

CSC 433/533

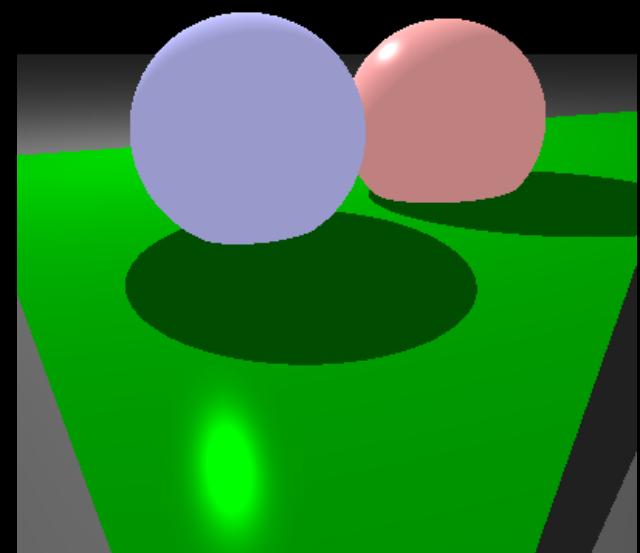
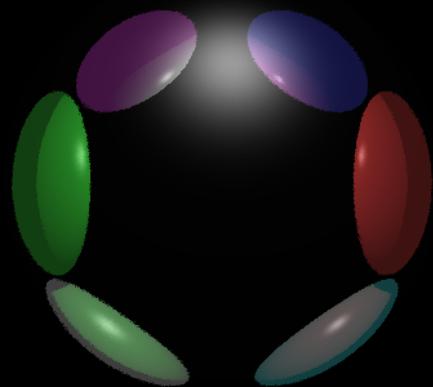
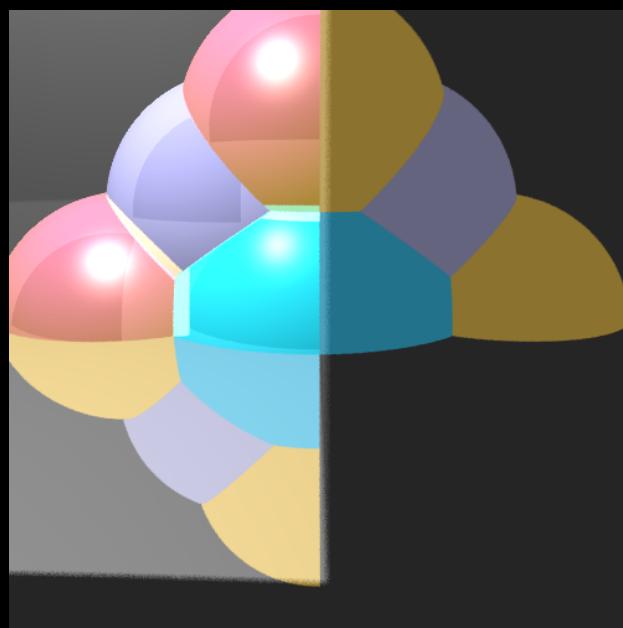
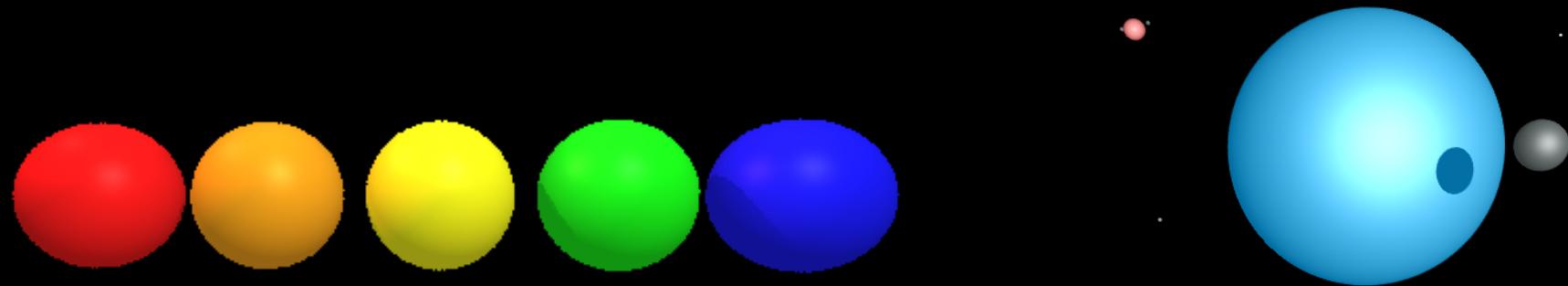
Computer Graphics

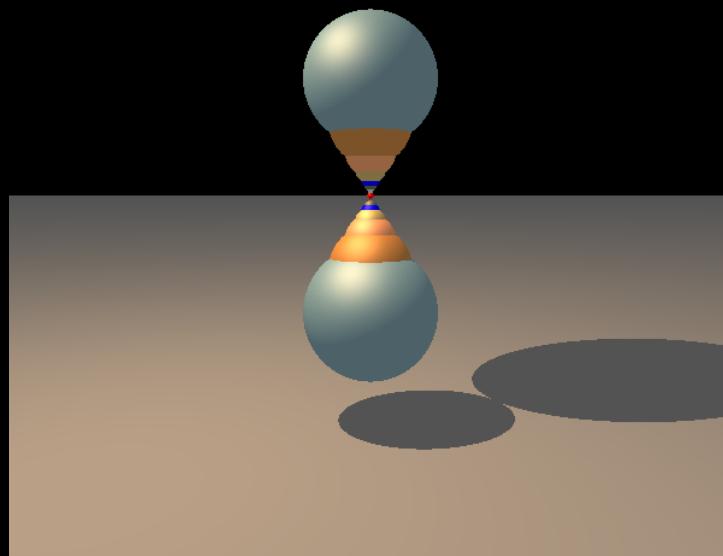
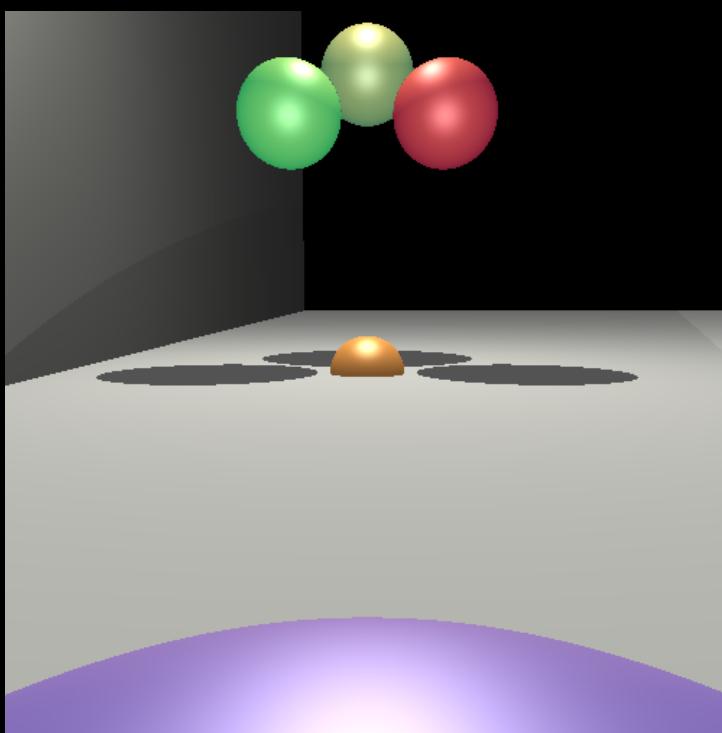
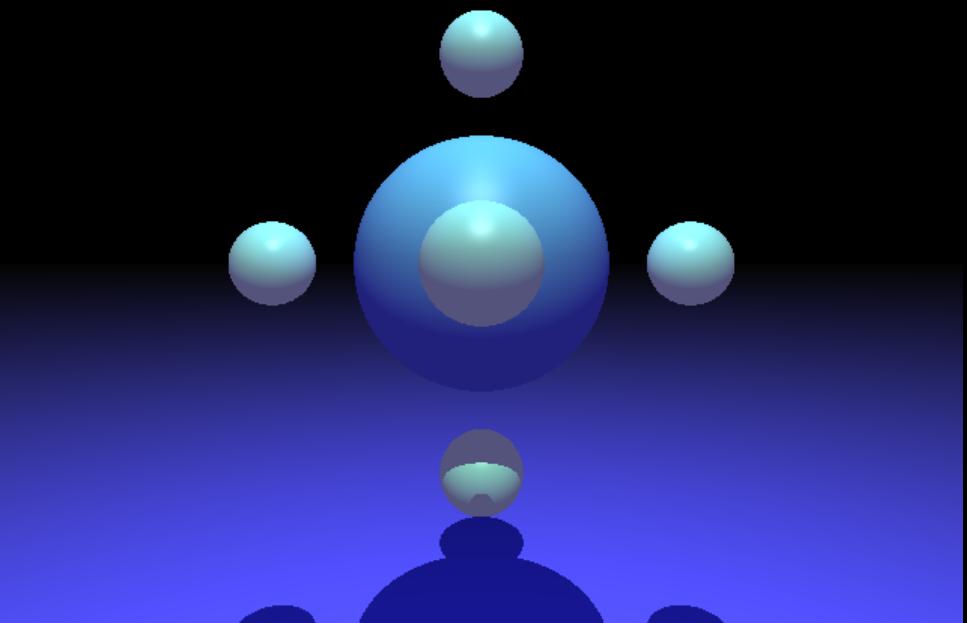
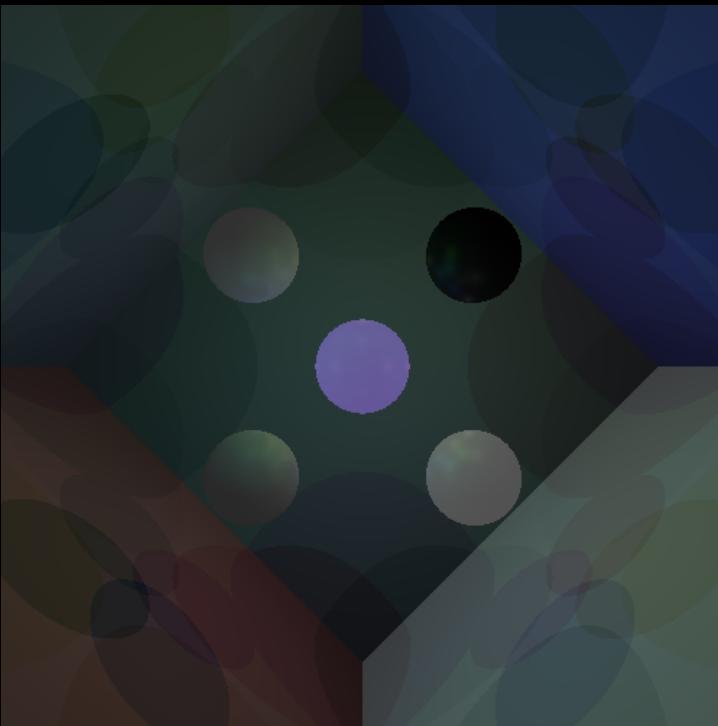
Alon Efrat

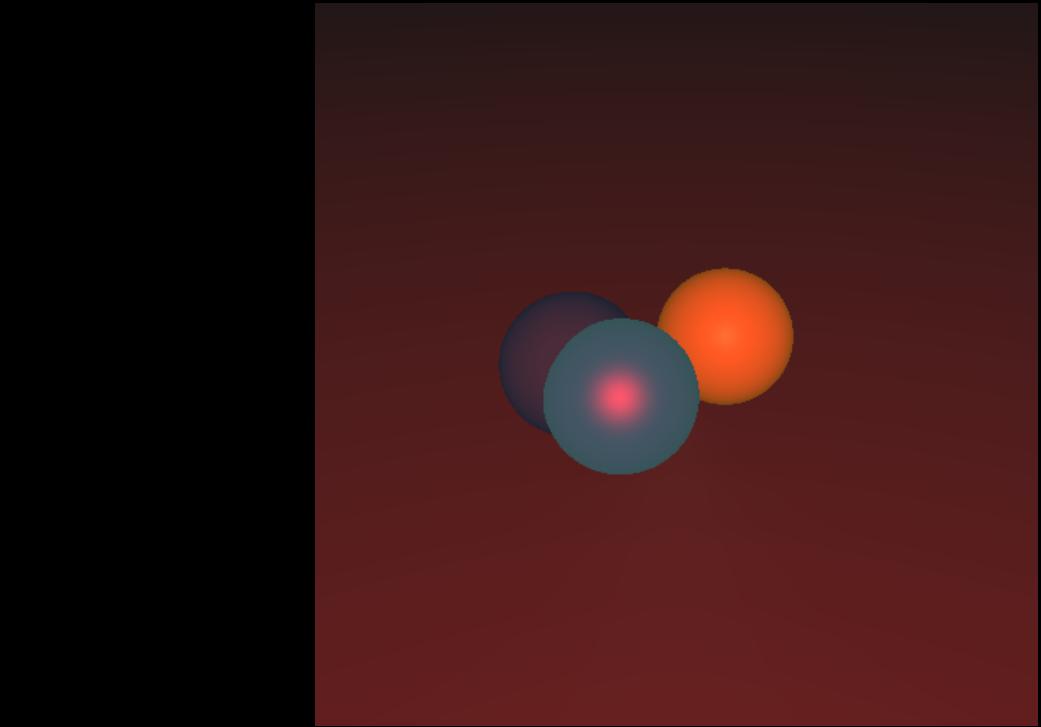
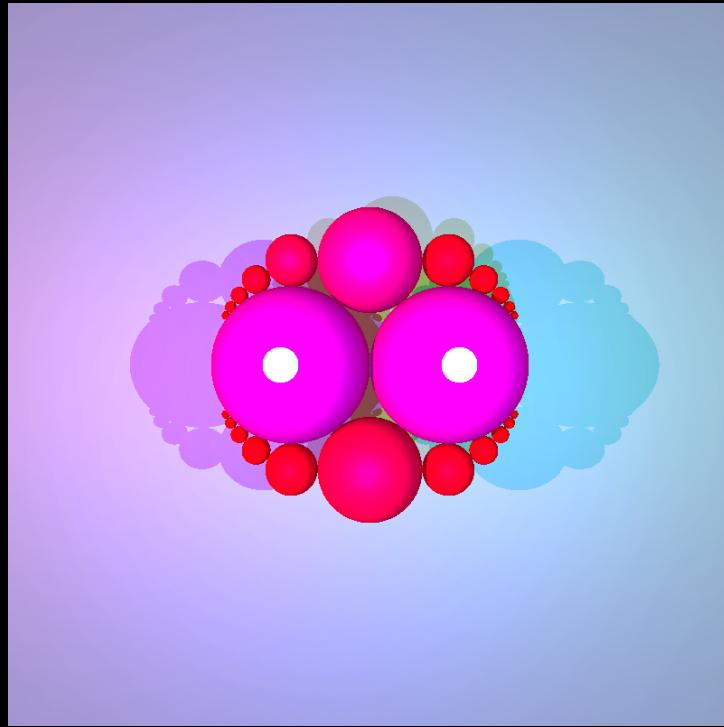
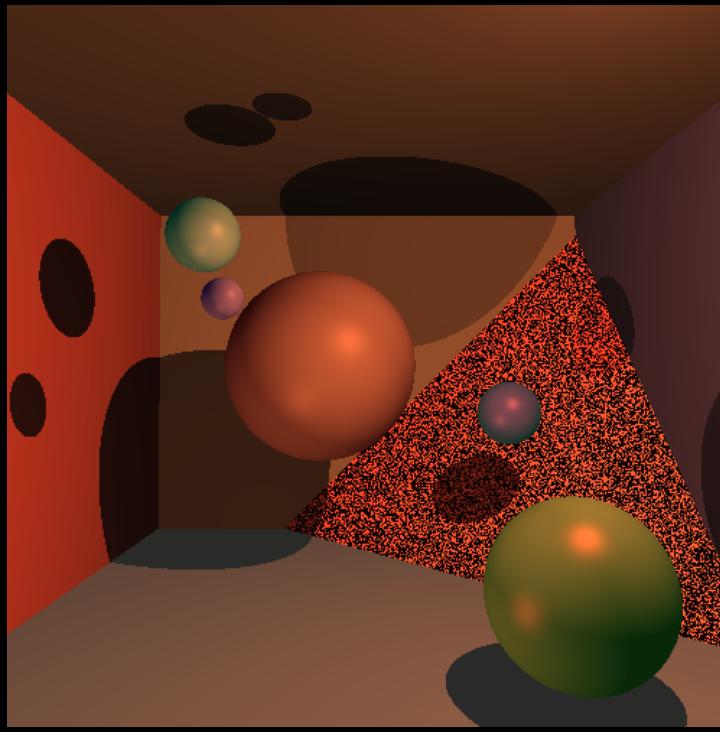
Credit: Joshua Levine

