

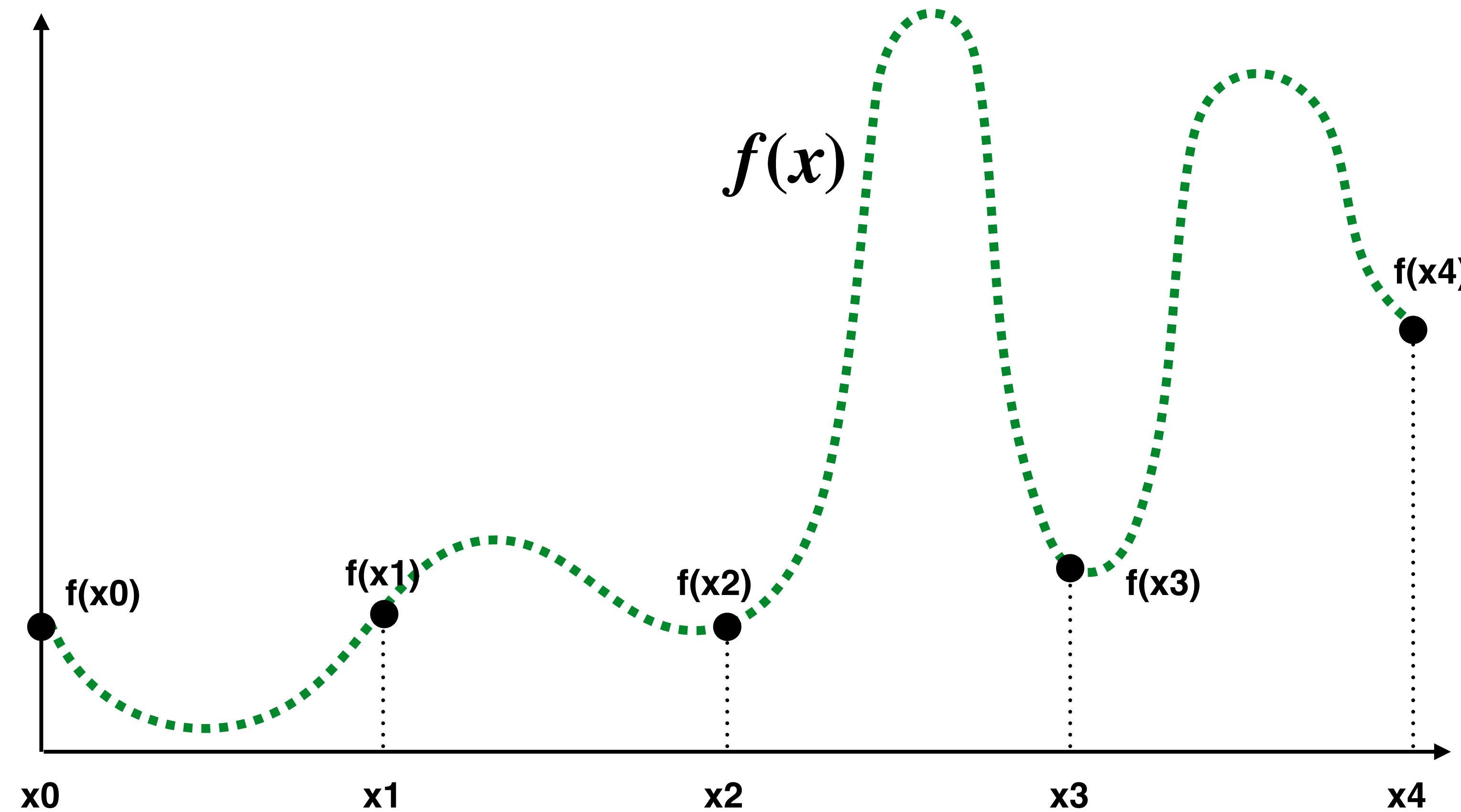
Lecture 3:

Coordinate Spaces and Transformations

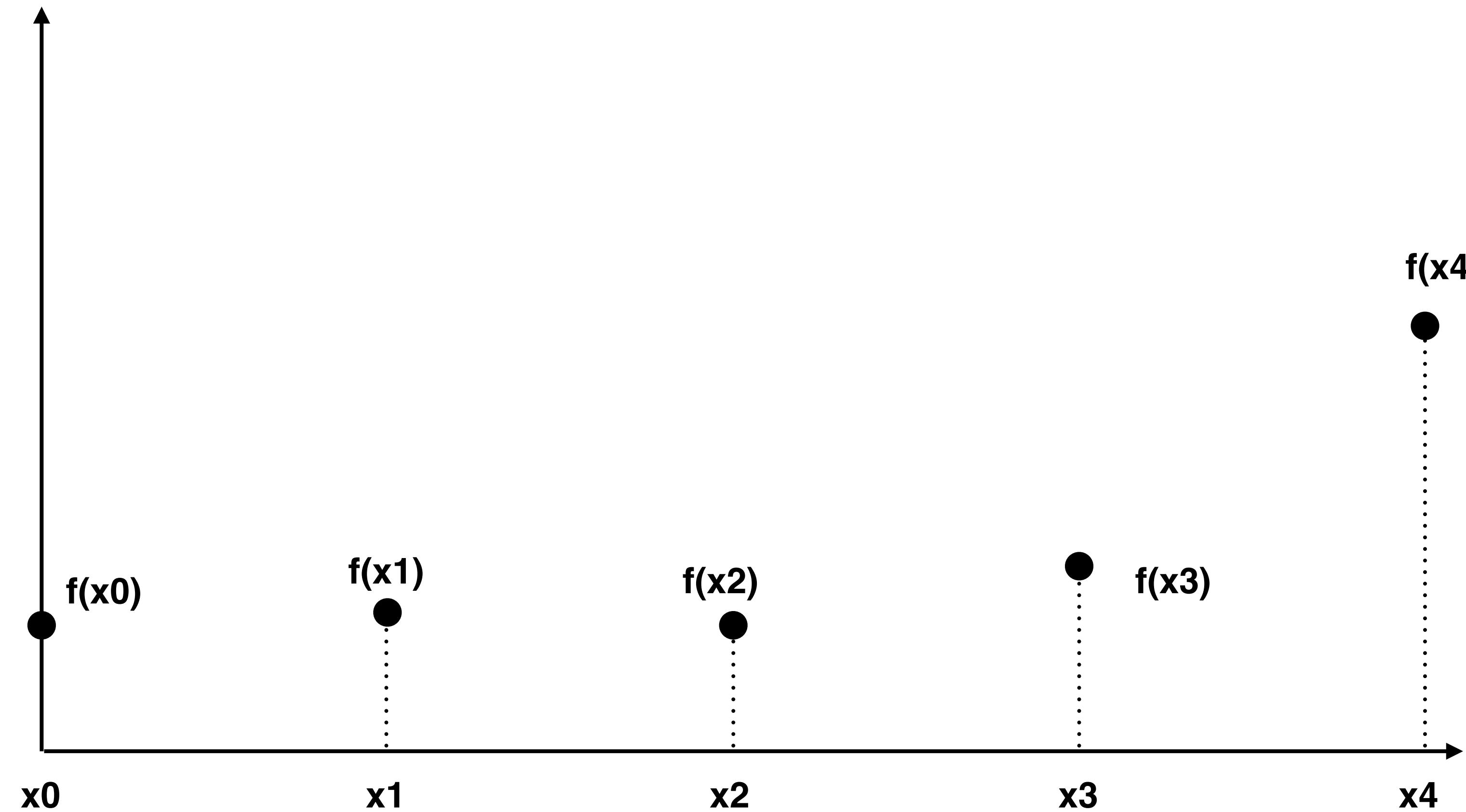
Computer Graphics: Rendering, Geometry, and Image Manipulation
Stanford CS248A, Winter 2026

Review:
Summarizing what we learned last time

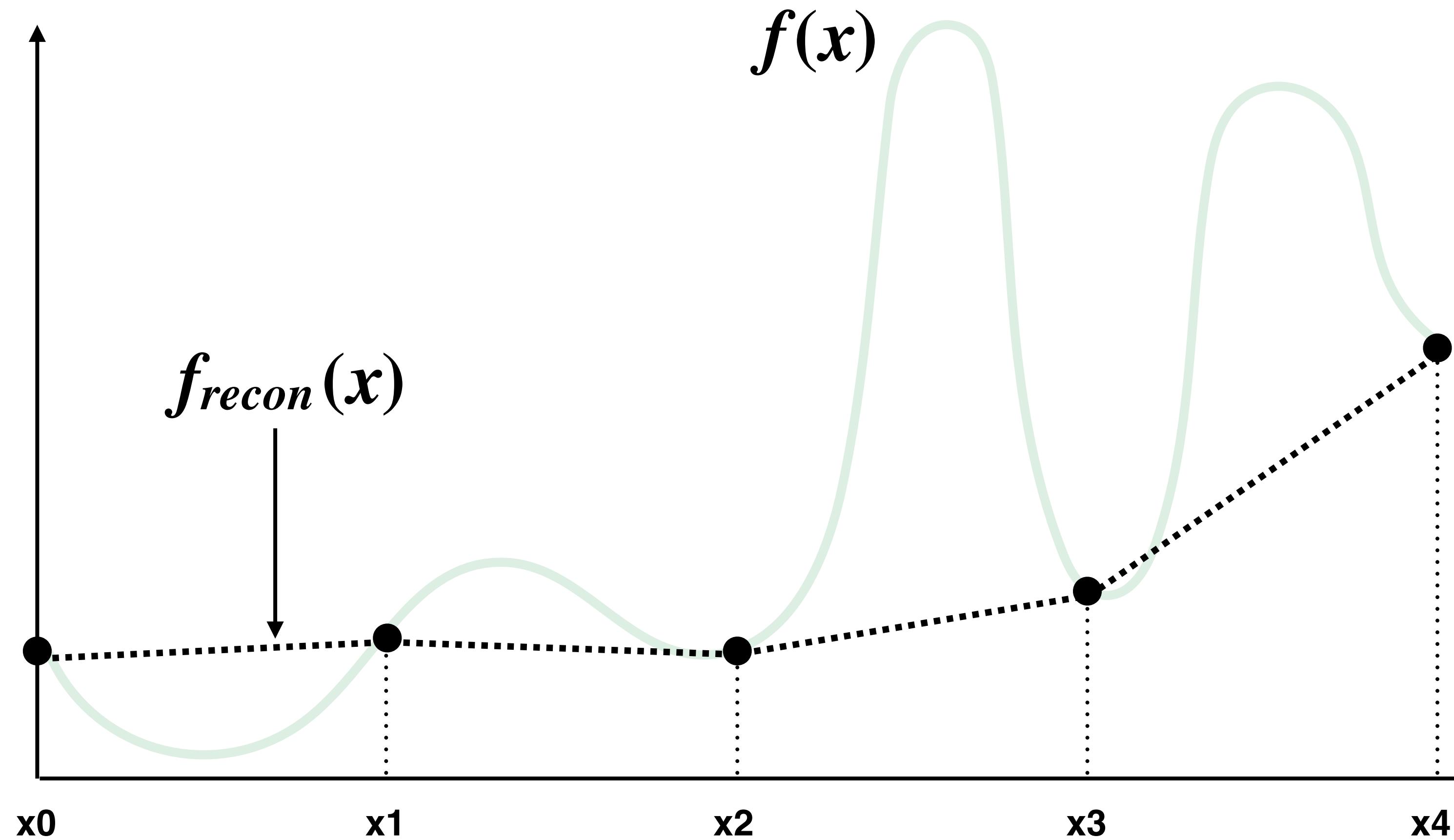
Sampling = taking measurements of a signal



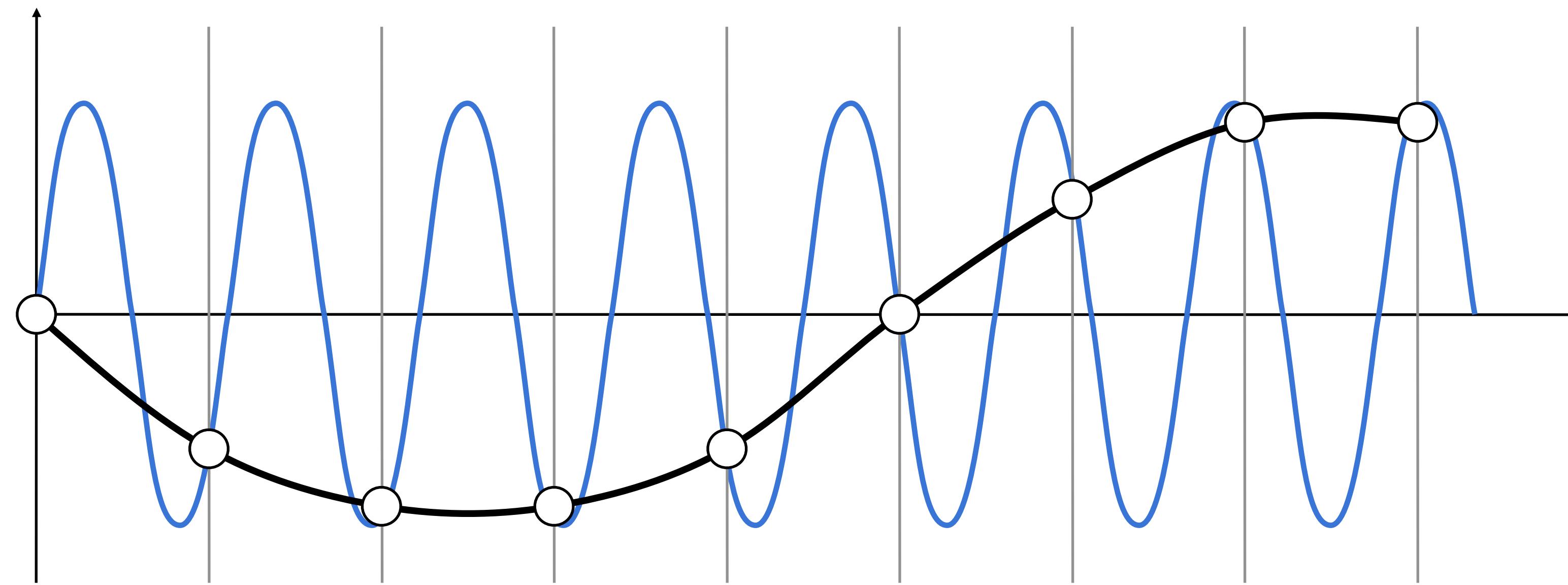
Sampling: taking measurements of a signal



Reconstruction: approximating continuous signal from the discrete set of measurements



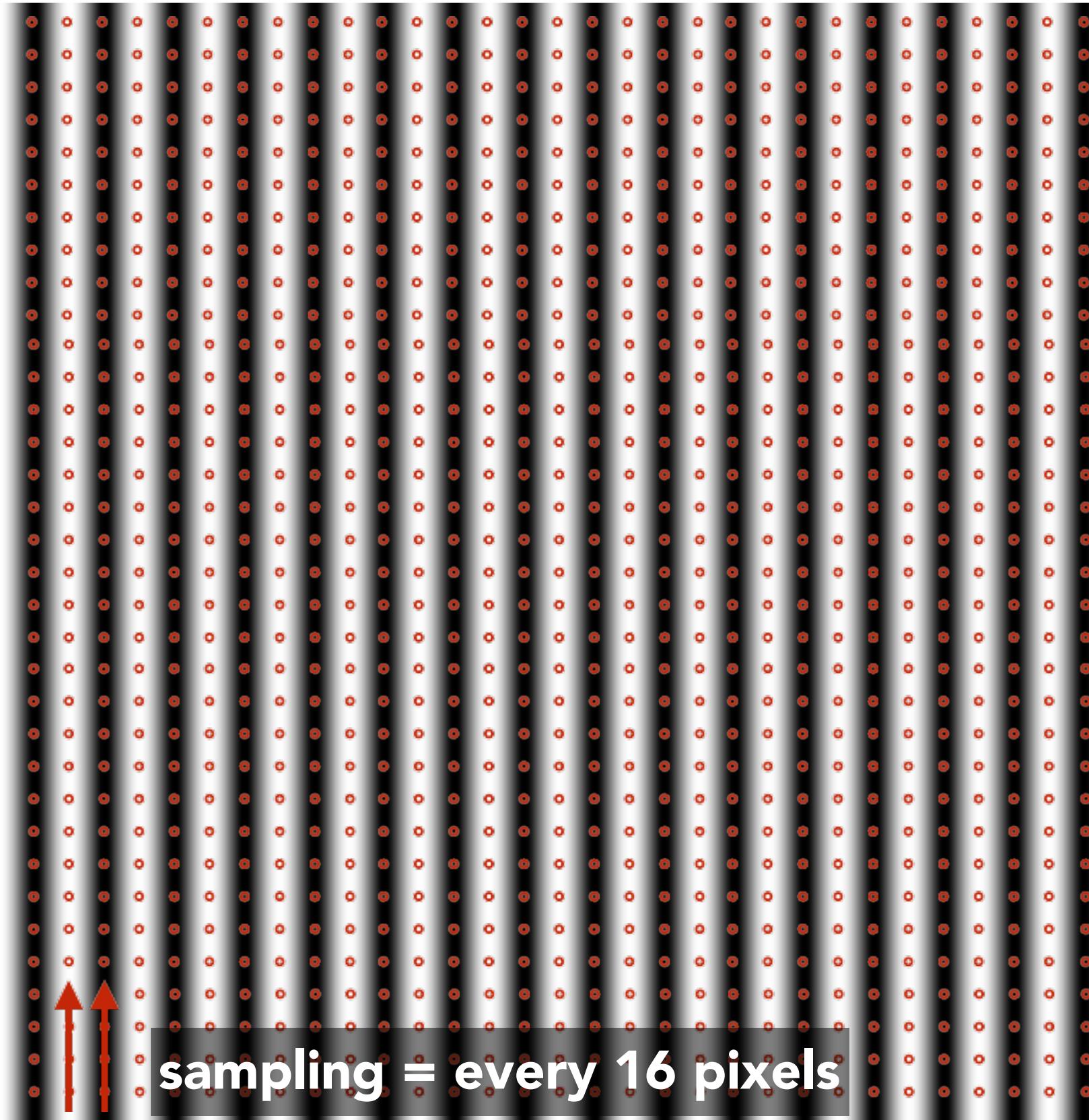
Sampling a signal too sparsely can result in aliasing



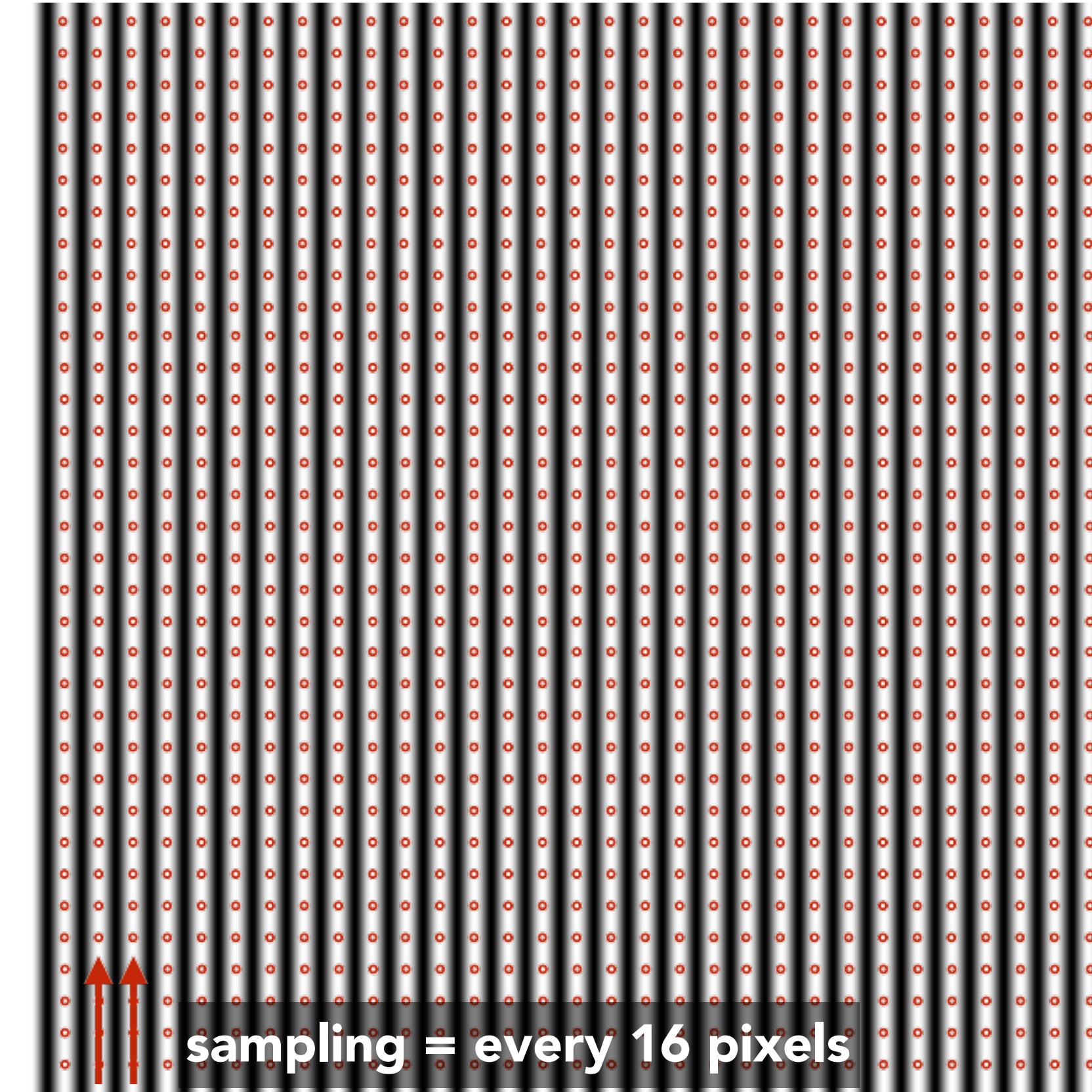
High-frequency signal is insufficiently sampled, so samples erroneously appear to be from a low-frequency signal

Two frequencies that are indistinguishable at a given sampling rate are called “aliases”

Sampling a signal too sparsely can result in aliasing



$\sin(2\pi/32)x$
frequency 1/32; 32 pixels per cycle



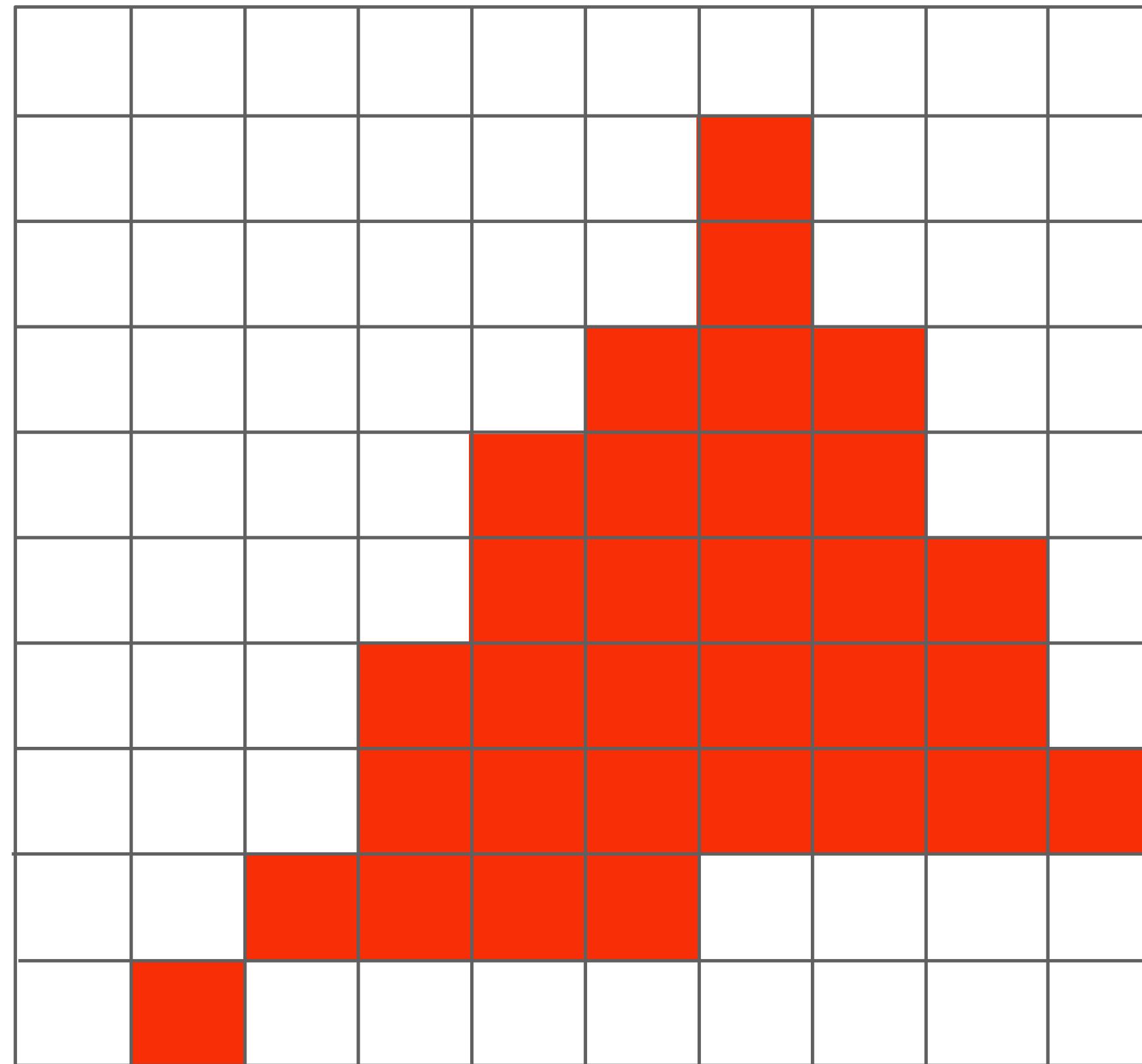
$\sin(2\pi/16)x$
frequency 1/16; 16 pixels per cycle

Sampling a signal too sparsely can result in aliasing



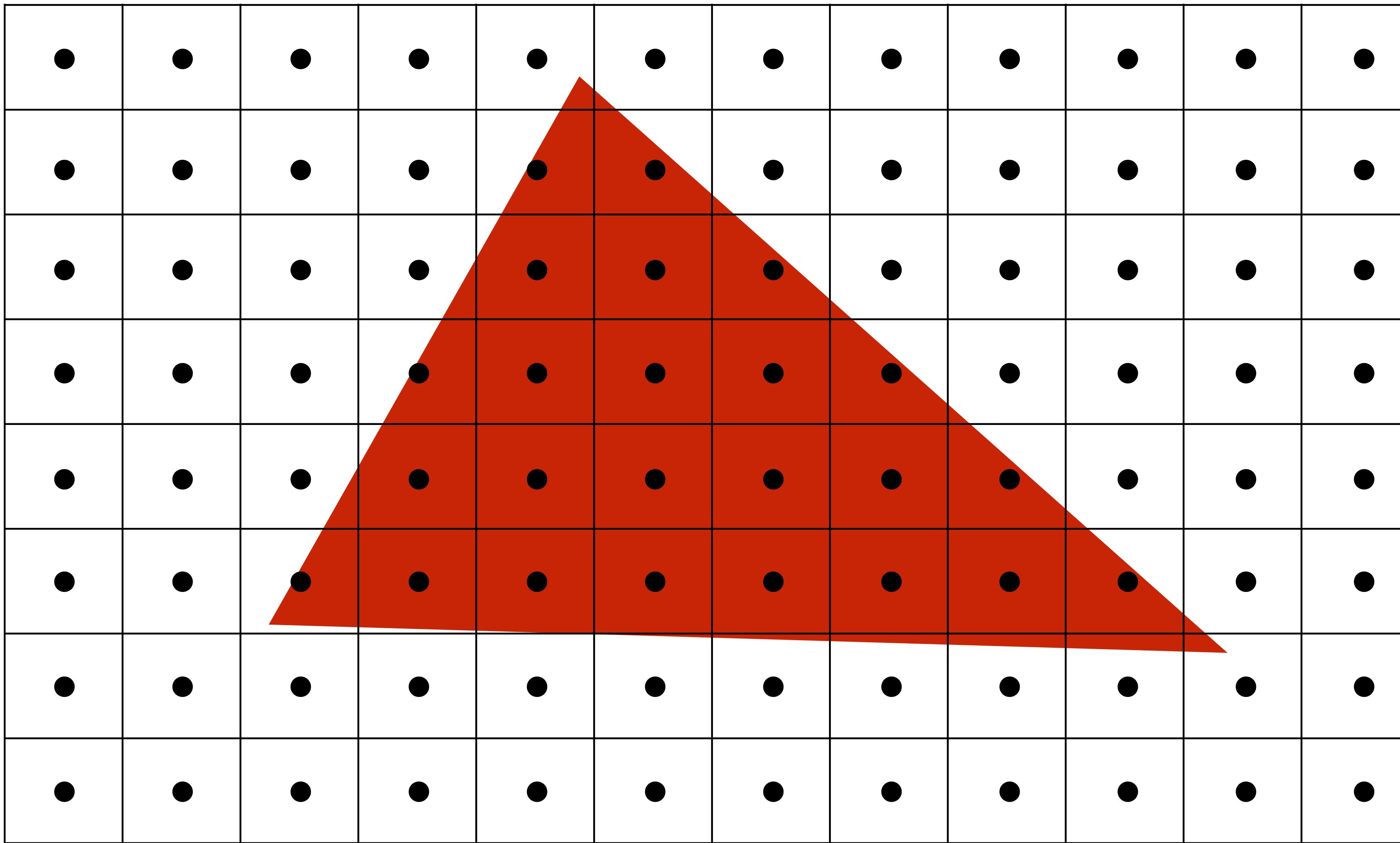
Shutter Speed = 1/800s

Sampling a signal too sparsely can result in aliasing

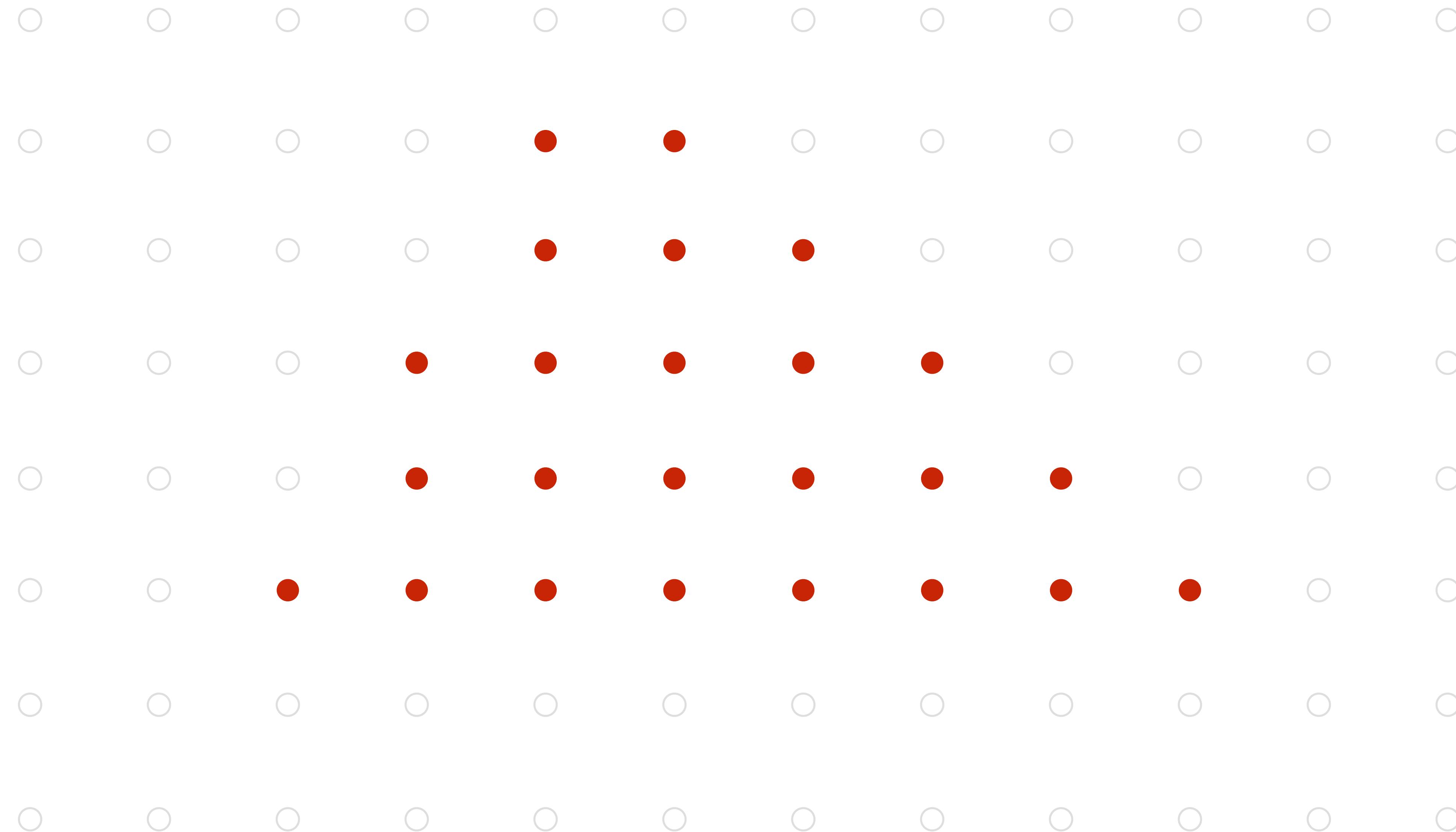


Jaggies!

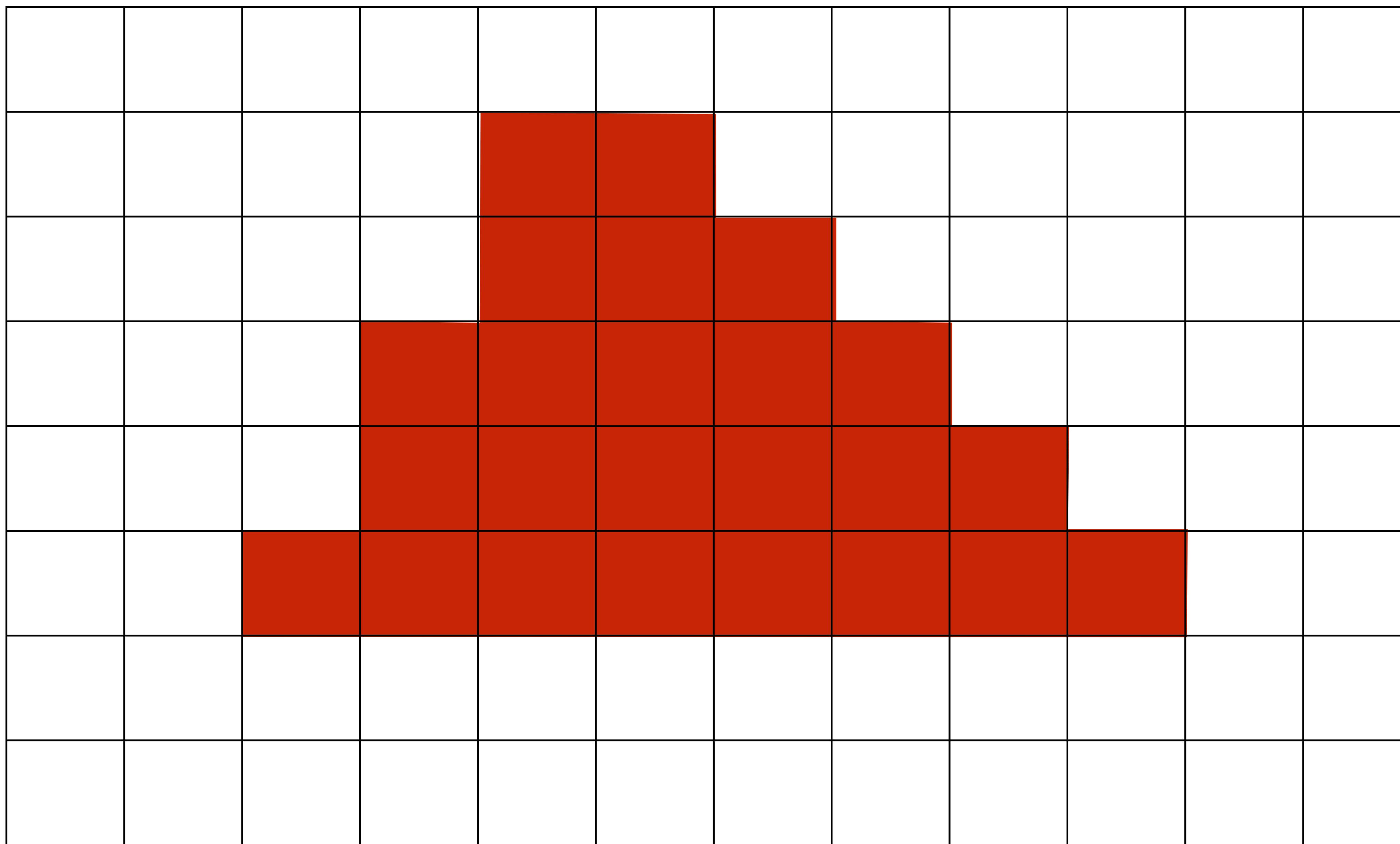
Sampling a triangle (1 sample/pixel)



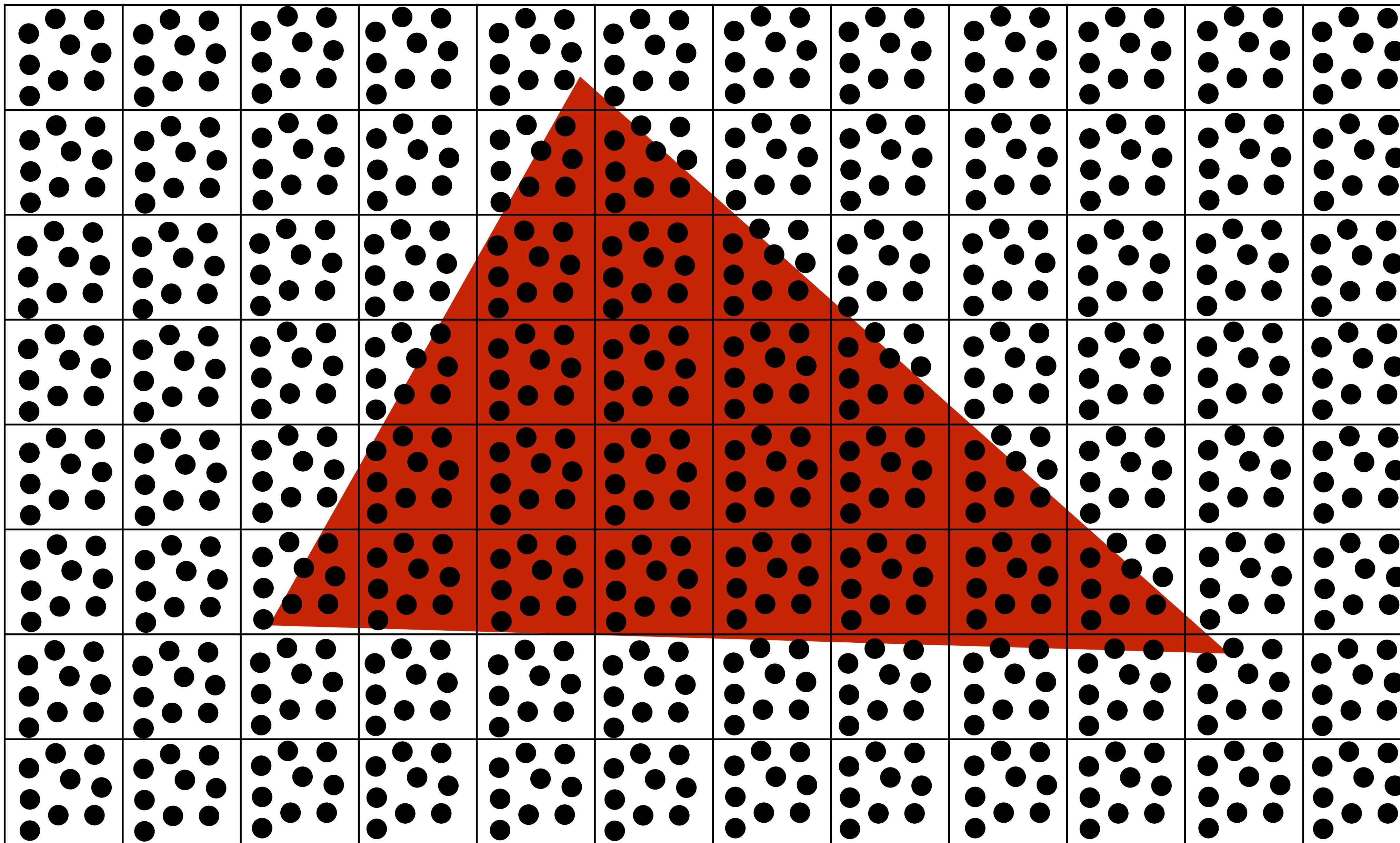
Results of sampling (1 sample/pixel)



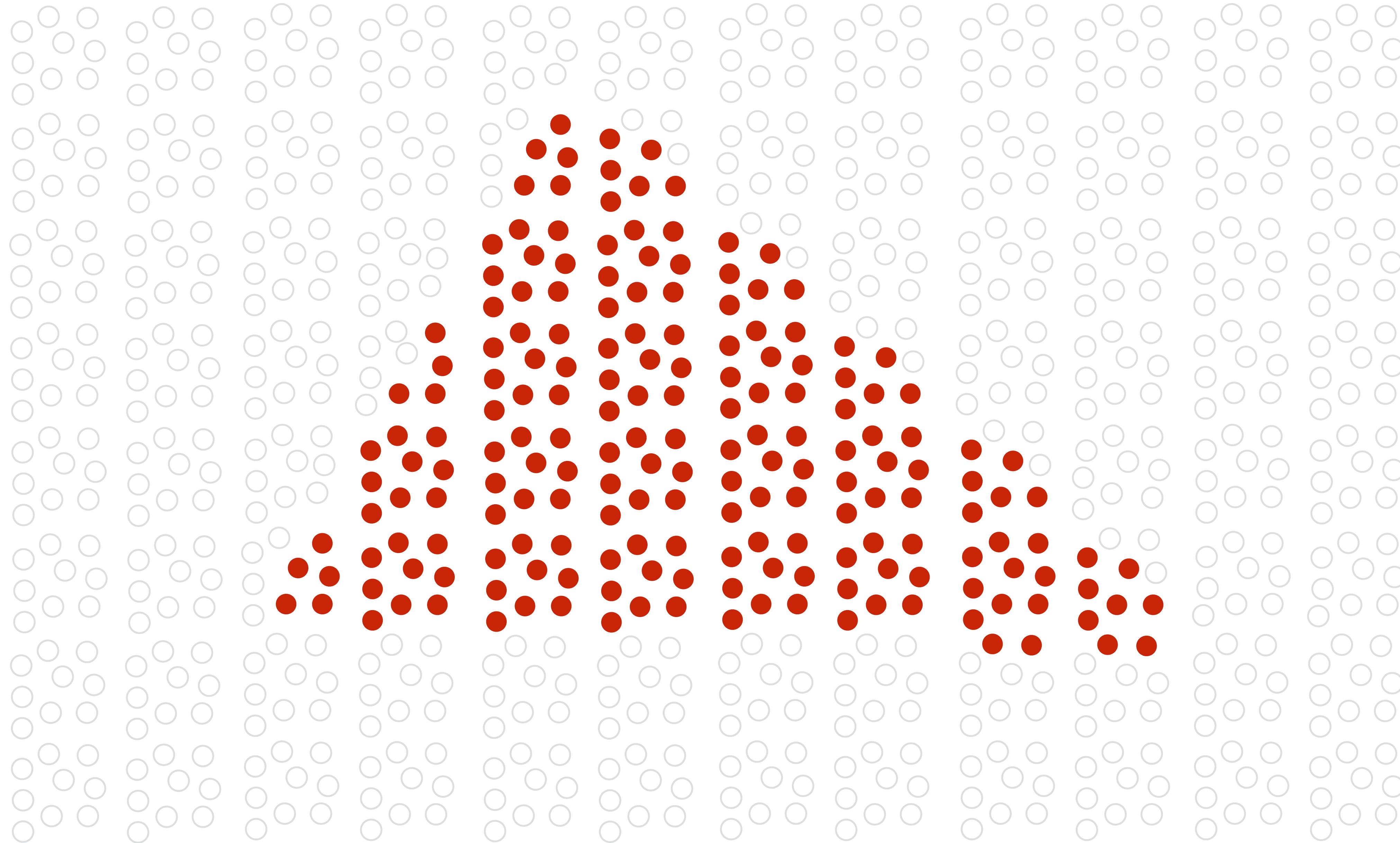
Displaying the results of sampling (square pixels on a display = nearest neighbor reconstruction)



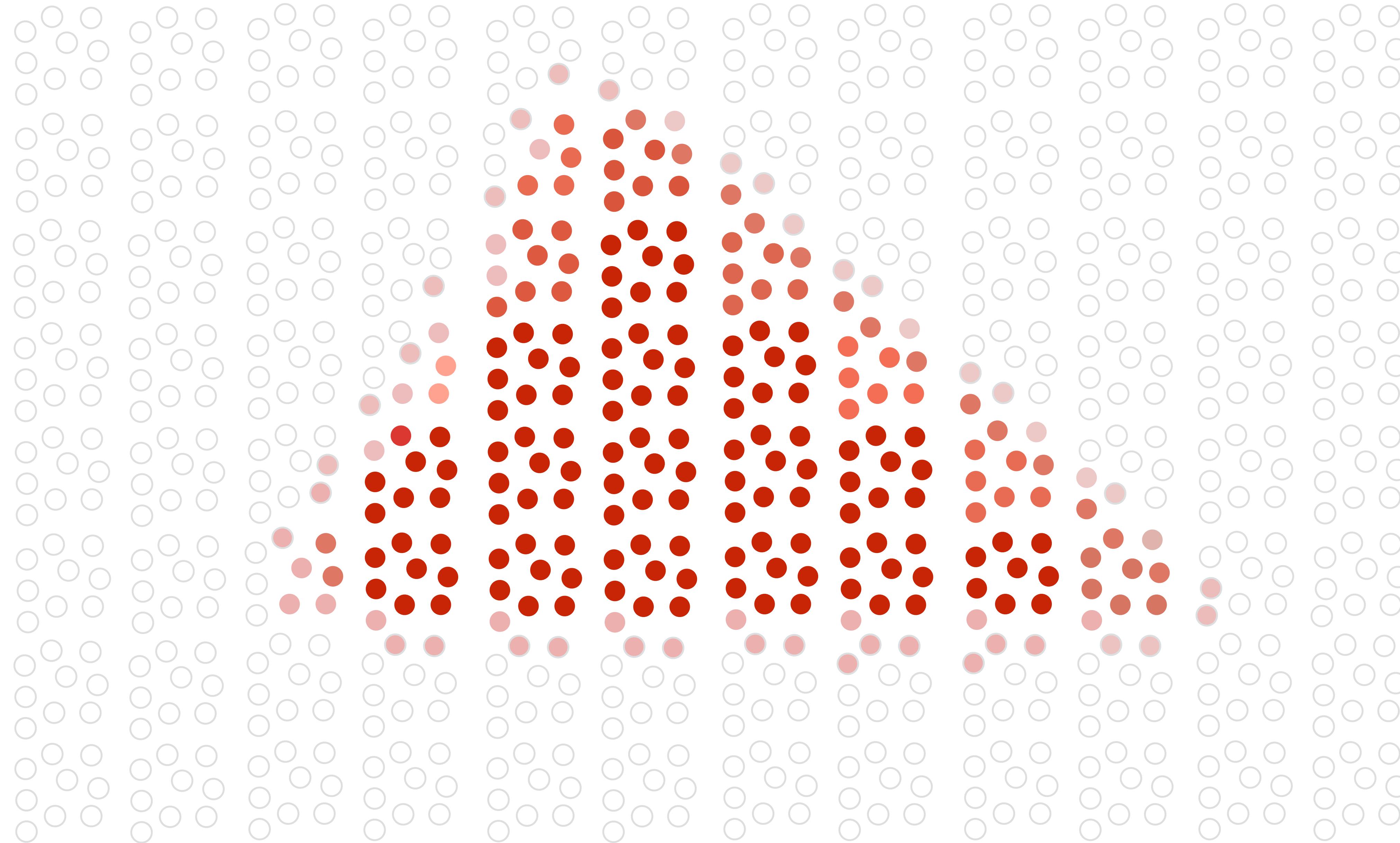
Sampling a triangle (9 samples/pixel)



