3D Viewing, part II

CS 465 Lecture 11

Viewing, backward and forward

- So far have used the backward approach to viewing
  - start from pixel
  - ask what part of scene projects to pixel
  - explicitly construct the ray corresponding to the pixel
- Next will look at the forward approach
  - start from a point in 3D
  - compute its projection into the image
- Central tool is matrix transformations
  - combines seamlessly with coordinate transformations used to position camera and model
  - ultimate goal: single matrix operation to map any 3D point to its correct screen location.

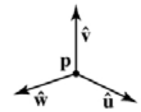
Ray generation with matrices

- We didn't use transformations in eye ray generation, but can we simplify things using them?
- Our ray generation process:
  - Step 0: build basis for image plane
  - Step 1: find (*u*,*v*) coordinates from pixel indices
  - Step 2: offset from the center of the image window to get **q**
  - Step 3: build the ray as (**p**, **q** – **p**)
- Steps 1 and 2 can be done with affine transformations
  - Step A: build a coordinate frame for the camera
  - Step B: make a 2D affine transformation to go from (*i*,*j*) to (*u*,*v*)
  - Step C: make a 3D affine transform to find **q** in camera coordinates
  - Step D: multiply it all together to get a transform that goes straight from (*i*,*j*) to **q**

Ray generation with matrices

- Step A: build a coordinate frame for the camera
  - Already did this, really
- Build ONB from image plane normal and up vector
  - Frame origin is the viewpoint
  - Axes aligned with image
- No longer need to worry about camera pose
  - rays all start at **0**
  - directions all on a plane

$$F_c = \begin{bmatrix} \hat{\mathbf{u}} & \hat{\mathbf{v}} & \hat{\mathbf{w}} & \mathbf{p} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Ray generation with matrices

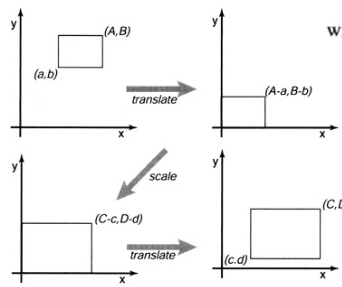
- Step B: affine transformation from (*i*,*j*) to (*u*,*v*)
  - slight change of (*u*,*v*) convention: let (*u*,*v*) be in [–1,1] x [–1,1]
- Simple to build:
  - origin goes to center of lower left pixel, which is (–1 + 1/*m*, –1 + 1/*n*) for an *m* by *n* image, so that is the translation part
  - scale by 2/*m* in *x* and 2/*n* in *y*

$$M_v = \begin{bmatrix} 2/m & 0 & 1/m-1 \\ 0 & 2/n & 1/n-1 \\ 0 & 0 & 1 \end{bmatrix}$$

  - I'll call this the ray generation viewport matrix

Windowing transforms

- This transformation is worth generalizing: take one axis-aligned rectangle or box to another
  - a useful, if mundane, piece of a transformation chain



$$\text{window} = \begin{bmatrix} 1 & 0 & c \\ 0 & 1 & d \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{C-c}{A-a} & 0 & 0 \\ 0 & \frac{D-d}{B-b} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -a \\ 0 & 1 & -b \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \frac{C-c}{A-a} & 0 & \frac{cA-Ca}{A-a} \\ 0 & \frac{D-d}{B-b} & \frac{dB-Db}{B-b} \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{window3D} = \begin{bmatrix} \frac{D-d}{A-a} & 0 & 0 & \frac{dA-Da}{A-a} \\ 0 & \frac{E-e}{B-b} & 0 & \frac{eB-Eb}{B-b} \\ 0 & 0 & \frac{F-f}{C-c} & \frac{fC-Fc}{C-c} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

[Shirley f. 6-16; eqs. 6-6 and 7-5]

Windowing transforms

- Our viewport matrix is an instance of a windowing transform
  - source: [–1/2, *m* – 1/2] x [–1/2, *n* – 1/2] = [*a*, *A*] x [*b*, *B*]
  - destination: [–1, 1] x [–1, 1] = [*c*, *C*] x [*d*, *D*]

$$\text{window} = \begin{bmatrix} \frac{C-c}{A-a} & 0 & \frac{cA-Ca}{A-a} \\ 0 & \frac{D-d}{B-b} & \frac{dB-Db}{B-b} \\ 0 & 0 & 1 \end{bmatrix}$$

