Overview

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

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:

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

Viewing in 3D

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

Transformations for 3D Viewing

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 ( ... )
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: gluLookAt()

In OpenGL the view frame is specified by calling:

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

All arguments are of type GLdouble. Where the following are given in world coordinates:

eye = (eyex, eyey, eyez) is the location of the eye.

```
att = (atx, aty, atz) is a point that the viewer is looking at. The vector
```

```
at - eye is the viewing direction.
```

```
up = (upx, upy, upz) is a vector indicating which direction is 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.

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 $v_x$, $v_y$, and $e$. 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 $TR$, where $T$ is a pure translation matrix (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 $M^{-1} = (TR)^{-1} = R^{-1}T^{-1} = R^TT^{-1}$.

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

Step 2-a: View direction is at - eye.
Normalize to unit length:

Step 2-b: $v_x = \text{up} \times v_y$ and normalize to get $v_x = v_x/\|v_x\|$.
Step 2-c: $v_y = v_x \times v_z$. (Normalization is not needed since $v_x$ and $v_y$ are already normalized and orthogonal.)

World Coordinates to View Coordinates

Step 2 (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:

$$ R = R^T. $$

The upper-left $3 \times 3$ submatrix is orthogonal, and thus the inverse is:

$$ R^{-1} = R^T. $$

The final view transformation matrix produced by gluLookAt() is $RT^{-1}$.

Where does gluLookAt() go?

Typical redisplay callback for 3D viewing:

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

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.

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

Depth Information:
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 GLdouble.

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.
The transformation matrix for glOrtho() transforms intervals:

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

The glOrtho() matrix is:

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

where:

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

Where does glOrtho() go?

Typical redisplay callback for 3D viewing:

```cpp
void myDisplay () {
    glClear ( GL_COLOR_BUFFER_BIT | ... ); // clear the buffer
    glLoadIdentity ( );
    glMatrixMode ( GL_PROJECTION );
    glOrtho ( left, right, bottom, top, near, far );
    glMatrixMode ( GL_MODELVIEW );
    myWorld.draw ( );
    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.

\[
\begin{bmatrix}
  3, 2, 1 \\
  6, 4, 2 \\
  -1.5, -1, -0.5
\end{bmatrix}
\]

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 \((x/w, y/w)\).
Homogeneous Coordinates for Projective Geom

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:
  \[ -2k, -3k, 1 \to [2, 3, 1/k] \]
- even if the y-intercept is not zero: \([2k, 3k+b, 1] \to [2, 3(1/k), 1/k] \to [2, 3, 0] \)

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

Perspective Projection: Simple Case

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

\[
\begin{pmatrix} x \, x/(z/d) \, 1 \end{pmatrix} T_p \begin{pmatrix} z \, -z/d \, 1 \end{pmatrix}
\]

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

\[
T_p = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1/d \end{pmatrix}
\]

Perspective Projection: Simple Case

Projection Transformation: Two-step process.

Affine part: Perform the matrix multiplication:

\[
\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1/d \end{pmatrix} \begin{pmatrix} x \, x/(z/d) \, 1 \end{pmatrix}
\]

Projective part: Perform perspective normalization.

\[
\begin{pmatrix} x/(z/d) \\ y/(z/d) \\ -z/d \end{pmatrix}
\]

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:

\[
L_i(s) = \begin{pmatrix} 1 \, s \, b \end{pmatrix} \text{ for } 0 \leq s < \infty
\]

Parallel lines meet at a common point vanishing point:
- 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 \( b \) (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

Symmetric Viewing: There is a simpler form of glFrustum() for when the viewing situation is symmetric about the z-axis:

gluPerspective (fovy, aspect, near, far), where
- fovy: y-field of view (angle given in degrees)
- aspect: window's aspect ratio w/h = (right-left)/(top-bottom).

glFrustum and gluPerspective(): As with glOrtho(), they generate a transformation matrix and multiply it times the top of the current matrix stack (usually the projection stack).

Example:

```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 ( );
}
```

Normalized View Volume

<table>
<thead>
<tr>
<th>nx</th>
<th>ny</th>
<th>nz</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>+1</td>
<td>-1</td>
</tr>
<tr>
<td>+1</td>
<td>+1</td>
<td>+1</td>
</tr>
</tbody>
</table>

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-Reduction: Near is mapped to nz = -1 and far to nz = +1.
Perspective + Depth

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

$$
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & 0 & -b \\
0 & a - b & 1 & 0 \\
0 & -1 & 0 & 1
\end{bmatrix}
$$

- This produces the desired $(x, y)$ projected coordinates (for $d = 1$).
- The $z$-coordinate value $-a + (b/2)$ encodes the depth information. Assuming that $b$ is positive, as $d$ increases, $z$ decreases, and so $-a + (b/2)$ also increases.
- How shall we pick $a$ and $b$?

Determine the Perspective + Depth Transformation:

$$
\begin{bmatrix}
x/2 \\
y/2 \\
-x/2 \\
-y/2
\end{bmatrix}
= 
\begin{bmatrix}
x/z \\
y/z \\
a + (b/2) \\
a - (b/2)
\end{bmatrix}
$$

want to map the $z$-interval $[-d, +d]$, that is $-a + (b/(d-near)) = -1$ and $-a + (b/(d-far)) = -1$.

Solving this system for $a$ and $b$ yields (let $f =$ far, $n =$ near):

$$
2 - (b/f) - (b/n) = -1 - a + (b/(d-near))
$$

$$
2 - b(f/n) - (1/f) = a - 1 - (b/f)
$$

$$
b = 2fn/(f-n)
$$

$$
a = -(f+n)/(f-n)
$$

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:

$$
\begin{bmatrix}
x \\
y \\
x/z \\
1
\end{bmatrix}
$$

- These terms handle shear.
- These terms scale the $x$ and $y$ coordinates.

**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.
- OpenGL uses the $(x, y)$ coordinates of the projected point for drawing and the $z$-coordinate for depth.
- `glOrtho()` (Orthogonal) parallel projection.
- `gluPerspective()` (General perspective projection).

Perspective + Depth: OpenGL automatically does perspective normalization afterwards.

**Case 1:**

$$
\begin{bmatrix}
x/2 \\
y/2 \\
-x/2 \\
-y/2
\end{bmatrix}
= 
\begin{bmatrix}
x/z \\
y/z \\
a + (b/2) \\
a - (b/2)
\end{bmatrix}
$$

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

$$
\text{Final Transformation Matrix:}
\begin{bmatrix}
\text{cot}(\text{fovy}/2) \\
0 \\
0 \\
0
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
x/z \\
1
\end{bmatrix}
$$

- Normalize: After multiplication do perspective normalization.

What’s Next?

- Illumination

---

Copyright © D. M. Mount and A. Varshney

Chapter 4, Slide 42
Copyright © D. M. Mount and A. Varshney

Chapter 4, Slide 43
Copyright © D. M. Mount and A. Varshney

Chapter 4, Slide 44
Copyright © D. M. Mount and A. Varshney

Chapter 4, Slide 45
Copyright © D. M. Mount and A. Varshney

Chapter 4, Slide 46
Copyright © D. M. Mount and A. Varshney

Chapter 4, Slide 47
Copyright © D. M. Mount and A. Varshney

Chapter 4, Slide 48
Copyright © D. M. Mount and A. Varshney