Index of /opengl/docs/man_pages/hardcopy/GL/html/glu/

Name	Last modified	Size	Description
Parent Directory			
begincurve.html	09-Sep-97 16:18	1K	
beginpolygon.html	09-Sep-97 16:18	2K	
beginsurface.html	09-Sep-97 16:18	2K	
begintrim.html	09-Sep-97 16:18	3K	
build1dmipmaps.html	09-Sep-97 16:18	5K	
build2dmipmaps.html	09-Sep-97 16:18	6K	
cylinder.html	09-Sep-97 16:18	2K	
deletenurbsrenderer.h+	09-Sep-97 16:18	1K	
deletequadric.html	09-Sep-97 16:18	1K	
deletetess.html	09-Sep-97 16:18	1K	
disk.html	09-Sep-97 16:18	2K	
errorstring.html	09-Sep-97 16:18	1K	
getnurbsproperty.html	09-Sep-97 16:18	1K	
getstring.html	09-Sep-97 16:18	2K	
gettessproperty.html	09-Sep-97 16:18	1K	
loadsamplingmatrices.+	09-Sep-97 16:18	2K	
lookat.html	09-Sep-97 16:18	2K	
newnurbsrenderer.html	09-Sep-97 16:18	1K	
newquadric.html	09-Sep-97 16:18	1K	
newtess.html	09-Sep-97 16:18	1K	
nextcontour.html	09-Sep-97 16:18	3K	
nurbscallback.html	09-Sep-97 16:18	9K	
nurbscallbackdataext.+	09-Sep-97 16:18	1K	
nurbscurve.html	09-Sep-97 16:18	3K	

nurbsproperty.html	09-Sep-97 16:18	9K
nurbssurface.html	09-Sep-97 16:18	4K
ortho2d.html	09-Sep-97 16:18	1K
partialdisk.html	09-Sep-97 16:18	2K
perspective.html	09-Sep-97 16:18	2K
pickmatrix.html	09-Sep-97 16:18	2K
project.html	09-Sep-97 16:18	2K
pwlcurve.html	09-Sep-97 16:18	2K
quadriccallback.html	09-Sep-97 16:18	1K
quadricdrawstyle.html	09-Sep-97 16:18	1K
quadricnormals.html	09-Sep-97 16:18	1K
quadricorientation.ht+	09-Sep-97 16:18	1K
quadrictexture.html	09-Sep-97 16:18	1K
scaleimage.html	09-Sep-97 16:18	3K
sphere.html	09-Sep-97 16:18	1K
tessbegincontour.html	09-Sep-97 16:18	1K
tessbeginpolygon.html	09-Sep-97 16:18	2K
tesscallback.html	09-Sep-97 16:18	12K
tessendpolygon.html	09-Sep-97 16:18	2K
tessnormal.html	09-Sep-97 16:18	2K
tessproperty.html	09-Sep-97 16:18	4K
tessvertex.html	09-Sep-97 16:18	3K
unproject.html	09-Sep-97 16:18	2K

NAME

gluBeginCurve, **gluEndCurve** - delimit a NURBS curve definition

C SPECIFICATION

void **gluBeginCurve**(GLUnurbs* *nurb*)

void **gluEndCurve**(GLUnurbs* *nurb*)

PARAMETERS

nurb Specifies the NURBS object (created with **gluNewNurbsRenderer**).

DESCRIPTION

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

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

EXAMPLE

The following commands render a textured NURBS curve with normals; texture coordinates and normals are also specified as NURBS curves:

```
gluBeginCurve(nobj);
   gluNurbsCurve(nobj, ..., GL_MAP1_TEXTURE_COORD_2);
   gluNurbsCurve(nobj, ..., GL_MAP1_NORMAL);
   gluNurbsCurve(nobj, ..., GL_MAP1_VERTEX_4);
gluEndCurve(nobj);
```

SEE ALSO

gluBeginSurface, **gluBeginTrim**, **gluNewNurbsRenderer**, **gluNurbsCurve**, **glPopAttrib**, **glPushAttrib**

NAME

gluBeginPolygon, gluEndPolygon - delimit a polygon description

C SPECIFICATION

void **gluBeginPolygon**(GLUtesselator* *tess*)

void **gluEndPolygon**(GLUtesselator* *tess*)

PARAMETERS

tess Specifies the tessellation object (created with **gluNewTess**).

DESCRIPTION

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

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

NOTES

This command is obsolete and is provided for backward compatibility only. Calls to **gluBeginPolygon** are mapped to **gluTessBeginPolygon** followed by **gluTessBeginContour**. Calls to **gluEndPolygon** are mapped to **gluTessEndContour** followed by **gluTessEndPolygon**.

EXAMPLE

A quadrilateral with a triangular hole in it can be described like this:

```
gluBeginPolygon(tobj);
   gluTessVertex(tobj, v1, v1);
   gluTessVertex(tobj, v2, v2);
   gluTessVertex(tobj, v3, v3);
   gluTessVertex(tobj, v4, v4); gluNextContour(tobj,
GLU_INTERIOR);
   gluTessVertex(tobj, v5, v5);
   gluTessVertex(tobj, v6, v6);
   gluTessVertex(tobj, v7, v7); gluEndPolygon(tobj);
```

SEE ALSO

gluNewTess, **gluNextContour**, **gluTessCallback**, **gluTessVertex**, **gluTessBeginPolygon**, **gluTessBeginContour**

NAME

gluBeginSurface, **gluEndSurface** - delimit a NURBS surface definition

C SPECIFICATION

void **gluBeginSurface**(GLUnurbs* *nurb*)

void **gluEndSurface**(GLUnurbs* *nurb*)

PARAMETERS

nurb Specifies the NURBS object (created with **gluNewNurbsRenderer**).

DESCRIPTION

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

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

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

EXAMPLE

The following commands 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);
```

SEE ALSO

gluBeginCurve, **gluBeginTrim**, **gluNewNurbsRenderer**, **gluNurbsCurve**, **gluNurbsSurface**, **gluPwlCurve**

NAME

gluBeginTrim, **gluEndTrim** - delimit a NURBS trimming loop definition

C SPECIFICATION

```
void gluBeginTrim( GLUnurbs* nurb )

void gluEndTrim( GLUnurbs* nurb )
```

PARAMETERS

nurb Specifies the NURBS object (created with **gluNewNurbsRenderer**).

DESCRIPTION

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

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

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

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

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

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

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

EXAMPLE

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);
```

SEE ALSO

gluBeginSurface, **gluNewNurbsRenderer**, **gluNurbsCallback**, **gluNurbsCurve**, **gluPwlCurve**

NAME

gluBuild1DMipmaps - builds a 1-D mipmap

C SPECIFICATION

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

PARAMETERS

target Specifies the target texture. Must be **GL_TEXTURE_1D**.

internalFormat Requests the internal storage format of the texture image. Must be 1, 2, 3, or 4 or one of the following symbolic constants: **GL_ALPHA**, **GL_ALPHA4**, **GL_ALPHA8**, **GL_ALPHA12**, **GL_ALPHA16**, **GL_LUMINANCE**, **GL_LUMINANCE4**, **GL_LUMINANCE8**, **GL_LUMINANCE12**, **GL_LUMINANCE16**, **GL_LUMINANCE_ALPHA**, **GL_LUMINANCE4_ALPHA4**, **GL_LUMINANCE6_ALPHA2**, **GL_LUMINANCE8_ALPHA8**, **GL_LUMINANCE12_ALPHA4**, **GL_LUMINANCE12_ALPHA12**, **GL_LUMINANCE16_ALPHA16**, **GL_INTENSITY**, **GL_INTENSITY4**, **GL_INTENSITY8**, **GL_INTENSITY12**, **GL_INTENSITY16**, **GL_RGB**, **GL_R3_G3_B2**, **GL_RGB4**, **GL_RGB5**, **GL_RGB8**, **GL_RGB10**, **GL_RGB12**, **GL_RGB16**, **GL_RGBA**, **GL_RGBA2**, **GL_RGBA4**, **GL_RGB5_A1**, **GL_RGBA8**, **GL_RGB10_A2**, **GL_RGBA12** or **GL_RGBA16**.

width Specifies the width, in pixels, of the texture image.

format Specifies the format of the pixel data. Must be one of **GL_COLOR_INDEX**, **GL_RED**, **GL_GREEN**, **GL_BLUE**, **GL_ALPHA**, **GL_RGB**, **GL_RGBA**, **GL_LUMINANCE**, and **GL_LUMINANCE_ALPHA**.

type Specifies the data type for *data*. Must be one of **GL_UNSIGNED_BYTE**, **GL_BYTE**, **GL_BITMAP**, **GL_UNSIGNED_SHORT**, **GL_SHORT**, **GL_UNSIGNED_INT**, **GL_INT**, or **GL_FLOAT**.

data Specifies a pointer to the image data in memory.

DESCRIPTION

gluBuild1DMipmaps builds a series of prefiltered 1-D texture maps of decreasing resolutions called a mipmap. This is used for the antialiasing of texture mapped primitives.

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

Initially, the *width* of *data* is checked to see if it is a power of two. If not, a copy of *data* (not *data*) is scaled up or down to the nearest power of two. This copy will be used for subsequent mipmapping operations described below. (If *width* is exactly between powers of 2, then the copy of *data* will scale upwards.) For example, if *width* is 57 then a copy of *data* will scale up to 64 before mipmapping takes place.

Then, proxy textures (see **glTexImage1D**) are used to determine if the implementation can fit the requested texture. If not, *width* is continually halved until it fits.

Next, a series of mipmap levels is built by decimating a copy of *data* in half until size 1 is reached. At each level, each texel in the halved mipmap level is an average of the corresponding two texels in the larger mipmap level.

glTexImage1D is called to load each of these mipmap levels. Level 0 is a copy of *data*. The highest level is log2(width). For example, if width is 64 and the implementation can store a texture of this size, the following mipmap levels are built: 64x1, 32x1, 16x1, 8x1, 4x1, 2x1 and 1x1. These correspond to levels 0 through 6, respectively.

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

NOTES

Note that there is no direct way of querying the maximum level. This can be derived indirectly via **glGetTexLevelParameter**. First, query for the width actually used at level 0. (The width may not be equal to *width* since proxy textures might have scaled it to fit the implementation.) Then the maximum level can be derived from the formula log2(width).

ERRORS

GLU_INVALID_VALUE is returned if *width* is < 1.

GLU_INVALID_ENUM is returned if *internalFormat*, *format* or *type* are not legal.

SEE ALSO

glDrawPixels, **glTexImage1D**, **glTexImage2D**, **gluBuild2DMipmaps**, **gluErrorString**, **gluScaleImage**

NAME

gluBuild2DMipmaps - builds a 2-D mipmap

C SPECIFICATION

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

PARAMETERS

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

internalFormat Requests the internal storage format of the texture image. Must be 1, 2, 3, or 4 or one of the following symbolic constants: **GL_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, *height* Specifies the width and height, respectively, in pixels of the texture image.

format Specifies the format of the pixel data. Must be one of: **GL_COLOR_INDEX**, **GL_RED**, **GL_GREEN**, **GL_BLUE**, **GL_ALPHA**, **GL_RGB**, **GL_RGBA**, **GL_LUMINANCE**, and **GL_LUMINANCE_ALPHA**.

type Specifies the data type for *data*. Must be one of: **GL_UNSIGNED_BYTE**, **GL_BYTE**, **GL_BITMAP**, **GL_UNSIGNED_SHORT**, **GL_SHORT**, **GL_UNSIGNED_INT**, **GL_INT**, or **GL_FLOAT**.

data Specifies a pointer to the image data in memory.

