Lecture 4 of 41

Lab 1a:
OpenGL Basics

William H. Hsu
Department of Computing and Information Sciences, KSU

KSOL course pages: **http://snipurl.com/1y5gc**
Course web site: **http://www.kddresearch.org/Courses/CIS636**
Instructor home page: **http://www.cis.ksu.edu/~bhsu**

Readings:
SIGGRAPH 2000 OpenGL tutorial (Shreiner, Angel, Shreiner): **http://bit.ly/fVBjk8**
Today: Chapter 2 (review), Chapter 16, Eberly *2e* – see **http://snurl.com/1ye72**
Next class: Section 2.3 (esp. 2.3.7), 2.6, 2.7, Eberly 2e
Angel, *OpenGL: A Primer*, 3e
This week: FVFH slides on Viewing

Lecture Outline

- **Reading for Last Class: §2.3 (esp. 2.3.4), Eberly 2[e]; Foley *et al.* Slides**
- **Reading for Today: Chapters 2, 16, Eberly 2[e]; Foley *et al.* Slides**
- **Reading for Next Class: §2.3 (esp. 2.3.7), 2.6, 2.7, Eberly 2[e]**
- **Last Time: Matrix Stack for 3-D Viewing Transformation**
  - $N = D_{Persp} S_{Far} S_{xy} M_{Rot} T_{Trans}$
  - **Perspective: optical principles, terminology**
- **Today: Highlights from First of Three Tutorials on OpenGL (Three Parts)**
  - **1. OpenGL and GL Utility Toolkit (GLUT) – V. Shreiner**
  - **2. Basic rendering – V. Shreiner**
  - **3. 3-D viewing setup – E. Angel**
- **Next Class: Scan Conversion (Rasterization) of Lines, Polygons**

Where We Are

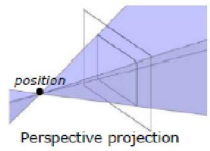
Lecture	Topic	Primary Source(s)
0	Course Overview	Chapter 1, Eberly 2[e]
1	**CG Basics: Transformation Matrices; Lab 0**	**Sections (§) 2.1, 2.2**
2	Viewing 1: Overview, Projections	§ 2.2.3 – 2.2.4, 2.8
3	Viewing 2: Viewing Transformation	§ 2.3 esp. 2.3.4; FVFH slides
4	**Lab 1a: Flash & OpenGL Basics**	**Ch. 2, 16[1], Angel *Primer***
5	Viewing 3: Graphics Pipeline	§ 2.3 esp. 2.3.7; 2.6, 2.7
6	Scan Conversion 1: Lines, Midpoint Algorithm	§ 2.5.1, 3.1; FVFH slides
7	**Viewing 4: Clipping & Culling; Lab 1b**	**§ 2.3.5, 2.4, 3.1.3**
8	Scan Conversion 2: Polygons, Clipping Intro	§ 2.4, 2.5 esp. 2.5.4, 3.1.6
9	Surface Detail 1: Illumination & Shading	§ 2.5, 2.6.1 – 2.6.2, 4.3.2, 20.2
10	**Lab 2a: Direct3D / DirectX Intro**	**§ 2.7, Direct3D handout**
11	Surface Detail 2: Textures; OpenGL Shading	§ 2.6.3, 20.3 – 20.4, *Primer*
12	Surface Detail 3: Mappings; OpenGL Textures	§ 20.5 – 20.13
13	**Surface Detail 4: Pixel/Vertex Shad.; Lab 2b**	**§ 3.1**
14	Surface Detail 5: Direct3D Shading; OGLSL	§ 3.2 – 3.4, Direct3D handout
15	Demos 1: CGA, Fun; Scene Graphs: State	§ 4.1 – 4.3, **CGA handout**
16	**Lab 3a: Shading & Transparency**	**§ 2.6, 20.1, *Primer***
17	**Animation 1: Basics, Keyframes; HW/Exam**	**§ 5.1 – 5.2**
	Exam 1 review; Hour Exam 1 (evening)	**Chapters 1 – 4, 20**
18	**Scene Graphs: Rendering; Lab 3b: Shader**	**§ 4.4 – 4.7**
19	**Demos 2: SFX; Skinning, Morphing**	**§ 5.3 – 5.5, CGA handout**
20	Demos 3: Surfaces; B-reps/Volume Graphics	§ 10.4, 12.7, **Mesh handout**

Lightly-shaded entries denote the due date of a written problem set; heavily-shaded entries, that of a machine problem (programming assignment); blue-shaded entries, that of a paper review; and the green-shaded entry, that of the term project.

