CMSC 427: Chapter 4
3D Viewing

Reading: Chapt 7 in Shirley.
Overview:
- Camera Frame and Eye Coordinates
  - World-to-Camera Transformation and gluLookAt( )
- Projection
  - Parallel projection and glOrtho( )
  - Projective Geometry
  - Perspective projection and glFrustum( ) and gluPerspective( )

Overview

- Overview of 3-d viewing process
- World-to-Camera transformation and gluLookAt( )
- Parallel projection and glOrtho( )
- Perspective projection and Projective geometry
- Perspective in OpenGL: glFrustum( ) and gluPerspective( )
- Normalized view volume and Perspective plus depth
Viewing in 3D

Viewing in 3D: This is a complex process involving a number of coordinated elements. The goal is to map a 3D scene to a 2D image according to viewing and projection parameters, given by the user.

OpenGL Support:
OpenGL performs this entire process for us. However, it is important to understand "how" the process works, since it is fundamental to understanding computer graphics.

Additional Issues: We will concentrate here on geometric issues.
Other issues, which will be ignored now, include:
- Hidden-surface removal.
- Lighting, shading, texturing.

Transformations for 3D Viewing

3D viewing involves three distinct transformations:

Modelview: (3D affine transformation)
Transforms 3D world coordinates to 3D view coordinates.
- gluLookAt (…)

Projection: (3D affine or projective transformation)
Projects 3D objects onto the 2D view plane (plus depth info)
- gluOrtho2D (…) or
- glFrustum (…) or
- gluPerspective (…)

Viewport: (2D affine transformation)
Transforms points on the view plane to the graphics viewport.
- glViewport (…)

Chapter 4, Slide 3
Copyright © D. M. Mount and A. Varshney
Transformations for 3D Viewing

Modelview: Map 3D world coordinates to 3D view coordinates.
Projection: Projects 3D objects onto the 2D view plane.
Viewport: Map the view plane to the graphics viewport.

Overview of the Viewing Process:

3D Viewing Pipeline: An Expanded View

Chapter 4, Slide 5
Copyright © D. M. Mount and A. Vaidhney

Chapter 4, Slide 6
Copyright © D. M. Mount and A. Vaidhney
Overview

- Overview of 3-d viewing process
- World-to-Camera transformation and gluLookAt()
- Parallel projection and glOrtho()
- Perspective projection and Projective geometry
- Perspective in OpenGL: glFrustum() and gluPerspective()
- Normalized view volume and Perspective plus depth

Specifying the Camera Position

World frame: The frame in which you represent your points.

View Frame: (also called camera or eye frame) Is specified by the following quantities, relative to world coordinates:

Eye Location: The center of projection, also called the view reference point or VRP.

Viewing Direction: A unit vector that is normal to the view plane, called the view-plane normal, or VPN.

Camera twist: The camera's rotation about the viewing axis, specified by indicating the "up" direction for the camera, called view-plane up, or VUP.
Specifying the View Frame: \texttt{gluLookAt( )}

In OpenGL the view frame is specified by calling:

\begin{verbatim}
\texttt{gluLookAt( eye\_x, eye\_y, eye\_z, at\_x, at\_y, at\_z, up\_x, up\_y, up\_z )}
\end{verbatim}

All arguments are of type \texttt{GLdouble}. Where the following are given in world coordinates:

- \( \text{eye} = (\text{eye\_x, eye\_y, eye\_z}) \) is the \textit{location of the eye}.

- \( \text{at} = (\text{at\_x, at\_y, at\_z}) \) is a point that the viewer is \textit{looking at}. The vector \( \text{at} - \text{eye} \) is the \textit{viewing direction}.

- \( \text{up} = (\text{up\_x, up\_y, up\_z}) \) is a vector indicating which direction is \textit{up relative to the camera}. It is used to encode the camera's twist about the viewing direction. (It need not be orthogonal to the view direction, but it cannot be parallel to the view direction vector.)
Specifying the View Frame: `gluLookAt()`

- `gluLookAt()` constructs a matrix that converts from world coordinates to view coordinates and multiplies it times the top of the `modelview` matrix stack.
- Normally this is the first matrix on the modelview stack.

```c
    glLoadIdentity();
    gluLookAt( ... );  // V
    glPushMatrix();
    glRotatef( ... );  // R
    glTranslatef( ... ); // T
    // ... do some drawing ...
    glPopMatrix();
```

- Each point given in the drawing process will be transformed first by T, then by R, and finally converted into view coordinates by V.