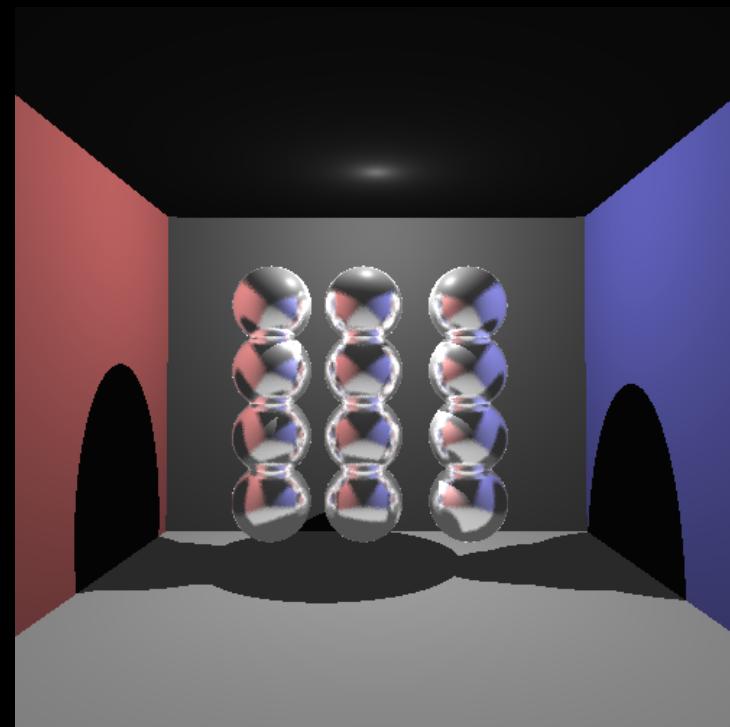
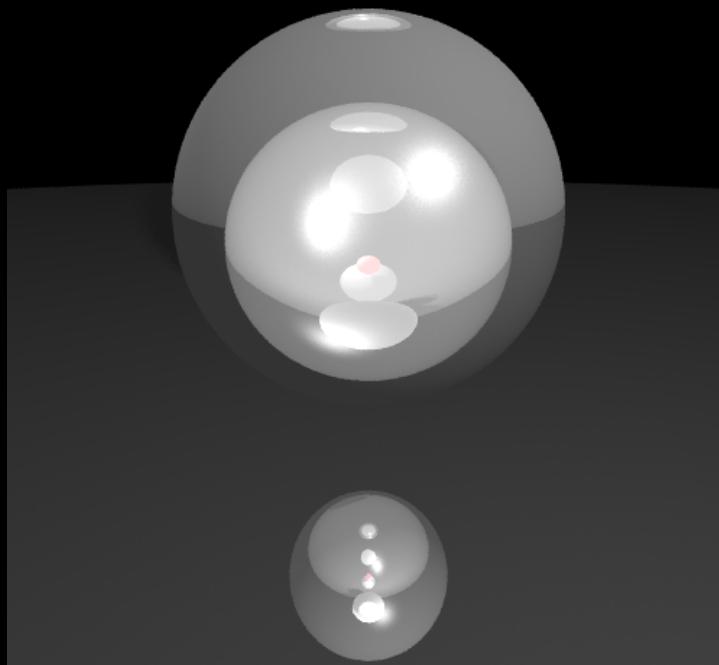
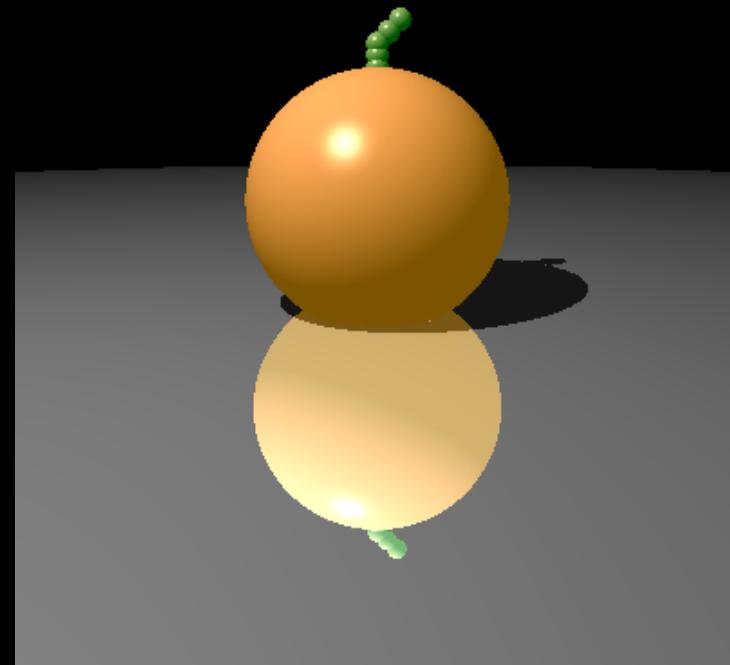
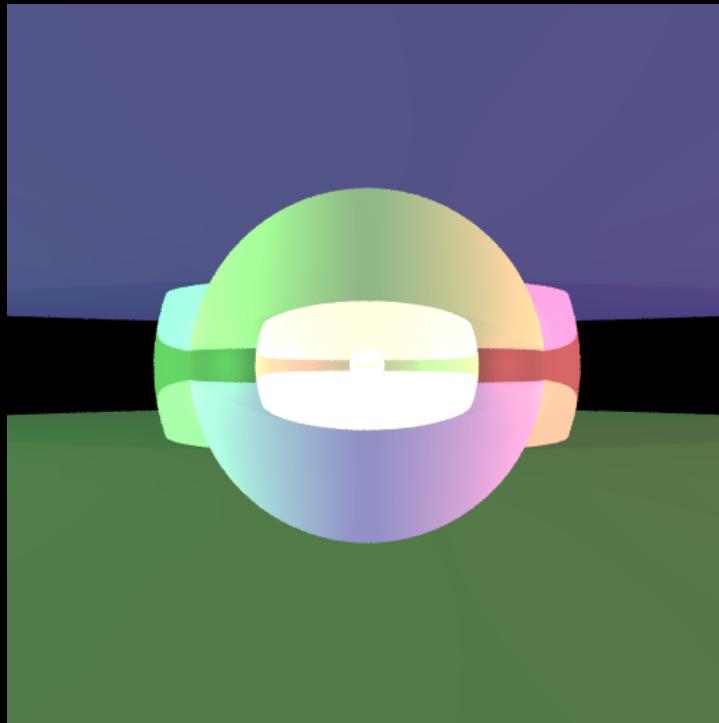
November 10

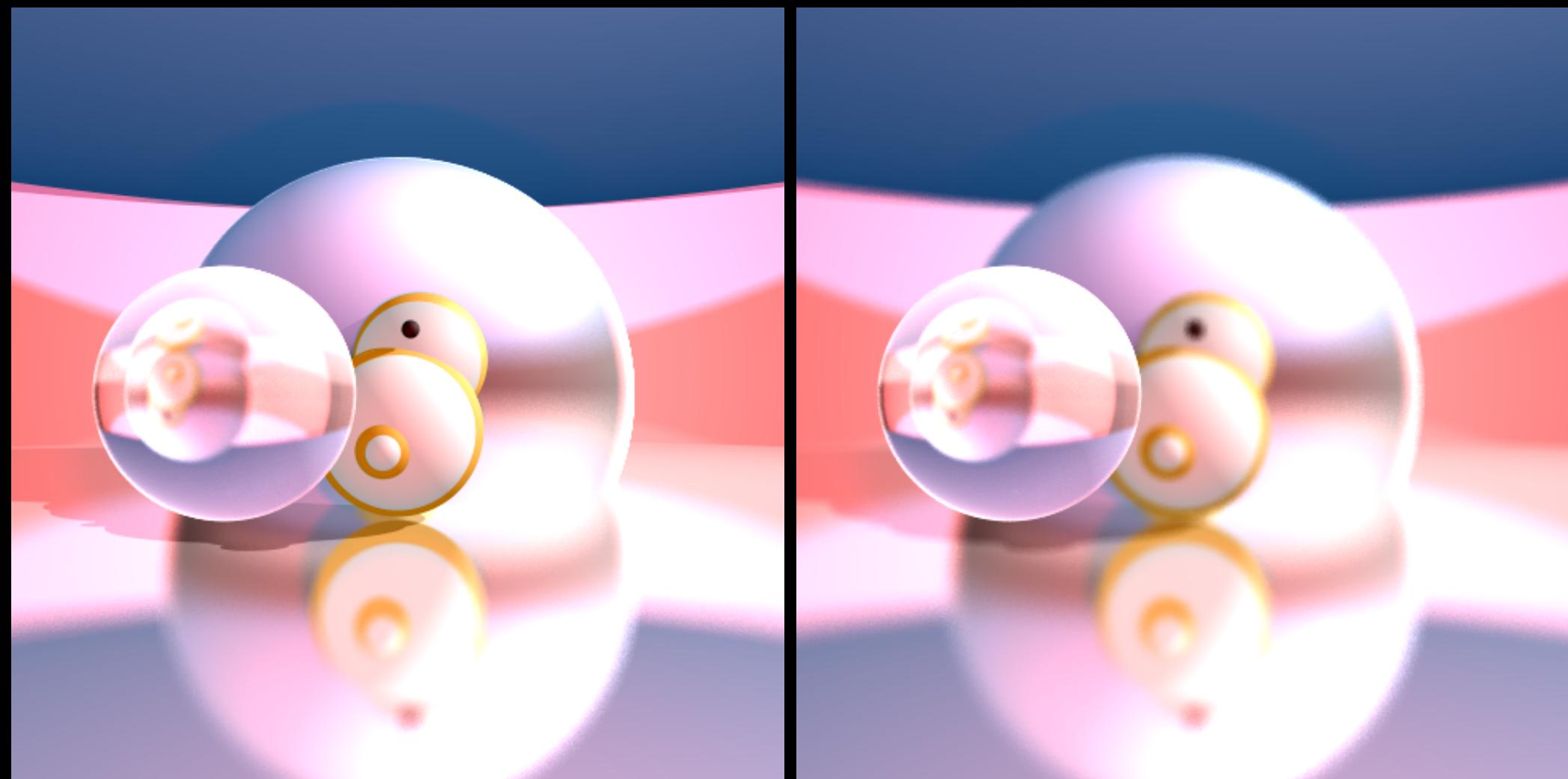
Today: More transformations and viewing

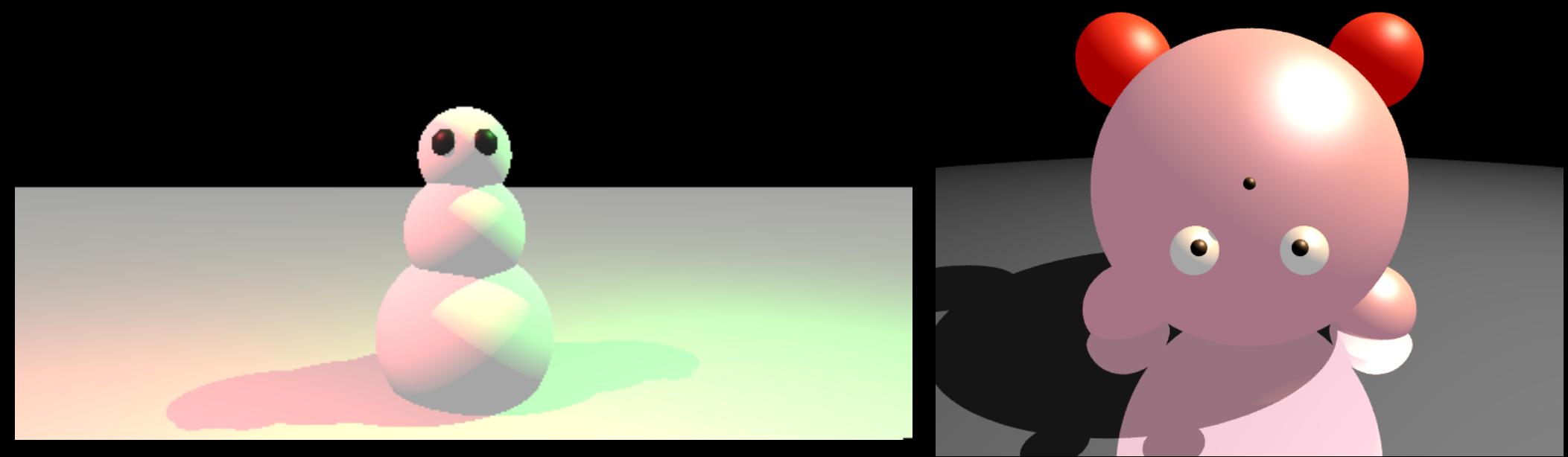
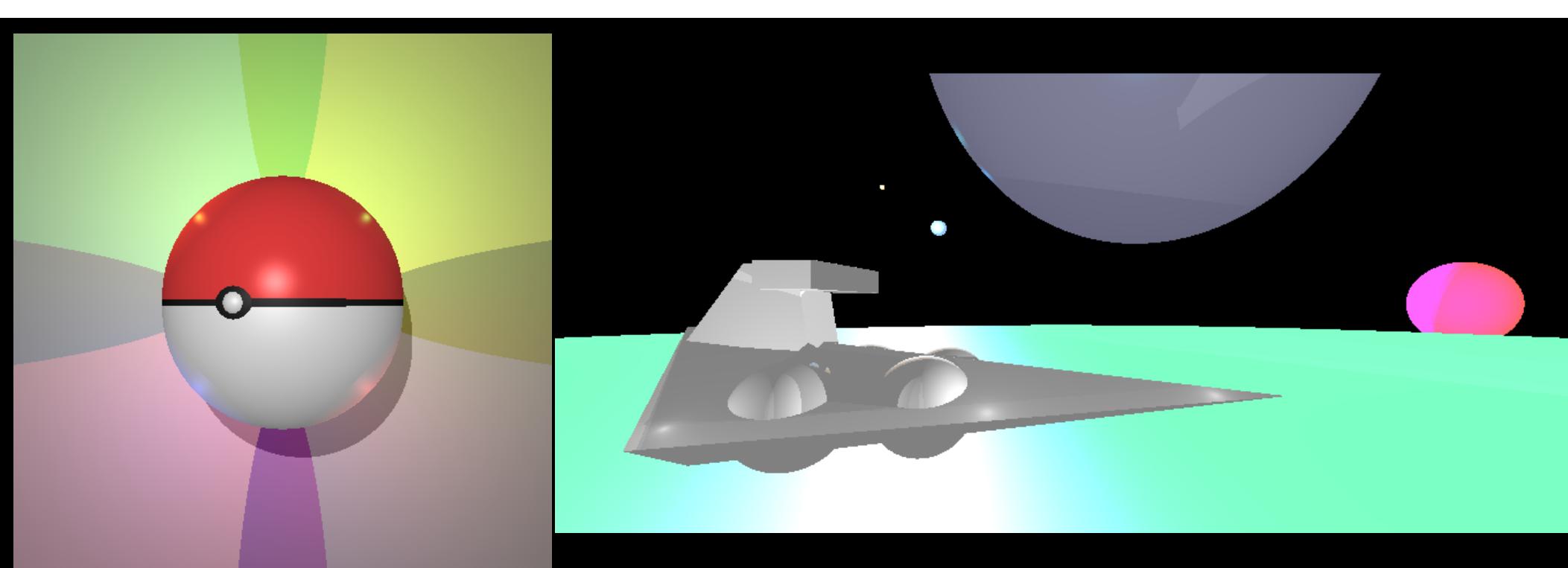












3D Linear Transformations

- We can transform points in a 3D coordinate system by multiplying the point (a vector) by a matrix (the transformation), just like in 2D!
- The only difference is we will use 3x3 matrices A by $\mathbf{x} = (x, y, z)$, or $A\mathbf{x}$, e.g. for scale and shear:

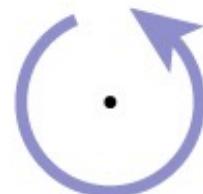
$$\text{scale}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}$$

$$\text{shear-x}(d_y, d_z) = \begin{bmatrix} 1 & d_y & d_z \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

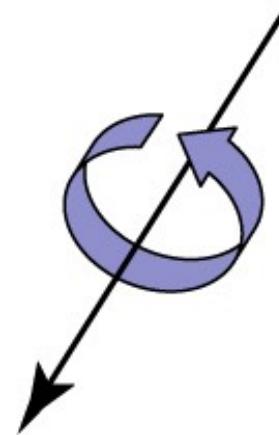
Rotations in 3D

- In 2D, a rotation is about a point
- In 3D, a rotation is about an axis

convention: positive rotation is CCW



2D



3D

convention: positive rotation is CCW when axis vector is pointing at you

Rotations about 3D Axes

- In 3D, we need to pick an axis to rotate about

$$\text{rotate-z}(\phi) = \begin{bmatrix} \cos \phi & -\sin \phi & 0 \\ \sin \phi & \cos \phi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

