Pipeline and Rasterization

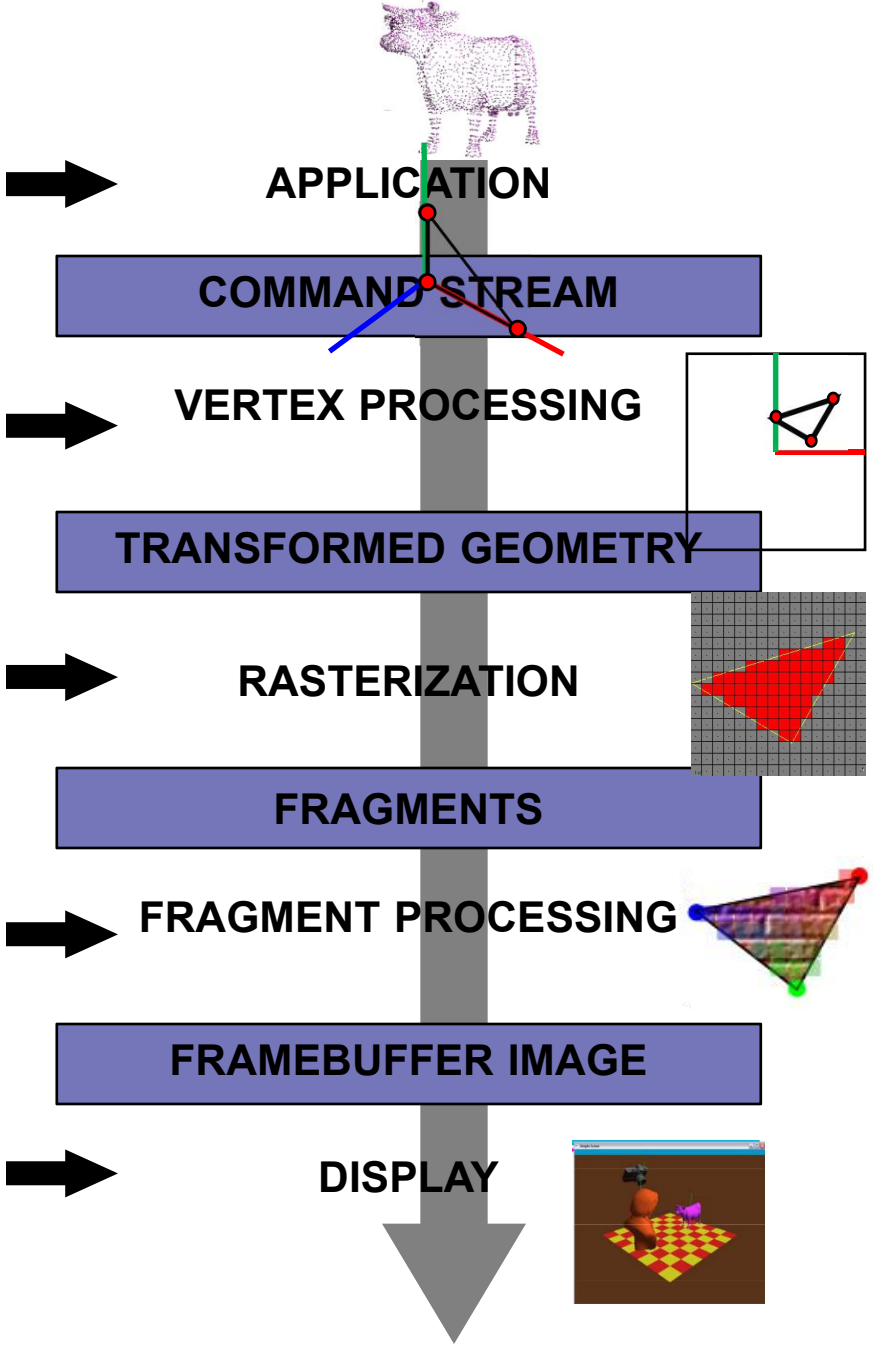
Lecture 7

The graphics pipeline

- The standard approach to object-order graphics
- Many versions exist
  - software, e.g. Pixar's REYES architecture
    - many options for quality and flexibility
  - hardware, e.g. graphics cards in PCs
    - amazing performance: millions of triangles per frame
- We'll focus on an abstract version of hardware pipeline
- "Pipeline" because of the many stages
  - very parallelizable
  - leads to remarkable performance of graphics cards (many times the flops of the CPU at ~1/5 the clock speed)

Pipeline overview

you are here



3D transformations; shading

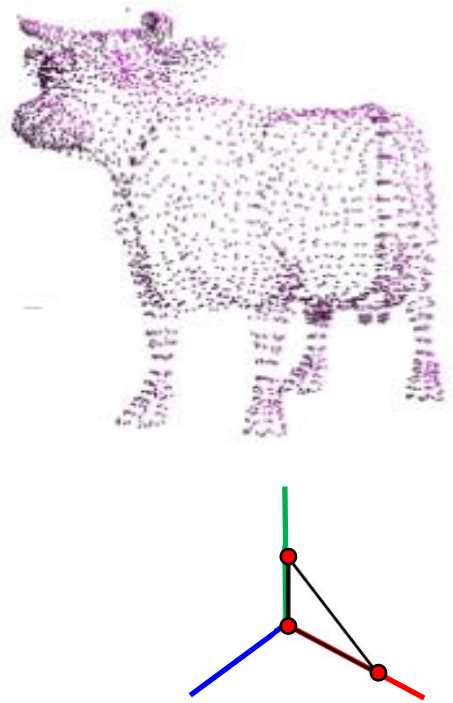
conversion of primitives to pixels

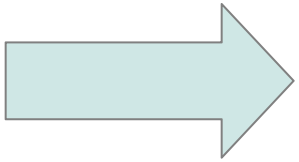
blending, compositing, shading

user sees this

Primitives

- Points
- Line segments
  - and chains of connected line segments
- Triangles
- And that's all!
  - Curves? Approximate them with chains of line segments
  - Polygons? Break them up into triangles
  - Curved regions? Approximate them with triangles
- Trend has been toward minimal primitives
  - simple, uniform, repetitive: good for parallelism

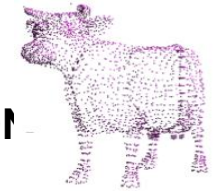




3D Viewing

Pipeline overview

you are here



APPLICATION

COMMAND STREAM

3D transformations; shading

VERTEX PROCESSING

TRANSFORMED GEOMETRY

conversion of primitives to pixels

RASTERIZATION

FRAGMENTS

blending, compositing, shading

FRAGMENT PROCESSING

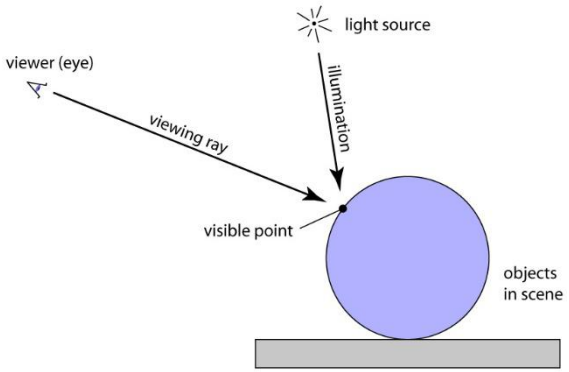
FRAMEBUFFER IMAGE

user sees this

DISPLAY

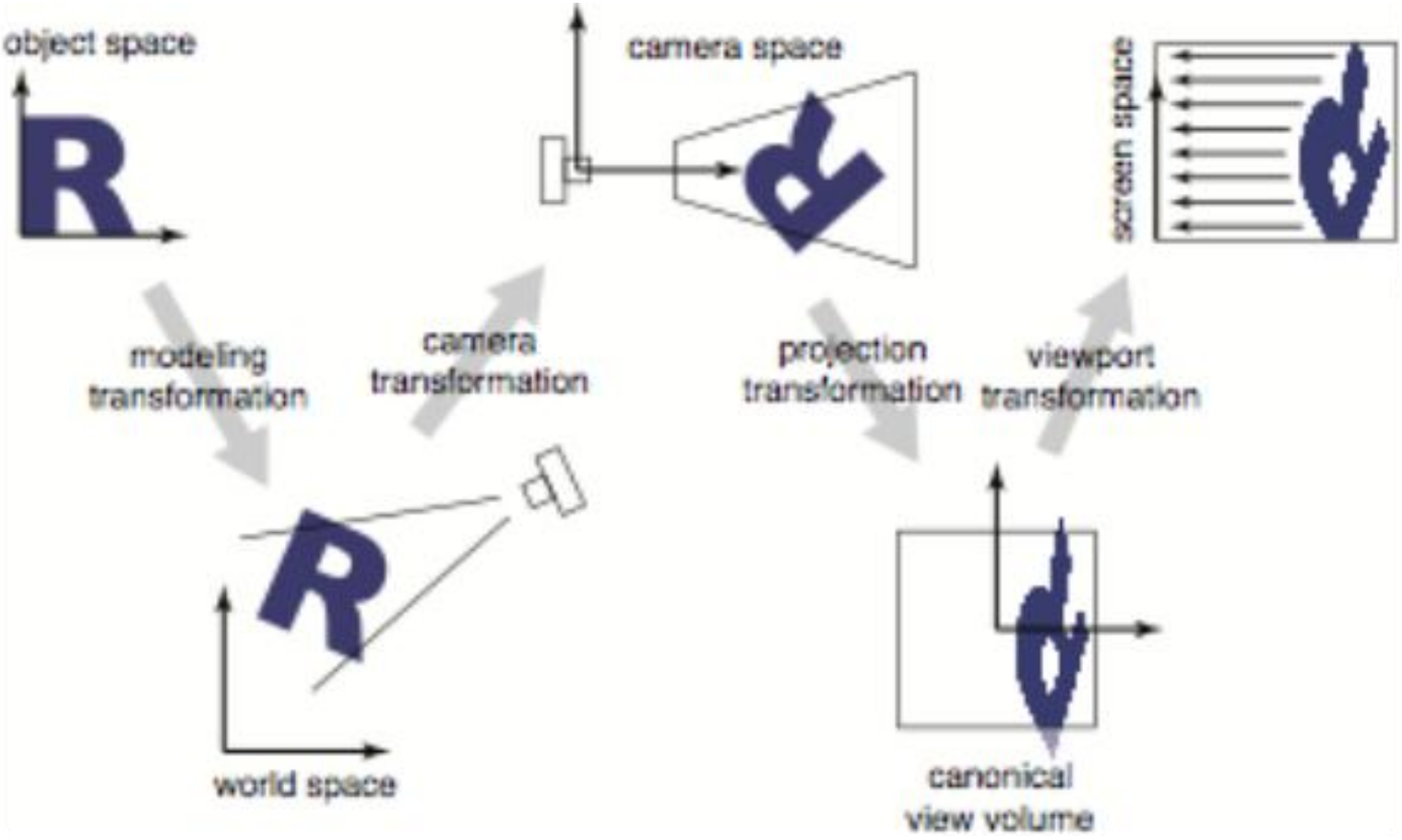
Viewing, backward and forward

- 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.



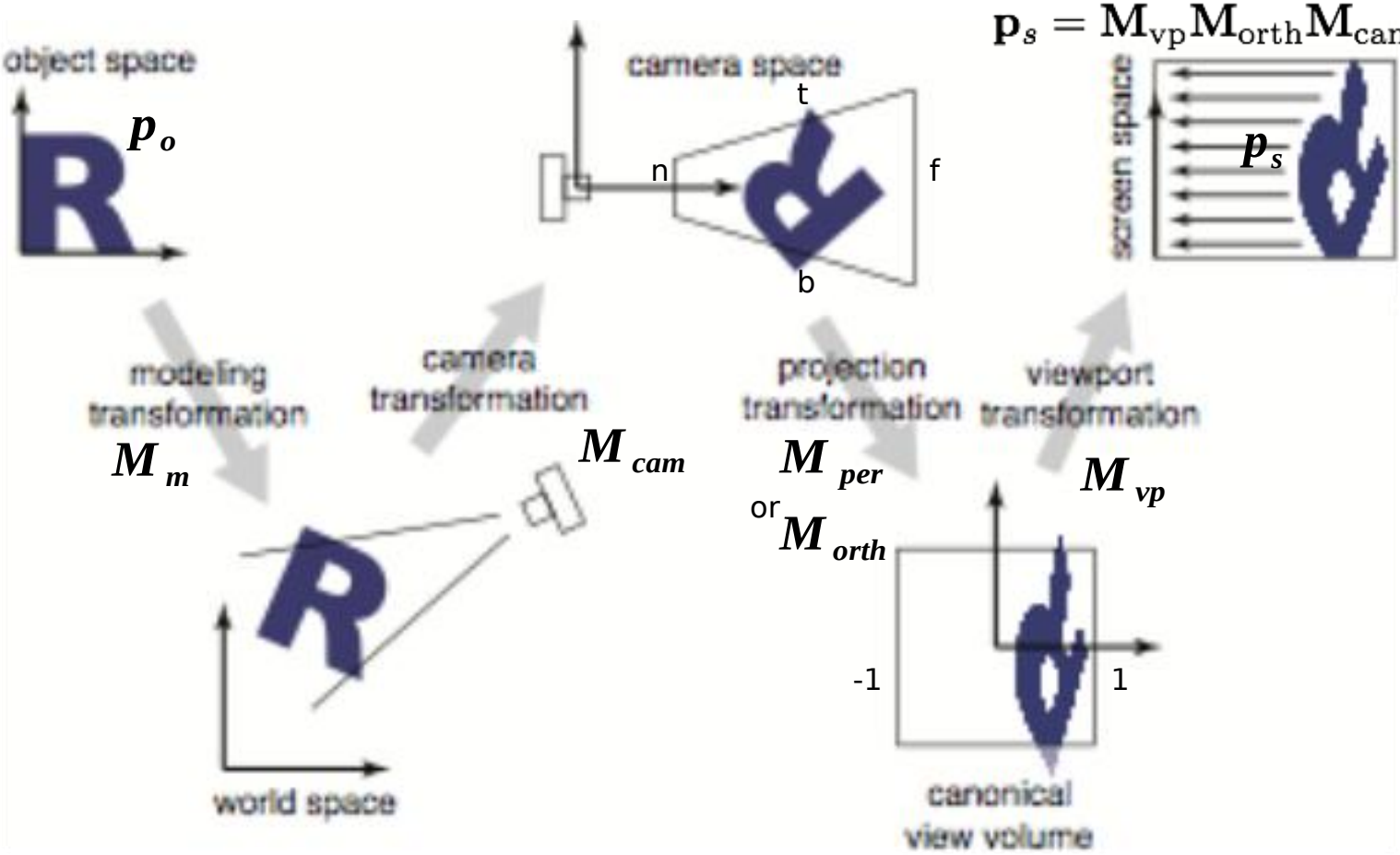
Pipeline of transformations

- Standard sequence of transforms



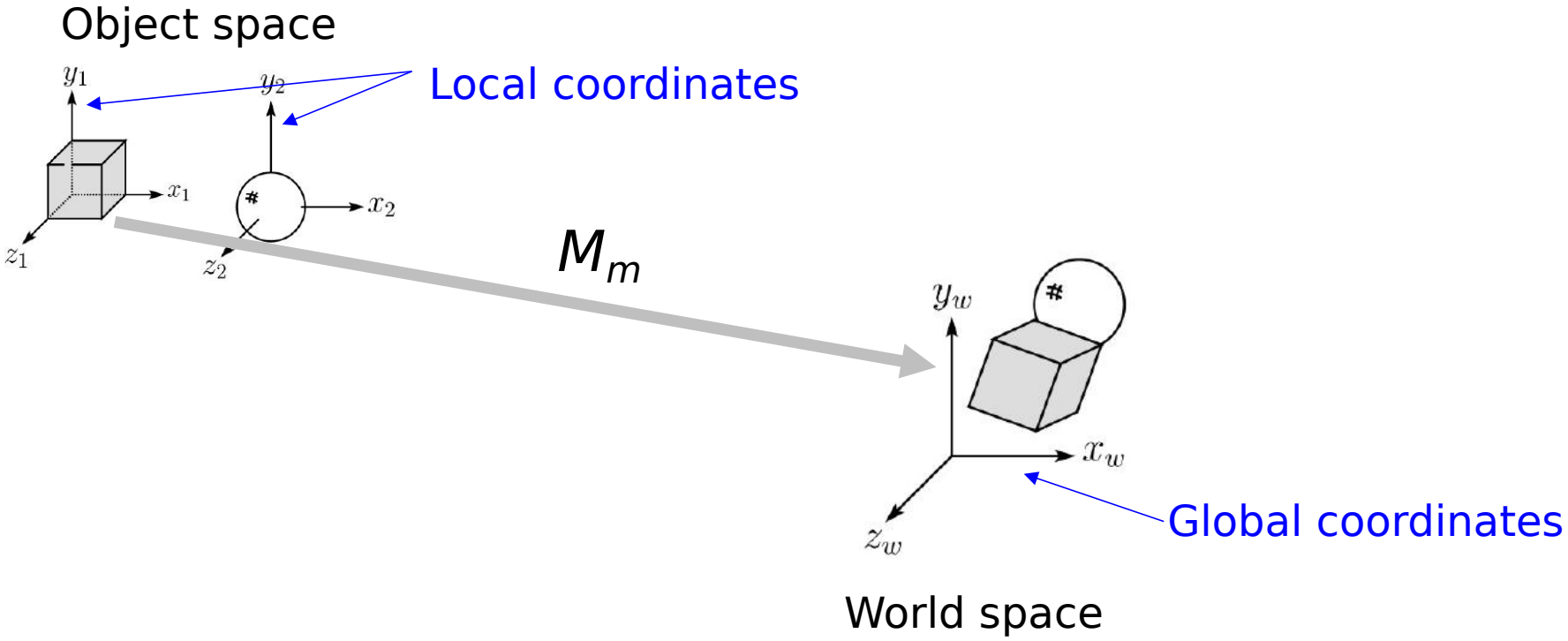
Pipeline of transformations

- Standard sequence of transforms



Modeling matrices

- Geometry would originally have been in the object's local coordinates; transform into world coordinates is called the *modeling matrix*, M_m
- Composite affine transformations