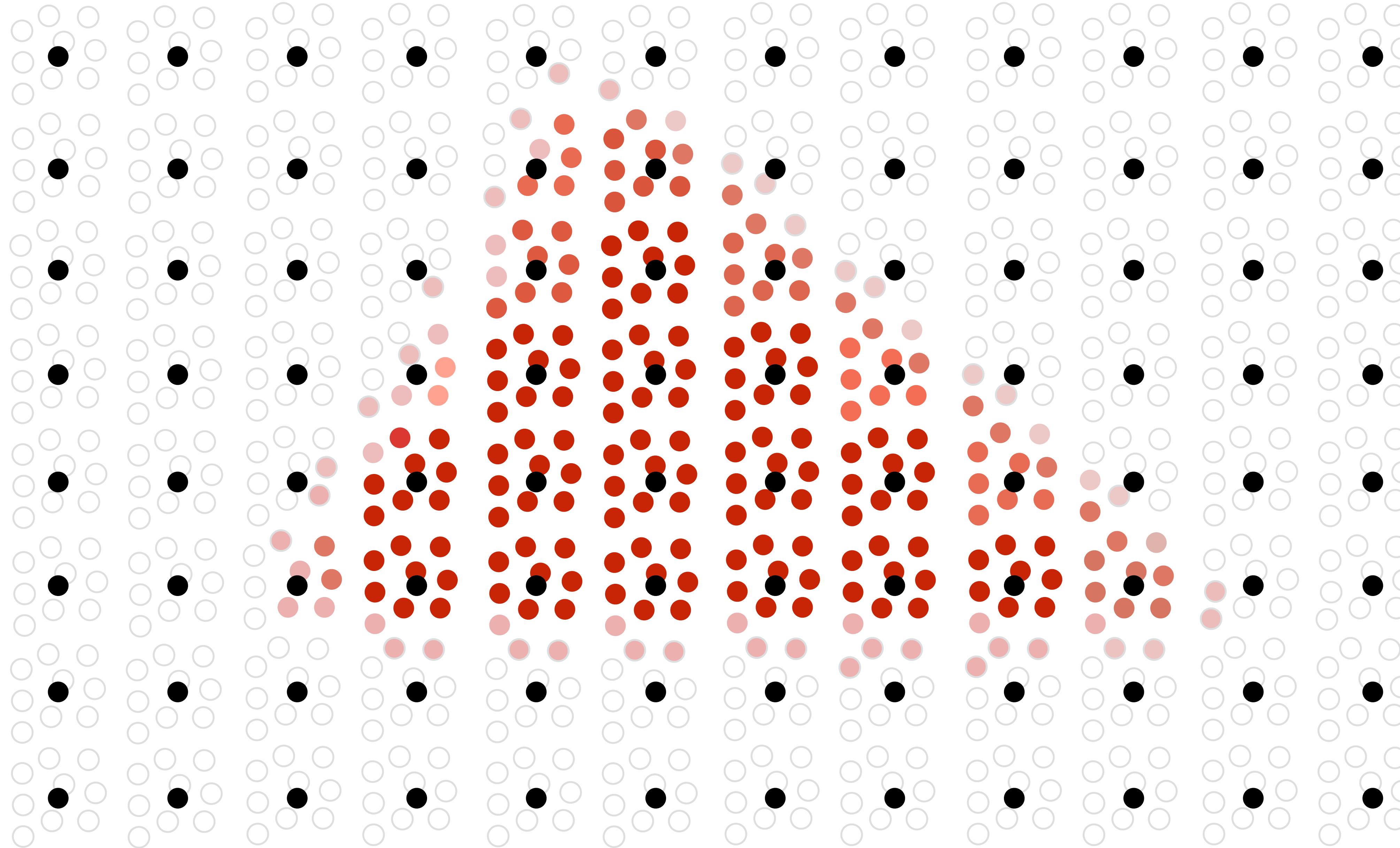
Results of sampling (9 samples/pixel)



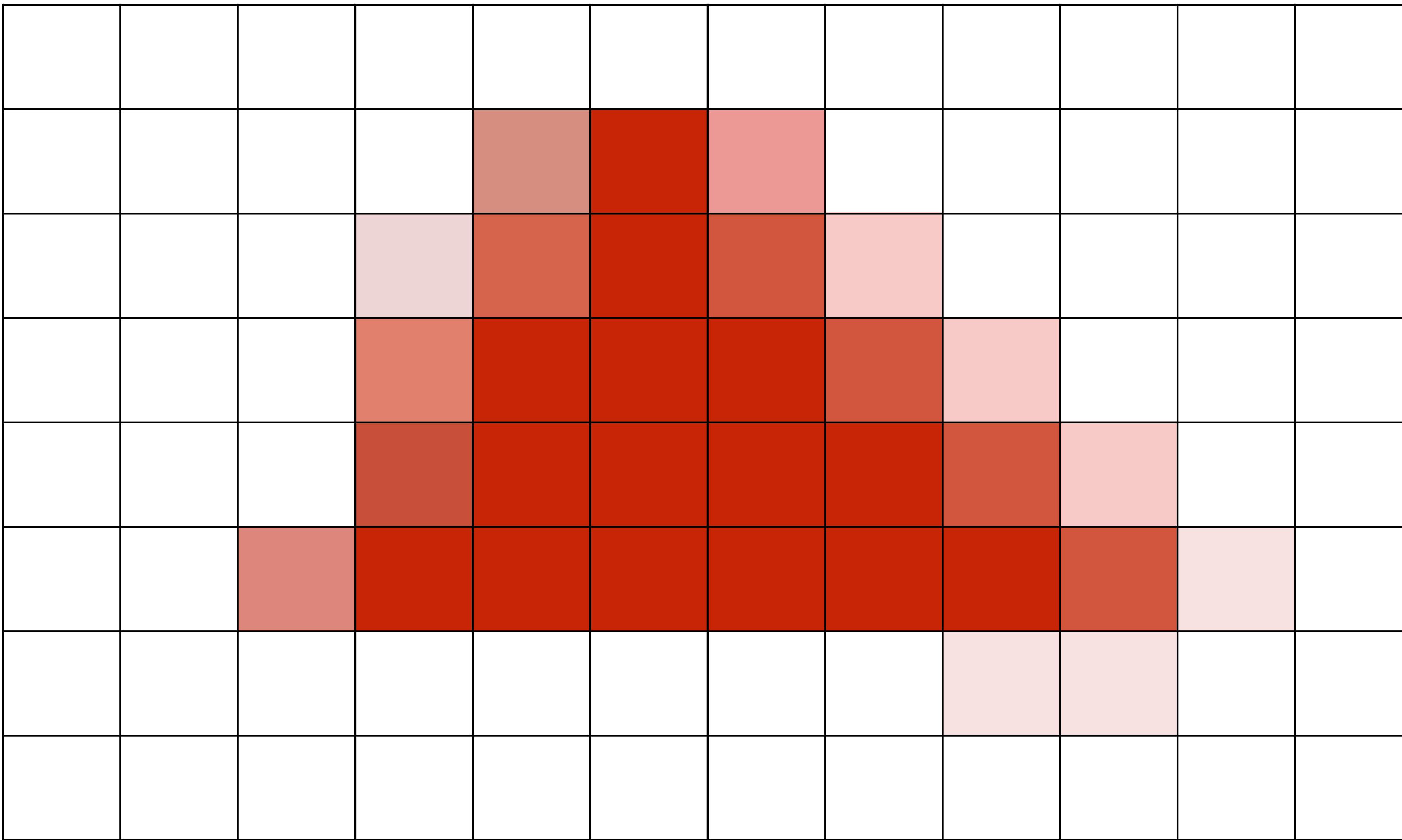
Blurring (“filtering”) the densely sampled result (remove high frequencies)



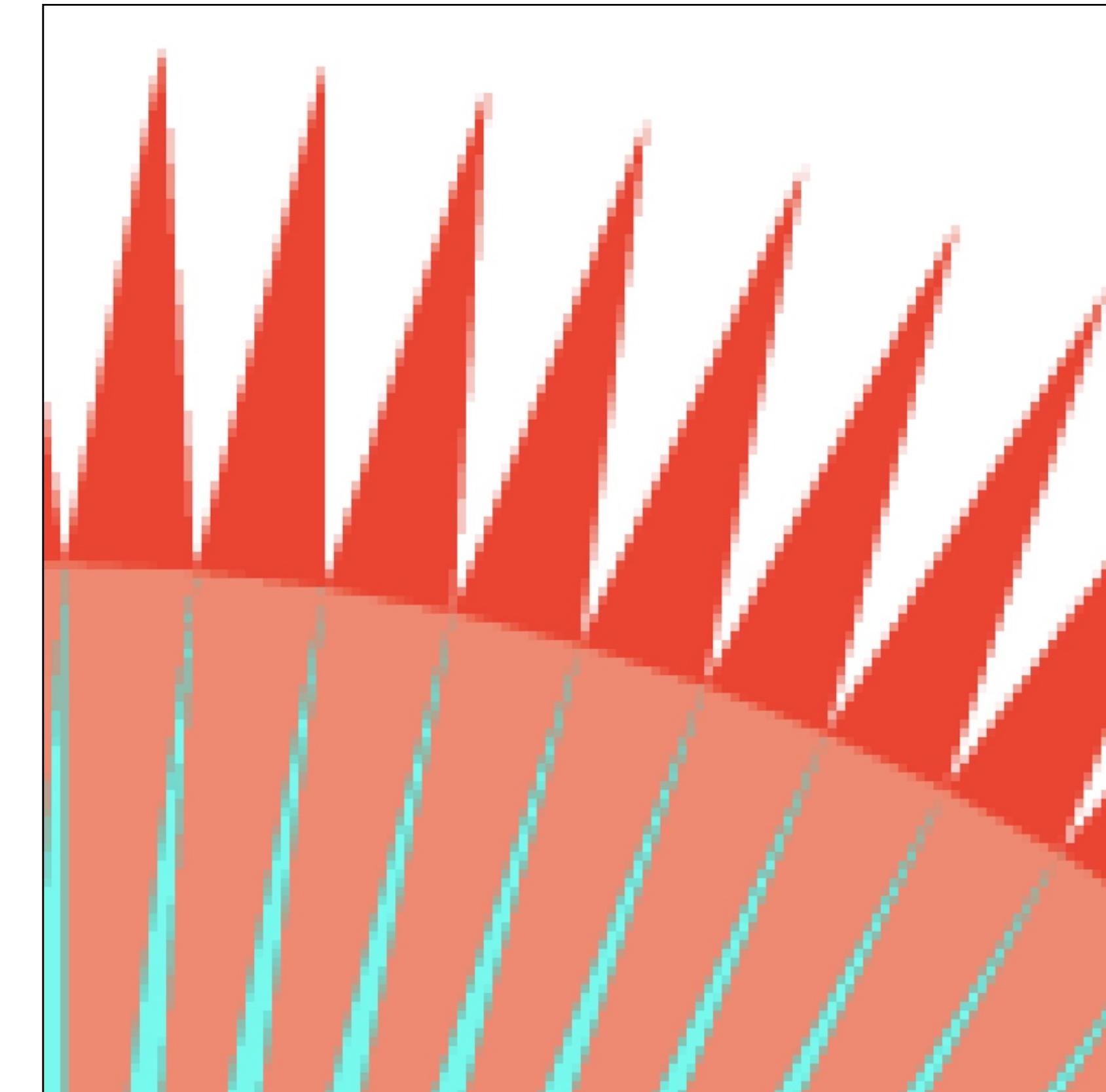
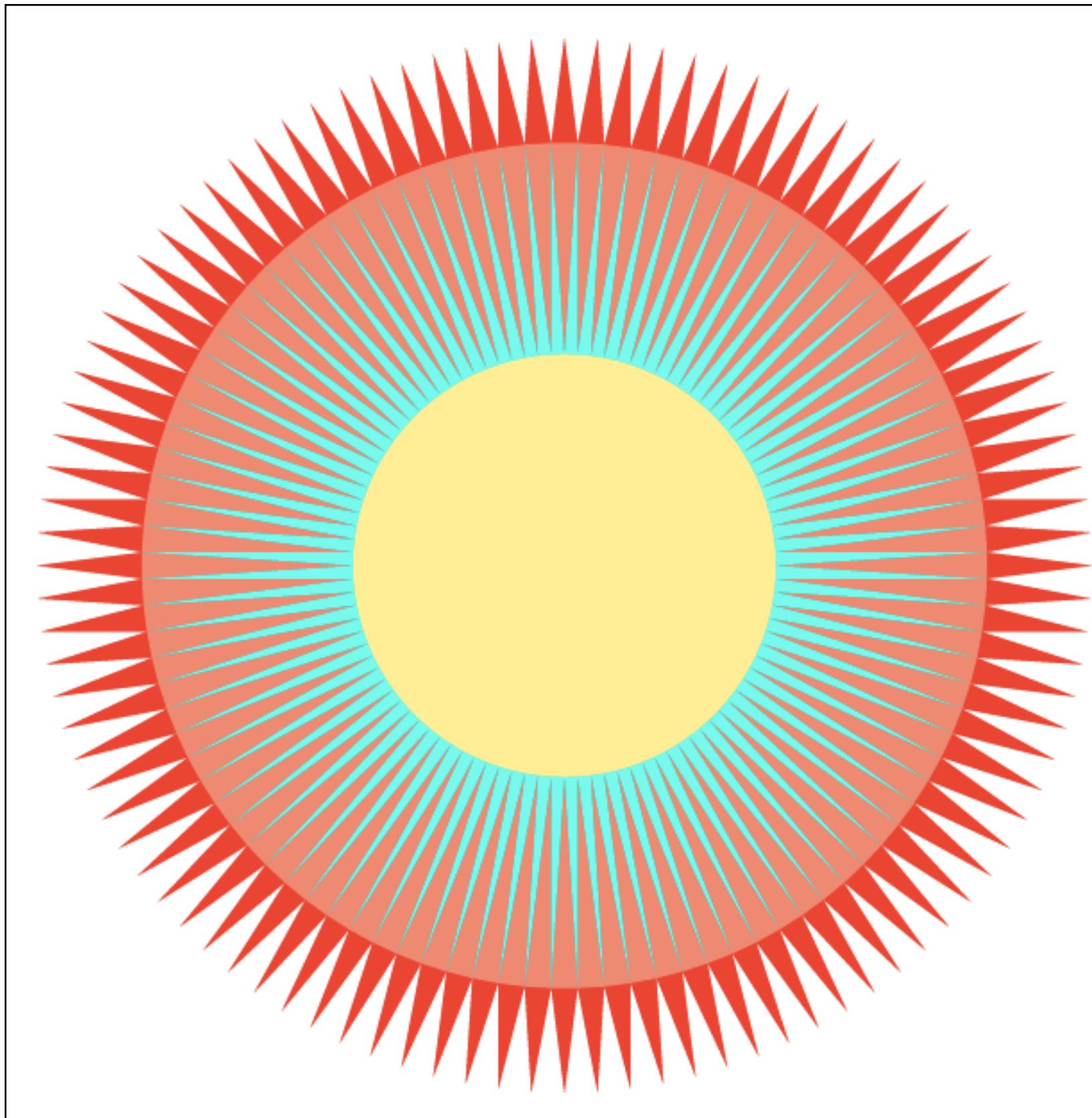
Resample blurred signal at pixel centers (one sample per pixel)



Displaying the results of sampling (square pixels on a display)

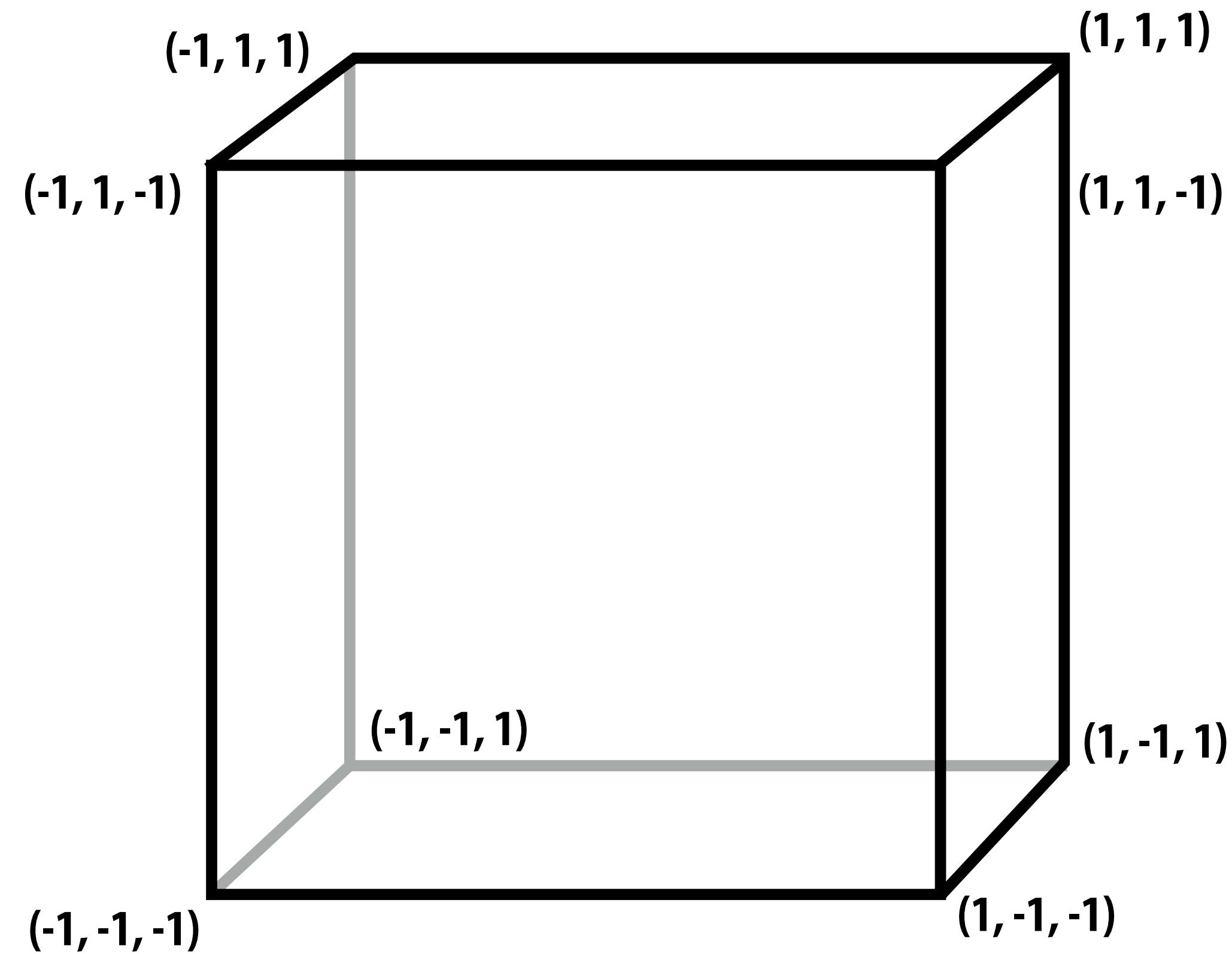


Example: anti-aliased results

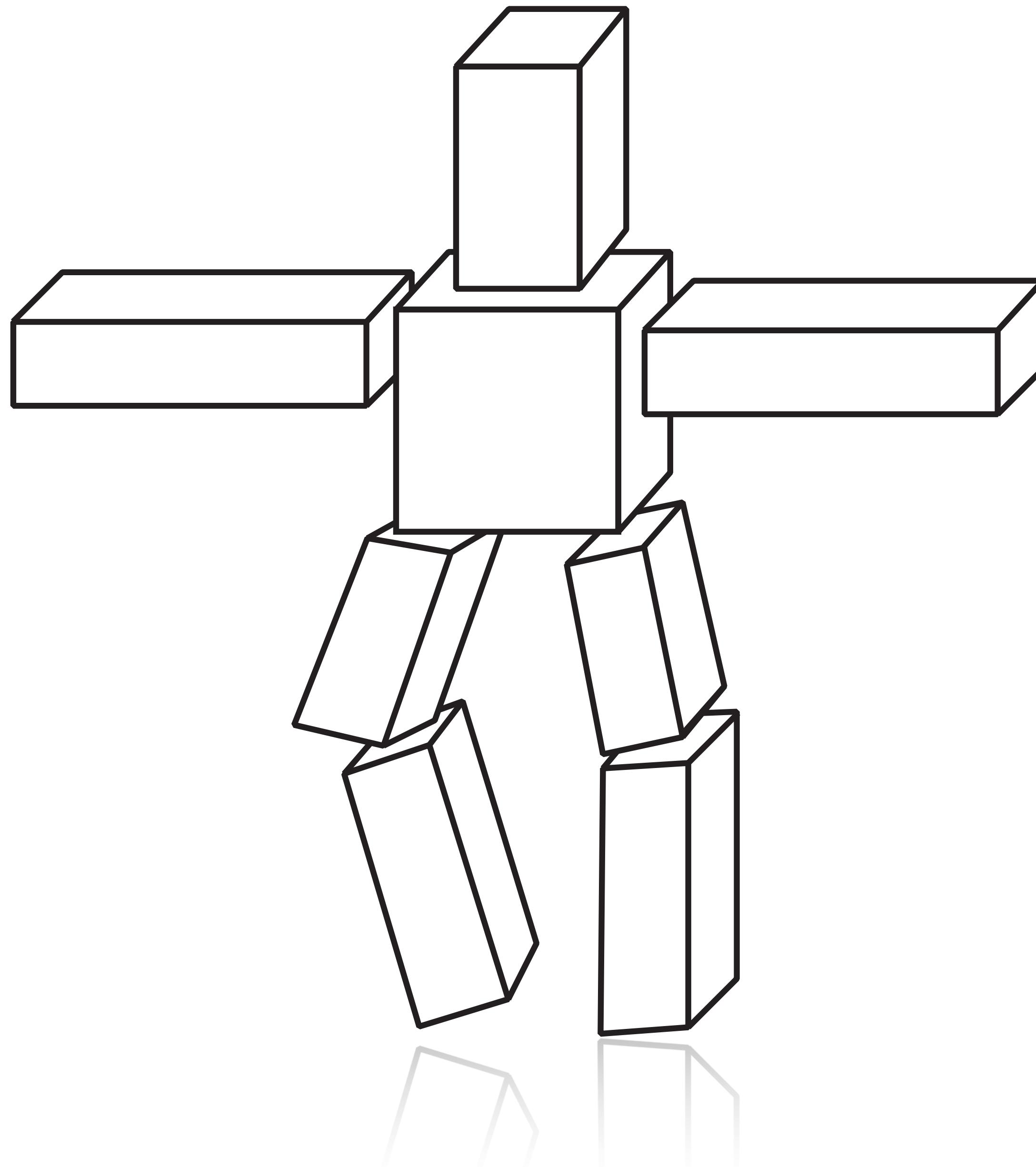


Transformations

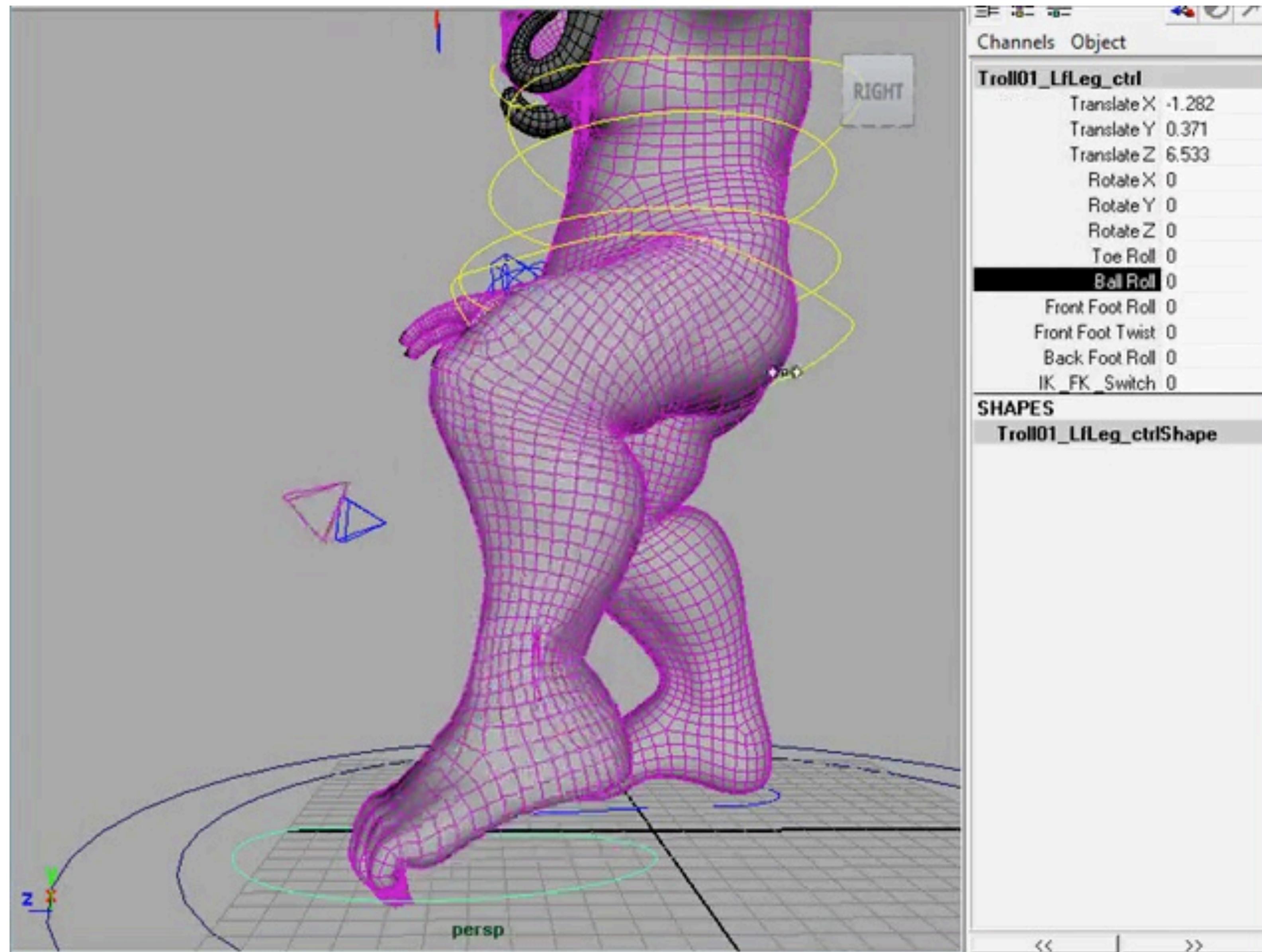
A cube, centered at the origin, with faces of size 2 x 2



Consider drawing a cube person



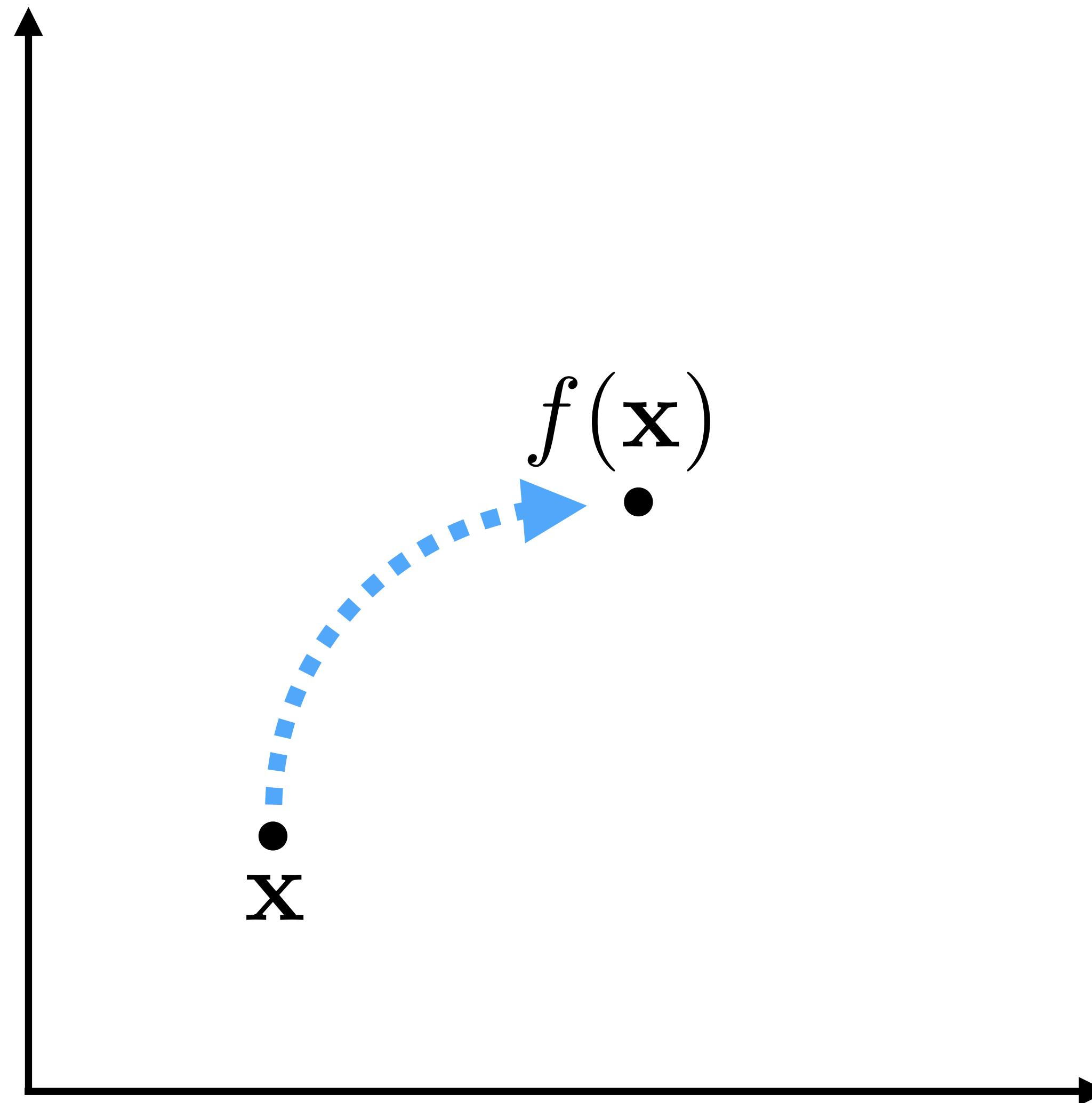
Transformations in character rigging



Transformations for geometry instancing



Basic idea: f transforms x to $f(x)$



Linear transformations

- What does *linear* mean?

$$f(\mathbf{x} + \mathbf{y}) = f(\mathbf{x}) + f(\mathbf{y})$$

$$f(a\mathbf{x}) = af(\mathbf{x})$$

- Composition of linear transformations is linear
 - Leads to uniform representation of many types of transformations
- As we will see... linear transformations are cheap to compute

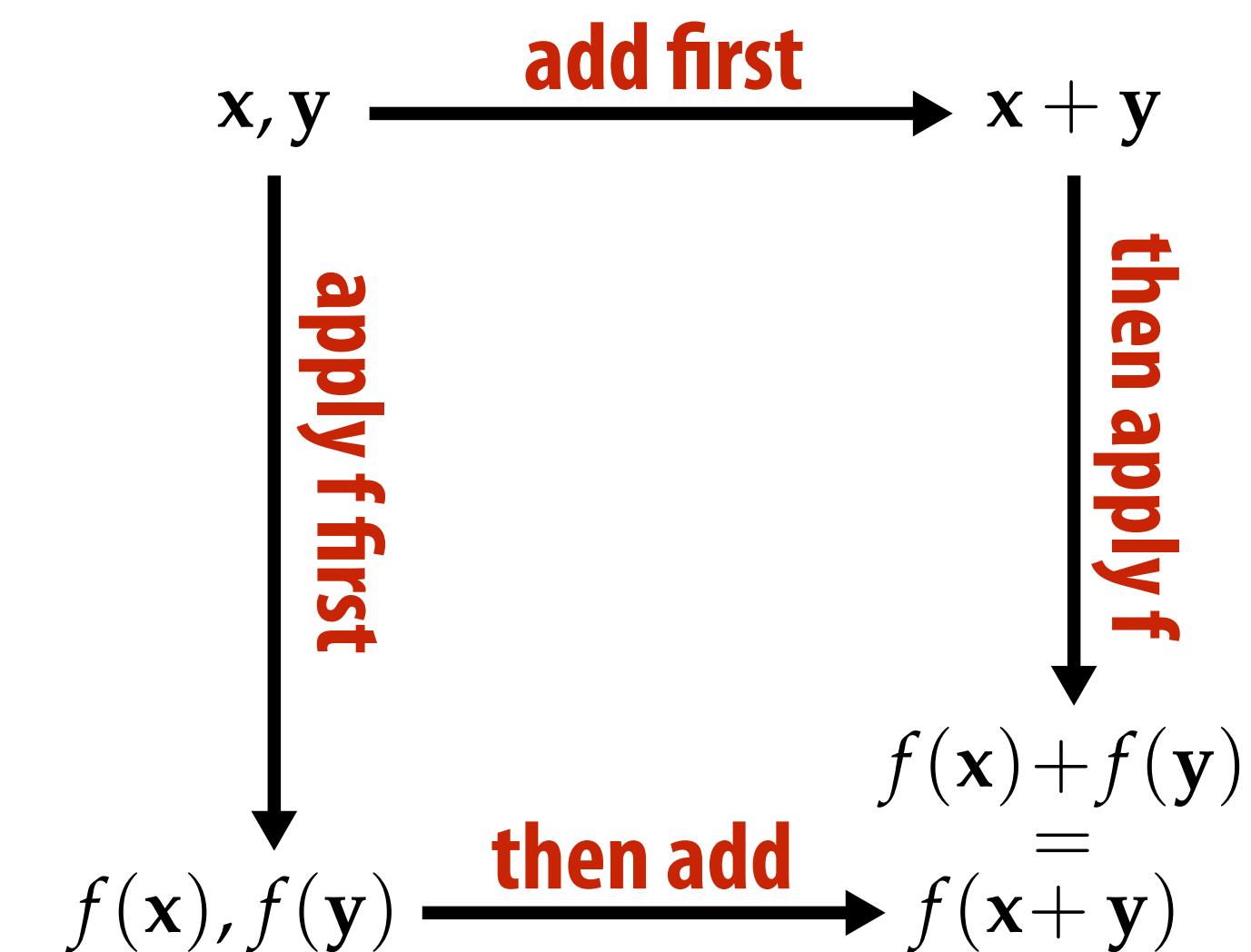
Linear transformations

$$f(\mathbf{x} + \mathbf{y}) = f(\mathbf{x}) + f(\mathbf{y})$$

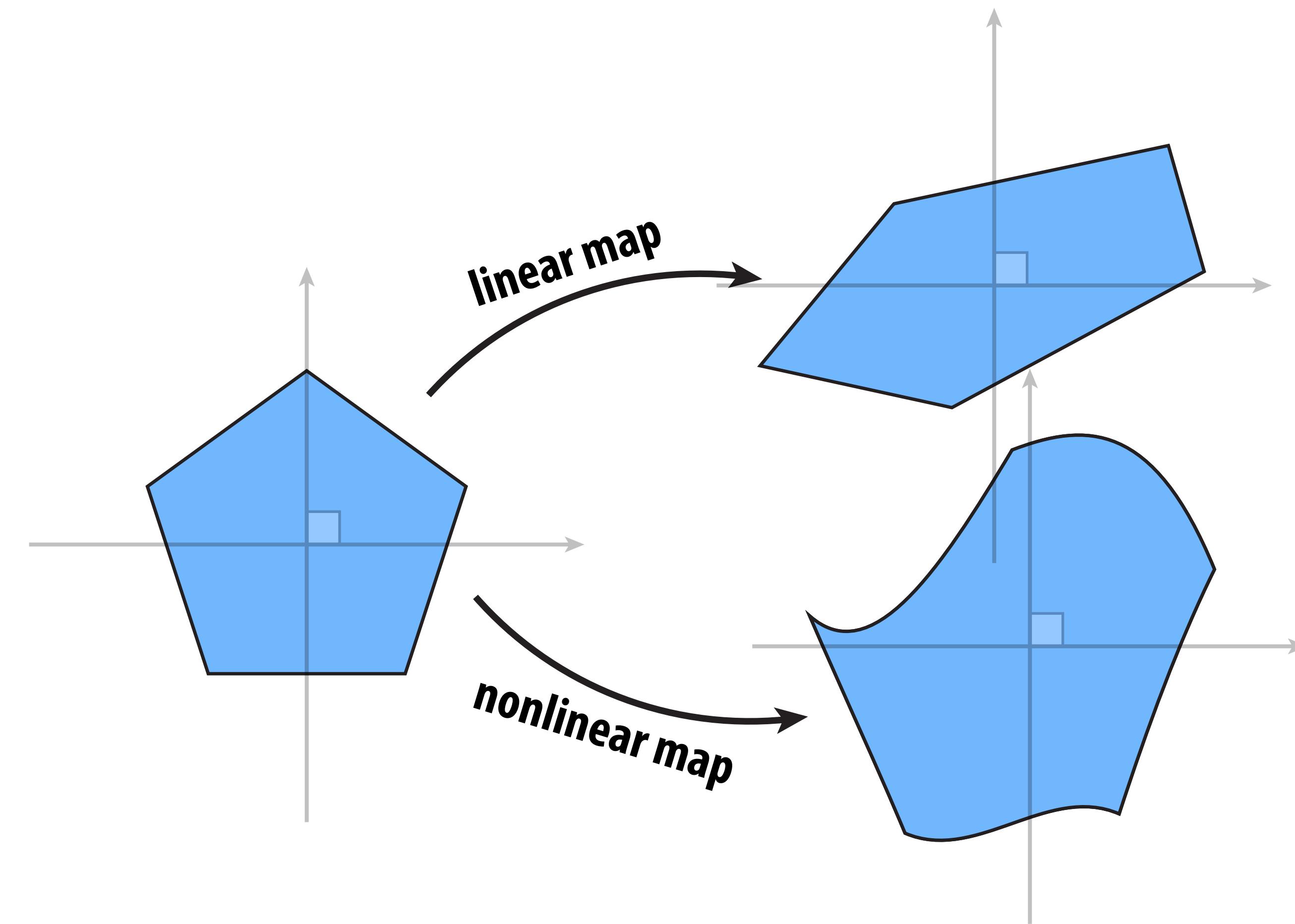
$$f(a\mathbf{x}) = af(\mathbf{x})$$

In other words: if it doesn't matter whether we add the vectors and then apply the map, or apply the map and then add the vectors

(and likewise for scaling)

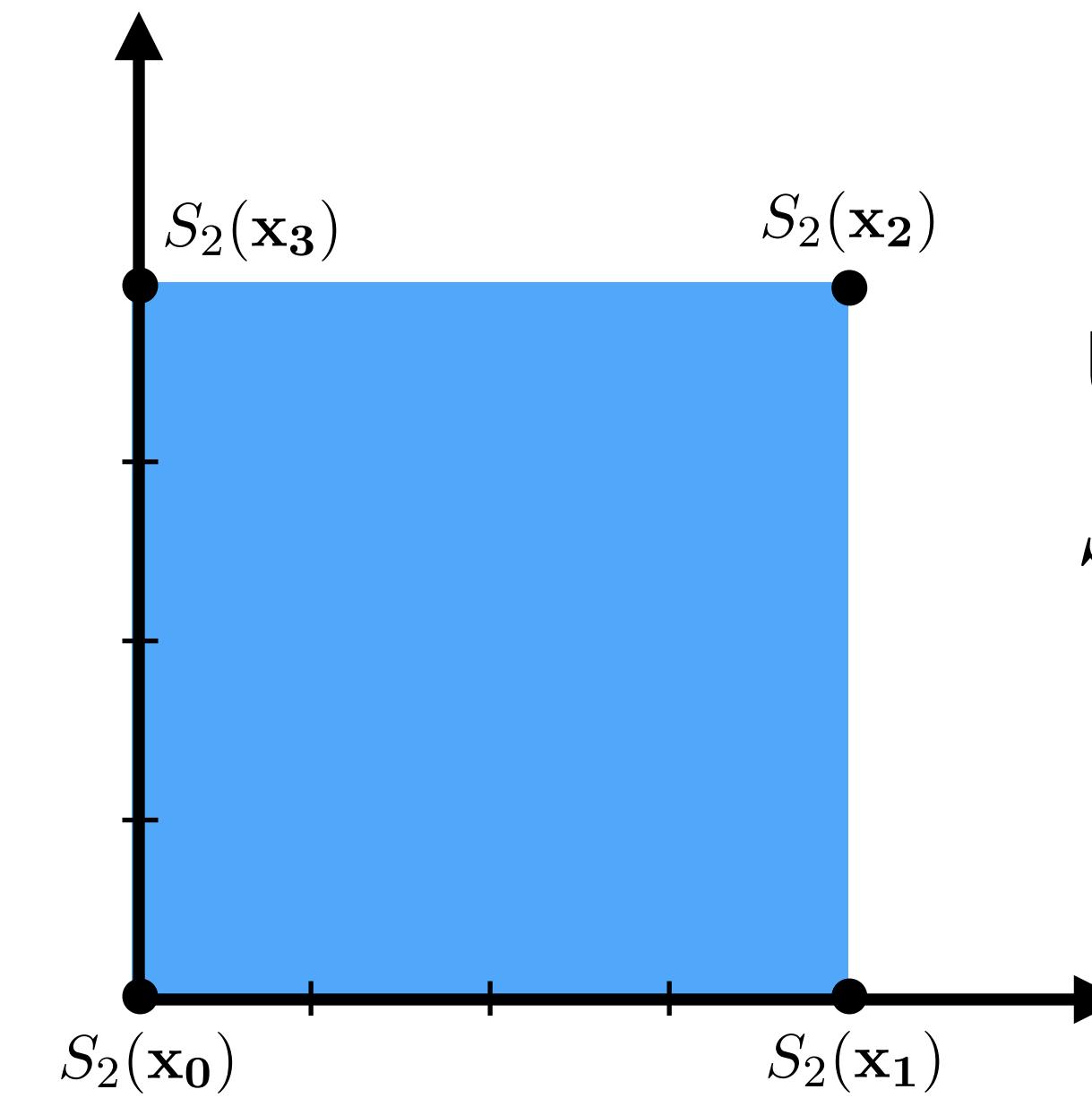
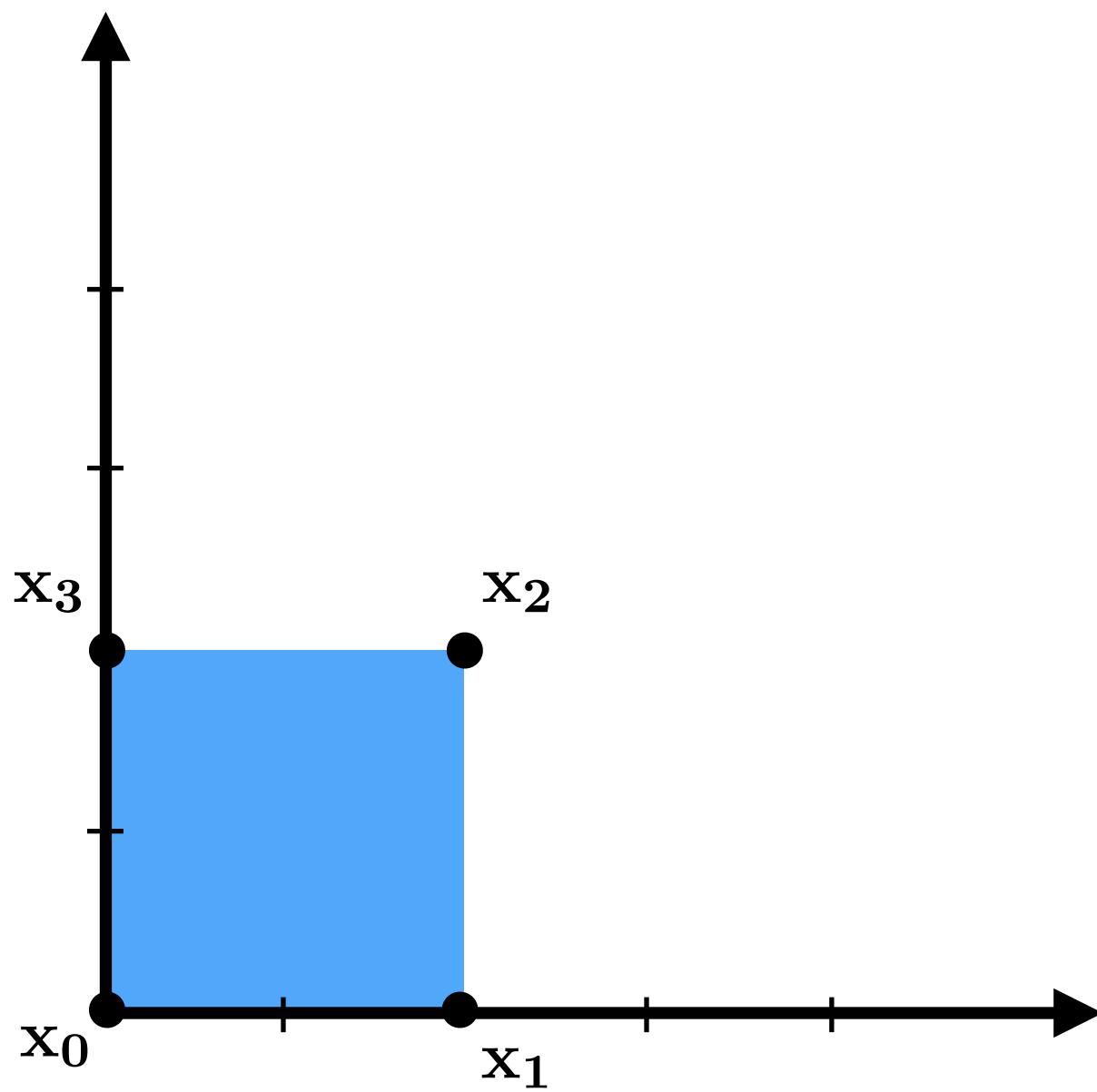