Green, **blue** and **red** letters denote **exam review**, **exam**, and **exam solution review** dates.

Review: Viewing Transformation

- **Placement** of view volume (visible part of world) specified by camera's position and orientation
  - *Position* (a point)
  - *Look* and *Up vectors*
- **Shape** of view volume specified by
  - *horizontal* and *vertical view angles*
  - *front and back clipping planes*
- Perspective projection: projectors intersect at *Position*
- Parallel projection: projectors parallel to *Look vector*, but never intersect (or intersect at infinity)



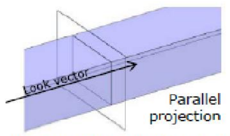
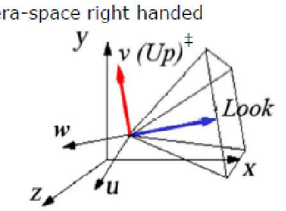


Image Source: Steve Marschner, Cornell

- Coordinate Systems
  - **world coordinates** – standard right-handed *xyz* 3-space
  - **camera coordinates** – camera-space right handed coordinate system (*u*, *v*, *w*); origin at *Position* and axes rotated by orientation; used for transforming arbitrary view into canonical view



Arbitrary Perspective Frustum

‡ v isn't strictly the *Up vector* but the projection of *Up*

Adapted from slides © 1997 – 2010 van Dam *et al.*, Brown University
http://bit.ly/hiSt0f Reused with permission.

Review: CTM for "Polygons-to-Pixels" Pipeline

- Entire problem can be reduced to a composite matrix multiplication of vertices, clipping, and a final matrix multiplication to produce screen coordinates.
- Final composite matrix (*CTM*) is composite of all modeling (instance) transformations (*CMTM*) accumulated during scene graph traversal from root to leaf, composited with the final composite normalizing transformation *N* applied to the root/world coordinate system:

1) $N = D_{persp} S_{far} S_{xy} M_{rot} T_{trans}$

2) $CTM = N \cdot CMTM$

3) $P' = CTM \cdot P$ for every vertex P defined in its own coordinate system

4) $P_{screen} = 512 \cdot P' + 1$ for all **clipped** *P′*

Adapted from slides © 1997 – 2010 van Dam *et al.*, Brown University
http://bit.ly/hiSt0f Reused with permission.

Lab 1 of 7, Part A [1]: Setup, OpenGL/Mesa, & XWindows

CIS 636 Interactive Computer Graphics
CIS 736 Computer Graphics

Lab 1a of 7
OpenGL Setup and Basics

The purpose of this lab exercise is to help you get up and running with Mesa (Linux OpenGL) in the CIS Department's Linux environment and over XWindows, and to show you some basic rendering.

This lab assignment is worth a total of 10 points (1%).

Upload an electronic copy of the assignment in PDF form (converted from your word processor, or scanned) to your K-State Online (KSOL) drop box before the due date and time.

References

NeonHelium tutorials: http://nehe.gamedev.net
Mesa home page: http://www.mesa3d.org
OpenGL FAQ: http://www.opengl.org/resources/faq/
OpenGL viewing docs: http://www.opengl.org/resources/faq/technical/viewing.htm

Problems 4-7 of this lab are adapted from:

Dunham, D. (2008). Lab 5, *CS 5721 (Computer Graphics)*, fall, 2008. Duluth, USA: University of Minnesota. Retrieved from: http://bit.ly/9dLERE

1. **(20%) Modelview transformation: 3-D Rotation of 2-D objects.** Follow Lesson 03 to rotate the flat polygons and then render them. Turn in `lab1_1.c` and `lab1_1.jpg`.

2. **(20%) Modelview transformation: 3-D Rotation of 3-D objects.** Follow Lesson 04 to draw 3-D polyhedra and rotate them. Turn in `lab1_2.c` and `lab1_2.jpg`.

3. **(20%) XWindows.** Repeat Lesson 04 from a notebook computer or PC running Mac OS X, Windows XP, Windows Vista, or Windows 7. Turn in `lab1_3.jpg`.

