CS380: Introduction to Computer Graphics

Projection

Chapter 10

Min H. Kim

KAIST School of Computing

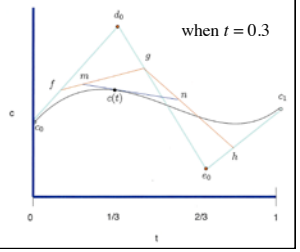
Smooth Interpolation

SUMMARY

Bezier evaluation

- To evaluate the function $c(t)$ at any value of t, we perform the following sequence of linear interpolations:

$$f = (1-t)c_0 + td_0$$
$$g = (1-t)d_0 + te_0$$
$$h = (1-t)e_0 + tc_0$$
$$m = (1-t)f + tg$$
$$n = (1-t)g + th$$
$$c(t) = (1-t)m + tn$$

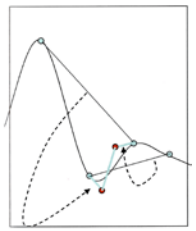


Catmull-Rom Splines construction

- This also can be stated as $c'(t)|_{i+1} = \frac{1}{2}(c_{i+2} - c_i)$
- Since $c'(i) = 3(d_i - c_i)$ and $c'(i+1) = 3(c_{i+1} - e_i)$, in the Bezier representation, this tells us that we need to set:

$$d_i = \frac{1}{6}(c_{i+1} - c_{i-1}) + c_i$$

$$e_i = \frac{-1}{6}(c_{i+2} - c_i) + c_{i+1}$$



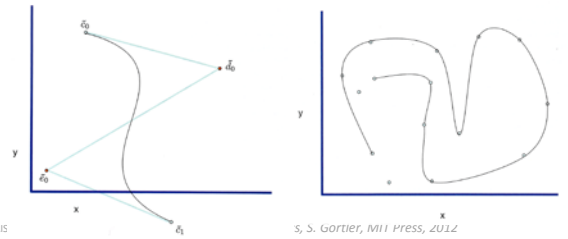
Quaternion Splining

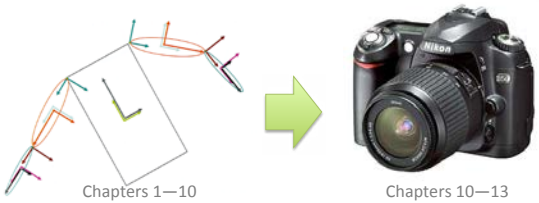
- Bezier evaluation steps of the form:
$$r = (1-t)p + tq$$
- Become: r = slerp(p, q, t)
- And the d_i and e_i equation values are defined as
$$d_i = ((c_{i+1}c_{i-1}^{-1})^{1/6})c_i \quad \Leftarrow d_i = \frac{1}{6}(c_{i+1} - c_{i-1}) + c_i$$
$$e_i = ((c_{i+2}c_i^{-1})^{-1/6})c_{i+1} \quad \Leftarrow e_i = \frac{-1}{6}(c_{i+2} - c_i) + c_{i+1}$$
- In order to interpolate "the short way", we do conditional negation on $(c_{i+1}c_{i-1}^{-1})$ before applying the power operator.

Curves

- We can cleanly apply the scalar theory to describe curves in the plane or space.
- The spline curve is controlled by a set of *control points* $\tilde{c}_i$ in 2D or 3D.

Bezier curve (2D) Catmull-Rom curve (2D)





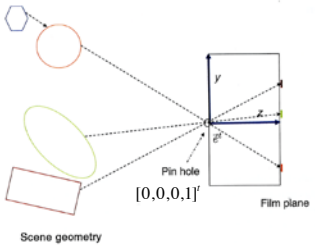
Chapter 10

PROJECTION

Camera transforms

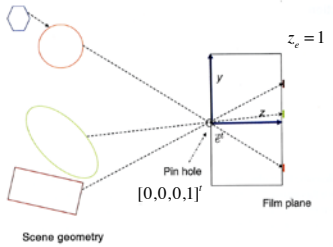
- Until now we have considered all of our geometry in a 3D space
- Ultimately everything ended up in eye coordinates with coordinates $[x_e, y_e, z_e, 1]^t$
- We said that the camera is placed at the origin of the eye frame $\vec{\mathbf{e}}^t$, and that it is looking down the eye's negative z-axis.
- This somehow produces a 2D image.
- We had a magic matrix which created gl_Position
- Now we will study this step

