Perspective Projection and Rasterization

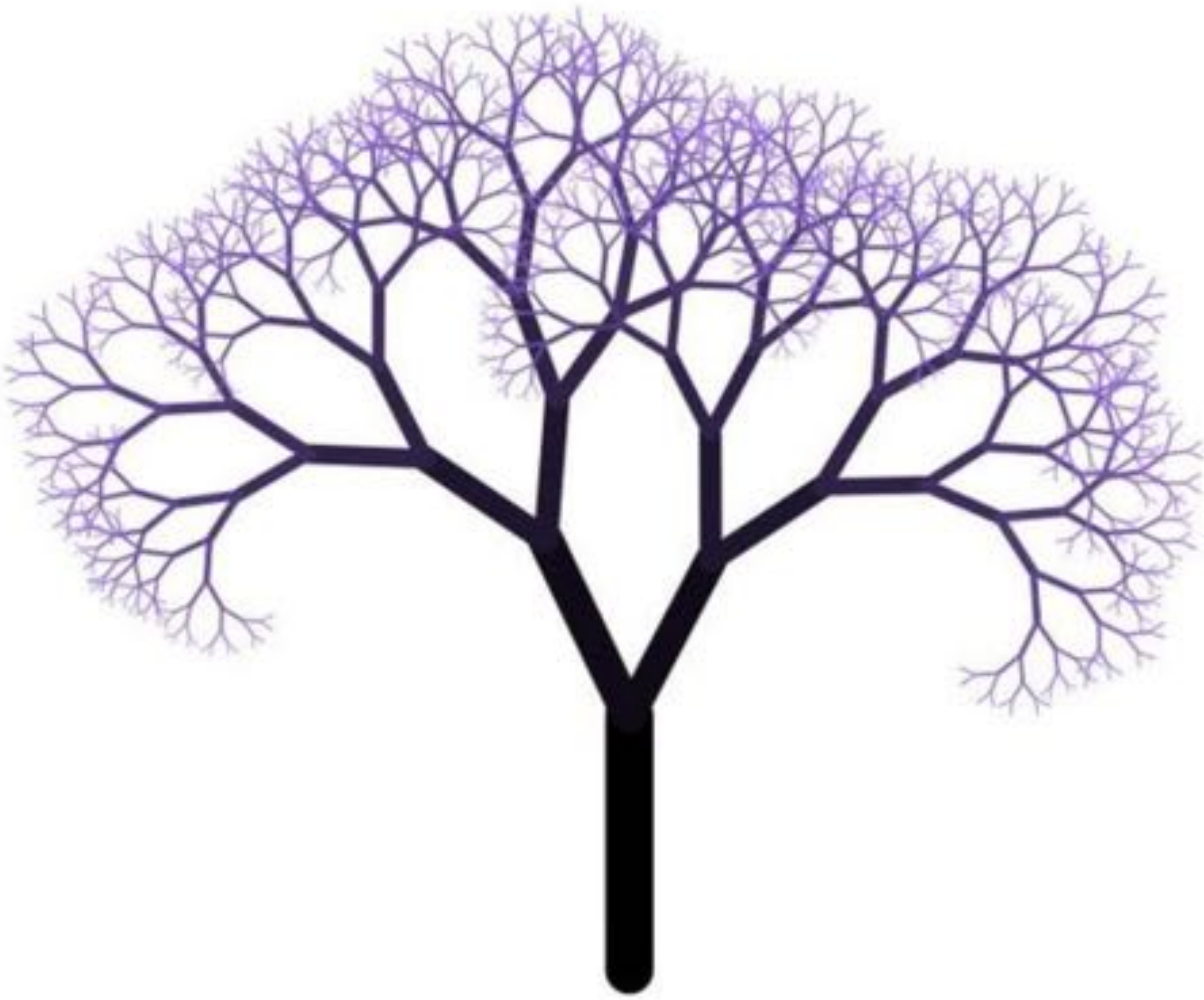
Computer Graphics
CMU 15-462/15-662

Rasterizer A1.0 due Friday Feb 3

- **Checkpoint A1.0 [40pts]:**
  - **A1T1 transforms [5pts]** Last Wednesday's class
  - **A1T2 lines [15pts]** Today
  - **A1T3 flat triangles [15pts]**
  - **A1T4 depth + blending [3pts]** Wednesday's class
  - **writeup-A1.txt [2pts]**

Mini-HW 1 is out — also due Friday Feb 3

Mini HW 1: Trees and Transformations

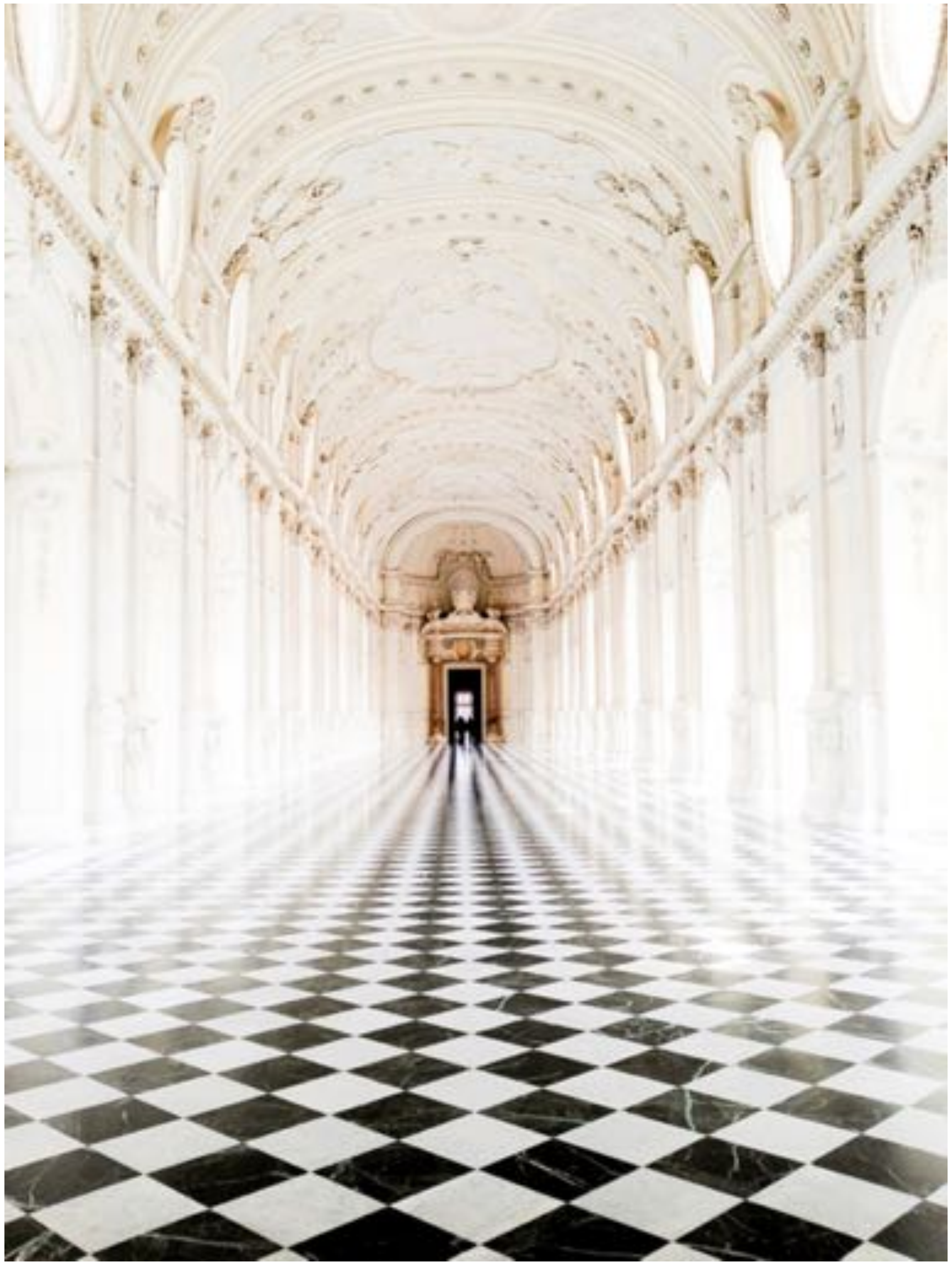


Reminder: you may omit up to 2 Mini HW without penalty

(You may not want to omit this one)

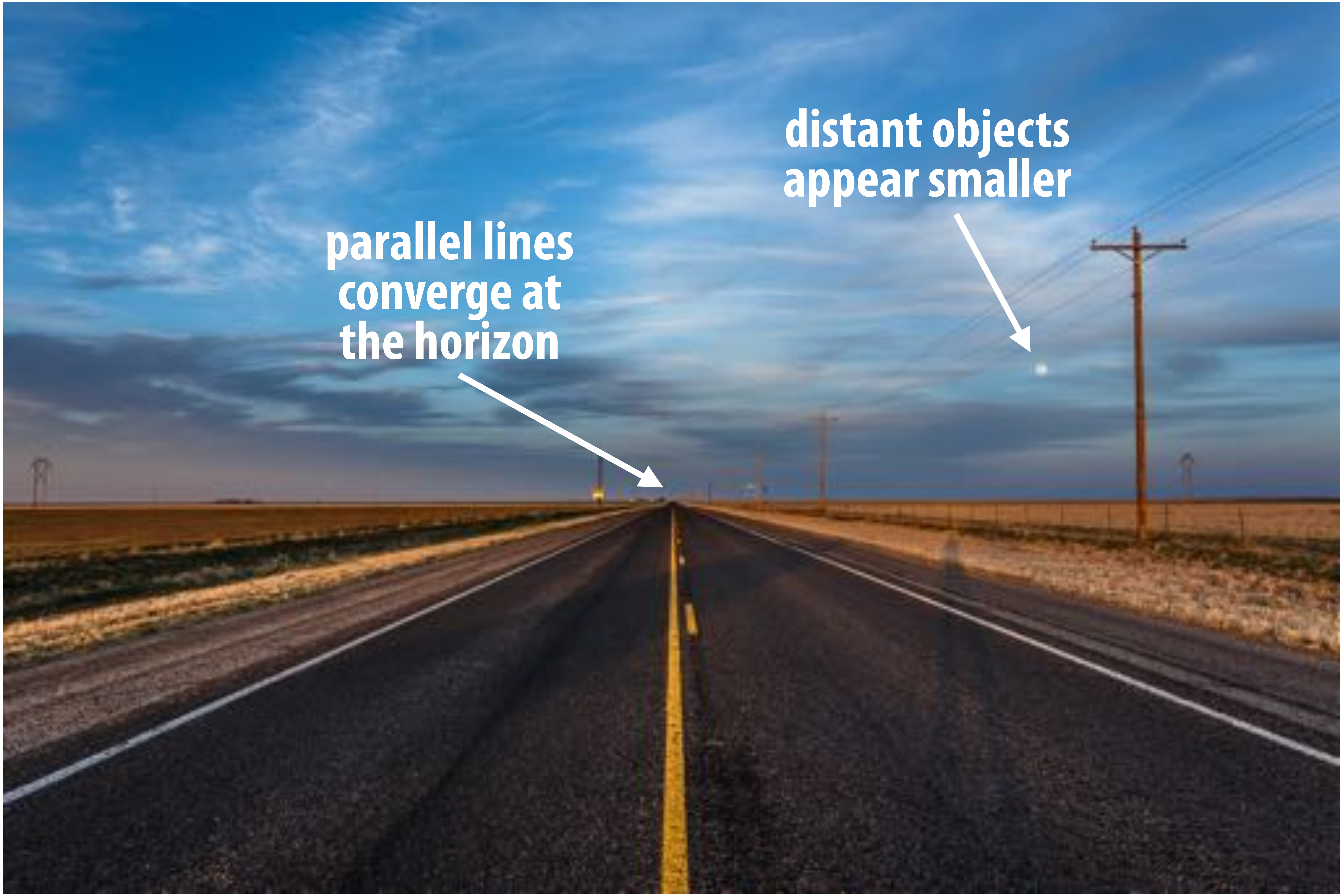
Perspective & Rasterization

- **PREVIOUSLY:**
  - **transformations**
    (how to manipulate primitives in space)
- **TODAY:**
  - **special case of perspective projection**
  - **using our "camera" to turn triangles into pixels on the screen**



Perspective Projection

Perspective projection



Early painting: incorrect perspective

Carolingian painting, 8-9th century

Evolution toward correct perspective


Lorenzetti, c. 1344

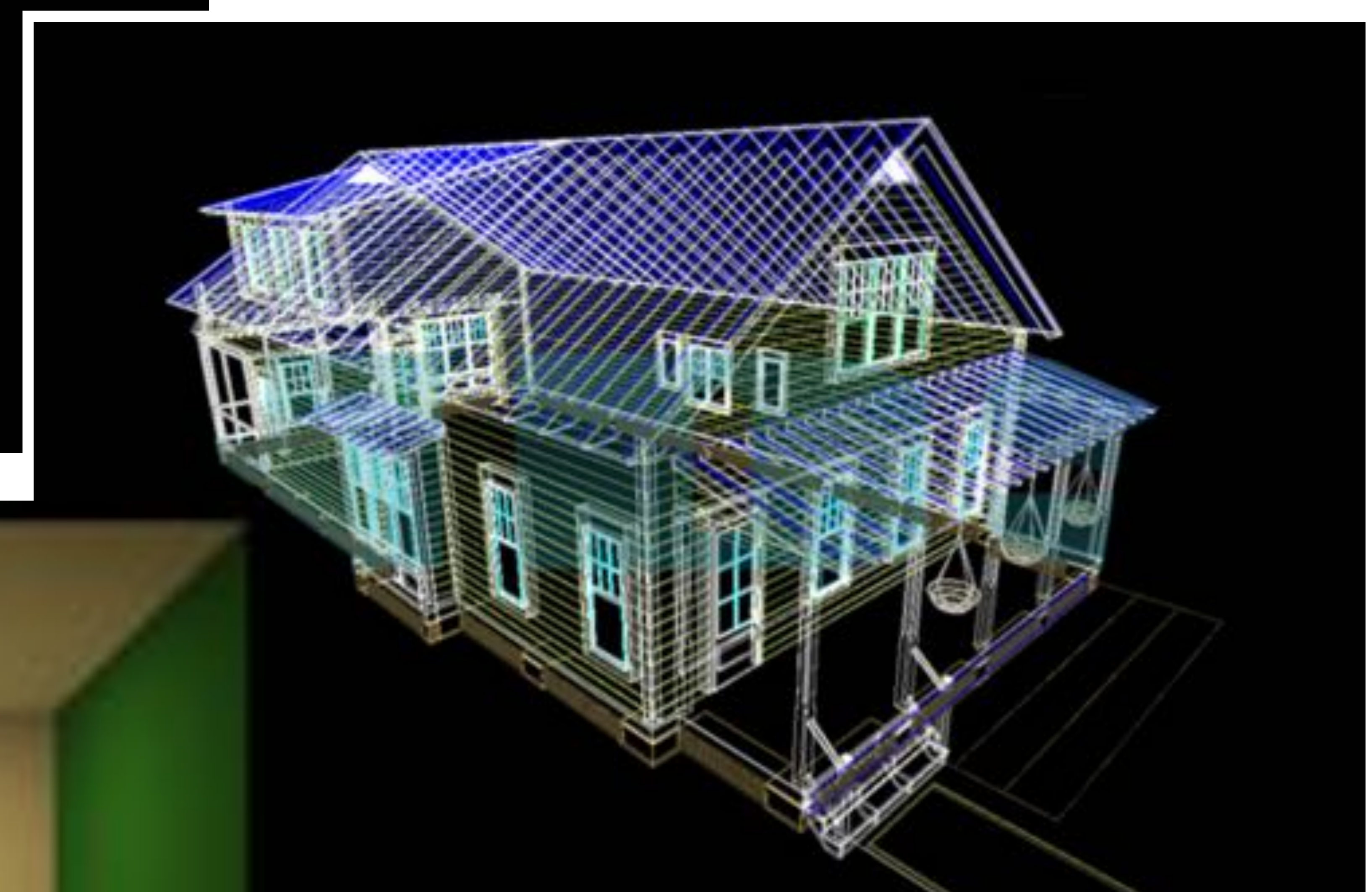

Brunelleschi, c. 1428


Masaccio, c.1427

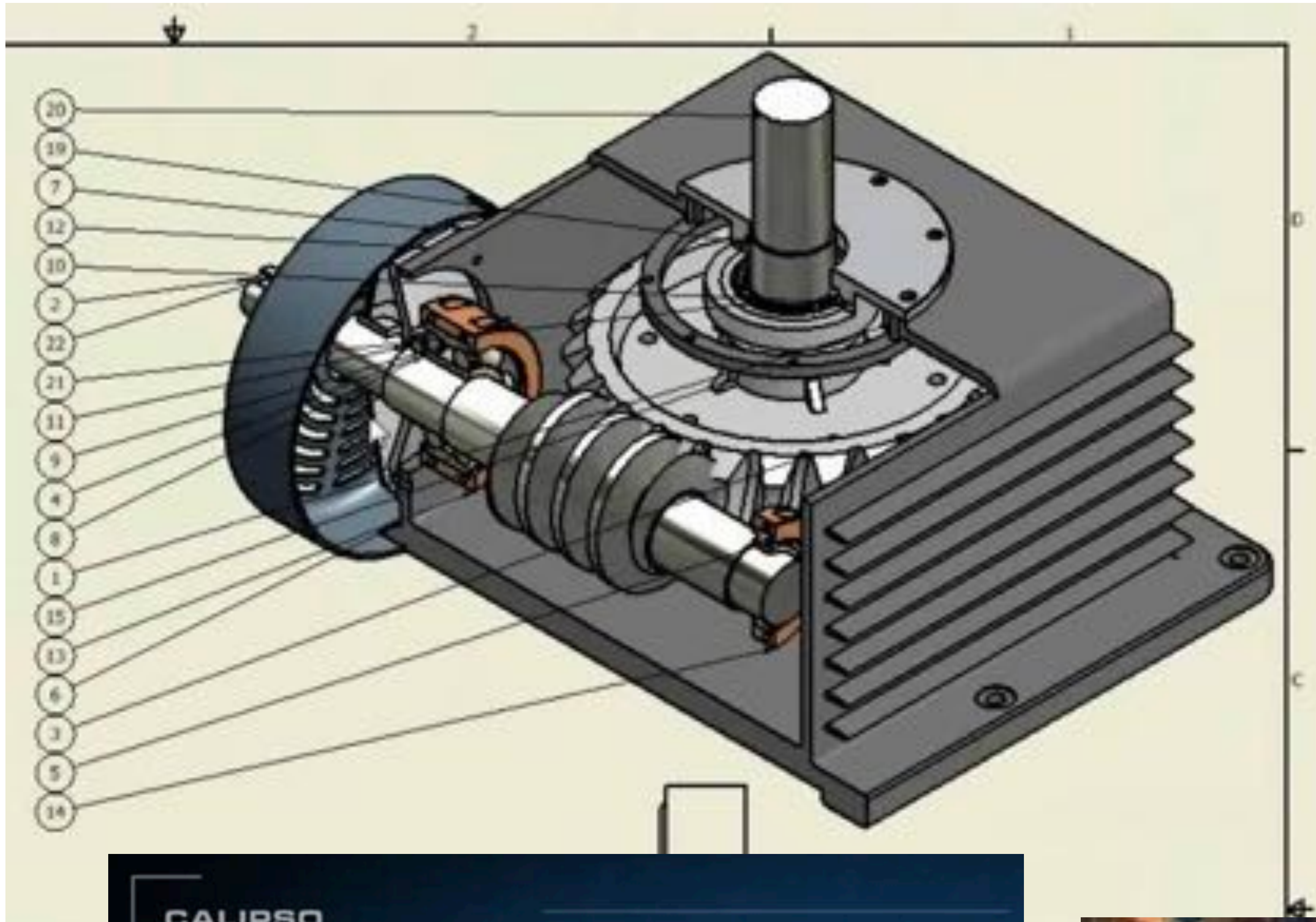
Later… rejection of proper perspective projection

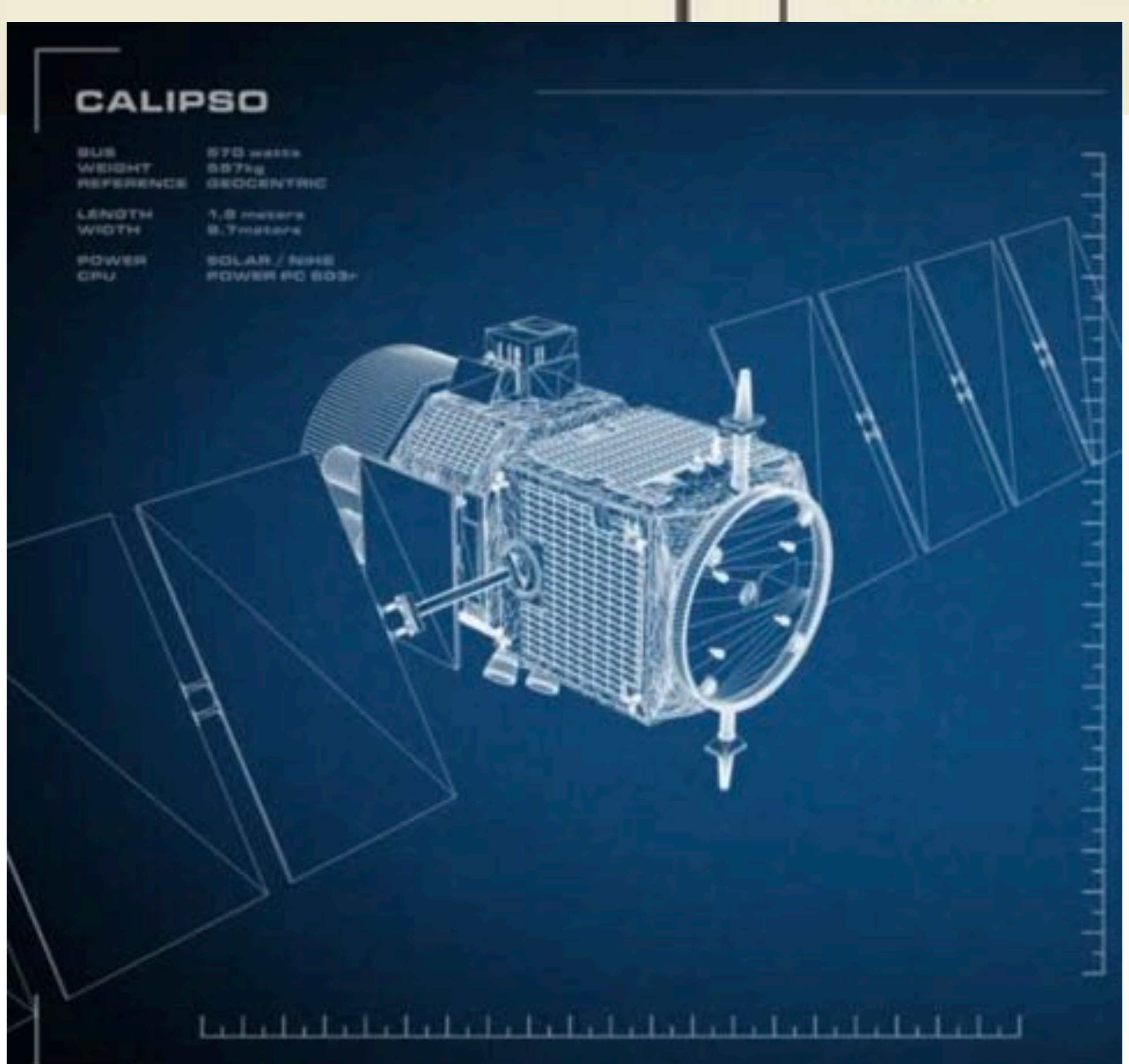
Picasso, 1910

Return of perspective in computer graphics



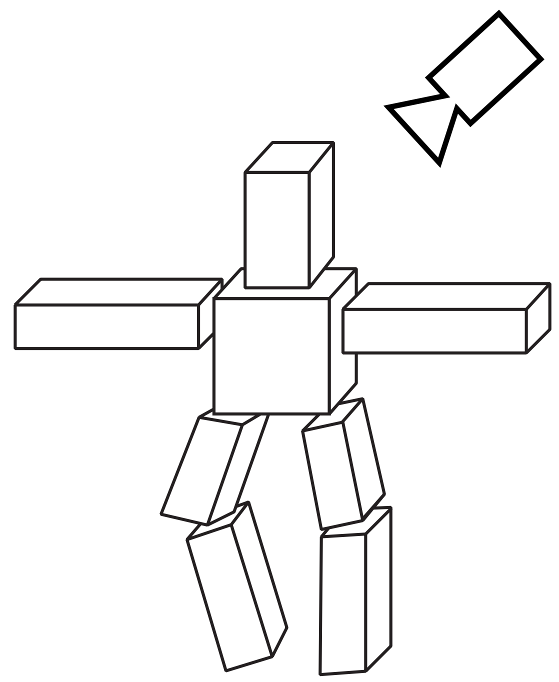
Rejection of perspective in computer graphics





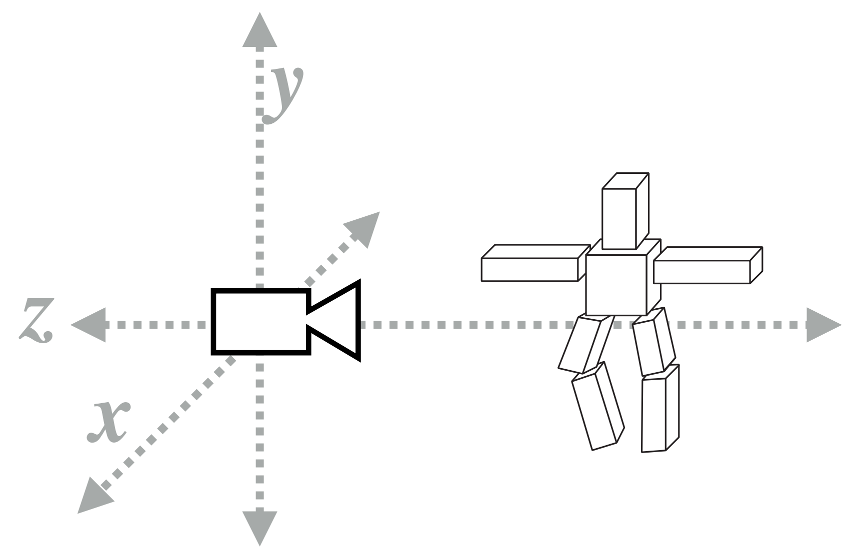
Transformations + Perspective Projection

[WORLD COORDINATES]



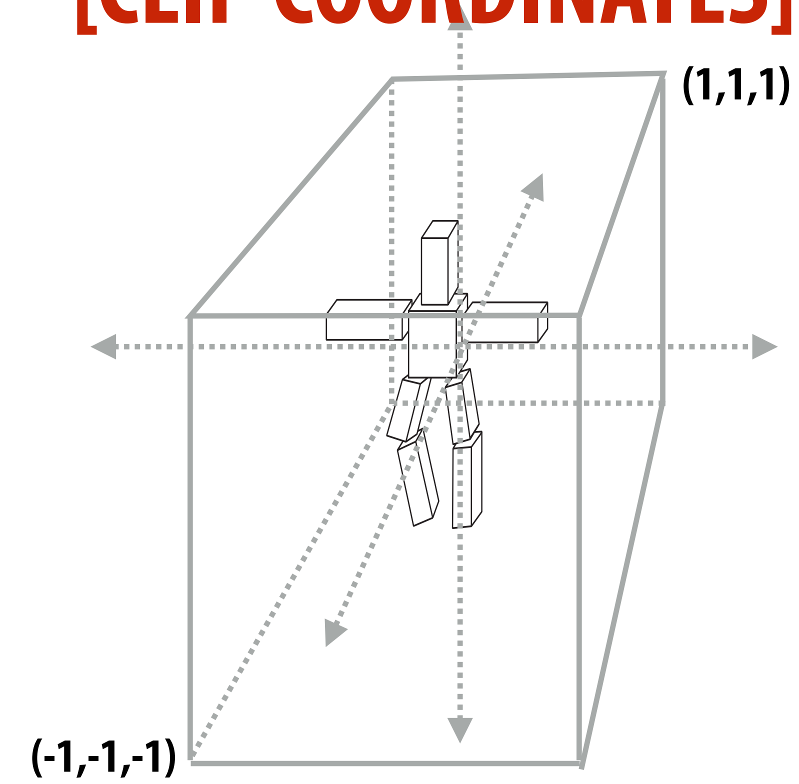
original description of objects

[VIEW COORDINATES]



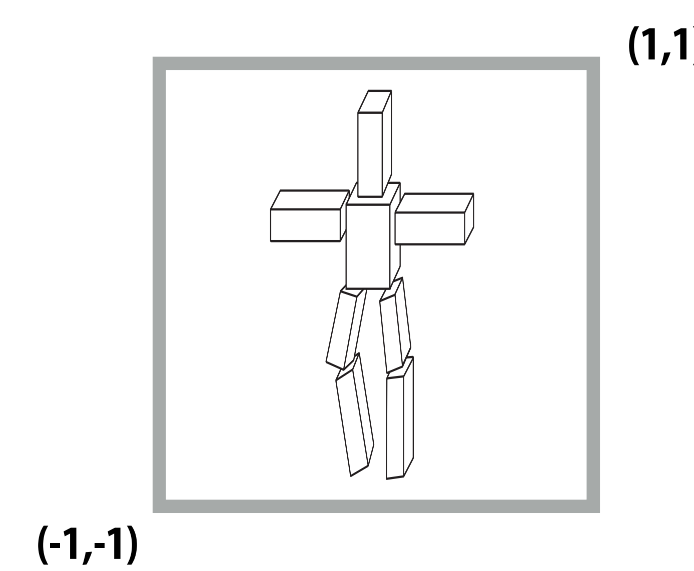
all positions now expressed relative to camera; camera is sitting at origin looking down -z direction

[CLIP COORDINATES]



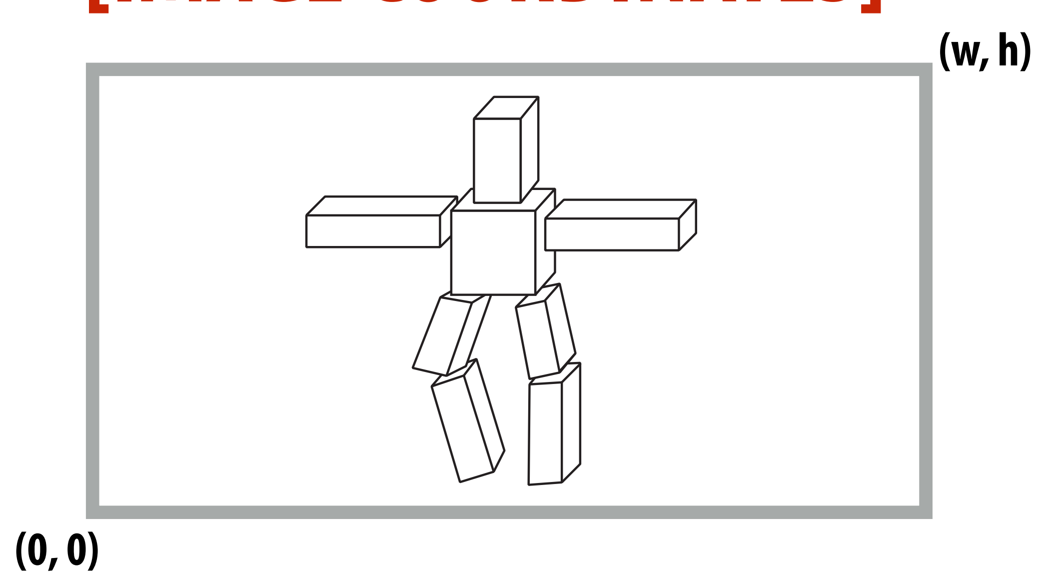
everything visible to the camera is mapped to unit cube for easy “clipping”

