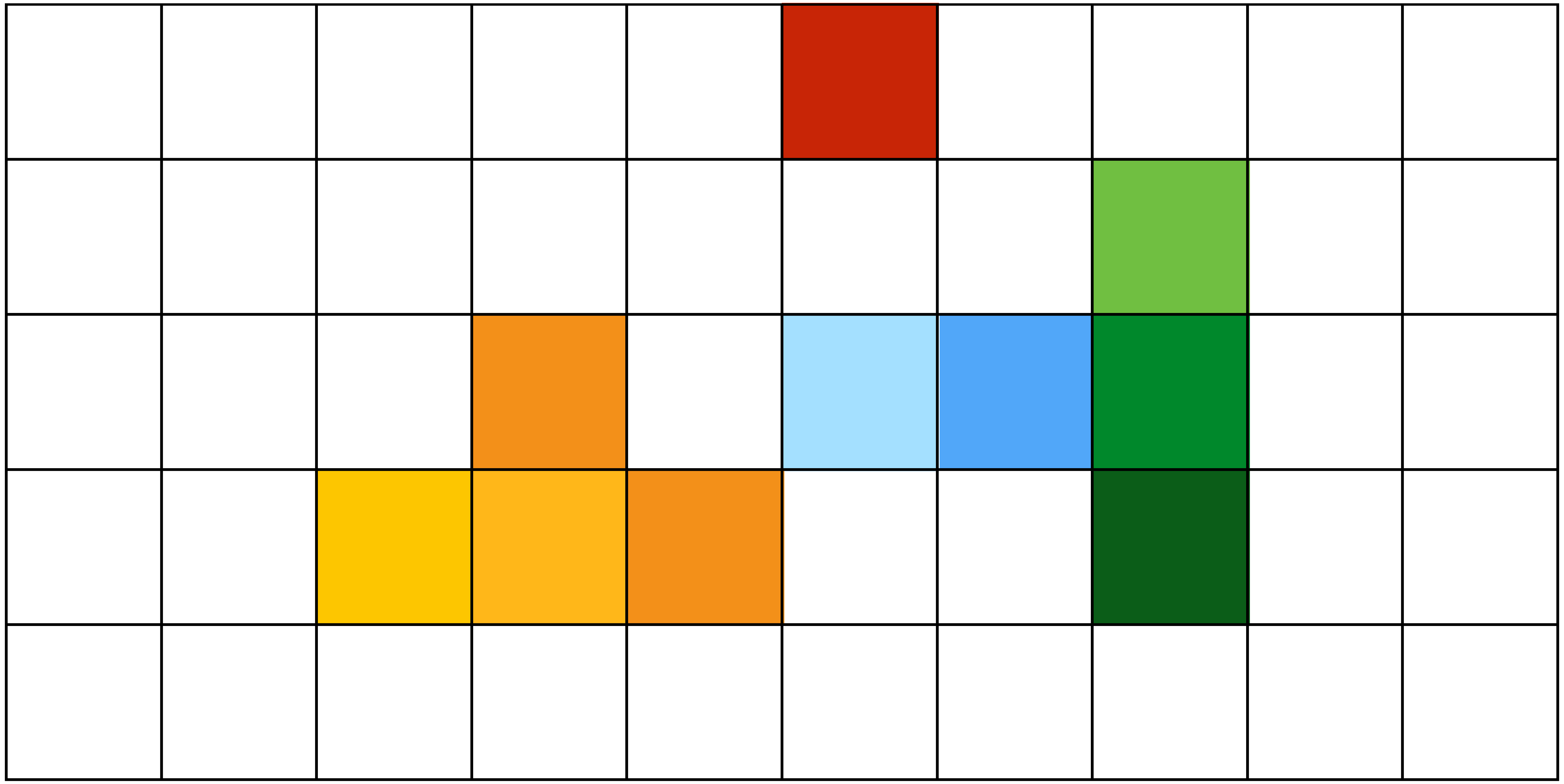
Lecture 2:

Sampling and Anti-aliasing

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

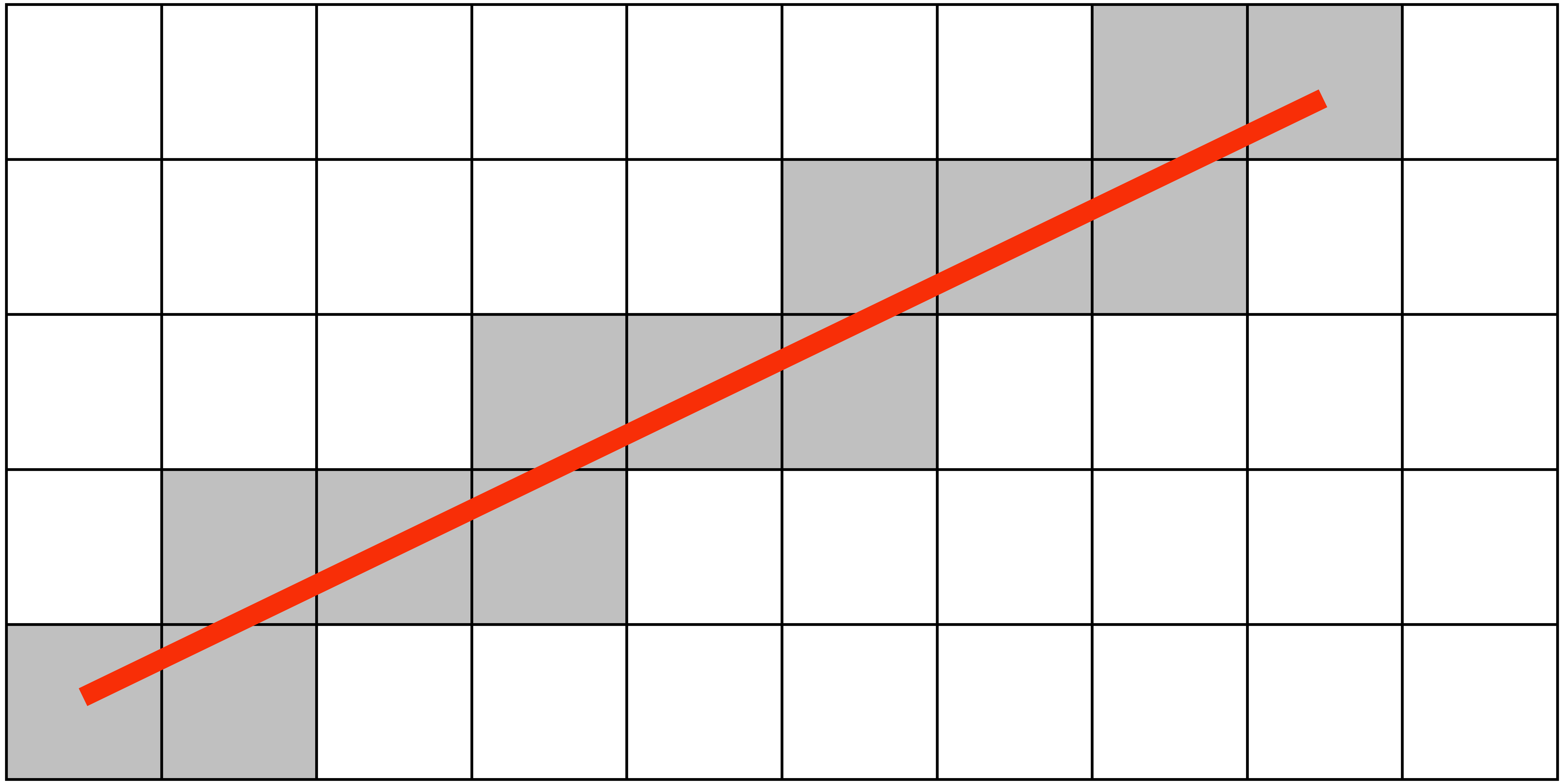
Last time

- A very simple notion of digital image representation (that we are about to challenge!)
- An image = a 2D array of color values



Last time: displaying an image

Last time: what pixels should we color in to draw a line?

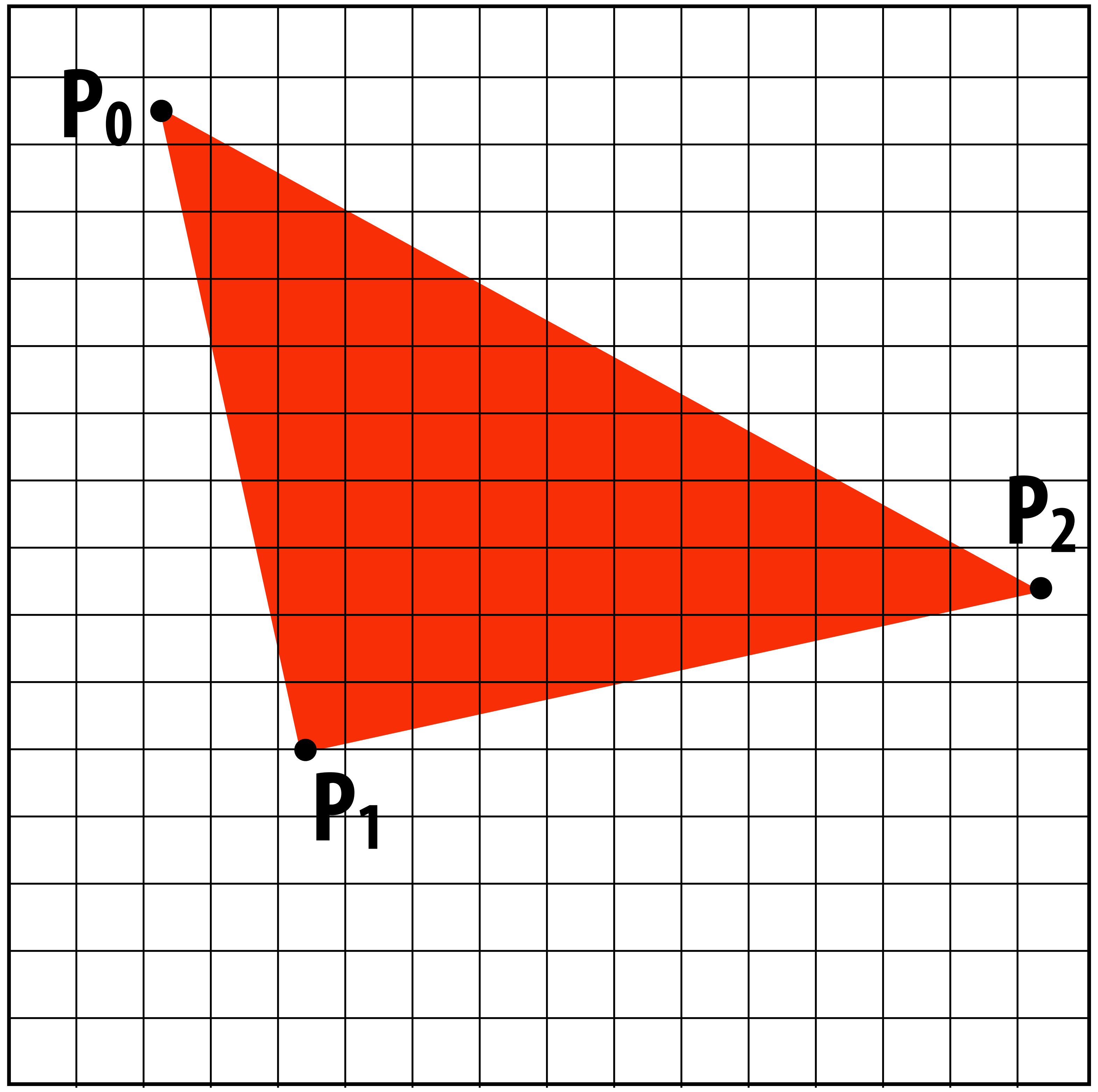


One possible heuristic: light up all pixels intersected by the line?

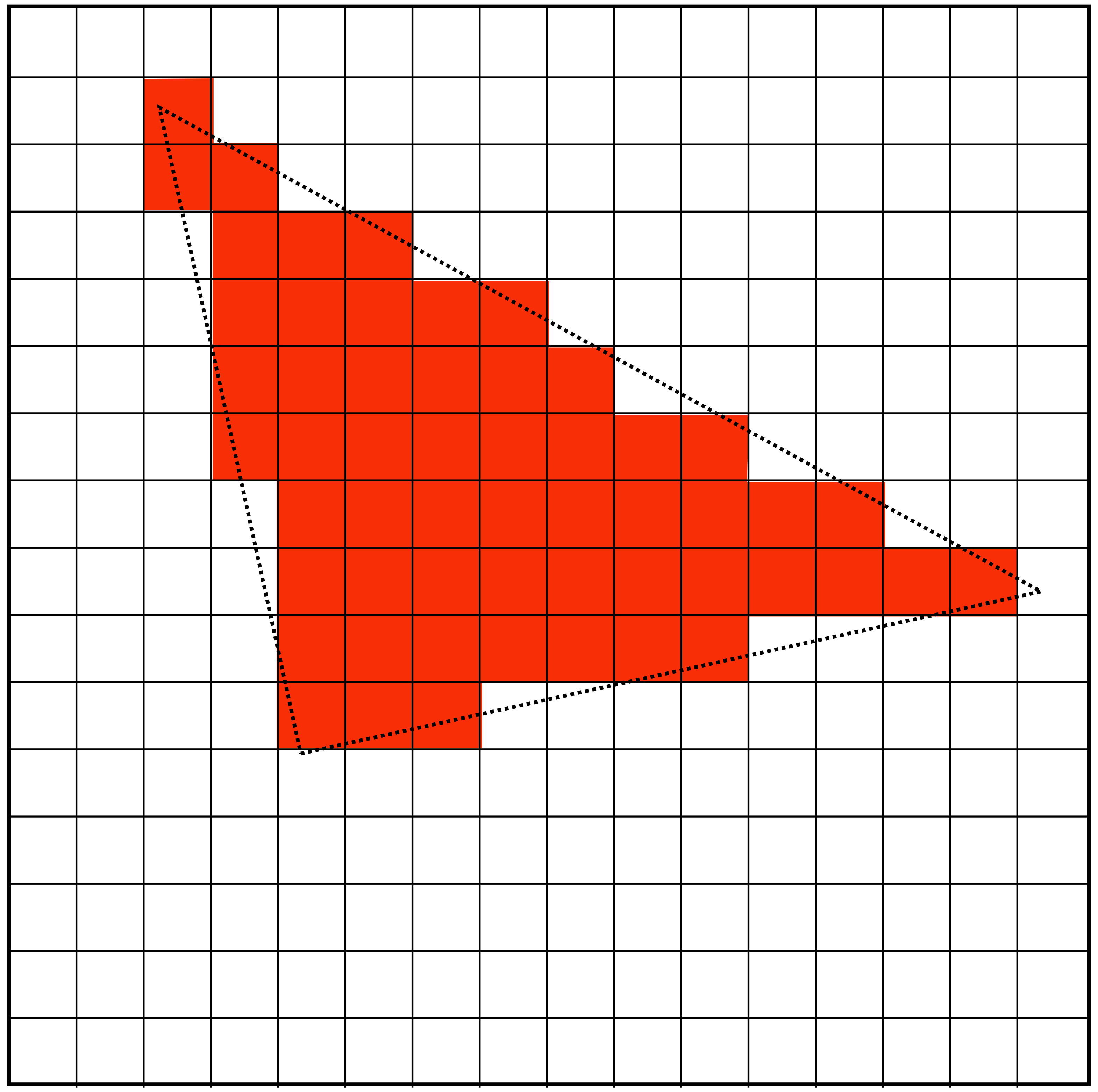
Last time: drawing a triangle in 2D

(Converting a representation of a triangle into an image)

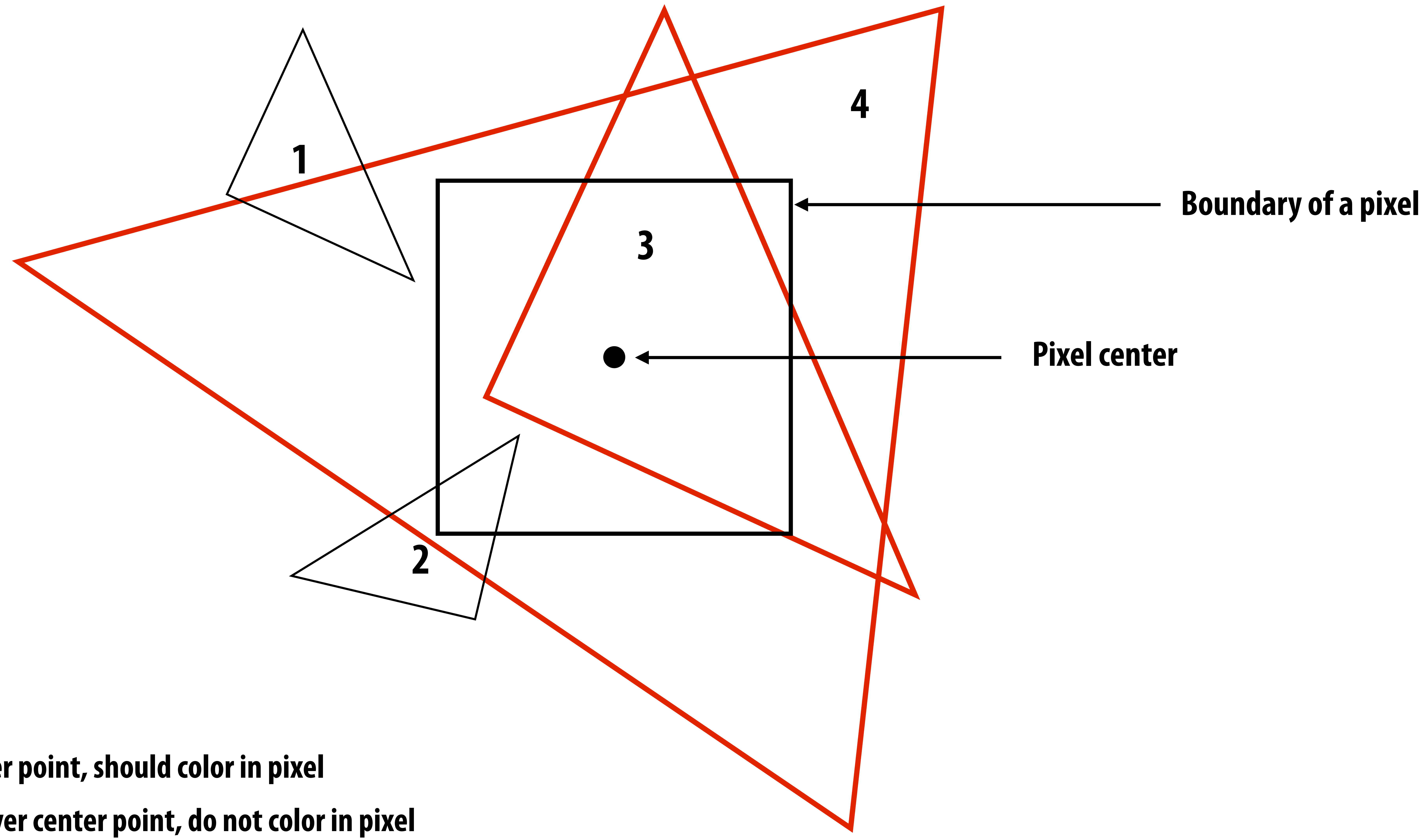
Input:
2D position of triangle vertices: P_0, P_1, P_2

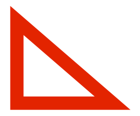


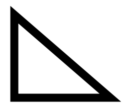
Output:
set of pixels "covered" by the triangle



Last time: when drawing triangles we filled pixels if the pixel center was inside the triangle

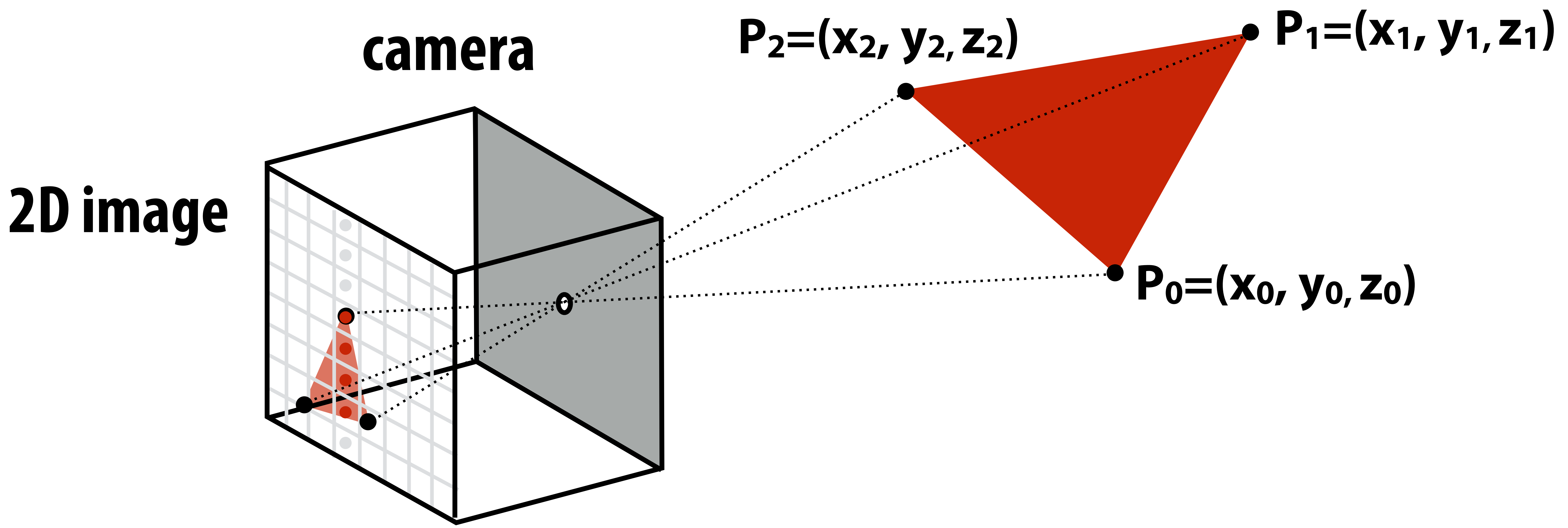


 = triangle covers center point, should color in pixel

 = triangle does not cover center point, do not color in pixel

Last time: drawing a 3D triangle: rasterization perspective

Think: "What pixels does the projected triangle cover?"



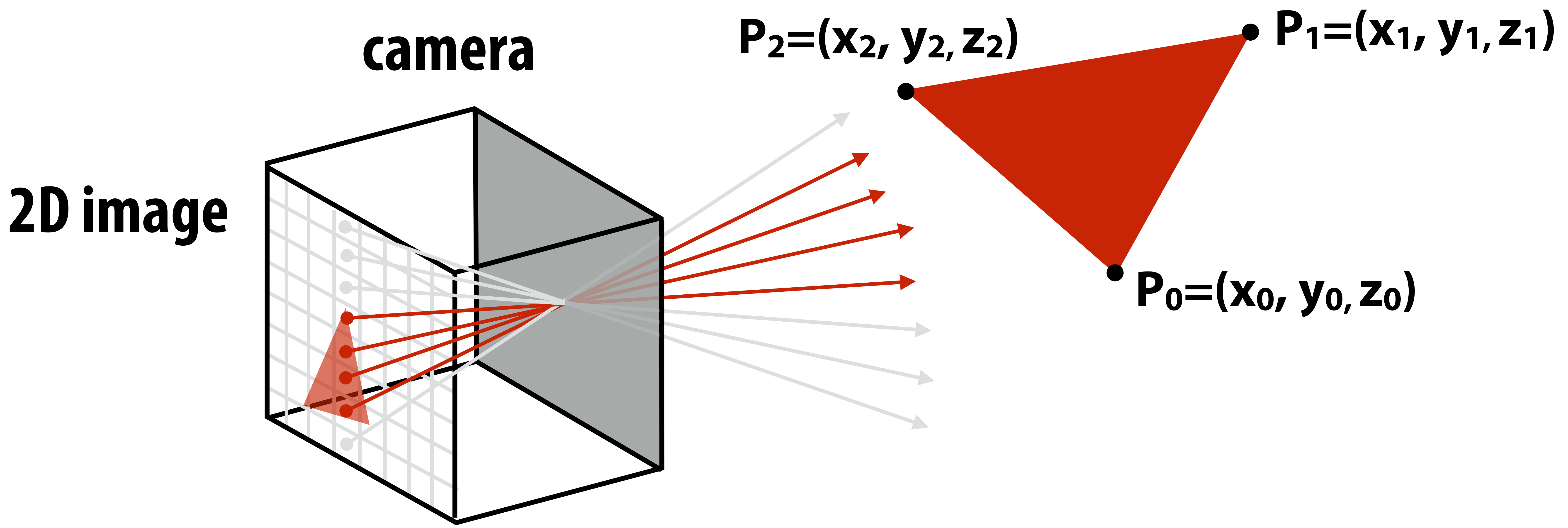
Simple pseudocode:

```
tri_projected = project_triangle_verts(tri)
  for each image pixel p:
     if (center of p is inside tri_projected)
       color pixel p the color of tri
```

Last time: drawing a 3D triangle: ray casting perspective

Think: "Is the triangle visible along the ray from a pixel through the pinhole?"

Aka. Does a ray originating at the pixel center and leaving the camera "hit" the triangle?

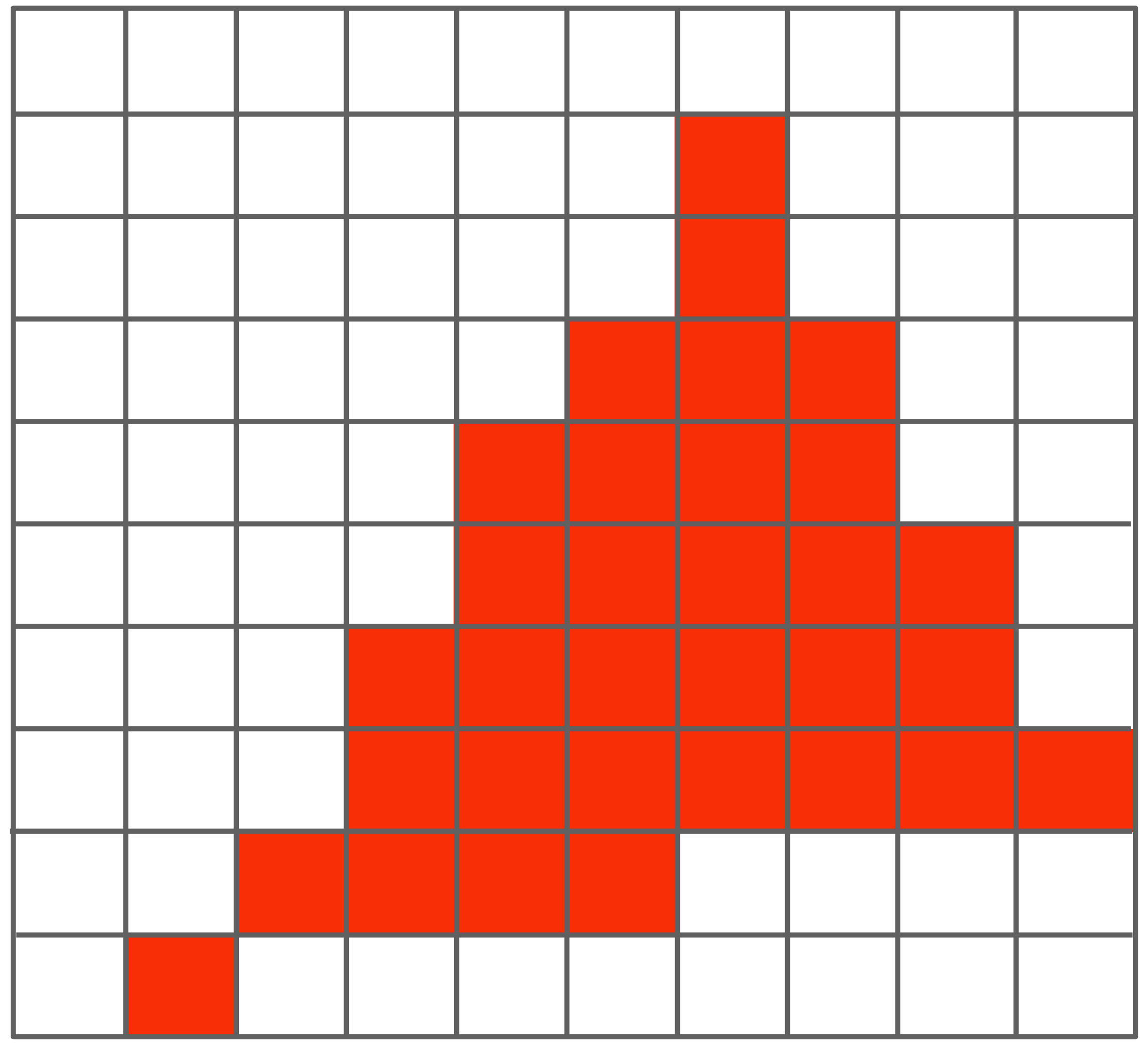


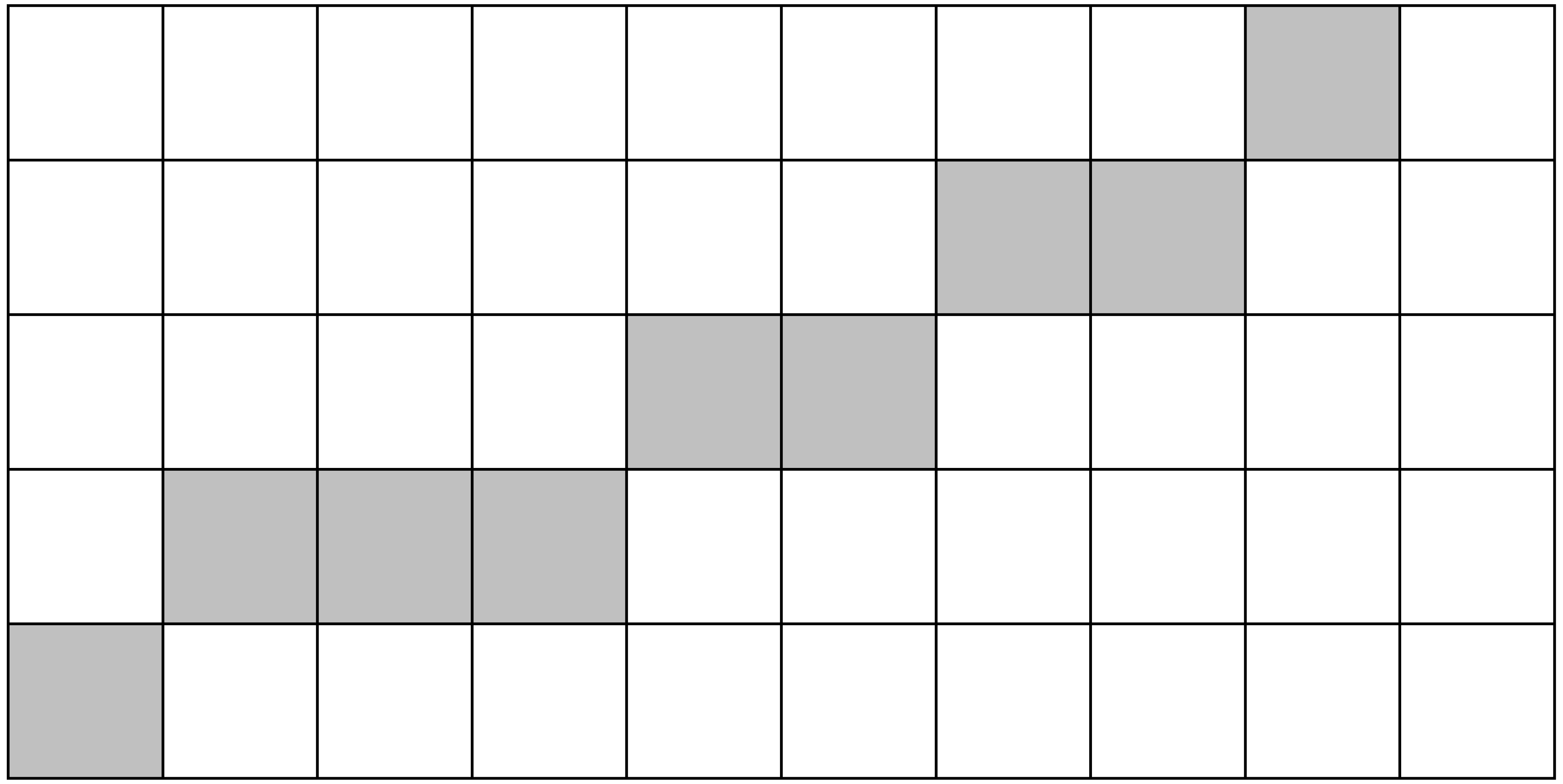
Simple pseudocode:

```
for each image pixel p:
   let r = ray from center of p leaving camera through pinhole
   if (r hits tri)
     color pixel p the color of tri
```

Not everyone was happy with our renderings

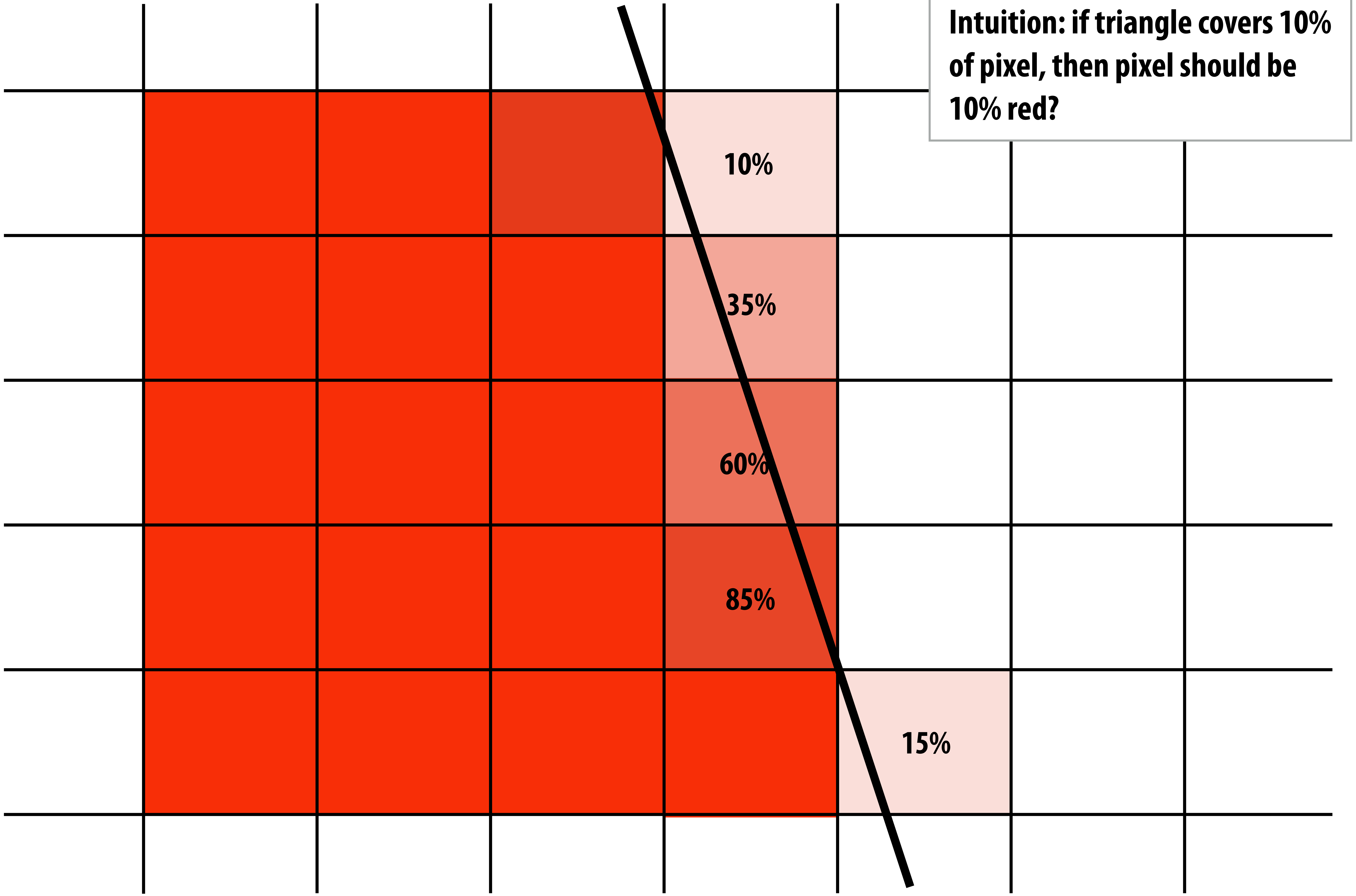
- **Students mentioned "jaggy edges"**
- **Commented on how they desired something "more smooth", etc.**



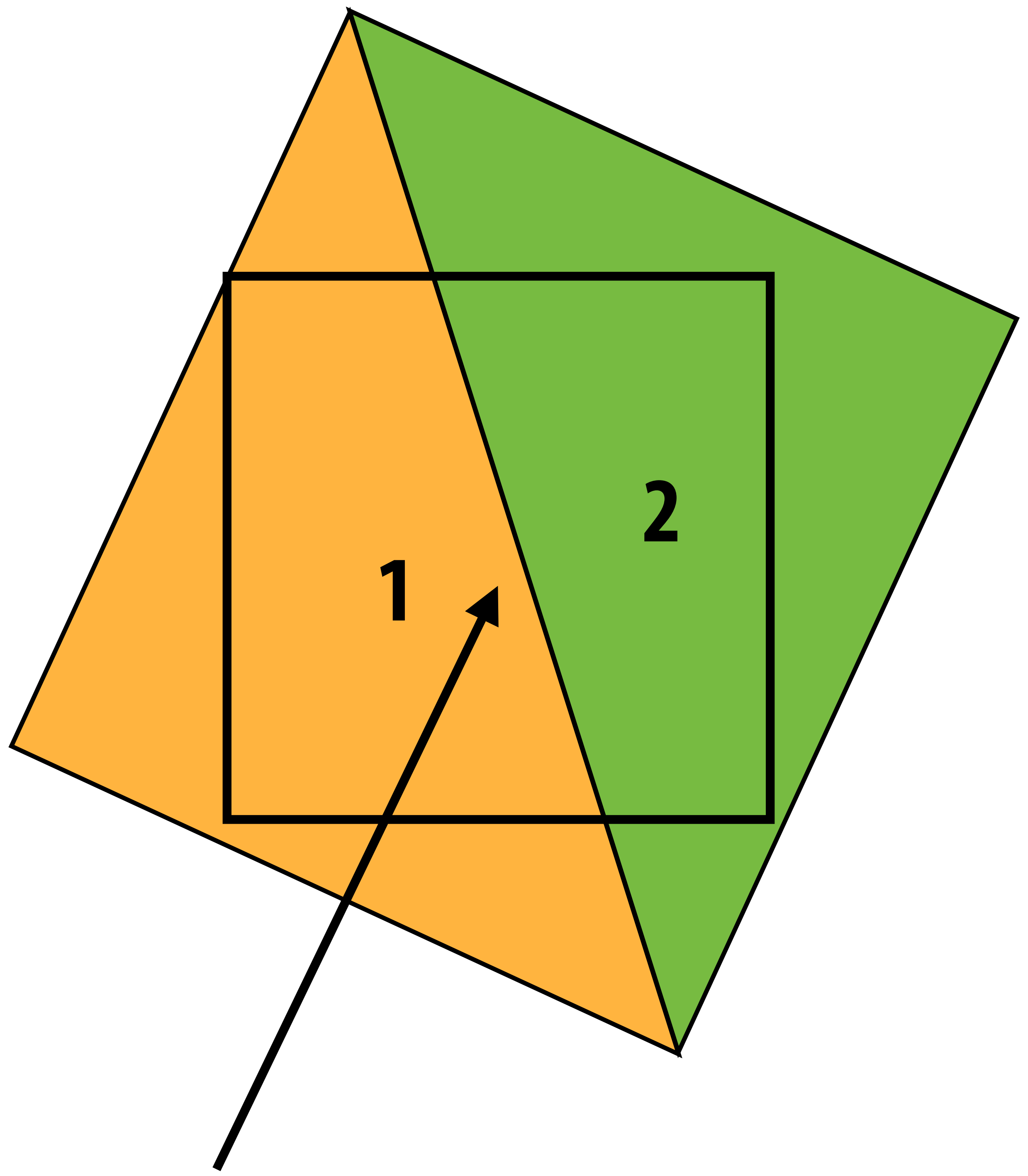


One option floated by the class: compute fraction of pixel area covered by triangle, then color pixel according to this fraction.

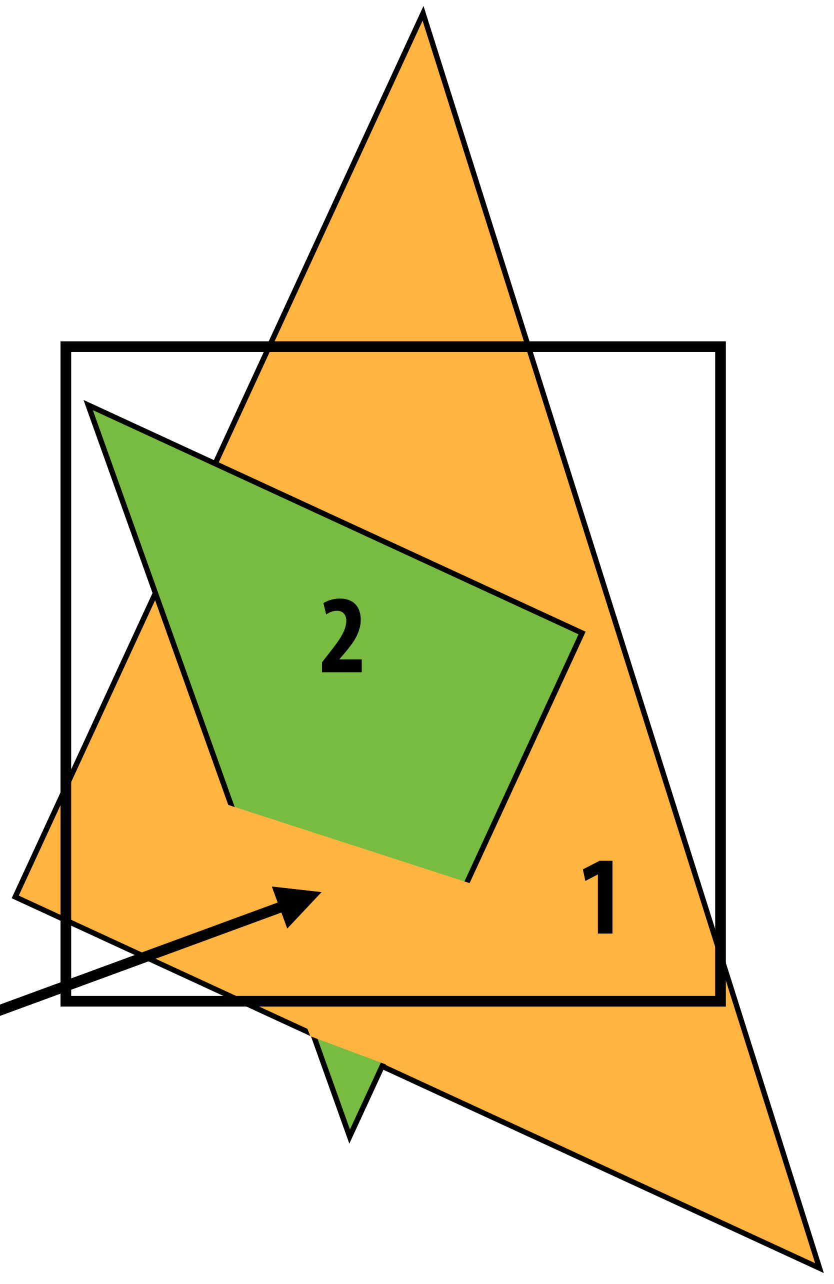
Intuition: if triangle covers 10% of pixel, then pixel should be 10% red?



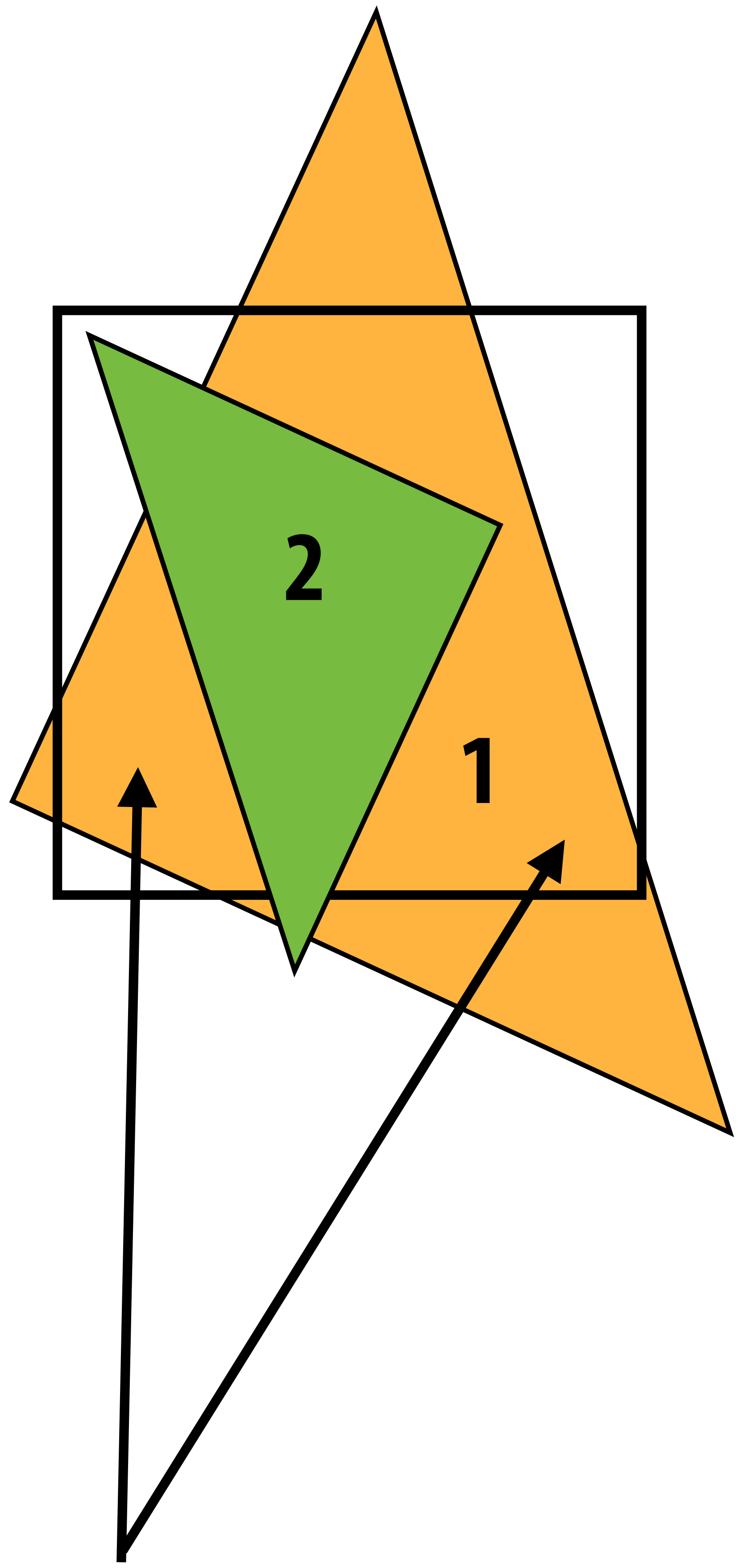
Analytical coverage schemes get tricky when considering scenes with



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



Interpenetration of triangles: even trickier



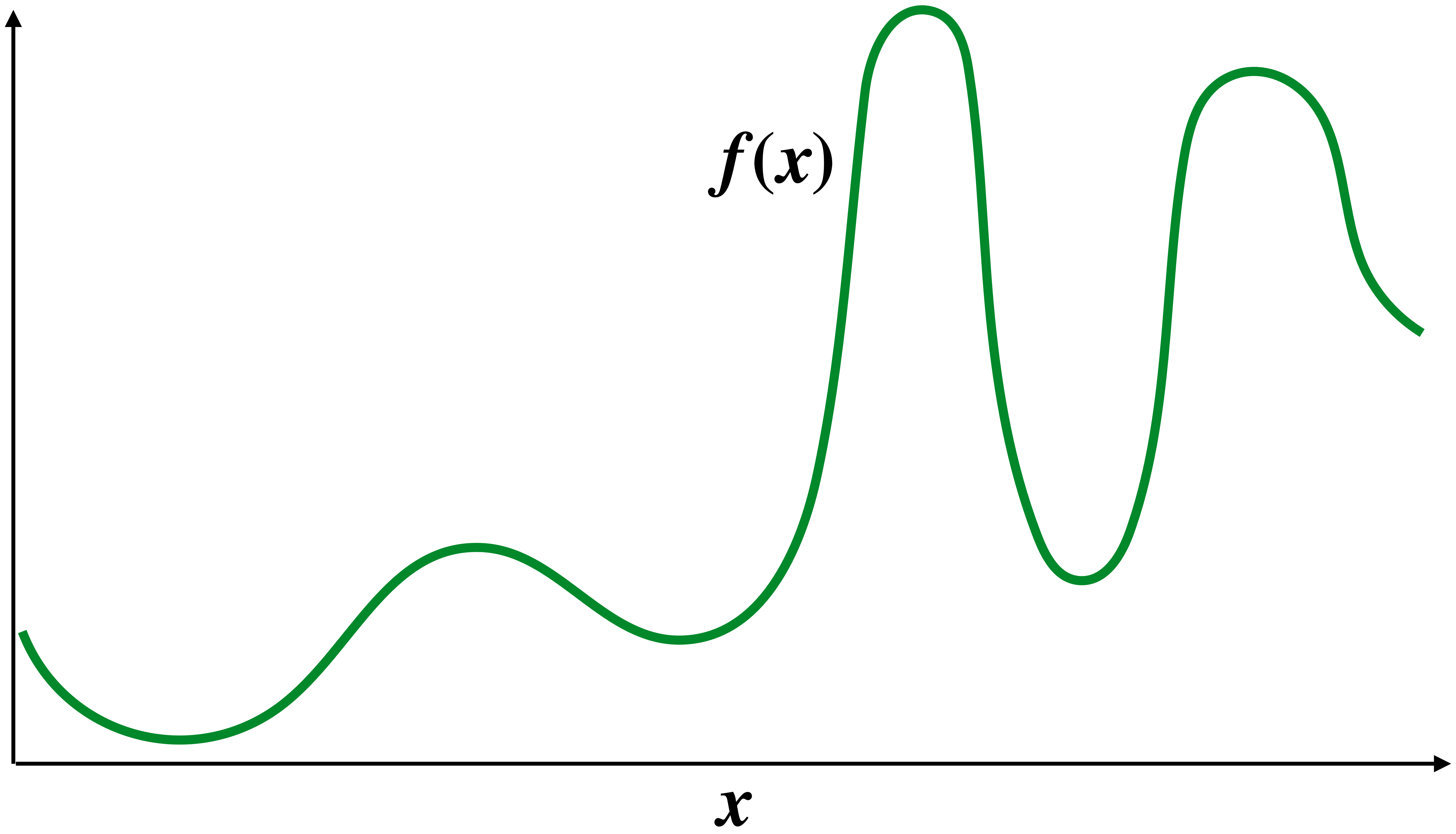
Two regions of triangle 1 contribute to pixel. One of these regions is not even convex.

In Lecture 1 we drew triangles using a simple method: point sampling

Which we implemented by testing whether specific points were inside the triangle (or if rays in specific directions hit a triangle)

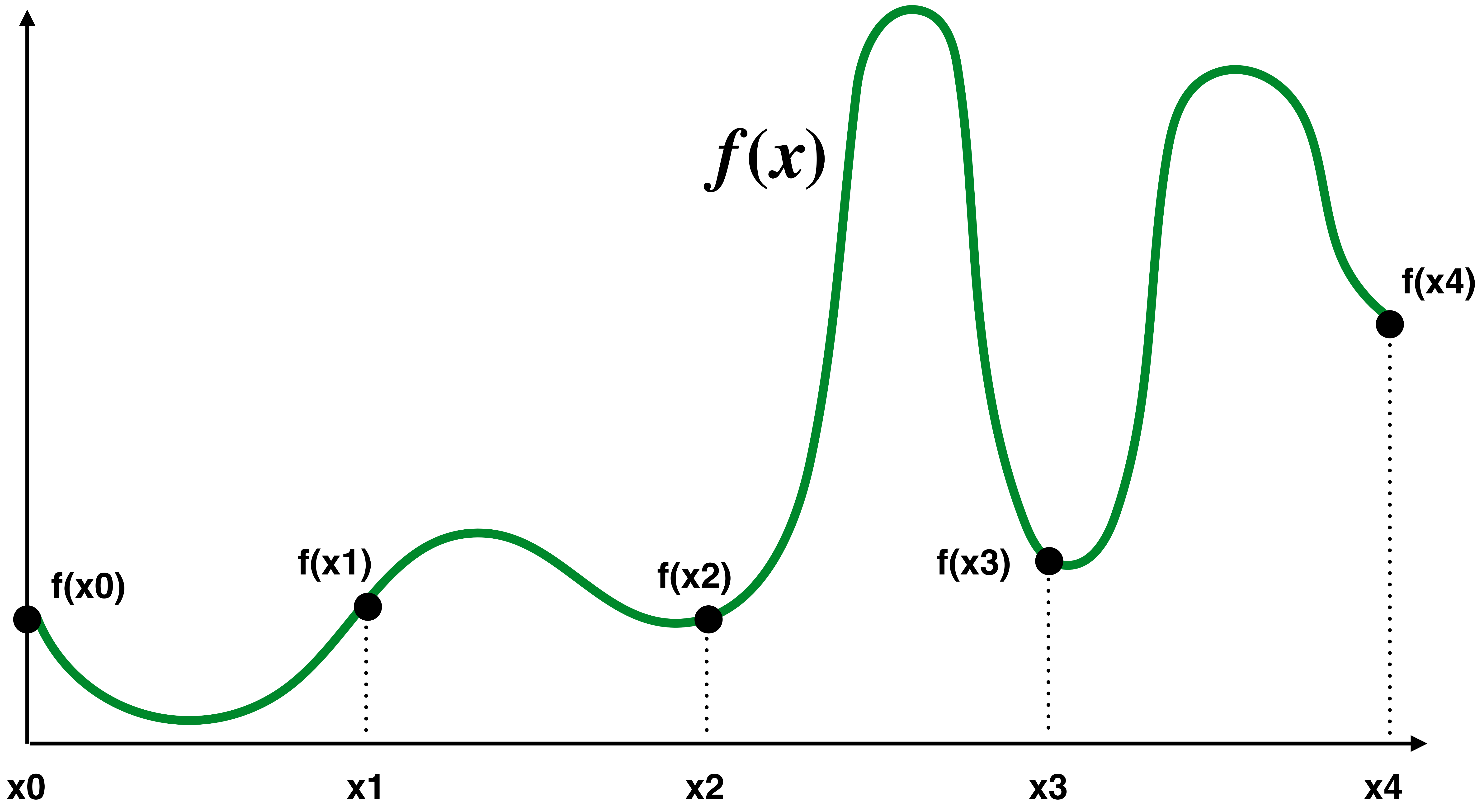
Before talking about sampling in 2D or 3D, let's consider sampling in 1D first…

Consider a 1D signal: f(x)



Sampling: taking measurements of a signal

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



A discrete representation of f(x) is given by the samples $f(x_0)$, $f(x_1)$, $f(x_2)$, $f(x_3)$, $f(x_4)$

Audio file: stores samples of a 1D signal

Audio is often sampled at 44.1 KHz

Sampling a function

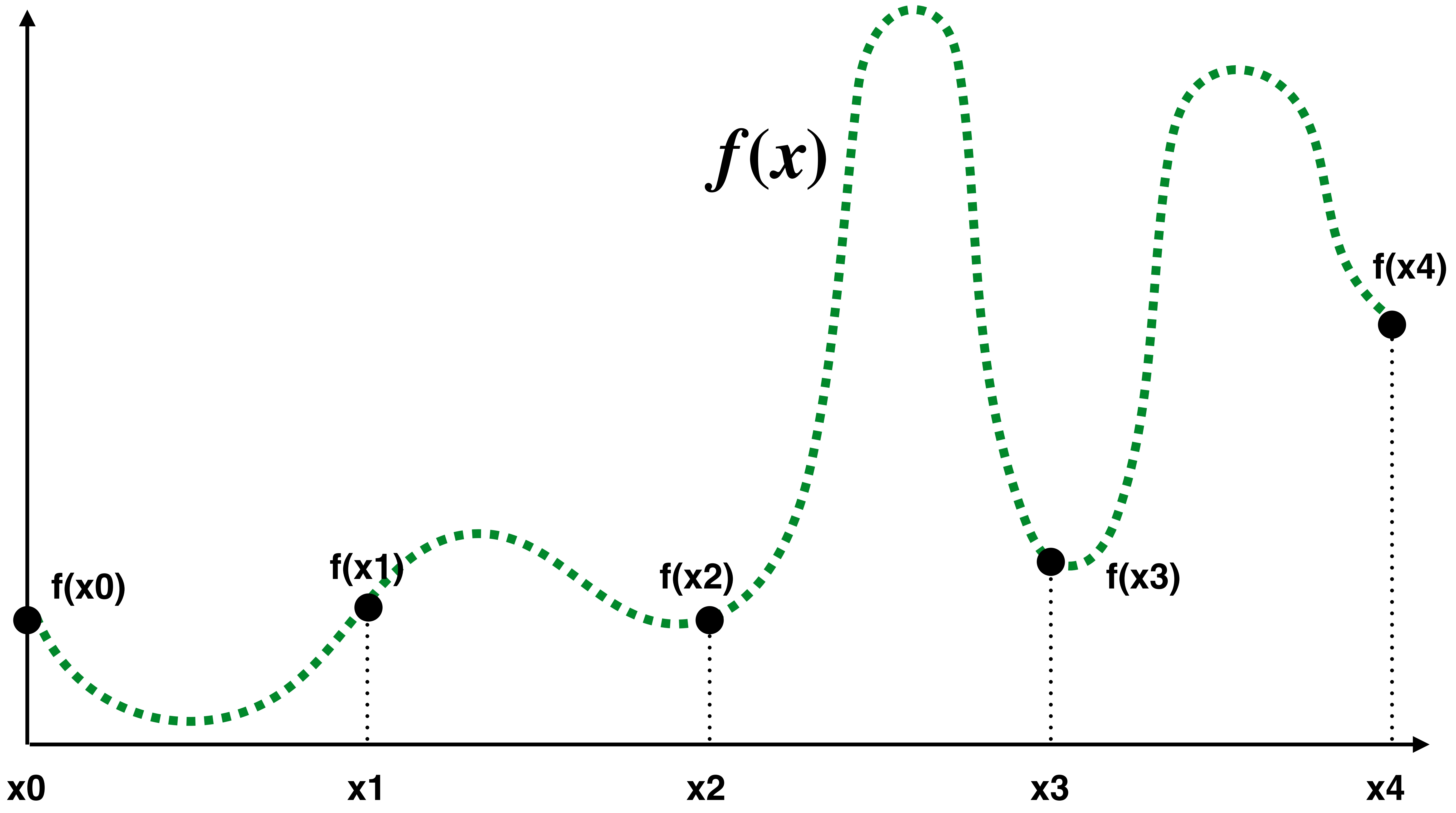
- Evaluating a function at a point is sampling the function's value

- We can discretize a function by periodic sampling

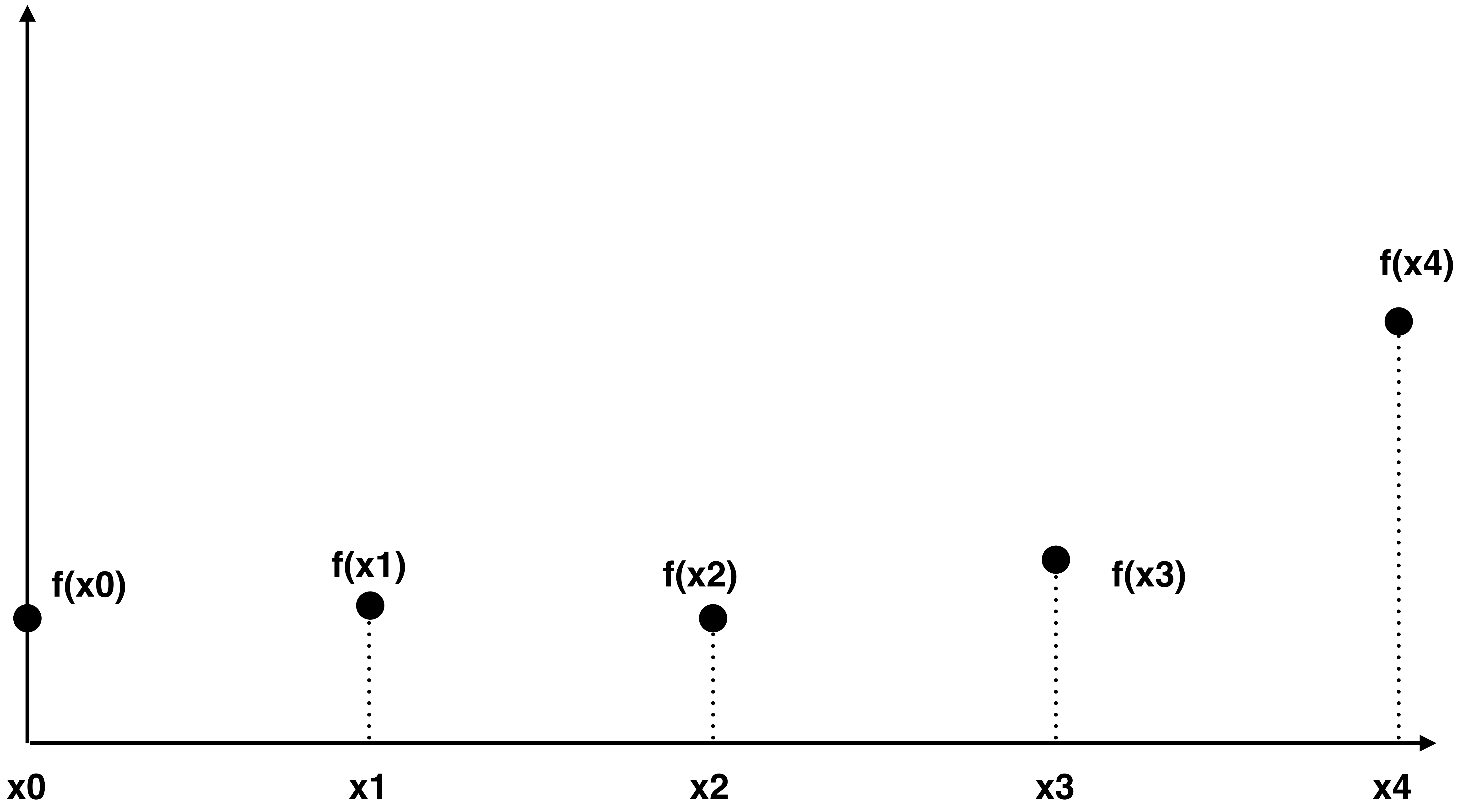
```
for(int x = 0; x < xmax; x++)
    output[x] = f(x);
```

- Sampling is a core idea in graphics. In this class we'll sample signals parameterized by: time (1D), area (2D), angle (2D), volume (3D), paths through a scene (infinite-D) etc …

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



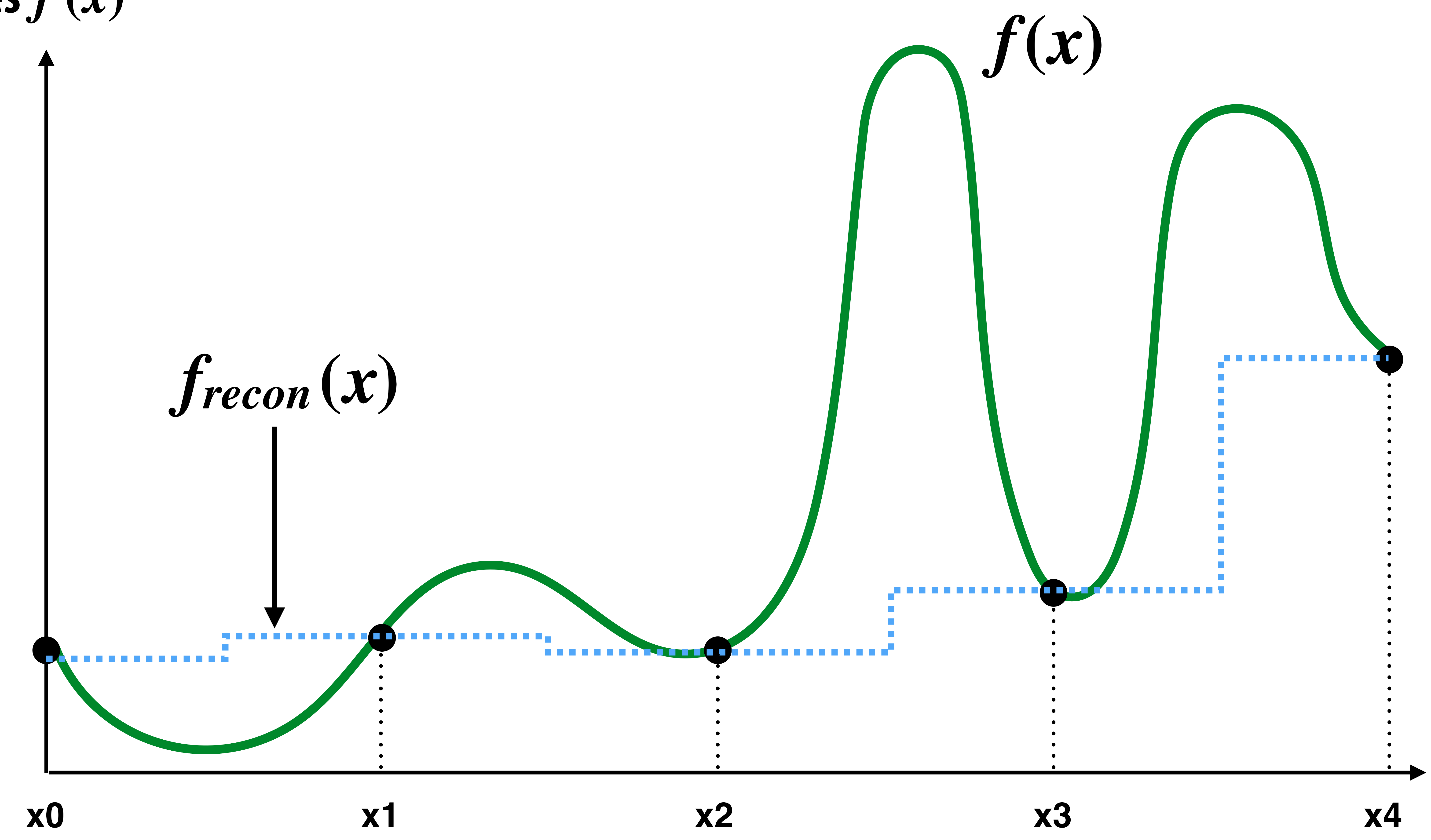
Reconstruction: given a set of samples, how might we attempt to reconstruct the original (continuous) signal $f(x)$?



