Appendix C

The OpenGL Utility Library

OpenGL provides a powerful but small set of drawing operations, and all higher−level drawing must be done in terms of these. To help simplify some of your programming tasks, the OpenGL Utility Library (GLU) includes several routines that encapsulate OpenGL commands. Many of these routines are described in earlier chapters as their topics arise; these routines are briefly listed here for completeness. GLU routines that aren't discussed earlier are described in more depth here. Nevertheless, you might want to consult the *OpenGL Reference Manual* for more detailed descriptions of all these routines. This appendix groups the GLU routines functionally as follows:

- **"Manipulating Images for Use in Texturing"**
- **"Transforming Coordinates"**
- **"Polygon Tessellation"**
- **"Rendering Spheres, Cylinders, and Disks"**
- **"NURBS Curves and Surfaces"**
- **"Describing Errors"**

Manipulating Images for Use in Texturing

As you set up texture mapping in your application, you'll probably want to take advantage of mipmapping, which requires a series of reduced images (or texture maps). To support mipmapping, the GLU includes a general routine that scales images (*gluScaleImage()*) and routines that generate a complete set of mipmaps given an original image in one or two dimensions (*gluBuild1DMipmaps()* and *gluBuild2DMipmaps()*). These routines are all discussed in some detail in **Chapter 9**, so here only their prototypes are listed:

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

GLint *gluBuild1DMipmaps* (GLenum *target*, GLint *components*, GLint *width*, GLenum *format*, GLenum *type*, void * *data*);

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

Transforming Coordinates

The GLU includes routines that create matrices for standard perspective and orthographic viewing (*gluPerspective()* and *gluOrtho2D()*). In addition, a viewing routine allows you to place your eye at any point in space and look at any other point (*gluLookAt()*). These routines are discussed in **Chapter 3**. In addition, the GLU includes a routine to help you create a picking matrix (*gluPickMatrix()*); this routine is discussed in **Chapter 12**. For your convenience, the prototypes for these four routines are listed here.

void *gluPerspective* (GLdouble *fovy*, GLdouble *aspect*, GLdouble *zNear*, GLdouble *zFar*);

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

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

void *gluPickMatrix* (GLdouble *x*, GLdouble *y*, GLdouble *width*, GLdouble *height*, GLint *viewport* [4]);

In addition, GLU provides two routines that convert between object coordinates and screen coordinates, *gluProject()* and *gluUnProject()*.

GLint *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*);

Transforms the specified object coordinates *objx*, *objy*, and *objz* into window coordinates using *modelMatrix* , *projMatrix* , and *viewport* . The result is stored in *winx* , *winy* , and *winz* . A return value of GL_TRUE indicates success, and GL_FALSE indicates failure.

GLint *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*);

Transforms the specified window coordinates *winx* , *winy* , and *winz* into object coordinates using *modelMatrix* , *projMatrix* , and *viewport* . The result is stored in *objx*, *objy*, and *objz*. A return value of GL_TRUE indicates success, and GL_FALSE indicates failure.

Polygon Tessellation

As discussed in **"Describing Points, Lines, and Polygons,"** OpenGL can directly display only simple convex polygons. A polygon is simple if the edges intersect only at vertices, there are no duplicate vertices, and exactly two edges meet at any vertex. If your application requires the display of simple nonconvex polygons or of simple polygons containing holes, those polygons must first be subdivided into convex polygons before they can be displayed. Such subdivision is called tessellation . GLU provides a collection of routines that perform tessellation. Note that the GLU tessellation routines can't handle nonsimple polygons; there's no standard OpenGL method to handle such polygons.

Since tessellation is often required and can be rather tricky, this section describes the GLU tessellation routines in detail. These routines take as input arbitrary simple polygons that might include holes, and they return some combination of triangles, triangle meshes, and triangle fans. You can insist on only triangles if you don't want to have to deal with meshes or fans. If you care about performance, however, you should probably take advantage of any available mesh or fan information.

