Schedule

- 4/5 (Tue) Lecture – 3D rotation (Chap 6~8) / Interpolation (Chap 9)
- 4/6 (Wed) Open Lab
- 4/7 (Thur) Lecture – Projection& Depth (Chap10,11)

- 4/12 (Tue) TA's Special Session for OpenGL (RE: **HW#3 out**!)
- 4/13 <Election Day> No lab session ----------- **HW#2 DUE**
- 4/14 (Thur) Lecture – From Vertex to Pixel (Chap.12)

- 4/19 (Tue) Lecture – Modeling (Chap.22) or Varying variable (Chap 13)
- 4/20~26 (Midterm Week)
  4/26 (Tuesday) 4~7 PM **Midterm Exam** @ E3-1 #1501

- 4/27 (Wed) Open Lab
- 4/28 (Thur) Lecture – Lighting (Chap. 13) ---------- **HW#3 DUE**

Questions from last lecture

$$R_\alpha := (R_1 R_0^{-1})^\alpha R_0$$

- About cn

$$\left(cn\left(\begin{bmatrix} \cos\left(\frac{\theta_1}{2}\right) \\ \sin\left(\frac{\theta_1}{2}\right)\hat{\mathbf{k}}_1 \end{bmatrix} \begin{bmatrix} \cos\left(\frac{\theta_0}{2}\right) \\ \sin\left(\frac{\theta_0}{2}\right)\hat{\mathbf{k}}_0 \end{bmatrix}^{-1} \right) \right)^\alpha \begin{bmatrix} \cos\left(\frac{\theta_0}{2}\right) \\ \sin\left(\frac{\theta_0}{2}\right)\hat{\mathbf{k}}_0 \end{bmatrix}$$

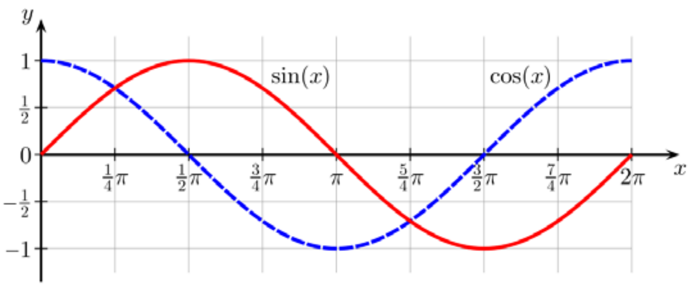
- Note that a rotation of $-\theta$ degrees about the axis $-\hat{\mathbf{k}}$ gives us the same quaternion.
- A rotation of $\theta + 4\pi$ degrees about an axis $\hat{\mathbf{k}}$ also gives us the same quaternion
- A rotation of $\theta + 2\pi$ degrees about an axis $\hat{\mathbf{k}}$, which in fact is the same rotation, gives us the negated quaternion
- So antipodes represent the same rotation transformation

Questions from last lecture

$$R_\alpha := (R_1 R_0^{-1})^\alpha R_0$$

- If the transition quaternion $\begin{bmatrix}\cos\left(\frac{\theta}{2}\right)\\ \sin\left(\frac{\theta}{2}\right)\hat{\mathbf{k}}\end{bmatrix}$ presents a θ of more than π (180) degrees, in particular, if $\cos\left(\frac{\theta}{2}\right) < 0$ then $\theta \in [\pi ... 2\pi]$, $\alpha\theta$ would go more than 180 degrees which we don't want during interpolation
- In this case, suppose we had swapped to the antipode before calling power. Then $\cos\left(\frac{\theta}{2}\right) > 0$, we get $\theta/2 \in [-\pi/2 ... \pi/2]$. And thus $\theta \in [-\pi ... \pi]$
- So when we interpolate, before calling the power operator, we first check the sign of the first coordinate, and conditionally negate the quaternion.
- We call this the conditional negation operator *cn*

$$\left(cn\left(\begin{bmatrix}\cos\left(\frac{\theta_1}{2}\right)\\ \sin\left(\frac{\theta_1}{2}\right)\hat{\mathbf{k}}_1\end{bmatrix} \begin{bmatrix}\cos\left(\frac{\theta_0}{2}\right)\\ \sin\left(\frac{\theta_0}{2}\right)\hat{\mathbf{k}}_0\end{bmatrix}^{-1} \right) \right)^\alpha \begin{bmatrix}\cos\left(\frac{\theta_0}{2}\right)\\ \sin\left(\frac{\theta_0}{2}\right)\hat{\mathbf{k}}_0\end{bmatrix}$$



Gimbal Lock Problem

Extracting Euler angles

$$F = \begin{pmatrix} f_{00} & f_{01} & f_{02} \\ f_{10} & f_{11} & f_{12} \\ f_{20} & f_{21} & f_{22} \end{pmatrix} = R_z(r)R_x(p)R_y(h) = E(h,p,r)$$

With $\cos(a) = C_a$, $\sin(a) = S_a$

$$= \begin{pmatrix} C_rC_h - S_rS_pS_h & -S_rC_p & C_rS_h + S_rS_pC_h \\ S_rC_h + C_rS_pS_h & C_rC_p & S_rS_h - C_rS_pC_h \\ -C_pS_h & S_p & C_pC_h \end{pmatrix}$$

$$\frac{f_{01}}{f_{11}} = \frac{-\sin r}{\cos r} = -\tan r$$

$$\frac{f_{20}}{f_{22}} = \frac{-\sin h}{\cos h} = -\tan h$$

$$f_{21} = \sin p$$

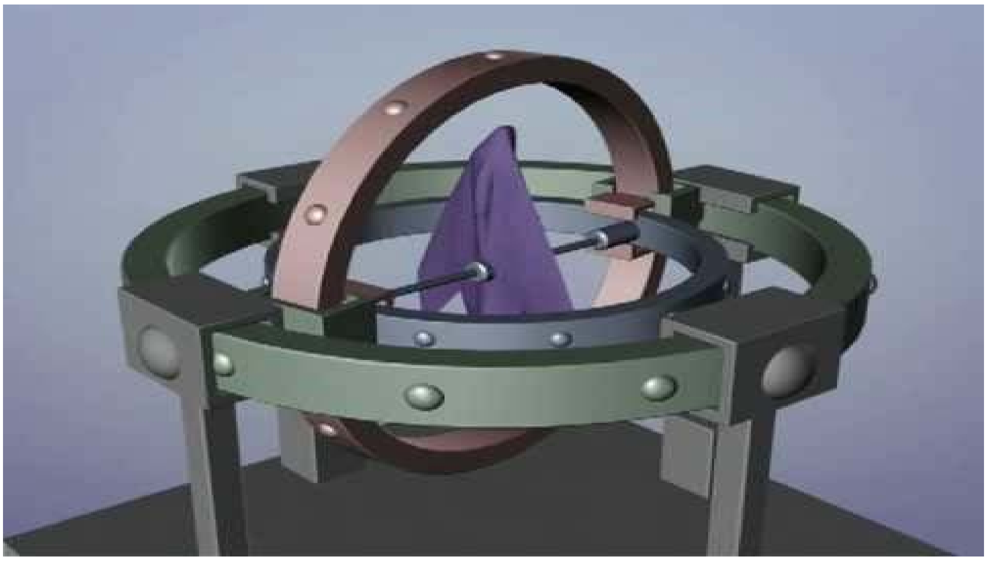
$r = \text{atan2}(-f_{01}, f_{11})$

$h = \text{atan2}(-f_{20}, f_{22})$

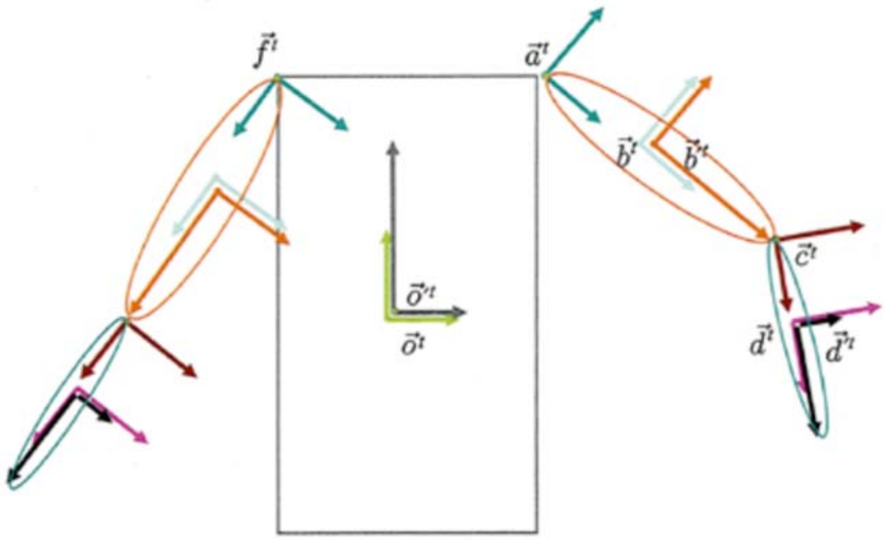
$p = \arcsin(f_{21})$

Problem: what if $\cos p = 0$?