Pinhole camera model



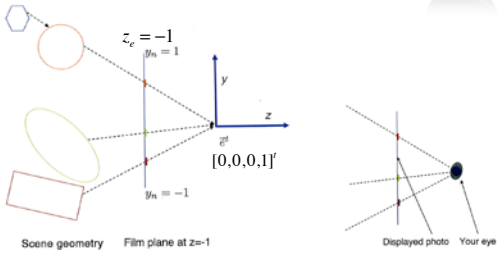
- As light travels towards the film plane, most is blocked by an opaque surface placed at the $z_e = 0$ plane.
- But we place a very small hole in the center of the surface, at the point with eye coordinates $[0,0,0,1]^t$

Pinhole camera model



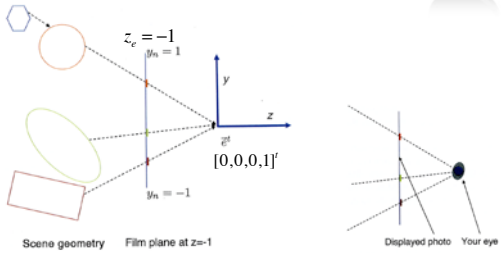
- Only rays of light that pass through this point reach the film plane and have their intensity recorded on film. The image is recorded at a film plane placed at, say, $z_e = 1$

Pinhole camera model



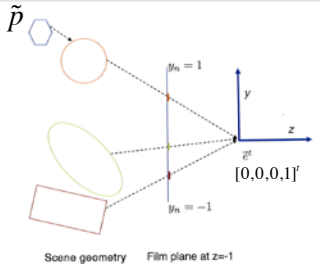
- A physical camera needs a finite aperture and a lens, but we will ignore this.
- To avoid the image flip, we can mathematically model this with the film plane in front of the pinhole, say at the $z_e = -1$

Pinhole camera model



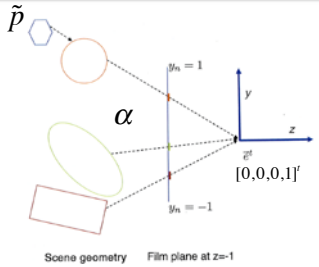
- If we hold up the photograph at the $z_e = -1$ plane, and observe it with our own eye, placed at the origin, it will look to us just like the origin scene would have.

Basic mathematical model



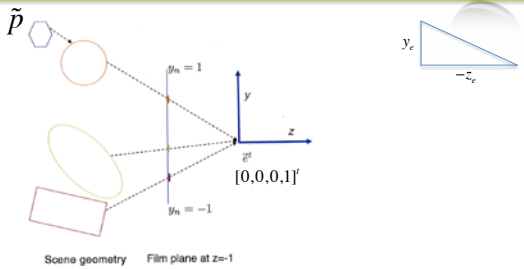
- Let us use normalized coordinates $[x_n, y_n]^t$ to specify points on our film plane.
  - For now, let them match eye coordinates on this film plane.
- Where does the ray from $\tilde{p}$ to the origin hits the film plane?

Basic mathematical model



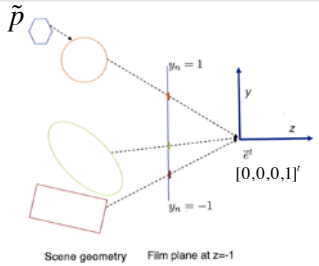
- All points on the ray hit the same pixel.
- All points on the ray are all scales
- So points on ray are: $[x_e, y_e, z_e]^t = \alpha[x_n, y_n, -1]^t$

Basic mathematical model



- So $[x_e, y_e, z_e]^t = -z_e[x_n, y_n, -1]^t$
- So $x_n = -\frac{x_e}{z_e}, \; y_n = -\frac{y_e}{z_e}$