Piecewise constant approximation

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

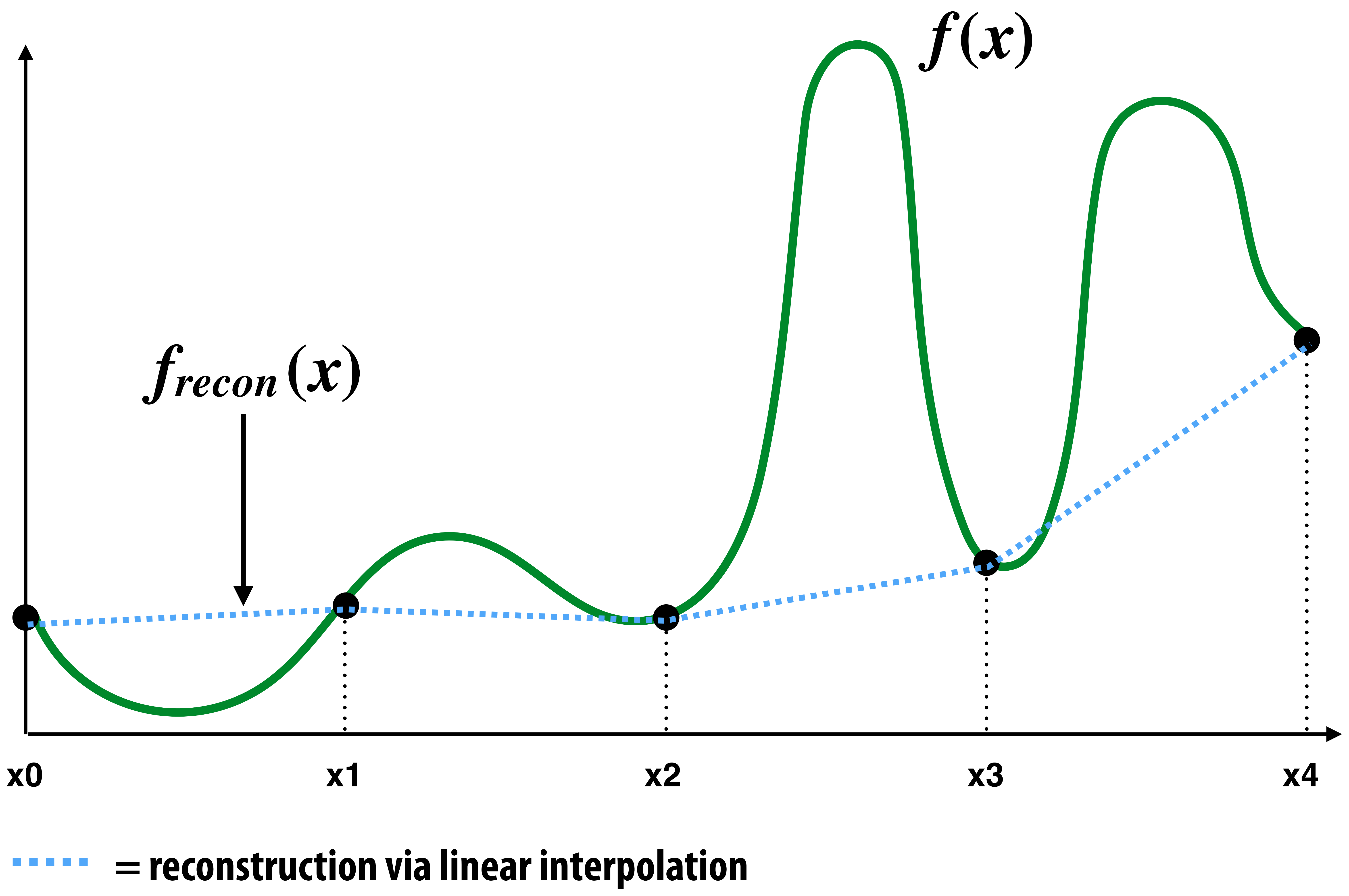
$f_{recon}(x)$ approximates $f(x)$



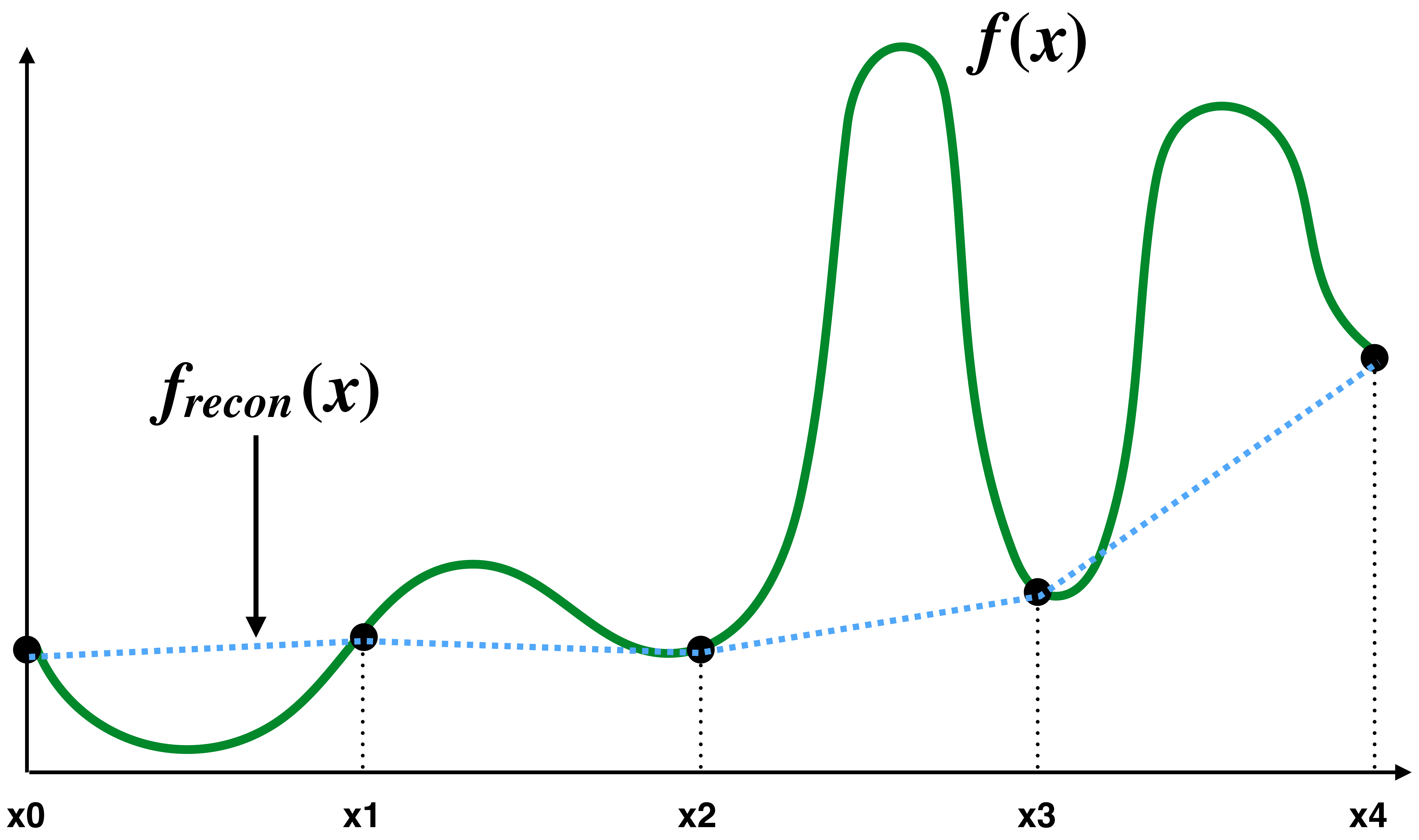
= reconstruction via piece-wise constant interpolation (nearest neighbor)

Piecewise linear approximation

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



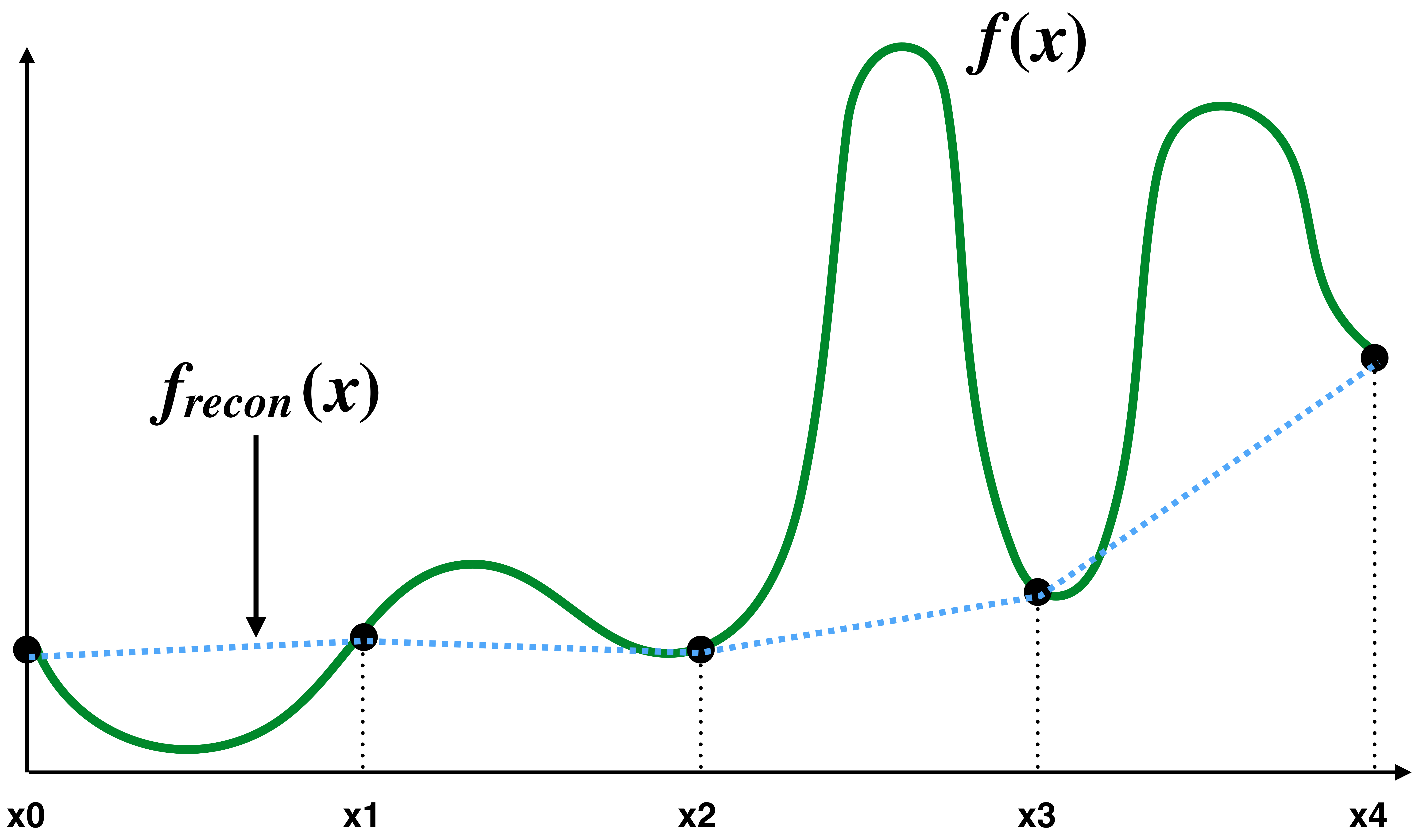
How can we represent the signal more accurately?



Answer: sample signal more densely (increase sampling rate)

Reconstruction from sparse sampling

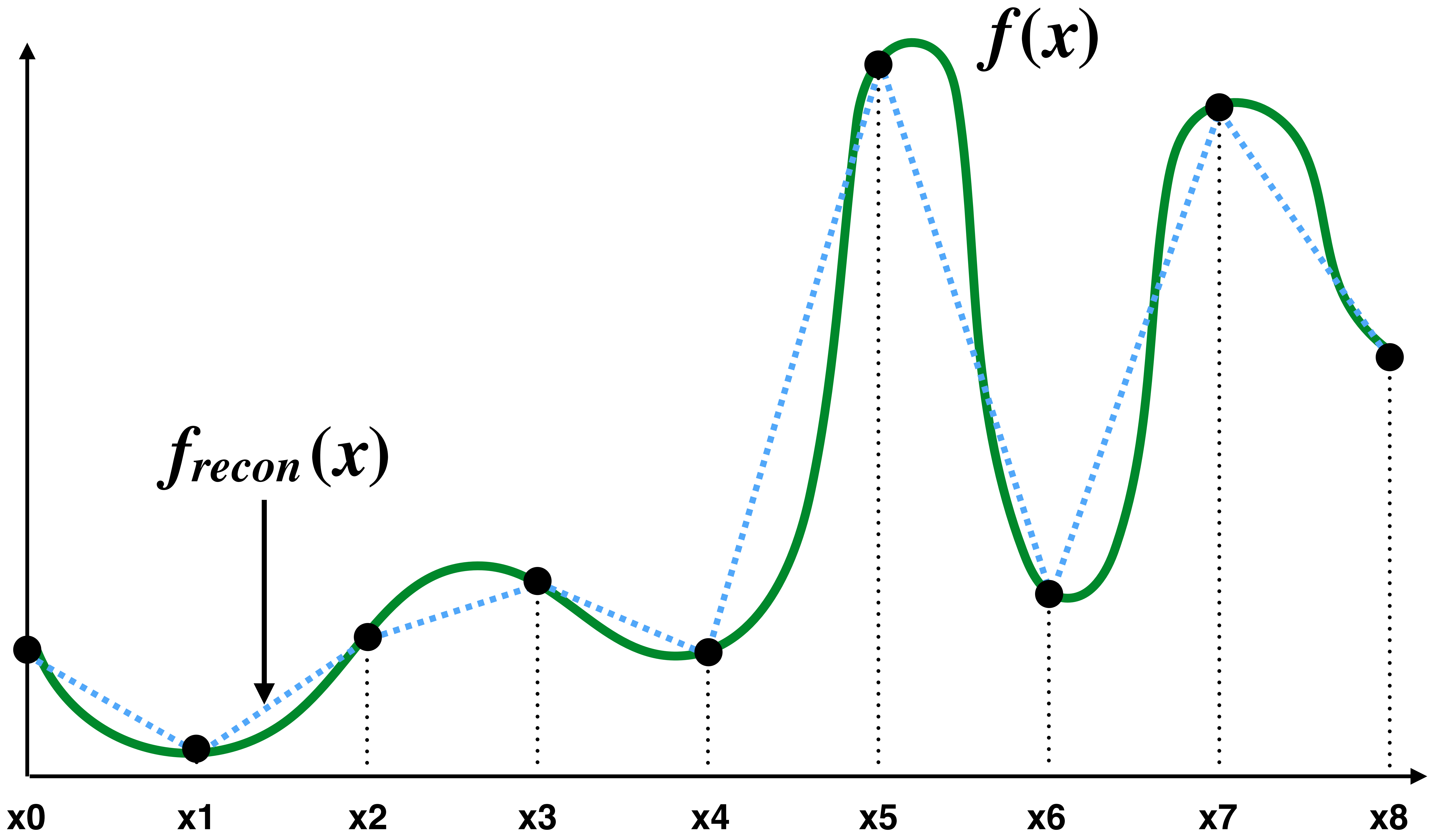
(5 samples)



= reconstruction via linear interpolation

More accurate reconstructions result from denser sampling

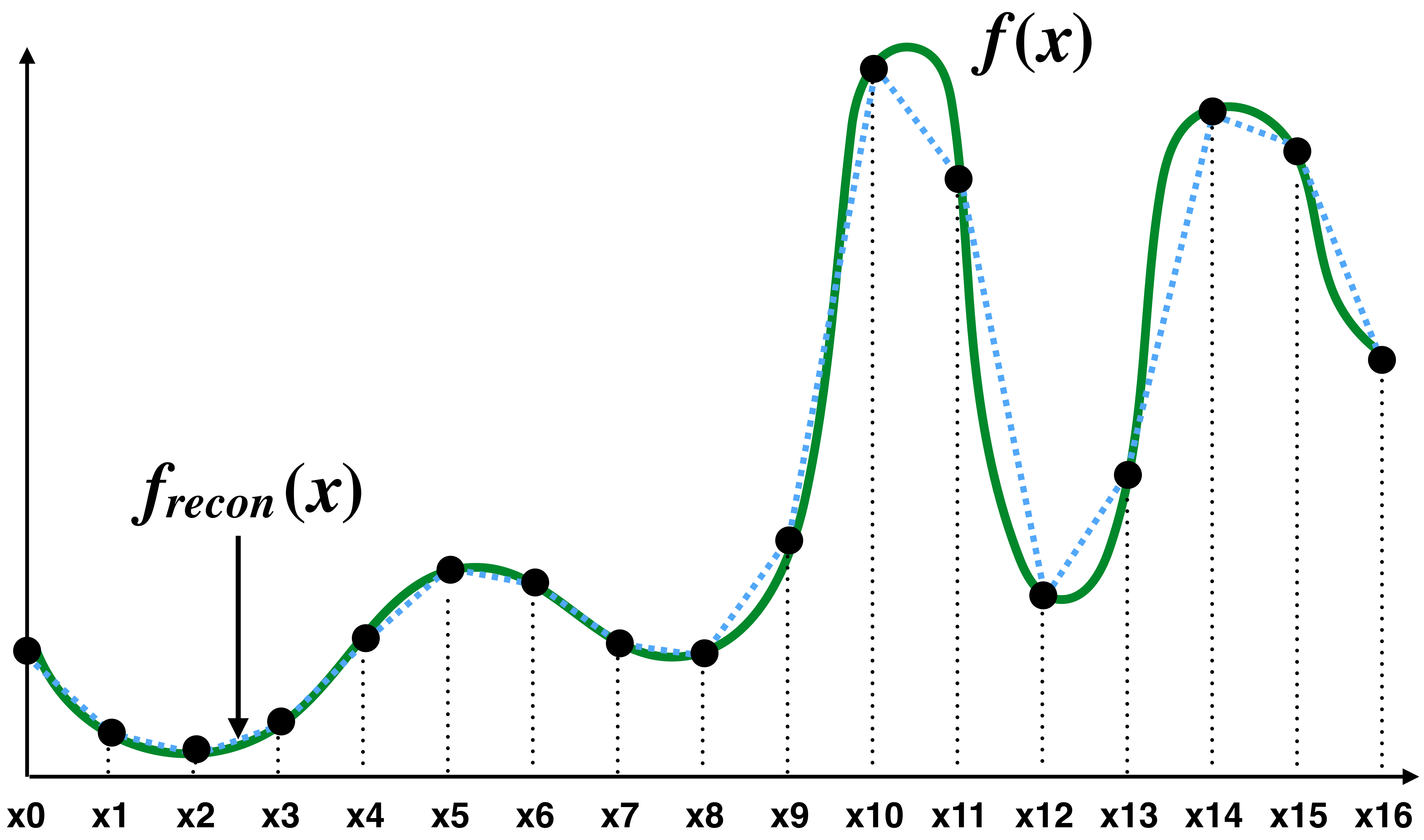
(9 samples)



= reconstruction via linear interpolation

More accurate reconstructions result from denser sampling

(17 samples)



= reconstruction via linear interpolation

Drawing a triangle by sampling 2D points

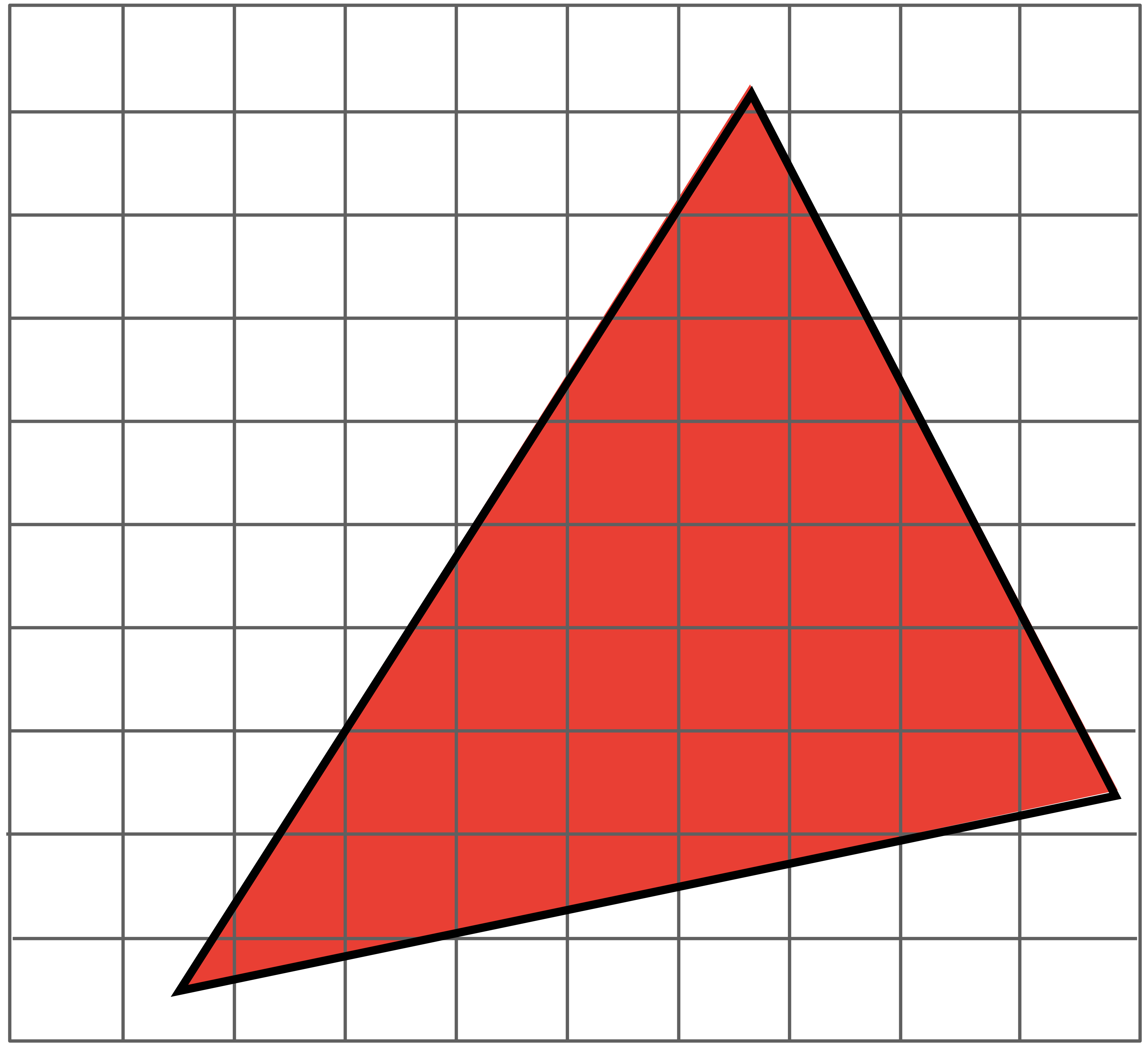
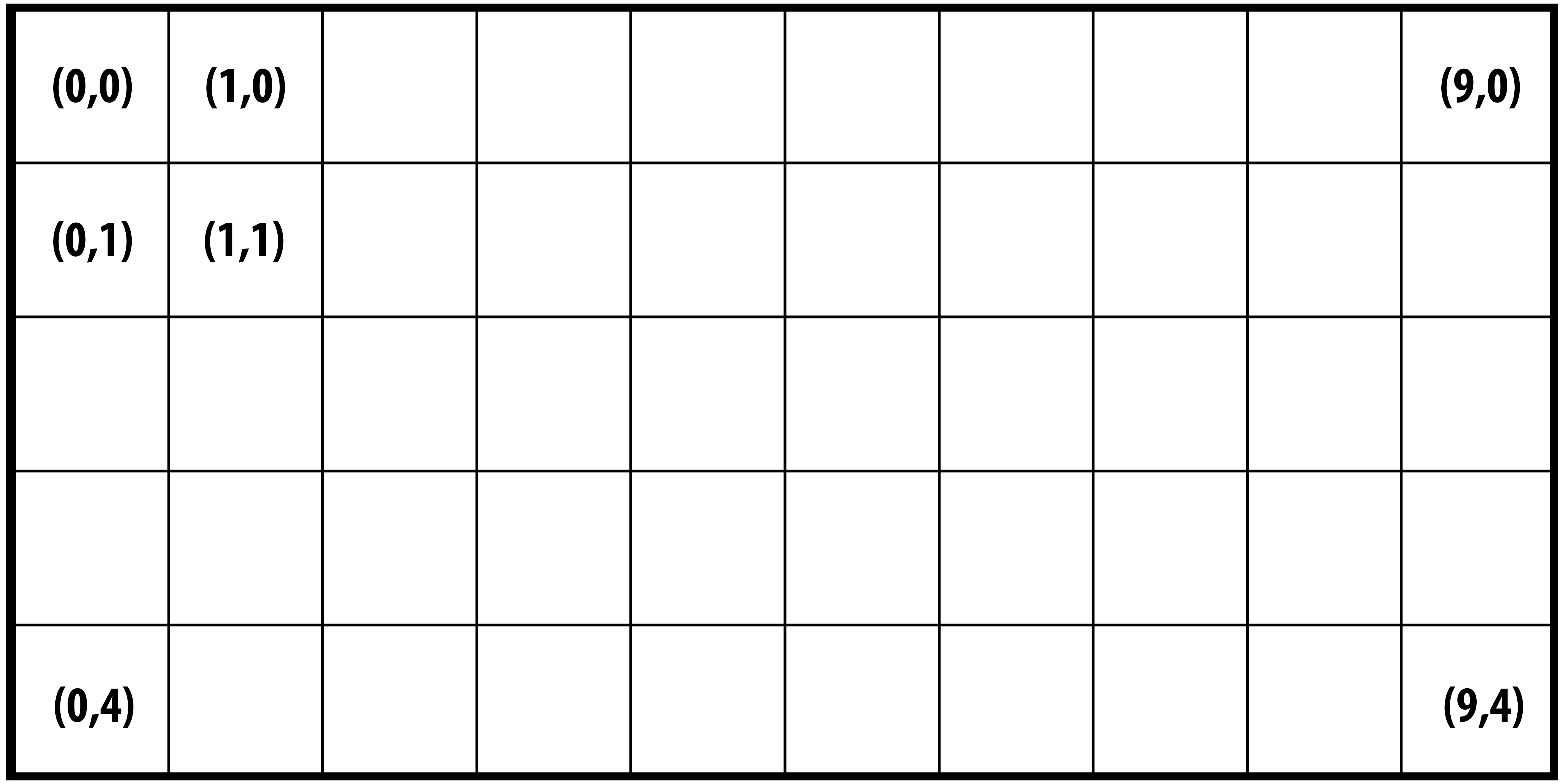


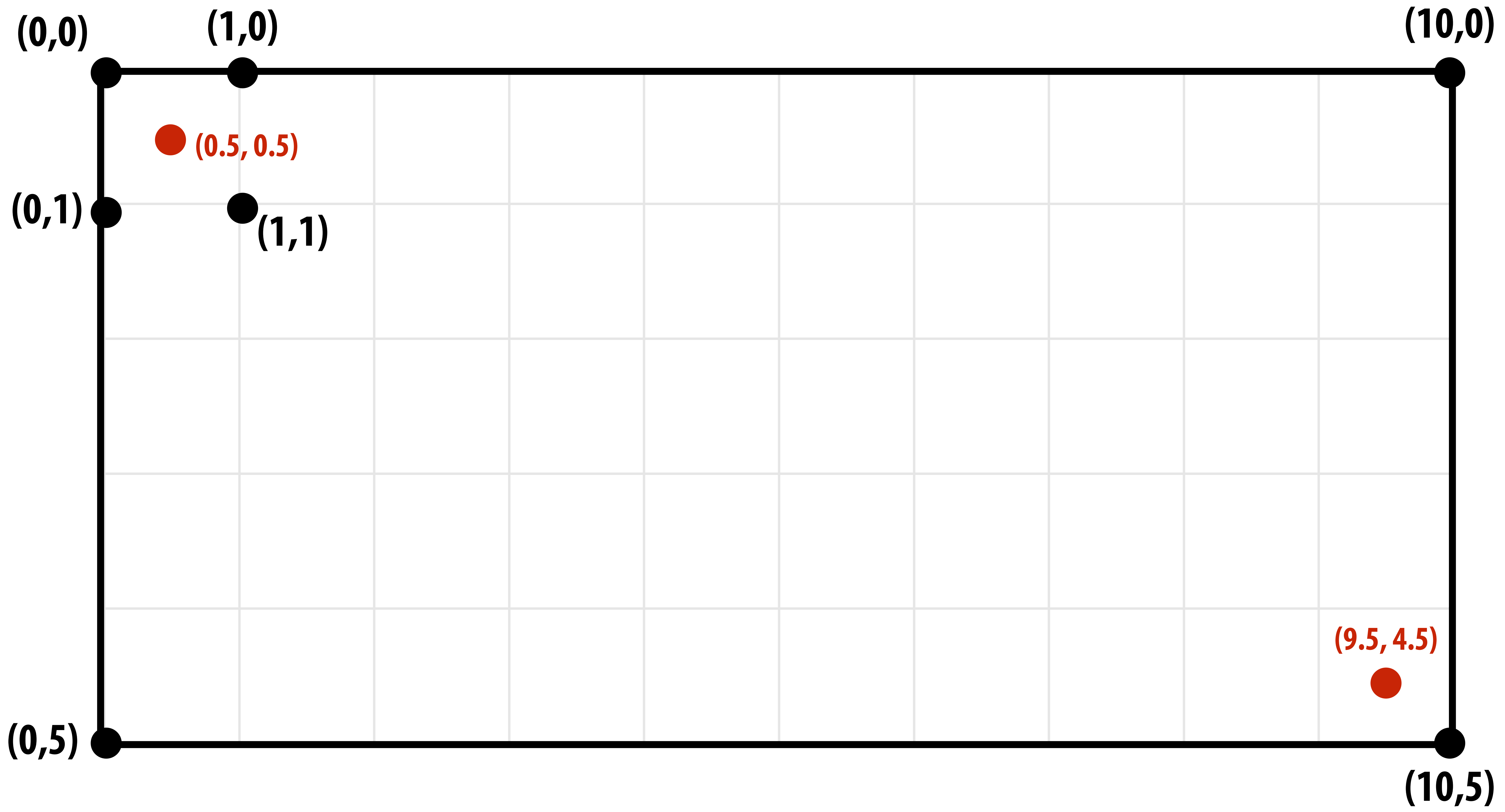
Image as a 2D matrix of pixels

Here I'm showing a 10 x 5 pixel image

Identify pixel by its integer (x,y) coordinates



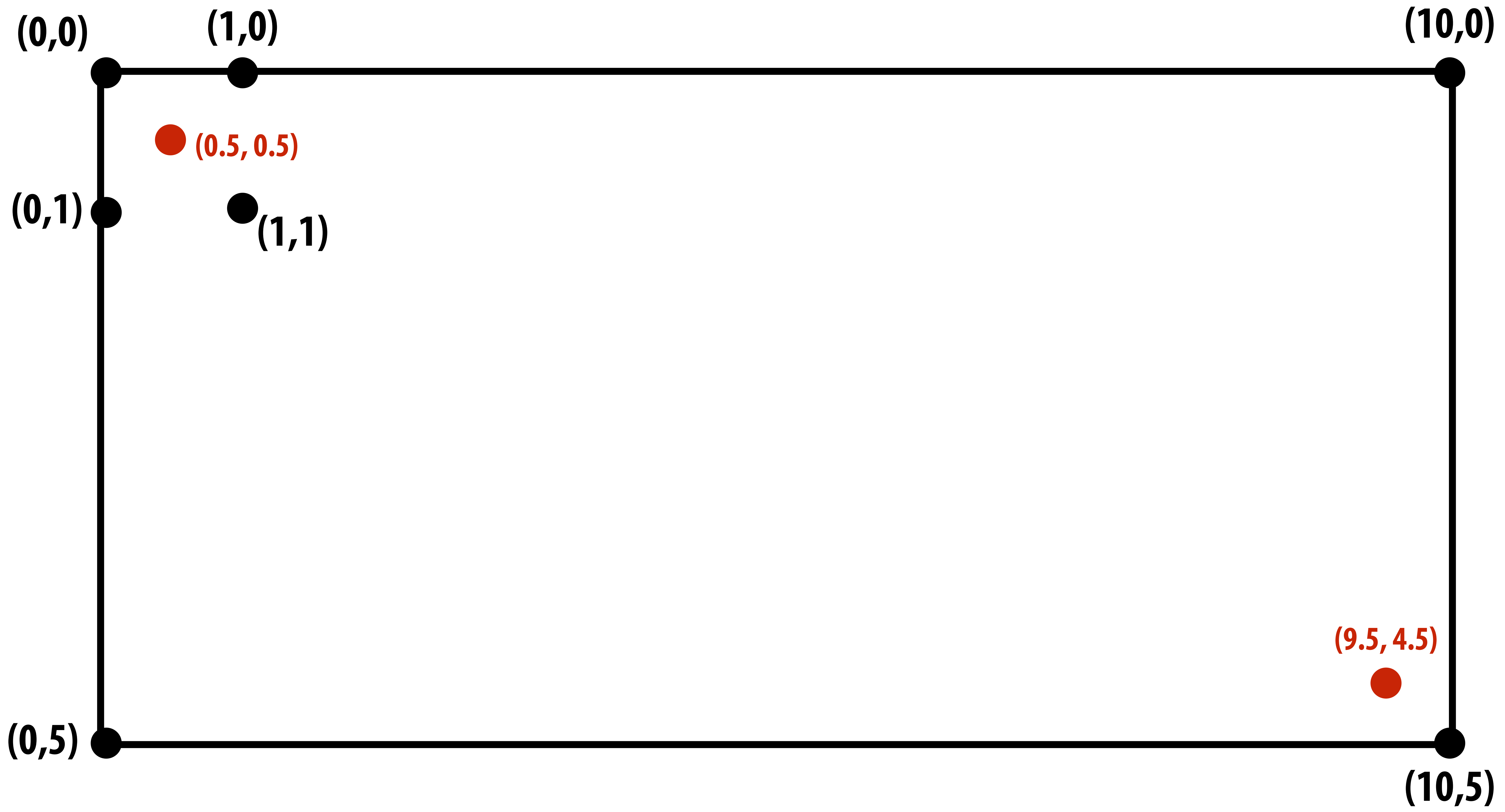
Continuous coordinate space over image



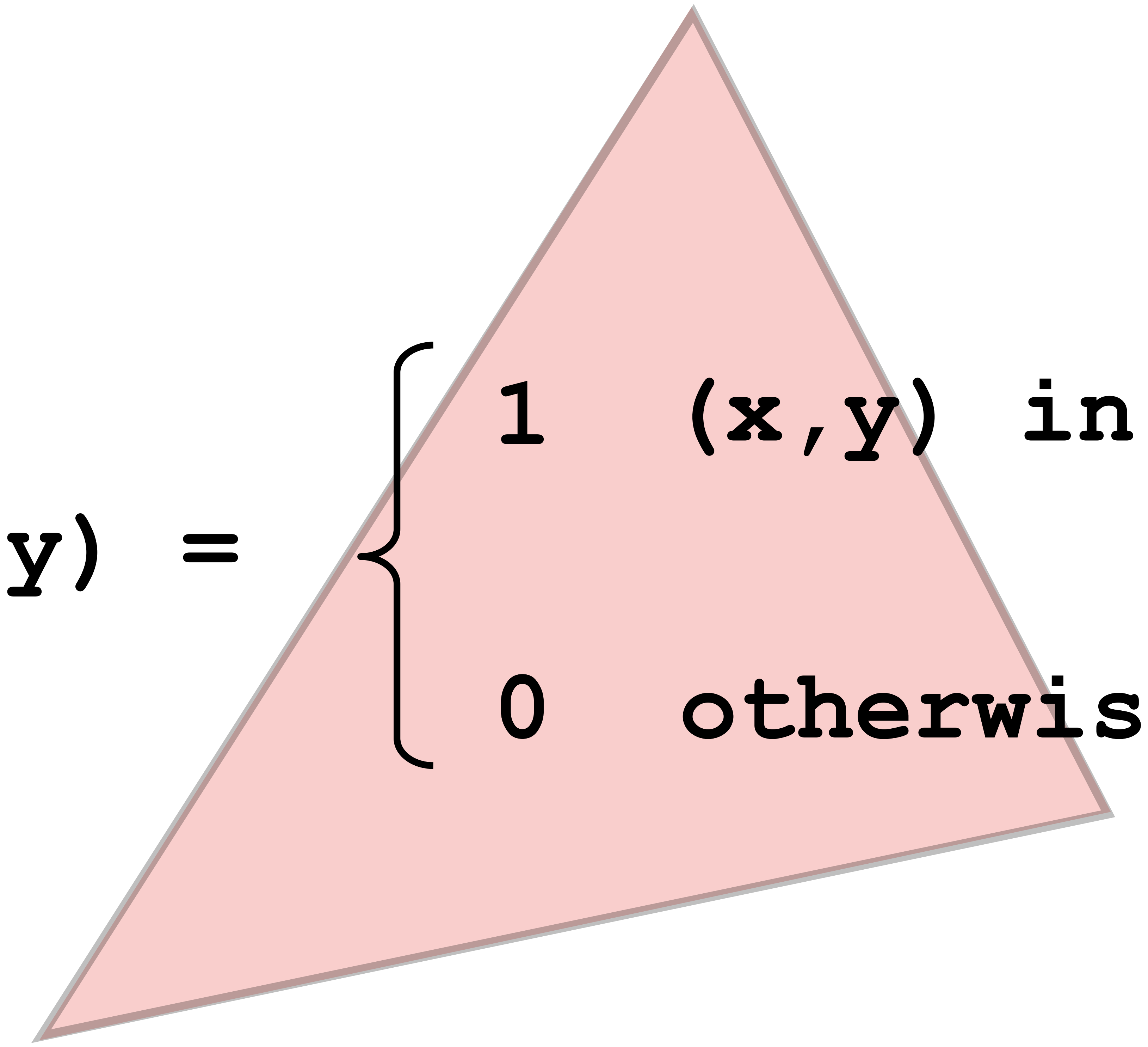
Continuous coordinate space over image

Ok, now forget about pixels!

(I removed pixel boundaries from the figure to encourage you to forget about pixels!)

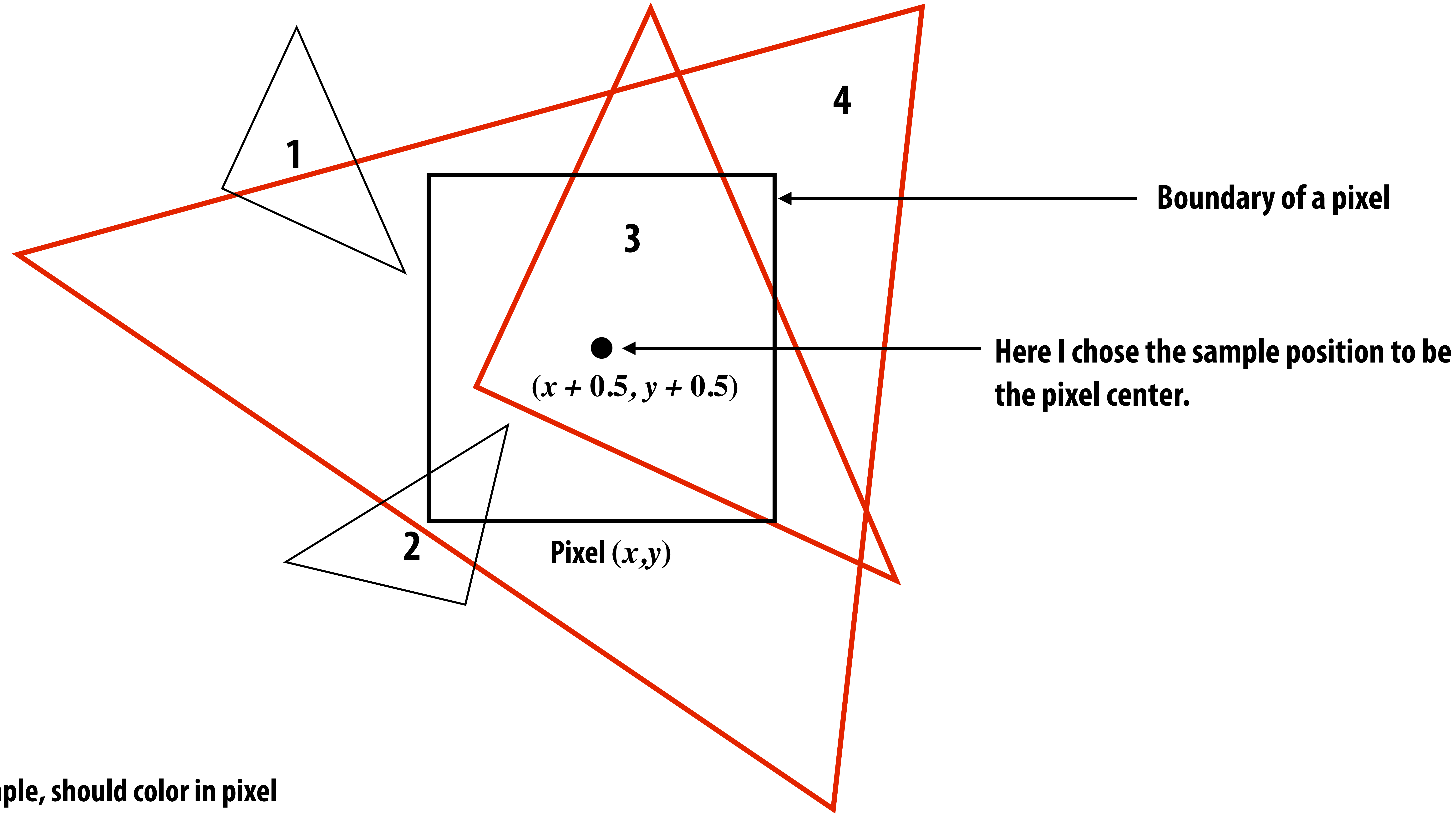


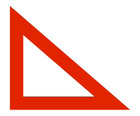
Define binary function: `inside(t,x,y)`

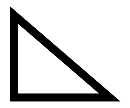


$$\texttt{inside(t,x,y)} = \begin{cases} \texttt{1} & \texttt{(x,y) in triangle t} \\ \texttt{0} & \texttt{otherwise} \end{cases}$$

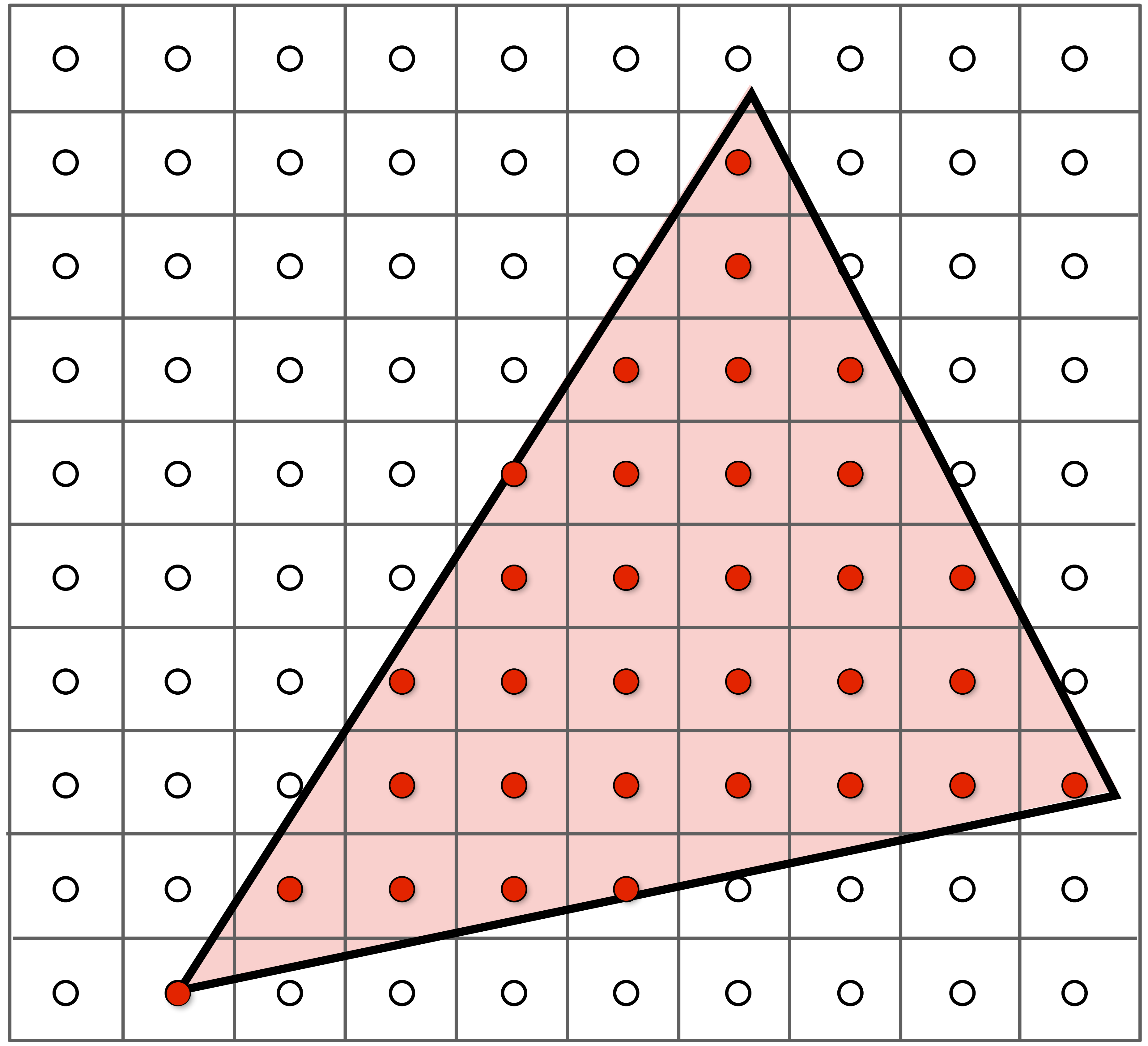
Sampling the binary function: `inside(t,x,y)`




= triangle covers sample, should color in pixel


= triangle does not cover sample, do not color in pixel

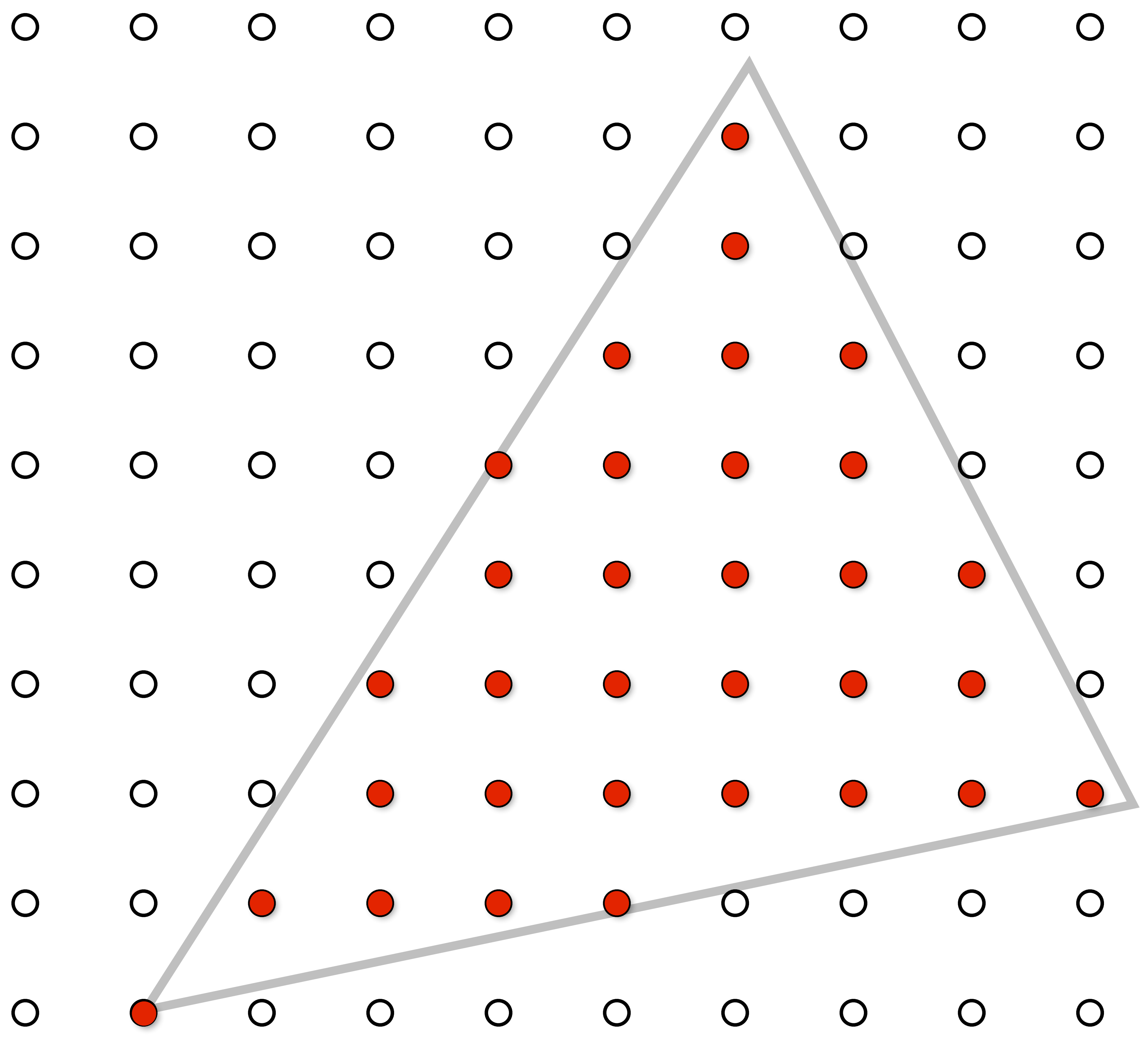
Sample coverage at pixel centers



Sample coverage at pixel centers

I only want you to think about evaluating triangle-point coverage!

NOT TRIANGLE-PIXEL OVERLAP!