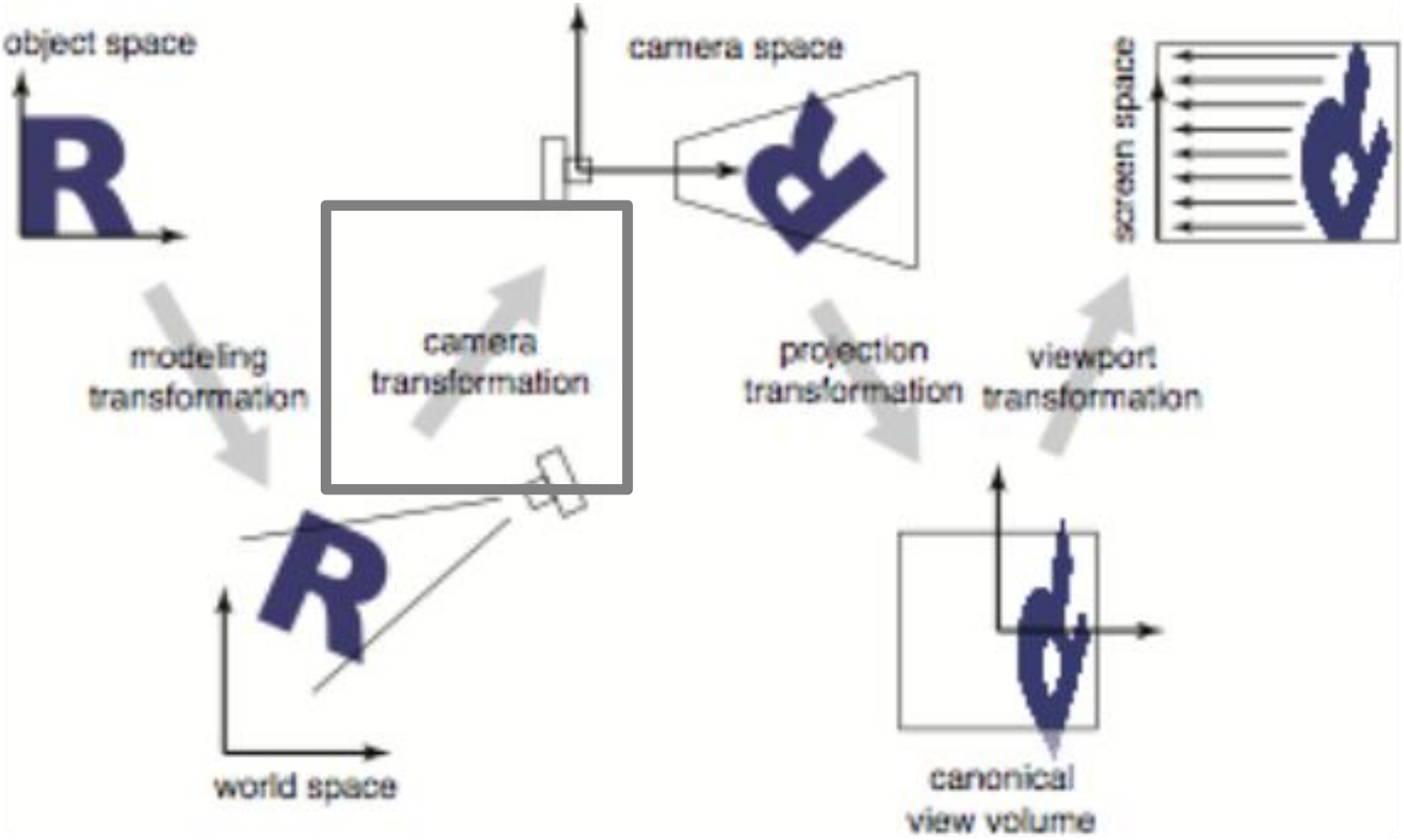
Transforming points

$$\begin{bmatrix} M & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} = \begin{bmatrix} M\mathbf{p} + \mathbf{t} \\ 1 \end{bmatrix}$$

- M : linear transformation (such as rotation)
- t : translation

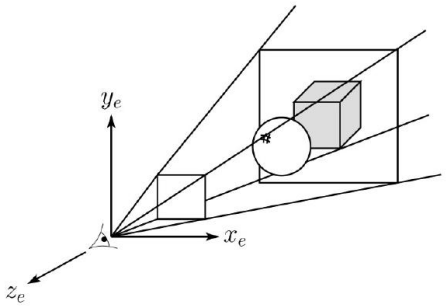
Pipeline of transformations

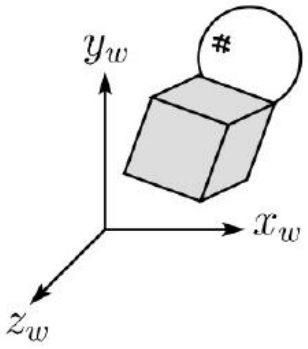
- Standard sequence of transforms



Viewing transformation

View space
(Camera space)





M_v

Placing the “camera”

World space

the camera matrix rewrites all coordinates in eye space

Camera matrices

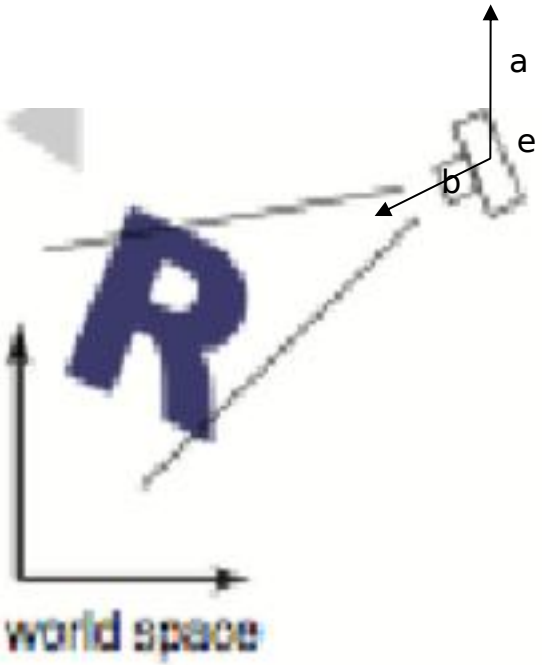
- Eye coordinates (= camera space)
  - before we do any of this stuff we need to transform objects into that space
- Transform from world to eye space is traditionally called the *viewing matrix*, M_v
- Remember the *modeling matrix*, M_m
- OpenGL combines the two into a *modelview* matrix

$M = M_v * M_m$

Viewing transformation M_{view}

- Upvector a, view dir b, eye position e
- u=bxa / ||bxa||
- v=uxb (=new upvector)
- w=uxv
- Then M_view=

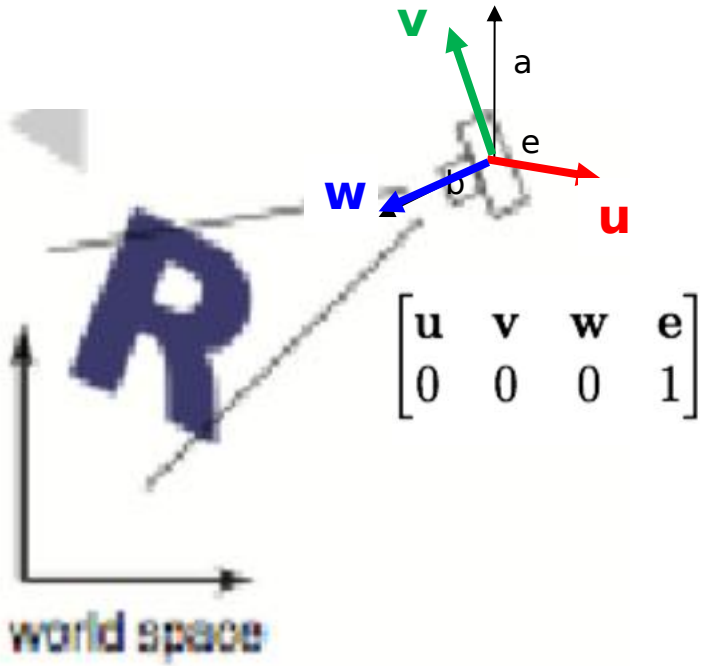
$$\begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{e} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1}$$



Viewing transformation M_{view}

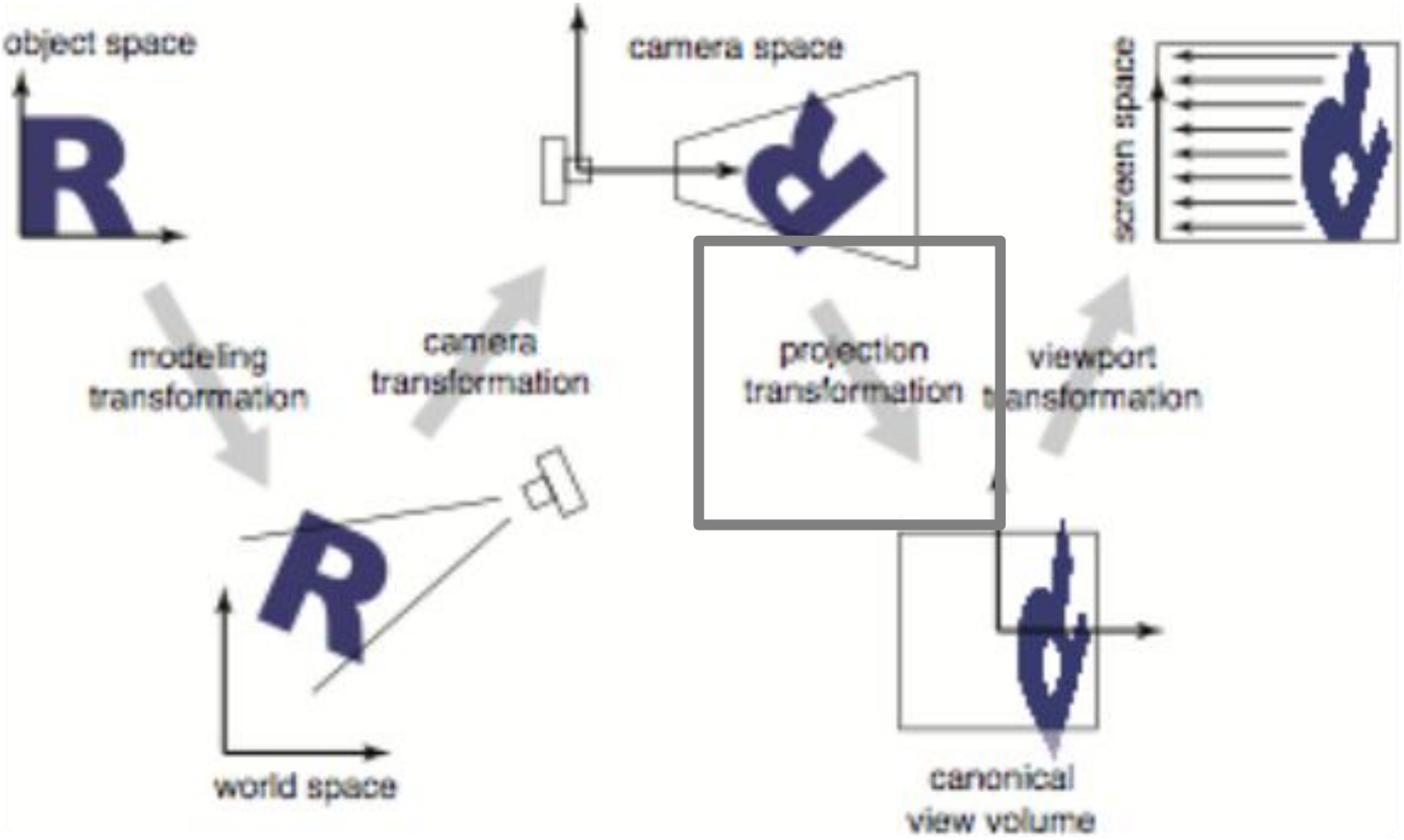
- Upvector a, view dir b, eye position e
- u=bxa / ||bxa||
- v=uxb (=new upvector)
- w=uxv
- Then M_view=

$$\begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{e} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1}$$



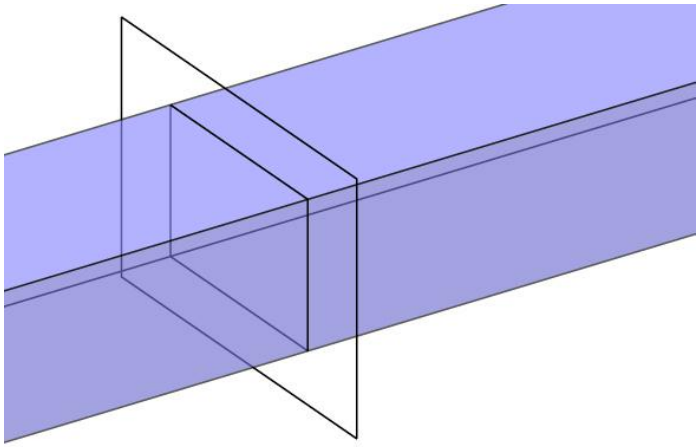
Pipeline of transformations

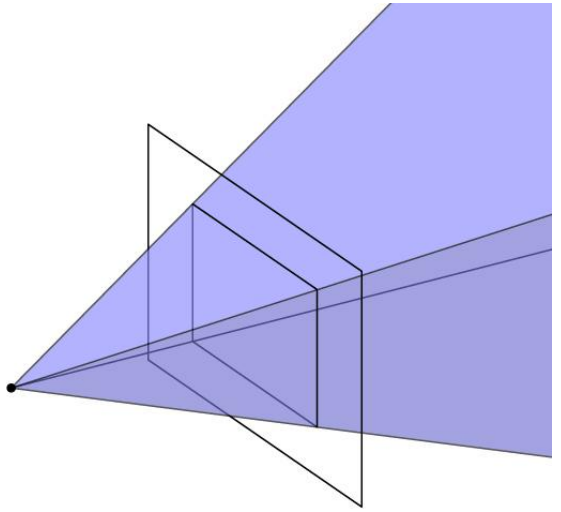
- Standard sequence of transforms



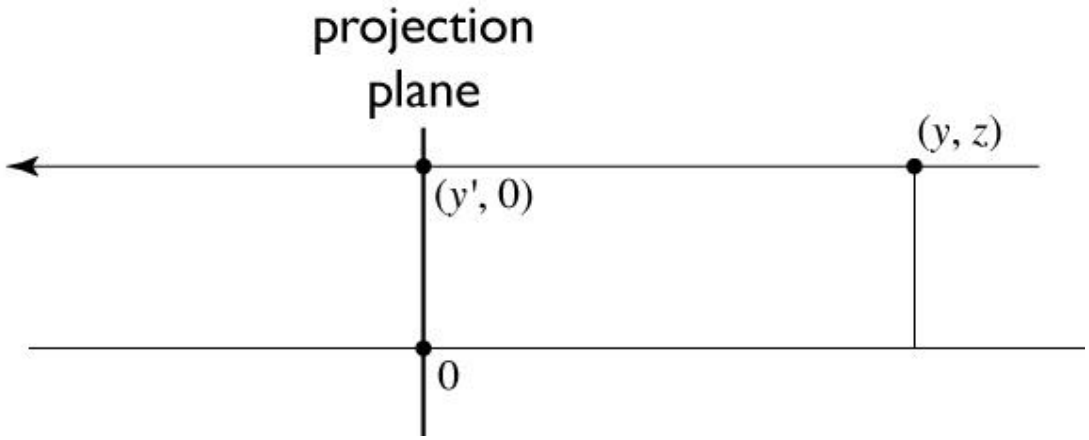
Mathematics of projection

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





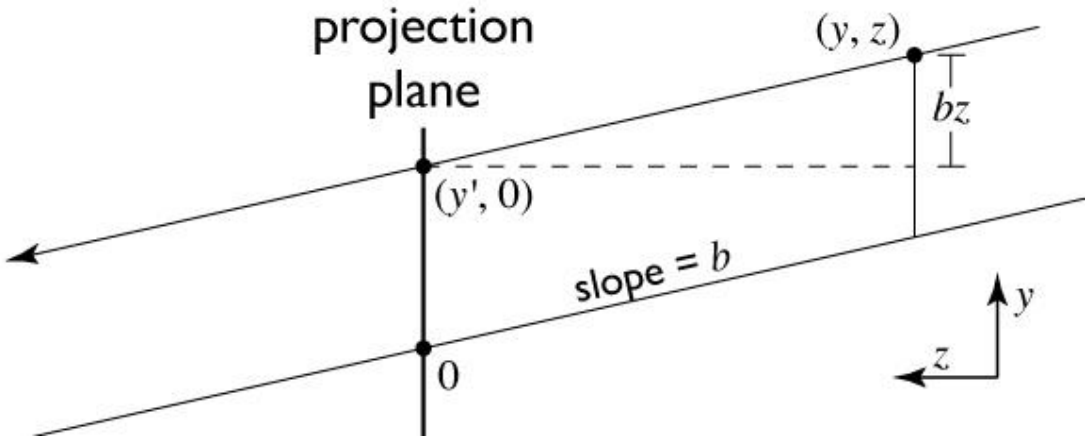
Parallel projection: orthographic



to implement orthographic, just toss out *z*:

