Geometry vs. Topology

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

Geometry = changed
Topology = same (1-2-3-4-1)

Geometry = same
Topology = changed (1-2-4-3-1)

Since Homer Simpson uses Right-handed Coordinates, then we will too

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

- X
- Y
- Z

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 state's display-characteristics
  Here is the topology to be used with these vertices

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_TRIANGLE_STRIP
- GL_TRIANGLE_FAN
- GL_QUAD_STRIP
- GL_QUADS
- GL_POLYGON

OpenGL Topologies – Polygon Requirements

- 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

Probably what you meant to do

GL_LINE_LOOP

V0 V3 V1 V2

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

Listing a lot of vertices explicitly gets old in a hurry

```c
void display() {
    glColor3f( r, g, b );
    glBegin( GL_LINE_LOOP );
    glVertex3f( x0, y0, 0. );
    glVertex3f( x1, y1, 0. );
    ... // More vertices
    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.

```c
void display() {
    glColor3f( r, g, b );
    glBegin( GL_LINE_LOOP );
    glVertex3f( x0, y0, 0. );
    float dang = 2. * M_PI / (float)( NUMSEGS – 1 );
    float ang = 0.;
    for( int i = 0; i < NUMSEGS; i++ )
        glVertex3f( RADIUS*cos(ang),  RADIUS*sin(ang),  0. );
    ang += dang;
    glEnd( );
}
```
Color

```
glColor3f(r, g, b);
0.0 ≤ r, g, b ≤ 1.0
```

This is referred to as “Additive Color”

```
Red
Yellow
Magenta
White
Cyan
Blue
```

Cyan = Green + Blue
Magenta = Red + Blue
Yellow = Red + Green
White = Red + Green + Blue

OpenGL Transformations

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
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();
```

Single Transformations

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(tx, ty, tz);
glRotatef(degrees, ax, ay, az);
glScalef(sx, sy, sz);
glColor3f(r, g, b);
glBegin(GL_LINE_STRIP);
```

Compound Transformations

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
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();
```

Why do the Compound Transformations Take Effect in Reverse Order?

```
glTranslatef(tx, ty, tz);
glRotatef(degrees, ax, ay, az);
glScalef(sx, sy, sz);
glColor3f(r, g, b);
glBegin(GL_LINE_STRIP);
```

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

1. 3.
2. 2.
1.

These transformations “add up”, and look like they take effect in this order.
Order Matters! Compound Transformations are Not Commutative

Y
Write, 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.

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

glOrtho( x1, x2, y1, y2, z1, z2 );

Perspective projection

gluPerspective( fovy, aspect, zn, zf );

Parallel lines remain parallel

Parallel lines appear to converge

"Vanishing Point"

Projecting on Object from 3D to 2D

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

Perspective is more realistic-looking

OpenGL Projection Functions

Use one of these, but not both!

OpenGL Projection Functions

How the Viewing Volumes Look from the Outside

Parallel/Orthographic:

Perspective:
The Perspective Viewing Frustum

gluPerspective( fovy, aspect, zn, zf );

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

Arbitrary Viewing

glMatrixMode( GL_PROJECTION );
gLoadIdentity( );
gluPerspective( fovy, aspect, zn, zf );
gMatrixMode( GL_MODELVIEW );
gLoadIdentity( );
gluLookAt( ex, ey, ez, lx, ly, lz, ux, uy, uz );
gTranslatef( tx, ty, tz );
gRotatef( degrees, ax, ay, az );
gScalef( sx, sy, sz );
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( );

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

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 `ΔE` or `fovy` so that the object fits in the viewing volume:

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

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

Specifying a Viewport

Be sure the x:y aspect ratios match!

Specifying a Viewport

Note: setting the viewport is not part of setting either the Modelview or the Projection transformations.

Saving and Restoring the Current Transformation

gViewport( ixl, iyb, idx, idy );
gMatrixMode( GL_PROJECTION );
gLoadIdentity( );
gluPerspective( fovy, aspect, zn, zf );
gMatrixMode( GL_MODELVIEW );
gLoadIdentity( );
gluLookAt( ex, ey, ez, lx, ly, lz, ux, uy, uz );
gTranslatef( tx, ty, tz );
gPushMatrix( );
gRotatef( degrees, ax, ay, az );
gScalef( sx, sy, sz );
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( );
gPopMatrix( );
**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)
#include "glew.h"
#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 };
```

### Initialized Global Variables

```c
const GLfloat BACKCOLOR[ ] = { 0., 0., 0., 1. };
const GLfloat AXES_WIDTH   = { 3. };
char * ColorNames[ ] =
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
```

### 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 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

```c
void    Animate( );
void    Display( );
void    DoAxesMenu( int );
void    DoColorMenu( int );
void    DoDepthMenu( 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
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( )
    Reset( );
    // setup all the user interface stuff:
    InitMenus( );
    // draw the scene once and wait for some interaction:
    // (this will never return)
    glutSetWindow( MainWindow );
    glutMainLoop( );
    return 0;
}
```
Display() - IV

// draw some gratuitous text that just rotates on top of the scene:
glDisable(GL_DEPTH_TEST);
setColor(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 identify as well.
// want to transform these coordinates

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

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

Subtractive Colors (CMYK)

Additive Colors:

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!