The Basic Computer Graphics Pipeline

Geometry vs. Topology

3D Coordinate Systems
Since Homer Simpson uses Right-handed Coordinates, then we will too.

Right-handed 3D Coordinate System for a CNC Machine

Right-handed Positive Rotations

Drawing in 3D

This is a wonderfully understandable way to start with 3D graphics, but it is also incredibly inefficient! We’ll talk about that later…
OpenGL Topologies

GL_POINTS

GL_LINES

GL_LINE_STRIP

GL_LINE_LOOP

GL_TRIANGLES

GL_TRIANGLE_STRIP

GL_TRIANGLE_FAN

GL_QUADS

GL_QUAD_STRIP

GL_POLYGON

OpenGL Topologies – Polygon Requirements

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 -- Orientation

Polygons are traditionally:
- CCW when viewed from outside the solid object

It doesn’t matter much, but there is an advantage in being consistent.
OpenGL Topologies – Vertex Order Matters

GL_LINE_LOOP

V₀ V₁ V₂ V₃
V₂ V₃ V₀

Probably what you meant to do
 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: Stays within the polygon
Not Convex: 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

```c
setColor3f( r, g, b );

begin( GL_LINE_LOOP );

gleVertext3f( x0, y0, 0. );
gleVertext3f( x1, y1, 0. );

end( );
```

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.

```c
setColor3f( r, g, b );

float dang = 2. * M_PI / (float)( NUMSEGS – 1 );
float ang = 0.;

begin( GL_LINE_LOOP );

for( int i = 0; i < NUMSEGS; i++ )
{

gleVertext3f( RADIUS*cos(ang), RADIUS*sin(ang), 0. );
ang += dang;
}

end( );
```

Listing a lot of vertices explicitly gets old in a hurry.

Color

This is referred to as “Additive Color”

Transformations

Translation

Rotation

Scaling
OpenGL Transformations

- `glTranslatef( tx, ty, tz );`
- `glRotatef( degrees, ax, ay, az );`
- `glScalef( sx, sy, sz );`

Single Transformations

- `glMatrixMode( GL_MODELVIEW );`
- `glLoadIdentity();`
- `glRotatef( degrees, ax, ay, az );`
- `glColor3f( r, g, b );`
- `glBegin( GL_LINE_STRIP );`
  - `glVertex3f( x0, y0, z0 );`
  - `glVertex3f( x1, y1, z1 );`
  - `glVertex3f( x2, y2, z2 );`
  - `glVertex3f( x3, y3, z3 );`
  - `glVertex3f( x4, y4, z4 );`
- `glEnd( );`

Why do the Compound Transformations Take Effect in Reverse Order?

These transformations “add up”, and look like they take effect in this order:

1. `glLoadIdentity();`
2. `glTranslatef( tx, ty, tz );`
3. `glRotatef( degrees, ax, ay, az );`
4. `glScalef( sx, sy, sz );`

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

Order Matters!
Compound Transformations are Not Commutative

Translate, then rotate

Rotate, then translate

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:

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

Yuch! 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.

Set the state first

Use the state second

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

Projecting an Object from 3D into 2D

Orthographic (or Parallel) projection

```c
glOrtho( xi, xr, yb, yt, zn, zf );
```

Parallel lines remain parallel

Perspective projection

```c
gluPerspective( fovy, aspect, zn, zf );
```

Parallel lines appear to converge

"Vanishing Point"

The OpenGL Drawing State

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

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

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

Perspective is more realistic-looking
**OpenGL Projection Functions**

- `glMatrixMode(GL_PROJECTION);`
- `glLoadIdentity();`
- `glMatrixMode(GL_MODELVIEW);`
- `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);`
- `glBegin(GL_LINE_STRIP);`  
  - `glVertex3f(x0, y0, z0);`  
  - `glVertex3f(x1, y1, z1);`  
  - `glVertex3f(x2, y2, z2);`  
  - `glVertex3f(x3, y3, z3);`  
  - `glVertex3f(x4, y4, z4);`  
- `glEnd();`

**How the Viewing Volumes Look from the Outside**

- `glOrtho(xl, xr, yb, yt, zn, zf);`  
  - Parallel/Orthographic
- `gluPerspective(fovy, aspect, zn, zf);`  
  - Perspective

