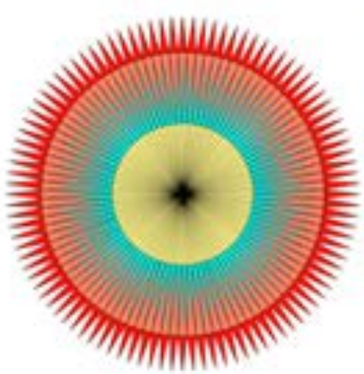
Lecture 4:

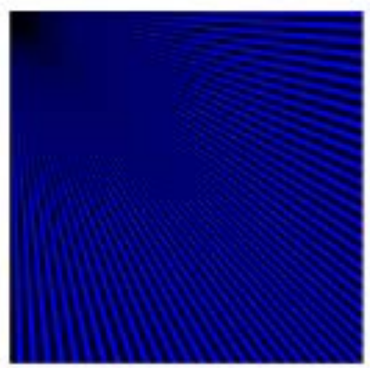
Drawing a Triangle (and an Intro to Sampling)

Computer Graphics
CMU 15-462/15-662

HW 0.5 due, HW 1 out soon!

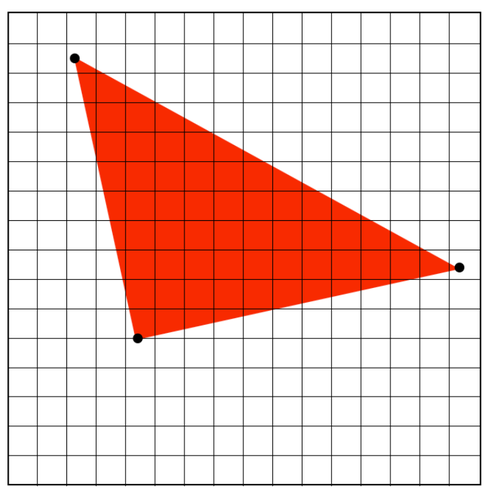
- GOAL: Implement a basic "rasterizer"
  - (Topic of today's lecture)
  - We hand you a bunch of lines, triangles, etc.
  - You draw them by lighting up pixels on the screen!
- Code skeleton available (later today) from course webpage
- <u>Final code must build on Linux!</u>
- DUE FEBRUARY 17 (~2 weeks)

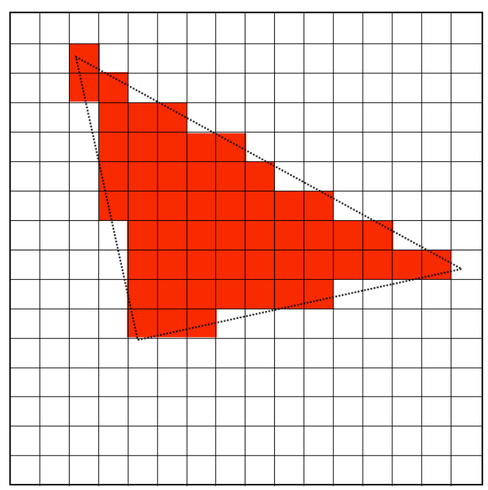


TODAY: Rasterization

- 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!

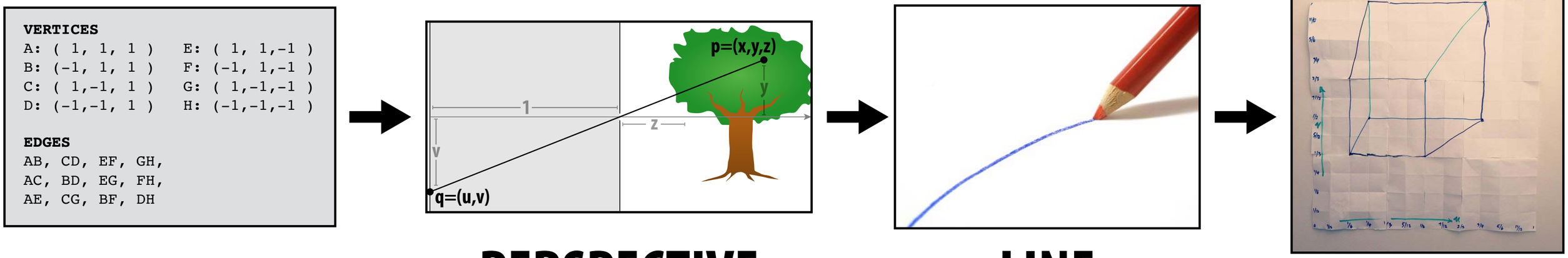




3D Image Generation Pipeline(s)

- Can talk about image generation in terms of a "pipeline":
  - INPUTS — what image do we want to draw?
  - STAGES — sequence of transformations from input → output
  - OUTPUTS — the final image

E.g., our pipeline from the first lecture:



INPUT

PERSPECTIVE PROJECTION STAGE

LINE DRAWING STAGE

OUTPUT

Rasterization Pipeline

- Modern real time image generation based on *rasterization*
  - INPUT: 3D "primitives"—essentially all triangles!
    - possibly with additional attributes (e.g., color)
  - OUTPUT: bitmap image (possibly w/ depth, alpha, . . .)
- Our goal: understand the stages in between*

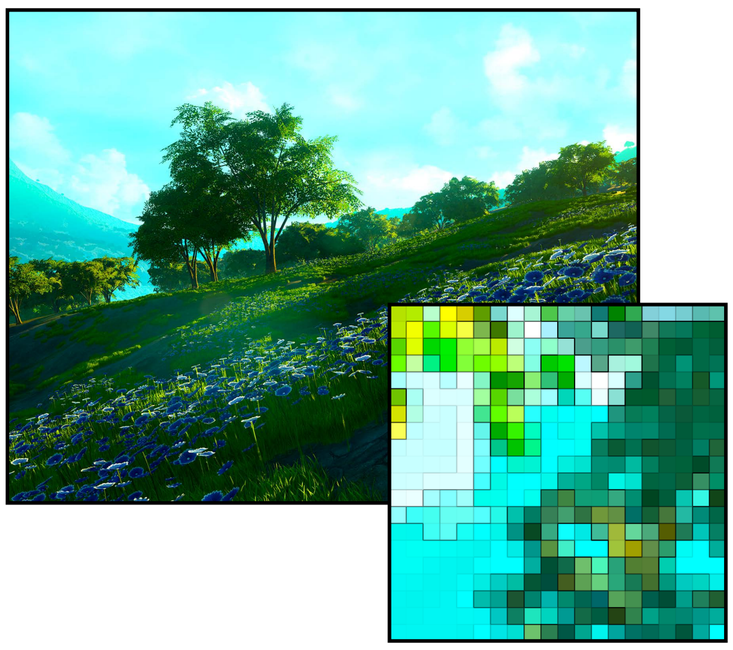
INPUT (TRIANGLES)

```
VERTICES
A: ( 1, 1, 1 )   E: ( 1, 1,-1 )
B: (-1, 1, 1 )   F: (-1, 1,-1 )
C: ( 1,-1, 1 )   G: ( 1,-1,-1 )
D: (-1,-1, 1 )   H: (-1,-1,-1 )

TRIANGLES
EHF, GFH, FGB, CBG,
GHC, DCH, ABD, CDB,
HED, ADE, EFA, BAF
```

RASTERIZATION PIPELINE

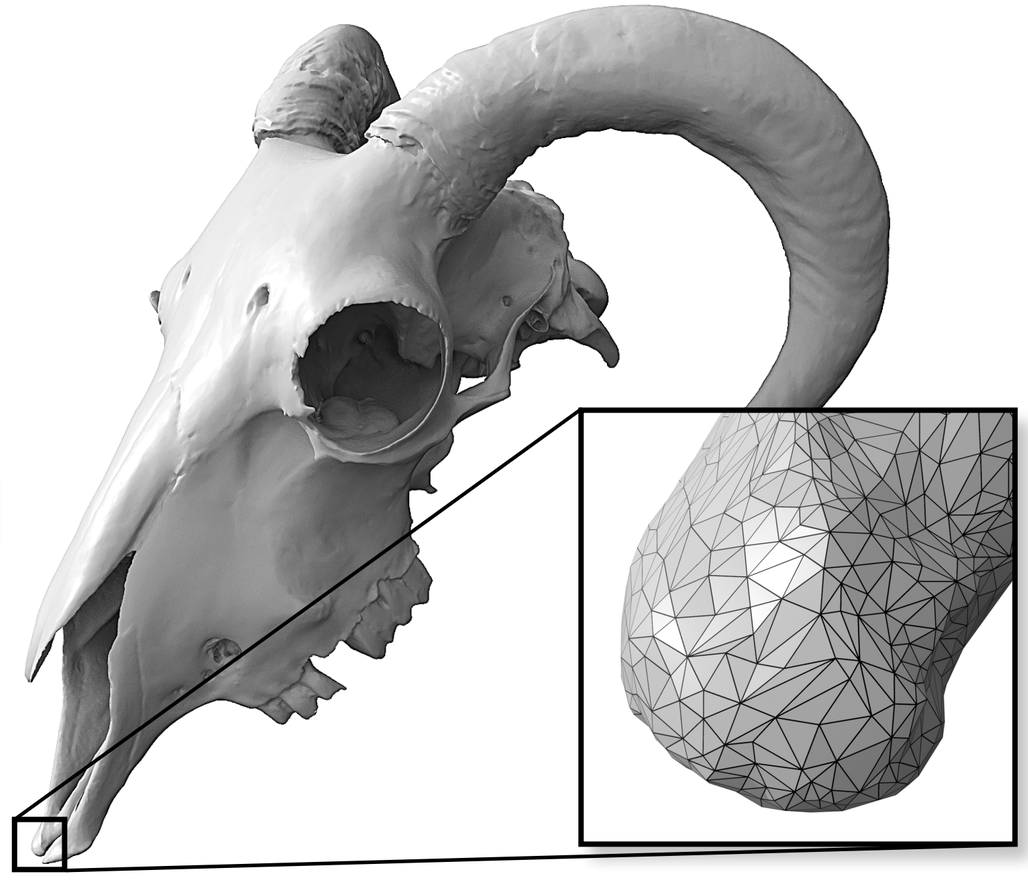
OUTPUT (BITMAP IMAGE)



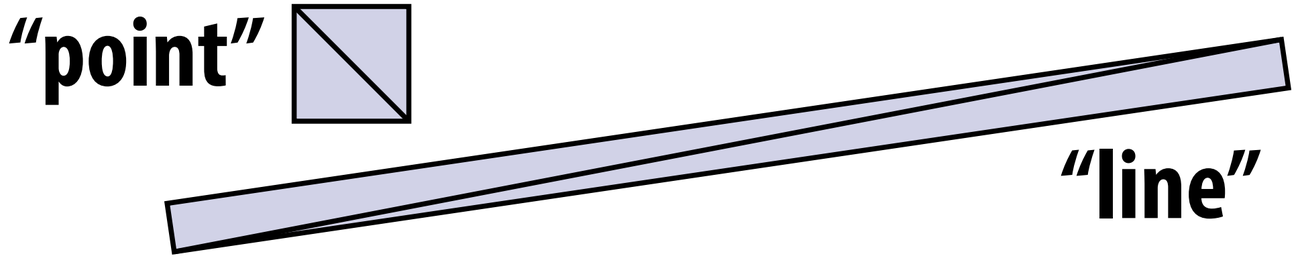
*In practice, usually executed by *graphics processing unit (GPU)*

Why triangles?

- Rasterization pipeline converts <u>all</u> 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"*

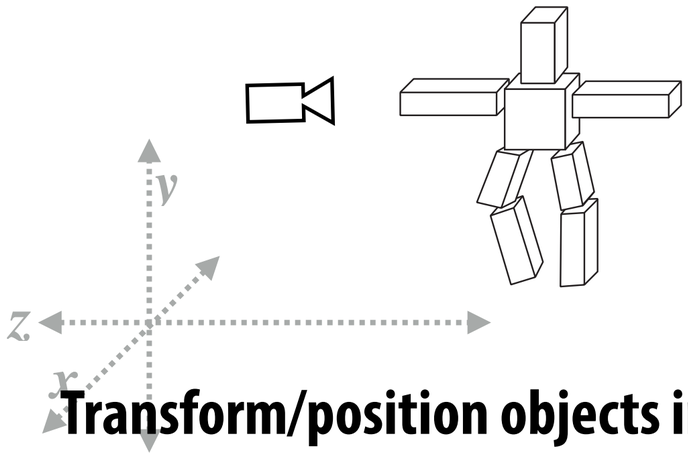


- *Key reason*: once everything is reduced to triangles, can focus on making an extremely well-optimized <u>pipeline</u> for drawing them

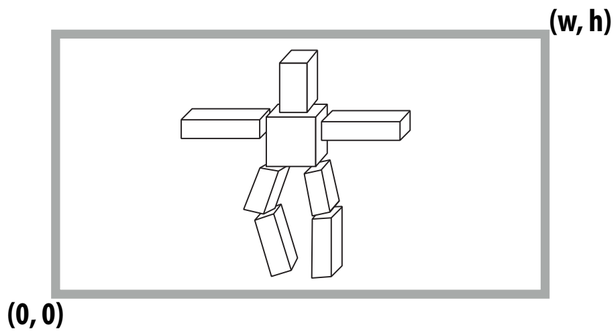


The Rasterization Pipeline

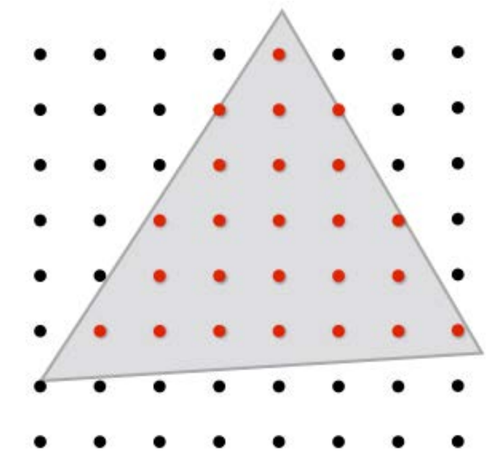
Rough sketch of rasterization pipeline:



Transform/position objects in the world

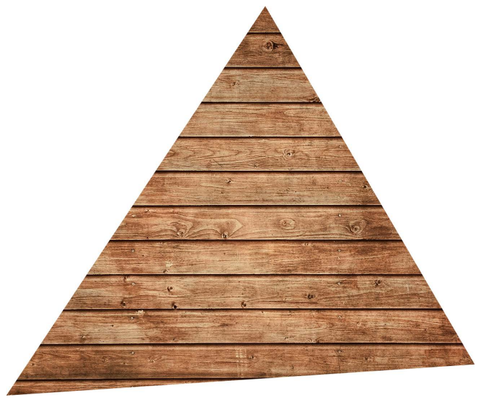


Project objects onto the screen

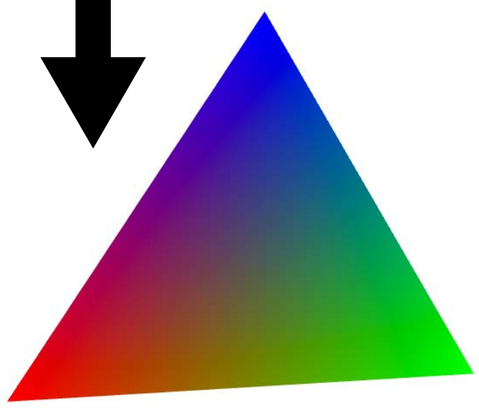


Sample triangle coverage

Combine samples into final image (depth, alpha, . . .)



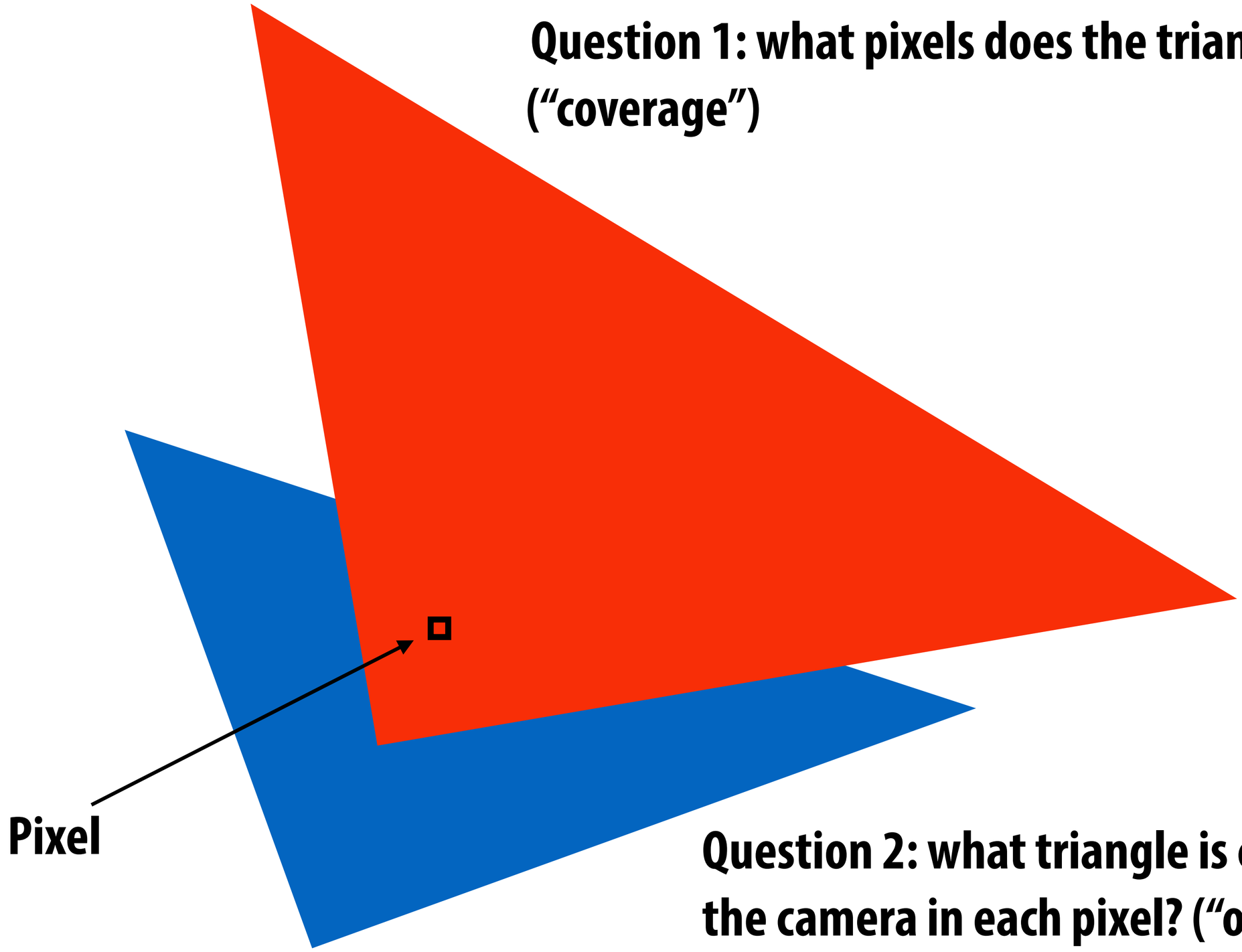
Sample texture maps / evaluate shaders



Interpolate triangle attributes at covered samples

- Reflects standard "real world" pipeline (OpenGL/Direct3D)
  - the rest is just details (e.g., API calls); will discuss in recitation

Let's draw some triangles on the screen

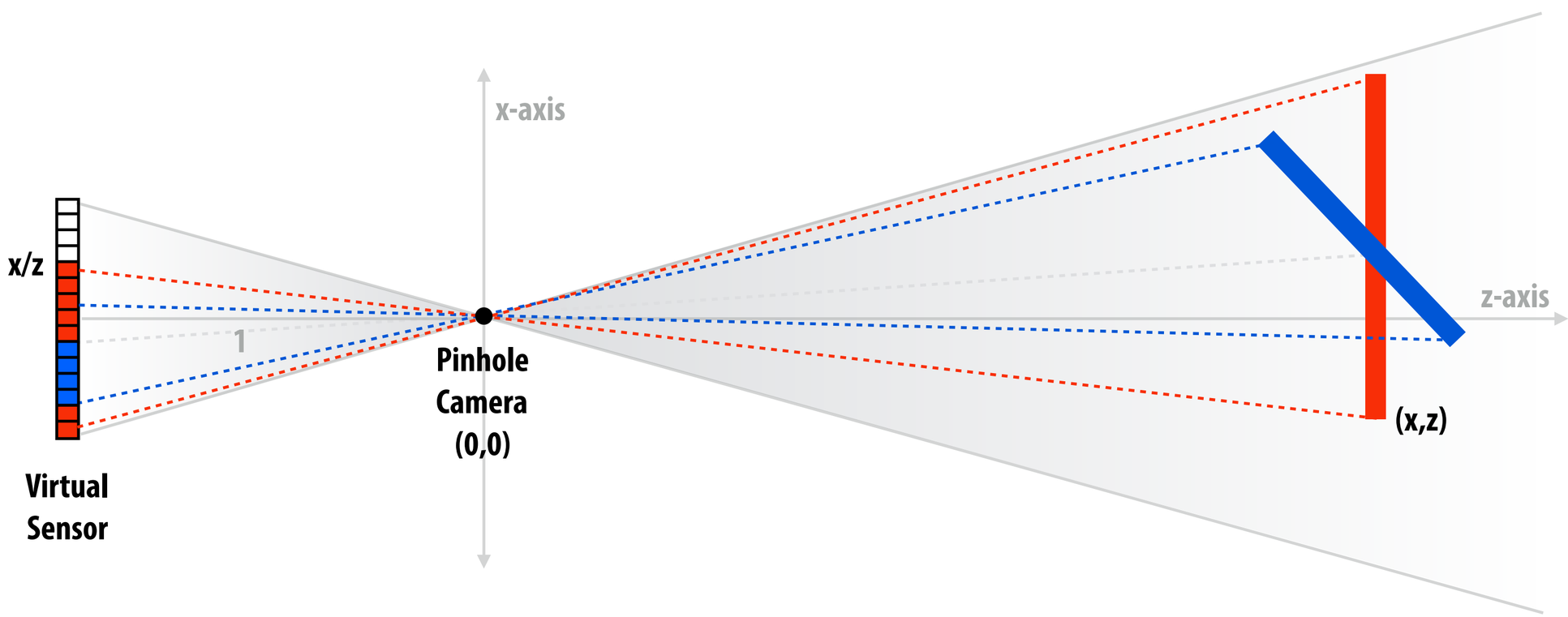


Question 1: what pixels does the triangle overlap? ("coverage")

Question 2: what triangle is closest to the camera in each pixel? ("occlusion")

The visibility problem

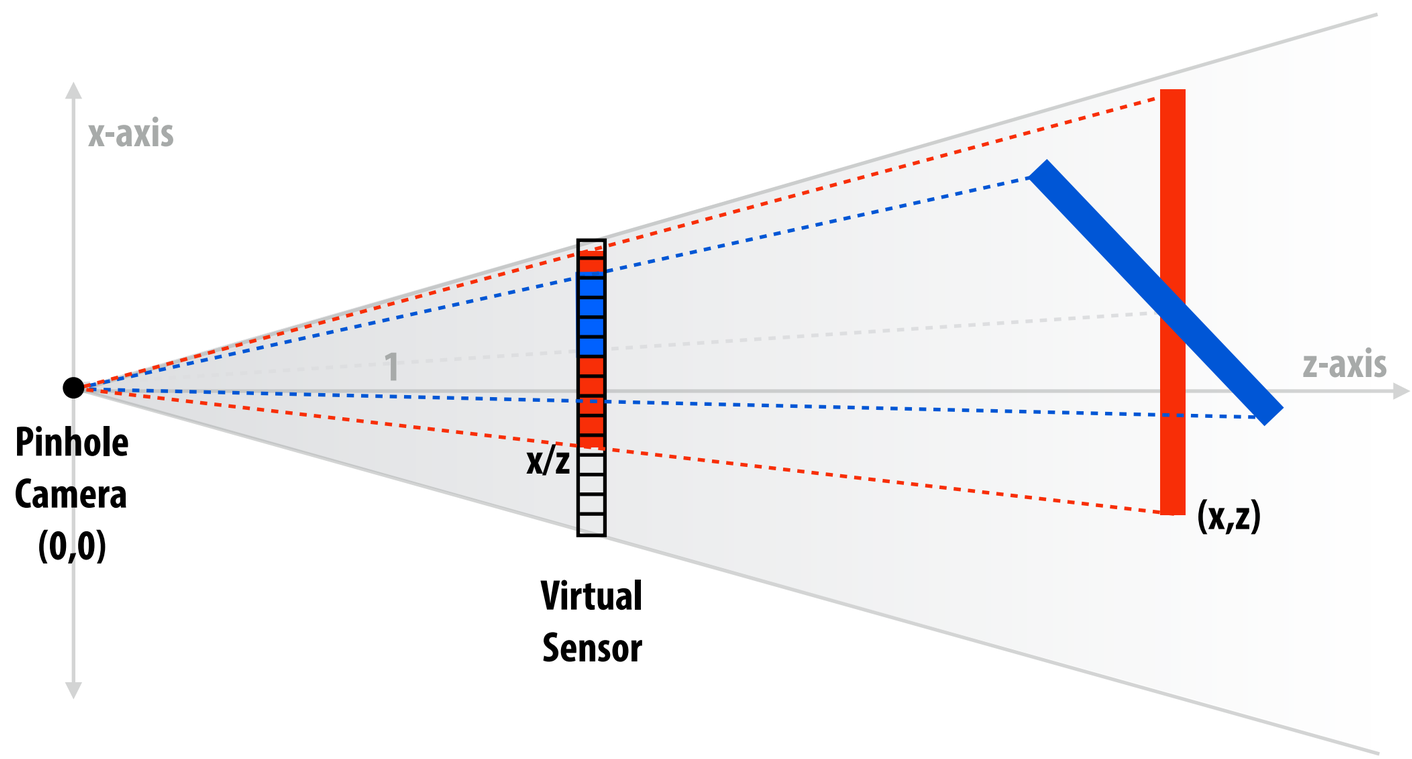
- An informal definition: what scene geometry is visible within each screen pixel?
  - What scene geometry projects into a screen pixel? (coverage)
  - Which geometry is visible from the camera at that pixel? (occlusion)



(Recall *pinhole camera* from first lecture)

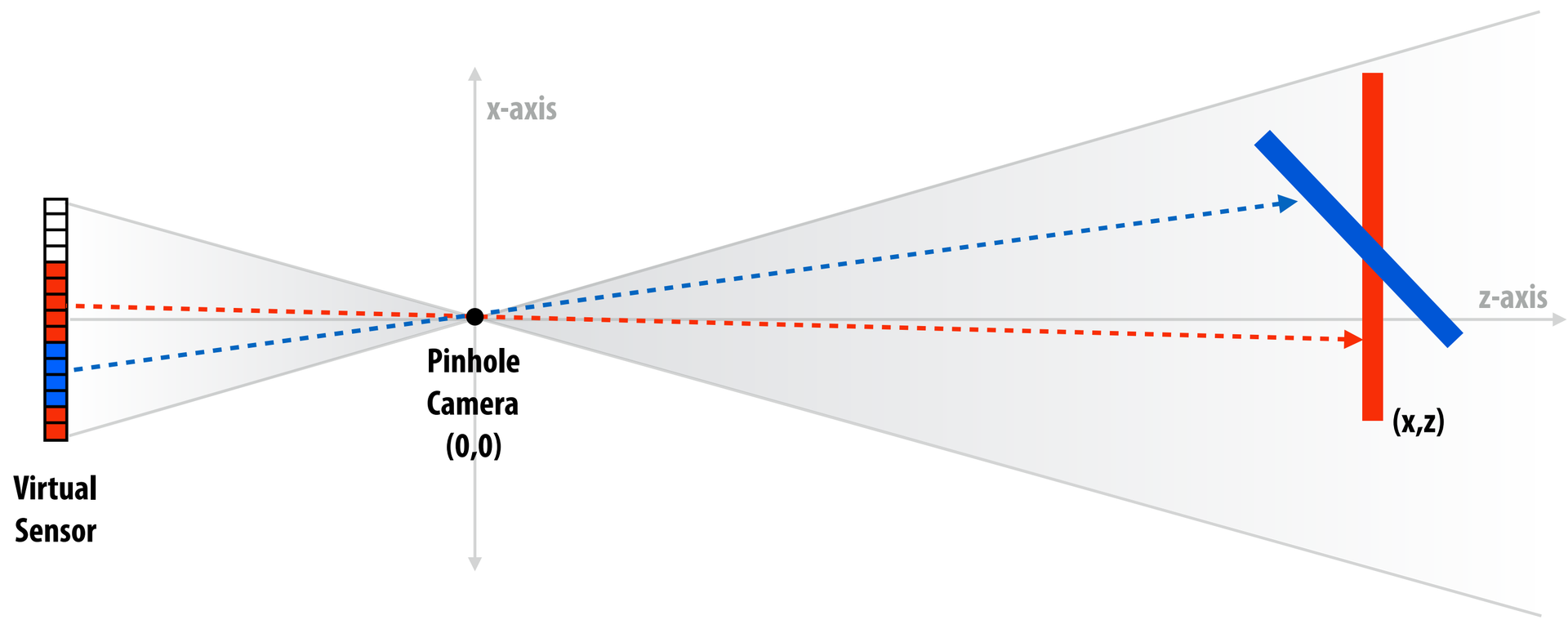
The visibility problem

- **An informal definition: what scene geometry is visible within each screen pixel?**
  - **What scene geometry projects into a screen pixel? (coverage)**
  - **Which geometry is visible from the camera at that pixel? (occlusion)**