Rendering = sampling a 2D binary function

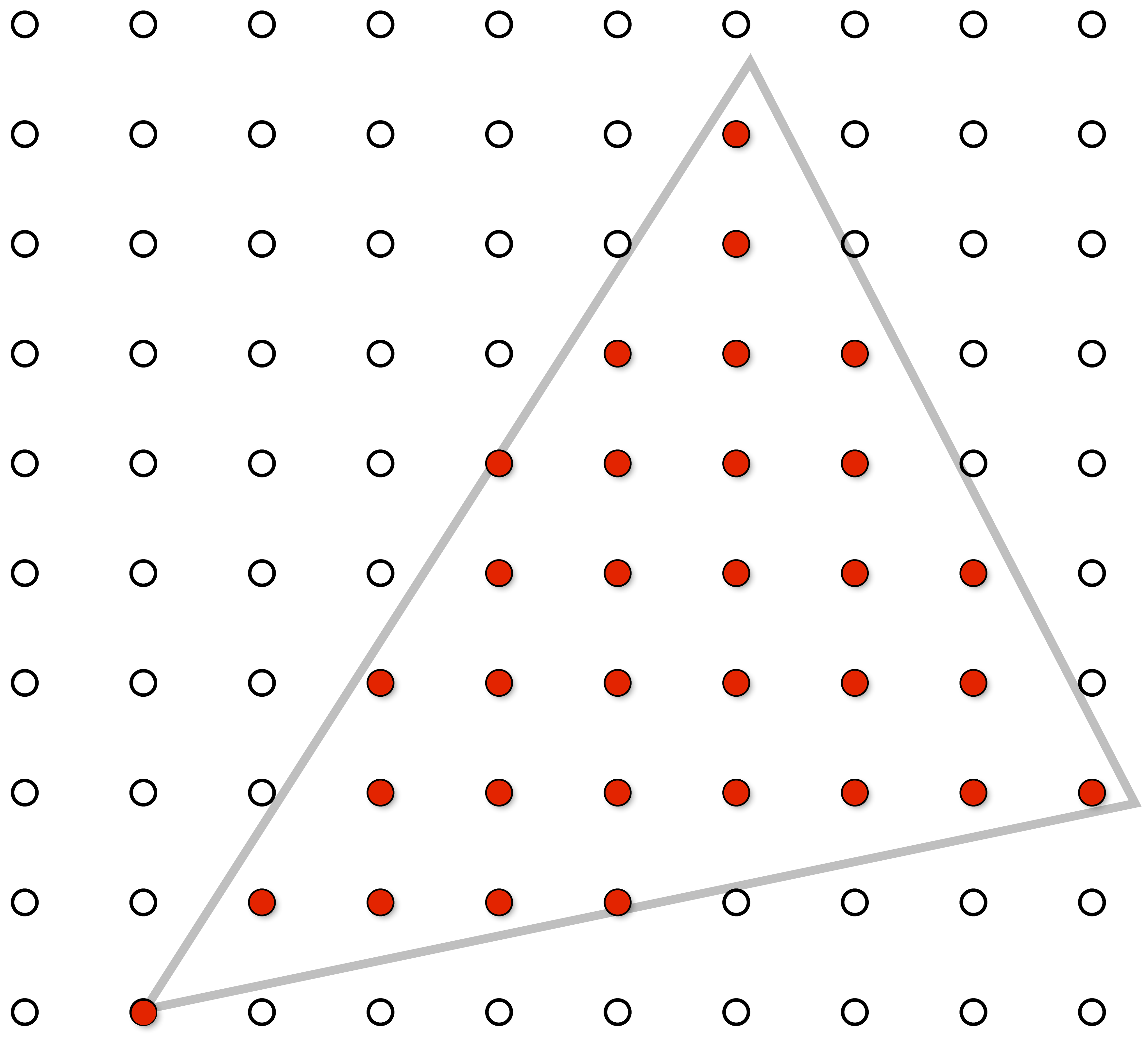
- **The basic top-level rendering loop for sampling visibility**

```
for (int x = 0; x < xmax; x++)
  for (int y = 0; y < ymax; y++)
    image[x][y] = f(x + 0.5, y + 0.5);
```

- **For a rasterizer:** `f(x,y) = pointInsideTriangle(ProjectVerts(t), x, y)`
- **For a ray caster:** `f(x,y) = rayTriangleIsect(t, RayFromScreenCoord(x,y))`

Where are we now

- We have the ability to determine if any point in the image is inside or outside the triangle

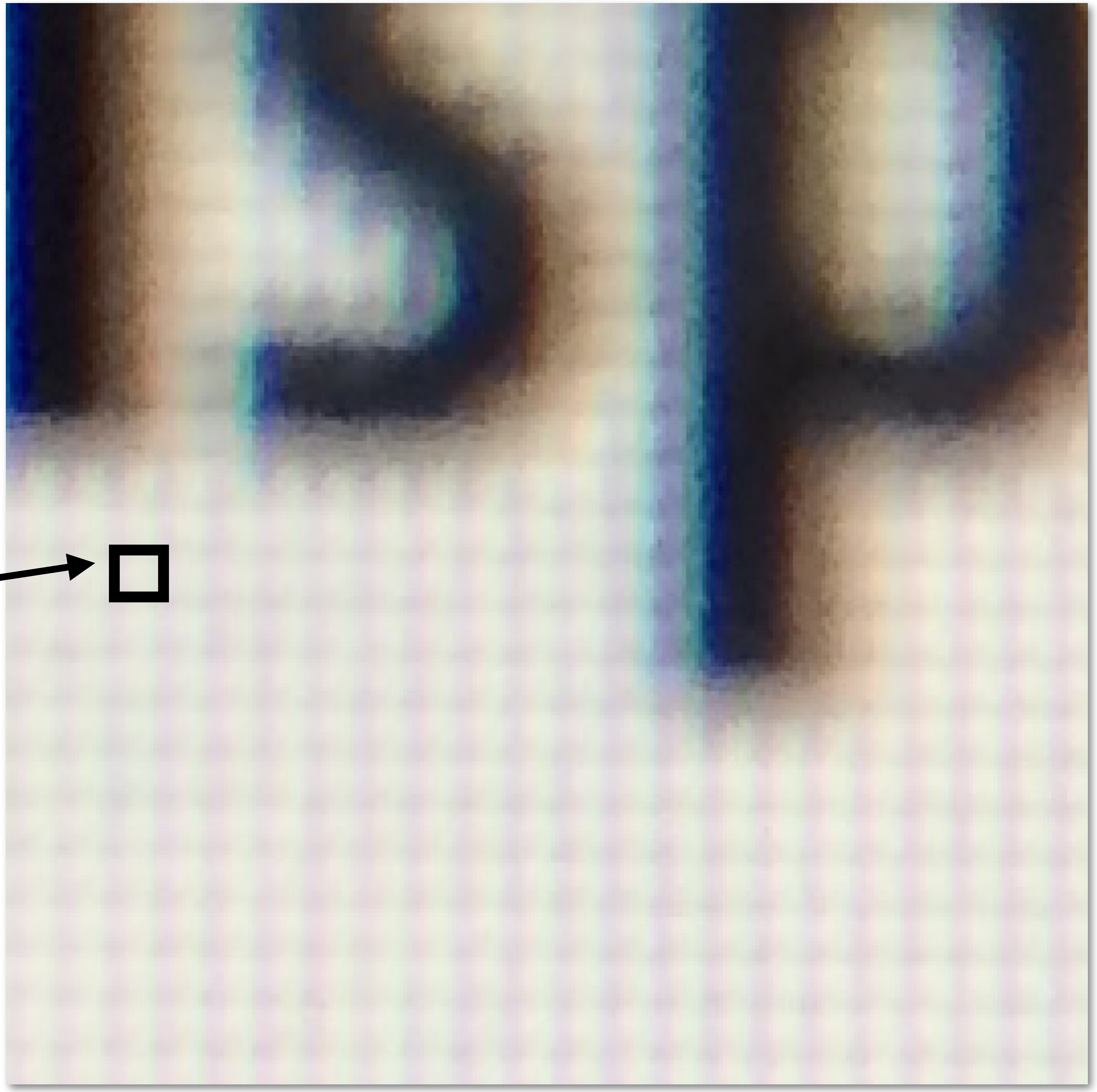


- How to we interpret these results as an image to display?
  (Recall, there's no pixels above, just samples)

Recall: pixels on a screen

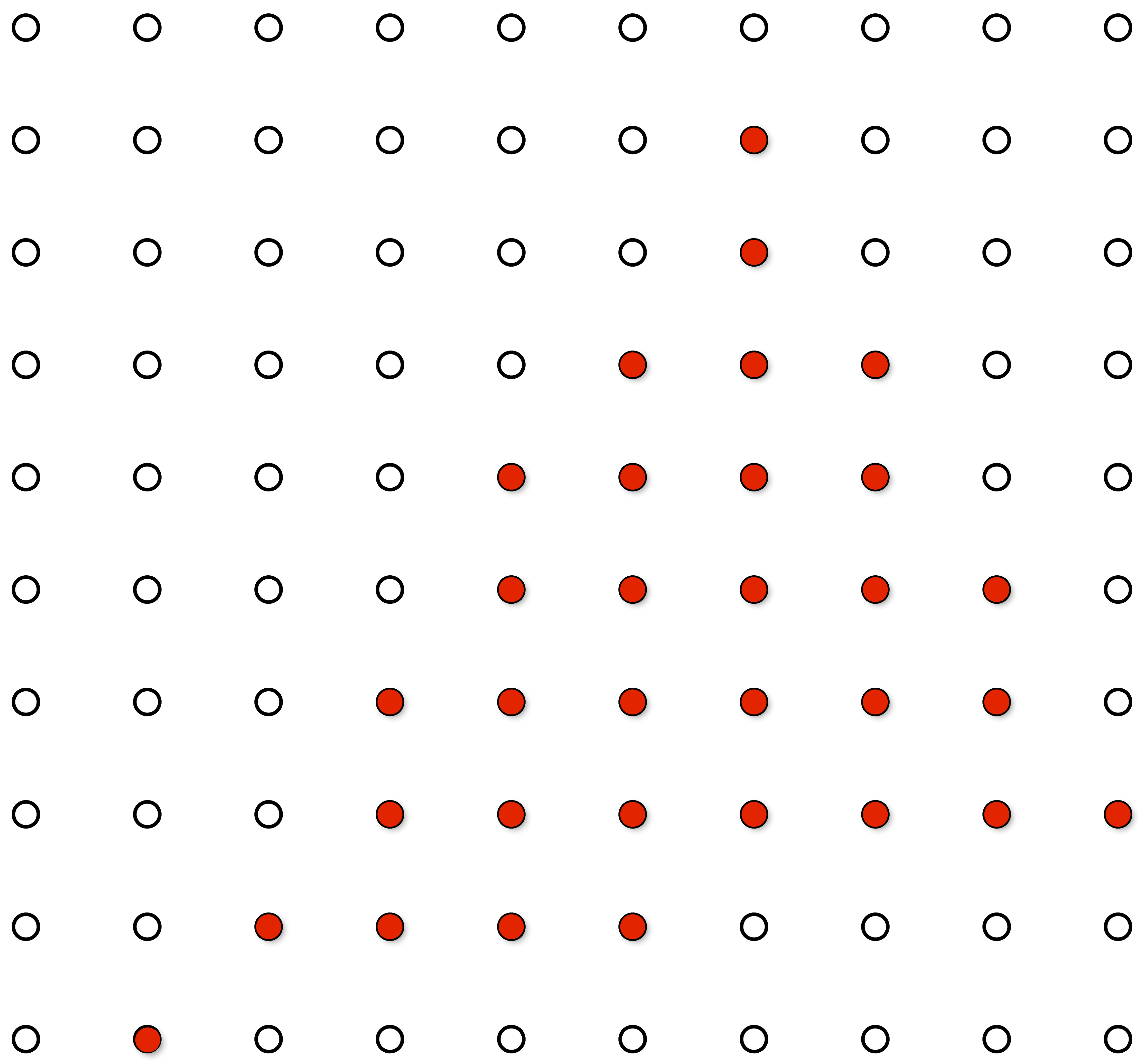
Each image sample sent to the display is converted into a little square of light of the appropriate color:
(a pixel = picture element)

Laptop display pixel



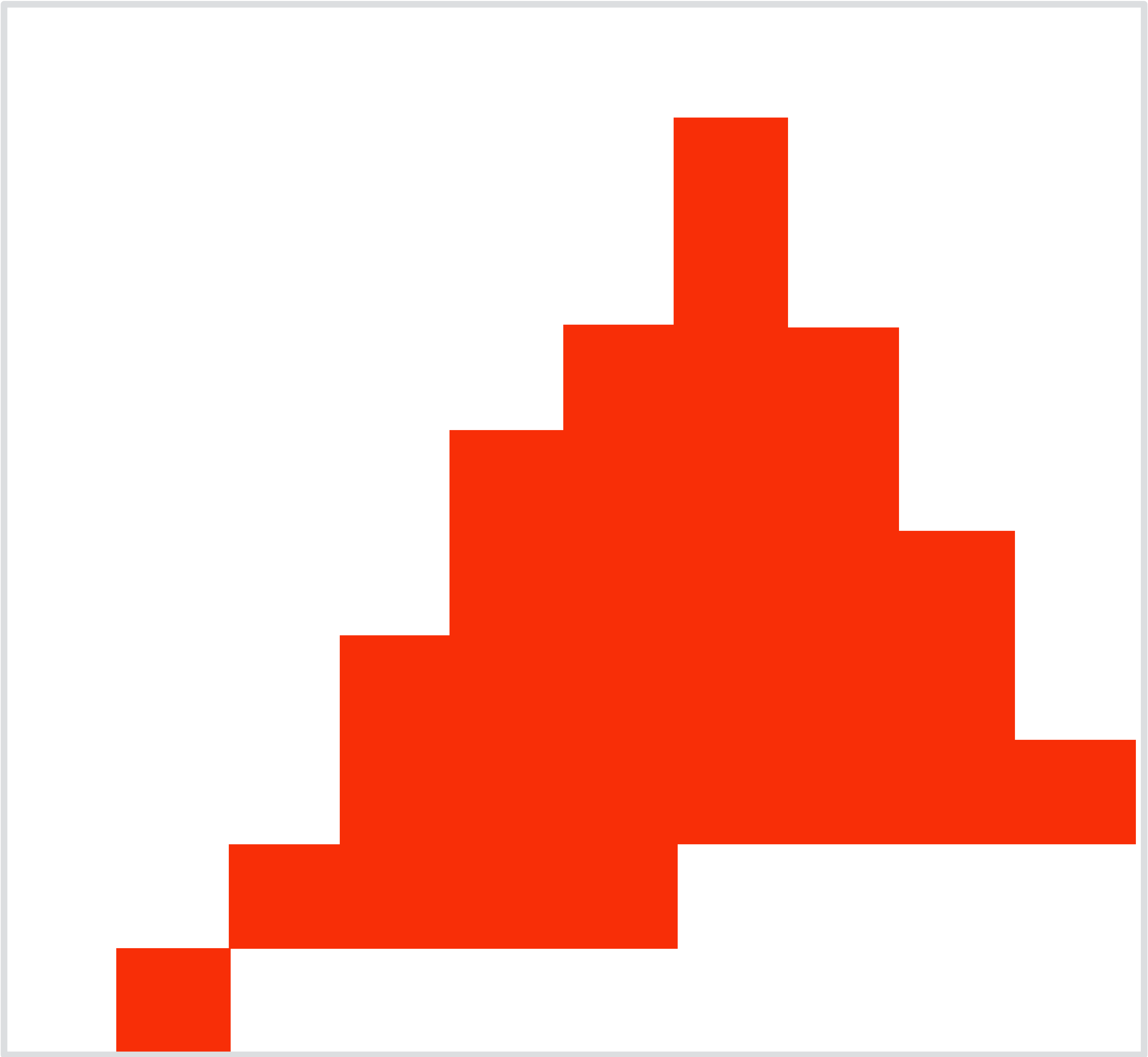
*** Thinking of each screen pixel as emitting a square of uniform intensity light of a single color is an approximation to how real displays work, but it will do for now.**

So, if we send the display this sampled signal…



…and each value determines the light emitted from a pixel…

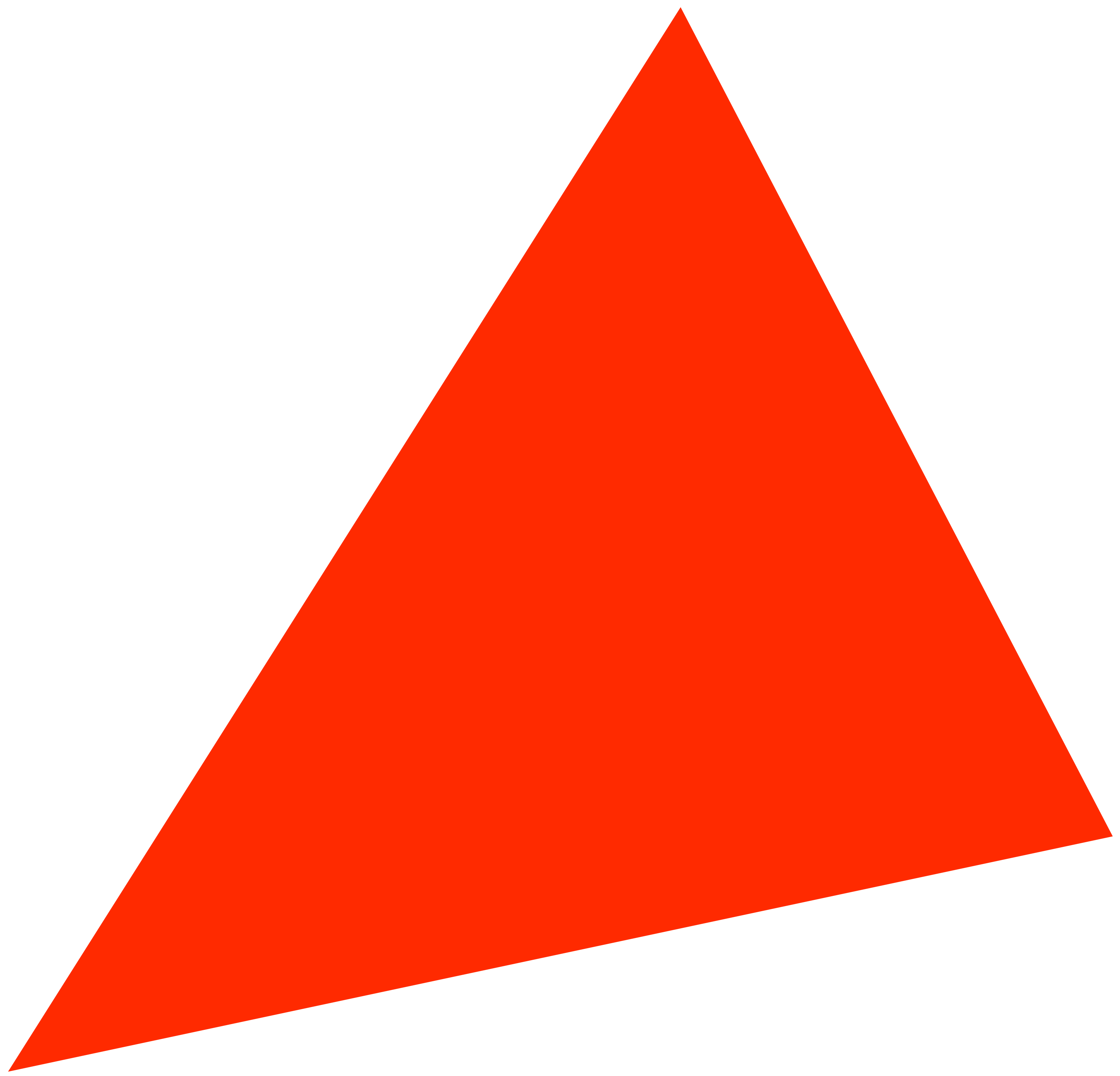
The display physically emits this signal



Given our simplified "square pixel" display assumption, the emitted light is a piecewise constant reconstruction of the samples

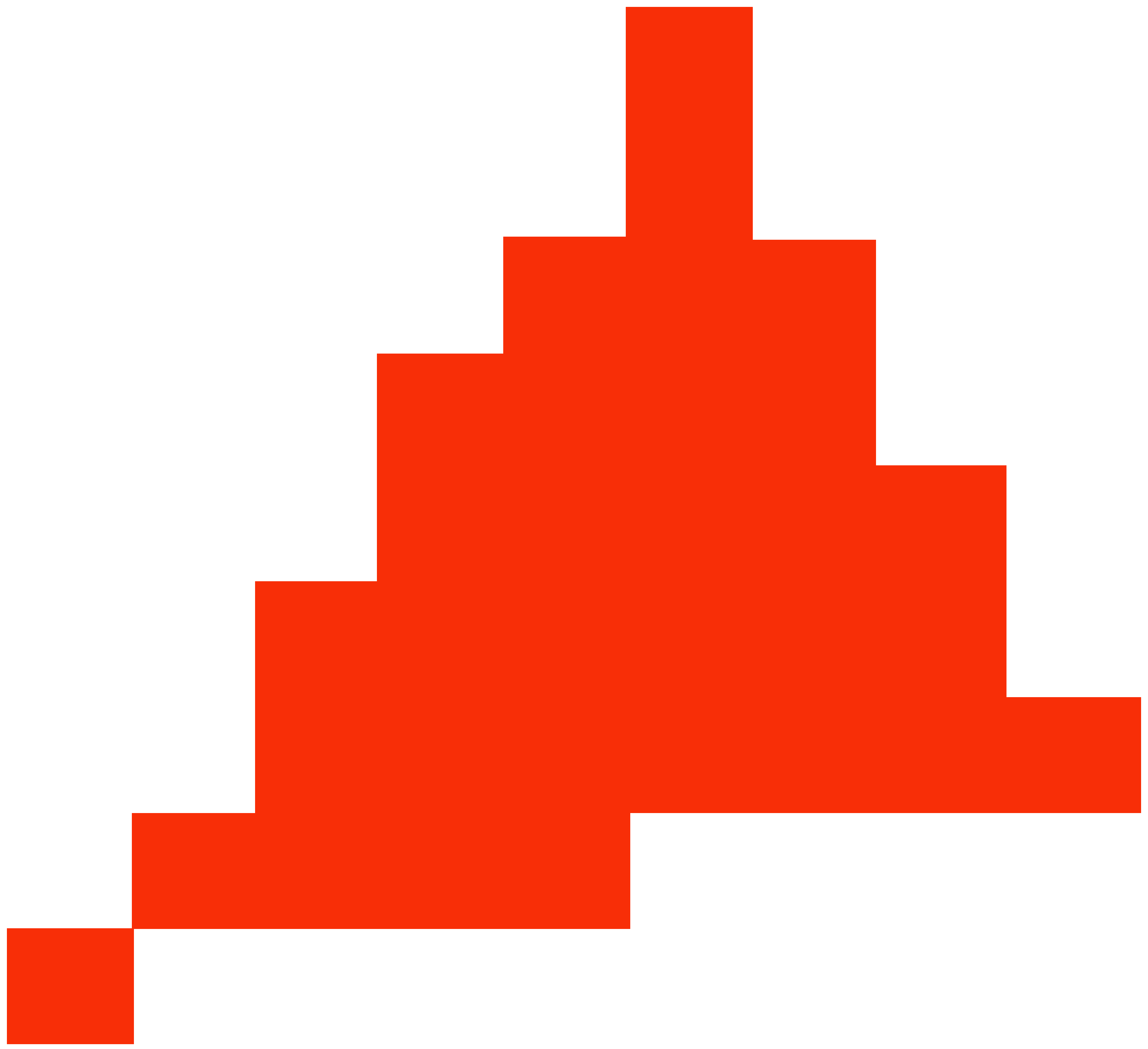
Compare: the continuous triangle function

(This is the function we sampled)



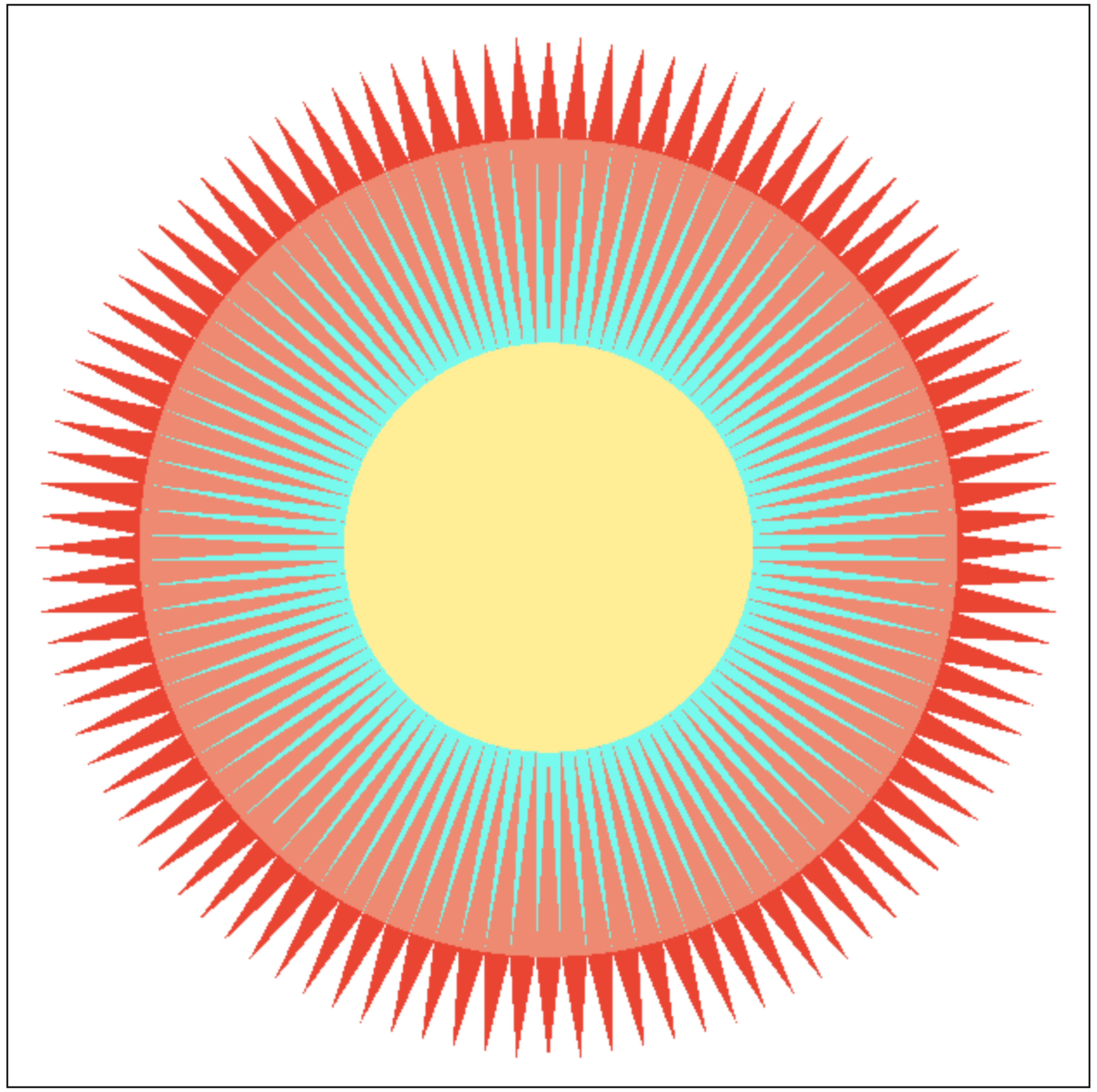
What's wrong with this picture?

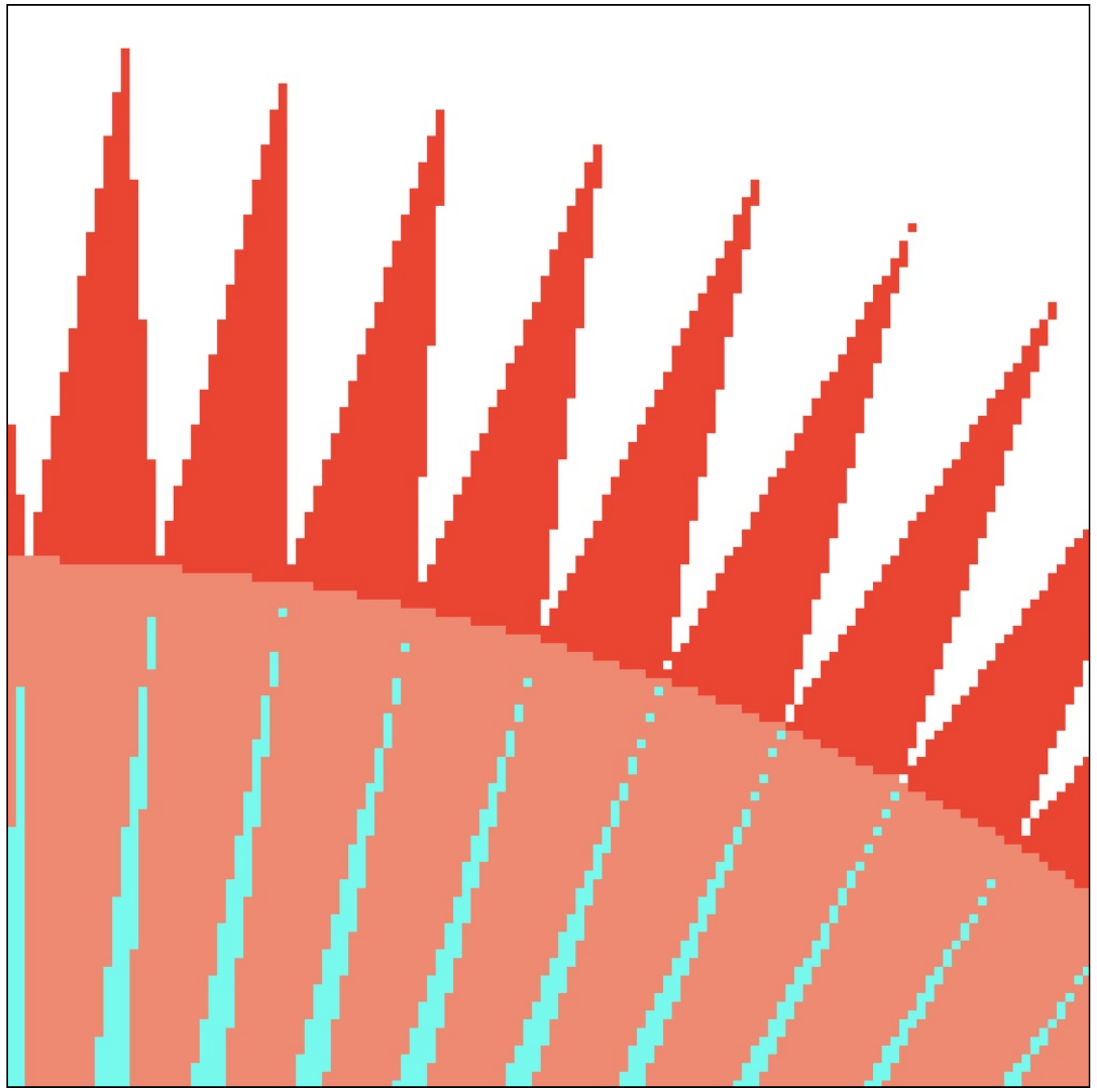
(This is the reconstruction emitted by the display)



Jaggies!

Jaggies (staircase pattern)

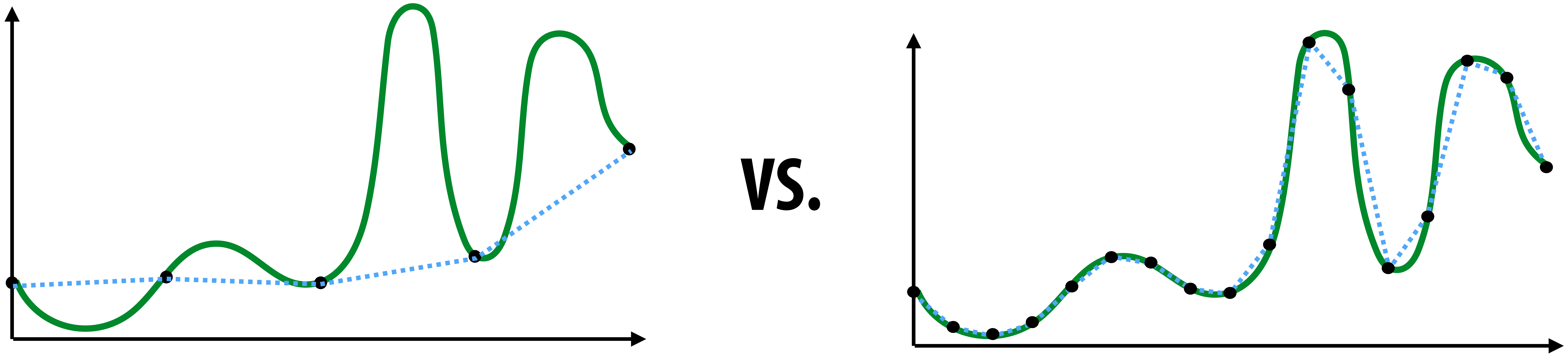




Is this the best we can do?

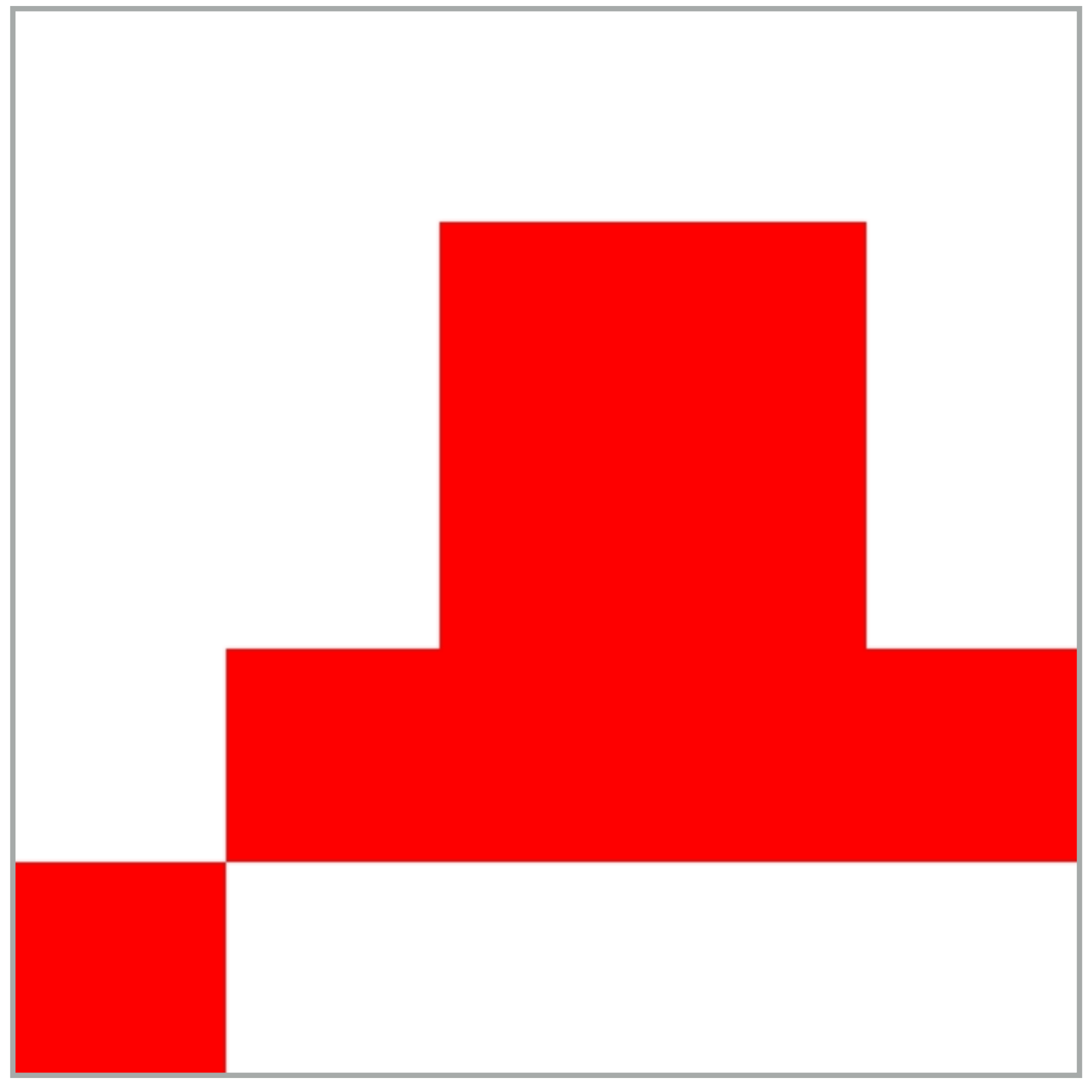
Reminder: how can we represent a signal more accurately?

Sample signal more densely! (increase sampling rate)

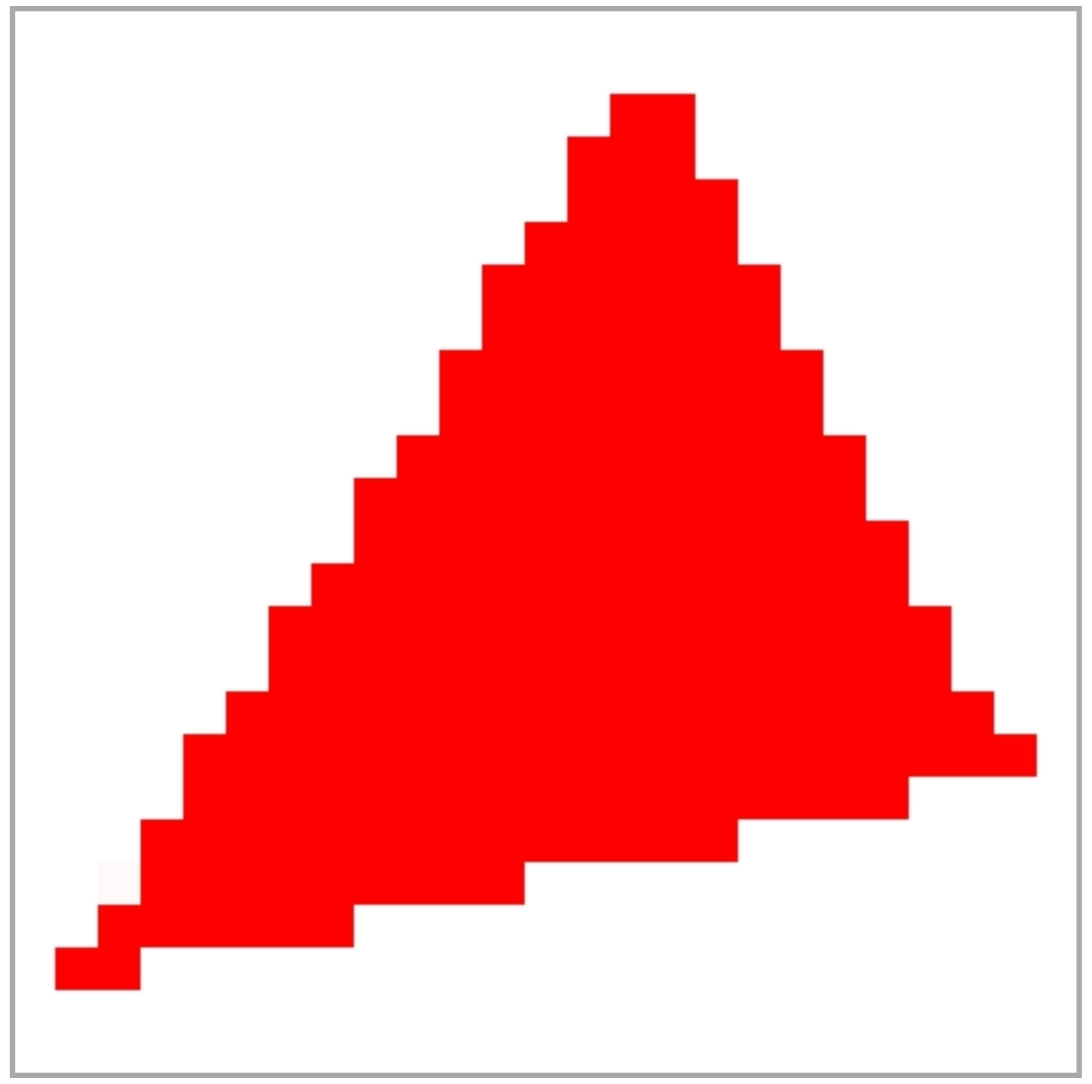


One solution: increase image resolution

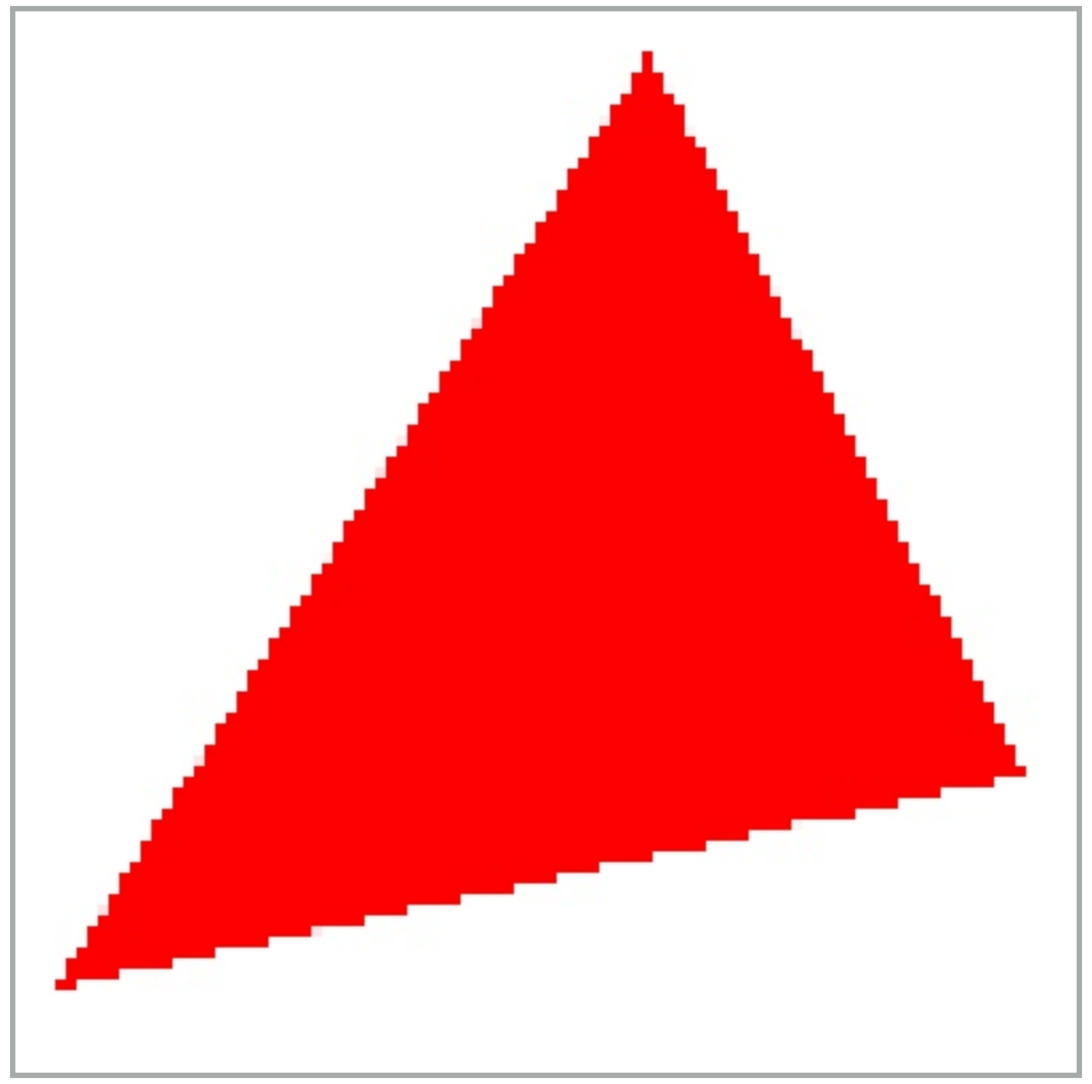
Increase number of pixels in image —> denser sampling of signal



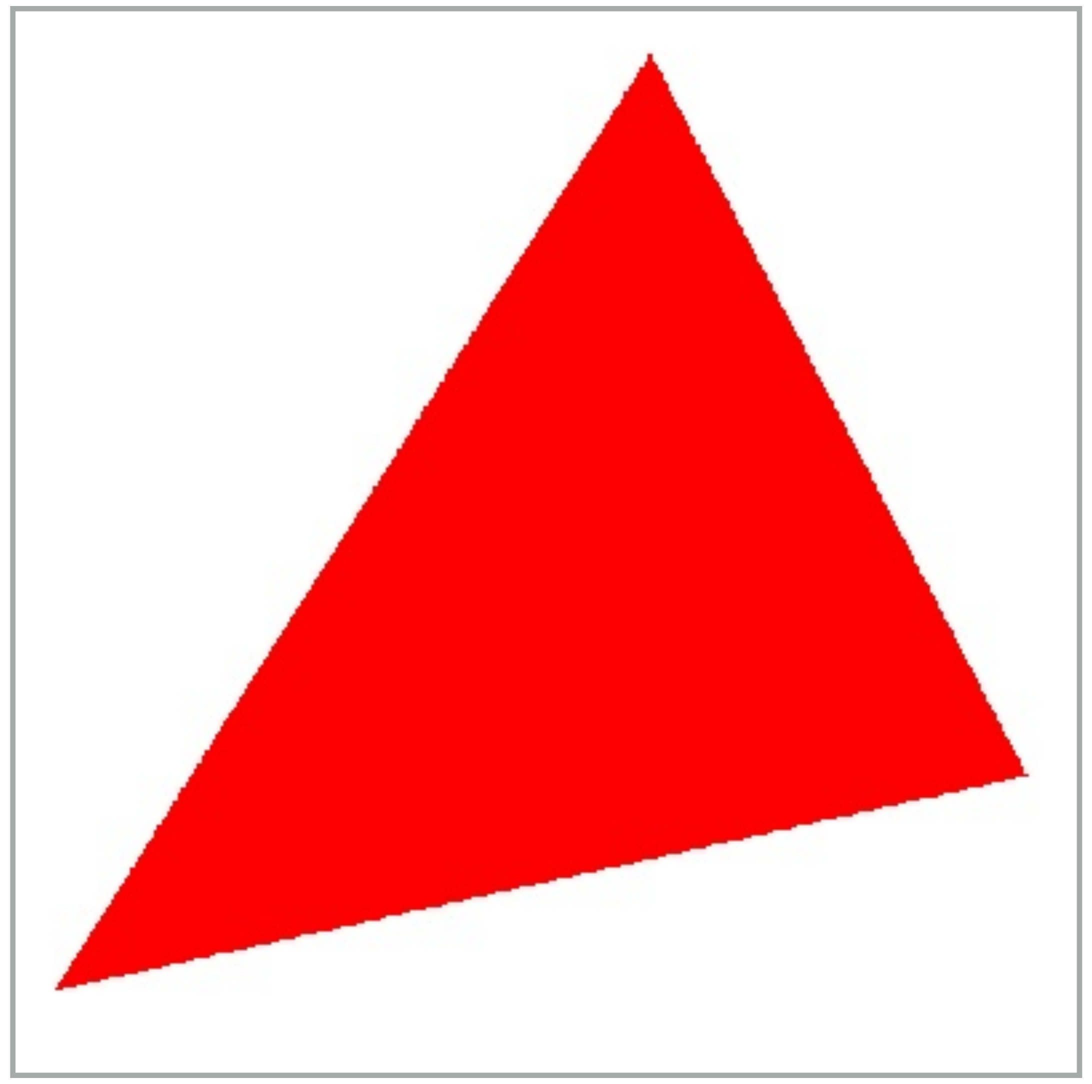
5x5 image



25x25 image



100x100 image



400x400 image

Increasing resolution of displays can be costly

8K TV
7680 x 4320 pixels
(32.7 megapixels)

About $3000 at Best Buy in Jan 2026

I don't think you can buy a 16K TV in 2026, although Sony demo'ed on in 2019 for a few million $$$

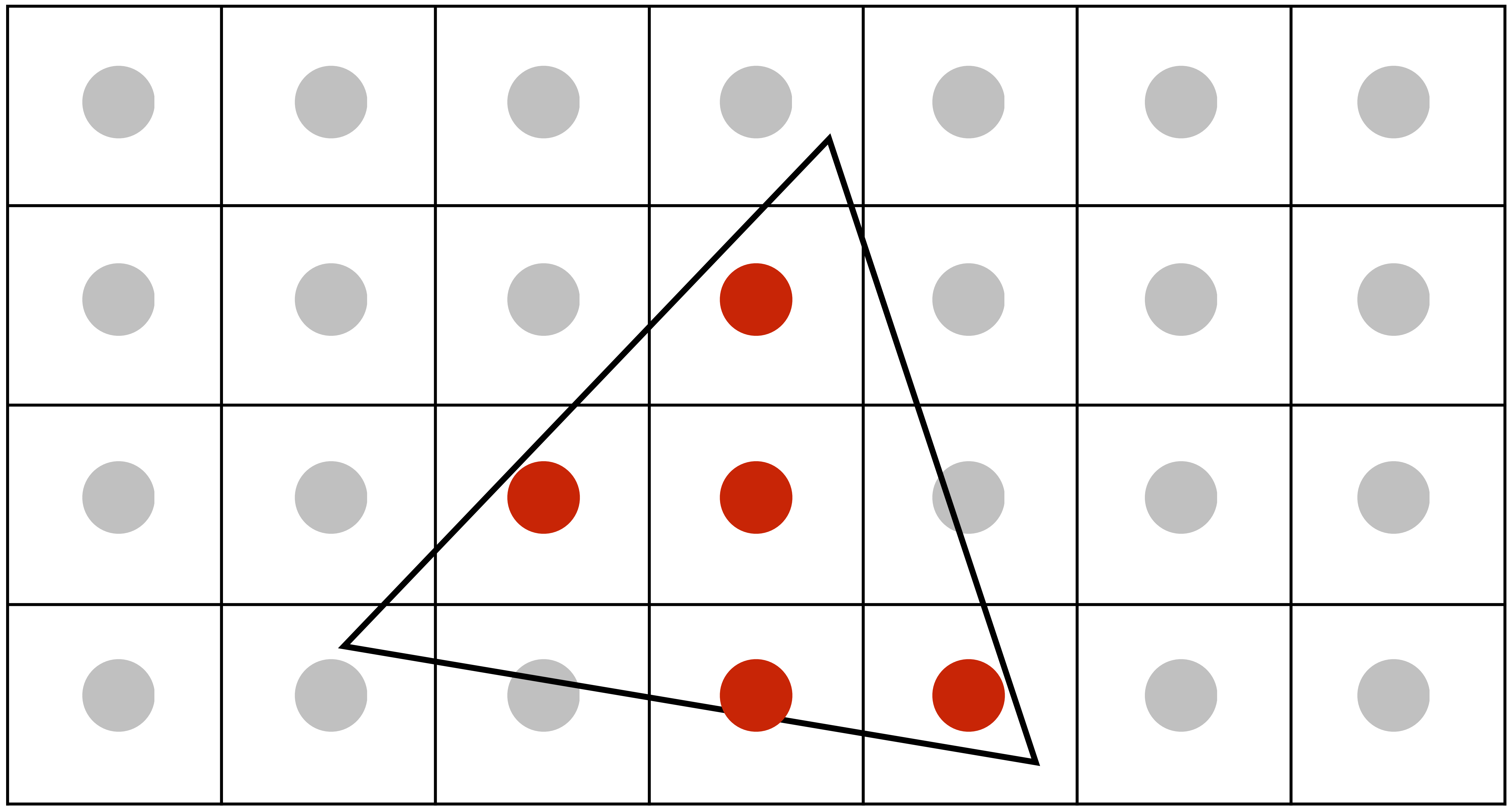
iPhone 12
2532 x 1170 pixels
(2.9 megapixels)

About 460 pixels per inch

Let’s say we want to render a high-quality image for a given display.

(We have to accept a given number of pixels)

Sampling using one sample per pixel

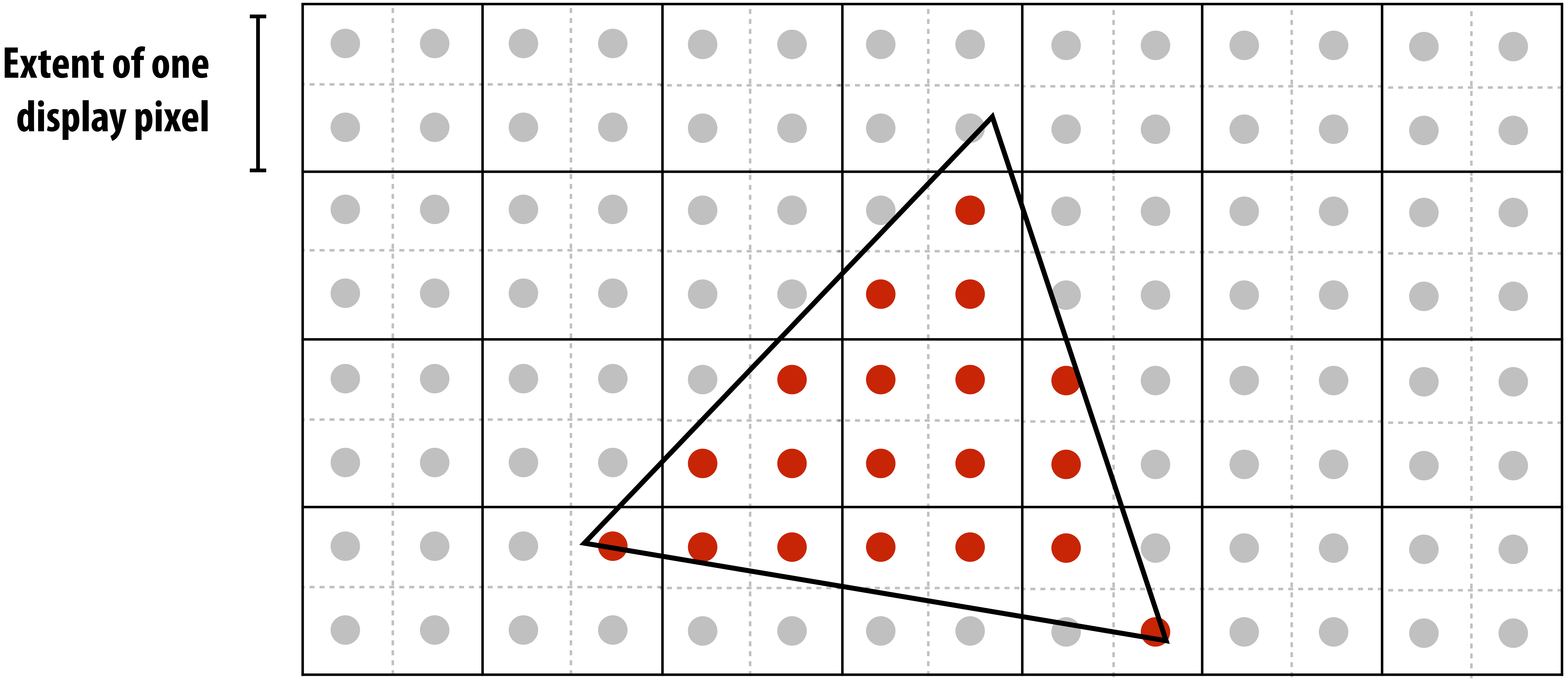


Supersampling: step 1

Sample the input signal more densely in the image plane

In this example: take four samples in the area spanned by a pixel

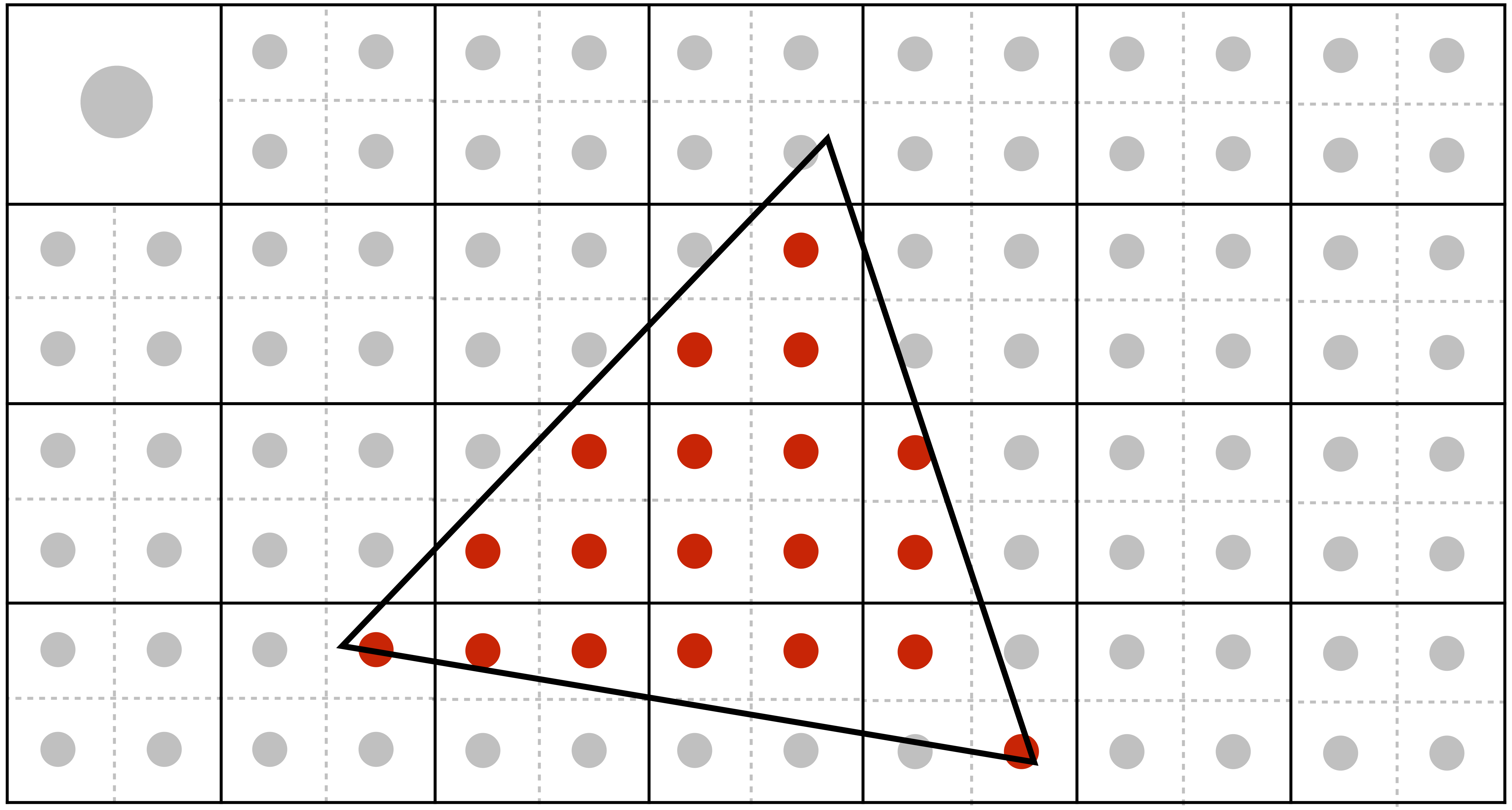
2x2 supersampling



But how do we use these samples to drive a display, since there are four times more samples than display pixels? 🤔

Supersampling: step 2

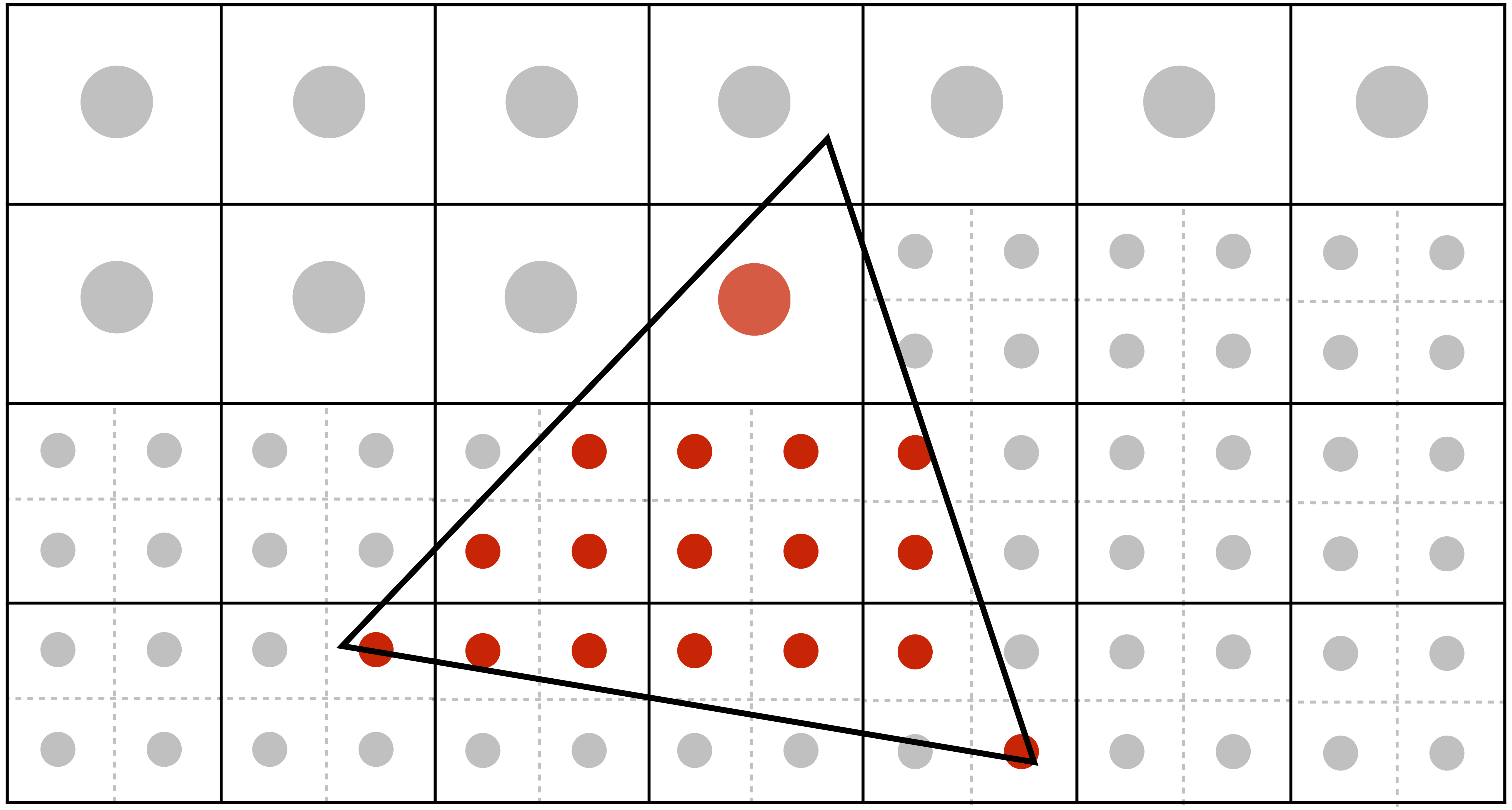
Average the N x N samples “inside” each pixel



Averaging down

Supersampling: step 2

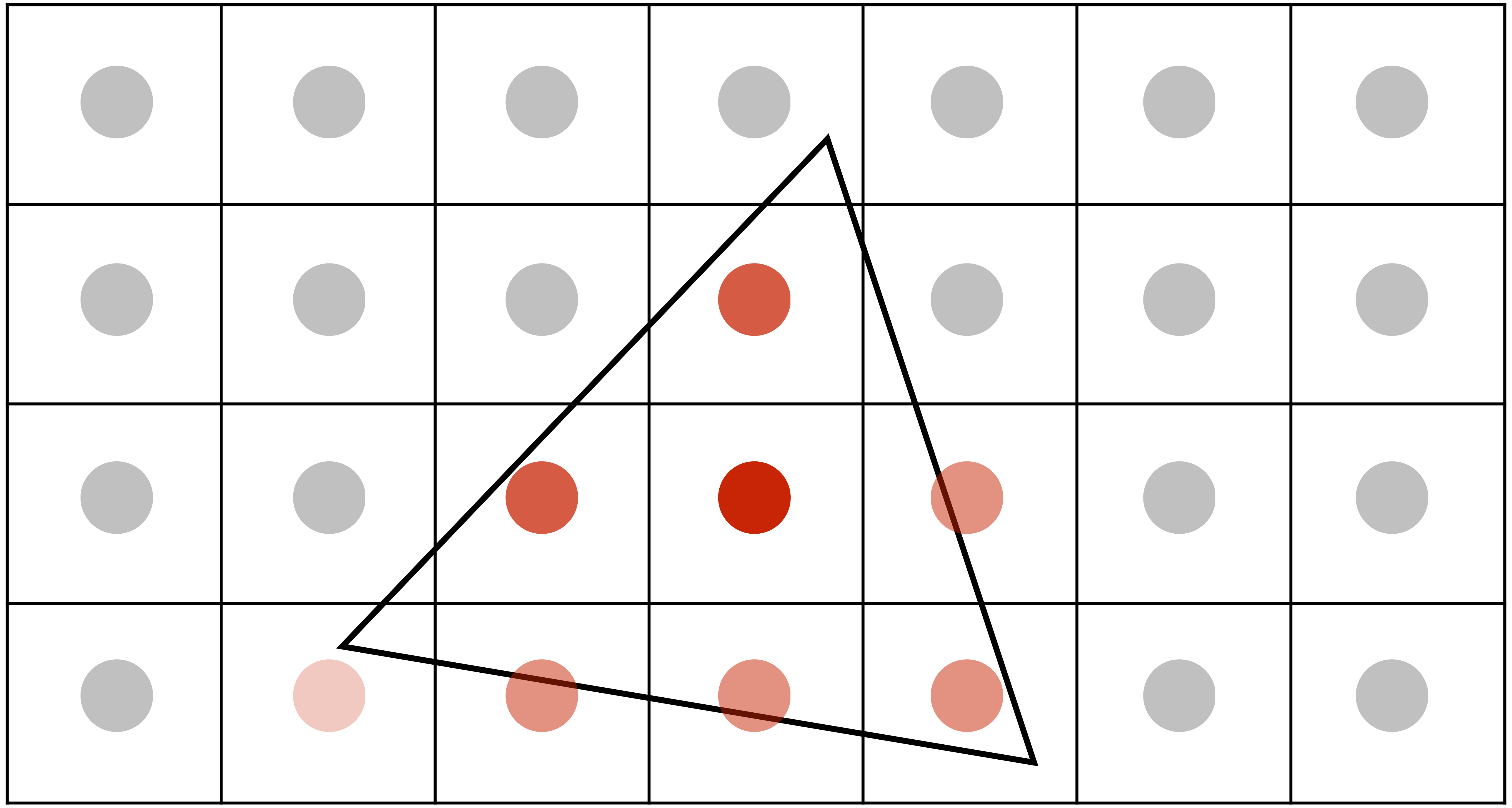
Average the N x N samples "inside" each pixel



Averaging down

Supersampling: step 2

Average the N x N samples "inside" each pixel

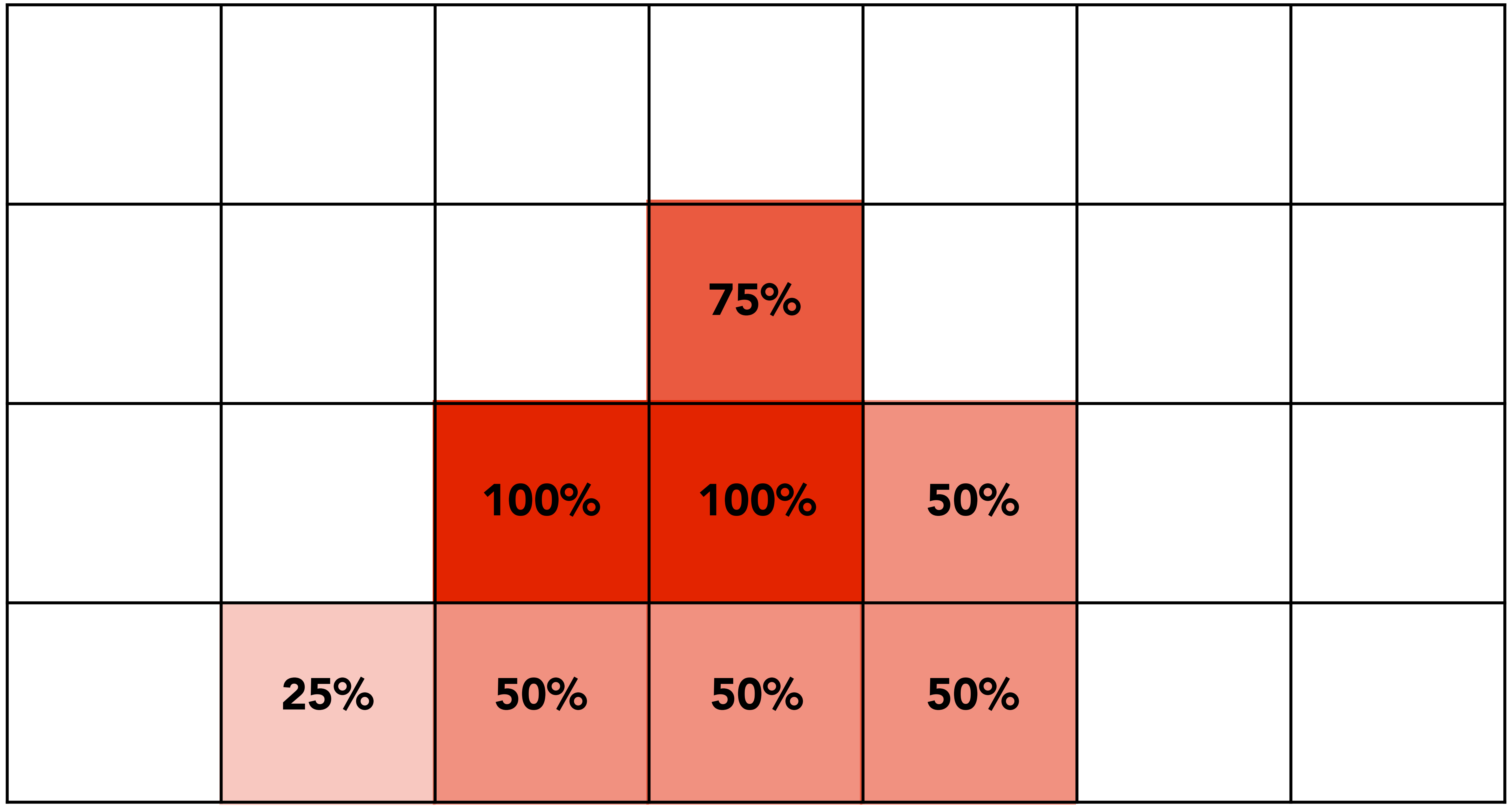


Averaging down

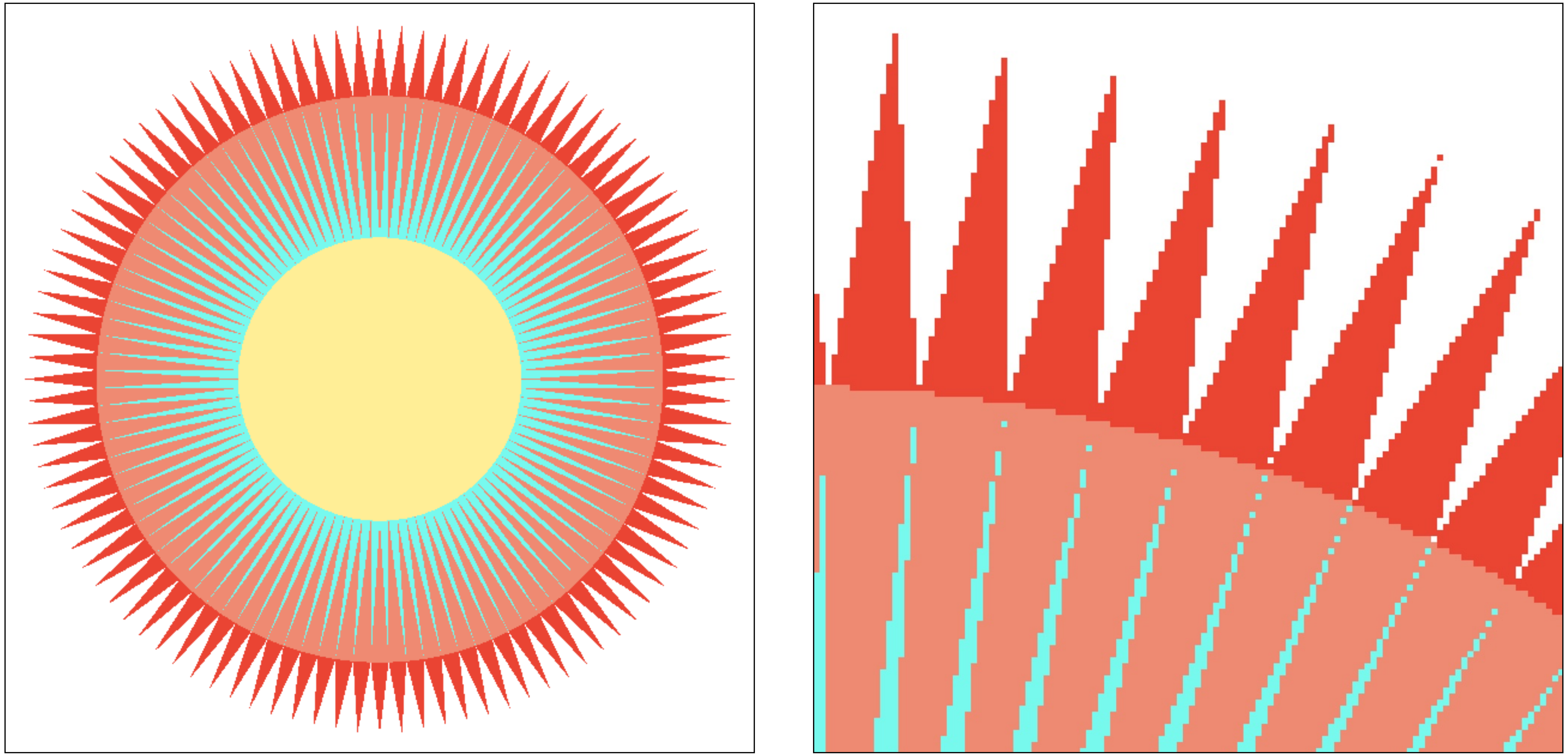
Displayed result

This is the corresponding signal emitted by the display

(The value provided to each display pixel is the average of the values sampled in that region)

