The OpenGL™ Graphics System Utility Library

Kevin P. Smith
Silicon Graphics

1 Overview

The GL Utilities (GLU) library is a set of routines designed to complement the **OpenGL™** graphics system by providing support for mipmapping, matrix manipulation, polygon tessellation, quadrics, NURBS, and error handling. Mipmapping routines include image scaling and automatic mipmap generation. A variety of matrix manipulation functions build projection and viewing matrices, or project vertices from one coordinate system to another. Polygon tessellation routines convert concave polygons into triangles for easy rendering. Quadrics support renders a few basic quadrics such as spheres and cones. NURBS code maps complicated NURBS curves and trimmed surfaces into simpler **OpenGL** evaluators. Lastly, an error lookup routine translates **OpenGL** and GLU error codes into strings.

2 Mipmapping

GLU provides image scaling and automatic mipmapping functions to simplify the creation of textures. The image scaling function can scale any image to a legal texture size. The resulting image can then be passed to **OpenGL** as a texture. The automatic mipmapping routines will take an input image, create mipmap textures from it, and pass them to **OpenGL**. With this interface, the user need only supply an image and the rest is automatic.

2.1 Image Scaling

The following routine magnifies or shrinks an image:

> int **gluScaleImage**(GLenum *format*, GLsizei *widthin*, GLsizei *heightin*, GLenum *typein*, const void **datain*, GLsizei *widthout*, GLsizei *heightout*, GLenum *typeout*, void **dataout*);

gluScaleImage will scale an image using the appropriate pixel store modes to unpack data from the input image and pack the result into the output image. *format* specifies the image format used by both images (and may be any of the formats supported by **glDrawPixels**). The input image is described by *widthin*, *heightin*, *typein*, and *datain*, where *widthin* and *heightin* specify the size of the image, *typein* specifies the data type used (as in **glDrawPixels**), and *datain* is a pointer to the image data in memory. The output image is similarly described by *widthout*, *heightout*, *typeout*, and *dataout*, where *widthout* and *heightout* specify the desired size of the image, *typeout* specifies the desired data type, and *dataout* points to the memory location where the image is to be stored.

gluScaleImage reconstructs the input image by linear interpolation, convolves it with a one-pixel-square box kernel, and then samples the result to produce the output image.

A return value of 0 indicates success. Otherwise the return value is a GLU error code indicating the cause of the problem (see **gluErrorString** below).

2.2 Automatic Mipmapping

These routines will automatically generate mipmaps for any image provided by the user and then pass them to **OpenGL**:

> int **gluBuild1DMipmaps**(GLenum *target*, GLint *components*, GLsizei *width*, GLenum *format*, GLenum *type*, const void **data*);

> int **gluBuild2DMipmaps**(GLenum *target*, GLint *components*, GLsizei *width*, GLsizei *height*, GLenum *format*, GLenum *type*, const void **data*);

gluBuild1DMipmaps and **gluBuild2DMipmaps** both take an input image and derive from it a pyramid of scaled images suitable for use as mipmapped textures. The resulting textures are then passed to **glTexImage1D** or **glTexImage2D** as appropriate. *target*, *components*, *format*, *type*, *width*, *height*, and *data* define the level 0 texture, and have the same meaning as the corresponding arguments to **glTexImage1D** and **glTexImage2D**. Note that the image size does not need to be a power of 2, because the image will be automatically scaled to the nearest power of 2 size if necessary.

A return value of 0 indicates success. Otherwise the return value is a GLU error code indicating the cause of the problem.

3 Matrix Manipulation

The GLU library includes support for matrix creation and coordinate projection (transformation). The matrix routines create matrices and multiply the current **OpenGL** matrix by the result. They are used for setting projection and viewing parameters. The coordinate projection routines are used to transform object space coordinates into screen coordinates or vice-versa. This makes it possible to determine where in the window an object is being drawn.

3.1 Matrix setup

The following routines create projection and viewing matrices and apply them to the current matrix using **glMultMatrix**. With these routines, a user can construct a clipping volume and set viewing parameters to render a scene.

gluOrtho2D and **gluPerspective** build commonly-needed projection matrices.

> void **gluOrtho2D**(GLdouble *left*, GLdouble *right*, GLdouble *bottom*, GLdouble *top*);

sets up a two dimensional orthographic viewing region. The parameters define the bounding box of the region to be viewed. Calling **gluOrtho2D (***left***,** *right***,** *bottom***,** *top***)** is equivalent to calling **glOrtho(***left***,** *right***,** *bottom***,** *top***, -1, 1)**.

