-index buffer are not clamped. However, before an index is dithered (if enabled) and written to the framebuffer, 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 framebuffer are masked out.

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

The following function retrieves information related to **glIndexsv**:

glGet with argument GL_CURRENT_INDEX

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glEnd

glGet

glIndexMask function

2/22/2020 • 2 minutes to read • Edit Online

The **glIndexMask** function controls the writing of individual bits in the color-index buffers.

Syntax

```
void WINAPI glIndexMask(
   GLuint mask
);
```

Parameters

m
a
s
k

A bit mask to enable and disable the writing of individual bits in the color-index buffers. Initially, the mask is all ones.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glIndexMask** function controls the writing of individual bits in the color-index buffers. The least significant *n* bits of *mask*, where *1* is the number of bits in a color-index buffer, specify a mask. Wherever a one appears in the mask, the corresponding bit in the color-index buffer (or buffers) is made writable. Where a zero appears, the bit is write-protected.

This mask is used only in color-index mode, and it affects only the buffers currently selected for writing (see **glDrawBuffer**). Initially, all bits are enabled for writing.

The following function retrieves information related to **glIndexMask**:

glGet with argument GL_INDEX_WRITEMASK

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

i
n

g
l
D
e
p
t
h
M
a
s
k

g
l
D
r
a
w
B
u
f
f
e
r

g
l
E
n
d

g
l
I
n
d
e
x

g
l
S
t
e
n
c
i
l
M
a

s
k

glIndexPointer function

2/22/2020 • 2 minutes to read • Edit Online

The **glIndexPointer** function defines an array of color indexes.

Syntax

```
void WINAPI glIndexPointer(
         GLenum  type,
         GLsizei stride,
   const GLvoid  *pointer
);
```

Parameters

type

The data type of each color index in the array using the following symbolic constants: GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE.

stride

The byte offset between consecutive color indexes. When *stride* is zero, the color indexes are tightly packed in the array.

pointer

A pointer to the first color index in the array.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*type* was not an accepted value.
GL_INVALID_VALUE	*stride* or *count* was negative.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*type* was not an accepted value.
GL_INVALID_VALUE	*stride* or *count* was negative.

Remarks

The **glIndexPointer** function specifies the location and data of an array of color indexes to use when rendering. The *type* parameter specifies the data type of each color index and *stride* determines the byte offset from one color index to the next, enabling the packing of vertices and attributes in a single array or storage in separate arrays. In some implementations, storing the vertices and attributes in a single array can be more efficient than using separate arrays. For more information, see **glInterleavedArrays**.

A color-index array is enabled when you specify the GL_INDEX_ARRAY constant with **glEnableClientState**. When enabled, **glDrawArrays** and **glArrayElement** use the color-index array. By default the color-index array is disabled.

You cannot include **glIndexPointer** in display lists.

When you specify a color-index array using **glIndexPointer**, the values of all the function's color-index array parameters are saved in a client-side state and static array elements can be cached. Because the color-

index array parameters are client-side state, their values are not saved or restored by **glPushAttrib** and **glPopAttrib**.

Although no error is generated when you call **glIndexPointer** within **glBegin** and **glEnd** pairs, the results are undefined.

The following functions retrieve information related to **glIndexPointer**:

glIsEnabled with argument GL_INDEX_ARRAY

glGet with argument GL_INDEX_ARRAY_STRIDE

glGet with argument GL_INDEX_ARRAY_COUNT

glGet with argument GL_INDEX_ARRAY_TYPE

glGet with argument GL_INDEX_ARRAY_SIZE

glGetPointerv with argument GL_INDEX_ARRAY_POINTER

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glArrayElement

t

g
l
C
o
l
o
r
P
o
i
n
t
e
r

g
l
D
r
a
w
A
r
r
a
y
s

g
l
E
d
g
e
F
l
a
g
P
o
i
n
t
e
r

g
l
G
e
t

P
o
i
n
t
e
r
v

g
l
G
e
t
S
t
r
i
n
g

g
l
N
o
r
m
a
l
P
o
i
n
t
e
r

g
l
P
u
s
h
A
t
t
r
i
b

g
l
T

e
x
C
o
o
r
d
P
o
i
n
t
e
r

g
l
V
e
r
t
e
x
P
o
i
n
t
e
r

glInitNames function

2/22/2020 • 2 minutes to read • Edit Online

The **glInitNames** function initializes the name stack.

Syntax

```
void WINAPI glInitNames(void);
```

Parameters

This function has no parameters.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glInitNames** function causes the name stack to be initialized to its default empty state. The name stack is used during selection mode to allow sets of rendering commands to be uniquely identified. It consists of an ordered set of unsigned integers.

The name stack is always empty while the render mode is not GL_SELECT. Calls to **glInitNames** while the render mode is not GL_SELECT are ignored.

The following functions retrieve information related to **glInitNames**:

glGet with argument GL_NAME_STACK_DEPTH

glGet with argument GL_MAX_NAME_STACK_DEPTH

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glLoa

d
N
a
m
e

g
l
P
u
s
h
N
a
m
e

g
l
R
e
n
d
e
r
M
o
d
e

g
l
S
e
l
e
c
t
B
u
f
f
e
r

glInterleavedArrays function

2/22/2020 • 2 minutes to read • Edit Online

The **glInterleavedArrays** function simultaneously specifies and enables several interleaved arrays in a larger aggregate array.

Syntax

```
void WINAPI glInterleavedArrays(
         GLenum  format,
         GLsizei stride,
   const GLvoid  *pointer
);
```

Parameters

format

The type of array to enable. The parameter can assume one of the following symbolic values: GL_V2F, GL_V3F, GL_C4UB_V2F, GL_C4UB_V3F, GL_C3F_V3F, GL_N3F_V3F, GL_C4F_N3F_V3F, GL_T2F_V3F, GL_T4F_V4F, GL_T2F_C4UB_V3F, GL_T2F_C3F_V3F, GL_T2F_N3F_V3F, GL_T2F_C4F_N3F_V3F, or GL_T4F_C4F_N3F_V4F.

stride

The offset in bytes between each aggregate array element.

pointer

A pointer to the first element of an aggregate array.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*format* was not an accepted value.
GL_INVALID_VALUE	*stride* was a negative value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*format* was not an accepted value.
GL_INVALID_VALUE	*stride* was a negative value.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Remarks

With the **glInterleavedArrays** function, you can simultaneously specify and enable several interleaved color, normal, texture, and vertex arrays whose elements are part of a larger aggregate array element. For some memory architectures, this is more efficient than specifying the arrays separately.

If the *stride* parameter is zero then the aggregate array elements are stored consecutively; otherwise *stride* bytes occur between aggregate array elements.

The *format* parameter serves as a key that describes how to extract individual arrays from the aggregate array:

- If *format* contains a T, then texture coordinates are extracted from the interleaved array.
- If C is present, color values are extracted.
- If N is present, normal coordinates are extracted.
- Vertex coordinates are always extracted.
- The digits 2, 3, and 4 denote how many values are extracted.
- F indicates that values are extracted as floating point values.
- If 4UB follows the C, colors may also be extracted as 4 unsigned bytes. If a color is extracted as 4 unsigned bytes, the vertex array element that follows is located at the first possible floating-point aligned address.

If you call **glInterleavedArrays** while compiling a display list, it is not compiled into the list but is executed immediately.

You cannot include calls to **glInterleavedArrays** in **glDisableClientState** between calls to glBegin and the corresponding call to **glEnd**.

> **NOTE**
> The **glInterleavedArrays** function is only available in OpenGL version 1.1 or later.

The **glInterleavedArrays** function is implemented on the client side with no protocol. Because the vertex array parameters are client-side state, they are not saved or restored by glPushAttrib and **glPopAttrib**. Use glPushClientAttrib and **glPopClientAttrib** instead.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glArrayElement

glColorPointer

glDraw

A
r
r
a
y
s

g
l
D
r
a
w
E
l
e
m
e
n
t
s

g
l
E
d
g
e
F
l
a
g
P
o
i
n
t
e
r

g
l
E
n
a
b
l
e
C
l
i
e
n

t
S
t
a
t
e

g
l
G
e
t
P
o
i
n
t
e
r
v

g
l
I
n
d
e
x
P
o
i
n
t
e
r

g
l
N
o
r
m
a
l
P
o
i
n
t
e
r

g

l
P
u
s
h
A
t
t
r
i
b

g
l
P
u
s
h
C
l
i
e
n
t
A
t
t
r
i
b

g
l
T
e
x
C
o
o
r
d
P
o
i
n
t
e
r

g
l
V
e

r
t
e
x
P
o
i
n
t
e
r

glIsEnabled function

2/22/2020 • 2 minutes to read

• Edit Online

The **glIsEnabled** function tests whether a capability is enabled.

Syntax

```
GLboolean WINAPI glIsEnabled(
   GLenum cap
);
```

Parameters

c
a
p

A symbolic constant indicating an OpenGL capability. The following capabilities are accepted.

VALUE	MEANING
GL_ALPHA_TEST	See glAlphaFunc
GL_AUTO_NORMAL	See glEvalCoord
GL_BLEND	See glBlendFunc

VALUE	MEANING
GL_CLIP_PLANE *i*	See glClipPlane
GL_COLOR_ARRAY	See glColorPointer
GL_COLOR_LOGIC_OP	See glLogicOp
GL_COLOR_MATERIAL	See glColorMaterial
GL_CULL_FACE	See glCullFace
GL_DEPTH_TEST	See glDepthFunc and glDepthRange
GL_DITHER	See glEnable
GL_FOG	See glFog

VALUE	MEANING
GL_INDEX_ARRAY	See glIndexPointer
GL_INDEX_LOGIC_OP	See glLogicOp
GL_LIGHT *i*	See glLightModel and glLight
GL_LIGHTING	See glMaterial, glLightModel, and glLight
GL_LINE_SMOOTH	See glLineWidth
GL_LINE_STIPPLE	See glLineStipple
GL_MAP1_COLOR_4	See glMap1
GL_MAP1_INDEX	See glMap1

VALUE	MEANING
GL_MAP1_NORMAL	See glMap1
GL_MAP1_TEXTURE_COORD_1	See glMap1
GL_MAP1_TEXTURE_COORD_2	See glMap1
GL_MAP1_TEXTURE_COORD_3	See glMap1
GL_MAP1_TEXTURE_COORD_4	See glMap1

VALUE	MEANING
GL_MAP1_VERTEX_3	See glMap1
GL_MAP1_VERTEX_4	See glMap1
GL_MAP2_COLOR_4	See glMap2
GL_MAP2_INDEX	See glMap2
GL_MAP2_NORMAL	See glMap2
GL_MAP2_TEXTURE_COORD_1	See glMap2

VALUE	MEANING
GL_MAP2_TEXTURE_COORD_2	See glMap2
GL_MAP2_TEXTURE_COORD_3	See glMap2
GL_MAP2_TEXTURE_COORD_4	See glMap2
GL_MAP2_VERTEX_3	See glMap2
GL_MAP2_VERTEX_4	See glMap2
GL_NORMAL_ARRAY	See glNormalPointer

VALUE	MEANING
GL_NORMALIZE	See glNormal
GL_POINT_SMOOTH	See glPointSize
GL_POLYGON_OFFSET_FILL	See glPolygonOffset
GL_POLYGON_OFFSET_LINE	See glPolygonOffset
GL_POLYGON_OFFSET_POINT	See glPolygonOffset
GL_POLYGON_SMOOTH	See glPolygonMode

VALUE	MEANING
GL_POLYGON_STIPPLE	See glPolygonStipple
GL_SCISSOR_TEST	See glScissor
GL_STENCIL_TEST	See glStencilFunc and glStencilOp
GL_TEXTURE_1D	See glTexImage1D
GL_TEXTURE_2D	See glTexImage2D
GL_TEXTURE_COORD_ARRAY	See glTexCoordPointer
GL_TEXTURE_GEN_Q	See glTexGen

VALUE	MEANING
GL_TEXTURE_GEN_R	See glTexGen
GL_TEXTURE_GEN_S	See glTexGen
GL_TEXTURE_GEN_T	See glTexGen
GL_VERTEX_ARRAY	See glVertexPointer

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*cap* was not an accepted value.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*cap* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Remarks

The **glIsEnabled** function returns GL_TRUE if *cap* is an enabled capability and returns GL_FALSE otherwise.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnable

glEnd

glIsList function

2/22/2020 • 2 minutes to read • Edit Online

The **glIsList** function tests for display list existence.

Syntax

```
GLboolean WINAPI glIsList(
   GLuint list
);
```

Parameters

list	A potential display list name.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glIsList** function returns GL_TRUE if *list* is the name of a display list and returns GL_FALSE otherwise.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glCallList

glCall

i
s
t
s

g
l
D
e
l
e
t
e
L
i
s
t
s

g
l
E
n
d

g
l
G
e
n
L
i
s
t
s

g
l
N
e
w
L
i
s
t

glIsTexture function

2/22/2020 • 2 minutes to read • Edit Online

The **glIsTexture** function determines if a name corresponds to a texture.

Syntax

```
GLboolean WINAPI glIsTexture(
   GLuint texture
);
```

Parameters

texture	A value that is the name of a texture.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

If the *texture* parameter is currently the name of a texture, the **glIsTexture** function returns GL_TRUE. The **glIsTexture** function returns GL_FALSE if *texture* is zero. It also returns GL_FALSE if it is a non-zero value that is not currently the name of a texture, or if an error occurs.

You cannot include calls to **glIsTexture** in display lists.

> **NOTE**
> The **glIsTexture** function is only available in OpenGL version 1.1 or later.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

glBindTexture

glEnd

glGenTextures

glGet

glGetTexPar

a
m
e
t
e
r

g
l
T
e
x
I
m
a
g
e
1
D

g
l
T
e
x
I
m
a
g
e
2
D

g
l
T
e
x
P
a
r
a
m
e
t
e
r

glLight Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions return light source parameter values:

- glLightf
- glLighti
- glLightfv
- glLightiv

glLightf function

2/22/2020 • 3 minutes to read • Edit Online

The **glLightf** function returns light source parameter values.

Syntax

```
void WINAPI glLightf(
   GLenum  light,
   GLenum  pname,
   GLfloat param
);
```

Parameters

light

The identifier of a light. 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* is a value: 0 to GL_MAX_LIGHTS - 1.

pname

A single-valued light source parameter for *light*. The following symbolic names are accepted.

VALUE	MEANING
GL_SPOT_EXPONENT	The *param* parameter is a single floating-point value that specifies the intensity distribution of the light. 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. The default spot exponent is 0, resulting in uniform light distribution.

VALUE	MEANING
GL_SPOT_CUTOFF	The *param* parameter is a single floating-point value that specifies the maximum spread angle of a light source. Floating-point values are mapped directly. Only values in the range [0, 90], and the special value 180, are accepted. If the angle between the direction of the light and the direction from the light to the vertex being lighted is greater than the spot cutoff angle, then the light is completely masked. Otherwise, its intensity is controlled by the spot exponent and the attenuation factors. The default spot cutoff is 180, resulting in uniform light distribution.
GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION, GL_QUADRATIC_ATTENUATION	The *param* parameter is a single floating-point value that specifies one of the three light attenuation factors. Floating-point values are mapped directly. Only nonnegative values are accepted. If the light is positional, rather than directional, its intensity is attenuated by the reciprocal of the sum of: the constant factor, the linear factor multiplied by the distance between the light and the vertex being lighted, and the quadratic factor multiplied by the square of the same distance. The default attenuation factors are (1,0,0), resulting in no attenuation.

param Specifies the value that parameter *pname* of light source *light* will be set to.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*light* or *pname* was not an accepted value.
GL_INVALID_VALUE	A spot exponent value was specified outside the range [0, 128], or spot cutoff was specified outside the range [0, 90] (except for the special value 180), or a negative attenuation factor was specified.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*light* or *pname* was not an accepted value.
GL_INVALID_VALUE	A spot exponent value was specified outside the range [0, 128], or spot cutoff was specified outside the range [0, 90] (except for the special value 180), or a negative attenuation factor was specified.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glLightf** function sets the value or values of individual light source parameters. The *light* parameter names the light and is a symbolic name of the form GL_LIGHT*i*, where 0 = *i* < GL_MAX_LIGHTS.

The *pname* parameter specifies one of the light source parameters, again by symbolic name. The *param* parameter is either a single value or a pointer to an array that contains the new values.

Lighting calculation is enabled and disabled using **glEnable** and **glDisable** with argument GL_LIGHTING. When lighting is enabled, light sources that are enabled contribute to the lighting calculation. Light source *i* is enabled and disabled using **glEnable** and **glDisable** with argument GL_LIGHT*i*.

It is always the case that GL_LIGHT*i* = GL_LIGHT0 + *i*.

The following functions retrieve information related to the **glLightf** function:

glGetLight

glIsEnabled with argument GL_LIGHTING

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

g

glBegin

glColorMaterial

glEnd

glLightModel

glMaterial

glLightfv function

2/22/2020 • 5 minutes to read • Edit Online

The **glLightfv** function returns light source parameter values.

Syntax

```
void WINAPI glLightfv(
         GLenum  light,
         GLenum  pname,
   const GLfloat *params
);
```

Parameters

light

The identifier of a light. 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* is a value: 0 to GL_MAX_LIGHTS - 1.

pname

A light source parameter for *light*. The following symbolic names are accepted.

VALUE	MEANING
GL_AMBIENT	The *params* parameter contains four floating-point values that specify the ambient RGBA intensity of the light. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default ambient light intensity is (0.0, 0.0, 0.0, 1.0).
GL_DIFFUSE	The *params* parameter contains four floating-point values that specify the diffuse RGBA intensity of the light. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default diffuse intensity is (0.0, 0.0, 0.0, 1.0) for all lights other than light zero. The default diffuse intensity of light zero is (1.0, 1.0, 1.0, 1.0).

VALUE	MEANING
GL_SPECULAR	The *params* parameter contains four floating-point values that specify the specular RGBA intensity of the light. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default specular intensity is (0.0, 0.0, 0.0, 1.0) for all lights other than light zero. The default specular intensity of light zero is (1.0, 1.0, 1.0, 1.0).
GL_POSITION	The *params* parameter contains four 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 **glLightfv** is called (just as if it were a point), and it is stored in eye coordinates. If the *w* component of the position is 0.0, the light is treated as a directional source. Diffuse and specular lighting calculations take the 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 default position is (0,0,1,0); thus, the default light source is directional, parallel to, and in the direction of the -*z* axis.
GL_SPOT_DIRECTION	The *params* parameter contains three floating-point values that specify the direction of the light in homogeneous object coordinates. Both integer and floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The spot direction is transformed by the inverse of the modelview matrix when **glLightfv** is called (just as if it were a normal), and it is stored in eye coordinates. It is significant only when GL_SPOT_CUTOFF is not 180, which it is by default. The default direction is (0,0,1).

VALUE	MEANING
GL_SPOT_EXPONENT	The *params* parameter is a single 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. The default spot exponent is 0, resulting in uniform light distribution.
GL_SPOT_CUTOFF	The *params* parameter is a single floating-point value that specifies the maximum spread angle of a light source. Integer and floating-point values are mapped directly. Only values in the range [0, 90], and the special value 180, are accepted. If the angle between the direction of the light and the direction from the light to the vertex being lighted is greater than the spot cutoff angle, then the light is completely masked. Otherwise, its intensity is controlled by the spot exponent and the attenuation factors. The default spot cutoff is 180, resulting in uniform light distribution.

VALUE	MEANING
GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION, GL_QUADRATIC_ATTENUATION	The *params* parameter is a single floating-point value that specifies one of the three light attenuation factors. Integer and floating-point values are mapped directly. Only nonnegative values are accepted. If the light is positional, rather than directional, its intensity is attenuated by the reciprocal of the sum of: the constant factor, the linear factor multiplied by the distance between the light and the vertex being lighted, and the quadratic factor multiplied by the square of the same distance. The default attenuation factors are (1,0,0), resulting in no attenuation.

p
a
r
a
m
s

Specifies the value that parameter *pname* of light source *light* will be set to.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*light* or *pname* was not an accepted value.

NAME	MEANING
GL_INVALID_VALUE	A spot exponent value was specified outside the range [0, 128], or spot cutoff was specified outside the range [0, 90] (except for the special value 180), or a negative attenuation factor was specified.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*light* or *pname* was not an accepted value.
GL_INVALID_VALUE	A spot exponent value was specified outside the range [0, 128], or spot cutoff was specified outside the range [0, 90] (except for the special value 180), or a negative attenuation factor was specified.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glLightfv** function sets the value or values of individual light source parameters. The *light* parameter names the light and is a symbolic name of the form GL_LIGHT*i*, where 0 = *i* < GL_MAX_LIGHTS.

The *pname* parameter specifies one of the light source parameters, again by symbolic name. The *params* parameter is either a single value or a pointer to an array that contains the new values.

Lighting calculation is enabled and disabled using **glEnable** and **glDisable** with argument GL_LIGHTING. When lighting is enabled, light sources that are enabled contribute to the lighting calculation. Light source *i* is enabled and disabled using **glEnable** and **glDisable** with argument GL_LIGHT*i*.

It is always the case that GL_LIGHT*i* = GL_LIGHT0 + *i*.

The following functions retrieve information related to the **glLightfv** function:

glGetLight

glIsEnabled with argument GL_LIGHTING

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColorMa

t
e
r
i
a
l

g
l
E
n
d

g
l
L
i
g
h
t
M
o
d
e
l

g
l
M
a
t
e
r
i
a
l

glLighti function

2/22/2020 • 3 minutes to read • Edit Online

The **glLighti** function returns light source parameter values.

Syntax

```
void WINAPI glLighti(
   GLenum light,
   GLenum pname,
   GLint  param
);
```

Parameters

light

The identifier of a light. 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* is a value: 0 to GL_MAX_LIGHTS - 1.

pname

A single-valued light source parameter for *light*. The following symbolic names are accepted.

VALUE	MEANING
GL_SPOT_EXPONENT	The *param* parameter is a single integer 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. The default spot exponent is 0, resulting in uniform light distribution.

VALUE	MEANING
GL_SPOT_CUTOFF	The *param* parameter is a single integer value that specifies the maximum spread angle of a light source. Integer and floating-point values are mapped directly. Only values in the range [0, 90], and the special value 180, are accepted. If the angle between the direction of the light and the direction from the light to the vertex being lighted is greater than the spot cutoff angle, then the light is completely masked. Otherwise, its intensity is controlled by the spot exponent and the attenuation factors. The default spot cutoff is 180, resulting in uniform light distribution.
GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION, GL_QUADRATIC_ATTENUATION	The *param* parameter is a single integer value that specifies one of the three light attenuation factors. Integer and floating-point values are mapped directly. Only nonnegative values are accepted. If the light is positional, rather than directional, its intensity is attenuated by the reciprocal of the sum of: the constant factor, the linear factor multiplied by the distance between the light and the vertex being lighted, and the quadratic factor multiplied by the square of the same distance. The default attenuation factors are (1,0,0), resulting in no attenuation.

param Specifies the value that parameter *pname* of light source *light* will be set to.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*light* or *pname* was not an accepted value.
GL_INVALID_VALUE	A spot exponent value was specified outside the range [0, 128], or spot cutoff was specified outside the range [0, 90] (except for the special value 180), or a negative attenuation factor was specified.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*light* or *pname* was not an accepted value.
GL_INVALID_VALUE	A spot exponent value was specified outside the range [0, 128], or spot cutoff was specified outside the range [0, 90] (except for the special value 180), or a negative attenuation factor was specified.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glLighti** function sets the value or values of individual light source parameters. The *light* parameter names the light and is a symbolic name of the form GL_LIGHT*i*, where 0 = *i* < GL_MAX_LIGHTS.

The *pname* parameter specifies one of the light source parameters, again by symbolic name. The *param* parameter is either a single value or a pointer to an array that contains the new values.

Lighting calculation is enabled and disabled using **glEnable** and **glDisable** with argument GL_LIGHTING. When lighting is enabled, light sources that are enabled contribute to the lighting calculation. Light source *i* is enabled and disabled using **glEnable** and **glDisable** with argument GL_LIGHT*i*.

It is always the case that GL_LIGHT*i* = GL_LIGHT0 + *i*.

The following functions retrieve information related to the **glLighti** function:

glGetLight

glIsEnabled with argument GL_LIGHTING

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

g

glBegin

glColorMaterial

glEnd

glLightModel

glMaterial

glLightiv function

2/22/2020 • 5 minutes to read • Edit Online

The **glLightiv** function returns light source parameter values.

Syntax

```
void WINAPI glLightiv(
         GLenum light,
         GLenum pname,
   const GLint  *params
);
```

Parameters

light

The identifier of a light. 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* is a value: 0 to GL_MAX_LIGHTS - 1.

pname

A light source parameter for *light*. The following symbolic names are accepted.

VALUE	MEANING
GL_AMBIENT	The *params* parameter contains four integer values that specify the ambient RGBA intensity of the light. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default ambient light intensity is (0.0, 0.0, 0.0, 1.0).

VALUE	MEANING
GL_DIFFUSE	The *params* parameter contains four integer values that specify the diffuse RGBA intensity of the light. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default diffuse intensity is (0.0, 0.0, 0.0, 1.0) for all lights other than light zero. The default diffuse intensity of light zero is (1.0, 1.0, 1.0, 1.0).
GL_SPECULAR	The *params* parameter contains four integer values that specify the specular RGBA intensity of the light. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to 1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default specular intensity is (0.0, 0.0, 0.0, 1.0) for all lights other than light zero. The default specular intensity of light zero is (1.0, 1.0, 1.0, 1.0).
GL_POSITION	The *params* parameter contains four integer 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 **glLightiv** is called (just as if it were a point), and it is stored in eye coordinates. If the *w* component of the position is 0.0, the light is treated as a directional source. Diffuse and specular lighting calculations take the 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 default position is (0,0,1,0); thus, the default light source is directional, parallel to, and in the direction of the -*z* axis.

VALUE	MEANING
GL_SPOT_DIRECTION	The *params* parameter contains three integer values that specify the direction of the light in homogeneous object coordinates. Both integer and floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The spot direction is transformed by the inverse of the modelview matrix when **glLightiv** is called (just as if it were a normal), and it is stored in eye coordinates. It is significant only when GL_SPOT_CUTOFF is not 180, which it is by default. The default direction is (0,0,1).
GL_SPOT_EXPONENT	The *params* parameter is a single integer 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. The default spot exponent is 0, resulting in uniform light distribution.
GL_SPOT_CUTOFF	The *params* parameter is a single integer value that specifies the maximum spread angle of a light source. Integer and floating-point values are mapped directly. Only values in the range [0, 90], and the special value 180, are accepted. If the angle between the direction of the light and the direction from the light to the vertex being lighted is greater than the spot cutoff angle, then the light is completely masked. Otherwise, its intensity is controlled by the spot exponent and the attenuation factors. The default spot cutoff is 180, resulting in uniform light distribution.

VALUE	MEANING
GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION, GL_QUADRATIC_ATTENUATION	The *params* parameter is a single integer value that specifies one of the three light attenuation factors. Integer and floating-point values are mapped directly. Only nonnegative values are accepted. If the light is positional, rather than directional, its intensity is attenuated by the reciprocal of the sum of: the constant factor, the linear factor multiplied by the distance between the light and the vertex being lighted, and the quadratic factor multiplied by the square of the same distance. The default attenuation factors are (1,0,0), resulting in no attenuation.

p
a
r
a
m
s

Specifies the value that parameter *pname* of light source *light* will be set to.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*light* or *pname* was not an accepted value.

NAME	MEANING
GL_INVALID_VALUE	A spot exponent value was specified outside the range [0, 128], or spot cutoff was specified outside the range [0, 90] (except for the special value 180), or a negative attenuation factor was specified.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*light* or *pname* was not an accepted value.
GL_INVALID_VALUE	A spot exponent value was specified outside the range [0, 128], or spot cutoff was specified outside the range [0, 90] (except for the special value 180), or a negative attenuation factor was specified.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glLightiv** function sets the value or values of individual light source parameters. The *light* parameter names the light and is a symbolic name of the form GL_LIGHT*i*, where 0 = *i* < GL_MAX_LIGHTS.

The *pname* parameter specifies one of the light source parameters, again by symbolic name. The *params* parameter is either a single value or a pointer to an array that contains the new values.

Lighting calculation is enabled and disabled using **glEnable** and **glDisable** with argument GL_LIGHTING. When lighting is enabled, light sources that are enabled contribute to the lighting calculation. Light source *i* is enabled and disabled using **glEnable** and **glDisable** with argument GL_LIGHT*i*.

It is always the case that GL_LIGHT*i* = GL_LIGHT0 + *i*.

The following functions retrieve information related to the **glLightiv** function:

glGetLight

glIsEnabled with argument GL_LIGHTING

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColorMa

t
e
r
i
a
l

g
l
E
n
d

g
l
L
i
g
h
t
M
o
d
e
l

g
l
M
a
t
e
r
i
a
l

glLightModel Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions set lighting model parameters:

- **glLightModelf**
- **glLightModelfv**
- **glLightModeli**
- **glLightModeliv**

glLightModelf function

2/22/2020 • 3 minutes to read • Edit Online

The **glLightModelf** function sets lighting model parameters.

Syntax

```
void WINAPI glLightModelf(
   GLenum  pname,
   GLfloat *param
);
```

Parameters

pname

A single-valued lighting model parameter. The following values are accepted.

VALUE	MEANING
GL_LIGHT_MODEL_LOCAL_VIEWER	The *param* parameter is a single floating-point value that specifies how specular reflection angles are computed. If *param* is 0 (or 0.0), specular reflection angles take the view direction to be parallel to and in the direction of the -*z* axis, regardless of the location of the vertex in eye coordinates. Otherwise, specular reflections are computed from the origin of the eye coordinate system. The default is 0.
GL_LIGHT_MODEL_TWO_SIDE	The *param* parameter is a single floating-point value that specifies whether one-sided or two-sided lighting calculations are done for polygons. It has no effect on the lighting calculations for points, lines, or bitmaps. If *param* is 0 (or 0.0), one-sided lighting is specified, and only the front material parameters are used in the lighting equation. Otherwise, two-sided lighting is specified. In this case, vertices of back-facing polygons are lighted using the back material parameters, and have their normals reversed before the lighting equation is evaluated. Vertices of front-facing polygons are always lighted using the front material parameters, with no change to their normals. The default is 0.

param

The value to which *param* will be set.

a

m

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*pname* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*pname* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glLightModelf** function sets lighting model parameter. The *pname* parameter names a parameter and *param* gives the new value.the value or values of individual light source parameters.

In RGBA mode, the lighted color of a vertex is the sum of the material emission intensity, the product of the material ambient reflectance and the lighting model full-scene ambient intensity, and the contribution of each enabled light source. Each light source contributes the sum of three terms: ambient, diffuse, and specular.

- The ambient light source contribution is the product of the material ambient reflectance and the light's ambient intensity.
- The diffuse light source contribution is the product of the material diffuse reflectance, the light's diffuse intensity, and the dot product of the vertex's normal with the normalized vector from the vertex to the light source.
- The specular light source contribution is the product of the material specular reflectance, the light's specular intensity, and the dot product of the normalized vertex-to-eye and vertex-to-light vectors, raised to the power of the shininess of the material.

All three light source contributions are attenuated equally based on the distance from the vertex to the light source and on light source direction, spread exponent, and spread cutoff angle. All dot products are replaced with zero if they evaluate to a negative value.

The alpha component of the resulting lighted color is set to the alpha value of the material diffuse reflectance.

In color-index mode, the value of the lighted index of a vertex ranges from the ambient to the specular values passed to glMaterial using GL_COLOR_INDEXES. Diffuse and specular coefficients, computed with a (.30, .59, .11) weighting of the light's colors, the shininess of the material, and the same reflection and attenuation equations as in the RGBA case, determine how much above ambient the resulting index is.

The following functions retrieve information related to the **glLightModelf** function:

glGet with argument GL_LIGHT_MODEL_LOCAL_VIEWER

glGet with argument GL_LIGHT_MODEL_TWO_SIDE

glIsEnabled with argument GL_LIGHTING

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glBegin

glEnd

glLight

glMaterial

glLightModelfv function

2/22/2020 • 3 minutes to read • Edit Online

The **glLightModelfv** function sets lighting model parameters.

Syntax

```
void WINAPI glLightModelfv(
         GLenum  pname,
   const GLfloat *params
);
```

Parameters

pname

A lighting model parameter. The following values are accepted.

VALUE	MEANING
GL_LIGHT_MODEL_AMBIENT	The *params* parameter contains four floating-point values that specify the ambient RGBA intensity of the entire scene. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default ambient scene intensity is (0.2, 0.2, 0.2, 1.0).
GL_LIGHT_MODEL_LOCAL_VIEWER	The *params* parameter is a single floating-point value that specifies how specular reflection angles are computed. If *param* is 0 (or 0.0), specular reflection angles take the view direction to be parallel to and in the direction of the -*z* axis, regardless of the location of the vertex in eye coordinates. Otherwise, specular reflections are computed from the origin of the eye coordinate system. The default is 0.

VALUE	MEANING
GL_LIGHT_MODEL_TWO_SIDE	The *params* parameter is a single floating-point value that specifies whether one-sided or two-sided lighting calculations are done for polygons. It has no effect on the lighting calculations for points, lines, or bitmaps. If *param* is 0 (or 0.0), one-sided lighting is specified, and only the front material parameters are used in the lighting equation. Otherwise, two-sided lighting is specified. In this case, vertices of back-facing polygons are lighted using the back material parameters, and have their normals reversed before the lighting equation is evaluated. Vertices of front-facing polygons are always lighted using the front material parameters, with no change to their normals. The default is 0.

p
a
r
a
m
s

A pointer to the value or values to which *params* will be set.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*pname* was not an accepted value.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*pname* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glLightModelfv** function sets lighting model parameter. The *pname* parameter names a parameter and *param* gives the new value.the value or values of individual light source parameters.

In RGBA mode, the lighted color of a vertex is the sum of the material emission intensity, the product of the material ambient reflectance and the lighting model full-scene ambient intensity, and the contribution of each enabled light source. Each light source contributes the sum of three terms: ambient, diffuse, and specular.

- The ambient light source contribution is the product of the material ambient reflectance and the light's ambient intensity.
- The diffuse light source contribution is the product of the material diffuse reflectance, the light's diffuse intensity, and the dot product of the vertex's normal with the normalized vector from the vertex to the light source.
- The specular light source contribution is the product of the material specular reflectance, the light's specular intensity, and the dot product of the normalized vertex-to-eye and vertex-to-light vectors, raised to the power of the shininess of the material.

All three light source contributions are attenuated equally based on the distance from the vertex to the light source and on light source direction, spread exponent, and spread cutoff angle. All dot products are replaced with zero if

they evaluate to a negative value.

The alpha component of the resulting lighted color is set to the alpha value of the material diffuse reflectance.

In color-index mode, the value of the lighted index of a vertex ranges from the ambient to the specular values passed to **glMaterial** using GL_COLOR_INDEXES. Diffuse and specular coefficients, computed with a (.30, .59, .11) weighting of the light's colors, the shininess of the material, and the same reflection and attenuation equations as in the RGBA case, determine how much above ambient the resulting index is.

The following functions retrieve information related to the **glLightModelfv** function:

glGet with argument GL_LIGHT_MODEL_LOCAL_VIEWER

glGet with argument GL_LIGHT_MODEL_TWO_SIDE

glIsEnabled with argument GL_LIGHTING

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

gl

L
i
g
h
t

g
l
M
a
t
e
r
i
a
l

glLightModeli function

2/22/2020 • 3 minutes to read • Edit Online

The **glLightModeli** function sets lighting model parameters.

Syntax

```
void WINAPI glLightModeli(
   GLenum pname,
   GLint  param
);
```

Parameters

pname

A single-valued lighting model parameter. The following values are accepted.

VALUE	MEANING
GL_LIGHT_MODEL_LOCAL_VIEWER	The *param* parameter is a single integer value that specifies how specular reflection angles are computed. If *param* is 0 (or 0.0), specular reflection angles take the view direction to be parallel to and in the direction of the -*z* axis, regardless of the location of the vertex in eye coordinates. Otherwise, specular reflections are computed from the origin of the eye coordinate system. The default is 0.
GL_LIGHT_MODEL_TWO_SIDE	The *param* parameter is a single integer value that specifies whether one-sided or two-sided lighting calculations are done for polygons. It has no effect on the lighting calculations for points, lines, or bitmaps. If *param* is 0 (or 0.0), one-sided lighting is specified, and only the front material parameters are used in the lighting equation. Otherwise, two-sided lighting is specified. In this case, vertices of back-facing polygons are lighted using the back material parameters, and have their normals reversed before the lighting equation is evaluated. Vertices of front-facing polygons are always lighted using the front material parameters, with no change to their normals. The default is 0.

param

The value to which *param* will be set.

m

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*pname* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*pname* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glLightModeli** function sets lighting model parameter. The *pname* parameter names a parameter and *param* gives the new value.the value or values of individual light source parameters.

In RGBA mode, the lighted color of a vertex is the sum of the material emission intensity, the product of the material ambient reflectance and the lighting model full-scene ambient intensity, and the contribution of each enabled light source. Each light source contributes the sum of three terms: ambient, diffuse, and specular.

- The ambient light source contribution is the product of the material ambient reflectance and the light's ambient intensity.
- The diffuse light source contribution is the product of the material diffuse reflectance, the light's diffuse intensity, and the dot product of the vertex's normal with the normalized vector from the vertex to the light source.
- The specular light source contribution is the product of the material specular reflectance, the light's specular intensity, and the dot product of the normalized vertex-to-eye and vertex-to-light vectors, raised to the power of the shininess of the material.

All three light source contributions are attenuated equally based on the distance from the vertex to the light source and on light source direction, spread exponent, and spread cutoff angle. All dot products are replaced with zero if they evaluate to a negative value.

The alpha component of the resulting lighted color is set to the alpha value of the material diffuse reflectance.

In color-index mode, the value of the lighted index of a vertex ranges from the ambient to the specular values passed to **glMaterial** using GL_COLOR_INDEXES. Diffuse and specular coefficients, computed with a (.30, .59, .11) weighting of the light's colors, the shininess of the material, and the same reflection and attenuation equations as in the RGBA case, determine how much above ambient the resulting index is.

The following functions retrieve information related to the **glLightModeli** function:

glGet with argument GL_LIGHT_MODEL_LOCAL_VIEWER

glGet with argument GL_LIGHT_MODEL_TWO_SIDE

glIsEnabled with argument GL_LIGHTING

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glLight

glMaterial

glLightModeliv function

2/22/2020 • 3 minutes to read • Edit Online

The **glLightModeliv** function sets lighting model parameters.

Syntax

```
void WINAPI glLightModeliv(
         GLenum pname,
   const GLint  *params
);
```

Parameters

pname

A lighting model parameter. The following values are accepted.

VALUE	MEANING
GL_LIGHT_MODEL_AMBIENT	The *params* parameter contains four integer values that specify the ambient RGBA intensity of the entire scene. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default ambient scene intensity is (0.2, 0.2, 0.2, 1.0).
GL_LIGHT_MODEL_LOCAL_VIEWER	The *params* parameter is a single integer value that specifies how specular reflection angles are computed. If *param* is 0 (or 0.0), specular reflection angles take the view direction to be parallel to and in the direction of the -*z* axis, regardless of the location of the vertex in eye coordinates. Otherwise, specular reflections are computed from the origin of the eye coordinate system. The default is 0.

VALUE	MEANING
GL_LIGHT_MODEL_TWO_SIDE	The *params* parameter is a single integer value that specifies whether one-sided or two-sided lighting calculations are done for polygons. It has no effect on the lighting calculations for points, lines, or bitmaps. If *param* is 0 (or 0.0), one-sided lighting is specified, and only the front material parameters are used in the lighting equation. Otherwise, two-sided lighting is specified. In this case, vertices of back-facing polygons are lighted using the back material parameters, and have their normals reversed before the lighting equation is evaluated. Vertices of front-facing polygons are always lighted using the front material parameters, with no change to their normals. The default is 0.

params

A pointer to the value or values to which *params* will be set.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*pname* was not an accepted value.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*pname* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glLightModeliv** function sets lighting model parameter. The *pname* parameter names a parameter and *param* gives the new value.the value or values of individual light source parameters.

In RGBA mode, the lighted color of a vertex is the sum of the material emission intensity, the product of the material ambient reflectance and the lighting model full-scene ambient intensity, and the contribution of each enabled light source. Each light source contributes the sum of three terms: ambient, diffuse, and specular.

- The ambient light source contribution is the product of the material ambient reflectance and the light's ambient intensity.
- The diffuse light source contribution is the product of the material diffuse reflectance, the light's diffuse intensity, and the dot product of the vertex's normal with the normalized vector from the vertex to the light source.
- The specular light source contribution is the product of the material specular reflectance, the light's specular intensity, and the dot product of the normalized vertex-to-eye and vertex-to-light vectors, raised to the power of the shininess of the material.

All three light source contributions are attenuated equally based on the distance from the vertex to the light source and on light source direction, spread exponent, and spread cutoff angle. All dot products are replaced with zero if

they evaluate to a negative value.

The alpha component of the resulting lighted color is set to the alpha value of the material diffuse reflectance.

In color-index mode, the value of the lighted index of a vertex ranges from the ambient to the specular values passed to **glMaterial** using GL_COLOR_INDEXES. Diffuse and specular coefficients, computed with a (.30, .59, .11) weighting of the light's colors, the shininess of the material, and the same reflection and attenuation equations as in the RGBA case, determine how much above ambient the resulting index is.

The following functions retrieve information related to the **glLightModeliv** function:

glGet with argument GL_LIGHT_MODEL_LOCAL_VIEWER

glGet with argument GL_LIGHT_MODEL_TWO_SIDE

glIsEnabled with argument GL_LIGHTING

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

gl

L
i
g
h
t

g
l
M
a
t
e
r
i
a
l

glLineStipple function

2/22/2020 • 2 minutes to read • Edit Online

The **glLineStipple** function specifies the line stipple pattern.

Syntax

```
void WINAPI glLineStipple(
   GLint    factor,
   GLushort pattern
);
```

Parameters

factor

A multiplier for each bit in the line stipple pattern. If *factor* is 3, for example, each bit in the pattern will be used three times before the next bit in the pattern is used. The *factor* parameter is clamped to the range [1, 256] and defaults to one.

pattern

A 16-bit integer whose bit pattern determines which fragments of a line will be drawn when the line is rasterized. Bit zero is used first, and the default pattern is all ones.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Error codes

The following error code can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Remarks

The **glLineStipple** function specifies the line stipple pattern. Line stippling masks out certain fragments produced by rasterization; those fragments will not be drawn. The masking is achieved by using three parameters: the 16-bit line stipple pattern *pattern*, the repeat count *factor*, and an integer stipple counter *s*.

Counter *s* is reset to zero whenever glBegin is called, and before each line segment of a **glBegin**(GL_LINES)/**glEnd** sequence is generated. It is incremented after each fragment of a unit width aliased line segment is generated, or after each *i* fragments of an *i* width line segment are generated. The *i* fragments associated with count *s* are masked out if *pattern* bit (*s* / *factor*) mod 16 is zero. Otherwise these fragments are sent to the framebuffer. Bit zero of *pattern* is the least significant bit.

Antialiased lines are treated as a sequence of 1x*width* rectangles for purposes of stippling. Rectangle *s* is rasterized or not based on the fragment rule described for aliased lines; it counts rectangles rather than groups of fragments.

Line stippling is enabled or disabled using glEnable and **glDisable** with argument GL_LINE_STIPPLE. When enabled, the line stipple pattern is applied as described above. When disabled, it is as if the pattern were all ones. Initially, line stippling is disabled.

The following functions retrieve information related to **glLineStipple**:

glGet with argument GL_LINE_STIPPLE_PATTERN

glGet with argument GL_LINE_STIPPLE_REPEAT

glIsEnabled with argument GL_LINE_STIPPLE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib
DLL	Ope ngl3 2.dll

See also

g
l
B
e
g
i
n

g
l
E
n
d

g
l
L
i
n
e
W
i
d
t
h

g
l
P
o

l
y
g
o
n
S
t
i
p
p
l
e

glLineWidth function

2/22/2020 • 2 minutes to read • Edit Online

The **glLineWidth** function specifies the width of rasterized lines.

Syntax

```
void WINAPI glLineWidth(
   GLfloat width
);
```

Parameters

w *i* *d* *t* *h*	The width of rasterized lines. The default is 1.0.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_ INV ALI D_V AL UE	*width* was less than or equal to zero.
GL_ INV ALI D_ OP ERA TIO N	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	*width* was less than or equal to zero.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glLineWidth** function specifies the rasterized width of both aliased and antialiased lines. Using a line width other than 1.0 has different effects, depending on whether line antialiasing is enabled. Line antialiasing is controlled by calling **glEnable** and **glDisable** with argument GL_LINE_SMOOTH.

If line antialiasing is disabled, the actual width is determined by rounding the supplied width to the nearest integer. (If the rounding results in the value 0.0, it is as if the line width were 1.0) 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 1.0 is guaranteed to be supported; others depend on the implementation. The range of supported widths and the size difference between supported widths within the range can be queried by calling **glGet** with arguments GL_LINE_WIDTH_RANGE and GL_LINE_WIDTH_GRANULARITY.

The line width specified by **glLineWidth** is always returned when GL_LINE_WIDTH is queried. Clamping and rounding for aliased and antialiased lines have no effect on the specified value.

Non-antialiased line width may be clamped to an implementation-dependent maximum. Although this maximum cannot be queried, it must be no less than the maximum value for antialiased lines, rounded to the nearest integer value.

The following functions retrieve information related to **glLineWidth**:

glGet with argument GL_LINE_WIDTH

glGet with argument GL_LINE_WIDTH_RANGE

glGet with argument GL_LINE_WIDTH_GRANULARITY

glIsEnabled with argument GL_LINE_SMOOTH

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnable

glEnd

glIsE

n
a
b
l
e
d

glListBase function

2/22/2020 • 2 minutes to read • Edit Online

The **glListBase** function sets the display list base for **glCallLists**.

Syntax

```
void WINAPI glListBase(
   GLuint base
);
```

Parameters

b
a
s
e

An integer offset that will be added to **glCallLists** offsets to generate display list names. Initial value is zero.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glListBase** function specifies an array of offsets. Display list names are generated by adding *base* to each offset. Names that reference valid display lists are executed; others are ignored.

The following function retrieves information related to **glListBase**:

glGet with argument GL_LIST_BASE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

gl

C
a
l
l
L
i
s
t
s

g
l
E
n
d

glLoadIdentity function

2/22/2020 • 2 minutes to read • Edit Online

The **glLoadIdentity** function replaces the current matrix with the identity matrix.

Syntax

```
void WINAPI glLoadIdentity(void);
```

Parameters

This function has no parameters.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glLoadIdentity** function replaces the current matrix with the identity matrix. It is semantically equivalent to calling glLoadMatrix with the following identity matrix.

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

However, in some cases, it is more efficient.

The following functions retrieve information related to **glLoadIdentity**:

glGet with argument GL_MATRIX_MODE

glGet with argument GL_MODELVIEW_MATRIX

glGet with argument GL_PROJECTION_MATRIX

glGet with argument GL_TEXTURE_MATRIX

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib
DLL	Ope ngl3 2.dll

See also

g
l
B
e
g
i
n

g
l
E
n
d

g

l
L
o
a
d
M
a
t
r
i
x

g
l
M
a
t
r
i
x
M
o
d
e

g
l
M
u
l
t
M
a
t
r
i
x

g
l
P
u
s
h
M
a
t
r
i
x

glLoadMatrix Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions replace the current matrix with an arbitrary matrix:

- glLoadMatrixd
- glLoadMatrixf

glLoadMatrixd function

2/22/2020 • 2 minutes to read • Edit Online

The **glLoadMatrixd** and **glLoadMatrixf** functions replace the current matrix with an arbitrary matrix.

Syntax

```
void WINAPI glLoadMatrixd(
   const GLdouble *m
);
```

Parameters

m	A pointer to a 4x4 matrix stored in column-major order as 16 consecutive values.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glLoadMatrix** function replaces the current matrix with the one specified in *m*. The current matrix is the projection matrix, modelview matrix, or texture matrix, determined by the current matrix mode (see **glMatrixMode**).

The *m* parameter points to a 4x4 matrix of single-precision or double-precision floating-point values stored in column-major order. That is, the matrix is stored as shown in the following image.

$$\begin{bmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{bmatrix}$$

The following functions retrieve information related to **glLoadMatrix**:

glGet with argument GL_MATRIX_MODE

glGet with argument GL_MODELVIEW_MATRIX

glGet with argument GL_PROJECTION_MATRIX

glGet with argument GL_TEXTURE_MATRIX

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glLoadIdentity

glMatrixMode

glMultMatrix

glPush

M
a
t
r
i
x

glLoadMatrixf function

2/22/2020 • 2 minutes to read • Edit Online

The **glLoadMatrixd** and **glLoadMatrixf** functions replace the current matrix with an arbitrary matrix.

Syntax

```
void WINAPI glLoadMatrixf(
   const GLfloat *m
);
```

Parameters

m	A pointer to a 4x4 matrix stored in column-major order as 16 consecutive values.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glLoadMatrix** function replaces the current matrix with the one specified in *m*. The current matrix is the projection matrix, modelview matrix, or texture matrix, determined by the current matrix mode (see glMatrixMode).

The *m* parameter points to a 4x4 matrix of single-precision or double-precision floating-point values stored in column-major order. That is, the matrix is stored as shown in the following image.

$$\begin{bmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{bmatrix}$$

The following functions retrieve information related to **glLoadMatrix**:

glGet with argument GL_MATRIX_MODE

glGet with argument GL_MODELVIEW_MATRIX

glGet with argument GL_PROJECTION_MATRIX

glGet with argument GL_TEXTURE_MATRIX

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glLoadIdentity

glMatrixMode

glMultMatrix

glPush

M
a
t
r
i
x

glLoadName function

2/22/2020 • 2 minutes to read • Edit Online

The **glLoadName** function loads a name onto the name stack.

Syntax

```
void WINAPI glLoadName(
   GLuint name
);
```

Parameters

name	A name that will replace the top value on the name stack.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called while the name stack was empty.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called while the name stack was empty.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glLoadName** function causes *name* to replace the value on the top of the name stack, which is initially empty. The name stack is used during selection mode to allow sets of rendering commands to be uniquely identified. It consists of an ordered set of unsigned integers.

The name stack is always empty while the render mode is not GL_SELECT. Calls to **glLoadName** while the render mode is not GL_SELECT are ignored.

The following functions retrieve information related to **glLoadName**:

glGet with argument GL_NAME_STACK_DEPTH

glGet with argument GL_MAX_NAME_STACK_DEPTH

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib

DLL	Opengl32.dll

See also

glBegin

glEnd

glInitNames

glPushName

glRende

r
M
o
d
e

g
l
S
e
l
e
c
t
B
u
f
f
e
r

glLogicOp function

2/22/2020 • 2 minutes to read • Edit Online

The **glLogicOp** function specifies a logical pixel operation for color index rendering.

Syntax

```
void WINAPI glLogicOp(
   GLenum opcode
);
```

Parameters

opcode

A symbolic constant that selects a logical operation. The following symbols are accepted where s equals the value of the source bit and d is the value of the destination bit.

VALUE	MEANING
GL_CLEAR	0
GL_SET	1
GL_COPY	s
GL_COPY_INVERTED	!s

VALUE	MEANING
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

VALUE	MEANING
GL_OR_REVERSE	s | !d
GL_OR_INVERTED	!s | d

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*opcode* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*opcode* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glLogicOp** function specifies a logical operation that, when enabled, is applied between the incoming color index and the color index at the corresponding location in the framebuffer. The logical operation is enabled or disabled with **glEnable** and **glDisable** using the symbolic constant GL_LOGIC_OP.

The *opcode* parameter is a symbolic constant chosen from the list below. In the explanation of the logical operations, *s* represents the incoming color index and *d* represents the index in the framebuffer. Standard C-language operators are used. As these bitwise operators suggest, the logical operation is applied independently to each bit pair of the source and destination indexes.

Logical pixel operations are not applied to RGBA color buffers.

When more than one color-index buffer is enabled for drawing, logical operations are done separately for each enabled buffer, using the contents of that buffer for the destination index (see **glDrawBuffer**).

The *opcode* parameter must be one of the 16 accepted values. Other values result in an error.

The following functions retrieve information related to **glLogicOp**:

glGet with argument GL_LOGIC_OP_MODE

glIsEnabled with argument GL_LOGIC_OP

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glAlphaFunc

glBegin

glBlendFunc

glDrawB

u
f
f
e
r

g
l
E
n
a
b
l
e

g
l
E
n
d

g
l
I
s
E
n
a
b
l
e
d

g
l
S
t
e
n
c
i
l
O
p

glMap1 Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions define a one-dimensional evaluator:

- **glMap1d**
- **glMap1f**

glMap1d function

2/22/2020 • 5 minutes to read • Edit Online

The **glMap1d** and **glMap1f** functions define a one-dimensional evaluator.

Syntax

```
void WINAPI glMap1d(
         GLenum   target,
         GLdouble u1,
         GLdouble u2,
         GLint    stride,
         GLint    order,
   const GLdouble *points
);
```

Parameters

target

The kind of values that are generated by the evaluator. Symbolic constants. The *target* parameter is a symbolic constant that indicates what kind of control points are provided in *points*, and what output is generated when the map is evaluated. It can assume one of nine predefined values.

VALUE	MEANING
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.

VALUE	MEANING
GL_MAP1_INDEX	Each control point is a single floating-point value representing a color index. Internal **glIndex** commands are generated when the map is evaluated. However, the current index is not updated with the value of these **glIndex** commands.
GL_MAP1_COLOR_4	Each control point is four floating-point values representing red, green, blue, and alpha. Internal **glColor4** commands are generated when the map is evaluated. However, the current color is not updated with the value of these **glColor4** commands.
GL_MAP1_NORMAL	Each control point is three floating-point values representing the *x*, *y*, and *z* components of a normal vector. Internal **glNormal** commands are generated when the map is evaluated. However, the current normal is not updated with the value of these **glNormal** commands.
GL_MAP1_TEXTURE_COORD_1	Each control point is a single floating-point value representing the *s* texture coordinate. Internal **glTexCoord1** commands are generated when the map is evaluated. However, the current texture coordinates are not updated with the value of these **glTexCoord** commands.
GL_MAP1_TEXTURE_COORD_2	Each control point is two floating-point values representing the *s* and *t* texture coordinates. Internal **glTexCoord2** commands are generated when the map is evaluated. However, the current texture coordinates are not updated with the value of these **glTexCoord** commands.
GL_MAP1_TEXTURE_COORD_3	Each control point is three floating-point values representing the *s*, *t*, and *r* texture coordinates. Internal **glTexCoord3** commands are generated when the map is evaluated. However, the current texture coordinates are not updated with the value of these **glTexCoord** commands.

VALUE	MEANING
GL_MAP1_TEXTURE_COORD_4	Each control point is four floating-point values representing the *s, t, r,* and *q* texture coordinates. Internal **glTexCoord4** commands are generated when the map is evaluated. However, the current texture coordinates are not updated with the value of these **glTexCoord** commands.

u1

A linear mapping of *u*, as presented to **glEvalCoord1**, to *u*^, the variable that is evaluated by the equations specified by this command.

u2

A linear mapping of *u*, as presented to **glEvalCoord1**, to *u*^, the variable that is evaluated by the equations specified by this command.

stride

The number of floats or doubles between the beginning of one control point and the beginning of the next one in the data structure referenced in *points*. This allows control points to be embedded in arbitrary data structures. The only constraint is that the values for a particular control point must occupy contiguous memory locations.

order

The number of control points. Must be positive.

points

A pointer to the array of control points.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* was not an accepted value.
GL_INVALID_VALUE	*u1* was equal to *u2*.
GL_INVALID_VALUE	*stride* was less than the number of values in a control point.
GL_INVALID_VALUE	*order* was less than one or GL_MAX_EVAL_ORDER.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* was not an accepted value.
GL_INVALID_VALUE	*u1* was equal to *u2*.
GL_INVALID_VALUE	*stride* was less than the number of values in a control point.
GL_INVALID_VALUE	*order* was less than one or GL_MAX_EVAL_ORDER.
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Remarks

Evaluators provide a way to use polynomial or rational polynomial mapping to produce vertices, normals, texture coordinates, and colors. The values produced by an evaluator are sent to further stages of OpenGL processing just as if they had been presented using glVertex, glNormal, glTexCoord, and glColor commands, except that the generated values do not update the current normal, texture coordinates, or color.

All polynomial or rational polynomial splines of any degree (up to the maximum degree supported by the OpenGL implementation) can be described using evaluators. These include almost all splines used in computer graphics, including B-splines, Bezier curves, Hermite splines, and so on.

Evaluators define curves based on Bernstein polynomials. Define **p** () as

$$\mathbf{p}(\hat{u}) = \sum_{i=0}^{n} B_i^n(\hat{u})\mathbf{R}_i$$

where **R** *i* is a control point and () is the *i* the Bernstein polynomial of degree *n* (*order* = *n* + 1):

$$B_i^n(\hat{u}) = \binom{n}{i} \hat{u}^i (1-\hat{u})^{n-i}$$

Recall that

$$0^0 \equiv 1 \text{ and } \binom{n}{0} \equiv 1$$

The **glMap1** function is used to define the basis and to specify what kind of values are produced. Once defined, a map can be enabled and disabled by calling **glEnable** and **glDisable** with the map name, one of the nine predefined values for *target* described above. The glEvalCoord1 function evaluates the one-dimensional maps that are enabled. When **glEvalCoord1** presents a value *u*, the Bernstein functions are evaluated using *u*^, where

$$\hat{u} = \frac{u - ul}{u2 - ul}$$

The *stride*, *order*, and *points* parameters define the array addressing for accessing the control points. The *points* parameter is the location of the first control point, which occupies one, two, three, or four contiguous memory locations, depending on which map is being defined. The *order* parameter is the number of control points in the array. The *stride* parameter tells how many float or double locations to advance the internal memory pointer to reach the next control point.

As is the case with all OpenGL commands that accept pointers to data, it is as if the contents of *points* were copied by **glMap1** before it returned. Changes to the contents of *points* have no effect after **glMap1** is called.

The following functions retrieve information related to **glMap1**:

glGet with argument GL_MAX_EVAL_ORDER

glGetMap

glIsEnabled with argument GL_MAP1_VERTEX_3

glIsEnabled with argument GL_MAP1_VERTEX_4

glIsEnabled with argument GL_MAP1_INDEX

glIsEnabled with argument GL_MAP1_COLOR_4

glIsEnabled with argument GL_MAP1_NORMAL

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_1

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_2

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_3

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_4

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glEnable

glEnd

glE

v
a
l
C
o
o
r
d

g
l
E
v
a
l
M
e
s
h

g
l
E
v
a
l
P
o
i
n
t

g
l
M
a
p
2

g
l
M
a
p
G
r
i
d

g
l
N
o

r
m
a
l

g
l
T
e
x
C
o
o
r
d

g
l
V
e
r
t
e
x

glMap1f function

2/22/2020 • 5 minutes to read • Edit Online

The **glMap1d** and **glMap1f** functions define a one-dimensional evaluator.

Syntax

```
void WINAPI glMap1f(
         GLenum  target,
         GLfloat u1,
         GLfloat u2,
         GLint   stride,
         GLint   order,
   const GLfloat *points
);
```

Parameters

target

The kind of values that are generated by the evaluator. Symbolic constants. The *target* parameter is a symbolic constant that indicates what kind of control points are provided in *points*, and what output is generated when the map is evaluated. It can assume one of nine predefined values.

VALUE	MEANING
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.

VALUE	MEANING
GL_MAP1_INDEX	Each control point is a single floating-point value representing a color index. Internal glIndex commands are generated when the map is evaluated. However, the current index is not updated with the value of these **glIndex** commands.
GL_MAP1_COLOR_4	Each control point is four floating-point values representing red, green, blue, and alpha. Internal glColor4 commands are generated when the map is evaluated. However, the current color is not updated with the value of these **glColor4** commands.
GL_MAP1_NORMAL	Each control point is three floating-point values representing the *x, y,* and *z* components of a normal vector. Internal glNormal commands are generated when the map is evaluated. However, the current normal is not updated with the value of these **glNormal** commands.
GL_MAP1_TEXTURE_COORD_1	Each control point is a single floating-point value representing the *s* texture coordinate. Internal glTexCoord1 commands are generated when the map is evaluated. However, the current texture coordinates are not updated with the value of these **glTexCoord** commands.
GL_MAP1_TEXTURE_COORD_2	Each control point is two floating-point values representing the *s* and *t* texture coordinates. Internal glTexCoord2 commands are generated when the map is evaluated. However, the current texture coordinates are not updated with the value of these **glTexCoord** commands.
GL_MAP1_TEXTURE_COORD_3	Each control point is three floating-point values representing the *s, t,* and *r* texture coordinates. Internal glTexCoord3 commands are generated when the map is evaluated. However, the current texture coordinates are not updated with the value of these **glTexCoord** commands.

VALUE	MEANING
GL_MAP1_TEXTURE_COORD_4	Each control point is four floating-point values representing the *s, t, r,* and *q* texture coordinates. Internal **glTexCoord4** commands are generated when the map is evaluated. However, the current texture coordinates are not updated with the value of these **glTexCoord** commands.

u1

A linear mapping of *u*, as presented to **glEvalCoord1**, to *u*^, the variable that is evaluated by the equations specified by this command.

u2

A linear mapping of *u*, as presented to **glEvalCoord1**, to *u*^, the variable that is evaluated by the equations specified by this command.

stride

The number of floats or doubles between the beginning of one control point and the beginning of the next one in the data structure referenced in *points*. This allows control points to be embedded in arbitrary data structures. The only constraint is that the values for a particular control point must occupy contiguous memory locations.

order

The number of control points. Must be positive.

points

A pointer to the array of control points.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* was not an accepted value.
GL_INVALID_VALUE	*u1* was equal to *u2*.
GL_INVALID_VALUE	*stride* was less than the number of values in a control point.
GL_INVALID_VALUE	*order* was less than one or GL_MAX_EVAL_ORDER.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* was not an accepted value.
GL_INVALID_VALUE	*u1* was equal to *u2*.
GL_INVALID_VALUE	*stride* was less than the number of values in a control point.
GL_INVALID_VALUE	*order* was less than one or GL_MAX_EVAL_ORDER.
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Remarks

Evaluators provide a way to use polynomial or rational polynomial mapping to produce vertices, normals, texture coordinates, and colors. The values produced by an evaluator are sent to further stages of OpenGL processing just as if they had been presented using glVertex, glNormal, glTexCoord, and glColor commands, except that the generated values do not update the current normal, texture coordinates, or color.

All polynomial or rational polynomial splines of any degree (up to the maximum degree supported by the OpenGL implementation) can be described using evaluators. These include almost all splines used in computer graphics, including B-splines, Bezier curves, Hermite splines, and so on.

Evaluators define curves based on Bernstein polynomials. Define **p** () as

$$\mathbf{p}(\hat{u}) = \sum_{i=0}^{n} B_i^n(\hat{u})\mathbf{R}_i$$

where **R** *i* is a control point and () is the *i* the Bernstein polynomial of degree *n* (*order* = *n* + 1):

$$B_i^n(\hat{u}) = \binom{n}{i} \hat{u}^i (1-\hat{u})^{n-i}$$

Recall that

$$0^0 \equiv 1 \text{ and } \binom{n}{0} \equiv 1$$

The **glMap1** function is used to define the basis and to specify what kind of values are produced. Once defined, a map can be enabled and disabled by calling **glEnable** and **glDisable** with the map name, one of the nine predefined values for *target* described above. The glEvalCoord1 function evaluates the one-dimensional maps that are enabled. When **glEvalCoord1** presents a value *u*, the Bernstein functions are evaluated using *u*^, where

$$\hat{u} = \frac{u - ul}{u2 - ul}$$

The *stride*, *order*, and *points* parameters define the array addressing for accessing the control points. The *points* parameter is the location of the first control point, which occupies one, two, three, or four contiguous memory locations, depending on which map is being defined. The *order* parameter is the number of control points in the array. The *stride* parameter tells how many float or double locations to advance the internal memory pointer to reach the next control point.

As is the case with all OpenGL commands that accept pointers to data, it is as if the contents of *points* were copied by **glMap1** before it returned. Changes to the contents of *points* have no effect after **glMap1** is called.

The following functions retrieve information related to **glMap1**:

glGet with argument GL_MAX_EVAL_ORDER

glGetMap

glIsEnabled with argument GL_MAP1_VERTEX_3

glIsEnabled with argument GL_MAP1_VERTEX_4

glIsEnabled with argument GL_MAP1_INDEX

glIsEnabled with argument GL_MAP1_COLOR_4

glIsEnabled with argument GL_MAP1_NORMAL

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_1

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_2

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_3

glIsEnabled with argument GL_MAP1_TEXTURE_COORD_4

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glEnable

glEnd

glE

v
a
l
C
o
o
r
d

g
l
E
v
a
l
M
e
s
h

g
l
E
v
a
l
P
o
i
n
t

g
l
M
a
p
2

g
l
M
a
p
G
r
i
d

g
l
N
o

r
m
a
l

g
l
T
e
x
C
o
o
r
d

g
l
V
e
r
t
e
x

glMap2 Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions define a two-dimensional evaluator:

- glMap2d
- glMap2f

glMap2d function

2/22/2020 • 6 minutes to read • Edit Online

The **glMap2d** and **glMap2f** functions define a two-dimensional evaluator.

Syntax

```
void WINAPI glMap2d(
         GLenum   target,
         GLdouble u1,
         GLdouble u2,
         GLint    ustride,
         GLint    uorder,
         GLdouble v1,
         GLdouble v2,
         GLint    vstride,
         GLint    vorder,
   const GLdouble *points
);
```

Parameters

target

The kind of values that are generated by the evaluator. The following symbolic constants are accepted.

VALUE	MEANING
GL_MAP2_VERTEX_3	Each control point is three floating-point values representing *x*, *y*, and *z*. Internal **glVertex3** commands are generated when the map is evaluated.
GL_MAP2_VERTEX_4	Each control point is four floating-point values representing *x*, *y*, *z*, and *w*. Internal **glVertex4** commands are generated when the map is evaluated.
GL_MAP2_INDEX	Each control point is a single floating-point value representing a color index. Internal **glIndex** commands are generated when the map is evaluated. The current index is not updated with the value of these **glIndex** commands, however.

VALUE	MEANING
GL_MAP2_COLOR_4	Each control point is four floating-point values representing red, green, blue, and alpha. Internal glColor4 commands are generated when the map is evaluated. The current color is not updated with the value of these **glColor4** commands, however.
GL_MAP2_NORMAL	Each control point is three floating-point values representing the *x*, *y*, and *z* components of a normal vector. Internal glNormal commands are generated when the map is evaluated. The current normal is not updated with the value of these **glNormal** commands, however.
GL_MAP2_TEXTURE_COORD_1	Each control point is a single floating-point value representing the *s* texture coordinate. Internal glTexCoord1 commands are generated when the map is evaluated. The current texture coordinates are not updated with the value of these **glTexCoord** commands, however.
GL_MAP2_TEXTURE_COORD_2	Each control point is two floating-point values representing the *s* and *t* texture coordinates. Internal glTexCoord2 commands are generated when the map is evaluated. The current texture coordinates are not updated with the value of these **glTexCoord** commands, however.
GL_MAP2_TEXTURE_COORD_3	Each control point is three floating-point values representing the *s*, *t*, and *r* texture coordinates. Internal glTexCoord3 commands are generated when the map is evaluated. The current texture coordinates are not updated with the value of these **glTexCoord** commands, however.

VALUE	MEANING
GL_MAP2_TEXTURE_COORD_4	Each control point is four floating-point values representing the *s, t, r,* and *q* texture coordinates. Internal **glTexCoord4** commands are generated when the map is evaluated. The current texture coordinates are not updated with the value of these **glTexCoord** commands, however.

u1

A linear mapping of *u*, as presented to **glEvalCoord2**, to *u*^, one of the two variables that is evaluated by the equations specified by this command.

u2

A linear mapping of *u*, as presented to **glEvalCoord2**, to *u*^, one of the two variables that is evaluated by the equations specified by this command.

ustride

The number of floats or doubles between the beginning of control point **R** *ij* and the beginning of control point **R** $_{(i\ +1\ )\ j}$, where *i* and *j* are the *u* and *v* control point indexes, respectively. This allows control points to be embedded in arbitrary data structures. The only constraint is that the values for a particular control point must occupy contiguous memory locations.

uorder The dimension of the control point array in the *u*-axis. Must be positive.

v1

A linear mapping of *v*, as presented to **glEvalCoord2**, to *v*^, one of the two variables that is evaluated by the equations specified by this command.

v2

A linear mapping of *v*, as presented to **glEvalCoord2**, to *v*^, one of the two variables that is evaluated by the equations specified by this command.

v
s
t
r
i
d
e

The number of floats or doubles between the beginning of control point **R** *ij* and the beginning of control point **R** $_{i(j\backslash +1\backslash)}$, where *i* and *j* are the *u* and *v* control point indexes, respectively. This allows control points to be embedded in arbitrary data structures. The only constraint is that the values for a particular control point must occupy contiguous memory locations.

v
o
r
d
e
r

The dimension of the control point array in the *v*-axis. Must be positive.

p
o
i
n
t
s

A pointer to the array of control points.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* was not an accepted value.
GL_INVALID_VALUE	*u1* was equal to *u2*, or *v1* was equal to *v2*.

NAME	MEANING
GL_INVALID_VALUE	Either *ustride* or *vstride* was less than the number of values in a control point.
GL_INVALID_VALUE	Either *uorder* or *vorder* was less than one or GL_MAX_EVAL_ORDER.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* was not an accepted value.
GL_INVALID_VALUE	*u1* was equal to *u2*, or *v1* was equal to *v2*.
GL_INVALID_VALUE	Either *ustride* or *vstride* was less than the number of values in a control point.

NAME	MEANING
GL_ INV ALI D_V AL UE	Either *uorder* or *vorder* was less than one or GL_MAX_EVAL_ORDER.
GL_ INV ALI D_ OP ERA TIO N	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

Evaluators provide a way to use polynomial or rational polynomial mapping to produce vertices, normals, texture coordinates, and colors. The values produced by an evaluator are sent on to further stages of OpenGL processing just as if they had been presented using **glVertex**, **glNormal**, **glTexCoord**, and **glColor** commands, except that the generated values do not update the current normal, texture coordinates, or color.

All polynomial or rational polynomial splines of any degree (up to the maximum degree supported by the OpenGL implementation) can be described using evaluators. These include almost all surfaces used in computer graphics, including B-spline surfaces, NURBS surfaces, Bezier surfaces, and so on.

Evaluators define surfaces based on bivariate Bernstein polynomials. Define **p** (*u*^,*v*^) as

$$\mathbf{p}(\hat{u},\hat{v}) = \sum_{i=0}^{n}\sum_{j=0}^{m} B_i^n(\hat{u})\, B_j^m(\hat{v})\, \mathbf{R}_{ij}$$

where **R** *ij* is a control point, () is the *i*th Bernstein polynomial of degree

n (*uorder* = *n* + 1)

$$B_i^n(\hat{u}) = \binom{n}{i} \hat{u}^i (1-\hat{u})^{n-i}$$

and () is the *j*th Bernstein polynomial of degree *m* (*vorder* = *m* + 1)

$$B_j^m(\hat{v}) = \binom{m}{j} \hat{v}^j (1-\hat{v})^{m-j}$$

Recall that

$$0^0 \equiv 1 \text{ and } \binom{n}{0} \equiv 1$$

The **glMap2** function is used to define the basis and to specify what kind of values are produced. Once defined, a map can be enabled and disabled by calling **glEnable** and **glDisable** with the map name, one of the nine predefined values for *target*, described above. When **glEvalCoord2** presents values *u* and *v*, the bivariate Bernstein polynomials are evaluated using *u*^ and *v*^, where

$$\hat{u} = \frac{u - u1}{u2 - u1}$$

and

$$\hat{v} = \frac{v - v1}{v2 - v1}$$

The *target* parameter is a symbolic constant that indicates what kind of control points are provided in *points*, and what output is generated when the map is evaluated.

The *ustride*, *uorder*, *vstride*, *vorder*, and *points* parameters define the array addressing for accessing the control points. The *points* parameter is the location of the first control point, which occupies one, two, three, or four contiguous memory locations, depending on which map is being defined. There are *uorder* x *vorder* control points in the array. The *ustride* parameter tells how many float or double locations are skipped to advance the internal memory pointer from control point **R** *ij* to control point **R** $_{(\backslash i+1\backslash)j}$. The *vstride* parameter tells how many float or double locations are skipped to advance the internal memory pointer from control point **R** *ij* to control point **R**$_{i(j\backslash +1\backslash)}$.

As is the case with all OpenGL commands that accept pointers to data, it is as if the contents of *points* were copied by **glMap2** before it returned. Changes to the contents of *points* have no effect after **glMap2** is called.

The following functions retrieve information related to **glMap2**:

glGet with argument GL_MAX_EVAL_ORDER

glGetMap

glIsEnabled with argument GL_MAP2_VERTEX_3

glIsEnabled with argument GL_MAP2_VERTEX_4

glIsEnabled with argument GL_MAP2_INDEX

glIsEnabled with argument GL_MAP2_COLOR_4

glIsEnabled with argument GL_MAP2_NORMAL

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_1

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_2

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_3

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_4

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glEnable

glEnd

glEvalCo

o
r
d

g
l
E
v
a
l
M
e
s
h

g
l
E
v
a
l
P
o
i
n
t

g
l
M
a
p
1

g
l
M
a
p
G
r
i
d

g
l
N
o
r
m
a
l

g

l
T
e
x
C
o
o
r
d

g
l
V
e
r
t
e
x

glMap2f function

2/22/2020 • 6 minutes to read • Edit Online

The **glMap2d** and **glMap2f** functions define a two-dimensional evaluator.

Syntax

```
void WINAPI glMap2f(
         GLenum  target,
         GLfloat u1,
         GLfloat u2,
         GLint   ustride,
         GLint   uorder,
         GLfloat v1,
         GLfloat v2,
         GLint   vstride,
         GLint   vorder,
   const GLfloat *points
);
```

Parameters

target

The kind of values that are generated by the evaluator. The following symbolic constants are accepted.

VALUE	MEANING
GL_MAP2_VERTEX_3	Each control point is three floating-point values representing *x*, *y*, and *z*. Internal **glVertex3** commands are generated when the map is evaluated.
GL_MAP2_VERTEX_4	Each control point is four floating-point values representing *x*, *y*, *z*, and *w*. Internal **glVertex4** commands are generated when the map is evaluated.
GL_MAP2_INDEX	Each control point is a single floating-point value representing a color index. Internal **glIndex** commands are generated when the map is evaluated. The current index is not updated with the value of these **glIndex** commands, however.

VALUE	MEANING
GL_MAP2_COLOR_4	Each control point is four floating-point values representing red, green, blue, and alpha. Internal glColor4 commands are generated when the map is evaluated. The current color is not updated with the value of these **glColor4** commands, however.
GL_MAP2_NORMAL	Each control point is three floating-point values representing the *x*, *y*, and *z* components of a normal vector. Internal glNormal commands are generated when the map is evaluated. The current normal is not updated with the value of these **glNormal** commands, however.
GL_MAP2_TEXTURE_COORD_1	Each control point is a single floating-point value representing the *s* texture coordinate. Internal glTexCoord1 commands are generated when the map is evaluated. The current texture coordinates are not updated with the value of these **glTexCoord** commands, however.
GL_MAP2_TEXTURE_COORD_2	Each control point is two floating-point values representing the *s* and *t* texture coordinates. Internal glTexCoord2 commands are generated when the map is evaluated. The current texture coordinates are not updated with the value of these **glTexCoord** commands, however.
GL_MAP2_TEXTURE_COORD_3	Each control point is three floating-point values representing the *s*, *t*, and *r* texture coordinates. Internal glTexCoord3 commands are generated when the map is evaluated. The current texture coordinates are not updated with the value of these **glTexCoord** commands, however.

VALUE	MEANING
GL_MAP2_TEXTURE_COORD_4	Each control point is four floating-point values representing the *s, t, r,* and *q* texture coordinates. Internal **glTexCoord4** commands are generated when the map is evaluated. The current texture coordinates are not updated with the value of these **glTexCoord** commands, however.

u1

A linear mapping of *u*, as presented to **glEvalCoord2**, to *u*^, one of the two variables that is evaluated by the equations specified by this command.

u2

A linear mapping of *u*, as presented to **glEvalCoord2**, to *u*^, one of the two variables that is evaluated by the equations specified by this command.

ustride

The number of floats or doubles between the beginning of control point **R** $_{ij}$ and the beginning of control point **R** $_{(i\ +1\ )\ j}$, where *i* and *j* are the *u* and *v* control point indexes, respectively. This allows control points to be embedded in arbitrary data structures. The only constraint is that the values for a particular control point must occupy contiguous memory locations.

uorder The dimension of the control point array in the *u*-axis. Must be positive.

v1

A linear mapping of *v*, as presented to **glEvalCoord2**, to *v*^, one of the two variables that is evaluated by the equations specified by this command.

v2

A linear mapping of *v*, as presented to **glEvalCoord2**, to *v*^, one of the two variables that is evaluated by the equations specified by this command.

v
s
t
r
i
d
e

The number of floats or doubles between the beginning of control point **R** *ij* and the beginning of control point **R** $_{i(j\backslash +1\backslash)}$, where *i* and *j* are the *u* and *v* control point indexes, respectively. This allows control points to be embedded in arbitrary data structures. The only constraint is that the values for a particular control point must occupy contiguous memory locations.

v
o
r
d
e
r

The dimension of the control point array in the *v*-axis. Must be positive.

p
o
i
n
t
s

A pointer to the array of control points.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* was not an accepted value.
GL_INVALID_VALUE	*u1* was equal to *u2*, or *v1* was equal to *v2*.

NAME	MEANING
GL_INVALID_VALUE	Either *ustride* or *vstride* was less than the number of values in a control point.
GL_INVALID_VALUE	Either *uorder* or *vorder* was less than one or GL_MAX_EVAL_ORDER.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* was not an accepted value.
GL_INVALID_VALUE	*u1* was equal to *u2*, or *v1* was equal to *v2*.
GL_INVALID_VALUE	Either *ustride* or *vstride* was less than the number of values in a control point.

NAME	MEANING
GL_INVALID_VALUE	Either *uorder* or *vorder* was less than one or GL_MAX_EVAL_ORDER.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

Evaluators provide a way to use polynomial or rational polynomial mapping to produce vertices, normals, texture coordinates, and colors. The values produced by an evaluator are sent on to further stages of OpenGL processing just as if they had been presented using **glVertex**, **glNormal**, **glTexCoord**, and **glColor** commands, except that the generated values do not update the current normal, texture coordinates, or color.

All polynomial or rational polynomial splines of any degree (up to the maximum degree supported by the OpenGL implementation) can be described using evaluators. These include almost all surfaces used in computer graphics, including B-spline surfaces, NURBS surfaces, Bezier surfaces, and so on.

Evaluators define surfaces based on bivariate Bernstein polynomials. Define **p** (*u*^,*v*^) as

$$\mathbf{p}(\hat{u},\hat{v}) = \sum_{i=0}^{n}\sum_{j=0}^{m} B_i^n(\hat{u})\,B_j^m(\hat{v})\,\mathbf{R}_{ij}$$

where **R** *ij* is a control point, () is the *i*th Bernstein polynomial of degree

n (*uorder* = *n* + 1)

$$B_i^n(\hat{u}) = \binom{n}{i}\hat{u}^i\,(1-\hat{u})^{n-i}$$

and () is the *j*th Bernstein polynomial of degree *m* (*vorder* = *m* + 1)

$$B_j^m(\hat{v}) = \binom{m}{j}\hat{v}^j\,(1-\hat{v})^{m-j}$$

Recall that

$$0^0 \equiv 1 \text{ and } \binom{n}{0} \equiv 1$$

The **glMap2** function is used to define the basis and to specify what kind of values are produced. Once defined, a map can be enabled and disabled by calling **glEnable** and **glDisable** with the map name, one of the nine predefined values for *target*, described above. When **glEvalCoord2** presents values *u* and *v*, the bivariate Bernstein polynomials are evaluated using *u*^ and *v*^, where

$$\hat{u} = \frac{u - u1}{u2 - u1}$$

and

$$\hat{v} = \frac{v - v1}{v2 - v1}$$

The *target* parameter is a symbolic constant that indicates what kind of control points are provided in *points*, and what output is generated when the map is evaluated.

The *ustride*, *uorder*, *vstride*, *vorder*, and *points* parameters define the array addressing for accessing the control points. The *points* parameter is the location of the first control point, which occupies one, two, three, or four contiguous memory locations, depending on which map is being defined. There are *uorder* x *vorder* control points in the array. The *ustride* parameter tells how many float or double locations are skipped to advance the internal memory pointer from control point **R** *ij* to control point **R** $_{(\backslash i+1\backslash)j}$. The *vstride* parameter tells how many float or double locations are skipped to advance the internal memory pointer from control point **R** *ij* to control point **R**$_{i(j\backslash +1\backslash)}$.

As is the case with all OpenGL commands that accept pointers to data, it is as if the contents of *points* were copied by **glMap2** before it returned. Changes to the contents of *points* have no effect after **glMap2** is called.

The following functions retrieve information related to **glMap2**:

glGet with argument GL_MAX_EVAL_ORDER

glGetMap

glIsEnabled with argument GL_MAP2_VERTEX_3

glIsEnabled with argument GL_MAP2_VERTEX_4

glIsEnabled with argument GL_MAP2_INDEX

glIsEnabled with argument GL_MAP2_COLOR_4

glIsEnabled with argument GL_MAP2_NORMAL

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_1

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_2

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_3

glIsEnabled with argument GL_MAP2_TEXTURE_COORD_4

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glEnable

glEnd

glEvalCo

o
r
d

g
l
E
v
a
l
M
e
s
h

g
l
E
v
a
l
P
o
i
n
t

g
l
M
a
p
1

g
l
M
a
p
G
r
i
d

g
l
N
o
r
m
a
l

g

lTexCoord

glVertex

glMapGrid Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions defines a one-dimensional or two-dimensional mesh:

- glMapGrid1d
- glMapGrid1f
- glMapGrid2d
- glMapGrid2f

glMapGrid1d function

2/22/2020 • 2 minutes to read • Edit Online

Defines a one-dimensional mesh.

Syntax

```
void WINAPI glMapGrid1d(
   GLint    un,
   GLdouble u1,
   GLdouble u2
);
```

Parameters

un	The number of partitions in the grid range interval [u1, u2]. This value must be positive.
u1	A value used as the mapping for integer grid domain value i = 0.
u2	A value used as the mapping for integer grid domain value i = un.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_VALUE	Either *un* or *vn* was not positive.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	Either *un* or *vn* was not positive.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

Use the **glMapGrid** and glEvalMesh functions togetherto efficiently generate and evaluate a series of evenly spaced map domain values. The glEvalMesh function steps through the integer domain of a one- or two-dimensional grid, whose range is the domain of the evaluation maps specified by **glMap1** and **glMap2**.

The **glMapGrid1** and **glMapGrid2** functions specify the linear grid mappings between the i (or i and j) integer grid coordinates, to the u (or u and v) floating-point evaluation map coordinates. See **glMap1** and **glMap2** for details of how u and v coordinates are evaluated.

The **glMapGrid1** function specifies a single linear mapping such that integer grid coordinate 0 maps exactly to u1, and integer grid coordinate *un* maps exactly to *u2*. All other integer grid coordinates *i* are mapped such that:

u = i(u2 u1)/un + u1

The **glMapGrid2** function specifies two such linear mappings. One maps integer grid coordinate *i = 0* exactly to *u1*, and integer grid coordinate *i = un* exactly to *u2*. The other maps integer grid coordinate *j = 0* exactly to *v1*, and integer grid coordinate *j = vn* exactly to *v2*. Other integer grid coordinates i and j are mapped such that

u = i(u2 u1)/un + u1

v = j (v2 v1)/vn + v1

The mappings specified by **glMapGrid** are used identically by glEvalMesh and **glEvalPoint**.

The following functions retrieve information related to **glMapGrid**:

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

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glEval

Coord

glEvalMesh

glEvalPoint

glMap1

glMap2

glMapGrid1f function

2/22/2020 • 2 minutes to read • Edit Online

Defines a one-dimensional mesh.

Syntax

```
void WINAPI glMapGrid1f(
   GLint   un,
   GLfloat u1,
   GLfloat u2
);
```

Parameters

un	The number of partitions in the grid range interval [u1, u2]. This value must be positive.
u1	A value used as the mapping for integer grid domain value i = 0.
u2	A value used as the mapping for integer grid domain value i = un.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	Either *un* or *vn* was not positive.

NAME	MEANING
GL_ INV ALI D_ OP ERA TIO N	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_ INV ALI D_V AL UE	Either *un* or *vn* was not positive.
GL_ INV ALI D_ OP ERA TIO N	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glMapGrid** and glEvalMesh functions are used in tandem to efficiently generate and evaluate a series of evenly spaced map domain values. The glEvalMesh function steps through the integer domain of a one- or two-dimensional grid, whose range is the domain of the evaluation maps specified by **glMap1** and **glMap2**.

The **glMapGrid1** and **glMapGrid2** functions specify the linear grid mappings between the i (or i and j) integer grid coordinates, to the u (or u and v) floating-point evaluation map coordinates. See **glMap1** and **glMap2** for details of how u and v coordinates are evaluated.

The **glMapGrid1** function specifies a single linear mapping such that integer grid coordinate 0 maps exactly to u1, and integer grid coordinate *un* maps exactly to *u2*. All other integer grid coordinates *i* are mapped such that:

u = i(u2 u1)/un + u1

The **glMapGrid2** function specifies two such linear mappings. One maps integer grid coordinate *i = 0* exactly to *u1*, and integer grid coordinate *i = un* exactly to *u2*. The other maps integer grid coordinate *j = 0* exactly to *v1*, and integer grid coordinate *j = vn* exactly to *v2*. Other integer grid coordinates i and j are mapped such that

u = i(u2 u1)/un + u1

v = j (v2 v1)/vn + v1

The mappings specified by **glMapGrid** are used identically by glEvalMesh and **glEvalPoint**.

The following functions retrieve information related to **glMapGrid**:

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

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib
DLL	Ope ngl3 2.dll

See also

g
l
B
e
g
i
n

g
l
E
n
d

g
l
E
v
a
l

Coord

glEvalMesh

glEvalPoint

glMap1

glMap2

glMapGrid2d function

2/22/2020 • 2 minutes to read • Edit Online

Defines a one-dimensional mesh.

Syntax

```
void WINAPI glMapGrid2d(
   GLint    un,
   GLdouble u1,
   GLdouble u2,
   GLint    vn,
   GLdouble v1,
   GLdouble v2
);
```

Parameters

un	The number of partitions in the grid range interval [u1, u2]. This value must be positive.
u1	A value used as the mapping for integer grid domain value i = 0.
u2	A value used as the mapping for integer grid domain value i = un.
vn	The number of partitions in the grid range interval [v1, v2].
v1	A value used as the mapping for integer grid domain value j = 0.
v2	A value used as the mapping for integer grid domain value j = vn.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	Either *un* or *vn* was not positive.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	Either *un* or *vn* was not positive.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glMapGrid** and glEvalMesh functions are used in tandem to efficiently generate and evaluate a series of evenly spaced map domain values. The glEvalMesh function steps through the integer domain of a one- or two-dimensional grid, whose range is the domain of the evaluation maps specified by **glMap1** and **glMap2**.

The **glMapGrid1** and **glMapGrid2** functions specify the linear grid mappings between the i (or i and j) integer grid coordinates, to the u (or u and v) floating-point evaluation map coordinates. See **glMap1** and **glMap2** for details of how u and v coordinates are evaluated.

