Introduction to OpenGL

OpenGL is a low-level graphics library specification. It makes available to the programmer a small set of geometric primitives - points, lines, polygons, images, and bitmaps. OpenGL provides a set of commands that allow the specification of geometric objects in two or three dimensions, using the provided primitives, together with commands that control how these objects are rendered (drawn).

Since OpenGL drawing commands are limited to those that generate simple geometric primitives (points, lines, and polygons), the OpenGL Utility Toolkit (GLUT) has been created to aid in the development of more complicated three-dimensional objects such as a sphere, a torus, and even a teapot. GLUT may not be satisfactory for full-featured OpenGL applications, but it is a useful starting point for learning OpenGL.

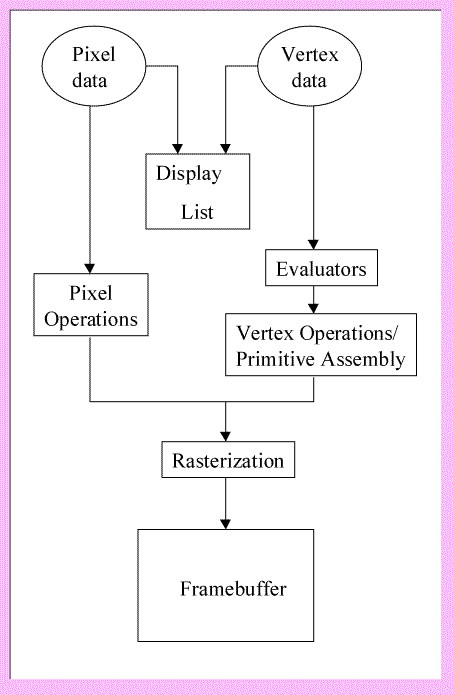
GLUT is designed to fill the need for a window system independent programming interface for OpenGL programs. The interface is designed to be simple yet still meet the needs of useful OpenGL programs. Removing window system operations from OpenGL is a sound decision because it allows the OpenGL graphics system to be retargeted to various systems including powerful but expensive graphics workstations as well as mass-production graphics systems like video games, set-top boxes for interactive television, and PCs.

GLUT simplifies the implementation of programs using OpenGL rendering. The GLUT application programming interface (API) requires very few routines to display a graphics scene rendered using OpenGL. The GLUT routines also take relatively few parameters.

1.1 Rendering Pipeline

Most implementations of OpenGL have a similar order of operations, a series of processing stages called the OpenGL rendering pipeline. Although this is not a strict rule of how OpenGL is implemented, it provides a reliable guide for predicting what OpenGL will do. Geometric data (vertices, line, and polygons) follow a path through the row of boxes that includes evaluators and per-vertex operations, while pixel data (pixels, images and bitmaps) are treated differently for part of the process. Both types of data undergo the same final step (rasterization) before the final pixel data is written to the frame buffer.

Page No. -1



Display Lists: All data, whether it describes geometry or pixels, can be saved in a display list for current or later use. (The alternative to retaining data in a display list is processing the data immediately-known as immediate mode.) When a display list is executed, the retained data is sent from the display list just as if it were sent by the application in immediate mode.

Evaluators: All geometric primitives are eventually described by vertices. Evaluators provide a method for deriving the vertices used to represent the surface from the control points. The method is a polynomial mapping, which can produce surface normal, colors, and spatial coordinate values from the control points.

Page No. -2-

Per-Vertex and Primitive Assembly: For vertex data, the next step converts the vertices into   
primitives. Some types of vertex data are transformed by 4x4 floating-point matrices. Spatial   
coordinates are projected from a position in the 3D world to a position on your screen. In some   
cases, this is followed by perspective division, which makes distant geometric objects appear   
smaller than closer objects. Then view port and depth operations are applied. The results at this   
point are geometric primitives, which are transformed with related color and depth values and   
guidelines for the rasterization step.

Pixel Operations: While geometric data takes one path through the OpenGL rendering pipeline, pixel data takes a different route. Pixels from an array in system memory are first unpacked form one of a variety of formats into the proper number of components. Next the data is scaled, biased, processed by a pixel map, and sent to the rasterization step.