> void **gluPerspective**(GLdouble *fovy*, GLdouble *aspect*, GLdouble *near*,
> GLdouble *far*);

sets up a perspective viewing volume. *fovy* defines the field-of-view angle (in degrees) in the y direction. *aspect* is the aspect ratio used to determine the field-of-view in the x direction. It is the ratio of x (width) to y (height). *near* and *far* define the near and far clipping planes (as positive distances from the eye point).

gluLookAt creates a commonly-used viewing matrix:

> void **gluLookAt**(GLdouble *eyex*, GLdouble *eyey*, GLdouble *eyez*,
> GLdouble *centerx*, GLdouble *centery*, GLdouble *centerz*,
> GLdouble *upx*, GLdouble *upy*, GLdouble *upz*);

The viewing matrix created is based on an eye point (*eyex*, *eyey*, *eyez*), a reference point that represents the center of the scene (*centerx*, *centery*, *centerz*), and an up vector (*upx*, *upy*, *upz*). The matrix is designed to map the center of the scene to the negative Z axis, so that when a typical projection matrix is used, the center of the scene will map to the center of the viewport. Similarly, the projection of the up vector on the viewing plane is mapped to the positive Y axis so that it will point upward in the viewport. The up vector must not be parallel to the line-of-sight from the eye to the center of the scene.

gluPickMatrix is designed to simplify selection by creating a matrix that restricts drawing to a small region of the viewport. This is typically used to determine which objects are being drawn near the cursor. First restrict drawing to a small region around the cursor, then rerender the scene with selection mode turned on. All objects that were being drawn near the cursor will be selected and stored in the selection buffer.

> void **gluPickMatrix**(GLdouble *x*, GLdouble *y*, GLdouble *deltax*,
> GLdouble *deltay*, GLint *viewport*[4]);

gluPickMatrix should be called just before applying a projection matrix to the stack (effectively pre-multiplying the projection matrix by the selection matrix). *x* and *y* specify the center of the selection bounding box in pixel coordinates; *deltax* and *deltay* specify its width and height in pixels. *viewport* should specify the current viewport's x, y, width, and height. A convenient way to obtain this information is to call **glGetIntegerv(`GL_VIEWPORT`, viewport)**.

3.2 Coordinate Projection

Two routines are provided to project coordinates back and forth from object space to screen space. **gluProject** projects from object space to screen space, and **gluUn-**

Project does the reverse.

> int **gluProject**(GLdouble *objx*, GLdouble *objy*, GLdouble *objz*, const GLdouble *modelMatrix*[16], const GLdouble *projMatrix*[16], const GLint *viewport*[4], GLdouble **winx*, GLdouble **winy*, GLdouble **winz*);

gluProject performs the projection with the given *modelMatrix*, *projectionMatrix*, and *viewport*. The format of these arguments is the same as if they were obtained from **glGetDoublev** and **glGetIntegerv**. A return value of **`GL_TRUE`** indicates success, and **`GL_FALSE`** indicates failure.

> int **gluUnProject**(GLdouble *winx*, GLdouble *winy*, GLdouble *winz*, const GLdouble *modelMatrix*[16], const GLdouble *projMatrix*[16], const GLint *viewport*[4], GLdouble **objx*, GLdouble **objy*, GLdouble **objz*);

gluUnProject uses the given *modelMatrix*, *projectionMatrix*, and *viewport* to perform the projection. A return value of **`GL_TRUE`** indicates success, and **`GL_FALSE`** indicates failure.

4 Polygon Tessellation

The polygon tessellation routines triangulate concave polygons with one or more contours. To use these routines, first create a tessellation object. Second, define callback routines that will be used to process the triangles generated by the tessellator. Finally, specify the concave polygon to be tessellated.

The triangles produced by tessellation will be oriented in the same direction as the largest input contour. For example, if the largest input contour is oriented counter-clockwise in some plane, then all of the output triangles will also be oriented counter-clockwise in that plane.

These routines do not support self-intersecting polygons. This includes polygons with multiple coincident vertices.

4.1 The Tessellation Object

A new tessellation object is created with **gluNewTess**:

> GLUtriangulatorObj **tessobj*;
> *tessobj* = **gluNewTess**(void);

gluNewTess returns a new tessellation object, which is used by the other tessellation functions. A return value of 0 indicates an out-of-memory error.

