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

Geometry vs. Topology

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

Topology: How things are connected

Homer Simpson uses Right-handed Coordinates. Who are we to argue with Homer Simpson?

Right-handed 3D Coordinate System for a CNC Machine
Right-Handed Coordinate System

Drawing in 3D

```c
gColor3f( r, g, b );
Begin( GL_LINE_STRIP);
gVertex3f( x0, y0, z0 );
gVertex3f( x1, y1, z1 );
gVertex3f( x2, y2, z2 );
gVertex3f( x3, y3, z3 );
gVertex3f( x4, y4, z4 );
End();
```

This is a wonderfully understandable way to start with 3D graphics – it is like holding a marker in your hand and sweeping out linework in the 3D air in front of you!

But it is also incredibly internally inefficient! We’ll talk about that later and what to do about it…

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

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

*GL_LINE_STRIP* and *GL_TRIANGLES* are considered to be 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 V1 V2 V3

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

Not Convex

Stays within the polygon

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

Not Convex

2 edge intersections

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

OpenGL Drawing Can Be Done Procedurally

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.

Listing a lot of vertices explicitly gets old in a hurry

glColor3f( r, g, b );
gBegin( GL_LINE_LOOP );
gVertex3f( x0, y0, 0. );
. . .
gEnd();

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

\[ \text{glColor3f}(r, g, b) ; \]
\[ 0.0 \leq r, g, b \leq 1.0 \]

This is referred to as “Additive Color”

\[ \text{Red} \]
\[ \text{Yellow} \]
\[ \text{Green} \]
\[ \text{Cyan} = \text{Green} + \text{Blue} \]
\[ \text{Magenta} = \text{Red} + \text{Blue} \]
\[ \text{Yellow} = \text{Red} + \text{Green} \]
\[ \text{White} = \text{Red} + \text{Green} + \text{Blue} \]

OpenGL Transformations

\[ \text{glTranslatef}(tx, ty, tz) ; \]
\[ \text{glRotatef}(\text{degrees}, ax, ay, az) ; \]
\[ \text{glScalef}(sx, sy, sz) ; \]

Single Transformations

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

Compound Transformations

\[ \text{glMatrixMode}(\text{GL_MODELVIEW}) ; \]
\[ \text{glLoadIdentity}() ; \]
\[ \text{glTranslatef}(tx, ty, tz) ; \]
\[ \text{glRotatef}(\text{degrees}, ax, ay, az) ; \]
\[ \text{glScalef}(sx, sy, sz) ; \]
\[ \text{glColor3f}(r, g, b) ; \]
\[ \text{glBegin}(\text{GL_LINE_STRIP}) ; \]
\[ \text{glVertex3f}(x0, y0, z0) ; \]
\[ \text{glVertex3f}(x1, y1, z1) ; \]
\[ \text{glVertex3f}(x2, y2, z2) ; \]
\[ \text{glVertex3f}(x3, y3, z3) ; \]
\[ \text{glVertex3f}(x4, y4, z4) ; \]
\[ \text{glEnd}() ; \]

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

1. 2. 3.
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
Draw with that 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( xl, xr, yb, yt, zn, zf );
```

Perspective projection

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

Parallel lines remain parallel

Parallel lines appear to converge

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

Perspective is more realistic-looking

OpenGL Projection Functions

```c
glMatrixMode( GL_PROJECTION );
gLoadIdentity();

glMatrixMode( GL_MODELVIEW );
gLoadIdentity();

gluLookAt( ex, ey, ez, lx, ly, lz, ux, uy, uz );
gLoadIdentity();

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();
```

https://www.gocomics.com/rubes

"The vanishing point? ... It’s straight ahead. You can’t miss it."

The Vanishing Point

Use one of these, but not both!

https://www.gocomics.com/rubes
How the Viewing Volumes Look from the Outside

```
glOrtho( xl, xr, yb, yt, zn, zf);
gluPerspective( fovy, aspect, zn, zf);
```

### Parallel/Orthographic

```
X -> y top
Y -> y bottom
Z -> znear
```

### Perspective

```
X -> y top
Y -> y bottom
Z -> znear
```

The Perspective Viewing Frustum

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

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

Arbitrary Viewing

```
gluMatrixMode( GL_PROJECTION );
gluLoadIdentity();
gluPerspective( fovy, aspect, zn, zf );
gluMatrixMode( GL_MODELVIEW );
gluLoadIdentity();
gluLookAt( ex, ey, ez, lx, ly, lz, ux, uy, uz );
gluMatrixMode( GL_PROJECTION );
gluLoadIdentity();
gluOrtho( xl, xr, yb, yt, zn, zf );
gluMatrixMode( GL_MODELVIEW );
gluLoadIdentity();
gluLookAt( ex, ey, ez, lx, ly, lz, ux, uy, uz );
gluMatrixMode( GL_PROJECTION );
gluLoadIdentity();
gluOrtho( xl, xr, yb, yt, zn, zf );
```

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

How Can You Be Sure You See Your Scene?

