Lecture 4:

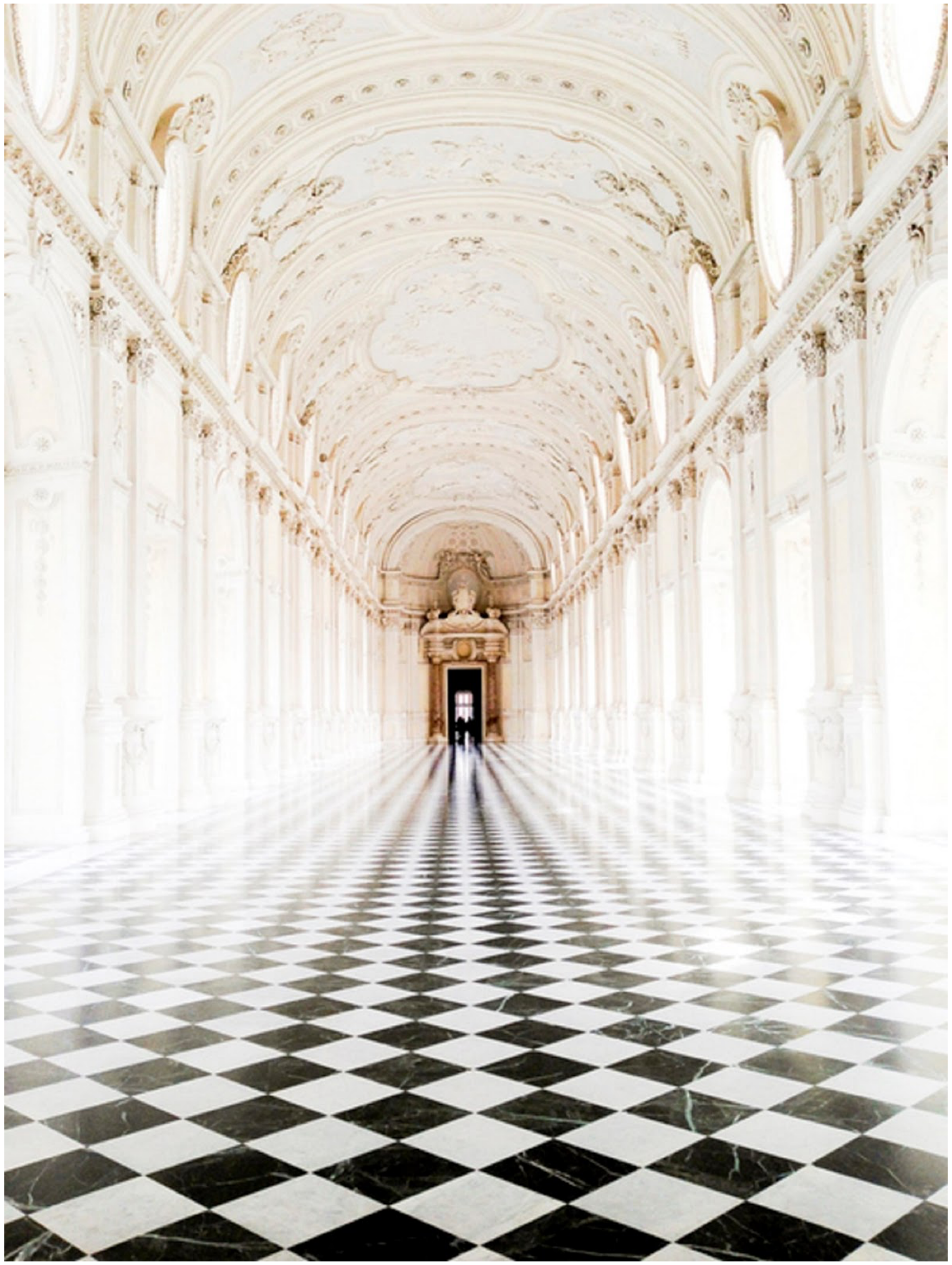
Perspective Projection and Texture Mapping

Interactive Computer Graphics

Stanford CS248, Winter 2019

Perspective and texture

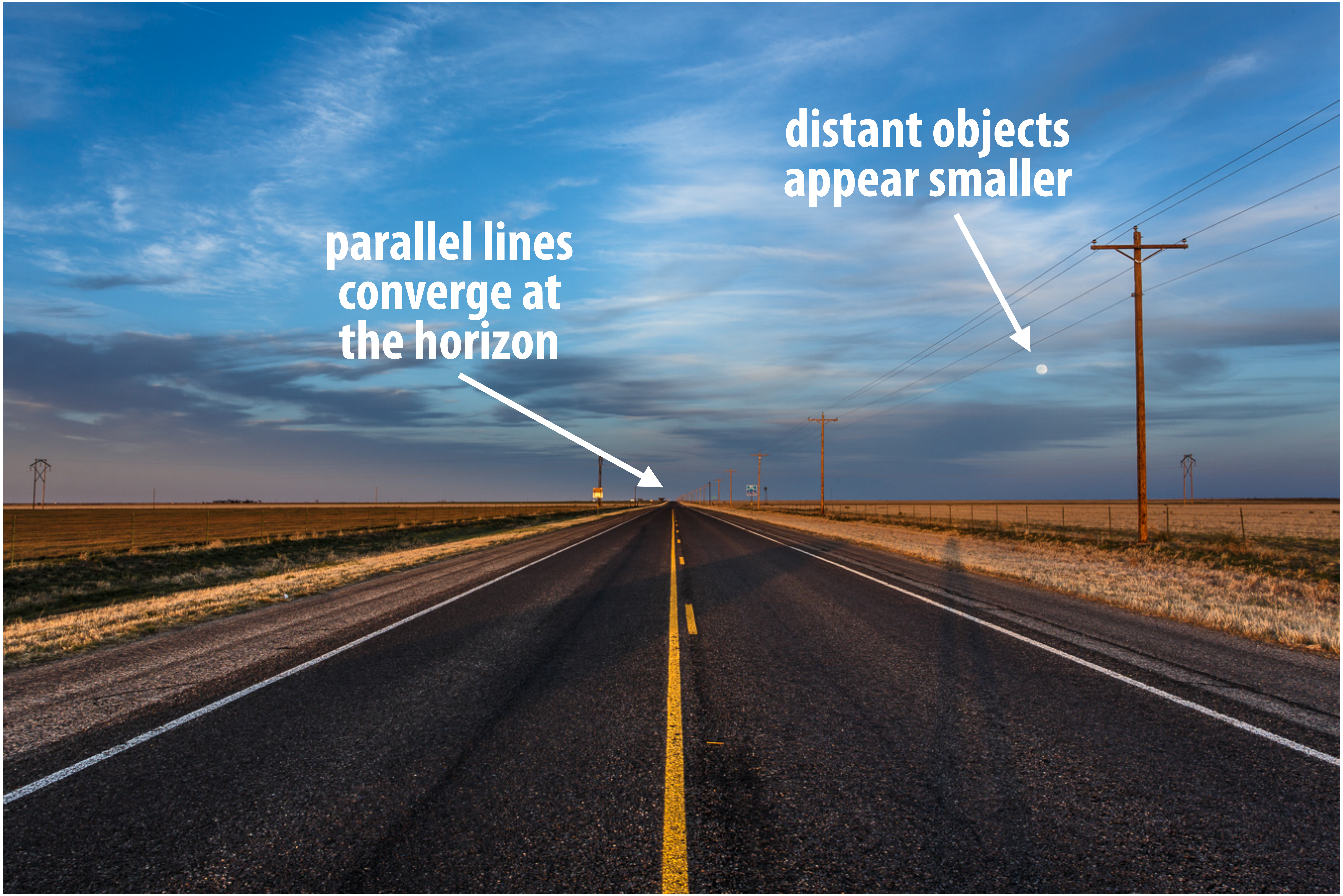
- **PREVIOUSLY:**
  - *transformation* (how to manipulate primitives in space)
  - *rasterization* (how to turn primitives into pixels)
- **TODAY:**
  - see where these two ideas come crashing together!
  - revisit *perspective* transformations
  - talk about how to map *texture* onto a primitive to get more detail
  - …and how perspective creates challenges for texture mapping!



Why is it hard to render an image like this?