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

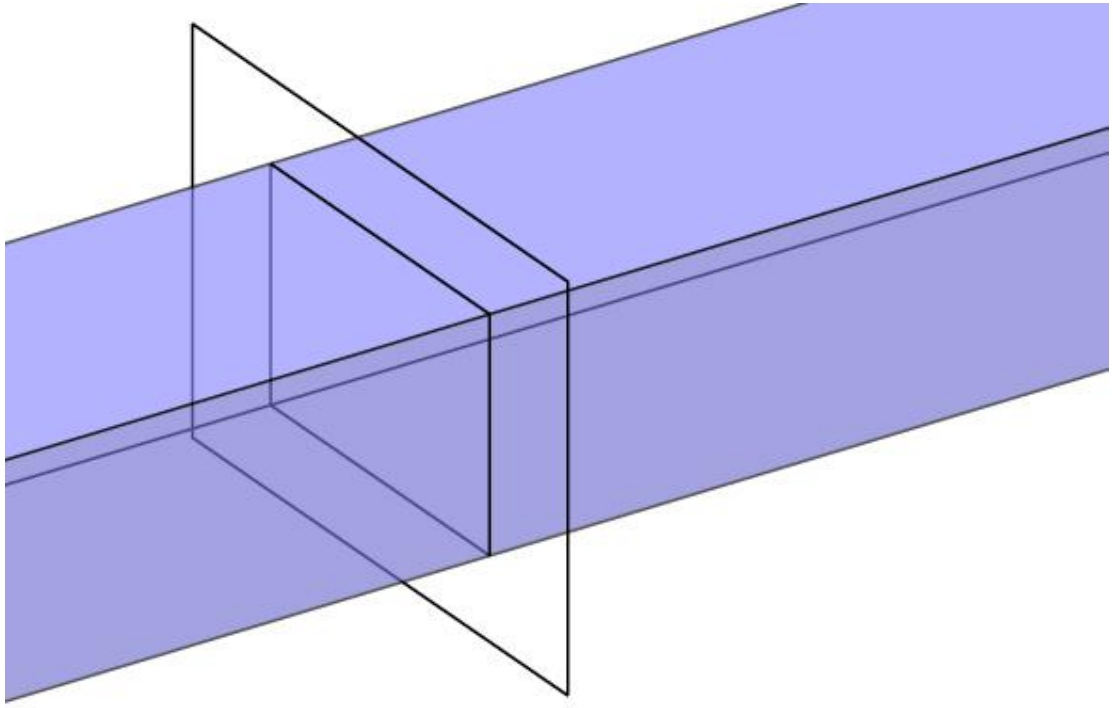
Parallel projection: oblique



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



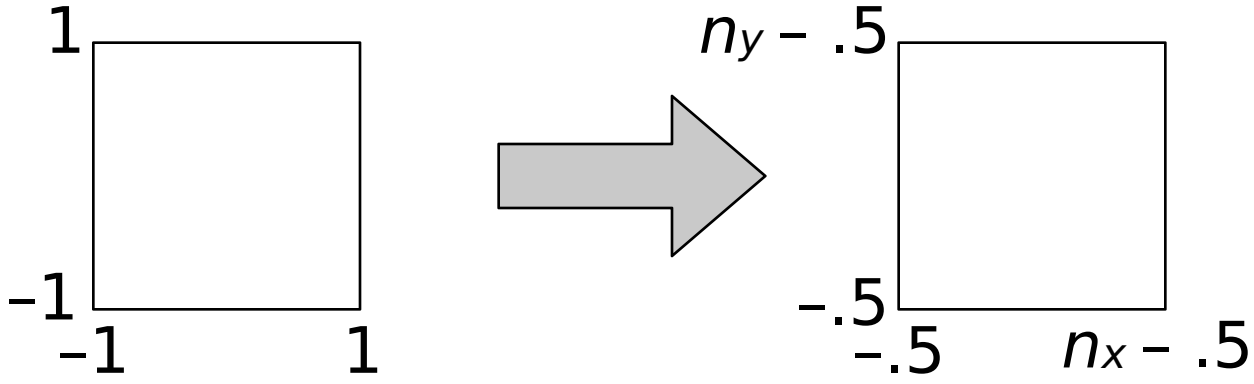
Viewing a cube of size 2

- Start by looking at a restricted case: the *canonical view volume*
- It is the cube $[-1,1]^3$, viewed from the *z* direction
- Matrix to project it into a square image in $[-1,1]^2$ is trivial:

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

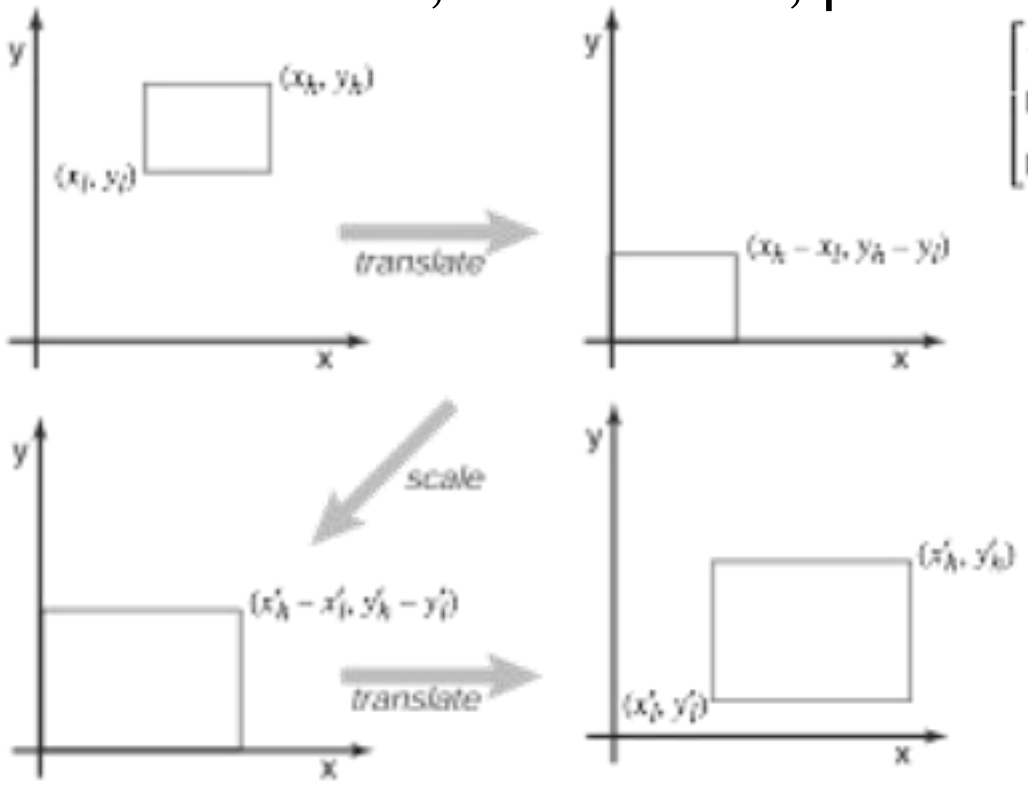
Viewing a cube of size 2

- To draw in image, need coordinates in pixel units, though
- Exactly the opposite of mapping (i,j) to (u,v) in ray generation



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



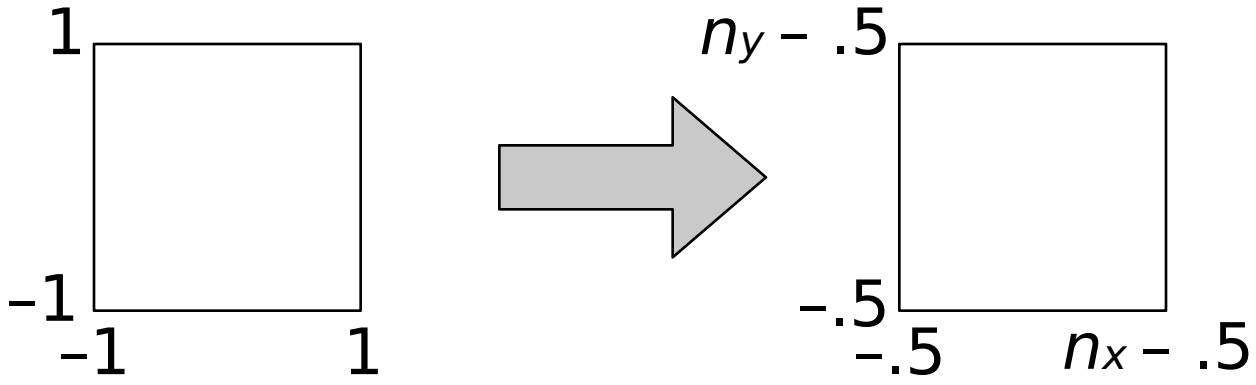
$$\begin{bmatrix} 1 & 0 & x'_l \\ 0 & 1 & y'_l \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{x'_h - x'_l}{x_h - x_l} & 0 & 0 \\ 0 & \frac{y'_h - y'_l}{y_h - y_l} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_l \\ 0 & 1 & -y_l \\ 0 & 0 & 1 \end{bmatrix}$$

translate scale translate

$$\begin{bmatrix} \frac{x'_h - x'_l}{x_h - x_l} & 0 & \frac{x'_l x_h - x'_h x_l}{x_h - x_l} \\ 0 & \frac{y'_h - y'_l}{y_h - y_l} & \frac{y'_l y_h - y'_h y_l}{y_h - y_l} \\ 0 & 0 & 1 \end{bmatrix}$$

[Shirley3e f. 6-16; eq. 6-6]

Viewport transformation



$$\begin{bmatrix} x_{\text{screen}} \\ y_{\text{screen}} \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{n_x}{2} & 0 & \frac{n_x-1}{2} \\ 0 & \frac{n_y}{2} & \frac{n_y-1}{2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{\text{canonical}} \\ y_{\text{canonical}} \\ 1 \end{bmatrix}$$

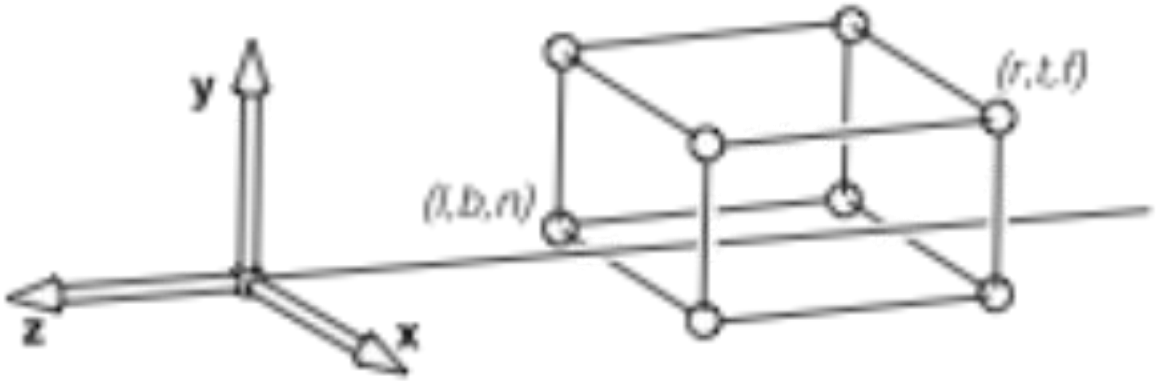
Viewport transformation

- In 3D, carry along *z* for the ride
  - one extra row and column

$$\mathbf{M}_{\text{vp}} = \begin{bmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x-1}{2} \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y-1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Orthographic projection

- First generalization: different view rectangle
  - retain the minus-*z* view direction



specify view by left, right, top, bottom (as in RT)

also near, far

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)

Orthographic projection

- We can implement this by mapping the view volume
  to the canonical view volume.
- This is just a 3D windowing transformation!

$$\begin{bmatrix} \frac{x'_n - x'_1}{x_n - x_1} & 0 & 0 & \frac{x'_n x_1 - x'_1 x_n}{x_n - x_1} \\ 0 & \frac{y'_n - y'_1}{y_n - y_1} & 0 & \frac{y'_1 y_n - y'_n y_1}{y_n - y_1} \\ 0 & 0 & \frac{z'_n - z'_1}{z_n - z_1} & \frac{z'_1 z_n - z'_n z_1}{z_n - z_1} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{M}_{\text{orth}} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Orthographic projection

- Verification: transform the two corners of the view volume

$$\boldsymbol{M}_{orth}\begin{bmatrix} l \\ b \\ f \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ -1 \\ 1 \end{bmatrix}$$

$$\boldsymbol{M}_{orth}\begin{bmatrix} r \\ t \\ n \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\mathbf{M}_{\text{orth}} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

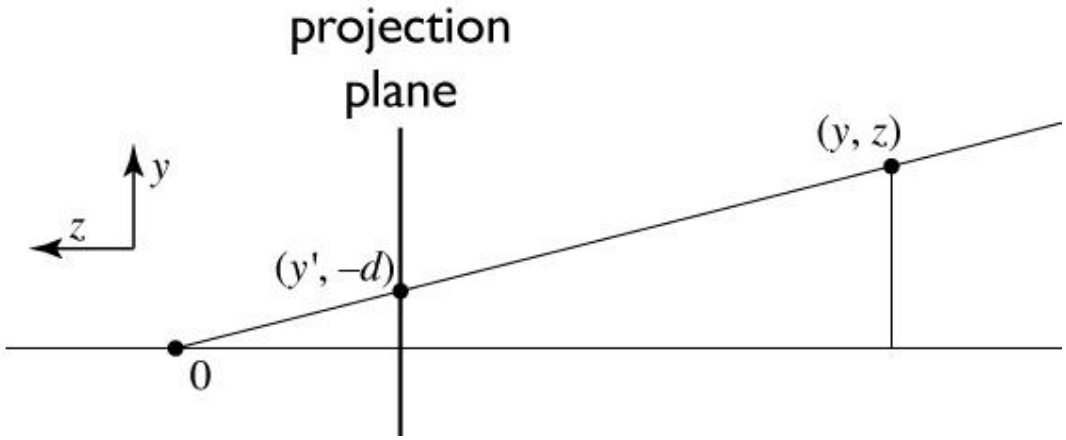
Orthographic transformation chain

- Start with coordinates in object's local coordinates
- Transform into world coords (modeling transform, M_m)
- Transform into eye coords (camera xf., $M_{\text{cam}} = F_c^{-1}$)
- Orthographic projection, M_{orth}
- Viewport transform, M_{vp}

$$\mathbf{p}_s = \mathbf{M}_{\text{vp}}\mathbf{M}_{\text{orth}}\mathbf{M}_{\text{cam}}\mathbf{M}_{\text{m}}\mathbf{p}_o$$

$$\begin{bmatrix} x_s \\ y_s \\ z_c \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x-1}{2} \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y-1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{e} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \mathbf{M}_{\text{m}} \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix}$$

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*

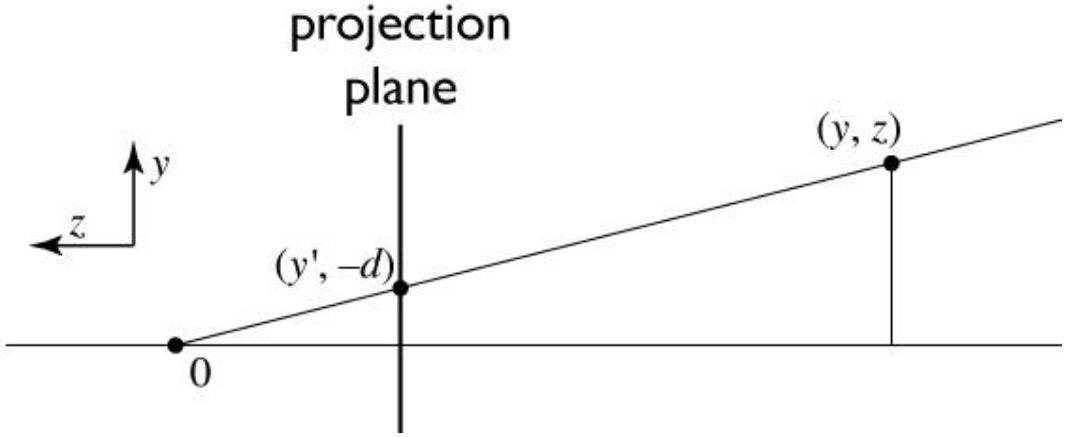
$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \sim \begin{bmatrix} wx \\ wy \\ wz \\ w \end{bmatrix}$$

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

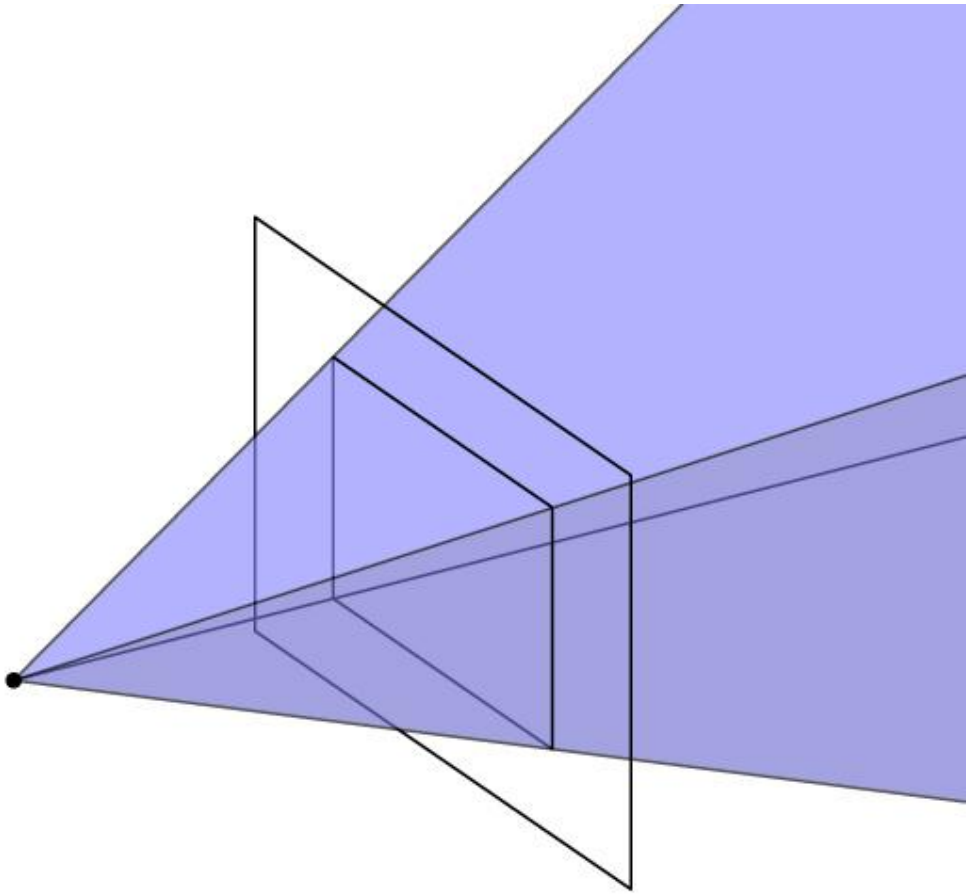
Perspective projection



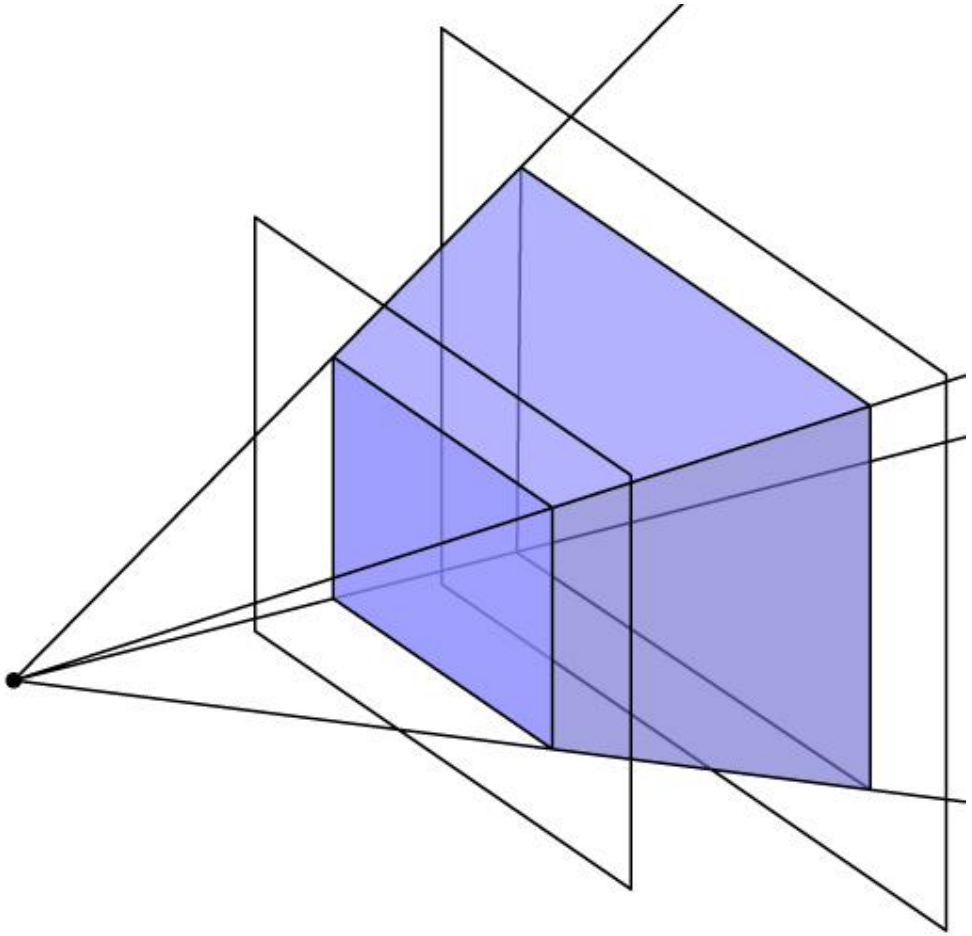
to implement perspective, just move z to w:

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

View volume: perspective



View volume: perspective (clipped)



Carrying depth through perspective

- Perspective has a varying denominator—can't preserve depth!
- Compromise: preserve depth on near and far planes

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

  - that is, choose a and b so that $z'(n) = n$ and $z'(f) = f$.