```
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 ΔE or fovy so that the object fits in the viewing volume:
   - **fovy** = 2 * arctan( H / 2 * ΔE )
   - **ΔE** = \( \frac{H}{2 \tan( \frac{\text{fovy}}{2})} \)

Specifying a Viewport

```
glViewport( ixl, iyb, idx, idy );
gluPerspective( fovy, aspect, zn, zf );
gluMatrixMode( GL_MODELVIEW );
gluLoadIdentity();
gluLookAt( ex, ey, ez, lx, ly, lz, ux, uy, uz );
gluMatrixMode( GL_PROJECTION );
gluLoadIdentity();
gluOrtho( xl, xr, yb, yt, zn, zf );
gluMatrixMode( GL_MODELVIEW );
gluLoadIdentity();
gluLookAt( ex, ey, ez, lx, ly, lz, ux, uy, uz );
gluMatrixMode( GL_PROJECTION );
gluLoadIdentity();
gluOrtho( xl, xr, yb, yt, zn, zf );
gluMatrixMode( GL_MODELVIEW );
gluLoadIdentity();
gluLookAt( ex, ey, ez, lx, ly, lz, ux, uy, uz );
gluMatrixMode( GL_PROJECTION );
gluLoadIdentity();
gluOrtho( xl, xr, yb, yt, zn, zf );
```

Note: setting the viewport is not part of setting either the Modelview or the Projection transformations.
Saving and Restoring the Current Transformation

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

gluLookAt(ux, uy, uz);
gluOrtho2D(-1, 1, -1, 1);
gluOrtho2D(-1, 1, 0, 1);
gluOrtho2D(-1, 1, 0, 1, -1);
gluOrtho2D(-1, 1, 1, 1, -1);

sample.cpp Program Structure

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

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#define _USE_MATH_DEFINES
#include <math.h>

const char *WINDOWTITLE = { "OpenGL / GLUT Sample -- Joe Graphics" };
const char *GLUITITLE = { "User Interface Window" };
const int GLUITRUE = { true };
const int GLUIFALSE = { false };
const int 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

const GLfloat BACKCOLOR[4] = { 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
};
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

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
int BoxList; // object display list
int MainWindow; // window id for main graphics window
float Scale; // scaling factor
int WhichColor; // index into Colors[]
enum Projections WhichProjection; // ORTHO or PERSP
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:
  if( WhichProjection == ORTHO )
     glutOrtho( -3., 3., -3., 3., 0.1, 1000. );
  else
     gluPerspective( 90., 1., 0.1, 1000. );
  // draw the scene once and wait for some interaction:
  // (this will never return)
  glutSetWindow( MainWindow );
  glutMainLoop( );
  return 0;
}
```

InitGraphics( ), I

```c
void InitGraphics( )
{
  // request the display modes:
  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 viewport to a square centered in the window:
  GLsizei vx = glutGet( GLUT_WINDOW_WIDTH );
  GLsizei vy = glutGet( GLUT_WINDOW_HEIGHT );
  GLsizei v = vx < vy ? vx : vy;                  // minimum dimension
  GLint xl = ( vx - v ) / 2;
  GLint yb = ( vy - v ) / 2;
  glViewport( xl, yb, v, v );
  // 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 )
     glutOrtho( -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 );
}
```

InitGraphics( ), II

```c
GLenum err = glewInit( );
if( err != GLEW_OK )
  fprintf( stderr, "glewInit Error\n" );
```

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 );
  glEnable( GL_DEPTH_TEST );
  // set shading to be flat:
  glShadeModel( GL_FLAT );
  // set the viewport to be a square centered in the window:
  GLint w = glutGet( GLUT_WINDOW_WIDTH );
  GLint h = glutGet( GLUT_WINDOW_HEIGHT );
  GLint x = w / 2; GLint y = h / 2;
  if( h < w )
     x = h / 2; y = w / 2;
  glViewport( x, y, w, h );
```

Display( ), II

```c
glMatrixMode( GL_PROJECTION );
glLoadIdentity( );
if( WhichProjection == ORTHO )
   glutOrtho( -3., 3., -3., 3., 0.1, 1000. );
else
   gluPerspective( 90., 1., 0.1, 1000. );
// project 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;
```
Display( ), III

// set the fog parameters:
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 )
{
  glColor3fv( &Colors[WhichColor][0] );
  glCallList( AxesList );
}

// draw the current object:
glCallList( BoxList );

// swap the double-buffered framebuffers:

Display( ), IV

// swap the double-buffered framebuffers:

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, the word "pixel" is defined as having its full RGBA already computed. A fragment does not yet have its final RGBA computed, but all of the information needed to compute the RGBA is available to it.

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!