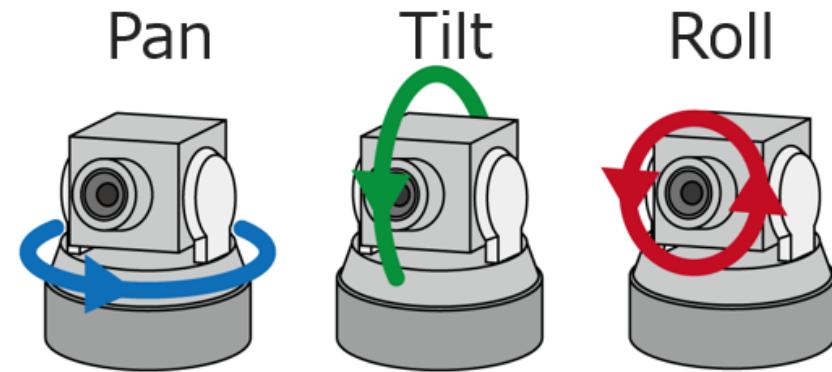
- And we can pick any of the three axes

$$\text{rotate-x}(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \phi & -\sin \phi \\ 0 & \sin \phi & \cos \phi \end{bmatrix}$$

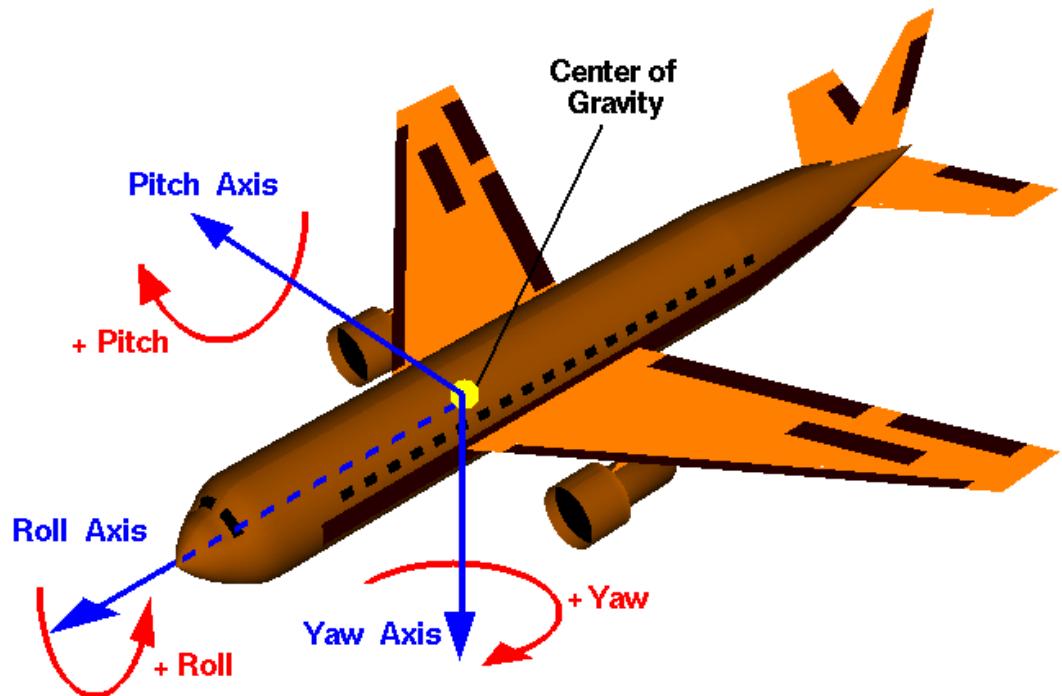
$$\text{rotate-y}(\phi) = \begin{bmatrix} \cos \phi & 0 & \sin \phi \\ 0 & 1 & 0 \\ -\sin \phi & 0 & \cos \phi \end{bmatrix}$$

Building Complex Rotations from Axis-Aligned Rotations

- Rotations about **x**, **y**, **z** are sometimes called **Euler angles**
- Build a combined rotation using matrix composition



Ishikawa Watanabe Laboratory



Wikipedia

Arbitrary Rotations

- To rotate about any axis: we change the coordinate space we are working in, using orthogonal matrices.
- Consider orthogonal matrix R_{uvw} , form by taking three orthogonal vectors \mathbf{u} , \mathbf{v} , and \mathbf{w} :

Property of orthogonal vectors:

$$\mathbf{u} \cdot \mathbf{u} = \mathbf{v} \cdot \mathbf{v} = \mathbf{w} \cdot \mathbf{w} = 1$$

$$\mathbf{u} \cdot \mathbf{v} = \mathbf{v} \cdot \mathbf{w} = \mathbf{w} \cdot \mathbf{u} = 0$$

$$R_{uvw} = \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \\ \mathbf{w} \end{bmatrix}$$

Arbitrary Rotations

- What happens when we apply R_{uvw} to any of the basis vectors, e.g.:

$$R_{uvw} \mathbf{u} = \begin{bmatrix} \mathbf{u} \cdot \mathbf{u} \\ \mathbf{v} \cdot \mathbf{u} \\ \mathbf{w} \cdot \mathbf{u} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \mathbf{x}$$

- But this means that if we apply R_{uvw}^T to the Cartesian coordinate vectors, e.g.:

$$R_{uvw}^T \mathbf{y} = \begin{bmatrix} x_u & x_v & x_w \\ y_u & y_v & y_w \\ z_u & z_v & z_w \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} x_v \\ y_v \\ z_v \end{bmatrix} = \mathbf{v}$$

Arbitrary Rotations

- This means that if we want to rotation around an arbitrary axis, we need only to use a change of coordinates
- E.g. to rotate around a direction w , we
 - Compute orthogonal directions u , v , and w . All unit vectors.
 - Change the uvw axes to be xyz (R_{uvw})
 - Apply a rotate-z()
 - Finally, change the axes back to uvw (R_{uvw}^T)

$$\begin{bmatrix} x_u & x_v & x_w \\ y_u & y_v & y_w \\ z_u & z_v & z_w \end{bmatrix} \begin{bmatrix} \cos \phi & -\sin \phi & 0 \\ \sin \phi & \cos \phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_u & y_u & z_u \\ x_v & y_v & z_v \\ x_w & y_w & z_w \end{bmatrix}$$

R_{uvw}^T rotate-z() R_{uvw}

Notation: x_u is the x-coordinate of the vector u

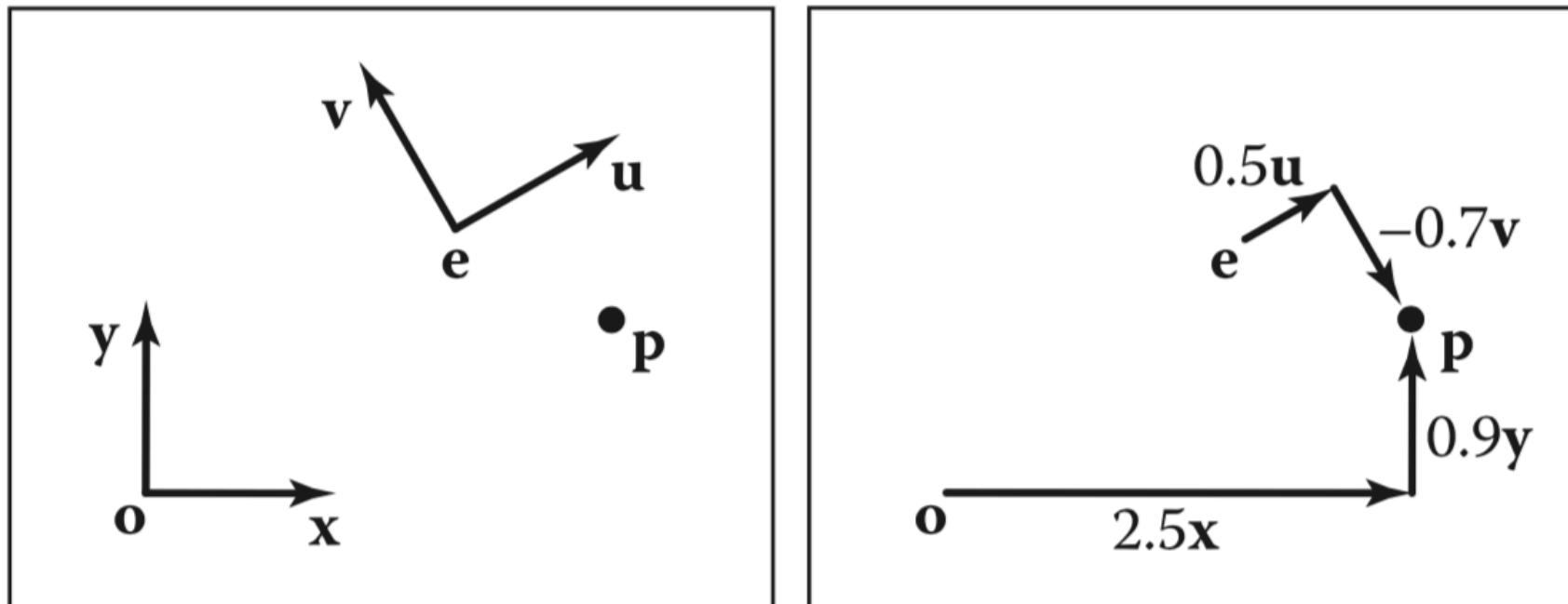
Coordinate Transformations

Coordinate Systems

- Points in space can be represented using an origin position and a set of orthogonal basis vectors:

$$\mathbf{p} = (x_p, y_p) \equiv \mathbf{0} + x_p \mathbf{x} + y_p \mathbf{y} \quad \mathbf{p} = (u_p, v_p) \equiv \mathbf{e} + u_p \mathbf{u} + v_p \mathbf{v}$$

- Any point can be described in either coordinate system



Matrices for Converting Coordinate Systems

- (Remember: Rotating the world CCW is equivalent to rotating the coordinate system CW)
- Using homogenous coordinates and affine transformations, we can convert between coordinate systems:

$$\begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_e \\ 0 & 1 & y_e \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_u & x_v & 0 \\ y_u & y_v & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_p \\ v_p \\ 1 \end{bmatrix} = \begin{bmatrix} x_u & x_v & x_e \\ y_u & y_v & y_e \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_p \\ v_p \\ 1 \end{bmatrix}$$

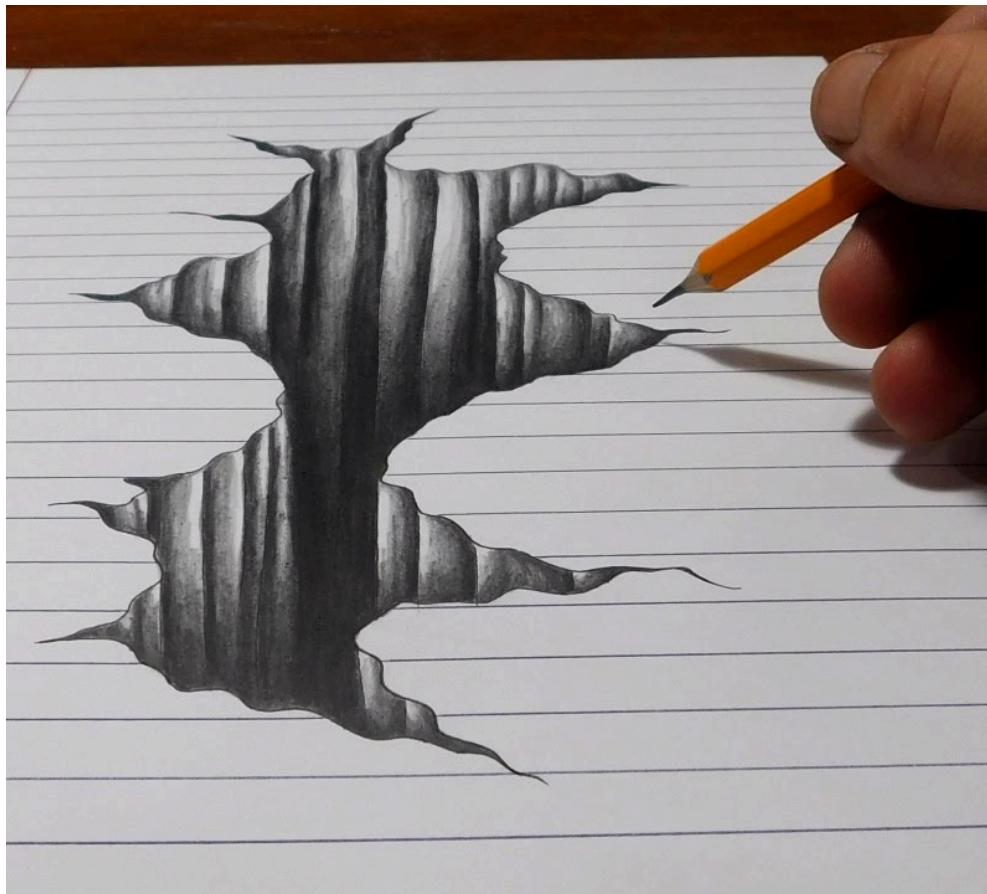
- More generally, any arbitrary coordinate system transform:

$$\mathbf{P}_{uv} = \begin{bmatrix} \mathbf{x}_{uv} & \mathbf{y}_{uv} & \mathbf{o}_{uv} \\ 0 & 0 & 1 \end{bmatrix} \mathbf{p}_{xy}$$

Viewing

Recall: Two Ways to Think About How We Make Images

- Drawing
- Photography



Recall: Two Ways to Think About Rendering

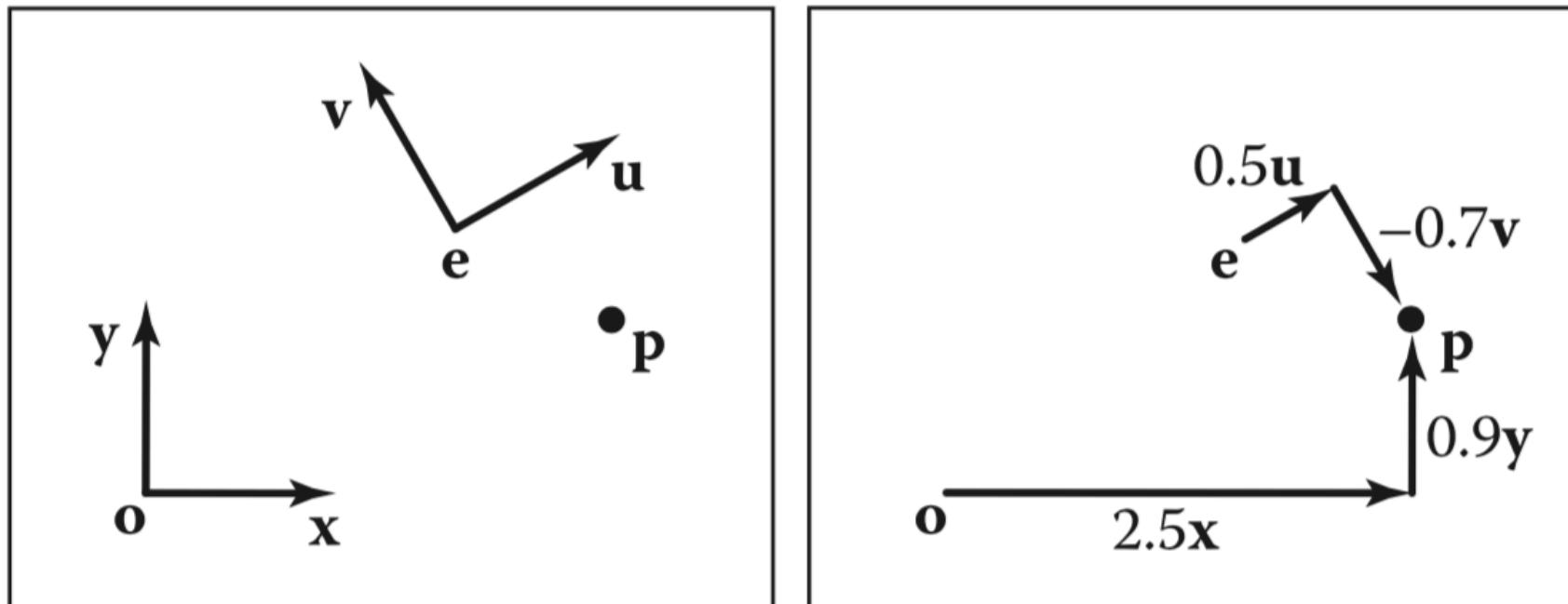
- Object-Ordered
 - Decide, for every object in the scene, its contribution to the image
- 
- Image-Ordered
 - Decide, for every pixel in the image, its contribution from every object

Coordinate Systems

- Points in space can be represented using an origin position and a set of orthogonal basis vectors:

$$\mathbf{p} = (x_p, y_p) \equiv \mathbf{0} + x_p \mathbf{x} + y_p \mathbf{y} \quad \mathbf{p} = (u_p, v_p) \equiv \mathbf{e} + u_p \mathbf{u} + v_p \mathbf{v}$$

