3D Programming and Viewing
Week 3

David Breen
Department of Computer Science
Drexel University

Based on material from Ed Angel, University of New Mexico
3D Objects
3D Quadrilateral

```c
glBegin(GL_POLYGON)
glVertex3f(-1.0, -1.0, -1.0);
glVertex3f(-1.0, 1.0, -1.0);
glVertex3f(-1.0, 1.0, 1.0);
glVertex3f(-1.0, -1.0, 1.0);
glEnd();
```

Vertices should be coplanar!

Angel: Interactive Computer Graphics 3E
© Addison-Wesley 2002
Modeling a Cube

Model a color cube for rotating cube program

Define global arrays for vertices and colors

```c
GLfloat vertices[][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
{1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},
{1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};

GLfloat colors[][3] = {{0.0,0.0,0.0},{1.0,0.0,0.0},
{1.0,1.0,0.0}, {0.0,1.0,0.0}, {0.0,0.0,1.0},
{1.0,0.0,1.0}, {1.0,1.0,1.0}, {0.0,1.0,1.0}};
```
Drawing a polygon from a list of indices

Draw a quadrilateral from a list of indices into the array *vertices* and use color corresponding to first index

```c
void polygon(int a, int b, int c, int d)
{
    glBegin(GL_POLYGON);
    glColor3fv(colors[a]);
    glVertex3fv(vertices[a]);
    glVertex3fv(vertices[b]);
    glVertex3fv(vertices[c]);
    glVertex3fv(vertices[d]);
    glEnd();
}
```
void colorcube( )
{
    polygon(0,3,2,1);
    polygon(2,3,7,6);
    polygon(0,4,7,3);
    polygon(1,2,6,5);
    polygon(4,5,6,7);
    polygon(0,1,5,4);
}

Note that vertices are ordered so that we obtain correct outward facing normals
Efficiency

• The weakness of our approach is that we are building the model in the application and must do many function calls to draw the cube.

• Drawing a cube by its faces in the most straightforward way requires:
  – 6 glBegin, 6 glEnd
  – 6 glColor
  – 24 glVertex
  – More if we use texture and lighting
Vertex Arrays

- OpenGL provides a facility called *vertex arrays* that allow us to store array data in the implementation.

- Six types of arrays supported:
  - Vertices
  - Colors
  - Color indices
  - Normals
  - Texture coordinates
  - Edge flags

- We will need only colors and vertices.
Initialization

• Using the same color and vertex data, first we enable
  
  glEnableClientState(GL_COLOR_ARRAY);
  glEnableClientState(GL_VERTEX_ARRAY);

• Identify location of arrays
  
  glVertexPointer(3, GL_FLOAT, 0, vertices);
  glColorPointer(3, GL_FLOAT, 0, colors);
  glShadeModel(GL_FLAT); /* 1 color/face */
Mapping indices to faces

• Form an array of face indices

\[
\text{GLubyte cubeIndices}[24] = \{0, 3, 2, 1, 2, 3, 7, 6, 0, 4, 7, 3, 1, 2, 6, 5, 4, 5, 6, 7, 0, 1, 5, 4\};
\]

• Each successive four indices describe a face of the cube

• Draw through \texttt{glDrawElements} which replaces all \texttt{glVertex} and \texttt{glColor} calls in the display callback
Vertex Arrays

Glfloat vertices [][][3] = 
{\{-1.0,-1.0,1.0\}, \{-1.0,1.0,1.0\},
{1.0,1.0,1.0\}, \{1.0,-1.0,1.0\},
\{-1.0,-1.0,-1.0\}, \{-1.0,1.0,-1.0\},
{1.0,1.0,-1.0\}, \{1.0,-1.0,-1.0\}\};

Glfloat colors [][][3] =
{\{1.0,0.0,0.0\}, \{0.0,1.0,1.0\},
\{1.0,1.0,0.0\}, \{0.0,1.0,0.0\},
\{0.0,0.0,1.0\}, \{1.0,0.0,1.0\},
\{1.0,1.0,1.0\}, \{0.0,0.0,0.0\}\};