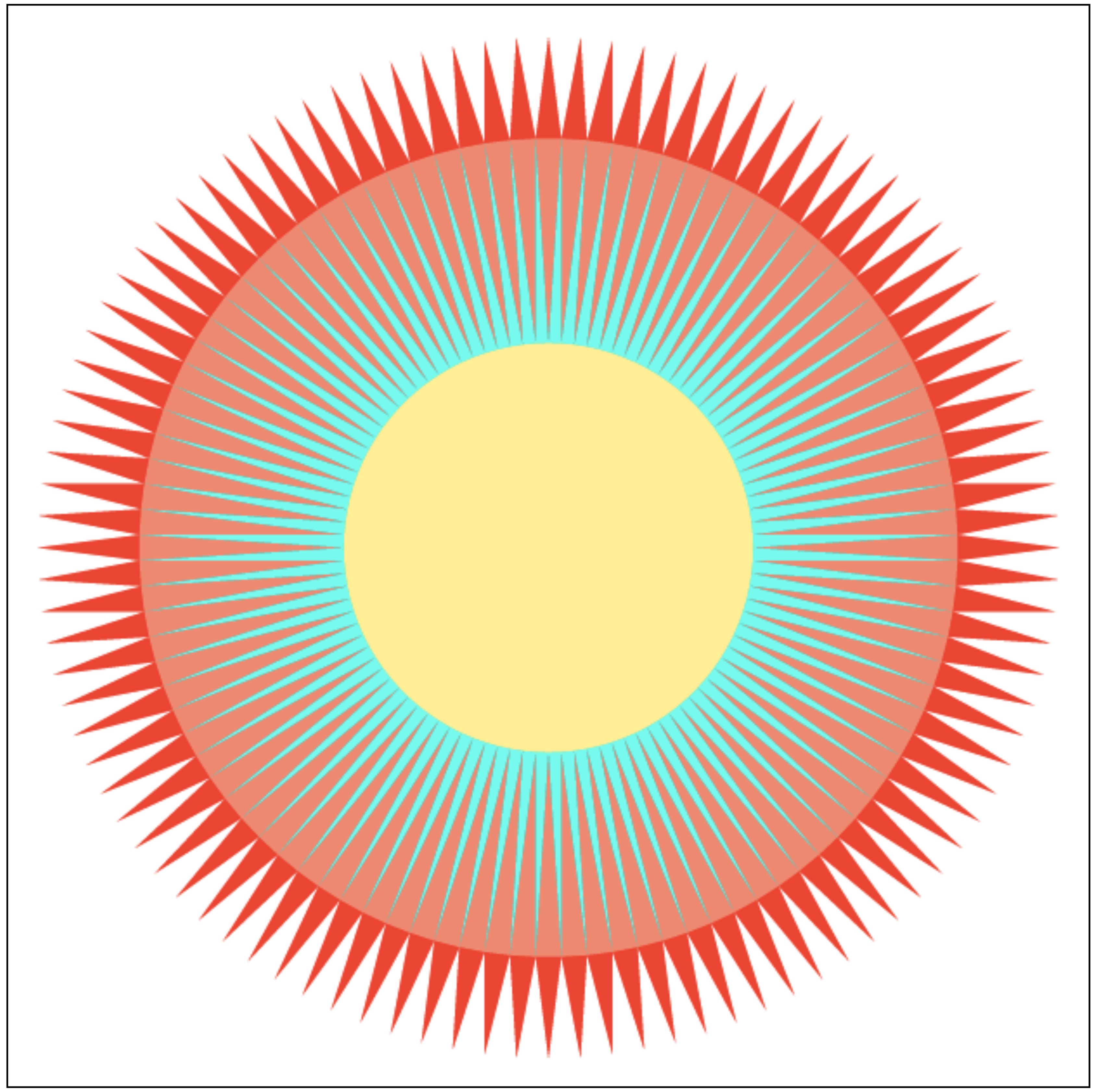


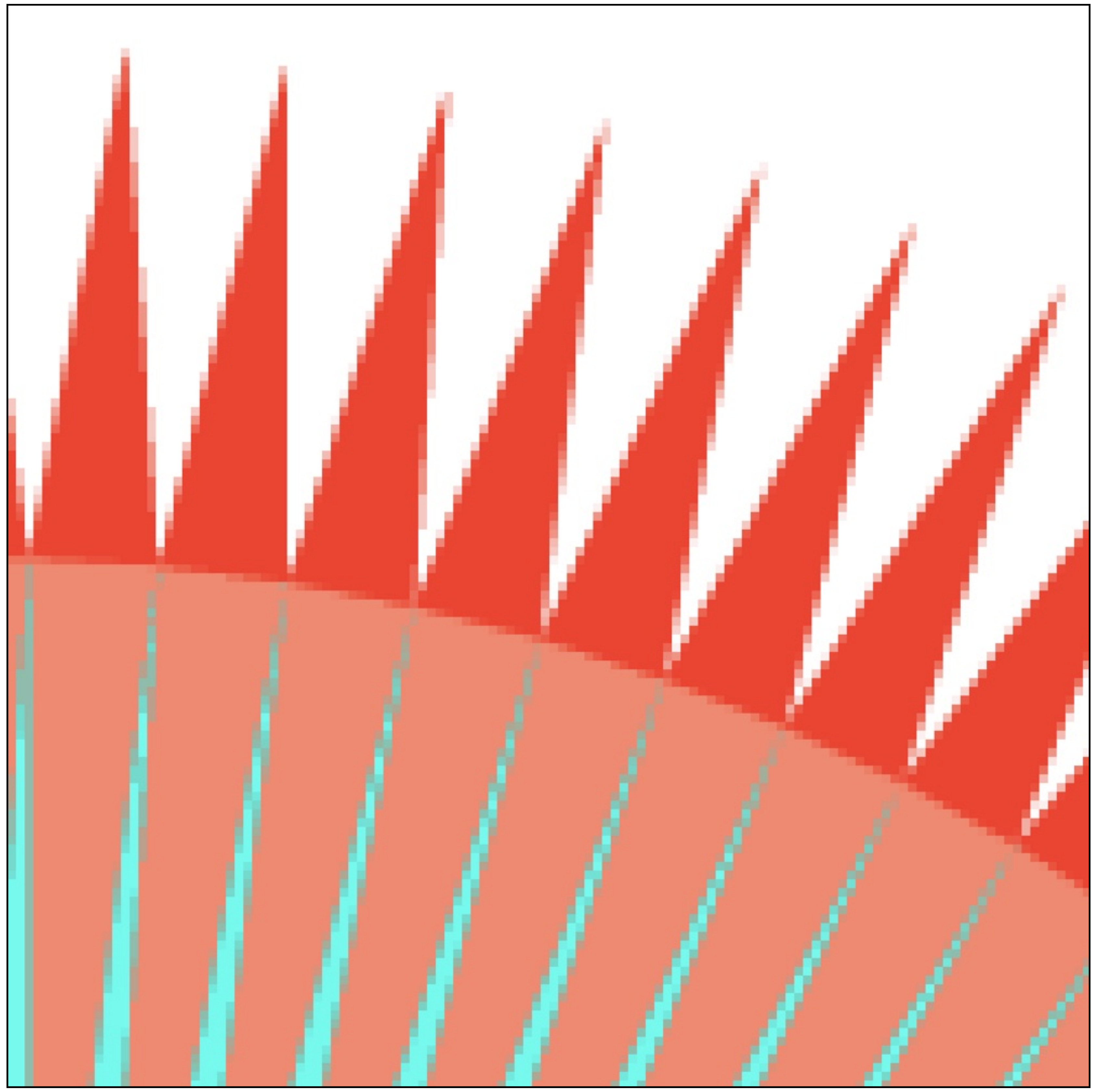
Images rendered using one sample per pixel



4x4 supersampling + downsampling

(16 samples per pixel)



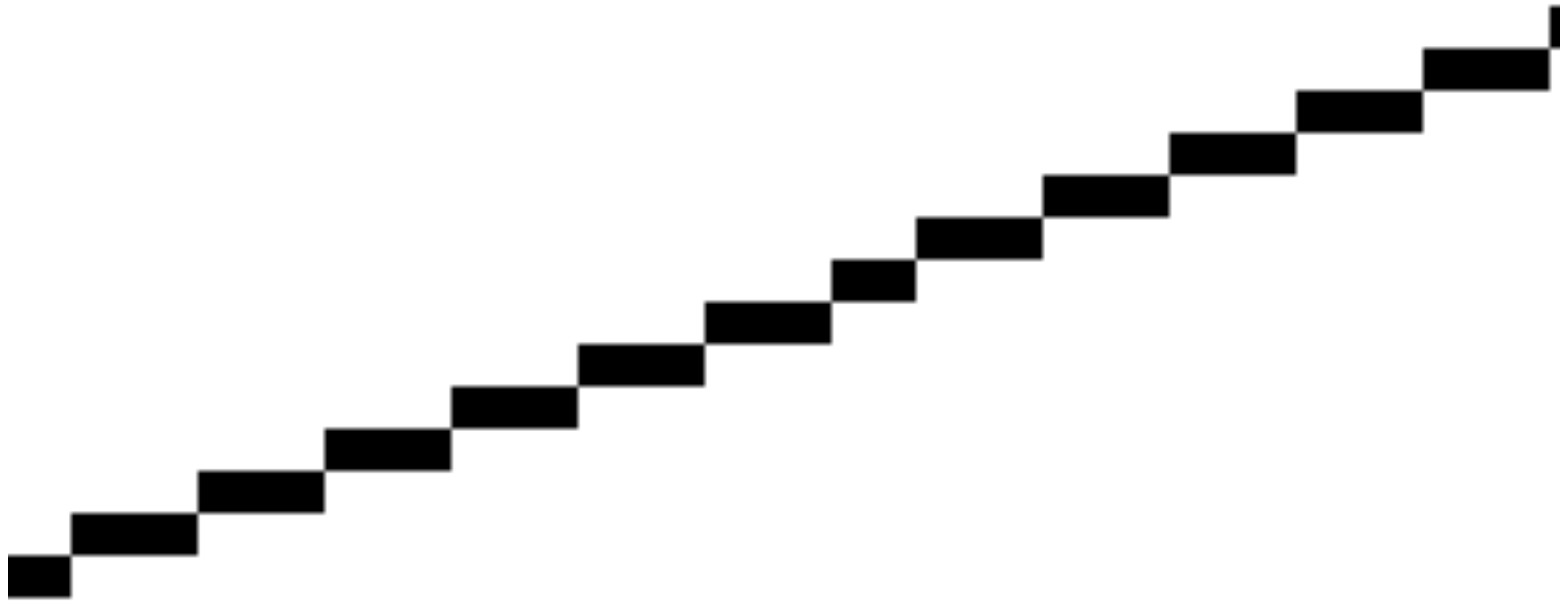


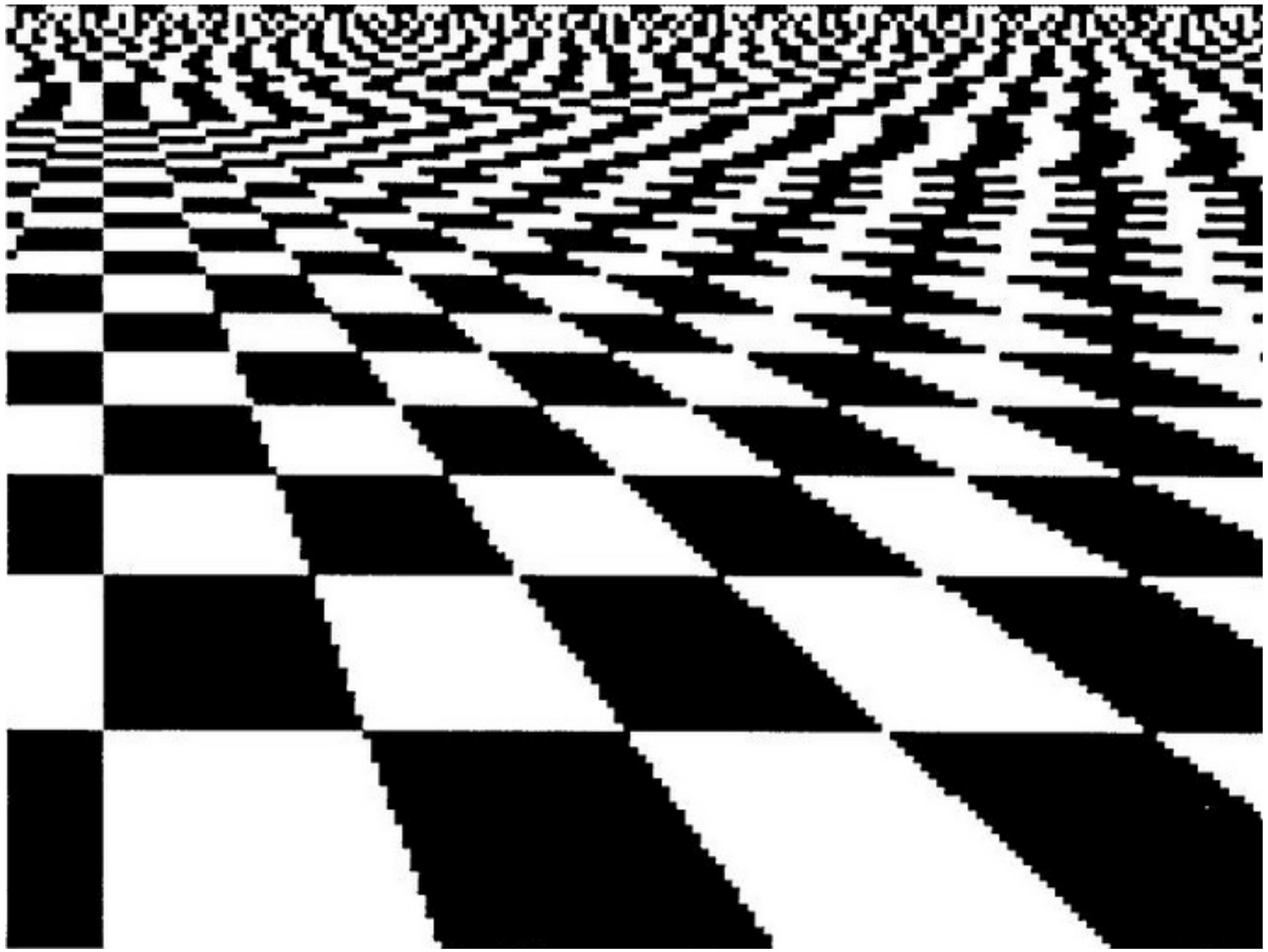
The images above contain same number of pixels as the images on the prior slide.
But now each pixel's value is the average of the 16 samples taken per pixel.

Let's understand what just happened in a more principled way

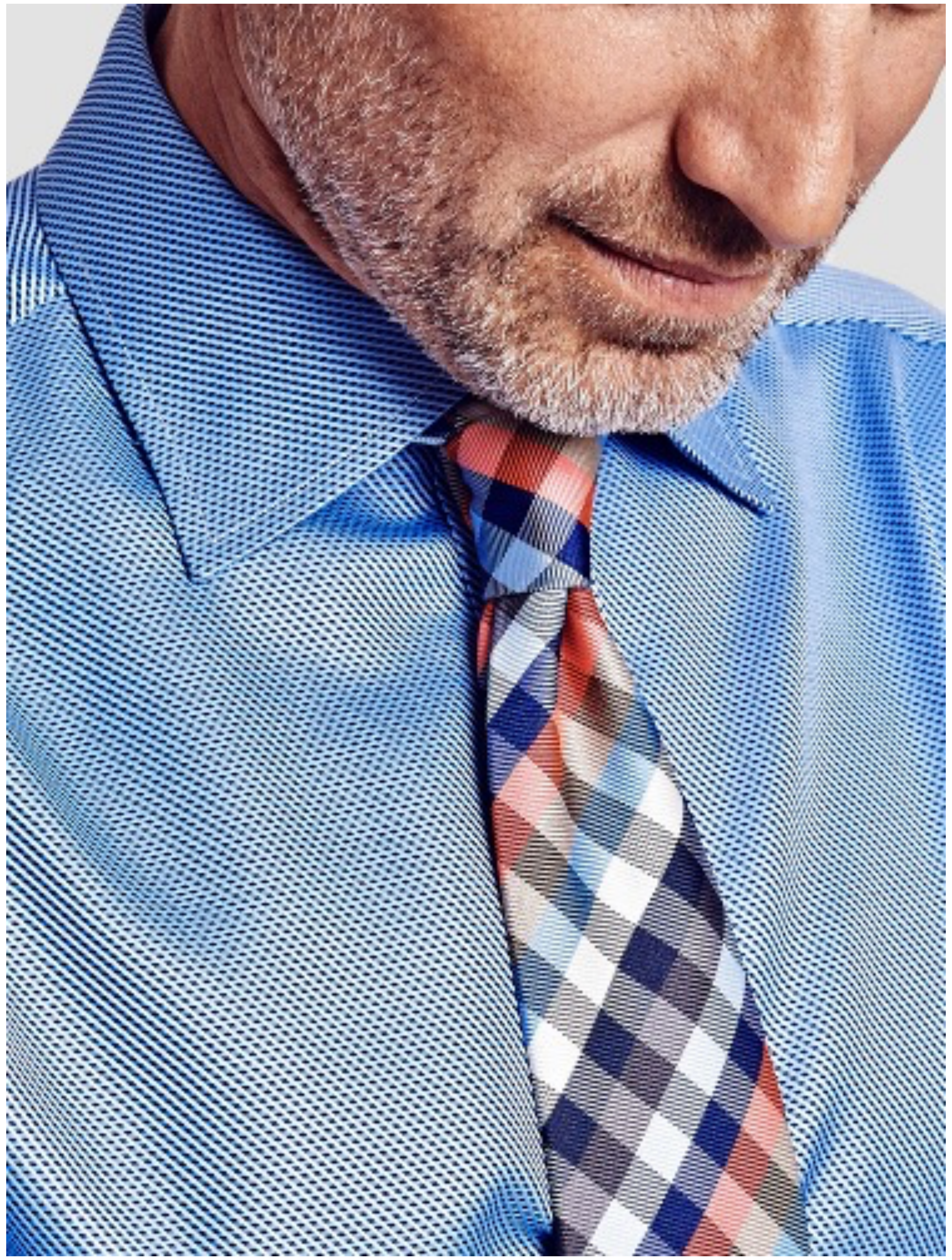
More examples of sampling artifacts in computer graphics

Jaggies (staircase pattern)

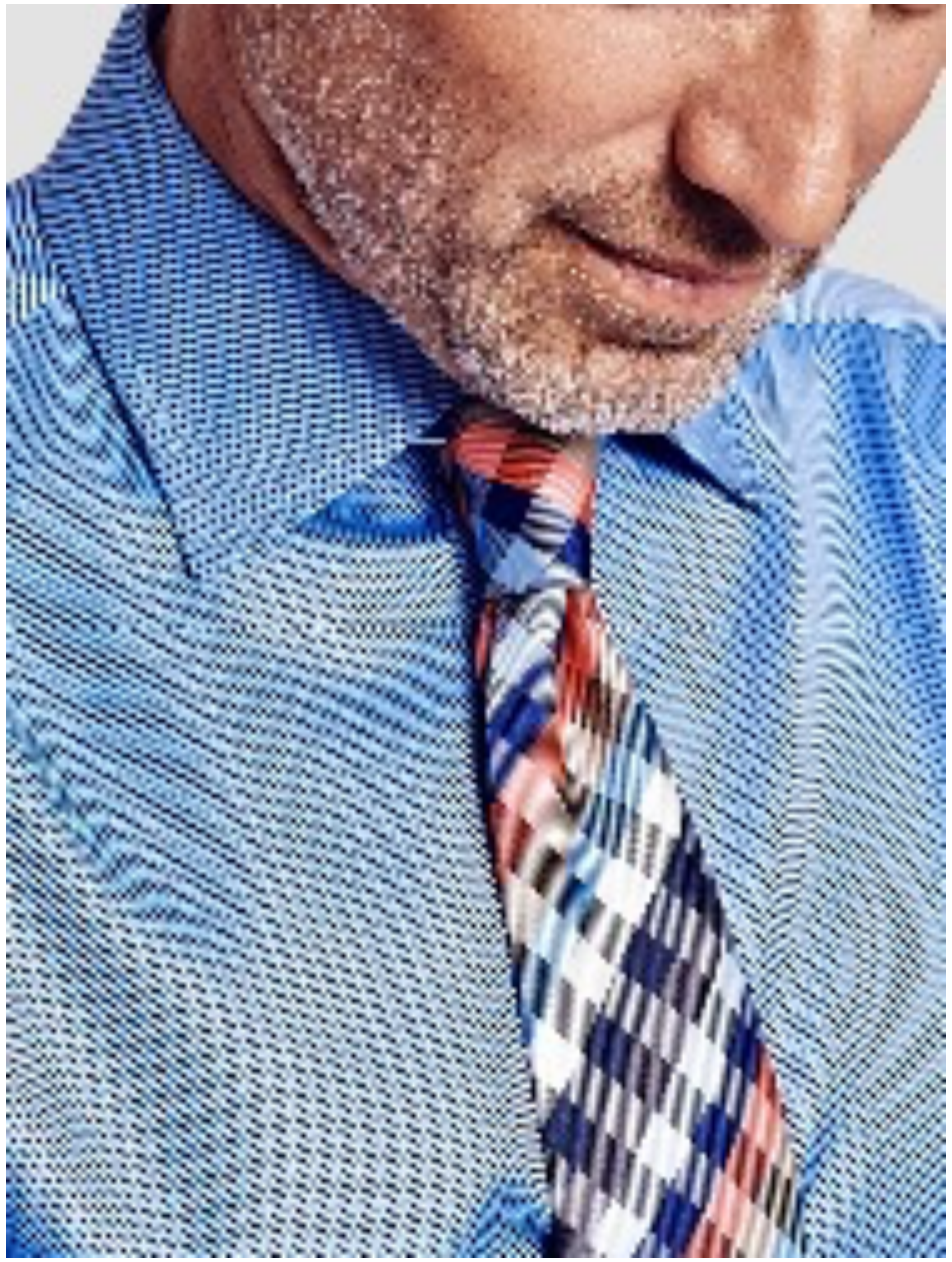




Moiré patterns in imaging



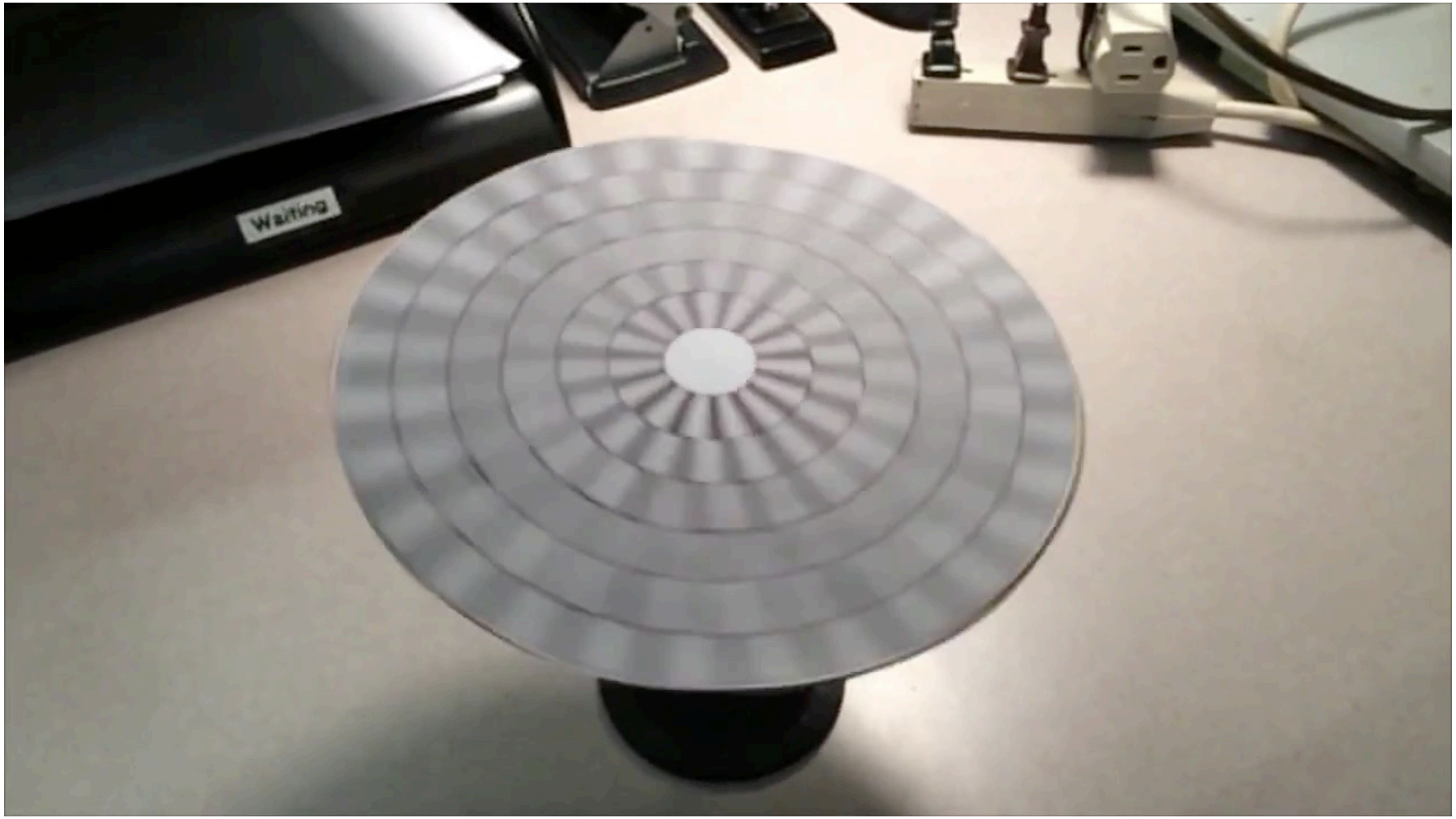
Full resolution image



1/2 resolution image:
skip pixel odd rows and columns

lystit.com

Wagon wheel illusion (false motion)



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

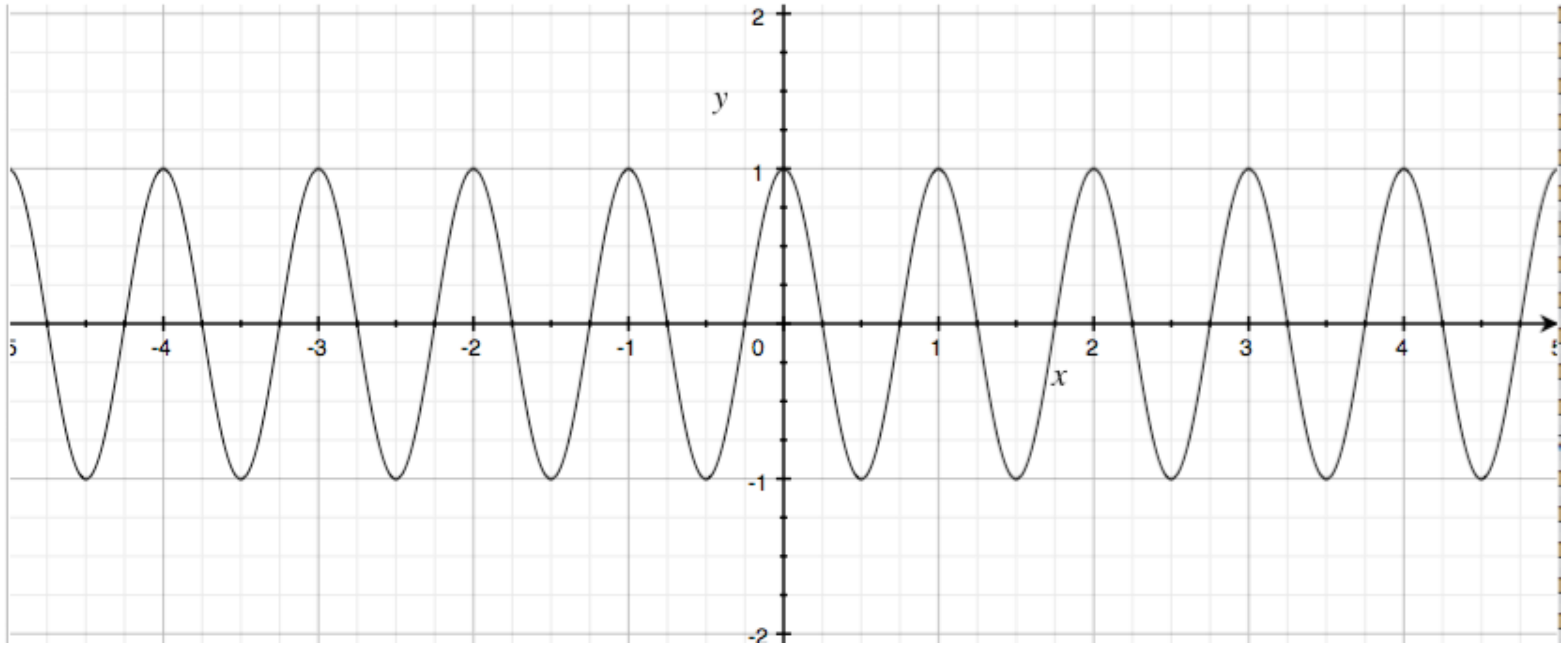
Created by Jesse Mason, https://www.youtube.com/watch?v=QOwzkND_ooU

Sampling artifacts in computer graphics

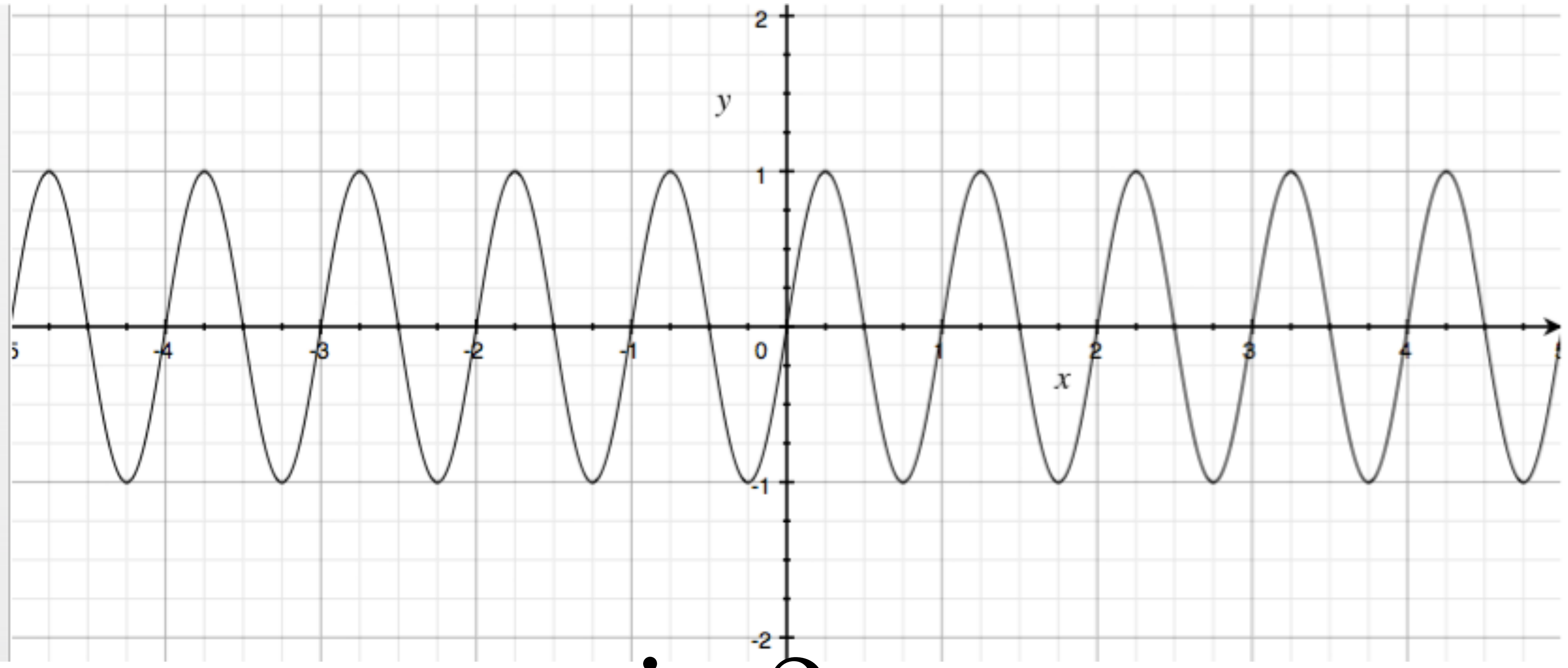
- Artifacts due to sampling - "Aliasing"
  - Jaggies – sampling to sparsely in space
  - Wagon wheel effect – sampling to sparsely in time
  - Moire – undersampling images (and texture maps)
  - [Many more] . . .

- We notice this in fast-changing signals, when we sample the signal too sparsely

Sines and cosines



$$\cos 2\pi x$$

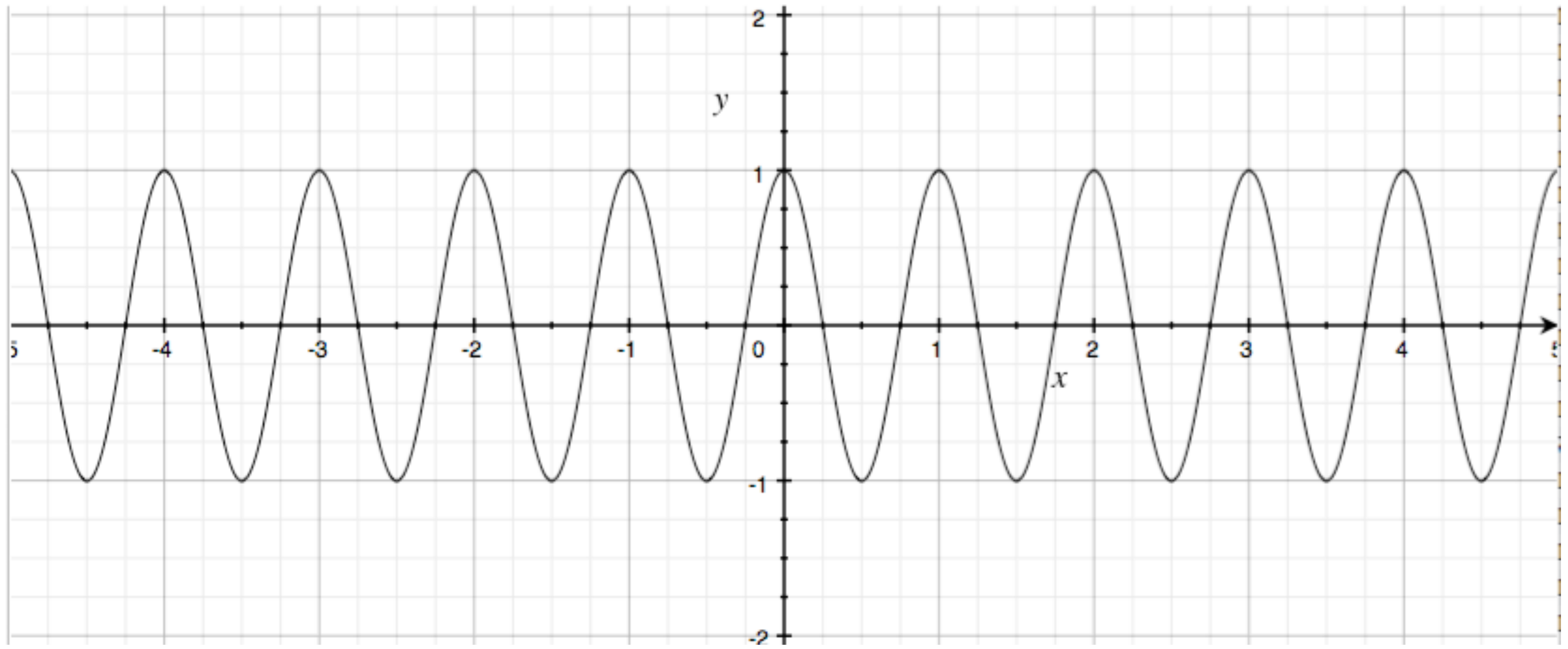


$$\sin 2\pi x$$

Frequencies

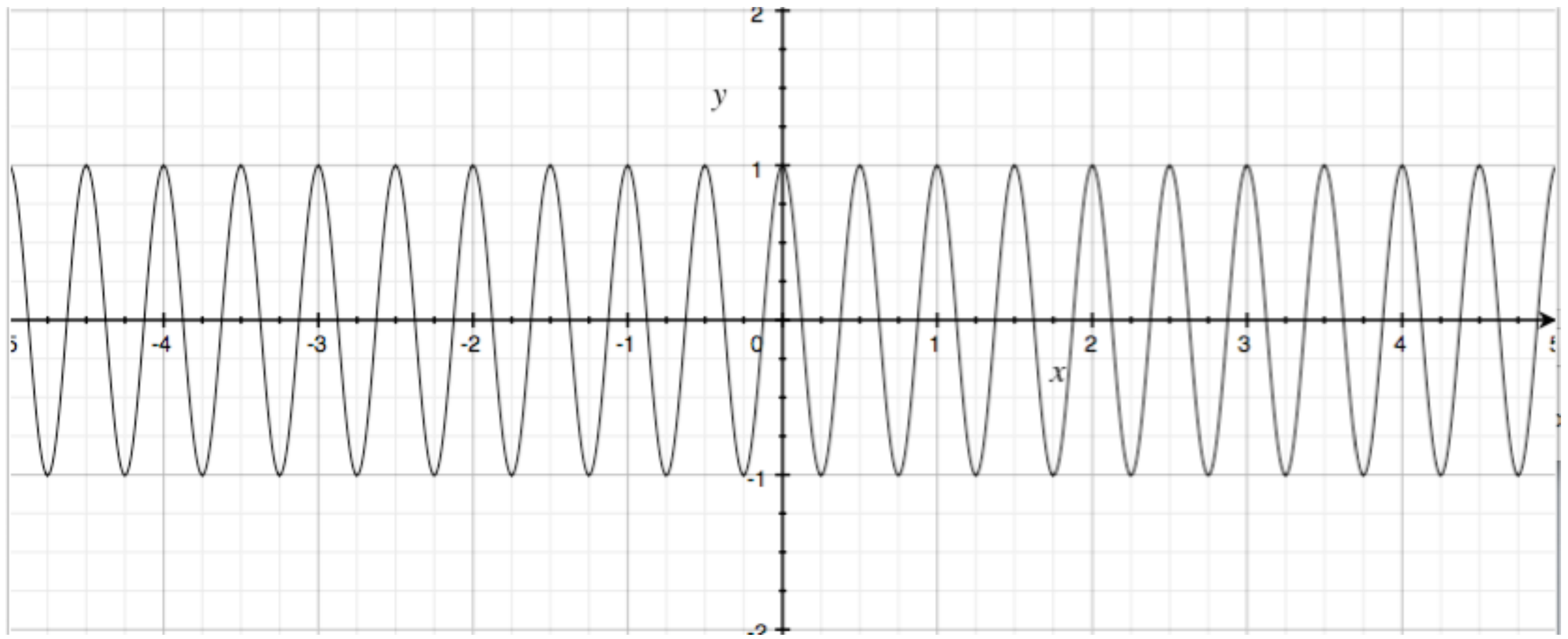
$$\cos 2\pi f x$$

$$f = \frac{1}{T}$$



$$f = 1$$

$$\cos 2\pi x$$



$$f = 2$$

$$\cos 4\pi x$$

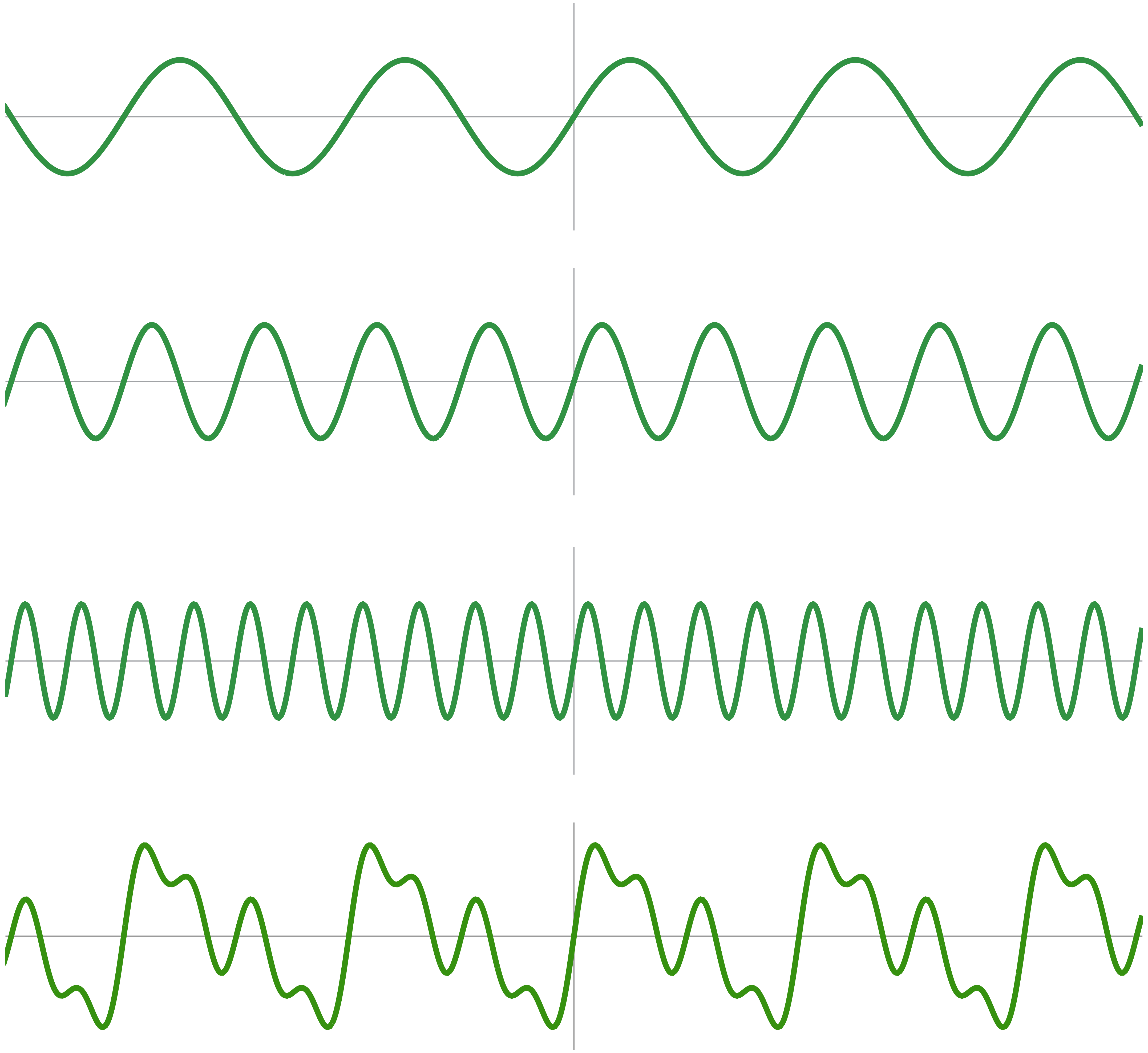
Representing sound wave as a superposition (linear combination) of frequencies

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

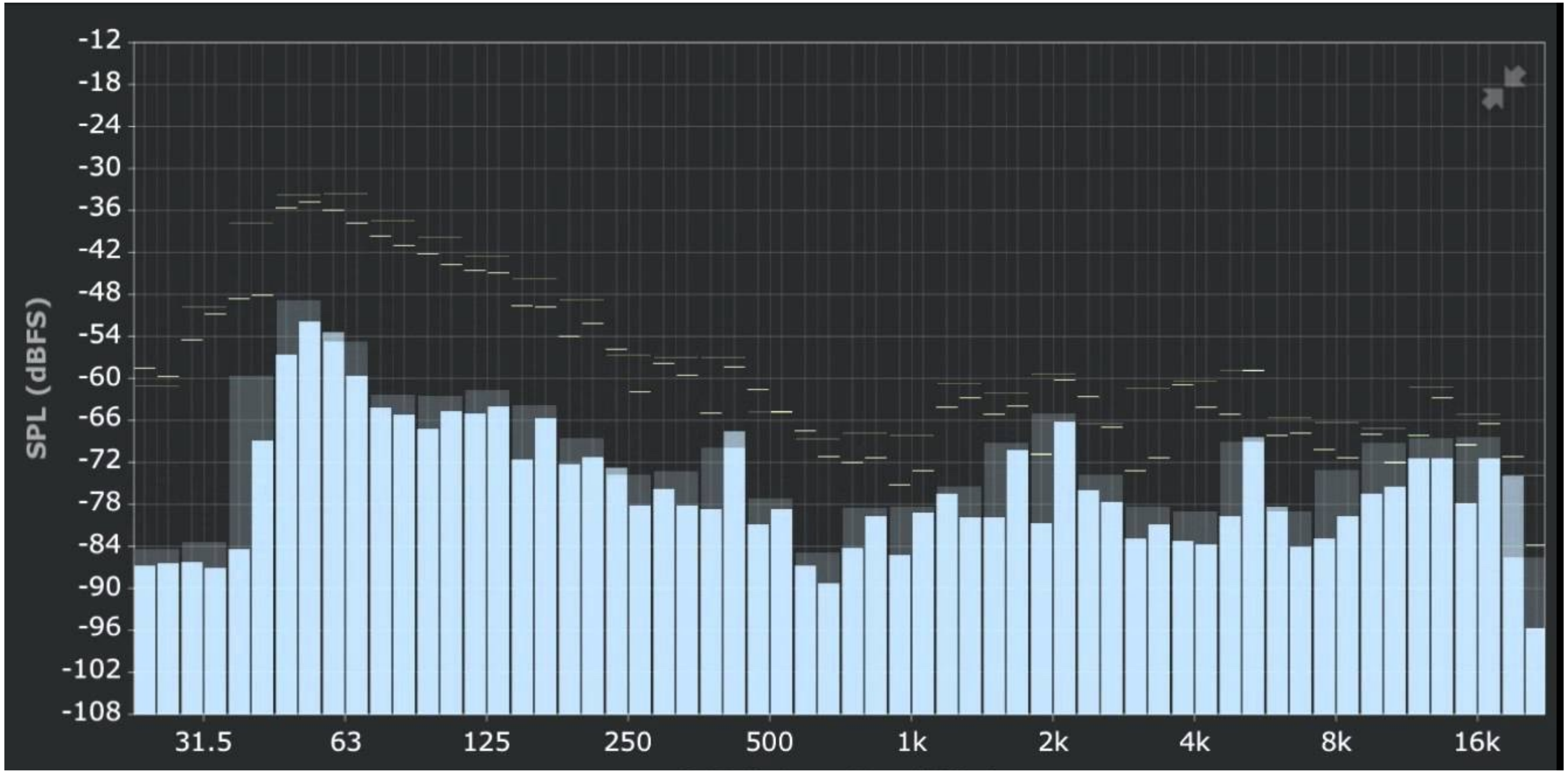
$$f_2(x) = \sin(2\pi x)$$

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

$$f(x) = 1.0\, f_1(x) + 0.75\, f_2(x) + 0.5\, f_4(x)$$



Audio spectrum analyzer: representing sound as a sum of its constituent frequencies



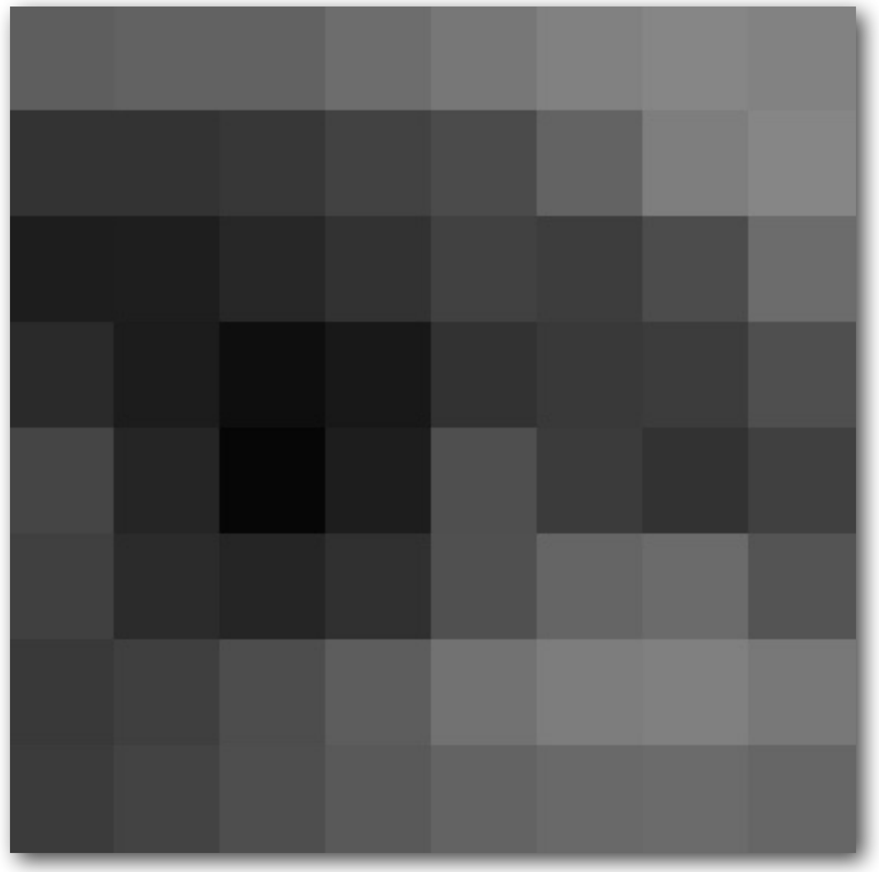
Intensity of
low-frequencies (bass)

Intensity of
high frequencies

Image credit: ONYX Apps

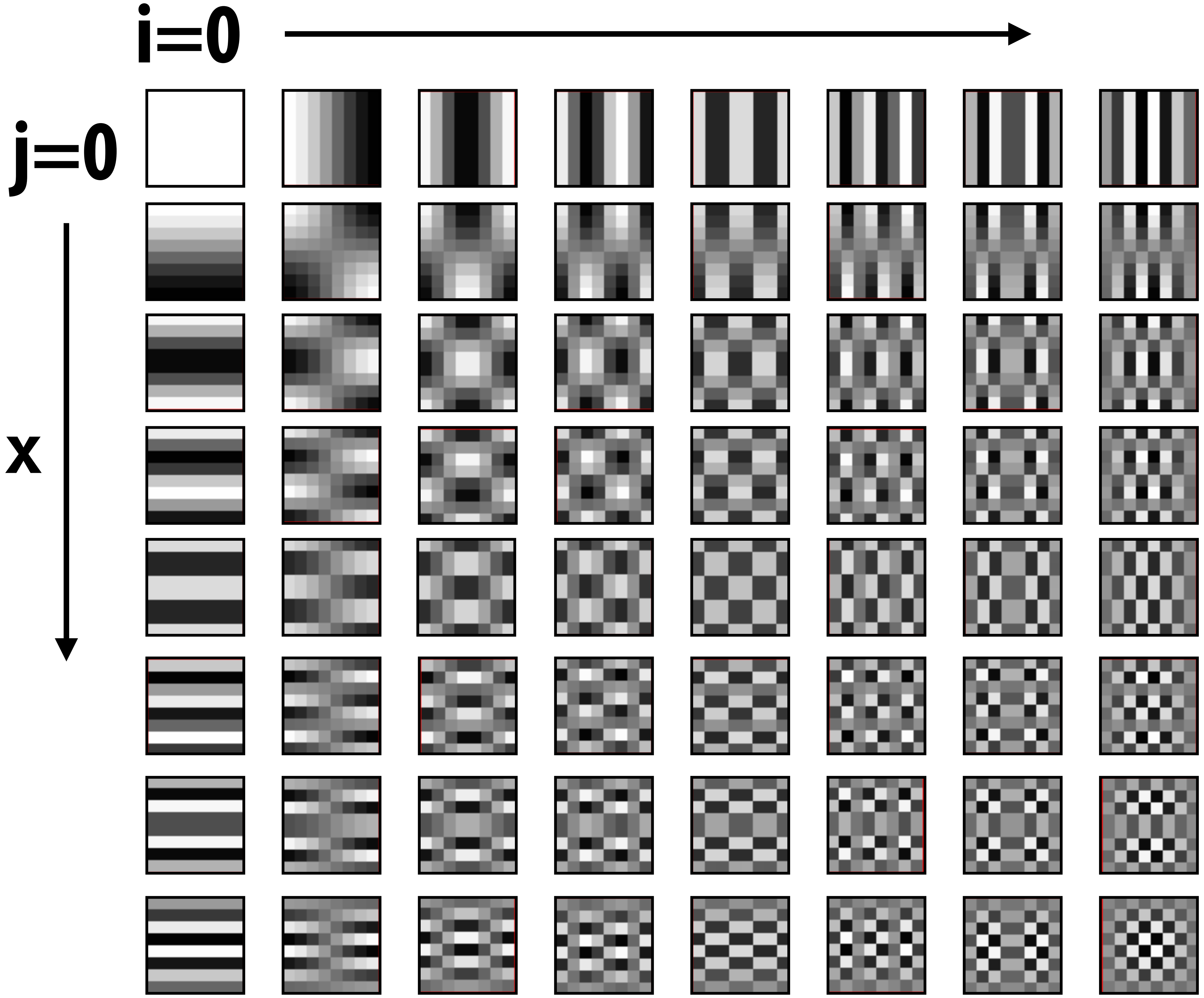
Images as a superposition of cosines

8x8 images

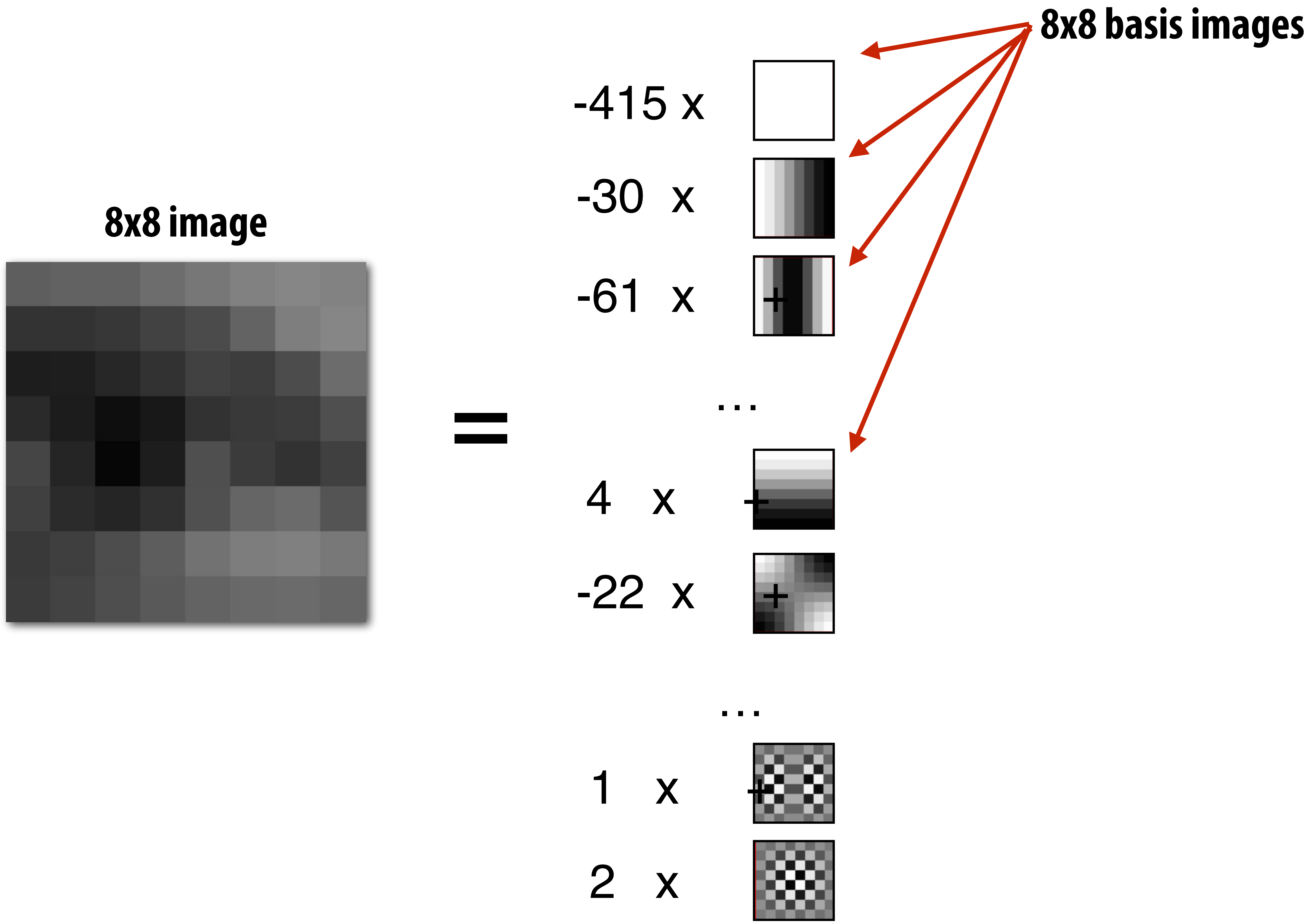


$$= \begin{bmatrix} -415 & -30 & -61 & 27 & 56 & -20 & -2 & 0 \\ 4 & -22 & -61 & 10 & 13 & -7 & -9 & 5 \\ -47 & 7 & 77 & -25 & -29 & 10 & 5 & -6 \\ -49 & 12 & 34 & -15 & -10 & 6 & 2 & 2 \\ 12 & -7 & -13 & -4 & -2 & 2 & -3 & 3 \\ -8 & 3 & 2 & -6 & -2 & 1 & 4 & 2 \\ -1 & 0 & 0 & -2 & -1 & -3 & 4 & -1 \\ 0 & 0 & -1 & -4 & -1 & 0 & 1 & 2 \end{bmatrix} \times$$

$$\cos\left[\pi \frac{i}{N}\left(x + \frac{1}{2}\right)\right] \times \underset{j}{\cos}\left[\pi \frac{j}{N}\left(y + \frac{1}{2}\right)\right]$$



Images as a superposition of cosines

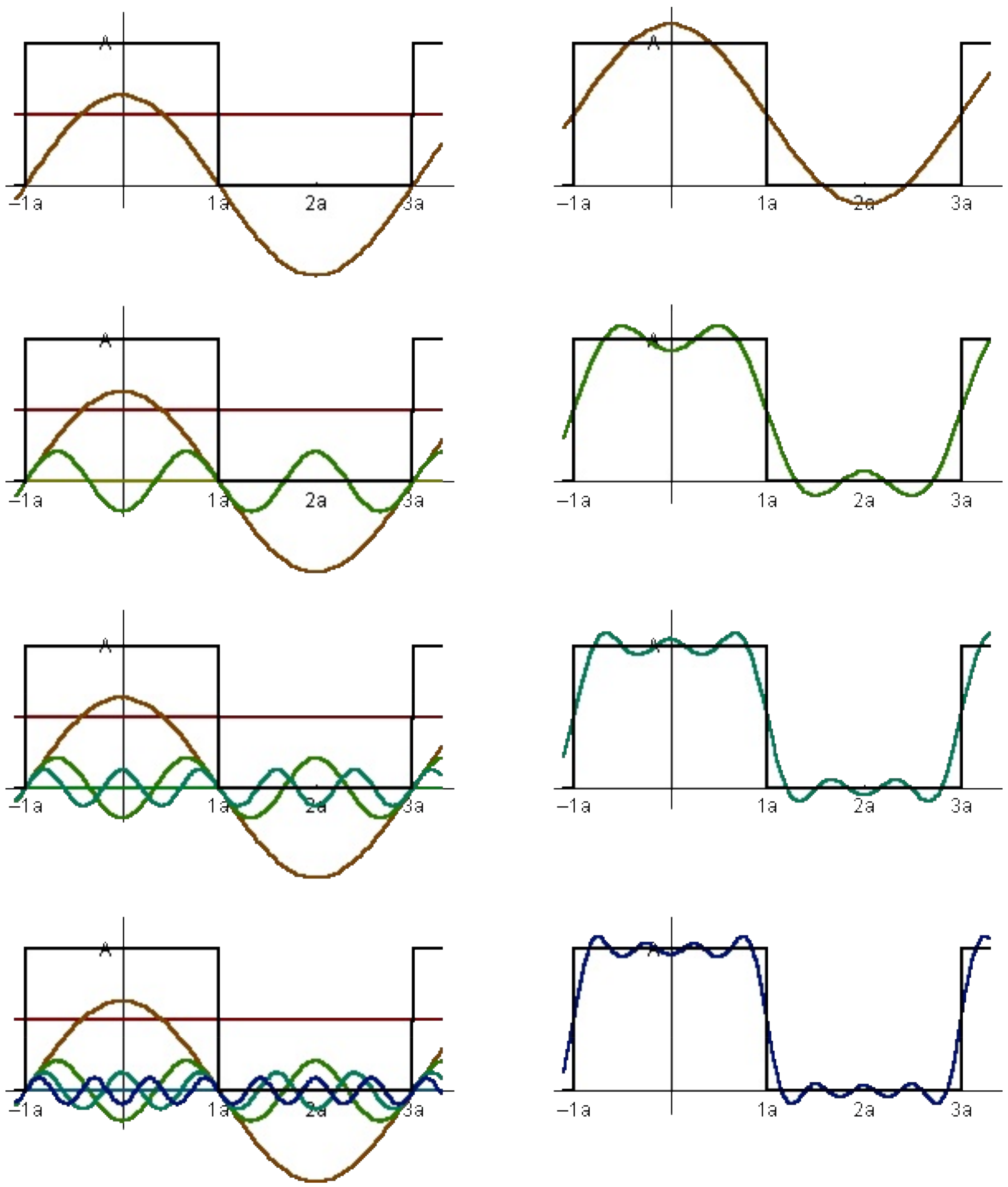


How to compute frequency-domain representation of a signal?