Some references on quaternion (paper) and gimbal lock (demo) are posted on KLMS



O’ ?



$$\vec{\mathbf{o}}^t = \vec{\mathbf{w}}^t O$$
$$\vec{\mathbf{o}}'^t = \vec{\mathbf{o}}^t O'$$
$$\vec{\mathbf{a}}^t = \vec{\mathbf{o}}^t A$$
$$\vec{\mathbf{b}}^t = \vec{\mathbf{a}}^t B$$
$$\vec{\mathbf{b}}'^t = \vec{\mathbf{b}}^t B'$$
$$\vec{\mathbf{c}}^t = \vec{\mathbf{b}}^t C$$
$$\vec{\mathbf{d}}^t = \vec{\mathbf{c}}^t D$$
$$\vec{\mathbf{d}}'^t = \vec{\mathbf{d}}^t D'$$
$$\vec{\mathbf{f}}^t = \vec{\mathbf{o}}^t F$$

Projection & Depth

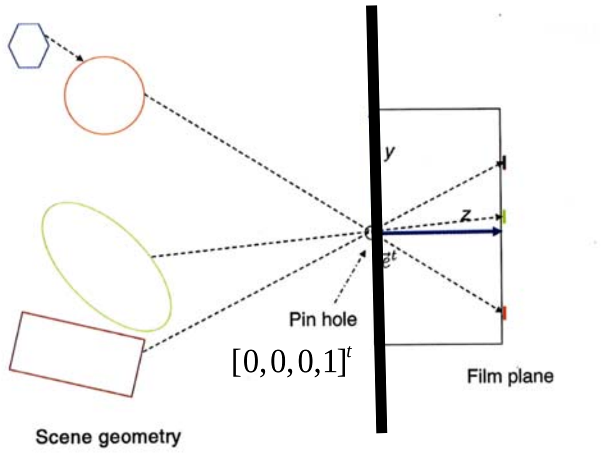
Chapter 10 & 11

Slide from Prof. MH Kim

Camera transforms

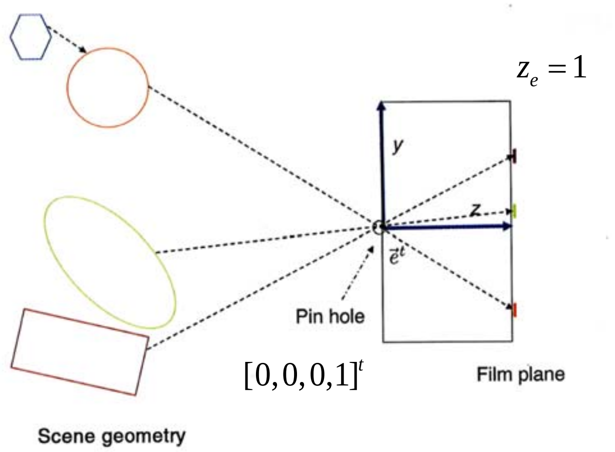
- Until now we have considered all of our geometry in a 3D space
- Ultimately everything ended up in eye coordinates with coordinates $[x_e, y_e, z_e, 1]^t$
- We said that the camera is placed at the origin of the eye frame $\vec{\mathbf{e}}^t$, and that it is looking down the eye's negative z-axis.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

Pinhole camera model



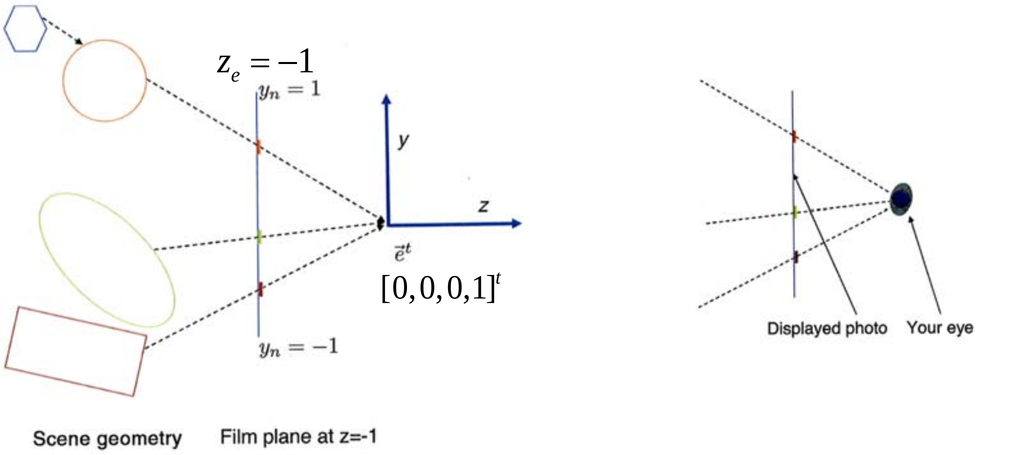
- As light travels towards the film plane, most is blocked by an opaque surface placed at the $z_e = 0$ plane.
- But we place a very small hole in the center of the surface, at the point with eye coordinates $[0,0,0,1]^t$

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, $z_e = 1$

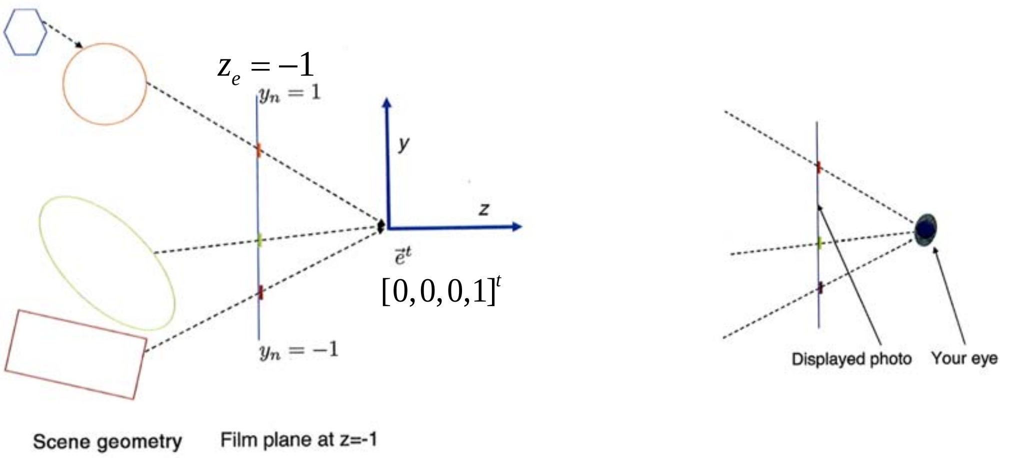
Pinhole camera model



- A physical camera needs a finite aperture and a lens, but we will ignore this.