The Callback Mechanism

To tessellate a polygon using the GLU, first you need to create a tessellation object, and then provide a series of callback routines to be called at appropriate times during the tessellation. After you specify the callbacks, you describe the polygon and any holes using GLU routines, which are similar to the OpenGL polygon routines. When the polygon description is complete, the tessellation facility invokes your callback routines as necessary.

The callback routines typically save the data for the triangles, triangle meshes, and triangle fans in user−defined data structures, or in OpenGL display lists (see **Chapter 4**). To render the polygons, other code traverses the data structures or calls the display lists. Although the callback routines could call OpenGL commands to display them directly, this is usually not done, as tessellation can be computationally expensive. It's a good idea to save the data if there is any chance that you want to display it again. The GLU tessellation routines are guaranteed never to return any new vertices, so interpolation of vertices, texture coordinates, or colors is never required.

The Tessellation Object

As a complex polygon is being described and tessellated, it has associated data, such as the vertices, edges, and callback functions. All this data is tied to a single tessellation object. To do tessellation, your program first has to create a tessellation object using the routine *gluNewTess()* .

GLUtriangulatorObj* *gluNewTess* (void);

Creates a new tessellation object and returns a pointer to it. A null pointer is returned if the creation fails.

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

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

Deletes the specified tessellation object, *tessobj* , and frees all associated memory.

A single tessellation object can be reused for all your tessellations. This object is required only because library routines might need to do their own tessellations, and they should be able to do so without interfering with any tessellation that your program is doing. It might also be useful to have multiple tessellation objects if you want to use different sets of callbacks for different tessellations. A typical program, however, allocates a single tessellation object and uses it for all its tessellations. There's no real need to free it because it uses a small amount of memory. On the other hand, if you're writing a library routine that uses the GLU tessellation, you'll want to be careful to free any tessellation objects you create.

Specifying Callbacks

You can specify up to five callback functions for a tessellation. Any functions that are omitted are simply not called during the tessellation, and any information they might have returned to your program is lost. All are specified by the single routine *gluTessCallback()* .

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

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

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

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

As tessellation proceeds, these routines are called in a manner similar to the way you would use the OpenGL commands *glBegin()* , *glEdgeFlag*()* , *glVertex*()* , and *glEnd()* . (See **"Marking Polygon Boundary Edges" in Chapter 2** for more information about *glEdgeFlag*()* .) The error callback is invoked during the tessellation only if something goes wrong.

The GLU_BEGIN callback is invoked with one of three possible parameters: GL_TRIANGLE_FAN, GL_TRIANGLE_STRIP, or GL_TRIANGLES. After this routine is called, and before the callback associated with GLU_END is called, some combination of the GLU_EDGE_FLAG and GLU_VERTEX callbacks is invoked. The associated vertices and edge flags are interpreted exactly as they are in OpenGL between *glBegin* (GL_TRIANGLE_FAN), *glBegin* (GL_TRIANGLE_STRIP), or *glBegin* (GL_TRIANGLES) and the matching *glEnd()* . Since edge flags make no sense in a triangle fan or triangle strip, if there is a callback associated with GLU_EDGE_FLAG, the GLU_BEGIN callback is called only with GL_TRIANGLES. The GLU_EDGE_FLAG callback works exactly analogously to the OpenGL *glEdgeFlag*()* call.

The error callback is passed a GLU error number. A character string describing the error can be obtained using the routine *gluErrorString()* . See **"Describing Errors"** for more information about this routine.

Describing the Polygon to Be Tessellated

The polygon to be tessellated, possibly containing holes, is specified using the following four routines: *gluBeginPolygon()*, *gluTessVertex()*, *gluNextContour()*, and *gluEndPolygon()*. For polygons without holes, the specification is exactly as in OpenGL: start with *gluBeginPolygon()*, call *gluTessVertex()* for each vertex in the boundary, and end the polygon with a call to *gluEndPolygon()*. If a polygon consists of multiple contours, including holes and holes within holes, the contours are specified one after the other, each preceded by *gluNextContour()*. When *gluEndPolygon()* is called, it signals the end of the final contour and starts the tessellation. You can omit the call to *gluNextContour()* before the first contour. The detailed descriptions of these functions follow.

