Lecture Outline

- Reading for Last Class: §2.3 (esp. 2.3.4), Eberly 2°; Foley et al. Slides
- Reading for Today: Chapters 2, 16, Eberly 2°; Foley et al. Slides
- Reading for Next Class: §2.3 (esp. 2.3.7), 2.6, 2.7, Eberly 2°
- Last Time: Matrix Stack for 3-D Viewing Transformation
  - \( N = D_{\text{Persp}} S_{\text{Far}} S_{\text{xy}} M_{\text{Rot}} T_{\text{Trans}} \)
  - 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

<table>
<thead>
<tr>
<th>Lecture</th>
<th>Topic</th>
<th>Primary Source(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Course Overview</td>
<td>Chapter 1, Eberly</td>
</tr>
<tr>
<td>1</td>
<td>CG Basics: Transformation Matrices; Lab 0</td>
<td>Sections (1) 2.1, 2.2, 2.8</td>
</tr>
<tr>
<td>2</td>
<td>Viewing 1: Overview, Projections</td>
<td>Section 1 esp. 2.2.4, 2.2.5</td>
</tr>
<tr>
<td>3</td>
<td>Viewing 2: Viewing Transformation</td>
<td>Section 1 esp. 2.2.6, 2.2.7</td>
</tr>
<tr>
<td>4</td>
<td>Lab 1a: Plain &amp; OpenGL Basics</td>
<td>Ch. 2, 18, Angel, Primer</td>
</tr>
<tr>
<td>5</td>
<td>Viewing 3: Graphics Pipeline</td>
<td>Section 2 esp. 2.6.1, 2.6.2</td>
</tr>
<tr>
<td>6</td>
<td>Scan Conversion 1: Lines, Midpoint Algorithm</td>
<td>Section 2 esp. 2.6.3, 2.6.4</td>
</tr>
<tr>
<td>7</td>
<td>Viewing 4: Clipping &amp; Culling; Lab 1b</td>
<td>Section 2 esp. 2.6.5, 2.6.6</td>
</tr>
<tr>
<td>8</td>
<td>Scan Conversion 2: Polygons, Clipping Intro</td>
<td>Section 2 esp. 2.6.7, 2.6.8</td>
</tr>
<tr>
<td>9</td>
<td>Surface Detail 1: Illumination &amp; Shading</td>
<td>Section 2 esp. 2.6.9, 2.6.10</td>
</tr>
<tr>
<td>10</td>
<td>Lab 2a: DirectX/DirectX Intro</td>
<td>Section 2 esp. 2.6.11, DirectX handout</td>
</tr>
<tr>
<td>11</td>
<td>Surface Detail 2: Textures, OpenGL Shading</td>
<td>Section 2 esp. 2.6.12, DirectX handout</td>
</tr>
<tr>
<td>12</td>
<td>Surface Detail 3: Mappings, OpenGL Textures</td>
<td>Section 2 esp. 2.6.13, DirectX handout</td>
</tr>
<tr>
<td>13</td>
<td>Surface Detail 4: Pixel/Vertex Shad.: Lab 2b</td>
<td>Section 2 esp. 2.6.14, DirectX handout</td>
</tr>
<tr>
<td>14</td>
<td>Surface Detail 5: DirectX Shading, CGLSL</td>
<td>Section 2 esp. 2.6.15, DirectX handout</td>
</tr>
<tr>
<td>15</td>
<td>Demos 1: CGA, Fun; Scene Graphs: State</td>
<td>Section 2 esp. 2.6.16, CGA handout</td>
</tr>
<tr>
<td>16</td>
<td>Lab 3a: Shading &amp; Transparency</td>
<td>Section 2 esp. 2.6.17, Primer</td>
</tr>
<tr>
<td>17</td>
<td>Animation 1: Basics, Keyframes; HW/Exam</td>
<td>Section 2 esp. 2.6.18, Primer</td>
</tr>
<tr>
<td>18</td>
<td>Scene Graphs: Rendering; Lab 3b: Shader</td>
<td>Section 2 esp. 2.6.19, Primer</td>
</tr>
<tr>
<td>19</td>
<td>Demos 2: SFX; Skinning, Morphing</td>
<td>Section 2 esp. 2.6.20, CGA handout</td>
</tr>
<tr>
<td>20</td>
<td>Demos 3: Surfaces, B-eps/Volume Graphics</td>
<td>Section 2 esp. 2.6.21, CGA handout</td>
</tr>
</tbody>
</table>

Lightly-shaded cells 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)
- **Coordinate Systems**
  - World coordinates – standard right-handled, xyz
  - Camera coordinates – camera-space right handed coordinate system
  - (u, v, w); origin at Position and axes related by orientation; used for transforming arbitrary view into canonical view

Adapted from slides © 1997 – 2010 van Dam et al., Brown University

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. \( N = D_{\text{persp}} S_{\text{far}} S_{\text{xy}} M_{\text{rot}} T_{\text{trans}} \)
  2. \( \text{CTM} = N \cdot \text{CMTM} \)
  3. \( P' = \text{CTM} \cdot P \) for every vertex \( P \) defined in its own coordinate system
  4. \( P_{\text{screen}} = 512 \cdot P' + 1 \) for all clipped \( P' \)

Adapted from slides © 1997 – 2010 van Dam et al., Brown University

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 X-Server Online GPLS drop box before the due date and time.