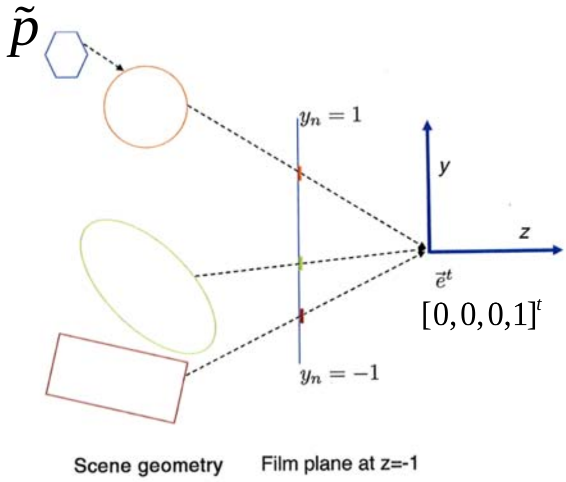
- To avoid the image flip, we can mathematically model this with the film plane in front of the pinhole, say at the $z_e = -1$

Pinhole camera model



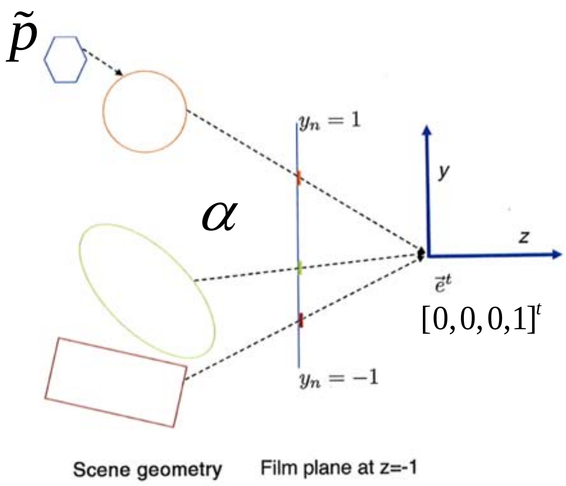
- If we hold up the photograph at the $z_e = -1$ plane, and observe it with our own eye, placed at the origin, it will look to us just like the origin scene would have.

Basic mathematical model



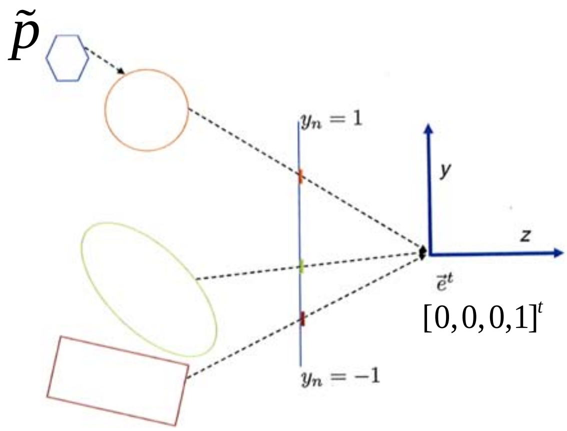
- Let us use normalized coordinates $[x_n, y_n]^t$ to specify points on our film plane.
  - For now, let them match eye coordinates on this film plane.
- Where does the ray from $\tilde{p}$ to the origin hits the film plane?

Basic mathematical model

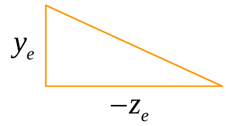


- All points on the ray hit the same pixel.
- All points on the ray are all scales
- So points on ray are: $[x_e, y_e, z_e]^t = \alpha[x_n, y_n, -1]^t$

Basic mathematical model



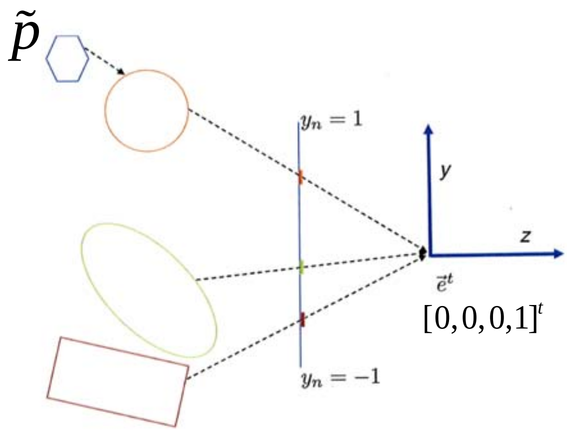
Scene geometry Film plane at z=-1



- So $[x_e, y_e, z_e]^t = -z_e [x_n, y_n, -1]^t$
- So

$$x_n = -\frac{x_e}{z_e}, \quad y_n = -\frac{y_e}{z_e}$$

Projection matrix

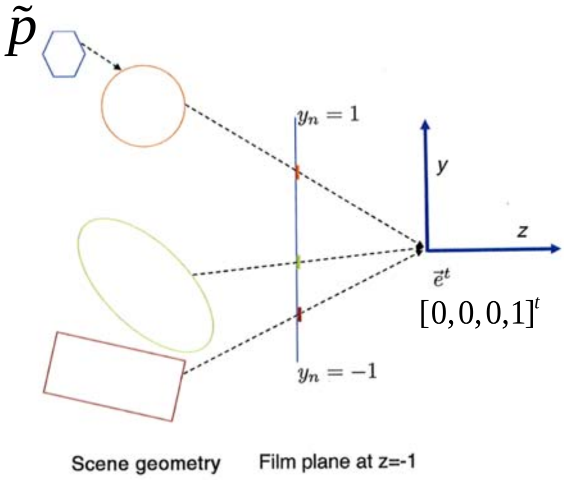


Scene geometry Film plane at z=-1

- We can model this expression as a matrix operation.

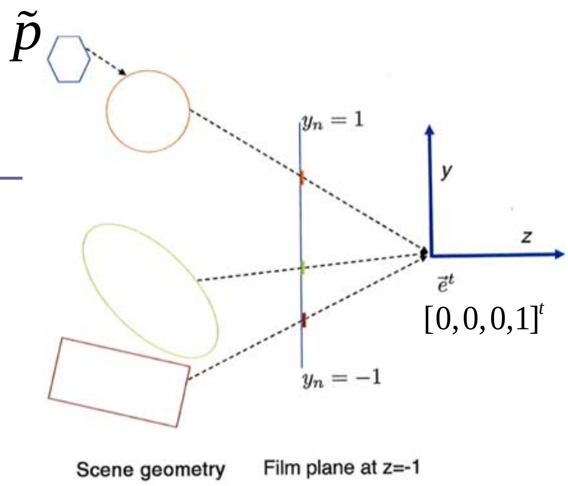
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ - & - & - & - \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix} = \begin{bmatrix} x_n w_n \\ y_n w_n \\ - \\ w_n \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \\ - \\ w_c \end{bmatrix}$$

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.

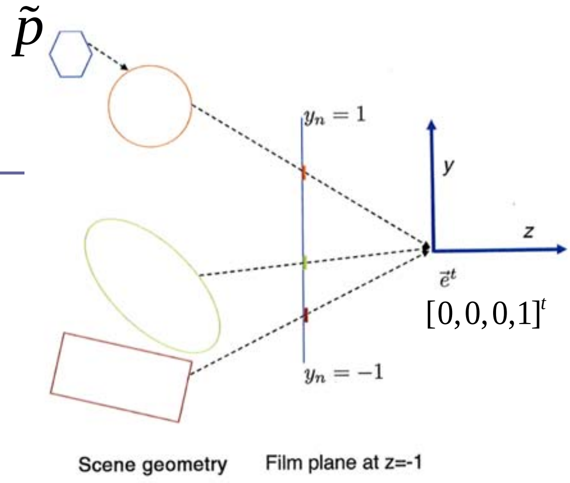
Divide by *w*



- 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

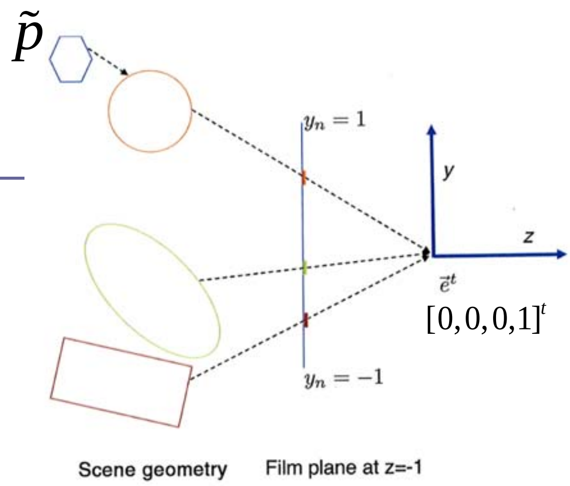
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ - & - & - & - \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix} = \begin{bmatrix} x_n w_n \\ y_n w_n \\ - \\ w_n \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \\ - \\ w_c \end{bmatrix}$$

Divide by *w*



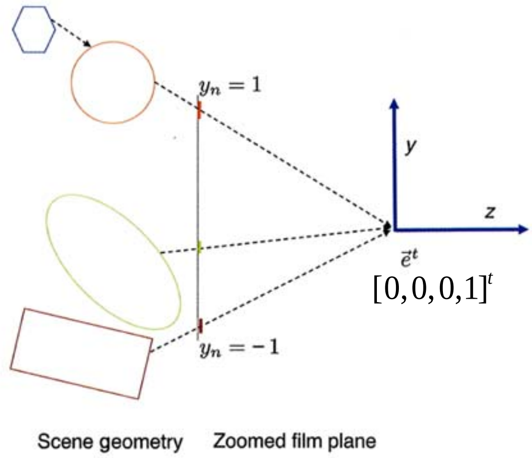
- Our output coordinates, with subscripts 'n', are called *normalized device coordinates (NDC)* because they address points on the image in abstract units without specific reference to numbers of pixels.