DESCRIPTION

gluBuild2DMipmaps builds a series of prefiltered 2-D texture maps of decreasing resolutions called a mipmap. This is used for the antialiasing of texture mapped primitives.

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

Initially, the *width* and *height* of *data* are checked to see if they are a power of two. If not, a copy of *data* (not *data*), is scaled up or down to the nearest power of two. This copy will be used for subsequent mipmapping operations described below. (If *width* or *height* is exactly between powers of 2, then the copy of *data* will scale upwards.) For example, if *width* is 57 and *height* is 23 then a copy of *data* will scale up to 64 and down to 16, respectively, before mipmapping takes place.

Then, proxy textures (see **glTexImage2D**) are used to determine if the implementation can fit the requested texture. If not, both dimensions are continually halved until it fits. (If the OpenGL version is <= 1.0, both maximum texture dimensions are clamped to the value returned by **glGetIntegerv** with the argument **GL_MAX_TEXTURE_SIZE**.)

Next, a series of mipmap levels is built by decimating a copy of *data* in half along both dimensions until size 1x1 is reached. At each level, each texel in the halved mipmap level is an average of the corresponding four texels in the larger mipmap level. (In the case of rectangular images, the decimation will ultimately reach an N x 1 or 1 x N configuration. Here, two texels are averaged instead.)

glTexImage2D is called to load each of these mipmap levels. Level 0 is a copy of *data*. The highest level is log2(max(width,height)). For example, if width is 64 and height is 16 and the implementation can store a texture of this size, the following mipmap levels are built: 64x16, 32x8, 16x4, 8x2, 4x1, 2x1 and 1x1. These correspond to levels 0 through 6, respectively.

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

NOTES

Note that there is no direct way of querying the maximum level. This can be derived indirectly via **glGetTexLevelParameter**. First, query for the width & height actually used at level 0. (The width & height may not be equal to *width* & *height* respectively since proxy textures might have scaled them to fit the implementation.) Then the maximum level can be derived from the formula log2(max(width,height)).

ERRORS

GLU_INVALID_VALUE is returned if *width* or *height* are < 1.

GLU_INVALID_ENUM is returned if *internalFormat*, *format* or *type* are not legal.

SEE ALSO

glDrawPixels, **glTexImage1D**, **glTexImage2D**, **gluBuild1DMipmaps**, **gluErrorString**, **gluScaleImage**

NAME

gluCylinder - draw a cylinder

C SPECIFICATION

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

PARAMETERS

quad Specifies the quadrics object (created with **gluNewQuadric**).

base Specifies the radius of the cylinder at *z* = 0.

top Specifies the radius of the cylinder at *z* = *height*.

height Specifies the height of the cylinder.

slices Specifies the number of subdivisions around the *z* axis.

stacks Specifies the number of subdivisions along the *z* axis.

DESCRIPTION

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

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

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

If texturing is turned on (with **gluQuadricTexture**), then texture coordinates are generated so that *t* ranges linearly from 0.0 at *z* = 0 to 1.0 at *z* = *height*, and *s* ranges from 0.0 at the +*y* axis, to 0.25 at the +*x* axis, to 0.5 at the -*y* axis, to 0.75 at the -*x* axis, and back to 1.0 at the +*y* axis.

SEE ALSO

gluDisk, **gluNewQuadric**, **gluPartialDisk**, **gluQuadricTexture**, **gluSphere**

NAME

gluDeleteNurbsRenderer - destroy a NURBS object

C SPECIFICATION

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

PARAMETERS

nurb Specifies the NURBS object to be destroyed.

DESCRIPTION

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

SEE ALSO

gluNewNurbsRenderer

NAME

gluDeleteQuadric - destroy a quadrics object

C SPECIFICATION

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

PARAMETERS

quad Specifies the quadrics object to be destroyed.

DESCRIPTION

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

SEE ALSO

gluNewQuadric

NAME

gluDeleteTess - destroy a tessellation object

C SPECIFICATION

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

PARAMETERS

tess Specifies the tessellation object to destroy.

DESCRIPTION

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

SEE ALSO

gluBeginPolygon, **gluNewTess**, **gluTessCallback**

NAME

gluDisk - draw a disk

C SPECIFICATION

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

PARAMETERS

quad Specifies the quadrics object (created with **gluNewQuadric**).

inner Specifies the inner radius of the disk (may be 0).

outer Specifies the outer radius of the disk.

slices Specifies the number of subdivisions around the *z* axis.

loops Specifies the number of concentric rings about the origin into which the disk is subdivided.

DESCRIPTION

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

With respect to orientation, the +*z* side of the disk is considered to be "outside" (see **gluQuadricOrientation**). This means that if the orientation is set to **GLU_OUTSIDE**, then any normals generated point along the +*z* axis. Otherwise, they point along the -*z* axis.

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

SEE ALSO

gluCylinder, **gluNewQuadric**, **gluPartialDisk**, **gluQuadricOrientation**, **gluQuadricTexture**, **gluSphere**

NAME

gluErrorString - produce an error string from a GL or GLU error code

C SPECIFICATION

const GLubyte * **gluErrorString**(GLenum *error*)

PARAMETERS

error Specifies a GL or GLU error code.

DESCRIPTION

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

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

SEE ALSO

glGetError, **gluNurbsCallback**, **gluQuadricCallback**, **gluTessCallback**

NAME

gluGetNurbsProperty - get a NURBS property

C SPECIFICATION

```
void gluGetNurbsProperty( GLUnurbs* nurb,
                          GLenum property,
                          GLfloat* data )
```

PARAMETERS

nurb Specifies the NURBS object (created with **gluNewNurbsRenderer**).

property Specifies the property whose value is to be fetched. Valid values are **GLU_CULLING**, **GLU_SAMPLING_TOLERANCE**, **GLU_DISPLAY_MODE**, **GLU_AUTO_LOAD_MATRIX**, **GLU_PARAMETRIC_TOLERANCE**, **GLU_SAMPLING_METHOD**, **GLU_U_STEP**, and **GLU_V_STEP**.

data Specifies a pointer to the location into which the value of the named property is written.

DESCRIPTION

gluGetNurbsProperty retrieves properties stored in a NURBS object. These properties affect the way that NURBS curves and surfaces are rendered. See the **gluNurbsProperty** reference page for information about what the properties are and what they do.

SEE ALSO

gluNewNurbsRenderer, **gluNurbsProperty**

NAME

gluGetString - return a string describing the GLU version or GLU extensions

C SPECIFICATION

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

PARAMETERS

name Specifies a symbolic constant, one of **GLU_VERSION**, or **GLU_EXTENSIONS**.

DESCRIPTION

gluGetString returns a pointer to a static string describing the GLU version or the GLU extensions that are supported.

The version number is one of the following forms:

major_number.*minor_number*
major_number.*minor_number*.*release_number*.

The version string is of the following form:

version number<space>vendor-specific information

Vendor-specific information is optional. Its format and contents depend on the implementation.

The standard GLU contains a basic set of features and capabilities. If a company or group of companies wish to support other features, these may be included as extensions to the GLU. If *name* is **GLU_EXTENSIONS**, then **gluGetString** returns a space-separated list of names of supported GLU extensions. (Extension names never contain spaces.)

All strings are null-terminated.

NOTES

gluGetString only returns information about GLU extensions. Call **glGetString** to get a list of GL extensions.

gluGetString is an initialization routine. Calling it after a **glNewList** results in undefined behavior.

ERRORS

NULL is returned if *name* is not **GLU_VERSION** or **GLU_EXTENSIONS**.

SEE ALSO

glGetString

NAME

gluGetTessProperty - get a tessellation object property

C SPECIFICATION

```
void gluGetTessProperty( GLUtesselator* tess,
                         GLenum which,
                         GLdouble* data )
```

PARAMETERS

tess Specifies the tessellation object (created with **gluNewTess**).

which Specifies the property whose value is to be fetched. Valid values are **GLU_TESS_WINDING_RULE**, **GLU_TESS_BOUNDARY_ONLY**, and **GLU_TESS_TOLERANCE**.

data Specifies a pointer to the location into which the value of the named property is written.

DESCRIPTION

gluGetTessProperty retrieves properties stored in a tessellation object. These properties affect the way that tessellation objects are interpreted and rendered. See the **gluTessProperty** reference page for information about the properties and what they do.

SEE ALSO

gluNewTess, **gluTessProperty**

NAME

gluLoadSamplingMatrices - load NURBS sampling and culling matrices

C SPECIFICATION

```
void gluLoadSamplingMatrices( GLUnurbs* nurb,
                              const GLfloat *model,
                              const GLfloat *perspective,
                              const GLint *view )
```

PARAMETERS

nurb Specifies the NURBS object (created with **gluNewNurbsRenderer**).

model Specifies a modelview matrix (as from a **glGetFloatv** call).

perspective Specifies a projection matrix (as from a **glGetFloatv** call).

view Specifies a viewport (as from a **glGetIntegerv** call).

DESCRIPTION

gluLoadSamplingMatrices uses *model*, *perspective*, and *view* to recompute the sampling and culling matrices stored in *nurb*. The sampling matrix determines how finely a NURBS curve or surface must be tessellated to satisfy the sampling tolerance (as determined by the **GLU_SAMPLING_TOLERANCE** property). The culling matrix is used in deciding if a NURBS curve or surface should be culled before rendering (when the **GLU_CULLING** property is turned on).

gluLoadSamplingMatrices is necessary only if the **GLU_AUTO_LOAD_MATRIX** property is turned off (see **gluNurbsProperty**). Although it can be convenient to leave the **GLU_AUTO_LOAD_MATRIX** property turned on, there can be a performance penalty for doing so. (A round trip to the GL server is needed to fetch the current values of the modelview matrix, projection matrix, and viewport.)

SEE ALSO

gluGetNurbsProperty, **gluNewNurbsRenderer**, **gluNurbsProperty**

NAME

gluLookAt - define a viewing transformation

C SPECIFICATION

```
void gluLookAt( GLdouble eyeX,
                GLdouble eyeY,
                GLdouble eyeZ,
                GLdouble centerX,
                GLdouble centerY,
                GLdouble centerZ,
                GLdouble upX,
                GLdouble upY,
                GLdouble upZ )
```

PARAMETERS

eyeX, *eyeY*, *eyeZ*
Specifies the position of the eye point.

centerX, *centerY*, *centerZ*
Specifies the position of the reference point.

upX, *upY*, *upZ* Specifies the direction of the *up* vector.

DESCRIPTION

gluLookAt creates a viewing matrix derived from an eye point, a reference point indicating the center of the scene, and an *UP* vector.

The matrix maps the reference point to the negative *z* axis and the eye point to the origin. When a typical projection matrix is used, the center of the scene therefore maps to the center of the viewport. Similarly, the direction described by the *UP* vector projected onto the viewing plane is mapped to the positive *y* axis so that it points upward in the viewport. The *UP* vector must not be parallel to the line of sight from the eye point to the reference point.