Linear transforms/maps—visualized

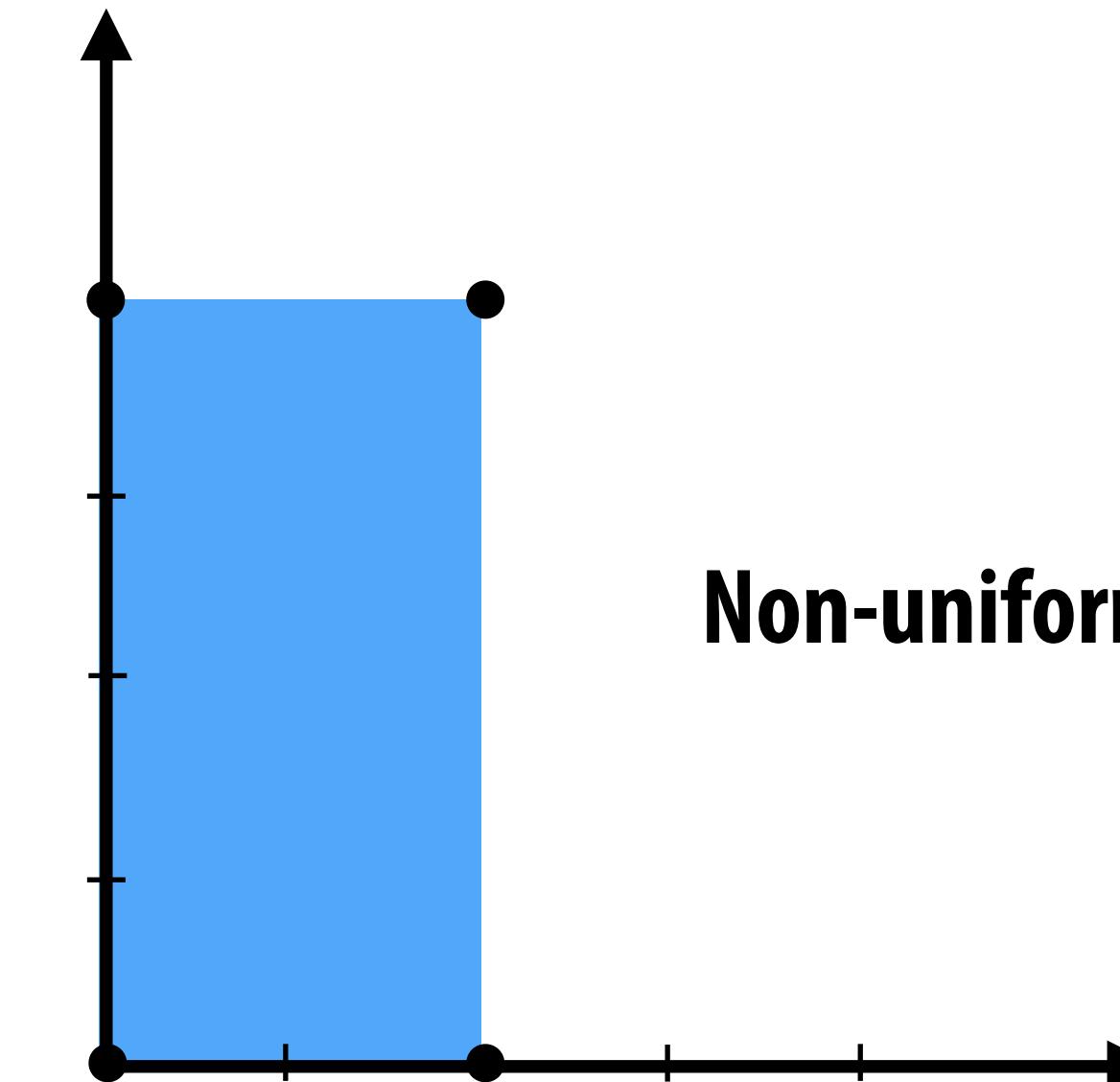


Key idea: *linear maps take lines to lines*

Scale transform

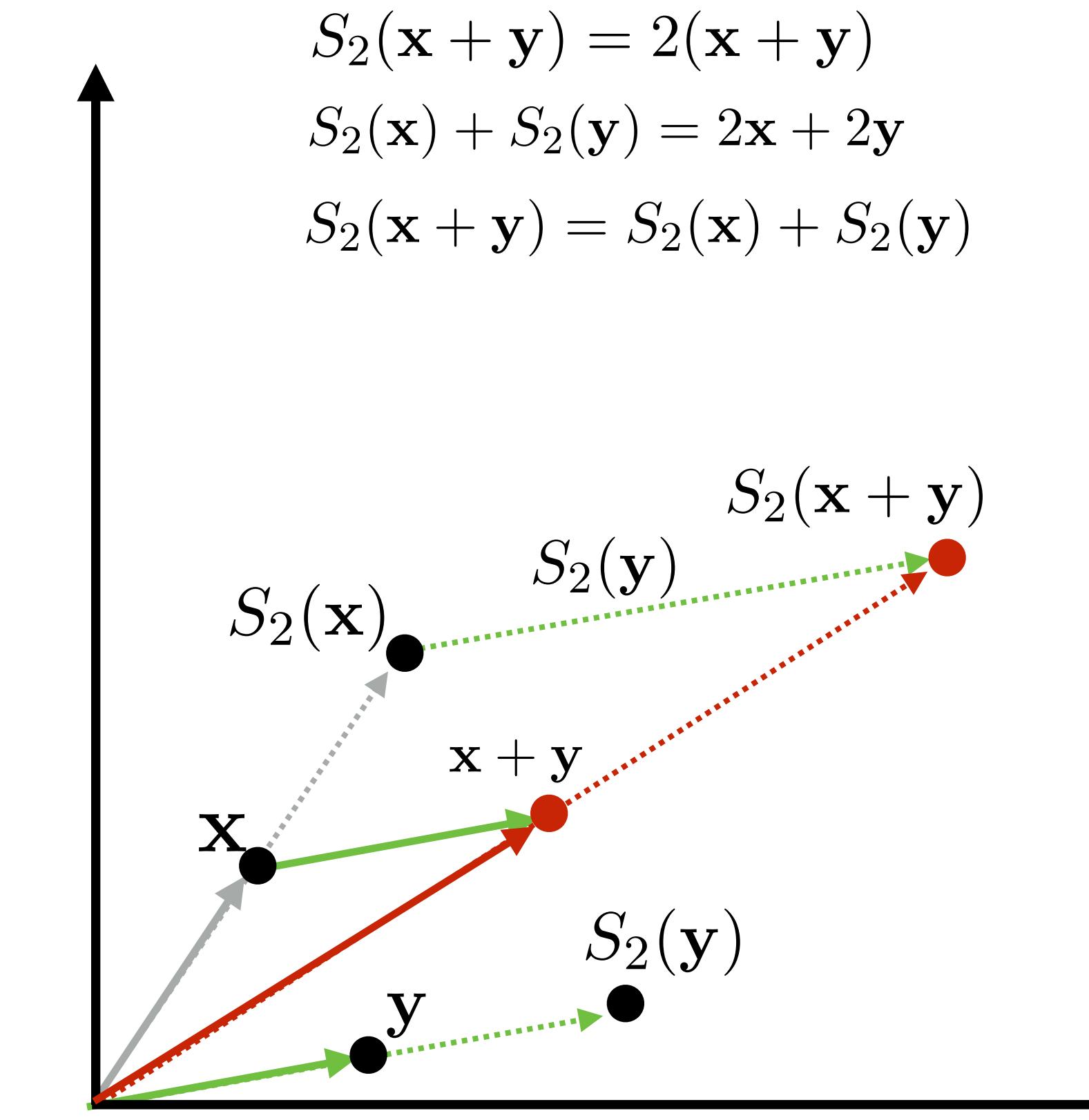
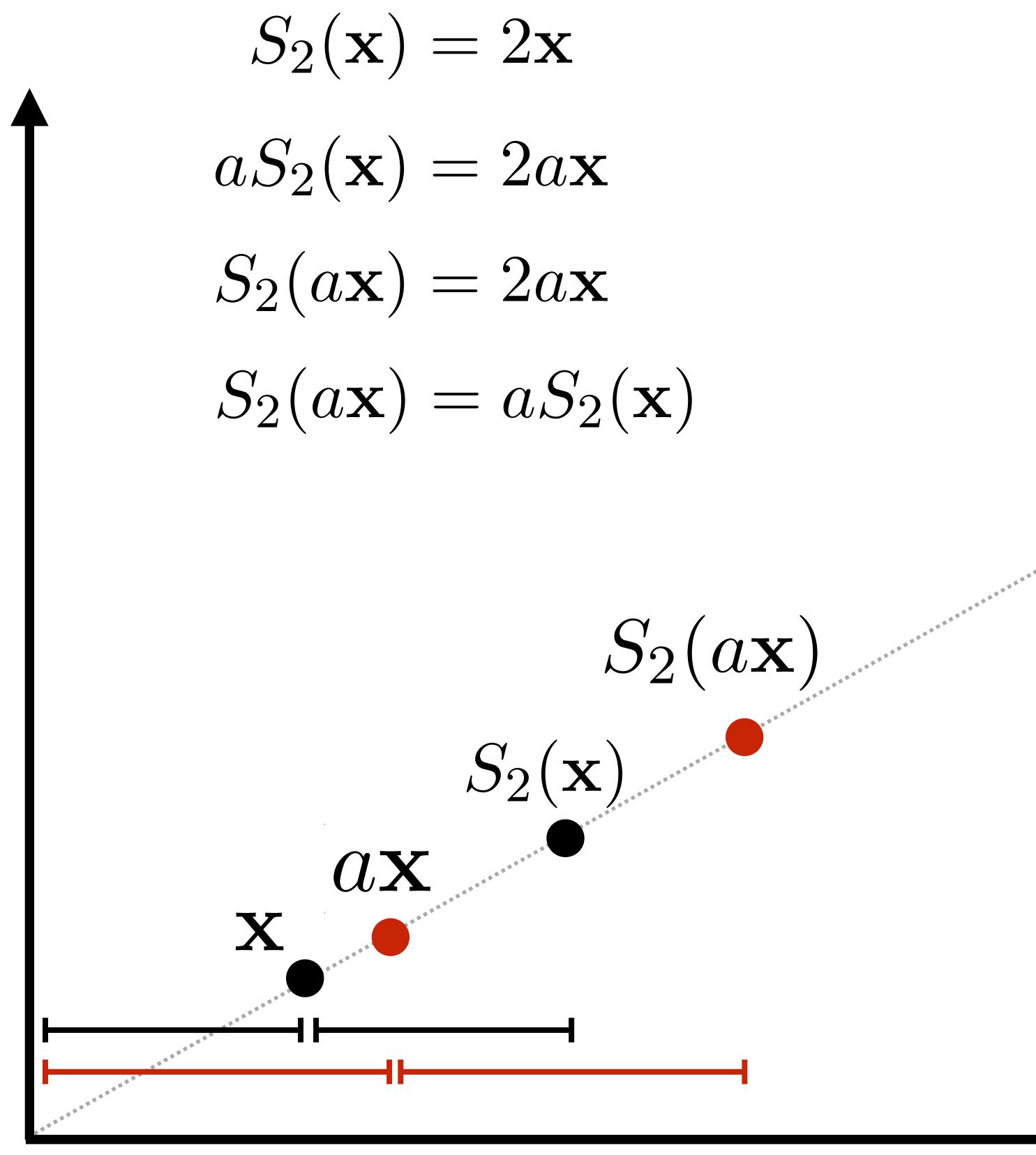


Uniform scale:
 $S_a(\mathbf{x}) = a\mathbf{x}$



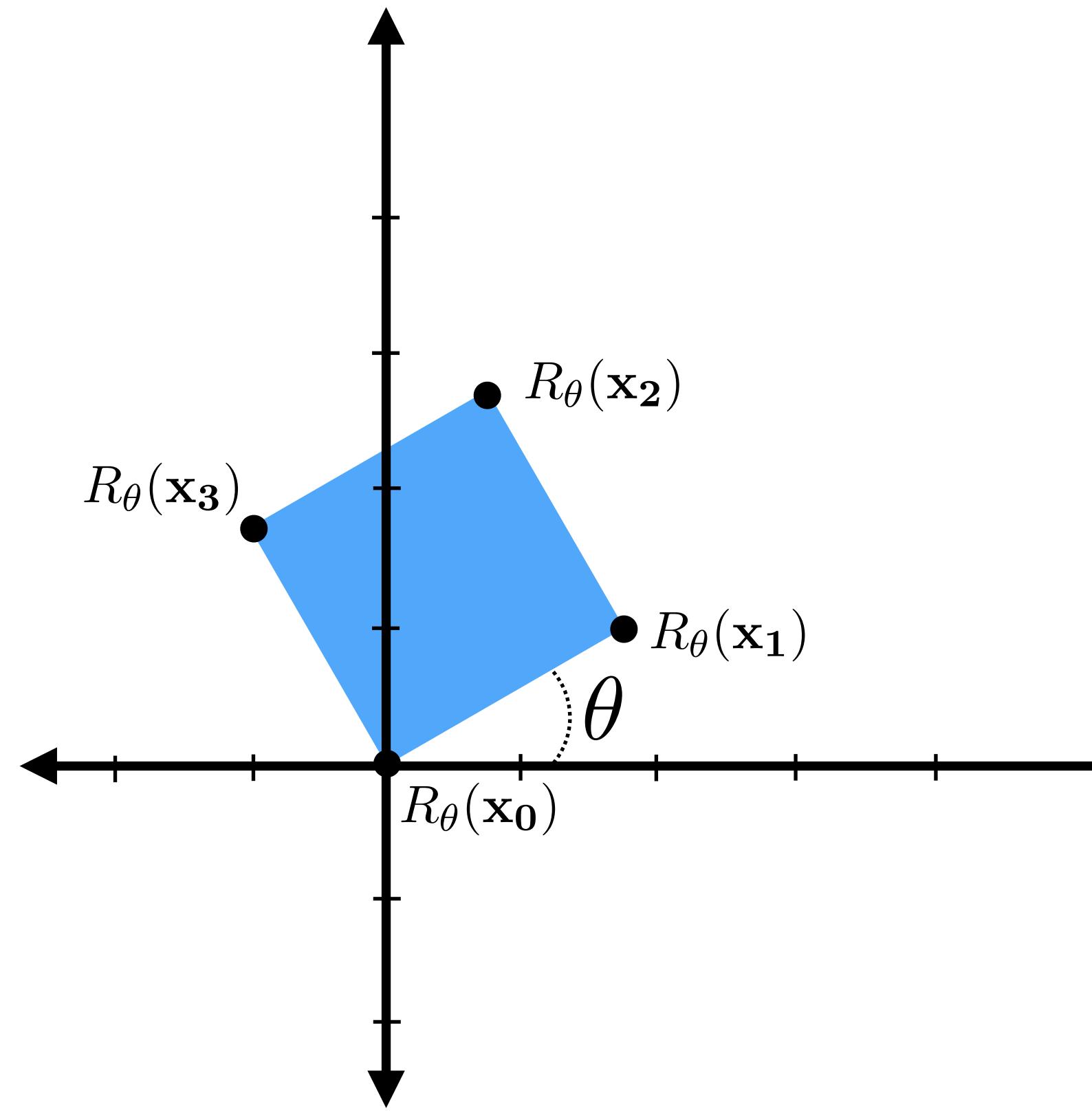
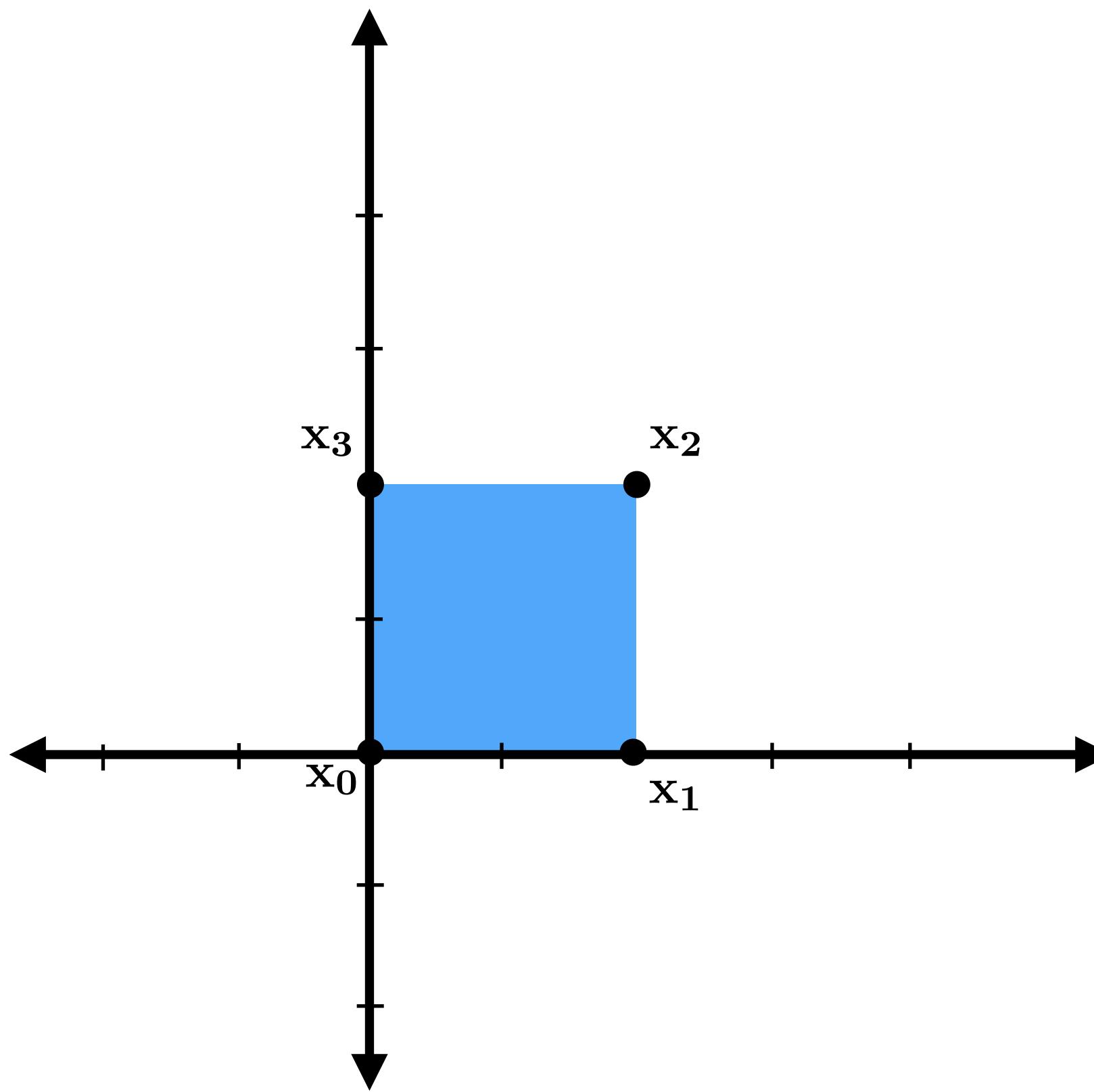
Non-uniform scale??

Is scale a linear transform?



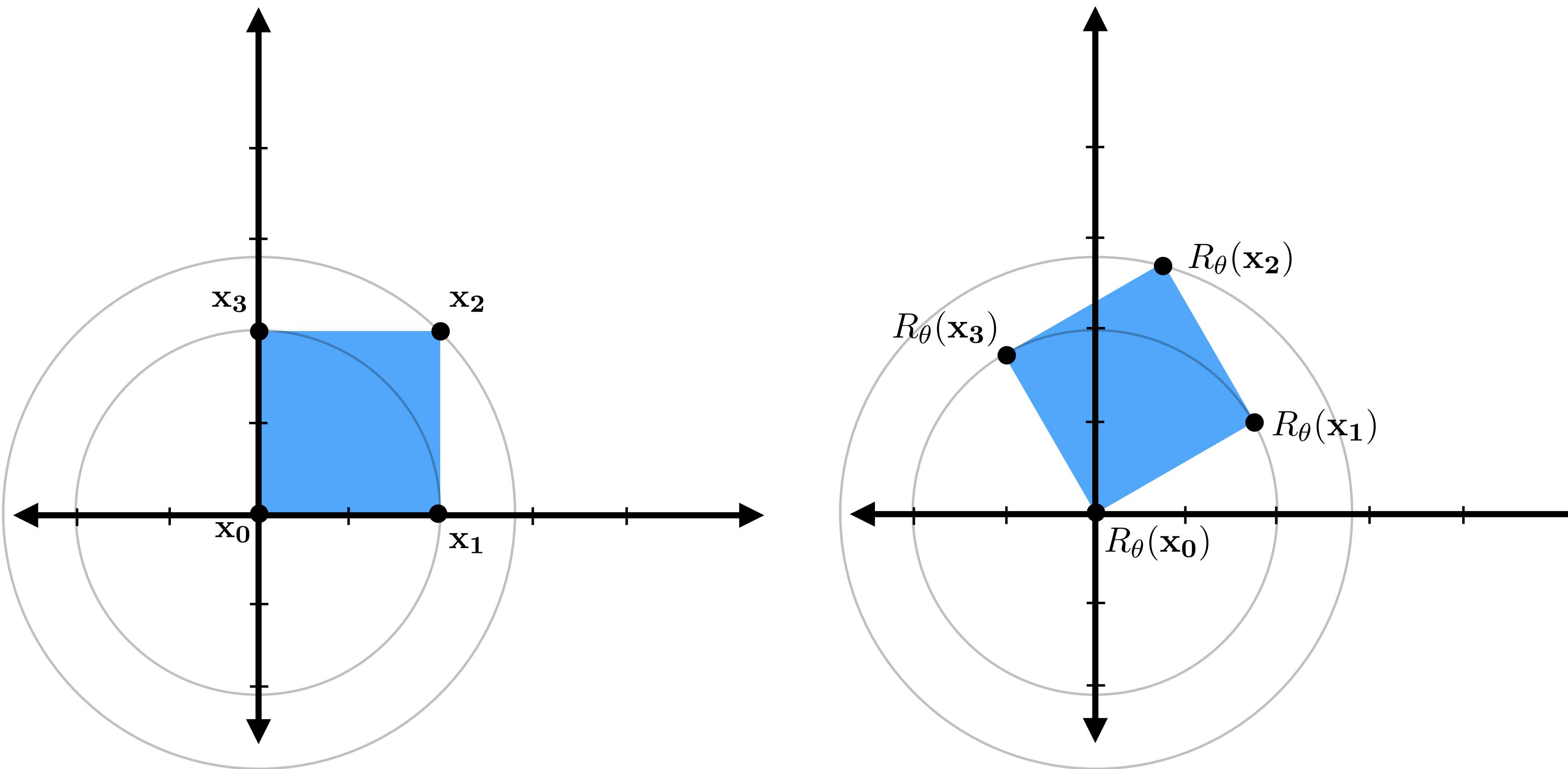
Yes!

Rotation transform



R_θ = **rotate counter-clockwise by θ**

Rotation as circular motion

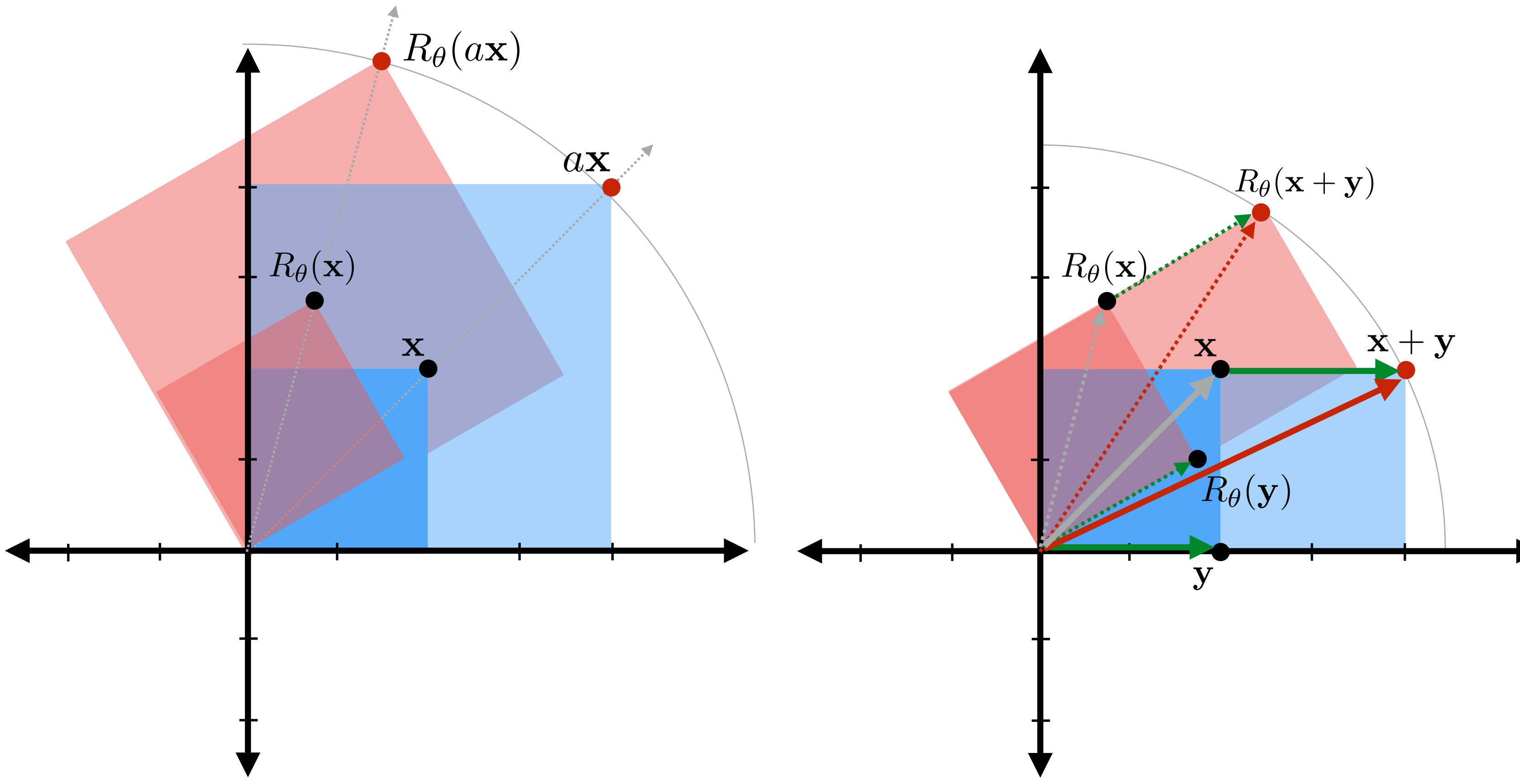


R_θ = **rotate counter-clockwise by θ**

As angle changes, points move along *circular* trajectories.

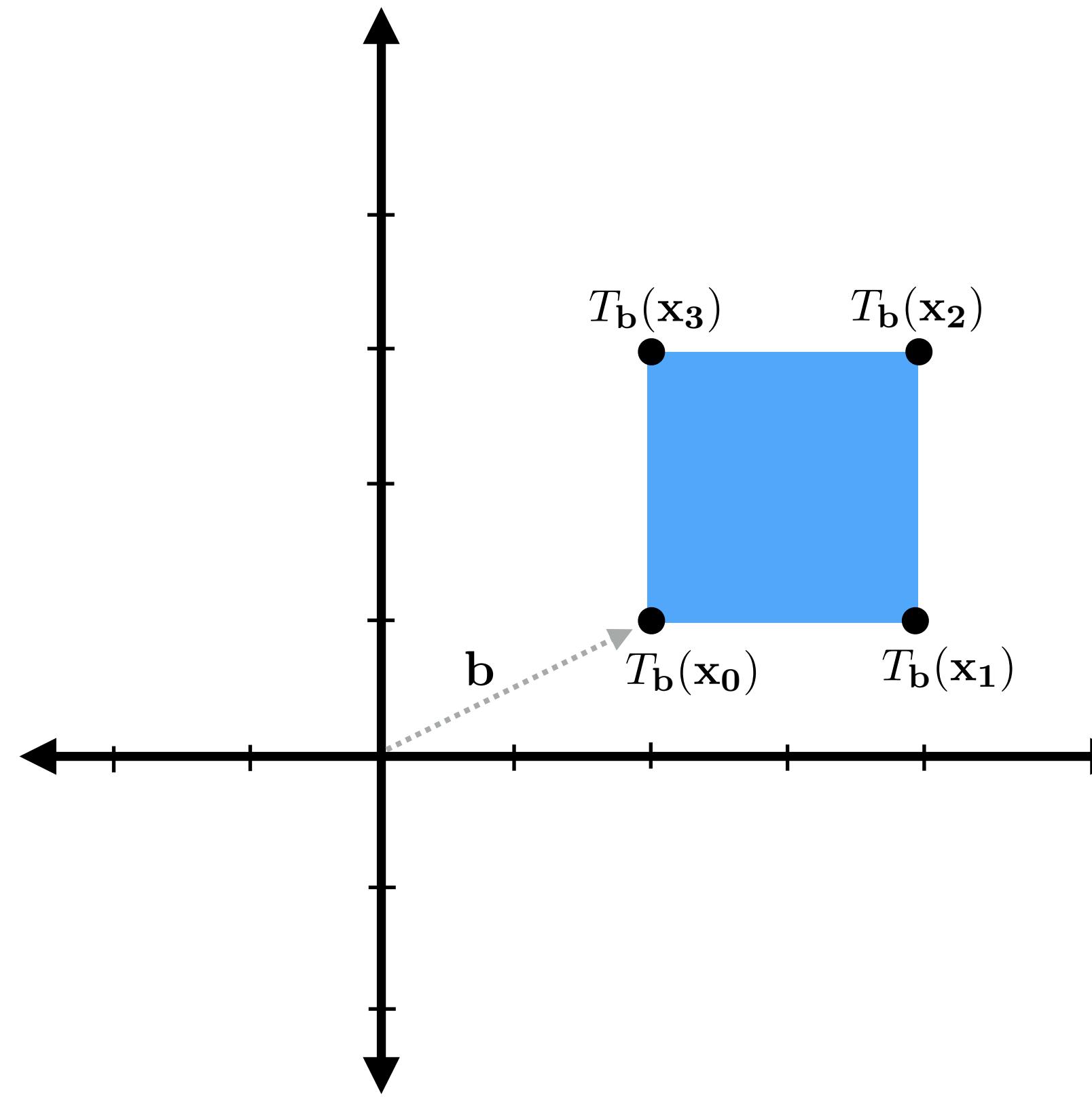
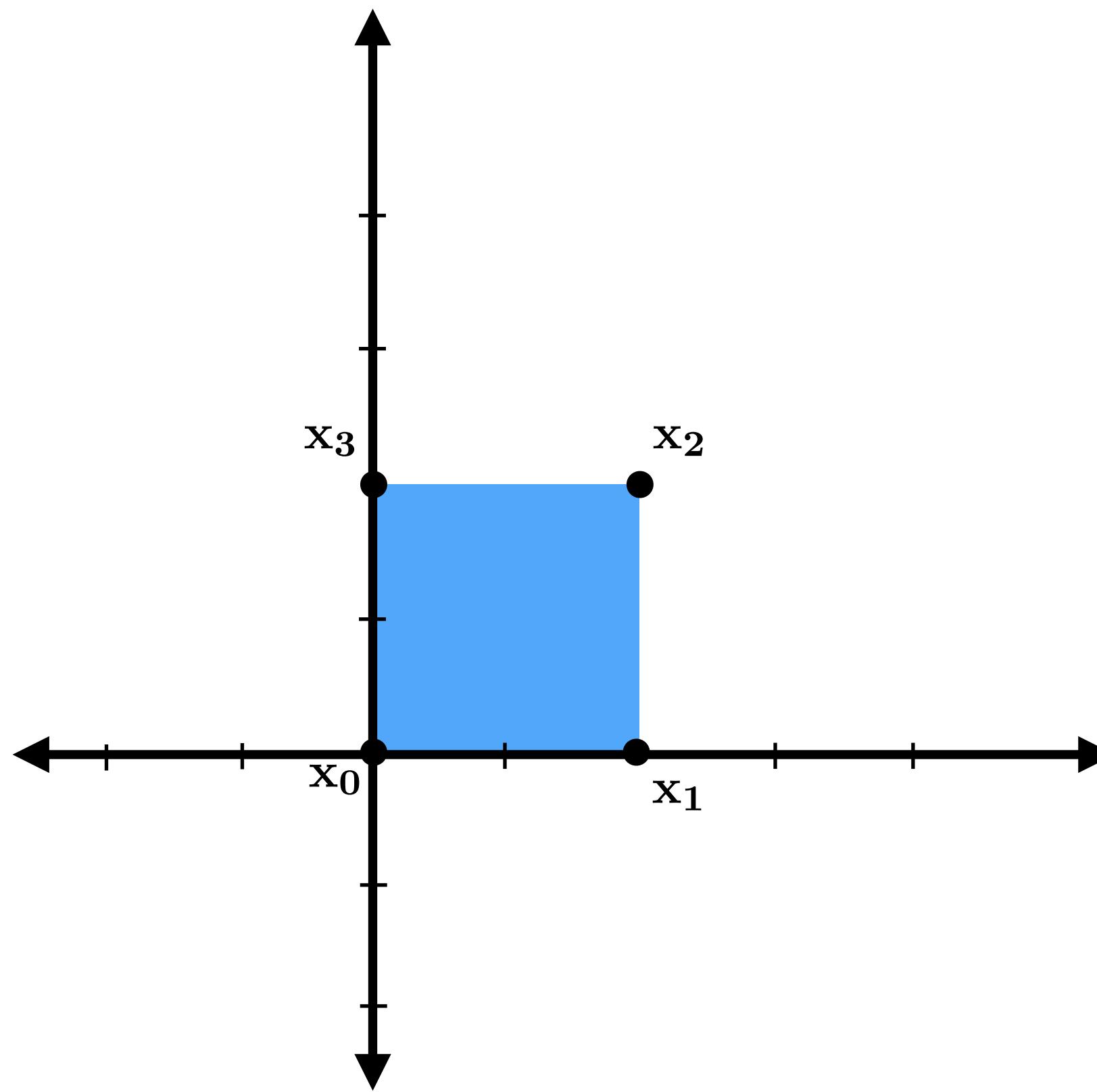
Hence, rotations preserve length of vectors: $|R_\theta(\mathbf{x})| = |\mathbf{x}|$

Is rotation linear?



Yes!

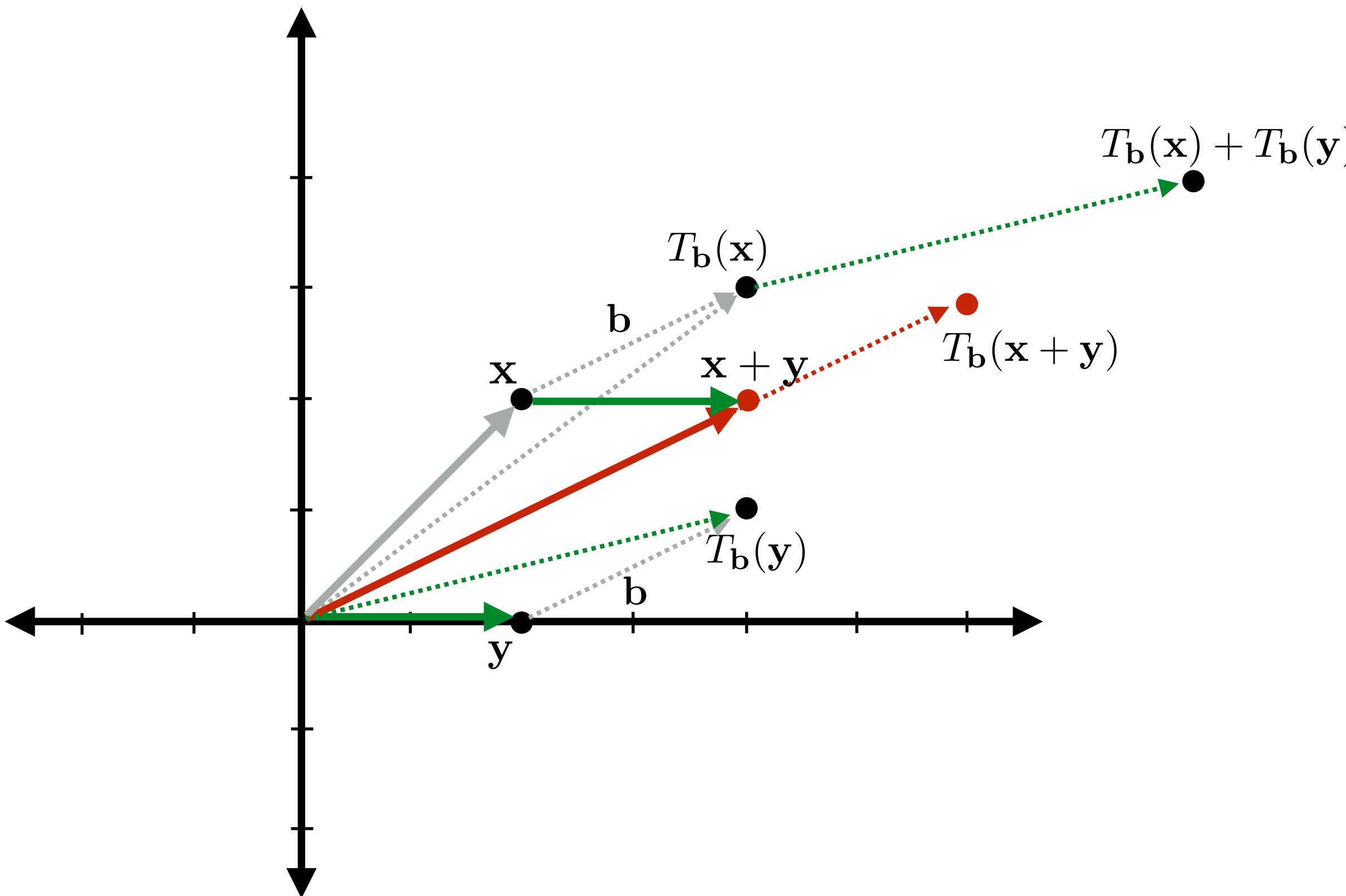
Translation transform



T_b — “translate by b ”

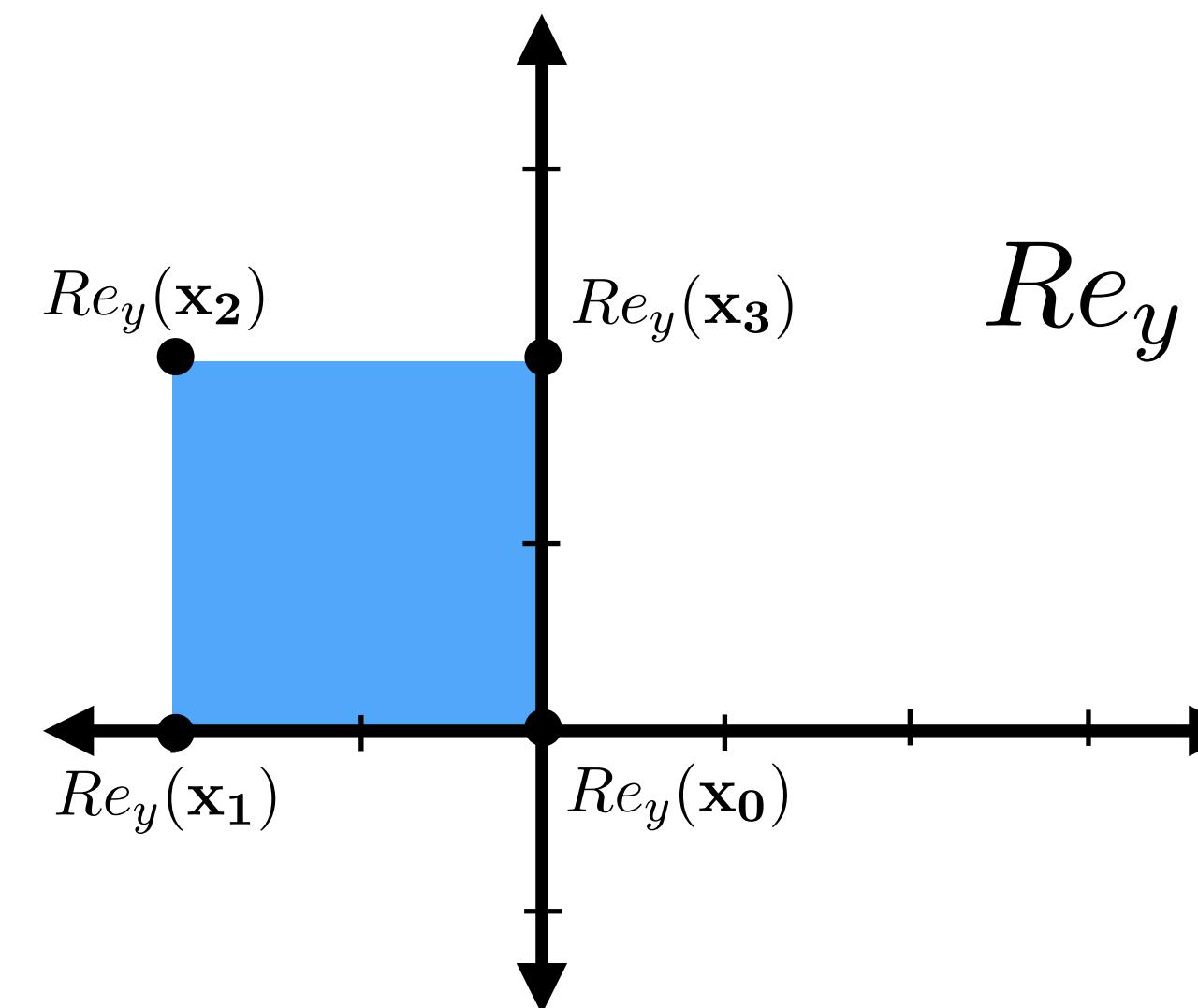
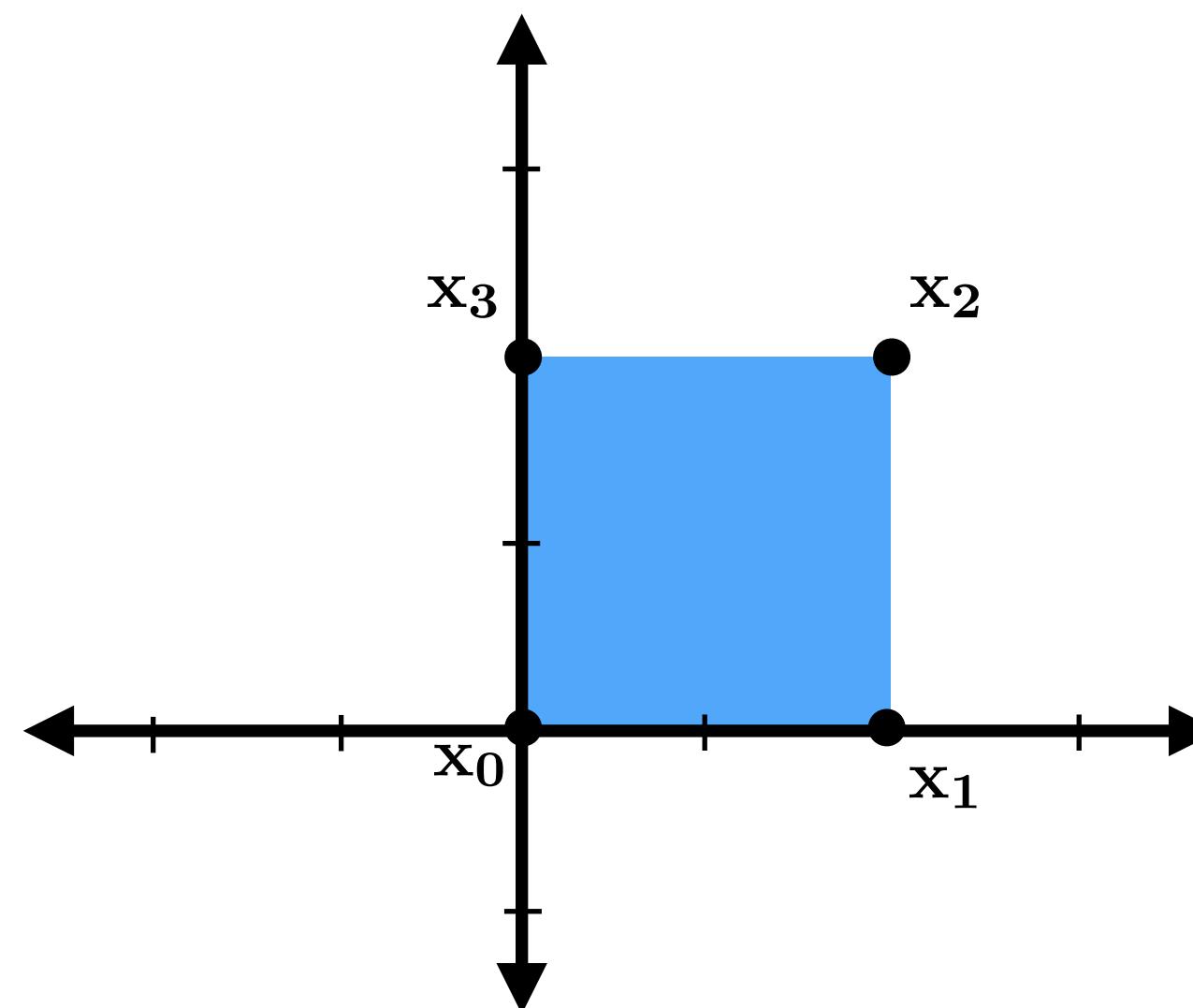
$$T_b(\mathbf{x}) = \mathbf{x} + \mathbf{b}$$

Is translation linear?