Perspective Projection

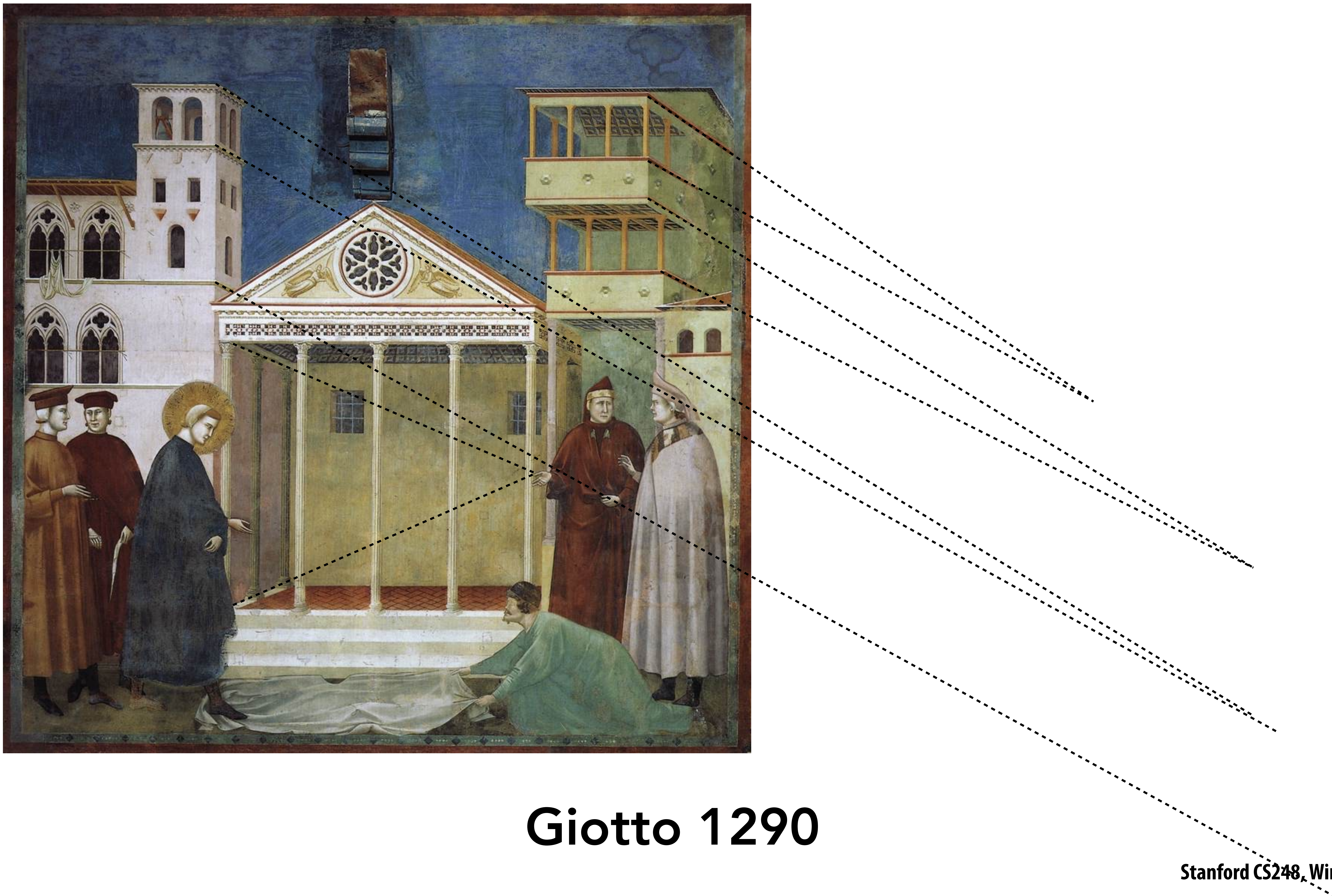
Perspective projection



Early painting: incorrect perspective

Carolingian painting from the 8-9th century

Perspective in art



Giotto 1290

Evolution toward correct perspective

Ambrogio Lorenzetti
Annunciation, 1344

Brunelleschi, elevation of Santo Spirito, 1434-83, Florence

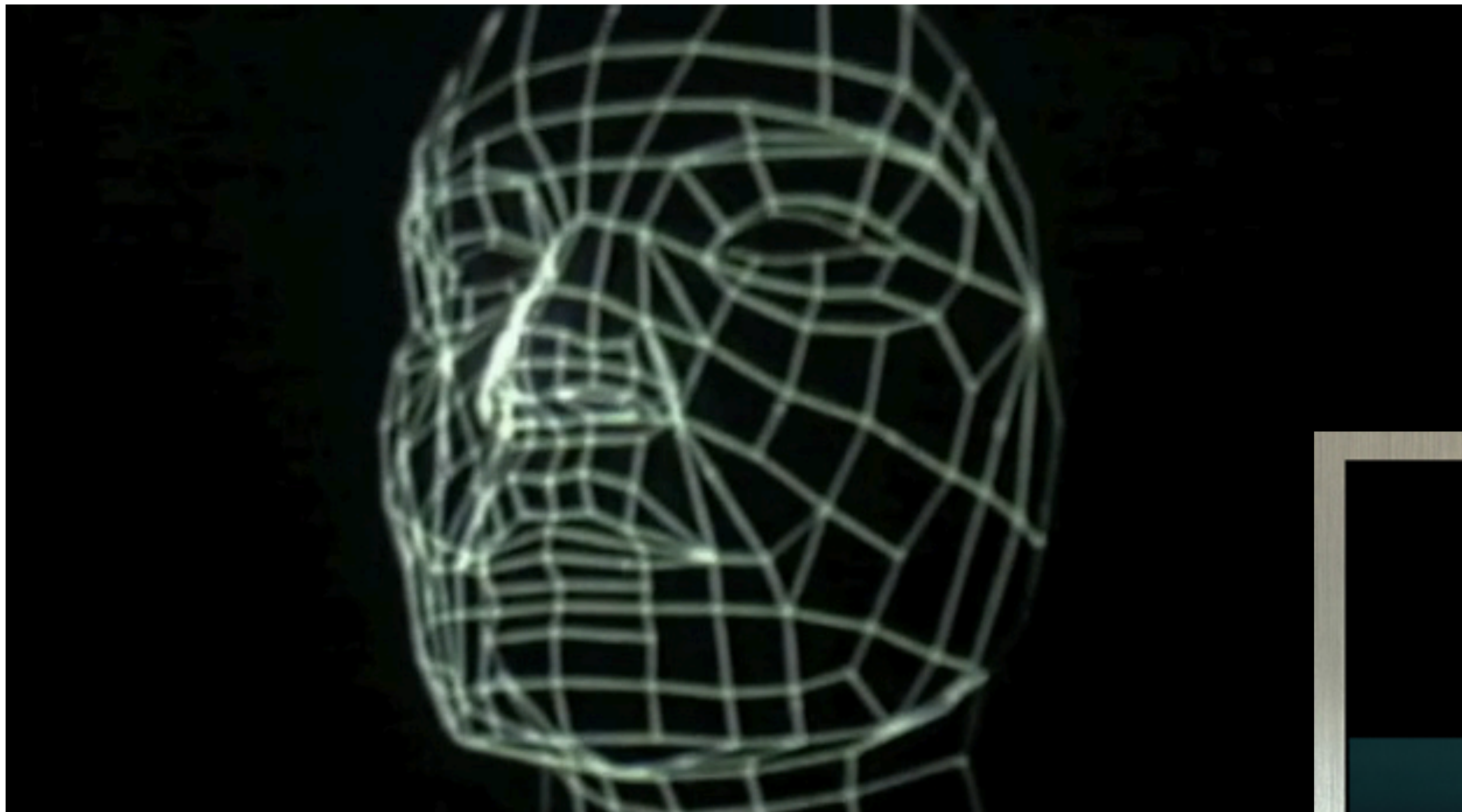
Masaccio – The Tribute Money c.1426-27
Fresco, The Brancacci Chapel, Florence

