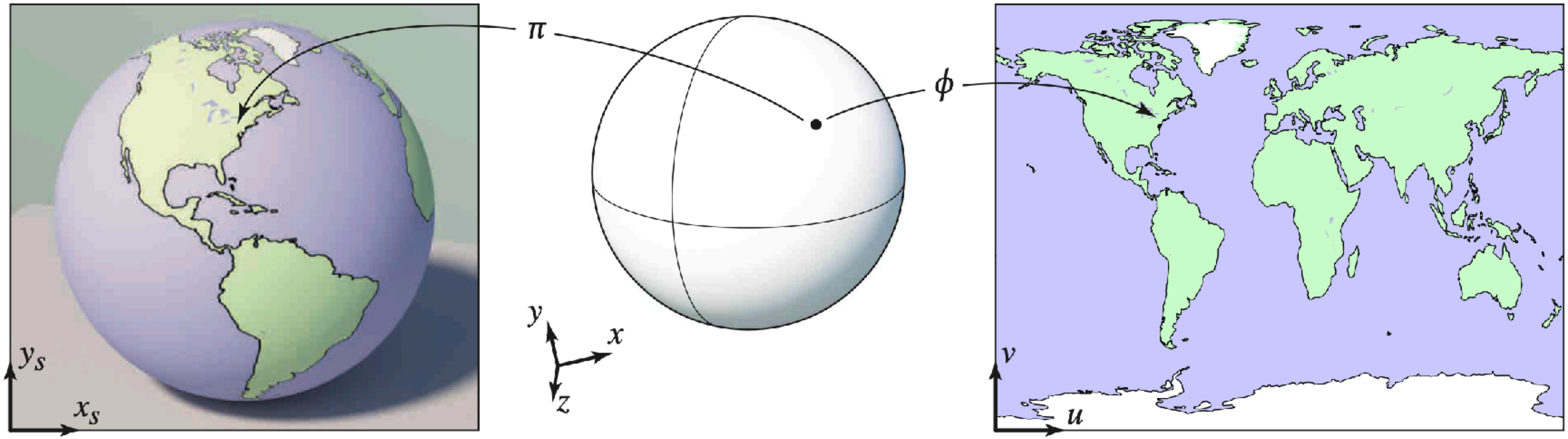
COL781: Computer Graphics

9. Texture Filtering

Olano et al. 2001

Texture mapping



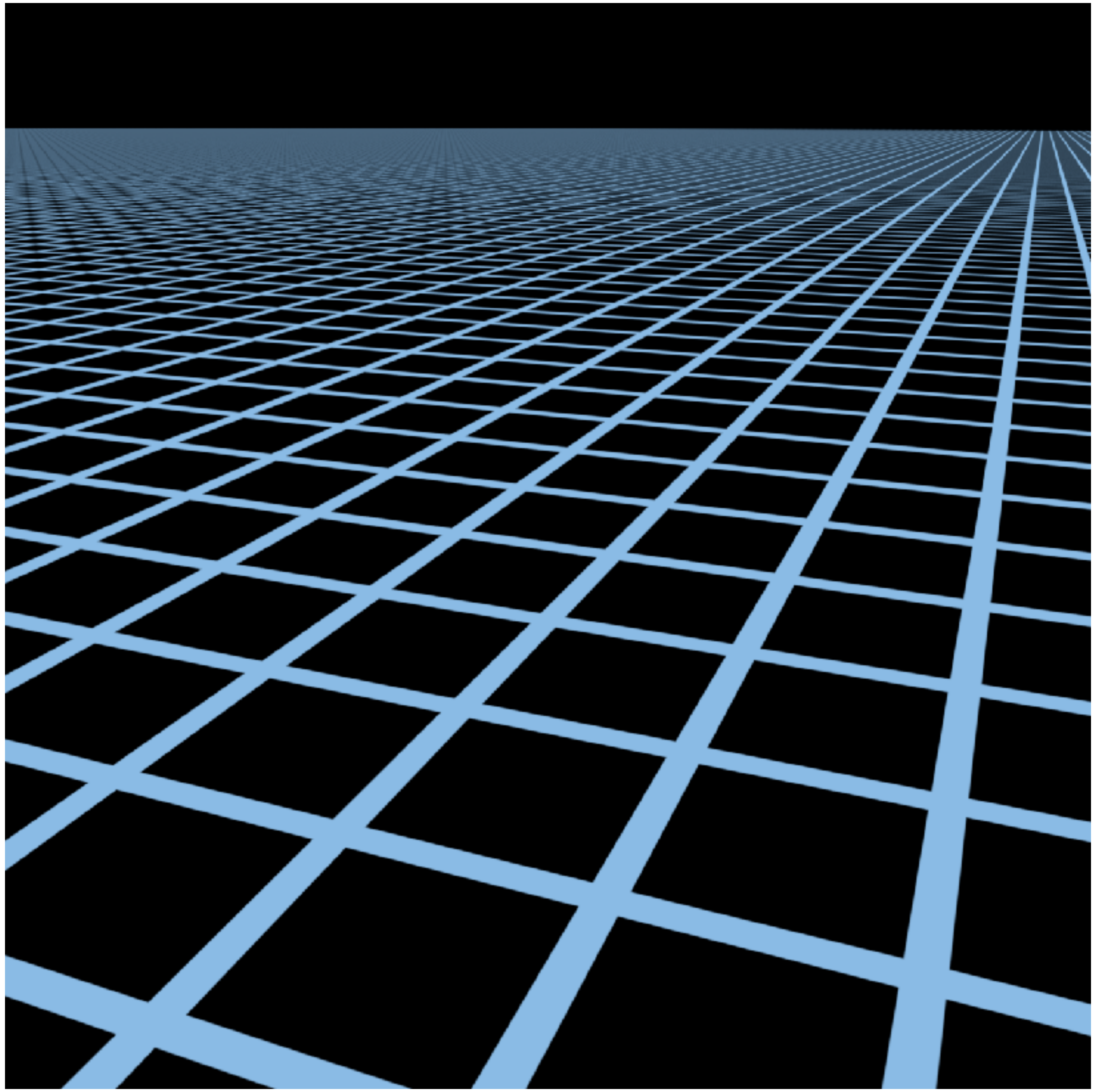
Screen space **World space** **Texture space**

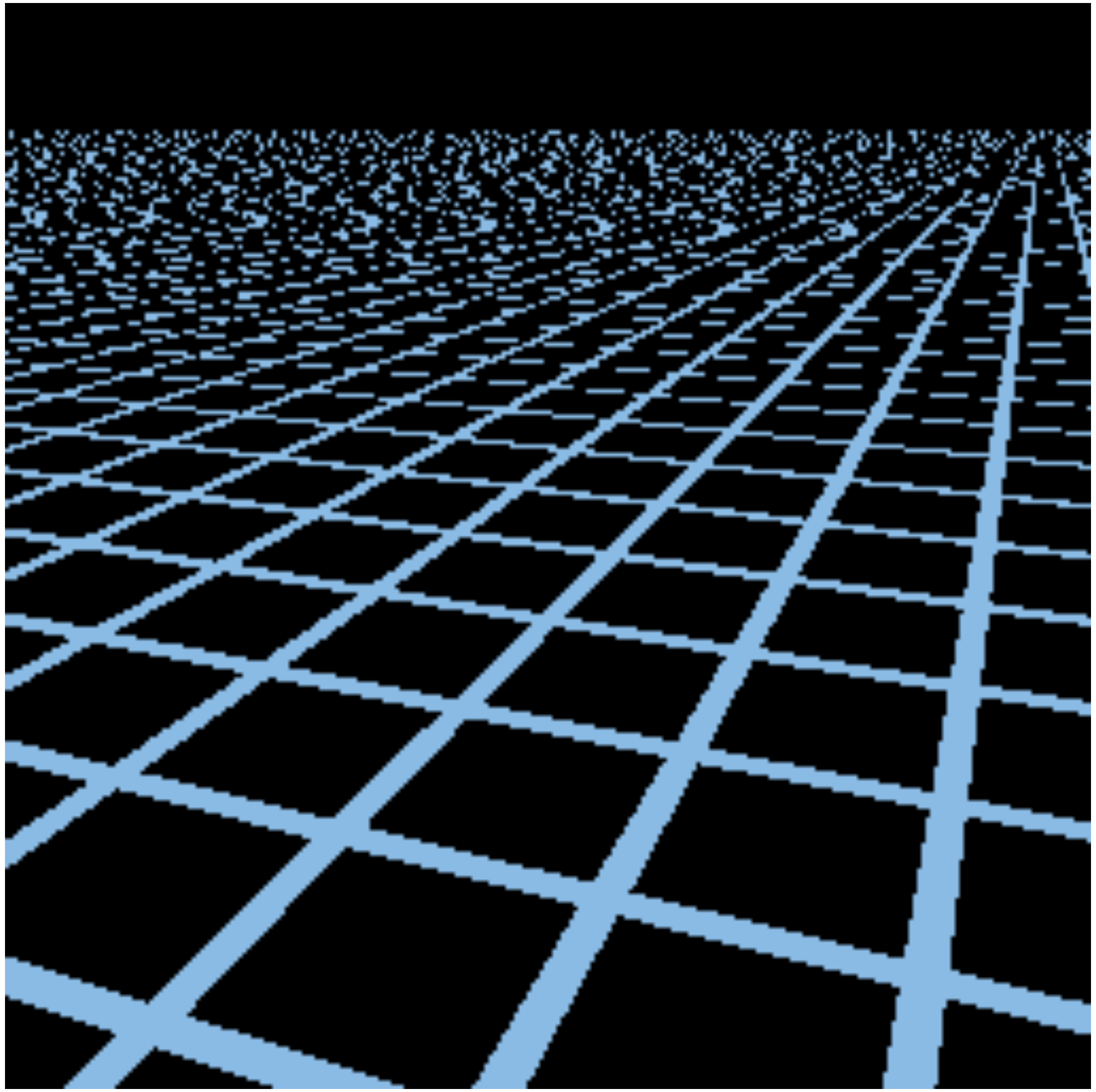
Drawing textured triangles

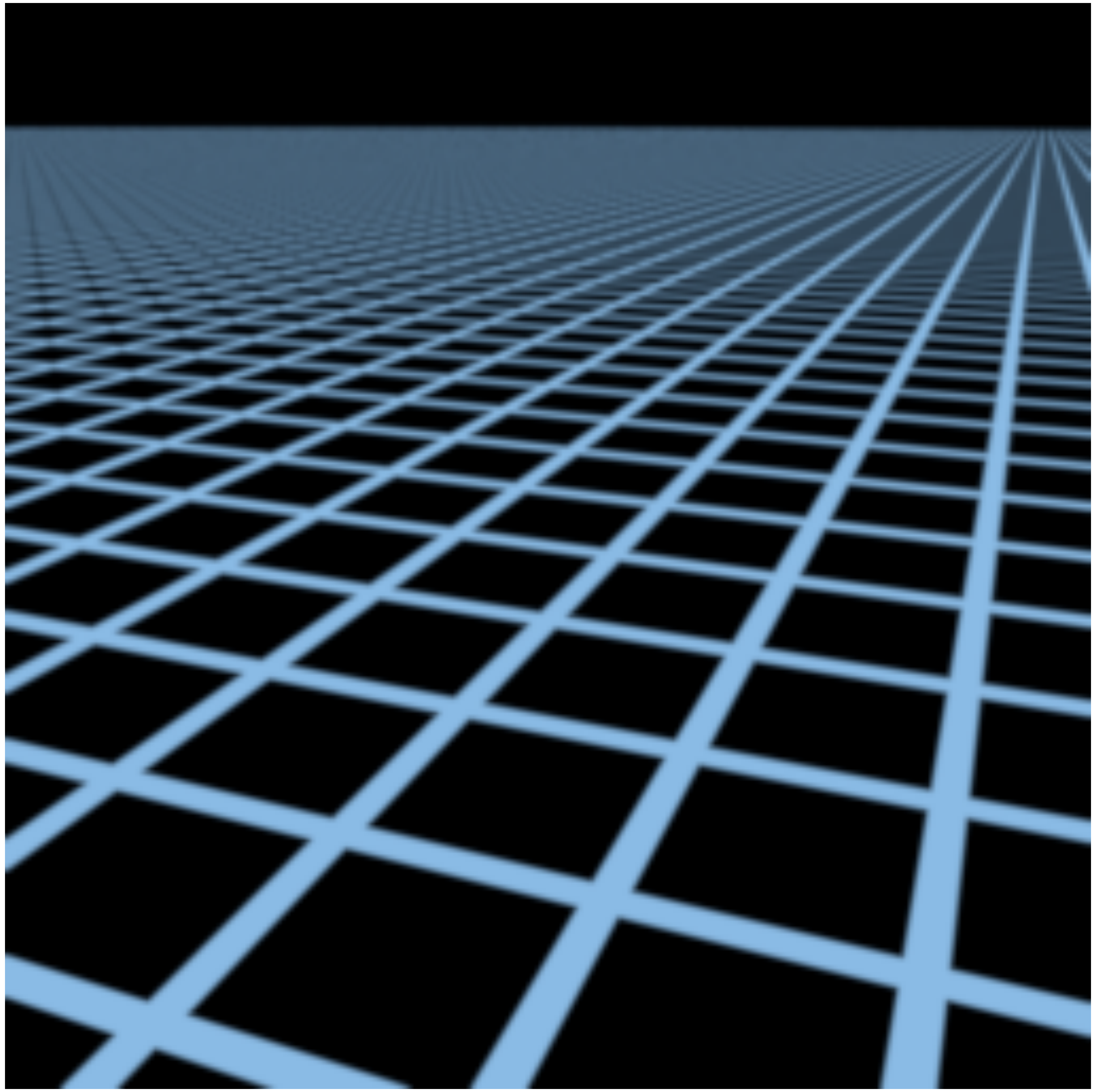
Inputs: (i) mesh with vertex positions (*x*,*y*,*z*) and texture coordinates (*u*,*v*), (ii) texture image

