

# **Antialiasing**

**CS4620 Lecture 22**

# Pixel coverage

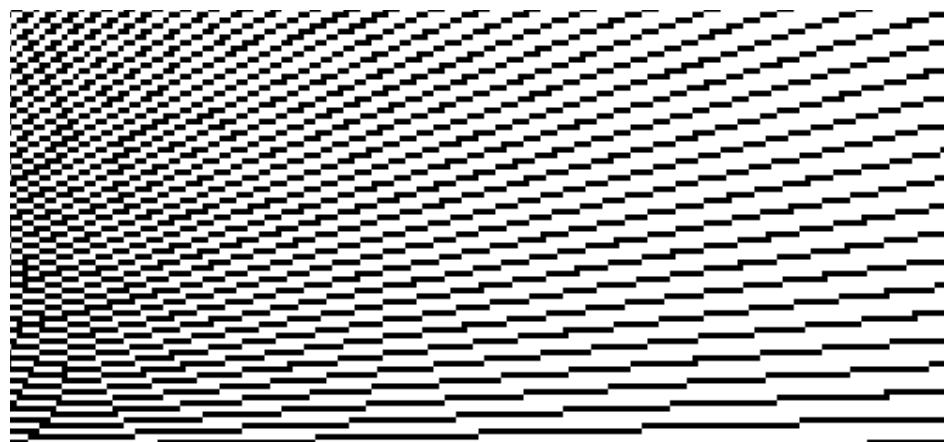
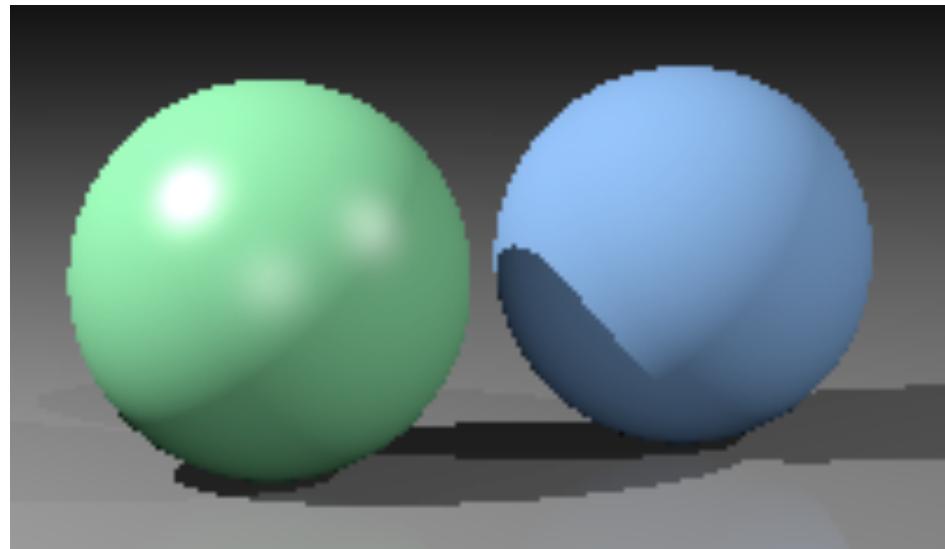
- Antialiasing and compositing both deal with questions of pixels that contain unresolved detail
- Antialiasing: how to carefully throw away the detail
- Compositing: how to account for the detail when combining images

# Aliasing

point sampling a  
continuous image:

continuous image defined  
by ray tracing procedure

continuous image defined  
by a bunch of black rectangles

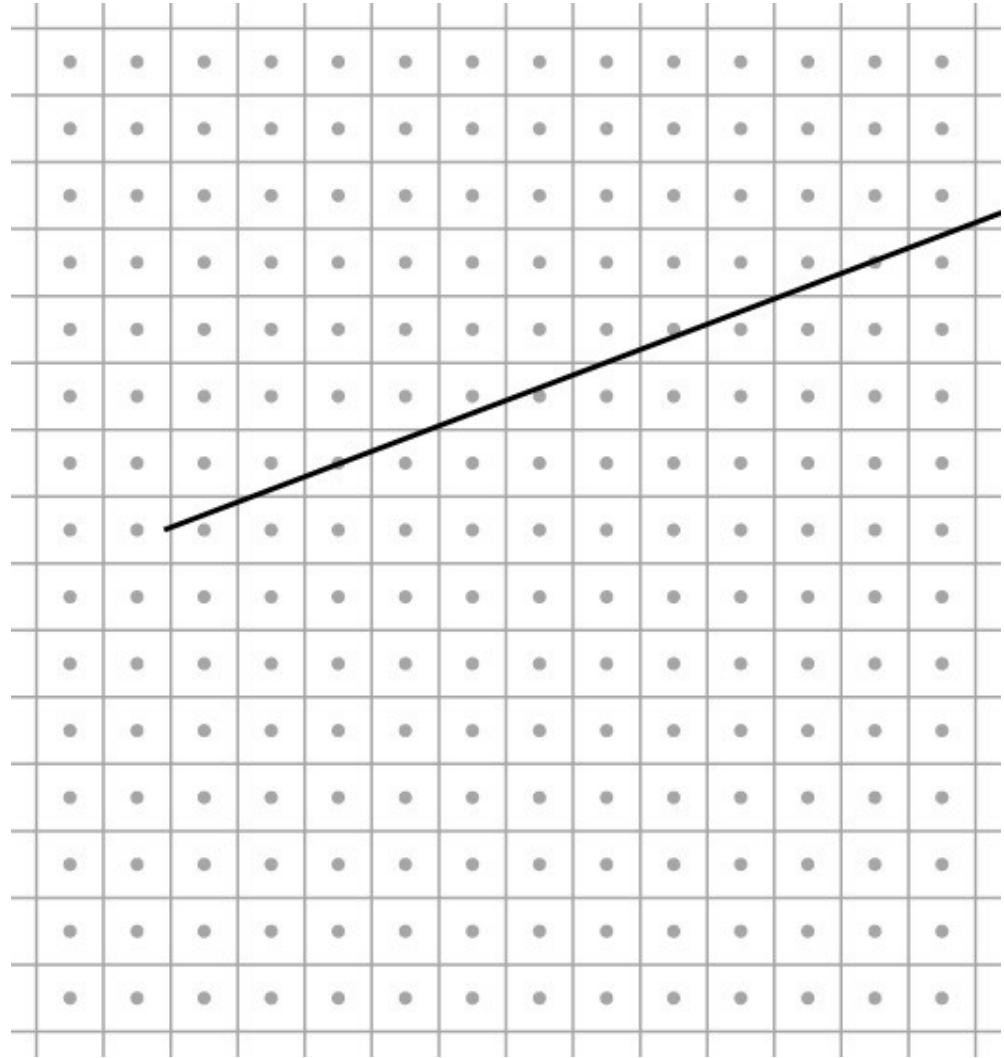


# Antialiasing

- A name for techniques to prevent aliasing
- In image generation, we need to *filter*
  - Boils down to averaging the image over an area
  - Weight by a filter
- Methods depend on source of image
  - Rasterization (lines and polygons)
  - Point sampling (e.g. raytracing)
  - Texture mapping

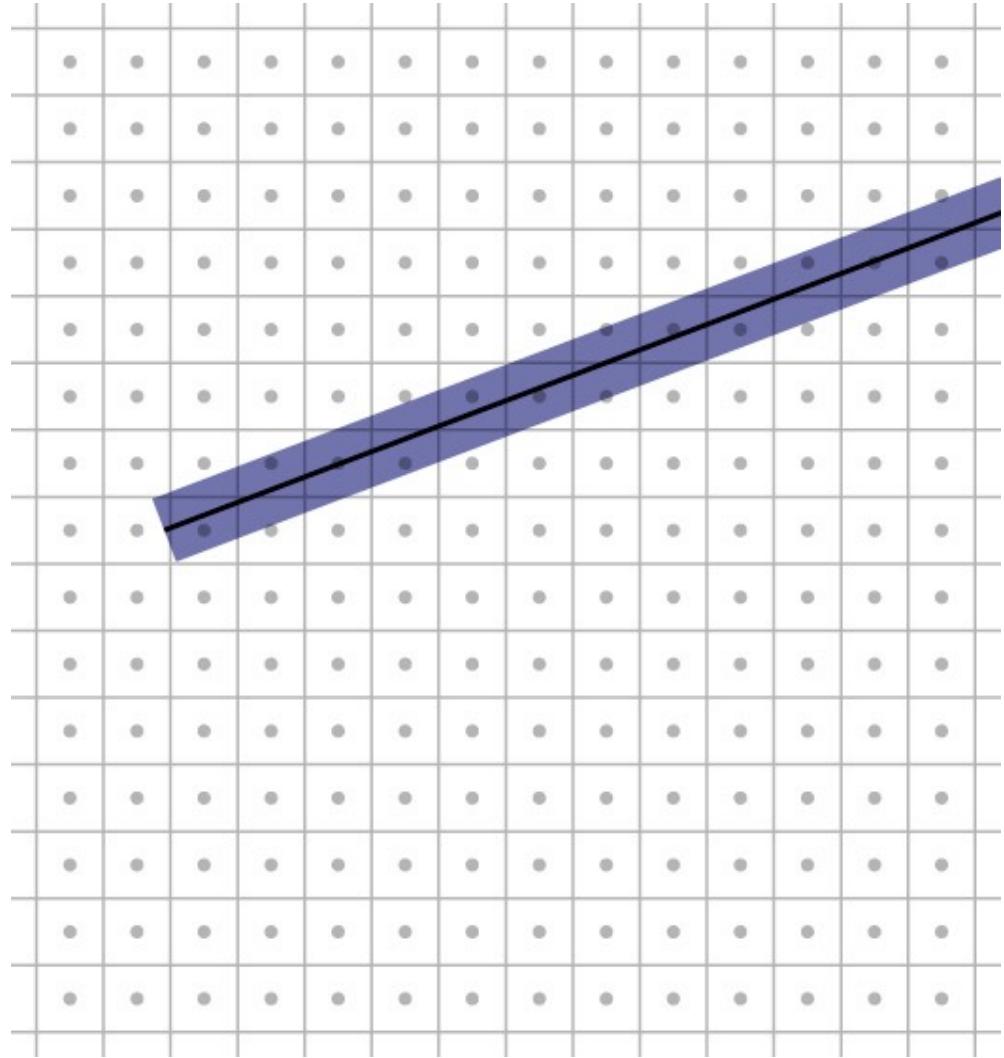
# Rasterizing lines

- Define line as a rectangle
- Specify by two endpoints
- Ideal image: black inside, white outside



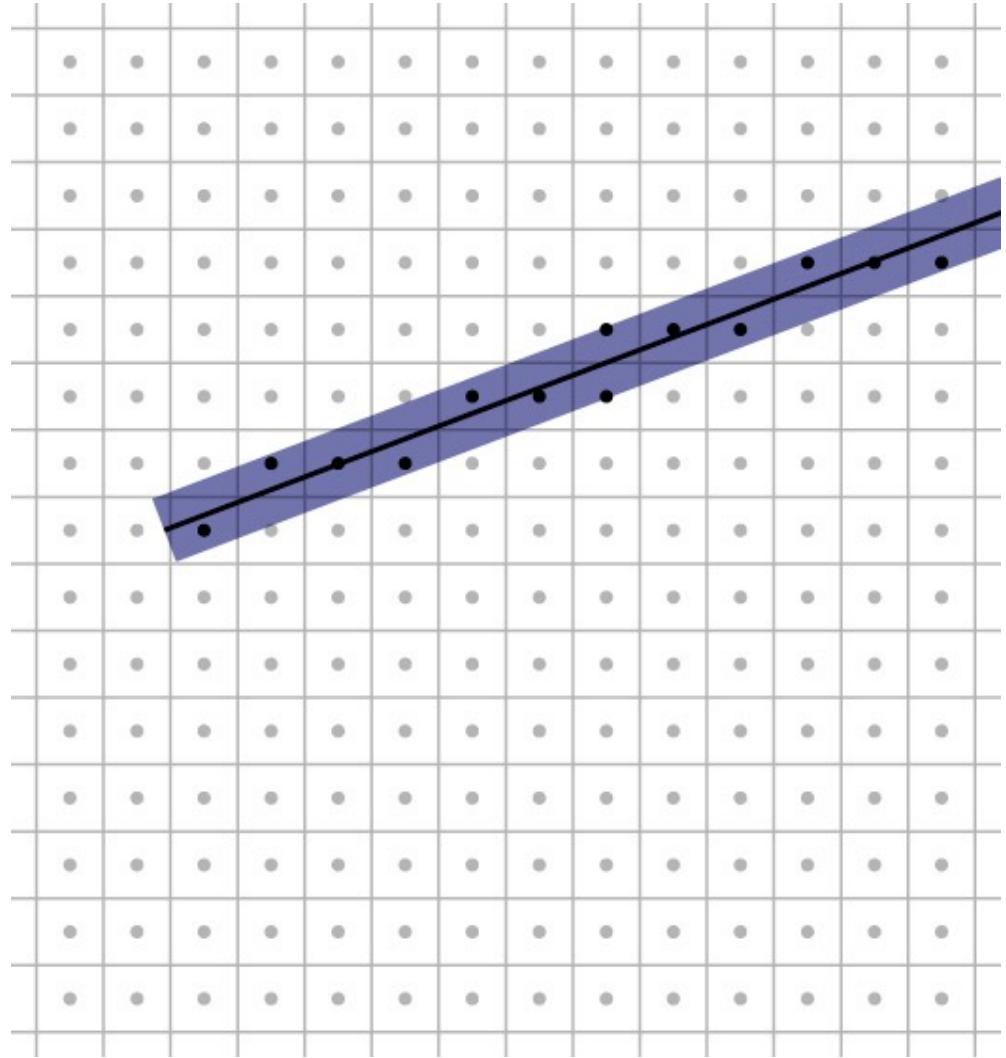
# Rasterizing lines

- Define line as a rectangle
- Specify by two endpoints
- Ideal image: black inside, white outside



