CS380: Computer Graphics

Viewing Transformation

Sung-Eui Yoon

Course URL:
http://sglab.kaist.ac.kr/~sungeui/CG/
Class Objectives (Ch. 7)

- Know camera setup parameters
- Understand viewing and projection processes
Viewing Transformations

- Map points from world spaces to eye space
  - Can be composed from rotations and translations
Viewing Transformations

- Goal: specify position and orientation of our camera
  - Defines a coordinate frame for eye space
“Framing” the Picture

- A new camera coordinate
  - Camera position at the origin
  - Z-axis aligned with the view direction
  - Y-axis aligned with the up direction

- More natural to think of camera as an object positioned in the world frame
Viewing Steps

- Rotate to align the two coordinate frames and, then, translate to move world space origin to camera’s origin
An Intuitive Specification

- Specify three quantities:
  - Eye point (e) - position of the camera
  - Look-at point (p) - center of the image
  - Up-vector ($\vec{u}_a$) - will be oriented upwards in the image
Deriving the Viewing Transformation

- First compute the look-at vector and normalize
  \[ \vec{l} = \vec{p} - \vec{e} \quad \hat{\vec{l}} = \frac{\vec{l}}{||\vec{l}||} \]

- Compute right vector and normalize
  - Perpendicular to the look-at and up vectors
    \[ \vec{r} = \vec{l} \times \vec{u}_a \quad \hat{\vec{r}} = \frac{\vec{r}}{||\vec{r}||} \]

- Compute up vector
  - \( \vec{u}_a \) is only approximate direction
  - Perpendicular to right and look-at vectors
    \[ \vec{u} = \hat{\vec{r}} \times \hat{\vec{l}} \]
Rotation Component

- Map our vectors to the cartesian coordinate axes
  
  \[
  \begin{bmatrix}
  1 & 0 & 0 \\
  0 & 1 & 0 \\
  0 & 0 & 1 \\
  \end{bmatrix} = \begin{bmatrix}
  \hat{r} & \hat{u} & -\hat{i} \\
  \end{bmatrix} R_v
  \]

- To compute $R_v$ we invert the matrix on the right
  - This matrix $M$ is orthonormal (or orthogonal) - its rows are orthonormal basis vectors: vectors mutually orthogonal and of unit length
  - Then, $M^{-1} = M^T$
  - So,

  \[
  R_v = \begin{bmatrix}
  \hat{r}^t \\
  \hat{u}^t \\
  -\hat{i}^t \\
  \end{bmatrix}
  \]
Translation Component

- The rotation that we just derived is specified about the eye point in world space
  - Need to translate all world-space coordinates so that the eye point is at the origin
  - Composing these transformations gives our viewing transform,

\[
V = R_v T_{-\hat{e}}
\]

\[
V = R_v T_{-\hat{e}} = \begin{bmatrix}
\hat{r}_x & \hat{r}_y & \hat{r}_z & 0 \\
\hat{u}_x & \hat{u}_y & \hat{u}_z & 0 \\
-\hat{l}_x & -\hat{l}_y & -\hat{l}_z & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & -e_x \\
0 & 1 & 0 & -e_y \\
0 & 0 & 1 & -e_z \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
\hat{r} \\
\hat{u} \\
-\hat{l} \\
0
\end{bmatrix} = \begin{bmatrix}
\hat{r} \\
\hat{u} \\
-\hat{l} \\
0
\end{bmatrix} - \hat{r} \cdot \hat{e} - \hat{u} \cdot \hat{e} - \hat{l} \cdot \hat{e}
\]

Transform a world-space point into a point in the eye-space
Viewing Transform in OpenGL

- OpenGL utility (glu) library provides a viewing transformation function:

  `gluLookAt (double eyex, double eyey, double eyez,
           double centerx, double centery, double centerz,
           double upx, double upy, double upz)`

  - Computes the same transformation that we derived and composes it with the current matrix
Example in the Skeleton Codes of PA2

```cpp
void setCamera ()
{
    // initialize camera frame transforms
    for (i=0; i < cameraCount; i++)
    {
        double* c = cameras[i];
        wld2cam.push_back(FrameXform());
        glPushMatrix();
        glLoadIdentity();
        gluLookAt(c[0], c[1], c[2], c[3], c[4], c[5], c[6], c[7], c[8]);
        glGetDoublev( GL_MODELVIEW_MATRIX, wld2cam[i].matrix() );
        glPopMatrix();
        cam2wld.push_back(wld2cam[i].inverse());
    }
    ....
}
```
Projections