The View Frame

View Frame:
- Origin at the center of projection (eye location).
- $v_x$ points to camera right.
- $v_y$ points to camera up.
- $v_z$ points in the opposite direction of the view direction.

Why? To make the frame right-handed ($v_x \times v_y = v_z$).

**Objective:** Compute a matrix that converts world coordinates to view frame coordinates.
World Coordinates to View Coordinates

Fact 1: Let \( M \) be the 4x4 matrix whose columns are the homogeneous coordinates of \( v_x, v_y, v_z, \) and \( \text{eye} \). Then \( M^{-1} \) is the desired change of coordinates transformation.

Intuition: You can think of an affine transformation \( M \) as mapping points to points (with frame fixed) or \( M^{-1} \) as mapping frames to frames (with points fixed).

Fact 2: The matrix \( M \) can be expressed as the product of two matrices \( T \cdot R \), where \( T \) is a pure translation matrix (\( \text{eye} \)) and \( R \) is a pure rotation (\( v_x, v_y, v_z \)).

Fact 3: Because view frame is orthonormal, \( R^{-1} = R^T \).

Fact 4: Final matrix is \( M^{-1} = (T \cdot R)^{-1} = R^{-1} \cdot T^{-1} = R^T \cdot T^{-1} \).

Step 1: Find the translation \( T \) that maps world origin to view origin. The view origin is \( \text{eye} = (e_x, e_y, e_z) \) so we have:

\[
T = \begin{bmatrix}
1 & 0 & 0 & e_x \\
0 & 1 & 0 & e_y \\
0 & 0 & 1 & e_z \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

The inverse \( T^{-1} \) arises by translating by the negation of this:

\[
T^{-1} = \begin{bmatrix}
1 & 0 & 0 & -e_x \\
0 & 1 & 0 & -e_y \\
0 & 0 & 1 & -e_z \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
World Coordinates to View Coordinates

**Step 2:** Find the rotation \( R \) that maps the world basis vectors to the view frame basis vectors.

2-a: \( v_z \) is the normalized negation of the view direction.
2-b: \( v_x \) is orthogonal to the \( v_z \) and the up vector.
2-c: \( v_y \) is orthogonal to \( v_z \) and \( v_x \).

**Step 2-a:** View direction is at - eye.

Normalize to unit length:

\[
\mathbf{v}_z = \frac{-\mathbf{v}_\text{view}}{||\mathbf{v}_\text{view}||}
\]

**Step 2-b:** \( \mathbf{v}_x = \mathbf{up} \times \mathbf{v}_z \) and normalize to get \( \mathbf{v}_x = \mathbf{v}_x / ||\mathbf{v}_x|| \).

**Step 2-c:** \( \mathbf{v}_y = \mathbf{v}_z \times \mathbf{v}_x \). (Normalization is not needed since \( \mathbf{v}_z \) and \( \mathbf{v}_x \) already normalized and orthogonal.)

**World Coordinates to View Coordinates (cont):** These calculations are all done in world coordinates, so we have the view basis vectors \( v_x, v_y, v_z \) expressed in world coordinates. Thus, the desired rotation matrix \( R \) that maps the world basis vectors to the view basis vectors is:

\[
R = \begin{bmatrix}
    v_{xx} & v_{xy} & v_{xz} & 0 \\
    v_{yx} & v_{yy} & v_{yz} & 0 \\
    v_{zx} & v_{zy} & v_{zz} & 0 \\
    0 & 0 & 0 & 1
\end{bmatrix}
\]

The upper-left 3x3 submatrix is orthogonal, and thus the inverse is:

\[
R^{-1} = R^T = \begin{bmatrix}
    v_{xx} & v_{xy} & v_{xz} & 0 \\
    v_{yx} & v_{yy} & v_{yz} & 0 \\
    v_{zx} & v_{zy} & v_{zz} & 0 \\
    0 & 0 & 0 & 1
\end{bmatrix}
\]

The final view transformation matrix produced by `gluLookAt` is \( R^T \cdot T^{-1} \).
Where does gluLookAt() go?

