camera transforms

• until now we have considered all of our geometry in a 3d space
• ultimately everything ended up in eye coordinates with coordinates $[x_e, y_e, z_e, 1]^t$.
• we said that the camera is placed at the origin of the eye frame $e^e$, and that it is looking down the eye’s negative $z$-axis.
• this somehow produces a 2d image.
• we had a magic matrix which created $\text{gl} \_\text{Position}$
• now we will study this step.

Pinhole Camera model

• see fig
• As light travels towards the film plane, most is blocked by an opaque surface placed at the $z_e = 0$ plane.
• But we place a very small hole in the center of the surface, at the point with eye coordinates $[0, 0, 0, 1]^t$.
• Only rays of light that pass through this point reach the film plane and have their intensity recorded on film. The image is recorded at a film plane placed at, say, $z_e = 1$.
  
  – a physical camera needs a finite aperture and a lens, but we will ignore this.
• to avoid the image flip, we can mathematically model this with the film plane in front of the pinhole, say at the $z_e = -1$
• if we hold up the photograph at the $z_e = -1$ plane, and observe it with our own eye placed at the origin (see Figure), it will look to us just like the original scene would have.

Basic Mathematical Model

• let $\tilde{p}$ have eye coordinates $[x_e, y_e, z_e]^t$
• where does the ray from $\tilde{p}$ to the origin hits the film plane?
• all points on the ray hit the same pixel.
• all points on the ray are all scales of each other
• Let us use eye coordinates $[x_n, y_n, -1]^t$ to specify the hit point on our film plane.
• all points on ray must be of the form : $\alpha[x_n, y_n, -1]^t$
• so $[x_e, y_e, z_e]^t$ is of this form with $\alpha = -z_e$.
• $[x_e, y_e, z_e]^t = -z_e[x_n, y_n, -1]^t$
• so

$$x_n = -\frac{x_e}{z_e}$$

$$y_n = -\frac{y_e}{z_e}$$

in matrix form

• We can model this expression as a matrix operation as follows.

$$\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
-1 & -1 & 0 & 0 \\
0 & 0 & -1 & 1
\end{bmatrix}
\begin{bmatrix}
x_e \\
y_e \\
z_e \\
w_e
\end{bmatrix}
= 
\begin{bmatrix}
x_c \\
y_c \\
y_n \cdot w_c \\
w_n
\end{bmatrix}$$

• The raw output of the matrix multiply, $[x_c, y_c, -, w_c]^t$, are called the clip coordinates of $\tilde{p}$. 

1
• \( w_n = w_c \) is a new variable called the w-coordinate.

  – In such clip coordinates, the fourth entry of the coordinate 4-vector is not necessarily a zero or a one.

divide by w

• We say that \( x_n w_n = x_c \) and \( y_n w_n = y_c \).

• To extract \( x_n \) alone, we must perform the division \( x_n = \frac{x_n w_n}{w_n} \)

• this recovers our camera model

• Our output coordinates, with subscripts “n”, are called normalized device coordinates because they address points on the image in abstract image units without specific reference to numbers of pixels.

• we keep all of the image data in the canonical square \(-1 \leq x_n \leq +1, -1 \leq y_n \leq +1\), and ultimately map this onto a window on the screen.

  – Data outside of this square is not be recorded or displayed.
  
  – This is exactly the model we used to describe 2D OpenGL visibility

• note that at this point, all points along the line will project to the same film position

• but in the real world, we only see the one in front.

• also note that even points along the line backwards behind the eye will also map to the same film position

• but in the real world, cameras do not see behind themselves.

• we will deal with these issues in due time.

scales

• By changing the entries in the projection matrix we can slightly alter the geometry of the camera transformation.

• we could push the film plane out to \( z_e = n \), where \( n \) is some negative number (zoom lens)

• eye coordinates of points on the film are of the form: \([x_n, y_n, n]^t\)

• points along its ray are of the form \( \alpha[x_n, y_n, n]^t\)

• so a point \([x_e, y_e, z_e]^t\) on the ray must satisfy \([x_e, y_e, z_e]^t = \frac{z_e}{n}[x_n, y_n, n]^t\)

• and so

\[
\begin{align*}
x_n &= \frac{x_e n}{z_e} \\
y_n &= \frac{y_e n}{z_e}
\end{align*}
\]

in matrix form

• In matrix form, this becomes

\[
\begin{pmatrix}
x_n w_n \\
y_n w_n \\
- \\
w_n
\end{pmatrix} = \begin{pmatrix}
-n & 0 & 0 & 0 \\
0 & -n & 0 & 0 \\
0 & 0 & -1 & 0
\end{pmatrix} \begin{pmatrix}
x_e \\
y_e \\
z_e \\
1
\end{pmatrix}
\]