Fourier transform

Represent any function as a weighted sum of sines and cosines

Joseph Fourier 1768 - 1830



Fourier transform

Convert representation of signal from primal domain (spatial/temporal) to frequency domain by projecting signal into its component frequencies

$$F(\omega) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x \omega} dx$$

$$= \int_{-\infty}^{\infty} f(x)(\cos(2\pi\omega x) - i\sin(2\pi\omega x)) dx$$

Recall:

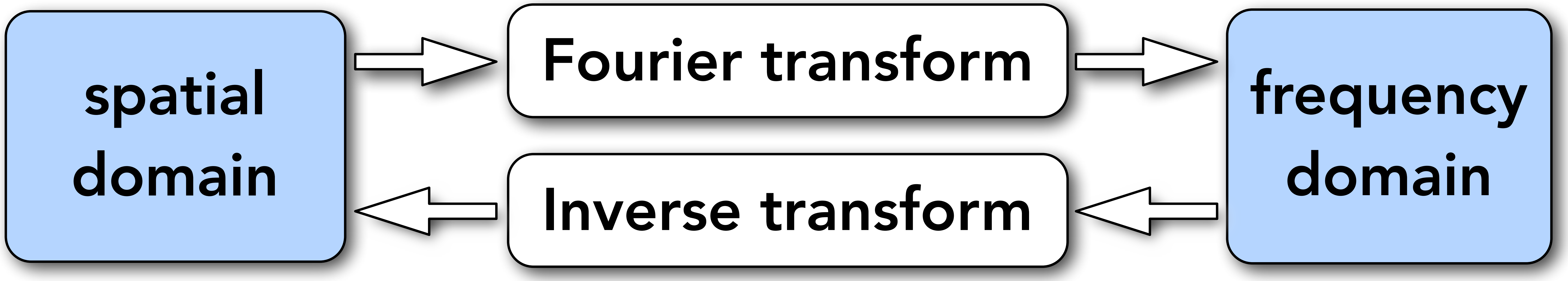
$$e^{ix} = \cos x + i \sin x$$

2D form:

$$F(u,v) = \int\int f(x,y) e^{-2\pi i (ux+vy)} dx dy$$

The Fourier transform decomposes a signal into its constituent frequencies

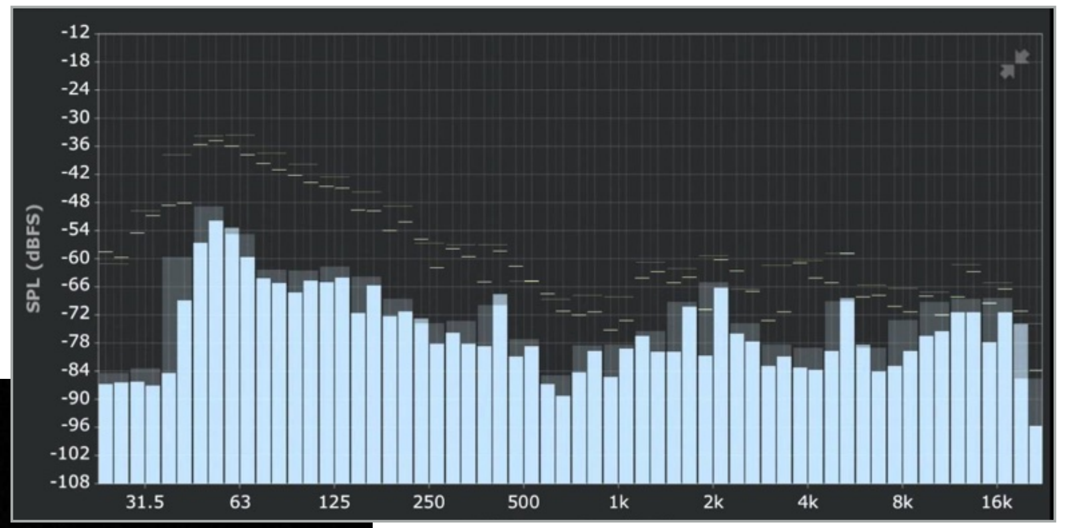
$$f(x) \qquad F(\omega) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i \omega x} dx \qquad F(\omega)$$



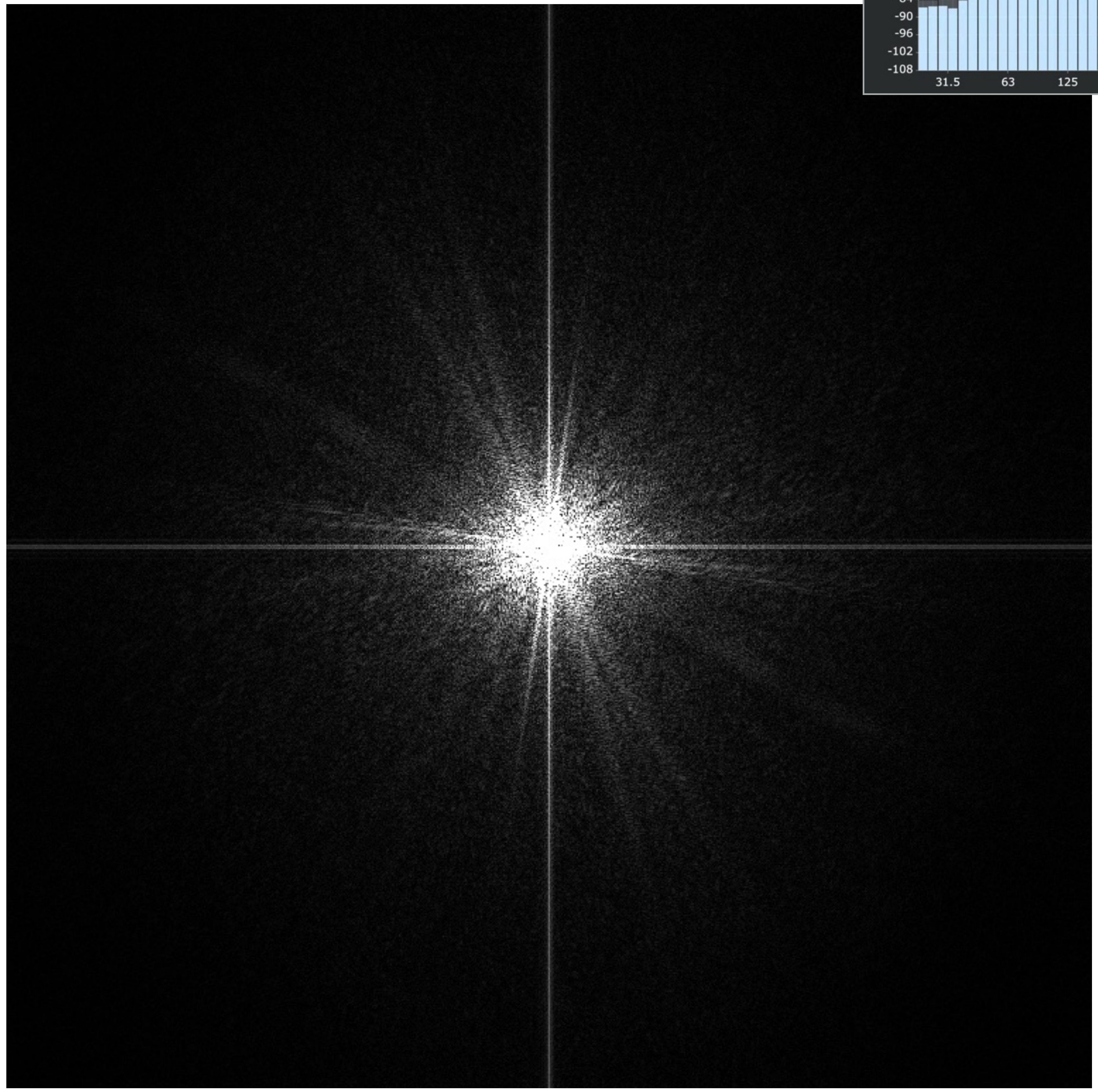
$$f(x) = \int_{-\infty}^{\infty} F(\omega) e^{2\pi i \omega x} d\omega$$

Visualizing the frequency content of images

The visualization below is the 2D frequency domain equivalent of the 1D audio spectrum I showed you earlier *

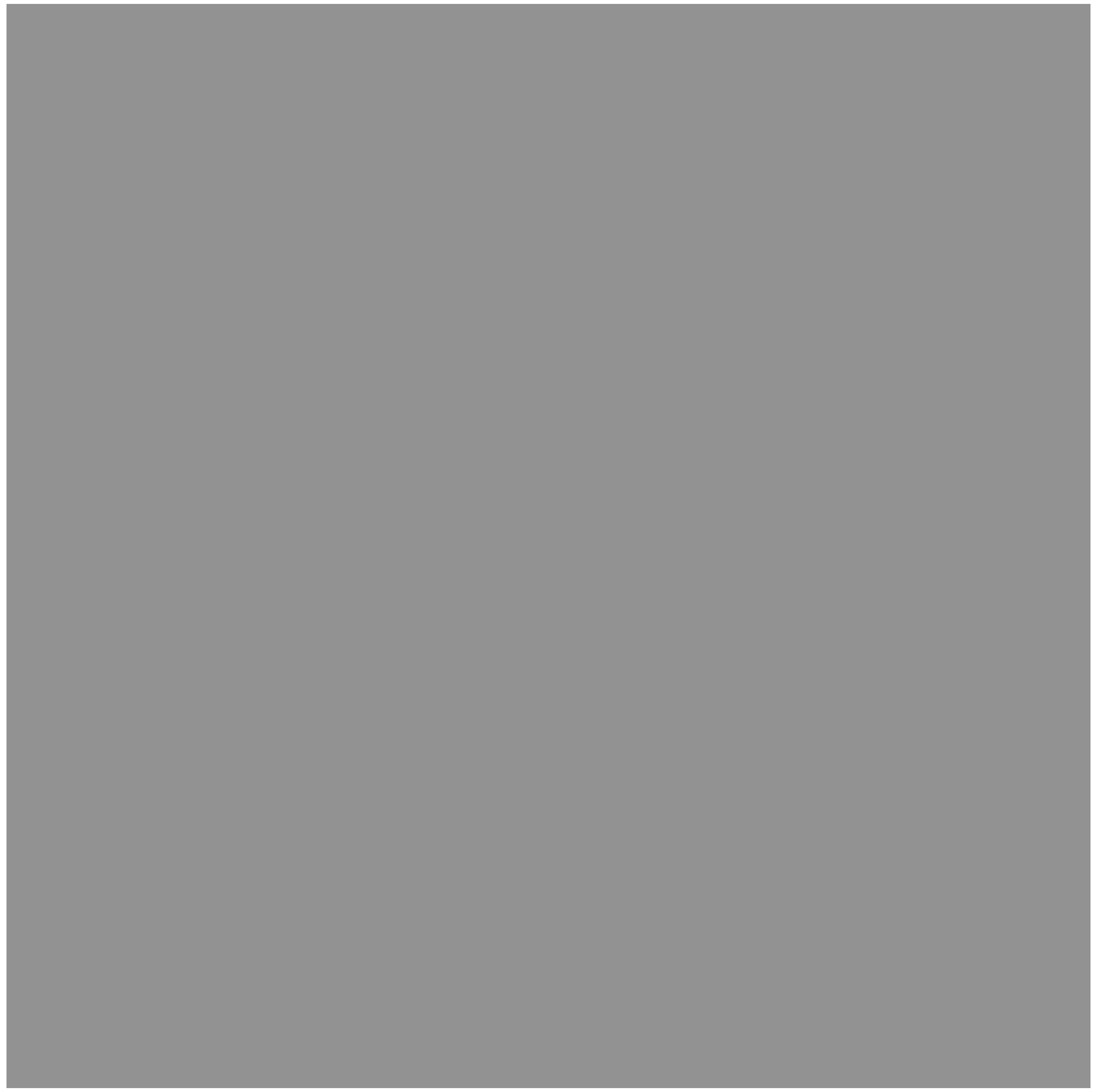


Spatial domain result

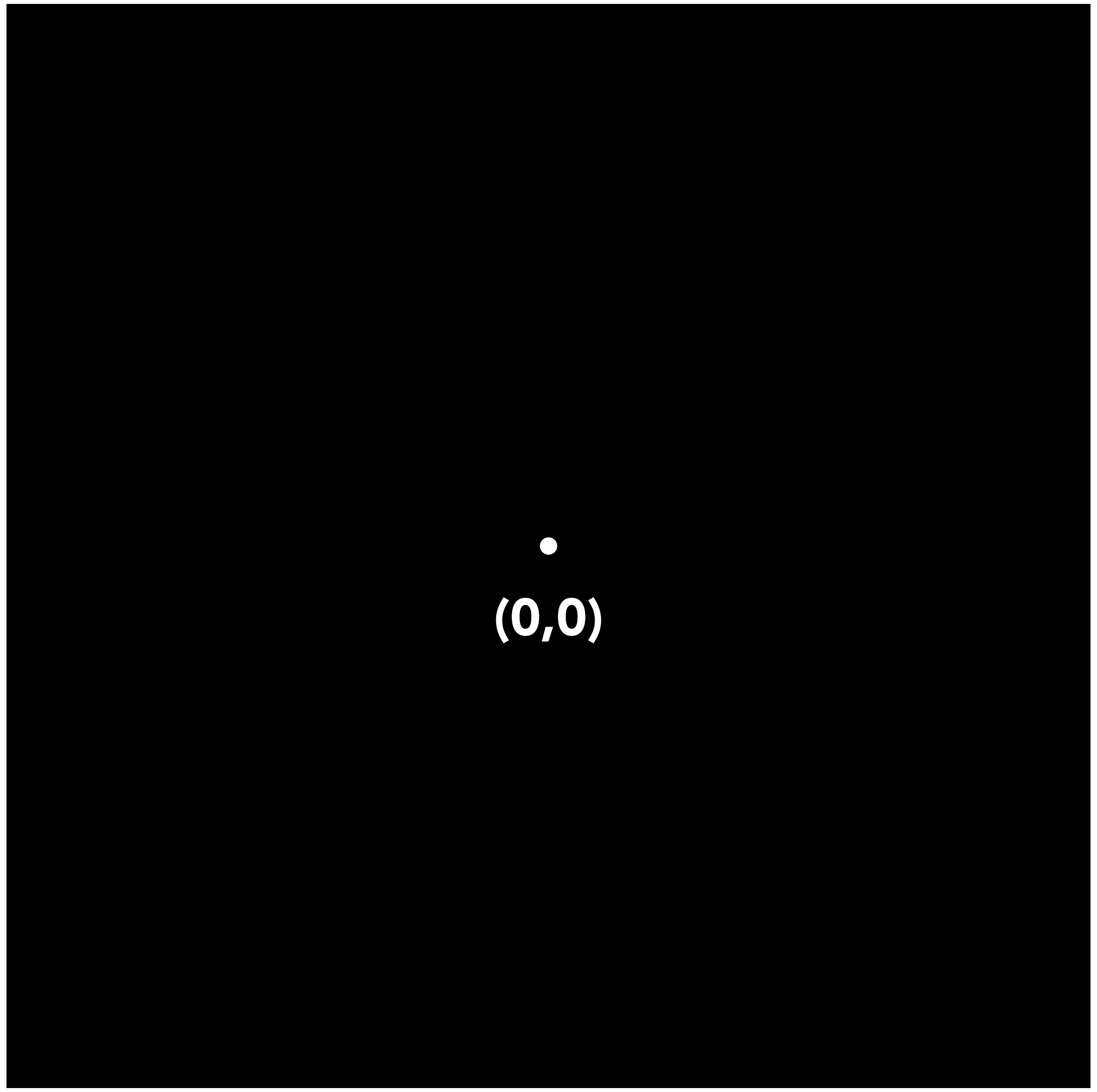


Spectrum

Constant signal (in primal domain)

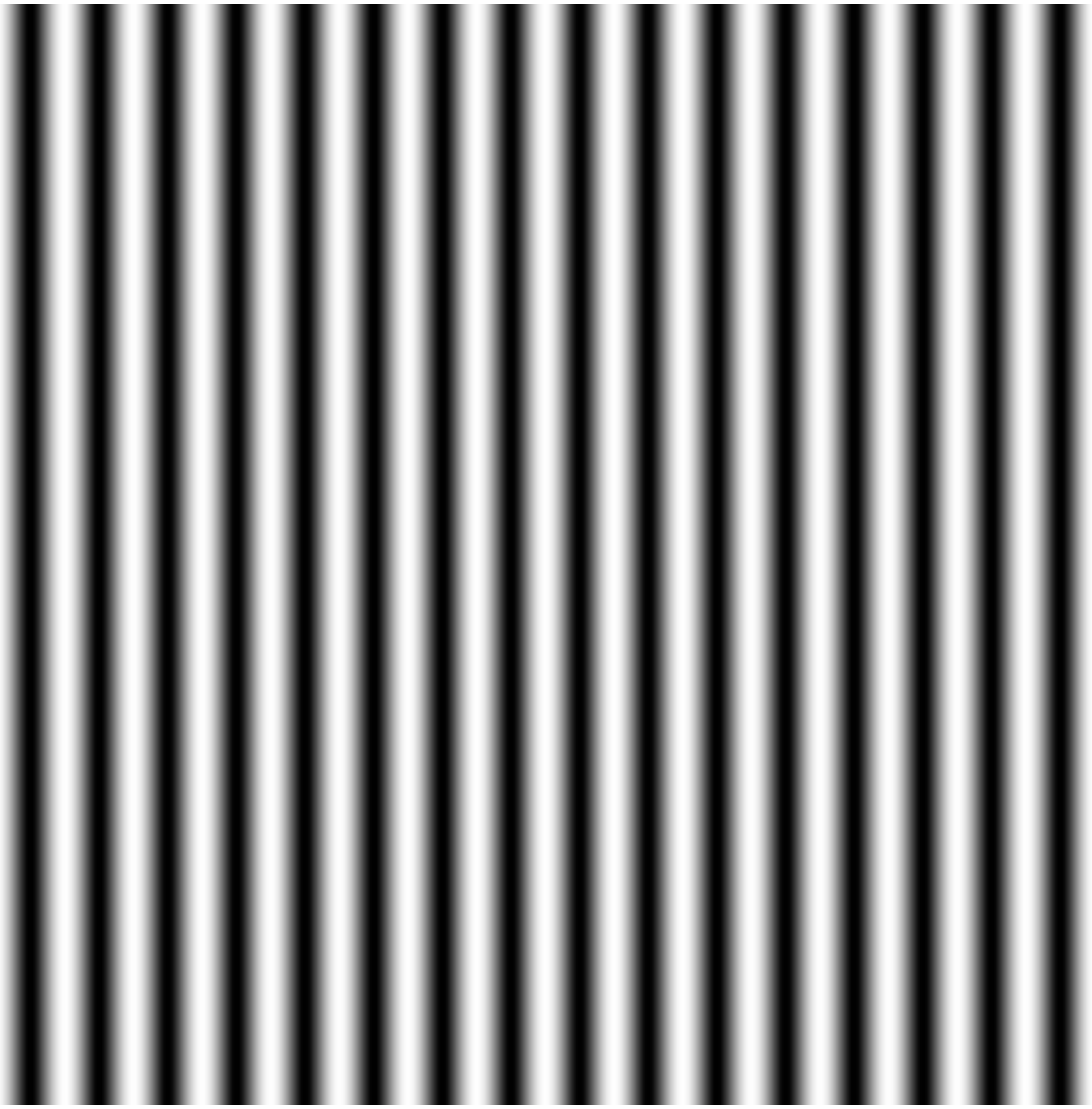


Spatial domain

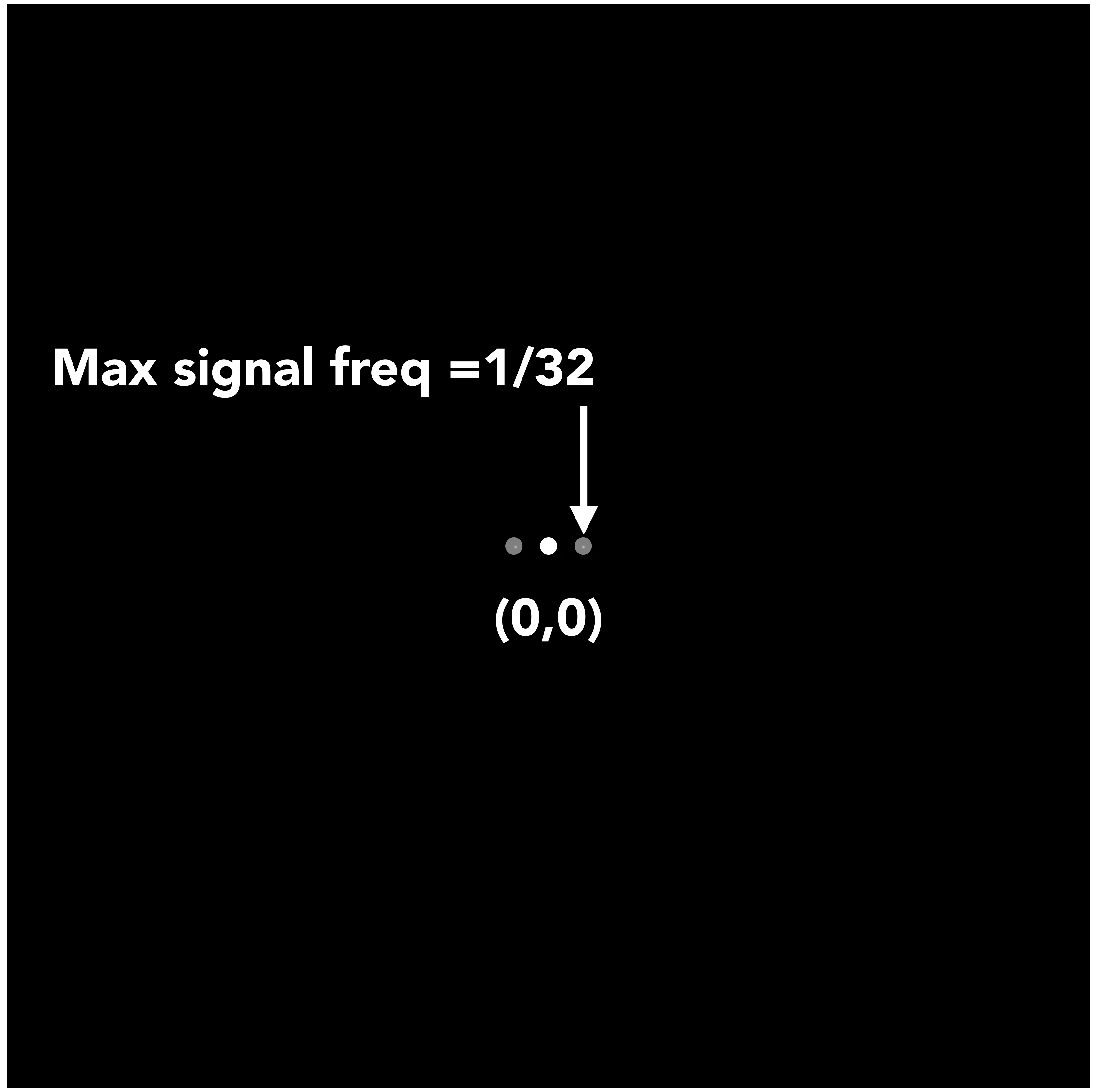


Frequency domain

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

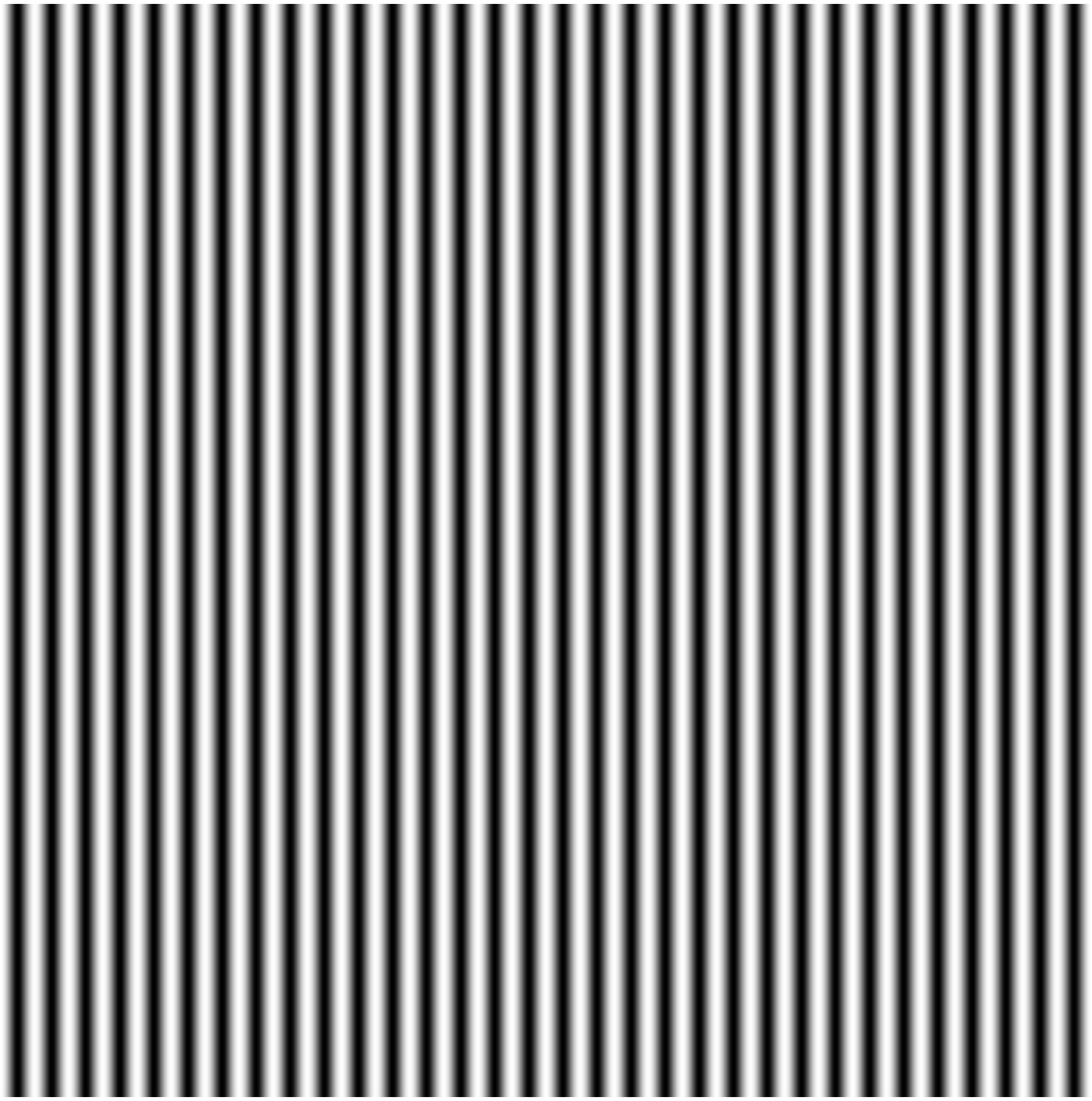


Spatial domain

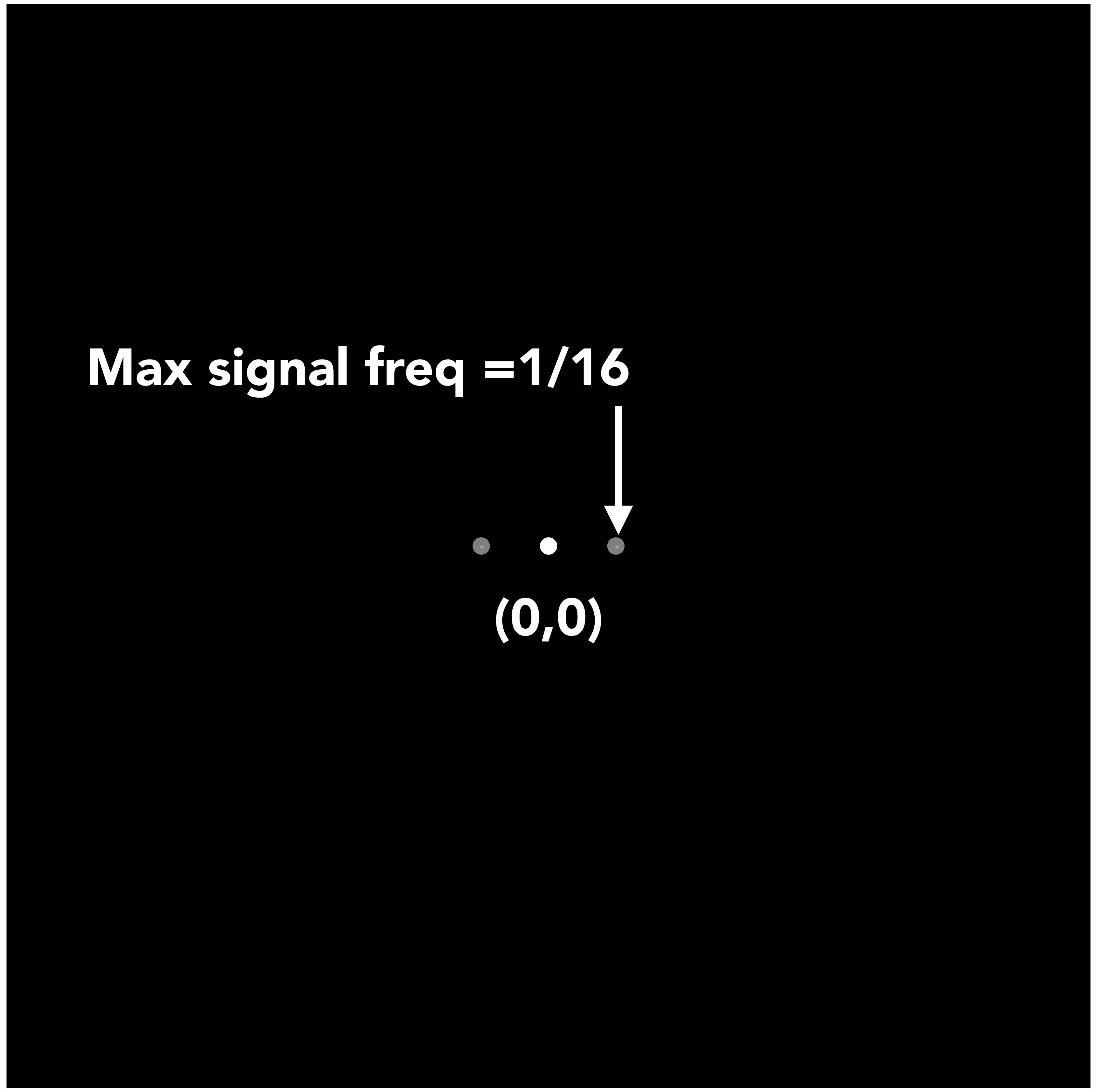


Frequency domain

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

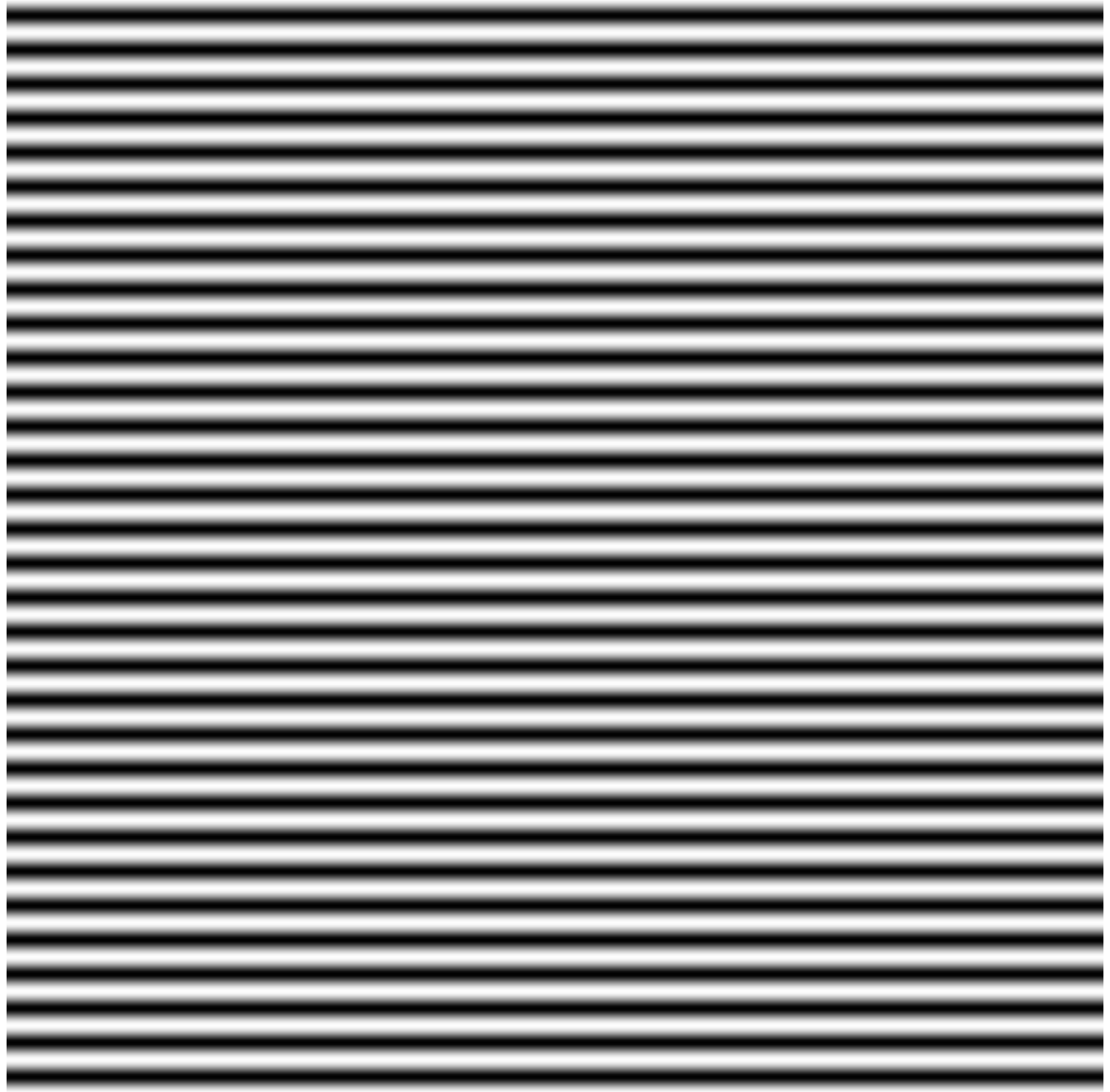


Spatial domain

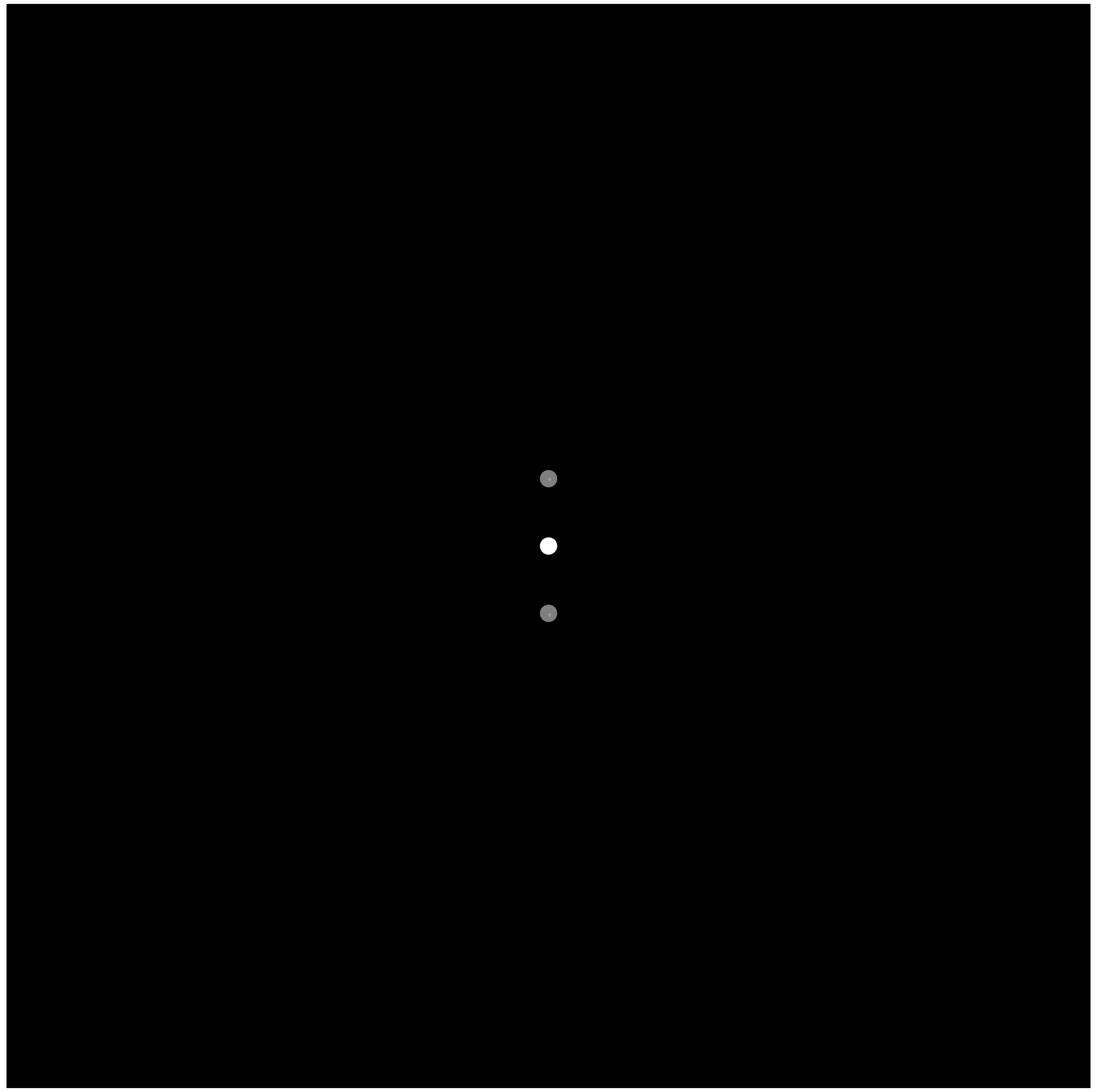


Frequency domain

$$\sin(2\pi/16)y$$

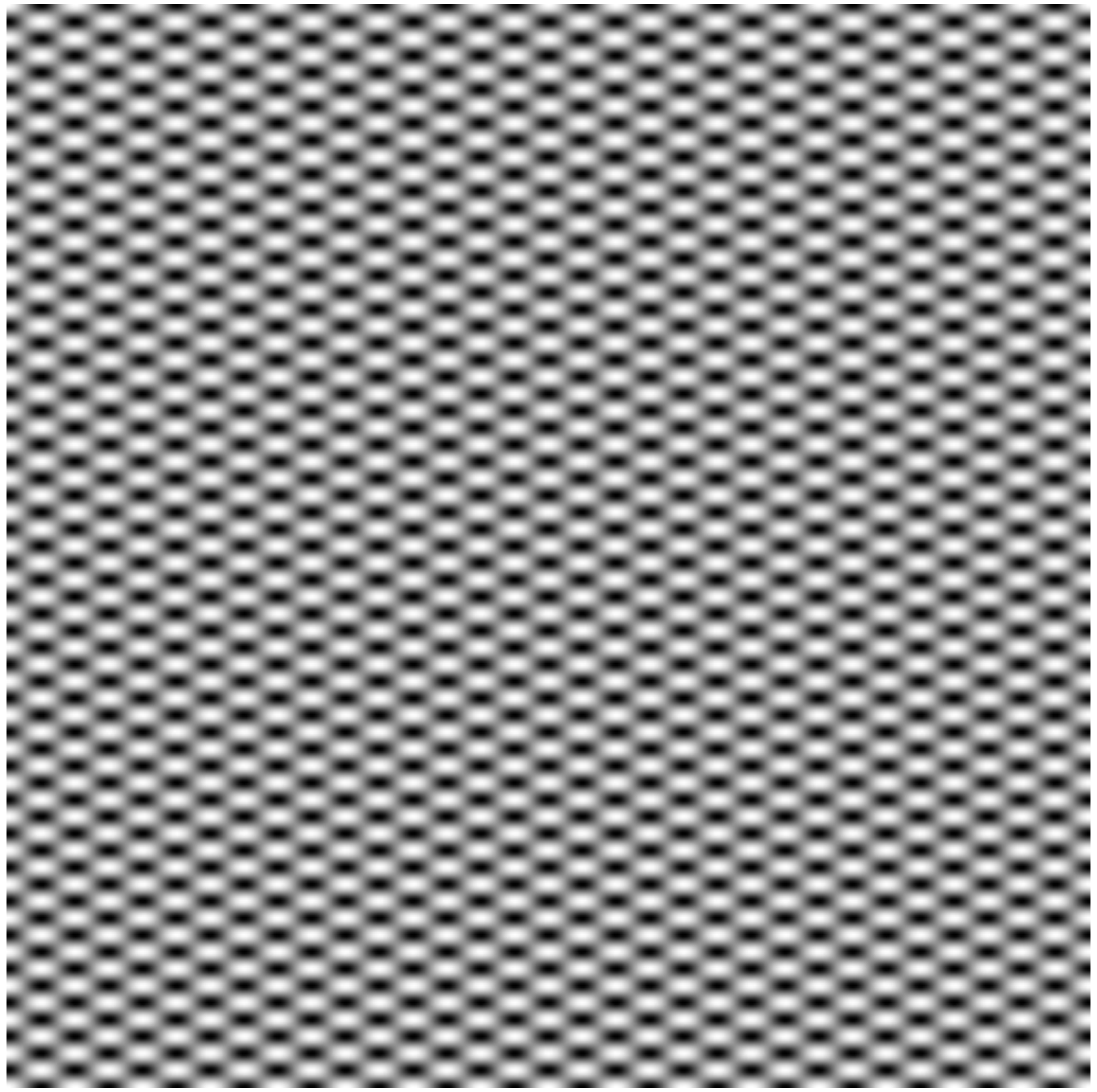


Spatial domain

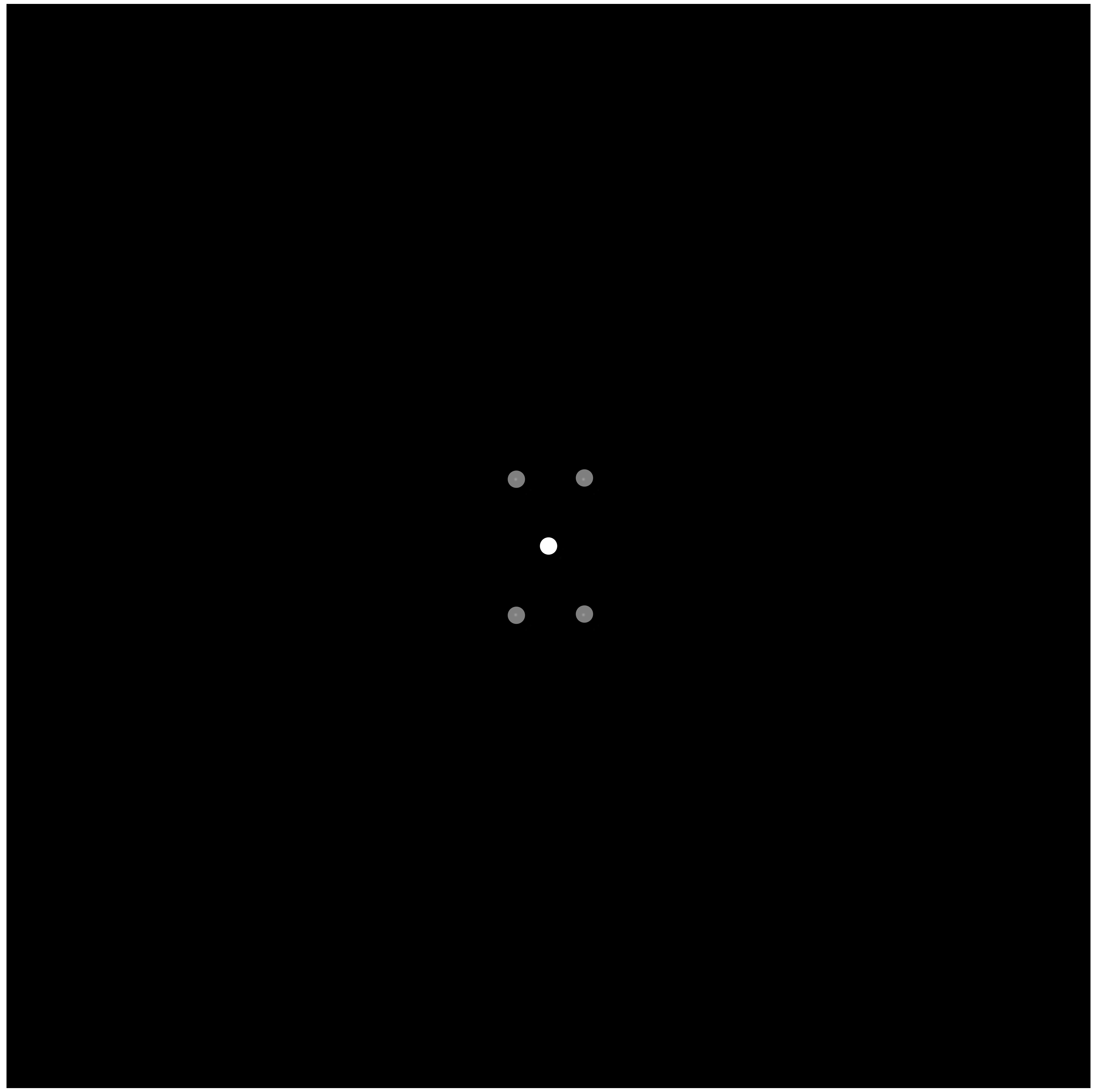


Frequency domain

$$\sin(2\pi/32)x \times \sin(2\pi/16)y$$

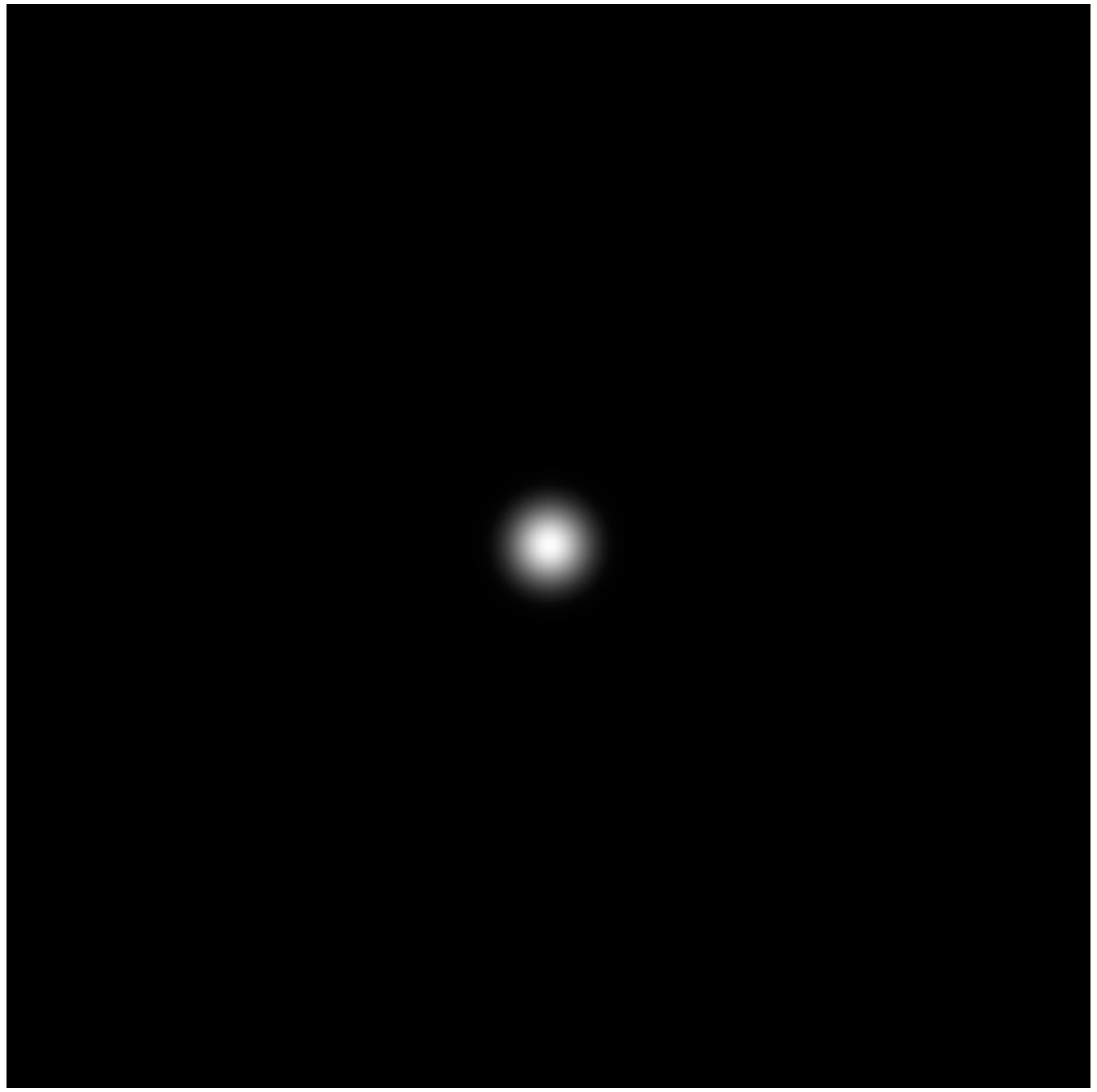


Spatial domain

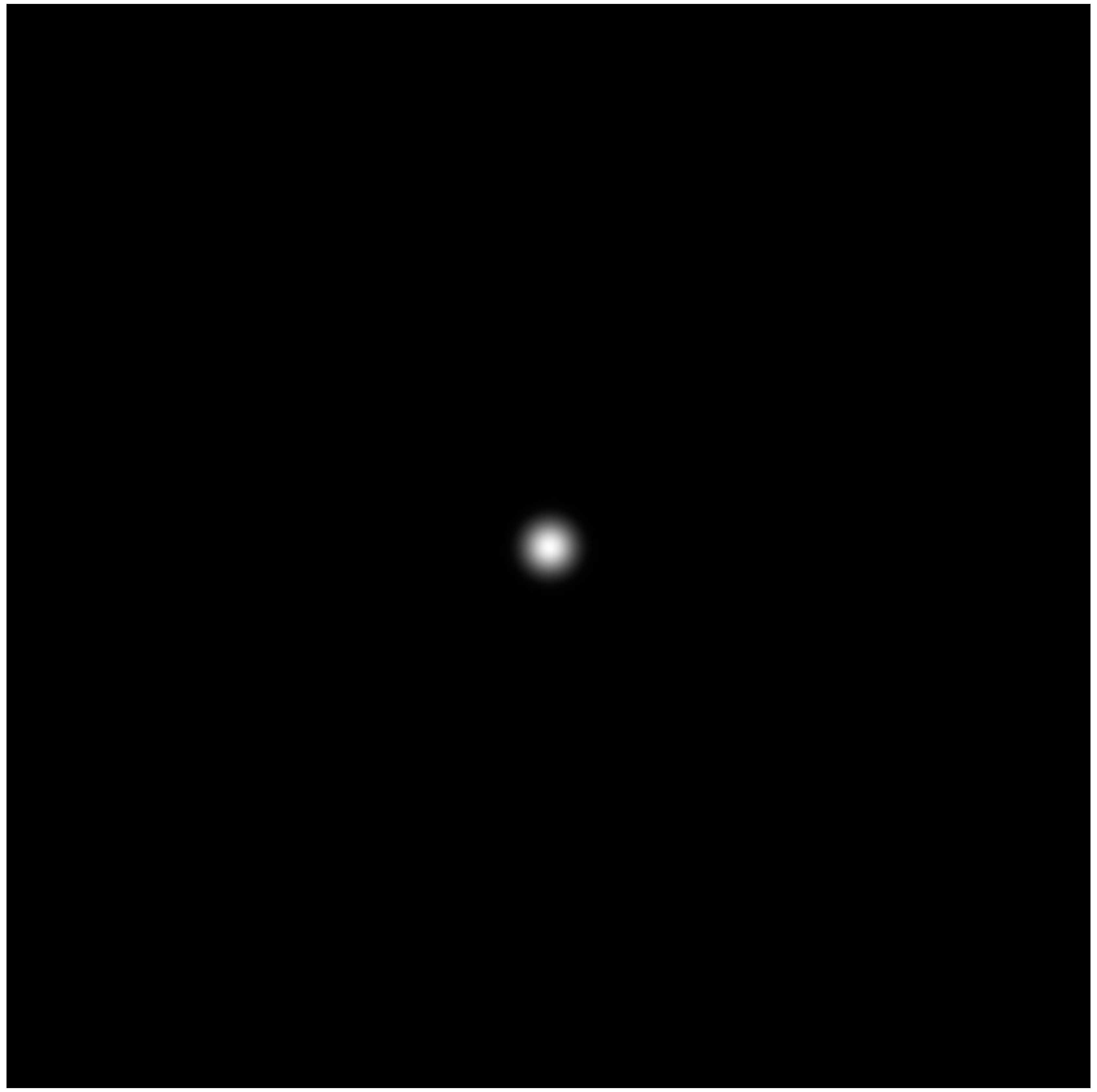


Frequency domain

$$\exp(-r^2/16^2)$$

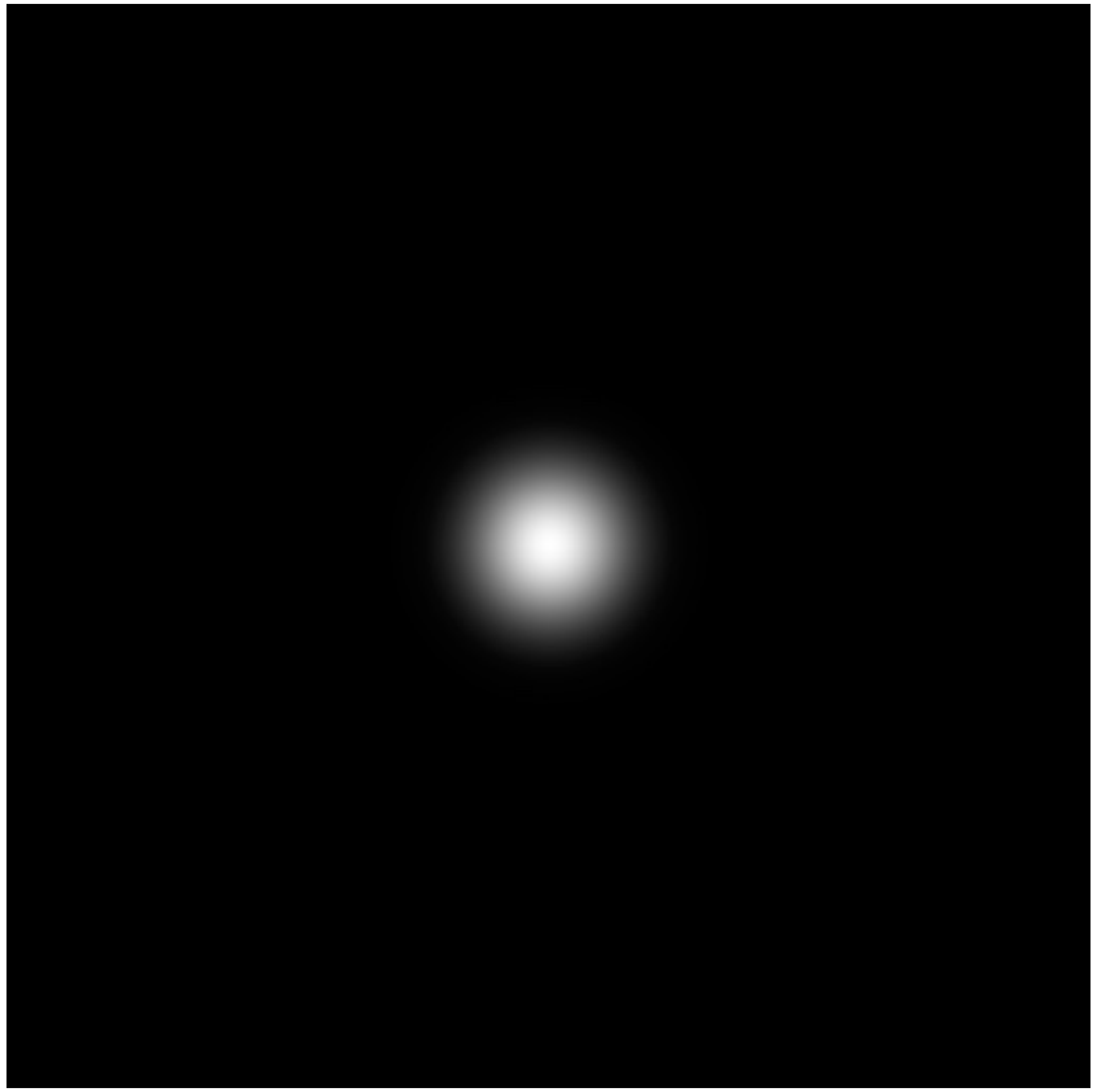


Spatial domain

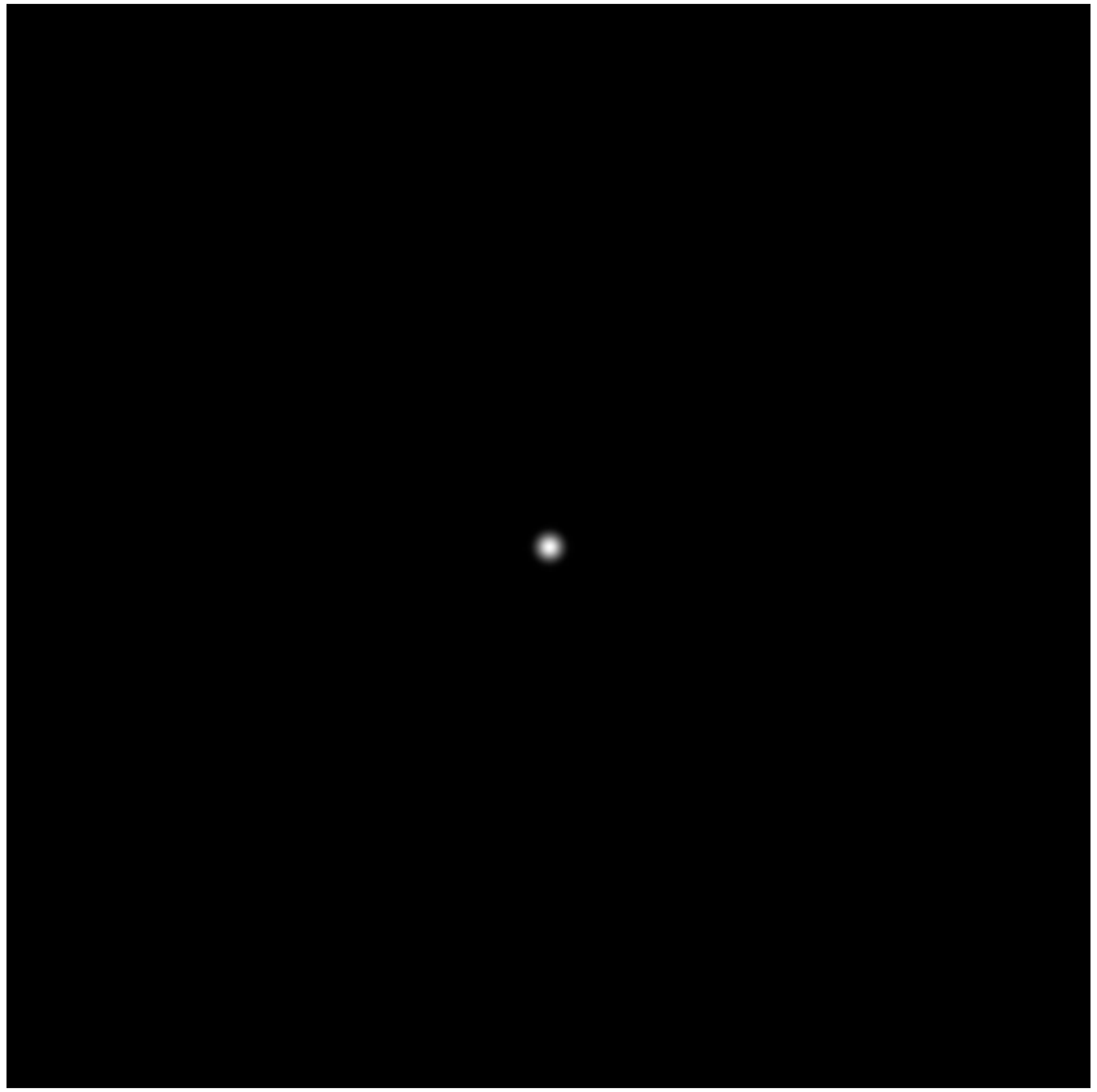


Frequency domain

$$\exp(-r^2/32^2)$$



Spatial domain



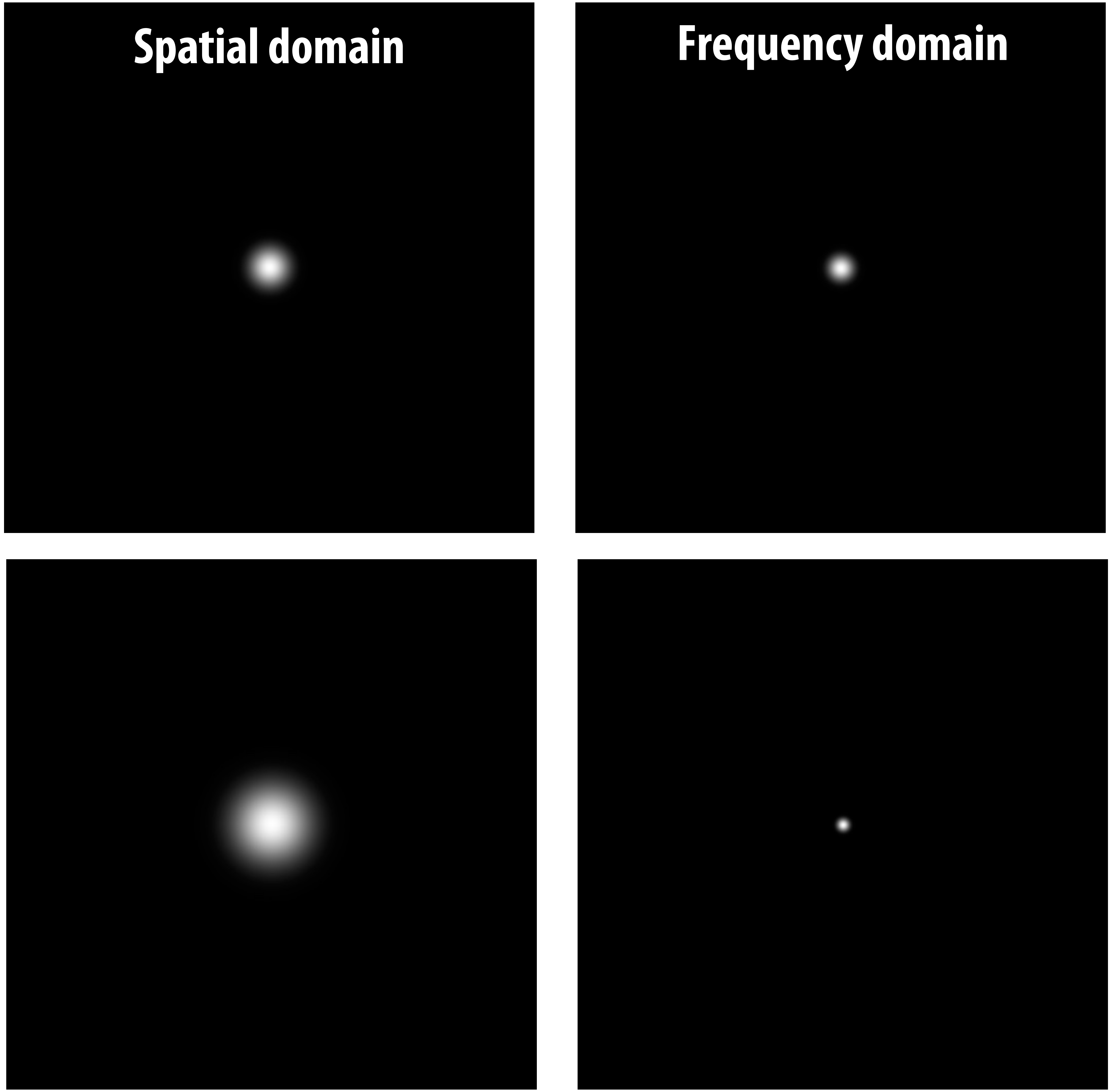
Frequency domain

Question:

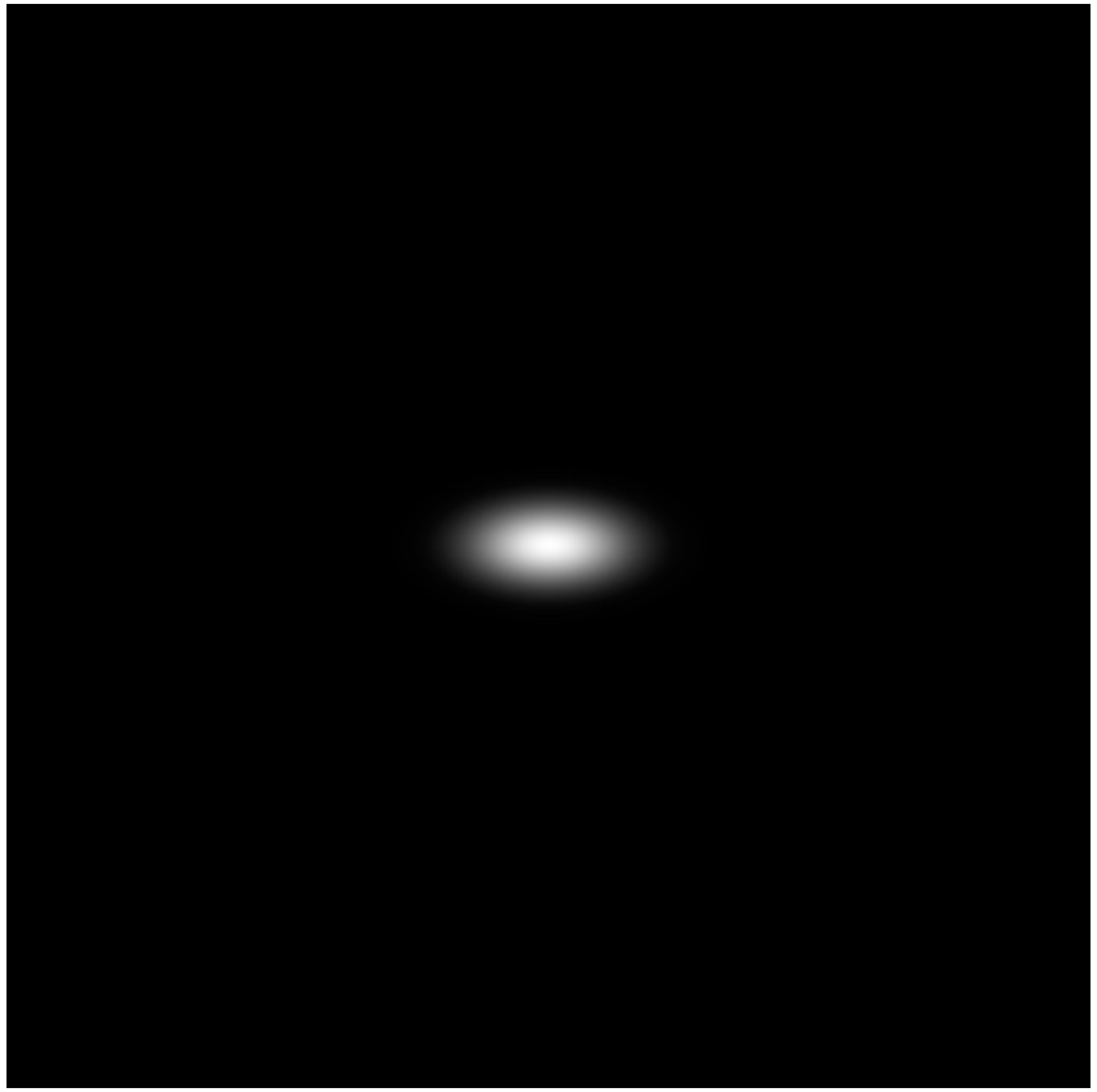
$$\exp(-r^2/16^2)$$

Why does a "smoother" exponential function in the spatial domain look "more compact" in the frequency domain?

