OpenGL Quick Reference Guide

The following is a brief summary of a number of functions for OpenGL (GL), the OpenGL utilities (GLU), and the OpenGL utility toolkit (GLUT). This is not a complete reference manual, but it should provide enough basic information to point you in the right direction for reference information. For detailed reference information on GL and GLU consult the *OpenGL Reference Manual* (2nd edition) and for GLUT consult *OpenGL: Programming for the X Window System*, by M. J. Kilgard. (Both are available on reserve in our department library in AVW 3164).

It is also possible to get information online through the web. The problem is that the information is not in the most easily accessible format. The following web sites are good sources of information on OpenGL, GLUT and GLU.

`http://www.cs.uwm.edu/~grafix1`
`http://www.opengl.org/Coding/Coding.html`
`http://www.opengl.org/About/FAQ/Technical.html`
`http://reality.sgi.com/mjk_asd/spec3/spec3.html`

The first web site has a tutorial software that covers most of the fundamental topics in openGL. Before reading the following sections, here are some general pieces of information which you should be aware of.

Generic Function Naming: Many OpenGL procedures come in a variety of forms, depending on the argument types. For example, `glVertex3f()` takes 3 `float` arguments, `glVertex4d()` takes 4 `double` arguments, and `glVertex3iv()` takes a single pointer to an array of 3 `int`'s. The naming conventions are quite regular. These procedures are lumped under the single name `glVertex*()` in the descriptions below. See the reference manual for the actual calling sequences.

GL Data Types: OpenGL defines a number of data types. Most of these translate directly to types in C or C++. For example, `GLdouble` is the same as `double`, and `GLuint` is an `unsigned int`. The type `GLClampf` is a `float` that has been "clamped" to the interval $[0, 1]$, meaning that values less than 0 are set to 0 and values greater than 1 are set to 1. See the file `Mesa/include/GL/gl.h` for exact type definitions.

RGBA: OpenGL colors are typically defined using RGB (red, green, blue) components and a special A (or alpha) component The A component has varying interpretations depending on context (typically in providing for transparency and color blending). When no special effects are desired, set $A = 1$.

Error Messages: OpenGL and Mesa are "quiet" about errors. When an error has been committed, the system does not display any error message. Rather it sets an error code, which the user can query through `glGetError()`. It is the programmer's responsibility to check the error code after each call to an OpenGL procedure. However, if you do not want to be troubled by this, set the environment variable `MESA_DEBUG`. This will cause Mesa to output a message whenever an error is detected (and produces a few annoying warning messages that you may not care about). This can be done by adding the command "`setenv MESA_DEBUG`" to your login initialization file.

By the way, Mesa always outputs a message about how many colors from the colormap it was able to allocate. For best results, kill any other colormap hogs (like netscape and xv) before running any Mesa program.

Initializations: The following procedures are typically invoked once in the main program, before the graphics processing begins.

`glutInit(int *argcp, char **argv):`
Initialize GLUT and OpenGL. Pass in command-line arguments.

`glutInitDisplayMode(unsigned int mode):`
Set GLUT's display mode. The mode is the logical-or "`|`" of one or more of the following:

Select one of the following three:

`GLUT_RGB`	use RGB colors
`GLUT_RGBA`	use RGBA colors
`GLUT_INDEX`	use color-mapped colors (not recommended)

Select one of the following two:

`GLUT_SINGLE`	use single-buffering
`GLUT_DOUBLE`	allow double-buffering (for smooth animation)

Select the following for hidden surface removal:

`GLUT_DEPTH`	allow depth-buffering

There are a number of other options, which we have omitted. This last option make depth-buffering possible. It is also necessary to enable the operation (see `glEnable()` below).

`glutInitWindowSize(int width, int height):`
  Set the window size. Also see: `glutInitWindowPosition()`.

`glutCreateWindow(char *title):`
  Create window with the given title (`argv[0]` sets the name to the program's name).

`glutMainLoop(void):`
  This starts the main event loop. Control returns only through one of the callback functions given below.

Callbacks: After calling `glutMainLoop`, control returns to your program only if some event occurs. The following routines specify the events that you wish to listen for, and which procedure of yours to call for each.