- Any point can be described in either coordinate system



Matrices for Converting Coordinate Systems

- (Remember: Rotating the world CCW is equivalent to rotating the coordinate system CW)
- Using homogenous coordinates and affine transformations, we can convert between coordinate systems:

$$\begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_e \\ 0 & 1 & y_e \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_u & x_v & 0 \\ y_u & y_v & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_p \\ v_p \\ 1 \end{bmatrix} = \begin{bmatrix} x_u & x_v & x_e \\ y_u & y_v & y_e \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_p \\ v_p \\ 1 \end{bmatrix}$$

- More generally, any arbitrary coordinate system transform:

$$\mathbf{P}_{uv} = \begin{bmatrix} \mathbf{x}_{uv} & \mathbf{y}_{uv} & \mathbf{o}_{uv} \\ 0 & 0 & 1 \end{bmatrix} \mathbf{p}_{xy}$$

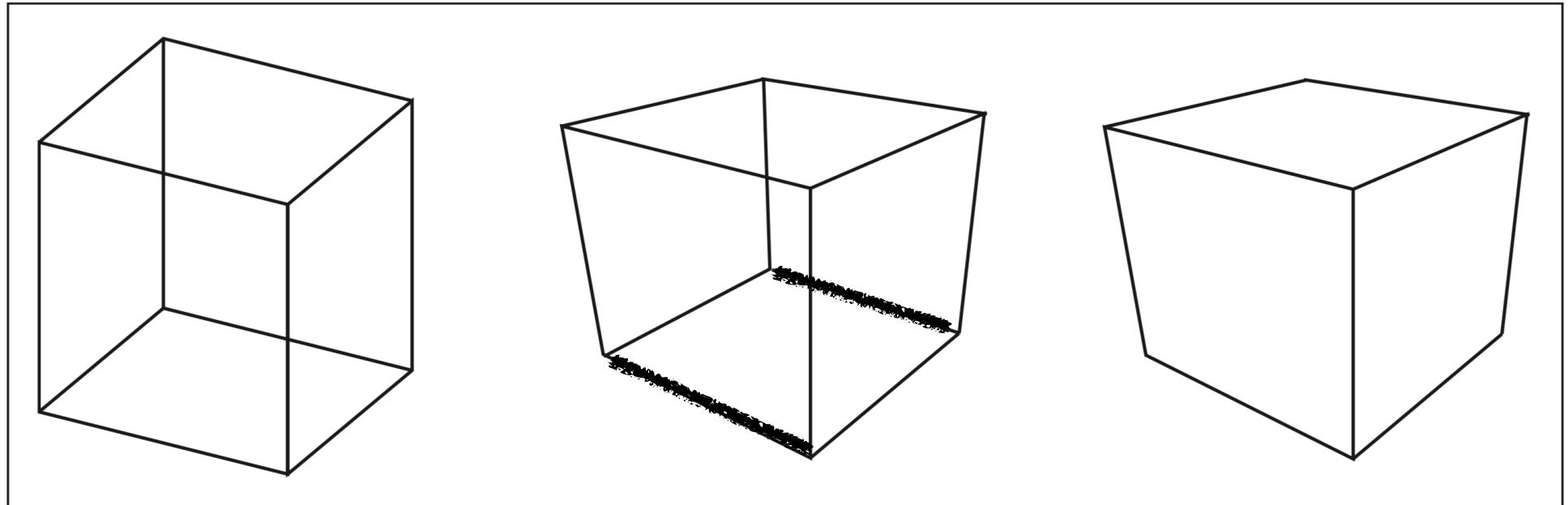
View Transformations

Using Transformations for Rendering

- Idea for today: Matrices can be used to move objects from 3D spaces to the 2D space of an image
 - Broadly, this reduction of dimensions is called **viewing transformation**
 - We will compose multiple matrix-based transformations to rethink cameras

Drawing by Transformation

- For now, we will consider drawing wireframe objects (collections of 3D line segments)



Orthographic

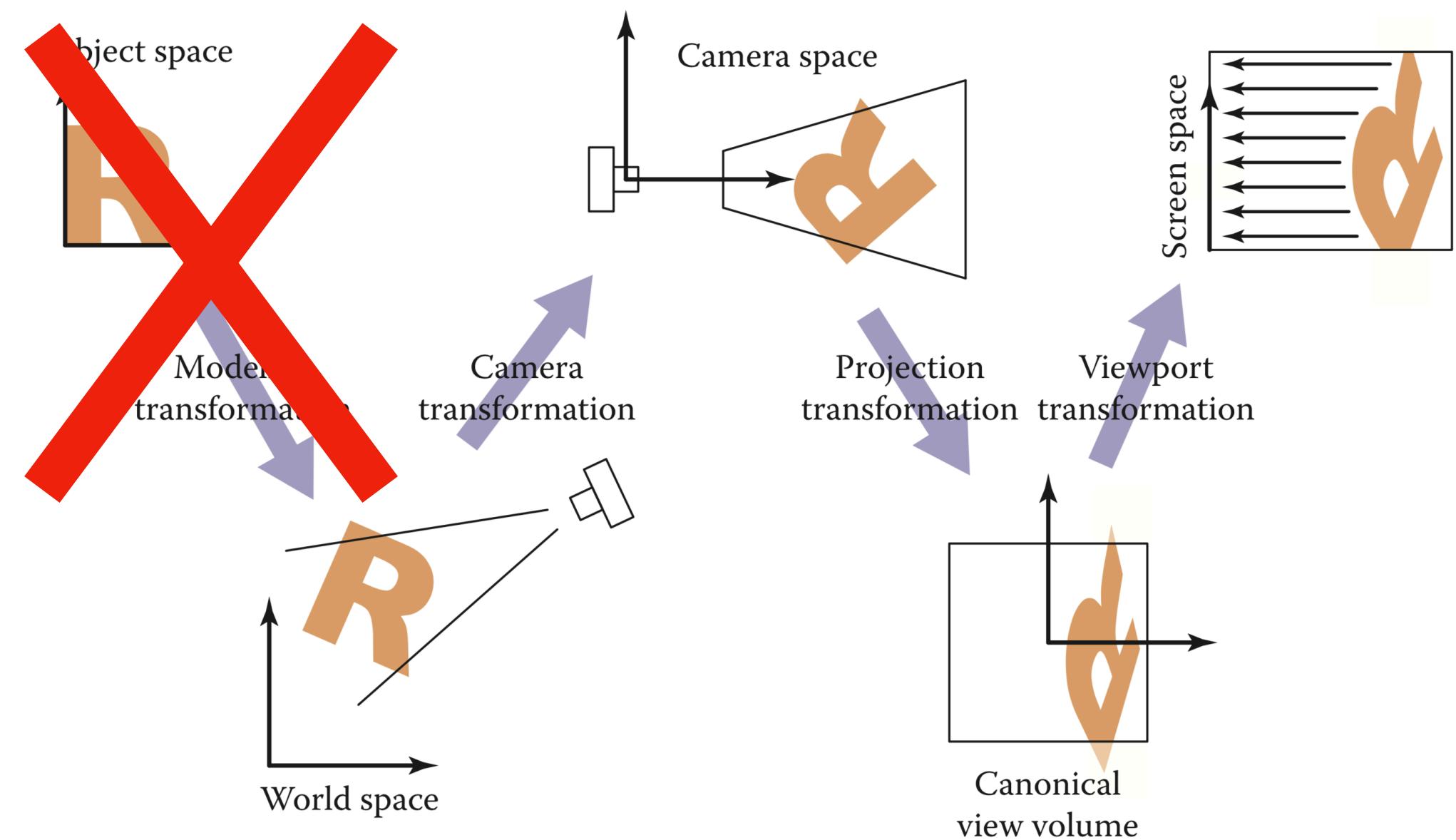
Perspective
farther objects looks smaller
Parallel lines might meet

**Perspective +
Hidden Line Removal**

Step-by-Step Viewing Transformations

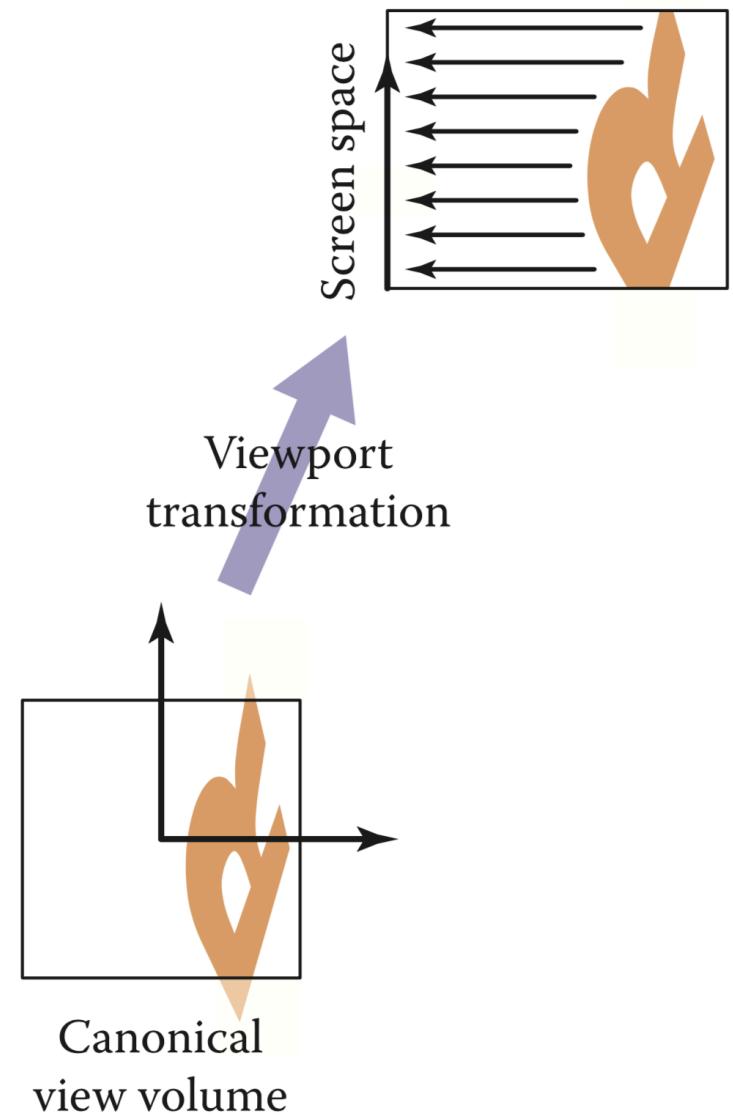
(Each arrow is a matrix)

We'll Discuss Later



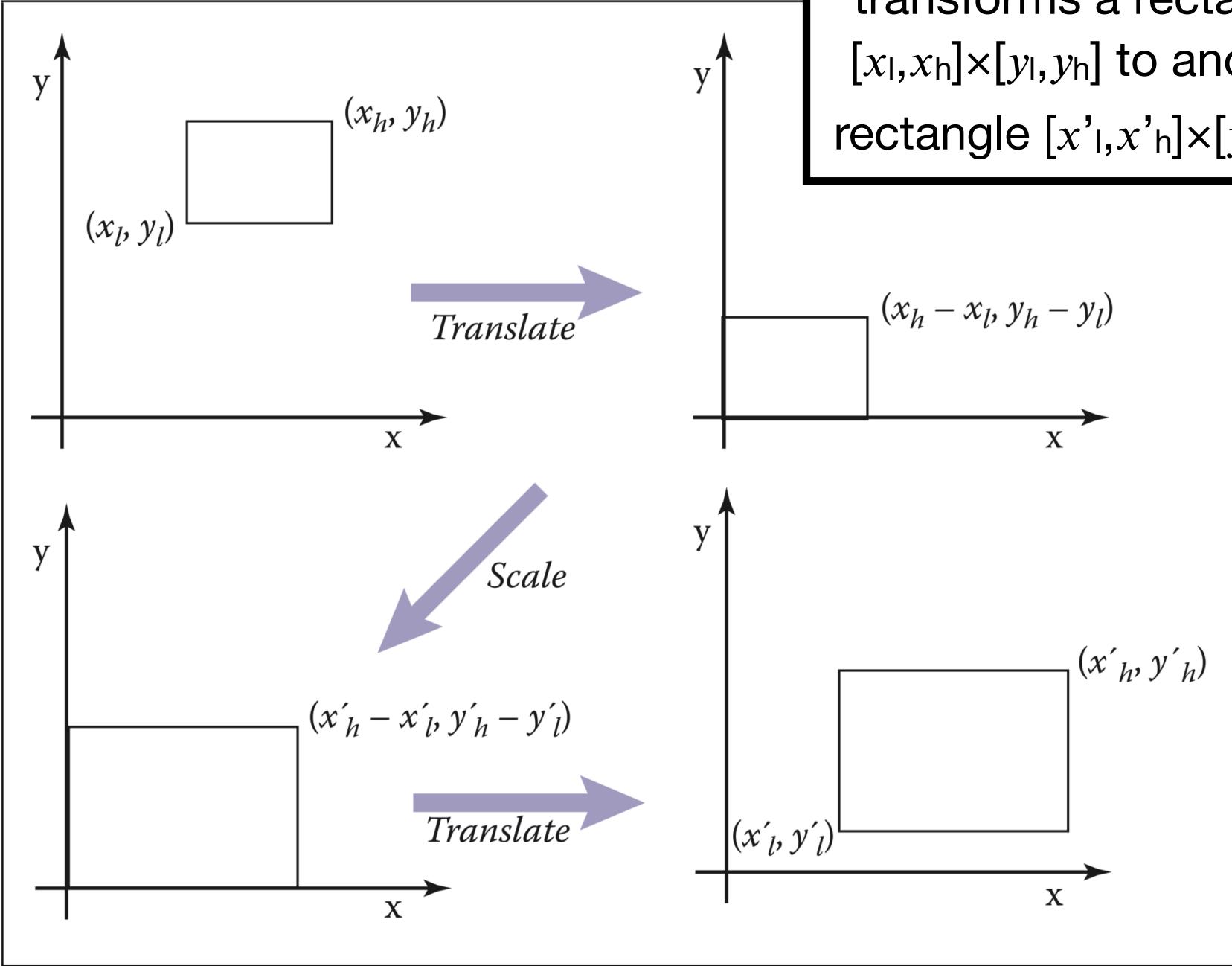
Viewport Transformation

- Goal: Transform from a canonical 2D space to pixel coordinates
 - Canonical space:
 $(x_{\text{canonical}}, y_{\text{canonical}}) \in [-1, 1] \times [-1, 1]$
 - Pixel space: Integers
 $(x_{\text{screen}}, y_{\text{screen}}) \in [0, n_x] \times [0, n_y]$
 - n_x, n_y . are the number of columns and rows
 - To be precise,
 $(x_{\text{screen}}, y_{\text{screen}}) \in [0.5, n_x - 0.5] \times [0.5, n_y - 0.5]$
- Initially, we will think of this as transformation of a 2D to 2D space



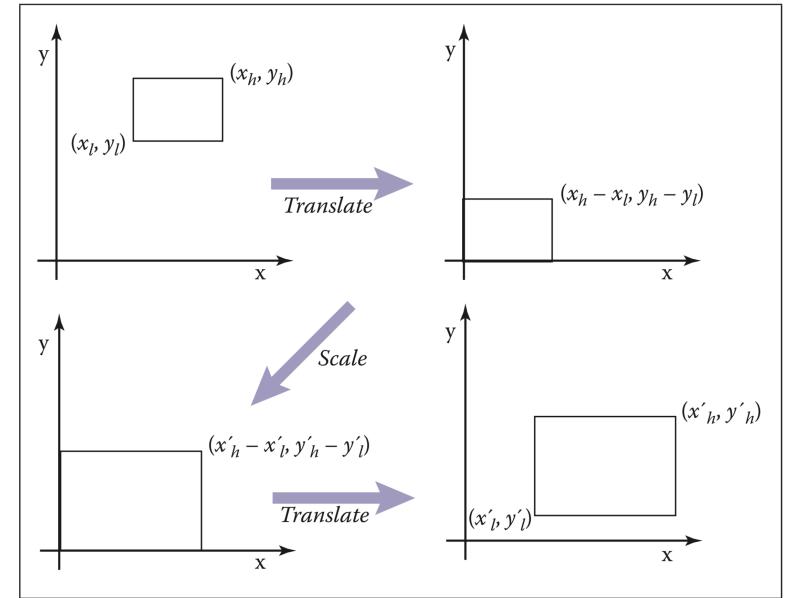
Viewports as Windowing

A **windowing operation** transforms a rectangle $[x_l, x_h] \times [y_l, y_h]$ to another rectangle $[x'_l, x'_h] \times [y'_l, y'_h]$



Viewports as Windowing

- Decompose windowing into three steps



$\text{translate}(x'_l, y'_l) \text{ scale}\left(\frac{x'_h - x'_l}{x_h - x_l}, \frac{y'_h - y'_l}{y_h - y_l}\right) \text{ translate}(-x_l, -y_l)$