The visibility problem (said differently)

- In terms of rays:
  - What scene geometry is hit by a ray from a pixel through the pinhole? (coverage)
  - What object is the first hit along that ray? (occlusion)

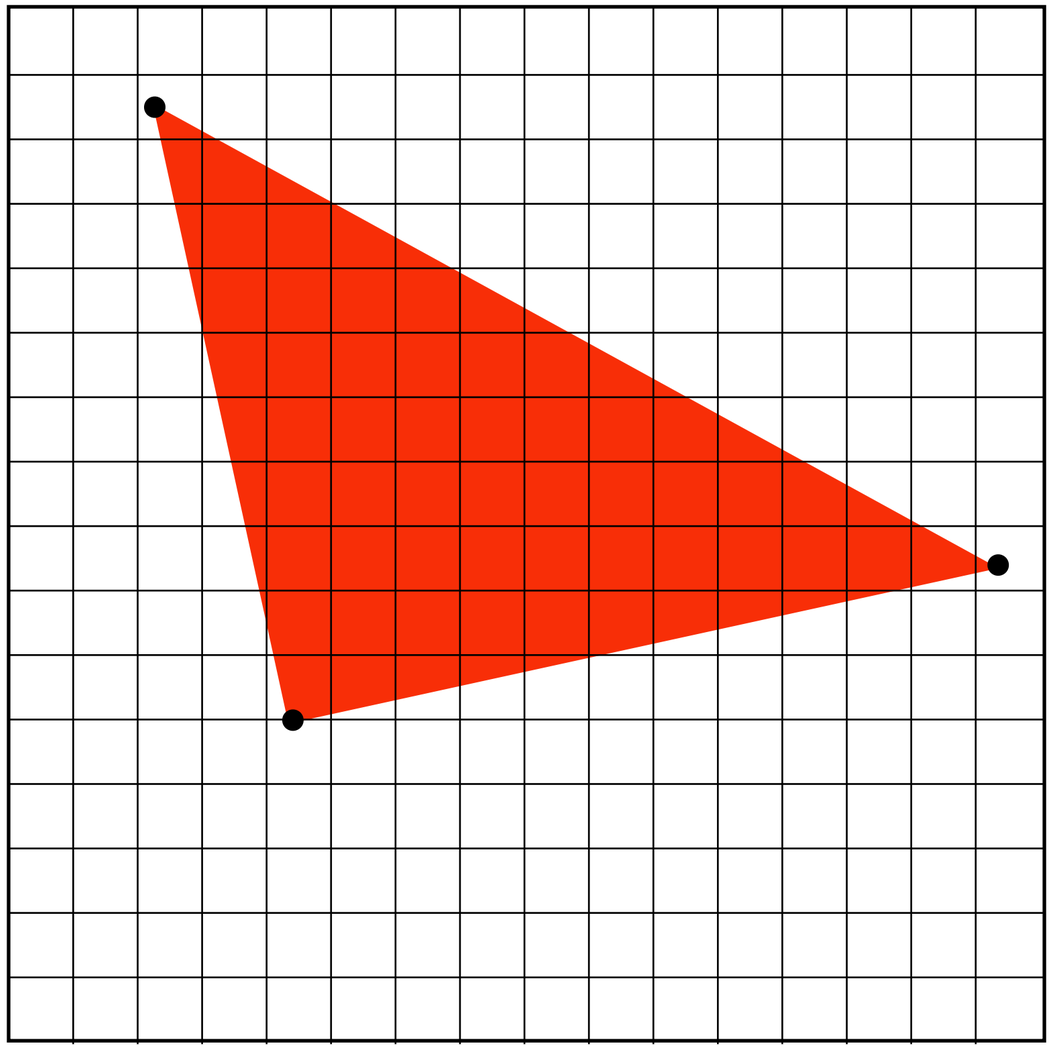


Hold onto this thought for later in the semester.

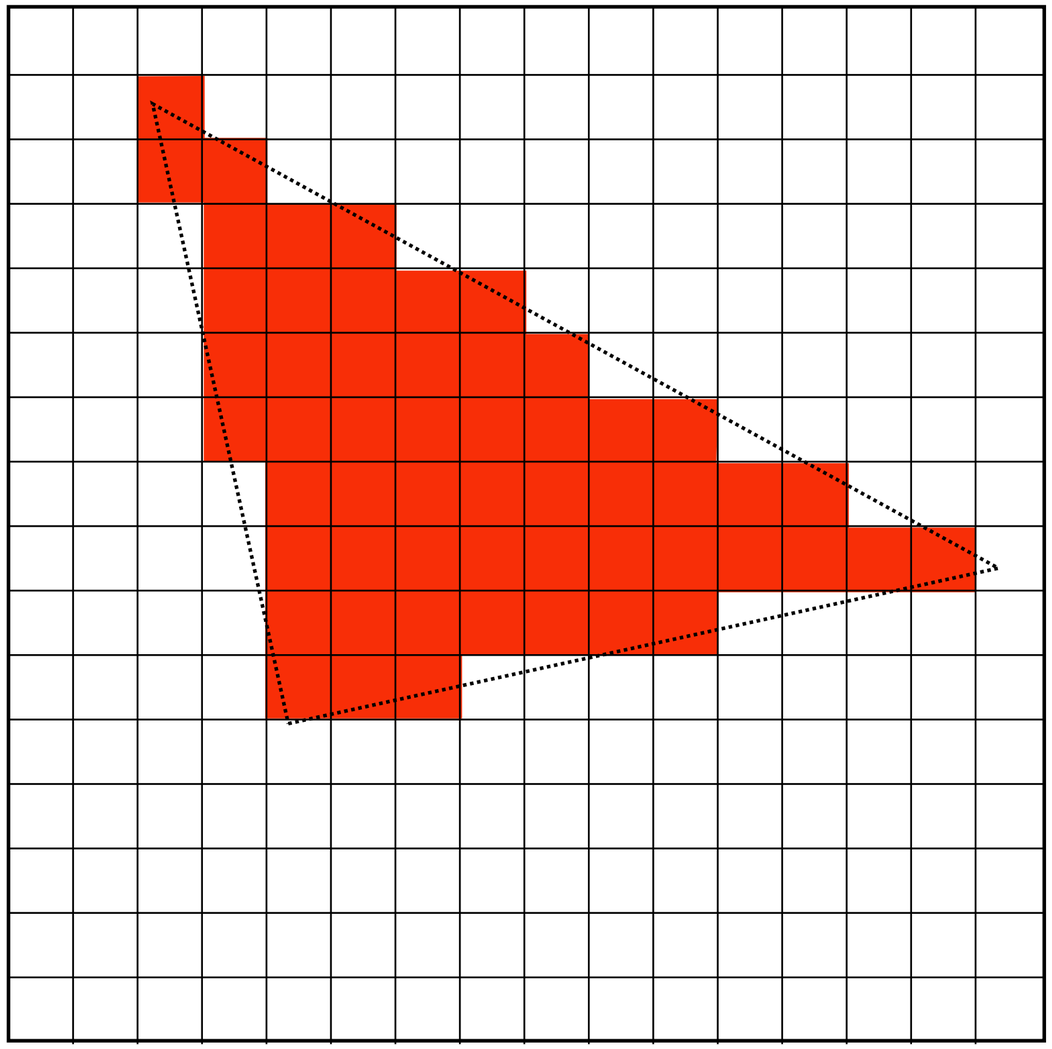
Computing triangle coverage

What pixels does the triangle overlap?

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

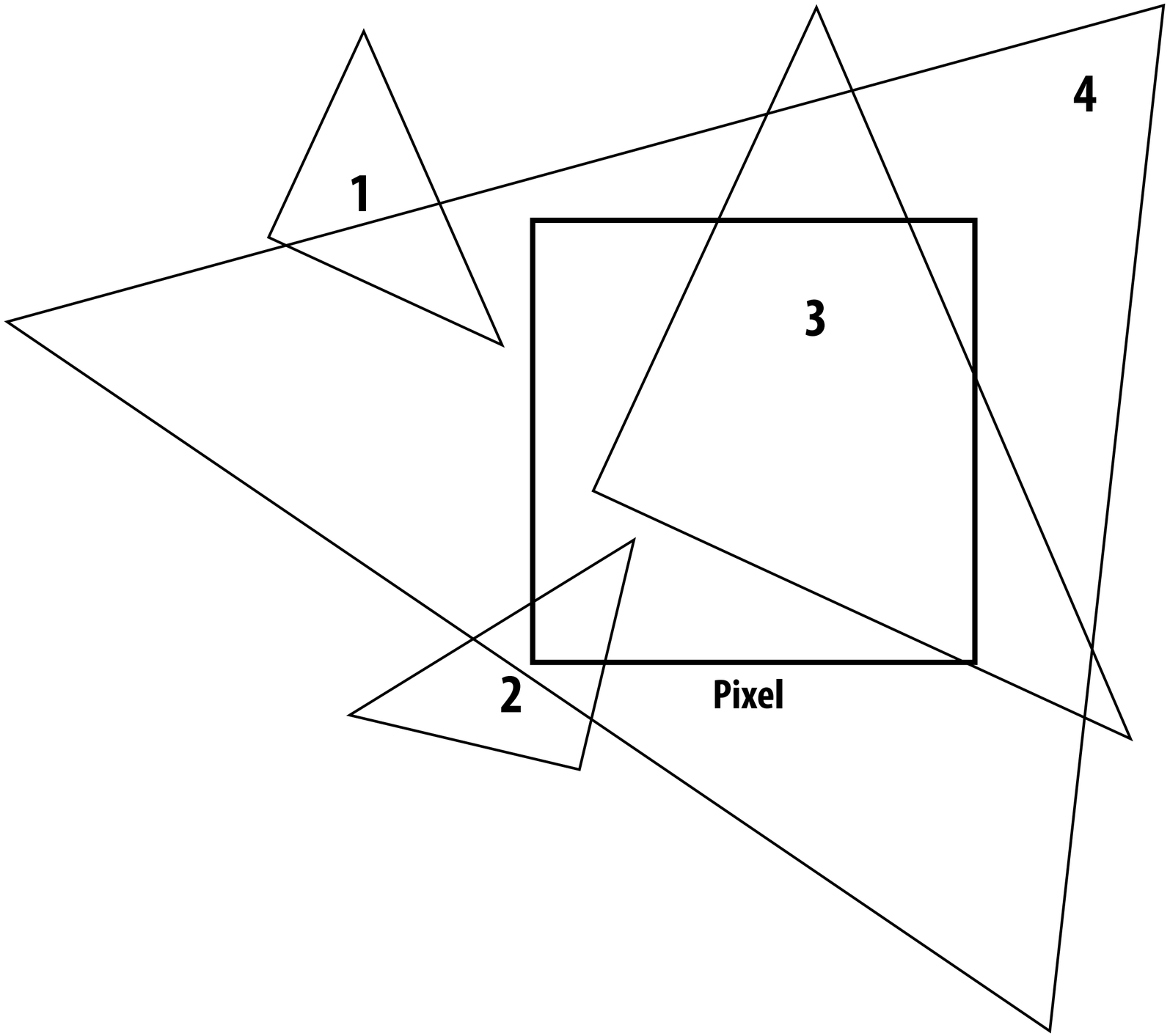


Output:
set of pixels "covered" by the triangle

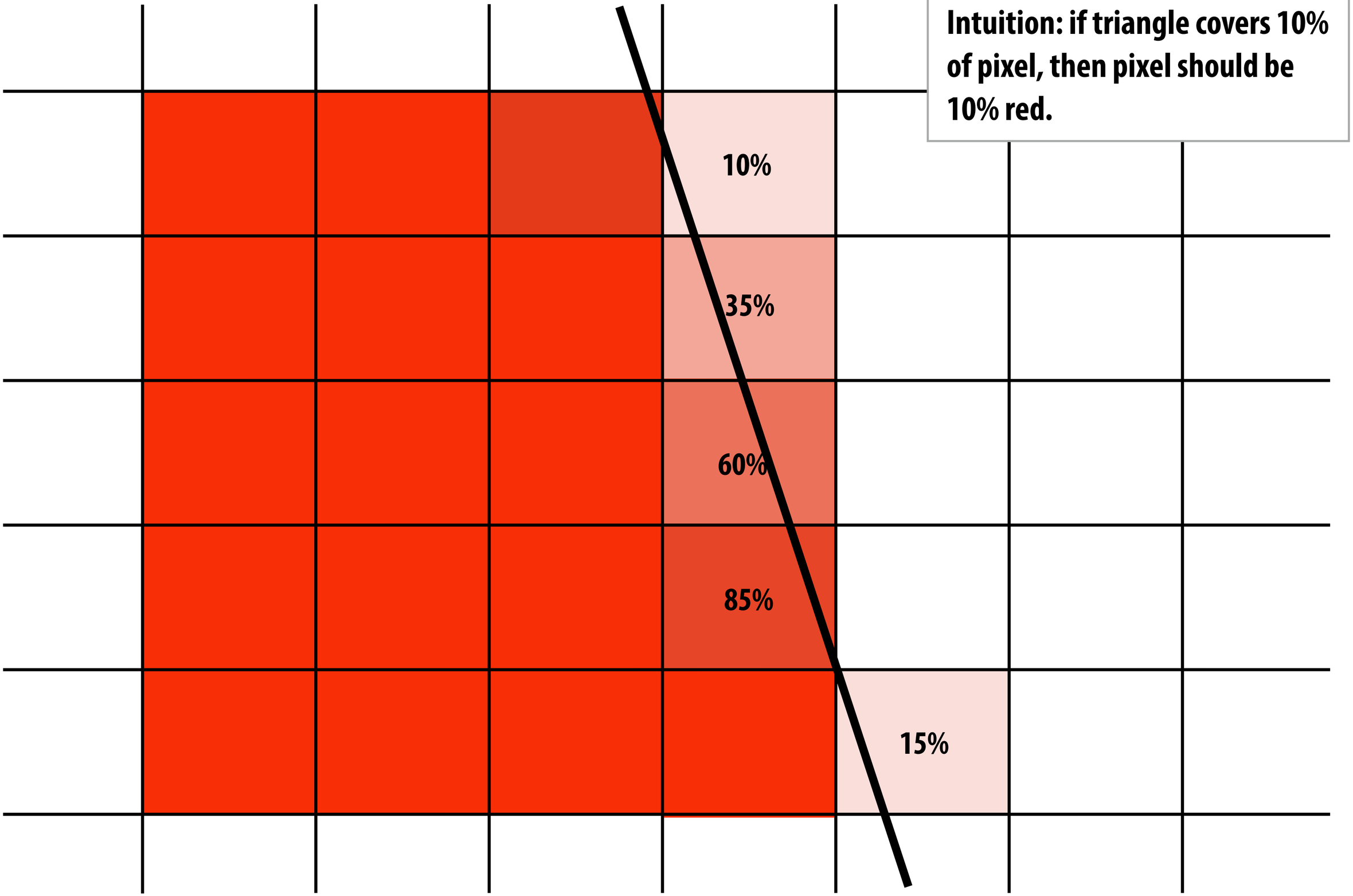


What does it mean for a pixel to be covered by a triangle?

Question: which triangles "cover" this pixel?

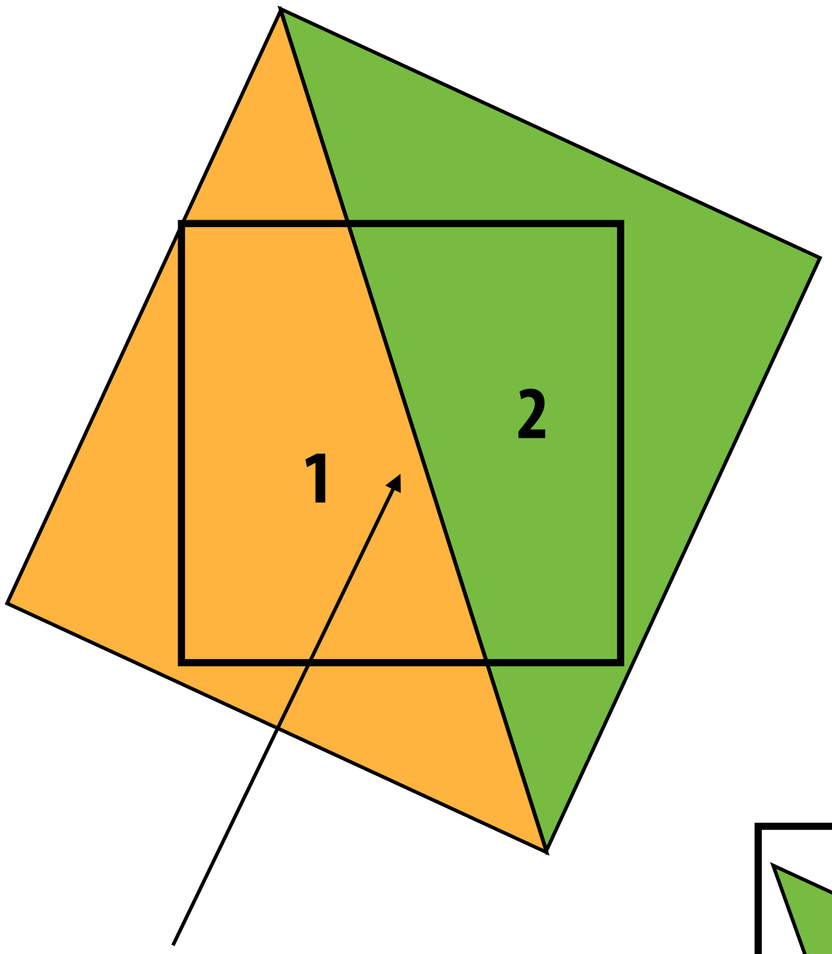


One option: compute fraction of pixel area covered by triangle, then color pixel according to this fraction.

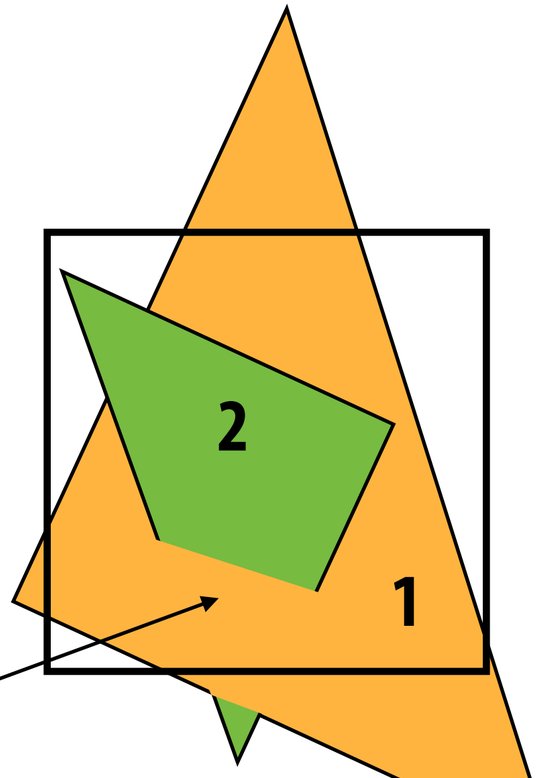


Intuition: if triangle covers 10% of pixel, then pixel should be 10% red.

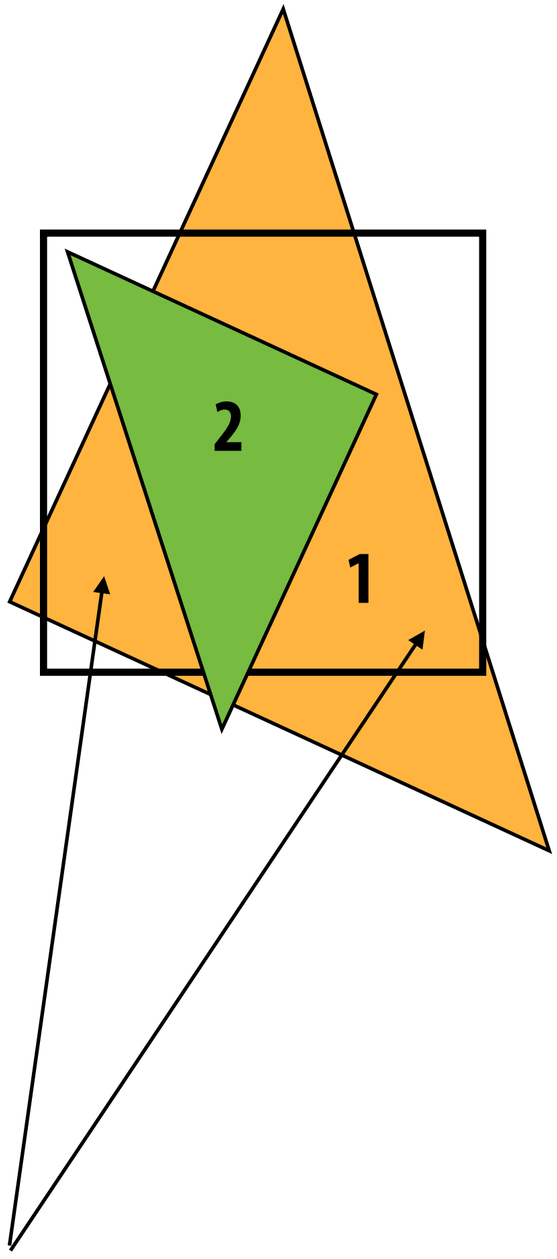
Coverage gets tricky when considering occlusion



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

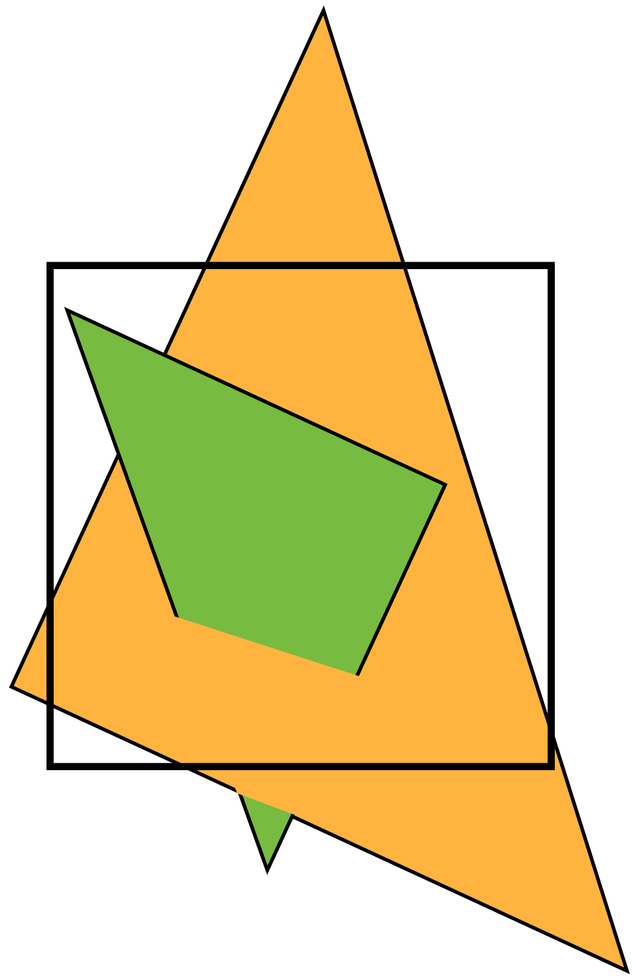


Interpenetration of triangles: even trickier

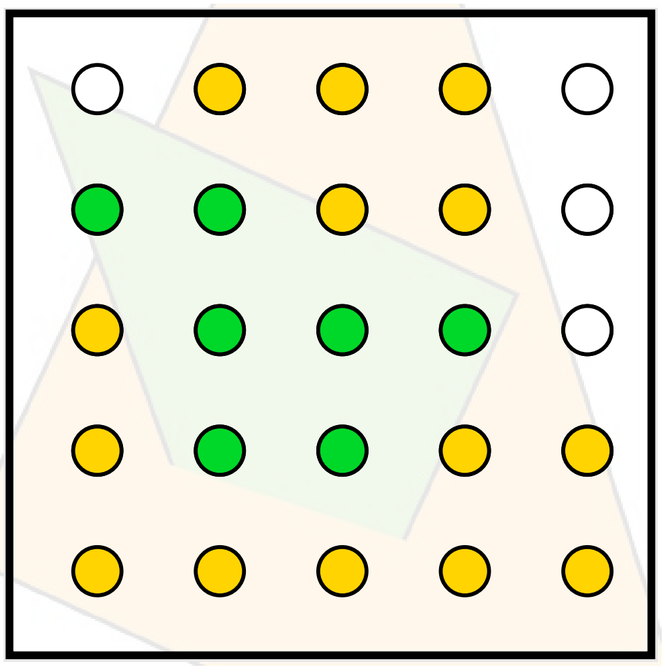


Two regions of triangle 1 contribute to pixel. One of these regions is not even convex.