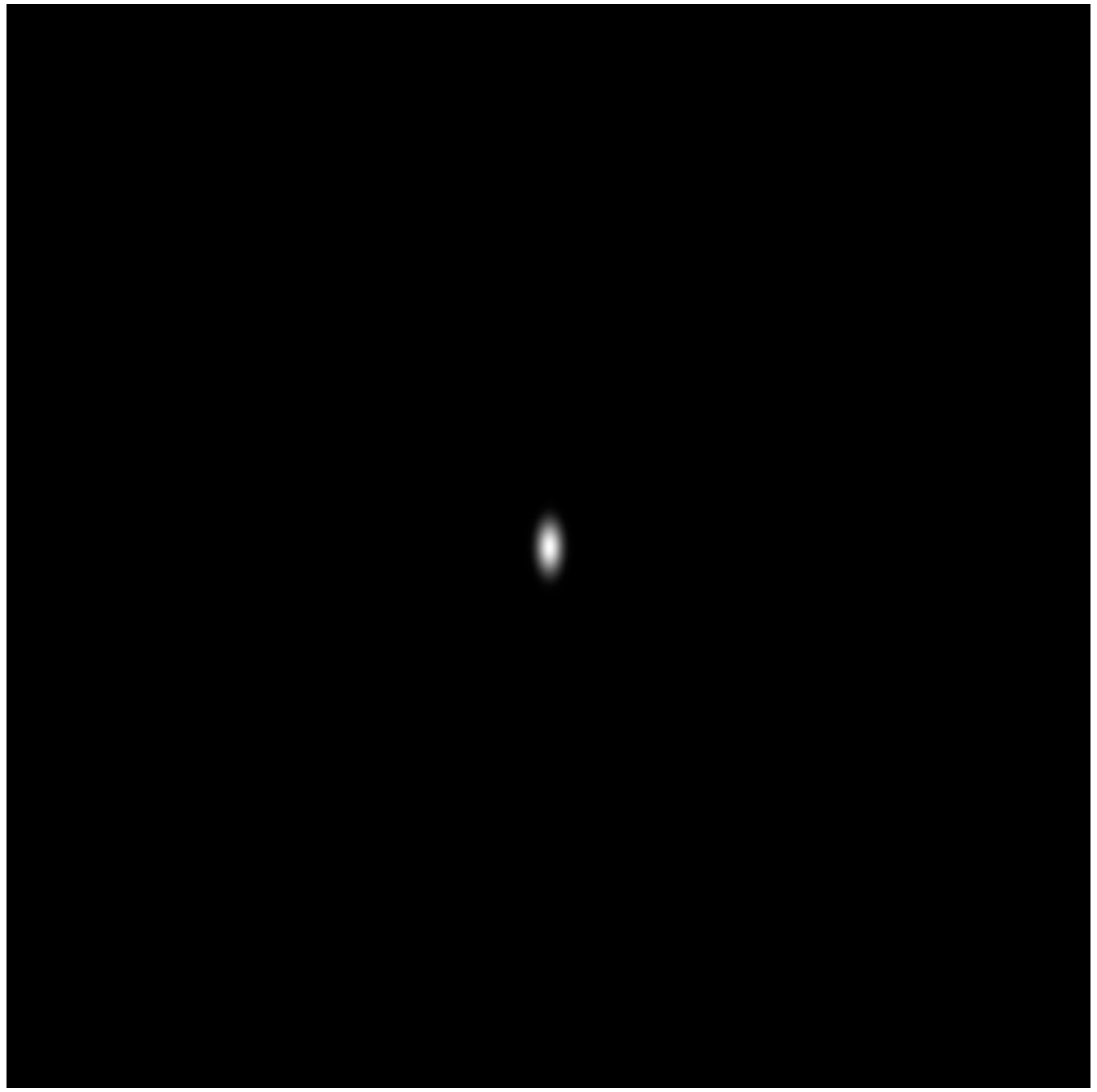
$$\exp(-r^2/32^2)$$



$$\exp(-x^2/32^2) \times \exp(-y^2/16^2)$$



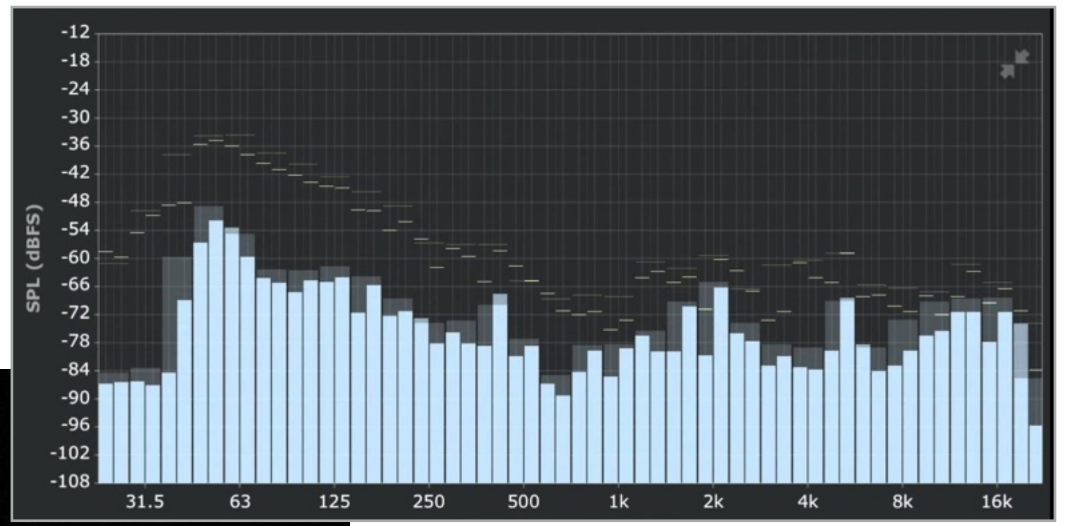
Spatial domain



Frequency domain

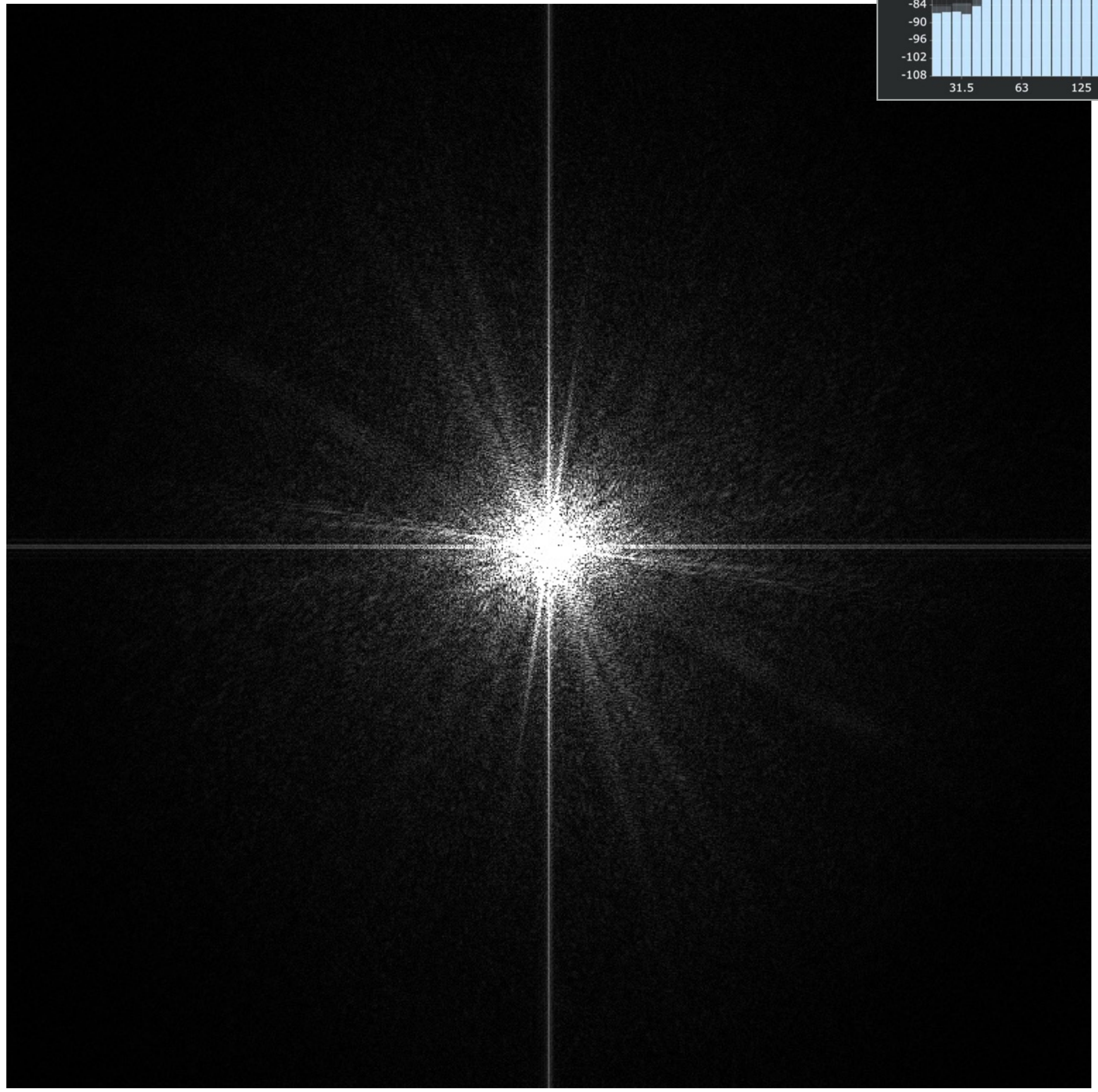
Image filtering
(in the frequency domain)

Manipulating the frequency content of images



The visualization below is the 2D frequency domain equivalent of the 1D audio spectrum I showed you earlier *

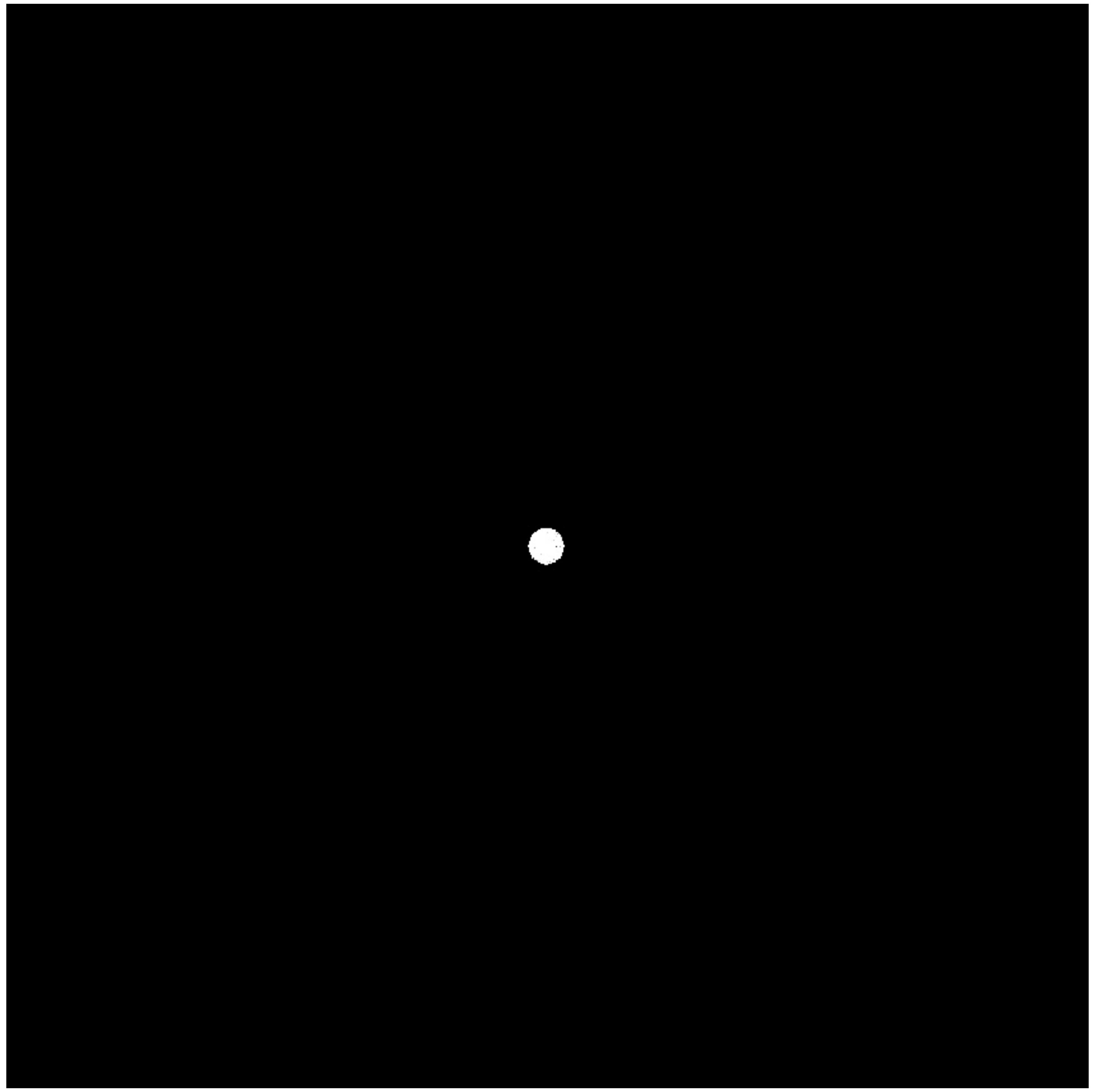
Spatial domain



Frequency domain

Low frequencies only (smooth gradients)

Spatial domain



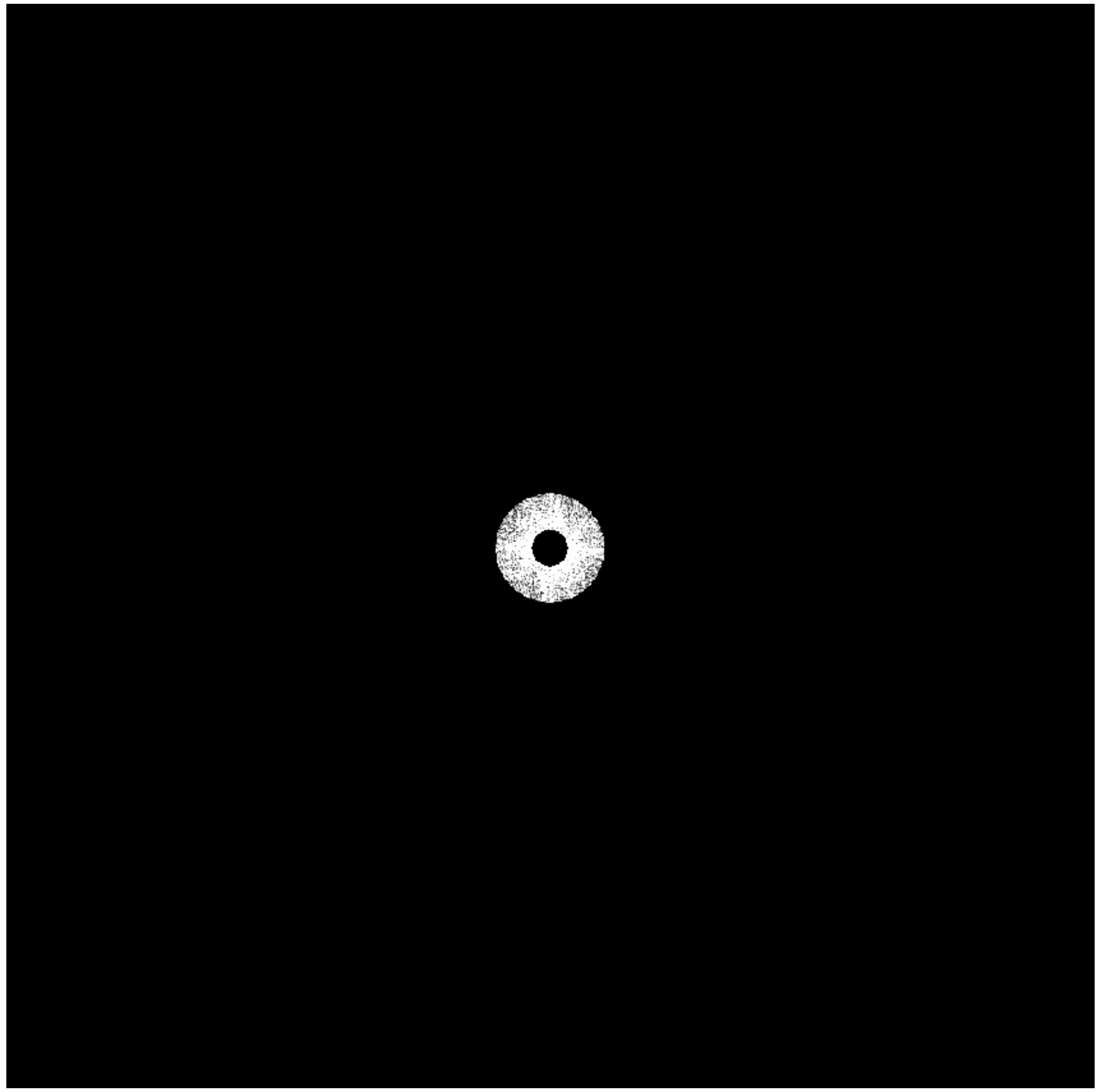
Frequency domain

(after low-pass filter)

All frequencies above cutoff have 0 magnitude

Mid-range frequencies

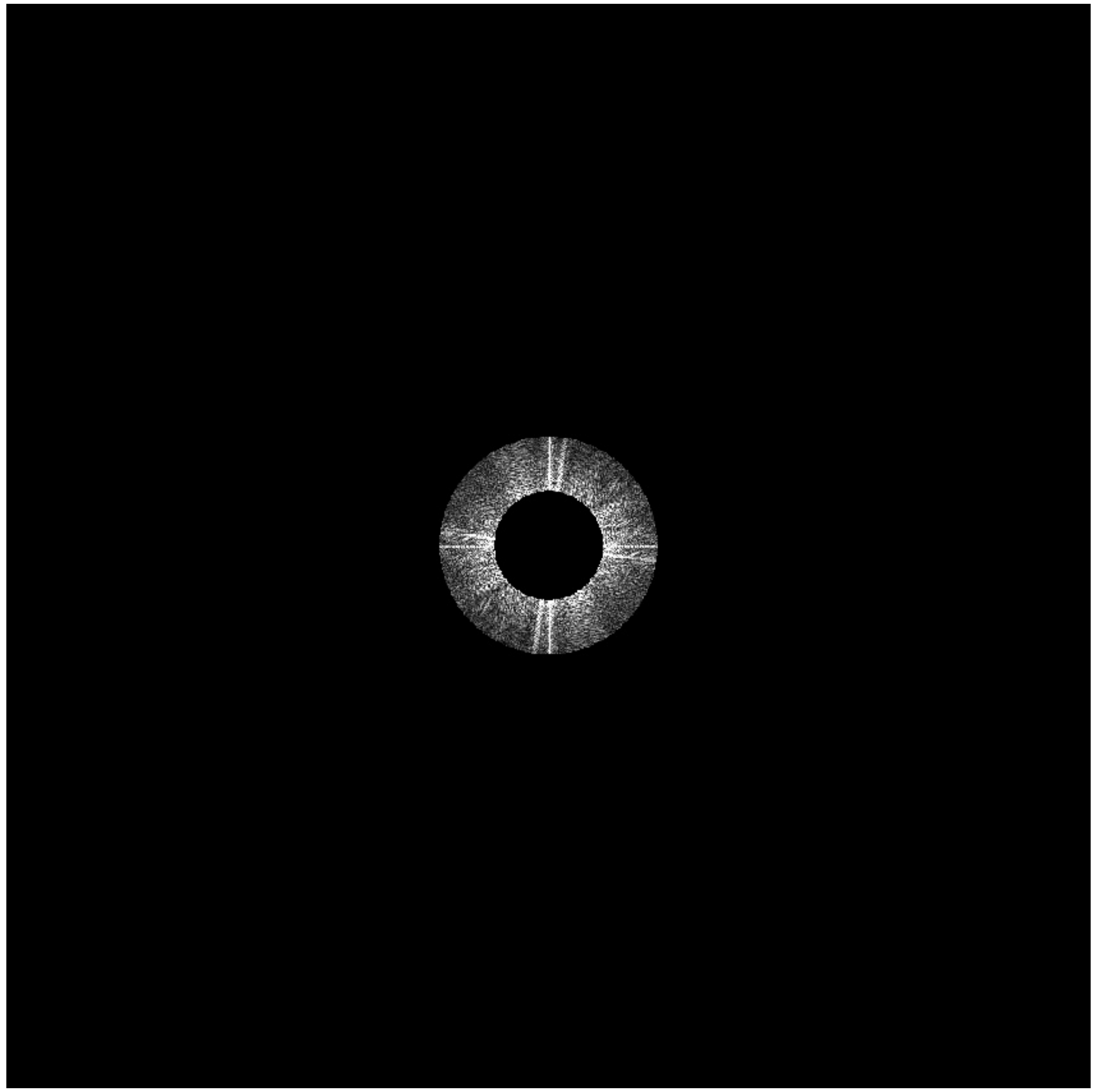
Spatial domain



Frequency domain
(after band-pass filter)

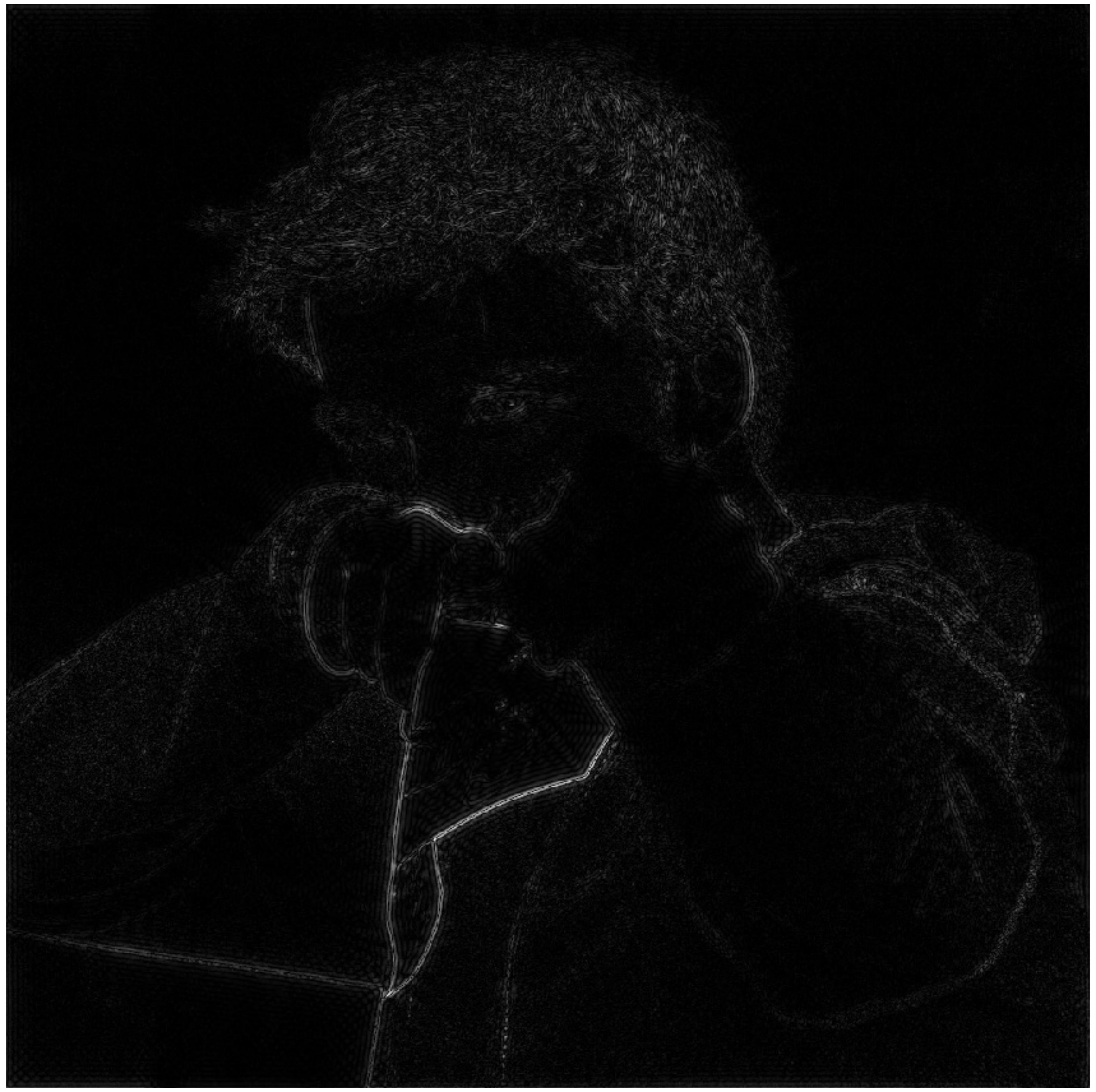
Mid-range frequencies

Spatial domain

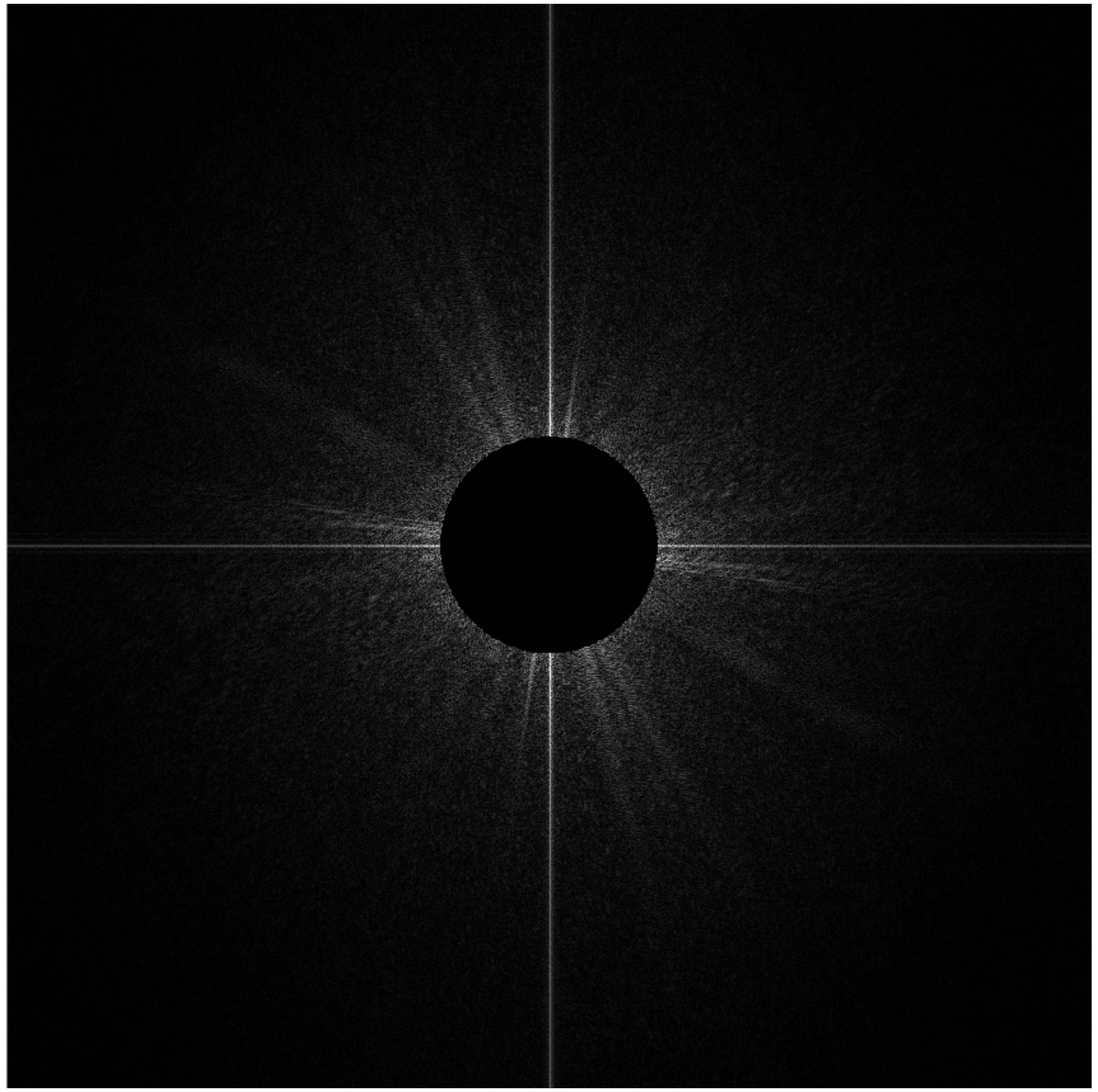


Frequency domain
(after band-pass filter)

High frequencies (edges)

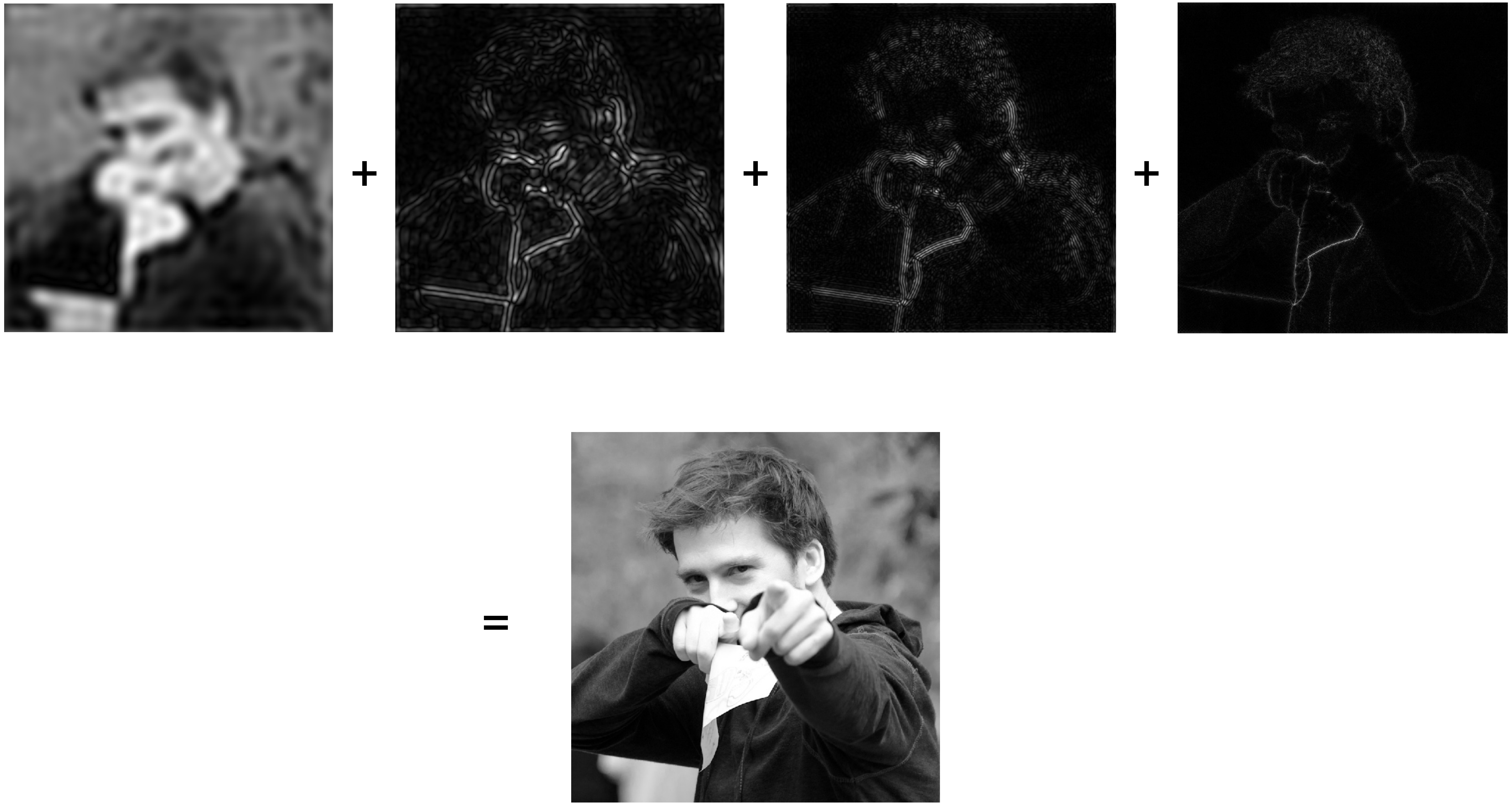


Spatial domain
(strongest edges)

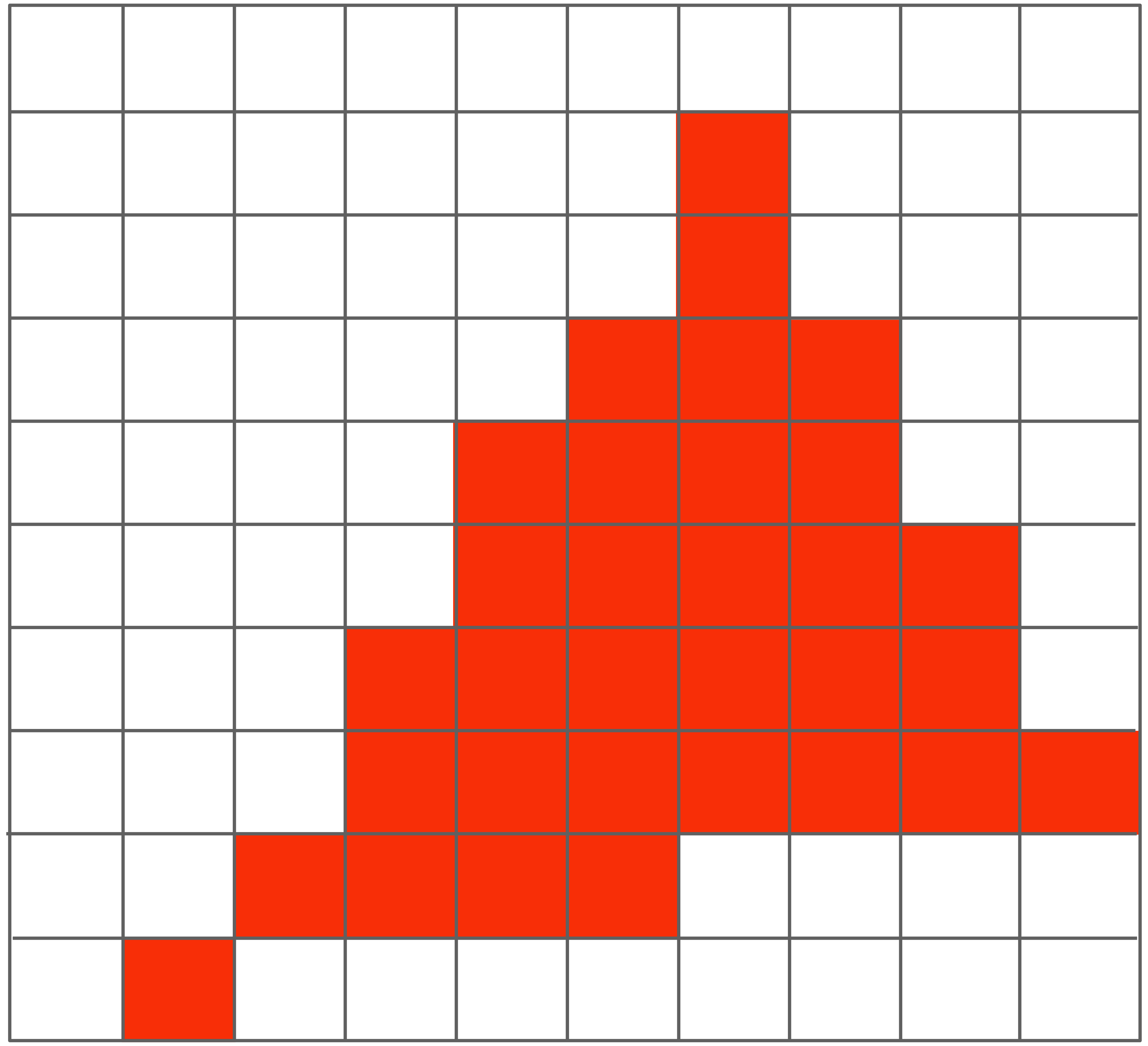


Frequency domain
(after high-pass filter)
All frequencies below threshold have 0 magnitude

An image as a sum of its frequency components

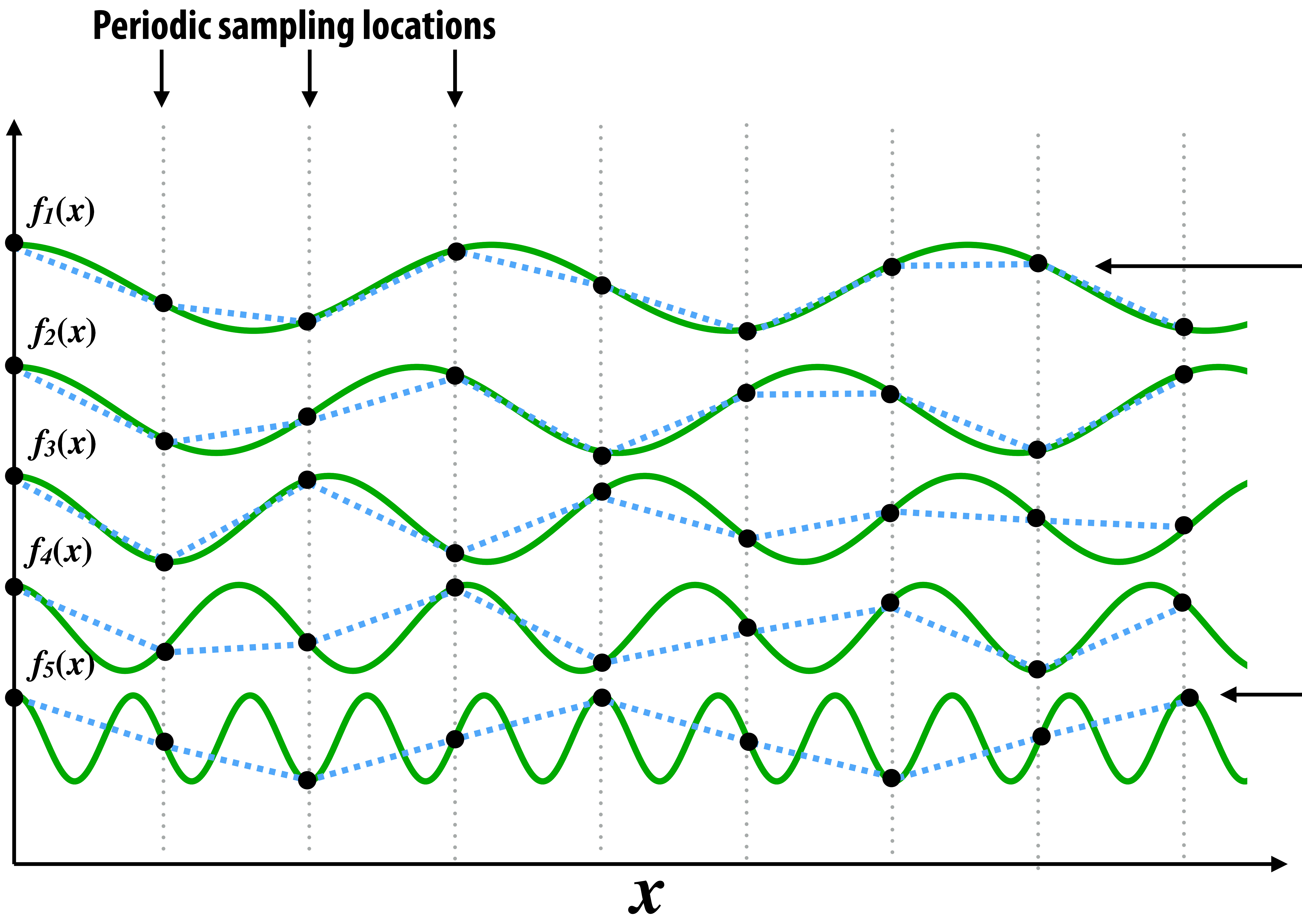


Back to our problem of artifacts in images



Jaggies!

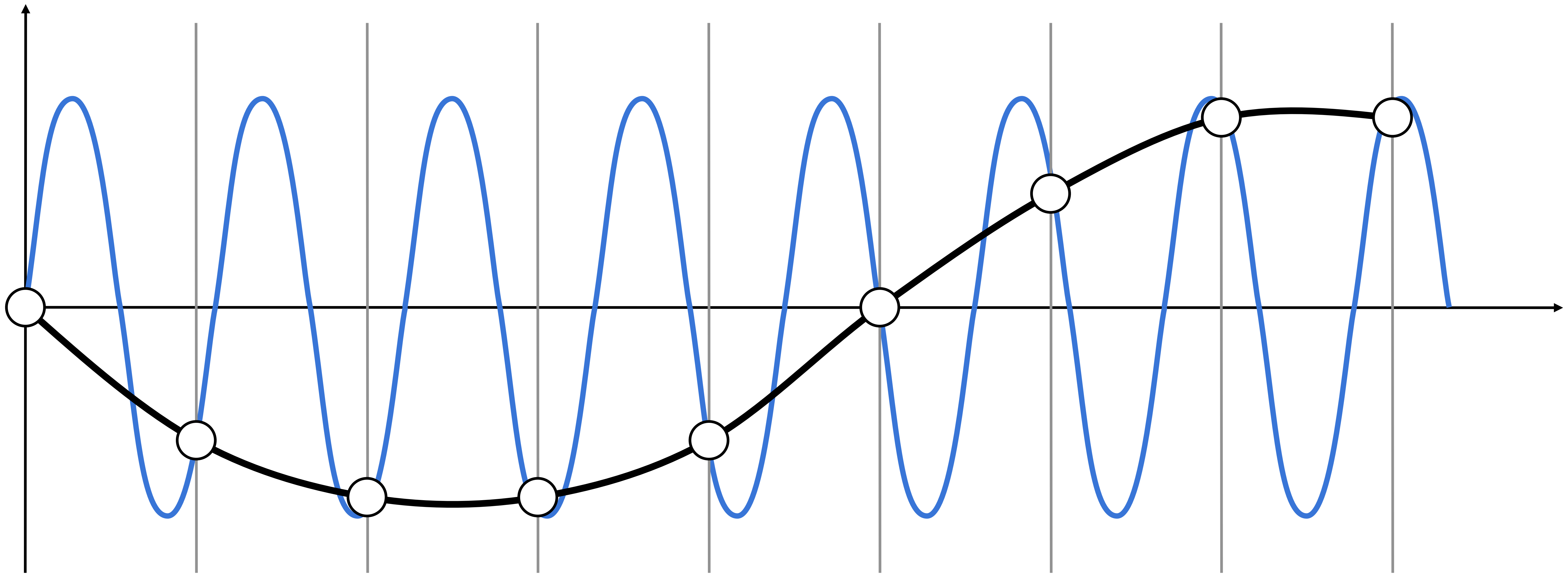
Higher frequencies need denser sampling



Low-frequency signal: sampled adequately for reasonable reconstruction

High-frequency signal is insufficiently sampled: reconstruction incorrectly appears to be from a low frequency signal

Undersampling creates frequency “aliases”

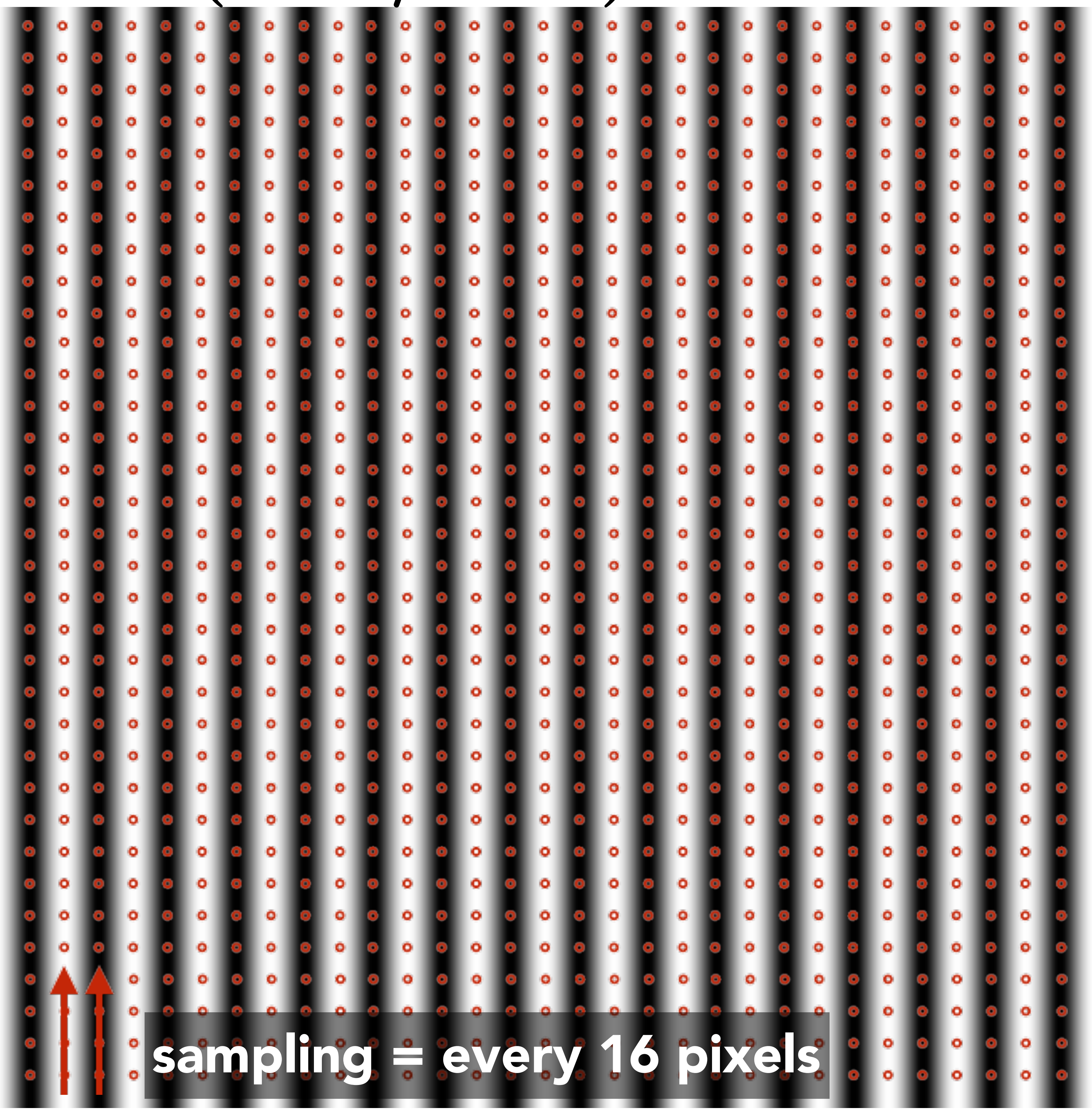


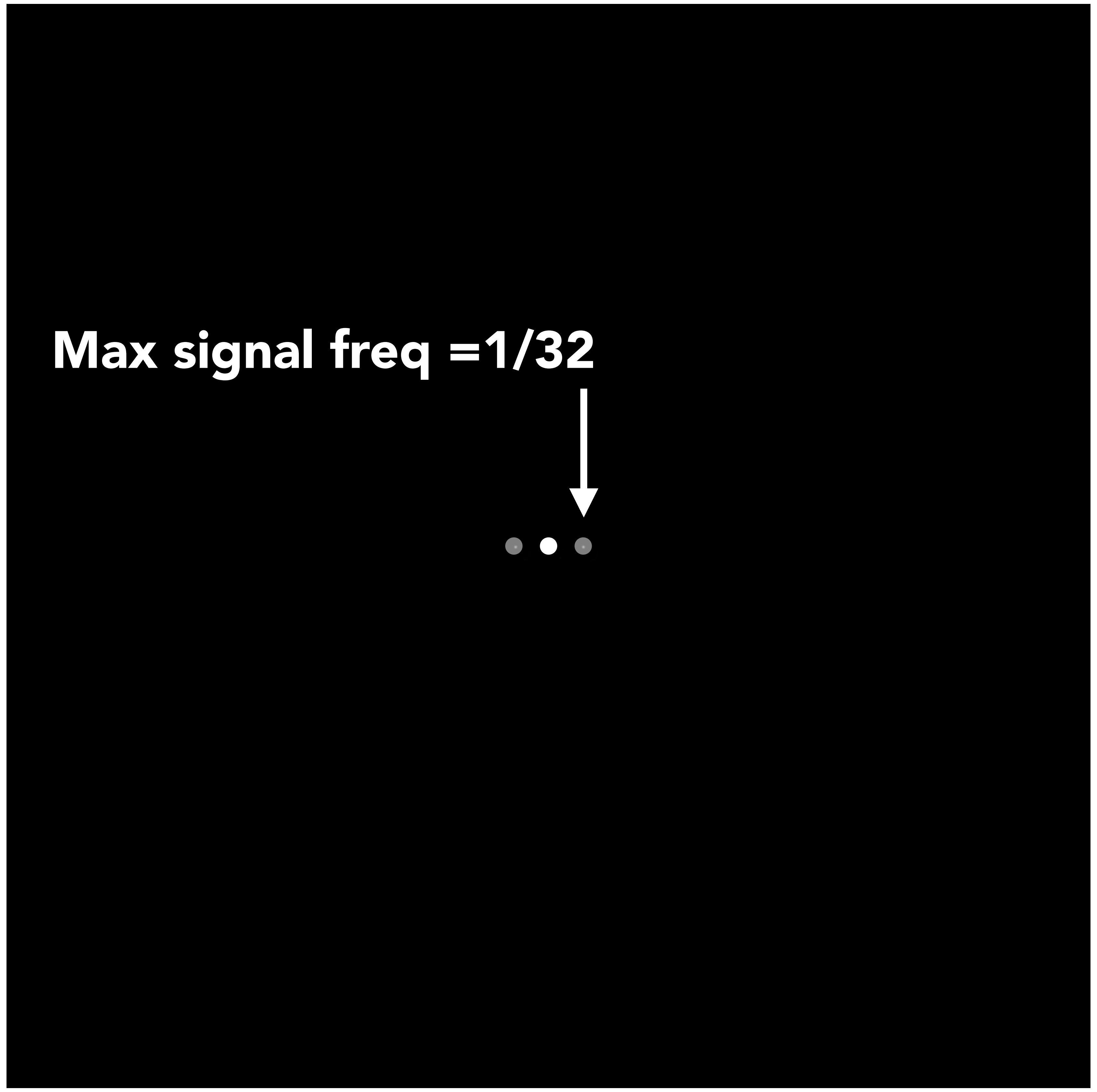
High-frequency signal is insufficiently sampled: samples erroneously appear to be from a low-frequency signal

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

Example: sampling rate vs signal frequency

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





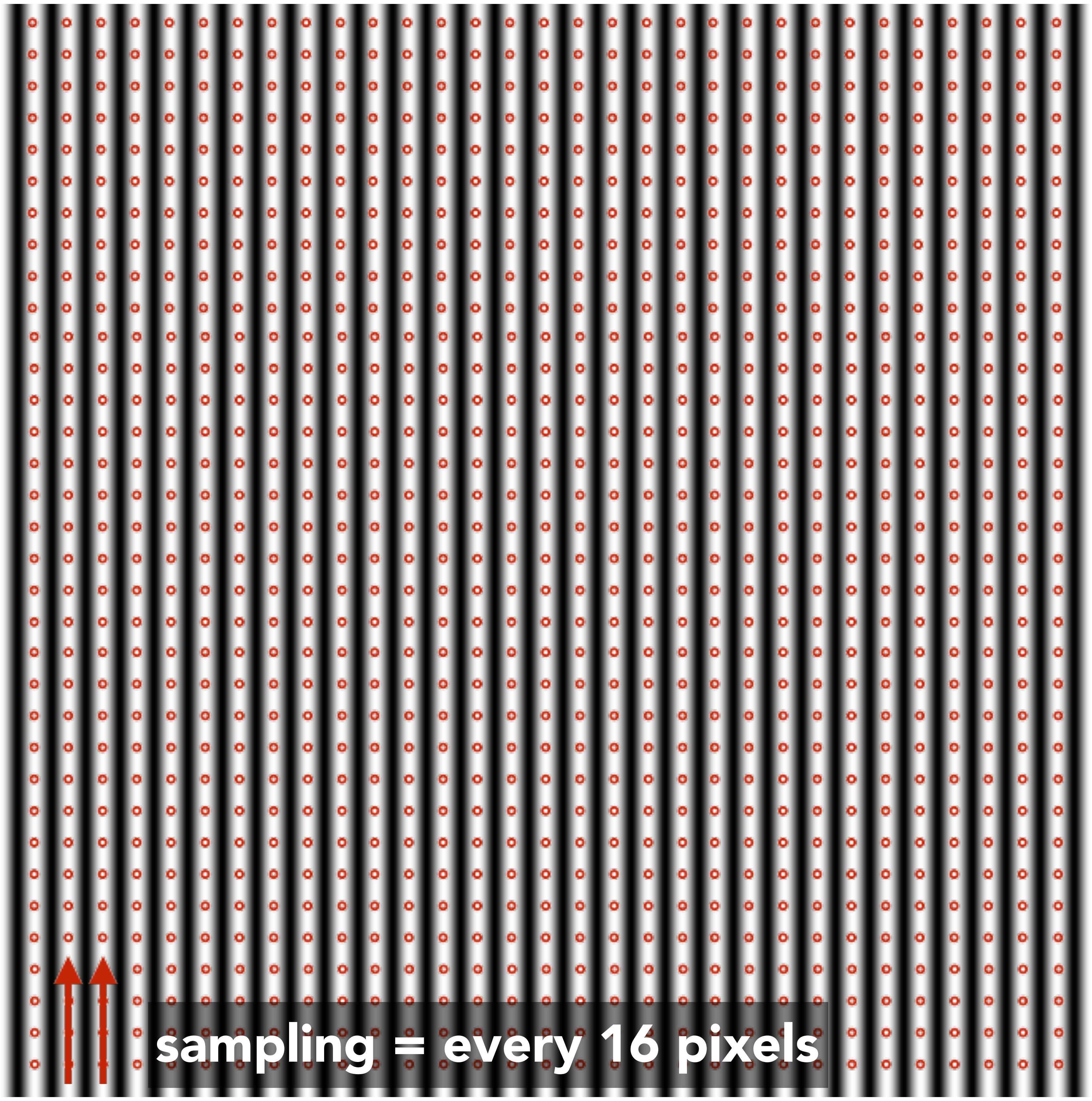
Spatial domain | **Frequency domain**

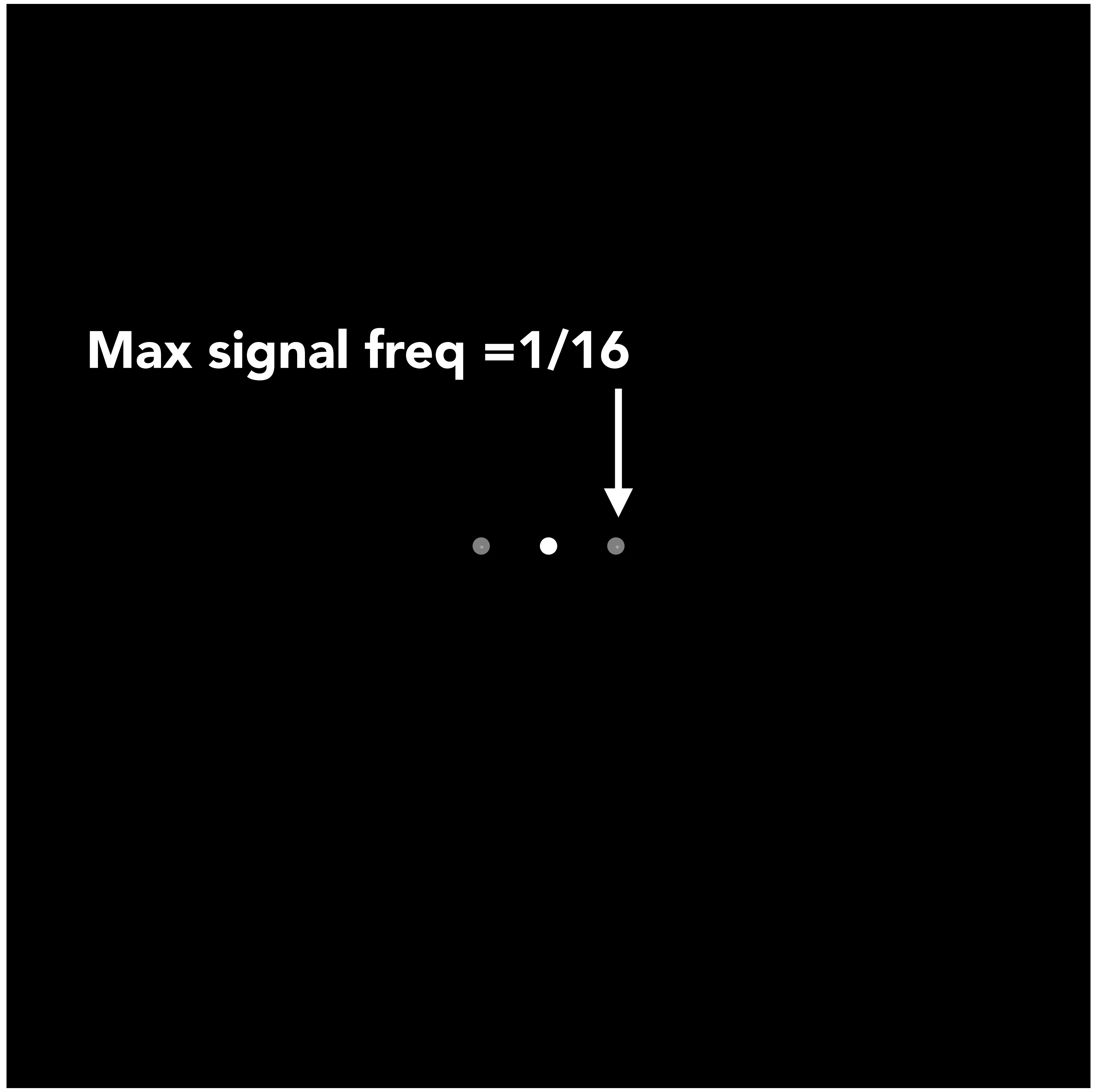
Sampling at twice the frequency of the signal: no aliasing! *

* Technically in this example there is no "pre-aliasing". There is "post-aliasing" if reconstruction from these measurements is not perfect

Example: sampling rate vs signal frequency

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



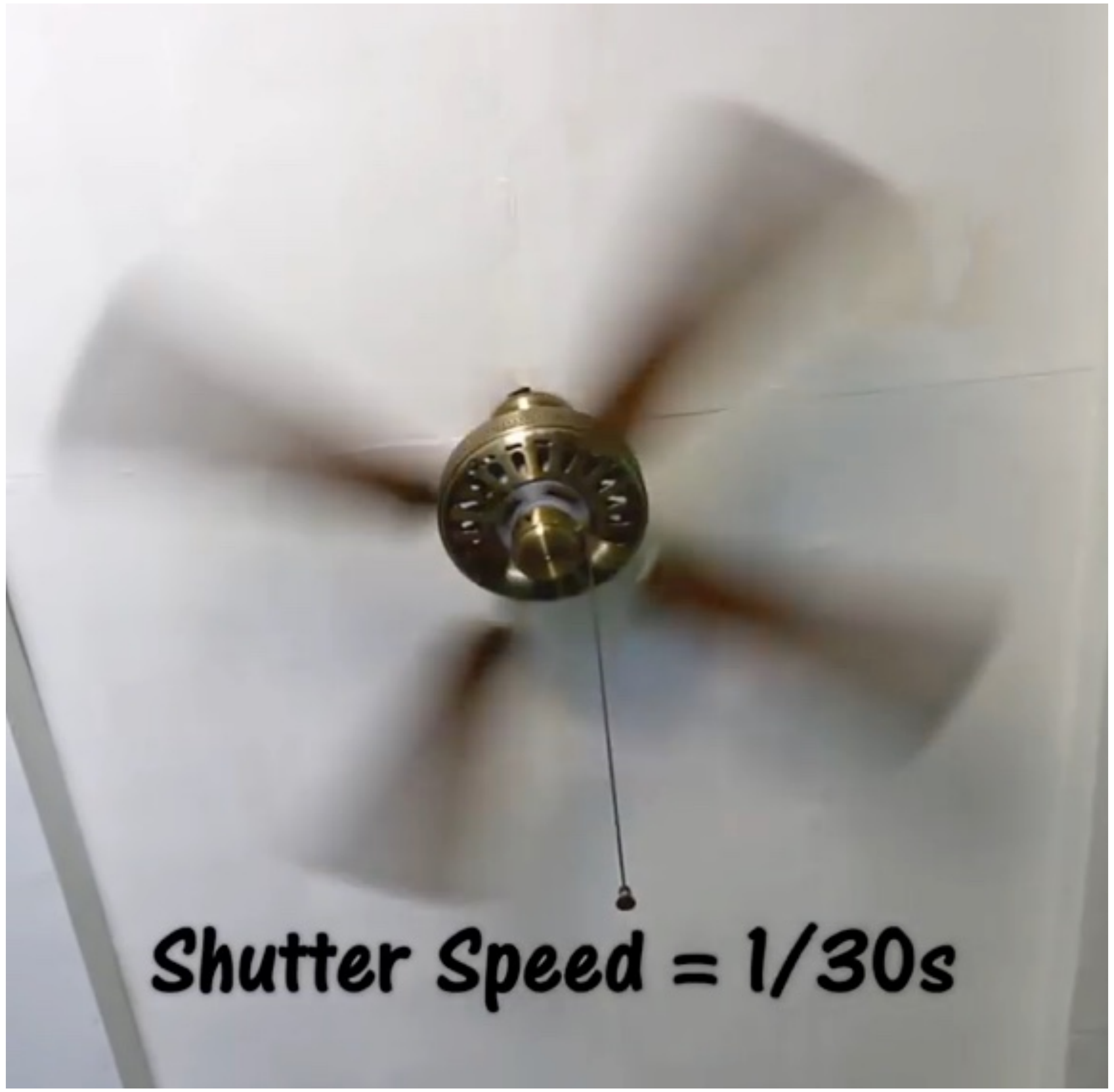


Sampling at same frequency as signal: dramatic aliasing! (due to undersampling)

Anti-aliasing idea:
remove high frequency information from a signal before sampling it

Video: point vs antialiased sampling

Single point in time



Motion blurred

Video: point sampling in time

Credit: Aris & cams youtube, https://youtu.be/NoWwxTktoFs

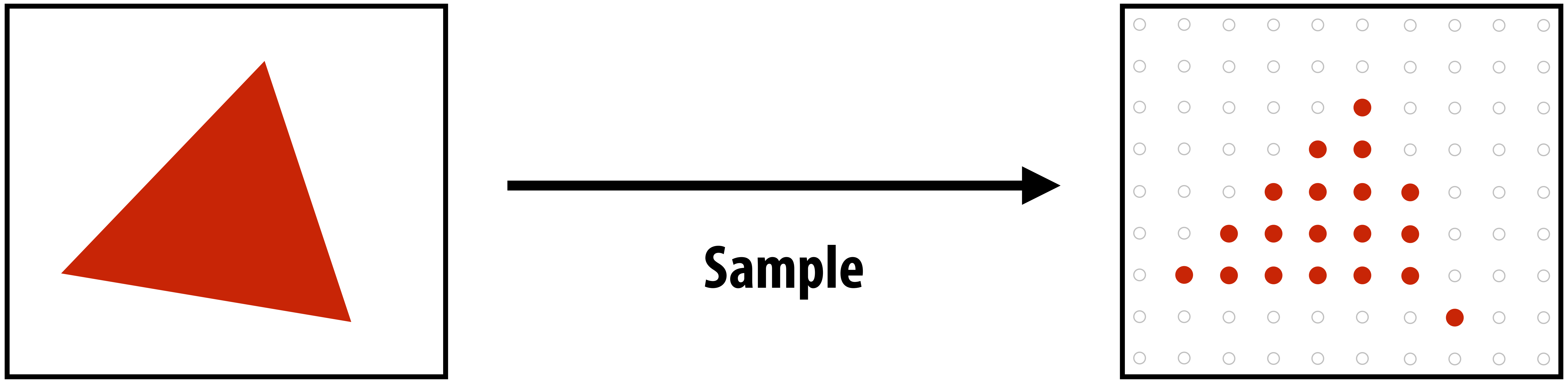
30 fps video. 1/800 second exposure is sharp in time, causes time aliasing.