Viewports as Windowing

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

- Multiplying together:

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

Sidebar: Combining a 3x3 Linear Matrix Followed by a Translation

- Translation *after* the linear transformation can always be read off separately.
- Often useful for debugging.

$$\begin{bmatrix} 1 & 0 & 0 & x_t \\ 0 & 1 & 0 & y_t \\ 0 & 0 & 1 & z_t \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & x_t \\ a_{21} & a_{22} & a_{23} & y_t \\ a_{31} & a_{32} & a_{33} & z_t \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Using Windowing to define the Viewport Transformation

- Plugging in with our known constants:
- (recall - canonical are between [-1,1]. Screen in pixels)

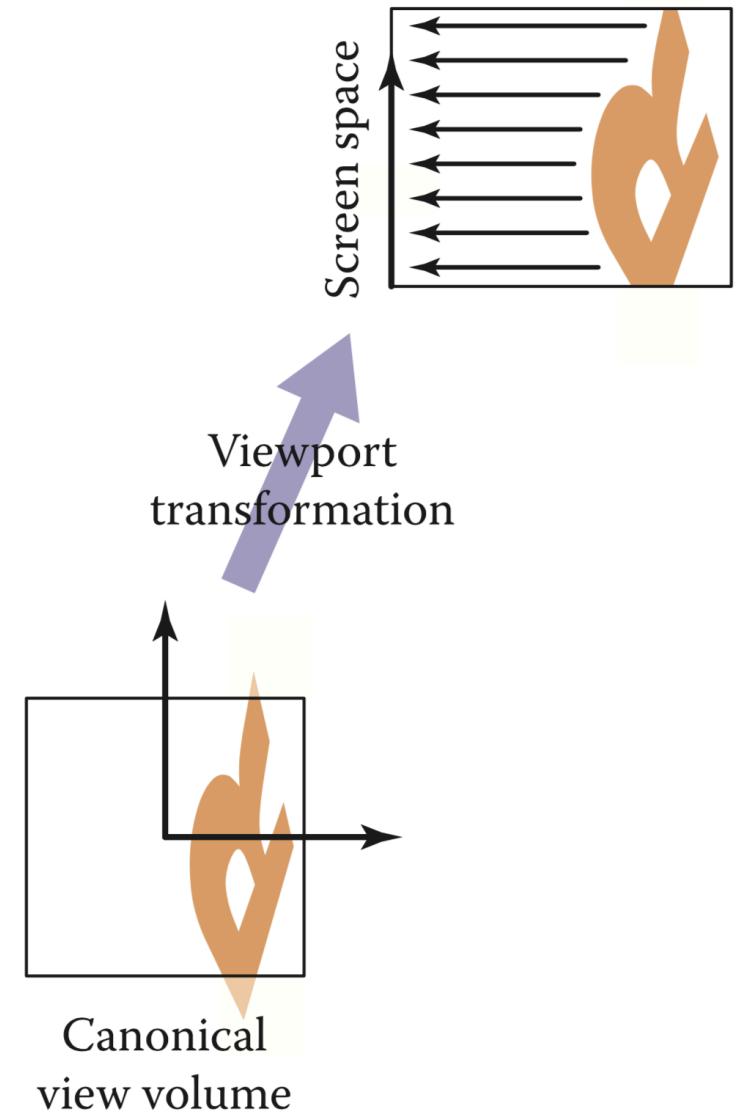
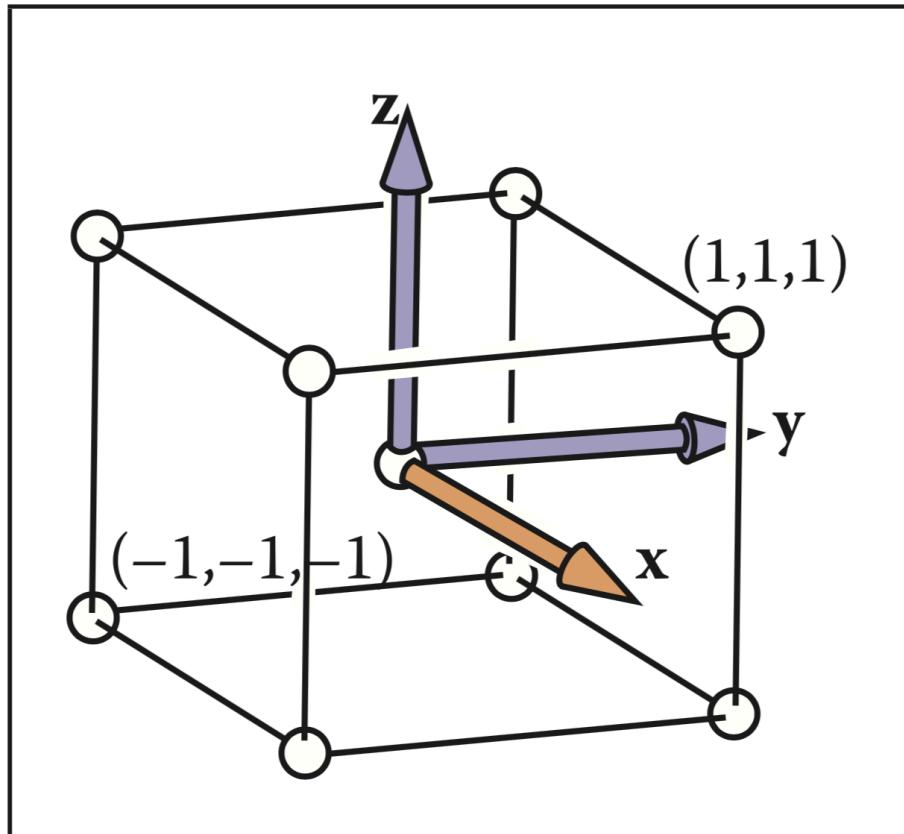
$$\begin{bmatrix} x_{\text{screen}} \\ y_{\text{screen}} \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{n_x}{2} & 0 & \frac{n_x-1}{2} \\ 0 & \frac{n_y}{2} & \frac{n_y-1}{2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{\text{canonical}} \\ y_{\text{canonical}} \\ 1 \end{bmatrix}$$

- Right now, we do not need z-values, but eventually we will need to carry them through with no changes:

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

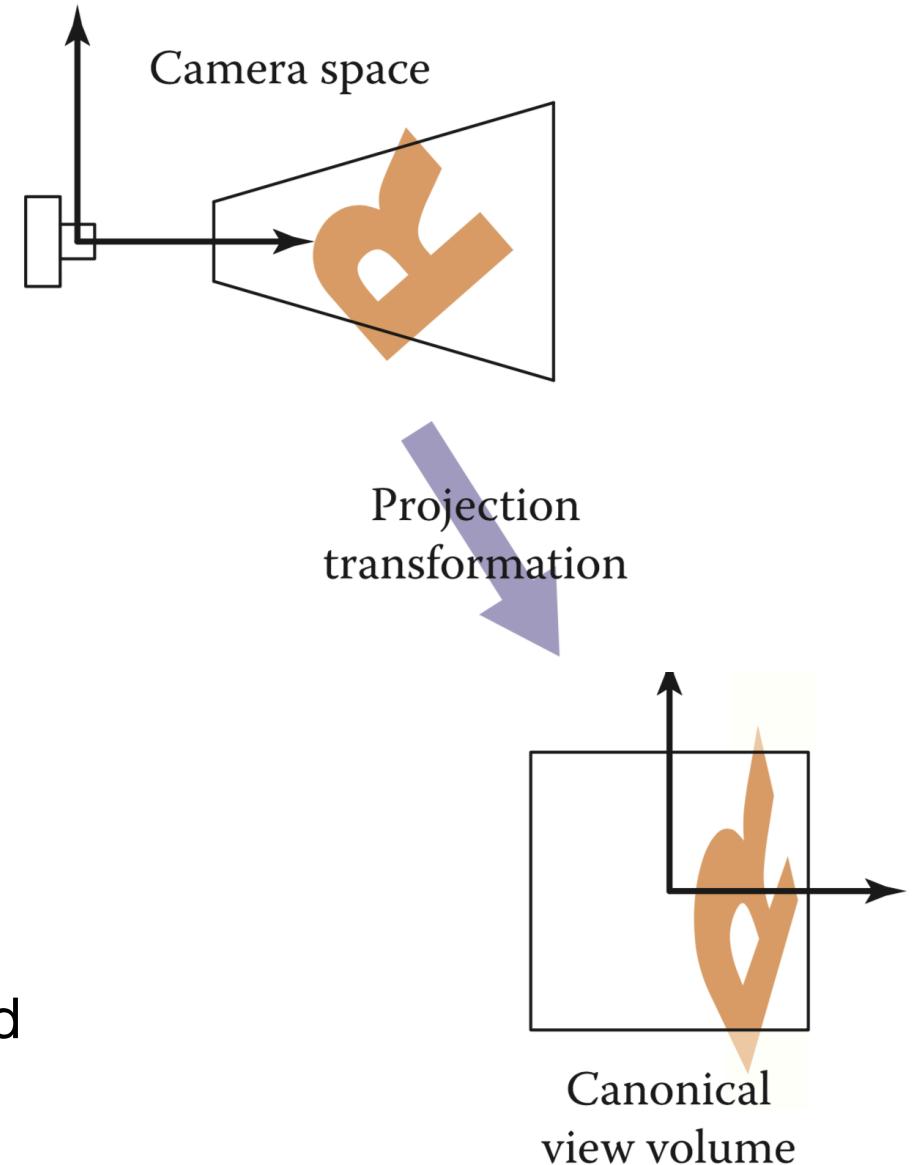
Canonical View Volume

- In actuality, our viewport transformation will work with the **canonical view volume**



Orthographic Projection

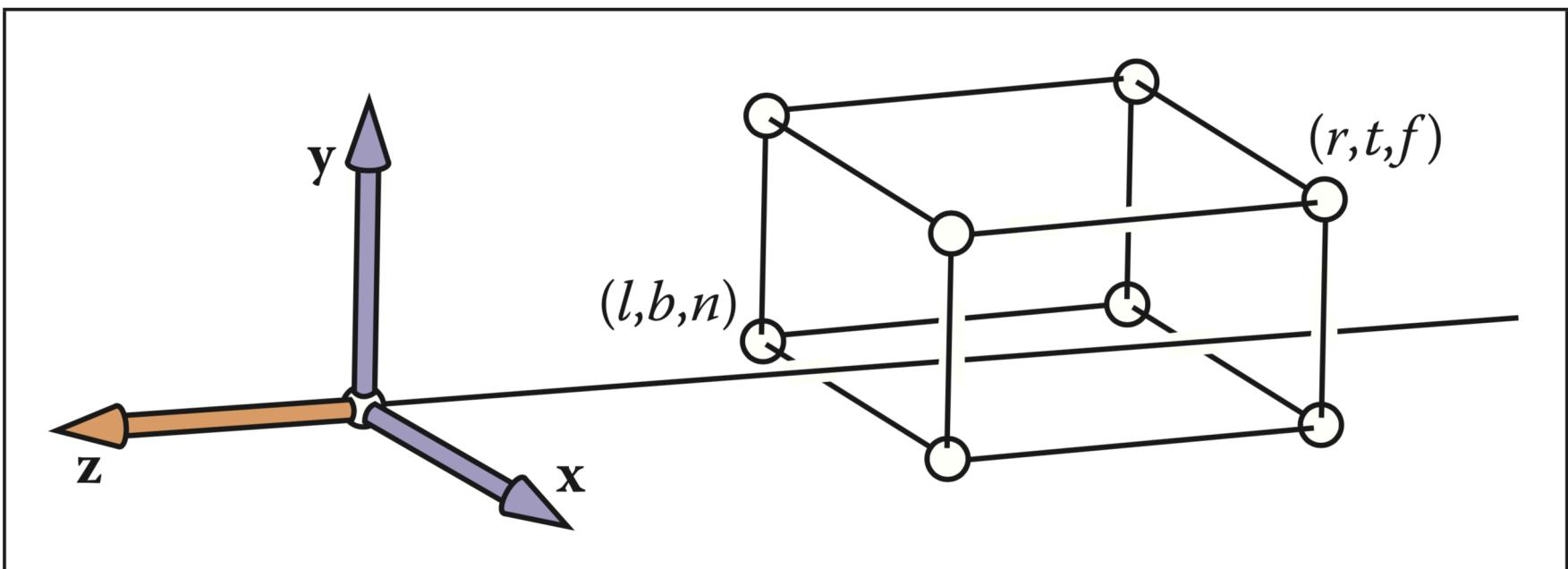
- Goal: Convert objects from 3D representation to canonical view volume
- We will start by modeling this 3D space as an axis-aligned box
 - View volume: $[l,r] \times [b,t] \times [f,n]$
 - Canonical view volume: $[-1,1] \times [-1,1] \times [-1,1]$
- Reshapes the view volume as defined by the camera



Orthographic Projection

- **Orthographic view volume**
defined by six scalars:
- Convention: $n > f$, but note that
both are negative

$x = l \equiv$ left plane,
 $x = r \equiv$ right plane,
 $y = b \equiv$ bottom plane,
 $y = t \equiv$ top plane,
 $z = n \equiv$ near plane,
 $z = f \equiv$ far plane.



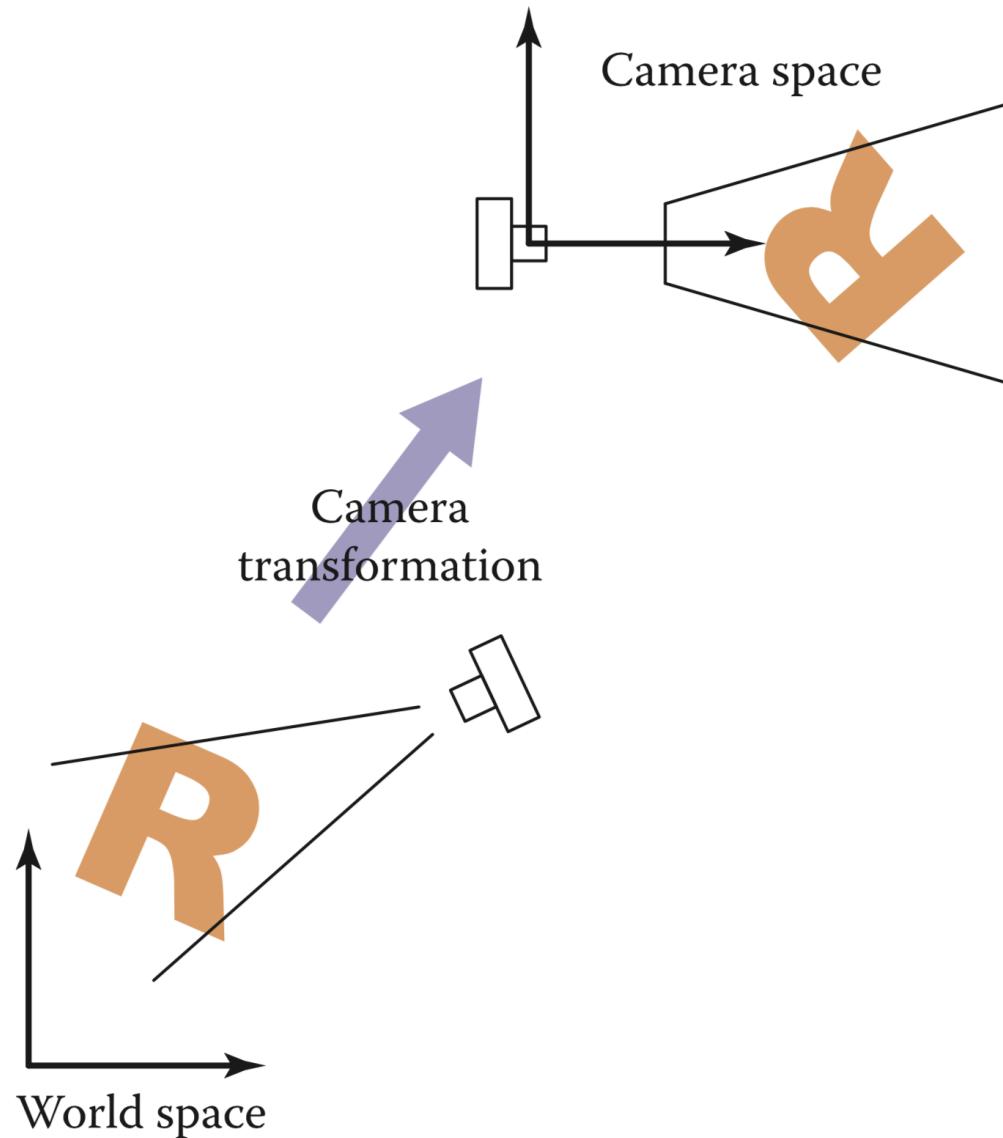
Orthographic Projection

- Just a 3D windowing transformation!