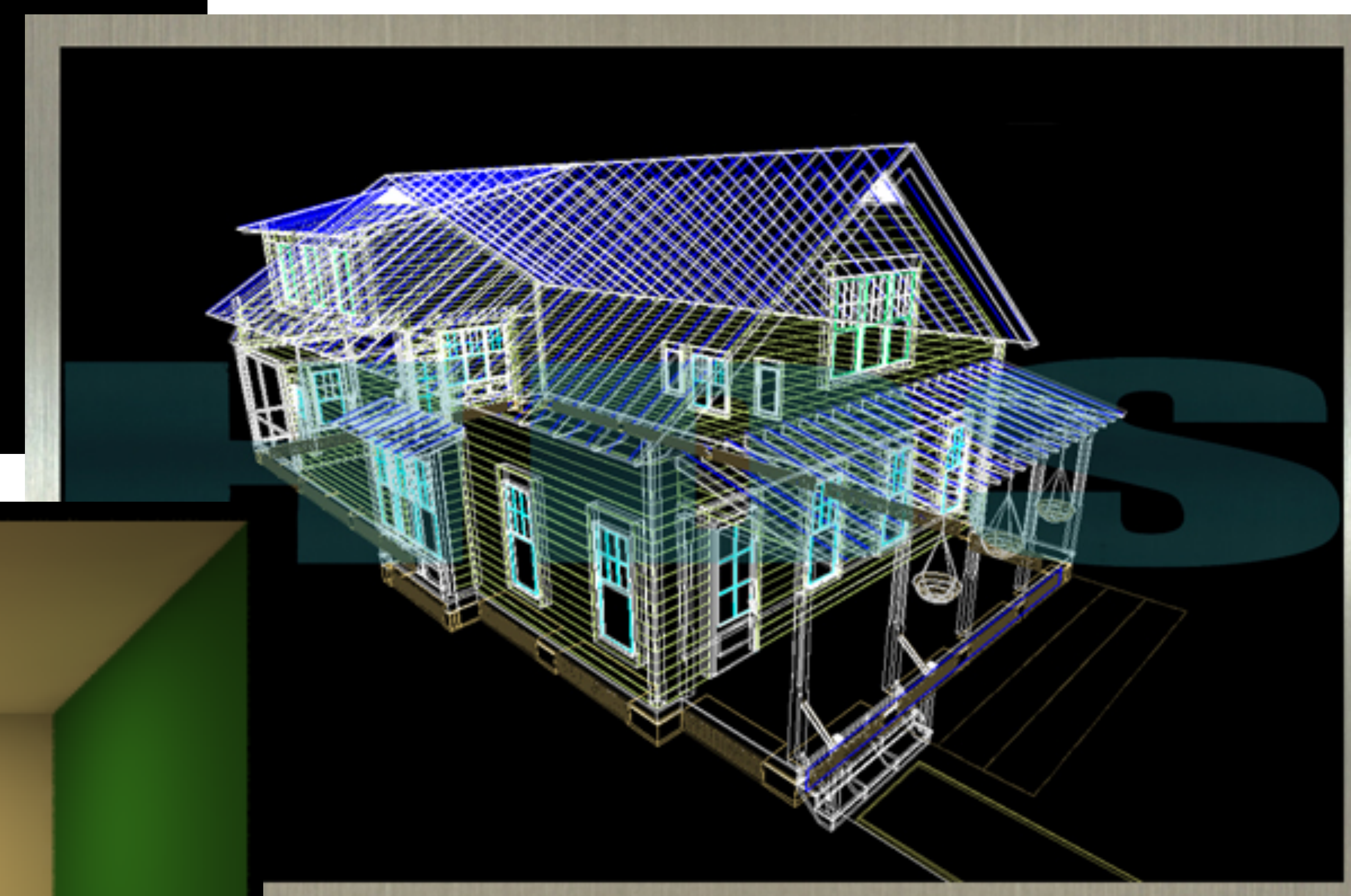
Perspective in art

Delivery of the Keys (Sistine Chapel), Perugino, 1482

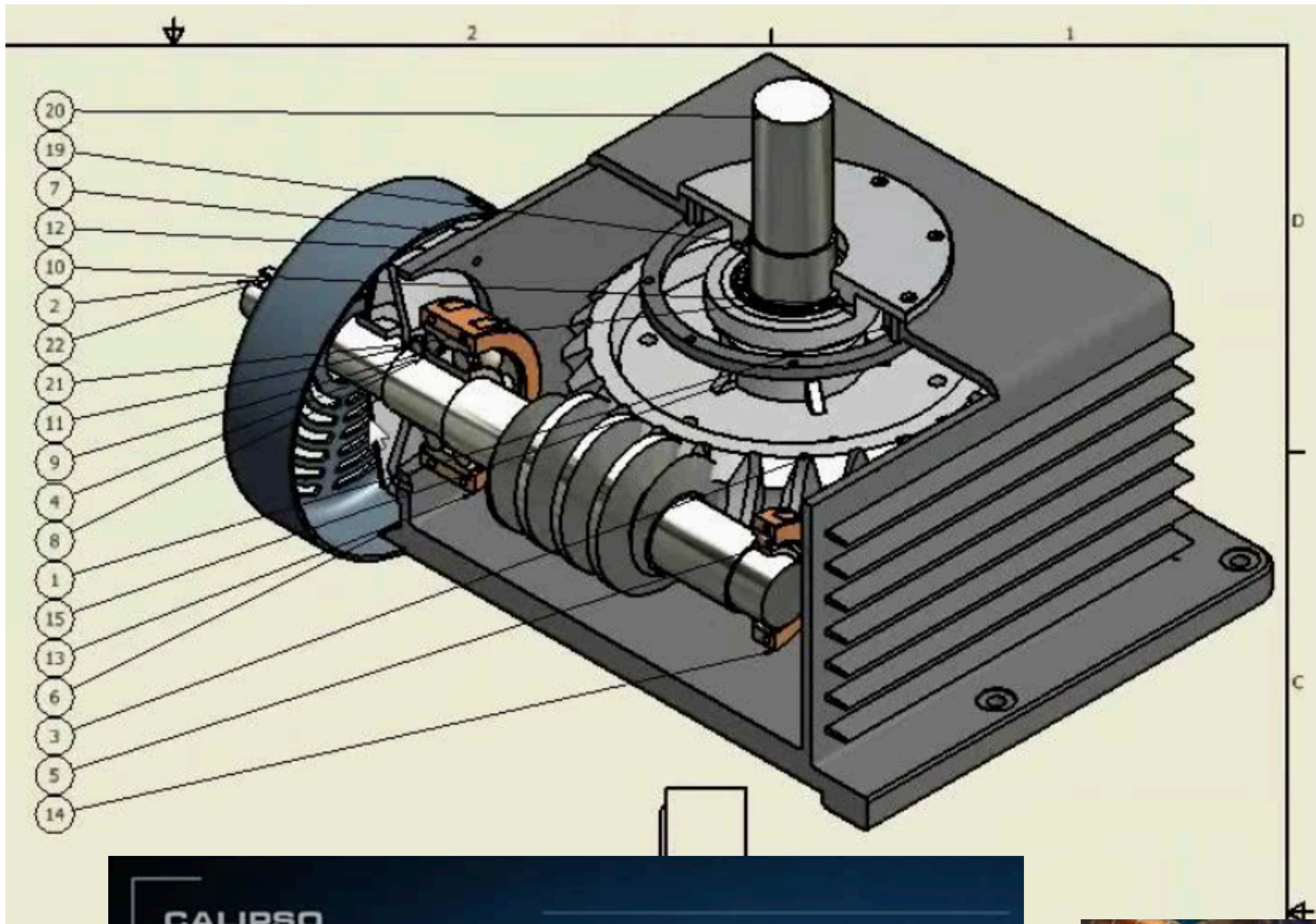
Later… rejection of proper perspective projection

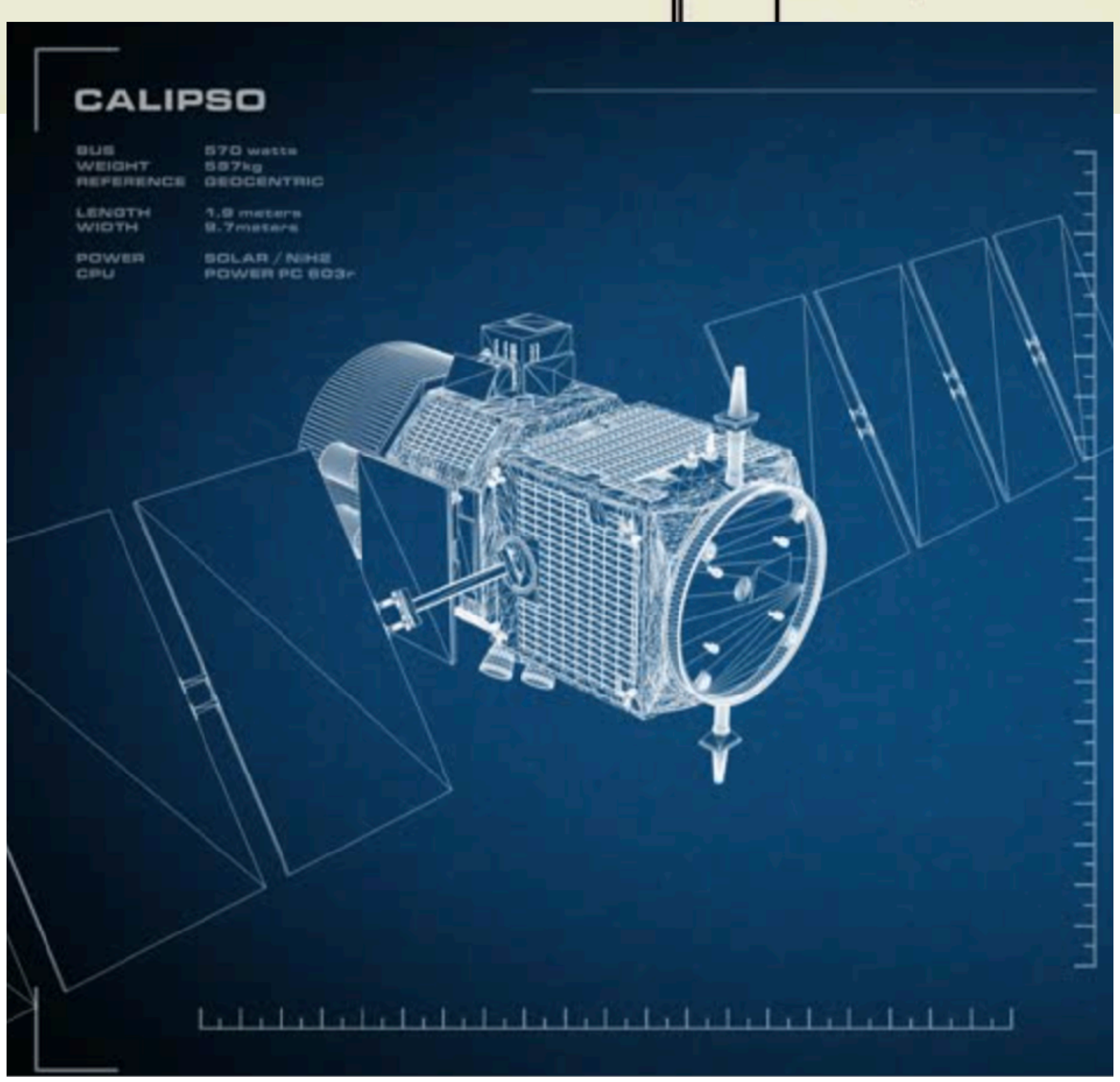
Return of perspective in computer graphics