Typical redraw callback for 3D viewing:

```c
void myDisplay() {
    glClear(GL_COLOR_BUFFER_BIT | ...);  // clear the buffer
    glLoadIdentity();                    // set up view frame
    gluLookAt(...);                     // set up projection
    glMatrixMode(GL_PROJECTION);         // ... we'll discuss projection later
    glMatrixMode(GL_MODELVIEW);
    myWorld.draw();                     // draw everything
    glutSwapBuffers();
}
```

Overview

- Overview of 3-d viewing process
- World-to-Camera transformation and gluLookAt()
- Parallel projection and glOrtho()
- Perspective projection and Projective geometry
- Perspective in OpenGL: glFrustum() and gluPerspective()
- Normalized view volume and Perspective plus depth
Projection

**Next step:** After transforming points into view coordinates, the next step is to **project** them onto the 2-dimensional image plane.

**Types of Projections:**
- **Parallel projection:** Projects everything in the same direction.
  - **Orthogonal projection:** Direction of projection is orthogonal to the image plane. (Most common.)
  - **Oblique projection:** Direction of projection need not be orthogonal.
- **Perspective projection:** Projects everything towards a single point (the eye), called the **center of projection**.

![2D Side View](image.png)

**Properties of Projections**

**Parallel Projection:**
- Lengths and angles are **not preserved**.
- Parallel lines project to parallel lines.
- **Affine transformation** (midpoints map to midpoints).
- No foreshortening: Near objects are the same size as far objects.

**Perspective Projection:**
- Lengths and angles are **not preserved**.
- Parallel lines may be mapped to non-parallel lines.
- **Not** an affine transformation (midpoints not generally preserved).
- **Foreshortening:** Near objects appear larger than far objects.
Orthogonal Projection in OpenGL

**Orthogonal Projection**: is easy using view coordinates:
- Convert points to **view coordinates**.
- Simply throw away the **z-coordinate**.

**Projection + Depth**: We can make use of the z-coordinate, however. For **hidden surface removal**, we need to know which points are closer to the camera.

**Depth Buffer**: In addition to storing the color of each pixel in the frame buffer, we store its **distance** from the eye (-pz) in a separate depth buffer. When scan converting only the closest pixel color is kept.
**Depth Information:** Only a finite number of bits can be stored in the depth buffer. So, the user specifies the minimum and maximum distance values.  

`glOrtho ( left, right, bottom, top, near, far )`

All arguments are of type `GLfloat`.

**View volume:** This defines a 3D rectangular view volume. All objects outside this volume are removed, or clipped.

**Normalized View Volume:** `glOrtho( )` transforms objects within the view volume to a 2x2x2 cube, centered at the origin, called the normalized view volume.

**Clipping:** is done in the normalized view volume.
**glOrtho( ) Transformation Matrix**

The transformation matrix for `glOrtho()` transforms intervals:

- \( x: \text{[left, right]} \rightarrow [-1, +1] \)
- \( y: \text{[bottom, top]} \rightarrow [-1, +1] \)
- \( z: \text{[near, -far]} \rightarrow [-1, +1] \)

The `glOrtho()` matrix is:

\[
\begin{bmatrix}
  s_x & 0 & 0 & -t_x s_x \\
  0 & s_y & 0 & -t_y s_y \\
  0 & 0 & s_z & -t_z s_z \\
  0 & 0 & 0 & 1
\end{bmatrix}
\]

where:

\[
\begin{align*}
  t_x &= (\text{right} + \text{left})/2 \\
  t_y &= (\text{bottom} + \text{top})/2 \\
  t_z &= (\text{near} + \text{far})/2 \\
  s_x &= 2/(\text{right} - \text{left}) \\
  s_y &= 2/(\text{top} - \text{bottom}) \\
  s_z &= -2/(\text{far} - \text{near})
\end{align*}
\]