  - *a* = –1/2, *A* = *m* – 1/2; *b* = –1/2, *B* = *n* – 1/2
  - *c* = –1, *C* = 1; *d* = –1, *D* = 1

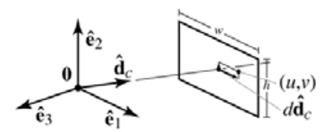
$$M_v = \begin{bmatrix} 2/m & 0 & 1/m-1 \\ 0 & 2/n & 1/n-1 \\ 0 & 0 & 1 \end{bmatrix}$$

Ray generation with matrices

- Step C: affine transform from (*u*,*v*) to **q**
- This is easy because the way we computed it before is directly a matrix operation
  - note this matrix is 4x3 (maps 2D homog. to 3D homog.)

$$M_s = \begin{bmatrix} wu/2 & 0 & d\,d_u \\ 0 & hv/2 & d\,d_v \\ 0 & 0 & d\,d_w \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{q} = d\hat{\mathbf{d}}_c + \frac{wu}{2}\hat{\mathbf{e}}_1 + \frac{hv}{2}\hat{\mathbf{e}}_2$$



Ray generation with matrices

- Step D: put it all together
- To transform pixel (i,j) to the point **q**:
  - multiply by M_v to get (u,v)
  - multiply by M_s to get $\mathbf{q}_c$ (**q** in camera frame)
  - ray is (**0**, $\mathbf{q}_c$ – **0**); multiply by ***F*** to get into world coords
- Subtracting the point **0** is the same as zeroing the w coord
  - can do in transformation world by multiplying by

$$\Pi = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

  - could call this the "point-to-vector" matrix

Ray generation with matrices

- So, for pixel (i,j), start with **x** = [i j 1]^T and:

$$\text{ray} = (\mathbf{p}, F_c \Pi M_s M_v \mathbf{x}) = (\mathbf{p}, M_{\text{raygen}} \mathbf{x})$$

  - starts at **p**; direction is computed by multiplication with a single matrix
- That's all there is to ray generation!
  - typical of transformation approach: all the work is in the setup
  - generating many rays this way is quite efficient (a few multiplications and additions, with no conditionals)
- What we did here:
  - worked in a convenient coordinate system (eye coordinates)
  - expressed several distinct steps as transformations
    - kept parameters separate
    - camera pose, camera intrinsics, image resolution don't interact directly
  - concatenated transformations together

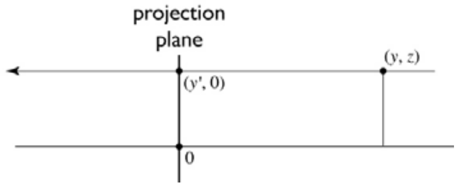
Forward viewing

- Would like to just invert the ray generation process
- Two problems (really two symptoms of same problem)
  - ray generation matrix is not invertible (it is 4 by 3)
  - ray generation produces rays, not points in scene
- Inverting the ray tracing process requires division for the perspective case

Mathematics of projection

- Always work in eye coords
  - assume eye point at **0** and plane perpendicular to z
- Orthographic case
  - a simple projection: just toss out z
- Perspective case: scale diminishes with z
  - and increases with d

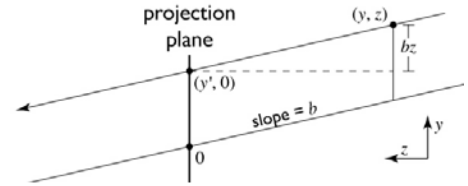
Parallel projection: orthographic



to implement orthographic, just toss out *z*:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

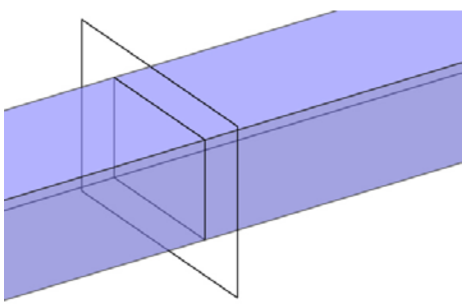
Parallel projection: oblique



to implement oblique, shear then toss out *z*:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} x + az \\ y + bz \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & a & 0 \\ 0 & 1 & b & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