• note this matrix is the same as

\[
\begin{pmatrix}
-n & 0 & 0 & 0 \\
0 & -n & 0 & 0 \\
0 & 0 & 1
\end{pmatrix} \begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & -1 & 0
\end{pmatrix}
\]

2
• this has the same effect as starting with our original camera, scaling the image by $-n$, and cropping to the canonical square.
• so at this point, it we may as well think of the $[x_n, y_n]^T$ as coordinates resulting from a camera transform (and not the eye coordinates of points on a film plane).
  - so now we can simply think up some constraints on this transform and find the correct matrix.

**fovY**

• scale can be determined by vertical angular field of view of the desired camera.
• if we want our camera to have a field of view of $\theta$ degrees.
• we want any point with the maximal vertical angle to map to the boundary of the image.
• one such point has eye coordinates: $[0, \tan(\frac{\theta}{2}), -1, 1]^T$
• we want the point with eye coordinates: $[0, \tan(\frac{\theta}{2}), -1, 1]^T$ maps to normalized device coordinates $[0, 1]^T$.
• so we can use the matrix

$$
\begin{bmatrix}
\frac{1}{\tan(\frac{\theta}{2})} & 0 & 0 & 0 \\
0 & \frac{1}{\tan(\frac{\theta}{2})} & 0 & 0 \\
- & - & - & - \\
0 & 0 & -1 & 0
\end{bmatrix}
$$

(5)

dealing with aspect ratio

• Suppose the window is wider than it is high. In our camera transform, we need to squish things horizontally so a wider horizontal field of view fits into our retained canonical square.
• When the data is later mapped to the window, it will be stretched out correspondingly and will not appear distorted.
• Define $a$, the aspect ratio of a window, to be its width divided by its height (measured say in pixels).
• We can then set our projection matrix to be

$$
\begin{bmatrix}
\frac{1}{a \tan(\frac{\theta}{2})} & 0 & 0 & 0 \\
0 & \frac{1}{\tan(\frac{\theta}{2})} & 0 & 0 \\
- & - & - & - \\
0 & 0 & -1 & 0
\end{bmatrix}
$$

(6)
• so when the window is wide, we will keep more horizontal FOV, and when the window is tall, we will keep less horizontal FOV
  - but in asst1, we liked the behavior that didn’t crop any data as the aspect ration went above or below 1.
  - so in our code, we specify a minFov, and check the aspect ratio, and alter the fovY when the window is tall

**FOV issues**

• to be a “window” onto the world, the FOV should match the angular extents of the window in the viewers field.
• this might give a too limited view onto the world
• so we can increase it to see more
• but this might give a somewhat unnatural look (demo)

**motivate shifts**

• look at the two street pics
  - what is the difference between the two cameras.
  - hint: if a geometric plane recedes from the film, it appears smaller.
• imagine we want to model the screen as a window, what should this camera look like
Shifts

- sometimes, we wish to crop the image non-centrally
- this can be modeled as translating the NDC’s and then cropping centrally.

\[
\begin{bmatrix}
x_n w_n \\
y_n w_n \\
- \\
w_n \\
\end{bmatrix} = \begin{bmatrix}
1 & 0 & 0 & c_x \\
0 & 1 & 0 & c_y \\
0 & 0 & 1 & 0 \\
0 & 0 & -1 & 0 \\
\end{bmatrix} \begin{bmatrix}
x_e \\
y_e \\
z_e \\
1 \\
\end{bmatrix}
\]

- also useful for tiled displays, stereo viewing on a single screen.

Frustum

- shifts are often specified by first specifying a near plane \( z_e = n \).
- On this plane, a rectangle is specified with the eye coordinates of an axis aligned rectangle. (For non-distorted output, the aspect ratio of this rectangle should match that of the final window.)
  - using \( l, r, t, b \).

\[
\begin{bmatrix}
-\frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\
0 & -\frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\
0 & 0 & -1 & 0 \\
\end{bmatrix}
\]

- (fig)

Context

- projection could be applied to every point in the scene
- in CG, we will apply it to the verts to position a triangle on the screen
- the rest of the triangle will then get filled in on the screen as we shall see.