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

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

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

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

Official perspective matrix

- Use near plane distance as the projection distance
  - i.e., $d = -n$
- Scale by -1 to have fewer minus signs
  - scaling the matrix does not change the projective transformation

$$\mathbf{P} = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Official perspective matrix

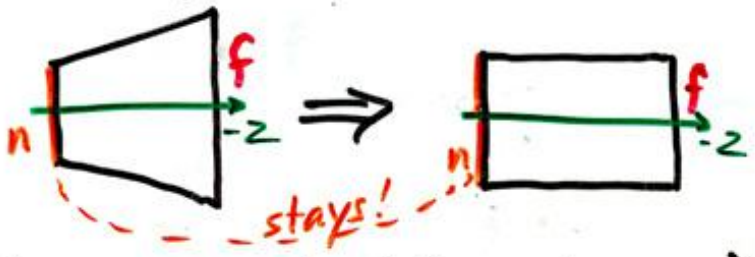
$$\mathbf{P} = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\boldsymbol{P}\begin{bmatrix} x \\ y \\ n \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ (n+f)n - fn \\ n \end{bmatrix} \approx \begin{bmatrix} x \\ y \\ n \\ 1 \end{bmatrix}$$

$$\boldsymbol{P}\begin{bmatrix} x \\ y \\ f \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ (n+f)f - fn \\ f \end{bmatrix} \approx \begin{bmatrix} x(n/f) \\ y(n/f) \\ f \\ 1 \end{bmatrix}$$

The Perspective Matrix

Shirley page 170 [152]



$$P\,x = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} nx \\ ny \\ nz+fz-nf \\ z \end{pmatrix}$$

This does perspective division!

homogenize to 3D: $\Rightarrow \begin{pmatrix} \frac{n}{z}x \\ \frac{n}{z}y \\ n+f-\frac{nf}{z} \\ 1 \end{pmatrix}$

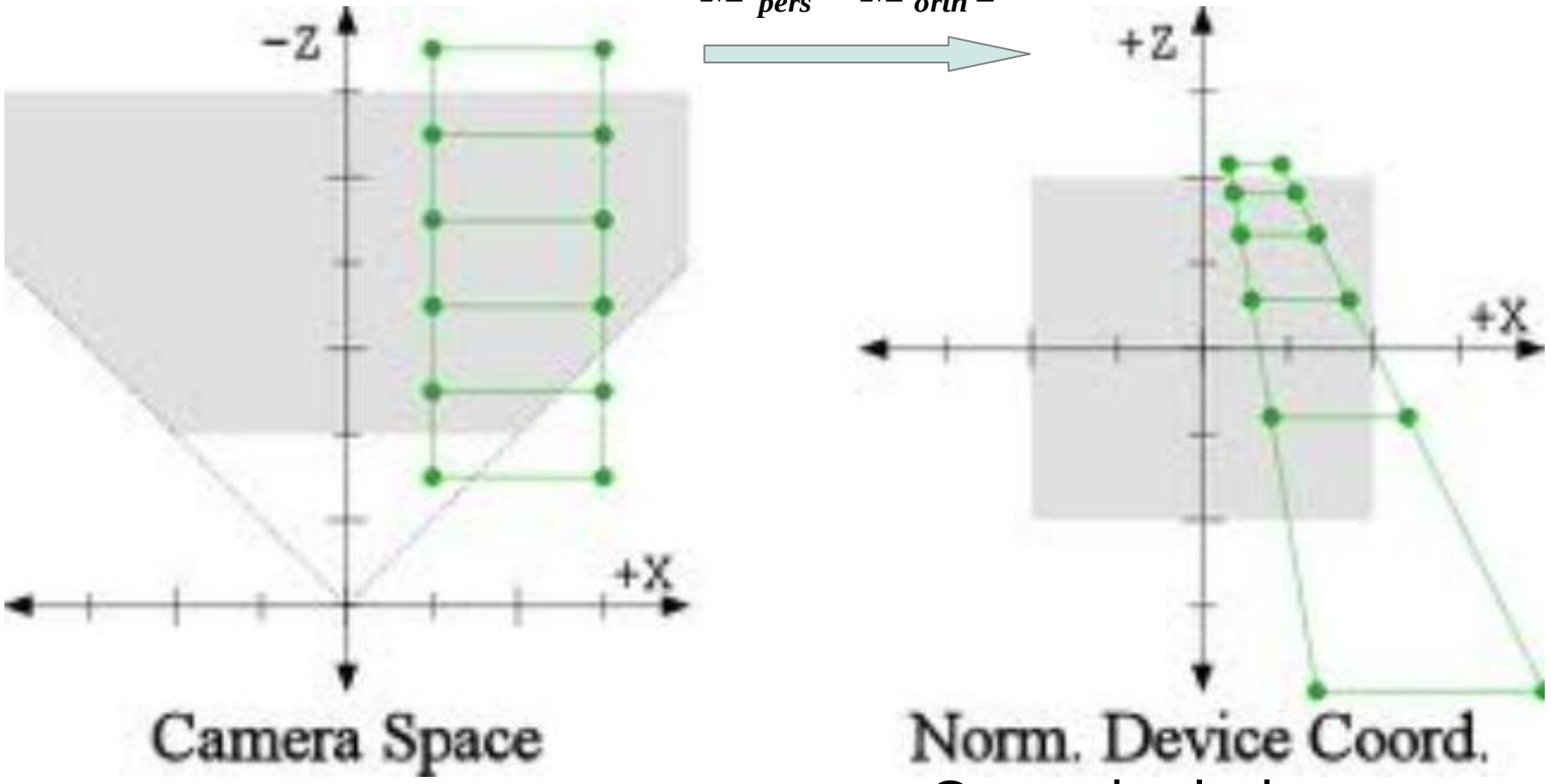
This does not depend on x,y →

⇒ Plane z=const remains a plane ⊥ to z-axis!

...Midplane at: $z = \frac{n+f}{2}$ goes to: $z' = n+f-\frac{2nf}{n+f}$

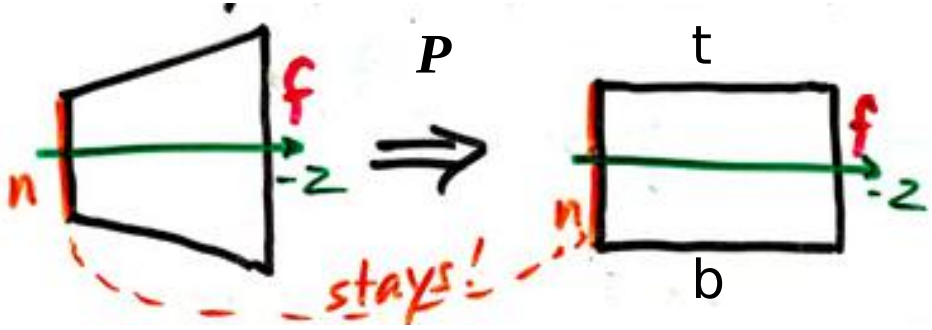
or: $z' = \frac{n+f}{2} + \frac{(n-f)^2}{2(n+f)}$ (⇒ shift to −z)

$$M_{pers} = M_{orth} P$$



Canonical view volume

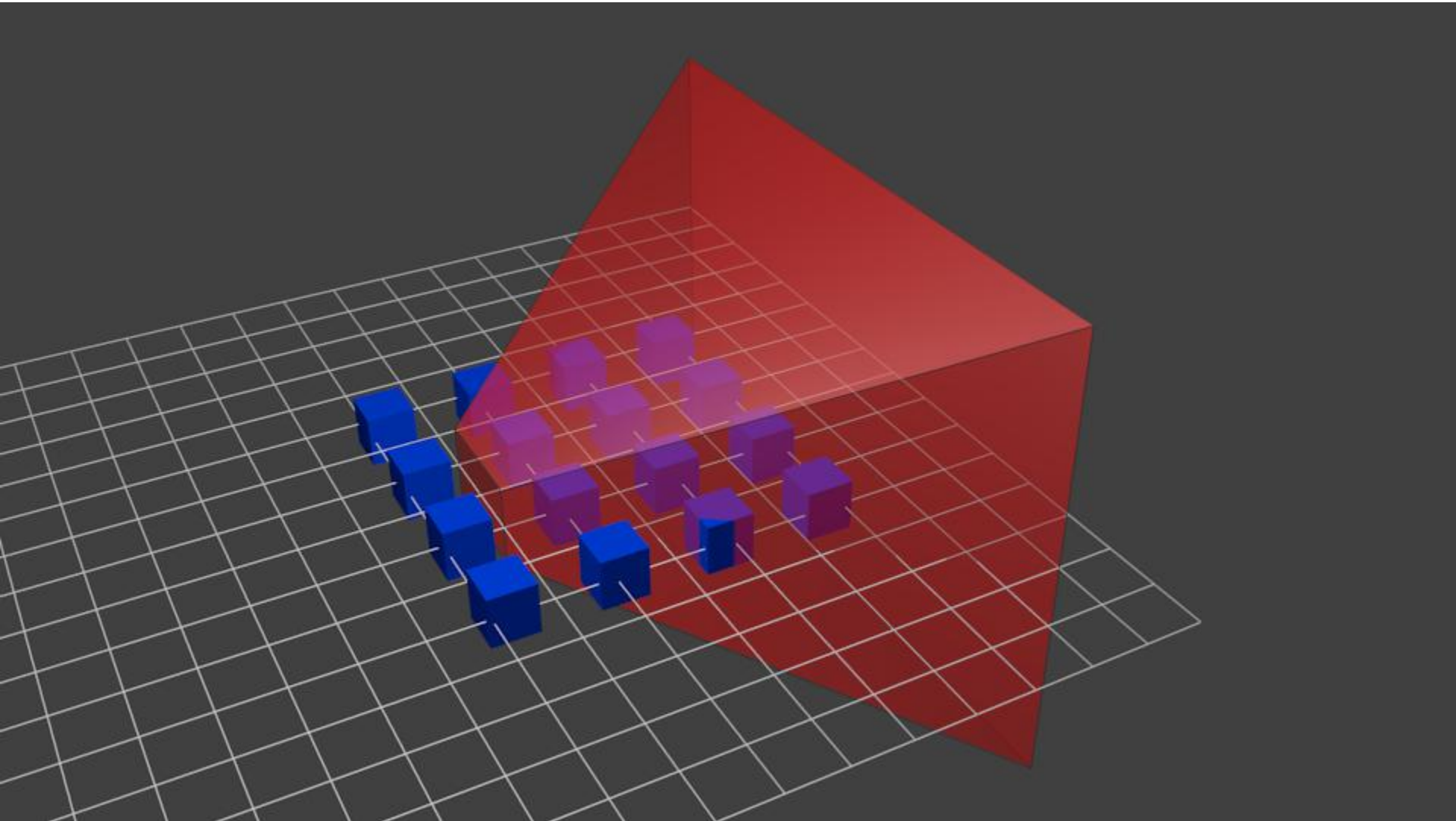
M_{orth}



An Example of Perspective Projection

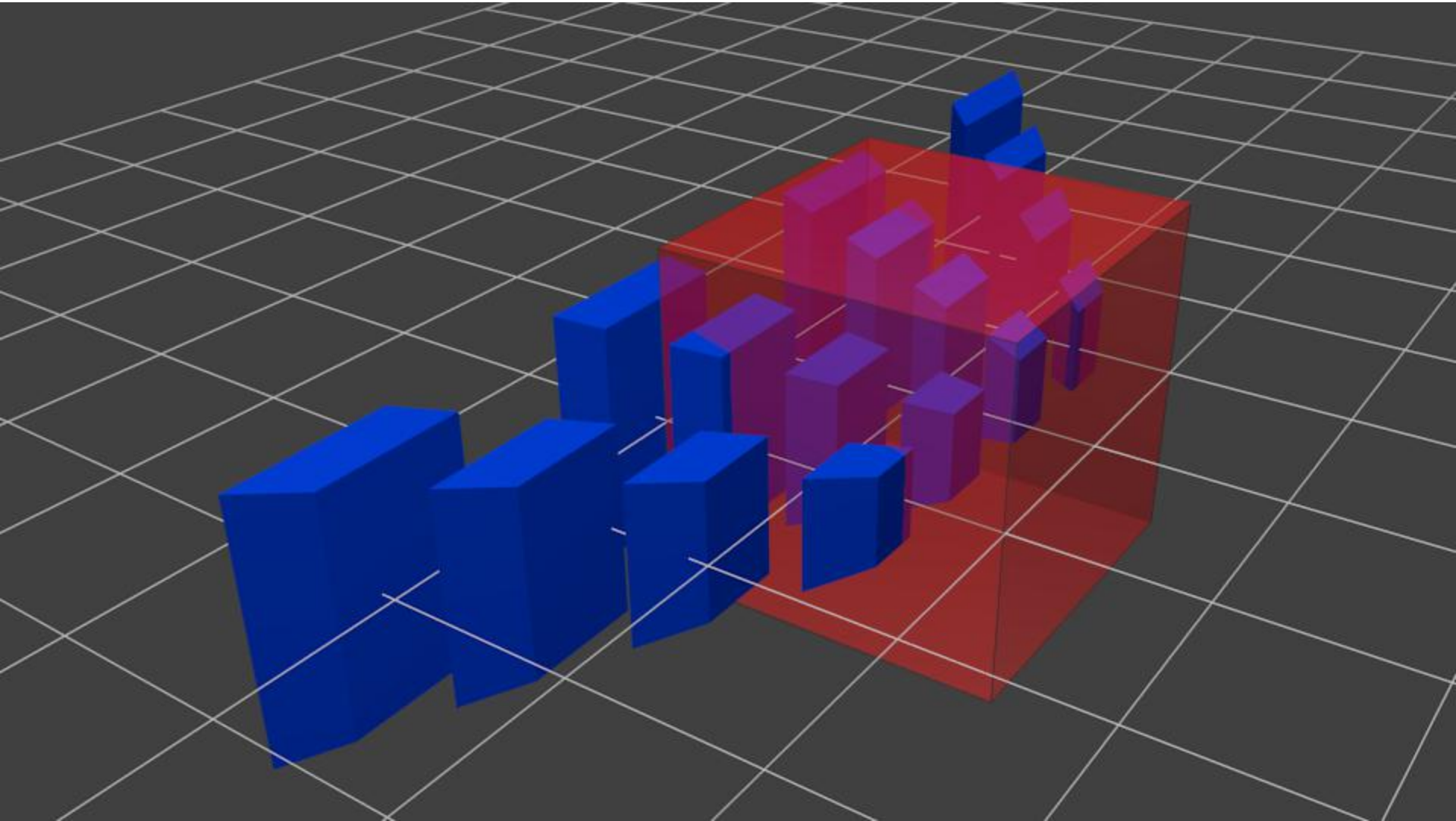
Original 3D scene

Red: viewing frustum, Blue: objects



An Example of Perspective Projection

After perspective projection



Perspective projection matrix

- Product of perspective matrix with orth. projection matrix

$$\mathbf{M}_{\text{per}} = \mathbf{M}_{\text{orth}}\mathbf{P}$$

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

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

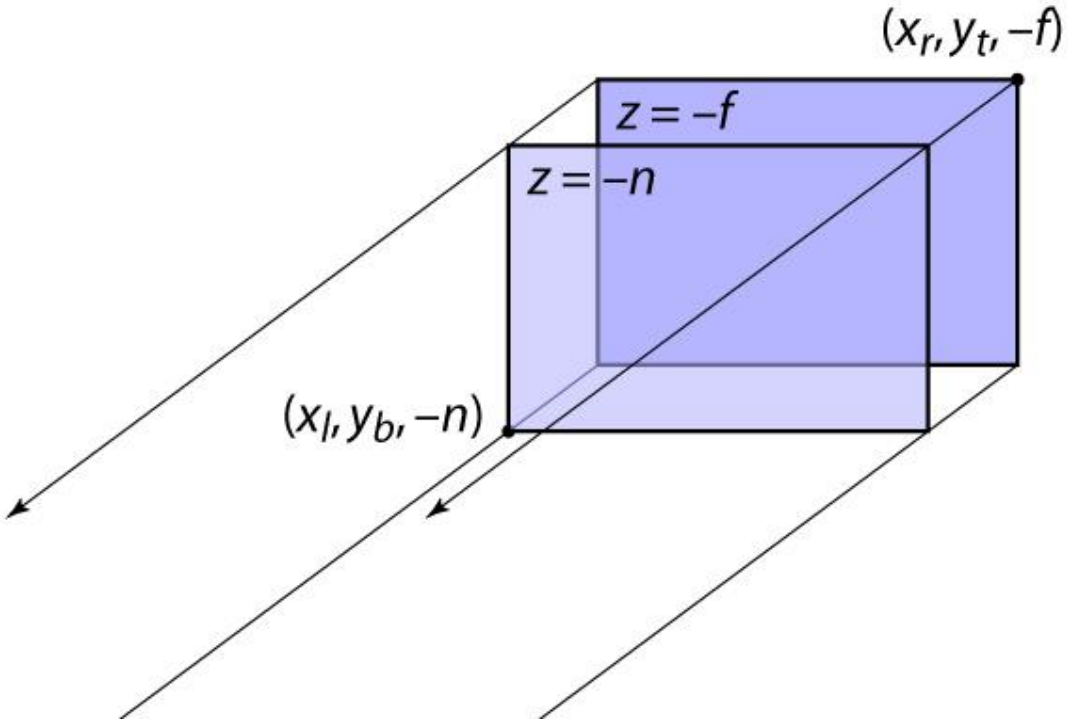
Perspective transformation chain

- Transform into world coords (modeling transform, M_m)
- Transform into eye coords (camera xf., $M_{\text{cam}} = F_c^{-1}$)
- Perspective matrix, P
- Orthographic projection, M_{orth}
- Viewport transform, M_{vp}

$$\mathbf{p}_s = \mathbf{M}_{\text{vp}}\mathbf{M}_{\text{orth}}\mathbf{P}\mathbf{M}_{\text{cam}}\mathbf{M}_{\text{m}}\mathbf{p}_o$$

