CS130: Computer Graphics
Lecture 6: Viewing Transformations

Tamar Shinar
Computer Science & Engineering
UC Riverside
Viewing Transformations
Viewing transformations

- Move objects from their 3D locations to their positions in a 2D view

The viewing transformation also project any pixels viewing ray back to the pixel’s position in image space.
Decomposition of viewing transforms

- Camera transform
  - rigid body transformation
  - place camera at origin

- Projection transform
  - x, y, z in [-1, 1]
  - depends on type of projection

- Viewport transform
  - map to pixel coordinates

Viewing transforms depend on: camera position and orientation, type of projection, field of view, image resolution

There are several names for these spaces: “camera space” = “eye space”, “canonical view volume” = “clip space” = “normalized device coordinates”, “screen space = pixel coordinates” and for the transforms: “camera transformation” = “viewing transformation”
Viewport transform

\[ (x, y, z) \rightarrow (x', y', z') \]

\[ (x, y, z) \in [-1, 1]^3 \quad x' \in [-0.5, n_x - 0.5] \]
\[ y' \in [-0.5, n_y - 0.5] \]
Viewport transform

Camera transform

Projection transform

Viewport transform

$M_{vp}$
Orthographic Projection Transform

<whiteboard>
Line drawing algorithm

construct $M_{vp}$
construct $M_{orth}$
$M = M_{vp}M_{orth}$

for each line segment $(a_i, b_i)$ do
  $p = Ma_i$
  $q = Mb_i$

drawline $(x_p, y_p, x_q, y_q)$

Shirley, Marschner 7.1

draw lines specified in camera space
Camera Transform

*How do we specify the camera configuration?*

*(orthogonal case)*
Camera Transform

*How do we specify the camera configuration?*

*e*ye *position*
Camera Transform

How do we specify the camera configuration?

\[ g \quad \text{gaze direction} \]
Camera Transform

How do we specify the camera configuration?

up vector
Camera Transform

How do we specify the camera configuration?
Camera Transform

\[ \mathbf{w} = -\frac{\mathbf{g}}{||\mathbf{g}||} \]
\[ \mathbf{u} = \frac{\mathbf{t} \times \mathbf{w}}{||\mathbf{t} \times \mathbf{w}||} \]
\[ \mathbf{v} = \mathbf{w} \times \mathbf{u} \]

\[ M_{\text{cam}} \ <\ \text{whiteboard} > \]
Line drawing algorithm

construct $M_{vp} M_{cam}$
construct $M_{orth}$
$M = M_{vp} M_{orth} M_{cam}$
for each line segment $(a_i, b_i)$ do
    $p = M a_i$
    $q = M b_i$
    drawline $(x_p, y_p, x_q, y_q)$

draw lines specified in world space

Shirley, Marschner 7.1
Perspective Viewing
rigid – translation and rotation only – parallel lines and angles are preserved
affine – scaling, shear, translation, rotation – parallel lines preserved, angles **not** preserved
projective – parallel lines and angles **not** preserved
Projective Transformations
note that the height, $y'$, in camera space is proportional to $y$ and inversely proportion to $z$. We want to be able to specify such a transformation with our 4x4 matrix machinery
Projective Transformations

Example:

\[
M = \begin{pmatrix}
2 & 0 & -1 \\
0 & 3 & 0 \\
0 & \frac{2}{3} & \frac{1}{3}
\end{pmatrix}
\]

Note: this makes our homogeneous representation for points unique only up to a constant
Projective Transformations

Example:

\[
M = \begin{pmatrix}
2 & 0 & -1 \\
0 & 3 & 0 \\
0 & \frac{2}{3} & \frac{1}{3}
\end{pmatrix}
\]

We can now implement perspective projection!
Perspective Projection

\[ y' = \frac{d}{z} y \]

both x and y get multiplied by d/z

note that both x and y will be transformed
This achieves a simple perspective projection onto the view plane $z = d$

but we’ve lost all information about $z$!
Perspective Projection

\[ P = \begin{pmatrix} n & 0 & 0 & 0 & 0 \\ 0 & n & 0 & 0 & 0 \\ 0 & 0 & n + f & -fn & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} \quad z' = (n + f) - \frac{nf}{z} \]

Example:

The perspective transformation does not preserve \( z \) completely, but it preserves \( z = n, f \) and is \textbf{monotone} (preserves ordering) with respect to \( z \).
So far we’ve mapped the view frustrum to a rectangular box. This rectangular box has the same near face as the view frustrum. The far face has been mapped down to the far face of the box. This mapping is given by $P$. The bottom figure shows how lines in the view frustrum get mapped to the rect. box.
We’re not quite done yet thought, because the projection transform should map the view frustrum to the canonical view volume.
We need a second mapping to get our points into the canonical view volume. This second mapping is a mapping from one box to another. So it’s given by an orthographic mapping, \( M_{\text{orth}} \). The final perspective transformation is the composition of \( P \) and \( M_{\text{orth}} \).

\[
M_{\text{per}} = M_{\text{orth}} P
\]
Line drawing algorithm

construct $M_{vp}$ $M_{cam}$

construct $M_{per}$

$M = M_{vp}M_{per}M_{cam}$

for each line segment $(a_i, b_i)$ do

$p = Ma_i$

$q = Mb_i$

drawline $(x_p/w_p, y_p/w_p, x_q/w_p, y_q/w_p)$

draw lines specified in world space

Note the two lines that have changed: 1. we put our perspective transformation into the overall transformation matrix $M$. 2. When we call the drawline function, we have to divide the $x$ and $y$ coordinates by the $w$ coordinate.
OpenGL Perspective Viewing

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

Here’s how you set up a perspective view in OpenGL. Note that near and far are both negative, but you pass their absolute values to OpenGL.
Using Field of View

With \texttt{glFrustum} it is often difficult to get the desired view \texttt{gluPerspective(fovy, aspect, near, far)} often provides a better interface.

Sometimes it’s more convenient to just give an angle, the field-of-view, and an aspect ratio, instead of \( l, r, t, b \). The \texttt{glu} library provides such a function. It will figure out \( l, r, t, b \), and call \texttt{glFrustum} for you.
Clipping after the perspective transformation can cause problems
OpenGL clips **after** projection and **before** perspective division

\[-w \leq x \leq w\]
\[-w \leq y \leq w\]
\[-w \leq z \leq w\]

[Shirley, Marschner]