$$\begin{bmatrix} \frac{x'_h - x'_l}{x_h - x_l} & 0 & 0 & \frac{x'_l x_h - x'_h x_l}{x_h - x_l} \\ 0 & \frac{y'_h - y'_l}{y_h - y_l} & 0 & \frac{y'_l y_h - y'_h y_l}{y_h - y_l} \\ 0 & 0 & \frac{z'_h, z'_l}{z_h - z_l} & \frac{z'_l z_h - z'_h z_l}{z_h - z_l} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{M}_{\text{orth}} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

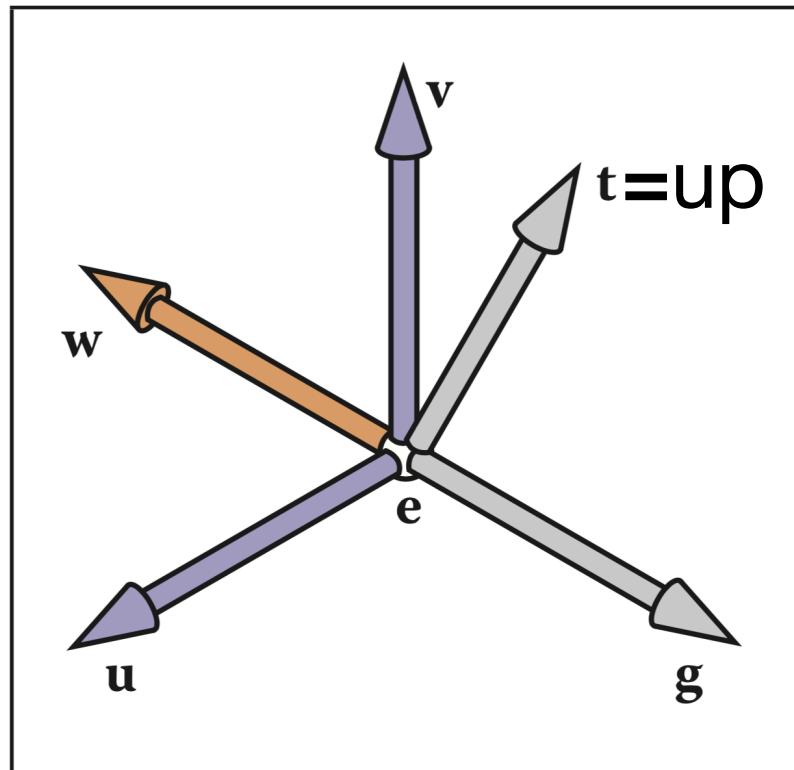
Camera Transformations

- Goal: Transform 3D space to arbitrary camera parameters
- Camera modeled with three vectors:
 - e , the eye position
 - g , the gaze direction
 - t , the view **up** direction



Camera Coordinates

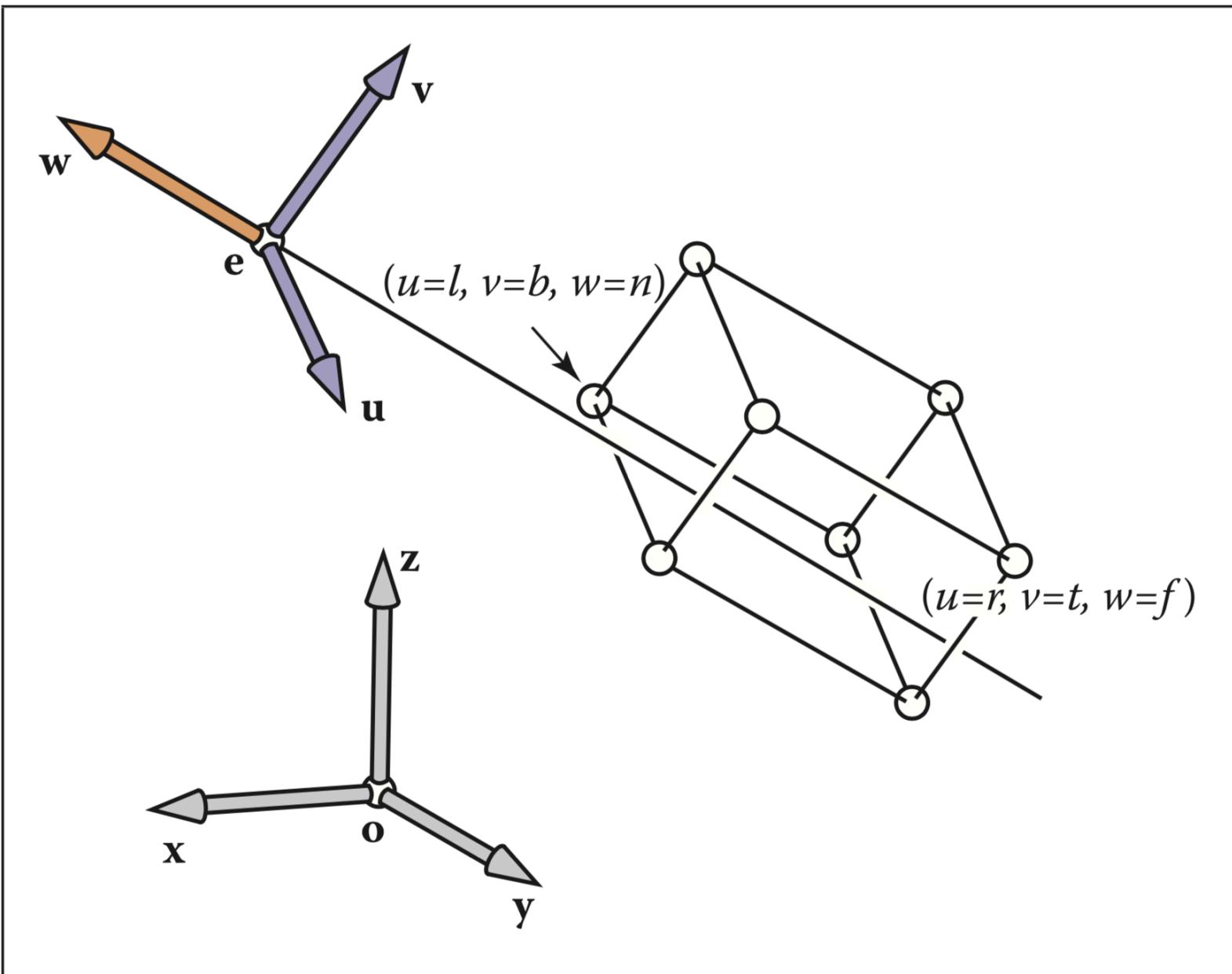
- We will convert to a camera coordinate system with origin, e , and orthogonal basis vectors \mathbf{u} , \mathbf{v} , and \mathbf{w} .



e, the eye position
g, the gaze direction
t, the view up direction

$$\begin{aligned}\mathbf{w} &= -\frac{\mathbf{g}}{\|\mathbf{g}\|}, \\ \mathbf{u} &= -\frac{\mathbf{t} \times \mathbf{w}}{\|\mathbf{t} \times \mathbf{w}\|}, \\ \mathbf{v} &= \mathbf{w} \times \mathbf{u}.\end{aligned}$$

Camera Coordinates



Changing Coordinates

- We need to both translate the origin and change coordinate systems

$$\mathbf{M}_{\text{cam}} = \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{e} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

What is M_orth

- Project points for (xyz) into (xy) ?
- (x,y,z)-> (x,y,0)
- (x,y,z,w)-> (x,y,0,w) ??

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \textcolor{blue}{0} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Viewing Algorithm

```
construct M_vp  
construct M_orth  
construct M_cam  
M = M_vp * M_orth * M_cam  
for each 3D object O {  
    O_screen = M * O  
    draw(O_screen)  
}
```

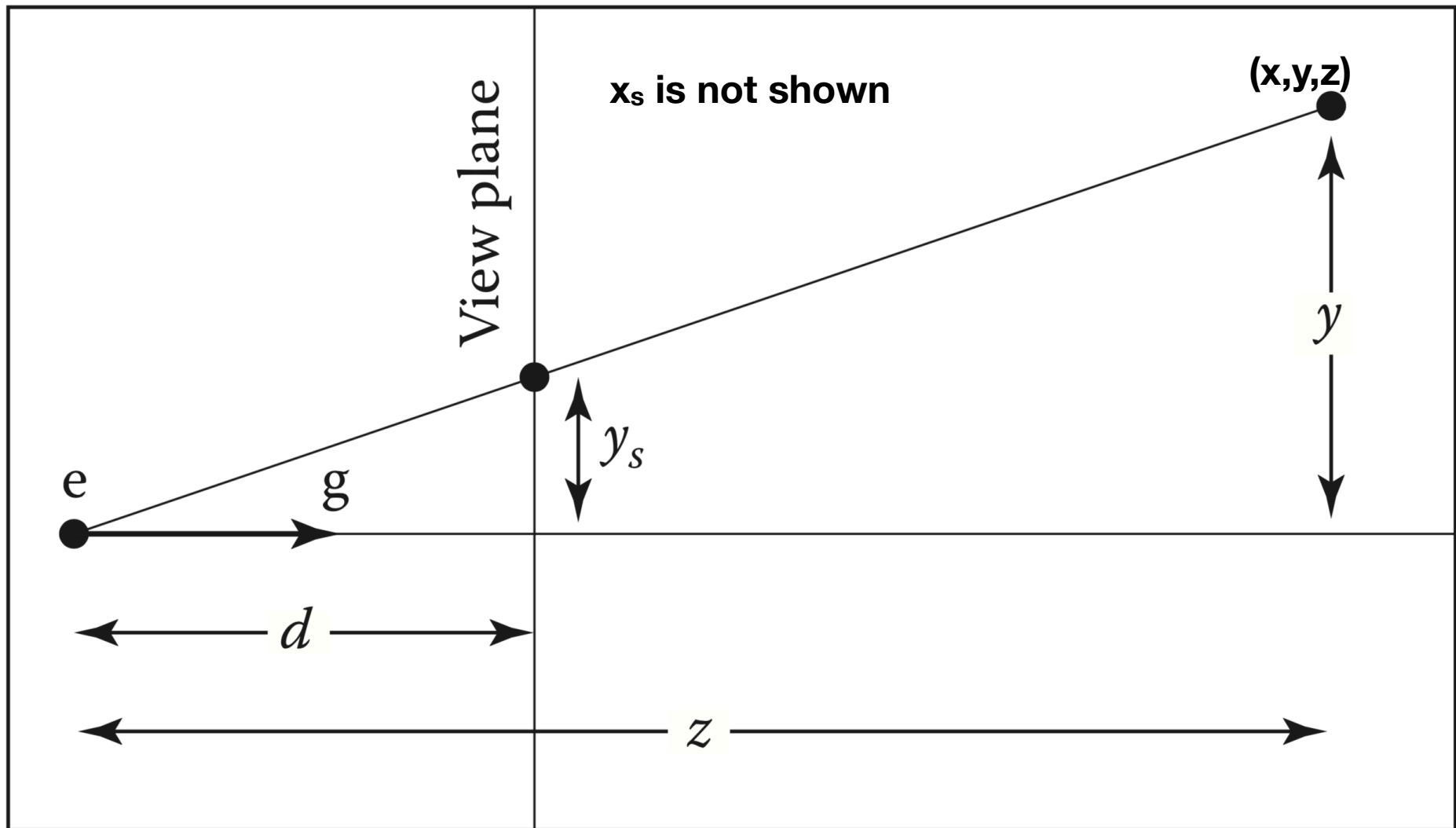
For example, if O is a triangle, with vertices **a**, **b**, and **c**, transform all 3 vertices **Ma**, **Mb**, and **Mc**

Projective Transformations

Relative Size Based on Distance

- Key idea of perspective: the size of an object on the screen is proportional to $1/z$
 (x_s, y_s) are coordinates of the point (x, y, z) on the screen

$$y_s = \frac{d}{z} y$$



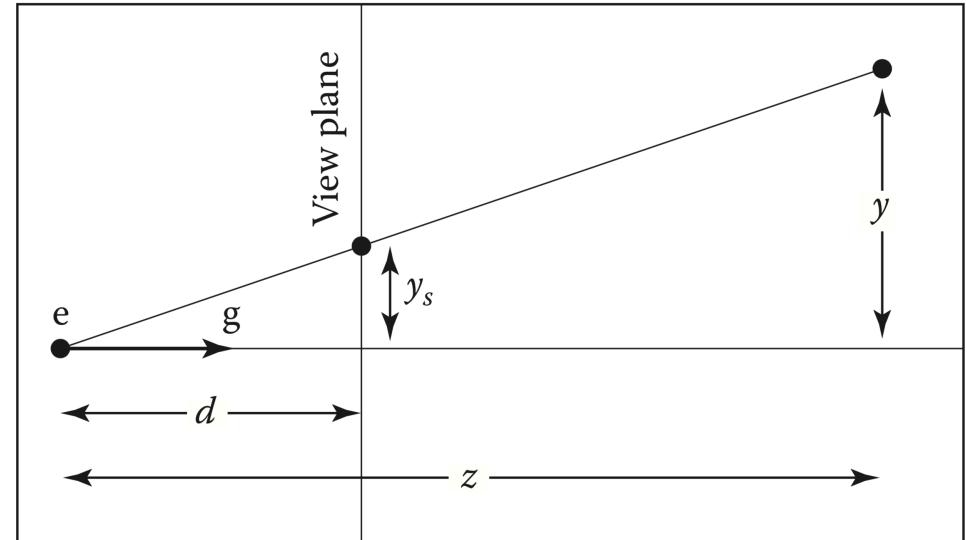
Using Homographies for Perspective

- We can now replace:

$$y_s = \frac{d}{z} y$$

- With:

$$\begin{bmatrix} y_s \\ 1 \end{bmatrix} \sim \begin{bmatrix} d & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} y \\ z \\ 1 \end{bmatrix}$$

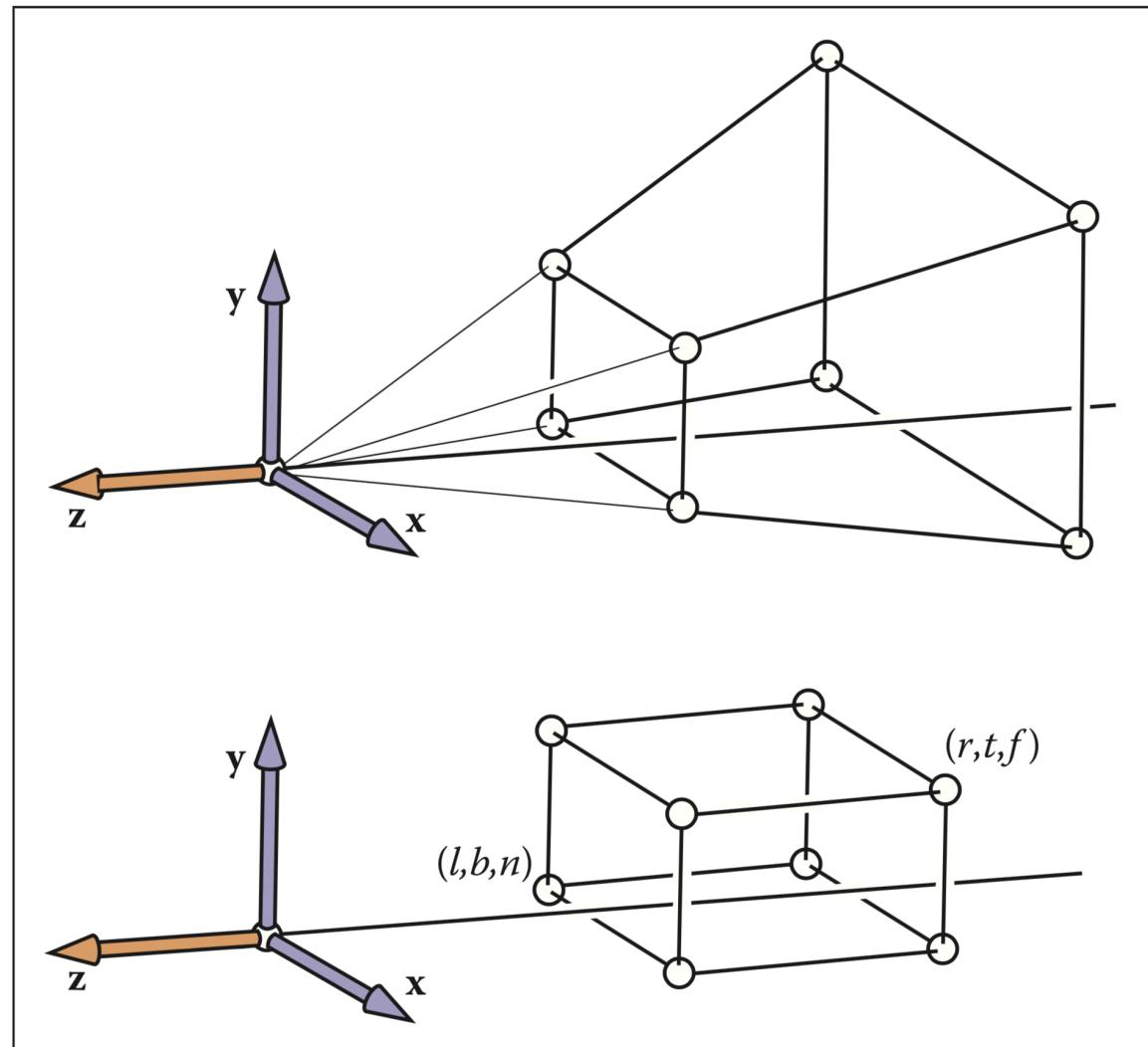


Perspective Matrix

- Our matrix:

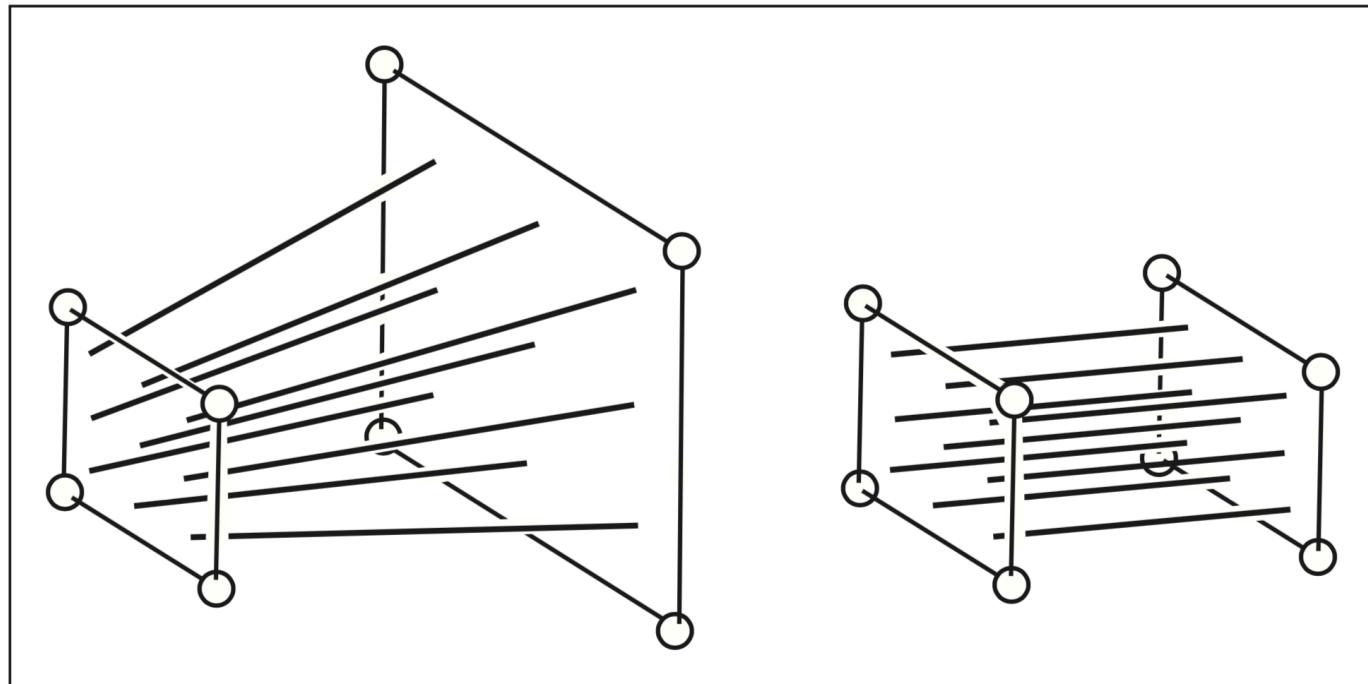
$$\mathbf{P} = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n + f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

- Keeps near plane fixed, maps far plane to back of the box



Perspective Distortion

- Effect on view rays / lines:



- Note that affine transformation cannot do this because it keeps parallel lines parallel

Perspective Distortion

- Perspective matrix effect on coordinates is nonlinear distortion in z:

$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- But it does, however, preserve order in the z-coordinate (which will become useful very soon)

Perspective Projection Matrix

- Concatenating the perspective matrix with the orthographic projection provides the perspective projection matrix:

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

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

- We can define l , r , b , and t relative to the near plane, since we keep it fixed

Putting it all together

```
construct M_vp  
construct M_per  
construct M_cam  
M = M_vp * M_per * M_cam  
for each 3D object O {  
    O_screen = M * O  
    draw(O_screen)  
}
```

Equivalently:

$$\mathbf{M} = \mathbf{M}_{vp} \mathbf{M}_{orth} \mathbf{P} \mathbf{M}_{cam}$$

For a given vertex $\mathbf{a} = (x, y, z)$,
 $\mathbf{p} = \mathbf{Ma}$ should result in
drawing $(x_p/w_p, y_p/w_p, z_p/w_p)$
on the screen

The Graphics Pipeline

- The sequence of operations that we'll perform to transform from objects in 3D to color pixels on the screen.
 - A “pipeline” because there are many stages
 - All variants of object-ordered rendering follow a similar sequence to what we'll discuss
 - Generally, there are plenty of opportunities for optimization in this and it can be much faster than raytracing with the right hardware

The Graphics Pipeline

Geometric **primitives**
described by **vertices**

TODAY

Primitives converted
into **fragments**, one
for each pixel that
store **interpolated**
per-vertex data

Application

Command Stream

Vertex Processing

Transformed Geometry

Rasterization

Fragments

Fragment Processing

Blending

Framebuffer Image

Display

Updated **positions** of
vertices and computed
per-vertex data

Fragments processed
to compute final colors
for each **pixel**

Geometric Primitives

- Points, lines, and triangles **only**.
- What about?
 - Curves/Text? Approximate them with chains of line segments
 - Polygons? Break them up into triangles
 - Curved surfaces? Approximate them with triangles
- Using minimal primitives keeps the pipeline simple and uniform for efficiency.

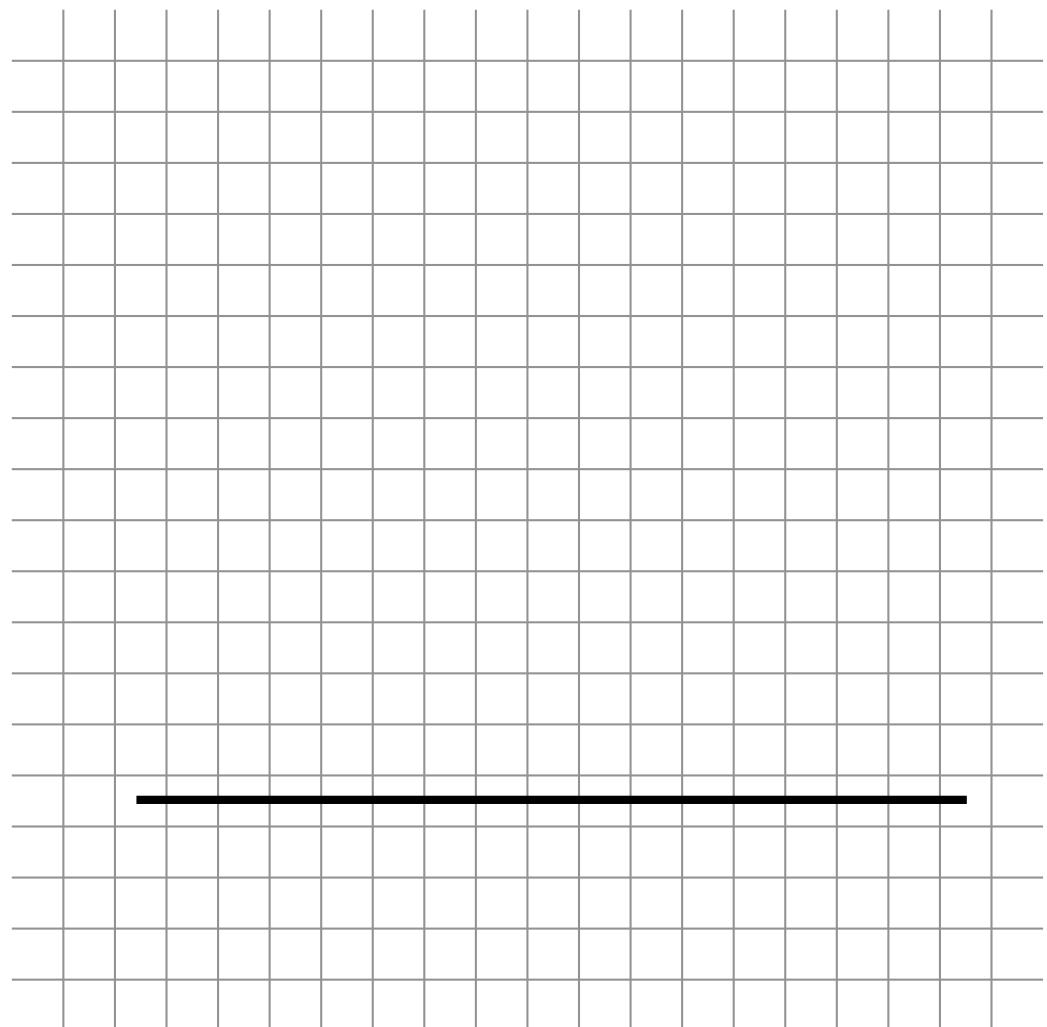
Rasterization

Two Jobs of a Rasterizer

- To *enumerate* the pixels that are covered by each primitive
- To *interpolate* the values/attributes across the primitives, e.g. normals, colors, texture coordinates
- Output: a set of **fragments**, one for each pixel.
 - Fragments live at a pixel and store the interpolated attributes.

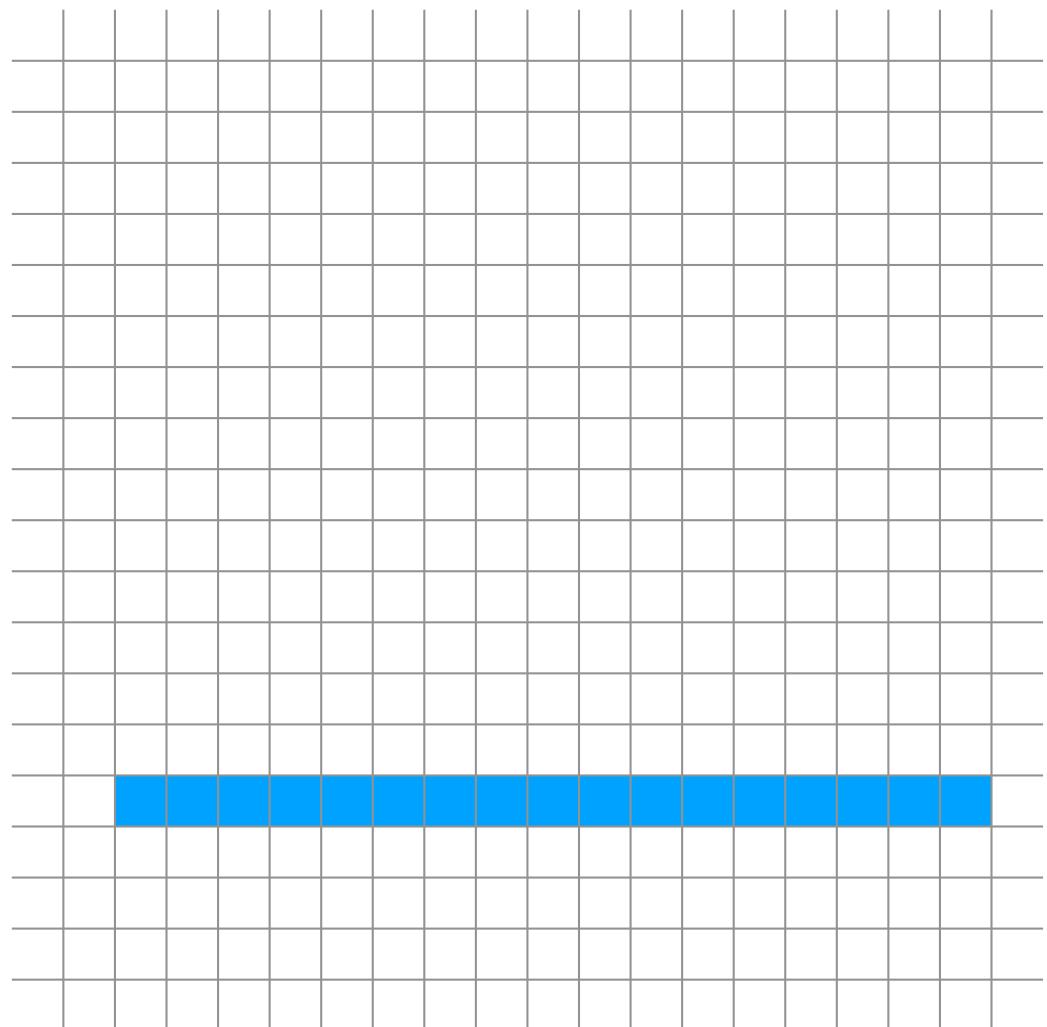
Rasterizing Lines to Fragments w/ Point Sampling

- Option 1: Fatten the line and decide which pixels hit the rectangle



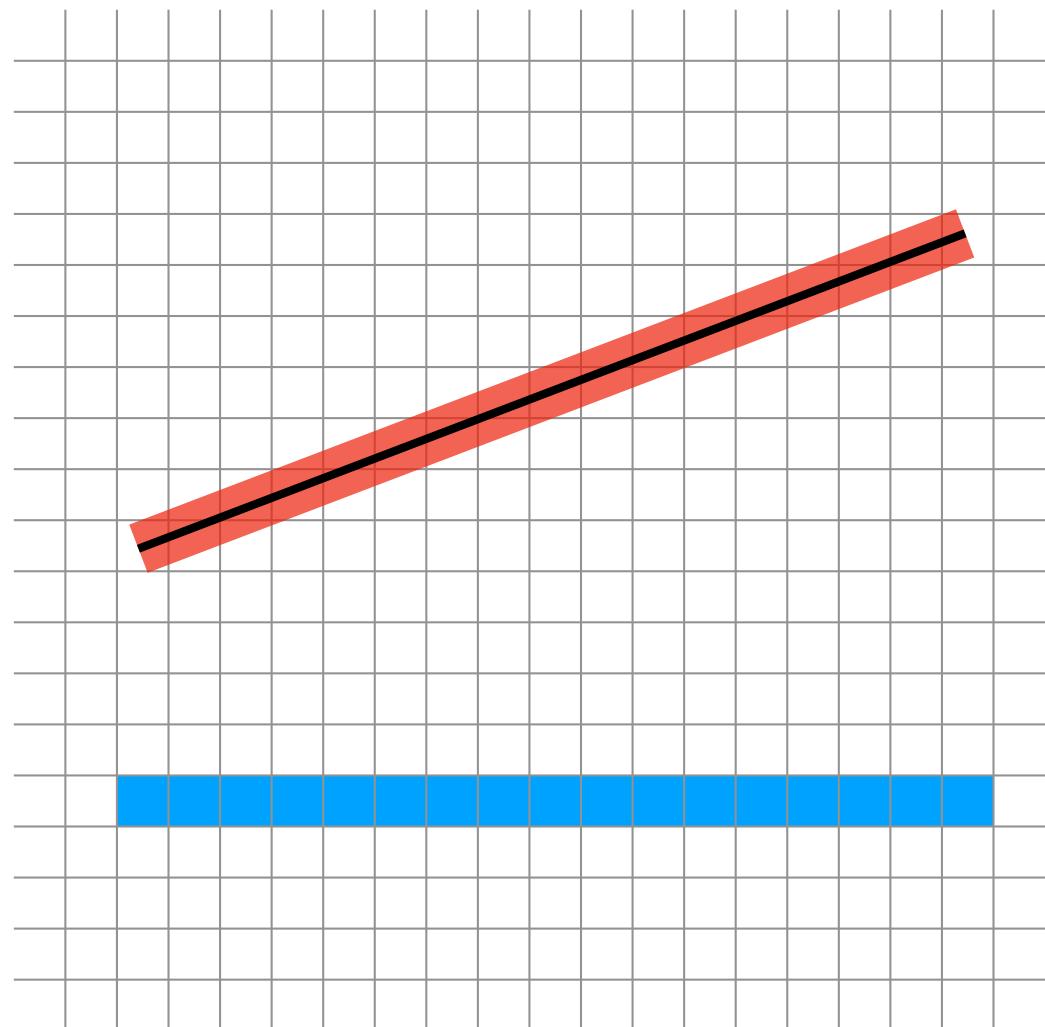
Rasterizing Lines to Fragments w/ Point Sampling