The **glMapGrid1** function specifies a single linear mapping such that integer grid coordinate 0 maps exactly to u1, and integer grid coordinate *un* maps exactly to *u2*. All other integer grid coordinates *i* are mapped such that:

u = i(u2 u1)/un + u1

The **glMapGrid2** function specifies two such linear mappings. One maps integer grid coordinate *i = 0* exactly to *u1*, and integer grid coordinate *i = un* exactly to *u2*. The other maps integer grid coordinate *j = 0* exactly to *v1*, and integer grid coordinate *j = vn* exactly to *v2*. Other integer grid coordinates i and j are mapped such that

u = i(u2 u1)/un + u1

v = j (v2 v1)/vn + v1

The mappings specified by **glMapGrid** are used identically by glEvalMesh and **glEvalPoint**.

The following functions retrieve information related to **glMapGrid**:

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

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib
DLL	Ope ngl3 2.dll

See also

glBegin

glEnd

glEvalCoord

glEvalMesh

glEvalPoint

glMap1

glMap2

glMapGrid2f function

2/22/2020 • 2 minutes to read • Edit Online

Defines a one-dimensional mesh.

Syntax

```
void WINAPI glMapGrid2f(
   GLint   un,
   GLfloat u1,
   GLfloat u2,
   GLint   vn,
   GLfloat v1,
   GLfloat v2
);
```

Parameters

un	The number of partitions in the grid range interval [u1, u2]. This value must be positive.
u1	A value used as the mapping for integer grid domain value i = 0.
u2	A value used as the mapping for integer grid domain value i = un.
vn	The number of partitions in the grid range interval [v1, v2].
v1	A value used as the mapping for integer grid domain value j = 0.
v2	A value used as the mapping for integer grid domain value j = vn.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	Either *un* or *vn* was not positive.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	Either *un* or *vn* was not positive.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glMapGrid** and glEvalMesh functions are used in tandem to efficiently generate and evaluate a series of evenly spaced map domain values. The glEvalMesh function steps through the integer domain of a one- or two-dimensional grid, whose range is the domain of the evaluation maps specified by **glMap1** and **glMap2**.

The **glMapGrid1** and **glMapGrid2** functions specify the linear grid mappings between the i (or i and j) integer grid coordinates, to the u (or u and v) floating-point evaluation map coordinates. See **glMap1** and **glMap2** for details of how u and v coordinates are evaluated.

The **glMapGrid1** function specifies a single linear mapping such that integer grid coordinate 0 maps exactly to u1, and integer grid coordinate *un* maps exactly to *u2*. All other integer grid coordinates *i* are mapped such that:

u = i(u2 u1)/un + u1

The **glMapGrid2** function specifies two such linear mappings. One maps integer grid coordinate *i = 0* exactly to *u1*, and integer grid coordinate *i = un* exactly to *u2*. The other maps integer grid coordinate *j = 0* exactly to *v1*, and integer grid coordinate *j = vn* exactly to *v2*. Other integer grid coordinates i and j are mapped such that

u = i(u2 u1)/un + u1

v = j (v2 v1)/vn + v1

The mappings specified by **glMapGrid** are used identically by glEvalMesh and **glEvalPoint**.

The following functions retrieve information related to **glMapGrid**:

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

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glEvalCoord

glEvalMesh

glEvalPoint

glMap1

glMap2

glMaterial Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions specify material parameters for the lighting model:

- glMaterialf
- glMaterialI
- glMaterialfv
- glMaterialiv

glMaterialf function

2/22/2020 • 2 minutes to read • Edit Online

The **glMaterialf** function specifies material parameters for the lighting model.

Syntax

```
void WINAPI glMaterialf(
   GLenum  face,
   GLenum  pname,
   GLfloat param
);
```

Parameters

face

The face or faces that are being updated. Must be one of the following: GL_FRONT, GL_BACK, or GL_FRONT and GL_BACK.

pname

The single-valued material parameter of the face or faces being updated. Must be GL_SHININESS.

VALUE	MEANING
GL_SHININESS	The *param* parameter is a single floating-point value that specifies the RGBA specular exponent of the material. Integer values are mapped directly. Only values in the range [0, 128] are accepted. The default specular exponent for both front-facing and back-facing materials is 0.

param

The value to which parameter GL_SHININESS will be set.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	Either *face* or *pname* was not an accepted value.
GL_INVALID_VALUE	A specular exponent outside the range of [0, 128] was specified.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	Either *face* or *pname* was not an accepted value.
GL_INVALID_VALUE	A specular exponent outside the range of [0, 128] was specified.

Remarks

The **glMaterialf** function assigns values to material parameters. There are two matched sets of material parameters. One, the *front-facing* set, is used to shade points, lines, bitmaps, and all polygons (when two-sided lighting is disabled), or just front-facing polygons (when two-sided lighting is enabled). The other set, *back-facing*, is used to shade back-facing polygons only when two-sided lighting is enabled. Refer to **glLightModel** for details concerning one-sided and two-sided lighting calculations.

The **glMaterialf** function takes three arguments. The first, *face*, specifies whether the GL_FRONT materials, the GL_BACK materials, or both GL_FRONT_AND_BACK materials will be modified. The second, *pname*, specifies which of several parameters in one or both sets will be modified. The third, *param*, specifies what value will be assigned to the specified parameter.

Material parameters are used in the lighting equation that is optionally applied to each vertex. The equation is discussed in **glLightModel**.

The material parameters can be updated at any time. In particular, **glMaterialf** can be called between a call to

glBegin and the corresponding call to glEnd. If only a single material parameter is to be changed per vertex, however, glColorMaterial is preferred over **glMaterialf**.

The following function retrieves information related to **glMaterialf**:

glGetMaterial

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glColorMaterial

glLight

g
l
L
i
g
h
t
M
o
d
e
l

glMaterialfv function

2/22/2020 • 3 minutes to read • Edit Online

The **glMaterialfv** function specifies material parameters for the lighting model.

Syntax

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

Parameters

face

The face or faces that are being updated. Must be one of the following: GL_FRONT, GL_BACK, or GL_FRONT and GL_BACK.

pname

The material parameter of the face or faces being updated. The parameters that can be specified using **glMaterialfv**, and their interpretations by the lighting equation, are as follows.

VALUE	MEANING
GL_AMBIENT	The params parameter contains four floating-point values that specify the ambient RGBA reflectance of the material. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default ambient reflectance for both front-facing and back-facing materials is (0.2, 0.2, 0.2, 1.0).

VALUE	MEANING
GL_DIFFUSE	The params parameter contains four floating-point values that specify the diffuse RGBA reflectance of the material. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default diffuse reflectance for both front-facing and back-facing materials is (0.8, 0.8, 0.8, 1.0).
GL_SPECULAR	The params parameter contains four floating-point values that specify the specular RGBA reflectance of the material. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default specular reflectance for both front-facing and back-facing materials is (0.0, 0.0, 0.0, 1.0).
GL_EMISSION	The params parameter contains four floating-point values that specify the RGBA emitted light intensity of the material. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default emission intensity for both front-facing and back-facing materials is (0.0, 0.0, 0.0, 1.0).
GL_SHININESS	The *param* parameter is a single integer value that specifies the RGBA specular exponent of the material. Integer values are mapped directly. Only values in the range [0, 128] are accepted. The default specular exponent for both front-facing and back-facing materials is 0.
GL_AMBIENT_AND_DIFFUSE	Equivalent to calling **glMaterial** twice with the same parameter values, once with GL_AMBIENT and once with GL_DIFFUSE.
GL_COLOR_INDEXES	The params parameter contains three floating-point values specifying the color indexes for ambient, diffuse, and specular lighting. These three values, and GL_SHININESS, are the only material values used by the color-index mode lighting equation. Refer to **glLightModel** for a discussion of color-index lighting.

pa The value to which parameter GL_SHININESS will be set.

r
a
m
s

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	Either *face* or *pname* was not an accepted value.
GL_INVALID_VALUE	A specular exponent outside the range of [0, 128] was specified.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	Either *face* or *pname* was not an accepted value.
GL_INVALID_VALUE	A specular exponent outside the range of [0, 128] was specified.

Remarks

The glMaterialfv function assigns values to material parameters. There are two matched sets of material parameters. One, the *front-facing* set, is used to shade points, lines, bitmaps, and all polygons (when two-sided lighting is disabled), or just front-facing polygons (when two-sided lighting is enabled). The other set, *back-facing*, is used to shade back-facing polygons only when two-sided lighting is enabled. Refer to glLightModel for details concerning one-sided and two-sided lighting calculations.

The glMaterialfv function takes three arguments. The first, *face*, specifies whether the GL_FRONT materials, the GL_BACK materials, or both GL_FRONT_AND_BACK materials will be modified. The second, *pname*, specifies which of several parameters in one or both sets will be modified. The third, *param*, specifies what value will be assigned to the specified parameter.

Material parameters are used in the lighting equation that is optionally applied to each vertex. The equation is discussed in glLightModel.

The material parameters can be updated at any time. In particular, glMaterialfv can be called between a call to glBegin and the corresponding call to glEnd. If only a single material parameter is to be changed per vertex, however, glColorMaterial is preferred over glMaterialfv.

The following function retrieves information related to glMaterialfv:

glGetMaterial

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glColorMat

e
r
i
a
l

g
l
L
i
g
h
t

g
l
L
i
g
h
t
M
o
d
e
l

glMateriali function

2/22/2020 • 2 minutes to read • Edit Online

The**glMateriali** function specifies material parameters for the lighting model.

Syntax

```
void WINAPI glMateriali(
   GLenum face,
   GLenum pname,
   GLint  param
);
```

Parameters

face

The face or faces that are being updated. Must be one of the following: GL_FRONT, GL_BACK, or GL_FRONT and GL_BACK.

pname

The single-valued material parameter of the face or faces being updated. Must be GL_SHININESS.

VALUE	MEANING
GL_SHININESS	The *param* parameter is a single integer that specifies the RGBA specular exponent of the material. Integer values are mapped directly. Only values in the range [0, 128] are accepted. The default specular exponent for both front-facing and back-facing materials is 0.

param

The value to which parameter GL_SHININESS will be set.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	Either *face* or *pname* was not an accepted value.
GL_INVALID_VALUE	A specular exponent outside the range of [0, 128] was specified.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	Either *face* or *pname* was not an accepted value.
GL_INVALID_VALUE	A specular exponent outside the range of [0, 128] was specified.

Remarks

The **glMaterialiv** function assigns values to material parameters. There are two matched sets of material parameters. One, the *front-facing* set, is used to shade points, lines, bitmaps, and all polygons (when two-sided lighting is disabled), or just front-facing polygons (when two-sided lighting is enabled). The other set, *back-facing*, is used to shade back-facing polygons only when two-sided lighting is enabled. Refer to **glLightModel** for details concerning one-sided and two-sided lighting calculations.

The **glMaterialiv** function takes three arguments. The first, *face*, specifies whether the GL_FRONT materials, the GL_BACK materials, or both GL_FRONT_AND_BACK materials will be modified. The second, *pname*, specifies which of several parameters in one or both sets will be modified. The third, *param*, specifies what value will be assigned to the specified parameter.

Material parameters are used in the lighting equation that is optionally applied to each vertex. The equation is discussed in **glLightModel**.

The material parameters can be updated at any time. In particular, **glMaterialiv** can be called between a call to

glBegin and the corresponding call to glEnd. If only a single material parameter is to be changed per vertex, however, glColorMaterial is preferred over **glMaterialf**.

The following function retrieves information related to **glMaterialf**:

glGetMaterial

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glColorMaterial

glLight

g
l
L
i
g
h
t
M
o
d
e
l

glMaterialiv function

2/22/2020 • 3 minutes to read • Edit Online

The **glMaterialiv** function specifies material parameters for the lighting model.

Syntax

```
void WINAPI glMaterialfv(
         GLenum face,
         GLenum pname,
   const GLint  *params
);
```

Parameters

face

The face or faces that are being updated. Must be one of the following: GL_FRONT, GL_BACK, or GL_FRONT and GL_BACK.

pname

The material parameter of the face or faces being updated. The parameters that can be specified using glMaterialiv, and their interpretations by the lighting equation, are as follows.

VALUE	MEANING
GL_AMBIENT	The params parameter contains four integer values that specify the ambient RGBA reflectance of the material. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default ambient reflectance for both front-facing and back-facing materials is (0.2, 0.2, 0.2, 1.0).
GL_DIFFUSE	The params parameter contains four integer values that specify the diffuse RGBA reflectance of the material. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default diffuse reflectance for both front-facing and back-facing materials is (0.8, 0.8, 0.8, 1.0).

VALUE	MEANING
GL_SPECULAR	The params parameter contains four integer values that specify the specular RGBA reflectance of the material. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default specular reflectance for both front-facing and back-facing materials is (0.0, 0.0, 0.0, 1.0).
GL_EMISSION	The params parameter contains four integer values that specify the RGBA emitted light intensity of the material. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default emission intensity for both front-facing and back-facing materials is (0.0, 0.0, 0.0, 1.0).
GL_SHININESS	The *param* parameter is a single integer that specifies the RGBA specular exponent of the material. Integer values are mapped directly. Only values in the range [0, 128] are accepted. The default specular exponent for both front-facing and back-facing materials is 0.
GL_AMBIENT_AND_DIFFUSE	Equivalent to calling **glMaterial** twice with the same parameter values, once with GL_AMBIENT and once with GL_DIFFUSE.
GL_COLOR_INDEXES	The params parameter contains three integer values specifying the color indexes for ambient, diffuse, and specular lighting. These three values, and GL_SHININESS, are the only material values used by the color-index mode lighting equation. Refer to **glLightModel** for a discussion of color-index lighting.

params The value to which parameter GL_SHININESS will be set.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	Either *face* or *pname* was not an accepted value.
GL_INVALID_VALUE	A specular exponent outside the range of [0, 128] was specified.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	Either *face* or *pname* was not an accepted value.
GL_INVALID_VALUE	A specular exponent outside the range of [0, 128] was specified.

Remarks

The glMaterialiv function assigns values to material parameters. There are two matched sets of material parameters. One, the *front-facing* set, is used to shade points, lines, bitmaps, and all polygons (when two-sided lighting is disabled), or just front-facing polygons (when two-sided lighting is enabled). The other set, *back-facing*, is used to shade back-facing polygons only when two-sided lighting is enabled. Refer to glLightModel for details concerning one-sided and two-sided lighting calculations.

The glMaterialiv function takes three arguments. The first, *face*, specifies whether the GL_FRONT materials, the GL_BACK materials, or both GL_FRONT_AND_BACK materials will be modified. The second, *pname*, specifies which of several parameters in one or both sets will be modified. The third, *param*, specifies what value will be assigned to the specified parameter.

Material parameters are used in the lighting equation that is optionally applied to each vertex. The equation is discussed in glLightModel.

The material parameters can be updated at any time. In particular, glMaterialiv can be called between a call to glBegin and the corresponding call to glEnd. If only a single material parameter is to be changed per vertex, however, glColorMaterial is preferred over glMaterialiv.

The following function retrieves information related to glMaterialiv:

glGetMaterial

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glColorMaterial

glLi

g
h
t

g
l
L
i
g
h
t
M
o
d
e
l

glMatrixMode function

2/22/2020 • 2 minutes to read • Edit Online

The **glMatrixMode** function specifies which matrix is the current matrix.

Syntax

```
void WINAPI glMatrixMode(
   GLenum mode
);
```

Parameters

m
o
d
e

The matrix stack that is the target for subsequent matrix operations. The *mode* parameter can assume one of three values.

VALUE	MEANING
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.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*mode* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*mode* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glMatrixMode** function sets the current matrix mode.

The following function retrieves information related to **glMatrixMode**:

glGet with argument GL_MATRIX_MODE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glLoadMatrix

glPus

h
M
a
t
r
i
x

glMultMatrix Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions multiply the current matrix by an arbitrary matrix:

- glMultMatrixd
- glMultMatrixf

glMultMatrixd function

2/22/2020 • 2 minutes to read • Edit Online

The **glMultMatrixd** and glMultMatrixf functions multiply the current matrix by an arbitrary matrix.

Syntax

```
void WINAPI glMultMatrixd(
   const GLdouble *m
);
```

Parameters

m	A pointer to a 4x4 matrix stored in column-major order as 16 consecutive values.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Error codes

The following error code can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Remarks

The **glMultMatrix** function multiplies the current matrix by the one specified in *m*. That is, if M is the current matrix and T is the matrix passed to **glMultMatrix**, then M is replaced with M T.

The current matrix is the projection matrix, modelview matrix, or texture matrix, determined by the current matrix mode (see **glMatrixMode**).

The *m* parameter points to a 4x4 matrix of single-precision or double-precision floating-point values stored in column-major order. That is, the matrix is stored as shown in the following image.

$$\begin{pmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{pmatrix}$$

The following functions retrieve information related to **glMultMatrix**:

glGet with argument GL_MATRIX_MODE

glGet with argument GL_MODELVIEW_MATRIX

glGet with argument GL_PROJECTION_MATRIX

glGet with argument GL_TEXTURE_MATRIX

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

g
l
E
n
d

g
l
L
o
a
d
I
d
e
n
t
i
t
y

g
l
L
o
a
d
M
a
t
r
i
x

g
l
M
a
t
r
i
x
M
o
d
e

g
l
P
u

s
h
M
a
t
r
i
x

glMultMatrixf function

2/22/2020 • 2 minutes to read • Edit Online

The **glMultMatrixd** and **glMultMatrixf** functions multiply the current matrix by an arbitrary matrix.

Syntax

```
void WINAPI glMultMatrixf(
   const GLfloat *m
);
```

Parameters

m	A pointer to a 4x4 matrix stored in column-major order as 16 consecutive values.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glMultMatrix** function multiplies the current matrix by the one specified in *m*. That is, if M is the current matrix and T is the matrix passed to **glMultMatrix**, then M is replaced with M T.

The current matrix is the projection matrix, modelview matrix, or texture matrix, determined by the current matrix mode (see **glMatrixMode**).

The *m* parameter points to a 4x4 matrix of single-precision or double-precision floating-point values stored in column-major order. That is, the matrix is stored as shown in the following image.

$$\begin{pmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{pmatrix}$$

The following functions retrieve information related to **glMultMatrix**:

glGet with argument GL_MATRIX_MODE

glGet with argument GL_MODELVIEW_MATRIX

glGet with argument GL_PROJECTION_MATRIX

glGet with argument GL_TEXTURE_MATRIX

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

g
l
E
n
d

g
l
L
o
a
d
I
d
e
n
t
i
t
y

g
l
L
o
a
d
M
a
t
r
i
x

g
l
M
a
t
r
i
x
M
o
d
e

g
l
P
u

s
h
M
a
t
r
i
x

glNewList function

2/22/2020 • 2 minutes to read • Edit Online

The **glNewList** and glEndList functions create or replace a display list.

Syntax

```
void WINAPI glNewList(
   GLuint list,
   GLenum mode
);
```

Parameters

list — The display list name.

mode — The compilation mode. The following values are accepted.

VALUE	MEANING
GL_COMPILE	Commands are merely compiled.
GL_COMPILE_AND_EXECUTE	Commands are executed as they are compiled into the display list.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_VALUE	*list* was zero.
GL_INVALID_ENUM	*mode* was not an accepted value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	*list* was zero.
GL_INVALID_ENUM	*mode* was not an accepted value.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Remarks

Display lists are groups of OpenGL commands that have been stored for subsequent execution. The display lists are created with **glNewList**. All subsequent commands are placed in the display list, in the order issued, until **glEndList** is called.

The **glNewList** function has two parameters. The first parameter, *list*, is a positive integer that becomes the unique name for the display list. Names can be created and reserved with glGenLists and tested for uniqueness with glIsList. The second parameter, *mode*, is a symbolic constant that can assume one of the two preceding values.

Certain commands are not compiled into the display list, but are executed immediately, regardless of the display list mode. These commands are glColorPointer, glDeleteLists, glDisableClientState, glEdgeFlagPointer, glEnableClientState, glFeedbackBuffer, glFinish, glFlush, glGenLists, glIndexPointer, glInterleavedArrays, glIsEnabled, glIsList, glNormalPointer, glPopClientAttrib, glPixelStore, glPushClientAttrib, glReadPixels, glRenderMode, glSelectBuffer, glTexCoordPointer, glVertexPointer, and all of the glGet routines.

Similarly, glTexImage2D and glTexImage1D are executed immediately and not compiled into the display list when their first argument is GL_PROXY_TEXTURE_2D or GL_PROXY_TEXTURE_1D, respectively.

When the **glEndList** function is encountered, the display list definition is completed by associating the list with the unique name *list* (specified in the **glNewList** command). If a display list with name *list* already exists, it is replaced only when **glEndList** is called.

The glCallList and glCallLists functions can be entered into display lists. The commands in the display list or lists executed by **glCallList** or **glCallLists** are not included in the display list being created, even if the list creation mode is GL_COMPILE_AND_EXECUTE.

The following function retrieves information related to **glNewList**:

glGet with argument GL_MATRIX_MODE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glCallList

glCallLists

glDelete

L
i
s
t
s

g
l
E
n
d

g
l
E
n
d
L
i
s
t

g
l
G
e
n
L
i
s
t
s

g
l
l
s
L
i
s
t

glNormal Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions set the current normal vector:

- glNormal3b
- glNormal3d
- glNormal3f
- glNormal3i
- glNormal3s
- glNormal3bv
- glNormal3dv
- glNormal3fv
- glNormal3iv
- glNormal3sv

glNormal3b function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current normal vector.

Syntax

```
void WINAPI glNormal3b(
   GLbyte nx,
   GLbyte ny,
   GLbyte nz
);
```

Parameters

nx	Specifies the x-coordinate for the new current normal vector.
ny	Specifies the y-coordinate for the new current normal vector.
nz	Specifies the z-coordinate for the new current normal vector.

Return value

This function does not return a value.

Remarks

The current normal is set to the given coordinates whenever you call the **glNormal3b** function.

Byte, short, or integer arguments are converted to floating-point format by using 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 **glNormal3b** need not have unit length. If normalization is enabled, then normals specified with **glNormal3b** are normalized after transformation. You can control normalization by using **glEnable** and **glDisable** with the argument GL_NORMALIZE. By default, normalization is disabled. You can update the current normal at any time. In particular, you can call**glNormal3b**between a call to **glBegin** and the corresponding call to **glEnd**. The following functions retrieve information related to **glNormal3b**:

glGet with argument GL_CURRENT_NORMAL

glIsEnable with argument GL_NORMALIZE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glEnd

glIndex

glTe

x
C
o
o
r
d

g
l
V
e
r
t
e
x

glNormal3bv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current normal vector.

Syntax

```
void WINAPI glNormal3bv(
   const GLbyte *v
);
```

Parameters

v	A pointer to an array of three elements: the x, y, and z coordinates of the new current normal.

Return value

This function does not return a value.

Remarks

The current normal is set to the given coordinates whenever you call the **glNormal3bv** function.

Byte, short, or integer arguments are converted to floating-point format by using 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 by using **glNormal3bv** need not have unit length. If normalization is enabled, then normals specified by using **glNormal3bv** are normalized after transformation. You can control normalization by using **glEnable** and **glDisable** with the argument GL_NORMALIZE. By default, normalization is disabled. You can update the current normal any time. In particular, you can call **glNormal3bv** between a call to **glBegin** and the corresponding call to **glEnd**. The following functions retrieve information related to **glNormal3bv**:

glGet with argument GL_CURRENT_NORMAL

glIsEnable with argument GL_NORMALIZE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glEnd

glIndex

glTexCoor

d

g
l
V
e
r
t
e
x

glNormal3d function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current normal vector.

Syntax

```
void WINAPI glNormal3d(
   GLdouble nx,
   GLdouble ny,
   GLdouble nz
);
```

Parameters

nx	Specifies the x-coordinate for the new current normal vector.
ny	Specifies the y-coordinate for the new current normal vector.
nz	Specifies the z-coordinate for the new current normal vector.

Return value

This function does not return a value.

Remarks

The current normal is set to the given coordinates whenever you call the **glNormal3d**function.

Byte, short, or integer arguments are converted to floating-point format by using 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 by using**glNormal3d** need not have unit length. If normalization is enabled, then normals specified with **glNormal3d** are normalized after transformation. You can control normalization by using **glEnable** and **glDisable** with the argument GL_NORMALIZE. By default, normalization is disabled. You can update the current normal at any time. In particular, you can call**glNormal3d**between a call to **glBegin** and the corresponding call to **glEnd**. The following functions retrieve information related to **glNormal3d**:

glGet with argument GL_CURRENT_NORMAL

glIsEnable with argument GL_NORMALIZE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glEnd

glIndex

glTe

x
C
o
o
r
d

g
l
V
e
r
t
e
x

glNormal3dv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current normal vector.

Syntax

```
void WINAPI glNormal3dv(
   const GLdouble *v
);
```

Parameters

v	A pointer to an array of three elements: the x, y, and z coordinates of the new current normal.

Return value

This function does not return a value.

Remarks

The current normal is set to the given coordinates whenever you call the **glNormal3dv** function.

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 by using **glNormal3dv** need not have unit length. If normalization is enabled, then normals specified with **glNormal3dv** are normalized after transformation. You can control normalization by using **glEnable** and **glDisable** with the argument GL_NORMALIZE. By default, normalization is disabled. You can update the current normal at any time. In particular, you can call **glNormal3dv** between a call to **glBegin** and the corresponding call to **glEnd**. The following functions retrieve information related to **glNormal3dv**:

glGet with argument GL_CURRENT_NORMAL

glIsEnable with argument GL_NORMALIZE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glEnd

glIndex

glTexCoor

d

g
l
V
e
r
t
e
x

glNormal3f function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current normal vector.

Syntax

```
void WINAPI glNormal3f(
   GLfloat nx,
   GLfloat ny,
   GLfloat nz
);
```

Parameters

nx	Specifies the x-coordinate for the new current normal vector.
ny	Specifies the y-coordinate for the new current normal vector.
nz	Specifies the z-coordinate for the new current normal vector.

Return value

This function does not return a value.

Remarks

The current normal is set to the given coordinates whenever you call the **glNormal3f** function.

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 by using **glNormal3f** need not have unit length. If normalization is enabled, then normals specified with **glNormal3f** are normalized after transformation. You can control normalization by using **glEnable** and **glDisable** with the argument GL_NORMALIZE. By default, normalization is disabled. You can update the current normal at any time. In particular, you can call **glNormal3f** between a call to **glBegin** and the corresponding call to **glEnd**. The following functions retrieve information related to **glNormal3f**:

glGet with argument GL_CURRENT_NORMAL

glIsEnable with argument GL_NORMALIZE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glEnd

glIndex

glTe

x
C
o
o
r
d

g
l
V
e
r
t
e
x

glNormal3fv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current normal vector.

Syntax

```
void WINAPI glNormal3fv(
   const GLfloat *v
);
```

Parameters

v	A pointer to an array of three elements: the x, y, and z coordinates of the new current normal.

Return value

This function does not return a value.

Remarks

The current normal is set to the given coordinates whenever you call the **glNormal3fv** function.

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 by using **glNormal3fv** need not have unit length. If normalization is enabled, then normals specified with **glNormal3fv** are normalized after transformation. You can control normalization by using **glEnable** and **glDisable** with the argument GL_NORMALIZE. By default, normalization is disabled. You can update the current normal can at any time. In particular, you can call **glNormal3fv** between a call to **glBegin** and the corresponding call to **glEnd**. The following functions retrieve information related to **glNormal3fv**:

glGet with argument GL_CURRENT_NORMAL

glIsEnable with argument GL_NORMALIZE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glEnd

glIndex

glTexCoor

d

g
l
V
e
r
t
e
x

glNormal3i function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current normal vector.

Syntax

```
void WINAPI glNormal3i(
   GLint nx,
   GLint ny,
   GLint nz
);
```

Parameters

nx	Specifies the x-coordinate for the new current normal vector.
ny	Specifies the y-coordinate for the new current normal vector.
nz	Specifies the z-coordinate for the new current normal vector.

Return value

This function does not return a value.

Remarks

The current normal is set to the given coordinates whenever you call the **glNormal3i** function.

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 by using **glNormal3i** need not have unit length. If normalization is enabled, then normals specified with **glNormal3i** are normalized after transformation. You can control normalization by using glEnable and glDisable with the argument GL_NORMALIZE. By default, normalization is disabled. You can update the current normal at any time. In particular, you can call **glNormal3i** between a call to glBegin and the corresponding call to glEnd. The following functions retrieve information related to **glNormal3i**:

glGet with argument GL_CURRENT_NORMAL

glIsEnable with argument GL_NORMALIZE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glEnd

glIndex

glTe

x
C
o
o
r
d

g
l
V
e
r
t
e
x

glNormal3iv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current normal vector.

Syntax

```
void WINAPI glNormal3iv(
   const GLint *v
);
```

Parameters

v	A pointer to an array of three elements: the x, y, and z coordinates of the new current normal.

Return value

This function does not return a value.

Remarks

The current normal is set to the given coordinates whenever you call the **glNormal3iv** function.

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 by using **glNormal3iv** need not have unit length. If normalization is enabled, then normals specified with **glNormal3iv** are normalized after transformation. You can control normalization by using **glEnable** and **glDisable** with the argument GL_NORMALIZE. By default, normalization is disabled. You can update the current normal at any time. In particular, you can call **glNormal3iv** between a call to **glBegin** and the corresponding call to **glEnd**. The following functions retrieve information related to **glNormal3iv**:

glGet with argument GL_CURRENT_NORMAL

glIsEnable with argument GL_NORMALIZE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glEnd

glIndex

glTexCoor

d

g
l
V
e
r
t
e
x

glNormal3s function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current normal vector.

Syntax

```
void WINAPI glNormal3s(
   GLshort nx,
   GLshort ny,
   GLshort nz
);
```

Parameters

nx	Specifies the x-coordinate of the new current normal vector.
ny	Specifies the y-coordinate of the new current normal vector.
nz	Specifies the z-coordinate of the new current normal vector.

Return value

This function does not return a value.

Remarks

The current normal is set to the given coordinates whenever you call the **glNormal3s** function.

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 by using **glNormal3s** need not have unit length. If normalization is enabled, then normals specified with **glNormal3s** are normalized after transformation. You can control normalization by using **glEnable** and **glDisable** with the argument GL_NORMALIZE. By default, normalization is disabled. You can update the current normal at any time. In particular, you can call **glNormal3s** between a call to **glBegin** and the corresponding call to **glEnd**. The following functions retrieve information related to **glNormal3s**:

glGet with argument GL_CURRENT_NORMAL

glIsEnable with argument GL_NORMALIZE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glEnd

glIndex

glTe

x
C
o
o
r
d

g
l
V
e
r
t
e
x

glNormal3sv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current normal vector.

Syntax

```
void WINAPI glNormal3sv(
   const GLshort *v
);
```

Parameters

v	A pointer to an array of three elements: the x, y, and z coordinates of the new current normal.

Return value

This function does not return a value.

Remarks

The current normal is set to the given coordinates whenever you call the **glNormal3sv** function.

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 by using **glNormal3sv** need not have unit length. If normalization is enabled, then normals specified with **glNormal3sv** are normalized after transformation. You can control normalization by using **glEnable** and **glDisable** with the argument GL_NORMALIZE. By default, normalization is disabled. You can update the current normal at any time. In particular, you can call **glNormal3sv** between a call to **glBegin** and the corresponding call to **glEnd**. The following functions retrieve information related to **glNormal3sv**:

glGet with argument GL_CURRENT_NORMAL

glIsEnable with argument GL_NORMALIZE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glEnd

glIndex

glTexCoor

d

g
l
V
e
r
t
e
x

glNormalPointer function

2/22/2020 • 2 minutes to read • Edit Online

The **glNormalPointer** function defines an array of normals.

Syntax

```
void WINAPI glNormalPointer(
         GLenum  type,
         GLsizei stride,
   const GLvoid  *pointer
);
```

Parameters

type

The data type of each coordinate in the array using the following symbolic constants: GL_BYTE, GL_SHORT, GL_INT, GL_FLOAT, and GL_DOUBLE.

stride

The byte offset between consecutive normals. When *stride* is zero, the normals are tightly packed in the array.

pointer

A pointer to the first normal in the array.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*type* was not an accepted value.
GL_INVALID_OPERATION	*stride* or *count* was negative.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*type* was not an accepted value.
GL_INVALID_OPERATION	*stride* or *count* was negative.

Remarks

The **glNormalPointer** function specifies the location and data of an array of normals to use when rendering. The *type* parameter specifies the data type of each normal coordinate. The *stride* parameter determines the byte offset from one normal to the next, enabling the packing of vertices and attributes in a single array or storage in separate arrays. In some implementations storing the vertices and attributes in a single array can be more efficient than using separate arrays; see **glInterleavedArrays** for details.

A normal array is enabled when you specify the GL_NORMAL_ARRAY constant with **glEnableClientState**. When enabled, **glDrawArrays**, **glDrawElements** and **glArrayElement** use the normal array. By default the normal array is disabled.

You cannot include **glNormalPointer** in display lists.

When you specify a normal array using **glNormalPointer**, the values of all the function's normal array parameters are saved in a client-side state. Because the normal array parameters are saved in a client-side state, their values are not saved or restored by glPushAttrib and glPopAttrib.

Although no error is generated when you call **glNormalPointer** within glBegin and glEnd pairs, the results are undefined.

The following functions are associated with **glNormalPointer**:

glGet with argument GL_NORMAL_ARRAY_STRIDE

glGet with argument GL_NORMAL_ARRAY_COUNT

glGet with argument GL_NORMAL_ARRAY_TYPE

glGetPointerv with argument GL_NORMAL_ARRAY_POINTER

glIsEnabled with argument GL_NORMAL_ARRAY

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glArrayElement

glColorPointer

glDrawElements

glDrawArrays

glEnable

glEdgeFlagPointer

glGetPointerv

glIndexPointer

glIsEn

a
b
l
e
d

g
l
I
n
t
e
r
l
e
a
v
e
d
A
r
r
a
y
s

g
l
T
e
x
C
o
o
r
d
P
o
i
n
t
e
r

g
l
V
e
r
t
e
x
P

o
i
n
t
e
r

g
l
G
e
t
S
t
r
i
n
g

glOrtho function

2/22/2020 • 2 minutes to read • Edit Online

The **glOrtho** function multiplies the current matrix by an orthographic matrix.

Syntax

```
void WINAPI glOrtho(
   GLdouble left,
   GLdouble right,
   GLdouble bottom,
   GLdouble top,
   GLdouble zNear,
   GLdouble zFar
);
```

Parameters

left

The coordinates for the left vertical clipping plane.

right

The coordinates for theright vertical clipping plane.

bottom

The coordinates for the bottom horizontal clipping plane.

top

The coordinates for the top horizontal clipping plans.

zNear

The distances to the nearer depth clipping plane. This distance is negative if the plane is to be behind the viewer.

z

F
a
r

The distances to the farther depth clipping plane. This distance is negative if the plane is to be behind the viewer.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glOrtho** function describes a perspective matrix that produces a parallel projection. The (*left*, *bottom*, *near*) and (*right*, *top*, *near*) parameters specify the points on the near clipping plane that are mapped to the lower-left and upper-right corners of the window, respectively, assuming that the eye is located at (0, 0, 0). The *far* parameter specifies the location of the far clipping plane. Both *zNear* and *zFar* can be either positive or negative. The corresponding matrix is shown in the following image.

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

where

$$t_x = -\frac{right+left}{right-left}$$

$$t_y = -\frac{top+bottom}{top-bottom}$$

$$t_z = -\frac{far+near}{far-near}$$

The current matrix is multiplied by this matrix with the result replacing the current matrix. That is, if M is the current matrix and O is the ortho matrix, then M is replaced with M O.

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

The following functions retrieve information related to **glOrtho**:

glGet with argument GL_MATRIX_MODE

glGet with argument GL_MODELVIEW_MATRIX

glGet with argument GL_PROJECTION_MATRIX

glGet with argument GL_TEXTURE_MATRIX

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glFrustum

glMatrixMode

glMultMatrix

g

lPushMatrix

glViewport

glPassThrough function

2/22/2020 • 2 minutes to read • Edit Online

The **glPassThrough** function places a marker in the feedback buffer.

Syntax

```
void WINAPI glPassThrough(
   GLfloat token
);
```

Parameters

t
o
k
e
n

A marker value to be placed in the feedback buffer. It is indicated with the following unique identifying value.

VALUE	MEANING
GL_ PAS S_T HR OU GH _TO KEN	The order of **glPassThrough** commands with respect to the specification of graphics primitives is maintained.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

Feedback is an OpenGL render mode selected by calling **glRenderMode** with GL_FEEDBACK. When OpenGL is in feedback mode, no pixels are produced by rasterization. Instead, information about primitives that would have been rasterized is fed back to the application by OpenGL. See **glFeedbackBuffer** for a description of the feedback buffer and the values in it.

The **glPassThrough** function inserts a user-defined marker in the feedback buffer when it is executed in feedback mode. The *token* parameter is returned as if it were a primitive.

The **glPassThrough** function is ignored if OpenGL is not in feedback mode.

The following function retrieves information related to **glPassThrough**:

glGet with argument GL_RENDER_MODE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glFeedbackBuffer

glRender

Mode

glPixelMap Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions set up pixel transfer maps:

- **glPixelMapfv**
- **glPixelMapuiv**
- **glPixelMapusv**

glPixelMapfv function

2/22/2020 • 3 minutes to read • Edit Online

The **glPixelMapfv** function sets up pixel transfer maps.

Syntax

```
void WINAPI glPixelMapfv(
         GLenum  map,
         GLsizei mapsize,
   const GLfloat *values
);
```

Parameters

m
a
p

A symbolic map name. The ten maps are as follows.

VALUE	MEANING
GL_PIXEL_MAP_I_TO_I	Maps color indexes to color indexes.
GL_PIXEL_MAP_S_TO_S	Maps stencil indexes to stencil indexes.
GL_PIXEL_MAP_I_TO_R	Maps color indexes to red components.

VALUE	MEANING
GL_PIXEL_MAP_I_TO_G	Maps color indexes to green components.
GL_PIXEL_MAP_I_TO_B	Maps color indexes to blue components.
GL_PIXEL_MAP_I_TO_A	Maps color indexes to alpha components.
GL_PIXEL_MAP_R_TO_R	Maps red components to red components.
GL_PIXEL_MAP_G_TO_G	Maps green components to green components.
GL_PIXEL_MAP_B_TO_B	Maps blue components to blue components.

VALUE	MEANING
GL_PIXEL_MAP_A_TO_A	Maps alpha components to alpha components.

mapsize The size of the map being defined.

values An array of *mapsize* values.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*map* was not an accepted value.
GL_INVALID_VALUE	*mapsize* was negative or larger than GL_PIXEL_MAP_TABLE.

NAME	MEANING
GL_INVALID_VALUE	*map* was GL_PIXEL_MAP_I_TO_I, GL_PIXEL_MAP_S_TO_S, GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, or GL_PIXEL_MAP_I_TO_A, and *mapsize* was not a power of two.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*map* was not an accepted value.
GL_INVALID_VALUE	*mapsize* was negative or larger than GL_PIXEL_MAP_TABLE.
GL_INVALID_VALUE	*map* was GL_PIXEL_MAP_I_TO_I, GL_PIXEL_MAP_S_TO_S, GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, or GL_PIXEL_MAP_I_TO_A, and *mapsize* was not a power of two.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glPixelMap** function sets up translation tables, or *maps*, used by **glCopyPixels**, **glCopyTexImage1D**, **glCopyTexImage2D**, **glCopyTexSubImage1D**, **glCopyTexSubImage2D**, **glDrawPixels**, **glReadPixels**, **glTexImage1D**, **glTexImage2D**, **glTexSubImage1D**, and **glTexSubImage2D**. Use of these maps is described completely in the **glPixelTransfer** topic, and partly in the topics for the pixel and texture image commands. Only the specification of the maps is described in this topic.

The *map* parameter is a symbolic map name, indicating one of ten maps to set. The *mapsize* parameter specifies the number of entries in the map, and *values* is a pointer to an array of *mapsize* map values.

The entries in a map can be specified as single-precision floating-point numbers, unsigned short integers, or unsigned long integers. Maps that store color component values (all but GL_PIXEL_MAP_I_TO_I and GL_PIXEL_MAP_S_TO_S) retain their values in floating-point format, with unspecified mantissa and exponent sizes. Floating-point values specified by **glPixelMapfv** are converted directly to the internal floating-point format of these maps, and then clamped to the range [0,1]. Unsigned integer values specified by **glPixelMapusv** and **glPixelMapuiv** are converted linearly such that the largest representable integer maps to 1.0, and zero maps to 0.0.

Maps that store indexes, GL_PIXEL_MAP_I_TO_I and GL_PIXEL_MAP_S_TO_S, retain their values in fixed-point format, with an unspecified number of bits to the right of the binary point. Floating-point values specified by **glPixelMapfv** are converted directly to the internal fixed-point format of these maps. Unsigned integer values specified by **glPixelMapusv** and **glPixelMapuiv** specify integer values, with all zeros to the right of the binary point.

The following table shows the initial sizes and values for each of the maps. Maps that are indexed by either color or stencil indexes must have *mapsize* = 2 ^ *n* for some *n* or results are undefined. The maximum allowable size for each map depends on the implementation and can be determined by calling **glGet** with argument GL_MAX_PIXEL_MAP_TABLE. The single maximum applies to all maps, and it is at least 32.

MAP	LOOKUP INDEX	LOOKUP VALUE	INITIAL SIZE	INITIAL VALUE
GL_PIXEL_MAP_I_TO_I	color index	color index	1	0.0
GL_PIXEL_MAP_S_TO_S	stencil index	stencil index	1	0.0
GL_PIXEL_MAP_I_TO_R	color index	R	1	0.0
GL_PIXEL_MAP_I_TO_G	color index	G	1	0.0

MAP	LOOKUP INDEX	LOOKUP VALUE	INITIAL SIZE	INITIAL VALUE
GL_PIXEL_MAP_I_TO_B	color index	B	1	0.0
GL_PIXEL_MAP_I_TO_A	color index	A	1	0.0
GL_PIXEL_MAP_R_TO_R	R	R	1	0.0
GL_PIXEL_MAP_G_TO_G	G	G	1	0.0
GL_PIXEL_MAP_B_TO_B	B	B	1	0.0
GL_PIXEL_MAP_A_TO_A	A	A	1	0.0

The following functions retrieve information related to **glPixelMap**:

glGet with argument GL_PIXEL_MAP_I_TO_I_SIZE

glGet with argument GL_PIXEL_MAP_S_TO_S_SIZE

glGet with argument GL_PIXEL_MAP_I_TO_R_SIZE

glGet with argument GL_PIXEL_MAP_I_TO_G_SIZE

glGet with argument GL_PIXEL_MAP_I_TO_B_SIZE

glGet with argument GL_PIXEL_MAP_I_TO_A_SIZE

glGet with argument GL_PIXEL_MAP_R_TO_R_SIZE

glGet with argument GL_PIXEL_MAP_G_TO_G_SIZE

glGet with argument GL_PIXEL_MAP_B_TO_B_SIZE

glGet with argument GL_PIXEL_MAP_A_TO_A_SIZE

glGet with argument GL_MAX_PIXEL_MAP_TABLE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glCopyPixels

glDrawPixels

glEnd

glPixelStore

glPixelTransfer

glReadPixels

glTexImage1

D

g
l
T
e
x
I
m
a
g
e
2
D

glPixelMapuiv function

2/22/2020 • 3 minutes to read • Edit Online

The **glPixelMapuiv** function sets up pixel transfer maps.

Syntax

```
void WINAPI glPixelMapuiv(
         GLenum  map,
         GLsizei mapsize,
   const GLuint  *values
);
```

Parameters

m
a
p

A symbolic map name. The ten maps are as follows.

VALUE	MEANING
GL_PIXEL_MAP_I_TO_I	Maps color indexes to color indexes.
GL_PIXEL_MAP_S_TO_S	Maps stencil indexes to stencil indexes.
GL_PIXEL_MAP_I_TO_R	Maps color indexes to red components.

VALUE	MEANING
GL_PIXEL_MAP_I_TO_G	Maps color indexes to green components.
GL_PIXEL_MAP_I_TO_B	Maps color indexes to blue components.
GL_PIXEL_MAP_I_TO_A	Maps color indexes to alpha components.
GL_PIXEL_MAP_R_TO_R	Maps red components to red components.
GL_PIXEL_MAP_G_TO_G	Maps green components to green components.
GL_PIXEL_MAP_B_TO_B	Maps blue components to blue components.

VALUE	MEANING
GL_PIXEL_MAP_A_TO_A	Maps alpha components to alpha components.

mapsize The size of the map being defined.

values An array of *mapsize* values.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*map* was not an accepted value.
GL_INVALID_VALUE	*mapsize* was negative or larger than GL_PIXEL_MAP_TABLE.

NAME	MEANING
GL_INVALID_VALUE	*map* was GL_PIXEL_MAP_I_TO_I, GL_PIXEL_MAP_S_TO_S, GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, or GL_PIXEL_MAP_I_TO_A, and *mapsize* was not a power of two.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*map* was not an accepted value.
GL_INVALID_VALUE	*mapsize* was negative or larger than GL_PIXEL_MAP_TABLE.
GL_INVALID_VALUE	*map* was GL_PIXEL_MAP_I_TO_I, GL_PIXEL_MAP_S_TO_S, GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, or GL_PIXEL_MAP_I_TO_A, and *mapsize* was not a power of two.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glPixelMap** function sets up translation tables, or *maps*, used by **glCopyPixels**, **glCopyTexImage1D**, **glCopyTexImage2D**, **glCopyTexSubImage1D**, **glCopyTexSubImage2D**, **glDrawPixels**, **glReadPixels**, **glTexImage1D**, **glTexImage2D**, **glTexSubImage1D**, and **glTexSubImage2D**. Use of these maps is described completely in the **glPixelTransfer** topic, and partly in the topics for the pixel and texture image commands. Only the specification of the maps is described in this topic.

The *map* parameter is a symbolic map name, indicating one of ten maps to set. The *mapsize* parameter specifies the number of entries in the map, and *values* is a pointer to an array of *mapsize* map values.

The entries in a map can be specified as single-precision floating-point numbers, unsigned short integers, or unsigned long integers. Maps that store color component values (all but GL_PIXEL_MAP_I_TO_I and GL_PIXEL_MAP_S_TO_S) retain their values in floating-point format, with unspecified mantissa and exponent sizes. Floating-point values specified by **glPixelMapfv** are converted directly to the internal floating-point format of these maps, and then clamped to the range [0,1]. Unsigned integer values specified by **glPixelMapusv** and **glPixelMapuiv** are converted linearly such that the largest representable integer maps to 1.0, and zero maps to 0.0.

Maps that store indexes, GL_PIXEL_MAP_I_TO_I and GL_PIXEL_MAP_S_TO_S, retain their values in fixed-point format, with an unspecified number of bits to the right of the binary point. Floating-point values specified by **glPixelMapfv** are converted directly to the internal fixed-point format of these maps. Unsigned integer values specified by **glPixelMapusv** and **glPixelMapuiv** specify integer values, with all zeros to the right of the binary point.

The following table shows the initial sizes and values for each of the maps. Maps that are indexed by either color or stencil indexes must have *mapsize* = 2 ^ *n* for some *n* or results are undefined. The maximum allowable size for each map depends on the implementation and can be determined by calling **glGet** with argument GL_MAX_PIXEL_MAP_TABLE. The single maximum applies to all maps, and it is at least 32.

MAP	LOOKUP INDEX	LOOKUP VALUE	INITIAL SIZE	INITIAL VALUE
GL_PIXEL_MAP_I_TO_I	color index	color index	1	0.0
GL_PIXEL_MAP_S_TO_S	stencil index	stencil index	1	0.0
GL_PIXEL_MAP_I_TO_R	color index	R	1	0.0
GL_PIXEL_MAP_I_TO_G	color index	G	1	0.0

MAP	LOOKUP INDEX	LOOKUP VALUE	INITIAL SIZE	INITIAL VALUE
GL_PIXEL_MAP_I_TO_B	color index	B	1	0.0
GL_PIXEL_MAP_I_TO_A	color index	A	1	0.0
GL_PIXEL_MAP_R_TO_R	R	R	1	0.0
GL_PIXEL_MAP_G_TO_G	G	G	1	0.0
GL_PIXEL_MAP_B_TO_B	B	B	1	0.0
GL_PIXEL_MAP_A_TO_A	A	A	1	0.0

The following functions retrieve information related to **glPixelMap**:

glGet with argument GL_PIXEL_MAP_I_TO_I_SIZE

glGet with argument GL_PIXEL_MAP_S_TO_S_SIZE

glGet with argument GL_PIXEL_MAP_I_TO_R_SIZE

glGet with argument GL_PIXEL_MAP_I_TO_G_SIZE

glGet with argument GL_PIXEL_MAP_I_TO_B_SIZE

glGet with argument GL_PIXEL_MAP_I_TO_A_SIZE

glGet with argument GL_PIXEL_MAP_R_TO_R_SIZE

glGet with argument GL_PIXEL_MAP_G_TO_G_SIZE

glGet with argument GL_PIXEL_MAP_B_TO_B_SIZE

glGet with argument GL_PIXEL_MAP_A_TO_A_SIZE

glGet with argument GL_MAX_PIXEL_MAP_TABLE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glCopyPixels

glDrawPixels

glEnd

glPixelStore

glPixelTransfer

glReadPixels

glTexImage1

D

g
l
T
e
x
I
m
a
g
e
2
D

glPixelMapusv function

2/22/2020 • 3 minutes to read • Edit Online

The **glPixelMapusv** function sets up pixel transfer maps.

Syntax

```
void WINAPI glPixelMapusv(
         GLenum   map,
         GLsizei  mapsize,
   const GLushort *values
);
```

Parameters

m
a
p

A symbolic map name. The ten maps are as follows.

VALUE	MEANING
GL_PIXEL_MAP_I_TO_I	Maps color indexes to color indexes.
GL_PIXEL_MAP_S_TO_S	Maps stencil indexes to stencil indexes.
GL_PIXEL_MAP_I_TO_R	Maps color indexes to red components.

VALUE	MEANING
GL_PIXEL_MAP_I_TO_G	Maps color indexes to green components.
GL_PIXEL_MAP_I_TO_B	Maps color indexes to blue components.
GL_PIXEL_MAP_I_TO_A	Maps color indexes to alpha components.
GL_PIXEL_MAP_R_TO_R	Maps red components to red components.
GL_PIXEL_MAP_G_TO_G	Maps green components to green components.
GL_PIXEL_MAP_B_TO_B	Maps blue components to blue components.

VALUE	MEANING
GL_PIXEL_MAP_A_TO_A	Maps alpha components to alpha components.

mapsize The size of the map being defined.

values An array of *mapsize* values.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*map* was not an accepted value.
GL_INVALID_VALUE	*mapsize* was negative or larger than GL_PIXEL_MAP_TABLE.

NAME	MEANING
GL_INVALID_VALUE	*map* was GL_PIXEL_MAP_I_TO_I, GL_PIXEL_MAP_S_TO_S, GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, or GL_PIXEL_MAP_I_TO_A, and *mapsize* was not a power of two.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*map* was not an accepted value.
GL_INVALID_VALUE	*mapsize* was negative or larger than GL_PIXEL_MAP_TABLE.
GL_INVALID_VALUE	*map* was GL_PIXEL_MAP_I_TO_I, GL_PIXEL_MAP_S_TO_S, GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, or GL_PIXEL_MAP_I_TO_A, and *mapsize* was not a power of two.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glPixelMap** function sets up translation tables, or *maps*, used by **glCopyPixels**, **glCopyTexImage1D**, **glCopyTexImage2D**, **glCopyTexSubImage1D**, **glCopyTexSubImage2D**, **glDrawPixels**, **glReadPixels**, **glTexImage1D**, **glTexImage2D**, **glTexSubImage1D**, and **glTexSubImage2D**. Use of these maps is described completely in the **glPixelTransfer** topic, and partly in the topics for the pixel and texture image commands. Only the specification of the maps is described in this topic.

The *map* parameter is a symbolic map name, indicating one of ten maps to set. The *mapsize* parameter specifies the number of entries in the map, and *values* is a pointer to an array of *mapsize* map values.

The entries in a map can be specified as single-precision floating-point numbers, unsigned short integers, or unsigned long integers. Maps that store color component values (all but GL_PIXEL_MAP_I_TO_I and GL_PIXEL_MAP_S_TO_S) retain their values in floating-point format, with unspecified mantissa and exponent sizes. Floating-point values specified by **glPixelMapfv** are converted directly to the internal floating-point format of these maps, and then clamped to the range [0,1]. Unsigned integer values specified by **glPixelMapusv** and **glPixelMapuiv** are converted linearly such that the largest representable integer maps to 1.0, and zero maps to 0.0.

Maps that store indexes, GL_PIXEL_MAP_I_TO_I and GL_PIXEL_MAP_S_TO_S, retain their values in fixed-point format, with an unspecified number of bits to the right of the binary point. Floating-point values specified by **glPixelMapfv** are converted directly to the internal fixed-point format of these maps. Unsigned integer values specified by **glPixelMapusv** and **glPixelMapuiv** specify integer values, with all zeros to the right of the binary point.

The following table shows the initial sizes and values for each of the maps. Maps that are indexed by either color or stencil indexes must have *mapsize* = 2 ^ *n* for some *n* or results are undefined. The maximum allowable size for each map depends on the implementation and can be determined by calling **glGet** with argument GL_MAX_PIXEL_MAP_TABLE. The single maximum applies to all maps, and it is at least 32.

MAP	LOOKUP INDEX	LOOKUP VALUE	INITIAL SIZE	INITIAL VALUE
GL_PIXEL_MAP_I_TO_I	color index	color index	1	0.0
GL_PIXEL_MAP_S_TO_S	stencil index	stencil index	1	0.0
GL_PIXEL_MAP_I_TO_R	color index	R	1	0.0
GL_PIXEL_MAP_I_TO_G	color index	G	1	0.0

MAP	LOOKUP INDEX	LOOKUP VALUE	INITIAL SIZE	INITIAL VALUE
GL_PIXEL_MAP_I_TO_B	color index	B	1	0.0
GL_PIXEL_MAP_I_TO_A	color index	A	1	0.0
GL_PIXEL_MAP_R_TO_R	R	R	1	0.0
GL_PIXEL_MAP_G_TO_G	G	G	1	0.0
GL_PIXEL_MAP_B_TO_B	B	B	1	0.0
GL_PIXEL_MAP_A_TO_A	A	A	1	0.0

The following functions retrieve information related to **glPixelMap**:

glGet with argument GL_PIXEL_MAP_I_TO_I_SIZE

glGet with argument GL_PIXEL_MAP_S_TO_S_SIZE

glGet with argument GL_PIXEL_MAP_I_TO_R_SIZE

glGet with argument GL_PIXEL_MAP_I_TO_G_SIZE

glGet with argument GL_PIXEL_MAP_I_TO_B_SIZE

glGet with argument GL_PIXEL_MAP_I_TO_A_SIZE

glGet with argument GL_PIXEL_MAP_R_TO_R_SIZE

glGet with argument GL_PIXEL_MAP_G_TO_G_SIZE

glGet with argument GL_PIXEL_MAP_B_TO_B_SIZE

glGet with argument GL_PIXEL_MAP_A_TO_A_SIZE

glGet with argument GL_MAX_PIXEL_MAP_TABLE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glCopyPixels

glDrawPixels

glEnd

glPixelStore

glPixelTransfer

glReadPixels

glTexImage1

D

g
l
T
e
x
I
m
a
g
e
2
D

glPixelStore Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions set pixel storage modes:

- glPixelStoref
- glPixelStorei

glPixelStoref function

2/22/2020 • 6 minutes to read • Edit Online

Sets pixel storage modes.

Syntax

```
void WINAPI glPixelStoref(
   GLenum  pname,
   GLfloat param
);
```

Parameters

p
n
a
m
e

The symbolic name of the parameter to be set. Six of the storage parameters affect how pixel data is returned to client memory, and are therefore significant only for **glReadPixels** commands. They are as follows:

STORAGE PARAMETER	DESCRIPTION
GL_PACK_SWAP_BYTES	If true, byte ordering for multibyte color components, depth components, color indexes, or stencil indexes is reversed. That is, if a four-byte component is made up of bytes *b*0 , *b*1 , *b*2 , *b*3 , it is stored in memory as *b*3 , *b*2 , *b*1 , *b*0 if GL_PACK_SWAP_BYTES is true. GL_PACK_SWAP_BYTES has no effect on the memory order of components within a pixel, only on the order of bytes within components or indexes. For example, the three components of a GL_RGB format pixel are always stored with red first, green second, and blue third, regardless of the value of GL_PACK_SWAP_BYTES.
GL_PACK_LSB_FIRST	If true, bits are ordered within a byte from least significant to most significant; otherwise, the first bit in each byte is the most significant one. This parameter is significant for bitmap data only.

STORAGE PARAMETER	DESCRIPTION
GL_PACK_ROW_LENGTH	If greater than zero, GL_PACK_ROW_LENGTH defines the number of pixels in a row. If the first pixel of a row is placed at location p in memory, then the location of the first pixel of the next row is obtained by skipping $k = \begin{cases} nl & s \geq a \\ \frac{a}{s}\left\lceil \frac{snl}{a} \right\rceil & s < a \end{cases}$ [newline] components or indexes, where *n* is the number of components or indexes in a pixel, *l* is the number of pixels in a row (gl-pack-row-length if it is greater than zero, the width argument to the pixel routine otherwise), *a* is the value of gl-pack-alignment, and *s* is the size, in bytes, of a single component (if *a* < *s*, then it is as if *a* = *s*). in the case of 1-bit values, the location of the next row is obtained by skipping $k = 8a\left\lceil \frac{nl}{8a} \right\rceil$ components or indexes. The word *component* in this description refers to the nonindex values red, green, blue, alpha, and depth. Storage format GL_RGB, for example, has three components per pixel: first red, then green, and finally blue.
GL_PACK_SKIP_PIXELS and GL_PACK_SKIP_ROWS	These values are provided as a convenience to the programmer; they provide no functionality that cannot be duplicated simply by incrementing the pointer passed to glReadPixels. Setting GL_PACK_SKIP_PIXELS to *i* is equivalent to incrementing the pointer by *i n* components or indexes, where *n* is the number of components or indexes in each pixel. Setting GL_PACK_SKIP_ROWS to *j* is equivalent to incrementing the pointer by *j k* components or indexes, where *k* is the number of components or indexes per row, as computed above in the GL_PACK_ROW_LENGTH section.
GL_PACK_ALIGNMENT	Specifies the alignment requirements for the start of each pixel row in memory. The allowable values are 1 (byte-alignment), 2 (rows aligned to even-numbered bytes), 4 (word alignment), and 8 (rows start on double-word boundaries).

The other six storage parameters affect how pixel data is read from client memory. These values are significant for glDrawPixels, glTexImage1D, glTexImage2D, glBitmap, and glPolygonStipple. They are as follows:

STORAGE PARAMETER	DESCRIPTION
GL_UNPACK_SWAP_BYTES	If true, byte ordering for multibyte color components, depth components, color indexes, or stencil indexes is reversed. That is, if a four-byte component is made up of bytes *b*0 , *b*1 , *b*2 , *b*3 , it is stored in memory as *b*3 , *b*2 , *b*1 , *b*0 if GL_UNPACK_SWAP_BYTES is true. GL_UNPACK_SWAP_BYTES has no effect on the memory order of components within a pixel, only on the order of bytes within components or indexes. For example, the three components of a GL_RGB format pixel are always stored with red first, green second, and blue third, regardless of the value of GL_UNPACK_SWAP_BYTES.

STORAGE PARAMETER	DESCRIPTION
GL_UNPACK_LSB_FIRST	If true, bits are ordered within a byte from least significant to most significant; otherwise, the first bit in each byte is the most significant one. This is significant for bitmap data only.
GL_UNPACK_ROW_LENGTH	If greater than zero, GL_UNPACK_ROW_LENGTH defines the number of pixels in a row. If the first pixel of a row is placed at location p in memory, then the location of the first pixel of the next row is obtained by skipping $k=\begin{cases} nl & s \geq a \\ \frac{a}{s}\left\lceil \frac{snl}{a} \right\rceil & s < a \end{cases}$ [newline] components or indexes, where *n* is the number of components or indexes in a pixel, *l* is the number of pixels in a row (gl-pack-row-length if it is greater than zero, the width argument to the pixel routine otherwise), *a* is the value of gl-pack-alignment, and *s* is the size, in bytes, of a single component (if *a* < *s*, then it is as if *a* = *s*). in the case of 1-bit values, the location of the next row is obtained by skipping $k = 8a\left\lceil \frac{nl}{8a} \right\rceil$ components or indexes. The word *component* in this description refers to the nonindex values red, green, blue, alpha, and depth. Storage format GL_RGB, for example, has three components per pixel: first red, then green, and finally blue.
GL_UNPACK_SKIP_PIXELS and GL_UNPACK_SKIP_ROWS	These values are provided as a convenience to the programmer; they provide no functionality that cannot be duplicated simply by incrementing the pointer passed to **glDrawPixels**, **glTexImage1D**, **glTexImage2D**, **glBitmap**, or **glPolygonStipple**. Setting GL_UNPACK_SKIP_PIXELS to *i* is equivalent to incrementing the pointer by *i n* components or indexes, where *n* is the number of components or indexes in each pixel. Setting GL_UNPACK_SKIP_ROWS to *j* is equivalent to incrementing the pointer by *j k* components or indexes, where *k* is the number of components or indexes per row, as computed above in the GL_UNPACK_ROW_LENGTH section.
GL_UNPACK_ALIGNMENT	Specifies the alignment requirements for the start of each pixel row in memory. The allowable values are 1 (byte-alignment), 2 (rows aligned to even-numbered bytes), 4 (word alignment), and 8 (rows start on double-word boundaries).

p
a
r
a
m

The value that *pname* is set to.

Return value

This function does not return a value.

Remarks

The **glPixelStore** function sets pixel storage modes that affect the operation of subsequent glDrawPixels and glReadPixels as well as the unpacking of polygon stipple patterns (see glPolygonStipple), bitmaps (see glBitmap), and texture patterns (see glTexImage1D, glTexImage2D, glTexSubImage1D, and glTexSubImage2D).

The following table gives the type, initial value, and range of valid values for each of the storage parameters that can be set with **glPixelStore**.

PNAME	TYPE	INITIAL VALUE	VALID RANGE
GL_PACK_SWAP_BYTES	Boolean	false	true or false
GL_PACK_SWAP_BYTES	Boolean	false	true or false
GL_PACK_ROW_LENGTH	integer	0	[0,?)
GL_PACK_SKIP_ROWS	integer	0	[0,?)
GL_PACK_SKIP_PIXELS	integer	0	[0,?)
GL_PACK_ALIGNMENT	integer	4	1, 2, 4, or 8
GL_UNPACK_SWAP_BYTES	Boolean	false	true or false
GL_UNPACK_LSB_FIRST	Boolean	false	true or false
GL_UNPACK_ROW_LENGTH	integer	0	[0,?)
GL_UNPACK_SKIP_ROWS	integer	0	[0,?)
GL_UNPACK_SKIP_PIXELS	integer	0	[0,?)
GL_UNPACK_ALIGNMENT	integer	4	1, 2, 4, or 8

The **glPixelStoref** function can be used to set any pixel store parameter. If the parameter type is Boolean, and if *param* is 0.0, then the parameter is false; otherwise it is set to true. If *pname* is an integer type parameter, then *param* is rounded to the nearest integer.

Likewise, the **glPixelStorei** function can also be used to set any of the pixel store parameters. Boolean parameters are set to false if *param* is 0 and true otherwise. The *param* parameter is converted to floating point before being assigned to real-valued parameters.

The pixel storage modes in effect when glDrawPixels, glReadPixels, glTexImage1D, glTexImage2D, glBitmap, or glPolygonStipple is placed in a display list control the interpretation of memory data. The pixel storage modes in effect when a display list is executed are not significant.

The following functions retrieve information related to **glPixelStore**:

glGet with argument GL_PACK_SWAP_BYTES

glGet with argument GL_PACK_LSB_FIRST

glGet with argument GL_PACK_ROW_LENGTH

glGet with argument GL_PACK_SKIP_ROWS

glGet with argument GL_PACK_SKIP_PIXELS

glGet with argument GL_PACK_ALIGNMENT

glGet with argument GL_UNPACK_SWAP_BYTES

glGet with argument GL_UNPACK_LSB_FIRST

glGet with argument GL_UNPACK_ROW_LENGTH

glGet with argument GL_UNPACK_SKIP_ROWS

glGet with argument GL_UNPACK_SKIP_PIXELS

glGet with argument GL_UNPACK_ALIGNMENT

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glBitmap

p

glDrawPixels

glEnd

glPixelMap

glPixelTransfer

glPix

e
l
Z
o
o
m

g
l
P
o
l
y
g
o
n
S
t
i
p
p
l
e

g
l
R
e
a
d
P
i
x
e
l
s

g
l
T
e
x
l
m
a
g
e
1
D

g
l
T

e
x
I
m
a
g
e
2
D

glPixelStorei function

2/22/2020 • 6 minutes to read • Edit Online

Sets pixel storage modes.

Syntax

```
void WINAPI glPixelStorei(
   GLenum pname,
   GLint  param
);
```

Parameters

p
n
a
m
e

The symbolic name of the parameter to be set. Six of the storage parameters affect how pixel data is returned to client memory, and are therefore significant only for glReadPixels commands. They are as follows.

STORAGE PARAMETER	DESCRIPTION
GL_PACK_SWAP_BYTES	If true, byte ordering for multibyte color components, depth components, color indexes, or stencil indexes is reversed. That is, if a four-byte component is made up of bytes *b*0 , *b*1 , *b*2 , *b*3 , it is stored in memory as *b*3 , *b*2 , *b*1 , *b*0 if GL_PACK_SWAP_BYTES is true. GL_PACK_SWAP_BYTES has no effect on the memory order of components within a pixel, only on the order of bytes within components or indexes. For example, the three components of a GL_RGB format pixel are always stored with red first, green second, and blue third, regardless of the value of GL_PACK_SWAP_BYTES.
GL_PACK_LSB_FIRST	If true, bits are ordered within a byte from least significant to most significant; otherwise, the first bit in each byte is the most significant one. This parameter is significant for bitmap data only.

STORAGE PARAMETER	DESCRIPTION
GL_PACK_ROW_LENGTH	If greater than zero, GL_PACK_ROW_LENGTH defines the number of pixels in a row. If the first pixel of a row is placed at location p in memory, then the location of the first pixel of the next row is obtained by skipping $k = \begin{cases} nl & s \geq a \\ \frac{a}{s}\left\lceil \frac{snl}{a} \right\rceil & s < a \end{cases}$ [newline] components or indexes, where *n* is the number of components or indexes in a pixel, *l* is the number of pixels in a row (gl-pack-row-length if it is greater than zero, the width argument to the pixel routine otherwise), *a* is the value of gl-pack-alignment, and *s* is the size, in bytes, of a single component (if *a* < *s*, then it is as if *a* = *s*). in the case of 1-bit values, the location of the next row is obtained by skipping $k = 8a\left\lceil \frac{nl}{8a} \right\rceil$ components or indexes. The word *component* in this description refers to the nonindex values red, green, blue, alpha, and depth. Storage format GL_RGB, for example, has three components per pixel: first red, then green, and finally blue.
GL_PACK_SKIP_PIXELS and GL_PACK_SKIP_ROWS	These values are provided as a convenience to the programmer; they provide no functionality that cannot be duplicated simply by incrementing the pointer passed to glReadPixels. Setting GL_PACK_SKIP_PIXELS to *i* is equivalent to incrementing the pointer by *i n* components or indexes, where *n* is the number of components or indexes in each pixel. Setting GL_PACK_SKIP_ROWS to *j* is equivalent to incrementing the pointer by *j k* components or indexes, where *k* is the number of components or indexes per row, as computed above in the GL_PACK_ROW_LENGTH section.
GL_PACK_ALIGNMENT	Specifies the alignment requirements for the start of each pixel row in memory. The allowable values are 1 (byte-alignment), 2 (rows aligned to even-numbered bytes), 4 (word alignment), and 8 (rows start on double-word boundaries).

The other six storage parameters affect how pixel data is read from client memory. These values are significant for glDrawPixels, glTexImage1D, glTexImage2D, glBitmap, and glPolygonStipple. They are as follows:

STORAGE PARAMETER	DESCRIPTION
GL_UNPACK_SWAP_BYTES	If true, byte ordering for multibyte color components, depth components, color indexes, or stencil indexes is reversed. That is, if a four-byte component is made up of bytes *b*0 , *b*1 , *b*2 , *b*3 , it is stored in memory as *b*3 , *b*2 , *b*1 , *b*0 if GL_UNPACK_SWAP_BYTES is true. GL_UNPACK_SWAP_BYTES has no effect on the memory order of components within a pixel, only on the order of bytes within components or indexes. For example, the three components of a GL_RGB format pixel are always stored with red first, green second, and blue third, regardless of the value of GL_UNPACK_SWAP_BYTES.

STORAGE PARAMETER	DESCRIPTION
GL_UNPACK_LSB_FIRST	If true, bits are ordered within a byte from least significant to most significant; otherwise, the first bit in each byte is the most significant one. This is significant for bitmap data only.
GL_UNPACK_ROW_LENGTH	If greater than zero, GL_UNPACK_ROW_LENGTH defines the number of pixels in a row. If the first pixel of a row is placed at location p in memory, then the location of the first pixel of the next row is obtained by skipping $k=\begin{cases} nl & s \geq a \\ \frac{a}{s}\left\lceil \frac{snl}{a} \right\rceil & s < a \end{cases}$ [newline] components or indexes, where *n* is the number of components or indexes in a pixel, *l* is the number of pixels in a row (gl-pack-row-length if it is greater than zero, the width argument to the pixel routine otherwise), *a* is the value of gl-pack-alignment, and *s* is the size, in bytes, of a single component (if *a* < *s*, then it is as if *a* = *s*). in the case of 1-bit values, the location of the next row is obtained by skipping $k = 8a\left\lceil \frac{nl}{8a} \right\rceil$ components or indexes. The word *component* in this description refers to the nonindex values red, green, blue, alpha, and depth. Storage format GL_RGB, for example, has three components per pixel: first red, then green, and finally blue.
GL_UNPACK_SKIP_PIXELS and GL_UNPACK_SKIP_ROWS	These values are provided as a convenience to the programmer; they provide no functionality that cannot be duplicated simply by incrementing the pointer passed to **glDrawPixels**, **glTexImage1D**, **glTexImage2D**, **glBitmap**, or **glPolygonStipple**. Setting GL_UNPACK_SKIP_PIXELS to *i* is equivalent to incrementing the pointer by *i n* components or indexes, where *n* is the number of components or indexes in each pixel. Setting GL_UNPACK_SKIP_ROWS to *j* is equivalent to incrementing the pointer by *j k* components or indexes, where *k* is the number of components or indexes per row, as computed above in the GL_UNPACK_ROW_LENGTH section.
GL_UNPACK_ALIGNMENT	Specifies the alignment requirements for the start of each pixel row in memory. The allowable values are 1 (byte-alignment), 2 (rows aligned to even-numbered bytes), 4 (word alignment), and 8 (rows start on double-word boundaries).

p
a
r
a
m

The value that *pname* is set to.

Return value

This function does not return a value.

Remarks

The **glPixelStore** function sets pixel storage modes that affect the operation of subsequent glDrawPixels and glReadPixels as well as the unpacking of polygon stipple patterns (see glPolygonStipple), bitmaps (see glBitmap), and texture patterns (see glTexImage1D, glTexImage2D, glTexSubImage1D, and glTexSubImage2D).

The following table gives the type, initial value, and range of valid values for each of the storage parameters that can be set with **glPixelStore**.

PNAME	TYPE	INITIAL VALUE	VALID RANGE
GL_PACK_SWAP_BYTES	Boolean	false	true or false
GL_PACK_SWAP_BYTES	Boolean	false	true or false
GL_PACK_ROW_LENGTH	integer	0	[0,?)
GL_PACK_SKIP_ROWS	integer	0	[0,?)
GL_PACK_SKIP_PIXELS	integer	0	[0,?)
GL_PACK_ALIGNMENT	integer	4	1, 2, 4, or 8
GL_UNPACK_SWAP_BYTES	Boolean	false	true or false
GL_UNPACK_LSB_FIRST	Boolean	false	true or false
GL_UNPACK_ROW_LENGTH	integer	0	[0,?)
GL_UNPACK_SKIP_ROWS	integer	0	[0,?)
GL_UNPACK_SKIP_PIXELS	integer	0	[0,?)
GL_UNPACK_ALIGNMENT	integer	4	1, 2, 4, or 8

The glPixelStoref function can be used to set any pixel store parameter. If the parameter type is Boolean, and if *param* is 0.0, then the parameter is false; otherwise it is set to true. If *pname* is an integer type parameter, then *param* is rounded to the nearest integer.

Likewise, the **glPixelStorei** function can also be used to set any of the pixel store parameters. Boolean parameters are set to false if *param* is 0 and true otherwise. The *param* parameter is converted to floating point before being assigned to real-valued parameters.

The pixel storage modes in effect when glDrawPixels, glReadPixels, glTexImage1D, glTexImage2D, glBitmap, or glPolygonStipple is placed in a display list control the interpretation of memory data. The pixel storage modes in effect when a display list is executed are not significant.

The following functions retrieve information related to **glPixelStore**:

glGet with argument GL_PACK_SWAP_BYTES

glGet with argument GL_PACK_LSB_FIRST

glGet with argument GL_PACK_ROW_LENGTH

glGet with argument GL_PACK_SKIP_ROWS

glGet with argument GL_PACK_SKIP_PIXELS

glGet with argument GL_PACK_ALIGNMENT

glGet with argument GL_UNPACK_SWAP_BYTES

glGet with argument GL_UNPACK_LSB_FIRST

glGet with argument GL_UNPACK_ROW_LENGTH

glGet with argument GL_UNPACK_SKIP_ROWS

glGet with argument GL_UNPACK_SKIP_PIXELS

glGet with argument GL_UNPACK_ALIGNMENT

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glBitmap

p

glDrawPixels

glEnd

glPixelMap

glPixelTransfer

glPix

e
l
Z
o
o
m

g
l
P
o
l
y
g
o
n
S
t
i
p
p
l
e

g
l
R
e
a
d
P
i
x
e
l
s

g
l
T
e
x
I
m
a
g
e
1
D

g
l
T

e
x
I
m
a
g
e
2
D

glPixelTransfer Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions set pixel transfer modes:

- glPixelTransferf
- glPixelTransferi

glPixelTransferf function

2/22/2020 • 5 minutes to read • Edit Online

The **glPixelTransferf** and **glPixelTransferi** functions set pixel transfer modes.

Syntax

```
void WINAPI glPixelTransferf(
   GLenum  pname,
   GLfloat param
);
```

Parameters

p
n
a
m
e

The symbolic name of the pixel transfer parameter to be set. The following table gives the type, initial value, and range of valid values for each of the pixel transfer parameters that are set with **glPixelTransfer**.

PNAME	TYPE	INITIAL VALUE	VALID RANGE
GL_MAP_COLOR	Boolean	false	true/false
GL_MAP_STENCIL	Boolean	false	true/false
GL_INDEX_SHIFT	integer	0	(8,8)
GL_INDEX_OFFSET	integer	0	(8,8)
GL_RED_SCALE	integer	1.0	(8,8)
GL_GREEN_SCALE	float	1.0	(8,8)
GL_BLUE_SCALE	float	1.0	(8,8)
GL_ALPHA_SCALE	float	1.0	(8,8)
GL_DEPTH_SCALE	float	1.0	(8,8)
GL_RED_BIAS	float	0.0	(8,8)
GL_GREEN_BIAS	float	0.0	(8,8)
GL_BLUE_BIAS	float	0.0	(8,8)

PNAME	TYPE	INITIAL VALUE	VALID RANGE
GL_ALPHA_BIAS	float	0.0	(8,8)
GL_DEPTH_BIAS	float	0.0	(8,8)

p a r a m The value that *pname* is set to.

Return value

This function does not return a value.

Remarks

The **glPixelTransfer** function sets pixel transfer modes that affect the operation of subsequent **glCopyPixels**, **glCopyTexImage1D**, **glCopyTexImage2D**, **glCopyTexSubImage1D**, **glCopyTexSubImage2D**, **glDrawPixels**, **glReadPixels**, **glTexImage1D**, **glTexImage2D**, **glTexSubImage1D**, and **glTexSubImage2D** commands. The algorithms that are specified by pixel transfer modes operate on pixels after they are read from the framebuffer (**glReadPixels** and **glCopyPixels**) or unpacked from client memory (**glDrawPixels**, **glTexImage1D**, and **glTexImage2D**). Pixel transfer operations happen in the same order, and in the same manner, regardless of the command that resulted in the pixel operation. Pixel storage modes (**glPixelStore**) control the unpacking of pixels being read from client memory, and the packing of pixels being written back into client memory.

Pixel transfer operations handle four fundamental pixel types: *color*, *color index*, *depth*, and *stencil*.Color pixels are made up of four floating-point values with unspecified mantissa and exponent sizes, scaled such that 0.0 represents zero intensity and 1.0 represents full intensity. Color indexes comprise a single fixed-point value, with unspecified precision to the right of the binary point. Depth pixels comprise a single floating-point value, with unspecified mantissa and exponent sizes, scaled such that 0.0 represents the minimum depth buffer value, and 1.0 represents the maximum depth buffer value. Finally, stencil pixels comprise a single fixed-point value, with unspecified precision to the right of the binary point.

The pixel transfer operations performed on the four basic pixel types are as follows:

PIXEL TYPE	PIXEL TRANSFER OPERATION

PIXEL TYPE	PIXEL TRANSFER OPERATION
Color	Each of the four color components is multiplied by a scale factor, and then added to a bias factor. That is, the red component is multiplied by GL_RED_SCALE, and then added to GL_RED_BIAS; the green component is multiplied by GL_GREEN_SCALE, and then added to GL_GREEN_BIAS; the blue component is multiplied by GL_BLUE_SCALE, and then added to GL_BLUE_BIAS; and the alpha component is multiplied by GL_ALPHA_SCALE, and then added to GL_ALPHA_BIAS. After all four color components are scaled and biased, each is clamped to the range [0,1]. All color scale and bias values are specified with **glPixelTransfer**. If GL_MAP_COLOR is true, each color component is scaled by the size of the corresponding color-to-color map, and then replaced by the contents of that map indexed by the scaled component. That is, the red component is scaled by GL_PIXEL_MAP_R_TO_R_SIZE, and then replaced by the contents of GL_PIXEL_MAP_R_TO_R indexed by itself. The green component is scaled by GL_PIXEL_MAP_G_TO_G_SIZE, and then replaced by the contents of GL_PIXEL_MAP_G_TO_G indexed by itself. The blue component is scaled by GL_PIXEL_MAP_B_TO_B_SIZE, and then replaced by the contents of GL_PIXEL_MAP_B_TO_B indexed by itself. The alpha component is scaled by GL_PIXEL_MAP_A_TO_A_SIZE, and then replaced by the contents of GL_PIXEL_MAP_A_TO_A indexed by itself. All components taken from the maps are then clamped to the range [0,1]. GL_MAP_COLOR is specified with **glPixelTransfer**. The contents of the various maps are specified with **glPixelMap**.
Color index	Each color index is shifted left by GL_INDEX_SHIFT bits, filling with zeros any bits beyond the number of fraction bits carried by the fixed-point index. If GL_INDEX_SHIFT is negative, the shift is to the right, again zero filled. GL_INDEX_OFFSET is then added to the index. GL_INDEX_SHIFT and GL_INDEX_OFFSET are specified with **glPixelTransfer**. From this point, operation diverges depending on the required format of the resulting pixels. If the resulting pixels are to be written to a color-index buffer, or if they are being read back to client memory in GL_COLOR_INDEX format, the pixels continue to be treated as indexes. If GL_MAP_COLOR is true, then each index is masked by 2 ^ *n* 1, where *n* is GL_PIXEL_MAP_I_TO_I_SIZE, and then replaced by the contents of GL_PIXEL_MAP_I_TO_I indexed by the masked value. GL_MAP_COLOR is specified with **glPixelTransfer**. The contents of the index map are specified with **glPixelMap**. If the resulting pixels are to be written to an RGBA color buffer, or if they are being read back to client memory in a format other than GL_COLOR_INDEX, the pixels are converted from indexes to colors by referencing the four maps GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, and GL_PIXEL_MAP_I_TO_A. Before being dereferenced, the index is masked by 2 n 1, where n is GL_PIXEL_MAP_I_TO_R_SIZE for the red map, GL_PIXEL_MAP_I_TO_G_SIZE for the green map, GL_PIXEL_MAP_I_TO_B_SIZE for the blue map, and GL_PIXEL_MAP_I_TO_A_SIZE for the alpha map. All components taken from the maps are then clamped to the range [0,1]. The contents of the four maps are specified with **glPixelMap**.

PIXEL TYPE	PIXEL TRANSFER OPERATION
Depth	Each depth value is multiplied by GL_DEPTH_SCALE, added to GL_DEPTH_BIAS, and then clamped to the range [0,1].
Stencil	Each index is shifted GL_INDEX_SHIFT bits just as a color index is, and then added to GL_INDEX_OFFSET. If GL_MAP_STENCIL is true, each index is masked by 2n 1, where *n* is GL_PIXEL_MAP_S_TO_S_SIZE, then replaced by the contents of GL_PIXEL_MAP_S_TO_S indexed by the masked value.

The **glPixelTransferf** function can be used to set any pixel transfer parameter. If the parameter type is Boolean, 0.0 implies false and any other value implies true. If *pname* is an integer parameter, *param* is rounded to the nearest integer.

Likewise, **glPixelTransferi** can also be used to set any of the pixel transfer parameters. Boolean parameters are set to false if *param* is 0 and true otherwise. The *param* parameter is converted to floating point before being assigned to real-valued parameters.

If a **glDrawPixels**, **glReadPixels**, **glCopyPixels**, **glTexImage1D**, or **glTexImage2D** command is placed in a display list (see **glNewList** and **glCallList**), the pixel transfer mode settings in effect when the display list is *executed* are the ones that are used. They may be different from the settings when the command was compiled into the display list.

The following functions retrieve information related to **glPixelTransfer**:

glGet with argument GL_MAP_COLOR

glGet with argument GL_MAP_STENCIL

glGet with argument GL_INDEX_SHIFT

glGet with argument GL_INDEX_OFFSET

glGet with argument GL_RED_SCALE

glGet with argument GL_RED_BIAS

glGet with argument GL_GREEN_SCALE

glGet with argument GL_GREEN_BIAS

glGet with argument GL_BLUE_SCALE

glGet with argument GL_BLUE_BIAS

glGet with argument GL_ALPHA_SCALE

glGet with argument GL_ALPHA_BIAS

glGet with argument GL_DEPTH_SCALE

glGet with argument GL_DEPTH_BIAS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glCallList

glCopyPixels

gl

D
r
a
w
P
i
x
e
l
s

g
l
E
n
d

g
l
N
e
w
L
i
s
t

g
l
P
i
x
e
l
M
a
p

g
l
P
i
x
e
l
S
t
o
r
e

g
l

P
i
x
e
l
Z
o
o
m

g
l
R
e
a
d
P
i
x
e
l
s

g
l
T
e
x
I
m
a
g
e
1
D

g
l
T
e
x
I
m
a
g
e
2
D

glPixelTransferi function

2/22/2020 • 5 minutes to read • Edit Online

The **glPixelTransferf** and **glPixelTransferi** functions set pixel transfer modes.

Syntax

```
void WINAPI glPixelTransferi(
   GLenum pname,
   GLint  param
);
```

Parameters

p
n
a
m
e

The symbolic name of the pixel transfer parameter to be set. The following table gives the type, initial value, and range of valid values for each of the pixel transfer parameters that are set with **glPixelTransfer**.

PNAME	TYPE	INITIAL VALUE	VALID RANGE
GL_MAP_COLOR	Boolean	false	true/false
GL_MAP_STENCIL	Boolean	false	true/false
GL_INDEX_SHIFT	integer	0	(8,8)
GL_INDEX_OFFSET	integer	0	(8,8)
GL_RED_SCALE	integer	1.0	(8,8)
GL_GREEN_SCALE	float	1.0	(8,8)
GL_BLUE_SCALE	float	1.0	(8,8)
GL_ALPHA_SCALE	float	1.0	(8,8)
GL_DEPTH_SCALE	float	1.0	(8,8)
GL_RED_BIAS	float	0.0	(8,8)
GL_GREEN_BIAS	float	0.0	(8,8)
GL_BLUE_BIAS	float	0.0	(8,8)

PNAME	TYPE	INITIAL VALUE	VALID RANGE
GL_ALPHA_BIAS	float	0.0	(8,8)
GL_DEPTH_BIAS	float	0.0	(8,8)

p
a
r
a
m

The value that *pname* is set to.

Return value

This function does not return a value.

Remarks

The **glPixelTransfer** function sets pixel transfer modes that affect the operation of subsequent **glCopyPixels**, **glCopyTexImage1D**, **glCopyTexImage2D**, **glCopyTexSubImage1D**, **glCopyTexSubImage2D**, **glDrawPixels**, **glReadPixels**, **glTexImage1D**, **glTexImage2D**, **glTexSubImage1D**, and **glTexSubImage2D** commands. The algorithms that are specified by pixel transfer modes operate on pixels after they are read from the framebuffer (**glReadPixels** and **glCopyPixels**) or unpacked from client memory (**glDrawPixels**, **glTexImage1D**, and **glTexImage2D**). Pixel transfer operations happen in the same order, and in the same manner, regardless of the command that resulted in the pixel operation. Pixel storage modes (**glPixelStore**) control the unpacking of pixels being read from client memory, and the packing of pixels being written back into client memory.

Pixel transfer operations handle four fundamental pixel types: *color*, *color index*, *depth*, and *stencil*.Color pixels are made up of four floating-point values with unspecified mantissa and exponent sizes, scaled such that 0.0 represents zero intensity and 1.0 represents full intensity. Color indexes comprise a single fixed-point value, with unspecified precision to the right of the binary point. Depth pixels comprise a single floating-point value, with unspecified mantissa and exponent sizes, scaled such that 0.0 represents the minimum depth buffer value, and 1.0 represents the maximum depth buffer value. Finally, stencil pixels comprise a single fixed-point value, with unspecified precision to the right of the binary point.

The pixel transfer operations performed on the four basic pixel types are as follows:

PIXEL TYPE	PIXEL TRANSFER OPERATION

PIXEL TYPE	PIXEL TRANSFER OPERATION
Color	Each of the four color components is multiplied by a scale factor, and then added to a bias factor. That is, the red component is multiplied by GL_RED_SCALE, and then added to GL_RED_BIAS; the green component is multiplied by GL_GREEN_SCALE, and then added to GL_GREEN_BIAS; the blue component is multiplied by GL_BLUE_SCALE, and then added to GL_BLUE_BIAS; and the alpha component is multiplied by GL_ALPHA_SCALE, and then added to GL_ALPHA_BIAS. After all four color components are scaled and biased, each is clamped to the range [0,1]. All color scale and bias values are specified with **glPixelTransfer**. If GL_MAP_COLOR is true, each color component is scaled by the size of the corresponding color-to-color map, and then replaced by the contents of that map indexed by the scaled component. That is, the red component is scaled by GL_PIXEL_MAP_R_TO_R_SIZE, and then replaced by the contents of GL_PIXEL_MAP_R_TO_R indexed by itself. The green component is scaled by GL_PIXEL_MAP_G_TO_G_SIZE, and then replaced by the contents of GL_PIXEL_MAP_G_TO_G indexed by itself. The blue component is scaled by GL_PIXEL_MAP_B_TO_B_SIZE, and then replaced by the contents of GL_PIXEL_MAP_B_TO_B indexed by itself. The alpha component is scaled by GL_PIXEL_MAP_A_TO_A_SIZE, and then replaced by the contents of GL_PIXEL_MAP_A_TO_A indexed by itself. All components taken from the maps are then clamped to the range [0,1]. GL_MAP_COLOR is specified with **glPixelTransfer**. The contents of the various maps are specified with **glPixelMap**.
Color index	Each color index is shifted left by GL_INDEX_SHIFT bits, filling with zeros any bits beyond the number of fraction bits carried by the fixed-point index. If GL_INDEX_SHIFT is negative, the shift is to the right, again zero filled. GL_INDEX_OFFSET is then added to the index. GL_INDEX_SHIFT and GL_INDEX_OFFSET are specified with **glPixelTransfer**. From this point, operation diverges depending on the required format of the resulting pixels. If the resulting pixels are to be written to a color-index buffer, or if they are being read back to client memory in GL_COLOR_INDEX format, the pixels continue to be treated as indexes. If GL_MAP_COLOR is true, then each index is masked by 2 ^ *n* 1, where *n* is GL_PIXEL_MAP_I_TO_I_SIZE, and then replaced by the contents of GL_PIXEL_MAP_I_TO_I indexed by the masked value. GL_MAP_COLOR is specified with **glPixelTransfer**. The contents of the index map are specified with **glPixelMap**. If the resulting pixels are to be written to an RGBA color buffer, or if they are being read back to client memory in a format other than GL_COLOR_INDEX, the pixels are converted from indexes to colors by referencing the four maps GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, and GL_PIXEL_MAP_I_TO_A. Before being dereferenced, the index is masked by 2 n 1, where n is GL_PIXEL_MAP_I_TO_R_SIZE for the red map, GL_PIXEL_MAP_I_TO_G_SIZE for the green map, GL_PIXEL_MAP_I_TO_B_SIZE for the blue map, and GL_PIXEL_MAP_I_TO_A_SIZE for the alpha map. All components taken from the maps are then clamped to the range [0,1]. The contents of the four maps are specified with **glPixelMap**.

