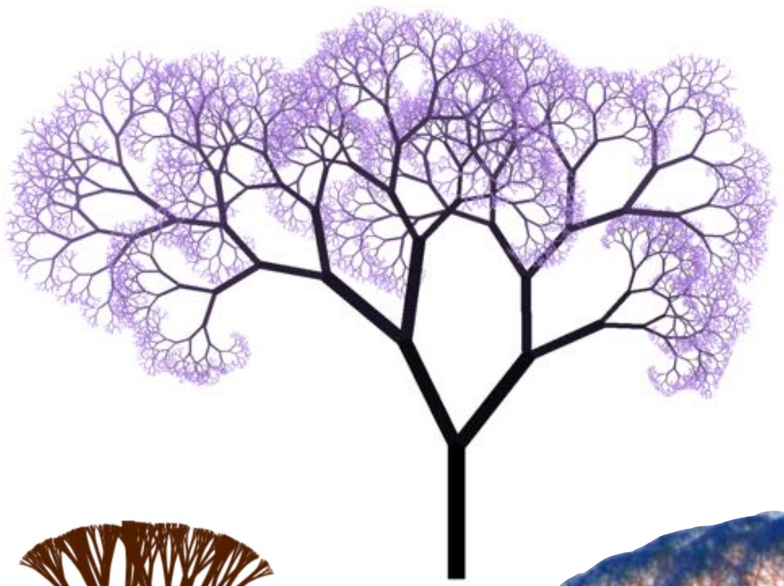
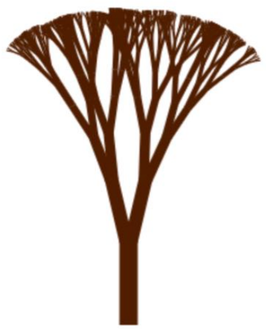
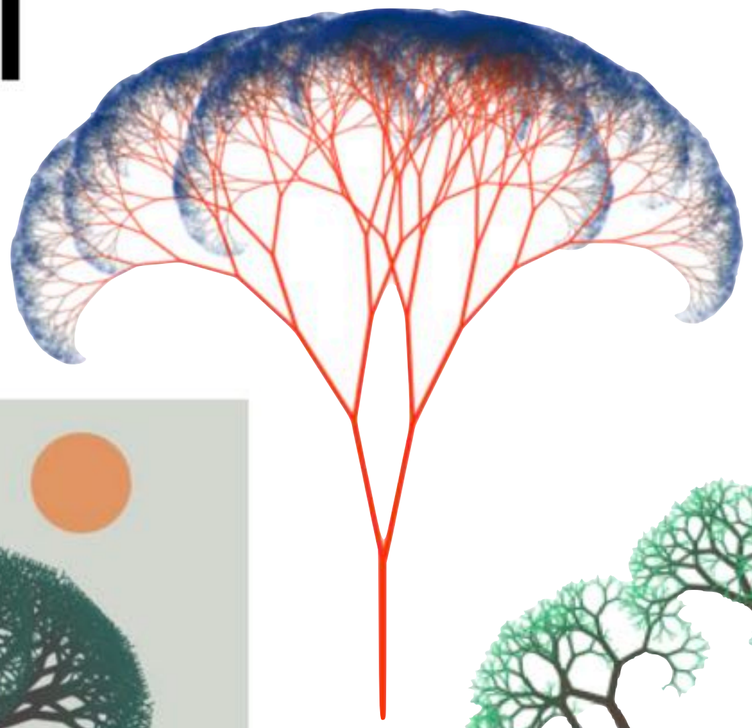
Texture Mapping and Supersampling

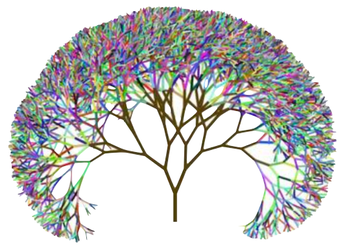
Computer Graphics
CMU 15-462/15-662

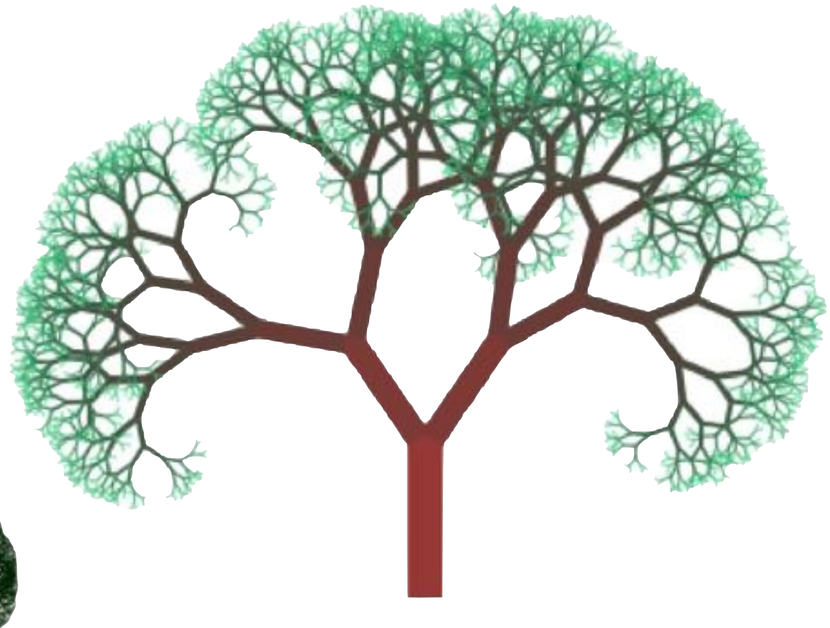
MiniHW2: awesome pictures!

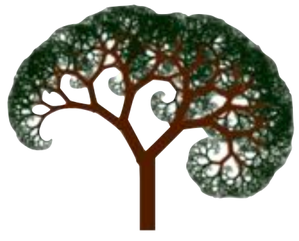


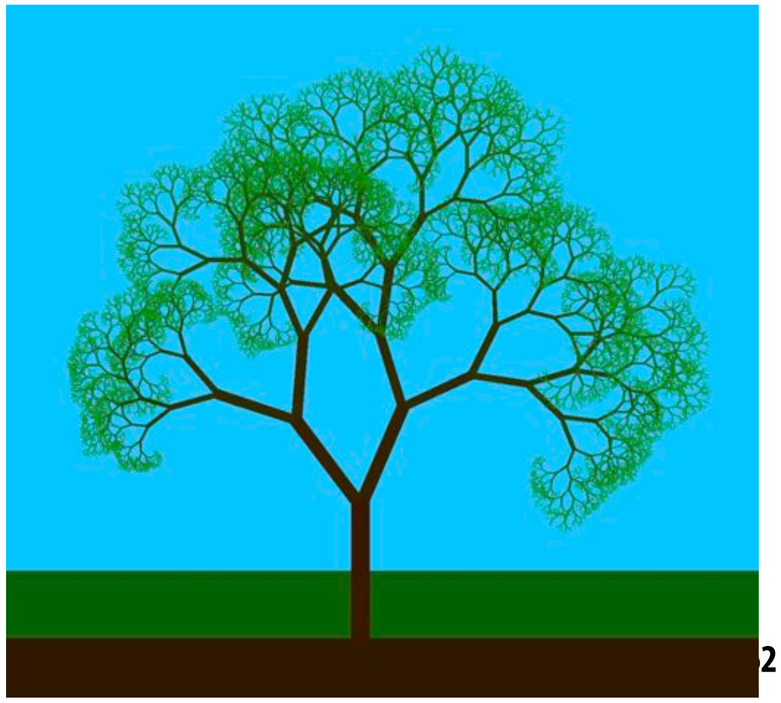




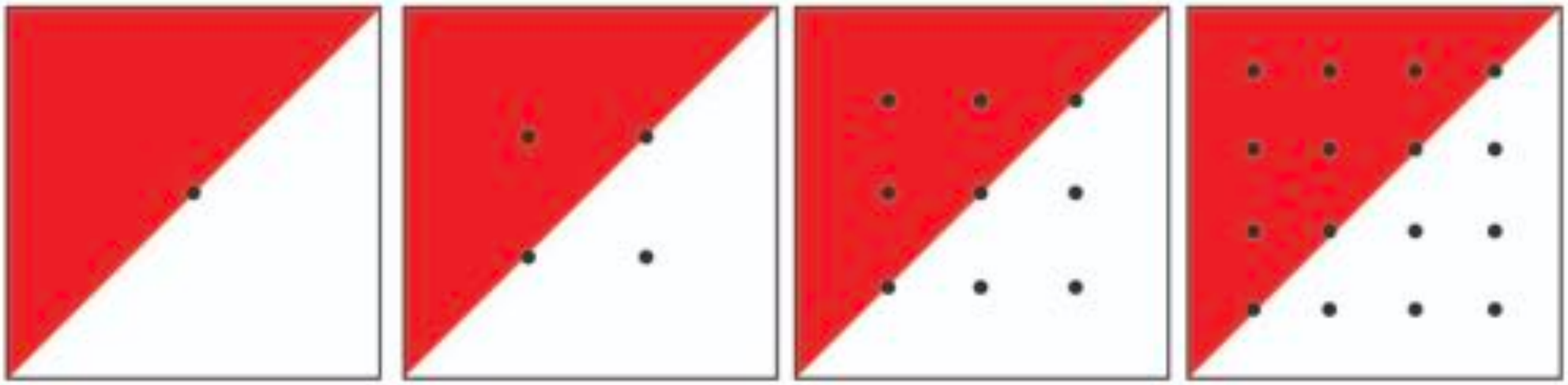








MiniHW3 — due before class Monday 2/13



A1.5 — due 11:59pm Monday 2/13

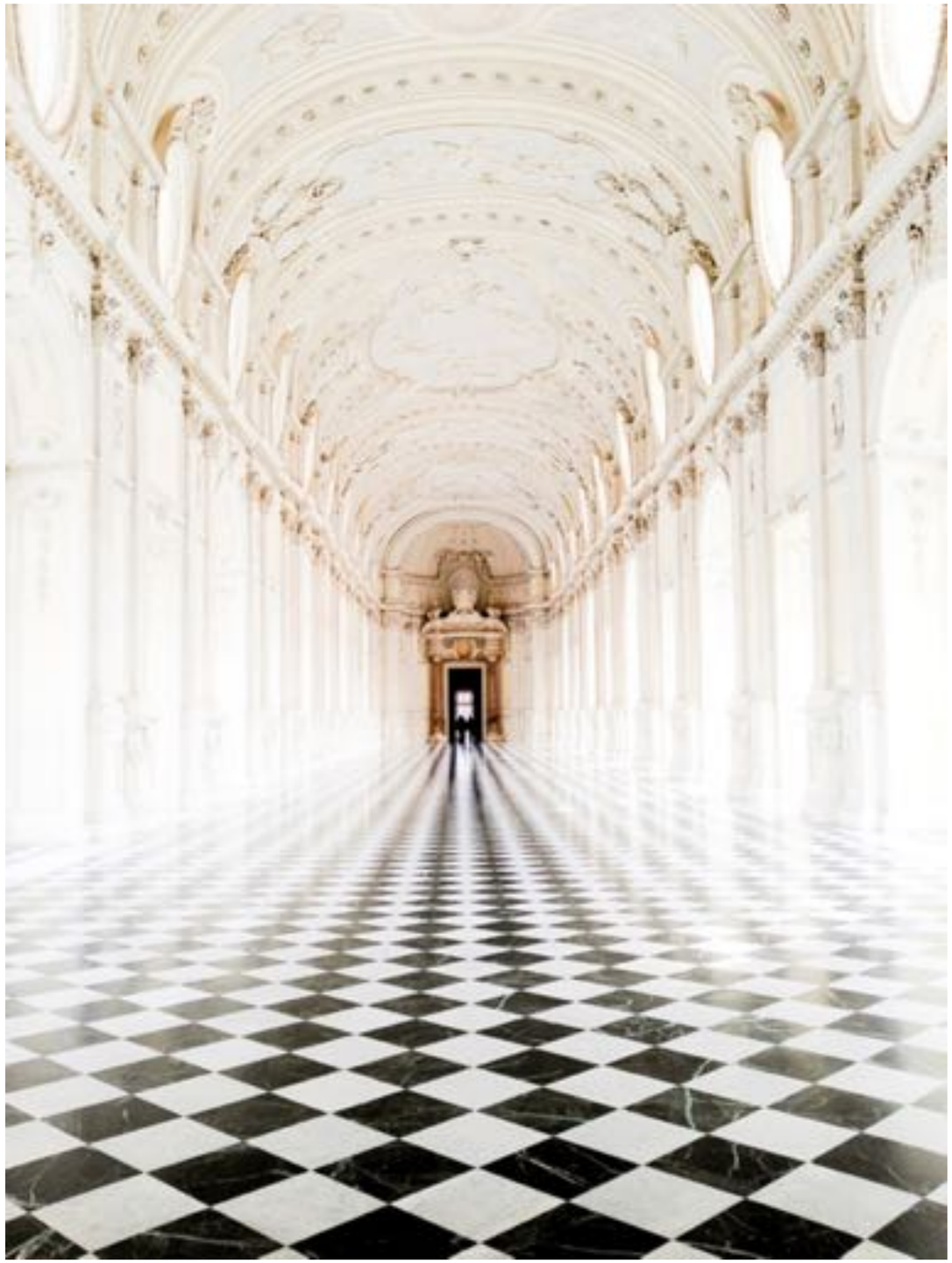
- Checkpoint A1.5 [60pts]:
  - A1T5 interpolation [20pts]
    - screen-space [4pts]
    - perspective-correct [6pts]
    - derivatives [10pts]
  - A1T6 mip-mapping [18pts]
    - sampling [8pts]
    - generation [8pts]
    - lod [2pts]
  - A1T7 supersampling [12pts]
    - storage [2pts]
    - for (samples) [5pts]
    - resolve [5pts]
  - writeup-A1.txt [2pts]
  - render.png + render.js3d [8pts]

covered in class today (and maybe a little bit of Wednesday)

Texture Mapping and Supersampling

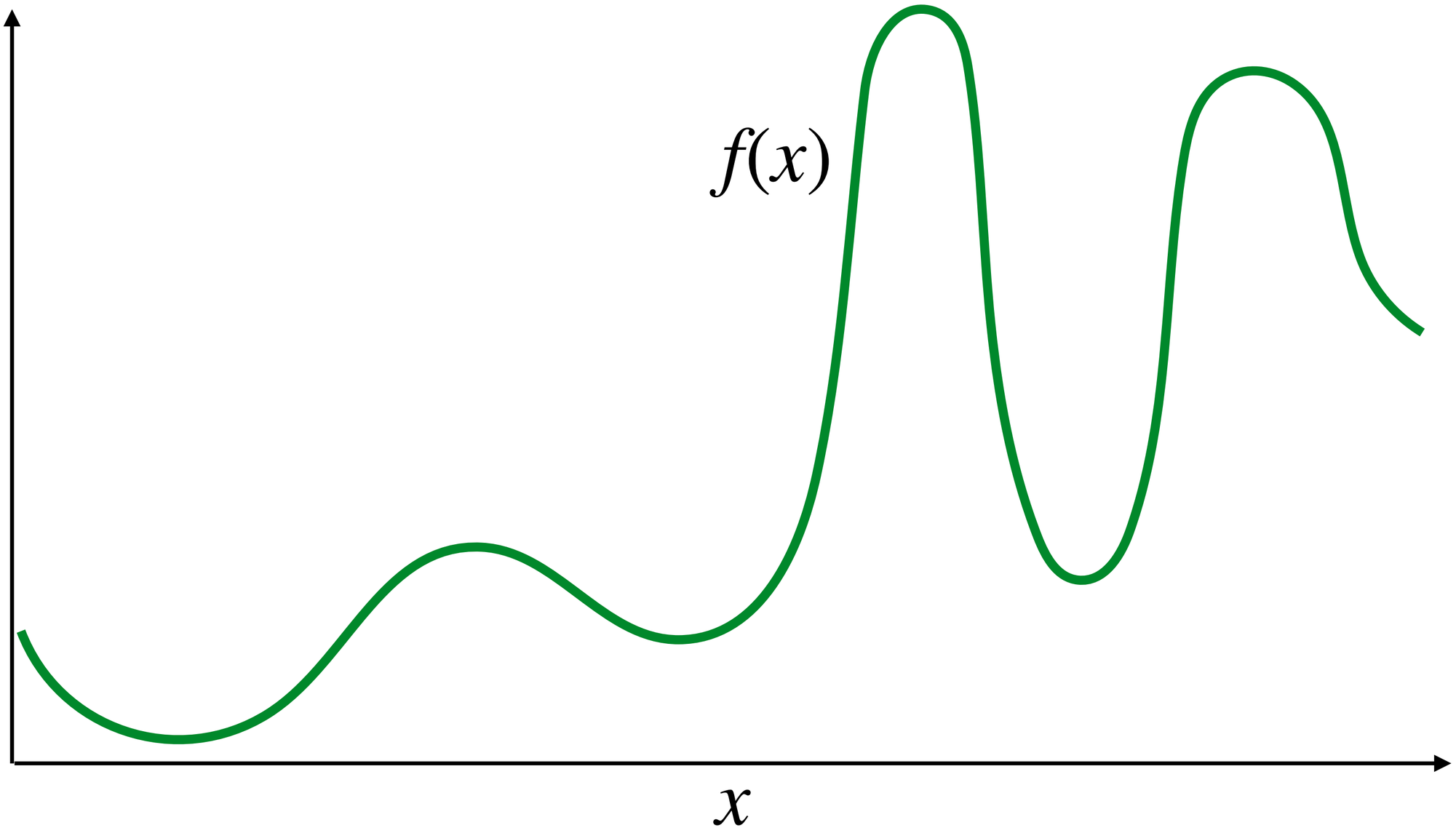
Overall goal — display nice looking textures on objects in our scene, both for closeups and for distance shots!

- **Part I:**
  - Sampling, aliasing, and supersampling
- **Part II:**
  - Perspective correct interpolation (or .. using barycentric coordinates properly)
- **Part III:**
  - Texture mapping
  - Upsampling and downsampling with the mipmap



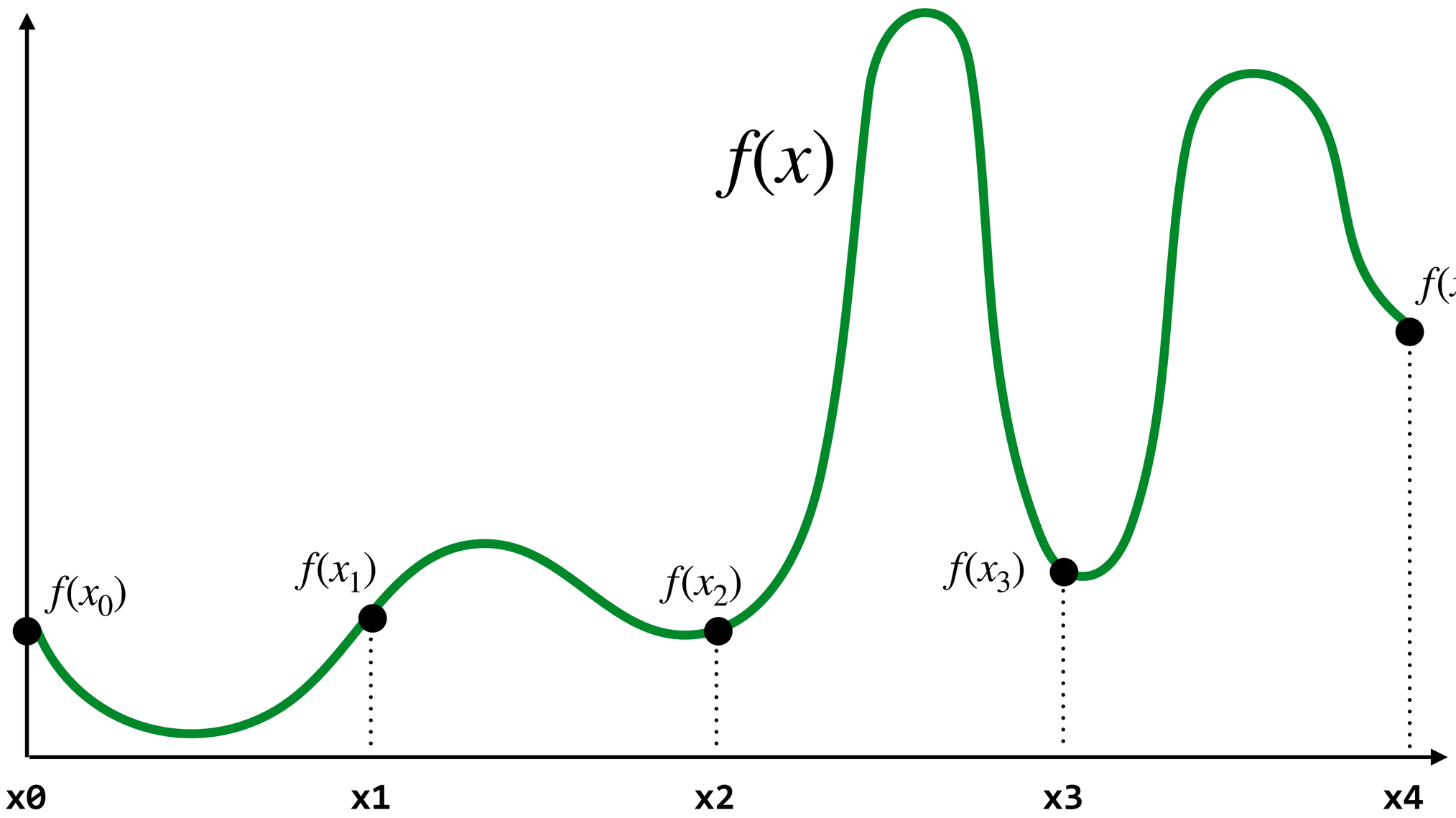
Part I: Sampling, Aliasing, and Supersampling

Sampling 101: Sampling a 1D signal

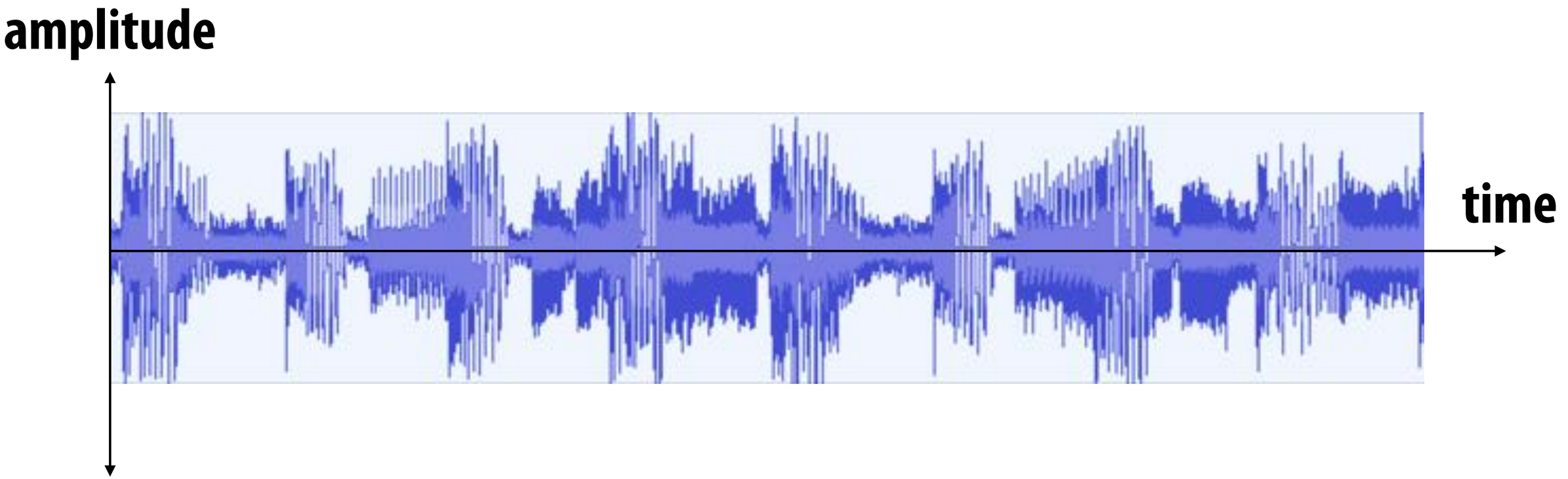


Sampling = taking measurements of a signal

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

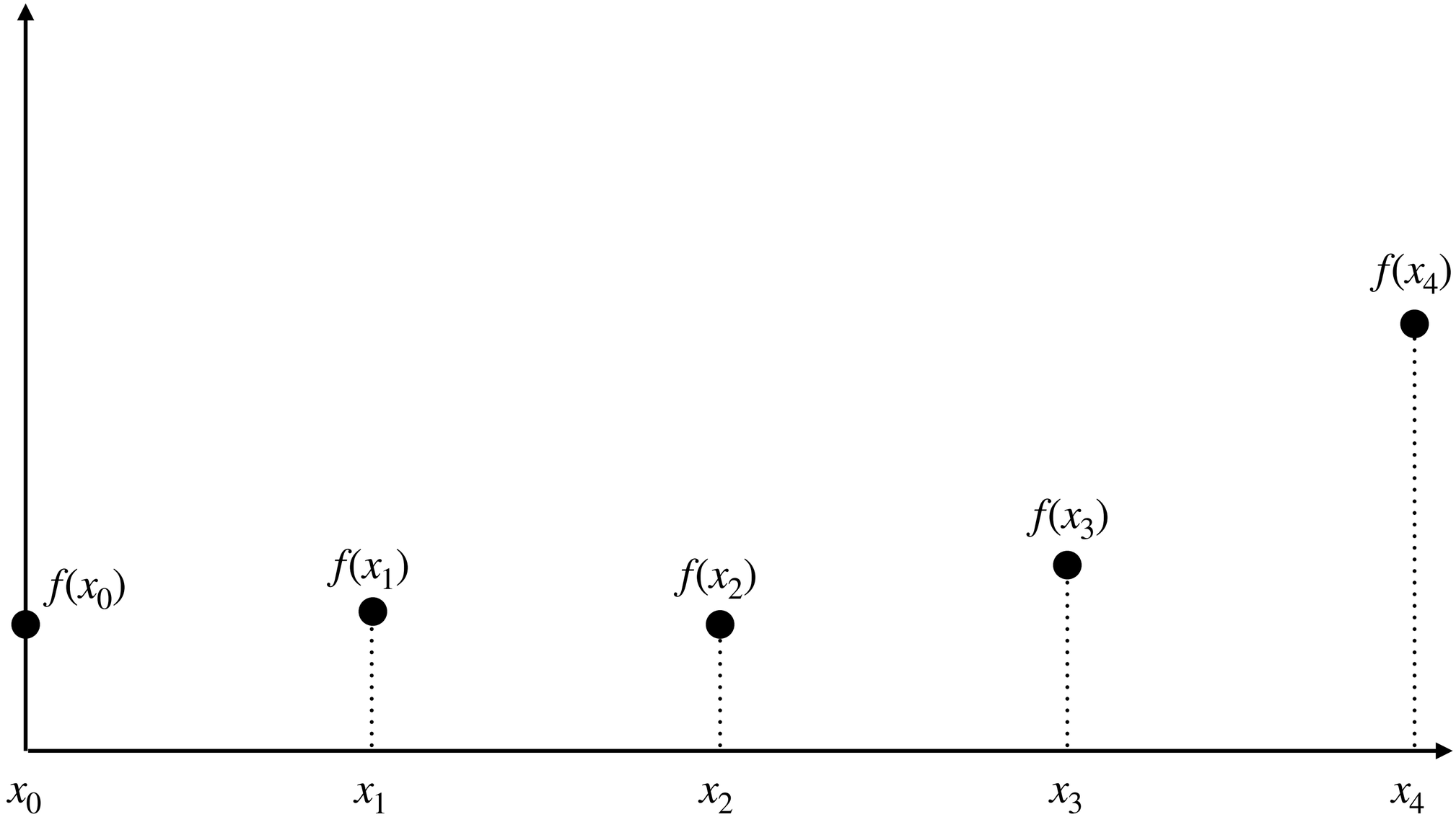


Audio file: stores samples of a 1D signal



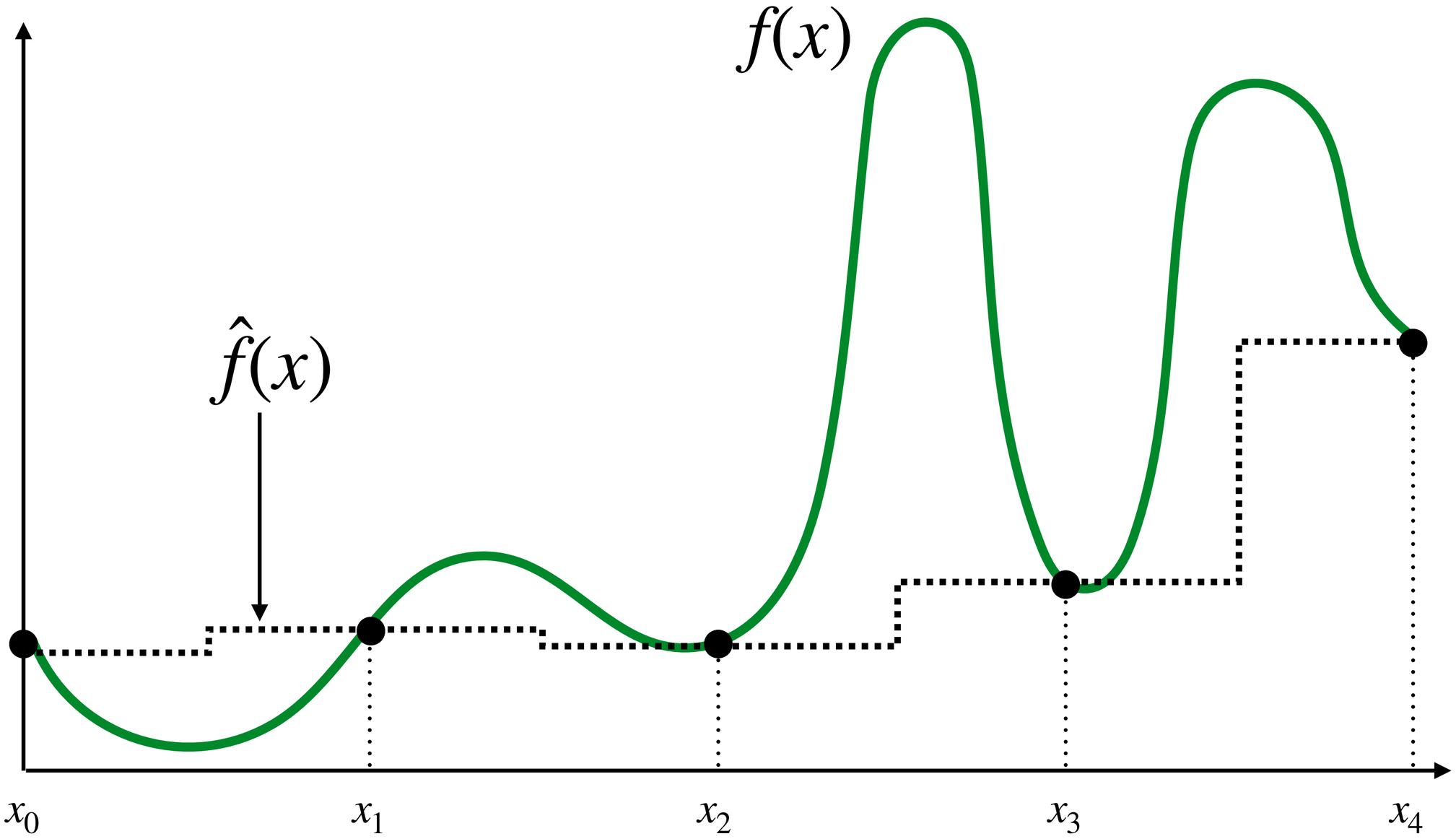
(most consumer audio is sampled 44,100 times per second, i.e., at 44.1 KHz)

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



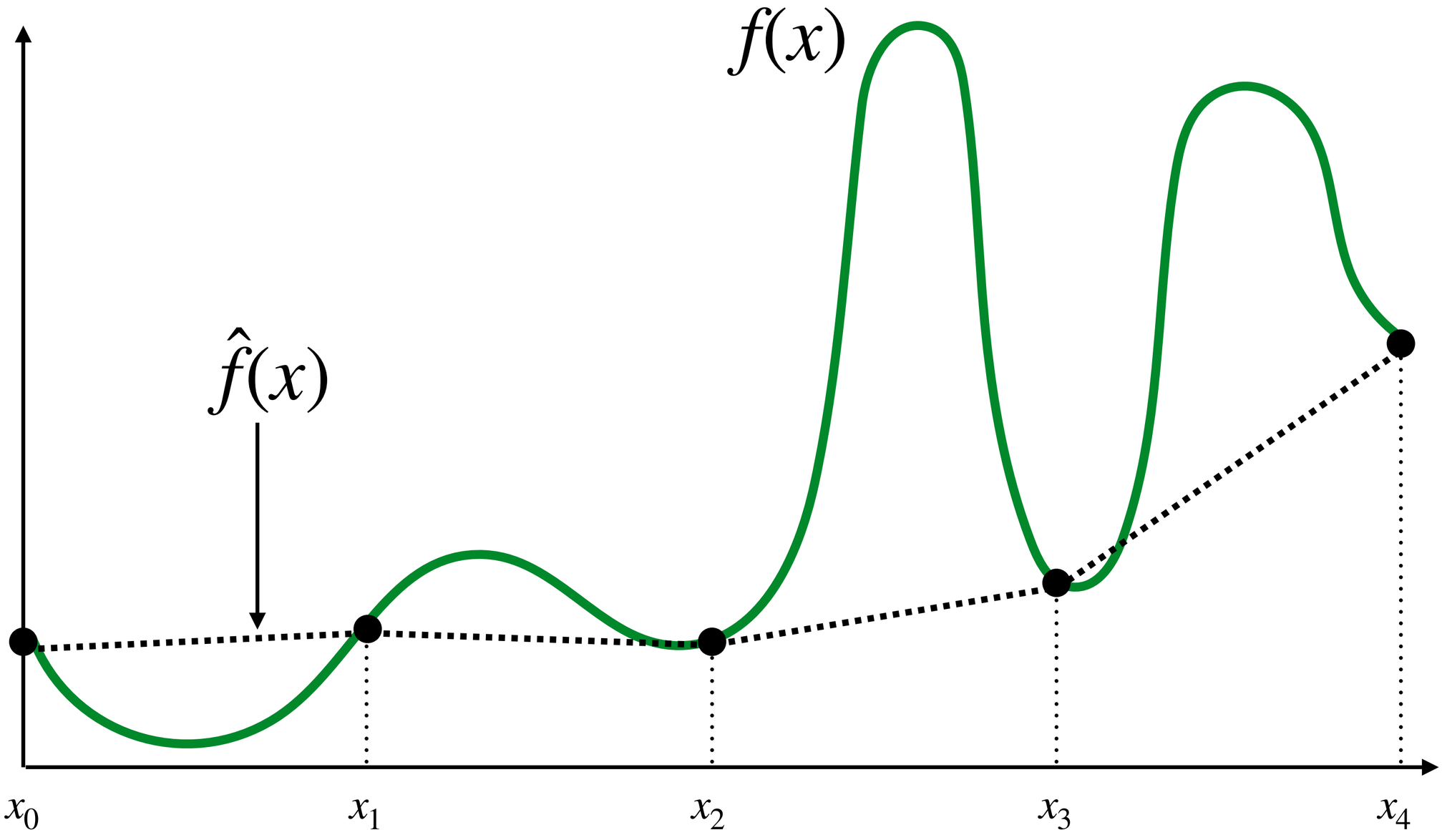
Piecewise constant approximation

$\hat{f}(x)$ = value of sample closest to x



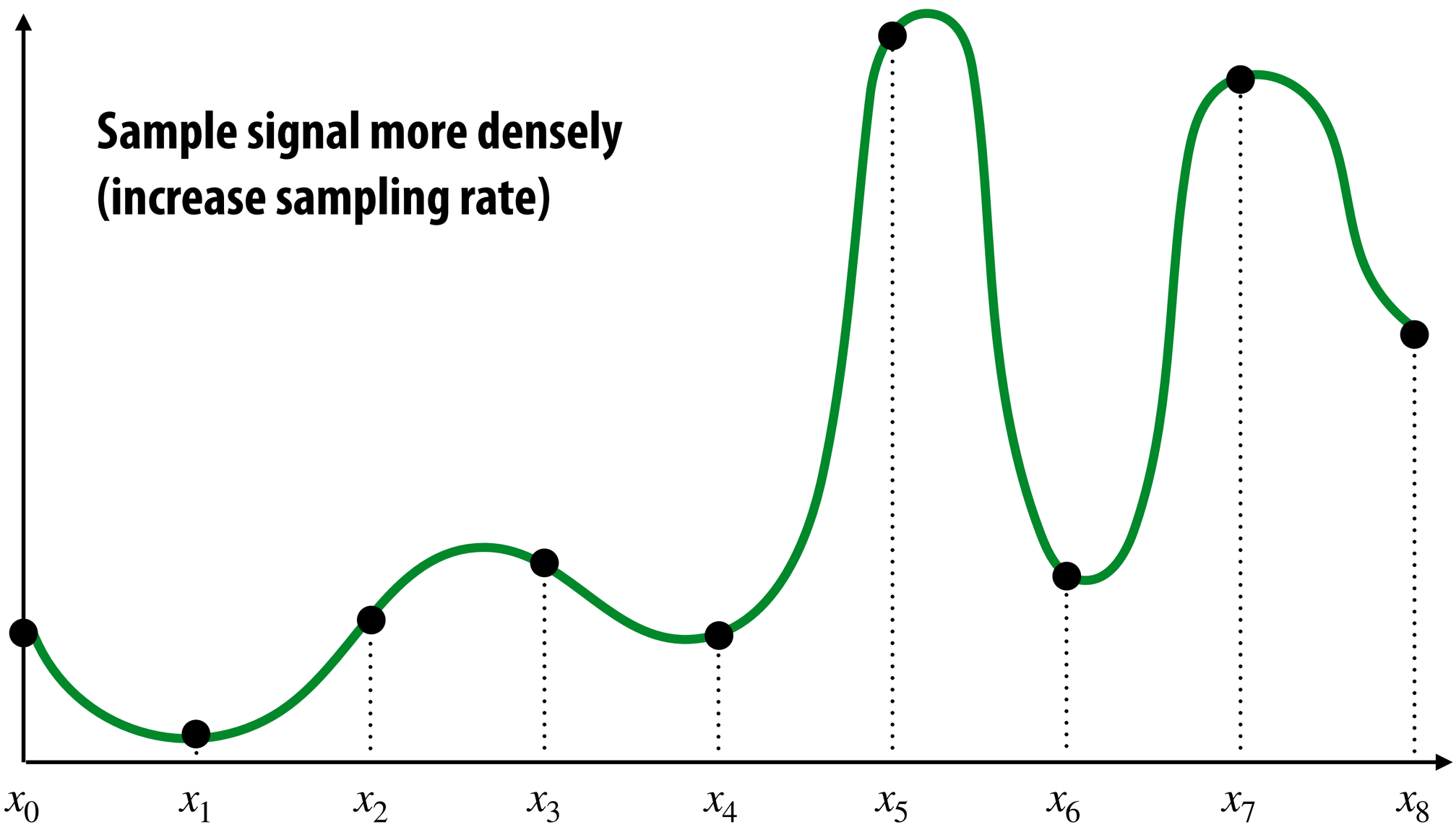
Piecewise linear approximation

$\hat{f}(x)$ = linear interpolation between values of two closest samples to x

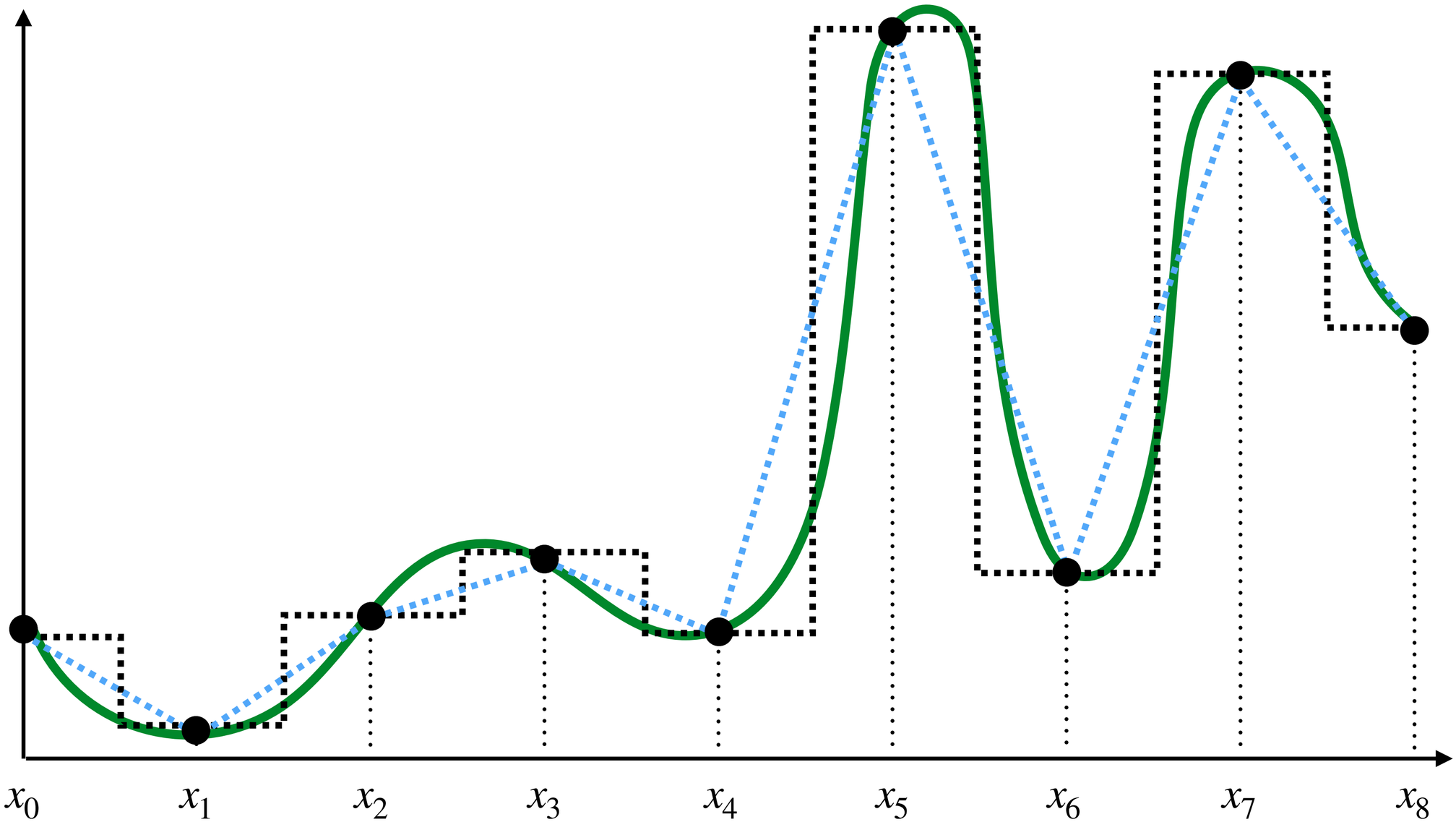


How can we represent the signal more accurately?