Rasterization: Rasterization is the conversion of both geometric and pixel data into fragments.   
Each fragment square corresponds to a pixel in the frame buffer. Line width, point size, shading   
model, and coverage calculations to support antialiasing are taken it to consideration as vertices   
are connected into lines or the interior pixels are calculated for a filled polygon. Color and depth   
values are assigned for each fragment square. The processed fragment is then drawn into the   
appropriate buffer, where it has finally advanced to be a pixel and achieved its final resting place.

1.2 Libraries

OpenGL provides a powerful but primitive set of rendering command, and all higher-level drawing must be done in terms of these commands. There are several libraries that allow you to simplify your programming tasks, including the following:

 OpenGL Utility Library (GLU) contains several routines that use lower-level OpenGL   
 commands to perform such tasks as setting up matrices for specific viewing orientations

and projections and rendering surfaces.

 OpenGL Utility Toolkit (GLUT) is a window-system-independent toolkit, written by Mark   
 Kilgard, to hide the complexities of differing window APIs.

1.3 Include Files

For all OpenGL applications, you want to include the gl.h header file in every file. Almost all OpenGL applications use GLU, the aforementioned OpenGL Utility Library, which also requires inclusion of the glu.h header file. So almost every OpenGL source file begins with:

Page No. -3

#include<GL/gl.h>   
#include <GL/glu.h>

If you are using the OpenGL Utility Toolkit (GLUT) for managing your window manager tasks, you should include:

#include <GL/glut.h>

Note that glut.h guarantees that gl.h and glu.h are properly included for you so including these three files is redundant. To make your GLUT programs portable, include glut.h and do not include gl.h or glu.h explicitly.

1.4 Setting up Compilers

 Windows Using MS Visual C++

Installing GLUT

Most of the following files (ie. OpenGL and GLU) will already be present if you have installed MS Visual C++ v5.0 or later. The following GLUT files will need to be copied into the specified directories.

1. To install:

 right-click each link

 choose Save Link As...

 accept the default name (just click Save)

 libraries (place in the lib\ subdirectory of Visual C++)   
  [opengl32.lib](ftp://ftp.csis.gvsu.edu/wolffe/graphics/lib/opengl32.lib)

 [glu32.lib](ftp://ftp.csis.gvsu.edu/wolffe/graphics/lib/glu32.lib)

 [glut32.lib](ftp://ftp.csis.gvsu.edu/wolffe/graphics/lib/glut32.lib)

 include files (place in the include\GL\ subdirectory of Visual C++)   
  [gl.h](ftp://ftp.csis.gvsu.edu/wolffe/graphics/include/gl.h)

 [glu.h](ftp://ftp.csis.gvsu.edu/wolffe/graphics/include/glu.h)   
 [glut.h](ftp://ftp.csis.gvsu.edu/wolffe/graphics/include/glut.h)

 dynamically-linked libraries (place in the \Windows\System subdirectory   
  [opengl32.dll](ftp://ftp.csis.gvsu.edu/wolffe/graphics/dll/opengl32.dll)

 [glu32.dll](ftp://ftp.csis.gvsu.edu/wolffe/graphics/dll/glu32.dll)

 [glut32.dll](ftp://ftp.csis.gvsu.edu/wolffe/graphics/dll/glut32.dll)

Page No. -4-

Compiling OpenGL/GLUT Programs

1. Create a new project:

o choose File | New from the File Menu

o select the Projects tab

o choose Win32 Console Application

o fill in your Project name

2. Designate library files for the linker to use:

o choose Project | Settings from the File Menu

o under Oject/library modules: enter "opengl32.lib glu32.lib glut32.lib" 3. Add/Create files to the project:

o choose Project | Add to Project | Files from the File menu

o add the required program files

4. Build and Execute

1.5 Initialization

The first thing we need to do is call the [glutInit()](http://cs.uccs.edu/~semwal/man.html#glutInit) procedure. It should be called before any   
other GLUT routine because it initializes the GLUT library. The parameters to [glutInit()](http://cs.uccs.edu/~semwal/man.html#glutInit) should   
be the same as those to main(), specifically main(int argc, char\*\* argv) and [glutInit(&argc, argv),](http://cs.uccs.edu/~semwal/man.html#glutInit)   
where argcp is a pointer to the program's unmodified argc variable from main. Upon return, the value pointed to by argcp will be updated, and argv is the program's unmodified argv variable from main. Like argcp, the data for argv will be updated.

