The Basic Computer Graphics Pipeline

- Model Transform
- View Transform
- Per-vertex Lighting
- Projection Transform
- Homogeneous Division
- Viewport Transform
- Fragment Processing, Texturing, Per-fragment Lighting
- Rasterization
- Raster Ops
- Framebuffer
- EC
- WC
- MC
- CC
- NDC
- SC

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

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

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

3D Coordinate Systems

OpenGL uses this one
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 Positive Rotations

Drawing in 3D

```c
setColor( r, g, b );
begin( GL_LINE_STRIP );
setDisplayState( ... );
Vertex3f( x0, y0, z0 );
Vertex3f( x1, y1, z1 );
Vertex3f( x2, y2, z2 );
Vertex3f( x3, y3, z3 );
Vertex3f( 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 inefficient! We’ll talk about that later and what to do about it…
OpenGL Topologies

GL_POINTS

GL_LINES

GL_LINE_STRIP

GL_TRIANGLES

GL_LINE_LOOP

GL_TRIANGLE_STRIP

GL_TRIANGLE_FAN

GL_QUADS

GL_QUAD_STRIP

GL_POLYGON

OpenGL Topologies – Polygon Requirements

Polygons must be:

• Convex and
• Planar

GL_TRIANGLE_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

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.

Convinx

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

Not Convex

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

Color

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

This is referred to as "Additive Color"
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();`

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?

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

*These transformations “add up”, and look like they take effect in this 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, …   );
```

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

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

Parallel lines remain parallel

Perspective projection

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

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

The Vanishing Point

Perspective is more realistic-looking
OpenGL Projection Functions

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

Use one of these, but not both!

How the Viewing Volumes Look from the Outside

- \texttt{glOrtho( xl, xr, yb, yt, zn, zf );}
- \texttt{gluPerspective( fovy, aspect, zn, zf );}

The Perspective Viewing Frustum

- \texttt{gluPerspective( fovy, aspect, zn, zf );}

- \texttt{aspect = DX/DY}
- \texttt{fovy = vertical field of view angle (degrees)}
  (good values are 50-100°)
### Arbitrary Viewing

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

// Modelview

// Use gluLookAt() to position the eye and up vector.
gluLookAt( ex, ey, ez, lx, ly, lz, ux, uy, uz );

// Use glBegin() and glVertex3f() to draw the object.
```

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

#### Screen shot:

- **Eye Position**
- **Look-at Position**
- **Up vector**

#### Glue Code:

```c
// Setting the eye view
ex = 100; ey = 100; ez = 200;
```

### How Can You Be Sure You See Your Scene?

```c
// Setting up the viewing volume

// Use gluPerspective() to set the viewing volume.
gluPerspective( fovy, aspect, zn, zf );
```

#### Equations:

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

### Specifying a Viewport

```c
glViewport( ixl, iyb, idx, idy );
```

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

```c
glViewport( ixl, iyb, idx, idy );
glMatrixMode( GL_PROJECTION );
gluPerspective( fovy, aspect, zn, zf );
glMatrixMode( GL_MODELVIEW );
gluLookAt( ex, ey, ez, lx, ly, lz, ux, uy, uz );
glLoadIdentity();
glPushMatrix();
glTranslatef( tx, ty, tz );
glRotatef( degrees, ax, ay, az );
glScalef( 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

```c
#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 = 2.0f;
const float ANGFACT = 1.0f;
const float SCLFACT = 0.005f;
const float MINSCALE = 0.05f;
const int LEFT = 4;
const int MIDDLE = 2;
const int RIGHT = 1;
```

consts and #defines

```c
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. };`
- `const GLfloat Colors[8][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`
- `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 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 Reset();`
- `void Resize(int, int);`
- `void Visibility(int);`
- `void Axes();`
- `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();
    // this is here to make the compiler happy:
    return 0;
}
```
```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] );
    glutSetWindow( MainWindow );
    glutDisplayFunc( Display );
    glutReshapeFunc( Resize );
    glutKeyboardFunc( Keyboard );
    glutMouseFunc( MouseButton );
    glutMotionFunc( MouseMotion );
    glutTimerFunc( -1, NULL, 0 );
    glutIdleFunc( NULL );
}

GLenum err = glewInit( );
if( err != GLEW_OK )
{
    fprintf( stderr, "glewInit Error
" );
}
```

```c
void Display( )
{
    // set which window we want to do the graphics into:
    glutSetWindow( MainWindow );
    // erase the background:
    glDrawBuffer( GL_BACK );
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    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( );
    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 );
}
```
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 );

Display( ), IV

// draw some gratuitous text that just rotates on top of the scene:
glDisable( GL_DEPTH_TEST );
glColor3f( 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 );
glMatrixMode( GL_PROJECTION );
glLoadIdentity( );
gluOrtho2D( 0., 100., 0., 100. );
glMatrixMode( GL_MODELVIEW );
glLoadIdentity( );
glColor3f( 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();

The OSU ColorPicker Program

Red, Green, Blue
Hue, Saturation, Value
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!