Naïve algorithm:

```
for each triangle (i,j,k):
    for each rasterized sample:
        (u,v) = interpolate (ui,vi), (uj,vj), (uk,vk)
        texColor = sample texture at (u,v)
        sample.color = texcolor
```


High-res reference (1280×1280)


Point sampling (256×256)



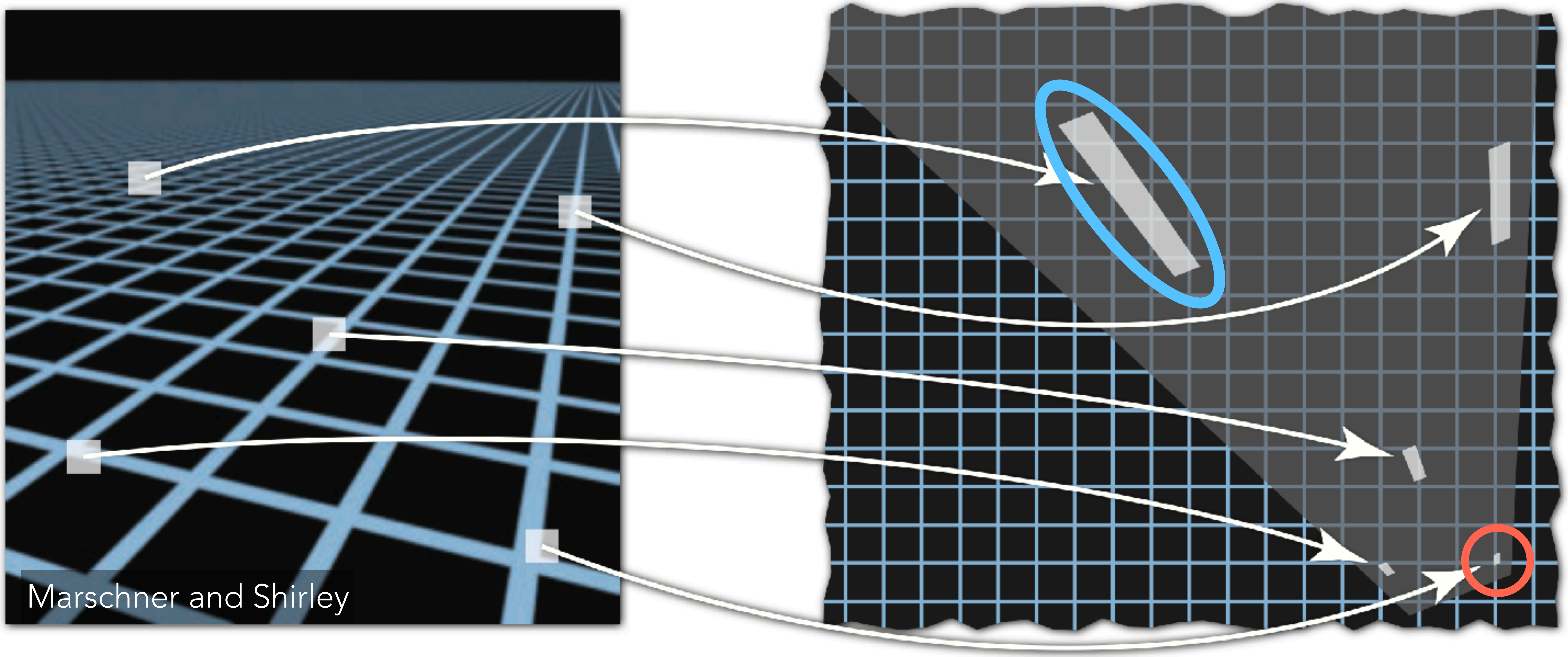
Supersampling (256×256, 512 spp)

"Easy, just do supersampling"

Yes, but:

- Higher frequencies, finer detail ⇒ need more samples to avoid aliasing
- Perspective projection creates arbitrarily high frequencies!
- Texture sampling can be expensive (memory latency)

Can we antialias textures more efficiently?



Texture mapping creates a very irregular sampling pattern!

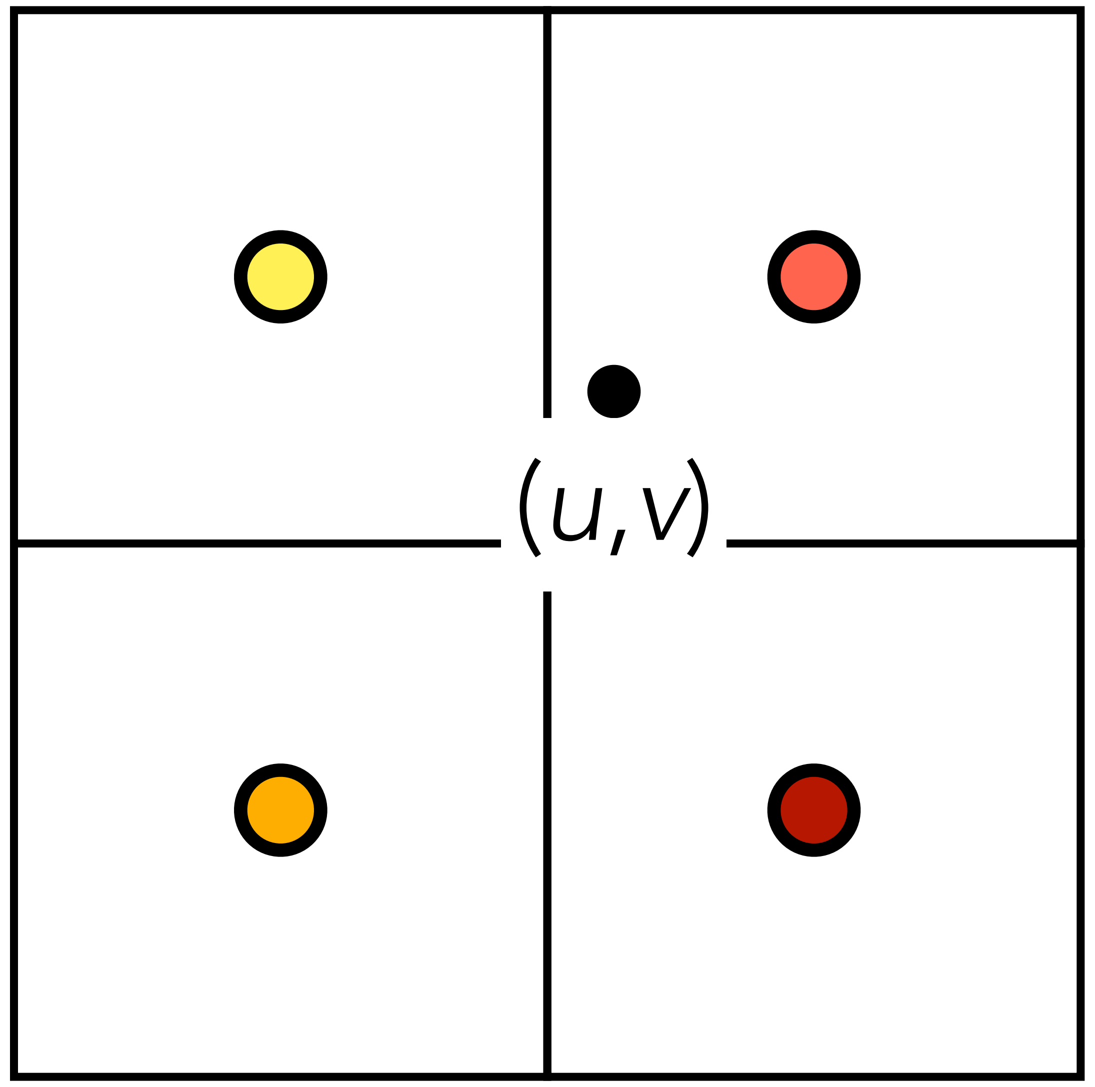
- Some regions are **magnified**: multiple screen samples per texture pixel (texel)
- Some regions are "**minified**": multiple texels per sample

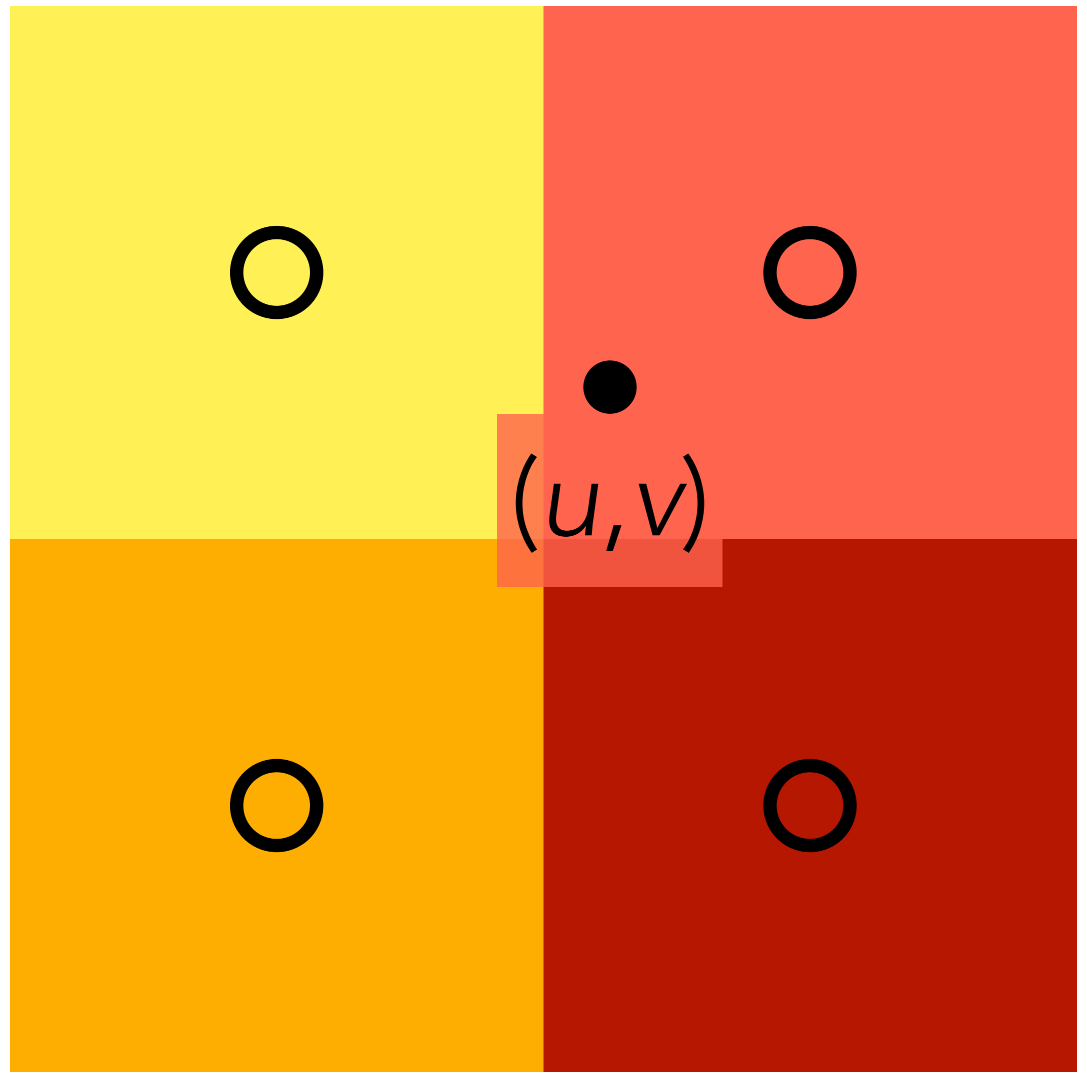
Magnification

Easy case, no aliasing. Just need to "look up" texture value at non-integer location (u,v)