- Map 3D points in eye space to 2D points in image space

- Two common methods
  - Orthographic projection
  - Perspective projection
Orthographic Projection

- Projects points along lines parallel to z-axis
  - Also called parallel projection
  - Used for top and side views in drafting and modeling applications
- Appears unnatural due to lack of perspective foreshortening

Notice that the parallel lines of the tiled floor remain parallel after orthographic projection!
Orthographic Projection

- The projection matrix for orthographic projection is very simple

\[
\begin{bmatrix}
  x' \\
  y' \\
  z' \\
  1
\end{bmatrix} = \begin{bmatrix}
  1 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 \\
  0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix}\begin{bmatrix}
  x \\
  y \\
  z \\
  1
\end{bmatrix}
\]

- Next step is to convert points to NDC
View Volume and Normalized Device Coordinates

- Define a view volume
- Compose projection with a scale and a translation that maps eye coordinates to normalized device coordinates
Orthographic Projections to NDC

\[
\begin{bmatrix}
  x' \\
  y' \\
  z' \\
  1
\end{bmatrix} = \begin{bmatrix}
  \frac{2}{\text{right} - \text{left}} & 0 & 0 & \frac{-(\text{right} + \text{left})}{\text{right} - \text{left}} \\
  0 & \frac{2}{\text{top} - \text{bottom}} & 0 & \frac{-(\text{top} + \text{bottom})}{\text{top} - \text{bottom}} \\
  0 & 0 & \frac{2}{\text{far} - \text{near}} & \frac{-(\text{far} + \text{near})}{\text{far} - \text{near}} \\
  0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
  x \\
  y \\
  z \\
  1
\end{bmatrix}
\]

Scale the z coordinate in exactly the same way. Technically, this coordinate is not part of the projection. But, we will use this value of z for other purposes.

Some sanity checks:

\[
x = \text{left} \Rightarrow x' = \frac{2 \cdot \text{left}}{\text{right} - \text{left}} - \frac{\text{right} + \text{left}}{\text{right} - \text{left}} = \frac{-\text{right} - \text{left}}{\text{right} - \text{left}} = -1
\]

\[
x = \text{right} \Rightarrow x' = \frac{2 \cdot \text{right}}{\text{right} - \text{left}} - \frac{\text{right} + \text{left}}{\text{right} - \text{left}} = \frac{\text{right} - \text{left}}{\text{right} - \text{left}} = 1
\]
Orthographic Projection in OpenGL

- This matrix is constructed by the following OpenGL call:

```c
void glOrtho(double left, double right,
             double bottom, double top,
             double near, double far);
```

- 2D version (another GL utility function):

```c
void gluOrtho2D( double left, GLdouble right,
                double bottom, GLdouble top);
```

, which is just a call to glOrtho( ) with near = -1 and far = 1
Perspective Projection

- Artists (Donatello, Brunelleschi, Durer, and Da Vinci) during the renaissance discovered the importance of perspective for making images appear realistic.
- Perspective causes objects nearer to the viewer to appear larger than the same object would appear farther away.
- Homogenous coordinates allow perspective projections using linear operators.
Signs of Perspective

- Lines in projective space always intersect at a point.
Perspective Projection

\[ y_s = d \frac{y}{z} \]
Perspective Projection Matrix

● The simplest transform for perspective projection is:

$$\begin{bmatrix} wx' \\ wy' \\ wz' \\ w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

● We divide by w to make the fourth coordinate 1

   ● In this example, w = z
   ● Therefore, x' = x / z, y' = y / z, z' = 0
Normalized Perspective

- As in the orthographic case, we map to normalized device coordinates (NDC).

![Diagram showing normalized perspective transformation from orthographic case to NDC](image-url)
NDC Perspective Matrix

\[
\begin{bmatrix}
w x' \\
w y' \\
w z' \\
w
\end{bmatrix} =
\begin{bmatrix}
\frac{2 \cdot \text{near}}{\text{right} - \text{left}} & 0 & -\frac{(\text{right} + \text{left})}{\text{right} - \text{left}} & 0 \\
0 & \frac{2 \cdot \text{near}}{\text{top} - \text{bottom}} & -\frac{(\text{top} + \text{bottom})}{\text{top} - \text{bottom}} & 0 \\
0 & 0 & \frac{\text{far} + \text{near}}{\text{far} - \text{near}} & -\frac{2 \cdot \text{far} \cdot \text{near}}{\text{far} - \text{near}} \\
0 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix}
\]

