Pinhole Camera
- ingredients
  - box, film, hole punch
  - result
  - picture

Real Cameras
- pinhole camera has small aperture (lens opening)
  - minimize blur
  - problem: hard to get enough light to expose the film
  - solution: lens
    - permits larger apertures
    - permits changing distance to film plane without actually moving it
    - cost: limited depth of field where image is in focus

Projections I
- review: world to view coordinates
  - translate eye to origin
  - rotate view vector (lookat – eye) to w axis
  - rotate around w to bring up into vw-plane
- perspective projection
  - our camera must model perspective
  - our camera must model perspective
- projective transformations
  - planar geometric projections
    - planar: onto a plane
    - geometric: using straight lines
    - projections: 3D -> 2D
    - aka projective mappings
  - counterexamples?

Graphics Cameras
- real pinhole camera: image inverted
- computer graphics camera: convenient equivalent

Review: Camera Motion
- rotate/translate/scale difficult to control
- arbitrary viewing position
  - eye point, gaze/lookat direction, up vector

Review: World to View Coordinates
- translate eye to origin
- rotate view vector (lookat – eye) to w axis
- rotate around w to bring up into vw-plane

General Projection
- image plane need not be perpendicular to view plane

Perspective Projection
- our camera must model perspective
  - through common center of projection (eye point)
  - onto an image plane

Perspective Projection
- project all geometry
  - onto an image plane

Projective Transformations
- properties
  - lines mapped to lines and triangles to triangles
    - parallel lines do NOT remain parallel
      - e.g. rails vanishing at infinity
    - affine combinations are NOT preserved
      - e.g. center of a line does not map to center of projected line (perspective foreshortening)

Basic Perspective Projection

- similar triangles

\[ \begin{align*}
\text{P'(x',y',z')} &= \frac{x'}{z'} \cdot \text{P(x,y,z)} \\
&= \frac{y'}{z'} \cdot \text{y} \\
&= \frac{z'}{z} \cdot \text{d} \\
&= \frac{x'}{z'} \cdot \frac{y'}{z'} \cdot \frac{z'}{z} \cdot \text{d}
\end{align*} \]

- nonuniform foreshortening
- not affine

Perspective Projection

- desired result for a point \([x, y, z, 1]^T\) projected onto the view plane:

\[ \begin{align*}
x' &= \frac{x}{z} \\
y' &= \frac{y}{z} \\
z' &= \frac{z}{d}
\end{align*} \]

- what could a matrix look like to do this?

Simple Perspective Projection Matrix

\[ \begin{bmatrix}
x \\
y \\
z \\
d
\end{bmatrix}
\]

is homogenized version of where \(w = z/d\)

Moving COP to Infinity

- as COP moves away, lines approach parallel
- when COP at infinity, orthographic view

Orthographic Camera Projection

\[ \begin{bmatrix}
x_p \\
y_p \\
z_p
\end{bmatrix}
\]

Why Canonical View Volumes?

- permits standardization
- clipping
- easier to determine if an arbitrary point is enclosed in volume with canonical view volume vs. clipping to six arbitrary planes
- rendering
- projection and rasterization algorithms can be reused

Canonical View Volumes

- standardized viewing volume representation
  - perspective
  - orthographic
  - orthogonal
  - parallel
- x or y

Understanding Z

- near, far always positive in OpenGL calls
- RHS before projection (eye/view coords)
- LHS after projection (clip, norm device coords)

Normalized Device Coordinates

- convention
  - viewing frustum mapped to specific parallelepiped
  - Normalized Device Coordinates (NDC)
    - same as clipping coords
    - only objects inside the parallelepiped get rendered
    - which parallelepiped? - depends on rendering system

Frustum

Frustum

Parallelepiped (box)
Understanding Z

• why near and far plane?
  • near plane:
    • avoid singularity (division by zero, or very small numbers)
  • far plane:
    • store depth in fixed-point representation (integer), thus have to have fixed range of values (0...1)
    • avoid/reduce numerical precision artifacts for distant objects

Orthographic Derivation

• scale, translate, reflect for new coord sys
  \[ y' = a \cdot y + b \]
  \[ y = \text{top} \rightarrow y' = 1 \]
  \[ y = \text{bot} \rightarrow y' = -1 \]

Orthographic OpenCL

  `glMatrixMode(GL_PROJECTION);`  
  `glLoadIdentity();`  
  `glOrtho(left, right, bot, top, near, far);`