The next thing we need to do is call the [glutInitDisplayMode()](http://cs.uccs.edu/~semwal/man.html#glutInitDisplayMode) procedure to specify the   
display mode for a window. You must first decide whether you want to use an RGBA   
(GLUT\_RGBA) or color-index (GLUT\_INDEX) color model. The RGBA mode stores its color   
buffers as red, green, blue, and alpha color components. The forth color component, alpha,   
corresponds to the notion of opacity. An alpha value of 1.0 implies complete opacity, and an   
alpha value of 0.0 complete transparancy. Color-index mode, in contrast, stores color buffers in   
indices. Your decision on color mode should be based on hardware availability and what you   
application requires. More colors can usually be simultaneously represented with RGBA mode   
than with color-index mode. And for special effects, such as shading, lighting, and fog, RGBA   
mode provides more flexibility. In general, use RGBA mode whenever possible. RGBA mode is   
the default.

Page No. -5-

Another decision you need to make when setting up the display mode is whether you   
want to use single buffering (GLUT\_SINGLE) or double buffering (GLUT\_DOUBLE).   
Applications that use both front and back color buffers are double-buffered. Smooth animation is   
accomplished by rendering into only the back buffer (which isn't displayed), then causing the   
front and back buffers to be swapped. If you aren't using animation, stick with single buffering,   
which is the default.

Finally, you must decide if you want to use a depth buffer (GLUT\_DEPTH), a stencil   
buffer (GLUT\_STENCIL) and/or an accumulation buffer (GLUT\_ACCUM). The depth buffer   
stores a depth value for each pixel. By using a "depth test", the depth buffer can be used to   
display objects with a smaller depth value in front of objects with a larger depth value. The   
second buffer, the stencil buffer is used to restrict drawing to certain portions of the screen, just   
as a cardboard stencil can be used with a can of spray paint to make a printed image. Finally, the   
accumulation buffer is used for accumulating a series of images into a final composed image.   
None of these are default buffers.

We need to create the characteristics of our window. A call to [glutInitWindowSize()](http://cs.uccs.edu/~semwal/man.html#glutInitWindowSize) will   
be used to specify the size, in pixels, of your initial window. The arguments indicate the height   
and width (in pixels) of the requested window. Similarly, [glutInitWindowPosition()](http://cs.uccs.edu/~semwal/man.html#glutInitWindowPosition) is used to   
specify the screen location for the upper-left corner of your initial window. The arguments, x and y, indicate the location of the window relative to the entire display.

1.6 Creating a Window

To actually create a window, the with the previously set characteristics (display mode,   
size, location, etc), the programmer uses the [glutCreateWindow()](http://cs.uccs.edu/~semwal/man.html#glutCreateWindow) command. The command takes   
a string as a parameter which may appear in the title bar if the window system you are using   
supports it. The window is not actually displayed until the [glutMainLoop()](http://cs.uccs.edu/~semwal/man.html#glutMainLoop) is entered.

1.7 Display Function

The [glutDisplayFunc()](http://cs.uccs.edu/~semwal/man.html#glutDisplayFunc) procedure is the first and most important event callback function   
you will see. A callback function is one where a programmer-specified routine can be registered   
to be called in response to a specific type of event. For example, the argument of   
[glutDisplayFunc()](http://cs.uccs.edu/~semwal/man.html#glutDisplayFunc) is the function that is called whenever GLUT determines that the contents of

Page No. -6-

the window needs to be redisplayed. Therefore, you should put all the routines that you need to draw a scene in this display callback function.