Warning: Unlike all OpenGL functions, in which (x, y) coordinates are made relative to an idealized window, the (x, y) mouse coordinates are given in viewport coordinates (i.e., pixels). Furthermore, the origin is in the upper left corner, so y values increase as you go down the window.

`glutDisplayFunc(void (*)(void)):`
  Call the given function to redisplay everything (required).

`glutReshapeFunc(void (*)(int width, int height)):`
  Call the given function when window is resized (and originally created). See the function `glViewport()`, which typically should be called after each resizing.

`glutIdleFunc(void (*)(void)):`
  Call the given function with each refresh cycle of the display.

`glutMouseFunc(void (*)(int button, int state, int x, int y)):`
  Call the given function when a mouse button is clicked. The *button* argument is any of `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON`, `GLUT_RIGHT_BUTTON`, and the *state* is either `GLUT_DOWN` or `GLUT_UP`. The (x, y) coordinates give the current mouse position, relative to the upper left corner.

`glutMotionFunc(void (*)(int x, int y)):`
  Call the given function when the mouse is dragged (with button held down). The arguments (x, y) give the mouse coordinates relative to the upper left corner.

`glutPassiveMotionFunc(void (*)(int x, int y))`:
Call the given function when the mouse is moved.

`glutKeyboardFunc(void (*)(unsigned char key, int x, int y))`:
Call the given function when a keyboard key is hit. (This only handles ASCII characters. There is a different callback for function and arrow keys.)

Miscellaneous Display Control Functions: Here are some miscellaneous routines that affect how drawing and displaying take place.

`glutPostRedisplay(void)`:
Request that the image be redrawn.

`glutSwapBuffers(void)`:
Swap the front and back buffers (used in double buffering).

`glClearColor(GLclampf R, GLclampf G, GLclampf B, GLclampf A)`:
Specifies the background color to be used by `glClear()`, when clearing the buffer.

`glClear(GLbitfield mask)`:
Clears one or more of the existing buffers. The mask is a logical or ("`|`") of any of the following: `GL_COLOR_BUFFER_BIT`, `GL_DEPTH_BUFFER_BIT`, `GL_ACCUM_BUFFER_` or `GL_STENCIL_BUFFER_BIT`.

`glFlush(void)`:
OpenGL normally saves or "buffers" drawing commands for greater efficiency. This forces the image to be redrawn, in case the next update will be far in the future.

`glFinish(void)`:
This is like `glFlush()`, but it waits until all drawing has been finished before it returns.

`glHint(GLenum target, GLenum mode)`:
Provides a hint as to how OpenGL should behave. Normally this is to suggest whether speed or accuracy is more important. The *mode* may be `GL_DONT_CARE`, `GL_FASTEST`, or `GL_NICEST`. Some *target* values include

`GL_FOG_HINT`	accuracy of fog computation
`GL_PERSPECTIVE_CORRECTION_HINT`	accuracy of interpolation

GL_POINT(/LINE/POLYGON)_SMOOTH_HINT	in texture mapping accuracy of sampling in antialiasing operations

OpenGL Basic Drawing: With the exception of `glRect()`, most drawing is done by creating geometric objects using `glBegin()...glEnd()` combination, where between each such pair you specify a collection of vertices that define the object. Along with specifying vertices, you can specify a number of attributes that affect how the vertices will appear. These include things such as color, surface normals (for shading), and texture coordinates (for texture mapping).

`glRect*(...)`:
Draw a rectangle on the $z = 0$ plane. The arguments give the (x, y) coordinates of any two opposite corners of the rectangle. Some forms include `glRectd()`, which takes four `double`'s, `glRectf()`, which takes four `float`'s, `glRectfv()` takes two vectors, each consisting of two `float`'s.

`glBegin(GLenum mode)...glEnd(void)`:
Define an object, where *mode* may be any of:

GL_POINTS	set of points
GL_LINES	set of line segments (not connected)
GL_LINE_STRIP	chain of connected line segments
GL_LINE_LOOP	closed polygon (may self intersect)
GL_TRIANGLES	set of triangles (not connected)
GL_TRIANGLE_STRIP	linear chain of triangles
GL_TRIANGLE_FAN	fan of triangles (joined to one point)
GL_QUADS	set of quadrangles (not connected)
GL_QUAD_STRIP	linear chain of quadrangles
GL_POLYGON	a convex polygon

There is a limited set of commands that can be placed within `glBegin()..-glEnd()` pairs. These include `glVertex*()`, `glColor*()`, `glNormal*()`, `glTexCoord*()`, `glMaterial*()`, and `glCallList()`. See the reference manual for the exact specification of vertex order and orientation.