Divide by *w*



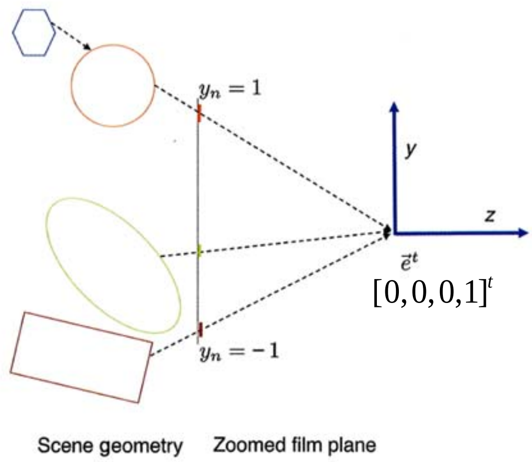
- We keep all of the image data in the *canonical square,* $-1 \le x_n \le +1, -1 \le y_n \le +1$, and ultimately map this onto a window on the screen.
  - Data outside of this square does not be recorded or displayed.
  - This is exactly the model we used to describe 2D OpenGL

Scales



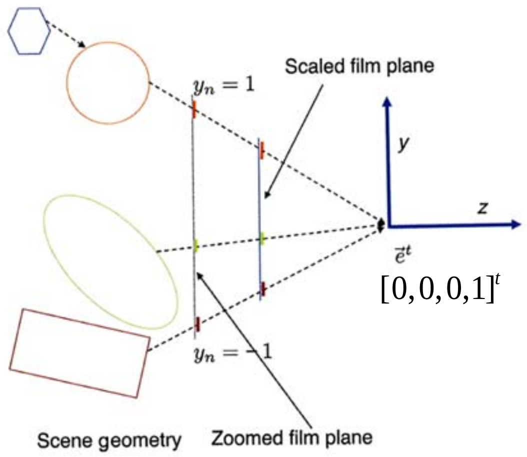
- By changing the entries in the projection matrix,
  we can slightly alter geometry of the camera transformation.
- We could push the film plane out to $z_e = n$,
  where n is some negative number (zoom lens)

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
$$x_n = \frac{x_e n}{z_e}, \quad y_n = \frac{y_e n}{z_e}$$

In matrix form

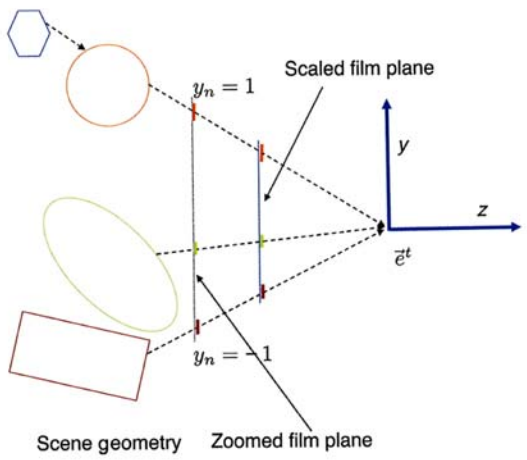


- In matrix form, this becomes:

(supposing *n* is some negative number)

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

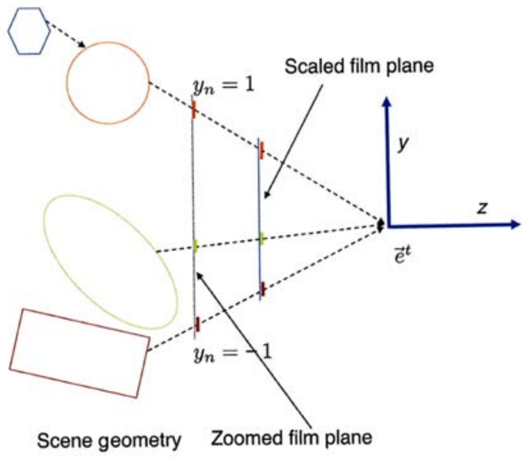
In matrix form



- Note this matrix is the same as

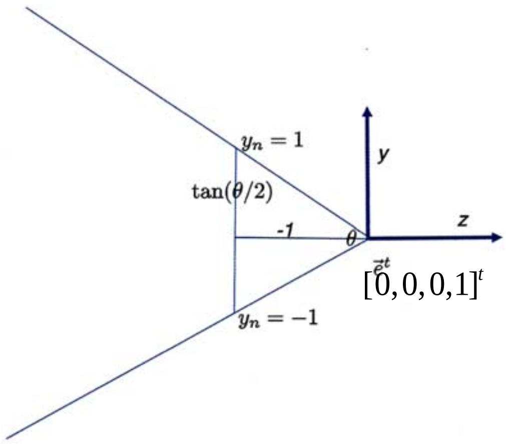
$$\begin{bmatrix} -n & 0 & 0 & 0 \\ 0 & -n & 0 & 0 \\ - & - & - & - \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ - & - & - & - \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

In matrix form



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

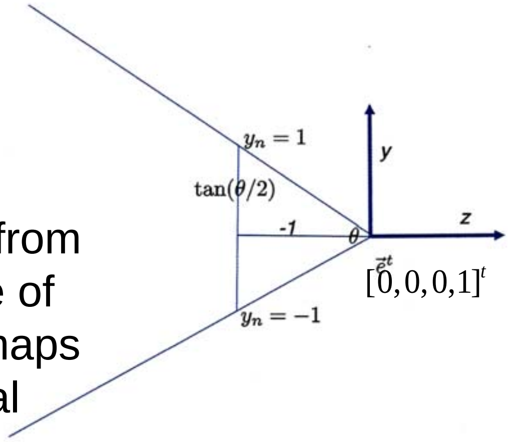
fovY



- 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 $-n = \dfrac{1}{\tan\left(\dfrac{\theta}{2}\right)}$ 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



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

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

  normalized device coordinates

  $[0,1]^t$

Dealing with aspect ratio

- Suppose the window is wider than its height. 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:

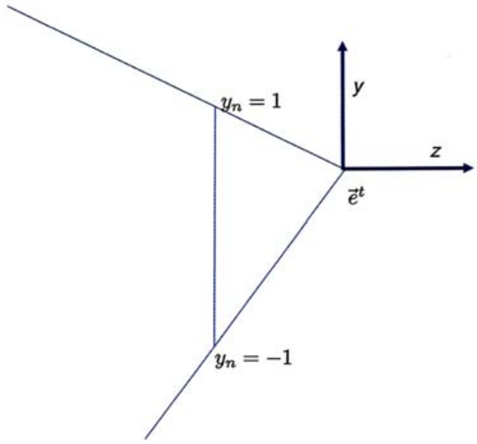
$$\begin{bmatrix} \dfrac{1}{\alpha \tan\left(\dfrac{\theta}{2}\right)} & 0 & 0 & 0 \\ 0 & \dfrac{1}{\tan\left(\dfrac{\theta}{2}\right)} & 0 & 0 \\ - & - & - & - \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

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

FOV issues

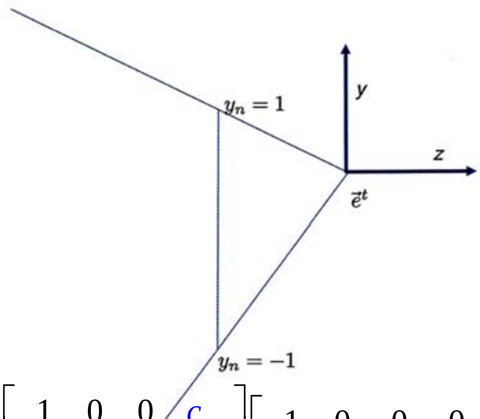
- To be a “window” onto the world,
  the FOV should match the angular extents of the window in the viewers field.
- This might give a too limited view onto the world.
- So we can increase it to see more.
- But this might give a somewhat unnatural look.