Glubyte cubeIndices [] =
\{0, 3, 2, 1, 2, 3, 7, 6, 0, 4, 7, 3, 1, 2, 6, 5, 4, 5, 6, 7, 0, 1, 5, 4\};
Drawing the cube

• Method 1:

```c
for(i=0; i<6; i++) glDrawElements(GL_POLYGON, 4,
    GL_UNSIGNED_BYTE, &cubeIndices[4*i]);
```

• Method 2:

```c
glDrawElements(GL_QUADS, 24,
    GL_UNSIGNED_BYTE, cubeIndices);
```

Draws cube with 1 function call!!
Drawing a Colored Cube

```c
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);

glColorPointer(3, GL_FLOAT, 0, colors);
glVertexPointer(3, GL_FLOAT, 0, vertices);

glShadeModel(GL_FLAT) /* 1 color/face */

for (i = 0; i < 6; i++)
    glDrawElements(GL_POLYGON, 4,
                   GL_UNSIGNED_BYTE, cubeIndices+(4*i));

    glDrawElements(GL_QUADS, 24,
                   GL_UNSIGNED_BYTE, cubeIndices);
```
Display Lists
Immediate and Retained Modes

• Recall that in a standard OpenGL program, once an object is rendered there is no memory of it and to redisplay it, we must re-execute the code for it
  – Known as *immediate mode graphics*
  – Can be especially slow if the objects are complex and must be sent over a network

• Alternative is define objects and keep them in some form that can be redisplayed easily
  – *Retained mode graphics*
  – Accomplished in OpenGL via *display lists*
Display Lists

• Conceptually similar to a graphics file
  – Must define (name, create)
  – Add contents
  – Close

• In client-server environment, display list is placed on server
  – Can be redisplayed without sending primitives over network each time
Display List Functions

• Creating a display list

```c
GLuint id;

void init( void )
{
    id = glGenLists( 1 );
    glNewList( id, GL_COMPILE );
    /* or use GL_COMPILE_AND_EXECUTE */
    /* other OpenGL routines */
    glEndList();
}
```

• Call a created list

```c
void display( void )
{
    glCallList( id );
}
```
Display Lists and State

• Most OpenGL functions can be put in display lists
• State changes made inside a display list persist after the display list is executed
• Can avoid unexpected results by using `glPushAttrib` and `glPushMatrix` upon entering a display list and `glPopAttrib` and `glPopMatrix` before exiting
Hierarchy and Display Lists

- Consider model of a car
  - Create display list for chassis
  - Create display list for wheel

```cpp
glNewList( CAR, GL_COMPILE );
glCallList( CHASSIS );
glTranslatef( ... );
glCallList( WHEEL );
glTranslatef( ... );
glCallList( WHEEL );
...
glEndList();
```
Vertex Buffer Objects

- An extension added in OGL 1.5
- Regions of memory (buffers) accessible through identifiers
- Made active through binding, like display lists and textures
- Provides control over the mappings and unmappings of buffer objects
- Defines the usage type of the buffers
  - Allows graphics drivers to optimize internal memory management of the buffers
- Possible to share VBO data among various clients
- Clients are able to bind common buffers in the same way as textures or display lists.
GLU Quadrics

```c
GLUquadricObj  *mySphere;
mySphere = gluNewQuadric();
gluQuadricDrawStyle(mySphere, GLU_FILL);
gluQuadricNormals(mySphere, GLU_FLAT);
gluQuadricTexture(mySphere, GL_TRUE);
gluSphere(mySphere, 1.0, 12, 12);
```

...  

```c
gluDeleteQuadric(mySphere);
```

```c
gluCylinder
gluDisk
gluPartialDisk
```
GLU Quadrics

GLUquadricObj *myCyl;
myCyl = gluNewQuadric();
gluQuadricDrawStyle(myCyl, GLU_FILL);
gluQuadricNormals(myCyl, GLU_SMOOTH);
gluQuadricTexture(myCyl, GL_FALSE);
gluCylinder(myCyl, 3.0, 2.0, 2.0, 2, 20);
...