1.8 Reshape Function

The [glutReshapeFunc()](http://cs.uccs.edu/~semwal/man.html#glutReshapeFunc) is a callback function that specifies the function that is called   
whenever the window is resized or moved. Typically, the function that is called when needed by   
the reshape function displays the window to the new size and redefines the viewing   
characteristics as desired. If [glutReshapeFunc()](http://cs.uccs.edu/~semwal/man.html#glutReshapeFunc) is not called, a default reshape function is called   
which sets the view to minimize distortion and sets the display to the new height and width.

1.9 Main Loop

The very last thing you must do is call [glutMainLoop().](http://cs.uccs.edu/~semwal/man.html#glutMainLoop) All windows that have been   
created can now be shown, and rendering those windows is now effective. The program will now   
be able to handle events as they occur (mouse clicks, window resizing, etc). In addition, the   
registered display callback (from our [glutDisplayFunc())](http://cs.uccs.edu/~semwal/man.html#glutDisplayFunc) is triggered. Once this loop is entered, it   
is never exited!

Page No. -7-

Geometric Objects used in graphics

2.1 Point, Lines and Polygons

Each geometric object is described by a set of vertices and the type of primitive to be   
drawn. A vertex is no more than a point defined in three dimensional space. Whether and how   
these vertices are connected is determined by the primitive type. Every geometric object is   
ultimately described as an ordered set of vertices. We use the [glVertex\*()](http://cs.uccs.edu/~semwal/glman.html#glVertex) command to specify a   
vertex. The '\*' is used to indicate that there are variations to the base command [glVertex().](http://cs.uccs.edu/~semwal/glman.html#glVertex)

Some OpenGL command names have one, two, or three letters at the end to denote the number   
and type of parameters to the command. The first character indicates the number of values of the   
indicated type that must be presented to the command. The second character indicates the   
specific type of the arguments. The final character, if present, is 'v', indicating that the command   
takes a pointer to an array (a vector) of values rather than a series of individual agreements.

For example, in the command [glVertex3fv(),](http://cs.uccs.edu/~semwal/glman.html#glVertex) '3' is used to indicate three arguments, 'f' is used to   
indicate the arguments are floating point, and 'v' indicates that the arguments are in vector format.

Points: A point is represented by a single vertex. Vertices specified by the user as two-  
dimensional (only x- and y-coordinates) are assigned a z-coordinate equal to zero. To control the   
size of a rendered point, use [glPointSize()](http://cs.uccs.edu/~semwal/glman.html#glPointSize) and supply the desired size in pixels as the argument.   
The default is as 1 pixel by 1 pixel point. If the width specified is 2.0, the point will be a square   
of 2 by 2 pixels. [glVertex\*()](http://cs.uccs.edu/~semwal/glman.html#glVertex) is used to describe a point, but it is only effective between a   
[glBegin()](http://cs.uccs.edu/~semwal/glman.html#glBegin) and a [glEnd()](http://cs.uccs.edu/~semwal/glman.html#glEnd) pair. The argument passed to glBegin() determines what sort of   
geometric primitive is constructed from the vertices.

Lines: In OpenGL, the term line refers to a line segment, not the mathematician's version that   
extends to infinity in both directions. The easiest way to specify a line is in terms of the vertices   
at the endpoints. As with the points above, the argument passed to [glBegin()](http://cs.uccs.edu/~semwal/glman.html#glBegin) tells it what to do   
with the vertices. The option for lines includes:

GL\_LINES: Draws a series of unconnected line segments drawn between each set of vertices. An

extraneous vertex is ignored.

GL\_LINE\_STRIP: Draws a line segment from the first vertex to the last. Lines can intersect

arbitrarily.

Page No. -8-

GL\_LINE\_LOOP: Same as GL\_STRIP, except that a final line segment is drawn from the last vertex back to the first.

With OpenGL, the description of the shape of an object being drawn is independent of the   
description of its color. When a particular geometric object is drawn, it's drawn using the   
currently specified coloring scheme. In general, an OpenGL programmer first sets the color,   
using [glColor\*()](http://cs.uccs.edu/~semwal/glman.html#glColor) and then draws the objects. Until the color is changed, all objects are drawn in   
that color or using that color scheme.