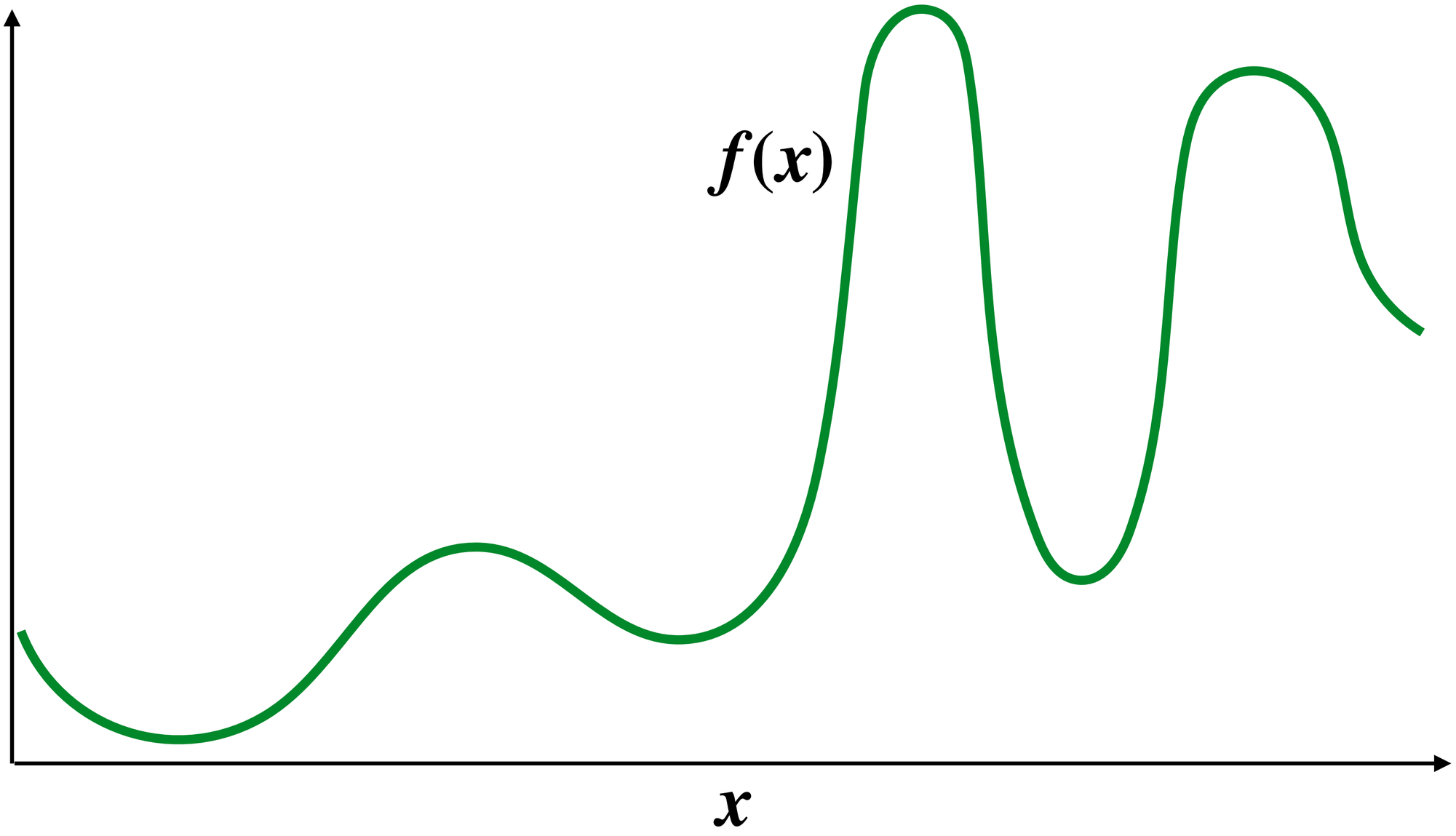
Coverage via sampling



- Real scenes are *complicated!*
  - occlusion, transparency, …
- Computing *exact* coverage is not practical
- Instead: view coverage as a sampling 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
- First, let’s talk about sampling in general…

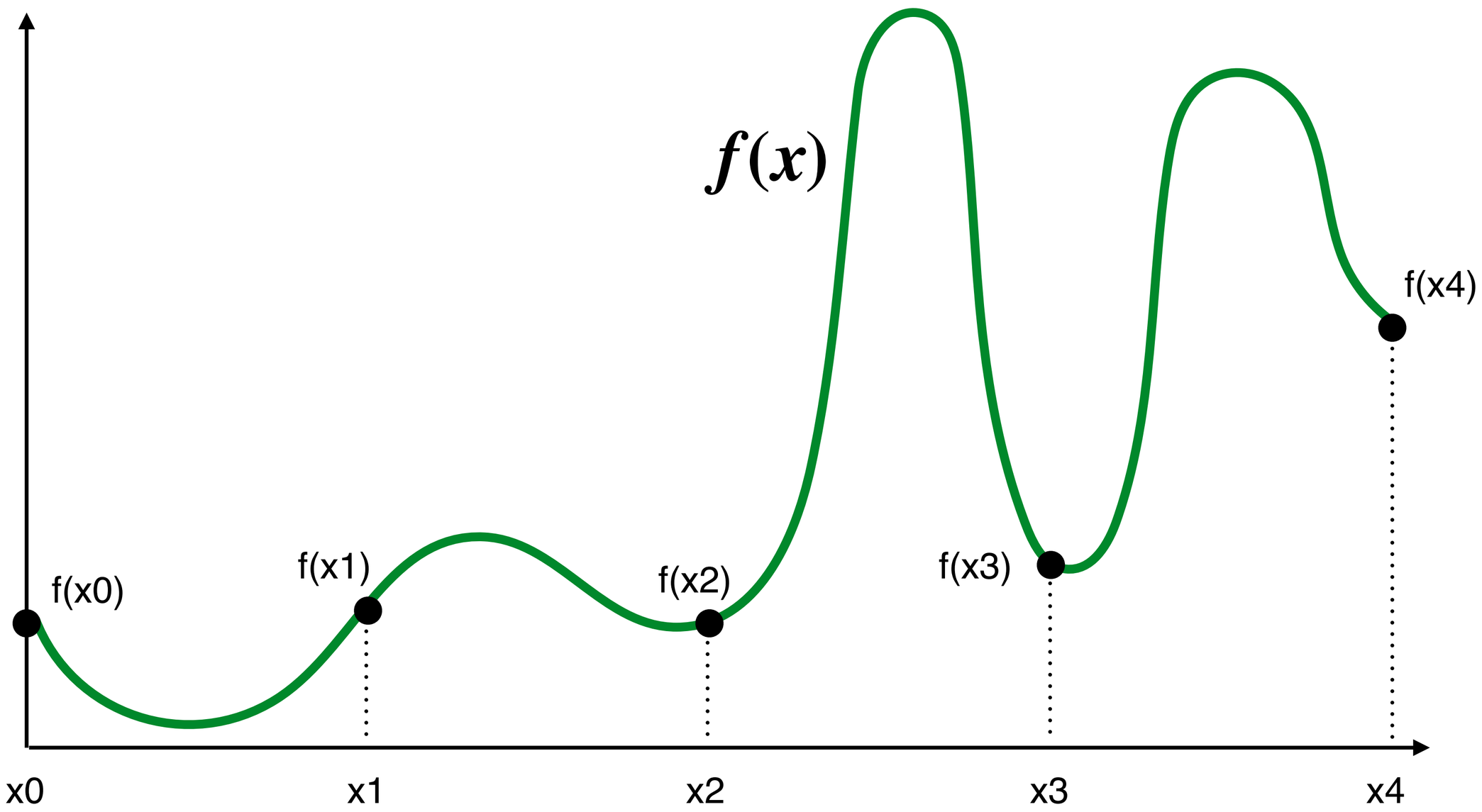


Sampling 101: Sampling a 1D signal



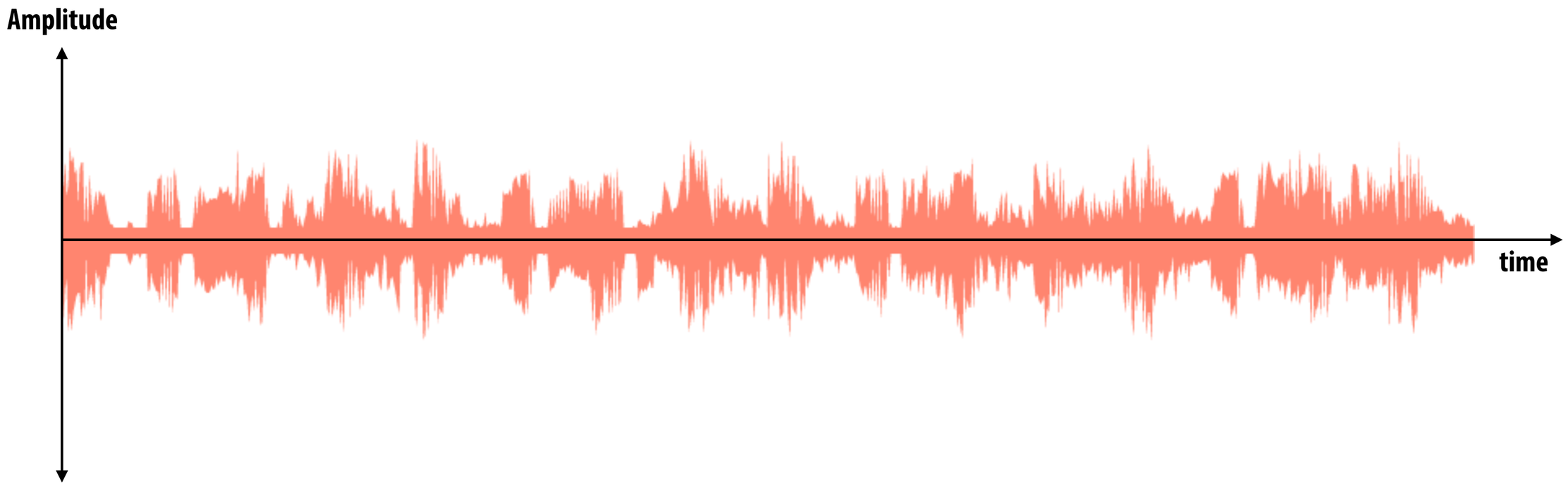
Sampling = taking measurements of a signal

Below: 5 measurements ("samples") of $f(x)$

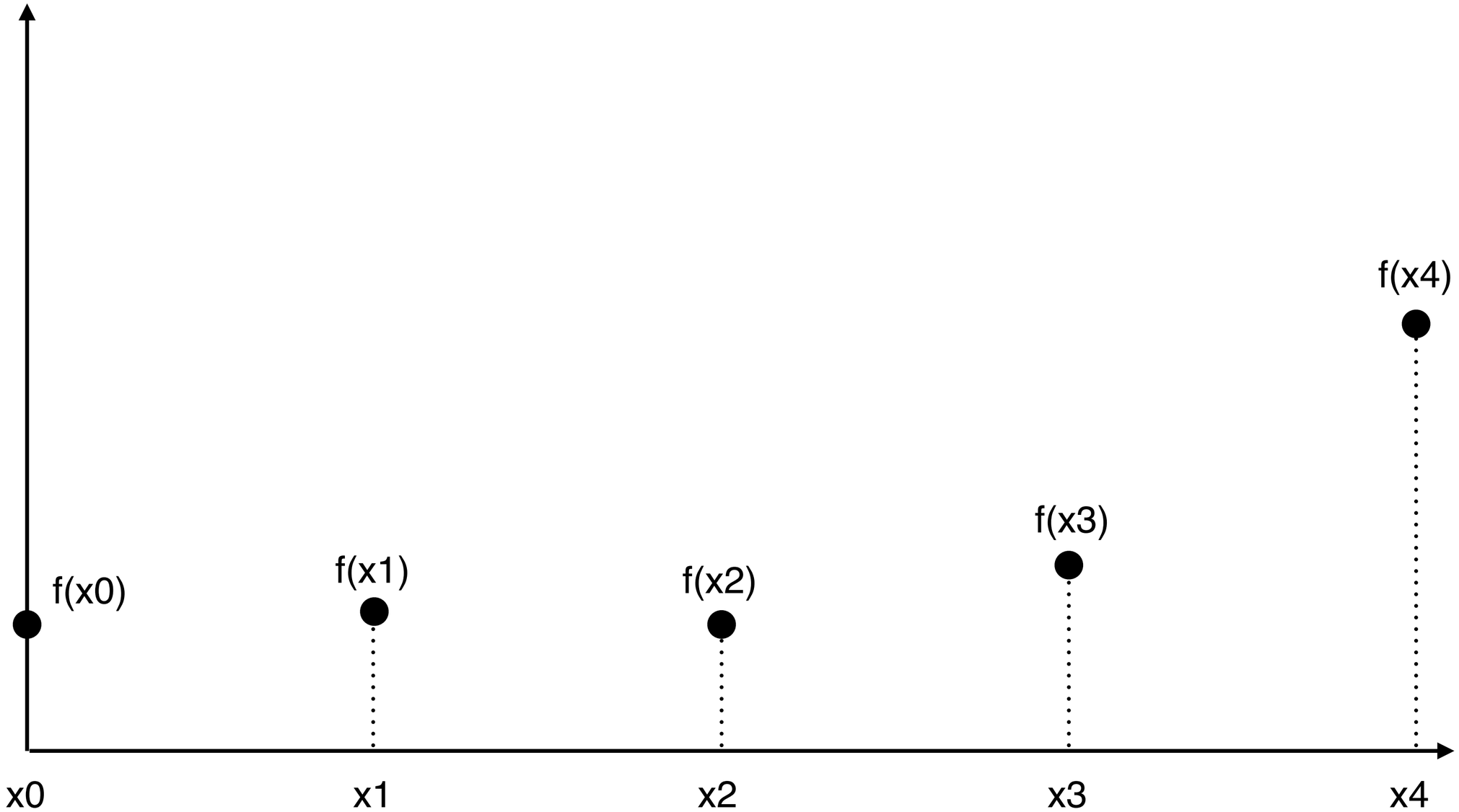


Audio file: stores samples of a 1D signal

Most consumer audio is sampled at 44.1 KHz



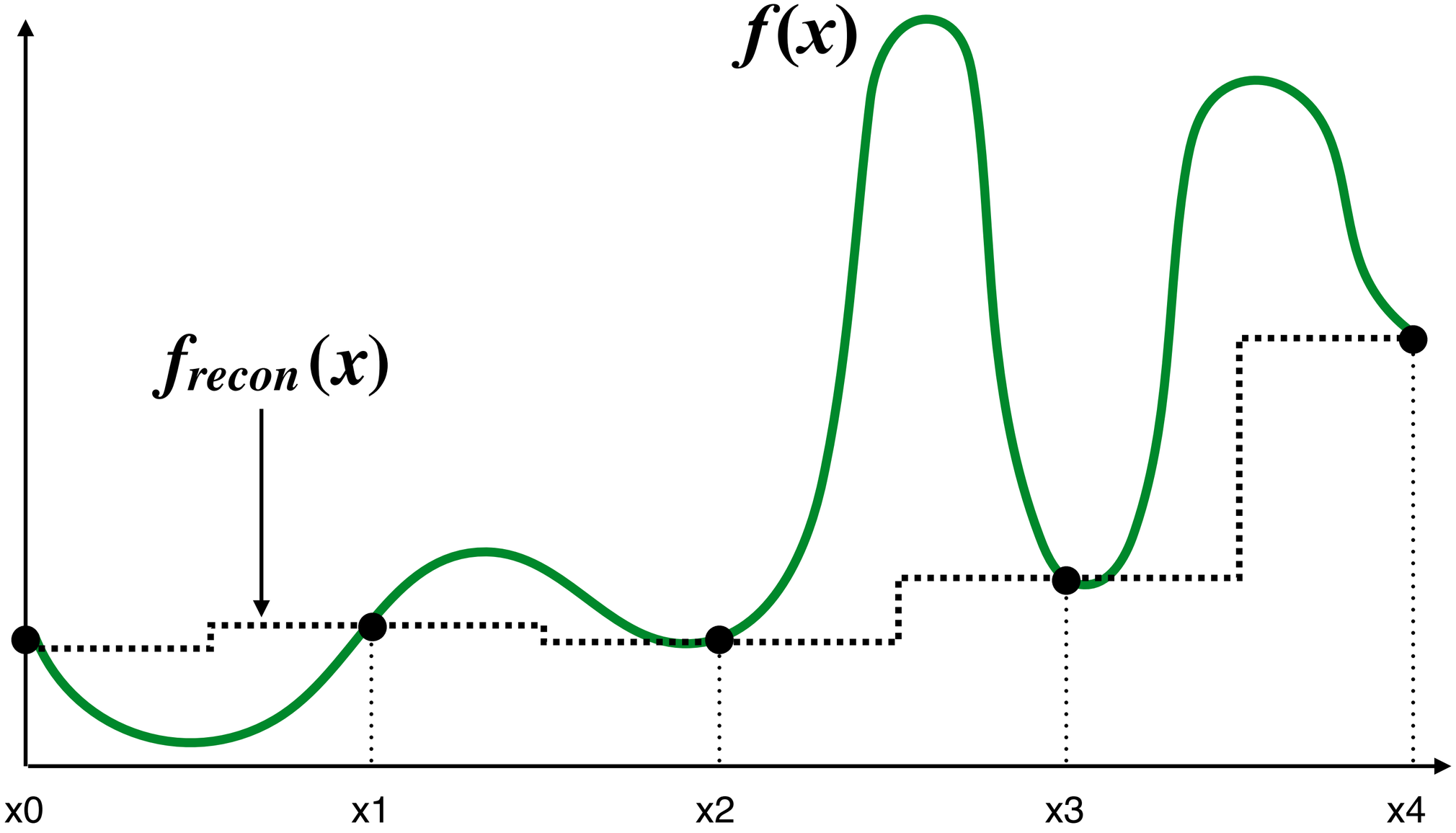
Reconstruction: given a set of samples, how might we attempt to reconstruct the original signal $f(x)$?



Piecewise constant approximation

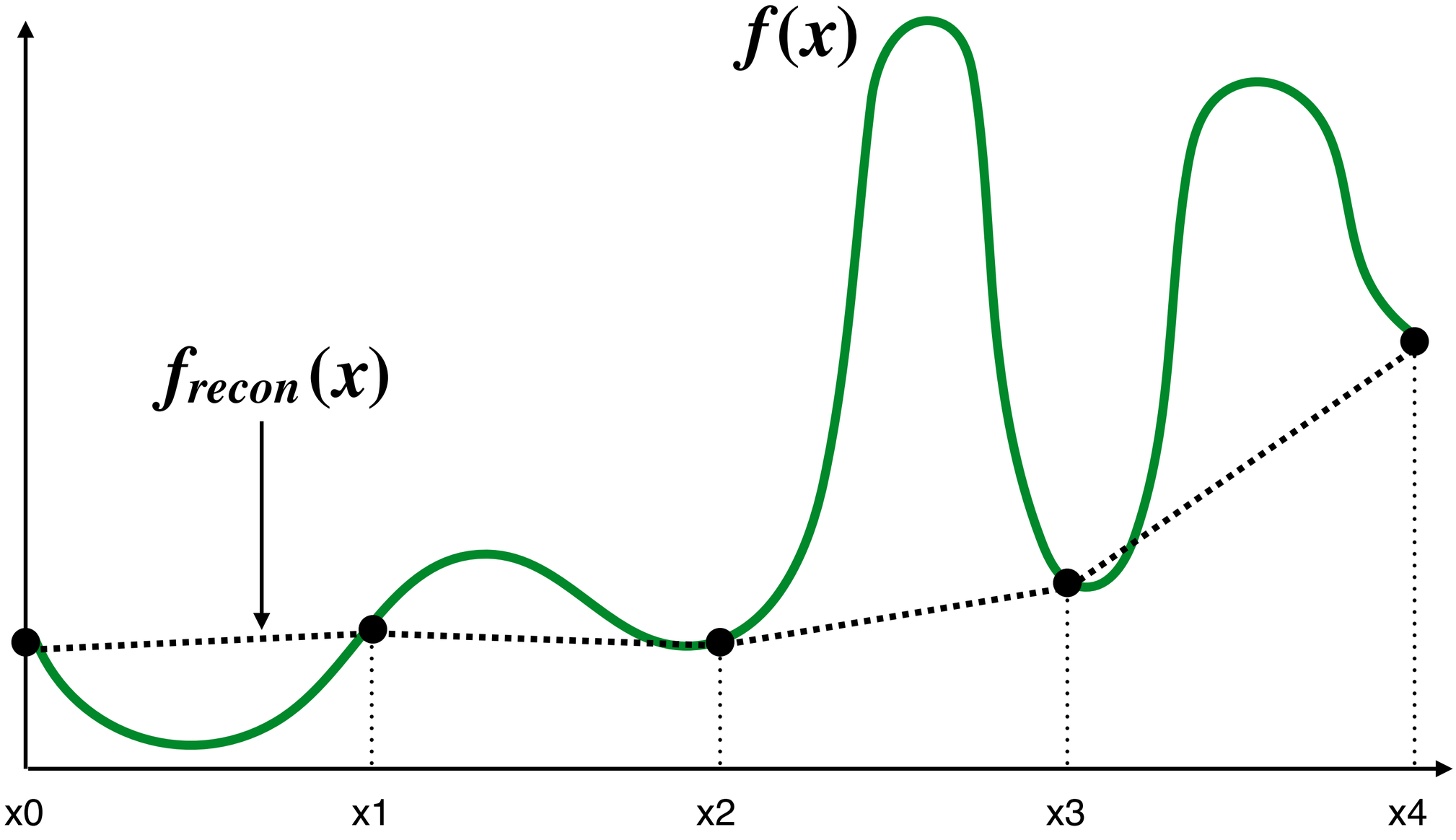
$f_{recon}(x)$ = value of sample closest to x

$f_{recon}(x)$ approximates $f(x)$



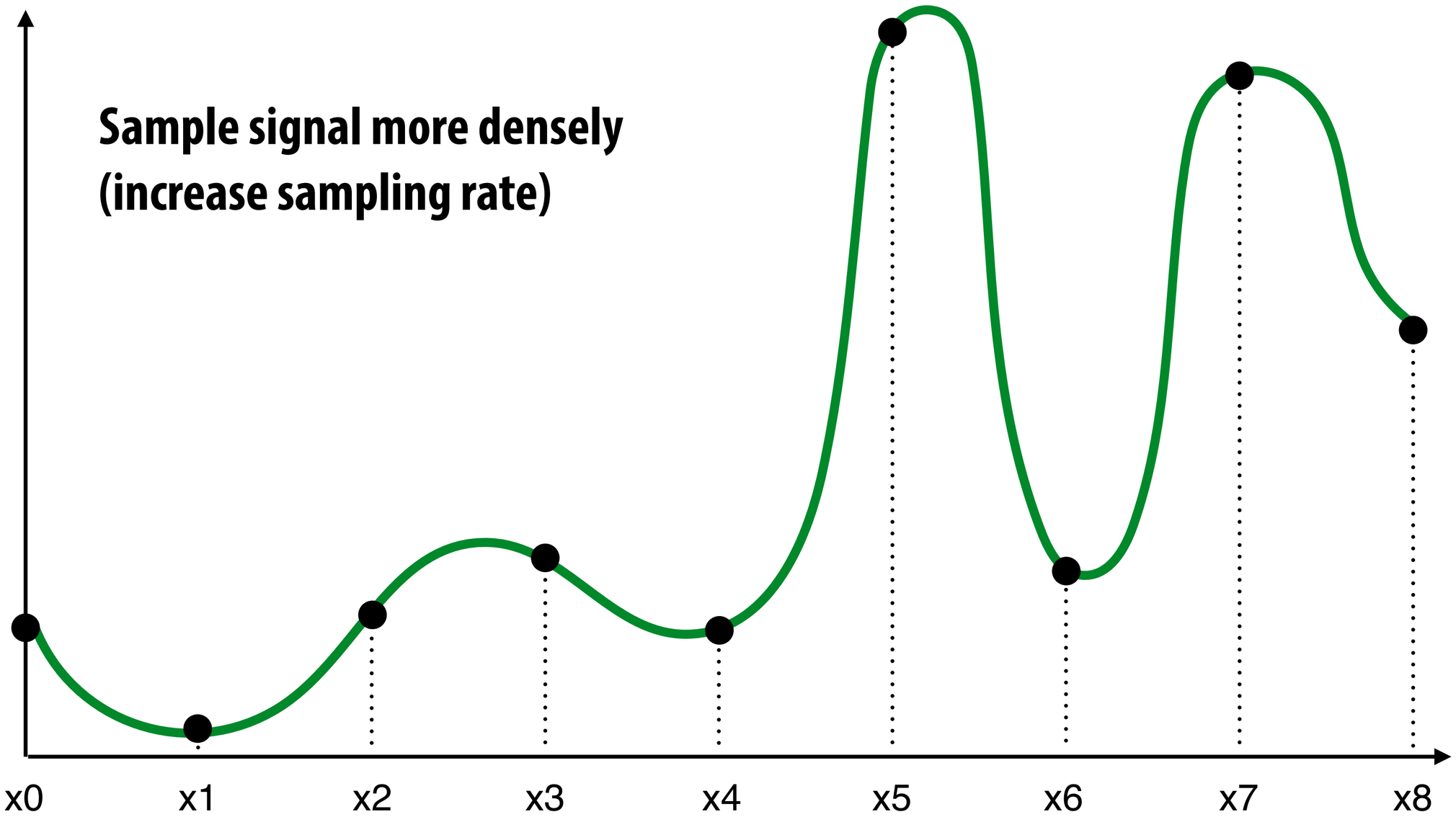
Piecewise linear approximation

$f_{recon}(x)$ = linear interpolation between values of two closest samples to x

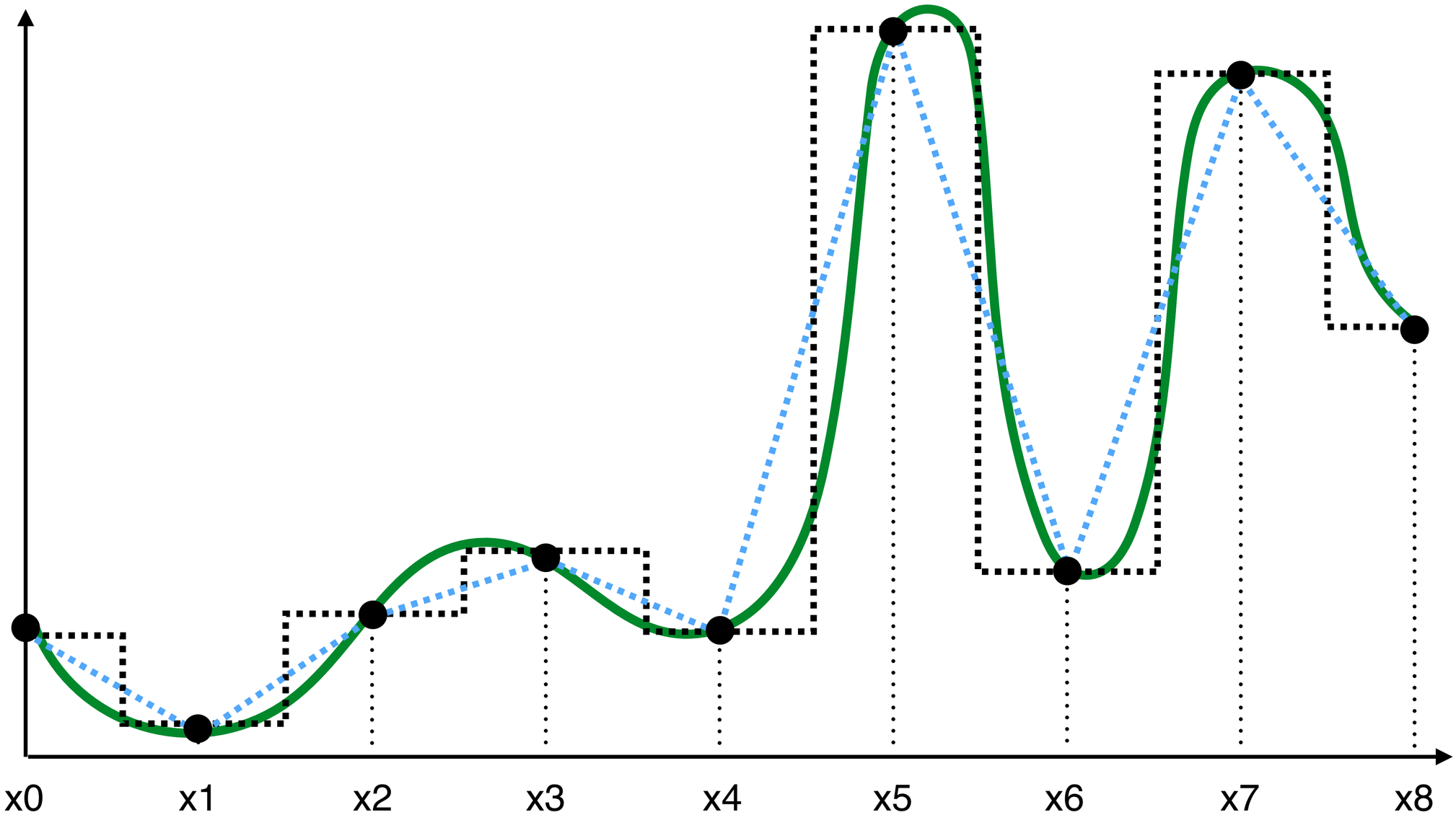


How can we represent the signal more accurately?

Sample signal more densely
(increase sampling rate)



Reconstruction from denser sampling



= reconstruction via nearest

= reconstruction via linear interpolation

2D Sampling & Reconstruction

- Basic story doesn't change much for images:
  - sample values measure image (i.e., signal) at sample points
  - apply interpolation/reconstruction filter to approximate image

original

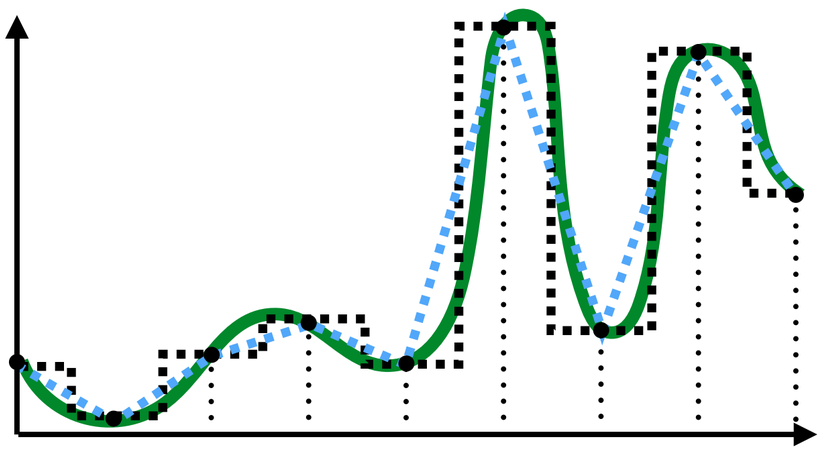
piecewise constant ("nearest neighbor")

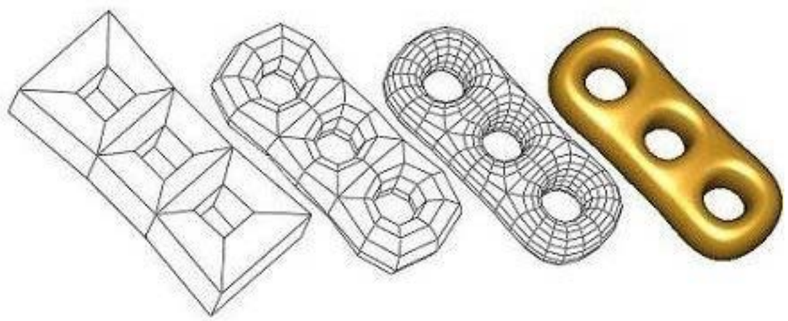
piecewise *bi*-linear

Sampling 101: Summary

- Sampling = measurement of a signal
  - Encode signal as discrete set of samples
  - In principle, represent values at specific points (though hard to measure in reality!)