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

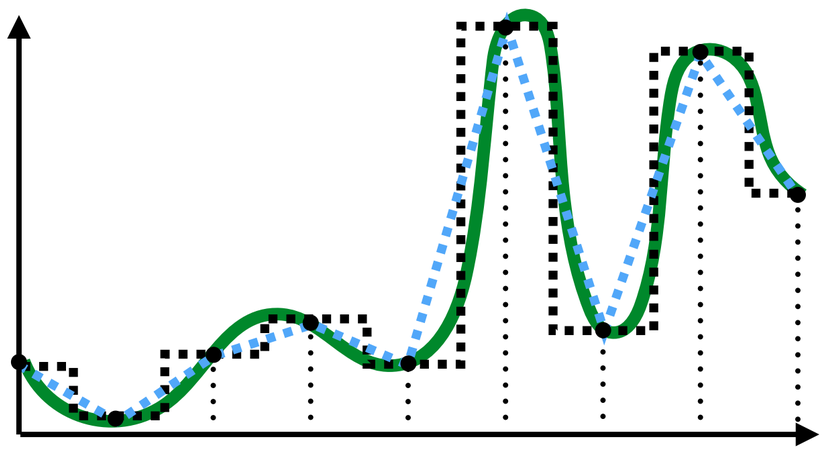
piecewise constant ("nearest neighbor")

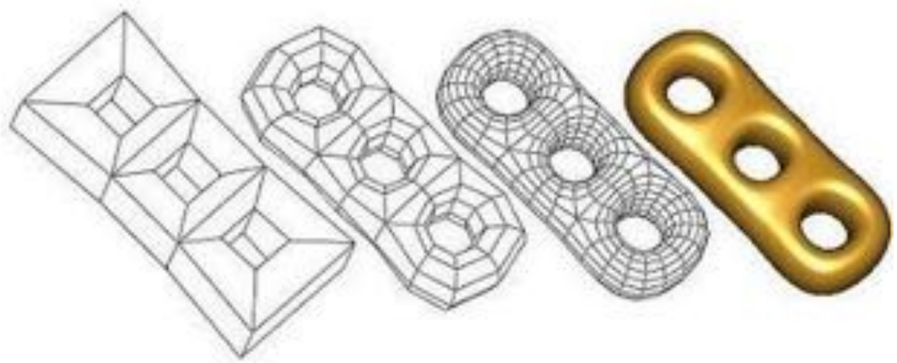
piecewise bi-linear

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…)

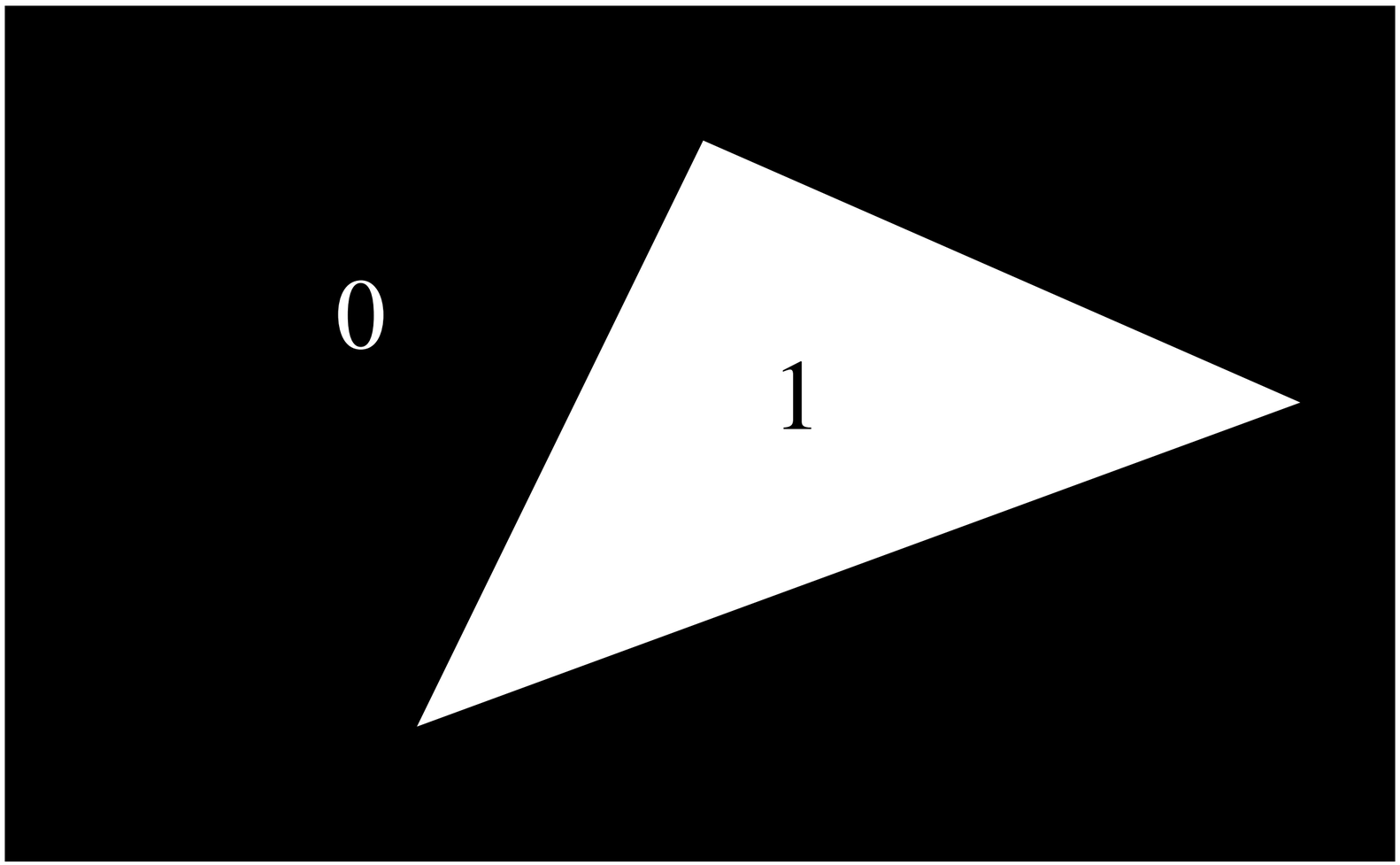




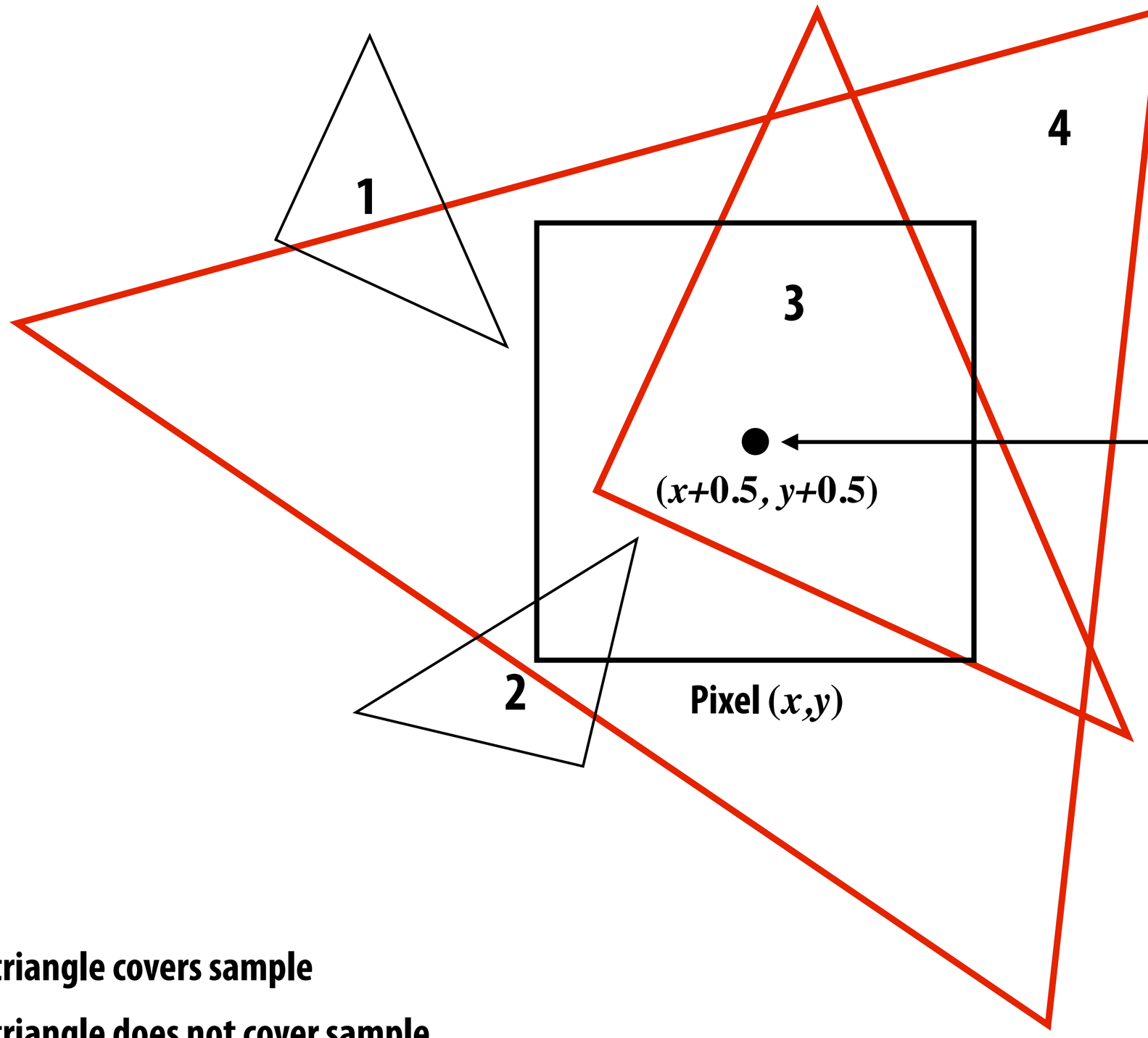
[Image credit: Wikipedia]

For rasterization, what function are we sampling?

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



Simple rasterization: just sample the coverage function

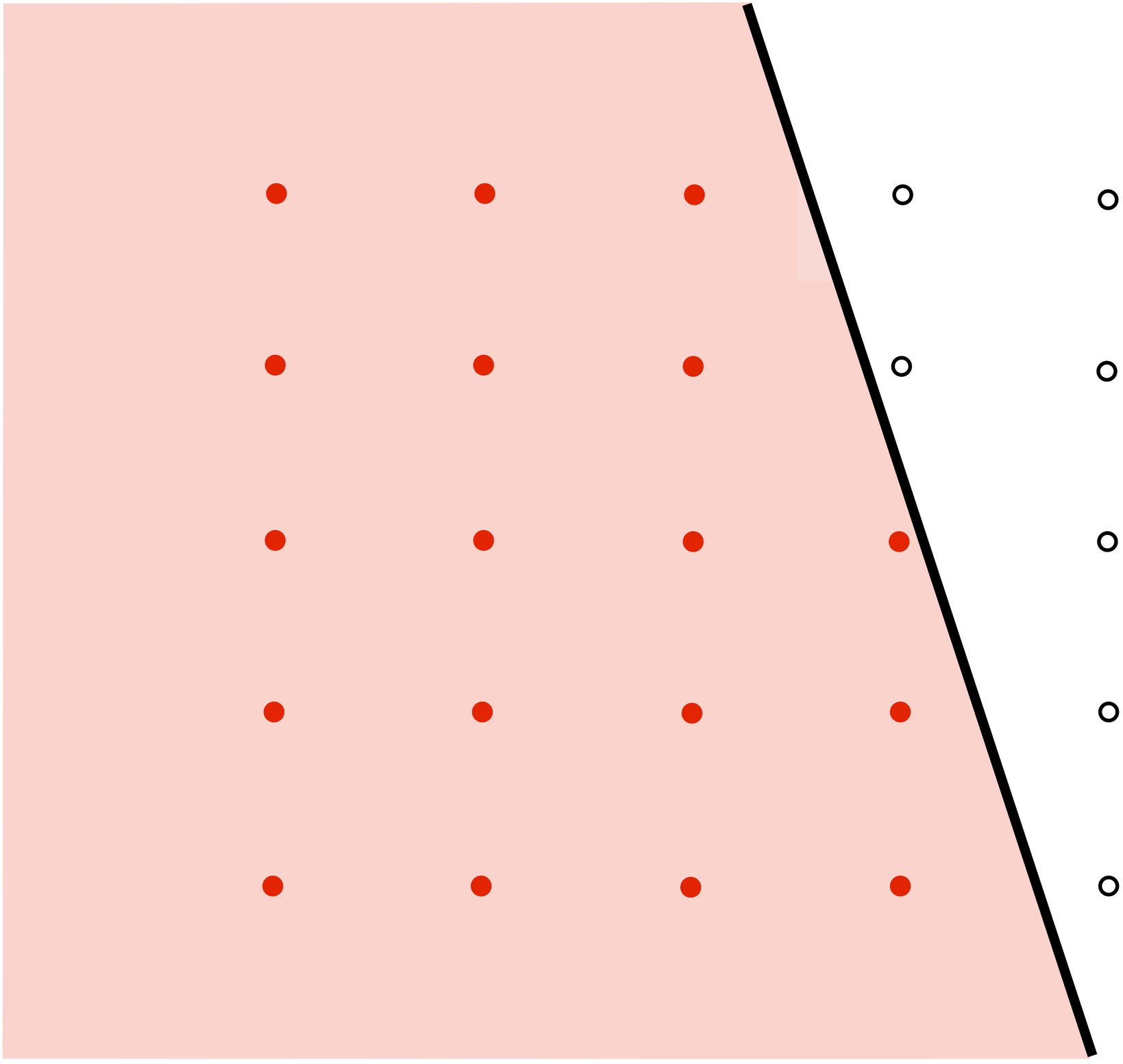


Example:
Here I chose the coverage sample point to be at a point corresponding to the pixel center.

= triangle covers sample

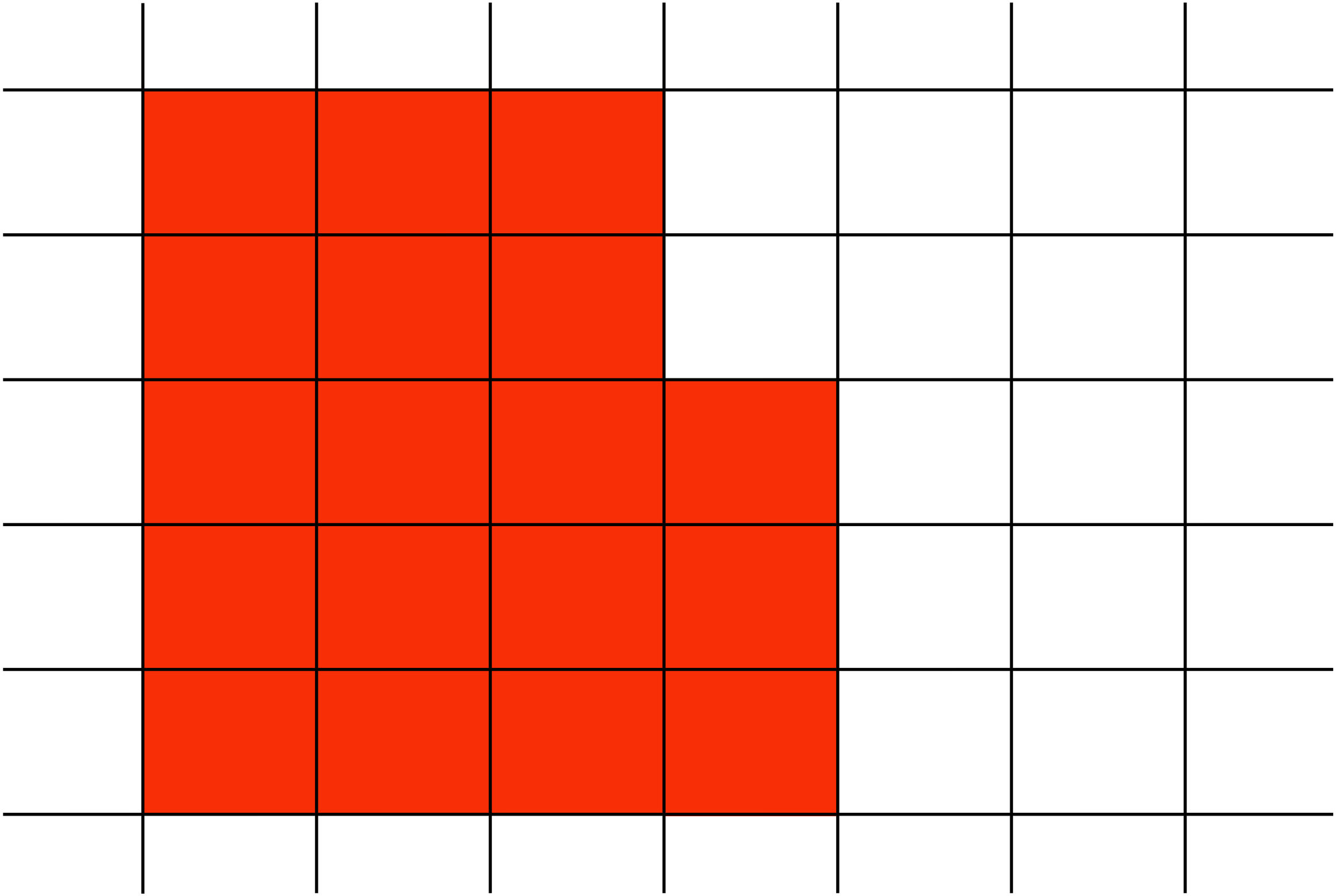
= triangle does not cover sample

Results of sampling triangle coverage



We see this when we look at the screen

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

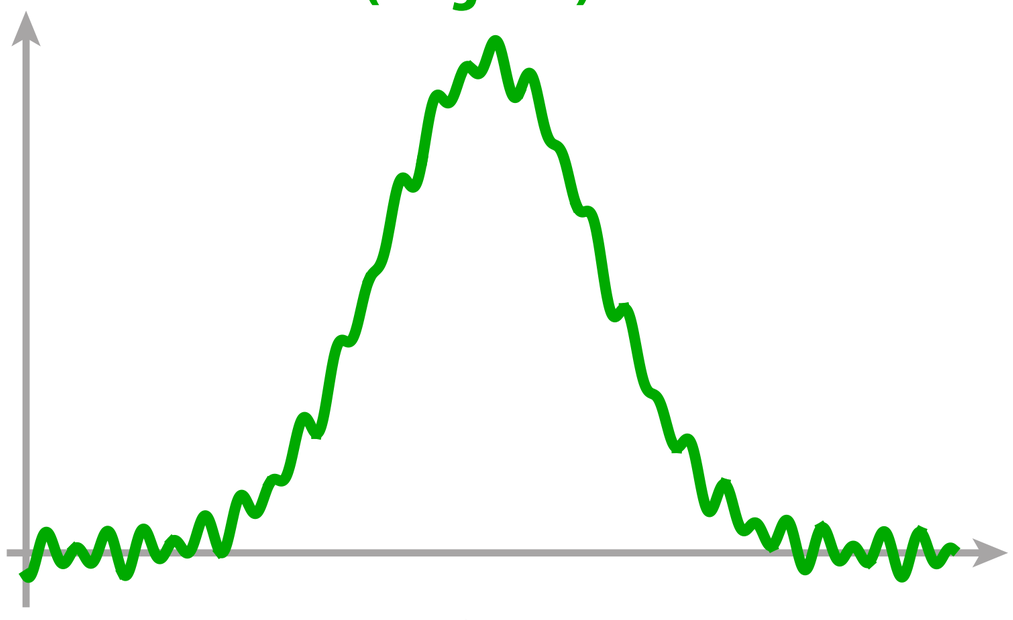


But the real coverage signal looked like this!

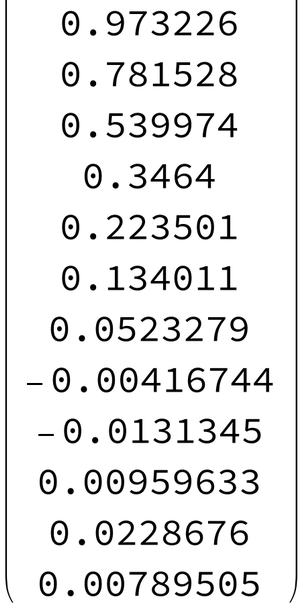
Aliasing

Sampling & Reconstruction

continuous signal (original)



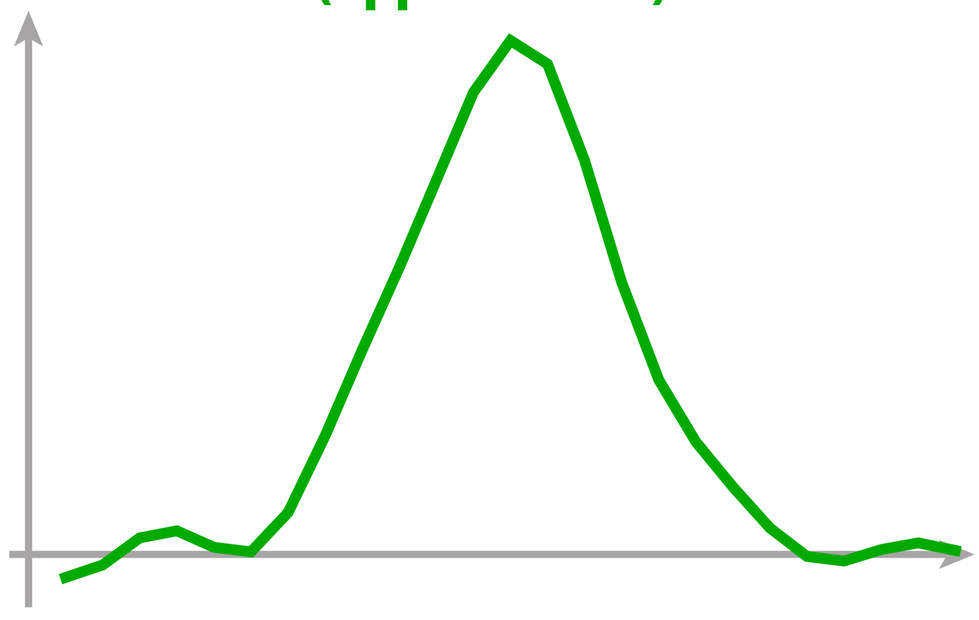
sample



digital information

reconstruct

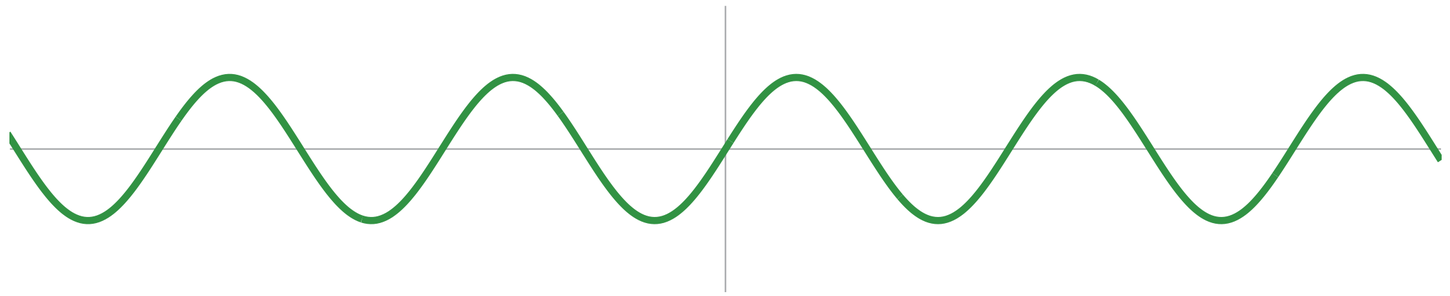
continuous signal (approximate)



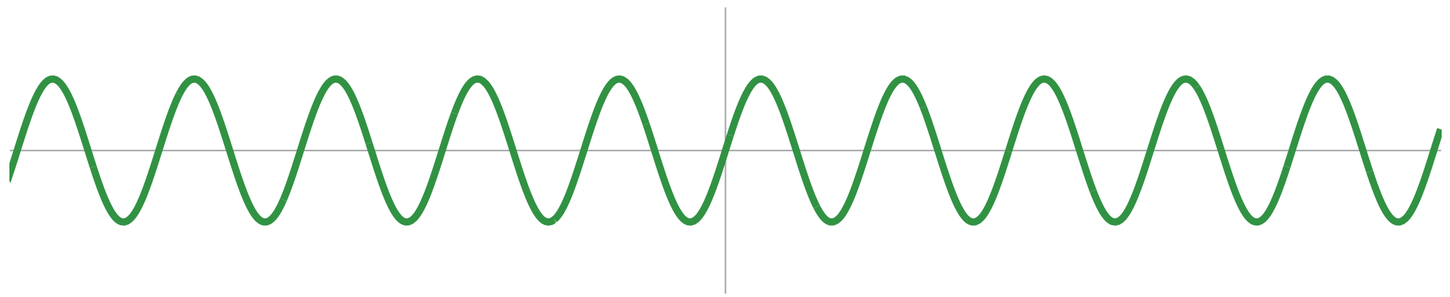
Goal: reproduce original signal as accurately as possible.

1D signal can be expressed as a superposition of frequencies

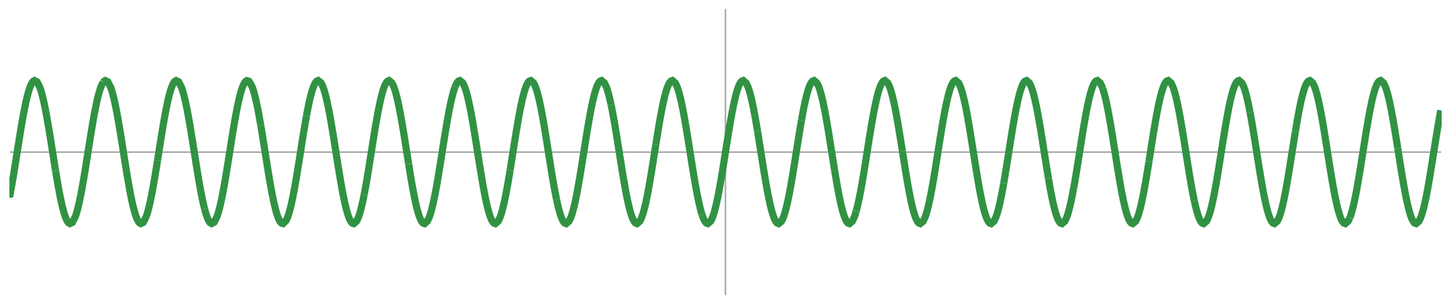
$f_1(x) = sin(\pi x)$



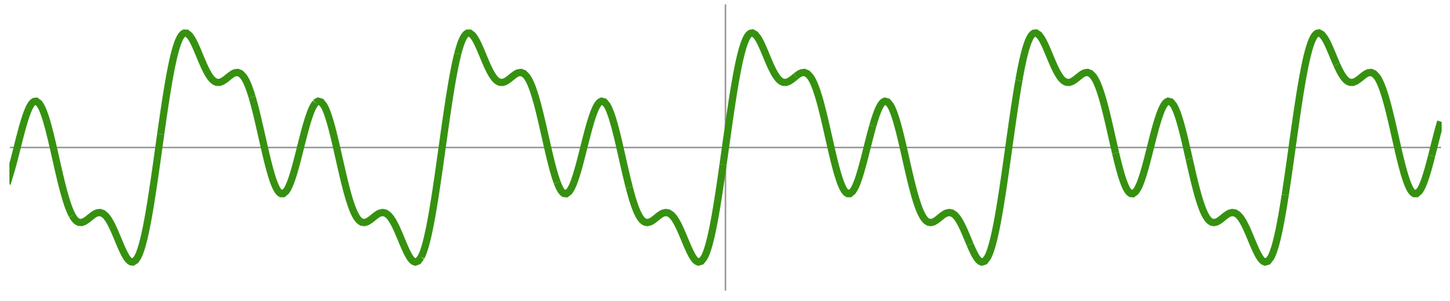
$f_2(x) = sin(2\pi x)$



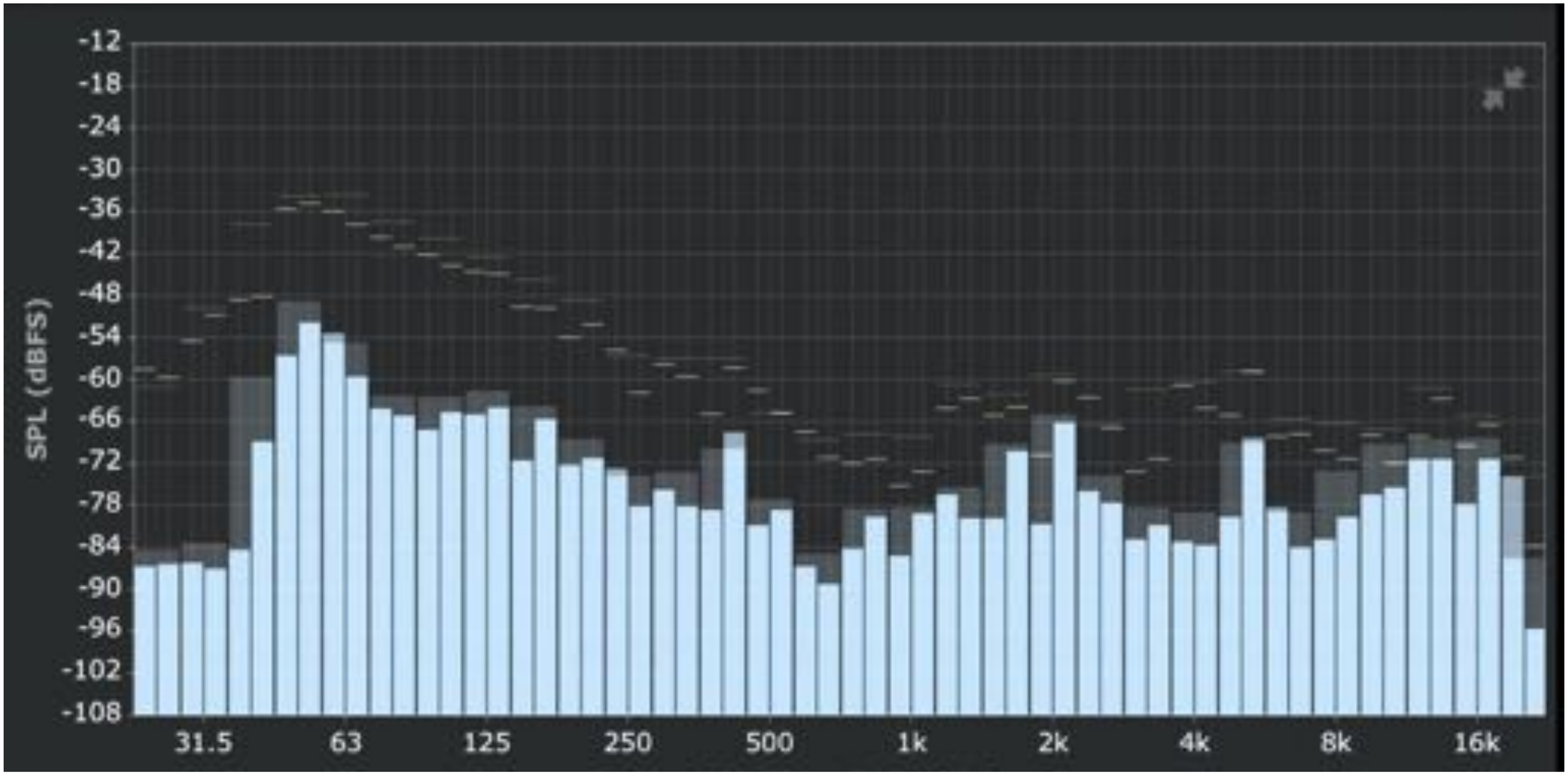
$f_4(x) = sin(4\pi x)$



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



E.g., audio spectrum analyzer shows the amplitude of each frequency



Intensity of low-frequencies (bass)

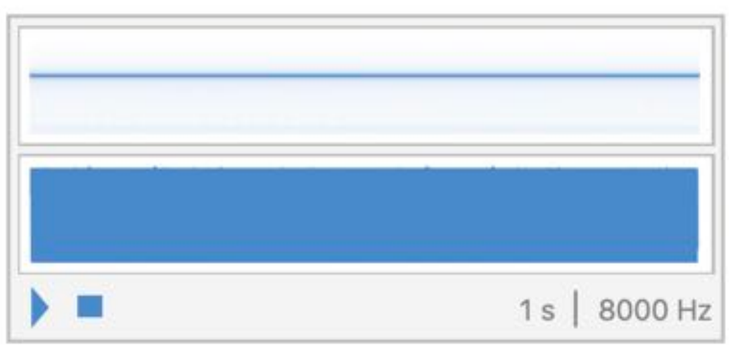
Intensity of high frequencies

Image credit: ONYX Apps

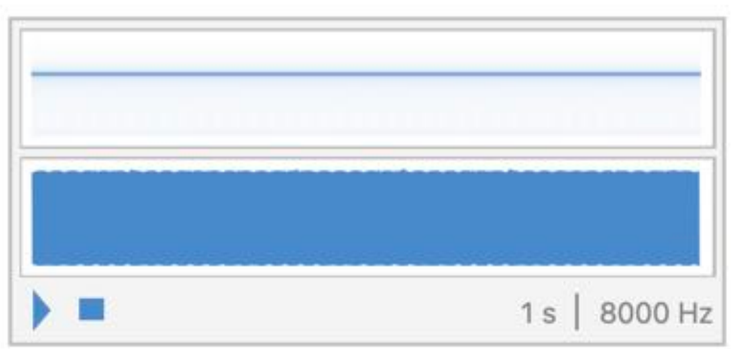
Aliasing in Audio

Get a constant tone by playing a sinusoid of frequency ω:

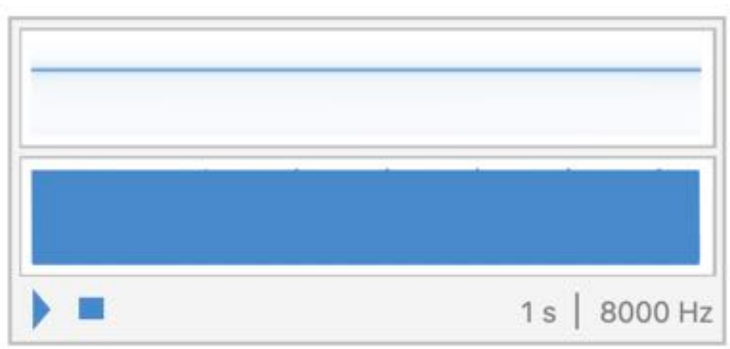
```
Play[Sin[4000 t], {t, 0, 1}]
```




```
Play[Sin[5000 t], {t, 0, 1}]
```

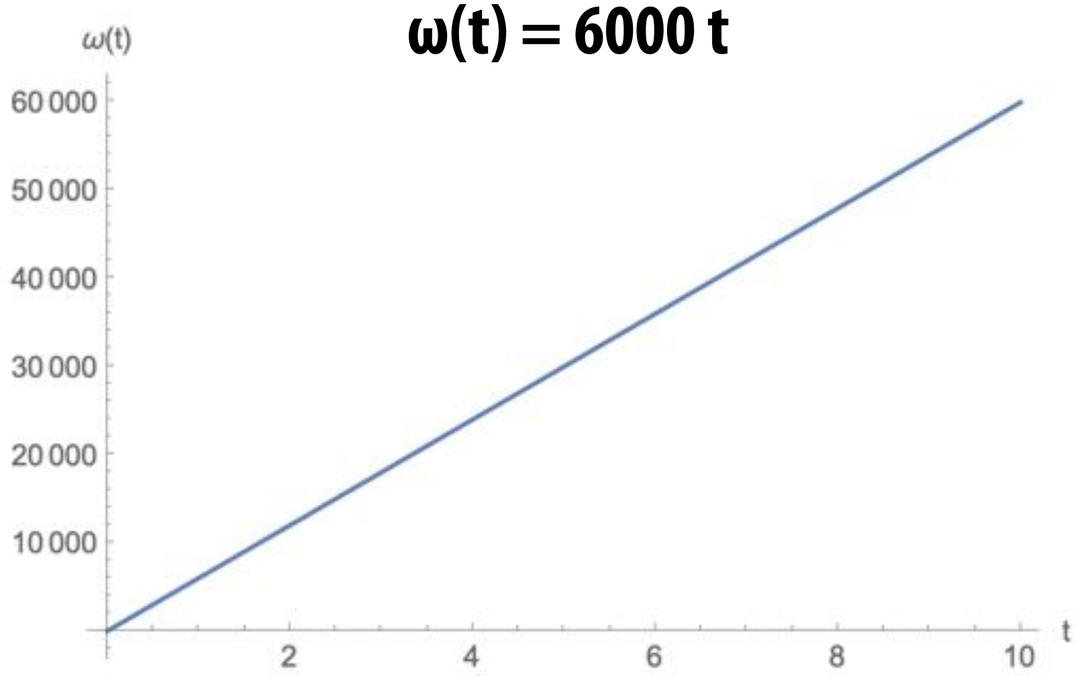



```
Play[Sin[6000 t], {t, 0, 1}]
```

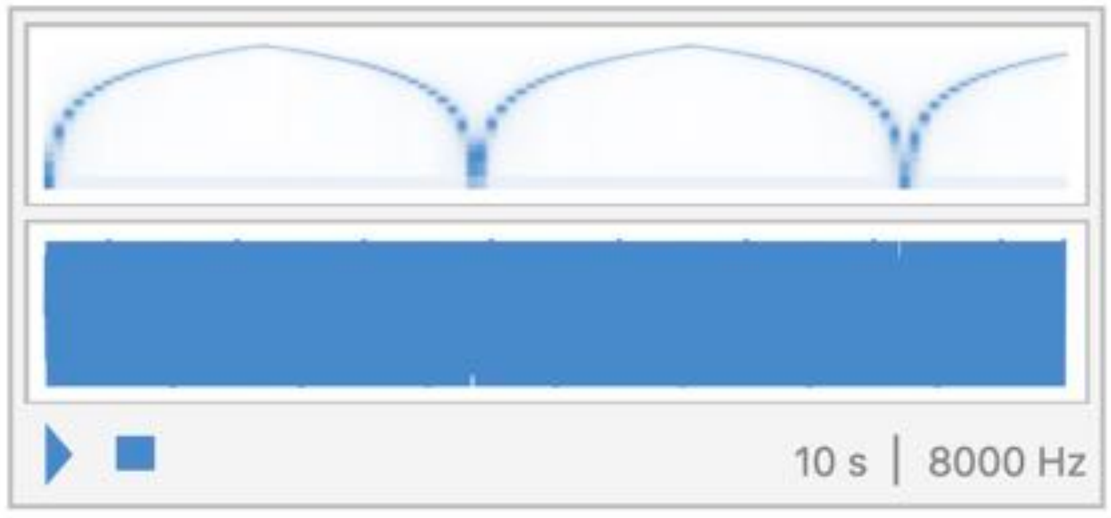


Q: What happens if we <u>increase</u> ω over time?