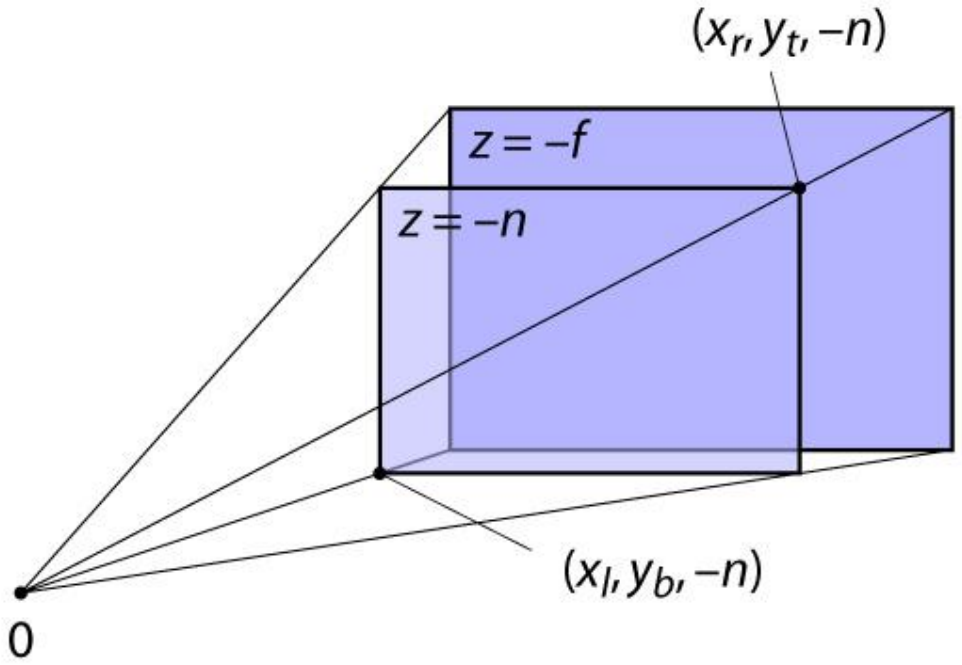
$$\begin{bmatrix} x_s \\ y_s \\ z_c \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x-1}{2} \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y-1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix} \mathbf{M}_{\text{cam}}\mathbf{M}_{\text{m}} \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix}$$

OpenGL view frustum: orthographic



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

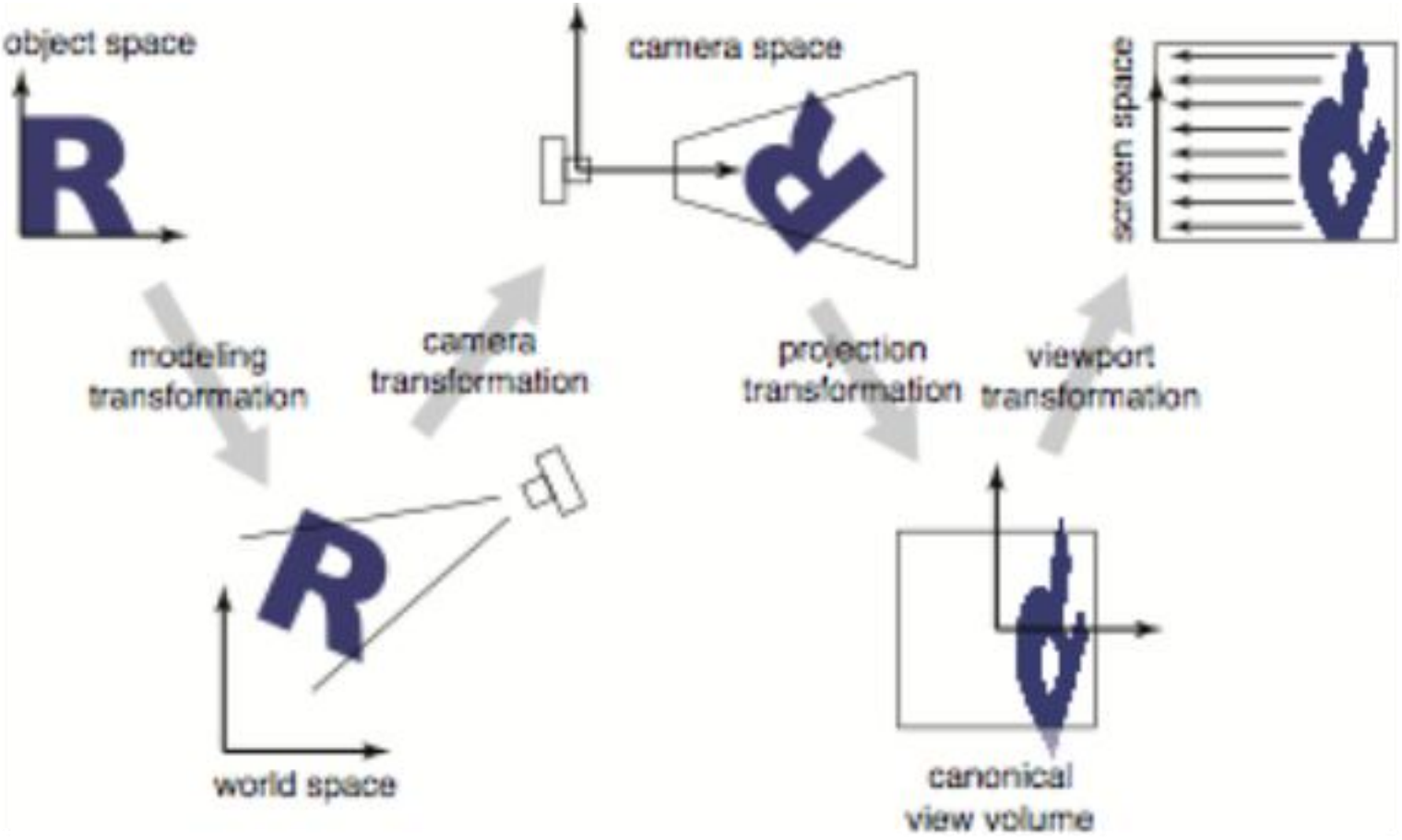
OpenGL view frustum: perspective



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

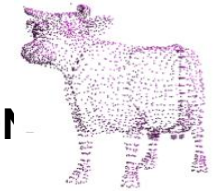
Pipeline of transformations

- Standard sequence of transforms



Pipeline overview

you are here



APPLICATION

COMMAND STREAM

3D transformations; shading

VERTEX PROCESSING

TRANSFORMED GEOMETRY

conversion of primitives to pixels

RASTERIZATION

FRAGMENTS

blending, compositing, shading

FRAGMENT PROCESSING

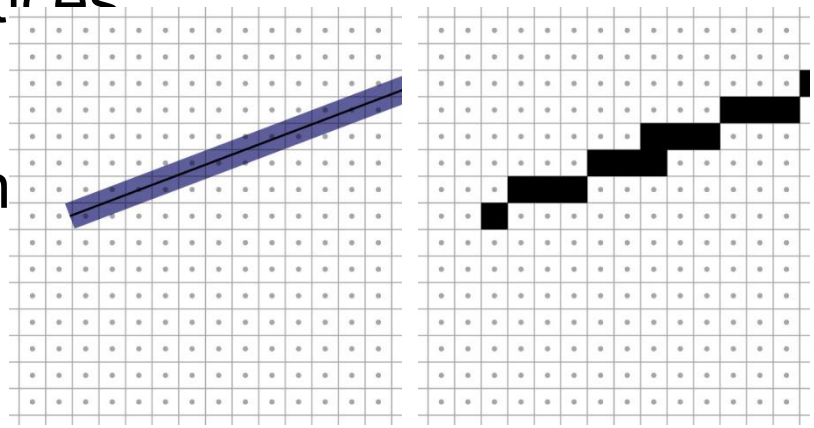
FRAMEBUFFER IMAGE

user sees this

DISPLAY

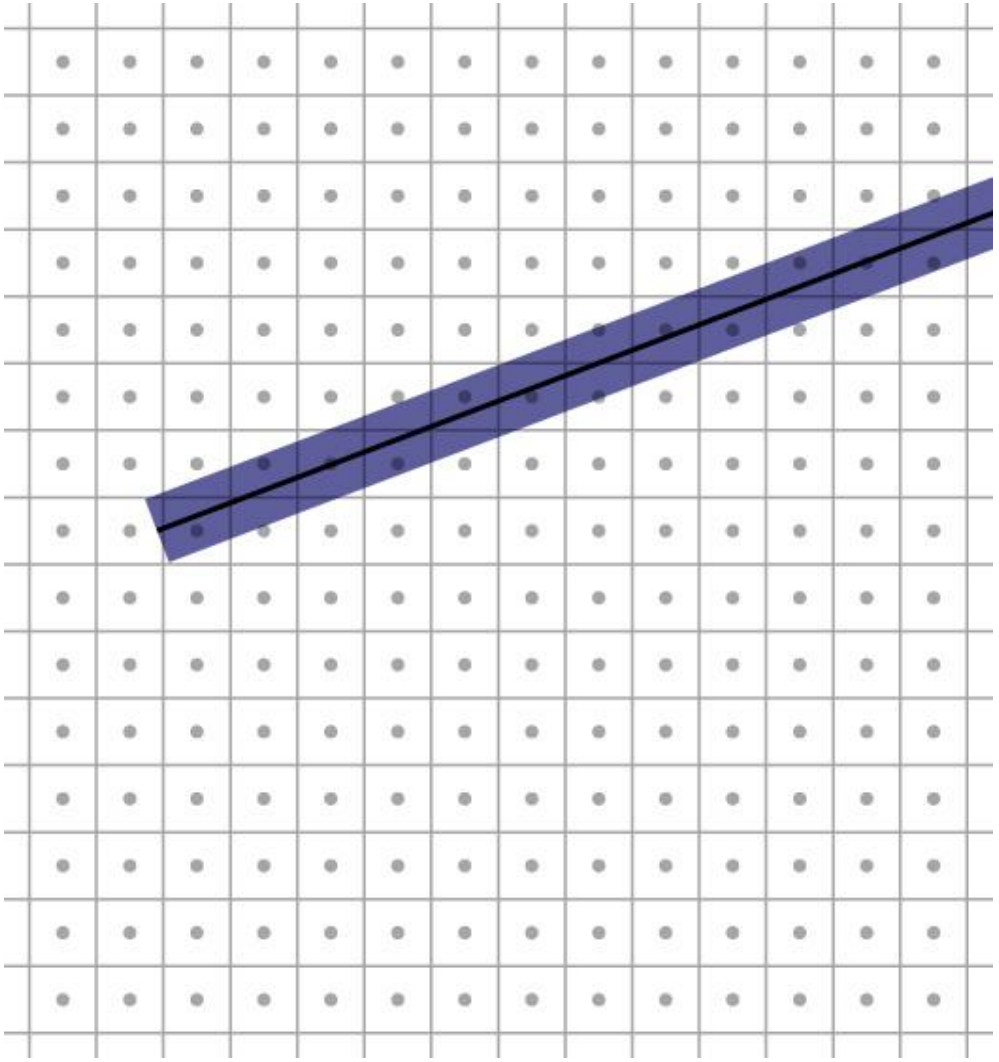
Rasterization

- First job: enumerate the pixels covered by a primitive
  - simple, aliased definition: pixels whose centers fall inside
- Second job: interpolate values across the primitive
  - e.g. colors computed at vertices
  - e.g. normals at vertices
  - will see applications later on



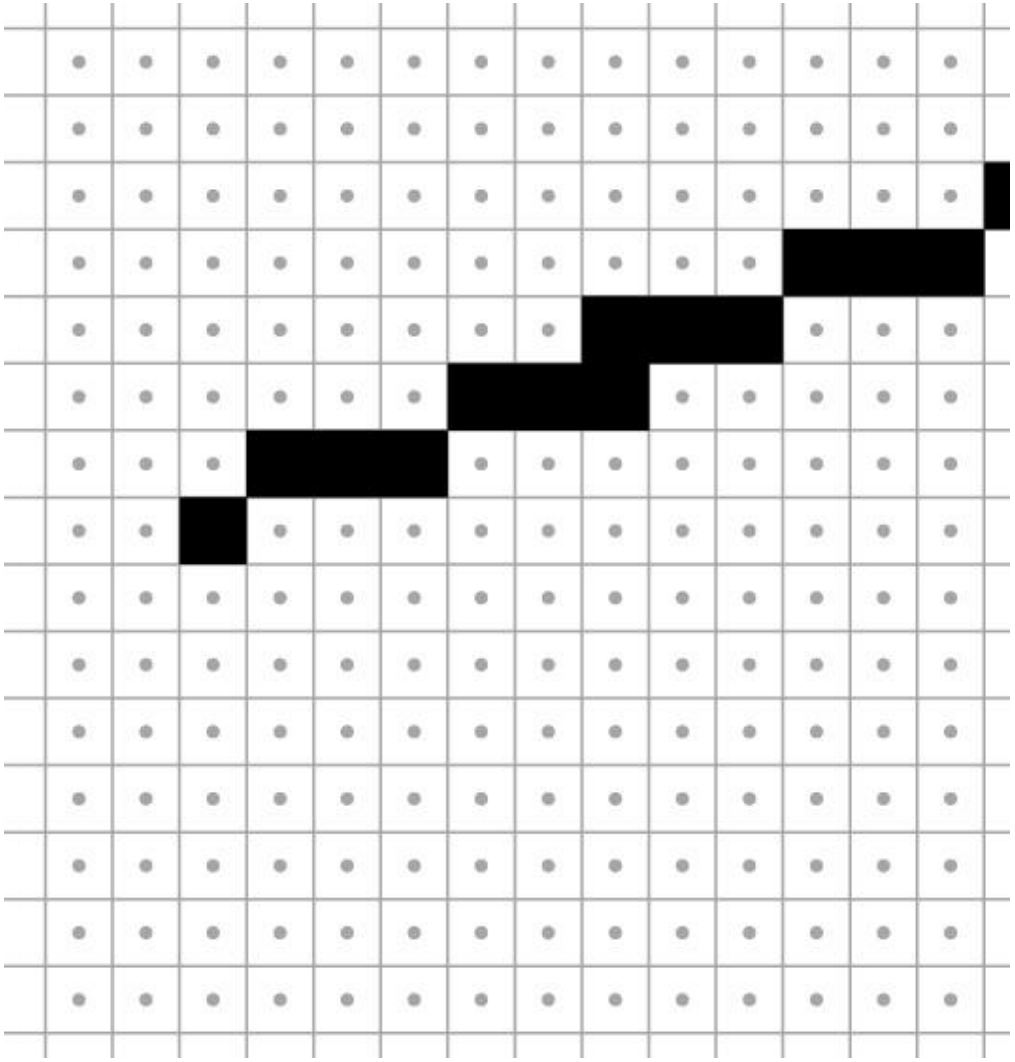
Rasterizing lines

- Define line as a rectangle
- Specify by two endpoints
- Ideal image: black inside, white outside