- Reconstruction = generating signal from a discrete set of samples
  - Construct a function that interpolates or approximates function values
  - E.g., piecewise constant/"nearest neighbor", or piecewise linear
  - Many more possibilities! For all kinds of signals (audio, images, geometry…)



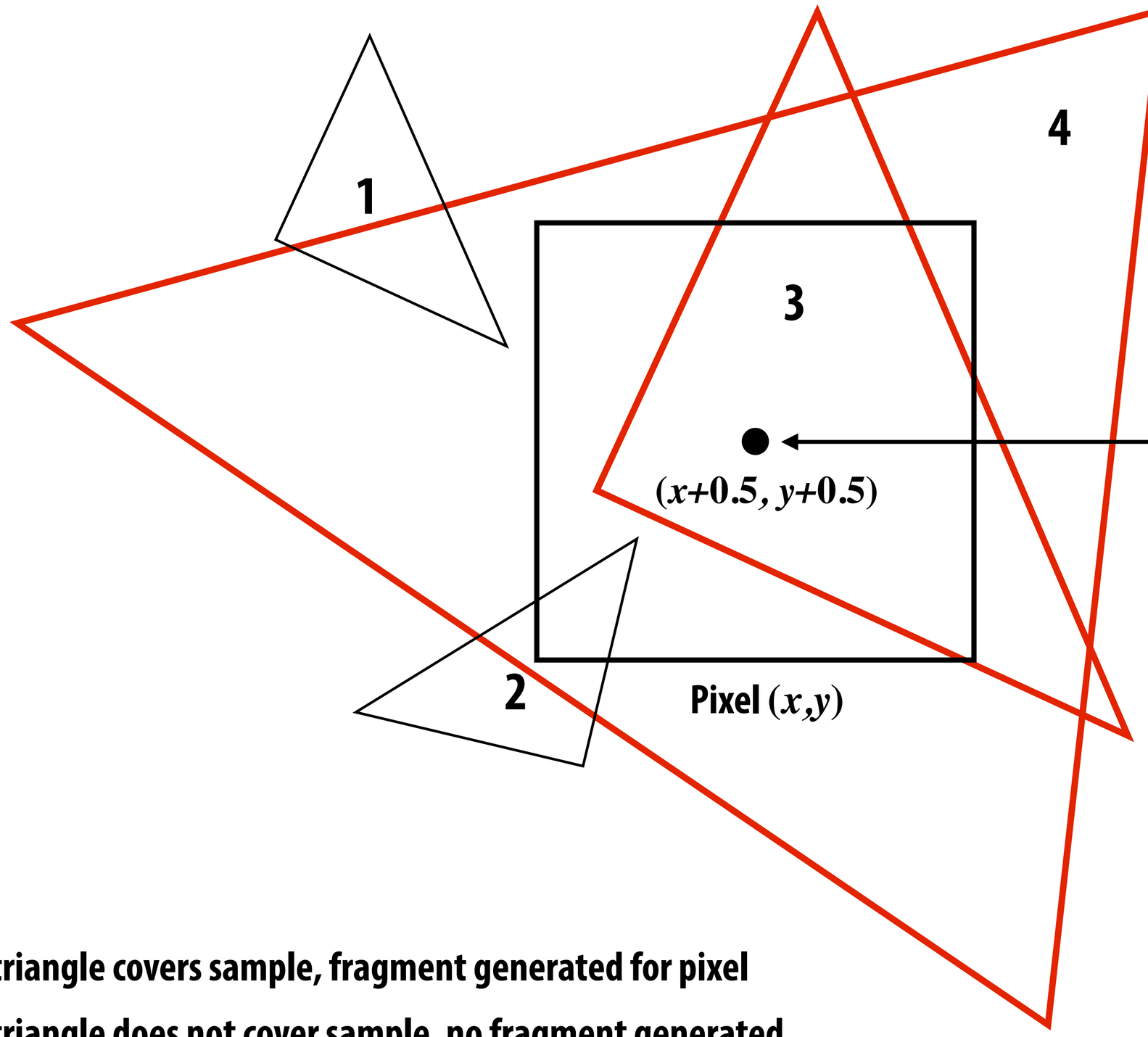


[Image credit: Wikipedia]

For coverage, what function are we sampling?

$$\text{coverage}(x,y) = \begin{cases} 1 & \text{if the triangle contains point } (x,y) \\ 0 & \text{otherwise} \end{cases}$$

Estimate triangle-screen coverage by sampling the binary function: coverage(x,y)



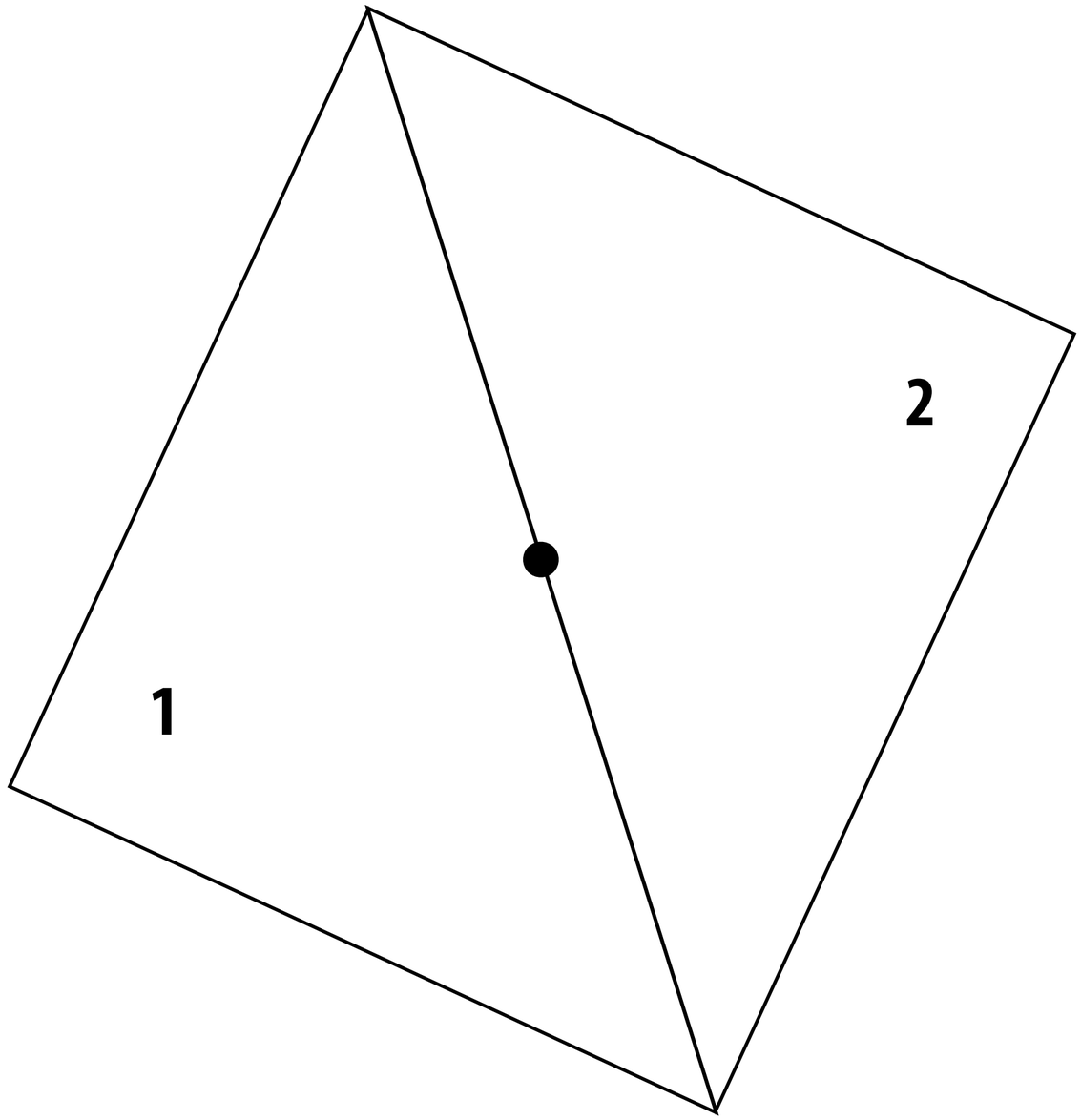
Example:
Here I chose the coverage sample point to be at a point corresponding to the pixel center.

= triangle covers sample, fragment generated for pixel

= triangle does not cover sample, no fragment generated

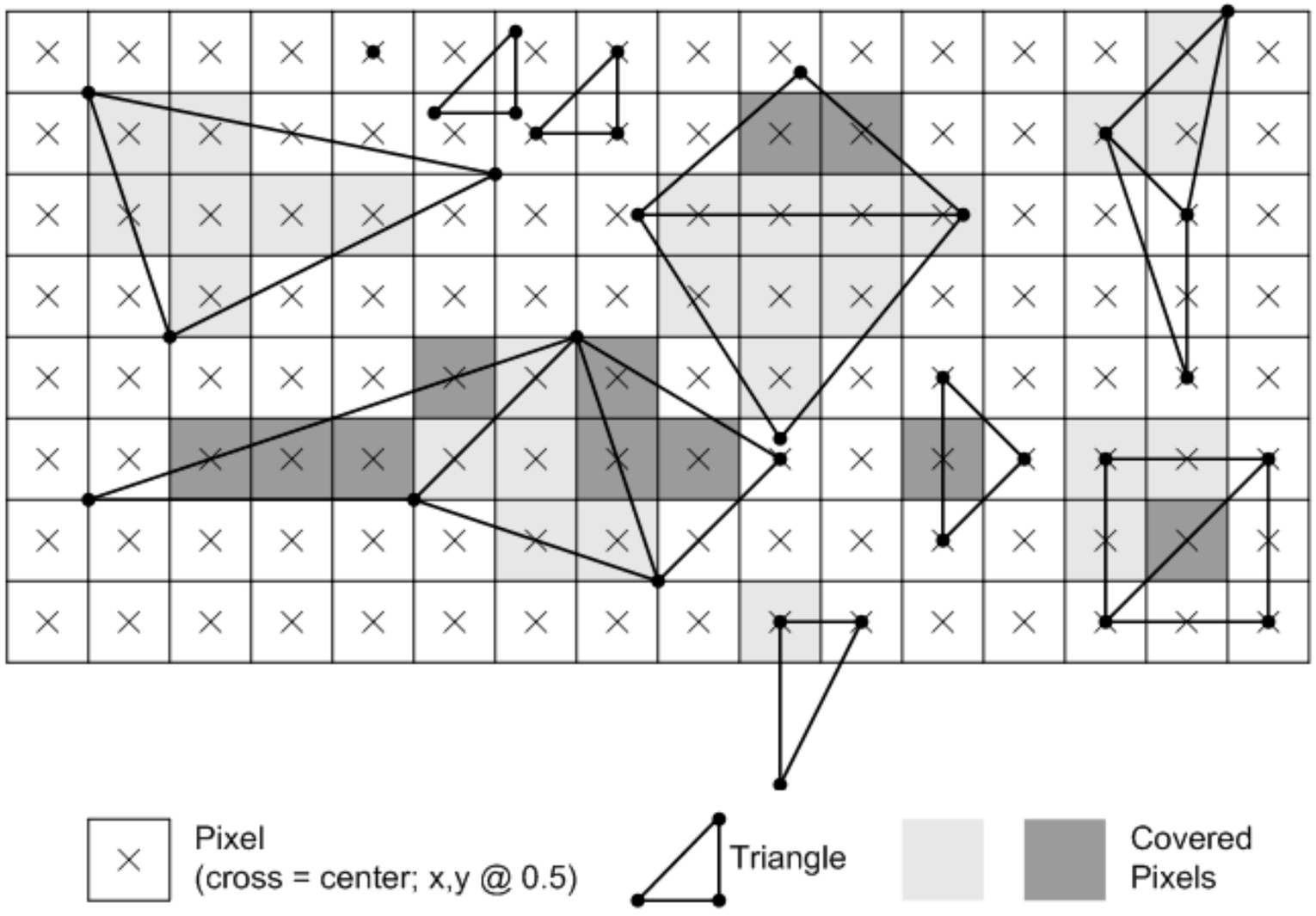
Edge cases (literally)

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



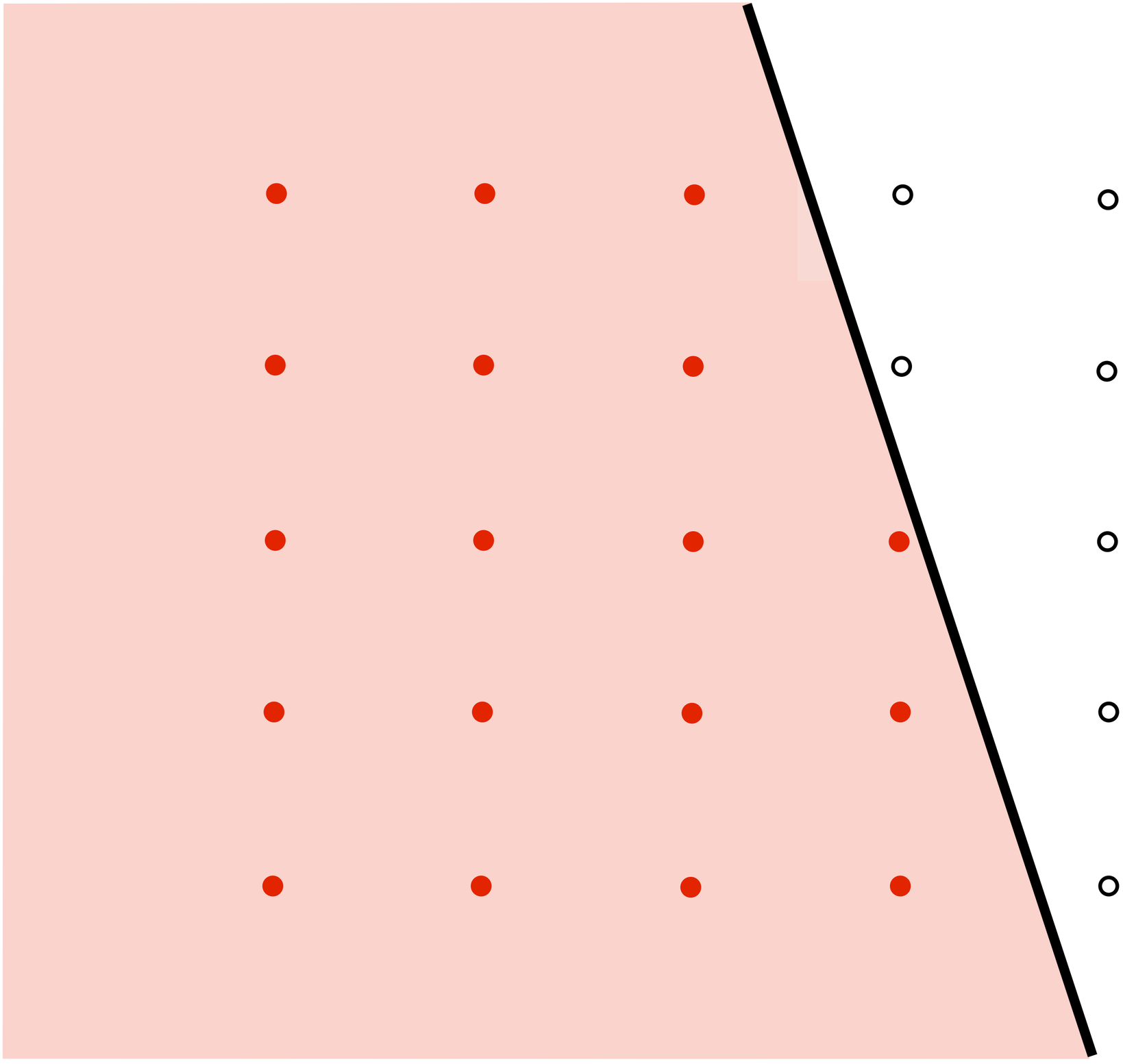
Breaking Ties*

- 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)

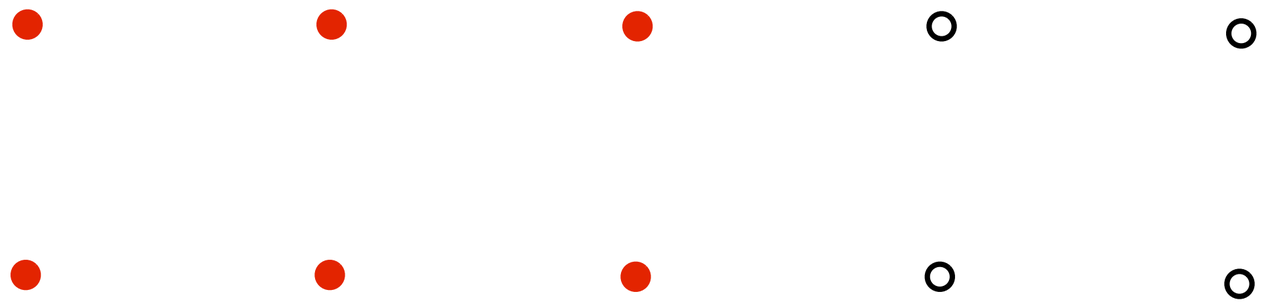


*These are the rules used in OpenGL/Direct3D, i.e., in modern GPUs. Source: Direct3D Programming Guide, Microsoft

Results of sampling triangle coverage

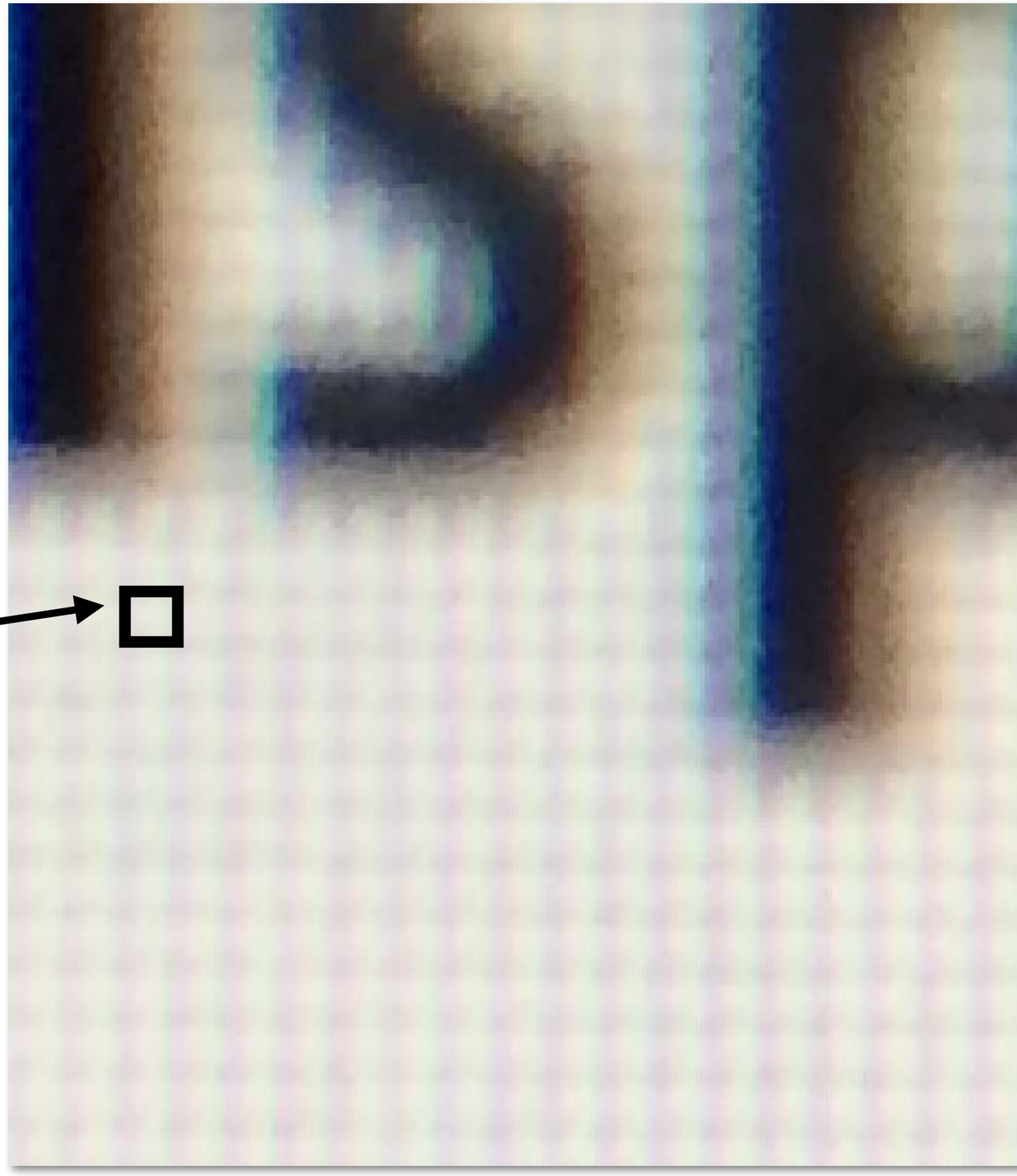


I have a sampled signal, now I want to display it on a screen



Pixels on a screen

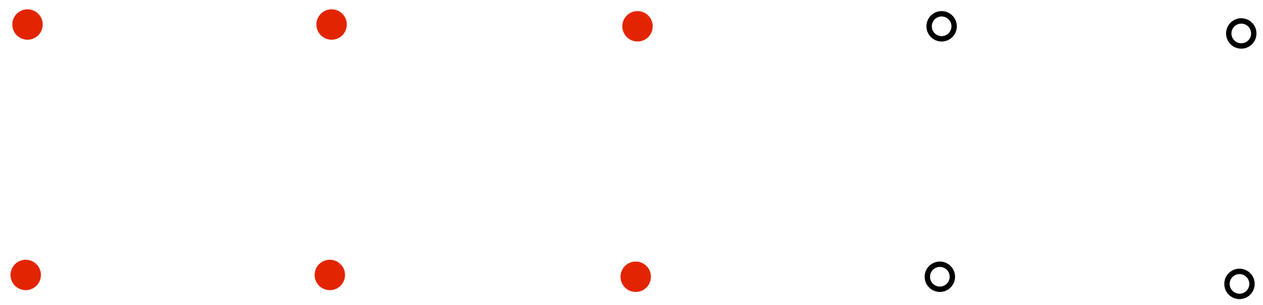
Each image sample sent to the display is converted into a little square of light of the appropriate color:
(a pixel = picture element)



LCD display pixel on my laptop

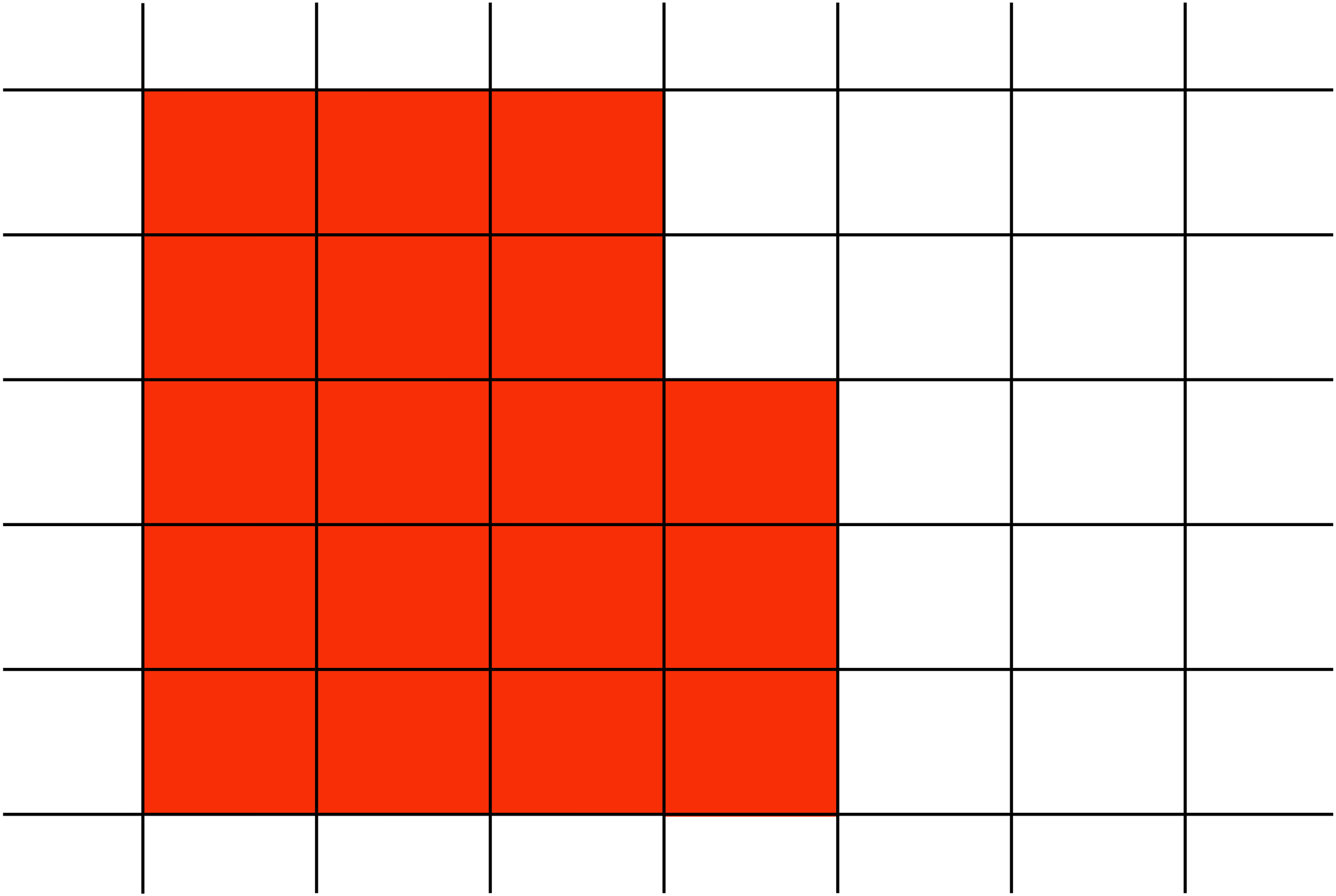
*** Thinking of each LCD pixel as emitting a square of uniform intensity light of a single color is a bit of an approximation to how real displays work, but it will do for now.**

So if we send the display this:



We see this when we look at the screen

(assuming a screen pixel emits a square of perfectly uniform intensity of light)

