

projection and pin hole cameras

- need way of going from 3d to 2d
- will model this using a projection matrix
- we will look at how “z” should be dealt with
- we will talk about how to properly do texture mapping

pinhole model

- a pinhole and a plane
 - described in the eye coordinate system
 - plane at $z_e = 1$
- film on back plane records intensity along ray
- when film is developed, we get a flip
- so we can think of the film in front
 - plane at $z_e = -1$
- this works since we can replace the pinhole by your eye
 - even if the geometry is not exact, it still looks realistic

pinhole tform

- the pinhole camera can be modeled as

$$\begin{aligned}x_n &= -x_e/z_e \\ y_n &= -y_e/z_e\end{aligned}$$

- can sort of be done as a matrix operation

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

- the raw data is called clip coordinates
- must divide out the w_n
- the “n” stands for normalized device coordinates
- the assumption is that the $[-1.. +1]$ range in x and y are kept and mapped to the window
 - but here we make no reference to the windows size in pixels.

change the image plane

- put image plane at $z = n$
 - We will typically be thinking of n as a negative number.

$$\begin{aligned}x_n &= x_e n / z_e \\ y_n &= y_e n / z_e\end{aligned}$$

- which can be expressed as the matrix equation

$$\begin{bmatrix} x_n w_n \\ y_n w_n \\ - \\ w_n \end{bmatrix} = \begin{bmatrix} -n & 0 & 0 & 0 \\ 0 & -n & 0 & 0 \\ - & - & - & - \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix}$$

scaled image

- put the film back to $z = -1$
- enlarge the image by a factor of s_x horizontally and s_y vertically.
- composition of two appropriate matrices.
 - We can delay the division by w_n until the end.

$$\begin{aligned} \begin{bmatrix} x_n w_n \\ y_n w_n \\ - \\ w_n \end{bmatrix} &= \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ - & - & - & - \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ - & - & - & - \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix} \end{aligned}$$

- so changing focal distance is same as uniform scaling the image

API: perspective

- recall that the bounds of the picture in n.d. coordinates are between -1 and 1
- the projection matrix maps the desired region of the world to the canonical square.
- one can just give vertical field of view θ , and the aspect ratio $a = w/h$
- matrix is then

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

unsquished later

- plug in points on the $z_e = -1$ plane to see who maps to [-1..1].
- for non unit ar, this will produce have some squishing in ndc's
- but this will get mapped onto a non square window, which will undo the squishing
- just like in your assignment 1

minFOV

- instead of fixing the vertical fov, we can fix a min fov, and use the aspect ratio to determine the matrix.
 - for resizing
- if $a > 1.0$ then use above, else use

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

skewed camera

- closest point of pinhole to plane has n.d. coordinates $[0, 0]^t$.
- suppose we shift all of the n.d. coordinates (lower the image plane) by $[c_x, c_y]^t$
- the center (the projection of the z axis) is now at $[c_x, c_y]^t$

- We call such a camera a “skewed” camera.

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

why skewed cameras

- assume that we “crop” to +-1
- then skewed camera has different geometry
- actual cameras are typically somewhat skewed
- may want to tile screens with viewers eye not in center of screen
 - flight simulator
 - tiled walls
 - stereo viewing

general form: skew and scale

$$\begin{bmatrix} 1 & 0 & 0 & c_x \\ 0 & 1 & 0 & c_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ - & - & - & - \\ 0 & 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} s_x & 0 & -c_x & 0 \\ 0 & s_y & -c_y & 0 \\ - & - & - & - \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

full frustum model

- image plane placed at at $z = n$,
 - n stands for the near plane.
 - typically n will be a negative number.
- user then describes the boundaries of the valid image domain on the near plane

$$\begin{aligned} l &< x_e < r \\ b &< y_e < t \end{aligned}$$

- one should choose the aspect ratio of the bounds $\frac{r-l}{t-b}$ to be equal to the aspect ratio of the final w by h pixel image.
- we can map to canonical square using the “frustum” matrix

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

we need a z coordinate

- a camera does not record z
- we need it for visibility (z-buffer)

- proposal: compute a z_n using

$$\begin{bmatrix} x_n w_n \\ y_n w_n \\ z_n w_n \\ w_n \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix}$$

- $z_n = -1/z_e$
- we now have a 3D to 3D transform
- we will call this a “projective” transform

why this works

- model 1: apply this transform to each point of the geometry.
- for points in front of the eye $z_e < 0$, this transform preserves depth ordering
- $z_0 < z_1 \Leftrightarrow -1/z_0 < -1/z_1$.

...why this works

- model 2: apply this only to the 3 vertices of a triangle.
- at each pixel in the screen compute z_n value using “affine” (linear) interpolation
- correct for triangles, incorrect for not flat things
- so we need the projective transform to map flat triangles to flat triangles.
- thm: projective transforms map flat shapes to flat shapes
 - (this would not be true if we defined our pointwise transform with $z_n = z_e$).

a detail z resolution

- z_n makes poor use of its bits
- As z_e increases toward zero, the calculated z_n grows to infinity
- as $z_e \ll 0$, then z_n distances get compressed unresolvable close
- instead use

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

- if a and b are positive, then z order for objects in front of eye are maintained.

near and far

- in practice we define a near plane at $z_e = n$ and a far plane at $z_e = f$
- and set

$$\begin{aligned} a &= \frac{f+n}{f-n} \\ b &= -\frac{2fn}{f-n} \end{aligned}$$

- this maps $z_e \in [f..n]$ reasonably well to $z_n \in [-1..1]$.
- we then clip everything outside of this range

- near plane cant be too close to $z_e = 0$
- disadvantage: must add this sort of arbitrary clipping

frustum II

- adding the near and far to the frustum matrix, we get

$$\begin{bmatrix} -\frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & -\frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

- full disclosure: OpenGL by default assumes flipped z and then uses “smaller is closer” in depth test
 - change the depth test sign.
 - (also could multiply 3rd row by -1 .)

lets now put this in context

- at the end of the vertex shader, compute “clip coordinates” of each point and put it in `gl_position`.

$$\begin{bmatrix} x_n & w_n \\ y_n & w_n \\ z_n & w_n \\ w_n & \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix} = \mathbf{PM} \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix}$$

- three vertices of one triangle are grabbed by the primitive assembler.
- the triangle is now “clipped” to the $[-1..1]$ cube (more later)
- it then divides by w_n to obtain normalized device coordinates
- the “rasterizer” then takes over and finds all of the samples (pixels) that are inside a triangle
- for each sample location x_n, y_n , it computes the z_n value using affine interpolation.
 - more later about other interpolated data.
- this data is sent by the rasterizer out to the pixel shader to determine the color
- the pixel color on the screen is updated iff the point is in front of whatever is already there.