Getting Started with OpenGL Graphics Programming

Oregon State University

Mike Bailey
mjb@cs.oregonstate.edu

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License

The Basic Computer Graphics Pipeline

Model Transform → WC
View Transform → EC
Per-vertex Lighting
Projection Transform → CC
Homogeneous Division
Viewport Transform → NDC

Transferr
Raster Ops
Fragment Processing, Texturing, Per-fragment Lighting

We’ll come back to this later. For now, understand that there are multiple steps to go from your 3D geometry to pixels on the screen.

MC = Model Coordinates
WC = World Coordinates
EC = Eye Coordinates
CC = Clip Coordinates
NDC = Normalized Device Coordinates
SC = Screen Coordinates

Geometry vs. Topology

Geometry: Where things are (e.g., coordinates)
Topology: How things are connected

3D Coordinate Systems

Left-handed
Right-handed

Homer Simpson uses Right-handed Coordinates. So then we will too.

Right-handed 3D Coordinate System for a CNC Machine
Right-handed Positive Rotations

Right-Handed Coordinate System

Drawing in 3D

Set any display-characteristics state that you want to have in effect when you do the drawing.

Begin the drawing. Use the current stack's display-characteristics. Here is the topology to be used with these vertices.

OpenGL Topologies

OpenGL Topologies -- Polygon Requirements

OpenGL Topologies -- Orientation

Polygons are traditionally:

- CCW when viewed from outside the solid object

This is a wonderfully understandable way to start with 3D graphics, but it is also incredibly internally inefficient! We'll talk about that later...

Polygons must be:

- Convex and
- Planar

For that reason, GL_TRIANGLE_STRIP and GL_TRIANGLE are considered preferable to GL_QUAD_STRIP and GL_QUADS. GL_POLYGON is rarely used.
OpenGL Topologies – Vertex Order Matters

GL_LINE_LOOP

V0 V1 V2 V3

Probably what you meant to do

GL_LINE_LOOP

V0 V2 V1

Probably not what you meant to do

This disease is referred to as “The Bowtie” 😊

What does “Convex Polygon” Mean?

We can go all mathematical here, but let’s go visual instead. In a convex polygon, a line between any two points inside the polygon never leaves the inside of the polygon.

Convex

V0 V1 V2 V3

Stays within the polygon

Not Convex

V0 V2 V1

Leaves the polygon

Why is there a Requirement for Polygons to be Convex?

Graphics polygon-filling hardware can be highly optimized if you know that, no matter what direction you fill the polygon in, there will be two and only two intersections between the scanline and the polygon’s edges.

Convex

2 edge intersections

Not Convex

4 edge intersections

What if you need to display Polygons that are not Convex?

There are two good solutions I know of (and there are probably more):

1. OpenGL’s utility (gluXxx) library has a built-in tessellation capability to break a non-convex polygon into convex polygons.

2. There is an open source library to break a non-convex polygon into convex polygons. It is called Polypartition, and the source code can be found here: https://github.com/ivanfratric/polypartition

If you ever need to do this, contact me. I have working code for each approach…

Why is there a Requirement for Polygons to be Planar?

Graphics hardware assumes that a polygon has a definite front and a definite back, and that you can only see one of them at a time.

OpenGL Drawing Can Be Done Procedurally

Listing a lot of vertices explicitly gets old in a hurry

glColor3f( r, g, b );
glBegin( GL_LINE_LOOP );
glVertex3f( x0, y0, 0. );
. . .
glEnd( );

The graphics card can’t tell how the numbers in the glVertex3f calls were produced: both explicitly listed and procedurally computed look the same to glVertex3f.

gColor3f( r, g, b );
float dang = 2.*M_PI / (float)( NUMSEGS – 1 );
float ang = 0.;

gBegin( GL_LINE_LOOP );
for( int i = 0; i < NUMSEGS; i++ )
{
gVertex3f( RADIUS*cos(ang), RADIUS*sin(ang), 0. );
ang += dang;
}
gEnd( );

Listing a lot of vertices explicitly gets old in a hurry

glColor3f( r, g, b );
float dang = 2.*M_PI / (float)( NUMSEGS – 1 );
float ang = 0.;
gBegin( GL_LINE_LOOP );
for( int i = 0; i < NUMSEGS; i++ )
{
gVertex3f( RADIUS*cos(ang), RADIUS*sin(ang), 0. );
ang += dang;
}
gEnd( );
Color