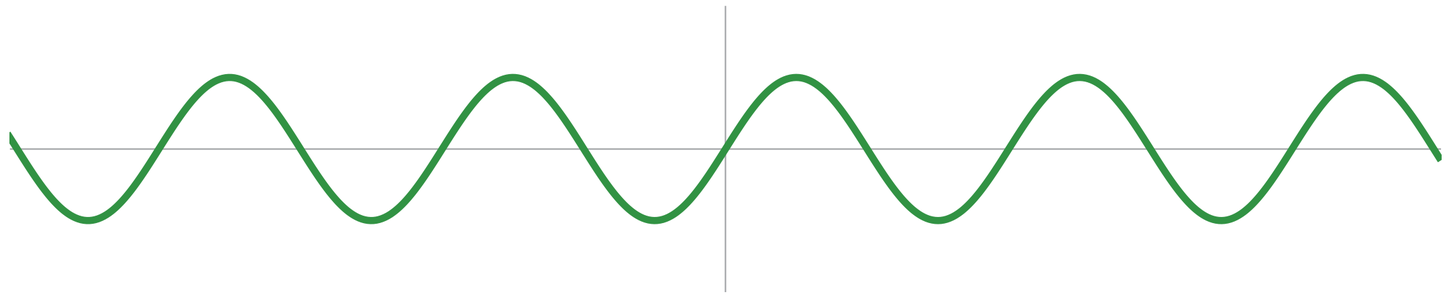


Recall: the real coverage signal was this

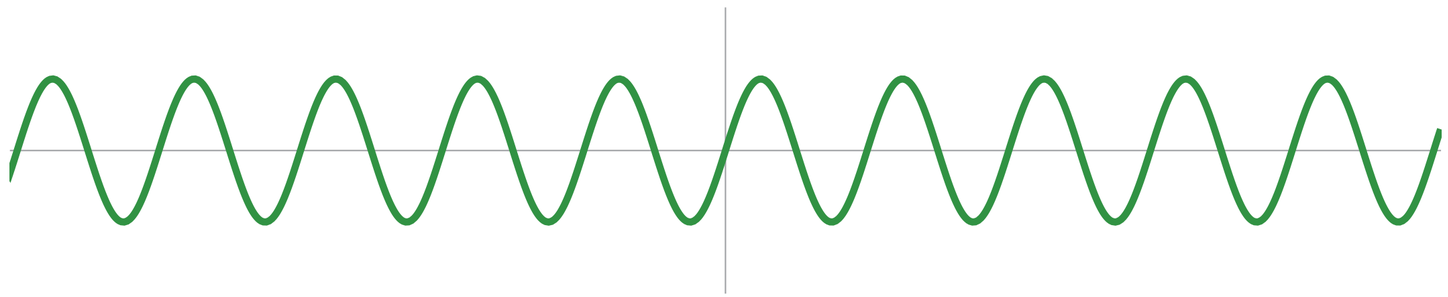
Aliasing

Example: sound can be expressed as a superposition of frequencies

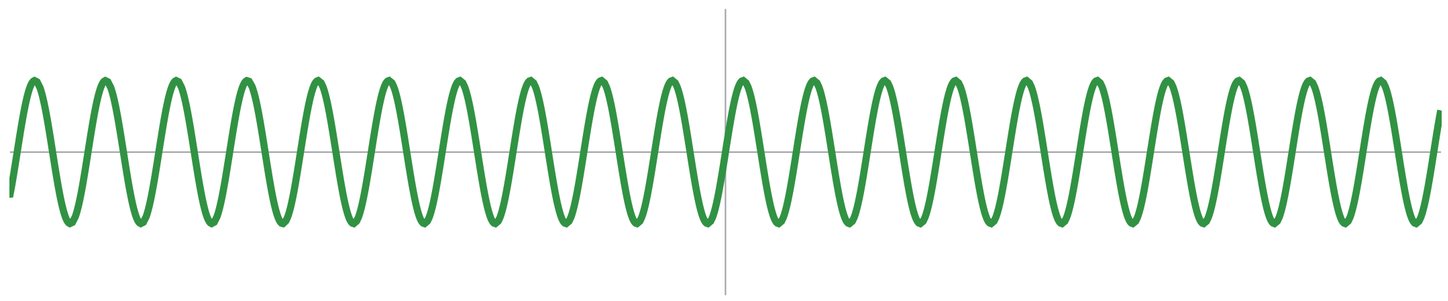
$f_1(x) = sin(\pi x)$



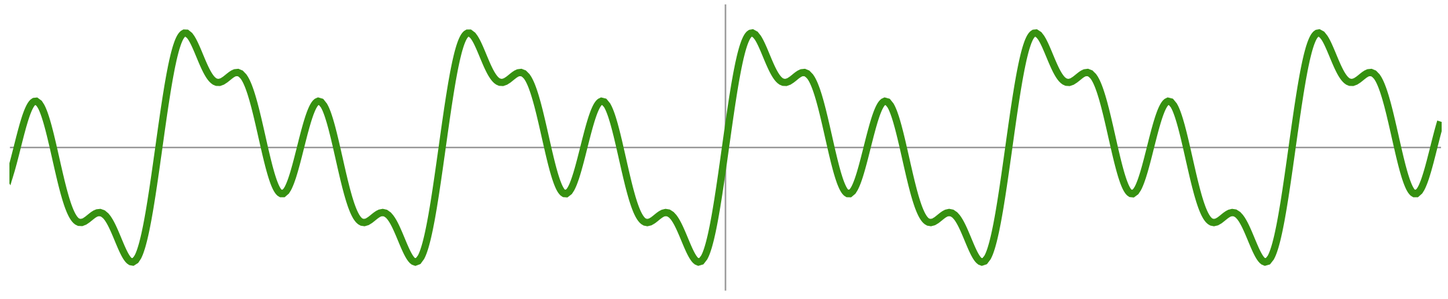
$f_2(x) = sin(2\pi x)$



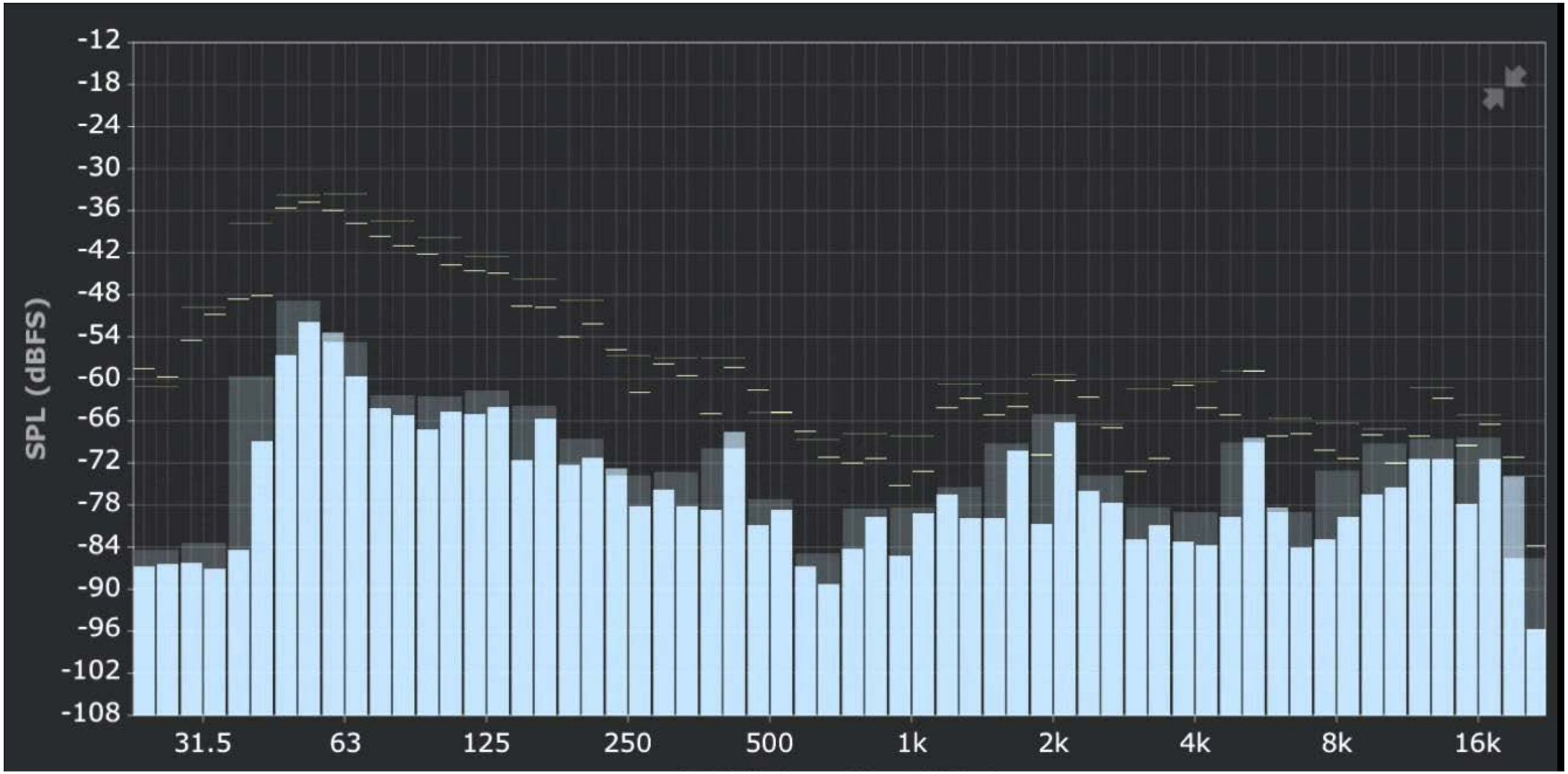
$f_4(x) = sin(4\pi x)$



$f(x) = f_1(x) + 0.75\, f_2(x) + 0.5\, f_4(x)$



An audio spectrum analyzer shows the amplitude of each frequency



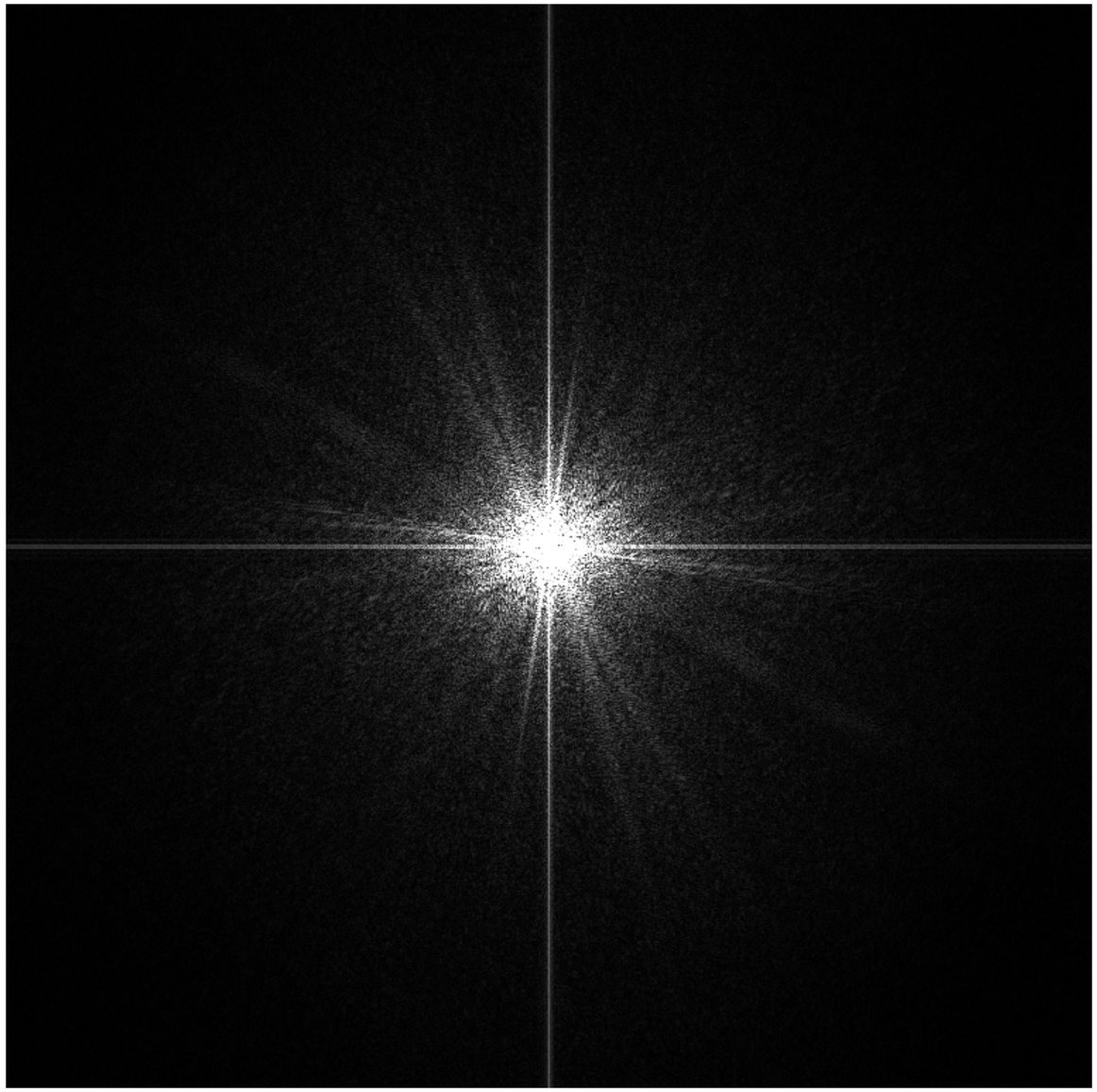
Intensity of low-frequencies (bass)

Intensity of high frequencies

Image credit: ONYX Apps

Images can also be decomposed into “frequencies”

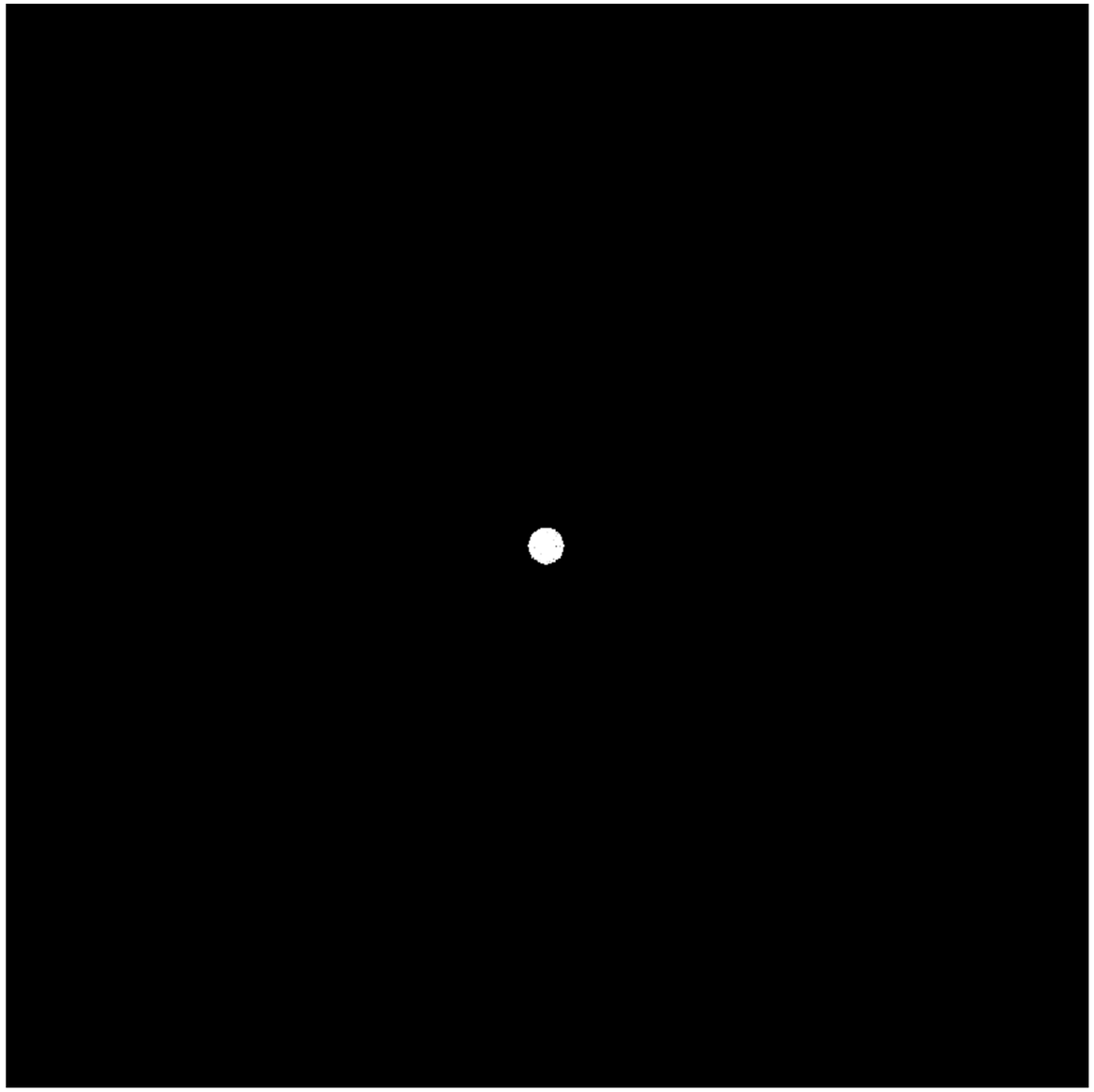
Spatial domain result



Spectrum

Low frequencies only (smooth gradients)

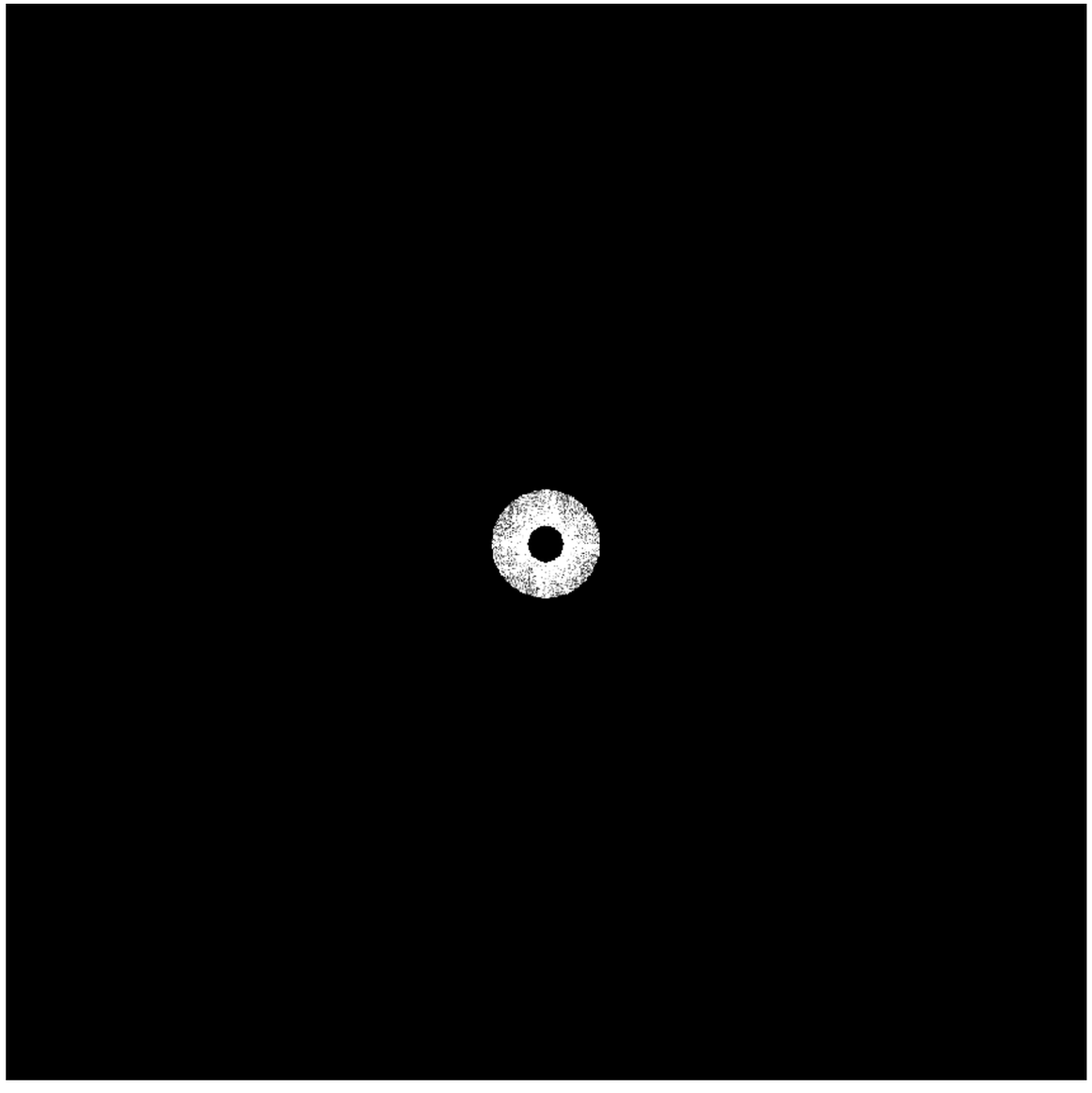
Spatial domain result



Spectrum (after low-pass filter)
All frequencies above cutoff have 0 magnitude

Mid-range frequencies

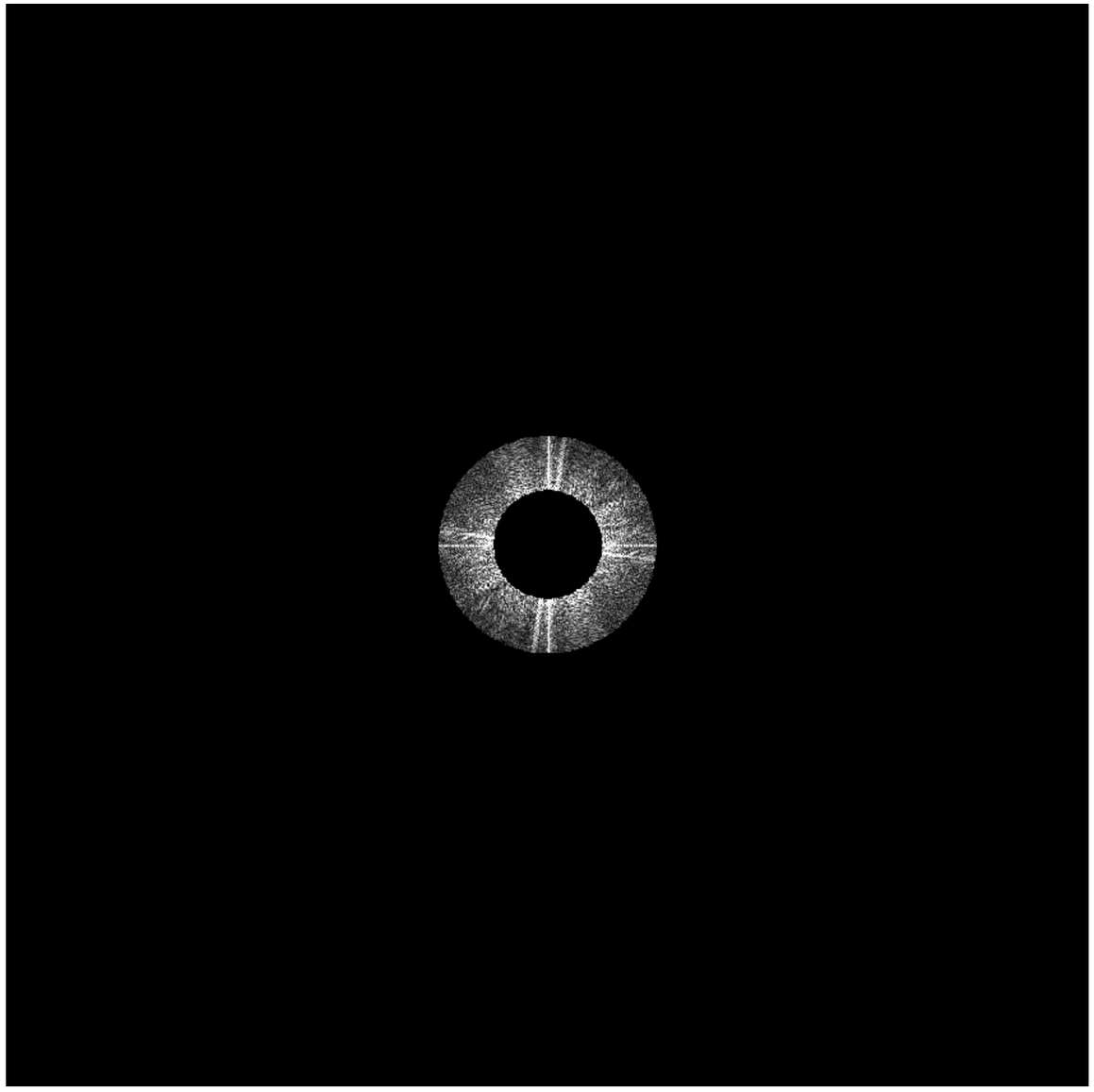
Spatial domain result



Spectrum (after band-pass filter)

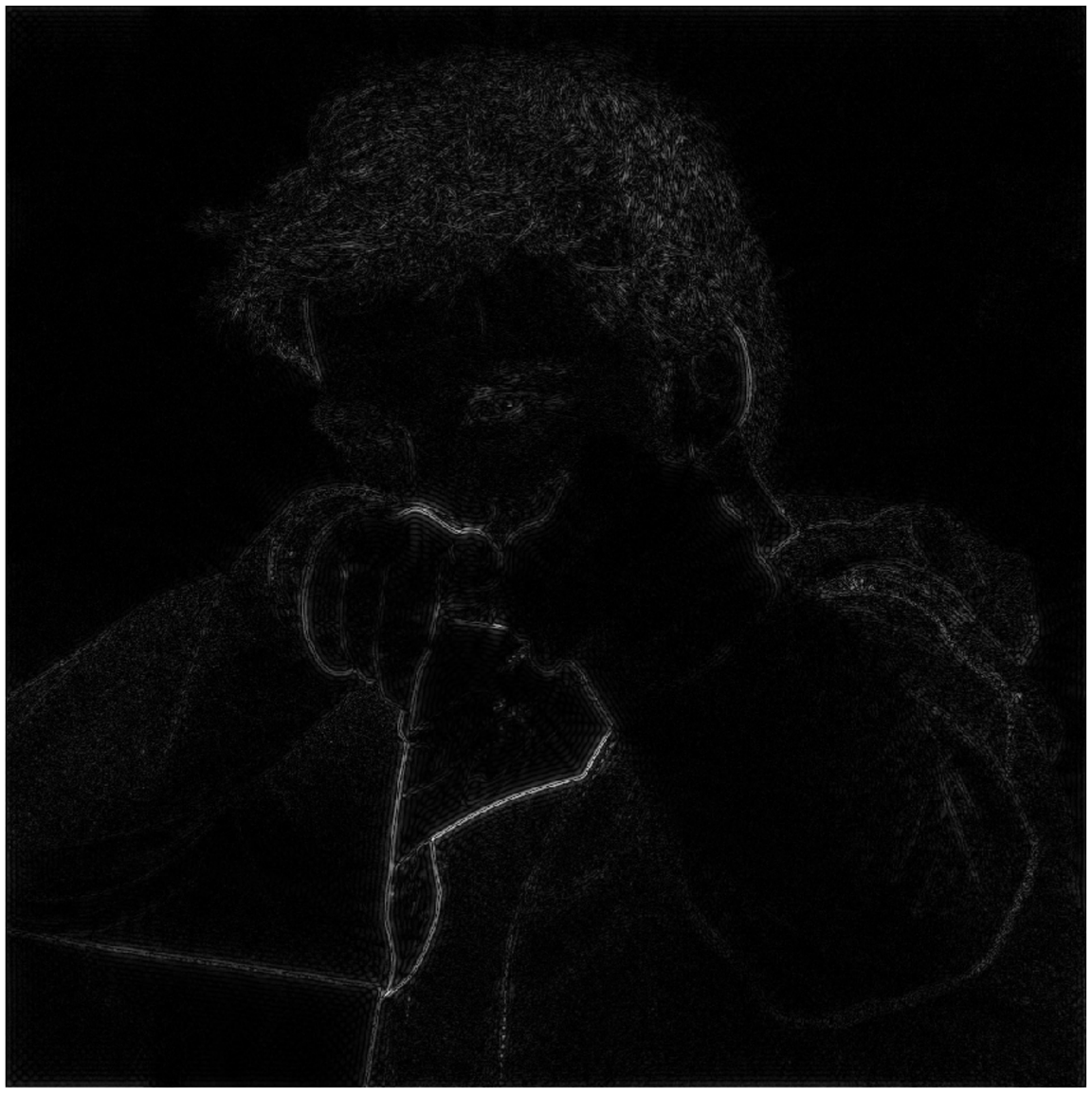
Mid-range frequencies

Spatial domain result

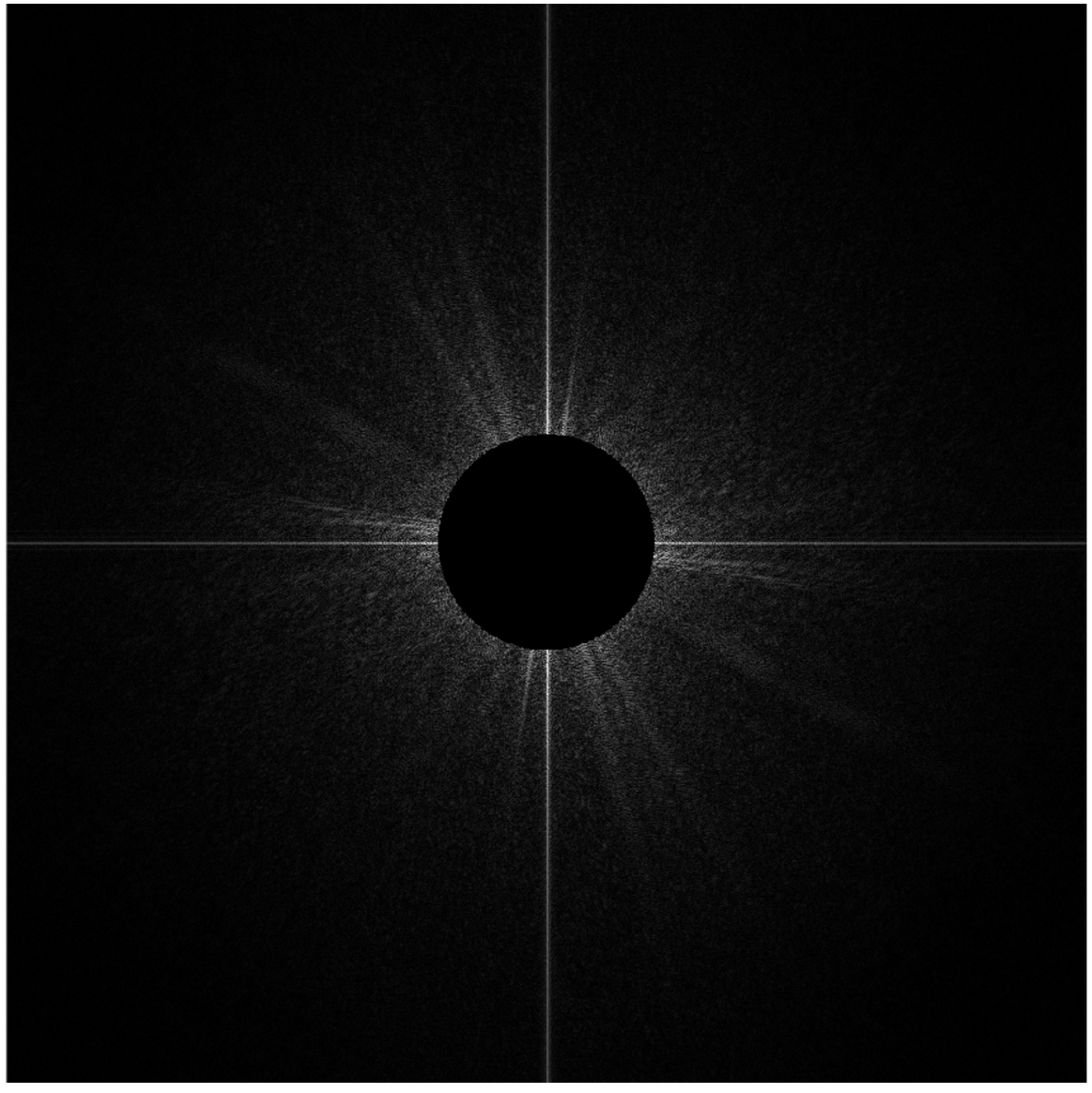


Spectrum (after band-pass filter)