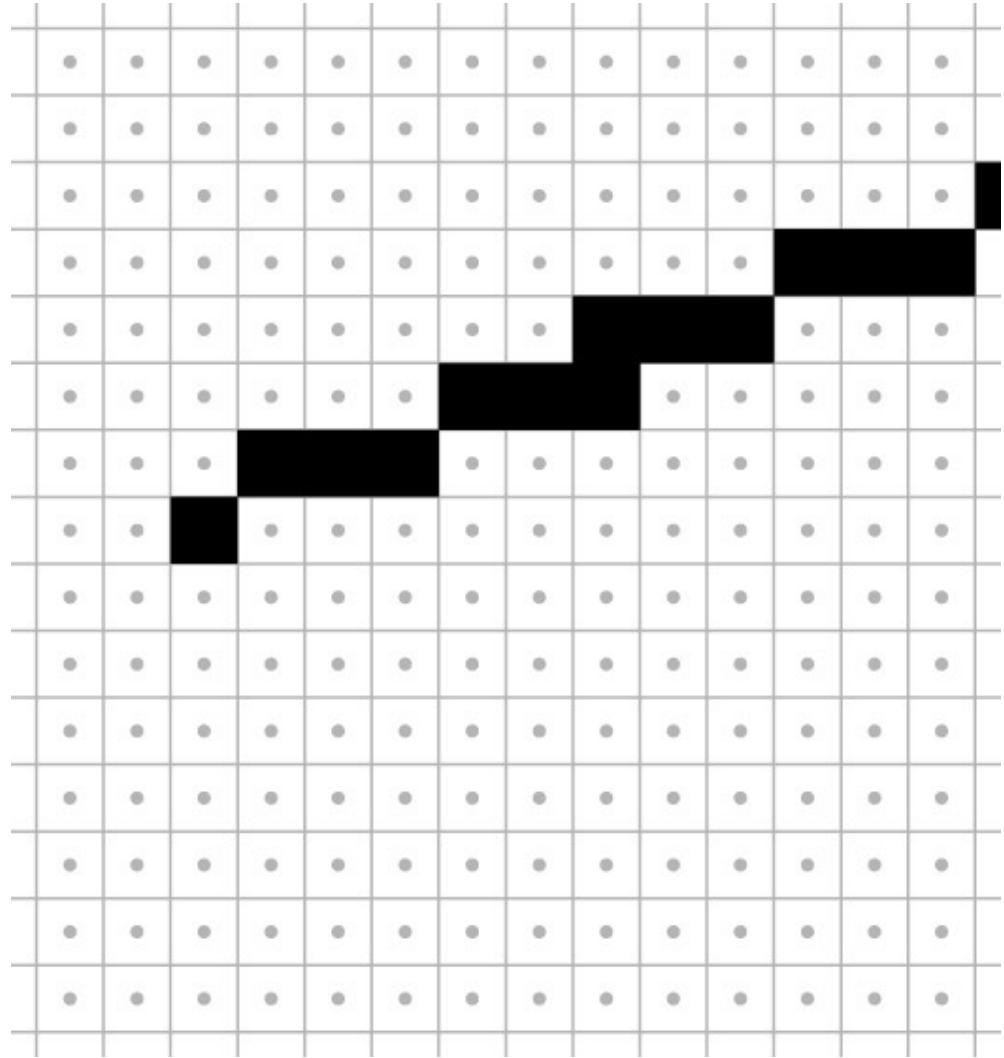
# Point sampling

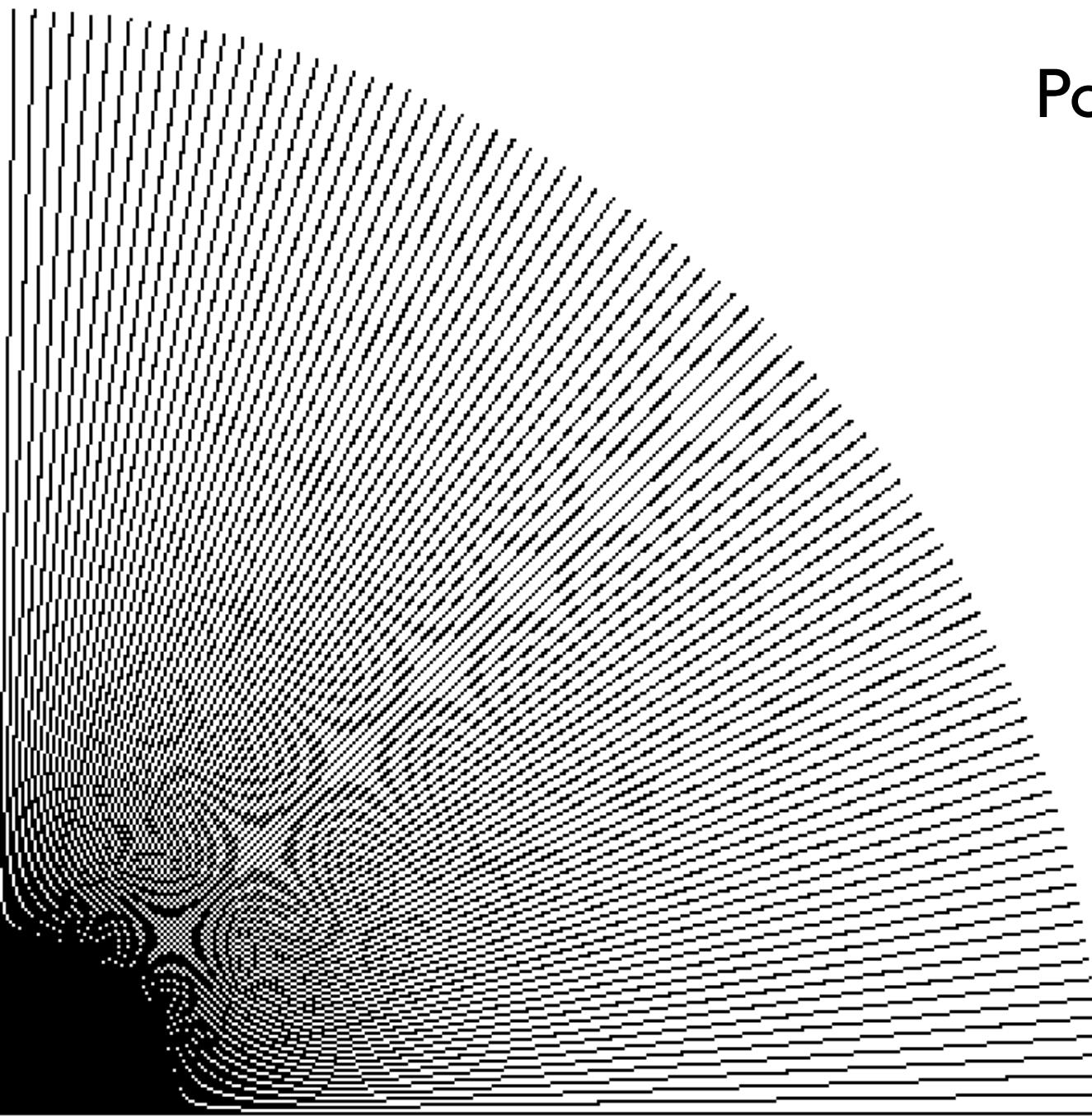
- Approximate rectangle by drawing all pixels whose centers fall within the line
- Problem: all-or-nothing leads to jaggies
  - this is sampling with no filter (aka. point sampling)



# Point sampling

- Approximate rectangle by drawing all pixels whose centers fall within the line
- Problem: all-or-nothing leads to jaggies
  - this is sampling with no filter (aka. point sampling)

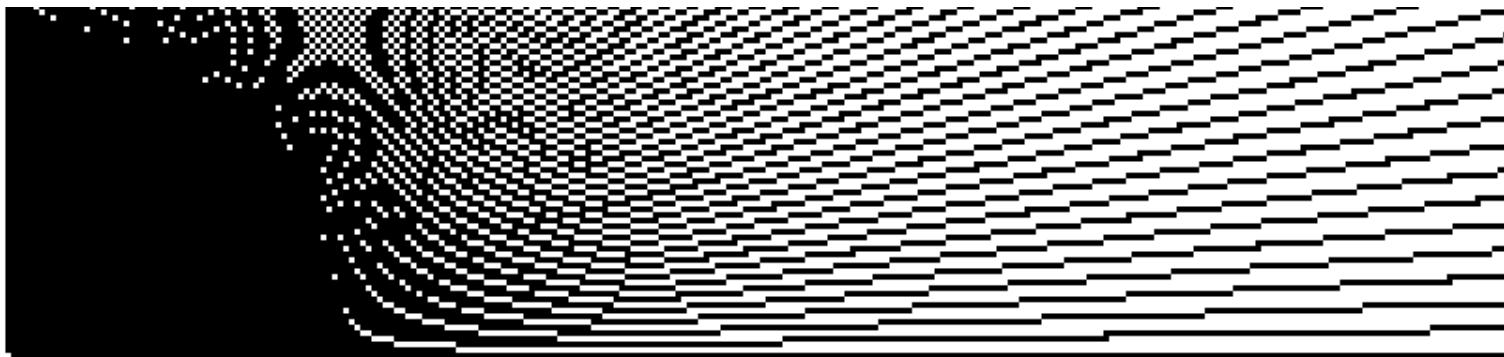




# Point sampling in action

# Aliasing

- Point sampling is fast and simple
- But the lines have stair steps and variations in width
- This is an aliasing phenomenon
  - Sharp edges of line contain high frequencies
- Introduces features to image that are not supposed to be there!

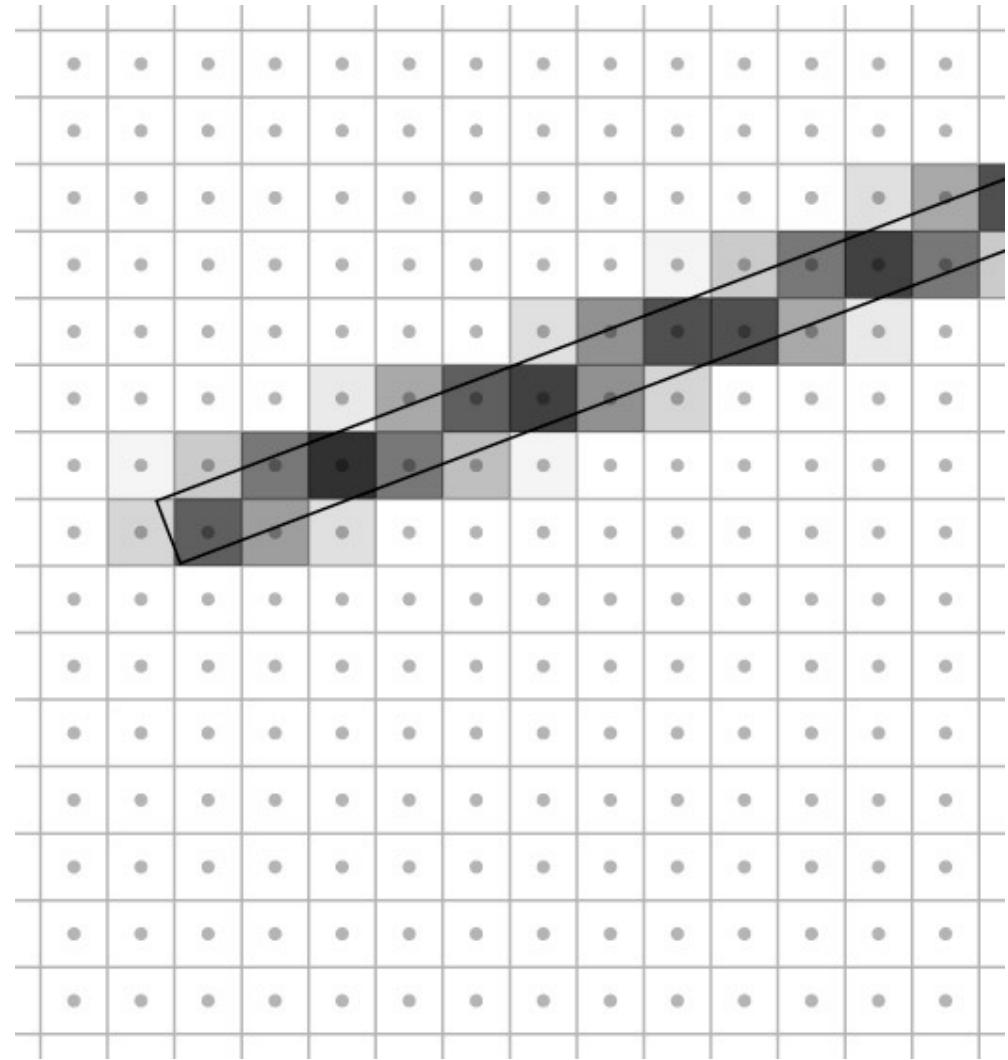


# Antialiasing

- Point sampling makes an all-or-nothing choice in each pixel
  - therefore steps are inevitable when the choice changes
  - yet another example where discontinuities are bad
- On bitmap devices this is necessary
  - hence high resolutions required
  - 600+ dpi in laser printers to make aliasing invisible
- On continuous-tone devices we can do better

# Antialiasing

- Basic idea: replace “is the image black at the pixel center?” with “how much is pixel covered by black?”
- Replace yes/no question with quantitative question.