- Option 1: Fatten the line and decide which pixels hit the rectangle



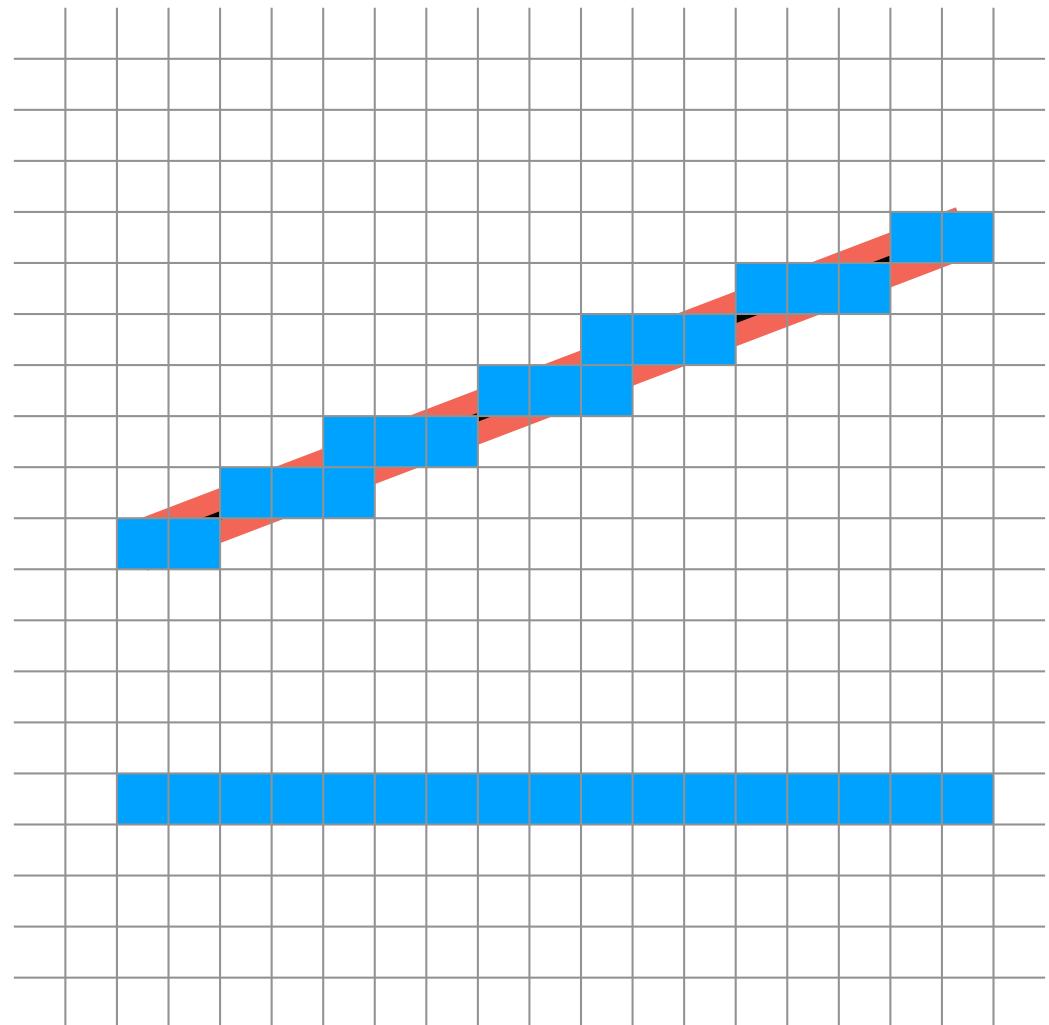
Rasterizing Lines to Fragments w/ Point Sampling

- Option 1: Fatten the line and decide which pixels hit the rectangle
- Decide which pixels are turned on if their center is inside the line



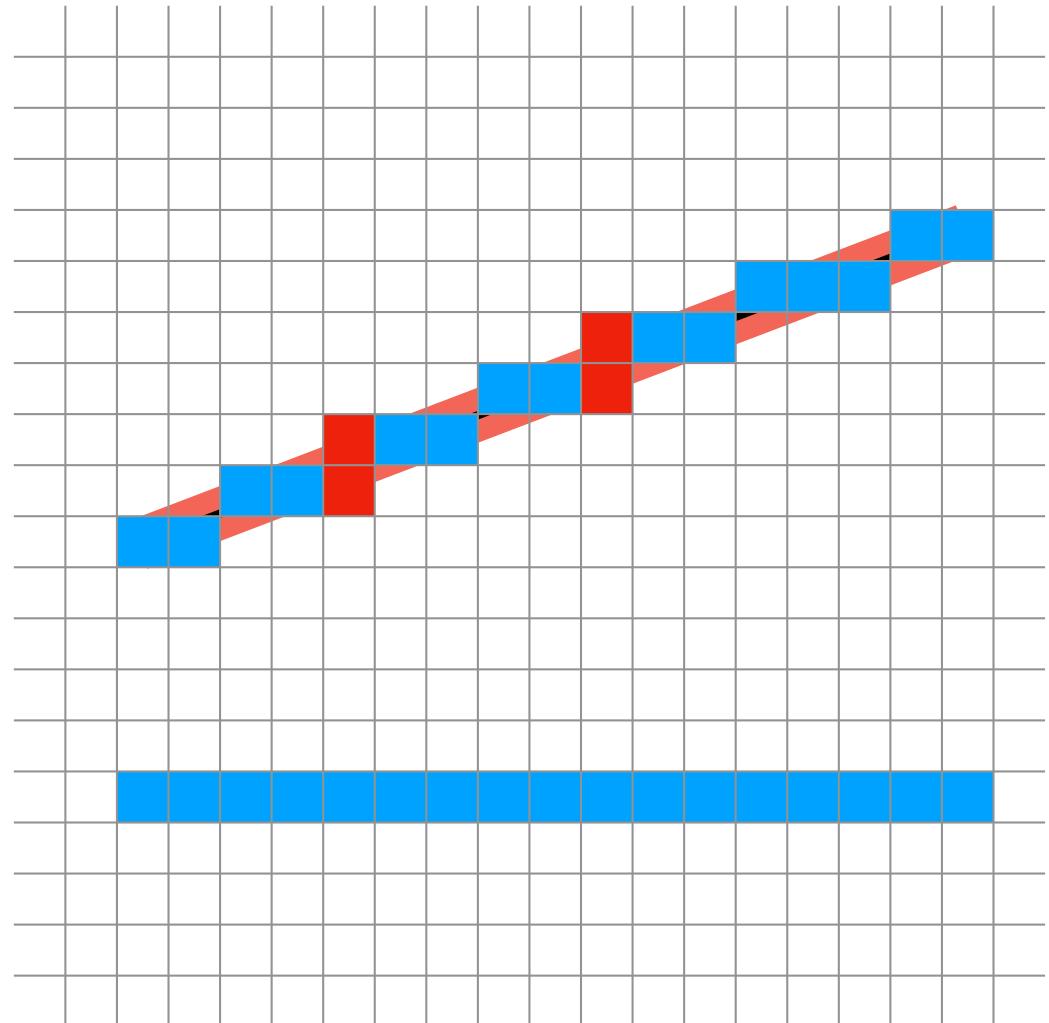
Rasterizing Lines to Fragments w/ Point Sampling

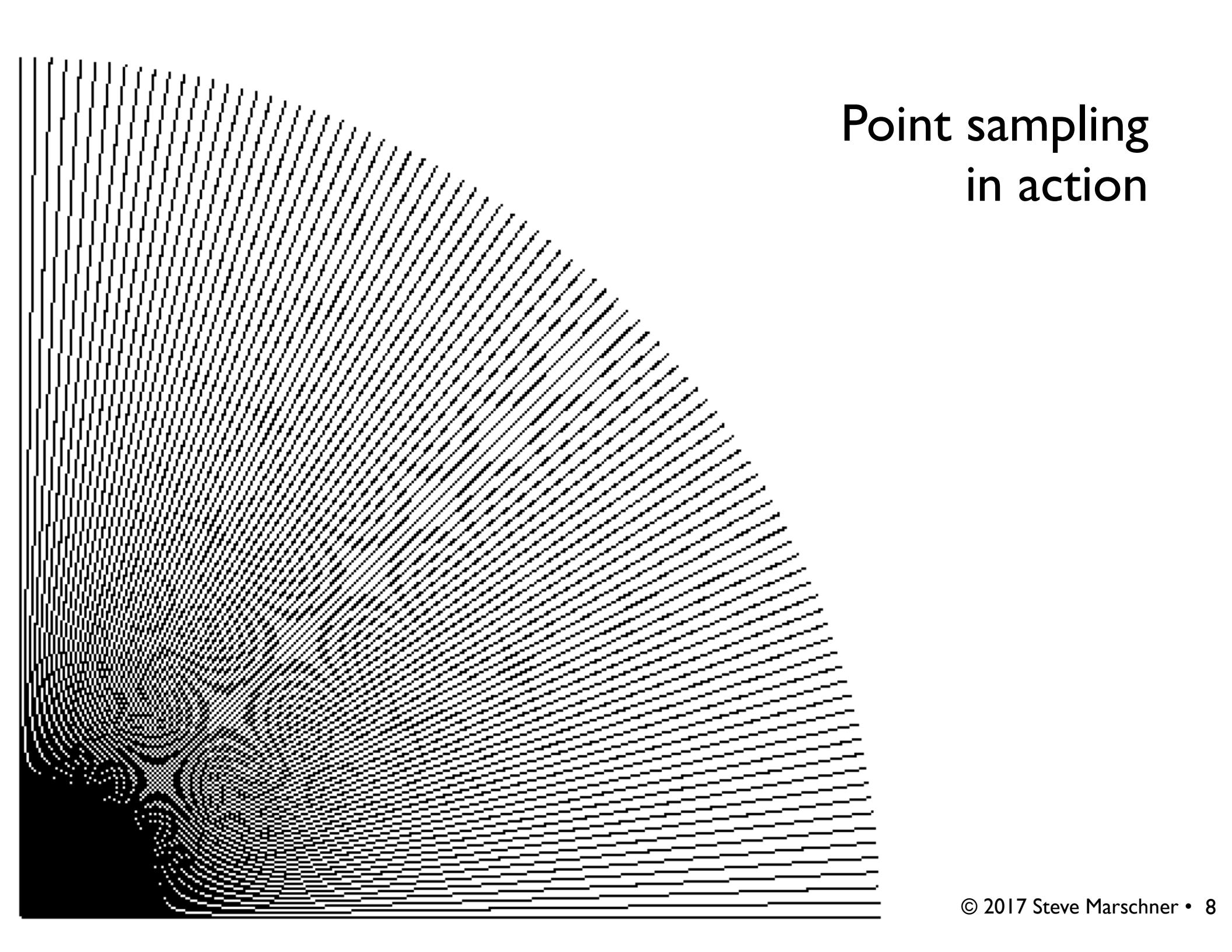
- Option 1: Fatten the line and decide which pixels hit the rectangle
- Decide which pixels are turned on if their center is inside the line



Rasterizing Lines to Fragments w/ Point Sampling

- Option 1: Fatten the line and decide which pixels hit the rectangle
- Decide which pixels are turned on if their center is inside the line
- Problem: line width is not kept constant





Point sampling in action

Bresenham's Algorithm

- Foundational algorithm in computer graphics while at IBM from
- Published in 1965

An algorithm is given for computer control of a digital plotter.

The algorithm may be programmed without multiplication or division instructions and is efficient with respect to speed of execution and memory utilization.

Algorithm for computer control of a digital plotter

by J. E. Bresenham

This paper describes an algorithm for computer control of a type of digital plotter that is now in common use with digital computers.¹

The plotter under consideration is capable of executing, in response to an appropriate pulse, any one of the eight linear movements shown in Figure 1. Thus, the plotter can move linearly from a point on a mesh to any adjacent point on the mesh. A typical mesh size is 1/100th of an inch.

The data to be plotted are expressed in an (x, y) rectangular coordinate system which has been scaled with respect to the mesh; i.e., the data points lie on mesh points and consequently have integral coordinates.

It is assumed that the data include a sufficient number of appropriately selected points to produce a satisfactory representation of the curve by connecting the points with line segments, as illustrated in Figure 2. In Figure 3, the line segment connecting

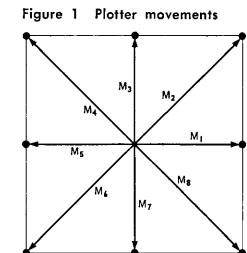


Figure 2 Curve defined by linear segments joining data points



Equations for a line

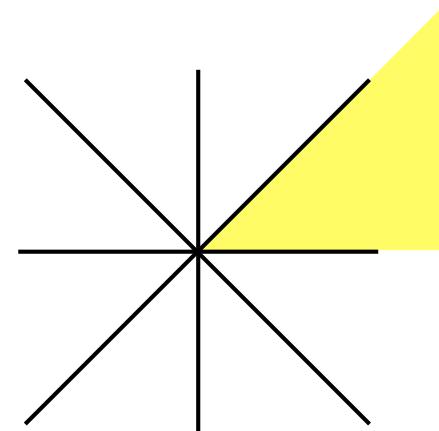
- Given points (x_0, y_0) and (x_1, y_1) , the implicit equation for the line is:

$$f(x, y) \equiv (y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0 = 0$$

- We'll consider lines only defined in one octant: $x_0 \leq x_1$ (otherwise, swap the points) and that the slope of the line, $m \in (0, 1]$ where

$$m = \frac{y_1 - y_0}{x_1 - x_0}$$

- But the same technique can be used for any octant

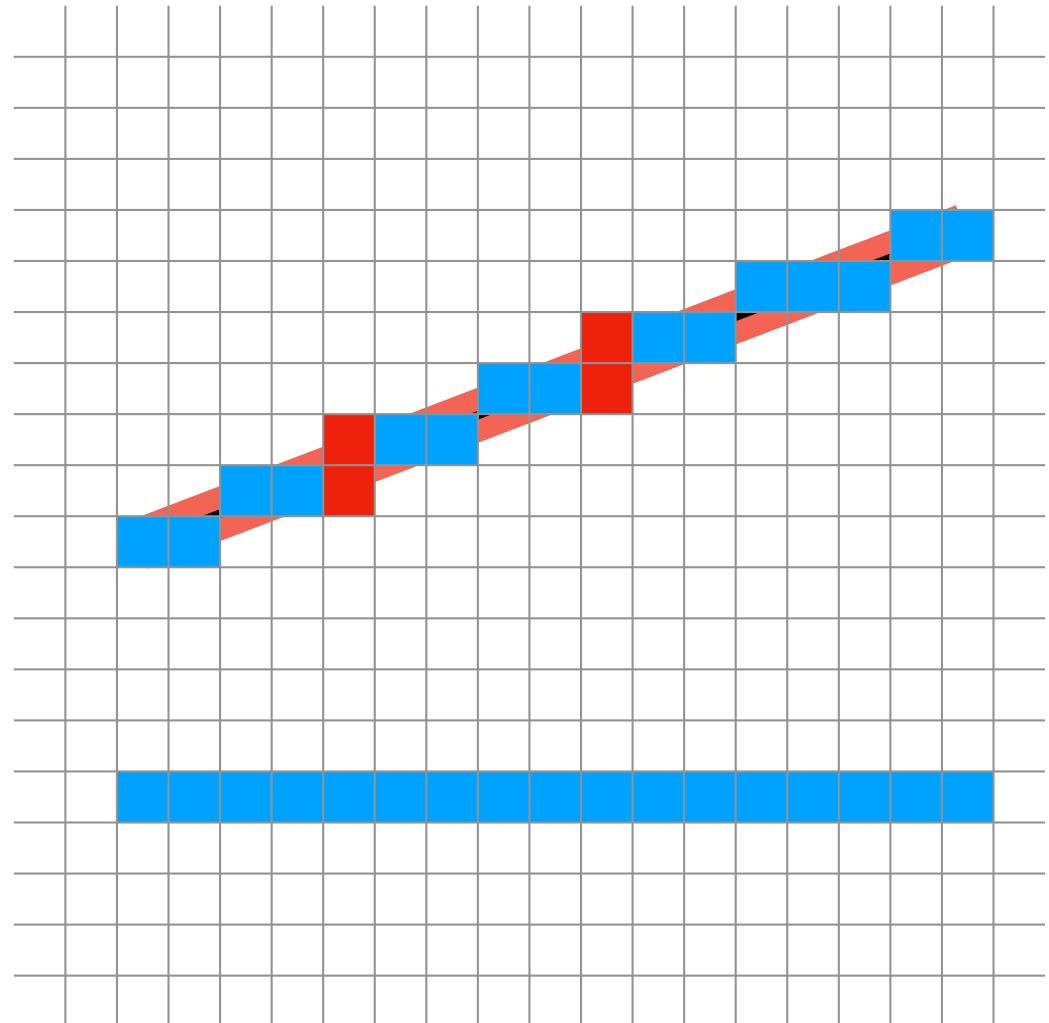


Bresenham's Approach

- Key idea: only turn on one pixel per column
- Multiple variants of this algorithm, we'll discuss the **midpoint algorithm**

Resolving Sampling Issues with the Midpoint Algorithm

- Turn on only the pixel closest to the line in each column



Resolving Sampling Issues with the Midpoint Algorithm

- Turn on only the pixel closest to the line in each column