PIXEL TYPE	PIXEL TRANSFER OPERATION
Depth	Each depth value is multiplied by GL_DEPTH_SCALE, added to GL_DEPTH_BIAS, and then clamped to the range [0,1].
Stencil	Each index is shifted GL_INDEX_SHIFT bits just as a color index is, and then added to GL_INDEX_OFFSET. If GL_MAP_STENCIL is true, each index is masked by 2n 1, where *n* is GL_PIXEL_MAP_S_TO_S_SIZE, then replaced by the contents of GL_PIXEL_MAP_S_TO_S indexed by the masked value.

The **glPixelTransferf** function can be used to set any pixel transfer parameter. If the parameter type is Boolean, 0.0 implies false and any other value implies true. If *pname* is an integer parameter, *param* is rounded to the nearest integer.

Likewise, **glPixelTransferi** can also be used to set any of the pixel transfer parameters. Boolean parameters are set to false if *param* is 0 and true otherwise. The *param* parameter is converted to floating point before being assigned to real-valued parameters.

If a **glDrawPixels**, **glReadPixels**, **glCopyPixels**, **glTexImage1D**, or **glTexImage2D** command is placed in a display list (see **glNewList** and **glCallList**), the pixel transfer mode settings in effect when the display list is *executed* are the ones that are used. They may be different from the settings when the command was compiled into the display list.

The following functions retrieve information related to **glPixelTransfer**:

glGet with argument GL_MAP_COLOR

glGet with argument GL_MAP_STENCIL

glGet with argument GL_INDEX_SHIFT

glGet with argument GL_INDEX_OFFSET

glGet with argument GL_RED_SCALE

glGet with argument GL_RED_BIAS

glGet with argument GL_GREEN_SCALE

glGet with argument GL_GREEN_BIAS

glGet with argument GL_BLUE_SCALE

glGet with argument GL_BLUE_BIAS

glGet with argument GL_ALPHA_SCALE

glGet with argument GL_ALPHA_BIAS

glGet with argument GL_DEPTH_SCALE

glGet with argument GL_DEPTH_BIAS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glCallList

glCopyPixels

gl

D
r
a
w
P
i
x
e
l
s

g
l
E
n
d

g
l
N
e
w
L
i
s
t

g
l
P
i
x
e
l
M
a
p

g
l
P
i
x
e
l
S
t
o
r
e

g
l

P
i
x
e
l
Z
o
o
m

g
l
R
e
a
d
P
i
x
e
l
s

g
l
T
e
x
I
m
a
g
e
1
D

g
l
T
e
x
I
m
a
g
e
2
D

glPixelZoom function

2/22/2020 • 2 minutes to read • Edit Online

The **glPixelZoom** function specifies the pixel zoom factors.

Syntax

```
void WINAPI glPixelZoom(
   GLfloat xfactor,
   GLfloat yfactor
);
```

Parameters

xfactor	The *x* zoom factor for pixel write operations.
yfactor	The *y* zoom factor for pixel write operations.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glPixelZoom** function specifies values for the *x* and *y* zoom factors. During the execution of **glDrawPixels** or **glCopyPixels**, if (x_r, y_r) is the current raster position, and a given element is in the *n*th row and *m*th column of the pixel rectangle, then pixels whose centers are in the rectangle with corners at

$$(x_r + n \cdot xfactor, y_r + m \cdot yfactor)$$
$$(x_r + (n+1) \cdot xfactor, y_r + (m+1) \cdot yfactor)$$

are candidates for replacement. Any pixel whose center lies on the bottom or left edge of this rectangular region is also modified.

Pixel zoom factors are not limited to positive values. Negative zoom factors reflect the resulting image about the current raster position.

The following functions retrieve information related to **glPixelZoom**:

glGet with argument GL_ZOOM_X

glGet with argument GL_ZOOM_Y

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib

DLL	Opengl32.dll

See also

glBegin

glCopyPixels

glDrawPixels

glEnd

glPointSize function

2/22/2020 • 2 minutes to read • Edit Online

The **glPointSize** function specifies the diameter of rasterized points.

Syntax

```
void WINAPI glPointSize(
   GLfloat size
);
```

Parameters

s *i* *z* *e*	The diameter of rasterized points. The default is 1.0.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	*size* was less than or equal to zero.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	*size* was less than or equal to zero.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glPointSize** function specifies the rasterized diameter of both aliased and antialiased points. Using a point size other than 1.0 has different effects, depending on whether point antialiasing is enabled. Point antialiasing is controlled by calling **glEnable** and **glDisable** with argument GL_POINT_SMOOTH.

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

$(x_w + .5, y_w + .5)$

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 points (x_w, y_w). The coverage value for each fragment is the window coordinate area of the intersection of the circular region with the corresponding pixel square. This value is saved and used in the final rasterization step. The data associated with each fragment is the data associated with the point being rasterized.

Not all sizes are supported when point antialiasing is enabled. If an unsupported size is requested, the nearest supported size is used. Only size 1.0 is guaranteed to be supported; others depend on the implementation. The range of supported sizes and the size difference between supported sizes within the range can be queried by calling **glGet** with arguments GL_POINT_SIZE_RANGE and GL_POINT_SIZE_GRANULARITY.

The point size specified by **glPointSize** is always returned when GL_POINT_SIZE is queried. Clamping and rounding for aliased and antialiased points have no effect on the specified value.

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

the nearest integer value.

The following functions retrieve information related to **glPointSize**:

glGet with argument GL_POINT_SIZE

glGet with argument GL_POINT_SIZE_RANGE

glGet with argument GL_POINT_SIZE_GRANULARITY

glIsEnabled with argument GL_POINT_SMOOTH

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib
DLL	Ope ngl3 2.dll

See also

glBegin

glEnable

glE

n
d

g
l
l
s
E
n
a
b
l
e
d

glPolygonMode function

2/22/2020 • 2 minutes to read • Edit Online

The **glPolygonMode** function selects a polygon rasterization mode.

Syntax

```
void WINAPI glPolygonMode(
   GLenum face,
   GLenum mode
);
```

Parameters

face

The polygons that *mode* applies to. Must be GL_FRONT for front-facing polygons, GL_BACK for back-facing polygons, or GL_FRONT_AND_BACK for front- and back-facing polygons.

mode

The way polygons will be rasterized. The following modes are defined and can be specified in *mode*. The default is GL_FILL for both front- and back-facing polygons.

VALUE	MEANING
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.

VALUE	MEANING
GL_ FIL L	The interior of the polygon is filled. Polygon attributes such as GL_POLYGON_STIPPLE and GL_POLYGON_SMOOTH control the rasterization of the polygon.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_ INV ALI D_E NU M	Either *face* or *mode* was not an accepted value.
GL_ INV ALI D_ OP ERA TIO N	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_ INV ALI D_E NU M	Either *face* or *mode* was not an accepted value.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glPolygonMode** function controls the interpretation of polygons for rasterization. The *face* parameter describes which polygons *mode* applies to: front-facing polygons (GL_FRONT), back-facing polygons (GL_BACK), or both (GL_FRONT_AND_BACK). The polygon mode affects only the final rasterization of polygons. In particular, a polygon's vertices are lit and the polygon is clipped and possibly culled before these modes are applied.

To draw a surface with filled back-facing polygons and outlined front-facing polygons, call

glPolygonMode(GL_FRONT, GL_LINE);

Vertices are marked as boundary or nonboundary with an edge flag. Edge flags are generated internally by OpenGL when it decomposes polygons, and they can be set explicitly using **glEdgeFlag**.

The following function retrieves information related to **glPolygonMode**:

glGet with argument GL_POLYGON_MODE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

g

lBegin

glEdgeFlag

glEnd

glLineStipple

glLineWidth

glP

o
i
n
t
S
i
z
e

g
l
P
o
l
y
g
o
n
S
t
i
p
p
l
e

glPolygonOffset function

2/22/2020 • 2 minutes to read • Edit Online

The **glPolygonOffset** function sets the scale and units OpenGL uses to calculate depth values.

Syntax

```
void WINAPI glPolygonOffset(
   GLfloat factor,
   GLfloat units
);
```

Parameters

factor

Specifies a scale factor that is used to create a variable depth offset for each polygon. The initial value is zero.

units

Specifies a value that is multiplied by an implementation-specific value to create a constant depth offset. The initial value is 0.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Error codes

The following error code can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Remarks

When GL_POLYGON_OFFSET is enabled, each fragment's depth value will be offset after it is interpolated from the depth values of the appropriate vertices. The value of the offset is *factor* * ?z + r **units*, where ?z is a measurement of the change in depth relative to the screen area of the polygon, and r is the smallest value that is guaranteed to produce a resolvable offset for a given implementation. The offset is added before the depth test is performed and before the value is written into the depth buffer.

The **glPolygonOffset** function is useful for rendering hidden-line images, for applying decals to surfaces, and for rendering solids with highlighted edges.

The **glPolygonOffset** function has no effect on depth coordinates placed in the feedback buffer. It also has no effect on selection.

The following functions retrieve information related to **glPolygonOffset**:

- glGet with argument GL_POLYGON_OFFSET_FACTOR
- glGet with argument GL_POLYGON_OFFSET_UNITS
- glIsEnabled with argument GL_POLYGON_OFFSET_FILL
- glIsEnabled with argument GL_POLYGON_OFFSET_LINE
- glIsEnabled with argument GL_POLYGON_OFFSET_POINT

> **NOTE**
>
> The **glPolygonOffset** function is only available in OpenGl version 1.1 or greater.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glDepthFunc

glDisable

glEnable

glGet

glIsEnabled

glLineWidth

glStencilOp

glTexEnv

glPolygonStipple function

2/22/2020 • 2 minutes to read • Edit Online

The **glPolygonStipple** function sets the polygon stippling pattern.

Syntax

```
void WINAPI glPolygonStipple(
   const GLubyte *mask
);
```

Parameters

m
a
s
k

A pointer to a 32x32 stipple pattern that will be unpacked from memory in the same way that **glDrawPixels** unpacks pixels.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glPolygonStipple** function sets the polygon stippling pattern. Polygon stippling, like line stippling (see **glLineStipple**), masks out certain fragments produced by rasterization, creating a pattern. Stippling is independent of polygon antialiasing.

The *mask* parameter is a pointer to a 32x32 stipple pattern that is stored in memory just like the pixel data supplied to **glDrawPixels** with *height* and *width* both equal to 32, a pixel *format* of GL_COLOR_INDEX, and data *type* of GL_BITMAP. That is, the stipple pattern is represented as a 32x32 array of 1-bit color indexes packed in unsigned bytes. The **glPixelStore** function parameters, such as GL_UNPACK_SWAP_BYTES and GL_UNPACK_LSB_FIRST, affect the assembling of the bits into a stipple pattern. Pixel transfer operations (shift, offset, and pixel map) are not applied to the stipple image, however.

Polygon stippling is enabled and disabled with **glEnable** and **glDisable**, using argument GL_POLYGON_STIPPLE. If enabled, a rasterized polygon fragment with window coordinates x_w and y_w is sent to the next stage of OpenGL if and only if the (x_w mod 32)th bit in the (y_w mod 32)th row of the stipple pattern is one. When polygon stippling is disabled, it is as if the stipple pattern were all ones.

The following functions retrieve information related to **glPolygonStipple**:

glGetPolygonStipple

glIsEnabled with argument GL_POLYGON_STIPPLE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib

DLL	Opengl32.dll

See also

glBegin

glDrawPixels

glEnd

glLineStipple

glP

i
x
e
l
S
t
o
r
e

g
l
P
i
x
e
l
T
r
a
n
s
f
e
r

glPopAttrib function

2/22/2020 • 2 minutes to read • Edit Online

Pops the attribute stack.

Syntax

```
void WINAPI glPopAttrib(void);
```

Parameters

This function has no parameters.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_STACK_UNDERFLOW	The function was called while the attribute stack was empty.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_STACK_UNDERFLOW	The function was called while the attribute stack was empty.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glPushAttrib** function takes one argument, a mask that indicates which groups of state variables to save on the attribute stack. Symbolic constants are used to set bits in the mask. The mask parameter is typically constructed by **OR**ing several of these constants together. The special mask GL_ALL_ATTRIB_BITS can be used to save all stackable states.

The **glPopAttrib** function restores the values of the state variables saved with the last **glPushAttrib** command. Those not saved are left unchanged.

It is an error to push attributes onto a full stack, or to pop attributes off an empty stack. In either case, the error flag is set and no other change is made to the OpenGL state.

Initially, the attribute stack is empty.

Not all values for the OpenGL state can be saved on the attribute stack. For example, pixel pack and unpack state, render mode state, and select and feedback state cannot be saved.

The depth of the attribute stack depends on the implementation, but it must be at least 16.

The following functions retrieve information related to **glPushAttrib** and **glPopAttrib**:

glGet with argument GL_ATTRIB_STACK_DEPTH

glGet with argument GL_MAX_ATTRIB_STACK_DEPTH

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGet

glGetClipPlane

glGe

t
E
r
r
o
r

g
l
G
e
t
L
i
g
h
t

g
l
G
e
t
M
a
p

g
l
G
e
t
M
a
t
e
r
i
a
l

g
l
G
e
t
P
i
x
e
l
M
a

p

g
l
G
e
t
P
o
l
y
g
o
n
S
t
i
p
p
l
e

g
l
G
e
t
S
t
r
i
n
g

g
l
G
e
t
T
e
x
E
n
v

g
l
G
e
t
T
e

x
G
e
n

g
l
G
e
t
T
e
x
I
m
a
g
e

g
l
G
e
t
T
e
x
L
e
v
e
l
P
a
r
a
m
e
t
e
r

g
l
G
e
t
T
e
x
P
a
r

a
m
e
t
e
r

g
l
l
s
E
n
a
b
l
e
d

glPopClientAttrib function

2/22/2020 • 2 minutes to read • Edit Online

The **glPushClientAttrib** and **glPopClientAttrib** functions save and restore groups of client-state variables on the client-attribute stack.

Syntax

```
void WINAPI glPopClientAttrib(void);
```

Parameters

This function has no parameters.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_STACK_OVERFLOW	The function was called while the client-attribute stack was full.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_STACK_OVERFLOW	The function was called while the client-attribute stack was full.

Remarks

The **glPushClientAttrib** function uses its mask parameter to determine which groups of client-state variables are saved on the client-attribute stack. You can use the bitwise OR operator to join together accepted symbolic constants to set bits and construct a mask.

The **glPopClientAttrib** function restores the values of the client-state variables last saved with **glPushclientAttrib**. Client-state variables not previously saved are left unchanged. Pushing attributes onto a full client-attribute stack or popping attributes off an empty stack sets an error flag and no other change is made to the OpenGL state. By default the client attribute stack is empty.

Some OpenGL client-state values cannot be saved on the client-attribute stack. For example, you cannot save the select or feedback states on the client-attribute stack. The depth of the client-attribute stack is at least 16.

The **glPushclientAttrib** and **glPopClientAttrib** functions are not compiled into display lists, but are executed immediately.

The **glPushClientAttrib** and **glPopClientAttrib** functions can only push and pop pixel storage modes and vertex array client states. You must use glPushAttrib and glPopAttrib to push and pop states that are kept on the server.

> **NOTE**
> The **glPushClientAttrib** and **glPopClientAttrib** functions are only available in OpenGL version 1.1 or later.

The following functions retrieve information related to **glPushClientAttrib** and **glPopClientAttrib**:

glGet with argument GL_CLIENT_ATTRIB_STACK_DEPTH

glGet with argument GL_MAX_CLIENT_ATTRIB_STACK_DEPTH

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

g
l
c

o
l
o
r
P
o
i
n
t
e
r

g
l
D
i
s
a
b
l
e
C
l
i
e
n
t
S
t
a
t
e

g
l
E
d
g
e
F
l
a
g
P
o
i
n
t
e
r

g
l

EnableClientState

glGet

glGetError

glIndexPointer

glN

o
r
m
a
l
P
o
i
n
t
e
r

g
l
N
e
w
L
i
s
t

g
l
P
i
x
e
l
S
t
o
r
e

g
l
P
u
s
h
A
t
t
r
i
b

g
l
T
e

x
C
o
o
r
d
P
o
i
n
t
e
r

g
l
V
e
r
t
e
x
P
o
i
n
t
e
r

glPopMatrix function

2/22/2020 • 2 minutes to read • Edit Online

The **glPushMatrix** and **glPopMatrix** functions push and pop the current matrix stack.

Syntax

```
void WINAPI glPopMatrix(void);
```

Parameters

This function has no parameters.

Return value

This function does not return a value.

Error codes

It is an error to push a full matrix stack, or to pop a matrix stack that contains only a single matrix. In either case, the error flag is set and no other change is made to the OpenGL state.

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_STACK_UNDERFLOW	The function was called while the current matrix stack contained only a single matrix.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

It is an error to push a full matrix stack, or to pop a matrix stack that contains only a single matrix. In either case, the error flag is set and no other change is made to the OpenGL state.

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_STACK_UNDERFLOW	The function was called while the current matrix stack contained only a single matrix.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

There is a stack of matrices for each of the matrix modes. In GL_MODELVIEW mode, the stack depth is at least 32. In the other two modes, GL_PROJECTION and GL_TEXTURE, the depth is at least 2. The current matrix in any mode is the matrix on the top of the stack for that mode.

The **glPushMatrix** function pushes the current matrix stack down by one, duplicating the current matrix. That is, after a **glPushMatrix** call, the matrix on the top of the stack is identical to the one below it. The **glPopMatrix** function pops the current matrix stack, replacing the current matrix with the one below it on the stack. Initially, each of the stacks contains one matrix, an identity matrix.

The following functions retrieve information related to **glPushMatrix** and **glPopMatrix**:

glGet with argument GL_MATRIX_MODE

glGet with argument GL_MODELVIEW_MATRIX

glGet with argument GL_PROJECTION_MATRIX

glGet with argument GL_TEXTURE_MATRIX

glGet with argument GL_MODELVIEW_STACK_DEPTH

glGet with argument GL_PROJECTION_STACK_DEPTH

glGet with argument GL_TEXTURE_STACK_DEPTH

glGet with argument GL_MAX_MODELVIEW_STACK_DEPTH

glGet with argument GL_MAX_PROJECTION_STACK_DEPTH

glGet with argument GL_MAX_TEXTURE_STACK_DEPTH

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glFrustum

glLoadIden

t
i
t
y

g
l
L
o
a
d
M
a
t
r
i
x

g
l
M
a
t
r
i
x
M
o
d
e

g
l
M
u
l
t
M
a
t
r
i
x

g
l
O
r
t
h
o

g

lPushMatrix

glRotate

glScale

glTranslate

glViewport

glPopName function

2/22/2020 • 2 minutes to read • Edit Online

The **glPushName** and **glPopName** functions push and pop the name stack.

Syntax

```
void WINAPI glPopName(void);
```

Parameters

This function has no parameters.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_STACK_UNDERFLOW	The function was called while the current matrix stack contained only a single matrix.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_STACK_UNDERFLOW	The function was called while the current matrix stack contained only a single matrix.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glPushName** function causes name to be pushed onto the name stack, which is initially empty. The **glPopName** function pops one name off the top of the stack. The name stack is used during selection mode to allow sets of rendering commands to be uniquely identified. It consists of an ordered set of unsigned integers.

The name stack is always empty while the render mode is not GL_SELECT. Calls to **glPushName** or **glPopName** while the render mode is not GL_SELECT are ignored.

The following functions retrieve information related to **glPushName** and **glPopName**:

glGet with argument GL_NAME_STACK_DEPTH

glGet with argument GL_MAX_NAME_STACK_DEPTH

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glInitNames

glLoadName

glRenderMode

g

I
S
e
l
e
c
t
B
u
f
f
e
r

glPrioritizeTextures function

2/22/2020 • 2 minutes to read • Edit Online

The **glPrioritizeTextures** function sets the residence priority of textures.

Syntax

```
void WINAPI glPrioritizeTextures(
         GLsizei  n,
   const GLuint   *textures,
   const GLclampf *priorities
);
```

Parameters

n
The number of textures to be prioritized.

textures
A pointer to the first element of an array containing the names of the textures to be prioritized.

priorities
A pointer to the first element of an array containing the texture priorities. A priority given in an element of the *priorities* parameter applies to the texture named by the corresponding element of the *textures* parameter.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	*n* was a negative value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	*n* was a negative value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glPrioritizeTextures** function assigns the *n* texture priorities specified in the *priorities* parameter to the *n* textures named in the *textures* parameter. On computers with a limited amount of texture memory, OpenGL establishes a "working set" of textures that are resident in texture memory. These textures can be bound to a texture target much more efficiently than textures that are not resident.

By specifying a priority for each texture, the **glPrioritizeTextures** function enables you to determine which textures should be resident.

The texture priorities elements in *priorities* are clamped to the range [0.0, 1.0] before being assigned. Zero indicates the lowest priority; thus textures with priority zero are least likely to be resident. The value 1.0 indicates the highest priority; thus textures with priority 1.0 are most likely to be resident. However, textures are not

guaranteed to be resident until they are bound.

The **glPrioritizeTextures** function ignores attempts to prioritize texture 0, or any texture name that does not correspond to an existing texture. None of the functions named by the *textures* parameter need to be bound to a texture target.

If a texture is currently bound, you can also use the glTexParameter function to set its priority. This is the only way to set the priority of a default texture.

You can include **glPrioritizeTextures** in display lists.

The following function retrieves the priority of a currently-bound texture related to **glPrioritizeTextures**:

- glGetTexParameter with parameter name GL_TEXTURE_PRIORITY

> **NOTE**
> The **glPrioritizeTextures** function is only available in OpenGL version 1.1 or later.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glAreTexture

s
R
e
s
i
d
e
n
t

g
l
B
e
g
i
n

g
l
E
n
d

g
l
G
e
t
T
e
x
P
a
r
a
m
e
t
e
r

g
l
T
e
x
I
m
a
g
e
1

D

g
l
T
e
x
I
m
a
g
e
2
D

g
l
T
e
x
P
a
r
a
m
e
t
e
r

glPushAttrib function

2/22/2020 • 4 minutes to read • Edit Online

Pushes the attribute stack.

Syntax

```
void WINAPI glPushAttrib(
   GLbitfield mask
);
```

Parameters

m
a
s
k

A mask that indicates which attributes to save. The symbolic mask constants and their associated OpenGL state are as follows (the indented paragraphs list which attributes are saved):

GL_ACCUM_BUFFER_BIT	Accumulation buffer clear value
GL_COLO	GL_ALPHA_TEST enable bit Alpha test function and reference value GL_BLEND enable bit Blending source and destination functions GL_DITHER enable bit

R_BUFFER_BIT	GL_DRAW_BUFFER setting
	GL_LOGIC_OP enable bit
	Logic op function
	Color-mode and index-mode clear values
	Color-mode and index-mode writemasks
GL_CURRENT_BIT	Current RGBA color
	Current color index
	Current normal vector
	Current texture coordinates
	Current raster position GL_CURRENT_RASTER_POSITION_VALID flag
	RGBA color associated with current raster position
	Color index associated with current raster position
	Texture coordinates associated with current raster position
	GL_EDGE_FLAG flag
GL_DEPTH_BUFFER_BIT	GL_DEPTH_TEST enable bit
	Depth buffer test function
	Depth buffer clear value
	GL_DEPTH_WRITEMASK enable bit
GL_EN	GL_ALPHA_TEST flag
	GL_AUTO_NORMAL flag
	GL_BLEND flag

ABLE_BIT	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_NORMALIZE flag
	GL_POINT_SMOOTH flag
	GL_POLYGON_OFFSET_LINE flag
	GL_POLYGON_OFFSET_FILL flag
	GL_POLYGON_OFFSET_POINT flag
	GL_POLYGON_SMOOTH flag
	GL_POLYGON_STIPPLE flag
	GL_SCISSOR_TEST flag
	GL_STENCIL_TEST flag
	GL_TEXTURE_1D flag
	GL_TEXTURE_2D flag
	Flags GL_TEXTURE_GEN_x where x is S, T, R, or Q
GL_EVAL_BIT	GL_MAP1_x enable bits, where x is a map type
	GL_MAP2_x enable bits, where x is a map type
	1-D grid endpoints and divisions
	2-D grid endpoints and divisions
	GL_AUTO_NORMAL enable bit

GL_FOG_BIT	GL_FOG enable flag
	Fog color
	Fog density
	Linear fog start
	Linear fog end
	Fog index
	GL_FOG_MODE value
GL_HINT_BIT	GL_PERSPECTIVE_CORRECTION_HINT setting
	GL_POINT_SMOOTH_HINT setting
	GL_LINE_SMOOTH_HINT setting
	GL_POLYGON_SMOOTH_HINT setting
	GL_FOG_HINT setting
GL_LIGHTING_BIT	GL_COLOR_MATERIAL enable bit
	GL_COLOR_MATERIAL_FACE value
	Color material parameters that are tracking the current color
	Ambient scene color
	GL_LIGHT_MODEL_LOCAL_VIEWER value
	GL_LIGHT_MODEL_TWO_SIDE setting
	GL_LIGHTING enable bit
	Enable bit for each light
	Ambient, diffuse, and specular intensity for each light
	Direction, position, exponent, and cutoff angle for each light
	Constant, linear, and quadratic attenuation factors for each light
	Ambient, diffuse, specular, and emissive color for each material
	Ambient, diffuse, and specular color indexes for each material
	Specular exponent for each material GL_SHADE_MODEL setting
GL_LINE	GL_LINE_SMOOTH flag
	GL_LINE_STIPPLE enable bit
	Line stipple pattern and repeat counter
	Line width

_
B
I
T

G
L
_
L
I
S
T
_
B
I
T

GL_LIST_BASE setting

G
L
_
P
I
X
E
L
_
M
O
D
E
_
B
I
T

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

G
L
_
P
O
I
N
T
_
B
I
T

GL_POINT_SMOOTH flag

Point size

G
L
_
P
O

GL_CULL_FACE enable bit

GL_CULL_FACE_MODE value

GL_FRONT_FACE indicator

L Y G O N _ B I T	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
G L _ P O L Y G O N _ S T I P P L E _ B I T	Polygon stipple image
G L _ S C I S S O R _ B I T	GL_SCISSOR_TEST flag Scissor box
G L	GL_STENCIL_TEST enable bit

GL_STENCIL_BUFFER_BIT	Stencil function and reference value
	Stencil value mask
	Stencil fail, pass, and depth buffer pass actions
	Stencil buffer clear value
	Stencil buffer writemask
GL_TEXTURE_BIT	Enable bits for the four texture coordinates
	Border color for each texture image
	Minification function for each texture image
	Magnification function for each texture image
	Texture coordinates and wrap mode for each texture image
	Color and mode for each texture environment
	Enable bits GL_TEXTURE_GEN_*x*, *x* is S, T, R, and Q
	GL_TEXTURE_GEN_MODE setting for S, T, R, and Q
	glTexGen plane equations for S, T, R, and Q
GL_TRANSFORM_BIT	Coefficients of the six clipping planes
	Enable bits for the user-definable clipping planes
	GL_MATRIX_MODE value
	GL_NORMALIZE flag
G	Depth range (near and far)

L
_
V
I
E
W
P
O
R
T
_
B
I
T

Viewport origin and extent

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_STACK_OVERFLOW	The function was called while the attribute stack was full.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_STACK_OVERFLOW	The function was called while the attribute stack was full.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glPushAttrib** function takes one argument, a mask that indicates which groups of state variables to save on the attribute stack. Symbolic constants are used to set bits in the mask. The mask parameter is typically constructed by applying the logical **OR** operation to several of these constants. You can use the special mask GL_ALL_ATTRIB_BITS to save all stackable states.

The **glPopAttrib** function restores the values of the state variables saved with the last **glPushAttrib** command. Those not saved are left unchanged.

It is an error to push attributes onto a full stack, or to pop attributes off an empty stack. In either case, the error flag is set and no other change is made to the OpenGL state.

Initially, the attribute stack is empty.

Not all values for the OpenGL state can be saved on the attribute stack. For example, you cannot save pixel pack and unpack state, render mode state, and select and feedback state.

The depth of the attribute stack depends on the implementation, but it must be at least 16.

The following functions retrieve information related to **glPushAttrib** and **glPopAttrib**:

glGet with argument GL_ATTRIB_STACK_DEPTH

glGet with argument GL_MAX_ATTRIB_STACK_DEPTH

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glGet

glGetClipPlane

glGe

t
E
r
r
o
r

g
l
G
e
t
L
i
g
h
t

g
l
G
e
t
M
a
p

g
l
G
e
t
M
a
t
e
r
i
a
l

g
l
G
e
t
P
i
x
e
l
M
a

p

glGetPolygonStipple

glGetString

glGetTexEnv

glGetTe

x
G
e
n

g
l
G
e
t
T
e
x
I
m
a
g
e

g
l
G
e
t
T
e
x
L
e
v
e
l
P
a
r
a
m
e
t
e
r

g
l
G
e
t
T
e
x
P
a
r

a
m
e
t
e
r

g
l
l
s
E
n
a
b
l
e
d

glPushClientAttrib function

2/22/2020 • 2 minutes to read • Edit Online

The **glPushClientAttrib** and **glPopClientAttrib** functions save and restore groups of client-state variables on the client-attribute stack.

Syntax

```
void WINAPI glPushClientAttrib(
   GLbitfield mask
);
```

Parameters

m
a
s
k

A mask that indicates which attributes to save. The following are the symbolic mask constants and their associated OpenGL client states.

VALUE	MEANING
GL_CLIENT_PIXEL_STORE_BIT	Pixel storage mode attributes.
GL_CLIENT_VERTEX_ARRAY_BIT	Vertex array attributes.

VALUE	MEANING
GL_ CLI ENT _AL L_A TTR IB_ BITs	all stackable client-state attributes.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the glGetError function.

NAME	MEANING
GL_ STA CK_ OV ERF LO W	The function was called while the client-attribute stack was full.

Error codes

The following error code can be retrieved by the glGetError function.

NAME	MEANING
GL_ STA CK_ OV ERF LO W	The function was called while the client-attribute stack was full.

Remarks

The **glPushClientAttrib** function uses its mask parameter to determine which groups of client-state variables are saved on the client-attribute stack. You can use the bitwise OR operator to join together accepted symbolic constants to set bits and construct a mask.

The glPopClientAttrib function restores the values of the client-state variables last saved with **glPushclientAttrib**. Client-state variables not previously saved are left unchanged. Pushing attributes onto a full client-attribute stack or popping attributes off an empty stack sets an error flag and no other change is

made to the OpenGL state. By default the client attribute stack is empty.

Some OpenGL client-state values cannot be saved on the client-attribute stack. For example, you cannot save the select or feedback states on the client-attribute stack. The depth of the client-attribute stack is at least 16.

The **glPushclientAttrib** and **glPopClientAttrib** functions are not compiled into display lists, but are executed immediately.

The **glPushClientAttrib** and **glPopClientAttrib** functions can only push and pop pixel storage modes and vertex array client states. You must use **glPushAttrib** and **glPopAttrib** to push and pop states that are kept on the server.

> **NOTE**
>
> The **glPushClientAttrib** and **glPopClientAttrib** functions are only available in OpenGL version 1.1 or later.

The following functions retrieve information related to **glPushClientAttrib** and **glPopClientAttrib**:

glGet with argument GL_CLIENT_ATTRIB_STACK_DEPTH

glGet with argument GL_MAX_CLIENT_ATTRIB_STACK_DEPTH

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glColorPoi

n
t
e
r

g
l
D
i
s
a
b
l
e
C
l
i
e
n
t
S
t
a
t
e

g
l
E
d
g
e
F
l
a
g
P
o
i
n
t
e
r

g
l
E
n
a
b
l
e
C

glClientState

glGet

glGetError

glIndexPointer

glNormalPo

i
n
t
e
r

g
l
N
e
w
L
i
s
t

g
l
P
i
x
e
l
S
t
o
r
e

g
l
P
u
s
h
A
t
t
r
i
b

g
l
T
e
x
C
o
o
r
d
P

o
i
n
t
e
r

g
l
V
e
r
t
e
x
P
o
i
n
t
e
r

glPushMatrix function

2/22/2020 • 2 minutes to read • Edit Online

The **glPushMatrix** and **glPopMatrix** functions push and pop the current matrix stack.

Syntax

```
void WINAPI glPushMatrix(void);
```

Parameters

This function has no parameters.

Return value

This function does not return a value.

Error codes

It is an error to push a full matrix stack, or to pop a matrix stack that contains only a single matrix. In either case, the error flag is set and no other change is made to the OpenGL state.

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_STACK_OVERFLOW	The function was called while the current matrix stack was full.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

It is an error to push a full matrix stack, or to pop a matrix stack that contains only a single matrix. In either case, the error flag is set and no other change is made to the OpenGL state.

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_STACK_OVERFLOW	The function was called while the current matrix stack was full.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

There is a stack of matrices for each of the matrix modes. In GL_MODELVIEW mode, the stack depth is at least 32. In the other two modes, GL_PROJECTION and GL_TEXTURE, the depth is at least 2. The current matrix in any mode is the matrix on the top of the stack for that mode.

The **glPushMatrix** function pushes the current matrix stack down by one, duplicating the current matrix. That is, after a **glPushMatrix** call, the matrix on the top of the stack is identical to the one below it. The **glPopMatrix** function pops the current matrix stack, replacing the current matrix with the one below it on the stack. Initially, each of the stacks contains one matrix, an identity matrix.

The following functions retrieve information related to **glPushMatrix** and **glPopMatrix**:

glGet with argument GL_MATRIX_MODE

glGet with argument GL_MODELVIEW_MATRIX

glGet with argument GL_PROJECTION_MATRIX

glGet with argument GL_TEXTURE_MATRIX

glGet with argument GL_MODELVIEW_STACK_DEPTH

glGet with argument GL_PROJECTION_STACK_DEPTH

glGet with argument GL_TEXTURE_STACK_DEPTH

glGet with argument GL_MAX_MODELVIEW_STACK_DEPTH

glGet with argument GL_MAX_PROJECTION_STACK_DEPTH

glGet with argument GL_MAX_TEXTURE_STACK_DEPTH

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glFrustum

glLoadId

e
n
t
i
t
y

g
l
L
o
a
d
M
a
t
r
i
x

g
l
M
a
t
r
i
x
M
o
d
e

g
l
M
u
l
t
M
a
t
r
i
x

g
l
O
r
t
h
o

glPopMatrix

glRotate

glScale

glTranslate

glViewport

glPushName function

2/22/2020 • 2 minutes to read • Edit Online

The **glPushName** and **glPopName** functions push and pop the name stack.

Syntax

```
void WINAPI glPushName(
   GLuint name
);
```

Parameters

name	A name that will be pushed onto the name stack.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_STACK_OVERFLOW	The function was called while the current matrix stack was full.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_STACK_OVERFLOW	The function was called while the current matrix stack was full.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glPushName** function causes name to be pushed onto the name stack, which is initially empty. The **glPopName** function pops one name off the top of the stack. The name stack is used during selection mode to allow sets of rendering commands to be uniquely identified. It consists of an ordered set of unsigned integers.

The name stack is always empty while the render mode is not GL_SELECT. Calls to **glPushName** or **glPopName** while the render mode is not GL_SELECT are ignored.

The following functions retrieve information related to **glPushName** and **glPopName**:

glGet with argument GL_NAME_STACK_DEPTH

glGet with argument GL_MAX_NAME_STACK_DEPTH

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib

DLL	Opengl32.dll

See also

glBegin

glEnd

glInitNames

glLoadName

glRende

r
M
o
d
e

g
l
S
e
l
e
c
t
B
u
f
f
e
r

glRasterPos Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions specify the raster position for pixel operations:

- glRasterPos2d
- glRasterPos2f
- glRasterPos2i
- glRasterPos2s
- glRasterPos3d
- glRasterPos3f
- glRasterPos3i
- glRasterPos3s
- glRasterPos4d
- glRasterPos4f
- glRasterPos4i
- glRasterPos4s
- glRasterPos2dv
- glRasterPos2fv
- glRasterPos2iv
- glRasterPos2sv
- glRasterPos3dv
- glRasterPos3fv
- glRasterPos3iv
- glRasterPos3sv
- glRasterPos4dv
- glRasterPos4fv
- glRasterPos4iv
- glRasterPos4sv

glRasterPos2d function

2/22/2020 • 2 minutes to read • Edit Online

Specifies the raster position for pixel operations.

Syntax

```
void WINAPI glRasterPos2d(
   GLdouble x,
   GLdouble y
);
```

Parameters

x	Specifies the x-coordinate for the current raster position.
y	Specifies the y-coordinate for the current raster position.

Return value

This function does not return a value.

Remarks

OpenGL maintains a 3-D position in window coordinates. This position, called the raster position, is maintained with subpixel accuracy. It is used to position pixel and bitmap write operations. See **glBitmap**, **glDrawPixels**, and **glCopyPixels**.

The current raster position consists of three window coordinates (*x, y, z*), a clip coordinate *w* value, an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The *w* coordinate is a clip coordinate, because *w* is not projected to window coordinates. The glRasterPos4 function specifies object coordinates *x, y, z*, and *w* explicitly. The glRasterPos3 function specifies object coordinates *x, y,* and *z* explicitly, while *w* is implicitly set to one. The glRasterPos2 function uses the argument values for *x* and *y* while implicitly setting *z* and *w* to zero and one.

The object coordinates presented by glRasterPos are treated just like those of a glVertex command. They are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, then it is projected and scaled to window coordinates, which become the new current raster position, and the GL_CURRENT_RASTER_POSITION_VALID flag is set. If the vertex is culled, then the valid bit is cleared and the current raster position and associated color and texture coordinates are undefined.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then GL_CURRENT_RASTER_COLOR, in RGBA mode, or the GL_CURRENT_RASTER_INDEX, in color-index mode, is set to the color produced by the lighting calculation (see glLight, glLightModel, and **glShadeModel**). If lighting is disabled, current color (in RGBA mode, state variable GL_CURRENT_COLOR) or color index (in color-index mode, state variable GL_CURRENT_INDEX) is used to update the current raster color.

Likewise, GL_CURRENT_RASTER_TEXTURE_COORDS is updated as a function of GL_CURRENT_TEXTURE_COORDS, based on the texture matrix and the texture generation functions (see glTexGen). Finally, the distance from the origin of the eye coordinate system to the vertex, as transformed by only the modelview matrix, replaces

GL_CURRENT_RASTER_DISTANCE.

Initially, the current raster position is (0,0,0,1), the current raster distance is 0, the valid bit is set, the associated RGBA color is (1,1,1,1), the associated color index is 1, and the associated texture coordinates are (0, 0, 0, 1). In RGBA mode, GL_CURRENT_RASTER_INDEX is always 1; in color-index mode, the current raster RGBA color always maintains its initial value.

> **NOTE**
> The raster position is modified both by glRasterPos and by **glBitmap**.

> **NOTE**
> When the raster position coordinates are invalid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to the OpenGL state).

The following functions retrieve information related to glRasterPos:

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID
glGet with argument GL_CURRENT_RASTER_DISTANCE
glGet with argument GL_CURRENT_RASTER_COLOR
glGet with argument GL_CURRENT_RASTER_INDEX
glGet with argument GL_CURRENT_RASTER_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBe

g
i
n

g
l
B
i
t
m
a
p

g
l
C
o
p
y
P
i
x
e
l
s

g
l
D
r
a
w
P
i
x
e
l
s

g
l
E
n
d

g
l
L
i
g
h
t

g

lLightModel

glShadeModel

glTexCoord

glTexGen

glVertex

glRasterPos2dv function

2/22/2020 • 2 minutes to read • Edit Online

Specifies the raster position for pixel operations.

Syntax

```
void WINAPI glRasterPos2dv(
   const GLdouble *v
);
```

Parameters

v	A pointer to an array of two elements, specifying x and y coordinates for the current raster position.

Return value

This function does not return a value.

Remarks

OpenGL maintains a 3-D position in window coordinates. This position, called the raster position, is maintained with subpixel accuracy. It is used to position pixel and bitmap write operations. See **glBitmap**, **glDrawPixels**, and **glCopyPixels**.

The current raster position consists of three window coordinates (*x, y, z*), a clip coordinate *w* value, an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The *w* coordinate is a clip coordinate, because *w* is not projected to window coordinates. The glRasterPos4 function specifies object coordinates *x, y, z,* and *w* explicitly. The glRasterPos3 function specifies object coordinates *x, y,* and *z* explicitly, while *w* is implicitly set to one. The glRasterPos2 function uses the argument values for *x* and *y* while implicitly setting *z* and *w* to zero and one.

The object coordinates presented by glRasterPos are treated just like those of a glVertex command. They are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, then it is projected and scaled to window coordinates, which become the new current raster position, and the GL_CURRENT_RASTER_POSITION_VALID flag is set. If the vertex is culled, then the valid bit is cleared and the current raster position and associated color and texture coordinates are undefined.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then GL_CURRENT_RASTER_COLOR, in RGBA mode, or the GL_CURRENT_RASTER_INDEX, in color-index mode, is set to the color produced by the lighting calculation (see glLight, glLightModel, and **glShadeModel**). If lighting is disabled, current color (in RGBA mode, state variable GL_CURRENT_COLOR) or color index (in color-index mode, state variable GL_CURRENT_INDEX) is used to update the current raster color.

Likewise, GL_CURRENT_RASTER_TEXTURE_COORDS is updated as a function of GL_CURRENT_TEXTURE_COORDS, based on the texture matrix and the texture generation functions (see glTexGen). Finally, the distance from the origin of the eye coordinate system to the vertex, as transformed by only the modelview matrix, replaces GL_CURRENT_RASTER_DISTANCE.

Initially, the current raster position is (0,0,0,1), the current raster distance is 0, the valid bit is set, the associated

RGBA color is (1,1,1,1), the associated color index is 1, and the associated texture coordinates are (0, 0, 0, 1). In RGBA mode, GL_CURRENT_RASTER_INDEX is always 1; in color-index mode, the current raster RGBA color always maintains its initial value.

> **NOTE**
> The raster position is modified both by glRasterPos and by **glBitmap**.

> **NOTE**
> When the raster position coordinates are invalid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to the OpenGL state).

The following functions retrieve information related to glRasterPos:

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID
glGet with argument GL_CURRENT_RASTER_DISTANCE
glGet with argument GL_CURRENT_RASTER_COLOR
glGet with argument GL_CURRENT_RASTER_INDEX
glGet with argument GL_CURRENT_RASTER_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

glBitmap

glCopyPixels

glDrawPixels

glEnd

glLight

glL

i
g
h
t
M
o
d
e
l

g
l
S
h
a
d
e
M
o
d
e
l

g
l
T
e
x
C
o
o
r
d

g
l
T
e
x
G
e
n

g
l
V
e
r
t
e
x

glRasterPos2f function

2/22/2020 • 2 minutes to read • Edit Online

Specifies the raster position for pixel operations.

Syntax

```
void WINAPI glRasterPos2f(
   GLfloat x,
   GLfloat y
);
```

Parameters

x	Specifies the x-coordinate for the current raster position.
y	Specifies the y-coordinate for the current raster position.

Return value

This function does not return a value.

Remarks

OpenGL maintains a 3-D position in window coordinates. This position, called the raster position, is maintained with subpixel accuracy. It is used to position pixel and bitmap write operations. See **glBitmap**, **glDrawPixels**, and **glCopyPixels**.

The current raster position consists of three window coordinates (*x, y, z*), a clip coordinate *w* value, an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The *w* coordinate is a clip coordinate, because *w* is not projected to window coordinates. The glRasterPos4 function specifies object coordinates *x, y, z*, and *w* explicitly. The glRasterPos3 function specifies object coordinates *x, y,* and *z* explicitly, while *w* is implicitly set to one. The glRasterPos2 function uses the argument values for *x* and *y* while implicitly setting *z* and *w* to zero and one.

The object coordinates presented by glRasterPos are treated just like those of a glVertex command. They are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, then it is projected and scaled to window coordinates, which become the new current raster position, and the GL_CURRENT_RASTER_POSITION_VALID flag is set. If the vertex is culled, then the valid bit is cleared and the current raster position and associated color and texture coordinates are undefined.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then GL_CURRENT_RASTER_COLOR, in RGBA mode, or the GL_CURRENT_RASTER_INDEX, in color-index mode, is set to the color produced by the lighting calculation (see glLight, glLightModel, and **glShadeModel**). If lighting is disabled, current color (in RGBA mode, state variable GL_CURRENT_COLOR) or color index (in color-index mode, state variable GL_CURRENT_INDEX) is used to update the current raster color.

Likewise, GL_CURRENT_RASTER_TEXTURE_COORDS is updated as a function of GL_CURRENT_TEXTURE_COORDS, based on the texture matrix and the texture generation functions (see glTexGen). Finally, the distance from the origin of the eye coordinate system to the vertex, as transformed by only the modelview matrix, replaces

GL_CURRENT_RASTER_DISTANCE.

Initially, the current raster position is (0,0,0,1), the current raster distance is 0, the valid bit is set, the associated RGBA color is (1,1,1,1), the associated color index is 1, and the associated texture coordinates are (0, 0, 0, 1). In RGBA mode, GL_CURRENT_RASTER_INDEX is always 1; in color-index mode, the current raster RGBA color always maintains its initial value.

> **NOTE**
> The raster position is modified both by glRasterPos and by **glBitmap**.

> **NOTE**
> When the raster position coordinates are invalid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to the OpenGL state).

The following functions retrieve information related to glRasterPos:

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID
glGet with argument GL_CURRENT_RASTER_DISTANCE
glGet with argument GL_CURRENT_RASTER_COLOR
glGet with argument GL_CURRENT_RASTER_INDEX
glGet with argument GL_CURRENT_RASTER_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBe

g
i
n

g
l
B
i
t
m
a
p

g
l
C
o
p
y
P
i
x
e
l
s

g
l
D
r
a
w
P
i
x
e
l
s

g
l
E
n
d

g
l
L
i
g
h
t

g

lLightModel

glShadeModel

glTexCoord

glTexGen

glVertex

glRasterPos2fv function

2/22/2020 • 2 minutes to read • Edit Online

Specifies the raster position for pixel operations.

Syntax

```
void WINAPI glRasterPos2fv(
   const GLfloat *v
);
```

Parameters

v	A pointer to an array of two elements, specifying x and y coordinates for the current raster position.

Return value

This function does not return a value.

Remarks

OpenGL maintains a 3-D position in window coordinates. This position, called the raster position, is maintained with subpixel accuracy. It is used to position pixel and bitmap write operations. See **glBitmap**, **glDrawPixels**, and **glCopyPixels**.

The current raster position consists of three window coordinates (*x, y, z*), a clip coordinate *w* value, an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The *w* coordinate is a clip coordinate, because *w* is not projected to window coordinates. The glRasterPos4 function specifies object coordinates *x, y, z,* and *w* explicitly. The glRasterPos3 function specifies object coordinates *x, y,* and *z* explicitly, while *w* is implicitly set to one. The glRasterPos2 function uses the argument values for *x* and *y* while implicitly setting *z* and *w* to zero and one.

The object coordinates presented by glRasterPos are treated just like those of a glVertex command. They are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, then it is projected and scaled to window coordinates, which become the new current raster position, and the GL_CURRENT_RASTER_POSITION_VALID flag is set. If the vertex is culled, then the valid bit is cleared and the current raster position and associated color and texture coordinates are undefined.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then GL_CURRENT_RASTER_COLOR, in RGBA mode, or the GL_CURRENT_RASTER_INDEX, in color-index mode, is set to the color produced by the lighting calculation (see glLight, glLightModel, and **glShadeModel**). If lighting is disabled, current color (in RGBA mode, state variable GL_CURRENT_COLOR) or color index (in color-index mode, state variable GL_CURRENT_INDEX) is used to update the current raster color.

Likewise, GL_CURRENT_RASTER_TEXTURE_COORDS is updated as a function of GL_CURRENT_TEXTURE_COORDS, based on the texture matrix and the texture generation functions (see glTexGen). Finally, the distance from the origin of the eye coordinate system to the vertex, as transformed by only the modelview matrix, replaces GL_CURRENT_RASTER_DISTANCE.

Initially, the current raster position is (0,0,0,1), the current raster distance is 0, the valid bit is set, the associated

RGBA color is (1,1,1,1), the associated color index is 1, and the associated texture coordinates are (0, 0, 0, 1). In RGBA mode, GL_CURRENT_RASTER_INDEX is always 1; in color-index mode, the current raster RGBA color always maintains its initial value.

> **NOTE**
> The raster position is modified both by glRasterPos and by **glBitmap**.

> **NOTE**
> When the raster position coordinates are invalid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to the OpenGL state).

The following functions retrieve information related to glRasterPos:

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID
glGet with argument GL_CURRENT_RASTER_DISTANCE
glGet with argument GL_CURRENT_RASTER_COLOR
glGet with argument GL_CURRENT_RASTER_INDEX
glGet with argument GL_CURRENT_RASTER_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

glBitmap

glCopyPixels

glDrawPixels

glEnd

glLight

glL

i
g
h
t
M
o
d
e
l

g
l
S
h
a
d
e
M
o
d
e
l

g
l
T
e
x
C
o
o
r
d

g
l
T
e
x
G
e
n

g
l
V
e
r
t
e
x

glRasterPos2i function

2/22/2020 • 2 minutes to read • Edit Online

Specifies the raster position for pixel operations.

Syntax

```
void WINAPI glRasterPos2i(
   GLint x,
   GLint y
);
```

Parameters

x	Specifies the x-coordinate for the current raster position.
y	Specifies the y-coordinate for the current raster position.

Return value

This function does not return a value.

Remarks

OpenGL maintains a 3-D position in window coordinates. This position, called the raster position, is maintained with subpixel accuracy. It is used to position pixel and bitmap write operations. See **glBitmap**, **glDrawPixels**, and **glCopyPixels**.

The current raster position consists of three window coordinates (*x, y, z*), a clip coordinate *w* value, an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The *w* coordinate is a clip coordinate, because *w* is not projected to window coordinates. The glRasterPos4 function specifies object coordinates *x, y, z*, and *w* explicitly. The glRasterPos3 function specifies object coordinates *x, y,* and *z* explicitly, while *w* is implicitly set to one. The glRasterPos2 function uses the argument values for *x* and *y* while implicitly setting *z* and *w* to zero and one.

The object coordinates presented by glRasterPos are treated just like those of a glVertex command. They are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, then it is projected and scaled to window coordinates, which become the new current raster position, and the GL_CURRENT_RASTER_POSITION_VALID flag is set. If the vertex is culled, then the valid bit is cleared and the current raster position and associated color and texture coordinates are undefined.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then GL_CURRENT_RASTER_COLOR, in RGBA mode, or the GL_CURRENT_RASTER_INDEX, in color-index mode, is set to the color produced by the lighting calculation (see glLight, glLightModel, and **glShadeModel**). If lighting is disabled, current color (in RGBA mode, state variable GL_CURRENT_COLOR) or color index (in color-index mode, state variable GL_CURRENT_INDEX) is used to update the current raster color.

Likewise, GL_CURRENT_RASTER_TEXTURE_COORDS is updated as a function of GL_CURRENT_TEXTURE_COORDS, based on the texture matrix and the texture generation functions (see glTexGen). Finally, the distance from the origin of the eye coordinate system to the vertex, as transformed by only the modelview matrix, replaces

GL_CURRENT_RASTER_DISTANCE.

Initially, the current raster position is (0,0,0,1), the current raster distance is 0, the valid bit is set, the associated RGBA color is (1,1,1,1), the associated color index is 1, and the associated texture coordinates are (0, 0, 0, 1). In RGBA mode, GL_CURRENT_RASTER_INDEX is always 1; in color-index mode, the current raster RGBA color always maintains its initial value.

> **NOTE**
> The raster position is modified both by glRasterPos and by **glBitmap**.

> **NOTE**
> When the raster position coordinates are invalid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to the OpenGL state).

The following functions retrieve information related to glRasterPos:

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID
glGet with argument GL_CURRENT_RASTER_DISTANCE
glGet with argument GL_CURRENT_RASTER_COLOR
glGet with argument GL_CURRENT_RASTER_INDEX
glGet with argument GL_CURRENT_RASTER_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBe

g
i
n

g
l
B
i
t
m
a
p

g
l
C
o
p
y
P
i
x
e
l
s

g
l
D
r
a
w
P
i
x
e
l
s

g
l
E
n
d

g
l
L
i
g
h
t

g

lLightModel

glShadeModel

glTexCoord

glTexGen

glVertex

glRasterPos2iv function

2/22/2020 • 2 minutes to read • Edit Online

Specifies the raster position for pixel operations.

Syntax

```
void WINAPI glRasterPos2dv(
   const GLint *v
);
```

Parameters

v	A pointer to an array of two elements, specifying x and y coordinates for the current raster position.

Return value

This function does not return a value.

Remarks

OpenGL maintains a 3-D position in window coordinates. This position, called the raster position, is maintained with subpixel accuracy. It is used to position pixel and bitmap write operations. See **glBitmap**, **glDrawPixels**, and **glCopyPixels**.

The current raster position consists of three window coordinates (*x, y, z*), a clip coordinate *w* value, an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The *w* coordinate is a clip coordinate, because *w* is not projected to window coordinates. The glRasterPos4 function specifies object coordinates *x, y, z*, and *w* explicitly. The glRasterPos3 function specifies object coordinates *x, y,* and *z* explicitly, while *w* is implicitly set to one. The glRasterPos2 function uses the argument values for *x* and *y* while implicitly setting *z* and *w* to zero and one.

The object coordinates presented by glRasterPos are treated just like those of a glVertex command. They are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, then it is projected and scaled to window coordinates, which become the new current raster position, and the GL_CURRENT_RASTER_POSITION_VALID flag is set. If the vertex is culled, then the valid bit is cleared and the current raster position and associated color and texture coordinates are undefined.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then GL_CURRENT_RASTER_COLOR, in RGBA mode, or the GL_CURRENT_RASTER_INDEX, in color-index mode, is set to the color produced by the lighting calculation (see glLight, glLightModel, and **glShadeModel**). If lighting is disabled, current color (in RGBA mode, state variable GL_CURRENT_COLOR) or color index (in color-index mode, state variable GL_CURRENT_INDEX) is used to update the current raster color.

Likewise, GL_CURRENT_RASTER_TEXTURE_COORDS is updated as a function of GL_CURRENT_TEXTURE_COORDS, based on the texture matrix and the texture generation functions (see glTexGen). Finally, the distance from the origin of the eye coordinate system to the vertex, as transformed by only the modelview matrix, replaces GL_CURRENT_RASTER_DISTANCE.

Initially, the current raster position is (0,0,0,1), the current raster distance is 0, the valid bit is set, the associated

RGBA color is (1,1,1,1), the associated color index is 1, and the associated texture coordinates are (0, 0, 0, 1). In RGBA mode, GL_CURRENT_RASTER_INDEX is always 1; in color-index mode, the current raster RGBA color always maintains its initial value.

> **NOTE**
> The raster position is modified both by glRasterPos and by **glBitmap**.

> **NOTE**
> When the raster position coordinates are invalid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to the OpenGL state).

The following functions retrieve information related to glRasterPos:

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID
glGet with argument GL_CURRENT_RASTER_DISTANCE
glGet with argument GL_CURRENT_RASTER_COLOR
glGet with argument GL_CURRENT_RASTER_INDEX
glGet with argument GL_CURRENT_RASTER_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

glBitmap

glCopyPixels

glDrawPixels

glEnd

glLight

glL

i
g
h
t
M
o
d
e
l

g
l
S
h
a
d
e
M
o
d
e
l

g
l
T
e
x
C
o
o
r
d

g
l
T
e
x
G
e
n

g
l
V
e
r
t
e
x

glRasterPos2s function

2/22/2020 • 2 minutes to read • Edit Online

Specifies the raster position for pixel operations.

Syntax

```
void WINAPI glRasterPos2s(
   GLshort x,
   GLshort y
);
```

Parameters

x	Specifies the x-coordinate for the current raster position.
y	Specifies the y-coordinate for the current raster position.

Return value

This function does not return a value.

Remarks

OpenGL maintains a 3-D position in window coordinates. This position, called the raster position, is maintained with subpixel accuracy. It is used to position pixel and bitmap write operations. See **glBitmap**, **glDrawPixels**, and **glCopyPixels**.

The current raster position consists of three window coordinates (*x, y, z*), a clip coordinate *w* value, an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The *w* coordinate is a clip coordinate, because *w* is not projected to window coordinates. The glRasterPos4 function specifies object coordinates *x, y, z*, and *w* explicitly. The glRasterPos3 function specifies object coordinates *x, y,* and *z* explicitly, while *w* is implicitly set to one. The glRasterPos2 function uses the argument values for *x* and *y* while implicitly setting *z* and *w* to zero and one.

The object coordinates presented by glRasterPos are treated just like those of a glVertex command. They are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, then it is projected and scaled to window coordinates, which become the new current raster position, and the GL_CURRENT_RASTER_POSITION_VALID flag is set. If the vertex is culled, then the valid bit is cleared and the current raster position and associated color and texture coordinates are undefined.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then GL_CURRENT_RASTER_COLOR, in RGBA mode, or the GL_CURRENT_RASTER_INDEX, in color-index mode, is set to the color produced by the lighting calculation (see glLight, glLightModel, and **glShadeModel**). If lighting is disabled, current color (in RGBA mode, state variable GL_CURRENT_COLOR) or color index (in color-index mode, state variable GL_CURRENT_INDEX) is used to update the current raster color.

Likewise, GL_CURRENT_RASTER_TEXTURE_COORDS is updated as a function of GL_CURRENT_TEXTURE_COORDS, based on the texture matrix and the texture generation functions (see glTexGen). Finally, the distance from the origin of the eye coordinate system to the vertex, as transformed by only the modelview matrix, replaces

GL_CURRENT_RASTER_DISTANCE.

Initially, the current raster position is (0,0,0,1), the current raster distance is 0, the valid bit is set, the associated RGBA color is (1,1,1,1), the associated color index is 1, and the associated texture coordinates are (0, 0, 0, 1). In RGBA mode, GL_CURRENT_RASTER_INDEX is always 1; in color-index mode, the current raster RGBA color always maintains its initial value.

> **NOTE**
> The raster position is modified both by glRasterPos and by **glBitmap**.

> **NOTE**
> When the raster position coordinates are invalid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to the OpenGL state).

The following functions retrieve information related to glRasterPos:

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID
glGet with argument GL_CURRENT_RASTER_DISTANCE
glGet with argument GL_CURRENT_RASTER_COLOR
glGet with argument GL_CURRENT_RASTER_INDEX
glGet with argument GL_CURRENT_RASTER_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBe

g
i
n

g
l
B
i
t
m
a
p

g
l
C
o
p
y
P
i
x
e
l
s

g
l
D
r
a
w
P
i
x
e
l
s

g
l
E
n
d

g
l
L
i
g
h
t

g

lLightModel

glShadeModel

glTexCoord

glTexGen

glVertex

glRasterPos2sv function

2/22/2020 • 2 minutes to read • Edit Online

Specifies the raster position for pixel operations.

Syntax

```
void WINAPI glRasterPos2sv(
   const GLshort *v
);
```

Parameters

v	A pointer to an array of two elements, specifying x and y coordinates for the current raster position.

Return value

This function does not return a value.

Remarks

OpenGL maintains a 3-D position in window coordinates. This position, called the raster position, is maintained with subpixel accuracy. It is used to position pixel and bitmap write operations. See **glBitmap**, **glDrawPixels**, and **glCopyPixels**.

The current raster position consists of three window coordinates (*x, y, z*), a clip coordinate *w* value, an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The *w* coordinate is a clip coordinate, because *w* is not projected to window coordinates. The glRasterPos4 function specifies object coordinates *x, y, z*, and *w* explicitly. The glRasterPos3 function specifies object coordinates *x, y,* and *z* explicitly, while *w* is implicitly set to one. The glRasterPos2 function uses the argument values for *x* and *y* while implicitly setting *z* and *w* to zero and one.

The object coordinates presented by glRasterPos are treated just like those of a glVertex command. They are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, then it is projected and scaled to window coordinates, which become the new current raster position, and the GL_CURRENT_RASTER_POSITION_VALID flag is set. If the vertex is culled, then the valid bit is cleared and the current raster position and associated color and texture coordinates are undefined.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then GL_CURRENT_RASTER_COLOR, in RGBA mode, or the GL_CURRENT_RASTER_INDEX, in color-index mode, is set to the color produced by the lighting calculation (see glLight, glLightModel, and **glShadeModel**). If lighting is disabled, current color (in RGBA mode, state variable GL_CURRENT_COLOR) or color index (in color-index mode, state variable GL_CURRENT_INDEX) is used to update the current raster color.

Likewise, GL_CURRENT_RASTER_TEXTURE_COORDS is updated as a function of GL_CURRENT_TEXTURE_COORDS, based on the texture matrix and the texture generation functions (see glTexGen). Finally, the distance from the origin of the eye coordinate system to the vertex, as transformed by only the modelview matrix, replaces GL_CURRENT_RASTER_DISTANCE.

Initially, the current raster position is (0,0,0,1), the current raster distance is 0, the valid bit is set, the associated

RGBA color is (1,1,1,1), the associated color index is 1, and the associated texture coordinates are (0, 0, 0, 1). In RGBA mode, GL_CURRENT_RASTER_INDEX is always 1; in color-index mode, the current raster RGBA color always maintains its initial value.

> **NOTE**
> The raster position is modified both by glRasterPos and by **glBitmap**.

> **NOTE**
> When the raster position coordinates are invalid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to the OpenGL state).

The following functions retrieve information related to glRasterPos:

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID
glGet with argument GL_CURRENT_RASTER_DISTANCE
glGet with argument GL_CURRENT_RASTER_COLOR
glGet with argument GL_CURRENT_RASTER_INDEX
glGet with argument GL_CURRENT_RASTER_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

glBitmap

glCopyPixels

glDrawPixels

glEnd

glLight

glL

i
g
h
t
M
o
d
e
l

g
l
S
h
a
d
e
M
o
d
e
l

g
l
T
e
x
C
o
o
r
d

g
l
T
e
x
G
e
n

g
l
V
e
r
t
e
x

glRasterPos3d function

2/22/2020 • 2 minutes to read • Edit Online

Specifies the raster position for pixel operations.

Syntax

```
void WINAPI glRasterPos3d(
   GLdouble x,
   GLdouble y,
   GLdouble z
);
```

Parameters

x	Specifies the x-coordinate for the current raster position.
y	Specifies the y-coordinate for the current raster position.
z	Specifies the z-coordinate for the current raster position.

Return value

This function does not return a value.

Remarks

OpenGL maintains a 3-D position in window coordinates. This position, called the raster position, is maintained with subpixel accuracy. It is used to position pixel and bitmap write operations. See **glBitmap**, **glDrawPixels**, and **glCopyPixels**.

The current raster position consists of three window coordinates (*x, y, z*), a clip coordinate *w* value, an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The *w* coordinate is a clip coordinate, because *w* is not projected to window coordinates. The glRasterPos4 function specifies object coordinates *x, y, z*, and *w* explicitly. The glRasterPos3 function specifies object coordinates *x, y,* and *z* explicitly, while *w* is implicitly set to one. The glRasterPos2 function uses the argument values for *x* and *y* while implicitly setting *z* and *w* to zero and one.

The object coordinates presented by glRasterPos are treated just like those of a glVertex command. They are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, then it is projected and scaled to window coordinates, which become the new current raster position, and the GL_CURRENT_RASTER_POSITION_VALID flag is set. If the vertex is culled, then the valid bit is cleared and the current raster position and associated color and texture coordinates are undefined.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then GL_CURRENT_RASTER_COLOR, in RGBA mode, or the GL_CURRENT_RASTER_INDEX, in color-index mode, is set to the color produced by the lighting calculation (see glLight, glLightModel, and **glShadeModel**). If lighting is disabled, current color (in RGBA mode, state variable GL_CURRENT_COLOR) or color index (in color-index mode, state variable GL_CURRENT_INDEX) is used to update the current raster color.

Likewise, GL_CURRENT_RASTER_TEXTURE_COORDS is updated as a function of GL_CURRENT_TEXTURE_COORDS, based on the texture matrix and the texture generation functions (see glTexGen). Finally, the distance from the origin of the eye coordinate system to the vertex, as transformed by only the modelview matrix, replaces GL_CURRENT_RASTER_DISTANCE.

Initially, the current raster position is (0,0,0,1), the current raster distance is 0, the valid bit is set, the associated RGBA color is (1,1,1,1), the associated color index is 1, and the associated texture coordinates are (0, 0, 0, 1). In RGBA mode, GL_CURRENT_RASTER_INDEX is always 1; in color-index mode, the current raster RGBA color always maintains its initial value.

> **NOTE**
> The raster position is modified both by glRasterPos and by **glBitmap**.

> **NOTE**
> When the raster position coordinates are invalid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to the OpenGL state).

The following functions retrieve information related to glRasterPos:

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID
glGet with argument GL_CURRENT_RASTER_DISTANCE
glGet with argument GL_CURRENT_RASTER_COLOR
glGet with argument GL_CURRENT_RASTER_INDEX
glGet with argument GL_CURRENT_RASTER_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

g

glBegin

glBitmap

glCopyPixels

glDrawPixels

glEnd

glLig

h
t

g
l
L
i
g
h
t
M
o
d
e
l

g
l
S
h
a
d
e
M
o
d
e
l

g
l
T
e
x
C
o
o
r
d

g
l
T
e
x
G
e
n

g
l
V
e

r
t
e
x

glRasterPos3dv function

2/22/2020 • 2 minutes to read • Edit Online

Specifies the raster position for pixel operations.

Syntax

```
void WINAPI glRasterPos3dv(
   const GLdouble *v
);
```

Parameters

v	A pointer to an array of three elements, specifying x, y, and z coordinates for the current raster position.

Return value

This function does not return a value.

Remarks

OpenGL maintains a 3-D position in window coordinates. This position, called the raster position, is maintained with subpixel accuracy. It is used to position pixel and bitmap write operations. See **glBitmap**, **glDrawPixels**, and **glCopyPixels**.

The current raster position consists of three window coordinates (*x, y, z*), a clip coordinate *w* value, an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The *w* coordinate is a clip coordinate, because *w* is not projected to window coordinates. The glRasterPos4 function specifies object coordinates *x, y, z*, and *w* explicitly. The glRasterPos3 function specifies object coordinates *x, y,* and *z* explicitly, while *w* is implicitly set to one. The glRasterPos2 function uses the argument values for *x* and *y* while implicitly setting *z* and *w* to zero and one.

The object coordinates presented by glRasterPos are treated just like those of a glVertex command. They are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, then it is projected and scaled to window coordinates, which become the new current raster position, and the GL_CURRENT_RASTER_POSITION_VALID flag is set. If the vertex is culled, then the valid bit is cleared and the current raster position and associated color and texture coordinates are undefined.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then GL_CURRENT_RASTER_COLOR, in RGBA mode, or the GL_CURRENT_RASTER_INDEX, in color-index mode, is set to the color produced by the lighting calculation (see glLight, glLightModel, and **glShadeModel**). If lighting is disabled, current color (in RGBA mode, state variable GL_CURRENT_COLOR) or color index (in color-index mode, state variable GL_CURRENT_INDEX) is used to update the current raster color.

Likewise, GL_CURRENT_RASTER_TEXTURE_COORDS is updated as a function of GL_CURRENT_TEXTURE_COORDS, based on the texture matrix and the texture generation functions (see glTexGen). Finally, the distance from the origin of the eye coordinate system to the vertex, as transformed by only the modelview matrix, replaces GL_CURRENT_RASTER_DISTANCE.

Initially, the current raster position is (0,0,0,1), the current raster distance is 0, the valid bit is set, the associated

RGBA color is (1,1,1,1), the associated color index is 1, and the associated texture coordinates are (0, 0, 0, 1). In RGBA mode, GL_CURRENT_RASTER_INDEX is always 1; in color-index mode, the current raster RGBA color always maintains its initial value.

> **NOTE**
> The raster position is modified both by glRasterPos and by **glBitmap**.

> **NOTE**
> When the raster position coordinates are invalid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to the OpenGL state).

The following functions retrieve information related to glRasterPos:

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID
glGet with argument GL_CURRENT_RASTER_DISTANCE
glGet with argument GL_CURRENT_RASTER_COLOR
glGet with argument GL_CURRENT_RASTER_INDEX
glGet with argument GL_CURRENT_RASTER_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

glBitmap

glCopyPixels

glDrawPixels

glEnd

glLight

glL

i
g
h
t
M
o
d
e
l

g
l
S
h
a
d
e
M
o
d
e
l

g
l
T
e
x
C
o
o
r
d

g
l
T
e
x
G
e
n

g
l
V
e
r
t
e
x

glRasterPos3f function

2/22/2020 • 2 minutes to read • Edit Online

Specifies the raster position for pixel operations.

Syntax

```
void WINAPI glRasterPos3f(
   GLfloat x,
   GLfloat y,
   GLfloat z
);
```

Parameters

x	Specifies the x-coordinate for the current raster position.
y	Specifies the y-coordinate for the current raster position.
z	Specifies the z-coordinate for the current raster position.

Return value

This function does not return a value.

Remarks

OpenGL maintains a 3-D position in window coordinates. This position, called the raster position, is maintained with subpixel accuracy. It is used to position pixel and bitmap write operations. See **glBitmap**, **glDrawPixels**, and **glCopyPixels**.

The current raster position consists of three window coordinates (*x, y, z*), a clip coordinate *w* value, an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The *w* coordinate is a clip coordinate, because *w* is not projected to window coordinates. The glRasterPos4 function specifies object coordinates *x, y, z,* and *w* explicitly. The glRasterPos3 function specifies object coordinates *x, y,* and *z* explicitly, while *w* is implicitly set to one. The glRasterPos2 function uses the argument values for *x* and *y* while implicitly setting *z* and *w* to zero and one.

The object coordinates presented by glRasterPos are treated just like those of a glVertex command. They are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, then it is projected and scaled to window coordinates, which become the new current raster position, and the GL_CURRENT_RASTER_POSITION_VALID flag is set. If the vertex is culled, then the valid bit is cleared and the current raster position and associated color and texture coordinates are undefined.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then GL_CURRENT_RASTER_COLOR, in RGBA mode, or the GL_CURRENT_RASTER_INDEX, in color-index mode, is set to the color produced by the lighting calculation (see glLight, glLightModel, and **glShadeModel**). If lighting is disabled, current color (in RGBA mode, state variable GL_CURRENT_COLOR) or color index (in color-index mode, state variable GL_CURRENT_INDEX) is used to update the current raster color.

Likewise, GL_CURRENT_RASTER_TEXTURE_COORDS is updated as a function of GL_CURRENT_TEXTURE_COORDS, based on the texture matrix and the texture generation functions (see glTexGen). Finally, the distance from the origin of the eye coordinate system to the vertex, as transformed by only the modelview matrix, replaces GL_CURRENT_RASTER_DISTANCE.

Initially, the current raster position is (0,0,0,1), the current raster distance is 0, the valid bit is set, the associated RGBA color is (1,1,1,1), the associated color index is 1, and the associated texture coordinates are (0, 0, 0, 1). In RGBA mode, GL_CURRENT_RASTER_INDEX is always 1; in color-index mode, the current raster RGBA color always maintains its initial value.

> **NOTE**
> The raster position is modified both by glRasterPos and by **glBitmap**.

> **NOTE**
> When the raster position coordinates are invalid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to the OpenGL state).

The following functions retrieve information related to glRasterPos:

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID
glGet with argument GL_CURRENT_RASTER_DISTANCE
glGet with argument GL_CURRENT_RASTER_COLOR
glGet with argument GL_CURRENT_RASTER_INDEX
glGet with argument GL_CURRENT_RASTER_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

g

glBegin

glBitmap

glCopyPixels

glDrawPixels

glEnd

glLig

h
t

g
l
L
i
g
h
t
M
o
d
e
l

g
l
S
h
a
d
e
M
o
d
e
l

g
l
T
e
x
C
o
o
r
d

g
l
T
e
x
G
e
n

g
l
V
e

r
t
e
x

glRasterPos3fv function

2/22/2020 • 2 minutes to read • Edit Online

Specifies the raster position for pixel operations.

Syntax

```
void WINAPI glRasterPos3fv(
   const GLfloat *v
);
```

Parameters

v	A pointer to an array of three elements, specifying x, y, and z coordinates for the current raster position.

Return value

This function does not return a value.

Remarks

OpenGL maintains a 3-D position in window coordinates. This position, called the raster position, is maintained with subpixel accuracy. It is used to position pixel and bitmap write operations. See **glBitmap**, **glDrawPixels**, and **glCopyPixels**.

The current raster position consists of three window coordinates (*x, y, z*), a clip coordinate *w* value, an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The *w* coordinate is a clip coordinate, because *w* is not projected to window coordinates. The glRasterPos4 function specifies object coordinates *x, y, z,* and *w* explicitly. The glRasterPos3 function specifies object coordinates *x, y,* and *z* explicitly, while *w* is implicitly set to one. The glRasterPos2 function uses the argument values for *x* and *y* while implicitly setting *z* and *w* to zero and one.

The object coordinates presented by glRasterPos are treated just like those of a glVertex command. They are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, then it is projected and scaled to window coordinates, which become the new current raster position, and the GL_CURRENT_RASTER_POSITION_VALID flag is set. If the vertex is culled, then the valid bit is cleared and the current raster position and associated color and texture coordinates are undefined.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then GL_CURRENT_RASTER_COLOR, in RGBA mode, or the GL_CURRENT_RASTER_INDEX, in color-index mode, is set to the color produced by the lighting calculation (see glLight, glLightModel, and **glShadeModel**). If lighting is disabled, current color (in RGBA mode, state variable GL_CURRENT_COLOR) or color index (in color-index mode, state variable GL_CURRENT_INDEX) is used to update the current raster color.

Likewise, GL_CURRENT_RASTER_TEXTURE_COORDS is updated as a function of GL_CURRENT_TEXTURE_COORDS, based on the texture matrix and the texture generation functions (see glTexGen). Finally, the distance from the origin of the eye coordinate system to the vertex, as transformed by only the modelview matrix, replaces GL_CURRENT_RASTER_DISTANCE.

Initially, the current raster position is (0,0,0,1), the current raster distance is 0, the valid bit is set, the associated

RGBA color is (1,1,1,1), the associated color index is 1, and the associated texture coordinates are (0, 0, 0, 1). In RGBA mode, GL_CURRENT_RASTER_INDEX is always 1; in color-index mode, the current raster RGBA color always maintains its initial value.

> **NOTE**
> The raster position is modified both by glRasterPos and by **glBitmap**.

> **NOTE**
> When the raster position coordinates are invalid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to the OpenGL state).

The following functions retrieve information related to glRasterPos:

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID
glGet with argument GL_CURRENT_RASTER_DISTANCE
glGet with argument GL_CURRENT_RASTER_COLOR
glGet with argument GL_CURRENT_RASTER_INDEX
glGet with argument GL_CURRENT_RASTER_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

glBitmap

glCopyPixels

glDrawPixels

glEnd

glLight

glL

i
g
h
t
M
o
d
e
l

g
l
S
h
a
d
e
M
o
d
e
l

g
l
T
e
x
C
o
o
r
d

g
l
T
e
x
G
e
n

g
l
V
e
r
t
e
x

glRasterPos3i function

2/22/2020 • 2 minutes to read • Edit Online

Specifies the raster position for pixel operations.

Syntax

```
void WINAPI glRasterPos3i(
   GLint x,
   GLint y,
   GLint z
);
```

Parameters

x	Specifies the x-coordinate for the current raster position.
y	Specifies the y-coordinate for the current raster position.
z	Specifies the z-coordinate for the current raster position.

Return value

This function does not return a value.

Remarks

OpenGL maintains a 3-D position in window coordinates. This position, called the raster position, is maintained with subpixel accuracy. It is used to position pixel and bitmap write operations. See **glBitmap**, **glDrawPixels**, and **glCopyPixels**.

The current raster position consists of three window coordinates (*x, y, z*), a clip coordinate *w* value, an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The *w* coordinate is a clip coordinate, because *w* is not projected to window coordinates. The glRasterPos4 function specifies object coordinates *x, y, z,* and *w* explicitly. The glRasterPos3 function specifies object coordinates *x, y,* and *z* explicitly, while *w* is implicitly set to one. The glRasterPos2 function uses the argument values for *x* and *y* while implicitly setting *z* and *w* to zero and one.

The object coordinates presented by glRasterPos are treated just like those of a glVertex command. They are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, then it is projected and scaled to window coordinates, which become the new current raster position, and the GL_CURRENT_RASTER_POSITION_VALID flag is set. If the vertex is culled, then the valid bit is cleared and the current raster position and associated color and texture coordinates are undefined.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then GL_CURRENT_RASTER_COLOR, in RGBA mode, or the GL_CURRENT_RASTER_INDEX, in color-index mode, is set to the color produced by the lighting calculation (see glLight, glLightModel, and **glShadeModel**). If lighting is disabled, current color (in RGBA mode, state variable GL_CURRENT_COLOR) or color index (in color-index mode, state variable GL_CURRENT_INDEX) is used to update the current raster color.

Likewise, GL_CURRENT_RASTER_TEXTURE_COORDS is updated as a function of GL_CURRENT_TEXTURE_COORDS, based on the texture matrix and the texture generation functions (see glTexGen). Finally, the distance from the origin of the eye coordinate system to the vertex, as transformed by only the modelview matrix, replaces GL_CURRENT_RASTER_DISTANCE.

Initially, the current raster position is (0,0,0,1), the current raster distance is 0, the valid bit is set, the associated RGBA color is (1,1,1,1), the associated color index is 1, and the associated texture coordinates are (0, 0, 0, 1). In RGBA mode, GL_CURRENT_RASTER_INDEX is always 1; in color-index mode, the current raster RGBA color always maintains its initial value.

> **NOTE**
> The raster position is modified both by glRasterPos and by **glBitmap**.

> **NOTE**
> When the raster position coordinates are invalid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to the OpenGL state).

The following functions retrieve information related to glRasterPos:

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID
glGet with argument GL_CURRENT_RASTER_DISTANCE
glGet with argument GL_CURRENT_RASTER_COLOR
glGet with argument GL_CURRENT_RASTER_INDEX
glGet with argument GL_CURRENT_RASTER_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

g

glBegin

glBitmap

glCopyPixels

glDrawPixels

glEnd

glLig

h
t

g
l
L
i
g
h
t
M
o
d
e
l

g
l
S
h
a
d
e
M
o
d
e
l

g
l
T
e
x
C
o
o
r
d

g
l
T
e
x
G
e
n

g
l
V
e

r
t
e
x

glRasterPos3iv function

2/22/2020 • 2 minutes to read • Edit Online

Specifies the raster position for pixel operations.

Syntax

```
void WINAPI glRasterPos3iv(
   const GLint *v
);
```

Parameters

v	A pointer to an array of three elements, specifying x, y, and z coordinates for the current raster position.

Return value

This function does not return a value.

Remarks

OpenGL maintains a 3-D position in window coordinates. This position, called the raster position, is maintained with subpixel accuracy. It is used to position pixel and bitmap write operations. See **glBitmap**, **glDrawPixels**, and **glCopyPixels**.

The current raster position consists of three window coordinates (*x, y, z*), a clip coordinate *w* value, an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The *w* coordinate is a clip coordinate, because *w* is not projected to window coordinates. The glRasterPos4 function specifies object coordinates *x, y, z*, and *w* explicitly. The glRasterPos3 function specifies object coordinates *x, y,* and *z* explicitly, while *w* is implicitly set to one. The glRasterPos2 function uses the argument values for *x* and *y* while implicitly setting *z* and *w* to zero and one.

The object coordinates presented by glRasterPos are treated just like those of a glVertex command. They are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, then it is projected and scaled to window coordinates, which become the new current raster position, and the GL_CURRENT_RASTER_POSITION_VALID flag is set. If the vertex is culled, then the valid bit is cleared and the current raster position and associated color and texture coordinates are undefined.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then GL_CURRENT_RASTER_COLOR, in RGBA mode, or the GL_CURRENT_RASTER_INDEX, in color-index mode, is set to the color produced by the lighting calculation (see glLight, glLightModel, and **glShadeModel**). If lighting is disabled, current color (in RGBA mode, state variable GL_CURRENT_COLOR) or color index (in color-index mode, state variable GL_CURRENT_INDEX) is used to update the current raster color.

Likewise, GL_CURRENT_RASTER_TEXTURE_COORDS is updated as a function of GL_CURRENT_TEXTURE_COORDS, based on the texture matrix and the texture generation functions (see glTexGen). Finally, the distance from the origin of the eye coordinate system to the vertex, as transformed by only the modelview matrix, replaces GL_CURRENT_RASTER_DISTANCE.

Initially, the current raster position is (0,0,0,1), the current raster distance is 0, the valid bit is set, the associated

RGBA color is (1,1,1,1), the associated color index is 1, and the associated texture coordinates are (0, 0, 0, 1). In RGBA mode, GL_CURRENT_RASTER_INDEX is always 1; in color-index mode, the current raster RGBA color always maintains its initial value.

> **NOTE**
> The raster position is modified both by glRasterPos and by **glBitmap**.

> **NOTE**
> When the raster position coordinates are invalid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to the OpenGL state).

The following functions retrieve information related to glRasterPos:

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID
glGet with argument GL_CURRENT_RASTER_DISTANCE
glGet with argument GL_CURRENT_RASTER_COLOR
glGet with argument GL_CURRENT_RASTER_INDEX
glGet with argument GL_CURRENT_RASTER_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

glBitmap

glCopyPixels

glDrawPixels

glEnd

glLight

glL

i
g
h
t
M
o
d
e
l

g
l
S
h
a
d
e
M
o
d
e
l

g
l
T
e
x
C
o
o
r
d

g
l
T
e
x
G
e
n

g
l
V
e
r
t
e
x

glRasterPos3s function

2/22/2020 • 2 minutes to read • Edit Online

Specifies the raster position for pixel operations.

Syntax

```
void WINAPI glRasterPos3s(
   GLshort x,
   GLshort y,
   GLshort z
);
```

Parameters

x	Specifies the x-coordinate for the current raster position.
y	Specifies the y-coordinate for the current raster position.
z	Specifies the z-coordinate for the current raster position.

Return value

This function does not return a value.

Remarks

OpenGL maintains a 3-D position in window coordinates. This position, called the raster position, is maintained with subpixel accuracy. It is used to position pixel and bitmap write operations. See **glBitmap**, **glDrawPixels**, and **glCopyPixels**.

The current raster position consists of three window coordinates (*x, y, z*), a clip coordinate *w* value, an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The *w* coordinate is a clip coordinate, because *w* is not projected to window coordinates. The glRasterPos4 function specifies object coordinates *x, y, z,* and *w* explicitly. The glRasterPos3 function specifies object coordinates *x, y,* and *z* explicitly, while *w* is implicitly set to one. The glRasterPos2 function uses the argument values for *x* and *y* while implicitly setting *z* and *w* to zero and one.

The object coordinates presented by glRasterPos are treated just like those of a glVertex command. They are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, then it is projected and scaled to window coordinates, which become the new current raster position, and the GL_CURRENT_RASTER_POSITION_VALID flag is set. If the vertex is culled, then the valid bit is cleared and the current raster position and associated color and texture coordinates are undefined.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then GL_CURRENT_RASTER_COLOR, in RGBA mode, or the GL_CURRENT_RASTER_INDEX, in color-index mode, is set to the color produced by the lighting calculation (see glLight, glLightModel, and **glShadeModel**). If lighting is disabled, current color (in RGBA mode, state variable GL_CURRENT_COLOR) or color index (in color-index mode, state variable GL_CURRENT_INDEX) is used to update the current raster color.

Likewise, GL_CURRENT_RASTER_TEXTURE_COORDS is updated as a function of GL_CURRENT_TEXTURE_COORDS, based on the texture matrix and the texture generation functions (see glTexGen). Finally, the distance from the origin of the eye coordinate system to the vertex, as transformed by only the modelview matrix, replaces GL_CURRENT_RASTER_DISTANCE.

Initially, the current raster position is (0,0,0,1), the current raster distance is 0, the valid bit is set, the associated RGBA color is (1,1,1,1), the associated color index is 1, and the associated texture coordinates are (0, 0, 0, 1). In RGBA mode, GL_CURRENT_RASTER_INDEX is always 1; in color-index mode, the current raster RGBA color always maintains its initial value.

> **NOTE**
> The raster position is modified both by glRasterPos and by **glBitmap**.

> **NOTE**
> When the raster position coordinates are invalid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to the OpenGL state).

The following functions retrieve information related to glRasterPos:

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID
glGet with argument GL_CURRENT_RASTER_DISTANCE
glGet with argument GL_CURRENT_RASTER_COLOR
glGet with argument GL_CURRENT_RASTER_INDEX
glGet with argument GL_CURRENT_RASTER_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

g

glBegin

glBitmap

glCopyPixels

glDrawPixels

glEnd

glLig

h
t

g
l
L
i
g
h
t
M
o
d
e
l

g
l
S
h
a
d
e
M
o
d
e
l

g
l
T
e
x
C
o
o
r
d

g
l
T
e
x
G
e
n

g
l
V
e

r
t
e
x

glRasterPos3sv function

2/22/2020 • 2 minutes to read • Edit Online

Specifies the raster position for pixel operations.

Syntax

```
void WINAPI glRasterPos3sv(
   const GLshort *v
);
```

Parameters

v	A pointer to an array of three elements, specifying x, y, and z coordinates for the current raster position.

Return value

This function does not return a value.

Remarks

OpenGL maintains a 3-D position in window coordinates. This position, called the raster position, is maintained with subpixel accuracy. It is used to position pixel and bitmap write operations. See **glBitmap**, **glDrawPixels**, and **glCopyPixels**.

The current raster position consists of three window coordinates (*x, y, z*), a clip coordinate *w* value, an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The *w* coordinate is a clip coordinate, because *w* is not projected to window coordinates. The glRasterPos4 function specifies object coordinates *x, y, z*, and *w* explicitly. The glRasterPos3 function specifies object coordinates *x, y,* and *z* explicitly, while *w* is implicitly set to one. The glRasterPos2 function uses the argument values for *x* and *y* while implicitly setting *z* and *w* to zero and one.

The object coordinates presented by glRasterPos are treated just like those of a glVertex command. They are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, then it is projected and scaled to window coordinates, which become the new current raster position, and the GL_CURRENT_RASTER_POSITION_VALID flag is set. If the vertex is culled, then the valid bit is cleared and the current raster position and associated color and texture coordinates are undefined.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then GL_CURRENT_RASTER_COLOR, in RGBA mode, or the GL_CURRENT_RASTER_INDEX, in color-index mode, is set to the color produced by the lighting calculation (see glLight, glLightModel, and **glShadeModel**). If lighting is disabled, current color (in RGBA mode, state variable GL_CURRENT_COLOR) or color index (in color-index mode, state variable GL_CURRENT_INDEX) is used to update the current raster color.