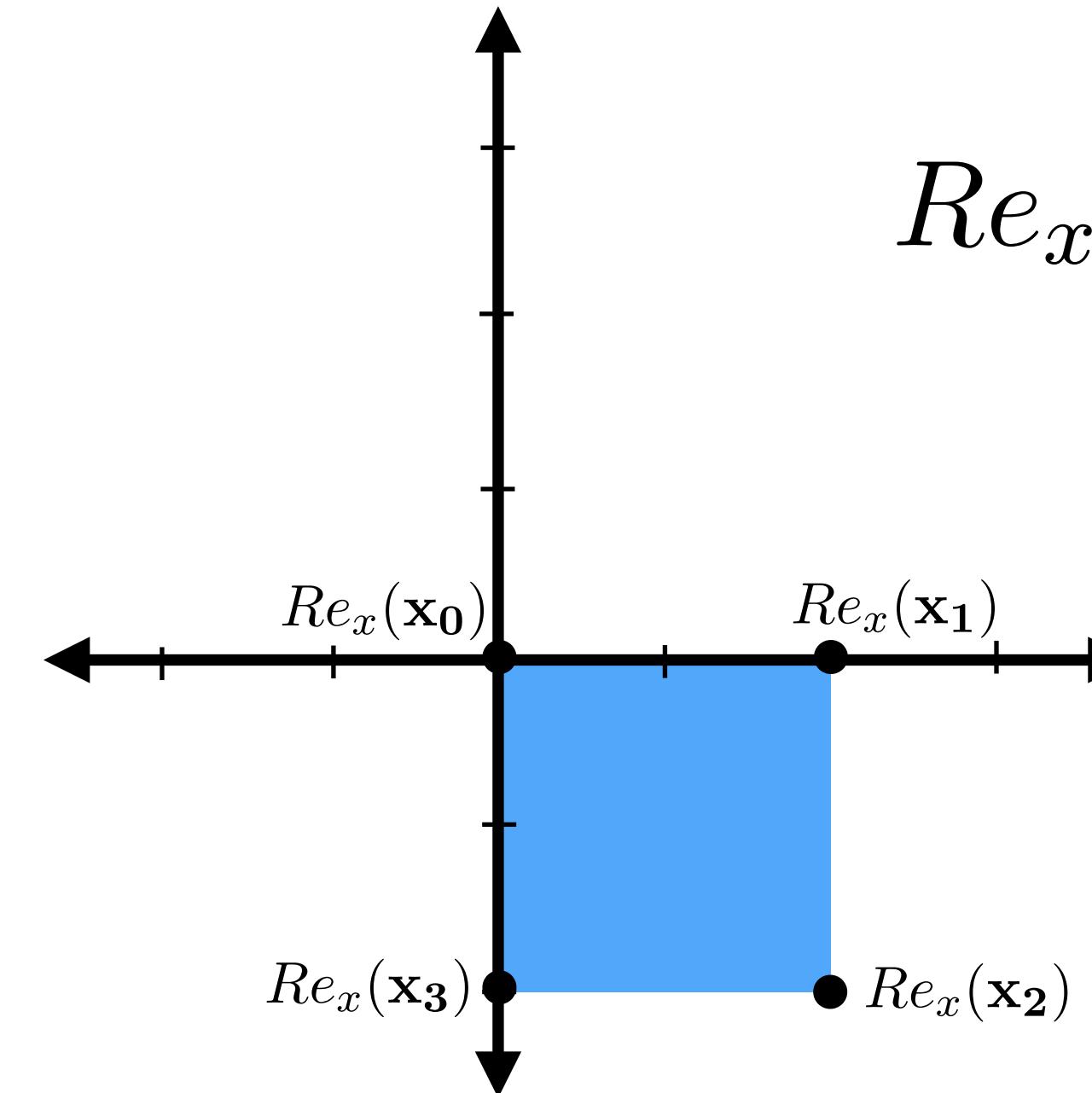


No. Translation is affine.

Reflection transform

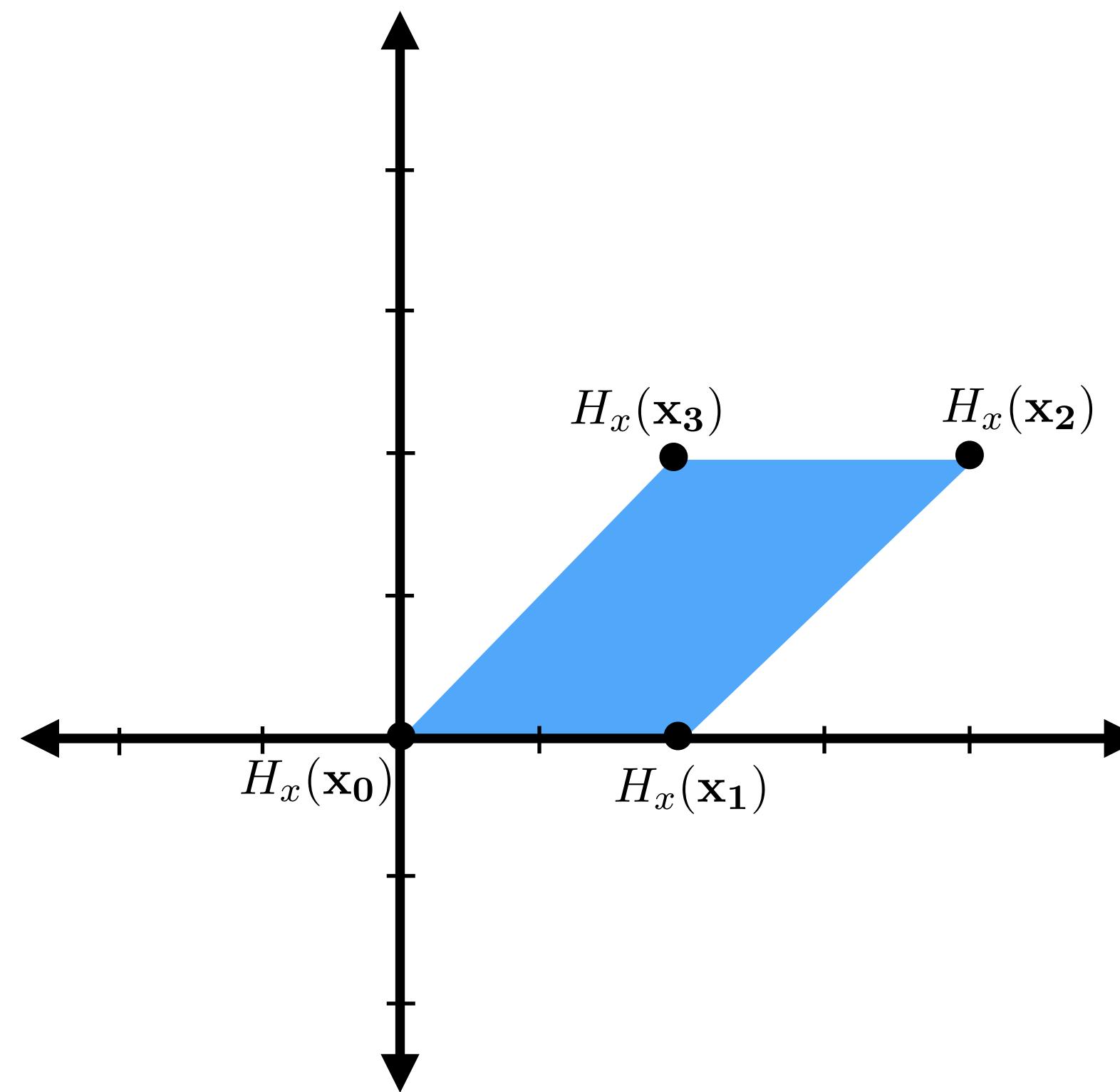
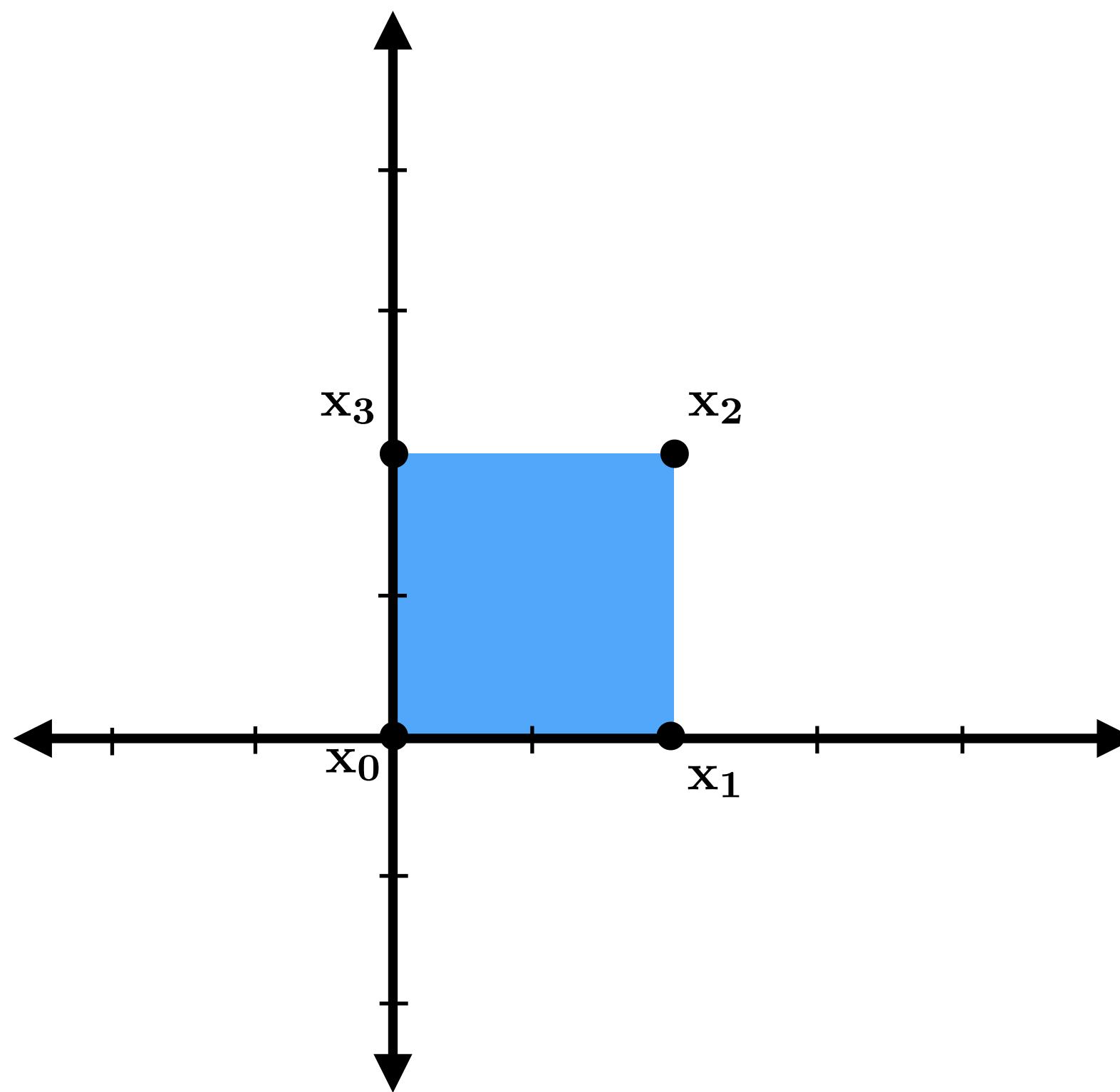


Re_y = reflection about y

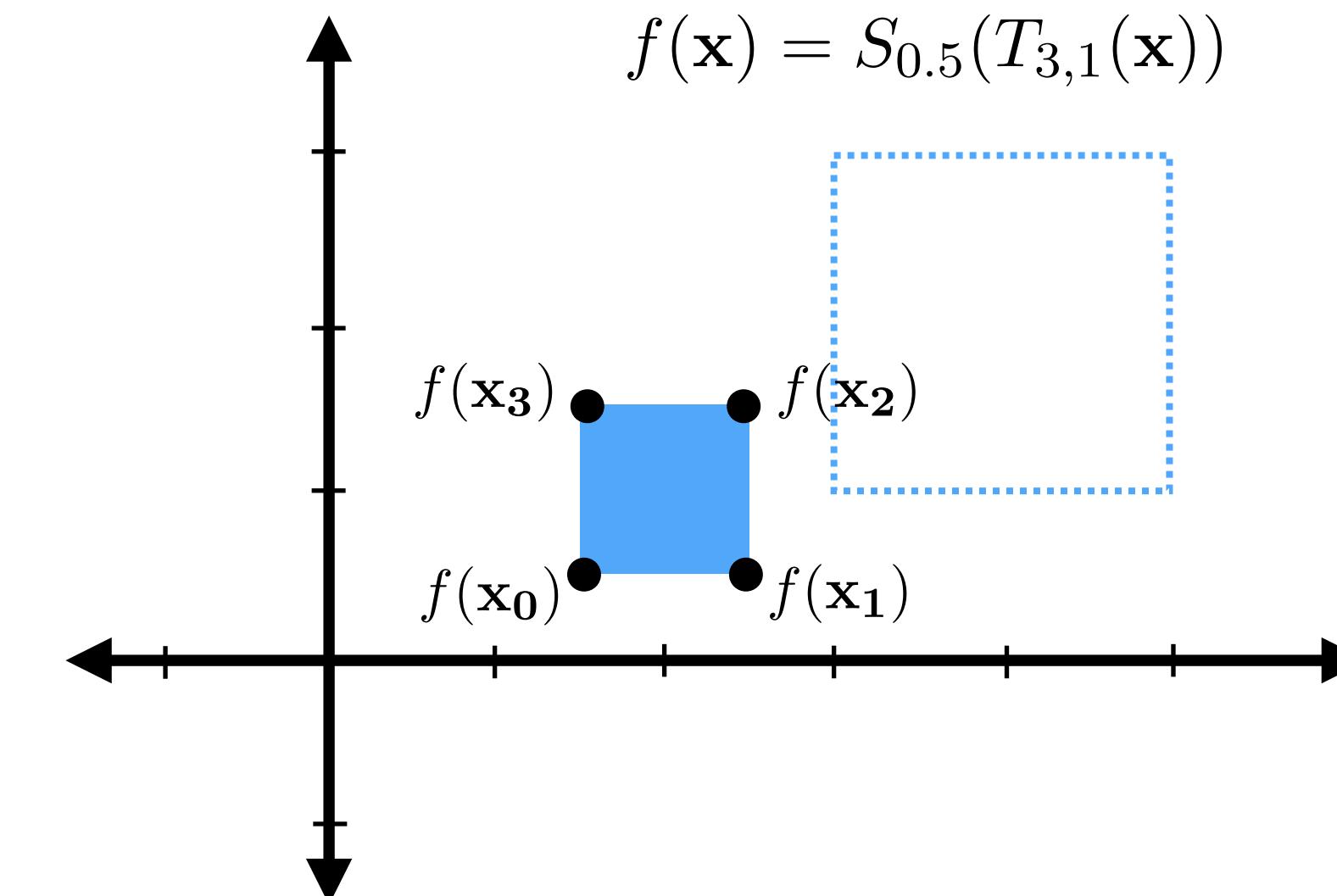
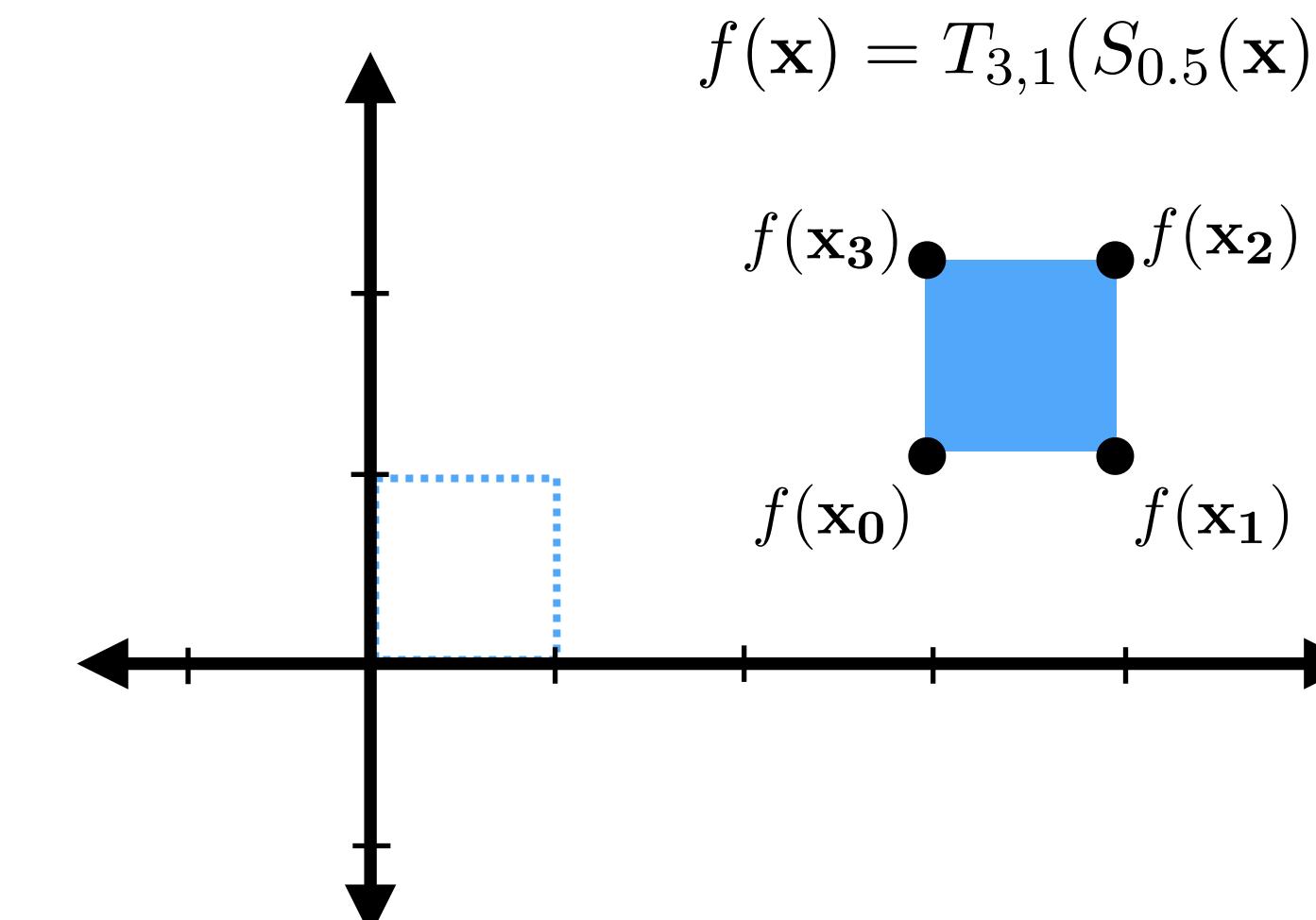
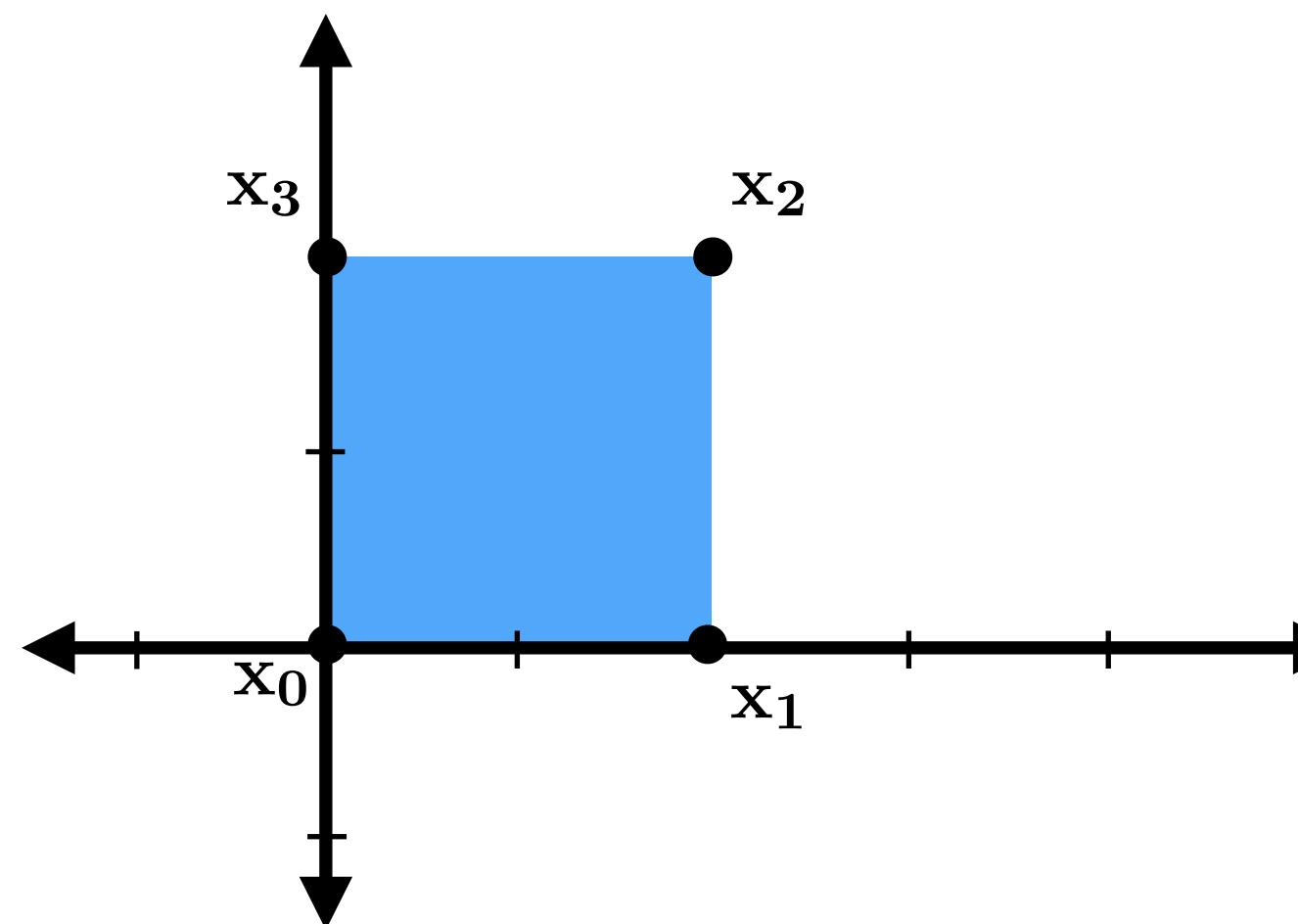


Re_x = reflection about x

Shear transform (shear in x direction)



Compose basic transformations to construct more complicated ones

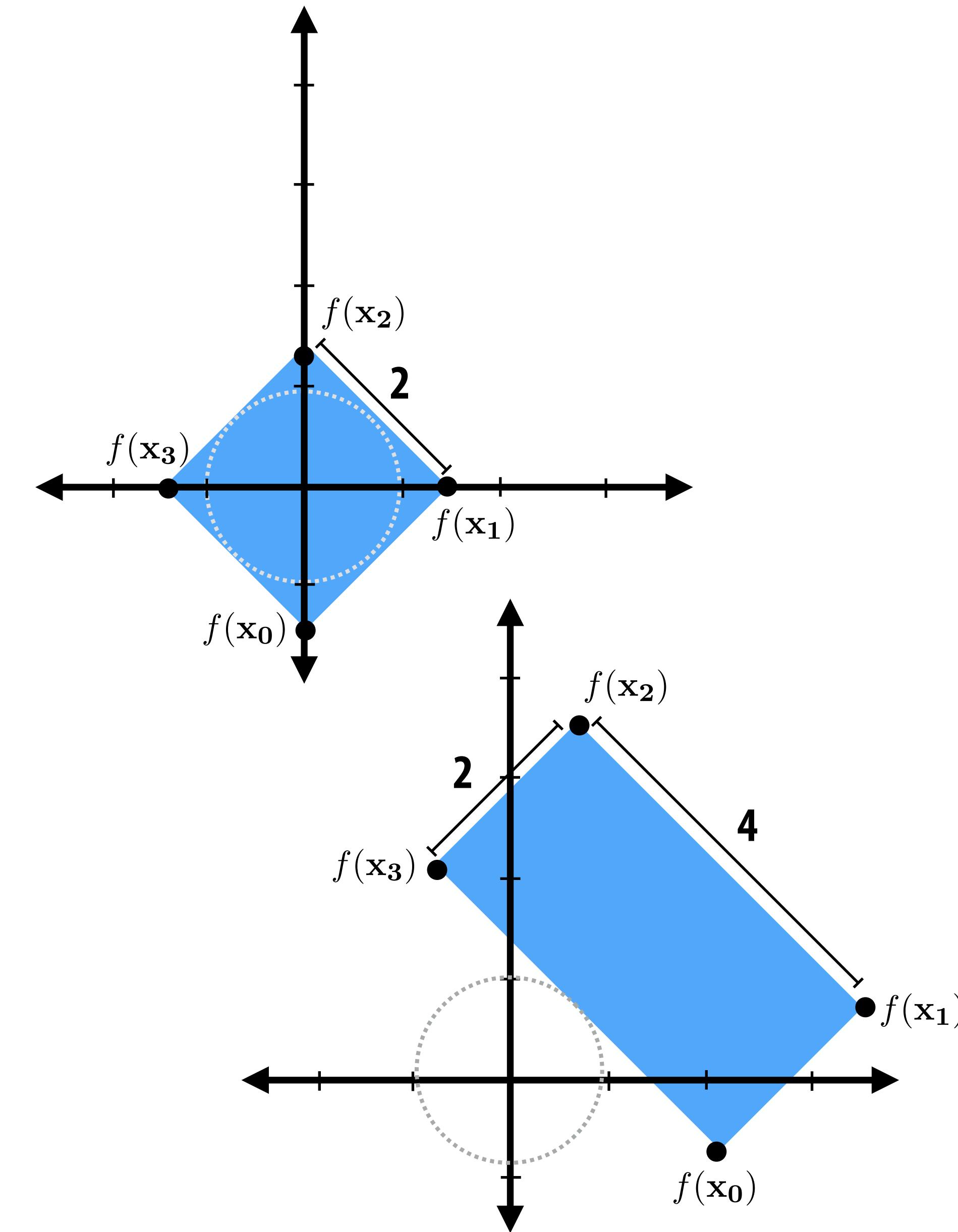
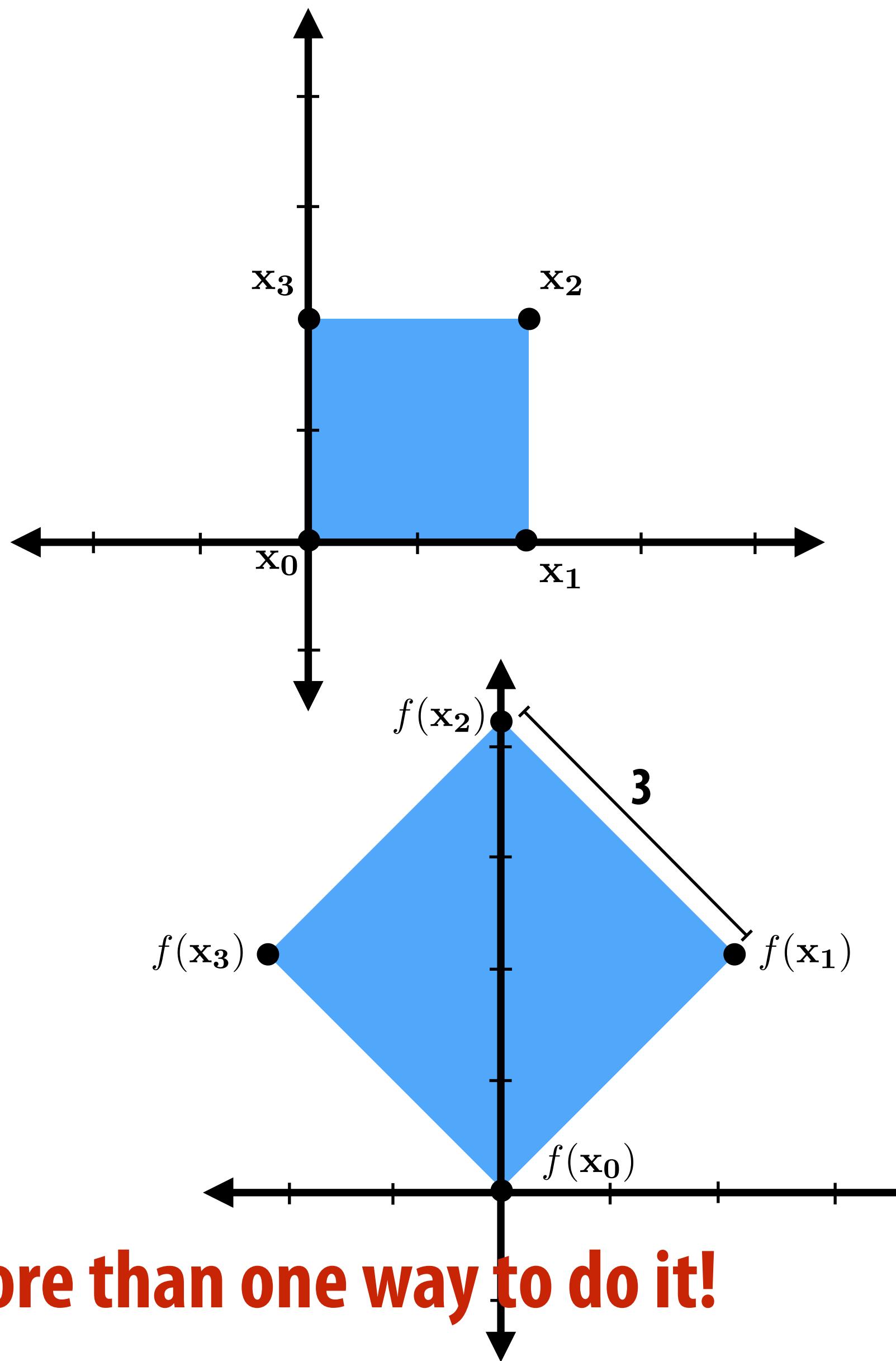


Note: order of composition matters

Top-right: scale, then translate

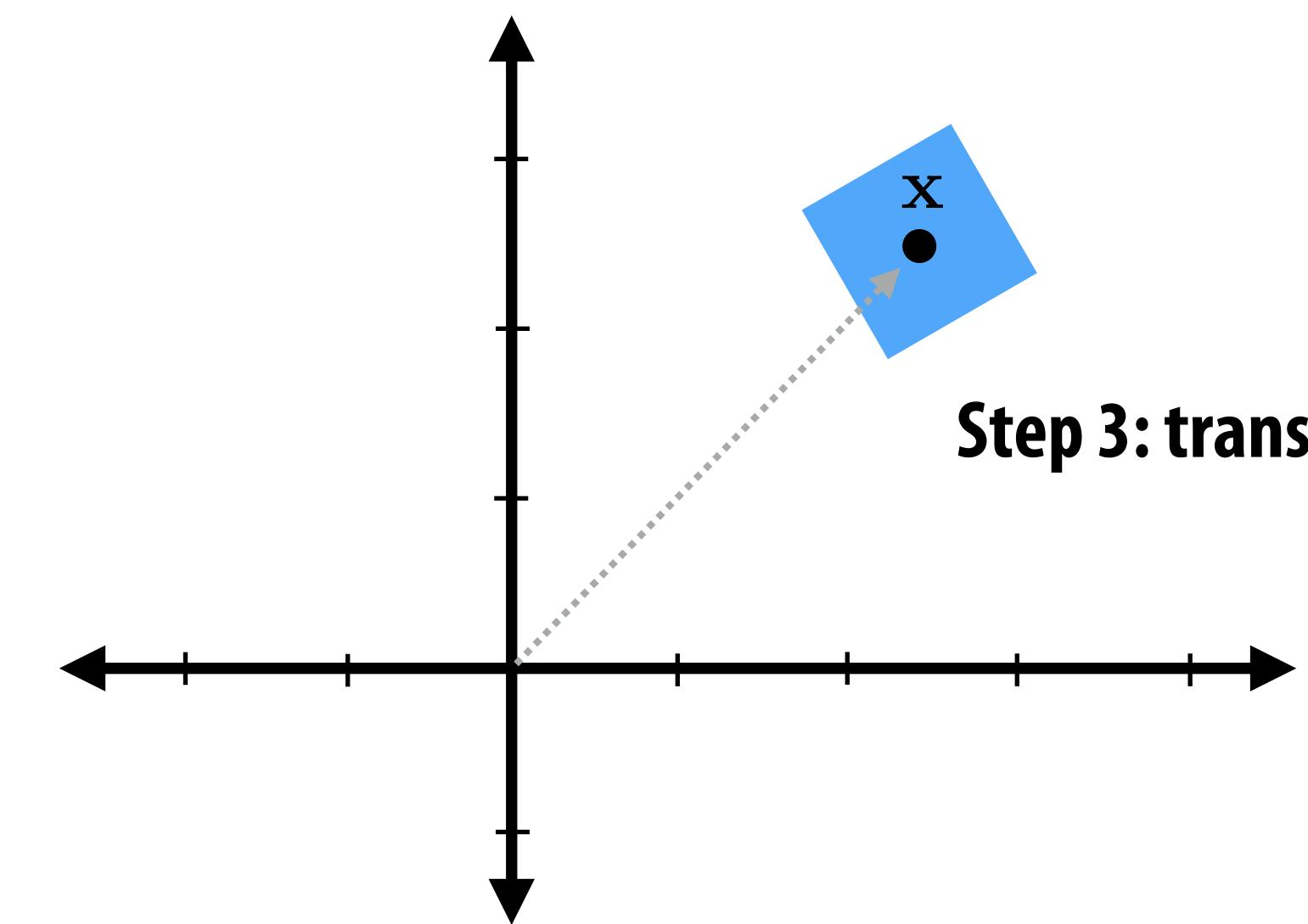
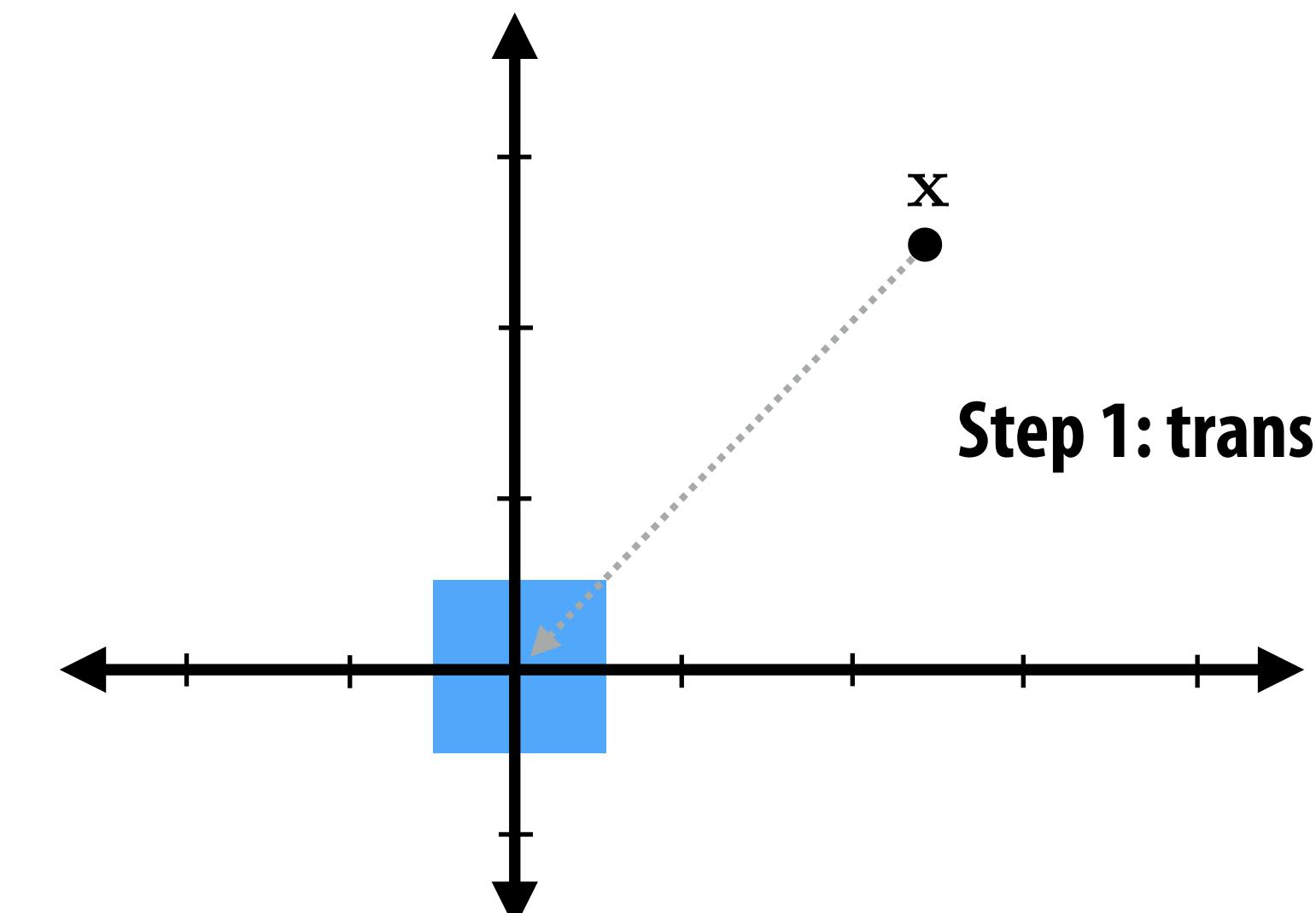
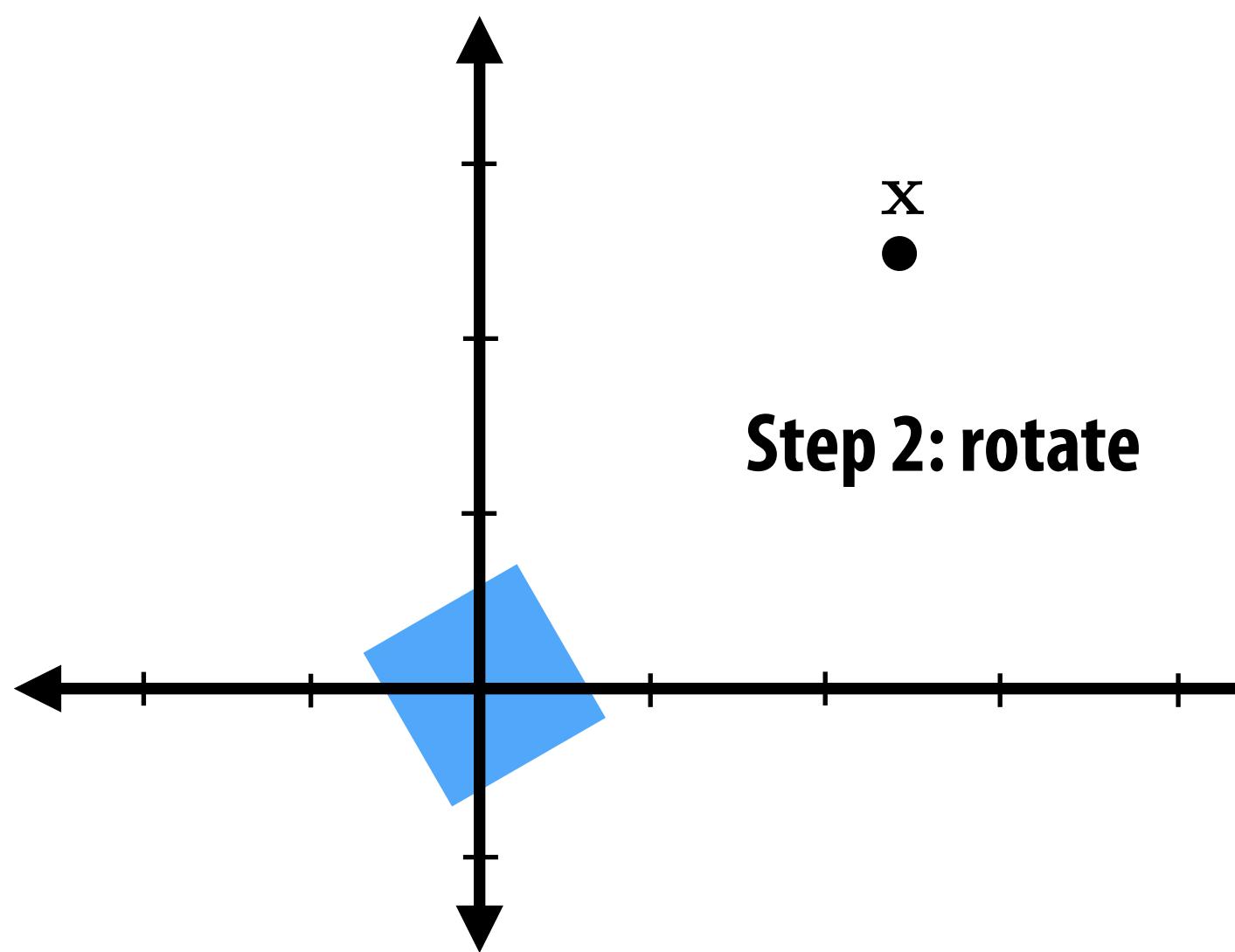
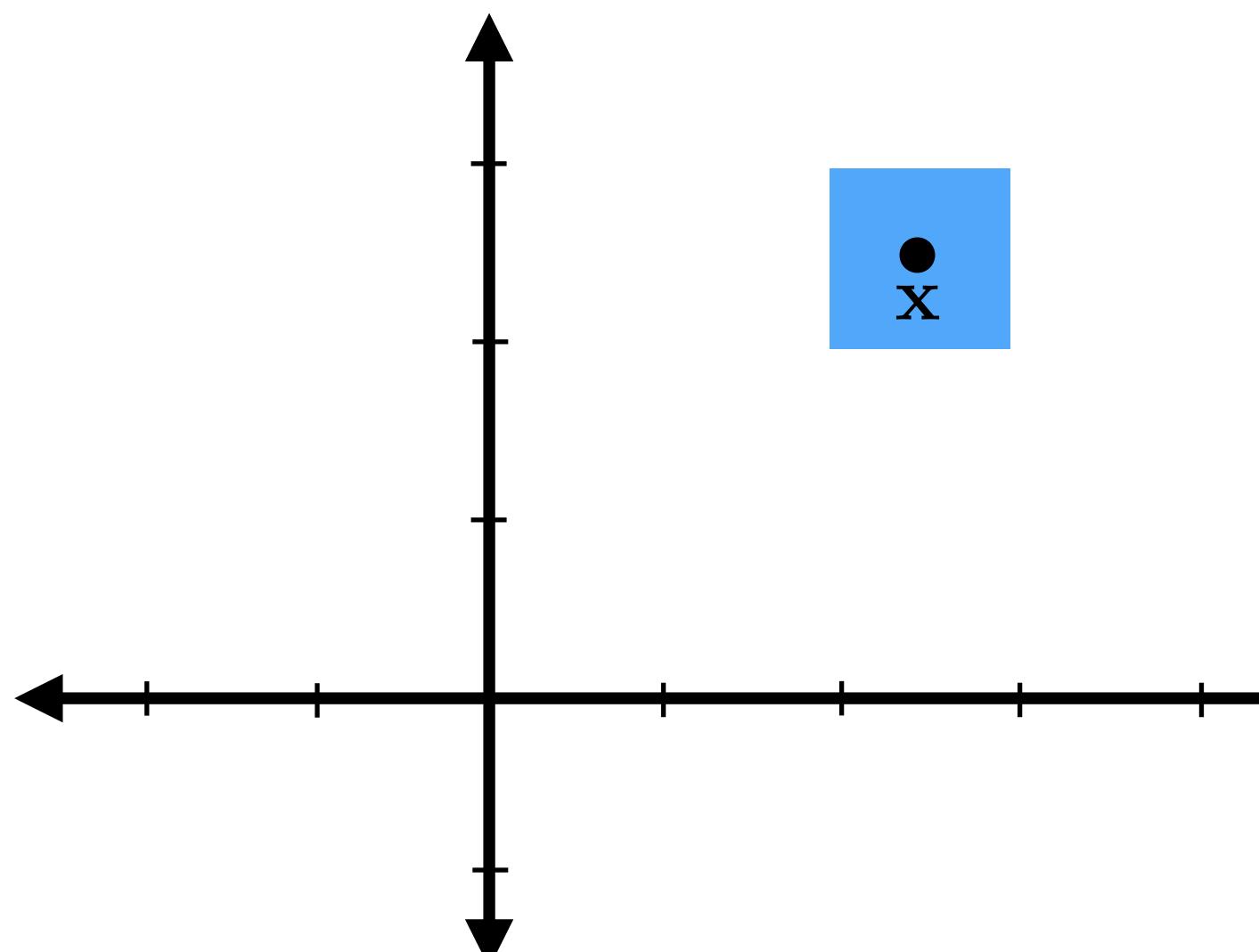
Bottom-right: translate, then scale

How would you perform these transformations?



Usually more than one way to do it!