... base radius top radius height slices stacks

 gluDeleteQuadric(myCyl);

 gluQuadricDrawStyle(myDisk, GLU_LINE);
gluDisk(1.0, 5.0, 6, 16);
gluQuadricDrawStyle(myPartDisk, GLU_POINT);
gluPartialDisk (1.0, 5.0, 6, 16, 25.0, 210.0);
GLU Quadrics

**gluSphere**

**gluCylinder**

**gluDisk**

**gluPartialDisk**

http://resumbrae.com/ub/dms423_f04/08
GLUT Objects

glutWireCube, glutSolidCube
  centered around origin with length size

glutWireSphere, glutSolidSphere
  centered around origin with radius length

glutWireCone, glutSolidCone
  base on z=0  specify base radius and height

glutWireTorus, glutSolidTorus
  aligned with z axis
  specify inner and outer radii

Specify # of sides, rings, slices and stacks
GLUT Objects

glutWireTetrahedron, glutSolidTetrahedron

glutWireOctahedron, glutSolidOctahedron

glutWireDodecahedron, glutSolidDodecahedron

glutWireIcosahedron, glutSolidIcosahedron

Generate Platonic solids with all vertices on unit sphere

glutWireTeapot, glutSolidTeapot
  size may be adjusted
GLUT Objects

http://resumbrae.com/ub/dms423_f04/08
Viewing Overview
Classical Viewing

- Viewing requires three basic elements
  - One or more objects
  - A viewer with a projection surface
  - Projectors that go from the object(s) to the projection surface
- Classical views are based on the relationship among these elements
  - The viewer picks up the object and orients it how she would like to see it
  - Or the object is stationary and the viewer moves around it
- Each object is assumed to be constructed from flat \textit{principal faces}
  - Buildings, polyhedra, manufactured objects
Planar Geometric Projections

- Standard projections project onto a plane
- Projectors are lines that either
  - converge at a center of projection
  - are parallel
- Such projections preserve lines
  - but not necessarily angles
- Nonplanar projections are needed for applications such as map construction
Perspective vs Parallel

- Computer graphics treats all projections the same and implements them with a single pipeline
- Classical viewing developed different techniques for drawing each type of projection
- Fundamental distinction is between parallel and perspective viewing even though mathematically parallel viewing is the limit of perspective viewing
Taxonomy of Planar Geometric Projections

planar geometric projections

parallel

multiview
orthographic

axonometric
oblique

isometric
dimetric
trimetric

perspective

1 point
2 point
3 point
Perspective Projection

- Object
- Projector
- Projection plane

COP
Parallel Projection
Orthographic Projection

Projectors are orthogonal to projection surface
Advantages and Disadvantages

• Preserves both distances and angles
  – Shapes preserved
  – Can be used for measurements
    • Building plans
    • Manuals

• Cannot see what object really looks like because many surfaces hidden from view
  – Often we add the isometric
Oblique Projection

Arbitrary relationship between projectors and projection plane
Perspective Projection

Projectors converge at center of projection
Vanishing Points

- Parallel lines (not parallel to the projection plan) on the object converge at a single point in the projection (the *vanishing point*).
- Drawing simple perspectives by hand uses these vanishing point(s).
Advantages and Disadvantages

- Objects further from viewer are projected smaller than the same sized objects closer to the viewer (*diminution*)
  - Looks realistic
- Equal distances along a line are not projected into equal distances (*nonuniform foreshortening*)
- Angles preserved only in planes parallel to the projection plane
- More difficult to construct by hand than parallel projections (but not more difficult by computer)
Computer Viewing
Objectives

• Introduce OpenGL viewing functions
• Look at alternate viewing APIs
Computer Viewing