---

**Where does glOrtho( ) go?**

Typical **redisplay callback** for 3D viewing:

```c
void myDisplay ( ) {
    glClear ( GL_COLOR_BUFFER_BIT | ... ); // clear the buffer
    glLoadIdentity ( );
    gluLookAt ( ... ); // set up view frame
    glMatrixMode ( GL_PROJECTION ); // set up projection
    glLoadIdentity ( );
    glOrtho ( left, right, bottom, top, near, far );
    glMatrixMode ( GL_MODELVIEW );
    myWorld.draw ( ); // draw everything
    glutSwapBuffers ( );
}
```

It is **safest** to put it in your display callback. In fact, it only needs to be redone when the **camera view** is changed.
Overview

- Overview of 3-d viewing process
- World-to-Camera transformation and gluLookAt()
- Parallel projection and glOrtho()
- Perspective projection and Projective geometry
- Perspective in OpenGL: glFrustum() and gluPerspective()
- Normalized view volume and Perspective plus depth

Perspective Projection

**Perspective Projection:** Points are projected towards the center of projection (eye).

To understand perspective projection, we need to understand some projective geometry.
Projective Geometry

Euclidean geometry: Any two distinct lines intersect in exactly one point, unless they are parallel.

Projective geometry: Any two distinct lines intersect in exactly one point. To handle parallel lines, we add points at infinity.

Point at Infinity (Ideal Point): For each direction $u$ there is a point infinitely far away in this direction. All lines parallel to this direction intersect at this point.

Regular Points: Standard Euclidean points are called regular points.

Wrap Around: The point at infinity in direction $u$ is the same as the one in direction $-u$. Thus, projective space wraps around.

Homogeneous Coordinates for Projective Geom

How to represent points at infinity? We will use homogeneous coordinates, but they have a new interpretation.

(Projective) Homogeneous Coordinates in 2D: Given a regular point at standard coordinates $(x, y)$, it is represented by any coordinate vector of the form $[w \cdot x, w \cdot y, w]$, for $w \neq 0$.

A single point has multiple representations, e.g.

$[3, 2, 1] \quad [6, 4, 2] \quad [-1.5, -1, -0.5]$

all represent the same point: $(3, 2)$.

Perspective Normalization: Given homogeneous coordinates $[x, y, w]$, where $w \neq 0$, divide by the last coordinate to yield $[x/w, y/w, 1]$. This represents the regular point $(x/w, y/w)$. 
Points at Infinity: We represent a point at infinity in some direction, say, \( u = (a, b) \) as \([a, b, 0]\).

Intuition: To see why this works, consider the homogeneous coordinates of a sequence of points lying on a line with slope 3/2. As \( k \to \infty \) the limit is \([2, 3, 0]\).

Notes: This limit is the same:
- for the sequence going in the opposite direction:
  \([-2k, -3k, 1] = [2, 3, -1/k] \to [2, 3, 0]\).
- even if the y-intercept is not zero: \([2k, 3k+b, 1] = [2, 3+(1/k), 1/k] \to [2, 3, 0]\).

Beware: Do not confuse the two distinct uses of homogeneous coordinates: affine geometry and projective geometry.

---

Perspective Projection: Simple Case

Assume that points have been transformed into view frame coordinates.
- \( p = (x, y, z) \) is the point to project.
- \( d \) is distance from the origin to the image plane.
- \( p' = (x', y', z') \) is the desired projected point.

Projection Transformation: \( p' = Tp \) where:
- By similar triangles, \( y/(-z) = y'/d \).
- Thus, \( y' = y/(-z/d) \).
- In standard coordinates we have:

\[
TP = \begin{pmatrix} x \\ -z/d \\ y \\ -z/d \\ -d \end{pmatrix} = \begin{pmatrix} x/(-z/d) \\ y/(-z/d) \\ -d/(-z/d) \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ -d \end{pmatrix}.
\]
**Perspective Projection: Simple Case**

Projection Transformation: This is not an affine transformation (due to division by \( z \)). Let's express in homogeneous coordinates.

\[
\begin{bmatrix}
  x \\
  y \\
  z \\
  1
\end{bmatrix}
= \begin{bmatrix}
  x / (-z/d) \\
  y / (-z/d) \\
  -d \\
  -z/d
\end{bmatrix}

**Good News:** We can represent this as a matrix-vector product:

\[
T_p = T \begin{bmatrix}
  x \\
  y \\
  z \\
  1
\end{bmatrix}
\]

**Perspective Projection: Simple Case**

Projection Transformation: Two-step process.