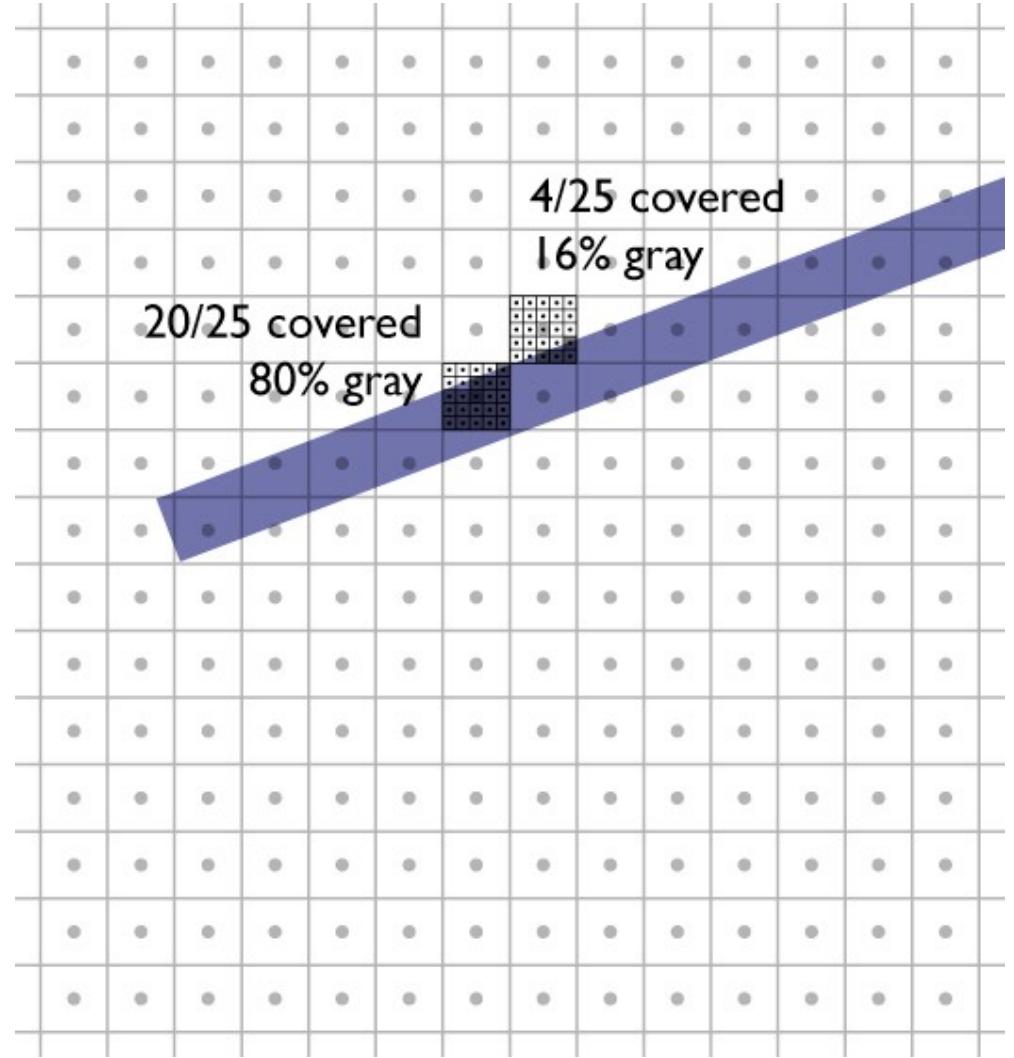


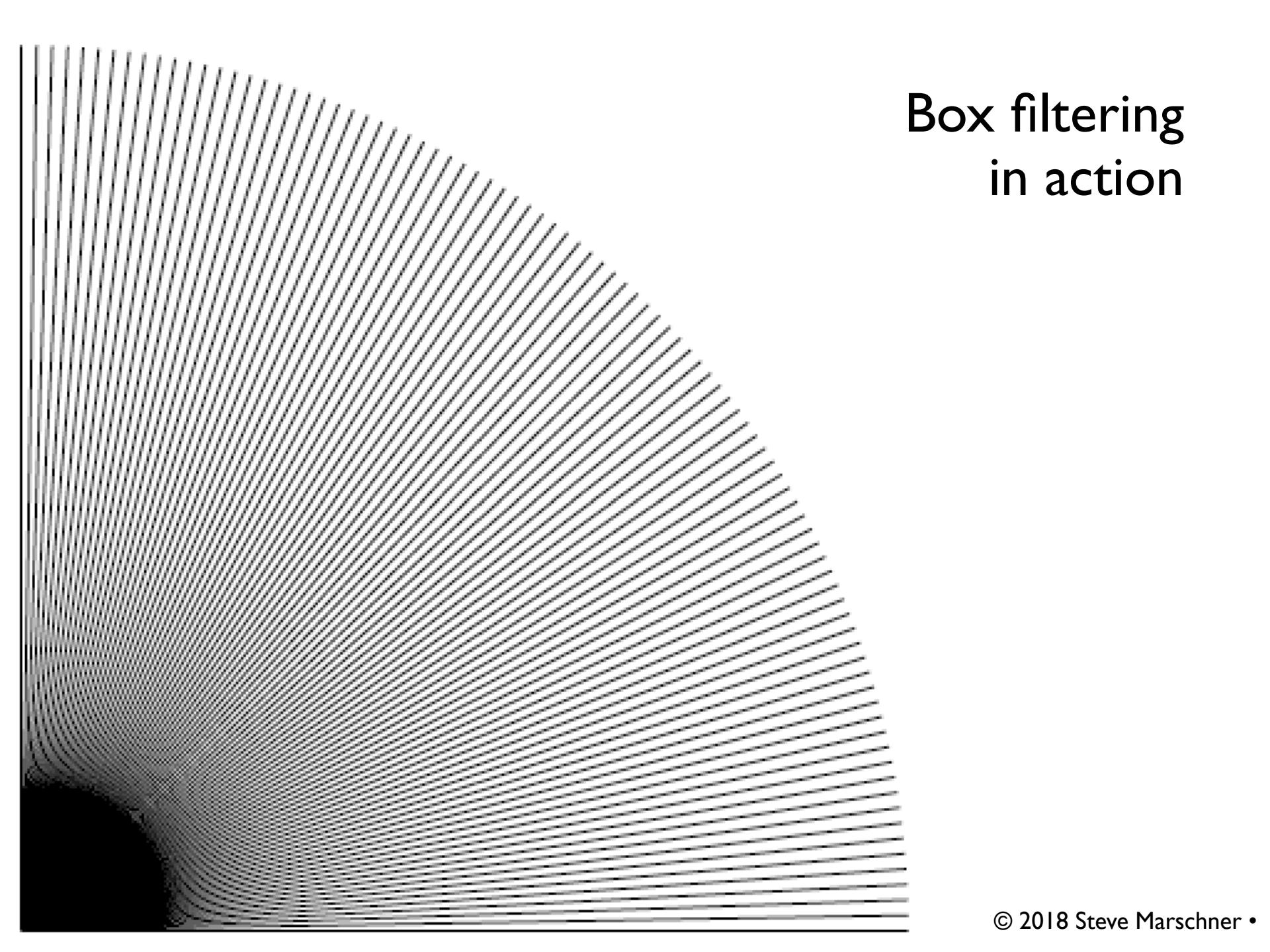
# Box filtering

- Pixel intensity is proportional to area of overlap with square pixel area
- Also called “unweighted area averaging”

# Box filtering by supersampling

- Compute coverage fraction by counting subpixels
- Simple, accurate
- But slow





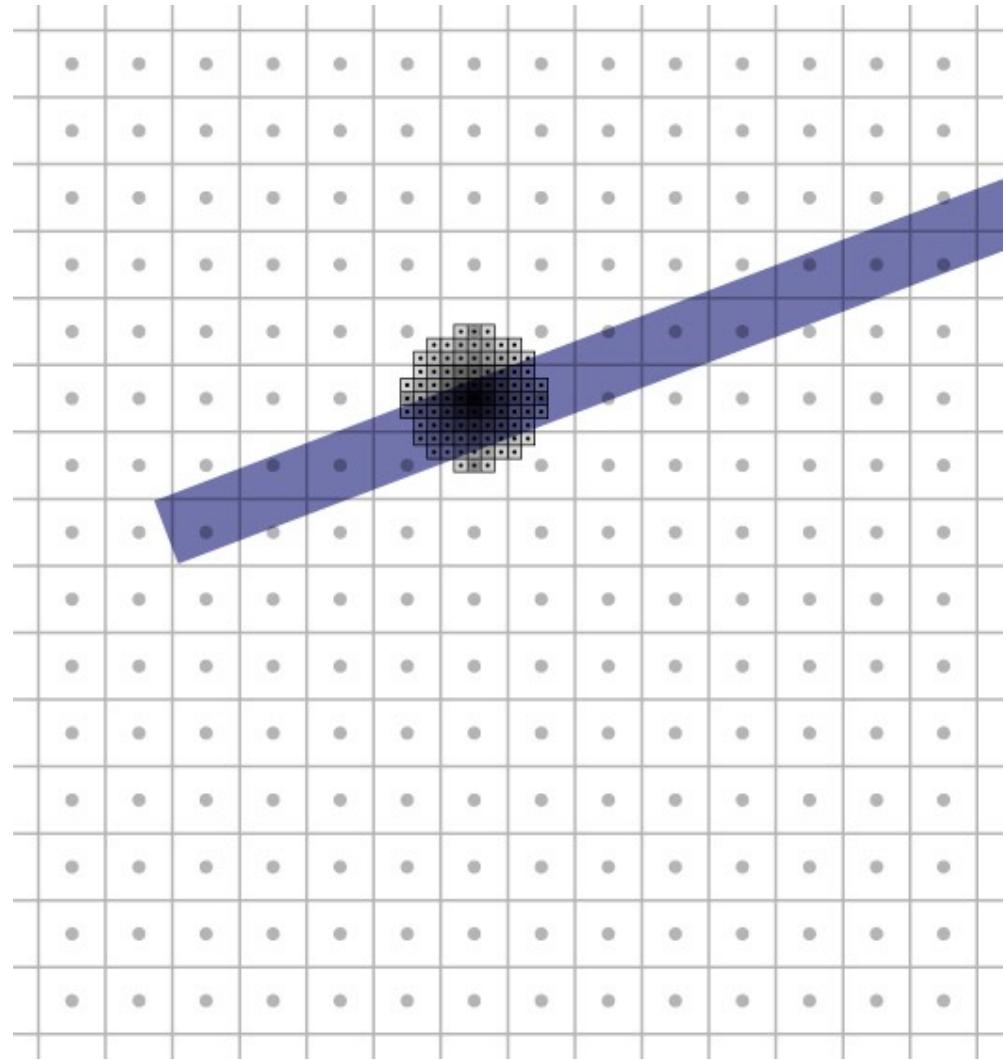
# Box filtering in action

# Weighted filtering

- Box filtering problem: treats area near edge same as area near center
  - results in pixel turning on “too abruptly”
- Alternative: weight area by a smooth function
  - unweighted averaging corresponds to using a box function
  - a gaussian is a popular choice of smooth filter
  - important property: normalization (unit integral)

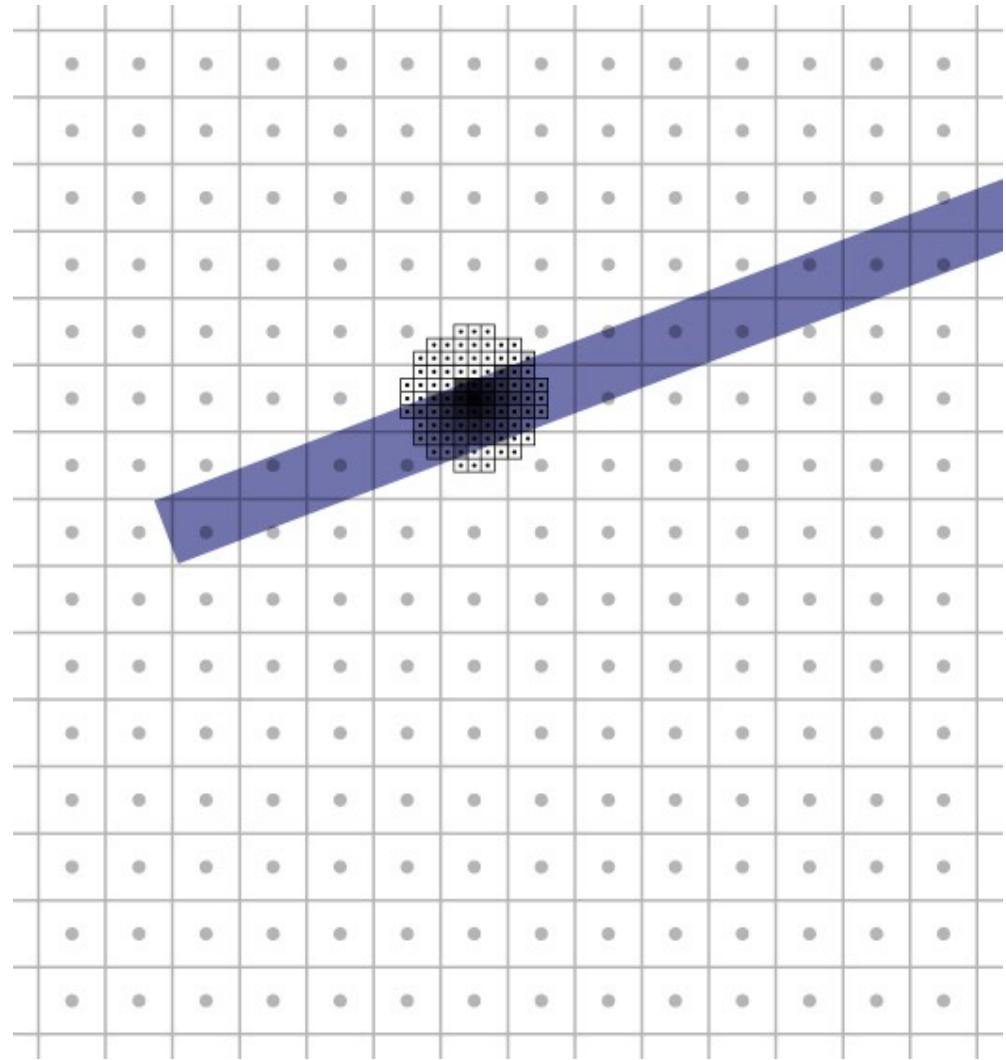
# Weighted filtering by supersampling

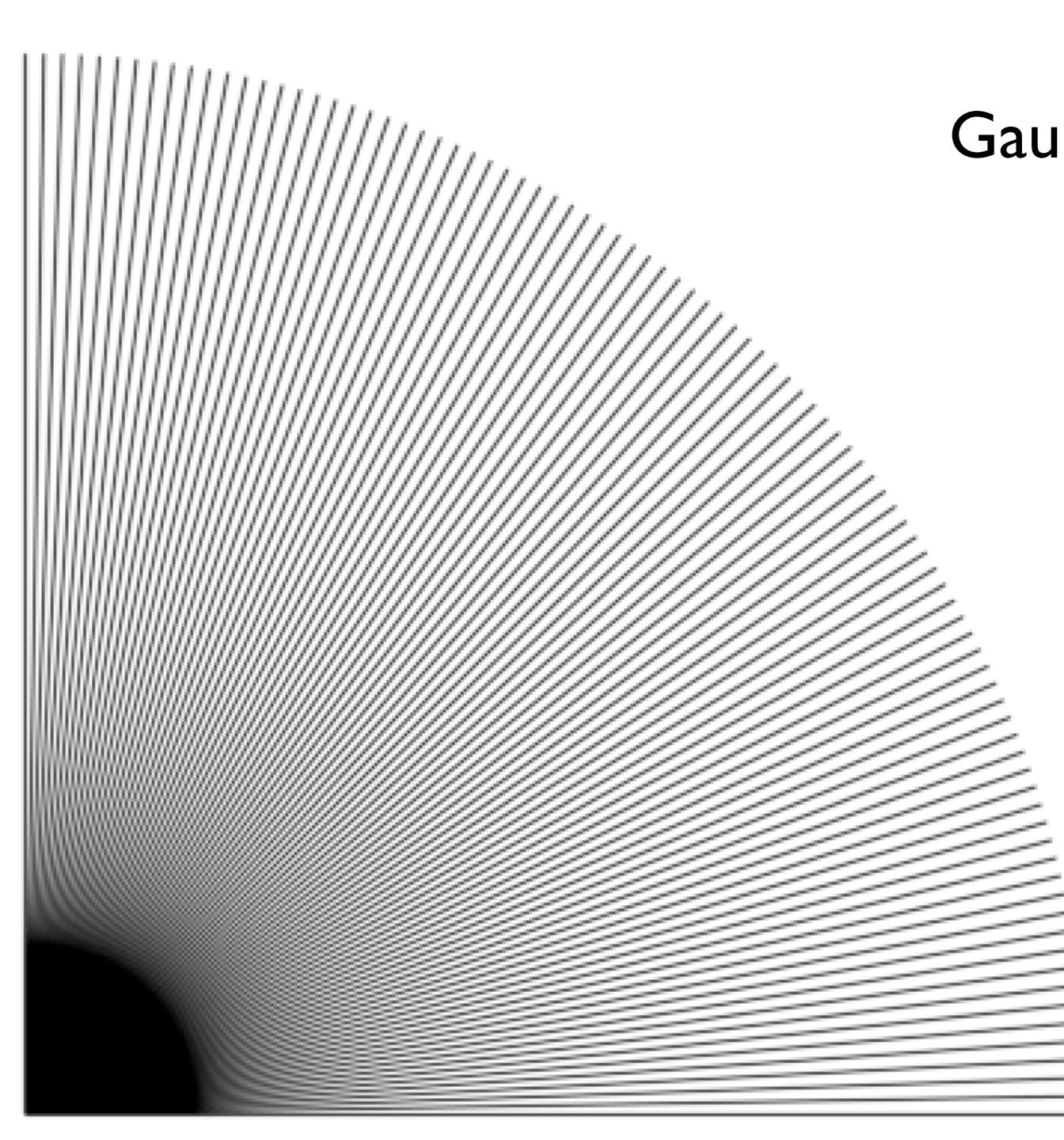
- Compute filtering integral by summing filter values for covered subpixels
- Simple, accurate
- But really slow