• There are three aspects of the viewing process, all of which are implemented in the pipeline,
  – Positioning the camera
    • Setting the model-view matrix
  – Selecting a lens
    • Setting the projection matrix
  – Clipping
    • Setting the view volume
The OpenGL Camera

• In OpenGL, initially the world and camera frames are the same
  – Default model-view matrix is an identity
• The camera is located at origin and points in the negative z direction
• OpenGL also specifies a default view volume that is a cube with sides of length 2 centered at the origin
  – Default projection matrix is an identity
Default Projection

Default projection is orthographic

clipped out

Projection plane $z=0$
Moving the Camera Frame

• If we want to visualize an object with both positive and negative z values we can either
  – Move the camera in the positive z direction
    • Translate the camera frame
  – Move the objects in the negative z direction
    • Translate the world frame

• Both of these views are equivalent and are determined by the model-view matrix
  – Want a translation \( \text{glTranslatef}(0.0,0.0,-d); \)
  – \( d > 0 \)
Moving Camera back from Origin

frames after translation by \(-d\)

\(d > 0\)

default frames
Moving the Camera

- We can move the camera to any desired position by a sequence of rotations and translations.
- Example: side view
  - Move it away from origin
  - Rotate the camera
  - Model-view matrix $C = TR$
The LookAt Function

• The GLU library contains the function gluLookAt to form the required modelview matrix through a simple interface
• Note the need for setting an up direction
• Still need to initialize
  – Can concatenate with modeling transformations
• Example: isometric view of cube aligned with axes

```c
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0., 1.0. 0.0);
```
The LookAt Function

```c
gluLookAt(eyex, eyey, eyez, atx, aty, atz, 
upx, upy, upz)
```

Diagram showing the parameters `eyex`, `eyey`, `eyez`, `atx`, `aty`, `atz`, `upx`, `upy`, and `upz` with corresponding 3D coordinates and vectors.
LookAt Example

```c
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(1.0,1.0,1.0, 0.0,0.0,0.0, 0.0,1.0,0.0);
    glutWireCube(0.5);
    glutSwapBuffers();
}
```
Other Viewing APIs

• The LookAt function is only one possible API for positioning the camera
• Others include
  – View reference point, view plane normal, view up (PHIGS, GKS-3D)
  – Yaw, pitch, roll
  – Elevation, azimuth, twist
  – Direction angles
Projections and Normalization

- The default projection in the eye (camera) frame is orthographic.
- For points within the default view volume:
  \[ x_p = x \]
  \[ y_p = y \]
  \[ z_p = 0 \]
- Most graphics systems use view normalization:
  - All other views are converted to the default view by transformations that determine the projection matrix.
  - Allows use of the same pipeline for all views.
OpenGL Orthographic Viewing

\texttt{glOrtho(xmin, xmax, ymin, ymax, near, far)}

all values measured from camera
\texttt{near} and \texttt{far} can be positive and negative
OpenGL Perspective

glFrustum(xmin, xmax, ymin, ymax, near, far)

all values measured from camera
near and far should be positive
Using Field of View

- With `glFrustum` it is often difficult to get the desired view
- `gluPerspective(fovy, aspect, near, far)` often provides a better interface

![Diagram showing the field of view with labels for `w`, `h`, `x`, `y`, `z`, `fov`, `front plane`, `aspect = w/h`, and `fovy given in degrees`.]

© Addison-Wesley 2002
OpenGL Perspective

- `glFrustum` allows for an unsymmetric viewing frustum
- `gluPerspective` does not
Another LookAt Example

void reshape(int w, int h)
{
    glViewport(0, 0, (Glsizei) w, (Glsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (GLfloat)w/(GLfloat)h, 1.0, 20.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(1.0,1.0,1.0, 0.0,0.0,0.0, 0.0,1.0,0.0); 
}
Hidden-Surface Removal

/* back-face culling for convex objects */

glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);

/* More general Z-buffer algorithm */

glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);

glEnable(GL_DEPTH_TEST);

/* Must clear depth buffer too! */
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);