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.

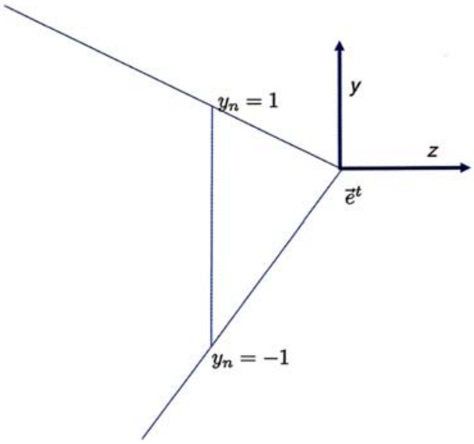
Shifts

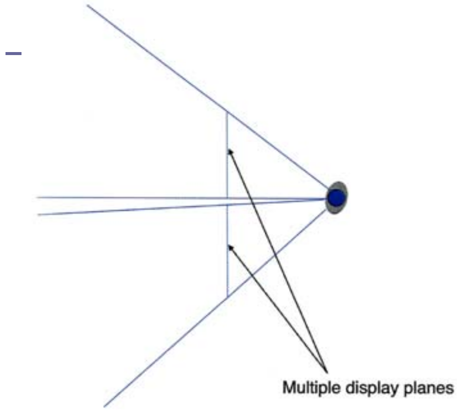


$$\begin{bmatrix} x_n w_n \\ y_n w_n \\ - \\ w_n \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & c_x \\ 0 & 1 & 0 & c_y \\ - & - & - & - \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ - & - & - & - \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix}$$

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

Shifts





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

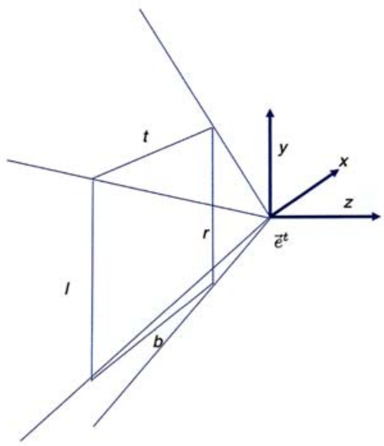
Frustum

- 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 $l, r, t, b.$

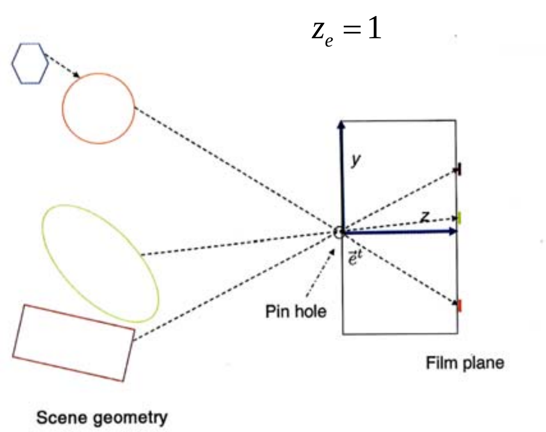
$$\begin{bmatrix} -\frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & -\frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ - & - & - & - \\ 0 & 0 & -1 & 0 \end{bmatrix}$$



Context

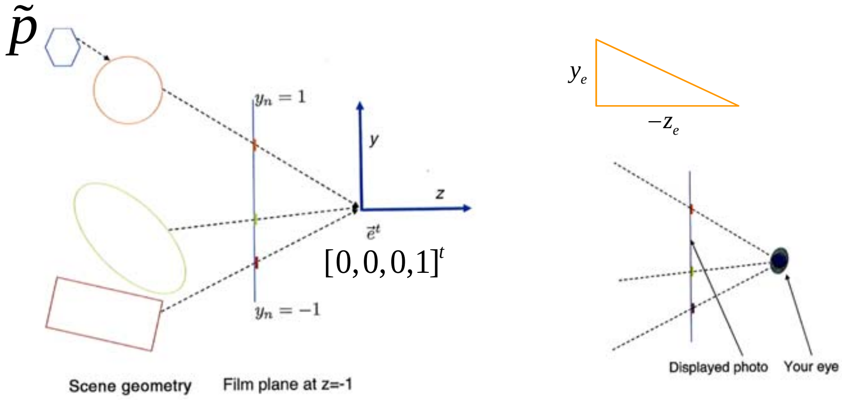
- Projection could be applied to every point in the scene.
- In CG, we will apply it to the vertices to position a triangle on the screen.
- The rest of the triangle will then get filled in on the screen as we shall see.

Summary:
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, $z_e = 1$

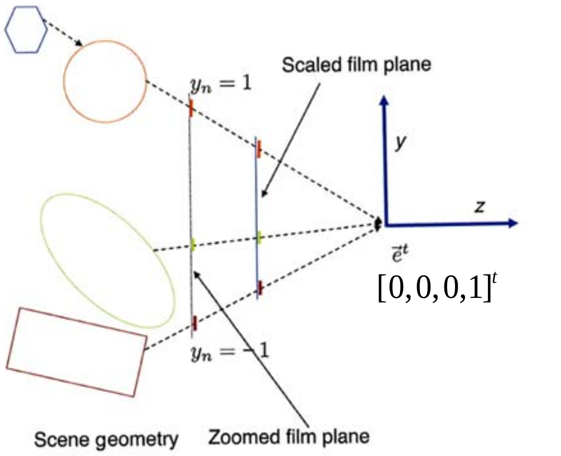
Summary: Normalized device coordinates



- Canonical square space:

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

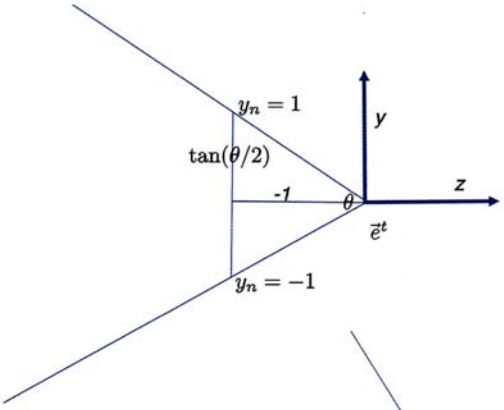
Summary: Scale factor *n*



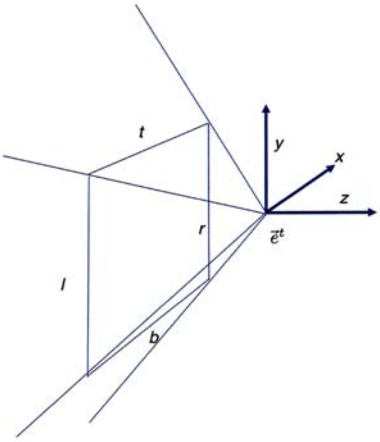
- Controlling aspect ratio of film space

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

Summary:
Frustum: Eye coor. → NDC

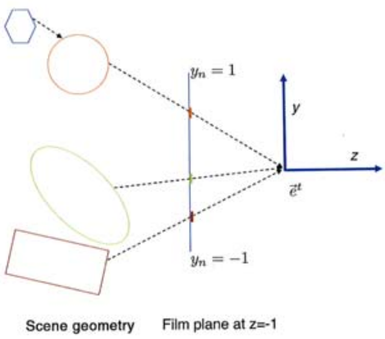


$$\begin{bmatrix} \frac{1}{\alpha \tan\left(\frac{\theta}{2}\right)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\left(\frac{\theta}{2}\right)} & 0 & 0 \\ - & - & - & - \\ 0 & 0 & -1 & 0 \end{bmatrix}$$



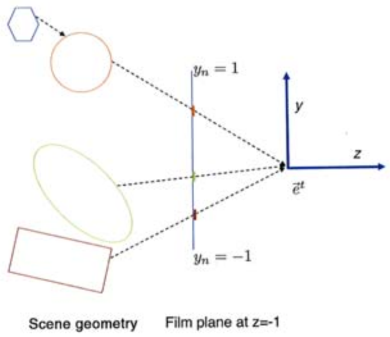
$$\begin{bmatrix} -\frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & -\frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ - & - & - & - \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Visibility



- In the real world, opaque objects block light.
- We need to model this computationally.
- One idea is to render back to front and use overwriting
  - This will have problem with visibility cycles.