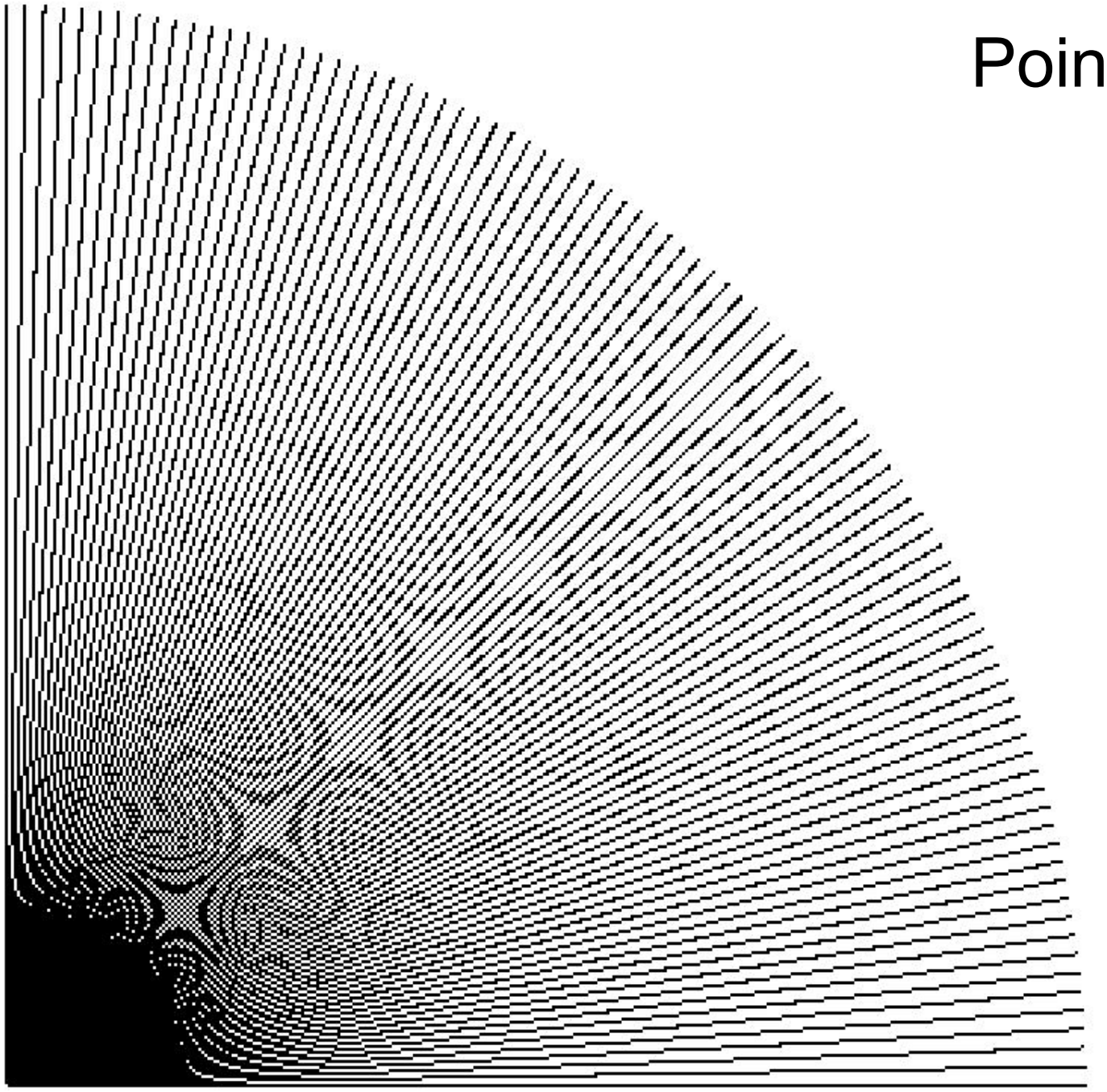


Point sampling

- Approximate rectangle by drawing all pixels whose centers fall within the line
- Problem: sometimes turns on adjacent pixels

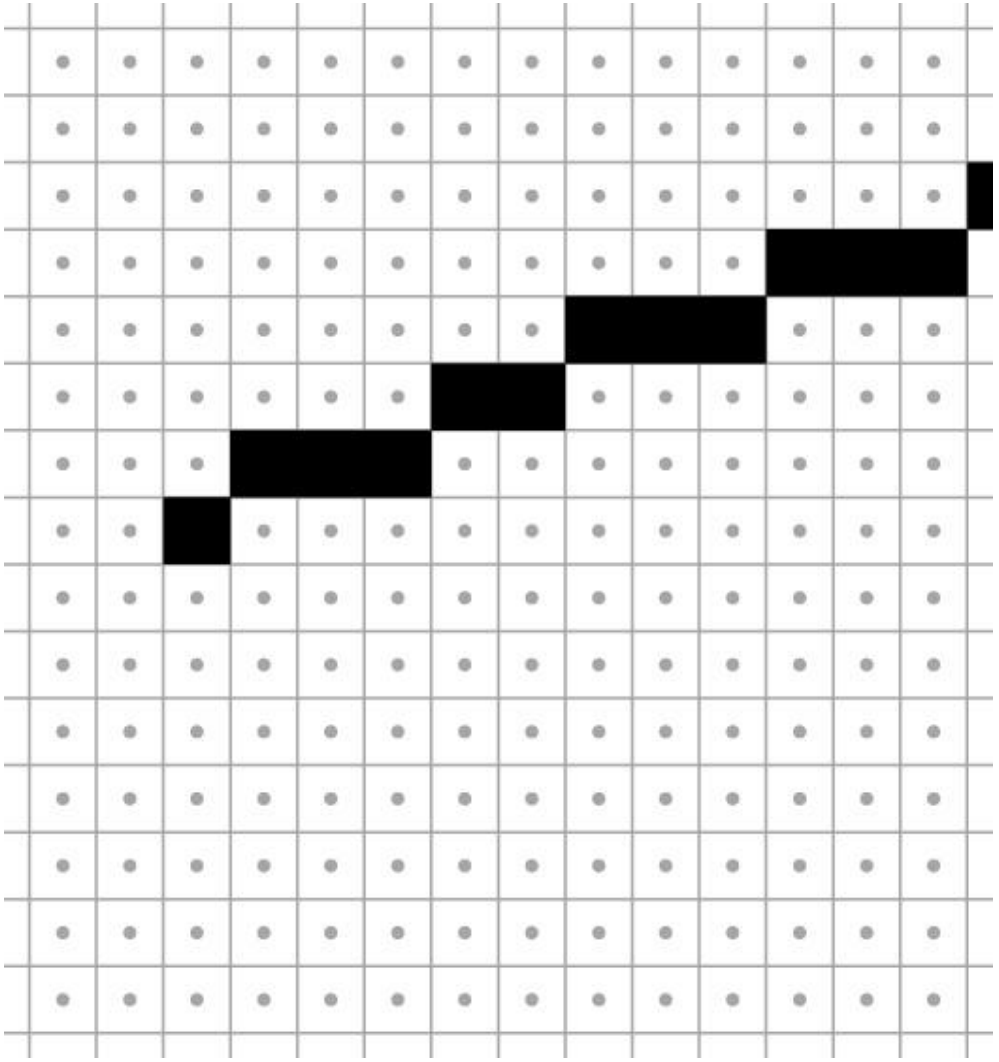


Point sampling in action

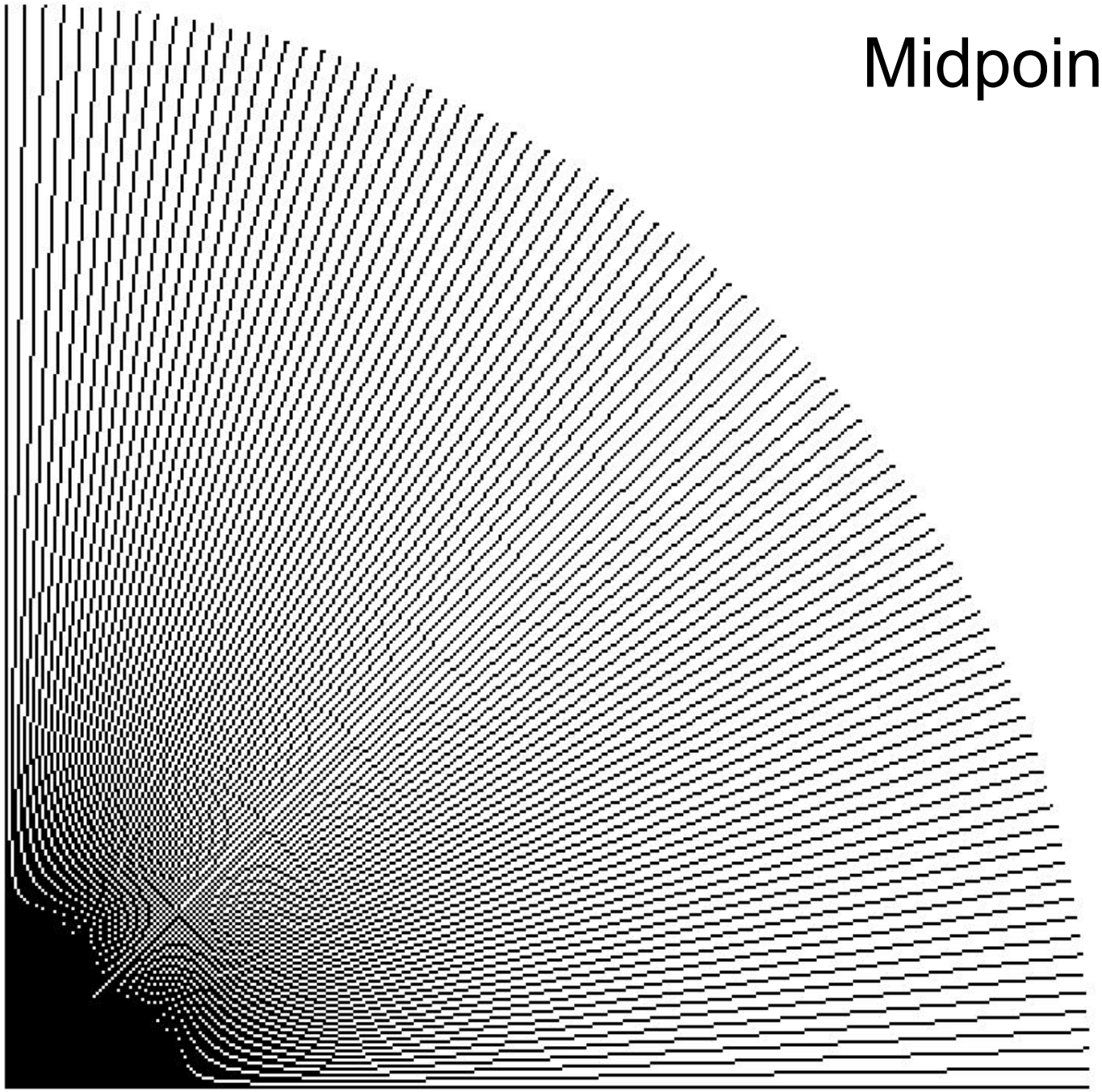


Bresenham lines (midpoint alg.)

- Point sampling unit width rectangle leads to uneven line width
- Define line width parallel to pixel grid
- That is, turn on the single nearest pixel in each column
- Note that 45° lines are now thinner



Midpoint algorithm in action



Algorithms for drawing lines

- line equation:

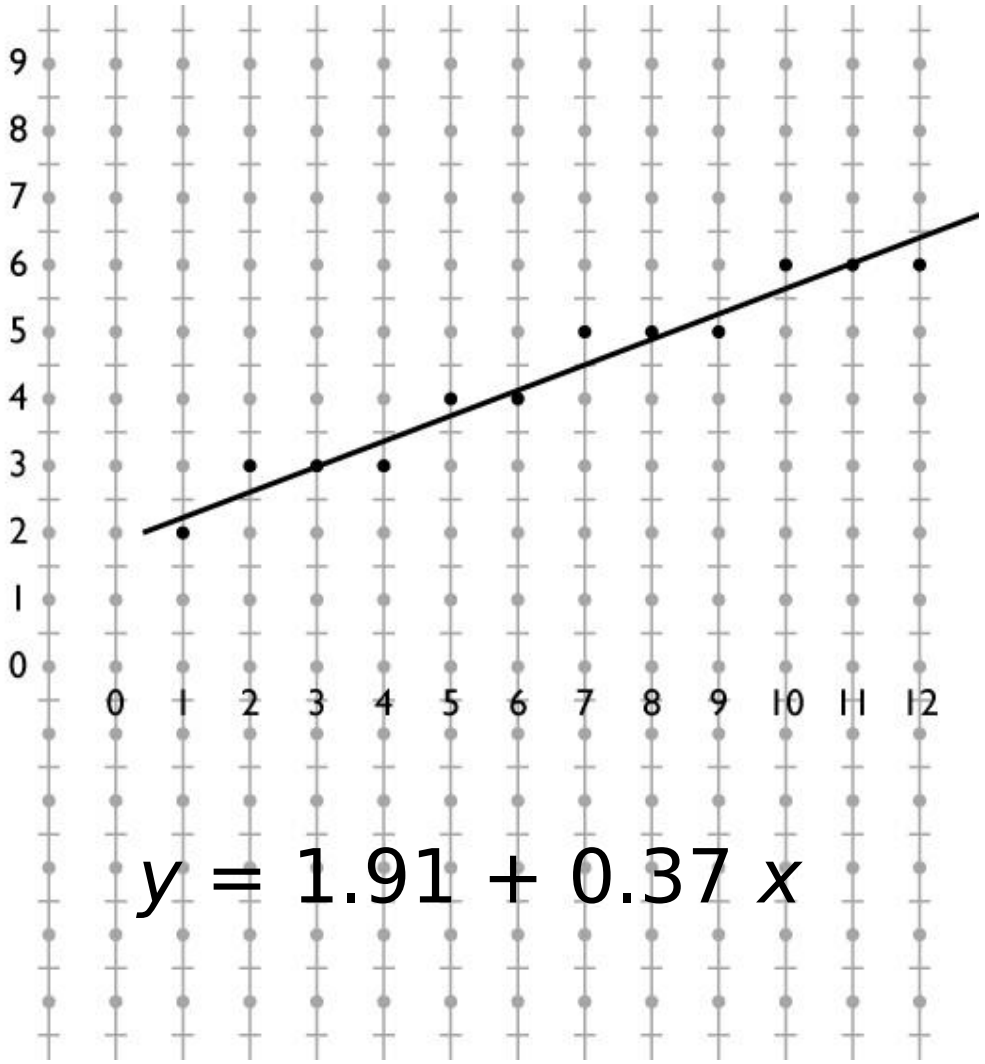
  $y = b + m\,x$

- Simple algorithm: evaluate line equation per column

- W.l.o.g. $x_0 < x_1$;

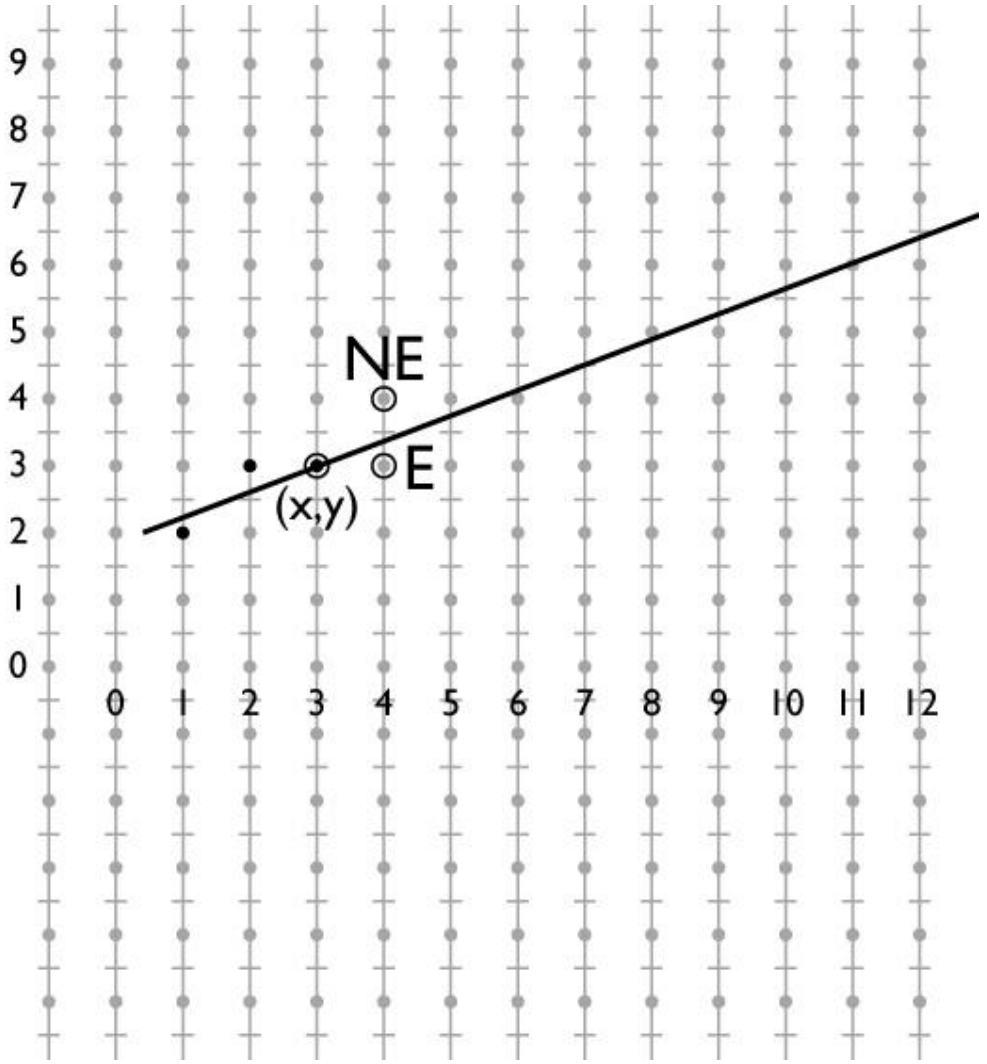
  $0 \le m \le 1$

```
for x = ceil(x0) to floor(x1)
    y = b + m*x
    output(x, round(y))
```



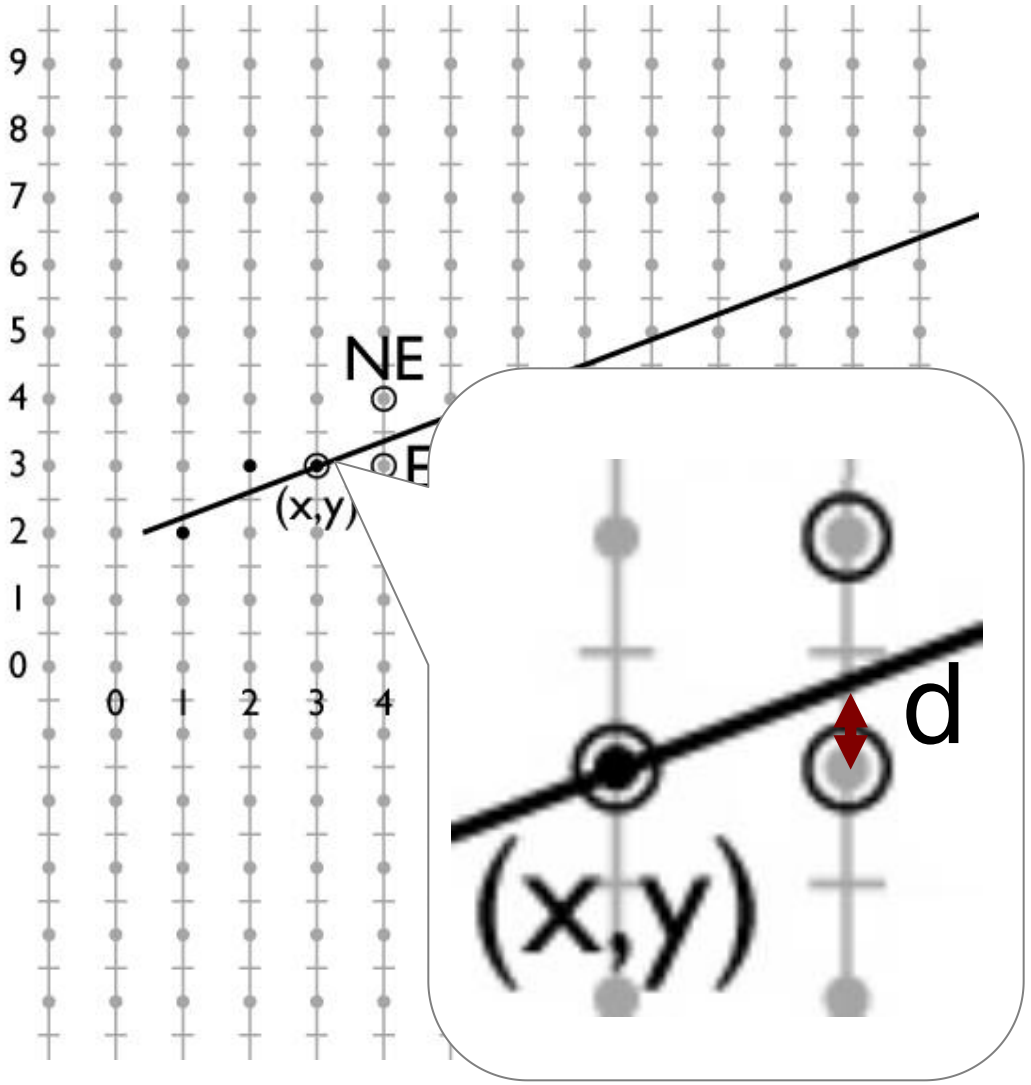
Optimizing line drawing

- Multiplying and rounding is slow
- At each pixel the only options are E and NE
- $d = m(x + 1) + b - y$
- $d > 0.5$ decides between E and NE