View volume: orthographic



Choosing the view rectangle

- So far have just assumed we keep the *x* and *y* coords unchanged
- But they eventually have to get mapped into the image
  - As with ray generation example, do this in two steps:
    1. **M**ap desired view window to [–1, 1] x [–1, 1]
       (maps projected *x* and *y* coordinates to *canonical coordinates*)
    2. **M**ap canonical coordinates to pixel coordinates
- Window specification: top, left, bottom, right coords (*t, l, b, r*)
  - so first transform is [l,r] x [b,t] to [–1,1] x [–1,1]

$$M_o = \begin{bmatrix} \frac{2}{r-l} & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & -\frac{t+b}{t-b} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{window} = \begin{bmatrix} \frac{C-c}{A-a} & 0 & \frac{cA-Ca}{A-a} \\ 0 & \frac{D-d}{B-b} & \frac{dB-Db}{B-b} \\ 0 & 0 & 1 \end{bmatrix}$$

  - this product is known as the projection matrix for an orthographic view

Viewport matrix

- The second windowing step is to map the canonical coordinates to pixel coordinates
- Another viewport transformation, going from [–1,1] x [–1,1] to [–1/2, *m* – 1/2] x [–1/2, *n* – 1/2]

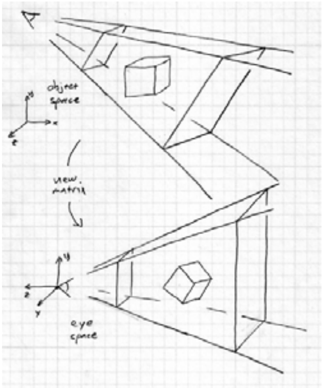
$$M_{vp} = \begin{bmatrix} \frac{m}{2} & 0 & \frac{m-1}{2} \\ 0 & \frac{n}{2} & \frac{n-1}{2} \\ 0 & 0 & 1 \end{bmatrix} \qquad \text{window} = \begin{bmatrix} \frac{C-c}{A-a} & 0 & \frac{cA-Ca}{A-a} \\ 0 & \frac{D-d}{B-b} & \frac{dB-Db}{B-b} \\ 0 & 0 & 1 \end{bmatrix}$$

- This matrix is known as the ***viewport matrix***

Viewing and modeling matrices

- We worked out all the preceding transforms starting from eye coordinates
  - before we do any of this stuff we need to transform into that space
- Transform from world (canonical) to eye space is traditionally called the ***viewing matrix***
  - it is the canonical-to-frame matrix for the camera frame
  - that is, F_c^{-1}
- Remember that geometry would originally have been in the object's local coordinates; transform into world coordinates is called the ***modeling matrix***, M_m
- Note some systems (e.g. OpenGL) combine the two into a ***modelview*** matrix and just skip world coordinates

Viewing transformation



the view matrix rewrites all coordinates in eye space

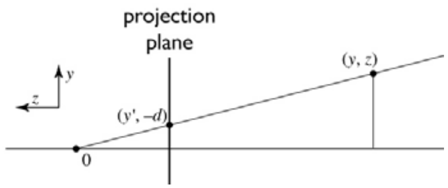
Orthographic transformation chain

- Start with coordinates in object's local coordinates
- Transform into world coords (modeling transform, M_m)
- Transform into eye coords (camera canonical-to-frame, F_c^{-1})
- Orthographic projection, M_o
- Viewport transform, M_{vp}

$$\begin{bmatrix} x_{\text{pixel}} \\ y_{\text{pixel}} \\ 1 \end{bmatrix} = M_{vp} M_o F_c^{-1} M_m \begin{bmatrix} x_{\text{object}} \\ y_{\text{object}} \\ z_{\text{object}} \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x_{\text{pixel}} \\ y_{\text{pixel}} \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{m}{2} & 0 & \frac{m-1}{2} \\ 0 & \frac{n}{2} & \frac{n-1}{2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{\hat{u}} & \mathbf{\hat{v}} & \mathbf{\hat{w}} & \mathbf{p} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} x_{\text{world}} \\ y_{\text{world}} \\ z_{\text{world}} \\ 1 \end{bmatrix}$$