[NORMALIZED COORDINATES]



unit cube mapped to unit square via perspective divide

[IMAGE COORDINATES]

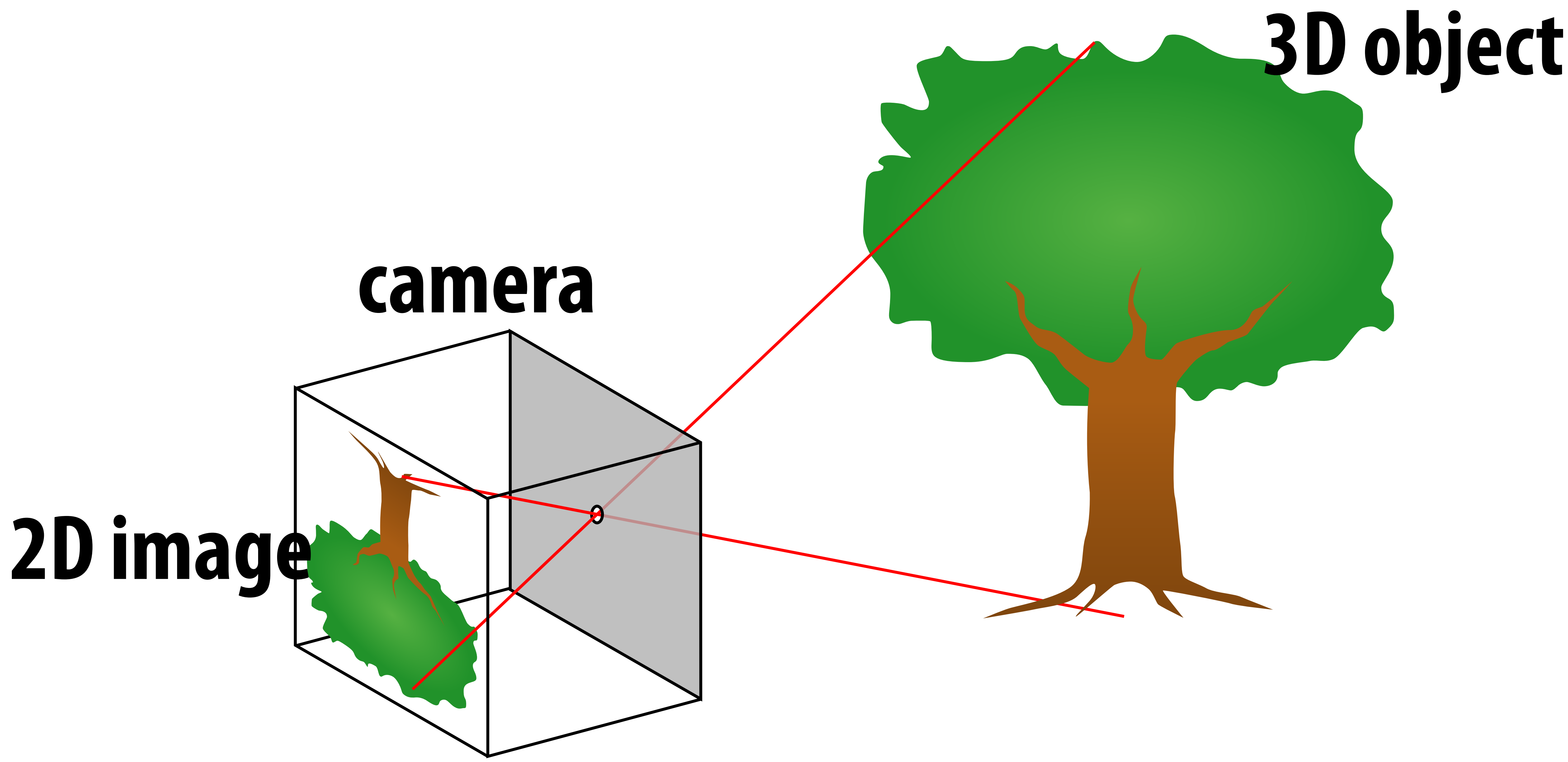


coordinates stretched to match image dimensions (and flipped upside-down)

2D primitives can now be drawn via rasterization

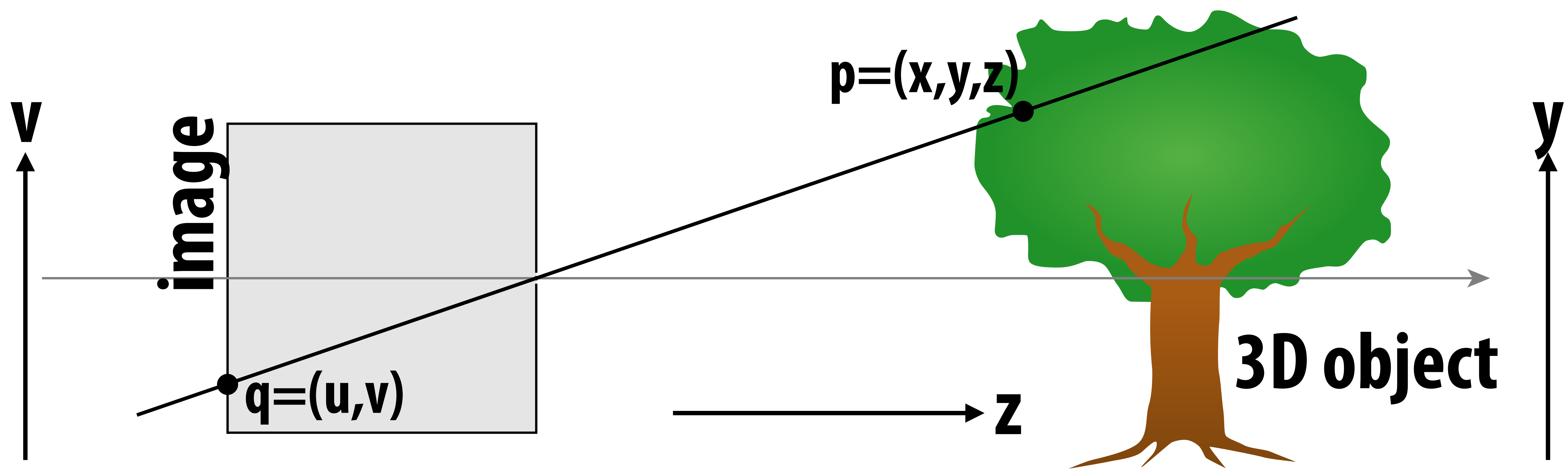
Simple Perspective Projection

- Objects look smaller as they get further away ("perspective")
- Why does this happen?



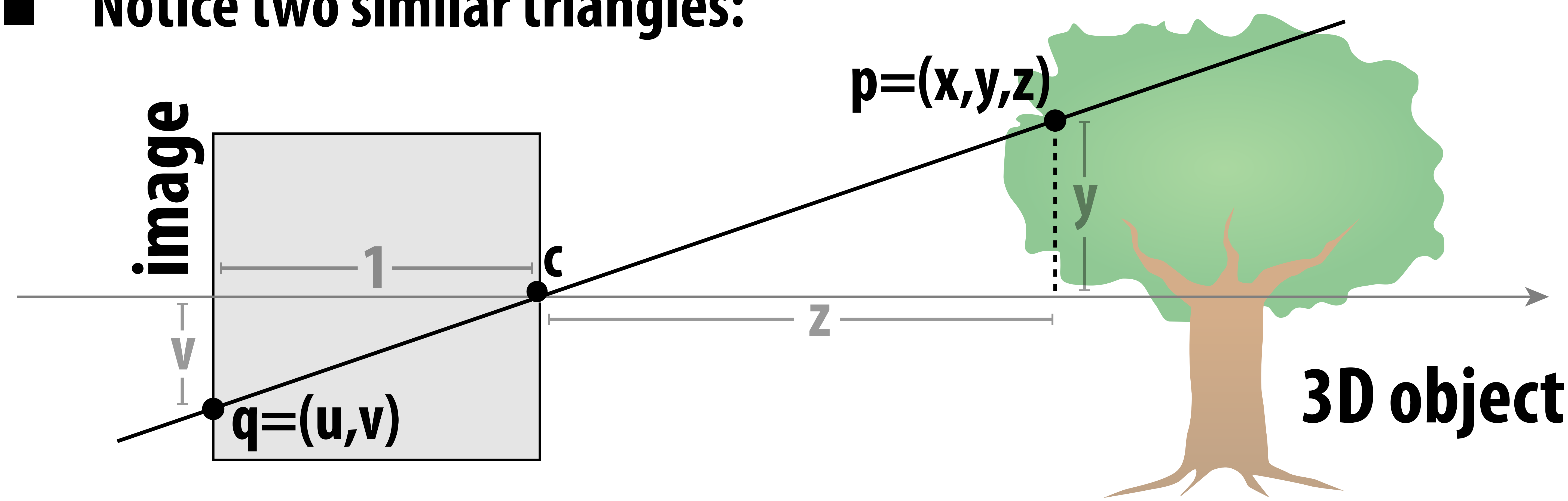
Perspective projection: side view

- Where exactly does a point p = (x,y,z) end up on the image?
- Let's call the image point q=(u,v)



Perspective projection: side view

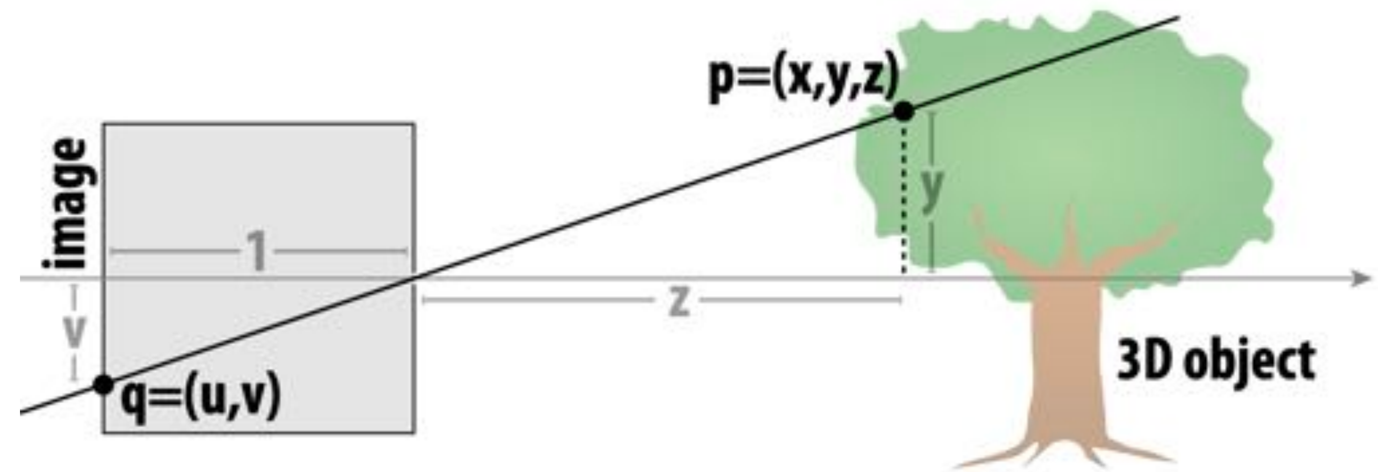
- Where exactly does a point p = (x,y,z) end up on the image?
- Let's call the image point q=(u,v)
- Notice two similar triangles:



- Assume camera has unit size, origin is at pinhole c
- Then v/1 = y/z, i.e., vertical coordinate is just the slope y/z

Perspective Projection in Homogeneous Coordinates

- **Q: How can we perform perspective projection* using homogeneous coordinates?**
- **The basic idea of the pinhole camera model is to "divide by z"**
- **So, we can build a matrix that "copies" the z coordinate into the homogeneous coordinate**
- **Division by the homogeneous coordinate now gives us perspective projection onto the plane $z = 1$**



$$(x, y, z) \mapsto (x/z, y/z)$$

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

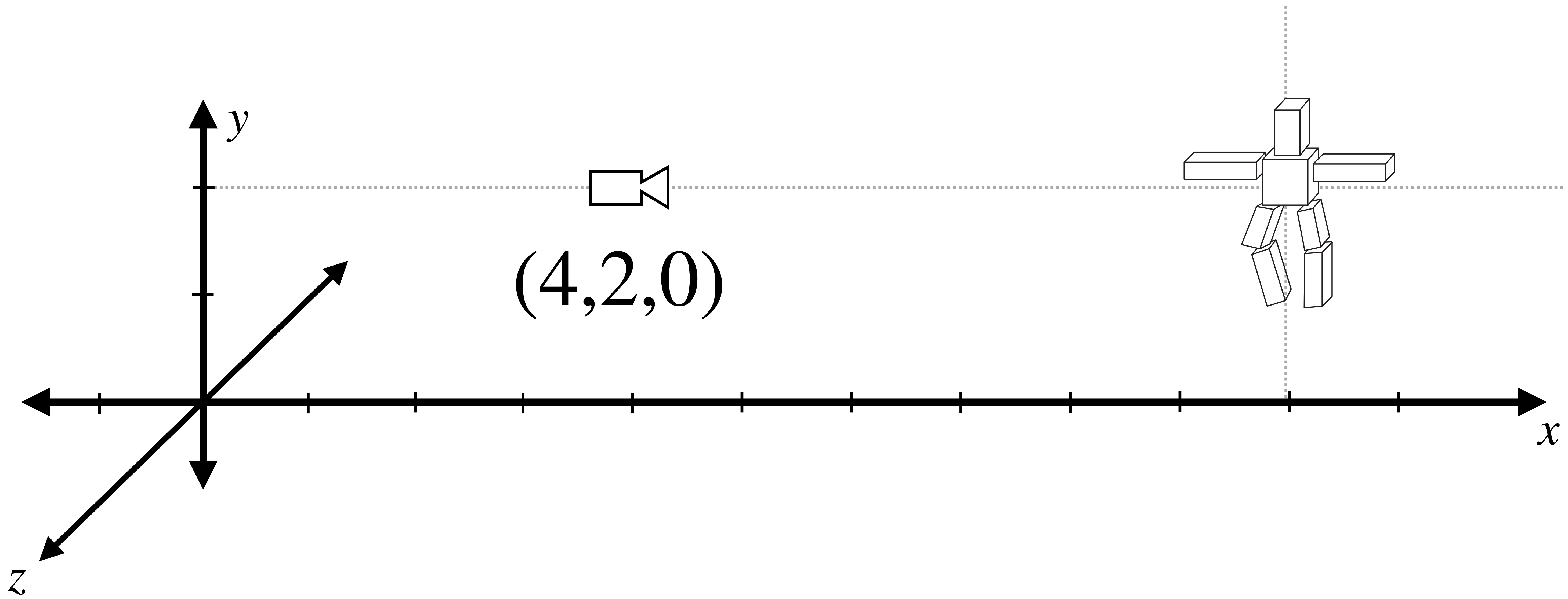
$$\Longrightarrow \begin{bmatrix} x/z \\ y/z \\ 1 \end{bmatrix}$$

***Assuming a pinhole camera at (0,0,0) looking down the z-axis**

Let's make this a little more interesting

Simple camera transform

Consider camera at $(4,2,0)$, looking down x-axis, object given in world coordinates:

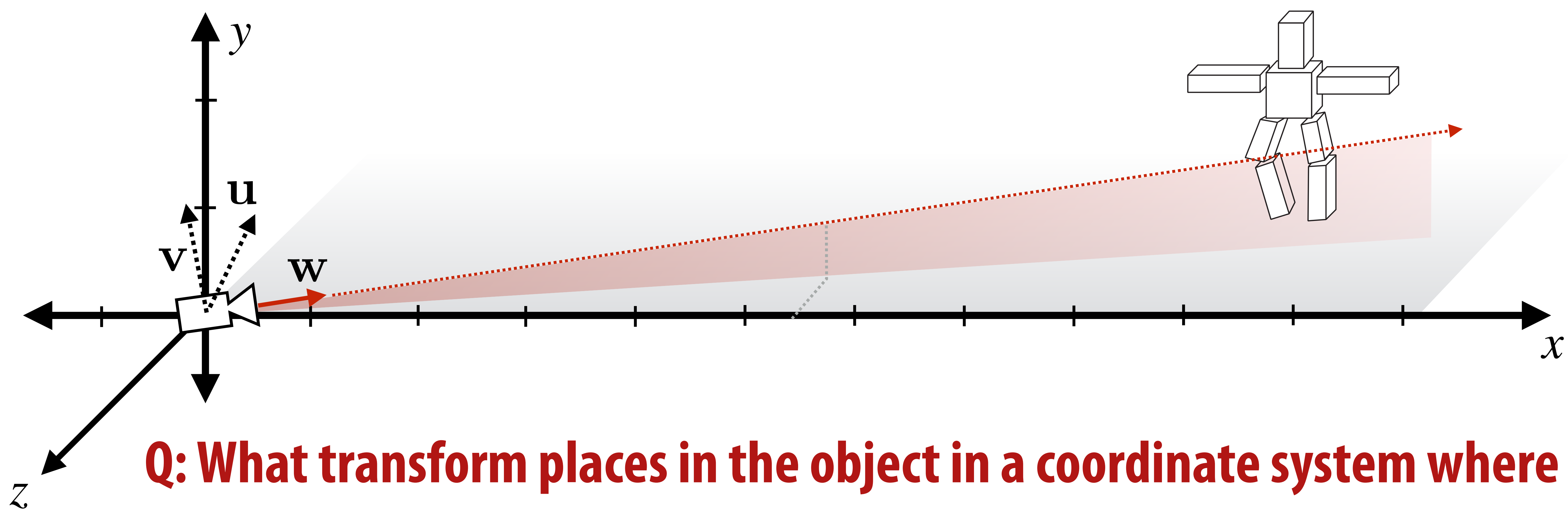


Q: What spatial transformation puts in the object in a coordinate system where the camera is at the origin, looking down the $-z$ axis?

- 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

Camera looking in a different direction

Now consider a camera looking in a direction $\mathbf{w} \in \mathbb{R}^3$



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

- Construct vectors $\mathbf{u}$, $\mathbf{v}$ orthogonal to $\mathbf{w}$
  – e.g., pick an "up" vector $\mathbf{v}$, let $\mathbf{u} := \mathbf{v} \times \mathbf{w}$
- Build corresponding rotation matrix

$$R = \begin{bmatrix} -u_x & v_x & -w_x \\ -u_y & v_y & -w_y \\ -u_z & v_z & -w_z \end{bmatrix}$$

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

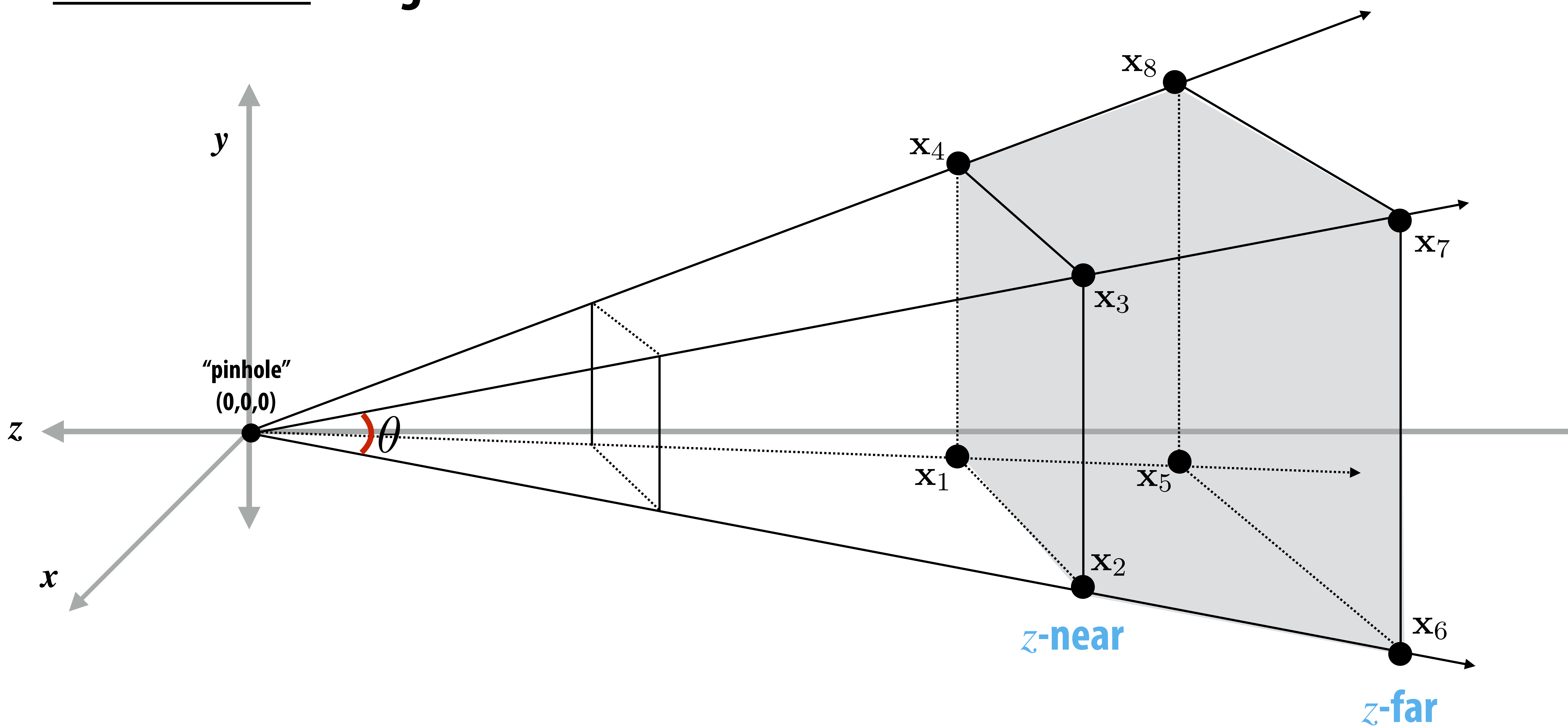
Now invert. (How do we do that?)

$$R^{-1} = R^{\mathsf{T}}$$

$$\begin{bmatrix} -u_x & -u_y & -u_z \\ v_x & v_y & v_z \\ -w_x & -w_y & -w_z \end{bmatrix}$$

View frustum

View frustum is region the camera can see:



- Top / bottom / left / right planes correspond to four sides of the image
- Near / far planes correspond to closest/furthest thing we want to draw

Clipping

- "Clipping" eliminates triangles not visible to the camera / in view frustum
  - Don't waste time rasterizing primitives (e.g., triangles) you can't see!
  - Discarding individual fragments is expensive ("fine granularity")
  - Makes more sense to toss out whole primitives ("coarse granularity")
  - Still need to deal with primitives that are partially clipped…

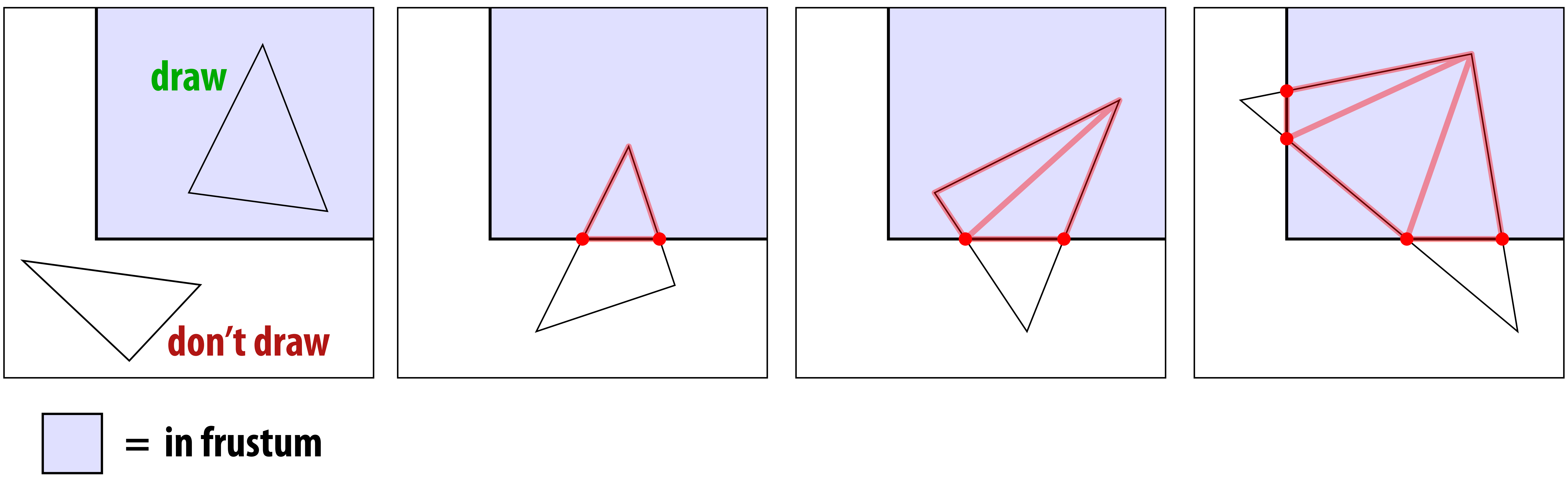
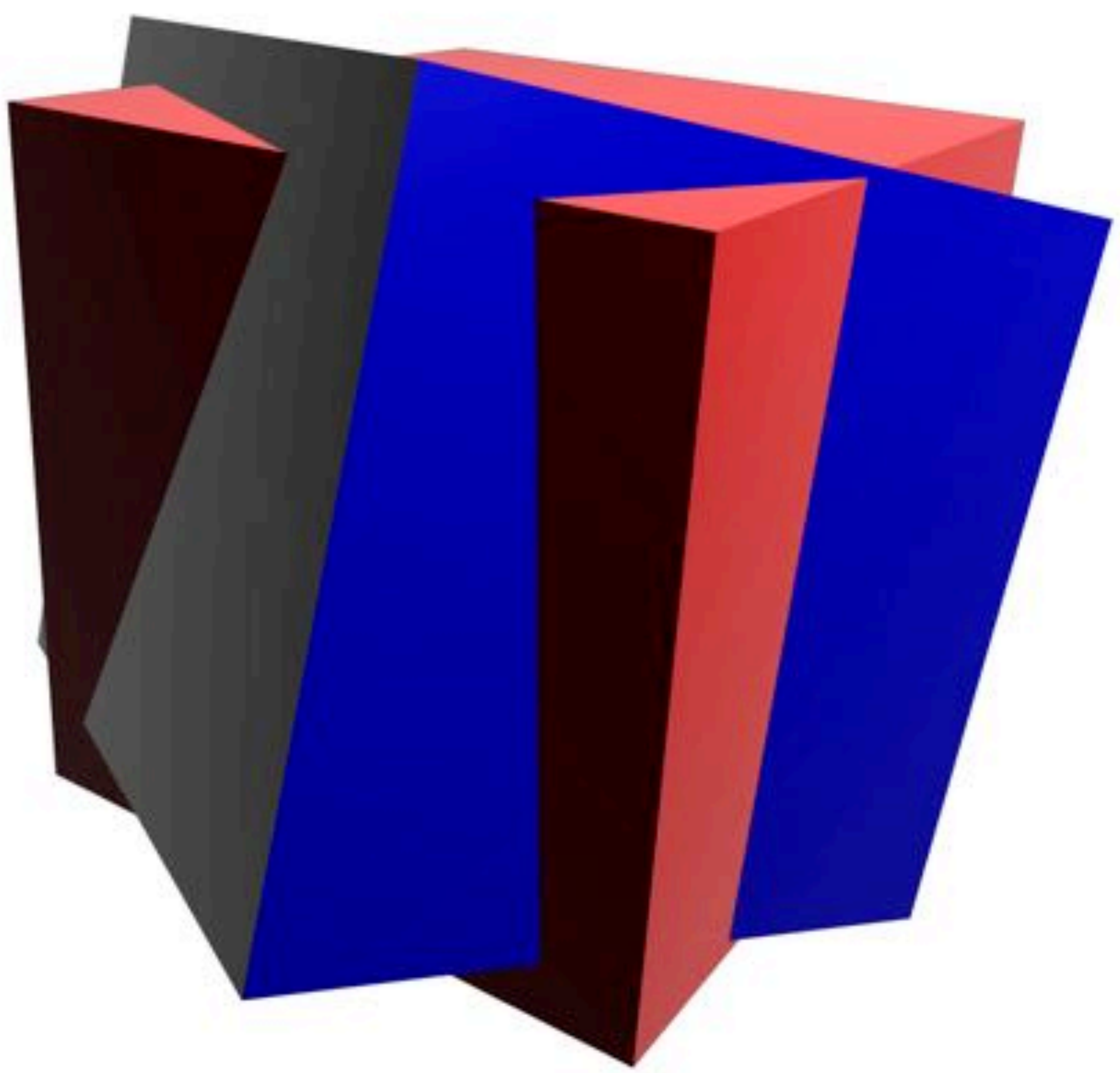


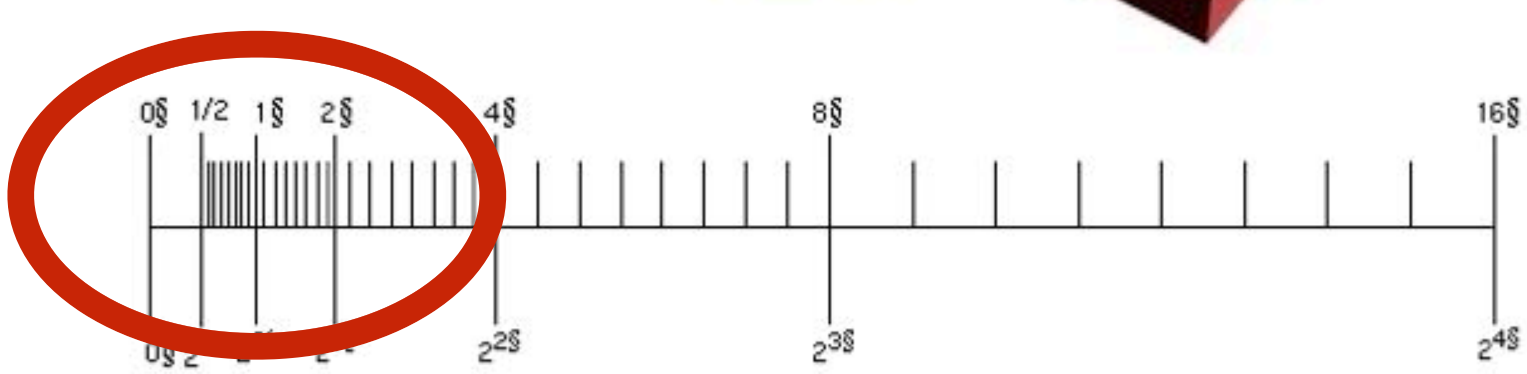
image credit: Jason L. McKesson (https://paroj.github.io/gltut/)