Lab 1 of 7, Part A [2]: Perspective View Volume Specification

4. (10%) Step 1. Perspective Viewing with OpenGL

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

5. (10%) Step 2: Making things move in Perspective

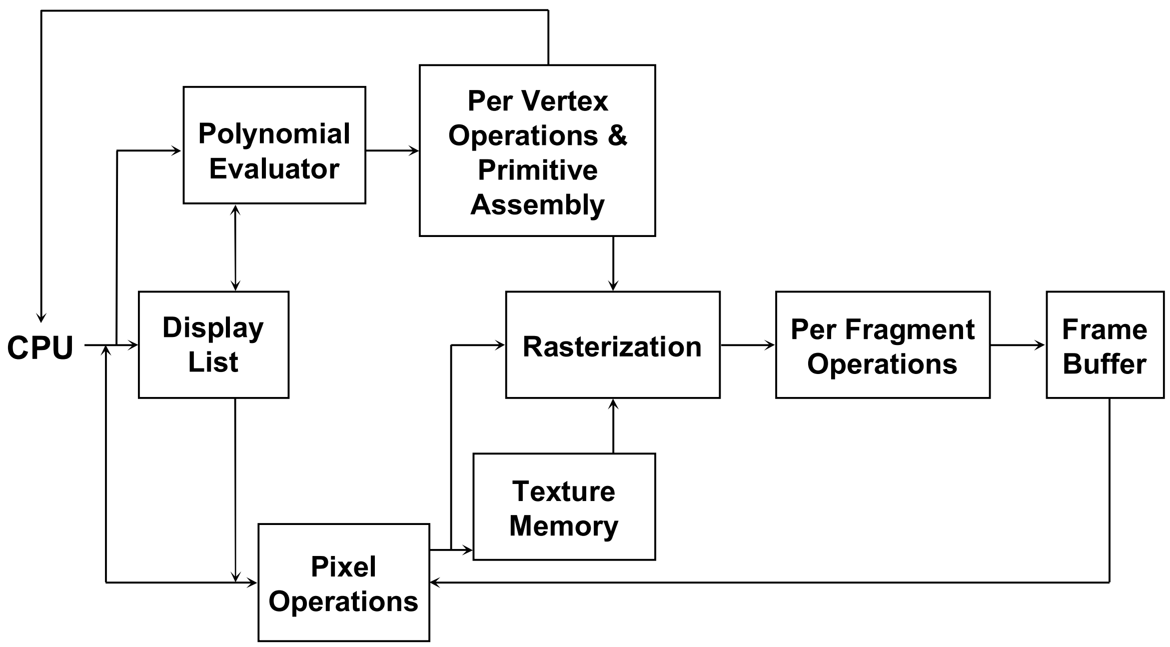
```
glFrustum( -width/2.0, width/2.0, -height/2.0, height/2.0, -1.0,
-20.0 );
gluPerspective( field_of_view, aspect, near, far );
```

6. (10%) Step 3: Specifying the Viewing matrix

```
gluLookAt(  eyeX, eyeY, eyeZ,
            centerX, centerY, centerZ,
            upX, upY, upZ                   );
```

7. (10%) Step 4: Instancing Objects

OpenGL Architecture



© 2000 Shreiner, D., Angel, E., Shreiner, V.

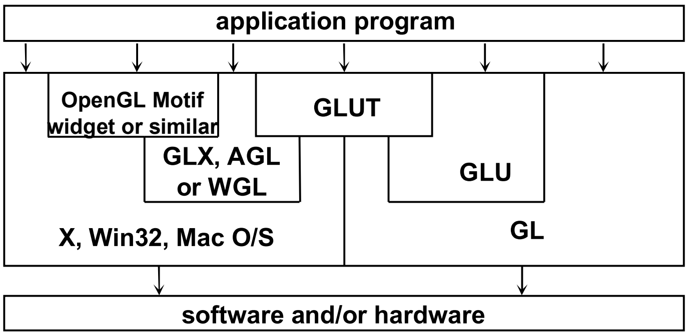
OpenGL Rendering Application Programmer Interface

- Geometric primitives
  - points, lines and polygons
- Image Primitives
  - images and bitmaps
  - separate pipeline for images and geometry
    - linked through texture mapping
- Rendering depends on state
  - colors, materials, light sources, etc.

