Computer Viewing

CS 432/537 Interactive Computer Graphics
Prof. David E. Breen
Department of Computer Science
Objectives

• Introduce the mathematics of projection
• Introduce OpenGL viewing functions
• Look at alternate viewing APIs
Computer Viewing

• There are three aspects of the viewing process, all of which should be / 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 object 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
Moving the Camera Frame

- If we want to visualize objects 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 \( \text{Translate}(0.0, 0.0, -d); \)
  - \( d > 0 \)
Moving Camera back from Origin

frames after translation by \(-d\)
\[ d > 0 \]

default frames

(a)

(b)
Moving Camera back from Origin

frames after translation by $-d$
$d > 0$

default frames

World is actually being moved relative to camera frame.
Moving the Camera

• We can move the camera to any desired position by a sequence of rotations and translations

• Example: side view
  - Rotate the scene
  - Move it away from origin
  - Model-view matrix $C = TR$
• Remember that last transformation specified is first to be applied

// Using mat.h

mat4 t = Translate(0.0, 0.0, -d);
mat4 ry = RotateY(-90.0);
mat4 m = t*ry;

This is code for moving world frame relative to camera frame. It is the inverse of the desired camera movement.
The LookAt Function

• The GLU library contained the function gluLookAt to form the required modelview matrix through a simple interface

• Note the need for setting an up direction
  - Should not be parallel to look-at direction

• Replaced by LookAt() in mat.h
  - Can concatenate with modeling transformations

• Example: isometric view of cube aligned with axes

  mat4 mv = LookAt(vec4 eye, vec4 at, vec4 up);
LookAt Function

LookAt(eye, at, up)
- eye - location of camera
- at - look-at point
- up - up vector

\((\text{eye}_x, \text{eye}_y, \text{eye}_z)\)

\((\text{at}_x, \text{at}_y, \text{at}_z)\)

\((u_p_x, u_p_y, u_p_z)\)
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
Homogeneous Coordinate Representation

default orthographic projection

\[ x_p = x \]
\[ y_p = y \]
\[ z_p = 0 \]
\[ w_p = 1 \]

\[ p_p = M p \]

\[ M = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix} \]

In practice, we can let \( M = I \) and set the \( z \) term to zero later.
Simple Perspective

- Center of projection at the origin
- Projection plane $z = d$, $d < 0$
Consider top and side views

\[ x_p = \frac{x}{z/d} \quad y_p = \frac{y}{z/d} \quad z_p = d \]
Homogeneous Coordinate Form

Consider \( q = Mp \) where \( M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \)

\[
q = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad \Rightarrow \quad p = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} = \begin{bmatrix} x/(d/z) \\ y/(d/z) \\ d \\ 1 \end{bmatrix}
\]
Perspective Division

• However $w \neq 1$, so we must divide by $w$ to return from homogeneous coordinates

• This \textit{perspective division} yields

$$
x_p = \frac{x}{z/d} \quad y_p = \frac{y}{z/d} \quad z_p = d
$$

the desired perspective equations

• We will consider the corresponding clipping volume with \texttt{mat.h} functions that are equivalent to deprecated OpenGL functions
OpenGL Orthogonal Viewing

Ortho(left, right, bottom, top, near, far)

near and far measured from camera
OpenGL Perspective

Frustum(left, right, bottom, top, near, far)
Using Field of View

• With Frustum it is often difficult to get the desired view.

• Perspective(fovy, aspect, near, far) often provides a better interface.

\[
\text{aspect} = \frac{w}{h}
\]
Projection Matrices

CS 432/537 Interactive Computer Graphics
Prof. David E. Breen
Department of Computer Science
Objectives

• Derive the projection matrices used for standard OpenGL projections
• Introduce oblique projections
• Introduce projection normalization
Normalization

• Rather than derive a different projection matrix for each type of projection, we can convert all projections to orthogonal projections with the default view volume.
• This strategy allows us to use standard transformations in the pipeline and makes for efficient clipping.
Pipeline View

1. modelview transformation
2. projection transformation
3. perspective division
   - 4D → 3D
4. clipping
5. projection
   - 3D → 2D
   - against default cube

nonsingular
Notes

• We stay in four-dimensional homogeneous coordinates through both the modelview and projection transformations
  - Both these transformations are nonsingular
  - Default to identity matrices (orthogonal view)

• Normalization lets us clip against simple cube regardless of type of projection

• Delay final projection until end
  - Important for hidden-surface removal to retain depth information as long as possible
Orthographic Normalization

Ortho(left, right, bottom, top, near, far)

normalization ⇒ find transformation to convert specified clipping volume to default

left < right    bottom < top    near < far
Orthographic Matrix

• Two steps
  
  - Move center to origin
    \[ T(-\frac{\text{left}+\text{right}}{2}, -\frac{\text{bottom}+\text{top}}{2}, \frac{\text{near}+\text{far}}{2}) \]
  
  - Scale to have sides of length 2
    \[ S(\frac{2}{\text{left-right}}, \frac{2}{\text{top-bottom}}, \frac{2}{\text{near-far}}) \]

\[ P = ST = \begin{bmatrix}
\frac{2}{\text{right} - \text{left}} & 0 & 0 & -\frac{\text{right} + \text{left}}{\text{right} - \text{left}} \\
0 & \frac{2}{\text{top} - \text{bottom}} & 0 & -\frac{\text{top} + \text{bottom}}{\text{top} - \text{bottom}} \\
0 & 0 & \frac{2}{\text{near} - \text{far}} & \frac{\text{far} + \text{near}}{\text{far} - \text{near}} \\
0 & 0 & 0 & 1
\end{bmatrix} \]
Final Projection

• Set $z = 0$

• Equivalent to the homogeneous coordinate transformation

$$M_{\text{orth}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

• Hence, general orthographic projection in 4D is

$$P = M_{\text{orth}} \mathbf{ST}$$
Oblique Projections

• The OpenGL projection functions cannot produce general parallel projections such as

• However if we look at the example of the cube it appears that the cube has been sheared

• Oblique Projection = Shear + Orthographic Projection
General Shear

Top view: The object is shown from above, with the x and z axes visible. The point \((x, z)\) is moved to \((x, y)\) with the angle \(\theta\).