# Weighted filtering by supersampling

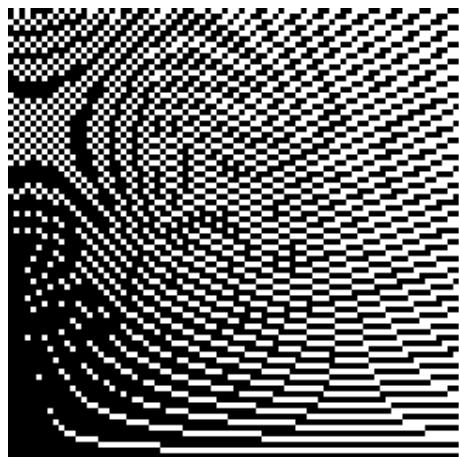
- Compute filtering integral by summing filter values for covered subpixels
- Simple, accurate
- But really slow



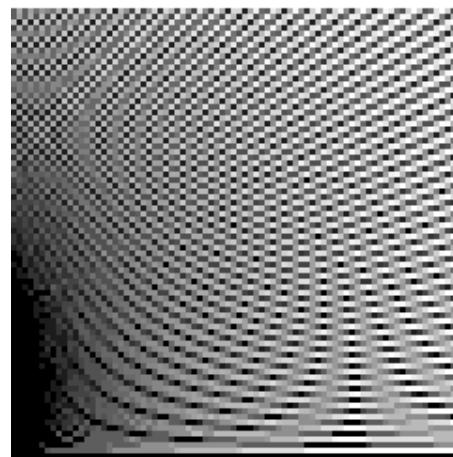


# Gaussian filtering in action

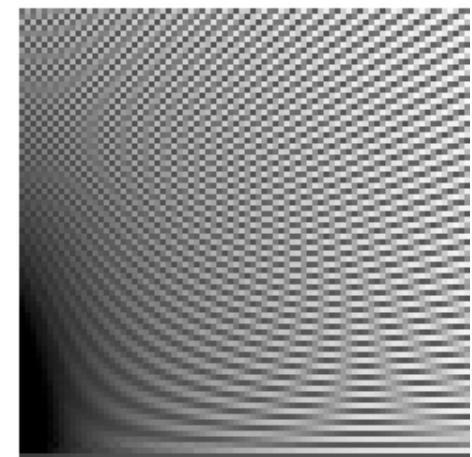
# Filter comparison



Point sampling

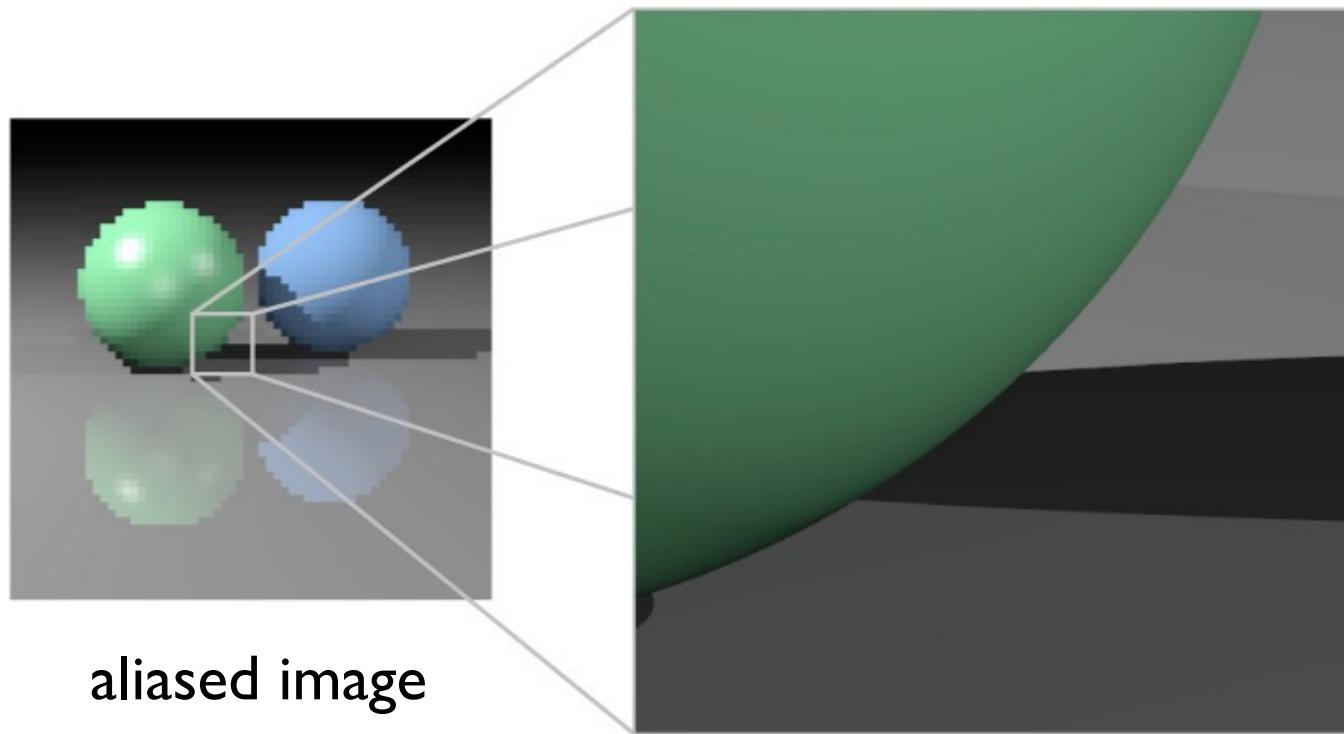


Box filtering

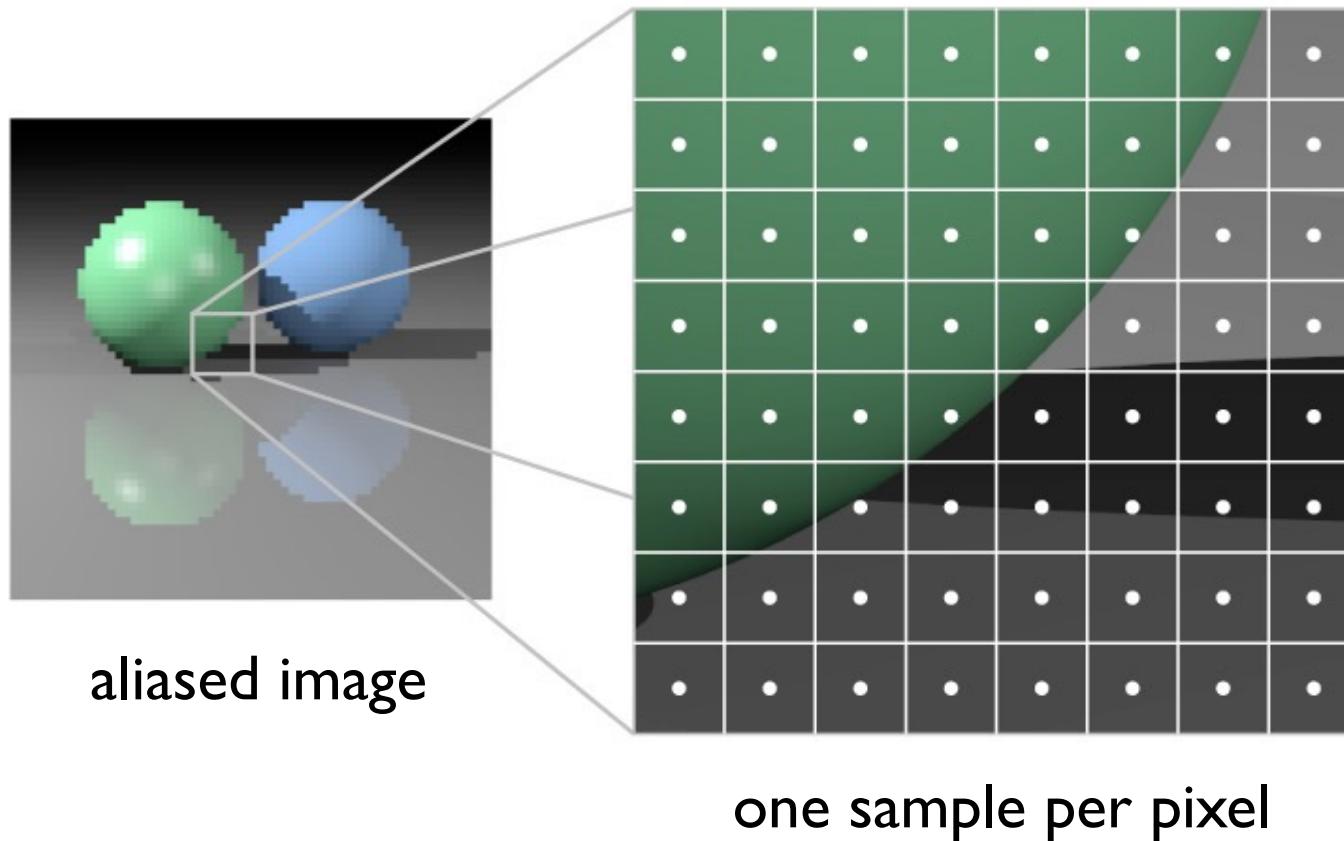


Gaussian filtering

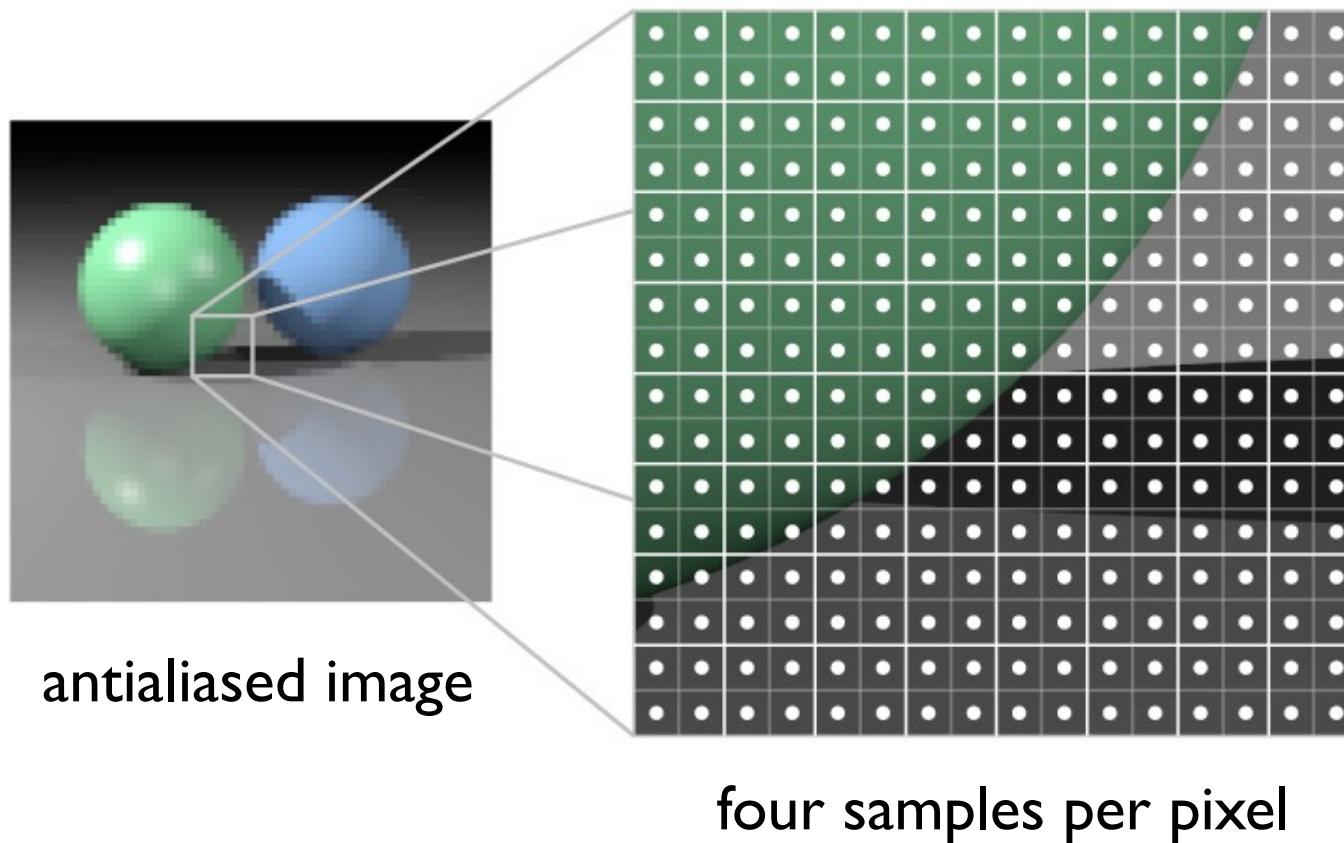
# Antialiasing in ray tracing



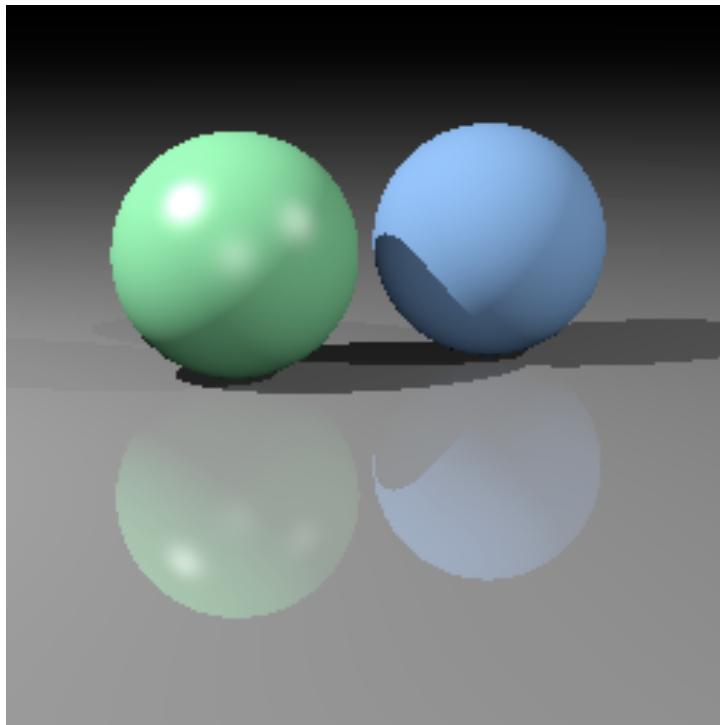
# Antialiasing in ray tracing



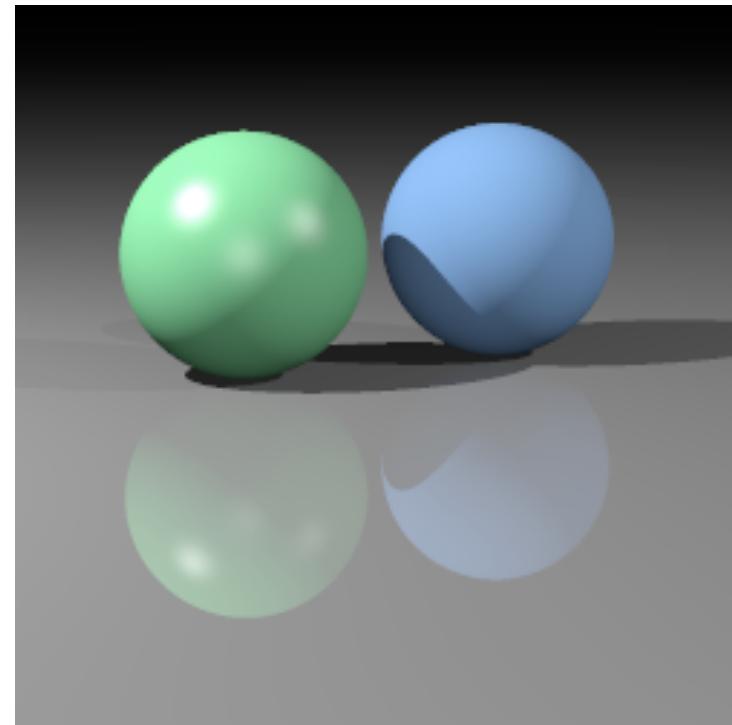
# Antialiasing in ray tracing



# Antialiasing in ray tracing



one sample/pixel

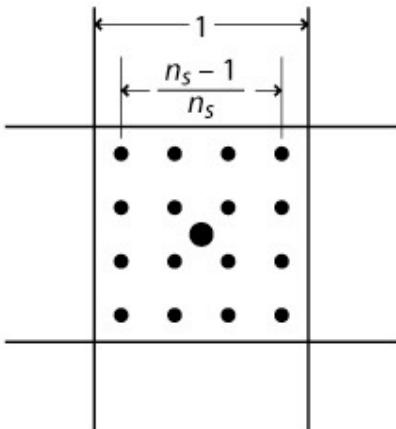


9 samples/pixel

# Details of supersampling

- For image coordinates with integer pixel centers:

```
// one sample per pixel
for iy = 0 to (ny-1) by 1
  for ix = 0 to (nx-1) by 1 {
    ray = camera.getRay(ix, iy);
    image.set(ix, iy, trace(ray));
  }
```



```
// ns^2 samples per pixel
for iy = 0 to (ny-1) by 1
  for ix = 0 to (nx-1) by 1 {
    Color sum = 0;
    for dx = -(ns-1)/2 to (ns-1)/2 by 1
      for dy = -(ns-1)/2 to (ns-1)/2 by 1 {
        x = ix + dx / ns;
        y = iy + dy / ns;
        ray = camera.getRay(x, y);
        sum += trace(ray);
      }
    image.set(ix, iy, sum / (ns*ns));
  }
```

# Details of supersampling

- For image coordinates in unit square

```
// one sample per pixel
for iy = 0 to (ny-1) by 1
    for ix = 0 to (nx-1) by 1 {