$\omega(t) = 6000\,t$

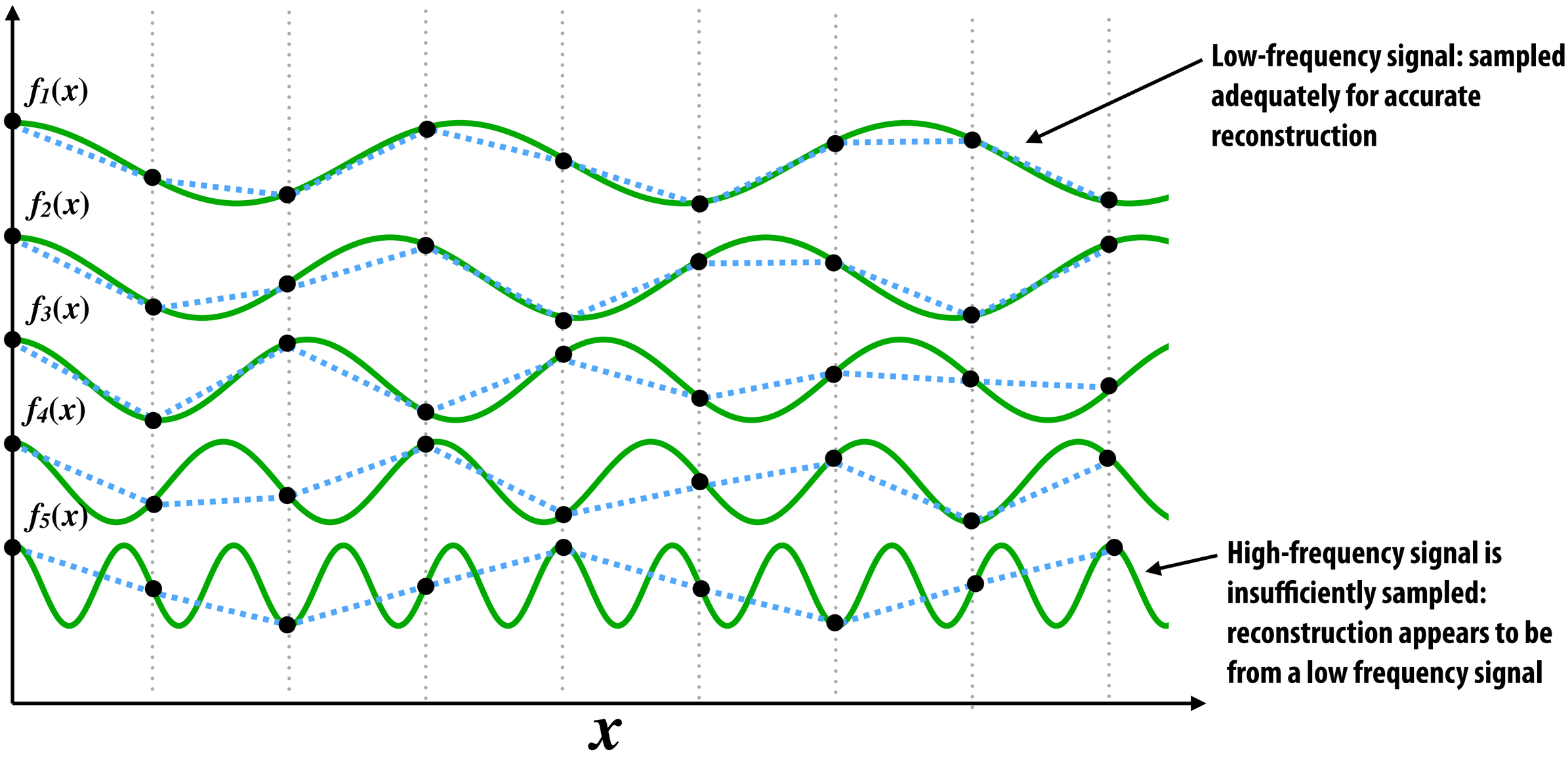



```
Play[Sin[ω t], {t, 0, 10}]
```



Why did that happen?

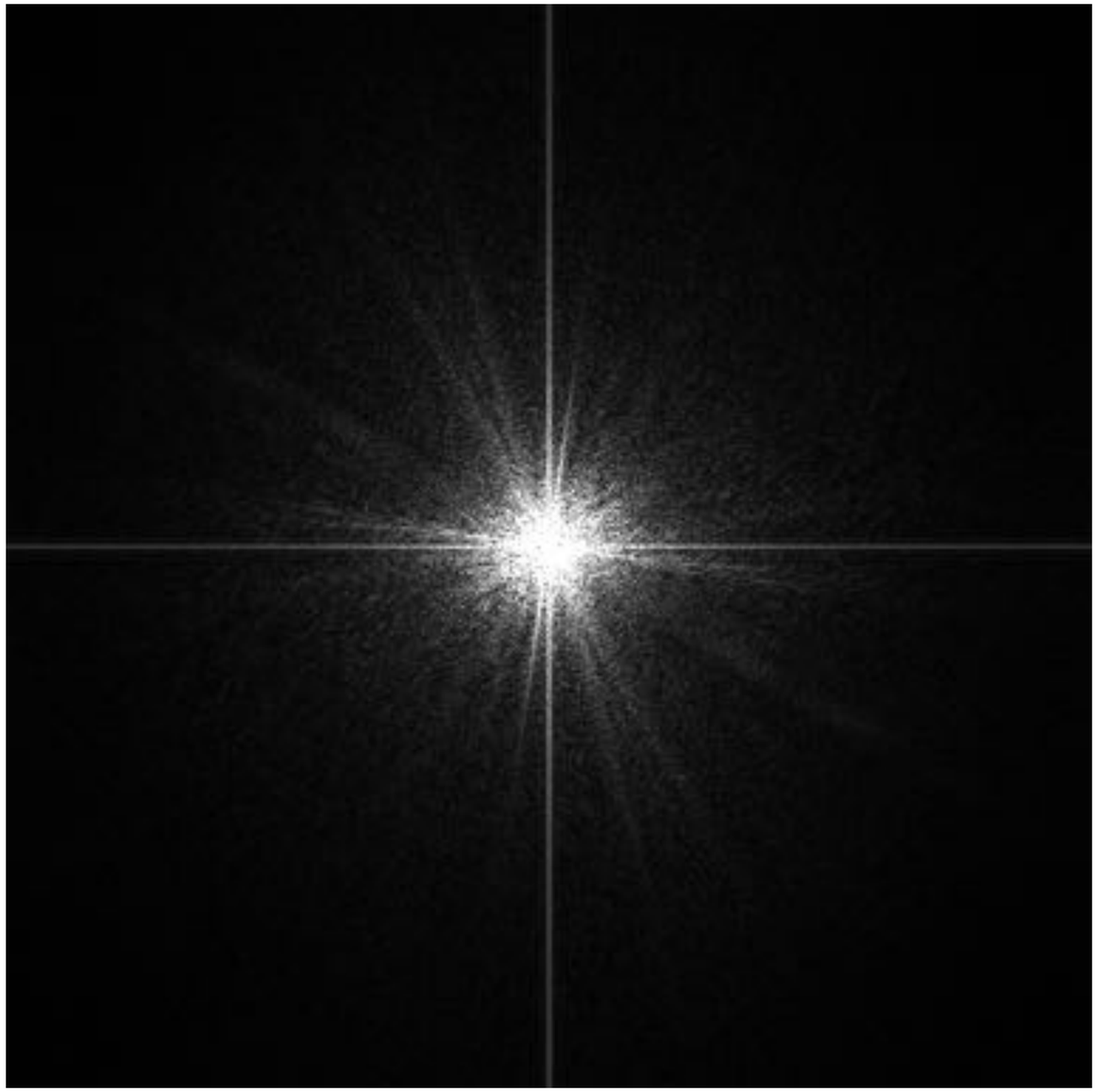
Undersampling high-frequency signals results in aliasing



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

Images can also be decomposed into “frequencies”

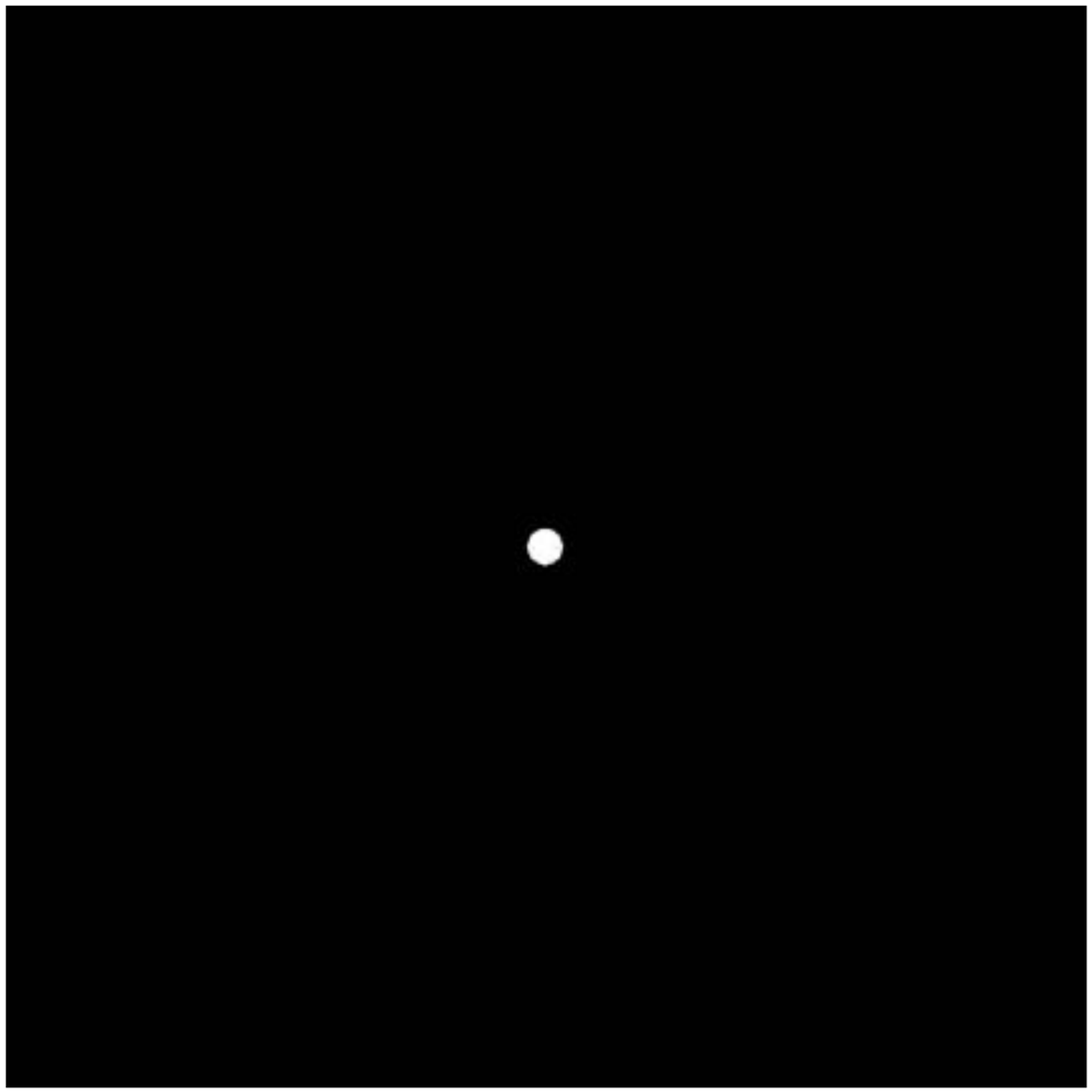
Spatial domain result



Spectrum

Low frequencies only (smooth gradients)

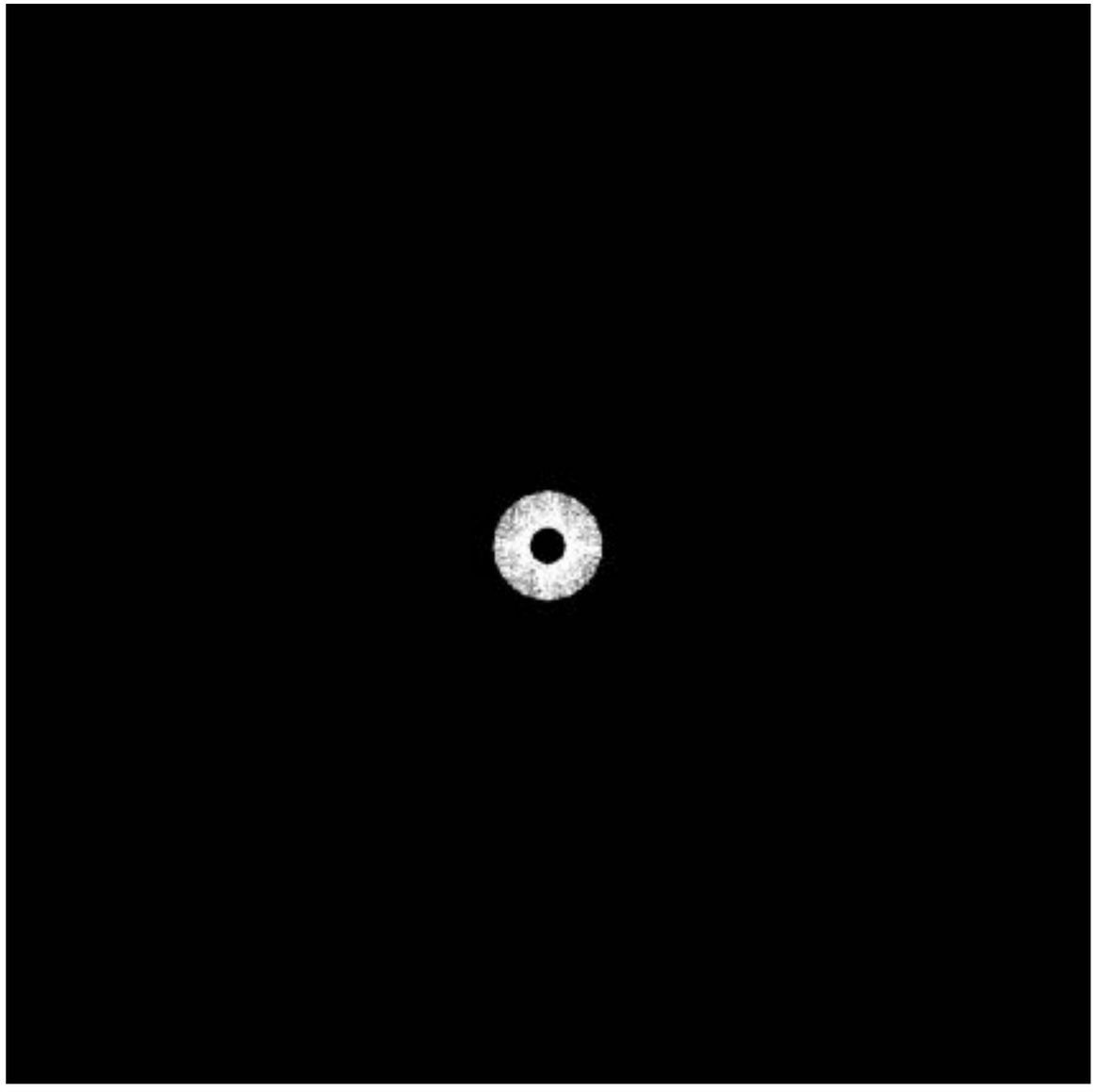
Spatial domain result



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

Mid-range frequencies

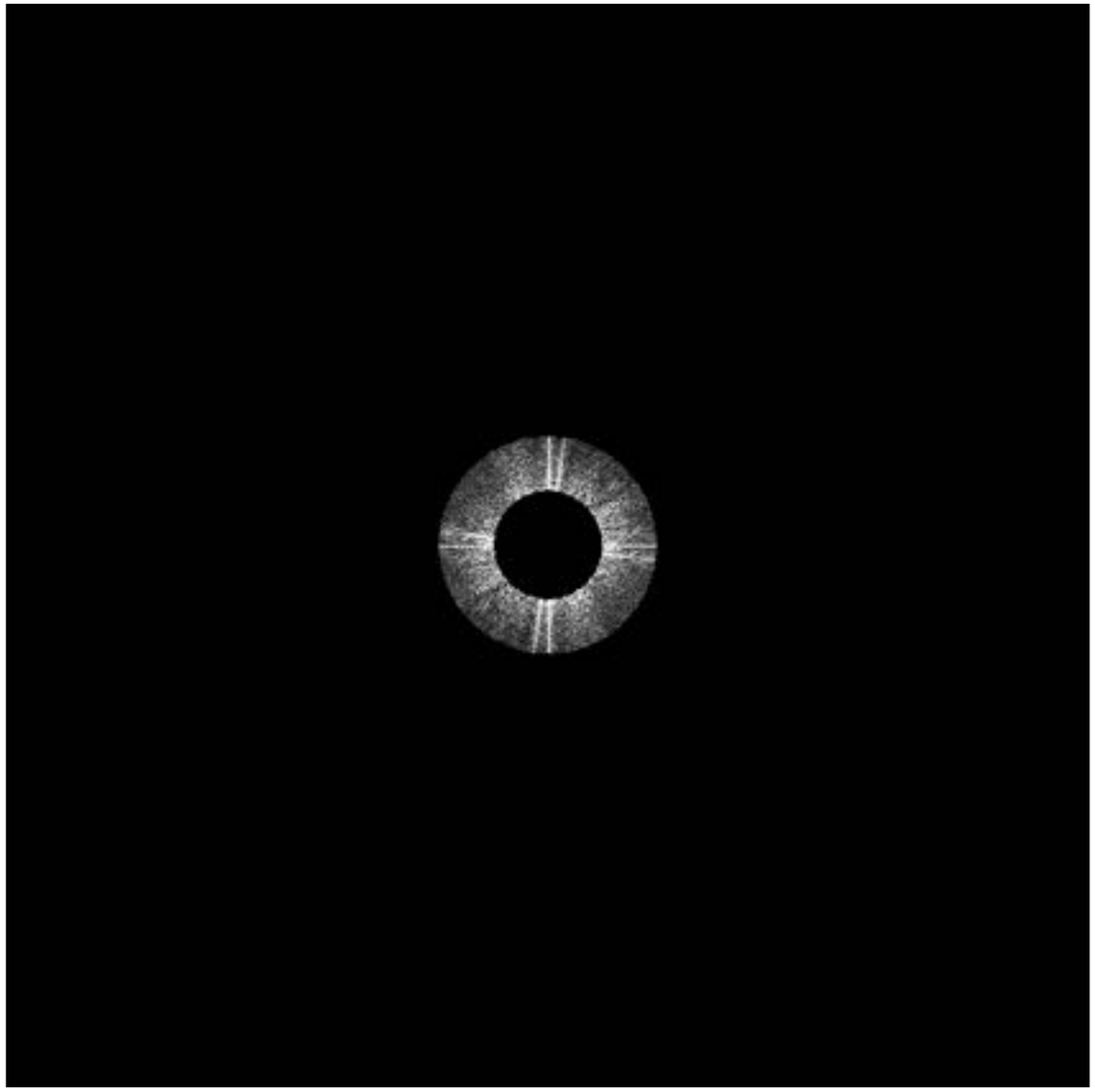
Spatial domain result



Spectrum (after band-pass filter)

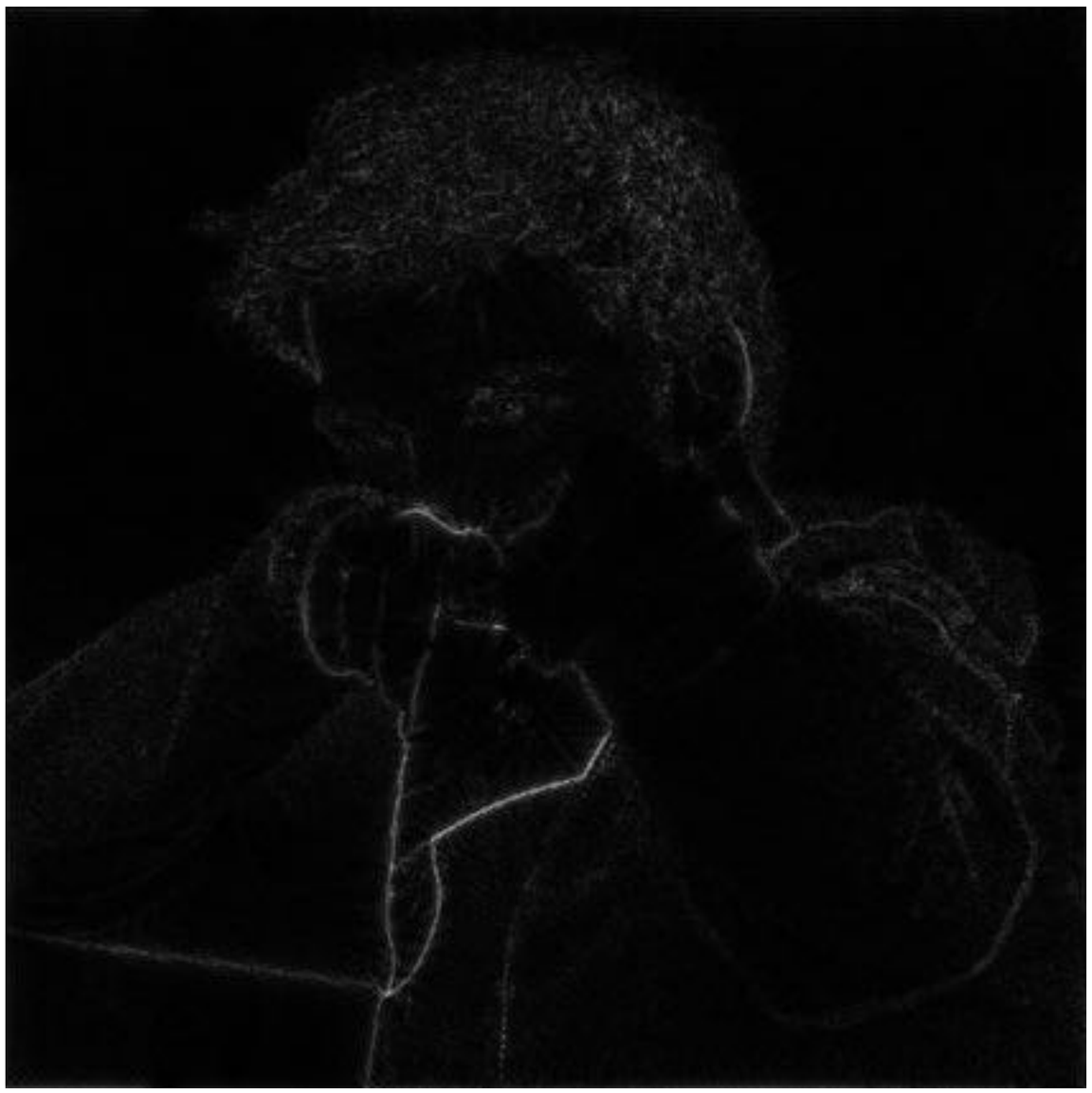
Mid-range frequencies

Spatial domain result

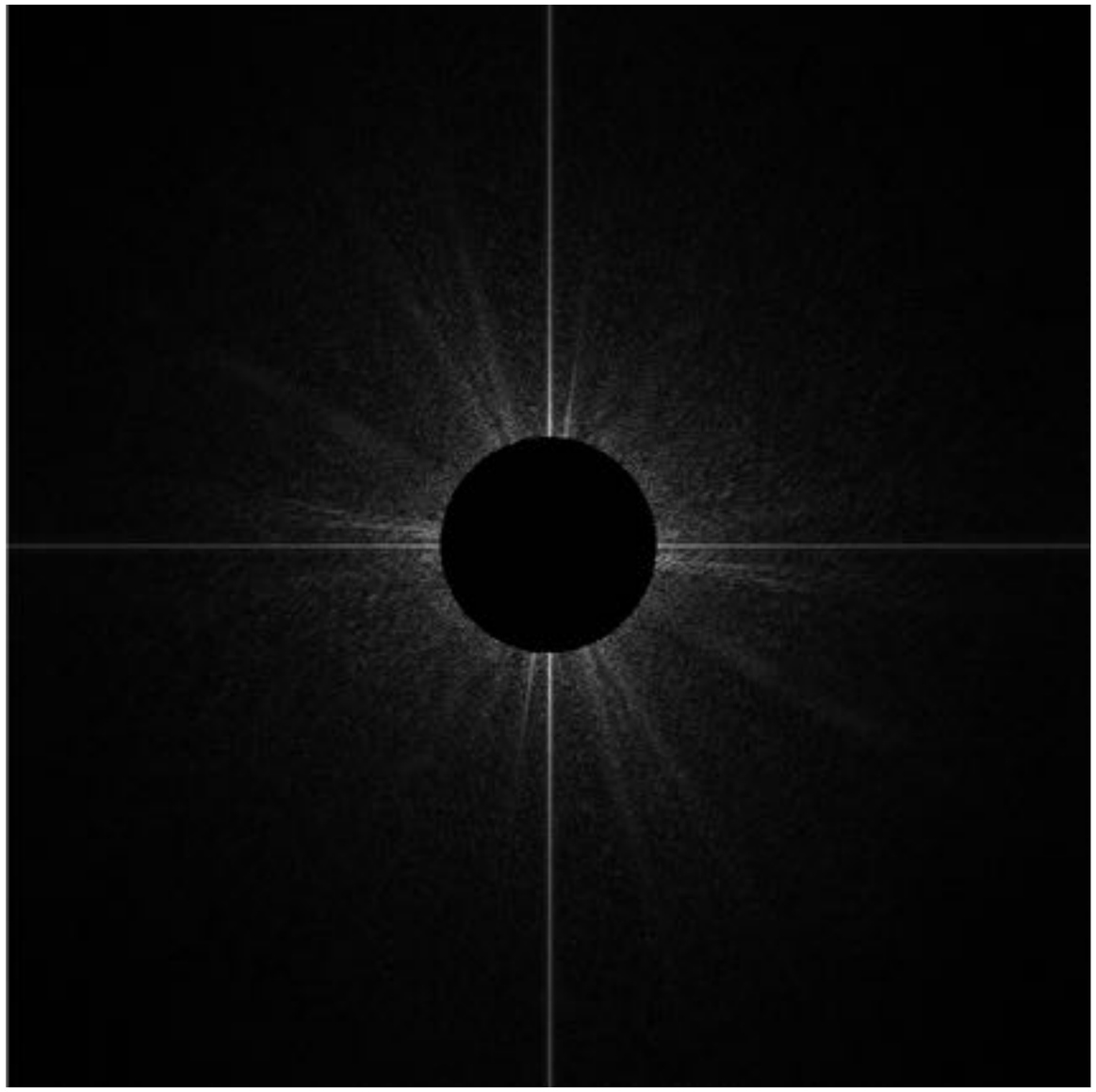


Spectrum (after band-pass filter)

High frequencies (edges)

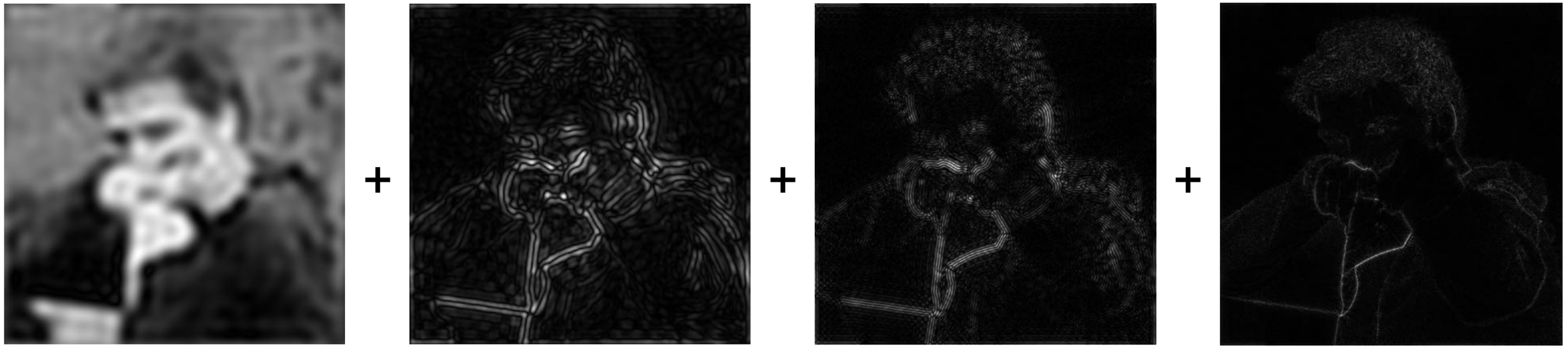


Spatial domain result
(strongest edges)



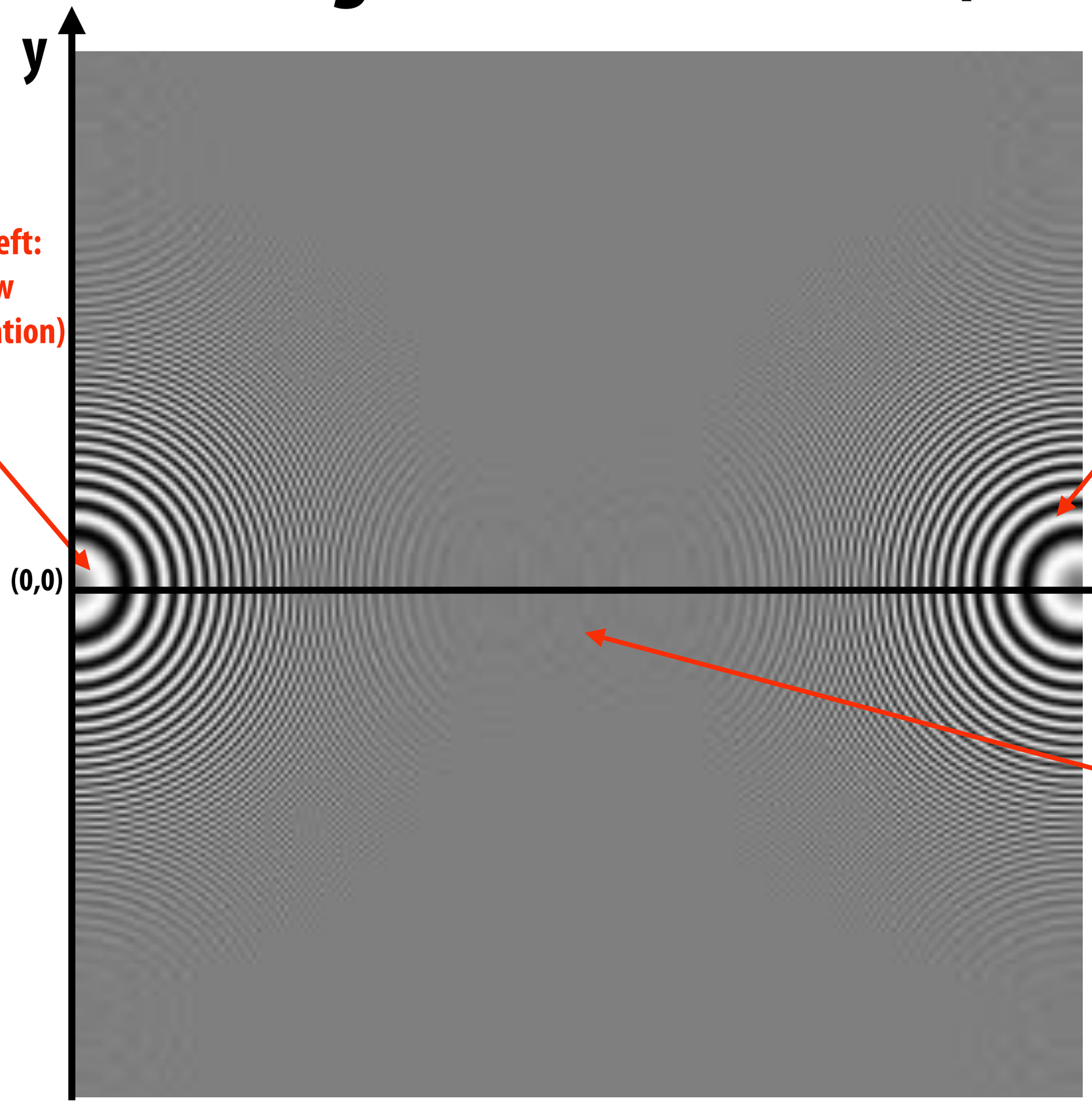
Spectrum (after high-pass filter)
All frequencies below threshold
have 0 magnitude

An image as a sum of its frequency components



=

Spatial aliasing: the function $\sin(x^2 + y^2)$



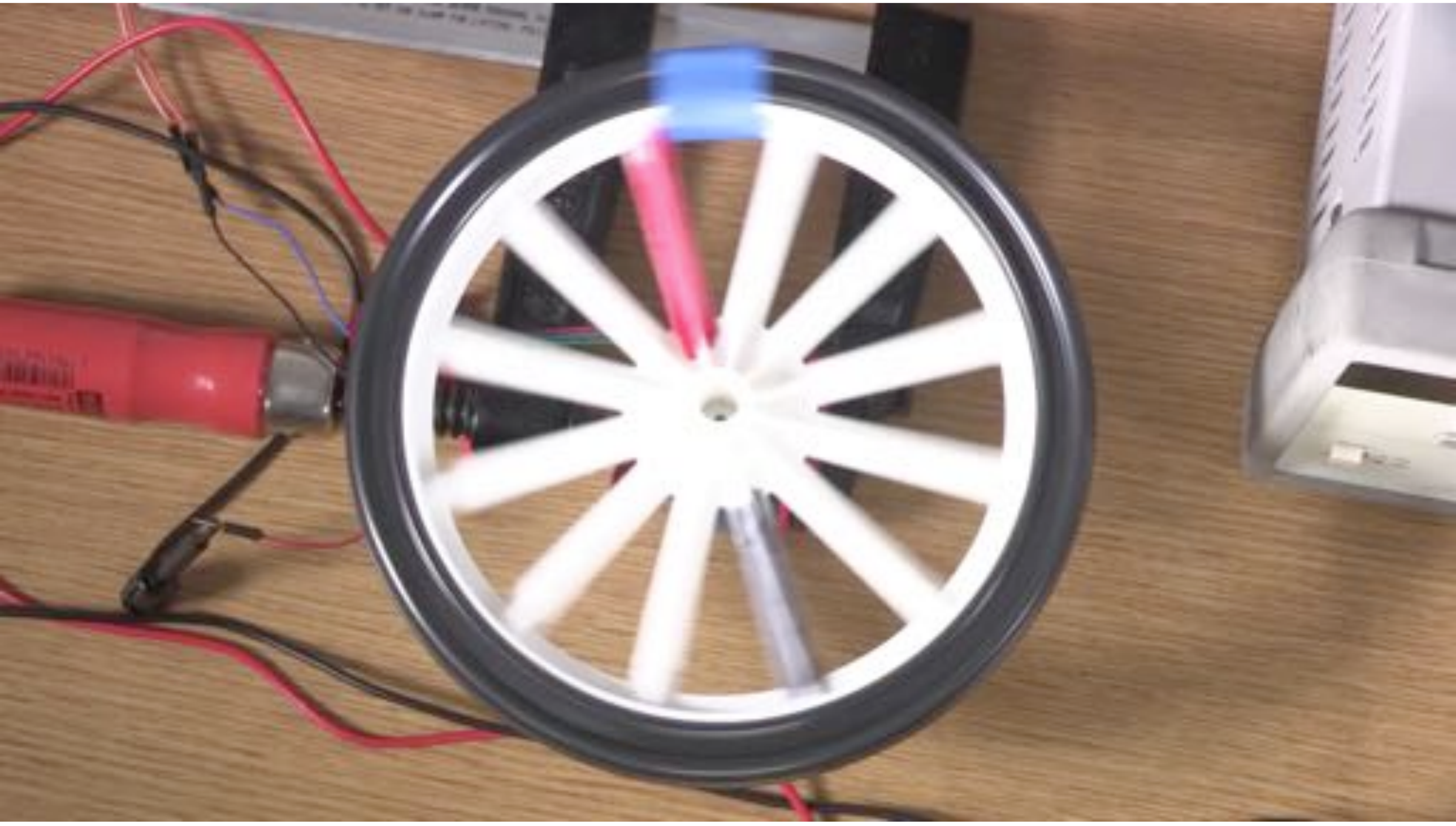
Rings in center-left: Actual signal (low frequency oscillation)

Right: aliasing from undersampling high frequency oscillation makes it appear that rings are low-frequency (they're not!)

Middle: ring frequency approaches limit of what we can represent w/ individual pixels

Figure credit: Pat Hanrahan and Bryce Summers

Temporal aliasing: wagon wheel effect

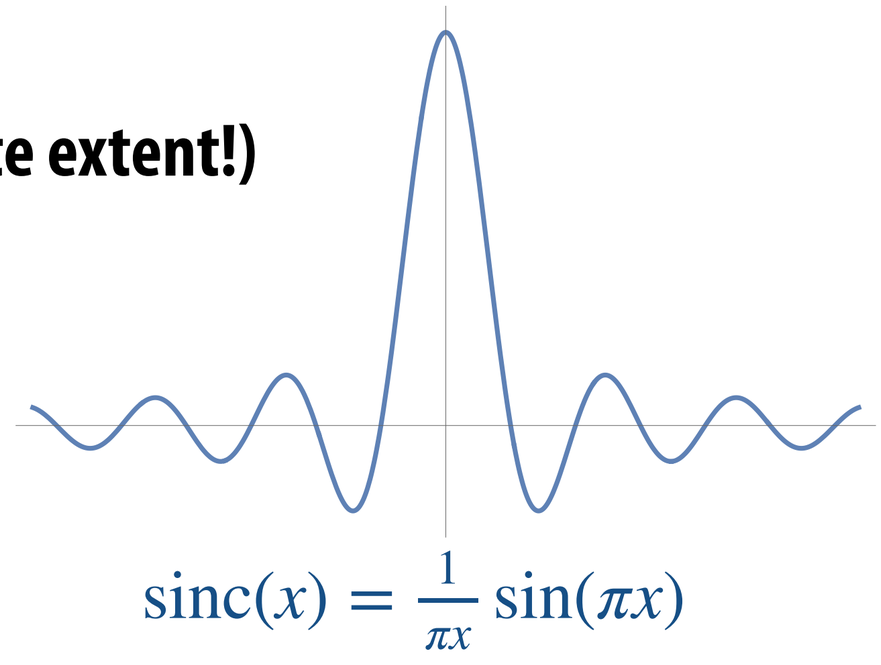


Camera's frame rate (temporal sampling rate) is too low for rapidly spinning wheel.

Nyquist-Shannon theorem

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

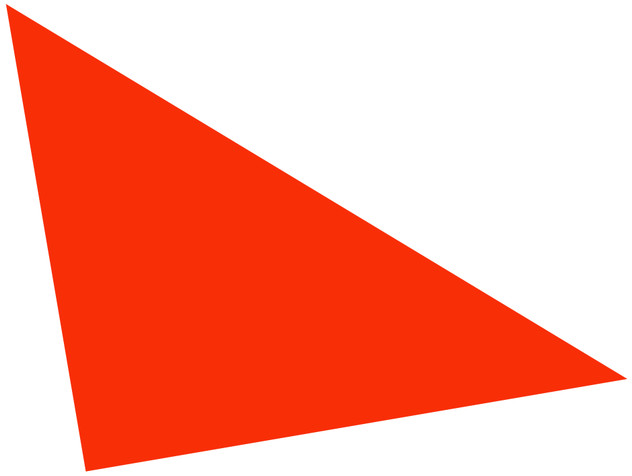
- The signal can be perfectly reconstructed if sampled with period $T = 1 / 2\omega_0$
- …and if interpolation is performed using a "sinc filter"
  - ideal filter with no frequencies above cutoff (infinite extent!)