Let

```
    ( centerX   -   eyeX  )
F = |                     |
    | centerY   -   eyeY  |
    ( centerZ   -   eyeZ  )
```

Let *UP* be the vector (upX,upY,upZ).

```
Then normalize as follows: f = _____
                               ||F||

UP' = ______
      ||UP||
```

Finally, let s = f x UP', and u = s x f.

M is then constructed as follows:

```
    ( s[0]   s[1]   s[2]  0  )
    | u[0]   u[1]   u[2]  0  |
M = |                        |
    |-f[0]  -f[1]  -f[2]  0  |
    |  0      0      0    1  |
    (                        )
```

and **gluLookAt** is equivalent to glMultMatrixf(M);
glTranslated (-eyex, -eyey, -eyez);

SEE ALSO

glFrustum, **gluPerspective**

NAME

gluNewNurbsRenderer - create a NURBS object

C SPECIFICATION

```
GLUnurbs* gluNewNurbsRenderer( void )
```

DESCRIPTION

gluNewNurbsRenderer creates and returns a pointer to a new NURBS object. This object must be referred to when calling NURBS rendering and control functions. A return value of 0 means that there is not enough memory to allocate the object.

SEE ALSO

gluBeginCurve, **gluBeginSurface**, **gluBeginTrim**, **gluDeleteNurbsRenderer**, **gluNurbsCallback**, **gluNurbsProperty**

NAME

gluNewQuadric - create a quadrics object

C SPECIFICATION

```
GLUquadric* gluNewQuadric( void )
```

DESCRIPTION

gluNewQuadric creates and returns a pointer to a new quadrics object. This object must be referred to when calling quadrics rendering and control functions. A return value of 0 means that there is not enough memory to allocate the object.

SEE ALSO

gluCylinder, **gluDeleteQuadric**, **gluDisk**, **gluPartialDisk**, **gluQuadricCallback**, **gluQuadricDrawStyle**, **gluQuadricNormals**, **gluQuadricOrientation**, **gluQuadricTexture**, **gluSphere**

NAME

gluNewTess - create a tessellation object

C SPECIFICATION

GLUtesselator* **gluNewTess**(void)

DESCRIPTION

gluNewTess creates and returns a pointer to a new tessellation object. This object must be referred to when calling tessellation functions. A return value of 0 means that there is not enough memory to allocate the object.

SEE ALSO

gluTessBeginPolygon, **gluDeleteTess**, **gluTessCallback**

NAME

gluNextContour - mark the beginning of another contour

C SPECIFICATION

```
void gluNextContour( GLUtesselator* tess,
                     GLenum type )
```

PARAMETERS

tess Specifies the tessellation object (created with **gluNewTess**).

type Specifies the type of the contour being defined. Valid values are **GLU_EXTERIOR**, **GLU_INTERIOR**, **GLU_UNKNOWN**, **GLU_CCW**, and **GLU_CW**.

DESCRIPTION

gluNextContour is used in describing polygons with multiple contours. After the first contour has been described through a series of **gluTessVertex** calls, a **gluNextContour** call indicates that the previous contour is complete and that the next contour is about to begin. Another series of **gluTessVertex** calls is then used to describe the new contour. This process can be repeated until all contours have been described.

type defines what type of contour follows. The legal contour types are as follows:

GLU_EXTERIOR An exterior contour defines an exterior boundary of the polygon.

GLU_INTERIOR An interior contour defines an interior boundary of the polygon (such as a hole).

GLU_UNKNOWN An unknown contour is analyzed by the library to determine if it is interior or exterior.

GLU_CCW,

GLU_CW The first **GLU_CCW** or **GLU_CW** contour defined is considered to be exterior. All other contours are considered to be exterior if they are oriented in the same direction (clockwise or counterclockwise) as the first contour, and interior if they are not.

If one contour is of type **GLU_CCW** or **GLU_CW**, then all contours must be of the same type (if they are not, then all **GLU_CCW** and **GLU_CW** contours will be changed to **GLU_UNKNOWN**).

Note that there is no real difference between the **GLU_CCW** and **GLU_CW** contour types.

Before the first contour is described, **gluNextContour** can be called to define the type of the first contour. If **gluNextContour** is not called before the first contour, then the first contour is marked **GLU_EXTERIOR**.

This command is obsolete and is provided for backward compatibility only. Calls to **gluNextContour** are mapped to **gluTessEndContour** followed by **gluTessBeginContour**.

EXAMPLE

A quadrilateral with a triangular hole in it can be described as follows:

```
gluBeginPolygon(tobj);
   gluTessVertex(tobj, v1, v1);
   gluTessVertex(tobj, v2, v2);
   gluTessVertex(tobj, v3, v3);
   gluTessVertex(tobj, v4, v4); gluNextContour(tobj,
GLU_INTERIOR);
   gluTessVertex(tobj, v5, v5);
   gluTessVertex(tobj, v6, v6);
   gluTessVertex(tobj, v7, v7); gluEndPolygon(tobj);
```

SEE ALSO

gluBeginPolygon, **gluNewTess**, **gluTessCallback**, **gluTessVertex**, **gluTessBeginContour**

NAME

gluNurbsCallback - define a callback for a NURBS object

C SPECIFICATION

```
void gluNurbsCallback( GLUnurbs* nurb,
                       GLenum which,
                       GLvoid (*CallBackFunc)( )
```

PARAMETERS

nurb Specifies the NURBS object (created with **gluNewNurbsRenderer**).

which Specifies the callback being defined. Valid values are **GLU_NURBS_BEGIN_EXT**, **GLU_NURBS_VERTEX_EXT**, **GLU_NORMAL_EXT**, **GLU_NURBS_COLOR_EXT**, **GLU_NURBS_TEXTURE_COORD_EXT**, **GLU_END_EXT**, **GLU_NURBS_BEGIN_DATA_EXT**, **GLU_NURBS_VERTEX_DATA_EXT**, **GLU_NORMAL_DATA_EXT**, **GLU_NURBS_COLOR_DATA_EXT**, **GLU_NURBS_TEXTURE_COORD_DATA_EXT**, **GLU_END_DATA_EXT**, and **GLU_ERROR**.

CallBackFunc Specifies the function that the callback calls.

DESCRIPTION

gluNurbsCallback is used to define a callback to be used by a NURBS object. If the specified callback is already defined, then it is replaced. If *CallBackFunc* is NULL, then this callback will not get invoked and the related data, if any, will be lost.

Except the error callback, these callbacks are used by NURBS tessellator (when **GLU_NURBS_MODE_EXT** is set to be **GLU_NURBS_TESSELLATOR_EXT**) to return back the openGL polygon primitives resulted from the tessellation. Note that there are two versions of each callback: one with a user data pointer and one without. If both versions for a particular callback are specified then the callback with the user data pointer will be used. Note that "userData" is a copy of the pointer that was specified at the last call to **gluNurbsCallbackDataEXT**.

The error callback function is effective no matter which value that **GLU_NURBS_MODE_EXT** is set to. All other callback functions are effective only when **GLU_NURBS_MODE_EXT** is set to **GLU_NURBS_TESSELLATOR_EXT**.

The legal callbacks are as follows:

GLU_NURBS_BEGIN_EXT

The begin callback indicates the start of a primitive. The function takes a single argument of type GLenum which can be one of **GL_LINES**, **GL_LINE_STRIPS**, **GL_TRIANGLE_FAN**, **GL_TRIANGLE_STRIP**, **GL_TRIANGLES**, or **GL_QUAD_STRIP**. The default begin callback function is NULL. The function prototype for this callback looks like:

```
void begin ( GLenum type );
```

GLU_NURBS_BEGIN_DATA_EXT

The same as the **GLU_NURBS_BEGIN_EXT** callback except that it takes an additional pointer argument. This pointer is a copy of the pointer that was specified at the last call to **gluNurbsCallbackDataEXT**. The default callback function is NULL. The function prototype for this callback function looks like:

```
void beginData (GLenum type, void *userData);
```

GLU_NURBS_VERTEX_EXT

The vertex callback indicates a vertex of the primitive. The coordinates of the vertex are stored in the parameter "vertex". All the generated vertices have dimension 3, that is, homogeneous coordinates have been transformed into affine coordinates. The default vertex callback function is NULL. The function prototype for this callback function looks like:

```
void vertex ( GLfloat *vertex );
```

GLU_NURBS_VERTEX_DATA_EXT

The same as the **GLU_NURBS_VERTEX_EXT** callback except that it takes an additional pointer argument. This pointer is a copy of the pointer that was specified at the last call to **gluNurbsCallbackDataEXT**. The default callback function is NULL. The function prototype for this callback function looks like:

void vertexData (GLfloat *vertex, void *userData);

GLU_NORMAL_EXT

The normal callback is invoked as the vertex normal is generated. The components of the normal are stored in the parameter "normal". In the case of a NURBS curve, the callback function is effective only when the user provides a normal map (**GL_MAP1_NORMAL**). In the case of a NURBS surface, if a normal map (**GL_MAP2_NORMAL**) is provided, then the generated normal is computed from the normal map. If a normal map is not provided then a surface normal is computed in a manner similar to that described for evaluators when **GL_AUTO_NORMAL** is enabled. The default normal callback function is NULL. The function prototype for this callback function looks like:
void normal (GLfloat *normal);

GLU_NORMAL_DATA_EXT

The same as the **GLU_NURBS_NORMAL_EXT** callback except that it takes an additional pointer argument. This pointer is a copy of the pointer that was specified at the last call to **gluNurbsCallbackDataEXT**. The default callback function is NULL. The function prototype for this callback function looks like:
void normalData (GLfloat *normal, void *userData);

GLU_NURBS_COLOR_EXT

The color callback is invoked as the color of a vertex is generated. The components of the color are stored in the parameter "color". This callback is effective only when the user provides a color map (**GL_MAP1_COLOR_4** or **GL_MAP2_COLOR_4**). "color" contains four components: R,G,B,A. The default color callback function is NULL. The prototype for this callback function looks like:
void color (GLfloat *color);

GLU_NURBS_COLOR_DATA_EXT

The same as the **GLU_NURBS_COLOR_EXT** callback except that it takes an additional pointer argument. This pointer is a copy of the pointer

that was specified at the last call to **gluNurbsCallbackDataEXT**. The default callback function is NULL. The function prototype for this callback function looks like:
void colorData (GLfloat *color, void *userData);

GLU_NURBS_TEXTURE_COORD_EXT

The texture callback is invoked as the texture coordinates of a vertex are generated. These coordinates are stored in the parameter "texCoord". The number of texture coordinates can be 1, 2, 3, or 4 depending on which type of texture map is specified (**GL_MAP*_TEXTURE_COORD_1**, **GL_MAP*_TEXTURE_COORD_2**, **GL_MAP*_TEXTURE_COORD_3**, **GL_MAP*_TEXTURE_COORD_4** where * can be either 1 or 2). If no texture map is specified, this callback function will not be called. The default texture callback function is NULL. The function prototype for this callback function looks like:
void texCoord (GLfloat *texCoord);

GLU_NURBS_TEXTURE_COORD_DATA_EXT

The same as the **GLU_NURBS_TEXTURE_COORD_EXT** callback except that it takes an additional pointer argument. This pointer is a copy of the pointer that was specified at the last call to **gluNurbsCallbackDataEXT**. The default callback function is NULL. The function prototype for this callback function looks like:
void texCoordData (GLfloat *texCoord, void *userData);

GLU_END_EXT

The end callback is invoked at the end of a primitive. The default end callback function is NULL. The function prototype for this callback function looks like:
void end (void);

GLU_END_DATA_EXT

The same as the **GLU_NURBS_TEXTURE_COORD_EXT** callback except that it takes an additional pointer argument. This pointer is a copy of the pointer that was specified at the last call to **gluNurbsCallbackDataEXT**. The default callback function is NULL. The function prototype for this

callback function looks like:
void endData (void *userData);

GLU_ERROR The error function is called when an error is encountered. Its single argument is of type GLenum, and it indicates the specific error that occurred. There are 37 errors unique to NURBS named **GLU_NURBS_ERROR1** through **GLU_NURBS_ERROR37**. Character strings describing these errors can be retrieved with **gluErrorString**.