Common task: rotate about a point x



Summary of basic transformations

Linear:

$$f(\mathbf{x} + \mathbf{y}) = f(\mathbf{x}) + f(\mathbf{y})$$

$$f(a\mathbf{x}) = af(\mathbf{x})$$

Scale

Rotation

Reflection

Shear

Not linear:

Translation

Affine:

Composition of linear transform + translation

(all examples on previous two slides)

$$f(\mathbf{x}) = g(\mathbf{x}) + \mathbf{b}$$

Not affine: perspective projection (will discuss later)

Euclidean: (Isometries)

Preserve distance between points (preserves length)

$$|f(\mathbf{x}) - f(\mathbf{y})| = |\mathbf{x} - \mathbf{y}|$$

Translation

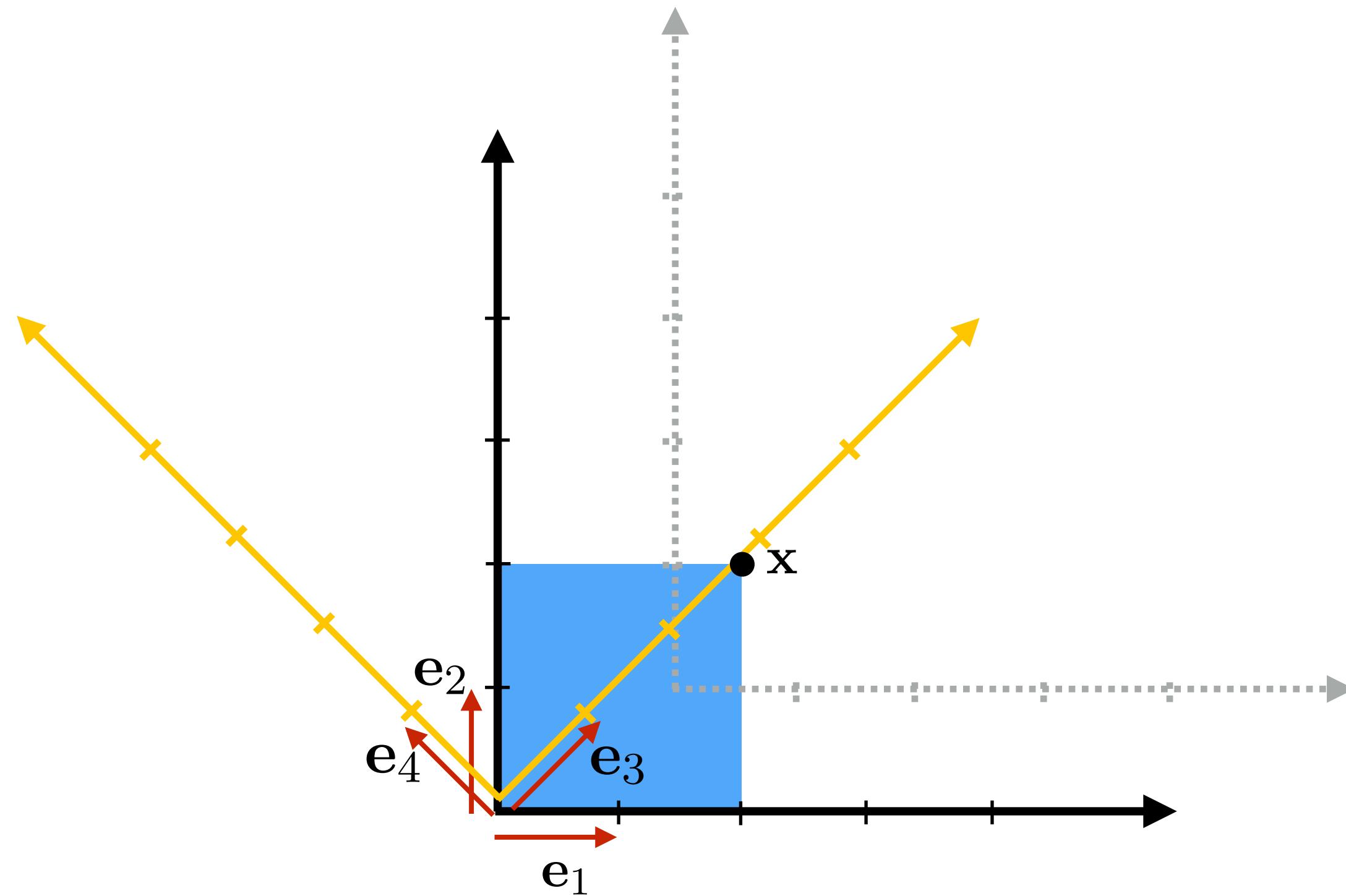
Rotation

Reflection

“Rigid body” transformations are distance-preserving motions that also preserve *orientation* (i.e., does not include reflection)

Representing transformations in coordinates

Review: representing points in a coordinate space



**It's the same point: x
But x is represented via different coordinates
in difference coordinate spaces!**

Consider coordinate space defined by orthogonal vectors e_1 and e_2

$$x = 2e_1 + 2e_2$$

$$x = [2 \ 2]$$

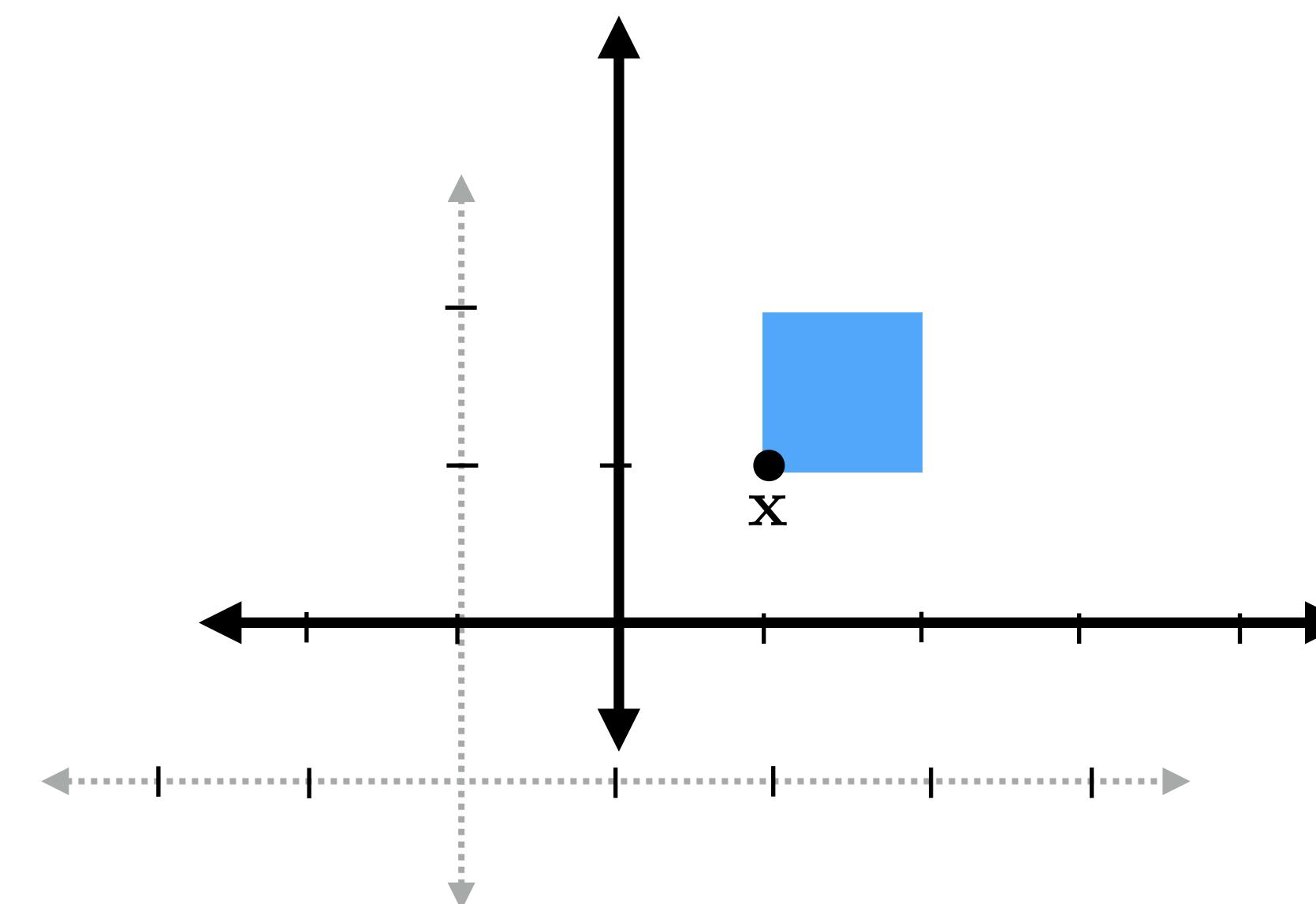
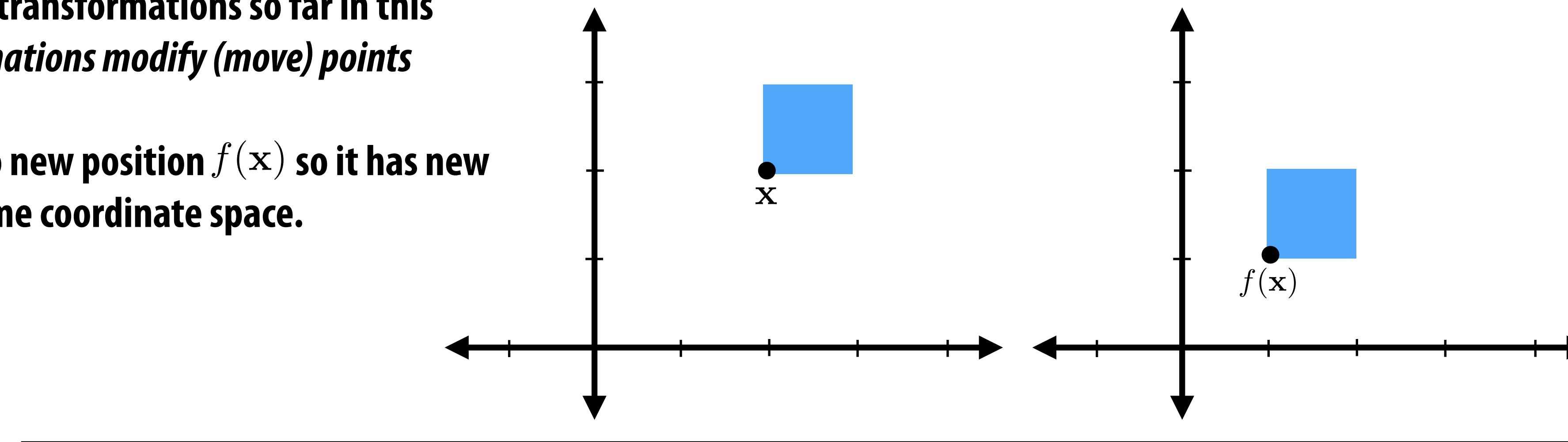
$x = [0.5 \ 1]$ in coordinate space defined by e_1 and e_2 , with origin at $(1.5, 1)$

$x = [\sqrt{8} \ 0]$ in coordinate space defined by e_3 and e_4 , with origin at $(0, 0)$

Another way to think about transformations: change of coordinates

Interpretation of transformations so far in this lecture: *transformations modify (move) points*

Point x moved to new position $f(x)$ so it has new coordinates in same coordinate space.



Alternative interpretation:

Transformations induce of change of coordinate frame:
Representation of x changes since point is now expressed in new coordinates

Review: 2D matrix multiplication

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} =$$

$$x \begin{bmatrix} a \\ c \end{bmatrix} + y \begin{bmatrix} b \\ d \end{bmatrix} =$$

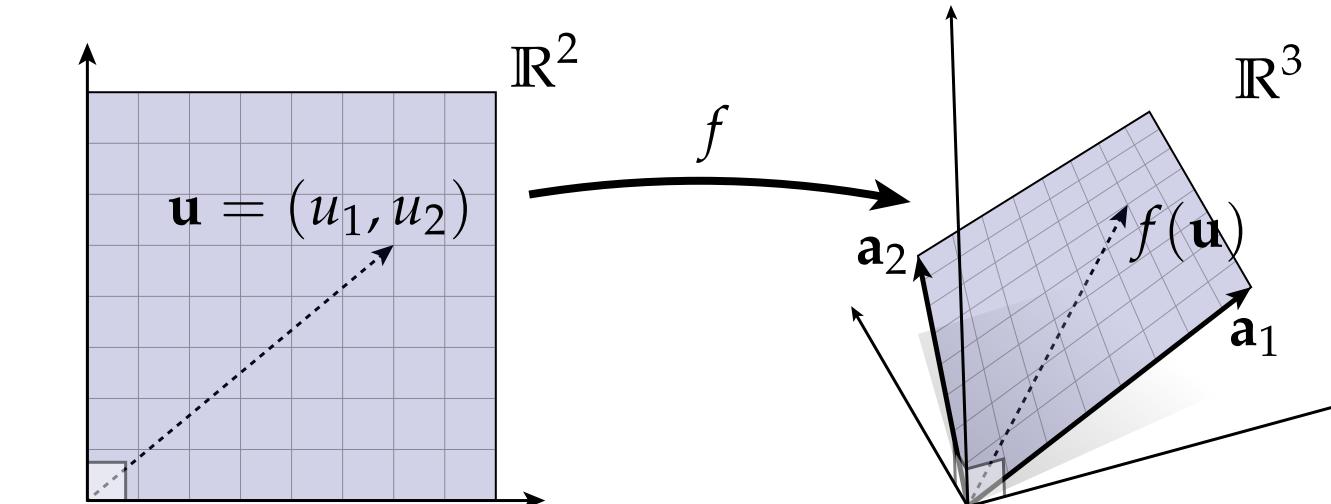
$$\begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$$

- Matrix multiplication is linear combination of columns
- Encodes a linear map!

Linear maps via matrices

- Example: suppose I have a linear map

$$f(\mathbf{u}) = u_1 \mathbf{a}_1 + u_2 \mathbf{a}_2$$



- Encoding the map as a matrix: “a” vectors become matrix columns:

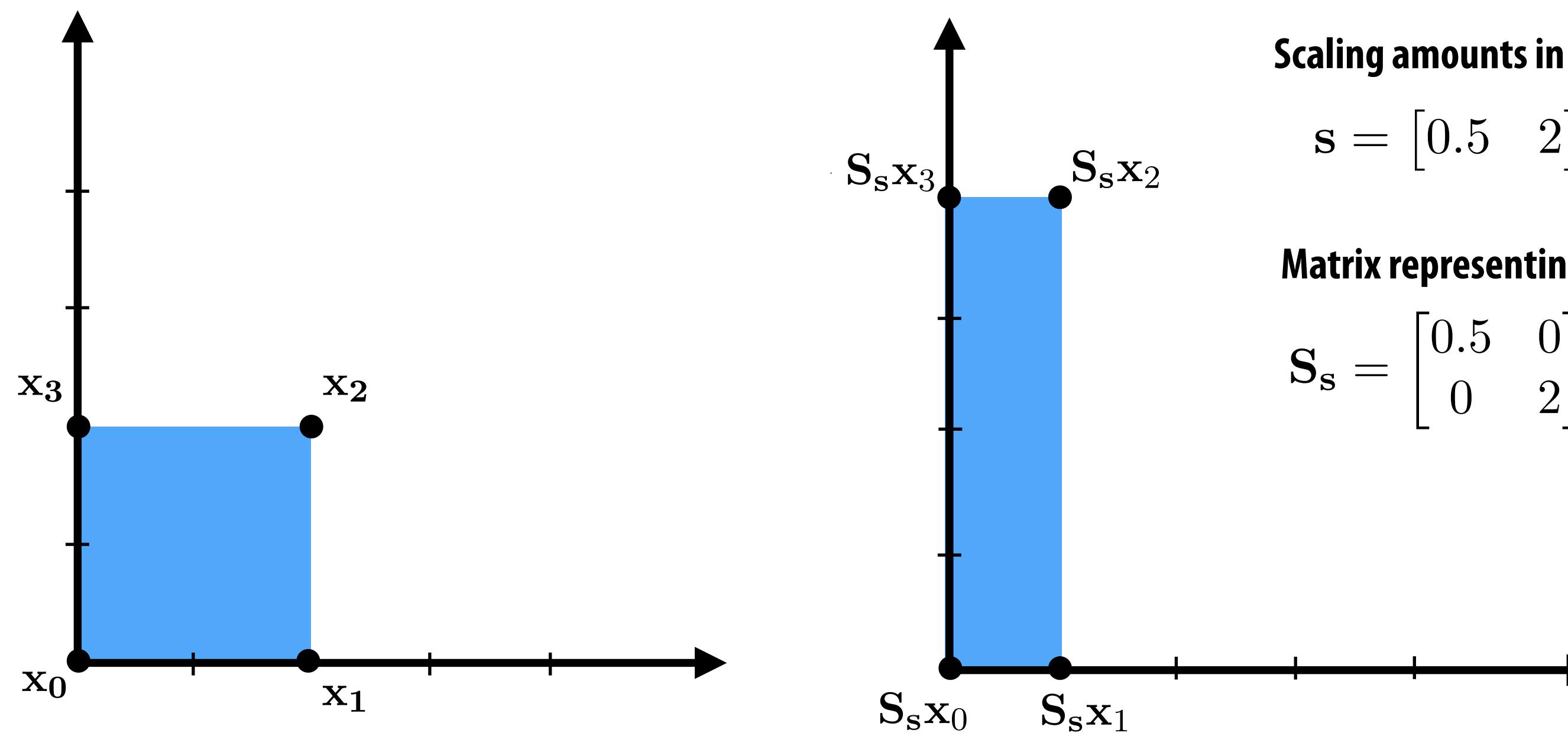
$$A := \begin{bmatrix} a_{1,x} & a_{2,x} \\ a_{1,y} & a_{2,y} \\ a_{1,z} & a_{2,z} \end{bmatrix}$$

- Matrix-vector multiply computes same output as original map:

$$\begin{bmatrix} a_{1,x} & a_{2,x} \\ a_{1,y} & a_{2,y} \\ a_{1,z} & a_{2,z} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} a_{1,x}u_1 + a_{2,x}u_2 \\ a_{1,y}u_1 + a_{2,y}u_2 \\ a_{1,z}u_1 + a_{2,z}u_2 \end{bmatrix} = u_1 \mathbf{a}_1 + u_2 \mathbf{a}_2$$

Linear transformations in 2D can be represented as 2x2 matrices

Consider non-uniform scale: $S_s = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$



Scaling amounts in each direction:

$$s = [0.5 \ 2]^T$$

Matrix representing scale transform:

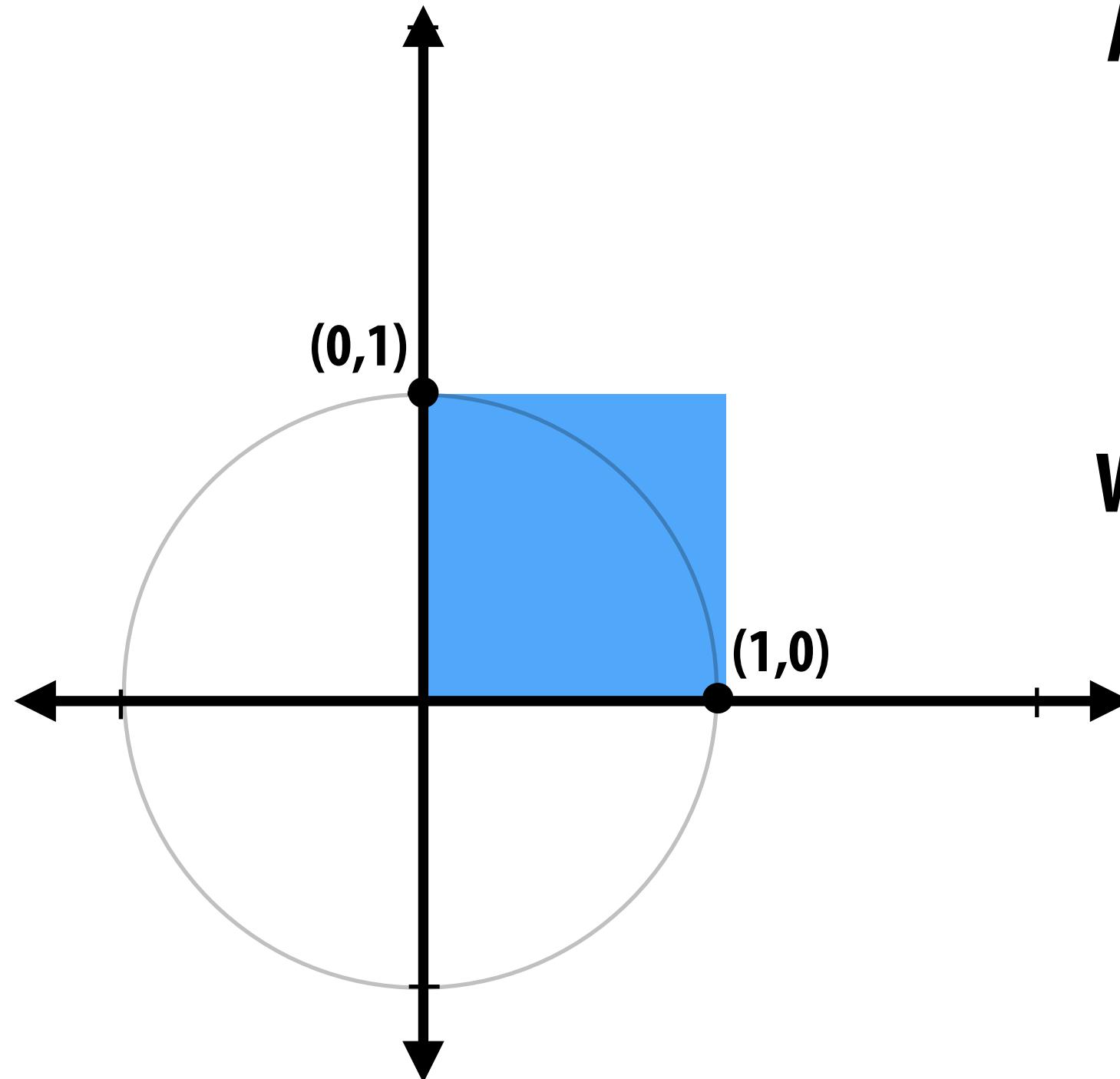
$$S_s = \begin{bmatrix} 0.5 & 0 \\ 0 & 2 \end{bmatrix}$$

Rotation matrix (2D)

Question: what happens to $(1, 0)$ and $(0, 1)$ after rotation by θ ?

Reminder: rotation moves points along circular trajectories.

(Recall that $\cos \theta$ and $\sin \theta$ are the coordinates of a point on the unit circle.)



Answer:

$$R_\theta(1, 0) = (\cos(\theta), \sin(\theta))$$

$$R_\theta(0, 1) = (\cos(\theta + \pi/2), \sin(\theta + \pi/2))$$

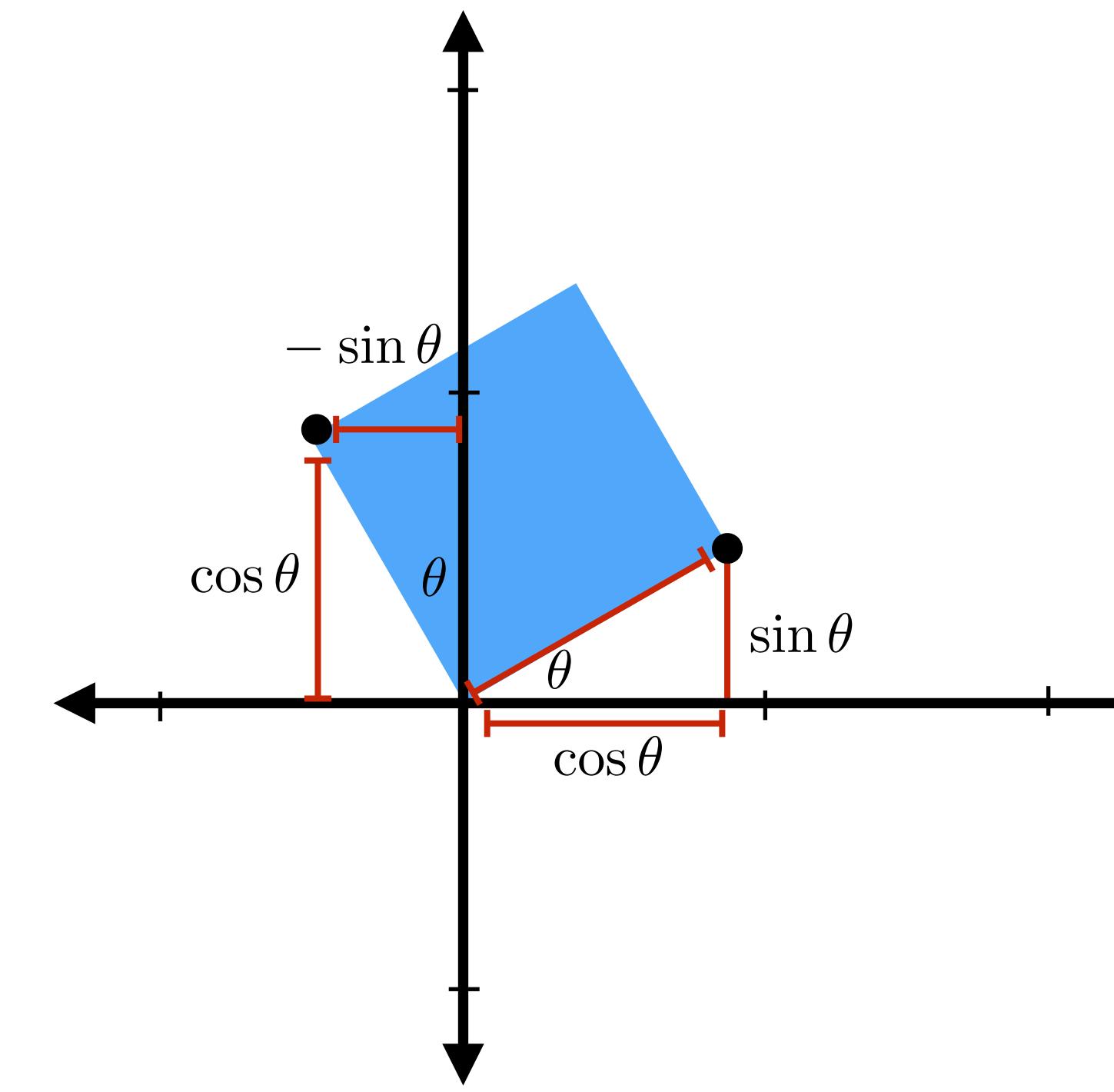
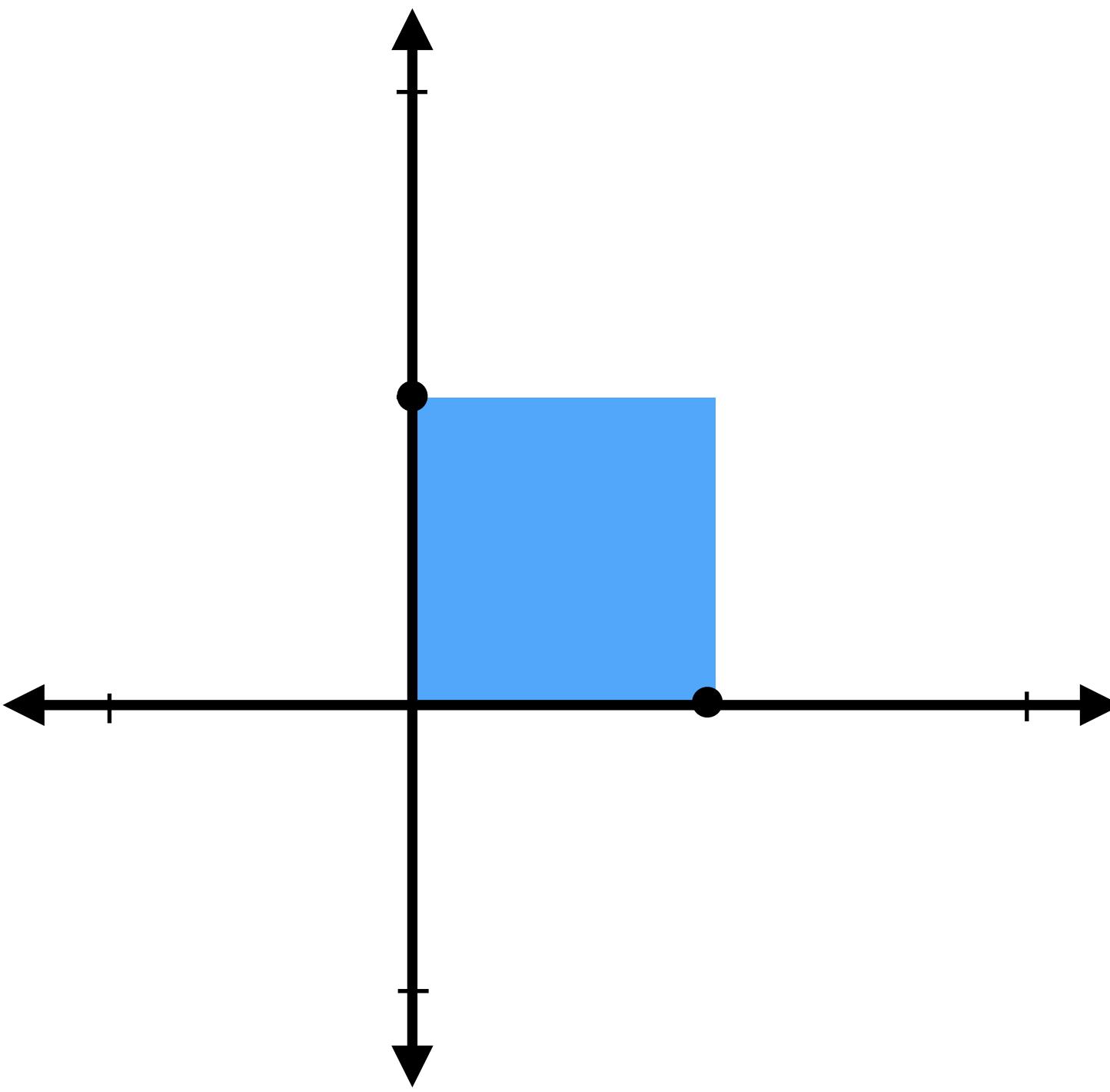
Which means the matrix must look like:

$$R_\theta = \begin{bmatrix} \cos(\theta) & \cos(\theta + \pi/2) \\ \sin(\theta) & \sin(\theta + \pi/2) \end{bmatrix}$$

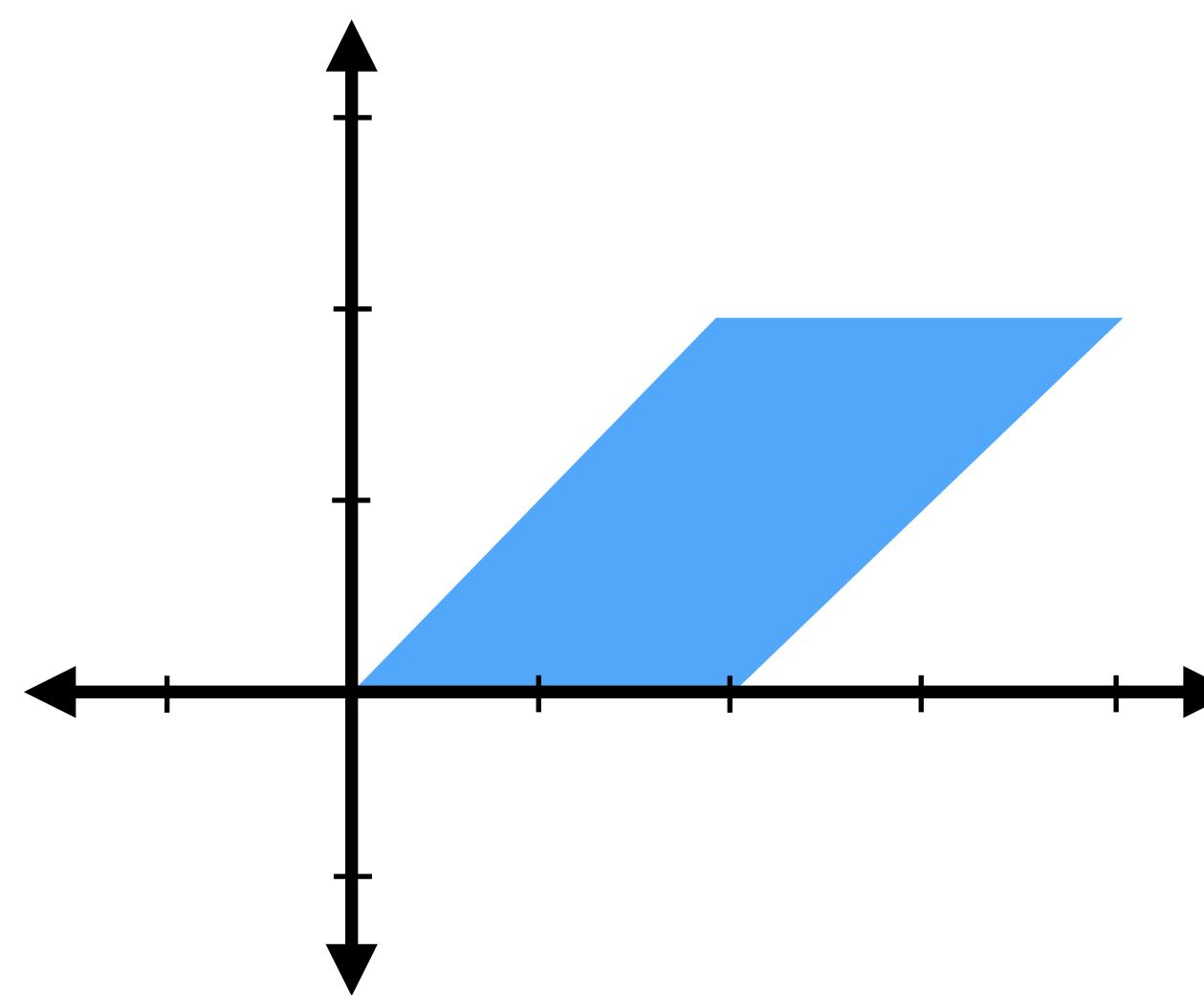
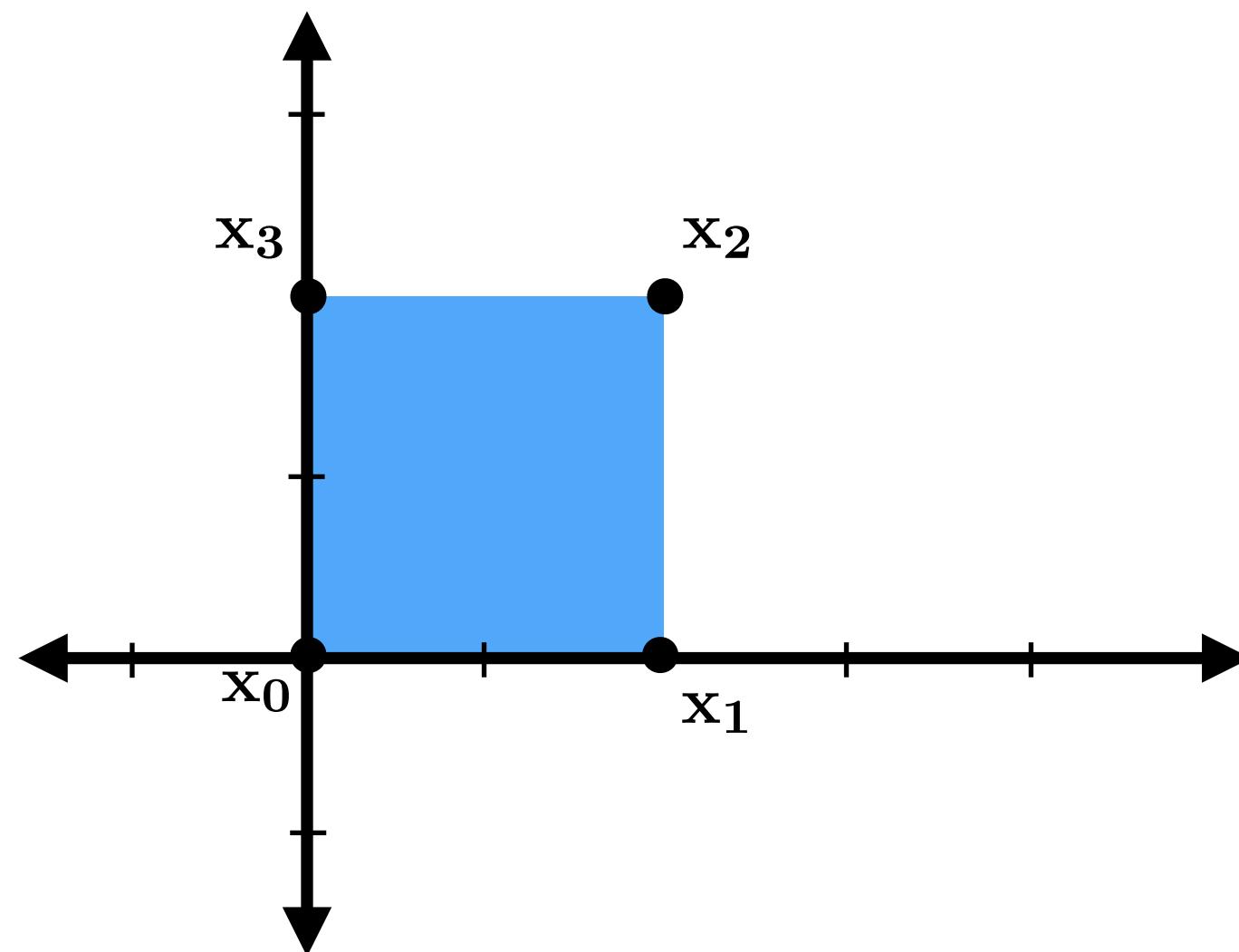
$$= \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

Rotation matrix (2D): another way...

$$\mathbf{R}_\theta = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$

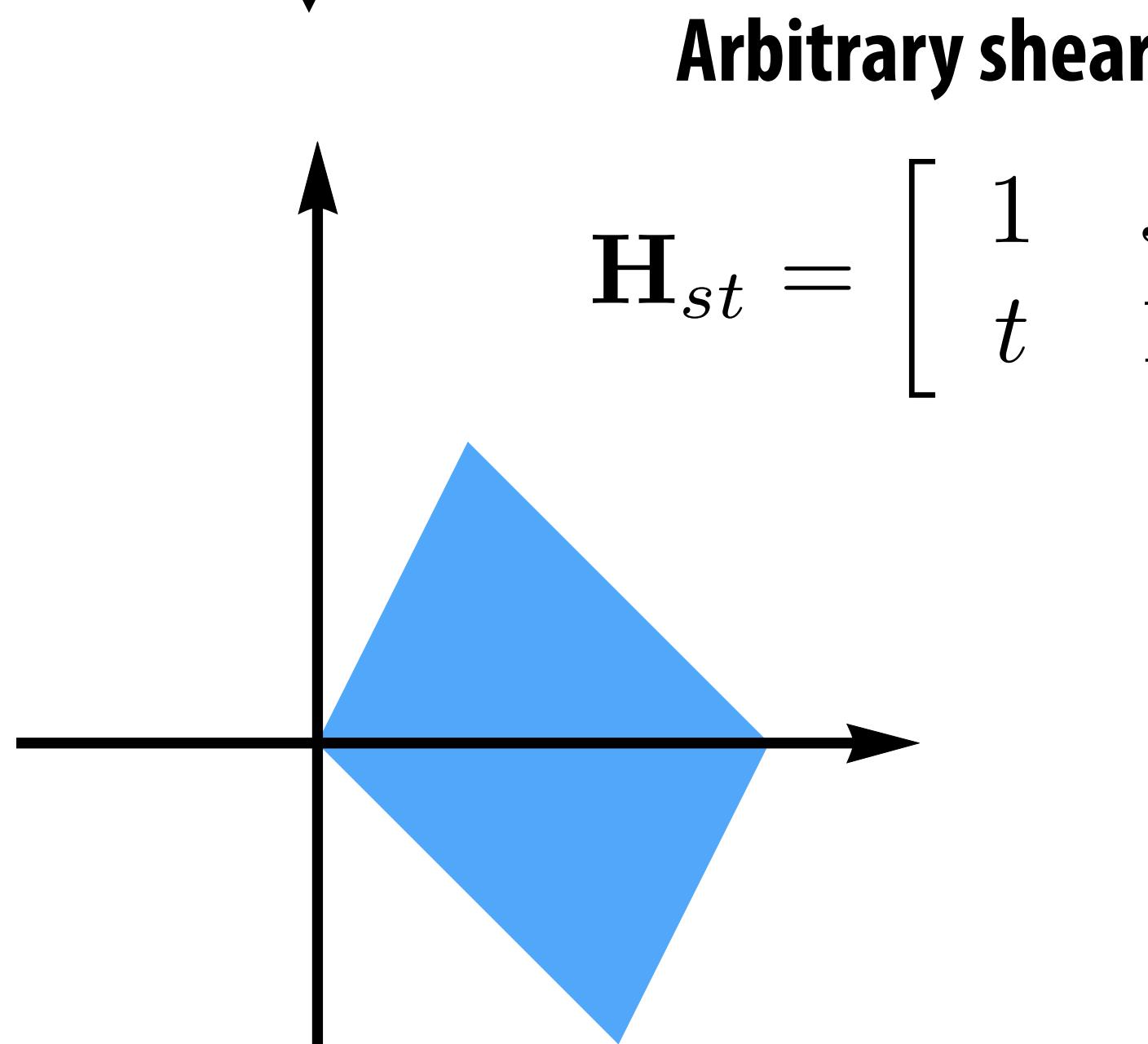


Shear

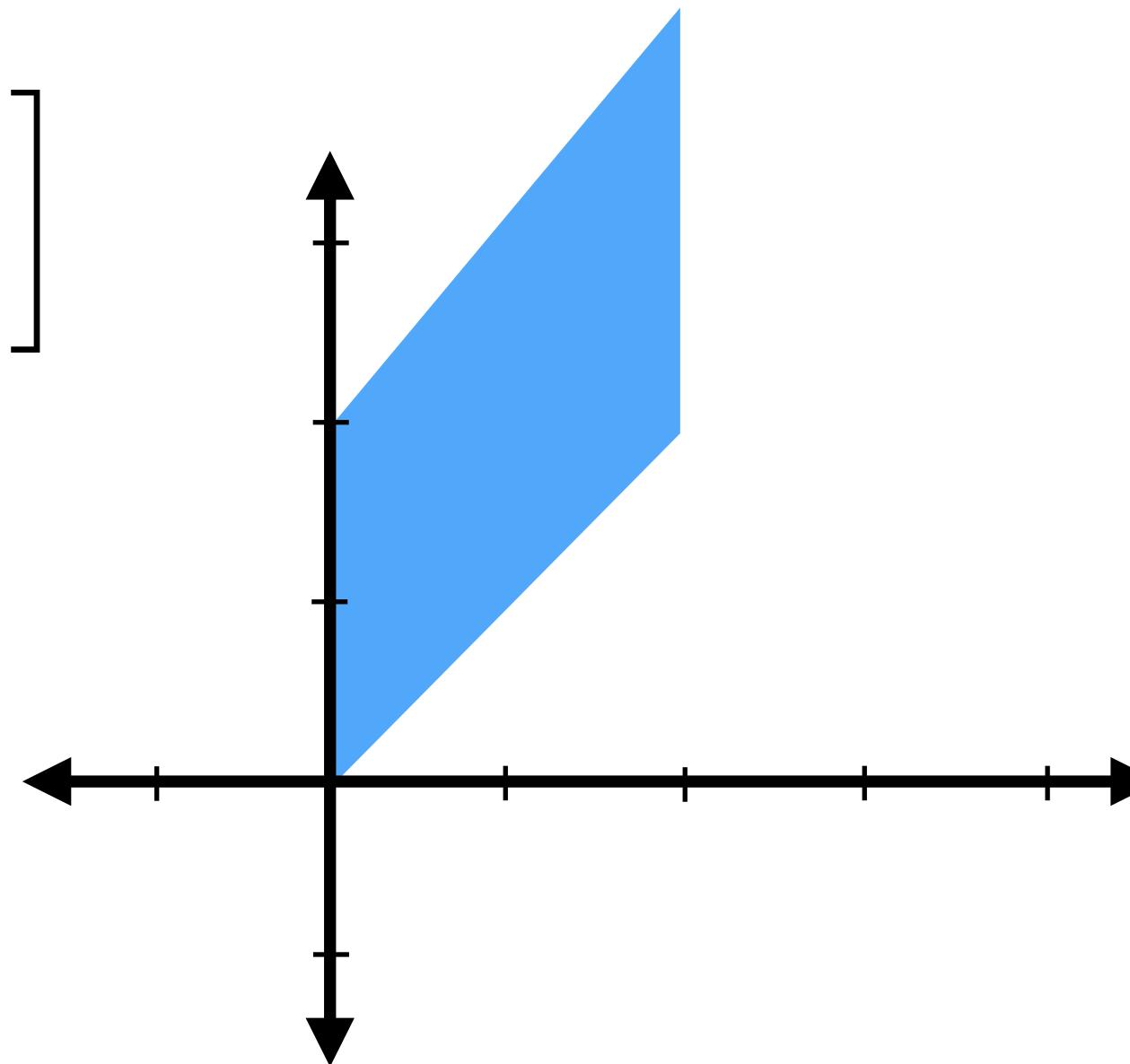


Shear in x :

$$H_{xs} = \begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix}$$



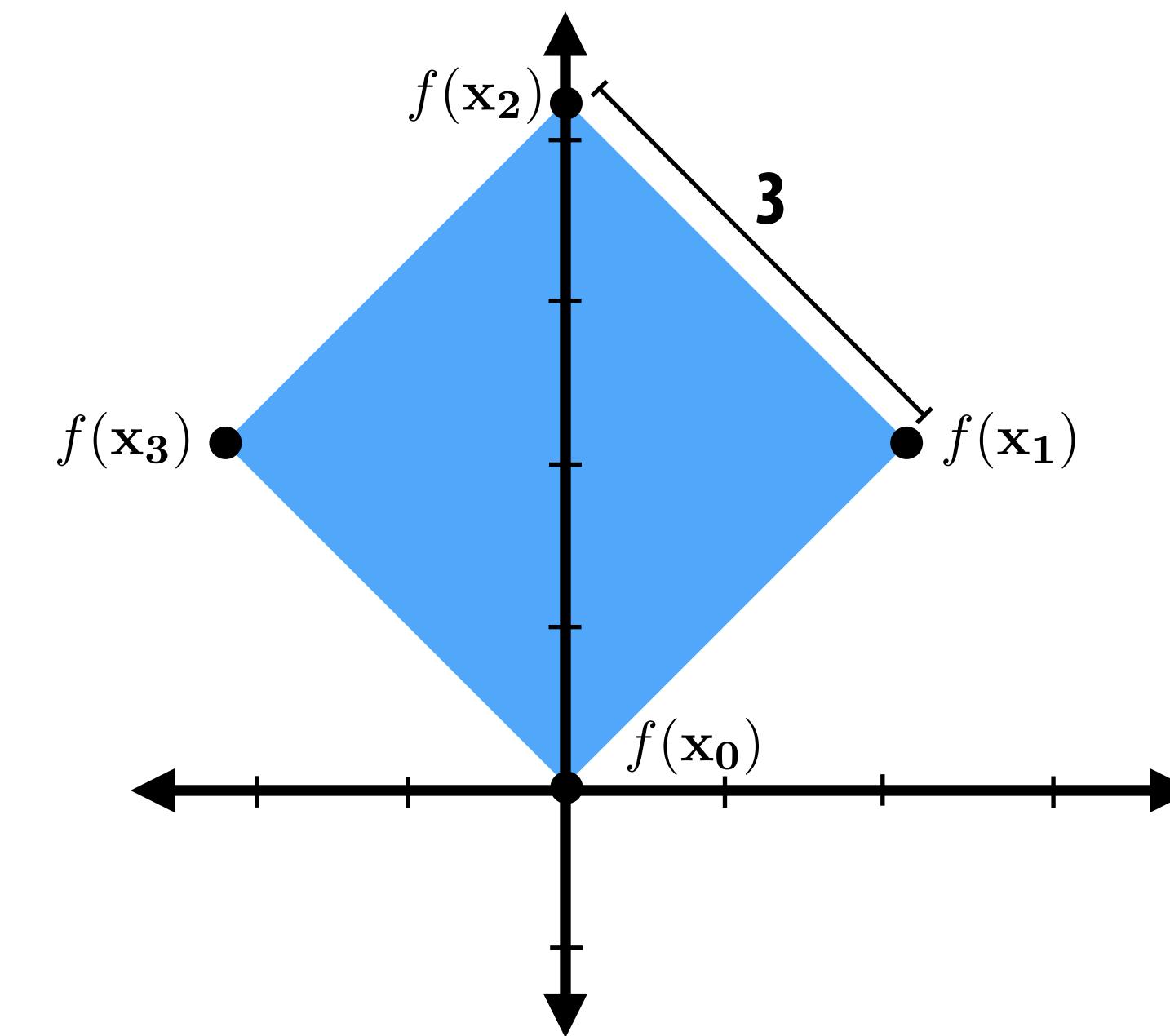
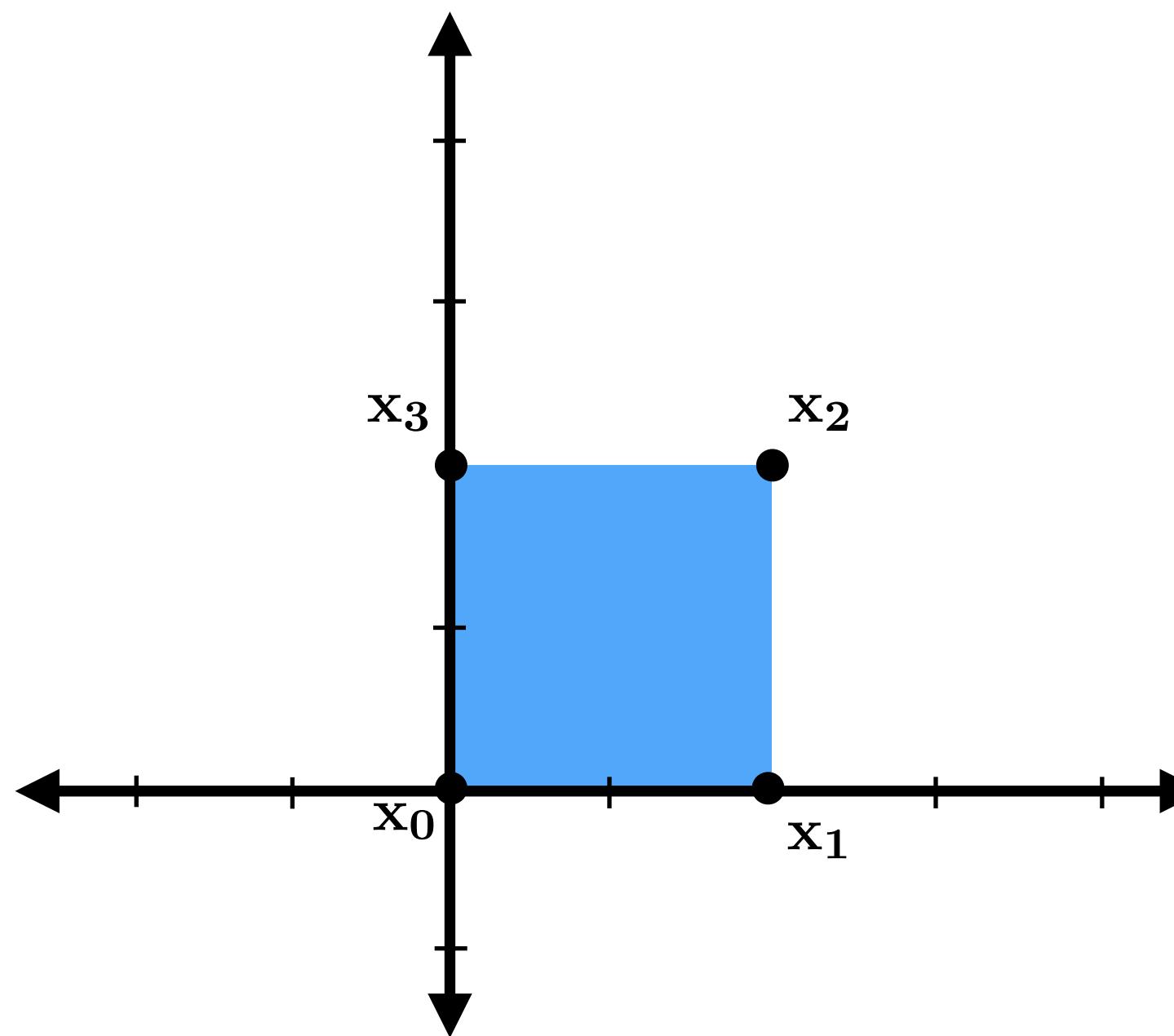
$$H_{st} = \begin{bmatrix} 1 & s \\ t & 1 \end{bmatrix}$$



Shear in y :

$$H_{ys} = \begin{bmatrix} 1 & 0 \\ s & 1 \end{bmatrix}$$

How do we compose linear transformations?



