The OpenGL® Graphics System Utility Library (Version 1.3)

Norman Chin
Chris Frazier
Paul Ho
Zicheng Liu
Kevin P. Smith

Editor (version 1.3): Jon Leech

Contents

Chapter 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. GLU library routines may call OpenGL library routines. Thus, an OpenGL context should be made current before calling any GLU functions. Otherwise an OpenGL error may occur.

All GLU routines, except for the initialization routines listed in Section 2, may be called during display list creation. This will cause any OpenGL commands that are issued as a result of the call to be stored in the display list. The result of calling the intialization routines after **glNewList** is undefined.

Chapter 2

Initialization

To get the GLU version number or supported GLU extensions call:

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

If *name* is `GLU_VERSION` or `GLU_EXTENSIONS`, then a pointer to a static zero-terminated string that describes the version or available extensions respectively is returned; otherwise `NULL` is returned.

The version string is laid out as follows:

<*version number*><*space*><*vendor-specific information*>

version number is either of the form *major_number.minor_number* or *major_number.minor_number.release_number*, where the numbers all have one or more digits. The version number determines which interfaces are provided by the GLU client library. If the underlying OpenGL implementation is an older version than that corresponding to this version of GLU, some of the GL calls made by GLU may fail. Chapter 9 describes how GLU versions and OpenGL versions correspond.

The vendor specific information is optional. However, if it is present the format and contents are implementation dependent.

The extension string is a space separated list of extensions to the GLU library. The extension names themselves do not contain any spaces. To determine if a specific extension name is present in the extension string, call

`GLboolean` **gluCheckExtension**(`char` **extName*,
 `const GLubyte` **extString*);

where *extName* is the extension name to check, and *extString* is the extension string. `GL_TRUE` is returned if *extName* is present in *extString*, `GL_FALSE` otherwise. **gluCheckExtension** correctly handles boundary cases where one extension name is a substring of another. It may also be used to checking for the presence of OpenGL or GLX extensions by passing the extension strings returned by **glGetString** or **glXGetClientString**, instead of the GLU extension string.

gluGetString is not available in GLU 1.0. One way to determine whether this routine is present when using the X Window System is to query the GLX version. If the client version is 1.1 or greater then this routine is available. Operating system dependent methods may also be used to check for the existence of this function.

Chapter 3

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.

3.1 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. 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, 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. The pixel formats and types supported are the same as those supported by **glDrawPixels** for the underlying OpenGL implementation.

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).

3.2 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 *internalFormat*, GLsizei *width*, GLenum *format*,
GLenum *type*, const void **data*);

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

int **gluBuild3DMipmaps**(GLenum *target*,
GLint *internalFormat*, GLsizei *width*, GLsizei *height*,
GLsizei *depth*, GLenum *format*, GLenum *type*,
const void **data*);

gluBuild1DMipmaps, **gluBuild2DMipmaps**, and **gluBuild3DMipmaps** all 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**, **glTexImage2D**, or **glTexImage3D** as appropriate. *target*, *internalFormat*, *format*, *type*, *width*, *height*, *depth*, and *data* define the level 0 texture, and have the same meaning as the corresponding arguments to **glTexImage1D**, **glTexImage2D**, and **glTexImage3D**. 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.

To load only a subset of mipmap levels, call

int **gluBuild1DMipmapLevels**(GLenum *target*,
GLint *internalFormat*, GLsizei *width*, GLenum *format*,

```
    GLenum type, GLint level, GLint base, GLint max,
    const void *data );

int gluBuild2DMipmapLevels( GLenum target,
    GLint internalFormat, GLsizei width, GLsizei height,
    GLenum format, GLenum type, GLint level, GLint base,
    GLint max, const void *data );

int gluBuild3DMipmapLevels( GLenum target,
    GLint internalFormat, GLsizei width, GLsizei height,
    GLsizei depth, GLenum format, GLenum type, GLint level,
    GLint base, GLint max, const void *data );
```

level specifies the mipmap level of the *input* image. *base* and *max* determine the minimum and maximum mipmap levels which will be passed to **glTexImage*x*D**. Other parameters are the same as for **gluBuild*x*DMipmaps**. If $level > base$, $base < 0$, $max < base$, or *max* is larger than the highest mipmap level for a texture of the specified size, no mipmap levels will be loaded, and the calls will return GLU_INVALID_VALUE.