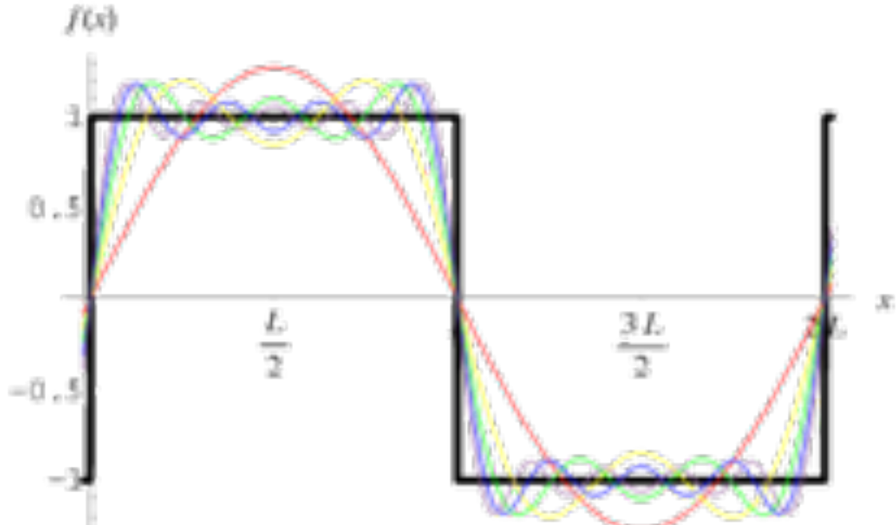


Challenges of sampling in computer graphics

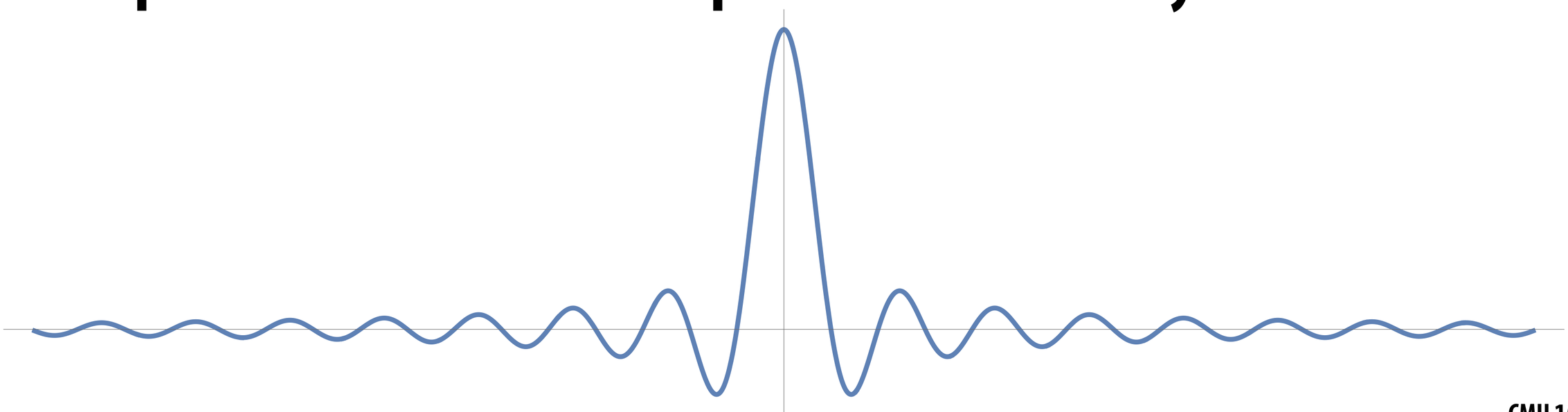
- **Signals are often not band-limited in computer graphics. Why?**

Hint:



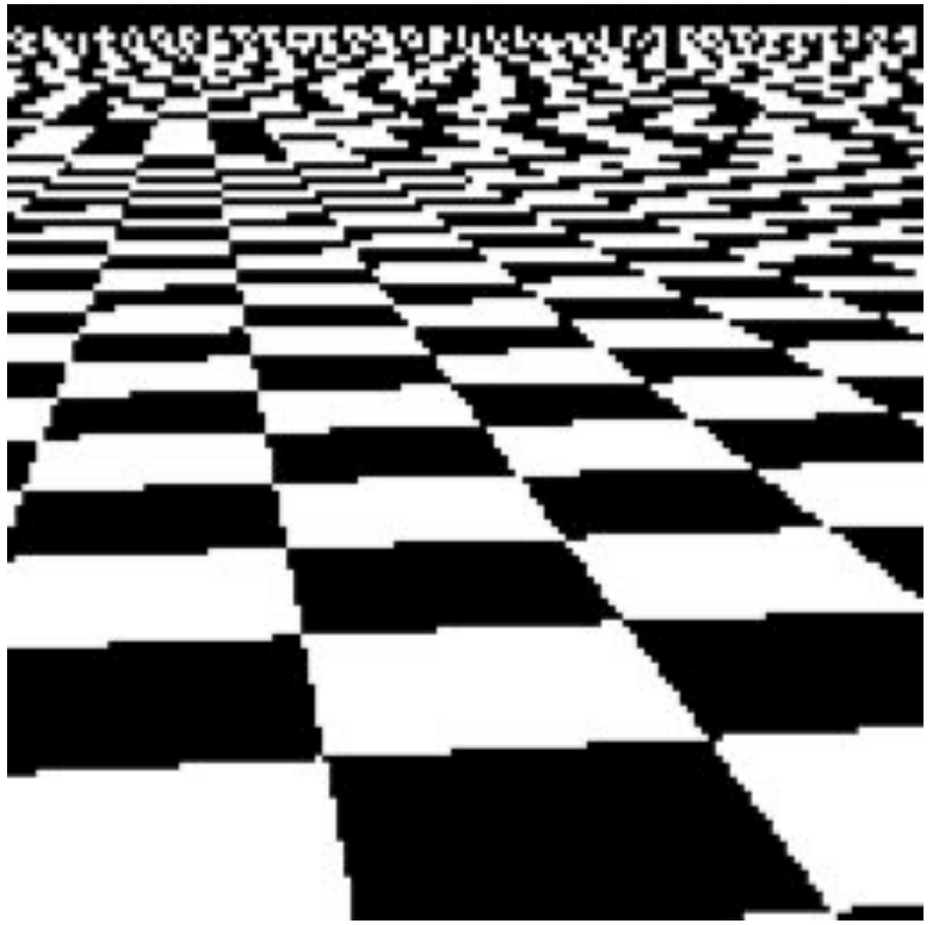


- **Also, infinite extent of "ideal" reconstruction filter (sinc) is impractical for efficient implementations. Why?**



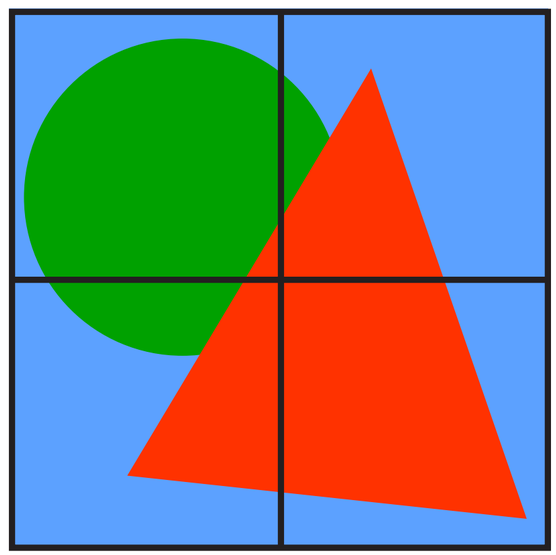
Aliasing artifacts in images

- Imperfect sampling + imperfect reconstruction leads to image artifacts
  - "Jaggies" in a static image
  - "Roping" or "shimmering" of images when animated
  - Moiré patterns in high-frequency areas of images



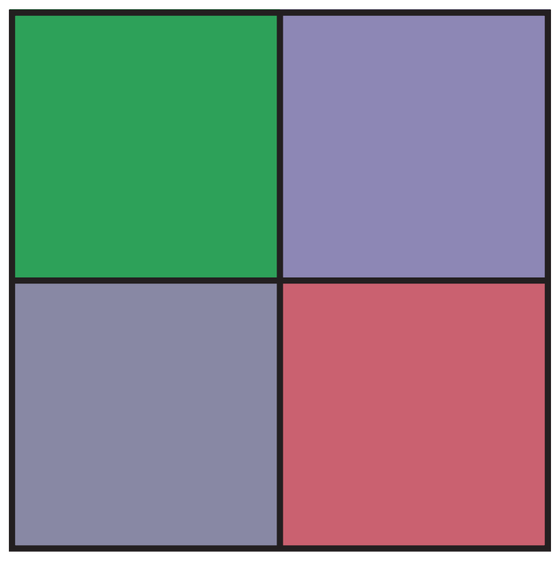
How can we reduce aliasing?

- No matter what we do, aliasing is a fact of life: any sampled representation eventually fails to capture frequencies that are too high.



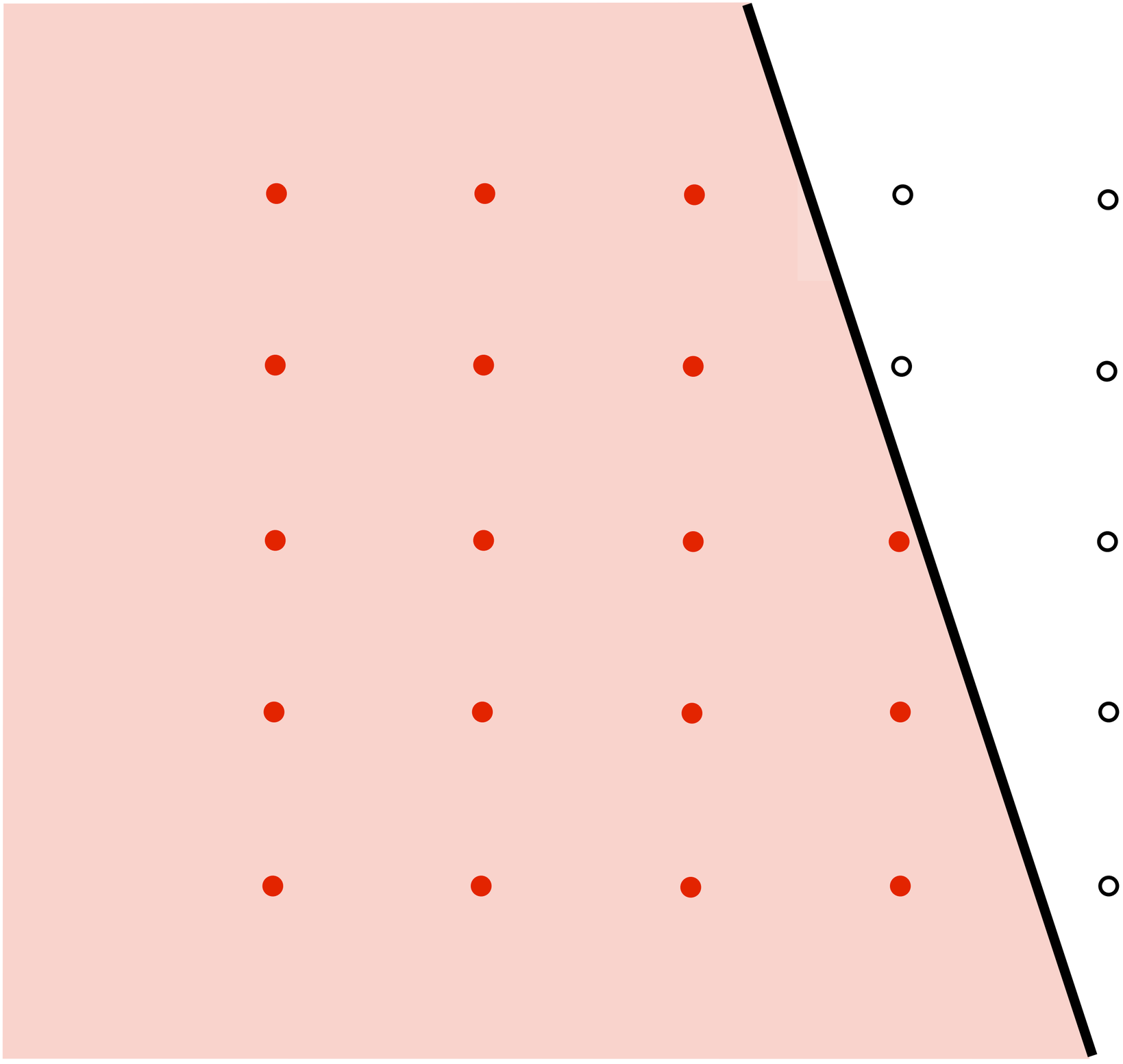
- But we can still do our best to try to match sampling and reconstruction so that the signal we reproduce looks as much as possible like the signal we acquire

- For instance, if we think of a pixel as a "little square" of light, then we want the total light emitted to be the same as the total light in that pixel
  - I.e., we want to integrate the signal over the pixel ("box filter")

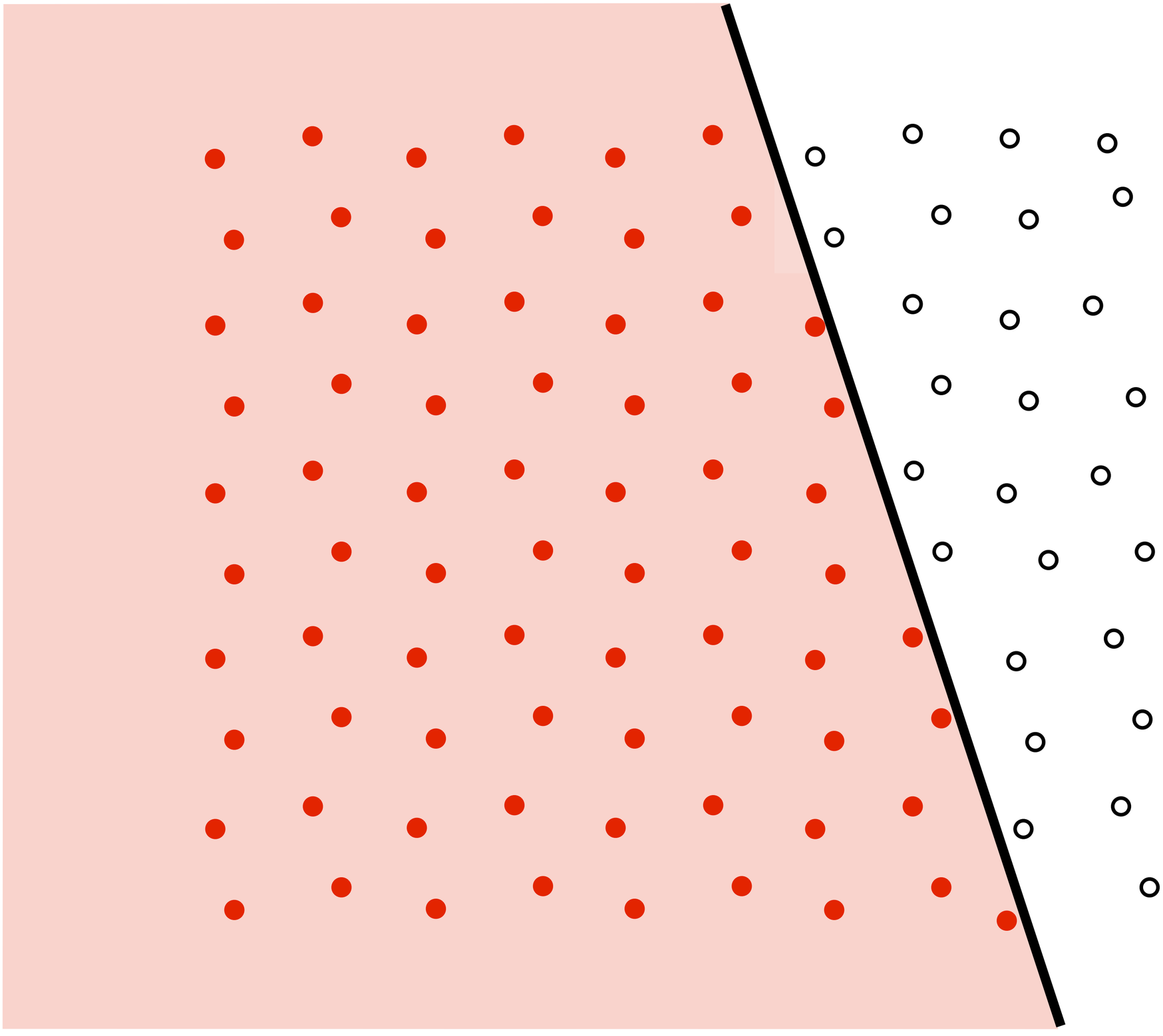


Let's (approximately) integrate the signal coverage (x,y) by sampling…

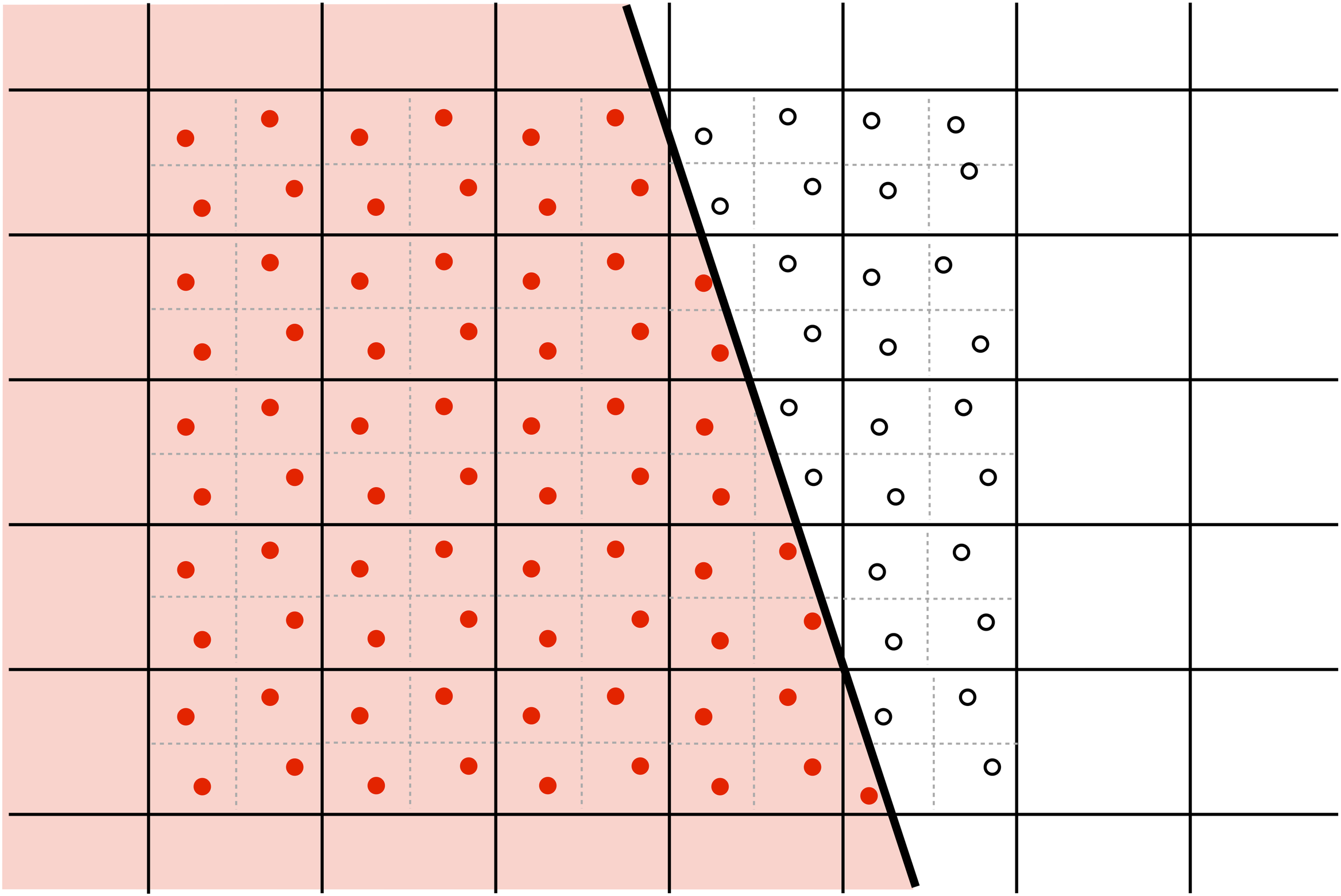
Initial coverage sampling rate (1 sample per pixel)



Increase frequency of sampling coverage signal

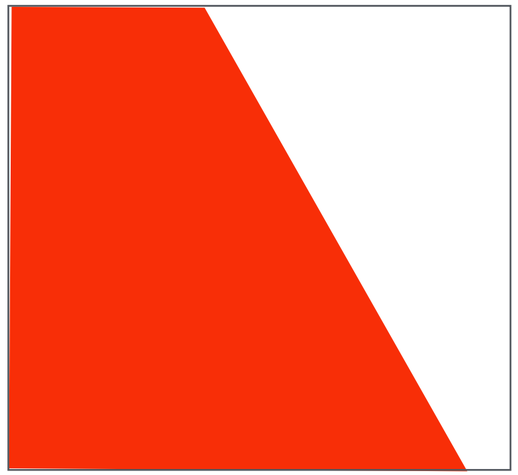


Supersampling

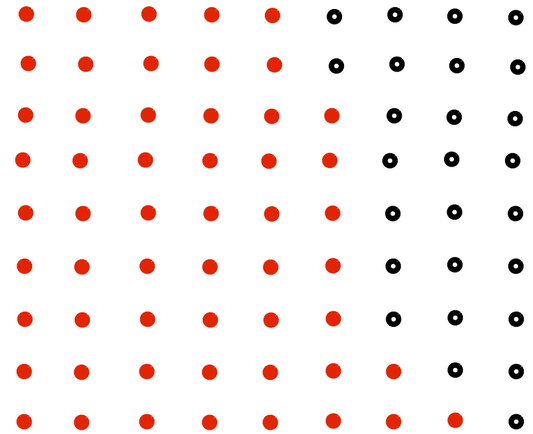


Resampling

Converting from one discrete sampled representation to another

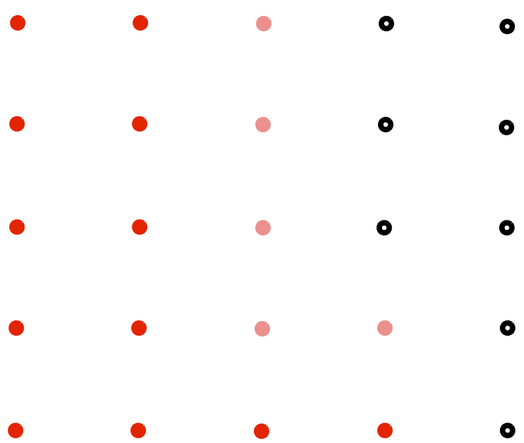


Original signal
(high frequency edge)



Dense sampling of
reconstructed signal

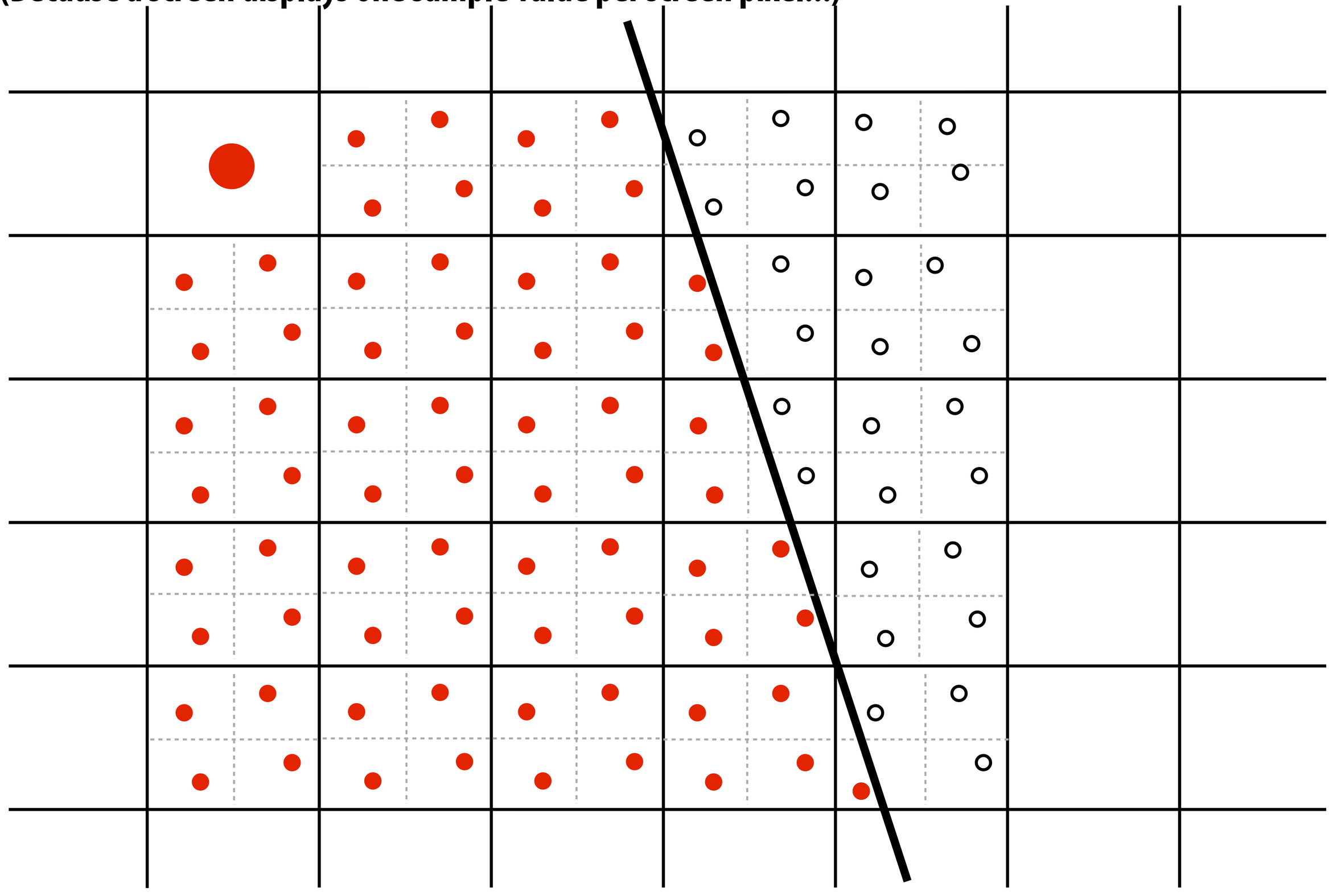
Reconstructed signal
(lacks high frequencies)



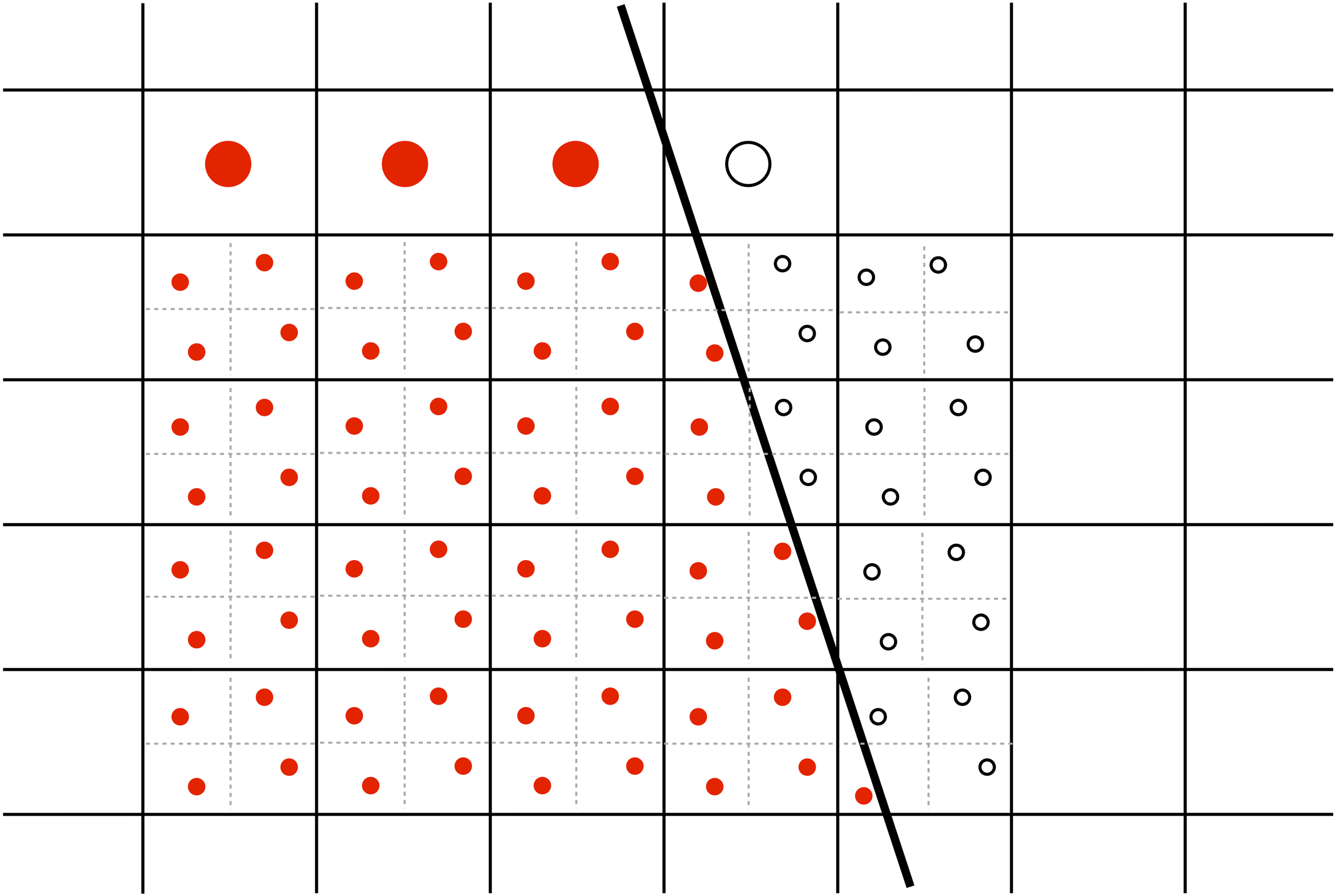
Coarsely sampled signal

Resample to display's pixel resolution

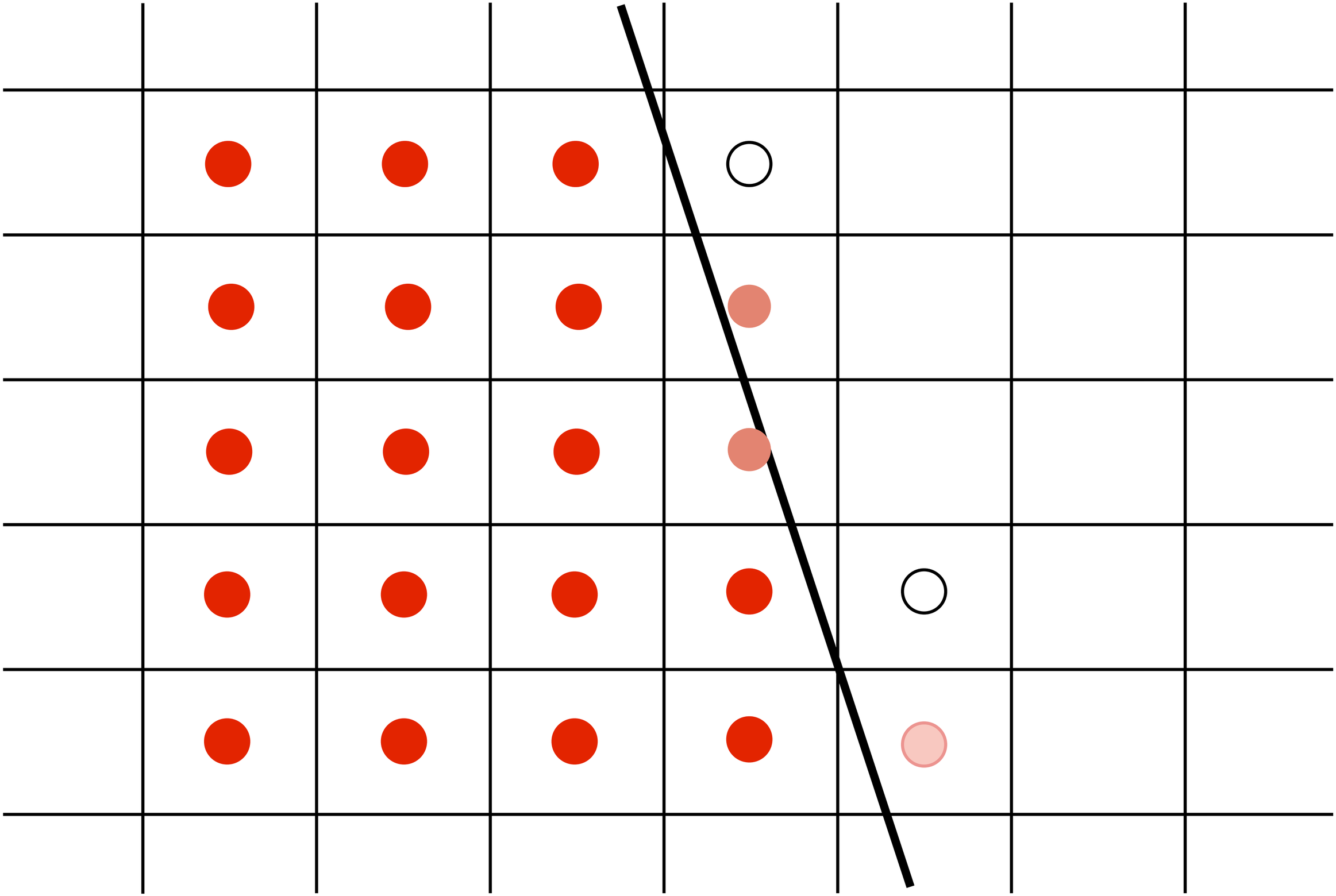
(Because a screen displays one sample value per screen pixel...)



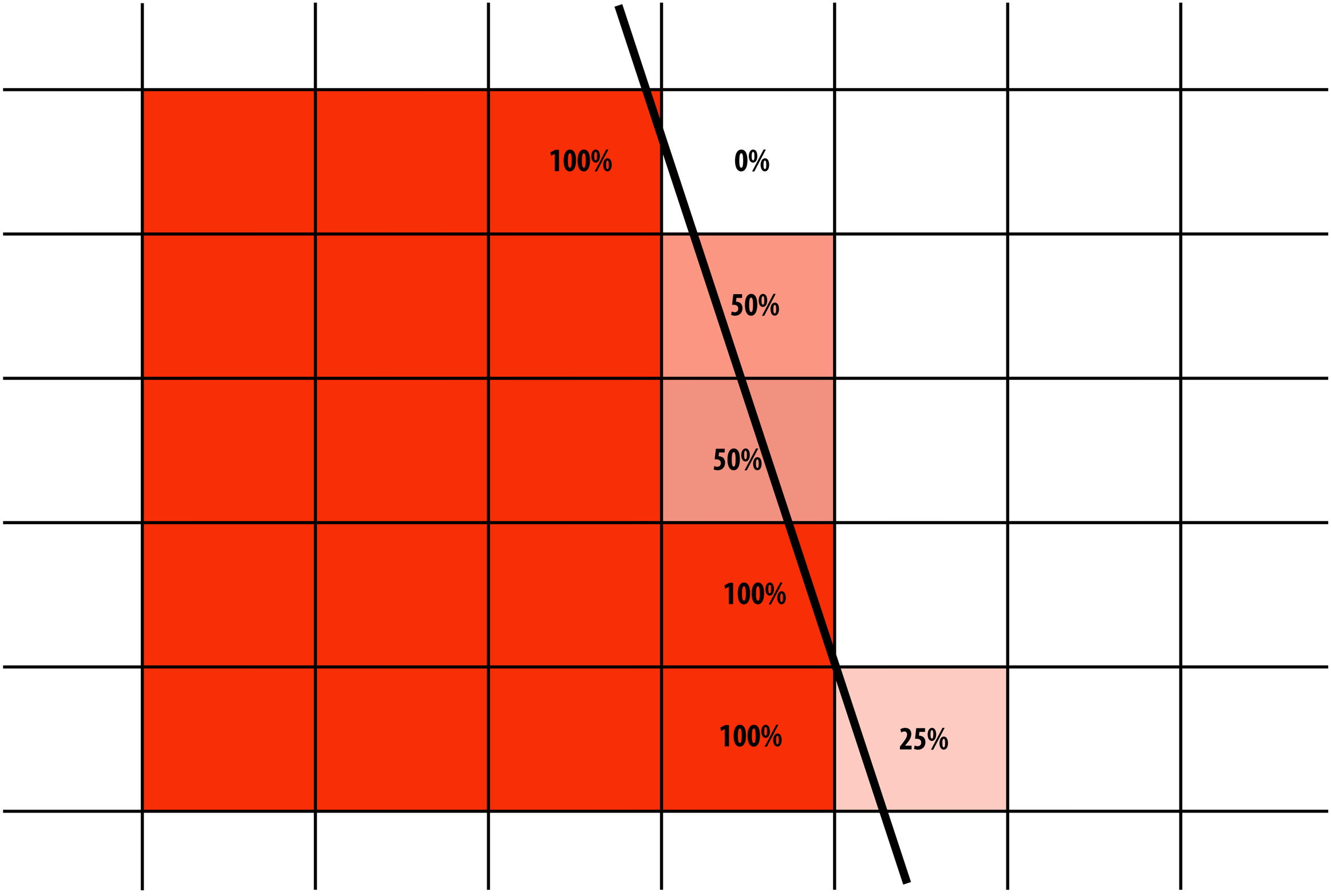
Resample to display's pixel rate (box filter)



Resample to display's pixel rate (box filter)

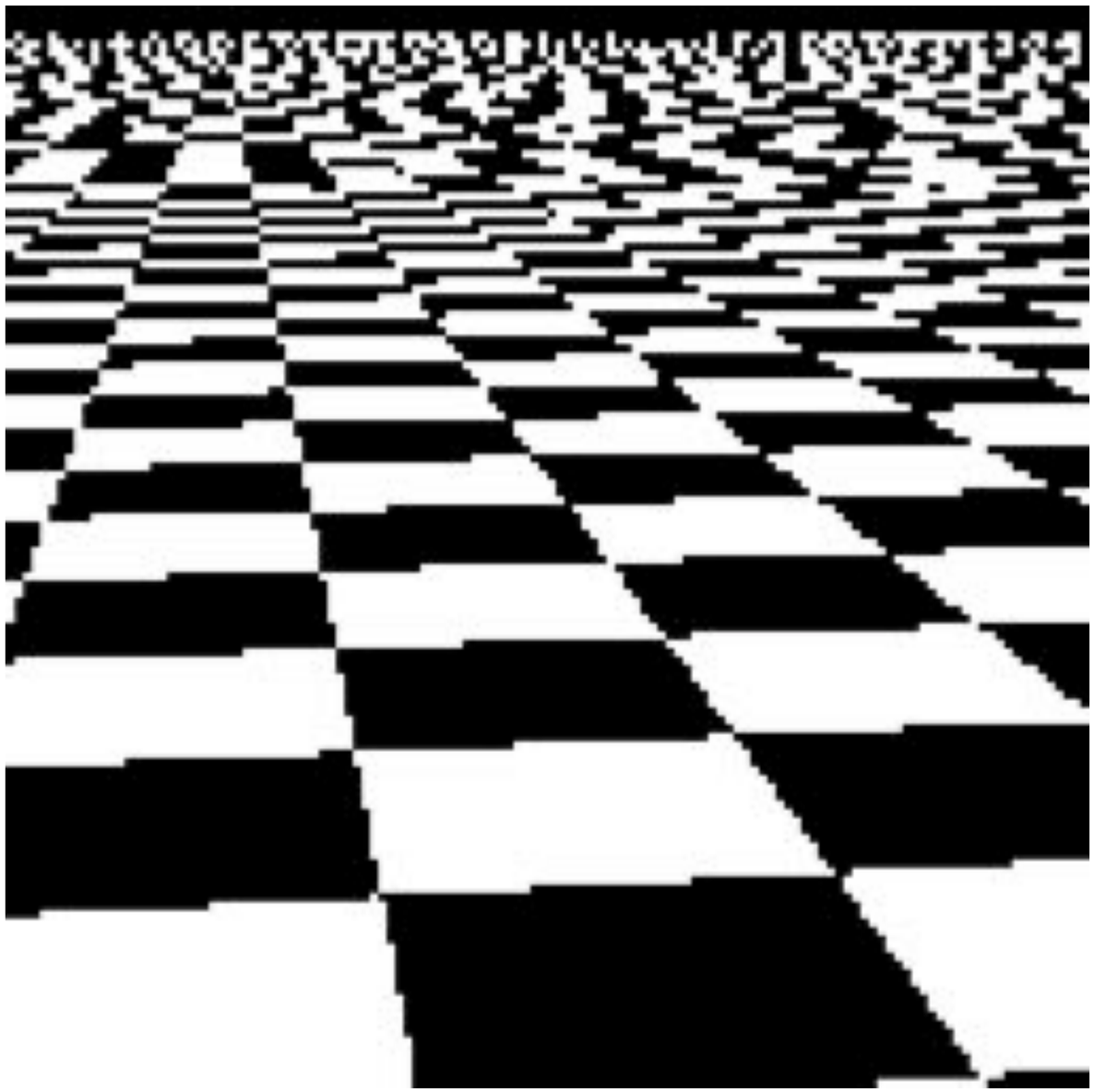


Displayed result (note anti-aliased edges)