When a tessellation object is no longer needed, it should be deleted with **gluDele-**

teTess:

void **gluDeleteTess**(GLUtriangulatorObj **tessobj*);

This will destroy the object and free any memory used by it.

4.2 Callbacks

When a concave polygon is tessellated, the triangles created by the process are described to the user by a series of callbacks. These callbacks are specified with **gluTessCallback**:

void **gluTessCallback**(GLUtriangulatorObj **tessobj*, GLenum *which*, void (**fn*)());

This routine replaces the callback selected by *which* with the function specified by *fn*. If *fn* is equal to **NULL**, then any previously defined callback is discarded. Any callbacks discarded or left unspecified will not be called, and any information that they would have provided is lost.

It is legal to leave any of the callbacks undefined. However, the information that they would have provided is lost.

which may be one of **GLU_BEGIN**, **GLU_EDGE_FLAG**, **GLU_VERTEX**, **GLU_END**, or **GLU_ERROR**. The five callbacks have the following prototypes:

void *begin*(GLenum *type*);
void *edgeFlag*(GLboolean *flag*);
void *vertex*(void **data*);
void *end*(void);
void *error*(GLenum *errno*);

The *begin* callback is invoked like **glBegin** to indicate the start of a triangle primitive. This callback will be called with either **GL_TRIANGLE_FAN**, **GL_TRIANGLE_STRIP**, or **GL_TRIANGLES**.

The *edgeFlag* callback is similar to **glEdgeFlag**. It is used to indicate which edges of the created triangles were part of the original polygon, and which were created by the tessellation process. If *flag* is **GL_TRUE**, then each vertex that follows begins an edge that was part of the original polygon. If *flag* is **GL_FALSE**, then each vertex that follows begins an edge that was created by the tessellator. To avoid confusion with the first few edges, the *edgeFlag* callback will be invoked before the first *vertex* callback is made. Since triangle fans and triangle strips do not support edge flags, the *begin* callback will not be invoked with **GL_TRIANGLE_FAN** or **GL_TRIANGLE_STRIP** if an *edgeFlag* callback is provided. Instead, fans and strips will be converted to independent triangles.

The *vertex* callback is invoked between the *begin* and *end* callbacks. It is similar to **glVertex**, and defines the vertices of the triangles created by the tessellation process. *data* is a copy of the pointer that the user provided when the vertex was specified (see **gluTessVertex** below).

The *end* callback serves the same purpose as **glEnd**, and indicates the end of a primitive.

The *error* callback is invoked when an error is encountered. There are eight errors unique to polygon tessellation and they are named **GLU_TESS_ERROR1** through **GLU_TESS_ERROR8**. Character strings describing these errors can be retrieved with the **gluErrorString** call (see below).

4.3 Polygon Definition

The input polygon is specified with the following routines:

> void **gluBeginPolygon**(GLUtriangulatorObj **tessobj*);
> void **gluTessVertex**(GLUtriangulatorObj **tessobj*, GLdouble *location*[3],
>   void **data*);
> void **gluNextContour**(GLUtriangulatorObj **tessobj*, GLenum *type*);
> void **gluEndPolygon**(GLUtriangulatorObj **tessobj*);

gluBeginPolygon indicates the start of the polygon, and it must be called first.

gluTessVertex describes a polygon vertex. Successive **gluTessVertex** calls describe a closed contour; for example, if the input polygon is a quadrilateral, then **gluTessVertex** should be called four times. *location* specifies the position of the vertex. *data* is an opaque pointer that is passed back to the user when the *vertex* callback is invoked; it typically contains a copy of the vertex location, color, surface normal, and other per-vertex attributes. The current tessellator never generates new vertices, so it never interpolates these attributes.

gluNextContour is called once before each contour, and specifies the type of contour that follows. *type* is one of **GLU_EXTERIOR**, **GLU_INTERIOR**, **GLU_CCW**, **GLU_CW**, or **GLU_UNKNOWN**. A **GLU_EXTERIOR** contour defines an exterior boundary of the polygon, and a **GLU_INTERIOR** contour defines an interior boundary of the polygon (a hole). **GLU_UNKNOWN** contours will be analyzed by the library to determine if they are interior or exterior.

The **GLU_CCW** and **GLU_CW** contour types are global types, and 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** or **GLU_CW** contours will be changed to **GLU_UNKNOWN**).