Signal reconstruction ≈ interpolation

Simple and crude: nearest neighbour

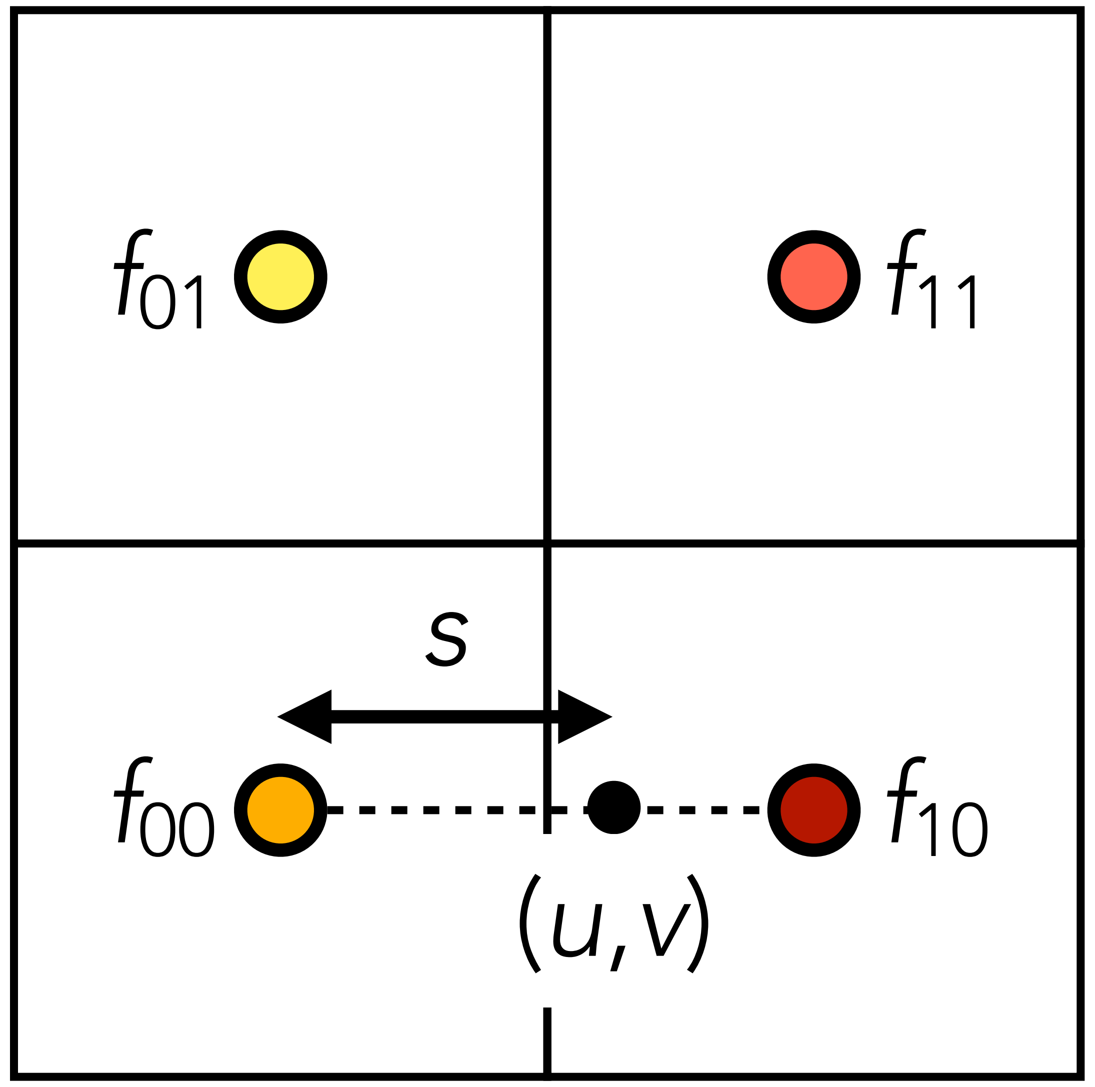




Bilinear interpolation

If sample point lay exactly on a row, we could do linear interpolation:

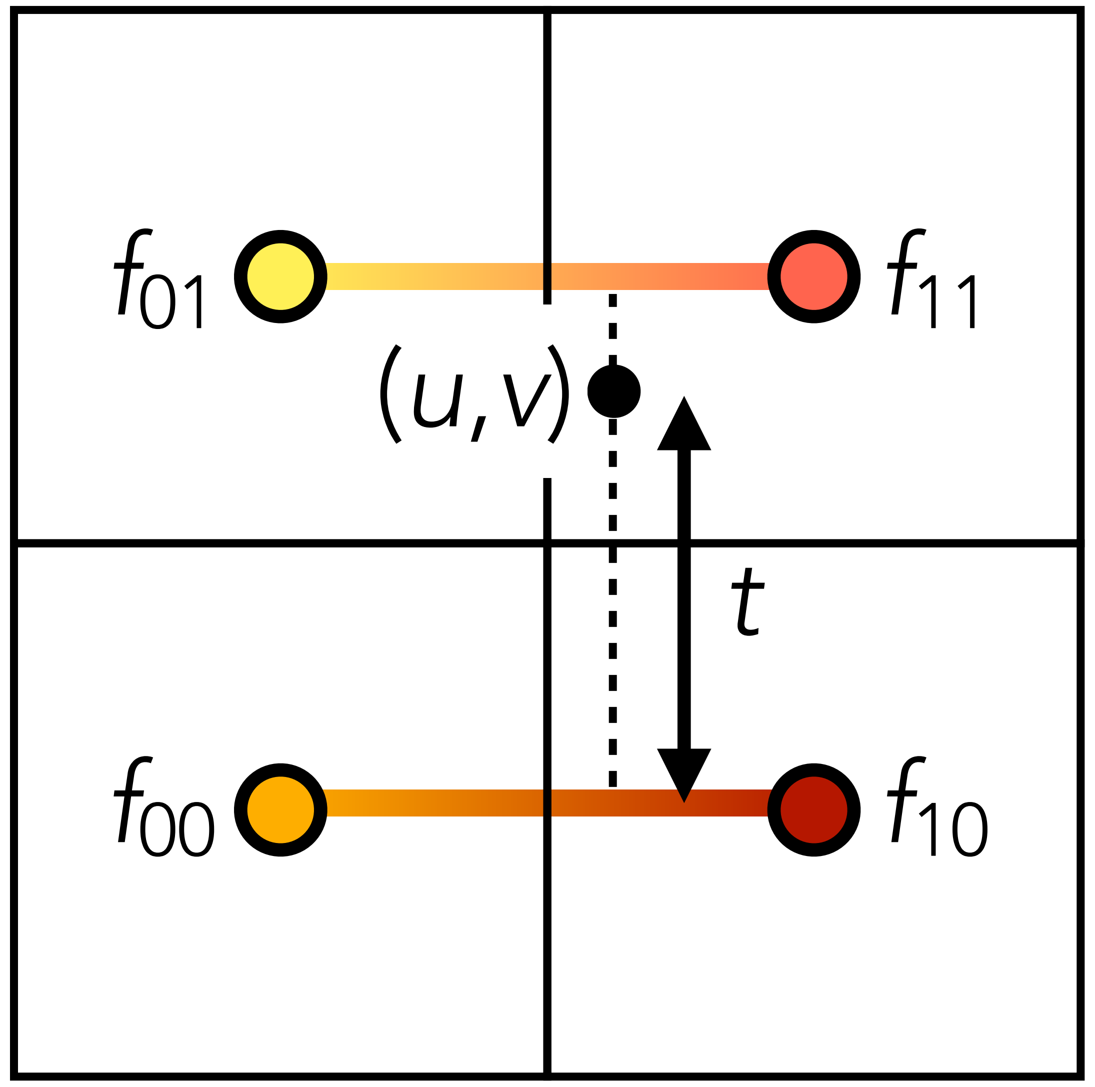
$$f(u,v) = \text{lerp}(s, f_{00}, f_{10}) = (1-s)\, f_{00} + s\, f_{10}$$



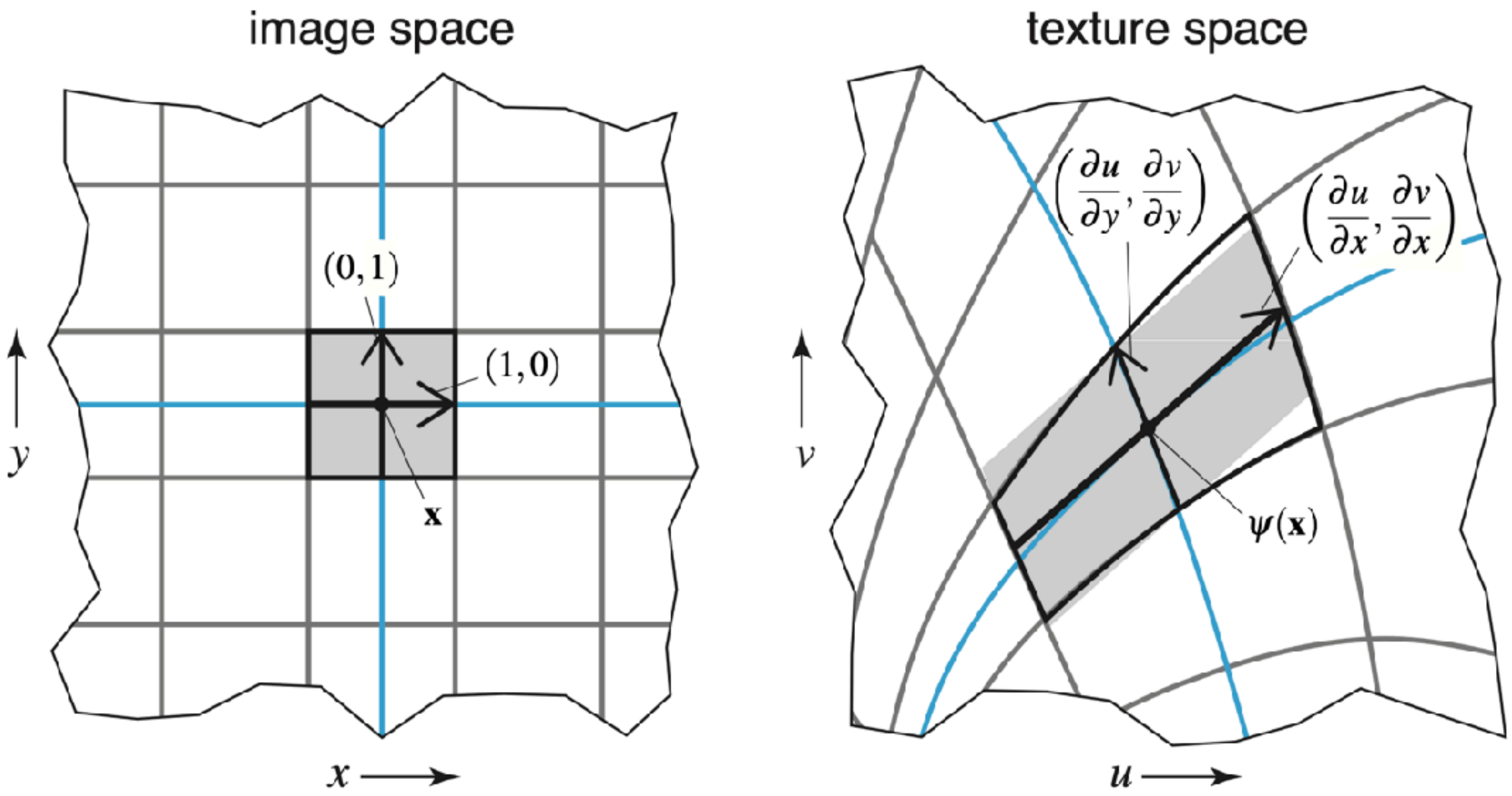
In general position:

$$f(u,v) = \text{lerp}(t, \text{lerp}(s, f_{00}, f_{10}), \text{lerp}(s, f_{01}, f_{11}))$$

$$= (1-s)(1-t)\, f_{00} + s\,(1-t)\, f_{10} + (1-s)\, t\, f_{01} + s\, t\, f_{01}$$



Minification: How to find a pixel's "footprint"?



Marschner and Shirley

Evaluated for each sample while rasterizing the triangle
(analytically… or just take differences with adjacent pixels)

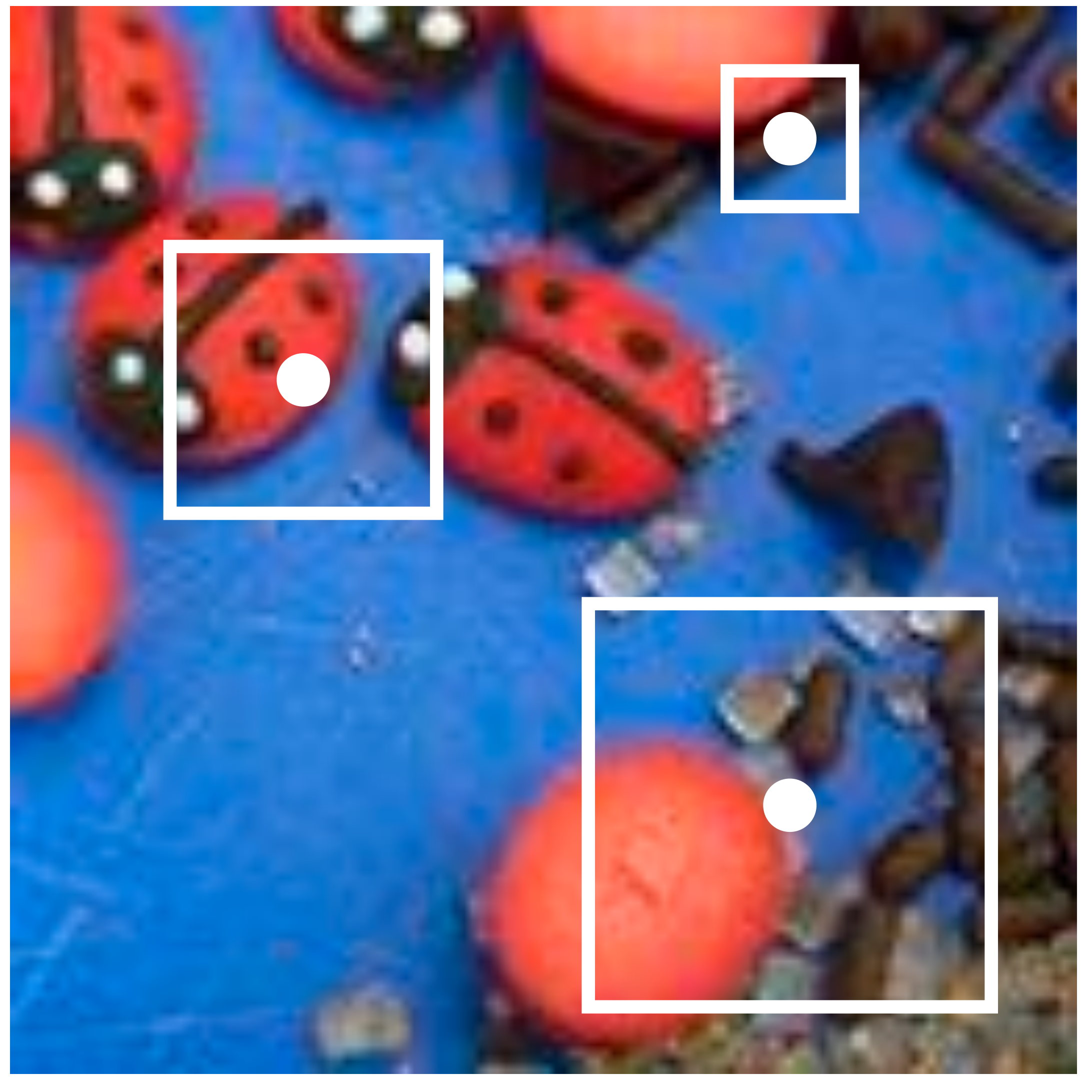
To start, let's assume the footprint is square with side D