`glVertex*(...)`:
Specify the coordinates of a vertex.

`glNormal*(...)`:
: Specify the surface normal for subsequent vertices. The normal should be of unit length after the modelview transformation is applied. Call `glEnable(GL_NORMALIZE)` to have OpenGL do normalization automatically.

`glColor*(...)`:
: Specify the color of subsequent vertices. This is normally used if lighting is *not* enabled. Otherwise, colors are defined by `glMaterial*()`, defined below under lighting.

`glLineWidth(GLfloat width)`:
: Sets the line width for subsequent line drawing. The argument is the width of the line (in pixels). (Note that not all widths are necessarily supported.)

`glPolygonMode(GLenum face, GLenum mode)`:
: Controls the drawing mode for a polygon's front and back faces. The parameter *face* can be `GL_FRONT_AND_BACK`, `GL_FRONT`, or `GL_BACK`; *mode* can be `GL_POINT`, `GL_LINE`, or `GL_FILL` to indicate whether the polygon should be drawn as points, outlined, or filled.

GLUT Utilities for Complex Objects: GLUT provides a number of procedures for drawing complex objects.

`glRasterPos*(...)`: Set the current raster position (in 3D window coordinates) for subsequent `glutBitmapCharacter()` and other pixel copying operations.

`glutBitmapCharacter(void *font, int character)`:
: Draw the given *character* of the given *font* at the current raster position and advance the current raster position. Fonts include `GLUT_BITMAP_9_BY_15`, `GLUT_BITMAP_8_BY_13`, `GLUT_BITMAP_TIMES_ROMAN_10`, `GLUT_BITMAP_TIMES_ROMAN_24`. When a character is drawn, the raster position is advanced. So to draw a string, initialize the raster position (using `glRasterPos*()`) and then call this on each character of the string.

`glutSolidSphere(GLdouble radius, GLint slices, GLint stacks)`:
: Draw a solid sphere of the given *radius*, subdivided into *slices* longitudinal strips, and *stacks* latitudinal strips. Normal vectors are also defined for proper illumination. Other shapes include
`glutSolidCone()`

```
glutSolidCube()
glutSolidTorus()
glutSolidDodecahedron()
glutSolidTeapot()
glutSolidOctahedron()
glutSolidTetrahedron()
glutSolidIcosahedron()
```

All of these also have wire-frame versions that draw only the edges of the shape.

GLUT Menu Functions: GLUT provides a simple mechanism for creating pop-up menus. Each menu is associated with a procedure (which you provide) when you create the menu. When the menu item is selected, this procedure is called and passed the index of the item selected. Submenus are also possible (but not discussed here).

`glutCreateMenu(void (*)(int)):`
: Creates an empty menu. The argument is the callback function, which is of the form `void myfunc(int value)`, where *value* holds the index of the menu item which was selected.

`glutAddMenuEntry(char *name, int value):`
: Adds a menu entry to the bottom of the current menu. The character string *name* is the text to be displayed in the menu entry, and the integer *value* is passed to your callback procedure to identify the selected item.

`glutAttachMenu(int button):`
: Attaches the current menu to the specified mouse button, which is either `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON`, or `GLUT_RIGHT_BUTTON`.

Display lists: Because of its structure, OpenGL requires lots of procedure calls to render a complex object. To improve efficiency, OpenGL allows you to generate an object, called a *display list*, into which OpenGL commands are stored in an efficient internal format, which can be called back again in the future.

`GLuint glGenLists(GLsizei range):`
: Allocates *range* new display lists. It returns a number n, where the new lists are numbered $n, n+1, \ldots, n + \mathit{range} - 1$.

`glNewList(GLuint list, GLenum mode)...glEndList(void)`:
: All OpenGL commands between such a pair are stored in display list *list*. The mode is either `GL_COMPILE`, to compile the list, or `GL_COMPILE_AND_EXECUTE` to both compile and execute. Note that some OpenGL commands (mostly those having to do with accessing the current state) are executed immediately and not compiled. See the reference manual for details.