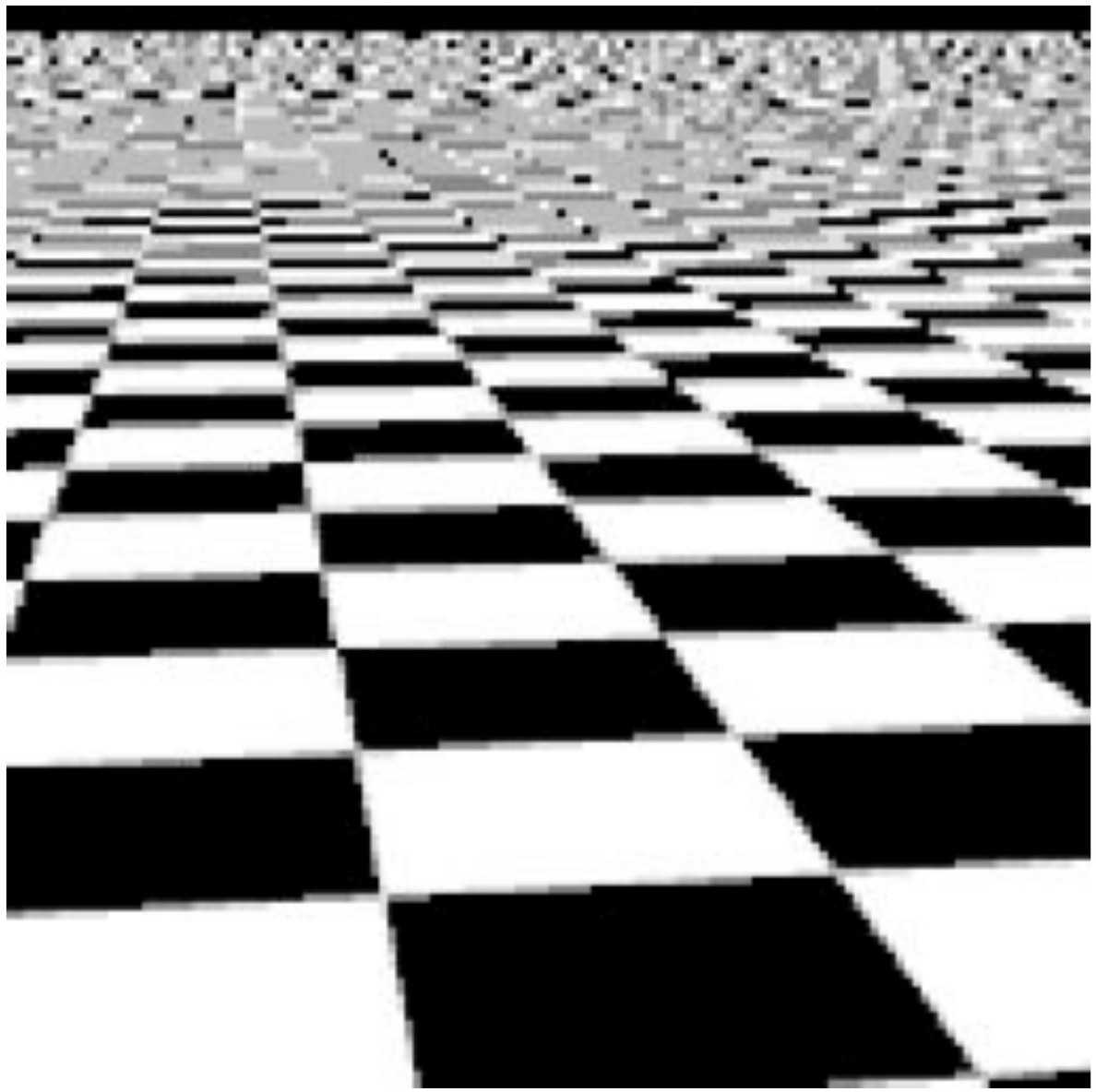


Recall: the real coverage signal was this

Single Sample vs. Supersampling

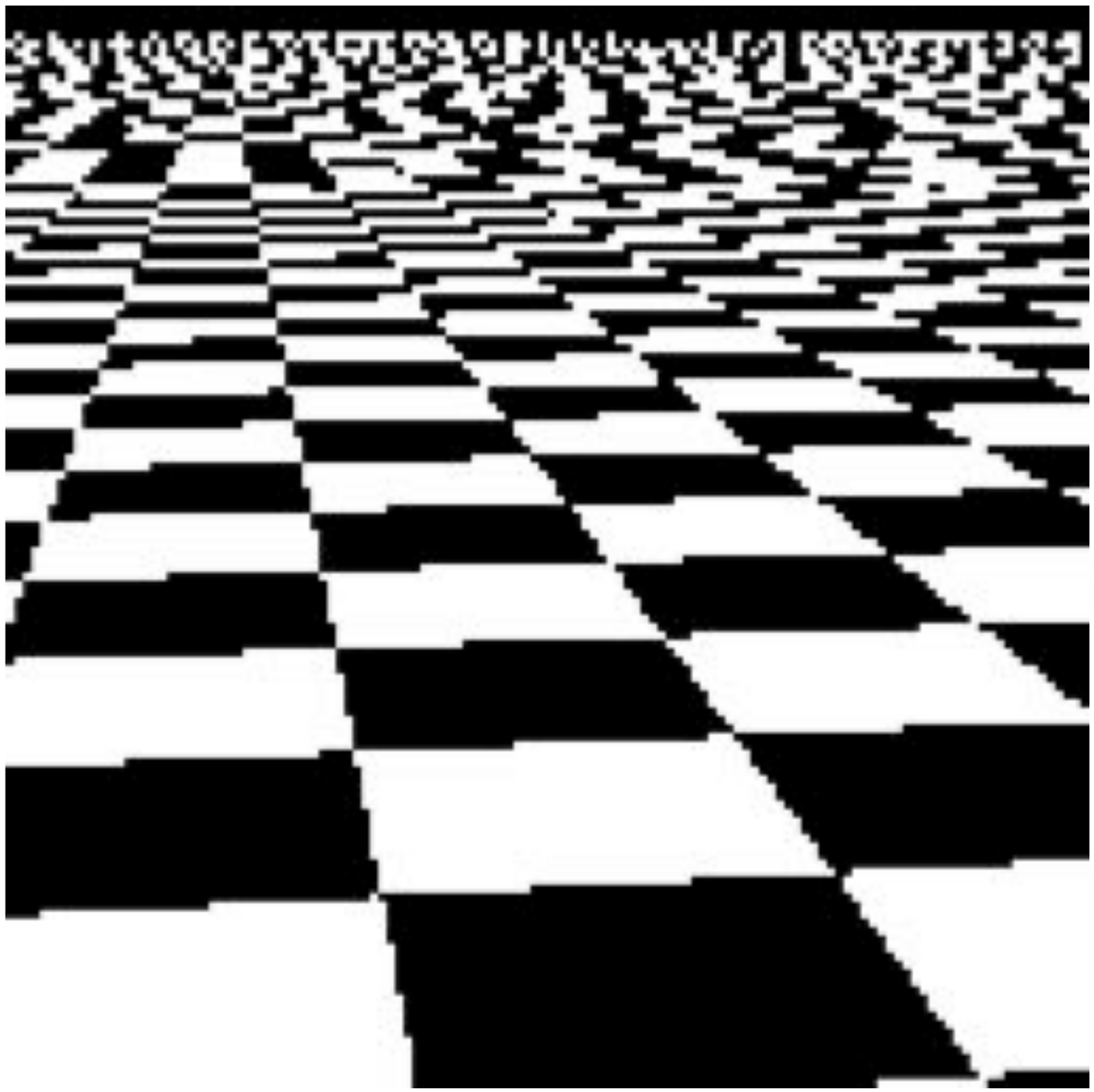


single sampling

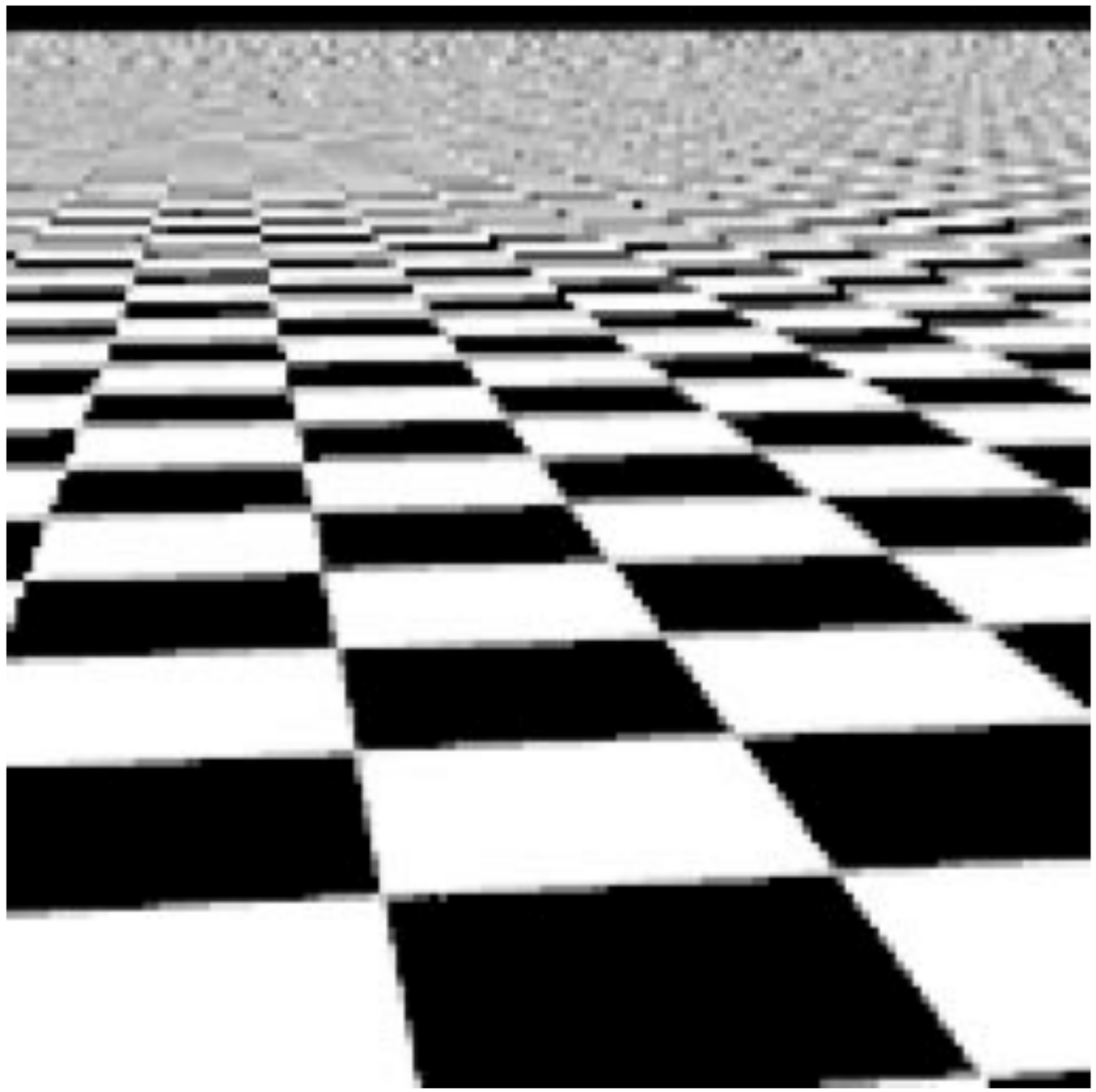


2x2 supersampling

Single Sample vs. Supersampling

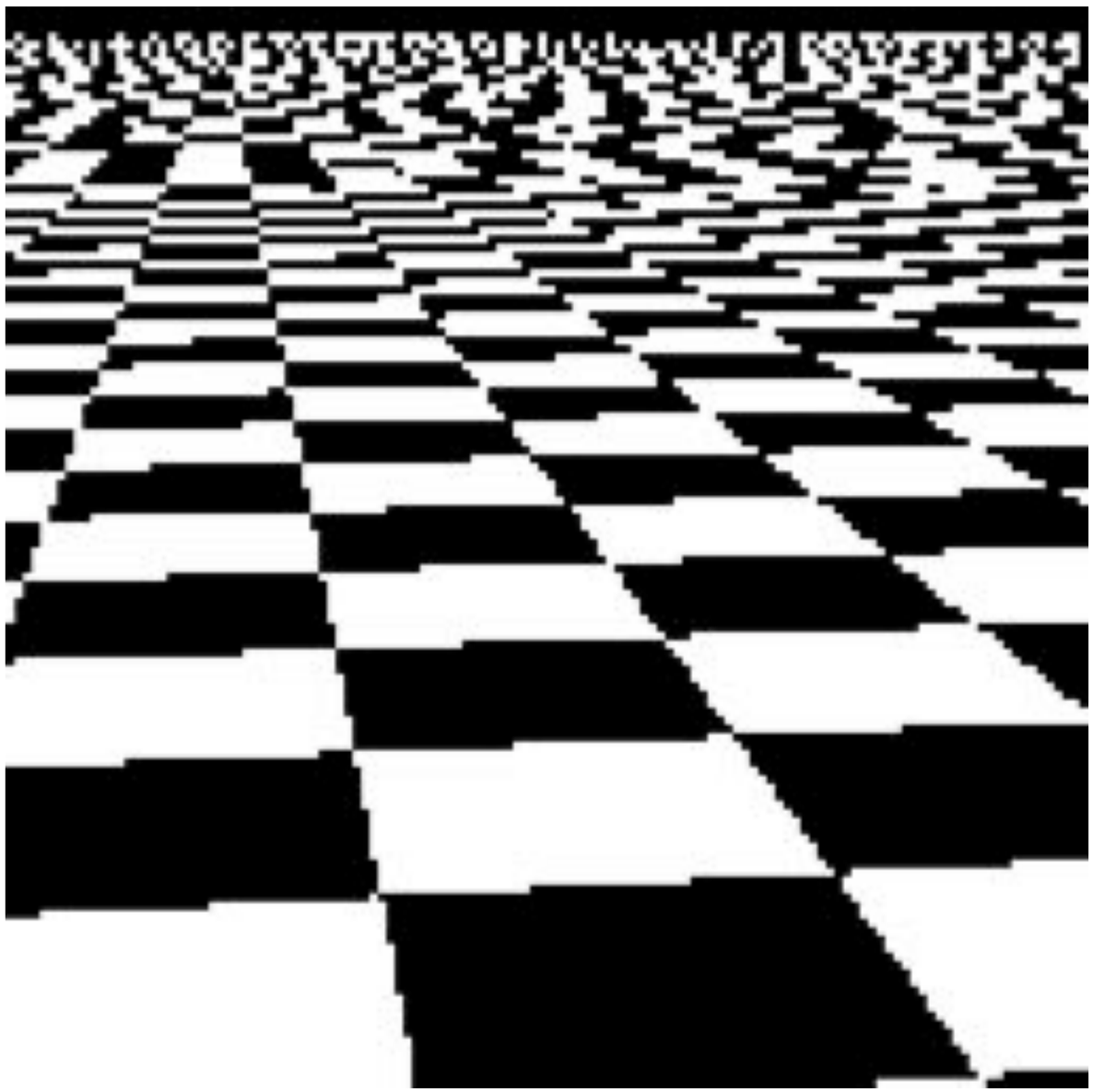


single sampling

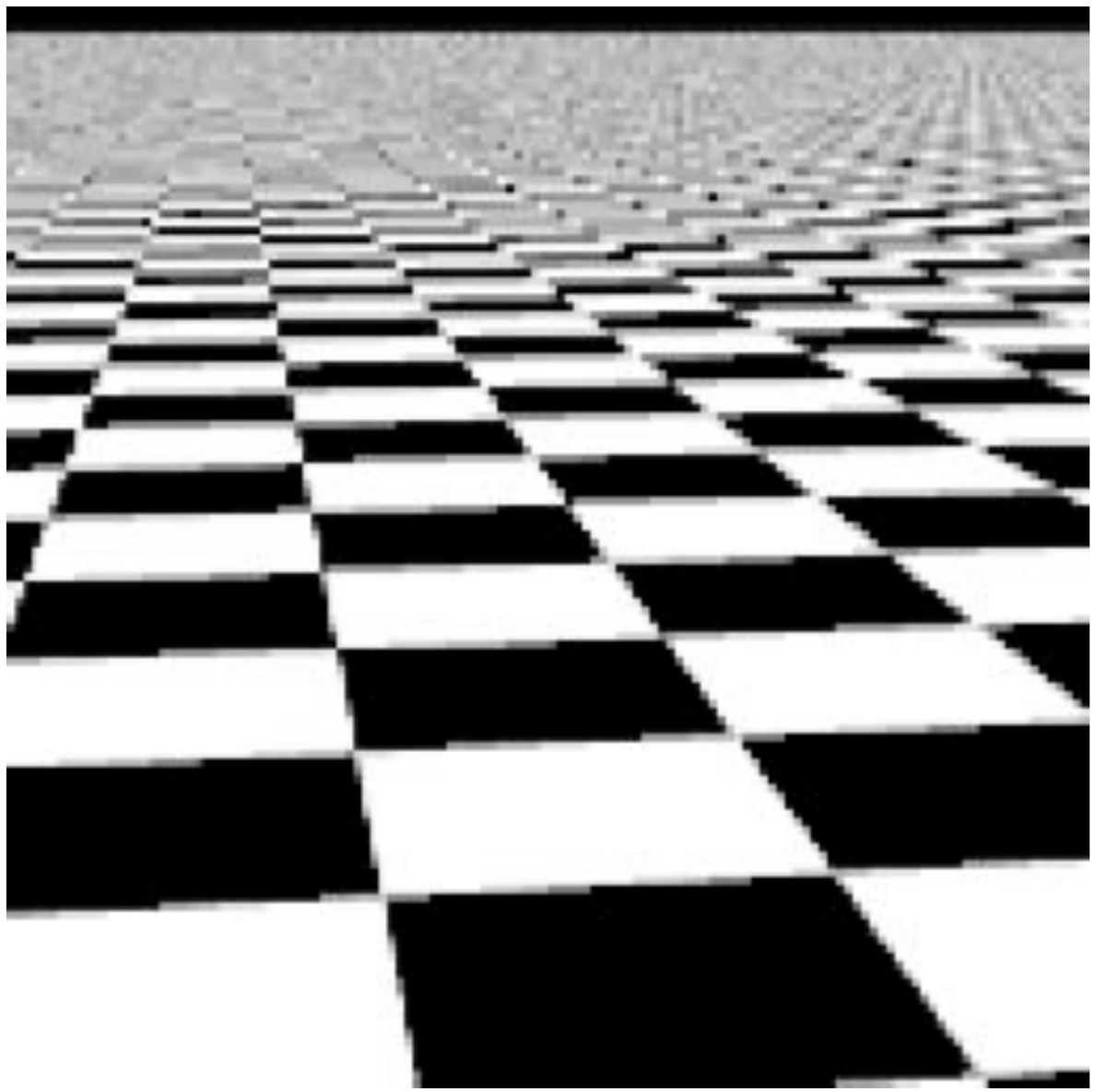


4x4 supersampling

Single Sample vs. Supersampling



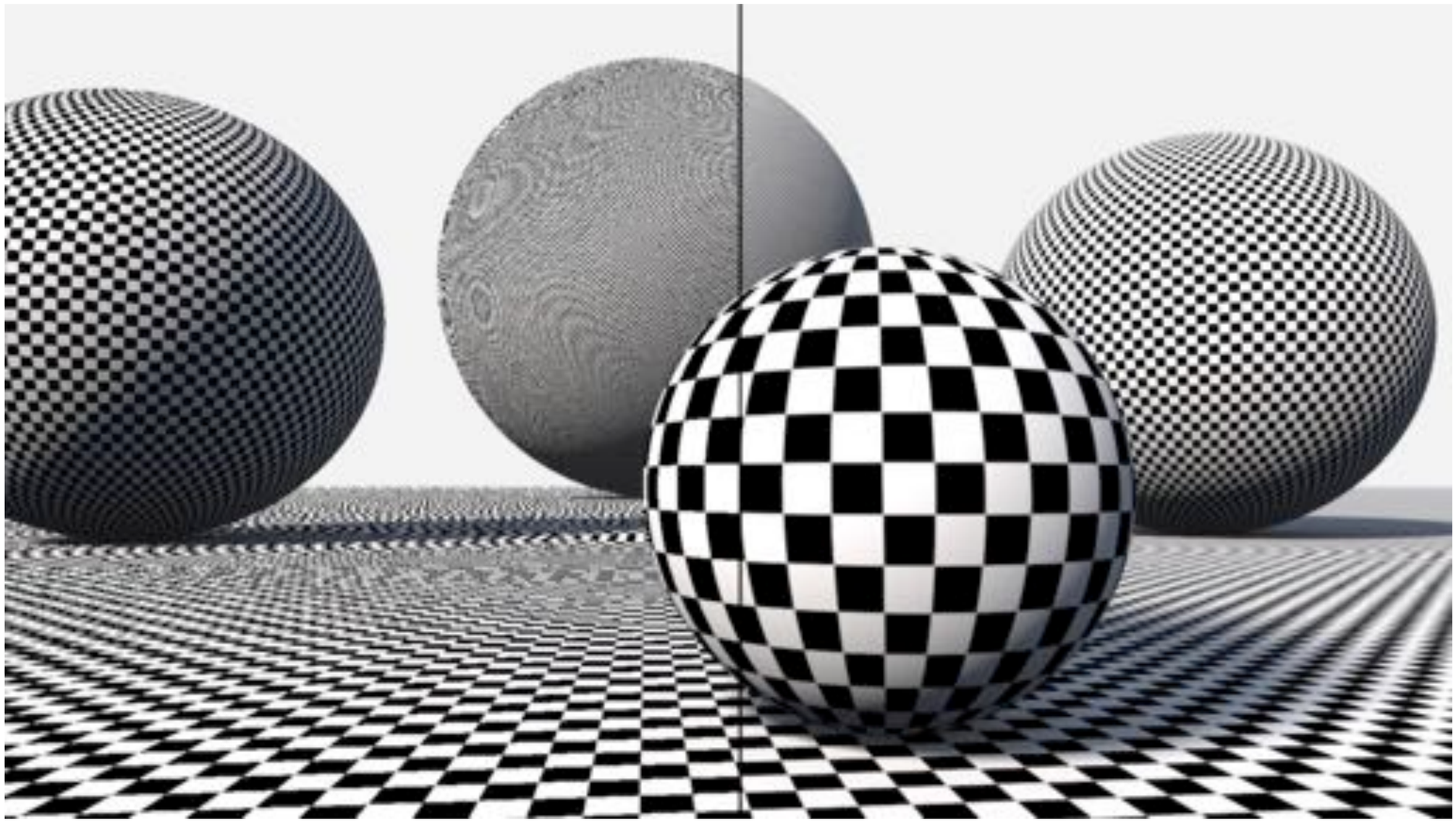
single sampling



32x32 supersampling

Checkerboard — Exact Solution

In very special cases we can compute the exact coverage:



Such cases are extremely rare—want solutions that will work in the general case!

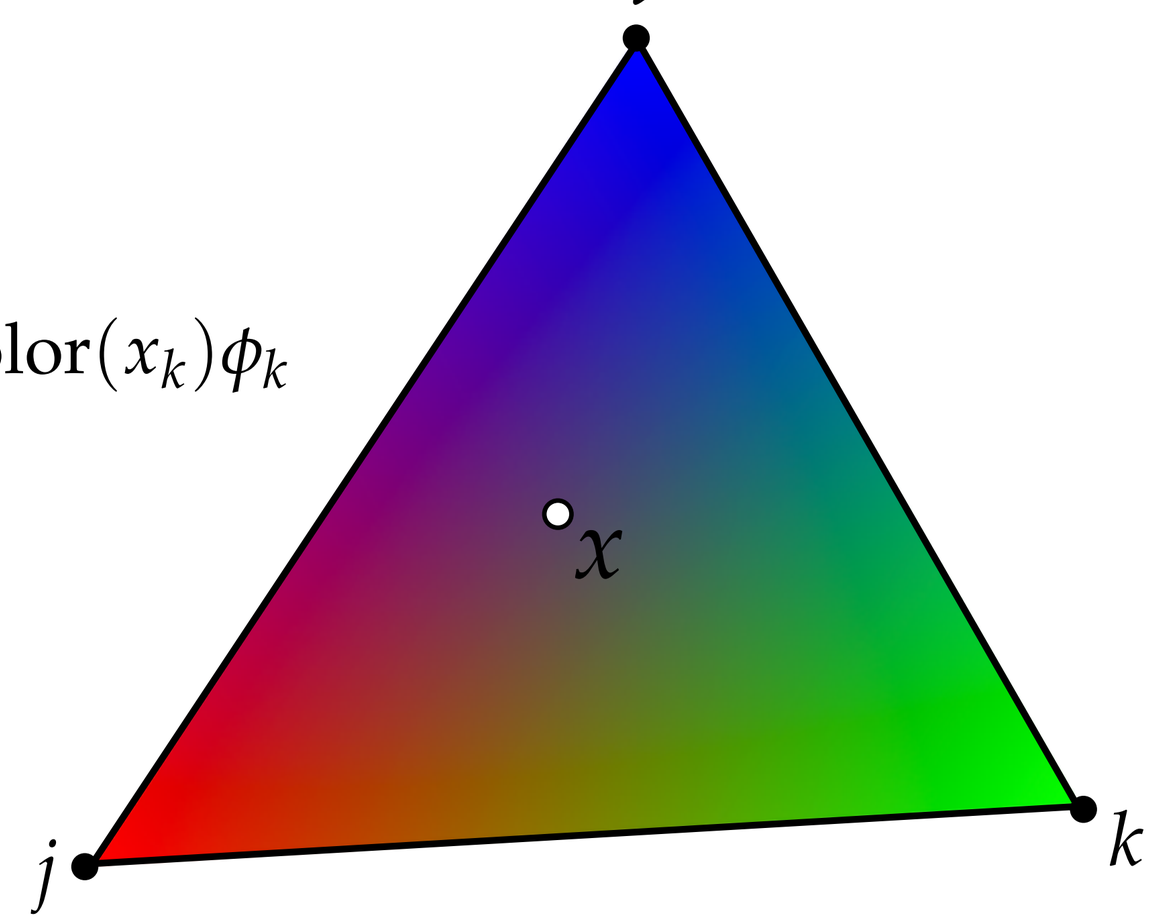
See: Inigo Quilez, *"Filtering the Checkerboard Pattern"* & Apodaca et al, *"Advanced Renderman"* (p. 273)

Part II: Perspective correct interpolation (or .. using barycentric coordinates properly)

Remember Barycentric Coordinates?

- Values of the three functions $\phi_i(\mathbf{x}), \phi_j(\mathbf{x}), \phi_k(\mathbf{x})$ for a given point are called barycentric coordinates
- Can be computed from triangle area ratios, as byproduct of half-plane tests used for rasterization, among other techniques
- Can be used to interpolate any attribute associated with vertices. (color*, texture coordinates, etc.)

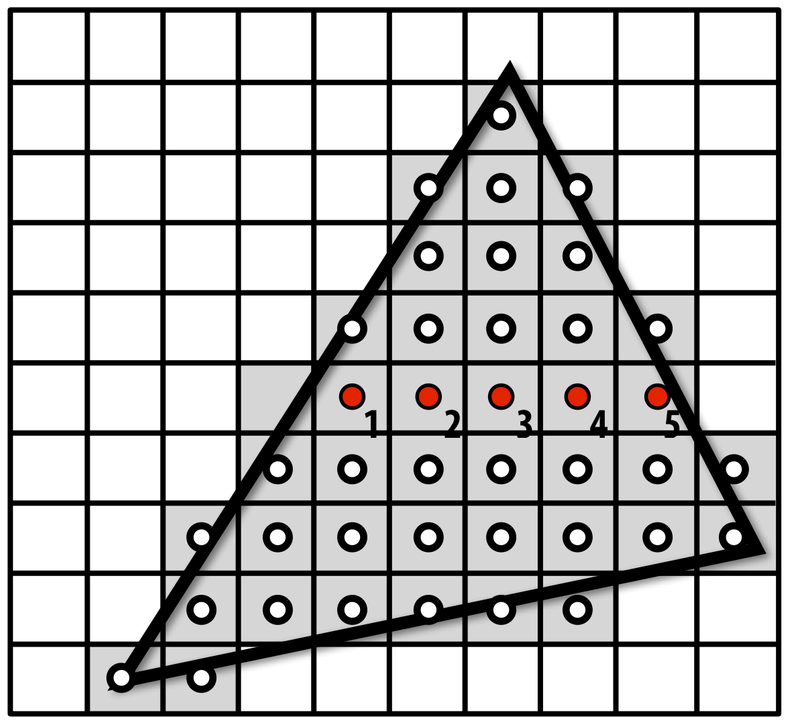
$$\text{color}(x) = \text{color}(x_i)\phi_i + \text{color}(x_j)\phi_j + \text{color}(x_k)\phi_k$$



There is a small difficulty with this simple and beautiful idea…

In general, interpolation in screen space using barycentric coordinates will give an incorrect **answer**

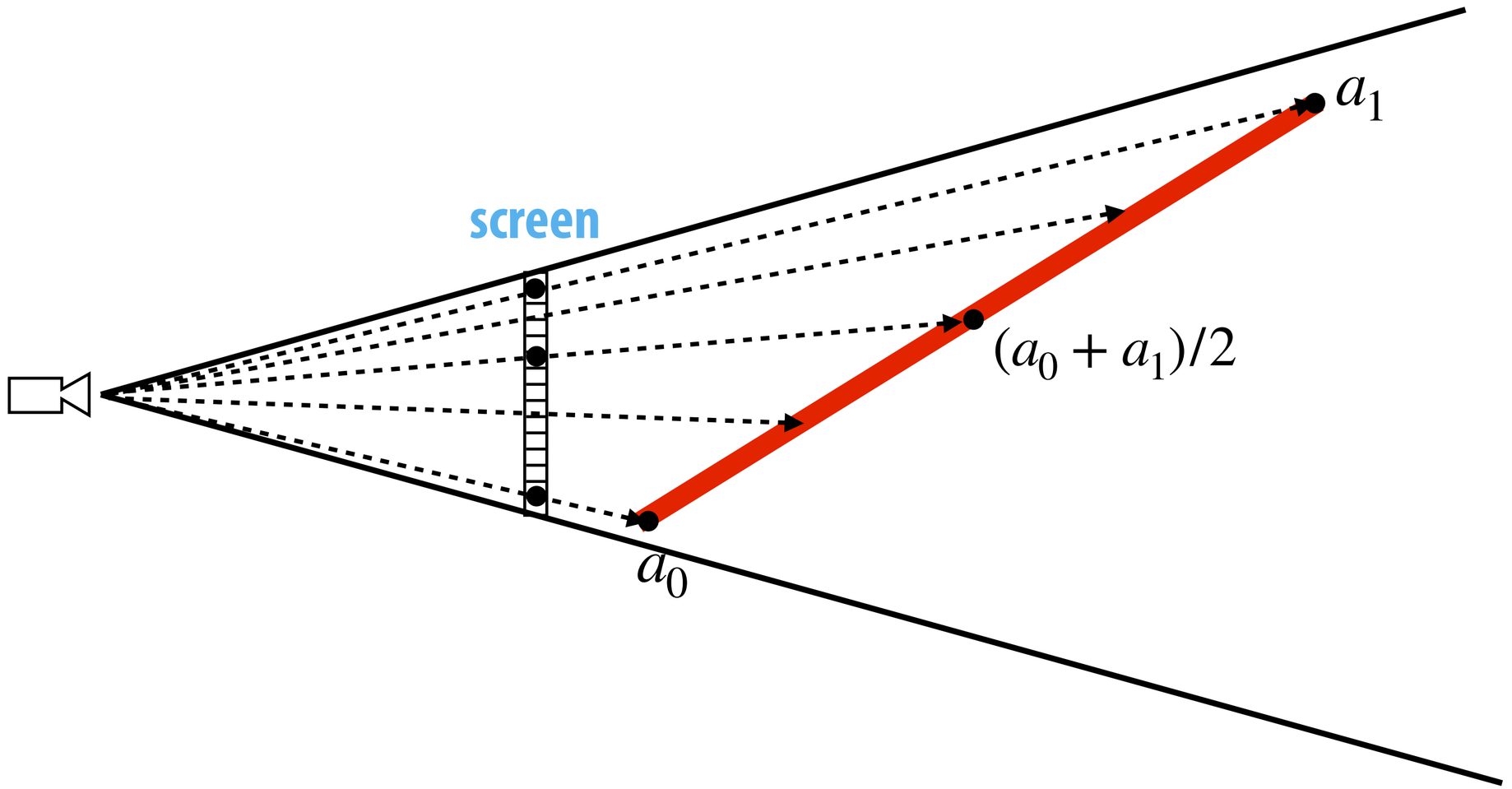
Why?



It has to do with perspective projection…

Perspective-incorrect interpolation

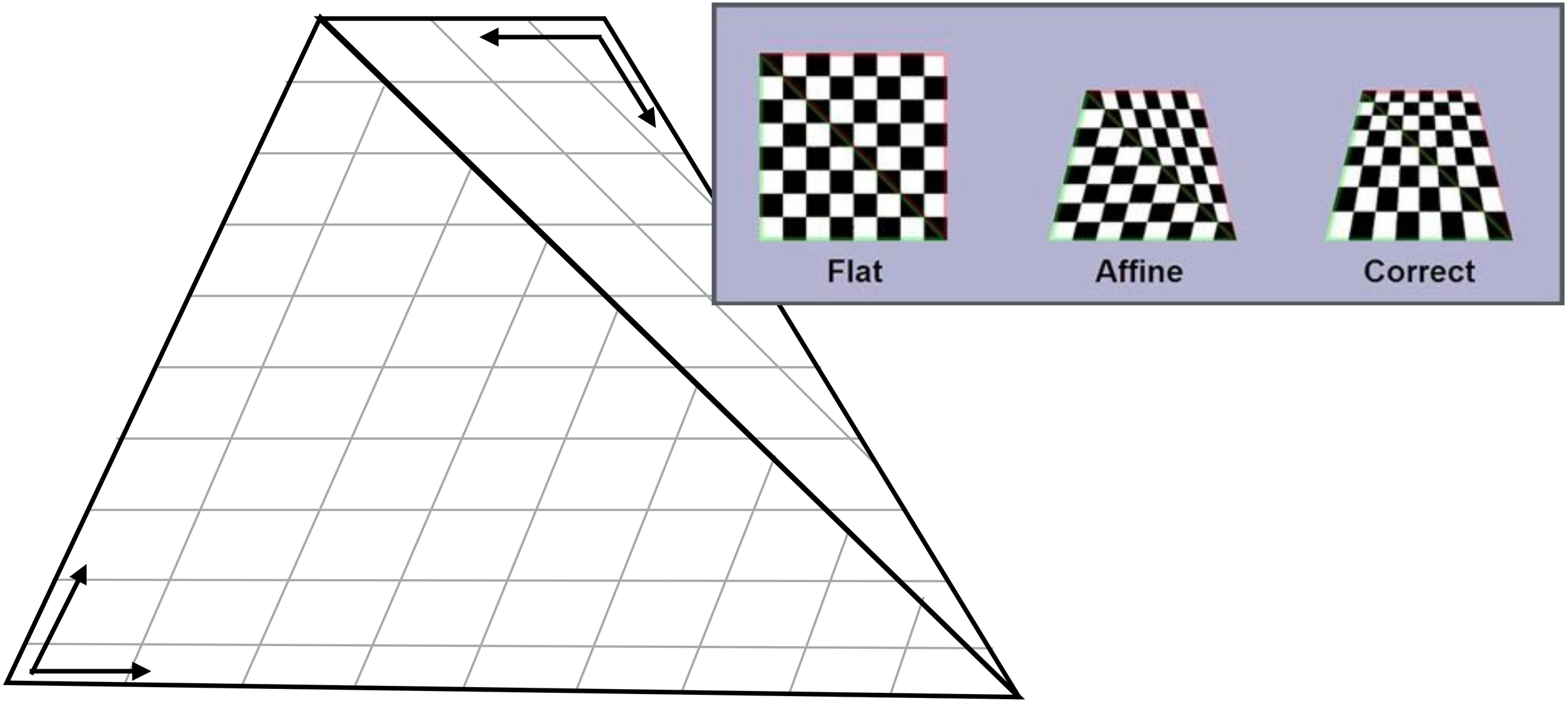
Due to perspective projection (homogeneous divide), barycentric interpolation of values on a triangle with different depths is not an affine function of screen XY coordinates



Want to interpolate attribute values linearly in 3D object space, not image space.

Example: perspective incorrect interpolation

Consider a quadrilateral split into two triangles:



If we compute barycentric coordinates using 2D (projected) coordinates, leads to (derivative) discontinuity in interpolation where quad was split

Perspective Correct Interpolation

- Goal: interpolate some attribute ϕ at vertices
- Basic recipe:
  - Compute depth z at each vertex
  - Evaluate $Z := 1/z$ and $P := \phi/z$ at each vertex
  - Interpolate Z and P using standard (2D) barycentric coords
  - At each fragment, divide interpolated P by interpolated Z to get final value

For a derivation, see Low, "Perspective-Correct Interpolation"

Part III: Texture mapping + Upsampling and downsampling with the mipmap

Texture Mapping

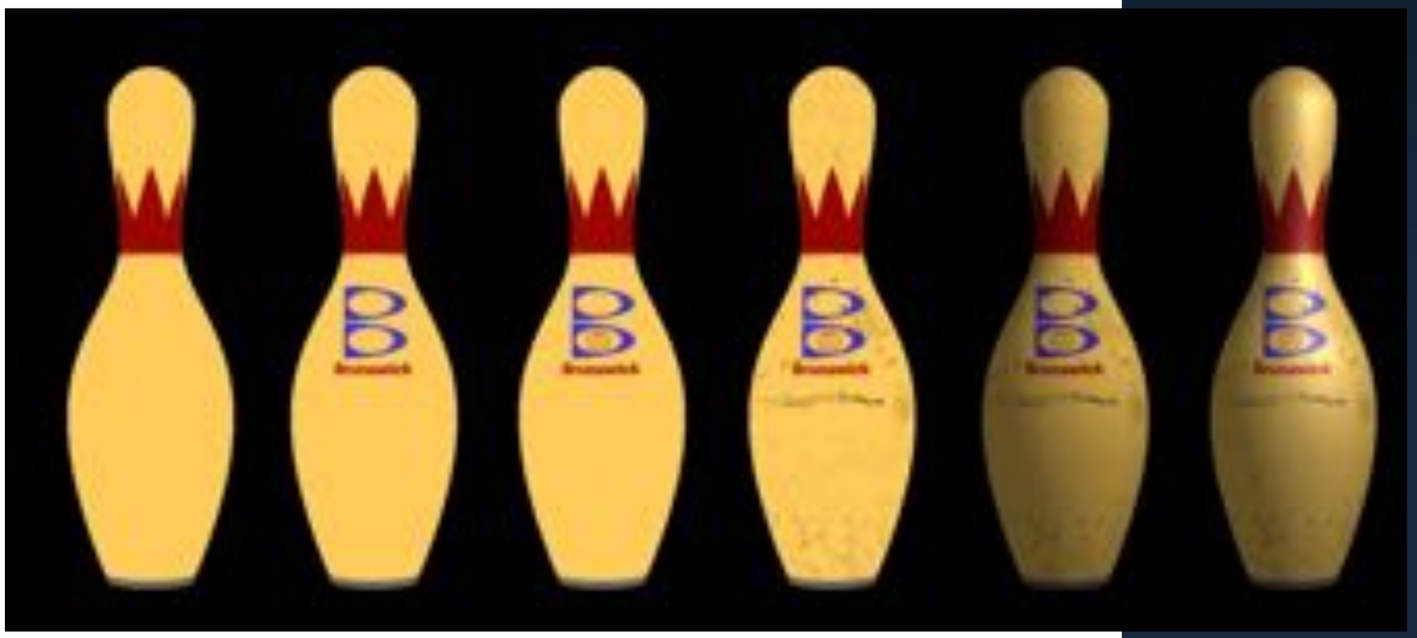
Many uses of texture mapping

Define variation in surface reflectance

Pattern on ball

Wood grain on floor

Describe surface material properties



Multiple layers of texture maps for color, logos, scratches, etc.

