Perspective transformations, transformation pipeline

Transformation pipeline

Modelview: model (position objects) + view (position the camera)
Projection: map viewing volume to a standard cube
Perspective division: project 3D to 2D
Viewport: map the square [-1,1]x[-1,1] in normalized device coordinates to the screen
Coordinate systems

World coordinates - fixed initial coord system; everything is defined with respect to it

Eye coordinates - coordinate system attached to the camera; in this system camera looks down negative Z-axis

Positioning the camera

- Modeling transformation: reshape the object, orient the object, position the object with respect to the world coordinate system
- Viewing transformation: transform world coordinates to eye coordinates
- Viewing transformation is the inverse of the camera positioning transformation
- Viewing transformation should be rigid: rotation + translation
- Steps to get the right transform: first, orient the camera correctly, then translate it
Positioning the camera

Viewing transformation is the inverse of the camera positioning transformation:

\[
\begin{bmatrix}
/c90/c82/c85/c79/c71 \\
/c72/c92/c72
\end{bmatrix}
\]

Camera positioning: translate by \((t_x, t_z)\)

Viewing transformation (world to eye):

\[
x_{\text{eye}} = x_{\text{world}} - t_z \\
z_{\text{eye}} = z_{\text{world}} - t_x
\]

Look-at positioning

Find the viewing transform given the eye (camera) position, point to look at, and the up vector

- Need to specify two transforms: rotation and translation.
- Translation is easy
- Natural rotation: define implicitly using a point at which we want to look and a vector indicating the vertical in the image (up vector)

Can easily convert the eye point to the direction vector of the camera axis; can assume up vector perpendicular to view vector
Look-at positioning

Problem: given two pairs of perpendicular unit vectors, find the transformation mapping the first pair into the second

\[ v = \frac{l - c}{|l - c|} \]

Determine rotation first, looking how coord vectors change:

\[
\begin{align*}
R \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} &= v \\
R \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} &= v \times u \\
R \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} &= u \\
R \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} &= R = [v \times u u - v]
\end{align*}
\]
Look-at positioning

Recall the matrix for translation:

\[
T = \begin{bmatrix}
1 & 0 & 0 & c_x \\
0 & 1 & 0 & c_y \\
0 & 0 & 1 & c_z \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Now we have the camera positioning matrix, \(TR\)

To get the viewing transform, invert: \((TR)^{-1} = R^{-1}T^{-1}\)

For rotation the inverse is the transpose!

\[
R^{-1} = [v \times u \ u - v]^T = \begin{bmatrix}
(v \times u)^T \\
u^T \\
-v^T
\end{bmatrix}
\]

Look-at viewing transformation

\[
T^{-1} = \begin{bmatrix}
1 & 0 & 0 & -c_x \\
0 & 1 & 0 & -c_y \\
0 & 0 & 1 & -c_z \\
0 & 0 & 0 & 1
\end{bmatrix} = [e_x \ e_y \ e_z \ -c]
\]

\[
V = R^{-1}T^{-1} = \begin{bmatrix}
(v \times u)^T & -(v \times u, c) \\
u^T & -(u, c) \\
-v^T & (v, c) \\
[0, 0, 0] & 1
\end{bmatrix}
\]
Positioning the camera in OpenGL

- imagine that the camera is an object and write a sequence of rotations and translations positioning it
- change each transformation in the sequence to the opposite
- reverse the sequence
- Camera positioning is done in the code before modeling transformations
- OpenGL does not distinguish between viewing and modeling transformation and joins them into the modelview matrix

Space to plane projection

In eye coordinate system

\[
\begin{align*}
C &= \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \\
V &= \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}
\end{align*}
\]

\[
\text{Proj}(p) = \begin{bmatrix} -
\frac{p_x}{p_z} \\ -
\frac{p_y}{p_z} \\ -1 
\end{bmatrix}
\]

\[
\text{Proj}(p) = \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
p_x \\
p_y \\
p_z 
\end{bmatrix}
\]
Visibility

Objects that are closer to the camera occlude the objects that are further away

- All objects are made of planar polygons
- A polygon typically projects 1 to 1
- Idea: project polygons in turn; for each pixel, record distance to the projected polygon
- When writing pixels, replace the old color with the new one only if the new distance to camera for this pixel is less than the recorded one

Z-buffering idea

- Problem: need to compare distances for each projected point
- Solution: convert all points to a coordinate system in which (x,y) are image plane coords and the distance to the image plane increases when the z coordinate increases
- In OpenGL, this is done by the projection matrix
Viewing frustum

Volume in space that will be visible in the image

\[ \gamma \] is the aspect ratio of the image width/height

Projection transformation

Maps the viewing frustum into a standard cube extending from -1 to 1 in each coordinate

(normalized device coordinates)

3 steps:
- change the matrix of projection to keep z:
- result is a parallelepiped
- translate:
- parallelepiped centered at 0
- scale in all directions:
- cube of size 2 centered at 0
Projection transformation

change

\[ \text{Proj}(\rho) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} \]

so that we keep \( z \):

\[ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \\ 1 \\ -p_z \end{bmatrix} \]

the homogeneous result corresponds to \( \begin{bmatrix} p_x/p_z \\ p_y/p_z \\ 1/p_z \end{bmatrix} \)

the last component increases monotonically with \( z \)!

---

Projection transformation

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

maps the frustum to an axis-aligned parallelepiped

already centered in \((x,y)\), center in \(z\)-direction and scale:

\[ T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 \end{bmatrix}; \quad S = \begin{bmatrix} \frac{1}{r \tan \frac{\alpha}{2}} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan \frac{\alpha}{2}} & 0 & 0 \\ 0 & 0 & \frac{2}{(\hat{z} - \hat{z})} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]
Projection transformation

Combined matrix, mapping frustum to a cube:

\[
P = ST = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & -1 & 0
\end{bmatrix} = \begin{bmatrix}
\frac{1}{\tan \frac{f}{2}} & 0 & 0 & 0 \\
0 & \frac{1}{\tan \frac{f}{2}} & 0 & 0 \\
0 & 0 & \frac{f + n}{n - f} & 2 \frac{fn}{n - f} \\
0 & 0 & \frac{n - f}{n - f} & 0
\end{bmatrix}
\]

To get normalized image plane coordinates (valid range [-1,1] both), just drop z in the result and convert from homogeneous to regular.

To get pixel coordinates, translate by 1, and scale x and y (Viewport transformation)

Transformation pipeline

Modelview: model (position objects) + view (position the camera)
Projection: map viewing volume to a standard cube
Perspective division: project 3D to 2D
Viewport: map the square [-1,1]x[-1,1] in normalized device coordinates to the screen