High frequencies (edges)

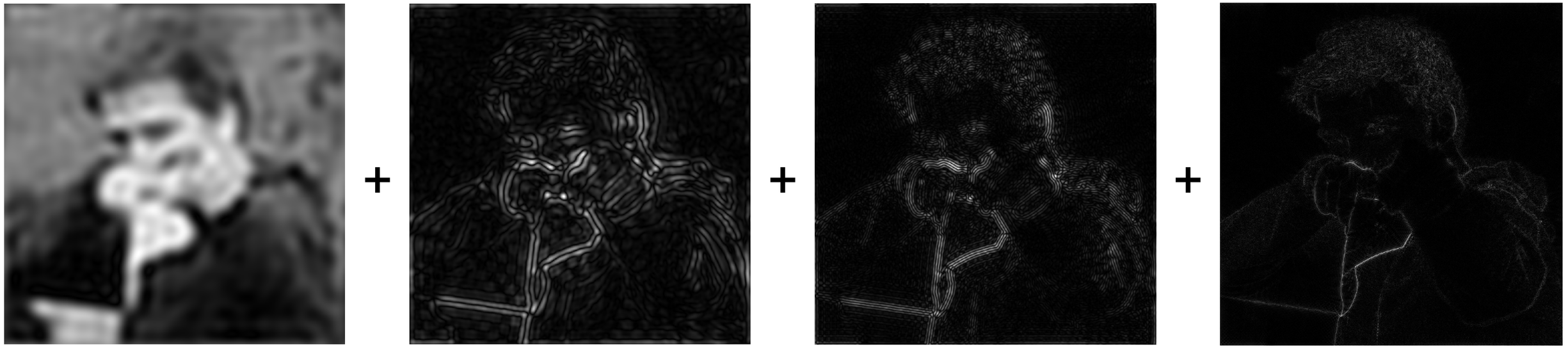


Spatial domain result
(strongest edges)



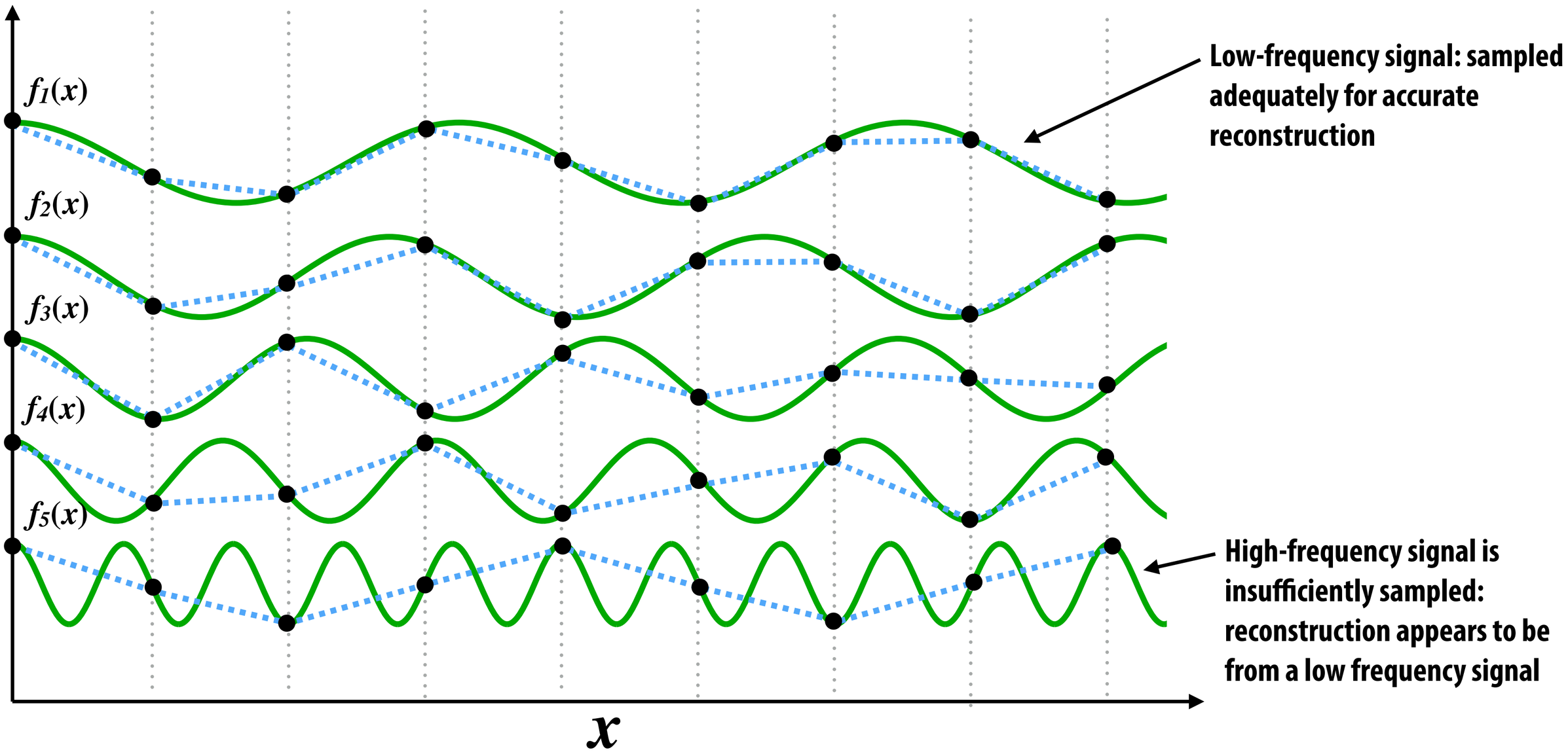
Spectrum (after high-pass filter)
All frequencies below threshold
have 0 magnitude

An image as a sum of its frequency components



=

1D example:
Undersampling high-frequency signals results in aliasing



"Aliasing": high frequencies in the original signal masquerade as low frequencies after reconstruction (due to undersampling)

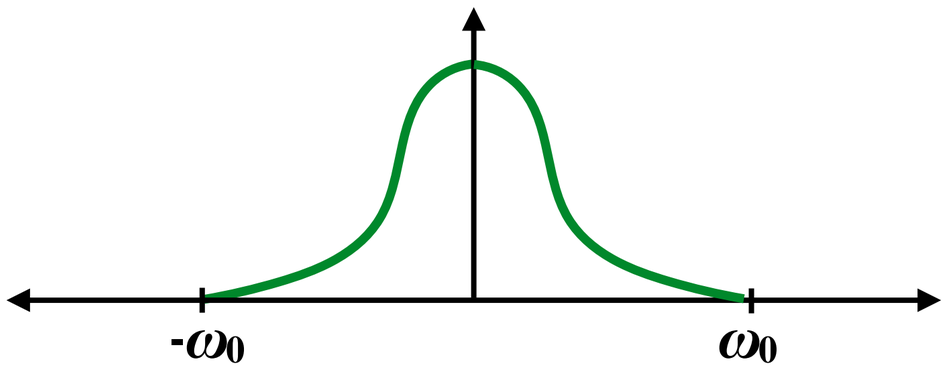
Temporal aliasing: wagon wheel effect

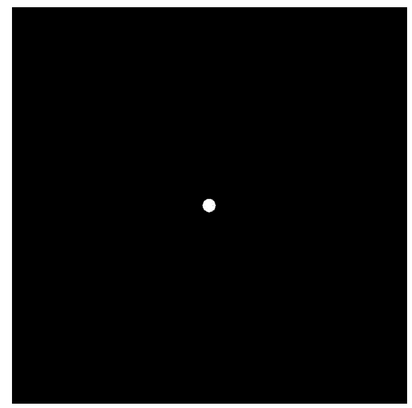
Camera's frame rate (temporal sampling rate) is too low for rapidly spinning wheel.

https://www.youtube.com/watch?v=VNftf5qLpiA

Nyquist-Shannon theorem

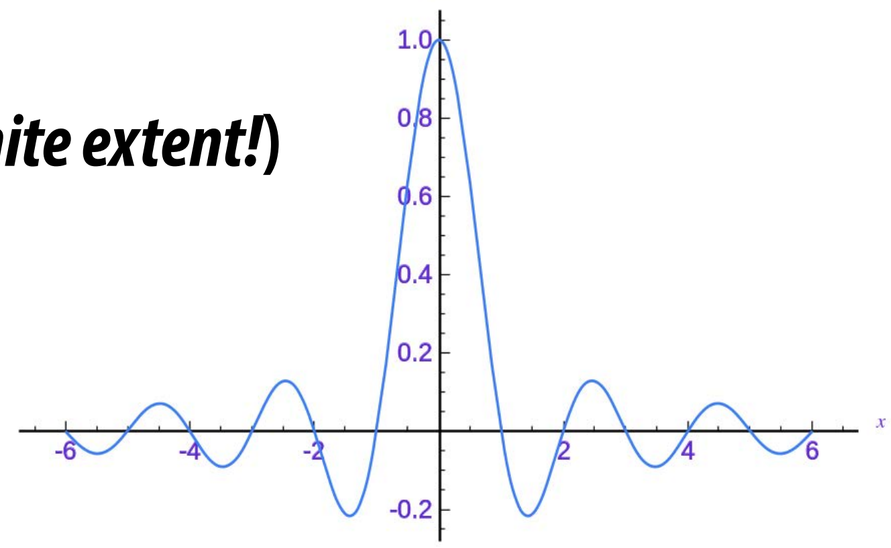
- Consider a band-limited signal: has no frequencies above ω_0
  - 1D: consider low-pass filtered audio signal
  - 2D: recall the blurred image example from a few slides ago





- The signal can be perfectly reconstructed if sampled with period $T = 1 / 2\omega_0$
- And interpolation is performed using a "*sinc filter*"
  - Ideal filter with no frequencies above cutoff (*infinite extent!*)

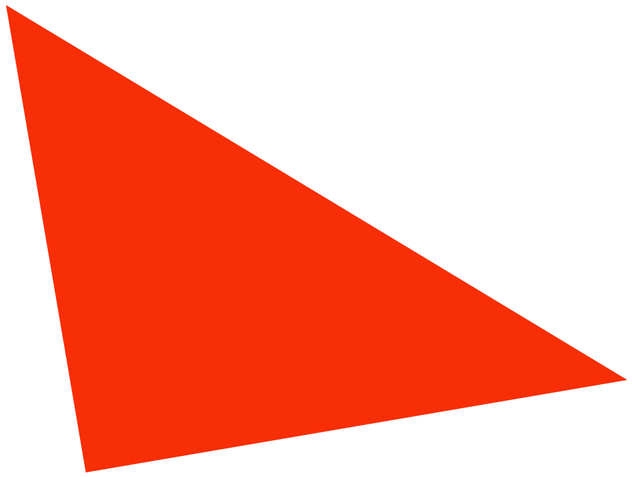
$$sinc(x) = \frac{sin(\pi x))}{\pi x}$$

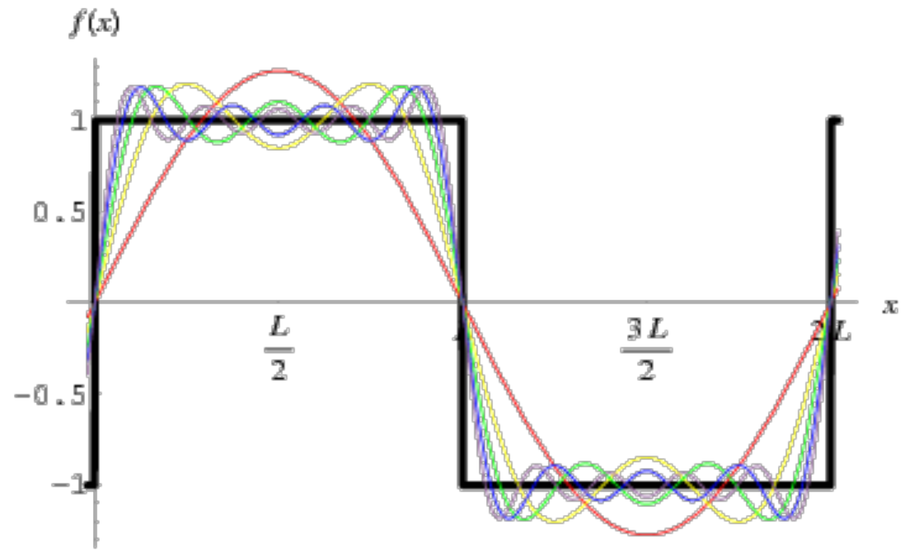


Challenges of sampling-based approaches in graphics

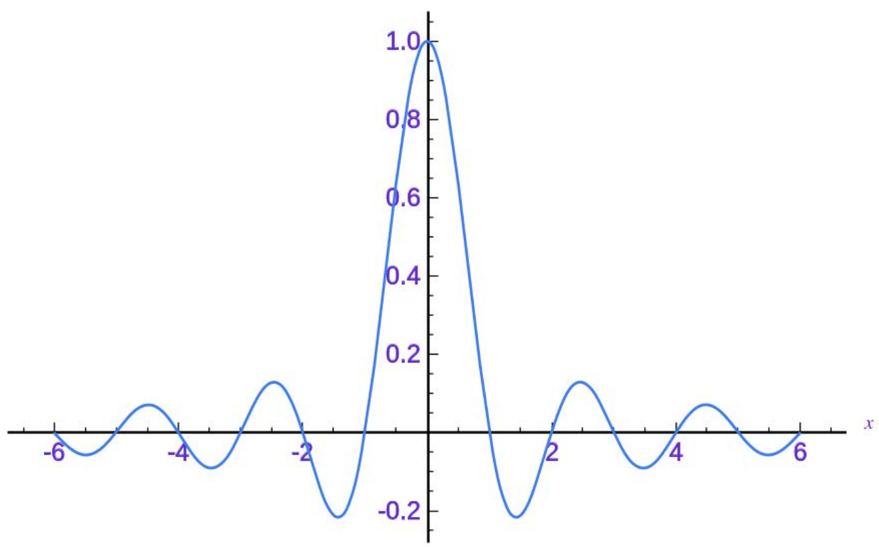
- **Our signals are not always band-limited in computer graphics. Why?**

Hint:



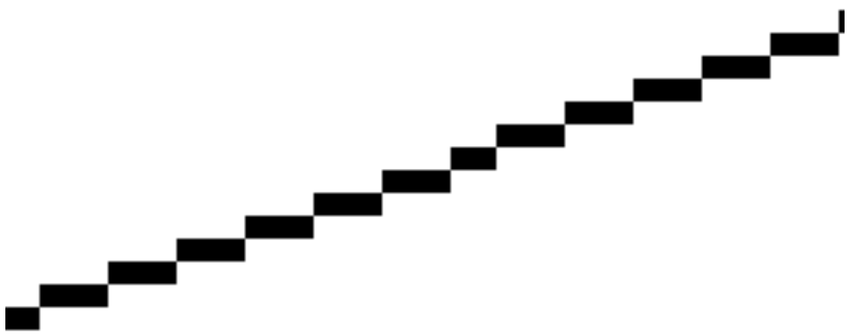


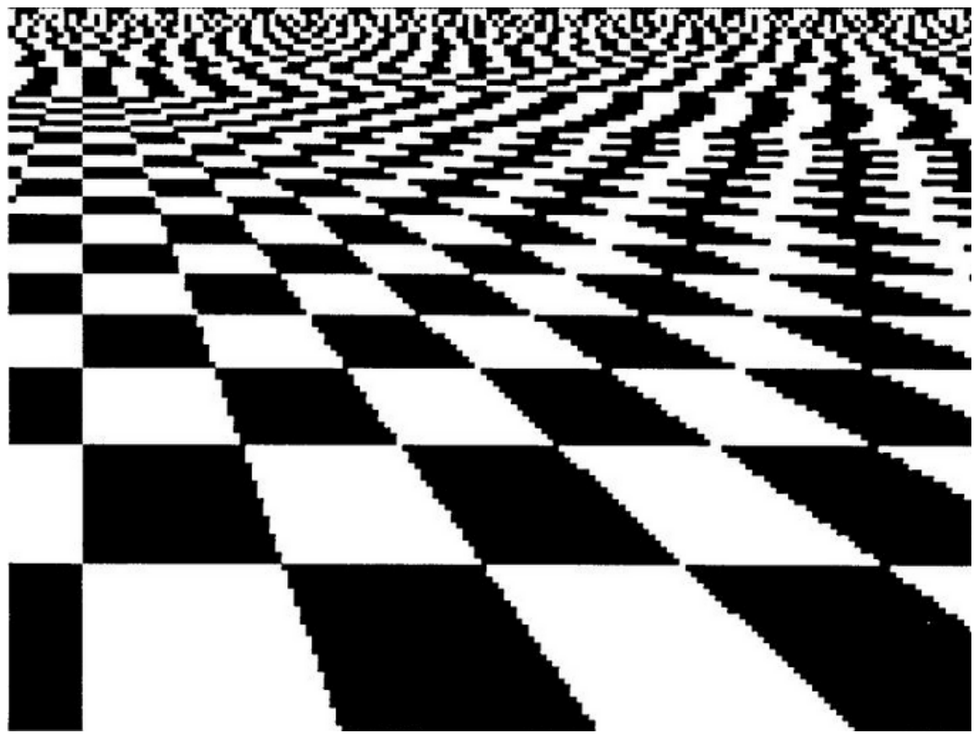
- **Also, infinite extent of "ideal" reconstruction filter (sinc) is impractical for efficient implementations. Why?**



Aliasing artifacts in images

- **Undersampling high-frequency signals and the use of non-ideal resampling filters yields image artifacts**
  - **“Jaggies” in a single image**
  - **“Roping” or “shimmering” of images when animated**
  - **Moiré patterns in high-frequency areas of images**

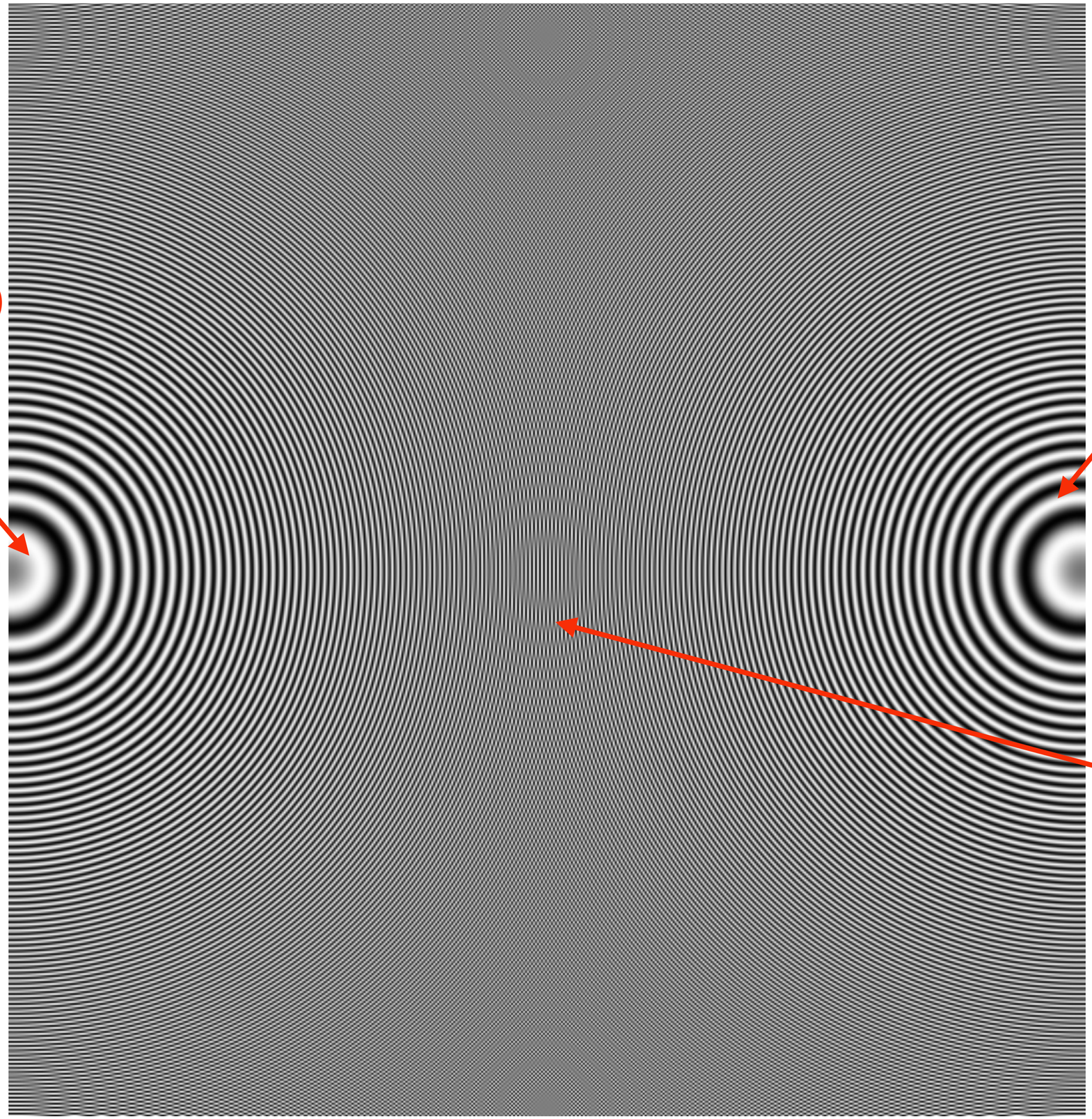




Sampling a zone plate: $\sin(x^2 + y^2)$

Rings in center-left: Actual signal (low frequency oscillation)

(0,0)

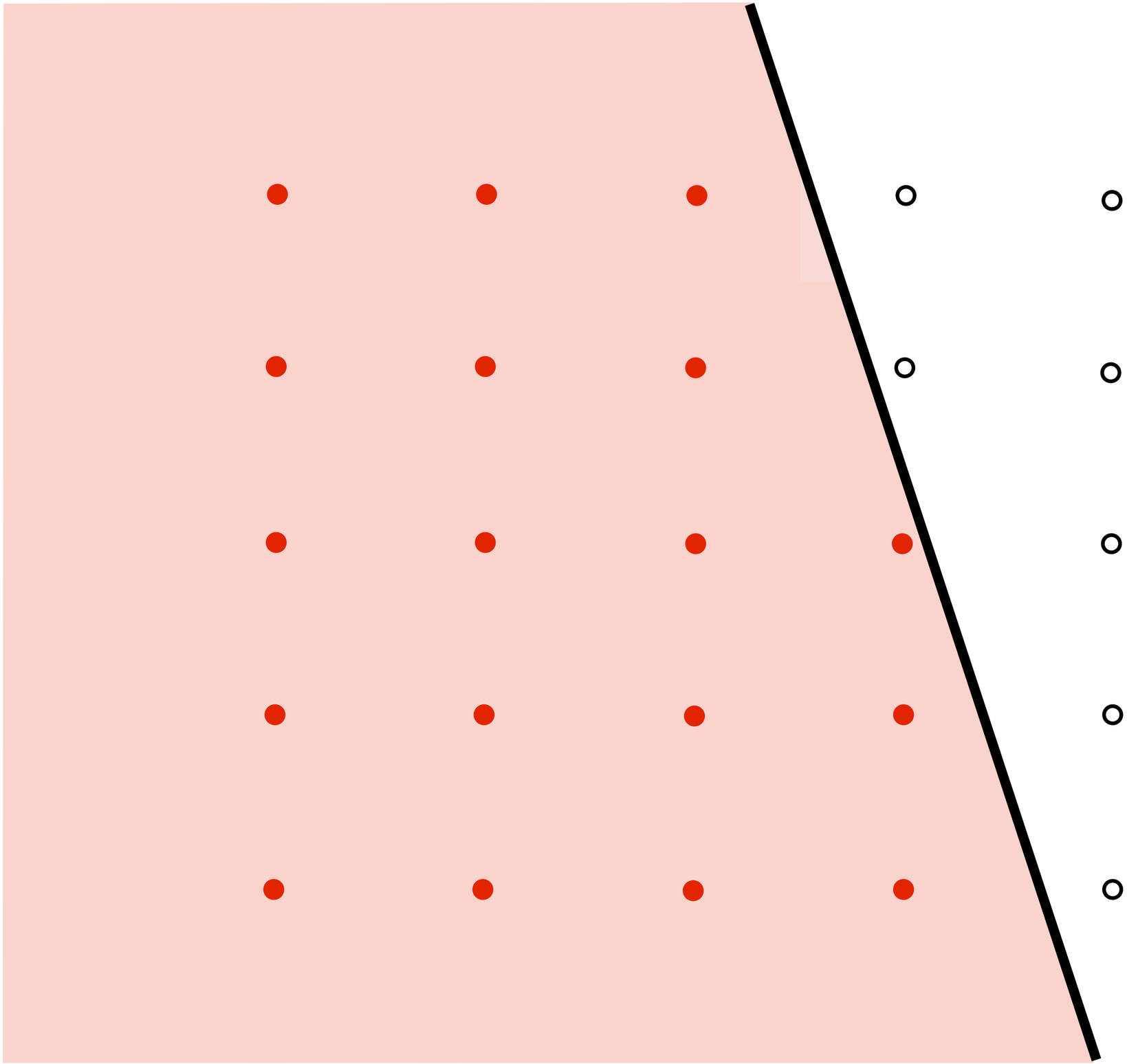


Rings on right: aliasing from undersampling high frequency oscillation and then resampling back to Keynote slide resolution