Normal & Displacement Mapping

normal mapping

Use texture value to perturb surface normal to "fake" appearance of a bumpy surface

displacement mapping

dice up surface geometry into tiny triangles & offset positions according to texture values (note bumpy silhouette and shadow boundary)

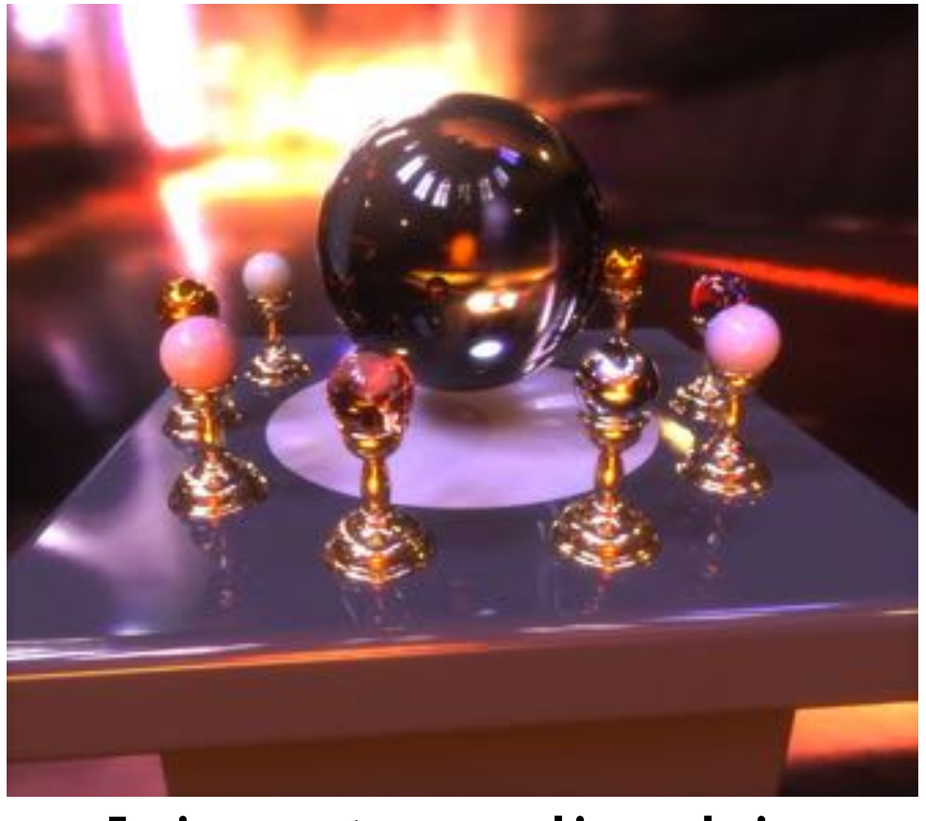
Represent precomputed lighting and shadows

Original model

With ambient occlusion

Extracted ambient occlusion map

Grace Cathedral environment map

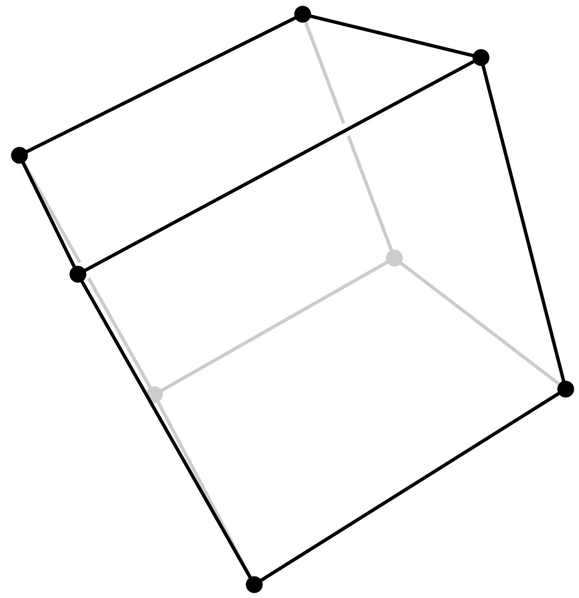


Environment map used in rendering

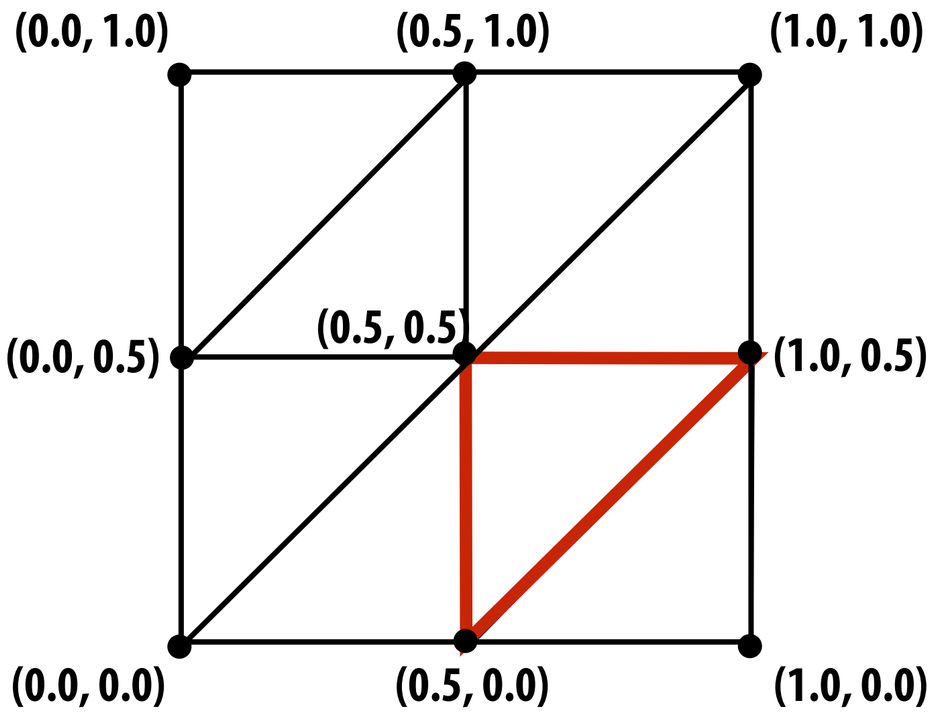
Texture coordinates

- "Texture coordinates" define a mapping from surface coordinates to points in texture domain
- Often defined by linearly interpolating texture coordinates at triangle vertices

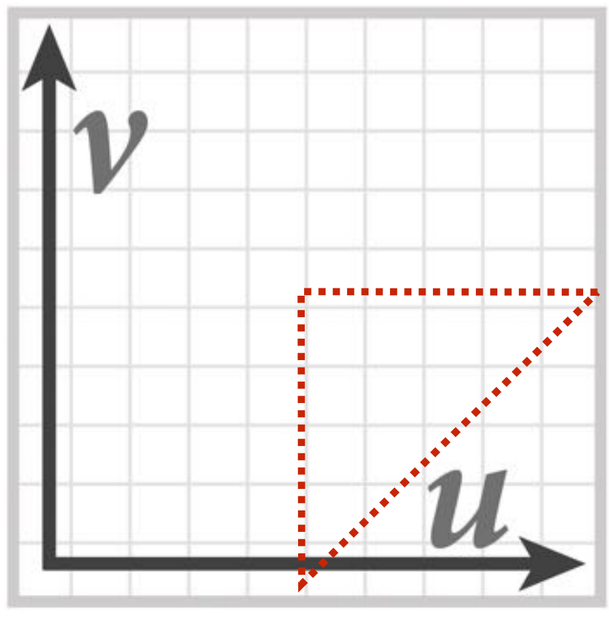
example: texture this cube



Suppose each cube face is split into eight triangles, with texture coordinates (u,v) at each vertex

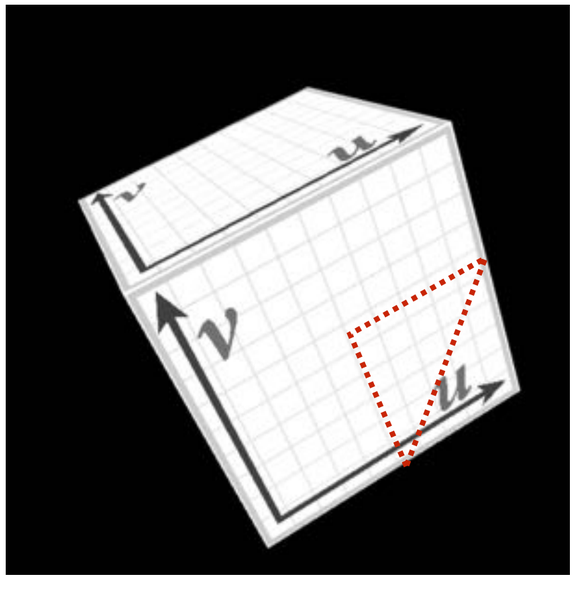


A texture on the $[0,1]^2$ domain can be specified by a 2048x2048 image



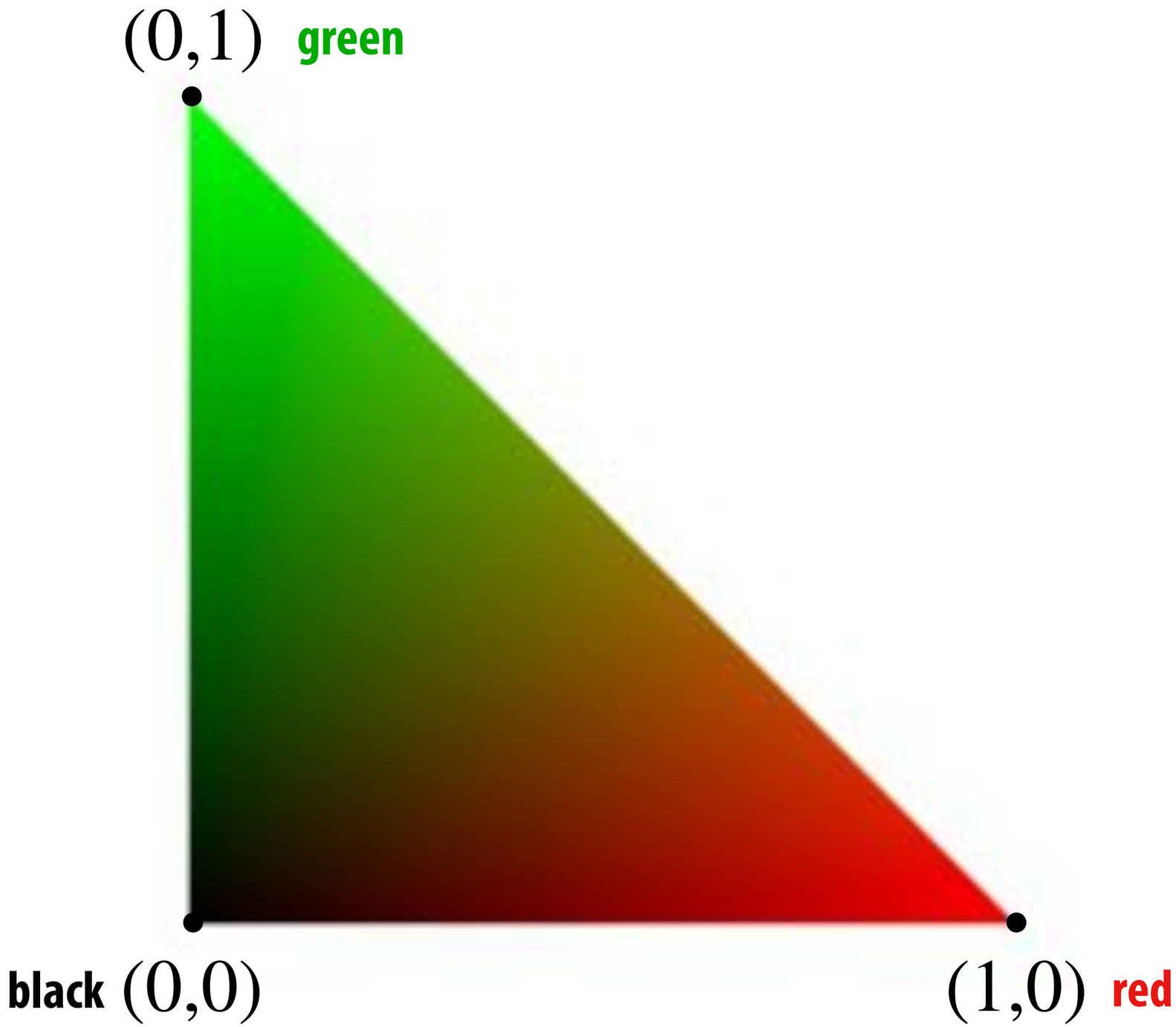
(location of highlighted triangle in texture space shown in red)

Linearly interpolating texture coordinates & "looking up" color in texture gives this image:



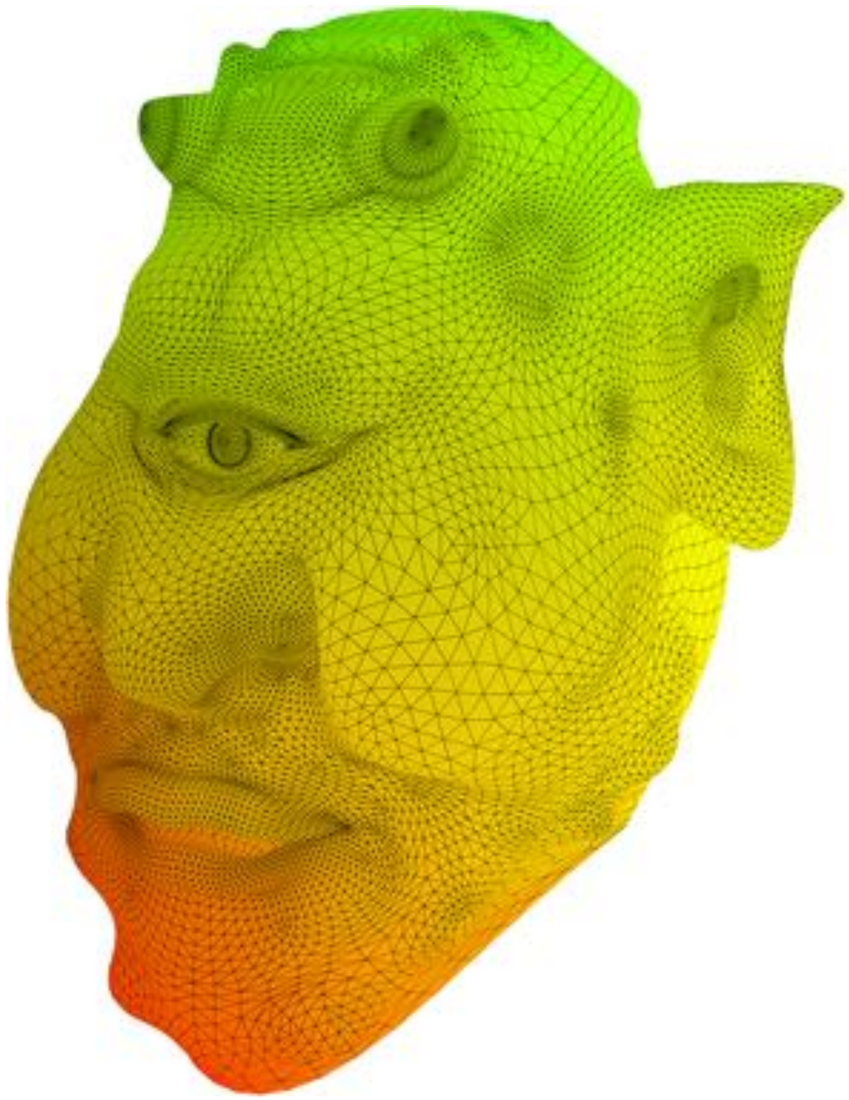
Visualization of texture coordinates

Associating texture coordinates (u, v) with colors helps to visualize mapping

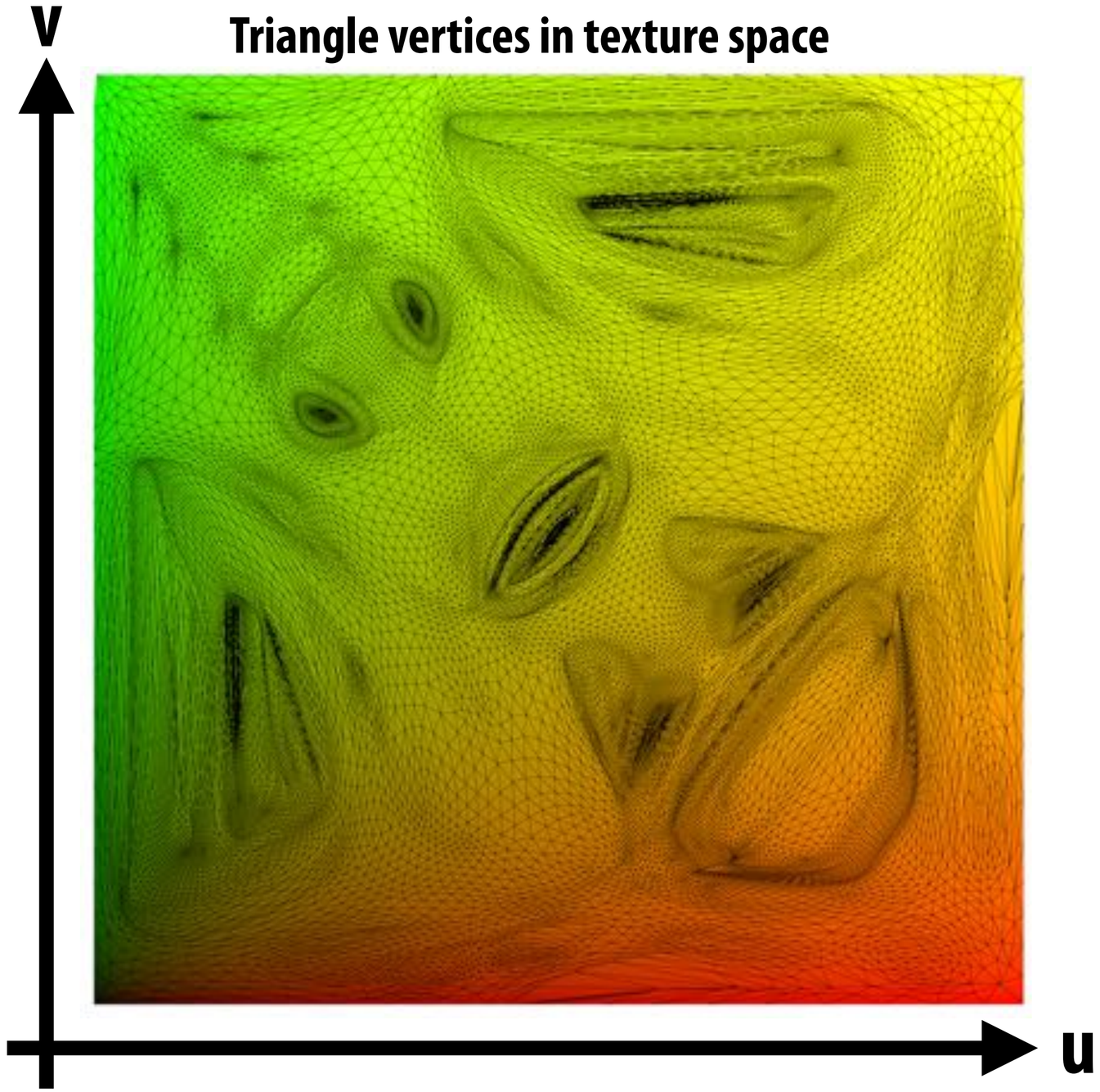


More complex mapping

Visualization of texture coordinates



Triangle vertices in texture space

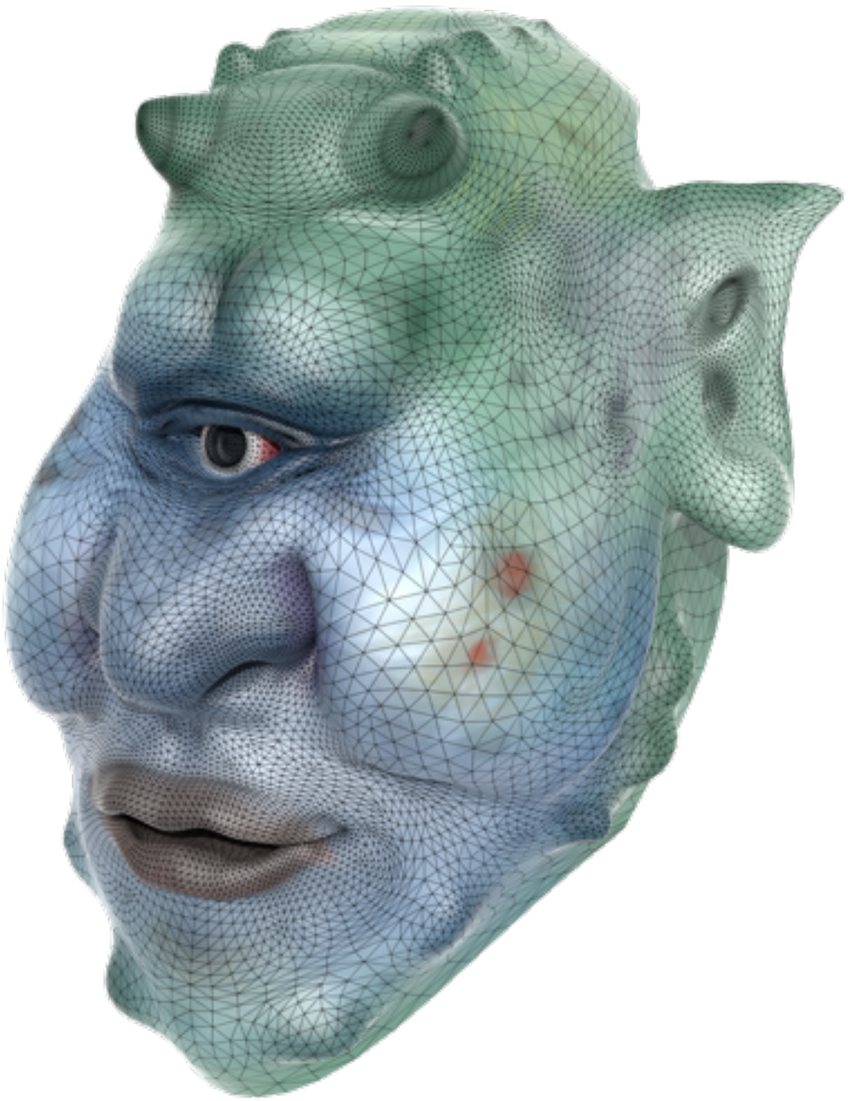


Each vertex has a coordinate (u,v) in texture space

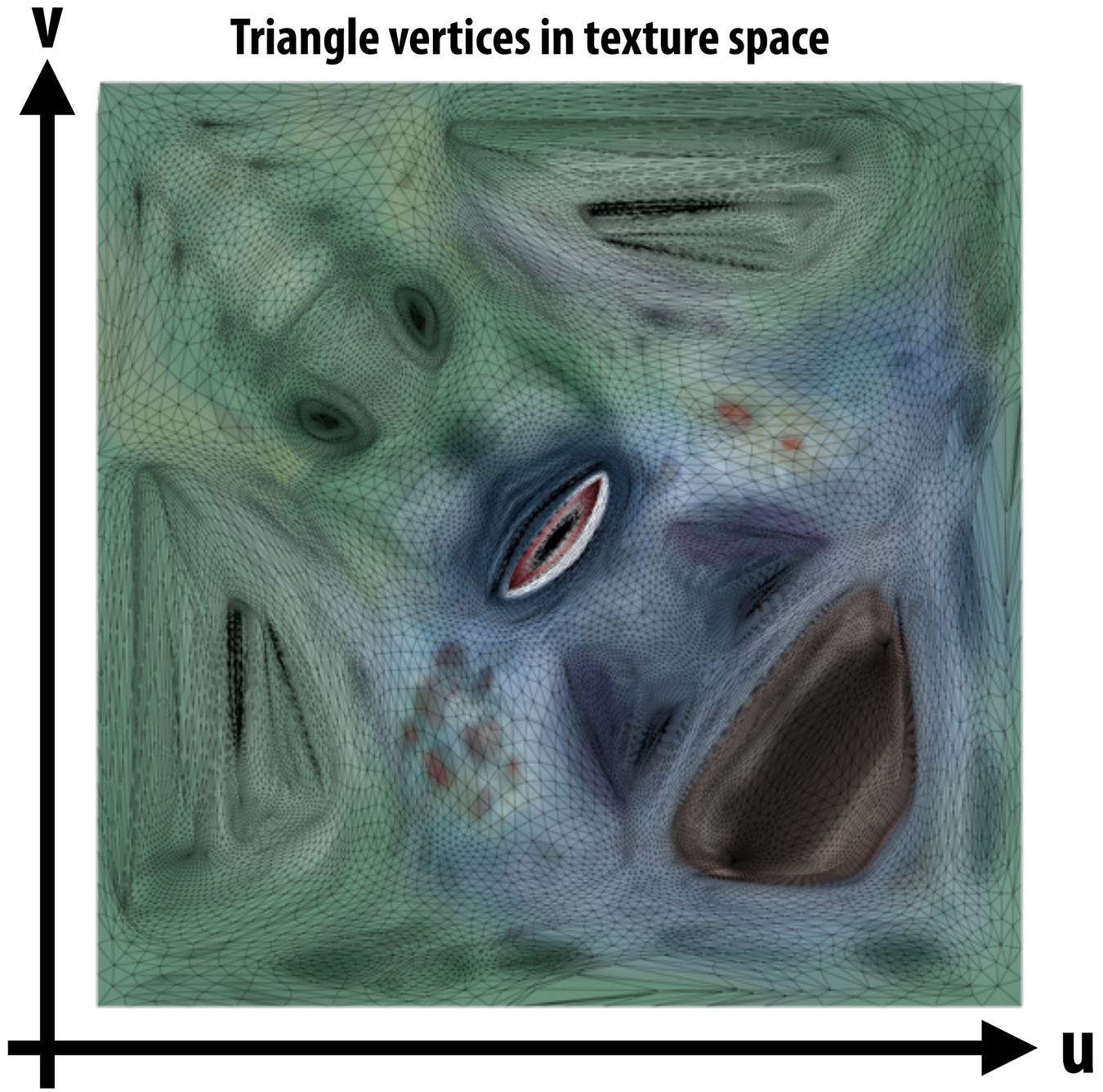
(Actually coming up with these coordinates is another story!)

Texture mapping adds detail

Rendered result



Triangle vertices in texture space



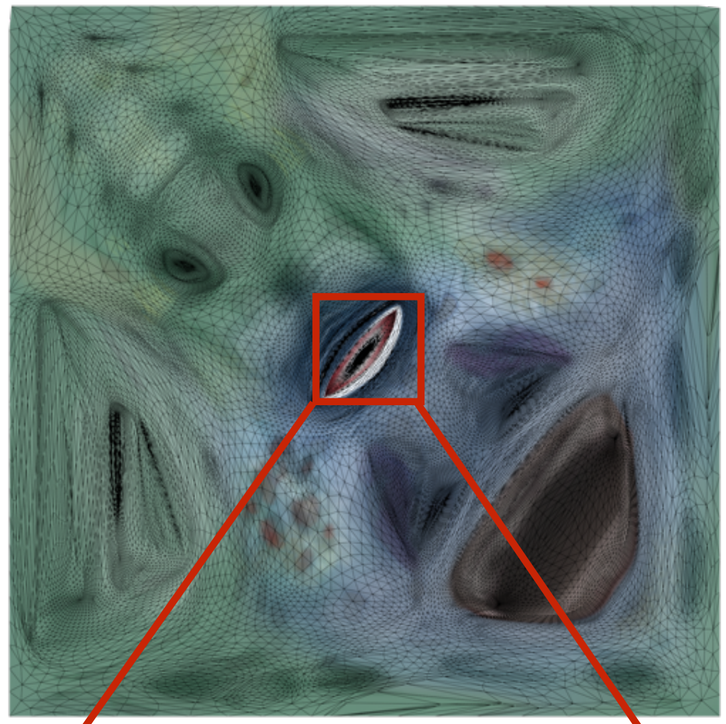
Each triangle "copies" a piece of the image back to the surface

Texture mapping adds detail

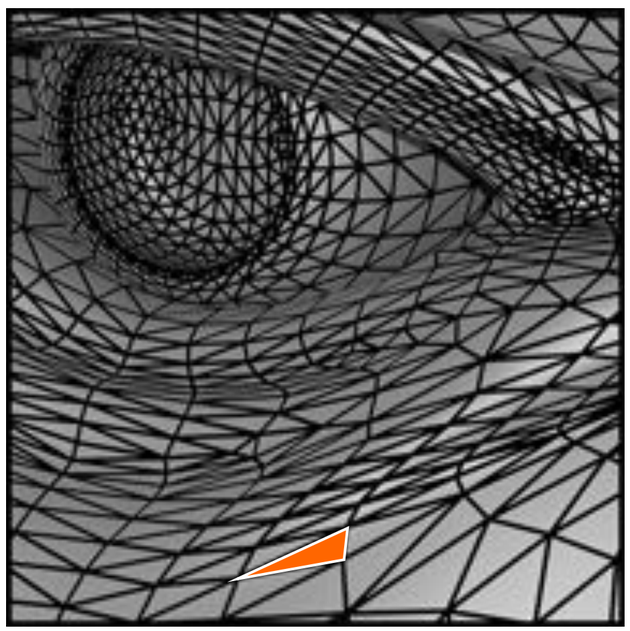
rendering without texture

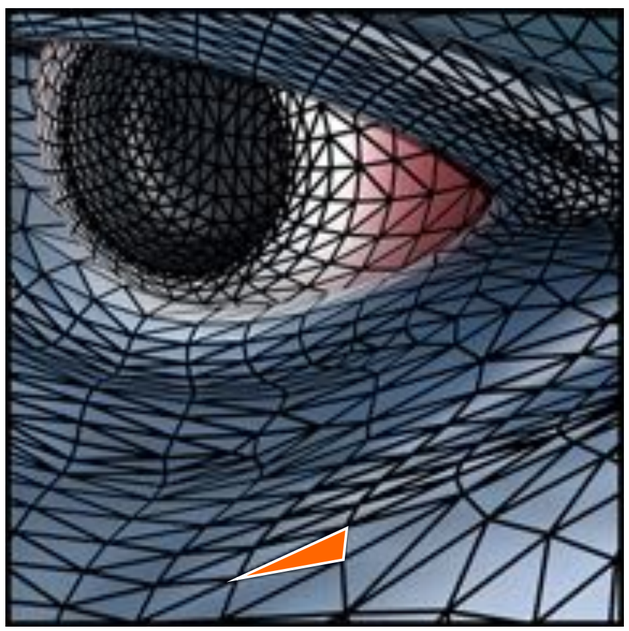
rendering with texture

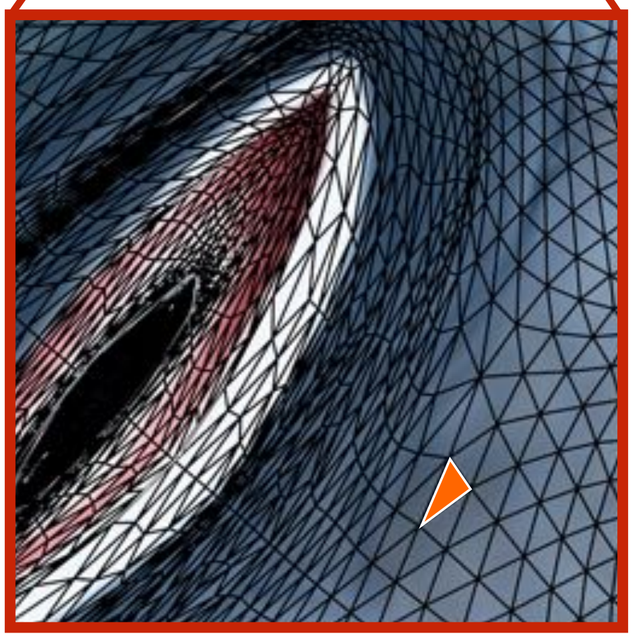
texture image



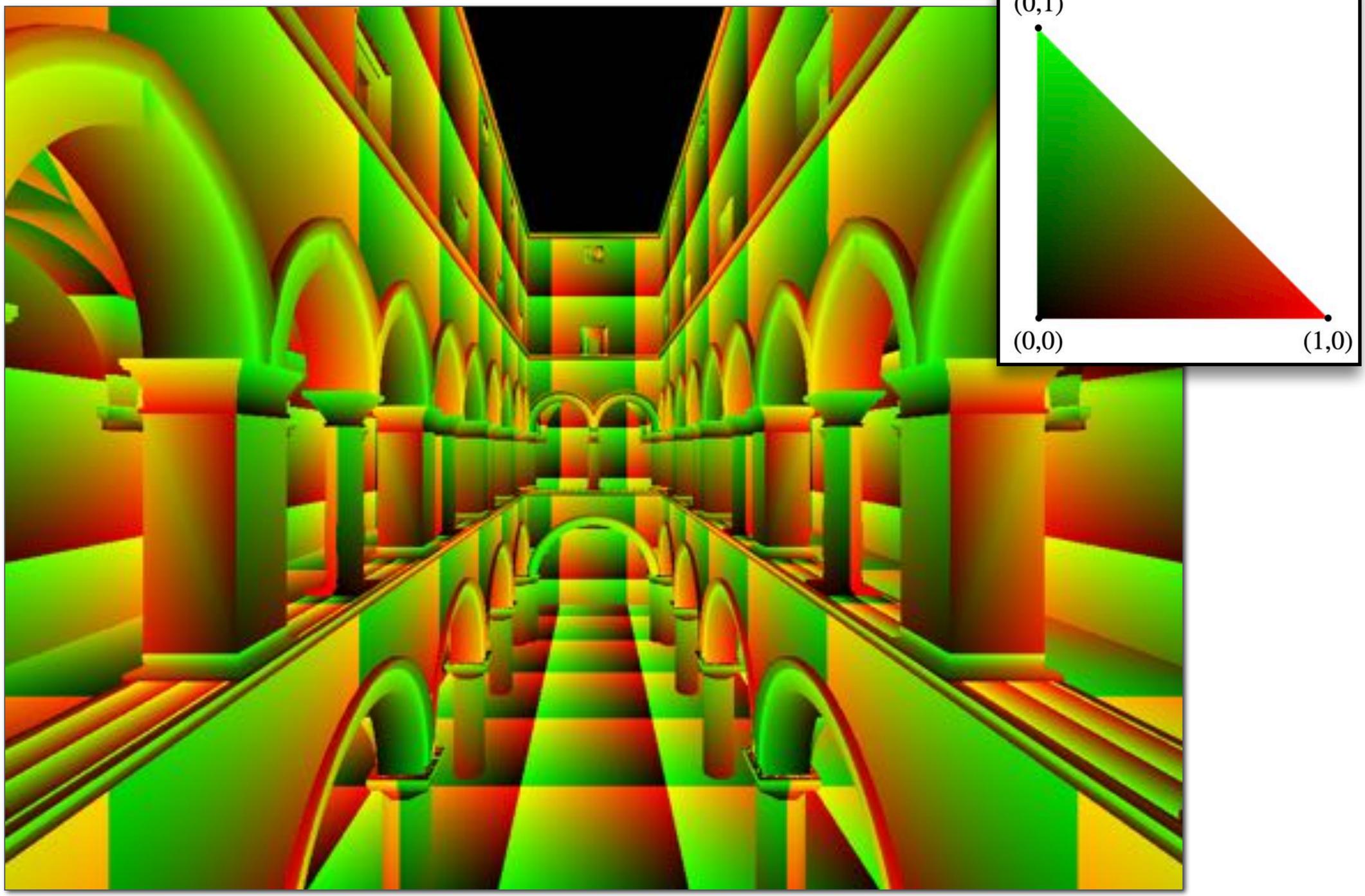
zoom







Another example: periodic coordinates



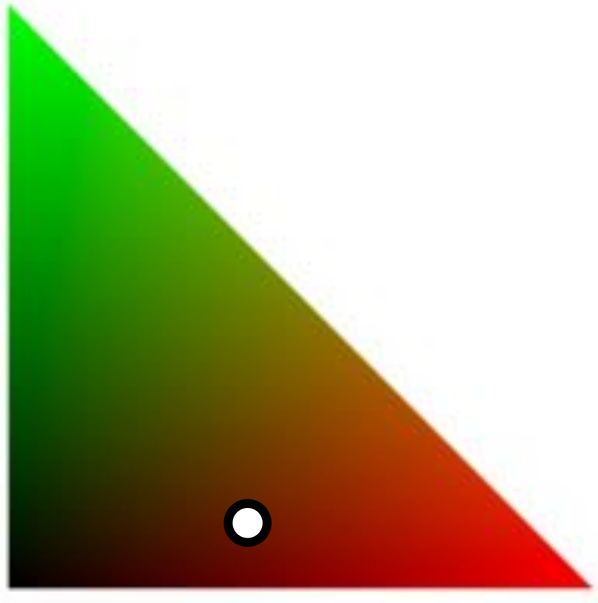
Q: Why do you think texture coordinates might repeat over the surface?

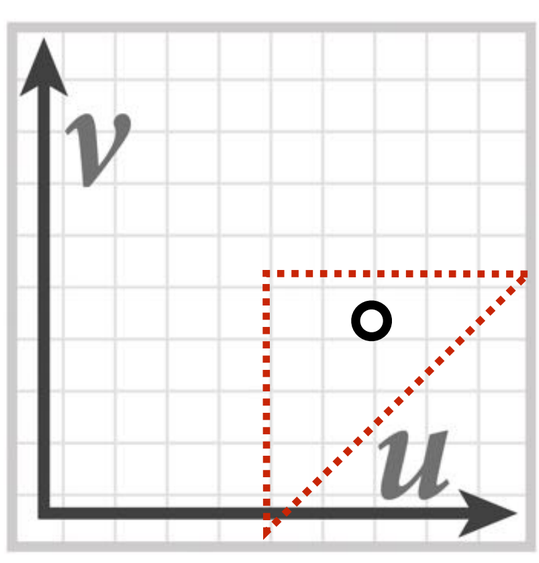
Textured Sponza

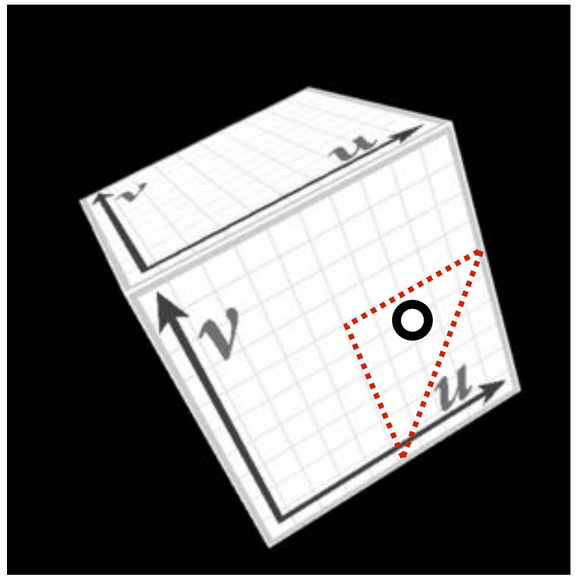
A: Want to tile a texture many times (rather than store a huge image!)

Texture Sampling 101

- **Basic algorithm for texture mapping:**
  - `for each pixel in the rasterized image:`
    - `interpolate` (u, v) `coordinates across triangle`
    - `sample (evaluate) texture at interpolated` (u, v)
    - `set color of fragment to sampled texture value`



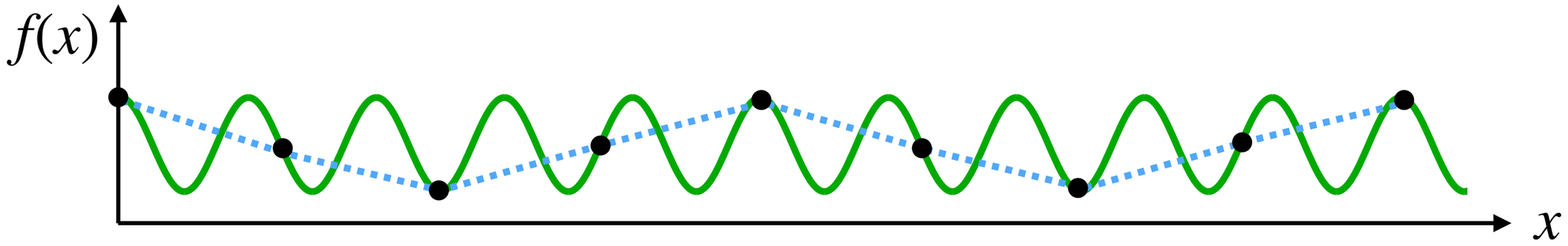


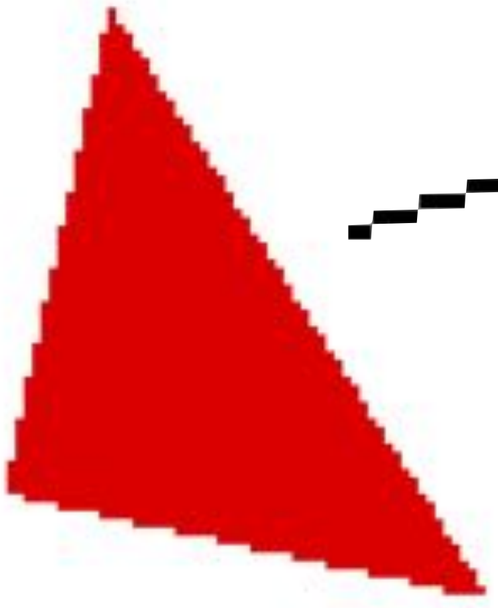


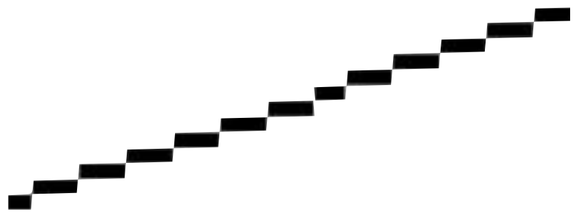
. . . sadly not this easy in general!