© 2000 Shreiner, D., Angel, E., Shreiner, V.

OpenGL & Related APIs

- AGL, GLX, WGL
  - glue between OpenGL and windowing systems
- GLU (OpenGL Utility Library)
  - part of OpenGL
  - NURBS, tessellators, quadric shapes, etc.
- GLUT (OpenGL Utility Toolkit)
  - portable windowing API
  - not officially part of OpenGL



Adapted from slides © 2000 Shreiner, D., Angel, E., Shreiner, V.

Preliminaries

- **Headers Files**
  - ⇨ **#include <GL/gl.h>**
  - ⇨ **#include <GL/glu.h>**
  - ⇨ **#include <GL/glut.h>**
- **Libraries**
- **Enumerated Types**
  - **OpenGL defines numerous types for compatibility**
    - ⇨ **GLfloat**
    - ⇨ **GLint**
    - ⇨ **Glenum**
    - ⇨ ***etc.***

© 2000 Shreiner, D., Angel, E., Shreiner, V.

GLUT Callback Functions

- **Application Structure**
  - **Configure and open window**
  - **Initialize OpenGL state**
  - **Register input callback functions**
    - ⇨ **render**
    - ⇨ **resize**
    - ⇨ **input: keyboard, mouse, etc.**
  - **Enter event processing loop**

© 2000 Shreiner, D., Angel, E., Shreiner, V.

Sample Program

```
void main( int argc, char** argv )
{
  int mode = GLUT_RGB|GLUT_DOUBLE;
  glutInitDisplayMode( mode );
  glutCreateWindow( argv[0] );
  init();
  glutDisplayFunc( display );
  glutReshapeFunc( resize );
  glutKeyboardFunc( key );
  glutIdleFunc( idle );
  glutMainLoop();
}
```

© 2000 Shreiner, D., Angel, E., Shreiner, V.

OpenGL Initialization

- **Set up whatever state you're going to use**

```
void init( void )
{
  glClearColor( 0.0, 0.0, 0.0, 1.0 );
  glClearDepth( 1.0 );

  glEnable( GL_LIGHT0 );
  glEnable( GL_LIGHTING );
  glEnable( GL_DEPTH_TEST );
}
```

© 2000 Shreiner, D., Angel, E., Shreiner, V.

GLUT Callback Functions

- Routine to call when something happens
  - window resize or redraw
  - user input
  - animation
- "Register" callbacks with GLUT

```
glutDisplayFunc( display );
glutIdleFunc( idle );
glutKeyboardFunc( keyboard );
```

© 2000 Shreiner, D., Angel, E., Shreiner, V.

Rendering Callback

- **Do all of your drawing here**

```
glutDisplayFunc( display );
```

```
void display( void )
{
  glClear( GL_COLOR_BUFFER_BIT );
  glBegin( GL_TRIANGLE_STRIP );
    glVertex3fv( v[0] );
    glVertex3fv( v[1] );
    glVertex3fv( v[2] );
    glVertex3fv( v[3] );
  glEnd();
  glutSwapBuffers();
}
```

© 2000 Shreiner, D., Angel, E., Shreiner, V.