Visibility



- We could explicitly store everything hit along a ray and then compute the closest.
  - Make sense in a ray tracing setting, where we are working one pixel per ray at time, but not for OpenGL, where we are working one triangle at a time.

Z-buffer

- We will use z-buffer
- Triangle are drawn in ***any*** order
- Each pixel in frame buffer stores ‘depth’ value of closest geometry observed so far.
- When a new triangle tries to set the color of a pixel, we first compare its depth to the value stored in the z-buffer.
- Only if the observed point in this triangle is closer, we overwrite the color and depth values of this pixel.
- This is done per-pixel, so there is no cycle problems.
- There are optimizations, where z-testing is done, <u>before the fragment shading is done.</u>

Other uses of visibility calculations

- Visibility to a light source is useful for shadows.
  - We will talk about shadow mapping later.
  - We will do shadow calculations in a ray tracer.
- Visibility computation can also be used to speed up the rendering process.
  - If we know that some object is occluded from the cam era, then we don't have to render the object in the first place.
  - We can use a conservative test.

Basic mathematical model

- For every point, we define its $[x_n, y_n, z_n]^t$ coordinates, using the following matrix expression:

$$\begin{bmatrix} x_n w_n \\ y_n w_n \\ z_n w_n \\ w_n \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix} = \begin{bmatrix} s_x & 0 & -c_x & 0 \\ 0 & s_y & -c_y & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix}$$

- We now also have the value $z_n = \frac{-1}{z_e}$
- Our plan is to use this z_n value to do depth comparison in our z-buffer.

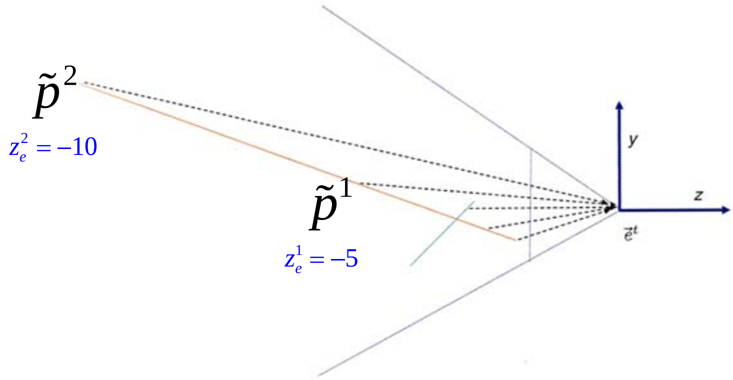
Correct ordering

- Given two points $\tilde{p}^1$ and $\tilde{p}^2$ with eye coordinates $[x_e^1, y_e^1, z_e^1, 1]^t$ and $[x_e^2, y_e^2, z_e^2, 1]^t$.
- Suppose that they both are in front of the eye, i.e., $z_e^1 < 0$ and $z_e^2 < 0$.
- And suppose that $\tilde{p}^1$ is closer to the eye than $\tilde{p}^2$, that is $z_e^2 < z_e^1$
- Then

$$-\frac{1}{z_e^2} < -\frac{1}{z_e^1},$$

meaning

$$z_e^2 < z_e^1$$

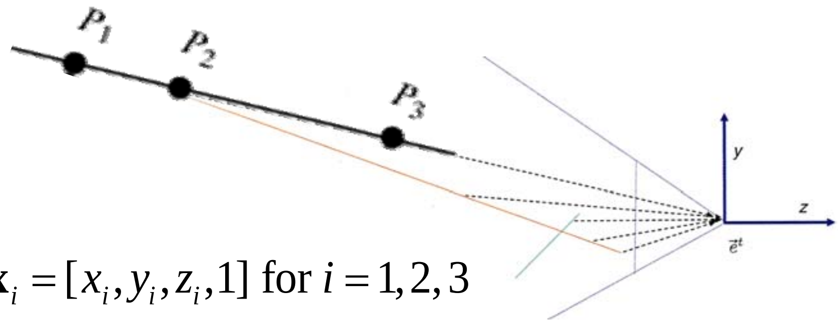


Projective transform

- We can now think of the process of taking points (given by eye coordinates) to points (given by normalized device coordinates) as an honest-to-goodness 3D geometric transformation.
- This kind of transformation is generally neither linear nor affine, but is something called a *3D projective transformation.*
- Projective transformation preserve co-linearity and co-planarity of points.

Co-linearity of points

- If three or more points are on a single line, the transformed points will also be on some single line.

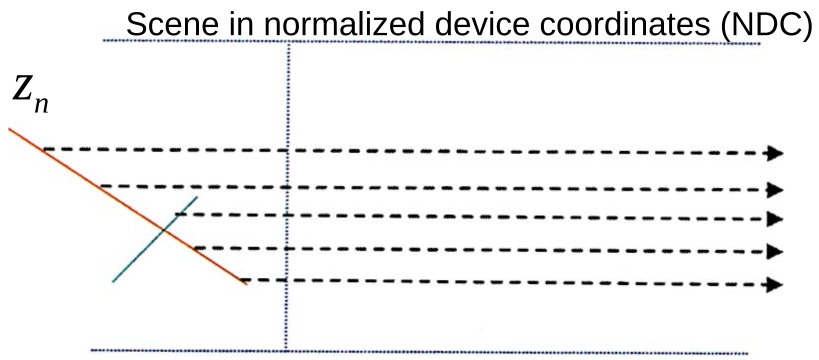


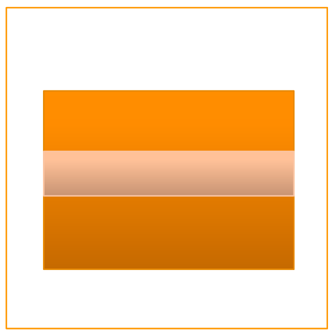
-
- Three points $\mathbf{x}_i = [x_i, y_i, z_i, 1]$ for $i = 1, 2, 3$

$$x_2 - x_1 : y_2 - y_1 : z_2 - z_1 = x_3 - x_1 : y_3 - y_1 : z_3 - z_1$$

$$\left|(\mathbf{p}_2 - \mathbf{p}_1) \times (\mathbf{p}_1 - \mathbf{p}_3)\right| = 0$$

Co-planarity of points

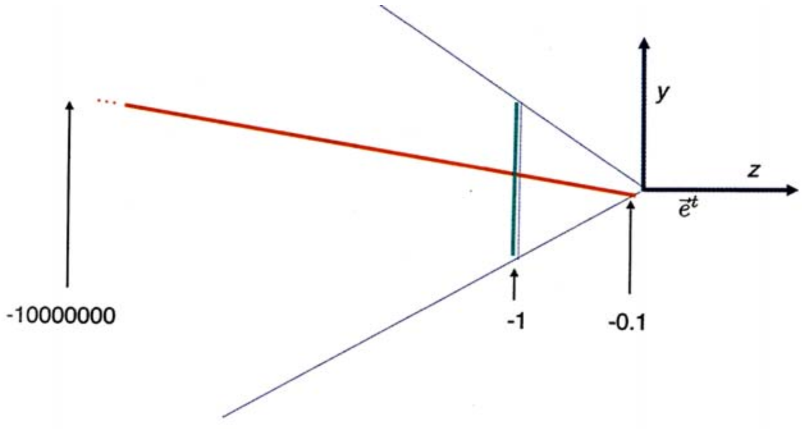




- Note that distances are not preserved by a projective transform.
- Evenly spaced pixel on the film do not correspond to evenly spaced points on the geometry in eye space.
- Meanwhile, such *evenly spaced pixels* correspond with evenly spaced points in *normalized device coordinates.*

z_n interpolation is right

- Preservation of coplanarity: for points on a fixed triangle, we will have $z_n = ax_n + by_n + c$, for some fixed a, b and c
- Thus, the correct z_n value for a point can be computed using linear interpolation over the 2D image domain as long as we know its value at the three vertices of the triangle.