---

void *gluBeginPolygon* (GLUtriangulatorObj * *tessobj*);

Begins the specification of a polygon to be tessellated and associates a tessellation object, *tessobj*, with it. The callback functions to be used are those that were bound to the tessellation object using the routine *gluTessCallback()*.

---

void *gluTessVertex* (GLUtriangulatorObj * *tessobj* ,GLdouble *v*[3], void * *data*);

Specifies a vertex in the polygon to be tessellated. Call this routine for each vertex in the polygon to be tessellated. *tessobj* is the tessellation object to use, *v* contains the three−dimensional vertex coordinates, and *data* is an arbitrary pointer that's sent to the callback associated with GLU_VERTEX. Typically, it contains vertex data, texture coordinates, color information, or whatever else the application may find useful.

---

void *gluNextContour* (GLUtriangulatorObj * *tessobj* , GLenum *type*);

Marks the beginning of the next contour when multiple contours make up the boundary of the polygon to be tessellated. *type* can be GLU_EXTERIOR, GLU_INTERIOR, GLU_CCW, GLU_CW, or GLU_UNKNOWN. These serve only as hints to the tessellation. If you get them right, the tessellation might go faster. If you get them wrong, they're ignored, and the tesselation still works. For a polygon with holes, one contour is the exterior contour and the others interior. *gluNextContour()* can be called immediately after *gluBeginPolygon()*, but if it isn't, the first contour is assumed to be of type GLU_EXTERIOR. GLU_CW and GLU_CCW indicate clockwise− and counterclockwise− oriented polygons. Choosing which are clockwise and which are counterclockwise is arbitrary in three dimensions, but in any plane, there are two different orientations, and the GLU_CW and GLU_CCW types should be used consistently. Use GLU_UNKNOWN if you don't have a clue.

---

void *gluEndPolygon* (GLUtriangulatorObj * *tessobj*);

Indicates the end of the polygon specification and that the tessellation can begin using the tessellation object *tessobj*.

Rendering Spheres, Cylinders, and Disks

The GLU includes a set of routines to draw various simple surfaces (spheres, cylinders, disks, and parts of disks) in a variety of styles and orientations. These routines are described in detail in the *OpenGL Reference Manual*; their use is discussed briefly in the following paragraphs, and their prototypes are also listed.

To create a quadric object, use *gluNewQuadric()*. (To destroy this object when you're finished with it, use *gluDeleteQuadric()*.) Then specify the desired rendering style, as follows, with the appropriate routine (unless you're satisfied with the default values):

- Whether surface normals should be generated, and if so, whether there should be one normal per vertex or one normal per face: *gluQuadricNormals()*
- Whether texture coodinates should be generated: *gluQuadricTexture()*

- Which side of the quadric should be considered the outside and which the inside: *gluQuadricOrientation()*
- Whether the quadric should be drawn as a set of polygons, lines, or points: *gluQuadricDrawStyle()*

After you've specified the rendering style, simply invoke the rendering routine for the desired type of quadric object *: gluSphere()* , *gluCylinder()* , *gluDisk()* , or *gluPartialDisk()* . If an error occurs during rendering, the error−handling routine you've specified with *gluQuadricCallBack()* is invoked.

It's better to use the **Radius* , *height* , and similar arguments to scale the quadrics rather than the *glScale*()* command, so that unit−length normals that are generated don't have to be renormalized. Set the *loops* and *stacks* arguments to values other than 1 to force lighting calculations at a finer granularity, especially if the material specularity is high.

The prototypes are listed in three categories.