Polygons: Polygons are the areas enclosed by single closed loops of line segments, where the   
line segments are specified by the vertices at their endpoints. Polygons are typically drawn with   
the pixels in the interior filled in, but you can also draw them as outlines or a set of points. In   
OpenGL, there are a few restrictions on what constitutes a primitive polygon. For example, the   
edges of a polygon cannot intersect and they must be convex (no indentations). There are special   
commands for a three-sided (triangle) and four-sided (quadrilateral) polygons,   
[glBegin(GL\_TRIANGLES)](http://cs.uccs.edu/~semwal/glman.html#glBegin) and [glBegin(GL\_QUADS),](http://cs.uccs.edu/~semwal/glman.html#glBegin) respectively. However, the general case   
of a polygon can be defined using [glBegin(GL\_POLYGON).](http://cs.uccs.edu/~semwal/glman.html#glBegin)

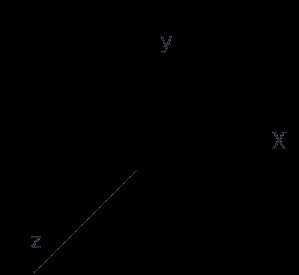
2.2 Drawing 3-D Objects

GLUT has a series of drawing routines that are used to create three-dimensional models. This means we don't have to reproduce the code necessary to draw these models in each program. These routines render all their graphics in immediate mode. Each object comes in two flavors, wire or solid. Available objects are:

2.3 Transformations

A modeling transformation is used to position and orient the model. For example, you can rotate, translate, or scale the model - or some combination of these operations. To make an object appear further away from the viewer, two options are available - the viewer can move closer to the object or the object can be moved further away from the viewer. Moving the viewer will be discussed later when we talk about viewing transformations. For right now, we will keep the default "camera" location at the origin, pointing toward the negative z-axis, which goes into the screen perpendicular to the viewing plane.

Page No. -9-



When transformations are performed, a series of matrix multiplications are actually performed to   
affect the position, orientation, and scaling of the model. You must, however, think of these   
matrix multiplications occuring in the opposite order from how they appear in the code. The   
order of transformations is critical. If you perform transformation A and then perform   
transformation B, you almost always get something different than if you do them in the opposite   
order.

Scaling: The scaling command [glScale()](http://cs.uccs.edu/~semwal/glman.html#glScale) multiplies the current matrix by a matrix that stretches,   
shrinks, or reflects an object along the axes. Each x-, y-, and z-coordinate of every point in the   
object is multiplied by the corresponding argument x, y, or z. The [glScale\*()](http://cs.uccs.edu/~semwal/glman.html#glScale) is the only one of   
the three modeling transformations that changes the apparent size of an object: scaling with values greater than 1.0 stretches an object, and using values less than 1.0 shrinks it. Scaling with a -1.0 value reflects an object across an axis.

Translation: The translation command [glTranslate()](http://cs.uccs.edu/~semwal/glman.html#glTranslate) multiplies the current matrix by a matrix   
that moves (translates) an object by the given x-, y-, and z-values.

Rotation: The rotation command [glRotate()](http://cs.uccs.edu/~semwal/glman.html#glRotate) multiplies the current matrix that rotates an object in   
a counterclockwise direction about the ray from the origin through the point (x,y,z). The angle   
parameter specifies the angle of rotation in degrees. An object that lies farther from the axis of   
rotation is more dramatically rotated (has a larger orbit) than an object drawn near the axis.

Page No. -10-

Sample Programs

Sample Program: 01

/\* A basic Open GL window\*/ #include<GL/glut.h>

void display (void)   
{

glClearColor (0.0,0.0,0.0,1.0); glClear (GL\_COLOR\_BUFFER\_BIT); glLoadIdentity ();

gluLookAt (0.0,0.0,5.0,0.0,0.0,0.0,0.0,1.0,0.0); glFlush ();

}

int main (int argc,char \*\*argv)   
{

glutInit (&argc,argv);

glutInitDisplayMode (GLUT\_SINGLE);

glutInitWindowSize (500,500);

glutInitWindowPosition (100,100);

glutCreateWindow ("A basic open GL window"); glutDisplayFunc (display);

glutMainLoop ();

return 0;

}