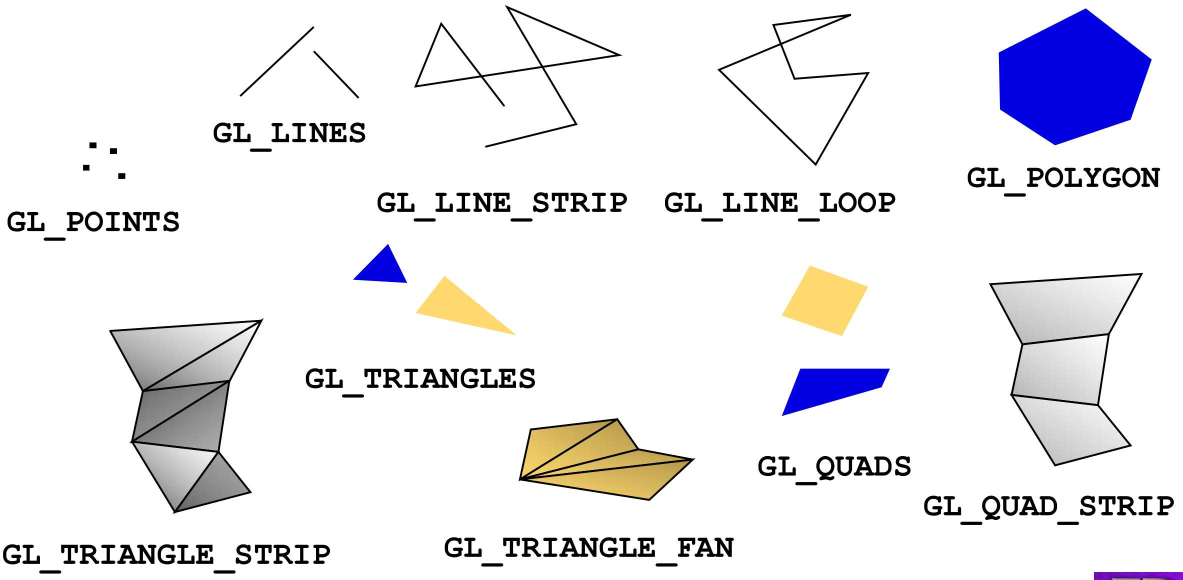
Elementary Rendering

- Geometric Primitives
- Managing OpenGL State
- OpenGL Buffers

© 2000 Shreiner, D., Angel, E., Shreiner, V.

OpenGL Geometric Primitives

- All geometric primitives are specified by vertices



© 2000 Shreiner, D., Angel, E., Shreiner, V.

Simple Example

```
void drawRhombus( GLfloat color[] )
{
    glBegin( GL_QUADS );
    glColor3fv( color );
    glVertex2f( 0.0, 0.0 );
    glVertex2f( 1.0, 0.0 );
    glVertex2f( 1.5, 1.118 );
    glVertex2f( 0.5, 1.118 );
    glEnd();
}
```

© 2000 Shreiner, D., Angel, E., Shreiner, V.

OpenGL Command Formats

```
glVertex3fv( v )
```

Number of components

```
2 - (x,y)
3 - (x,y,z)
4 - (x,y,z,w)
```

Data Type

```
b  - byte
ub - unsigned byte
s  - short
us - unsigned short
i  - int
ui - unsigned int
f  - float
d  - double
```

Vector

omit "v" for scalar form

```
glVertex2f( x, y )
```

© 2000 Shreiner, D., Angel, E., Shreiner, V.

Specifying Geometric Primitives

- **Primitives are specified using**

```
glBegin( primType );
glEnd();
```

  - ***primType*** **determines how vertices are combined**

```
GLfloat red, green, blue;
Glfloat coords[3];
glBegin( primType );
for ( i = 0; i < nVerts; ++i ) {
  glColor3f( red, green, blue );
  glVertex3fv( coords );
}
glEnd();
```

© 2000 Shreiner, D., Angel, E., Shreiner, V.

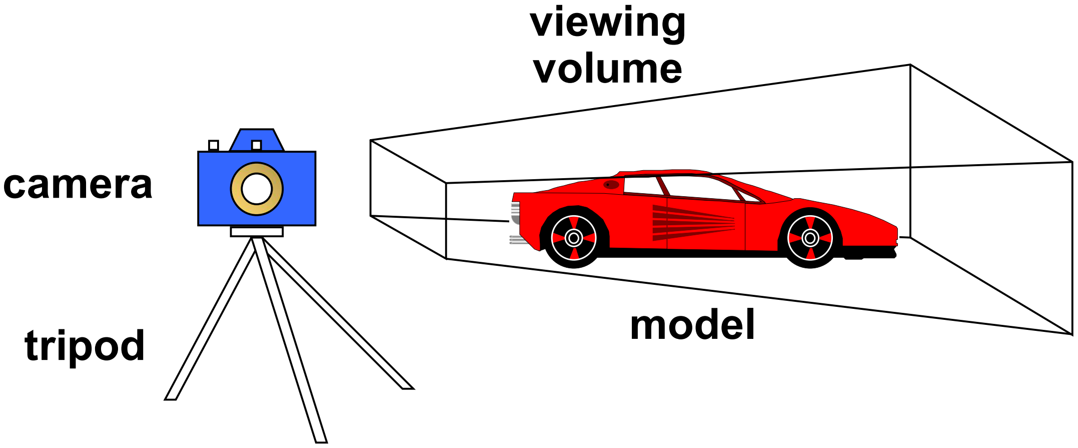
Transformations in OpenGL

- Modeling
- Viewing
  - orient camera
  - projection
- Animation
- Map to screen

© 2000 Shreiner, D., Angel, E., Shreiner, V.