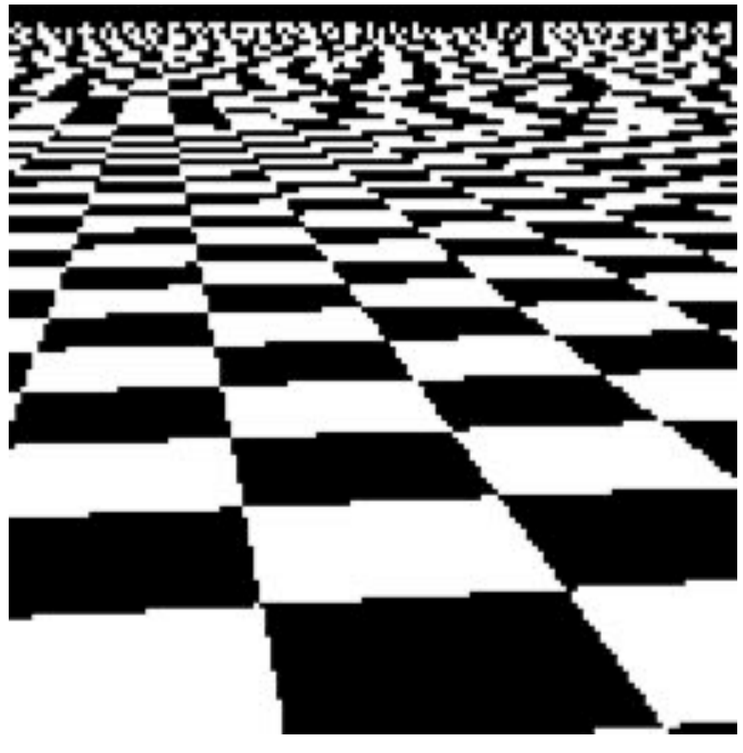
Recall: aliasing

Undersampling a high-frequency signal can result in aliasing





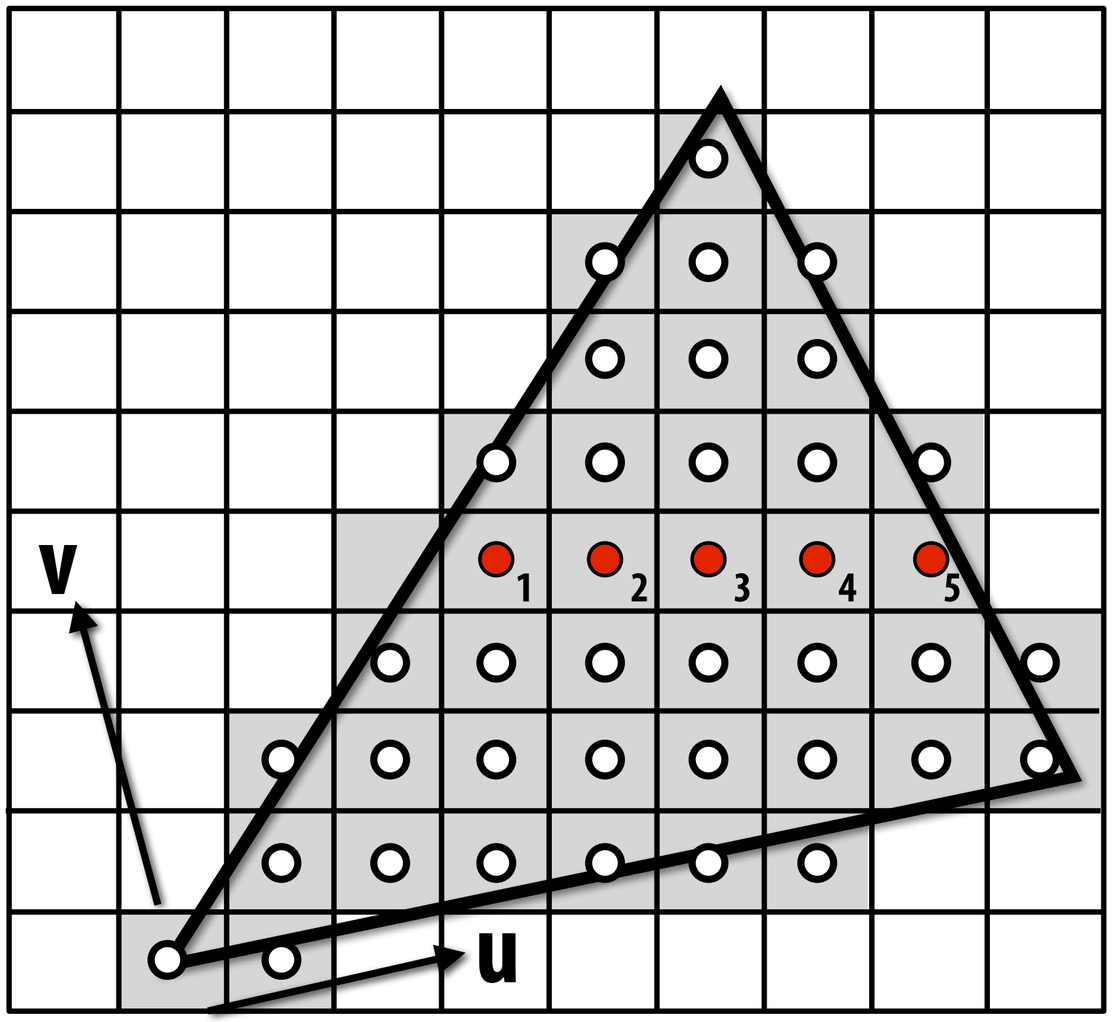




Visualizing texture samples

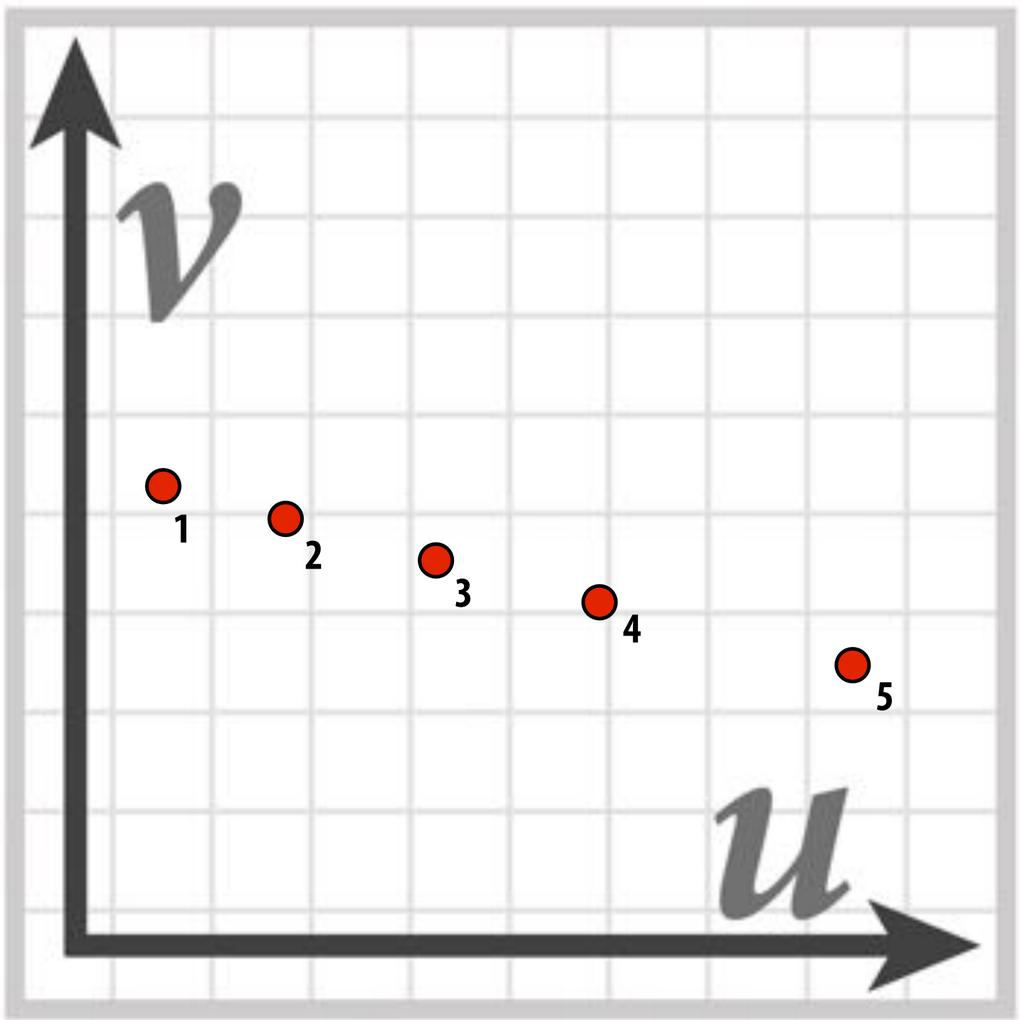
Since triangles are projected from 3D to 2D, pixels in screen space will correspond to regions of varying size & location in texture

sample positions in screen space



Sample positions are uniformly distributed in screen space (rasterizer samples triangle's appearance at these locations)

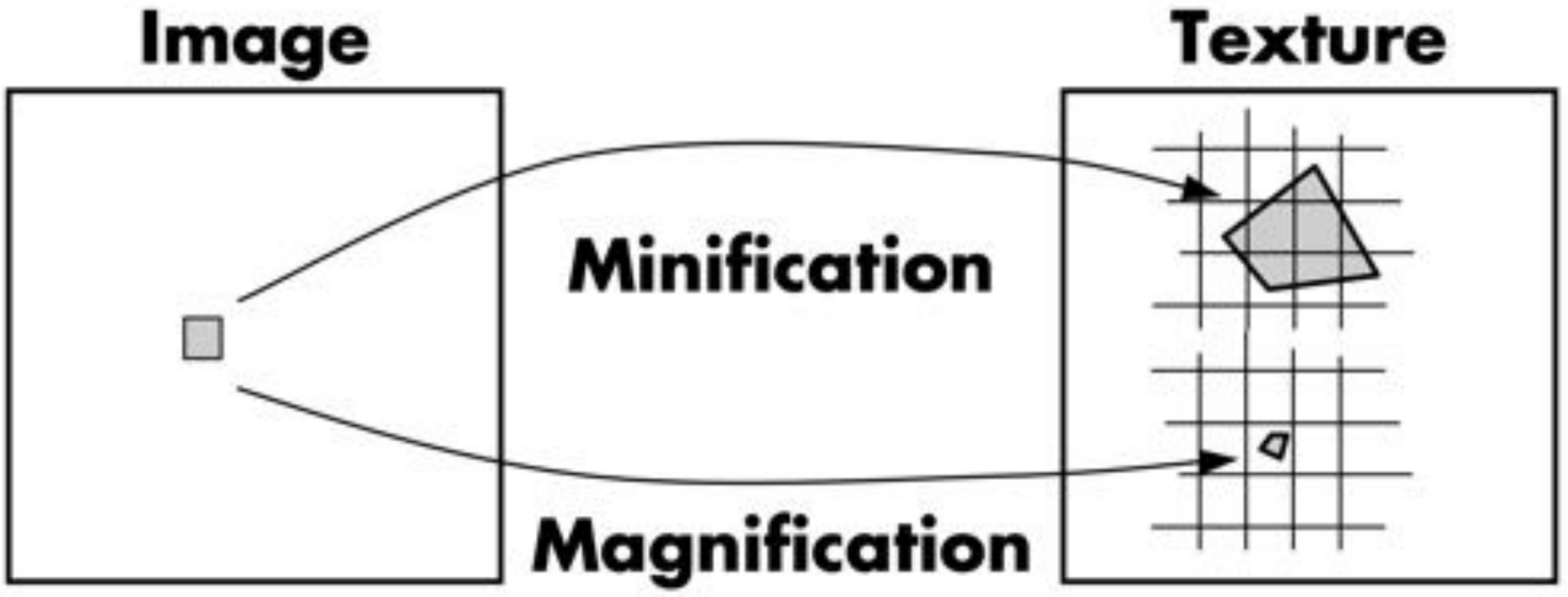
sample positions in texture space



Sample positions in texture space are not uniform (texture function is sampled at these locations)

Irregular sampling pattern makes it hard to avoid aliasing!

Magnification vs. Minification

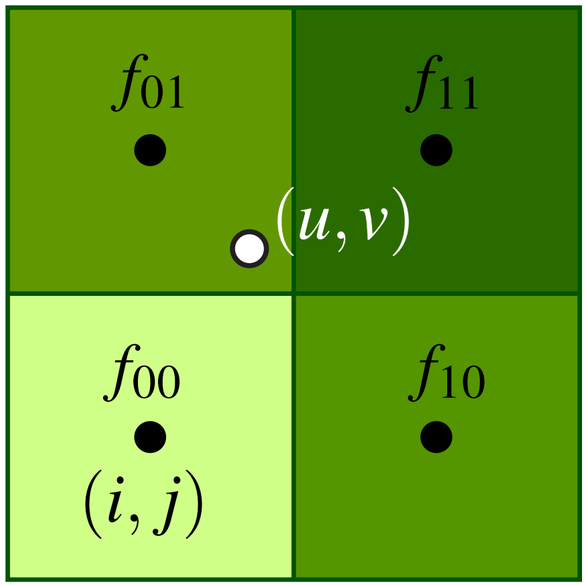


- **Magnification (easier):**
  - Example: camera is very close to scene object
  - Single screen pixel maps to tiny region of texture
  - Can just interpolate value at screen pixel center
- **Minification (harder):**
  - Example: scene object is very far away
  - Single screen pixel maps to large region of texture
  - Need to compute average texture value over pixel to avoid aliasing

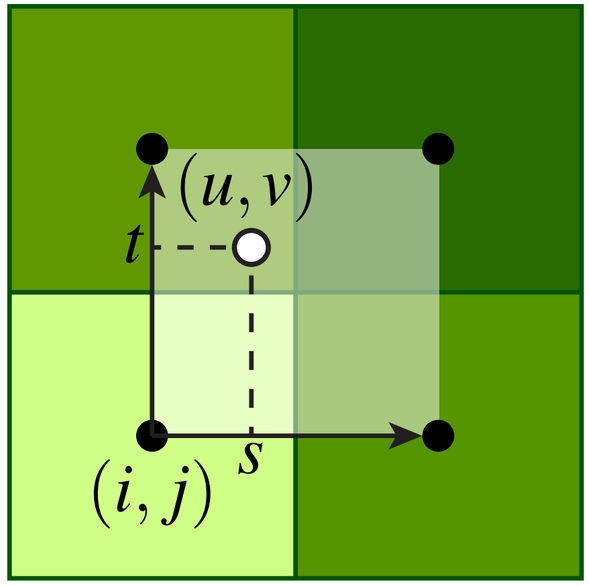
Figure credit: Akeley and Hanrahan

Bilinear interpolation (magnification)

How can we "look up" a texture value at a non-integer location (u, v)?



$$i = \lfloor u - \tfrac{1}{2} \rfloor$$
$$j = \lfloor v - \tfrac{1}{2} \rfloor$$

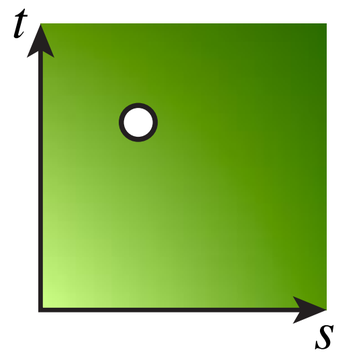


$$s = u - \left(i + \tfrac{1}{2}\right) \in [0,1]$$
$$t = v - \left(j + \tfrac{1}{2}\right) \in [0,1]$$

linear (each row)

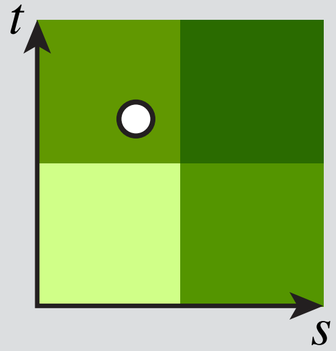
$$(1-s) f_{01} + s f_{11}$$
$$(1-s) f_{00} + s f_{10}$$

bilinear



$$(1-t)\left((1-s) f_{00} + s f_{10}\right) + t\left((1-s) f_{01} + s f_{11}\right)$$

nearest neighbor

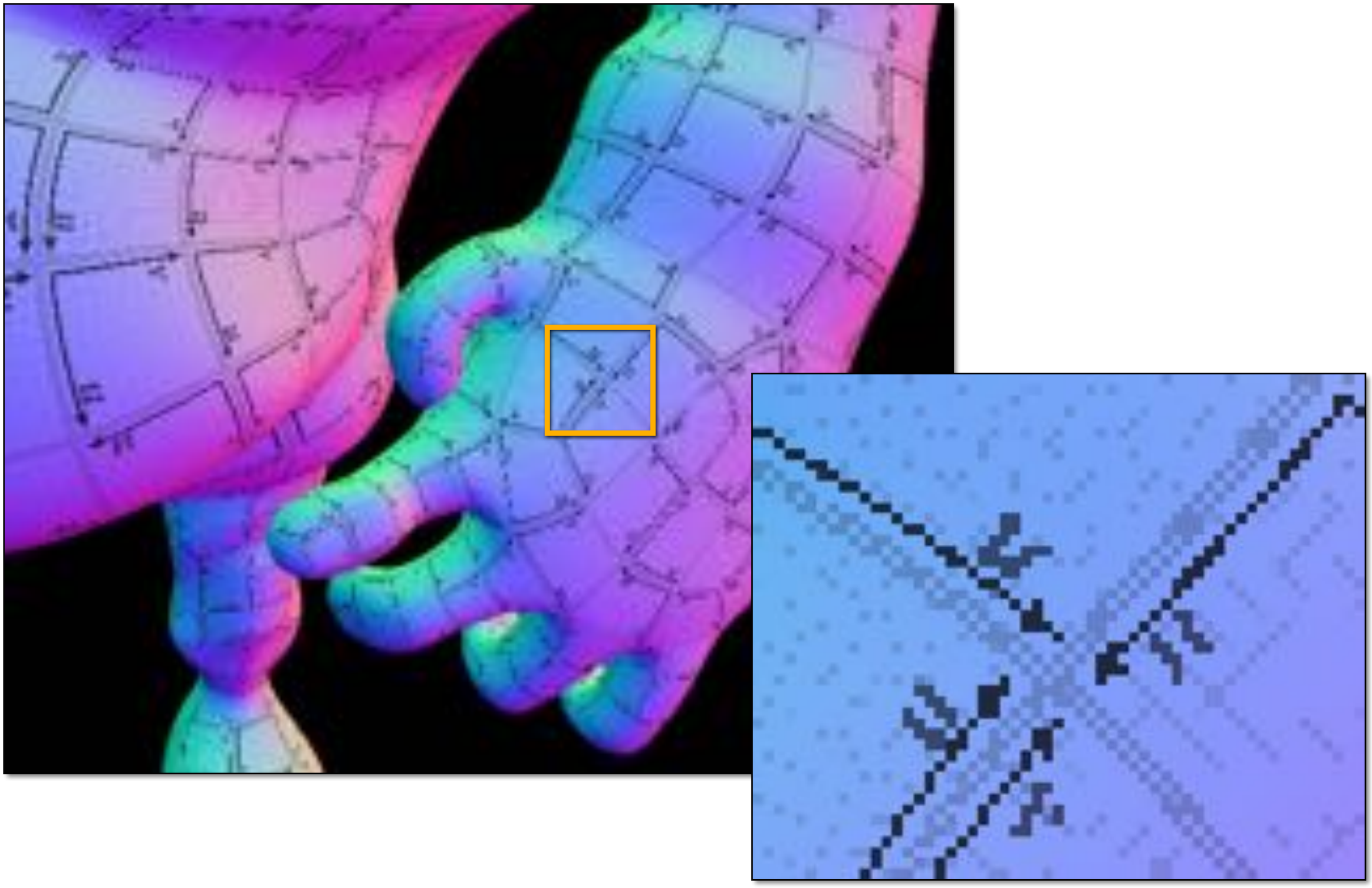


fast but ugly:

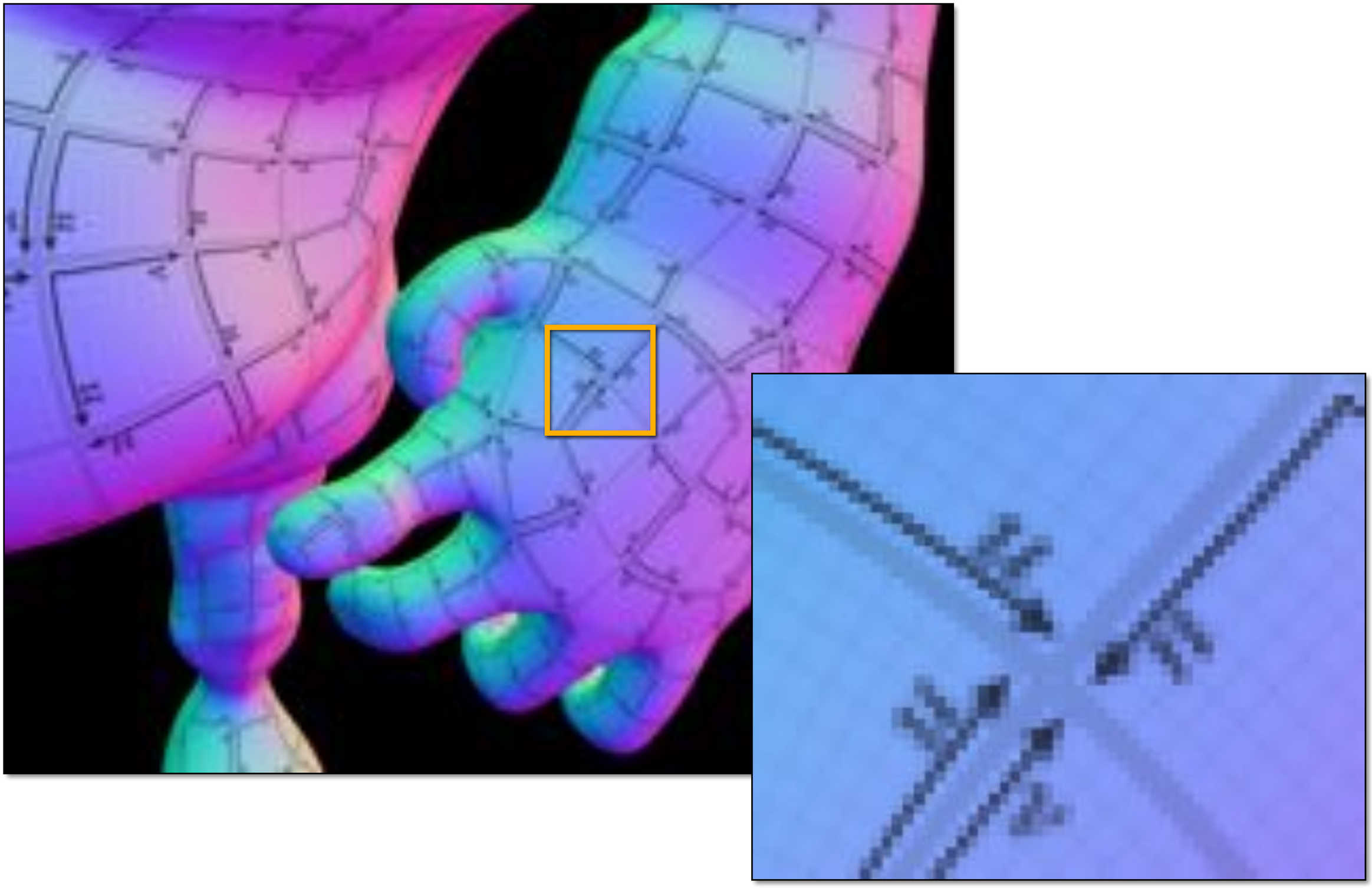
just grab value of nearest "texel" (texture pixel)

Q: What happens if we interpolate vertically first?

Aliasing due to minification



“Pre-filtering” texture (minification)



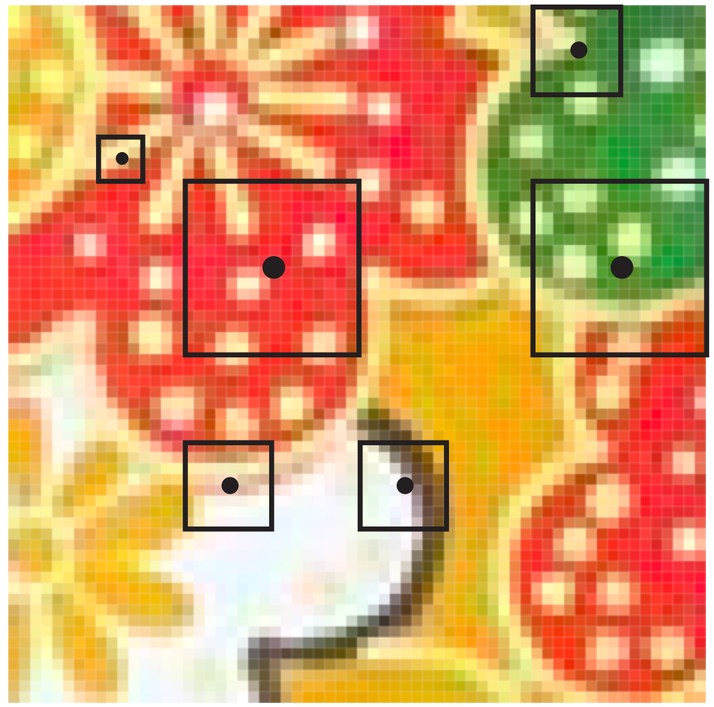
Texture prefiltering — basic idea

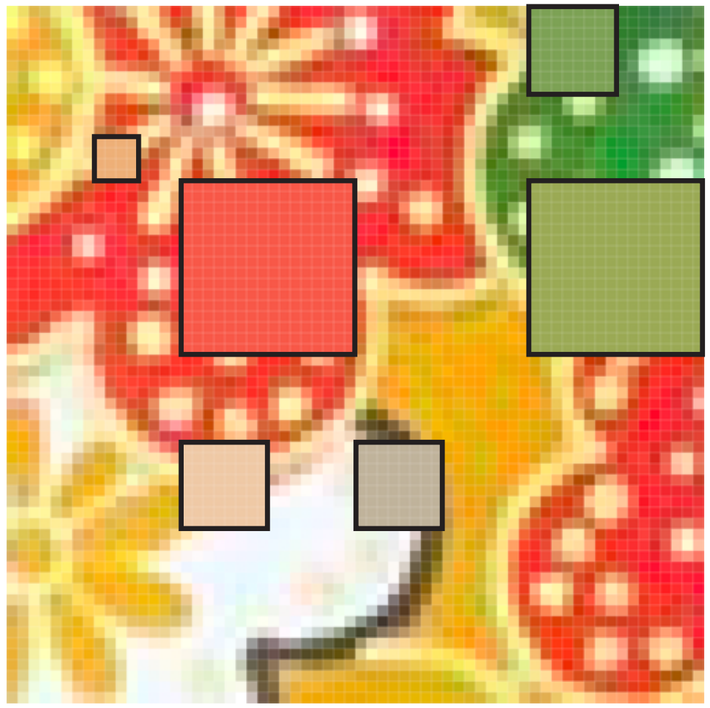
- Texture aliasing often occurs because a single pixel on the screen covers many pixels of the texture

- If we just grab the texture value at the center of the pixel, we get aliasing (get a "random" color that changes if the sample moves even very slightly)

- Ideally, would use the average texture value—but this is expensive to compute

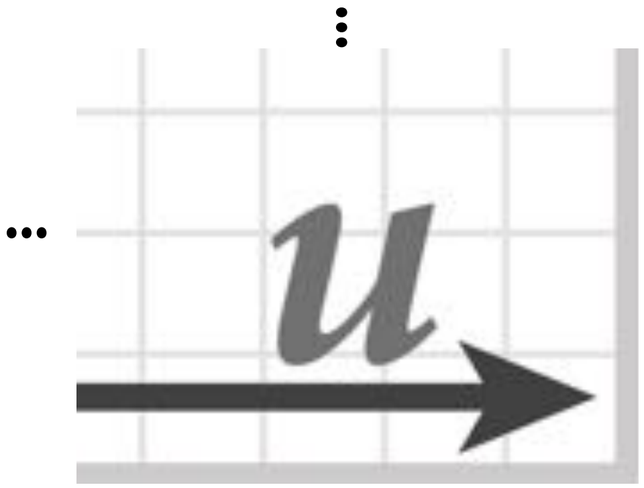
- Instead, we can pre-compute the averages (once) and just look up these averages (many times) at run-time



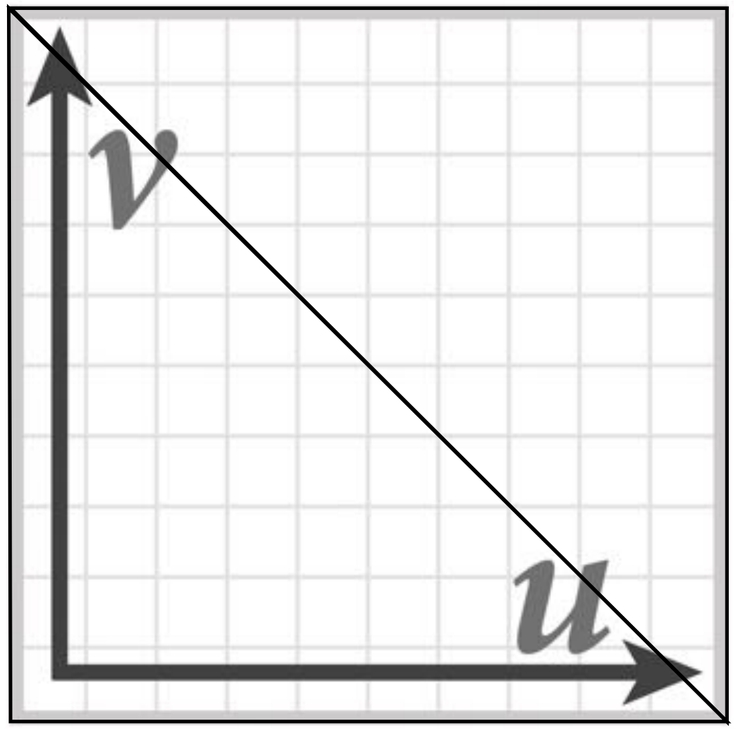


But which averages should we store? Can't precompute them all!

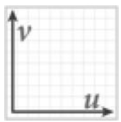
Prefiltered textures



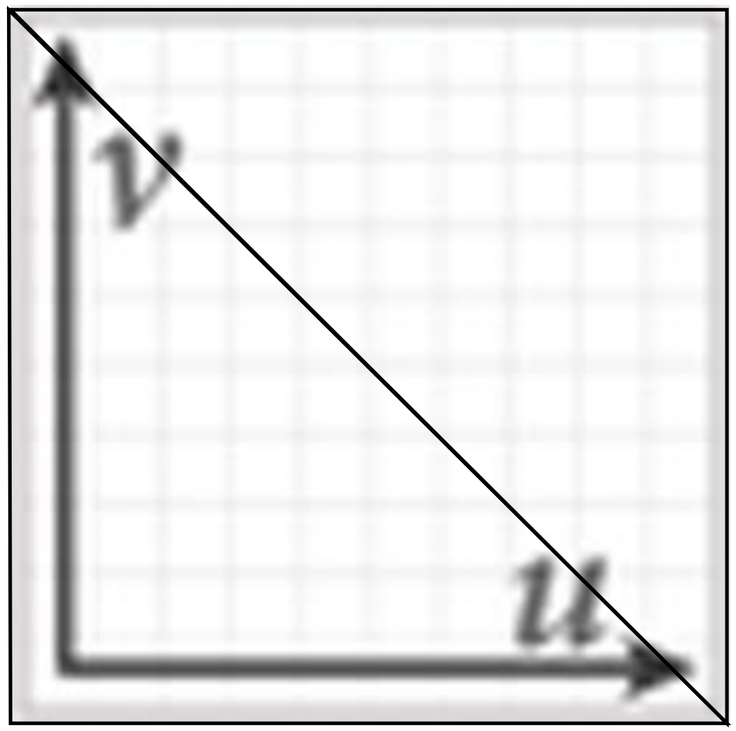
Actual texture: 700x700 image
(only a crop is shown)



Texture minification



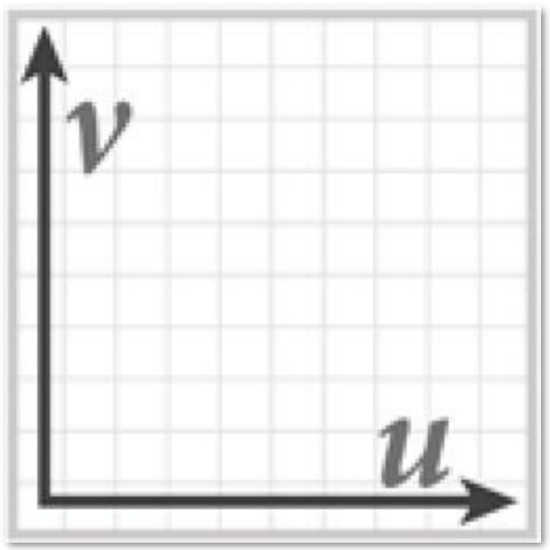
Actual texture: 64x64 image



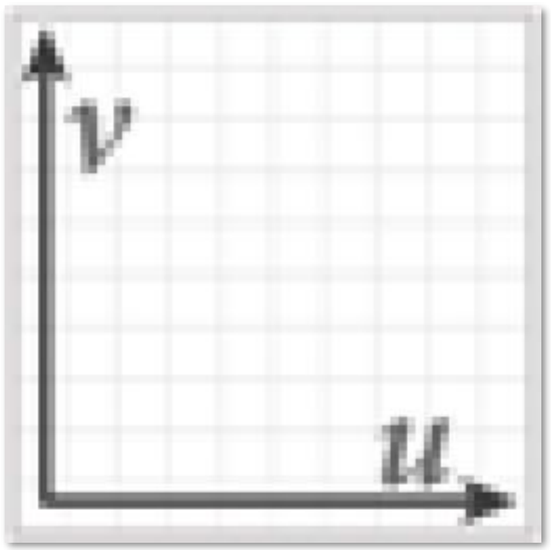
Texture magnification

Q: Are two resolutions enough? A: No…

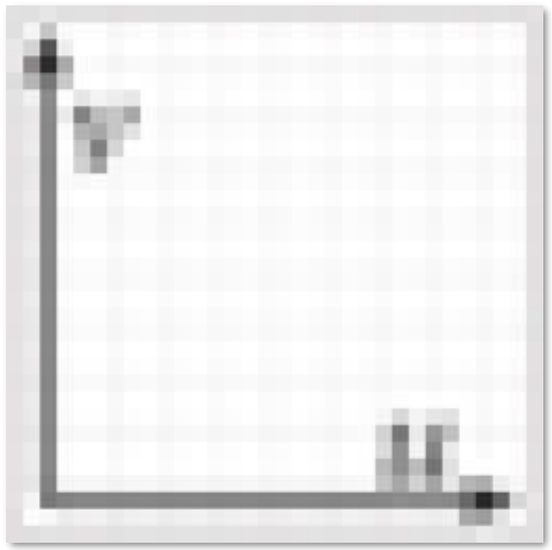
MIP map (L. Williams 83)



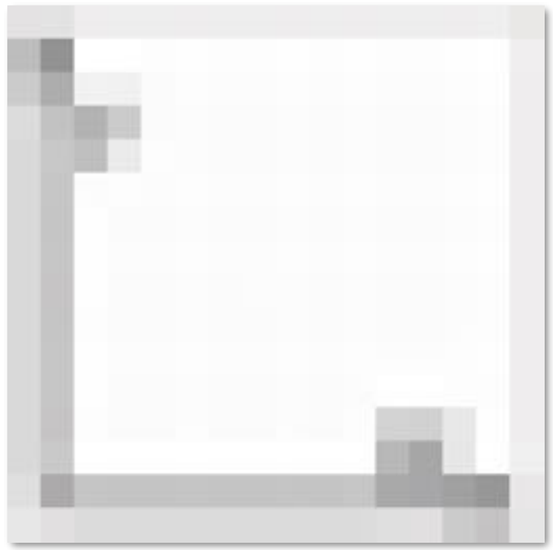
Level 0 = 128x128



Level 1 = 64x64

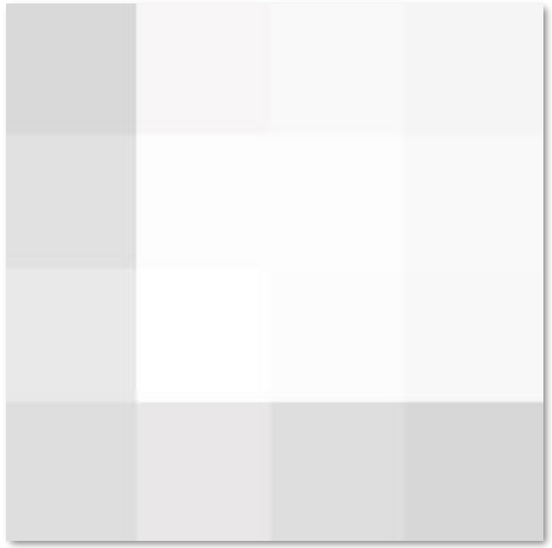


Level 2 = 32x32



Level 3 = 16x16

Level 4 = 8x8



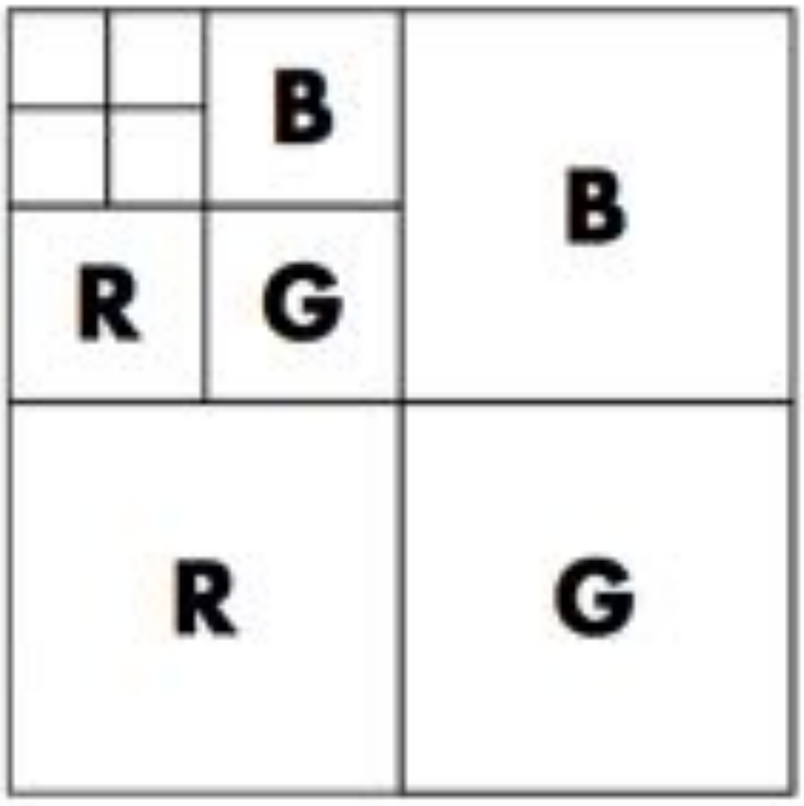
Level 5 = 4x4

Level 6 = 2x2

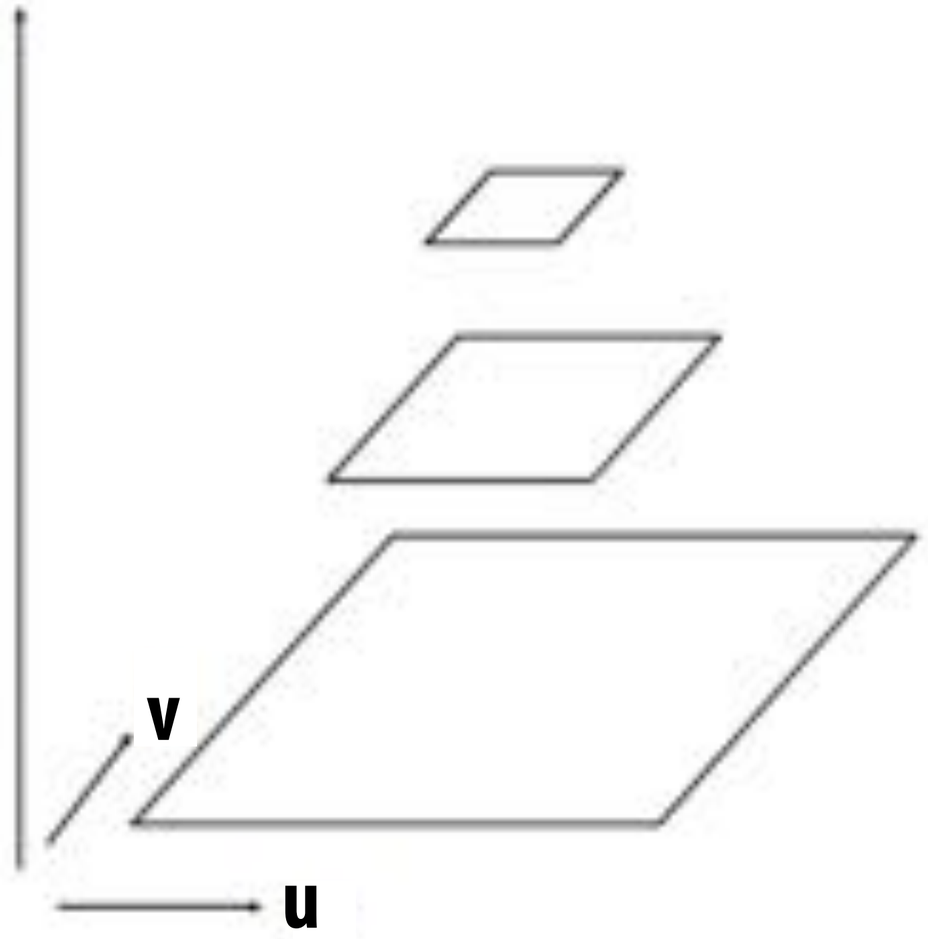
Level 7 = 1x1

- Rough idea: store prefiltered image at "every possible scale"
- Texels at higher levels store average of texture over a region of texture space (downsampled)
- Later: look up a single pixel from MIP map of appropriate size

Mipmap (L. Williams 83)



Williams' original proposed mip-map layout



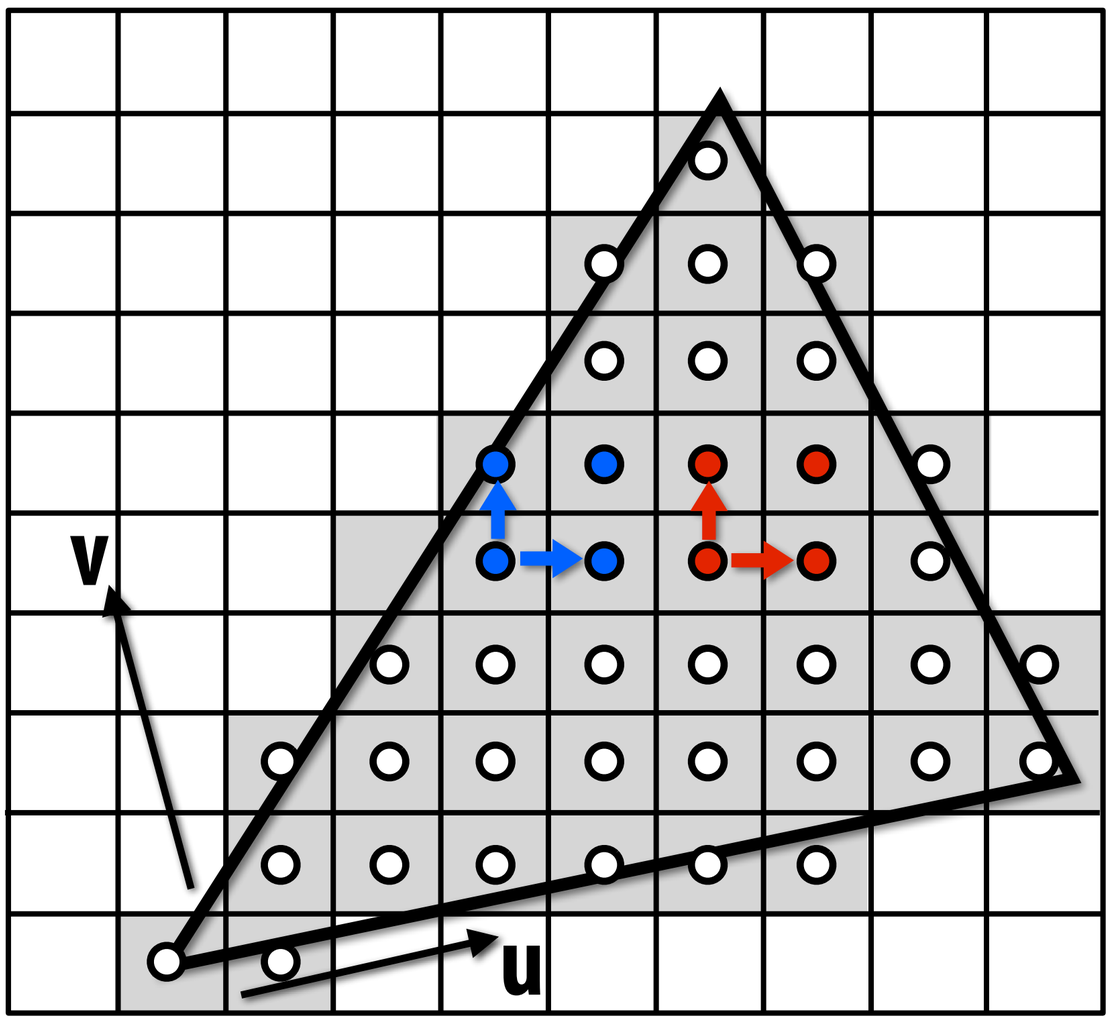
"Mip hierarchy" level = d

Q: What's the storage overhead of a mipmap?