Solution: near/far

- Solution: replacing the third row of the matrix with more general row $\begin{bmatrix} 0 & 0 & \alpha & \beta \end{bmatrix}$

$$\Rightarrow \begin{bmatrix} x_n w_n \\ y_n w_n \\ z_n w_n \\ w_n \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix} = \begin{bmatrix} s_x & 0 & -c_x & 0 \\ 0 & s_y & -c_y & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix}$$

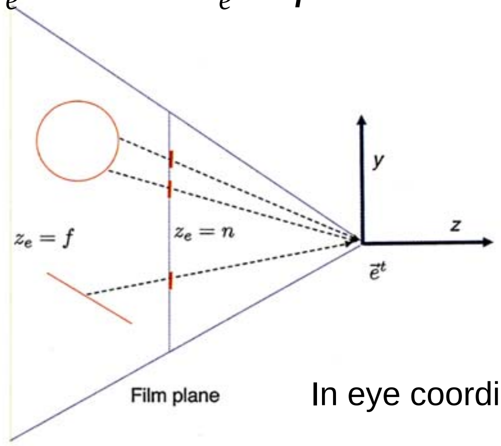
- It is easy to verify that if the value α and β are both positive, then the z-ordering of points (assuming they all have negative z_e values) is preserved under the projective transform.

Solution: near/far

- To set α and β, we first select depth value n and f called the *near* and *far* values (both negative), such that our main region of interest in the scene is sandwiched between $z_e = n$ and $z_e = f$
- Given these selections we set

$$\alpha = \frac{f+n}{f-n}$$

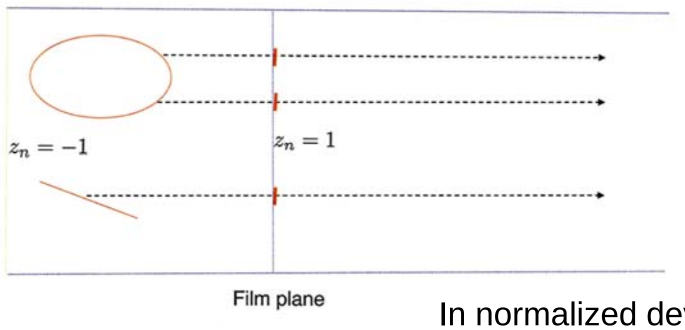
$$\beta = -\frac{2fn}{f-n}$$



In eye coordinates

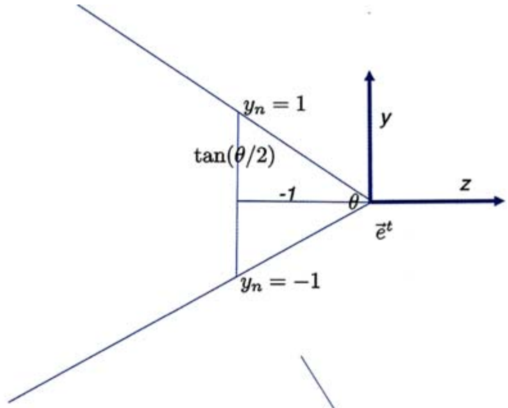
Solution: near/far

- We can verify now that any point with $z_e = f$ maps to a point with $z_n = -1$ and that a point with $z_e = n$ maps to a point with $z_n = 1$
- Any geometry not in this [near…far] range is clip ped away by OpenGL and ignored.

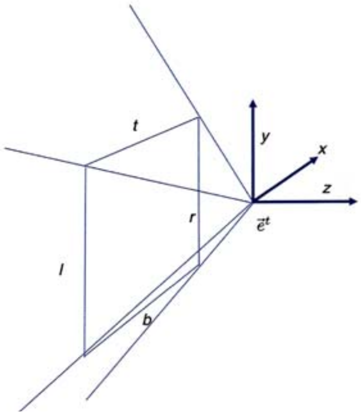


In normalized device coordinates

Proj. Trans.: Eye coor. → NDC



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

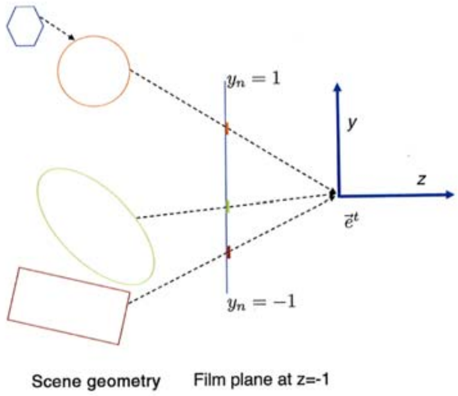


$$\begin{bmatrix} -\frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & -\frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Codes

- In OpenGL, use the z-buffer is turned on with a call to glEnable(GL_DEPTH_TEST).
- We also need a call to glDepthFunc(GL_GREATER), since we are using a right handed coordinate system wh ere ‘more-negative’ is ‘farther from the eye’.
- In real life, you may see other conventions (for how to interpret n and f, some of the signs of the matrix, and the handedness of the ultimate z-test.

Summary: Visibility



- We could explicitly store everything hit along a ray and then compute the closest.
  - Make sense in a ray tracing setting, where we are worki ng one pixel per ray at a time, but not for OpenGL, wher e we are working one triangle at a time.

Summary: Z-buffer

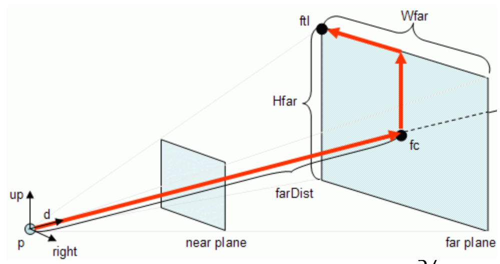
- We will use z-buffer
- Triangle are drawn in any order
- Each pixel in frame buffer stores 'depth' value of closest geometry observed so far.
- When a new triangle tries to set the color of a pix el, we first compare its depth to the value stored in the z-buffer.
- Only if the observed point in this triangle is close r, we overwrite the color and depth values of this pixel.

Summary: Proj. Trans.: Eye coor. → NDC

- Camera projection transformation

$$
\begin{bmatrix} x_n w_n \\ y_n w_n \\ z_n w_n \\ w_n \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix} = \begin{bmatrix} -\dfrac{2n}{r-l} & 0 & \dfrac{r+l}{r-l} & 0 \\ 0 & -\dfrac{2n}{t-b} & \dfrac{t+b}{t-b} & 0 \\ 0 & 0 & \dfrac{f+n}{f-n} & -\dfrac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix}
$$

$$
M = \begin{bmatrix} \dfrac{1}{\alpha \tan\left(\frac{\theta}{2}\right)} & 0 & 0 & 0 \\ 0 & \dfrac{1}{\tan\left(\frac{\theta}{2}\right)} & 0 & 0 \\ 0 & 0 & \dfrac{f+n}{f-n} & -\dfrac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}
$$



Projection Normalization

"window space"

