The Basic Computer Graphics Pipeline

- **Model Transform**
- **View Transform**
- **Per-vertex Lighting**
- **Projection Transform**
- **Homogeneous Division**
- **Viewport Transform**
- **Fragment Processing, Texturing, Per-fragment Lighting**
- **Raster Ops**
- **Rasterization**

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.

**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)

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

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

Topology: How things are connected

3D Coordinate Systems

- Left-handed
- Right-handed

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
setColor3f( r, g, b );

begin(GL_LINE_STRIP);
  glVertex3f( x0, y0, z0 );
  glVertex3f( x1, y1, z1 );
  glVertex3f( x2, y2, z2 );
  glVertex3f( x3, y3, z3 );
  glVertex3f( 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_POINTS

GL_LINES

GL_LINE_STRIP

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

V₀  V₁  V₂  V₃

GL_LINE_LOOP

V₀  V₂  V₁  V₃

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

V₀  V₁  V₂  V₃

Not Convex

V₀  V₁  V₂  V₃

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

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.

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.

Color

This is referred to as “Additive Color”.

Transformations

Translation

Rotation

Scaling
OpenGL Transformations

```
glTranslatef( tx, ty, tz );

// Rotation

glRotatef( degrees, ax, ay, az );

// Scaling

glScalef( sx, sy, sz );
```

Single Transformations

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

// Rotation

glRotatef( degrees, ax, ay, az );

// Color

gColor3f( r, g, b );

// Rendering

glBegin( GL_LINE_STRIP );
gVertex3f( x0, y0, z0 );
gVertex3f( x1, y1, z1 );
gVertex3f( x2, y2, z2 );
gVertex3f( x3, y3, z3 );
gVertex3f( x4, y4, z4 );
gEnd( );
```

Compound Transformations

```
glMatrixMode( GL_MODELVIEW );
gLoadIdentity( );

// Translation

glTranslatef( tx, ty, tz );

// Rotation

glRotatef( degrees, ax, ay, az );

// Scaling

gScalef( sx, sy, sz );

// Color

gColor3f( r, g, b );

// Rendering

glBegin( GL_LINE_STRIP );
gVertex3f( x0, y0, z0 );
gVertex3f( x1, y1, z1 );
gVertex3f( x2, y2, z2 );
gVertex3f( x3, y3, z3 );
gVertex3f( x4, y4, z4 );
gEnd( );
```

Why do the Compound Transformations Take Effect in Reverse Order?

```
1.
2.
3. These transformations “add up”, and look like they take effect in this order.
```
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:

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.

1. Set the state

2. Draw with that state

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

Projecting an Object from 3D into 2D

Orthographic (or Parallel) projection

glOrtho( xL, 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

```cpp
gMatrixMode( GL_PROJECTION );
gLoadIdentity( );

if( WhichProjection == ORTHO )
gOrtho( -2.f, 2.f, -2.f, 2.f, 0.1f, 1000.f );
else
  gluPerspective( 70.f, 1.f, 0.1f, 1000.f );

MatrixMode( GL_MODELVIEW );
gLoadIdentity( );
gluOrtho2D( xleft, xright, ybottom, ytop );
gluPerspective( fovy, aspect, znear, zfar );
```

Use one of (gOrtho, gluPerspective), but not both!

How the Viewing Volumes Look from the Outside
The Perspective Viewing Frustum

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

- \( fovy \) = vertical field of view angle (degrees)
- \( \text{good values are 50-100°} \)

Arbitrary Viewing

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

Eye Position

Look-at Position

Up vector

How Can You Be Sure You See Your Scene?

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

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 \( ex, ey, ez \) to be \( 0 \)
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:

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

or

\[ \Delta E = \frac{H}{2\tan \left( \frac{fovy}{2} \right)} \]
### Specifying a Viewport

Be sure the y:x aspect ratios match!!