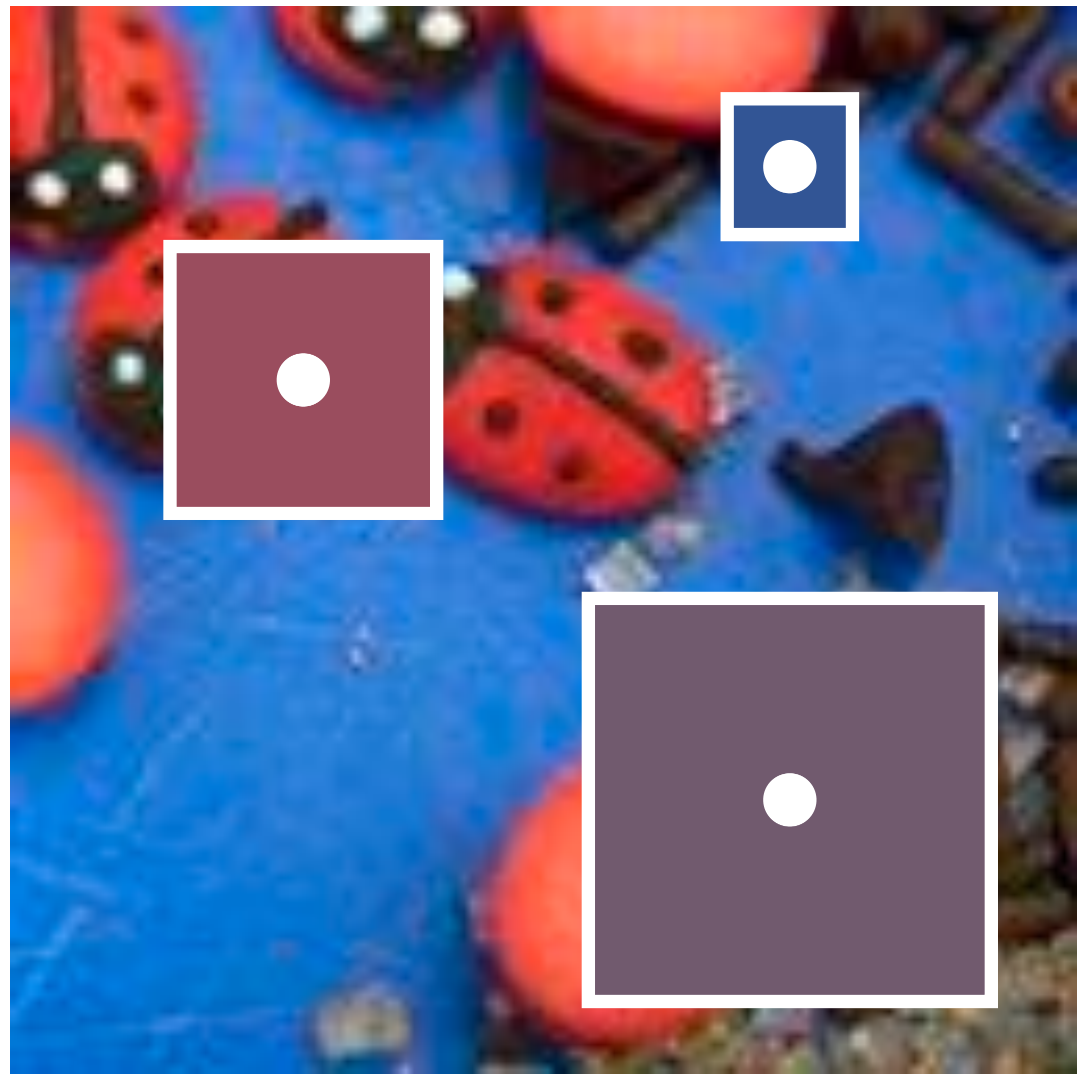
⇒ Need to compute (weighted?) average of D^2 texels!

Solution:

- Precompute filtered (blurred) version of texture
- For each sample, look up just 1 texel in filtered image

But D will be different for different pixels…





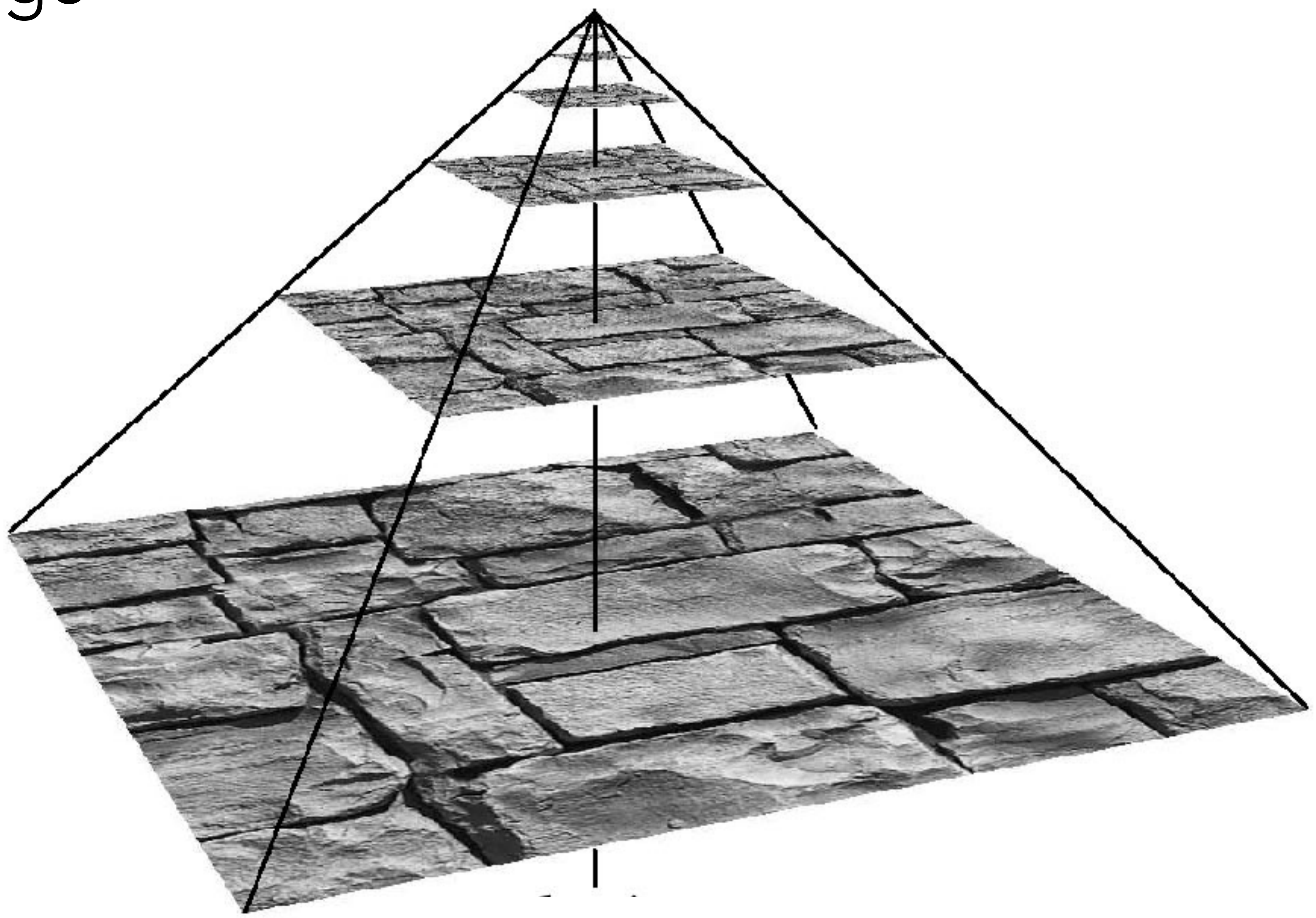
Mipmaps

Store pre-filtered versions of texture image for many different filter sizes

(Basically the same as image pyramids in image processing / computer vision)

Compute recursively by averaging and downsampling

Proposed by Lance Williams in 1983.
MIP = *multum in parvo* ("much in little")




Level 0 (128×128)
Level 1 (64×64)
Level 2 (32×32)
Level 3 (16×16)
Level 4 (8×8)
Level 5 (4×4)
Level 6 (2×2)
Level 7 (1×1)

Ren Ng

Everything at level 0 (no filtering)

Everything at level 2 (downsampled by 4x)

Everything at level 4 (downsampled by 16x)

Using the mipmap

1 texel at level $k \approx$ square of width 2^k texels in original texture

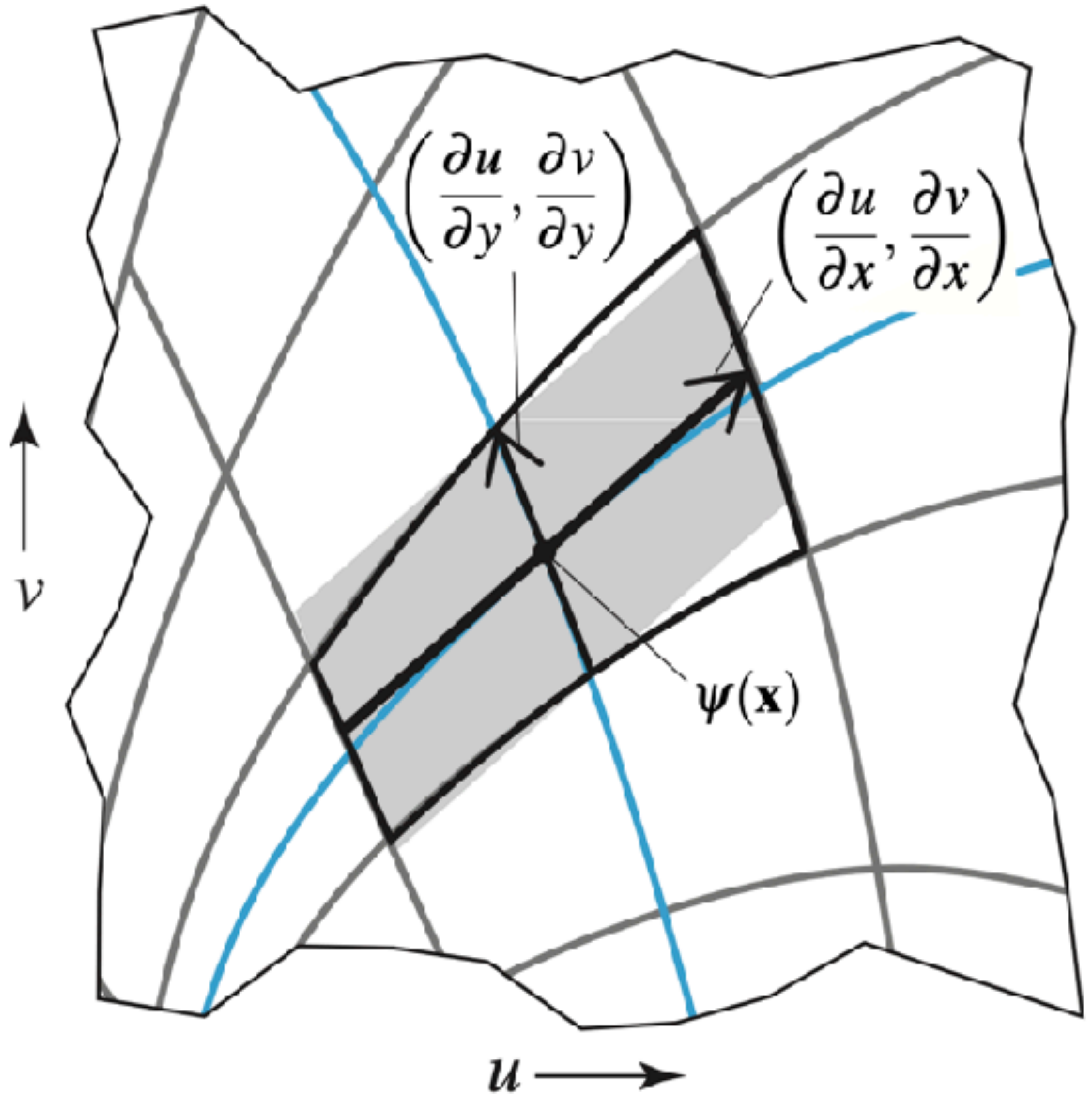
So if pixel footprint is square of width D, look up mipmap at level $k = \log_2 D$