Near/Far Clipping

- Why have near/far clipping planes?
  - Some primitives (e.g., triangles) may have vertices both in front & behind eye! (Causes headaches for rasterization, e.g., checking if fragments are behind eye)
  - Also important for dealing with finite precision of depth buffer / limitations on storing depth as floating point values

near = 10^{-1}
far = 10^{3}

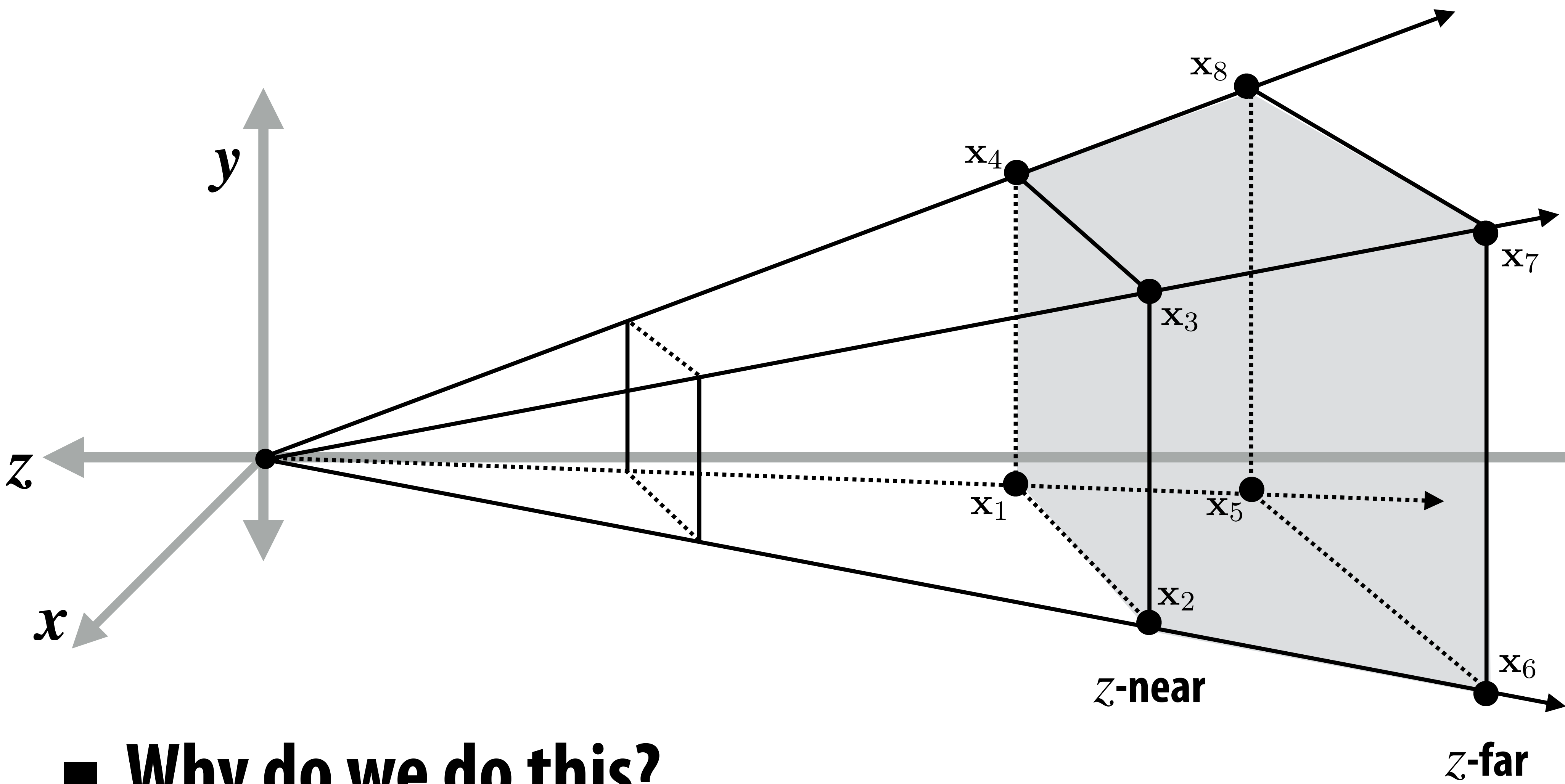


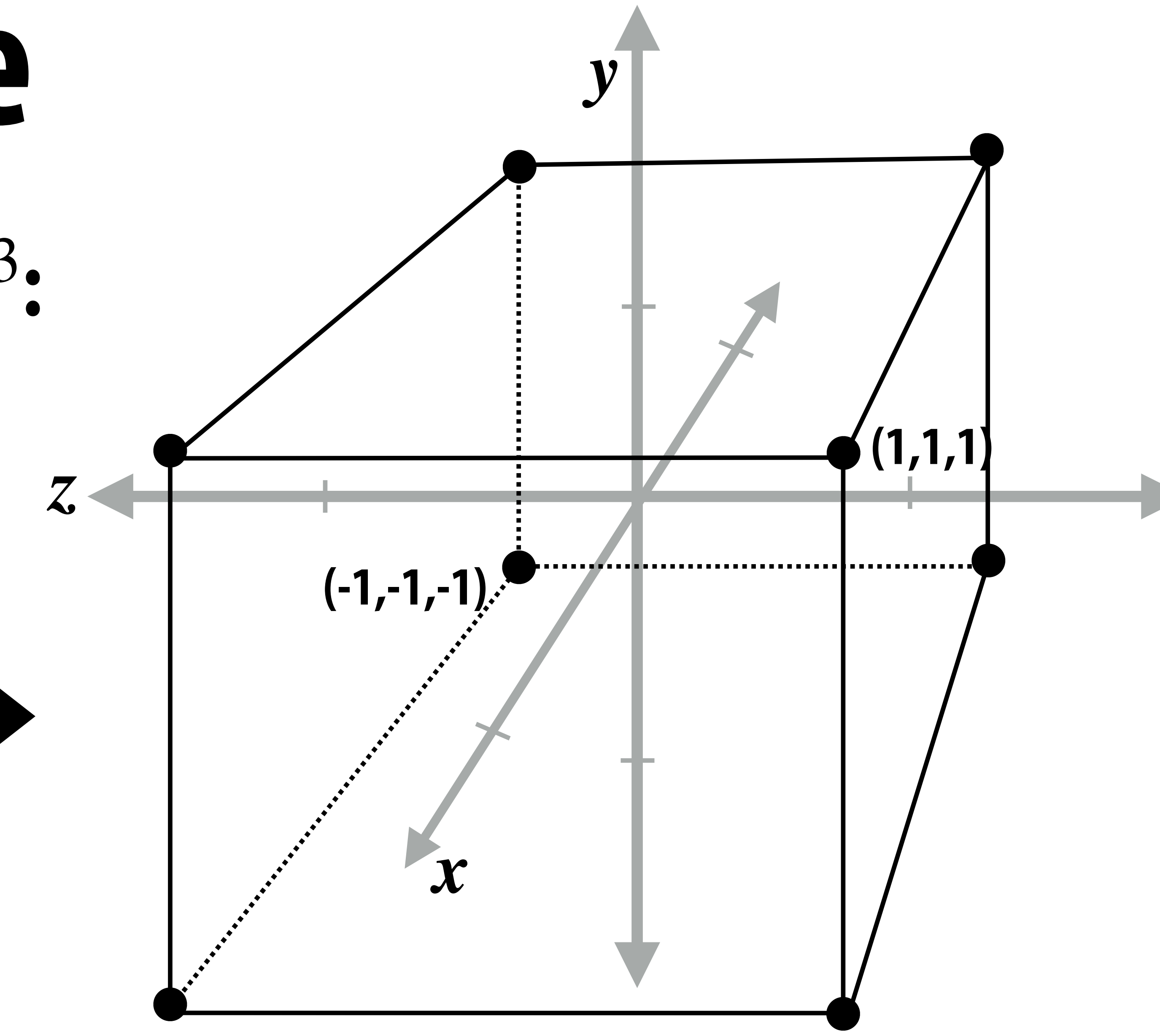


floating point has more "resolution" near zero—hence more precise resolution of primitive-primitive intersection

Mapping frustum to unit cube

Before projecting to 2D, map view frustum to cube $[-1,1]^3$:





- Why do we do this?
- Makes clipping much easier!
  - just discard points outside range [-1,1]
  - need to think about partially-clipped triangles
- Q: How can we express this mapping as a matrix?
- A: Solve $A\mathbf{x}_i = \mathbf{y}_i$ for unknown entries of A

scale to size 2

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

translate to origin

l = left b = bottom n = near
r = right t = top f = far

$$\begin{aligned} \mathbf{x}_1 &= \{l,b,n,1\} \\ \mathbf{x}_2 &= \{r,b,n,1\} \\ \mathbf{x}_3 &= \{r,t,n,1\} \\ \mathbf{x}_4 &= \{l,t,n,1\} \\ \mathbf{x}_5 &= \{l,b,f,1\} \\ \mathbf{x}_6 &= \{r,b,f,1\} \\ \mathbf{x}_7 &= \{r,t,f,1\} \\ \mathbf{x}_8 &= \{l,t,f,1\} \end{aligned}$$

$$\begin{aligned} \mathbf{y}_1 &= \{-1,-1,1,1\} \\ \mathbf{y}_2 &= \{1,-1,1,1\} \\ \mathbf{y}_3 &= \{1,1,1,1\} \\ \mathbf{y}_4 &= \{-1,1,1,1\} \\ \mathbf{y}_5 &= \{-1,-1,-1,1\} \\ \mathbf{y}_6 &= \{1,-1,-1,1\} \\ \mathbf{y}_7 &= \{1,1,-1,1\} \\ \mathbf{y}_8 &= \{-1,1,-1,1\} \end{aligned}$$

(orthographic projection)

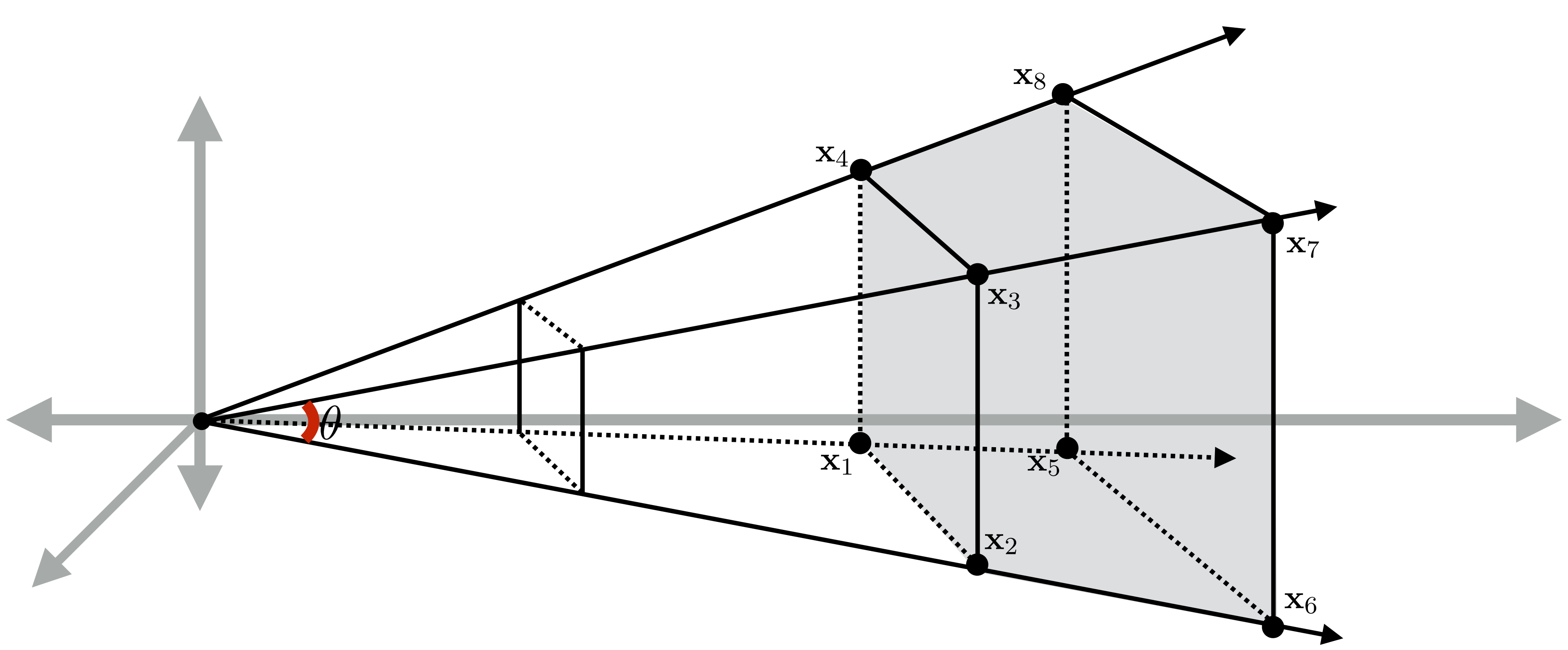
Matrix for Perspective Transform

Recall our basic perspective projection matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z \end{bmatrix} \longmapsto \begin{bmatrix} x/z \\ y/z \\ 1 \\ 1 \end{bmatrix}$$

objects shrink in distance

Full perspective matrix takes geometry of view frustum into account:

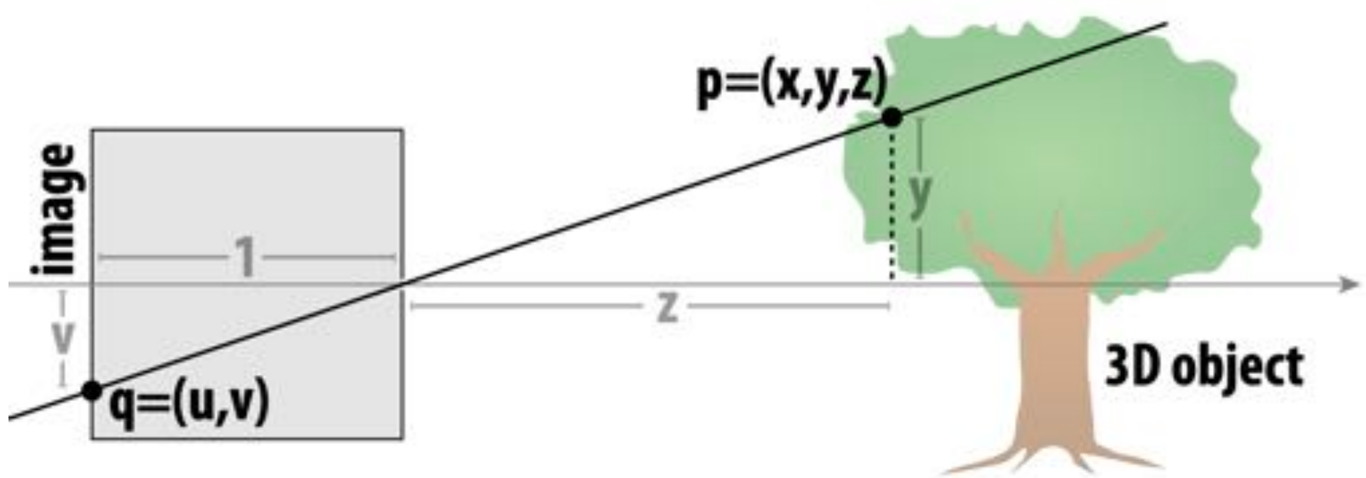


$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

l = left b = bottom n = near
r = right t = top f = far

For a derivation: http://www.songho.ca/opengl/gl_projectionmatrix.html

Does this look like our pinhole projection matrix?

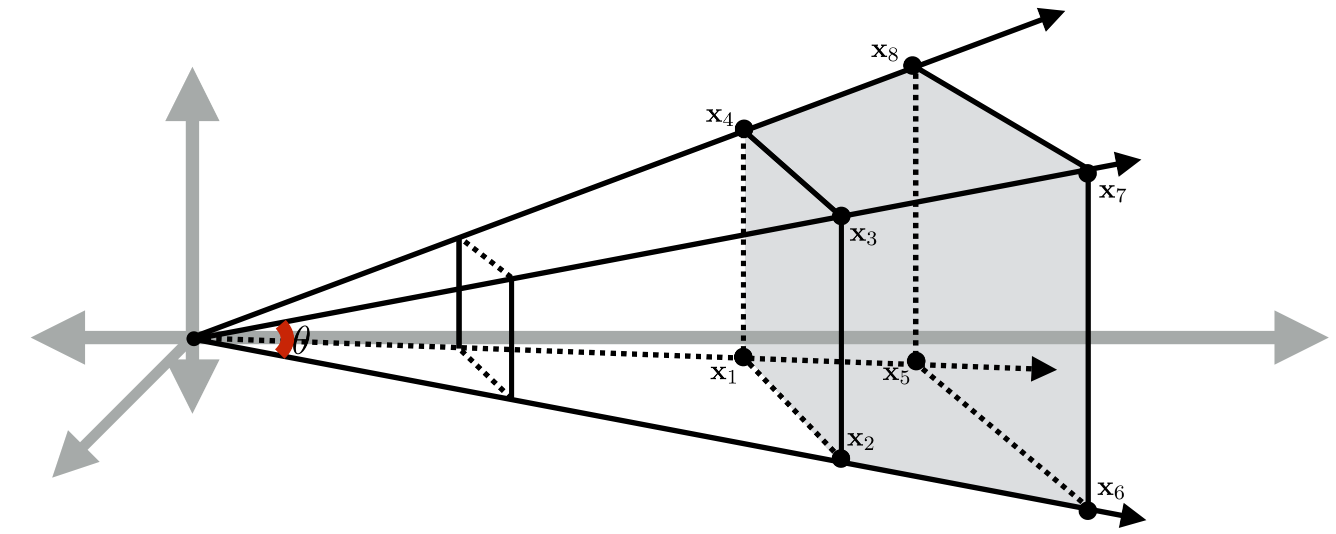


$$(x, y, z) \mapsto (x/z, y/z)$$

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

$$\Longrightarrow \begin{bmatrix} x/z \\ y/z \\ 1 \end{bmatrix}$$

*Assuming a pinhole camera at (0,0,0) looking down the z-axis



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

l = left $\quad$ b = bottom $\quad$ n = near

r = right $\quad$ t = top $\quad$ f = far

Screen Transformation (Vulkan, Direct3D)

- One last transformation is needed in the rasterization pipeline: transform from viewing plane to pixel coordinates
- E.g., suppose we want to draw all points that fall inside the square [-1,1] x [-1,1] on the z = 1 plane, into a W x H pixel image with upper-left origin.

"normalized device coordinates"

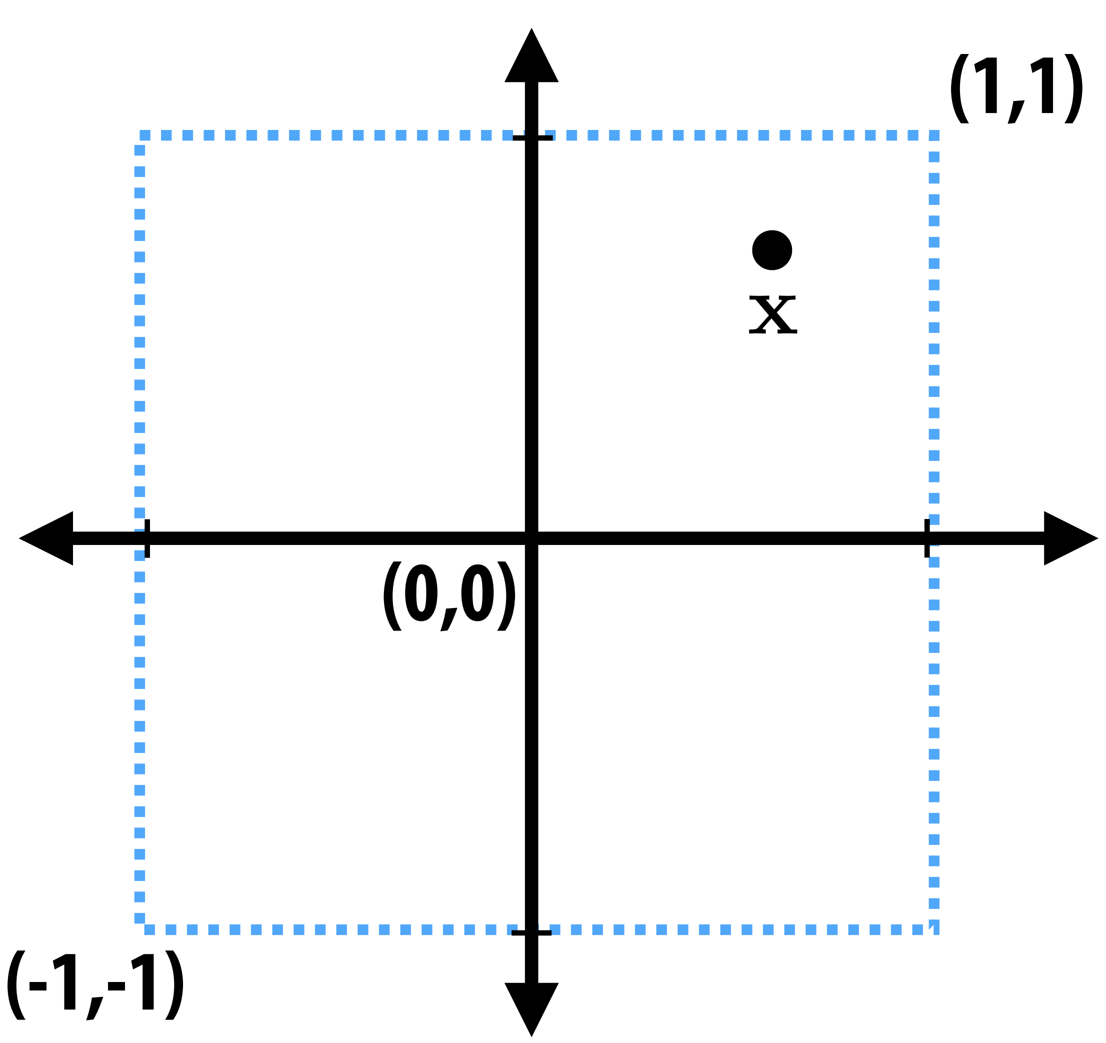
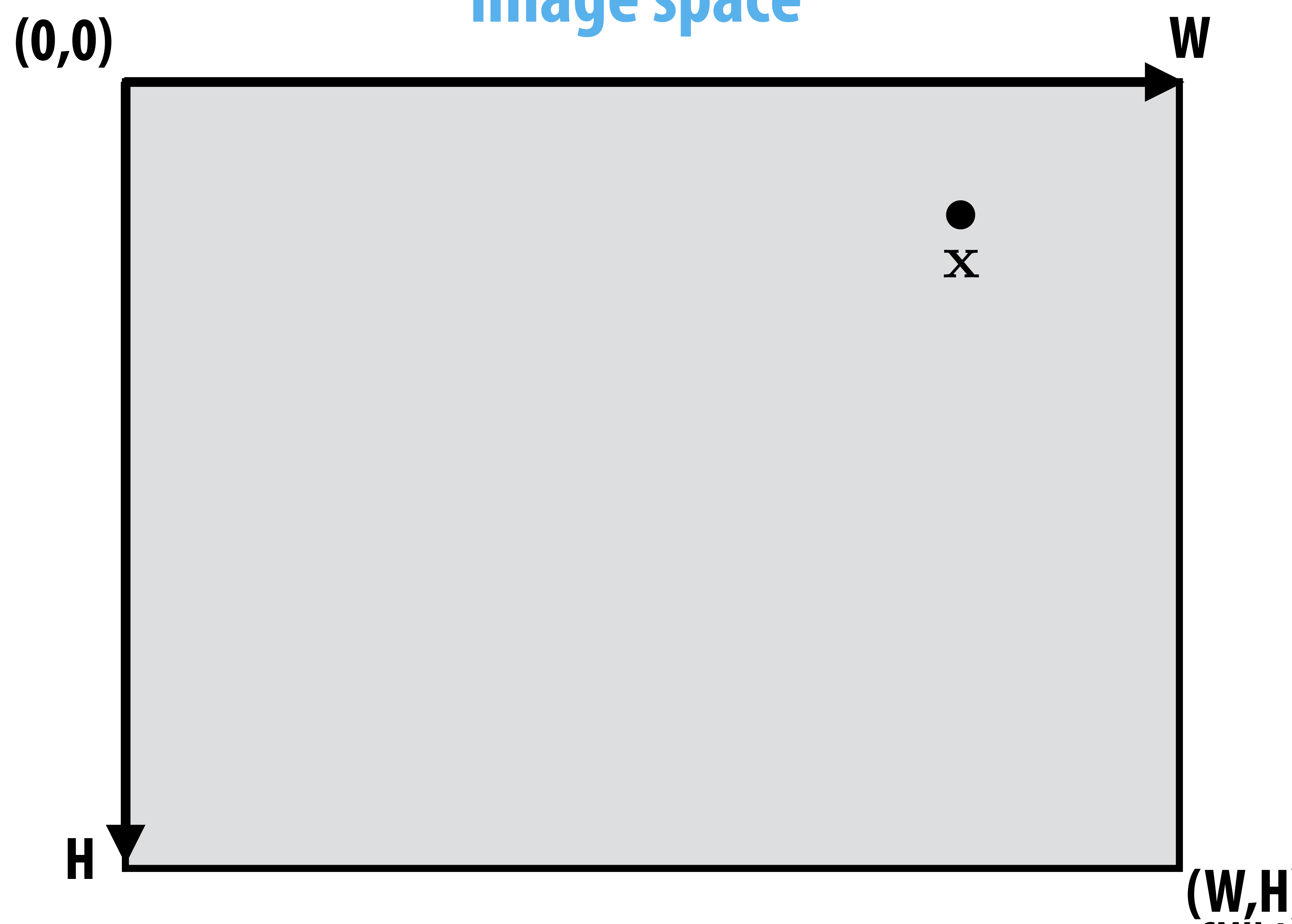


image space

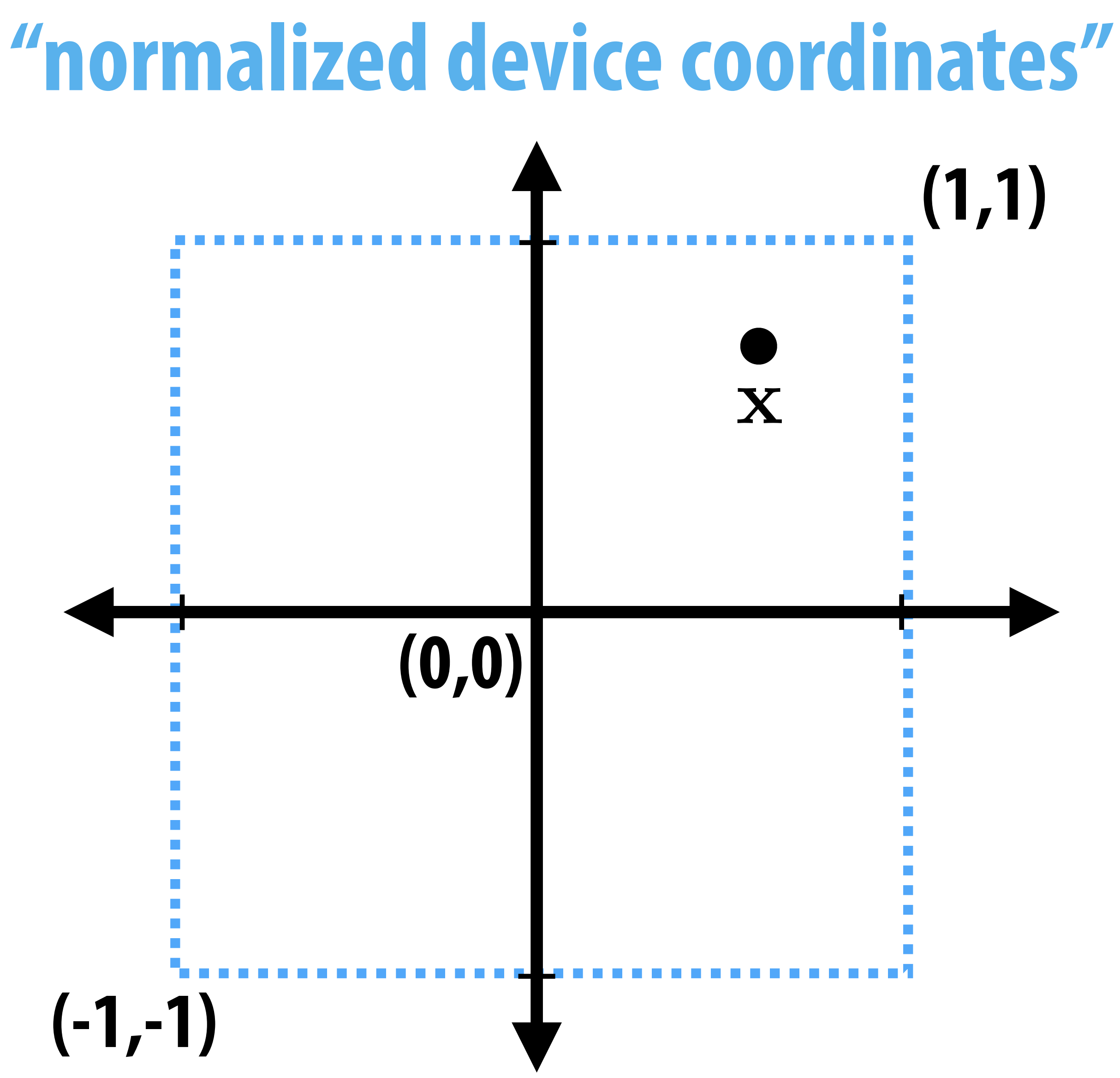


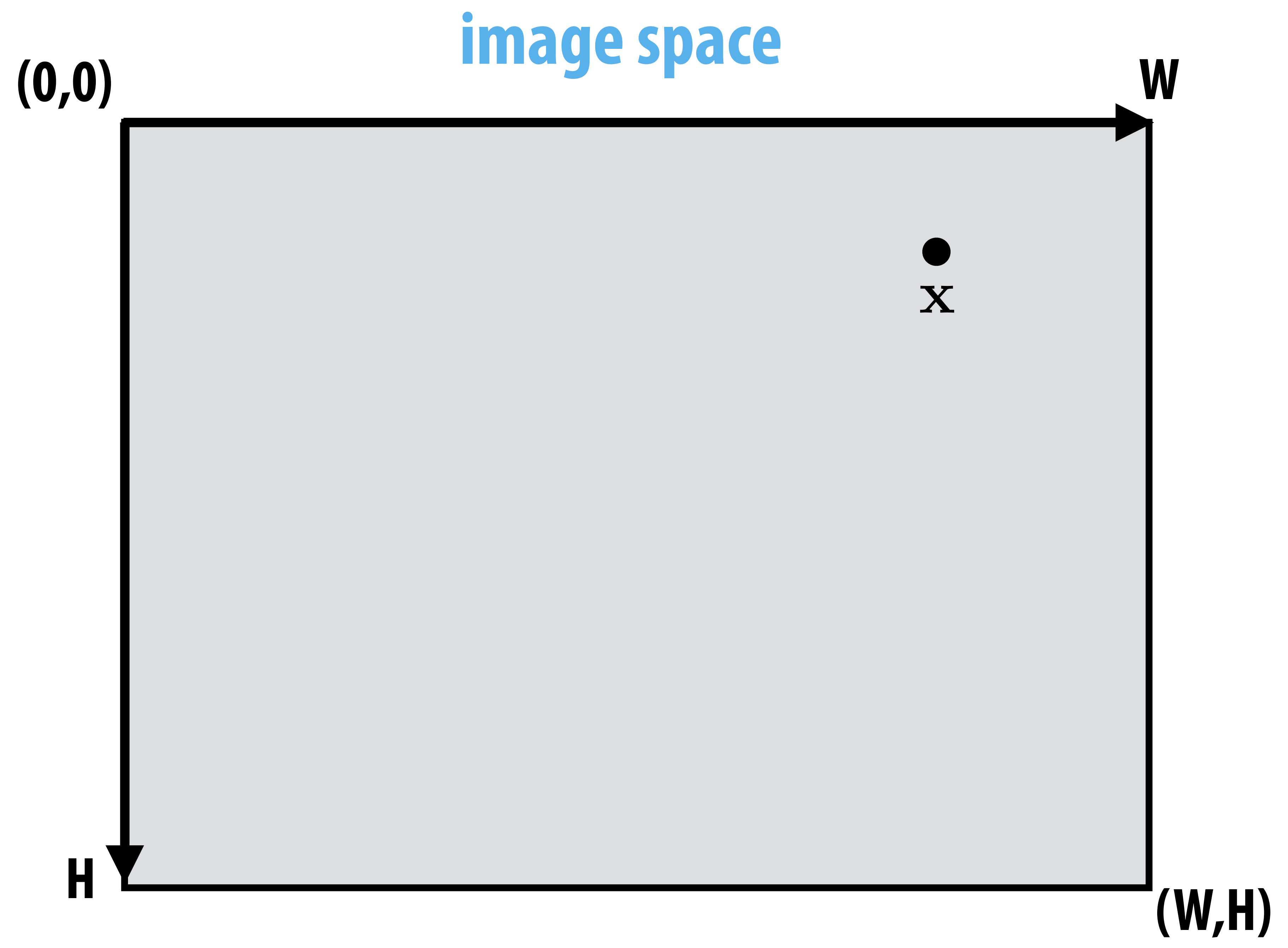
Q: What transformation(s) would you apply?