Camera Analogy

- 3D is just like taking a photograph (lots of photographs!)



© 2000 Shreiner, D., Angel, E., Shreiner, V.

Camera Analogy & Transformations

- **Projection transformations**
  - adjust the lens of the camera
- **Viewing transformations**
  - tripod–define position and orientation of the viewing volume in the world
- **Modeling transformations**
  - moving the model
- **Viewport transformations**
  - enlarge or reduce the physical photograph

© 2000 Shreiner, D., Angel, E., Shreiner, V.

Coordinate Systems & Transformation Steps

- Steps in Forming an Image
  - specify geometry (world coordinates)
  - specify camera (camera coordinates)
  - project (window coordinates)
  - map to viewport (screen coordinates)
- Each step uses transformations
- Every transformation is equivalent to a change in coordinate systems (frames)

© 2000 Shreiner, D., Angel, E., Shreiner, V.

Preview: Fixed Function Pipeline, Spaces & Matrices

(See Eberly 2e § 2.3.2 – 2.3.7, pp. 48-66, especially p. 58)

1. model coordinates / object coordinates — $X_{model} \rightarrow (H_{world})$
2. world coordinates / scene coordinates — $X_{world} \rightarrow (H_{view})$
3. camera coordinates / eye coordinates — $X_{view} \rightarrow (H_{proj})$
4. (optional) view coordinates / clip coordinates — $X_{clip} \rightarrow$ (perspective division)
5. normalized device coordinates (NDC) — $X_{ndc} \rightarrow (H_{window})$
6. screen coordinates — X_{window}

H_{world}: modelview transformation

Normalizing transformation: $X_{world} \rightarrow X_{ndc}$ {
- H_{view}: "view matrix" **(really NT!)**
- H_{proj}: projection matrix
- /w: perspective division

H_{window}: window matrix
(*aka* viewport transformation)

Matrix Operations

- Specify Current Matrix Stack

  `glMatrixMode( GL_MODELVIEW or GL_PROJECTION )`
- Other Matrix or Stack Operations

  `glLoadIdentity()` `glPushMatrix()`

  `glPopMatrix()`
- Viewport
  - usually same as window size
  - viewport aspect ratio should be same as projection transformation or resulting image may be distorted

  `glViewport( x, y, width, height )`

© 2000 Shreiner, D., Angel, E., Shreiner, V.

Projection Transformation

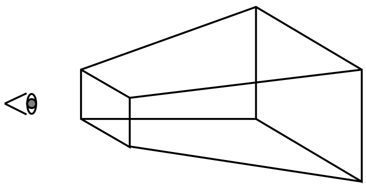
- Shape of viewing frustum
- Perspective projection

```
gluPerspective( fovy, aspect, zNear, zFar )
glFrustum( left, right, bottom, top, zNear, zFar )
```

- Orthographic parallel projection

```
glOrtho( left, right, bottom, top, zNear, zFar )
gluOrtho2D( left, right, bottom, top )
```

  - calls `glOrtho` with z values near zero

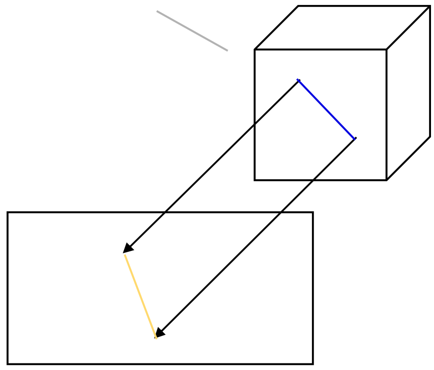


© 2000 Shreiner, D., Angel, E., Shreiner, V.

Applying Projection Transformations

- Typical use (orthographic projection)

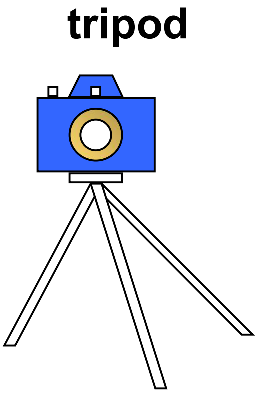
```
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
glOrtho( left, right, bottom, top, zNear, zFar );
```



© 2000 Shreiner, D., Angel, E., Shreiner, V.