**Affine part:** Perform the matrix multiplication:

\[
T_p = \begin{bmatrix}
  1 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 \\
  0 & 0 & 1 & 0 \\
  0 & 0 & -1/d & 0
\end{bmatrix}
\begin{bmatrix}
  x \\
  y \\
  z \\
  1
\end{bmatrix}
= \begin{bmatrix}
  x \\
  y \\
  z \\
  -z/d
\end{bmatrix}

**Projective part:** Perform perspective normalization.

\[
T_p = \begin{bmatrix}
  x / (-z/d) \\
  y / (-z/d) \\
  -d \\
  1
\end{bmatrix}
\]

**Note:** This transformation is undefined if \( z = 0 \). (Actually it maps these regular points to points at infinity.) It can also map points at infinity to regular points (try it on \([x, y, z, 0]\)).
In-Class Exercise

Parallel lines meet at a common point vanishing point:
- Consider a collection of parallel lines drawn along the y = -1 plane.
- The i-th line can be expressed parametrically as

\[
L_i(s) = \begin{bmatrix} i + s \cdot b \\ -1 \\ -s \\ 1 \end{bmatrix} \quad \text{for } 0 \leq s \leq \infty
\]

- Compute the projection of the point \( L_i(s) \) by applying the above perspective transformation and perspective normalization.
- To find the vanishing point, \( v(b) \), compute the limit as \( s \to \infty \), and ignore the z-coordinate.
- Does the final result depend on \( i \)? (What does this imply?)
- As \( b \) (the slope) varies, what can you say about where the resulting vanishing points lie on the image plane?
Overview

- Overview of 3-d viewing process
- World-to-Camera transformation and gluLookAt()
- Parallel projection and glOrtho()
- Perspective projection and Projective geometry
  - Perspective in OpenGL: glFrustum() and gluPerspective()
- Normalized view volume and Perspective plus depth

Perspective Projection in OpenGL

Depth Information: As with glOrtho(), OpenGL computes depth information. You specify the nearest and farthest distances. glFrustum(left, right, bottom, top, near, far)

All arguments are of type GLdouble.

View frustum: All objects outside this frustum are clipped.
Symmetric Viewing: There is a simpler form of `glFrustum()` for when the viewing situation is symmetric about the z-axis:

```c
gluPerspective ( fovy, aspect, near, far ), where
```

- `fovy`: y-field of view (angle given in degrees)
- `aspect`: window's aspect ratio \( w/h = (\text{right-left})/(\text{top-bottom}) \).

```c
void myDisplay () {
    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glLoadIdentity ( );
    gluLookAt ( ... ); // set up view frame
    glMatrixMode ( GL_PROJECTION ); // set up projection
    glLoadIdentity ( );
    gluPerspective ( fovy, aspect, near, far ); // or glFrustum( )
    glMatrixMode ( GL_MODELVIEW );
    myWorld.draw ( ); // draw everything
    glutSwapBuffers ( );
}
```
Overview

- Overview of 3-d viewing process
- World-to-Camera transformation and gluLookAt()
- Parallel projection and glOrtho()
- Perspective projection and Projective geometry
- Perspective in OpenGL: glFrustum() and gluPerspective()
- Normalized view volume and Perspective plus depth

Normalized View Volume

Normalized View Volume: As in glOrtho(), the transformation will map objects within the view volume to a 2x2x2 cube, centered at the origin, called the normalized view volume.

z-Reversal: Near is mapped to \( n_z = -1 \) and far to \( n_z = +1 \).
Perspective + Depth

To see how gluPerspective() encodes depth information, let us first consider a simple case based on the following generic transformation matrix (for the case \( d = 1 \)).

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & a & -b \\
0 & 0 & -1 & 0
\end{bmatrix}
\begin{bmatrix}
x' \\
y' \\
az-b \\
-z
\end{bmatrix}
= \begin{bmatrix}
-x/z \\
-y/z \\
-a + (b/z) \\
1
\end{bmatrix}
\]

- This produces the desired \((x, y)\) projected coordinates (for \( d = 1 \)).
- The z-coordinate value \(-a + (b/z)\) encodes the depth information. Assuming that \( b \) is positive, as depth increases, \( z \) decreases, and so \(-a + (b/z)\) also increases.
- How shall we pick \( a \) and \( b \)?