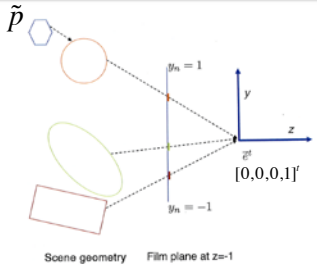
Projection matrix



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

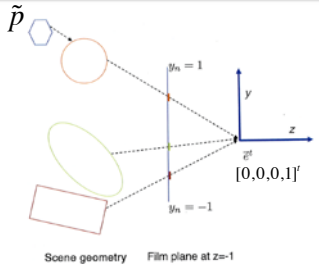
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ - & - & - & - \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix} = \begin{bmatrix} x_n w_n \\ y_n w_n \\ - \\ w_n \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \\ - \\ w_c \end{bmatrix}$$

In matrix form



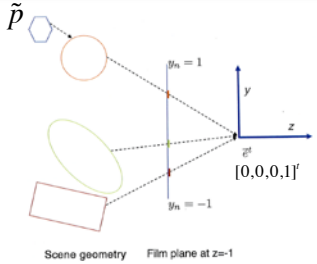
- The raw output of the matrix multiply, $[x_c, y_c, -, w_c]^t$ are called the clip coordinates of $\tilde{p}$.
- $w_n = w_c$ is a new variable called the w-coordinate.
  - In such clip coordinates, the fourth entry of the coordinate 4-vector is not necessarily a zero or a one.

Divide by *w*



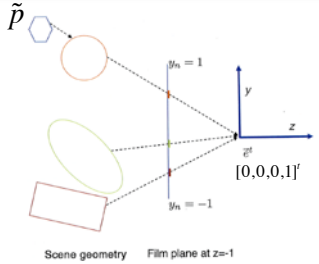
- We say that $x_n w_n = x_c$ and $y_n w_n = y_c$. If we want to extract x_n alone, we must perform the division $x_n = \frac{x_n w_n}{w_n}$
- This recovers our camera model $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ - & - & - & - \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} = \begin{bmatrix} x_n w_n \\ y_n w_n \\ - \\ w_n \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \\ - \\ w_c \end{bmatrix}$

Divide by *w*



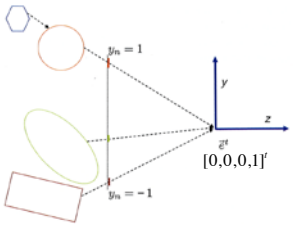
- Our output coordinates, with subscripts 'n', are called *normalized device coordinates (NDC)* because they address points on the image in abstract units without specific reference to numbers of pixels.

Divide by *w*



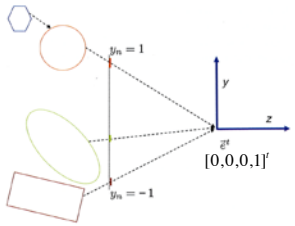
- We keep all of the image data in the *canonical square*, $-1 \le x_n \le +1, -1 \le y_n \le +1,$ and ultimately map this onto a window on the screen.
  - Data outside of this square does not be recorded or displayed.
  - This is exactly the model we used to describe 2D OpenGL

Scales



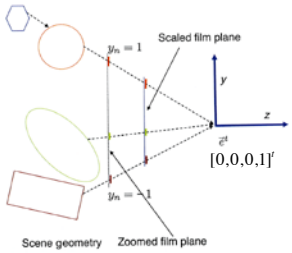
- By changing the entries in the projection matrix, we can slightly alter geometry of the camera transformation.
- We could push the film plane out to $z_e = n$, where n is some negative number (zoom lens)

Scales



- So points on ray are: $[x_e, y_e, z_e]^t = \alpha[x_n, y_n, z_n]^t$
- So $[x_e, y_e, z_e]^t = \frac{z_e}{n}[x_n, y_n, z_n]^t$
- So $x_n = \frac{x_e n}{z_e}$, $y_n = \frac{y_e n}{z_e}$