This is referred to as “Additive Color”

<table>
<thead>
<tr>
<th>Red</th>
<th>Green</th>
<th>Cyan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Magenta</td>
<td>Yellow</td>
<td>White</td>
</tr>
<tr>
<td>Blue</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

glColor3f( r, g, b );

0.0 ≤ r, g, b ≤ 1.0

Transformations

Translation
Rotation
Scaling

OpenGL Transformations

glTranslatef( tx, ty, tz );

glRotatef( degrees, ax, ay, az );

glScalef( sx, sy, sz );

Single Transformations

glMatrixMode( GL_MODELVIEW );

gLoadIdentity( )

glRotatef( degrees, ax, ay, az );

gColor3f( r, g, b );

gBegin( GL_LINE_STRIP );

gVertex3f( x0, y0, z0 );

gVertex3f( x1, y1, z1 );

gVertex3f( x2, y2, z2 );

gVertex3f( x3, y3, z3 );

gVertex3f( x4, y4, z4 );

gEnd( );

Compounding Transformations

These transformations ‘add up’, and look like they take effect in this order

1. glMatrixMode( GL_MODELVIEW );
2. gLoadIdentity( )
3. glTranslatef( tx, ty, tz );
4. glRotatef( degrees, ax, ay, az );
5. glScalef( sx, sy, sz );
6. glColor3f( r, g, b );
7. glBegin( GL_LINE_STRIP );
8. glVertex3f( x0, y0, z0 );
9. glVertex3f( x1, y1, z1 );
10. glVertex3f( x2, y2, z2 );
11. glVertex3f( x3, y3, z3 );
12. glVertex3f( x4, y4, z4 );
13. glEnd( );

Why do the Compound Transformations Take Effect in Reverse Order?

Envision fully-parenthesizing what is going on. In that case, it makes perfect sense that the most recently-set transformation would take effect first.
Order Matters!

Compound Transformations are Not Commutative

Rotate, then translate

Translate, then rotate

The OpenGL Drawing State

The designers of OpenGL could have put lots and lots of arguments on the glVertex3f call to totally define the appearance of your drawing, like this:

```
glVertex3f( x, y, z, r, g, b, m00, ..., m33, s, t, nx, ny, nz, linewidth, ... );
```

Yuck! That would have been ugly. Instead, they decided to let you create a “current drawing state”. You set all of these characteristics first, then they take effect when you do the drawing. They continue to remain in effect for future drawing calls, until you change them.

You must set the transformations before you expect them to take effect!

Set the state first

Use the state second

Projecting an Object from 3D into 2D

Orthographic (or Parallel) projection

```
glOrtho( xL, xR, yB, yT, zN, zF );
```

Perspective projection

```
gluPerspective( fovy, aspect, zN, zF );
```

Parallel lines remain parallel

Parallel lines appear to converge

“Vanishing Point”

OpenGL Projection Functions

```
glMatrixMode( GL_PROJECTION );
glLoadIdentity( );
gluLookAt( ex, ey, ez, lx, ly, lz, ux, uy, uz );
glTranslatef( tx, ty, tz );
glRotatef( degrees, ax, ay, az );
glScalef( sx, sy, sz );
```

```
glColor3f( r, g, b );
gBegin( GL_LINE_STRIP );
gVertex3f( x0, y0, z0 );
gVertex3f( x1, y1, z1 );
gVertex3f( x2, y2, z2 );
gVertex3f( x3, y3, z3 );
gVertex3f( x4, y4, z4 );
gEnd( );
```

Use one of these, but not both!

Projecting on Object from 3D to 2D

Parallel/Orthographic is good for lining things up and comparing sizes

Perspective is more realistic-looking

How the Viewing Volumes Look from the Outside

```
glOrtho( xL, xR, yB, yT, zN, zF );
gluPerspective( fovy, aspect, zN, zF );
```

Parallel/Orthographic

Perspective
The Perspective Viewing Frustum

\[ \text{gluPerspective}( \text{fovy}, \text{aspect}, \text{zn}, \text{zf}); \]

- **fovy** = vertical field of view angle (degrees)
  - (good values are 50-100°)