How to compute "width" in general?

$$D = \max(|du/dx|, |dv/dx|, |du/dy|, |dv/dy|)$$

$$D = \max\left(\sqrt{(du/dx)^2 + (dv/dx)^2}, \sqrt{(du/dy)^2 + (dv/dy)^2}\right)$$

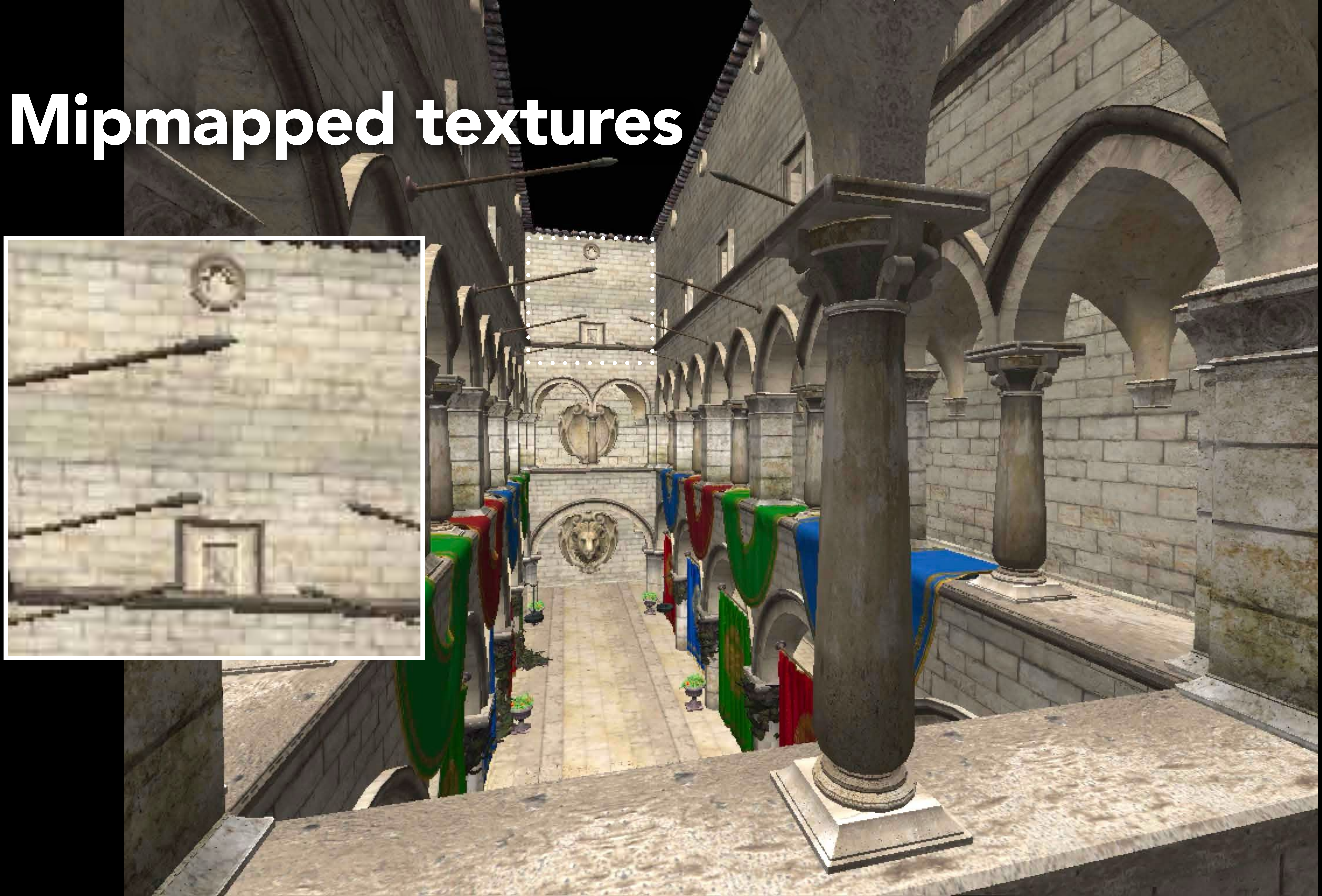
(Why max and not min or average?)



Visualization of mipmap level

Mipmap level $k = \log_2 D$ rounded to nearest integer

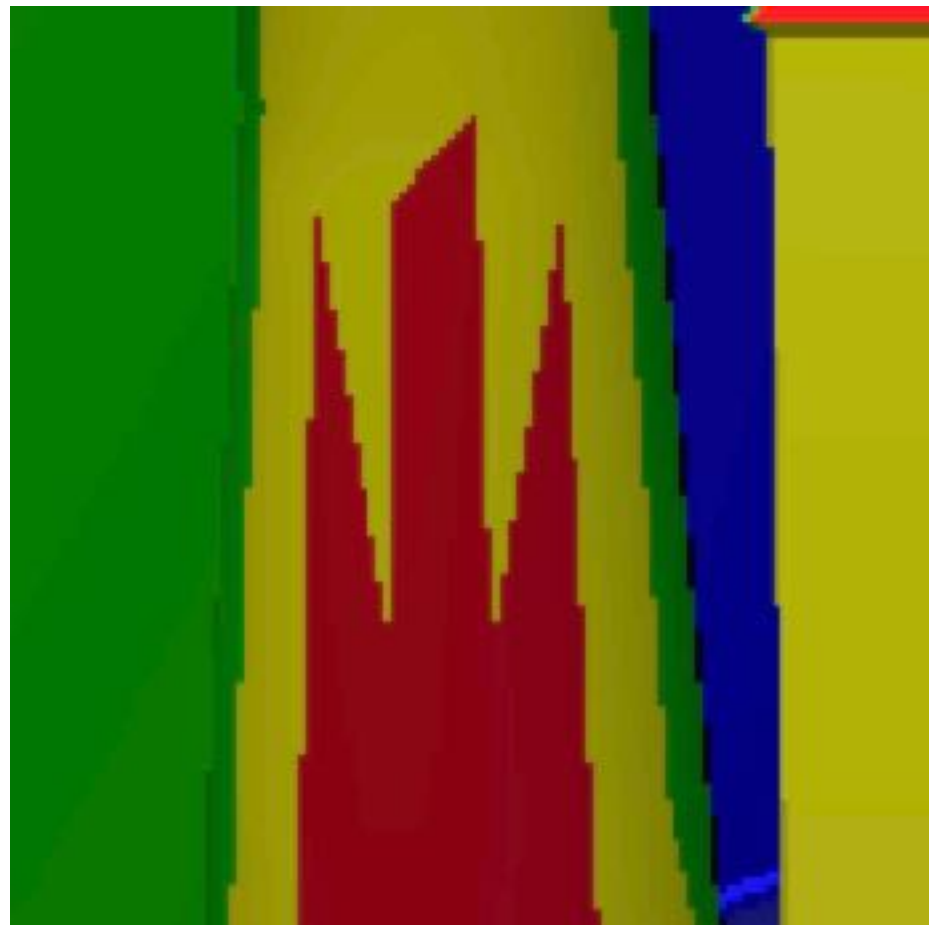
Mipmapped textures



Visualization of mipmap level

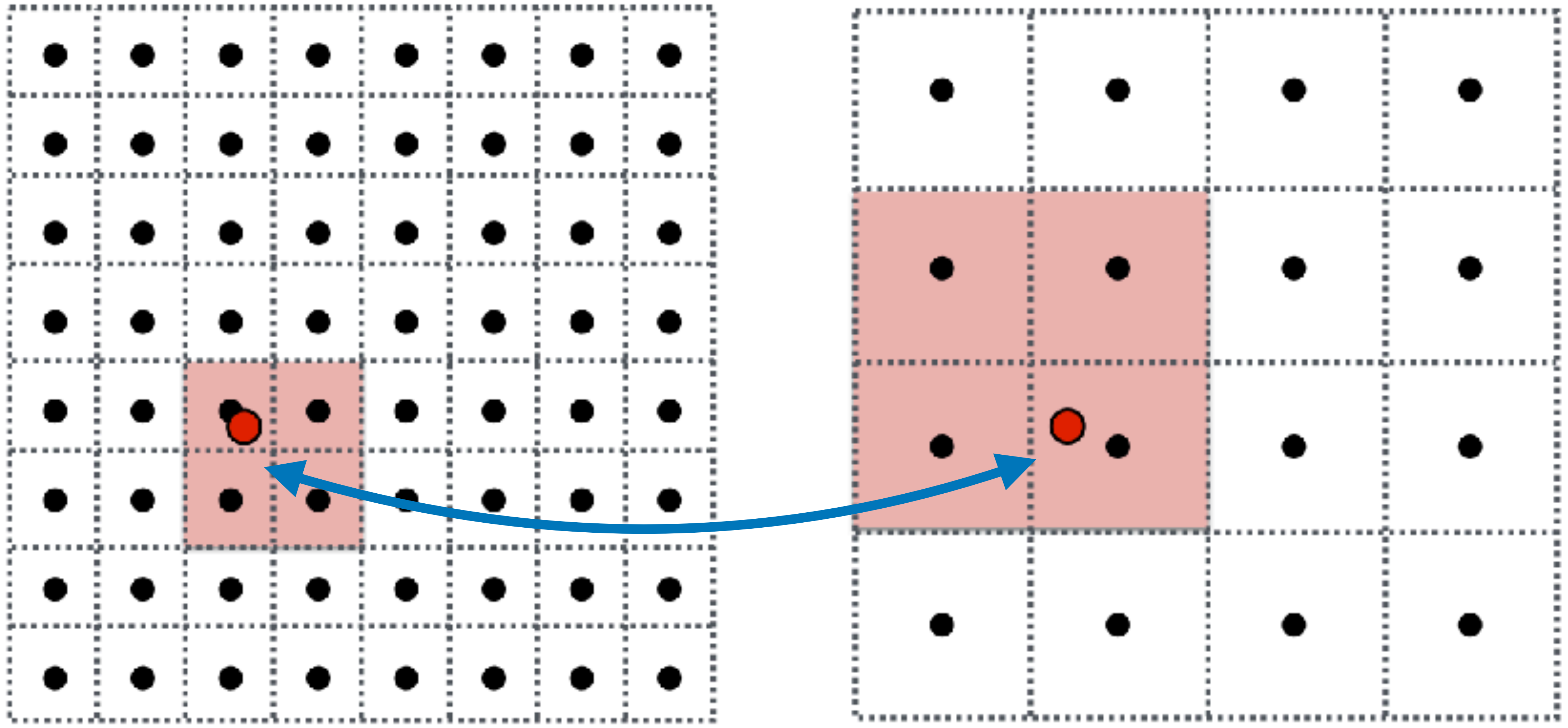
Continuous mipmap level $k = \log_2 D$

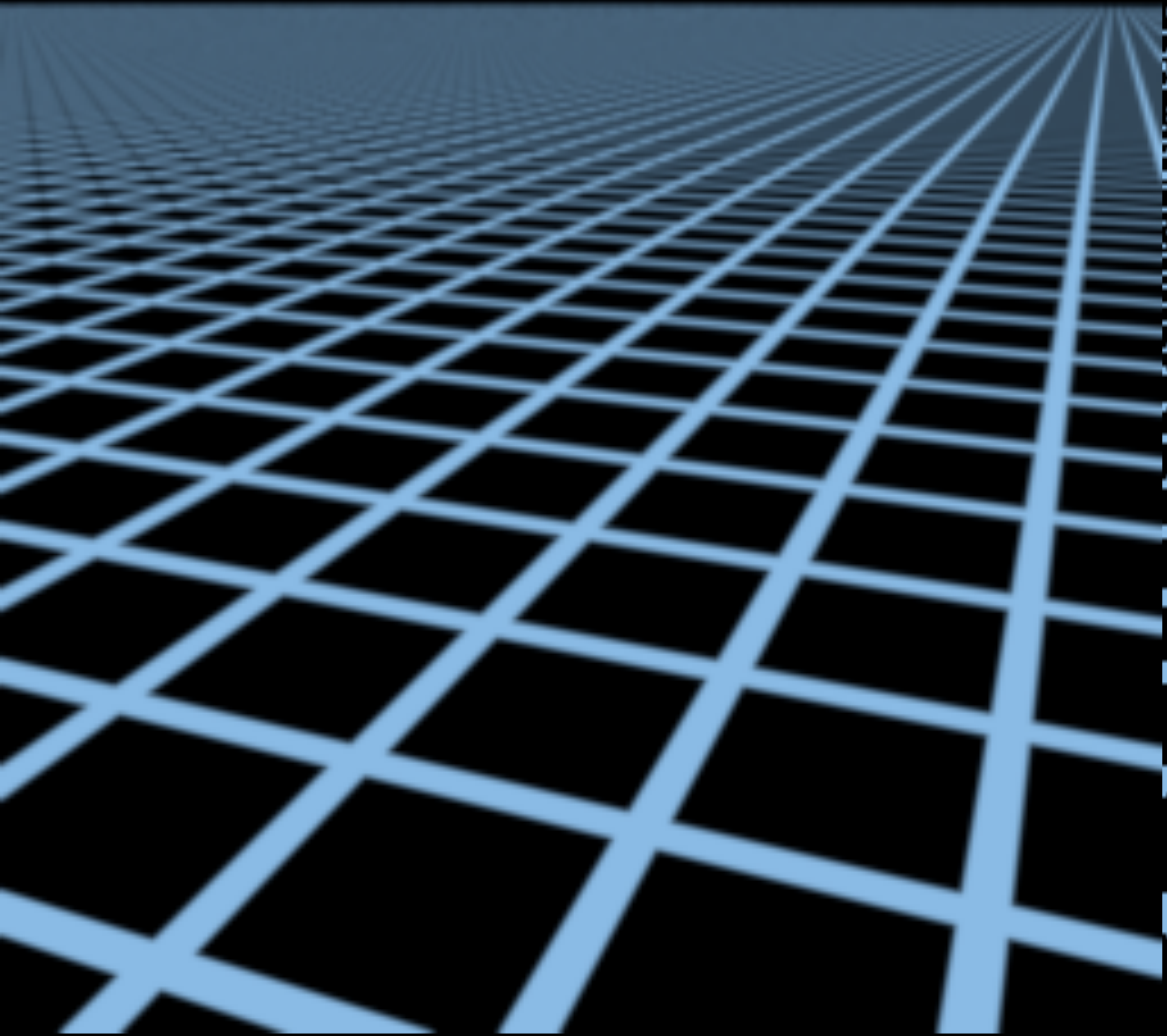
Basic mipmapping produces discontinuous "jumps" in texture detail