(Careful: y is down!)

Screen Transformation

- **Projection will take points to [-1,1] x [-1,1] on the z = 1 plane; transform into a W x H pixel image**





Step 1: reflect about x-axis
Step 2: translate by (1,1)
Step 3: scale by (W/2,H/2)

Screen Transformation (OpenGL)

- One last transformation is needed in the rasterization pipeline: transform from viewing plane to pixel coordinates
- E.g., suppose we want to draw all points that fall inside the square [-1,1] x [-1,1] on the z = 1 plane, into a W x H pixel image

"normalized device coordinates"

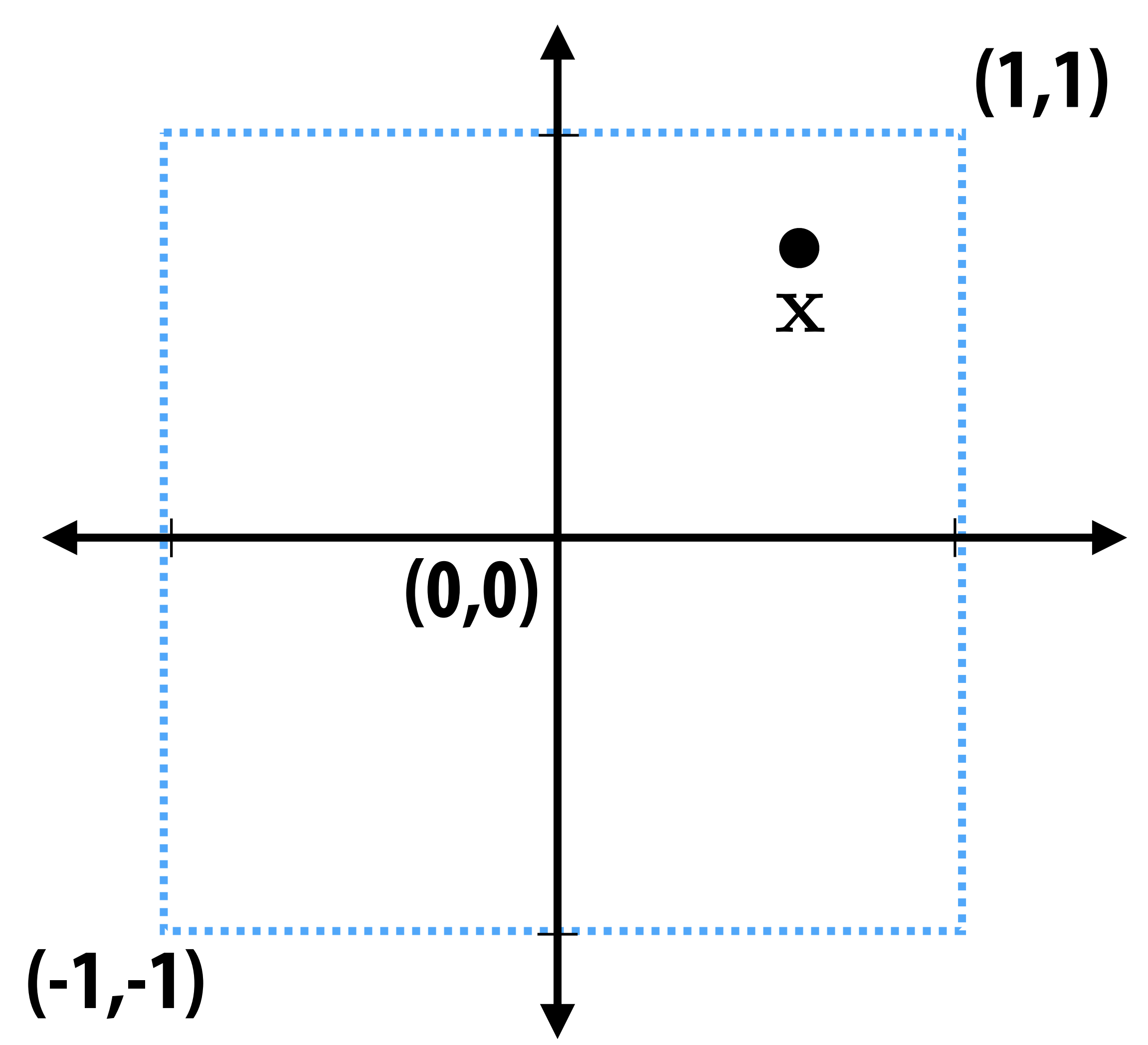
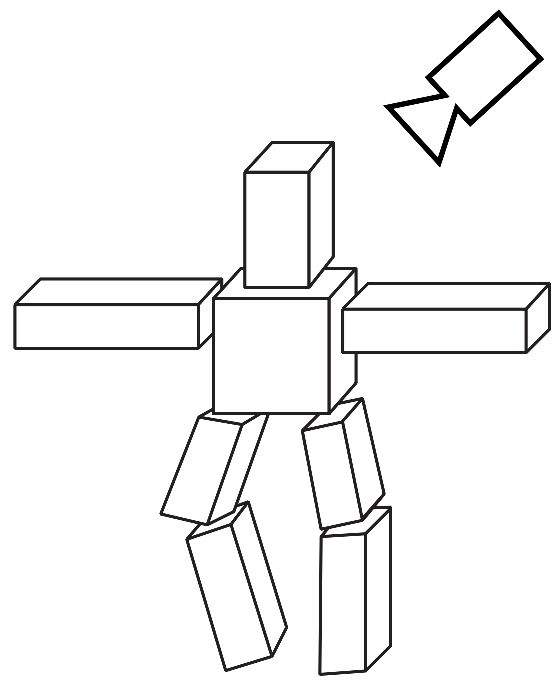


image space

Q: What transformation(s) would you apply?

Transformations: From Objects to the Screen

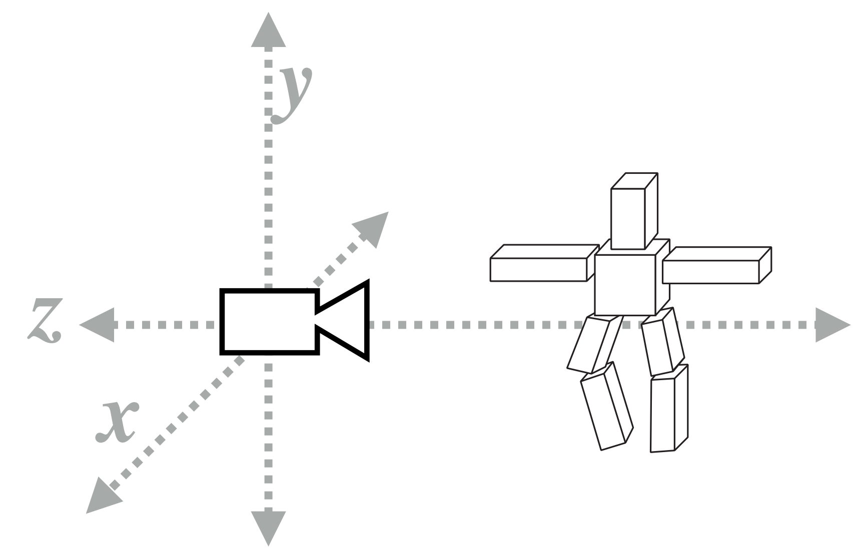
[WORLD COORDINATES]



original description of objects

view transform

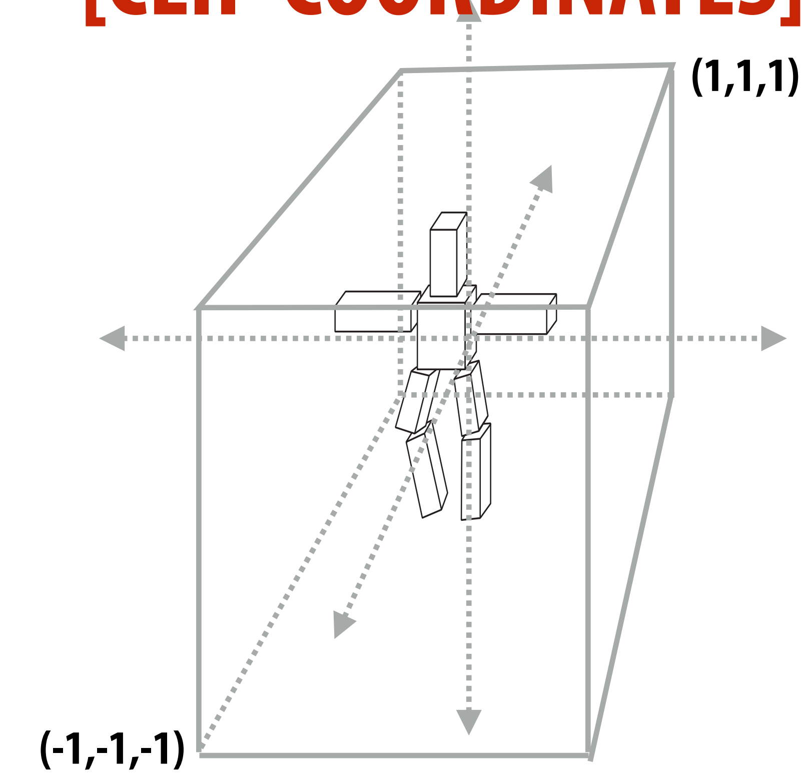
[VIEW COORDINATES]



all positions now expressed relative to camera; camera is sitting at origin looking down -z direction

projection transform

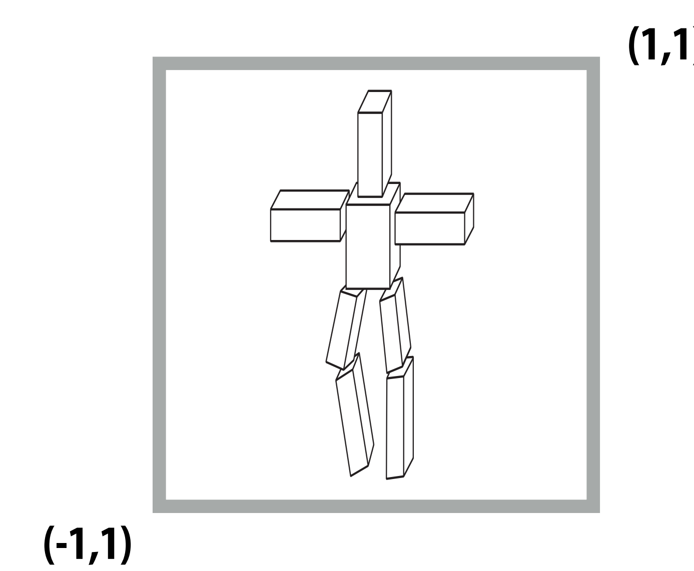
[CLIP COORDINATES]



everything visible to the camera is mapped to unit cube for easy "clipping"

perspective divide

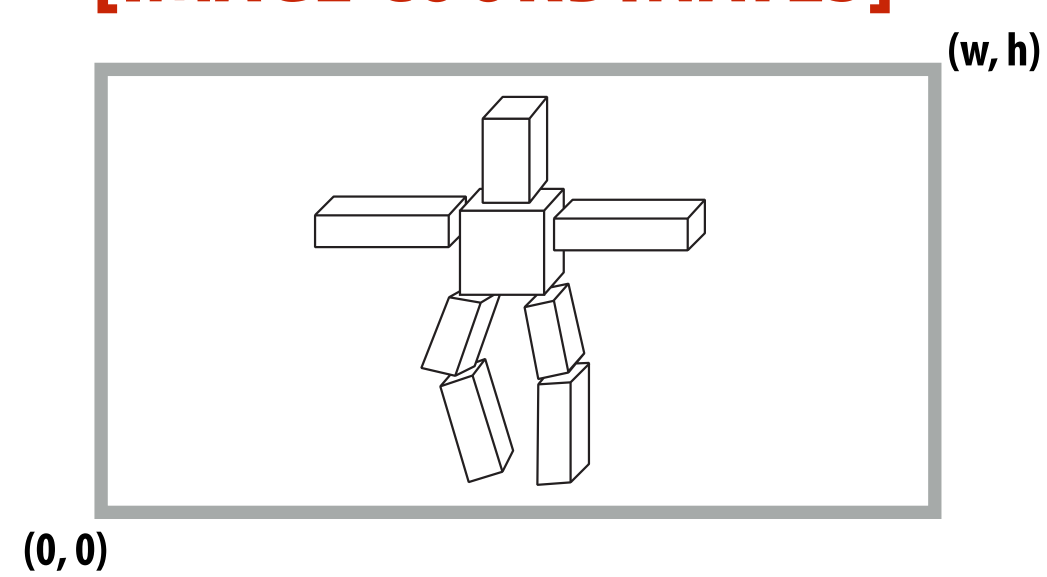
[NORMALIZED COORDINATES]



unit cube mapped to unit square via perspective divide

screen transform

[IMAGE COORDINATES]



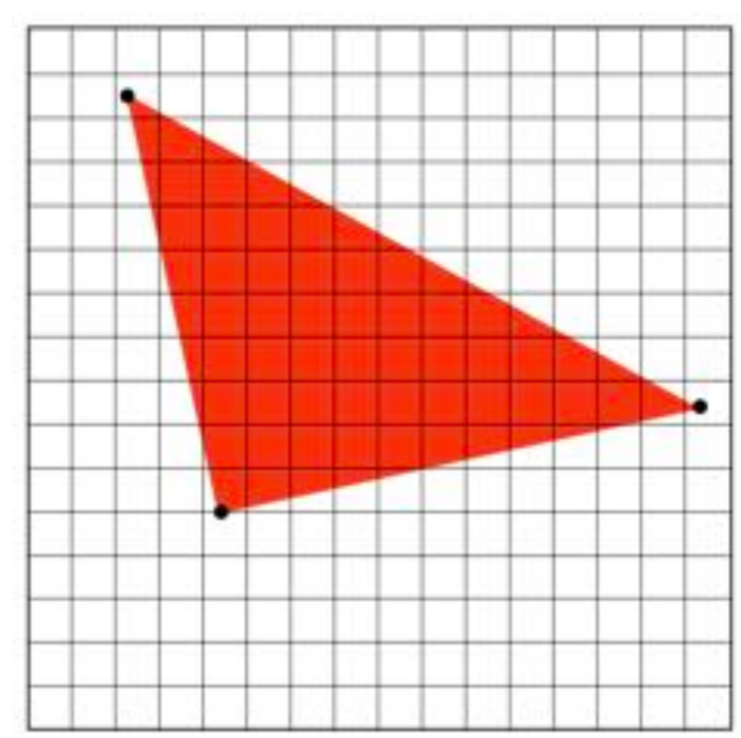
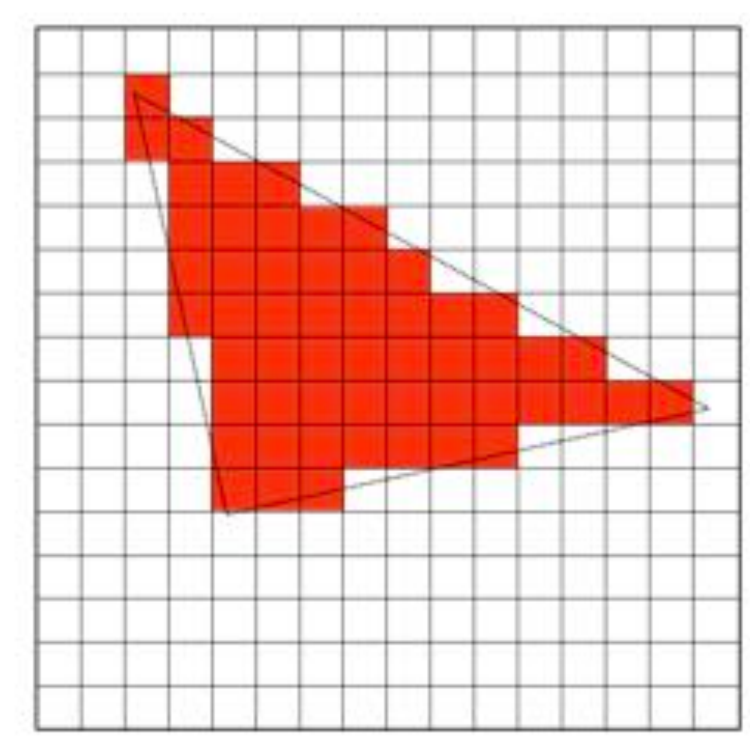
coordinates stretched to match image dimensions (and flipped upside-down)

2D primitives can now be drawn via rasterization

Drawing a Triangle
(and introduction to sampling)

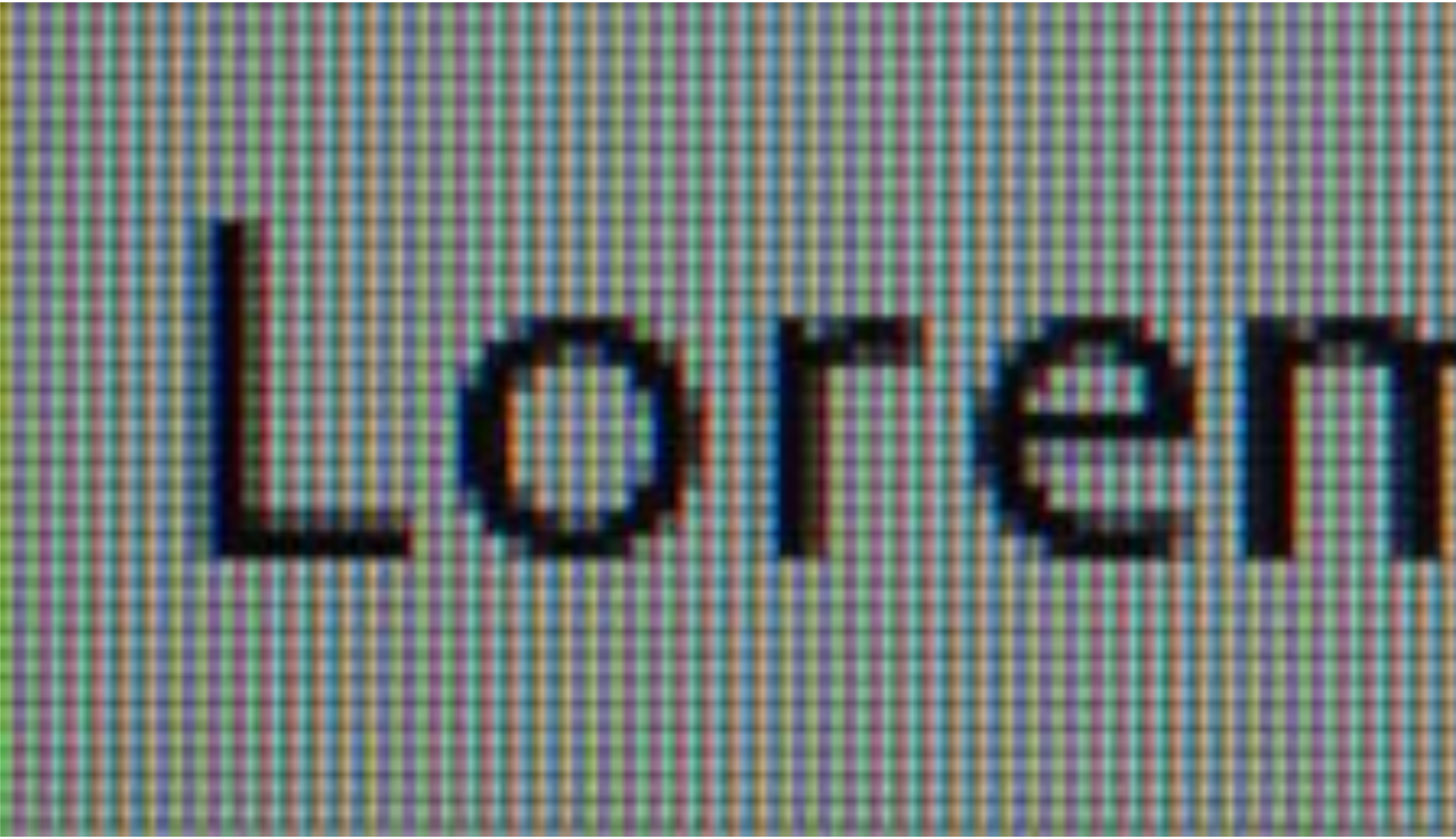
Rasterization

- Two major techniques for "getting stuff on the screen"
- Rasterization (TODAY)
  - for each primitive (e.g., triangle), which pixels light up?
  - extremely fast (BILLIONS of triangles per second on GPU)
  - harder (but not impossible) to achieve photorealism
  - perfect match for 2D vector art, fonts, quick 3D preview, …
- Ray tracing (LATER)
  - for each pixel, which primitives are seen?
  - easier to get photorealism
  - generally slower
  - much more later in the semester!

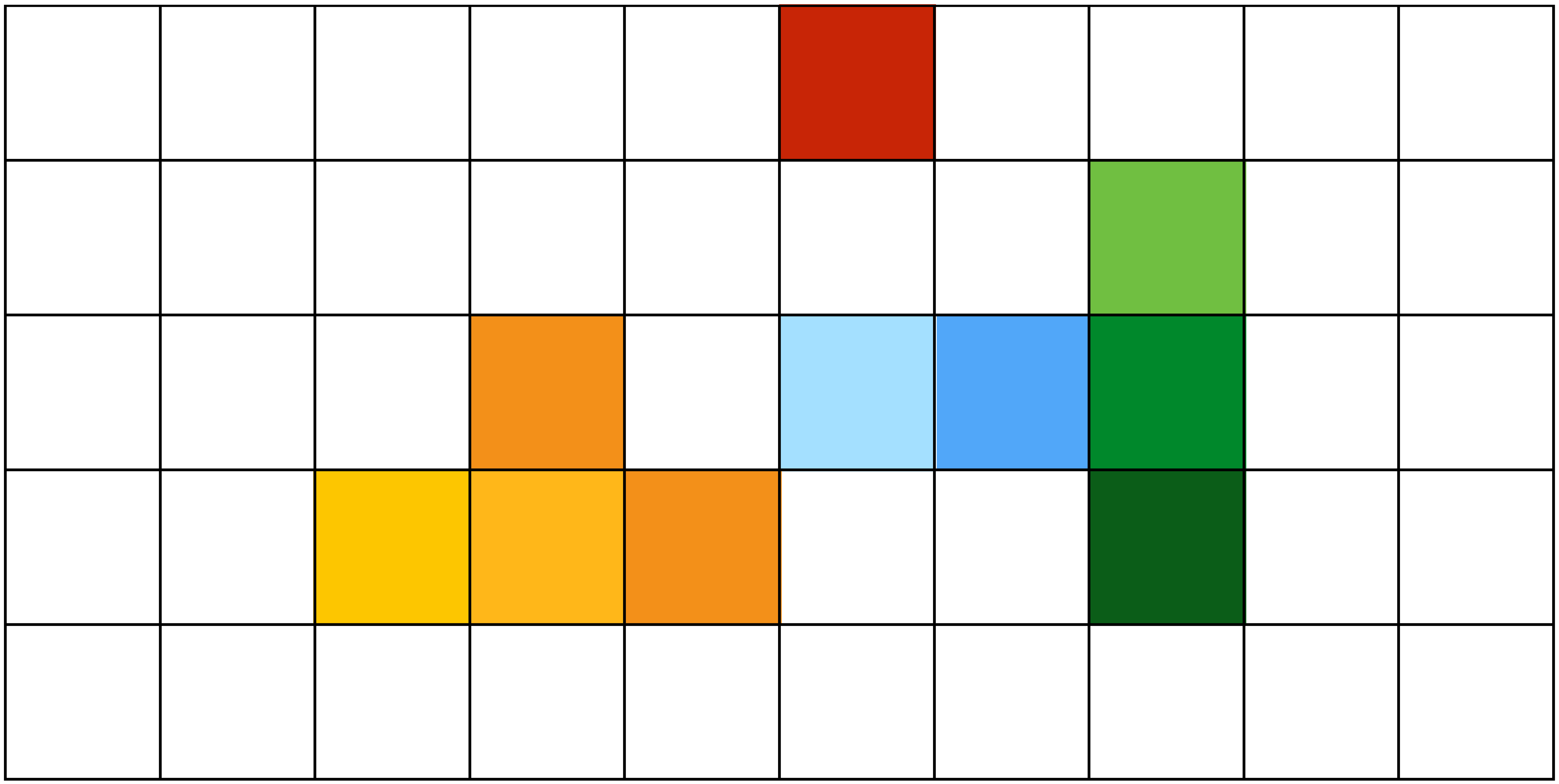
Let’s warm up by drawing some lines

Close up photo of pixels on a modern display



Output for a raster display

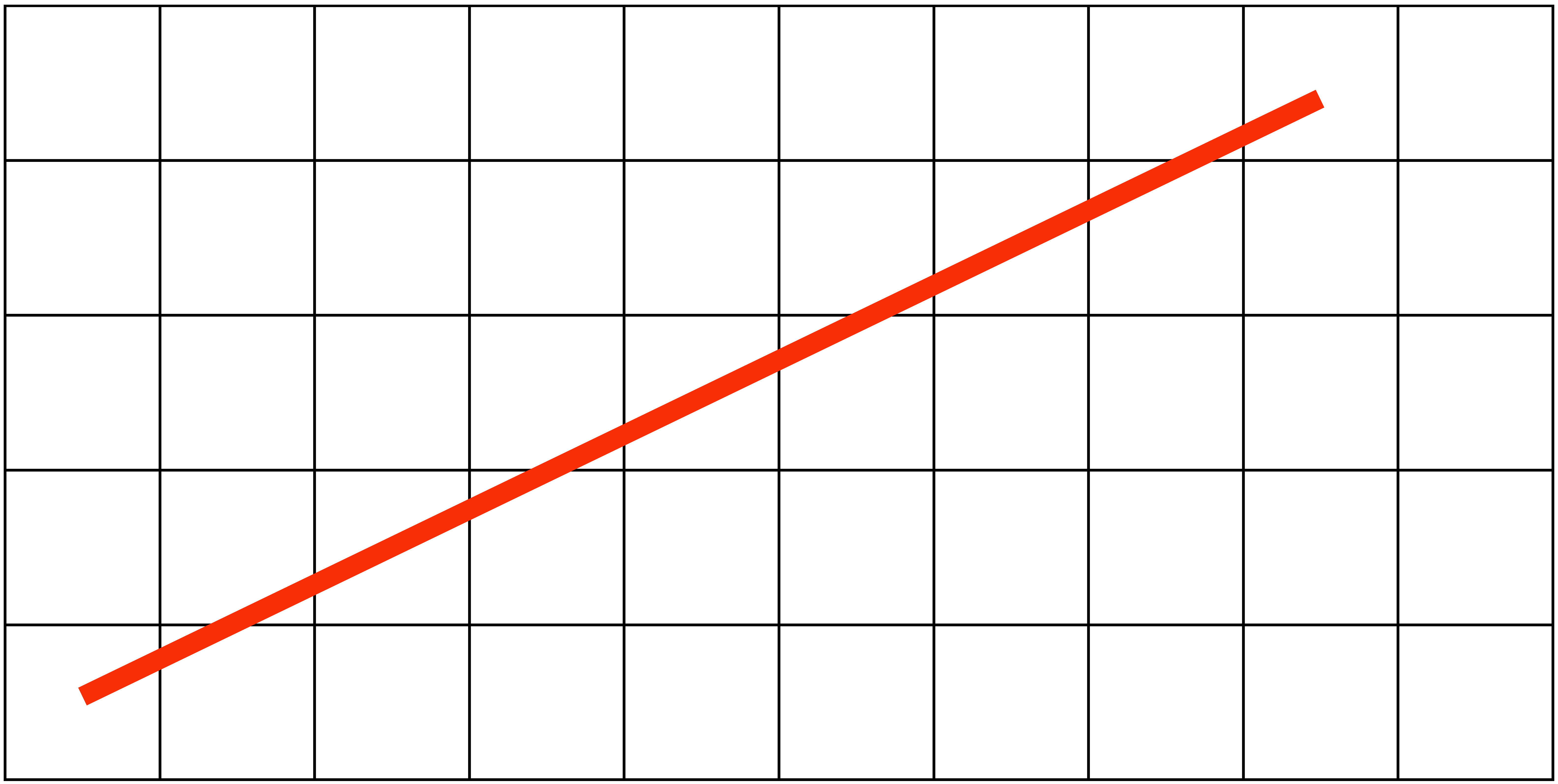
- Common abstraction of a raster display:
  - Image represented as a 2D grid of "pixels" (picture elements) **
  - Each pixel can can take on a unique color value



** We will strongly challenge this notion of a pixel "as a little square" soon enough.
But let's go with it for now. ;-)

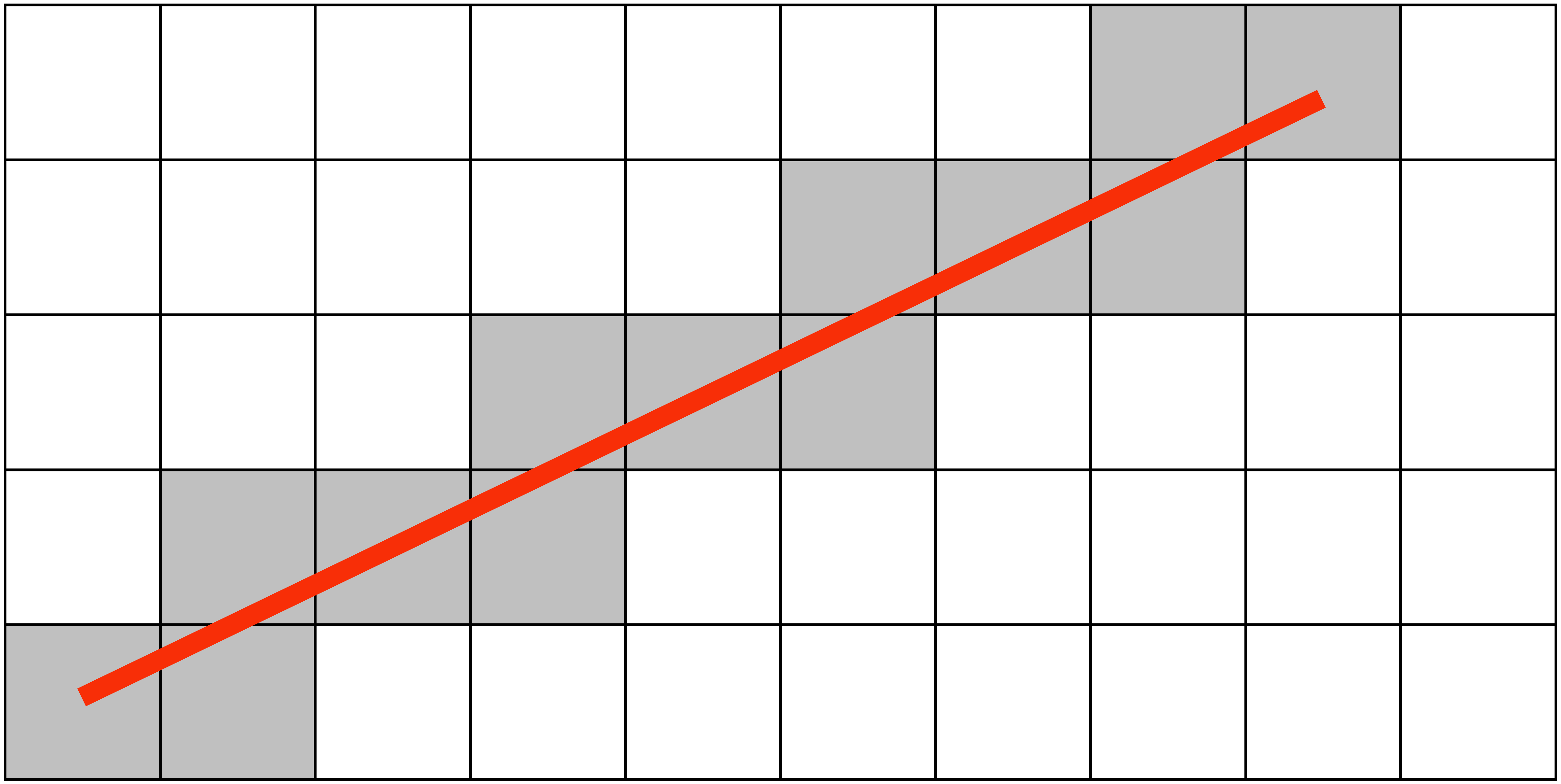
What pixels should we color in to depict a line?