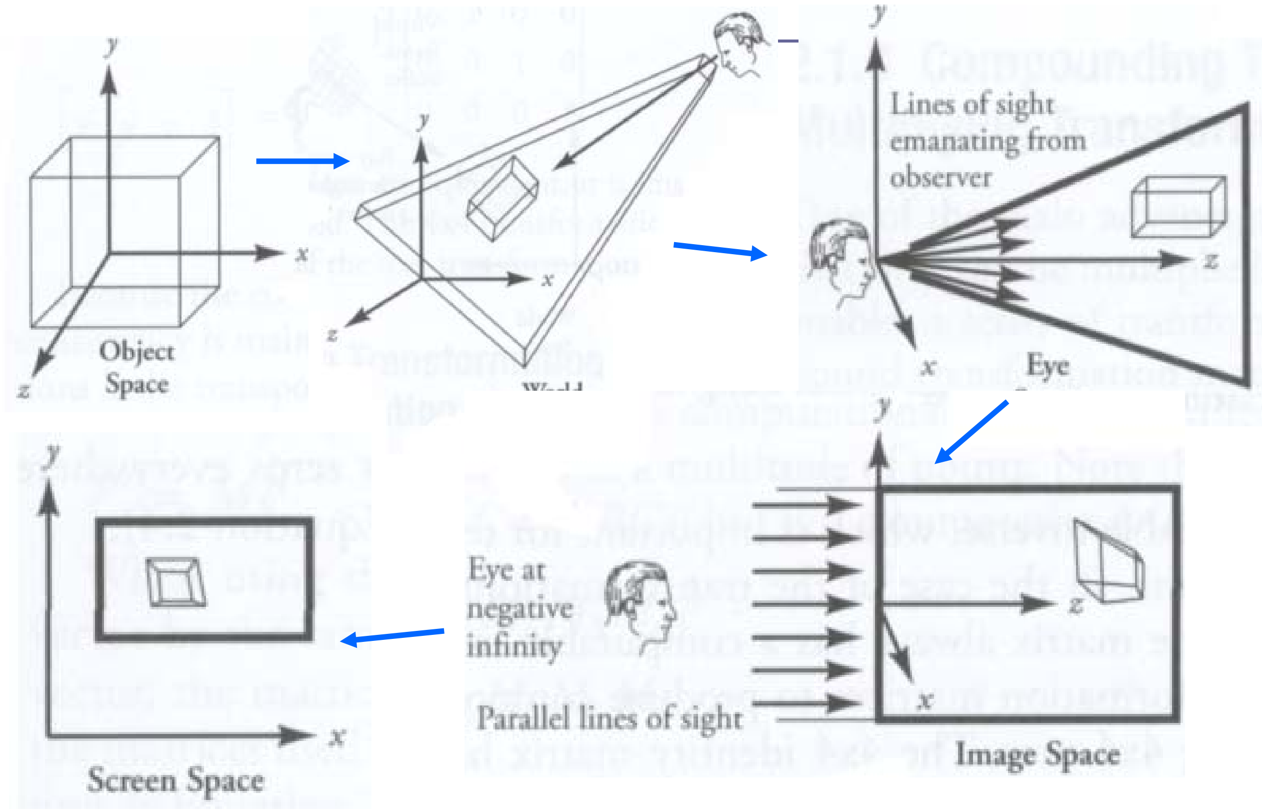
"screen space"

3D object in the camera coordinates → **perspective projection** → **2D object in the screen coordinates (normalized device coord)**

3D object in the camera coordinates → **projection normalization** → **orthographic projection** → **2D object in the screen coordinates (normalized device coord)**

3D object in the distorted coordinates (clip coordinates)

Display pipeline



Rendering pipeline (geometry stage)

vertex (x, y, z, w)

ModelView Transformation

eye coordinate

Projection Transformation

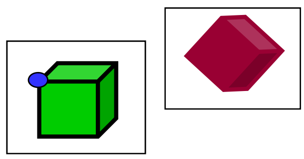
clip coordinate

Projective Division

normalized device coordinate

Viewport Transformation

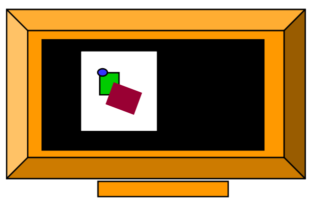
window(device) coordinate (x, y)



Model / object coordinates

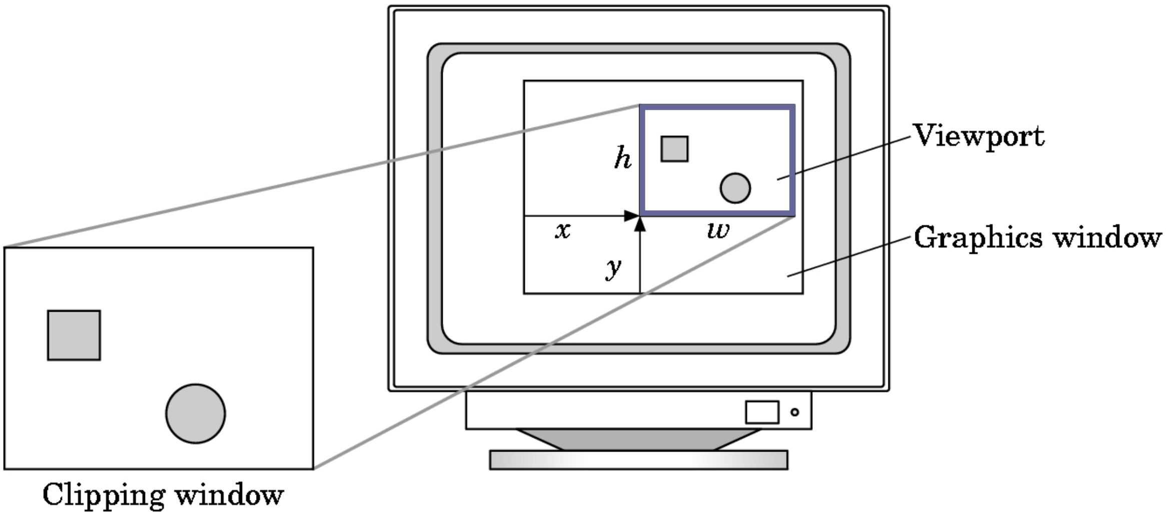
World coordinates

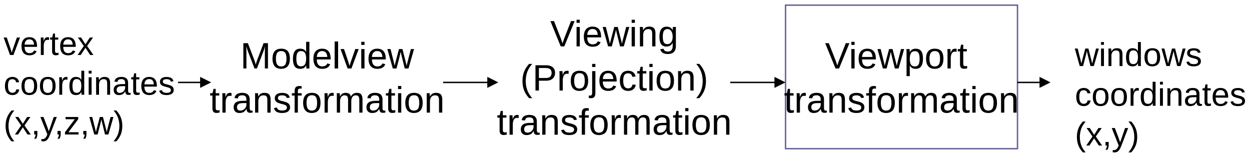
View coordinates



Display coordinates

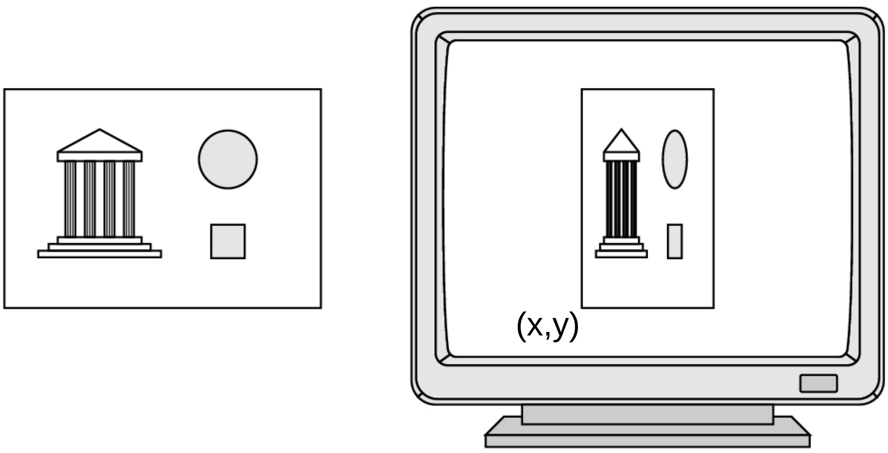
Viewport





Viewport

- Usually same as window size
- Aspect ratio = width/height




```
glViewprot(x,y,width,height)
```

Faux Plafond - Cosmic Promenade - *Mikros Image*

- Siggraph 2000

- http://www.siggraph.org/publications/video-review/SVR_2000/134/

Exercise

- Represent the following rotation using ..
    1. Matrix
    2. Euler angle / Fixed angle
    3. Angle and Axis
    4. Unit quaternion