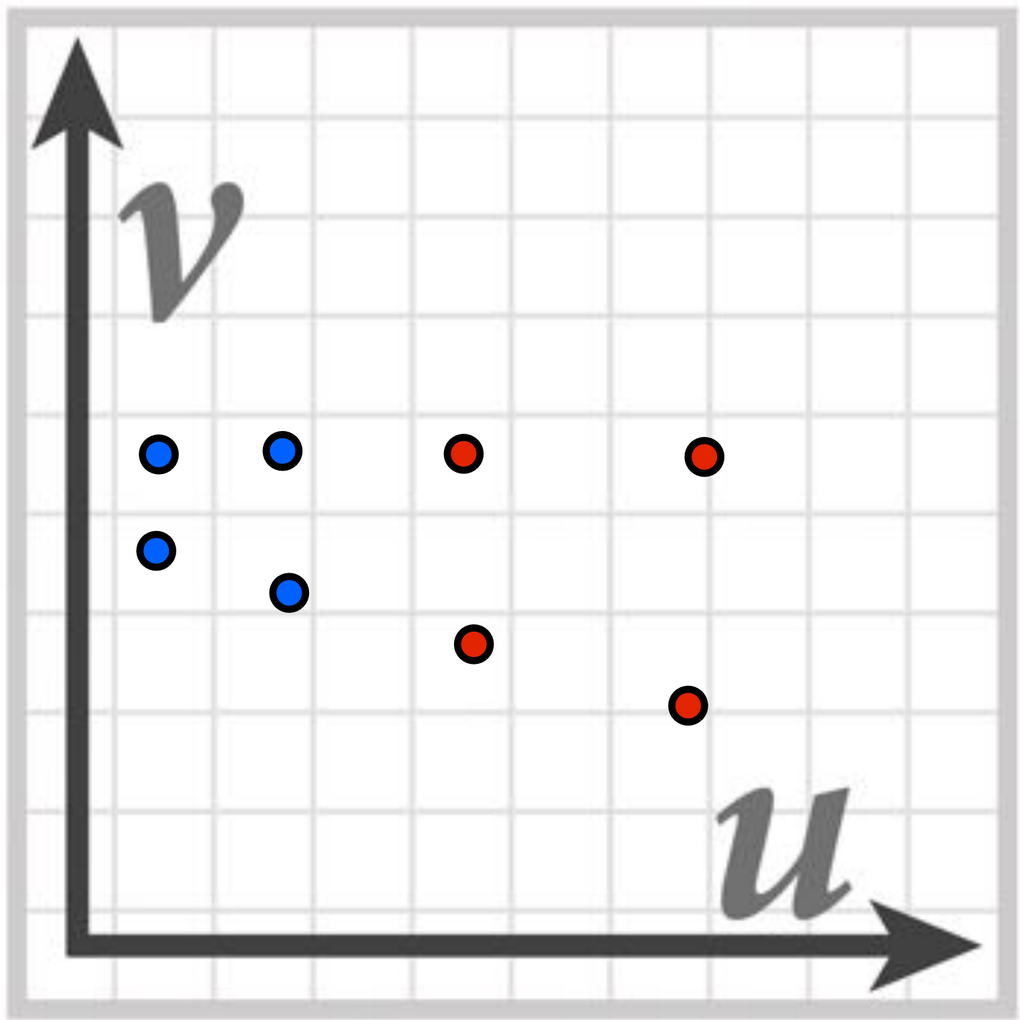
Slide credit: Akeley and Hanrahan

Computing MIP Map Level

Even within a single triangle, may want to sample from different MIP map levels:



Screen space

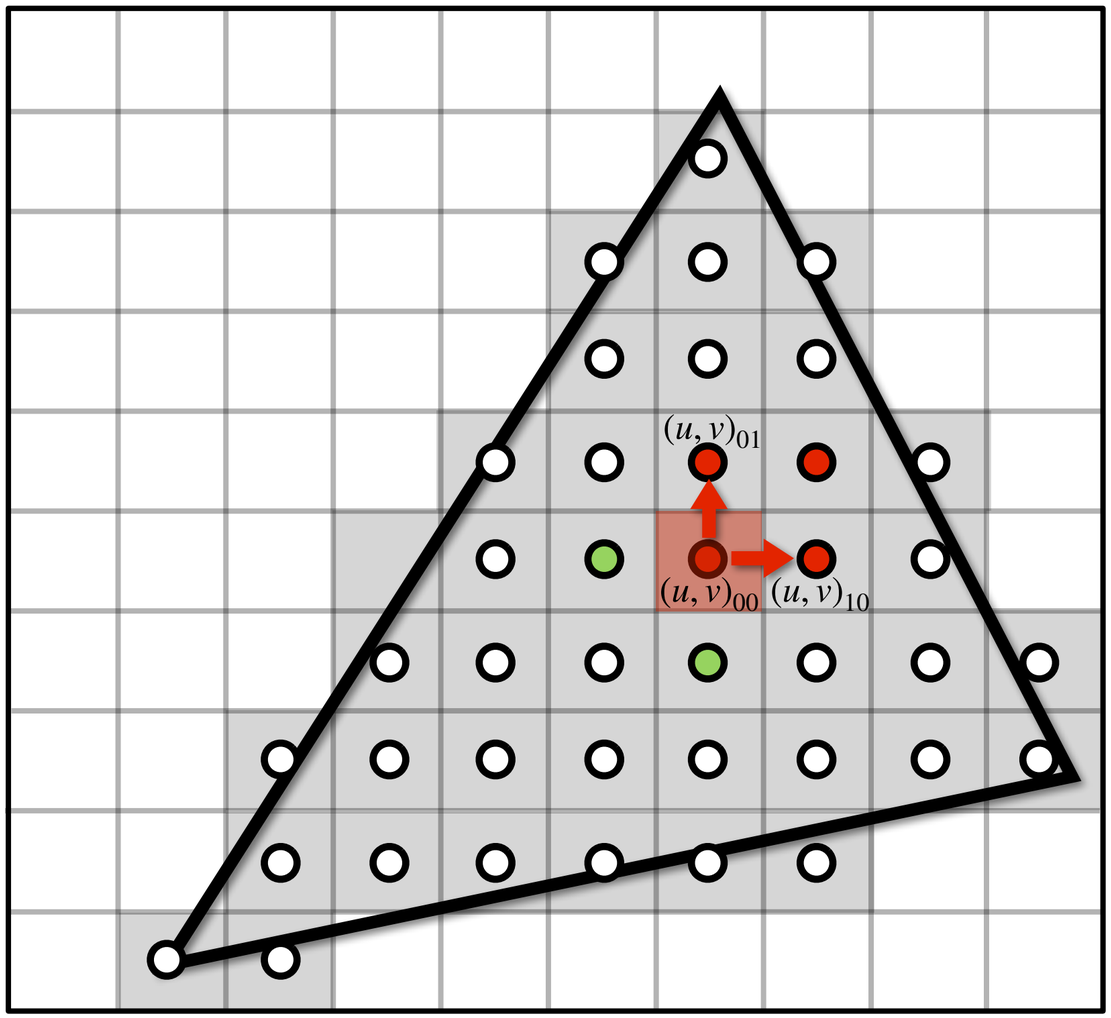


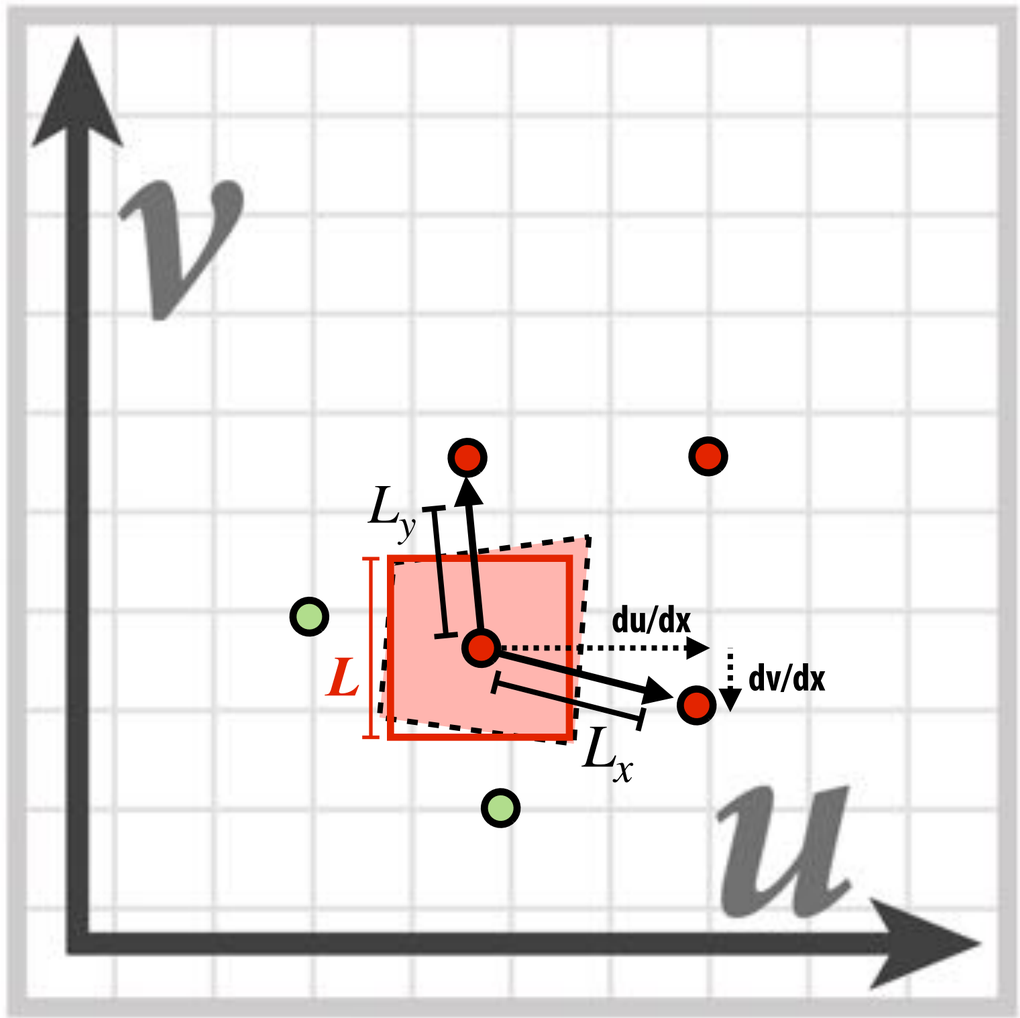
Texture space

Q: Which pixel should sample from a coarser MIP map level: the blue one, or the red one?

Computing Mip Map Level

Compute differences between texture coordinate values at neighboring samples





$$\frac{du}{dx} = u_{10} - u_{00} \qquad \frac{dv}{dx} = v_{10} - v_{00}$$

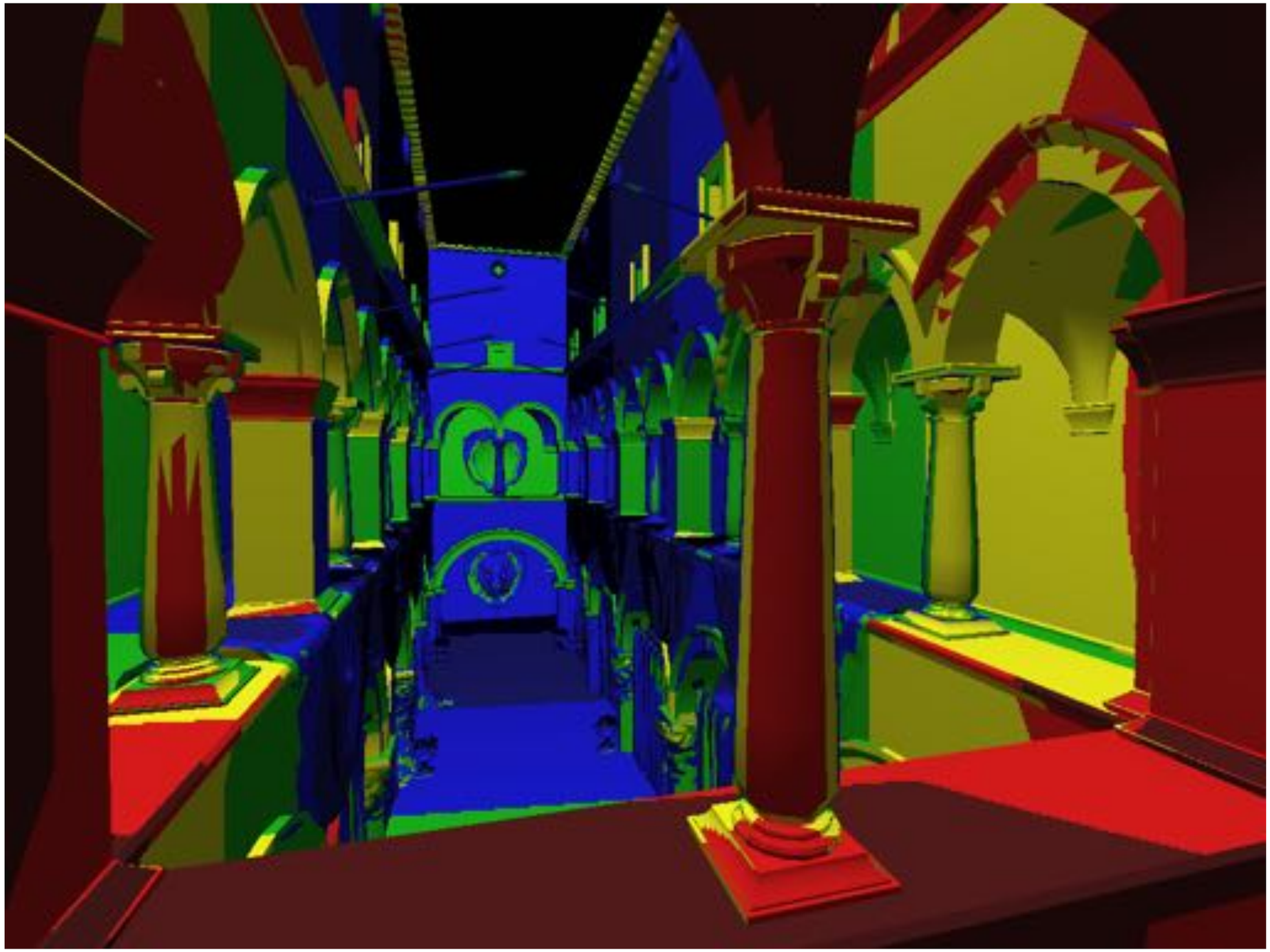
$$\frac{du}{dy} = u_{01} - u_{00} \qquad \frac{dv}{dy} = v_{01} - v_{00}$$

$$L_x^2 = \left(\frac{du}{dx}\right)^2 + \left(\frac{dv}{dx}\right)^2 \qquad L_y^2 = \left(\frac{du}{dy}\right)^2 + \left(\frac{dv}{dy}\right)^2$$

$$L = \sqrt{\max(L_x^2, L_y^2)}$$

mip-map level: $d = \log_2 L$

Visualization of mip-map level
(d clamped to nearest level)



Sponza (bilinear resampling at level 0)

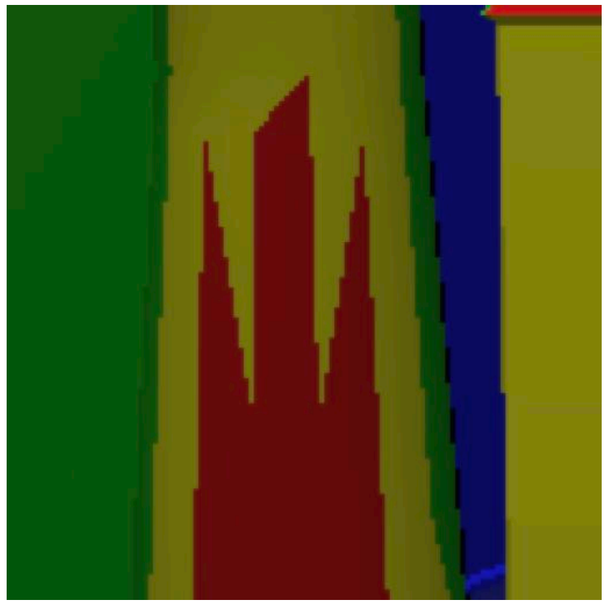
Sponza (bilinear resampling at level 2)

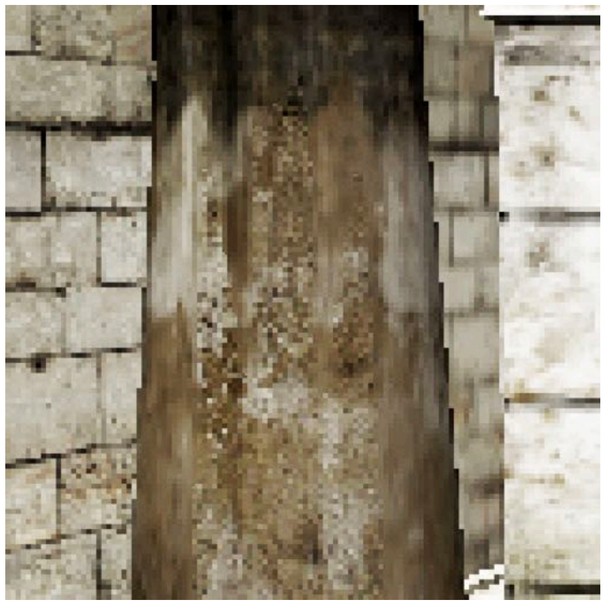
Sponza (bilinear resampling at level 4)

Sponza (MIP mapped)

Problem with basic MIP mapping

- If we just use the nearest level, can get artifacts where level "jumps"—appearance sharply transitions from detailed to blurry texture
- **IDEA**: rather than clamping the MIP map level to the closest integer, use the original (continuous) MIP map level d
- **PROBLEM**: we only computed a fixed number of MIP map levels. How do we interpolate between levels?

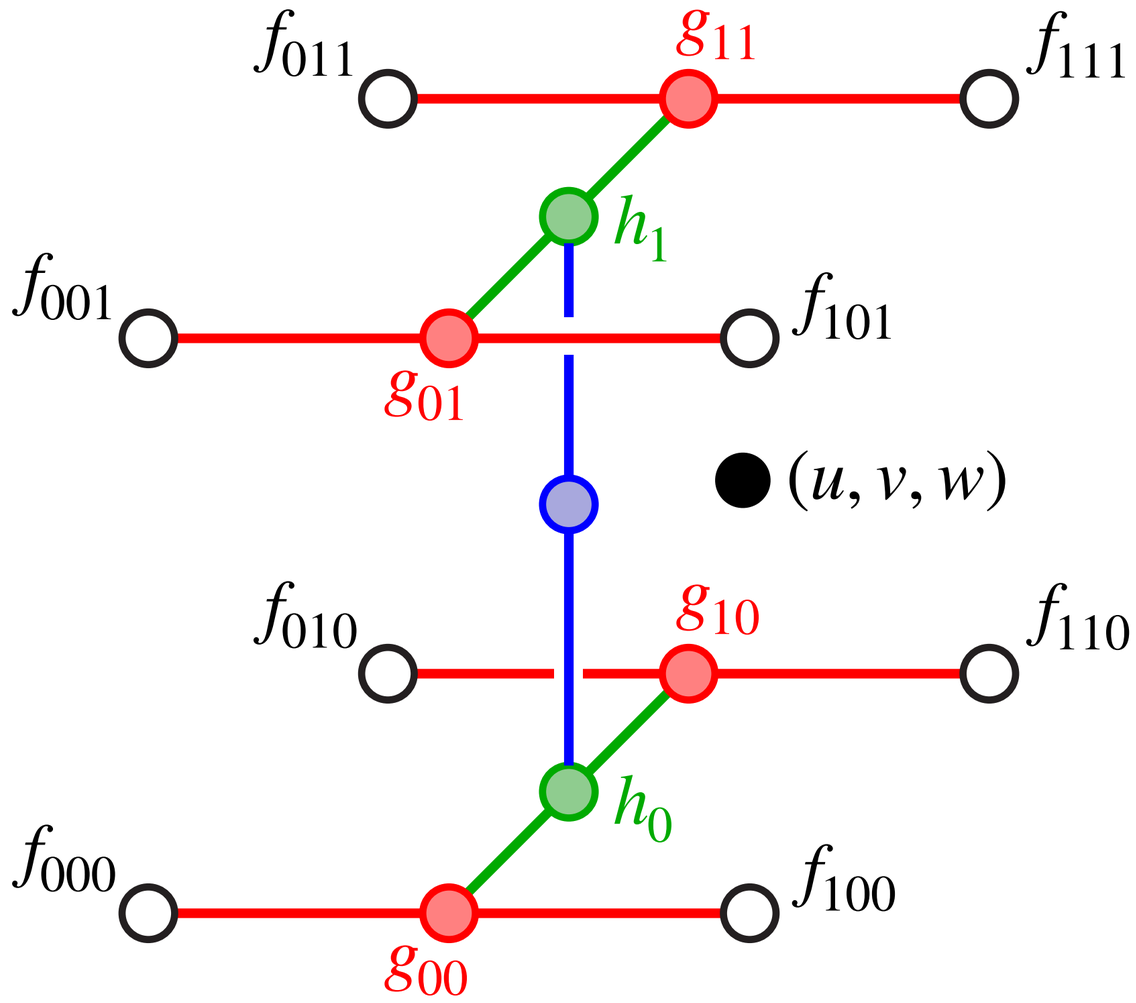




Trilinear Filtering

image adapted from: Akeley and Hanrahan

- Used <u>bilinear</u> filtering for 2D data; can use <u>trilinear</u> filtering for 3D data
- Given a point $(u, v, w) \in [0,1]^3$, and eight closest values f_{ijk}
- Just iterate linear filtering:
  - weighted average along u
  - weighted average along v
  - weighted average along w



$$g_{00} = (1-u)f_{000} + uf_{100}$$
$$g_{10} = (1-u)f_{010} + uf_{110}$$
$$g_{01} = (1-u)f_{001} + uf_{101}$$
$$g_{11} = (1-u)f_{011} + uf_{111}$$

$$h_0 = (1-v)g_{00} + vg_{10}$$
$$h_1 = (1-v)g_{01} + vg_{11}$$

$$(1-w)h_0 + wh_1$$

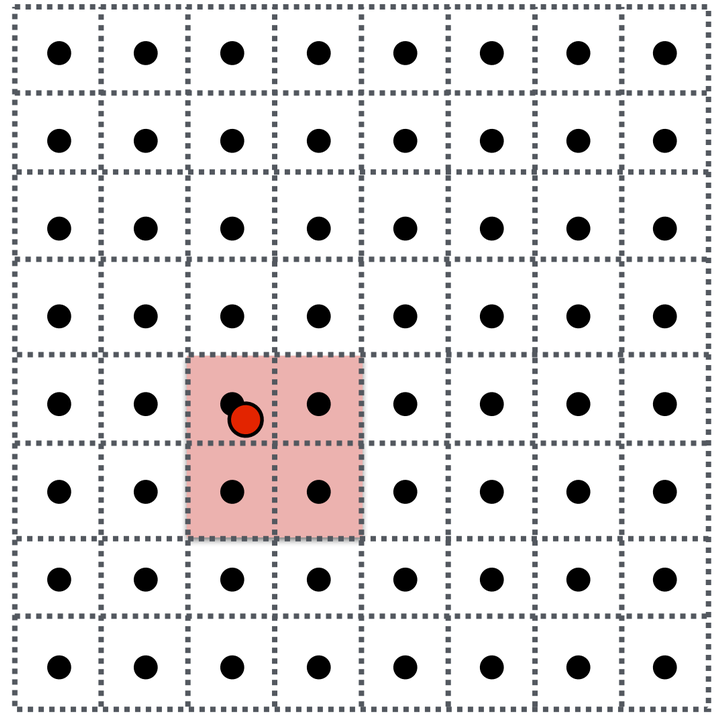
MIP Map Lookup

- MIP map interpolation works essentially the same way
  - not interpolating from 3D grid
  - interpolate from two MIP map levels closest to $d \in \mathbb{R}$
  - perform bilinear interpolation independently in each level
  - interpolate between two bilinear values using $w = d - \lfloor d \rfloor$

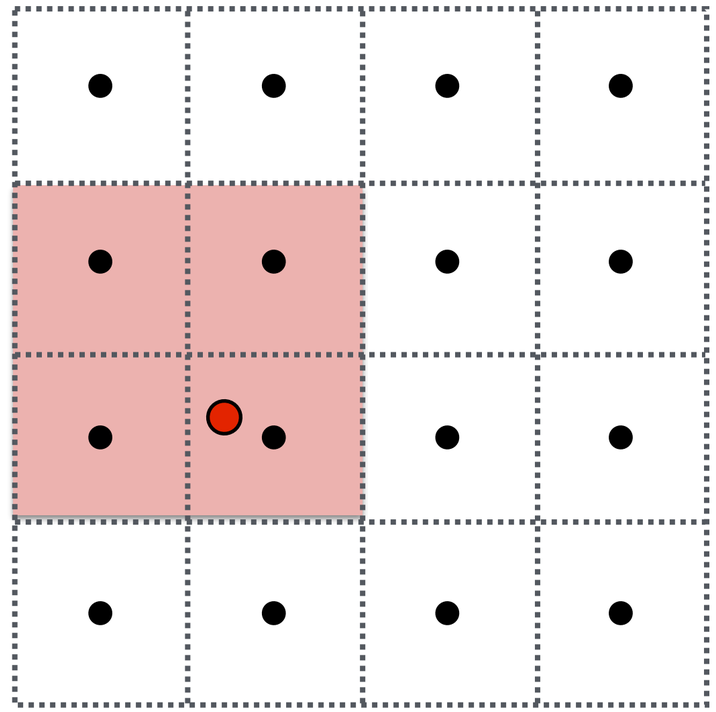
Starts getting expensive! (➟ specialized hardware)

Bilinear interpolation:
four texel reads
3 linear interpolations (3 mul + 6 add)

Trilinear/MIP map interpolation:
eight texel reads
7 linear interpolations (7 mul + 14 add)



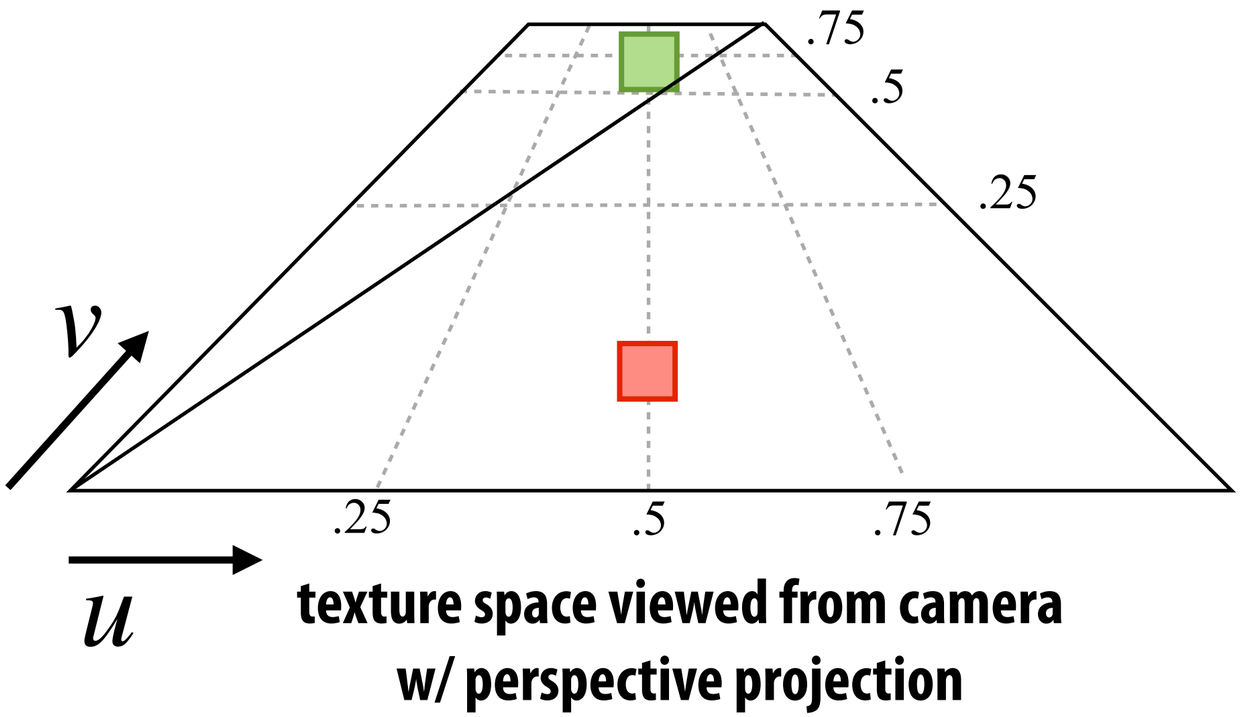
mip-map texels: level $\lfloor d \rfloor$



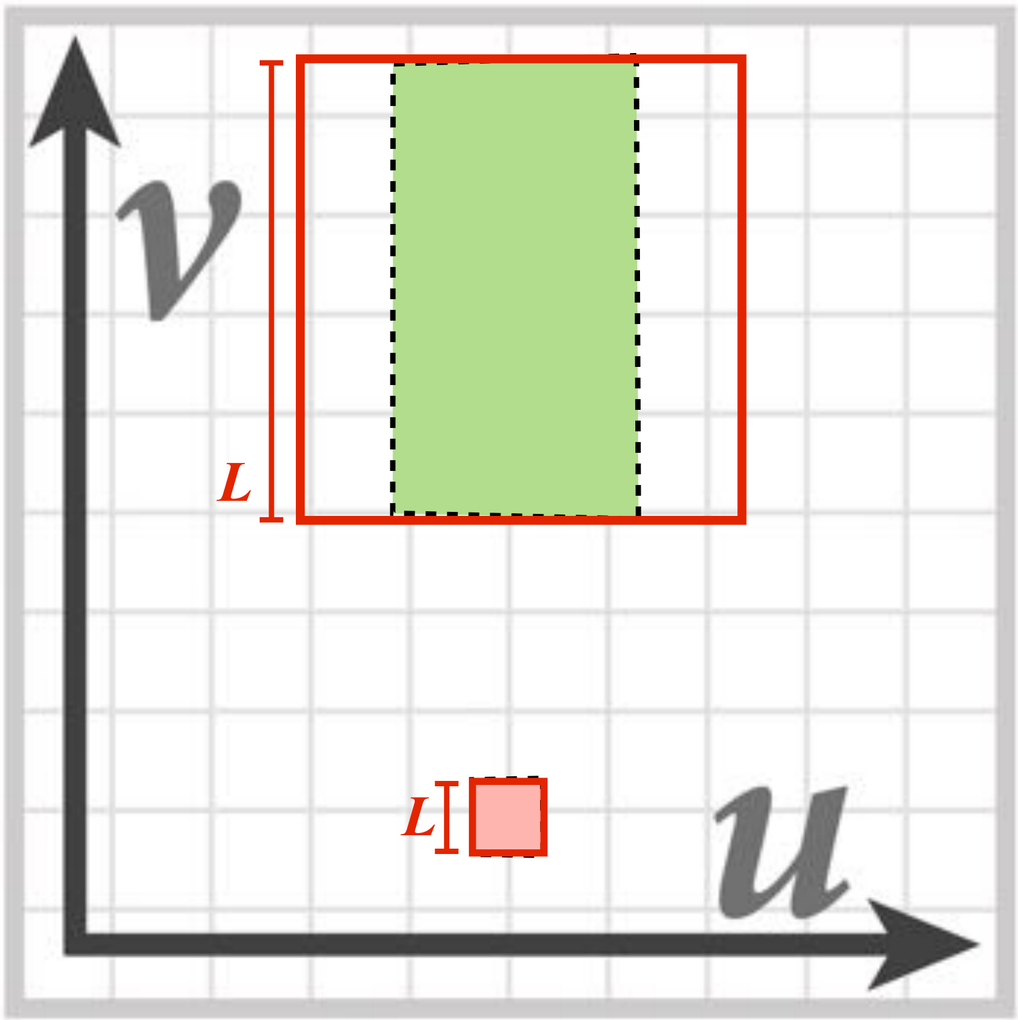
mip-map texels: level $\lfloor d \rfloor + 1$

Anisotropic Filtering

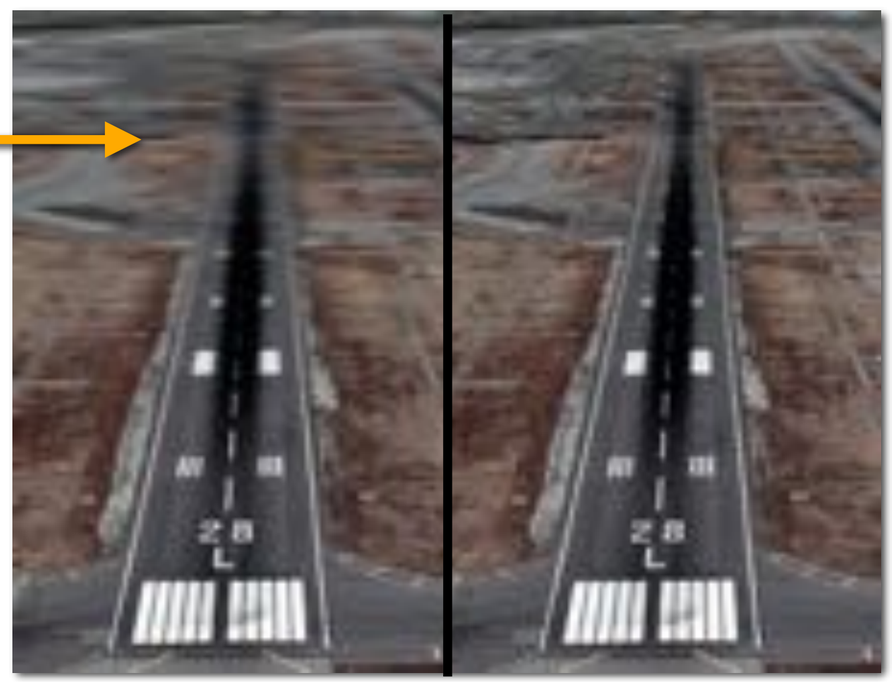
At grazing angles, samples may be stretched out by (very) different amounts along u and v



texture space viewed from camera
w/ perspective projection



Overblurring in u direction



isotropic Filtering (trilinear)

anisotropic Filtering

Common solution: combine multiple MIP map samples
(even more arithmetic/bandwidth!)

Texture Sampling Pipeline

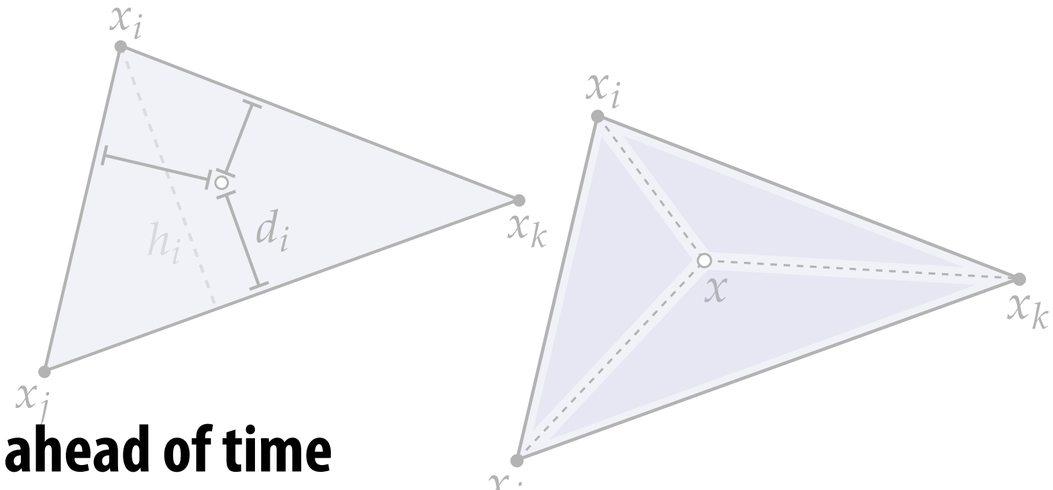
1. **Compute u and v from screen sample (x, y) via barycentric interpolation**
2. **Approximate $\frac{du}{dx}, \frac{du}{dy}, \frac{dv}{dx}, \frac{dv}{dy}$ by taking differences of screen-adjacent samples**
3. **Compute mip map level d**
4. **Convert normalized $[0,1]$ texture coordinate (u, v) to pixel locations $(U, V) \in [W, H]$ in texture image**
5. **Determine addresses of texels needed for filter (e.g., eight neighbors for trilinear)**
6. **Load texels into local registers**
7. **Perform tri-linear interpolation according to (U, V, d)**
8. **(…even more work for anisotropic filtering…)**

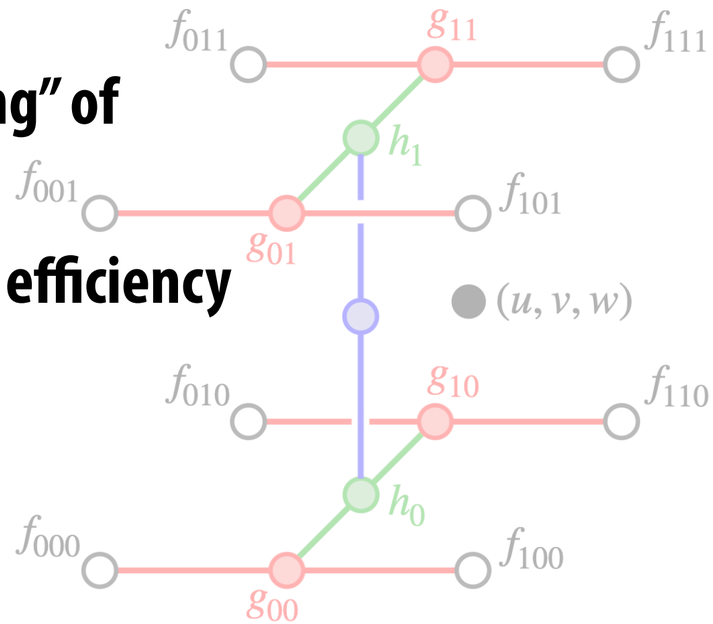
Takeaway: high-quality texturing requires far more work than just looking up a pixel in an image! Each sample demands significant arithmetic & bandwidth

For this reason, graphics processing units (GPUs) have dedicated, fixed-function hardware support to perform texture sampling operations

Texture Mapping—Summary

- Once we have 2D primitives, can interpolate attributes across vertices using barycentric coordinates
- Important example: texture coordinates, used to copy pieces of a 2D image onto a 3D surface
- Careful texture filtering is needed to avoid aliasing
  - Key idea: what's the average color covered by a pixel?
  - For magnification, can just do a bilinear lookup
  - For minification, use prefiltering to compute averages ahead of time
    - a MIP map stores averages at different levels
    - blend between levels using trilinear filtering
- At grazing angles, anisotropic filtering needed to deal w/ "stretching" of samples
- In general, no perfect solution to aliasing! Try to balance quality & efficiency





Next Time: 3D Rotations