Perspective transformations
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:

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

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

Eye coords

World coords
Look-at positioning

Determine rotation first, looking how coord vectors change:

\[
R \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} = v, \quad R \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = v \times u, \quad R \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} = u
\]

\[
R \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = R = [v \times u, u, -v]
\]
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

\[
T^{-1} = \begin{bmatrix}
1 & 0 & 0 & -c_x \\
0 & 1 & 0 & -c_y \\
0 & 0 & 1 & -c_z \\
0 & 0 & 0 & 1
\end{bmatrix} = \begin{bmatrix}
ex & ey & ez & -c
\end{bmatrix}
\]

\[
V = R^{-1}T^{-1} = \begin{bmatrix}
(v \times u)^T & -(v \times u \cdot c) \\
u^T & -(u \cdot c) \\
-v^T & (v \cdot c) \\
[0, 0, 0] & 1
\end{bmatrix}
\]
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
In eye coordinate system

\[ c = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad v = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} \]

projecting to the plane \( z = -1 \)

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

- each pixel has storage for a z-value, in addition to RGB
- all objects are “scanconvertible” (typically are polygons, lines or points)

Algorithm:

initialize zbuf to maximal value

for each object
  for each pixel (i,j) obtained by scan conversion
    if znew(i,j) < zbuf(i,j)
      zbuf(i,j) = znew(i,j) ;
      write pixel(i,j)
What are z values?

Z values are obtained by applying the projection transform, that is, mapping the viewing frustum to the standard cube.

Z value increases with the distance to the camera.

Z values for each pixel are computed for each pixel covered by a polygon using linear interpolation of z values at vertices.

Typical Z buffer size: 24 bits (same as RGB combined).
Camera specification

Define the dimensions of the viewing volume (frustum)

- most general `glFrustum(left,right,bottom,top,near,far)`

In the picture:
- `l` = left
- `r` = right
- `b` = bottom
- `t` = top
- `n` = near
- `f` = far
- `s` = far/near
Camera specification

Less general but more convenient:
```
gluPerspective(field_of_view, aspect_ratio, near, far)
```

In the picture:
- `fov` = field of view,
- `h/w` = `a` = aspect ratio

Relationship to frustum:
- `left` = `-a*near*tan(fov/2)`
- `right` = `a*near*tan(fov/2)`
- `bottom` = `-a*near*tan(fov/2)`
- `top` = `a*near*tan(fov/2)`

`gluPerspective` requires `fov` in degrees, not radians!
Viewing frustum

Volume in space that will be visible in the image

$r$ 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

\[
\text{Proj}(p) = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & -1 & 0
\end{bmatrix}
\begin{bmatrix}
px \\
py \\
pz \\
1
\end{bmatrix}
\]

so that we keep \( z \):

\[
\begin{bmatrix}
p_x \\
p_y \\
p_z \\
1
\end{bmatrix} = \begin{bmatrix}
px \\
py \\
pz \\
1
\end{bmatrix}
\]

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

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

maps the frustum to an axis-aligned parallelepiped

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

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

\[
S = \begin{bmatrix}
\frac{1}{r \tan \frac{\alpha}{2}} & 0 & 0 & 0 \\
0 & \frac{1}{\tan \frac{\alpha}{2}} & 0 & 0 \\
0 & 0 & 2 \left( \frac{1}{n - \frac{1}{f}} \right) & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
Projection transformation

Combined matrix, mapping frustum to a cube:

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

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)