Perspective projection



similar triangles:

$$\frac{y'}{d} = \frac{y}{-z}$$
$$y' = -dy/z$$

Homogeneous coordinates revisited

- Perspective requires division
  - that is not part of affine transformations
  - in affine, parallel lines stay parallel
    - therefore not vanishing point
    - therefore no rays converging on viewpoint
- “True” purpose of homogeneous coords: *projection*

Homogeneous coordinates revisited

- Introduced *w* = 1 coordinate as a placeholder

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

  - used as a convenience for unifying translation with linear
- Can also allow arbitrary *w*

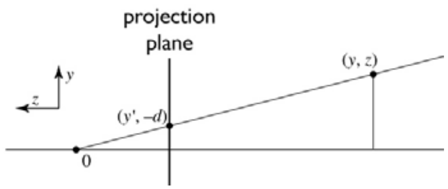
$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \sim \begin{bmatrix} wx \\ wy \\ wz \\ w \end{bmatrix}$$

Implications of *w*

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \sim \begin{bmatrix} wx \\ wy \\ wz \\ w \end{bmatrix}$$

- All scalar multiples of a 4-vector are equivalent
- When *w* is not zero, can divide by *w*
  - therefore these points represent “normal” affine points
- When *w* is zero, it’s a ***point at infinity***, a.k.a. a direction
  - this is the point where parallel lines intersect
  - can also think of it as the vanishing point
- Digression on projective space

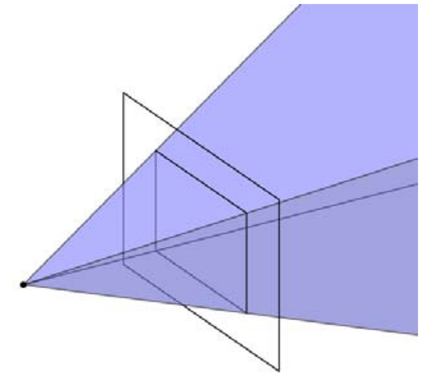
Perspective projection



to implement perspective, just move z to w:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -dx/z \\ -dy/z \\ 1 \end{bmatrix} \sim \begin{bmatrix} dx \\ dy \\ -z \end{bmatrix} = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

View volume: perspective



Choosing the view rectangle

- We can use exactly the same windowing transform as in the orthographic case to map the view window to the canonical rectangle:

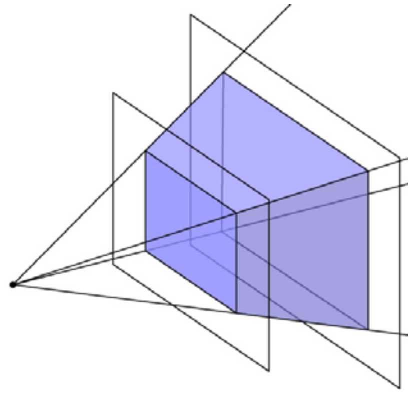
$$M_p = \begin{bmatrix} \frac{2}{r-l} & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & -\frac{t+b}{t-b} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$
$$= \begin{bmatrix} \frac{2d}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2d}{t-b} & \frac{t+b}{t-b}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

  - note that this transform entirely ignores *w*
  - this makes sense because scaling a point around the origin (i.e. viewpoint, in eye space) doesn't change its projection
- This is the *projection matrix* for perspective projection

Clipping planes

- In object-order systems we always use at least two *clipping planes* that further constrain the view volume
  - near plane: parallel to view plane; things between it and the viewpoint will not be rendered
  - far plane: also parallel; things behind it will not be rendered
- These planes are:
  - partly to remove unnecessary stuff (e.g. behind the camera)
  - but really to constrain the range of depths
    (we'll see why later)

View volume: perspective (clipped)



Preserving depth through projection

- In practice, when projecting we don't throw away *z*
  - there is still a need to keep track of what is in front and what is behind
- Orthographic: projection simply preserves *z*, and windowing treats *z* the same as *x* and *y*
  - the *near* and *far* planes, at *z* = *n* and *z* = *f*, define the window extent
  - map [*l*,*r*] x [*t*,*b*] x [*n*,*f*] to [–1,1] x [–1,1] x [–1,1]

$$\text{old:}\quad M_o = \begin{bmatrix} \frac{2}{r-l} & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & -\frac{t+b}{t-b} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{new:}\quad M_o = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Preserving depth through projection