Side view: The object is shown from the side, with the y and z axes visible. The point \((0, y_p)\) is moved to \((z, y)\) with the angle \(\phi\).

Back clipping plane and front clipping plane are also shown, along with the DOP (Distance of Projection).
Shear Matrix

**$xy$ shear (z values unchanged)**

$$H(\theta, \phi) = \begin{bmatrix} 1 & 0 & -\cot \theta & 0 \\ 0 & 1 & -\cot \phi & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

**Projection matrix**

$$P = M_{\text{orth}} H(\theta, \phi)$$

**General case:**

$$P = M_{\text{orth}} STH(\theta, \phi)$$
Equivalency
Effect on Clipping

• The projection matrix $P = STH$ transforms the original clipping volume to the default clipping volume.

- Object
- Top view
- Near plane
- Far plane
- Clipping volume
- Distorted object (projects correctly)

$z = -1$
$x = -1$
$z = 1$
$x = 1$
Consider a simple perspective with the COP at the origin, the near clipping plane at $z = -1$, and a 90 degree field of view determined by the planes $x = \pm z, y = \pm z$.
Perspective Matrices

Simple projection matrix in homogeneous coordinates

\[
M = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & -1 & 0 \\
\end{bmatrix}
\]

Note that this matrix is independent of the far clipping plane

\[
x' = x \\
y' = y \\
z' = z \\
w' = -z
\]

\[
x' = -\frac{x}{z} \\
y' = -\frac{y}{z} \\
z' = -1
\]
Generalization

\[ N = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{bmatrix} \]

\[ x' = x \]
\[ y' = y \]
\[ z' = \alpha z + \beta \]
\[ w' = -z \]

After perspective division, the point \((x, y, z, 1)\) goes to

\[ x'' = -x/z \]
\[ y'' = -y/z \]
\[ z'' = -(\alpha + \beta/z) \]

which projects orthogonally to the desired point regardless of \(\alpha\) and \(\beta\)
If we pick $\alpha = \frac{\text{near} + \text{far}}{\text{far} - \text{near}}$

$\beta = \frac{2\text{near} \times \text{far}}{\text{near} - \text{far}}$

the near plane is mapped to $z = -1$

the far plane is mapped to $z = 1$

and the sides are mapped to $x = \pm 1$, $y = \pm 1$

Hence the new clipping volume is the default clipping volume

$z'' = -(\alpha + \beta/z)$
Normalization Transformation

Original clipping volume

Original object

COP

New clipping volume

distorted object projects correctly

$z = -x$

$z = x$

$z = \text{-far}$

$z = \text{-near}$

$x = -1$

$x = 1$

$z = 1$

$z = -1$
Normalization and Hidden-Surface Removal

- Although our selection of the form of the perspective matrices may appear somewhat arbitrary, it was chosen so that if $z_1 > z_2$ in the original clipping volume then the transformed points $z_1' > z_2'$. 

- Thus hidden surface removal works if we first apply the normalization transformation.

- However, the formula $z'' = -(\alpha + \beta/z)$ implies that the distances are distorted by the normalization which can cause numerical problems especially if the near distance is small.
OpenGL Perspective

- Frustum allows for an unsymmetric viewing frustum (although Perspective does not)

Frustum(left, right, bottom, top, near, far)

$Z = Z_{\text{min}}$

$(x_{\text{min}}, y_{\text{min}}, z_{\text{max}})$

$(x_{\text{max}}, y_{\text{max}}, z_{\text{max}})$

COP

left < right
bottom < top
near < far

near & far positive
Perspective provides less flexible, but more intuitive perspective viewing.

```
Perspective(fov, aspect, near, far);
```

- Field of view in angles
- aspect: w/h
- near < far, both positive
OpenGL Perspective Matrix

• The normalization in Frustum requires an initial shear to form a right viewing pyramid, followed by a scaling to get the normalized perspective volume. Finally, the perspective matrix results in needing only a final orthographic transformation.

\[
P = \text{NSH}
\]

our previously defined perspective matrix  shear and scale
Why do we do it this way?

- Normalization allows for a single pipeline for both perspective and orthogonal viewing
- We stay in four dimensional homogeneous coordinates as long as possible to retain three-dimensional information needed for hidden-surface removal and shading
- We simplify clipping
Viewing OpenGL Code

• It’s not too bad, thanks to Ed Angel
• In application

```cpp
model_view = LookAt(eye, at, up);
projection = Ortho(left, right, bottom, top, near, far);

or

projection = Perspective(fov, aspect, near, far);
```

• In vertex shader

```cpp
gl_Position = projection*model_view*vPosition;
```
OpenGL code

• Remember that matrices are column major order in GLSL, so …

Transpose your matrices when sending them to the shaders!

```c
glUniformMatrix4fv(matrix_loc, 1, GL_TRUE,
                    model_view);
```