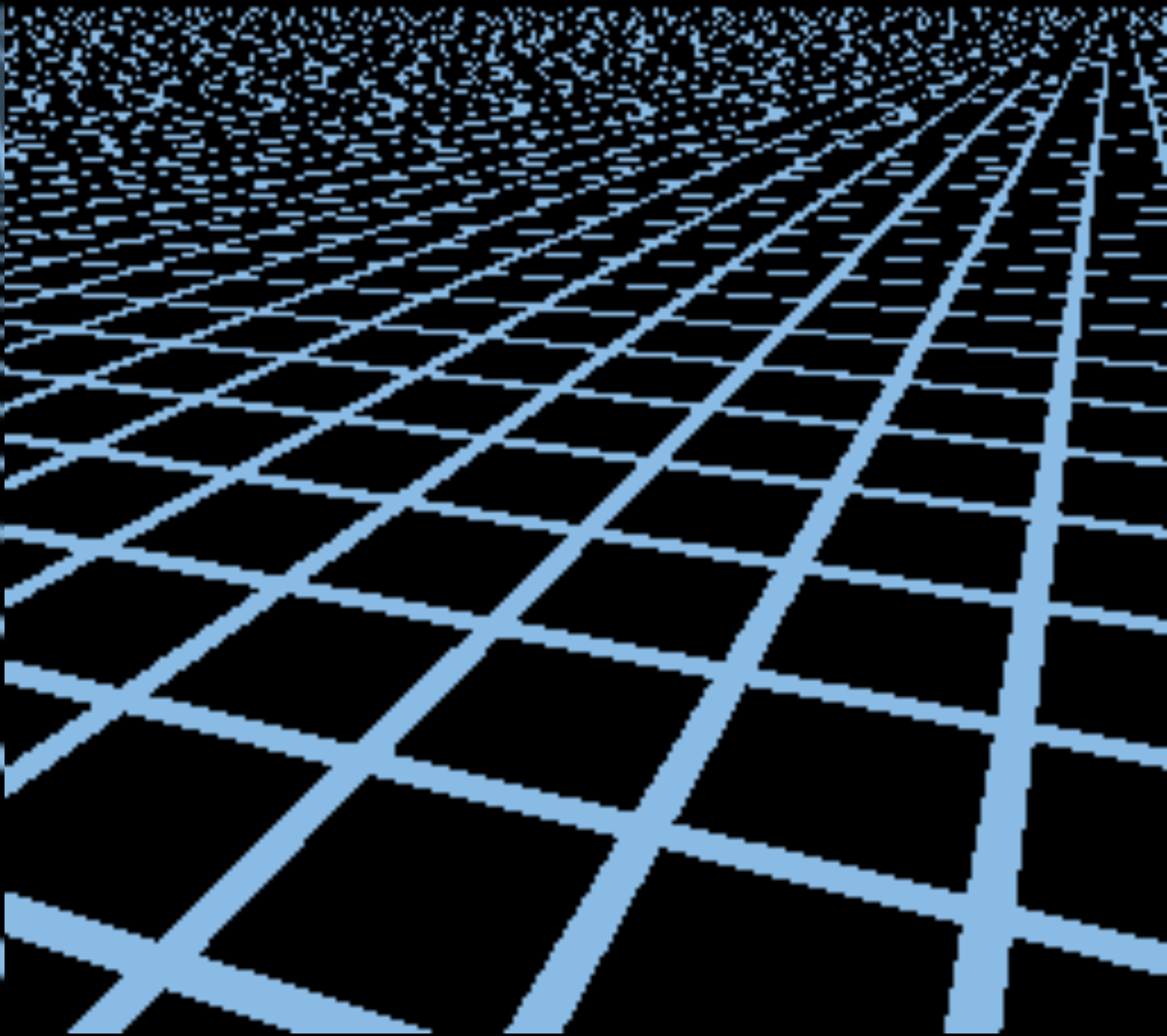


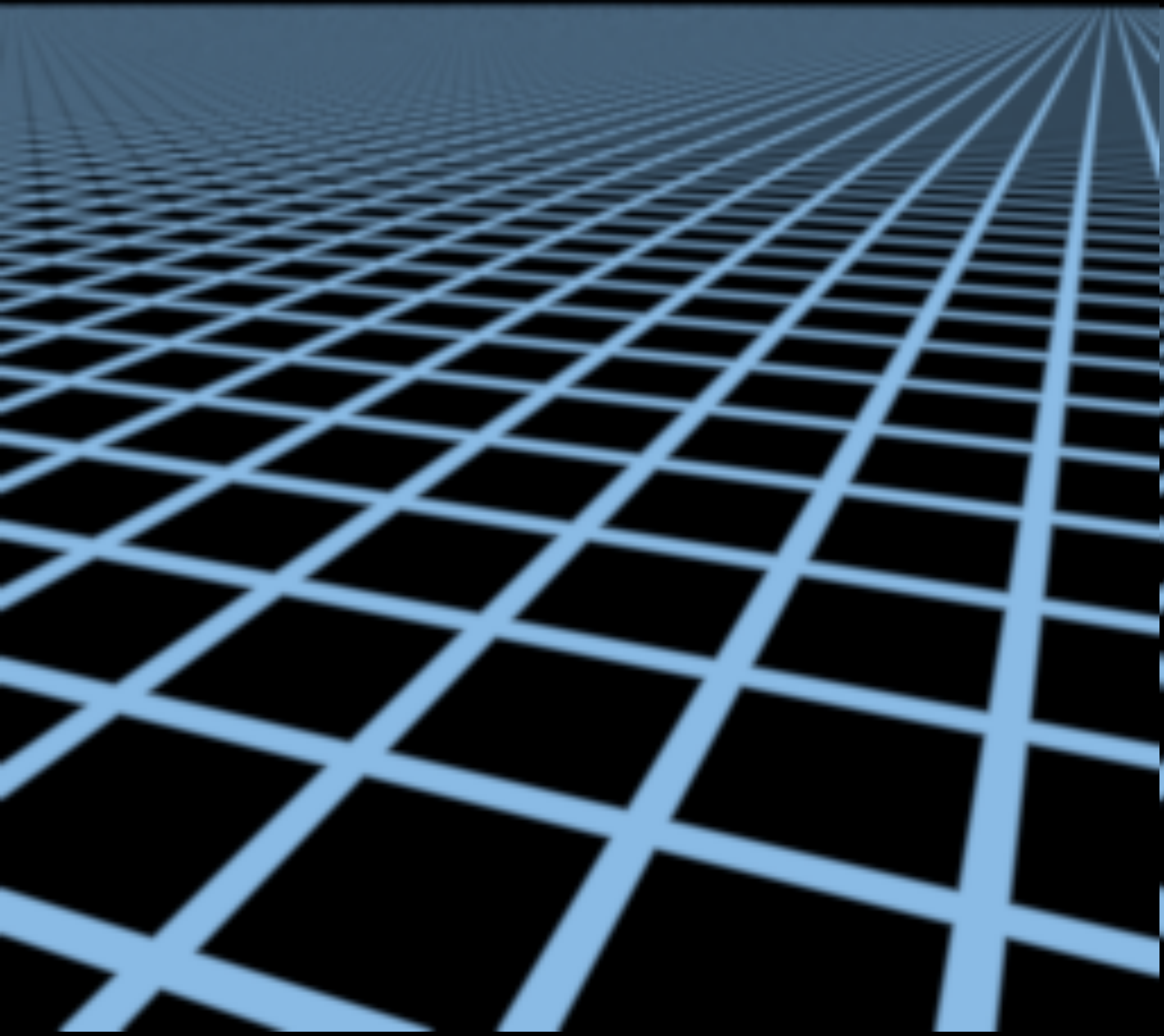
Trilinear filtering: interpolate between results of two adjacent mipmap levels

- Bilinear interpolation at level $\lfloor k \rfloor$
- Bilinear interpolation at level $\lfloor k \rfloor + 1$
- Linear interpolation between them

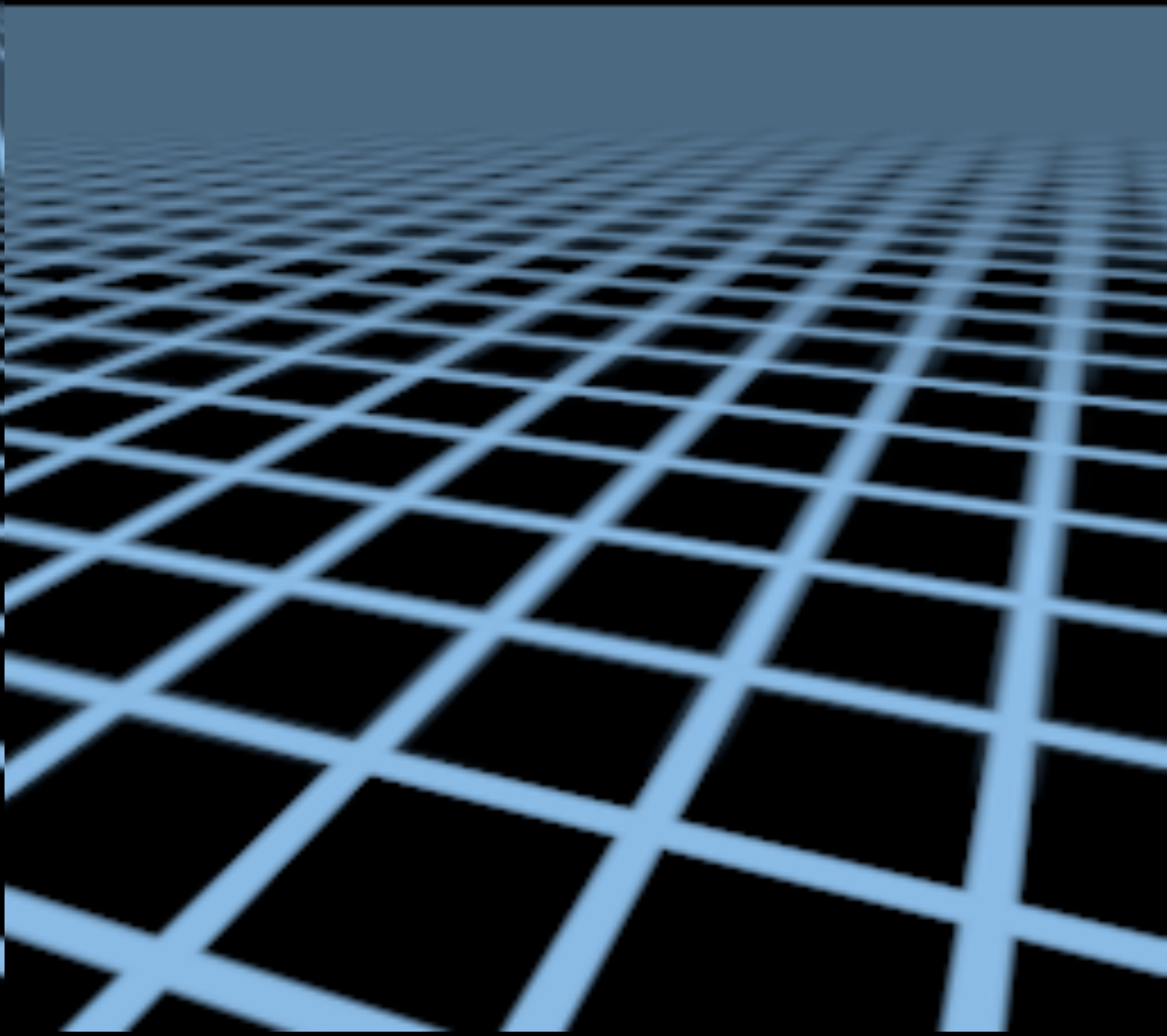



Supersampled reference (256×256, 512 spp)


Point sampling (256×256)