SEE ALSO

gluErrorString, **gluNewNurbsRenderer**

NAME

gluNurbsCallbackDataEXT - set a user data pointer

C SPECIFICATION

```
void gluNurbsCallbackDataEXT( GLUnurbs* nurb,
                              GLvoid* userData )
```

PARAMETERS

nurb Specifies the NURBS object (created with **gluNewNurbsRenderer**).

userData Specifies a pointer to the user's data.

DESCRIPTION

gluNurbsCallbackDataEXT is used to pass a pointer to the application's data to NURBS tessellator. A copy of this pointer will be passed by the tessellator in the NURBS callback functions (set by **gluNurbsCallback**).

SEE ALSO

gluNurbsCallback

NAME

gluNurbsCurve - define the shape of a NURBS curve

C SPECIFICATION

```
void gluNurbsCurve( GLUnurbs* nurb,
                    GLint knotCount,
                    GLfloat *knots,
                    GLint stride,
                    GLfloat *control,
                    GLint order,
                    GLenum type )
```

PARAMETERS

nurb Specifies the NURBS object (created with **gluNewNurbsRenderer**).

knotCount Specifies the number of knots in *knots*. *knotCount* equals the number of control points plus the order.

knots Specifies an array of *knotCount* nondecreasing knot values.

stride Specifies the offset (as a number of single-precision floating-point values) between successive curve control points.

control Specifies a pointer to an array of control points. The coordinates must agree with *type*, specified below.

order Specifies the order of the NURBS curve. *order* equals degree + 1, hence a cubic curve has an order of 4.

type Specifies the type of the curve. If this curve is defined within a **gluBeginCurve**/**gluEndCurve** pair, then the type can be any of the valid one-dimensional evaluator types (such as **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**.

DESCRIPTION

Use **gluNurbsCurve** to describe a NURBS curve.

When **gluNurbsCurve** appears between a **gluBeginCurve**/**gluEndCurve** pair, it is used to describe a curve to be rendered. Positional, texture, and color coordinates are associated by presenting each as a separate **gluNurbsCurve** between a **gluBeginCurve**/**gluEndCurve** pair. No more than one call to **gluNurbsCurve** for each of color, position, and texture data can be made within a single **gluBeginCurve**/**gluEndCurve** pair. Exactly one call must be made to describe the position of the curve (a *type* of **GL_MAP1_VERTEX_3** or **GL_MAP1_VERTEX_4**).

When **gluNurbsCurve** appears between a **gluBeginTrim**/**gluEndTrim** pair, it is used to describe a trimming curve on a NURBS surface. If *type* is **GLU_MAP1_TRIM_2**, then it describes a curve in two-dimensional (*u* and *v*) parameter space. If it is **GLU_MAP1_TRIM_3**, then it describes a curve in two-dimensional homogeneous (*u*, *v*, and *w*) parameter space. See the **gluBeginTrim** reference page for more discussion about trimming curves.

EXAMPLE

The following commands render a textured NURBS curve with normals:

```
gluBeginCurve(nobj);
   gluNurbsCurve(nobj, ..., GL_MAP1_TEXTURE_COORD_2);
   gluNurbsCurve(nobj, ..., GL_MAP1_NORMAL);
   gluNurbsCurve(nobj, ..., GL_MAP1_VERTEX_4);
gluEndCurve(nobj);
```

NOTES

To define trim curves which stitch well, use **gluPwlCurve**.

SEE ALSO

gluBeginCurve, **gluBeginTrim**, **gluNewNurbsRenderer**, **gluPwlCurve**

NAME

gluNurbsProperty - set a NURBS property

C SPECIFICATION

```
void gluNurbsProperty( GLUnurbs* nurb,
                       GLenum property,
                       GLfloat value )
```

PARAMETERS

nurb Specifies the NURBS object (created with **gluNewNurbsRenderer**).

property Specifies the property to be set. Valid values are **GLU_SAMPLING_TOLERANCE**, **GLU_DISPLAY_MODE**, **GLU_CULLING**, **GLU_AUTO_LOAD_MATRIX**, **GLU_PARAMETRIC_TOLERANCE**, **GLU_SAMPLING_METHOD**, **GLU_U_STEP**, **GLU_V_STEP**, or **GLU_NURBS_MODE_EXT**.

value Specifies the value of the indicated property. It may be a numeric value, or one of **GLU_OUTLINE_POLYGON**, **GLU_FILL**, **GLU_OUTLINE_PATCH**, **GL_TRUE**, **GL_FALSE**, **GLU_PATH_LENGTH**, **GLU_PARAMETRIC_ERROR**, **GLU_DOMAIN_DISTANCE**, **GLU_NURBS_RENDERER_EXT**, or **GLU_NURBS_TESSELLATOR_EXT**.

DESCRIPTION

gluNurbsProperty is used to control properties stored in a NURBS object. These properties affect the way that a NURBS curve is rendered. The accepted values for *property* are as follows:

GLU_NURBS_MODE_EXT

value should be set to be either **GLU_NURBS_RENDERER_EXT** or **GLU_NURBS_TESSELLATOR_EXT**. When set to **GLU_NURBS_RENDERER_EXT**, NURBS objects are tessellated into openGL primitives and sent to the pipeline for rendering. When set to **GLU_NURBS_TESSELLATOR_EXT**, NURBS objects are tessellated into openGL primitives but the vertices, normals, colors, and/or textures are retrieved back through a callback interface (see **gluNurbsCallback**). This allows the user to cache the tessellated results for further processing.

GLU_SAMPLING_METHOD

Specifies how a NURBS surface should be tessellated. *value* may be one of **GLU_PATH_LENGTH**, **GLU_PARAMETRIC_ERROR**, **GLU_DOMAIN_DISTANCE**, **GLU_OBJECT_PATH_LENGTH_EXT**, or **GLU_OBJECT_PARAMETRIC_ERROR_EXT**. When set to **GLU_PATH_LENGTH**, the surface is rendered so that the maximum length, in pixels, of the edges of the tessellation polygons is no greater than what is specified by **GLU_SAMPLING_TOLERANCE**.

GLU_PARAMETRIC_ERROR specifies that the surface is rendered in such a way that the value specified by **GLU_PARAMETRIC_TOLERANCE** describes the maximum distance, in pixels, between the tessellation polygons and the surfaces they approximate.

GLU_DOMAIN_DISTANCE allows users to specify, in parametric coordinates, how many sample points per unit length are taken in *u*, *v* direction.

GLU_OBJECT_PATH_LENGTH_EXT is similar to **GLU_PATH_LENGTH** except that it is view independent, that is, the surface is rendered so that the maximum length, in object space, of edges of the tessellation polygons is no greater than what is specified by **GLU_SAMPLING_TOLERANCE**.

GLU_OBJECT_PARAMETRIC_ERROR_EXT is similar to **GLU_PARAMETRIC_ERROR** except that it is view independent, that is, the surface is rendered in such a way that the value specified by **GLU_PARAMETRIC_TOLERANCE** describes the maximum distance, in object space, between the tessellation polygons and the surfaces they approximate.

The initial value of **GLU_SAMPLING_METHOD** is **GLU_PATH_LENGTH**.

GLU_SAMPLING_TOLERANCE

Specifies the maximum length, in pixels or in object space length unit, to use when the sampling method is set to **GLU_PATH_LENGTH** or **GLU_OBJECT_PATH_LENGTH_EXT**. The NURBS code is conservative when rendering a curve or surface, so the actual length can be somewhat shorter. The initial value is 50.0 pixels.

GLU_PARAMETRIC_TOLERANCE

Specifies the maximum distance, in pixels or in object space length unit, to use when the sampling method is **GLU_PARAMETRIC_ERROR** or **GLU_OBJECT_PARAMETRIC_ERROR_EXT**. The initial value is 0.5.

GLU_U_STEP

Specifies the number of sample points per unit length taken along the *u* axis in parametric coordinates. It is needed when **GLU_SAMPLING_METHOD** is set to **GLU_DOMAIN_DISTANCE**. The initial value is 100.

GLU_V_STEP

Specifies the number of sample points per unit length taken along the *v* axis in parametric coordinate. It is needed when **GLU_SAMPLING_METHOD** is set to **GLU_DOMAIN_DISTANCE**. The initial value is 100.

GLU_DISPLAY_MODE

value can be set to **GLU_OUTLINE_POLYGON**, **GLU_FILL**, or **GLU_OUTLINE_PATCH**. When **GLU_NURBS_MODE_EXT** is set to be **GLU_NURBS_RENDERER_EXT**, *value* defines how a NURBS surface should be rendered. When *value* is set to **GLU_FILL**, the surface is rendered as a set of polygons. When *value* is set to **GLU_OUTLINE_POLYGON**, the NURBS library draws only the outlines of the polygons created by tessellation. When *value* is set to **GLU_OUTLINE_PATCH** just the outlines of patches and trim curves defined by the user are drawn.

When **GLU_NURBS_MODE_EXT** is set to be **GLU_NURBS_TESSELLATOR_EXT**, *value* defines how a NURBS surface should be tessellated. When **GLU_DISPLAY_MODE** is set to **GLU_FILL** or **GLU_OUTLINE_POLY**, the NURBS surface is tessellated into openGL triangle primitives which can be retrieved back through callback functions. If **GLU_DISPLAY_MODE** is set to **GLU_OUTLINE_PATCH**, only the outlines of the patches and trim curves are generated as a sequence of line strips which can be retrieved back through callback functions.

The initial value is **GLU_FILL**.

GLU_CULLING *value* is a boolean value that, when set to **GL_TRUE**, indicates that a NURBS curve should be discarded prior to tessellation if its control points lie outside the current viewport. The initial value is **GL_FALSE**.

GLU_AUTO_LOAD_MATRIX

value is a boolean value. When set to **GL_TRUE**, the NURBS code downloads the projection matrix, the modelview matrix, and the viewport from the GL server to compute sampling and culling matrices for each NURBS curve that is rendered. Sampling and culling matrices are required to determine the tessellation of a NURBS surface into line segments or polygons and to cull a NURBS surface if it lies outside the viewport.

If this mode is set to **GL_FALSE**, then the program needs to provide a projection matrix, a modelview matrix, and a viewport for the NURBS renderer to use to construct sampling and culling matrices. This can be done with the **gluLoadSamplingMatrices** function. This mode is initially set to **GL_TRUE**. Changing it from **GL_TRUE** to **GL_FALSE** does not affect the sampling and culling matrices until **gluLoadSamplingMatrices** is called.

NOTES

If **GLU_AUTO_LOAD_MATRIX** is true, sampling and culling may be executed incorrectly if NURBS routines are compiled into a display list.