**The Perspective Viewing Frustum**

- `gluPerspective(fovy, aspect, zn, zf);`

**Arbitrary Viewing**

- `glMatrixMode(GL_PROJECTION);`
- `glLoadIdentity();`
- `gluPerspective(fpvy, aspect, zn, zf);`
- `glMatrixMode(GL_MODELVIEW);`
- `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);`
- `glBegin(GL_LINE_STRIP);`  
  - `glVertex3f(x0, y0, z0);`  
  - `glVertex3f(x1, y1, z1);`  
  - `glVertex3f(x2, y2, z2);`  
  - `glVertex3f(x3, y3, z3);`  
  - `glVertex3f(x4, y4, z4);`  
- `glEnd();`
How Can You Be Sure You See Your Scene?

```c
    gluPerspective( fovy, aspect, zn, zf );
    gluLookAt( ex, ey, ez, lx, ly, lz, ux, uy, uz );
```

Here’s a good way to start:

1. Set `lx,ly,lz` to be the average of all the vertices
2. Set `ux,uy,uz` to be 0.,1.,0.
3. Set `ex=lx` and `ey=ly`
4. Now, you change $\Delta E$ or `fovy` so that the object fits in the viewing volume:

$$\tan\left(\frac{fovy}{2}\right) = \frac{H/2}{\Delta E}$$

Giving:

$$\Delta E \cdot \tan\left(\frac{fovy}{2}\right) = \frac{H}{2}$$

or:

$$\Delta E = \frac{H}{2 \tan\left(\frac{fovy}{2}\right)}$$

Specifying a Viewport

```c
    glViewport( ixl, iyb, idx, idy );
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity( );
    gluPerspective( fovy, aspect, zn, zf );
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity( );
    gluLookAt( ex, ey, ez, lx, ly, lz, ux, uy, uz );
```

Saving and Restoring the Current Transformation

```c
    glViewport( ixl, iyb, idx, idy );
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity( );
    gluPerspective( fovy, aspect, zn, zf );
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity( );
    glPushMatrix( );
    glRotatef( degrees, ax, ay, az );
    glScalef( sx, sy, sz );
    glColor3f( r, g, b );
    glBegin( GL_LINE_STRIP );
    glVertex3f( x0, y0, z0 );
    glVertex3f( x1, y1, z1 );
    glVertex3f( x2, y2, z2 );
    glVertex3f( x3, y3, z3 );
    glVertex3f( x4, y4, z4 );
    glEnd( );
    glPopMatrix( );
```
sample.cpp Program Structure

- `#includes`
- Constants 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)
#include "glew.h"
#endif
#include <GL/gl.h>
#include <GL/glu.h>
#include "glut.h"
```

### Constants and `#defines`

- `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 };
```

**Consts are always preferred over `#defines`.** But, Visual Studio does not allow consts to be used in case statements or as array sizes.

### Initialized Global Variables

- `const GLfloat BACKCOLOR[ ] = { 0., 0., 0., 1. };
- `const GLfloat AXES_WIDTH = { 3. };
- `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 FOCOLOR[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

- `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 DepthCueOn;` // != 0 means to use intensity depth cueing
- `GLuint BoxList;` // 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

- `void Animate( );`
- `void Display( );`
- `void DoAxesMenu( int );`
- `void DoColorMenu( int );`
- `void DoDepthMenu( int );`
- `void DoDebugMenu( int );`
- `void DoMainMenu( int );`
- `void DoProjectMenu( int );`
- `void DoRotateString( float, float, float, char * );`
- `void DoStrokeString( 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, int );`
- `void Reset( );`
- `void Resize( int );`
- `void Set( int );`
- `void Visibility( int );`
- `void Axes( float );`
- `void HsvRgb( float[3], float[3] );`

Main Program

```c
int 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)
    glutMainLoop( );
    // this is here to make the compiler happy:
    return 0;
}
```

InitGraphics( ), I
```c
#ifdef WIN32
    GLenum err = glewInit();
    if (err != GLEW_OK)
        printf(stderr, "glewInit Error
    
#endif

void Display()
{
    if (WhichWindow == MainWindow)
        glutSetWindow(MainWindow);
    if (WhichProjection == ORTHO)
        gluOrtho2D(-3.0, 3.0, -3.0, 3.0, 0.1, 1000.0);
    else
        gluPerspective(90.0, 1.0, 0.1, 1000.0);
    if (AxesOn != 0)
    {
        glColor3fv(&Colors[WhichColor][0]);
        glCallList(AxesList);
    }
    else
    {
        glDisable(GL_FOG);
    }
    if (DepthCueOn != 0)
    {
        if (WhichColor == 0)
            glCallList(AxesList);
        else
            glCallList(BoxList);
    }
    else
    {
        glCallList(BoxList);
    }
Extra Topics: (Don’t need to get started with OpenGL programming)

Subtractive Colors (CMYK)

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

float hsv[3], rgb[3];
HsvRgb( hsv, rgb );
glColor3fv( rgb );

The HsvRgb function is in your sample code

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!