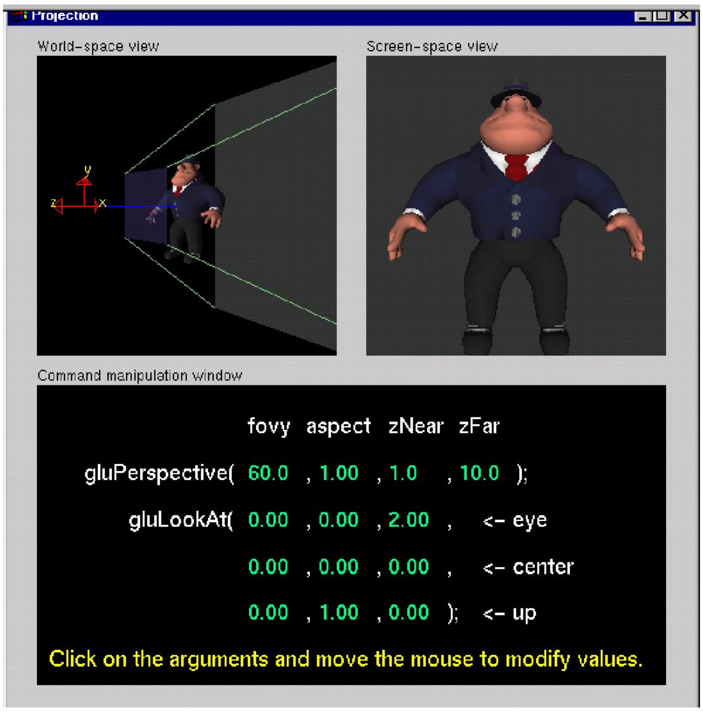
Viewing Transformations

- Position the camera/eye in the scene
  - place the tripod down; aim camera
- To "fly through" a scene
  - change viewing transformation and redraw scene
-
```
gluLookAt( eyex, eyey, eyez,
           aimx, aimy, aimz,
           upx, upy, upz )
```
  - up vector determines unique orientation
  - careful of degenerate positions



© 2000 Shreiner, D., Angel, E., Shreiner, V.

Projection Tutorial



© 2000 Shreiner, D., Angel, E., Shreiner, V.

Modeling Transformations

- **Move object**

  `glTranslate{fd}( x, y, z )`

- **Rotate object around arbitrary axis**

  `glRotate{fd}( angle, x, y, z )`

  - angle is in degrees

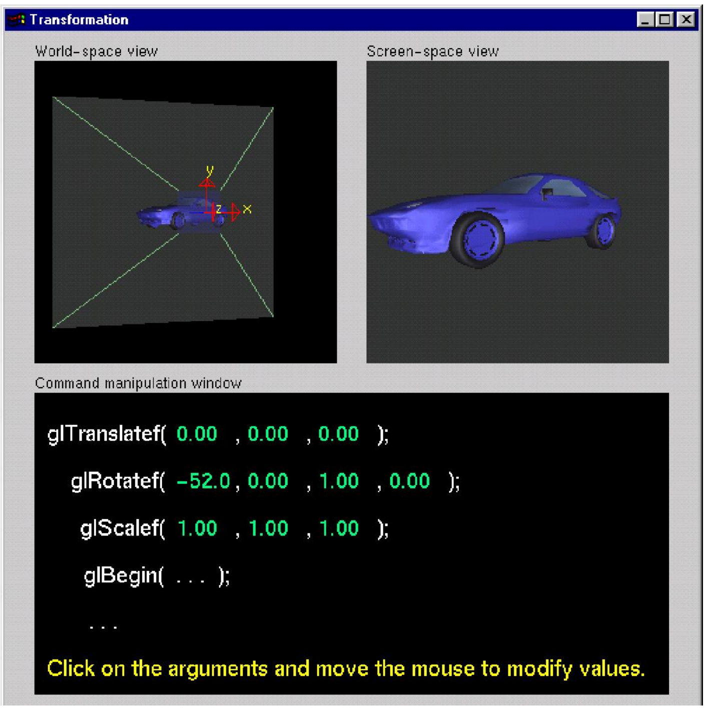
- **Dilate (stretch or shrink) or mirror object**

  `glScale{fd}( x, y, z )`

$$\begin{pmatrix} x & y & z \end{pmatrix}$$

© 2000 Shreiner, D., Angel, E., Shreiner, V.