A *property* of **GLU_PARAMETRIC_TOLERANCE**, **GLU_SAMPLING_METHOD**, **GLU_U_STEP**, or **GLU_V_STEP**, or a *value* of **GLU_PATH_LENGTH**, **GLU_PARAMETRIC_ERROR**, **GLU_DOMAIN_DISTANCE** are only available if the GLU version is 1.1 or greater. They are not valid parameters in GLU 1.0.

gluGetString can be used to determine the GLU version.

SEE ALSO

gluGetNurbsProperty, **gluLoadSamplingMatrices**, **gluNewNurbsRenderer**, **gluGetString**, **gluNurbsCallback**

NAME

gluNurbsSurface - define the shape of a NURBS surface

C SPECIFICATION

```
void gluNurbsSurface( GLUnurbs* nurb,
                      GLint sKnotCount,
                      GLfloat* sKnots,
                      GLint tKnotCount,
                      GLfloat* tKnots,
                      GLint sStride,
                      GLint tStride,
                      GLfloat* control,
                      GLint sOrder,
                      GLint tOrder,
                      GLenum type )
```

PARAMETERS

nurb Specifies the NURBS object (created with **gluNewNurbsRenderer**).

sKnotCount Specifies the number of knots in the parametric *u* direction.

sKnots Specifies an array of *sKnotCount* nondecreasing knot values in the parametric *u* direction.

tKnotCount Specifies the number of knots in the parametric *v* direction.

tKnots Specifies an array of *tKnotCount* nondecreasing knot values in the parametric *v* direction.

sStride Specifies the offset (as a number of single-precision floating point values) between successive control points in the parametric *u* direction in *control*.

tStride Specifies the offset (in single-precision floating-point values) between successive control points in the parametric *v* direction in *control*.

control Specifies an array containing control points for the NURBS surface. The offsets between successive control points in the parametric *u* and *v* directions are given by *sStride* and *tStride*.

sOrder Specifies the order of the NURBS surface in the parametric *u* direction. The order is one more than the degree, hence a surface that is cubic in *u* has a *u* order of 4.

tOrder Specifies the order of the NURBS surface in the parametric *v* direction. The order is one more than the degree, hence a surface that is cubic in *v* has a *v* order of 4.

type Specifies type of the surface. *type* can be any of the valid two-dimensional evaluator types (such as **GL_MAP2_VERTEX_3** or **GL_MAP2_COLOR_4**).

DESCRIPTION

Use **gluNurbsSurface** within a NURBS (Non-Uniform Rational B-Spline) surface definition to describe the shape of a NURBS surface (before any trimming). To mark the beginning of a NURBS surface definition, use the **gluBeginSurface** command. To mark the end of a NURBS surface definition, use the **gluEndSurface** command. Call **gluNurbsSurface** within a NURBS surface definition only.

Positional, texture, and color coordinates are associated with a surface by presenting each as a separate **gluNurbsSurface** between a **gluBeginSurface**/**gluEndSurface** pair. No more than one call to **gluNurbsSurface** for each of color, position, and texture data can be made within a single **gluBeginSurface**/**gluEndSurface** pair. Exactly one call must be made to describe the position of the surface (a *type* of **GL_MAP2_VERTEX_3** or **GL_MAP2_VERTEX_4**).

A NURBS surface can be trimmed by using the commands **gluNurbsCurve** and **gluPwlCurve** between calls to **gluBeginTrim** and **gluEndTrim**.

Note that a **gluNurbsSurface** with *sKnotCount* knots in the *u* direction and *tKnotCount* knots in the *v* direction with orders *sOrder* and *tOrder* must have (*sKnotCount* - *sOrder*) x (*tKnotCount* - *tOrder*) control points.

EXAMPLE

The following commands render a textured NURBS surface with normals; the texture coordinates and normals are also NURBS surfaces:

```
gluBeginSurface(nobj);
   gluNurbsSurface(nobj, ..., GL_MAP2_TEXTURE_COORD_2);
   gluNurbsSurface(nobj, ..., GL_MAP2_NORMAL);
   gluNurbsSurface(nobj, ..., GL_MAP2_VERTEX_4);
gluEndSurface(nobj);
```

SEE ALSO

gluBeginSurface, **gluBeginTrim**, **gluNewNurbsRenderer**, **gluNurbsCurve**, **gluPwlCurve**

NAME

gluOrtho2D - define a 2D orthographic projection matrix

C SPECIFICATION

```
void gluOrtho2D( GLdouble left,
                 GLdouble right,
                 GLdouble bottom,
                 GLdouble top )
```

PARAMETERS

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

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

DESCRIPTION

gluOrtho2D sets up a two-dimensional orthographic viewing region. This is equivalent to calling **glOrtho** with near=-1 and far=1.

SEE ALSO

glOrtho, **gluPerspective**

NAME

gluPartialDisk - draw an arc of a disk

C SPECIFICATION

```
void gluPartialDisk( GLUquadric* quad,
                     GLdouble inner,
                     GLdouble outer,
                     GLint slices,
                     GLint loops,
                     GLdouble start,
                     GLdouble sweep )
```

PARAMETERS

quad Specifies a quadrics object (created with **gluNewQuadric**).

inner Specifies the inner radius of the partial disk (can be 0).

outer Specifies the outer radius of the partial disk.

slices Specifies the number of subdivisions around the *z* axis.

loops Specifies the number of concentric rings about the origin into which the partial disk is subdivided.

start Specifies the starting angle, in degrees, of the disk portion.

sweep Specifies the sweep angle, in degrees, of the disk portion.

DESCRIPTION

gluPartialDisk renders a partial disk on the z=0 plane. A partial disk is similar to a full disk, except that only the subset of the disk from *start* through *start* + *sweep* is included (where 0 degrees is along the +*y* axis, 90 degrees along the +*x* axis, 180 along the -*y* axis, and 270 along the -*x* axis).

The partial disk has a radius of *outer*, and contains a concentric circular hole with a radius of *inner*. If *inner* is

0, then no hole is generated. The partial disk is subdivided around the *z* axis into slices (like pizza slices), and also about the *z* axis into rings (as specified by *slices* and *loops*, respectively).

With respect to orientation, the +*z* side of the partial disk is considered to be outside (see **gluQuadricOrientation**). This means that if the orientation is set to **GLU_OUTSIDE**, then any normals generated point along the +*z* axis. Otherwise, they point along the -*z* axis.

If texturing is turned on (with **gluQuadricTexture**), texture coordinates are generated linearly such that where r=outer, the value at (*r*, 0, 0) is (1.0, 0.5), at (0, *r*, 0) it is (0.5, 1.0), at (-*r*, 0, 0) it is (0.0, 0.5), and at (0, -*r*, 0) it is (0.5, 0.0).

SEE ALSO

gluCylinder, **gluDisk**, **gluNewQuadric**, **gluQuadricOrientation**, **gluQuadricTexture**, **gluSphere**

NAME

gluPerspective - set up a perspective projection matrix

C SPECIFICATION

```
void gluPerspective( GLdouble fovy,
                     GLdouble aspect,
                     GLdouble zNear,
                     GLdouble zFar )
```

PARAMETERS

fovy Specifies the field of view angle, in degrees, in the *y* direction.

aspect Specifies the aspect ratio that determines the field of view in the *x* direction. The aspect ratio is the ratio of *x* (width) to *y* (height).

zNear Specifies the distance from the viewer to the near clipping plane (always positive).

zFar Specifies the distance from the viewer to the far clipping plane (always positive).

DESCRIPTION

gluPerspective specifies a viewing frustum into the world coordinate system. In general, the aspect ratio in **gluPerspective** should match the aspect ratio of the associated viewport. For example, aspect=2.0 means the viewer's angle of view is twice as wide in *x* as it is in *y*. If the viewport is twice as wide as it is tall, it displays the image without distortion.

The matrix generated by **gluPerspective** is multipled by the current matrix, just as if **glMultMatrix** were called with the generated matrix. To load the perspective matrix onto the current matrix stack instead, precede the call to **gluPerspective** with a call to **glLoadIdentity**.

Given *f* defined as follows:

$$f = \text{cotangent}\left(\frac{\quad}{2}\right)$$

The generated matrix is

```
(                                        )
|  ______                                |
|  aspect  0      0            0         |
|                                        |
|    0     f      0            0         |
|             __________   ____________   |
|    0     0  zNear-zFar   zNear-zFar    |
(                                        )
     0     0      -1           0
```

NOTES

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

```
r = _____
    zNear
```

roughly log r bits of depth buffer precision are lost. Because r a**p**proaches infinity as *zNear* approaches 0, *zNear* must never be set to 0.

SEE ALSO

glFrustum, **glLoadIdentity**, **glMultMatrix**, **gluOrtho2D**

NAME

gluPickMatrix - define a picking region

C SPECIFICATION

```
void gluPickMatrix( GLdouble x,
                    GLdouble y,
                    GLdouble delX,
                    GLdouble delY,
                    GLint *viewport )
```

PARAMETERS

x, *y* Specify the center of a picking region in window coordinates.

delX, *delY*

Specify the width and height, respectively, of the picking region in window coordinates.

viewport

Specifies the current viewport (as from a **glGetIntegerv** call).

DESCRIPTION

gluPickMatrix creates a projection matrix that can be used to restrict drawing to a small region of the viewport. This is typically useful to determine what objects are being drawn near the cursor. Use **gluPickMatrix** to restrict drawing to a small region around the cursor. Then, enter selection mode (with **glRenderMode**) and rerender the scene. All primitives that would have been drawn near the cursor are identified and stored in the selection buffer.

The matrix created by **gluPickMatrix** is multiplied by the current matrix just as if **glMultMatrix** is called with the generated matrix. To effectively use the generated pick matrix for picking, first call **glLoadIdentity** to load an identity matrix onto the perspective matrix stack. Then call **gluPickMatrix**, and finally, call a command (such as **gluPerspective**) to multiply the perspective matrix by the pick matrix.

When using **gluPickMatrix** to pick NURBS, be careful to turn off the NURBS property **GLU_AUTO_LOAD_MATRIX**. If

GLU_AUTO_LOAD_MATRIX is not turned off, then any NURBS surface rendered is subdivided differently with the pick matrix than the way it was subdivided without the pick matrix.

EXAMPLE

When rendering a scene as follows:

```
glMatrixMode(GL_PROJECTION); glLoadIdentity();
gluPerspective(...); glMatrixMode(GL_MODELVIEW); /* Draw the
scene */
```

a portion of the viewport can be selected as a pick region like this:

```
glMatrixMode(GL_PROJECTION); glLoadIdentity();
gluPickMatrix(x, y, width, height, viewport);
gluPerspective(...); glMatrixMode(GL_MODELVIEW); /* Draw the
scene */
```

SEE ALSO

glGet, **glLoadIndentity**, **glMultMatrix**, **glRenderMode**, **gluPerspective**

NAME

gluProject - map object coordinates to window coordinates

C SPECIFICATION

```
GLint gluProject( GLdouble objX,
                  GLdouble objY,
                  GLdouble objZ,
                  const GLdouble *model,
                  const GLdouble *proj,
                  const GLint *view,
                  GLdouble* winX,
                  GLdouble* winY,
                  GLdouble* winZ )
```

PARAMETERS

objX, *objY*, *objZ*
Specify the object coordinates.

model Specifies the current modelview matrix (as from a **glGetDoublev** call).

proj Specifies the current projection matrix (as from a **glGetDoublev** call).

view Specifies the current viewport (as from a **glGetIntegerv** call).

winX, *winY*, *winZ*
Return the computed window coordinates.