Compose linear transformations via matrix multiplication.

This example: uniform scale, followed by rotation

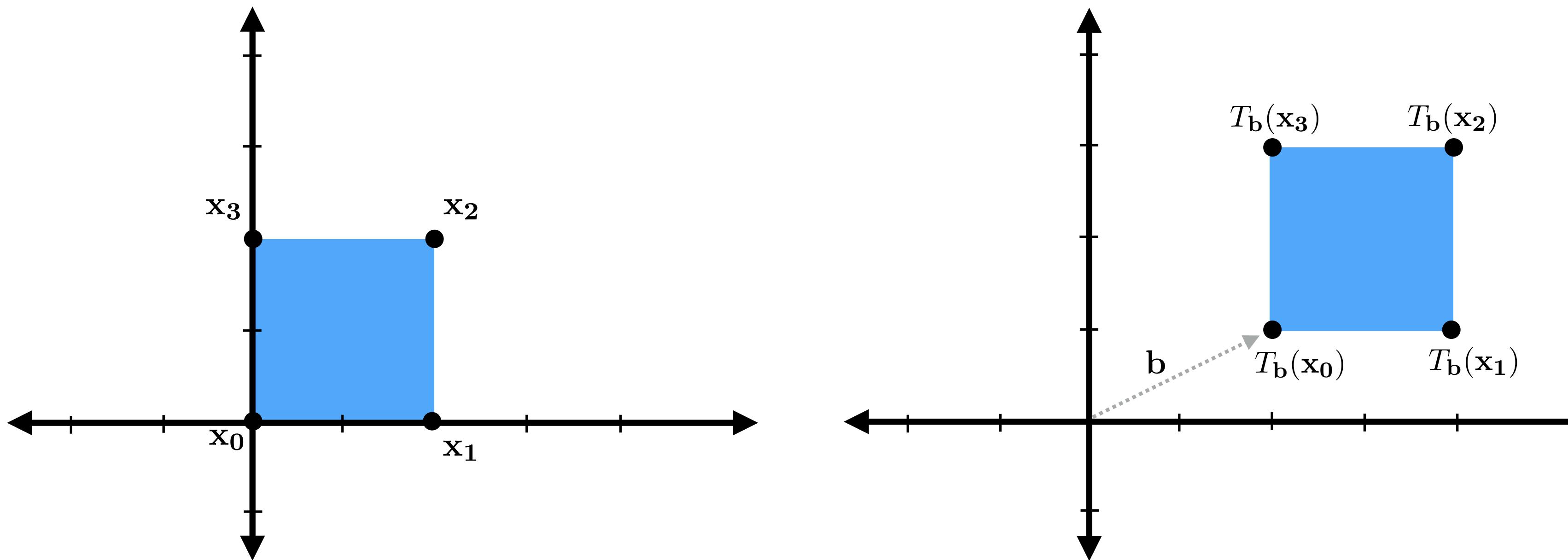
$$f(\mathbf{x}) = R_{\pi/4} \mathbf{S}_{[1.5, 1.5]} \mathbf{x} = \mathbf{M}\mathbf{x}$$

Where: $\mathbf{M} = R_{\pi/4} \mathbf{S}_{[1.5, 1.5]}$

Enables simple, efficient implementation: reduce complex chain of transformations to a single matrix multiplication!

How do we deal with translation? (Not linear)

$$T_b(\mathbf{x}) = \mathbf{x} + \mathbf{b}$$



Recall: translation is not a linear transform

- Output coefficients are not a linear combination of input coefficients
- Translation operation cannot be represented by a 2x2 matrix

$$\mathbf{x}_{\text{out}x} = \mathbf{x}_x + \mathbf{b}_x$$

$$\mathbf{x}_{\text{out}y} = \mathbf{x}_y + \mathbf{b}_y$$

2D homogeneous coordinates (2D-H)

Idea: represent 2D points with THREE values (“homogeneous coordinates”)

So the point (x, y) is represented as the 3-vector: $[x \ y \ 1]^T$

And transformations are represented a 3x3 matrices that transform these vectors.

Recover final 2D coordinates by dividing by “extra” (third) coordinate



$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} \Rightarrow \begin{bmatrix} x/w \\ y/w \end{bmatrix}$$

(More on this later...)

Example: scale and rotation in 2D-H coords

- For transformations that are already linear, not much changes:

$$\mathbf{S}_s = \begin{bmatrix} \mathbf{S}_x & 0 & 0 \\ 0 & \mathbf{S}_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \mathbf{R}_\theta = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Notice that the last row/column doesn't do anything interesting. E.g., for scaling:

$$\mathbf{S}_s \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{S}_x x \\ \mathbf{S}_y y \\ 1 \end{bmatrix}$$

Now we divide by the 3rd coordinate to get our final 2D coordinates (not too exciting!)

$$\begin{bmatrix} \mathbf{S}_x x \\ \mathbf{S}_y y \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} \mathbf{S}_x x / 1 \\ \mathbf{S}_y y / 1 \end{bmatrix} = \begin{bmatrix} \mathbf{S}_x x \\ \mathbf{S}_y y \end{bmatrix}$$

(Will get more interesting when we talk about *perspective*...)

Translation in 2D homogeneous coordinates

Translation expressed as 3x3 matrix multiplication:

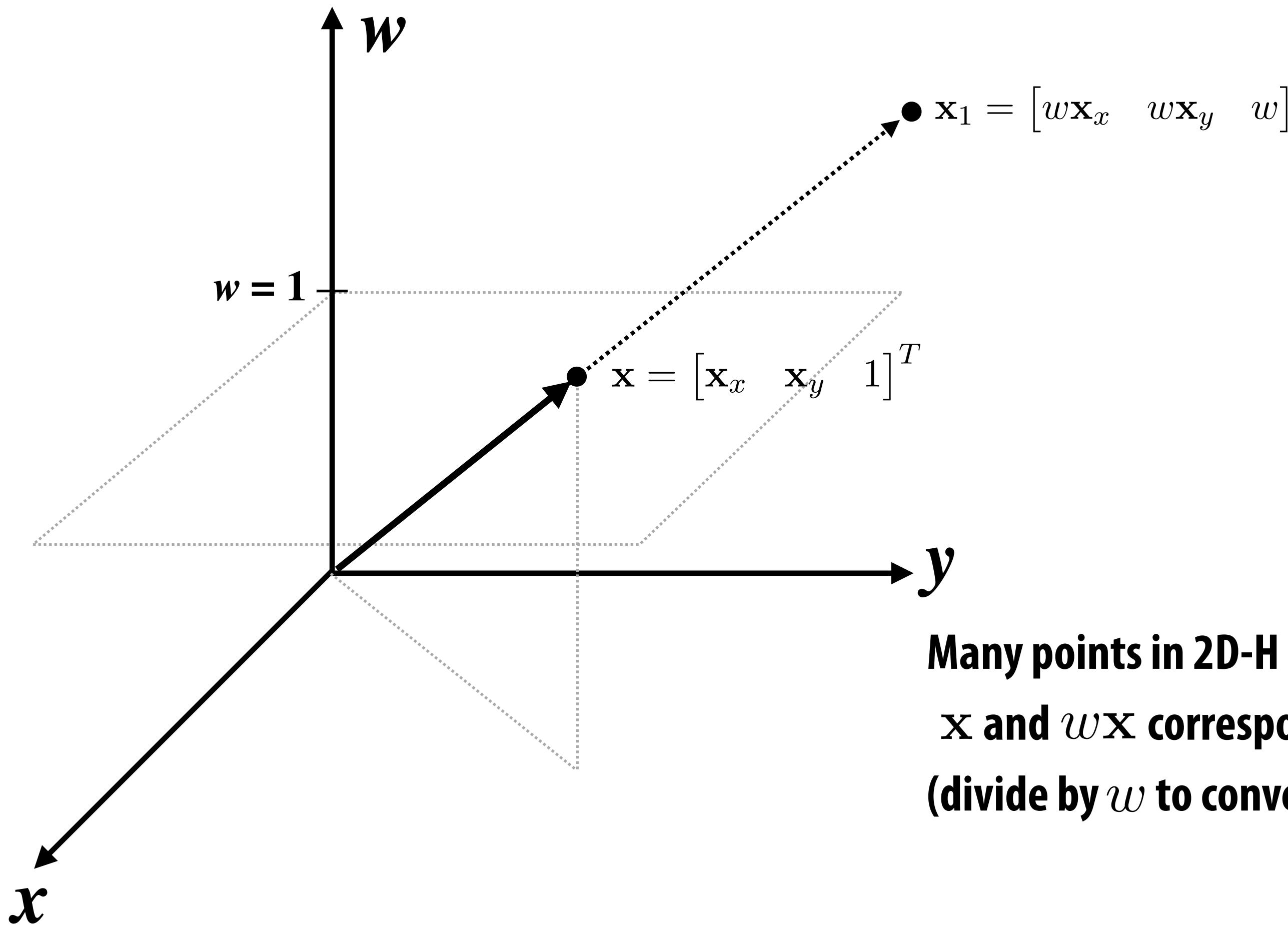
$$T_b = \begin{bmatrix} 1 & 0 & b_x \\ 0 & 1 & b_y \\ 0 & 0 & 1 \end{bmatrix}$$

$$T_b \mathbf{x} = \begin{bmatrix} 1 & 0 & b_x \\ 0 & 1 & b_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_x \\ x_y \\ 1 \end{bmatrix} = \begin{bmatrix} x_x + b_x \\ x_y + b_y \\ 1 \end{bmatrix}$$

(remember: just a linear combination of columns!)

Cool: homogeneous coordinates let us encode translations as *linear* transformations!

Homogeneous coordinates: some intuition

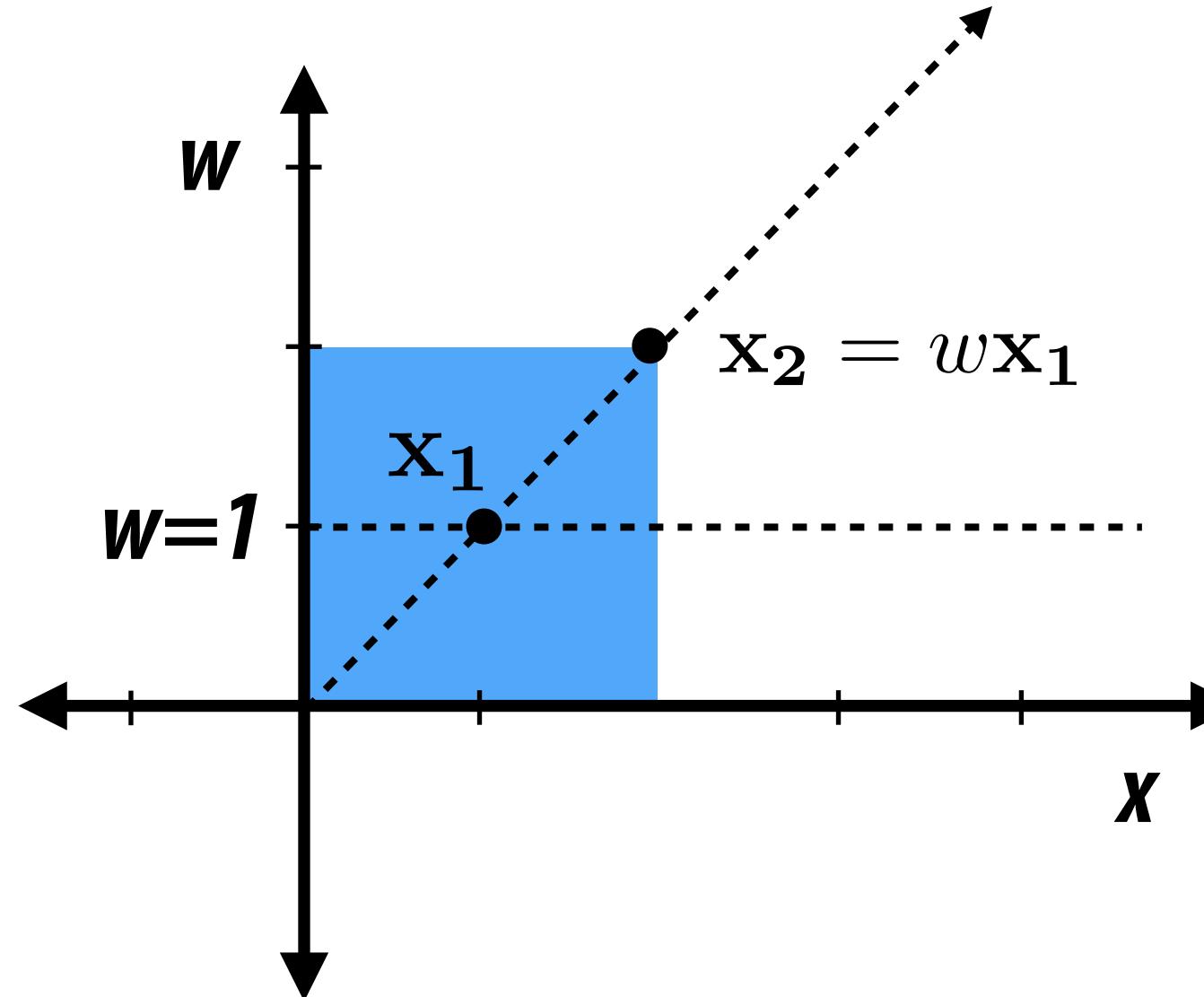


**Many points in 2D-H correspond to same point in 2D
 \mathbf{x} and $w\mathbf{x}$ correspond to the same 2D point
(divide by w to convert 2D-H back to 2D)**

Translation is a shear in *x* and *y* in 2D-H space

$$\mathbf{T}_b \mathbf{x} = \begin{bmatrix} 1 & 0 & \mathbf{b}_x \\ 0 & 1 & \mathbf{b}_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} w\mathbf{x}_x \\ w\mathbf{x}_y \\ w \end{bmatrix} = \begin{bmatrix} w\mathbf{x}_x + w\mathbf{b}_x \\ w\mathbf{x}_y + w\mathbf{b}_y \\ w \end{bmatrix}$$

Translation = shear in homogeneous space

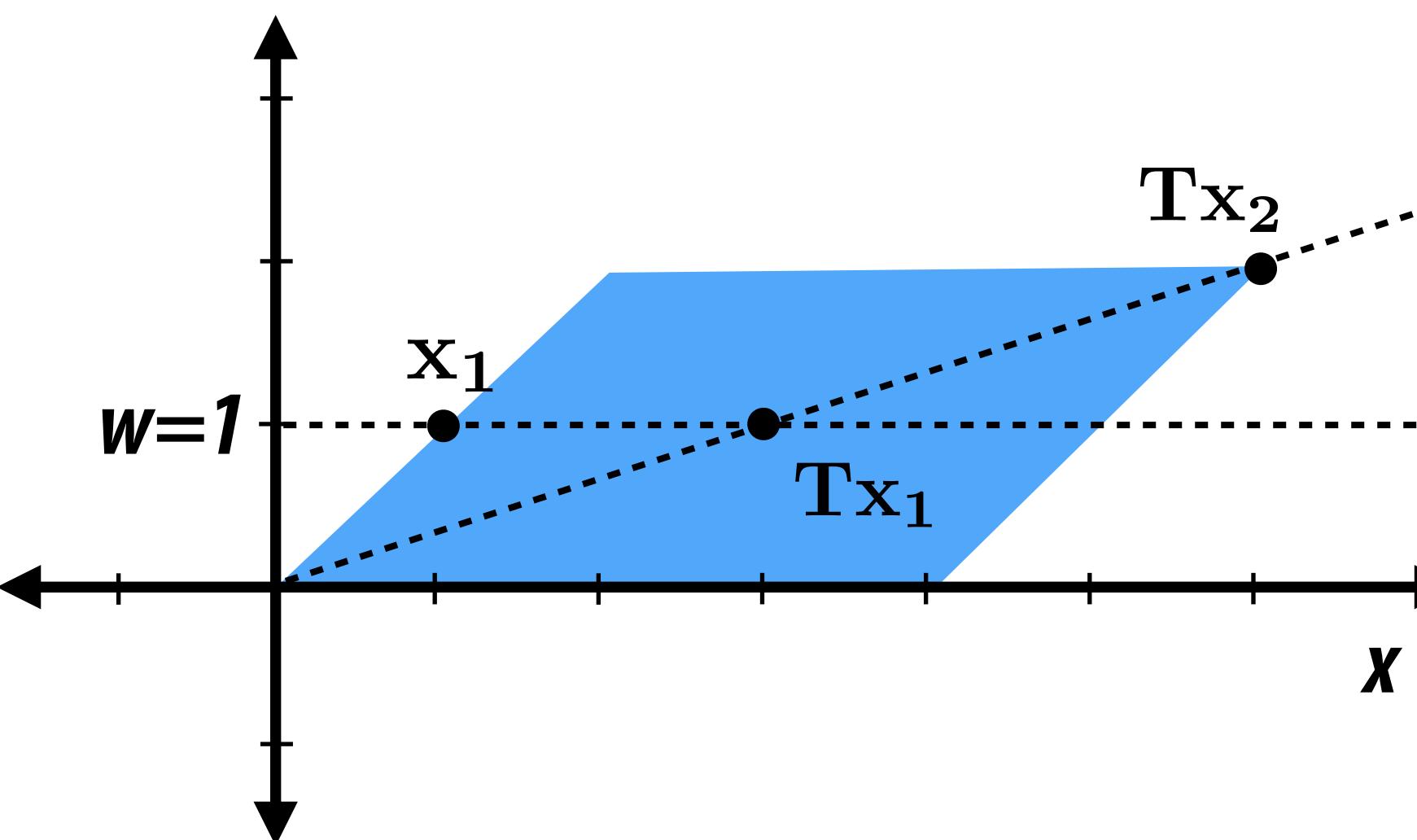


For simplicity, consider 1D-H:

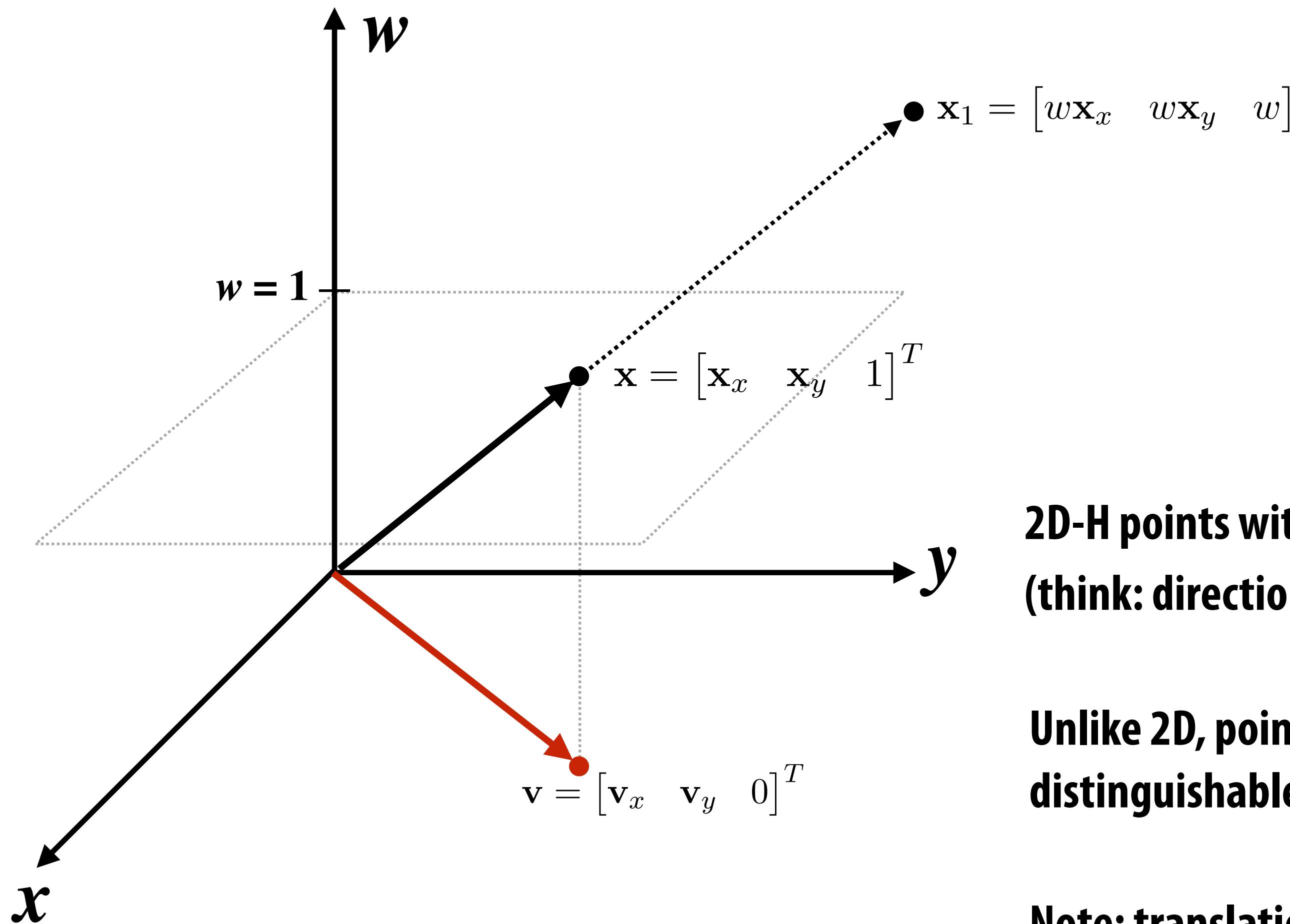
Translate by $t=2$: $T = \begin{bmatrix} 1 & t \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$

Recall: this is a shear in homogeneous x .

1D translation is affine in 1D ($x + t$),
but it is linear in 1D-H



Homogeneous coordinates: points vs. vectors



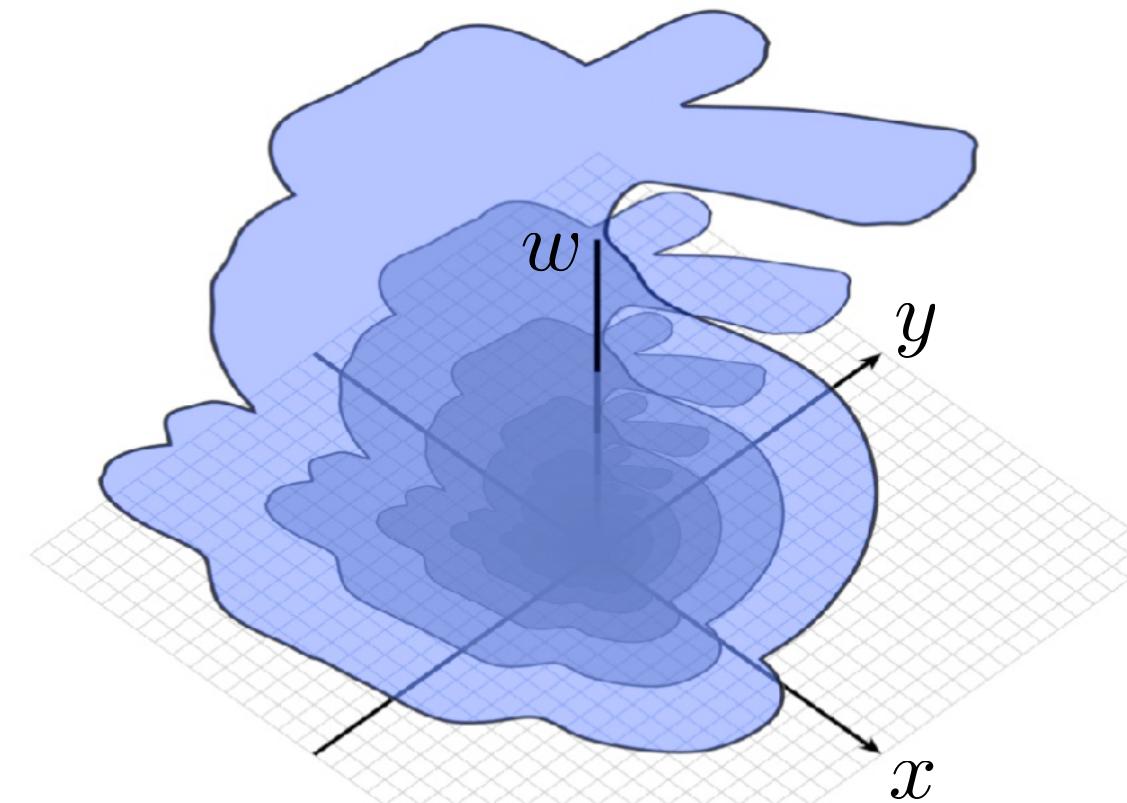
2D-H points with $w=0$ represent 2D vectors
(think: directions are points at infinity)

Unlike 2D, points and directions are
distinguishable by their representation in 2D-H

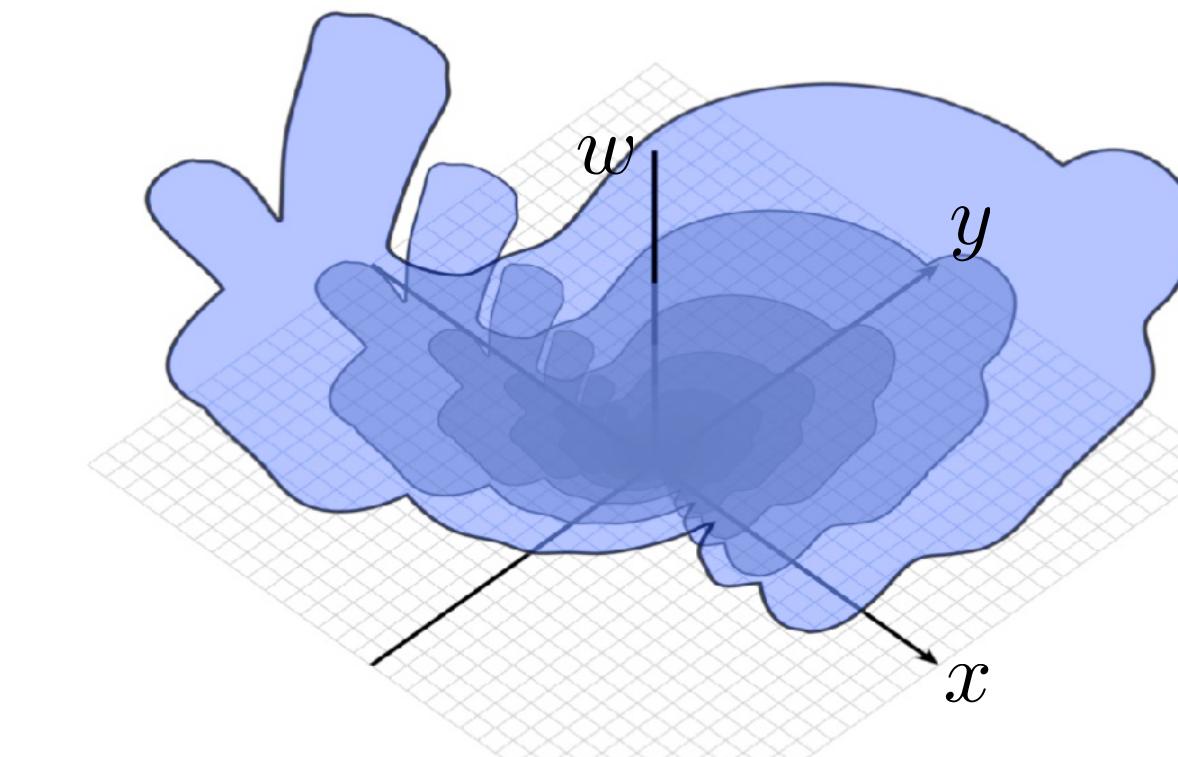
Note: translation does not modify directions:

$$\mathbf{T}_b \mathbf{v} = \begin{bmatrix} 1 & 0 & b_x \\ 0 & 1 & b_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{v}_x \\ \mathbf{v}_y \\ 0 \end{bmatrix} = \begin{bmatrix} \mathbf{v}_x \\ \mathbf{v}_y \\ 0 \end{bmatrix}$$

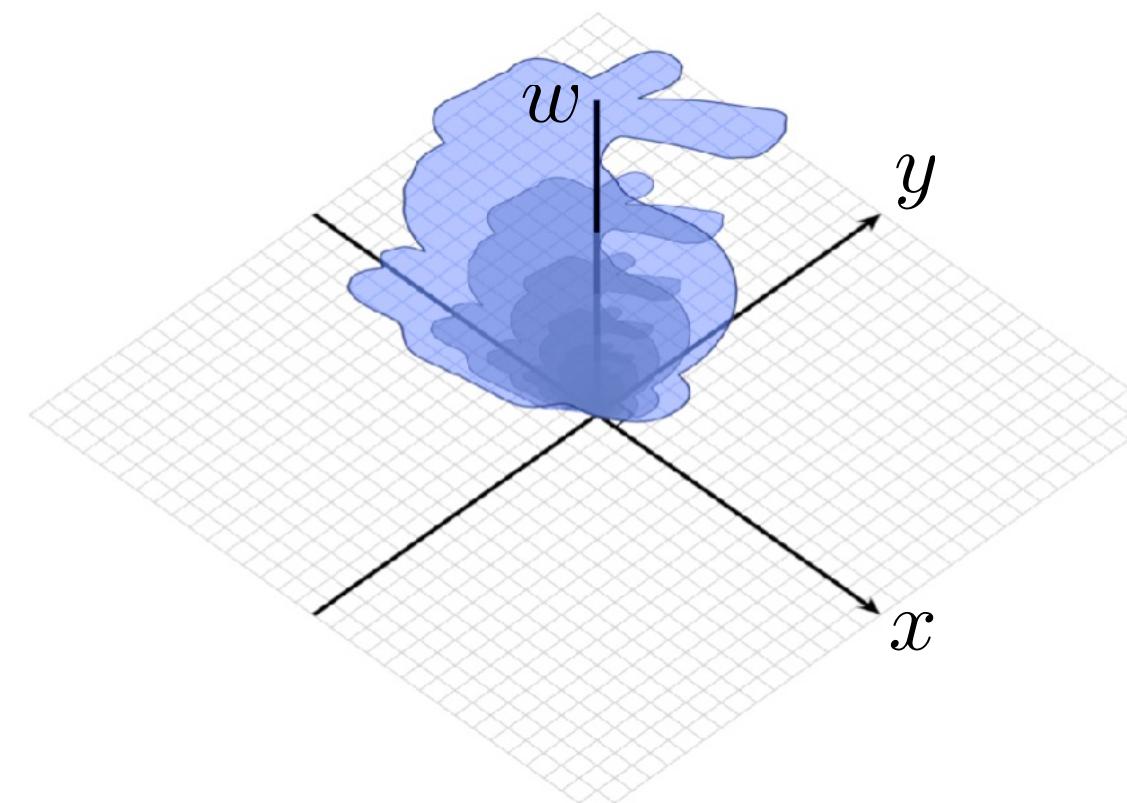
Visualizing 2D transformations in 2D-H



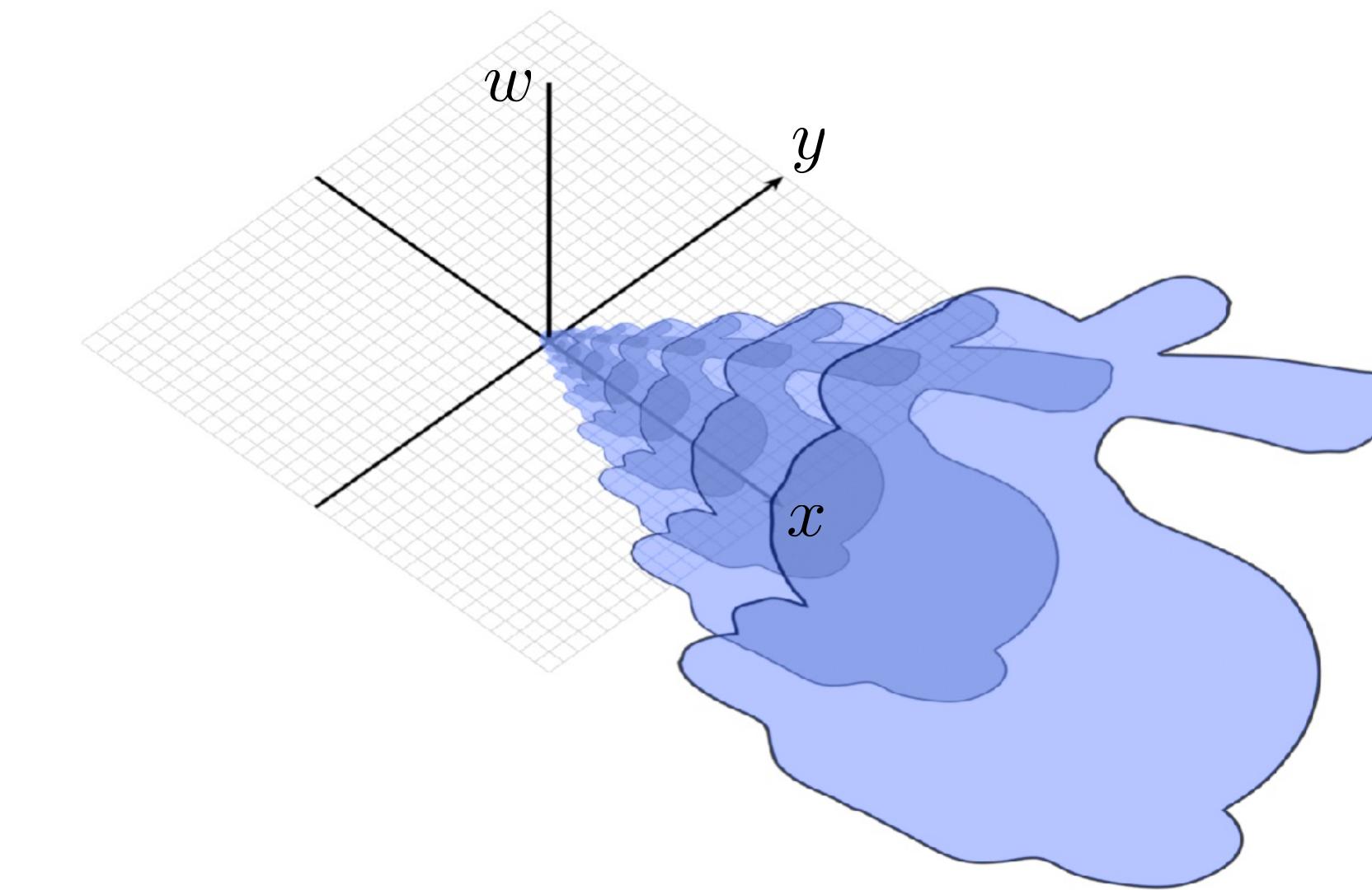
Original shape in 2D can be viewed as many copies, uniformly scaled by w.



2D rotation \leftrightarrow rotate around w



2D scale \leftrightarrow scale x and y; preserve w
(Question: what happens to 2D shape if you scale x, y, and w uniformly?)



**2D translate \leftrightarrow shear in 2D-H
(LINEAR!)**

Moving to 3D (and 3D-H)

Represent 3D transformations as 3x3 matrices and 3D-H transformations as 4x4 matrices

Scale:

$$\begin{array}{c} \text{3D} \\ \mathbf{S}_s = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \end{bmatrix} \quad \text{3D-H} \\ \mathbf{S}_s = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{array}$$

Shear (in x, based on y,z position):

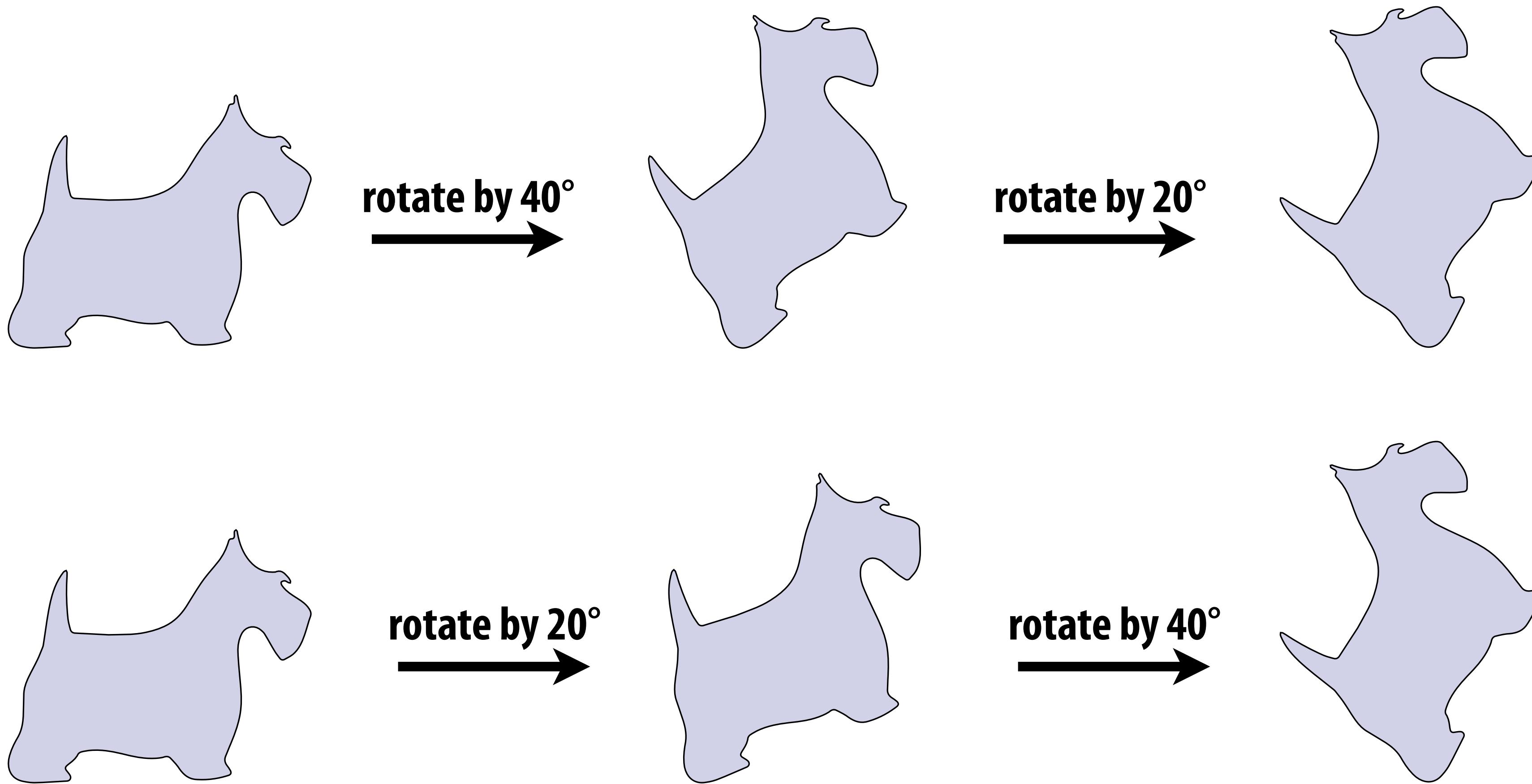
$$\mathbf{H}_{x,d} = \begin{bmatrix} 1 & d_y & d_z \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \mathbf{H}_{x,d} = \begin{bmatrix} 1 & d_y & d_z & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translate:

$$\begin{array}{c} \text{3D-H} \\ \mathbf{T}_b = \begin{bmatrix} 1 & 0 & 0 & b_x \\ 0 & 1 & 0 & b_y \\ 0 & 0 & 1 & b_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{array}$$

Commutativity of rotations—2D

- In 2D, order of rotations doesn't matter:



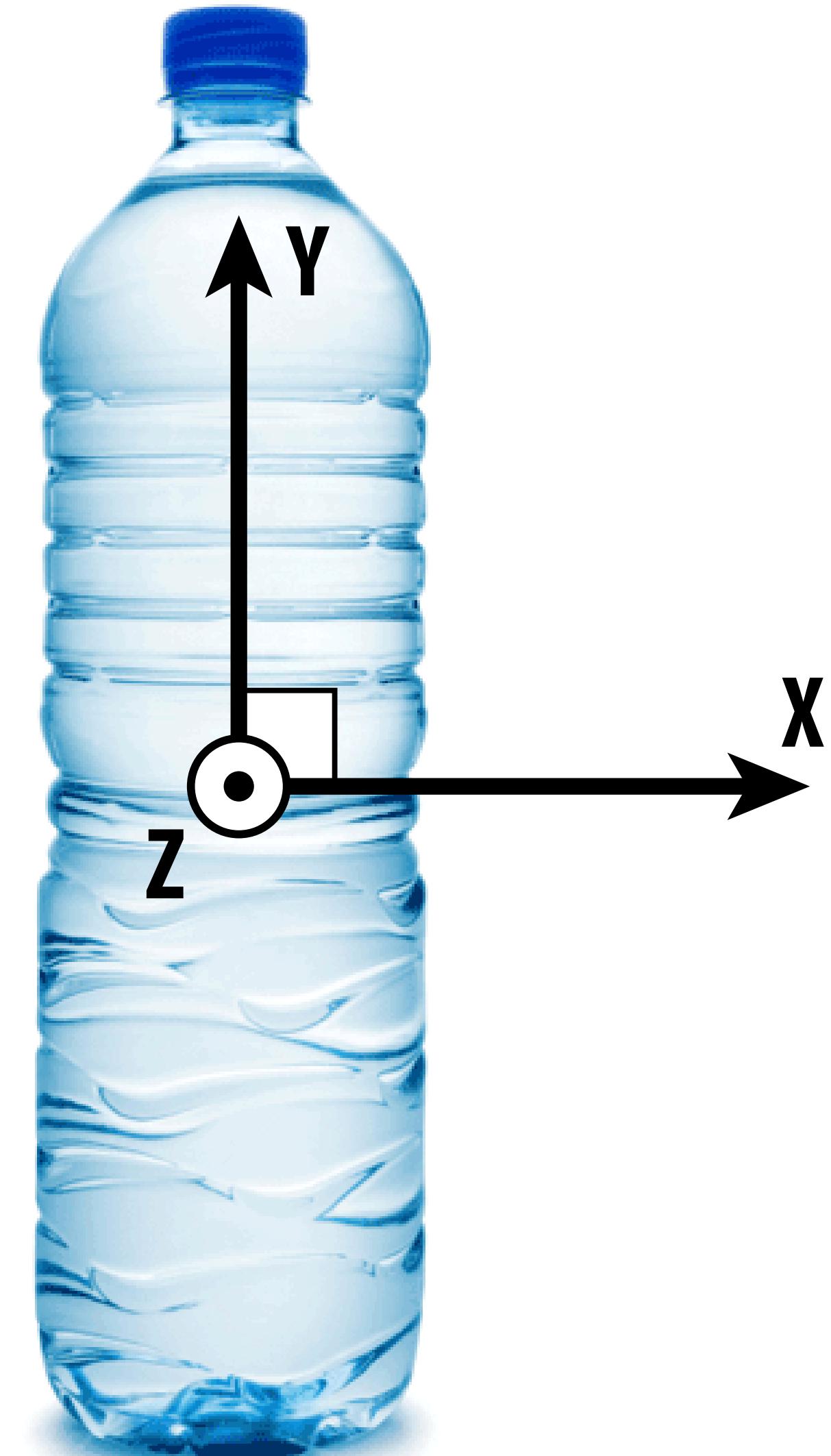
Same result! (“2D rotations commute”)

Commutativity of rotations—3D

- What about in 3D?

- IN-CLASS ACTIVITY:

- Rotate 90° around Y, then 90° around Z, then 90° around X
- Rotate 90° around Z, then 90° around Y, then 90° around X
- (Was there any difference?)