Likewise, GL_CURRENT_RASTER_TEXTURE_COORDS is updated as a function of GL_CURRENT_TEXTURE_COORDS, based on the texture matrix and the texture generation functions (see glTexGen). Finally, the distance from the origin of the eye coordinate system to the vertex, as transformed by only the modelview matrix, replaces GL_CURRENT_RASTER_DISTANCE.

Initially, the current raster position is (0,0,0,1), the current raster distance is 0, the valid bit is set, the associated

RGBA color is (1,1,1,1), the associated color index is 1, and the associated texture coordinates are (0, 0, 0, 1). In RGBA mode, GL_CURRENT_RASTER_INDEX is always 1; in color-index mode, the current raster RGBA color always maintains its initial value.

> **NOTE**
> The raster position is modified both by glRasterPos and by **glBitmap**.

> **NOTE**
> When the raster position coordinates are invalid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to the OpenGL state).

The following functions retrieve information related to glRasterPos:

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID
glGet with argument GL_CURRENT_RASTER_DISTANCE
glGet with argument GL_CURRENT_RASTER_COLOR
glGet with argument GL_CURRENT_RASTER_INDEX
glGet with argument GL_CURRENT_RASTER_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

glBitmap

glCopyPixels

glDrawPixels

glEnd

glLight

glL

i
g
h
t
M
o
d
e
l

g
l
S
h
a
d
e
M
o
d
e
l

g
l
T
e
x
C
o
o
r
d

g
l
T
e
x
G
e
n

g
l
V
e
r
t
e
x

glRasterPos4d function

2/22/2020 • 2 minutes to read • Edit Online

Specifies the raster position for pixel operations.

Syntax

```
void WINAPI glRasterPos4d(
   GLdouble x,
   GLdouble y,
   GLdouble z,
   GLdouble w
);
```

Parameters

x	Specifies the x-coordinate for the current raster position.
y	Specifies the y-coordinate for the current raster position.
z	Specifies the z-coordinate for the current raster position.
w	The w-coordinate for the current raster position.

Return value

This function does not return a value.

Remarks

OpenGL maintains a 3-D position in window coordinates. This position, called the raster position, is maintained with subpixel accuracy. It is used to position pixel and bitmap write operations. See **glBitmap**, **glDrawPixels**, and **glCopyPixels**.

The current raster position consists of three window coordinates (*x, y, z*), a clip coordinate *w* value, an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The *w* coordinate is a clip coordinate, because *w* is not projected to window coordinates. The glRasterPos4 function specifies object coordinates *x, y, z,* and *w* explicitly. The glRasterPos3 function specifies object coordinates *x, y,* and *z* explicitly, while *w* is implicitly set to one. The glRasterPos2 function uses the argument values for *x* and *y* while implicitly setting *z* and *w* to zero and one.

The object coordinates presented by glRasterPos are treated just like those of a glVertex command. They are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, then it is projected and scaled to window coordinates, which become the new current raster position, and the GL_CURRENT_RASTER_POSITION_VALID flag is set. If the vertex is culled, then the valid bit is cleared and the current raster position and associated color and texture coordinates are undefined.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then GL_CURRENT_RASTER_COLOR, in RGBA mode, or the GL_CURRENT_RASTER_INDEX, in color-index mode, is set to the color produced by the lighting calculation (see glLight, glLightModel, and **glShadeModel**). If lighting is

disabled, current color (in RGBA mode, state variable GL_CURRENT_COLOR) or color index (in color-index mode, state variable GL_CURRENT_INDEX) is used to update the current raster color.

Likewise, GL_CURRENT_RASTER_TEXTURE_COORDS is updated as a function of GL_CURRENT_TEXTURE_COORDS, based on the texture matrix and the texture generation functions (see glTexGen). Finally, the distance from the origin of the eye coordinate system to the vertex, as transformed by only the modelview matrix, replaces GL_CURRENT_RASTER_DISTANCE.

Initially, the current raster position is (0,0,0,1), the current raster distance is 0, the valid bit is set, the associated RGBA color is (1,1,1,1), the associated color index is 1, and the associated texture coordinates are (0, 0, 0, 1). In RGBA mode, GL_CURRENT_RASTER_INDEX is always 1; in color-index mode, the current raster RGBA color always maintains its initial value.

> **NOTE**
> The raster position is modified both by glRasterPos and by **glBitmap**.

> **NOTE**
> When the raster position coordinates are invalid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to the OpenGL state).

The following functions retrieve information related to glRasterPos:

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID
glGet with argument GL_CURRENT_RASTER_DISTANCE
glGet with argument GL_CURRENT_RASTER_COLOR
glGet with argument GL_CURRENT_RASTER_INDEX
glGet with argument GL_CURRENT_RASTER_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glBitmap

glCopyPixels

glDrawPixels

glEnd

gl

L
i
g
h
t

g
l
L
i
g
h
t
M
o
d
e
l

g
l
S
h
a
d
e
M
o
d
e
l

g
l
T
e
x
C
o
o
r
d

g
l
T
e
x
G
e
n

g

l
V
e
r
t
e
x

glRasterPos4dv function

2/22/2020 • 2 minutes to read • Edit Online

Specifies the raster position for pixel operations.

Syntax

```
void WINAPI glRasterPos4dv(
   const GLdouble *v
);
```

Parameters

v	A pointer to an array of four elements, specifying x, y, z, and w coordinates for the current raster position.

Return value

This function does not return a value.

Remarks

OpenGL maintains a 3-D position in window coordinates. This position, called the raster position, is maintained with subpixel accuracy. It is used to position pixel and bitmap write operations. See **glBitmap**, **glDrawPixels**, and **glCopyPixels**.

The current raster position consists of three window coordinates (*x, y, z*), a clip coordinate *w* value, an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The *w* coordinate is a clip coordinate, because *w* is not projected to window coordinates. The glRasterPos4 function specifies object coordinates *x, y, z*, and *w* explicitly. The glRasterPos3 function specifies object coordinates *x, y,* and *z* explicitly, while *w* is implicitly set to one. The glRasterPos2 function uses the argument values for *x* and *y* while implicitly setting *z* and *w* to zero and one.

The object coordinates presented by glRasterPos are treated just like those of a glVertex command. They are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, then it is projected and scaled to window coordinates, which become the new current raster position, and the GL_CURRENT_RASTER_POSITION_VALID flag is set. If the vertex is culled, then the valid bit is cleared and the current raster position and associated color and texture coordinates are undefined.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then GL_CURRENT_RASTER_COLOR, in RGBA mode, or the GL_CURRENT_RASTER_INDEX, in color-index mode, is set to the color produced by the lighting calculation (see glLight, glLightModel, and **glShadeModel**). If lighting is disabled, current color (in RGBA mode, state variable GL_CURRENT_COLOR) or color index (in color-index mode, state variable GL_CURRENT_INDEX) is used to update the current raster color.

Likewise, GL_CURRENT_RASTER_TEXTURE_COORDS is updated as a function of GL_CURRENT_TEXTURE_COORDS, based on the texture matrix and the texture generation functions (see glTexGen). Finally, the distance from the origin of the eye coordinate system to the vertex, as transformed by only the modelview matrix, replaces GL_CURRENT_RASTER_DISTANCE.

Initially, the current raster position is (0,0,0,1), the current raster distance is 0, the valid bit is set, the associated

RGBA color is (1,1,1,1), the associated color index is 1, and the associated texture coordinates are (0, 0, 0, 1). In RGBA mode, GL_CURRENT_RASTER_INDEX is always 1; in color-index mode, the current raster RGBA color always maintains its initial value.

> **NOTE**
> The raster position is modified both by glRasterPos and by **glBitmap**.

> **NOTE**
> When the raster position coordinates are invalid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to the OpenGL state).

The following functions retrieve information related to glRasterPos:

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID
glGet with argument GL_CURRENT_RASTER_DISTANCE
glGet with argument GL_CURRENT_RASTER_COLOR
glGet with argument GL_CURRENT_RASTER_INDEX
glGet with argument GL_CURRENT_RASTER_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

glBitmap

glCopyPixels

glDrawPixels

glEnd

glLight

glL

i
g
h
t
M
o
d
e
l

g
l
S
h
a
d
e
M
o
d
e
l

g
l
T
e
x
C
o
o
r
d

g
l
T
e
x
G
e
n

g
l
V
e
r
t
e
x

glRasterPos4f function

2/22/2020 • 2 minutes to read • Edit Online

Specifies the raster position for pixel operations.

Syntax

```
void WINAPI glRasterPos4f(
   GLfloat x,
   GLfloat y,
   GLfloat z,
   GLfloat w
);
```

Parameters

x	Specifies the x-coordinate for the current raster position.
y	Specifies the y-coordinate for the current raster position.
z	Specifies the z-coordinate for the current raster position.
w	The w-coordinate for the current raster position.

Return value

This function does not return a value.

Remarks

OpenGL maintains a 3-D position in window coordinates. This position, called the raster position, is maintained with subpixel accuracy. It is used to position pixel and bitmap write operations. See **glBitmap**, **glDrawPixels**, and **glCopyPixels**.

The current raster position consists of three window coordinates (*x, y, z*), a clip coordinate *w* value, an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The *w* coordinate is a clip coordinate, because *w* is not projected to window coordinates. The glRasterPos4 function specifies object coordinates *x, y, z,* and *w* explicitly. The glRasterPos3 function specifies object coordinates *x, y,* and *z* explicitly, while *w* is implicitly set to one. The glRasterPos2 function uses the argument values for *x* and *y* while implicitly setting *z* and *w* to zero and one.

The object coordinates presented by glRasterPos are treated just like those of a glVertex command. They are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, then it is projected and scaled to window coordinates, which become the new current raster position, and the GL_CURRENT_RASTER_POSITION_VALID flag is set. If the vertex is culled, then the valid bit is cleared and the current raster position and associated color and texture coordinates are undefined.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then GL_CURRENT_RASTER_COLOR, in RGBA mode, or the GL_CURRENT_RASTER_INDEX, in color-index mode, is set to the color produced by the lighting calculation (see glLight, glLightModel, and **glShadeModel**). If lighting is

disabled, current color (in RGBA mode, state variable GL_CURRENT_COLOR) or color index (in color-index mode, state variable GL_CURRENT_INDEX) is used to update the current raster color.

Likewise, GL_CURRENT_RASTER_TEXTURE_COORDS is updated as a function of GL_CURRENT_TEXTURE_COORDS, based on the texture matrix and the texture generation functions (see glTexGen). Finally, the distance from the origin of the eye coordinate system to the vertex, as transformed by only the modelview matrix, replaces GL_CURRENT_RASTER_DISTANCE.

Initially, the current raster position is (0,0,0,1), the current raster distance is 0, the valid bit is set, the associated RGBA color is (1,1,1,1), the associated color index is 1, and the associated texture coordinates are (0, 0, 0, 1). In RGBA mode, GL_CURRENT_RASTER_INDEX is always 1; in color-index mode, the current raster RGBA color always maintains its initial value.

> **NOTE**
> The raster position is modified both by glRasterPos and by **glBitmap**.

> **NOTE**
> When the raster position coordinates are invalid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to the OpenGL state).

The following functions retrieve information related to glRasterPos:

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID
glGet with argument GL_CURRENT_RASTER_DISTANCE
glGet with argument GL_CURRENT_RASTER_COLOR
glGet with argument GL_CURRENT_RASTER_INDEX
glGet with argument GL_CURRENT_RASTER_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glBitmap

glCopyPixels

glDrawPixels

glEnd

gl

L
i
g
h
t

g
l
L
i
g
h
t
M
o
d
e
l

g
l
S
h
a
d
e
M
o
d
e
l

g
l
T
e
x
C
o
o
r
d

g
l
T
e
x
G
e
n

g

l
V
e
r
t
e
x

glRasterPos4fv function

2/22/2020 • 2 minutes to read • Edit Online

Specifies the raster position for pixel operations.

Syntax

```
void WINAPI glRasterPos4fv(
   const GLfloat *v
);
```

Parameters

v	A pointer to an array of four elements, specifying x, y, z, and w coordinates for the current raster position.

Return value

This function does not return a value.

Remarks

OpenGL maintains a 3-D position in window coordinates. This position, called the raster position, is maintained with subpixel accuracy. It is used to position pixel and bitmap write operations. See **glBitmap**, **glDrawPixels**, and **glCopyPixels**.

The current raster position consists of three window coordinates (*x, y, z*), a clip coordinate *w* value, an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The *w* coordinate is a clip coordinate, because *w* is not projected to window coordinates. The glRasterPos4 function specifies object coordinates *x, y, z*, and *w* explicitly. The glRasterPos3 function specifies object coordinates *x, y,* and *z* explicitly, while *w* is implicitly set to one. The glRasterPos2 function uses the argument values for *x* and *y* while implicitly setting *z* and *w* to zero and one.

The object coordinates presented by glRasterPos are treated just like those of a glVertex command. They are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, then it is projected and scaled to window coordinates, which become the new current raster position, and the GL_CURRENT_RASTER_POSITION_VALID flag is set. If the vertex is culled, then the valid bit is cleared and the current raster position and associated color and texture coordinates are undefined.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then GL_CURRENT_RASTER_COLOR, in RGBA mode, or the GL_CURRENT_RASTER_INDEX, in color-index mode, is set to the color produced by the lighting calculation (see glLight, glLightModel, and **glShadeModel**). If lighting is disabled, current color (in RGBA mode, state variable GL_CURRENT_COLOR) or color index (in color-index mode, state variable GL_CURRENT_INDEX) is used to update the current raster color.

Likewise, GL_CURRENT_RASTER_TEXTURE_COORDS is updated as a function of GL_CURRENT_TEXTURE_COORDS, based on the texture matrix and the texture generation functions (see glTexGen). Finally, the distance from the origin of the eye coordinate system to the vertex, as transformed by only the modelview matrix, replaces GL_CURRENT_RASTER_DISTANCE.

Initially, the current raster position is (0,0,0,1), the current raster distance is 0, the valid bit is set, the associated

RGBA color is (1,1,1,1), the associated color index is 1, and the associated texture coordinates are (0, 0, 0, 1). In RGBA mode, GL_CURRENT_RASTER_INDEX is always 1; in color-index mode, the current raster RGBA color always maintains its initial value.

> **NOTE**
> The raster position is modified both by glRasterPos and by **glBitmap**.

> **NOTE**
> When the raster position coordinates are invalid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to the OpenGL state).

The following functions retrieve information related to glRasterPos:

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID
glGet with argument GL_CURRENT_RASTER_DISTANCE
glGet with argument GL_CURRENT_RASTER_COLOR
glGet with argument GL_CURRENT_RASTER_INDEX
glGet with argument GL_CURRENT_RASTER_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

glBitmap

glCopyPixels

glDrawPixels

glEnd

glLight

glL

i
g
h
t
M
o
d
e
l

g
l
S
h
a
d
e
M
o
d
e
l

g
l
T
e
x
C
o
o
r
d

g
l
T
e
x
G
e
n

g
l
V
e
r
t
e
x

glRasterPos4i function

2/22/2020 • 2 minutes to read • Edit Online

Specifies the raster position for pixel operations.

Syntax

```
void WINAPI glRasterPos4i(
   GLint x,
   GLint y,
   GLint z,
   GLint w
);
```

Parameters

x	Specifies the x-coordinate for the current raster position.
y	Specifies the y-coordinate for the current raster position.
z	Specifies the z-coordinate for the current raster position.
w	The w-coordinate for the current raster position.

Return value

This function does not return a value.

Remarks

OpenGL maintains a 3-D position in window coordinates. This position, called the raster position, is maintained with subpixel accuracy. It is used to position pixel and bitmap write operations. See **glBitmap**, **glDrawPixels**, and **glCopyPixels**.

The current raster position consists of three window coordinates (*x, y, z*), a clip coordinate *w* value, an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The *w* coordinate is a clip coordinate, because *w* is not projected to window coordinates. The glRasterPos4 function specifies object coordinates *x, y, z,* and *w* explicitly. The glRasterPos3 function specifies object coordinates *x, y,* and *z* explicitly, while *w* is implicitly set to one. The glRasterPos2 function uses the argument values for *x* and *y* while implicitly setting *z* and *w* to zero and one.

The object coordinates presented by glRasterPos are treated just like those of a glVertex command. They are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, then it is projected and scaled to window coordinates, which become the new current raster position, and the GL_CURRENT_RASTER_POSITION_VALID flag is set. If the vertex is culled, then the valid bit is cleared and the current raster position and associated color and texture coordinates are undefined.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then GL_CURRENT_RASTER_COLOR, in RGBA mode, or the GL_CURRENT_RASTER_INDEX, in color-index mode, is set to the color produced by the lighting calculation (see glLight, glLightModel, and **glShadeModel**). If lighting is

disabled, current color (in RGBA mode, state variable GL_CURRENT_COLOR) or color index (in color-index mode, state variable GL_CURRENT_INDEX) is used to update the current raster color.

Likewise, GL_CURRENT_RASTER_TEXTURE_COORDS is updated as a function of GL_CURRENT_TEXTURE_COORDS, based on the texture matrix and the texture generation functions (see glTexGen). Finally, the distance from the origin of the eye coordinate system to the vertex, as transformed by only the modelview matrix, replaces GL_CURRENT_RASTER_DISTANCE.

Initially, the current raster position is (0,0,0,1), the current raster distance is 0, the valid bit is set, the associated RGBA color is (1,1,1,1), the associated color index is 1, and the associated texture coordinates are (0, 0, 0, 1). In RGBA mode, GL_CURRENT_RASTER_INDEX is always 1; in color-index mode, the current raster RGBA color always maintains its initial value.

> **NOTE**
> The raster position is modified both by glRasterPos and by **glBitmap**.

> **NOTE**
> When the raster position coordinates are invalid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to the OpenGL state).

The following functions retrieve information related to glRasterPos:

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID
glGet with argument GL_CURRENT_RASTER_DISTANCE
glGet with argument GL_CURRENT_RASTER_COLOR
glGet with argument GL_CURRENT_RASTER_INDEX
glGet with argument GL_CURRENT_RASTER_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glBitmap

glCopyPixels

glDrawPixels

glEnd

gl

L
i
g
h
t

g
l
L
i
g
h
t
M
o
d
e
l

g
l
S
h
a
d
e
M
o
d
e
l

g
l
T
e
x
C
o
o
r
d

g
l
T
e
x
G
e
n

g

l
V
e
r
t
e
x

glRasterPos4iv function

2/22/2020 • 2 minutes to read • Edit Online

Specifies the raster position for pixel operations.

Syntax

```
void WINAPI glRasterPos4iv(
   const GLint *v
);
```

Parameters

v	A pointer to an array of four elements, specifying x, y, z, and w coordinates for the current raster position.

Return value

This function does not return a value.

Remarks

OpenGL maintains a 3-D position in window coordinates. This position, called the raster position, is maintained with subpixel accuracy. It is used to position pixel and bitmap write operations. See **glBitmap**, **glDrawPixels**, and **glCopyPixels**.

The current raster position consists of three window coordinates (*x, y, z*), a clip coordinate *w* value, an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The *w* coordinate is a clip coordinate, because *w* is not projected to window coordinates. The glRasterPos4 function specifies object coordinates *x, y, z,* and *w* explicitly. The glRasterPos3 function specifies object coordinates *x, y,* and *z* explicitly, while *w* is implicitly set to one. The glRasterPos2 function uses the argument values for *x* and *y* while implicitly setting *z* and *w* to zero and one.

The object coordinates presented by glRasterPos are treated just like those of a glVertex command. They are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, then it is projected and scaled to window coordinates, which become the new current raster position, and the GL_CURRENT_RASTER_POSITION_VALID flag is set. If the vertex is culled, then the valid bit is cleared and the current raster position and associated color and texture coordinates are undefined.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then GL_CURRENT_RASTER_COLOR, in RGBA mode, or the GL_CURRENT_RASTER_INDEX, in color-index mode, is set to the color produced by the lighting calculation (see glLight, glLightModel, and **glShadeModel**). If lighting is disabled, current color (in RGBA mode, state variable GL_CURRENT_COLOR) or color index (in color-index mode, state variable GL_CURRENT_INDEX) is used to update the current raster color.

Likewise, GL_CURRENT_RASTER_TEXTURE_COORDS is updated as a function of GL_CURRENT_TEXTURE_COORDS, based on the texture matrix and the texture generation functions (see glTexGen). Finally, the distance from the origin of the eye coordinate system to the vertex, as transformed by only the modelview matrix, replaces GL_CURRENT_RASTER_DISTANCE.

Initially, the current raster position is (0,0,0,1), the current raster distance is 0, the valid bit is set, the associated

RGBA color is (1,1,1,1), the associated color index is 1, and the associated texture coordinates are (0, 0, 0, 1). In RGBA mode, GL_CURRENT_RASTER_INDEX is always 1; in color-index mode, the current raster RGBA color always maintains its initial value.

> **NOTE**
> The raster position is modified both by glRasterPos and by **glBitmap**.

> **NOTE**
> When the raster position coordinates are invalid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to the OpenGL state).

The following functions retrieve information related to glRasterPos:

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID
glGet with argument GL_CURRENT_RASTER_DISTANCE
glGet with argument GL_CURRENT_RASTER_COLOR
glGet with argument GL_CURRENT_RASTER_INDEX
glGet with argument GL_CURRENT_RASTER_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

glBitmap

glCopyPixels

glDrawPixels

glEnd

glLight

glL

i
g
h
t
M
o
d
e
l

g
l
S
h
a
d
e
M
o
d
e
l

g
l
T
e
x
C
o
o
r
d

g
l
T
e
x
G
e
n

g
l
V
e
r
t
e
x

glRasterPos4s function

2/22/2020 • 2 minutes to read • Edit Online

Specifies the raster position for pixel operations.

Syntax

```
void WINAPI glRasterPos4s(
   GLshort x,
   GLshort y,
   GLshort z,
   GLshort w
);
```

Parameters

x	Specifies the x-coordinate for the current raster position.
y	Specifies the y-coordinate for the current raster position.
z	Specifies the z-coordinate for the current raster position.
w	The w-coordinate for the current raster position.

Return value

This function does not return a value.

Remarks

OpenGL maintains a 3-D position in window coordinates. This position, called the raster position, is maintained with subpixel accuracy. It is used to position pixel and bitmap write operations. See **glBitmap**, **glDrawPixels**, and **glCopyPixels**.

The current raster position consists of three window coordinates (*x, y, z*), a clip coordinate *w* value, an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The *w* coordinate is a clip coordinate, because *w* is not projected to window coordinates. The glRasterPos4 function specifies object coordinates *x, y, z,* and *w* explicitly. The glRasterPos3 function specifies object coordinates *x, y,* and *z* explicitly, while *w* is implicitly set to one. The glRasterPos2 function uses the argument values for *x* and *y* while implicitly setting *z* and *w* to zero and one.

The object coordinates presented by glRasterPos are treated just like those of a glVertex command. They are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, then it is projected and scaled to window coordinates, which become the new current raster position, and the GL_CURRENT_RASTER_POSITION_VALID flag is set. If the vertex is culled, then the valid bit is cleared and the current raster position and associated color and texture coordinates are undefined.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then GL_CURRENT_RASTER_COLOR, in RGBA mode, or the GL_CURRENT_RASTER_INDEX, in color-index mode, is set to the color produced by the lighting calculation (see glLight, glLightModel, and **glShadeModel**). If lighting is

disabled, current color (in RGBA mode, state variable GL_CURRENT_COLOR) or color index (in color-index mode, state variable GL_CURRENT_INDEX) is used to update the current raster color.

Likewise, GL_CURRENT_RASTER_TEXTURE_COORDS is updated as a function of GL_CURRENT_TEXTURE_COORDS, based on the texture matrix and the texture generation functions (see glTexGen). Finally, the distance from the origin of the eye coordinate system to the vertex, as transformed by only the modelview matrix, replaces GL_CURRENT_RASTER_DISTANCE.

Initially, the current raster position is (0,0,0,1), the current raster distance is 0, the valid bit is set, the associated RGBA color is (1,1,1,1), the associated color index is 1, and the associated texture coordinates are (0, 0, 0, 1). In RGBA mode, GL_CURRENT_RASTER_INDEX is always 1; in color-index mode, the current raster RGBA color always maintains its initial value.

> **NOTE**
> The raster position is modified both by glRasterPos and by **glBitmap**.

> **NOTE**
> When the raster position coordinates are invalid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to the OpenGL state).

The following functions retrieve information related to glRasterPos:

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID
glGet with argument GL_CURRENT_RASTER_DISTANCE
glGet with argument GL_CURRENT_RASTER_COLOR
glGet with argument GL_CURRENT_RASTER_INDEX
glGet with argument GL_CURRENT_RASTER_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glBitmap

glCopyPixels

glDrawPixels

glEnd

gl

L
i
g
h
t

g
l
L
i
g
h
t
M
o
d
e
l

g
l
S
h
a
d
e
M
o
d
e
l

g
l
T
e
x
C
o
o
r
d

g
l
T
e
x
G
e
n

g

l
V
e
r
t
e
x

glRasterPos4sv function

2/22/2020 • 2 minutes to read • Edit Online

Specifies the raster position for pixel operations.

Syntax

```
void WINAPI glRasterPos4sv(
   const GLshort *v
);
```

Parameters

v	A pointer to an array of four elements, specifying x, y, z, and w coordinates for the current raster position.

Return value

This function does not return a value.

Remarks

OpenGL maintains a 3-D position in window coordinates. This position, called the raster position, is maintained with subpixel accuracy. It is used to position pixel and bitmap write operations. See **glBitmap**, **glDrawPixels**, and **glCopyPixels**.

The current raster position consists of three window coordinates (*x, y, z*), a clip coordinate *w* value, an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The *w* coordinate is a clip coordinate, because *w* is not projected to window coordinates. The glRasterPos4 function specifies object coordinates *x, y, z*, and *w* explicitly. The glRasterPos3 function specifies object coordinates *x, y,* and *z* explicitly, while *w* is implicitly set to one. The glRasterPos2 function uses the argument values for *x* and *y* while implicitly setting *z* and *w* to zero and one.

The object coordinates presented by glRasterPos are treated just like those of a glVertex command. They are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, then it is projected and scaled to window coordinates, which become the new current raster position, and the GL_CURRENT_RASTER_POSITION_VALID flag is set. If the vertex is culled, then the valid bit is cleared and the current raster position and associated color and texture coordinates are undefined.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then GL_CURRENT_RASTER_COLOR, in RGBA mode, or the GL_CURRENT_RASTER_INDEX, in color-index mode, is set to the color produced by the lighting calculation (see glLight, glLightModel, and **glShadeModel**). If lighting is disabled, current color (in RGBA mode, state variable GL_CURRENT_COLOR) or color index (in color-index mode, state variable GL_CURRENT_INDEX) is used to update the current raster color.

Likewise, GL_CURRENT_RASTER_TEXTURE_COORDS is updated as a function of GL_CURRENT_TEXTURE_COORDS, based on the texture matrix and the texture generation functions (see glTexGen). Finally, the distance from the origin of the eye coordinate system to the vertex, as transformed by only the modelview matrix, replaces GL_CURRENT_RASTER_DISTANCE.

Initially, the current raster position is (0,0,0,1), the current raster distance is 0, the valid bit is set, the associated

RGBA color is (1,1,1,1), the associated color index is 1, and the associated texture coordinates are (0, 0, 0, 1). In RGBA mode, GL_CURRENT_RASTER_INDEX is always 1; in color-index mode, the current raster RGBA color always maintains its initial value.

> **NOTE**
> The raster position is modified both by glRasterPos and by **glBitmap**.

> **NOTE**
> When the raster position coordinates are invalid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to the OpenGL state).

The following functions retrieve information related to glRasterPos:

glGet with argument GL_CURRENT_RASTER_POSITION
glGet with argument GL_CURRENT_RASTER_POSITION_VALID
glGet with argument GL_CURRENT_RASTER_DISTANCE
glGet with argument GL_CURRENT_RASTER_COLOR
glGet with argument GL_CURRENT_RASTER_INDEX
glGet with argument GL_CURRENT_RASTER_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

glBitmap

glCopyPixels

glDrawPixels

glEnd

glLight

glL

i
g
h
t
M
o
d
e
l

g
l
S
h
a
d
e
M
o
d
e
l

g
l
T
e
x
C
o
o
r
d

g
l
T
e
x
G
e
n

g
l
V
e
r
t
e
x

glReadBuffer function

2/22/2020 • 2 minutes to read • Edit Online

The **glReadBuffer** function selects a color buffer source for pixels.

Syntax

```
void WINAPI glReadBuffer(
   GLenum mode
);
```

Parameters

m
o
d
e

A color buffer. Accepted values are GL_FRONT_LEFT, GL_FRONT_RIGHT, GL_BACK_LEFT, GL_BACK_RIGHT, GL_FRONT, GL_BACK, GL_LEFT, GL_RIGHT, and GL_AUX *i*, where *i* is between 0 and GL_AUX_BUFFERS 1.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*mode* was not an one of the twelve (or more) accepted values.
GL_INVALID_OPERATION	*mode* specified a buffer that does not exist.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*mode* was not an one of the twelve (or more) accepted values.
GL_INVALID_OPERATION	*mode* specified a buffer that does not exist.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glReadBuffer** function specifies a color buffer as the source for subsequent **glReadPixels** and **glCopyPixels** commands. The *mode* parameter accepts one of twelve or more predefined values. (GL_AUX0 through GL_AUX3 are always defined.) In a fully configured system, GL_FRONT, GL_LEFT, and GL_FRONT_LEFT all name the front-left buffer, GL_FRONT_RIGHT and GL_RIGHT name the front-right buffer, and GL_BACK_LEFT and GL_BACK name the back-left buffer.

Nonstereo double-buffered configurations have only a front-left and a back-left buffer. Single-buffered configurations have a front-left and a front-right buffer if stereo, and only a front-left buffer if nonstereo. It is an error to specify a nonexistent buffer to **glReadBuffer**.

By default, *mode* is GL_FRONT in single-buffered configurations, and GL_BACK in double-buffered configurations.

The following function retrieves information related to **glReadBuffer**:

glGet with argument GL_READ_BUFFER

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glCopyPixels

gl

D
r
a
w
B
u
f
f
e
r

g
l
E
n
d

g
l
R
e
a
d
P
i
x
e
l
s

glReadPixels function

2/22/2020 • 6 minutes to read • Edit Online

The **glReadPixels** function reads a block of pixels from the framebuffer.

Syntax

```
void WINAPI glReadPixels(
   GLint   x,
   GLint   y,
   GLsizei width,
   GLsizei height,
   GLenum  format,
   GLenum  type,
   GLvoid  *pixels
);
```

Parameters

x

The window *x* coordinate of the first pixel that is read from the framebuffer. Together with the *y* coordinate, specifies the location of the lower-left corner of a rectangular block of pixels.

y

The window *y* coordinates of the first pixel that is read from the framebuffer. Together with the *x* coordinate, specifies the location of the lower-left corner of a rectangular block of pixels.

width

The width of the pixel rectangle.

height

The height of the pixel rectangle. *Width* and *height* parameters of value "1" correspond to a single pixel.

format

The format of the pixel data. The following symbolic values are accepted:

VALUE	MEANING
GL_COLOR_INDEX	Color indexes are read from the color buffer selected by **glReadBuffer**. Each index is converted to fixed point, shifted left or right, depending on the value and sign of GL_INDEX_SHIFT, and added to GL_INDEX_OFFSET. If GL_MAP_COLOR is GL_TRUE, indexes are replaced by their mappings in the table GL_PIXEL_MAP_I_TO_I.
GL_STENCIL_INDEX	Stencil values are read from the stencil buffer. Each index is converted to fixed point, shifted left or right, depending on the value and sign of GL_INDEX_SHIFT, and added to GL_INDEX_OFFSET. If GL_MAP_STENCIL is GL_TRUE, indexes are replaced by their mappings in the table GL_PIXEL_MAP_S_TO_S.
GL_DEPTH_COMPONENT	Depth values are read from the depth buffer. Each component is converted to floating point such that the minimum depth value maps to 0.0 and the maximum value maps to 1.0. Each component is then multiplied by GL_DEPTH_SCALE, added to GL_DEPTH_BIAS, and finally clamped to the range [0,1].

VALUE	MEANING
GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_BGR_EXT, GL_BGRA_EXT, GL_LUMINANCE, GL_LUMINANCE_ALPHA	Processing differs depending on whether color buffers store color indexes or RGBA color components. If color indexes are stored, they are read from the color buffer selected by **glReadBuffer**. Each index is converted to fixed point, shifted left or right, depending on the value and sign of GL_INDEX_SHIFT, and added to GL_INDEX_OFFSET. Indexes are then replaced by the red, green, blue, and alpha values obtained by indexing the GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, and GL_PIXEL_MAP_I_TO_A tables. If RGBA color components are stored in the color buffers, they are read from the color buffer selected by **glReadBuffer**. Each color component is converted to floating point such that zero intensity maps to 0.0 and full intensity maps to 1.0. Each component is then multiplied by GL_c_SCALE and added to GL_c_BIAS, where c is GL_RED, GL_GREEN, GL_BLUE, and GL_ALPHA. Each component is clamped to the range [0,1]. Finally, if GL_MAP_COLOR is GL_TRUE, each color component c is replaced by its mapping in the table GL_PIXEL_MAP_c_TO_c, where c again is GL_RED, GL_GREEN, GL_BLUE, and GL_ALPHA. Each component is scaled to the size of its corresponding table before the lookup is performed. Finally, unneeded data is discarded. For example, GL_RED discards the green, blue, and alpha components, while GL_RGB discards only the alpha component. GL_LUMINANCE computes a single component value as the sum of the red, green, and blue components, and GL_LUMINANCE_ALPHA does the same, while keeping alpha as a second value.

type

The data type of the pixel data. Must be one of the following values.

TYPE	INDEX MASK	COMPONENT CONVERSION
GL_UNSIGNED_BYTE	281	(281)*c*
GL_BYTE	271	[(271)*c*-1]/2
GL_BITMAP	1	1

TYPE	INDEX MASK	COMPONENT CONVERSION
GL_UNSIGNED_SHORT	2 61	(2 61)*c*
GL_SHORT	2 51	[(2 51)*c*1]/2
GL_UNSIGNED_INT_	2 1	(2 1)*c*
GL_INT	2 1	[(2 1)*c*1]/2
GL_FLOAT	none	*c*

p
i
x
e
l
s

Returns the pixel data.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*format* or *type* was not an accepted value.
GL_INVALID_VALUE	Either *width* or *height* was negative.

NAME	MEANING
GL_INVALID_OPERATION	*format* was GL_COLOR_INDEX, and the color buffers stored RGBA or BGRA color components.
GL_INVALID_OPERATION	*format* was GL_STENCIL_INDEX, and there was no stencil buffer.
GL_INVALID_OPERATION	*format* was GL_DEPTH_COMPONENT, and there was no depth buffer.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*format* or *type* was not an accepted value.

NAME	MEANING
GL_INVALID_VALUE	Either *width* or *height* was negative.
GL_INVALID_OPERATION	*format* was GL_COLOR_INDEX, and the color buffers stored RGBA or BGRA color components.
GL_INVALID_OPERATION	*format* was GL_STENCIL_INDEX, and there was no stencil buffer.
GL_INVALID_OPERATION	*format* was GL_DEPTH_COMPONENT, and there was no depth buffer.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glReadPixels** function returns pixel data from the framebuffer, starting with the pixel whose lower-left corner is at location (*x*, *y*), into client memory starting at location *pixels*. Several parameters control the processing of the pixel data before it is placed into client memory. These parameters are set with three commands: **glPixelStore**, **glPixelTransfer**, and **glPixelMap**. This topic describes the effects on **glReadPixels** of most, but not all of the parameters specified by these three commands.

The **glReadPixels** function returns values from each pixel with lower-left corner at (x + i, y + j) for 0 = i < *width* and 0 = j < *height*. This pixel is said to be the *i*th pixel in the *j*th row. Pixels are returned in row order from the lowest to the highest row, left to right in each row.

The shift, scale, bias, and lookup factors described above are all specified by **glPixelTransfer**. The lookup table contents are specified by **glPixelMap**.

The final step involves converting the indexes or components to the proper format, as specified by *type*. If *format* is GL_COLOR_INDEX or GL_STENCIL_INDEX and *type* is not GL_FLOAT, each index is masked with the mask value given in the following table. If *type* is GL_FLOAT, then each integer index is converted to single-precision floating-point format.

If *format* is GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_BGR_EXT, GL_BGRA_EXT, GL_LUMINANCE, or GL_LUMINANCE_ALPHA and *type* is not GL_FLOAT, each component is multiplied by the multiplier shown in the preceding table. If type is GL_FLOAT, then each component is passed as is (or converted to the client's single-precision floating-point format if it is different from the one used by OpenGL).

Return values are placed in memory as follows. If *format* is GL_COLOR_INDEX, GL_STENCIL_INDEX, GL_DEPTH_COMPONENT, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, or GL_LUMINANCE, a single value is returned and the data for the *i*th pixel in the *j*th row is placed in location (j)*width* + *i*. GL_RGB and GL_BGR_EXT return three values, GL_RGBA and GL_BGRA_EXT return four values, and GL_LUMINANCE_ALPHA returns two values for each pixel, with all values corresponding to a single pixel occupying contiguous space in *pixels*. Storage parameters set by **glPixelStore**, such as GL_PACK_SWAP_BYTES and GL_PACK_LSB_FIRST, affect the way that data is written into memory. See **glPixelStore** for a description.

Values for pixels that lie outside the window connected to the current OpenGL context are undefined.

If an error is generated, no change is made to the contents of *pixels*.

The following function retrieves information related to **glReadPixels**:

glGet with argument GL_INDEX_MODE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glCopyPixels

glDrawPixels

glEnd

glPixelMap

glPixelStore

glPixelTransfer

glReadBuffer

glRect Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions draw a rectangle:

- glRectd
- glRectf
- glRecti
- glRects
- glRectdv
- glRectfv
- glRectiv
- glRectsv

glRectd function

2/22/2020 • 2 minutes to read • Edit Online

The **glRectd** function draws a rectangle.

Syntax

```
void WINAPI glRectd(
   GLdouble x1,
   GLdouble y1,
   GLdouble x2,
   GLdouble y2
);
```

Parameters

x 1 — The *x* coordinate of the vertex of a rectangle.

y 1 — The *y* coordinate of the vertex of a rectangle.

x 2 — The *x* coordinate of the opposite vertex of the rectangle.

y 2 — The *y* coordinate of the opposite vertex of the rectangle.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glRectd** function supports efficient specification of rectangles as two corner points. Each rectangle command takes four arguments, organized either as two consecutive pairs of (*x*, *y*) coordinates, or as two pointers to arrays, each containing an (*x*, *y*) pair. The resulting rectangle is defined in the *z* = 0 plane.

The **glRectd**(*x1, y1, x2, y2*) function is exactly equivalent to the following sequence:

glBegin(GL_POLYGON);

glVertex2(*x1, y1*);

glVertex2(*x2, y1*);

glVertex2(*x2, y2*);

glVertex2(*x1, y2*);

glEnd( );

Notice that if the second vertex is above and to the right of the first vertex, the rectangle is constructed with a counterclockwise winding.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glVertex

glRectdv function

2/22/2020 • 2 minutes to read • Edit Online

The **glRectdv** function draws a rectangle.

Syntax

```
void WINAPI glRectdv(
   const GLdouble *v1,
   const GLdouble *v2
);
```

Parameters

v1	A pointer to one vertex of a rectangle.
v2	a pointer to the opposite vertex of the rectangle.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glRectd** function supports efficient specification of rectangles as two corner points. Each rectangle command takes four arguments, organized either as two consecutive pairs of (*x*, *y*) coordinates, or as two pointers to arrays, each containing an (*x*, *y*) pair. The resulting rectangle is defined in the *z* = 0 plane.

The **glRectd**(*x1, y1, x2, y2*) function is exactly equivalent to the following sequence:

glBegin(GL_POLYGON);

glVertex2(*x1, y1*);

glVertex2(*x2, y1*);

glVertex2(*x2, y2*);

glVertex2(*x1, y2*);

glEnd( );

Notice that if the second vertex is above and to the right of the first vertex, the rectangle is constructed with a counterclockwise winding.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glVertex

glRectf function

2/22/2020 • 2 minutes to read • Edit Online

The **glRectf** function draws a rectangle.

Syntax

```
void WINAPI glRectf(
   GLfloat x1,
   GLfloat y1,
   GLfloat x2,
   GLfloat y2
);
```

Parameters

x 1 — The *x* coordinate of the vertex of a rectangle.

y 1 — The *y* coordinate of the vertex of a rectangle.

x 2 — The *x* coordinate of the opposite vertex of the rectangle.

y 2 — The *y* coordinate of the opposite vertex of the rectangle.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glRectf** function supports efficient specification of rectangles as two corner points. Each rectangle command takes four arguments, organized either as two consecutive pairs of (*x*, *y*) coordinates, or as two pointers to arrays, each containing an (*x*, *y*) pair. The resulting rectangle is defined in the *z* = 0 plane.

The **glRectf**(*x1, y1, x2, y2*) function is exactly equivalent to the following sequence:

glBegin(GL_POLYGON);

glVertex2(*x1, y1*);

glVertex2(*x2, y1*);

glVertex2(*x2, y2*);

glVertex2(*x1, y2*);

glEnd( );

Notice that if the second vertex is above and to the right of the first vertex, the rectangle is constructed with a counterclockwise winding.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glVertex

glRectfv function

2/22/2020 • 2 minutes to read • Edit Online

The **glRectfv** function draws a rectangle.

Syntax

```
void WINAPI glRectfv(
   const GLfloat *v1,
   const GLfloat *v2
);
```

Parameters

v1	A pointer to one vertex of a rectangle.
v2	a pointer to the opposite vertex of the rectangle.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glRectf** function supports efficient specification of rectangles as two corner points. Each rectangle command takes four arguments, organized either as two consecutive pairs of (*x*, *y*) coordinates, or as two pointers to arrays, each containing an (*x*, *y*) pair. The resulting rectangle is defined in the *z* = 0 plane.

The **glRectf**(*x1, y1, x2, y2*) function is exactly equivalent to the following sequence:

glBegin(GL_POLYGON);

glVertex2(*x1, y1*);

glVertex2(*x2, y1*);

glVertex2(*x2, y2*);

glVertex2(*x1, y2*);

glEnd( );

Notice that if the second vertex is above and to the right of the first vertex, the rectangle is constructed with a counterclockwise winding.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glVertex

glRecti function

2/22/2020 • 2 minutes to read • Edit Online

The **glRecti** function draws a rectangle.

Syntax

```
void WINAPI glRecti(
   GLint x1,
   GLint y1,
   GLint x2,
   GLint y2
);
```

Parameters

x1 — The *x* coordinate of the vertex of a rectangle.

y1 — The *y* coordinate of the vertex of a rectangle.

x2 — The *x* coordinate of the opposite vertex of the rectangle.

y2 — The *y* coordinate of the opposite vertex of the rectangle.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glRecti** function supports efficient specification of rectangles as two corner points. Each rectangle command takes four arguments, organized either as two consecutive pairs of (*x*, *y*) coordinates, or as two pointers to arrays, each containing an (*x*, *y*) pair. The resulting rectangle is defined in the *z* = 0 plane.

The **glRecti**(*x1, y1, x2, y2*) function is exactly equivalent to the following sequence:

glBegin(GL_POLYGON);

glVertex2(*x1, y1*);

glVertex2(*x2, y1*);

glVertex2(*x2, y2*);

glVertex2(*x1, y2*);

glEnd( );

Notice that if the second vertex is above and to the right of the first vertex, the rectangle is constructed with a counterclockwise winding.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glVertex

glRectiv function

2/22/2020 • 2 minutes to read • Edit Online

The **glRectiv** function draws a rectangle.

Syntax

```
void WINAPI glRectiv(
   const GLint *v1,
   const GLint *v2
);
```

Parameters

v1	A pointer to one vertex of a rectangle.
v2	a pointer to the opposite vertex of the rectangle.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glRecti** function supports efficient specification of rectangles as two corner points. Each rectangle command takes four arguments, organized either as two consecutive pairs of (*x*, *y*) coordinates, or as two pointers to arrays, each containing an (*x*, *y*) pair. The resulting rectangle is defined in the *z* = 0 plane.

The **glRecti**(*x1, y1, x2, y2*) function is exactly equivalent to the following sequence:

glBegin(GL_POLYGON);

glVertex2(*x1, y1*);

glVertex2(*x2, y1*);

glVertex2(*x2, y2*);

glVertex2(*x1, y2*);

glEnd( );

Notice that if the second vertex is above and to the right of the first vertex, the rectangle is constructed with a counterclockwise winding.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glVertex

glRects function

2/22/2020 • 2 minutes to read • Edit Online

The **glRects** function draws a rectangle.

Syntax

```
void WINAPI glRects(
   GLshort x1,
   GLshort y1,
   GLshort x2,
   GLshort y2
);
```

Parameters

x1 — The *x* coordinate of the vertex of a rectangle.

y1 — The *y* coordinate of the vertex of a rectangle.

x2 — The *x* coordinate of the opposite vertex of the rectangle.

y2 — The *y* coordinate of the opposite vertex of the rectangle.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glRects** function supports efficient specification of rectangles as two corner points. Each rectangle command takes four arguments, organized either as two consecutive pairs of (x, *y*) coordinates, or as two pointers to arrays, each containing an (*x*, *y*) pair. The resulting rectangle is defined in the *z* = 0 plane.

The **glRects**(*x1, y1, x2, y2*) function is exactly equivalent to the following sequence:

glBegin(GL_POLYGON);

glVertex2(*x1, y1*);

glVertex2(*x2, y1*);

glVertex2(*x2, y2*);

glVertex2(*x1, y2*);

glEnd( );

Notice that if the second vertex is above and to the right of the first vertex, the rectangle is constructed with a counterclockwise winding.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glVertex

glRectsv function

2/22/2020 • 2 minutes to read • Edit Online

The **glRectsv** function draws a rectangle.

Syntax

```
void WINAPI glRectsv(
   const GLshort *v1,
   const GLshort *v2
);
```

Parameters

v1	A pointer to one vertex of a rectangle.
v2	a pointer to the opposite vertex of the rectangle.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glRects** function supports efficient specification of rectangles as two corner points. Each rectangle command takes four arguments, organized either as two consecutive pairs of (*x*, *y*) coordinates, or as two pointers to arrays, each containing an (*x*, *y*) pair. The resulting rectangle is defined in the *z* = 0 plane.

The **glRects**(*x1, y1, x2, y2*) function is exactly equivalent to the following sequence:

glBegin(GL_POLYGON);

glVertex2(*x1, y1*);

glVertex2(*x2, y1*);

glVertex2(*x2, y2*);

glVertex2(*x1, y2*);

glEnd( );

Notice that if the second vertex is above and to the right of the first vertex, the rectangle is constructed with a counterclockwise winding.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glVertex

glRenderMode function

2/22/2020 • 2 minutes to read • Edit Online

The **glRenderMode** function sets the rasterization mode.

Syntax

```
GLint WINAPI glRenderMode(
   GLenum mode
);
```

Parameters

mode The rasterization mode. The following three values are accepted. The default value is GL_RENDER.

VALUE	MEANING
GL_RENDER	Render mode. Primitives are rasterized, producing pixel fragments, which are written into the framebuffer. This is the normal mode and also the default mode.
GL_SELECT	Selection mode. No pixel fragments are produced, and no change to the framebuffer contents is made. Instead, a record of the names of primitives that would have been drawn if the render mode was GL_RENDER is returned in a select buffer, which must be created (see **glSelectBuffer**) before selection mode is entered.
GL_FEEDBACK	Feedback mode. No pixel fragments are produced, and no change to the framebuffer contents is made. Instead, the coordinates and attributes of vertices that would have been drawn had the render mode been GL_RENDER are returned in a feedback buffer, which must be created (see **glFeedbackBuffer**) before feedback mode is entered.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*mode* was not one of three accepted values.
GL_INVALID_OPERATION	The function was called with argument GL_SELECT before **glSelectBuffer** was called at least once.
GL_INVALID_OPERATION	The function was called with argument GL_FEEDBACK before **glBeedbackBuffer** was called at least once.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*mode* was not one of three accepted values.

NAME	MEANING
GL_INVALID_OPERATION	The function was called with argument GL_SELECT before **glSelectBuffer** was called at least once.
GL_INVALID_OPERATION	The function was called with argument GL_FEEDBACK before **glBeedbackBuffer** was called at least once.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glRenderMode** function takes one argument, *mode*, which can assume one of three predefined values above.

The return value of the **glRenderMode** function is determined by the render mode at the time **glRenderMode** is called, rather than by *mode*. The values returned for the three render modes are as follows.

VALUE	MEANING
GL_RENDER	Zero.
GL_SELECT	The number of hit records transferred to the select buffer.
GL_FEEDBACK	The number of values (not vertices) transferred to the feedback buffer.

Refer to **glSelectBuffer** and **glFeedbackBuffer** for more details concerning selection and feedback operation.

If an error is generated, **glRenderMode** returns zero regardless of the current render mode.

The following function retrieves information related to **glRenderMode**:

glGet with argument GL_RENDER_MODE

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib
DLL	Ope ngl3 2.dll

See also

glBegin

glEnd

glFeedbackBuffer

glInitNames

glLoadName

glPassThrough

glPushName

glSel

e
c
t
B
u
f
f
e
r

glRotate Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions multiply the current matrix by a rotation matrix:

- **glRotated**
- **glRotatef**

glRotated function

2/22/2020 • 2 minutes to read • Edit Online

The **glRotated** function multiplies the current matrix by a rotation matrix.

Syntax

```
void WINAPI glRotated(
   GLdouble angle,
   GLdouble x,
   GLdouble y,
   GLdouble z
);
```

Parameters

angle	The angle of rotation, in degrees.
x	The *x* coordinate of a vector.
y	The *y* coordinate of a vector.
z	The *z* coordinate of a vector.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Remarks

The **glRotated** function computes a matrix that performs a counterclockwise rotation of *angle* degrees about the vector from the origin through the point (*x*, *y*, *z*).

The current matrix (see glMatrixMode) is multiplied by this rotation matrix, with the product replacing the current matrix. That is, if M is the current matrix and R is the translation matrix, then M is replaced with M R.

If the matrix mode is either GL_MODELVIEW or GL_PROJECTION, all objects drawn after **glRotated** is called are rotated. Use glPushMatrix and glPopMatrix to save and restore the unrotated coordinate system.

The following functions retrieve information related to **glRotated**:

glGet with argument GL_RENDER_MODE

glGet with argument GL_MATRIX_MODE

glGet with argument GL_MODELVIEW_MATRIX

glGet with argument GL_PROJECTION_MATRIX

glGet with argument GL_TEXTURE_MATRIX

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glMatrixMode

glMultMatrix

glPopMatrix

glPushMatrix

glScale

glTranslate

glRotatef function

2/22/2020 • 2 minutes to read • Edit Online

The **glRotatef** function multiplies the current matrix by a rotation matrix.

Syntax

```
void WINAPI glRotatef(
   GLfloat angle,
   GLfloat x,
   GLfloat y,
   GLfloat z
);
```

Parameters

angle	The angle of rotation, in degrees.
x	The *x* coordinate of a vector.
y	The *y* coordinate of a vector.
z	The *z* coordinate of a vector.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Remarks

The **glRotatef** function computes a matrix that performs a counterclockwise rotation of *angle* degrees about the vector from the origin through the point (*x*, *y*, *z*).

The current matrix (see glMatrixMode) is multiplied by this rotation matrix, with the product replacing the current matrix. That is, if M is the current matrix and R is the translation matrix, then M is replaced with M R.

If the matrix mode is either GL_MODELVIEW or GL_PROJECTION, all objects drawn after **glRotatef** is called are rotated. Use glPushMatrix and glPopMatrix to save and restore the unrotated coordinate system.

The following functions retrieve information related to **glRotatef**:

glGet with argument GL_RENDER_MODE

glGet with argument GL_MATRIX_MODE

glGet with argument GL_MODELVIEW_MATRIX

glGet with argument GL_PROJECTION_MATRIX

glGet with argument GL_TEXTURE_MATRIX

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glMatrixMode

glMultMatrix

glPopMatrix

glPushMatrix

glScale

glTranslate

glScale Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions multiply the current matrix by a general scaling matrix:

- glScaled
- glScalef

glScaled function

2/22/2020 • 2 minutes to read • Edit Online

The **glScaled** and glScalef functions multiply the current matrix by a general scaling matrix.

Syntax

```
void WINAPI glScaled(
   GLdouble x,
   GLdouble y,
   GLdouble z
);
```

Parameters

x	Scale factors along the *x* axis.
y	Scale factors along the *y* axis.
z	Scale factors along the *z* axis.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Error codes

The following error code can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glScaled** function produces a general scaling along the *x*, *y*, and *z* axes. The three arguments indicate the desired scale factors along each of the three axes. The resulting matrix is

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

The current matrix (see **glMatrixMode**) is multiplied by this scale matrix, with the product replacing the current matrix. That is, if M is the current matrix and S is the scale matrix, then M is replaced with M S.

If the matrix mode is either GL_MODELVIEW or GL_PROJECTION, all objects drawn after **glScaled** is called are scaled. Use **glPushMatrix** and **glPopMatrix** to save and restore the unscaled coordinate system.

If scale factors other than 1.0 are applied to the modelview matrix and lighting is enabled, automatic normalization of normals should probably also be enabled (**glEnable** and **glDisable** with argument GL_NORMALIZE).

The following functions retrieve information related to **glScaled**:

glGet with argument GL_MATRIX_MODE

glGet with argument GL_MODELVIEW_MATRIX

glGet with argument GL_PROJECTION_MATRIX

glGet with argument GL_TEXTURE_MATRIX

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib

DLL	Opengl32.dll

See also

glBegin

glEnd

glMatrixMode

glMultMatrix

glPo

pMatrix

glPushMatrix

glRotated

glRotatef

glTranslate

glScalef function

2/22/2020 • 2 minutes to read • Edit Online

The **glScaled** and **glScalef** functions multiply the current matrix by a general scaling matrix.

Syntax

```
void WINAPI glScalef(
   GLfloat x,
   GLfloat y,
   GLfloat z
);
```

Parameters

x	Scale factors along the *x* axis.
y	Scale factors along the *y* axis.
z	Scale factors along the *z* axis.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glScalef** function produces a general scaling along the *x*, *y*, and *z* axes. The three arguments indicate the desired scale factors along each of the three axes. The resulting matrix appears in the following image.

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

The current matrix (see **glMatrixMode**) is multiplied by this scale matrix, with the product replacing the current matrix. That is, if M is the current matrix and S is the scale matrix, then M is replaced with M S.

If the matrix mode is either GL_MODELVIEW or GL_PROJECTION, all objects drawn after **glScalef** is called are scaled. Use **glPushMatrix** and **glPopMatrix** to save and restore the unscaled coordinate system.

If scale factors other than 1.0 are applied to the modelview matrix and lighting is enabled, automatic normalization of normals should probably also be enabled (**glEnable** and **glDisable** with argument GL_NORMALIZE).

The following functions retrieve information related to **glScalef**:

glGet with argument GL_MATRIX_MODE

glGet with argument GL_MODELVIEW_MATRIX

glGet with argument GL_PROJECTION_MATRIX

glGet with argument GL_TEXTURE_MATRIX

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib

DLL	Opengl32.dll

See also

glBegin

glEnd

glMatrixMode

glMultMatrix

glPo

p
M
a
t
r
i
x

g
l
P
u
s
h
M
a
t
r
i
x

g
l
R
o
t
a
t
e
d

g
l
R
o
t
a
t
e
f

g
l
T
r
a
n
s
l
a
t
e

glScissor function

2/22/2020 • 2 minutes to read • Edit Online

The **glScissor** function defines the scissor box.

Syntax

```
void WINAPI glScissor(
   GLint   x,
   GLint   y,
   GLsizei width,
   GLsizei height
);
```

Parameters

x

The x (vertical axis) coordinate for the lower-left corner of the scissor box.

y

The y (horizontal axis) coordinate for the lower-left corner of the scissor box. Together, x and y specify the lower-left corner of the scissor box. Initially (0,0).

width

The width of the scissor box.

height

The height of the scissor box. When an OpenGL context is *first* attached to a window, *width* and *height* are set to the dimensions of that window.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	Either *width* or *height* was negative.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	Either *width* or *height* was negative.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glScissor** function defines a rectangle, called the scissor box, in window coordinates. The first two parameters, *x* and *y*, specify the lower-left corner of the box. The *width* and *height* parameters specify the width and height of the box.

The scissor test is enabled and disabled using **glEnable** and **glDisable** with argument GL_SCISSOR_TEST. While the scissor test is enabled, only pixels that lie within the scissor box can be modified by drawing commands. Window coordinates have integer values at the shared corners of framebuffer pixels, so **glScissor**(0,0,1,1) allows only the lower-left pixel in the window to be modified, and **glScissor**(0,0,0,0) disallows modification to all pixels in the window.

When the scissor test is disabled, it is as though the scissor box includes the entire window.

The following functions retrieve information related to **glScissor**:

glGet with argument GL_SCISSOR_BOX

glIsEnabled with argument GL_SCISSOR_TEST

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnable

glEnd

g

IsEnabled

glViewport

glSelectBuffer function

2/22/2020 • 3 minutes to read • Edit Online

The **glSelectBuffer** function establishes a buffer for selection mode values.

Syntax

```
void WINAPI glSelectBuffer(
   GLsizei size,
   GLuint  *buffer
);
```

Parameters

size	The size of *buffer*.
buffer	Returns the selection data.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	*size* was negative.

NAME	MEANING
GL_INVALID_OPERATION	The function was called while the render mode was GL_SELECT.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	*size* was negative.
GL_INVALID_OPERATION	The function was called while the render mode was GL_SELECT.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glSelectBuffer** function has two parameters: *buffer* is a pointer to an array of unsigned integers, and *size* indicates the size of the array. The *buffer* parameter returns values from the name stack (see **glInitNames**, **glLoadName**, **glPushName**) when the rendering mode is GL_SELECT (see **glRenderMode**). The **glSelectBuffer** function must be issued before selection mode is enabled, and it must not be issued while the rendering mode is GL_SELECT.

Selection is used by a programmer to determine which primitives are drawn into some region of a window. The region is defined by the current modelview and perspective matrices.

In selection mode, no pixel fragments are produced from rasterization. Instead, if a primitive intersects the clip volume defined by the viewing frustum and the user-defined clipping planes, this primitive causes a selection hit. (With polygons, no hit occurs if the polygon is culled.) When a change is made to the name stack, or when **glRenderMode** is called, a hit record is copied to *buffer* if any hits have occurred since the last such event (either a name stack change or a **glRenderMode** call). The hit record consists of the number of names in the name stack at the time of the event; followed by the minimum and maximum depth values of all vertices that hit since the previous event; followed by the name stack contents, bottom name first.

Returned depth values are mapped such that the largest unsigned integer value corresponds to window coordinate depth 1.0, and zero corresponds to window coordinate depth 0.0.

An internal index into *buffer* is reset to zero whenever selection mode is entered. Each time a hit record is copied into *buffer*, the index is incremented to point to the cell just past the end of the block of namesthat is, to the next available cell. If the hit record is larger than the number of remaining locations in *buffer*, as much data as can fit is copied, and the overflow flag is set. If the name stack is empty when a hit record is copied, that record consists of zero followed by the minimum and maximum depth values.

Selection mode is exited by calling **glRenderMode** with an argument other than GL_SELECT. Whenever **glRenderMode** is called while the render mode is GL_SELECT, it returns the number of hit records copied to *buffer*, resets the overflow flag and the selection buffer pointer, and initializes the name stack to be empty. If the overflow bit was set when **glRenderMode** was called, a negative hit record count is returned.

The contents of *buffer* are undefined until **glRenderMode** is called with an argument other than GL_SELECT.

The **glBegin/glEnd** primitives and calls to **glRasterPos** can result in hits.

The following function retrieves information related to **glSelectBuffer**:

glGet with argument GL_NAME_STACK_DEPTH

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glBegin

glEnd

glFeedbackBuffer

glInitNames

g

lLoadName

glPushName

glRenderMode

glShadeModel function

2/22/2020 • 2 minutes to read • Edit Online

The **glShadeModel** function selects flat or smooth shading.

Syntax

```
void WINAPI glShadeModel(
   GLenum mode
);
```

Parameters

m
o
d
e

A symbolic value representing a shading technique. Accepted values are GL_FLAT and GL_SMOOTH. The default is GL_SMOOTH.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*mode* was a value other than GL_GLAT or GL_SMOOTH.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*mode* was a value other than GL_GLAT or GL_SMOOTH.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

OpenGL primitives can have either flat or smooth shading. Smooth shading, the default, causes the computed colors of vertices to be interpolated as the primitive is rasterized, typically assigning different colors to each resulting pixel fragment. Flat shading selects the computed color of just one vertex and assigns it to all the pixel fragments generated by rasterizing a single primitive. In either case, the computed color of a vertex is the result of lighting, if lighting is enabled, or it is the current color at the time the vertex was specified, if lighting is disabled.

Flat and smooth shading are indistinguishable for points. Counting vertices and primitives from one, starting when **glBegin** is issued, each flat-shaded line segment *i* is given the computed color of vertex *i* + 1, its second vertex. Counting similarly from one, each flat-shaded polygon is given the computed color of the vertex listed in the following table. This is the last vertex to specify the polygon in all cases except single polygons, where the first vertex specifies the flat-shaded color.

PRIMITIVE TYPE OF POLYGON I	VERTEX
Single polygon (*I*=1)	1
Triangle strip	*i* + 2
Triangle fan	*i* + 2
Independent triangle	3*I*
Quad strip	2*i* + 2
Independent quad	4*I*

Flat and smooth shading are specified by **glShadeModel** with *mode* set to GL_FLAT and GL_SMOOTH, respectively.

The following function retrieves information related to **glShadeModel**:

glGet with argument GL_SHADE_MODEL

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glColor

glEnd

glLig

h
t

g
l
L
i
g
h
t
M
o
d
e
l

glStencilFunc function

2/22/2020 • 3 minutes to read • Edit Online

The **glStencilFunc** function sets the function and reference value for stencil testing.

Syntax

```
void WINAPI glStencilFunc(
   GLenum func,
   GLint  ref,
   GLuint mask
);
```

Parameters

f
u
n
c

The test function. The following eight tokens are valid.

VALUE	MEANING
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*).

VALUE	MEANING
GL_EQUAL	Passes if (*ref* & *mask*) = (*stencil* & *mask*).
GL_NOTEQUAL	Passes if (*ref* & *mask*) ? (*stencil* & *mask*).
GL_ALWAYS	Always passes.

ref

The reference value for the stencil test. The *ref* parameter is clamped to the range [0, 2*n* 1], where *n* is the number of bitplanes in the stencil buffer.

mask

A mask that is **AND**ed with both the reference value and the stored stencil value when the test is done.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*func* was not one of the eight accepted values.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*func* was not one of the eight accepted values.
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Remarks

Stenciling, like *z*-buffering, enables and disables drawing on a per-pixel basis. You draw into the stencil planes using OpenGL drawing primitives, then render geometry and images, using the stencil planes to mask out portions of the screen. Stenciling is typically used in multipass rendering algorithms to achieve special effects, such as decals, outlining, and constructive solid geometry rendering.

The stencil test conditionally eliminates a pixel based on the outcome of a comparison between the reference value and the value in the stencil buffer. The test is enabled by glEnable and glDisable with argument GL_STENCIL_TEST. Actions taken based on the outcome of the stencil test are specified with glStencilOp.

The *func* parameter is a symbolic constant that determines the stencil comparison function. It accepts one of the eight values shown above. The *ref* parameter is an integer reference value that is used in the stencil comparison. It is clamped to the range [0, 2*n* 1], where *n* is the number of bitplanes in the stencil buffer. The *mask* parameter is bitwise **AND**ed with both the reference value and the stored stencil value, with the **AND**ed values participating in the comparison.

If *stencil* represents the value stored in the corresponding stencil buffer location, the preceding list shows the effect of each comparison function that can be specified by *func*. Only if the comparison succeeds is

the pixel passed through to the next stage in the rasterization process (see glStencilOp). All tests treat *stencil* values as unsigned integers in the range [0, 2*n* 1], where *n* is the number of bitplanes in the stencil buffer.

Initially, the stencil test is disabled. If there is no stencil buffer, no stencil modification can occur and it is as if the stencil test always passes.

The following functions retrieve information related to **glStencilFunc**:

glGet with argument GL_STENCIL_FUNC

glGet with argument GL_STENCIL_VALUE_MASK

glGet with argument GL_STENCIL_REF

glGet with argument GL_STENCIL_BITS

glIsEnabled with argument GL_STENCIL_TEST

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glAlphaFunc

gl

B
e
g
i
n

g
l
B
l
e
n
d
F
u
n
c

g
l
D
e
p
t
h
F
u
n
c

g
l
E
n
a
b
l
e

g
l
E
n
d

g
l
I
s
E
n
a
b

l
e
d

g
l
L
o
g
i
c
O
p

g
l
S
t
e
n
c
i
l
O
p

glStencilMask function

2/22/2020 • 2 minutes to read • Edit Online

The **glStencilMask** function controls the writing of individual bits in the stencil planes.

Syntax

```
void WINAPI glStencilMask(
   GLuint mask
);
```

Parameters

m
a
s
k

A bit mask to enable and disable writing of individual bits in the stencil planes. Initially, the mask is all ones.

Return value

This function does not return a value.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error code can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glStencilMask** function controls the writing of individual bits in the stencil planes. The least significant *n* bits of *mask*, where *n* is the number of bits in the stencil buffer, specify a mask. Wherever a one appears in the mask, the corresponding bit in the stencil buffer is made writable. Where a zero appears, the bit is write-protected. Initially, all bits are enabled for writing.

The following functions retrieve information related to **glStencilMask**:

glGet with argument GL_STENCIL_WRITEMASK

glGet with argument GL_STENCIL_BITS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

glColorMask

glDepthMask

glEnd

glIndexMask

glStencil

F
u
n
c

g
l
S
t
e
n
c
i
l
O
p

glStencilOp function

2/22/2020 • 2 minutes to read • Edit Online

The **glStencilOp** function sets the stencil test actions.

Syntax

```
void WINAPI glStencilOp(
   GLenum fail,
   GLenum zfail,
   GLenum zpass
);
```

Parameters

fail

The action to take when the stencil test fails. The following six symbolic constants are accepted.

VALUE	MEANING
GL_KEEP	Keeps the current value.
GL_ZERO	Sets the stencil buffer value to zero.
GL_REPLACE	Sets the stencil buffer value to *ref*, as specified by **glStencilFunc**.
GL_INCR	Increments the current stencil buffer value. Clamps to the maximum representable unsigned value.
GL_DECR	Decrements the current stencil buffer value. Clamps to zero.
GL_INVERT	Bitwise inverts the current stencil buffer value.

zfail

Stencil action when the stencil test passes, but the depth test fails. Accepts the same symbolic constants as *fail.*

zpass

Stencil action when both the stencil test and the depth test pass, or when the stencil test passes and either there is no depth buffer or depth testing is not enabled. Accepts the same symbolic constants as *fail*.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*fail*, *zfail*, or *zpass* was any value other than the six defined constant values.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*fail*, *zfail*, or *zpass* was any value other than the six defined constant values.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

Stenciling, like *z*-buffering, enables and disables drawing on a per-pixel basis. You draw into the stencil planes using OpenGL drawing primitives, then render geometry and images, using the stencil planes to mask out portions of the screen. Stenciling is typically used in multipass rendering algorithms to achieve special effects, such as decals, outlining, and constructive solid geometry rendering.

The stencil test conditionally eliminates a pixel based on the outcome of a comparison between the value in the stencil buffer and a reference value. The test is enabled with **glEnable** and **glDisable** calls with argument GL_STENCIL_TEST, and controlled with **glStencilFunc**.

The **glStencilOp** function takes three arguments that indicate what happens to the stored stencil value while stenciling is enabled. If the stencil test fails, no change is made to the pixel's color or depth buffers, and *fail* specifies what happens to the stencil buffer contents.

Stencil buffer values are treated as unsigned integers. When incremented and decremented, values are clamped to 0 and 2*n* 1, where *n* is the value returned by querying GL_STENCIL_BITS.

The other two arguments to **glStencilOp** specify stencil buffer actions should subsequent depth buffer tests succeed (*zpass*) or fail (*zfail*). (See **glDepthFunc**.) They are specified using the same six symbolic constants as *fail*. Note that *zfail* is ignored when there is no depth buffer, or when the depth buffer is not enabled. In these cases, *fail* and *zpass* specify stencil action when the stencil test fails and passes, respectively.

Initially the stencil test is disabled. If there is no stencil buffer, no stencil modification can occur and it is as if the stencil tests always pass, regardless of any call to **glStencilOp**.

The following functions retrieve information related to **glStencilOp**:

glGet with argument GL_STENCIL_FAIL

glGet with argument GL_STENCIL_PASS_DEPTH_PASS

glGet with argument GL_STENCIL_PASS_DEPTH_FAIL

glGet with argument GL_STENCIL_BITS

glIsEnabled with argument GL_STENCIL_TEST

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glAlphaFunc

glBegin

glBlendFun

c

glDepthFunc

glEnable

glEnd

glIsEnabled

glLogicOp

glS

t
e
n
c
i
l
F
u
n
c

glTexCoord Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions set the current texture coordinates:

- glTexCoord1d
- glTexCoord1f
- glTexCoord1i
- glTexCoord1s
- glTexCoord2d
- glTexCoord2i
- glTexCoord2s
- glTexCoord3d
- glTexCoord3f
- glTexCoord3i
- glTexCoord3s
- glTexCoord4d
- glTexCoord4f
- glTexCoord4i
- glTexCoord4s
- glTexCoord1dv
- glTexCoord1fv
- glTexCoord1iv
- glTexCoord1sv
- glTexCoord2dv
- glTexCoord2fv
- glTexCoord2iv
- glTexCoord2sv
- glTexCoord3dv
- glTexCoord3fv
- glTexCoord3iv
- glTexCoord3sv
- glTexCoord4dv
- glTexCoord4fv
- glTexCoord4iv
- glTexCoord4sv

glTexCoord1d function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord1d(
   GLdouble s
);
```

Parameters

s	The s texture coordinate.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glVertex

glTexCoord1dv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord1dv(
   const GLdouble *v
);
```

Parameters

v	A pointer to an array of texture coordinates.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glVertex

glTexCoord1f function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord1f(
   GLfloat s
);
```

Parameters

s	The s texture coordinate.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glVertex

glTexCoord1fv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord1fv(
   const GLfloat *v
);
```

Parameters

v	A pointer to an array of texture coordinates.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glVertex

glTexCoord1i function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord1i(
   GLint s
);
```

Parameters

s	The s texture coordinate.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glVertex

glTexCoord1iv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord1iv(
   const GLint *v
);
```

Parameters

v	A pointer to an array of texture coordinates.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glVertex

glTexCoord1s function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord1s(
   GLshort s
);
```

Parameters

s	The s texture coordinate.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glVertex

glTexCoord1sv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord1sv(
   const GLshort *v
);
```

Parameters

v	A pointer to an array of texture coordinates.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glVertex

glTexCoord2d function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord2d(
   GLdouble s,
   GLdouble t
);
```

Parameters

s	The s texture coordinate.
t	The t texture coordinate.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glVertex

glTexCoord2dv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord2dv(
   const GLdouble *v
);
```

Parameters

v	A pointer to an array of two elements, which in turn specifies the s and t texture coordinates.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glVertex

glTexCoord2fv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord2fv(
   const GLfloat *v
);
```

Parameters

v	A pointer to an array of two elements, which in turn specifies the s and t texture coordinates.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glVertex

glTexCoord2i function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord2i(
   GLint s,
   GLint t
);
```

Parameters

s	The s texture coordinate.
t	The t texture coordinate.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glVertex

glTexCoord2iv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord2iv(
   const GLint *v
);
```

Parameters

v	A pointer to an array of two elements, which in turn specifies the s and t texture coordinates.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glVertex

glTexCoord2s function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord2s(
   GLshort s,
   GLshort t
);
```

Parameters

s	The s texture coordinate.
t	The t texture coordinate.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glVertex

glTexCoord2sv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord2sv(
   const GLshort *v
);
```

Parameters

v	A pointer to an array of two elements, which in turn specifies the s and t texture coordinates.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glVertex

glTexCoord3d function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord3d(
   GLdouble s,
   GLdouble t,
   GLdouble r
);
```

Parameters

s	The s texture coordinate.
t	The t texture coordinate.
r	The r texture coordinate.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Ope ngl3 2.lib
DLL	Ope ngl3 2.dll

See also

g
l
V
e
r
t
e
x

glTexCoord3dv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord3dv(
   const GLdouble *v
);
```

Parameters

v	A pointer to an array of three elements, which in turn specifies the s, t, and r texture coordinates.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glVertex

cglTexCoord3f function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord3f(
   GLfloat s,
   GLfloat t,
   GLfloat r
);
```

Parameters

s	The s texture coordinate.
t	The t texture coordinate.
r	The r texture coordinate.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Ope ngl3 2.lib
DLL	Ope ngl3 2.dll

See also

g
l
V
e
r
t
e
x

glTexCoord3fv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord3fv(
   const GLfloat *v
);
```

Parameters

v	A pointer to an array of three elements, which in turn specifies the s, t, and r texture coordinates.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glVertex

glTexCoord3i function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord3i(
   GLint s,
   GLint t,
   GLint r
);
```

Parameters

s	The s texture coordinate.
t	The t texture coordinate.
r	The r texture coordinate.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Ope ngl3 2.lib
DLL	Ope ngl3 2.dll

See also

g
l
V
e
r
t
e
x

glTexCoord3iv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord3iv(
   const GLint *v
);
```

Parameters

v	A pointer to an array of three elements, which in turn specifies the s, t, and r texture coordinates.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glVertex

glTexCoord3s function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord3s(
   GLshort s,
   GLshort t,
   GLshort r
);
```

Parameters

s	The s texture coordinate.
t	The t texture coordinate.
r	The r texture coordinate.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Ope ngl3 2.lib
DLL	Ope ngl3 2.dll

See also

g
l
V
e
r
t
e
x

glTexCoord3sv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord3sv(
   const GLshort *v
);
```

Parameters

v	A pointer to an array of three elements, which in turn specifies the s, t, and r texture coordinates.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glVertex

glTexCoord4d function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord4d(
   GLdouble s,
   GLdouble t,
   GLdouble r,
   GLdouble q
);
```

Parameters

s	The s texture coordinate.
t	The t texture coordinate.
r	The r texture coordinate.
q	The q texture coordinate.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glVertex

glTexCoord4dv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord4dv(
   const GLdouble *v
);
```

Parameters

v	A pointer to an array of four elements, which in turn specifies the s, t, r, and q texture coordinates.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glVertex

glTexCoord4f function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord4f(
   GLfloat s,
   GLfloat t,
   GLfloat r,
   GLfloat q
);
```

Parameters

s	The s texture coordinate.
t	The t texture coordinate.
r	The r texture coordinate.
q	The q texture coordinate.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glVertex

glTexCoord4fv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord4fv(
   const GLfloat *v
);
```

Parameters

v	A pointer to an array of four elements, which in turn specifies the s, t, r, and q texture coordinates.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glVertex

glTexCoord4i function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord4i(
   GLint s,
   GLint t,
   GLint r,
   GLint q
);
```

Parameters

s	The s texture coordinate.
t	The t texture coordinate.
r	The r texture coordinate.
q	The q texture coordinate.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glVertex

glTexCoord4iv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord4iv(
   const GLint *v
);
```

Parameters

v	A pointer to an array of four elements, which in turn specifies the s, t, r, and q texture coordinates.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glVertex

glTexCoord4s function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord4s(
   GLshort s,
   GLshort t,
   GLshort r,
   GLshort q
);
```

Parameters

s	The s texture coordinate.
t	The t texture coordinate.
r	The r texture coordinate.
q	The q texture coordinate.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glVertex

glTexCoord4sv function

2/22/2020 • 2 minutes to read • Edit Online

Sets the current texture coordinates.

Syntax

```
void WINAPI glTexCoord4dv(
   const GLshort *v
);
```

Parameters

v	A pointer to an array of four elements, which in turn specifies the s, t, r, and q texture coordinates.

Return value

This function does not return a value.

Remarks

The **glTexCoord** function sets the current texture coordinates that are part of the data associated with polygon vertices. The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. The glTexCoord1 function sets the current texture coordinates to (s, 0, 0, 1); a call to glTexCoord2 sets them to (s, t, 0, 1). Similarly, glTexCoord3 specifies the texture coordinates as (s, t, r, 1), and glTexCoord4 defines all four components explicitly as (s, t, r, q). You can update the current texture coordinates at any time. In particular, you can call glTexCoord between a call to **glBegin** and the corresponding call to **glEnd**. The following function retrieves information related to **glTexCoord**:

glGet with argument GL_CURRENT_TEXTURE_COORDS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glVertex

glTexCoordPointer function

2/22/2020 • 2 minutes to read • Edit Online

The **glTexCoordPointer** function defines an array of texture coordinates.

Syntax

```
void WINAPI glTexCoordPointer(
         GLint   size,
         GLenum  type,
         GLsizei stride,
   const GLvoid  *pointer
);
```

Parameters

size The number of coordinates per array element. The value of *size* must be 1, 2, 3, or 4.

type The data type of each texture coordinate in the array using the following symbolic constants: **GL_SHORT**, **GL_INT**, **GL_FLOAT**, and **GL_DOUBLE**.

stride The byte offset between consecutive array elements. When *stride* is zero, the array elements are tightly packed in the array.

pointer A pointer to the first coordinate of the first element in the array.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*type* was not an accepted value.
GL_INVALID_VALUE	*size* was not 1, 2, 3, or 4.
GL_INVALID_VALUE	*stride* was negative.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*type* was not an accepted value.
GL_INVALID_VALUE	*size* was not 1, 2, 3, or 4.

NAME	MEANING
GL_ INV ALI D_V AL UE	*stride* was negative.

Remarks

The **glTexCoordPointer** function specifies the location and data of an array of texture coordinates to use when rendering.The *size* parameter specifies the number of coordinates used for each element of the array.The *type* parameter specifies the data type of each texture coordinate. The *stride* parameter determines the byte offset from one array element to the next, enabling the packing of vertices and attributes in a single array or storage in separate arrays. In some implementations, storing the vertices and attributes in a single array can be more efficient than using separate arrays. For more information, see **glInterleavedArrays**. When a texture coordinate array is specified, size, type, stride, and pointer are saved client-side state.

A texture coordinate array is enabled when you specify the **GL_TEXTURE_COORD_ARRAY** constant with **glEnableClientState**. When enabled, **glDrawArrays**, **glDrawElements**, and **glArrayElement** use the texture coordinate array. By default the texture coordinate array is disabled.

You cannot include **glTexCoordPointer** in display lists.

When you specify a texture coordinate array using **glTexCoordPointer**, the values of all the function's texture coordinate array parameters are saved in a client-side state, and static array elements can be cached. Because the texture coordinate array parameters are client-side state, their values are not saved or restored by **glPushAttrib** and **glPopAttrib**.

Although no error is generated when you call **glTexCoordPointer** within **glBegin** and **glEnd** pairs, the results are undefined.

The following functions retrieve information related to **glTexCoordPointer**:

glIsEnabled with argument **GL_TEXTURE_COORD_ARRAY**

glGet with argument **GL_TEXTURE_COORD_ARRAY_SIZE**

glGet with argument **GL_TEXTURE_COORD_ARRAY_STRIDE**

glGet with argument **GL_TEXTURE_COORD_ARRAY_COUNT**

glGet with argument **GL_TEXTURE_COORD_ARRAY_TYPE**

glGetPointerv with argument **GL_TEXTURE_COORD_ARRAY_POINTER**

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glArrayElement

glColorPointer

glDraw

A
r
r
a
y
s

g
l
D
r
a
w
E
l
e
m
e
n
t
s

g
l
E
d
g
e
F
l
a
g
P
o
i
n
t
e
r

g
l
E
n
a
b
l
e

g
l
G
e

t
P
o
i
n
t
e
r
v

g
l
G
e
t
S
t
r
i
n
g

g
l
I
n
d
e
x
P
o
i
n
t
e
r

g
l
I
s
E
n
a
b
l
e
d

g
l
N
o

r
m
a
l
P
o
i
n
t
e
r

g
l
P
o
p
C
l
i
e
n
t
A
t
t
r
i
b

g
l
P
u
s
h
C
l
i
e
n
t
A
t
t
r
i
b

g
l
T
e

x
C
o
o
r
d

g
l
V
e
r
t
e
x
P
o
i
n
t
e
r

glTexEnv Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions set a texture environment parameter:

- glTexEnvf
- glTexEnvi
- glTexEnvfv
- glTexEnviv

glTexEnvf function

2/22/2020 • 2 minutes to read • Edit Online

The **glTexEnvf** function sets a texture environment parameter.

Syntax

```
void WINAPI glTexEnvf(
   GLenum  target,
   GLenum  pname,
   GLfloat param
);
```

Parameters

target

A texture environment. Must be GL_TEXTURE_ENV.

pname

The symbolic name of a single-valued texture environment parameter. Must be GL_TEXTURE_ENV_MODE.

param

A single symbolic constant, one of GL_MODULATE, GL_DECAL, GL_BLEND, or GL_REPLACE.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *pname* was not one of the accepted defined values, or when *params* should have had a defined constant value (based on the value of *pname*) and did not.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *pname* was not one of the accepted defined values, or when *params* should have had a defined constant value (based on the value of *pname*) and did not.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

A texture environment specifies how texture values are interpreted when a fragment is textured. The *target* parameter must be GL_TEXTURE_ENV. The *pname* parameter is GL_TEXTURE_ENV_MODE. Three texture functions are defined: GL_MODULATE, GL_DECAL, and GL_BLEND.

A texture function acts on the fragment to be textured using the texture image value that applies to the fragment (see **glTexParameter**) and produces an RGBA color for that fragment. The following table shows how the RGBA color is produced for each of the three texture functions that can be chosen. C is a triple of color values (RGB) and A is the associated alpha value. RGBA values extracted from a texture image are in the range [0, 1]. The subscript f refers to the incoming fragment, the subscript t to the texture image, the subscript c to the texture environment color, and subscript v indicates a value produced by the texture function.

A texture image can have up to four components per texture element (see glTexImage1D and glTexImage2D). In a one-component image, Lt indicates that single component. A two-component image uses *L?* and *A?*. A three-component image has only a color value, *C?*. A four-component image has both a color value *C?* and an alpha value *A?*.

NUMBER OF COMPONENTS	GL_MODULATE	GL_DECAL	GL_BLEND
1${REMOVE}$	$C_v = L? C_f$	undefined${REMOVE}$	$C_v = (1 - L?)C_f + L? C_c$
	$A_v = A_f$		$A_v = A_f$
2${REMOVE}$	$C_v = L? C_f$	undefined${REMOVE}$	$C_v = (1 - L?)C_f + L? C_c$
	$A_v = A_f$		$A_v = A_f$
3${REMOVE}$	$C_v = C? C_f$	$C_v = C?$	undefined${REMOVE}$
	$A_v = A_f$	$A_v = A_f$	
4${REMOVE}$	$C_v = C? C_f$	$C_v = (1 - A?)C_f + A? C?$	undefined${REMOVE}$
	$A_v = A? A_f$	$A_v = A_f$	

GL_TEXTURE_ENV_MODE defaults to GL_MODULATE.

The following function retrieves information related to **glTexEnvf**:

glTexGetEnvfv

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

g
l

B
e
g
i
n

g
l
E
n
d

g
l
T
e
x
I
m
a
g
e
1
D

g
l
T
e
x
I
m
a
g
e
2
D

g
l
T
e
x
P
a
r
a
m
e
t
e
r

glTexEnvfv function

2/22/2020 • 3 minutes to read • Edit Online

The **glTexEnvfv** function sets a texture environment parameter.

Syntax

```
void WINAPI glTexEnvfv(
         GLenum  target,
         GLenum  pname,
   const GLfloat *params
);
```

Parameters

target

A texture environment. Must be GL_TEXTURE_ENV.

pname

The symbolic name of a single-valued texture environment parameter. Accepted values are GL_TEXTURE_ENV_MODE and GL_TEXTURE_ENV_COLOR.

params

A pointer to an array of parameters: either a single symbolic constant or an RGBA color.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *pname* was not one of the accepted defined values, or when *params* should have had a defined constant value (based on the value of *pname*) and did not.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *pname* was not one of the accepted defined values, or when *params* should have had a defined constant value (based on the value of *pname*) and did not.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

A texture environment specifies how texture values are interpreted when a fragment is textured. The *target* parameter must be GL_TEXTURE_ENV. The *pname* parameter can be either GL_TEXTURE_ENV_MODE or GL_TEXTURE_ENV_COLOR.

If *pname* is GL_TEXTURE_ENV_MODE, then *params* is (or points to) the symbolic name of a texture function. Three texture functions are defined: GL_MODULATE, GL_DECAL, and GL_BLEND.

A texture function acts on the fragment to be textured using the texture image value that applies to the fragment (see **glTexParameter**) and produces an RGBA color for that fragment. The following table shows how the RGBA color is produced for each of the three texture functions that can be chosen. *C* is a triple of color values (RGB) and *A*

is the associated alpha value. RGBA values extracted from a texture image are in the range [0, 1]. The subscript *f* refers to the incoming fragment, the subscript *t* to the texture image, the subscript *c* to the texture environment color, and subscript *v* indicates a value produced by the texture function.

A texture image can have up to four components per texture element (see **glTexImage1D** and **glTexImage2D**). In a one-component image, Lt indicates that single component. A two-component image uses *L?* and *A?*. A three-component image has only a color value, *C?*. A four-component image has both a color value *C?* and an alpha value *A?*.

NUMBER OF COMPONENTS	GL_MODULATE	GL_DECAL	GL_BLEND
1${REMOVE}$	$C_v = L? C_f$	undefined${REMOVE}$	$C_v = (1 - L?)C_f + L? C_c$
	$A_v = A_f$		$A_v = A_f$
2${REMOVE}$	$C_v = L? C_f$	undefined${REMOVE}$	$C_v = (1 - L?)C_f + L? C_c$
	$A_v = A_f$		$A_v = A_f$
3${REMOVE}$	$C_v = C? C_f$	$C_v = C?$	undefined${REMOVE}$
	$A_v = A_f$	$A_v = A_f$	
4${REMOVE}$	$C_v = C? C_f$	$C_v = (1 - A?)C_f + A? C?$	undefined${REMOVE}$
	$A_v = A? A_f$	$A_v = A_f$	

If pname is GL_TEXTURE_ENV_COLOR, *params* is a pointer to an array that holds an RGBA color consisting of four values. Integer color components are interpreted linearly such that the most positive integer maps to 1.0, and the most negative integer maps to -1.0. The values are clamped to the range [0, 1] when they are specified. C_c takes these four values.

GL_TEXTURE_ENV_MODE defaults to GL_MODULATE and GL_TEXTURE_ENV_COLOR defaults to (0, 0, 0, 0).

The following function retrieves information related to **glTexEnvfv**:

glTexGetEnvfv

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glBegin

glEnd

glTexImage1D

glTexImage2D

glTe

x
P
a
r
a
m
e
t
e
r

glTexEnvi function

2/22/2020 • 2 minutes to read • Edit Online

The **glTexEnvi** function sets a texture environment parameter.

Syntax

```
void WINAPI glTexEnvi(
   GLenum target,
   GLenum pname,
   GLint  param
);
```

Parameters

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

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *pname* was not one of the accepted defined values, or when *params* should have had a defined constant value (based on the value of *pname*) and did not.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *pname* was not one of the accepted defined values, or when *params* should have had a defined constant value (based on the value of *pname*) and did not.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

A texture environment specifies how texture values are interpreted when a fragment is textured. The *target* parameter must be GL_TEXTURE_ENV. The *pname* parameter is GL_TEXTURE_ENV_MODE. Three texture functions are defined: GL_MODULATE, GL_DECAL, and GL_BLEND.