DESCRIPTION

gluProject transforms the specified object coordinates into window coordinates using *model*, *proj*, and *view*. The result is stored in *winX*, *winY*, and *winZ*. A return value of **GL_TRUE** indicates success, a return value of **GL_FALSE** indicates failure.

To compute the coordinates, let v=(objX,objY,objZ,1.0) represented as a matrix with 4 rows and 1 column. Then **gluProject** computes v' as follows:

v' = P x M x v

where P is the current projection matrix *proj*, M is the current modelview matrix *model* (both represented as 4x4 matrices in column-major order) and 'x' represents matrix multiplication.

The window coordinates are then computed as follows:

```
winX = view(0) + view(2) * (v'(0) + 1) / 2

winY = view(1) + view(3) * (v'(1) + 1) / 2

winZ = (v'(2) + 1) / 2
```

SEE ALSO

glGet, **gluUnProject**

NAME

gluPwlCurve - describe a piecewise linear NURBS trimming curve

C SPECIFICATION

```
void gluPwlCurve( GLUnurbs* nurb,
                  GLint count,
                  GLfloat* data,
                  GLint stride,
                  GLenum type )
```

PARAMETERS

nurb Specifies the NURBS object (created with **gluNewNurbsRenderer**).

count Specifies the number of points on the curve.

data Specifies an array containing the curve points.

stride Specifies the offset (a number of single-precision floating-point values) between points on the curve.

type Specifies the type of curve. Must be either **GLU_MAP1_TRIM_2** or **GLU_MAP1_TRIM_3**.

DESCRIPTION

gluPwlCurve describes a piecewise linear trimming curve for a NURBS surface. A piecewise linear curve consists of a list of coordinates of points in the parameter space for the NURBS surface to be trimmed. These points are connected with line segments to form a curve. If the curve is an approximation to a curve that is not piecewise linear, the points should be close enough in parameter space that the resulting path appears curved at the resolution used in the application.

If *type* is **GLU_MAP1_TRIM_2**, then it describes a curve in two-dimensional (*u* and *v*) parameter space. If it is **GLU_MAP1_TRIM_3**, then it describes a curve in two-dimensional homogeneous (*u*, *v*, and *w*) parameter space. See the **gluBeginTrim** reference page for more information about trimming curves.

NOTES

To describe a trim curve that closely follows the contours of a NURBS surface, call **gluNurbsCurve**.

SEE ALSO

gluBeginCurve, **gluBeginTrim**, **gluNewNurbsRenderer**, **gluNurbsCurve**

NAME

gluQuadricCallback - define a callback for a quadrics object

C SPECIFICATION

```
void gluQuadricCallback( GLUquadric* quad,
                         GLenum which,
                         GLvoid (*CallBackFunc)( )
```

PARAMETERS

quad Specifies the quadrics object (created with **gluNewQuadric**).

which Specifies the callback being defined. The only valid value is **GLU_ERROR**.

CallBackFunc Specifies the function to be called.

DESCRIPTION

gluQuadricCallback is used to define a new callback to be used by a quadrics object. If the specified callback is already defined, then it is replaced. If *CallBackFunc* is NULL, then any existing callback is erased.

The one legal callback is **GLU_ERROR**:

GLU_ERROR The function is called when an error is encountered. Its single argument is of type GLenum, and it indicates the specific error that occurred. Character strings describing these errors can be retrieved with the **gluErrorString** call.

SEE ALSO

gluErrorString, **gluNewQuadric**

NAME

gluQuadricDrawStyle - specify the draw style desired for quadrics

C SPECIFICATION

```
void gluQuadricDrawStyle( GLUquadric* quad,
                          GLenum draw )
```

PARAMETERS

quad Specifies the quadrics object (created with **gluNewQuadric**).

draw Specifies the desired draw style. Valid values are **GLU_FILL**, **GLU_LINE**, **GLU_SILHOUETTE**, and **GLU_POINT**.

DESCRIPTION

gluQuadricDrawStyle specifies the draw style for quadrics rendered with *quad*. The legal values are as follows:

GLU_FILL Quadrics are rendered with polygon primitives. The polygons are drawn in a counterclockwise fashion with respect to their normals (as defined with **gluQuadricOrientation**).

GLU_LINE Quadrics are rendered as a set of lines.

GLU_SILHOUETTE Quadrics are rendered as a set of lines, except that edges separating coplanar faces will not be drawn.

GLU_POINT Quadrics are rendered as a set of points.

SEE ALSO

gluNewQuadric, **gluQuadricNormals**, **gluQuadricOrientation**, **gluQuadricTexture**

NAME

gluQuadricNormals - specify what kind of normals are desired for quadrics

C SPECIFICATION

```
void gluQuadricNormals( GLUquadric* quad,
                        GLenum normal )
```

PARAMETERS

quad Specifes the quadrics object (created with **gluNewQuadric**).

normal Specifies the desired type of normals. Valid values are **GLU_NONE**, **GLU_FLAT**, and **GLU_SMOOTH**.

DESCRIPTION

gluQuadricNormals specifies what kind of normals are desired for quadrics rendered with *quad*. The legal values are as follows:

GLU_NONE	No normals are generated.
GLU_FLAT	One normal is generated for every facet of a quadric.
GLU_SMOOTH	One normal is generated for every vertex of a quadric. This is the initial value.

SEE ALSO

gluNewQuadric, **gluQuadricDrawStyle**, **gluQuadricOrientation**, **gluQuadricTexture**

NAME

gluQuadricOrientation - specify inside/outside orientation for quadrics

C SPECIFICATION

```
void gluQuadricOrientation( GLUquadric* quad,
                            GLenum orientation )
```

PARAMETERS

quad Specifies the quadrics object (created with **gluNewQuadric**).

orientation Specifies the desired orientation. Valid values are **GLU_OUTSIDE** and **GLU_INSIDE**.

DESCRIPTION

gluQuadricOrientation specifies what kind of orientation is desired for quadrics rendered with *quad*. The *orientation* values are as follows:

GLU_OUTSIDE Quadrics are drawn with normals pointing outward (the initial value).

GLU_INSIDE Quadrics are drawn with normals pointing inward.

Note that the interpretation of *outward* and *inward* depends on the quadric being drawn.

SEE ALSO

gluNewQuadric, **gluQuadricDrawStyle**, **gluQuadricNormals**, **gluQuadricTexture**

NAME

gluQuadricTexture - specify if texturing is desired for quadrics

C SPECIFICATION

```
void gluQuadricTexture( GLUquadric* quad,
                        GLboolean texture )
```

PARAMETERS

quad Specifies the quadrics object (created with **gluNewQuadric**).

texture Specifies a flag indicating if texture coordinates should be generated.

DESCRIPTION

gluQuadricTexture specifies if texture coordinates should be generated for quadrics rendered with *quad*. If the value of *texture* is **GL_TRUE**, then texture coordinates are generated, and if *texture* is **GL_FALSE**, they are not. The initial value is **GL_FALSE**.

The manner in which texture coordinates are generated depends upon the specific quadric rendered.

SEE ALSO

gluNewQuadric, **gluQuadricDrawStyle**, **gluQuadricNormals**, **gluQuadricOrientation**

NAME

gluScaleImage - scale an image to an arbitrary size

C SPECIFICATION

```
GLint gluScaleImage( GLenum format,
                     GLsizei wIn,
                     GLsizei hIn,
                     GLenum typeIn,
                     const void *dataIn,
                     GLsizei wOut,
                     GLsizei hOut,
                     GLenum typeOut,
                     GLvoid* dataOut )
```

PARAMETERS

format Specifies the format of the pixel data. The following symbolic values are valid: **GL_COLOR_INDEX**, **GL_STENCIL_INDEX**, **GL_DEPTH_COMPONENT**, **GL_RED**, **GL_GREEN**, **GL_BLUE**, **GL_ALPHA**, **GL_RGB**, **GL_RGBA**, **GL_LUMINANCE**, and **GL_LUMINANCE_ALPHA.**

wIn, *hIn* Specify the width and height, respectively, of the source image that is scaled.

typeIn Specifies the data type for *dataIn*. Must be one of **GL_UNSIGNED_BYTE**, **GL_BYTE**, **GL_BITMAP**, **GL_UNSIGNED_SHORT**, **GL_SHORT**, **GL_UNSIGNED_INT**, **GL_INT**, or **GL_FLOAT**.

dataIn Specifies a pointer to the source image.

wOut, *hOut*

Specify the width and height, respectively, of the destination image.

typeOut Specifies the data type for *dataOut*. Must be one of **GL_UNSIGNED_BYTE**, **GL_BYTE**, **GL_BITMAP**, **GL_UNSIGNED_SHORT**, **GL_SHORT**, **GL_UNSIGNED_INT**, **GL_INT**, or **GL_FLOAT**.

dataOut Specifies a pointer to the destination image.

DESCRIPTION

gluScaleImage scales a pixel image using the appropriate pixel store modes to unpack data from the source image and pack data into the destination image.

When shrinking an image, **gluScaleImage** uses a box filter to sample the source image and create pixels for the destination image. When magnifying an image, the pixels from the source image are linearly interpolated to create the destination image.

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

See the **glReadPixels** reference page for a description of the acceptable values for *format*, *typeIn*, and *typeOut*.

ERRORS

GLU_INVALID_VALUE is returned if *wIn*, *hIn*, *wOut*, or *hOut* are < 0.

GLU_INVALID_ENUM is returned if *format*, *typeIn*, or *typeOut* are not legal.

SEE ALSO

glDrawPixels, **glReadPixels**, **gluBuild1DMipmaps**, **gluBuild2DMipmaps**, **gluErrorString**

NAME

gluSphere - draw a sphere

C SPECIFICATION

```
void gluSphere( GLUquadric* quad,
                GLdouble radius,
                GLint slices,
                GLint stacks )
```

PARAMETERS

quad Specifies the quadrics object (created with **gluNewQuadric**).

radius Specifies the radius of the sphere.

slices Specifies the number of subdivisions around the *z* axis (similar to lines of longitude).

stacks Specifies the number of subdivisions along the *z* axis (similar to lines of latitude).

DESCRIPTION

gluSphere draws a sphere of the given radius centered around the origin. The sphere is subdivided around the *z* axis into slices and along the *z* axis into stacks (similar to lines of longitude and latitude).

If the orientation is set to **GLU_OUTSIDE** (with **gluQuadricOrientation**), then any normals generated point away from the center of the sphere. Otherwise, they point toward the center of the sphere.

If texturing is turned on (with **gluQuadricTexture**), then texture coordinates are generated so that *t* ranges from 0.0 at *z*=-radius to 1.0 at *z*=radius (*t* increases linearly along longitudinal lines), and *s* ranges from 0.0 at the +*y* axis, to 0.25 at the +*x* axis, to 0.5 at the -*y* axis, to 0.75 at the -*x* axis, and back to 1.0 at the +*y* axis.

SEE ALSO

gluCylinder, **gluDisk**, **gluNewQuadric**, **gluPartialDisk**, **gluQuadricOrientation**, **gluQuadricTexture**

NAME

gluTessBeginContour, gluTessEndContour - delimit a contour description

C SPECIFICATION

void **gluTessBeginContour**(GLUtesselator* *tess*)

void **gluTessEndContour**(GLUtesselator* *tess*)

PARAMETERS

tess Specifies the tessellation object (created with **gluNewTess**).