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 counter-clockwise) as the first contour, and interior if they are not. Note that there is no real difference between the **GLU_CCW** and **GLU_CW** types.

If **gluNextContour** is not called before the first contour, then the first contour is assumed to be a **GLU_EXTERIOR** contour.

gluEndPolygon defines the end of the polygon. When **gluEndPolygon** is called the polygon will be tessellated, and the resulting triangles will be described through the callbacks.

5 Quadrics

The GLU library quadrics routines will render spheres, cylinders and disks in a variety of styles as specified by the user. To use these routines, first create a quadrics object. This object contains state indicating how a quadric should be rendered. Second, modify this state using the function calls described below. Finally, render the desired quadric by invoking the appropriate quadric rendering routine.

5.1 The Quadrics Object

A quadrics object is created with **gluNewQuadric**:

> GLUquadricObj *_quadobj_;
> _quadobj_ = **gluNewQuadric**(void);

gluNewQuadric returns a new quadrics object. This object contains state describing how a quadric should be constructed and rendered. A return value of 0 indicates an out-of-memory error.

When the object is no longer needed, it should be deleted with **gluDeleteQuadric**:

> void **gluDeleteQuadric**(GLUquadricObj *_quadobj_);

This will delete the quadrics object and any memory used by it.

5.2 Callbacks

To associate a callback with the quadrics object, use **gluQuadricCallback**:

> void **gluQuadricCallback**(GLUquadricObj *_quadobj_, GLenum _which_,
> void (*_fn_)());

The only callback provided for quadrics is the **GLU_ERROR** callback (identical to the polygon tessellation callback described above). This callback takes an error code as its only argument. To translate the error code to an error message, see **gluErrorString** below.

5.3 Rendering Styles

A variety of variables control how a quadric will be drawn. These are *normals*, *textureCoords*, *orientation*, and *drawStyle*. *normals* indicates if surface normals should be generated, and if there should be one normal per vertex or one normal per face. *textureCoords* determines whether texture coordinates should be generated. *orientation* describes which side of the quadric should be the "outside." Lastly, *drawStyle* indicates if the quadric should be drawn as a set of polygons, lines, or points.

To specify the kind of normals desired, use **gluQuadricNormals**:

> void **gluQuadricNormals**(GLUquadricObj **quadobj*, GLenum *normals*);

normals is either **GLU_NONE** (no normals), **GLU_FLAT** (one normal per face) or **GLU_SMOOTH** (one normal per vertex). The default is **GLU_SMOOTH**.

Texture coordinate generation can be turned on and off with **gluQuadricTexture**:

> void **gluQuadricTexture**(GLUquadricObj **quadobj*, GLboolean *textureCoords*);

If *textureCoords* is **GL_TRUE**, then texture coordinates will be generated when a quadric is rendered. Note that how texture coordinates are generated depends upon the specific quadric. The default is **GL_FALSE**.

An orientation can be specified with **gluQuadricOrientation**:

> void **gluQuadricOrientation**(GLUquadricObj **quadobj*, GLenum *orientation*);

If *orientation* is **GLU_OUTSIDE** then quadrics will be drawn with normals pointing outward. If *orientation* is **GLU_INSIDE** then the normals will point inward (faces are rendered counter-clockwise with respect to the normals). Note that "outward" and "inward" are defined by the specific quadric. The default is **GLU_OUTSIDE**.

A drawing style can be chosen with **gluQuadricDrawStyle**:

> void **gluQuadricDrawStyle**(GLUquadricObj **quadobj*, GLenum *drawStyle*);

drawStyle is one of **GLU_FILL**, **GLU_LINE**, **GLU_POINT**, or **GLU_SILHOUETTE**. In **GLU_FILL** mode, the quadric is rendered as a set of polygons, in **GLU_LINE** mode as a set of lines, and in **GLU_POINT** mode as a set of points. **GLU_SILHOUETTE** mode is similar to **GLU_LINE** mode except that edges separating coplanar faces are not drawn. The default style is **GLU_FILL**.

5.4 Quadrics Primitives

The four supported quadrics are spheres, cylinders, disks, and partial disks. Each of these quadrics may be subdivided into arbitrarily small pieces.