A texture function acts on the fragment to be textured using the texture image value that applies to the fragment (see **glTexParameter**) and produces an RGBA color for that fragment. The following table shows how the RGBA color is produced for each of the three texture functions that can be chosen. C is a triple of color values (RGB) and A is the associated alpha value. RGBA values extracted from a texture image are in the range [0, 1]. The subscript f refers to the incoming fragment, the subscript t to the texture image, the subscript c to the texture environment color, and subscript v indicates a value produced by the texture function.

A texture image can have up to four components per texture element (see glTexImage1D and glTexImage2D). In a one-component image, Lt indicates that single component. A two-component image uses *L?* and *A?*. A three-component image has only a color value, *C?*. A four-component image has both a color value *C?* and an alpha value *A?*.

NUMBER OF COMPONENTS	GL_MODULATE	GL_DECAL	GL_BLEND
1${REMOVE}$	$C_v = L? C_f$	undefined${REMOVE}$	$C_v = (1 - L?)C_f + L? C_c$
	$A_v = A_f$		$A_v = A_f$
2${REMOVE}$	$C_v = L? C_f$	undefined${REMOVE}$	$C_v = (1 - L?)C_f + L? C_c$
	$A_v = A_f$		$A_v = A_f$
3${REMOVE}$	$C_v = C? C_f$	$C_v = C?$	undefined${REMOVE}$
	$A_v = A_f$	$A_v = A_f$	
4${REMOVE}$	$C_v = C? C_f$	$C_v = (1 - A?)C_f + A? C?$	undefined${REMOVE}$
	$A_v = A? A_f$	$A_v = A_f$	

GL_TEXTURE_ENV_MODE defaults to GL_MODULATE.

The following function retrieves information related to **glTexEnvi**:

glTexGetEnviv

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

g
l

B
e
g
i
n

g
l
E
n
d

g
l
T
e
x
I
m
a
g
e
1
D

g
l
T
e
x
I
m
a
g
e
2
D

g
l
T
e
x
P
a
r
a
m
e
t
e
r

glTexEnviv function

2/22/2020 • 3 minutes to read • Edit Online

The **glTexEnviv** function sets a texture environment parameter.

Syntax

```
void WINAPI glTexEnviv(
         GLenum target,
         GLenum pname,
   const GLint  *params
);
```

Parameters

target
A texture environment. Must be GL_TEXTURE_ENV.

pname
The symbolic name of a single-valued texture environment parameter. Accepted values are GL_TEXTURE_ENV_MODE and GL_TEXTURE_ENV_COLOR.

params
A pointer to an array of parameters: either a single symbolic constant or an RGBA color.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *pname* was not one of the accepted defined values, or when *params* should have had a defined constant value (based on the value of *pname*) and did not.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *pname* was not one of the accepted defined values, or when *params* should have had a defined constant value (based on the value of *pname*) and did not.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

A texture environment specifies how texture values are interpreted when a fragment is textured. The *target* parameter must be GL_TEXTURE_ENV. The *pname* parameter can be either GL_TEXTURE_ENV_MODE or GL_TEXTURE_ENV_COLOR.

If *pname* is GL_TEXTURE_ENV_MODE, then *params* is (or points to) the symbolic name of a texture function. Three texture functions are defined: GL_MODULATE, GL_DECAL, and GL_BLEND.

A texture function acts on the fragment to be textured using the texture image value that applies to the fragment (see **glTexParameter**) and produces an RGBA color for that fragment. The following table shows how the RGBA color is produced for each of the three texture functions that can be chosen. *C* is a triple of color values (RGB) and *A*

is the associated alpha value. RGBA values extracted from a texture image are in the range [0, 1]. The subscript *f* refers to the incoming fragment, the subscript *t* to the texture image, the subscript *c* to the texture environment color, and subscript *v* indicates a value produced by the texture function.

A texture image can have up to four components per texture element (see **glTexImage1D** and **glTexImage2D**). In a one-component image, Lt indicates that single component. A two-component image uses *L?* and *A?*. A three-component image has only a color value, *C?*. A four-component image has both a color value *C?* and an alpha value *A?*.

NUMBER OF COMPONENTS	GL_MODULATE	GL_DECAL	GL_BLEND
1${REMOVE}$	$C_v = L? C_f$	undefined${REMOVE}$	$C_v = (1 - L?)C_f + L? C_c$
	$A_v = A_f$		$A_v = A_f$
2${REMOVE}$	$C_v = L? C_f$	undefined${REMOVE}$	$C_v = (1 - L?)C_f + L? C_c$
	$A_v = A_f$		$A_v = A_f$
3${REMOVE}$	$C_v = C? C_f$	$C_v = C?$	undefined${REMOVE}$
	$A_v = A_f$	$A_v = A_f$	
4${REMOVE}$	$C_v = C? C_f$	$C_v = (1 - A?)C_f + A? C?$	undefined${REMOVE}$
	$A_v = A? A_f$	$A_v = A_f$	

If pname is GL_TEXTURE_ENV_COLOR, *params* is a pointer to an array that holds an RGBA color consisting of four values. Integer color components are interpreted linearly such that the most positive integer maps to 1.0, and the most negative integer maps to -1.0. The values are clamped to the range [0, 1] when they are specified. C_c takes these four values.

GL_TEXTURE_ENV_MODE defaults to GL_MODULATE and GL_TEXTURE_ENV_COLOR defaults to (0, 0, 0, 0).

The following function retrieves information related to **glTexEnviv**:

glTexGetEnviv

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glBegin

glEnd

glTexImage1D

glTexImage2D

glTe

x
P
a
r
a
m
e
t
e
r

glTexGen Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions control the generation of texture coordinates:

- glTexGend
- glTexGenf
- glTexGeni
- glTexGendv
- glTexGenfv
- glTexGeniv

glTexGend function

2/22/2020 • 3 minutes to read • Edit Online

Controls the generation of texture coordinates.

Syntax

```
void WINAPI glTexGend(
   GLenum   coord,
   GLenum   pname,
   GLdouble param
);
```

Parameters

coord A texture coordinate. Must be one of the following: GL_S, GL_T, GL_R, or GL_Q.

pname The symbolic name of the texture coordinate generation function.

param

A single valued texture generation parameter, one of GL_OBJECT_LINEAR, GL_EYE_LINEAR, or GL_SPHERE_MAP.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*coord* or *pname* was not an accepted defined value, or *pname* was GL_TEXTURE_GEN_MODE and *params* was not an accepted defined value.
GL_INVALID_ENUM	*pname* was GL_TEXTURE_GEN_MODE, *params* was GL_SPHERE_MAP, and *coord* was either GL_R or GL_Q
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*coord* or *pname* was not an accepted defined value, or *pname* was GL_TEXTURE_GEN_MODE and *params* was not an accepted defined value.
GL_INVALID_ENUM	*pname* was GL_TEXTURE_GEN_MODE, *params* was GL_SPHERE_MAP, and *coord* was either GL_R or GL_Q

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glTexGen** function selects a texture-coordinate generation function or supplies coefficients for one of the functions. The *coord* parameter names one of the (s,t,r,q) texture coordinates, and it must be one of these symbols: GL_S, GL_T, GL_R, or GL_Q. The *pname* parameter must be one of three symbolic constants: GL_TEXTURE_GEN_MODE, GL_OBJECT_PLANE, or GL_EYE_PLANE. If *pname* is GL_TEXTURE_GEN_MODE, then *param* specifies a mode, one of GL_OBJECT_LINEAR, GL_EYE_LINEAR, or GL_SPHERE_MAP. If *pname* is either GL_OBJECT_PLANE or GL_EYE_PLANE, *param* 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; p1, p2, p3, and p4 are the four values supplied in params; and x?, y?, z?, and w? are the object coordinates of the vertex. You can use this function to texture-map terrain by using sea level as a reference plane (defined by p1, p2, p3, and p4). The GL_OBJECT_LINEAR coordinate generation function computes the altitude of a terrain vertex as its distance from sea level; that altitude is used to index the texture image to map white snow onto peaks and green grass onto foothills, for example.

If the texture generation function is GL_EYE_LINEAR, the function

$$g = p_1 x_o + p_2 y_o + p_3 z_o + p_4 w_o$$

is used, where

$$g = p_{1'} x_e + p_{2'} y_e + p_{3'} z_e + p_{4'} w_e$$

and x?, y?, z?, and w? are the eye coordinates of the vertex, p1, p2, p3, and p4 are the values supplied in *param*, and M is the modelview matrix when you call **glTexGen**. If M is poorly conditioned or singular, texture coordinates generated by the resulting function can be inaccurate or undefined.

The values in *param* define a reference plane in eye coordinates. The modelview matrix that is applied to them cannot be the same one in effect when the polygon vertices are transformed. This function establishes a field of texture coordinates that can produce dynamic contour lines on moving objects.

If *pname* is GL_SPHERE_MAP and *coord* is either GL_S or GL_T, s and t texture coordinates are generated as follows. Let u be the unit vector pointing from the origin to the polygon vertex (in eye coordinates). Let n be the current normal, after transformation to eye coordinates. Let f = (fx ( ) fy ( ) fz)T be the reflection vector such that

$$\mathbf{f} = \mathbf{u} - 2\mathbf{n}' \mathbf{n}'^T \mathbf{u}$$

Finally, let

$$m = 2\sqrt{f_x^2 + f_y^2 + (f_z + 1)^2}$$

Then the values assigned to the i and t texture coordinates are

$$s = \frac{f_x}{m} + \frac{1}{2}$$

$$t = \frac{f_y}{m} + \frac{1}{2}$$

You can enable or disable a texture-coordinate generation function by using **glEnable** or **glDisable** with one of the symbolic texture-coordinate names (GL_TEXTURE_GEN_S, GL_TEXTURE_GEN_T, GL_TEXTURE_GEN_R, or GL_TEXTURE_GEN_Q) as the argument. When this function is enabled, the specified texture coordinate is computed according to the generating function associated with that coordinate. When the function is disabled, subsequent vertices take the specified texture coordinate from the current set of texture coordinates. Initially, all texture generation functions are set to GL_EYE_LINEAR and are disabled. Both s plane equations are (1,0,0,0); both t plane equations are (0,1,0,0); and all r and q plane equations are (0,0,0,0).

The following functions retrieve information related to glTexGen:

glGetTexGen
glIsEnabled with argument GL_TEXTURE_GEN_S
glIsEnabled with argument GL_TEXTURE_GEN_T
glIsEnabled with argument GL_TEXTURE_GEN_R
glIsEnabled with argument GL_TEXTURE_GEN_Q

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegi

n

glEnd

glCopyTexImage2D

glCopyTexSubImage2D

glGetTex

G
e
n

g
l
I
s
E
n
a
b
l
e
d

g
l
T
e
x
E
n
v

g
l
T
e
x
I
m
a
g
e
1
D

g
l
T
e
x
P
a
r
a
m
e
t
e
r

glTexSubImage1D

glTexSubImage2D

glTexGendv function

2/22/2020 • 3 minutes to read • Edit Online

Controls the generation of texture coordinates.

Syntax

```
void WINAPI glTexGendv(
         GLenum   coord,
         GLenum   pname,
   const GLdouble *params
);
```

Parameters

coord	A texture coordinate. Must be one of the following: GL_S, GL_T, GL_R, or GL_Q.
pname	The symbolic name of the texture coordinate generation function.
params	An array that contains the coefficients for the corresponding texture generation function.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*coord* or *pname* was not an accepted defined value, or *pname* was GL_TEXTURE_GEN_MODE and *params* was not an accepted defined value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*coord* or *pname* was not an accepted defined value, or *pname* was GL_TEXTURE_GEN_MODE and *params* was not an accepted defined value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glTexGen** function selects a texture-coordinate generation function or supplies coefficients for one of the functions. The *coord* parameter names one of the (s,t,r,q) texture coordinates, and it must be one of these symbols: GL_S, GL_T, GL_R, or GL_Q. The *pname* parameter must be one of three symbolic constants: GL_TEXTURE_GEN_MODE, GL_OBJECT_PLANE, or GL_EYE_PLANE. If *pname* is either GL_OBJECT_PLANE or GL_EYE_PLANE, *param* 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; p1, p2, p3, and p4 are the four values supplied in params; and x?, y?, z?, and w? are the object coordinates of the vertex. You can use this function to texture-map terrain by using sea level as a reference plane (defined by p1, p2, p3, and p4). The GL_OBJECT_LINEAR coordinate generation function computes the altitude of a terrain vertex as its distance from sea level; that altitude is used to index the texture image to map white snow onto peaks and green grass onto foothills, for example.

If the texture generation function is GL_EYE_LINEAR, the function

$$g = p_1 x_o + p_2 y_o + p_3 z_o + p_4 w_o$$

is used, where

$$g = p_{1'} x_e + p_{2'} y_e + p_{3'} z_e + p_{4'} w_e$$

and x?, y?, z?, and w? are the eye coordinates of the vertex, p1, p2, p3, and p4 are the values supplied in *param*, and M is the modelview matrix when you **callglTexGen**. If M is poorly conditioned or singular, texture coordinates generated by the resulting function can be inaccurate or undefined.

Note that the values in *param* define a reference plane in eye coordinates. The modelview matrix that is applied to them may not be the same one in effect when the polygon vertices are transformed. This function establishes a field of texture coordinates that can produce dynamic contour lines on moving objects.

If *pname* is GL_SPHERE_MAP and *coord* is either GL_S or GL_T, s and t texture coordinates are generated as follows. Let u be the unit vector pointing from the origin to the polygon vertex (in eye coordinates). Let n be the current normal, after transformation to eye coordinates. Let f = (fx ( ) fy ( ) fz)T be the reflection vector such that

$$\mathbf{f} = \mathbf{u} - 2\mathbf{n}' \mathbf{n}'^T \mathbf{u}$$

Finally, let

$$m = 2\sqrt{f_x^2 + f_y^2 + (f_z + 1)^2}$$

Then the values assigned to the i and t texture coordinates are

$$s = \frac{f_x}{m} + \frac{1}{2}$$

$$t = \frac{f_y}{m} + \frac{1}{2}$$

You can enable or disable a texture-coordinate generation function by using **glEnable** or **glDisable** with one of the symbolic texture-coordinate names (GL_TEXTURE_GEN_S, GL_TEXTURE_GEN_T, GL_TEXTURE_GEN_R, or GL_TEXTURE_GEN_Q) as the argument. When this function is enabled, the specified texture coordinate is computed according to the generating function associated with that coordinate. When this function is disabled, subsequent vertices take the specified texture coordinate from the current set of texture coordinates. Initially, all texture generation functions are set to GL_EYE_LINEAR and are disabled. Both s plane equations are (1,0,0,0); both t plane equations are (0,1,0,0); and all r and q plane equations are (0,0,0,0).

The following functions retrieve information related to glTexGen:

glGetTexGen
glIsEnabled with argument GL_TEXTURE_GEN_S
glIsEnabled with argument GL_TEXTURE_GEN_T
glIsEnabled with argument GL_TEXTURE_GEN_R
glIsEnabled with argument GL_TEXTURE_GEN_Q

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glCopyTexImage2D

glCopyTexSubImage2D

glGetTexGen

glIsEnabled

glTexEnv

glTexImage1D

glTexParameter

glTexSubImage1D

glTexSubI

m
a
g
e
2
D

glTexGenf function

2/22/2020 • 3 minutes to read • Edit Online

Controls the generation of texture coordinates.

Syntax

```
void WINAPI glTexGenf(
   GLenum  coord,
   GLenum  pname,
   GLfloat param
);
```

Parameters

coord A texture coordinate. Must be one of the following: GL_S, GL_T, GL_R, or GL_Q.

pname The symbolic name of the texture coordinate generation function.

param

A single valued texture generation parameter, one of GL_OBJECT_LINEAR, GL_EYE_LINEAR, or GL_SPHERE_MAP.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*coord* or *pname* was not an accepted defined value, or *pname* was GL_TEXTURE_GEN_MODE and *params* was not an accepted defined value.
GL_INVALID_ENUM	*pname* was GL_TEXTURE_GEN_MODE, *params* was GL_SPHERE_MAP, and *coord* was either GL_R or GL_Q
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*coord* or *pname* was not an accepted defined value, or *pname* was GL_TEXTURE_GEN_MODE and *params* was not an accepted defined value.
GL_INVALID_ENUM	*pname* was GL_TEXTURE_GEN_MODE, *params* was GL_SPHERE_MAP, and *coord* was either GL_R or GL_Q

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glTexGen** function selects a texture-coordinate generation function or supplies coefficients for one of the functions. The *coord* parameter names one of the (s,t,r,q) texture coordinates, and it must be one of these symbols: GL_S, GL_T, GL_R, or GL_Q. The *pname* parameter must be one of three symbolic constants: GL_TEXTURE_GEN_MODE, GL_OBJECT_PLANE, or GL_EYE_PLANE. If *pname* is GL_TEXTURE_GEN_MODE, then *param* specifies a mode, one of GL_OBJECT_LINEAR, GL_EYE_LINEAR, or GL_SPHERE_MAP. If *pname* is either GL_OBJECT_PLANE or GL_EYE_PLANE, *param* 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; p1, p2, p3, and p4 are the four values supplied in params; and x?, y?, z?, and w? are the object coordinates of the vertex. You can use this function to texture-map terrain by using sea level as a reference plane (defined by p1, p2, p3, and p4). The GL_OBJECT_LINEAR coordinate generation function computes the altitude of a terrain vertex as its distance from sea level; that altitude is used to index the texture image to map white snow onto peaks and green grass onto foothills, for example.

If the texture generation function is GL_EYE_LINEAR, the function

$$g = p_1 x_o + p_2 y_o + p_3 z_o + p_4 w_o$$

is used, where

$$g = p_{1'} x_e + p_{2'} y_e + p_{3'} z_e + p_{4'} w_e$$

and x?, y?, z?, and w? are the eye coordinates of the vertex, p1, p2, p3, and p4 are the values supplied in *param*, and M is the modelview matrix when you call **glTexGen**. If M is poorly conditioned or singular, texture coordinates generated by the resulting function can be inaccurate or undefined.

Note that the values in *param* define a reference plane in eye coordinates. The modelview matrix that is applied to them may not be the same one in effect when the polygon vertices are transformed. This function establishes a field of texture coordinates that can produce dynamic contour lines on moving objects.

If *pname* is GL_SPHERE_MAP and *coord* is either GL_S or GL_T, s and t texture coordinates are generated as follows. Let u be the unit vector pointing from the origin to the polygon vertex (in eye coordinates). Let n be the current normal, after transformation to eye coordinates. Let f = (fx ( ) fy ( ) fz)T be the reflection vector such that

$$\mathbf{f} = \mathbf{u} - 2\mathbf{n}' \mathbf{n}'^T \mathbf{u}$$

Finally, let

$$m = 2\sqrt{f_x^2 + f_y^2 + (f_z + 1)^2}$$

Then the values assigned to the i and t texture coordinates are

$$s = \frac{f_x}{m} + \frac{1}{2}$$

$$t = \frac{f_y}{m} + \frac{1}{2}$$

You can enable or disable a texture-coordinate generation function by using **glEnable** or **glDisable** with one of the symbolic texture-coordinate names (GL_TEXTURE_GEN_S, GL_TEXTURE_GEN_T, GL_TEXTURE_GEN_R, or GL_TEXTURE_GEN_Q) as the argument. When this function is enabled, the specified texture coordinate is computed according to the generating function associated with that coordinate. When this function is disabled, subsequent vertices take the specified texture coordinate from the current set of texture coordinates. Initially, all texture generation functions are set to GL_EYE_LINEAR and are disabled. Both s plane equations are (1,0,0,0); both t plane equations are (0,1,0,0); and all r and q plane equations are (0,0,0,0).

The following functions retrieve information related to glTexGen:

glGetTexGen
glIsEnabled with argument GL_TEXTURE_GEN_S
glIsEnabled with argument GL_TEXTURE_GEN_T
glIsEnabled with argument GL_TEXTURE_GEN_R
glIsEnabled with argument GL_TEXTURE_GEN_Q

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegi

n

glEnd

glCopyTexImage2D

glCopyTexSubImage2D

glGetTex

GenglIsEnabled

glTexEnv

glTexImage1D

glTexParameter

glTexSubImage1D

glTexSubImage2D

glTexGenfv function

2/22/2020 • 3 minutes to read • Edit Online

Controls the generation of texture coordinates.

Syntax

```
void WINAPI glTexGenfv(
         GLenum  coord,
         GLenum  pname,
   const GLfloat *params
);
```

Parameters

coord A texture coordinate. Must be one of the following: GL_S, GL_T, GL_R, or GL_Q.

pname The symbolic name of the texture coordinate generation function.

params An array of floats that contains the coefficients for the corresponding texture generation function.

```
GLfloat zPlane[] = { 0.0f, 0.0f, 1.0f, 0.0f };
```

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*coord* or *pname* was not an accepted defined value, or *pname* was GL_TEXTURE_GEN_MODE and *params* was not an accepted defined value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*coord* or *pname* was not an accepted defined value, or *pname* was GL_TEXTURE_GEN_MODE and *params* was not an accepted defined value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glTexGen** function selects a texture-coordinate generation function or supplies coefficients for one of the functions. The *coord* parameter names one of the (s,t,r,q) texture coordinates, and it must be one of these symbols: GL_S, GL_T, GL_R, or GL_Q. The *pname* parameter must be one of three symbolic constants: GL_TEXTURE_GEN_MODE, GL_OBJECT_PLANE, or GL_EYE_PLANE. If *pname* is either GL_OBJECT_PLANE or GL_EYE_PLANE, *param* 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; p1, p2, p3, and p4 are the four values supplied in params; and x?, y?, z?, and w? are the object coordinates of the vertex. You can use this function to texture-map terrain by using sea level as a reference plane (defined by p1, p2, p3, and p4). The GL_OBJECT_LINEAR coordinate generation function computes the altitude of a terrain vertex as its distance from sea level; that altitude is used to index the texture image to map white snow onto peaks and green grass onto foothills, for example.

If the texture generation function is GL_EYE_LINEAR, the function

$$g = p_1 x_o + p_2 y_o + p_3 z_o + p_4 w_o$$

is used, where

$$g = p_{1'} x_e + p_{2'} y_e + p_{3'} z_e + p_{4'} w_e$$

and x?, y?, z?, and w? are the eye coordinates of the vertex, p1, p2, p3, and p4 are the values supplied in *param*, and M is the modelview matrix when you call **glTexGen**. If M is poorly conditioned or singular, texture coordinates generated by the resulting function can be inaccurate or undefined.

Note that the values in *param* define a reference plane in eye coordinates. The modelview matrix that is applied to them may not be the same one in effect when the polygon vertices are transformed. This function establishes a field of texture coordinates that can produce dynamic contour lines on moving objects.

If *pname* is GL_SPHERE_MAP and *coord* is either GL_S or GL_T, s and t texture coordinates are generated as follows. Let u be the unit vector pointing from the origin to the polygon vertex (in eye coordinates). Let n be the current normal, after transformation to eye coordinates. Let f = (fx ( ) fy ( ) fz)T be the reflection vector such that

$$\mathbf{f} = \mathbf{u} - 2\mathbf{n}' \mathbf{n}'^T \mathbf{u}$$

Finally, let

$$m = 2\sqrt{f_x^2 + f_y^2 + (f_z + 1)^2}$$

Then the values assigned to the i and t texture coordinates are

$$s = \frac{f_x}{m} + \frac{1}{2}$$

$$t = \frac{f_y}{m} + \frac{1}{2}$$

You can enable or disable a texture-coordinate generation function by using **glEnable** or **glDisable** with one of the symbolic texture-coordinate names (GL_TEXTURE_GEN_S, GL_TEXTURE_GEN_T, GL_TEXTURE_GEN_R, or GL_TEXTURE_GEN_Q) as the argument. When this function is enabled, the specified texture coordinate is computed according to the generating function associated with that coordinate. When this function is disabled, subsequent vertices take the specified texture coordinate from the current set of texture coordinates. Initially, all texture generation functions are set to GL_EYE_LINEAR and are disabled. Both s plane equations are (1,0,0,0); both t plane equations are (0,1,0,0); and all r and q plane equations are (0,0,0,0).

The following functions retrieve information related to glTexGen:

glGetTexGen
glIsEnabled with argument GL_TEXTURE_GEN_S
glIsEnabled with argument GL_TEXTURE_GEN_T
glIsEnabled with argument GL_TEXTURE_GEN_R
glIsEnabled with argument GL_TEXTURE_GEN_Q

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glCopyTexImage2D

glCopyTexSubImage2D

glGetTexGen

glIsEnabled

glTexEnv

glTexImage1D

glTexParameter

glTexSubImage1D

glTexSubI

m
a
g
e
2
D

glTexGeni function

2/22/2020 • 3 minutes to read • Edit Online

Controls the generation of texture coordinates.

Syntax

```
void WINAPI glTexGeni(
   GLenum coord,
   GLenum pname,
   GLint  param
);
```

Parameters

coord A texture coordinate. Must be one of the following: GL_S, GL_T, GL_R, or GL_Q.

pname The symbolic name of the texture coordinate generation function.

param

A single valued texture generation parameter, one of GL_OBJECT_LINEAR, GL_EYE_LINEAR, or GL_SPHERE_MAP.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*coord* or *pname* was not an accepted defined value, or *pname* was GL_TEXTURE_GEN_MODE and *params* was not an accepted defined value.
GL_INVALID_ENUM	*pname* was GL_TEXTURE_GEN_MODE, *params* was GL_SPHERE_MAP, and *coord* was either GL_R or GL_Q
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*coord* or *pname* was not an accepted defined value, or *pname* was GL_TEXTURE_GEN_MODE and *params* was not an accepted defined value.
GL_INVALID_ENUM	*pname* was GL_TEXTURE_GEN_MODE, *params* was GL_SPHERE_MAP, and *coord* was either GL_R or GL_Q

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glTexGen** function selects a texture-coordinate generation function or supplies coefficients for one of the functions. The *coord* parameter names one of the (s,t,r,q) texture coordinates, and it must be one of these symbols: GL_S, GL_T, GL_R, or GL_Q. The *pname* parameter must be one of three symbolic constants: GL_TEXTURE_GEN_MODE, GL_OBJECT_PLANE, or GL_EYE_PLANE. If *pname* is GL_TEXTURE_GEN_MODE, then *param* specifies a mode, one of GL_OBJECT_LINEAR, GL_EYE_LINEAR, or GL_SPHERE_MAP. If *pname* is either GL_OBJECT_PLANE or GL_EYE_PLANE, *param* 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; p1, p2, p3, and p4 are the four values supplied in params; and x?, y?, z?, and w? are the object coordinates of the vertex. You can use this function to texture-map terrain by using sea level as a reference plane (defined by p1, p2, p3, and p4). The GL_OBJECT_LINEAR coordinate generation function computes the altitude of a terrain vertex as its distance from sea level; that altitude is used to index the texture image to map white snow onto peaks and green grass onto foothills, for example.

If the texture generation function is GL_EYE_LINEAR, the function

$$g = p_1 x_o + p_2 y_o + p_3 z_o + p_4 w_o$$

is used, where

$$g = p_{1'} x_e + p_{2'} y_e + p_{3'} z_e + p_{4'} w_e$$

and x?, y?, z?, and w? are the eye coordinates of the vertex, p1, p2, p3, and p4 are the values supplied in *param*, and M is the modelview matrix when you call **glTexGen**. If M is poorly conditioned or singular, texture coordinates generated by the resulting function can be inaccurate or undefined.

Note that the values in *param* define a reference plane in eye coordinates. The modelview matrix that is applied to them may not be the same one in effect when the polygon vertices are transformed. This function establishes a field of texture coordinates that can produce dynamic contour lines on moving objects.

If *pname* is GL_SPHERE_MAP and *coord* is either GL_S or GL_T, s and t texture coordinates are generated as follows. Let u be the unit vector pointing from the origin to the polygon vertex (in eye coordinates). Let n be the current normal, after transformation to eye coordinates. Let f = (fx ( ) fy ( ) fz)T be the reflection vector such that

$$\mathbf{f} = \mathbf{u} - 2\mathbf{n}' \mathbf{n}'^T \mathbf{u}$$

Finally, let

$$m = 2\sqrt{f_x^2 + f_y^2 + (f_z + 1)^2}$$

Then the values assigned to the i and t texture coordinates are

$$s = \frac{f_x}{m} + \frac{1}{2}$$

$$t = \frac{f_y}{m} + \frac{1}{2}$$

You can enable or disable a texture-coordinate generation function by using **glEnable** or **glDisable** with one of the symbolic texture-coordinate names (GL_TEXTURE_GEN_S, GL_TEXTURE_GEN_T, GL_TEXTURE_GEN_R, or GL_TEXTURE_GEN_Q) as the argument. When this function is enabled, the specified texture coordinate is computed according to the generating function associated with that coordinate. When this function is disabled, subsequent vertices take the specified texture coordinate from the current set of texture coordinates. Initially, all texture generation functions are set to GL_EYE_LINEAR and are disabled. Both s plane equations are (1,0,0,0); both t plane equations are (0,1,0,0); and all r and q plane equations are (0,0,0,0).

The following functions retrieve information related to glTexGen:

glGetTexGen
glIsEnabled with argument GL_TEXTURE_GEN_S
glIsEnabled with argument GL_TEXTURE_GEN_T
glIsEnabled with argument GL_TEXTURE_GEN_R
glIsEnabled with argument GL_TEXTURE_GEN_Q

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegi

n

glEnd

glCopyTexImage2D

glCopyTexSubImage2D

glGetTex

GenglIsEnabled

glTexEnv

glTexImage1D

glTexParameter

glTexSubImage1D

glTexSubImage2D

glTexGeniv function

2/22/2020 • 3 minutes to read • Edit Online

Controls the generation of texture coordinates.

Syntax

```
void WINAPI glTexGeniv(
         GLenum coord,
         GLenum pname,
   const GLint  *params
);
```

Parameters

coord	A texture coordinate. Must be one of the following: GL_S, GL_T, GL_R, or GL_Q.
pname	The symbolic name of the texture coordinate generation function.
params	An array that contains the coefficients for the corresponding texture generation function.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*coord* or *pname* was not an accepted defined value, or *pname* was GL_TEXTURE_GEN_MODE and *params* was not an accepted defined value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*coord* or *pname* was not an accepted defined value, or *pname* was GL_TEXTURE_GEN_MODE and *params* was not an accepted defined value.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glTexGen** function selects a texture-coordinate generation function or supplies coefficients for one of the functions. The *coord* parameter names one of the (s,t,r,q) texture coordinates, and it must be one of these symbols: GL_S, GL_T, GL_R, or GL_Q. The *pname* parameter must be one of three symbolic constants: GL_TEXTURE_GEN_MODE, GL_OBJECT_PLANE, or GL_EYE_PLANE. If *pname* is either GL_OBJECT_PLANE or GL_EYE_PLANE, *param* 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; p1, p2, p3, and p4 are the four values supplied in params; and x?, y?, z?, and w? are the object coordinates of the vertex. You can use this function to texture-map terrain by using sea level as a reference plane (defined by p1, p2, p3, and p4). The GL_OBJECT_LINEAR coordinate generation function computes the altitude of a terrain vertex as its distance from sea level; that altitude is used to index the texture image to map white snow onto peaks and green grass onto foothills, for example.

If the texture generation function is GL_EYE_LINEAR, the function

$$g = p_1 x_o + p_2 y_o + p_3 z_o + p_4 w_o$$

is used, where

$$g = p_{1'} x_e + p_{2'} y_e + p_{3'} z_e + p_{4'} w_e$$

and x?, y?, z?, and w? are the eye coordinates of the vertex, p1, p2, p3, and p4 are the values supplied in *param*, and M is the modelview matrix when you call **glTexGen**. If M is poorly conditioned or singular, texture coordinates generated by the resulting function can be inaccurate or undefined.

Note that the values in *param* define a reference plane in eye coordinates. The modelview matrix that is applied to them may not be the same one in effect when the polygon vertices are transformed. This function establishes a field of texture coordinates that can produce dynamic contour lines on moving objects.

If *pname* is GL_SPHERE_MAP and *coord* is either GL_S or GL_T, s and t texture coordinates are generated as follows. Let u be the unit vector pointing from the origin to the polygon vertex (in eye coordinates). Let n be the current normal, after transformation to eye coordinates. Let f = (fx ( ) fy ( ) fz)T be the reflection vector such that

$$\mathbf{f} = \mathbf{u} - 2\mathbf{n}' \mathbf{n}'^T \mathbf{u}$$

Finally, let

$$m = 2\sqrt{f_x^2 + f_y^2 + (f_z + 1)^2}$$

Then the values assigned to the i and t texture coordinates are

$$s = \frac{f_x}{m} + \frac{1}{2}$$

$$t = \frac{f_y}{m} + \frac{1}{2}$$

You can enable or disable a texture-coordinate generation function by using **glEnable** or **glDisable** with one of the symbolic texture-coordinate names (GL_TEXTURE_GEN_S, GL_TEXTURE_GEN_T, GL_TEXTURE_GEN_R, or GL_TEXTURE_GEN_Q) as the argument. When this function is enabled, the specified texture coordinate is computed according to the generating function associated with that coordinate. When this function is disabled, subsequent vertices take the specified texture coordinate from the current set of texture coordinates. Initially, all texture generation functions are set to GL_EYE_LINEAR and are disabled. Both s plane equations are (1,0,0,0); both t plane equations are (0,1,0,0); and all r and q plane equations are (0,0,0,0).

The following functions retrieve information related to glTexGen:

glGetTexGen
glIsEnabled with argument GL_TEXTURE_GEN_S
glIsEnabled with argument GL_TEXTURE_GEN_T
glIsEnabled with argument GL_TEXTURE_GEN_R
glIsEnabled with argument GL_TEXTURE_GEN_Q

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glCopyTexImage2D

glCopyTexSubImage2D

glGetTexGen

glIsEnabled

glTexEnv

glTexImage1D

glTexParameter

glTexSubImage1D

glTexSubI

m
a
g
e
2
D

glTexImage1D function

2/22/2020 • 7 minutes to read • Edit Online

The **glTexImage1D** function specifies a one-dimensional texture image.

Syntax

```
void WINAPI glTexImage1D(
         GLenum  target,
         GLint   level,
         GLint   internalformat,
         GLsizei width,
         GLint   border,
         GLint   format,
         GLenum  type,
   const GLvoid  *pixels
);
```

Parameters

target

The target texture. Must be GL_TEXTURE_1D.

level

The level-of-detail number. Level 0 is the base image level. Level *n* is the *n*Th mipmap reduction image.

internalformat

Specifies the number of color components in the texture. Must be 1, 2, 3, or 4, or one of the following symbolic constants: GL_ALPHA, GL_ALPHA4, GL_ALPHA8, GL_ALPHA12, GL_ALPHA16, GL_LUMINANCE, GL_LUMINANCE4, GL_LUMINANCE8, GL_LUMINANCE12, GL_LUMINANCE16, GL_LUMINANCE_ALPHA, GL_LUMINANCE4_ALPHA4, GL_LUMINANCE6_ALPHA2, GL_LUMINANCE8_ALPHA8, GL_LUMINANCE12_ALPHA4, GL_LUMINANCE12_ALPHA12, GL_LUMINANCE16_ALPHA16, GL_INTENSITY, GL_INTENSITY4, GL_INTENSITY8, GL_INTENSITY12, GL_INTENSITY16, GL_RGB, GL_R3_G3_B2, GL_RGB4, GL_RGB5, GL_RGB8, GL_RGB10, GL_RGB12, GL_RGB16, GL_RGBA, GL_RGBA2, GL_RGBA4, GL_RGB5_A1, GL_RGBA8, GL_RGB10_A2, GL_RGBA12, or GL_RGBA16.

width

The width of the texture image. Must be $2n + 2(\mathit{border})$ for some integer n. The height of the texture image is 1.

border

The width of the border. Must be either 0 or 1.

format

The format of the pixel data. It can assume one of nine symbolic values.

VALUE	MEANING
GL_COLOR_INDEX	Each element is a single value, a color index. It is converted to fixed point (with an unspecified number of 0 bits to the right of the binary point), shifted left or right, depending on the value and sign of GL_INDEX_SHIFT, and added to GL_INDEX_OFFSET (see glPixelTransfer). The resulting index is converted to a set of color components using the GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, and GL_PIXEL_MAP_I_TO_A tables, and clamped to the range [0,1].

VALUE	MEANING
GL_RED	Each element is a single red component. It is converted to floating point and assembled into an RGBA element by attaching 0.0 for green and blue, and 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).
GL_GREEN	Each element is a single green component. It is converted to floating point and assembled into an RGBA element by attaching 0.0 for red and blue, and 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).
GL_BLUE	Each element is a single blue component. It is converted to floating point and assembled into an RGBA element by attaching 0.0 for red and green, and 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).
GL_ALPHA	Each element is a single red component. It is converted to floating point and assembled into an RGBA element by attaching 0.0 for red, green, and blue. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).
GL_RGB	Each element is an RGB triple. It is converted to floating point and assembled into an RGBA element by attaching 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

VALUE	MEANING
GL_RGBA	Each element is a complete RGBA element. It is converted to floating point. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).
GL_BGR_EXT	Each pixel is a group of three components in this order: blue, green, red. GL_BGR_EXT provides a format that matches the memory layout of Windows device-independent bitmaps (DIBs). Thus your applications can use the same data with Windows function calls and OpenGL pixel function calls.
GL_BGRA_EXT	Each pixel is a group of four components in this order: blue, green, red, alpha. GL_BGRA_EXT provides a format that matches the memory layout of Windows device-independent bitmaps (DIBs). Thus your applications can use the same data with Windows function calls and OpenGL pixel function calls.
GL_LUMINANCE	Each element is a single luminance value. It is converted to floating point, and then assembled into an RGBA element by replicating the luminance value three times for red, green, and blue, and attaching 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).
GL_LUMINANCE_ALPHA	Each element is a luminance/alpha pair. It is converted to floating point, and then assembled into an RGBA element by replicating the luminance value three times for red, green, and blue. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

type

The data type of the pixel data. The following symbolic values are accepted:

GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, and GL_FLOAT.

pixels A pointer to the image data in memory.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* was not an GL_TEXTURE.
GL_INVALID_ENUM	*format* was not an accepted *format* constant. Only format constants other than GL_STENCIL_INDEX and GL_DEPTH_COMPONENT are accepted. See the parameter description of *format* for a list of possible values.
GL_INVALID_ENUM	*type* was not a *type* constant.
GL_INVALID_ENUM	*type* was GL_BITMAP and *format* was not GL_COLOR_INDEX.

NAME	MEANING
GL_INVALID_VALUE	*level* was less than zero or greater than log2 *max*, where *max* was the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_VALUE	*internalformat* was was not 1, 2, 3, or 4.
GL_INVALID_VALUE	*width* was less than zero or greater than 2 + GL_MAX_TEXTURE_SIZE, or it could not be represented as 2*n* + 2(border) for some integer value of *n*.
GL_INVALID_VALUE	*border* was not 0 or 1.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* was not an GL_TEXTURE.
GL_INVALID_ENUM	*format* was not an accepted *format* constant. Only format constants other than GL_STENCIL_INDEX and GL_DEPTH_COMPONENT are accepted. See the parameter description of *format* for a list of possible values.
GL_INVALID_ENUM	*type* was not a *type* constant.
GL_INVALID_ENUM	*type* was GL_BITMAP and *format* was not GL_COLOR_INDEX.
GL_INVALID_VALUE	*level* was less than zero or greater than log2*max*, where *max* was the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_VALUE	*internalformat* was was not 1, 2, 3, or 4.
GL_INVALID_VALUE	*width* was less than zero or greater than 2 + GL_MAX_TEXTURE_SIZE, or it could not be represented as 2*n* + 2(border) for some integer value of *n*.

NAME	MEANING
GL_INVALID_VALUE	*border* was not 0 or 1.
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Remarks

The **glTexImage1D** function specifies a one-dimensional texture image. Texturing maps a portion of a specified *texture image* onto each graphical primitive for which texturing is enabled. One-dimensional texturing is enabled and disabled using glEnable and **glDisable** with argument GL_TEXTURE_1D.

Texture images are defined with **glTexImage1D**. The arguments describe the parameters of the texture image, such as width, width of the border, level-of-detail number (see glTexParameter), and number of color components provided. The last three arguments describe the way the image is represented in memory. These arguments are identical to the pixel formats used for glDrawPixels.

Data is read from *pixels* as a sequence of signed or unsigned bytes, shorts or longs, or single-precision floating-point values, depending on *type*. These values are grouped into sets of one, two, three, or four values, depending on *format*, to form elements. If *type* is GL_BITMAP, the data is considered as a string of unsigned bytes (and *format* must be GL_COLOR_INDEX). Each data byte is treated as eight 1-bit elements, with bit ordering determined by GL_UNPACK_LSB_FIRST (see glPixelStore).

A texture image can have up to four components per texture element, depending on *components*. A one-component texture image uses only the red component of the RGBA color extracted from *pixels*. A two-component image uses the R and A values. A three-component image uses the R, G, and B values. A four-component image uses all of the RGBA components.

Texturing has no effect in color-index mode.

The texture image can be represented by the same data formats as the pixels in a glDrawPixels command, except that GL_STENCIL_INDEX and GL_DEPTH_COMPONENT cannot be used. The glPixelStore and glPixelTransfer modes affect texture images in exactly the way they affect **glDrawPixels**.

A texture image with zero width indicates the null texture. If the null texture is specified for level-of-detail 0, it is as if texturing were disabled.

The following functions retrieve information related to **glTexImageID**:

glGetTexImage

glIsEnabled with argument GL_TEXTURE_1D

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glCopyPixels

glCop

y
T
e
x
I
m
a
g
e
1
D

g
I
C
o
p
y
T
e
x
I
m
a
g
e
2
D

g
I
C
o
p
y
T
e
x
S
u
b
I
m
a
g
e
1
D

g
I
C
o

pyTexSublmage2D

glDrawPixels

glEnd

glFog

glGetTexImag

e

glIsEnabled

glPixelStore

glPixelTransfer

glTexEnv

g

lTexGen

glTexImage2D

glTexSubImage1D

glTexSubImage2D

glTexParameter

glTexImage2D function

2/22/2020 • 7 minutes to read • Edit Online

The **glTexImage2D** function specifies a two-dimensional texture image.

Syntax

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

Parameters

target

The target texture. Must be GL_TEXTURE_2D.

level

The level-of-detail number. Level 0 is the base image level. Level *n* is the *n* th mipmap reduction image.

internalformat

t

The number of color components in the texture. Must be 1, 2, 3, or 4, or one of the following symbolic constants: GL_ALPHA, GL_ALPHA4, GL_ALPHA8, GL_ALPHA12, GL_ALPHA16, GL_LUMINANCE, GL_LUMINANCE4, GL_LUMINANCE8, GL_LUMINANCE12, GL_LUMINANCE16, GL_LUMINANCE_ALPHA, GL_LUMINANCE4_ALPHA4, GL_LUMINANCE6_ALPHA2, GL_LUMINANCE8_ALPHA8, GL_LUMINANCE12_ALPHA4, GL_LUMINANCE12_ALPHA12, GL_LUMINANCE16_ALPHA16, GL_INTENSITY, GL_INTENSITY4, GL_INTENSITY8, GL_INTENSITY12, GL_INTENSITY16, GL_R3_G3_B2, GL_RGB, GL_RGB4, GL_RGB5, GL_RGB8, GL_RGB10, GL_RGB12, GL_RGB16, GL_RGBA, GL_RGBA2, GL_RGBA4, GL_RGB5_A1, GL_RGBA8, GL_RGB10_A2, GL_RGBA12, or GL_RGBA16.

width The width of the texture image. Must be $2n + 2(border)$ for some integer n.

height The height of the texture image. Must be $2^{*m*} + 2(border)$ for some integer m.

border The width of the border. Must be either 0 or 1.

format The format of the pixel data. It can assume one of nine symbolic values.

VALUE	MEANING
GL_COLOR_INDEX	Each element is a single value, a color index. It is converted to fixed point (with an unspecified number of 0 bits to the right of the binary point), shifted left or right depending on the value and sign of GL_INDEX_SHIFT, and added to GL_INDEX_OFFSET (see glPixelTransfer). The resulting index is converted to a set of color components using the GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, and GL_PIXEL_MAP_I_TO_A tables, and clamped to the range [0,1].

VALUE	MEANING
GL_RED	Each element is a single red component. It is converted to floating point and assembled into an RGBA element by attaching 0.0 for green and blue, and 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).
GL_GREEN	Each element is a single green component. It is converted to floating point and assembled into an RGBA element by attaching 0.0 for red and blue, and 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).
GL_BLUE	Each element is a single blue component. It is converted to floating point and assembled into an RGBA element by attaching 0.0 for red and green, and 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).
GL_ALPHA	Each element is a single red component. It is converted to floating point and assembled into an RGBA element by attaching 0.0 for red, green, and blue. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).
GL_RGB	Each element is an RGB triple. It is converted to floating point and assembled into an RGBA element by attaching 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

VALUE	MEANING
GL_RGBA	Each element is a complete RGBA element. It is converted to floating point. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see glPixelTransfer).
GL_BGR_EXT	Each pixel is a group of three components in this order: blue, green, red. GL_BGR_EXT provides a format that matches the memory layout of Windows device-independent bitmaps (DIBs). Thus your applications can use the same data with Windows function calls and OpenGL pixel function calls.
GL_BGRA_EXT	Each pixel is a group of four components in this order: blue, green, red, alpha. GL_BGRA_EXT provides a format that matches the memory layout of Windows device-independent bitmaps (DIBs). Thus your applications can use the same data with Windows function calls and OpenGL pixel function calls.
GL_LUMINANCE	Each element is a single luminance value. It is converted to floating point, and then assembled into an RGBA element by replicating the luminance value three times for red, green, and blue, and attaching 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).
GL_LUMINANCE_ALPHA	Each element is a luminance/alpha pair. It is converted to floating point, and then assembled into an RGBA element by replicating the luminance value three times for red, green, and blue. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see glPixelTransfer).

type

The data type of the pixel data. The following symbolic values are accepted:

GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, and GL_FLOAT.

pixels A pointer to the image data in memory.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*target* was not an GL_TEXTURE_2D.
GL_INVALID_ENUM	*format* was not an accepted *format* constant. Only format constants other than GL_STENCIL_INDEX and GL_DEPTH_COMPONENT are accepted. See the parameter description of *format* for a list of possible values.
GL_INVALID_ENUM	*type* was not a *type* constant.
GL_INVALID_ENUM	*type* was GL_BITMAP and *format* was not GL_COLOR_INDEX.

NAME	MEANING
GL_INVALID_VALUE	*level* was less than zero or greater than log2 *max*, where *max* was the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_VALUE	*internalformat* was was not 1, 2, 3, or 4.
GL_INVALID_VALUE	*width* or height was less than zero or greater than 2 + GL_MAX_TEXTURE_SIZE, or it could not be represented as 2*n* + 2(border) for some integer value of *n*.
GL_INVALID_VALUE	*border* was not 0 or 1.
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*target* was not an GL_TEXTURE_2D.

NAME	MEANING
GL_INVALID_ENUM	*format* was not an accepted *format* constant. Only format constants other than GL_STENCIL_INDEX and GL_DEPTH_COMPONENT are accepted. See the parameter description of *format* for a list of possible values.
GL_INVALID_ENUM	*type* was not a *type* constant.
GL_INVALID_ENUM	*type* was GL_BITMAP and *format* was not GL_COLOR_INDEX.
GL_INVALID_VALUE	*level* was less than zero or greater than log2 *max*, where *max* was the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_VALUE	*internalformat* was was not 1, 2, 3, or 4.
GL_INVALID_VALUE	*width* or height was less than zero or greater than 2 + GL_MAX_TEXTURE_SIZE, or it could not be represented as 2*n* + 2(border) for some integer value of *n*.
GL_INVALID_VALUE	*border* was not 0 or 1.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to glBegin and the corresponding call to glEnd.

Remarks

The **glTexImage2D** function specifies a two-dimensional texture image. Texturing maps a portion of a specified *texture image* onto each graphical primitive for which texturing is enabled. Two-dimensional texturing is enabled and disabled using glEnable and **glDisable** with argument GL_TEXTURE_2D.

Texture images are defined with **glTexImage2D**. The arguments describe the parameters of the texture image, such as height, width, width of the border, level-of-detail number (see glTexParameter), and number of color components provided. The last three arguments describe the way the image is represented in memory. These arguments are identical to the pixel formats used for glDrawPixels.

Data is read from *pixels* as a sequence of signed or unsigned bytes, shorts or longs, or single-precision floating-point values, depending on *type*. These values are grouped into sets of one, two, three, or four values, depending on *format*, to form elements. If *type* is GL_BITMAP, the data is considered as a string of unsigned bytes (and *format* must be GL_COLOR_INDEX). Each data byte is treated as eight 1-bit elements, with bit ordering determined by GL_UNPACK_LSB_FIRST (see glPixelStore). Please see glDrawPixels for a description of the acceptable values for the *type* parameter.

A texture image can have up to four components per texture element, depending on *components*. A one-component texture image uses only the red component of the RGBA color extracted from *pixels*. A two-component image uses the R and A values. A three-component image uses the R, G, and B values. A four-component image uses all of the RGBA components.

Texturing has no effect in color-index mode.

The texture image can be represented by the same data formats as the pixels in a **glDrawPixels** command, except that GL_STENCIL_INDEX and GL_DEPTH_COMPONENT cannot be used. The glPixelStore and glPixelTransfer modes affect texture images in exactly the way they affect **glDrawPixels**.

A texture image with zero height or width indicates the null texture. If the null texture is specified for level-of-detail 0, it is as if texturing were disabled.

The following functions retrieve information related to **glTexImage2D**:

glGetTexImage

glIsEnabled with argument GL_TEXTURE_2D

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glDrawPixels

glEnd

glFog

glIsEnabled

glPixelStore

glPixelTransfer

glTexEnv

glT

e
x
G
e
n

g
l
T
e
x
I
m
a
g
e
1
D

g
l
T
e
x
P
a
r
a
m
e
t
e
r

glTexParameter Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions sets texture parameters:

- glTexParameterf
- glTexParameteri
- glTexParameterfv
- glTexParameteriv

glTexParameterf function

2/22/2020 • 4 minutes to read • Edit Online

Sets texture parameters.

Syntax

```
void WINAPI glTexParameterf(
   GLenum  target,
   GLenum  pname,
   GLfloat param
);
```

Parameters

target The target texture, which must be either GL_TEXTURE_1D or GL_TEXTURE_2D.

pname The symbolic name of a single valued texture parameter. The following symbols are accepted in *pname*.

VALUE	MEANING
GL_TEXTURE_MIN_FILTER	The texture minifying function is used whenever the pixel being textured maps to an area greater than one texture element. There are six defined minifying functions. Two of them use the nearest one or nearest four texture elements to compute the texture value. The other four use mipmaps. A mipmap is an ordered set of arrays representing the same image at progressively lower resolutions. If the texture has dimensions $2n x 2^m$ there are max(n, m) + 1 mipmaps. The first mipmap is the original texture, with dimensions $2n x 2^m$. Each subsequent mipmap has dimensions $2^k 1 x 2^l 1$ where $2^k x 2^l$ are the dimensions of the previous mipmap, until either k = 0 or l = 0. At that point, subsequent mipmaps have dimension $1 x 2^l 1$ or $2^k 1 x 1$ until the final mipmap, which has dimension 1x1. Mipmaps are defined using **glTexImage1D** or **glTexImage2D** with the level-of-detail argument indicating the order of the mipmaps. Level 0 is the original texture; level bold max(n, m) is the final 1x1 mipmap.

VALUE	MEANING
GL_TEXTURE_MAG_FILTER	The texture magnification function is used when the pixel being textured maps to an area less than or equal to one texture element. It sets the texture magnification function to either GL_NEAREST or GL_LINEAR.
GL_TEXTURE_WRAP_S	Sets the wrap parameter for texture coordinate s to either GL_CLAMP or GL_REPEAT. GL_CLAMP causes s coordinates to be clamped to the range [0,1] and is useful for preventing wrapping artifacts when mapping a single image onto an object. GL_REPEAT causes the integer part of the s coordinate to be ignored; OpenGL uses only the fractional part, thereby creating a repeating pattern. Border texture elements are accessed only if wrapping is set to GL_CLAMP. Initially, GL_TEXTURE_WRAP_S is set to GL_REPEAT.
GL_TEXTURE_WRAP_T	Sets the wrap parameter for texture coordinate t to either GL_CLAMP or GL_REPEAT. See the discussion under GL_TEXTURE_WRAP_S. Initially, GL_TEXTURE_WRAP_T is set to GL_REPEAT

param The value of *pname*.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *pname* was not one of the accepted defined values, or when *param* should have had a defined constant value (based on the value of *pname*) and did not.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *pname* was not one of the accepted defined values, or when *param* should have had a defined constant value (based on the value of *pname*) and did not.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

Texture mapping is a technique that applies an image onto an object's surface as if the image were a decal or cellophane shrink-wrap. The image is created in texture space, with an (*s*, *t*) coordinate system. A texture is a one- or two-dimensional image and a set of parameters that determine how samples are derived from the image.

The **glTexParameter** function assigns the value or values in params to the texture parameter specified as pname. The target parameter defines the target texture, either GL_TEXTURE_1D or GL_TEXTURE_2D.

As more texture elements are sampled in the minification process, fewer aliasing artifacts will be apparent. While the GL_NEAREST and GL_LINEAR minification functions can be faster than the other four, they sample only one or four texture elements to determine the texture value of the pixel being rendered and can produce moire patterns or ragged transitions. The default value of GL_TEXTURE_MIN_FILTER is GL_NEAREST_MIPMAP_LINEAR.

Suppose that texturing is enabled (by calling **glEnable** with argument GL_TEXTURE_1D or GL_TEXTURE_2D) and GL_TEXTURE_MIN_FILTER is set to one of the functions that requires a mipmap. If either the dimensions of the texture images currently defined (with previous calls to **glTexImage1D** or **glTexImage2D**) do not follow the proper sequence for mipmaps, or there are fewer texture images defined than are needed, or the set of texture images have differing numbers of texture components, then it is as if texture mapping were disabled. Linear filtering accesses the four nearest texture elements only in 2-D textures. In 1-D textures, linear filtering accesses the

two nearest texture elements. The following function retrieves information related to **glTexParameterf**, **glTexParameteri**, **glTexParameterfv**, and **glTexParameteriv**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glBindTexture

glCop

y
P
i
x
e
l
s

g
l
C
o
p
y
T
e
x
I
m
a
g
e
1
D

g
l
C
o
p
y
T
e
x
I
m
a
g
e
2
D

g
l
C
o
p
y
T
e
x
S
u

b
l
m
a
g
e
2
D

g
l
D
r
a
w
P
i
x
e
l
s

g
l
E
n
d

g
l
G
e
t
T
e
x
P
a
r
a
m
e
t
e
r

g
l
P
i
x
e
l

S
t
o
r
e

g
l
P
i
x
e
l
T
r
a
n
s
f
e
r

g
l
P
r
i
o
r
i
t
i
z
e
T
e
x
t
u
r
e
s

g
l
T
e
x
E
n
v

g

lTexGen

glTexImage1D

glTexImage2D

glTexSubImage1D

glT

e
x
S
u
b
I
m
a
g
e
2
D

glTexParameterfv function

2/22/2020 • 6 minutes to read • Edit Online

Sets texture parameters.

Syntax

```
void WINAPI glTexParameterfv(
         GLenum  target,
         GLenum  pname,
   const GLfloat *params
);
```

Parameters

target The target texture, which must be either GL_TEXTURE_1D or GL_TEXTURE_2D.

pname The symbolic name of a single valued texture parameter. The following symbols are accepted in *pname*.

VALUE	MEANING
GL_TEXTURE_MIN_FILTER	The texture minifying function is used whenever the pixel being textured maps to an area greater than one texture element. There are six defined minifying functions. Two of them use the nearest one or nearest four texture elements to compute the texture value. The other four use mipmaps. A mipmap is an ordered set of arrays representing the same image at progressively lower resolutions. If the texture has dimensions $2n x 2^m$ there are max(n, m) + 1 mipmaps. The first mipmap is the original texture, with dimensions $2n x 2^m$. Each subsequent mipmap has dimensions $2^k 1 x 2^l 1$ where $2^k x 2^l$ are the dimensions of the previous mipmap, until either k = 0 or l = 0. At that point, subsequent mipmaps have dimension $1 x 2^l 1$ or $2^k 1 x 1$ until the final mipmap, which has dimension 1x1. Mipmaps are defined using **glTexImage1D** or **glTexImage2D** with the level-of-detail argument indicating the order of the mipmaps. Level 0 is the original texture; level bold max(n, m) is the final 1x1 mipmap.

VALUE	MEANING
GL_TEXTURE_MAG_FILTER	The texture magnification function is used when the pixel being textured maps to an area less than or equal to one texture element. It sets the texture magnification function to either GL_NEAREST or GL_LINEAR.
GL_TEXTURE_WRAP_S	Sets the wrap parameter for texture coordinate s to either GL_CLAMP or GL_REPEAT. GL_CLAMP causes s coordinates to be clamped to the range [0,1] and is useful for preventing wrapping artifacts when mapping a single image onto an object. GL_REPEAT causes the integer part of the s coordinate to be ignored; OpenGL uses only the fractional part, thereby creating a repeating pattern. Border texture elements are accessed only if wrapping is set to GL_CLAMP. Initially, GL_TEXTURE_WRAP_S is set to GL_REPEAT.
GL_TEXTURE_WRAP_T	Sets the wrap parameter for texture coordinate t to either GL_CLAMP or GL_REPEAT. See the discussion under GL_TEXTURE_WRAP_S. Initially, GL_TEXTURE_WRAP_T is set to GL_REPEAT.
GL_TEXTURE_BORDER_COLOR	Sets a border color. The *params* parameter contains four values that comprise the RGBA color of the texture border. Integer color components are interpreted linearly such that the most positive integer maps to 1.0, and the most negative integer maps to 1.0. The values are clamped to the range [0,1] when they are specified. Initially, the border color is (0, 0, 0, 0).
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.

p
a
r
a
m
s

A pointer to an array where the value or values of pname are stored. The params parameter supplies a function for minifying the texture as one of the following.

VALUE	MEANING
GL_NEAREST	Returns the value of the texture element that is nearest (in Manhattan distance) to the center of the pixel being textured.
GL_LINEAR	Returns the weighted average of the four texture elements that are closest to the center of the pixel being textured. These can include border texture elements, depending on the values of GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T, and on the exact mapping. GL_NEAREST is generally faster than GL_LINEAR, but it can produce textured images with sharper edges because the transition between texture elements is not as smooth. The default value of GL_TEXTURE_MAG_FILTER is GL_LINEAR.
GL_NEAREST_MIPMAP_NEAREST	Chooses the mipmap that most closely matches the size of the pixel being textured and uses the GL_NEAREST criterion (the texture element nearest to the center of the pixel) to produce a texture value.
GL_LINEAR_MIPMAP_NEAREST	Chooses the mipmap that most closely matches the size of the pixel being textured and uses the GL_LINEAR criterion (a weighted average of the four texture elements that are closest to the center of the pixel) to produce a texture value.
GL_NEAREST_MIPMAP_LINEAR	Chooses the two mipmaps that most closely match the size of the pixel being textured and uses the GL_NEAREST criterion (the texture element nearest to the center of the pixel) to produce a texture value from each mipmap. The final texture value is a weighted average of those two values.

VALUE	MEANING
GL_ LIN EAR _MI PM AP_ LIN EAR	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.

The *params* parameter supplies a function for magnifying the texture as one of the following.

VALUE	MEANING
GL_ NE ARE ST	Returns the value of the texture element that is nearest (in Manhattan distance) to the center of the pixel being textured.
GL_ LIN EAR	Returns the weighted average of the four texture elements that are closest to the center of the pixel being textured. These can include border texture elements, depending on the values of GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T, and on the exact mapping. GL_NEAREST is generally faster than GL_LINEAR, but it can produce textured images with sharper edges because the transition between texture elements is not as smooth. The default value of GL_TEXTURE_MAG_FILTER is GL_LINEAR.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_ INV ALI D_E NU M	*target* or *pname* was not one of the accepted defined values, or when *param* should have had a defined constant value (based on the value of *pname*) and did not.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *pname* was not one of the accepted defined values, or when *param* should have had a defined constant value (based on the value of *pname*) and did not.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

Texture mapping is a technique that applies an image onto an object's surface as if the image were a decal or cellophane shrink-wrap. The image is created in texture space, with an (*s*, *t*) coordinate system. A texture is a one- or two-dimensional image and a set of parameters that determine how samples are derived from the image.

The **glTexParameter** function assigns the value or values in params to the texture parameter specified as pname. The target parameter defines the target texture, either GL_TEXTURE_1D or GL_TEXTURE_2D.

As more texture elements are sampled in the minification process, fewer aliasing artifacts will be apparent. While the GL_NEAREST and GL_LINEAR minification functions can be faster than the other four, they sample only one or four texture elements to determine the texture value of the pixel being rendered and can produce moire patterns or ragged transitions. The default value of GL_TEXTURE_MIN_FILTER is GL_NEAREST_MIPMAP_LINEAR.

Suppose that texturing is enabled (by calling **glEnable** with argument GL_TEXTURE_1D or GL_TEXTURE_2D) and GL_TEXTURE_MIN_FILTER is set to one of the functions that requires a mipmap. If either the dimensions of the texture images currently defined (with previous calls to **glTexImage1D** or **glTexImage2D**) do not follow the proper sequence for mipmaps, or there are fewer texture images defined than are needed, or the set of texture images have differing numbers of texture components, then it is as if texture mapping were disabled. Linear filtering accesses the four nearest texture elements only in 2-D textures. In 1-D textures, linear filtering accesses the

two nearest texture elements. The following function retrieves information related to **glTexParameterf**, **glTexParameteri**, **glTexParameterfv**, and **glTexParameteriv**:

glGetTexParameter

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glBindTexture

glCo

p
y
P
i
x
e
l
s

g
l
C
o
p
y
T
e
x
I
m
a
g
e
1
D

g
l
C
o
p
y
T
e
x
I
m
a
g
e
2
D

g
l
C
o
p
y
T
e
x
S

u
b
l
m
a
g
e
2
D

g
l
D
r
a
w
P
i
x
e
l
s

g
l
E
n
d

g
l
G
e
t
T
e
x
P
a
r
a
m
e
t
e
r

g
l
P
i
x
e

lStore

glPixelTransfer

glPrioritizeTextures

glTexEnv

glTexGen

glTexImage1D

glTexImage2D

glTexSubImage1D

gl

TexSubImage2D

glTexParameteri function

2/22/2020 • 4 minutes to read • Edit Online

Sets texture parameters.

Syntax

```
void WINAPI glTexParameteri(
   GLenum target,
   GLenum pname,
   GLint  param
);
```

Parameters

target The target texture, which must be either GL_TEXTURE_1D or GL_TEXTURE_2D.

pname The symbolic name of a single valued texture parameter. The following symbols are accepted in *pname*.

VALUE	MEANING
GL_TEXTURE_MIN_FILTER	The texture minifying function is used whenever the pixel being textured maps to an area greater than one texture element. There are six defined minifying functions. Two of them use the nearest one or nearest four texture elements to compute the texture value. The other four use mipmaps. A mipmap is an ordered set of arrays representing the same image at progressively lower resolutions. If the texture has dimensions $2n x 2^m$ there are max(n, m) + 1 mipmaps. The first mipmap is the original texture, with dimensions $2n x 2^m$. Each subsequent mipmap has dimensions $2^k 1 x 2^l 1$ where $2^k x 2^l$ are the dimensions of the previous mipmap, until either k = 0 or l = 0. At that point, subsequent mipmaps have dimension $1 x 2^l 1$ or $2^k 1 x 1$ until the final mipmap, which has dimension 1x1. Mipmaps are defined using **glTexImage1D** or **glTexImage2D** with the level-of-detail argument indicating the order of the mipmaps. Level 0 is the original texture; level bold max(n, m) is the final 1x1 mipmap.

VALUE	MEANING
GL_TEXTURE_MAG_FILTER	The texture magnification function is used when the pixel being textured maps to an area less than or equal to one texture element. It sets the texture magnification function to either GL_NEAREST or GL_LINEAR.
GL_TEXTURE_WRAP_S	Sets the wrap parameter for texture coordinate s to either GL_CLAMP or GL_REPEAT. GL_CLAMP causes s coordinates to be clamped to the range [0,1] and is useful for preventing wrapping artifacts when mapping a single image onto an object. GL_REPEAT causes the integer part of the s coordinate to be ignored; OpenGL uses only the fractional part, thereby creating a repeating pattern. Border texture elements are accessed only if wrapping is set to GL_CLAMP. Initially, GL_TEXTURE_WRAP_S is set to GL_REPEAT.
GL_TEXTURE_WRAP_T	Sets the wrap parameter for texture coordinate t to either GL_CLAMP or GL_REPEAT. See the discussion under GL_TEXTURE_WRAP_S. Initially, GL_TEXTURE_WRAP_T is set to GL_REPEAT

param The value of *pname*.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *pname* was not one of the accepted defined values, or when *param* should have had a defined constant value (based on the value of *pname*) and did not.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *pname* was not one of the accepted defined values, or when *param* should have had a defined constant value (based on the value of *pname*) and did not.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

Texture mapping is a technique that applies an image onto an object's surface as if the image were a decal or cellophane shrink-wrap. The image is created in texture space, with an (*s*, *t*) coordinate system. A texture is a one- or two-dimensional image and a set of parameters that determine how samples are derived from the image.

The **glTexParameter** function assigns the value or values in params to the texture parameter specified as pname. The target parameter defines the target texture, either GL_TEXTURE_1D or GL_TEXTURE_2D.

As more texture elements are sampled in the minification process, fewer aliasing artifacts will be apparent. While the GL_NEAREST and GL_LINEAR minification functions can be faster than the other four, they sample only one or four texture elements to determine the texture value of the pixel being rendered and can produce moire patterns or ragged transitions. The default value of GL_TEXTURE_MIN_FILTER is GL_NEAREST_MIPMAP_LINEAR.

Suppose that texturing is enabled (by calling **glEnable** with argument GL_TEXTURE_1D or GL_TEXTURE_2D) and GL_TEXTURE_MIN_FILTER is set to one of the functions that requires a mipmap. If either the dimensions of the texture images currently defined (with previous calls to **glTexImage1D** or **glTexImage2D**) do not follow the proper sequence for mipmaps, or there are fewer texture images defined than are needed, or the set of texture images have differing numbers of texture components, then it is as if texture mapping were disabled. Linear filtering accesses the four nearest texture elements only in 2-D textures. In 1-D textures, linear filtering accesses the

two nearest texture elements. The following function retrieves information related to **glTexParameterf**, **glTexParameteri**, **glTexParameterfv**, and **glTexParameteriv**:

glGetTexParameter

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glBindTexture

glCo

p
y
P
i
x
e
l
s

g
l
C
o
p
y
T
e
x
I
m
a
g
e
1
D

g
l
C
o
p
y
T
e
x
I
m
a
g
e
2
D

g
l
C
o
p
y
T
e
x
S

u
b
l
m
a
g
e
2
D

g
l
D
r
a
w
P
i
x
e
l
s

g
l
E
n
d

g
l
G
e
t
T
e
x
P
a
r
a
m
e
t
e
r

g
l
P
i
x
e

lStore

glPixelTransfer

glPrioritizeTextures

glTexEnv

glTexGen

glTexImage1D

glTexImage2D

glTexSubImage1D

gl

TexSubImage2D

glTexParameteriv function

2/22/2020 • 6 minutes to read • Edit Online

Sets texture parameters.

Syntax

```
void WINAPI glTexParameterfv(
         GLenum target,
         GLenum pname,
   const GLint  *params
);
```

Parameters

target The target texture, which must be either GL_TEXTURE_1D or GL_TEXTURE_2D.

pname The symbolic name of a single valued texture parameter. The following symbols are accepted in *pname*.

VALUE	MEANING
GL_TEXTURE_MIN_FILTER	The texture minifying function is used whenever the pixel being textured maps to an area greater than one texture element. There are six defined minifying functions. Two of them use the nearest one or nearest four texture elements to compute the texture value. The other four use mipmaps. A mipmap is an ordered set of arrays representing the same image at progressively lower resolutions. If the texture has dimensions $2n x 2^m$ there are max(n, m) + 1 mipmaps. The first mipmap is the original texture, with dimensions $2n x 2^m$. Each subsequent mipmap has dimensions $2^k 1 x 2^l 1$ where $2^k x 2^l$ are the dimensions of the previous mipmap, until either k = 0 or l = 0. At that point, subsequent mipmaps have dimension $1 x 2^l 1$ or $2^k 1 x 1$ until the final mipmap, which has dimension 1x1. Mipmaps are defined using **glTexImage1D** or **glTexImage2D** with the level-of-detail argument indicating the order of the mipmaps. Level 0 is the original texture; level bold max(n, m) is the final 1x1 mipmap.

VALUE	MEANING
GL_TEXTURE_MAG_FILTER	The texture magnification function is used when the pixel being textured maps to an area less than or equal to one texture element. It sets the texture magnification function to either GL_NEAREST or GL_LINEAR.
GL_TEXTURE_WRAP_S	Sets the wrap parameter for texture coordinate s to either GL_CLAMP or GL_REPEAT. GL_CLAMP causes s coordinates to be clamped to the range [0,1] and is useful for preventing wrapping artifacts when mapping a single image onto an object. GL_REPEAT causes the integer part of the s coordinate to be ignored; OpenGL uses only the fractional part, thereby creating a repeating pattern. Border texture elements are accessed only if wrapping is set to GL_CLAMP. Initially, GL_TEXTURE_WRAP_S is set to GL_REPEAT.
GL_TEXTURE_WRAP_T	Sets the wrap parameter for texture coordinate t to either GL_CLAMP or GL_REPEAT. See the discussion under GL_TEXTURE_WRAP_S. Initially, GL_TEXTURE_WRAP_T is set to GL_REPEAT.
GL_TEXTURE_BORDER_COLOR	Sets a border color. The *params* parameter contains four values that comprise the RGBA color of the texture border. Integer color components are interpreted linearly such that the most positive integer maps to 1.0, and the most negative integer maps to 1.0. The values are clamped to the range [0,1] when they are specified. Initially, the border color is (0, 0, 0, 0).
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.

p
a
r
a
m
s

A pointer to an array where the value or values of pname are stored. The params parameter supplies a function for minifying the texture as one of the following.

VALUE	MEANING
GL_NEAREST	Returns the value of the texture element that is nearest (in Manhattan distance) to the center of the pixel being textured.
GL_LINEAR	Returns the weighted average of the four texture elements that are closest to the center of the pixel being textured. These can include border texture elements, depending on the values of GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T, and on the exact mapping. GL_NEAREST is generally faster than GL_LINEAR, but it can produce textured images with sharper edges because the transition between texture elements is not as smooth. The default value of GL_TEXTURE_MAG_FILTER is GL_LINEAR.
GL_NEAREST_MIPMAP_NEAREST	Chooses the mipmap that most closely matches the size of the pixel being textured and uses the GL_NEAREST criterion (the texture element nearest to the center of the pixel) to produce a texture value.
GL_LINEAR_MIPMAP_NEAREST	Chooses the mipmap that most closely matches the size of the pixel being textured and uses the GL_LINEAR criterion (a weighted average of the four texture elements that are closest to the center of the pixel) to produce a texture value.
GL_NEAREST_MIPMAP_LINEAR	Chooses the two mipmaps that most closely match the size of the pixel being textured and uses the GL_NEAREST criterion (the texture element nearest to the center of the pixel) to produce a texture value from each mipmap. The final texture value is a weighted average of those two values.

VALUE	MEANING
GL_ LIN EAR _MI PM AP_ LIN EAR	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.

The *params* parameter supplies a function for magnifying the texture as one of the following.

VALUE	MEANING
GL_ NE ARE ST	Returns the value of the texture element that is nearest (in Manhattan distance) to the center of the pixel being textured.
GL_ LIN EAR	Returns the weighted average of the four texture elements that are closest to the center of the pixel being textured. These can include border texture elements, depending on the values of GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T, and on the exact mapping. GL_NEAREST is generally faster than GL_LINEAR, but it can produce textured images with sharper edges because the transition between texture elements is not as smooth. The default value of GL_TEXTURE_MAG_FILTER is GL_LINEAR.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_ INV ALI D_E NU M	*target* or *pname* was not one of the accepted defined values, or when *param* should have had a defined constant value (based on the value of *pname*) and did not.

NAME	MEANING
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* or *pname* was not one of the accepted defined values, or when *param* should have had a defined constant value (based on the value of *pname*) and did not.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

Texture mapping is a technique that applies an image onto an object's surface as if the image were a decal or cellophane shrink-wrap. The image is created in texture space, with an (*s*, *t*) coordinate system. A texture is a one- or two-dimensional image and a set of parameters that determine how samples are derived from the image.

The **glTexParameter** function assigns the value or values in params to the texture parameter specified as pname. The target parameter defines the target texture, either GL_TEXTURE_1D or GL_TEXTURE_2D.

As more texture elements are sampled in the minification process, fewer aliasing artifacts will be apparent. While the GL_NEAREST and GL_LINEAR minification functions can be faster than the other four, they sample only one or four texture elements to determine the texture value of the pixel being rendered and can produce moire patterns or ragged transitions. The default value of GL_TEXTURE_MIN_FILTER is GL_NEAREST_MIPMAP_LINEAR.

Suppose that texturing is enabled (by calling **glEnable** with argument GL_TEXTURE_1D or GL_TEXTURE_2D) and GL_TEXTURE_MIN_FILTER is set to one of the functions that requires a mipmap. If either the dimensions of the texture images currently defined (with previous calls to **glTexImage1D** or **glTexImage2D**) do not follow the proper sequence for mipmaps, or there are fewer texture images defined than are needed, or the set of texture images have differing numbers of texture components, then it is as if texture mapping were disabled. Linear filtering accesses the four nearest texture elements only in 2-D textures. In 1-D textures, linear filtering accesses the

two nearest texture elements. The following function retrieves information related to **glTexParameterf**, **glTexParameteri**, **glTexParameterfv**, and **glTexParameteriv**:

glGetTexParameter

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glBindTexture

glCo

p
y
P
i
x
e
l
s

g
l
C
o
p
y
T
e
x
I
m
a
g
e
1
D

g
l
C
o
p
y
T
e
x
I
m
a
g
e
2
D

g
l
C
o
p
y
T
e
x
S

u
b
l
m
a
g
e
2
D

g
l
D
r
a
w
P
i
x
e
l
s

g
l
E
n
d

g
l
G
e
t
T
e
x
P
a
r
a
m
e
t
e
r

g
l
P
i
x
e

lStore

glPixelTransfer

glPrioritizeTextures

glTexEnv

glTexGen

glTexImage1D

glTexImage2D

glTexSubImage1D

gl

TexSubImage2D

glTexSubImage1D function

2/22/2020 • 5 minutes to read • Edit Online

The **glTexSubImage1D** function specifies a portion of an existing one-dimensional texture image. You cannot define a new texture with **glTexSubImage1D**.

Syntax

```
void WINAPI glTexSubImage1D(
         GLenum  target,
         GLint   level,
         GLint   xoffset,
         GLsizei width,
         GLenum  format,
         GLenum  type,
   const GLvoid  *pixels
);
```

Parameters

target The target texture. Must be GL_TEXTURE_1D.

level The level-of-detail number. Level 0 is the base image. Level *n* is the *n*th mipmap reduction image.

xoffset A texel offset in the *x* direction within the texture array.

width The width of the texture sub-image.

format The format of the pixel data. This parameter can assume one of the following symbolic values.

m
a
t

VALUE	MEANING
GL_COLOR_INDEX	Each element is a single value, a color index. It is converted to fixed point format (with an unspecified number of 0 bits to the right of the binary point), shifted left or right, depending on the value and sign of GL_INDEX_SHIFT, and added to GL_INDEX_OFFSET (see **glPixelTransfer**). The resulting index is converted to a set of color components using the GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, and GL_PIXEL_MAP_I_TO_A tables, and clamped to the range [0,1].
GL_RED	Each element is a single red component. It is converted to floating point format and assembled into an RGBA element by attaching 0.0 for green and blue, and 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).
GL_GREEN	Each element is a single green component. It is converted to floating point format and assembled into an RGBA element by attaching 0.0 for red and blue, and 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).
GL_BLUE	Each element is a single blue component. It is converted to floating point format and assembled into an RGBA element by attaching 0.0 for red and green, and 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).
GL_ALPHA	Each element is a single alpha component. It is converted to floating point format and assembled into an RGBA element by attaching 0.0 for red, green, and blue. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).
GL_RGB	Each element is an RGB triple. It is converted to floating point and assembled into an RGBA element by attaching 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

VALUE	MEANING
GL_RGBA	Each element is a complete RGBA element. It is converted to floating point format. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).
GL_LUMINANCE	Each element is a single luminance value. It is converted to floating point format, and then assembled into an RGBA element by replicating the luminance value three times for red, green, and blue, and attaching 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see glPixelTransfer).
GL_LUMINANCE_ALPHA	Each element is a luminance/alpha pair. It is converted to floating point format, and then assembled into an RGBA element by replicating the luminance value three times for red, green, and blue. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

type

The data type of the pixel data. The following symbolic values are accepted: GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, and GL_FLOAT.

pixels

A pointer to the image data in memory.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the glGetError function.

NAME	MEANING
GL_INVALID_ENUM	*target* was not GL_TEXTURE_1D.
GL_INVALID_ENUM	*format* was not an accepted constant.
GL_INVALID_ENUM	*type* was not an accepted constant.
GL_INVALID_ENUM	*type* was GL_BITMAP and *format* was not GL_COLOR_INDEX.
GL_INVALID_VALUE	*level* was less than zero or greater than log2 *max*, where *max* was the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_VALUE	*xoffset* was less than *b*, or *offset* + *width* was greater than *wb*, where *w* is the GL_TEXTURE_WIDTH, and *b* is the width of the GL_TEXTURE_BORDER of the texture image being modified. Note that *w* includes twice the border width.
GL_INVALID_VALUE	*width* was was less than *b*, where *b* is the border width of the texture array.

NAME	MEANING
GL_INVALID_VALUE	*border* was not zero or 1.
GL_INVALID_OPERATION	The texure array was not defined by a previous **glTexImage1D** operation.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* was not GL_TEXTURE_1D.
GL_INVALID_ENUM	*format* was not an accepted constant.

NAME	MEANING
GL_INVALID_ENUM	*type* was not an accepted constant.
GL_INVALID_ENUM	*type* was GL_BITMAP and *format* was not GL_COLOR_INDEX.
GL_INVALID_VALUE	*level* was less than zero or greater than log2 *max*, where *max* was the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_VALUE	*xoffset* was less than *b*, or *offset* + *width* was greater than *wb*, where *w* is the GL_TEXTURE_WIDTH, and *b* is the width of the GL_TEXTURE_BORDER of the texture image being modified. Note that *w* includes twice the border width.
GL_INVALID_VALUE	*width* was was less than *b*, where *b* is the border width of the texture array.
GL_INVALID_VALUE	*border* was not zero or 1.

NAME	MEANING
GL_INVALID_OPERATION	The texure array was not defined by a previous **glTexImage1D** operation.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

One-dimensional texturing for a primitive is enabled using **glEnable** and **glDisable** with the argument GL_TEXTURE_1D. During texturing, part of a specified texture image is mapped into each enabled primitive. You use the **glTexSubImage1D** function to specify a contiguous sub-image of an existing one-dimensional texture image for texturing.

The texels referenced by *pixels* replace a region of the existing texture array with *x* indexes of *xoffset* and *xoffset* + (*width* 1) inclusive. This region cannot include any texels outside the range of the originally specified texture array.

Specifying a sub-image with a *width* of zero has no effect and does not generate an error.

Texturing has no effect in color-index mode.

In general, texture images can be represented by the same data formats as the pixels in a **glDrawPixels** command, except that GL_STENCIL_INDEX and GL_DEPTH_COMPONENT cannot be used. The **glPixelStore** and **glPixelTransfer** modes affect texture images in exactly the way they affect **glDrawPixels**.

The following functions retrieve information related to **glTexSubImage1D**:

glGetTexImage

glIsEnabled with argument GL_TEXTURE_1D

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glCopyTexImage1D

glCopyTexImage2D

glCop

y
T
e
x
S
u
b
I
m
a
g
e
1
D

g
l
C
o
p
y
T
e
x
S
u
b
I
m
a
g
e
2
D

g
l
D
r
a
w
P
i
x
e
l
s

g
l
E
n
a

b
l
e

g
l
F
o
g

g
l
G
e
t
T
e
x
I
m
a
g
e

g
l
I
s
E
n
a
b
l
e
d

g
l
P
i
x
e
l
S
t
o
r
e

g
l
P
i

x
e
l
T
r
a
n
s
f
e
r

g
l
T
e
x
E
n
v

g
l
T
e
x
G
e
n

g
l
T
e
x
I
m
a
g
e
1
D

g
l
T
e
x
I
m
a
g
e

2
D

glTexParameter

glTexSubImage2D

glTexSubImage2D function

2/22/2020 • 6 minutes to read • Edit Online

The **glTexSubImage2D** function specifies a portion of an existing one-dimensional texture image. You cannot define a new texture with **glTexSubImage2D**.

Syntax

```
void WINAPI glTexSubImage2D(
         GLenum  target,
         GLint   level,
         GLint   xoffset,
         GLint   yoffset,
         GLsizei width,
         GLsizei height,
         GLenum  format,
         GLenum  type,
   const GLvoid  *pixels
);
```

Parameters

target The target texture. Must be GL_TEXTURE_2D.

level The level-of-detail number. Level 0 is the base image. Level *n* is the *n*th mipmap reduction image.

xoffset A texel offset in the *x* direction within the texture array.

yoffset A texel offset in the *y* direction within the texture array.

width The width of the texture sub-image.

height The height of the texture sub-image.

format The format of the pixel data. It can assume one of the following symbolic values.

VALUE	MEANING
GL_COLOR_INDEX	Each element is a single value, a color index. It is converted to fixed point format (with an unspecified number of 0 bits to the right of the binary point), shifted left or right, depending on the value and sign of GL_INDEX_SHIFT, and added to GL_INDEX_OFFSET (see **glPixelTransfer**). The resulting index is converted to a set of color components using the GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, and GL_PIXEL_MAP_I_TO_A tables, and clamped to the range [0,1].
GL_RED	Each element is a single red component. It is converted to floating point format and assembled into an RGBA element by attaching 0.0 for green and blue, and 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).
GL_GREEN	Each element is a single green component. It is converted to floating point format and assembled into an RGBA element by attaching 0.0 for red and blue, and 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

VALUE	MEANING
GL_BLUE	Each element is a single blue component. It is converted to floating point format and assembled into an RGBA element by attaching 0.0 for red and green, and 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).
GL_ALPHA	Each element is a single alpha component. It is converted to floating point format and assembled into an RGBA element by attaching 0.0 for red, green, and blue. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see glPixelTransfer).
GL_RGB	Each element is an RGB triple. It is converted to floating point format and assembled into an RGBA element by attaching 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).
GL_RGBA	Each element is a complete RGBA element. It is converted to floating point. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).
GL_LUMINANCE	Each element is a single luminance value. It is converted to floating point format, and then assembled into an RGBA element by replicating the luminance value three times for red, green, and blue, and attaching 1.0 for alpha. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see glPixelTransfer).
GL_LUMINANCE_ALPHA	Each element is a luminance/alpha pair. It is converted to floating point format, and then assembled into an RGBA element by replicating the luminance value three times for red, green, and blue. Each component is then multiplied by the signed scale factor GL_c_SCALE, added to the signed bias GL_c_BIAS, and clamped to the range [0,1] (see **glPixelTransfer**).

t

y

p
e

The data type of the pixel data. The following symbolic values are accepted: GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, and GL_FLOAT.

p
i
x
e
l
s

A pointer to the image data in memory.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* was not GL_TEXTURE_2D.
GL_INVALID_ENUM	*format* was not an accepted constant.
GL_INVALID_ENUM	*type* was not an accepted constant.
GL_INVALID_ENUM	*type* was GL_BITMAP and *format* was not GL_COLOR_INDEX.

NAME	MEANING
GL_INVALID_VALUE	*level* was less than zero or greater than log2 *max*, where *max* was the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_VALUE	*xoffset* was less than -*b*, or *xoffset* + *width* was greater than *w* - *b*, or *yoffset* was less than -*b*, or *yoffset* + *height* was greater than *h* - *b*, where *w* is the GL_TEXTURE_WIDTH, *h* is the GL_TEXTURE_HEIGHT, and *b* is the width of the GL_TEXTURE_BORDER of the texture image being modified. Note that *w* and *h* include twice the border width.
GL_INVALID_VALUE	*width* was was less than *b*, where *b* is the border width of the texture array.
GL_INVALID_VALUE	*border* was not zero or 1.
GL_INVALID_OPERATION	The texture array was not defined by a previous **glTexImage2D** operation.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_ENUM	*target* was not GL_TEXTURE_2D.
GL_INVALID_ENUM	*format* was not an accepted constant.
GL_INVALID_ENUM	*type* was not an accepted constant.
GL_INVALID_ENUM	*type* was GL_BITMAP and *format* was not GL_COLOR_INDEX.
GL_INVALID_VALUE	*level* was less than zero or greater than log2 *max*, where *max* was the returned value of GL_MAX_TEXTURE_SIZE.
GL_INVALID_VALUE	*xoffset* was less than -*b*, or *xoffset* + *width* was greater than *w* - *b*, or *yoffset* was less than -*b*, or *yoffset* + *height* was greater than *h* - *b*, where *w* is the GL_TEXTURE_WIDTH, *h* is the GL_TEXTURE_HEIGHT, and *b* is the width of the GL_TEXTURE_BORDER of the texture image being modified. Note that *w* and *h* include twice the border width.
GL_INVALID_VALUE	*width* was was less than *b*, where *b* is the border width of the texture array.

NAME	MEANING
GL_INVALID_VALUE	*border* was not zero or 1.
GL_INVALID_OPERATION	The texture array was not defined by a previous **glTexImage2D** operation.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

Two-dimensional texturing for a primitive is enabled using **glEnable** and **glDisable** with the argument GL_TEXTURE_2D. During texturing, part of a specified texture image is mapped into each enabled primitive. You use the **glTexSubImage2D** function to specify a contiguous sub-image of an existing two-dimensional texture image for texturing.

The texels referenced by *pixels* replace a region of the existing texture array with *x* indexes of *xoffset* and *xoffset* + (*width* 1) inclusive and *y* indexes of *yoffset* and *yoffset* + (*height* 1) inclusive. This region cannot include any texels outside the range of the originally specified texture array.

Specifying a sub-image with a *width* of zero has no effect and does not generate an error.

Texturing has no effect in color-index mode.

In general, texture images can be represented by the same data formats as the pixels in a **glDrawPixels** command, except that GL_STENCIL_INDEX and GL_DEPTH_COMPONENT cannot be used. The **glPixelStore** and **glPixelTransfer** modes affect texture images in exactly the way they affect **glDrawPixels**.

The following functions retrieve information related to **glTexSubImage2D**:

glGetTexImage

glIsEnabled with argument GL_TEXTURE_2D

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glCopyTexImage1D

glCopyTexImage2

D

g
l
C
o
p
y
T
e
x
S
u
b
I
m
a
g
e
1
D

g
l
C
o
p
y
T
e
x
S
u
b
I
m
a
g
e
2
D

g
l
D
r
a
w
P
i
x
e
l

s

glEnable

glFog

glGetTeximage

glIsEnabled

glPixelSto

r
e

g
l
P
i
x
e
l
T
r
a
n
s
f
e
r

g
l
T
e
x
E
n
v

g
l
T
e
x
G
e
n

g
l
T
e
x
I
m
a
g
e
1
D

g
l
T

e
x
I
m
a
g
e
2
D

g
l
T
e
x
S
u
b
I
m
a
g
e
1
D

g
l
T
e
x
I
m
a
g
e
2
D

g
l
T
e
x
P
a
r
a
m
e
t
e
r

glTranslate Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions multiply the current matrix by a translation matrix:

- glTranslated
- glTranslatef

glTranslated function

2/22/2020 • 2 minutes to read • Edit Online

The **glTranslated** function multiplies the current matrix by a translation matrix.