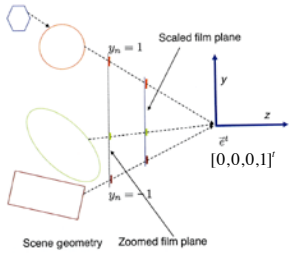
In matrix form



- In matrix form, this becomes:

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

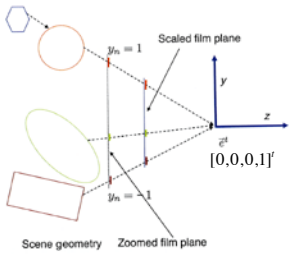
In matrix form



- Note this matrix is the same as

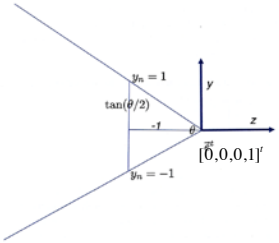
$$\begin{bmatrix} -n & 0 & 0 & 0 \\ 0 & -n & 0 & 0 \\ - & - & - & - \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ - & - & - & - \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

In matrix form



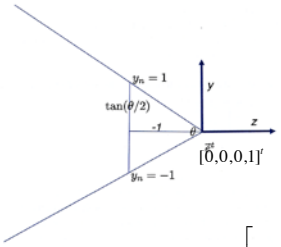
- This has the same effect as starting with our original camera, scaling by $-n$, and cropping to the canonical square.

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 θ degrees, then we can set $-n = \frac{1}{\tan\left(\frac{\theta}{2}\right)}$ giving us

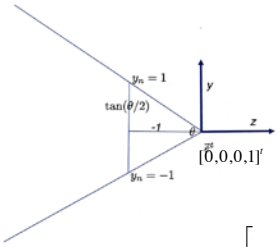
fovY



- Verify that any point who's ray from the origin forms a vertical angle of $\theta/2$ with the negative z axis maps to the boundary of the canonical square

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

fovY



- The point with eye coordinates: $[0,\tan\left(\frac{\theta}{2}\right),-1,1]^t$ maps to normalized device coordinates $[0,1]^t$

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

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). $a = \frac{(\text{width px})}{(\text{height px})}$

Dealing with aspect ratio

- We can then set our projection matrix to be:

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

- So when the window is wide, we will keep more horizontal FOV, and when the window is tall, we will keep less horizontal FOV.

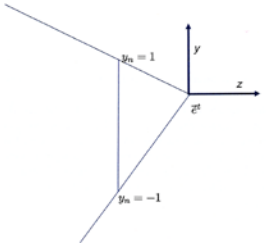
Dealing with aspect ratio

- As an alternative, we could have an fovMin, and when the window is tall, we would need to calculate an appropriate larger fovY and then build the matrix.

FOV issues

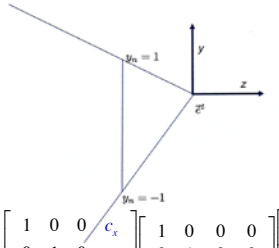
- To be a "window" onto the world, the FOV should match the angular extents of the window in the viewers field.
- This might give a too limited view onto the world.
- So we can increase it to see more.
- But this might give a somewhat unnatural look.

Shifts



- Sometimes, we wish to crop the image non-centrally.
- This can be modeled as translating the normalized device coordinates (NDC)'s and then cropping centrally.

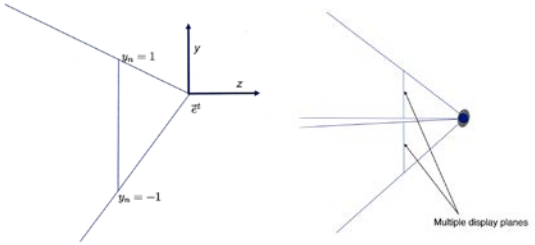
Shifts



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

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

Shifts

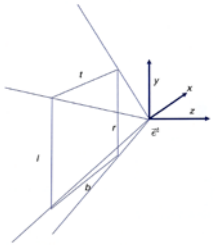


- Useful for tiled displays, stereo viewing, and certain kinds of images.

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.

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



Context

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