A sphere can be created with **gluSphere**:

> void **gluSphere**(GLUquadricObj **quadobj*, GLdouble *radius*, GLint *slices*, GLint *stacks*);

This renders a sphere of the given *radius* centered around the origin. The sphere is subdivided along the Z axis into the specified number of *stacks*, and each stack is then sliced evenly into the given number of *slices*. Note that the globe is subdivided in an analogous fashion, where lines of latitude represent *stacks*, and lines of longitude represent *slices*.

If texture coordinate generation is enabled then coordinates are computed 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.

A cylinder is specified with **gluCylinder**:

> void **gluCylinder**(GLUquadricObj **quadobj*, GLdouble *baseRadius*, GLdouble *topRadius*, GLdouble *height*, GLint *slices*, GLint *stacks*);

gluCylinder draws a frustum of a cone centered on the Z axis with the base at Z=0 and the top at Z=*height*. *baseRadius* specifies the radius at Z=0, and *topRadius* specifies the radius at Z=*height*. (If *baseRadius* equals *topRadius*, the result is a conventional cylinder.) Like a sphere, a cylinder is subdivided along the Z axis into *stacks*, and each stack is further subdivided into *slices*. When textured, t ranges linearly from 0.0 to 1.0 along the Z axis, and s ranges from 0.0 to 1.0 around the Z axis (in the same manner as it does for a sphere).

A disk is created with **gluDisk**:

> void **gluDisk**(GLUquadricObj **quadobj*, GLdouble *innerRadius*, GLdouble *outerRadius*, GLint *slices*, GLint *loops*);

This renders a disk on the Z=0 plane. The disk has the given *outerRadius*, and if *innerRadius* > 0.0 then it will contain a central hole with the given *innerRadius*. The disk is subdivided into the specified number of *slices* (similar to cylinders and spheres), and also into the specified number of *loops* (concentric rings about the origin). With respect to orientation, the +Z side of the disk is considered to be “outside”.

When textured, coordinates are generated in a linear grid such that the value of (s, t) at (*outerRadius*, 0, 0) is (1, 0.5), at (0, *outerRadius*, 0) it is (0.5, 1), at (-*outerRa-*

dius, 0, 0) it is (0, 0.5), and at (0, -*outerRadius*, 0) it is (0.5, 0). This allows a 2D texture to be mapped onto the disk without distortion.

A partial disk is specified with **gluPartialDisk**:

> void **gluPartialDisk**(GLUquadricObj **quadobj*, GLdouble *innerRadius*, GLdouble *outerRadius*, GLint *slices*, GLint *loops*, GLdouble *startAngle*, GLdouble *sweepAngle*);

This function is identical to **gluDisk** except that only the subset of the disk from *startAngle* through *startAngle* + *sweepAngle* is included (where 0 degrees is along the +Y axis, 90 degrees is along the +X axis, 180 is along the -Y axis, and 270 is along the -X axis). In the case that *drawStyle* is set to either **`GLU_FILL`** or **`GLU_SILHOUETTE`**, the edges of the partial disk separating the included area from the excluded arc will be drawn.

6 NURBS

NURBS curves and surfaces are converted to **OpenGL** evaluators by the functions in this section. The interface employs a NURBS object to describe the curves and surfaces and to specify how they should be rendered. Basic trimming support is included to allow more flexible definition of surfaces.

6.1 The NURBS Object

A NURBS object is created with **gluNewNurbsRenderer**:

> GLUnurbsObj **nurbsObj*;
> *nurbsObj* = **gluNewNurbsRenderer**(void);

nurbsObj is an opaque pointer to all of the state information needed to tessellate and render a NURBS curve or surface. Before any of the other routines in this section can be used, a NURBS object must be created. A return value of 0 indicates an out of memory error.

When a NURBS object is no longer needed, it should be deleted with **gluDeleteNurbsRenderer**:

> void **gluDeleteNurbsRenderer**(GLUnurbsObj **nurbsObj*);

This will destroy all state contained in the object, and free any memory used by it.

6.2 Callbacks

To define a callback for a NURBS object, use:

void **gluNurbsCallback**(GLUnurbsObj *nurbsObj*, GLenum *which*, void (*fn*)());

Currently, the only callback supported is the **GLU_ERROR** callback (identical to the **GLU_ERROR** callback used by **gluTessCallback**). When a NURBS function detects an error condition, the *error* callback is invoked with an error code as its only argument. There are 37 errors specific to NURBS functions, and they are named **GLU_NURBS_ERROR1** through **GLU_NURBS_ERROR37**. Strings describing the meaning of these error codes can be retrieved with **gluErrorString** (see below).