Arbitrary Viewing

\[ \text{glMatrixMode}( \text{GL_PROJECTION}); \]
\[ \text{glLoadIdentity}(); \]
\[ \text{gluPerspective}( \text{fpvy}, \text{aspect}, \text{zn}, \text{zf}); \]
\[ \text{glMatrixMode}( \text{GL_MODELVIEW}); \]
\[ \text{glLoadIdentity}(); \]
\[ \text{gluLookAt}( \text{ex, ey, ez}, \text{lx, ly, lz}, \text{ux, uy, uz}); \]
\[ \text{glTranslatef}( \text{tx, ty, tz}); \]
\[ \text{glRotatef}( \text{degrees}, \text{ax, ay, az}); \]
\[ \text{glScalef}( \text{sx, sy, sz}); \]
\[ \text{glColor3f}( \text{r, g, b}); \]
\[ \text{glBegin}( \text{GL_LINE_STRIP}); \]
\[ \text{glVertex3f}( \text{x0, y0, z0}); \]
\[ \text{glVertex3f}( \text{x1, y1, z1}); \]
\[ \text{glVertex3f}( \text{x2, y2, z2}); \]
\[ \text{glVertex3f}( \text{x3, y3, z3}); \]
\[ \text{glVertex3f}( \text{x4, y4, z4}); \]
\[ \text{glEnd}(); \]

Chicago Fly-through: Changing Eye, Look, and Up

Here’s a good way to start:
1. Set \( \text{lx, ly, lz} \) to be the average of all the vertices
2. Set \( \text{ux, uy, uz} \) to be 0, 1, 0
3. Set \( \text{ex} = \text{lx} \) and \( \text{ey} = \text{ly} \)
4. Now, you change \( \Delta E \) or \( \text{fovy} \) so that the object fits in the viewing volume:

\[
\frac{\Delta E}{\tan \left( \frac{\text{fovy}}{2} \right)} = \frac{H}{2}
\]

How Can You Be Sure You See Your Scene?

\[ \text{gluPerspective}( \text{fovy}, \text{aspect}, \text{zn}, \text{zf}); \]
\[ \text{gluLookAt}( \text{ex, ey, ez}, \text{lx, ly, lz}, \text{ux, uy, uz}); \]

Specifying a Viewport

\[ \text{glViewport}( \text{ixl, iyb, idx, idy}); \]

Be sure the y:x aspect ratios match!!

Saving and Restoring the Current Transformation

\[ \text{glMatrixMode}( \text{GL_PROJECTION}); \]
\[ \text{glLoadIdentity}(); \]
\[ \text{gluPerspective}( \text{fovy}, \text{aspect}, \text{zn}, \text{zf}); \]
\[ \text{glMatrixMode}( \text{GL_MODELVIEW}); \]
\[ \text{glLoadIdentity}(); \]
\[ \text{gluLookAt}( \text{ex, ey, ez}, \text{lx, ly, lz}, \text{ux, uy, uz}); \]
\[ \text{glTranslatef}( \text{tx, ty, tz}); \]
\[ \text{glRotatef}( \text{degrees}, \text{ax, ay, az}); \]
\[ \text{glScalef}( \text{sx, sy, sz}); \]
\[ \text{glColor3f}( \text{r, g, b}); \]
\[ \text{glBegin}( \text{GL_LINE_STRIP}); \]
\[ \text{glVertex3f}( \text{x0, y0, z0}); \]
\[ \text{glVertex3f}( \text{x1, y1, z1}); \]
\[ \text{glVertex3f}( \text{x2, y2, z2}); \]
\[ \text{glVertex3f}( \text{x3, y3, z3}); \]
\[ \text{glVertex3f}( \text{x4, y4, z4}); \]
\[ \text{glEnd}(); \]
\[ \text{glPopMatrix}(); \]

Note: setting the viewport is not part of setting either the Modelview or the Projection transformations.
sample.cpp Program Structure

- #includes
- Consts and #defines
- Global variables
- Function prototypes
- Main program
- InitGraphics function
- Display callback
- Keyboard callback

#includes

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#define _USE_MATH_DEFINES
#include <math.h>
#ifdef WIN32
#include <windows.h>
#pragma warning(disable:4996)
#endif
#include <GL/gl.h>
#include <GL/glu.h>
#include "glut.h"
```

consts and #defines