Syntax

```
void WINAPI glTranslated(
   GLdouble x,
   GLdouble y,
   GLdouble z
);
```

Parameters

x	The *x* coordinate of a translation vector.
y	The *y* coordinate of a translation vector.
z	The *z* coordinate of a translation vector.

Return value

This function does not return a value.

Remarks

The **glTranslated** function produces the translation specified by (*x*, *y*, *z*). The translation vector is used to compute a 4x4 translation matrix:

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

The current matrix (see **glMatrixMode**) is multiplied by this translation matrix, with the product replacing the current matrix. That is, if M is the current matrix and T is the translation matrix, then M is replaced with M T.

If the matrix mode is either GL_MODELVIEW or GL_PROJECTION, all objects drawn after **glTranslated** is called are translated. Use **glPushMatrix** and **glPopMatrix** to save and restore the untranslated coordinate system.

The following functions retrieve information related to **glTranslated**:

glGet with argument GL_MATRIX_MODE

glGet with argument GL_MODELVIEW_MATRIX

glGet with argument GL_PROJECTION_MATRIX

glGet with argument GL_TEXTURE_MATRIX

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glMatrixMode

glMul

t
M
a
t
r
i
x

g
l
P
u
s
h
M
a
t
r
i
x

g
l
R
o
t
a
t
e

g
l
S
c
a
l
e

glTranslatef function

2/22/2020 • 2 minutes to read • Edit Online

The **glTranslatef** function multiplies the current matrix by a translation matrix.

Syntax

```
void WINAPI glTranslatef(
   GLfloat x,
   GLfloat y,
   GLfloat z
);
```

Parameters

x	The *x* coordinate of a translation vector.
y	The *y* coordinate of a translation vector.
z	The *z* coordinate of a translation vector.

Return value

This function does not return a value.

Remarks

The **glTranslatef** function produces the translation specified by (*x*, *y*, *z*). The translation vector is used to compute a 4x4 translation matrix:

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

The current matrix (see **glMatrixMode**) is multiplied by this translation matrix, with the product replacing the current matrix. That is, if M is the current matrix and T is the translation matrix, then M is replaced with M T.

If the matrix mode is either GL_MODELVIEW or GL_PROJECTION, all objects drawn after **glTranslatef** is called are translated. Use **glPushMatrix** and **glPopMatrix** to save and restore the untranslated coordinate system.

The following functions retrieve information related to **glTranslated** and **glTranslatef**:

glGet with argument GL_MATRIX_MODE

glGet with argument GL_MODELVIEW_MATRIX

glGet with argument GL_PROJECTION_MATRIX

glGet with argument GL_TEXTURE_MATRIX

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glEnd

glMatrixMode

glMul

t
M
a
t
r
i
x

g
l
P
u
s
h
M
a
t
r
i
x

g
l
R
o
t
a
t
e

g
l
S
c
a
l
e

glVertex Functions

2/22/2020 • 2 minutes to read • Edit Online

These functions specify a vertex:

- glVertex2d
- glVertex2f
- glVertex2i
- glVertex2s
- glVertex3d
- glVertex3f
- glVertex3i
- glVertex3s
- glVertex4d
- glVertex4f
- glVertex4i
- glVertex4s
- glVertex2dv
- glVertex2fv
- glVertex2iv
- glVertex2sv
- glVertex3dv
- glVertex3fv
- glVertex3iv
- glVertex3sv
- glVertex4dv
- glVertex4fv
- glVertex4iv
- glVertex4sv

glVertex2d function

2/22/2020 • 2 minutes to read • Edit Online

Specifies a vertex.

Syntax

```
void WINAPI glVertex2d(
   GLdouble x,
   GLdouble y
);
```

Parameters

x	Specifies the x-coordinate of a vertex.
y	Specifies the y-coordinate of a vertex.

Return value

This function does not return a value.

Remarks

The glVertex function commands are used within **glBegin**/**glEnd** pairs to specify point, line, and polygon vertices. The current color, normal, and texture coordinates are associated with the vertex when glVertex is called. When only *x* and *y* are specified, *z* defaults to 0.0 and *w* defaults to 1.0. When *x*, *y*, and *z* are specified, *w* defaults to 1.0. Invoking glVertex outside of a **glBegin**/**glEnd** pair results in undefined behavior.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glCallList

glColor

glEdgeFlag

glEnd

glEvalC

o
o
r
d

g
l
I
n
d
e
x

g
l
M
a
t
e
r
i
a
l

g
l
N
o
r
m
a
l

g
l
R
e
c
t

g
l
T
e
x
C
o
o
r
d

glVertex2dv function

2/22/2020 • 2 minutes to read • Edit Online

Specifies a vertex.

Syntax

```
void WINAPI glVertex2dv(
   const GLdouble *v
);
```

Parameters

v	A pointer to an array of two elements. The elements are the x and y coordinates of a vertex.

Return value

This function does not return a value.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

glCallList

glColor

glEdgeFlag

glEnd

glEvalCoord

glIn

d
e
x

g
l
M
a
t
e
r
i
a
l

g
l
N
o
r
m
a
l

g
l
R
e
c
t

g
l
T
e
x
C
o
o
r
d

glVertex2f function

2/22/2020 • 2 minutes to read • Edit Online

Specifies a vertex.

Syntax

```
void WINAPI glVertex2f(
   GLfloat x,
   GLfloat y
);
```

Parameters

x	Specifies the x-coordinate of a vertex.
y	Specifies the y-coordinate of a vertex.

Return value

This function does not return a value.

Remarks

The glVertex function commands are used within **glBegin**/**glEnd** pairs to specify point, line, and polygon vertices. The current color, normal, and texture coordinates are associated with the vertex when glVertex is called. When only *x* and *y* are specified, *z* defaults to 0.0 and *w* defaults to 1.0. When *x*, *y*, and *z* are specified, *w* defaults to 1.0. Invoking glVertex outside of a **glBegin**/**glEnd** pair results in undefined behavior.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glCallList

glColor

glEdgeFlag

glEnd

glEvalC

o
o
r
d

g
l
I
n
d
e
x

g
l
M
a
t
e
r
i
a
l

g
l
N
o
r
m
a
l

g
l
R
e
c
t

g
l
T
e
x
C
o
o
r
d

glVertex2fv function

2/22/2020 • 2 minutes to read • Edit Online

Specifies a vertex.

Syntax

```
void WINAPI glVertex2fv(
   const GLfloat *v
);
```

Parameters

v	A pointer to an array of two elements. The elements are the x and y coordinates of a vertex.

Return value

This function does not return a value.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

glCallList

glColor

glEdgeFlag

glEnd

glEvalCoord

glIn

d
e
x

g
l
M
a
t
e
r
i
a
l

g
l
N
o
r
m
a
l

g
l
R
e
c
t

g
l
T
e
x
C
o
o
r
d

glVertex2i function

2/22/2020 • 2 minutes to read • Edit Online

Specifies a vertex.

Syntax

```
void WINAPI glVertex2i(
   GLint x,
   GLint y
);
```

Parameters

x	Specifies the x-coordinate of a vertex.
y	Specifies the y-coordinate of a vertex.

Return value

This function does not return a value.

Remarks

The glVertex function commands are used within **glBegin**/**glEnd** pairs to specify point, line, and polygon vertices. The current color, normal, and texture coordinates are associated with the vertex when glVertex is called. When only *x* and *y* are specified, *z* defaults to 0.0 and *w* defaults to 1.0. When *x*, *y*, and *z* are specified, *w* defaults to 1.0. Invoking glVertex outside of a **glBegin**/**glEnd** pair results in undefined behavior.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glCallList

glColor

glEdgeFlag

glEnd

glEvalC

o
o
r
d

g
l
I
n
d
e
x

g
l
M
a
t
e
r
i
a
l

g
l
N
o
r
m
a
l

g
l
R
e
c
t

g
l
T
e
x
C
o
o
r
d

glVertex2iv function

2/22/2020 • 2 minutes to read • Edit Online

Specifies a vertex.

Syntax

```
void WINAPI glVertex2iv(
   const GLint *v
);
```

Parameters

v	A pointer to an array of two elements. The elements are the x and y coordinates of a vertex.

Return value

This function does not return a value.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

glCallList

glColor

glEdgeFlag

glEnd

glEvalCoord

glIn

d
e
x

g
l
M
a
t
e
r
i
a
l

g
l
N
o
r
m
a
l

g
l
R
e
c
t

g
l
T
e
x
C
o
o
r
d

glVertex2s function

2/22/2020 • 2 minutes to read • Edit Online

Specifies a vertex.

Syntax

```
void WINAPI glVertex2s(
   GLshort x,
   GLshort y
);
```

Parameters

x	Specifies the x-coordinate of a vertex.
y	Specifies the y-coordinate of a vertex.

Return value

This function does not return a value.

Remarks

The glVertex function commands are used within **glBegin**/**glEnd** pairs to specify point, line, and polygon vertices. The current color, normal, and texture coordinates are associated with the vertex when glVertex is called. When only *x* and *y* are specified, *z* defaults to 0.0 and *w* defaults to 1.0. When *x*, *y*, and *z* are specified, *w* defaults to 1.0. Invoking glVertex outside of a **glBegin**/**glEnd** pair results in undefined behavior.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glCallList

glColor

glEdgeFlag

glEnd

glEvalC

o
o
r
d

g
l
I
n
d
e
x

g
l
M
a
t
e
r
i
a
l

g
l
N
o
r
m
a
l

g
l
R
e
c
t

g
l
T
e
x
C
o
o
r
d

glVertex2sv function

2/22/2020 • 2 minutes to read • Edit Online

Specifies a vertex.

Syntax

```
void WINAPI glVertex2sv(
   const GLshort *v
);
```

Parameters

v	A pointer to an array of two elements. The elements are the x and y coordinates of a vertex.

Return value

This function does not return a value.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

glCallList

glColor

glEdgeFlag

glEnd

glEvalCoord

glIn

d
e
x

g
l
M
a
t
e
r
i
a
l

g
l
N
o
r
m
a
l

g
l
R
e
c
t

g
l
T
e
x
C
o
o
r
d

glVertex3d function

2/22/2020 • 2 minutes to read • Edit Online

Specifies a vertex.

Syntax

```
void WINAPI glVertex3d(
   GLdouble x,
   GLdouble y,
   GLdouble z
);
```

Parameters

x	Specifies the x-coordinate of a vertex.
y	Specifies the y-coordinate of a vertex.
z	Specifies the z-coordinate of a vertex.

Return value

This function does not return a value.

Remarks

The glVertex function commands are used within **glBegin**/**glEnd** pairs to specify point, line, and polygon vertices. The current color, normal, and texture coordinates are associated with the vertex when glVertex is called. When only *x* and *y* are specified, *z* defaults to 0.0 and *w* defaults to 1.0. When *x*, *y*, and *z* are specified, *w* defaults to 1.0. Invoking glVertex outside of a **glBegin**/**glEnd** pair results in undefined behavior.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Ope ngl3 2.lib

DLL	Opengl32.dll

See also

glBegin

glCallList

glColor

glEdgeFlag

glEnd

glEvalCoord

glIndex

glMaterial

glNormal

glRect

glTexC

o
o
r
d

glVertex3dv function

2/22/2020 • 2 minutes to read • Edit Online

Specifies a vertex.

Syntax

```
void WINAPI glVertex3dv(
   const GLdouble *v
);
```

Parameters

v	A pointer to an array of three elements. The elements are the x, y, and z coordinates of a vertex.

Return value

This function does not return a value.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

glCallList

glColor

glEdgeFlag

glEnd

glEvalCoord

glIn

d
e
x

g
l
M
a
t
e
r
i
a
l

g
l
N
o
r
m
a
l

g
l
R
e
c
t

g
l
T
e
x
C
o
o
r
d

glVertex3f function

2/22/2020 • 2 minutes to read • Edit Online

Specifies a vertex.

Syntax

```
void WINAPI glVertex3f(
   GLfloat x,
   GLfloat y,
   GLfloat z
);
```

Parameters

x	Specifies the x-coordinate of a vertex.
y	Specifies the y-coordinate of a vertex.
z	Specifies the z-coordinate of a vertex.

Return value

This function does not return a value.

Remarks

The glVertex function commands are used within **glBegin**/**glEnd** pairs to specify point, line, and polygon vertices. The current color, normal, and texture coordinates are associated with the vertex when glVertex is called. When only *x* and *y* are specified, *z* defaults to 0.0 and *w* defaults to 1.0. When *x*, *y*, and *z* are specified, *w* defaults to 1.0. Invoking glVertex outside of a **glBegin**/**glEnd** pair results in undefined behavior.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib

DLL	Opengl32.dll

See also

glBegin

glCallList

glColor

glEdgeFlag

glEnd

glEvalCoord

glIndex

glMaterial

glNormal

glRect

glTexC

o
o
r
d

glVertex3fv function

2/22/2020 • 2 minutes to read • Edit Online

Specifies a vertex.

Syntax

```
void WINAPI glVertex3fv(
   const GLfloat *v
);
```

Parameters

v	A pointer to an array of three elements. The elements are the x, y, and z coordinates of a vertex.

Return value

This function does not return a value.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

glCallList

glColor

glEdgeFlag

glEnd

glEvalCoord

glIn

d
e
x

g
l
M
a
t
e
r
i
a
l

g
l
N
o
r
m
a
l

g
l
R
e
c
t

g
l
T
e
x
C
o
o
r
d

glVertex3i function

2/22/2020 • 2 minutes to read • Edit Online

Specifies a vertex.

Syntax

```
void WINAPI glVertex3i(
   GLint x,
   GLint y,
   GLint z
);
```

Parameters

x	Specifies the x-coordinate of a vertex.
y	Specifies the y-coordinate of a vertex.
z	Specifies the z-coordinate of a vertex.

Return value

This function does not return a value.

Remarks

The glVertex function commands are used within **glBegin**/**glEnd** pairs to specify point, line, and polygon vertices. The current color, normal, and texture coordinates are associated with the vertex when glVertex is called. When only *x* and *y* are specified, *z* defaults to 0.0 and *w* defaults to 1.0. When *x*, *y*, and *z* are specified, *w* defaults to 1.0. Invoking glVertex outside of a **glBegin**/**glEnd** pair results in undefined behavior.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib

DLL	Opengl32.dll

See also

glBegin

glCallList

glColor

glEdgeFlag

glEnd

glEvalCoord

glIndex

glMaterial

glNormal

glRect

glTexC

o
o
r
d

glVertex3iv function

2/22/2020 • 2 minutes to read • Edit Online

Specifies a vertex.

Syntax

```
void WINAPI glVertex3iv(
   const GLint *v
);
```

Parameters

v	A pointer to an array of three elements. The elements are the x, y, and z coordinates of a vertex.

Return value

This function does not return a value.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

glCallList

glColor

glEdgeFlag

glEnd

glEvalCoord

glIn

d
e
x

g
l
M
a
t
e
r
i
a
l

g
l
N
o
r
m
a
l

g
l
R
e
c
t

g
l
T
e
x
C
o
o
r
d

glVertex3s function

2/22/2020 • 2 minutes to read • Edit Online

Specifies a vertex.

Syntax

```
void WINAPI glVertex3s(
   GLshort x,
   GLshort y,
   GLshort z
);
```

Parameters

x	Specifies the x-coordinate of a vertex.
y	Specifies the y-coordinate of a vertex.
z	Specifies the z-coordinate of a vertex.

Return value

This function does not return a value.

Remarks

The glVertex function commands are used within **glBegin**/**glEnd** pairs to specify point, line, and polygon vertices. The current color, normal, and texture coordinates are associated with the vertex when glVertex is called. When only *x* and *y* are specified, *z* defaults to 0.0 and *w* defaults to 1.0. When *x*, *y*, and *z* are specified, *w* defaults to 1.0. Invoking glVertex outside of a **glBegin**/**glEnd** pair results in undefined behavior.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib

DLL	Opengl32.dll

See also

glBegin

glCallList

glColor

glEdgeFlag

glEnd

glEvalCoord

glIndex

glMaterial

glNormal

glRect

glTexC

o
o
r
d

glVertex3sv function

2/22/2020 • 2 minutes to read • Edit Online

Specifies a vertex.

Syntax

```
void WINAPI glVertex3sv(
   const GLshort *v
);
```

Parameters

v	A pointer to an array of three elements. The elements are the x, y, and z coordinates of a vertex.

Return value

This function does not return a value.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

glCallList

glColor

glEdgeFlag

glEnd

glEvalCoord

glIn

d
e
x

g
l
M
a
t
e
r
i
a
l

g
l
N
o
r
m
a
l

g
l
R
e
c
t

g
l
T
e
x
C
o
o
r
d

glVertex4d function

2/22/2020 • 2 minutes to read • Edit Online

Specifies a vertex.

Syntax

```
void WINAPI glVertex4d(
   GLdouble x,
   GLdouble y,
   GLdouble z,
   GLdouble w
);
```

Parameters

x	Specifies the x-coordinate of a vertex.
y	Specifies the y-coordinate of a vertex.
z	Specifies the z-coordinate of a vertex.
w	Specifies the w-coordinate of a vertex.

Return value

This function does not return a value.

Remarks

The glVertex function commands are used within **glBegin**/**glEnd** pairs to specify point, line, and polygon vertices. The current color, normal, and texture coordinates are associated with the vertex when glVertex is called. When only *x* and *y* are specified, *z* defaults to 0.0 and *w* defaults to 1.0. When *x*, *y*, and *z* are specified, *w* defaults to 1.0. Invoking glVertex outside of a **glBegin**/**glEnd** pair results in undefined behavior.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glCallList

glColor

glEdgeFlag

g

lEnd

glEvalCoord

glIndex

glMaterial

glNormal

glRect

gl

TexCoord

glVertex4dv function

2/22/2020 • 2 minutes to read • Edit Online

Specifies a vertex.

Syntax

```
void WINAPI glVertex4dv(
   const GLdouble *v
);
```

Parameters

v	A pointer to an array of four elements. The elements are the x, y, z, and w coordinates of a vertex.

Return value

This function does not return a value.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

glCallList

glColor

glEdgeFlag

glEnd

glEvalCoord

glIn

d
e
x

g
l
M
a
t
e
r
i
a
l

g
l
N
o
r
m
a
l

g
l
R
e
c
t

g
l
T
e
x
C
o
o
r
d

glVertex4f function

2/22/2020 • 2 minutes to read • Edit Online

Specifies a vertex.

Syntax

```
void WINAPI glVertex4f(
   GLfloat x,
   GLfloat y,
   GLfloat z,
   GLfloat w
);
```

Parameters

x	Specifies the x-coordinate of a vertex.
y	Specifies the y-coordinate of a vertex.
z	Specifies the z-coordinate of a vertex.
w	Specifies the w-coordinate of a vertex.

Return value

This function does not return a value.

Remarks

The glVertex function commands are used within **glBegin**/**glEnd** pairs to specify point, line, and polygon vertices. The current color, normal, and texture coordinates are associated with the vertex when glVertex is called. When only *x* and *y* are specified, *z* defaults to 0.0 and *w* defaults to 1.0. When *x*, *y*, and *z* are specified, *w* defaults to 1.0. Invoking glVertex outside of a **glBegin**/**glEnd** pair results in undefined behavior.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glCallList

glColor

glEdgeFlag

g

lEnd

glEvalCoord

glIndex

glMaterial

glNormal

glRect

gl

TexCoord

glVertex4fv function

2/22/2020 • 2 minutes to read • Edit Online

Specifies a vertex.

Syntax

```
void WINAPI glVertex4fv(
   const GLfloat *v
);
```

Parameters

v	A pointer to an array of four elements. The elements are the x, y, z, and w coordinates of a vertex.

Return value

This function does not return a value.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

glCallList

glColor

glEdgeFlag

glEnd

glEvalCoord

glIn

d
e
x

g
l
M
a
t
e
r
i
a
l

g
l
N
o
r
m
a
l

g
l
R
e
c
t

g
l
T
e
x
C
o
o
r
d

glVertex4i function

2/22/2020 • 2 minutes to read • Edit Online

Specifies a vertex.

Syntax

```
void WINAPI glVertex4i(
   GLint x,
   GLint y,
   GLint z,
   GLint w
);
```

Parameters

x	Specifies the x-coordinate of a vertex.
y	Specifies the y-coordinate of a vertex.
z	Specifies the z-coordinate of a vertex.
w	Specifies the w-coordinate of a vertex.

Return value

This function does not return a value.

Remarks

The glVertex function commands are used within **glBegin**/**glEnd** pairs to specify point, line, and polygon vertices. The current color, normal, and texture coordinates are associated with the vertex when glVertex is called. When only *x* and *y* are specified, *z* defaults to 0.0 and *w* defaults to 1.0. When *x*, *y*, and *z* are specified, *w* defaults to 1.0. Invoking glVertex outside of a **glBegin**/**glEnd** pair results in undefined behavior.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glCallList

glColor

glEdgeFlag

g

lEnd

glEvalCoord

glIndex

glMaterial

glNormal

glRect

gl

TexCoord

glVertex4iv function

2/22/2020 • 2 minutes to read • Edit Online

Specifies a vertex.

Syntax

```
void WINAPI glVertex4iv(
   const GLint *v
);
```

Parameters

v	A pointer to an array of four elements. The elements are the x, y, z, and w coordinates of a vertex.

Return value

This function does not return a value.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

glCallList

glColor

glEdgeFlag

glEnd

glEvalCoord

glIn

d
e
x

g
l
M
a
t
e
r
i
a
l

g
l
N
o
r
m
a
l

g
l
R
e
c
t

g
l
T
e
x
C
o
o
r
d

glVertex4s function

2/22/2020 • 2 minutes to read • Edit Online

Specifies a vertex.

Syntax

```
void WINAPI glVertex4s(
   GLshort x,
   GLshort y,
   GLshort z,
   GLshort w
);
```

Parameters

x	Specifies the x-coordinate of a vertex.
y	Specifies the y-coordinate of a vertex.
z	Specifies the z-coordinate of a vertex.
w	Specifies the w-coordinate of a vertex.

Return value

This function does not return a value.

Remarks

The glVertex function commands are used within **glBegin**/**glEnd** pairs to specify point, line, and polygon vertices. The current color, normal, and texture coordinates are associated with the vertex when glVertex is called. When only *x* and *y* are specified, *z* defaults to 0.0 and *w* defaults to 1.0. When *x*, *y*, and *z* are specified, *w* defaults to 1.0. Invoking glVertex outside of a **glBegin**/**glEnd** pair results in undefined behavior.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glCallList

glColor

glEdgeFlag

g

lEnd

glEvalCoord

glIndex

glMaterial

glNormal

glRect

gl

TexCoord

glVertex4sv function

2/22/2020 • 2 minutes to read • Edit Online

Specifies a vertex.

Syntax

```
void WINAPI glVertex4sv(
   const GLshort *v
);
```

Parameters

v	A pointer to an array of four elements. The elements are the x, y, z, and w coordinates of a vertex.

Return value

This function does not return a value.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

n

glCallList

glColor

glEdgeFlag

glEnd

glEvalCoord

glIn

d
e
x

g
l
M
a
t
e
r
i
a
l

g
l
N
o
r
m
a
l

g
l
R
e
c
t

g
l
T
e
x
C
o
o
r
d

glVertexPointer function

2/22/2020 • 2 minutes to read • Edit Online

The **glVertexPointer** function defines an array of vertex data.

Syntax

```
void WINAPI glVertexPointer(
         GLint   size,
         GLenum  type,
         GLsizei stride,
   const GLvoid  *pointer
);
```

Parameters

size

The number of coordinates per vertex. The value of *size* must be 2, 3, or 4.

type

The data type of each coordinate in the array using the following symbolic constants: GL_SHORT, GL_INT, GL_FLOAT, and GL_DOUBLE.

stride

The byte offset between consecutive vertices. When *stride* is zero, the vertices are tightly packed in the array.

pointer

A pointer to the first coordinate of the first vertex in the array.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	*size* was not 2, 3, or 4.
GL_INVALID_ENUM	*type* was not an accepted value.
GL_INVALID_VALUE	*stride* or *count* was negative.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	*size* was not 2, 3, or 4.
GL_INVALID_ENUM	*type* was not an accepted value.

NAME	MEANING
GL_ INV ALI D_V AL UE	*stride* or *count* was negative.

Remarks

The **glVertexPointer** function specifies the location and data of an array of vertex coordinates to use when rendering. The *size* parameter specifies the number of coordinates per vertex. The *type* parameter specifies the data type of each vertex coordinate. The *stride* parameter determines the byte offset from one vertex to the next, enabling the packing of vertices and attributes in a single array or storage in separate arrays. In some implementations, storing the vertices and attributes in a single array can be more efficient than using separate arrays (see **glInterleavedArrays**).

A vertex array is enabled when you specify the GL_VERTEX_ARRAY constant with **glEnableClientState**. When enabled, **glDrawArrays**, **glDrawElements**, and **glArrayElement** use the vertex array. By default, the vertex array is disabled.

You cannot include **glVertexPointer** in display lists.

When you specify a vertex array using **glVertexPointer**, the values of all the function's vertex array parameters are saved in a client-side state, and static array elements can be cached. Because the vertex array parameters are client-side state, their values are not saved or restored by **glPushAttrib** and **glPopAttrib**.

Although no error is generated if you call **glVertexPointer** within **glBegin** and **glEnd** pairs, the results are undefined.

The following functions retrieve information related to **glVertexPointer**:

glGet with argument GL_VERTEX_ARRAY_SIZE

glGet with argument GL_VERTEX_ARRAY_STRIDE

glGet with argument GL_VERTEX_ARRAY_COUNT

glGet with argument GL_VERTEX_ARRAY_TYPE

glGetPointerv with argument GL_VERTEX_ARRAY_POINTER

glIsEnabled with argument GL_VERTEX_ARRAY

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h

Library	Opengl32.lib
DLL	Opengl32.dll

See also

glArrayElement

glColorPointer

glDrawArr

a
y
s

g
l
E
d
g
e
F
l
a
g
P
o
i
n
t
e
r

g
l
E
n
a
b
l
e
C
l
i
e
n
t
S
t
a
t
e

g
l
G
e
t
P
o
i
n
t
e

r
v

g
l
G
e
t
S
t
r
i
n
g

g
l
I
n
d
e
x
P
o
i
n
t
e
r

g
l
I
s
E
n
a
b
l
e
d

g
l
N
o
r
m
a
l
P
o
i

n
t
e
r

g
l
T
e
x
C
o
o
r
d
P
o
i
n
t
e
r

glViewport function

2/22/2020 • 2 minutes to read • Edit Online

The **glViewport** function sets the viewport.

Syntax

```
void WINAPI glViewport(
   GLint   x,
   GLint   y,
   GLsizei width,
   GLsizei height
);
```

Parameters

x The lower-left corner of the viewport rectangle, in pixels. The default is (0,0).

y The lower-left corner of the viewport rectangle, in pixels. The default is (0,0).

width

The width of the viewport. When an OpenGL context is first attached to a window, *width* and *height* are set to the dimensions of that window.

height

The height of the viewport. When an OpenGL context is first attached to a window, *width* and *height* are set to the dimensions of that window.

Return value

This function does not return a value.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	Either *width* or *height* was negative.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Error codes

The following error codes can be retrieved by the **glGetError** function.

NAME	MEANING
GL_INVALID_VALUE	Either *width* or *height* was negative.
GL_INVALID_OPERATION	The function was called between a call to **glBegin** and the corresponding call to **glEnd**.

Remarks

The **glViewport** function specifies the affine transformation of *x* and *y* from normalized device coordinates to window coordinates. Let (x_{nd} , y_{nd}) be normalized device coordinates. The window coordinates (x_w , y_w) are then computed as follows:

$$x_w = (x_{nd} + 1)\left(\frac{width}{2}\right) + x$$

$$y_w = (y_{nd} + 1)\left(\frac{height}{2}\right) + y$$

Viewport width and height are silently clamped to a range that depends on the implementation. This range is queried by calling **glGet** with argument GL_MAX_VIEWPORT_DIMS.

The following functions retrieve information related to **glViewport**:

glGet with argument GL_VIEWPORT

glGet with argument GL_MAX_VIEWPORT_DIMS

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Gl.h
Library	Opengl32.lib
DLL	Opengl32.dll

See also

glBegin

glDepthRange

GLU Functions

2/22/2020 • 2 minutes to read • Edit Online

This section includes reference pages, in alphabetical order, for all OpenGL Utility Library functions. For background information about these functions, see OpenGL Utility Library.

FUNCTION	DESCRIPTION
gluBeginCurve, gluEndCurve	Delimit a Non-Uniform Rational B-Spline (NURBS) curve definition.
gluBeginPolygon, gluEndPolygon	Delimit a polygon description.
gluBeginSurface, gluEndSurface	Delimit a NURBS surface definition.
gluBeginTrim, gluEndTrim	Delimit a NURBS trimming loop definition.
gluBuild1DMipmaps	Creates 1-D mipmaps.
gluBuild2DMipmaps	Creates 2-D mipmaps.
gluCylinder	Draws a cylinder.
gluDeleteNurbsRenderer	Destroys a NURBS object.
gluDeleteQuadric	Destroys a quadric object.
gluDeleteTess	Destroys a tessellation object.
gluDisk	Draws a disk.
gluErrorString	Produces an error string from an OpenGL or GLU error code. The error string is ANSI only.
gluGetNurbsProperty	Retrieves a NURBS property.
gluGetString	Retrieves a string that describes the GLU version number or supported GLU extension calls.
gluGetTessProperty	Retrieves a tessellation object property.
gluLoadSamplingMatrices	Loads NURBS sampling and culling matrices.
gluLookAt	Defines a viewing transformation.
gluNewNurbsRenderer	Creates a NURBS object.
gluNewQuadric	Creates a quadric object.

FUNCTION	DESCRIPTION
gluNewTess	Creates a tessellation object.
gluNextContour	Marks the beginning of another contour.
gluNurbsCallback	Defines a callback for a NURBS object.
gluNurbsCurve	Defines the shape of a NURBS curve.
gluNurbsProperty	Sets a NURBS property.
gluNurbsSurface	Defines the shape of a NURBS surface.
gluOrtho2D	Defines a 2-D orthographic projection matrix.
gluPartialDisk	Draws an arc of a disk.
gluPerspective	Sets up a perspective projection matrix.
gluPickMatrix	Defines a picking region.
gluProject	Maps object coordinates to window coordinates.
gluPwlCurve	Describes a piecewise linear NURBS trimming curve.
gluQuadricCallback	Defines a callback for a quadric object.
gluQuadricDrawStyle	Specifies the draw style desired for quadrics.
gluQuadricNormals	Specifies what kind of normals are to be used for quadrics.
gluQuadricOrientation	Specifies inside or outside orientation for quadrics.
gluQuadricTexture	Specifies whether quadrics are to be textured.
gluScaleImage	Scales an image to an arbitrary size.
gluSphere	Draws a sphere.
gluTessBeginContour, gluTessEndContour	Delimit a contour description.
gluTessBeginPolygon, gluTessEndPolygon	Delimit a polygon description.
gluTessCallback	Defines a callback for a tessellation object.
gluTessNormal	Specifies a normal for a polygon.
gluTessProperty	Sets the property of a tessellation object.
gluTessVertex	Specifies a vertex on a polygon.

FUNCTION	DESCRIPTION
gluUnProject	Maps window coordinates to object coordinates.

gluBeginCurve function

2/22/2020 • 2 minutes to read • Edit Online

The **gluBeginCurve** and **gluEndCurve** functions delimit a Non-Uniform Rational B-Spline (NURBS) curve definition.

Syntax

```
void WINAPI gluBeginCurve(
   GLUnurbs *nobj
);
```

Parameters

nobj	The NURBS object (created with **gluNewNurbsRenderer**).

Return value

This function does not return a value.

Remarks

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

OpenGL evaluators are used to render the NURBS curve as a series of line segments. Evaluator state is preserved during rendering with **glPushAttrib** (GL_EVAL_BIT) and **glPopAttrib**. For information on exactly what state these calls preserve, see **glPushAttrib**.

Examples

The following functions render a textured NURBS curve with normals; texture coordinates and normals are also specified as NURBS curves:

```
gluBeginCurve(nobj);
gluNurbsCurve(nobj, . . ., GL_MAP1_TEXTURE_COORD_2);
gluNurbsCurve(nobj, . . ., GL_MAP1_NORMAL);
gluNurbsCurve(nobj, . . ., GL_MAP1_VERTEX_4);
gluEndCurve(nobj);
```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

glPushAttrib

gluBeginSurface

gl

u
B
e
g
i
n
T
r
i
m

g
l
u
N
e
w
N
u
r
b
s
R
e
n
d
e
r
e
r

g
l
u
N
u
r
b
s
C
u
r
v
e

gluBeginPolygon function

2/22/2020 • 2 minutes to read • Edit Online

[The **gluBeginPolygon** function is obsolete and is provided for backward compatibility only. The **gluBeginPolygon** function is mapped to **gluTessBeginPolygon** followed by **gluTessBeginContour**.]

The **gluBeginPolygon** and **gluEndPolygon** functions delimit a polygon description.

Syntax

```
void WINAPI gluBeginPolygon(
   GLUtesselator *tess
);
```

Parameters

t *e* *s* *s*	The tessellation object (created with **gluNewTess**).

Return value

This function does not return a value.

Remarks

Use **gluBeginPolygon** and **gluEndPolygon** to delimit the definition of a nonconvex polygon.

1. Call **gluBeginPolygon**.
2. Define the contours of the polygon by calling **gluTessVertex** for each vertex and **gluNextContour** to start each new contour.
3. Call **gluEndPolygon** to signal the end of the definition.

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

Examples

The following example describes a quadrilateral with a triangular hole:

```
gluBeginPolygon(tess);
    gluTessVertex(tess, v1, v1);
    gluTessVertex(tess, v2, v2);
    gluTessVertex(tess, v3, v3);
    gluTessVertex(tess, v4, v4);
gluNextContour(tess, GLU_INTERIOR);
    gluTessVertex(tess, v5, v5);
    gluTessVertex(tess, v6, v6);
    gluTessVertex(tess, v7, v7);
gluEndPolygon(tess);
```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

gluNewTess

gluNextC

o
n
t
o
u
r

g
l
u
T
e
s
s
B
e
g
i
n
C
o
n
t
o
u
r

g
l
u
T
e
s
s
B
e
g
i
n
P
o
l
y
g
o
n

g
l
u
T
e
s

s
C
a
l
l
b
a
c
k

g
l
u
T
e
s
s
V
e
r
t
e
x

gluBeginSurface function

2/22/2020 • 2 minutes to read • Edit Online

The **gluBeginSurface** and **gluEndSurface** functions delimit a Non-Uniform Rational B-Spline (NURBS) surface definition.

Syntax

```
void WINAPI gluBeginSurface(
   GLUnurbs *nobj
);
```

Parameters

nobj	The NURBS object (created with **gluNewNurbsRenderer**).

Return value

This function does not return a value.

Remarks

The **gluBeginSurface** and **gluEndSurface** functions mark the beginning and end of NURBS surface definitions, which are defined with calls to **gluNurbsSurface**.

1. Call **gluBeginSurface** to mark the beginning of a NURBS surface definition.
2. Make one or more calls to **gluNurbsSurface** to define the attributes of the surface.

   Exactly one of these calls to **gluNurbsSurface** must have a surface type of GL_MAP2_VERTEX_3 or GL_MAP2_VERTEX_4.
3. To mark the end of the NURBS surface definition, call **gluEndSurface**.

The **gluBeginTrim**, **gluPwlCurve**, **gluNurbsCurve**, and **gluEndTrim** functions support trimming of NURBS surfaces.

Use OpenGL evaluators to render the NURBS surface as a set of polygons. Preserve the evaluator state during rendering with **glPushAttrib**(GL_EVAL_BIT) and **glPopAttrib**.

Examples

The following functions render a textured NURBS surface with normals; the texture coordinates and normals are also described as NURBS surfaces:

```
gluBeginSurface(nobj);
    gluNurbsSurface(nobj, . . ., GL_MAP2_TEXTURE_COORD_2);
    gluNurbsSurface(nobj, . . ., GL_MAP2_NORMAL);
    gluNurbsSurface(nobj, . . ., GL_MAP2_VERTEX_4);
gluEndSurface(nobj);
```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

gluBeginCurve

gluBeginT

r
i
m

g
l
u
N
e
w
N
u
r
b
s
R
e
n
d
e
r
e
r

g
l
u
N
u
r
b
s
C
u
r
v
e

g
l
u
N
u
r
b
s
S
u
r
f
a
c
e

g
l
u
P
w
l
C
u
r
v
e

gluBeginTrim function

2/22/2020 • 2 minutes to read • Edit Online

The **gluBeginTrim** and **gluEndTrim** functions delimit a Non-Uniform Rational B-Spline (NURBS) trimming loop definition.

Syntax

```
void WINAPI gluBeginTrim(
   GLUnurbs *nobj
);
```

Parameters

nobj	The NURBS object (created with **gluNewNurbsRenderer**).

Return value

This function does not return a value.

Remarks

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 write a definition for a NURBS surface that resembles 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 punched-out hole. 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 series of line segments that form a linear curve (see **gluPwlCurve**), 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 displayed area of the NURBS surface 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 forced 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). You can use nested trimming loops as long as the curve orientations alternate correctly. Trimming curves cannot be self-intersecting, nor can they intersect one another (or an error results).

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

Examples

This code fragment defines a trimming loop that consists of one piecewise linear curve and two NURBS curves:

```
gluBeginTrim(nobj);
    gluPwlCurve(. . ., GLU_MAP1_TRIM_2);
    gluNurbsCurve(. . ., GLU_MAP1_TRIM_2);
    gluNurbsCurve(. . ., GLU_MAP1_TRIM_3);
gluEndTrim(nobj);
```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

gluBeginS

u
r
f
a
c
e

g
l
u
E
n
d
S
u
r
f
a
c
e

g
l
u
N
e
w
N
u
r
b
s
R
e
n
d
e
r
e
r

g
l
u
N
u
r
b
s
C
a
l
l

b
a
c
k

g
l
u
N
u
r
b
s
C
u
r
v
e

g
l
u
P
w
l
C
u
r
v
e

gluBuild1DMipmaps function

2/22/2020 • 2 minutes to read • Edit Online

The **gluBuild1DMipmaps** function creates 1-D mipmaps.

Syntax

```
void WINAPI gluBuild1DMipmaps(
         GLenum target,
         GLint  components,
         GLint  width,
         GLenum format,
         GLenum type,
   const void   *data
);
```

Parameters

target

The target texture. Must be GL_TEXTURE_1D.

components

The number of color components in the texture. Must be 1, 2, 3, or 4.

width

The width of the texture image.

format

The format of the pixel data. The following values are valid: GL_COLOR_INDEX, GL_RED, GL_GREEN,

GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_BGR_EXT, GL_BGRA_EXT, GL_LUMINANCE, or GL_LUMINANCE_ALPHA.

type

The data type for *data*. The following values are valid: GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, or GL_FLOAT.

data A pointer to the image data in memory.

Return value

This function does not return a value.

Remarks

The **gluBuild1DMipmaps** function obtains the input image and generates all mipmap images (using **gluScaleImage**) so that the input image can be used as a mipmapped texture image. The **glTexImage1D** function is then called to load each of the images. If the width of the input image is not a power of two, then the image is scaled to the nearest power of two before the mipmaps are generated.

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

For a description of the acceptable values for the *format* parameter, see **glTexImage1D**. For a description of the acceptable values for the *type* parameter, see **glDrawPixels**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

glDrawPixels

glTexImage1D

gluBuild2DMipmaps

gluScal

e
l
m
a
g
e

gluBuild2DMipmaps function

2/22/2020 • 2 minutes to read • Edit Online

The **gluBuild2DMipmaps** function creates 2-D mipmaps.

Syntax

```
void WINAPI gluBuild2DMipmaps(
         GLenum target,
         GLint  components,
         GLint  width,
         GLInt  height,
         GLenum format,
         GLenum type,
   const void   *data
);
```

Parameters

target	The target texture. Must be GL_TEXTURE_2D.
components	The number of color components in the texture. Must be 1, 2, 3, or 4.
width	The width of the texture image.
height	The height of the texture image.

format

The format of the pixel data. Must be one of the following: GL_COLOR_INDEX, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_BGR_EXT, GL_BGRA_EXT, GL_LUMINANCE, or GL_LUMINANCE_ALPHA.

type

The data type for *data*. Must be one of the following: GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, or GL_FLOAT.

data A pointer to the image data in memory.

Return value

This function does not return a value.

Remarks

The **gluBuild2DMipmaps** function obtains the input image and generates all mipmap images (using **gluScaleImage**) so the input image can be used as a mipmapped texture image. To load each of the images, call **glTexImage2D**. If the dimensions of the input image are not powers of two, then the image is scaled so that both the width and height are powers of two before the mipmaps are generated.

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

For a description of the acceptable values for the *format* parameter, see **glTexImage2D**. For a description of the acceptable values for *type*, see **glDrawPixels**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu. h

Library	Glu 32.li b
DLL	Glu 32.d ll

See also

gl
D
r
a
w
P
i
x
e
l
s

gl
T
e
x
I
m
a
g
e
2
D

gl
u
B
u
i
l
d
1
D
M
i
p

m
a
p
s

g
l
u
S
c
a
l
e
I
m
a
g
e

gluCylinder function

2/22/2020 • 2 minutes to read • Edit Online

The **gluCylinder** function draws a cylinder.

Syntax

```
void WINAPI gluCylinder(
   GLUquadric *qobj,
   GLdouble   baseRadius,
   GLdouble   topRadius,
   GLdouble   height,
   GLint      slices,
   GLint      stacks
);
```

Parameters

qobj — The quadric object (created with **gluNewQuadric**).

baseRadius — The radius of the cylinder at *z* = 0.

topRadius — The radius of the cylinder at *z* = *height*.

height — The height of the cylinder.

slices	The number of subdivisions around the z-axis.
stacks	The number of subdivisions along the z-axis.

Return value

This function does not return a value.

Remarks

The **gluCylinder** function 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 *topRadius* is set to zero, then this routine will generate 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**): 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 positive y-axis, to 0.25 at the positive x-axis, to 0.5 at the negative y-axis, to 0.75 at the negative x-axis, and back to 1.0 at the positive y-axis.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

glu
Disk

glu
NewQuadric

glu
PartialDisk

glu
Quad

r
i
c
O
r
i
e
n
t
a
t
i
o
n

g
l
u
Q
u
a
d
r
i
c
T
e
x
t
u
r
e

g
l
u
S
p
h
e
r
e

gluDeleteNurbsRenderer function

2/22/2020 • 2 minutes to read • Edit Online

The **gluDeleteNurbsRenderer** function destroys a Non-Uniform Rational B-Spline (NURBS) object.

Syntax

```
void WINAPI gluDeleteNurbsRenderer(
   GLUnurbs *nobj
);
```

Parameters

nobj	The NURBS object to be destroyed (created with **gluNewNurbsRenderer**).

Return value

This function does not return a value.

Remarks

The **gluDeleteNurbsRenderer** function destroys the NURBS object and frees any memory that it used. After you have called **gluDeleteNurbsRenderer**, you cannot use *nobj* again.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

glu
NewNurbsRenderer

gluDeleteQuadric function

2/22/2020 • 2 minutes to read • Edit Online

The **gluDeleteQuadric** function destroys a quadric object.

Syntax

```
void WINAPI gluDeleteQuadric(
   GLUquadricObj *state
);
```

Parameters

state	The quadric object to be destroyed (created with **gluNewQuadric**).

Return value

This function does not return a value.

Remarks

The **gluDeleteQuadric** function destroys the quadric object and frees any memory that it used. After you have called **gluDeleteQuadric**, you cannot use *state* again.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib

DLL	Glu 32.d ll

See also

gluNewQuadric

gluDeleteTess function

2/22/2020 • 2 minutes to read • Edit Online

The **gluDeleteTess** function destroys a tessellation object.

Syntax

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

Parameters

t *e* *s* *s*	The tessellation object to destroy (created with gluNewTess).

Return value

This function does not return a value.

Remarks

The **gluDeleteTess** function destroys the indicated tessellation object and frees any memory that it used.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu. h
Library	Glu 32.li b
DLL	Glu 32.d ll

See also

glu
NewTess

glu
TessBeginPolygon

glu
TessCallback

gluDisk function

2/22/2020 • 2 minutes to read • Edit Online

The **gluDisk** function draws a disk.

Syntax

```
void WINAPI gluDisk(
   GLUquadric *qobj,
   GLdouble   innerRadius,
   GLdouble   outerRadius,
   GLint      slices,
   GLint      loops
);
```

Parameters

qobj — The quadric object (created with **gluNewQuadric**).

innerRadius — The inner radius of the disk (may be zero).

outerRadius — The outer radius of the disk.

slices — The number of subdivisions around the z-axis.

c e s	
l o o p s	The number of concentric rings about the origin into which the disk is subdivided.

Return value

This function does not return a value.

Remarks

The **gluDisk** function renders a disk on the *z* = 0 plane. The disk has a radius of *outerRadius*, and contains a concentric circular hole with a radius of *innerRadius*. If *innerRadius* 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 positive *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 positive z-axis.

If texturing is turned on (with **gluQuadricTexture**), texture coordinates are generated linearly such that where *r* = *outerRadius*, 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).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

g

l
u
C
y
l
i
n
d
e
r

g
l
u
N
e
w
Q
u
a
d
r
i
c

g
l
u
P
a
r
t
i
a
l
D
i
s
k

g
l
u
Q
u
a
d
r
i
c
O
r
i

e
n
t
a
t
i
o
n

g
l
u
Q
u
a
d
r
i
c
T
e
x
t
u
r
e

g
l
u
S
p
h
e
r
e

gluEndCurve function

2/22/2020 • 2 minutes to read • Edit Online

The **gluBeginCurve** and **gluEndCurve** functions delimit a Non-Uniform Rational B-Spline (NURBS) curve definition.

Syntax

```
void WINAPI gluEndCurve(
   GLUnurbs *nobj
);
```

Parameters

nobj	The NURBS object (created with **gluNewNurbsRenderer**).

Return value

This function does not return a value.

Remarks

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

OpenGL evaluators are used to render the NURBS curve as a series of line segments. Evaluator state is preserved during rendering with **glPushAttrib** (GL_EVAL_BIT) and **glPopAttrib**. For information on exactly what state these calls preserve, see **glPushAttrib**.

Examples

The following functions render a textured NURBS curve with normals; texture coordinates and normals are also specified as NURBS curves:

```
gluBeginCurve(nobj);
gluNurbsCurve(nobj, . . ., GL_MAP1_TEXTURE_COORD_2);
gluNurbsCurve(nobj, . . ., GL_MAP1_NORMAL);
gluNurbsCurve(nobj, . . ., GL_MAP1_VERTEX_4);
gluEndCurve(nobj);
```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu. h
Library	Glu 32.li b
DLL	Glu 32.d ll

See also

g
l
P
u
s
h
A
t
t
r
i
b

g
l
u
B
e
g
i
n
S
u
r
f
a
c
e

g
l

u
B
e
g
i
n
T
r
i
m

g
l
u
N
e
w
N
u
r
b
s
R
e
n
d
e
r
e
r

g
l
u
N
u
r
b
s
C
u
r
v
e

gluEndPolygon function

2/22/2020 • 2 minutes to read • Edit Online

[The **gluEndPolygon** function is obsolete and is provided for backward compatibility only. The **gluEndPolygon** function is mapped to **gluTessEndPolygon** followed by **gluTessEndContour**.]

The **gluBeginPolygon** and **gluEndPolygon** functions delimit a polygon description.

Syntax

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

Parameters

tess	The tessellation object (created with **gluNewTess**).

Return value

This function does not return a value.

Remarks

Use **gluBeginPolygon** and **gluEndPolygon** to delimit the definition of a nonconvex polygon.

1. Call **gluBeginPolygon**.
2. Define the contours of the polygon by calling **gluTessVertex** for each vertex and **gluNextContour** to start each new contour.
3. Call **gluEndPolygon** to signal the end of the definition.

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

Examples

The following example describes a quadrilateral with a triangular hole:

```
gluBeginPolygon(tess);
    gluTessVertex(tess, v1, v1);
    gluTessVertex(tess, v2, v2);
    gluTessVertex(tess, v3, v3);
    gluTessVertex(tess, v4, v4);
gluNextContour(tess, GLU_INTERIOR);
    gluTessVertex(tess, v5, v5);
    gluTessVertex(tess, v6, v6);
    gluTessVertex(tess, v7, v7);
gluEndPolygon(tess);
```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

gluNewTess

gluNextC

ontour

gluTessBeginContour

gluTessBeginPolygon

g/uTes

s
C
a
l
l
b
a
c
k

g
l
u
T
e
s
s
V
e
r
t
e
x

gluEndSurface function

2/22/2020 • 2 minutes to read • Edit Online

The **gluBeginSurface** and **gluEndSurface** functions delimit a Non-Uniform Rational B-Spline (NURBS) surface definition.

Syntax

```
void WINAPI gluEndSurface(
   GLUnurbs *nobj
);
```

Parameters

nobj	The NURBS object (created with **gluNewNurbsRenderer**).

Return value

This function does not return a value.

Remarks

The **gluBeginSurface** and **gluEndSurface** functions mark the beginning and end of NURBS surface definitions, which are defined with calls to **gluNurbsSurface**.

1. Call **gluBeginSurface** to mark the beginning of a NURBS surface definition.
2. Make one or more calls to **gluNurbsSurface** to define the attributes of the surface.

   Exactly one of these calls to **gluNurbsSurface** must have a surface type of GL_MAP2_VERTEX_3 or GL_MAP2_VERTEX_4.
3. To mark the end of the NURBS surface definition, call **gluEndSurface**.

The **gluBeginTrim**, **gluPwlCurve**, **gluNurbsCurve**, and **gluEndTrim** functions support trimming of NURBS surfaces.

Use OpenGL evaluators to render the NURBS surface as a set of polygons. Preserve the evaluator state during rendering with **glPushAttrib** (GL_EVAL_BIT) and **glPopAttrib**.

Examples

The following functions render a textured NURBS surface with normals; the texture coordinates and normals are also described as NURBS surfaces:

```
gluBeginSurface(nobj);
    gluNurbsSurface(nobj, . . ., GL_MAP2_TEXTURE_COORD_2);
    gluNurbsSurface(nobj, . . ., GL_MAP2_NORMAL);
    gluNurbsSurface(nobj, . . ., GL_MAP2_VERTEX_4);
gluEndSurface(nobj);
```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

gluBeginCurve

gluBeginT

r
i
m

g
l
u
N
e
w
N
u
r
b
s
R
e
n
d
e
r
e
r

g
l
u
N
u
r
b
s
C
u
r
v
e

g
l
u
N
u
r
b
s
S
u
r
f
a
c
e

g
l
u
P
w
l
C
u
r
v
e

gluEndTrim function

2/22/2020 • 2 minutes to read • Edit Online

The **gluBeginTrim** and **gluEndTrim** functions delimit a Non-Uniform Rational B-Spline (NURBS) trimming loop definition.

Syntax

```
void WINAPI gluEndTrim(
   GLUnurbs *nobj
);
```

Parameters

nobj	The NURBS object (created with **gluNewNurbsRenderer**).

Return value

This function does not return a value.

Remarks

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 write a definition for a NURBS surface that resembles 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 punched-out hole. 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 series of line segments that form a linear curve (see **gluPwlCurve**), 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 displayed area of the NURBS surface 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 forced 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). You can use nested trimming loops as long as the curve orientations alternate correctly. Trimming curves cannot be self-intersecting, nor can they intersect one another (or an error results).

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

Examples

This code fragment defines a trimming loop that consists of one piecewise linear curve and two NURBS curves:

```
gluBeginTrim(nobj);
    gluPwlCurve(. . ., GLU_MAP1_TRIM_2);
    gluNurbsCurve(. . ., GLU_MAP1_TRIM_2);
    gluNurbsCurve(. . ., GLU_MAP1_TRIM_3);
gluEndTrim(nobj);
```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

gluBeginSur

f
a
c
e

g
l
u
E
n
d
S
u
r
f
a
c
e

g
l
u
N
e
w
N
u
r
b
s
R
e
n
d
e
r
e
r

g
l
u
N
u
r
b
s
C
a
l
l
b
a

c
k

g
l
u
N
u
r
b
s
C
u
r
v
e

g
l
u
P
w
l
C
u
r
v
e

gluErrorString function

2/22/2020 • 2 minutes to read • Edit Online

The **gluErrorString** function produces an error string from an OpenGL or GLU error code. The error string is ANSI only.

Syntax

```
const GLubyte* WINAPI gluErrorString(
   GLenum errCode
);
```

Parameters

errCode	An OpenGL or GLU error code.

Remarks

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

The standard GLU error codes are GLU_INVALID_ENUM, GLU_INVALID_VALUE, and GLU_OUT_OF_MEMORY. Certain other GLU functions can return specialized error codes through callbacks. For the list of OpenGL error codes, see **glGetError**.

The **gluErrorString** function produces error strings in ANSI only. Whenever possible, use **gluErrorStringWIN**, which allows ANSI or Unicode error strings. This makes it easier to localize your program for use with another language.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h

Library	Glu 32.li b
DLL	Glu 32.d ll

See also

g
l
G
e
t
E
r
r
o
r

g
l
u
N
u
r
b
s
C
a
l
l
b
a
c
k

g
l
u
Q
u
a
d
r
i
c
C

a
l
l
b
a
c
k

g
l
u
T
e
s
s
C
a
l
l
b
a
c
k

gluGetNurbsProperty function

2/22/2020 • 2 minutes to read • Edit Online

The **gluGetNurbsProperty** function gets a Non-Uniform Rational B-Spline (NURBS) property.

Syntax

```
void WINAPI gluGetNurbsProperty(
   GLUnurbs *nobj,
   GLenum   property,
   GLfloat  *value
);
```

Parameters

nobj

The NURBS object (created with **gluNewNurbsRenderer**).

property

The property whose value is to be retrieved. The following values are valid: GLU_SAMPLING_TOLERANCE, GLU_DISPLAY_MODE, GLU_CULLING, GLU_AUTO_LOAD_MATRIX, GLU_PARAMETRIC_TOLERANCE, GLU_SAMPLING_METHOD, GLU_U_STEP, and GLU_V_STEP.

value

A pointer to the location into which the value of the named property is written.

Return value

This function does not return a value.

Remarks

Use **gluGetNurbsProperty** to retrieve properties stored in a NURBS object. These properties affect the way NURBS curves and surfaces are rendered. For information about NURBS properties, see **gluNurbsProperty**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu. h
Library	Glu 32.li b
DLL	Glu 32.d ll

See also

gluNewNurbsRenderer

gluNurbsP

r
o
p
e
r
t
y

gluGetString function

2/22/2020 • 2 minutes to read • Edit Online

The **gluGetString** function gets a string that describes the GLU version number or supported GLU extension calls.

Syntax

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

Parameters

name

Either the version number of GLU (GLU_VERSION) or available vendor-specific extension calls (GLU_EXTENSIONS).

Remarks

The **gluGetString** function returns a pointer to a static, null-terminated string. When *name* is GLU_VERSION, the returned string is a value that represents the version number of GLU. The format of the version number is as follows:

```
<version number><space><vendor-specific information>
(for example, "1.2.11 Microsoft Windows")
```

The version number has the form "major_number.minor_number" or "major_number.minor_number.release_number". The vendor-specific information is optional, and the format and contents depend on the implementation.

When *name* is GLU_EXTENSIONS, the returned string contains a list of names of supported GLU extensions that are separated by spaces. The format of the returned list of names is as follows:

```
<extension_name><space><extension_name><space> . . .
(for example, "GLU_NURBS GL_TESSELATION")
```

The extension names cannot contain any spaces.

The **gluGetString** function is valid for GLU version 1.1 or later.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu. h
Library	Glu 32.li b
DLL	Glu 32.d ll

gluGetTessProperty function

2/22/2020 • 2 minutes to read • Edit Online

The **gluGetTessProperty** function gets a tessellation object property.

Syntax

```
void WINAPI gluGetTessProperty(
   GLUtesselator *tess,
   GLenum        which,
   GLdouble      *value
);
```

Parameters

tess
The tessellation object (created with **gluNewTess**).

which
The property whose value is to be retrieved. The following values are valid: GLU_TESS_WINDING_RULE, GLU_TESS_BOUNDARY_ONLY, and GLU_TESS_TOLERANCE.

value
A pointer to the location where the value of the named property is written.

Return value

This function does not return a value.

Remarks

Use **gluGetTessProperty** to retrieve properties stored in a tessellation object. These properties affect the way tessellation objects are interpreted and rendered. For information about what the properties are and what they do, see **gluTessProperty**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

gluNewTess

gluTessProperty

gluLoadSamplingMatrices function

2/22/2020 • 2 minutes to read • Edit Online

The **gluLoadSamplingMatrices** function loads Non-Uniform Rational B-Spline (NURBS) sampling and culling matrices.

Syntax

```
void WINAPI gluLoadSamplingMatrices(
         GLUnurbs *nobj,
   const GLfloat  modelMatrix[16],
   const GLfloat  projMatrix[16],
   const GLint    viewport[4]
);
```

Parameters

nobj	The NURBS object (created with **gluNewNurbsRenderer**).
modelMatrix	A modelview matrix (as from a **glGetFloatv** call).
projMatrix	A projection matrix (as from a **glGetFloatv** call).
view	A viewport (as from a **glGetIntegerv** call).

p
o
r
t

Return value

This function does not return a value.

Remarks

The **gluLoadSamplingMatrices** function uses *modelMatrix*, *projMatrix*, and *viewport* to recompute the sampling and culling matrices stored in *nobj*. 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).

The **gluLoadSamplingMatrices** function 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, doing so necessitates a round trip to the OpenGL server to get the current values of the modelview matrix, projection matrix, and viewport.)

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu. h
Library	Glu 32.li b
DLL	Glu 32.d ll

See also

g
l
G
e
t
F
l

o
a
t
v

g
l
G
e
t
I
n
t
e
g
e
r
v

g
l
u
G
e
t
N
u
r
b
s
P
r
o
p
e
r
t
y

g
l
u
N
e
w
N
u
r
b
s
R
e
n

d
e
r
e
r

gluLookAt function

2/22/2020 • 2 minutes to read • Edit Online

The **gluLookAt** function defines a viewing transformation.

Syntax

```
void WINAPI gluLookAt(
   GLdouble eyex,
   GLdouble eyey,
   GLdouble eyez,
   GLdouble centerx,
   GLdouble centery,
   GLdouble centerz,
   GLdouble upx,
   GLdouble upy,
   GLdouble upz
);
```

Parameters

eyex — The position of the eye point.

eyey — The position of the eye point.

eyez — The position of the eye point.

centerx — The position of the reference point.

centery — The position of the reference point.

c e n t e r z	The position of the reference point.
u p x	The direction of the up vector.
u p y	The direction of the up vector.
u p z	The direction of the up vector.

Return value

This function does not return a value.

Remarks

The **gluLookAt** function 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, so that when you use a typical projection matrix, the center of the scene 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 to the reference point.

The matrix generated by **gluLookAt** postmultiplies the current matrix.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib

DLL	Glu 32.d ll

See also

glFrustum

gluPerspective

gluNewNurbsRenderer function

2/22/2020 • 2 minutes to read • Edit Online

The **gluNewNurbsRenderer** function creates a Non-Uniform Rational B-Spline (NURBS) object.

Syntax

```
GLUnurbs* WINAPI gluNewNurbsRenderer(void);
```

Parameters

This function has no parameters.

Remarks

The **gluNewNurbsRenderer** function creates and returns a pointer to a new NURBS object. Refer to this object when calling NURBS rendering and control functions. A return value of zero means there is not enough memory to allocate to the object.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

g
l
u
B
e

g
i
n
C
u
r
v
e

g
l
u
B
e
g
i
n
S
u
r
f
a
c
e

g
l
u
B
e
g
i
n
T
r
i
m

g
l
u
D
e
l
e
t
e
N
u
r
b
s
R

e
n
d
e
r
e
r

g
l
u
N
u
r
b
s
C
a
l
l
b
a
c
k

g
l
u
N
u
r
b
s
P
r
o
p
e
r
t
y

gluNewQuadric function

2/22/2020 • 2 minutes to read • Edit Online

The **gluNewQuadric** function creates a quadric object.

Syntax

```
GLUquadric* WINAPI gluNewQuadric(void);
```

Parameters

This function has no parameters.

Remarks

The **gluNewQuadric** function creates and returns a pointer to a new quadric object. Refer to this object when calling quadric rendering and control functions. A return value of zero means there is not enough memory to allocate to the object.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

gluCy

l
i
n
d
e
r

g
l
u
D
e
l
e
t
e
Q
u
a
d
r
i
c

g
l
u
D
i
s
k

g
l
u
P
a
r
t
i
a
l
D
i
s
k

g
l
u
Q
u
a

d
r
i
c
C
a
l
l
b
a
c
k

g
l
u
Q
u
a
d
r
i
c
D
r
a
w
S
t
y
l
e

g
l
u
Q
u
a
d
r
i
c
N
o
r
m
a
l
s

g
l

u
Q
u
a
d
r
i
c
O
r
i
e
n
t
a
t
i
o
n

g
l
u
Q
u
a
d
r
i
c
T
e
x
t
u
r
e

g
l
u
S
p
h
e
r
e

gluNewTess function

2/22/2020 • 2 minutes to read • Edit Online

The **gluNewTess** function creates a tessellation object.

Syntax

```
GLUtesselator* WINAPI gluNewTess(void);
```

Parameters

This function has no parameters.

Remarks

The **gluNewTess** function creates and returns a pointer to a new tessellation object. Refer to this object when calling tessellation functions. A return value of zero means there is not enough memory to allocate to the object.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

gluDe

l
e
t
e
T
e
s
s

g
l
u
T
e
s
s
B
e
g
i
n
P
o
l
y
g
o
n

g
l
u
T
e
s
s
C
a
l
l
b
a
c
k

gluNextContour function

2/22/2020 • 2 minutes to read • Edit Online

[The **gluNextContour** function is obsolete and is provided for backward compatibility only. The **gluNextContour** function is mapped to **gluTessEndContour** followed by **gluTessBeginContour**.]

The **gluNextContour** function marks the beginning of another contour.

Syntax

```
void WINAPI gluNextContour(
   GLUtesselator *tess,
   GLenum        type
);
```

Parameters

tess — The tessellation object (created with **gluNewTess**).

type — The type of the contour being defined. The following values are valid.

VALUE	MEANING
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 whether it is interior or exterior.

VALUE	MEANING
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.

Return value

This function does not return a value.

Remarks

Use the **gluNextContour** function to describe polygons with multiple contours. After you describe the first contour 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. Perform another series of **gluTessVertex** calls to describe the new contour. Repeat this process until all contours have been described.

The *type* parameter defines what type of contour follows.

To define the type of the first contour, you can call **gluNextContour** before describing the first contour. If you do not call **gluNextContour** before the first contour, the first contour is marked GLU_EXTERIOR.

Examples

You can describe a quadrilateral with a triangular hole in it as follows:

```
gluBeginPolygon(tess);
    gluTessVertex(tess, v1, v1);
    gluTessVertex(tess, v2, v2);
    gluTessVertex(tess, v3, v3);
    gluTessVertex(tess, v4, v4);
gluNextContour(tess, GLU_INTERIOR);
    gluTessVertex(tess, v5, v5);
    gluTessVertex(tess, v6, v6);
    gluTessVertex(tess, v7, v7);
gluEndPolygon(tess);
```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

gluNewTess

gluTessBeginContour

gl

u
T
e
s
s
B
e
g
i
n
P
o
l
y
g
o
n

g
l
u
T
e
s
s
C
a
l
l
b
a
c
k

g
l
u
T
e
s
s
E
n
d
C
o
n
t
o
u
r

g

I
u
T
e
s
s
V
e
r
t
e
x

gluNurbsCallback function

2/22/2020 • 2 minutes to read • Edit Online

The **gluNurbsCallback** function defines a callback for a Non-Uniform Rational B-Spline (NURBS) object.

Syntax

```
void WINAPI gluNurbsCallback(
   GLUnurbs *nobj,
   GLenum   which,
   void (CALLBACK *fn)()
);
```

Parameters

nobj
The NURBS object (created with **gluNewNurbsRenderer**).

which
The callback being defined. The only valid value is GLU_ERROR. The meaning of GLU_ERROR means that 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**.

fn
A pointer to the callback function.

Return value

This function does not return a value.

Remarks

Use **gluNurbsCallback** to define a callback to be used by a NURBS object. If the specified callback is already defined, it is replaced. If *fn* is **NULL**, then any existing callback is erased.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

gluErrorString

gluNewNurbsRenderer

r

gluNurbsCurve function

2/22/2020 • 2 minutes to read • Edit Online

The **gluNurbsCurve** function defines the shape of a Non-Uniform Rational B-Spline (NURBS) curve.

Syntax

```
void WINAPI gluNurbsCurve(
   GLUnurbs *nobj,
   GLint    nknots,
   GLfloat  *knot,
   GLint    stride,
   GLfloat  *ctlarray,
   GLint    order,
   GLenum   type
);
```

Parameters

nobj The NURBS object (created with **gluNewNurbsRenderer**).

nknots The number of knots in *knot*. The *nknots* parameter equals the number of control points plus the order.

knot An array of *nknots* nondecreasing knot values.

stride The offset (as a number of single-precision floating-point values) between successive curve control points.

ctl A pointer to an array of control points. The coordinates must agree with *type*.

array

order

The order of the NURBS curve. The *order* parameter equals degree + 1; hence a cubic curve has an order of 4.

type

The type of the curve. If this curve is defined within a **gluBeginCurve**/**gluEndCurve** pair, then the type can be any of the valid one-dimensional evaluator types (such as GL_MAP1_VERTEX_3 or GL_MAP1_COLOR_4). Between a **gluBeginTrim**/**gluEndTrim** pair, the only valid types are GLU_MAP1_TRIM_2 and GLU_MAP1_TRIM_3.

Return value

This function does not return a value.

Remarks

When **gluNurbsCurve** appears between a **gluBeginCurve**/**gluEndCurve** pair, it describes a curve to be rendered. You associate positional, texture, and color coordinates by presenting each as a separate **gluNurbsCurve** between a **gluBeginCurve**/**gluEndCurve** pair. Do not make more than one call to **gluNurbsCurve** for color, position, and texture data within a single **gluBeginCurve**/**gluEndCurve** pair. Make exactly one call to describe the position of the curve (a *type* of GL_MAP1_VERTEX_3 or GL_MAP1_VERTEX_4).

When **gluNurbsCurve** appears between a **gluBeginTrim**/**gluEndTrim** pair, it describes a trimming curve on a NURBS surface. If *type* is GLU_MAP1_TRIM_2, it describes a curve in two-dimensional (*u* and *v*) parameter space. If it is GLU_MAP1_TRIM_3, it describes a curve in two-dimensional homogeneous (*u*, *v*, and *w*) parameter space. For more discussion about trimming curves, see **gluBeginTrim**.

Examples

The following functions render a textured NURBS curve with normals:

```
gluBeginCurve(nobj);
    gluNurbsCurve(nobj, ..., GL_MAP1_TEXTURE_COORD_2);
    gluNurbsCurve(nobj, ..., GL_MAP1_NORMAL);
    gluNurbsCurve(nobj, ..., GL_MAP1_VERTEX_4);
gluEndCurve(nobj);
```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu. h
Library	Glu 32.li b
DLL	Glu 32.d ll

See also

g
l
u
B
e
g
i
n
C
u
r
v
e

g
l
u
B
e
g
i
n
T
r
i
m

g
l

u
E
n
d
C
u
r
v
e

g
l
u
E
n
d
T
r
i
m

g
l
u
N
e
w
N
u
r
b
s
R
e
n
d
e
r
e
r

g
l
u
P
w
l
C
u
r
v
e

gluNurbsProperty function

2/22/2020 • 3 minutes to read • Edit Online

The **gluNurbsProperty** function sets a Non-Uniform Rational B-Spline (NURBS) property.

Syntax

```
void WINAPI gluNurbsProperty(
   GLUnurbs *nobj,
   GLenum   property,
   GLfloat  value
);
```

Parameters

nobj

The NURBS object (created with **gluNewNurbsRenderer**).

property

The property to be set. The following values are valid:

VALUE	MEANING
GLU_SAMPLING_TOLERANCE	Specifies the maximum length, in pixels, to use when the sampling method is set to GLU_PATH_LENGTH. The default value is 50.0 pixels.

VALUE	MEANING
GLU_DISPLAY_MODE	The *value* parameter defines how a NURBS surface is to be rendered. You can set *value* to GLU_FILL, GLU_OUTLINE_POLYGON, or GLU_OUTLINE_PATCH. GLU_FILL. The surface is rendered as a set of polygons. This is the default value. GLU_OUTLINE_POLYGON. The NURBS library draws only the outlines of the polygons created by tessellation. GLU_OUTLINE_PATCH. Only the outlines of patches and trim curves defined by the user are drawn.
GLU_CULLING	The *value* parameter is a Boolean value. When value is set to GL_TRUE, NURBS curves whose control points lie outside the current viewport are discarded prior to tessellation. The default is GL_FALSE (because a NURBS curve cannot fall entirely within the convex hull of its control points).

VALUE	MEANING
GLU_AUTO_LOAD_MATRIX	The *value* parameter is a Boolean value. When set to GL_TRUE, the NURBS code downloads the projection matrix, modelview matrix, and viewport from the OpenGL server to compute sampling and culling matrices for each NURBS curve that is rendered. Sampling and culling matrices are required to determine the tessellation of a NURBS surface into line segments or polygons and to cull a NURBS surface if it lies outside of the viewport. If this mode is set to GL_FALSE, you must provide a projection matrix, modelview matrix, and viewport for the NURBS renderer to use to construct sampling and culling matrices. You can do this with the **gluLoadSamplingMatrices** function. The default for this mode is GL_TRUE. Changing this mode from GL_TRUE to GL_FALSE does not affect the sampling and culling matrices until you call **gluLoadSamplingMatrices**. The following property parameters are supported in GLU version 1.1 or later and are not valid for GLU version 1.0: GLU_PARAMETRIC_TOLERANCE, GLU_SAMPLING_METHOD, GLU_U_STEP, and GLU_V_STEP. The following value parameters are supported in GLU version 1.1 or later and are not valid for GLU version 1.0: GLU_PATH_LENGTH, GLU_PARAMETRIC_ERROR, and GLU_DOMAIN_DISTANCE.

VALUE	MEANING
GLU_PARAMETRIC_TOLERANCE	Specifies the maximum distance, in pixels, to use when the sampling method is set to GLU_PARAMETRIC_ERROR. The default value is 0.5.
GLU_SAMPLING_METHOD	Specifies how to tessallate a NURBS surface. GLU_SAMPLING_METHOD can have one of the following three values. GLU_PATH_LENGTH. The default value. Specifies that surfaces rendered with the maximum length, in pixels, of the edges of the tessellation polygons are no greater than the value specified by GLU_SAMPLING_TOLERANCE. GLU_PARAMETRIC_ERROR. Specifies that in rendering the surface, the value of GLU_PARAMETRIC_TOLERANCE specifies the maximum distance, in pixels, between the tessellation polygons and the surfaces they approximate. GLU_DOMAIN_DISTANCE. Specifies, in parametric coordinates, how many sample points per unit length to take in the *u* and *v* dimensions.
GLU_U_STEP	Specifies the number of sample points per unit length taken along the *u* dimension in parametric coordinates. The value of GLU_U_STEP is used when GLU_SAMPLING_METHOD is set to GLU_DOMAIN_DISTANCE. The default value is 100.

VALUE	MEANING
GLU_V_STEP	Specifies the number of sample points per unit length taken along the *v* dimension in parametric coordinates. The value of GLU_V_STEP is used when GLU_SAMPLING_METHOD is set to GLU_DOMAIN_DISTANCE. The default value is 100.

value

The value to which to set the indicated property. The *value* parameter can be a numeric value or one of the following three values: GLU_PATH_LENGTH, GLU_PARAMETRIC_ERROR, or GLU_DOMAIN_DISTANCE.

VALUE	MEANING
GLU_PATH_LENGTH	The default value. Specifies that surfaces rendered with the maximum length, in pixels, of the edges of the tessellation polygons are no greater than the value specified by GLU_SAMPLING_TOLERANCE.
GLU_PARAMETRIC_ERROR	Specifies that in rendering the surface, the value of GLU_PARAMETRIC_TOLERANCE specifies the maximum distance, in pixels, between the tessellation polygons and the surfaces they approximate.
GLU_DOMAIN_DISTANCE	Specifies, in parametric coordinates, how many sample points per unit length to take in the *u* and *v* dimensions.

Return value

This function does not return a value.

Remarks

Use **gluNurbsProperty** to control properties stored in a NURBS object. These properties affect the way a NURBS curve is rendered.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

gluGetNurbsProperty

gluG

e
t
S
t
r
i
n
g

g
l
u
L
o
a
d
S
a
m
p
l
i
n
g
M
a
t
r
i
c
e
s

g
l
u
N
e
w
N
u
r
b
s
R
e
n
d
e
r
e
r

gluNurbsSurface function

2/22/2020 • 2 minutes to read • Edit Online

The **gluNurbsSurface** function defines the shape of a Non-Uniform Rational B-Spline (NURBS) surface.

Syntax

```
void WINAPI gluNurbsSurface(
   GLUnurbs *nobj,
   GLint    sknot_count,
   float    *sknot,
   GLint    tknot_count,
   GLfloat  *tknot,
   GLint    s_stride,
   GLint    t_stride,
   GLfloat  *ctlarray,
   GLint    sorder,
   GLint    torder,
   GLenum   type
);
```

Parameters

nobj — The NURBS object (created with **gluNewNurbsRenderer**).

sknot_count — The number of knots in the parametric *u* direction.

sknot — An array of *sknot_count* nondecreasing knot values in the parametric *u* direction.

tknot — The number of knots in the parametric *v* direction.

tknot_count

An array of *tknot_count* nondecreasing knot values in the parametric *v* direction.

s_stride

The offset (as a number of single precisionfloating-point values) between successive control points in the parametric *u* direction in *ctlarray*.

t_stride

The offset (in single precisionfloating-point values) between successive control points in the parametric *v* direction in *ctlarray*.

ctlarray

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 *s_stride* and *t_stride*.

sorde

r

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.

t
o
r
d
e
r

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.

t
y
p
e

The type of the surface. The *type* parameter can be any of the valid two-dimensional evaluator types (such as GL_MAP2_VERTEX_3 or GL_MAP2_COLOR_4).

Return value

This function does not return a value.

Remarks

Use **gluNurbsSurface** within a NURBS 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** function. To mark the end of a NURBS surface definition, use the **gluEndSurface** function. Call **gluNurbsSurface** within a NURBS surface definition only.

You associate positional, texture, and color coordinates with a surface by presenting each as a separate **gluNurbsSurface** between a **gluBeginSurface/gluEndSurface** pair. Within a single **gluBeginSurface/gluEndSurface** pair, you can make only one call to **gluNurbsSurface** for color, position, and texture data. Make exactly one call to describe the position of the surface (a *type* of GL_MAP2_VERTEX_3 or GL_MAP2_VERTEX_4).

You can trim a NURBS surface by using the **gluNurbsCurve** and **gluPwlCurve** functions between calls to **gluBeginTrim** and **gluEndTrim**.

A **gluNurbsSurface** with *sknot_count* knots in the *u* direction and *tknot_count* knots in the *v* direction with orders *sorder* and *torder* must have (*sknot_count* - *sorder*) multipied by (*tknot_count* - *torder*) control points.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

glu
Begin
Surface

glu
Begin
Trim

glu
End

S
u
r
f
a
c
e

g
l
u
E
n
d
T
r
i
m

g
l
u
N
e
w
N
u
r
b
s
R
e
n
d
e
r
e
r

g
l
u
N
u
r
b
s
C
u
r
v
e

g
l
u
P
w
l
C
u
r
v
e

gluOrtho2D function

2/22/2020 • 2 minutes to read • Edit Online

The **gluOrtho2D** function defines a 2-D orthographic projection matrix.

Syntax

```
void WINAPI gluOrtho2D(
   GLdouble left,
   GLdouble right,
   GLdouble top,
   GLdouble bottom
);
```

Parameters

left	The coordinate for the left vertical clipping plane.
right	The coordinate for the right vertical clipping plane.
top	The coordinate for the top horizontal clipping plane.
bottom	The coordinate for the bottom horizontal clipping plane.

Return value

This function does not return a value.

Remarks

The **gluOrtho2D** function sets up a two-dimensional orthographic viewing region. This is equivalent to calling **glOrtho** with near = 1 and far = 1.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu. h
Library	Glu 32.li b
DLL	Glu 32.d ll

See also

glOrtho

gluPerspective

gluPartialDisk function

2/22/2020 • 2 minutes to read • Edit Online

The **gluPartialDisk** function draws an arc of a disk.

Syntax

```
void WINAPI gluPartialDisk(
   GLUquadric *qobj,
   GLdouble   innerRadius,
   GLdouble   outerRadius,
   GLint      slices,
   GLint      loops,
   GLdouble   startAngle,
   GLdouble   sweepAngle
);
```

Parameters

qobj	A quadric object (created with **gluNewQuadric**).
innerRadius	The inner radius of the partial disk (can be zero).
outerRadius	The outer radius of the partial disk.
slices	The number of subdivisions around the z-axis.

slices	
loops	The number of concentric rings about the origin into which the partial disk is subdivided.
startAngle	The starting angle, in degrees, of the disk portion.
sweepAngle	The sweep angle, in degrees, of the disk portion.

Return value

This function does not return a value.

Remarks

The **gluPartialDisk** function 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 *startAngle* through *startAngle* + *sweepAngle* is included (where 0 degrees is along the positive y-axis, 90 degrees is along the positive x-axis, 180 degrees is along the negative y-axis, and 270 degrees is along the negative x-axis).

The partial disk has a radius of *outerRadius* and contains a concentric circular hole with a radius of *innerRadius*. If *innerRadius* is zero, 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 positive 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 positive z-axis.

If you have turned on texturing (with **gluQuadricTexture**), **gluPartialDisk** generates texture coordinates linearly such that where *r* = *outerRadius*, 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).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

gluCylinder

gluDisk

gluNewQ

u
a
d
r
i
c

g
l
u
Q
u
a
d
r
i
c
O
r
i
e
n
t
a
t
i
o
n

g
l
u
Q
u
a
d
r
i
c
T
e
x
t
u
r
e

g
l
u
S
p
h

e

r

e

gluPerspective function

2/22/2020 • 2 minutes to read • Edit Online

The **gluPerspective** function sets up a perspective projection matrix.

Syntax

```
void WINAPI gluPerspective(
   GLdouble fovy,
   GLdouble aspect,
   GLdouble zNear,
   GLdouble zFar
);
```

Parameters

fovy

The field of view angle, in degrees, in the y-direction.

aspect

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

The distance from the viewer to the near clipping plane (always positive).

zFar

The distance from the viewer to the far clipping plane (always positive).

Return value

This function does not return a value.

Remarks

The **gluPerspective** function specifies a viewing frustum into the world coordinate system. In general, the aspect

ratio in **gluPerspective** should match the aspect ratio of the associated viewport. For example, *aspect* = 2.0 means the viewer's angle of view is twice as wide in *x* as it is in *y*. If the viewport is twice as wide as it is tall, it displays the image without distortion.

The matrix generated by **gluPerspective** is multiplied by the current matrix, just as if **glMultMatrix** were called with the generated matrix. To load the perspective matrix onto the current matrix stack instead, precede the call to **gluPerspective** with a call to **glLoadIdentity**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

glFrustum

glLoadIdent

i
t
y

g
l
M
u
l
t
M
a
t
r
i
x

g
l
u
O
r
t
h
o
2
D

gluPickMatrix function

2/22/2020 • 2 minutes to read • Edit Online

The **gluPickMatrix** function defines a picking region.

Syntax

```
void WINAPI gluPickMatrix(
   GLdouble x,
   GLdouble y,
   GLdouble height,
   GLdouble width,
   GLint    viewport[4]
);
```

Parameters

x	The x window coordinate of a picking region.
y	The y window coordinate of a picking region.
height	The height of the picking region in window coordinates.
width	The width of the picking region in window coordinates.
viewport	The current viewport (as from a **glGetIntegerv** call).

Return value

This function does not return a value.

Remarks

The **gluPickMatrix** function creates a projection matrix you can use to restrict drawing to a small region of the

viewport.

1. Use **gluPickMatrix** to restrict drawing to a small region around the cursor.
2. Enter selection mode (with **glRenderMode**), and then 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** were called with the generated matrix.

1. Call **glLoadIdentity** to load an identity matrix onto the perspective matrix stack.
2. Call **gluPickMatrix**.
3. Call a function (such as **gluPerspective**) to multiply the perspective matrix by the pick matrix.

When using **gluPickMatrix** to pick Non-Uniform Rational B-Spline (NURBS), be careful to turn off the NURBS property, GLU_AUTO_LOAD_MATRIX. If GLU_AUTO_LOAD_MATRIX is not turned off, any NURBS surface rendered is subdivided differently with the pick matrix from how it was subdivided without the pick matrix.

Examples

When rendering a scene as follows:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity( );
gluPerspective(. . .);
glMatrixMode(GL_MODELVIEW);
/* Draw the scene */
```

the following code selects a portion of the viewport:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity( );
gluPickMatrix(x, y, width, height, viewport);
gluPerspective(. . .);
glMatrixMode(GL_MODELVIEW);
/* Draw the scene */
```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu. h
Library	Glu 32.li b

DLL	Glu 32.d ll

See also

glGetIntegerv

glLoadIdentity

glMultMatrix

gl

RenderMode

glUperspective

gluProject function

2/22/2020 • 2 minutes to read • Edit Online

The **gluProject** function maps object coordinates to window coordinates.

Syntax

```
int WINAPI gluProject(
         GLdouble objx,
         GLdouble objy,
         GLdouble objz,
   const GLdouble modelMatrix[16],
   const GLdouble projMatrix[16],
   const GLint    viewport[4],
         GLdouble *winx,
         GLdouble *winy,
         GLdouble *winz
);
```

Parameters

objx	The x object coordinate.
objy	The y object coordinate.
objz	The z object coordinate.
modelMatrix	The current modelview matrix (as from a **glGetDoublev** call).
proj	The current projection matrix (as from a **glGetDoublev** call).

j
M
a
t
r
i
x

v
i
e
w
p
o
r
t

The current viewport (as from a **glGetIntegerv** call).

w
i
n
x

The computed x window coordinate.

w
i
n
y

The computed y window coordinate.

w
i
n
z

The computed z window coordinate.

Return value

If the function succeeds, the return value is GL_TRUE.

If the function fails, the return value is GL_FALSE.

Remarks

The **gluProject** function transforms the specified object coordinates into window coordinates using *modelMatrix*, *projMatrix*, and *viewport*. The result is stored in *winx*, *winy*, and *winz*.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu. h

Library	Glu 32.li b
DLL	Glu 32.d ll

See also

gl
G
e
t
D
o
u
b
l
e
v

gl
G
e
t
I
n
t
e
g
e
r
v

gl
u
U
n
P
r
o
j
e
c
t

gluPwlCurve function

2/22/2020 • 2 minutes to read • Edit Online

The **gluPwlCurve** function describes a piecewise linear Non-Uniform Rational B-Spline (NURBS) trimming curve.

Syntax

```
void WINAPI gluPwlCurve(
   GLUnurbs *nobj,
   GLint    count,
   GLfloat  *array,
   GLint    stride,
   GLenum   type
);
```

Parameters

nobj	The NURBS object (created with **gluNewNurbsRenderer**).
count	The number of points on the curve.
array	An array containing the curve points.
stride	The offset (a number of single-precision floating-point values) between points on the curve.
type	The type of curve. Must be either GLU_MAP1_TRIM_2 or GLU_MAP1_TRIM_3.

Return value

This function does not return a value.

Remarks

The **gluPwlCurve** function 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 real curve, the points should be close enough that the resulting path appears curved at the resolution used in the application.

If *type* is GLU_MAP1_TRIM_2, 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. For more information about trimming curves, see **gluBeginTrim**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

gluBeginCurve

gl

u
B
e
g
i
n
T
r
i
m

g
l
u
N
e
w
N
u
r
b
s
R
e
n
d
e
r
e
r

g
l
u
N
u
r
b
s
C
u
r
v
e

gluQuadricCallback function

2/22/2020 • 2 minutes to read • Edit Online

The **gluQuadricCallback** function defines a callback for a quadric object.

Syntax

```
void WINAPI gluQuadricCallback(
   GLUquadric *qobj,
   GLenum     which,
   void (CALLBACK *fn)()
);
```

Parameters

qobj

The quadric object (created with **gluNewQuadric**).

which

The callback being defined. The only valid value is GLU_ERROR.

VALUE	MEANING
GLU_ERROR	The **gluQuadricCallback** 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.

fn

The function to be called.

Return value

This function does not return a value.

Remarks

Use **gluQuadricCallback** to define a new callback to be used by a quadric object. If the specified callback is already defined, it is replaced. If *fn* is **NULL**, any existing callback is erased.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu. h
Library	Glu 32.li b
DLL	Glu 32.d ll

See also

gluErrorString

gluNewQuadric

gluQuadricDrawStyle function

2/22/2020 • 2 minutes to read • Edit Online

The **gluQuadricDrawStyle** function specifies the draw style desired for quadrics.

Syntax

```
void WINAPI gluQuadricDrawStyle(
   GLUquadric *quadObject,
   GLenum     drawStyle
);
```

Parameters

quadObject

The quadric object (created with **gluNewQuadric**).

drawStyle

The desired draw style. The following values are valid.

VALUE	MEANING
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.

VALUE	MEANING
GLU_POINT	Quadrics are rendered as a set of points.

Return value

This function does not return a value.

Remarks

The **gluQuadricDrawStyle** function specifies the draw style for quadrics rendered with **quadObject**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

gluNewQuadri

c

g
l
u
Q
u
a
d
r
i
c
N
o
r
m
a
l
s

g
l
u
Q
u
a
d
r
i
c
O
r
i
e
n
t
a
t
i
o
n

g
l
u
Q
u
a
d
r
i
c
T

e
x
t
u
r
e

gluQuadricNormals function

2/22/2020 • 2 minutes to read • Edit Online

The **gluQuadricNormals** function specifies what kind of normals are to be used for quadrics.

Syntax

```
void WINAPI gluQuadricNormals(
   GLUquadric *quadObject,
   GLenum     normals
);
```

Parameters

quadObject

The quadric object (created with **gluNewQuadric**).

normals

The desired type of normals. The following values are valid.

VALUE	MEANING
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 default value.

Return value

This function does not return a value.

Remarks

The **gluQuadricNormals** function specifies what kind of normals are to be used for quadrics rendered with **quadObject**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

gluNewQuadric

gluQuadr

i
c
D
r
a
w
S
t
y
l
e

g
l
u
Q
u
a
d
r
i
c
O
r
i
e
n
t
a
t
i
o
n

g
l
u
Q
u
a
d
r
i
c
T
e
x
t
u
r
e

gluQuadricOrientation function

2/22/2020 • 2 minutes to read • Edit Online

The **gluQuadricOrientation** function specifies inside or outside orientation for quadrics.

Syntax

```
void WINAPI gluQuadricOrientation(
   GLUquadric *quadObject,
   GLenum     orientation
);
```

Parameters

quadObject	The quadric object (created with **gluNewQuadric**).
orientation	The desired orientation. The following values are valid.

VALUE	MEANING
GLU_OUTSIDE	Draw quadrics with normals pointing outward. This is the default value.
GLU_INSIDE	Draw quadrics with normals pointing inward.

Return value

This function does not return a value.

Remarks

The **gluQuadricOrientation** function specifies what kind of orientation is desired for quadrics rendered with

quadObject. The interpretation of outward and inward depends on the quadric being drawn.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

gluNewQuadric

gluQuadricDra

w
S
t
y
l
e

g
l
u
Q
u
a
d
r
i
c
N
o
r
m
a
l
s

g
l
u
Q
u
a
d
r
i
c
T
e
x
t
u
r
e

gluQuadricTexture function

2/22/2020 • 2 minutes to read • Edit Online

The **gluQuadricTexture** function specifies whether quadrics are to be textured.