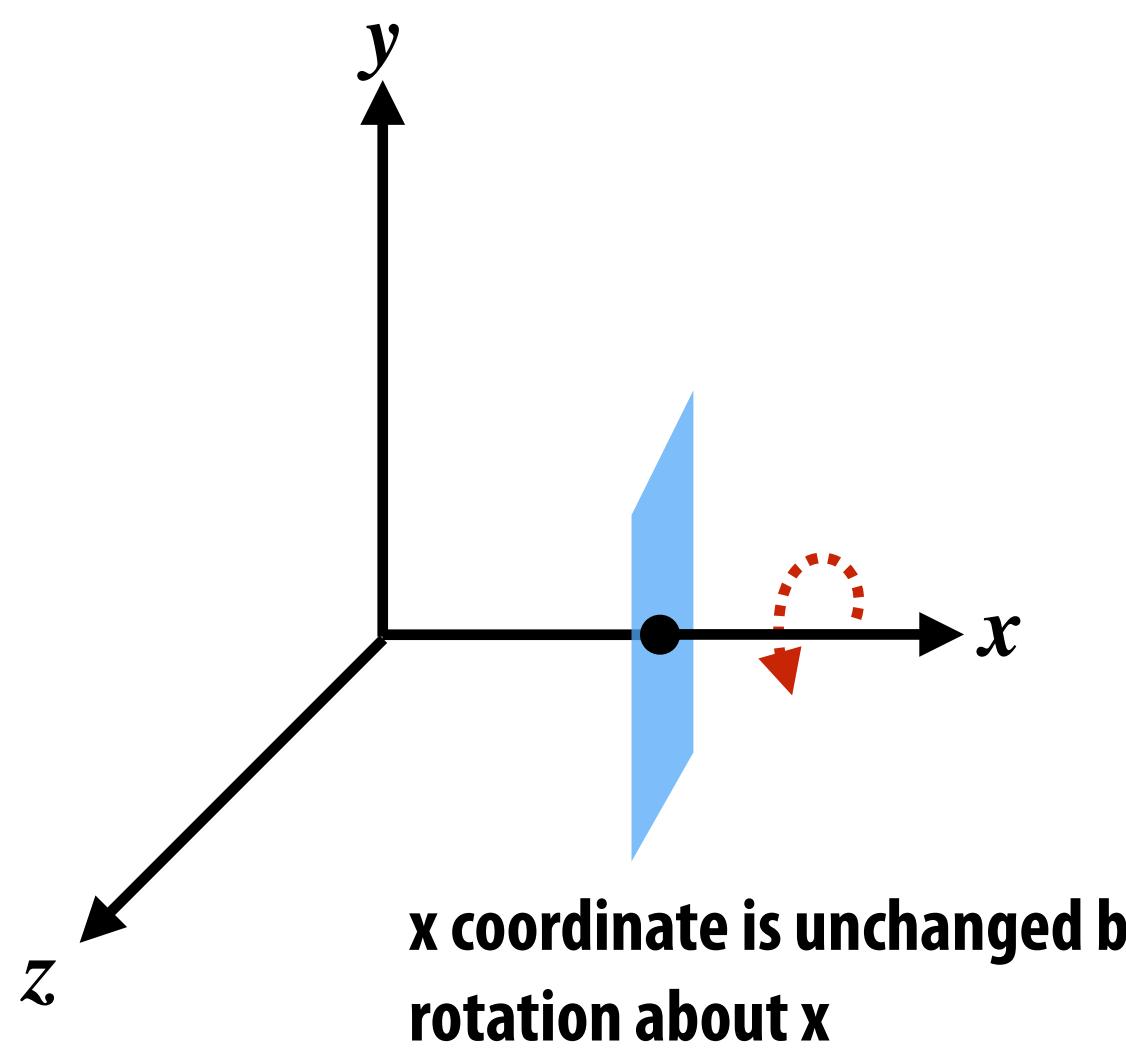


CONCLUSION: bad things can happen if we're not careful about the order in which we apply rotations!

Rotations in 3D

Rotation about x axis:

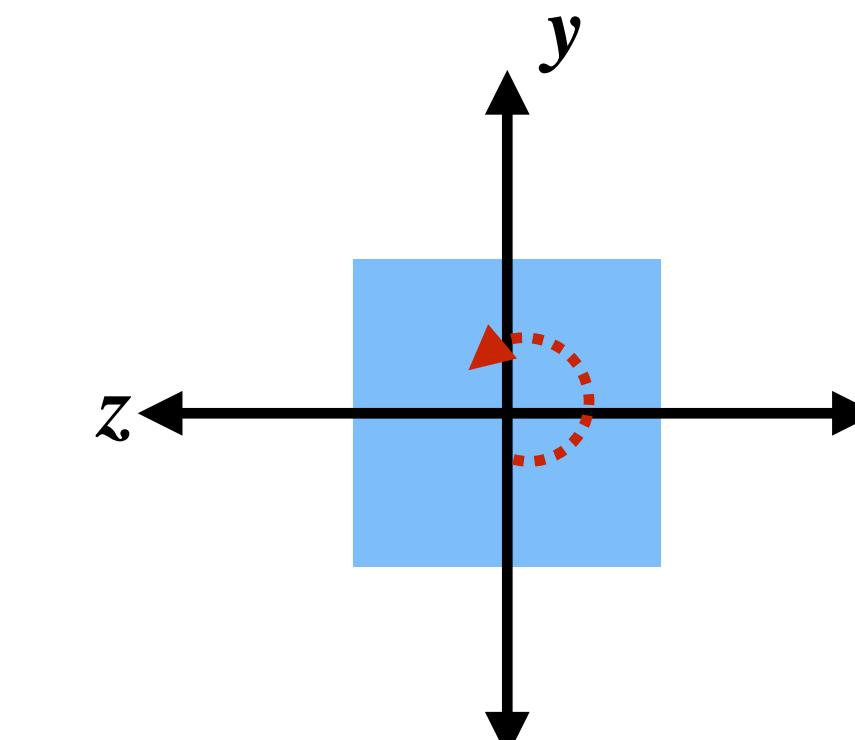
$$R_{x,\theta} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \\ 0 & \sin \theta & \cos \theta \end{bmatrix}$$



Rotation about y axis:

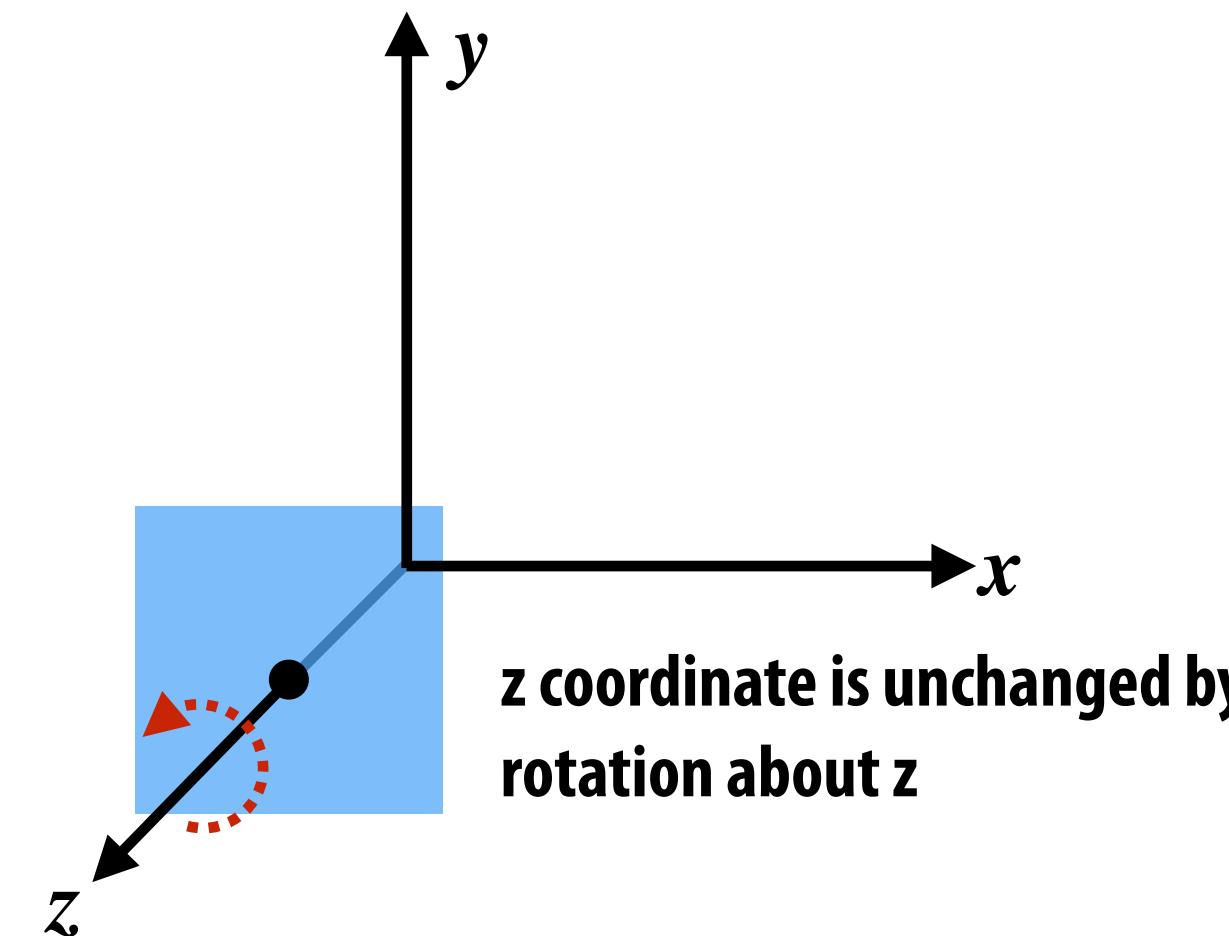
$$R_{y,\theta} = \begin{bmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{bmatrix}$$

View looking down -x axis:

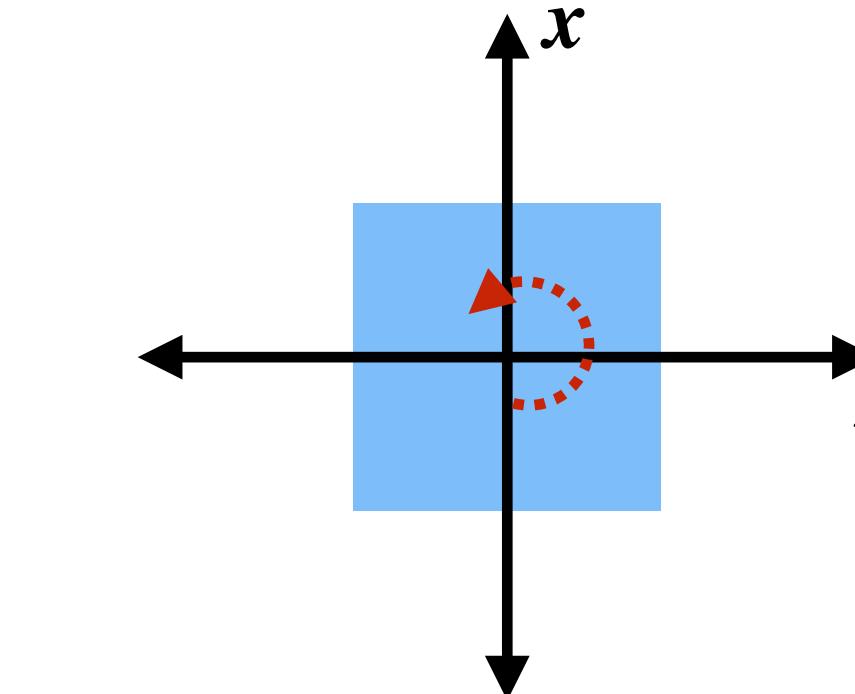


Rotation about z axis:

$$R_{z,\theta} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

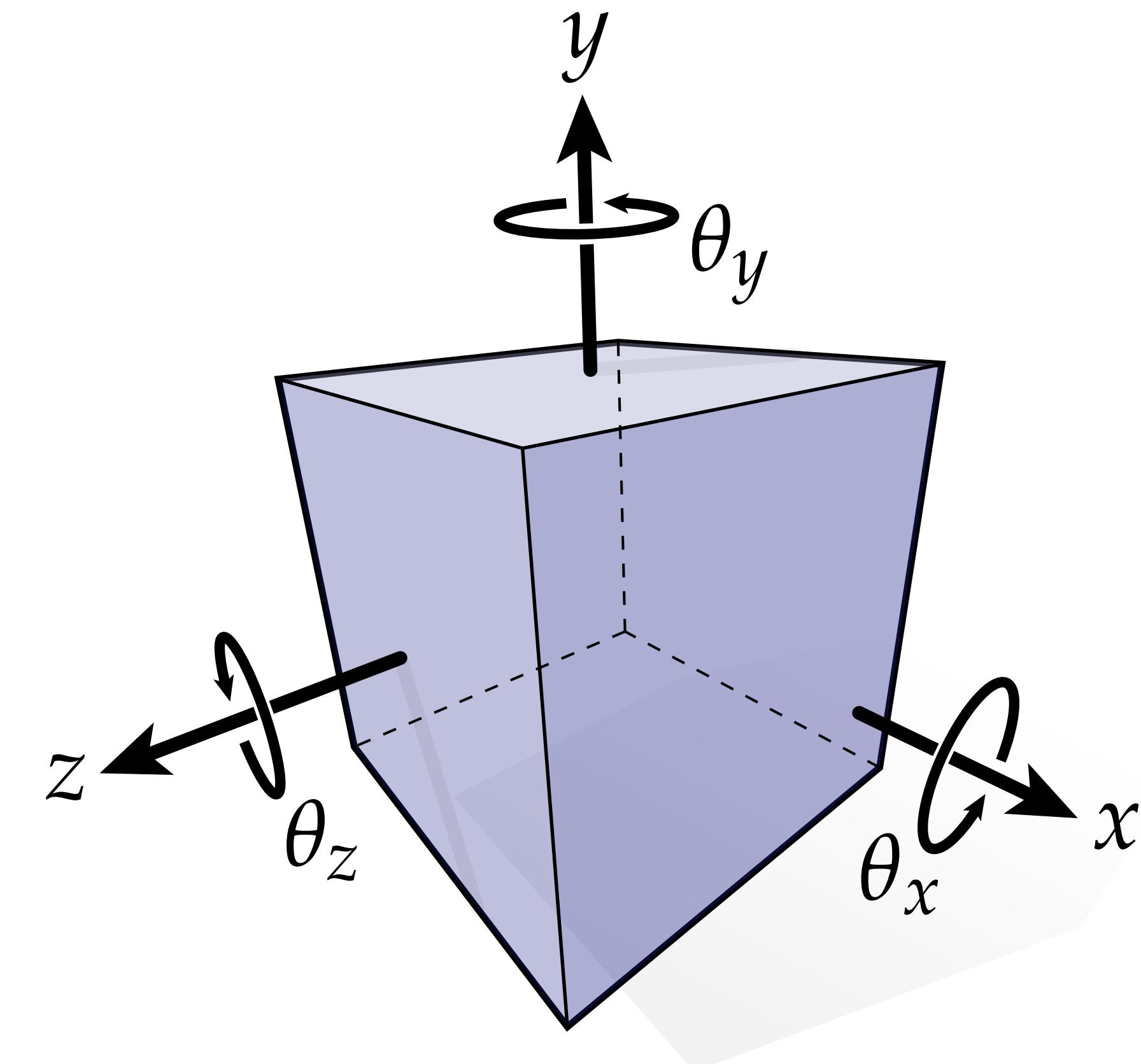
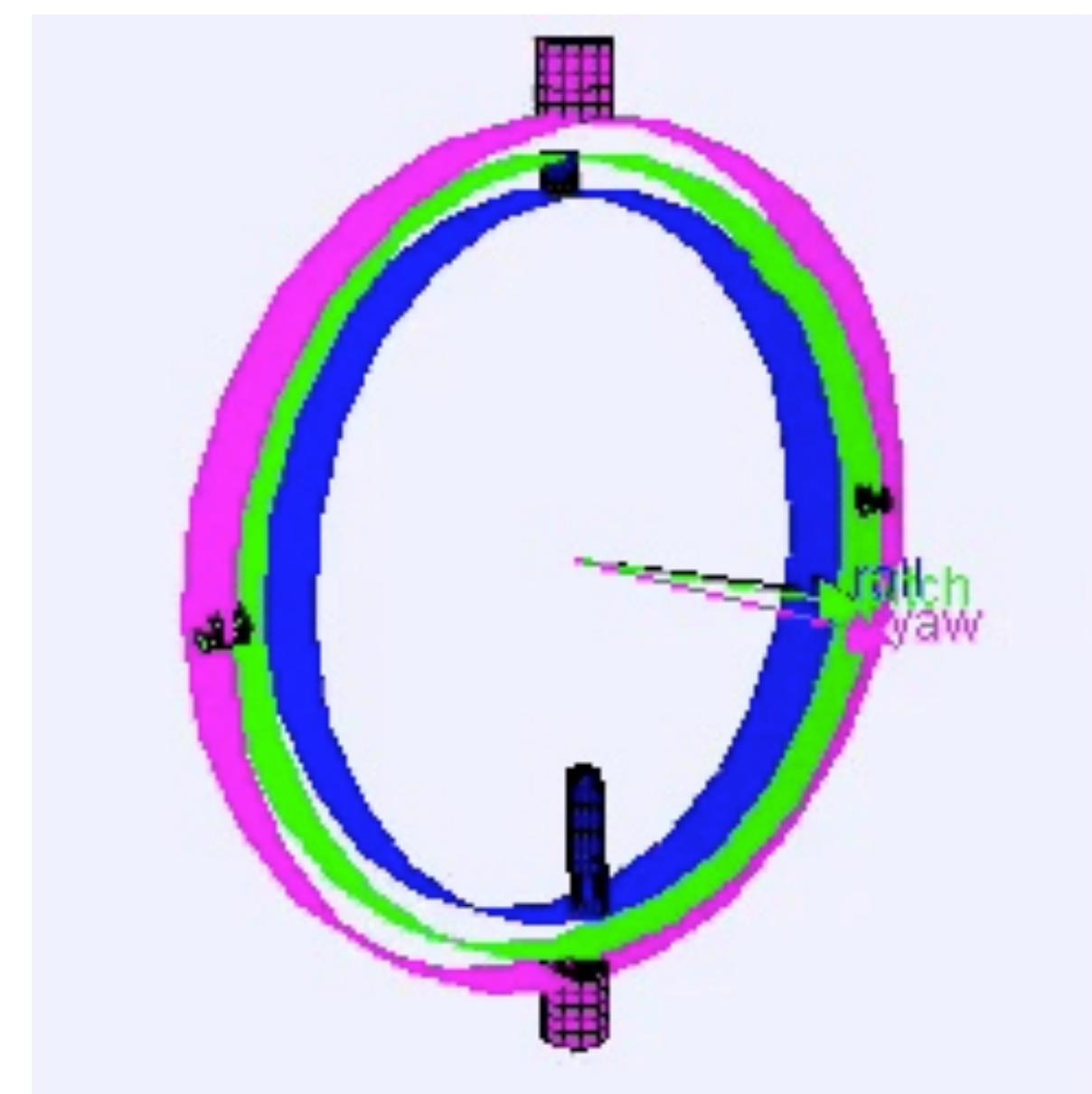


View looking down -y axis:



Representing rotations in 3D—euler angles

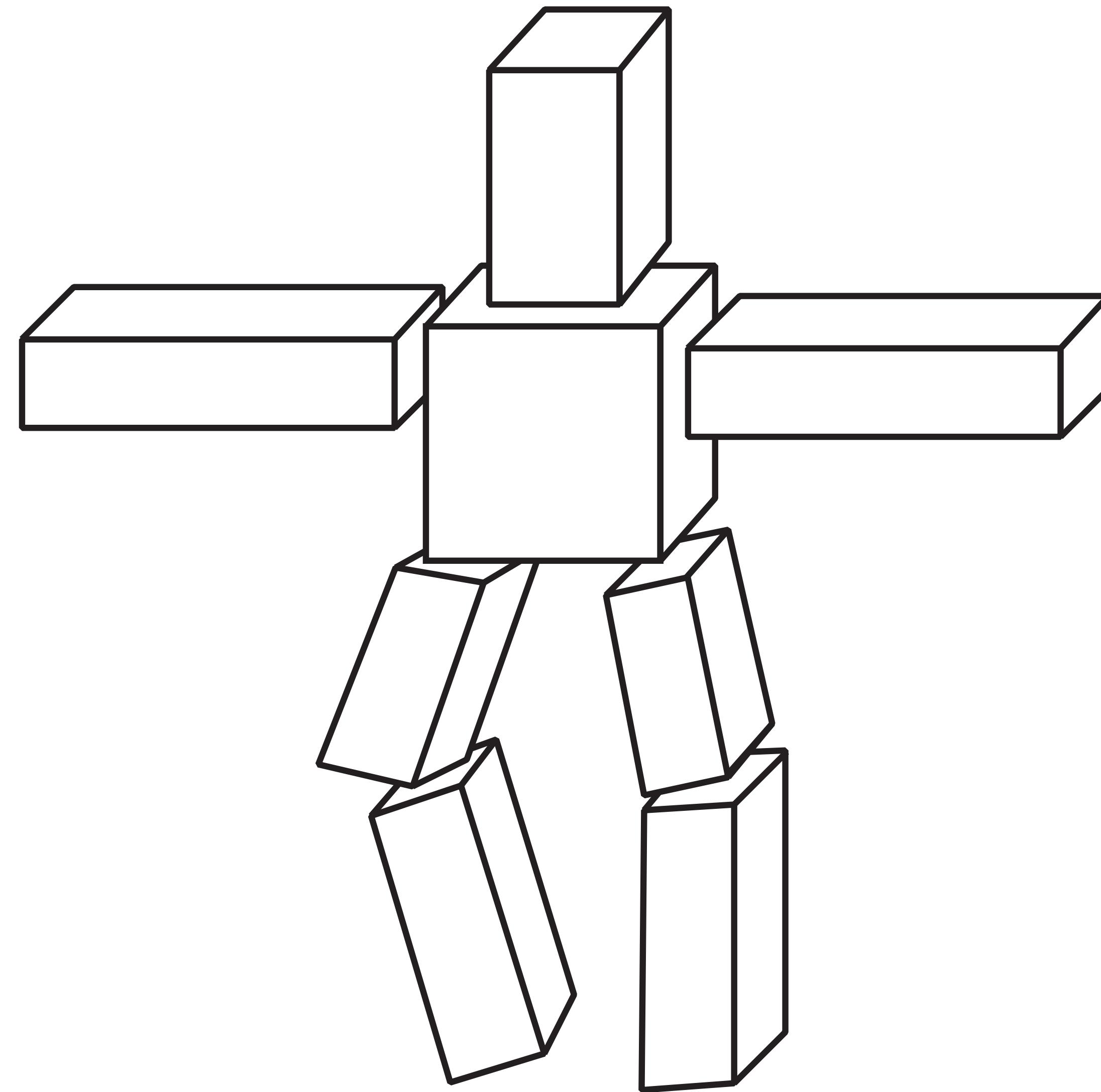
- How do we express rotations in 3D?
- One idea: we know how to do 2D rotations
- Why not simply apply rotations around the three axes? (X,Y,Z)
- Scheme is called *Euler angles*
- PROBLEM: “Gimbal Lock”



Alternative representations of 3D rotations

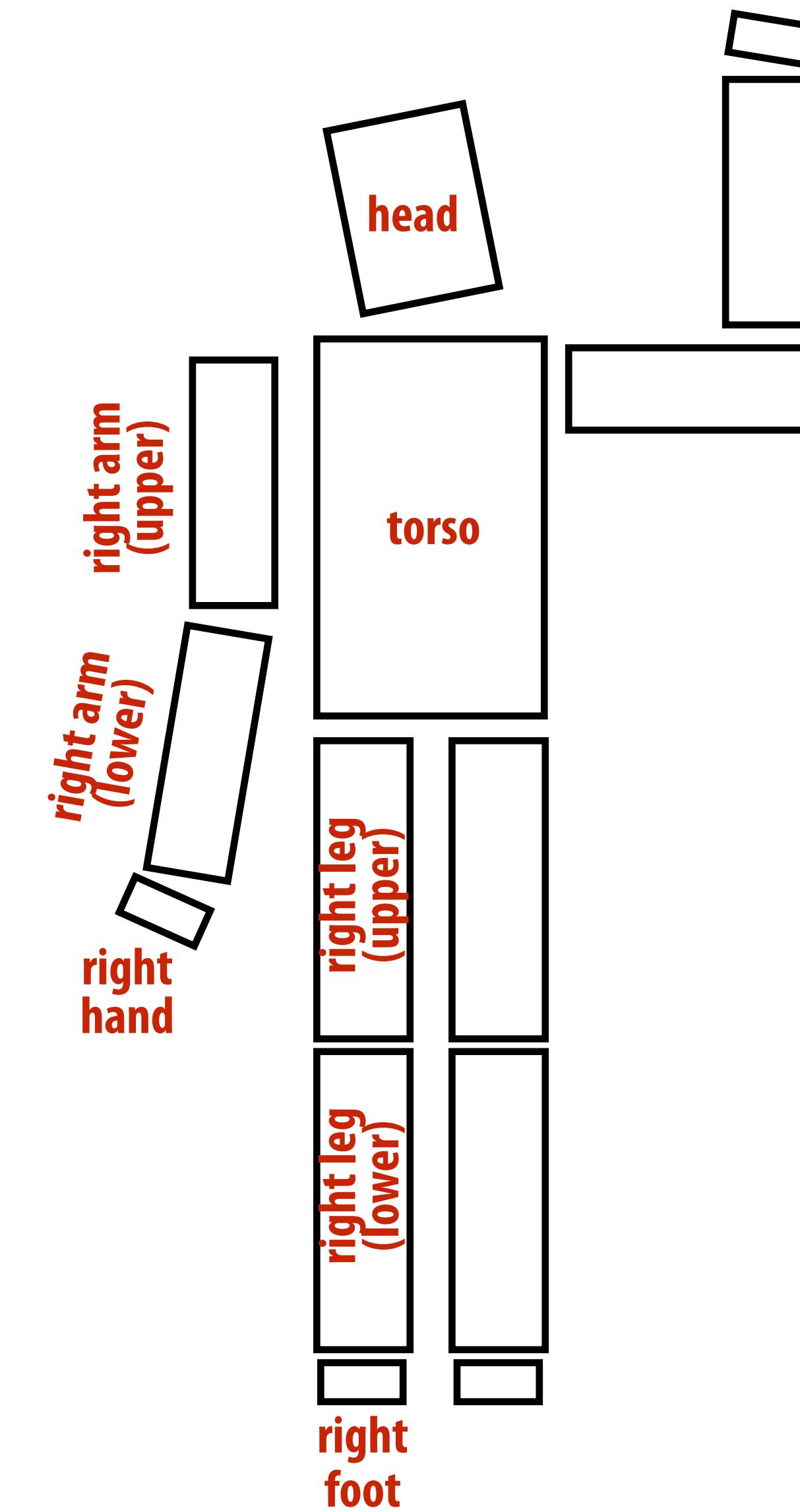
- Axis-angle rotations
- Quaternions (not today)

Let's make that cube person...



Skeleton - hierarchical representation

torso
head
right arm
 upper arm
 lower arm
 hand
left arm
 upper arm
 lower arm
 hand
right leg
 upper leg
 lower leg
 foot
left leg
 upper leg
 lower leg
 foot



Hierarchical representation

- **Grouped representation (tree)**
 - Each group contains subgroups and/or shapes
 - Each group is associated with a transform *relative to parent group*
 - Transform on leaf-node shape is concatenation of all transforms on path from root node to leaf
 - Changing a group's transform affects all descendent parts
 - Allows high level editing by changing only one node
 - E.g. raising left arm requires changing only one transform for that group

Skeleton - hierarchical representation

Transform used for drawing:

$T_{(0,10)}$

$T_{(0,10)} T_{(0,5)}$

$T_{(0,10)} T_{(0,5)} R_{\text{head}}$

$T_{(0,10)}$

$T_{(0,10)} T_{(-2,3)}$

$T_{(0,10)} T_{(-2,3)} R_{\text{sho}}$

$T_{(0,10)} T_{(-2,3)} R_{\text{sho}} T_{(0,-3)}$

$T_{(0,10)} T_{(-2,3)} R_{\text{sho}} T_{(0,-3)} R_{\text{elb}}$

$T_{(0,10)} T_{(-2,3)} R_{\text{sho}} T_{(0,-3)} R_{\text{elb}} T_{(0,-3)}$

$T_{(0,10)} T_{(-2,3)} R_{\text{sho}} T_{(0,-3)} R_{\text{elb}} T_{(0,-3)} R_{\text{wr}}$

Sequence of commands (transformation and drawing commands):

`translate(0, 10); // person centered at (0,10)`

`drawTorso();`

`pushTransform(); // push a copy of transform onto stack`

`translate(0, 5); // right-multiply onto current transform`

`rotate(headRotation); // right-multiply onto current transform`

`drawHead();`

`popTransform(); // pop current transform off stack`

`pushTransform();`

`translate(-2, 3);`

`rotate(rightShoulderRotation);`

`drawUpperArm();`

`pushTransform();`

`translate(0, -3);`

`rotate(elbowRotation);`

`drawLowerArm();`

`pushTransform();`

`translate(0, -3);`

`rotate(wristRotation);`

`drawHand();`

`popTransform();`

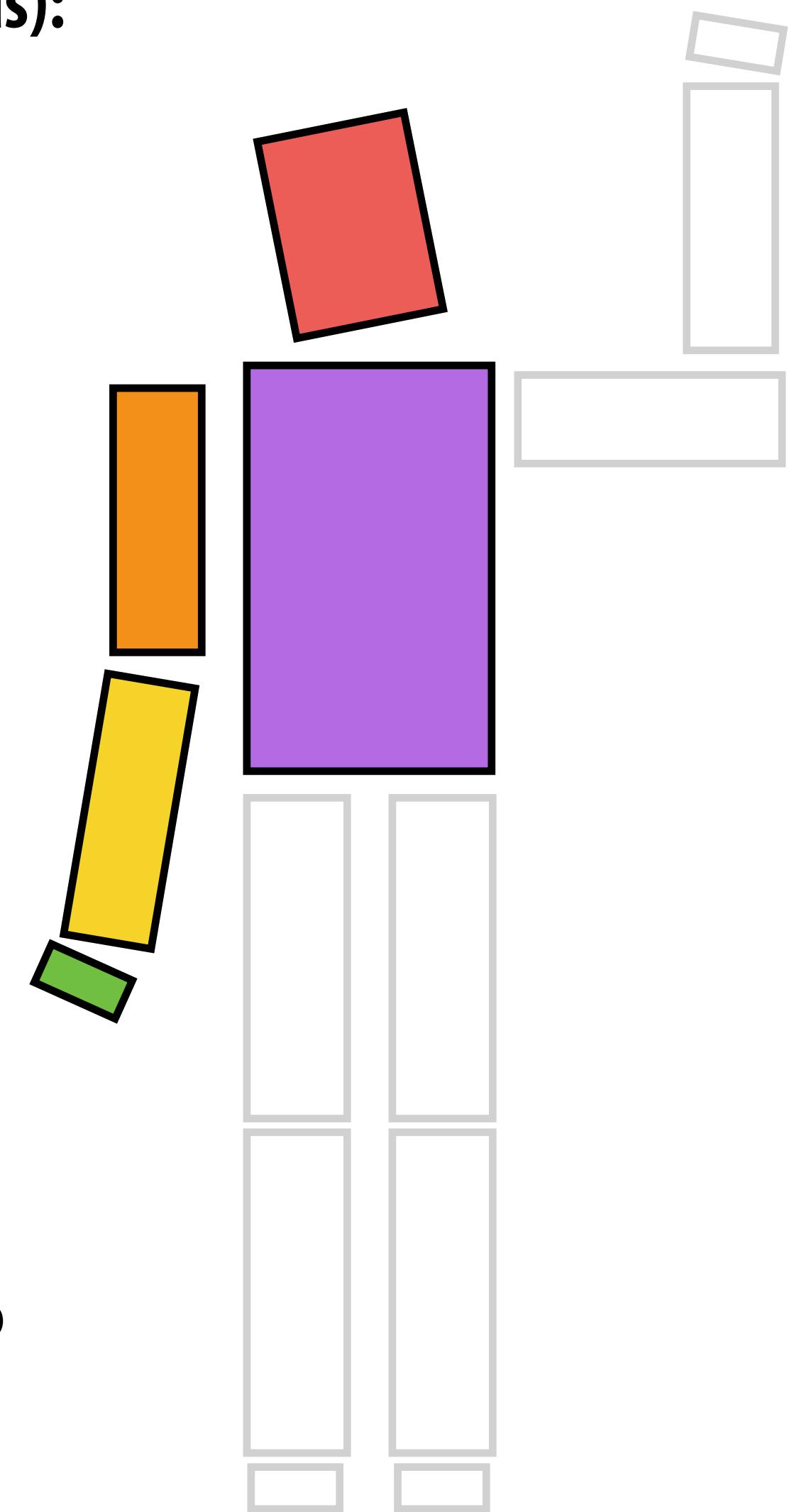
`popTransform();`

`....`

right hand

right lower arm group

right arm group



Skeleton - hierarchical representation

Transform used for drawing:

$T_{(0,10)}$

$T_{(0,10)} T_{(0,5)}$

$T_{(0,10)} T_{(0,5)} R_{\text{head}}$

$T_{(0,10)}$

$T_{(0,10)} T_{(-2,3)}$

$T_{(0,10)} T_{(-2,3)} R_{\text{sho}}$

$T_{(0,10)} T_{(-2,3)} R_{\text{sho}} T_{(0,-3)}$

$T_{(0,10)} T_{(-2,3)} R_{\text{sho}} T_{(0,-3)} R_{\text{elb}}$

$T_{(0,10)} T_{(-2,3)} R_{\text{sho}} T_{(0,-3)} R_{\text{elb}} T_{(0,-3)}$

$T_{(0,10)} T_{(-2,3)} R_{\text{sho}} T_{(0,-3)} R_{\text{elb}} T_{(0,-3)} R_{\text{wr}}$

Sequence of commands (transformation and drawing commands):

`translate(0, 10); // person centered at (0,10)`

`drawTorso();`

`pushTransform(); // push a copy of transform onto stack`

`translate(0, 5); // right-multiply onto current transform`

`rotate(headRotation); // right-multiply onto current transform`

`drawHead();`

`popTransform(); // pop current transform off stack`

`pushTransform();`

`translate(-2, 3);`

`rotate(rightShoulderRotation);`

`drawUpperArm();`

`pushTransform();`

`translate(0, -3);`

`rotate(elbowRotation);`

`drawLowerArm();`

`pushTransform();`

`translate(0, -3);`

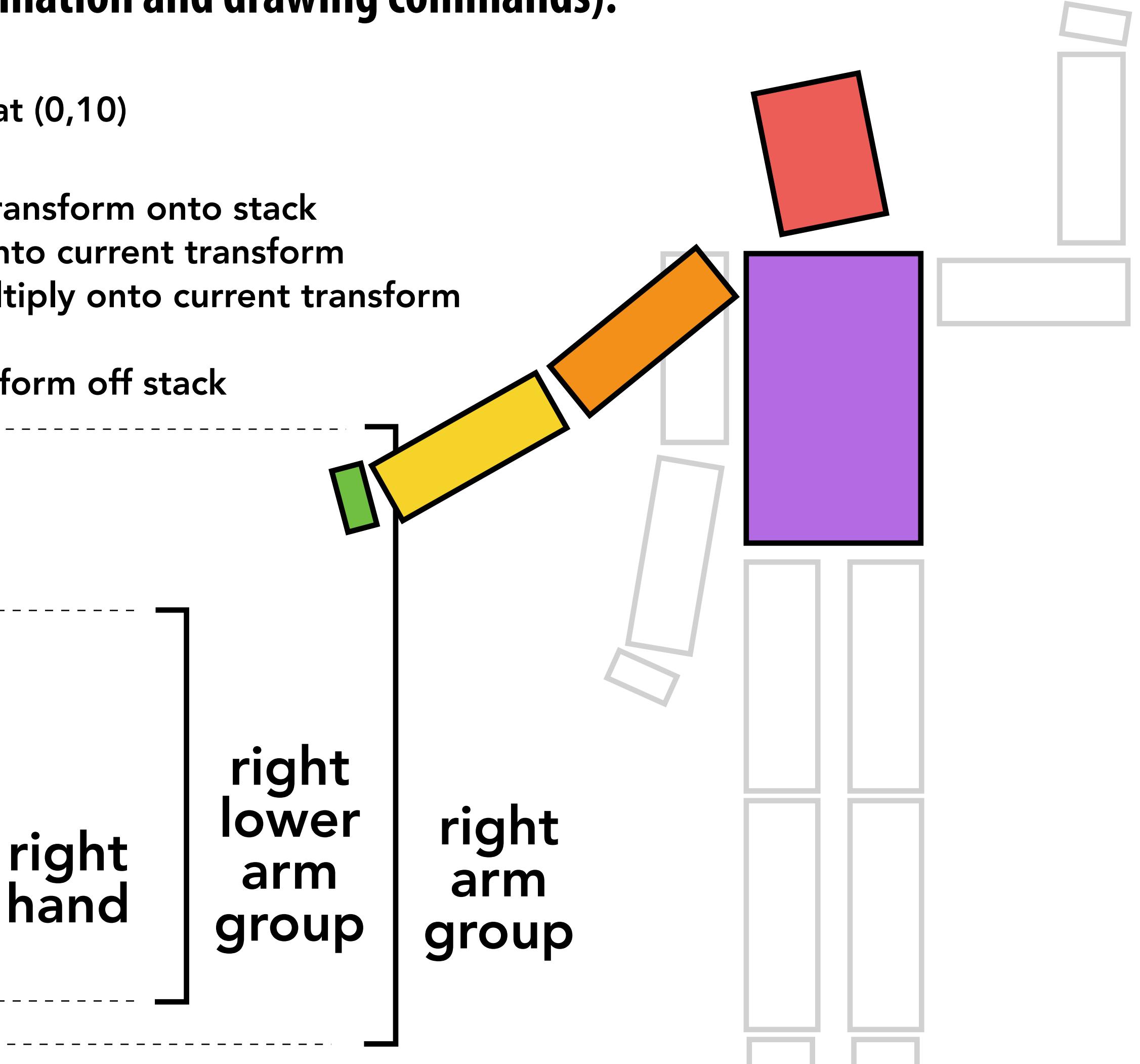
`rotate(wristRotation);`

`drawHand();`

`popTransform();`

`popTransform();`

`....`

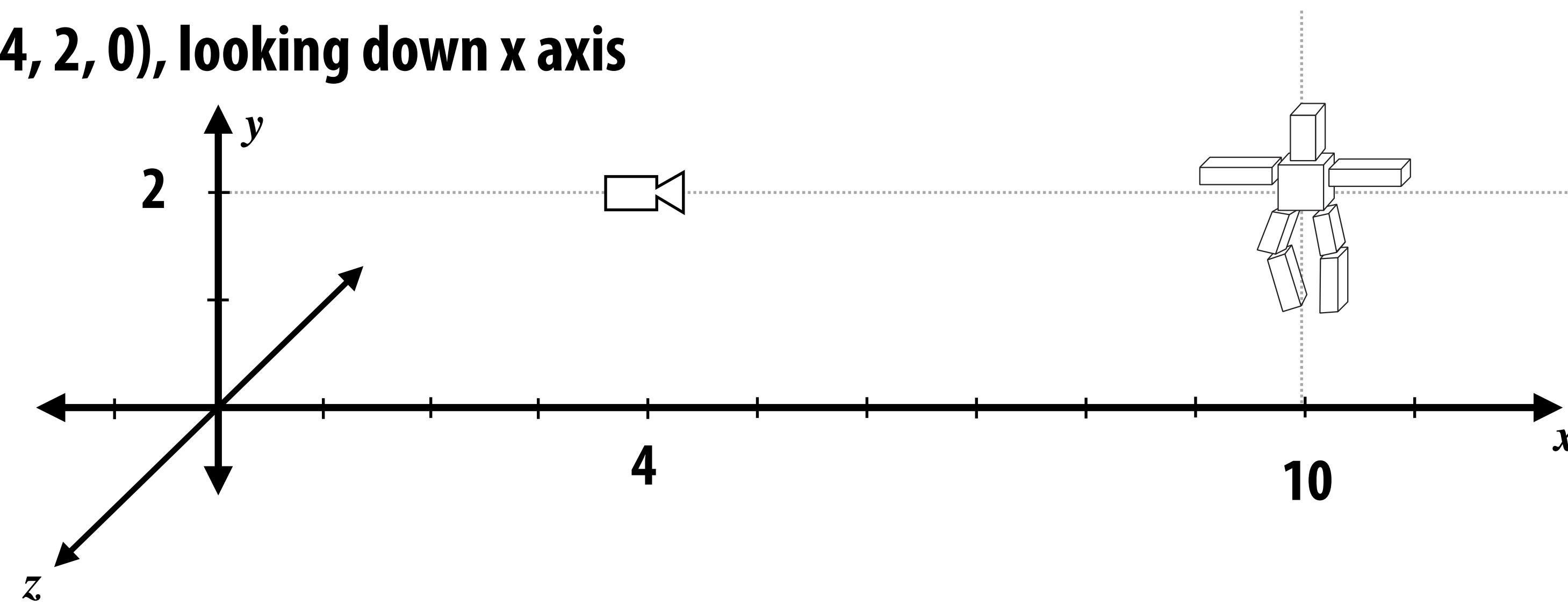


Transforming points into camera-relative coordinates

Example: simple camera transform

Consider object positioned in world at $(10, 2, 0)$

Consider camera at $(4, 2, 0)$, looking down x axis



What transform places the object in a coordinate space where the camera is at the origin and the camera is looking directly down the $-z$ axis?

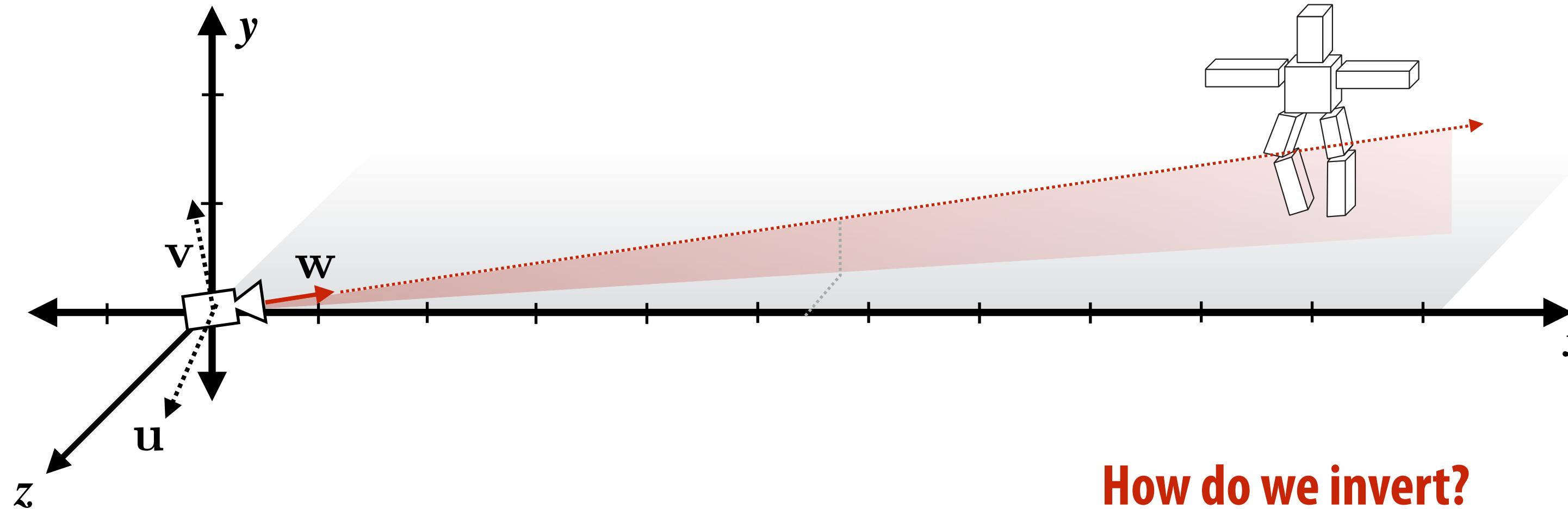
- Translating object vertex positions by $(-4, -2, 0)$ yields position relative to camera
- Rotation about y by $\pi/2$ gives position of object in new coordinate system where camera's view direction is aligned with the $-z$ axis *

* The convenience of such a coordinate system will become clear when we talk about projection!

Camera looking in a different direction

Consider camera at origin looking in direction \mathbf{w}

What transform places in the object in a coordinate space where the camera is at the origin and the camera is looking directly down the $-z$ axis?