```c
void SpecView( 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 );
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( );
```

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

### Saving and Restoring the Current Transformation

```c
void SaveView( ixl, iyb, idx, idy );
glMatrixMode( GL_PROJECTION );
gluPerspective( fovy, aspect, zn, zf );
glMatrixMode( GL_MODELVIEW );
gluLoadIdentity();
gluLookAt( ex, ey, ez, lx, ly, lz, ux, uy, uz );
glTranslatef( tx, ty, tz );
glPushMatrix( );
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>
#ifdef WIN32
#include <windows.h>
#pragma warning(disable:4996)
#include "glew.h"
#endif
#include <GL/gl.h>
#include <GL/glu.h>
#include "glut.h"
```

### #includes

- #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**

- `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;`
- `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. }, { 1., 1., 0. }, { 0., 1., 0. }, { 0., 1., 1. }, { 0., 0., 1. }, { 1., 0., 1. }, { 1., 1., 1. }, { 0., 0., 0. } };`
- `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 MouseMotion(int, 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();
    // 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( )

```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:
    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
" );
    }
}
```

Display( )

```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 );
    // specify shading to be flat:
    glShadeModel( GL_FLAT );
    // set the viewport to a square centered in the window:
    GLsizei vx = glutGet( GLUT_WINDOW_WIDTH );
    GLsizei vy = glutGet( GLUT_WINDOW_HEIGHT );
    GLint x = vx / 2;
    GLint y = vy / 2;
    // minimum dimension
    glViewport( x, y, vx, vy );
    // draw the scene:
    glutPostRedisplay();
}
```
// 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 )
grOrtho( -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:
gfRotatef( (GLfloat)Yrot, 0., 1., 0. );
gfRotatef( (GLfloat)Xrot, 1., 0., 0. );

// uniformly scale the scene:
if( Scale < MINSCALE )
    Scale = MINSCALE;
gfScalef( (GLfloat)Scale, (GLfloat)Scale, (GLfloat)Scale );

// set the fog parameters:
if( DepthCueOn != 0 )
{
    gfFogi( GL_FOG_MODE, FOGMODE );
gfFogfv( GL_FOG_COLOR, FOGCOLOR );
gfFogf( GL_FOG_DENSITY , FOGDENSITY );
gfFogf( GL_FOG_START, FOGSTART );
gfFogf( GL_FOG_END, FOGEND );
gfEnable( GL_FOG );
}
else
{
    gfDisable( GL_FOG );
}

// possibly draw the axes:
if( AxesOn != 0 )
{
    fColor3fv( &Colors[WhichColor][0] );
gfCallList( AxesList );
}

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

Display( ), II

// draw some gratuitous text that just rotates on top of the scene:

gfDisable( GL_DEPTH_TEST );
gfColor3f( 0., 1., 1. );
gfDoRasterString( 0., 1., 0., "Text That Moves" );

// draw some gratuitous text that is fixed on the screen
// the projection matrix is set 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

gfDisable( GL_DEPTH_TEST );
gfMatrixMode( GL_PROJECTION );
gfLoadIdentity();
gfOrtho(0., 100., 0., 100., 0., 100. );
gfMatrixMode( GL_MODELVIEW );
gfLoadIdentity();
gfColor3f( 1., 1., 1. );
gfDoRasterString( 5., 5., 0., "Text That Doesn't" );

// swap the double-buffered framebuffers:

gfSwapBuffers();

(x,y,z), to be translated by the ModelView matrix

Display( ), IV

Display( ), III

// swap the double-buffered framebuffers:

gfSwapBuffers();

// swap the double-buffered framebuffers:

gfSwapBuffers();

You draw into here

This is called the update

The monitor displays from here

"swap buffers" changes the role of the two framebuffers

You draw into here

This is called the refresh

The monitor displays from here

Display Lists have their own noteset.
The OSU ColorPicker Program

Red, Green, Blue
Hue, Saturation, Value

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!