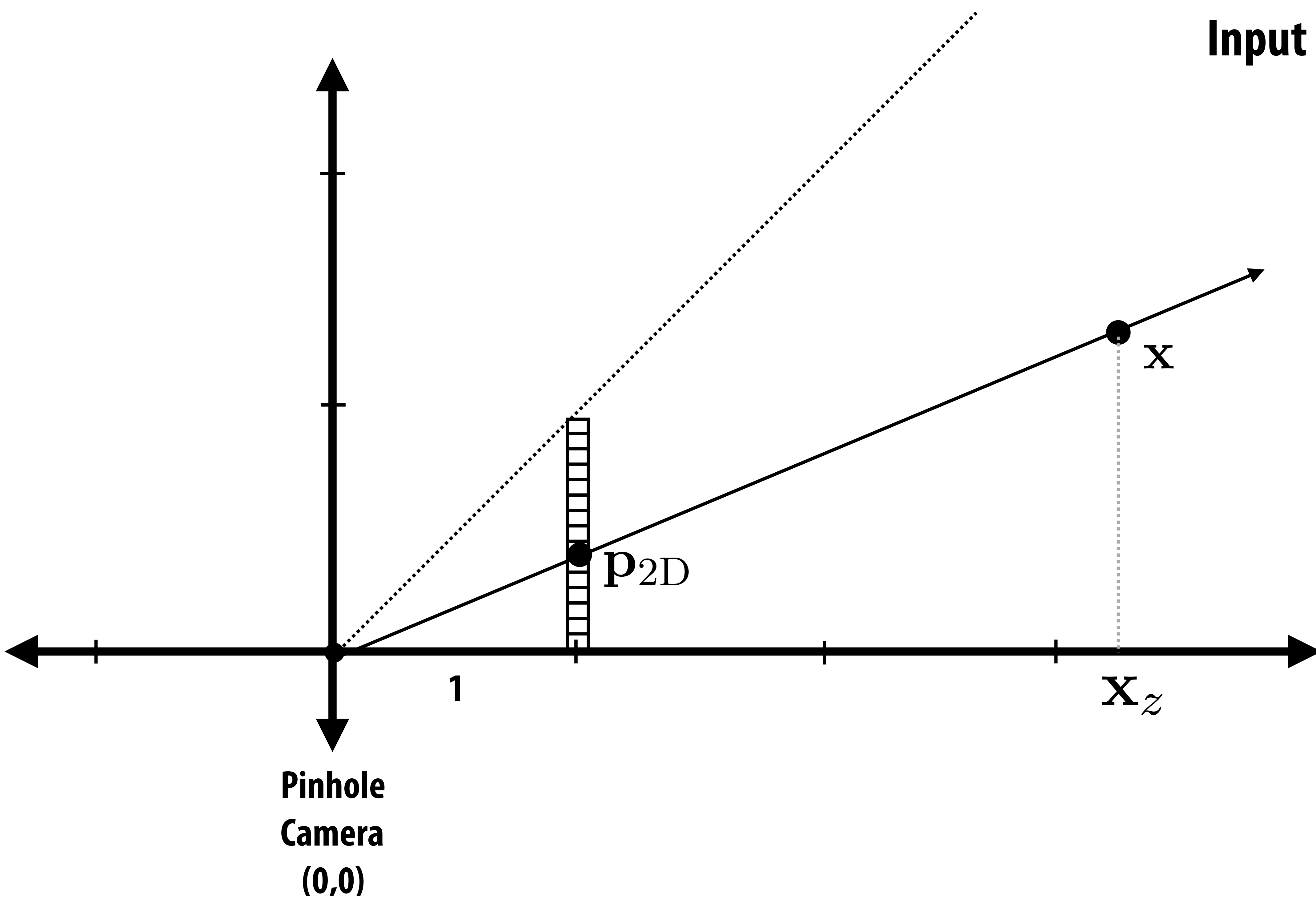


Rejection of perspective in computer graphics





Basic perspective projection



Input point in 3D-H: $\mathbf{x} = \begin{bmatrix} \mathbf{x}_x & \mathbf{x}_y & \mathbf{x}_z & 1 \end{bmatrix}^T$

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Assumption:

Pinhole camera at (0,0) looking down z

Perspective vs. orthographic projection

- **Most basic version of perspective matrix:**

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z \end{bmatrix} \longmapsto \begin{bmatrix} x/z \\ y/z \\ 1 \\ 1 \end{bmatrix}$$

objects shrink in distance

- **Most basic version of orthographic matrix:**

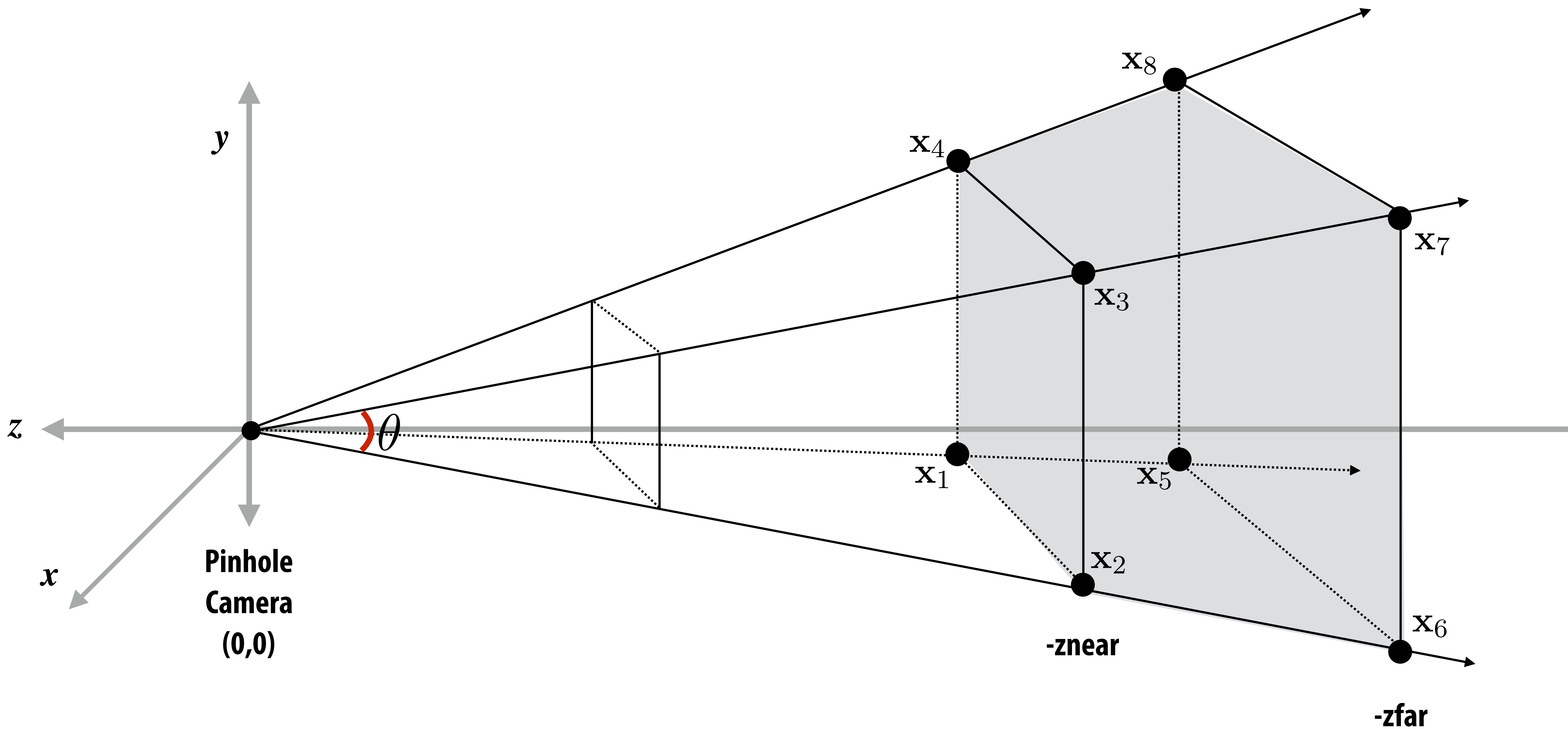
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \longmapsto \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

objects stay the same size

…real projection matrices are a bit more complicated! :-)

View frustum

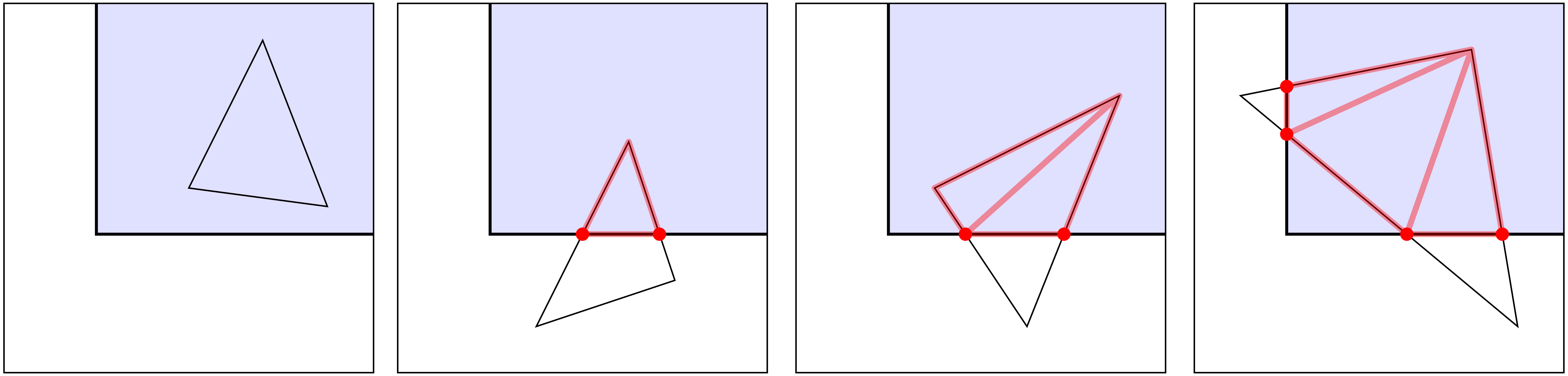
View frustum is the region of space the camera can see:



- Top/bottom/left/right planes correspond to sides of screen
- Near/far planes correspond to closest/furthest thing we want to draw

Clipping

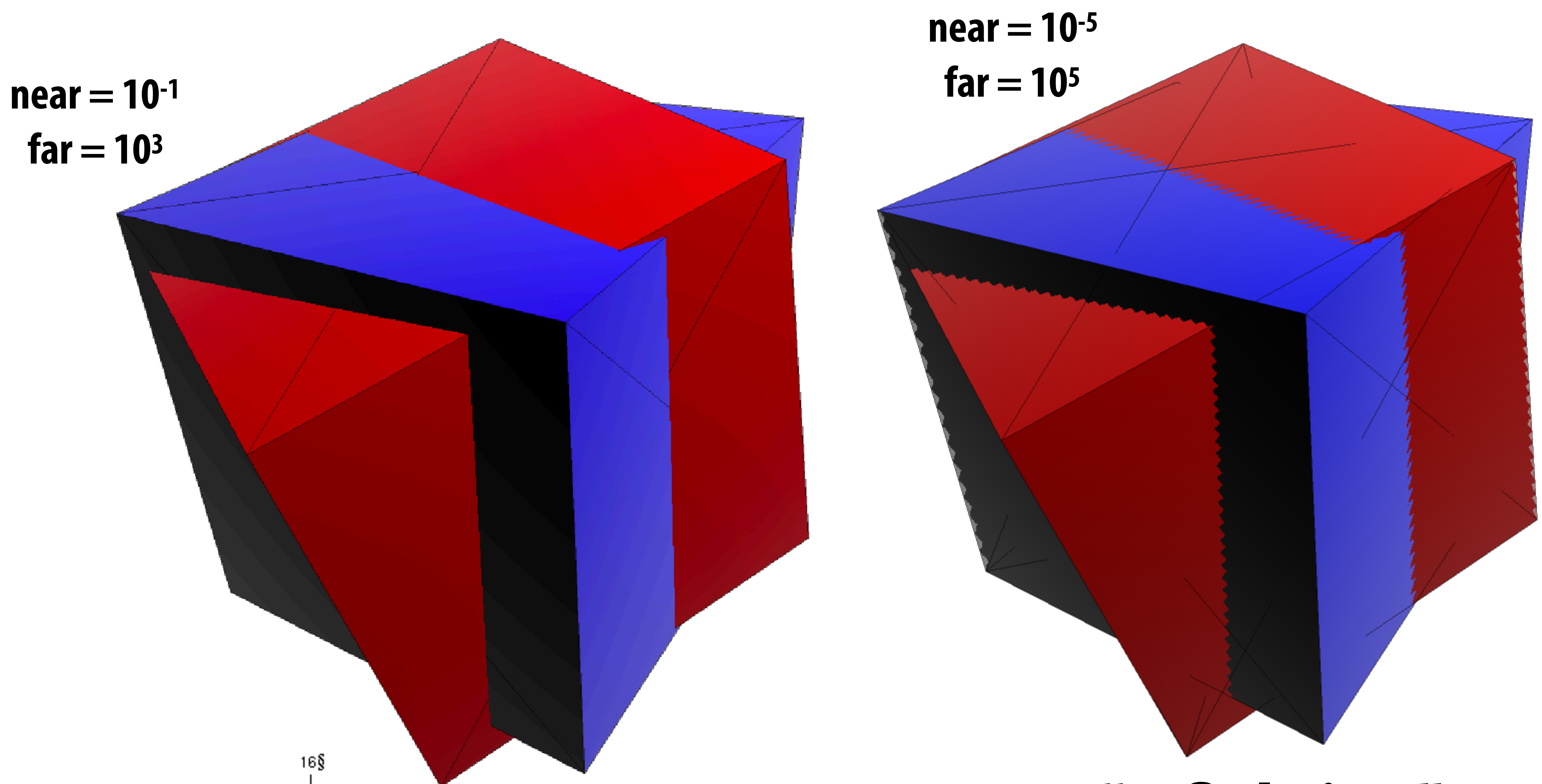
- "Clipping" is the process of eliminating triangles that aren't visible to the camera (outside the view frustum)
  - Don't waste time computing appearance of primitives you can't see!
  - Sample-covered-by-triangle tests are expensive ("fine granularity" visibility)
  - Makes more sense to toss out entire primitives ("coarse granularity")
  - Must deal with primitives that are partially clipped…



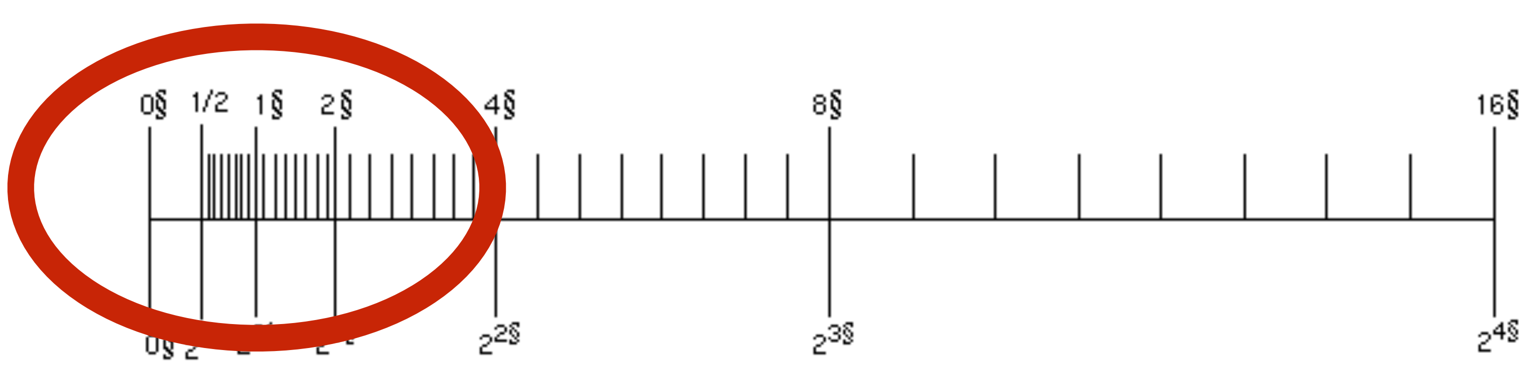
from: https://paroj.github.io/gltut/

Aside: near/far plane clipping

- But why *near/far plane* clipping?
  - Primitives (e.g., triangles) may have vertices both in front and behind camera! (Causes headaches for rasterization, e.g., checking if fragments are behind camera)
  - Also important for dealing with finite precision of depth buffer



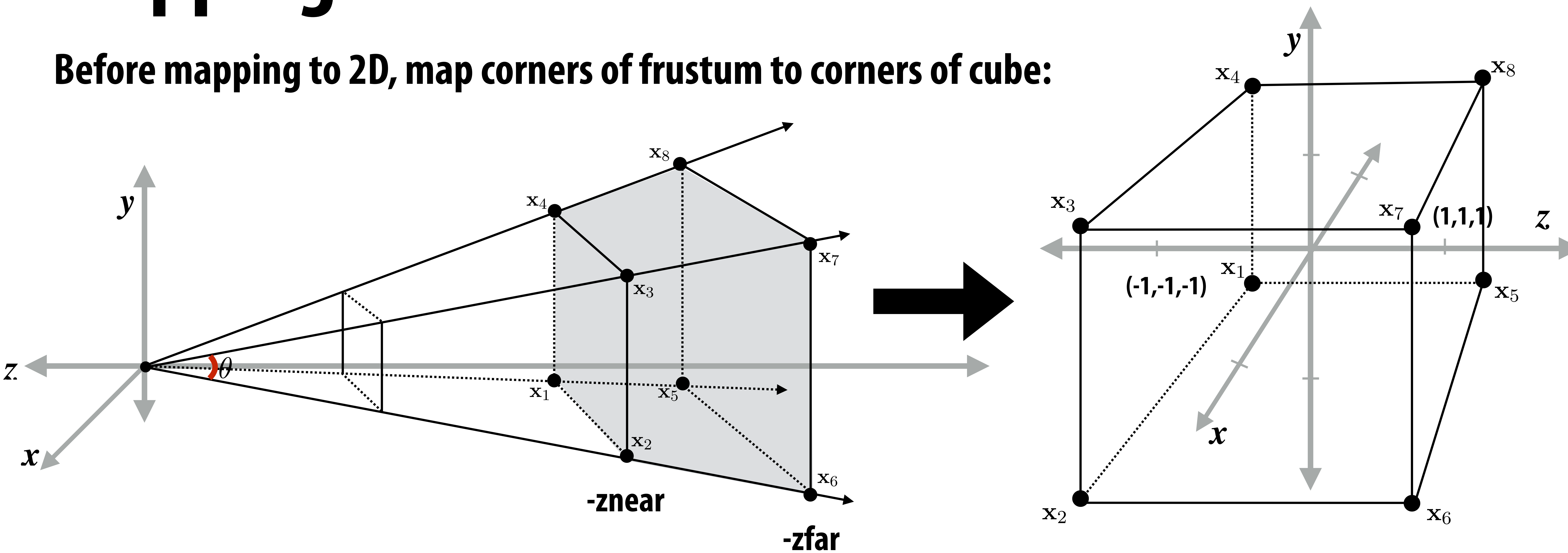
"Z-fighting"



floating point has more "resolution" near zero—hence more precise resolution of primitive-primitive intersection

Mapping frustum to normalized cube

Before mapping to 2D, map corners of frustum to corners of cube:



Why do we do this?