Manage quadric objects:
GLUquadricObj* *gluNewQuadric* (void);
void *gluDeleteQuadric* (GLUquadricObj * *state*);
void *gluQuadricCallback* (GLUquadricObj * *qobj*, GLenum *which* , void (**fn*)());

Control the rendering:
void *gluQuadricNormals* (GLUquadricObj * *quadObject* , GLenum *normals*);
void *gluQuadricTexture* (GLUquadricObj * *quadObject* , GLboolean *textureCoords*);
void *gluQuadricOrientation* (GLUquadricObj * *quadObject* , GLenum *orientation*);
void *gluQuadricDrawStyle* (GLUquadricObj * *quadObject* , GLenum *drawStyle*);

Specify a quadric primitive:
void *gluCylinder* (GLUquadricObj * *qobj*, GLdouble *baseRadius* , GLdouble *topRadius* , GLdouble *height* , GLint *slices*, GLint *stacks*);
void *gluDisk* (GLUquadricObj * *qobj*, GLdouble *innerRadius* , GLdouble *outerRadius* , GLint *slices*, GLint *loops*);
void *gluPartialDisk* (GLUquadricObj * *qobj*, GLdouble *innerRadius* , GLdouble *outerRadius* , GLint *slices* , GLint *loops*, GLdouble *startAngle* , GLdouble *sweepAngle*);
void *gluSphere* (GLUquadricObj * *qobj*, GLdouble *radius* , GLint *slices* ,GLint *stacks*);

NURBS Curves and Surfaces

NURBS routines provide general and powerful descriptions of curves and surfaces in two and three dimensions. They're used to represent geometry in many computer−aided mechanical design systems. The GLU NURBS routines can render such curves and surfaces in a variety of styles, and they can automatically handle adaptive subdivision that tessellates the domain into smaller triangles in regions of high curvature and near silhouette edges. All the GLU NURBS routines are described in **Chapter 9** ; their prototypes are listed here.

Manage a NURBS object:
GLUnurbsObj* *gluNewNurbsRenderer* (void);
void *gluDeleteNurbsRenderer* (GLUnurbsObj * *nobj*);
void *gluNurbsCallback* (GLUnurbsObj * *nobj*, GLenum *which* , void (**fn*)());

Create a NURBS curve:
void *gluBeginCurve* (GLUnurbsObj * *nobj*);
void *gluEndCurve* (GLUnurbsObj * *nobj*);
void *gluNurbsCurve* (GLUnurbsObj * *nobj*, GLint *nknots* , GLfloat * *knot* , GLint *stride* , GLfloat * *ctlarray* , GLint *order*, GLenum *type*);

Create a NURBS surface:
void *gluBeginSurface* (GLUnurbsObj * *nobj*);
void *gluEndSurface* (GLUnurbsObj * *nobj*);
void *gluNurbsSurface* (GLUnurbsObj * *nobj*, GLint *uknot_count* , GLfloat * *uknot* , GLint *vknot_count* ,

void *gluNurbsSurface* (GLUnurbsObj * *nobj*, GLint *uknot_count* , GLfloat * *uknot* , GLint *vknot_count* , GLfloat * *vknot* , GLint *u_stride* , GLint *v_stride* , GLfloat * *ctlarray* , GLint *uorder* , GLint *vorder* , GLenum *type*);

Define a trimming region:
void *gluBeginTrim* (GLUnurbsObj * *nobj*);
void *gluEndTrim* (GLUnurbsObj * *nobj*);
void *gluPwlCurve* (GLUnurbsObj * *nobj*, GLint *count* , GLfloat * *array* , GLint *stride* , GLenum *type*);

Control NURBS rendering:
void *gluLoadSamplingMatrices* (GLUnurbsObj * *nobj*, const GLfloat *modelMatrix* [16], const GLfloat *projMatrix* [16], const GLint *viewport* [4]);
void *gluNurbsProperty* (GLUnurbsObj * *nobj*, GLenum *property* , GLfloat *value*);
void *gluGetNurbsProperty* (GLUnurbsObj * *nobj*, GLenum *property* , GLfloat * *value*);

Describing Errors

The GLU provides a routine for obtaining a descriptive string for an error code. For information about OpenGL's error handling facility, see **"Error Handling."**

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

Returns a pointer to a descriptive string that corresponds to the OpenGL, GLU, or GLX error number passed in *errorCode* . The defined error codes are described in the *OpenGL Reference Manual* along with the command or routine that can generate them.