```c
const char *WINDOWTITLE = { "OpenGL / GLUT Sample -- Joe Graphics" };
const char *GLUITITLE   = { "User Interface Window" };
const int GLUITRUE  = { true  };
const int GLUIFALSE = { false };
#define ESCAPE          0x1b
const int INIT_WINDOW_SIZE = { 600 };
const float BOXSIZE = { 2.f };
const float ANGFACT = { 1. };
const float SCLFACT = { 0.005f };
const float MINSCALE = { 0.05f };
const int LEFT   = { 4 };
const int MIDDLE = { 2 };
const int RIGHT  = { 1 };
enum Projections
{ ORTHO, PERSP };
enum ButtonVals
{ RESET, QUIT };
enum Colors
{ RED, YELLOW, GREEN, CYAN, BLUE, MAGENTA, WHITE, BLACK };
```

Initialized Global Variables

```c
const GLfloat BACKCOLOR[4] = { 0., 0., 0., 1. };
const GLfloat AXES_WIDTH   = { 3. };
char * ColorNames[] =
{
    "Red",
    "Yellow",
    "Green",
    "Cyan",
    "Blue",
    "Magenta",
    "White",
    "Black"
};
const GLfloat Colors[][3] =
{
    { 1., 0., 0. }, // red
    { 1., 1., 0. }, // yellow
    { 0., 1., 0. }, // green
    { 0., 1., 1. }, // cyan
    { 0., 0., 1. }, // blue
    { 1., 0., 1. }, // magenta
    { 1., 1., 1. }, // white
    { 0., 0., 0. }, // black
};
const GLfloat FOGCOLOR[4] = { .0, .0, .0, 1. };
const GLenum FOGMODE     = { GL_LINEAR };
const GLfloat FOGDENSITY  = { 0.30f };
const GLfloat FOGSTART      = { 1.5 };
const GLfloat FOGEND          = { 4. };
```

Global Variables

```c
int ActiveButton; // current button that is down
GLuint AxesList; // list to hold the axes
int AxesOn; // != 0 means to draw the axes
int DebugOn; // != 0 means to print debugging info
int DepDepthCueOn; // != 0 means to use intensity depth cueing
GLuint List; // object display list
int MainWindow; // window id for main graphics window
float Scale; // scaling factor
int WhichColor; // index into Colors[]
int WhichProjection; // ORTHO or PERSP
int Xmouse, Ymouse; // mouse values
float Xrot, Yrot; // rotation angles in degrees
```

Function Prototypes

```c
void Animate();
void Display();
void DoAxesMenu(int);
void DoColorMenu(int);
void DoDebugMenu(int);
void DoMainMenu(int);
void DoProjectMenu(int);
void DoRasterString(float, float, float, char *);
void DoStrokeString(float, float, float, float, char *);
float ElapsedSeconds();
void InitGraphics();
void InitLists();
void InitMenus();
void Keyboard(unsigned char, int, int);
void MouseButton(int, int, int, int);
void MouseMotion(int, int);
void Reset();
void Resize(int, int);
void Visibility(int);
void Axes(float);
void HsvRgb(float[3], float[3]);
```
Main Program

```c
void
main( int argc, char *argv[ ] )
{
    // turn on the glut package:
    // (do this before checking argc and argv since it might
    // pull some command-line arguments out)
    glutInit( &argc, argv );
    // setup all the graphics stuff:
    InitGraphics();
    // create the display structures that will not change:
    InitLists();
    // init all the global variables used by Display() ;
    // this will also post a redisplay:
    Reset();
    // setup all the user interface stuff:
    InitMenus();
    // draw the scene once and wait for some interaction:
    // (this will never return)
    glutSetWindow( MainWindow );
    glutMainLoop( );
    // this is here to make the compiler happy:
    return 0;
}
```

InitGraphics(), I

```c
void
InitGraphics( )
{
    // request the display modes:
    // ask for red-green-blue-alpha color, double-buffering, and z-buffering:
    glutInitDisplayMode( GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH );
    // set the initial window configuration:
    glutInitWindowPosition( 0, 0 );
    glutInitWindowSize( INIT_WINDOW_SIZE, INIT_WINDOW_SIZE );
    // open the window and set its title:
    MainWindow = glutCreateWindow( WINDOWTITLE );
    glutSetWindowTitle( WINDOWTITLE );
    // set the framebuffer clear values:
    glClearColor( BACKCOLOR[0], BACKCOLOR[1], BACKCOLOR[2], BACKCOLOR[3] );
    // set the display callback functions:
    glutDisplayFunc( Display );
    glutReshapeFunc( Resize );
    glutKeyboardFunc( Keyboard );
    glutMouseFunc( MouseButton );
    glutMotionFunc( MouseMotion );
    glutTimerFunc( -1, NULL, 0 );
    glutIdleFunc( NULL );
}
```

InitGraphics(), II