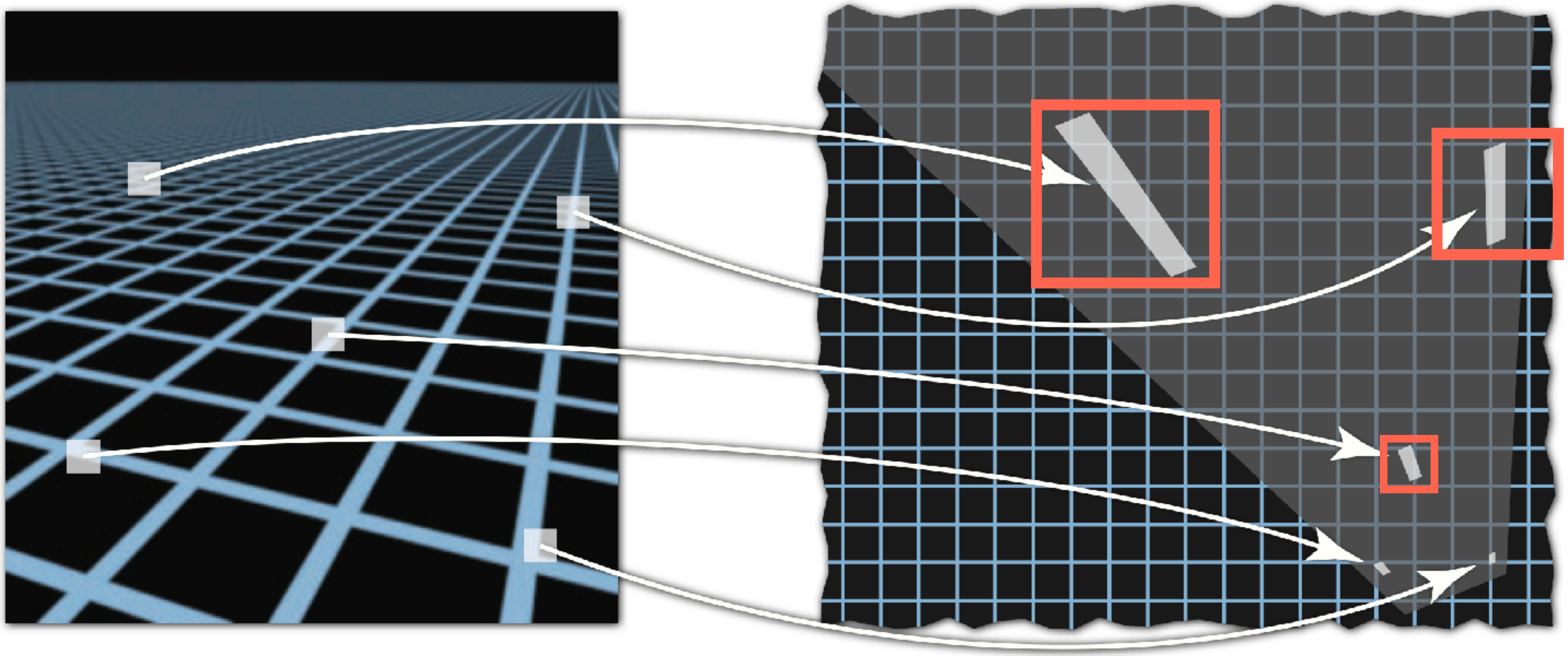


Supersampled reference (256×256, 512 spp)

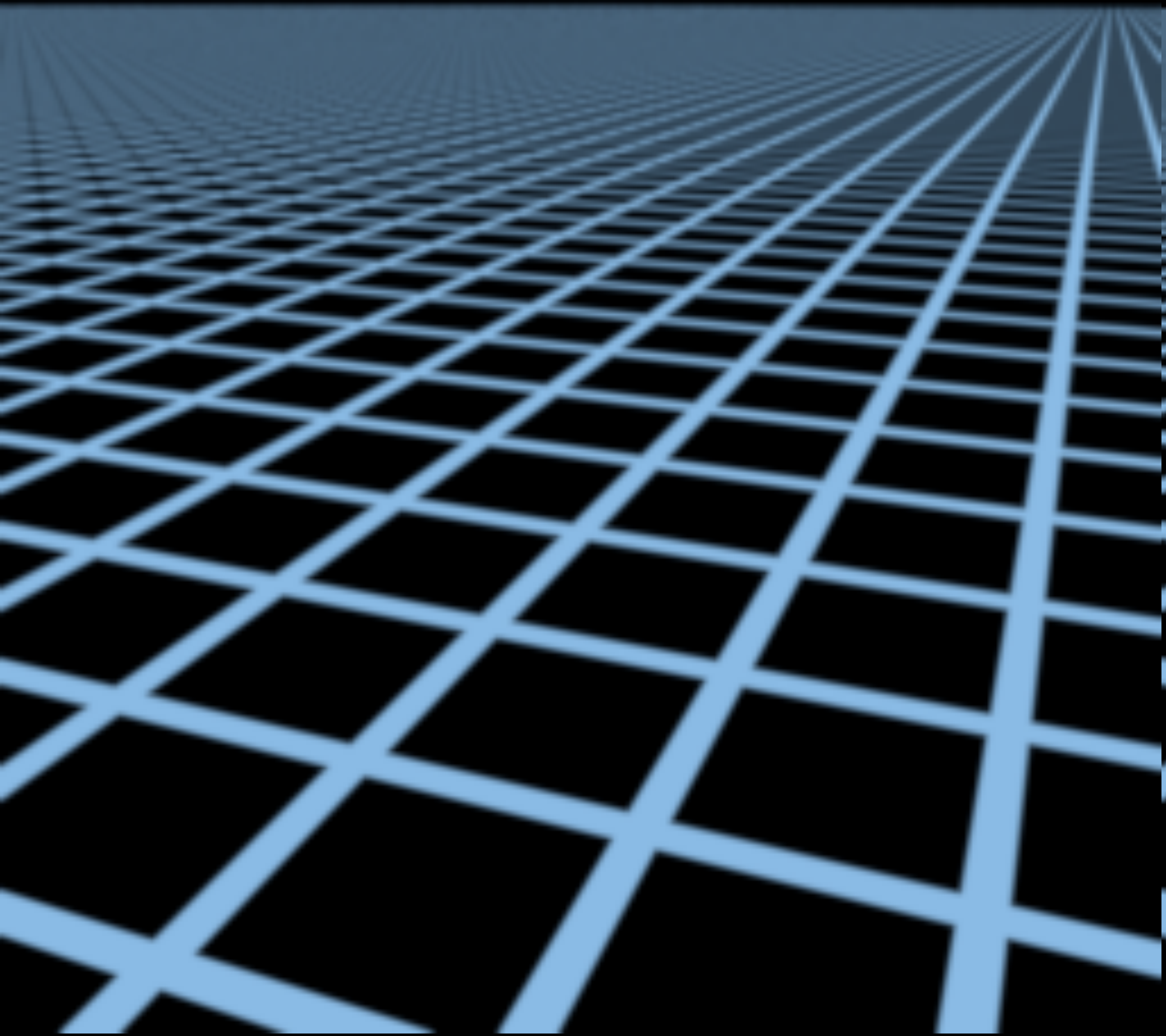


Mipmap with trilinear filtering

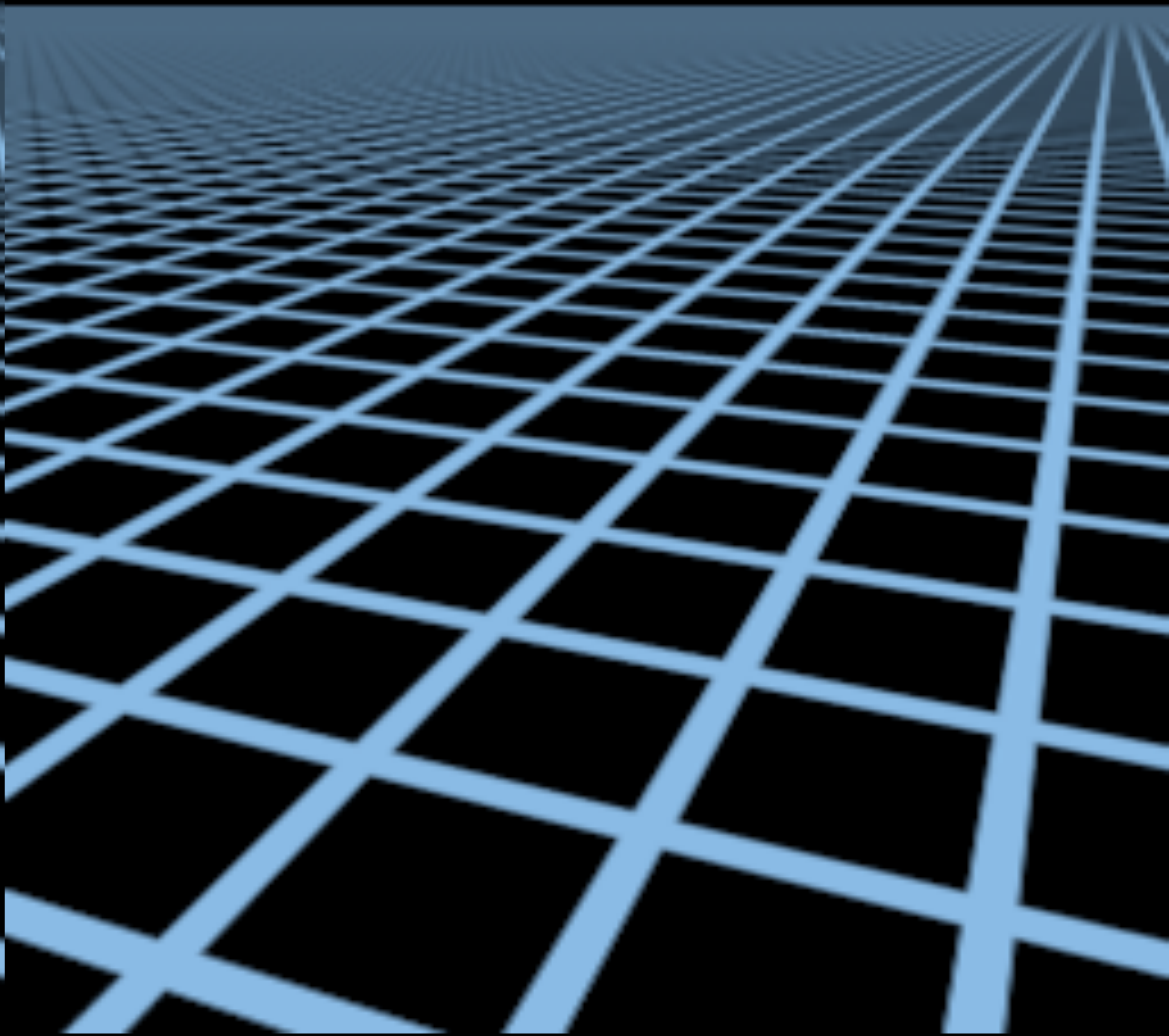
At grazing angles, pixel footprint is very stretched out!



Mipmaps only allow isotropic filtering (same in all directions)



Supersampled reference (256×256, 512 spp)

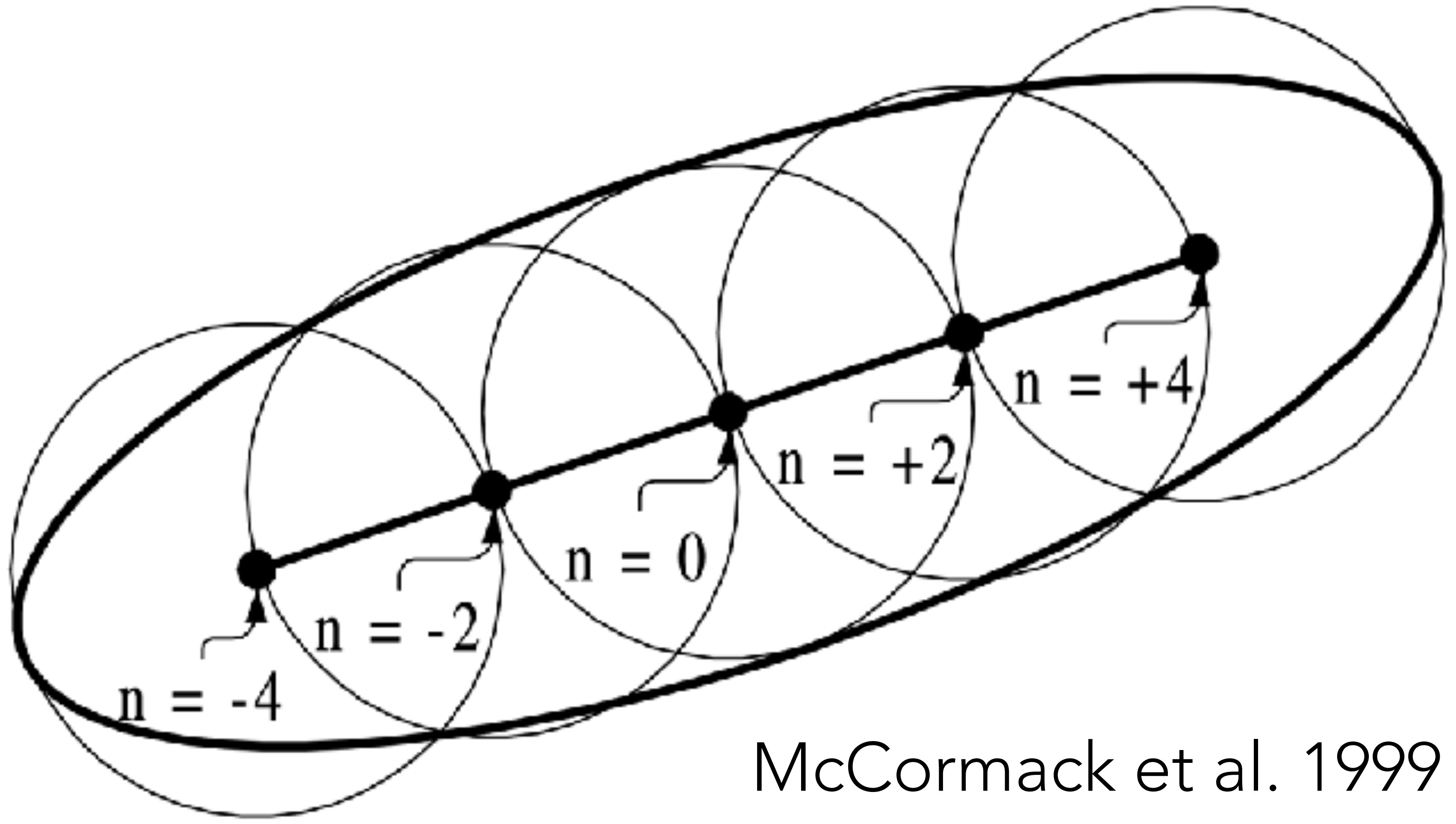


Elliptical weighted average (EWA)