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

Chapter 4

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.

4.1 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,
    const 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*).

4.2 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 **gluUnProject** does the reverse. **gluUnProject4** should be used instead of **gluUnProject** when a nonstandard **glDepthRange** is in effect, or when a clip-space w coordinate other than 1 needs to be specified, as for vertices in the OpenGL **glFeedbackBuffer** when data type `GL_4D_COLOR_TEXTURE` is returned.

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.

int **gluUnProject4**(GLdouble *winx*, GLdouble *winy*,
GLdouble *winz*, GLdouble *clipw*,
const GLdouble *modelMatrix[16]*,
const GLdouble *projMatrix[16]*, const GLint *viewport[4]*,
GLclampd *near*, GLclampd *far*, GLdouble **objx*,
GLdouble **objy*, GLdouble **objz*, GLdouble **objw*);

gluUnProject4 takes three additional parameters and returns one additional parameter *clipw* is the clip-space w coordinate of the screen-space vertex (e.g. the w_c value computed by OpenGL); normally, $clipw = 1$. *near* and *far* correspond to the current **glDepthRange**; normally, $near = 0$ and $far = 1$. The object-space w value of the unprojected vertex is returned in $objw$. Other parameters are the same as for **gluUnProject**.

Chapter 5

Polygon Tessellation

The polygon tessellation routines triangulate concave polygons with one or more closed contours. Several winding rules are supported to determine which parts of the polygon are on the "interior". In addition, boundary extraction is supported: instead of tessellating the polygon, a set of closed contours separating the interior from the exterior are generated.

To use these routines, first create a tessellation object. Second, define the callback routines and the tessellation parameters. (The callback routines are used to process the triangles generated by the tessellator.) Finally, specify the concave polygon to be tessellated.

Input contours can be intersecting, self-intersecting, or degenerate. Also, polygons with multiple coincident vertices are supported.

5.1 The Tessellation Object

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

```
GLUtesselator *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. Several tessellator objects can be used simultaneously.

When a tessellation object is no longer needed, it should be deleted with **gluDeleteTess**:

```
void gluDeleteTess( GLUtesselator *tessobj );
```

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

5.2 Polygon Definition

The input contours are specified with the following routines:

void **gluTessBeginPolygon**(GLUtesselator **tess*,
  void **polygon_data*);
void **gluTessBeginContour**(GLUtesselator **tess*);
void **gluTessVertex**(GLUtesselator **tess*,
  GLdouble *coords[3]*, void **vertex_data*);
void **gluTessEndContour**(GLUtesselator **tess*);
void **gluTessEndPolygon**(GLUtesselator **tess*);

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).

polygon_data is a pointer to a user-defined data structure. If the appropriate callback(s) are specified (see section 5.3), then this pointer is returned to the callback function(s). Thus, it is a convenient way to store per-polygon information.

coords give the coordinates of the vertex in 3-space. For useful results, all vertices should lie in some plane, since the vertices are projected onto a plane before tessellation. *vertex_data* is a pointer to a user-defined vertex structure, which typically contains other vertex information such as color, texture coordinates, normal, etc. It is used to refer to the vertex during rendering.

When **gluTessEndPolygon** is called, the tessellation algorithm determines which regions are interior to the given contours, according to one of several "winding rules" described below. The interior regions are then tessellated, and the output is provided as callbacks.

gluTessBeginPolygon indicates the start of a polygon, and it must be called first. It is an error to call **gluTessBeginContour** outside of a **gluTessBeginPolygon** / **gluTessEndPolygon** pair; it is also an error to call **gluTessVertex** outside of a **gluTessBeginContour** / **gluTessEndContour** pair. In addition, **gluTessBeginPolygon** / **gluTessEndPolygon** and **gluTessBeginContour** / **gluTessEndContour** calls must pair up.

5.3 Callbacks

Callbacks are specified with **gluTessCallback**:

void **gluTessCallback**(GLUtesselator **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 and becomes undefined. Any of the callbacks may be left undefined; if so, the corresponding information will not be supplied during rendering. (Note that, under some conditions, it is an error to leave the combine callback undefined. See the description of this callback below for details.)

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_TESS_BEGIN, GLU_TESS_EDGE_FLAG, GLU_TESS_VERTEX, GLU_TESS_END, GLU_TESS_ERROR, GLU_TESS_COMBINE, GLU_TESS_BEGIN_DATA, GLU_TESS_EDGE_FLAG_DATA, GLU_TESS_VERTEX_DATA, GLU_TESS_END_DATA, GLU_TESS_ERROR_DATA or GLU_TESS_COMBINE_DATA. The twelve callbacks have the following prototypes:

void **begin**(GLenum *type*);
void **edgeFlag**(GLboolean *flag*);
void **vertex**(void **vertex_data*);
void **end**(void);
void **error**(GLenum *errno*);
void **combine**(GLdouble *coords[3]*, void **vertex_data[4]*,
GLfloat *weight[4]*, void ***outData*);
void **beginData**(GLenum *type*, void **polygon_data*);
void **edgeFlagData**(GLboolean *flag*, void **polygon_data*);
void **endData**(void **polygon_data*);
void **vertexData**(void **vertex_data*, void **polygon_data*);
void **errorData**(GLenum *errno*, void **polygon_data*);
void **combineData**(GLdouble *coords[3]*,
void **vertex_data[4]*, GLfloat *weight[4]*, void ***outDatab*,
void **polygon_data*);

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 *polygon_data* will be used. Note that *polygon_data* is a copy of the pointer that was specified when **gluTessBeginPolygon** was called.

The **begin** callbacks indicate the start of a primitive. *type* is one of GL_TRIANGLE_FAN, GL_TRIANGLE_STRIP, or GL_TRIANGLES (but see the description of the edge flag callbacks below and the notes on *boundary extraction* in section 5.4 where the GLU_TESS_BOUNDARY_ONLY property is described).

It is followed by any number of **vertex** callbacks, which supply the vertices in the same order as expected by the corresponding **glBegin** call. *vertex_data* is a copy of the pointer that the user provided when the vertex was specified (see **gluTessVertex**). After the last vertex of a given primitive, the **end** or **endData** callback is called.