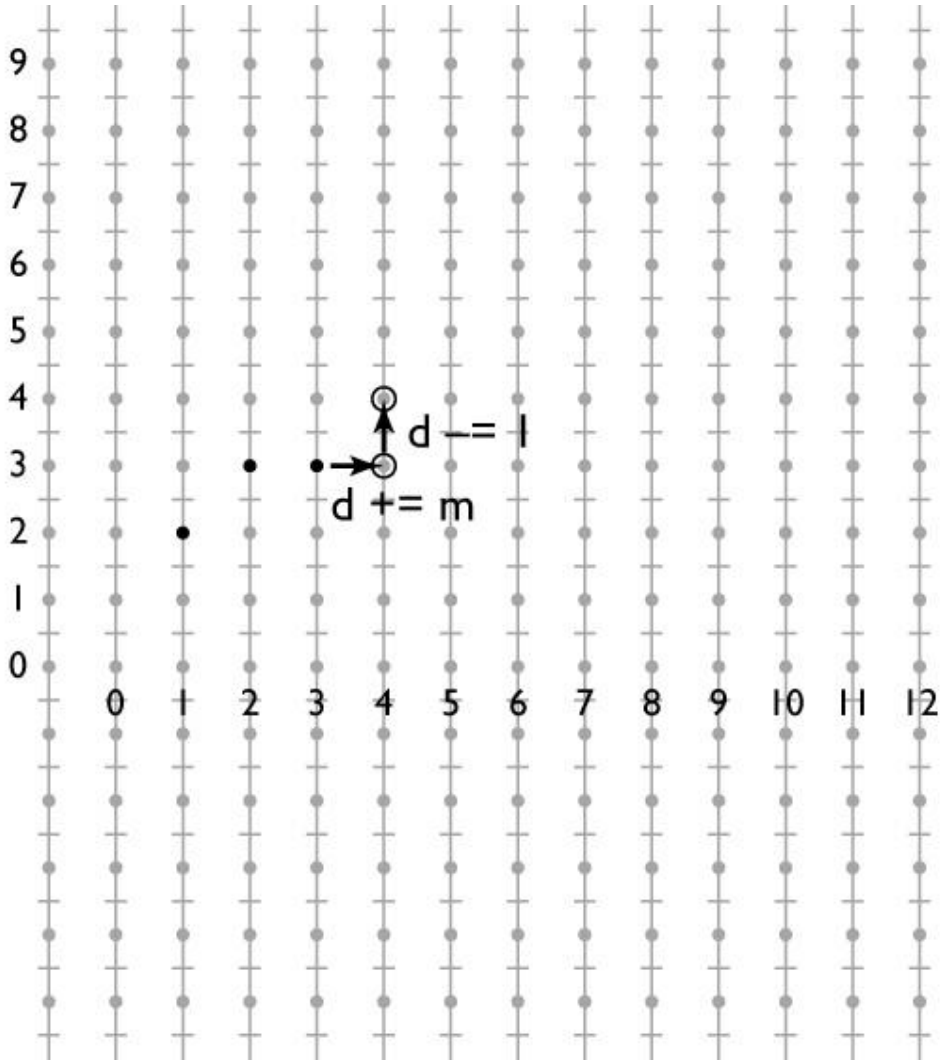
Optimizing line drawing

- Multiplying and rounding is slow
- At each pixel the only options are E and NE
- $d = m(x + 1) + b - y$
- $d > 0.5$ decides between E and NE



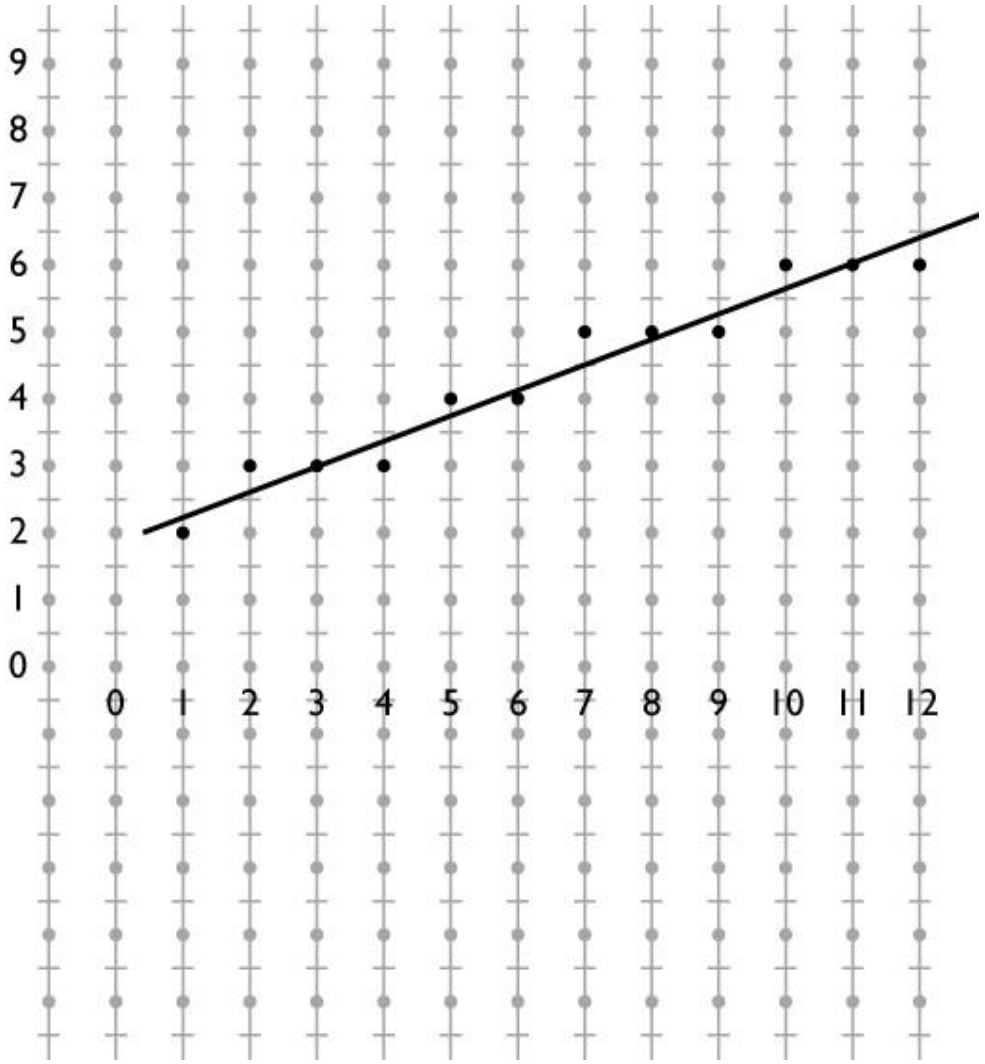
Optimizing line drawing

- $d = m(x + 1) + b - y$
- Only need to update d for integer steps in x and y
- Do that with addition
- Known as “DDA” (digital differential analyzer)



Midpoint line algorithm

```
x = ceil(x0)
y = round(m*x + b)
d = m*(x + 1) + b – y
while x < floor(x1)
    if d ! 0.5
        y += 1
        d –= 1
    x += 1
    d += m
    output(x, y)
```



Rasterizing triangles

- The most common case
  - some systems render a line as two skinny triangles
- Algorithm (point-sampling)
  - walk from pixel to pixel over the triangle's area
  - evaluate linear functions as you go
  - decide which pixels are inside

Rasterizing triangles

- Input:
  - three 2D points (the triangle's vertices in pixel space)
    - (x_0, y_0); (x_1, y_1); (x_2, y_2)
  - parameter values at each vertex
    - $q_{00}, \ldots, q_{0n}$; $q_{10}, \ldots, q_{1n}$; $q_{20}, \ldots, q_{2n}$
- Output: a list of fragments, each with
  - the integer pixel coordinates (x, y)
  - interpolated parameter values $q_0, \ldots, q_n$

Rasterizing triangles

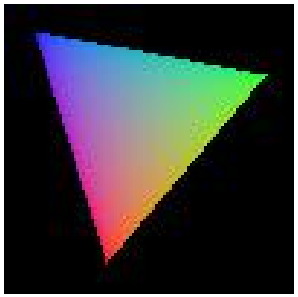
- Inside-outside test
- Linear interpolation of colors, depths, coordinates,...

Linear interpolation

- A linear function on the plane is:

$$q(x, y) = c_x x + c_y y + c_k$$

- To interpolate parameters across a triangle we need to find the c_x, c_y, and c_k that define the (unique) linear function that matches the given values at all 3 vertices



Defining parameter functions

- To interpolate parameters across a triangle we need to find the c_x, c_y, and c_k that define the (unique) linear function that matches the given values at all 3 vertices
  - this is 3 constraints on 3 unknown coefficients:

$$c_x x_0 + c_y y_0 + c_k = q_0$$
$$c_x x_1 + c_y y_1 + c_k = q_1$$
$$c_x x_2 + c_y y_2 + c_k = q_2$$

(each states that the function agrees with the given value at one vertex)

  - leading to a 3x3 matrix equation for the coefficients:

$$\begin{bmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{bmatrix} \begin{bmatrix} c_x \\ c_y \\ c_k \end{bmatrix} = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \end{bmatrix}$$

(singular iff triangle is degenerate)

Defining parameter functions

- More efficient version: shift origin to (x_0, y_0)

$$q(x,y) = c_x(x - x_0) + c_y(y - y_0) + q_0$$
$$q(x_1,y_1) = c_x(x_1 - x_0) + c_y(y_1 - y_0) + q_0 = q_1$$
$$q(x_2,y_2) = c_x(x_2 - x_0) + c_y(y_2 - y_0) + q_0 = q_2$$

  - now this is a $2\text{x}2$ linear system (since q_0 falls out):

$$\begin{bmatrix}(x_1 - x_0) & (y_1 - y_0)\\ (x_2 - x_0) & (y_2 - y_0)\end{bmatrix}\begin{bmatrix}c_x\\ c_y\end{bmatrix} = \begin{bmatrix}q_1 - q_0\\ q_2 - q_0\end{bmatrix}$$

  - solve using Cramer's rule (see Shirley):

$$c_x = (\Delta q_1 \Delta y_2 - \Delta q_2 \Delta y_1)/(\Delta x_1 \Delta y_2 - \Delta x_2 \Delta y_1)$$
$$c_y = (\Delta q_2 \Delta x_1 - \Delta q_1 \Delta x_2)/(\Delta x_1 \Delta y_2 - \Delta x_2 \Delta y_1)$$

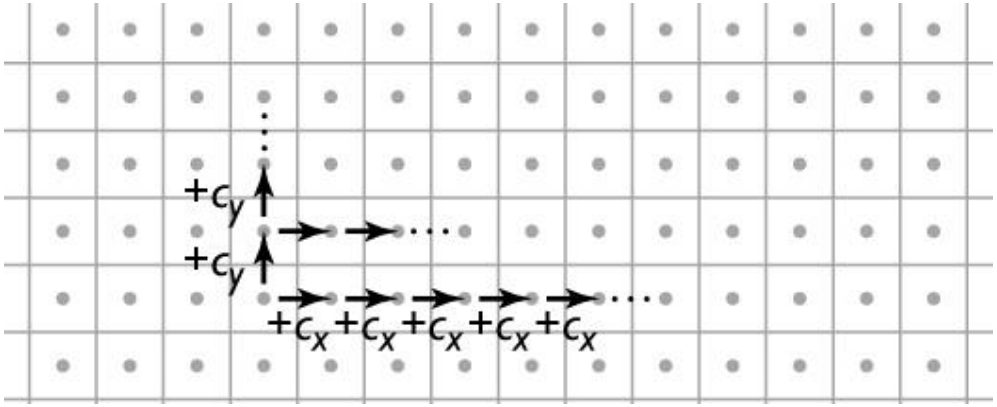
Obtain interpolated values simply by function evaluations

- A linear function on the plane is:

$$q(x, y) = c_x x + c_y y + c_k$$

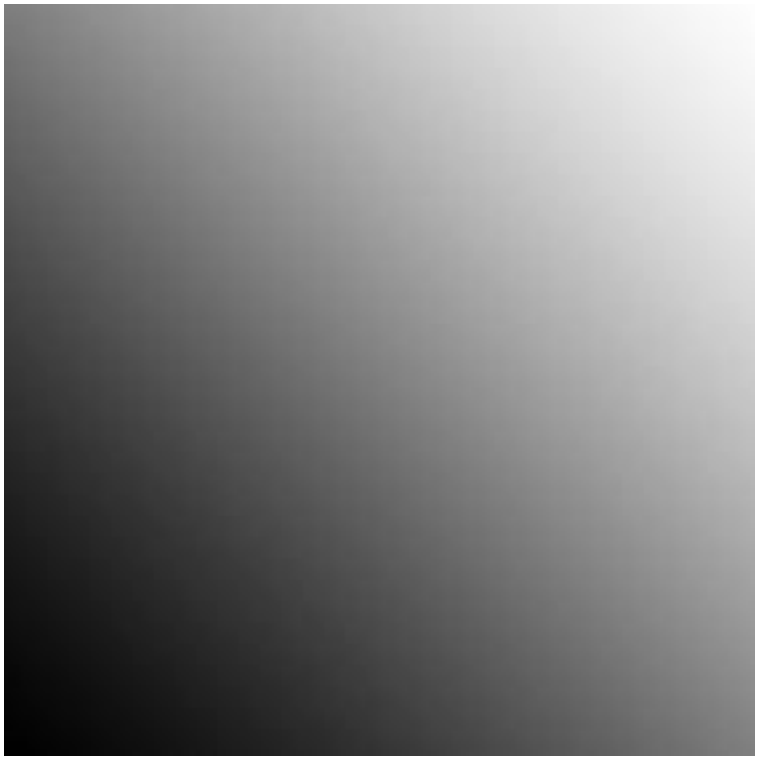
- Linear functions are efficient to evaluate on a grid (incremental linear evaluation):

$$q(x+1, y) = c_x(x+1) + c_y y + c_k = q(x, y) + c_x$$
$$q(x, y+1) = c_x x + c_y(y+1) + c_k = q(x, y) + c_y$$



Incremental linear evaluation

```
linEval(xl, xh, yl, yh, cx, cy, ck) {

    // setup
    qRow = cx*xl + cy*yl + ck;

    // traversal
    for y = yl to yh {
        qPix = qRow;
        for x = xl to xh {
            output(x, y, qPix);
            qPix += cx;
        }
        qRow += cy;
    }
}
```

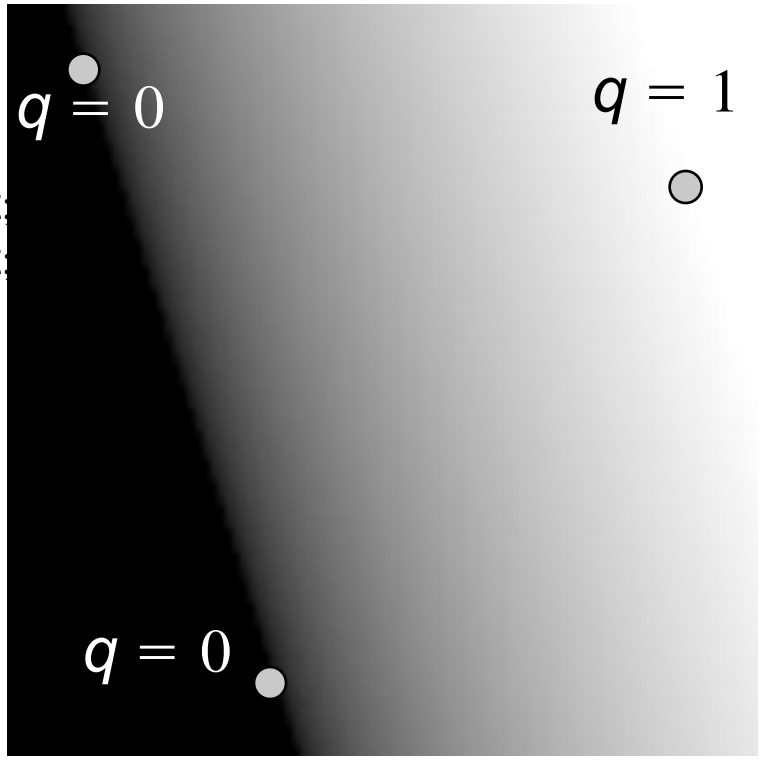


$c_x = .005$; $c_y = .005$; $c_k = 0$

(image size 100x100)