References:
NeonHec tutorial: http://neohhe.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/technotes/viewing.htm

Problems 4-7 of this lab are adapted from:


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 slab1_2-0 and slab1_2-0.ppm.

2. (20%) Modelview transformation: 3-D Rotation of 3-D objects. Follow Lesson 04 to draw 3-D polyhedra and rotate them. Turn in slab1_2-0 and slab1_2-3.ppm.

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 slab1_2-3.ppm.
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

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

OpenGL Architecture
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.

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.

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.`

---

**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
Sample Program

```c
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

```c
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);
  ```

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();
  }
  ```
Elementary Rendering

- Geometric Primitives
- Managing OpenGL State
- OpenGL Buffers

OpenGL Geometric Primitives

- All geometric primitives are specified by vertices

- GL_LINES
- GL_POINT
- GL_LINE_STRIP
- GL_LINE_LOOP
- GL_TRIANGLES
- GL_QUADS
- GL_QUAD_STRIP
- GL_TRIANGLES
- GL_TRIANGLE_FAN
Simple Example

```c
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();
}
```

OpenGL Command Formats

```c
glVertex3fv( v )
```

<table>
<thead>
<tr>
<th>Number of components</th>
<th>Data Type</th>
<th>Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 - (x, y)</td>
<td>b - byte</td>
<td>glVertex2f( x, y )</td>
</tr>
<tr>
<td>3 - (x, y, z)</td>
<td>ub - unsigned byte</td>
<td></td>
</tr>
<tr>
<td>4 - (x, y, z, w)</td>
<td>s - short</td>
<td></td>
</tr>
<tr>
<td></td>
<td>us - unsigned short</td>
<td></td>
</tr>
<tr>
<td></td>
<td>i - int</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ui - unsigned int</td>
<td></td>
</tr>
<tr>
<td></td>
<td>f - float</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d - double</td>
<td></td>
</tr>
</tbody>
</table>
```

© 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();
  ```

Transformations in OpenGL

- Modeling
- Viewing
  - orient camera
  - projection
- Animation
- Map to screen
3D is just like taking a photograph (lots of photographs!)

- 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
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)

Coordinate Systems & Transformation Steps

Preview: Fixed Function Pipeline, Spaces & Matrices

(See Eberly 2e § 2.3.2 – 2.3.7, pp. 48-86, especially p. 58)

1. model coordinates / object coordinates
2. world coordinates / scene coordinates
3. camera coordinates / eye coordinates
4. (optional) view coordinates / clip coordinates
5. normalized device coordinates (NDC)
6. screen coordinates

\[ H_{\text{world}}: \text{modelview transformation} \]

Normalizing transformation: \( X_{\text{world}} \rightarrow X_{\text{ndc}} \)

\[ H_{\text{view}}: \] “view matrix” (really NT!)
\[ H_{\text{proj}}: \] projection matrix
\[ H_{\text{w}}: \] perspective division

\[ H_{\text{window}}: \text{window matrix} \]
(aka viewport transformation)
Matrix Operations

- Specify Current Matrix Stack
  ```
  glMatrixMode( GL_MODELVIEW or GL_PROJECTION )
  ```
- Other Matrix or Stack Operations
  ```
  glLoadIdentity()     glPushMatrix()
  ```
- 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 )
  ```

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
Applying Projection Transformations

- Typical use (orthographic projection)
  ```
  glMatrixMode( GL_PROJECTION );
  glLoadIdentity();
  glOrtho( left, right, bottom, top, zNear, zFar );
  ```

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( eye_x, eye_y, eye_z, 
             aim_x, aim_y, aim_z, 
             up_x, up_y, up_z )
  ```
  * up vector determines unique orientation
  * careful of degenerate positions
Projection Tutorial

- Move object
  \[ \text{glTranslate(fd)} (x, y, z) \]
- Rotate object around arbitrary axis
  \[ \text{glRotate(fd)} (\text{angle}, x, y, z) \]
  \( \text{angle is in degrees} \)
- Dilate (stretch or shrink) or mirror object
  \[ \text{glScale(fd)} (x, y, z) \]

Modeling Transformations

- Move object
  \[ \text{glTranslate(fd)} (x, y, z) \]
- Rotate object around arbitrary axis
  \[ \text{glRotate(fd)} (\text{angle}, x, y, z) \]
  \( \text{angle is in degrees} \)
- Dilate (stretch or shrink) or mirror object
  \[ \text{glScale(fd)} (x, y, z) \]
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
- You can make your own polar view or pilot view
**Projection is Left-Handed**

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

![Left-Handed Right-Handed Diagram](image)

```c
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 );
}
```
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();
    glOrtho( left, right, bottom, top, near, far );
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    ... continued ...

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

- Three Tutorials from SIGGRAPH 2000
    - 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)