Video: motion-blurred sampling

Credit: Aris & cams youtube, https://youtu.be/NoWwxTktoFs

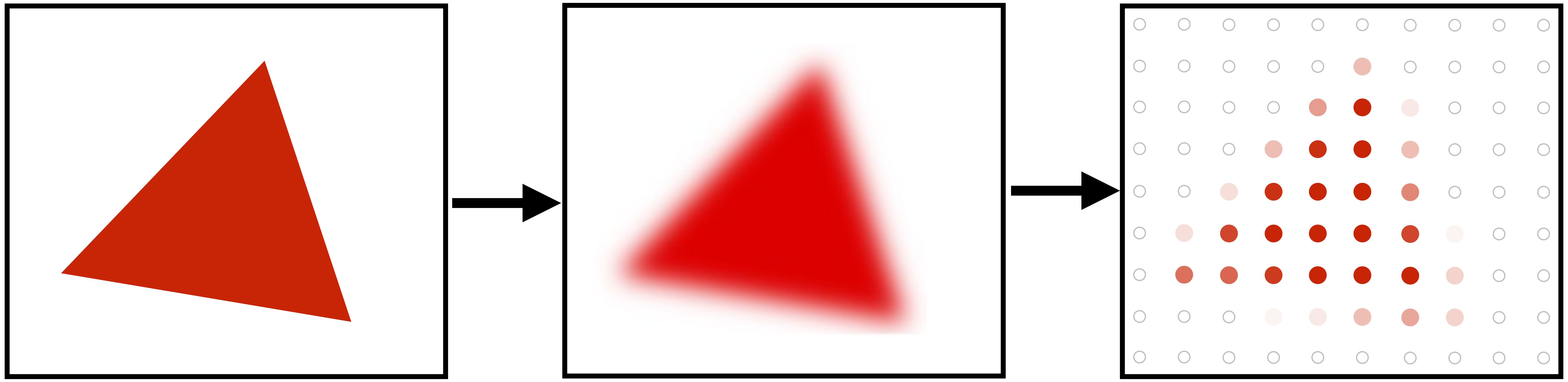
30 fps video. 1/30 second exposure is motion-blurred in time, reduces aliasing.

Drawing a triangle is sampling the triangle coverage signal



Note jaggies in rasterized triangle
(pixel values are either red or white: sample is in or out of triangle)

Anti-aliasing by pre-filtering the signal



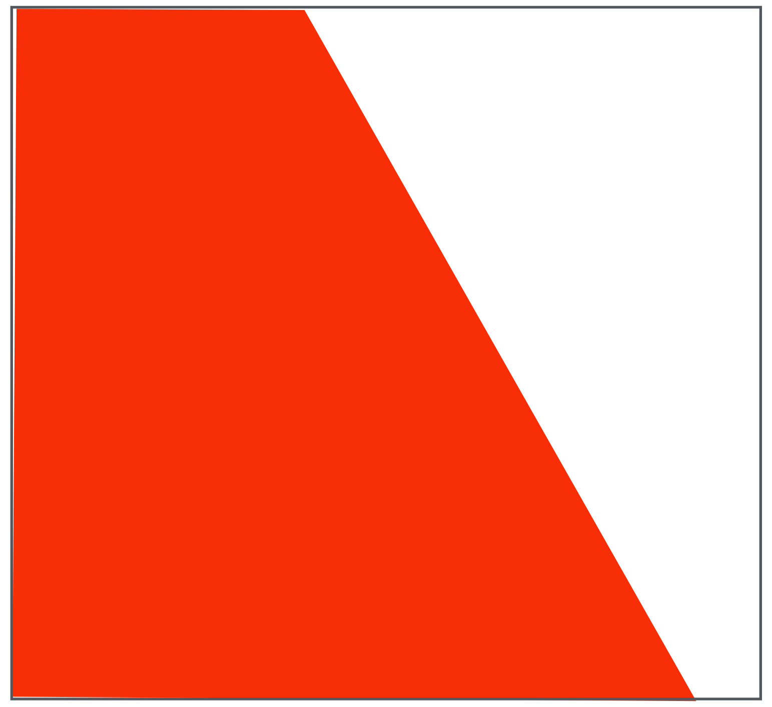
Pre-filter

(remove high frequency detail)

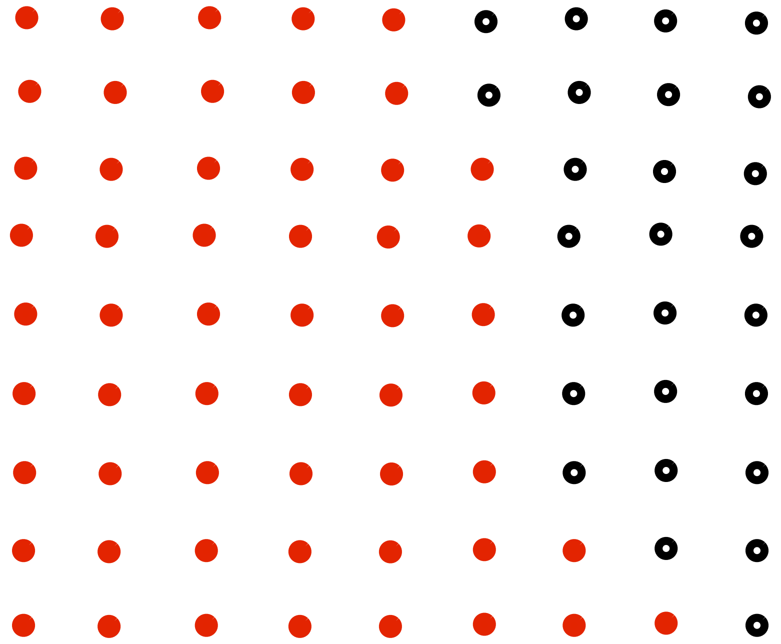
Sample

Note anti-aliased edges of rasterized triangle:
pixel values take intermediate values

Pre-filtering by “supersampling” then “blurring” (averaging)

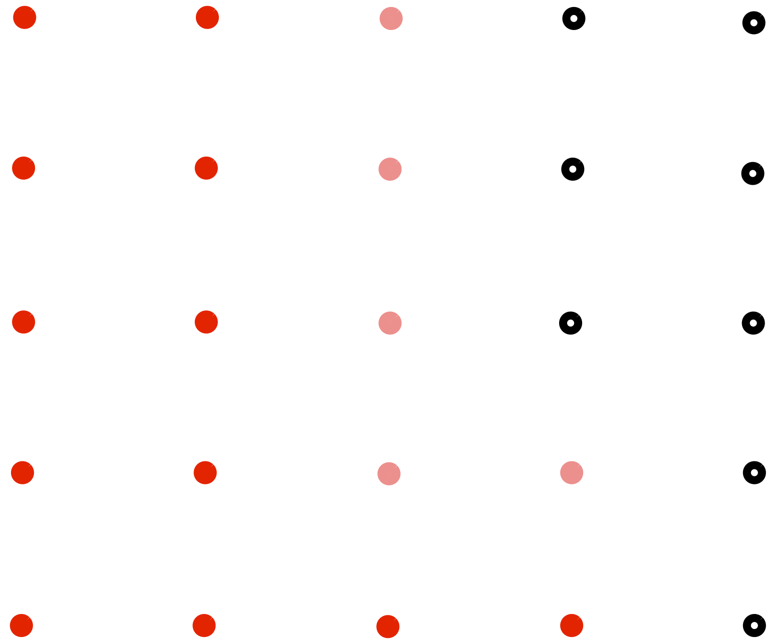


Original signal
(with high frequency edge)

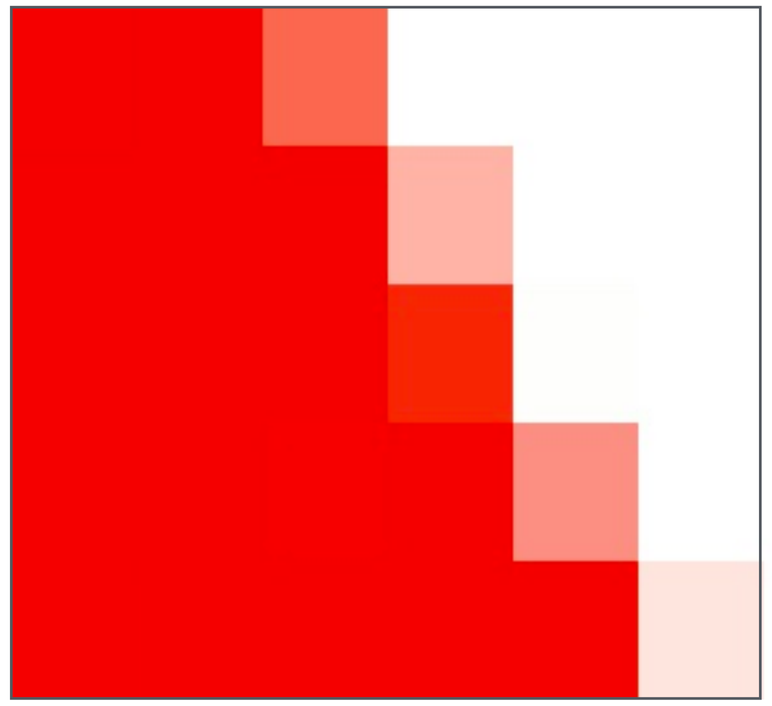


Dense sampling of signal
(supersampling)

Reconstructed signal with high frequencies reduced
(Blurring via averaging over pixel, etc)

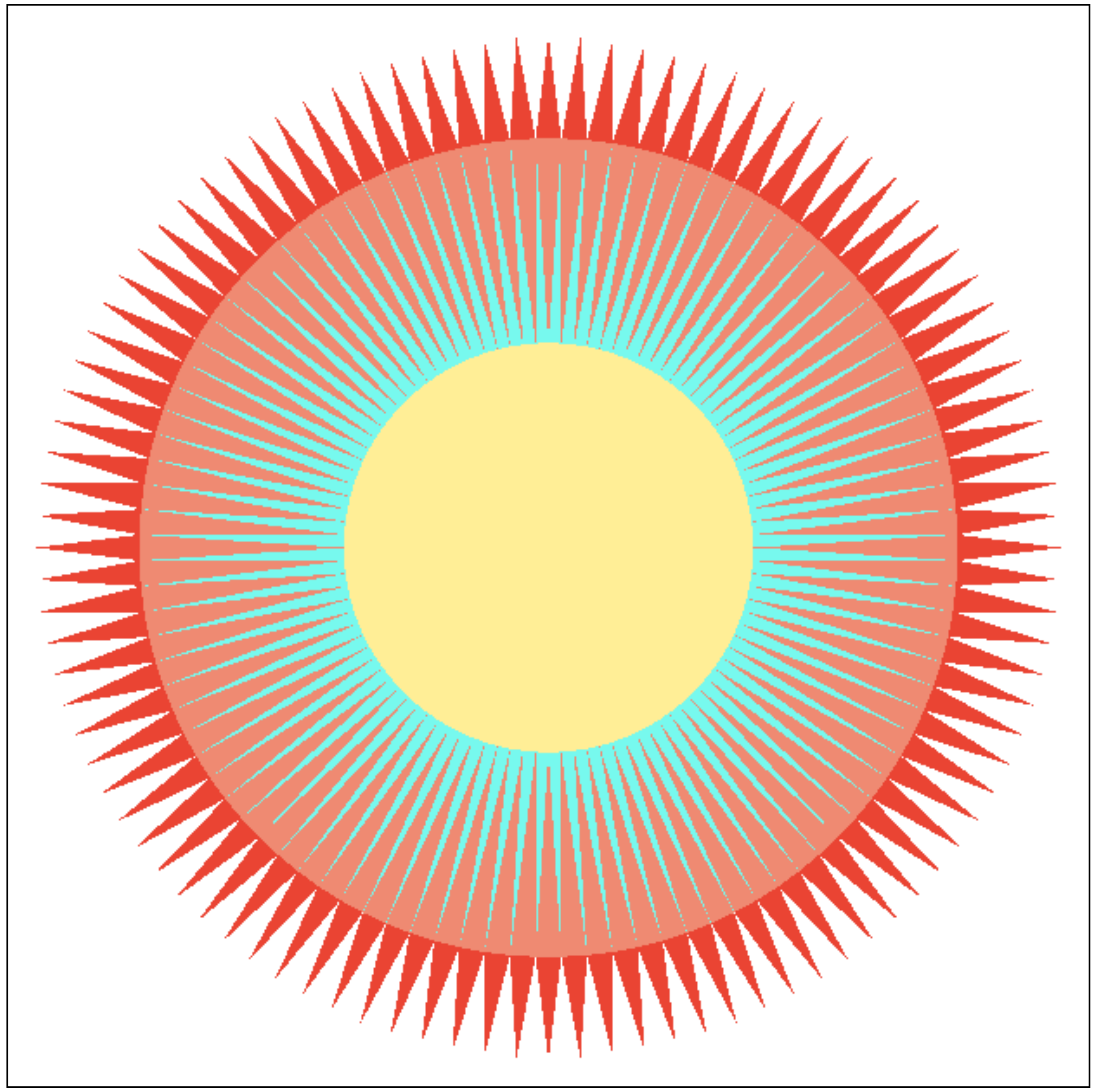


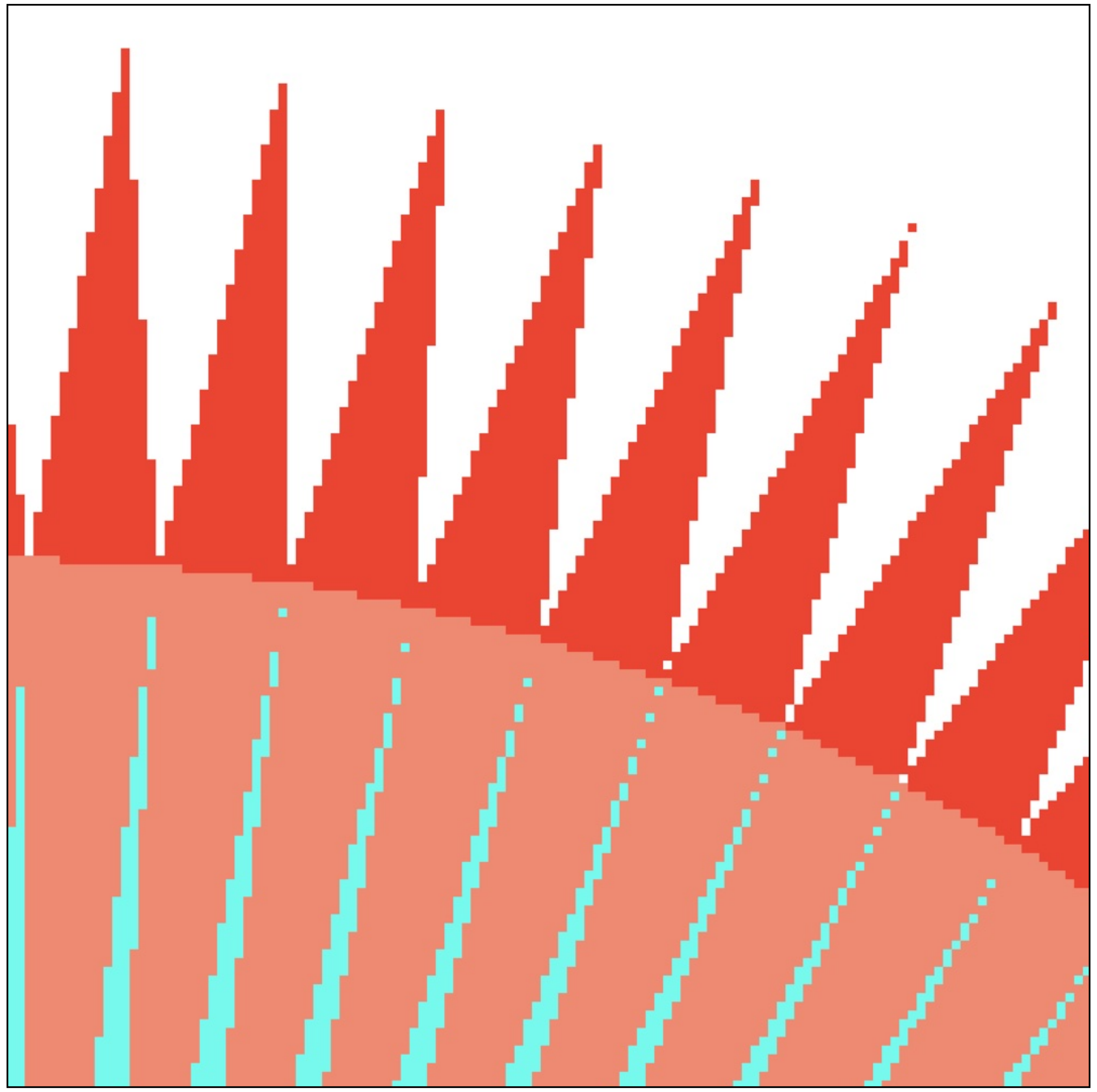
Coarsely sampled signal (e.g., once per pixel,
to store in image, or send to display)



Reconstruction on display

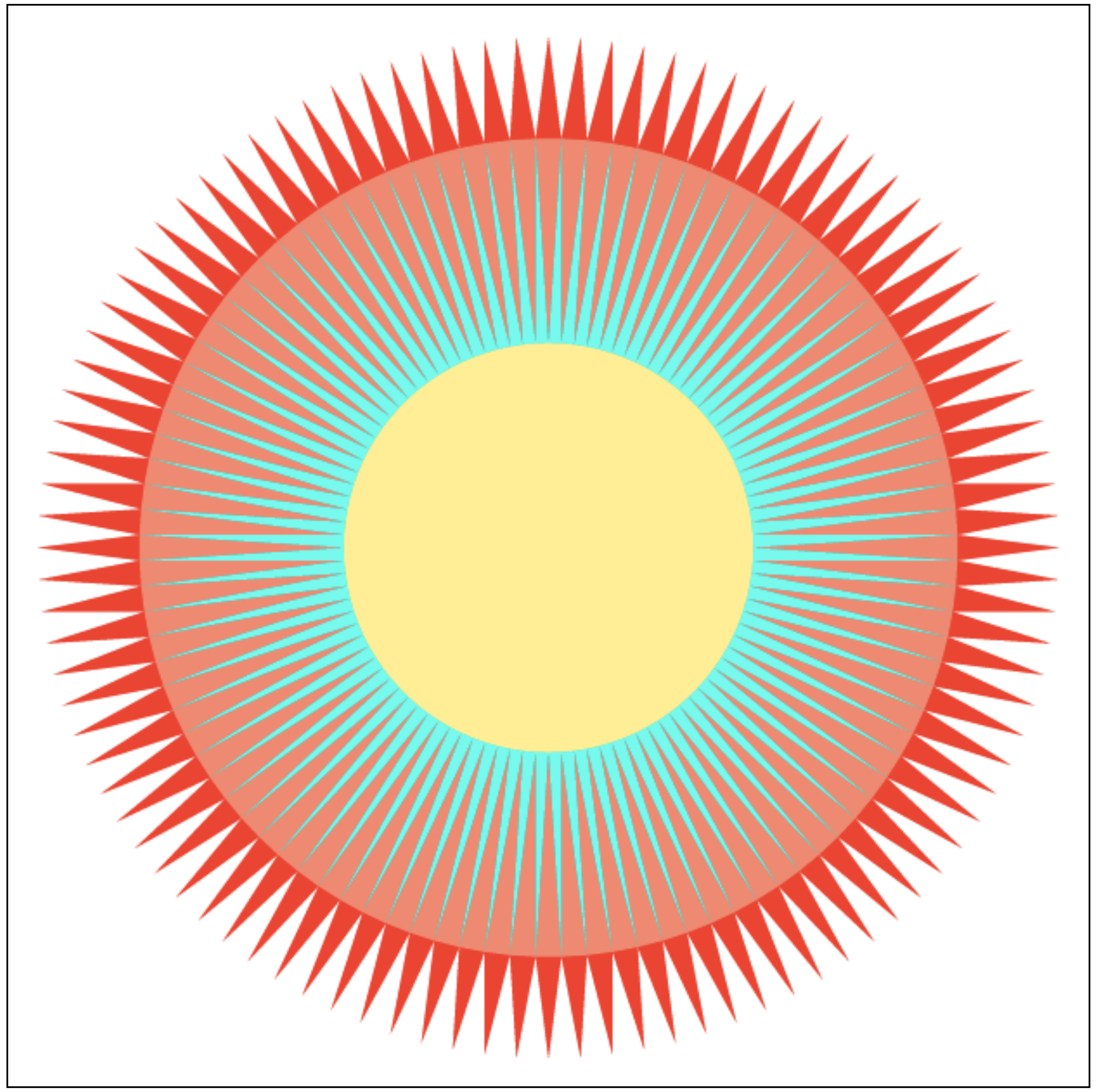
Images rendered using one sample per pixel

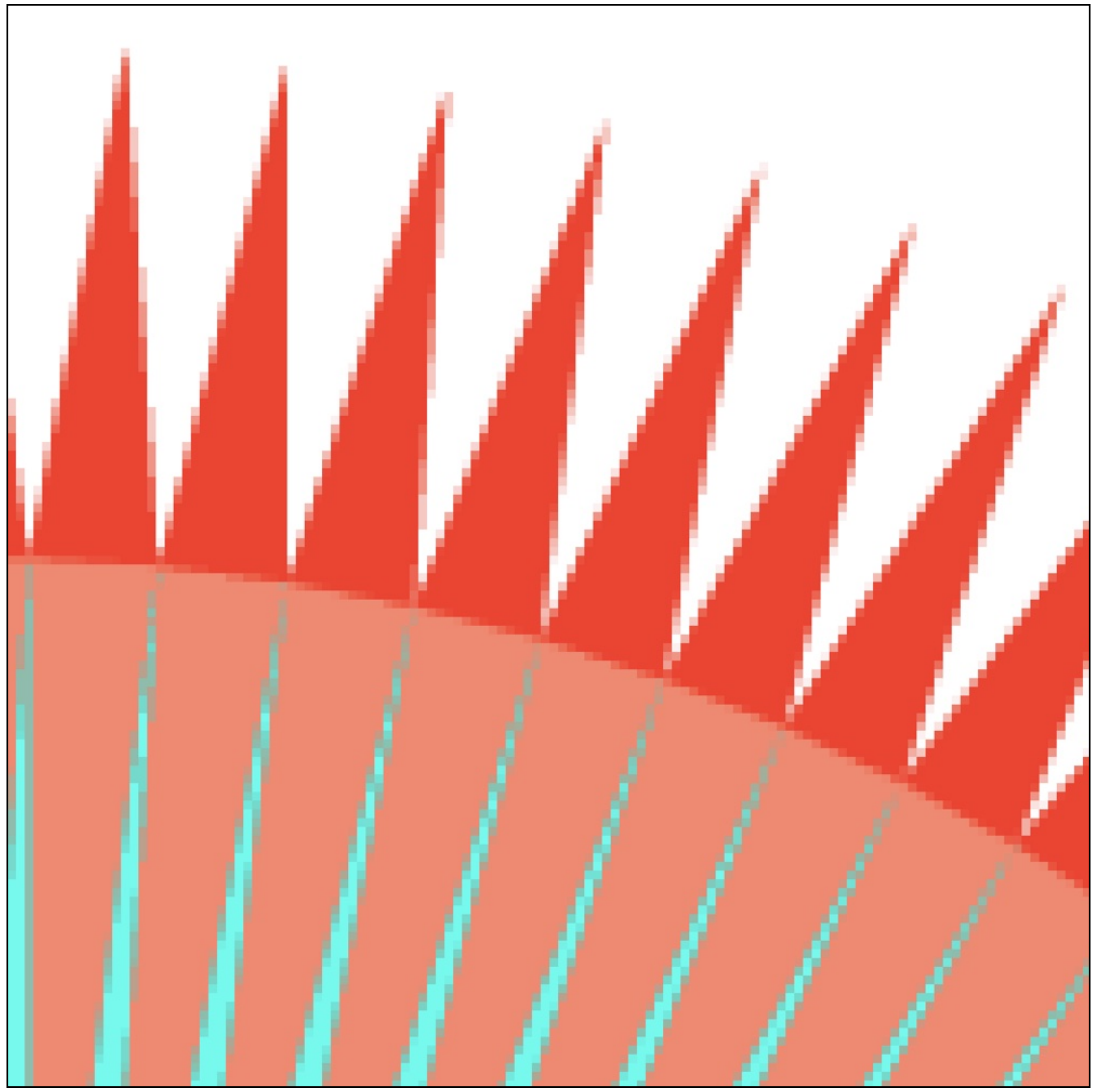




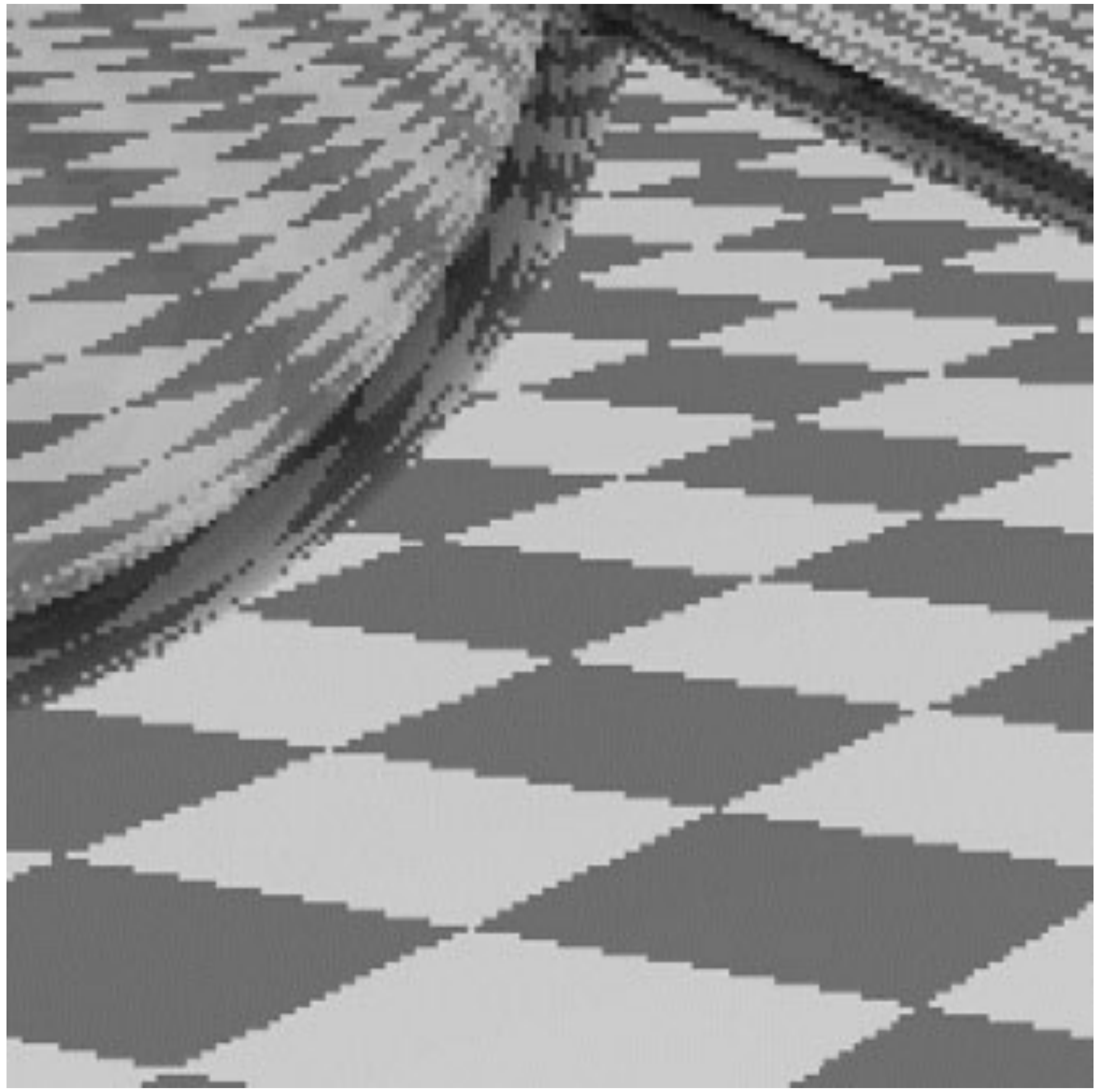
Anti-aliased results (multiple samples per pixel)

(Images below contain same number of pixels as images on prior slide)

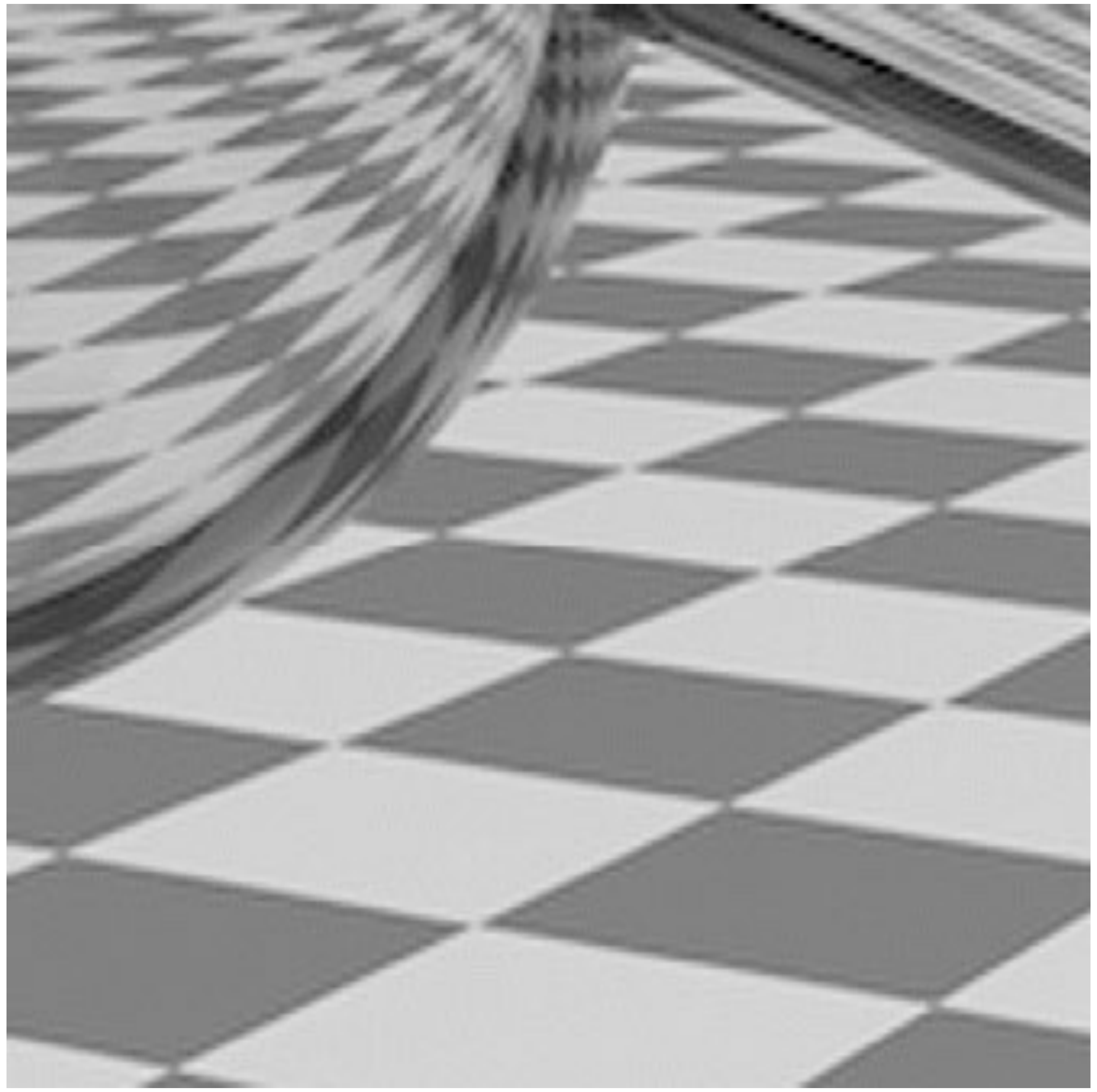




Benefits of anti-aliasing



Jaggies



Pre-filtered

Filtering = convolution

1D convolution ("weighted average over a window")

Signal

1	3	5	3	7	1	3	8	6	4

Filter

1	2	1

1D convolution ("weighted average over a window")

Signal

1	3	5	3	7	1	3	8	6	4

Filter

1	2	1

1x1 + 3x2 + 5x1 = 12

Result

12									

1D convolution

Signal										
	1	3	5	3	7	1	3	8	6	4

Filter			
	1	2	1

3x1 + 5x2 + 3x1 = 16

Result										
	12	16								

1D convolution

Signal

1	3	5	3	7	1	3	8	6	4

Filter

1	2	1

$$5 \times 1 + 3 \times 2 + 7 \times 1 = 18$$

Result

12	16	18							

Box filter (common filter used in a 2D convolution)

$$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Example: 3x3 box filter

2D convolution with box filter blurs the image

Original image

→

Blurred
(convolve with box filter)

Hmm… this reminds me of a low-pass filter…

Discrete 2D convolution

$$(f * g)(x, y) = \sum_{i,j=-\infty}^{\infty} f(i, j) I(x - i, y - j)$$

output image

filter

input image

Consider $f(i, j)$ **that is non-zero only when:** $-1 \leq i, j \leq 1$

Then:

$$(f * g)(x, y) = \sum_{i,j=-1}^{1} f(i, j) I(x - i, y - j)$$

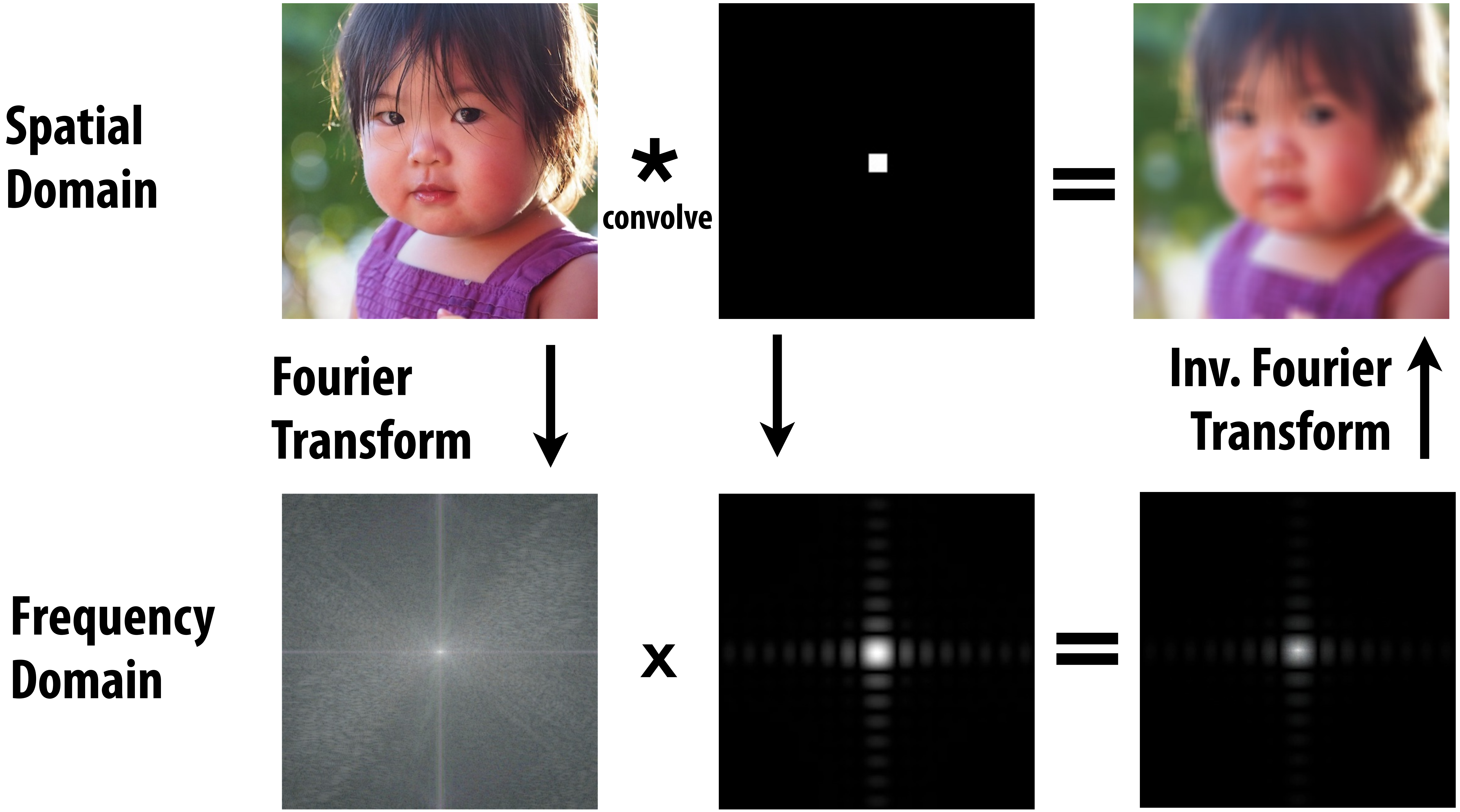
And we can represent f(i,j) as a 3x3 matrix of values where:

$$f(i, j) = \mathbf{F}_{i,j}$$

(often called: "filter weights", "filter kernel")

Convolution theorem

Convolution in the spatial domain is equal to multiplication in the frequency domain, and vice versa



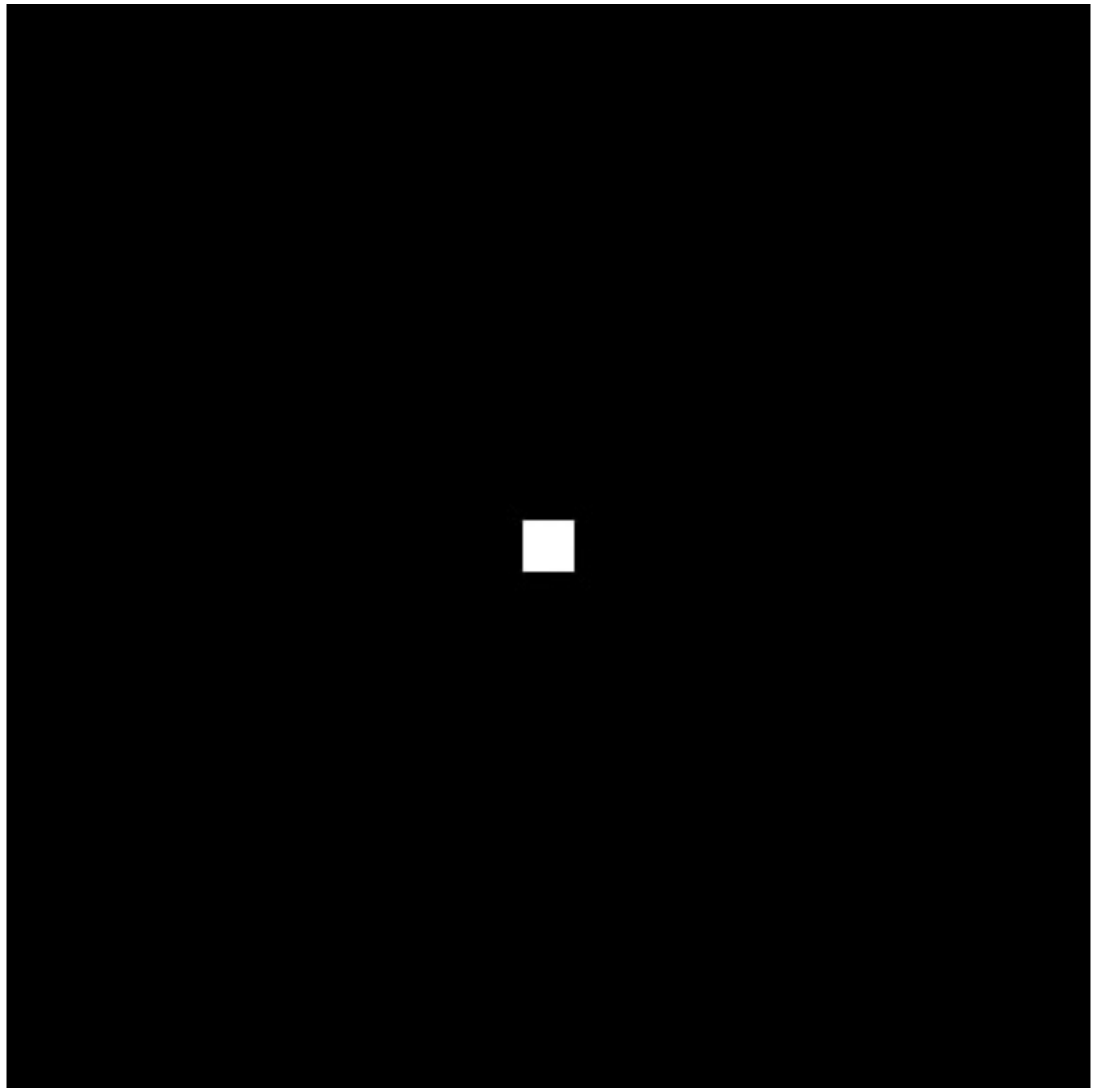
Convolution theorem

- Convolution in the spatial domain is equal to multiplication in the frequency domain, and vice versa

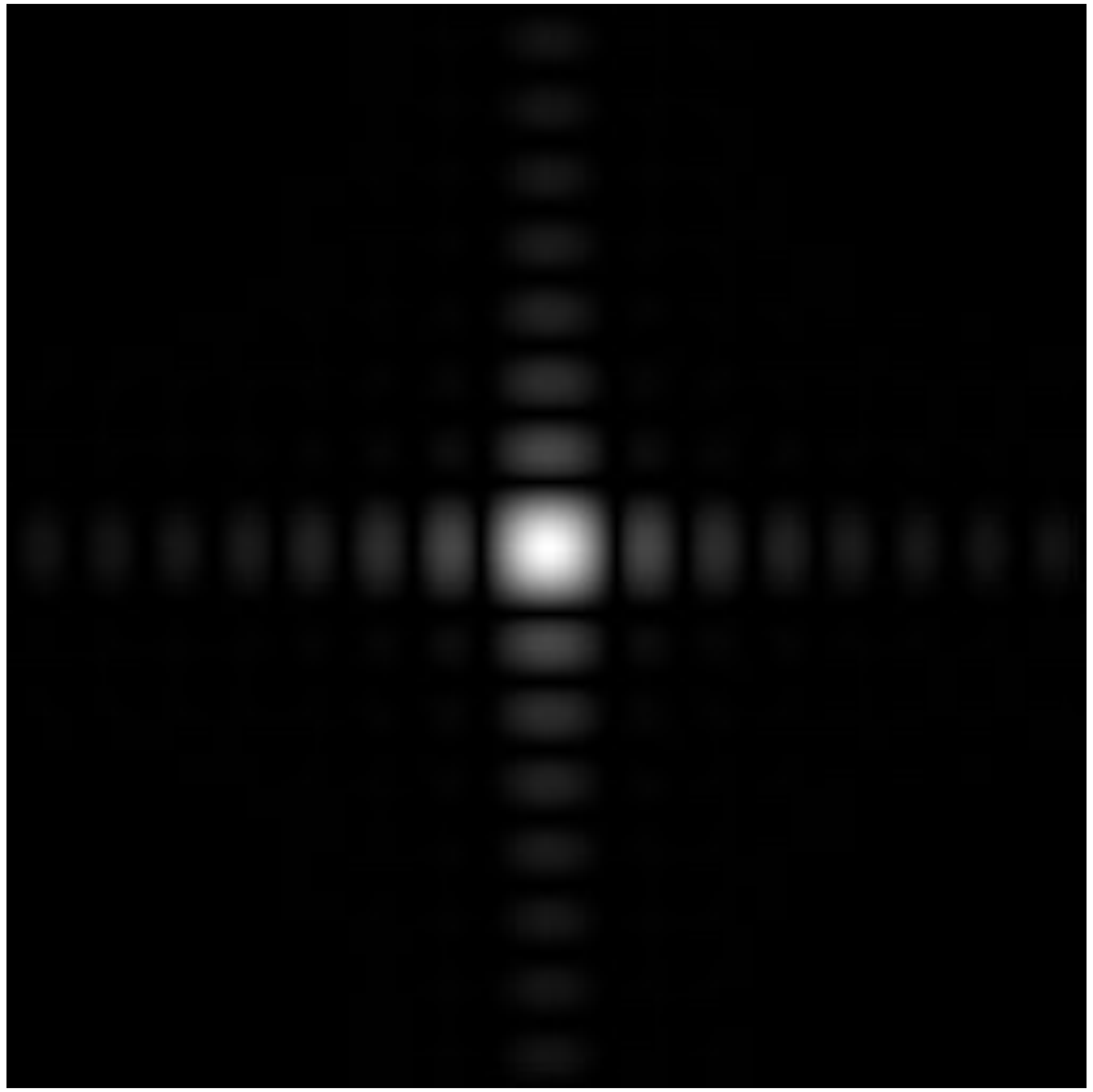
- Pre-filtering option 1:
  - Filter by convolution in the spatial domain

- Pre-filtering option 2:
  - Transform to frequency domain (Fourier transform)
  - Multiply by Fourier transform of convolution kernel
  - Transform back to spatial domain (inverse Fourier)

Box function = “low pass” filter

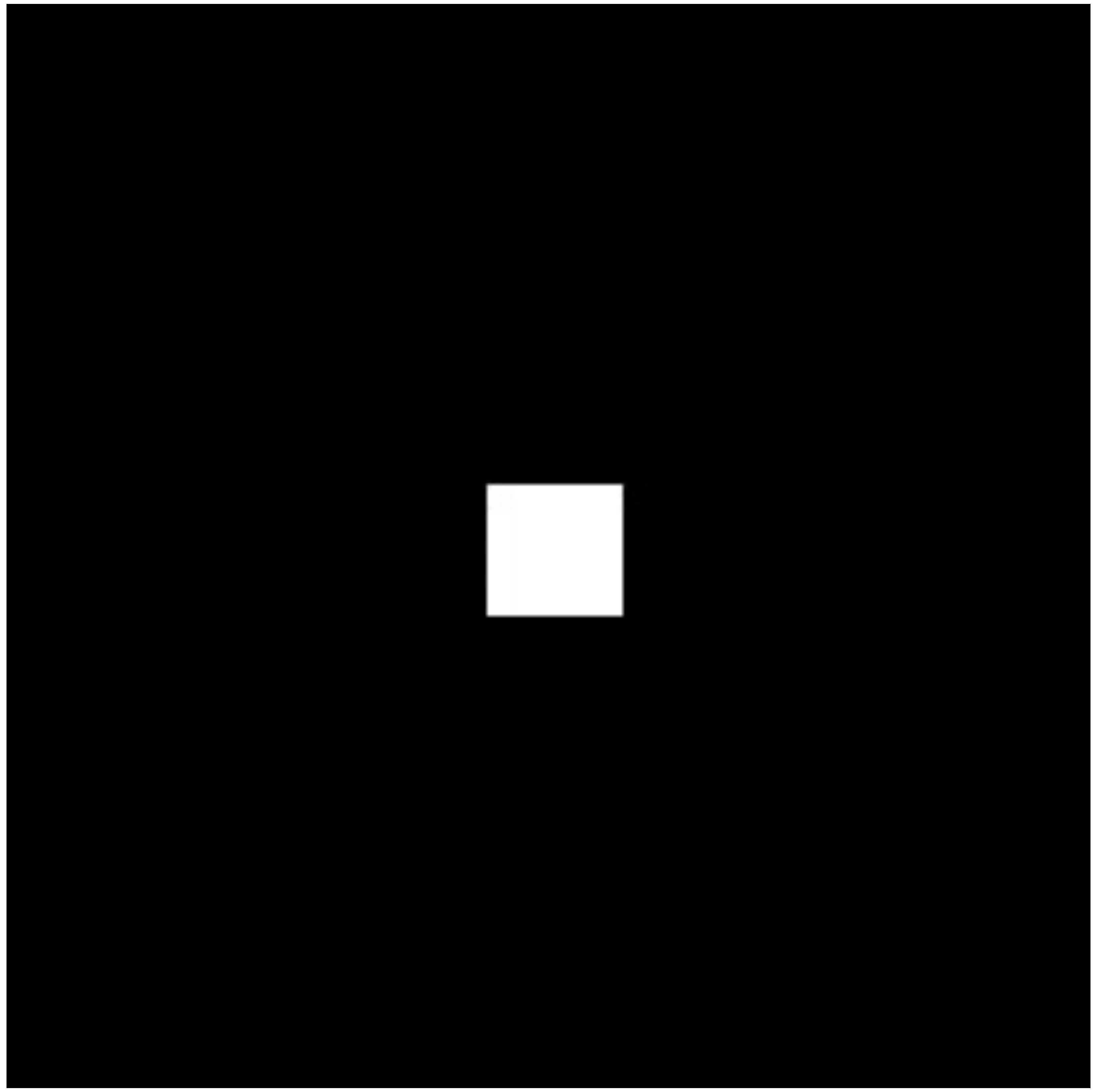


Spatial domain

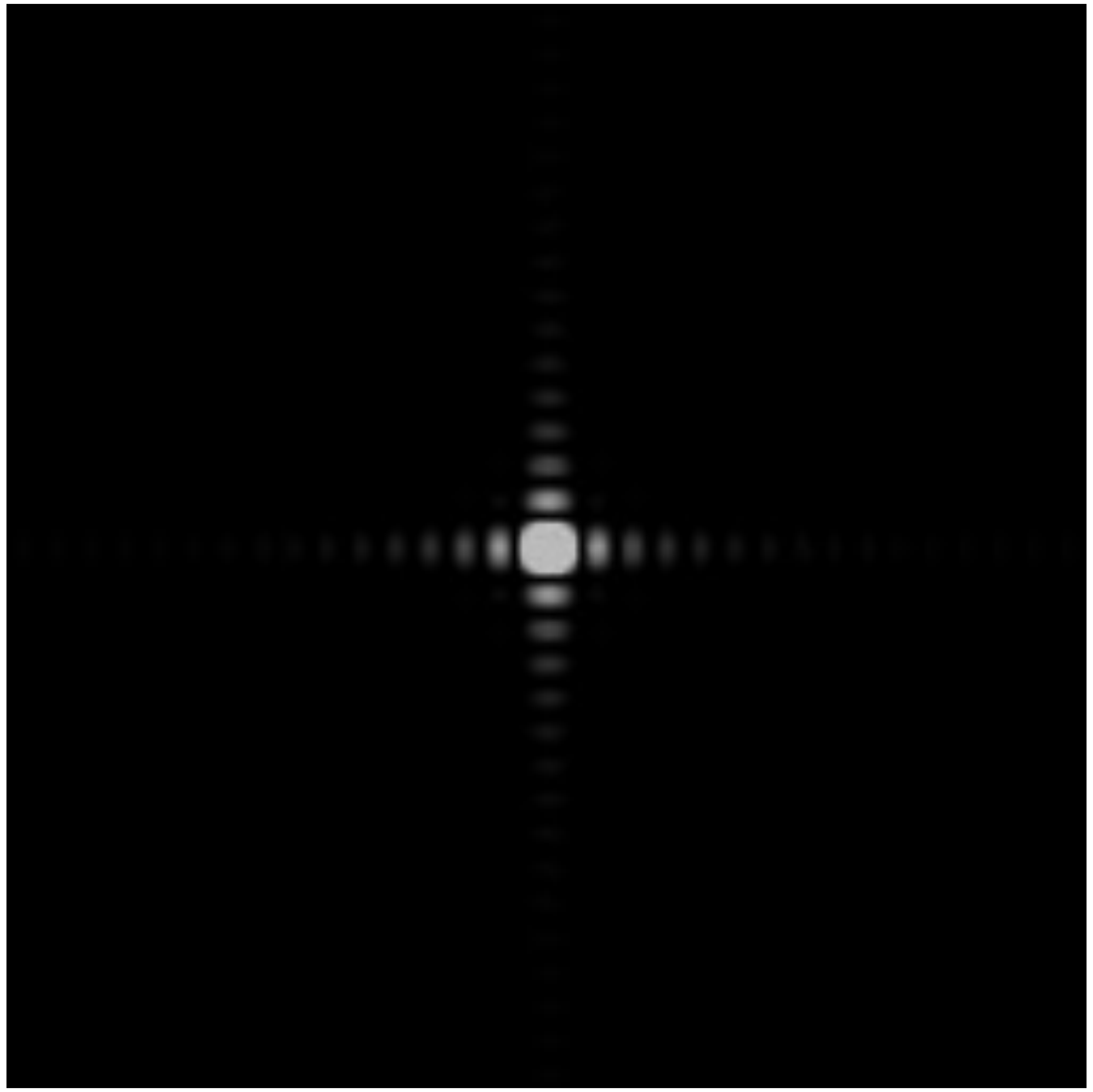


Frequency domain

Wider filter kernel = retain only lower frequencies



Spatial domain



Frequency domain

Wider filter kernel = lower frequencies

- As a filter is localized in the spatial domain,
  it spreads out in frequency domain

- Conversely, as a filter is localized in frequency domain, it spreads out in the spatial domain

How can we reduce aliasing error?