        double x = (ix + 0.5) / nx;
        double y = (iy + 0.5) / ny;
        ray = camera.getRay(x, y);
        image.set(ix, iy, trace(ray));
    }
```

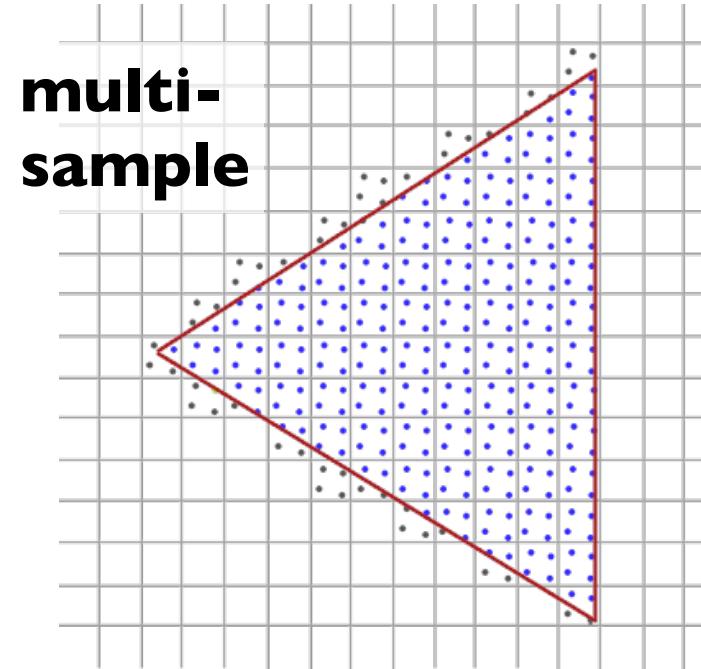
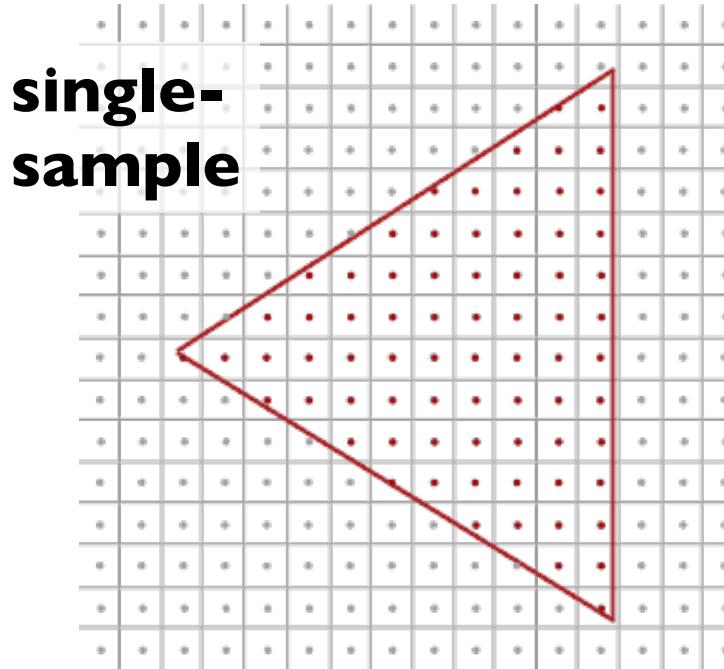
```
// ns^2 samples per pixel
for iy = 0 to (ny-1) by 1
    for ix = 0 to (nx-1) by 1 {
        Color sum = 0;
        for dx = 0 to (ns-1) by 1
            for dy = 0 to (ns-1) by 1 {
                x = (ix + (dx + 0.5) / ns) / nx;
                y = (iy + (dy + 0.5) / ns) / ny;
                ray = camera.getRay(x, y);
                sum += trace(ray);
            }
        image.set(ix, iy, sum / (ns*ns));
    }
```

# Supersampling vs. multisampling

- Supersampling is terribly expensive
- GPUs use an approximation called *multisampling*
  - Compute one shading value per pixel
  - Store it at many subpixel samples, each with its own depth

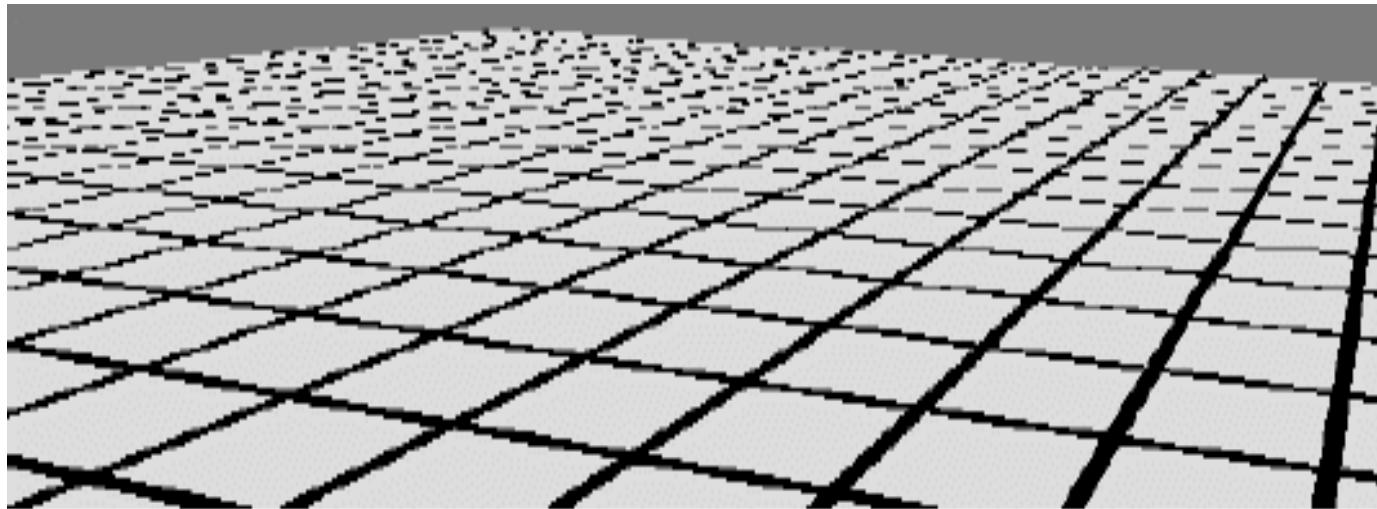
# Multisample rasterization

- Each fragment carries several (color,depth) samples
  - shading is computed per-fragment
  - depth test is resolved per-sample
  - final color is average of sample colors



# Antialiasing in textures

- Even with multisampling, we still only evaluate textures once per fragment
- Need to filter the texture somehow!
  - perspective produces very fine subpixel detail



[Akenine-Möller & Haines 2002]

# Texture mapping from 0 to infinity

- When you go close...

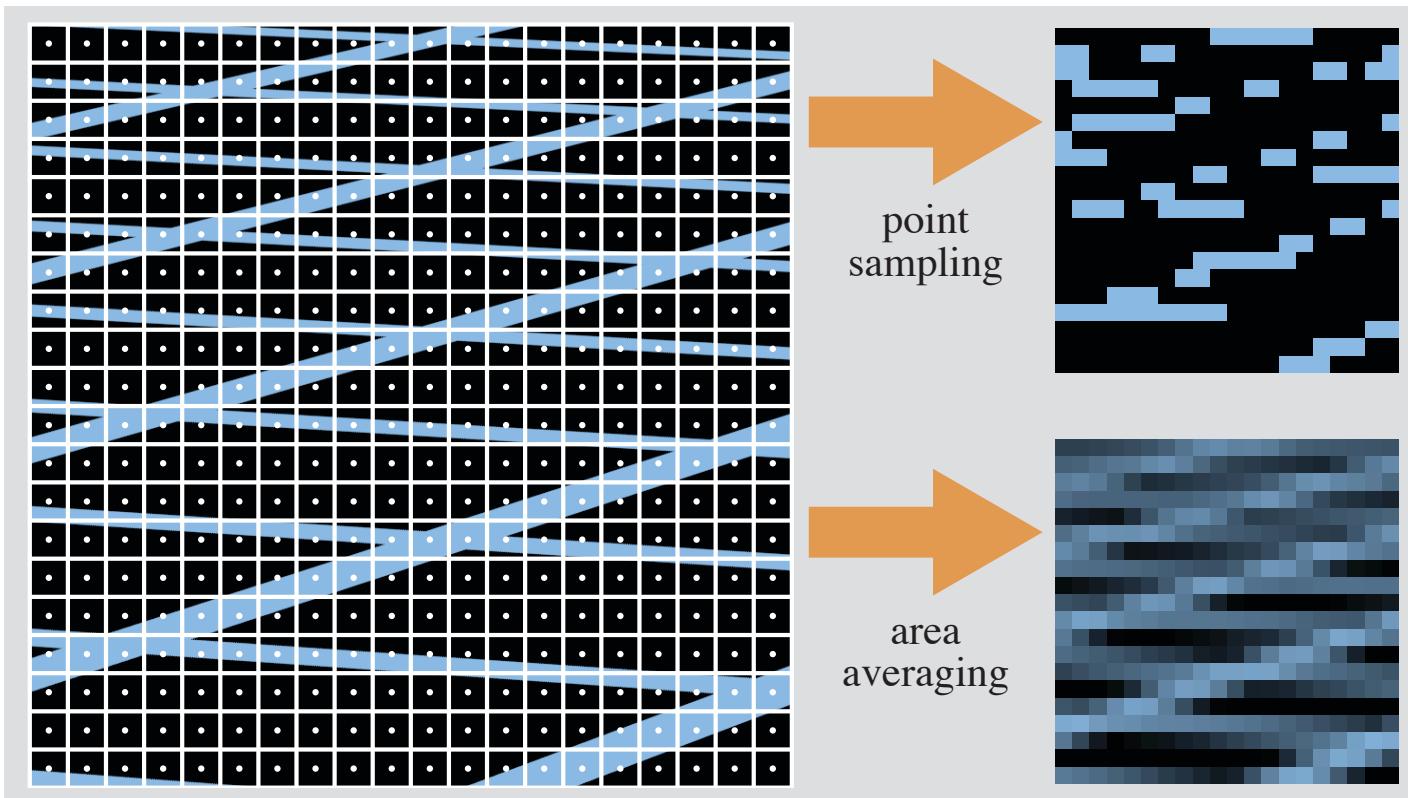


# Texture mapping from 0 to infinity

- When you go far...



