The Basic Computer Graphics Pipeline

MC → WC → KC → KC → KC → KC → Normalized Device
Homogeneous Division
Viewport
Transform
Fragment Processing, Texturing, Per-Fragment Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Per-Fragment
Lighting
Rasterization
Viewport
Transform
Homogeneous Division
Fragment
Processing,
Texturing,
Right-Handed Coordinate System

Right-handed Positive Rotations

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 – 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…

OpenGL Topologies

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

Probably what you meant to do

GL_LINE_LOOP

V3 V0 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

V0 V1 V2 V3

Stays within the polygon

Not Convex

V0 V1 V2 V3

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

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

Stays within the polygon

Not Convex

V0 V1 V2 V3

Leaves the polygon

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.

OK

OK

Not OK

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.

OpenGL Topologies – Vertex Order Matters

GL_LINE_LOOP

V0 V1 V2 V3

Probably what you meant to do

GL_LINE_LOOP

V3 V0 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

V0 V1 V2 V3

Stays within the polygon

Not Convex

V0 V1 V2 V3

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

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

Stays within the polygon

Not Convex

V0 V1 V2 V3

Leaves the polygon

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.

OK

OK

Not OK

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"

Color

Red

Yellow

Cyan

Green

Blue

Magenta

White

Cyan = Green + Blue

Magenta = Red + Blue

Yellow = Red + Green

White = Red + Green + Blue

glColor3f( r, g, b );

0.0 ≤ r, g, b ≤ 1.0

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( );

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( );

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

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:

\[ \text{glVertex3f}( x, y, z, r, g, b, m00, ..., m33, s, t, nx, ny, nz, \text{linewidth}, ... ); \]

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

\[ \text{glOrtho}( x_l, x_r, y_b, y_t, z_n, z_f ); \]

Perspective projection

\[ \text{gluPerspective}( \text{fovy}, \text{aspect}, z_n, z_f ); \]

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

Perspective is more realistic-looking

The Vanishing Point

https://www.gocomics.com/rubes

OpenGL Projection Functions

\[ \text{glMatrixMode}( \text{GL_PROJECTION} ); \]
\[ \text{glLoadIdentity}(); \]

\[ \text{glMatrixMode}( \text{GL_MODELVIEW} ); \]
\[ \text{glLoadIdentity}(); \]

\[ \text{gluLookAt}( \text{ex}, \text{ey}, \text{ez}, \text{lx}, \text{ly}, \text{lz}, \text{ux}, \text{uy}, \text{uz} ); \]
\[ \text{glTranslatef}( \text{tx}, \text{ty}, \text{tz} ); \]
\[ \text{glRotatef}( \text{degrees}, \text{ax}, \text{ay}, \text{az} ); \]
\[ \text{glScalef}( \text{sx}, \text{sy}, \text{sz} ); \]
\[ \text{glColor3f}( \text{r}, \text{g}, \text{b} ); \]

\[ \text{glBegin}( \text{GL_LINE_STRIP} ); \]
\[ \text{glVertex3f}( \text{x0}, \text{y0}, \text{z0} ); \]
\[ \text{glVertex3f}( \text{x1}, \text{y1}, \text{z1} ); \]
\[ \text{glVertex3f}( \text{x2}, \text{y2}, \text{z2} ); \]
\[ \text{glVertex3f}( \text{x3}, \text{y3}, \text{z3} ); \]
\[ \text{glVertex3f}( \text{x4}, \text{y4}, \text{z4} ); \]
\[ \text{glEnd}(); \]

Use one of (glOrtho, gluPerspective), but not both!
OpenGL Projection Functions

```c
glMatrixMode(GL_PROJECTION);
glLoadIdentity();

if( WhichProjection == ORTHO )
gluOrtho2D(xl, xr, yb, yt);  // ORTHO
else

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

How the Viewing Volumes Look from the Outside

Parallel/Orthographic

Perspective

Arbitrary Viewing

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

How Can You Be Sure You See Your Scene?

```c
float deltaE = fabs(H/2 * tan(fovy/2));
```

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 and ey to be 0
4. Now, you change \( \Delta E \) or fovy so that the object fits in the viewing volume:

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

or:

\[
\text{fovy} = 2 \arctan \left( \frac{H}{2 \Delta E} \right)
\]
### Sample Program Structure

- `#includes`
- `const` and `#defines`
- Global variables
- Function prototypes
- Main program
- InitGraphics function
- 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

```c
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
{
ORTHOGONAL,
PERSPECTIVE
};
enum ButtonVals
{
RESET,
QUIT
};
enum Colors
{
RED,
GREEN,
YELLOW,
CYAN,
BLUE,
MAGENTA,
WHITE,
BLACK
};
```
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
- 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 );
- 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 a buffering
    gluHints( GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH );
    // set the initial window configuration:
    gluWindowPosition( 0, 0 );
    gluWindowPosition( 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 );
}
```

Display()

```c
void Display( )
{
    // set which window we want to do the graphics into:
    gluSetWindow( MainWindow );
    // erase the background:
    glDrawBuffer( GL_BACK );
    glClearColor( GL_BACK );
    glEnable( GL_DEPTH_TEST );
    // specify shading to be flat:
    glShadeModel( GL_FLAT );
    // set the viewport to a square centered in the window:
    GLdouble vx = glutGet( GLUT_WINDOW_WIDTH );
    GLdouble vy = glutGet( GLUT_WINDOW_HEIGHT );
    GLdouble x = (vx - vy) / 2;
    GLdouble y = (vy - vx) / 2;
    glutSetWindow( x, y, x, y );
}
```
// set the viewing volume:
// remember that the Z clipping values are actually
// distances in front of the eye
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:

// set the eye position, look-at position, and up-vector:

// rotate the scene:

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

// 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:

// Replay the graphics commands from a
// previously-stored Display List.

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

// 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

// display the double-buffered framebuffers:

// swap the double-buffered framebuffers:

// draw the graphics commands from a
// previously-stored Display List.

// Display Lists have their own noteset.
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!