"Rasterization": process of converting a continuous object to a discrete representation on a raster grid (pixel grid)



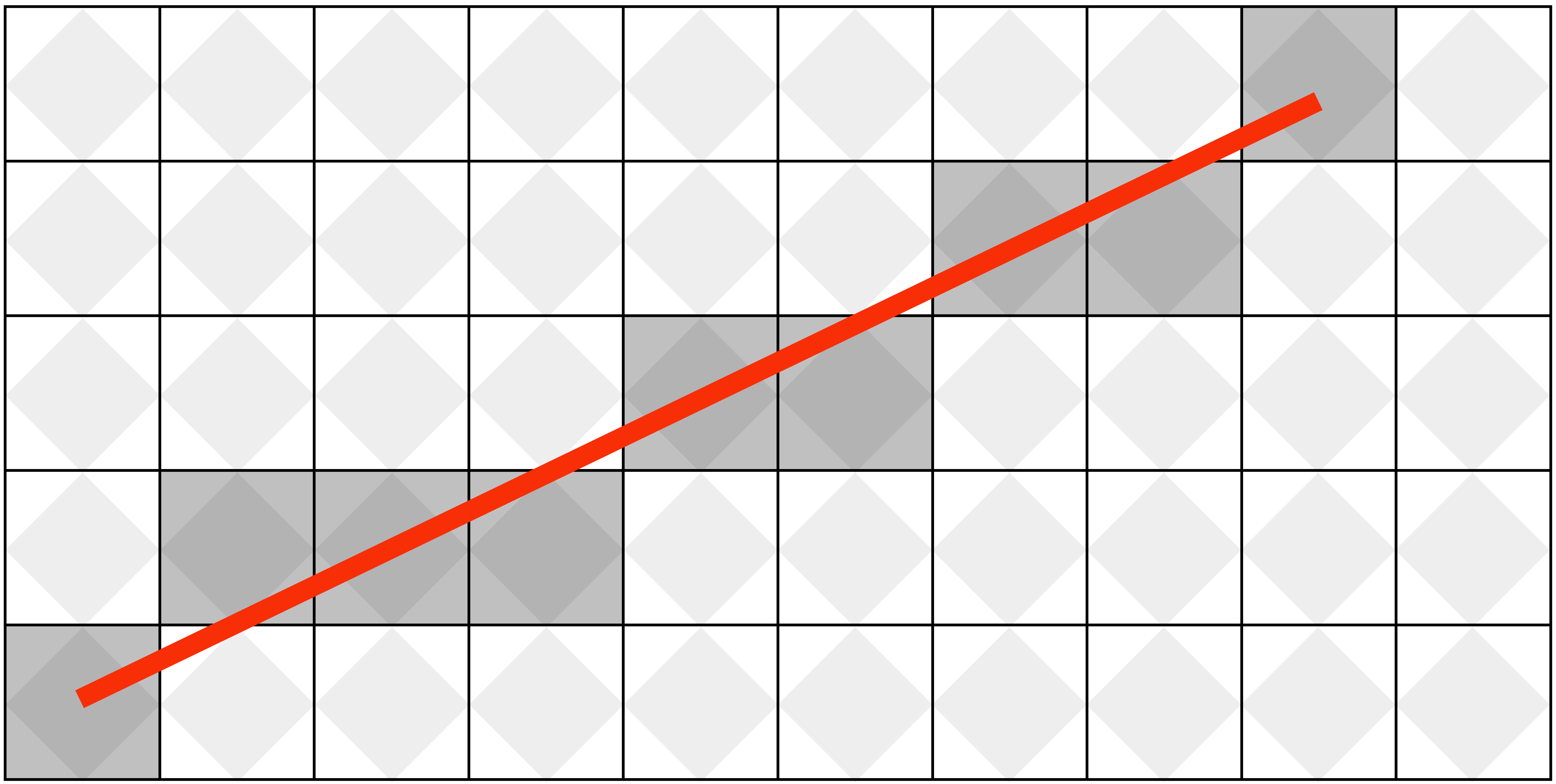
What pixels should we color in to depict a line?

Light up all pixels intersected by the line?



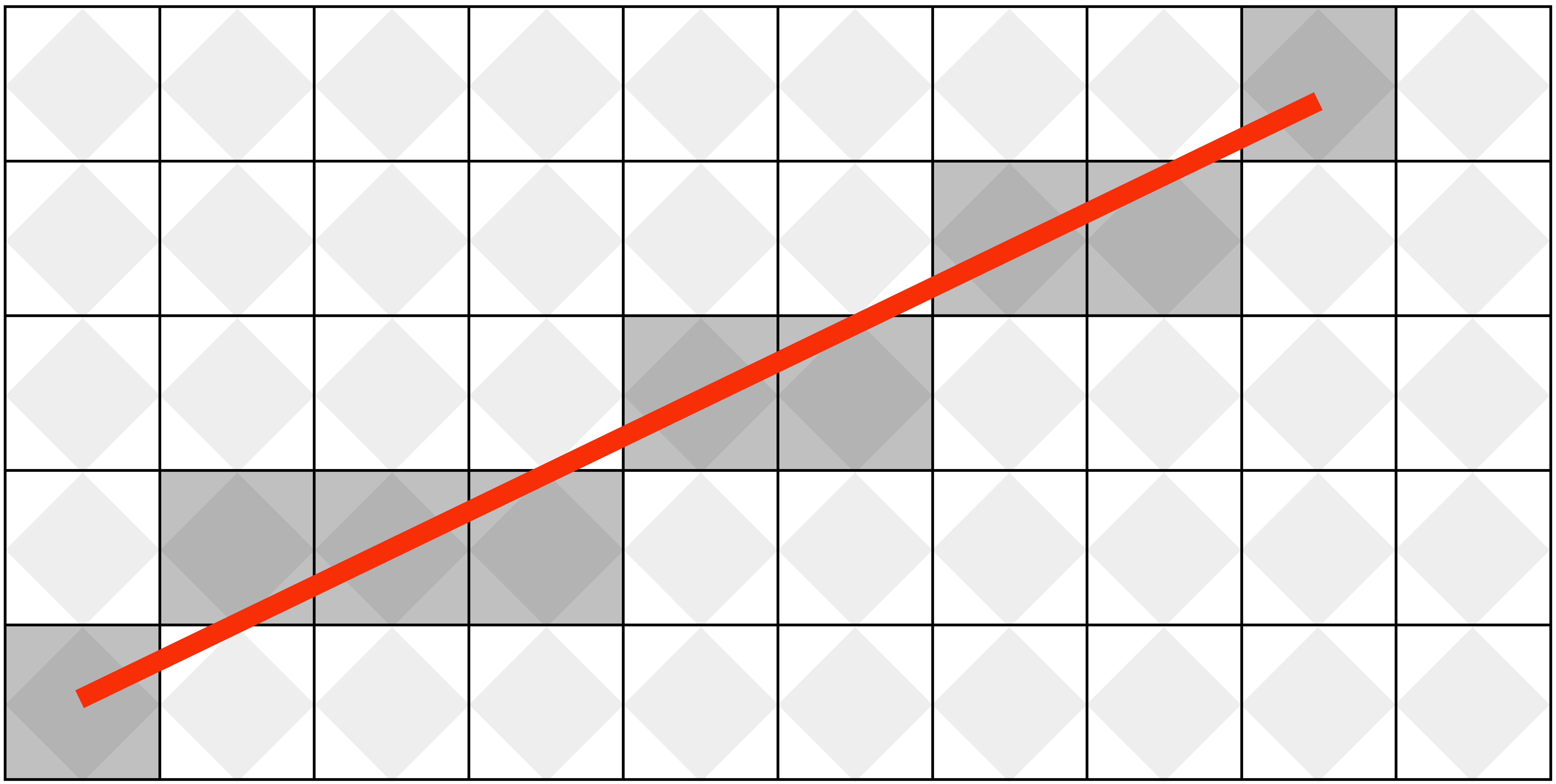
What pixels should we color in to depict a line?

Diamond rule (used by modern GPUs):
light up pixel if line passes through associated diamond



What pixels should we color in to depict a line?

Is there a right answer?
(consider a drawing a "line" with thickness)



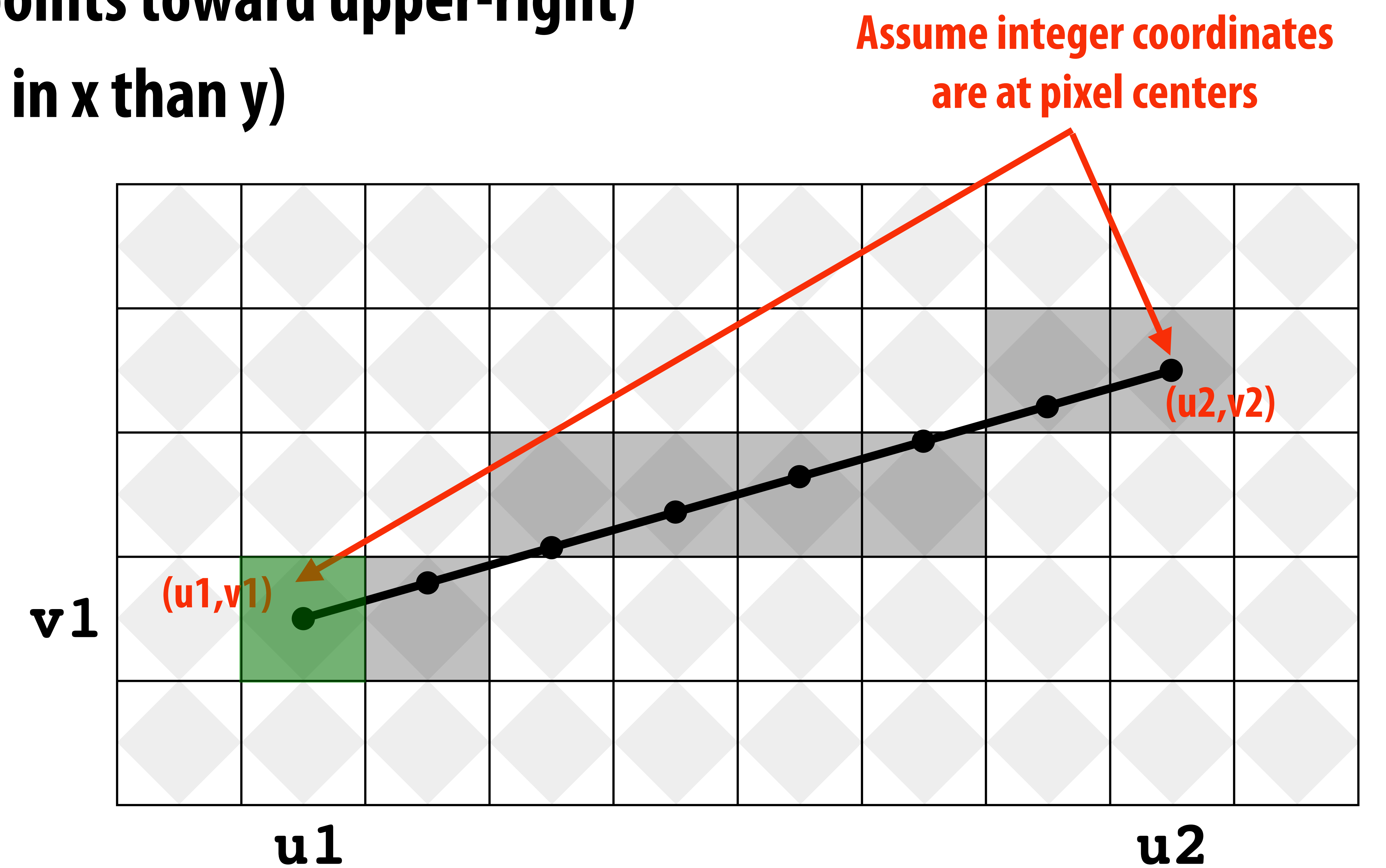
How do we find the pixels satisfying a chosen rasterization rule?

- Could check every single pixel in the image to see if it meets the condition...
  - $O(n^2)$ pixels in image vs. at most $O(n)$ "lit up" pixels
  - must be able to do better! (e.g., work proportional to number of pixels in the drawing of the line)

Incremental line rasterization

- Let's say a line is represented with integer endpoints: (u1,v1), (u2,v2)
- Slope of line: s = (v2-v1) / (u2-u1)
- Consider an easy special case:
  - u1 < u2, v1 < v2 (line points toward upper-right)
  - 0 < s < 1 (more change in x than y)

```
v = v1;
for(u=u1; u<=u2; u++)
{
   v += s;
   draw(u, round(v))
}
```

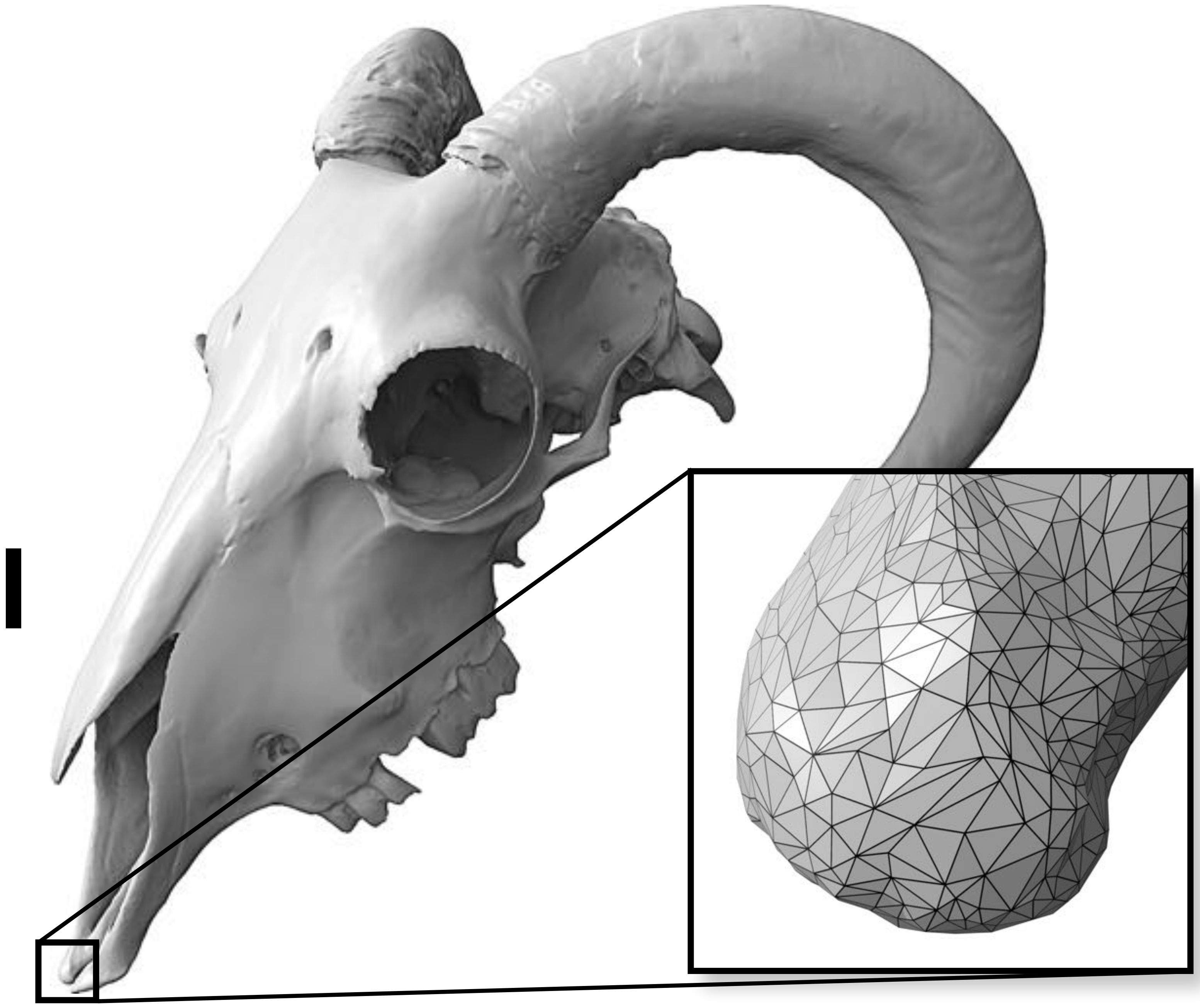


Easy to implement… not how lines are drawn in modern software/hardware!

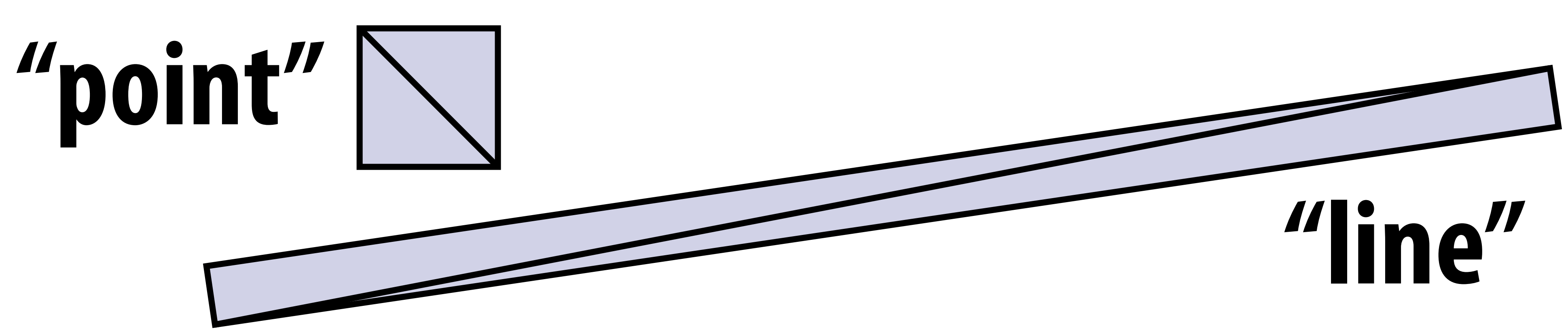
Ok, we have a basic line algorithm, what about triangles?

Why triangles?

- Rasterization pipeline converts all primitives to triangles
  - even points and lines!
- Why?
  - can approximate any shape
  - always planar, well-defined normal
  - easy to interpolate data at corners
    - "barycentric coordinates"

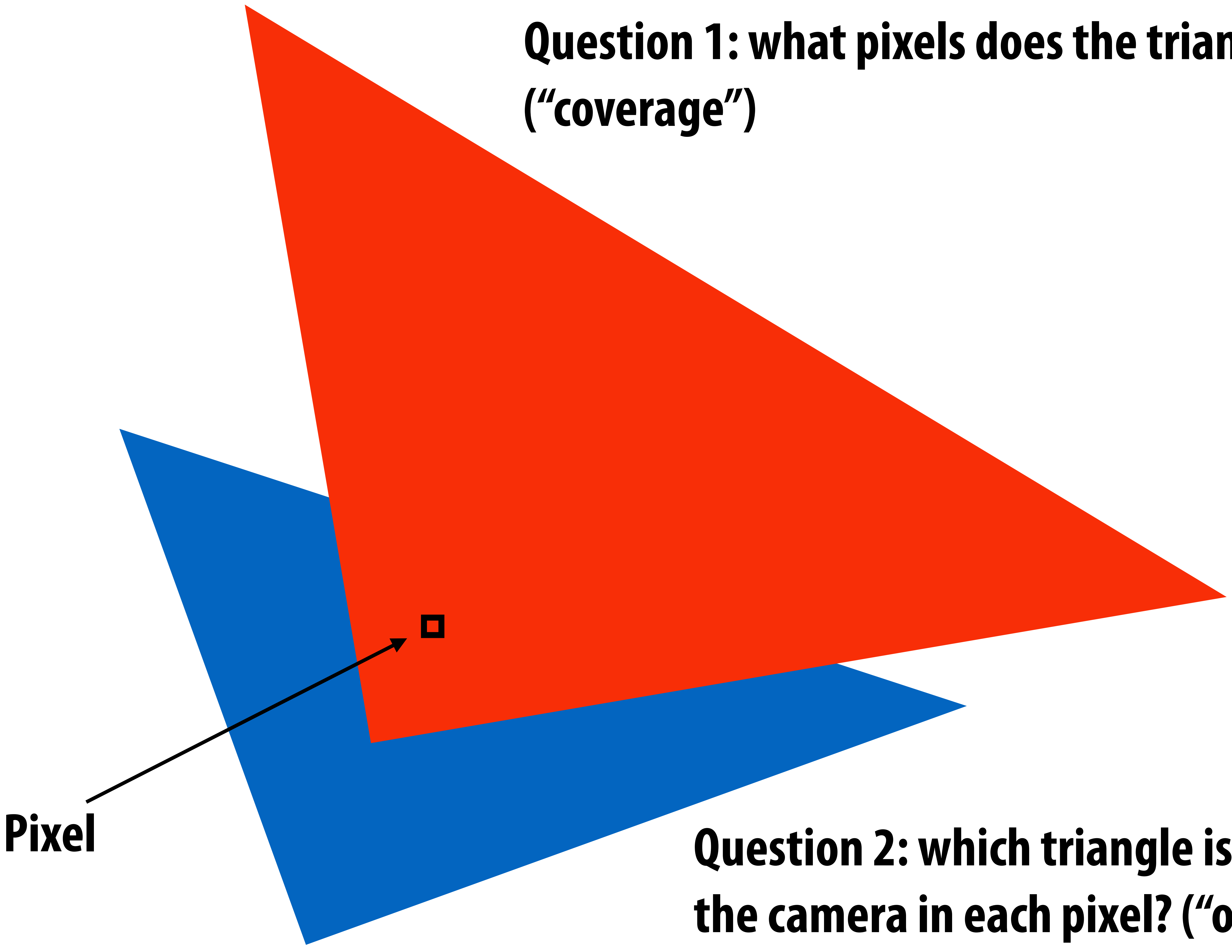


- Key reason: once everything is reduced to triangles, can focus on making an extremely well-optimized pipeline for drawing them



Let’s draw some triangles on the screen

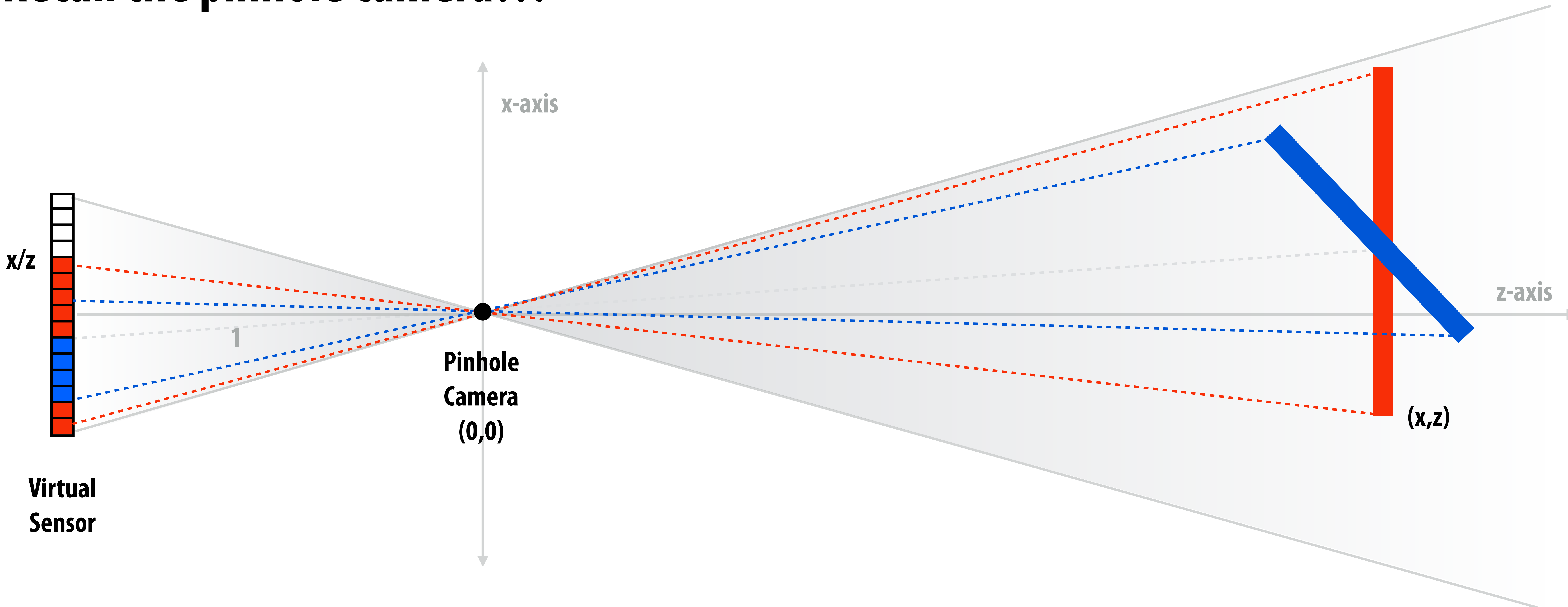
Question 1: what pixels does the triangle overlap? (“coverage”)



Question 2: which triangle is closest to the camera in each pixel? (“occlusion”)

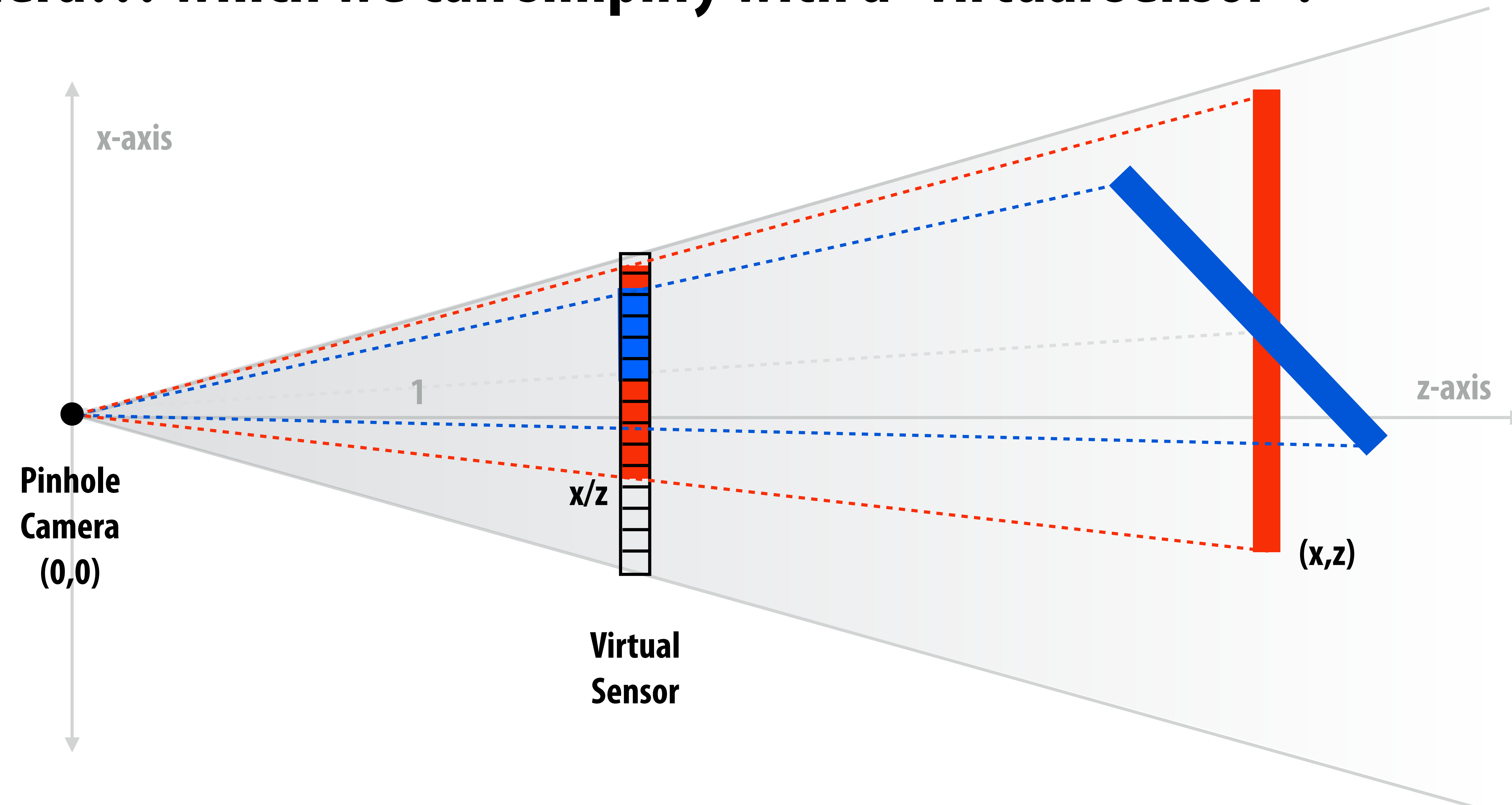
The visibility problem

Recall the pinhole camera…



The visibility problem

Recall the pinhole camera… which we can simplify with a "virtual sensor":