`glCallList(GLuint list)`:
: Invokes the given display list. (Note that the system does not make any effort to save its current state, thus the display list should make use of functions such as `glPushMatrix()` and `glPopMatrix()` to save and restore the current state.)

Transformations and Perspective: OpenGL provides three different matrices: *modelview*, *projection*, and *texture* (used in texture mapping). Actually, these matrices are maintained in three stacks. This allows you to push (save) the existing matrix on the stack, apply some new local transformations, draw some vertices under this transformation, and then pop the stack thus restoring the existing transformation. At any time one of the three matrices is active and is set by `glMatrixMode()`. Note that the transformation that has been multiplied last is the one that is applied first to the object vertices.

`glViewport(GLint x, GLint y, GLsizei width, GLsizei height)`:
: Sets the current viewport, that is, the portion of the window to which everything will be clipped. Typically this is the entire window: `glViewport(0, 0, win_width, win_height)`, and hence this should be called whenever the window is reshaped.

`glMatrixMode(GLenum mode)`:
: Set the current matrix mode to one of `GL_MODELVIEW`, `GL_PROJECTION`, or `GL_TEXTURE`. The default is `GL_MODELVIEW`.

`glLoadIdentity(void)`: Set the current matrix to the identity.

`glPushMatrix(void)`: Make a copy of the current matrix and push it onto the stack.

`glPopMatrix(void)`: Pop the top of the current matrix.

`glLoadMatrixf(const GLfloat *M)`: Sets the current matrix to M.

`glMultMatrixf(const GLfloat *M)`: Composes the current matrix with M.

Note: The matrices in the last two calls are stored as 4×4 homogeneous matrices in column-major order. Thus, each matrix is an array of 16 elements. The first 4 elements are the image of the x-unit vector, the second 4 are the image of the y-unit vector and so on.

(Typically) Modelview Transformations: These are typically applied in *modelview* mode. They are useful for positioning objects to make them easier to draw. The following procedures generate a transformation matrix for the given task and then compose with the current transforamation matrix. They are composed so that the most recently composed transformation is applied first to the object vertices. Except for `gluLookAt()`, they all have two forms, either ending in "d" (for `double` arguments) or "f" (for `float` arguments). Only the floating forms are given here.

`gluLookAt(eyex, eyey, eyez, centerx, centery, centerz, upx, upy, upz):`

All arguments are of type `double`. This generates a transformation which maps world coordinates into camera coordinates. This is typically called before any drawing is done (just after `glLoadIdentity()`). The camera position is specified by the eye location, looking at the given center point, with the given up-directional vector. It composes this with the current (typically modelview) transformation.

`glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z):`
Generate a rotation transformation about the specified vector (x, y, z) and compose it with the current transformation.

`glScalef(GLfloat x, GLfloat y, GLfloat z):`
Generate a scaling transformation and compose it with the current transformation.

`glTranslatef(GLfloat x, GLfloat y, GLfloat z):`
Generate a translation by vector (x, y, z) and compose it with the current transformation.

(Typically) Projection Transformations: One of the following is typically applied in *projection* mode. This sets the manner in which projection (either perspective or parallel) is performed. In all cases the arguments are of type `GLdouble`.

`glOrtho(left, right, bottom, top, near, far):`
: Generate an orthogonal projection transformation (projecting along the z axis) where the clipping region consists of points satisfying $\textit{left} \leq x \leq \textit{right}$, $\textit{bottom} \leq y \leq \textit{top}$ and $\textit{near} \leq z \leq \textit{far}$. All other points are clipped away.

`gluOrtho2D(left, right, bottom, top):`
: A two dimensional version of `glOrtho()`, when the z-coordinates are irrelevant. (It sets $\textit{near} = -1$ and $\textit{far} = +1$.)

`glFrustum(left, right, bottom, top, near, far):`
: Generate a perspective projection transformation and compose it with the current transformation. The parameters specify the bounds of the viewing frustum. The arguments *near* and *far* are the distances to the near and far clipping planes along the $-z$-axis. The other arguments specify the bounds of the viewing rectangle on the near clipping plane.