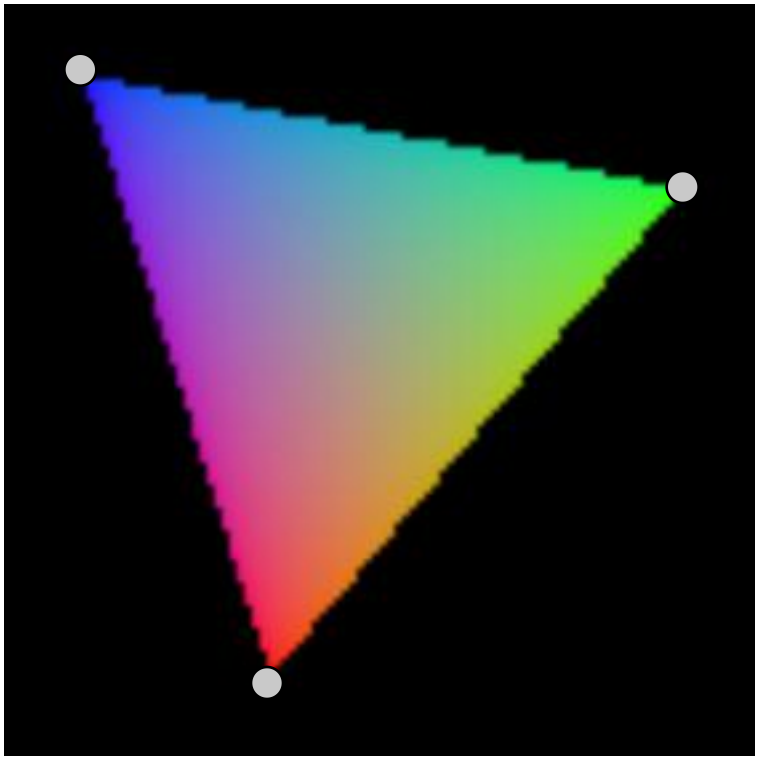
Defining parameter functions

```
linInterp(xl, xh, yl, yh, x0, y0, q0,
          x1, y1, q1, x2, y2, q2) {

    // setup
    det = (x1-x0)*(y2-y0) - (x2-x0)*(y1-y0);
    cx = ((q1-q0)*(y2-y0) - (q2-q0)*(y1-y0)) / det;
    cy = ((q2-q0)*(x1-x0) - (q1-q0)*(x2-x0)) / det;
    qRow = cx*(xl-x0) + cy*(yl-y0) + q0;

    // traversal (same as before)
    for y = yl to yh {
        qPix = qRow;
        for x = xl to xh {
            output(x, y, qPix);
            qPix += cx;
        }
        qRow += cy;
    }
}
```

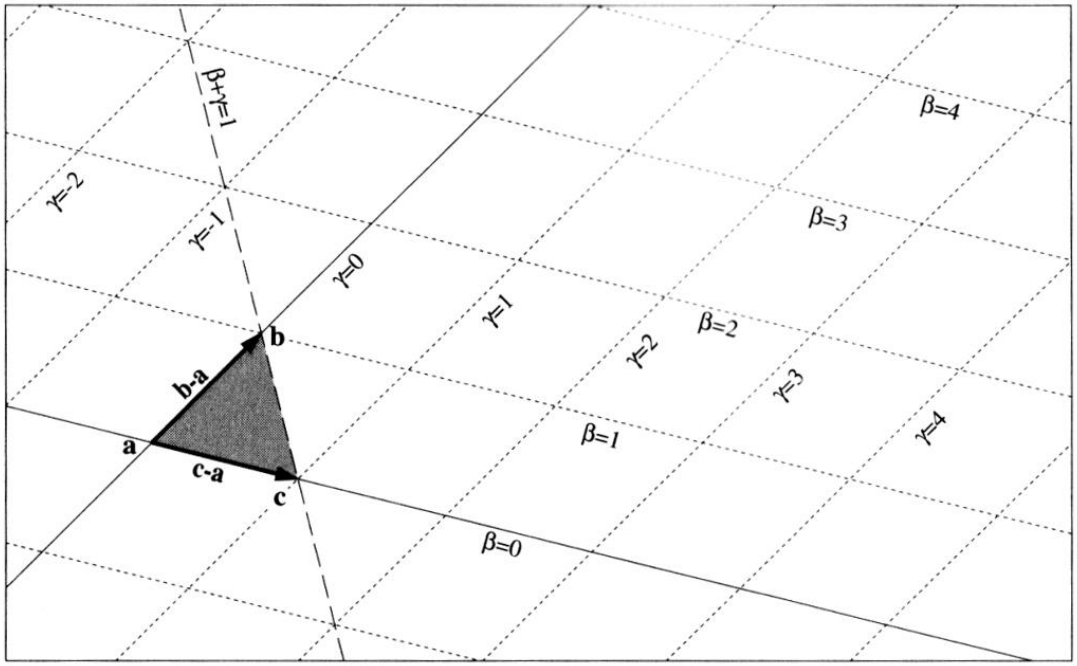


Clipping to the triangle

- Interpolate three *barycentric coordinates* across the plane
  - each barycentric coord is 1 at one vert. and 0 at the other two
- Output fragments only when all three are ≥ 0.



Barycentric coordinates



– linear viewpoint: basis of edges

$$\alpha = 1 - \beta - \gamma$$

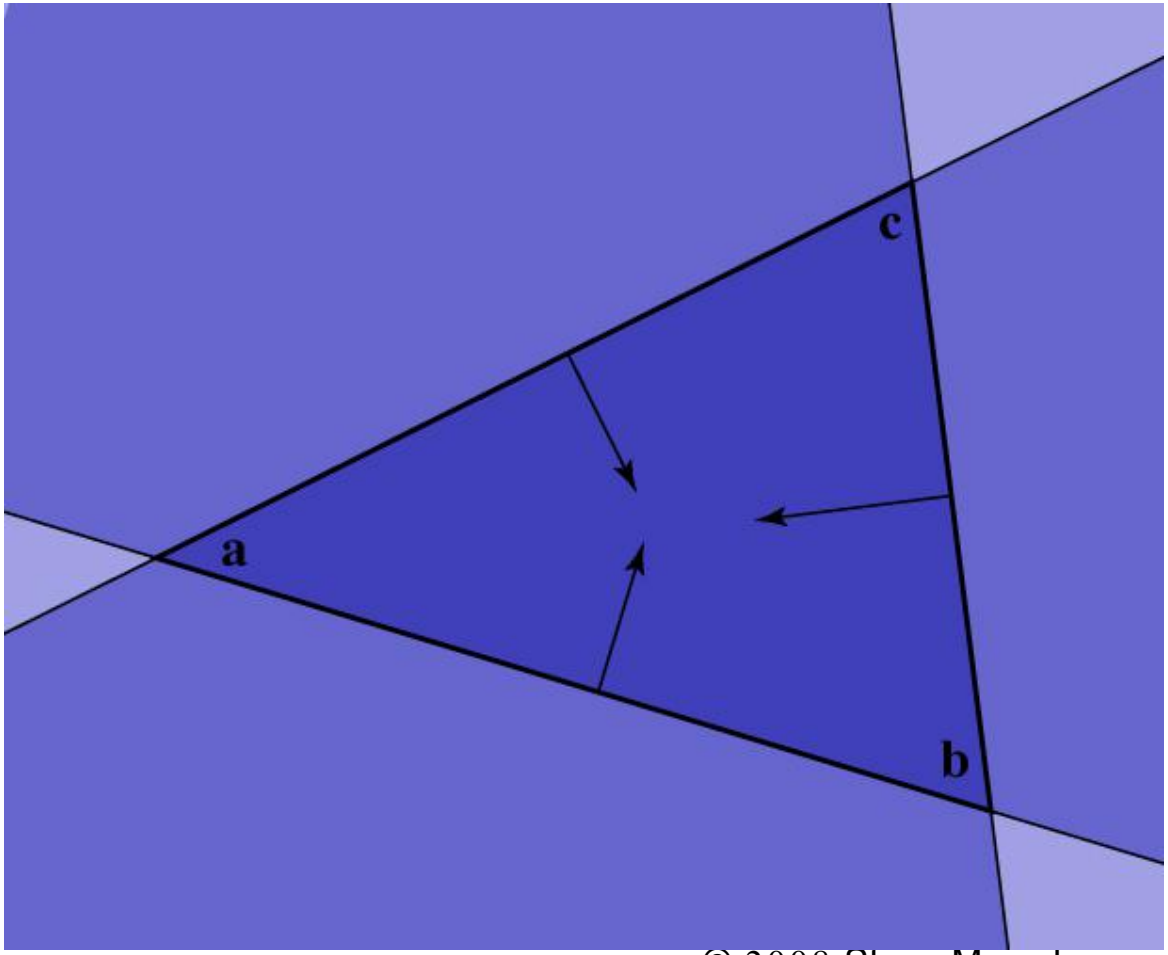
$$\mathbf{p} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a})$$

$$\beta > 0; \quad \gamma > 0; \quad \beta + \gamma < 1$$

Edge equations

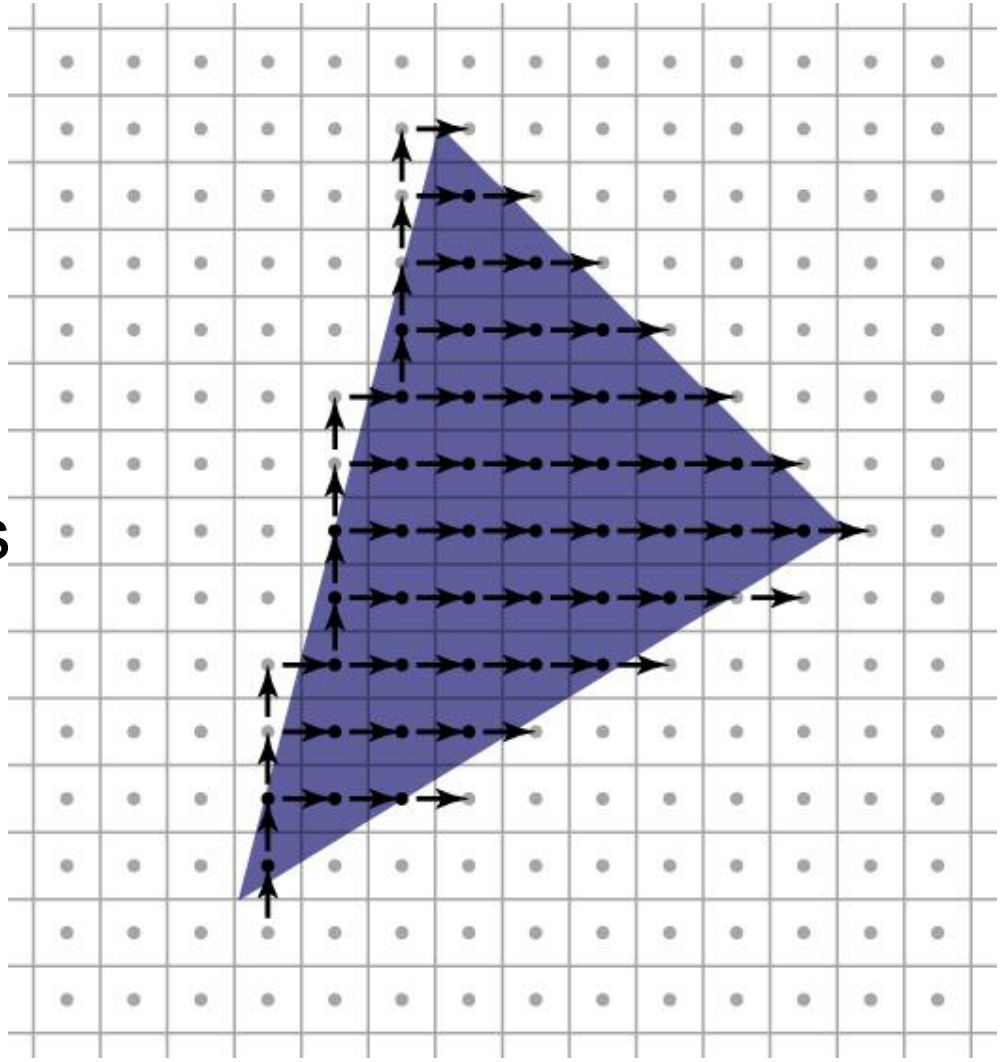
- In plane, triangle is the intersection of 3 half spaces

$$(\mathbf{x}-\mathbf{a})\cdot(\mathbf{b}-\mathbf{a})^{\perp} > 0$$
$$(\mathbf{x}-\mathbf{b})\cdot(\mathbf{c}-\mathbf{b})^{\perp} > 0$$
$$(\mathbf{x}-\mathbf{c})\cdot(\mathbf{a}-\mathbf{c})^{\perp} > 0$$



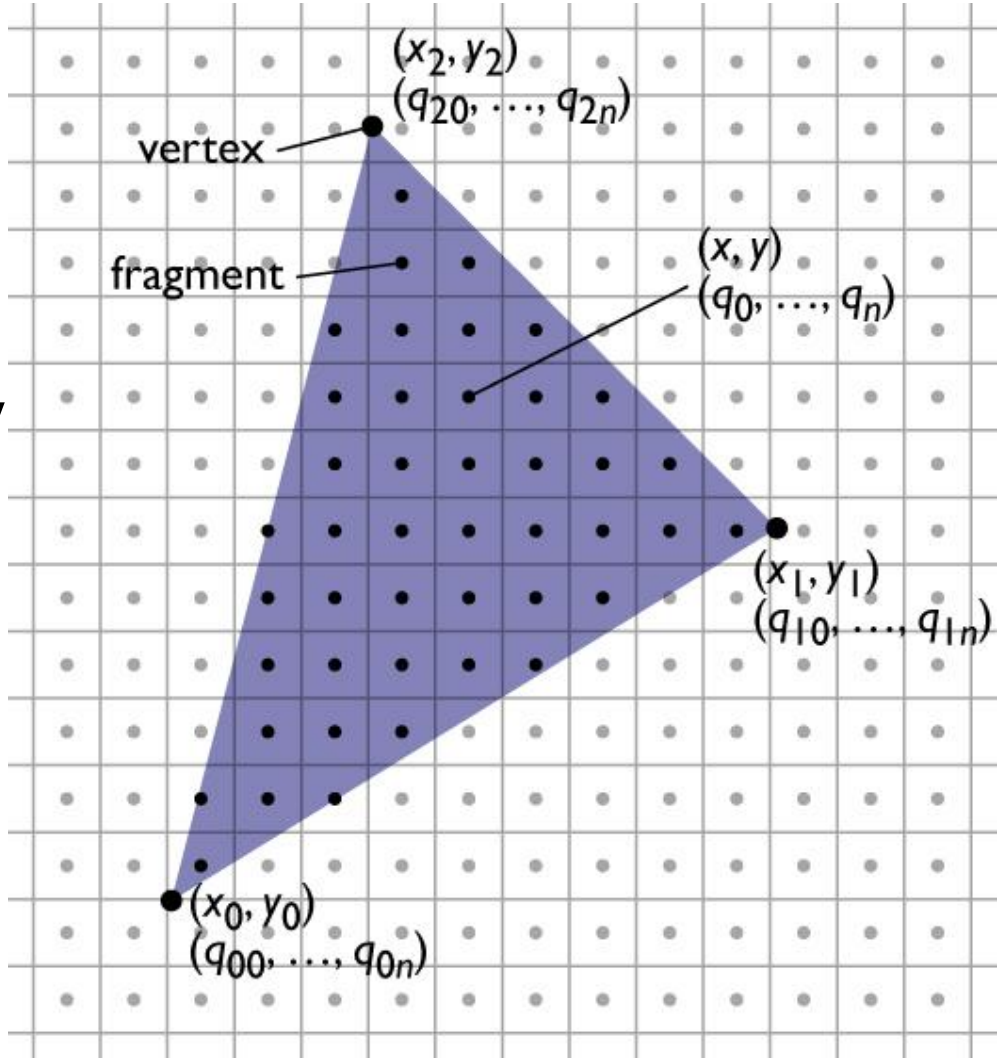
Pixel-walk (Pineda) rasterization

- Conservatively visit a superset of the pixels you want
- Interpolate linear functions
- Use those functions to determine when to emit a fragment

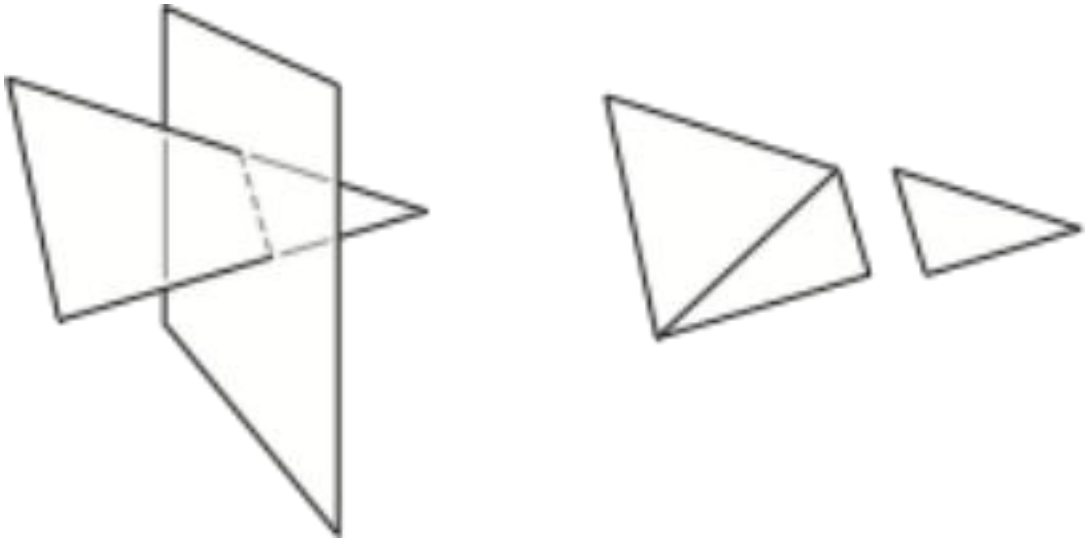


Rasterizing triangles

- Summary
  1. evaluation of linear functions on pixel grid
  2. functions defined by parameter values at vertices
  3. using extra parameters to determine fragment set



Clipping



- Rasterizer tends to assume triangles are on screen
  - particularly problematic to have triangles crossing the plane $z = 0$
- After projection, before perspective divide
  - clip against the planes $x, y, z = 1, -1$ (6 planes)
  - primitive operation: clip triangle against axis-aligned plane

Clipping a triangle against a plane

- 4 cases, based on sidedness of vertices
  - all in (keep)
  - all out (discard)
  - one in, two out (one clipped triangle)
  - two in, one out (two clipped triangles)