- The values of \text{left}, \text{right}, \text{top}, and \text{bottom} are specified at the \text{near} depth. Let’s try some sanity checks:

\[x = \text{left} \quad \Rightarrow \quad x' = \frac{2 \cdot \text{near} \cdot \text{left}}{\text{right} - \text{left}} - \frac{\text{near} \cdot (\text{right} + \text{left})}{\text{right} - \text{left}} = \frac{\text{near}}{\text{near}} = -1\]

\[z = \text{near} \quad \Rightarrow \quad x' = \frac{2 \cdot \text{near} \cdot \text{right}}{\text{right} - \text{left}} - \frac{\text{near} \cdot (\text{right} + \text{left})}{\text{right} - \text{left}} = \frac{\text{near}}{\text{near}} = 1\]
### NDC Perspective Matrix

\[
\begin{bmatrix}
wx' \\
wy' \\
wz' \\
w
\end{bmatrix} = \begin{bmatrix}
\frac{2 \cdot \text{near}}{\text{right} - \text{left}} & 0 & -\frac{(\text{right} + \text{left})}{\text{right} - \text{left}} & 0 \\
0 & \frac{2 \cdot \text{near}}{\text{top} - \text{bottom}} & -\frac{(\text{top} + \text{bottom})}{\text{top} - \text{bottom}} & 0 \\
0 & 0 & \frac{\text{far} + \text{near}}{\text{far} - \text{near}} & -\frac{2 \cdot \text{far} \cdot \text{near}}{\text{far} - \text{near}} \\
0 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix}
\]

- The values of left, right, top, and bottom are specified at the near depth. Let’s try some sanity checks:

\[
z = \text{far} \implies z' = \frac{\frac{\text{far} + \text{near}}{\text{far} - \text{near}} + \frac{-2 \cdot \text{far} \cdot \text{near}}{\text{far} - \text{near}}}{\text{far}} = \frac{\frac{\text{far} \cdot (\text{far} - \text{near})}{\text{far} - \text{near}}}{\text{far}} = 1
\]

\[
z = \text{near} \implies z' = \frac{\frac{\text{near} + \text{far} \cdot \text{near}}{\text{far} - \text{near}} + \frac{-2 \cdot \text{far} \cdot \text{near}}{\text{far} - \text{near}}}{\text{near}} = \frac{\frac{\text{near} \cdot (\text{near} - \text{far})}{\text{far} - \text{near}}}{\text{near}} = -1
\]
Perspective in OpenGL

- OpenGL provides the following function to define perspective transformations:
  
  ```
  void glFrustum(double left, double right, 
                  double bottom, double top, 
                  double near, double far);
  ```

- Some think that using `glFrustum()` is nonintuitive. So OpenGL provides a function with simpler, but less general capabilities
  
  ```
  void gluPerspective(double vertfov, double aspect, 
                      double near, double far);
  ```
gluPerspective()

Simple “camera-like” model
Can only specify symmetric frustums

- Substituting the extents into glFrustum()
**gluPerspective()**

- **Substituting the extents into glFrustum()**

\[
\begin{bmatrix}
wx' \\
wz' \\
wz' \\
w
\end{bmatrix} = \begin{bmatrix}
\frac{\cot(\text{vertfov}/2)}{\text{aspect}} & 0 & 0 & 0 \\
0 & \cot(\text{vertfov}/2) & 0 & 0 \\
0 & 0 & \frac{\text{far} + \text{near}}{\text{far} - \text{near}} & \frac{-2 \cdot \text{far} \cdot \text{near}}{\text{far} - \text{near}} \\
0 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix}
\]

Simple “camera-like” model

Can only specify *symmetric* frustums
Example in the Skeleton Codes of PA2

```c
void reshape( int w, int h)
{
    width = w;    height = h;
glViewport(0, 0, width, height);

    glMatrixMode(GL_PROJECTION);            // Select The Projection Matrix
    glLoadIdentity();                       // Reset The Projection Matrix
    // Define perspective projection frustum
    double aspect = width/double(height);

    gluPerspective(45, aspect, 1, 1024);    // Select The Projection Matrix
    glMatrixMode(GL_MODELVIEW);             // Select The Modelview Matrix

    glLoadIdentity();                     // Reset The Projection Matrix

}
```
Class Objectives were:

- Know camera setup parameters
- Understand viewing and projection processes
Homework

- Suggested reading:
  - Ch. 12, “Data Structure for Graphics”

- Watch SIGGRAPH Videos
- Go over the next lecture slides
PA3

- PA2: perform the transformation at the modeling space
- PA3: perform the transformation at the viewing space
Next Time

- Interaction