`gluPerspective(fovy, aspect, zNear, zFar):`
: A more restrictive, but more natural version of `glFrustum()`. It generates a perspective projection transformation for the given up-down field of view (in degrees), for a window with the given aspect ratio (width/height) and the given near and far clipping planes. It composes this with the current transformation.

Options: OpenGL offers a number of capabilities that may be enabled or disabled.

`glEnable(GLenum cap), glDisable(GLenum cap):`
: Enable/disable some option. Some common options include the following. By default, all are initially disabled.

`GL_CULL_FACE`	back-face culling (for hidden surface removal)
`GL_DEPTH_TEST`	depth-buffering (for hidden surface removal)
`GL_FOG`	fog
`GL_BLEND`	color blending
`GL_LIGHTING`	lighting
`GL_LIGHTi`	turn on/off the ith light source ($0 \leq i \leq 7$)
`GL_NORMALIZE`	automatic normalization of normals
`GL_TEXTURE_2D`	texture mapping

Note that options may be enabled and disabled for individual polygons. For example, texture mapping may be turned on before drawing one polygon, and then turned off for others.

Lighting: In OpenGL there may be up to 8 light sources (`GL_LIGHT0` through `GL_LIGHT7`). If lighting is enabled (see `glEnable()`) then the shading of each object depends on which light sources are turned on (enabled) and the materials and surface normals of each of the objects in the scene. Note that when lighting is enabled, it is important that each vertex be associated with a proper normal vector (by calling `glNormal*()`) prior to generating the vertex.

`glShadeModel(GLenum mode):`

The *mode* may be either `GL_FLAT` or `GL_SMOOTH`. In flat shading every point on a polygon is shaded according to its first vertex. In smooth shading the shading from each of the various vertices is interpolated.

`glLightModelf(GLenum pname, GLfloat param):`
`glLightModelfv(GLenum pname, const GLfloat *params):`

Defines general lighting model parameters. The first version is for defining scalar parameters, and the second is for vector parameters. One important parameter is the global intensity of ambient light (independent of any light sources). Its *pname* is `GL_LIGHT_MODEL_AMBIENT` and *params* is a pointer to an RGBA vector.

`glLightf(GLenum light, GLenum pname, GLfloat param):`
`glLightfv(GLenum light, GLenum pname, const GLfloat *params):`

Defines parameters for a single light source. The first version is for defining scalar parameters, and the second is for vector parameters. The first argument indicates which light source this applies to. The argument *pname* gives one of the properties to be assigned. These include the following:

`GL_POSITION`	(vector) (x, y, z, w) of position of light
`GL_AMBIENT`	(vector) RGBA of intensity of ambient light
`GL_DIFFUSE`	(vector) RGBA of intensity of diffuse light
`GL_SPECULAR`	(vector) RGBA of intensity of specular light

By default, illumination intensity does not decrease, or attenuate, with distance. In general, if d is the distance from the light source to the object, and the light source is not a point at infinity, then the intensity attenuation is given by $1/(a + bd + cd^2)$ where a, b, and c are specified by the following parameters:

GL_CONSTANT_ATTENUATION	(scalar) a-coefficient
GL_LINEAR_ATTENUATION	(scalar) b-coefficient
GL_QUADRATIC_ATTENUATION	(scalar) c-coefficient.

Normally light sources send light uniformly in all directions. To define a spotlight, set the following parameters.

GL_SPOT_CUTOFF	(scalar) maximum spread angle of spotlight
GL_SPOT_DIRECTION	(vector) (x, y, z, w) direction of spotlight
GL_SPOT_EXPONENT	(scalar) exponent of spotlight distribution

Note: In addition to defining these properties, each light source must also be enabled. See `glEnable()`.

`glMaterialf(GLenum face, GLenum pname, GLfloat param):`
`glMaterialfv(GLenum face, GLenum pname, const GLfloat *params):`

Defines surface material parameters for subsequently defined objects. The first version is for defining scalar parameters, and the second is for vector parameters. Polygonal objects in OpenGL have two sides. You can assign properties either to the front, back, or both sides. (The front side is the one from which the vertices appear in counterclockwise order.) The first argument indicates the side. The possible values are GL_FRONT, GL_BACK, and GL_FRONT_AND_BACK. The second argument is the specific property. Possbilities include:

GL_EMISSION	(vector) RGBA of the emitted coefficients
GL_AMBIENT	(vector) RGBA of the ambient coefficients
GL_DIFFUSE	(vector) RGBA of the diffuse coefficients
GL_SPECULAR	(vector) RGBA of the specular coefficients
GL_SHININESS	(scalar) single number in the range $[0, 128]$ that indicates degree of shininess.