```c
#define WIN32
#define glewInit() (GLEW_OK)
```

Display(), I

```c
void
Display( )
{
    // set which window we want to do the graphics into:
    glutSetWindow( MainWindow );
    // erase the background:
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    // specify shading to be flat:
    glShadeModel( GL_FLAT );
    // set the viewing volume:
    // remember that the Z clipping values are actually
    // given as DISTANCES IN FRONT OF THE EYE
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity( );
    if( WhichProjection == ORTHO )
        glOrtho( -3., 3.,     -3., 3.,     0.1, 1000. );
    else
        gluPerspective( 90., 1.,        0.1, 1000. );
    // place the objects into the scene:
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity( );
    // set the eye position, look-at position, and up-vector:
    gluLookAt( 0., 0., 3.,     0., 0., 0.,     0., 1., 0. );
    // rotate the scene:
    glRotatef( (GLfloat)Yrot, 0., 1., 0. );
    glRotatef( (GLfloat)Xrot, 1., 0., 0. );
    // uniformly scale the scene:
    if( Scale < MINSCALE )
        Scale = MINSCALE;
    glScalef( (GLfloat)Scale, (GLfloat)Scale, (GLfloat)Scale );
    // set the fog parameters:
    // (DepthCueOn != 0 )
    {
        if( DepthCueOn != 0 )
            {
                glFogi( GL_FOG_MODE, FOGMODE );
                glFogfv( GL_FOG_COLOR, FOGCOLOR );
                glFogf( GL_FOG_DENSITY, FOGDENSITY );
                glFogf( GL_FOG_START, FOGSTART );
                glFogf( GL_FOG_END, FOGEND );
                glEnable( GL_FOG );
            }
        else
            {
                glDisable( GL_FOG );
            }
        // possibly draw the axes:
        if( AxesOn != 0 )
            {
                glCallList( AxesList );
            }
    }
    // draw the current object:
    glCallList( BoxList );
    // Replay the graphics commands from a
    // previously-stored Display List.
    DisplayLists have their own indestructible
    Display() .
```
/* draw some gratuitous text that just rotates on top of the scene: */
glDisable( GL_DEPTH_TEST );
gColor3f( 0., 1., 1. );
DoRasterString( 0., 1., 0., "Text That Moves" );

/* draw some gratuitous text that is fixed on the screen: */
/* the projection matrix is reset to define a scene whose */
/* world-coordinate system goes from 0-100 in each axis */
/* this is called "percent units", and is just a convenience */
/* the modelview matrix is reset to identity as we don't */
/* want to transform these coordinates */
glDisable( GL_DEPTH_TEST );
gMatrixMode( GL_PROJECTION );
gLoadIdentity( );
gluOrtho2D( 0., 100.,     0., 100. );
gMatrixMode( GL_MODELVIEW );
gLoadIdentity( );
gColor3f( 1., 1., 1. );
DoRasterString( 5., 5., 0., "Text That Doesn't" );

/* swap the double-buffered framebuffers: */
glutSwapBuffers( );

/* be sure the graphics buffer has been sent: */
/* note: be sure to use glFlush() here, not glFinish()! */
gFlush( );

Display( ), IV

Extra Topics:
(Don't need this to get started with OpenGL programming)

Subtractive Colors (CMYK)

---

Sidebar: Subtractive Colors (CMYK)

---

Sidebar: Hue-Saturation-Value (HSV) -- Another way to specify additive color

---

The OSU ColorPicker Program

---
Sidebar: How Did We Make the Transition from Vertices to Pixels?

There is a piece of hardware called the Rasterizer. Its job is to interpolate a line or polygon, defined by vertices, into a collection of fragments. Think of it as filling in squares on graph paper.

A fragment is a “pixel-to-be.” In computer graphics, “pixel” is defined as having its full RGBA already computed. A fragment does not yet but all of the information needed to compute the RGBA is there.

A fragment is turned into a pixel by the fragment processing operation.

In CS 457/557, you will do some pretty snazzy things with your own fragment processing code!