Syntax

```
void WINAPI gluQuadricTexture(
   GLUquadric *quadObject,
   GLboolean  textureCoords
);
```

Parameters

quadObject The quadric object (created with **gluNewQuadric**).

textureCoords A flag indicating whether texture coordinates are to be generated. The following values are valid.

VALUE	MEANING
GL_TRUE	Generate texture coordinates.
GL_FALSE	Do not generate texture coordinates. This is the default value.

Return value

This function does not return a value.

Remarks

The **gluQuadricTexture** function specifies whether texture coordinates are to be generated for quadrics rendered with **quadObject**.

The manner in which texture coordinates are generated depends upon the specific quadric rendered.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

gluNewQuadric

gluQuadricDra

w
S
t
y
l
e

g
l
u
Q
u
a
d
r
i
c
N
o
r
m
a
l
s

gluScaleImage function

2/22/2020 • 2 minutes to read • Edit Online

The **gluScaleImage** function scales an image to an arbitrary size.

Syntax

```
int WINAPI gluScaleImage(
         GLenum format,
         GLint  widthin,
         GLint  heightin,
         GLenum typein,
   const void   *datain,
         GLint  widthout,
         GLint  heightout,
         GLenum typeout,
         void   *dataout
);
```

Parameters

format

The format of the pixel data. The following symbolic values are valid: GL_COLOR_INDEX, GL_STENCIL_INDEX, GL_DEPTH_COMPONENT, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_BGR_EXT, GL_BGRA_EXT, GL_LUMINANCE, and GL_LUMINANCE_ALPHA.

widthin

The width of the source image that is scaled.

heightin

The height of the source image that is scaled.

typ

p
e
i
n

The data type for *datain*. Must be one of the following: GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, or GL_FLOAT.

d
a
t
a
i
n

A pointer to the source image.

w
i
d
t
h
o
u
t

The width of the destination image.

h
e
i
g
h
t
o
u
t

The height of the destination image.

t
y
p
e
o
u
t

The data type for *dataout*. Must be one of the following: GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, or GL_FLOAT.

d
a
t
a
o
u
t

A pointer to the destination image.

Return value

If the function succeeds, the return value is zero.

If the function fails, the return value is a GLU error code (see **gluErrorString**).

Remarks

The **gluScaleImage** function 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.

For a description of the acceptable values for the *format*, *typein*, and *typeout* parameters, see **glReadPixels**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

glDrawPixels

g

lReadPixels

glBuild1DMipmaps

glBuild2DMipmaps

glUEr

r
o
r
S
t
r
i
n
g

gluSphere function

2/22/2020 • 2 minutes to read • Edit Online

The **gluSphere** function draws a sphere.

Syntax

```
void WINAPI gluSphere(
   GLUquadric *qobj,
   GLdouble   radius,
   GLint      slices,
   GLint      stacks
);
```

Parameters

qobj	The quadric object (created with **gluNewQuadric**).
radius	The radius of the sphere.
slices	The number of subdivisions around the z-axis (similar to lines of longitude).
stacks	The number of subdivisions along the z-axis (similar to lines of latitude).

Return value

This function does not return a value.

Remarks

The **gluSphere** function 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**), 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**): 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 positive y-axis, to 0.25 at the positive x-axis, to 0.5 at the negative y-axis, to 0.75 at the negative x-axis, and back to 1.0 at the positive y-axis.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

gluCylinder

gluDisk

glu
NewQuadric

glu
PartialDisk

glu
QuadricOrientation

gl

u
Q
u
a
d
r
i
c
T
e
x
t
u
r
e

gluTessBeginContour function

2/22/2020 • 2 minutes to read • Edit Online

The **gluTessBeginContour** and **gluTessEndContour** functions delimit a contour description.

Syntax

```
void WINAPI gluTessBeginContour(
   GLUtesselator *tess
);
```

Parameters

tess	The tessellation object (created with **gluNewTess**).

Return value

This function does not return a value.

Remarks

The **gluTessBeginContour** and **gluTessEndPolygon** functions delimit the definition of a polygon contour. Within each **gluTessBeginContour/gluTessEndPolygon** pair, there can be zero or more calls to **gluTessVertex**. The vertexes specify a closed contour (the last vertex of each contour is automatically linked to the first). You can call **gluTessBeginContour** only between **gluTessBeginPolygon** and **gluTessEndPolygon**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib

DLL	Glu 32.d ll

See also

gluNewTess

gluTessBeginPolygon

gluTessCallba

c
k

glu
Tess
End
Poly
gon

glu
Tess
Norm
al

glu
Tess
Prop
ert
y

gl

u
T
e
s
s
V
e
r
t
e
x

gluTessBeginPolygon function

2/22/2020 • 2 minutes to read • Edit Online

The **gluTessBeginPolygon** and **gluTessEndPolygon** functions delimit a polygon description.

Syntax

```
void WINAPI gluTessBeginPolygon(
   GLUtesselator *tess,
   void          *polygon_data
);
```

Parameters

tess	The tessellation object (created with **gluNewTess**).
polygon_data	A pointer to a programmer-defined polygon data structure.

Return value

This function does not return a value.

Remarks

The **gluTessBeginPolygon** and **gluTessEndPolygon** functions delimit the definition of a nonconvex polygon. Within each **gluTessBeginPolygon** / **gluTessEndPolygon** pair, include one or more calls to **gluTessBeginContour**. Within each contour, there are zero or more calls to **gluTessVertex**. The vertexes specify a closed contour (the last vertex of each contour is automatically linked to the first).

The *polygon_data* parameter is a pointer to a programmer-defined data structure. If the appropriate callbacks are specified (see *gluTessCallback*), this pointer is returned to the callback function or functions, making it a convenient way to store per-polygon information.

When you call **gluTessEndPolygon**, the polygon is tessellated, and the resulting triangles are described through callbacks. For descriptions of the callback functions, see *gluTessCallback*.

Examples

The following describes a quadrilateral with a triangular hole:

```
gluTessBeginPolygon(tobj, NULL);
  gluTessBeginContour(tobj);
    gluTessVertex(tobj, v1, v1);
    gluTessVertex(tobj, v2, v2);
    gluTessVertex(tobj, v3, v3);
    gluTessVertex(tobj, v4, v4);
  gluTessEndContour(tobj);
  gluTessBeginContour(tobj);
    gluTessVertex(tobj, v5, v5);
    gluTessVertex(tobj, v6, v6);
    gluTessVertex(tobj, v7, v7);
  gluTessEndContour(tobj);
gluTessEndPolygon(tobj);
```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

gluNewTess

gl

u
T
e
s
s
B
e
g
i
n
C
o
n
t
o
u
r

g
l
u
T
e
s
s
C
a
l
l
b
a
c
k

g
l
u
T
e
s
s
E
n
d
C
o
n
t
o
u
r

g

lutessNormal

glutessProperty

glutessVertex

gluTessCallback function

2/22/2020 • 6 minutes to read • Edit Online

The **gluTessCallback** function defines a callback for a tessellation object.

Syntax

```
void WINAPI gluTessCallback(
   GLUtesselator *tess,
   GLenum        which,
   void (CALLBACK *fn)()
);
```

Parameters

tess The tessellation object (created with **gluNewTess**).

which 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.

For more information on these callbacks, see the following Remarks section.

fn The function to be called.

Return value

This function does not return a value.

Remarks

Use **gluTessCallback** to specify a callback to be used by a tessellation object. If the specified callback is already defined, then it is replaced. If *fn* is **NULL**, then the existing callback becomes undefined.

The tessellation object uses these callbacks to describe how a polygon that you specify is broken into triangles.

There are two versions of each callback, one with polygon data that you can define and one without. If both versions of a particular callback are specified, the callback with the polygon data you specify will be used. The

polygon_data parameter of **gluTessBeginPolygon** is a copy of the pointer that was specified when **gluTessBeginPolygon** was called.

The following are valid callbacks:

CALLBACK	DESCRIPTION
GLU_TESS_BEGIN	The GLU_TESS_BEGIN callback is invoked like **glBegin** to indicate the start of a (triangle) primitive. The function takes a single argument of type **GLenum**. If you set the GLU_TESS_BOUNDARY_ONLY property to GL_FALSE, the argument is set to either GL_TRIANGLE_FAN, GL_TRIANGLE_STRIP, or GL_TRIANGLES. If you set the GLU_TESS_BOUNDARY_ONLY property to GL_TRUE, the argument is set to GL_LINE_LOOP. The function prototype for this callback is as follows: **void begin** (**GLenum** *type*);
GLU_TESS_BEGIN_DATA	GLU_TESS_BEGIN_DATA is 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 you call **gluTessBeginPolygon**. The function prototype for this callback is: **void beginData** (**GLenum** *type*, **void** * *polygon_data*);
GLU_TESS_EDGE_FLAG	The GLU_TESS_EDGE_FLAG callback is similar to **glEdgeFlag**. The function takes a single Boolean flag that indicates which edges lie on the polygon boundary. If the flag is GL_TRUE, then each vertex that follows begins an edge that lies on the polygon boundary; that is, an edge which separates an interior region from an exterior one. If the flag is GL_FALSE, then each vertex that follows begins an edge that lies in the polygon interior. The GLU_TESS_EDGE_FLAG callback (if defined) is invoked before the first vertex callback is made. Because triangle fans and triangle strips do not support edge flags, the begin callback is not called with GL_TRIANGLE_FAN or GL_TRIANGLE_STRIP if an edge flag callback is provided. Instead, the fans and strips are converted to independent triangles. The function prototype for this callback is: **void edgeFlag** (**GLboolean** *flag*);
GLU_TESS_EDGE_FLAG_DATA	The GLU_TESS_EDGE_FLAG_DATA callback is 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 you call **gluTessBeginPolygon**. The function prototype for this callback is: **void edgeFlagData** (**GLboolean** *flag*, **void** * *polygon_data*);
GLU_TESS_VERTEX	The GLU_TESS_VERTEX callback is invoked between the begin and end callbacks. It is similar to glVertex , and it defines the vertexes 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 that you provided when you defined the vertex (see **gluTessVertex**). The function prototype for this callback is: **void vertex** (**void** * *vertex_data*);

CALLBACK	DESCRIPTION
GLU_TESS_VERTEX_DATA	The GLU_TESS_VERTEX_DATA is 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 you call **gluTessBeginPolygon**. The function prototype for this callback is: **void vertexData** (void * *vertex_data*, **void** * *polygon_data*);
GLU_TESS_END	The GLU_TESS_END callback serves the same purpose as **glEnd**. It indicates the end of a primitive, and it takes no arguments. The function prototype for this callback is: **void end** (**void**);
GLU_TESS_END_DATA	The GLU_TESS_END_DATA callback is 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 you call **gluTessBeginPolygon**. The function prototype for this callback is: **void endData** (**void** * *polygon_data*);

CALLBACK	DESCRIPTION

GLU_TESS_COMBINE	Call the GLU_TESS_COMBINE callback to create a new vertex when the tessellation detects an intersection, or 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.
	A pointer to a pointer.
	The function prototype for this callback is:
	void combine(**GLdouble** *coords*[3], **void** * *vertex_data*[4], **GLfloat** *weight*[4], **void** ** *outData*);
	The vertex is defined as a linear combination of up to four existing vertexes, stored in vertex_data. The coefficients of the linear combination are given by weight; these weights always sum to 1.0. All vertex pointers are valid even when some of the weights are zero. The coords parameter gives the location of the new vertex.
	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. Free the memory sometime after calling **gluTessEndPolygon**.
	For example, if the polygon lies in an arbitrary plane in three-dimensional space, and you associate a color with each vertex, the GLU_TESS_COMBINE callback might look like the following:
	
	`void myCombine( GLdouble coords[3], VERTEX *d[4],`
	`GLfloat w[4], VERTEX **dataOut`
	`)`
	`{`
	`VERTEX *newVertex = new_vertex();`
	`newVertex->x = coords[0];`
	`newVertex->y = coords[1];`
	`newVertex->z = coords[2];`
	`newVertex->r = w[0]*d[0]->r + w[1]*d[1]->r + w[2]*d[2]->r +`
	`w[3]*d[3]->r;`
	`newVertex->g = w[0]*d[0]->g + w[1]*d[1]->g + w[2]*d[2]->g +`
	`w[3]*d[3]->g;`
	`newVertex->b = w[0]*d[0]->b + w[1]*d[1]->b + w[2]*d[2]->b +`
	`w[3]*d[3]->b;`
	`newVertex->a = w[0]*d[0]->a + w[1]*d[1]->a + w[2]*d[2]->a +`
	`w[3]*d[3]->a;`
	`*dataOut = newVertex;`
	`}`
	
	When the tessellation detects an intersection, the GLU_TESS_COMBINE or GLU_TESS_COMBINE_DATA callback (see below) must be defined, and must write a non-**NULL** pointer into dataOut. Otherwise the GLU_TESS_NEED_COMBINE_CALLBACK error occurs, and no output is generated. (This is the only error that can occur during tessellation and rendering.)

CALLBACK	DESCRIPTION
GLU_TESS_COMBINE_DATA	The GLU_TESS_COMBINE_DATA callback is 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 you call **gluTessBeginPolygon**. The function prototype for this callback is: **void combineData** (**GLdouble** *coords*[3], **void** **vertex_data*[4], **GLfloat** *weight*[4], **void** ***outData*, **void** * *polygon_data*);
GLU_TESS_ERROR	The GLU_TESS_ERROR callback is called when an error is encountered. The one argument is of type **GLenum**; it indicates the specific error that occurred and is set to one of the following: GLU_TESS_MISSING_BEGIN_POLYGON GLU_TESS_MISSING_END_POLYGON GLU_TESS_MISSING_BEGIN_CONTOUR GLU_TESS_MISSING_END_CONTOUR GLU_TESS_COORD_TOO_LARGE GLU_TESS_NEED_COMBINE_CALLBACK Call gluErrorString to retrieve character strings describing these errors. The function prototype for this callback is as follows: **void error** (**GLenum** *errno*); The GLU library recovers from the first four errors by inserting the missing call or calls. 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 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 will be generated.
GLU_TESS_ERROR_DATA	The GLU_TESS_ERROR_DATA callback is 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 you call **gluTessBeginPolygon**. The function prototype for this callback is: **void errorData** (**GLenum** *errno*, **void** * *polygon_data*);

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu. h

Library	Glu32.lib
DLL	Glu32.dll

See also

glBegin

glEdgeFlag

glEnd

glVertex

gluDe

l
e
t
e
T
e
s
s

g
l
u
E
r
r
o
r
S
t
r
i
n
g

g
l
u
N
e
w
T
e
s
s

g
l
u
T
e
s
s
B
e
g
i
n
P
o
l
y
g
o

n

g
l
u
T
e
s
s
E
n
d
P
o
l
y
g
o
n

g
l
u
T
e
s
s
V
e
r
t
e
x

gluTessEndContour function

2/22/2020 • 2 minutes to read • Edit Online

The **gluTessBeginContour** and **gluTessEndContour** functions delimit a contour description.

Syntax

```
void WINAPI gluTessEndContour(
   GLUtesselator *tess
);
```

Parameters

t *e* *s* *s*	The tessellation object (created with **gluNewTess**).

Return value

This function does not return a value.

Remarks

The **gluTessBeginContour** and **gluTessEndContour** functions delimit the definition of a polygon contour. Within each **gluTessBeginContour**/**gluTessEndContour** pair, there can be zero or more calls to **gluTessVertex**. The vertexes specify a closed contour (the last vertex of each contour is automatically linked to the first). You can call **gluTessBeginContour** only between **gluTessBeginPolygon** and **gluTessEndPolygon**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu. h
Library	Glu 32.li b

DLL	Glu 32.d ll

See also

glu

NewTess

gluTessBeginPolygon

glu

TessCallba

c
k

glu
Tess
Endpolygon

glu
Tess
Normal

glu
Tess
Property

gl

u
T
e
s
s
V
e
r
t
e
x

gluTessEndPolygon function

2/22/2020 • 2 minutes to read • Edit Online

The **gluTessBeginPolygon** and **gluTessEndPolygon** functions delimit a polygon description.

Syntax

```
void WINAPI gluTessEndPolygon(
   GLUtesselator *tess
);
```

Parameters

t *e* *s* *s*	The tessellation object (created with **gluNewTess**).

Return value

This function does not return a value.

Remarks

The **gluTessBeginPolygon** and **gluTessEndPolygon** functions delimit the definition of a nonconvex polygon. Within each **gluTessBeginPolygon** / **gluTessEndPolygon** pair, include one or more calls to **gluTessBeginContour**. Within each contour, there are zero or more calls to **gluTessVertex**. The vertexes specify a closed contour (the last vertex of each contour is automatically linked to the first).

The *polygon_data* parameter is a pointer to a programmer-defined data structure. If the appropriate callbacks are specified (see *gluTessCallback*), this pointer is returned to the callback function or functions, making it a convenient way to store per-polygon information.

When you call **gluTessEndPolygon**, the polygon is tessellated, and the resulting triangles are described through callbacks. For descriptions of the callback functions, see *gluTessCallback*.

Examples

The following describes a quadrilateral with a triangular hole:

```
gluTessBeginPolygon(tobj, NULL);
  gluTessBeginContour(tobj);
    gluTessVertex(tobj, v1, v1);
    gluTessVertex(tobj, v2, v2);
    gluTessVertex(tobj, v3, v3);
    gluTessVertex(tobj, v4, v4);
  gluTessEndContour(tobj);
  gluTessBeginContour(tobj);
    gluTessVertex(tobj, v5, v5);
    gluTessVertex(tobj, v6, v6);
    gluTessVertex(tobj, v7, v7);
  gluTessEndContour(tobj);
gluTessEndPolygon(tobj);
```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

gluNewTess

gluTe

s
s
B
e
g
i
n
C
o
n
t
o
u
r

g
l
u
T
e
s
s
C
a
l
l
b
a
c
k

g
l
u
T
e
s
s
E
n
d
C
o
n
t
o
u
r

g
l
u
T

e
s
s
N
o
r
m
a
l

g
l
u
T
e
s
s
P
r
o
p
e
r
t
y

g
l
u
T
e
s
s
V
e
r
t
e
x

gluTessNormal function

2/22/2020 • 2 minutes to read • Edit Online

The **gluTessNormal** function specifies a normal for a polygon.

Syntax

```
void WINAPI gluTessNormal(
   GLUtesselator *tess,
   GLdouble      x,
   GLdouble      y,
   GLdouble      z
);
```

Parameters

t *e* *s* *s*	The tessellation object (created with **gluNewTess**).
x	The x-coordinate component of a normal.
y	The y-coordinate component of a normal.
z	The z-coordinate component of a normal.

Return value

This function does not return a value.

Remarks

The **gluTessNormal** function describes a normal for a polygon that you define. All input data is projected onto a plane perpendicular to one of the three coordinate axes before tessellation, and all output triangles are oriented counterclockwise with respect to the normal. (To obtain clockwise orientation, reverse 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 default value), the normal is determined as follows:

1. The direction of the normal, up to its sign, is found by fitting a plane to the vertexes, without regard to how the vertexes are connected. It is expected that the input data lies approximately in the plane; otherwise projection perpendicular to one of the three coordinate axes can change the geometry substantially.
2. The sign of the normal is chosen so that the sum of the signed areas of all input contours is nonnegative (where a counterclockwise contour has positive area).

The supplied normal persists until another call to **gluTessNormal** changes it.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu. h
Library	Glu 32.li b
DLL	Glu 32.d ll

See also

gluNewTess

gluTessBeginPolygon

g
l
u
T
e
s
s
E
n
d
P
o
l
y
g
o
n

gluTessProperty function

2/22/2020 • 2 minutes to read • Edit Online

The **gluTessProperty** function sets the property of a tessellation object.

Syntax

```
void WINAPI gluTessProperty(
   GLUtesselator *tess,
   GLenum        which,
   GLdouble      value
);
```

Parameters

tess

The tessellation object (created with **gluNewTess**).

which

The property value to set. The following values are valid: GLU_TESS_WINDING_RULE, GLU_TESS_BOUNDARY_ONLY, and GLU_TESS_TOLERANCE.

VALUE	MEANING

VALUE	MEANING
GLU_TESS_WINDING_RULE	Determines which parts of the polygon are on the interior. The value parameter may be set to one of the following: 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, first 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 counterclockwise 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 (GLU_TESS_WINDING_NONZERO) is another common way to define the interior. The other three rules (GLU_TESS_WINDING_POSITIVE, GLU_TESS_WINDING_NEGATIVE, GLU_TESS_WINDING_ABS_GEQ_TWO) are useful for polygon CSG operations.
GLU_TESS_BOUNDARY_ONLY	Specifies a Boolean value (set value to GL_TRUE or GL_FALSE). When you set value to GL_TRUE, a set of closed contours separating the polygon interior and exterior is returned instead of a tessellation. Exterior contours are oriented counterclockwise with respect to the normal; interior contours are oriented clockwise. 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 vertexes 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 can 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 default tolerance is zero. The current implementation merges vertexes 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.

value	The value of the indicated property.

Return value

This function does not return a value.

Remarks

The **gluTessProperty** function controls properties stored in a tessellation object. These properties affect the way the polygons are interpreted and rendered.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu.h
Library	Glu32.lib
DLL	Glu32.dll

See also

gluGetTessPro

p
e
r
t
y

g
l
u
N
e
w
T
e
s
s

gluTessVertex function

2/22/2020 • 2 minutes to read • Edit Online

The **gluTessVertex** function specifies a vertex on a polygon.

Syntax

```
void WINAPI gluTessVertex(
   GLUtesselator *tess,
   GLdouble      coords[3],
   void          *data
);
```

Parameters

tess	The tessellation object (created with **gluNewTess**).
coords	The location of the vertex.
data	An pointer passed back to the program with the vertex callback (as specified by *gluTessCallback*).

Return value

This function does not return a value.

Remarks

The **gluTessVertex** function describes a vertex on a polygon that the user is defining. Successive **gluTessVertex** calls describe a closed contour. For example, to describe a quadrilateral, call **gluTessVertex** four times. You can only call **gluTessVertex** between **gluTessBeginContour** and **gluTessEndContour**.

The *data* parameter 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 program through the GLU_VERTEX callback after tessellation (see *gluTessCallback*).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu. h
Library	Glu 32.li b
DLL	Glu 32.d ll

See also

gluNewTess

gluTessBeginContour

glutessBeginPolygon

glutessCallback

glutessEndContou

r

g
l
u
T
e
s
s
N
o
r
m
a
l

g
l
u
T
e
s
s
P
r
o
p
e
r
t
y

gluUnProject function

2/22/2020 • 2 minutes to read • Edit Online

The **gluUnProject** function maps window coordinates to object coordinates.

Syntax

```
int WINAPI gluUnProject(
         GLdouble winx,
         GLdouble winy,
         GLdouble winz,
   const GLdouble modelMatrix[16],
   const GLdouble projMatrix[16],
   const GLint    viewport[4],
         GLdouble *objx,
         GLdouble *objy,
         GLdouble *objz
);
```

Parameters

winx — The x window coordinate to be mapped.

winy — The y window coordinate to be mapped.

winz — The z window coordinate to be mapped.

modelMatrix — The modelview matrix (as from a **glGetDoublev** call).

projMatrix — The projection matrix (as from a **glGetDoublev** call).

j
M
a
t
r
i
x

v
i
e
w
p
o
r
t

The viewport (as from a **glGetIntegerv** call).

o
b
j
x

The computed x object coordinate.

o
b
j
y

The computed y object coordinate.

o
b
j
z

The computed z object coordinate.

Return value

If the function succeeds, the return value is GL_TRUE.

If the function fails, the return value is GL_FALSE.

Remarks

The **gluUnProject** function maps the specified window coordinates into object coordinates using *modelMatrix*, *projMatrix*, and *viewport*. The result is stored in *objx*, *objy*, and *objz*.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Glu. h

Library	Glu 32.li b
DLL	Glu 32.d ll

See also

glGet

glGetDoublev

glGetIntegerv

gluPro

j
e
c
t

Porting to OpenGL

2/22/2020 • 2 minutes to read • Edit Online

OpenGL is designed for compatibility across hardware and operating systems. This design makes it easier for programmers to port OpenGL programs from one system to another. While each operating system has unique requirements, much of the OpenGL code in your current programs can be used as is. To port your OpenGL application to Microsoft Windows, you'll have to modify your programs to work with the Windows windowing systems.

In general applications are ported to OpenGL for Windows from one of two platforms:

- OpenGL applications developed for the X Window System and the X library (Xlib)
- IRIS GL applications

For more information, see the following topics:

- Introduction to Porting to OpenGL for Windows
- OpenGL Functions and Their IRIS GL Equivalents
- IRIS GL and OpenGL Differences

Introduction to Porting to OpenGL for Windows

2/22/2020 • 2 minutes to read • Edit Online

The following topics describe how to port your applications from each of these platforms. The topics discuss porting OpenGL and window management code only; there is no discussion of other operating system port issues such as reading files, messaging, thread creation, and so on. This porting guide focuses on specific porting issues and assumes that you have an understanding of OpenGL and Windows programming.

- Porting X Window System Applications
- Translating the GLX library
- Porting Device Contexts and Pixel Formats
- Porting Rendering Contexts
- Porting GLX Pixmap Code
- Porting Other GLX Code
- A Porting Sample
- Porting Applications from IRIS GL
- Special IRIS GL Porting Issues

Porting X Window System Applications

2/22/2020 • 2 minutes to read • Edit Online

Like Windows, the X Window System is an event-handling, message-based system that uses window controls and menus. The OpenGL code in your X Window System application is probably located in areas that roughly correspond to where it will appear when you port it to Windows. Most of your OpenGL code will not change, but you must rewrite any code that is specific to the X Window System.

Use the following general procedure to port your X Window System OpenGL programs to Windows

1. Rewrite the X Window System specific code using equivalent Windows code. Locate window-creation and event-handling code. The X Window System and Windows are event-handling, message-based windowing systems, which makes it easier to determine where to make the appropriate changes. (However, especially for large applications, rewriting an application from one operating system to another can be a complex and difficult undertaking.)
2. Locate any code that uses GLX functions. These are the functions you'll translate to their equivalent Windows functions.
3. Translate GLX pixel format functions and Visual/Drawable functions to appropriate Windows/OpenGL pixel format and device context functions.
4. Translate GLX rendering context functions to Windows/OpenGL rendering context functions.
5. Translate GLX Pixmap functions to equivalent Windows functions.
6. Translate GLX framebuffer and other GLX functions to the appropriate Windows functions.

Translating the GLX Library

2/22/2020 • 2 minutes to read • Edit Online

OpenGL X Window System programs use the OpenGL Extension with the X Window System (GLX) library. The library is a set of functions and routines that initialize the pixel format, control rendering, and perform other OpenGL specific tasks. It connects the OpenGL library to the X Window System by managing window handles and rendering contexts. You must translate these functions to their equivalent Windows functions. The following table lists the X Window System GLX functions and their equivalent Windows functions.

GLX/XLIB FUNCTION	WINDOWS FUNCTION
glXChooseVisual	ChoosePixelFormat
glXCopyContext	Not applicable.
glXCreateContext	wglCreateContext
glXCreateGLXPixmap	CreateDIBitmapCreateDIBSection
glXDestroyContext	wglDeleteContext
glXDestroyGLXPixmap	DeleteObject
glXGetConfig	DescribePixelFormat
glXGetCurrentContext	wglGetCurrentContext
glXGetCurrentDrawable	wglGetCurrentDC
glXIsDirect	Not applicable.
glXMakeCurrent	wglMakeCurrent
glXQueryExtension	GetVersion
glXQueryVersion	GetVersion
glXSwapBuffers	SwapBuffers
glXUseXFont	wglUseFontBitmaps
XGetVisualInfo	GetPixelFormat
XCreateWindow	CreateWindow, CreateWindowEx, GetDC, BeginPaint
XSync	GdiFlush
Not applicable.	SetPixelFormat

Some GLX functions don't have an equivalent Windows function. To port these functions to Windows, rewrite your code to achieve the same functionality. For example, **glXWaitGL** has no equivalent Windows function but you can achieve the same result, executing any pending OpenGL commands, by calling glFinish.

The following topics describe how to port GLX functions that set the pixel format, and manage rendering contexts, pixmaps and bitmaps.

Porting Device Contexts and Pixel Formats

2/22/2020 • 2 minutes to read • Edit Online

Each window in the Microsoft implementation of OpenGL for Windows has its own current pixel format. A pixel format is defined by a **PIXELFORMATDESCRIPTOR** data structure. Because each window has its own pixel format, you obtain a device context, set the pixel format of the device context, and then create an OpenGL rendering context for the device context. The rendering context automatically uses the pixel format of its device context.

The X Window System also uses a data structure, **XVisualInfo**, to specify the properties of pixels in a window. **XVisualInfo** structures contain a **Visual** data structure that describes how color resources are used in a specific screen.

In the X Window System, **XVisualInfo** is used to create a window by setting the window to the pixel format you want. The returned structure is used to create the window and a rendering context. In Windows, you first create a window and get a handle to a device context (HDC) of the window. The HDC is then used to set the pixel format for the window. The rendering context uses the pixel format of the window.

The following table compares the X Window System and GLX visual functions with their equivalent Windows pixel format functions.

X WINDOW/GLX VISUAL FUNCTION	WINDOWS PIXEL FORMAT FUNCTION
XVisualInfo*`glXChooseVisual`(Display *dpy,int screen,int *attribList)	int `ChoosePixelFormat`(HDC hdc,PIXELFORMATDESCRIPTOR *ppfd)
int `glXGetConfig`(Display *dpy,XVisualInfo *vis,int *attribList,int *value)	int `DescribePixelFormat`(HDC hdc,int iPixelFormat,UINT nBytes,LPPIXELFORMATDESCRIPTOR ppfd)
XVisualInfo*`XGetVisualInfo`(Display *dpy,long vinfo_mask,XVisualInfo *vinfo_templ,int *nitems)	int `GetPixelFormat`(HDC hdc)
The *visual* returned by `glxChooseVisual` is used when a window is created.	BOOL `SetPixelFormat`(HDC hdc,int iPixelFormat,PIXELFORMATDESCRIPTOR *ppfd)

The following sections give examples of pixel format code fragments for an X Window System program, and the same code after it has been ported to Windows.

- GLX Pixel Format Code Sample
- Windows Pixel Format Code Sample

For more information on pixel formats, see Pixel Formats.

GLX Pixel Format Code Sample

2/22/2020 • 2 minutes to read • Edit Online

The code sample below shows how an X Window System OpenGL program uses GLX visual/pixel formatting functions.

```
/* X globals, defines, and prototypes */
Display *dpy;
Window glwin;
static int attributes[] = {GLX_DEPTH_SIZE, 16, GLX_DOUBLEBUFFER, None};

    /* find an OpenGL-capable Color Index visual with depth buffer */
    vi = glXChooseVisual(dpy, DefaultScreen(dpy), attributes);
    if (vi == NULL) {
        fprintf(stderr, "could not get visual\n");
        exit(1);
    }
```

The Visual can be used to create a window and a rendering context.

Windows Pixel Format Code Sample

2/22/2020 • 2 minutes to read • Edit Online

The following code sample shows a function that sets the pixel format using Windows functions:

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

    ppfd = &pfd;

    ppfd->nSize = sizeof(PIXELFORMATDESCRIPTOR);
    ppfd->nVersion = 1;
    ppfd->dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL |
                        PFD_DOUBLEBUFFER;
    ppfd->dwLayerMask = PFD_MAIN_PLANE;
    ppfd->iPixelType = PFD_TYPE_COLORINDEX;
    ppfd->cColorBits = 8;
    ppfd->cDepthBits = 16;
    ppfd->cAccumBits = 0;
    ppfd->cStencilBits = 0;

    pixelformat = ChoosePixelFormat(hdc, ppfd);

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

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

    return TRUE;
}
```

Porting Rendering Contexts

2/22/2020 • 2 minutes to read • Edit Online

The X Window System and Windows render through rendering contexts. Six GLX functions manage rendering contexts and five of them have an equivalent Windows function.

The following table lists the GLX rendering functions and their equivalent Windows functions.

GLX RENDERING CONTEXT FUNCTION	WINDOWS RENDERING CONTEXT FUNCTION
GLXContext**glXCopyContext**(Display **dpy*,GLXContext *src*,GLXContext *dst*,GLuint *mask*)	Not applicable.
GLXContext**glXCreateContext**(Display **dpy*,XVisualInfo **vis*,GLXContext *shareList*,Bool *direct*)	HGLRC **wglCreateContext**(HDC *hdc*)
void **glXDeleteContext**(Display **dpy*,GLXContext *ctx*)	BOOL **wglDeleteContext**(HGLRC *hglrc*)
GLXContext **glXGetCurrentContext**(*void*)	HGLRC **wglGetCurrentContext**(*VOID*)
GLXDrawable **glXGetCurrentDrawable**(*void*)	HDC **wglGetCurrentDC**(*VOID*)
Bool**glXMakeCurrent**(Display **dpy*,GLXDrawable *draw*,GLXContext *ctx*)	BOOL **wglMakeCurrent**(HDC *hdc*,HGLRC *hglrc*)

Return types and other types have different names in the X Window System than in Windows. You can search for occurrences of GLXContext to help find parts of your code that need to be ported.

The following sections compare rendering context code samples in an X Window System program and the same code after it has been ported to Windows.

For more information on rendering contexts, see Rendering Contexts.

GLX Rendering Context Code Sample

2/22/2020 • 2 minutes to read • Edit Online

The following code sample shows how an X Window System OpenGL program uses GLX rendering context functions.

```
Display *dpy;             /* display variable */
XVisualInfo *vi;          /* visual variable */
Window win;               /* window variable */
GLXDrawable drawable;     /* drawable variable */
GLXContext cx, cxTemp;    /* rendering context variables */

/* Code to open a display and get a visual. */

/* Create a GLX context. */
cx = glXCreateContext(dpy, vi, 0, GL_FALSE);
if (!cx) {
    fprintf(stderr, "Cannot create context.\n");
    exit(-1);
}

        .
/* Connect the context to the window. */
glXMakeCurrent(dpy, win, cx);

        .
/* When it's time to destroy the rendering context. . . */
cx = glXGetCurrentContext;
glXDestroyContext(dpy, cx);
```

Windows Rendering Context Code Sample

2/22/2020 • 2 minutes to read • Edit Online

The following code sample shows how the GLX rendering context code in the previous section looks when it has been ported to Windows.

```
HGLRC hRC;           // rendering context variable

/* Create and initialize a window */

/* Window message switch in a window procedure */
case WM_CREATE:      // Message when window is created
{
    HDC hDC, hDCTemp;    // device context handles

    /* Get the handle of the windows device context. */
    hDC = GetDC(hWnd);

    /* Create a rendering context and make it the current context */
    hRC = wglCreateContext(hDC);
    if (!hRC)
    {
        MessageBox(NULL, "Cannot create context.", "Error", MB_OK);
        return FALSE;
    }
    wglMakeCurrent(hDC, hRC);
}
break;

        .
case WM_DESTROYED:   // Message when window is destroyed
{
    HGLRC hRC        // rendering context handle
    HDC   hDC;       // device context handle

    /* Release and free the device context and rendering context. */
    hDC = wglGetCurrentDC;
    hRC = wglGetCurrentContext;

    wglMakeCurrent(NULL, NULL);

    if (hRC)
        wglDeleteContext(hRC);

    if (hDC)
        ReleaseDC(hWnd, hDC);

    PostQuitMessage (0);
}
break;
```

Porting GLX Pixmap Code

2/22/2020 • 2 minutes to read • Edit Online

The X Window System uses *pixmaps*, which are off-screen virtual drawing surfaces in the form of a three-dimensional array of bits. You can think of a pixmap as a stack of bitmaps: a two-dimensional array of pixels with each pixel having a value from 0 to 2N1 where N is the depth of the pixmap.

For OpenGL programs you use the GLX functions, **glXCreateGLXPixmap** and **glXDestroyGLXPixmap**, to create and destroy GLX pixmaps used for off-screen rendering.

Windows uses device-independent bitmaps that serve the same function as X Window System pixmaps. Use the standard Windows bitmap functions to create and destroy bitmaps.

The following table lists the GLX pixmap functions and their equivalent Windows bitmap functions.

GLX PIXMAP AND FONT FUNCTION	WINDOWS BITMAP AND FONT FUNCTION
GLXPixmap**glXCreateGLXPixmap**(Display **dpy*,XVisualInfo **vis*,Pixmap *pixmap*)	HBITMAP CreateDIBitmap HDC *hdc*,LPBITMAPINFOHEADER *lpbmih*,DWORD *fdwInit*,CONST BYTE **lpbInit*,LPBITMAPINFO *lpbmi*,UINT *fuUsage***)**HBITMAP CreateDIBSection HDC *hdc*,LPBITMAPINFO *lpbmi*,DWORD *fInit*,DWORD *iUsage*)
void **glXDestroyGLXPixmap**(Display **dpy*,GLXPixmap *pix*)	BOOL DeleteObject(HGDIOBJ *hObject*)

Porting Other GLX Code

2/22/2020 • 2 minutes to read • Edit Online

In addition to the Xlib and GLX functions described in the preceding sections, your program probably contains some of the other GLX or Xlib functions listed in Translating the GLX Library. Rewrite your X Window System code to Windows, substituting the appropriate functions.

A Porting Sample

2/22/2020 • 2 minutes to read • Edit Online

It's easier to understand how to modify your X Window System OpenGL program for Windows if you can compare before and after samples, and you can better see how the translated code is used in the proper context. This section presents an example of an X Window System OpenGL program, and then shows how the program looks after it has been ported to Windows. Note that the OpenGL code is the same in both programs.

- An X Window System OpenGL Program
- The Program Ported to Windows

An X Window System OpenGL Program

2/22/2020 • 3 minutes to read • Edit Online

The following program is an X Window System OpenGL program with the same OpenGL code used in the AUXEDEMO.C sample program supplied with the Microsoft Platform SDK. Compare this program with the Windows OpenGL program in The Program Ported to Windows.

```
/*
 * Example of an X Window System OpenGL program.
 * OpenGL code is taken from auxdemo.c in the Platform SDK
 */
#include <GL/glx.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <X11/keysym.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <stdio.h>

/* X globals, defines, and prototypes */
Display *dpy;
Window glwin;
static int attributes[] = {GLX_DEPTH_SIZE, 16, GLX_DOUBLEBUFFER, None};

#define SWAPBUFFERS glXSwapBuffers(dpy, glwin)
#define BLACK_INDEX     0
#define RED_INDEX       1
#define GREEN_INDEX     2
#define BLUE_INDEX      4
#define WIDTH           300
#define HEIGHT          200


/* OpenGL globals, defines, and prototypes */
GLfloat latitude, longitude, latinc, longinc;
GLdouble radius;

#define GLOBE    1
#define CYLINDER 2
#define CONE     3

GLvoid resize(GLsizei, GLsizei);
GLvoid initializeGL(GLsizei, GLsizei);
GLvoid drawScene(GLvoid);
void polarView( GLdouble, GLdouble, GLdouble, GLdouble);

static Bool WaitForMapNotify(Display *d, XEvent *e, char *arg)
{
    if ((e->type == MapNotify) && (e->xmap.window == (Window)arg)) {
    return GL_TRUE;
    }
    return GL_FALSE;
}

void
main(int argc, char **argv)
{
    XVisualInfo    *vi;
    Colormap        cmap;
    XSetWindowAttributes swa;
    GLXContext      cx;
    XEvent          event;
```

```
    XEvent          event;
    GLboolean       needRedraw = GL_FALSE, recalcModelView = GL_TRUE;
    int             dummy;

    dpy = XOpenDisplay(NULL);
    if (dpy == NULL){
        fprintf(stderr, "could not open display\n");
        exit(1);
    }

    if(!glXQueryExtension(dpy, &dummy, &dummy)){
        fprintf(stderr, "could not open display");
        exit(1);
    }

    /* find an OpenGL-capable Color Index visual with depth buffer */
    vi = glXChooseVisual(dpy, DefaultScreen(dpy), attributes);
    if (vi == NULL) {
        fprintf(stderr, "could not get visual\n");
        exit(1);
    }

    /* create an OpenGL rendering context */
    cx = glXCreateContext(dpy, vi,  None, GL_TRUE);
    if (cx == NULL) {
        fprintf(stderr, "could not create rendering context\n");
        exit(1);
    }

    /* create an X colormap since probably not using default visual */
    cmap = XCreateColormap(dpy, RootWindow(dpy, vi->screen),
                               vi->visual, AllocNone);
    swa.colormap = cmap;
    swa.border_pixel = 0;
    swa.event_mask = ExposureMask | KeyPressMask | StructureNotifyMask;
    glwin = XCreateWindow(dpy, RootWindow(dpy, vi->screen), 0, 0, WIDTH,
                        HEIGHT, 0, vi->depth, InputOutput, vi->visual,
                        CWBorderPixel | CWColormap | CWEventMask, &swa);
    XSetStandardProperties(dpy, glwin, "xogl", "xogl", None, argv,
                               argc, NULL);

    glXMakeCurrent(dpy, glwin, cx);

    XMapWindow(dpy, glwin);
    XIfEvent(dpy,  &event,  WaitForMapNotify,  (char *)glwin);

    initializeGL(WIDTH, HEIGHT);
    resize(WIDTH, HEIGHT);

    /* Animation loop */
    while (1) {
    KeySym key;

    while (XPending(dpy)) {
        XNextEvent(dpy, &event);
        switch (event.type) {
        case KeyPress:
                XLookupString((XKeyEvent *)&event, NULL, 0, &key, NULL);
        switch (key) {
        case XK_Left:
            longinc += 0.5;
            break;
        case XK_Right:
            longinc -= 0.5;
            break;
        case XK_Up:
            latinc += 0.5;
            break;
        case XK_Down:
```

```
            latinc -= 0.5;
            break;
        }
        break;
        case ConfigureNotify:
        resize(event.xconfigure.width, event.xconfigure.height);
        break;
        }
    }
    drawScene();
    }
}

/* OpenGL code */

GLvoid resize( GLsizei width, GLsizei height )
{
    GLfloat aspect;

    glViewport( 0, 0, width, height );

    aspect = (GLfloat) width / height;

    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluPerspective( 45.0, aspect, 3.0, 7.0 );
    glMatrixMode( GL_MODELVIEW );
}

GLvoid createObjects()
{
    GLUquadricObj *quadObj;

    glNewList(GLOBE, GL_COMPILE);
        quadObj = gluNewQuadric ();
        gluQuadricDrawStyle (quadObj, GLU_LINE);
        gluSphere (quadObj, 1.5, 16, 16);
    glEndList();

    glNewList(CONE, GL_COMPILE);
        quadObj = gluNewQuadric ();
        gluQuadricDrawStyle (quadObj, GLU_FILL);
        gluQuadricNormals (quadObj, GLU_SMOOTH);
        gluCylinder(quadObj, 0.3, 0.0, 0.6, 15, 10);
    glEndList();

    glNewList(CYLINDER, GL_COMPILE);
        glPushMatrix ();
        glRotatef ((GLfloat)90.0, (GLfloat)1.0, (GLfloat)0.0, (GLfloat)0.0);
        glTranslatef ((GLfloat)0.0, (GLfloat)0.0, (GLfloat)-1.0);
        quadObj = gluNewQuadric ();
        gluQuadricDrawStyle (quadObj, GLU_FILL);
        gluQuadricNormals (quadObj, GLU_SMOOTH);
        gluCylinder (quadObj, 0.3, 0.3, 0.6, 12, 2);
        glPopMatrix ();
    glEndList();
}

GLvoid initializeGL(GLsizei width, GLsizei height)
{
    GLfloat    maxObjectSize, aspect;
    GLdouble    near_plane, far_plane;

    glClearIndex( (GLfloat)BLACK_INDEX);
    glClearDepth( 1.0 );

    glEnable(GL_DEPTH_TEST);

    glMatrixMode( GL_PROJECTION );
```

```
    aspect = (GLfloat) width / height;
    gluPerspective( 45.0, aspect, 3.0, 7.0 );
    glMatrixMode( GL_MODELVIEW );

    near_plane = 3.0;
    far_plane = 7.0;
    maxObjectSize = 3.0F;
    radius = near_plane + maxObjectSize/2.0;

    latitude = 0.0F;
    longitude = 0.0F;
    latinc = 6.0F;
    longinc = 2.5F;

    createObjects();
}

void polarView(GLdouble radius, GLdouble twist, GLdouble latitude,
           GLdouble longitude)
{
    glTranslated(0.0, 0.0, -radius);
    glRotated(-twist, 0.0, 0.0, 1.0);
    glRotated(-latitude, 1.0, 0.0, 0.0);
    glRotated(longitude, 0.0, 0.0, 1.0);

}

GLvoid drawScene(GLvoid)
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    glPushMatrix();

        latitude += latinc;
        longitude += longinc;

        polarView( radius, 0, latitude, longitude );

        glIndexi(RED_INDEX);
        glCallList(CONE);

        glIndexi(BLUE_INDEX);
        glCallList(GLOBE);

        glIndexi(GREEN_INDEX);
        glPushMatrix();
            glTranslatef(0.8F, -0.65F, 0.0F);
            glRotatef(30.0F, 1.0F, 0.5F, 1.0F);
            glCallList(CYLINDER);
        glPopMatrix();

    glPopMatrix();

    SWAPBUFFERS;
}
```

The Program Ported to Windows

2/22/2020 • 3 minutes to read • Edit Online

The following program is a Windows OpenGL program with the same OpenGL code used in the AUXDEMO.C sample program supplied with the Platform SDK. Compare this program with the X Window System OpenGL program in An X Window System OpenGL Program.

```
/*
 * Example of a Windows OpenGL program.
 * The OpenGL code is the same as that used in
 * the X Window System sample
 */
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>

/* Windows globals, defines, and prototypes */
CHAR szAppName[]="Win OpenGL";
HWND  ghWnd;
HDC   ghDC;
HGLRC ghRC;

#define SWAPBUFFERS SwapBuffers(ghDC)
#define BLACK_INDEX     0
#define RED_INDEX       13
#define GREEN_INDEX     14
#define BLUE_INDEX      16
#define WIDTH           300
#define HEIGHT          200

LONG WINAPI MainWndProc (HWND, UINT, WPARAM, LPARAM);
BOOL bSetupPixelFormat(HDC);

/* OpenGL globals, defines, and prototypes */
GLfloat latitude, longitude, latinc, longinc;
GLdouble radius;

#define GLOBE    1
#define CYLINDER 2
#define CONE     3

GLvoid resize(GLsizei, GLsizei);
GLvoid initializeGL(GLsizei, GLsizei);
GLvoid drawScene(GLvoid);
void polarView( GLdouble, GLdouble, GLdouble, GLdouble);

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    MSG        msg;
    WNDCLASS   wndclass;

    /* Register the frame class */
    wndclass.style         = 0;
    wndclass.lpfnWndProc   = (WNDPROC)MainWndProc;
    wndclass.cbClsExtra    = 0;
    wndclass.cbWndExtra    = 0;
    wndclass.hInstance     = hInstance;
    wndclass.hIcon         = LoadIcon (hInstance, szAppName);
    wndclass.hCursor       = LoadCursor (NULL,IDC_ARROW);
    wndclass.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
    wndclass.lpszMenuName  = szAppName;
    wndclass.lpszClassName = szAppName;
```

```
    wndclass.lpszClassName = szAppName;

    if (!RegisterClass (&wndclass) )
        return FALSE;

    /* Create the frame */
    ghWnd = CreateWindow (szAppName,
             "Generic OpenGL Sample",
         WS_OVERLAPPEDWINDOW | WS_CLIPSIBLINGS | WS_CLIPCHILDREN,
             CW_USEDEFAULT,
             CW_USEDEFAULT,
             WIDTH,
             HEIGHT,
             NULL,
             NULL,
             hInstance,
             NULL);

    /* make sure window was created */
    if (!ghWnd)
        return FALSE;

    /* show and update main window */
    ShowWindow (ghWnd, nCmdShow);

    UpdateWindow (ghWnd);

    /* animation loop */
    while (1) {
        /*
         *  Process all pending messages
         */

        while (PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE) == TRUE)
        {
            if (GetMessage(&msg, NULL, 0, 0) )
            {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
            } else {
                return TRUE;
            }
        }
        drawScene();
    }
}

/* main window procedure */
LONG WINAPI MainWndProc (
    HWND    hWnd,
    UINT    uMsg,
    WPARAM  wParam,
    LPARAM  lParam)
{
    LONG    lRet = 1;
    PAINTSTRUCT    ps;
    RECT rect;

    switch (uMsg) {

    case WM_CREATE:
        ghDC = GetDC(hWnd);
        if (!bSetupPixelFormat(ghDC))
            PostQuitMessage (0);

        ghRC = wglCreateContext(ghDC);
        wglMakeCurrent(ghDC, ghRC);
        GetClientRect(hWnd, &rect);
        initializeGL(rect.right, rect.bottom);
        break;
```

```
        break;

    case WM_PAINT:
        BeginPaint(hWnd, &ps);
        EndPaint(hWnd, &ps);
        break;

    case WM_SIZE:
        GetClientRect(hWnd, &rect);
        resize(rect.right, rect.bottom);
        break;

    case WM_CLOSE:
        if (ghRC)
            wglDeleteContext(ghRC);
        if (ghDC)
            ReleaseDC(hWnd, ghDC);
        ghRC = 0;
        ghDC = 0;

        DestroyWindow (hWnd);
        break;

    case WM_DESTROY:
        if (ghRC)
            wglDeleteContext(ghRC);
        if (ghDC)
            ReleaseDC(hWnd, ghDC);

        PostQuitMessage (0);
        break;

    case WM_KEYDOWN:
        switch (wParam) {
        case VK_LEFT:
            longinc += 0.5F;
            break;
        case VK_RIGHT:
            longinc -= 0.5F;
            break;
        case VK_UP:
            latinc += 0.5F;
            break;
        case VK_DOWN:
            latinc -= 0.5F;
            break;
        }

    default:
        lRet = DefWindowProc (hWnd, uMsg, wParam, lParam);
        break;
    }

    return lRet;
}

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

    ppfd = &pfd;

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

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

    pixelformat = ChoosePixelFormat(hdc, ppfd);

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

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

    return TRUE;
}

/* OpenGL code */

GLvoid resize( GLsizei width, GLsizei height )
{
    GLfloat aspect;

    glViewport( 0, 0, width, height );

    aspect = (GLfloat) width / height;

    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluPerspective( 45.0, aspect, 3.0, 7.0 );
    glMatrixMode( GL_MODELVIEW );
}

GLvoid createObjects()
{
    GLUquadricObj *quadObj;

    glNewList(GLOBE, GL_COMPILE);
        quadObj = gluNewQuadric ();
        gluQuadricDrawStyle (quadObj, GLU_LINE);
        gluSphere (quadObj, 1.5, 16, 16);
    glEndList();

    glNewList(CONE, GL_COMPILE);
        quadObj = gluNewQuadric ();
        gluQuadricDrawStyle (quadObj, GLU_FILL);
        gluQuadricNormals (quadObj, GLU_SMOOTH);
        gluCylinder(quadObj, 0.3, 0.0, 0.6, 15, 10);
    glEndList();

    glNewList(CYLINDER, GL_COMPILE);
        glPushMatrix ();
        glRotatef ((GLfloat)90.0, (GLfloat)1.0, (GLfloat)0.0, (GLfloat)0.0);
        glTranslatef ((GLfloat)0.0, (GLfloat)0.0, (GLfloat)-1.0);
        quadObj = gluNewQuadric ();
        gluQuadricDrawStyle (quadObj, GLU_FILL);
        gluQuadricNormals (quadObj, GLU_SMOOTH);
        gluCylinder (quadObj, 0.3, 0.3, 0.6, 12, 2);
        glPopMatrix ();
    glEndList();
}

GLvoid initializeGL(GLsizei width, GLsizei height)
{
```

```
    GLfloat     maxObjectSize, aspect;
    GLdouble    near_plane, far_plane;

    glClearIndex( (GLfloat)BLACK_INDEX);
    glClearDepth( 1.0 );

    glEnable(GL_DEPTH_TEST);

    glMatrixMode( GL_PROJECTION );
    aspect = (GLfloat) width / height;
    gluPerspective( 45.0, aspect, 3.0, 7.0 );
    glMatrixMode( GL_MODELVIEW );

    near_plane = 3.0;
    far_plane = 7.0;
    maxObjectSize = 3.0F;
    radius = near_plane + maxObjectSize/2.0;

    latitude = 0.0F;
    longitude = 0.0F;
    latinc = 6.0F;
    longinc = 2.5F;

    createObjects();
}

void polarView(GLdouble radius, GLdouble twist, GLdouble latitude,
           GLdouble longitude)
{
    glTranslated(0.0, 0.0, -radius);
    glRotated(-twist, 0.0, 0.0, 1.0);
    glRotated(-latitude, 1.0, 0.0, 0.0);
    glRotated(longitude, 0.0, 0.0, 1.0);

}

GLvoid drawScene(GLvoid)
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    glPushMatrix();

        latitude += latinc;
        longitude += longinc;

        polarView( radius, 0, latitude, longitude );

        glIndexi(RED_INDEX);
        glCallList(CONE);

        glIndexi(BLUE_INDEX);
        glCallList(GLOBE);

    glIndexi(GREEN_INDEX);
        glPushMatrix();
            glTranslatef(0.8F, -0.65F, 0.0F);
            glRotatef(30.0F, 1.0F, 0.5F, 1.0F);
            glCallList(CYLINDER);
        glPopMatrix();

    glPopMatrix();

    SWAPBUFFERS;
}
```

Porting Applications from IRIS GL

2/22/2020 • 2 minutes to read • Edit Online

This section lists important differences between IRIS GL and OpenGL and describes the basic steps for porting code from IRIS GL to OpenGL. For a complete list of the differences between IRIS GL and Open GL, see IRIS GL and OpenGL Differences.

Porting IRIS GL programs to OpenGL for Windows requires considerably more work than converting OpenGL programs from the X Window System. While IRIS GL programs are designed to run with specific hardware and software, OpenGL was designed for portability among various systems.

The following table lists some of the key differences between OpenGL and IRIS GL programs.

OPENGL CODE	IRIS GL CODE
Operating system independent; contains no functions for windowing, event handling, buffer allocation/management, and so on.	Dependent on operating system; windowing-system functions are mixed with rendering functions. There is no windows manager in IRIS GL.
Uses a standard, common naming convention. OpenGL functions and defined types begin with a "gl" prefix to prevent conflicts with other libraries.	Does not use a common naming convention for functions and defined types.
Manages state variables (such as color, fog, texture, lighting, and so on) directly and consistently. Does not use tables to load state-variable values.	Uses tables to manage state variables and must bind variables to table values.
Display lists cannot be edited.	Display lists can be edited.
Does not provide a file format for fonts.	Provides functions to handle fonts and text strings and a file format for fonts.
Includes a GL Utility (GLU) library that contains additional functions and routines (such as NURBS and quadratic rendering routines).	Does not support the GLU library.

Use the following general procedure to port your IRIS GL programs to OpenGL

1. Rewrite any code that makes calls to a window manager, window configuration, device, or event, or where you load a color map to equivalent Windows code. Rewriting an application from one operating system to another can be complex and difficult. This subject is beyond the scope of this section.
2. Locate any code that uses IRIS GL functions and routines. Translate these functions to their equivalent OpenGL functions. For a complete listing of IRIS GL functions and routines and their equivalent OpenGL counterparts, see OpenGL Functions and Their IRIS GL Equivalents.
3. Change IRIS GL code as described in Special IRIS GL Porting Issues.

1. Rewrite any code that makes calls to a window manager, window configuration, device, or event, or where you load a color map to equivalent Windows code. Rewriting an application from one operating system to another can be complex and difficult. This subject is beyond the scope of this section.
2. Locate any code that uses IRIS GL functions and routines. Translate these functions to their equivalent OpenGL

functions. For a complete listing of IRIS GL functions and routines and their equivalent OpenGL counterparts, see OpenGL Functions and Their IRIS GL Equivalents.

3. Change IRIS GL code as described in Special IRIS GL Porting Issues.

Special IRIS GL Porting Issues

2/22/2020 • 2 minutes to read • Edit Online

The following topics describe techniques for porting specific parts of your IRIS GL code to OpenGL code.

- Porting greset
- Porting IRIS GL "Get" Functions
- Porting Code that Requires a Current Graphics Position
- Porting Drawing Functions
- Porting Color, Shading, and Writemask Code
- Porting Pixel Operations
- Porting Depth Cueing and Fog Commands
- Porting Curve and Surface Functions
- Porting Antialiasing Functions
- Porting Accumulation Buffer Calls
- Porting Stencil Plane Calls
- Porting Display Lists
- Porting Defs, Binds, and Sets
- Porting Lighting and Materials Functions
- Porting Texture Functions
- Porting Picking Functions
- Porting Feedback Functions

Porting greset

2/22/2020 • 2 minutes to read • Edit Online

OpenGL replaces the IRIS GL function, **greset**, with the functions, **glPushAttrib** and **glPopAttrib**. Use these functions to save and restore groups of state variables. For example,

```
void glPushAttrib( GLbitfield mask );
```

This example takes a bitwise OR of symbolic constants, indicating which groups of state variables to push onto an attribute stack. Each constant refers to a group of state variables. The following table shows the attribute groups with their equivalent symbolic constant names. For a complete list of the OpenGL state variables associated with each constant, see **glPushAttrib**.

ATTRIBUTE	CONSTANT
accumulation buffer clear value	GL_ACCUM_BUFFER_BIT
color buffer	GL_COLOR_BUFFER_BIT
current	GL_CURRENT_BIT
depth buffer	GL_DEPTH_BUFFER_BIT
enable	GL_ENABLE_BIT
evaluators	EGL_VAL_BIT
fog	GL_FOG_BIT
GL_LIST_BASE setting	GL_LIST_BIT
hint variables	GL_HINT_BIT
lighting variables	GL_LIGHTING_BIT
line drawing mode	GL_LINE_BIT
pixel mode variables	GL_PIXEL_MODE_BIT
point variables	GL_POINT_BIT
polygon	GL_POLYGON_BIT
polygon stipple	GL_POLYGON_STIPPLE_BIT
scissor	GL_SCISSOR_BIT
stencil buffer	GL_STENCIL_BUFFER_BIT

ATTRIBUTE	CONSTANT
texture	GL_TEXTURE_BIT
transform	GL_TRANSFORM_BIT
viewport	GL_VIEWPORT_BIT
	GL_ALL_ATTRIB_BITS

To restore the values of the state variables to those saved with the last glPushAttrib, simply call glPopAttrib. The variables you didn't save will remain unchanged. The attribute stack has a finite depth of at least 16.

Porting IRIS GL Get Functions

2/22/2020 • 2 minutes to read • Edit Online

IRIS GL "get" functions take the following form:

```
int getthing();
```

and

```
int getthings( int *a, int *b);
```

Your IRIS GL code probably includes get function calls that look something like:

```
thing = getthing();
if (getthing() == THING) { /* some stuff here */ }
getthings (&a, &b);
```

In OpenGL you use one of the following four types of **glGet** functions in place of equivalent IRIS GL get functions:

- **glGetBooleanv**
- **glGetIntegerv**
- **glGetFloatv**
- **glGetDoublev**

The functions have the following syntax:

```
glGet<Datatype>v( value, *data );
```

where *value* is of type **GLenum** and data is of type **GLdatatype**. When you call **glGet** and it returns a type different from the type expected, the type is converted appropriately. For a complete list of **glGet** parameters, see **glGet**.

Port code that needs a current graphics position

2/22/2020 • 2 minutes to read • Edit Online

OpenGL does not maintain a current graphics position. IRIS GL functions that depend on the current graphics position, such as **move**, **draw**, and **rmv**, have no equivalents in OpenGL.

Older versions of IRIS GL included drawing commands that relied upon the current graphics position, though their use has been discouraged. You will need to rewrite your code if you relied on the current graphics position in any way, or used any of the following routines:

- **draw** and **move**
- **pmv**, **pdr**, and **pclos**
- **rdr**, **rmv**, **rpdr**, and **rpmv**
- **getgpos**

OpenGL has a concept of raster position that corresponds to IRIS GL's current character position. For more information on raster positioning, see Porting Pixel Operations.

This topic includes information on the following.

- Porting Screen and Buffer Clearing Commands
- Porting Matrix and Transformation Functions
- Porting MSINGLE Mode Code
- Porting Functions that Get Matrices and Transformations
- Porting Viewports, Screenmasks, and Scrboxes
- Porting Clipping Planes

Porting Screen and Buffer Clearing Commands

2/22/2020 • 2 minutes to read • Edit Online

OpenGL replaces a variety of IRIS GL **clear** functions (such as **zclear**, **aclear**, **sclear**, and so on) with a single function, **glClear**. Specify exactly what you want to clear by passing masks to **glClear**.

Keep the following points in mind when porting screen and buffer commands:

- OpenGL maintains clearing colors separately from drawing colors, with calls like **glClearColor** and **glClearIndex**. Be sure to set the clear color for each buffer before clearing.
- Instead of using one of several differently named clear calls, you now clear several buffers with one call, **glClear**, by **OR** ing together buffer masks. For example, **czclear** is replaced by:

  ```
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT )
  ```

- IRIS GL references the polygon stipple and the color writemask. OpenGL ignores the polygon stipple but references the color writemask. (The **czclear** function ignores both the polygon stipple and the color writemask.)

The following table lists the various IRIS GL clear functions with their equivalent OpenGL functions.

IRIS GL CALL	OPENGL CALL	MEANING
acbuf(AC_CLEAR)	**glClear**(GL_ACCUM_BUFFER_BIT)	Clear the accumulation buffer.
	glClearColor	Set the RGBA clear color.
	glClearIndex	Set the clear-color index.
clear	**glClear**(GL_COLOR_BUFFER_BIT)	Clear the color buffer.
	glClearDepth	Specify the clear value for the depth buffer.
zclear	**glClear**(GL_DEPTH_BUFFER_BIT)	Clear the depth buffer.
czclear	**glClear**(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT)	Clear the color buffer and the depth buffer.
	glClearAccum	Specify clear values for the accumulation buffer.
	glClearStencil	Specify the clear value for the stencil buffer.
sclear	**glClear**(GL_STENCIL_BUFFER_BIT)	Clear the stencil buffer.

When your IRIS GL code uses both **gclear** and **sclear**, you can combine them into a single **glClear** call; can improve your program's performance.

Porting Matrix and Transformation Functions

2/22/2020 • 2 minutes to read • Edit Online

IRIS GL and OpenGL handle matrices and transformations in a similar manner. But there are several differences to keep in mind when porting code from IRIS GL:

- In OpenGL you are always in double-matrix mode; there is no single-matrix mode.
- Angles are measured in degrees, instead of tenths of degrees.
- Projection matrix calls, like **glFrustum** and **glOrtho**, now multiply onto the current matrix, instead of being loaded onto the current matrix.
- The OpenGL function, **glRotate**, is very different from **rotate**. You can rotate around any arbitrary axis, instead of being confined to the x-, y-, and z- axes. For example, you can translate:

  ```
  rotate(200*(i+1), 'z');
  ```

  to:

  ```
  glRotate(.1*(200*(i+1), 0.0, 0.0, 1.0);
  ```

  When translating from **rotate** to **glRotate** you switch to degrees from tenths of degrees and replace 'z' with a vector for the z-axis.

- OpenGL has no equivalent to the **polarview** function. You can replace it easily with a translation and three rotations. For example, you can translate:

  ```
  polarview(distance, azimuth, incidence, twist);
  ```

  to:

  ```
  glTranslatef( 0.0, 0.0, -distance);
  glRotatef( -twist * 10.0, 0.0, 0.0, 1.0);
  glRotatef( -incidence * 10.0, 1.0, 0.0, 0.0);
  glRotatef( -azimuth * 10.0, 0.0, 0.0, 1.0);
  ```

The following table lists the OpenGL matrix functions and their equivalent IRIS GL functions.

IRIS GL FUNCTION	OPENGL FUNCTION	MEANING
mmode	glMatrixMode	Sets current matrix mode.
	glLoadIdentity	Replaces current matrix with the identity matrix.
loadmatrix	glLoadMatrixf,glLoadMatrixd	Replaces current matrix with the specified matrix.

IRIS GL FUNCTION	OPENGL FUNCTION	MEANING
multmatrix	glMultMatrixf,glMultMatrixd	Post-multiplies current matrix with the specified matrix (note that **multmatrix** pre-multiplied).
mapw, **mapw2**	gluUnProject	Projects world-space coordinates to object space (see also gluProject).
ortho	glOrtho	Multiplies current matrix by an orthographic projection matrix.
ortho2	gluOrtho2D	Defines a two-dimensional orthographic projection matrix.
perspective	gluPerspective	Defines a perspective projection matrix.
picksize	gluPickMatrix	Defines a picking region.
popmatrix	glPopMatrix	Pops current matrix stack, replacing the current matrix with the one below it.
pushmatrix	glPushMatrix	Pushes current matrix stack down by one, duplicating the current matrix.
rotate,**rot**	glRotated,glRotatef	Rotates current coordinate system by the given angle about the vector from the origin through the given point. Note that **rotate** rotated only about the x-, y-, and z-axes.
scale	glScaled,glScalef	Multiplies current matrix by a scaling matrix.
translate	glTranslatef,glTranslated	Moves coordinate-system origin to the point specified, by multiplying the current matrix by a translation matrix.
window	glFrustum	Given coordinates for clipping planes, multiplies the current matrix by a perspective matrix.

OpenGL has three matrix modes, which are set with glMatrixMode. The following table lists the modes available as parameters for **glMatrixMode**.

IRIS GL MATRIX MODE	OPENGL MODE	MEANING	MIN STACK DEPTH
MTEXTURE	GL_TEXTURE	Operates on the texture matrix stack.	2
MVIEWING	GL_MODELVIEW	Operates on the model view matrix stack.	32
MPROJECTION	GL_PROJECTION	Operates on the projection matrix stack.	2

Porting MSINGLE Mode Code

2/22/2020 • 2 minutes to read • Edit Online

OpenGL has no equivalent for **MSINGLE**, single-matrix mode. Though use of this mode has been discouraged, it is the default for IRIS GL. If your IRIS GL program uses the single-matrix mode, you need to rewrite it to use double-matrix mode only. OpenGL is always in double-matrix mode, and is initially in GL_MODELVIEW mode.

Most IRIS GL code in MSINGLE mode looks like this:

```
projectionmatrix();
```

where *projectionmatrix* is one of: **ortho**, **ortho2**, **perspective**, or **window**. To port to OpenGL, replace the **MSINGLE** -mode *projectionmatrix* function with:

```
glMatrixMode( GL_PROJECTION );
glLoadMatrix( identity matrix );

/* call one of these functions here: */
/* glFrustrum(), glOrtho(), glOrtho2(), gluPerspective()}; */

glMatrixMode( GL_MODELVIEW );
glLoadMatrix( identity matrix );
```

Porting Functions that Get Matrices and Transformations

2/22/2020 • 2 minutes to read • Edit Online

The following table lists the IRIS GL functions that get the state of matrices and transformations and their OpenGL equivalents.

IRIS GL MATRIX QUERY	OPENGL GLGET MATRIX QUERY	MEANING
getmmode	GL_MATRIX_MODE	Returns the current matrix mode.
getmatrix in **MVIEWING** mode	GL_MODELVIEW_MATRIX	Returns a copy of the current model-view matrix.
getmatrix in **MPROJECTION** mode	GL_PROJECTION_MATRIX	Returns a copy of the current projection matrix.
getmatrix in **MTEXTURE** mode	GL_TEXTURE_MATRIX	Returns a copy of the current texture matrix.
Not applicable.	GL_MAX_MODELVIEW_STACK_DEPTH	Returns maximum supported depth of the model-view matrix stack.
Not applicable.	GL_MAX_PROJECTION_STACK_DEPTH	Returns maximum supported depth of the projection matrix stack.
Not applicable.	GL_MAX_TEXTURE_STACK_DEPTH	Returns maximum supported depth of the texture matrix stack.
Not applicable.	GL_MODELVIEW_STACK_DEPTH	Returns number of matrices on the model view stack.
Not applicable.	GL_PROJECTION_STACK_DEPTH	Returns number of matrices on the projection stack.
Not applicable.	GL_TEXTURE_STACK_DEPTH	Returns number of matrices on the texture stack.

??

Porting Viewports, Screenmasks, and Scrboxes

2/22/2020 • 2 minutes to read • Edit Online

The following IRIS GL viewport functions have no OpenGL equivalent:

- **reshapeviewport**
- **scrbox**
- **getscrbox**

With the IRIS GL **viewport** function, you specify the x coordinates (in pixels) for the left and right of a viewport rectangle and the y coordinates for the top and bottom. With the OpenGL **glViewport** function, however, you specify the x and y coordinates (in pixels) of the lower-left corner of the viewport rectangle along with its width and height.

The following table lists IRIS GL viewport functions and their equivalent OpenGL functions.

IRIS GL FUNCTION	OPENGL FUNCTION	MEANING
viewport (left, right, bottom, top)	**glViewport** (x, y, width, height)	Sets the viewport.
popviewportpushviewport	**glPopAttribglPushAttrib** (GL_VIEWPORT_BIT)	Pushes and pops the stack.
getviewport	**glGet** (GL_VIEWPORT)	Returns viewport dimensions.

??

Porting Clipping Planes

2/22/2020 • 2 minutes to read • Edit Online

OpenGL implements clipping planes similarly to IRIS GL. In addition, in OpenGL you can query clipping planes. The following table lists IRIS GL clipping plane functions and their equivalent OpenGL functions.

IRIS GL FUNCTION	OPENGL FUNCTION	MEANING
clipplane (i, **CP_ON**, params)	**glEnable** (GL_CLIP_PLANEi)	Enables clipping on plane i.
clipplane(i, **CP_DEFINE**, plane)	**glClipPlane** (GL_CLIP_PLANEi, plane)	Defines clipping plane.
	glGetClipPlane	Returns clipping plane equation.
	glIsEnabled (GL_CLIP_PLANEi)	Returns true if clip plane i is enabled.
scrmask	**glScissor**	Defines the scissor box.
getscrmask	**glGet** (GL_SCISSOR_BOX)	Returns the current scissor box.

To turn on the scissor test, call **glEnable** using GL_SCISSOR_BOX as the parameter.

??

Porting Drawing Functions

2/22/2020 • 2 minutes to read • Edit Online

The following sections discuss how to port IRIS GL drawing primitives.

- The IRIS GL Sphere Library
- Porting v Functions
- Porting bgn/end Commands
- Porting Points
- Porting Lines
- Porting Polygons and Quadrilaterals
- Porting Triangles
- Porting Arcs and Circles
- Porting Spheres

The IRIS GL Sphere Library

2/22/2020 • 2 minutes to read • Edit Online

OpenGL does not support the IRIS GL sphere library. You can replace your sphere library calls with quadrics routines from the GLU library. For more information about the GLU library, see the *Open GL Programming Guide* and OpenGL Utility Library.

The following table lists the OpenGL quadrics functions.

OPENGL FUNCTION	MEANING
gluNewQuadric	Creates a new quadric object.
gluDeleteQuadric	Deletes a quadric object.
gluQuadricCallback	Associates a callback with a quadric object, for error handling.
gluQuadricNormals	Specifies normals: no normals, one per face, or one per vertex.
gluQuadricOrientation	Specifies direction of normals: outward or inward.
gluQuadricTexture	Turns texture-coordinate generation on or off.
gluQuadricDrawstyle	Specifies drawing style: polygons, lines, points, and so on.
gluSphere	Draws a sphere.
gluCylinder	Draws a cylinder or cone.
gluPartialDisk	Draws an arc.
gluDisk	Draws a circle or disk.

You can use one quadric object for all quadrics you want to render in similar ways. The following code sample uses two quadric objects to draw four quadrics, two of them textured.

```
GLUquadricObj    *texturedQuad, *plainQuad;

texturedQuad = gluNewQuadric(void);
gluQuadricTexture(texturedQuad, GL_TRUE);
gluQuadricOrientation(texturedQuad, GLU_OUTSIDE);
gluQuadricDrawStyle(texturedQuad, GLU_FILL);

plainQuad = gluNewQuadric(void);
gluQuadricDrawStyle(plainQuad, GLU_LINE);

glColor3f (1.0, 1.0, 1.0);

gluSphere(texturedQuad, 5.0, 20, 20);
glTranslatef(10.0, 10.0, 0.0);
gluCylinder(texturedQuad, 2.5, 5, 5, 10, 10);
glTranslatef(10.0, 10.0, 0.0);
gluDisk(plainQuad, 2.0, 5.0, 10, 10);
glTranslatef(10.0, 10.0, 0.0);
gluSphere(plainQuad, 5.0, 20, 20);
```

Porting v Functions

2/22/2020 • 2 minutes to read • Edit Online

In IRIS GL, you use variations on the **v** function to specify vertices. The equivalent OpenGL function is glVertex: Below are examples of **glVertex**.

```
glVertex2[d|f|i|s][v]( x, y );
glVertex3[d|f|i|s][v]( x, y, z);
glVertex4[d|f|i|s][v]( x, y, z, w);
```

The **glVertex** function takes suffixes the same way other OpenGL calls do. The vector versions of the call take arrays of the proper size as arguments. In the 2-D version, z=0 and w=1. In the 3-D version, w=1.

Porting bgn/end Commands

2/22/2020 • 2 minutes to read • Edit Online

IRIS GL uses the begin/end paradigm but has a different function for each graphics primitive. For example, you probably use **bgnpolygon** and **endpolygon** to draw polygons, and **bgnline** and **endline** to draw lines. In OpenGL, you use the **glBegin** / **glEnd** structure for both. In OpenGL you draw most geometric objects by enclosing a series of functions that specify vertices, normals, textures, and colors between pairs of **glBegin** and **glEnd** calls. For example:

```
void glBegin( GLenum mode) ;
   /* vertex list, colors, normals, textures, materials */
void glEnd( void );
```

The **glBegin** function takes a single parameter that specifies the drawing mode, and thus the primitive. Here's an OpenGL code sample that draws a polygon and then a line:

```
glBegin( GL_POLYGON ) ;
   glVertex2f(20.0, 10.0);
   glVertex2f(10.0, 30.0);
   glVertex2f(20.0, 50.0);
   glVertex2f(40.0, 50.0);
   glVertex2f(50.0, 30.0);
   glVertex2f(40.0, 10.0);
glEnd();
glBegin( GL_LINES ) ;
   glVertex2i(100,100);
   glVertex2i(500,500);
glEnd();
```

With OpenGL, you draw different geometric objects by specifying different parameters for **glBegin**. The following table lists the OpenGL **glBegin** parameters that correspond to equivalent IRIS GL functions.

IRIS GL FUNCTION	VALUE OF GLBEGIN MODE	MEANING
bgnpoint	GL_POINTS	Individual points.
bgnline	GL_LINE_STRIP	Series of connected line segments.
bgnclosedline	GL_LINE_LOOP	Series of connected line segments, with a segment added between first and last vertices.
	GL_LINES	Pairs of vertices interpreted as individual line segments.
bgnpolygon	GL_POLYGON	Boundary of a simple convex polygon.
	GL_TRIANGLES	Triples of vertices interpreted as triangles.
bgntmesh	GL_TRIANGLE_STRIP	Linked strips of triangles.

IRIS GL FUNCTION	VALUE OF GLBEGIN MODE	MEANING
	GL_TRIANGLE_FAN	Linked fans of triangles.
	GL_QUADS	Quadruples of vertices interpreted as quadrilaterals.
bgnqstrip	GL_QUAD_STRIP	Linked strips of quadrilaterals.

For a detailed discussion of the differences between triangle meshes, strips, and fans, see Porting Triangles.

There is no limit to the number of vertices you can specify between a **glBegin** / **glEnd** pair.

In addition to specifying vertices inside a **glBegin** / **glEnd** pair, you can specify a current normal, current texture coordinates, and a current color. The following table lists the commands valid inside a **glBegin** / **glEnd** pair.

IRIS GL FUNCTION	OPENGL FUNCTION	MEANING
v2, **v3**, **v4**	glVertex	Sets vertex coordinates.
RGBcolor, **cpack**	glColor	Sets current color.
color	glIndex	Sets current color index.
n3f	glNormal	Sets normal vector coordinates.
	glEvalCoord	Evaluates enabled one- and two-dimensional maps.
callobj	**glCallList**, **glCallLists**	Executes display list(s).
t2	glTexCoord	Sets texture coordinates.
	glEdgeFlag	Controls drawing edges.
lmbind	glMaterial	Sets material properties.

NOTE

If you use any OpenGL function other than those listed in the preceding table inside a **glBegin** / **glEnd** pair, you'll get unpredictable results, or possibly an error.

Porting Points

2/22/2020 • 2 minutes to read • Edit Online

OpenGL has no command to draw a single point. Otherwise, porting point functions is straightforward. The following table lists IRIS GL functions for drawing points and their equivalent OpenGL functions.

IRIS GL FUNCTION	OPENGL FUNCTION	MEANING
pnt		Draws a single point.
bgnpoint, **endpoint**	**glBegin** (GL_POINTS), **glEnd**	Interprets vertices as points.
pntsize	**glPointSize**	Sets point size in pixels.
pntsmooth	**glEnable** (GL_POINT_SMOOTH)	Turns on point antialiasing. (For more information on point antialiasing, see Porting Antialiasing Functions.)

For information about related get functions, see **glPointSize**.

??

Porting Lines

2/22/2020 • 2 minutes to read • Edit Online

Porting IRIS GL code that draws lines is fairly straightforward, though you should note the differences in the way OpenGL stipples. The following table lists IRIS GL functions for drawing lines and their equivalent OpenGL functions.

IRIS GL FUNCTION	OPENGL FUNCTION	MEANING
bgnclosedline,**endclosedline**	**glBegin** (GL_LINE_LOOP)**glEnd**	Draws a closed line.
bgnline	**glBegin** (GL_LINE_STRIP)	Draws line segments.
linewidth	**glLineWidth**	Sets line width.
getlwidth	**glGet** (GL_LINE_WIDTH)	Returns current line width.
deflinestyle,**setlinestyle**	**glLineStipple**	Specifies a line stipple pattern.
lsrepeat	factor argument of **glLineStipple**	Sets a repeat factor for the line style.
getlstyle	**glGet** (GL_LINE_STIPPLE_PATTERN)	Returns line stipple pattern.
getlsrepeat	**glGet** (GL_LINE_STIPPLE_REPEAT)	Returns repeat factor.
linesmooth, **smoothline**	**glEnable** (GL_LINE_SMOOTH)	Turns on line antialiasing (For more information on antialiasing, see Porting Antialiasing Functions.)

OpenGL doesn't use tables for line stipples; it maintains only one line-stipple pattern. You can use **glPushAttrib** and **glPopAttrib** to switch between different stipple patterns.

Older IRIS GL line style functions (such as **draw**, **lsbackup**, **getlsbackup**, and so on) are not supported by OpenGL.

For information on drawing antialiased lines, see Porting Antialiasing Functions.

??

Porting Polygons and Quadrilaterals

2/22/2020 • 4 minutes to read • Edit Online

Keep the following points in mind when porting polygons and quadrilaterals:

- There is no direct equivalent for **concave**(**TRUE**). Instead you can use the tessellation routines in the GLU, described in Tessellating Polygons.
- Polygon modes are set differently.
- These polygon drawing functions have no direct equivalents in OpenGL: the **poly** family of routines; the **polf** family of routines; **pmv**, **pdr**, and **pclos**; **rpmv** and **rpdr**; **splf**; and **spclos**.

If your IRIS GL code uses these functions, you'll have to rewrite the code using **glBegin**(GL_POLYGON).

The following table lists the IRIS GL polygon drawing functions and their equivalent OpenGL functions.

IRIS GL FUNCTION	OPENGL FUNCTION	MEANING
bgnpolygonendpolygon	**glBegin** (GL_POLYGON)**glEnd**	Vertices define boundary of a simple convex polygon.
	glBegin(GL_QUADS)**glEnd**	Interprets quadruples of vertices as quadrilaterals.
bgnqstripendqstrip	**glBegin** (GL_QUAD_STRIP)**glEnd**	Interprets vertices as linked strips of quadrilaterals.
	glEdgeFlag	
polymode	**glPolygonMode**	Sets polygon drawing mode.
rectrectf	glRect	Draws a rectangle.
sboxsboxf		Draws a screen-aligned rectangle.

??

Porting Polygon Modes

The OpenGL function **glPolygonMode** lets you specify which side of a polygon (the back or the front) the mode applies to. Its syntax is:

```
void glPolygonMode( GLenum face, GLenum mode );
```

where face is one of:

GL_FRONT	mode which applies to front-facing polygons
GL_BACK	mode which applies to back-facing polygons

GL_FRONT_AND_BACK	mode which applies to both front- and back-facing polygons

The GL_FRONT_AND_BACK mode is equivalent to the IRIS GL **polymode** function. The following table lists IRIS GL polygon modes and their equivalent OpenGL modes.

IRIS GL MODE	OPENGL MODE	MEANING
PYM_POINT	GL_POINT	Draws vertices as points.
PYM_LINE	GL_LINE	Draws boundary edges as line segments.
PYM_FILL	GL_FILL	Draws polygon interior filled.
PYM_HOLLOW		Fills only interior pixels at the boundaries.

Porting Polygon Stipples

When porting IRIS GL polygon stipples, keep the following points in mind:

- OpenGL doesn't use tables for polygon stipples; only one stipple pattern is kept. You can use display lists to store different stipple patterns.
- The OpenGL polygon stipple bitmap size is always a 32x32 bit pattern.
- Stipple encoding is affected by **glPixelStore**.

For more information on porting polygon stipples, see Porting Pixel Operations.

The following table lists IRIS GL polygon stipple functions and their equivalent OpenGL functions.

IRIS GL FUNCTION	OPENGL FUNCTION	MEANING
defpattern	**glPolygonStipple**	Sets the stipple pattern.
setpattern		OpenGL keeps only one polygon stipple pattern.
getpattern	**glGetPolygonStipple**	Returns the stipple bitmap (used to return an index).

In OpenGL, you enable and disable polygon stippling by passing GL_POLYGON_STIPPLE as a parameter for **glEnable** and **glDisable**.

The following OpenGL code sample demonstrates polygon stippling:

```
void display(void)
{
    GLubyte fly[] = {
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
      0x03, 0x80, 0x01, 0xC0, 0x06, 0xC0, 0x03, 0x60,
      0x04, 0x60, 0x06, 0x20, 0x04, 0x30, 0x0C, 0x20,
      0x04, 0x18, 0x18, 0x20, 0x04, 0x0C, 0x30, 0x20,
      0x04, 0x06, 0x60, 0x20, 0x44, 0x03, 0xC0, 0x22,
      0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
      0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
      0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
      0x66, 0x01, 0x80, 0x66, 0x33, 0x01, 0x80, 0xCC,
      0x19, 0x81, 0x81, 0x98, 0x0C, 0xC1, 0x83, 0x30,
      0x07, 0xe1, 0x87, 0xe0, 0x03, 0x3f, 0xfc, 0xc0,
      0x03, 0x31, 0x8c, 0xc0, 0x03, 0x33, 0xcc, 0xc0,
      0x06, 0x64, 0x26, 0x60, 0x0c, 0xcc, 0x33, 0x30,
      0x18, 0xcc, 0x33, 0x18, 0x10, 0xc4, 0x23, 0x08,
      0x10, 0x63, 0xC6, 0x08, 0x10, 0x30, 0x0c, 0x08,
      0x10, 0x18, 0x18, 0x08, 0x10, 0x00, 0x00, 0x08
    };
    GLubyte halftone[] = {
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55
    };

    glClear (GL_COLOR_BUFFER_BIT);

/*  draw all polygons in white*/
    glColor3f (1.0, 1.0, 1.0);

/*  draw one solid, unstippled rectangle,*/
/*  then two stippled rectangles*/
    glRectf (25.0, 25.0, 125.0, 125.0);
    glEnable (GL_POLYGON_STIPPLE);
    glPolygonStipple (fly);
    glRectf (125.0, 25.0, 225.0, 125.0);
    glPolygonStipple (halftone);
    glRectf (225.0, 25.0, 325.0, 125.0);
    glDisable (GL_POLYGON_STIPPLE);

    glFlush ();
}
```

Porting Tessellated Polygons

In IRIS GL, you use **concave**(**TRUE**) and then **bgnpolygon** to draw concave polygons. The OpenGL GLU includes functions you can use to draw concave polygons.

To draw a concave polygon with OpenGL

1. Create a tessellation object.

2. Define callbacks that will be used to process the triangles generated by the tessellator.
3. Specify the concave polygon to be tessellated.

The following table lists the OpenGL functions for drawing tessellated polygons.

OPENGL GLU FUNCTION	MEANING
gluNewTess	Creates a new tessellation object.
gluDeleteTess	Deletes a tessellation object.
gluTessCallback	
gluBeginPolygon	Begins the polygon specification.
gluTessVertex	Specifies a polygon vertex in a contour.
gluNextContour	Indicates that the next series of vertices describe a new contour.
gluEndPolygon	Ends the polygon specification.

Porting Triangles

2/22/2020 • 2 minutes to read • Edit Online

You can draw three types of triangles in OpenGL: separate triangles, triangle strips, and triangle fans.

OpenGL has no equivalent for the IRIS GL **swaptmesh** function. You can achieve the same effect using a combination of triangles, triangle strips, and triangle fans.

The following table lists the IRIS GL functions for drawing triangles and their equivalent OpenGL functions.

IRIS GL FUNCTION	EQUIVALENT GLBEGIN PARAMETER	MEANING
	GL_TRIANGLES	Triples of vertices interpreted as triangles.
bgntmesh, **endtmesh**	GL_TRIANGLE_STRIP	Linked strips of triangles.
	GL_TRIANGLE_FAN	Linked fans of triangles.

??

Porting Arcs and Circles

2/22/2020 • 2 minutes to read • Edit Online

With OpenGL, filled arcs and circles are drawn with the same calls as unfilled arcs and circles. The following table lists the IRIS GL arc and circle functions and their equivalent OpenGL (GLU) functions.

IRIS GL FUNCTION	OPENGL FUNCTION	MEANING
arcarcf	gluPartialDisk	Draws an arc.
circcircf	gluDisk	Draws a circle or disk.

You can do some things with OpenGL arcs and circles that you can't do with IRIS GL. OpenGL calls arcs and circles, disks and partial disks respectively.

When porting arcs and circles, keep the following points about OpenGL in mind:

- Angles are measured in degrees, and not in tenths of degrees.
- The start angle is measured from the positive y-axis, and not from the x-axis.
- The sweep angle is now clockwise instead of counterclockwise.

??

Porting Spheres

2/22/2020 • 2 minutes to read • Edit Online

When porting spheres to OpenGL, keep the following points in mind:

- You cannot control the type of primitives used to draw the sphere. You can control drawing precision in another way: use the slices and stacks parameters. Slices are longitudinal; stacks are latitudinal.
- Spheres are drawn centered at the origin. Instead of specifying the location, as you do with the IRIS GL **sphdraw** function, precede a call to the GLU **gluSphere** function with a translation.
- The sphere library is not yet available for OpenGL.

The following table lists the IRIS GL functions for drawing spheres and their equivalent GLU functions where available.

IRIS GL FUNCTION	GLU FUNCTION	MEANING
sphobj	**gluNewQuadric**	Creates a new sphere object.
sphfree	**gluDeleteQuadric**	Deletes sphere object and free memory used.
sphdraw	**gluSphere**	Draws a sphere.
sphmode		Sets sphere attributes.
sphrotmatrix		Controls sphere orientation.
sphgnpolys		Returns number of polygons in current sphere.

??

Porting Color, Shading, and Writemask Code

2/22/2020 • 2 minutes to read • Edit Online

When porting color, shading, and writemask code to OpenGL, keep the following points in mind:

- Though you can set color-map indexes with the OpenGL glIndex function, OpenGL does not have a function for loading color-map indexes.
- Color values are normalized to their data type. (For information about color values, see glColor).
- There is no simple equivalent for **cpack**.
- You may have to translate code that includes the **c** or **color** functions to **glClearColor** or **glClearIndex** instead of **glColor** or **glIndex**.
- The RGBA writemask applies to each component but not for each bit.
- IRIS GL provides defined color constants: BLACK, BLUE, RED, GREEN, MAGENTA, CYAN, YELLOW, and WHITE. OpenGL does not provide these constants.

This topic includes information on the following.

- Porting Color Calls
- Porting Shading Models

Porting Color Calls

2/22/2020 • 2 minutes to read • Edit Online

The following table lists IRIS GL color functions and their equivalent OpenGL functions.

IRIS GL FUNCTION	OPENGL FUNCTION	MEANING
c	glColor	Sets RGB color.
color	glIndex	Sets the color index.
getcolor	**glGet** (GL_CURRENT_INDEX)	Returns the current color index.
getmcolor		Gets a copy of the RGB values for a color map entry.
gRGBcolor	**glGet**(GL_CURRENT_COLOR)	Gets the current RGB color values.
mapcolor		
RGBcolor	**glColor**	Sets RGB color.
writemask	**glIndexMask**	Sets the color-index mode color mask.
wmpackRGBwritemask	**glColorMask**	Sets the RGB color mode mask.
getwritemask	**glGet** (GL_COLOR_WRITEMASK)**glGet**(GL_INDEX_WRITEMASK)	Gets the color mask.
gRGBmask	**glGet**(GL_COLOR_WRITEMASK)	Gets the color mask.
zwritemask	**glDepthMask**	

NOTE

Be careful when replacing **zwritemask** with **glDepthMask**; **glDepthMask** takes a Boolean argument, whereas **zwritemask** takes a bit field.

If you want to use multiple color maps, you need to use the appropriate Windows color map functions. Therefore, **multimap**, **onemap**, **getcmmode**, **setmap**, and **getmap** have no OpenGL equivalents.

Porting Shading Models

2/22/2020 • 2 minutes to read • Edit Online

Like IRIS GL, OpenGL lets you switch between smooth (Gouraud) shading and flat shading. The following table lists the IRIS GL shading and dithering functions and their equivalent OpenGL functions.

IRIS GL FUNCTION	OPENGL FUNCTION	MEANING
shademodel(FLAT)	**glShadeModel** (GL_FLAT)	Does flat shading.
shademodel(GOURAUD)	**glShadeModel**(GL_SMOOTH)	Does smooth shading.
getsm	**glGet** (GL_SHADE_MODEL)	Returns current shade model.
dither(DT_ON) **dither**(DT_OFF)	**glEnable** (GL_DITHER)**glDisable**(GL_DITHER)	Turns dithering on/off.

??

Porting Pixel Operations

2/22/2020 • 2 minutes to read • Edit Online

When porting code that involves pixel operations, keep the following points in mind:

- Logical pixel operations are not applied to RGBA color buffers. For more information, see **glLogicOp**.
- In general, IRIS GL uses the format ABGR for pixels, whereas OpenGL uses RGBA. You can change the format with **glPixelStore**.
- When porting **lrectwrite** functions, be careful to note where **lrectwrite** is writing (for example, it could be writing to the depth buffer).

OpenGL gives you some additional flexibility in pixel operations. The following table lists IRIS GL functions for pixel operations and their equivalent OpenGL functions.

IRIS GL FUNCTION	OPENGL FUNCTION	MEANING
lrectread, **rectread**,**readRGB**	**glReadPixels**	Reads a block of pixels from the framebuffer.
lrectwrite, **rectwrite**	**glDrawPixels**	Writes a block of pixels to the framebuffer.
rectcopy	**glCopyPixels**	Copies pixels in the framebuffer.
rectzoom	**glPixelZoom**	Specifies pixel zoom factors for **glDrawPixels** and **glCopyPixels**.
cmov	glRasterPos	Specifies raster position for pixel operations.
readsource	**glReadBuffer**	Selects a color buffer source for pixels.
pixmode	**glPixelStore**,**glPixelTransfer**	Sets pixel storage modes.Set pixel transfer modes.
logicop	**glLogicOp**	Specifies a logical operation for pixel writes.
	glEnable (GL_LOGIC_OP)	Turns on pixel logic operations.

For a complete list of possible logical operations, see **glLogicOp**.

This IRIS GL code sample shows a typical pixel write:

```
unsigned long *packedRaster;
..
packedRaster[k] = 0x00000000;
..
lrectwrite(0, 0, xSize, ySize, packedRaster);
```

The preceding code looks like this when translated to OpenGL:

```
glRasterPos2i( 0, 0);
glDrawPixels( xSize + 1, ySize + 1, GL_RGBA, GL_UNSIGNED_BYTE, packedRaster);
```

Porting Depth Cueing and Fog Commands

2/22/2020 • 2 minutes to read • Edit Online

When porting depth-cueing and fog commands, keep the following points in mind:

- The IRIS GL call, **fogvertex**, sets a mode and the parameters affecting that mode. In OpenGL, you call glFog once to set the mode, and then again twice or more to set various parameters.
- In OpenGL, depth cueing is not a separate feature. Use linear fog instead of depth cueing. (This section gives an example of how to do this.) The following IRIS GL functions have no direct OpenGL equivalent:

  depthcue

  lRGBrange

  lshaderange

  getdcm
- To adjust fog quality, use glHint(GL_FOG_HINT).

The following table lists the IRIS GL functions for managing fog and their equivalent OpenGL functions.

IRIS GL FUNCTION	OPENGL FUNCTION	MEANING
fogvertex	glFog	Sets various fog parameters.
fogvertex(FG_ON)	glEnable (GL_FOG)	Turns on fog.
fogvertex(FG_OFF)	glDisable (GL_FOG)	Turns off fog.
depthcue	glFog (GL_FOG_MOD, GL_LINEAR)	Uses linear fog for depth cueing.

The following table lists the parameters you can pass to **glFog**.

FOG PARAMETER	MEANING	DEFAULT
GL_FOG_DENSITY	Fog density.	1.0
GL_FOG_START	Near distance for linear fog.	0.0
GL_FOG_END	Far distance for linear fog.	1.0
GL_FOG_INDEX	Fog color index.	0.0
GL_FOG_COLOR	Fog RGBA color.	(0, 0, 0, 0)
GL_FOG_MODE	Fog mode.	See the following table.

The fog-density parameter of OpenGL differs from the one in IRIS GL. They are related as follows:

- if *fogMode* = EXP2

  openGLfogDensity = (*irisGLfogDensity*) (**sqrt** (**log** (1 / 255)))

- if *fogMode* = EXP

  openGLfogDensity = (*irisGLfogDensity*) (**log** (1 / 255))

where **sqrt** is the square root operation, **log** is the natural logarithm, *irisGLfogDensity* is the IRIS GL fog density, and *openGLfogDensity* is the OpenGL fog density.

To switch between calculating fog in per-pixel mode and per-vertex mode, use **glHint**(GL_FOG_HINT, *hintMode*). Two hint modes are available:

- GL_NICEST per-pixel fog calculation
- GL_FASTEST per-vertex fog calculation

The following table lists the IRIS GL fog modes and their OpenGL equivalents.

IRIS GL FOG MODE	OPENGL FOG MODE	HINT MODE	MEANING
FG_VTX_EXP,FG_PIX_EXP	GL_EXP	GL_FASTEST,GL_NICEST	Heavy fog mode (default).
FG_VTX_EXP2,FG_PIX_EXP2	GL_EXP2	GL_FASTEST,GL_NICEST	Haze mode.
FG_VTX_LIN,FG_PIX_LIN	GL_LINEAR	GL_FASTEST,GL_NICEST	Linear fog mode. (Use for depth cueing.)

The following code example demonstrates depth cueing in OpenGL:

```
/*
 *  depthcue.c
 *  This program draws a wire frame model, which uses
 *  intensity (brightness) to give clues to distance
 *  Fog is used to achieve this effect
 */
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"

/*  Initialize linear fog for depth cueing
 */
void myinit(void)
{
    GLfloat fogColor[4] = {0.0, 0.0, 0.0, 1.0};

    glEnable(GL_FOG);
    glFogi (GL_FOG_MODE, GL_LINEAR);
    glHint (GL_FOG_HINT, GL_NICEST);  /*  per pixel  */
    glFogf (GL_FOG_START, 3.0);
    glFogf (GL_FOG_END, 5.0);
    glFogfv (GL_FOG_COLOR, fogColor);
    glClearColor(0.0, 0.0, 0.0, 1.0);

    glDepthFunc(GL_LEQUAL);
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_FLAT);
}

/*  display() draws an icosahedron
 */
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0);
    auxWireIcosahedron(1.0);
    glFlush();
}

void myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective (45.0, (GLfloat) w/(GLfloat) h, 3.0, 5.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity ();
    glTranslatef (0.0, 0.0, -4.0); /*move object into view*/
}
/*  Main Loop
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
    auxInitPosition (0, 0, 400, 400);
    auxInitWindow (argv[0]);
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
}
```

Porting Curve and Surface Functions

2/22/2020 • 2 minutes to read • Edit Online

OpenGL doesn't support equivalents to the IRIS GL functions for curves and surface patches. You'll need to rewrite your code if it includes any of the following calls:

- **defbasis**
- **curvebasis**, **curveprecision**, **crv**, **crvn**, **rcrv**, **rcrvn**, and **curveit**
- **patchbasis**, **patchcurves**, **patchprecision**, **patch**, and **rpatch**

This topic includes information on the following.

- Porting NURBS Objects
- Porting NURBS Curves
- Porting Trimming Curves
- Porting NURBS Surfaces

Porting NURBS Objects

2/22/2020 • 2 minutes to read • Edit Online

OpenGL treats NURBS as objects, similar to the way it treats quadrics: you create a NURBS object and then specify how it should be rendered. The following table lists the OpenGL GLU functions for managing NURBS objects.

OPENGL GLU FUNCTION	MEANING
gluNewNurbsRenderer	Creates a new NURBS object.
gluDeleteNurbsRenderer	Deletes a NURBS object.
gluNurbsCallback	Associates a callback with a NURBS object, for error handling.

When porting IRIS GL NURBS code to OpenGL, keep the following points in mind:

- NURBS control points are floats, not doubles.
- The stride parameter is counted in floats, not bytes.
- If you're using lighting and you're not specifying normals, call **glEnable** with GL_AUTO_NORMAL as the parameter to generate normals automatically.

??

Porting NURBS Curves

2/22/2020 • 2 minutes to read • Edit Online

The OpenGL functions for drawing NURBS curves are very similar to the IRIS GL functions. You specify knot sequences and control points using a **gluNurbsCurve** function, which must be contained within a **gluBeginCurve** / **gluEndCurve** pair.

The following table lists the IRIS GL functions for drawing NURBS curves and their equivalent OpenGL functions.

IRIS GL FUNCTION	OPENGL FUNCTION	MEANING
bgncurve	**gluBeginCurve**	Begins a curve definition.
nurbscurve	**gluNurbsCurve**	Specifies curve attributes.
endcurve	**gluEndCurve**	Ends a curve definition.

Associate position, texture, and color coordinates by presenting each as a separate **gluNurbsCurve** inside the begin/end pair. You can make no more than one call to **gluNurbsCurve** for each piece of color, position, and texture data within a single **gluBeginCurve/gluEndCurve** pair. You must make exactly one call to describe the position of the curve (a GL_MAP1_VERTEX_3 or GL_MAP1_VERTEX_4 description). When you call **gluEndCurve**, the curve is tessellated into line segments and then rendered.

The following table lists IRIS GL and OpenGL NURBS curve types.

IRIS GL TYPE	OPENGL TYPE	MEANING
N_V3D	GL_MAP1_VERTEX_3	Polynomial curve.
N_V3DR	GL_MAP1_VERTEX_4	Rational curve.
	GL_MAP1_TEXTURE_COORD_*	Control points are texture coordinates.
	GL_MAP1_NORMAL	Control points are normals.

For more information on available evaluator types, see **glMap1**.

??

Porting Trimming Curves

2/22/2020 • 2 minutes to read • Edit Online

OpenGL trimming curves are very similar to IRIS GL trimming curves. The following table lists the IRIS GL functions for defining trimming curves and their equivalent OpenGL functions.

IRIS GL FUNCTION	OPENGL FUNCTION	MEANING
bgntrim	**gluBeginTrim**	Begins trimming-curve definition.
pwlcurve	**gluPwlCurve**	Defines a piecewise linear curve.
nurbscurve	**gluNurbsCurve**	Specifies trimming-curve attributes.
endtrim	**gluEndTrim**	Ends trimming-curve definition.

??

Porting NURBS Surfaces

2/22/2020 • 2 minutes to read • Edit Online

The following table lists the IRIS GL functions for drawing NURBS surfaces and their equivalent OpenGL functions.

IRIS GL FUNCTION	OPENGL FUNCTION	MEANING
bgnsurface	**gluBeginSurface**	Begins a surface definition.
nurbssurface	**gluNurbsSurface**	Specifies surface attributes.
endsurface	**gluEndSurface**	Ends a surface definition.

The following table lists IRIS GL parameters for surface types and their equivalent OpenGL parameters.

IRIS GL TYPE	OPENGL TYPE	MEANING
N_V3D	GL_MAP2_VERTEX_3	Polynomial curve.
N_V3DR	GL_MAP2_VERTEX_4	Rational curve.
N_C4D	GL_MAP2_COLOR_4	Control points define color surface in (R,G,B,A) form.
N_C4DR		
N_T2D	GL_MAP2_TEXTURE_COORD_2	Control points are texture coordinates.
N_T2DR	GL_MAP2_TEXTURE_COORD_3	Control points are texture coordinates.
	GL_MAP2_NORMAL	Control points are normals.

For more information on available evaluator types, see **glMap2**.

The following code sample draws a trimmed NURBS surface:

```
/*
 *  trim.c
 *  This program draws a NURBS surface in the shape of a
 *  symmetrical hill, using both a NURBS curve and pwl
 *  (piecewise linear) curve to trim part of the surface
 */
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"

GLfloat ctlpoints[4][4][3];

GLUnurbsObj *theNurb;
```

```
/*
 *  Initializes the control points of the surface to
 *  a small hill. The control points range from -3 to
 *  +3 in x, y, and z
 */
void init_surface(void)
{
    int u, v;
    for (u = 0; u < 4; u++) {
        for (v = 0; v < 4; v++) {
            ctlpoints[u][v][0] = 2.0*((GLfloat)u - 1.5);
            ctlpoints[u][v][1] = 2.0*((GLfloat)v - 1.5);

            if ( (u == 1 || u == 2) && (v == 1 || v == 2))
                ctlpoints[u][v][2] = 3.0;
            else
                ctlpoints[u][v][2] = -3.0;
        }
    }
}

/*  Initialize material property and depth buffer
 */
void myinit(void)
{
    GLfloat mat_diffuse[] = { 0.6, 0.6, 0.6, 1.0 };
    GLfloat mat_specular[] = { 0.9, 0.9, 0.9, 1.0 };
    GLfloat mat_shininess[] = { 128.0 };

    glClearColor (0.0, 0.0, 0.0, 1.0);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glDepthFunc(GL_LEQUAL);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_AUTO_NORMAL);
    glEnable(GL_NORMALIZE);

    init_surface();

    theNurb = gluNewNurbsRenderer();
    gluNurbsProperty(theNurb, GLU_SAMPLING_TOLERANCE, 50.0);
    gluNurbsProperty(theNurb, GLU_DISPLAY_MODE, GLU_FILL);
}

void display(void)
{
    GLfloat knots[8] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
    GLfloat edgePt[5][2] = /* counter clockwise */
    {{0.0, 0.0}, {1.0, 0.0}, {1.0, 1.0}, {0.0, 1.0},
     {0.0, 0.0}};
    GLfloat curvePt[4][2] = /* clockwise */
    {{0.25, 0.5}, {0.25, 0.75}, {0.75, 0.75}, {0.75, 0.5}};
    GLfloat curveKnots[8] =
        {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
    GLfloat pwlPt[4][2] = /* clockwise */
        {{0.75, 0.5}, {0.5, 0.25}, {0.25, 0.5}};

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    glRotatef(330.0, 1.,0.,0.);
    glScalef (0.5, 0.5, 0.5);

    gluBeginSurface(theNurb);
    gluNurbsSurface(theNurb,
            8, knots,
```

```
            8, knots,
            4 * 3,
            3,
            &ctlpoints[0][0][0],
            4, 4,
            GL_MAP2_VERTEX_3);
    gluBeginTrim (theNurb);
        gluPwlCurve (theNurb, 5, &edgePt[0][0], 2,
                     GLU_MAP1_TRIM_2);
    gluEndTrim (theNurb);
    gluBeginTrim (theNurb);
        gluNurbsCurve (theNurb, 8, curveKnots, 2,
                &curvePt[0][0], 4, GLU_MAP1_TRIM_2);
        gluPwlCurve (theNurb, 3, &pwlPt[0][0], 2,
                     GLU_MAP1_TRIM_2);
    gluEndTrim (theNurb);
    gluEndSurface(theNurb);

    glPopMatrix();
    glFlush();
}

void myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective (45.0, (GLdouble)w/(GLdouble)h, 3.0, 8.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef (0.0, 0.0, -5.0);
}

/*  Main Loop
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow (argv[0]);
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
}
```

Porting Antialiasing Functions

2/22/2020 • 2 minutes to read • Edit Online

In OpenGL the subpixel mode is always on, consequently the IRIS GL function **subpixel**(**TRUE**) is not necessary and has no OpenGL equivalent. The following topics describe aspects of porting IRIS GL antialiasing code.

- Porting Blending Code
- Porting afunction Test Functions
- Using Antialiasing Functions

Porting Blending Code

2/22/2020 • 2 minutes to read • Edit Online

In IRIS GL, when drawing to both front and back buffers, blending is done by reading one of the buffers, blending with that color, and then writing the result to both buffers. In OpenGL, however, each buffer is read in turn, blended, and then written.

The following table lists IRIS GL blending functions and their equivalent OpenGL functions.

IRIS GL FUNCTION	OPENGL FUNCTION	MEANING
	glEnable (GL_BLEND)	Turns on blending.
blendfunction	glBlendFunc	Specifies a blend function.

The OpenGL **glBlendFunc** function and the IRIS GL **blendfunction** function are almost identical. The following table lists IRIS GL blend factors and their OpenGL equivalents.

IRIS GL	OPENGL	NOTES
BF_ZERO	GL_ZERO	
BF_ONE	GL_ONE	
BF_SA	GL_SRC_ALPHA	
BF_MSA	GL_ONE_MINUS_SRC_ALPHA	
BF_DA	GL_DST_ALPHA	
BF_MDA	GL_ONE_MINUS_DST_ALPHA	
BF_SC	GL_SRC_COLOR	
BF_MSC	GL_ONE_MINUS_SRC_COLOR	Destination only.
BF_DC	GL_DST_COLOR	Source only.
BF_MDC	GL_ONE_MINUS_DST_COLOR	Source only.
BF_MIN_SA_MDA	GL_SRC_ALPHA_SATURATE	

??

Porting afunction Test Functions

2/22/2020 • 2 minutes to read • Edit Online

The following table lists the available IRIS GL alpha test functions and their equivalent OpenGL functions.

AFUNCTION	GLALPHAFUNC
AF_NOTEQUAL	GL_NOTEQUAL
AF_ALWAYS	GL_ALWAYS
AF_NEVER	GL_NEVER
AF_LESS	GL_LESS
AF_EQUAL	GL_EQUAL
AF_LEQUAL	GL_LEQUAL
AF_GREATER	GL_GREATER
AF_GEQUAL	GL_GEQUAL

Using Antialiasing Functions

2/22/2020 • 2 minutes to read • Edit Online

The following table lists IRIS GL antialiasing functions and their equivalent OpenGL functions.

IRIS GL FUNCTION	OPENGL FUNCTION	MEANING
pntsmooth	**glEnable** (GL_POINT_SMOOTH)	Enables antialiasing of points.
linesmooth	**glEnable**(GL_LINE_SMOOTH)	Enables antialiasing of lines.
polysmooth	**glEnable** (GL_POLYGON_SMOOTH)	Enables antialiasing of polygons.

Use the equivalent **glDisable** calls to turn off antialiasing.

In IRIS GL, you can control the quality of the antialiasing by calling:

```
linesmooth(SML_ON + SML_SMOOTHER);
```

OpenGL provides similar controluse **glHint**:

```
glHint(GL_POINT_SMOOTH_HINT, hintMode);
glHint(GL_LINE_SMOOTH_HINT, hintMode);
glHint(GL_POLYGON_SMOOTH_HINT, hintMode);
```

where *hintMode* is one of the following:

- GL_NICEST (Use the highest quality smoothing.)
- GL_FASTEST (Use the most efficient smoothing.)
- GL_DONT_CARE

IRIS GL also permits end-correction by calling:

```
linesmooth(SML_ON +  SML_END_CORRECT);
```

OpenGL has no equivalent for this function.

Porting Accumulation Buffer Calls

2/22/2020 • 2 minutes to read • Edit Online

You must allocate your accumulation buffer by requesting the appropriate pixel format with the OpenGL **auxInitDisplayMode** or **ChoosePixelFormat** function. The following table lists IRIS GL functions that affect the accumulation buffer and their equivalent OpenGL functions.

IRIS GL FUNCTION	OPENGL FUNCTION	MEANING
acsize	**auxInitDisplayMode** or **ChoosePixelFormat**	Specifies number of bitplanes per color component in the accumulation buffer.
acbuf	**glAccum**	Operates on the accumulation buffer.
	glClearAccum	Sets clear values for accumulation buffer.
acbuf(AC_CLEAR)	**glClear** (GL_ACCUM_BUFFER_BIT)	Clears the accumulation buffer.

The following table lists the IRIS GL **acbuf** parameters along with the equivalent OpenGL **glAccum** parameters.

IRIS GL PARAMETER	OPENGL PARAMETER
AC_ACCUMULATE	GL_ACCUM
AC_CLEAR_ACCUMULATE	GL_LOAD
AC_RETURN	GL_RETURN
AC_MULT	GL_MULT
AC_ADD	GL_ADD

Porting Stencil Plane Calls

2/22/2020 • 2 minutes to read • Edit Online

In OpenGL, you allocate stencil planes by requesting the appropriate pixel format with the OpenGL **auxInitDisplayMode** or **ChoosePixelFormat**. functions. The following table lists IRIS GL functions that affect the stencil planes and their equivalent OpenGL functions.

IRIS GL FUNCTION	OPENGL FUNCTION	MEANING
stensize	**ChoosePixelFormat**	
stencil(**TRUE**, ...)	**glEnable** (GL_STENCIL_TEST)	Enables stencil tests.
stencil	**glStencilOp**	Sets stencil test actions.
stencil(... func, ...)	**glStencilFunc**	Sets function and reference value for stencil testing.
swritemask	**glStencilMask**	Specifies which stencil bits can be written.
	glClearStencil	Specifies the clear value for the stencil buffer.
sclear	**glClear** (GL_STENCIL_BUFFER_BIT)	

Stencil-comparison functions and stencil pass/fail operations are nearly equivalent in OpenGL and IRIS GL. The IRIS GL stencil-function flags are prefaced with SF; the OpenGL flags with GL. IRIS GL pass/fail operation flags are prefaced with ST; the OpenGL flags with GL.

Porting Display Lists

2/22/2020 • 2 minutes to read • Edit Online

The OpenGL implementation of display lists is similar to the IRIS GL implementation, with two exceptions: in OpenGL you can't edit display lists once you've created them and you can't call functions from within display lists.

Because you can't edit or call functions from within display lists, these IRIS GL functions have no equivalent in OpenGL:

- **editobj**
- **objdelete**, **objinsert**, and **objreplace**
- **maketag**, **gentag**, **istag**, and **deltag**
- **callfunc**

In IRIS GL, you use the **makeobj** and **closeobj** functions to create display lists. In OpenGL, you use glNewList and glEndList.

The following table lists the IRIS GL display list functions with their equivalent OpenGL functions.

IRIS GL FUNCTION	OPENGL FUNCTION	MEANING
makeobj	**glNewList**	Creates a new display list.
closeobj	**glEndList**	Signals end of display list.
callobj	glCallList, glCallLists	Executes display list or lists.
isobj	glIsList	Tests for display list existence.
delobj	glDeleteLists	Deletes contiguous group of display lists.
genobj	glGenLists	Generates the given number of contiguous empty display lists.

This topic includes information on the following.

- Porting the bbox2 Function
- Porting Edited Display Lists
- A Sample Port of a Display List

Porting the bbox2 Function

2/22/2020 • 2 minutes to read • Edit Online

There is no OpenGL equivalent for the IRIS GL **bbox2** function.

To port code that contains bbox2 functions

1. Create a new (OpenGL) display list that contains everything in the equivalent IRIS GL display list except the call to **bbox2**.
2. Use appropriate Windows code to draw a rectangle the same size as the IRIS GL **bbox**.

Porting Edited Display Lists

2/22/2020 • 2 minutes to read • Edit Online

Although you can't edit OpenGL display lists, you can get similar results by nesting display lists and then destroying and creating new versions of the sublists. For example:

```
glNewList (1, GL_COMPILE);
    glIndexi(MY_RED);
glEndlist();

glNewList(2, GL_COMPILE);
    glScalef(1.2, 1.2, 1.0);
glEndList();

glNewList(3, GL_COMPILE);
    glCallList(1);
    glCallList(2);
glEndList();

glDeleteLists(1, 2);
glNewList(1, GL_COMPILE);
    glIndexi(MY_CYAN);
glEndList();
glNewList(2, GL_COPILE);
    glScalef(0.5, 0.5, 1.0);
glEndList;
```

A Sample Port of a Display List

2/22/2020 • 2 minutes to read • Edit Online

This topic gives an IRIS GL sample of code that defines three display lists; one of the display lists refers to the others in its definition. Following the IRIS GL sample is a sample of what the code looks like when ported to OpenGL.

IRIS GL Sample Display List Code

```
makeobj(10);  // 10 object
    cpack(0x0000FF);
    recti(164, 33, 364, 600);   // Hollow rectangle
closeobj();

makeobj(20);  // 20 object
    cpack(0xFFFF00);
    circle(0, 0, 25);           // Unfilled circle
    recti(100, 100, 200, 200);  // Filled rectangle
closeobj();

makeobj(30);  // 30 object
    callobj(10);
    cpack(0xFFFFFF);
    recti(400, 100, 500, 300);  // Draw filled rectangle
    callobj(20);
closeobj();

// Now draw by calling the lists
call(30);
```

OpenGL Sample Display List Code

Here is the preceding IRIS GL code translated to OpenGL:

```
glNewList(10, GL_COMPILE);            // List #10
    glColor3f(1, 0, 0);
    glRecti(164, 33, 364, 600);
glEndList();

glNewList(20, GL_COMPILE);            //List #20
    glColor3f(1, 1, 0);               // Set color to YELLOW
    glPolygonMode(GL_BOTH, GL_LINE);  // Unfilled mode
    glBegin(GL_POLYGON);  // Use polygon to approximate a circle
        for(i=0; i<100; i++) {
            cosine = 25 * cos(i * 2 * PI/100.0);
            sine =   25 * sin(i * 2 * PI/100.0);
            glVertex2f(cosine, sine);
        }
    glEnd();
    glBegin(GL_QUADS);
        glColorf(0, 1, 1);            // Set color to CYAN
        glVertex2i(100, 100);
        glVertex2i(100, 200);
        glVertex2i(200, 200);
        glVertex2i(100, 200);
    glEnd();
glEndList();

glNewList(30, GL_COMPILE);            // List #30
    glCallList(10);
        glColorf(1, 1, 1);            // Set color to WHITE
        glRecti(400, 100, 500, 300);
    glCallList(20);
glEndList();

// Execute the display lists
glCallList(30);
```

Porting Defs, Binds, and Sets

2/22/2020 • 2 minutes to read • Edit Online

OpenGL doesn't have tables of stored definitions; you can't define lighting models, material, textures, line styles, or patterns as separate objects as you can in IRIS GL. Thus OpenGL has no direct equivalents to the following IRIS GL functions:

- **lmdef** and **lmbind**
- **tevdef** and **tevbind**
- **textdef** and **textbind**
- **definestyle** and **setstyle**
- **defpattern** and **setpattern**

You can use OpenGL display lists to mimic the IRIS GL def/bind mechanism. For example, here is a material definition in IRIS GL:

```
float mat() = {
    AMBIENT, .1, .1, .1,
    DIFFUSE, 0, .369, .165,
    SPECULAR, .5, .5, .5,
    SHININESS, 10,
    LMNULL
};
lmdef(DEFMATERIAL, 1, 0, mat);
lmbind(MATERIAL, 1);
```

The following OpenGL code sample defines the same material in a display list that is referred to by the list number defined by **MYMATERIAL**.

```
#define MYMATERIAL 10

GLfloat        mat_amb[] = {.1, .1, .1, 1.0};
GLfloat        mat_dif[] = {0, .369, .165, 1.0};
GLfloat        mat_spec[] = {.5, .5, .5, 1.0};

glNewList(MYMATERIAL, GL_COMPILE);
    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_amb);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_dif);
    glMaterialfv(GL_FRONT, GL_SHININESS, 10);
glEndList();

glCallList( MYMATERIAL );
```

Porting Lighting and Materials Functions

2/22/2020 • 2 minutes to read • Edit Online

OpenGL functions for lighting and materials differ substantially from the IRIS GL functions. Unlike IRIS GL, OpenGL has separate functions for setting lights, light models and materials.

Keep the following points in mind when porting lighting and materials functions:

- OpenGL has no table of stored definitions. You can use display lists to mimic the IRIS GL def/bind mechanism. For more information on defs and binds, see Porting Defs, Binds, and Sets.
- With OpenGL, attenuation is associated with each light source, rather than the overall lighting model.
- Diffuse and specular components are separated in OpenGL light sources.
- OpenGL light sources have an alpha component. When porting your IRIS GL code, set this alpha component to 1.0, indicating 100 percent opaque. The alpha values are then determined by the alpha component of your materials only, so the objects in your scene will look the same as they did in IRIS GL.

The following table lists IRIS GL lighting and materials functions and their equivalent OpenGL functions.

IRIS GL FUNCTION	OPENGL FUNCTION	MEANING
Imdef(DEFLIGHT, ...)	glLight	Define a light source.
Imdef(DEFLMODEL, ...)	glLightModel	Define a lighting model.
Imbind	**glEnable** (GL_LIGHT *i*)	Enable light *i*.
Imbind	**glEnable**(GL_LIGHTING)	Enable lighting.
Imdef(DEFMATERIAL, ...)	**glMaterial**	Define a material.
Imcolor	**glColorMaterial**	Change the effect of color commands while lighting is active.
	glGetMaterial	Get material parameters.

The following table lists various IRIS GL material parameters and their equivalent OpenGL parameters.

IMDEF INDEX	GLMATERIAL PARAMETER	DEFAULT	MEANING
ALPHA	GL_DIFFUSE		The fourth value in the GL_DIFFUSE parameter specifies the alpha value.
AMBIENT	GL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	Ambient color.
DIFFUSE	GL_DIFFUSE	(0.8, 0.8, 0.8, 1.0)	Diffuse color.
SPECULAR	GL_SPECULAR	(0.0, 0.0, 0.0, 1.0)	Emissive color.

IMDEF INDEX	GLMATERIAL PARAMETER	DEFAULT	MEANING
SHININESS	GL_SHININESSGL_AMBIENT_AND_DIFFUSE	0.0	Specular exponent.Equivalent to calling **glMaterial** twice with the same values.
COLORINDEXES	GL_COLOR_INDEXES		Color indexes for ambient, diffuse, and specular lighting.

When the first parameter of **Imdef** is DEFMODEL, the equivalent OpenGL translation is the function **glLightModel**. The exception is when the parameter following DEFMODEL is ATTENUATION: then the equivalent OpenGL function is **glLight**.

The following table lists the equivalent lighting model parameters for IRIS GL and OpenGL.

IMDEF INDEX	GLLIGHTMODEL PARAMETER	DEFAULT	MEANING
AMBIENT	GL_LIGHT_MODEL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	Ambient color of scene.
ATTENUATION			See **glLight**.
LOCALVIEWER	GL_LIGHT_MODEL_LOCAL_VIEWER	GL_FALSE	Viewer local (**TRUE**) or infinite (**FALSE**).
TWOSIDE	GL_LIGHTMODEL_TWO_SIDE	GL_FALSE	Use two-sided lighting when **TRUE**.

When the first parameter of **Imdef** is DEFLIGHT, the equivalent OpenGL translation is the **glLight** function.

The following table lists the equivalent lighting parameters for IRIS GL and OpenGL.

IMDEF INDEX	GLLIGHT PARAMETER	DEFAULT	MEANING
AMBIENT	GL_AMBIENTGL_DIFFUSE GL_SPECULAR	(0.0, 0.0, 0.0, 1.0)(1.0, 1.0, 1.0, 1.0) (1.0, 1.0, 1.0, 1.0)	Ambient intensity.Diffuse intensity. Specular intensity.
LCOLOR	No equivalent.		
POSITION	GL_POSITION	(0.0, 0.0, 1.0, 0.0)	Position of light.
SPOTDIRECTION	GL_SPOT_DIRECTION	(0, 0, 1)	Direction of spotlight.
SPOTLIGHT	GL_SPOT_EXPONENTGL_SPOT_CUTOFF	0180	Intensity distribution.Maximum spread angle of light source.
DEFMODEL, ATTENUATION	GL_CONSTANT_ATTENUATIONGL_LINEAR_ATTENUATION GL_QUADRATIC_ATTENUATION	(1, 0, 0)	Attenuation factors.

Porting Texture Functions

2/22/2020 • 2 minutes to read • Edit Online

When porting IRIS GL texture functions to OpenGL, keep the following points in mind:

- OpenGL doesn't maintain tables of textures; it uses either 1-D texture and 2-D texture only. To reuse the textures from your IRIS GL code, put them in a display list.
- OpenGL doesn't automatically generate mipmaps. If you're using mipmaps, you must first call the **gluBuild2DMipmaps** function.
- In OpenGL, you use **glEnable** and **glDisable** to turn texturing capabilities on and off.
- In OpenGL, texture size is more strictly regulated than in IRIS GL. The size of an OpenGL texture must be:

  $2n + 2b$

  where n is an integer and b is:

  - 0, if the texture has no border
  - 1, if the texture has a border pixel (OpenGL textures can have 1-pixel borders.)

The following table lists IRIS GL texture functions and their general OpenGL equivalents.

IRIS GL FUNCTION	OPENGL FUNCTION	MEANING
textdef2d	glTexImage2DglTexParameter gluBuild2DMipmaps	Specifies a 2-D texture image.
textbind	glTexImage2DglTexParameter gluBuild2DMipmaps	Selects a texture function.
tevdef	glTexEnv	Defines a texture-mapping environment.
tevbind	glTexEnvglTexImage1D	Selects a texture environment.
t2	glTexCoord	Sets the current texture coordinates.
texgen	glTexGenglGetTexParameter gluBuild1DMipmaps gluBuild2DMipmaps gluScaleImage	Controls generation of texture coordinates.Scales an image to an arbitrary size.

For more information on texturing, see the *OpenGL Programming Guide*.

This topic includes information on the following.

- Translating tevdef
- Translating texdef
- Translating texgen

Translating tevdef

2/22/2020 • 2 minutes to read • Edit Online

The following code example is an IRIS GL texture-environment definition that specifies the TV_DECAL texture-environment parameter:

```
float tevprops[] = {TV_DECAL, TV_NULL};

tevdef(1, 0, tevprops);
```

and the same code translated to OpenGL:

```
glTexEnvfv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
```

The following table lists the IRIS GL texture-environment parameters and their equivalent OpenGL parameters.

IRIS GL PARAMETER	OPENGL PARAMETER
TV_MODULATE	GL_MODULATE
TV_DECAL	GL_DECAL
TV_BLEND	GL_BLEND
TV_COLOR	GL_TEXTURE_ENV_COLOR
TV_ALPHA	No direct OpenGL equivalent.
TV_COMPONENT_SELECT	No direct OpenGL equivalent.

For more information about texture-environment parameters, see glTexEnv.

Translating texdef

2/22/2020 • 2 minutes to read • Edit Online

The following code example is an IRIS GL texture definition:

```
float texprops[] = { TX_MINFILTER, TX_POINT,
                        TX_MAGFILTER, TX_POINT,
                        TX_WRAP_S, TX_REPEAT,
                        TX_WRAP_T, TX_REPEAT,
                        TX_NULL };

textdef2d(1, 1, 6, 6, granite_texture, 7, texprops);
```

In the preceding example, **texdef** specifies the TX_POINT filter as both the magnification and the minimizing filter, and TX_REPEAT as the wrapping mechanism. It also specifies the texture image: **granite_texture**.

In OpenGL, **glTexImage** specifies the image and glTexParameter sets the property. To translate IRIS GL texture definitions, replace the **textdef** function with **glTexImage** and one or more calls to **glTexParameter**.

The preceding IRIS GL code looks like this when translated to OpenGL:

```
GLfloat nearest[] = {GL_NEAREST};
GLfloat repeat = {GL_REPEAT};
glTexParameterfv( GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, nearest);
glTexParameterfv( GL_TEXTURE_1D, GL_TEXTURE_MAGILTER, nearest);
glTexParameterfv( GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, repeat);
glTexParameterfv( GL_TEXTURE_1D, GL_TEXTURE_WRAP_T, nearest);
glTexImage1D( GL_TEXTURE_1D, 0, 1, 6, 0, GL_RGB, GL_UNSIGNED_SHORT, granite_texture);
```

The following table lists the IRIS GL texture parameters and their equivalent OpenGL parameters.

IRIS GL TEXTURE PARAMETER	OPENGL TEXTURE PARAMETER
TX_MINFILTER	GL_TEXTURE_MIN_FILTER
TX_MAGFILTER	GL_TEXTURE_MAG_FILTER
TX_WRAP, TX_WRAP_S	GL_TEXTURE_WRAP_S
TX_WRAP, TX_WRAP_T	GL_TEXTURE_WRAP_TGL_TEXTURE_BORDER_COLOR

The following table lists the possible values of the IRIS GL texture parameters and their equivalent OpenGL parameters.

IRIS GL TEXTURE PARAMETER	OPENGL TEXTURE PARAMETER
TX_POINT	GL_NEAREST
TX_BILINEAR	GL_LINEAR
TX_MIPMAP_POINT	GL_NEAREST_MIPMAP_NEAREST

IRIS GL TEXTURE PARAMETER	OPENGL TEXTURE PARAMETER
TX_MIPMAP_BILINEAR	GL_LINEAR_MIPMAP_NEAREST
TX_MIPMAP_LINEAR	GL_NEAREST_MIPMAP_LINEAR
TX_TRILINEAR	GL_LINEAR_MIPMAP_LINEAR

Translating texgen

2/22/2020 • 2 minutes to read • Edit Online

The IRIS GL function **texgen** is translated to glTexGen for OpenGL.

With IRIS GL, you call **texgen** twice: once to set the mode and plane equation simultaneously, and once to enable texture-coordinate generation. For example:

```
texgen(TX_S, TG_LINEAR, planeParams);
texgen(TX_S, TG_ON, NULL);
```

With OpenGL, you make three calls: two to **glTexGen** (once to set the mode and once to set the plane equation), and one to **glEnable**. For example, the OpenGL equivalent to the IRIS GL code above is:

```
glTexGen(GL_S, GLTEXTURE_GEN_MODE, modeName);
glTextGen(GL_S, GL_OBJECT_PLANE, planeParameters);
glEnable(GL_TEXTURE_GEN_S);
```

The following table lists the IRIS GL texture-coordinate names and their OpenGL equivalents.

IRIS GL TEXTURE COORDINATE	OPENGL TEXTURE COORDINATE	GLENABLE ARGUMENT
TX_S	GL_S	GL_TEXTURE_GEN_S
TX_T	GL_T	GL_TEXTURE_GEN_T
TX_R	GL_R	GL_TEXTURE_GEN_R
TX_Q	GL_Q	GL_TEXTURE_GEN_Q

The following table lists the IRIS GL texture-generation modes and their equivalent OpenGL texture modes and plane names.

IRIS GL TEXTURE MODE	OPENGL TEXTURE MODE	OPENGL PLANE NAME
TG_LINEAR	GL_OBJECT_LINEAR	GL_OBJECT_PLANE
TG_CONTOUR	GL_EYE_LINEAR	GL_EYE_PLANE
TG_SPHEREMAP	GL_SPHERE_MAP	

Porting Picking Functions

2/22/2020 • 2 minutes to read • Edit Online

All IRIS GL picking functions have OpenGL equivalents, with the exception of **clearhitcode**. The following table lists the IRIS GL picking functions and their equivalent OpenGL functions.

IRIS GL FUNCTION	OPENGL FUNCTION	MEANING
clearhitcode	Not supported.	Clears global variable and hitcode.
pickselect	glRenderMode (GL_SELECT)	Switches to selection or picking mode.
endpickendselect	glRenderMode (GL_RENDER)	Switches to rendering mode.
picksize	gluPickMatrixglSelectBuffer	Sets the return array.
initnames	glInitNames	
pushnamepopname	glPushNameglPopName	
loadname	glLoadName	

For more information on picking, see gluPickMatrix.

Porting Feedback Functions

2/22/2020 • 2 minutes to read • Edit Online

With IRIS GL, the way that feedback is handled differs depending on the computer running IRIS GL. OpenGL standardizes feedback functions so you can rely on consistent feedback among various hardware platforms. The following table lists the IRIS GL feedback functions and their equivalent OpenGL functions.

IRIS GL FUNCTION	OPENGL FUNCTION	MEANING
feedback	**glRenderMode** (GL_FEEDBACK)	Switches to feedback mode.
endfeedback	**glRenderMode** (GL_RENDER)**glFeedbackBuffer**	Switches to rendering mode.
passthrough	**glPassThrough**	Places a token marker in the feedback buffer.

OpenGL Functions and Their IRIS GL Equivalents

2/22/2020 • 6 minutes to read • Edit Online

This appendix lists IRIS GL functions and their equivalent OpenGL functions. The first column is an alphabetical list of IRIS GL functions, the second column contains the corresponding functions to use in OpenGL.

> **NOTE**
> The following OpenGL functions listed may behave somewhat differently from the IRIS GL commands, and the parameters may be different as well. For more information on the differences between IRIS GL and OpenGL, see IRIS GL and OpenGL Differences.

IRIS GL FUNCTION	OPENGL, GLU, OR WINDOWS FUNCTION
acbuf	glAccum
acsize	ChoosePixelFormat
addtopup	Use Windows for menus.
afunction	glAlphaFunc
arc	gluPartialDisk
backbuffer	glDrawBuffer (GL_BACK)
backface	glCullFace (GL_BACK)
bbox2	Not supported.
bgnclosedline	glBegin (GL_LINE_LOOP)
bgncurve	gluBeginCurve
bgnline	glBegin (GL_LINE_STRIP)
bgnpoint	glBegin (GL_POINTS)
bgnpolygon	glBegin (GL_POLYGON)
bgnqstrip	glBegin (GL_QUAD_STRIP)
bgnsurface	gluBeginSurface
bgntmesh	glBegin (GL_TRIANGLE_STRIP)
bgntrim	gluBeginTrim

IRIS GL FUNCTION	OPENGL, GLU, OR WINDOWS FUNCTION
blankscreen	Use Windows for windowing.
blanktime	Use Windows for windowing.
blendfunction	glBlendFunc
blink	Use Windows for color maps.
blkqread	Use Windows for event handling.
c	glColor
callfunc	Not supported.
callobj	glCallList
charstr	glCallLists
chunksize	Not needed.
circ	gluDisk
clear	glClear (GL_COLOR_BUFFER_BIT)
clearhitcode	Not supported.
clipplane	glClipPlane
clkon	Use Windows for keyboard management.
clkoff	Use Windows for keyboard management.
closeobj	glEndList
cmode	ChoosePixelFormat
cmov	glRasterPos3
cmov2	glRasterPos2
color	glIndex
compactify	Not needed.
concave	gluBeginPolygon
cpack	glColor
crv	Not supported.

IRIS GL FUNCTION	OPENGL, GLU, OR WINDOWS FUNCTION
crvn	Not supported.
curorigin	Use Windows for cursors.
cursoff	Use Windows for cursors.
curson	Use Windows for cursors.
curstype	Use Windows for cursors.
curvebasis	glMap1
curveit	glEvalMesh1
curveprecision	Not supported.
cyclemap	Use Windows for color maps.
czclear	glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
dbtext	Not supported.
defbasis	glMap1
defcursor	Use Windows for cursors.
deflinestyle	glLineStipple
defpattern	glPolygonStipple
defpup	Use Windows for menus.
defrasterfont	wglUseFontBitmaps
delobj	glDeleteLists
deltag	Not supported.
depthcue	glFog
dglclose	Not needed. (OpenGL is network transparent.)
dglopen	Not needed. (OpenGL is network transparent.)
dither	glEnable (GL_DITHER)
dopup	Use Windows for menus.
doublebuffer	ChoosePixelFormat

IRIS GL FUNCTION	OPENGL, GLU, OR WINDOWS FUNCTION
draw	glBegin (GL_LINES)
drawmode	wglMakeCurrent
editobj	Not supported.
endclosedline	glEnd
endcurve	gluEndCurve
endfeedback	glRenderMode (GL_RENDER)
endfullscreen	Not supported.
endline	glEnd
endpick	glRenderMode (GL_RENDER)
endpoint	glEnd
endpolygon	glEnd
endpupmode	Use Windows for menus.
endqstrip	glEnd
endselect	glRenderMode (GL_RENDER)
endsurface	gluEndSurface
endtmesh	glEnd
endtrim	gluEndTrim
feedback	glFeedbackBuffer
finish	glFinish
fogvertex	glFog
font	glListBase
foreground	Use Windows for windowing.
freepup	Use Windows for menus.
frontbuffer	glDrawBuffer (GL_FRONT)
frontface	glCullFace

IRIS GL FUNCTION	OPENGL, GLU, OR WINDOWS FUNCTION
fudge	Use Windows for windowing.
fullscrn	Not supported.
gammaramp	Use Windows for color maps.
gbegin	Use Windows for windowing.
gconfig	No equivalent. (Not needed.)
genobj	glGenLists
gentag	Not supported.
getbackface	glGet
getbuffer	glGet
getbutton	Use Windows for windowing.
getcmmode	wglGetCurrentContext
getcolor	glGet
getcpos	glGet
getcursor	Not supported.
getdcm	glIsEnabled
getdepth	glGet
getdescender	Use Windows for fonts.
getdev	Not supported.
getdisplaymode	glGet
	wglGetCurrentContext
getdrawmode	wglGetCurrentContext
getfont	Use Windows for fonts.
getgdesc	glGet, DescribePixelFormat, wglGetCurrentContext , wglGetCurrentDC
getgpos	Not supported.
getheight	Use Windows for fonts.

IRIS GL FUNCTION	OPENGL, GLU, OR WINDOWS FUNCTION
gethitcode	Not supported.
getlsbackup	Not supported.
getlsrepeat	glGet
getlstyle	glGet
getlwidth	glGet
getmap(void)	Not supported.
getmatrix	glGet (GL_MODELVIEW_MATRIX), glGet (GL_PROJECTION_MATRIX)
getmcolor	Not supported.
getmmode	glGet (GL_MATRIX_MODE)
getmonitor	Not supported.
getnurbsproperty	gluGetNurbsProperty
getopenobj	Not supported.
getorigin	Use Windows for windowing.
getpattern	glGetPolygonStipple
getplanes	glGet (GL_RED_BITS), glGet (GL_GREEN_BITS), glGet (GL_BLUE_BITS)
getport	Use Windows for windowing.
getresetls	Not supported.
getscrbox	Not supported.
getscrmask	glGet (GL_SCISSOR_BOX)
getshade	glGet (GL_CURRENT_INDEX)
getsize	Use Windows for windowing.
getsm	glGet (GL_SHADE_MODEL)
getvaluator	Use Windows for event handling
getvideo	Not supported.

IRIS GL FUNCTION	OPENGL, GLU, OR WINDOWS FUNCTION
getviewport	glGet (GL_VIEWPORT)
getwritemask	glGet (GL_INDEX_WRITEMASK)
getwscrn	Use Windows for windowing.
getzbuffer	glIsEnabled (GL_DEPTH_TEST)
gexit	Use Windows for windowing.
gflush	glFlush
ginit	Use Windows for windowing.
glcompat	Not supported.
greset	Not supported.
gRGBcolor	glGet (GL_CURRENT_RASTER_COLOR)
gRGBcursor	Use Windows for cursors.
gRGBmask	glGet (GL_COLOR_WRITEMASK)
gselect	glSelectBuffer
gsync	Use Windows for windowing.
gversion	glGetString (GL_RENDERER)
iconsize	Use Windows.
icontitle	Use Windows.
imakebackground	Use Windows for event handling.
initnames	glInitNames
ismex	Not supported.
isobj	glIsList
isqueued	Use Windows for event handling.
istag	Not supported.
keepaspect	Use Windows for windowing.
lampoff	Not supported.

IRIS GL FUNCTION	OPENGL, GLU, OR WINDOWS FUNCTION
lampon	Not supported.
linesmooth	glEnable (GL_LINE_SMOOTH)
linewidth	glLineWidth
linewidthf	glLineWidth
lmbind	glEnable (GL_LIGHTING), **glEnable** (GL_LIGHT)
lmcolor	glColorMaterial
lmdef	glMaterial, glLight, glLightModel
loadmatrix	glLoadMatrix
loadname	glLoadName
logicop	glLogicOp
lookat	gluLookAt
lrectread	glReadPixels
lrectwrite(])	glDrawPixels
lRGBrange	Not supported. (See glFog.)
lsbackup	Not supported.
lsetdepth	glDepthRange
lshaderange	Not supported. (See glFog.)
lsrepeat	glLineStipple
makeobj	glNewList
maketag	Not supported.
mapcolor	Use Windows for color maps.
mapw	gluProject
maxsize	Use Windows for windowing.
minsize	Use Windows for windowing.
mmode	glMatrixMode

IRIS GL FUNCTION	OPENGL, GLU, OR WINDOWS FUNCTION
move	Not supported.
mswapbuffers	Use Windows for windowing.
multimap	Use Windows for color maps.
multmatrix	glMultMatrix
n3f	glNormal3fv
newpup	Use Windows for Menus.
newtag	Not supported.
nmode	glEnable (GL_NORMALIZE)
noborder	Use Windows for windowing.
noise	Use Windows for event handling.
noport	Use Windows for windowing.
normal	glNormal3fv
nurbscurve	gluNurbsCurve
nurbssurface	gluNurbsSurface
objdelete	Not supported.
objinsert	Not supported.
objreplace	Not supported.
onemap	Use Windows for color maps.
ortho	glOrtho
ortho2	gluOrtho2D
overlay	Use Windows.
pagecolor	Not supported.
passthrough	glPassThrough
patch	glEvalMesh2
patchbasis	glMap2

IRIS GL FUNCTION	OPENGL, GLU, OR WINDOWS FUNCTION
patchcurves	glMap2
patchprecision	Not supported.
pclos	Not supported. (See glEnd.)
pdr	Not supported. (See glVertex.)
perspective	gluPerspective
pick	gluPickMatrix, glRenderMode (GL_SELECT)
picksize	gluPickMatrix
pixmode	glPixelTransfer and 3
pmv	Not supported. (See glBegin and glVertex.)
pnt	glBegin (GL_POINTS)
pntsize	glPointSize
pntsizef	glPointSize
pntsmooth	glEnable (GL_POINT_SMOOTH)
polarview	Not supported. (See glRotate and glTranslate.)
polf	Not supported.
poly	Not supported.
polymode	glPolygonMode
polysmooth	glEnable (GL_POLYGON_SMOOTH)
popattributes	glPopAttrib
popmatrix	glPopMatrix
popname	glPopName
popviewport	glPopAttrib
prefposition	Use Windows for windowing.
prefsize	Use Windows for windowing.
pupmode	Use Windows for windowing.

IRIS GL FUNCTION	OPENGL, GLU, OR WINDOWS FUNCTION
pushattributes	glPushAttrib
pushmatrix	glPushMatrix
pushname	glPushName
pushviewport	glPushAttrib (GL_VIEWPORT)
pwlcurve	gluPWLCurve
qcontrol	Use Windows for event handling.
qdevice	Use Windows for event handling.
qenter	Use Windows for event handling.
qgetfd	Use Windows for event handling.
qread	Use Windows for event handling.
qreset	Use Windows for event handling.
qtest	Use Windows for event handling.
rcrv	Not supported.
rcrvn	Not supported.
rdr	Not supported.
readdisplay	Not supported.
readRGB	Not supported.
readsource	glReadBuffer
rect	glRect, glPolygonMode
rectf	glRect
rectcopy	glCopyPixels
rectread	glReadPixels
rectwrite	glDrawPixels
rectzoom	glPixelZoom
resetls	Not supported.

IRIS GL FUNCTION	OPENGL, GLU, OR WINDOWS FUNCTION
reshapeviewport	Not supported.
RGBcolor	glColor
RGBcursor	Use Windows for cursors.
RGBmode	Use Windows for windowing.
RGBrange	Not supported.
RGBwritemask	glColorMask
ringbell	Not supported.
rmv	Not supported.
rot	glRotate
rotate	glRotate
rpatch	Not supported.
rpdr	Not supported.
rpmv	Not supported.
sbox	glRect
scale	glScale
sclear	glClear (GL_STENCIL_BUFFER_BIT)
scrbox	Not supported.
screenspace	Not supported.
scrmask	glScissor
scrnattach	Use Windows for windowing.
scrnselect	Use Windows for windowing.
scrsubdivide	Not supported.
select	glRenderMode
setbell	Not supported.
setcursor	Use Windows for cursors.

IRIS GL FUNCTION	OPENGL, GLU, OR WINDOWS FUNCTION
setdblights	Not supported.
setdepth	glDepthRange
setlinestyle	glLineStipple
setmap	Use Windows for color maps.
setmonitor	Not supported.
setnurbsproperty	gluNurbsProperty
setpattern	glPolygonStipple
setpup	Use Windows for menus.
setvaluator	Use Windows for devices.
setvideo	Not supported.
shademodel	glShadeModel
shaderange	glFog
singlebuffer	Use Windows for windowing.
smoothline	glEnable (GL_LINE_SMOOTH)
spclos	Not supported.
splf	Not supported. (See glBegin.)
stencil	glStencilFunc, glStencilOp
stensize	glStencilMask
stepunit	Use Windows for windowing.
strwidth	Use Windows for fonts and strings.
subpixel	Not needed.
swapbuffers	SwapBuffers
swapinterval	Use Windows for windowing.
swaptmesh	Not supported. (See glBegin(GL_TRIANGLE_FAN)
swinopen	Use Windows for windowing.

IRIS GL FUNCTION	OPENGL, GLU, OR WINDOWS FUNCTION
swritemask	glStencilMask
t2	glTexCoord2
tevbind	glTexEnv
tevdef	glTexEnv
texbind	glTexImage2D, glTexParameter, gluBuild2DMipmaps
texdef2d	glTexImage2D, glTexParameter, gluBuild2DMipmaps
texgen	glTexGen
textcolor	Not supported.
textinit	Not supported.
textport	Not supported.
tie	Use Windows for event handling.
tpoff	Not supported.
tpon	Not supported.
translate	glTranslate
underlay	ChoosePixelFormat
unqdevice	Use Windows for event handling.
v	glVertex
videocmd	Not supported.
viewport	glViewport
winattach	Use Windows for windowing.
winclose	wglDeleteContext, CloseWindow
winconstraints	Use Windows for windowing.
windepth	Use Windows for windowing.
window	glFrustum
winget	wglGetCurrentContext

IRIS GL FUNCTION	OPENGL, GLU, OR WINDOWS FUNCTION
winmove	Use Windows for windowing.
winopen	Use Windows for windowing.
winpop	Use Windows for windowing.
winposition	Use Windows for windowing.
winpush	Use Windows for windowing.
winset	Use Windows for windowing.
wintitle	Use Windows for windowing.
wmpack	glColorMask
writemask	glIndexMask
writepixels	glDrawPixels
writeRGB	glDrawPixels
xfpt	Not supported.
zbuffer	glEnable (GL_DEPTH_TEST)
zclear	glClear (GL_DEPTH_BUFFER_BIT)
zdraw	Not supported.
zfunction	glDepthFunc
zsource	Not supported.
zwritemask	glDepthMask

IRIS GL and OpenGL Differences

2/22/2020 • 11 minutes to read • Edit Online

This appendix lists the differences between OpenGL and IRIS GL. A term for each difference is given, followed by a description.

accumulation wrapping	The OpenGL accumulation buffer operation is not defined when component values exceed 1.0 or drop below 1.0.
antialiased lines	OpenGL stipples antialiased lines. IRIS GL does not.
arc	OpenGL supports arcs in its utility library.
attribute lists	The attributes pushed by the IRIS GL pushattributes differ from any of the attribute sets pushed by the OpenGL glPushAttrib . Since all OpenGL states can be read back, however, you can implement any desired push/pop semantics using OpenGL.
automatic texture scaling	The OpenGL texture interface does not support automatic scaling of images to power-of-two dimensions. However, the GLU supports image scaling.
bbox	OpenGL doesn't support conditional execution of display lists.
callfunc	OpenGL doesn't support callback from display lists. Note that IRIS GL doesn't support this functionality either, when client and server are on different platforms.
circle	OpenGL supports circles with the GLU. In OpenGL both circles and arcs (disks and partial disks) can have holes. In addition, you can change subdivision of the primitives in OpenGL, whose surface normals are available for lighting.
clear options	OpenGL actually clears buffers. It doesn't apply currently specified pixel operations, such as blending and logicop, regardless of their modes. To clear using such features, you must render a window-size polygon.
closed lines	OpenGL renders all single-width aliased lines such that abutting lines share no pixels. This means that the last pixel of an independent line is not drawn.
color/normal flag	OpenGL lighting is explicitly enabled or disabled. When enabled, it is effective regardless of the order in which colors and normals are specified. You cannot enable or disable lighting between OpenGL glBegin and glEnd commands. To disable lighting between glBegin and glEnd, specify zero ambient, diffuse, and specular material reflectance, and then set the material emission to the desired color.

concave polygons	The core OpenGL API doesn't handle concave polygons, but the GLU supports decomposing concave, non-self-intersecting contours into triangles. These triangles can either be drawn immediately or returned.
current computed color	OpenGL has no equivalent to a current computed color. If you're using OpenGL as a lighting engine, you can use feedback to obtain colors generated by lighting calculations.
current graphics position	OpenGL doesn't maintain a current graphics position. IRIS GL commands that depend on current graphics position, such as relative lines and polygons, are not included in OpenGL.
curves	OpenGL does not support IRIS GL curves. Use of NURBS curves is recommended.
defs/binds	OpenGL doesn't have the concept of material, light, or texture objects; only of material, light, and texture properties. You can use display lists to create their own objects, however.
depthcue	OpenGL provides no direct support for depth cueing, but its fog support is a more general capability that you can easily use to emulate the IRIS GL depthcue function.
display list editing	OpenGL display lists can't be editedonly created and destroyed. Because you can specify display list names, however, you can redefine individual display lists in a hierarchy. OpenGL display lists are designed for data caching, not for database management. They are guaranteed to be stored on the server in client/server environments, so they are not limited by network bandwidth during execution. OpenGL display lists can be called between glBegin and glEnd commands, so the display list hierarchy can be made fine enough that it can, in effect, be edited.
error checking	OpenGL checks for errors more carefully than IRIS GL. For example, all OpenGL functions that are not accepted between glBegin and glEnd are detected as errors, and have no other effect.
error return values	When an OpenGL command that returns a value detects an error, it always returns zero. OpenGL commands that return data through passed pointers make no change to the array contents if an error is detected.
error side effects	When an OpenGL command results in an error, its only side effect is to update the error flag to the appropriate value. No other state changes are made. (An exception is the OUT_OF_MEMORY error, which is fatal.)
feedback	Feedback is standardized in OpenGL so it doesn't change from machine to machine.

fonts and strings	OpenGL requires character glyphs to be manipulated as individual display lists. It provides a display list calling function that accepts a list of display list names, each name represented as 1, 2, or 4 bytes. The glCallLists function adds a separately specified offset to each display list name before the call, allowing lists of display list names to be treated as strings. This mechanism provides all the functionality of IRIS GL fonts, and considerably more. For example, characters comprised of triangles can be easily manipulated.
frontbuffer	IRIS GL has complex rules for rendering to the front buffer in single buffer mode. OpenGL handles rendering to the front buffer in a straightforward way.
hollow polygons	You can use the OpenGL stencil capacity to render hollow polygons. OpenGL doesn't support other means for creating hollow polygons.
index clamping	Where possible, OpenGL treats color and stencil indexes as bit fields rather than numbers. Thus indexes are masked, rather than clamped, to the supported range of the framebuffer.
integer colors	Signed integer color components (red, green, blue, or alpha) are mapped in linear fashion to floating points so that the most negative integer maps to 1.0 and the most positive integer maps to 1.0. This mapping occurs when you specify the color before OpenGL replaces the current color. Unsigned integer color components are mapped linearly to floating points so that 0 maps to 0.0 and the largest integer maps to 1.0. This mapping occurs when you specify the color before OpenGL replaces the current color.
integer normals	Integer normal components are mapped just like signed color components. The most negative integer maps to 1.0, and the most positive integer maps to 1.0. pixel fragments. Pixels drawn by glDrawPixels or glCopyPixels are always rasterized and converted to fragments. The resulting fragments are textured, fogged, depth buffered, blended, and so on, just as if they were generated from geometric points. Fragment data that isn't provided by the source pixels is augmented from the current raster position. For example, RGBA pixels take the raster position Z and texture coordinates. Depth pixels take the raster position color and texture coordinates.
invariance	OpenGL guarantees a certain consistency that IRIS GL doesn't. For example, OpenGL guarantees that identical code sequences sent to the same system, differing only in the specified blending function, will generate the same pixel fragments. (The fragments differ, however, if blending is enabled and then disabled.)

lighting equation	The OpenGL lighting equation differs slightly from the IRIS GL equation. OpenGL supports separate attenuation for each light source, rather than a single attenuation for all the light sources like IRIS GL. OpenGL adjusts the equation so that ambient, diffuse, and specular lighting contributions are all attenuated. Also, OpenGL allows you to specify separate colors for the ambient, diffuse, and specular intensities of light sources, as well as for the ambient, diffuse, and specular reflectance of materials. All OpenGL light and material colors include alpha. Setting the specular exponent to zero does not defeat specular lighting in OpenGL.
mapw	OpenGL utilities support mapping between object and window coordinates.
matrix mode	Where the IRIS GL ortho, ortho2, perspective, and window functions operate on a particular matrix, all OpenGL matrix operations work on the current matrix. All OpenGL matrix operations except glLoadIdentity and glLoadMatrix multiply the current matrix rather than replacing it (as do ortho, ortho2, perspective, and window in the IRIS GL).
mipmaps, automatic generation	The OpenGL texture interface does not support automatic generation of mipmap images. However, the GLU supports the automatic generation of mipmap images for both 1-D and 2-D textures.
move/draw/pmove/pdraw/pclos	OpenGL supports only Begin/End style graphics, because it does not maintain a current graphics position. The scalar parameter specification of the old move/draw commands is accepted by OpenGL for all vertex related commands, however.
mprojection mode	IRIS GL doesn't transform geometry by the modelview matrix while in projection matrix mode. OpenGL always transforms by both the modelview and the projection matrix, regardless of matrix mode.
multi-buffer drawing	OpenGL renders to each color buffer individually, rather than computing a single, new, color value based on the contents of one color buffer and writing it to all the enabled color buffers, as IRIS GL does.
NURBS	OpenGL supports NURBS with a combination of core capability (evaluators) and GLU support. All IRIS GL NURBS capabilities are supported.
old polygon mode	Aliased OpenGL polygons are always point-sampled. IRIS GL's polygon compatibility mode, where pixels outside the polygon perimeter are included in its rasterization, is not supported. If your code uses this polygon mode, it is probably for rectangles. Old polygon mode rectangles appear one pixel wider and higher.

packed color formats	OpenGL accepts colors as 8-bit components, but these components are treated as an array of bytes rather than as bytes packed into larger words. By encouraging array indexing rather than shifting, OpenGL promotes endian-invariant programming. Just as IRIS GL accepts packed colors both for geometric and pixel rendering, OpenGL accepts arrays of color components for geometric and pixel rendering.
patches	OpenGL doesn't support IRIS GL patches.
per-bit color writemask	OpenGL writemasks for color components enable or disable changes to the entire component (red, green, blue, or alpha), not to individual bits of components. Note that per-bit writemasks are supported for both color indexes and stencil indexes, however.
per-bit depth writemask	OpenGL writemasks for depth components enable or disable changes to the entire component, not to individual bits of the depth component.
pick	The OpenGL Utility library includes support for generating a pick matrix.
pixel coordinates	In both OpenGL and IRIS GL, the origin of a window's coordinate system is at its lower left corner. OpenGL places the origin at the lower left corner of this pixel, however, while IRIS GL places it at the center of the lower left pixel.
pixel zoom	OpenGL negative zoom factors reflect about the current graphics position. IRIS GL doesn't define the operation of negative zoom factors, and instead provides RIGHT_TO_LEFT and TOP_TO_BOTTOM reflection pixmodes. These reflection modes reflect in place, rather than about the current raster position. OpenGL doesn't define reflection modes.
pixmode	OpenGL pixel transfers operate on individual color components, rather than on packed groups of four 8-bit components as does IRIS GL. While OpenGL provides substantially more pixel capability than IRIS GL, it doesn't support packed color constructs, and it doesn't enable color components to be reassigned (red to green, red to blue, and so on) during pixel copy operations.
polf/poly	OpenGL provides no direct support for vertex lists other than display lists. Functions like polf and poly can be implemented easily using the OpenGL API, however.
polygon provoking vertex	Flat shaded IRIS GL polygons take the color of the last vertex specified, while OpenGL polygons take the color of the first vertex specified.
polygon stipple	With IRIS GL the polygon stipple pattern is relative to the screen. With OpenGL it is relative to a window.
polygon vertex count	There is no limit to the number of vertices between glBegin and glEnd with OpenGL, even for glBegin(POLYGON). With IRIS GL, polygons are limited to no more than 255 vertices.

readdisplay	Reading pixels outside window boundaries is properly a window system capability, rather than a rendering capability. Use Windows functions to replace the IRIS GL readdisplay command.
relative move/draw/pmove/pdraw/pclos	OpenGL doesn't maintain a current graphics position, and therefore doesn't support relative vertex operations.
RGBA logicop()	OpenGL does not support logical operations on RGBA buffers.
sbox()	sbox is an IRIS GL rectangle primitive that is well-defined only if transformed without rotation. It is designed to be rendered faster than standard rectangles. While OpenGL doesn't support such a primitive, it can be tuned to render rectangles very quickly when the matrices and other modes are in states that simplify calculations.
scalar arguments	All OpenGL commands that are accepted between glBegin and glEnd have entry points that accept scalar arguments. For example, glColor4f (red, green, blue, alpha).
scissor	The OpenGL glScissor function doesn't track the viewport. The IRIS GL viewport command automatically updates the scrmask.
scrbox()	OpenGL doesn't support bounding box computation.
scrsubdivide()	OpenGL doesn't support screen subdivision.
single matrix mode	OpenGL always maintains two matrices: ModelView and Projection. While an OpenGL implementation can consolidate these into a single matrix for performance reasons, it must always present the two-matrix model to the programmer.
subpixel mode	All OpenGL rendering is subpixel positionedsubpixel mode is always on.
swaptmesh()	OpenGL doesn't support the swaptmesh capability. It does offer two types of triangle meshes, however: one that corresponds to the default "strip" behavior of the IRIS GL, and another that corresponds to calling swaptmesh prior to the third and all subsequent vertices when using IRIS GL.
vector arguments	All OpenGL commands that are accepted between glBegin and glEnd have entry points that accept vector arguments. For example, glColor4fv .
window management	OpenGL includes no window system commands. It is always supported as an extension to a window or operating system that includes capability for device and window control. Each extension provides a system-specific mechanism for creating, destroying, and manipulating OpenGL rendering contexts. For example, the OpenGL extension to the X window system (GLX) includes roughly 10 commands for this purpose. IRIS GL commands such as gconfig and drawmode are not implemented by OpenGL.

window offset	IRIS GL returns viewport and character positions in screen, rather than window, coordinates. OpenGL always uses window coordinates.
z rendering	OpenGL doesn't support rendering colors to the depth buffer. It does allow for additional color buffers, which can be implemented using the same memory that is used for depth buffers in other window configurations. But these additional color buffers cannot share memory with the depth buffer in any single configuration.

OpenGL Glossary

2/22/2020 • 2 minutes to read • Edit Online

The following topics provide a glossary of OpenGL terms.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

A

2/22/2020 • 2 minutes to read • Edit Online

a l i a s i n g

A rendering technique that assigns to pixels the color of the primitive being rendered, whether that primitive covers all or part of the pixel's area. Aliasing results in jagged edges, or jaggies. See also jaggies.

a l p h a

A fourth color component typically used to control color blending. The alpha component is never displayed directly. By convention, OpenGL alpha indicates opacity and transparency on a scale of 0.0 to 1.0. An alpha value of 1.0 implies complete opacity, an alpha value of 0.0 implies complete transparency.

a n i m a t i o n

The generation of repeated renderings of a scene quickly enough, with smoothly changing viewpoint or object positions, that the illusion of motion is achieved. OpenGL animation almost always uses double-buffering.

a n t i a l i a s

i
n
g

A rendering technique that assigns colors to pixels based on the fraction of the pixel area that is covered by the primitive being rendered. Antialiased rendering reduces or eliminates the jaggies that result from aliased rendering. See also jaggies, rendering.

a
p
p
l
i
c
a
t
i
o
n
-
s
p
e
c
i
f
i
c
c
l
i
p
p
i
n
g

Clipping of primitives against planes in eye coordinates. The planes are specified by the application using **glClipPlane**. See also eye coordinates.

B

2/22/2020 • 2 minutes to read • Edit Online

backface

See face.

bit

Binary digit. A state variable that has only two possible values: 0 or 1. Binary numbers are constructions of one or more bits.

bitmap

A rectangular array of bits. Also, the primitive rendered by the **glBitmap** command, which uses its bitmap parameter as a mask.

bitplane

A rectangular array of bits mapped one-to-one with pixels.

blending

Reducing two color components to one component, usually as a linear interpolation between the two components.

b
u
f
f
e
r

A group of bitplanes that stores a single component (such as depth or green) or a single index (such as the color index or the stencil index). Sometimes the red, green, blue, and alpha buffers are referred to collectively as the color buffer (rather than as the color buffers).

C

2/22/2020 • 4 minutes to read • Edit Online

client computer

The computer from which OpenGL commands are issued. The computer that issues OpenGL commands can be connected through a network to a different computer that executes the commands, or commands can be issued and executed on the same computer. See also server.

client memory

The main memory (where program variables are stored) of the client computer.

clipcoordinate

t
e
s

The coordinate system that follows transformation by the projection matrix and precedes perspective division. View-volume clipping is done in clip coordinates, but application-specific clipping is not. See also application-specific clipping.

c
l
i
p
p
i
n
g

Eliminating the portion of a geometric primitive that is outside the half-space defined by a clipping plane. Points are simply rejected if outside. The portion of a line or of a polygon that is outside the half-space is eliminated, and additional vertices are generated as necessary to complete the primitive within the clipping half-space. Geometric primitives and the current raster position (when specified) are always clipped against the six half-spaces defined by the left, right, bottom, top, near, and far planes of the view volume. Applications can specify optional application-specific clipping planes to be applied in eye coordinates.

c
o
l
o
r
i
n
d
e
x

A single value that represents a color by name, rather than by value. OpenGL color indexes are treated as continuous values (for example, floating-point numbers) while operations such as interpolation and dithering are performed on them. Color indexes stored in the frame buffer are always integer values, however. Floating-point indexes are converted to integers by rounding to the nearest integer value.

c
o
l
o
r
-
i
n
d
e
x
m
o

d
e

Mode of an OpenGL context in which its color buffers store color indexes, rather than red, green, blue, and alpha color components.

c
o
l
o
r
m
a
p

A table of index-to-RGB mappings that is accessed by the display hardware. Each color index is read from the color buffer, converted to an RGB triple by lookup in the color map, and sent to the monitor.

c
o
m
p
o
n
e
n
t

A single, continuous (for example, floating-point) value that represents an intensity or quantity. Usually, a component value of zero represents the minimum value or intensity, and a component value of one represents the maximum value or intensity, though other normalizations are sometimes used. Because component values are interpreted in a normalized range, they are specified independently of actual resolution. For example, the RGB triple (1, 1, 1) is white, regardless of whether the color buffers store 4, 8, or 12 bits each. Out-of-range components are typically clamped to the normalized range, not truncated or otherwise interpreted. For example, the RGB triple (1.4, 1.5, 0.9) is clamped to (1.0, 1.0, 0.9) before it's used to update the color buffer. Red, green, blue, alpha, and depth are always treated as components, never as indexes.

c
o
n
t
e
x
t

A complete set of OpenGL state variables. Note that framebuffer contents are not part of the OpenGL state, but that the configuration of the framebuffer is.

c
o
n
v
e

x

Condition of a polygon in which no straight line in the plane of the polygon intersects the polygon more than twice.

c
o
n
v
e
x
h
u
l
l

The smallest convex region enclosing a specified group of points. In two dimensions, the convex hull is found conceptually by stretching a rubber band around the points so that all of the points lie within the band.

c
o
o
r
d
i
n
a
t
e
s
y
s
t
e
m

In n-dimensional space, a set of n linearly independent vectors anchored to a point (called the origin). A group of coordinates specifies a point in spaceIn n-dimensional space, a set of n linearly independent vectors anchored to a point (called the origin). A group of coordinates specifies a point in space (or a vector from the origin) by indicating how far to travel along each vector to reach the point (or tip of the vector). (or a vector from the origin) by indicating how far to travel along each vector to reach the point (or tip of the vector).

c
u
l
l
i
n
g

The process of eliminating a front face or back face of a polygon so that the face isn't drawn.

c
u

r
r
e
n
t
m
a
t
r
i
x

A matrix that transforms coordinates in one coordinate system to coordinates of another system. There are three current matrices in OpenGL: the modelview matrix, which transforms object coordinates (coordinates specified by the programmer) to eye coordinates; the perspective matrix, which transforms eye coordinates to clip coordinates; and the texture matrix, which transforms specified or generated texture coordinates as described by the matrix. Each current matrix is the top element on a stack of matrices. Each of the three stacks can be manipulated with OpenGL matrix-manipulation commands.

c
u
r
r
e
n
t
r
a
s
t
e
r
p
o
s
i
t
i
o
n

A window coordinate position that specifies the placement of an image primitive when it's rasterized. The current raster position, and other current raster parameters, are updated when glRasterpos is called.

D

2/22/2020 • 2 minutes to read • Edit Online

depth Generally refers to the z window coordinate.

depthcueing A rendering technique that assigns color based on distance from the viewpoint.

displaylist

A named list of OpenGL commands. Display lists are always stored on the server, so display lists can be used to reduce the network traffic in client/server environments. The contents of a display list may be preprocessed, and might therefore execute more efficiently than the same set of OpenGL commands executed in immediate mode. Such preprocessing is especially important for computing intensive commands such as **glTexImage**.

dithering

A technique for increasing the perceived range of colors in an image at the cost of spatial resolution. Adjacent pixels are assigned differing color values. When viewed from a distance, these colors seem to blend into a single intermediate color. The technique is similar to the half-toning used in black-and-white publications to achieve shades of gray.

d o u b l e b u f f e r i n g

Using OpenGL contexts in which both front and back color buffers are double-buffered. Smooth animation is accomplished by rendering into only the back buffer (which isn't displayed), then causing the front and back buffers to be swapped.

E

2/22/2020 • 2 minutes to read • Edit Online

element A single component or index.

evaluation

The OpenGL process of generating object-coordinate vertices and parameters from previously specified Bézier equations.

execute

To call an OpenGL command in immediate mode or to call the display list of which the command is a part.

eyecoordinate

s

The coordinate system that follows transformation by the modelview matrix and that precedes transformation by the projection matrix. Lighting and application-specific clipping are done in eye coordinates.

F

2/22/2020 • 2 minutes to read • Edit Online

face

One side of a polygon. Each polygon has two faces: a front face and a back face. Only one face is ever visible in the window. Whether the back or front face is visible is effectively determined after the polygon is projected onto the window. After this projection, if the polygon's edges are directed clockwise, one of the faces is visible; if directed counterclockwise, the other face is visible. Whether clockwise corresponds to front or back (and counterclockwise corresponds to back or front) is determined by the OpenGL programmer.

flatshading

Refers to coloring a primitive with a single, constant color across its extent, rather than smoothly interpolating colors across the primitive. See Gouraud shading.

fog

A rendering technique that can be used to simulate atmospheric effects such as haze, fog, and smog by fading object colors to a background color based on distance from the viewer. Fog also aids in the perception of distance from the viewer, giving a depth cue. See also depth-cueing.

font

A group of graphical character representations usually used to display strings of text. The characters may be roman letters, mathematical symbols, Asian ideograms, Egyptian hieroglyphs, and so on.

fra

fragment

Graphic data generated by the rasterization of primitives. Each fragment corresponds to a single pixel and includes color, depth, and sometimes texture-coordinate values.

framebuffer

A stack of bitplanes. All the buffers of a given window or context. Sometimes includes all the pixel memory of the graphics hardware accelerator. See also bitplane.

front face See face.

frustum The view volume warped by perspective division.

G

2/22/2020 • 2 minutes to read • Edit Online

g
a
m
m
a
c
o
r
r
e
c
t
i
o
n

A function applied to colors stored in the frame buffer to correct for the nonlinear response of the eye (and sometimes of the monitor) to linear changes in color-intensity values.

g
e
o
m
e
t
r
i
c
m
o
d
e
l

The object-coordinate vertices and parameters that describe an object. Note that OpenGL doesn't define a syntax for geometric models, but rather a syntax and semantics for the rendering of geometric models.

g
e
o
m
e
t
r
i

A geometric model.

c
o
b
j
e
c
t

g
e
o
m
e
t
r
i
c
p
r
i
m
i
t
i
v
e

A point, line, or polygon.

G
o
u
r
a
u
d
s
h
a
d
i
n
g

Smooth interpolation of colors across a polygon or line segment. Colors are assigned at vertices and linearly interpolated across the primitive to produce a relatively smooth variation in color. Also called smooth shading.

g
r
o
u
p

A group of one, two, three, or four elements that represents each pixel of an image in client memory.

Thus, in the context of a client memory image, a group and a pixel are the same thing.

H

2/22/2020 • 2 minutes to read • Edit Online

half space The result of a plane dividing space. A plane divides space into two half spaces.

homogeneous coordinates A set of n+1 coordinates used to represent points in n-dimensional projective space. Points in projective space can be thought of as points in Euclidean space together with some points at infinity. The coordinates are homogenous because a scaling of each of the coordinates by the same nonzero constant doesn't alter the point to which the coordinates refer. Homogeneous coordinates are useful in the calculations of projective geometry, and thus in computer graphics, where scenes must be projected onto a window.

I

2/22/2020 • 2 minutes to read • Edit Online

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

image A rectangular array of pixels, either in client memory or in the framebuffer.

imageprimitive A bitmap or an image.

immediatemode

Mode in which an OpenGL command is called directly, rather than from a display list. No immediate-mode bit exists; the mode in immediate mode refers to usage of OpenGL, rather than to a specific bit of OpenGL state.

index

A single value that is interpreted as an absolute value, rather than as a normalized value in a specified range (as is a component). Color indexes are the names of colors, which are dereferenced by the display hardware using the color map. Indexes are typically masked, rather than clamped, when out of range. For example, the index 0xf7 is masked to 0x7 when written to a 4-bit buffer (color or stencil). Color indexes and stencil indexes are always treated as indexes, never as components.

I
R
I
S
G
L

Silicon Graphics' proprietary graphics library, developed from 1982 through 1992. OpenGL was designed with IRIS GL as a starting point.

J K

2/22/2020 • 2 minutes to read • Edit Online

j
a
g
g
i
e
s

Artifacts of aliased rendering. The edges of primitives that are rendered with aliasing are jagged rather than smooth. A near-horizontal aliased line, for example, is rendered as a set of horizontal lines on adjacent pixel rows, rather than as a smooth, continuous line.

L

2/22/2020 • 2 minutes to read • Edit Online

Lighting

The process of computing the color of a vertex based on current lights, material properties, and lighting-model modes.

Line

A straight region of finite width between two vertices. (Unlike mathematical lines, OpenGL lines have finite width and length.) Each segment of a strip of lines is itself a line.

Luminance

The perceived brightness of a surface. Often refers to a weighted average of red, green, and blue color values that gives the perceived brightness of the combination.

M

2/22/2020 • 2 minutes to read • Edit Online

matrices

Plural of matrix. See matrix.

matrix

A two-dimensional array of values. OpenGL matrices are all 4x4, though when they are stored in client memory they're treated as 1x16 single-dimension arrays.

modelviewmatrix

The 4x4 matrix that transforms points, lines, polygons, and raster positions from object coordinates to eye coordinates.

monitor

The device that displays the image in the framebuffer.

motionblurring

A technique that simulates what you get on a piece of film when you take a picture of a moving object, or when you move the camera when you take a picture of a stationary object. In animations without motion blur, object motion can appear jerky.

N

2/22/2020 • 2 minutes to read • Edit Online

network

A connection between two or more computers that allows each to transfer data to and from the others.

nonconvex

A state of a polygon in which a line in the plane of the polygon intersects the polygon more than twice.

normal

A three-component plane equation that defines the angular orientation, but not position, of a plane or surface.

normalize

To divide each of the components of a normal by the square root of the sum of their squares. Then, if the normal is thought of as a vector from the origin to the point (nx', ny', nx'), this vector has unit length:

nz' = nz/factor ny' = ny/factor nx' = nx/factor

normal vector See *normal*.

NURBS Non-Uniform Rational B-Spline. A common way to specify parametric curves and surfaces.

O

2/22/2020 • 2 minutes to read • Edit Online

object An object-coordinate model that is rendered as a collection of primitives.

object coordinates Coordinate system prior to any OpenGL transformation.

orthographic Nonperspective projection, as in some engineering drawings, with no foreshortening.

P

2/22/2020 • 2 minutes to read • Edit Online

parameter

A value passed as an argument to an OpenGL command. Sometimes one of the values passed by reference to an OpenGL command.

perspective division

The division of x, y, and z by w, carried out in clip coordinates. See also clip coordinates.

pixel

Picture element. The bits at location (x, y) of all the bitplanes in the framebuffer constitute the single pixel (x, y). In an image in client memory, a pixel is one group of elements. In OpenGL window coordinates, each pixel corresponds to a 1.0x1.0 screen area. The coordinates of the lower-left corner of the pixel names x, y are (x, y), and the upper-right corner are (x+1, y+1).

p

An exact location in space, which is rendered as a finite-diameter dot.

o
i
n
t

p
o
l
y
g
o
n

A near-planar surface bounded by edges specified by vertices. Each triangle of a triangle mesh is a polygon, as is each quadrilateral of a quadrilateral mesh. The rectangle specified by glRect is also a polygon.

p
r
i
m
i
t
i
v
e

A shape (such as a point, line, polygon, bitmap, or image) that can be drawn, stored, and manipulated as a discrete entity; elements from which large graphic designs are created.

p
r
o
j
e
c
t
i
o
n
m
a
t
r
i
x

The 4x4 matrix that transforms points, lines, polygons, and raster positions from eye coordinates to clip coordinates.

Q

2/22/2020 • 2 minutes to read • Edit Online

q u a d r i l a t e r a l	A polygon with four edges.

R

2/22/2020 • 2 minutes to read • Edit Online

rasterize

To convert a projected point, line, or polygon, or the pixels of a bitmap or image, to fragments, each corresponding to a pixel in the framebuffer. Note that all primitives are rasterized, not just points, lines, and polygons

rectangle

A quadrilateral whose alternate edges are parallel to each other in object coordinates. Polygons specified with glRect*( ) are always rectangles; other quadrilaterals might be rectangles.

rendering

Conversion of primitives specified in object coordinates to an image in the framebuffer. Rendering is the primary operation of OpenGL.

RGBA

The red, green, blue, and alpha color components of the RGBA mode.

R

GBAmode

An OpenGL context in which its color buffers store red, green, blue, and alpha color components, rather than color indexes.

S

2/22/2020 • 2 minutes to read • Edit Online

server

The computer on which OpenGL commands are executed. This might differ from the computer from which commands are issued. See also client.

shading

The process of interpolating color within the interior of a polygon, or between the vertices of a line, during rasterization.

single buffering

An OpenGL context without a back color buffer.

stipple

A one- or two-dimensional binary pattern that defeats the generation of fragments where its value is zero. Line stipples are one-dimensional and are applied relative to the start of a line. Polygon stipples are two-dimensional and are applied with a fixed orientation to the window.

T

2/22/2020 • 2 minutes to read • Edit Online

tessellation

Reduction of a portion of an analytic surface to a mesh of polygons, or of a portion of an analytic curve to a sequence of lines.

texel

A texture element. A texel is obtained from texture memory and represents the color of the texture to be applied to a corresponding fragment. See also fragment.

texture

A one- or two-dimensional image used to modify the color of fragments produced by rasterization. See also rasterize.

texturema

p
p
i
n
g

The process of applying an image (the texture) to a primitive. Texture mapping is often used to add realism to a scene. For example, you could apply a picture of a building facade to a polygon representing a wall. See also texture.

t
e
x
t
u
r
e
m
a
t
r
i
x

The 4x4 matrix that transforms texture coordinates from the coordinates that they're specified in to the coordinates that are used for interpolation and texture lookup.

t
r
a
n
s
f
o
r
m
a
t
i
o
n

A warping of space. In OpenGL, transformations are limited to projective transformations that include anything that can be represented by a 4x4 matrix. Such transformations include rotations, translations, (nonuniform) scalings along the coordinate axes, perspective transformations, and combinations of these.

t
r
i
a
n
g
l

A polygon with three edges. Triangles are always convex.

e

U V

2/22/2020 • 2 minutes to read • Edit Online

vertex — A point in three-dimensional space.

vertices — Preferred plural of vertex. See *vertex*.

viewpoint — The origin of either the eye- or the clip-coordinate system, depending on context. For example, when discussing lighting, the viewpoint is the origin of the eye-coordinate system. When discussing projection, the viewpoint is the origin of the clip-coordinate system. With a typical projection matrix, the eye-coordinate and clip-coordinate origins are at the same location.

viewvolume — The volume in clip coordinates whose coordinates satisfy the three conditions

- $w \le x \le w$
- $w \le y \le w$
- $w \le z \le w$

W

2/22/2020 • 2 minutes to read • Edit Online

window

A subregion of the framebuffer, usually rectangular, whose pixels all have the same buffer configuration. An OpenGL context renders to one window at a time.

window-aligned

When referring to line segments or polygon edges, implies that these are parallel to the window boundaries. (In OpenGL, the window is rectangular, with horizontal and vertical edges). When referring to a polygon pattern, implies that the pattern is fixed relative to the window origin.

windowcoordinate

s

The coordinate system of a window. It's important to distinguish between the names of pixels, which are discrete, and the window-coordinate system, which is continuous. For example, the pixel at the lower-left corner of a window is pixel (0, 0); the window coordinates of the center of this pixel are (0.5, 0.5, z). Note that window coordinates include a depth, or z, component, and that this component is continuous as well.

w
i
r
e
f
r
a
m
e

A representation of an object that contains line segments only. Typically, the line segments indicate polygon edges.

X Y Z

2/22/2020 • 2 minutes to read • Edit Online

X Window System A window system used by many of the machines on which OpenGL is implemented.

Appendix

2/22/2020 • 2 minutes to read • Edit Online

OpenGL, originally developed by Silicon Graphics Incorporated (SGI) for their graphics workstations, lets applications create high-quality color images independent of windowing systems, operating systems, and hardware.

The OpenGL Architecture Review Board (ARB), an industry consortium, is currently responsible for defining OpenGL. Members of the ARB include Silicon Graphics Incorporated, Microsoft Corporation, Intel, IBM, and Digital Equipment Corporation.

The official reference document for OpenGL version 1, is the *OpenGL Reference Manual*, by the OpenGL Architecture Review Board (ISBN 0-201-63276-4). The official guide to learning OpenGL version 1, is the *OpenGL Programming Guide*, by Jackie Neider, Tom Davis, and Mason Woo (ISBN 0-201-63274-8). Both books are published by Addison-Wesley.