6.3 NURBS curves

NURBS curves are specified with the following routines:

void **gluBeginCurve**(GLUnurbsObj *nurbsObj*);
void **gluNurbsCurve**(GLUnurbsObj *nurbsObj*, GLint *nknots*, GLfloat *knot*, GLint *stride*, GLfloat *ctlarray*, GLint *order*, GLenum *type*);
void **gluEndCurve**(GLUnurbsObj *nurbsObj*);

gluBeginCurve and **gluEndCurve** delimit a curve definition. After the **gluBeginCurve** and before the **gluEndCurve**, a series of **gluNurbsCurve** calls specify the attributes of the curve. *type* can be any of the one dimensional evaluators (such as **GL_MAP1_VERTEX_3**). *knot* points to an array of non-decreasing knot values, and *nknots* tells how many knots are in the array. *ctlarray* points to an array of control points, and *order* indicates the order of the curve. The number of control points in *ctlarray* will be equal to *nknots* - *order*. Lastly, *stride* indicates the offset (expressed in terms of single precision values) between control points.

The NURBS curve attribute definitions must include either a **GL_MAP1_VERTEX3** description or a **GL_MAP1_VERTEX4** description.

At the point that **gluEndCurve** is called, the curve will be tessellated into line segments and rendered with the aid of **OpenGL** evaluators. **glPushAttrib** and **glPopAttrib** are used to preserve the previous evaluator state during rendering.

6.4 NURBS surfaces

NURBS surfaces are described with the following routines:

void **gluBeginSurface**(GLUnurbsObj *nurbsObj*);
void **gluNurbsSurface**(GLUnurbsObj *nurbsObj*, GLint *sknot_count*, GLfloat *sknot*, GLint *tknot_count*, GLfloat *tknot*, GLint *s_stride*, GLint *t_stride*, GLfloat *ctlarray*, GLint *sorder*, GLint *torder*, GLenum *type*);
void **gluEndSurface**(GLUnurbsObj *nurbsObj*);

The surface description is almost identical to the curve description. **gluBeginSurface** and **gluEndSurface** delimit a surface definition. After the **gluBeginSurface**, and before the **gluEndSurface**, a series of **gluNurbsSurface** calls specify the attributes of the surface. *type* can be any of the two dimensional evaluators (such as **`GL_MAP2_VERTEX_3`**). *sknot* and *tknot* point to arrays of non-decreasing knot values, and *sknot_count* and *tknot_count* indicate how many knots are in each array. *ctlarray* points to an array of control points, and *sorder* and *torder* indicate the order of the surface in both the s and t directions. The number of control points in *ctlarray* will be equal to (*sknot_count* - *sorder*) x (*tknot_count* - *torder*). Finally, *s_stride* and *t_stride* indicate the offset in single precision values between control points in the s and t directions.

The NURBS surface, like the NURBS curve, must include an attribute definition of type **`GL_MAP2_VERTEX3`** or **`GL_MAP2_VERTEX4`**.

When **gluEndSurface** is called, the NURBS surface will be tessellated and rendered with the aid of **OpenGL** evaluators. The evaluator state is preserved during rendering with **glPushAttrib** and **glPopAttrib**.

6.5 Trimming

A trimming region defines a subset of the NURBS surface domain to be evaluated. By limiting the part of the domain that is evaluated, it is possible to create NURBS surfaces that contain holes or have smooth boundaries.

A trimming region is defined by a set of closed trimming loops in the parameter space of a surface. When a loop is oriented counter-clockwise, the area within the loop is retained, and the part outside is discarded. When the loop is oriented clockwise, the area within the loop is discarded, and the rest is retained. Loops may be nested, but a nested loop must be oriented oppositely from the loop that contains it. The outermost loop must be oriented counter-clockwise.

A trimming loop consists of a connected sequence of NURBS curves and piecewise linear curves. The last point of every curve in the sequence must be the same as the first point of the next curve, and the last point of the last curve must be the same as the first point of the first curve. Self-intersecting curves are not allowed.