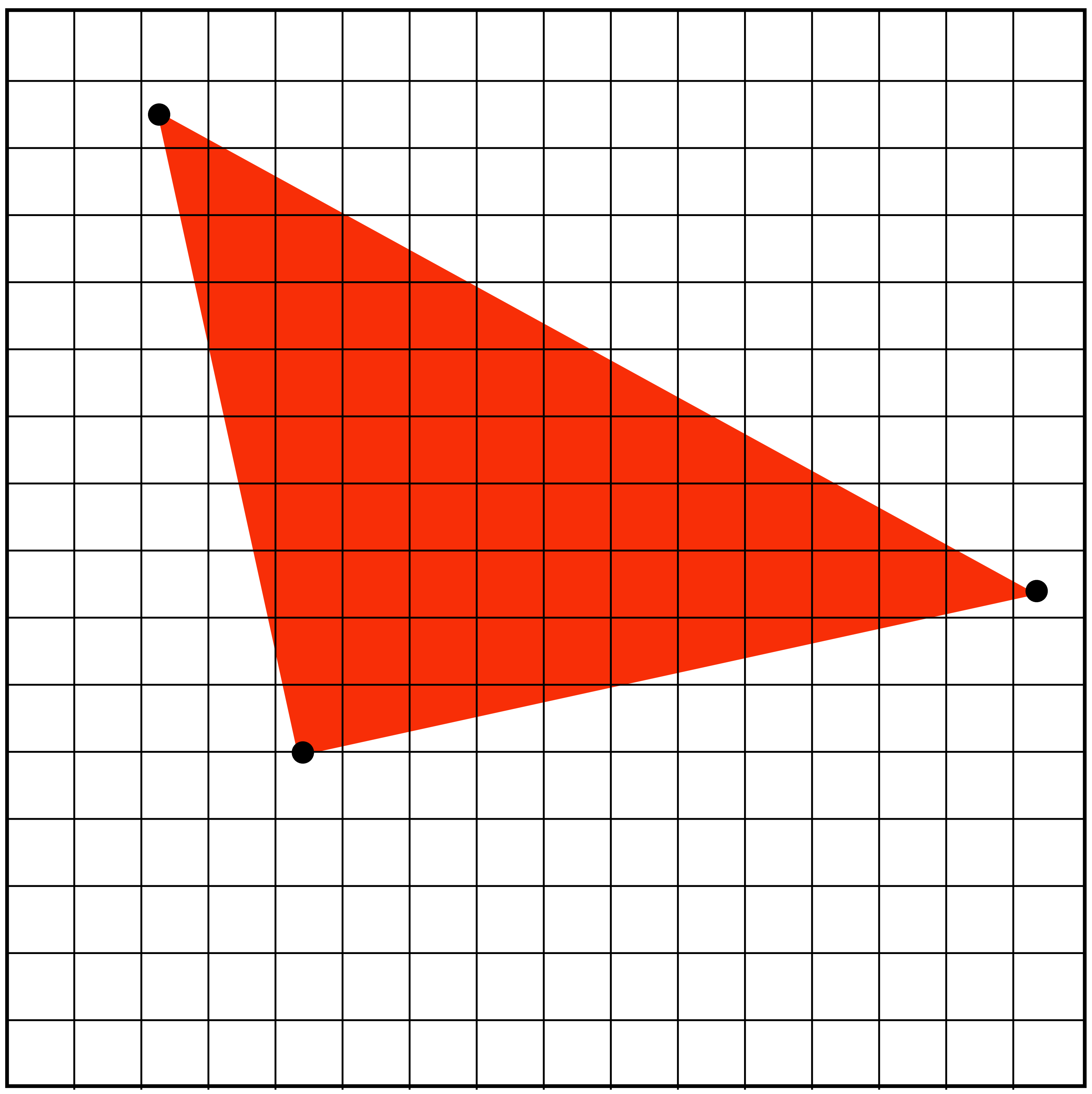


- **Visibility problem in terms of rays:**
  - **COVERAGE: What scene geometry is hit by a ray from a pixel through the pinhole?**
  - **OCCLUSION: Which object is the <u>first</u> hit along that ray?**

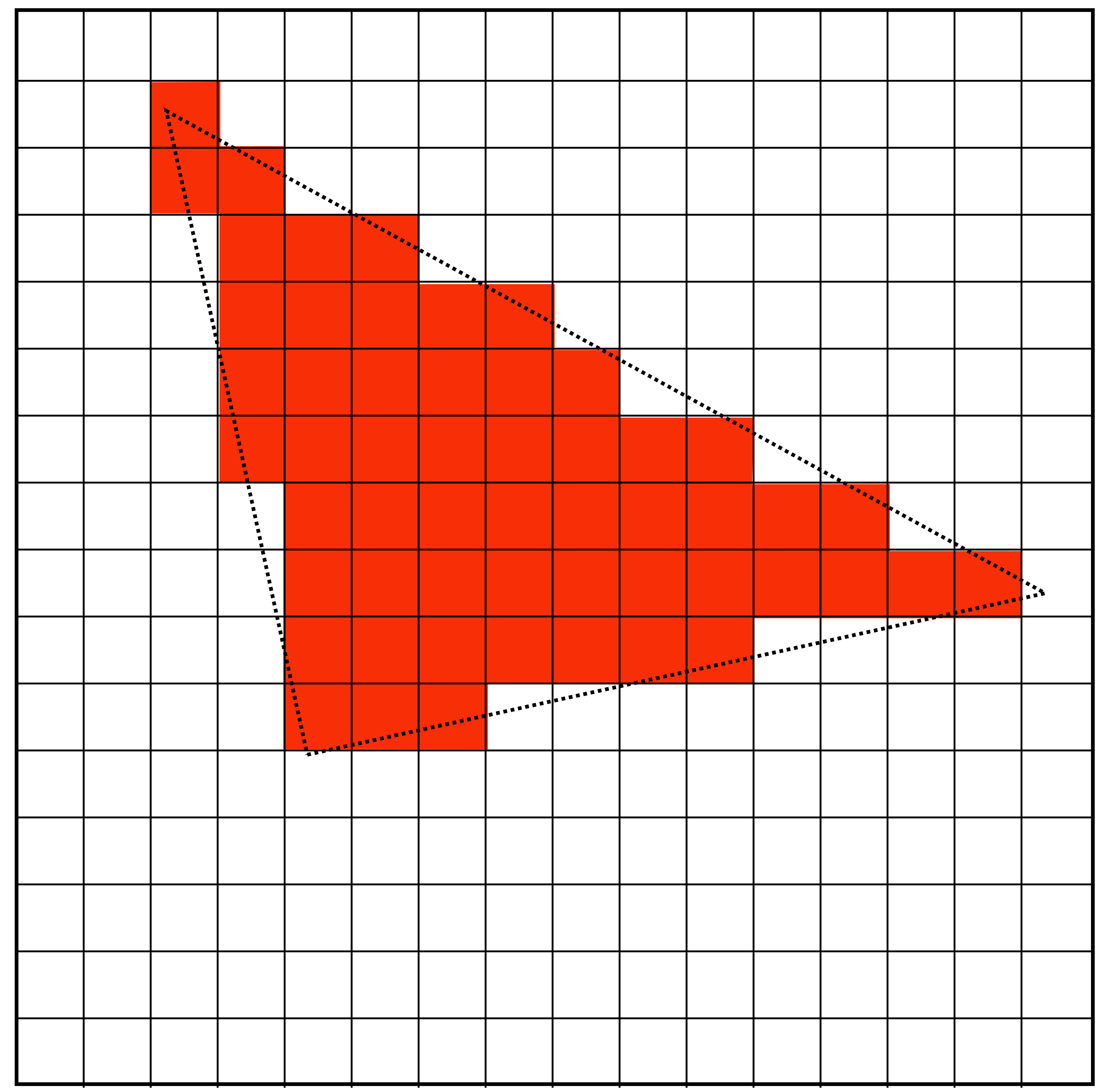
Computing triangle coverage

"Which pixels does the triangle overlap?"

Input:
projected position of triangle vertices: P_0, P_1, P_2

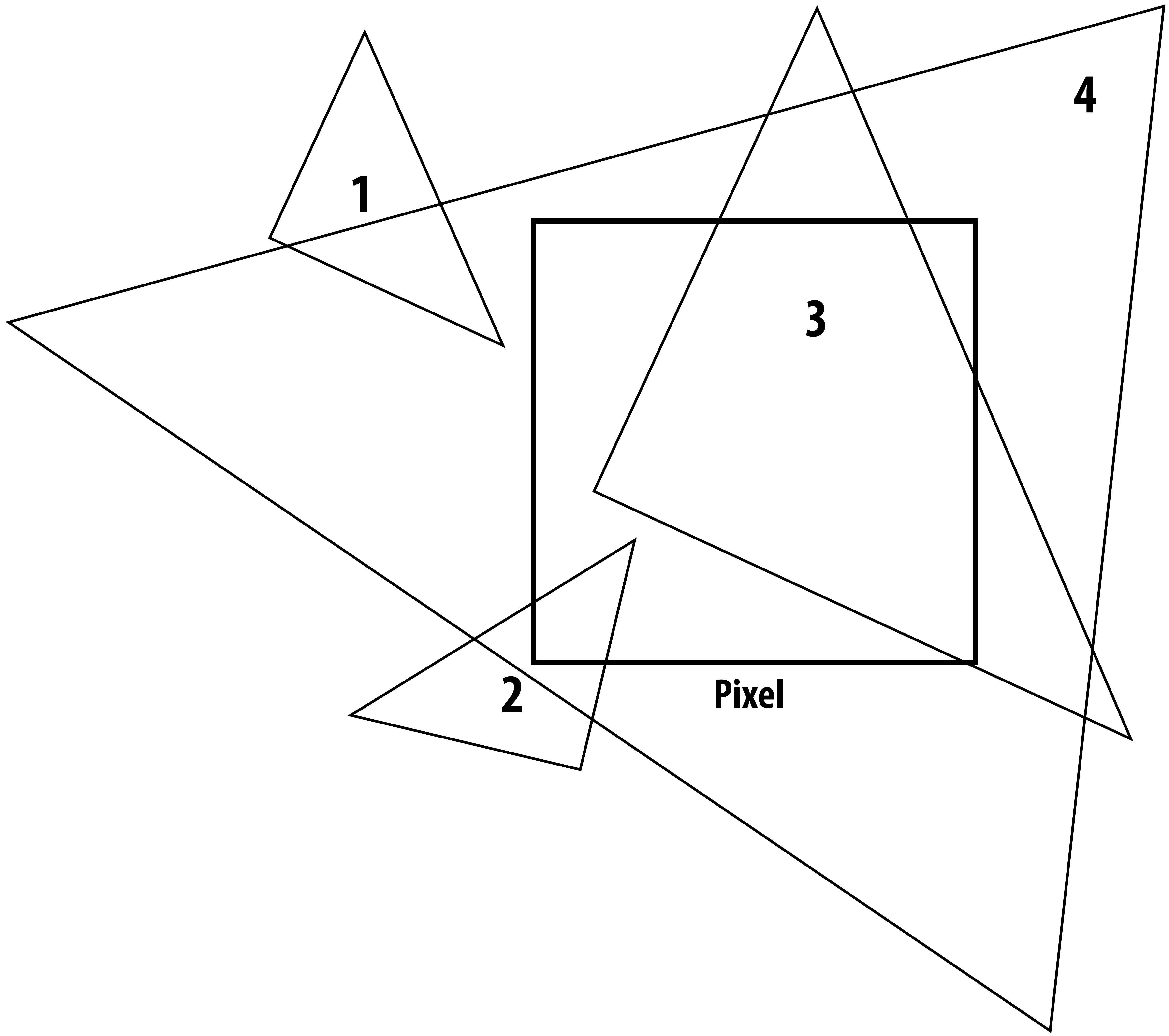


Output:
set of pixels "covered" by the triangle

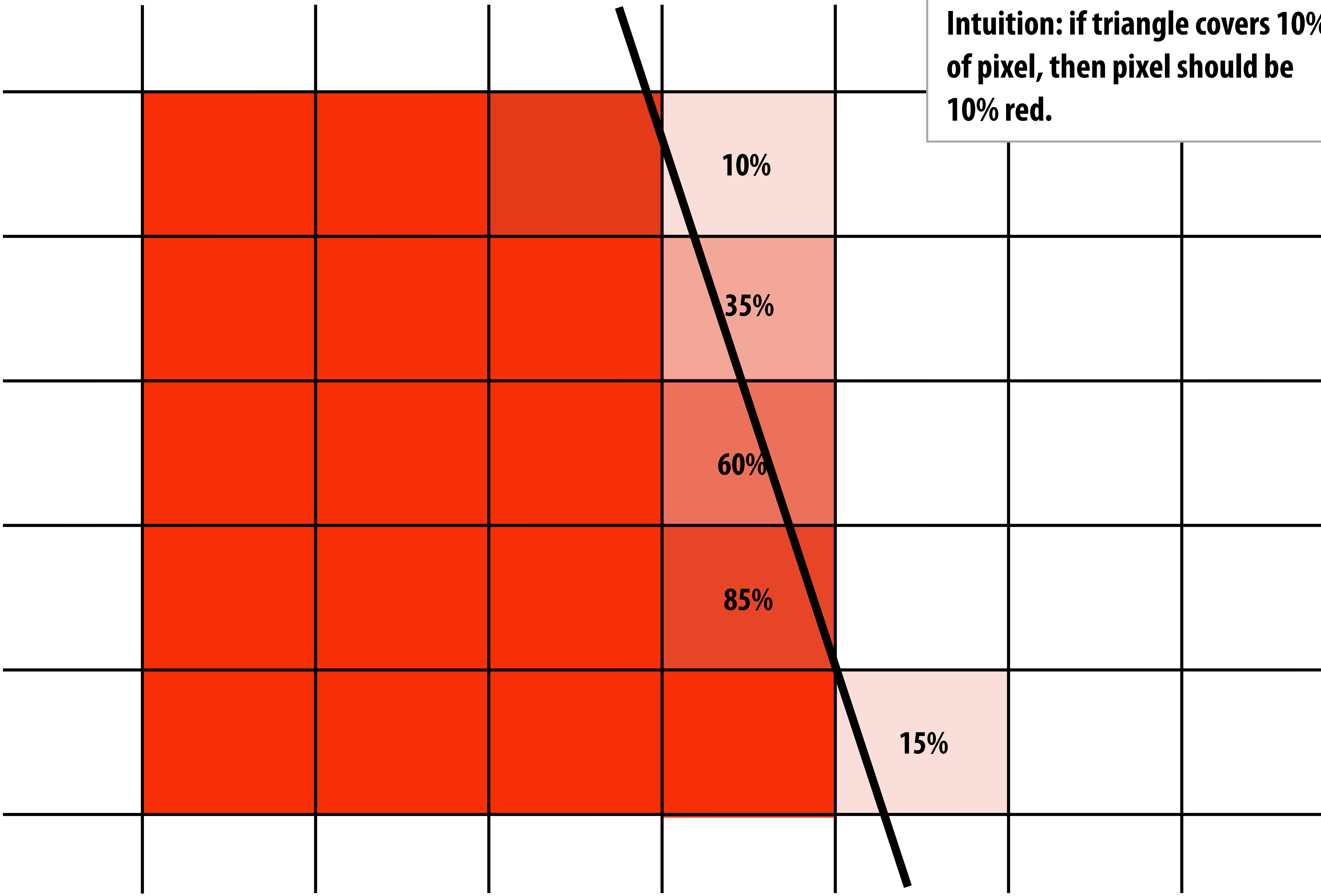


What does it mean for a pixel to be covered by a triangle?

Q: Which triangles "cover" this pixel?

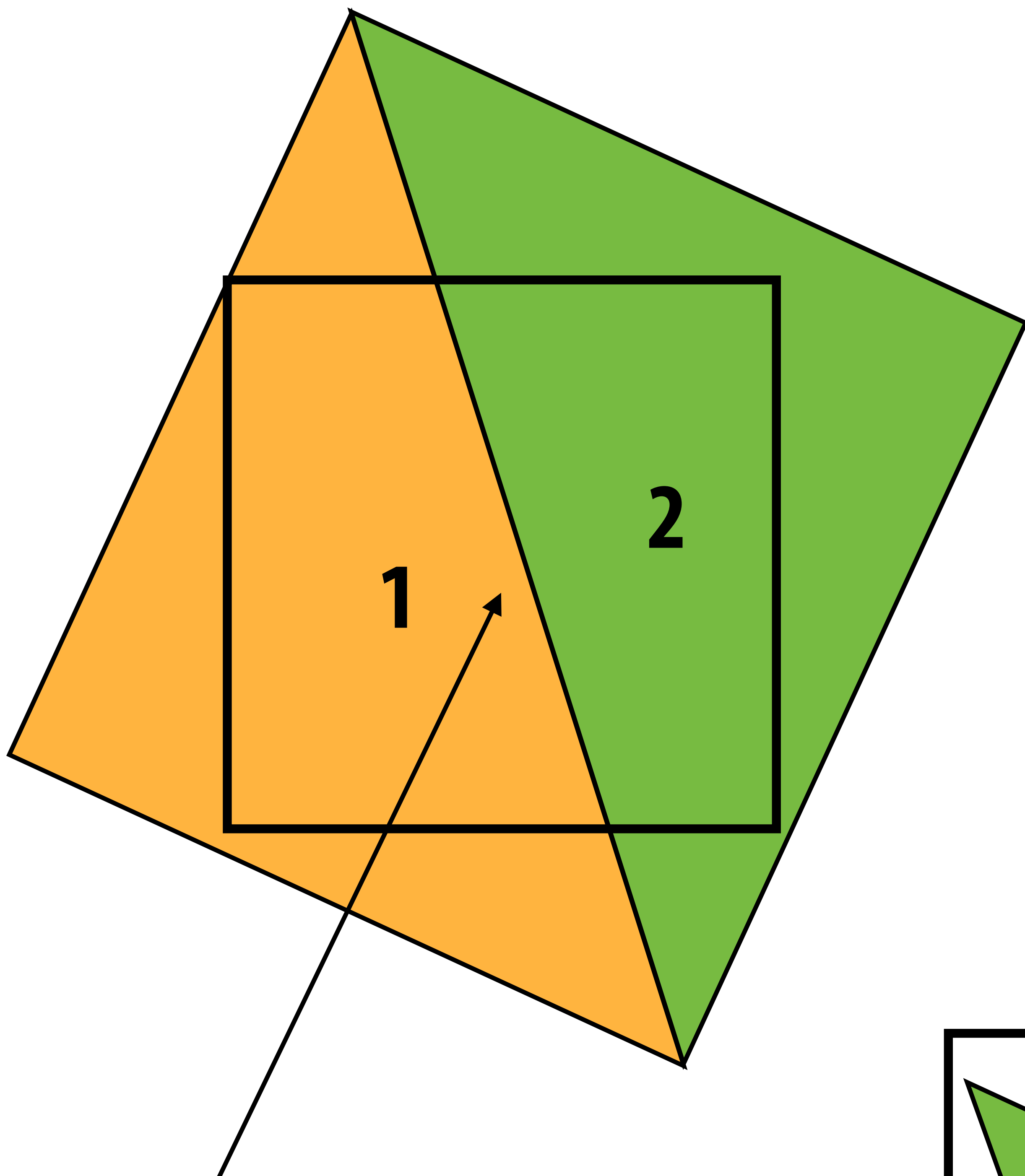


One option: compute fraction of pixel area covered by triangle, then color pixel according to this fraction.

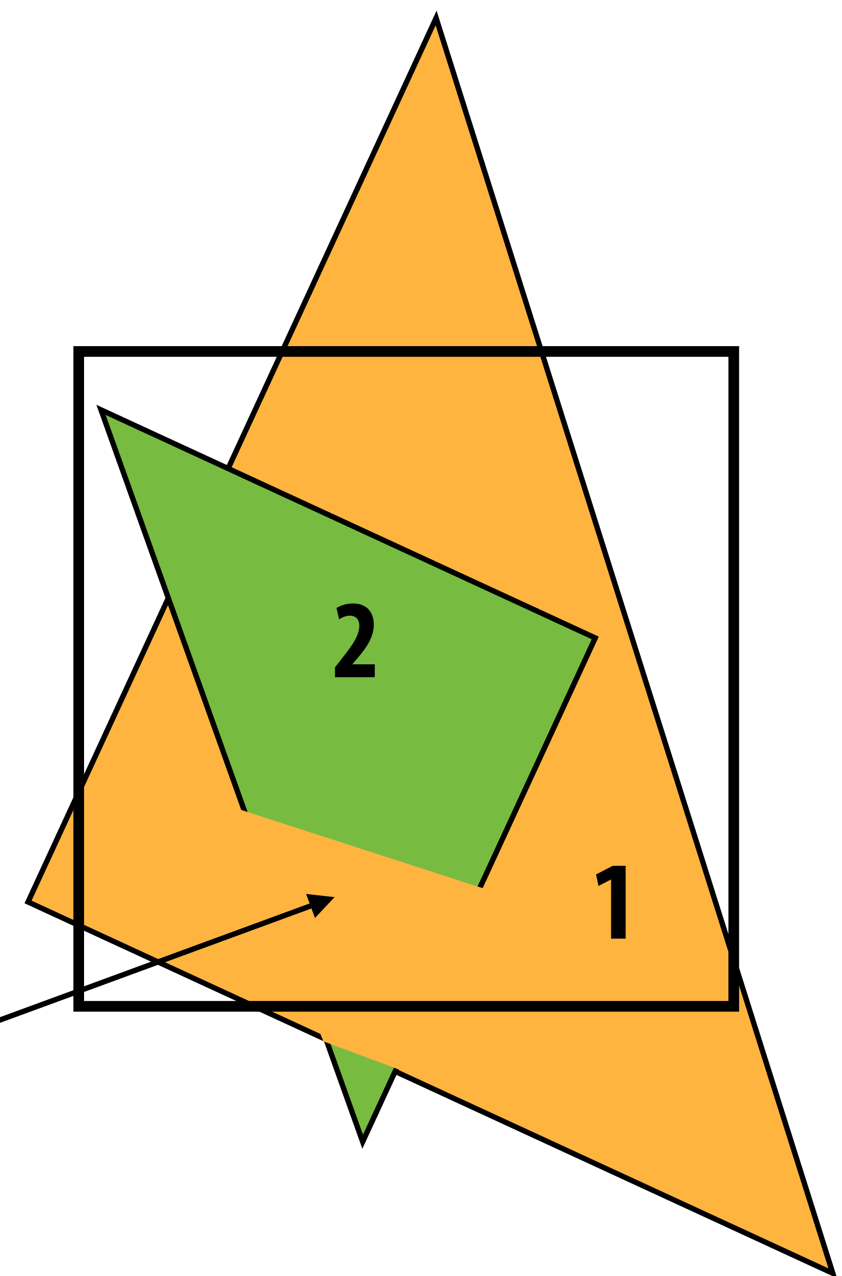


Intuition: if triangle covers 10% of pixel, then pixel should be 10% red.

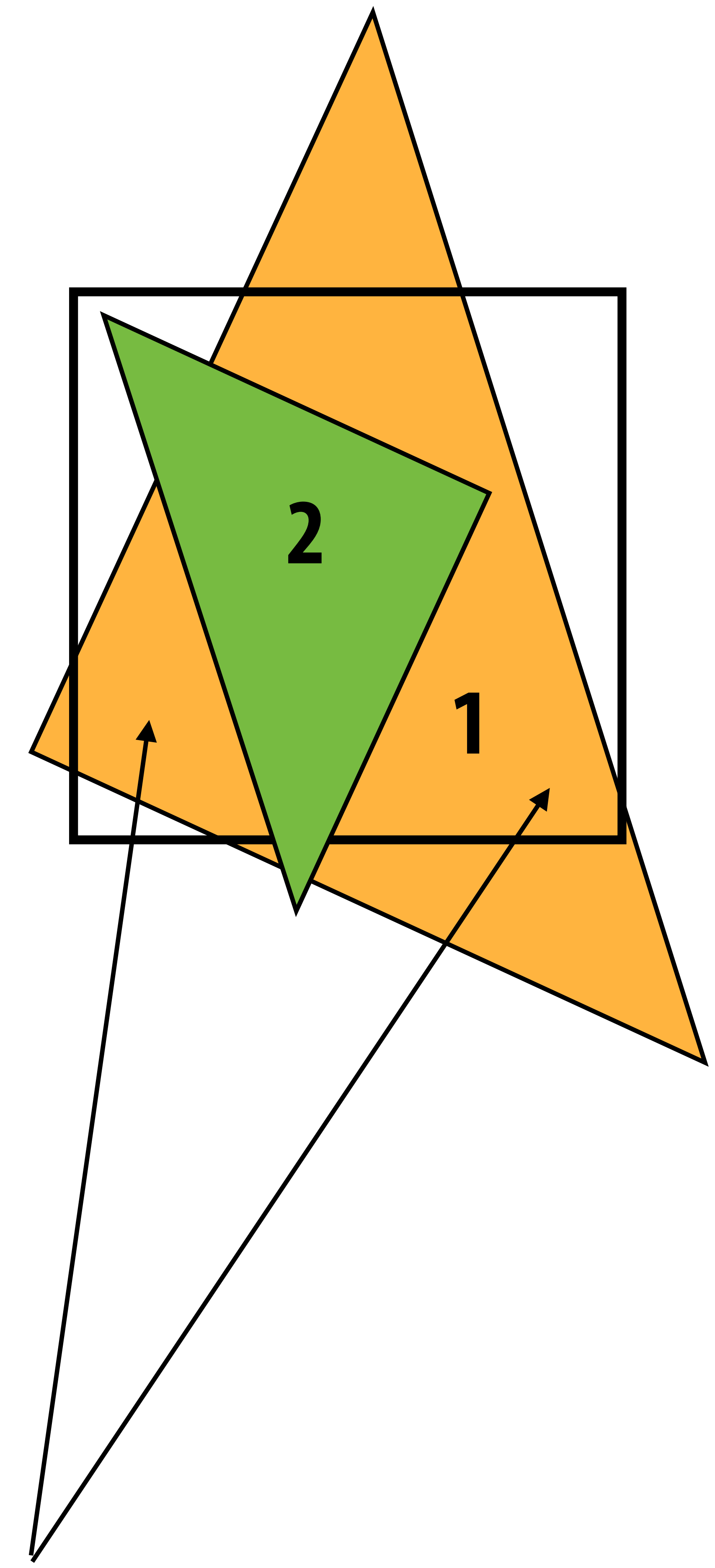
Coverage gets tricky when considering occlusion



Pixel covered by triangle 1, other
half covered by triangle 2



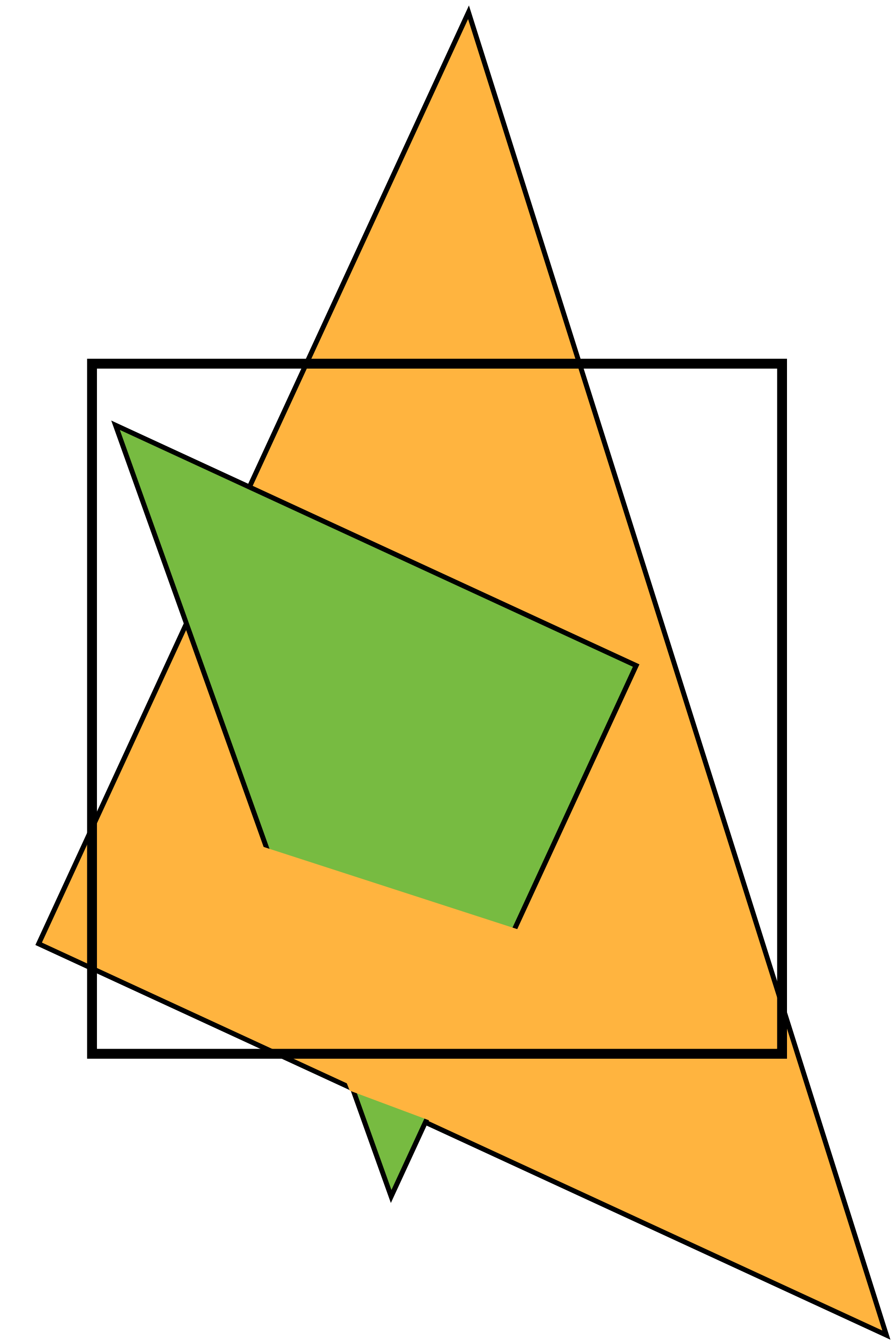
Interpenetration of triangles: even trickier

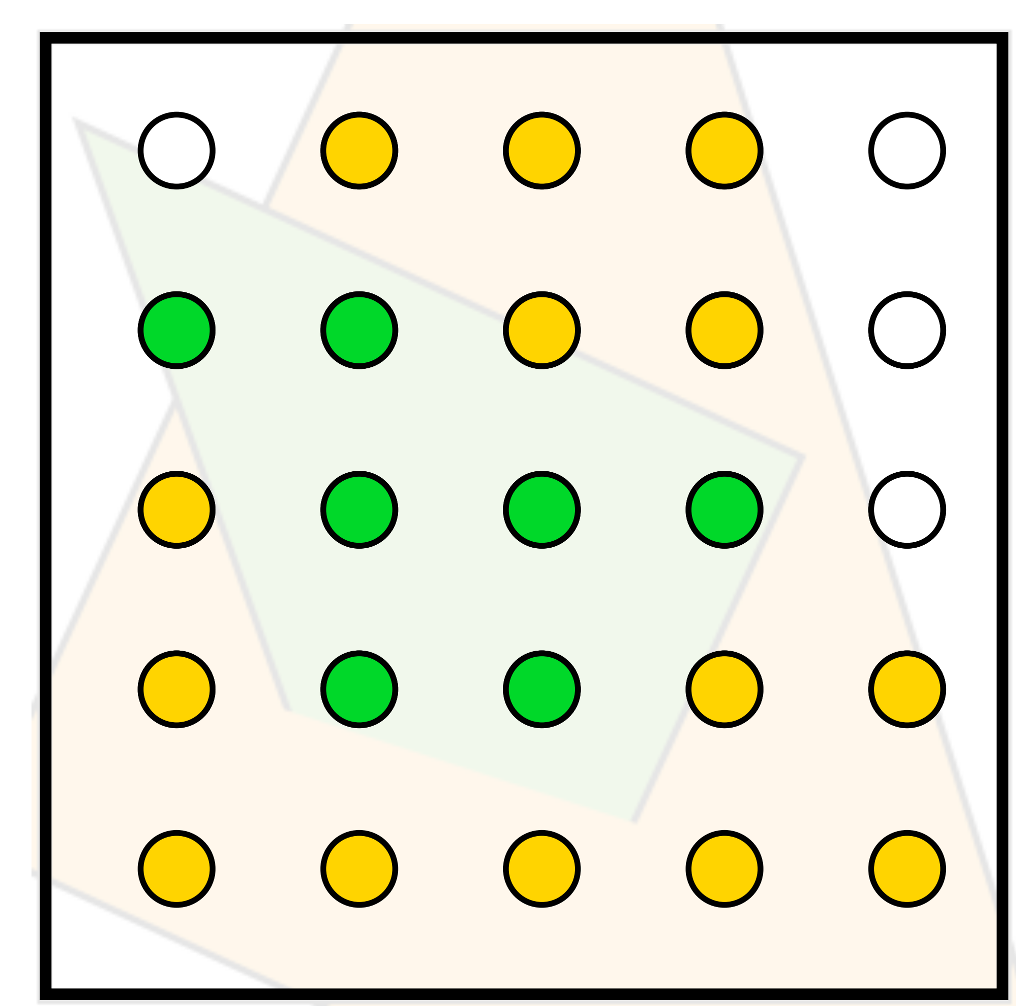


Two regions of triangle 1 contribute to pixel.
One of these regions is not even convex.