Determining the Perspective + Depth Transformation:

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & a & -b \\
0 & 0 & -1 & 0
\end{bmatrix}
\begin{bmatrix}
x' \\
y' \\
az-b \\
-z
\end{bmatrix}
= \begin{bmatrix}
-x/z \\
-y/z \\
-a + (b/z) \\
1
\end{bmatrix}
\]

Want to map the z-interval \([-\text{near}, -\text{far}]\) to \([-1, +1]\), that is \(-a + (b/(-\text{far})) = +1 \) and \(-a + (b/(-\text{near})) = -1 \). Solving this system for \( a \) and \( b \) yields (let \( f = \text{far}, n = \text{near} \)):

\[
\begin{align*}
2 &= -(b/f) + (b/n) \\
2 &= b(1/n - 1/f) \\
b &= 2fn/(f - n)
\end{align*}
\]

Solving:

\[
\begin{align*}
a &= -(f+n)/(f-n) \\
a &= -a + b/(-f) \\
a &= -1 - (b/f)
\end{align*}
\]
**glFrustum( ) Transformation Matrix**

**glFrustum( ) Transformation**: We also need to scale and shear the x, y intervals at the near clipping plane (z = -near):

- **x**: [left, right] → [-1, +1]
- **y**: [bottom, top] → [-1, +1]

As we did with z, it is possible to set up and solve the appropriate equations to achieve this. We’ll skip this.

**Final Transformation Matrix**:

\[
\begin{bmatrix}
2 \cdot \text{near} & 0 & \text{right} + \text{left} & 0 \\
\text{right} - \text{left} & 2 \cdot \text{near} & \text{top} + \text{bottom} & 0 \\
0 & \text{top} - \text{bottom} & \text{far} + \text{near} & 2 \cdot \text{far} \cdot \text{near} \\
0 & 0 & \text{far} - \text{near} & -1
\end{bmatrix}
\]

These terms handle shear.

These terms scale the x and y coordinates.

**gluPerspective( ) Transformation Matrix**

**gluPerspective( ) Transformation**: A special case of glFrustum( )

- By symmetry we have right+left = top+bottom = 0, so no shearing.
- \(2 \cdot \text{near}/(\text{top-bottom}) = \cot (\text{fov}/2)\). (First convert fov, to radians).
- \(2 \cdot \text{near}/(\text{right-left}) = 2 \cdot \text{near}/((\text{top-bottom}) \cdot \text{aspect}) = (\cot (\text{fov}/2)) / \text{aspect}\).

**Final Transformation Matrix**:

\[
\begin{bmatrix}
\text{cot}(\text{fov}/2) & 0 & 0 & 0 \\
0 & \text{cot}(\text{fov}/2) & 0 & 0 \\
\text{Scale x and y} & \text{far} \cdot \text{near} & 2 \cdot \text{far} \cdot \text{near} & 1 \\
0 & 0 & \text{far} - \text{near} & 0
\end{bmatrix}
\]

No shear

Normalize: After multiplication do perspective normalization.
Summary of 3D Viewing

Eye transformation: Transforms points from your world coordinates to view coordinates, a coordinate frame centered at the camera.
- This should be done in Modelview transformation mode.
- It is usually the first transformation in your Modelview stack.

Projection: There are two types of projection: parallel (and orthogonal in particular) and perspective.

Projection+Depth: OpenGL uses the (x, y) coordinates of the projected point for drawing and the z-coordinate for depth.
- glOrtho(): (Orthogonal) parallel projection.
- glFrustum(): General perspective projection.
- gluPerspective(): Symmetric perspective projection.
  - These are done in Projection transformation mode.
  - OpenGL automatically does perspective normalization afterwards.

Summary

Summary:
- Camera Frame and Eye Coordinates
  - World-to-Camera Transformation and gluLookAt()
- Projection
  - Parallel projection and glOrtho()
  - Projective Geometry
  - Perspective projection and glFrustum() and gluPerspective()

What’s Next?
- Illumination