Anisotropic filtering

Not on the exam :)

Treat pixel as circular (e.g. Gaussian kernel)
→ maps to ellipse in texture space
→ approximate as line of blobs



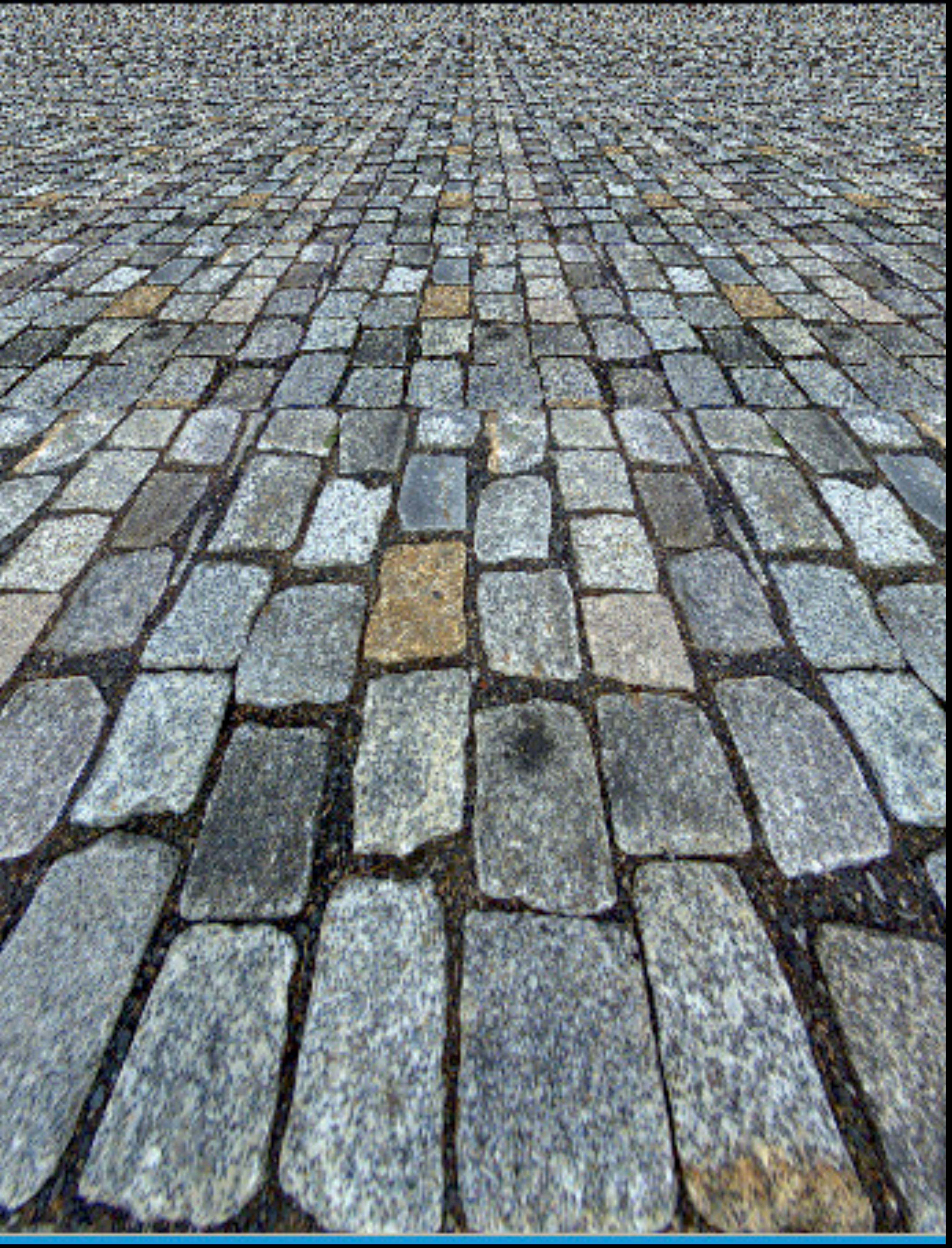
McCormack et al. 1999

Choose mipmap level using minor axis

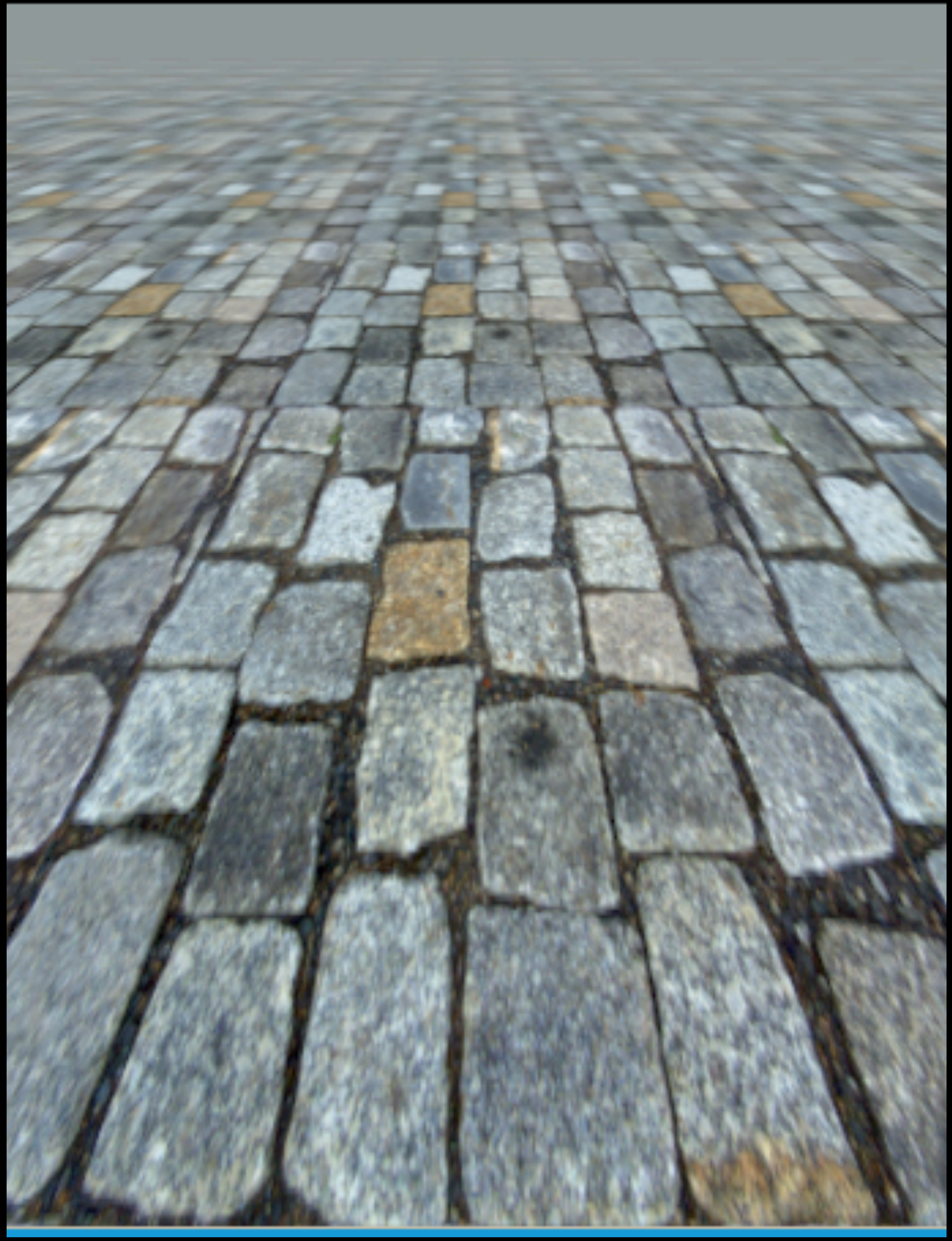
Take multiple samples along major axis

This is what GPUs do when they say e.g. “16x anisotropic filtering”

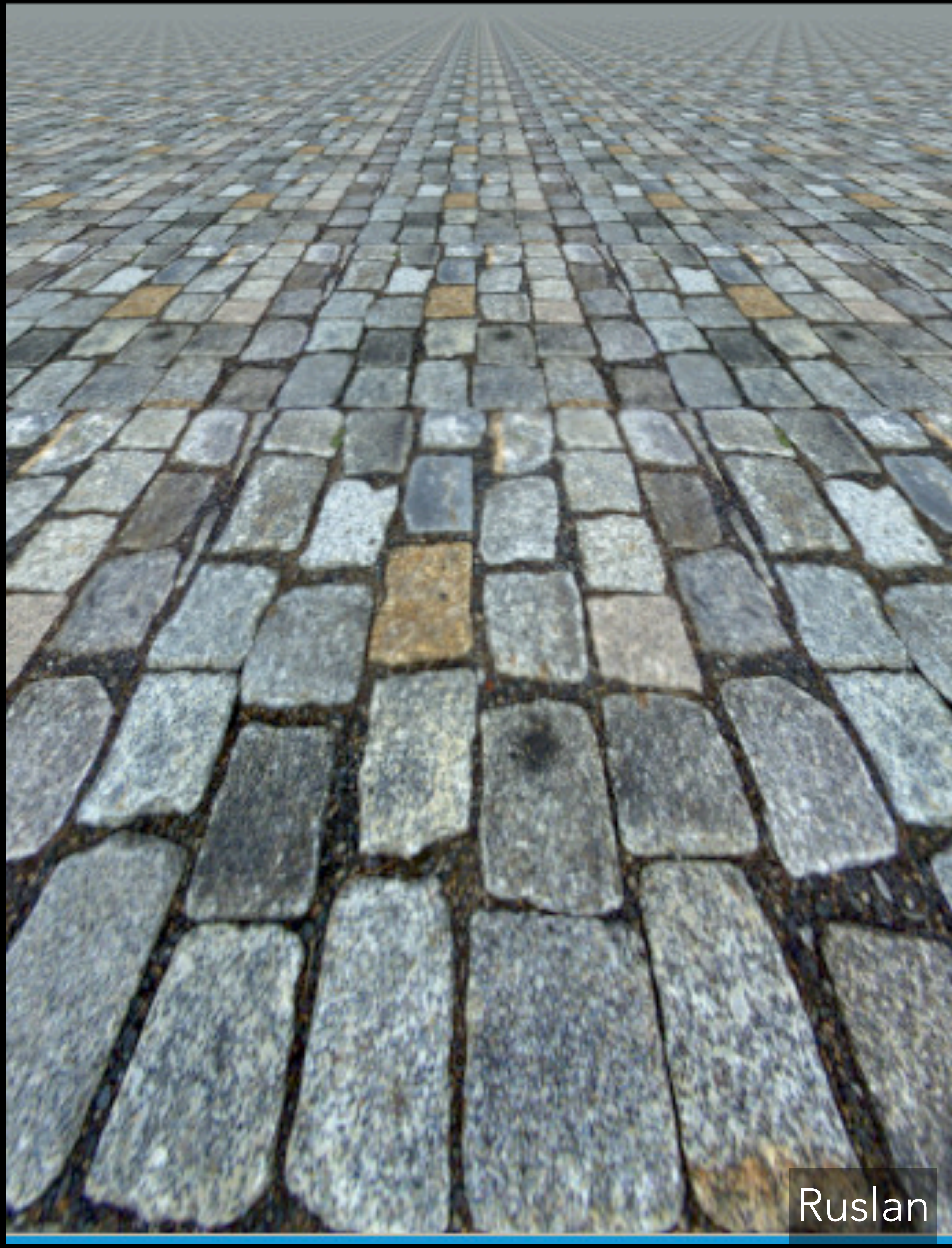
[Original idea by Greene and Heckbert 1986, faster approximation using mipmaps by McCormack et al. 1999]



No filtering



Mipmapping



Anisotropic filtering

Homework

Modify the starter code to draw this:

```
vertices = {
    (-0.8,  0.0, 0.0, 1.0),
    (-0.4, -0.8, 0.0, 1.0),
    ( 0.8,  0.8, 0.0, 1.0),
    (-0.4, -0.4, 0.0, 1.0)
};
indices[] = {
    (0, 1, 3),
    (1, 2, 3)
};
```