Middle: (interaction between actual signal and aliased resconstruction)

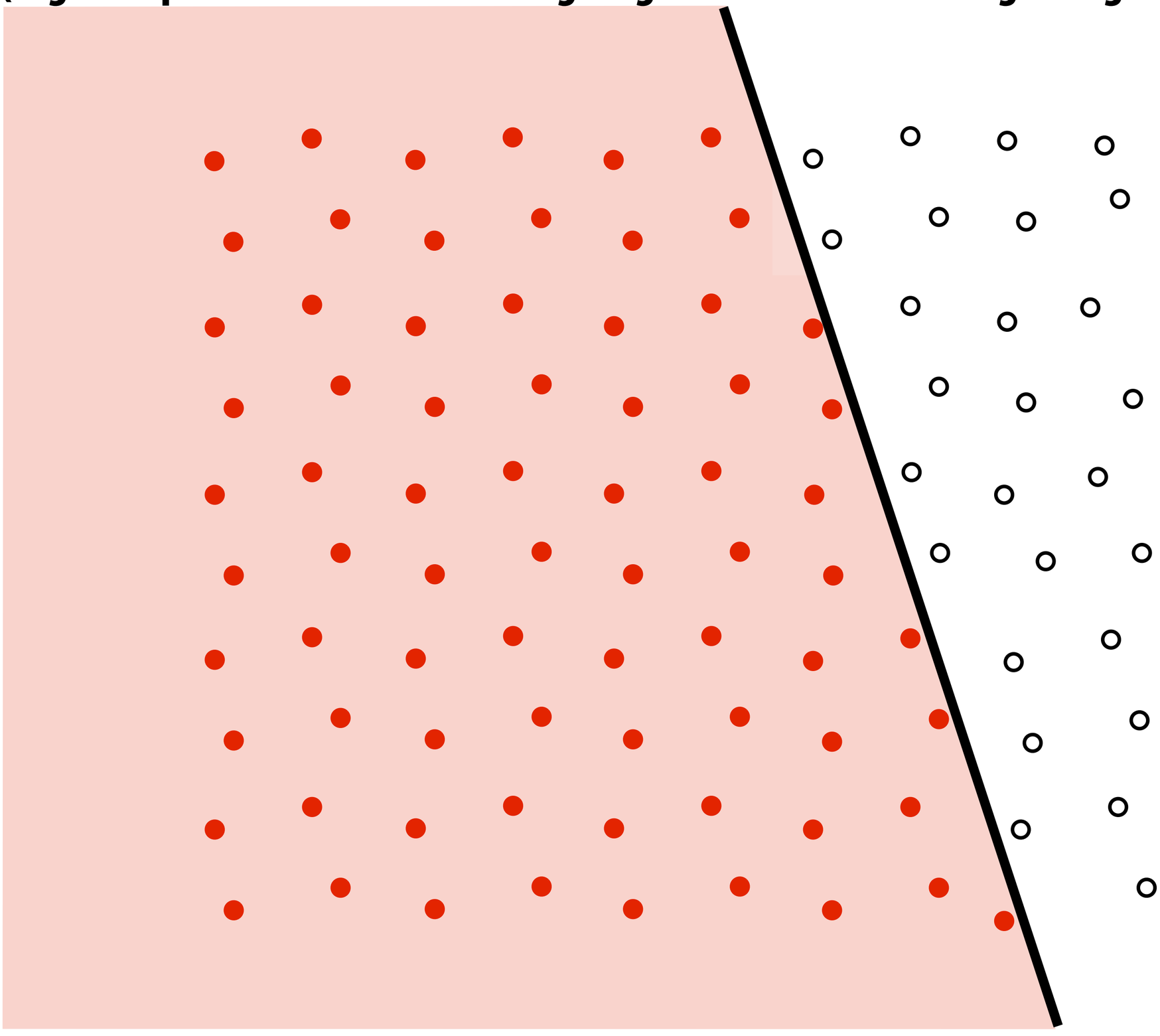
Figure credit: Pat Hanrahan and Bryce Summers

Initial coverage sampling rate (1 sample per pixel)



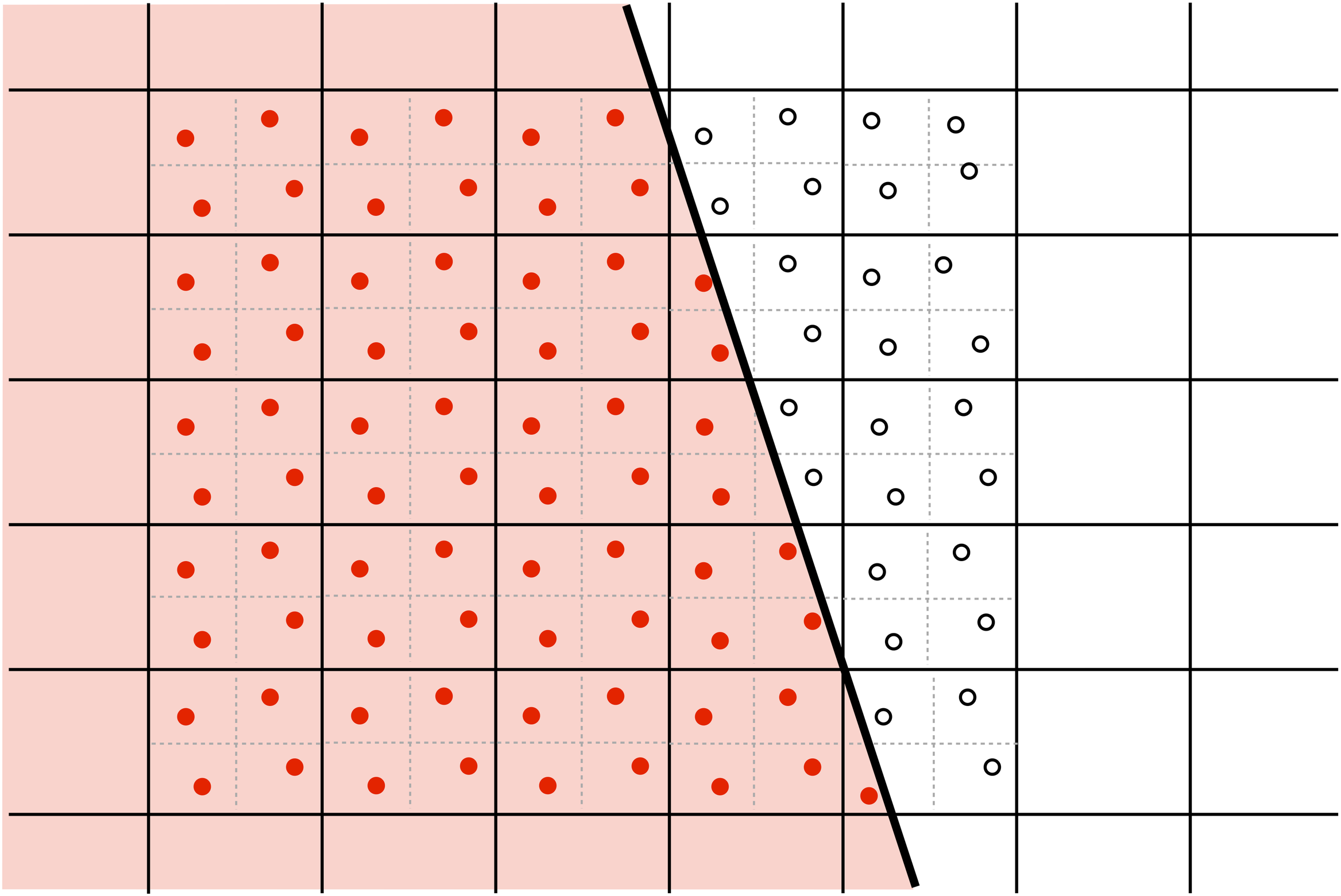
Increase density of sampling coverage signal

(high frequencies exist in coverage signal because of triangle edges)



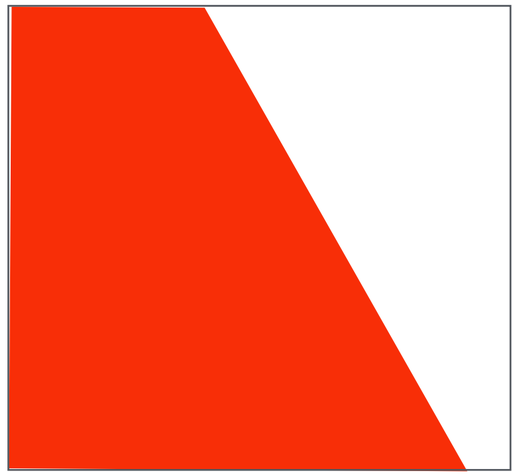
Supersampling

Example: stratified sampling using four samples per pixel

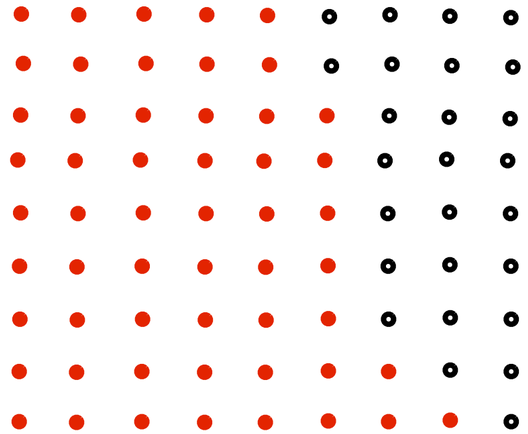


Resampling

Converting from one discrete sampled representation to another

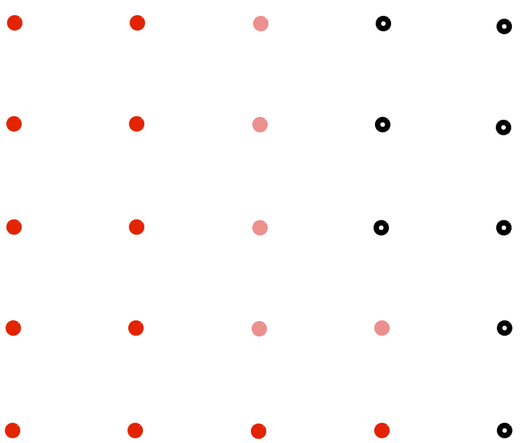


Original signal
(high frequency edge)



Dense sampling of
reconstructed signal

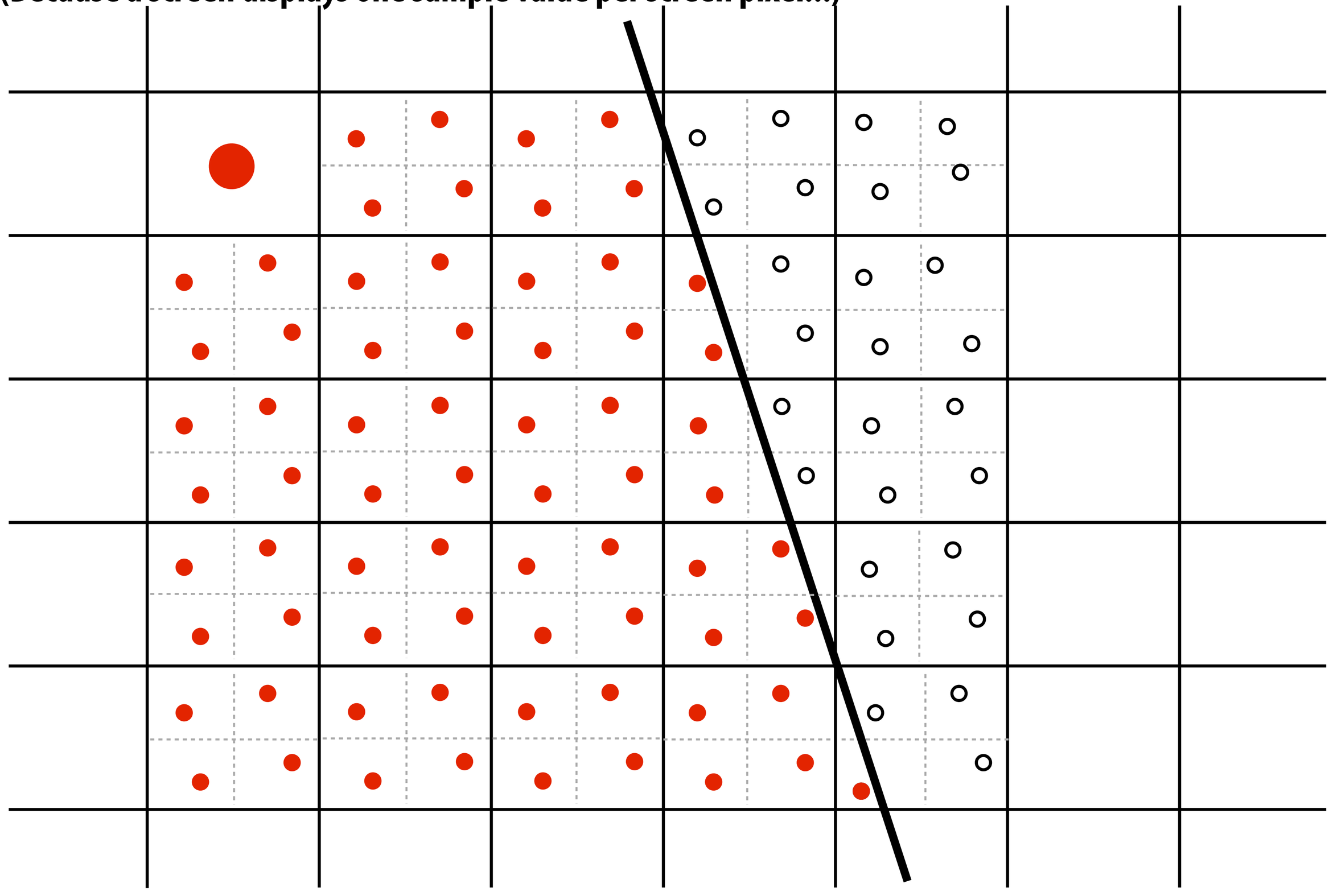
Reconstructed signal
(lacks high frequencies)



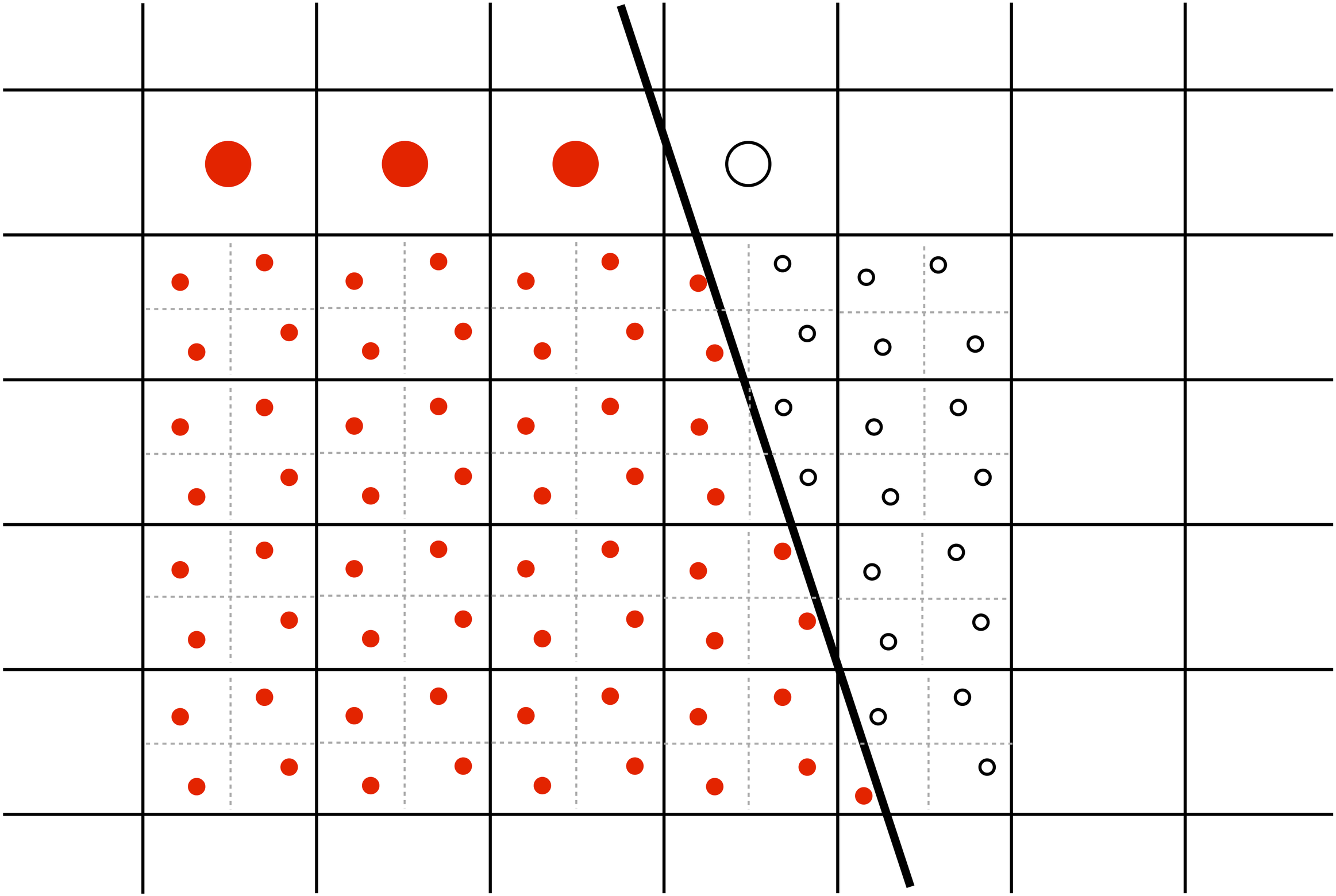
Coarsely sampled signal

Resample to display's pixel resolution

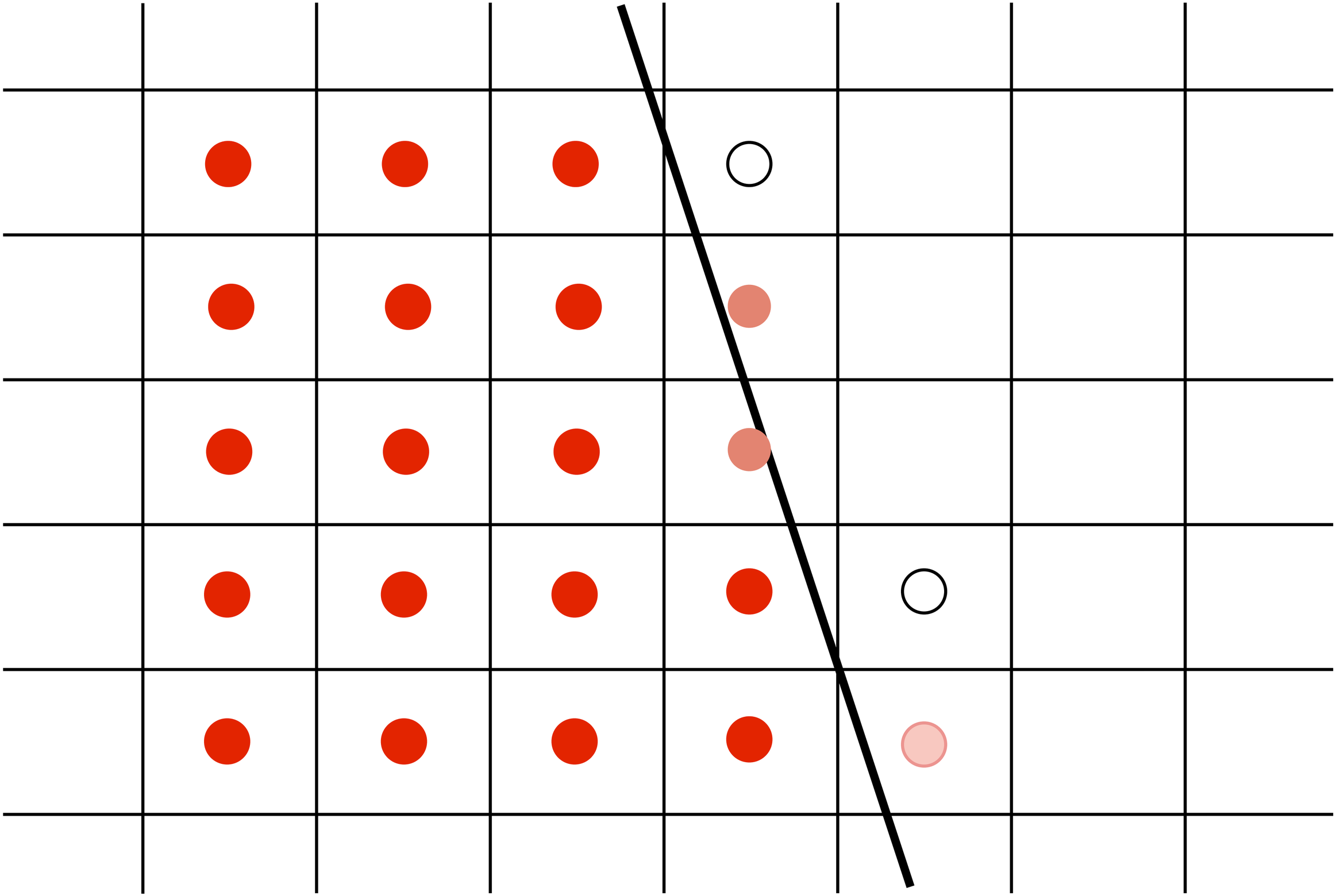
(Because a screen displays one sample value per screen pixel...)



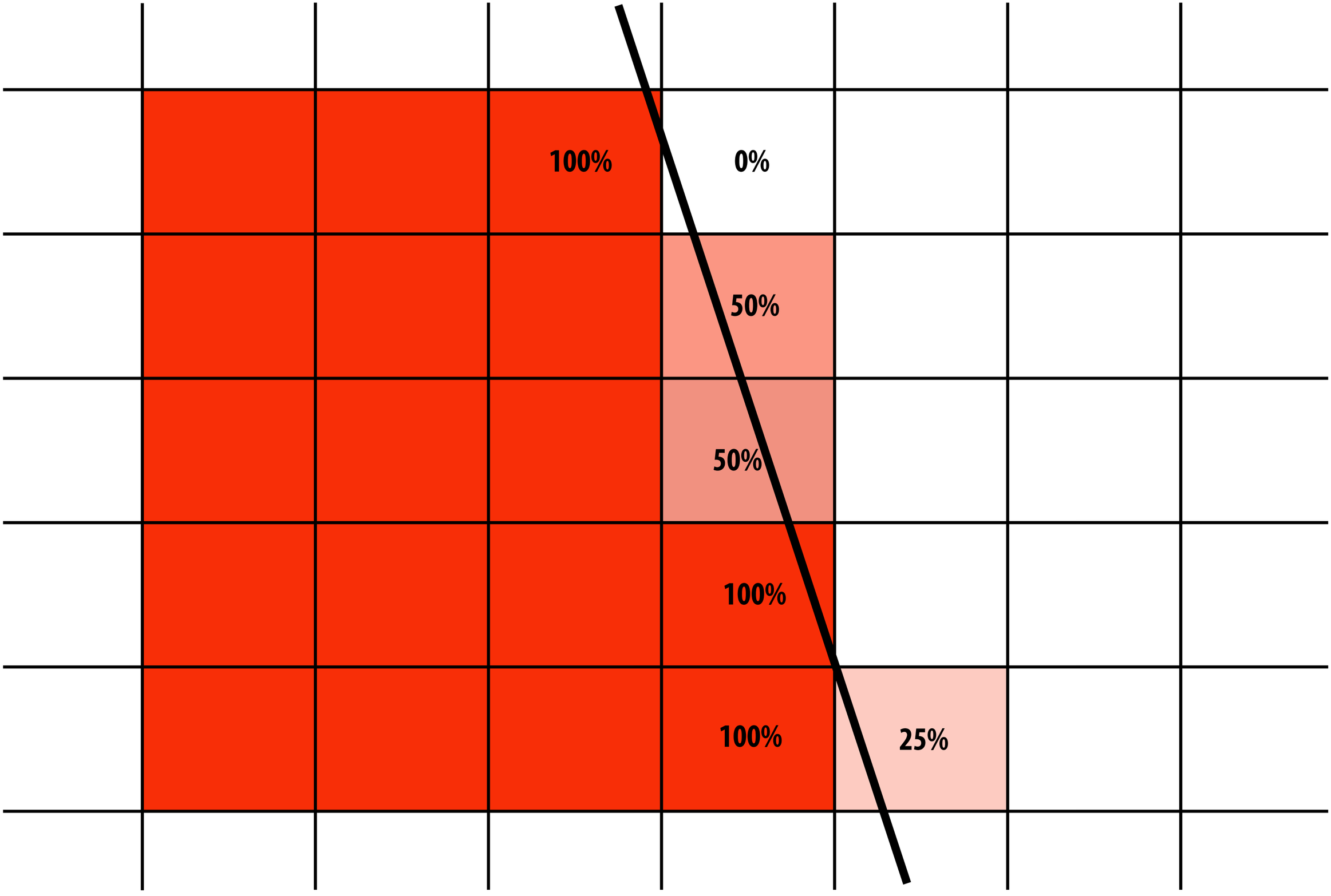
Resample to display's pixel rate (box filter)



Resample to display's pixel rate (box filter)



Displayed result (note anti-aliased edges)



Recall: the real coverage signal was this

Sampling coverage

- **We want the light emitted from a display to be an accurate to match the ground truth signal:** coverage(x,y))

- **Resampling a densely sampled signal (supersampled) integrates coverage values over the entire pixel region. The integrated result is sent to the display (and emitted by the pixel) so that the light emitted by the pixel is similar to what would be emitted in that screen region by an "infinite resolution display"**