1. Makes *clipping* much easier!
   - can quickly discard geometry outside range [-1,1]
2. Avoid issues of precision of perspective divide near origin
3. Different maps to cube yield different effects
   - specifically: perspective or orthographic
   - perspective is affine transformation, implemented via homogeneous coordinates
   - for orthographic view, just use identity matrix!

Perspective:

Set homogeneous coord to "z"

Distant objects get smaller

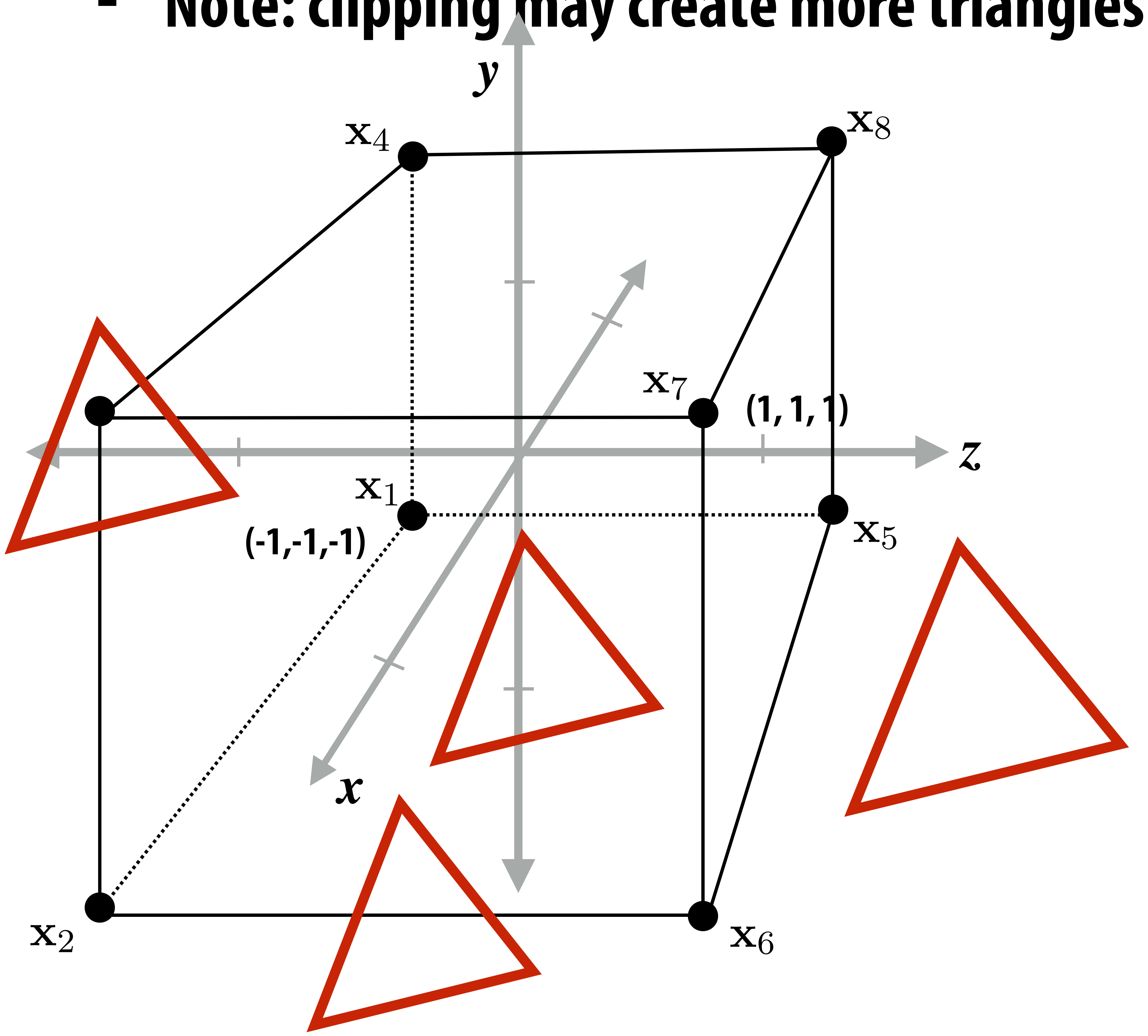
Orthographic: (not shown)

Set homogeneous coord to "1"

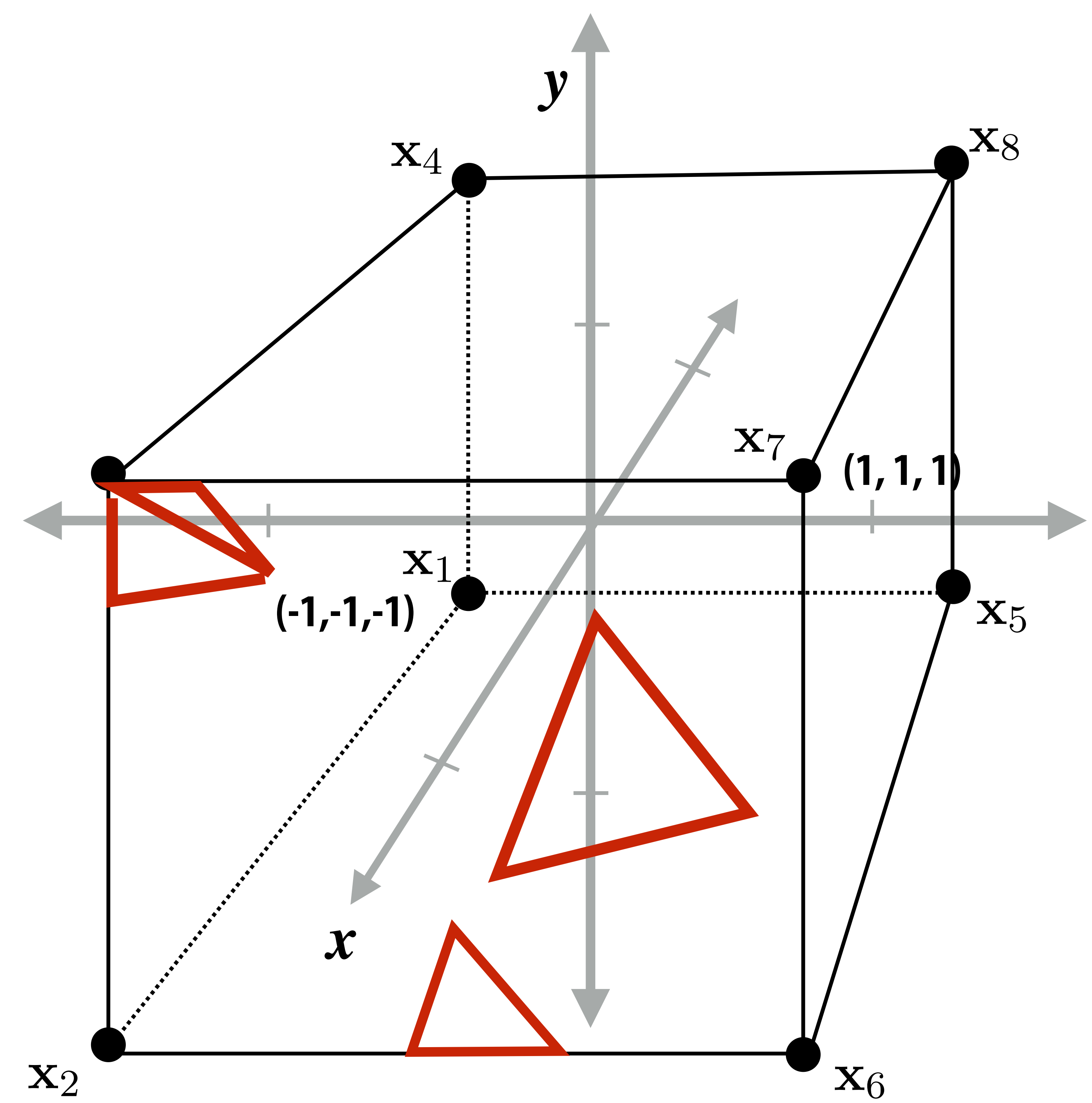
Distant objects remain same size

Clipping in normalized device coordinates (NDC)

- Discard triangles that lie complete outside the normalized cube (culling)
  - They are off screen, don't bother processing them further
- Clip triangles that extend beyond the cube… to the sides of the cube
  - Note: clipping may create more triangles