If one of the edge flag callbacks is provided, no triangle fans or strips will be used. When **edgeFlag** or **edgeFlagData** is called, if *flag* is GL_TRUE, then each vertex which follows begins an edge which lies on the polygon boundary (i.e., an edge which separates an interior region from an exterior one). If *flag* is GL_FALSE, each vertex which follows begins an edge which lies in the polygon interior. The edge flag callback will be called before the first call to the vertex callback.

The **error** or **errorData** callback is invoked when an error is encountered. The *errno* 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, or GLU_TESS_NEED_COMBINE_CALLBACK.

The first four errors are self-explanatory. The GLU library will recover from these errors by inserting the missing call(s). GLU_TESS_COORD_TOO_LARGE says that some vertex coordinate exceeded the predefined constant GLU_TESS_MAX_COORD_TOO_LARGE 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 says that the algorithm detected an intersection between two edges in the input data, and the *combine* callback (below) was not provided. No output will be generated.

The **combine** or **combineData** callback is invoked to create a new vertex when the algorithm detects an intersection, or wishes to merge features. The vertex is defined as a linear combination of up to 4 existing vertices, referenced by *vertex_data*[0..3]. The coefficients of the linear combination are given by *weight*[0..3]; these weights always sum to 1.0. All vertex pointers are valid even when some of the weights are zero. *coords* gives the location of the new vertex.

The user must allocate another vertex, interpolate parameters using *vertex_data* and *weights*, and return the new vertex pointer in *outData*. This handle is supplied during rendering callbacks. For example, if the polygon lies in an arbitrary plane in 3-space, and we associate a color with each vertex, the *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 algorithm detects an intersection, then the **combine** or **combineData** callback 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. This is the only error that can occur during tessellation and rendering.

5.4 Control Over Tessellation

The properties associated with a tessellator object affect the way the polygons are interpreted and rendered. The properties are set by calling:

void **gluTessProperty**(GLUtesselator *tess*, GLenum *which*, GLdouble *value*);

which indicates the property to be modified and must be set to one of GLU_TESS_WINDING_RULE, GLU_TESS_BOUNDARY_ONLY, or GLU_TESS_TOLERANCE.

value specifies the new property

The GLU_TESS_WINDING_RULE property determines which parts of the polygon are on the *interior.* It is an enumerated value; the possible values are: GLU_TESS_WINDING_ODD, GLU_TESS_WINDING_NONZERO, GLU_TESS_WINDING_NEGATIVE, GLU_TESS_WINDING_POSITIVE and GLU_TESS_WINDING_ABS_GEQ_TWO.

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

For a single contour C, the winding number of a point $\mathbf{x}$ is simply the signed number of revolutions we make around $\mathbf{x}$ as we travel once around C, where counter-clockwise (CCW) is positive. When there are several contours, the individual winding numbers are summed. This procedure associates a signed integer value with each point $\mathbf{x}$ 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 (see below).

The GLU_TESS_BOUNDARY_ONLY property 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 clockwise (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 which 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 default tolerance is zero.

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.

Property values can also be queried by calling

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

to load *value* with the value of the property specified by *which*.

To supply the polygon normal call:

```
void gluTessNormal( GLUtesselator tess, GLdouble x,
   GLdouble y, GLdouble z );
```

All input data will be projected into a plane perpendicular to the normal 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) (the default 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 plane; otherwise projection perpendicular to the computed normal 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 non-negative (where a CCW contour has positive area).

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

5.5 CSG Operations

The features of the tessellator make it easy to find the union, difference, or intersection of several polygons.

First, assume that each polygon is defined so that the winding number is 0 for each exterior region, and 1 for each interior region. Under this model, CCW contours define the outer boundary of the polygon, and CW contours define holes. Contours may be nested, but a nested contour must be oriented oppositely from the contour that contains it.

If the original polygons do not satisfy this description, they can be converted to this form by first running the tessellator with the `GLU_TESS_BOUNDARY_ONLY` property turned on. This returns a list of contours satisfying the restriction above. By allocating two tessellator objects, the callbacks from one tessellator can be fed directly to the input of another.

Given two or more polygons of the form above, CSG operations can be implemented as follows:

5.5.1 UNION

Draw all the input contours as a single polygon. The winding number of each resulting region is the number of original polygons which cover it. The union can be extracted using the `GLU_TESS_WINDING_NONZERO` or `GLU_TESS_WINDING_POSITIVE` winding rules. Note that with the nonzero rule, we would get the same result if all contour orientations were reversed.

5.5.2 INTERSECTION (two polygons at a time only)

Draw a single polygon using the contours from both input polygons. Extract the result using `GLU_TESS_WINDING_ABS_GEQ_TWO`. (Since this winding rule looks at the absolute value, reversing all contour orientations does not change the result.)

5.5.3 DIFFERENCE

Suppose we want to compute $A - (B \cup C \cup D)$. Draw a single polygon consisting of the unmodified contours from A, followed by the contours of B, C, and D with the vertex order reversed (this changes the winding number of the interior regions to -1). To extract the result, use the `GLU_TESS_WINDING_POSITIVE` rule.

If B, C, and D are the result of a `GLU_TESS_BOUNDARY_ONLY` call, an alternative to reversing the vertex order is to reverse the sign of the supplied normal. For example in the x-y plane, call **gluTessNormal**(*tess*, 0, 0, -1).

5.6 Performance

The tessellator is not intended for immediate-mode rendering; when possible the output should be cached in a user structure or display list. General polygon tessellation is an inherently difficult problem, especially given the goal of extreme robustness.

Single-contour input polygons are first tested to see whether they can be rendered as a triangle fan with respect to the first vertex (to avoid running the full decomposition algorithm on convex polygons). Non-convex polygons may be rendered by this "fast path" as well, if the algorithm gets lucky in its choice of a starting vertex.

For best performance follow these guidelines:

- supply the polygon normal, if available, using **gluTessNormal**. For example, if all polygons lie in the x-y plane, use **gluTessNormal**(*tess*, 0, 0, 1).

- render many polygons using the same tessellator object, rather than allocating a new tessellator for each one. (In a multi-threaded, multiprocessor environment you may get better performance using several tessellators.)

5.7 Backwards Compatibility

The polygon tessellation routines described previously are new in version 1.2 of the GLU library. For backwards compatibility, earlier versions of these routines are still supported:

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

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

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

gluBeginPolygon indicates the start of the polygon and **gluEndPolygon** defines the end of the polygon. **gluNextContour** is called once before each contour; however it does not need to be called when specifying a polygon with one contour. *type* is ignored by the GLU tessellator. *type* is one of `GLU_EXTERIOR`, `GLU_INTERIOR`, `GLU_CCW`, `GLU_CW` or `GLU_UNKNOWN`.

Calls to **gluBeginPolygon**, **gluNextContour** and **gluEndPolygon** are mapped to the new tessellator interface as follows:

gluBeginPolygon →	**gluTessBeginPolygon**
	gluTessBeginContour
gluNextContour →	**gluTessEndContour**
	gluTessBeginContour
gluEndPolygon →	**gluTessEndContour**
	gluTessEndPolygon

Constants and data structures used in the previous versions of the tesselator are also still supported. `GLU_BEGIN`, `GLU_VERTEX`, `GLU_END`, `GLU_ERROR` and `GLU_EDGE_FLAG` are defined as synonyms for `GLU_TESS_BEGIN`, `GLU_TESS_VERTEX`, `GLU_TESS_END`, `GLU_TESS_ERROR` and `GLU_TESS_EDGE_FLAG`. `GLUtriangulatorObj` is defined to be the same as `GLUtesselator`.

The preferred interface for polygon tessellation is the one described in sections 5.1-5.4. The routines described in this section are provided for backward compatibility only.

Chapter 6

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.

6.1 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.

6.2 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.

6.3 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`.

6.4 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 (-*outerRadius*,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.

Chapter 7

NURBS

NURBS curves and surfaces are converted to OpenGL primitives 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.

There are two ways to handle a NURBS object (curve or surface), to either render or to tessellate. In rendering mode, the objects are converted or tessellated to a sequence of OpenGL evaluators and sent to the OpenGL pipeline for rendering. In tessellation mode, objects are converted to a sequence of triangles and triangle strips and returned back to the application through a callback interface for further processing. The decomposition algorithm used for rendering and for returning tessellations are not guaranteed to produce identical results.

7.1 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.

7.2 Callbacks

To define a callback for a NURBS object, use:

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

The parameter *which* can be one of the following: GLU_NURBS_BEGIN, GLU_NURBS_VERTEX, GLU_NORMAL, GLU_NURBS_COLOR, GLU_NURBS_TEXTURE_COORD, GLU_END, GLU_NURBS_BEGIN_DATA, GLU_NURBS_VERTEX_DATA, GLU_NORMAL_DATA, GLU_NURBS_COLOR_DATA, GLU_NURBS_TEXTURE_COORD_DATA, GLU_END_DATA and GLU_ERROR.

These callbacks have the following prototypes:

```
void begin( GLenum type );
void vertex( GLfloat *vertex );
void normal( GLfloat *normal );
void color( GLfloat *color );
void texCoord( GLfloat *tex_coord );
void end( void );
void beginData( GLenum type, void *userData );
void vertexData( GLfloat *vertex, void *userData );
void normalData( GLfloat *normal, void *userData );
void colorData( GLfloat *color, void *userData );
void texCoordData( GLfloat *tex_coord, void *userData );
void endData( void *userData );
void error( GLenum errno );
```

The first 12 callbacks are for the user to get the primitives back from the NURBS tessellator when NURBS property GLU_NURBS_MODE is set to GLU_NURBS_TESSELLATOR (see section 7.6). These callbacks have no effect when GLU_NURBS_MODE is GLU_NURBS_RENDERER.

There are two forms of each callback: one with a pointer to application supplied data and one without. If both versions of a particular callback are specified then the callback with application data will be used. *userData* is specified by calling

`void` **gluNurbsCallbackData**(`GLUnurbsObj` **nurbsObj*,
`void` **userData*);

The value of *userData* passed to callback functions for a specific NURBS object is the value specified by the last call to **gluNurbsCallbackData**.

All callback functions can be set to `NULL` even when `GLU_NURBS_MODE` is set to `GLU_NURBS_TESSELLATOR`. When a callback function is set to `NULL`, this callback function will not get invoked and the related data, if any, will be lost.

The **begin** callback indicates the start of a primitive. *type* is 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 **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 **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 **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 **texture** callback is invoked as the texture coordinates of a vertex are generated. These coordinates are stored in the parameter *tex_coord*. 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 **end** callback is invoked at the end of a primitive. The default end callback function is `NULL`.

The **error** callback is invoked when a NURBS function detects an error condition. 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**.

7.3 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 monotonically increasing 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.

7.4 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 monotonically increasing 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 $(sknot\_count - sorder) \times (tknot\_count - torder)$. 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**.

7.5 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**).

7.6 NURBS Properties

A set of properties associated with a NURBS object affects the way that NURBS are rendered or tessellated. 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_SAMPLING_METHOD`, `GLU_PARAMETRIC_TOLERANCE`, `GLU_DISPLAY_MODE`, `GLU_AUTO_LOAD_MATRIX`, `GLU_U_STEP`, `GLU_V_STEP` and `GLU_NURBS_MODE`. *property* indicates the property to be modified, and *value* specifies the new value.

`GLU_NURBS_MODE` should be set to either `GLU_NURBS_RENDERER` or `GLU_NURBS_TESSELLATOR`. When set to `GLU_NURBS_RENDERER`, NURBS objects are tessellated into OpenGL evaluators and sent to the pipeline for rendering. When set to `GLU_NURBS_TESSELLATOR`, NURBS objects are tessellated into a sequence of primitives such as lines, triangles and triangle strips, but the vertices, normals, colors, and/or textures are retrieved back through a callback interface as specified in Section 7.2. This allows the user to cache the tessellated results for further processing. The default value is `GLU_NURBS_RENDERER`

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. The default is `GL_FALSE`.

`GLU_SAMPLING_METHOD` specifies how a NURBS surface should be tessellated. *value* may be set to one of `GLU_PATH_LENGTH`, `GLU_PARAMETRIC_ERROR`, `GLU_DOMAIN_DISTANCE`, `GLU_OBJECT_PATH_LENGTH` or `GLU_OBJECT_PARAMETRIC_ERROR`. When set to `GLU_PATH_LENGTH`, the surface is rendered so that the maximum length, in pixels, of the edges of the tessellation polygons is no greater than what is specified by `GLU_SAMPLING_TOLERANCE`. `GLU_PARAMETRIC_ERROR` specifies that the surface is rendered in such a way that the value specified by `GLU_PARAMETRIC_TOLERANCE` describes the maximum distance, in pixels, between the tessellation polygons and the surfaces they approximate. `GLU_DOMAIN_DISTANCE` allows users to specify, in parametric coordinates, how many sample points per unit length are taken in u, v dimension. `GLU_OBJECT_PATH_LENGTH` is similar to `GLU_PATH_LENGTH` except that it is view independent; that is, it specifies that the surface is rendered so that the maximum length, in object space, of edges of the tessellation polygons is no greater than what is specified by `GLU_SAMPLING_TOLERANCE`. `GLU_OBJECT_PARAMETRIC_ERROR` is similar to `GLU_PARAMETRIC_ERROR` except that 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 default value of `GLU_SAMPLING_METHOD` is `GLU_PATH_LENGTH`.