Form orthonormal basis around \mathbf{w} : (see \mathbf{u} and \mathbf{v})

Consider orthogonal matrix: \mathbf{R}

$$\mathbf{R} = \begin{bmatrix} \mathbf{u}_x & \mathbf{v}_x & -\mathbf{w}_x \\ \mathbf{u}_y & \mathbf{v}_y & -\mathbf{w}_y \\ \mathbf{u}_z & \mathbf{v}_z & -\mathbf{w}_z \end{bmatrix}$$

\mathbf{R} maps x-axis to \mathbf{u} , y-axis to \mathbf{v} , z axis to $-\mathbf{w}$

How do we invert?

$$\mathbf{R}^{-1} = \mathbf{R}^T = \begin{bmatrix} \mathbf{u}_x & \mathbf{u}_y & \mathbf{u}_z \\ \mathbf{v}_x & \mathbf{v}_y & \mathbf{v}_z \\ -\mathbf{w}_x & -\mathbf{w}_y & -\mathbf{w}_z \end{bmatrix}$$

Why is that the inverse?

$$\mathbf{R}^T \mathbf{u} = [\mathbf{u} \cdot \mathbf{u} \quad \mathbf{v} \cdot \mathbf{u} \quad -\mathbf{w} \cdot \mathbf{u}]^T = [1 \quad 0 \quad 0]^T$$

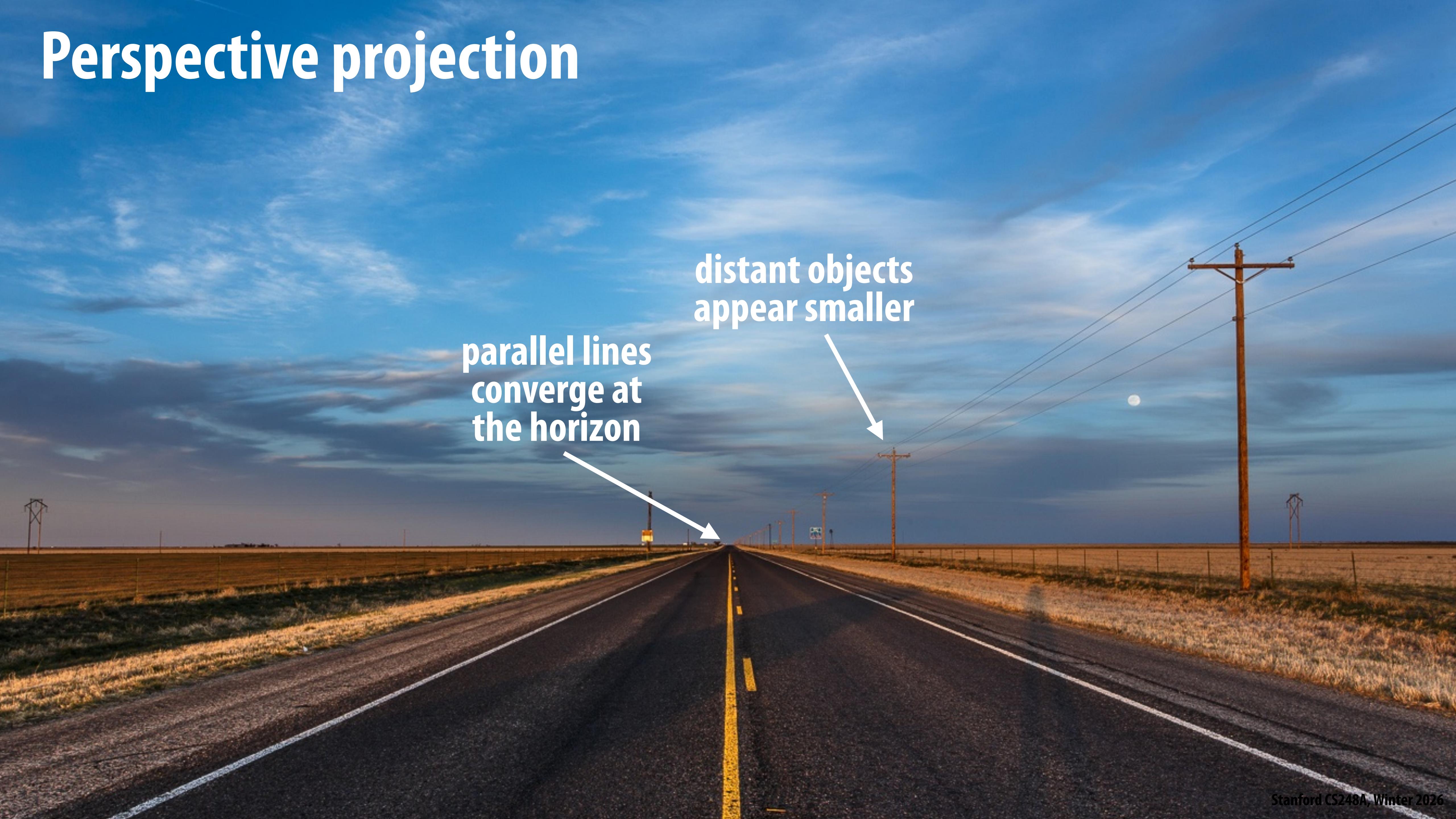
$$\mathbf{R}^T \mathbf{v} = [\mathbf{u} \cdot \mathbf{v} \quad \mathbf{v} \cdot \mathbf{v} \quad -\mathbf{w} \cdot \mathbf{v}]^T = [0 \quad 1 \quad 0]^T$$

$$\mathbf{R}^T \mathbf{w} = [\mathbf{u} \cdot \mathbf{w} \quad \mathbf{v} \cdot \mathbf{w} \quad -\mathbf{w} \cdot \mathbf{w}]^T = [0 \quad 0 \quad -1]^T$$

Self-check exercise (for home)

- Given a camera position P
- And a camera orientation given by orthonormal basis u, v, w (camera looking in w)
- What is a transformation matrix that places the scene in a coordinate space where...
 - The camera is at the origin
 - The camera is looking down -z.

Perspective projection

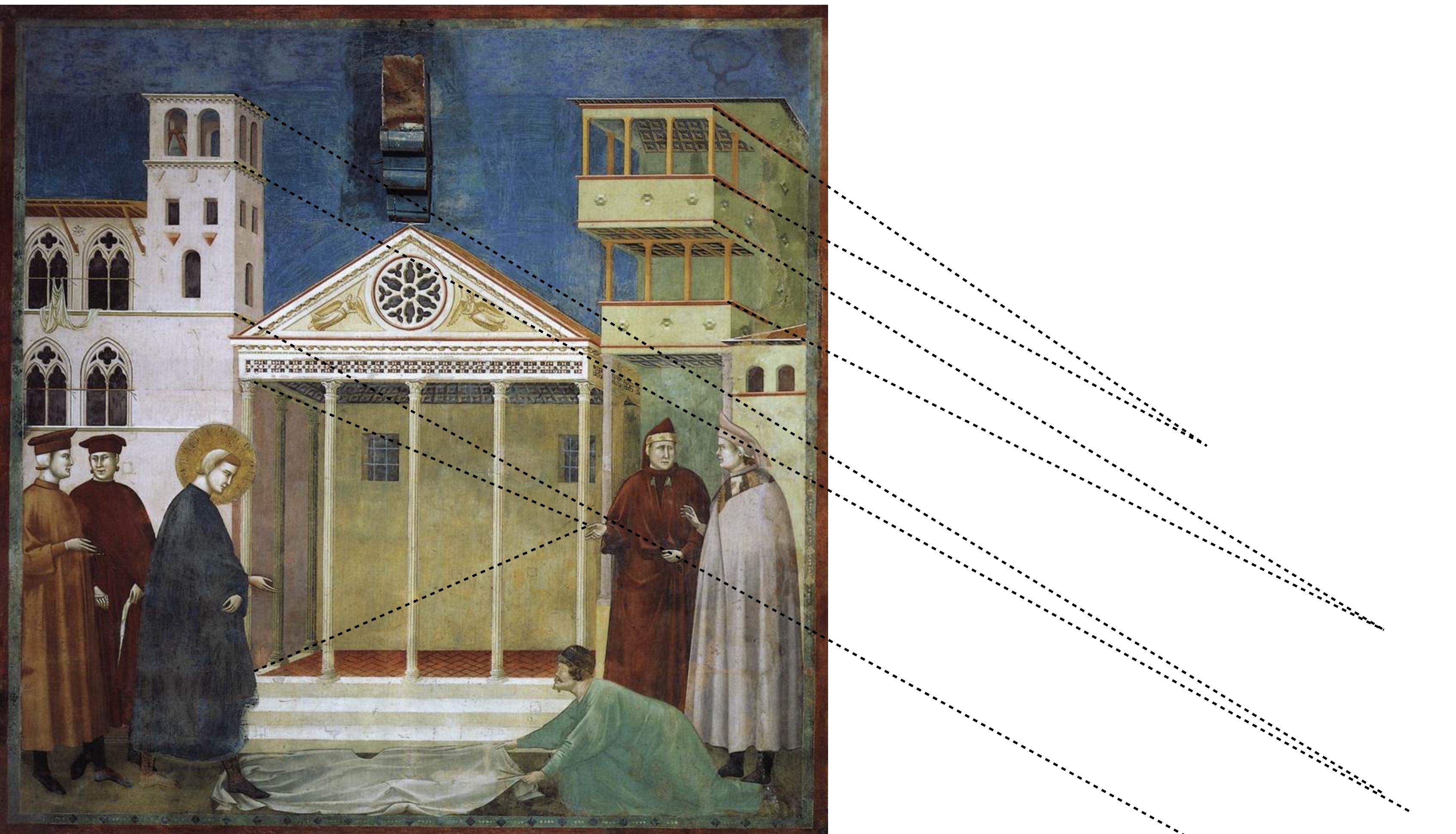


Early painting: incorrect perspective



Carolingian painting from the 8-9th century

Perspective in art

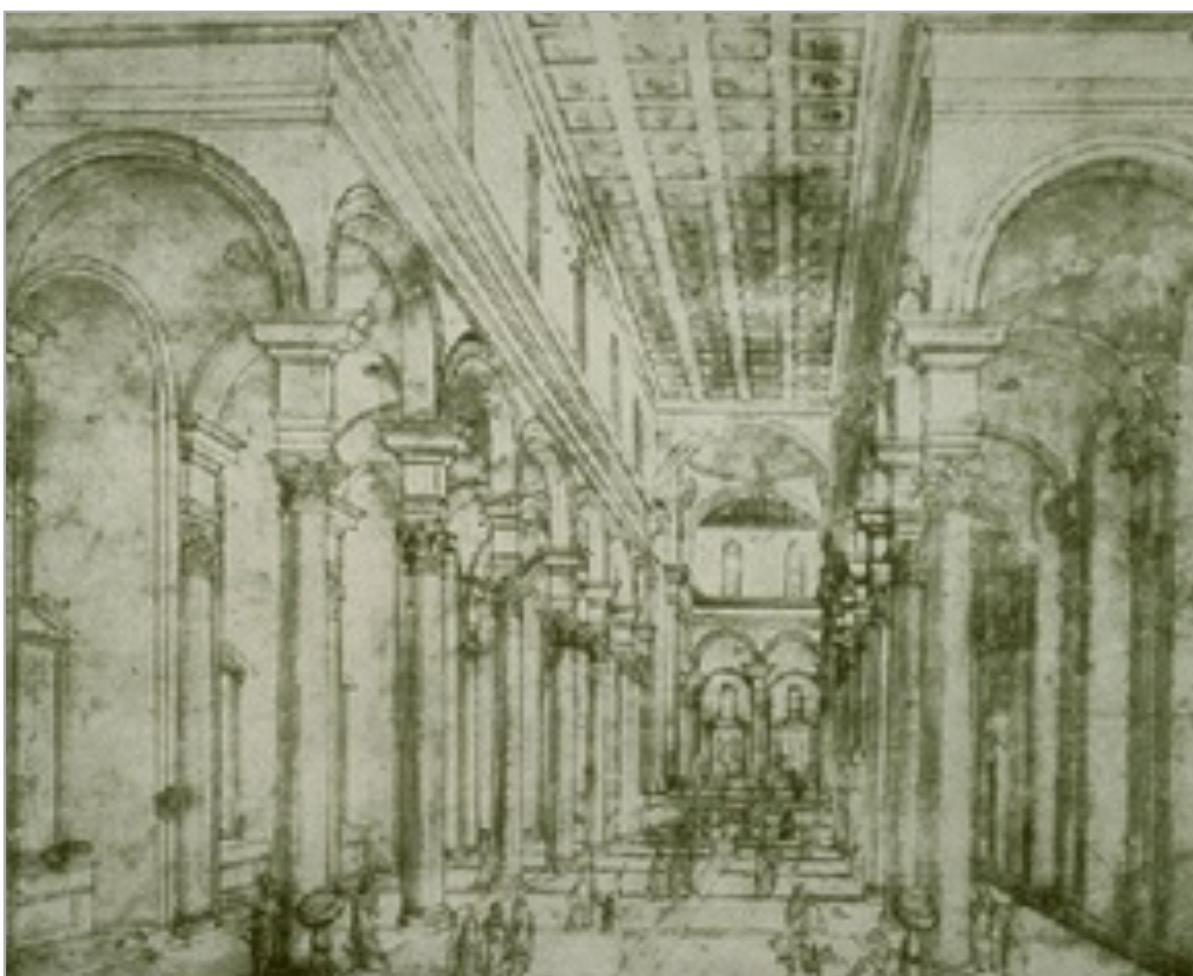


Giotto 1290

Evolution toward correct perspective



Ambrogio Lorenzetti
Annunciation, 1344



Brunelleschi, elevation of Santo Spirito,
1434-83, Florence

First known perspective painting
by Fillipo Brunelleschi



Masaccio – The Tribute Money c.1426-27
Fresco, The Brancacci Chapel, Florence

Perspective in art

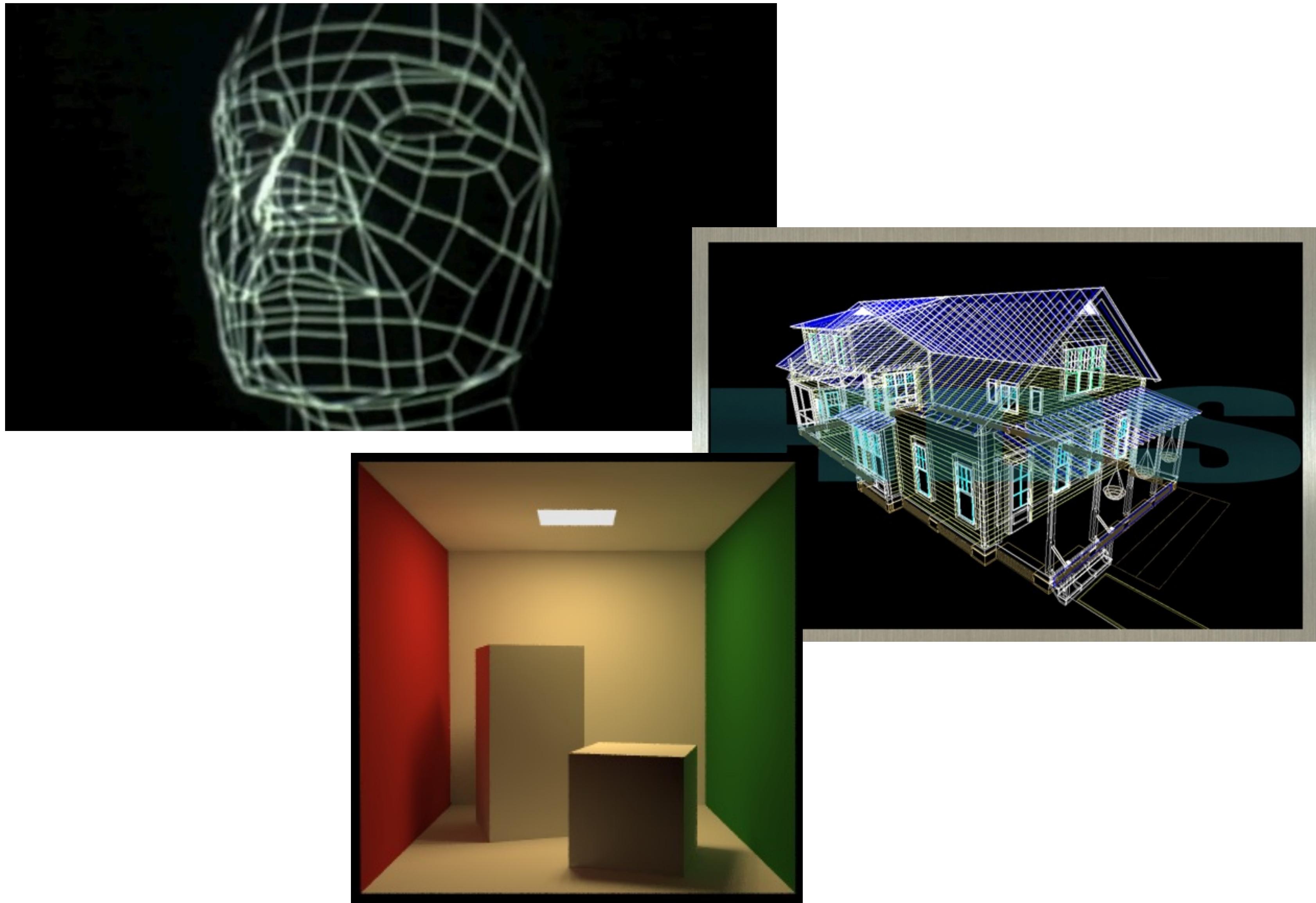


Delivery of the Keys (Sistine Chapel), Perugino, 1482

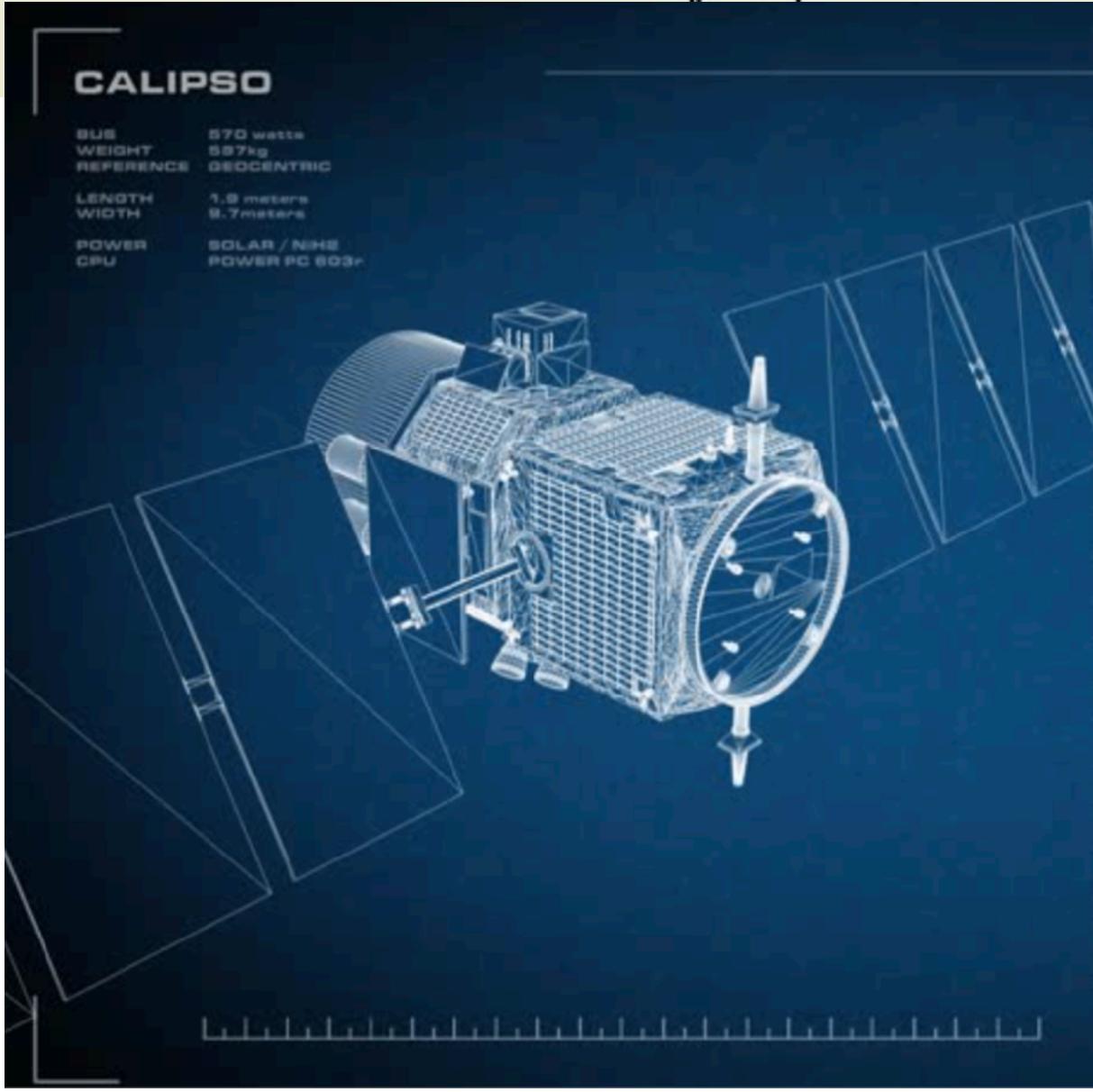
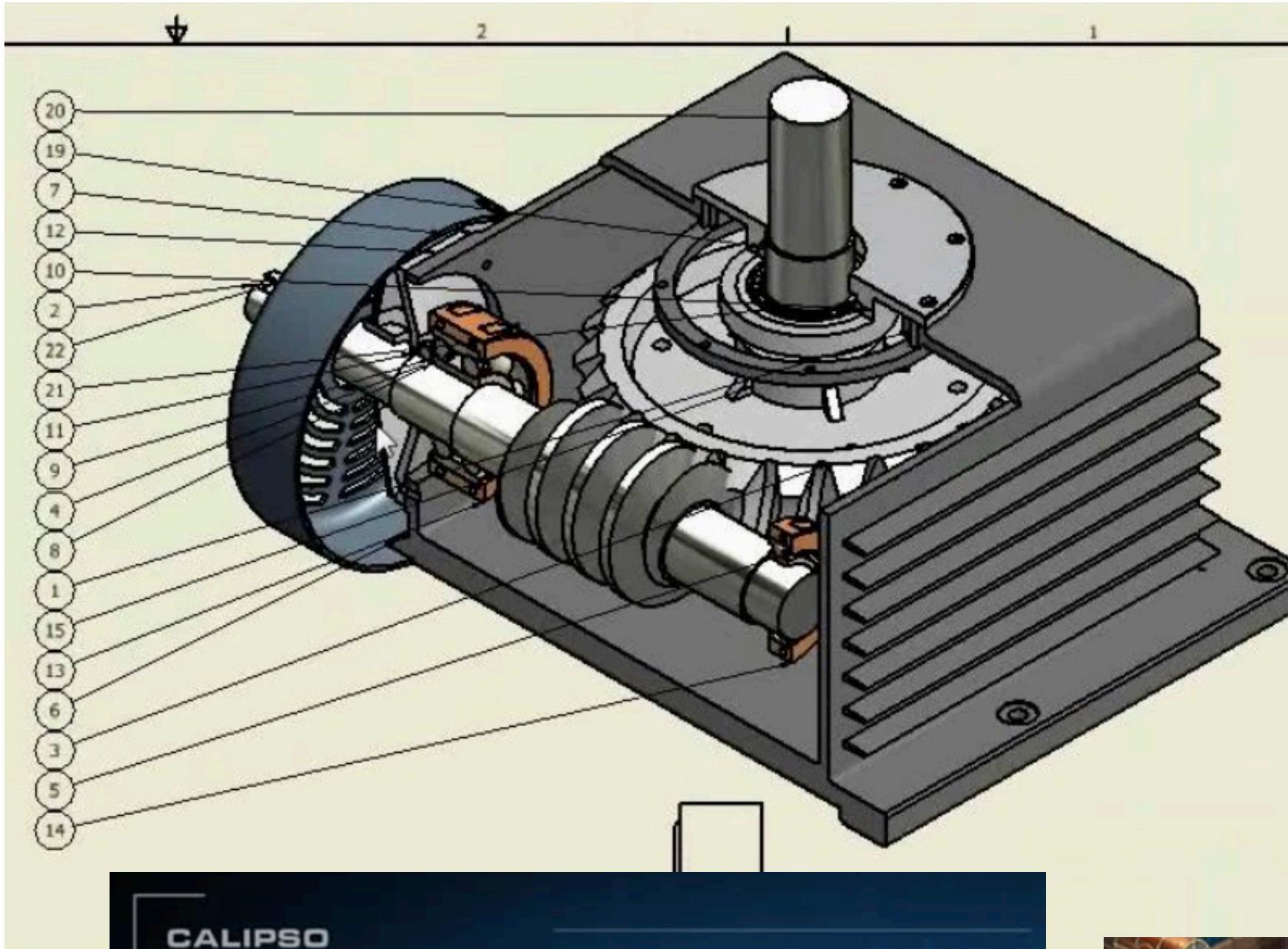
Later... rejection of proper perspective projection



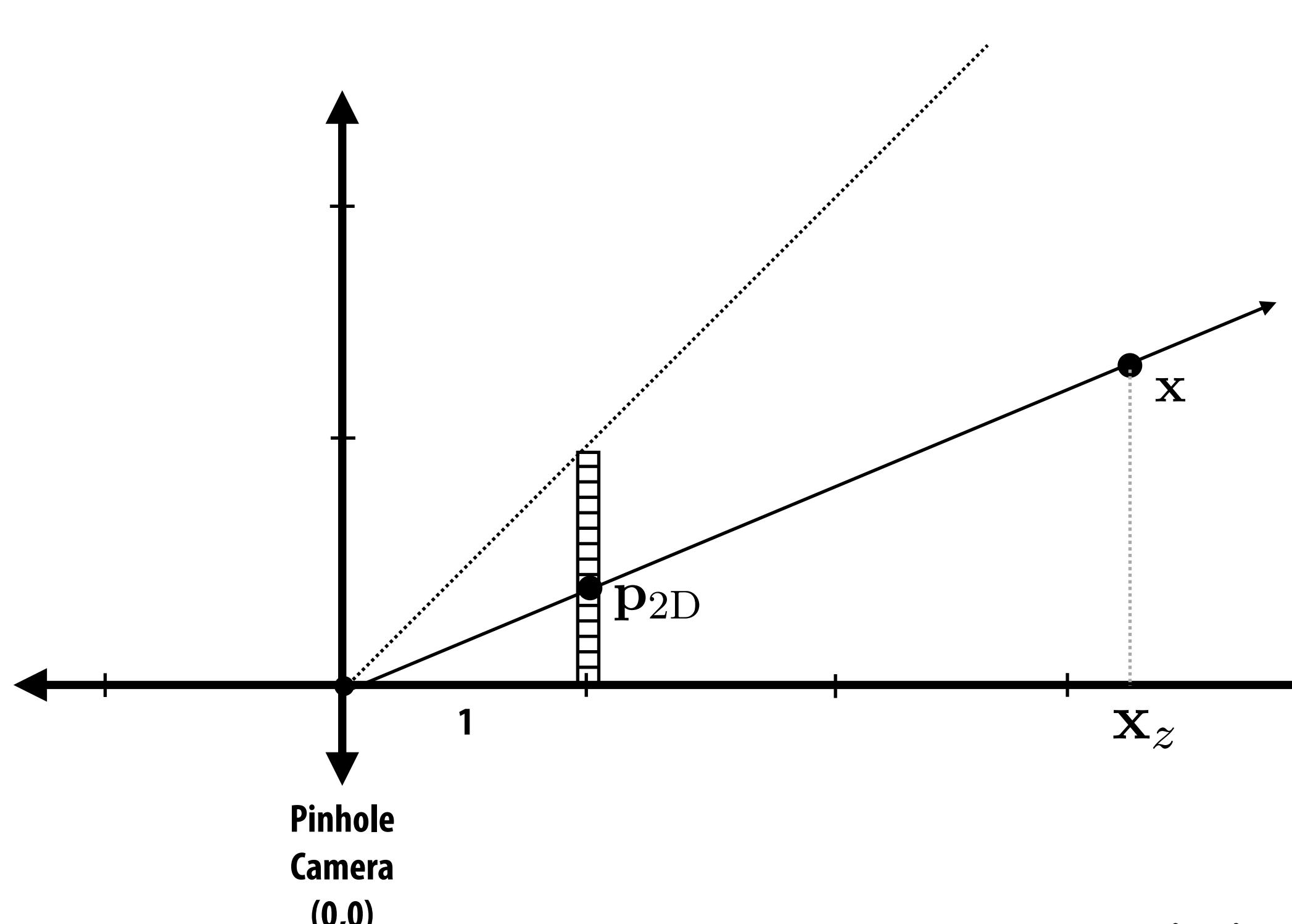
Correct perspective in computer graphics



Rejection of perspective in computer graphics



Basic perspective projection (in homogeneous coords)



Desired perspective projected result (2D point):

$$\mathbf{p}_{2D} = [\mathbf{x}_x/\mathbf{x}_z \quad \mathbf{x}_y/\mathbf{x}_z]^T$$

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Input: point in 3D-H

After applying \mathbf{P} : point in 3D-H

After homogeneous divide:

$$\mathbf{x} = [\mathbf{x}_x \quad \mathbf{x}_y \quad \mathbf{x}_z \quad 1]$$

$$\mathbf{P}\mathbf{x} = [\mathbf{x}_x \quad \mathbf{x}_y \quad \mathbf{x}_z \quad \mathbf{x}_z]^T$$

$$[\mathbf{x}_x/\mathbf{x}_z \quad \mathbf{x}_y/\mathbf{x}_z \quad 1]^T$$

(throw out third component to get 2D)

Assumption:

Pinhole camera at $(0,0)$ looking down z

**A good exercise:
Transforming points into screen-relative (pixel) coordinates**

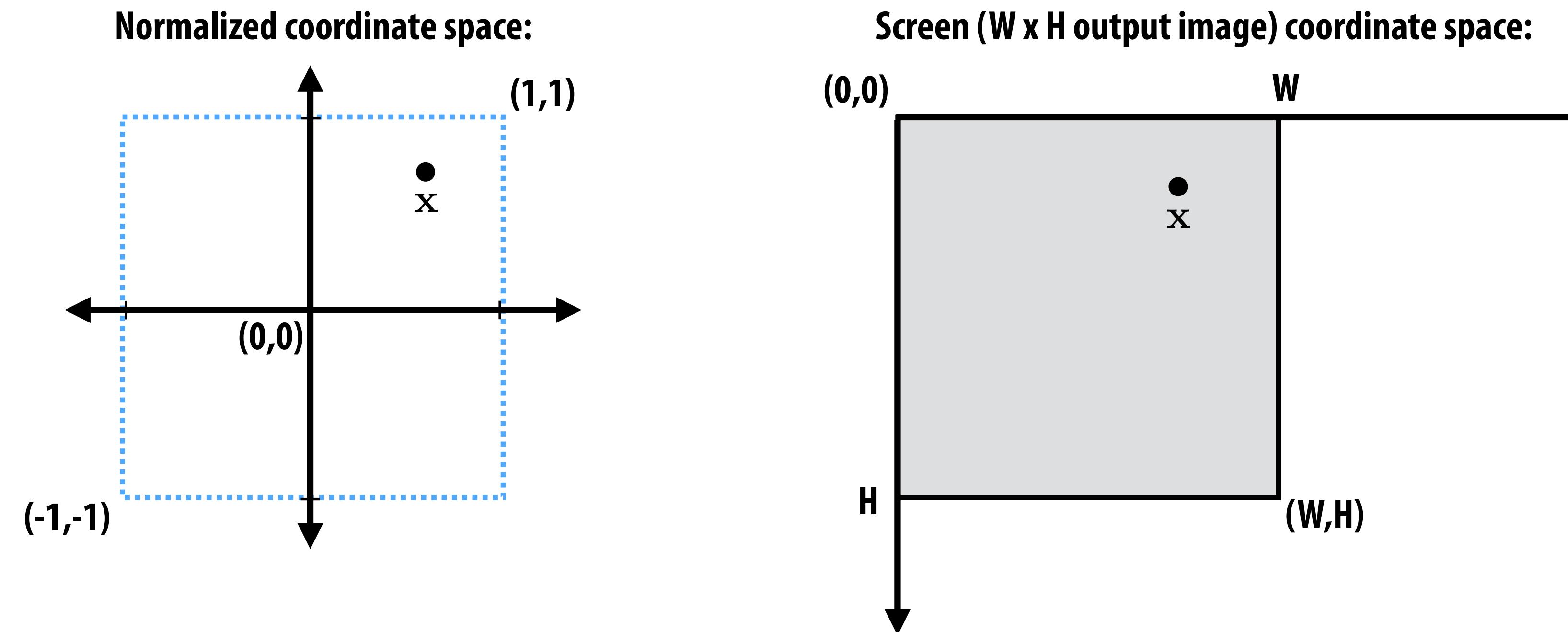
Screen transformation (in 2D) *

Convert 2D points in a normalized coordinate space to screen pixel coordinates

Example: all points within $(-1,1)$ to $(1,1)$ region are visible on screen

$(1,1)$ in normalized space maps to $(W,0)$ in screen space

$(-1,-1)$ in normalized space maps to $(0,H)$ in screen space



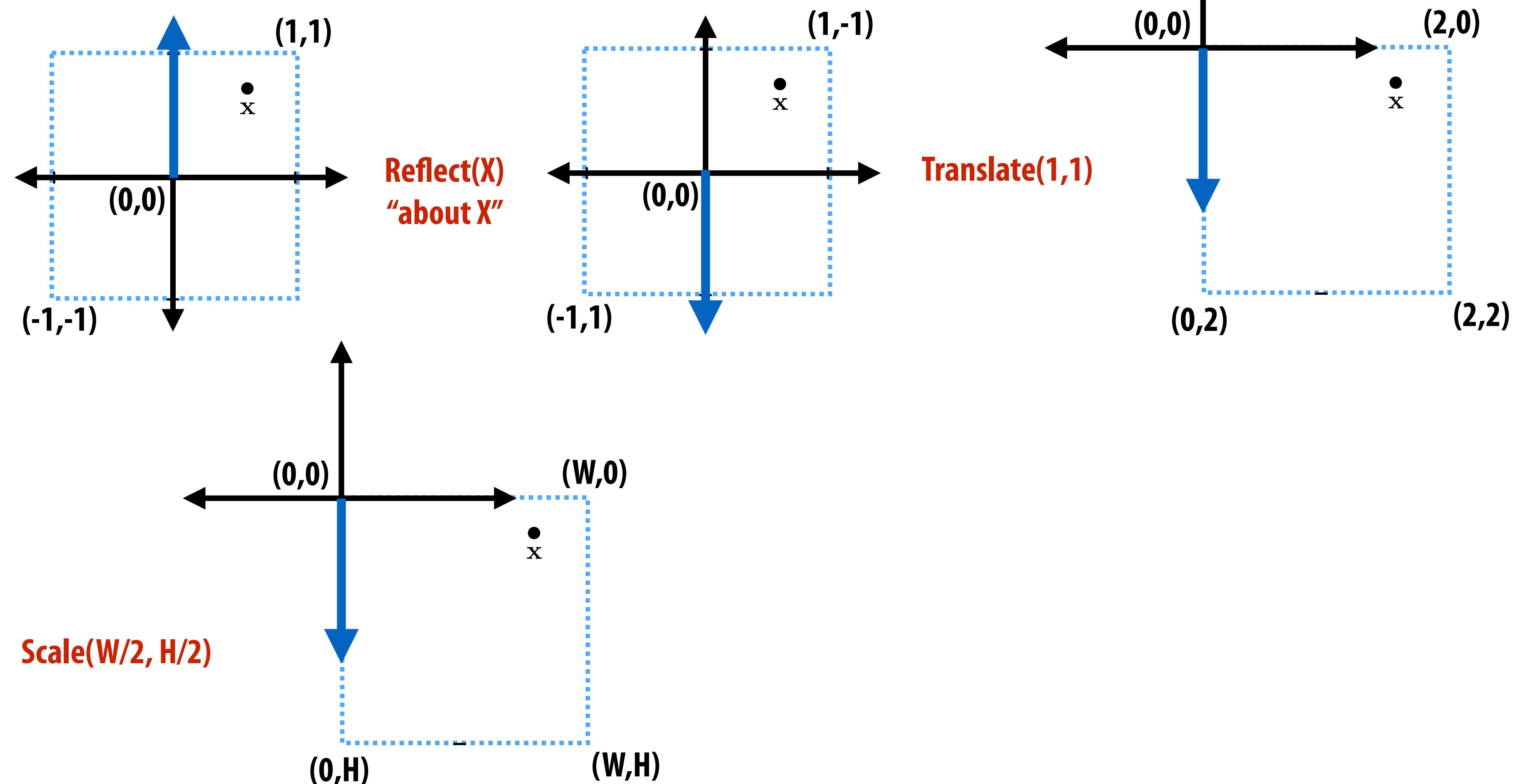
* This slide adopts convention that top-left of screen is $(0,0)$. But many 3D graphics systems place $(0,0)$ in bottom-left. In this case what would the transform be?

Screen transformation

Example: all points within $(-1,1)$ to $(1,1)$ region are on screen

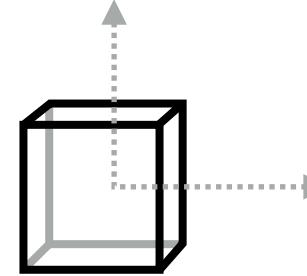
$(1,1)$ in normalized space maps to $(W,0)$ in screen space

$(-1,-1)$ in normalized space maps to $(0,H)$ in screen space



Transformations: from objects in 3D to their 2D screen positions

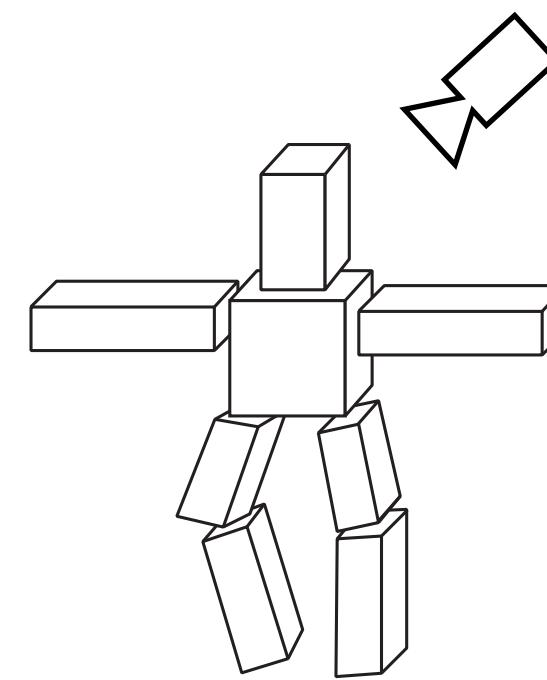
[OBJECT COORDINATES]



description of object geometry
(in a per-object coord frame)

modeling
transform

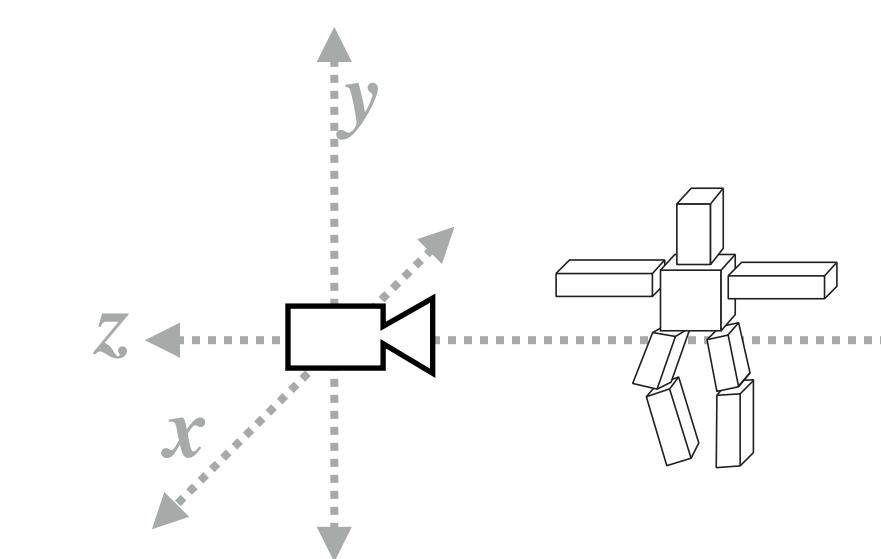
[WORLD COORDINATES]



original description
of objects

view
transform

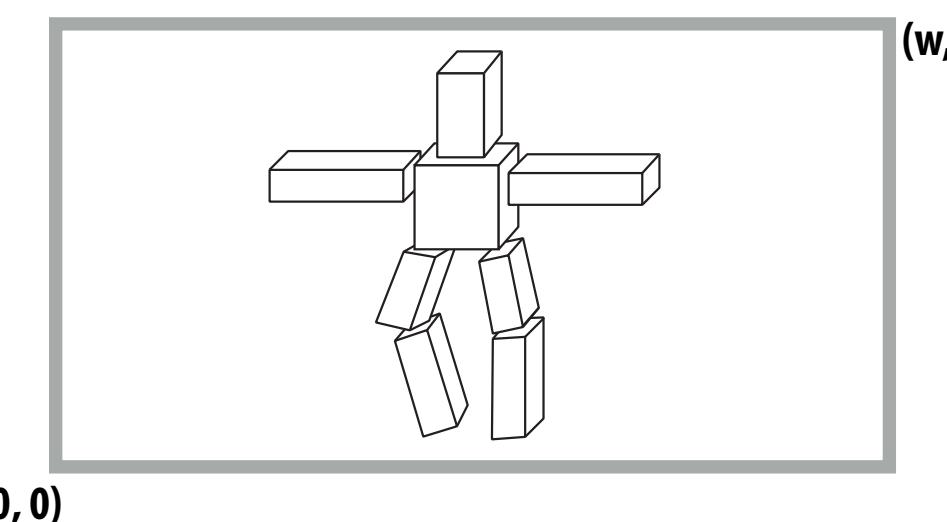
[VIEW COORDINATES] [aka CAMERA COORDINATES]



vertex positions now expressed relative to camera; camera
is sitting at origin looking down -z direction
(Canonical frame of reference allows for use of canonical
projection matrix in rasterization, or camera-position
independent camera ray generation in a ray caster)

screen
transform

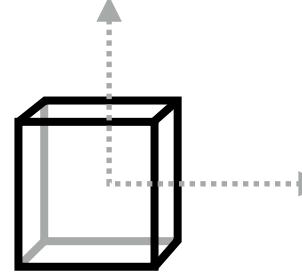
[SCREEN/WINDOW COORDINATES]



Coordinate system corresponds to unit-size pixels
(pixel centers at the half integer locations)

Transformations: from objects in 3D to their 2D screen positions

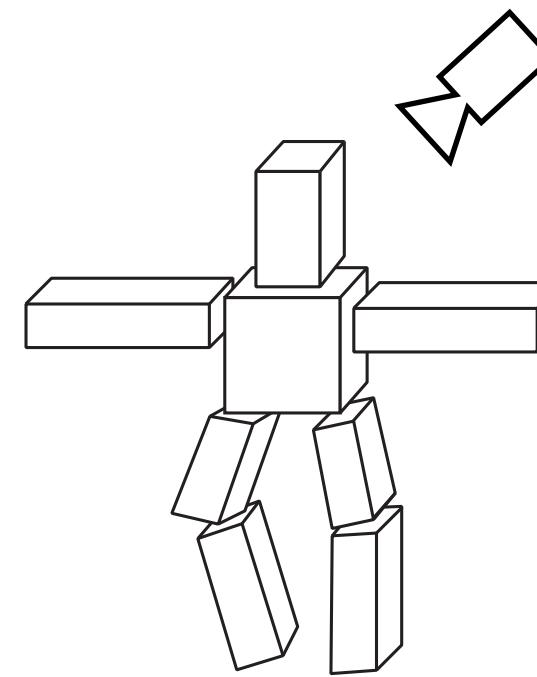
[OBJECT COORDINATES]



modeling
transform

description of object geometry
(in a per-object coord frame)

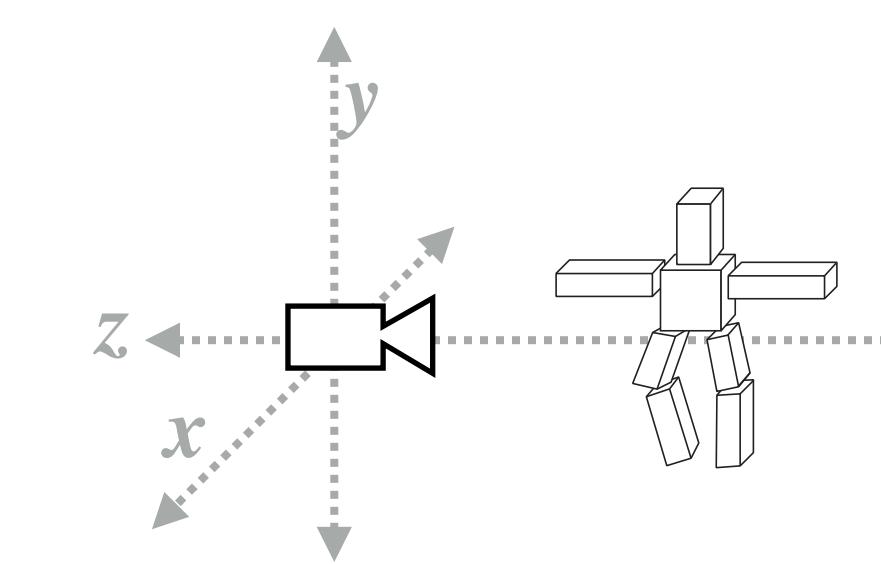
[WORLD COORDINATES]



view
transform

original description
of objects

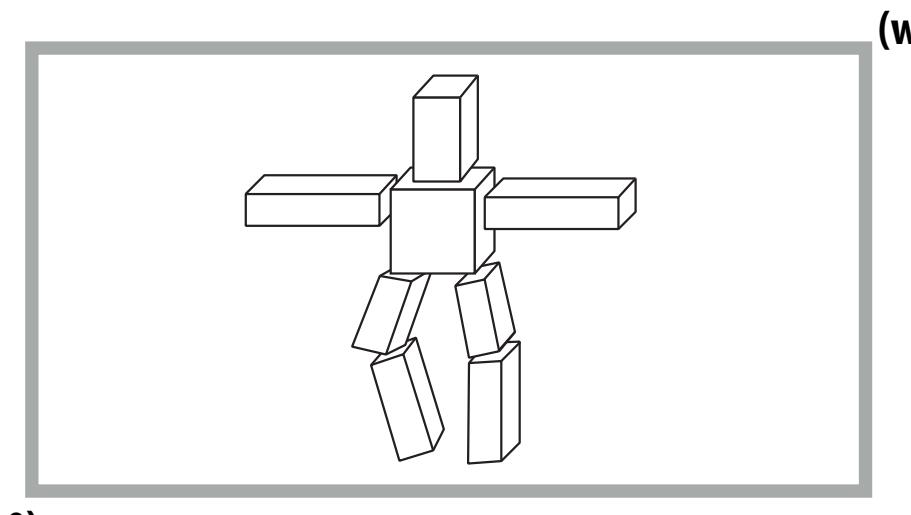
[VIEW COORDINATES] [aka CAMERA COORDINATES]



vertex positions now expressed relative to camera;
camera is sitting at origin looking down -z direction
(Canonical frame of reference allows for use of
canonical projection matrix in rasterization, or
simple camera ray generation in a ray caster)

projection
transform

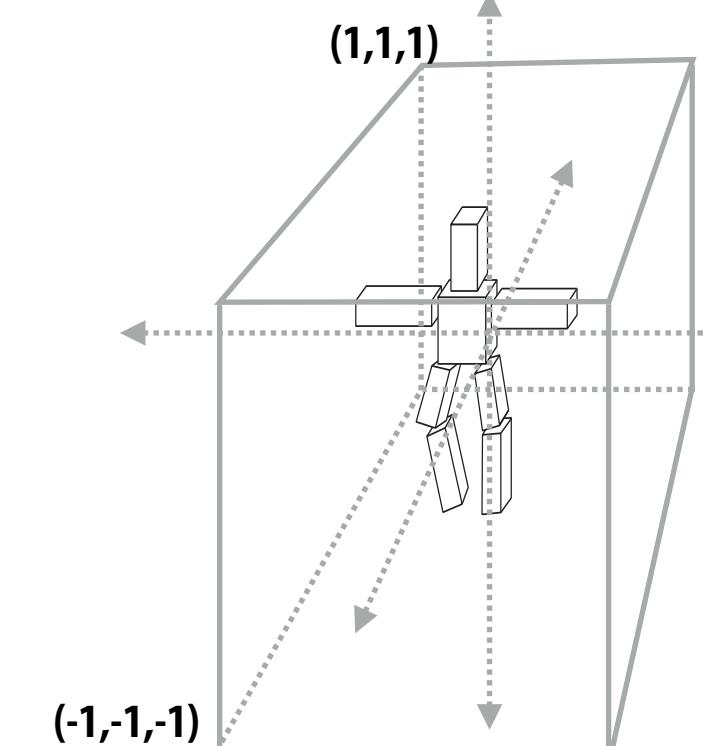
[SCREEN/WINDOW COORDINATES]



screen
transform

Coordinate system corresponds to unit-size pixels
(pixel centers at the half integer locations)

[NORMALIZED DEVICE COORDINATES]



Transformations summary

- **Transformations can be interpreted as operations that move points in space**
 - e.g., for modeling, animation
- **Or as a change of coordinate system**
 - e.g., screen and view transforms
- **Construct complex transformations as compositions of basic transforms**
- **Homogeneous coordinate representation allows for expression of non-linear transforms (e.g., translation, perspective projection) as matrix operations (linear transforms) in higher-dimensional space**
 - **Matrix representation affords simple implementation and efficient composition**