Transformation Tutorial



© 2000 Shreiner, D., Angel, E., Shreiner, V.

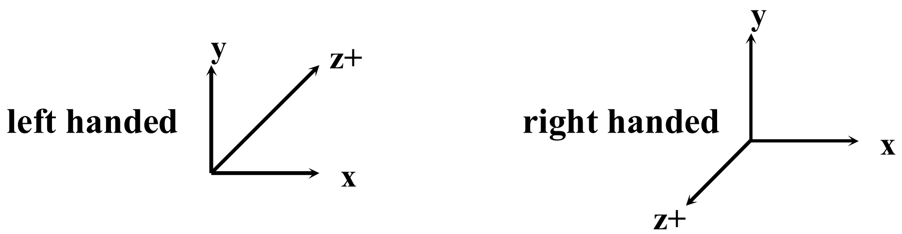
Connection: Viewing and Modeling

- **Moving camera is equivalent to moving every object in the world towards a stationary camera**
- **Viewing transformations are equivalent to several modeling transformations**
  - `gluLookAt()` **has its own command**
  - **can make your own *polar view* or *pilot view***

© 2000 Shreiner, D., Angel, E., Shreiner, V.

Projection is Left-Handed

- Projection transformations (`gluPerspective, glOrtho`) are left handed
  - think of ***zNear*** and ***zFar*** as distance from view point
- Everything else is right handed, including the vertices to be rendered



© 2000 Shreiner, D., Angel, E., Shreiner, V.

resize(): Perspective & LookAt

```
void resize( int w, int h )
{
   glViewport( 0, 0, (GLsizei) w, (GLsizei) h );
   glMatrixMode( GL_PROJECTION );
   glLoadIdentity();
   gluPerspective( 65.0, (GLfloat) w / h,
                   1.0, 100.0 );
   glMatrixMode( GL_MODELVIEW );
   glLoadIdentity();
   gluLookAt( 0.0, 0.0, 5.0,
              0.0, 0.0, 0.0,
              0.0, 1.0, 0.0 );
}
```

© 2000 Shreiner, D., Angel, E., Shreiner, V.

`resize()`: *Ortho (part 1)*

```
void resize( int width, int height )
{
   GLdouble aspect = (GLdouble) width / height;
   GLdouble left = -2.5, right = 2.5;
   GLdouble bottom = -2.5, top = 2.5;
   glViewport( 0, 0, (GLsizei) w, (GLsizei) h );
   glMatrixMode( GL_PROJECTION );
   glLoadIdentity();
```

… continued …

© 2000 Shreiner, D., Angel, E., Shreiner, V.

resize(): Ortho (part 2)

```
   if ( aspect < 1.0 ) {
      left /= aspect;
      right /= aspect;
   } else {
      bottom *= aspect;
      top *= aspect;
   }
   glOrtho( left, right, bottom, top, near, far );
   glMatrixMode( GL_MODELVIEW );
   glLoadIdentity();
}
```

© 2000 Shreiner, D., Angel, E., Shreiner, V.

Summary

- Three Tutorials from SIGGRAPH 2000
- Today (Part 1): OpenGL and GL Utility Toolkit (GLUT) – Vicki Shreiner
  - Overall architecture
  - Initialization
  - Viewport management
- Part 2: Basic Rendering – Vicki Shreiner
- Part 3: 3-D Viewing – Edward Angel
  - Math background (see CG Basics 1)
  - Viewing and normalization transformations (see CG Basics 4)
  - More on viewing in CG Basics 4
  - View volume specification
  - Automated part: clipping

Terminology

- OpenGL and GL Utility Toolkit (GLUT)
  - State machine
  - Using GLUT
  - Specifying perspective, parallel projections
- Transformations
  - Fixed function pipeline: modelview, normalizing, viewing
  - Rigid body: preserves distance (*e.g.*, translation, rotation)
  - Linear
    - Preserves vector addition, scalar multiplication
    - *e.g.*, rotation, scaling
  - Affine: linear transformation followed by translation
  - Non-affine: all others (*e.g.*, perspective-to-parallel transformation)