- Increase sampling rate
  - Higher resolution displays, sensors, framebuffers…
  - But: costly and may need very high resolution to sufficiently reduce aliasing

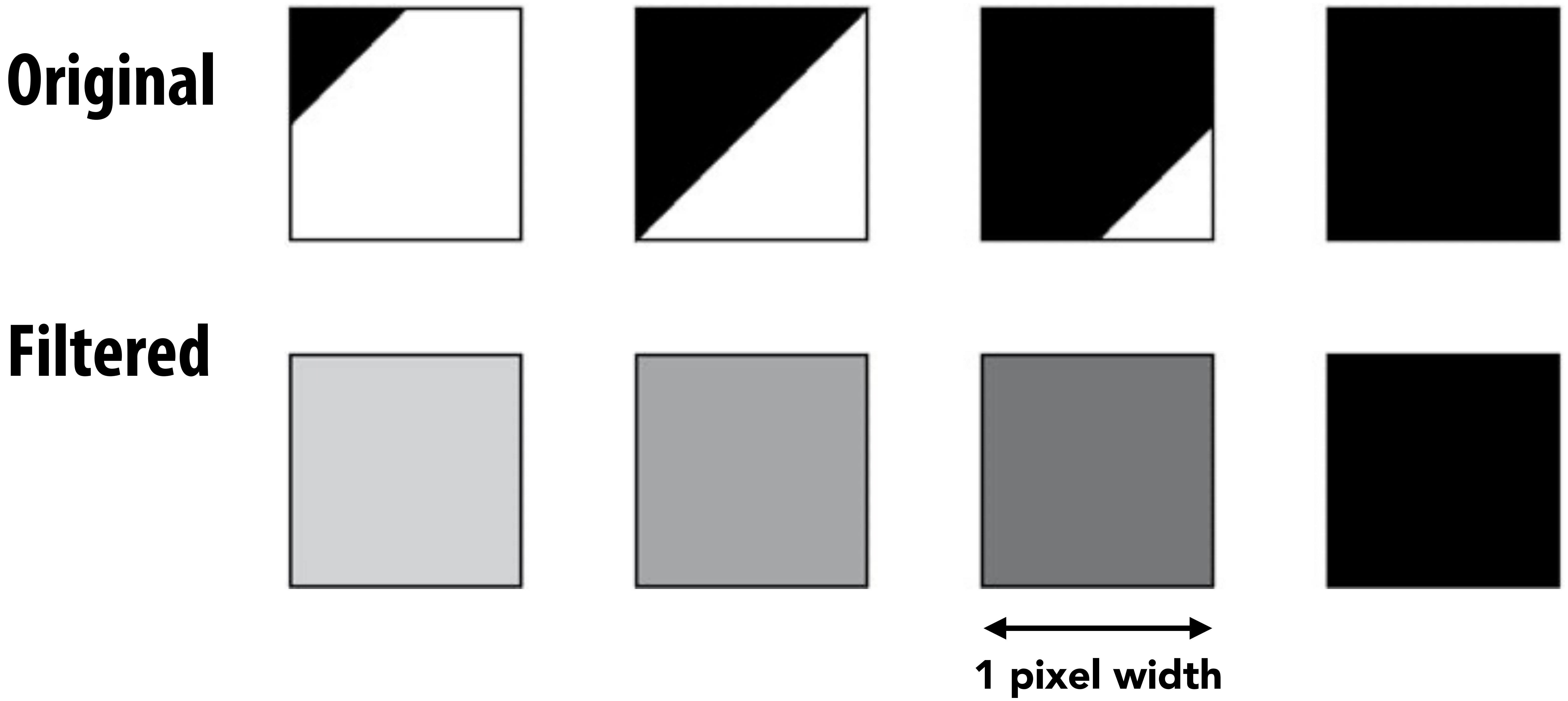
- Anti-aliasing
  - Simple idea: remove (or reduce) high frequencies before sampling
  - How to filter out high frequencies before sampling?

Anti-aliasing by averaging values in pixel area

- Convince yourself the following are the same:
- Option 1:
  - Convolve f(x,y) by a 1-pixel box-blur
  - Then sample the resulting signal at the center of every pixel
- Option 2:
  - Compute the average value of f(x,y) in the pixel

Anti-aliasing by computing average pixel value

When rendering one triangle, the value of f(x,y) = inside(tri,x,y) averaged over the area of a pixel is equal to the amount of the pixel covered by the triangle.



Summary

- Drawing a triangle = sampling triangle-screen coverage signal
- Pitfall of sampling: aliasing
- Reduce aliasing by prefiltering signal
  - Supersample
  - Reconstruct via convolution (average coverage over pixel)
    - Higher frequencies removed
  - Sample reconstructed signal once per pixel

- There is much, much more to sampling theory and practice…
  - If interested see: Stanford EE261 - The Fourier Transform and its Applications

Consider this task:
viewing a low-resolution image on a high-resolution display

Say we have an image:

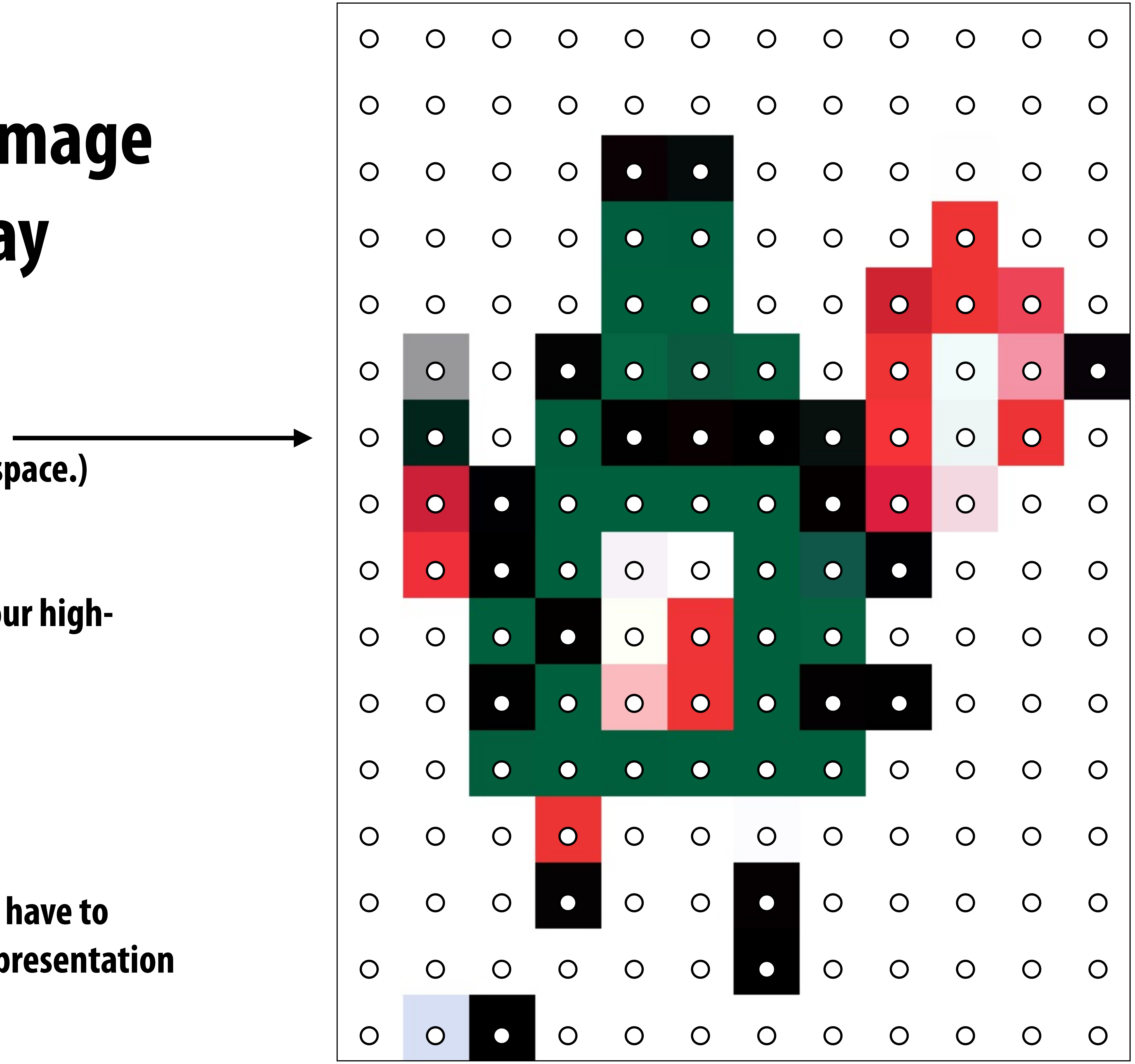
(Which is just a collection of color samples)

Consider this task: viewing a low-resolution image on a high-resolution display

Let's say this is a 12x16 image.
Which means we have 192 samples of a 2D signal.
(The white dots are the sample locations in image space.)

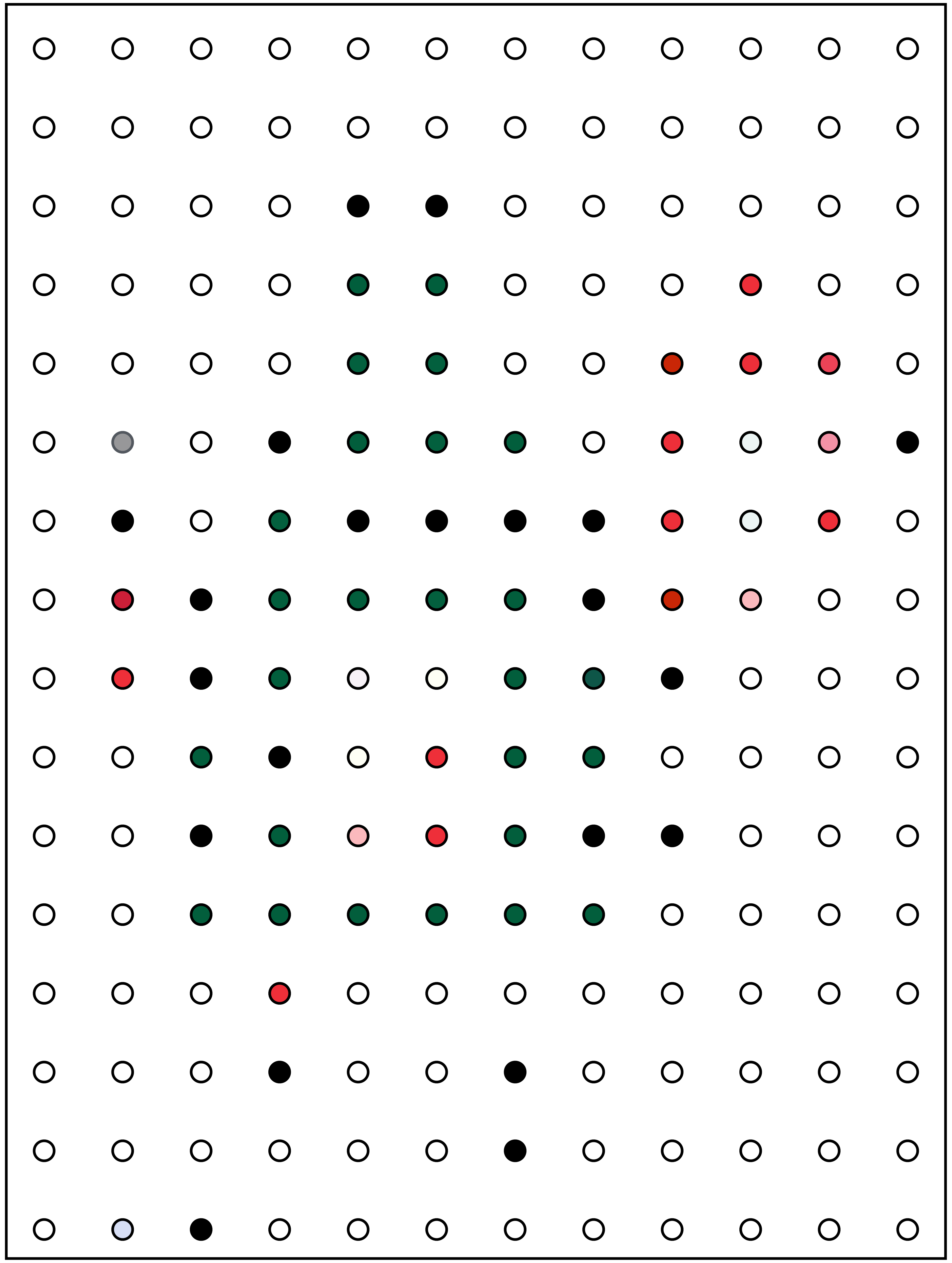
But I'm showing it to you nearly full-slide size on your high-resolution display:
Let's say its taking up 600x800 pixels on screen.

So to render the image in this "zoomed in" view, we have to perform "upsampling": converting a 192-sample representation of a signal to a 480,000-sample representation.



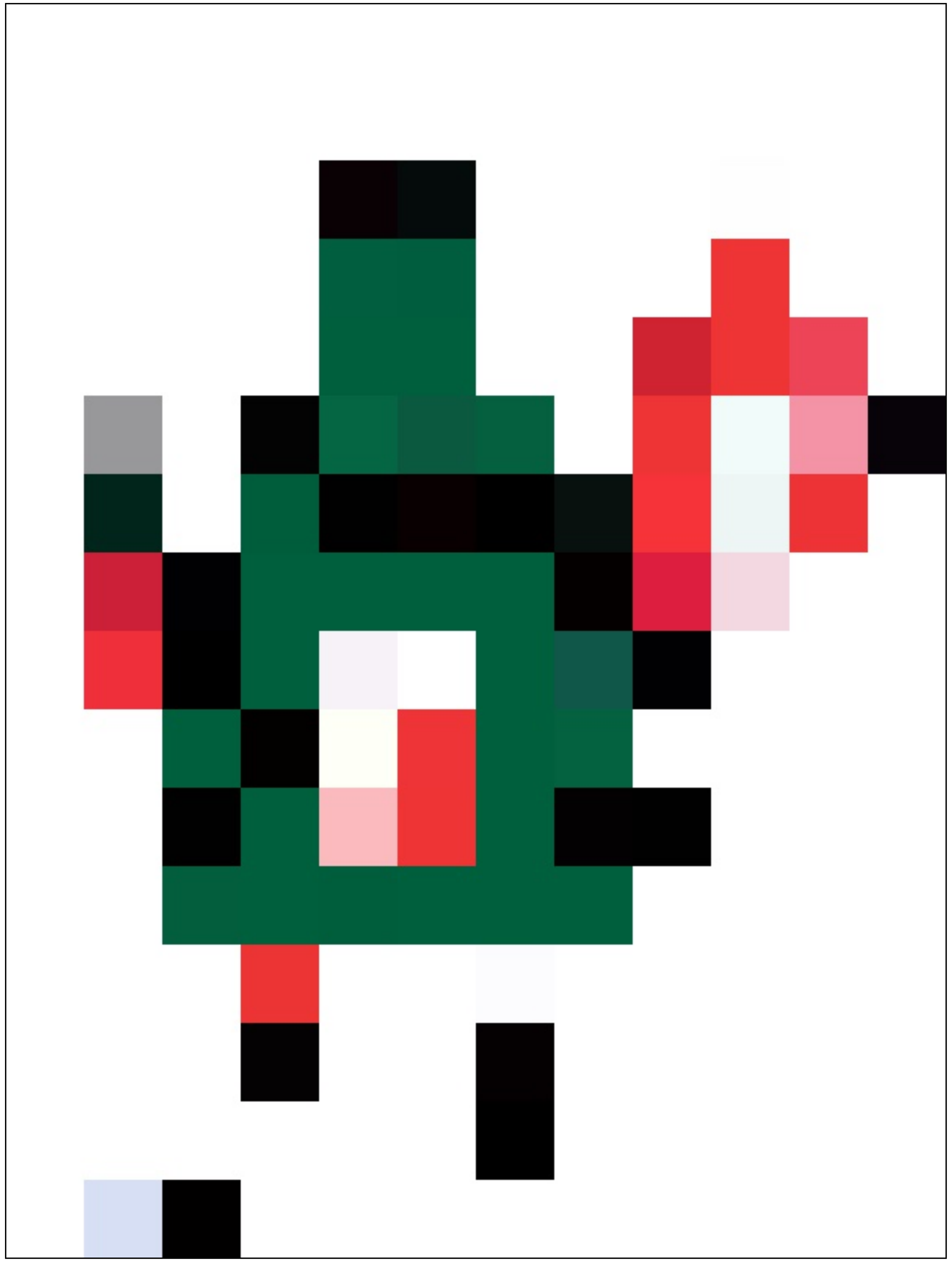
Consider this task:
viewing a low-resolution image on a high-resolution display

Visualization of the 192 samples in the image

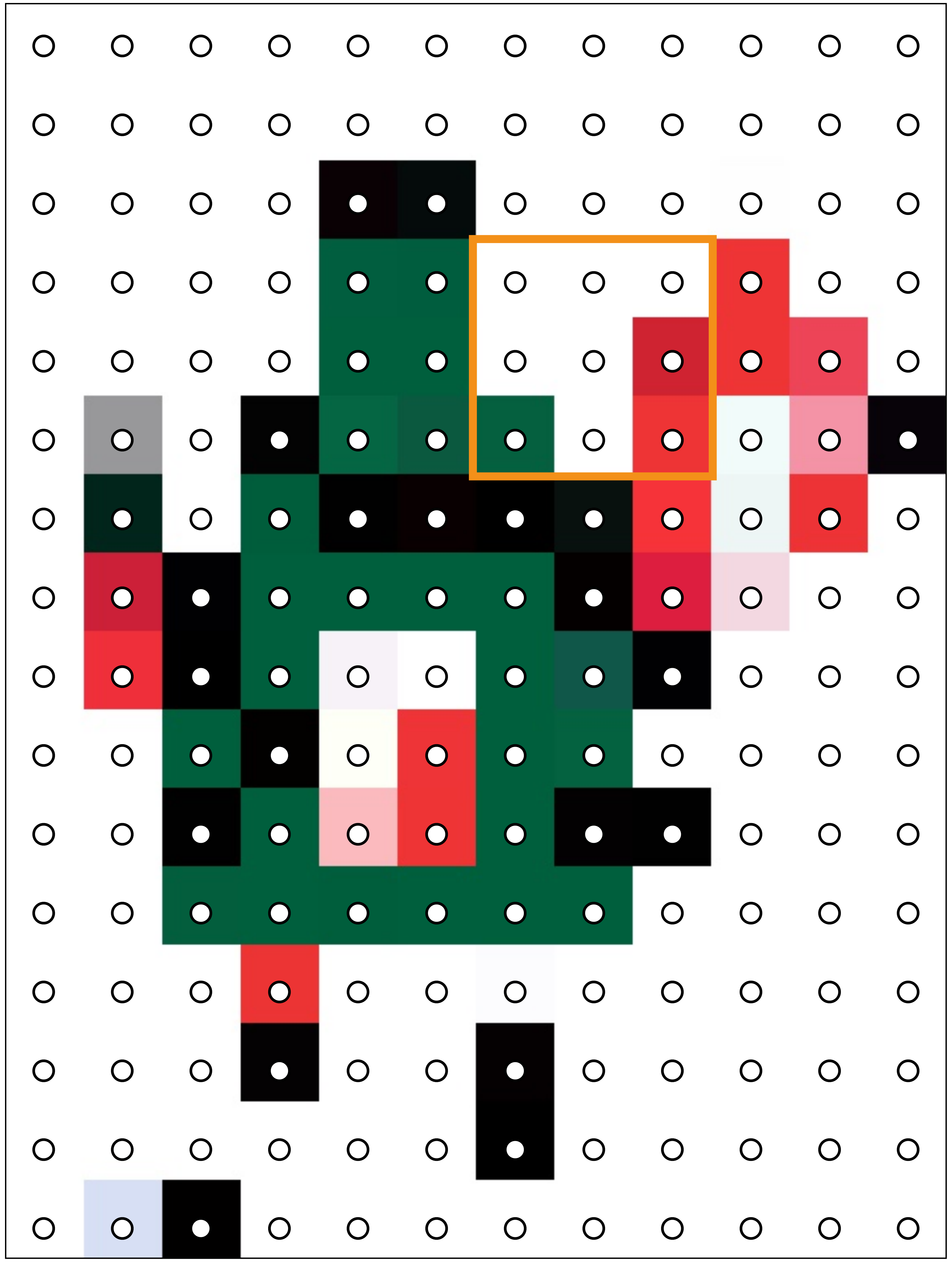


Consider this task: viewing a low-resolution image on a high-resolution display

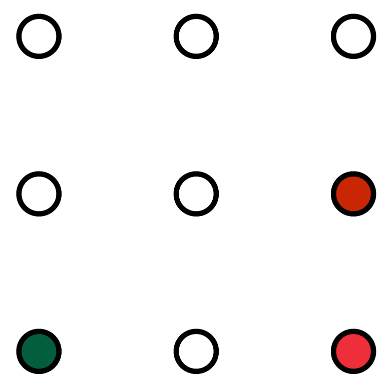
Displaying a new high resolution 600x800 image (480,000 samples) that was created from the original 192 samples



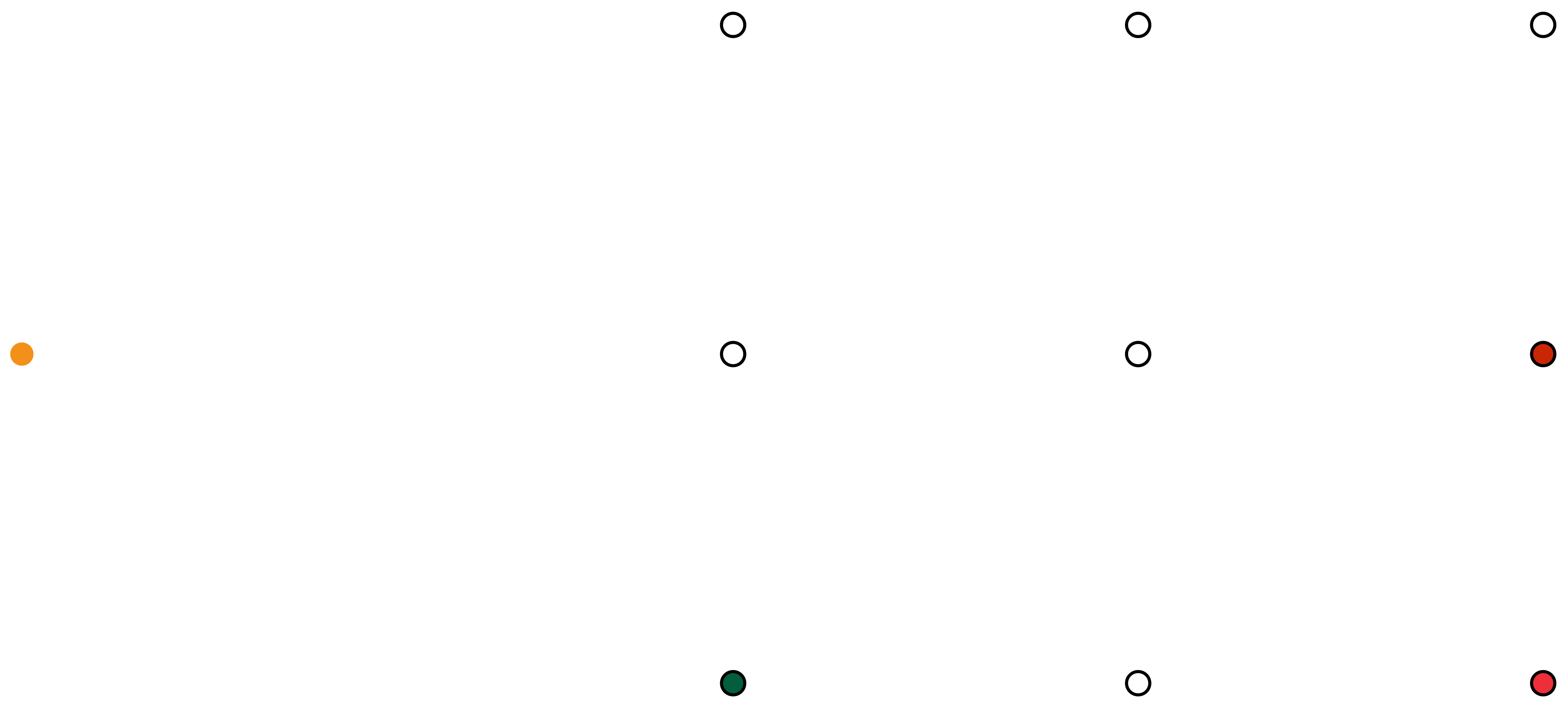
Let's consider the region highlighted in the orange box.



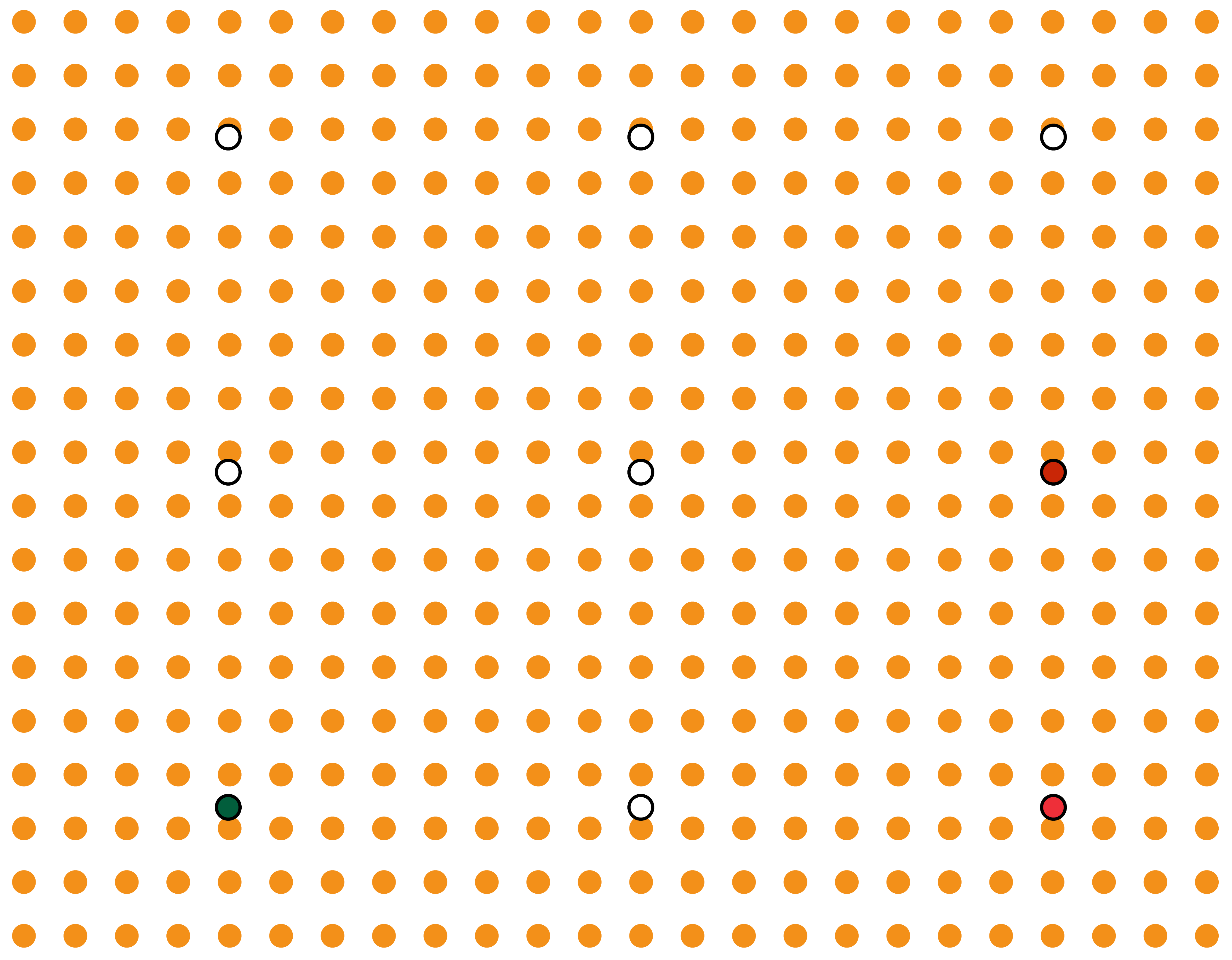
Let's consider the region highlighted in the orange box.



These are the samples from the 12x16 source image.



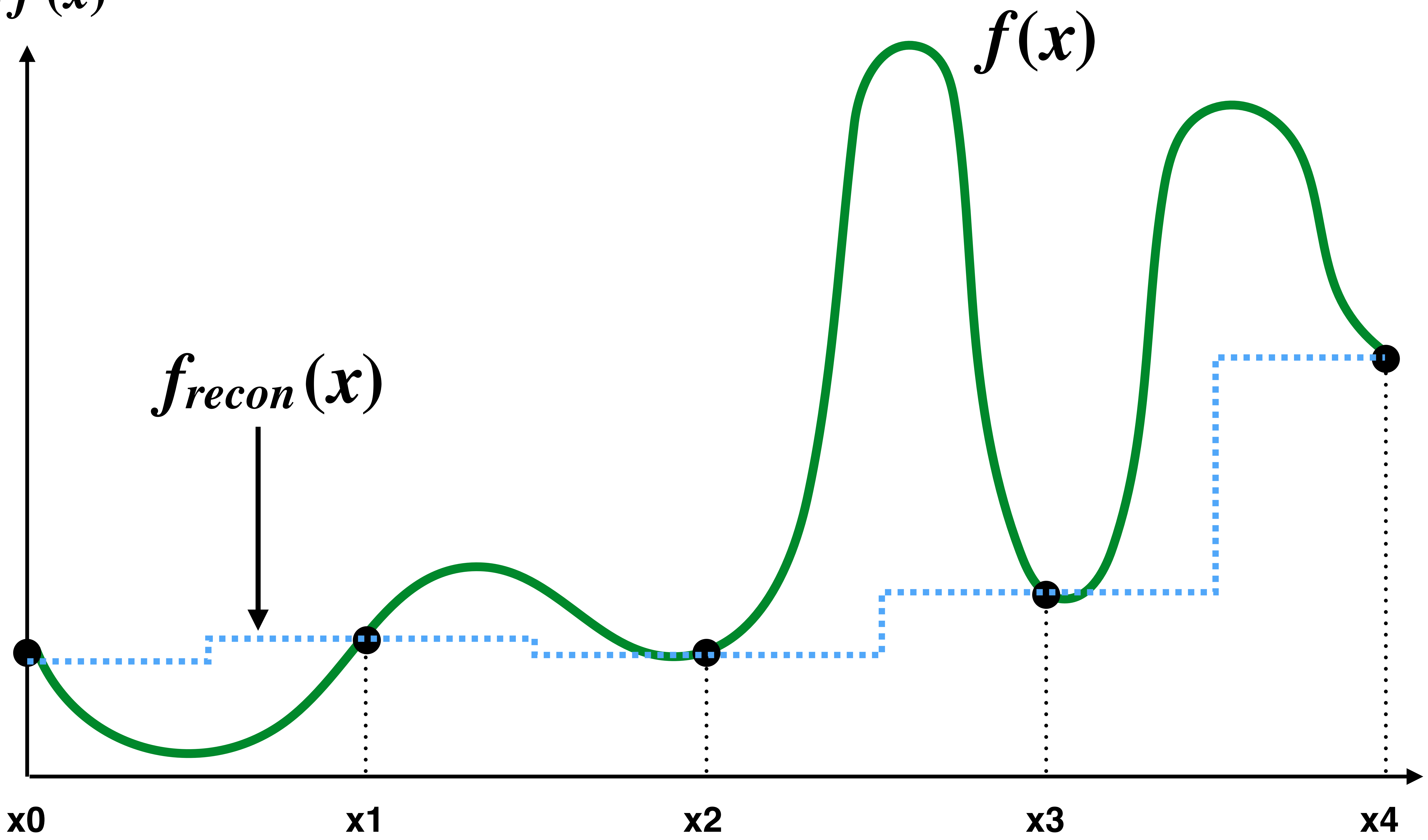
But to render it at high resolution, we need to sample the signal densely.
(At the positions shown by the orange dots)



Recall piecewise-constant reconstruction

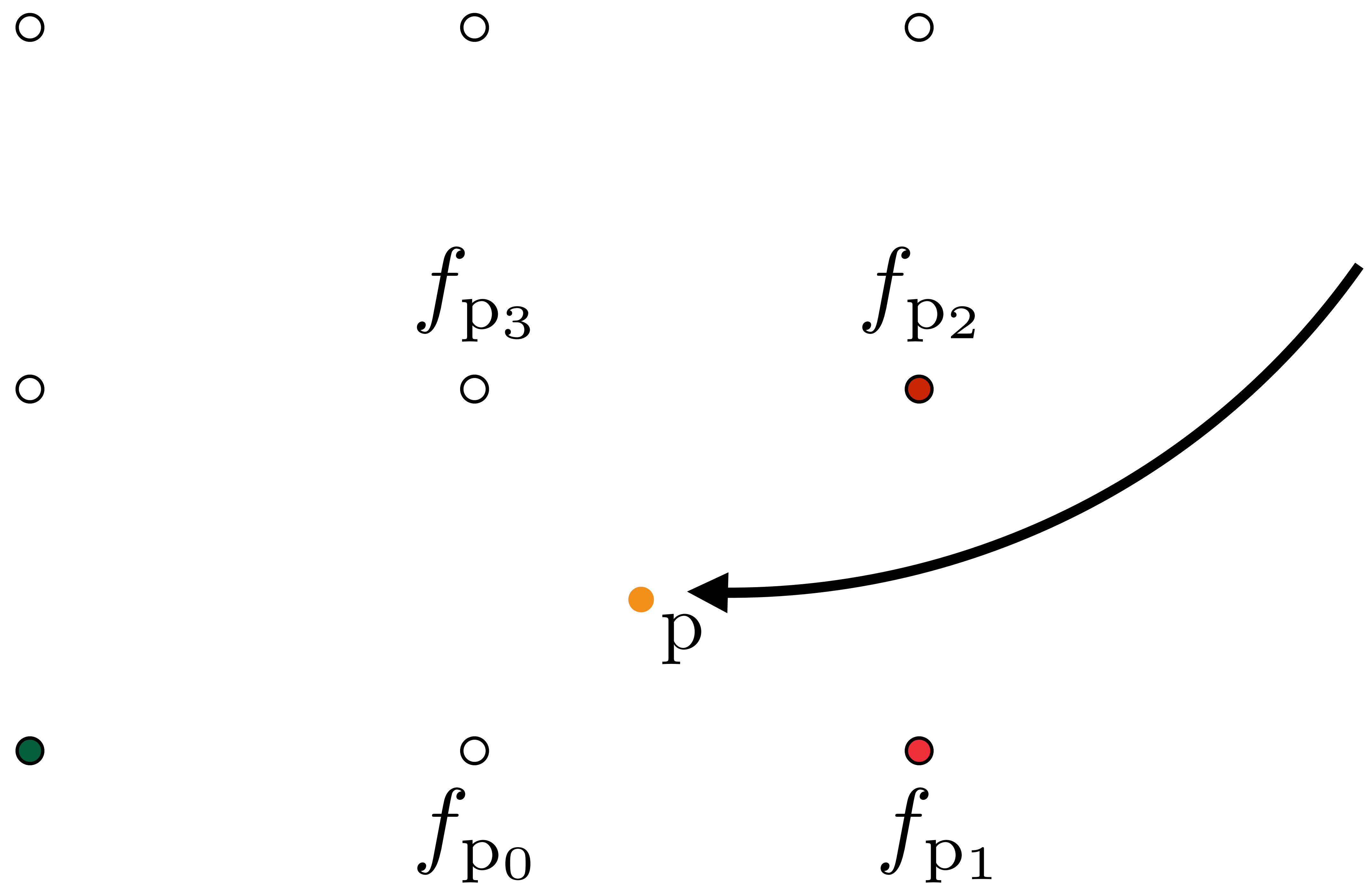
$f_{recon}(x)$ = value of sample closest to x

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



= reconstruction via piece-wise constant interpolation (nearest neighbor)

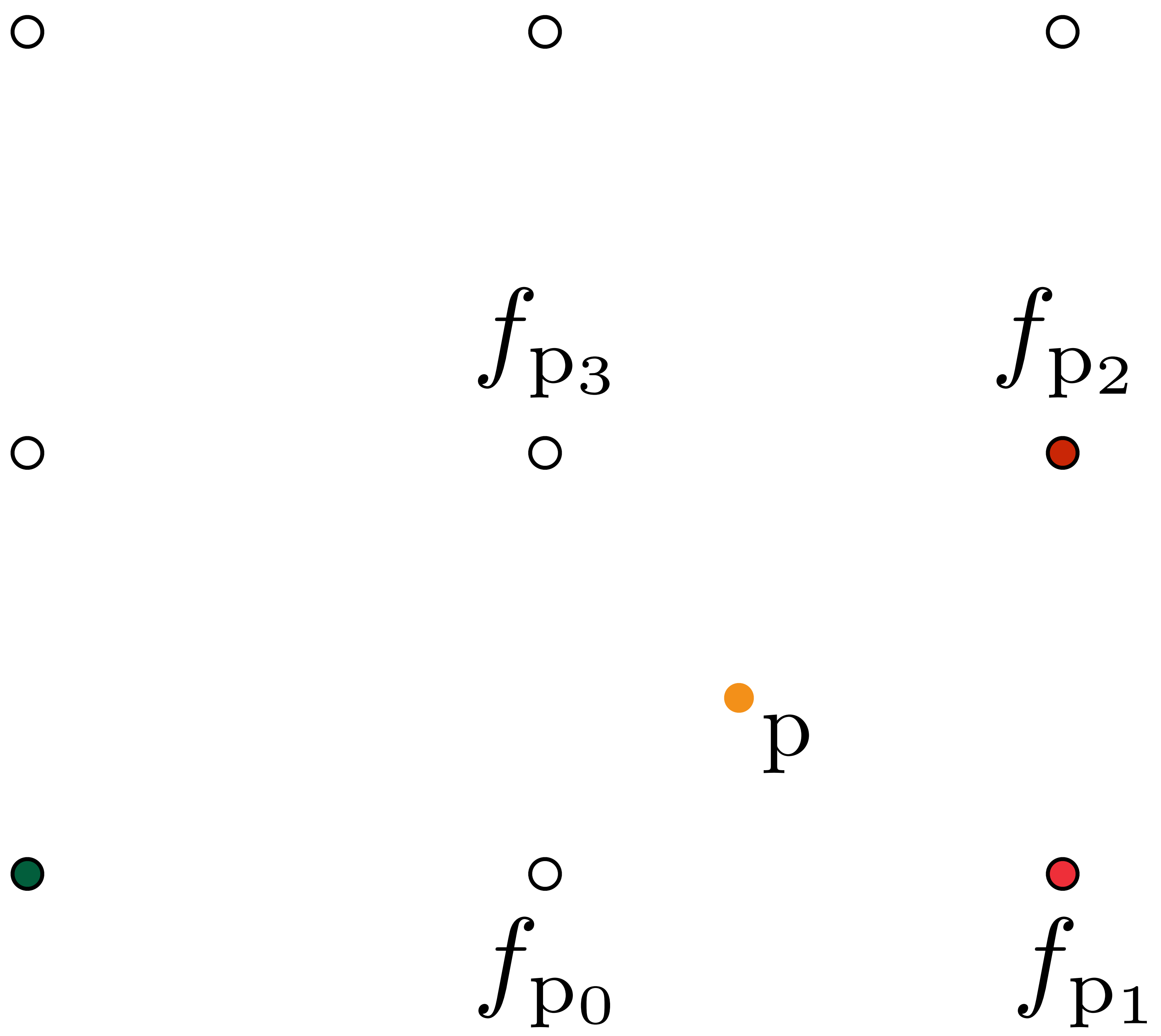
But to render it at high resolution, we need to sample the signal densely.
(At the positions shown by the orange dots)



Let's say we want to reconstruct the signal = f(x,y) at this point *p*.

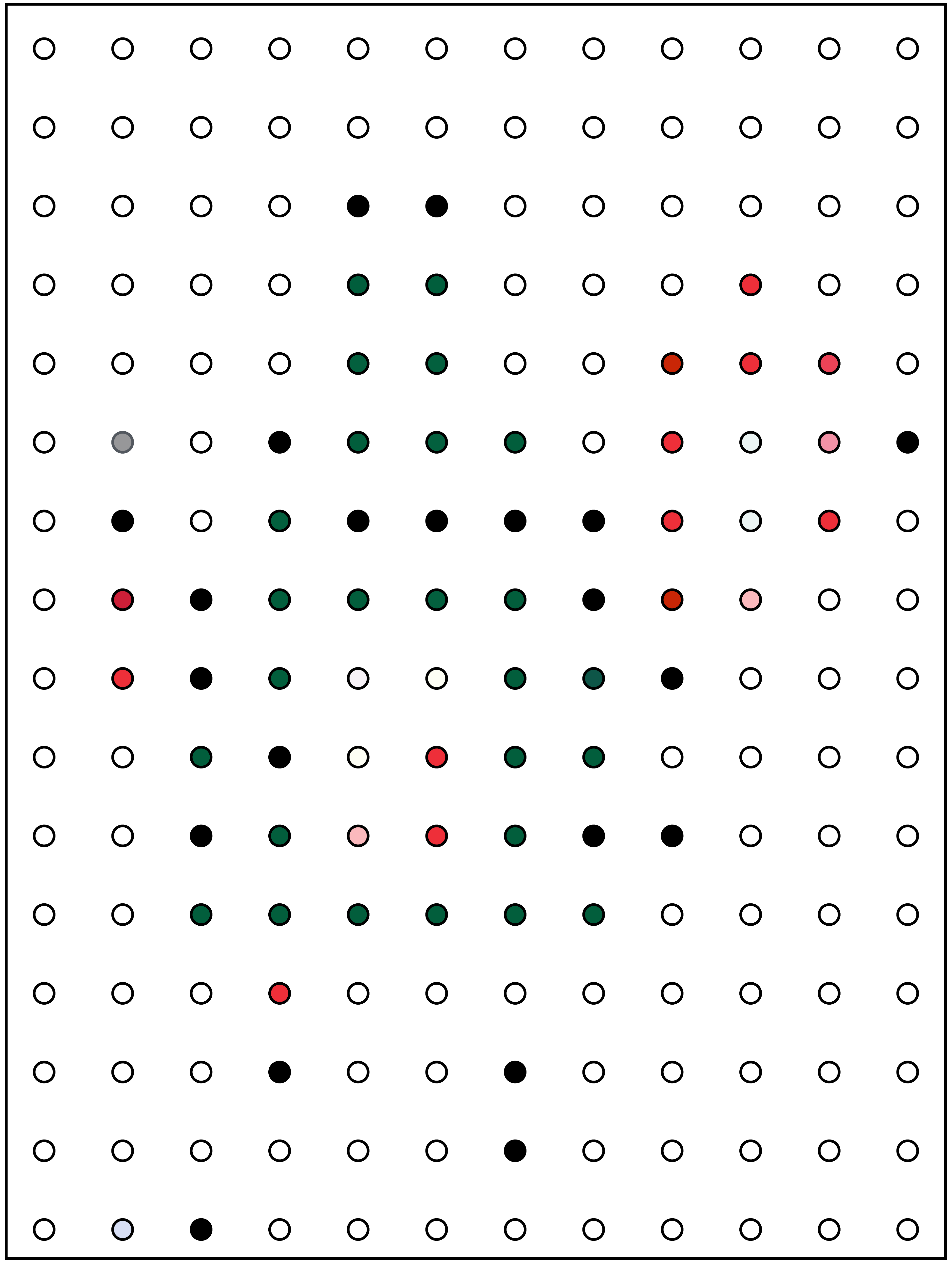
Given the sampled values at known sample points.

What is the piecewise-constant reconstruction of the signal at the orange dot?

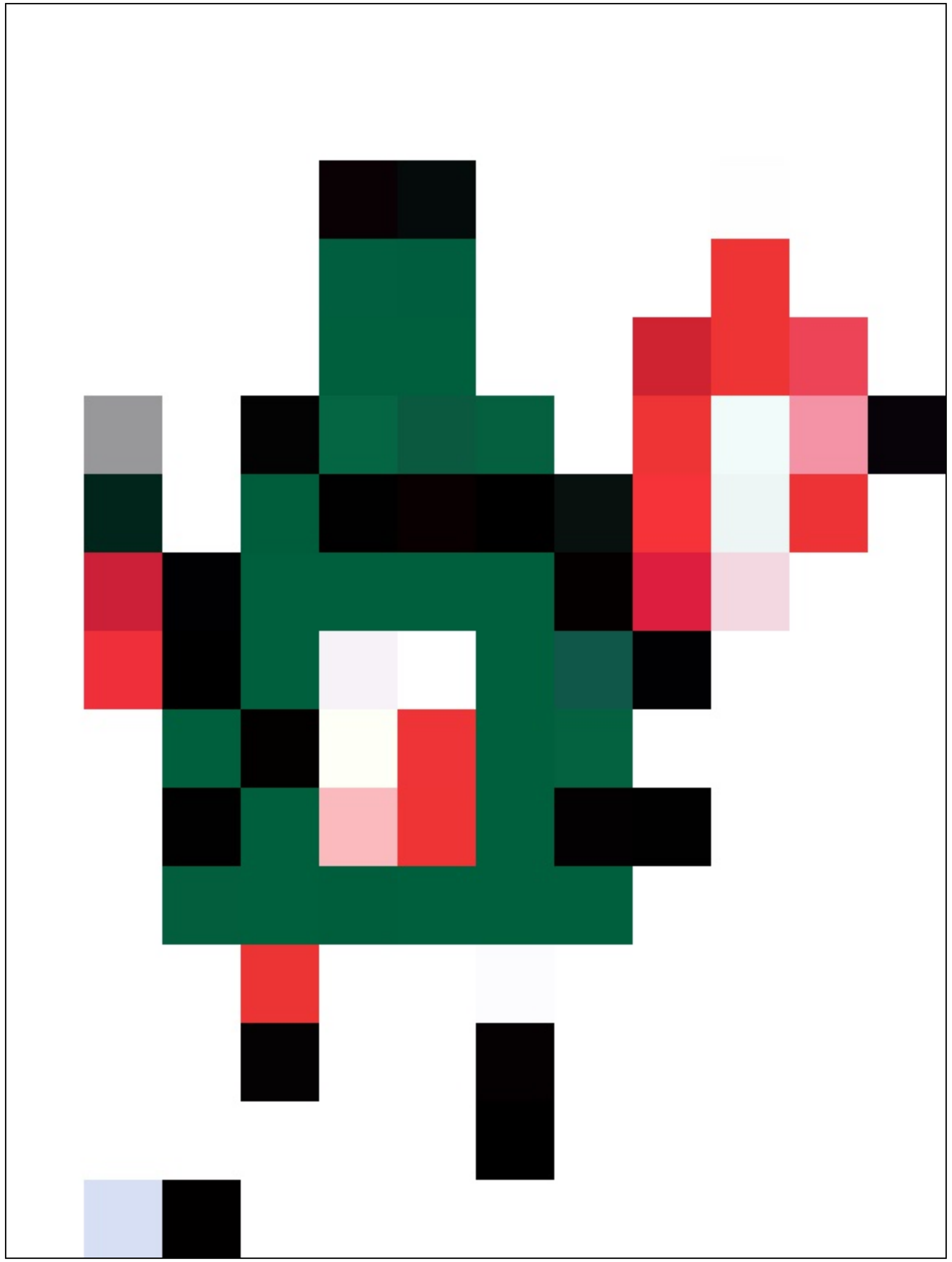


Answer: white

So if these are the input sample locations and values.

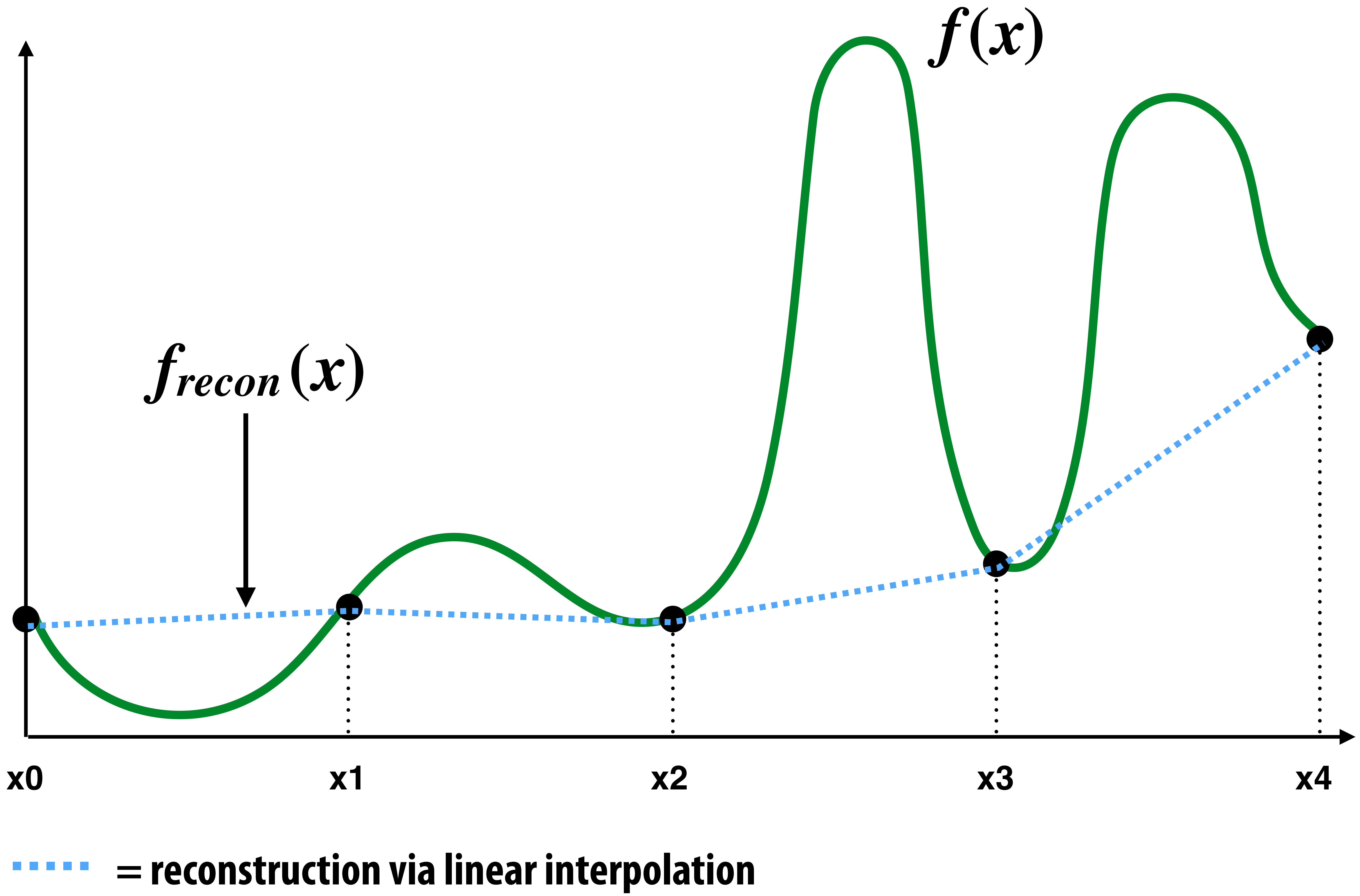


This is the piecewise constant (closest sample) reconstruction



Recall: piecewise linear reconstruction

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

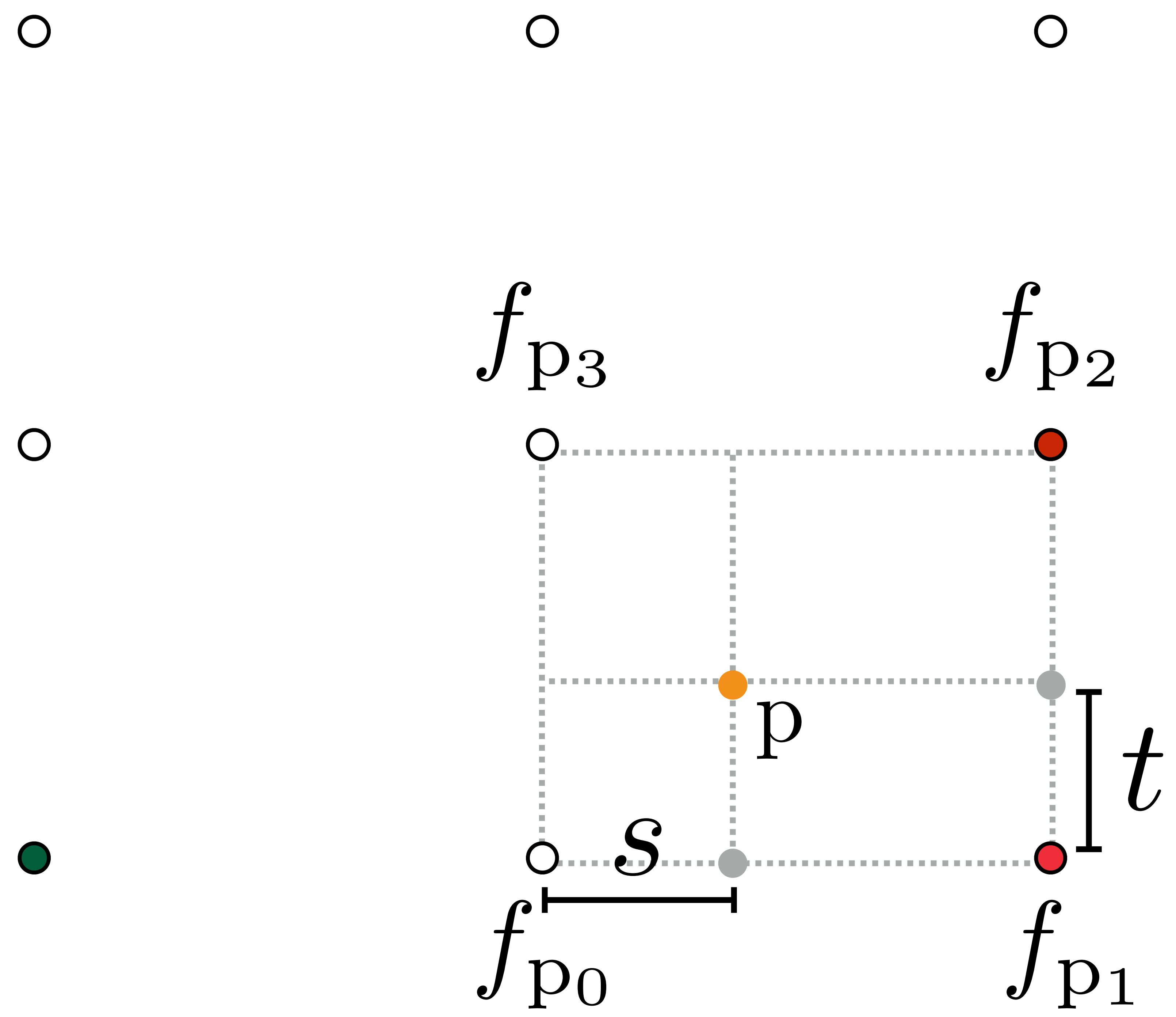


Bilinar interpolation

Compute fractional offsets (between samples)

$$s = \frac{\mathrm{p_x} - \mathrm{p_{0_x}}}{\mathrm{p_{1_x}} - \mathrm{p_{0_x}}}$$

$$t = \frac{\mathrm{p_y} - \mathrm{p_{0_y}}}{\mathrm{p_{3_y}} - \mathrm{p_{0_y}}}$$



Recall linear interpolation (1D)

$$\mathrm{lerp}(x, v_0, v_1) = v_0 + x(v_1 - v_0)$$

Two helper lerps (horizontal):

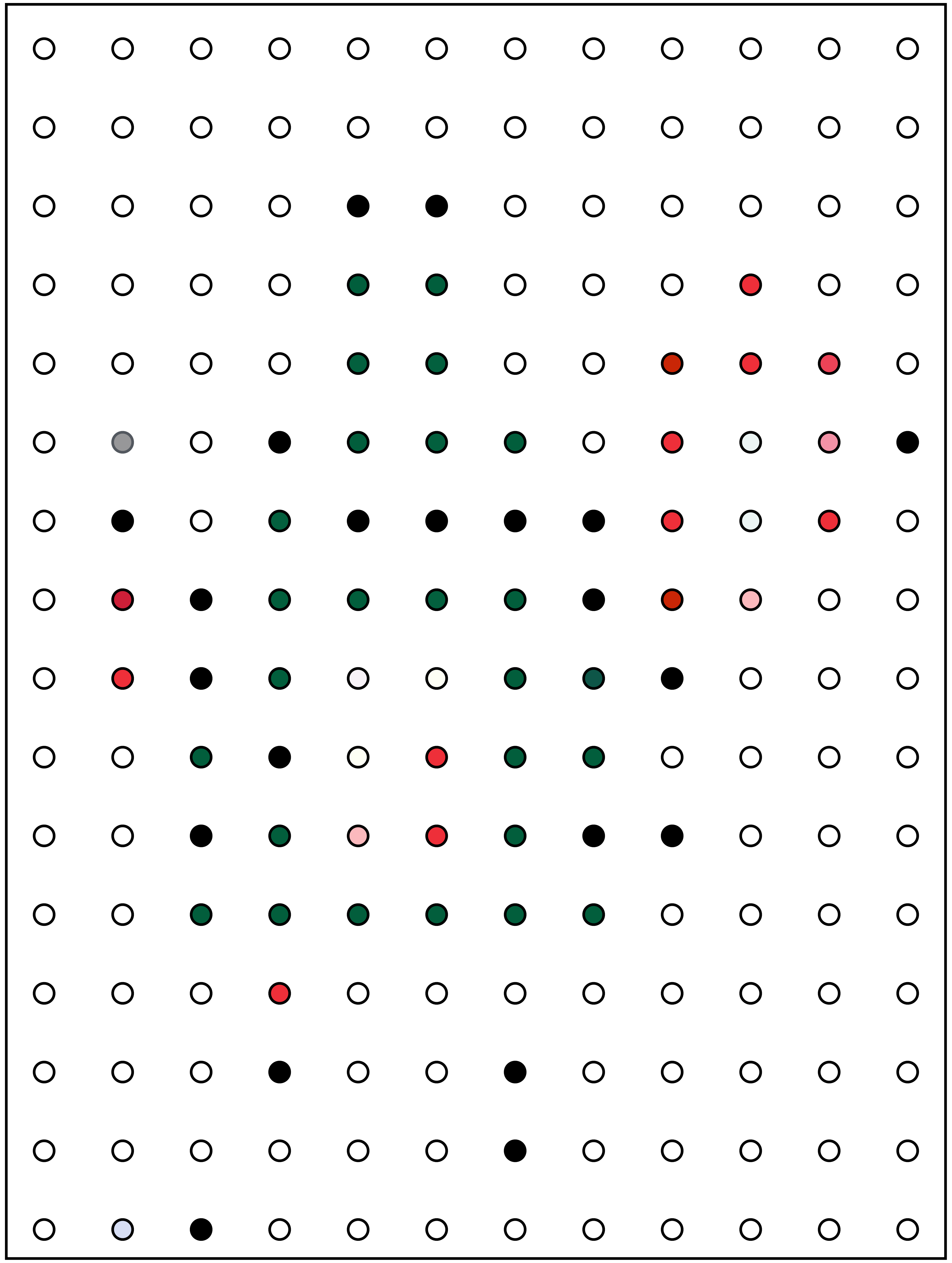
$$a = \mathrm{lerp}(s, f_{\mathrm{p_0}}, f_{\mathrm{p_1}})$$

$$b = \mathrm{lerp}(s, f_{\mathrm{p_3}}, f_{\mathrm{p_2}})$$

Final vertical lerp, to get result:

$$f_{\mathrm{p}} = \mathrm{lerp}(t, a, b)$$

So if these are the input sample locations and values.



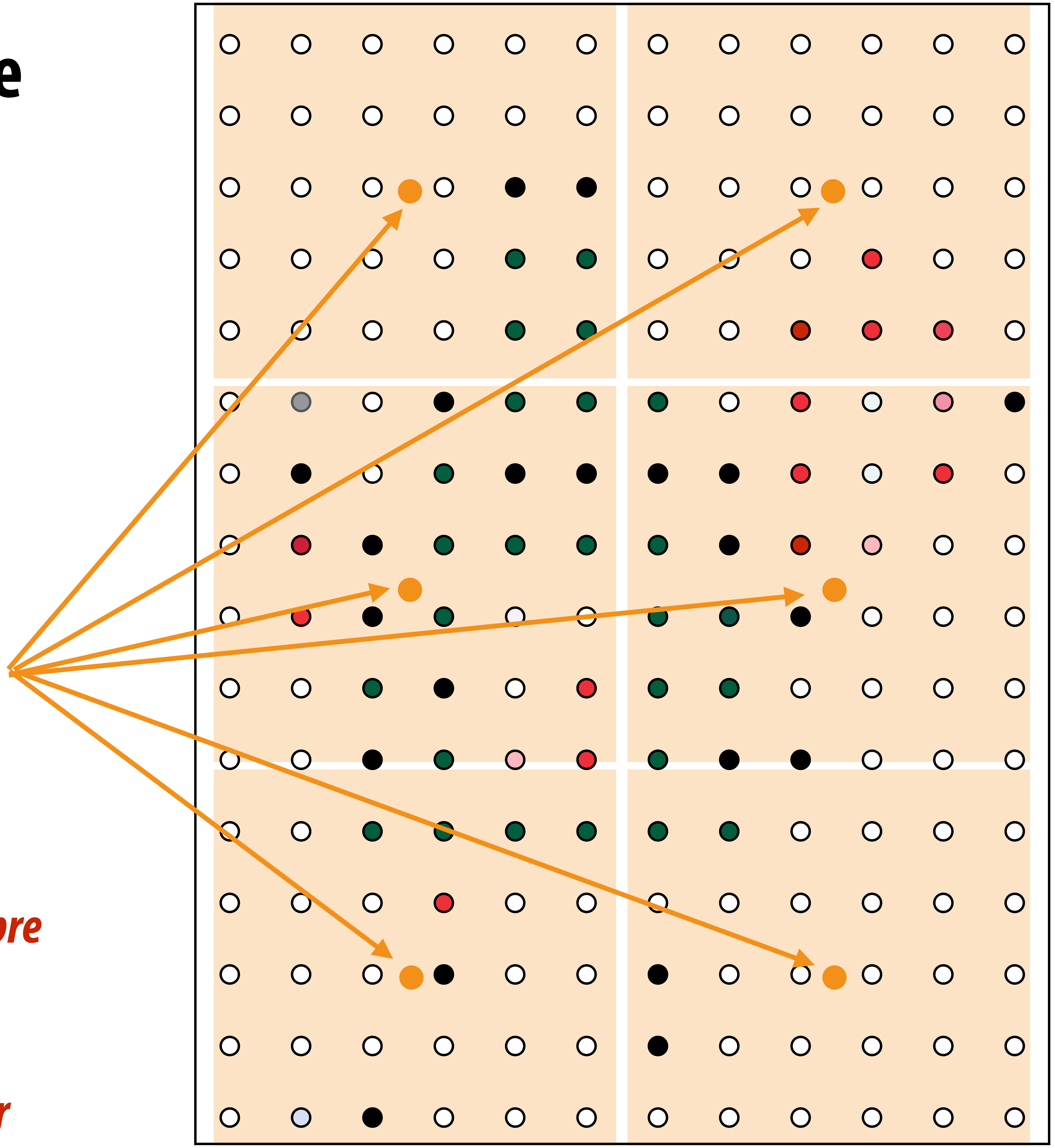
This is the reconstruction from bilinear interpolation

This is the reconstruction from bicubic interpolation

(even higher order interpolation)

Consider this task:
Downsampling a high-resolution image to a low-resolution one

Assume these are the input sample locations and values in the high-resolution source image…



We need to resample f(x,y) at the sample locations for the low-resolution image (orange dots)

To avoid aliasing due to coarse sampling, we need to pre filter the source image prior to sampling.
(Remember: convolution!)
Orange shaded regions figure shows convolution filter window sizes needed.

Summary: resampling a signal

- Resampling = converting one set of samples of a signal to another
- Requires reconstructing approximation of value of signal at new points in the domain
- Upsampling: requires interpolation of input samples
- Downsampling: requires filtering of reconstructed signal prior to resampling to avoid aliasing

Acknowledgements

■ Thanks to Ren Ng, Pat Hanrahan, Keenan Crane for slide materials