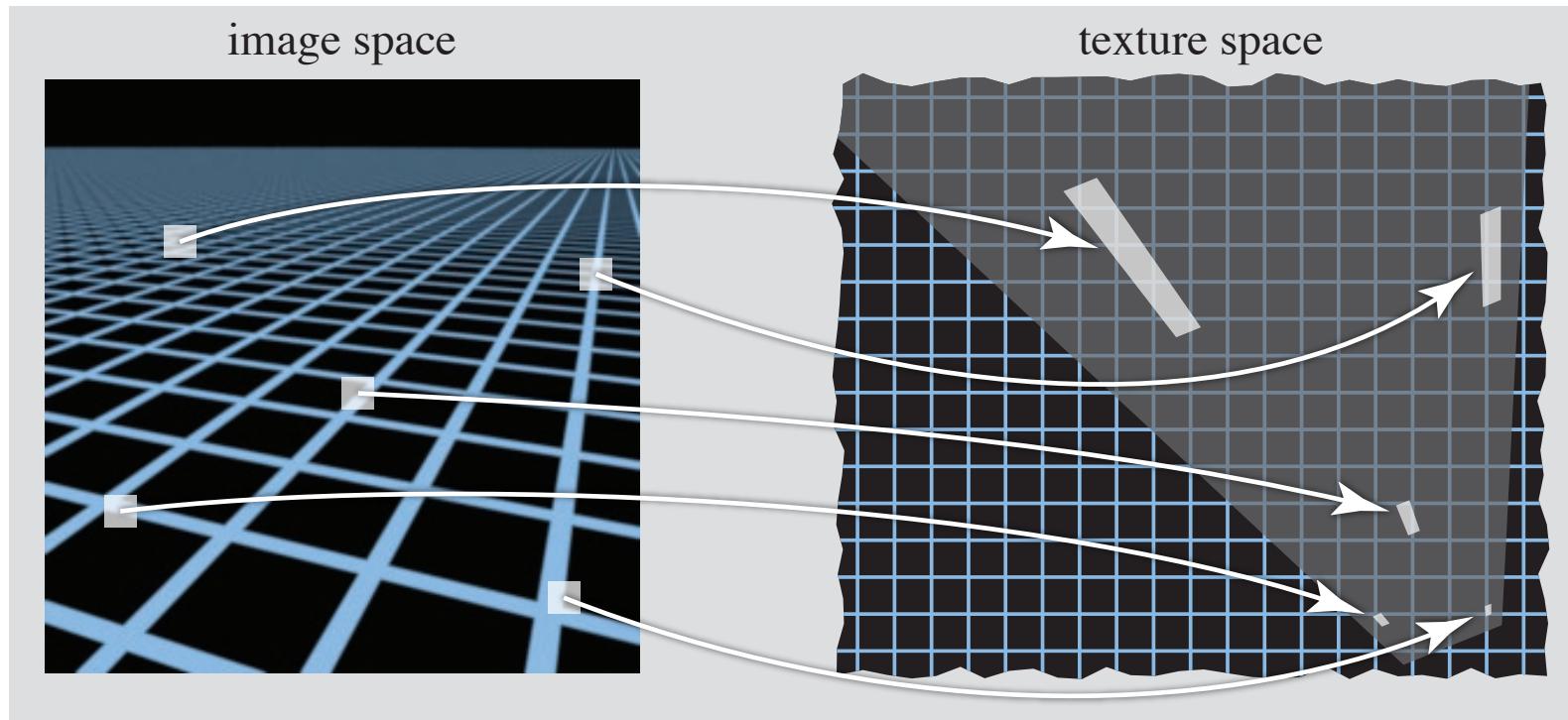
# Solution: pixel filtering



# Pixel filtering in texture space

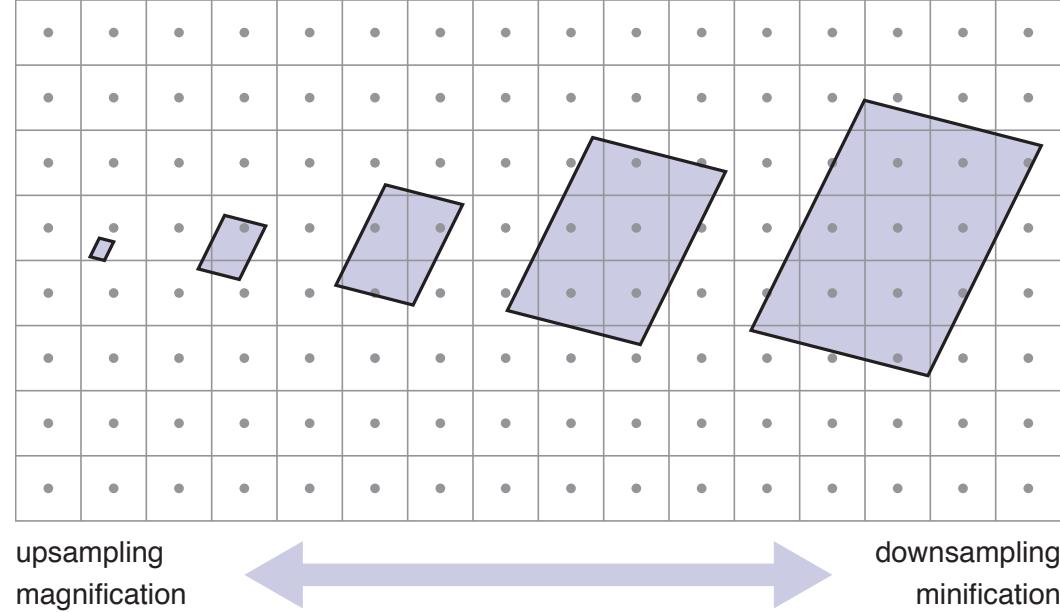
- Sampling is happening in image space
  - sample is a weighted average over a pixel-sized area
  - uniform, predictable, friendly problem!
- Signal is defined in texture space
  - mapping between image and texture is nonuniform
  - each sample is a weighted average over a different sized and shaped area
  - irregular, unpredictable, unfriendly!
- This is a change of variable
  - integrate over texture coordinates rather than image coordinates

# Pixel footprints

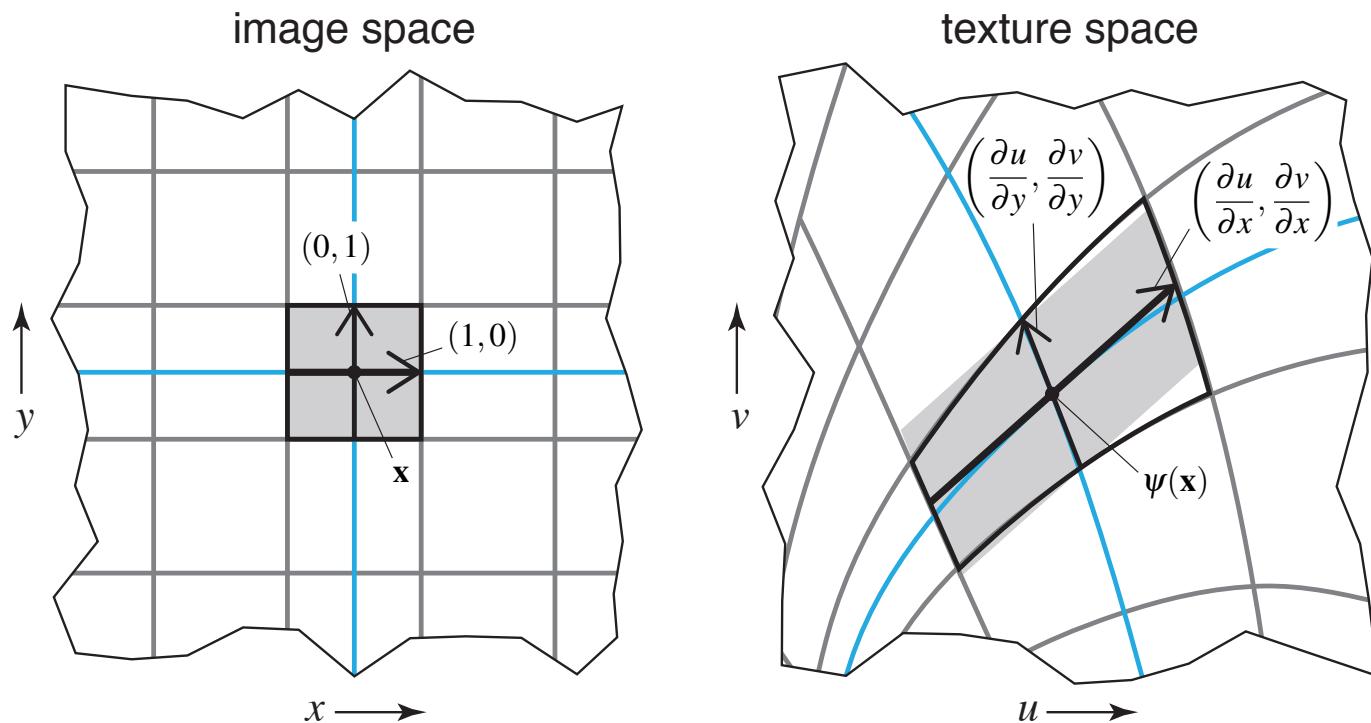


# How does area map over distance?

- At optimal viewing distance:
  - One-to-one mapping between pixel area and texel area
- When closer
  - Each pixel is a small part of the texel
  - magnification
- When farther
  - Each pixel could include many texels
  - “minification”

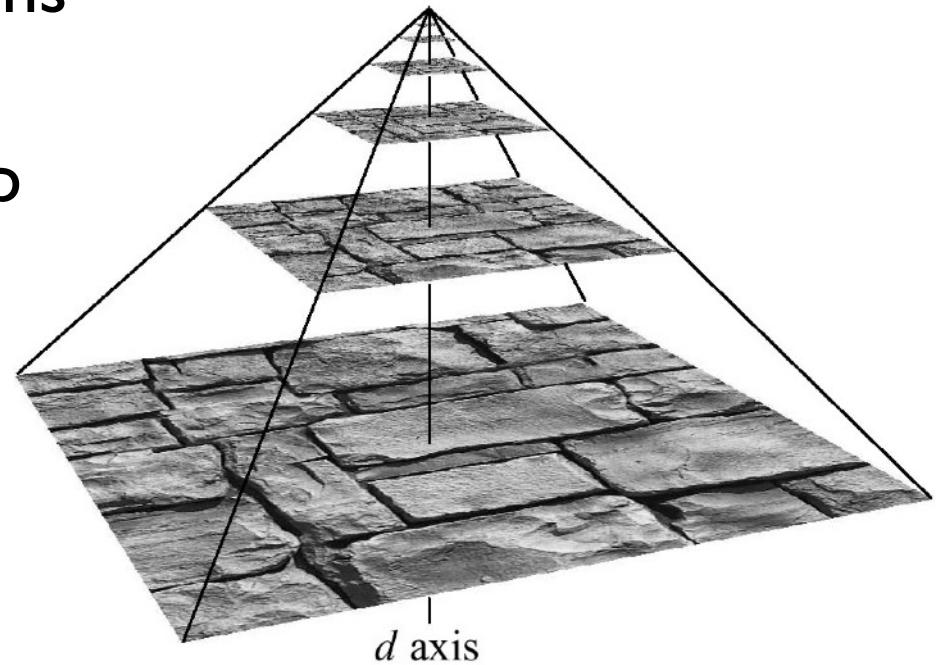


# Sizing up the situation with the Jacobian



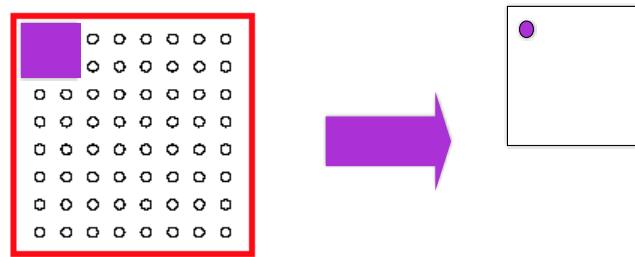
# Mipmap image pyramid

- MIP Maps
  - Multum in Parvo: Much in little, many in small places
  - Proposed by Lance Williams
- Stores pre-filtered versions of texture
- Supports very fast lookup
  - but only of circular filters at certain scales



[Akenine-Möller & Haines 2002]

# Filtering by Averaging



- Each pixel in a level corresponds to 4 pixels in lower level
  - Average
  - Gaussian filtering

# Using the MIP Map

- Find the MIP Map level where the pixel has a 1-to-1 mapping
- How?
  - Find largest side of pixel footprint in texture space
    - Pick level where that side corresponds to a texel
  - Compute derivatives to find pixel footprint

- x derivative:

$$\frac{\partial u}{\partial x} \quad \frac{\partial v}{\partial x}$$

- y derivative:

$$\frac{\partial u}{\partial y} \quad \frac{\partial v}{\partial y}$$

# Given derivatives: what is level?

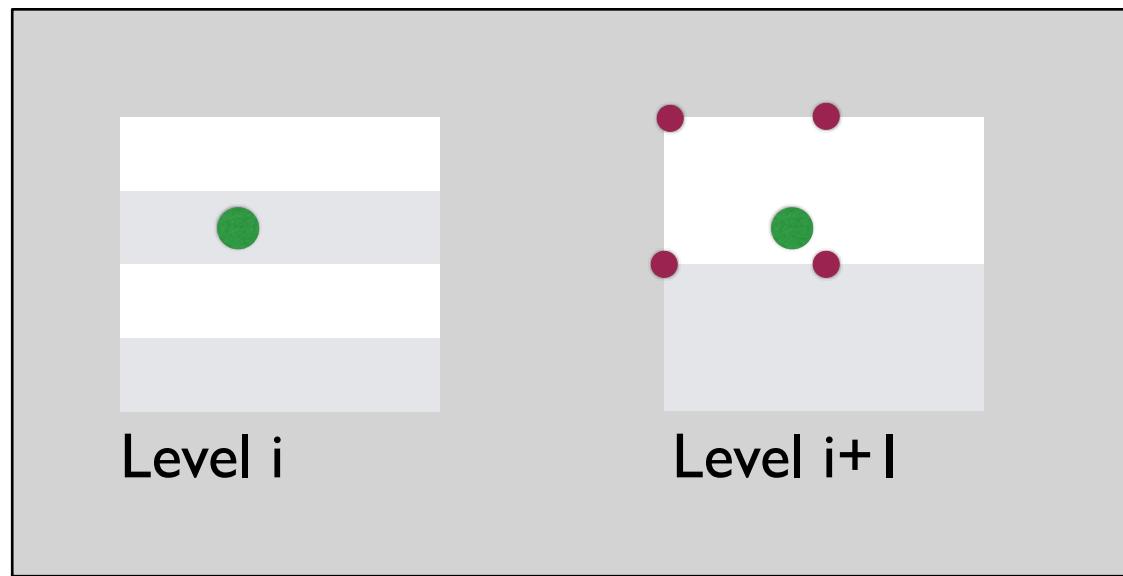
- Derivatives
  - Available in pixel shader (except where there is dynamic branching)

$$level = \log[\max\left(\frac{du}{dx}, \frac{dv}{dx}, \frac{du}{dy}, \frac{dv}{dy}\right)]$$

$$level = \log \sqrt{\left(\frac{du}{dx}\right)^2 + \left(\frac{dv}{dx}\right)^2 + \left(\frac{du}{dy}\right)^2 + \left(\frac{dv}{dy}\right)^2}$$