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

Point-in-triangle test

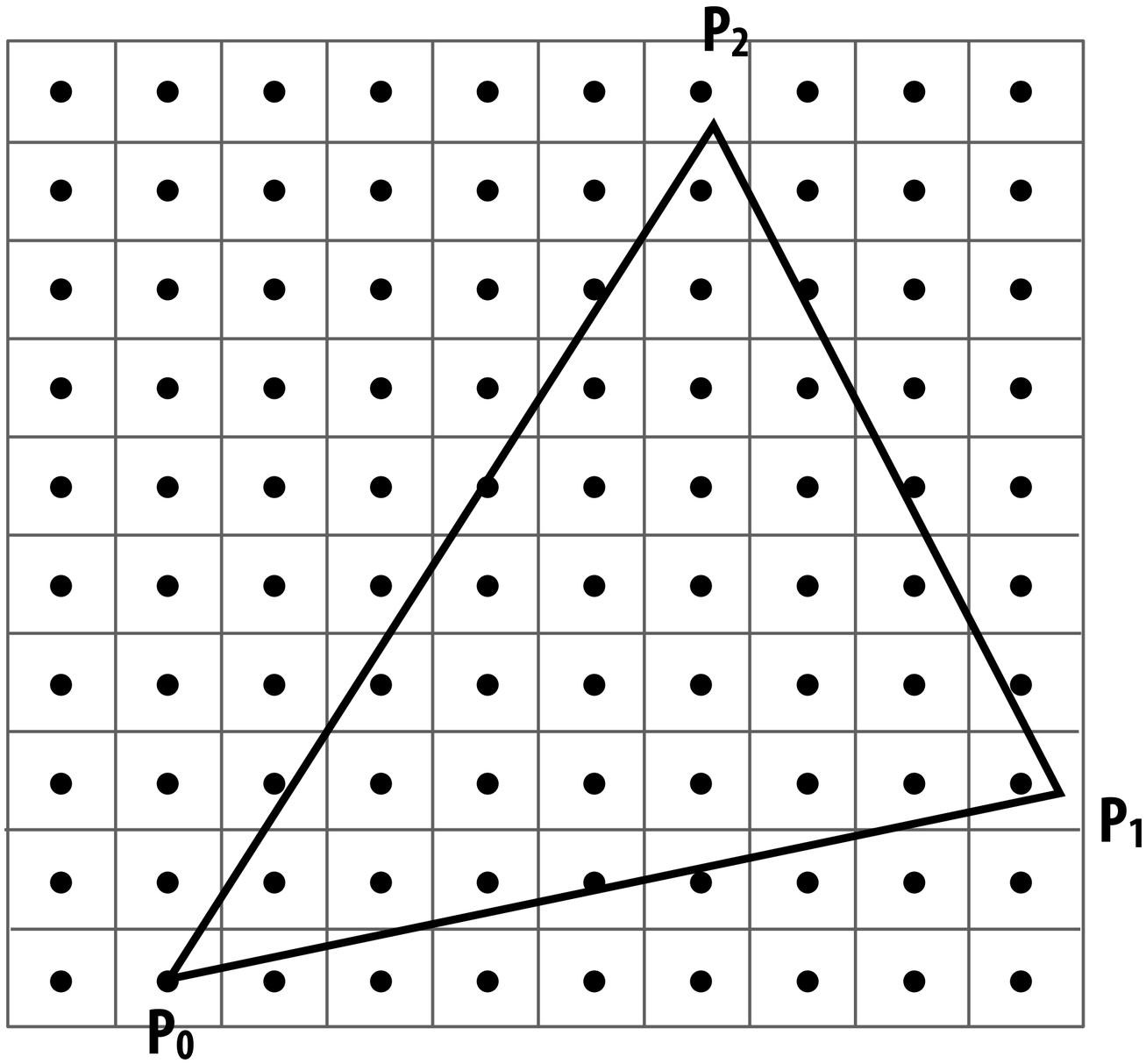
Compute triangle edge equations from projected positions of vertices

$$P_i = (X_i, Y_i)$$

$$dX_i = X_{i+1} - X_i$$
$$dY_i = Y_{i+1} - Y_i$$

$$E_i(x, y) = (x - X_i)\, dY_i - (y - Y_i)\, dX_i$$
$$= A_i x + B_i y + C_i$$

$E_i(x, y) = 0$: point on edge
> 0 : outside edge
< 0 : inside edge



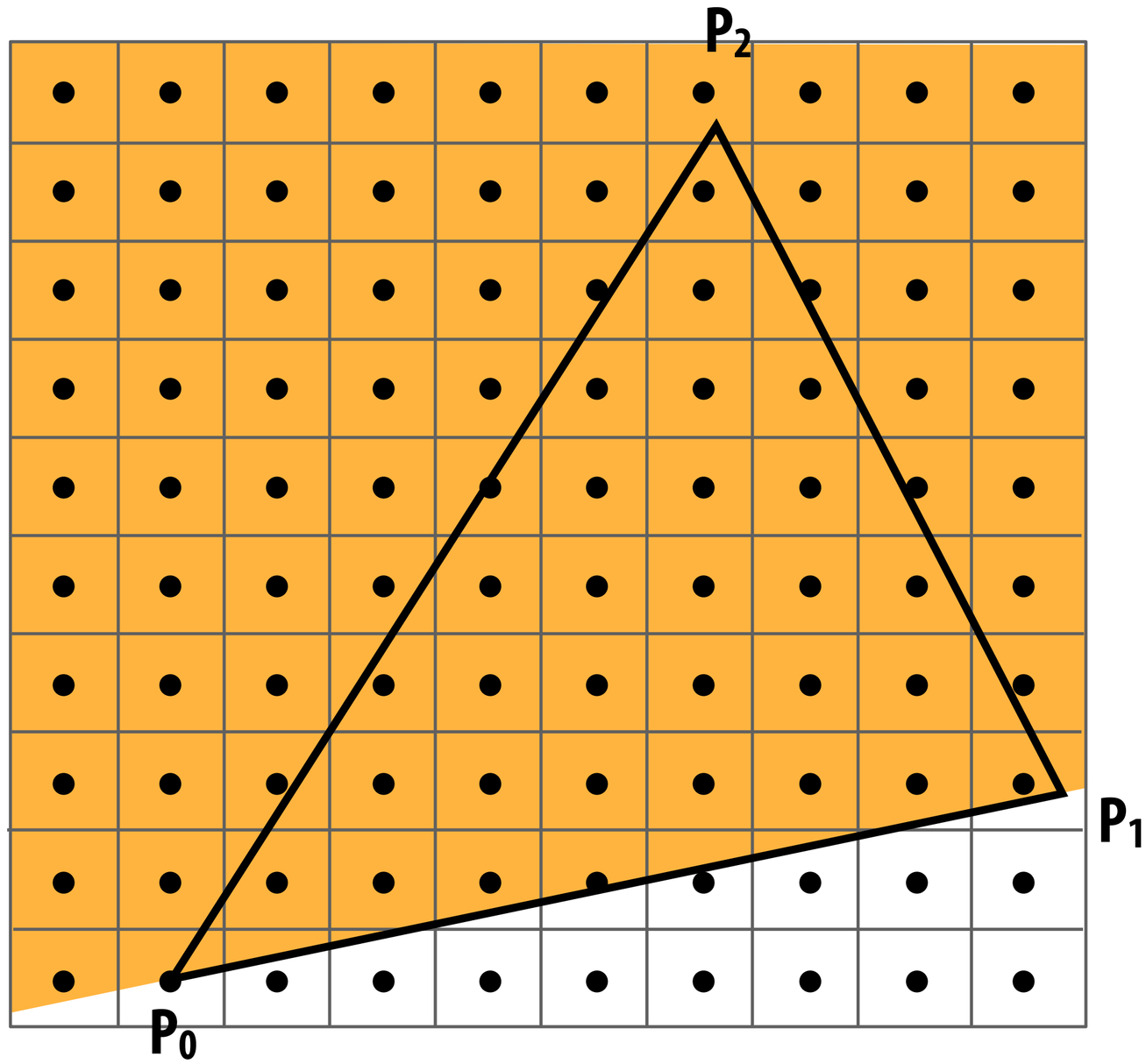
Point-in-triangle test

$$P_i = (X_i, Y_i)$$

$$dX_i = X_{i+1} - X_i$$
$$dY_i = Y_{i+1} - Y_i$$

$$E_i(x, y) = (x - X_i)\, dY_i - (y - Y_i)\, dX_i$$
$$= A_i\, x + B_i\, y + C_i$$

$E_i(x, y) = 0$: point on edge
> 0 : outside edge
< 0 : inside edge



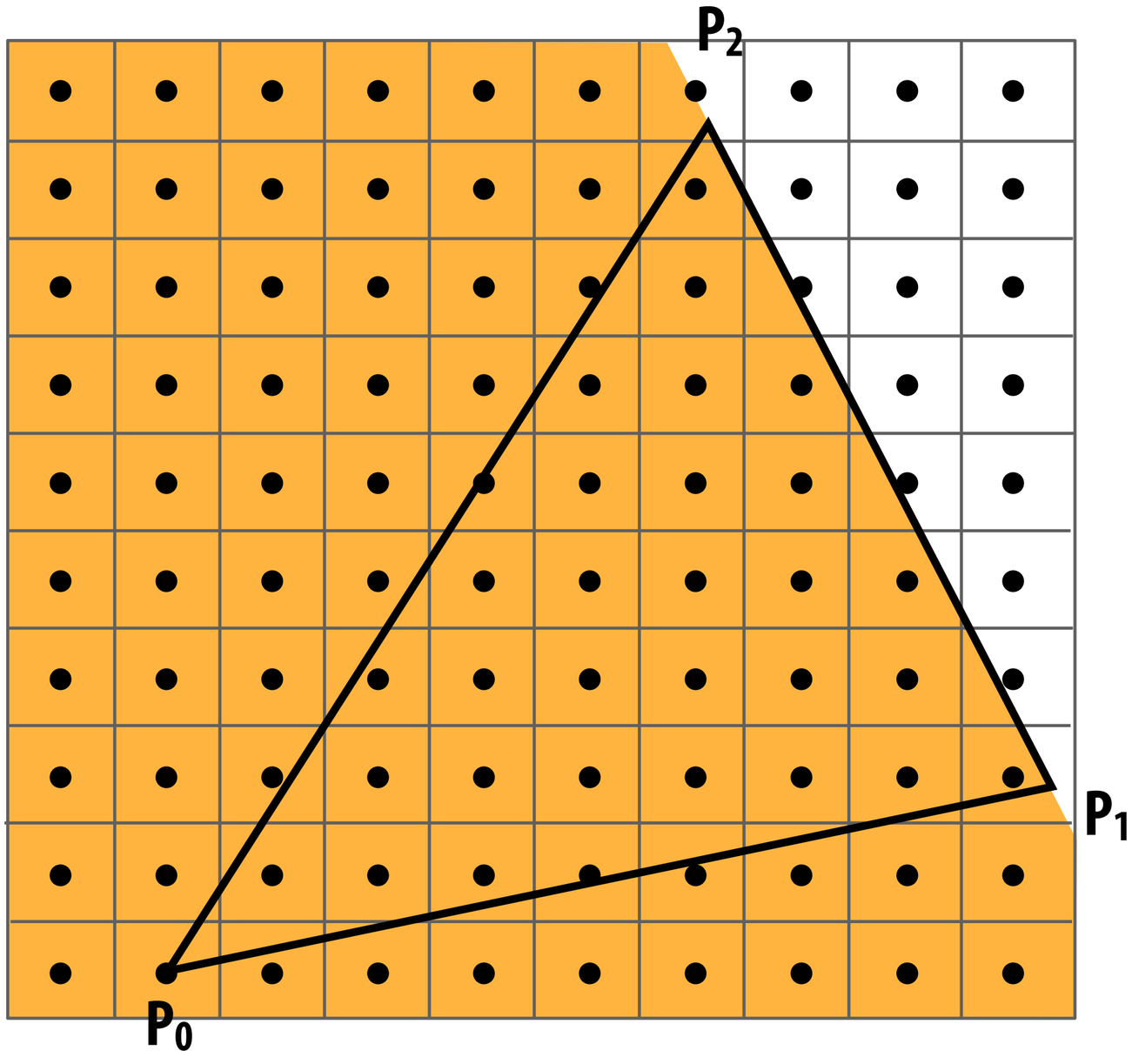
Point-in-triangle test

$$P_i = (X_i, Y_i)$$

$$dX_i = X_{i+1} - X_i$$
$$dY_i = Y_{i+1} - Y_i$$

$$E_i(x, y) = (x - X_i)\, dY_i - (y - Y_i)\, dX_i$$
$$= A_i\, x + B_i\, y + C_i$$

$E_i(x, y) = 0$: point on edge
> 0 : outside edge
< 0 : inside edge



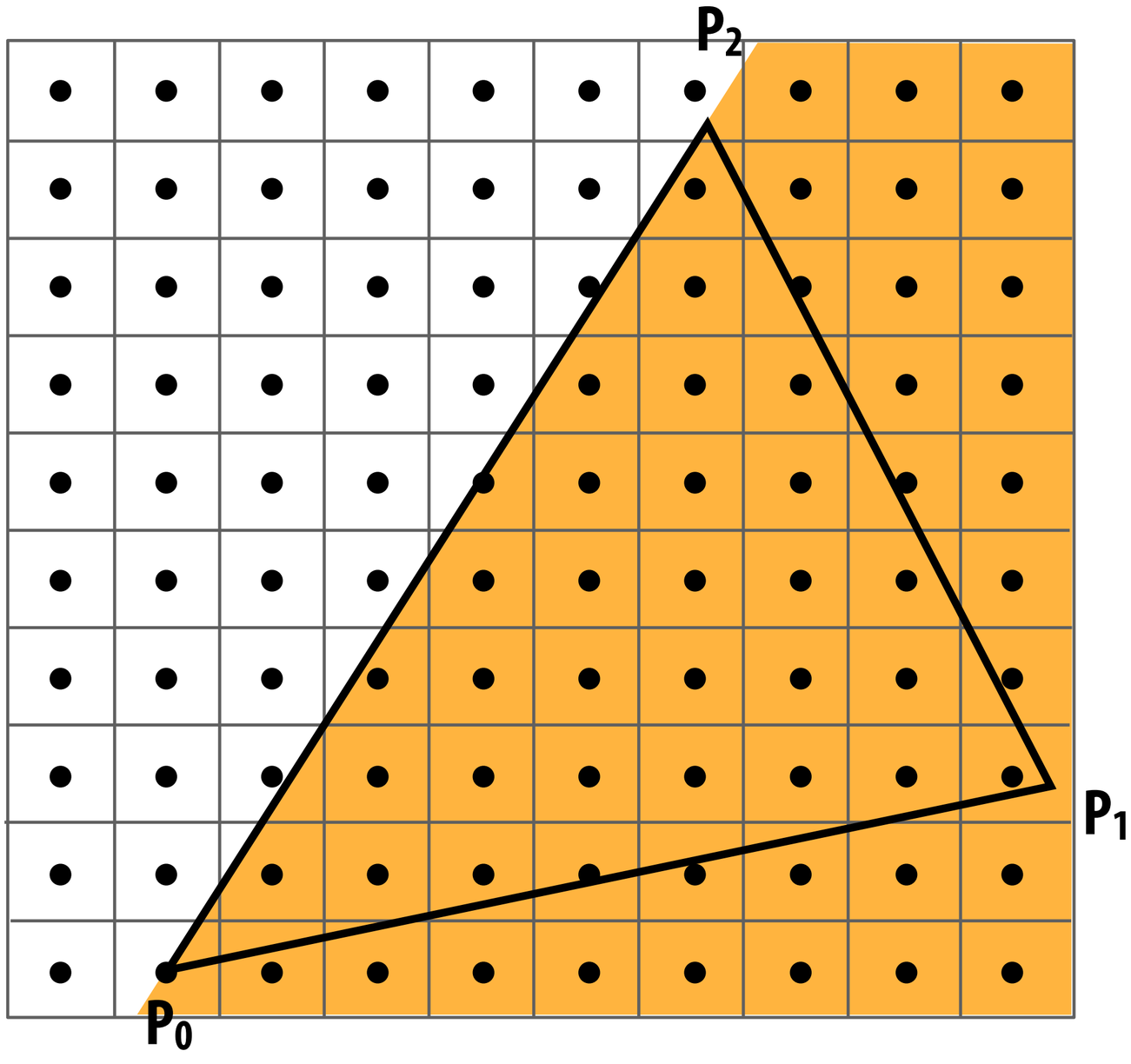
Point-in-triangle test

$$P_i = (X_i, Y_i)$$

$$dX_i = X_{i+1} - X_i$$
$$dY_i = Y_{i+1} - Y_i$$

$$E_i(x, y) = (x - X_i)\, dY_i - (y - Y_i)\, dX_i$$
$$= A_i\, x + B_i\, y + C_i$$

$E_i(x, y) = 0$: point on edge
> 0 : outside edge
< 0 : inside edge

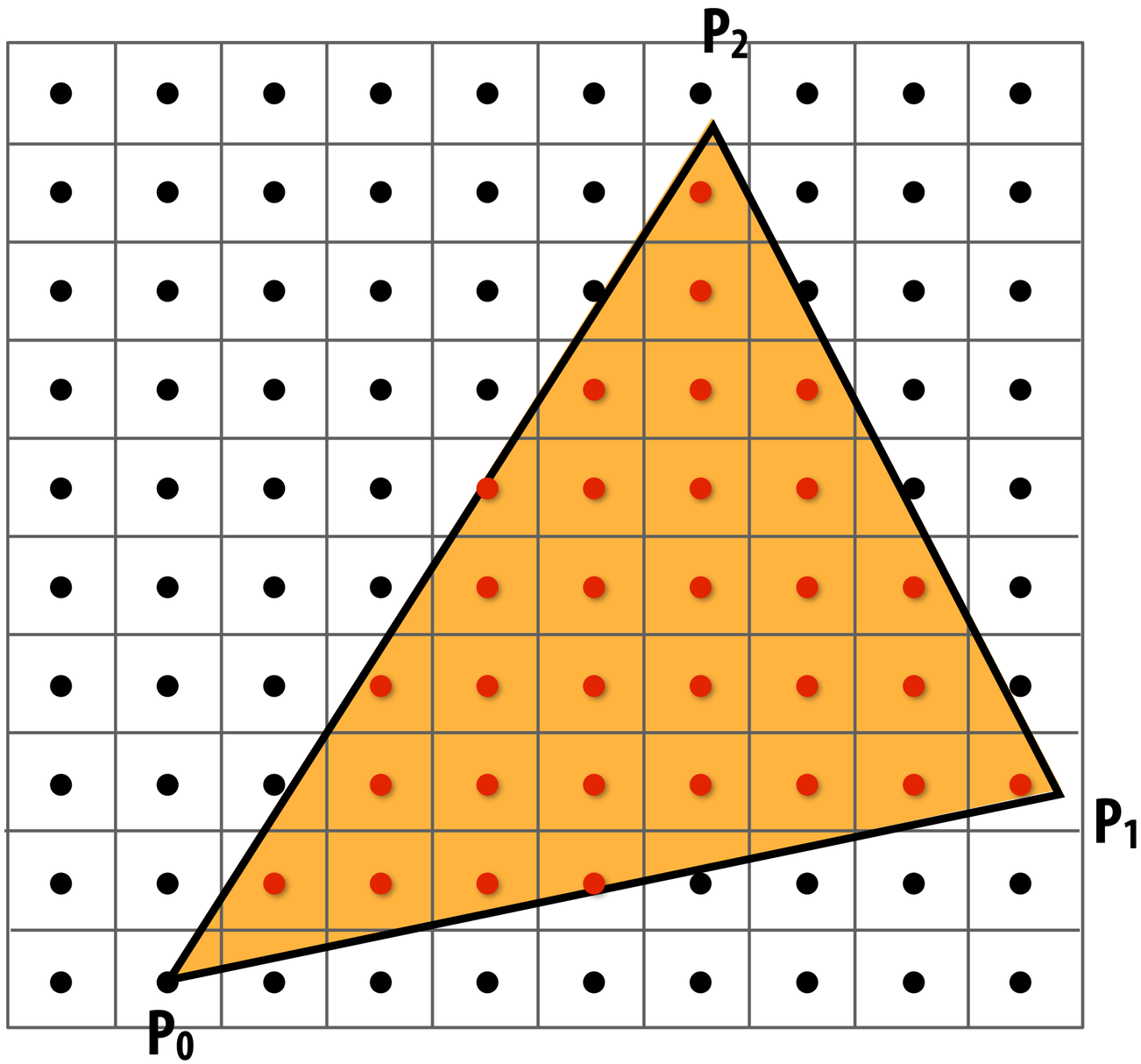


Point-in-triangle test

Sample point $s = (sx, sy)$ is inside the triangle if it is inside all three edges.

$$inside(sx, sy) =$$
$$E_0(sx, sy) < 0 \;\&\&$$
$$E_1(sx, sy) < 0 \;\&\&$$
$$E_2(sx, sy) < 0;$$

Note: actual implementation of $inside(sx,sy)$ involves ≤ checks based on the triangle coverage edge rules (see beginning of lecture)



Sample points inside triangle are highlighted red.

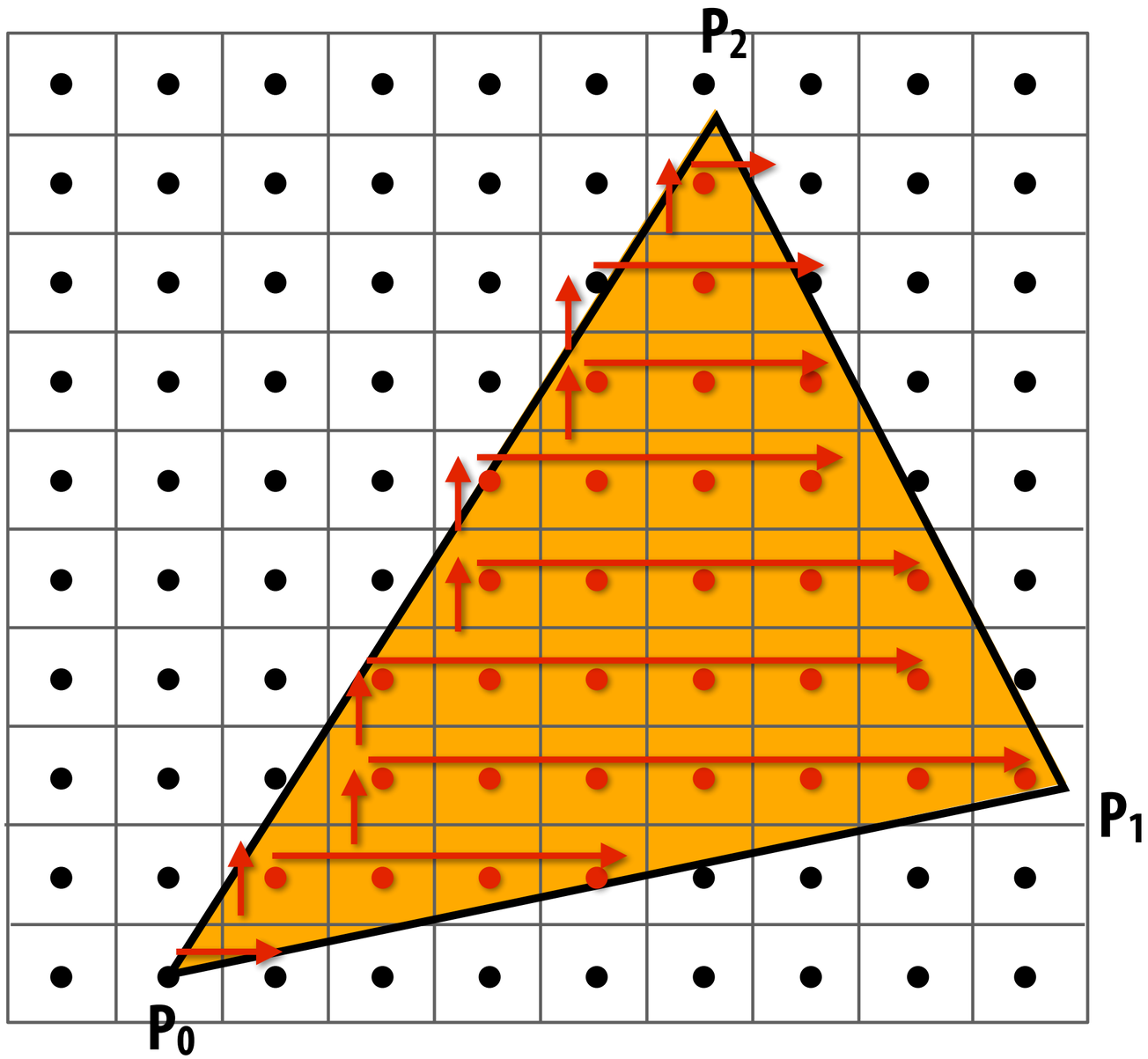
Incremental triangle traversal

$$P_i = (X_i, Y_i)$$

$$dX_i = X_{i+1} - X_i$$
$$dY_i = Y_{i+1} - Y_i$$

$$E_i(x, y) = (x - X_i)\, dY_i - (y - Y_i)\, dX_i$$
$$= A_i\, x + B_i\, y + C_i$$

$$E_i(x, y) = 0 \ : \text{point on edge}$$
$$> 0 \ : \text{outside edge}$$
$$< 0 \ : \text{inside edge}$$



Efficient incremental update:

$$dE_i(x+1,y) = E_i(x,y) + dY_i = E_i(x,y) + A_i$$
$$dE_i(x,y+1) = E_i(x,y) + dX_i = E_i(x,y) + B_i$$

Incremental update saves computation:
Only one addition per edge, per sample test

Many traversal orders are possible: backtrack, zig-zag, Hilbert/Morton curves (locality maximizing)

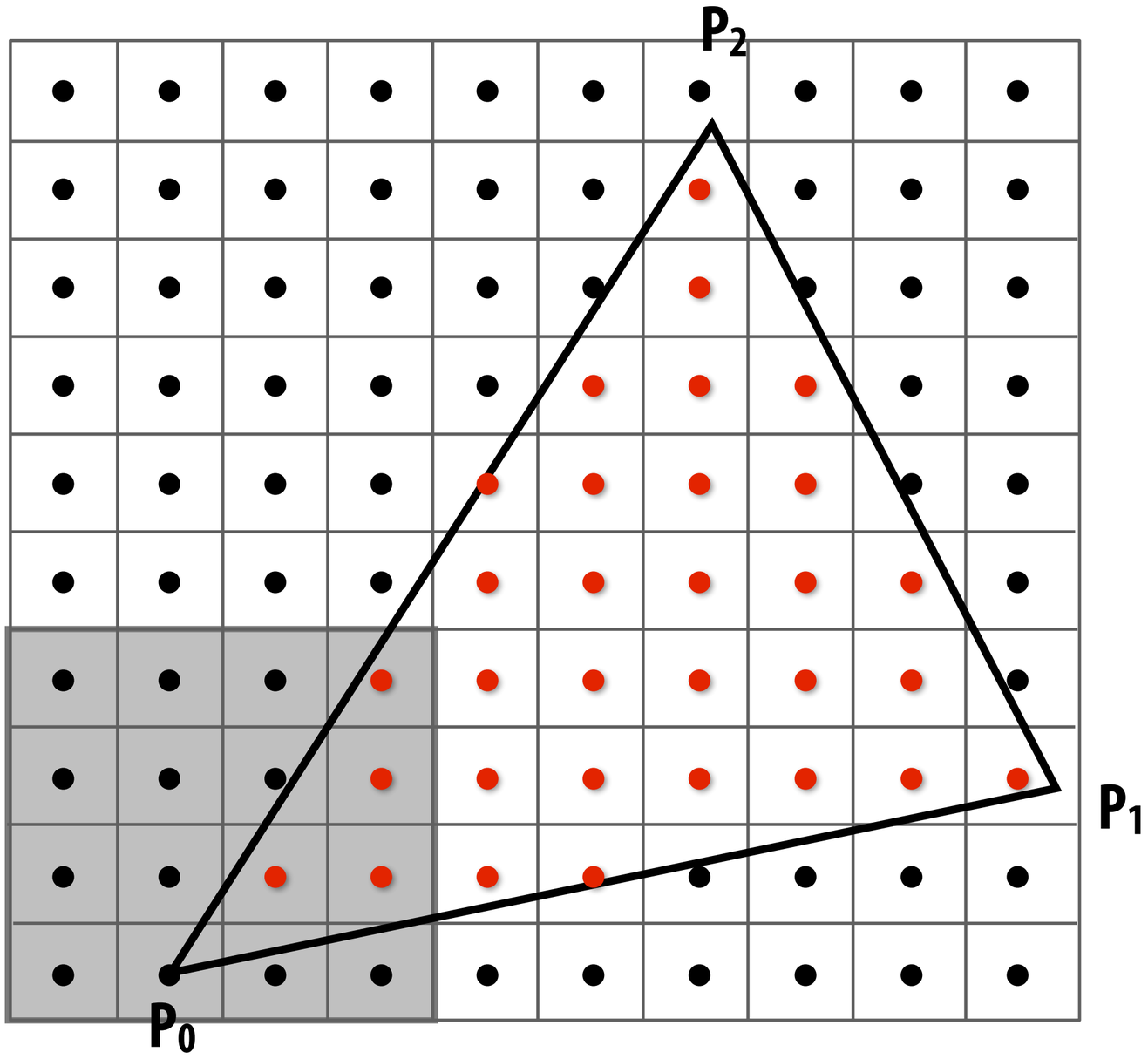
Modern approach: tiled triangle traversal

Traverse triangle in blocks

Test all samples in block against triangle in parallel

Advantages:

- **Simplicity of wide parallel execution overcomes cost of extra point-in-triangle tests (most triangles cover many samples, especially when super-sampling coverage)**
- **Can skip sample testing work: entire block not in triangle ("early out"), entire block entirely within triangle ("early in")**
- **Additional advantaged related to accelerating occlusion computations (not discussed today)**



All modern GPUs have special-purpose hardware for efficiently performing point-in-triangle tests

Summary

- **We formulated computing triangle-screen coverage as a sampling problem**
  - Triangle-screen coverage is a 2D signal
  - Undersampling and the use of simple (non-ideal) reconstruction filters may yield aliasing
  - In today's example, we reduced aliasing via supersampling

- **Image formation on a display**
  - When samples are 1-to-1 with display pixels, sample values are handed directly to display
  - When "supersampling", resample densely sampled signal down to display resolution

- **Sampling screen coverage of a projected triangle:**
  - Performed via three point-inside-edge tests
  - Real-world implementation challenge: balance conflicting goals of avoiding unnecessary point-in-triangle tests and maintaining parallelism in algorithm implementation

Next time: 3D Transformations