DESCRIPTION

gluTessBeginContour and **gluTessEndContour** delimit the definition of a polygon contour. Within each **gluTessBeginContour/gluTessEndContour** pair, there can be zero or more calls to **gluTessVertex**. The vertices specify a closed contour (the last vertex of each contour is automatically linked to the first). See the **gluTessVertex** reference page for more details. **gluTessBeginContour** can only be called between **gluTessBeginPolygon** and **gluTessEndPolygon**.

SEE ALSO

gluNewTess, **gluTessBeginPolygon**, **gluTessVertex**, **gluTessCallback**, **gluTessProperty**, **gluTessNormal**, **gluTessEndPolygon**

NAME

gluTessBeginPolygon - delimit a polygon description

C SPECIFICATION

```
void gluTessBeginPolygon( GLUtesselator* tess,
                          GLvoid* data )
```

PARAMETERS

tess Specifies the tessellation object (created with **gluNewTess**).

data Specifies a pointer to user polygon data.

DESCRIPTION

gluTessBeginPolygon and **gluTessEndPolygon** delimit the definition of a convex, concave or self-intersecting polygon. Within each **gluTessBeginPolygon**/**gluTessEndPolygon** pair, there must be one or more calls to **gluTessBeginContour**/**gluTessEndContour**. Within each contour, there are zero or more calls to **gluTessVertex**. The vertices specify a closed contour (the last vertex of each contour is automatically linked to the first). See the **gluTessVertex**, **gluTessBeginContour**, and **gluTessEndContour** reference pages for more details.

data is a pointer to a user-defined data structure. If the appropriate callback(s) are specified (see **gluTessCallback**), then this pointer is returned to the callback function(s). Thus, it is a convenient way to store per-polygon information.

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

EXAMPLE

A quadrilateral with a triangular hole in it can be described as follows:

```
gluTessBeginPolygon(tobj, NULL);
 gluTessBeginContour(tobj);
   gluTessVertex(tobj, v1, v1);
```

```
    gluTessVertex(tobj, v2, v2);
    gluTessVertex(tobj, v3, v3);
    gluTessVertex(tobj, v4, v4);
  gluTessEndContour(tobj);
  gluTessBeginContour(tobj);
    gluTessVertex(tobj, v5, v5);
    gluTessVertex(tobj, v6, v6);
    gluTessVertex(tobj, v7, v7);
  gluTessEndContour(tobj); gluTessEndPolygon(tobj);
```

SEE ALSO

gluNewTess, **gluTessBeginContour**, **gluTessVertex**, **gluTessCallback**, **gluTessProperty**, **gluTessNormal**, **gluTessEndPolygon**

NAME

gluTessCallback - define a callback for a tessellation object

C SPECIFICATION

```
void gluTessCallback( GLUtesselator* tess,
                      GLenum which,
                      GLvoid (*CallBackFunc)( )
```

PARAMETERS

tess Specifies the tessellation object (created with **gluNewTess**).

which Specifies the callback being defined. The following values are valid: **GLU_TESS_BEGIN**, **GLU_TESS_BEGIN_DATA**, **GLU_TESS_EDGE_FLAG**, **GLU_TESS_EDGE_FLAG_DATA**, **GLU_TESS_VERTEX**, **GLU_TESS_VERTEX_DATA**, **GLU_TESS_END**, **GLU_TESS_END_DATA**, **GLU_TESS_COMBINE**, **GLU_TESS_COMBINE_DATA**, **GLU_TESS_ERROR**, and **GLU_TESS_ERROR_DATA.**

CallBackFunc Specifies the function to be called.

DESCRIPTION

gluTessCallback is used to indicate a callback to be used by a tessellation object. If the specified callback is already defined, then it is replaced. If *CallBackFunc* is NULL, then the existing callback becomes undefined.

These callbacks are used by the tessellation object to describe how a polygon specified by the user is broken into triangles. Note that there are two versions of each callback: one with user-specified polygon data and one without. If both versions of a particular callback are specified, then the callback with user-specified polygon data will be used. Note that the *polygon_data* parameter used by some of the functions is a copy of the pointer that was specified when **gluTessBeginPolygon** was called. The legal callbacks are as follows:

GLU_TESS_BEGIN

The begin callback is invoked like **glBegin** to indicate the start of a (triangle) primitive. The function takes a single argument of type GLenum. If the **GLU_TESS_BOUNDARY_ONLY** property is set to **GL_FALSE**, then the argument is set to either **GL_TRIANGLE_FAN**, **GL_TRIANGLE_STRIP**, or **GL_TRIANGLES**. If the **GLU_TESS_BOUNDARY_ONLY** property is set to **GL_TRUE**, then the argument will be set to **GL_LINE_LOOP**. The function prototype for this callback is:

```
void begin ( GLenum type );
```

GLU_TESS_BEGIN_DATA

The same as the **GLU_TESS_BEGIN** callback except that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when **gluTessBeginPolygon** was called. The function prototype for this callback is:

```
void beginData ( GLenum type, void *polygon_data );
```

GLU_TESS_EDGE_FLAG

The edge flag callback is similar to **glEdgeFlag**. The function takes a single boolean flag that indicates which edges lie on the polygon boundary. If the flag is **GL_TRUE**, then each vertex that follows begins an edge that lies on the polygon boundary, that is, an edge that separates an interior region from an exterior one. If the flag is **GL_FALSE**, then each vertex that follows begins an edge that lies in the polygon interior. The edge flag callback (if defined) is invoked before the first vertex callback.

Since triangle fans and triangle strips do not support edge flags, the begin callback is not called with **GL_TRIANGLE_FAN** or **GL_TRIANGLE_STRIP** if a non-NULL edge flag callback is provided. (If the callback is initialized to NULL, there is no impact on performance). Instead, the fans and strips are converted to independent triangles. The function prototype for this callback is:

```
void edgeFlag ( GLboolean flag );
```

GLU_TESS_EDGE_FLAG_DATA

The same as the **GLU_TESS_EDGE_FLAG** callback except that it takes an additional pointer argument. This

pointer is identical to the opaque pointer provided when **gluTessBeginPolygon** was called. The function prototype for this callback is:
void edgeFlagData (GLboolean flag, void *polygon_data);

GLU_TESS_VERTEX

The vertex callback is invoked between the begin and end callbacks. It is similar to **glVertex**, and it defines the vertices of the triangles created by the tessellation process. The function takes a pointer as its only argument. This pointer is identical to the opaque pointer provided by the user when the vertex was described (see **gluTessVertex**). The function prototype for this callback is:
void vertex (void *vertex_data);

GLU_TESS_VERTEX_DATA

The same as the **GLU_TESS_VERTEX** callback except that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when **gluTessBeginPolygon** was called. The function prototype for this callback is:
void vertexData (void *vertex_data, void *polygon_data);

GLU_TESS_END

The end callback serves the same purpose as **glEnd**. It indicates the end of a primitive and it takes no arguments. The function prototype for this callback is:
void end (void);

GLU_TESS_END_DATA

The same as the **GLU_TESS_END** callback except that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when **gluTessBeginPolygon** was called. The function prototype for this callback is:
void endData (void *polygon_data);

GLU_TESS_COMBINE

The combine callback is called to create a new vertex when the tessellation detects an intersection, or wishes to merge features. The function takes four arguments: an array of three elements each of type GLdouble, an array of four pointers, an array of four elements each of type GLfloat, and a pointer to a pointer. The prototype is:

```
void combine( GLdouble coords[3], void
*vertex_data[4],
              GLfloat weight[4], void **outData );
```

The vertex is defined as a linear combination of up to four existing vertices, stored in *vertex_data*. The coefficients of the linear combination are given by *weight*; these weights always add up to 1. All vertex pointers are valid even when some of the weights are 0. *coords* gives the location of the new vertex.

The user must allocate another vertex, interpolate parameters using *vertex_data* and *weight*, and return the new vertex pointer in *outData*. This handle is supplied during rendering callbacks. The user is responsible for freeing the memory some time after **gluTessEndPolygon** is called.

For example, if the polygon lies in an arbitrary plane in 3-space, and a color is associated with each vertex, the **GLU_TESS_COMBINE** callback might look like this:

```
void myCombine( GLdouble coords[3], VERTEX *d[4],
                GLfloat w[4], VERTEX **dataOut ) {
   VERTEX *new = new_vertex();

   new->x = coords[0];
   new->y = coords[1];
   new->z = coords[2];
   new->r = w[0]*d[0]->r + w[1]*d[1]->r +
w[2]*d[2]->r + w[3]*d[3]->r;
   new->g = w[0]*d[0]->g + w[1]*d[1]->g +
w[2]*d[2]->g + w[3]*d[3]->g;
   new->b = w[0]*d[0]->b + w[1]*d[1]->b +
w[2]*d[2]->b + w[3]*d[3]->b;
   new->a = w[0]*d[0]->a + w[1]*d[1]->a +
w[2]*d[2]->a + w[3]*d[3]->a;
   *dataOut = new; }
```

If the tessellation detects an intersection, then the **GLU_TESS_COMBINE** or **GLU_TESS_COMBINE_DATA** callback (see below) must be defined, and it must write a non-NULL pointer into *dataOut*. Otherwise the **GLU_TESS_NEED_COMBINE_CALLBACK** error occurs, and no output is generated.

GLU_TESS_COMBINE_DATA

The same as the **GLU_TESS_COMBINE** callback except that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when **gluTessBeginPolygon** was called. The function prototype for this callback is:

```
void combineData ( GLdouble coords[3], void
*vertex_data[4],
                   GLfloat weight[4], void
**outData,
                   void *polygon_data );
```

GLU_TESS_ERROR

The error callback is called when an error is encountered. The one argument is of type GLenum; it indicates the specific error that occurred and will be set to one of **GLU_TESS_MISSING_BEGIN_POLYGON**, **GLU_TESS_MISSING_END_POLYGON**, **GLU_TESS_MISSING_BEGIN_CONTOUR**, **GLU_TESS_MISSING_END_CONTOUR**, **GLU_TESS_COORD_TOO_LARGE**, **GLU_TESS_NEED_COMBINE_CALLBACK** or **GLU_OUT_OF_MEMORY**. Character strings describing these errors can be retrieved with the **gluErrorString** call. The function prototype for this callback is:

```
void error ( GLenum errno );
```

The GLU library will recover from the first four errors by inserting the missing call(s). **GLU_TESS_COORD_TOO_LARGE** indicates that some vertex coordinate exceeded the predefined constant **GLU_TESS_MAX_COORD** in absolute value, and that the value has been clamped. (Coordinate values must be small enough so that two can be multiplied together without overflow.) **GLU_TESS_NEED_COMBINE_CALLBACK** indicates that the tessellation detected an intersection between two edges in the input data, and the **GLU_TESS_COMBINE** or **GLU_TESS_COMBINE_DATA** callback was not provided. No output is generated. **GLU_OUT_OF_MEMORY** indicates that there is not enough memory so no output is generated.