- Perspective: can no longer toss out *w*
- Arrange for projection matrix to preserve *n* and *f*

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} \sim \begin{bmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{z} \\ -z \end{bmatrix} = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

  - we're stuck with the *w* row, but choose *a* and *b* to ensure that *z'* = *n* when *z* = *n* and *z'* = *f* when *z* = *f*

$$\tilde{z}(z) = az + b$$

$$z'(z) = \frac{\tilde{z}}{-z} = \frac{az+b}{-z}$$

$$\text{want } z'(n) = n \text{ and } z'(f) = f$$

$$\text{result: } a = -(n+f) \text{ and } b = nf \text{ (try it)}$$

Preserving depth through projection

- So perspective transform (with windowing) is

$$\text{old:}\quad M_p = \begin{bmatrix} \frac{2}{r-l} & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & -\frac{t+b}{t-b} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} \frac{2d}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2d}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

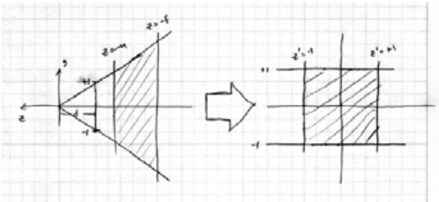
$$\text{new:}\quad M_p = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & -(n+f) & -nf \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

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

NOTE: Book assumes d=n 1

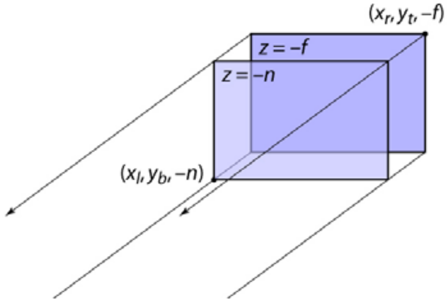
Clip coordinates

- Projection matrix maps from eye space to *clip space*
- In this space, the two-unit cube $[-1, 1]^3$ contains exactly what needs to be drawn
- It's called "clip" coordinates because everything outside of this box is clipped out of the view
  - this can be done at this point, geometrically
  - or it can be done implicitly later on by careful rasterization



OpenGL view frustum: orthographic

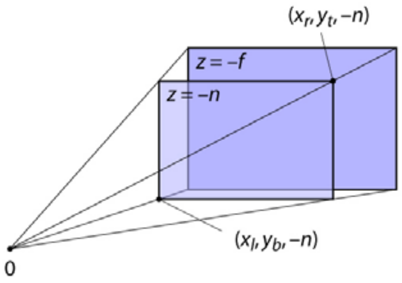
```
glOrtho(xmin, xmax, ymin, ymax, near, far)
```



Note OpenGL puts the near and far planes at $-n$ and $-f$ so that the user can give positive numbers

OpenGL view frustum: perspective

```
glFrustum(xmin, xmax, ymin, ymax, near, far);
```



Note OpenGL puts the near and far planes at $-n$ and $-f$ so that the user can give positive numbers

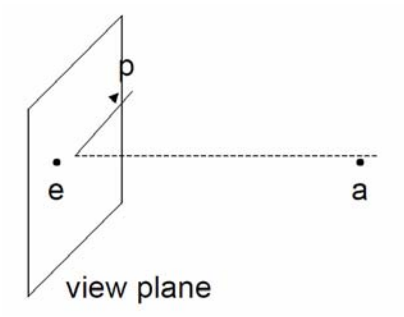
OpenGL: Specifying Perspective

Two approaches:

1. `glFrustum(xmin, xmax, ymin, ymax, near, far);`
   - Analogous to glOrtho(…)
   - Can be painful in practice
2. `gluPerspective(fovy, aspect, near, far);`
   - near and far as before
   - Fovy specifies field of view as height (y) angle

OpenGL: `gluLookAt()` Function

- Convenient way to position camera
- `gluLookAt(ex, ey, ez, ax, ay, az, px, py, pz);`
  - e = eye point
  - a = at point
  - p = up vector



Vertex processing: spaces

- Standard sequence of transforms