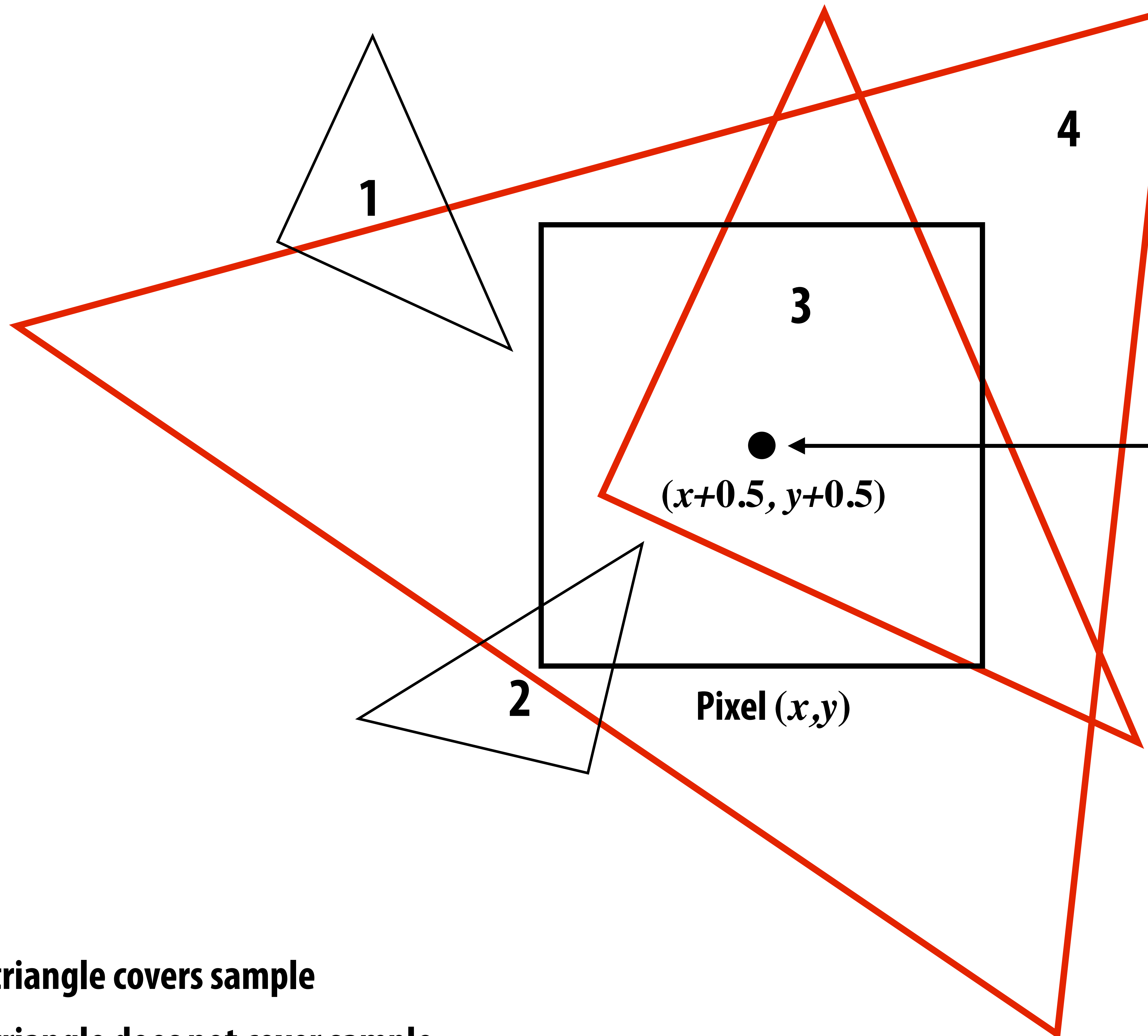
Coverage via sampling

- Real scenes are complicated!
  - occlusion, transparency, …
  - will talk about this more in a future lecture!
- Computing exact coverage is not practical
- Instead: view coverage as a <u>sampling</u> problem
  - don't compute exact/analytical answer
  - instead, test a collection of sample points
  - with enough points & smart choice of sample locations, can start to get a good estimate
- More on this in a week or so ..

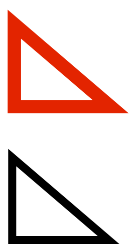




Simple rasterization: just sample the coverage function

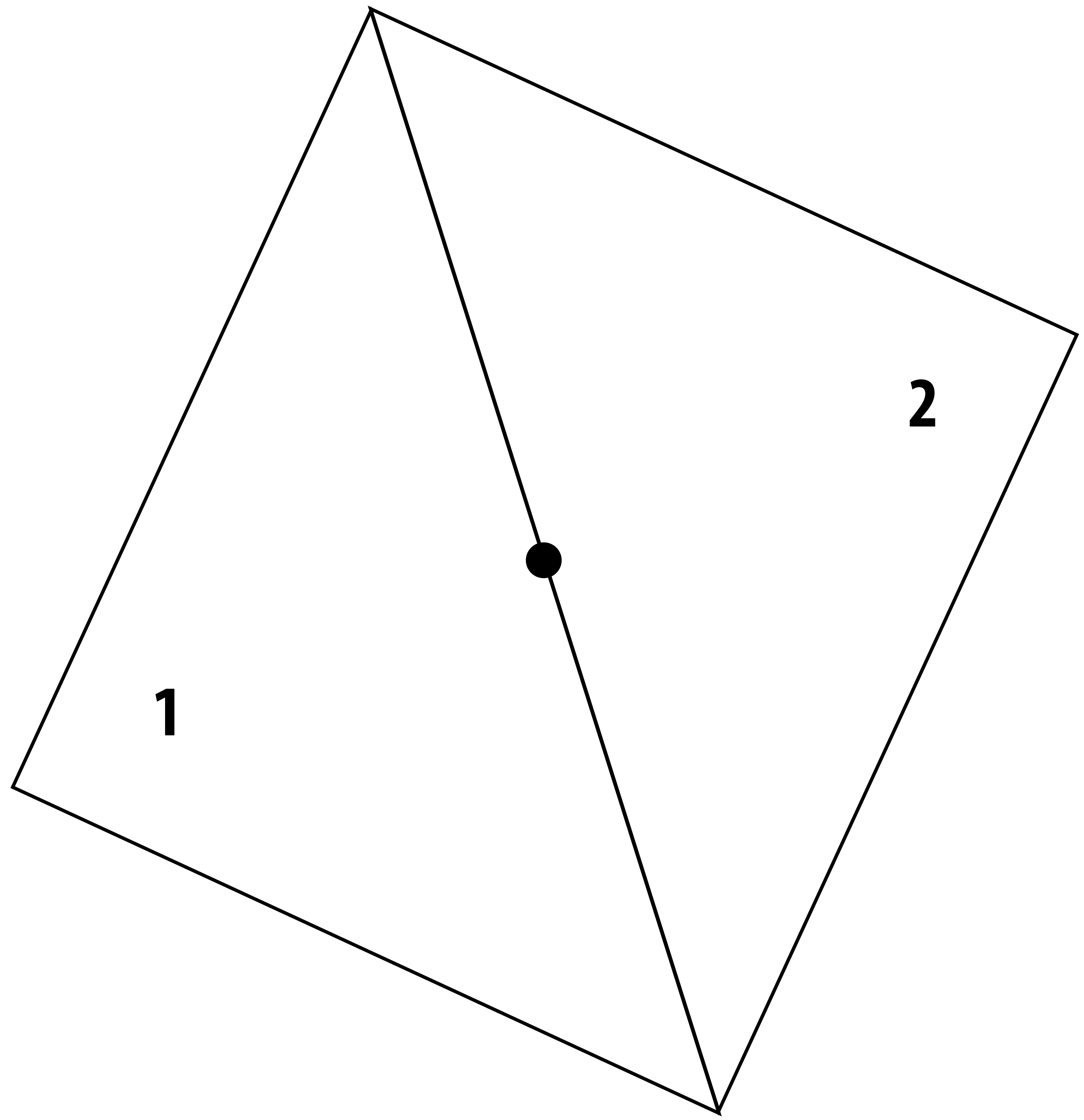


Example:
Here I chose the coverage sample point to be at a point corresponding to the pixel center.



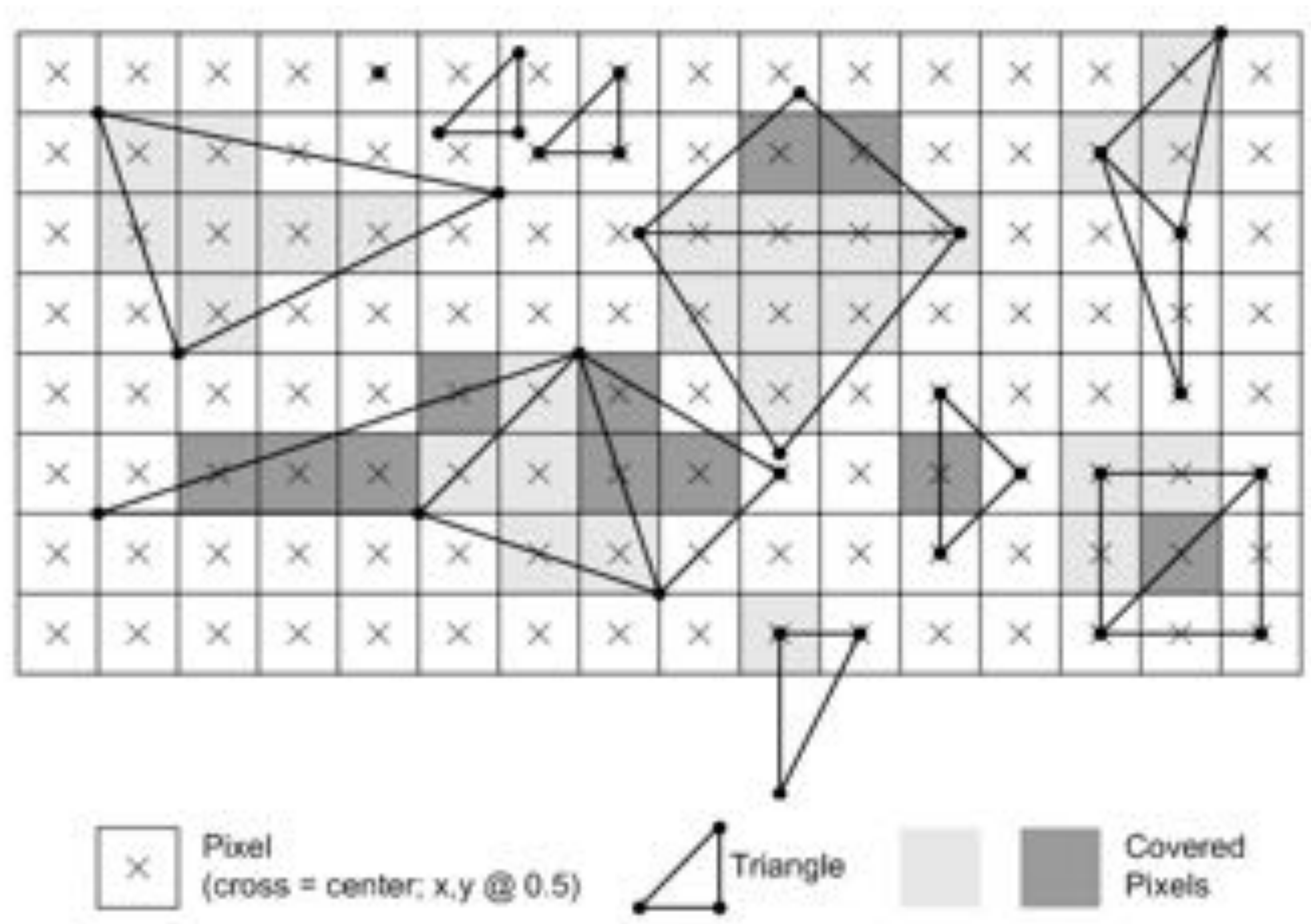
Edge cases (literally)

Is this sample point covered by triangle 1? or triangle 2? or both?



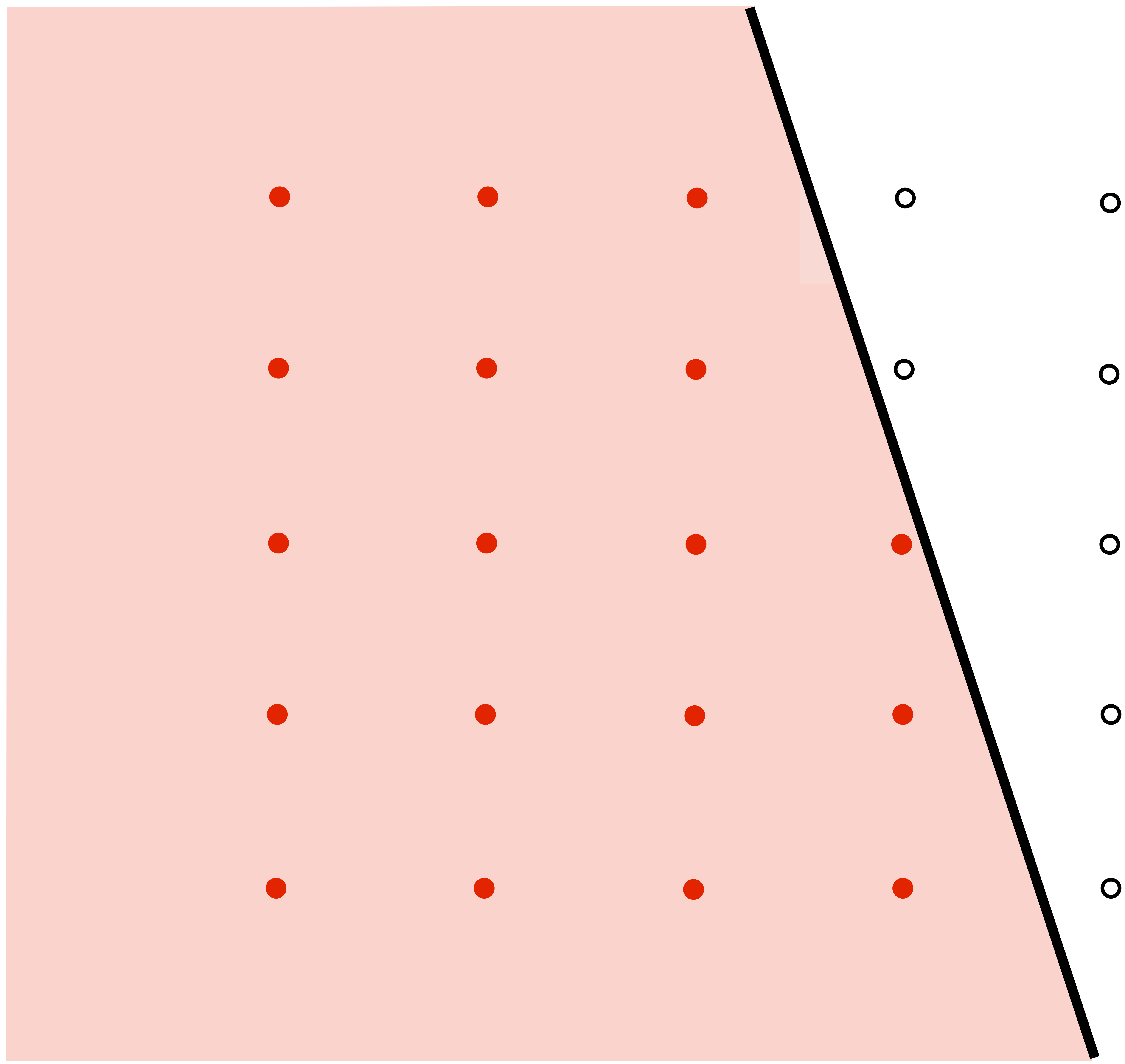
Breaking Ties*

- When edge falls directly on a screen sample point, the sample is classified as within triangle if the edge is a "top edge" or "left edge"
  - Top edge: horizontal edge that is above all other edges
  - Left edge: an edge that is not exactly horizontal and is on the left side of the triangle. (triangle can have one or two left edges)



*These are the rules used in OpenGL/Direct3D, i.e., in modern GPUs. Source: Direct3D Programming Guide, Microsoft

Results of sampling triangle coverage

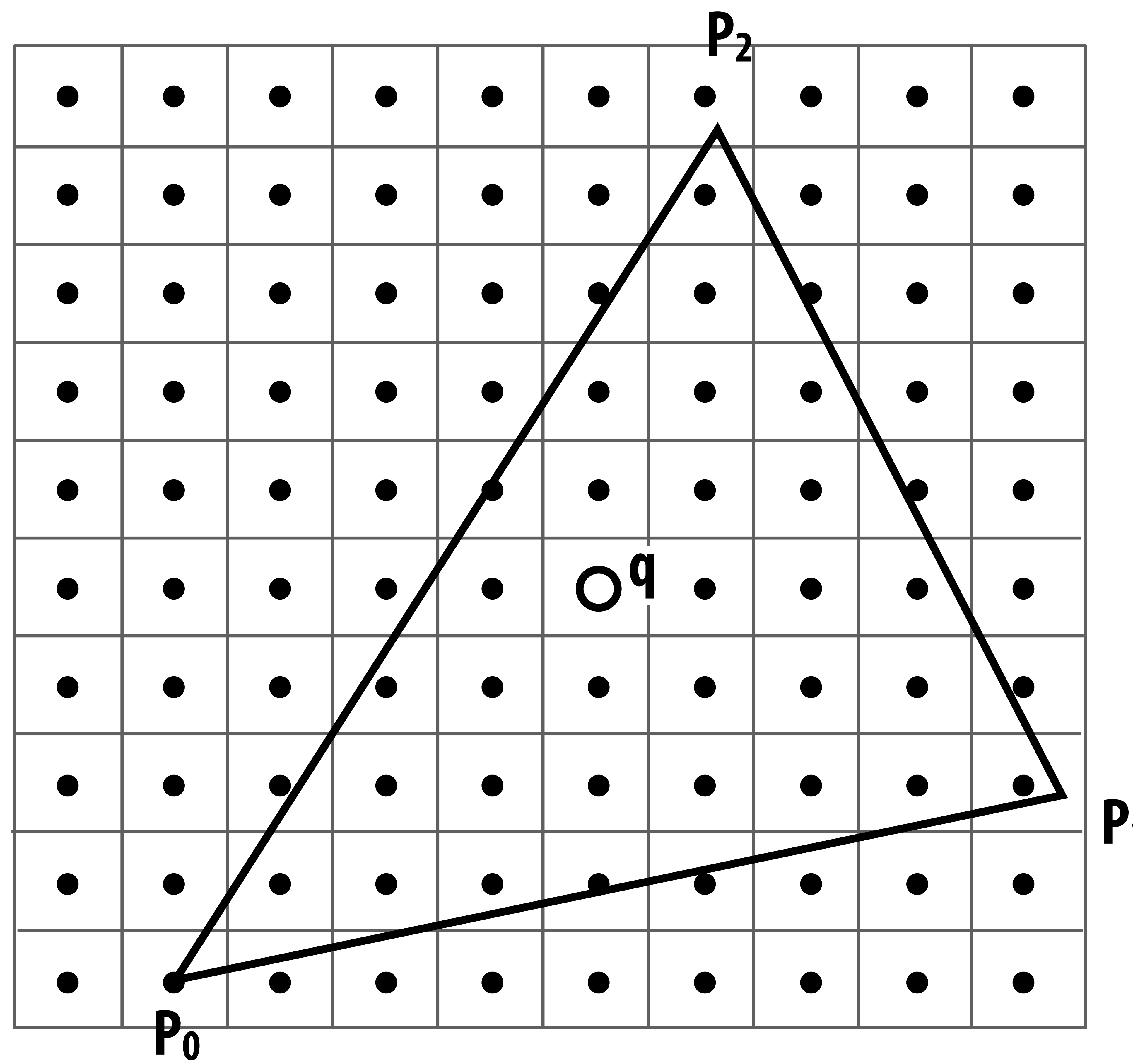


How do we actually evaluate coverage(x,y) for a triangle?

Point-in-triangle test

Q: How do we check if a given point q is inside a triangle?

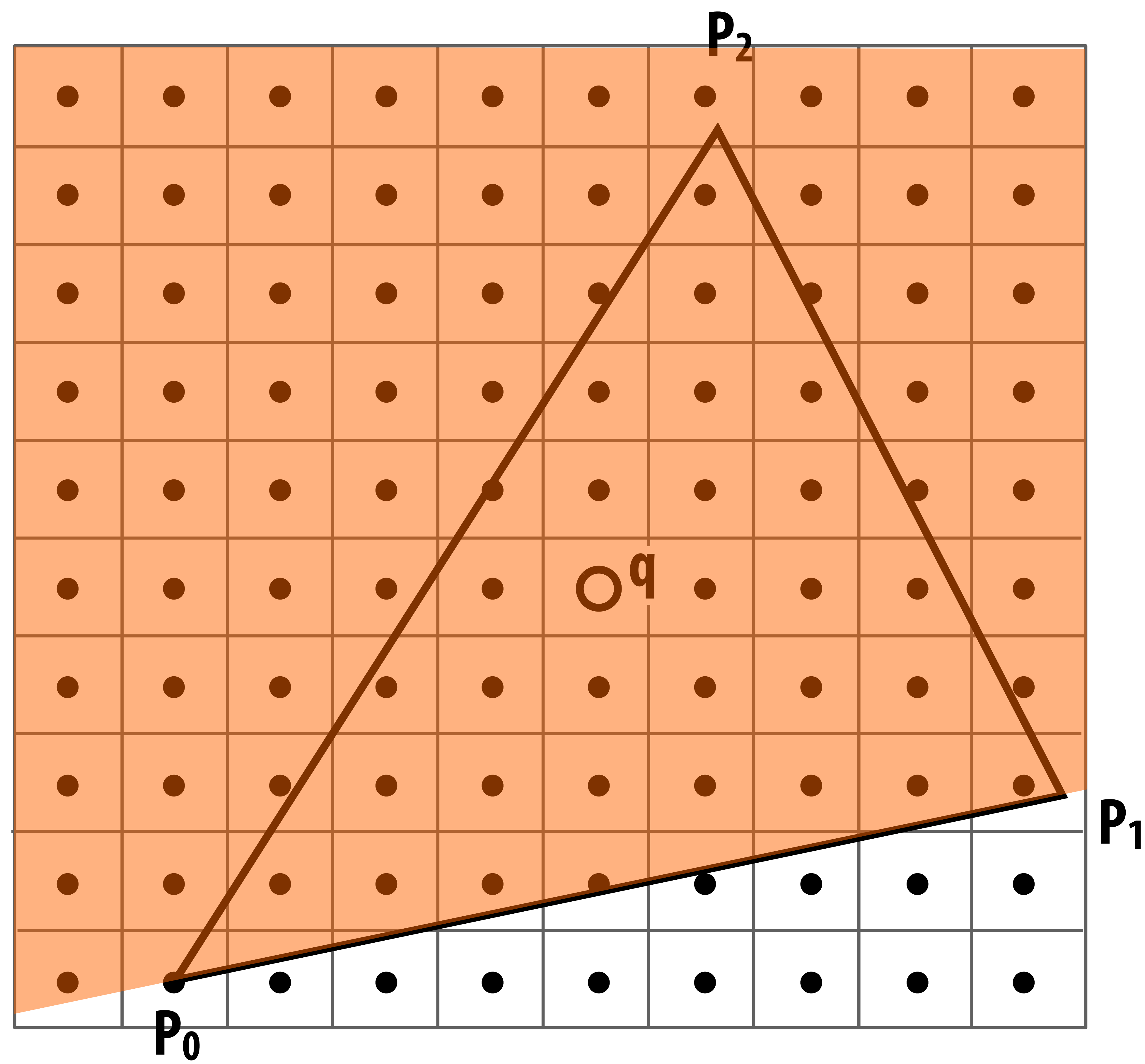
A: Check if it's contained in three <u>half planes</u> associated with the edges.



Point-in-triangle test

Q: How do we check if a given point q is inside a triangle?

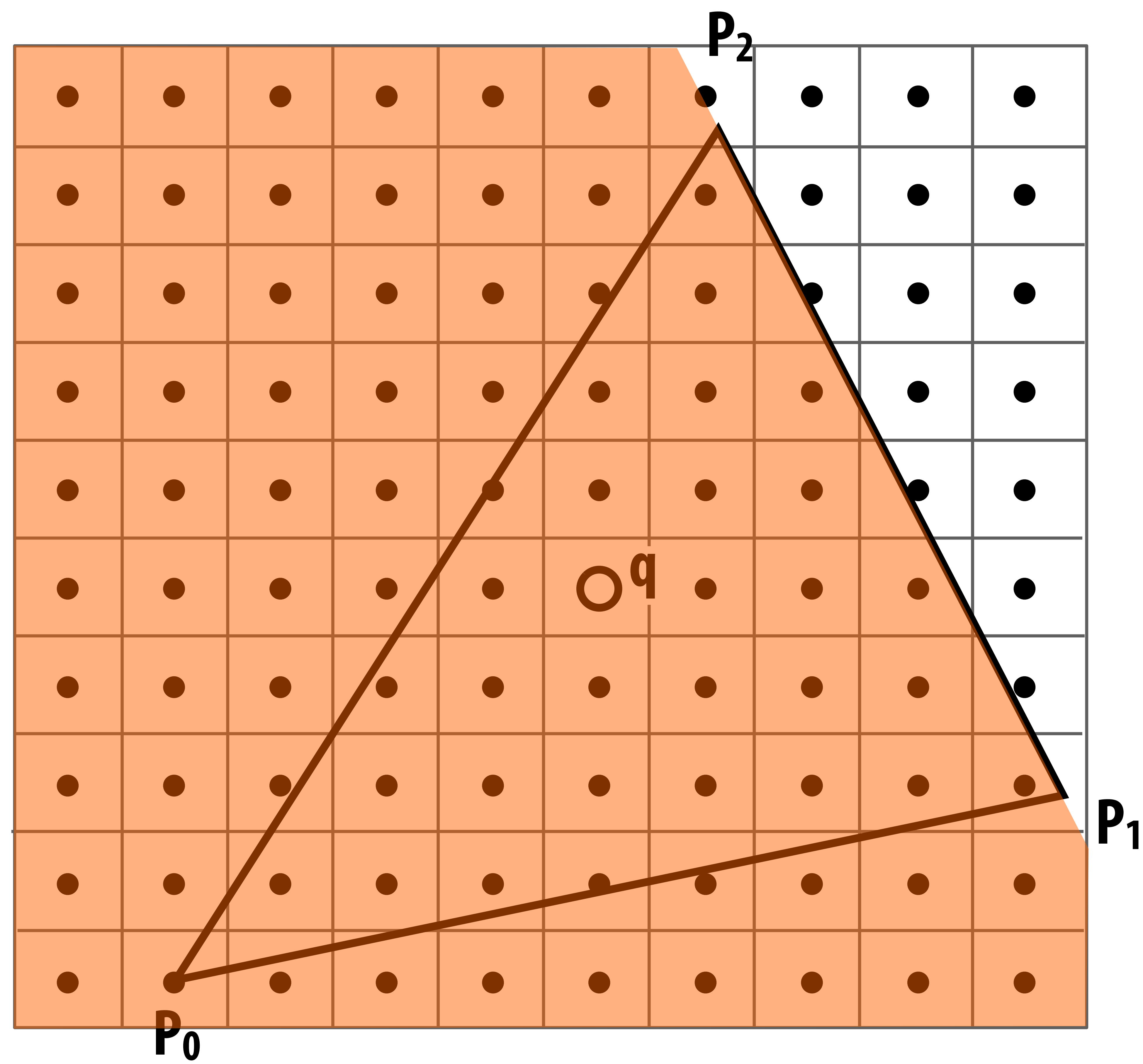
A: Check if it's contained in three <u>half planes</u> associated with the edges.



Point-in-triangle test

Q: How do we check if a given point q is inside a triangle?