Blending and Fog: Blending and fog are two OpenGL capabilities that allow you to produce interesting lighting and coloring affects. When a pixel is to be drawn on the screen, it normally overwrites any existing pixel color. When blending is enabled (by calling `glEnable(GL_BLEND)`) then the new (source) pixel is blended with the existing (destination) pixel in the frame buffer.

`glBlendFunc(GLenum sfactor, GLenum dfactor):`

Determines how new pixel values are blended with existing values. The arguments indicate what factors are multiplied by the source and destination RGBA values, respectively, before they are added together. See the reference manuals for complete information.

Fog produces an effect whereby more distant objects are blended increasingly with a *fog color*, typically some shade of gray. It is enabled by calling `glEnable(GL_FOG)`.

`glFogf(GLenum pname, GLfloat param):`
`glFogfv(GLenum pname, const GLfloat *params):`

Specifies the parameters that define how fog is computed. The first version is for defining scalar parameters, and the second is for vector parameters. See the reference manual for complete details.

Texture Mapping: Texture mapping is the process of taking an image, presented typically as a 2-dimensional array of RGB values and mapping it onto a polygon. Setting up texture mapping involves the following steps: define a texture by specifying the image and its format (through `glTexImage2d()`), specify how object vertices correspond to points in the texture, and finally enable texture mapping. First, the texture must be input or generated by the program. OpenGL provides a wide variety of other features, but we will only summarize a few here, which are sufficient for handling a single 2-dimensional texture. To handle multiple textures, see the commands `glGenTextures()` and `glBindTexture()`.

`glTexImage2D (GLenum target, int level, int internalFormat, int width, int height, int border, GLenum format, GLenum type, void *pixels):`

This converts a texture stored in the array *pixels* into an internal format and stored in *pixels*. The first argument is typically `GL_TEXTURE_2D`. The other arguments specify various elements of how the mapping is to be performed. See the reference manual for complete information.

`glTexEnvf(GLenum target, GLenum pname, GLfloat param):`

Specifies texture mapping environment parameters. The target must be `GL_TEXTURE_ENV`. The pname must be `GL_TEXTURE_ENV_MODE`. This determines how a color from the texture image is to be merged with an existing color on the surface of the polygon. The param may be any of the following:

`GL_MODULATE`	multiply color components together
`GL_BLEND`	linearly blend color components
`GL_DECAL`	use the texture color
`GL_REPLACE`	use the texture color

There are subtle differences between `GL_DECAL` and `GL_REPLACE` when different formats are used or when the A component of the RGBA color is not 1. See the reference manual. The default is `GL_MODULATE`.

`glTexParameterf(GLenum target, GLenum pname, GLfloat param):`
`glTexParameterfv(GLenum target, GLenum pname, const GLfloat *params):`

Specify how texture interpolation is to be performed. The first version is for defining scalar parameters, and the second is for vector parameters. Assuming 2-dimensional textures, the target is `GL_TEXTURE_2D`, the *pname* is either:

`GL_TEXTURE_MAG_FILTER`	magnification filter
`GL_TEXTURE_MIN_FILTER`	minification filter

Magnification is used when a pixel of the texture is smaller than the corresponding pixel of the screen onto which it is mapped and minification applies in the opposite case. Typical values are either

`GL_NEAREST`	take the nearest texture pixel
`GL_LINEAR`	take the weighted average of the 4 surrounding texture pixels

This procedure may also be invoked to specify other properties of texture mapping.

`glTexCoord*(...):`
Specifies the texture coordinates of subsequently defined vertices for texture mapping. For a standard 2-dimensional textures, the texture coordinates are a pair (s, t) in the interval $[0, 1] \times [0, 1]$. The texture coordinate specifies the point on the image that are to be mapped to this vertex. OpenGL interpolates the mapping of intermediate points of the polygon.