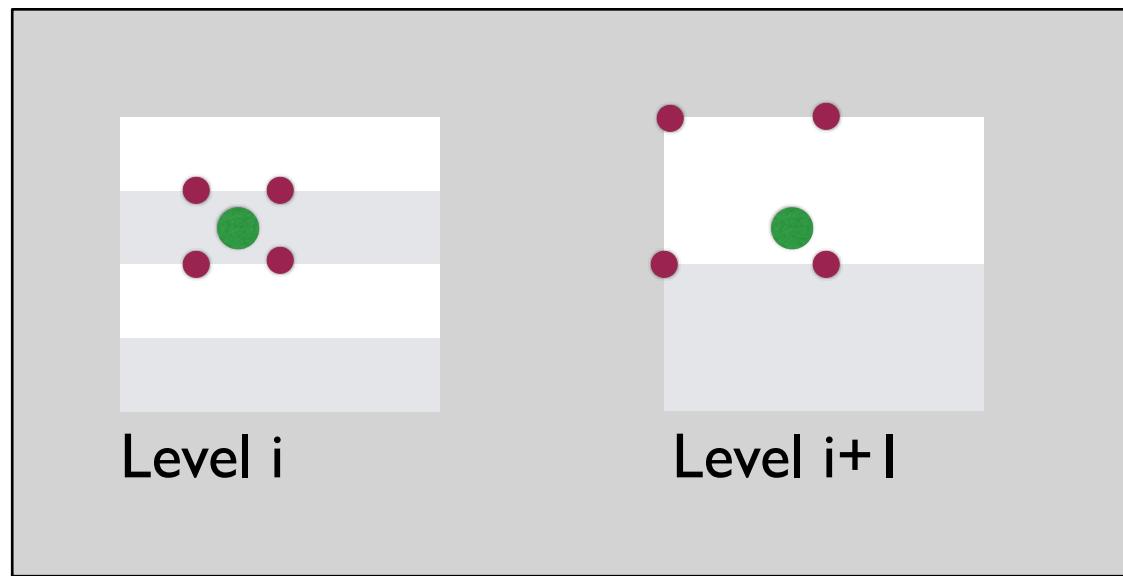
# Using the MIP Map

- In level, find texel and
  - Return the texture value: point sampling (but still better)!
  - Bilinear interpolation
  - Trilinear interpolation



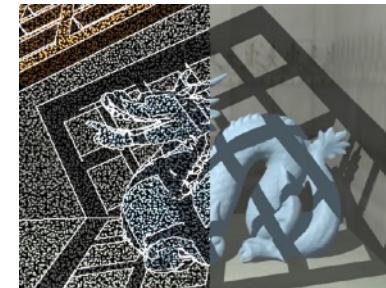
# Using the MIP Map

- In level, find texel and
  - Return the texture value: point sampling (but still better)!
  - Bilinear interpolation
  - Trilinear interpolation



# Memory Usage

- What happens to size of texture?

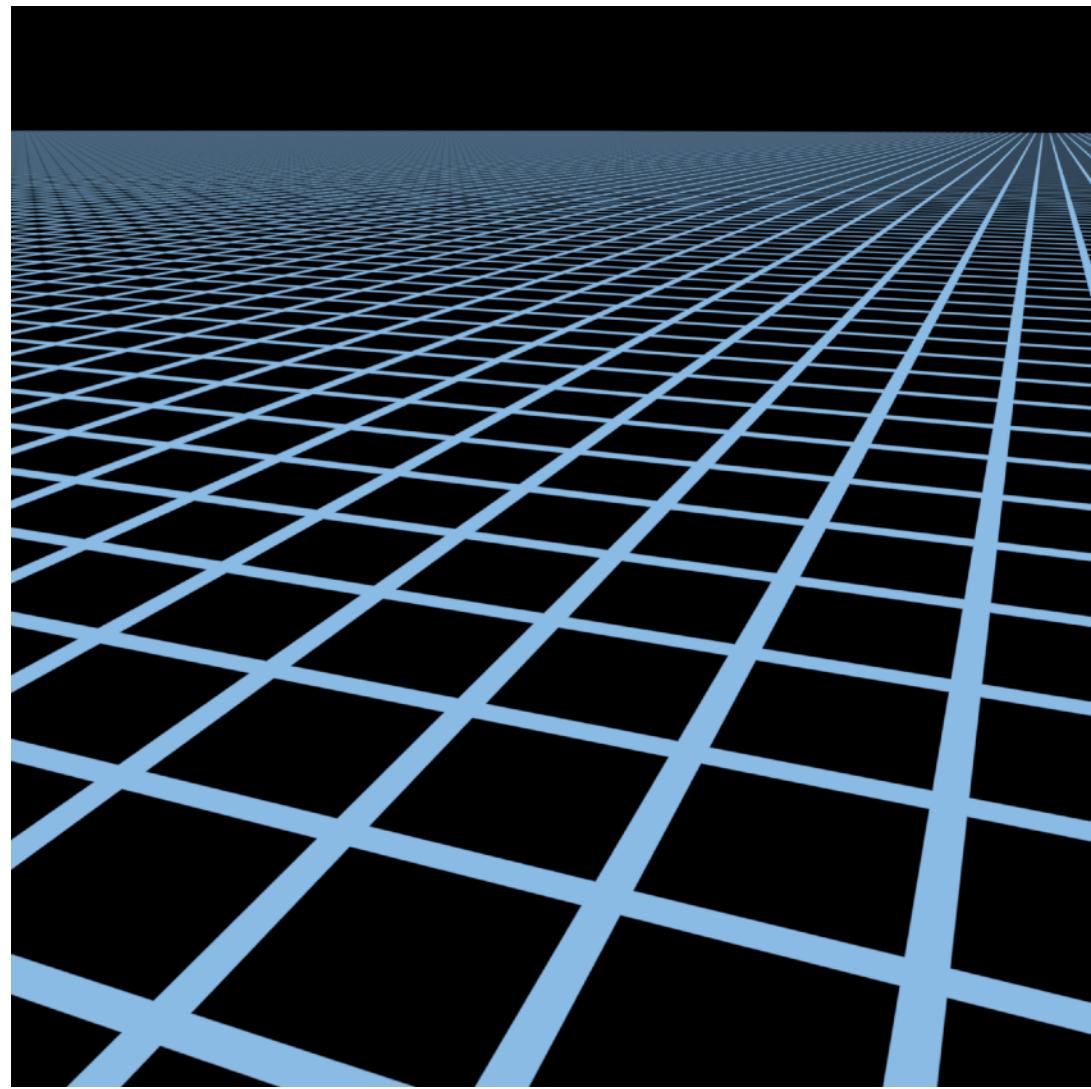


[Kavita Bala]

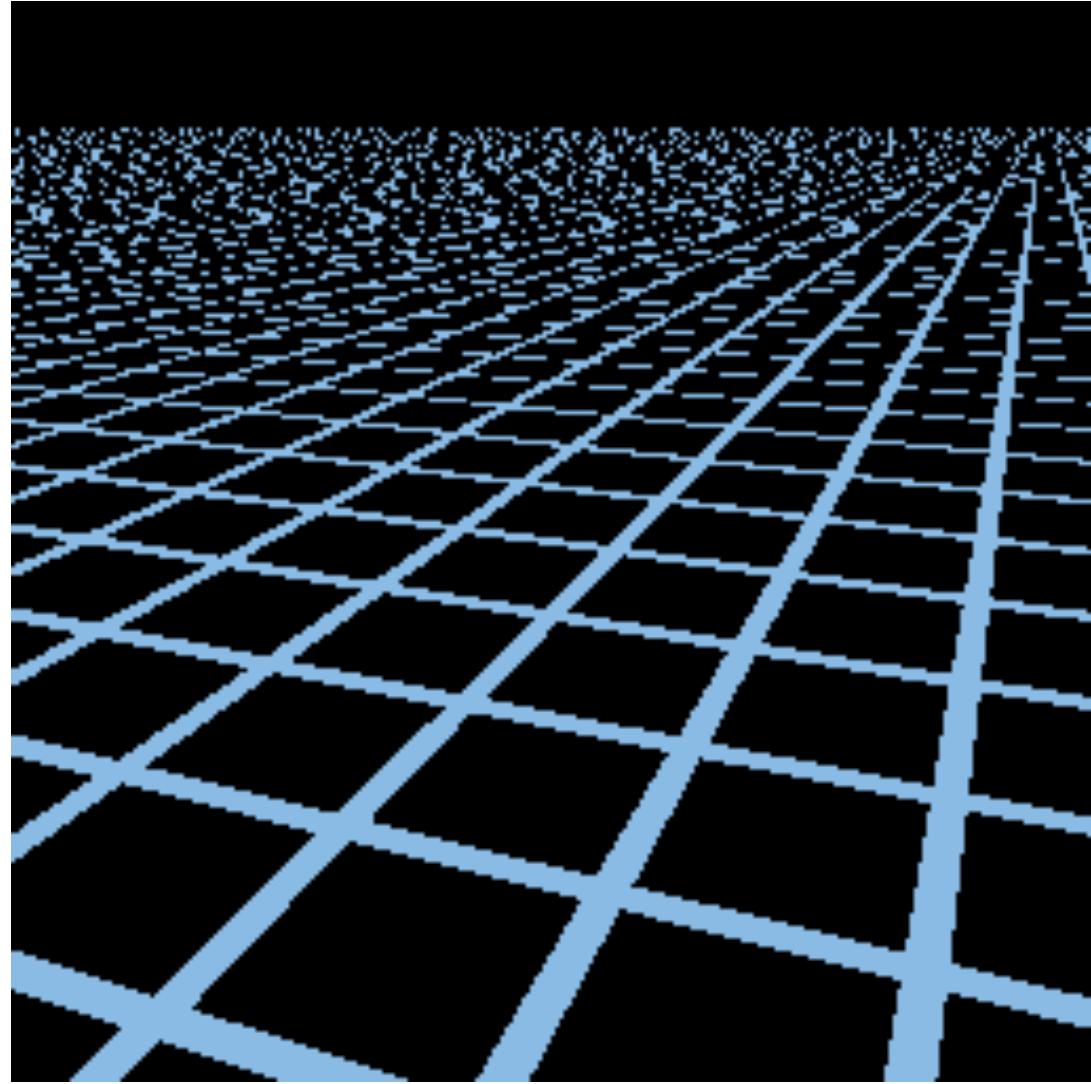
# MIPMAP

- Multi-resolution image pyramid
  - Pre-sampled computation of MIPMAP
  - 1/3 more memory
- Bilinear or Trilinear interpolation