To define trimming loops, use the following routines:

> void **gluBeginTrim**(GLUnurbsObj **nurbsObj*);
> void **gluPwlCurve**(GLUnurbsObj **nurbsObj*, GLint *count*, GLfloat **array*, GLint *stride*, GLenum *type*);
> void **gluNurbsCurve**(GLUnurbsObj **nurbsObj*, GLint *nknots*, GLfloat **knot*, GLint *stride*, GLfloat **ctlarray*, GLint *order*, GLenum *type*);
> void **gluEndTrim**(GLUnurbsObj **nurbsObj*);

A NURBS trimming curve is very similar to a regular NURBS curve, with the major difference being that a NURBS trimming curve exists in the parameter space of a NURBS surface.

gluPwlCurve defines a piecewise linear curve. *count* indicates how many points are on the curve, and *array* points to an array containing the curve points. *stride* indicates the offset in single precision values between curve points.

type for both **gluPwlCurve** and **gluNurbsCurve** can be either **GLU_MAP1_TRIM_2**, or **GLU_MAP1_TRIM_3**. **GLU_MAP1_TRIM_2** curves define trimming regions in two dimensional (s and t) parameter space. The **GLU_MAP1_TRIM_3** curves define trimming regions in two dimensional homogeneous (s, t, and q) parameter space.

Note that the trimming loops must be defined at the same time that the surface is defined (between **gluBeginSurface** and **gluEndSurface**).

6.6 NURBS properties

A set of properties associated with a NURBS object affects the way that NURBS are rendered. These properties can be adjusted by the user.

> void **gluNurbsProperty**(GLUnurbsObj **nurbsObj*, GLenum *property*,
> GLfloat *value*);

allows the user to set one of the following properties: **GLU_CULLING**, **GLU_SAMPLING_TOLERANCE**, **GLU_DISPLAY_MODE**, and **GLU_AUTO_LOAD_MATRIX**. *property* indicates the property to be modified, and *value* specifies the new value.

The **GLU_CULLING** property is a boolean value (*value* should be set to either **GL_TRUE** or **GL_FALSE**). When set to **GL_TRUE**, it indicates that a NURBS curve or surface should be discarded prior to tessellation if its control polyhedron lies outside the current viewport. Note that, in general, NURBS do not fall within the convex hull of their control points, so the default is **GL_FALSE**.

GLU_SAMPLING_TOLERANCE specifies the maximum length, in pixels, of edges of polygons used to render NURBS. The NURBS code is conservative when rendering a curve or surface, so the actual lengths may be somewhat shorter. The default value is 50.0 pixels.

GLU_AUTO_LOAD_MATRIX is a boolean value. When it is set to **GL_TRUE**, the NURBS code will download the projection matrix, the model view matrix, and the viewport from the **OpenGL** server in order to compute sampling and culling matrices for each curve or surface that is rendered. These matrices are required to tessellate a curve or surface and to cull it if it lies outside the viewport. If this mode is turned off, then the user needs to provide a projection matrix, a model view matrix, and a viewport that the NURBS code can use to construct sampling and culling matrices. This can be done with the **gluLoadSamplingMatrices** function:

void **gluLoadSamplingMatrices**(GLUnurbsObj **nurbsObj*, const GLfloat *modelMatrix*[16], const GLfloat *projMatrix*[16], const GLint *viewport*[4]);

Until the **GLU_AUTO_LOAD_MATRIX** property is turned back on, the NURBS routines will continue to use whatever sampling and culling matrices are stored in the NURBS object. The default for **GLU_AUTO_LOAD_MATRIX** is **GL_TRUE**.

GLU_DISPLAY_MODE specifies how a NURBS surface should be rendered. *value* may be set to one of **GLU_FILL**, **GLU_OUTLINE_POLY**, or **GLU_OUTLINE_PATCH**. When set to **GLU_FILL**, the surface is rendered as a set of polygons. **GLU_OUTLINE_POLY** instructs the NURBS library to draw only the outlines of the polygons created by tessellation. **GLU_OUTLINE_PATCH** will cause just the outlines of patches and trim curves defined by the user to be drawn. The default is **GLU_FILL**.

Property values can also be queried by the user:

void **gluGetNurbsProperty**(GLUnurbsObj **nurbsObj*, GLenum *property*, GLfloat **value*);

will load *value* with the value of the *property* specified.

7 Errors

const GLubyte ***gluErrorString**(GLenum *errorCode*);

gluErrorString produces an error string that corresponds to a GL or GLU error code. The error string is in ISO Latin 1 format. The standard GLU error codes are **GLU_INVALID_ENUM**, **GLU_INVALID_VALUE**, and **GLU_OUT_OF_MEMORY**. There are also specific error codes for polygon tessellation, quadrics, and NURBS as described in their respective sections.