GLU_TESS_ERROR_DATA

The same as the **GLU_TESS_ERROR** callback except that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when **gluTessBeginPolygon** was called. The function prototype for this callback is: void errorData (GLenum errno, void *polygon_data);

EXAMPLE

Polygons tessellated can be rendered directly like this:

```
gluTessCallback(tobj, GLU_TESS_BEGIN, glBegin);
gluTessCallback(tobj, GLU_TESS_VERTEX, glVertex3dv);
gluTessCallback(tobj, GLU_TESS_END, glEnd);
gluTessCallback(tobj, GLU_TESS_COMBINE, myCombine);
gluTessBeginPolygon(tobj, NULL);
  gluTessBeginContour(tobj);
    gluTessVertex(tobj, v, v);
    ...
  gluTessEndContour(tobj); gluTessEndPolygon(tobj);
```

Typically, the tessellated polygon should be stored in a display list so that it does not need to be retessellated every time it is rendered.

SEE ALSO

glBegin, **glEdgeFlag**, **glVertex**, **gluNewTess**, **gluErrorString**, **gluTessVertex**, **gluTessBeginPolygon**, **gluTessBeginContour**, **gluTessProperty**, **gluTessNormal**

NAME

gluTessEndPolygon - delimit a polygon description

C SPECIFICATION

void **gluTessEndPolygon**(GLUtesselator* *tess*)

PARAMETERS

tess Specifies the tessellation object (created with **gluNewTess**).

DESCRIPTION

gluTessBeginPolygon and **gluTessEndPolygon** delimit the definition of a convex, concave or self-intersecting polygon. Within each **gluTessBeginPolygon**/**gluTessEndPolygon** pair, there must be one or more calls to **gluTessBeginContour**/**gluTessEndContour**. Within each contour, there are zero or more calls to **gluTessVertex**. The vertices specify a closed contour (the last vertex of each contour is automatically linked to the first). See the **gluTessVertex**, **gluTessBeginContour** and **gluTessEndContour** reference pages for more details.

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

EXAMPLE

A quadrilateral with a triangular hole in it can be described like this:

```
gluTessBeginPolygon(tobj, NULL);
 gluTessBeginContour(tobj);
   gluTessVertex(tobj, v1, v1);
   gluTessVertex(tobj, v2, v2);
   gluTessVertex(tobj, v3, v3);
   gluTessVertex(tobj, v4, v4);
 gluTessEndContour(tobj);
 gluTessBeginContour(tobj);
   gluTessVertex(tobj, v5, v5);
   gluTessVertex(tobj, v6, v6);
   gluTessVertex(tobj, v7, v7);
 gluTessEndContour(tobj); gluTessEndPolygon(tobj);
```

In the above example the pointers, v1 through v7, should point to different addresses, since the values stored at these addresses will not be read by the tesselator until **gluTessEndPolygon** is called.

SEE ALSO

gluNewTess, **gluTessBeginContour**, **gluTessVertex**, **gluTessCallback**, **gluTessProperty**, **gluTessNormal**, **gluTessBeginPolygon**

NAME

gluTessNormal - specify a normal for a polygon

C SPECIFICATION

```
void gluTessNormal( GLUtesselator* tess,
                    GLdouble valueX,
                    GLdouble valueY,
                    GLdouble valueZ )
```

PARAMETERS

tess Specifies the tessellation object (created with **gluNewTess**).

valueX Specifies the first component of the normal.

valueY Specifies the second component of the normal.

valueZ Specifies the third component of the normal.

DESCRIPTION

gluTessNormal describes a normal for a polygon that the program is defining. All input data will be projected onto a plane perpendicular to one of the three coordinate axes before tessellation and all output triangles will be oriented CCW with respect to the normal (CW orientation can be obtained by reversing the sign of the supplied normal). For example, if you know that all polygons lie in the x-y plane, call **gluTessNormal**(tess, 0.0, 0.0, 1.0) before rendering any polygons.

If the supplied normal is (0.0, 0.0, 0.0) (the initial value), the normal is determined as follows. The direction of the normal, up to its sign, is found by fitting a plane to the vertices, without regard to how the vertices are connected. It is expected that the input data lies approximately in the plane; otherwise, projection perpendicular to one of the three coordinate axes may substantially change the geometry. The sign of the normal is chosen so that the sum of the signed areas of all input contours is nonnegative (where a CCW contour has positive area).

The supplied normal persists until it is changed by another call to **gluTessNormal**.

SEE ALSO

gluTessBeginPolygon, **gluTessEndPolygon**

NAME

gluTessProperty - set a tessellation object property

C SPECIFICATION

```
void gluTessProperty( GLUtesselator* tess,
                      GLenum which,
                      GLdouble data )
```

PARAMETERS

tess Specifies the tessellation object (created with **gluNewTess**).

which Specifies the property to be set. Valid values are **GLU_TESS_WINDING_RULE**, **GLU_TESS_BOUNDARY_ONLY**, **GLU_TESS_TOLERANCE**.

data Specifies the value of the indicated property.

DESCRIPTION

gluTessProperty is used to control properties stored in a tessellation object. These properties affect the way that the polygons are interpreted and rendered. The legal values for *which* are as follows:

GLU_TESS_WINDING_RULE

Determines which parts of the polygon are on the "interior". *data* may be set to one of **GLU_TESS_WINDING_ODD**, **GLU_TESS_WINDING_NONZERO**, **GLU_TESS_WINDING_POSITIVE**, or **GLU_TESS_WINDING_NEGATIVE**, or **GLU_TESS_WINDING_ABS_GEQ_TWO**.

To understand how the winding rule works, consider that the input contours partition the plane into regions. The winding rule determines which of these regions are inside the polygon.

For a single contour C, the winding number of a point x is simply the signed number of revolutions we make around x as we travel once around C (where CCW is positive). When there are several contours, the individual winding numbers are summed. This procedure associates a signed integer value with each point x in the plane. Note that the winding number is the same for all points in a single region.

The winding rule classifies a region as "inside" if its winding number belongs to the chosen category (odd, nonzero, positive, negative, or absolute value of at least two). The previous GLU tessellator (prior to GLU 1.2) used the "odd" rule. The "nonzero" rule is another common way to define the interior. The other three rules are useful for polygon CSG operations.

GLU_TESS_BOUNDARY_ONLY

Is a boolean value ("value" should be set to GL_TRUE or GL_FALSE). When set to GL_TRUE, a set of closed contours separating the polygon interior and exterior are returned instead of a tessellation. Exterior contours are oriented CCW with respect to the normal; interior contours are oriented CW. The **GLU_TESS_BEGIN** and **GLU_TESS_BEGIN_DATA** callbacks use the type GL_LINE_LOOP for each contour.

GLU_TESS_TOLERANCE

Specifies a tolerance for merging features to reduce the size of the output. For example, two vertices that are very close to each other might be replaced by a single vertex. The tolerance is multiplied by the largest coordinate magnitude of any input vertex; this specifies the maximum distance that any feature can move as the result of a single merge operation. If a single feature takes part in several merge operations, the total distance moved could be larger.

Feature merging is completely optional; the tolerance is only a hint. The implementation is free to merge in some cases and not in others, or to never merge features at all.

The initial tolerance is 0.

The current implementation merges vertices only if they are exactly coincident, regardless of the current tolerance. A vertex is spliced into an edge only if the implementation is unable to distinguish which side of the edge the vertex lies on. Two edges are merged only when both endpoints are identical.

SEE ALSO

gluGetTessProperty

NAME

gluTessVertex - specify a vertex on a polygon

C SPECIFICATION

```
void gluTessVertex( GLUtesselator* tess,
                    GLdouble *location,
                    GLvoid* data )
```

PARAMETERS

tess Specifies the tessellation object (created with **gluNewTess**).

location Specifies the location of the vertex.

data Specifies an opaque pointer passed back to the program with the vertex callback (as specified by **gluTessCallback**).

DESCRIPTION

gluTessVertex describes a vertex on a polygon that the program defines. Successive **gluTessVertex** calls describe a closed contour. For example, to describe a quadrilateral **gluTessVertex** should be called four times. **gluTessVertex** can only be called between **gluTessBeginContour** and **gluTessEndContour**.

data normally points to a structure containing the vertex location, as well as other per-vertex attributes such as color and normal. This pointer is passed back to the user through the **GLU_TESS_VERTEX** or **GLU_TESS_VERTEX_DATA** callback after tessellation (see the **gluTessCallback** reference page).

EXAMPLE

A quadrilateral with a triangular hole in it can be described as follows:

```
gluTessBeginPolygon(tobj, NULL);
 gluTessBeginContour(tobj);
   gluTessVertex(tobj, v1, v1);
   gluTessVertex(tobj, v2, v2);
   gluTessVertex(tobj, v3, v3);
   gluTessVertex(tobj, v4, v4);
 gluTessEndContour(tobj);
 gluTessBeginContour(tobj);
   gluTessVertex(tobj, v5, v5);
   gluTessVertex(tobj, v6, v6);
   gluTessVertex(tobj, v7, v7);
 gluTessEndContour(tobj); gluTessEndPolygon(tobj);
```

NOTES

It is a common error to use a local variable for *location* or *data* and store values into it as part of a loop. For example:

```
for (i = 0; i < NVERTICES; ++i) {
  GLdouble data[3];
  data[0] = vertex[i][0];
  data[1] = vertex[i][1];
  data[2] = vertex[i][2];
  gluTessVertex(tobj, data, data);
  }
```

This doesn't work. Because the pointers specified by *location* and *data* might not be dereferenced until **gluTessEndPolygon** is executed, all the vertex coordinates but the very last set could be overwritten before tessellation begins.

Two common symptoms of this problem are consists of a single point (when a local variable is used for *data*) and a **GLU_TESS_NEED_COMBINE_CALLBACK** error (when a local variable is used for *location*).

SEE ALSO

gluTessBeginPolygon, **gluNewTess**, **gluTessBeginContour**, **gluTessCallback**, **gluTessProperty**, **gluTessNormal**, **gluTessEndPolygon**

NAME

gluUnProject - map window coordinates to object coordinates

C SPECIFICATION

```
GLint gluUnProject( GLdouble winX,
                    GLdouble winY,
                    GLdouble winZ,
                    const GLdouble *model,
                    const GLdouble *proj,
                    const GLint *view,
                    GLdouble* objX,
                    GLdouble* objY,
                    GLdouble* objZ )
```

PARAMETERS

winX, *winY*, *winZ*
: Specify the window coordinates to be mapped.

model Specifies the modelview matrix (as from a **glGetDoublev** call).

proj Specifies the projection matrix (as from a **glGetDoublev** call).

view Specifies the viewport (as from a **glGetIntegerv** call).

objX, *objY*, *objZ*
: Returns the computed object coordinates.

DESCRIPTION

gluUnProject maps the specified window coordinates into object coordinates using *model*, *proj*, and *view*. The result is stored in *objX*, *objY*, and *objZ*. A return value of **GL_TRUE** indicates success; a return value of **GL_FALSE** indicates failure.

To compute the coordinates (*objX*, *objY*, and *objZ*), **gluUnProject** multiplies the normalized device coordinates by the inverse of *model**proj* as follows:

```
                   |                       |
                   |_________________      |
(       )          |     view[2]      - 1  |
| objX   |         |                       |
| objY   | = INV(PM)|_________________ - 1  |
|       |          |     view[3]           |
| objZ   |         |     2(winZ) - 1       |
(  W    )          |                       |
                   (          1            )
```

INV() denotes matrix inversion. W is an unused variable, included for consistent matrix notation.

SEE ALSO

glGet, **gluProject**