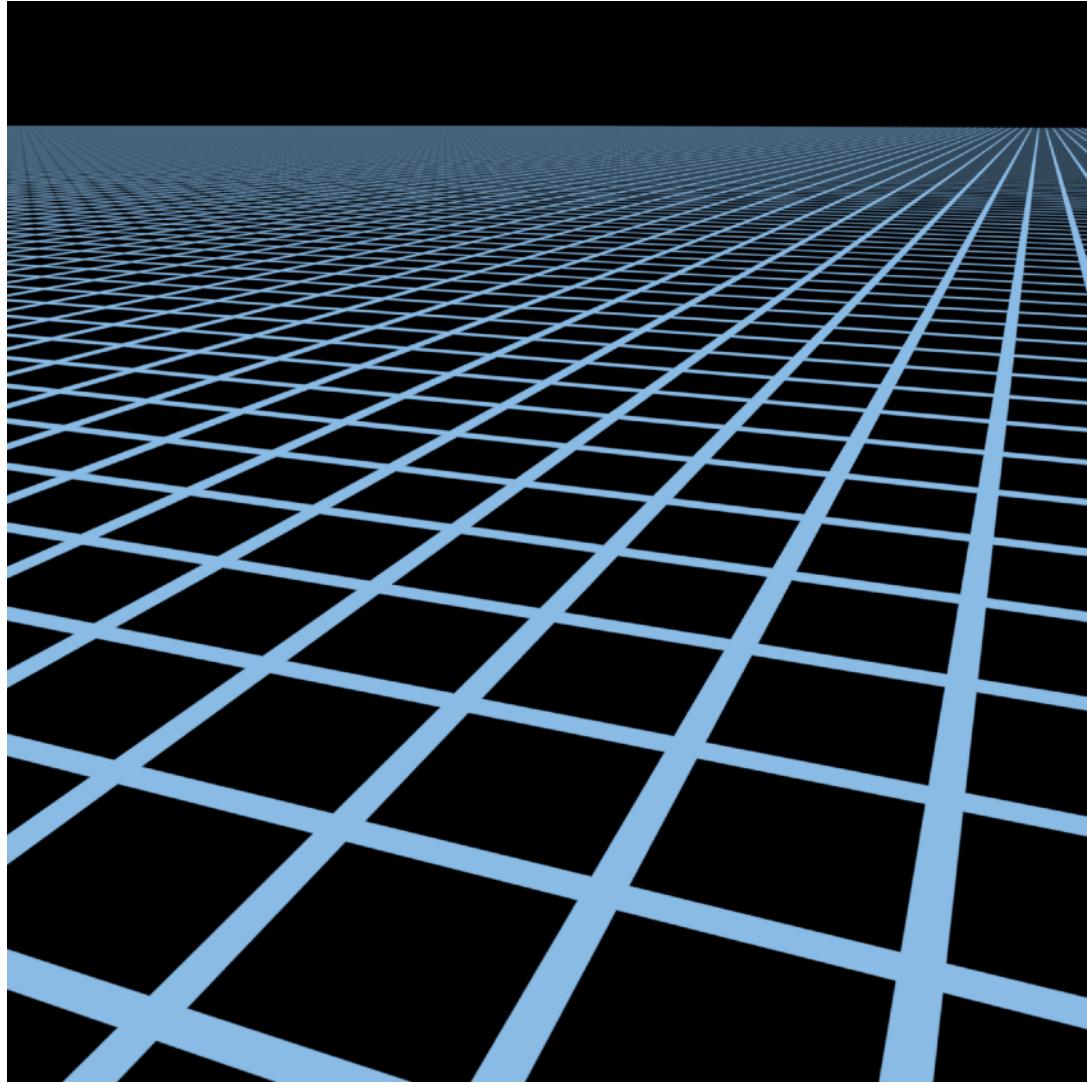
# Point sampling



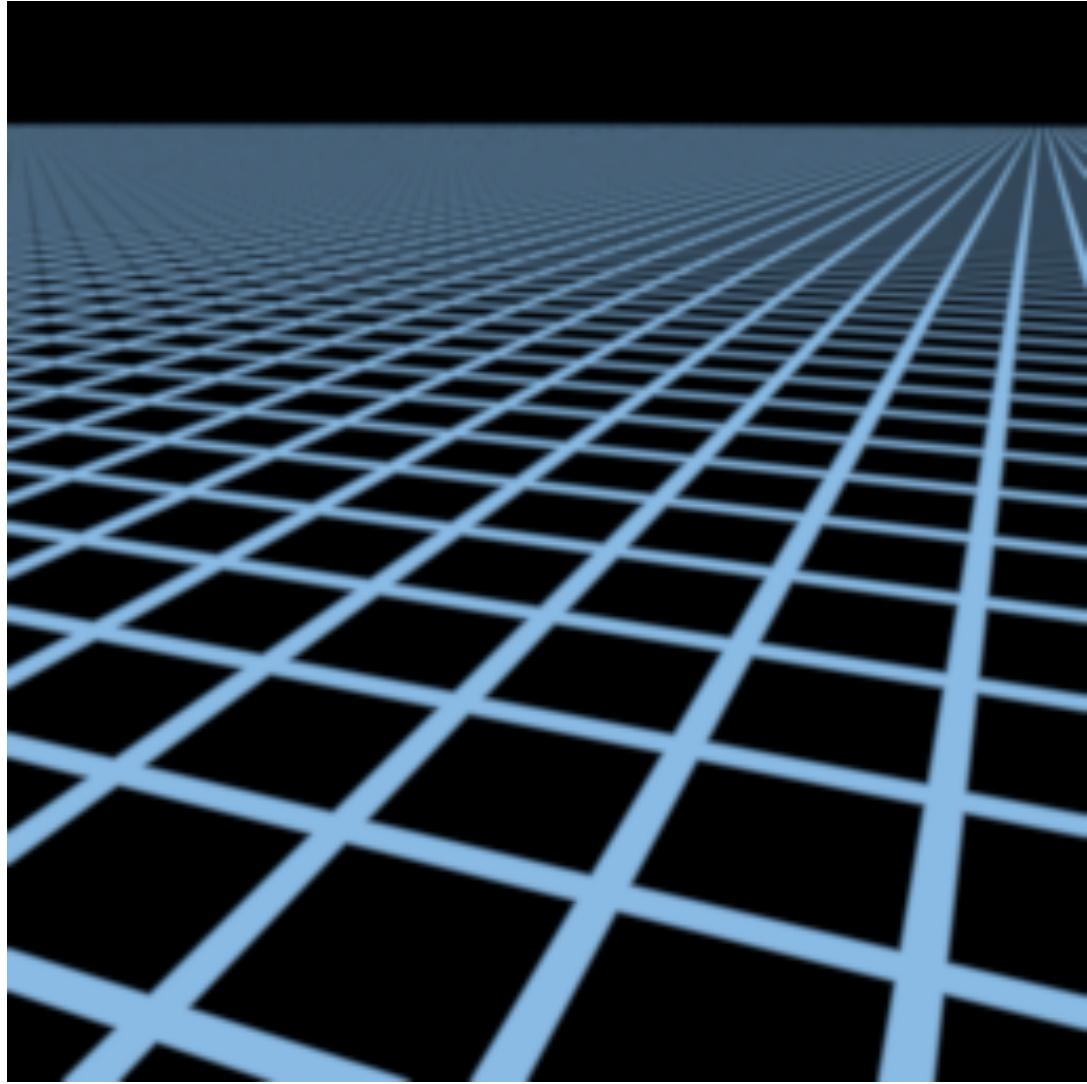
# Point sampling



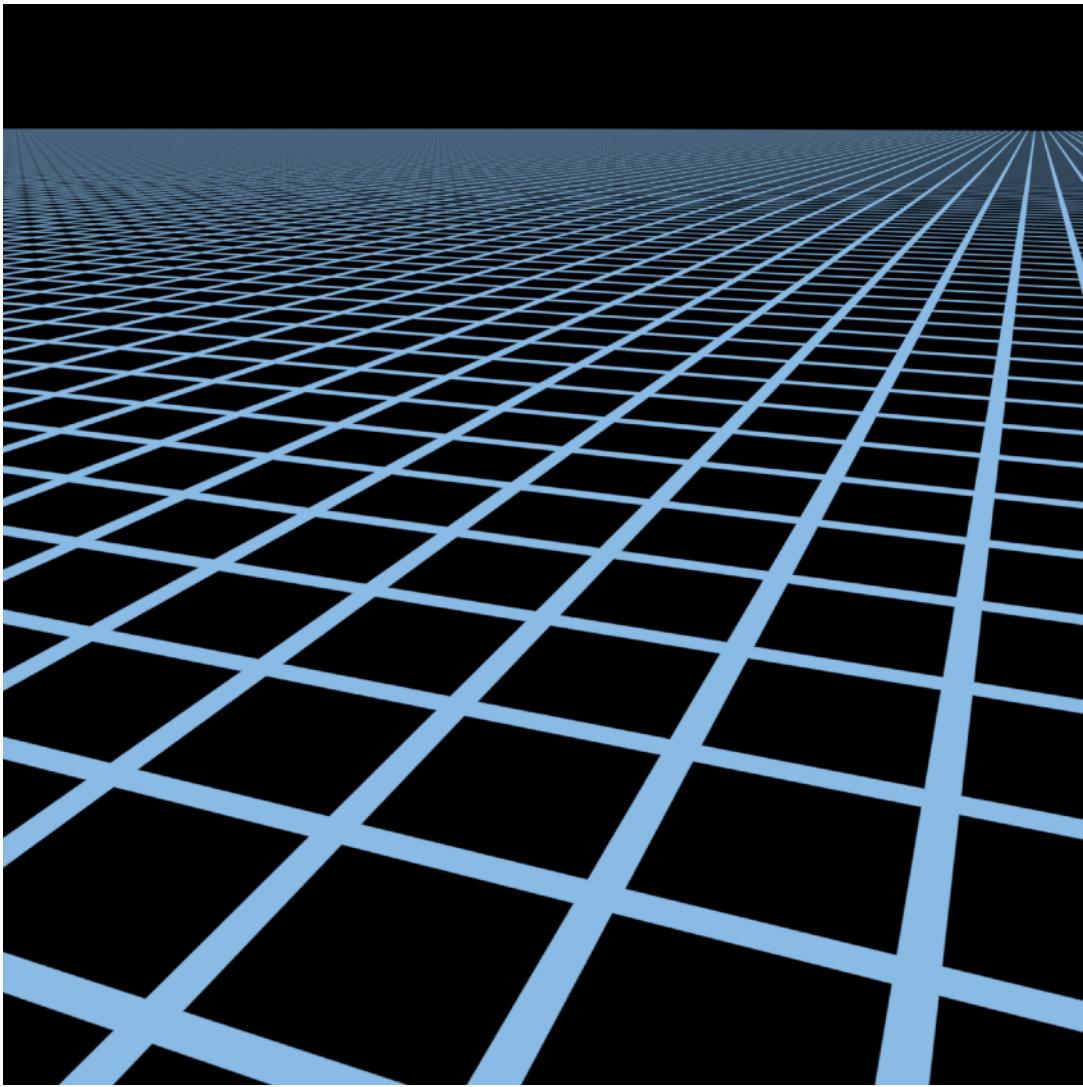
# Reference: gaussian sampling by



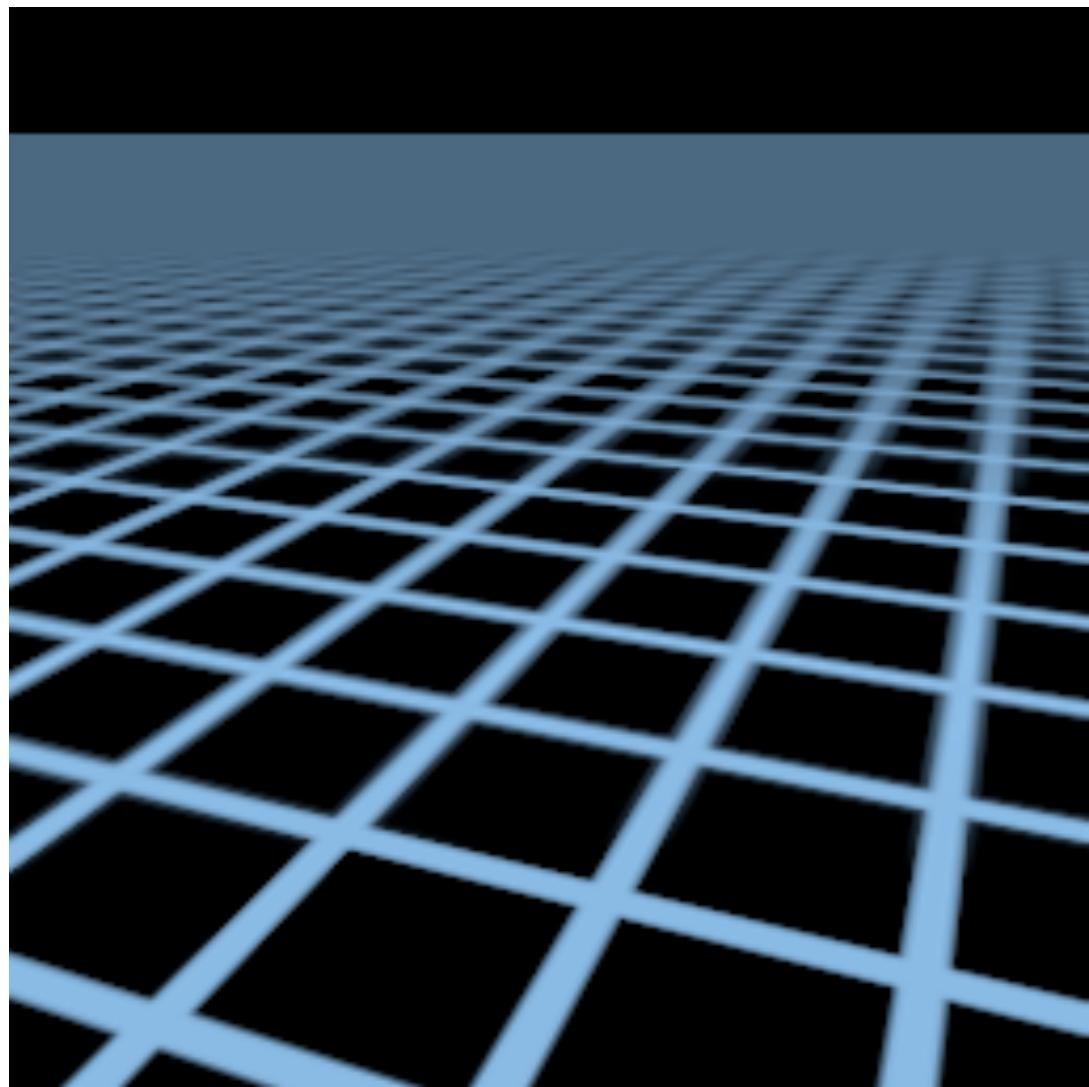
# Reference: gaussian sampling by



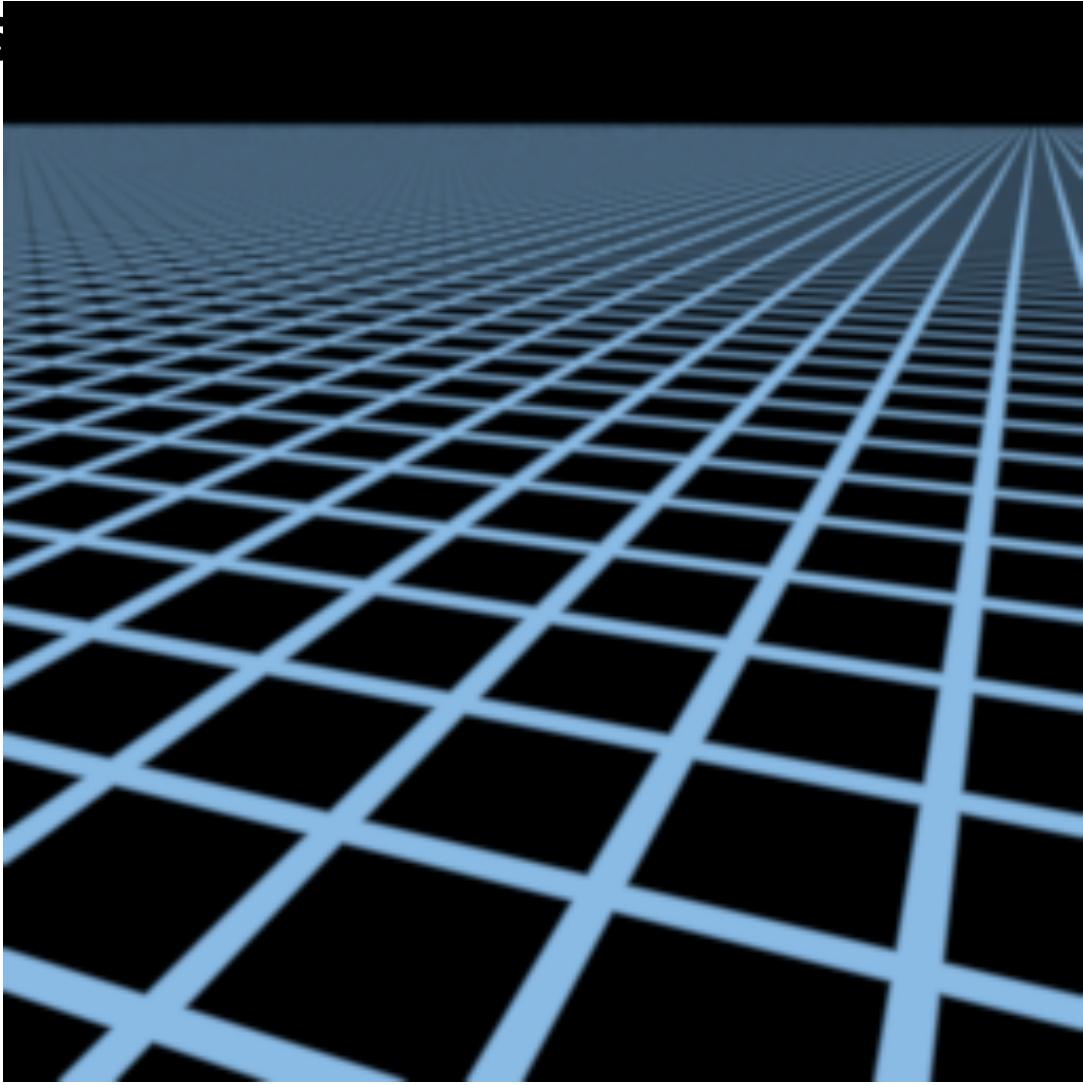
# Texture minification with a mipmap



# Texture minification with a mipmap



# Texture minification: supersampling vs.



# Texture minification: supersampling vs.