`GLU_SAMPLING_TOLERANCE` specifies the maximum length, in pixels or in object space length unit, to use when the sampling method is set to

GLU_PATH_LENGTH or GLU_OBJECT_PATH_LENGTH. The default value is 50.0.

GLU_PARAMETRIC_TOLERANCE specifies the maximum distance, in pixels or in object space length unit, to use when the sampling method is set to GLU_PARAMETRIC_ERROR or GLU_OBJECT_PARAMETRIC_ERROR. The default value for GLU_PARAMETRIC_TOLERANCE is 0.5.

GLU_U_STEP specifies the number of sample points per unit length taken along the u dimension in parametric coordinates. It is needed when GLU_SAMPLING_METHOD is set to GLU_DOMAIN_DISTANCE. The default value is 100.

GLU_V_STEP specifies the number of sample points per unit length taken along the v dimension in parametric coordinates. It is needed when GLU_SAMPLING_METHOD is set to GLU_DOMAIN_DISTANCE. The default value is 100.

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.

You may get unexpected results when GLU_AUTO_LOAD_MATRIX is enabled and the results of the NURBS tesselation are being stored in a display list, since the OpenGL matrices which are used to create the sampling and culling matrices will be those that are in effect when the list is created, not those in effect when it is executed.

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

Property values can be queried by calling

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

The specified *property* is returned in *value*.

Chapter 8

Errors

Calling

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

produces an error string corresponding 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`, `GLU_INVALID_OPERATION` and `GLU_OUT_OF_MEMORY`. There are also specific error codes for polygon tessellation, quadrics, and NURBS as described in their respective sections.

If an invalid call to the underlying OpenGL implementation is made by GLU, either GLU or OpenGL errors may be generated, depending on where the error is detected. This condition may occur only when making a GLU call introduced in a later version of GLU than that corresponding to the OpenGL implementation (see Chapter 9); for example, calling **gluBuild3DMipmaps** or passing packed pixel types to **gluScaleImage** when the underlying OpenGL version is earlier than 1.2.

Chapter 9

GLU Versions

Each version of GLU corresponds to the OpenGL version shown in Table 9.1; GLU features introduced in a particular version of GLU may not be usable if the underlying OpenGL implementation is an earlier version.

All versions of GLU are upward compatible with earlier versions, meaning that any program that runs with the earlier implementation will run unchanged with any later GLU implementation.

9.1 GLU 1.1

In GLU 1.1, **gluGetString** was added allowing the GLU version number and GLU extensions to be queried. Also, the NURBS properties `GLU_SAMPLING_METHOD`, `GLU_PARAMETRIC_TOLERANCE`, `GLU_U_STEP` and `GLU_V_STEP` were added providing support for different tesselation methods. In GLU 1.0, the only sampling method supported was `GLU_PATH_LENGTH`.

GLU Version	Corresponding OpenGL Version
GLU 1.0	OpenGL 1.0
GLU 1.1	OpenGL 1.0
GLU 1.2	OpenGL 1.1
GLU 1.3	OpenGL 1.2

Table 9.1: Relationship of OpenGL and GLU versions.

9.2 GLU 1.2

A new polygon tesselation interface was added in GLU 1.2. See section 5.7 for more information on the API changes.

A new NURBS callback interface and object space sampling methods was also added in GLU 1.2. See sections 7.2 and 7.6 for API changes.

9.3 GLU 1.3

The **gluCheckExtension** utility function was introduced.

gluScaleImage and **gluBuild*x*DMipmaps** support the new packed pixel formats and types introduced by OpenGL 1.2.

gluBuild3DMipmaps was added to support 3D textures, introduced by OpenGL 1.2.

gluBuild*x*DMipmapLevels was added to support OpenGL 1.2's ability to load only a subset of mipmap levels.

gluUnproject4 was added for use when non-default depth range or w values other than 1 need to be specified.

New **gluNurbsCallback** callbacks and the `GLU_NURBS_MODE` NURBS property were introduced to allow applications to capture NURBS tesselations. These features exactly match corresponding features of the `GLU_EXT_nurbs_tessellator` GLU extension, and may be used interchangeably with the extension.

New values of the `GLU_SAMPLING_METHOD` NURBS property were introduced to support object-space sampling criteria. These features exactly match corresponding features of the `GLU_EXT_object_space_tess` GLU extension, and may be used interchangeably with the extension.

Index of GLU Commands