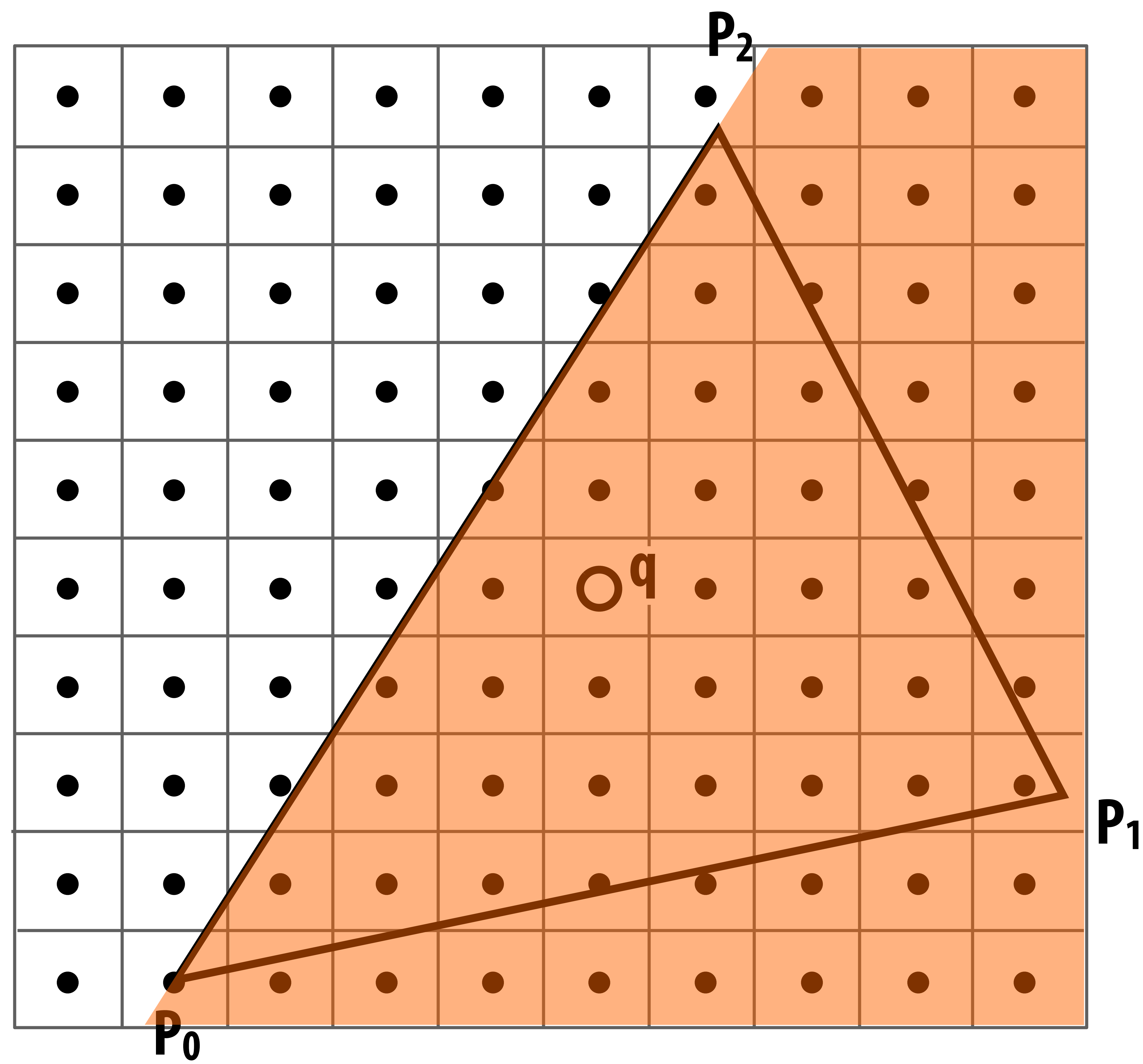
A: Check if it's contained in three <u>half planes</u> associated with the edges.



Point-in-triangle test

Q: How do we check if a given point q is inside a triangle?

A: Check if it's contained in three <u>half planes</u> associated with the edges.

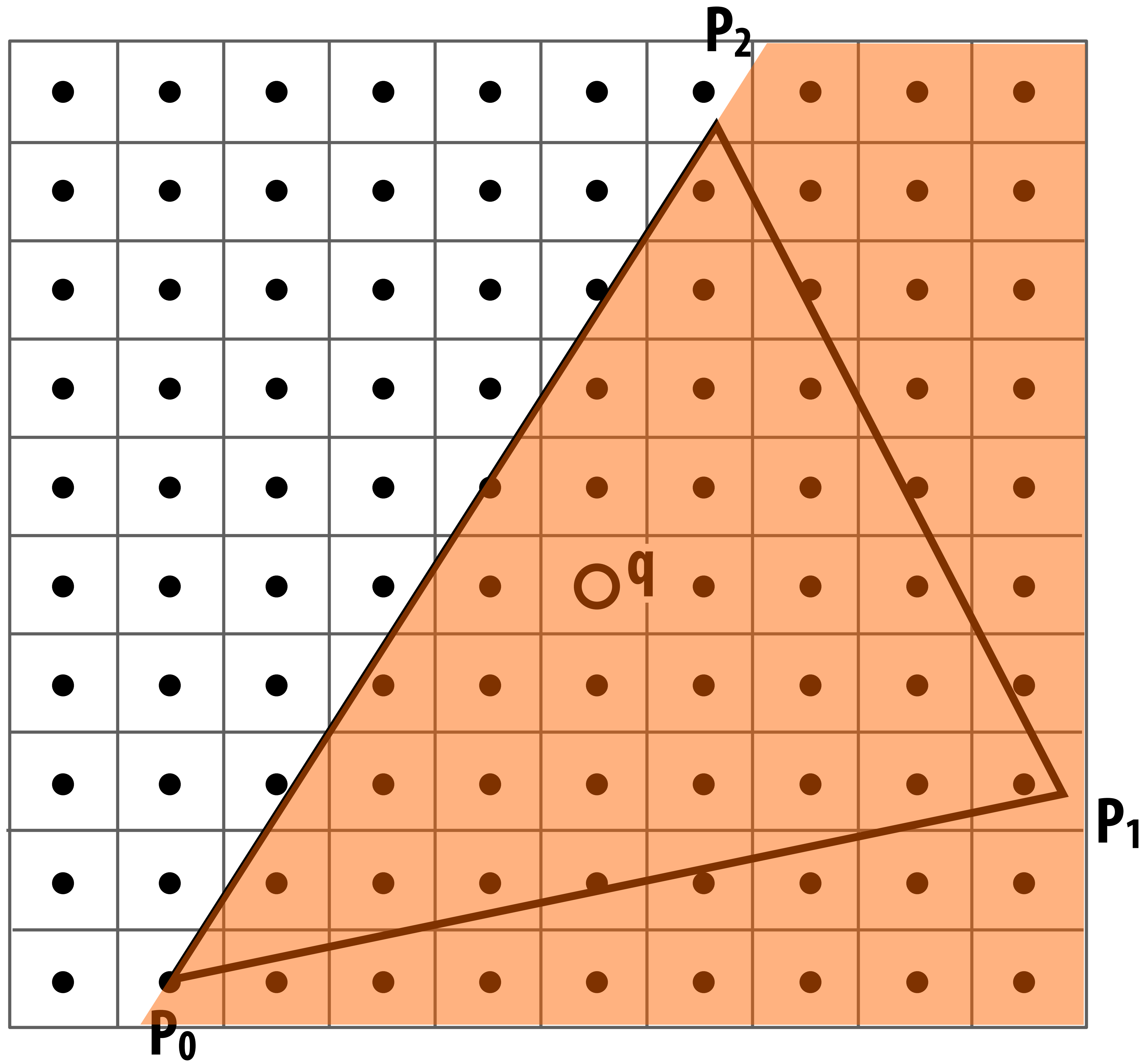


Point-in-triangle test

Q: How do we check if a given point q is inside a triangle?

A: Check if it's contained in three <u>half planes</u> associated with the edges.

Half plane test is then an exercise in linear algebra/ vector calculus:



GIVEN: points P_i, P_j along an edge, and a query point q

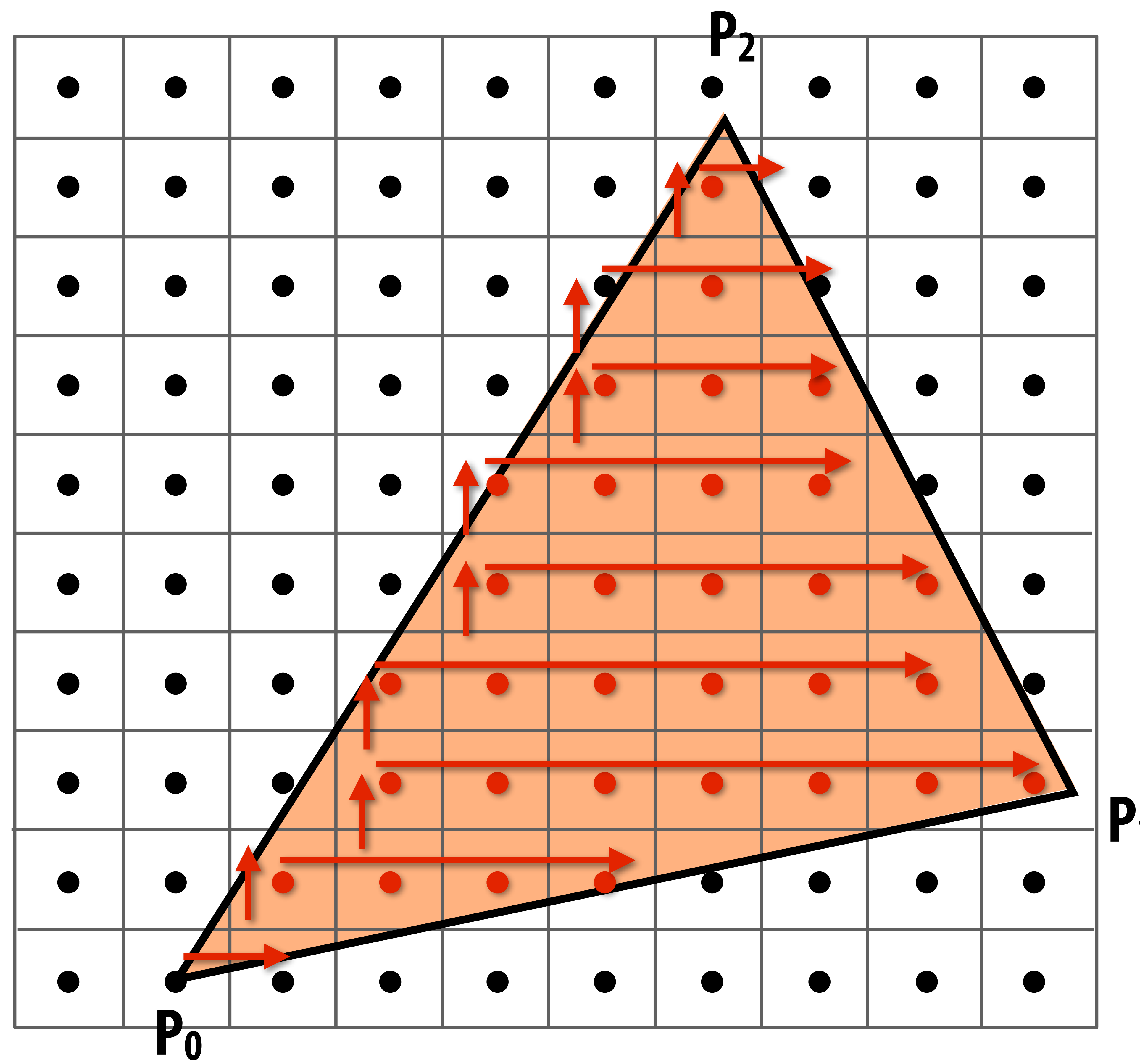
FIND: whether q is to the "left" or "right" of the line from P_i to P_j

(Careful to consider triangle coverage edge rules…)

Traditional approach: incremental traversal

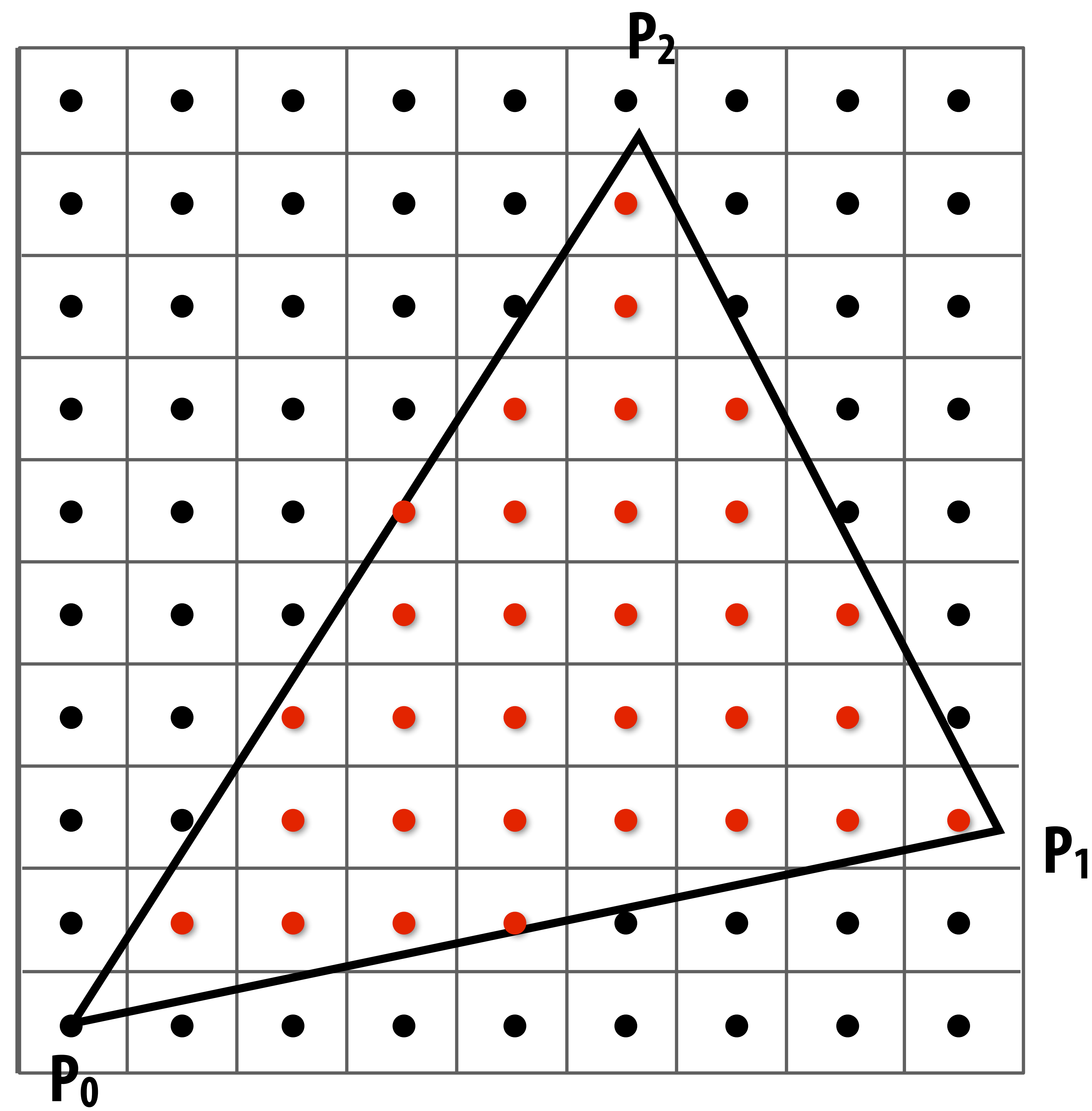
Since half-plane check looks very similar for different points, can save arithmetic by clever "incremental" schemes.

Incremental approach also visits pixels in an order that improves memory coherence: backtrack, zig-zag, Hilbert/Morton curves, …



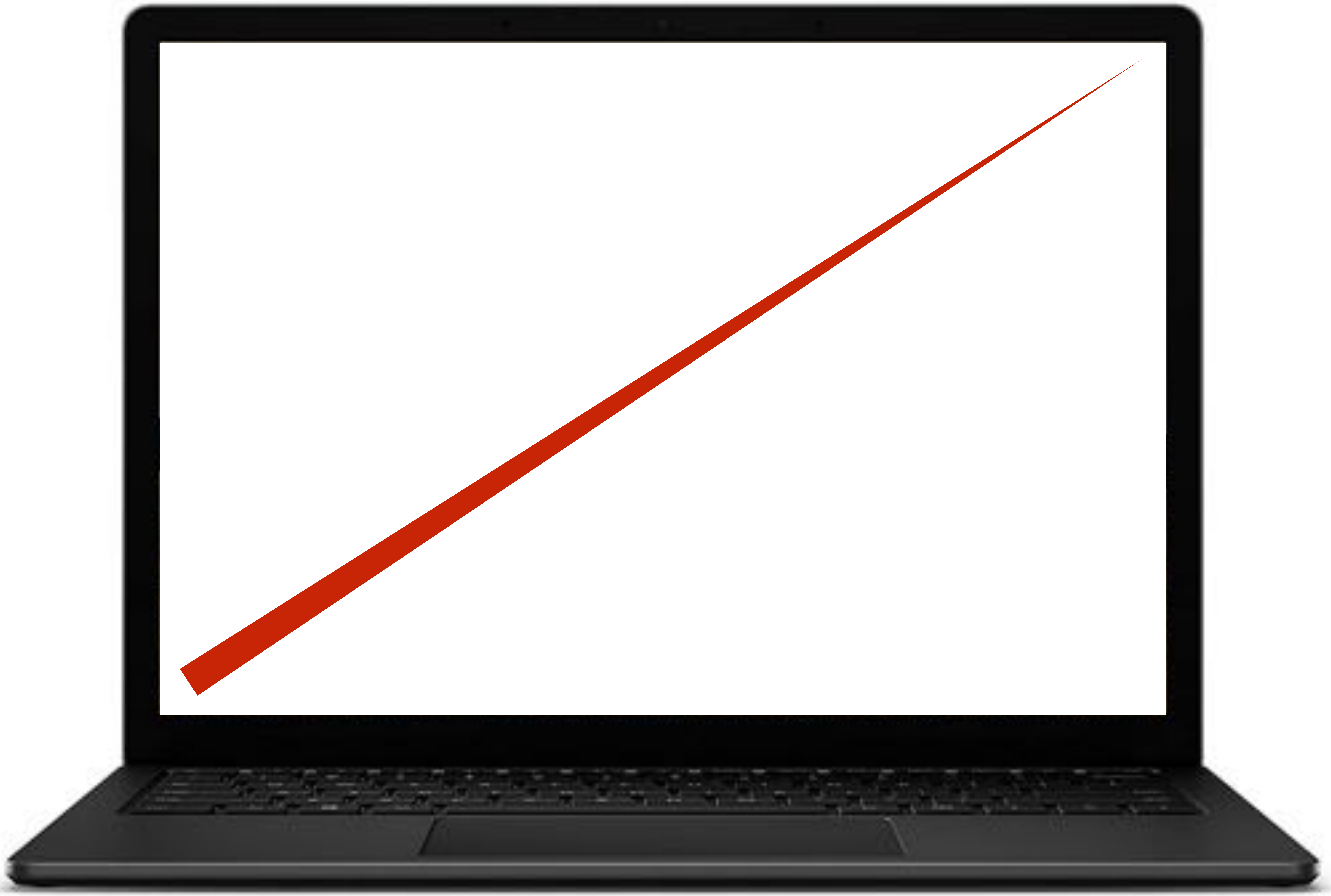
Modern approach: parallel coverage tests

- Incremental traversal is very serial; modern hardware is highly parallel
- Alternative: test all samples in triangle "bounding box" in parallel
- Wide parallel execution overcomes cost of extra tests (most triangles cover many samples, especially when super-sampling)
- All tests share some "setup" calculations
- Modern graphics processing unit (GPU) has special-purpose hardware for efficiently performing point-in-triangle tests



Q: What's a case where the naïve parallel approach is still very inefficient?

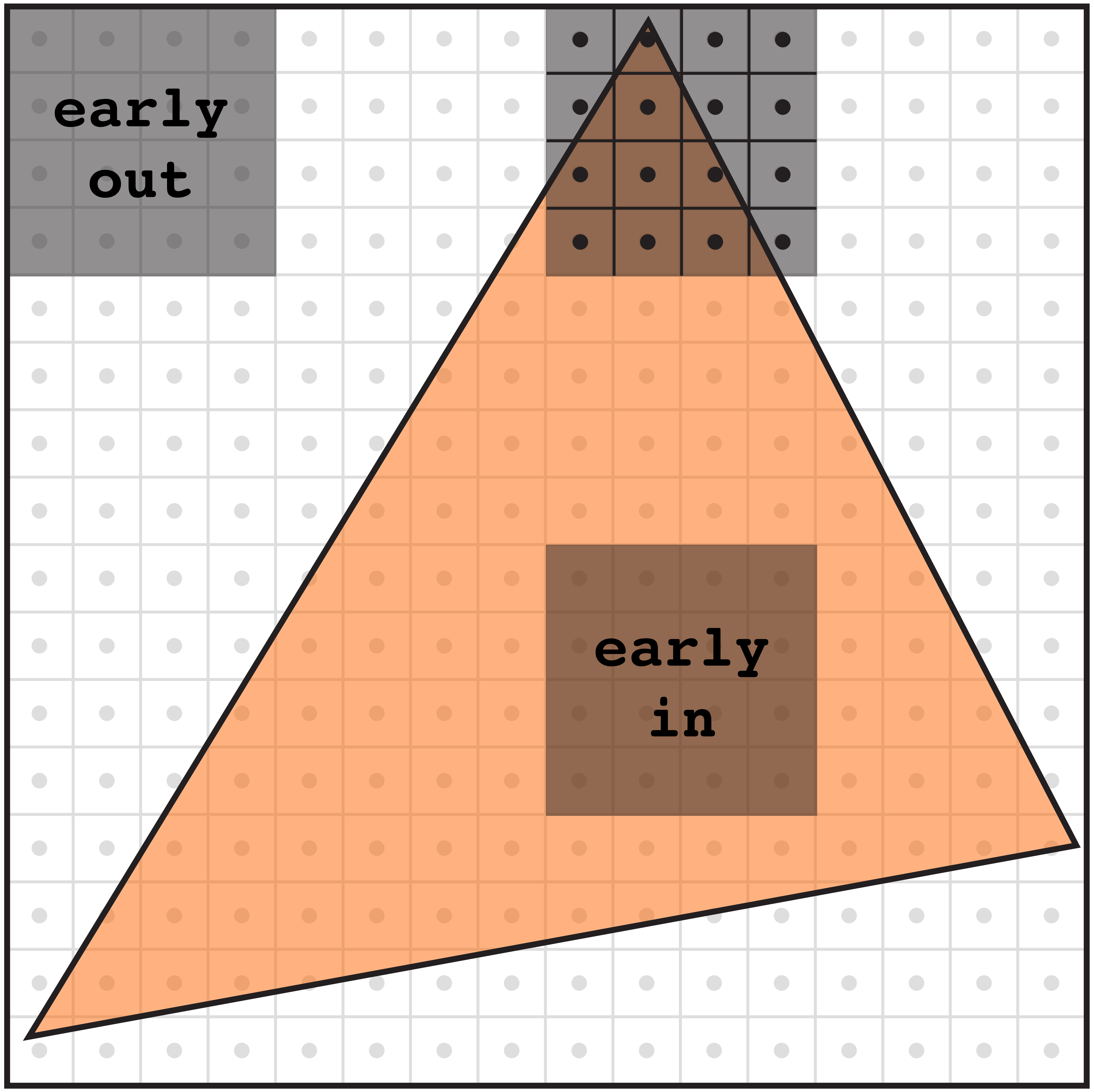
Naïve approach can be (very) wasteful…



Hybrid approach: tiled triangle traversal

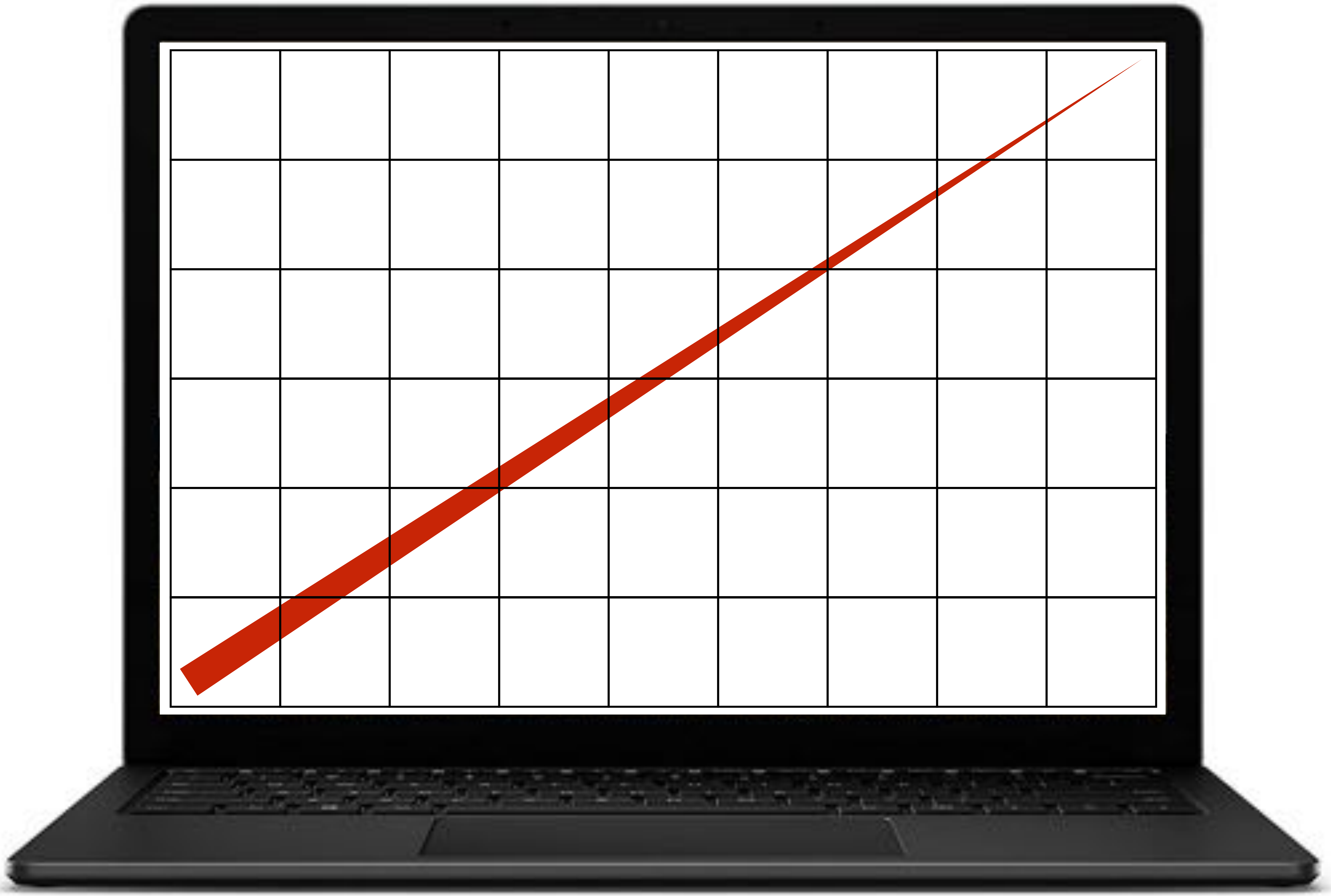
Idea: work "coarse to fine":

- First, check if large blocks intersect the triangle
- If not, skip this block entirely ("early out")
- If the block is contained inside the triangle, know <u>all</u> samples are covered ("early in")
- Otherwise, test individual sample points in the block, in parallel

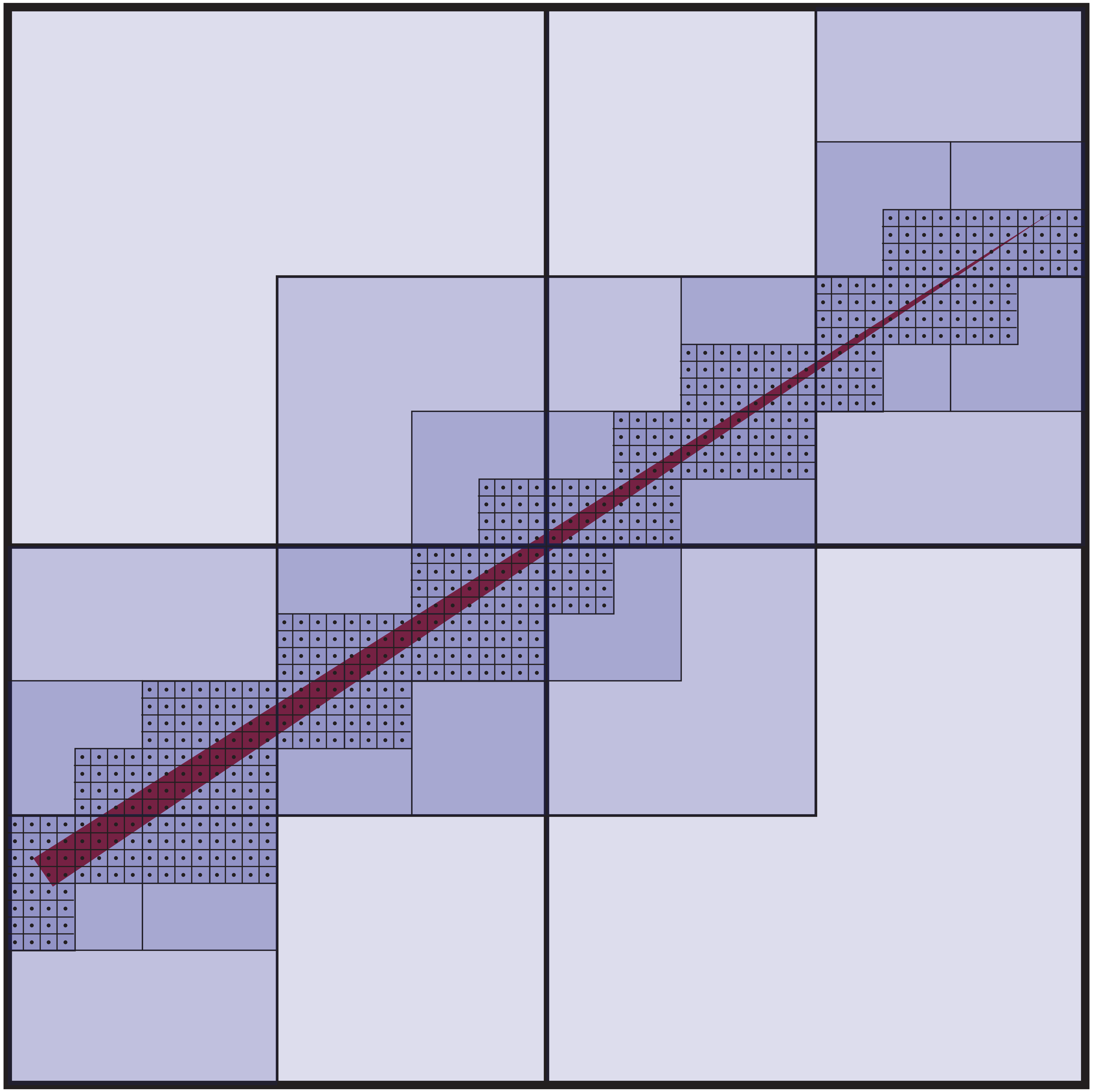


This how real graphics hardware works!

Can we do even better for this example?



Hierarchical strategies in computer graphics



Q: Better way to find finest blocks? **A: Maybe: incremental traversal!**

Summary

- Can frame many graphics problems in terms of sampling and reconstruction
  - sampling: turn a continuous signal into `digital` information
  - reconstruction: turn `digital` information into a continuous signal
- Can frame rasterization as sampling problem
  - sample coverage function into pixel grid
  - reconstruct by emitting a "little square" of light for each pixel
  - aliasing manifests as jagged edges, shimmering artifacts, …
  - we will talk about how to address such artifacts in a later lecture!
- Triangle rasterization is basic building block for graphics pipeline
  - amounts to three half-plane tests
  - atomic operation—make it fast!
  - several strategies: incremental, parallel, blockwise, hierarchical…

Next Time: Depth & Transparency