Triangles before clipping

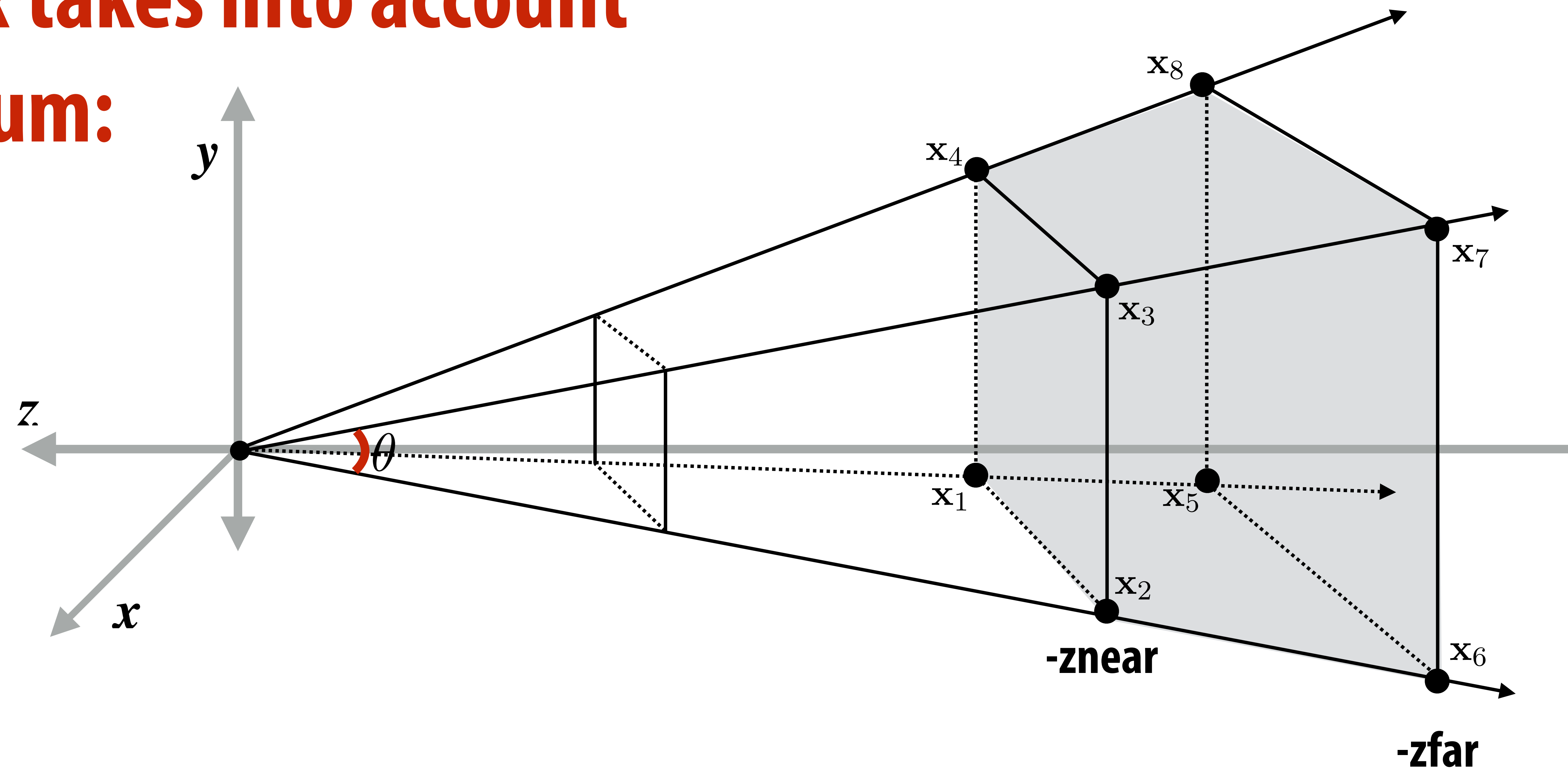


Triangles after clipping

* Figures are correct: OpenGL NDC is left-handed coordinate space

Matrix for perspective transform

Real perspective matrix takes into account geometry of view frustum:



$$\begin{pmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

left (l), right (r), top (t), bottom (b), near (n), far (f)

(matrix at left is perspective projection for frustum that is symmetric about x,y axes: l=-r, t=-b)

For a derivation: http://www.songho.ca/opengl/gl_projectionmatrix.html

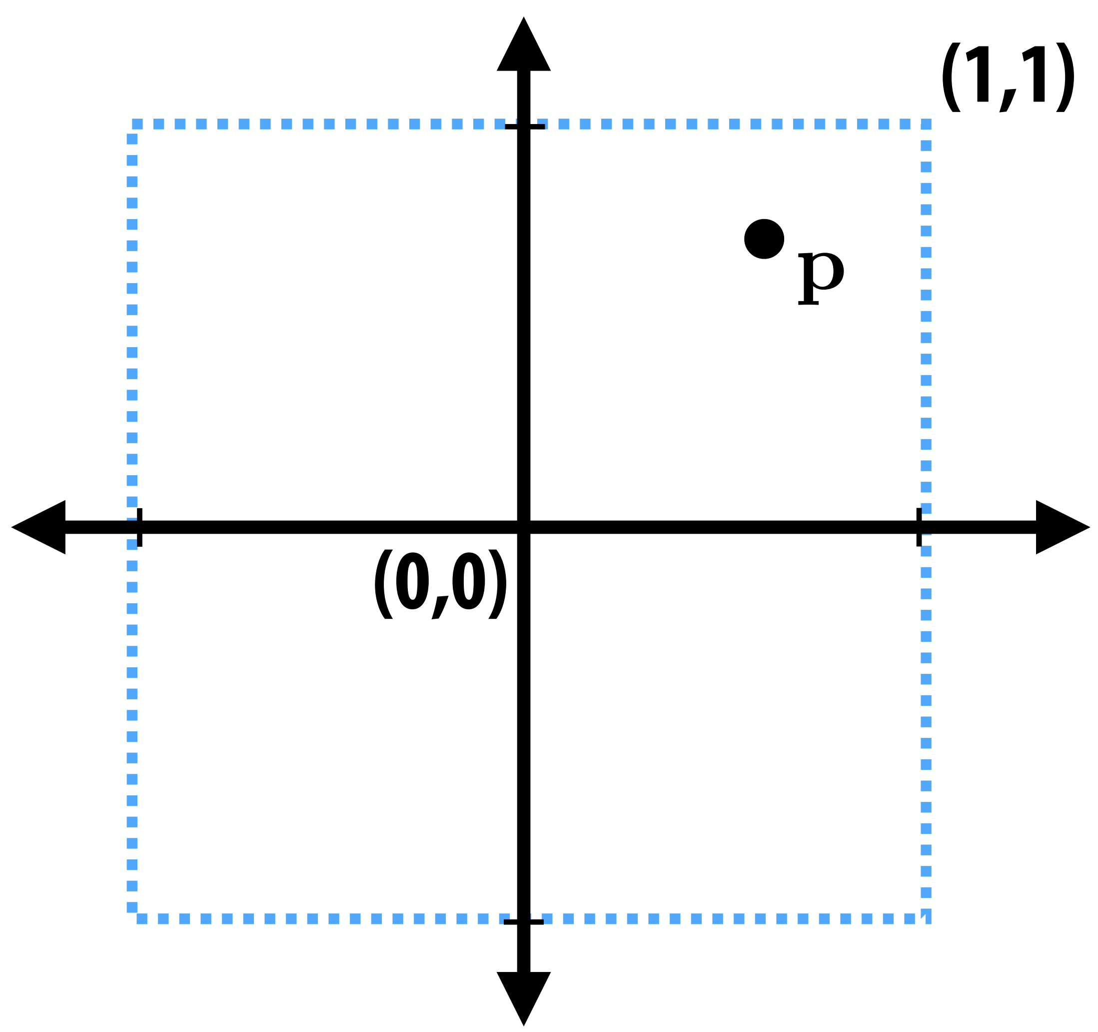
Review: screen transform

After divide, coordinates in [-1,1] have to be "stretched" to fit the screen

Example:

- All points within (-1,1) to (1,1) region are on screen
- (1,1) in normalized space maps to (W,0) in screen

Normalized coordinate space:

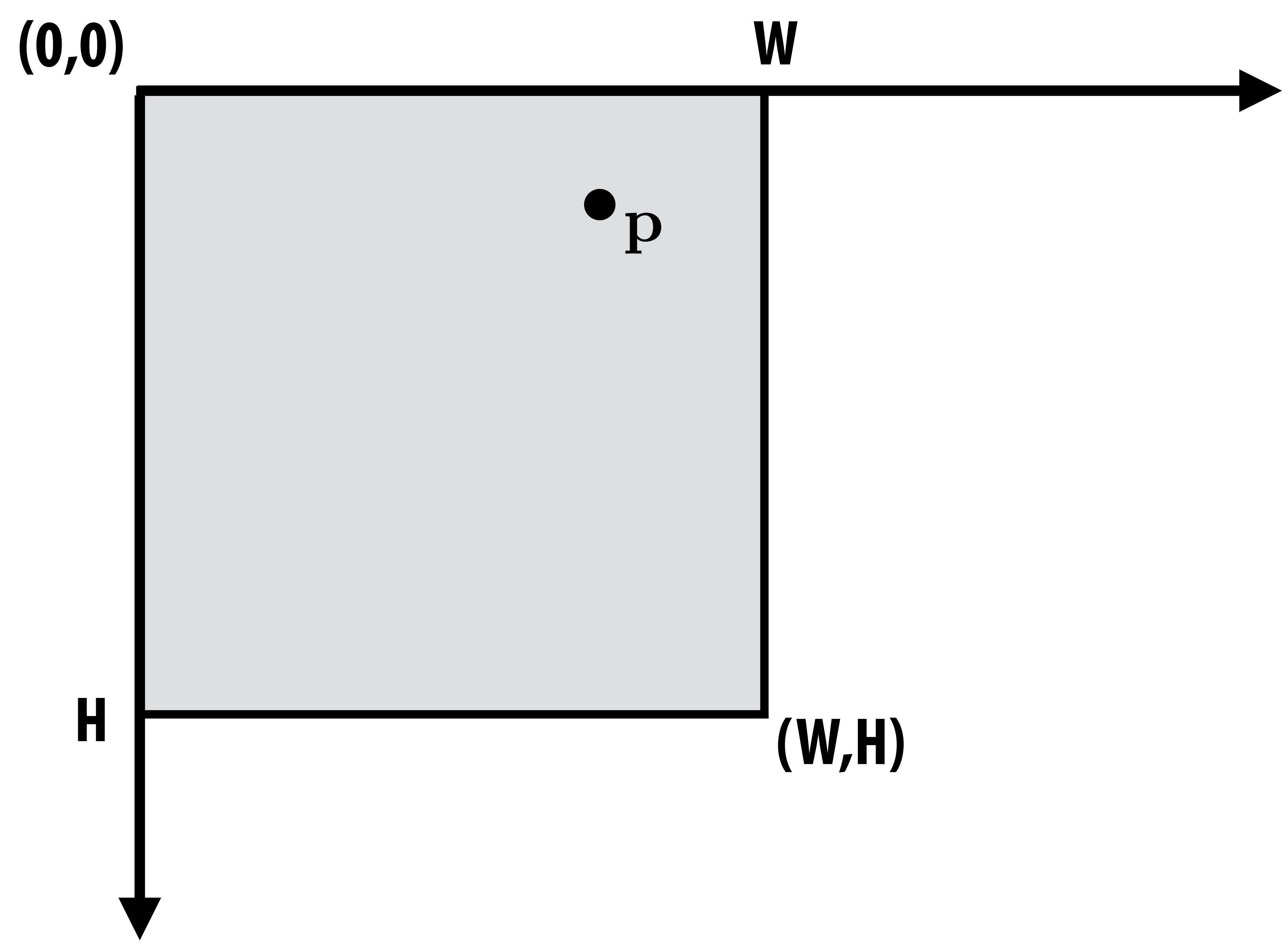


Step 1: reflect about x-axis

Step 2: translate by (1,1)

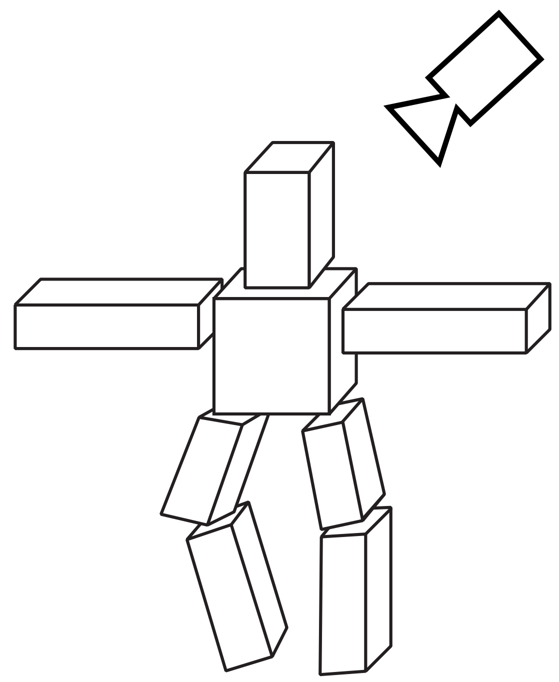
Step 3: scale by (W/2,H/2)

Screen (W x H output image) coordinate space:



Transformations: from objects to the screen

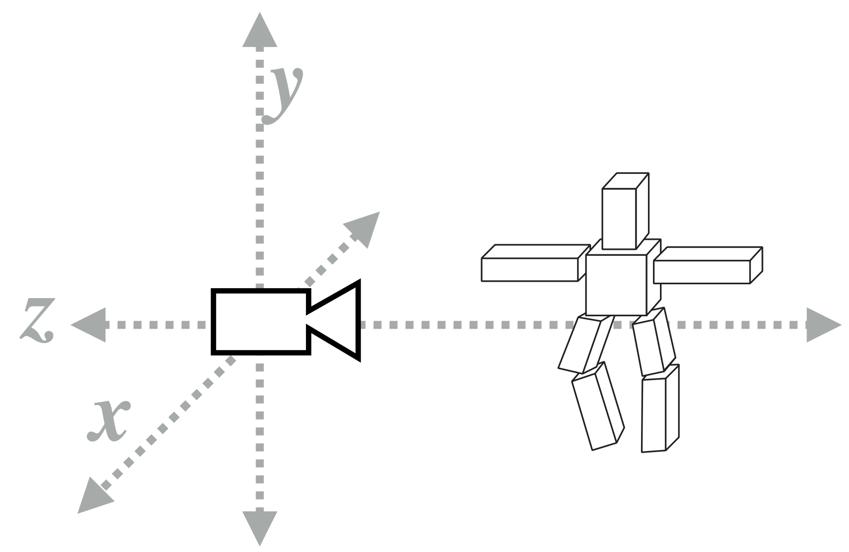
[WORLD COORDINATES]



original description of objects

view transform

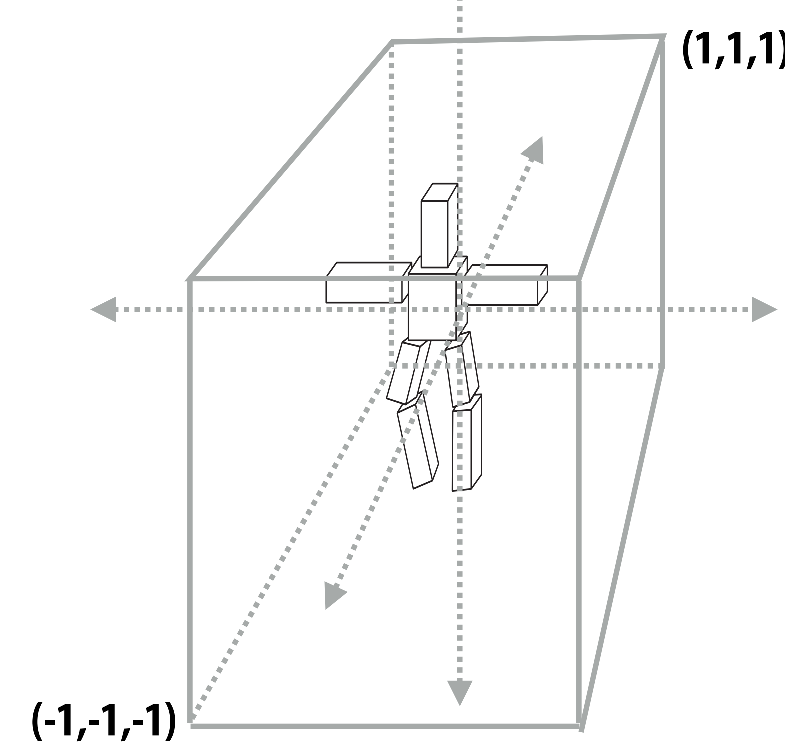
[VIEW COORDINATES]



all positions now expressed relative to camera; camera is sitting at origin looking down -z direction (can canonicalize projection matrix)

projection transform

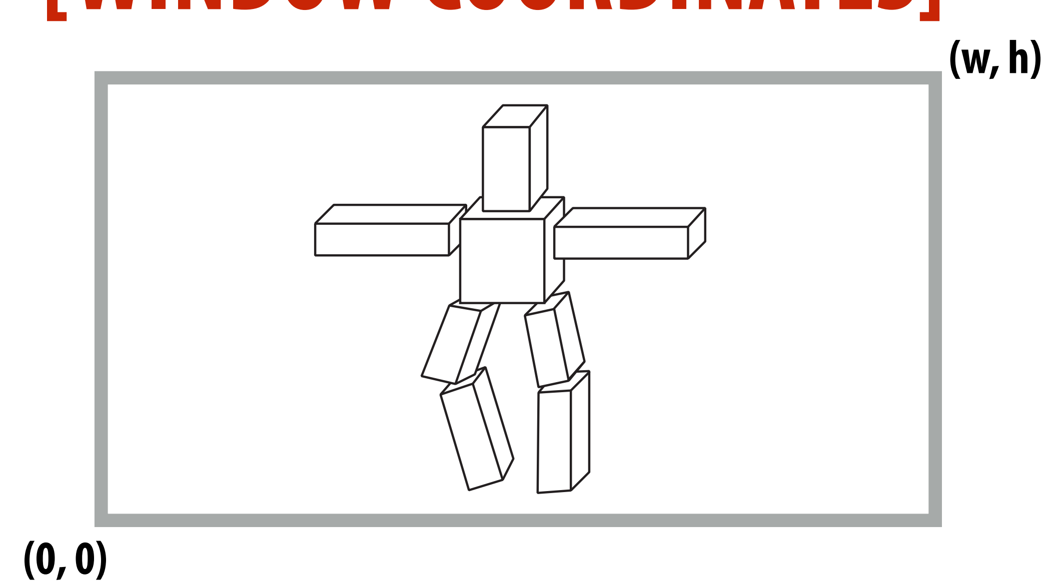
[CLIP COORDINATES]



everything visible to the camera is mapped to unit cube for easy "clipping"

screen transform

[WINDOW COORDINATES]



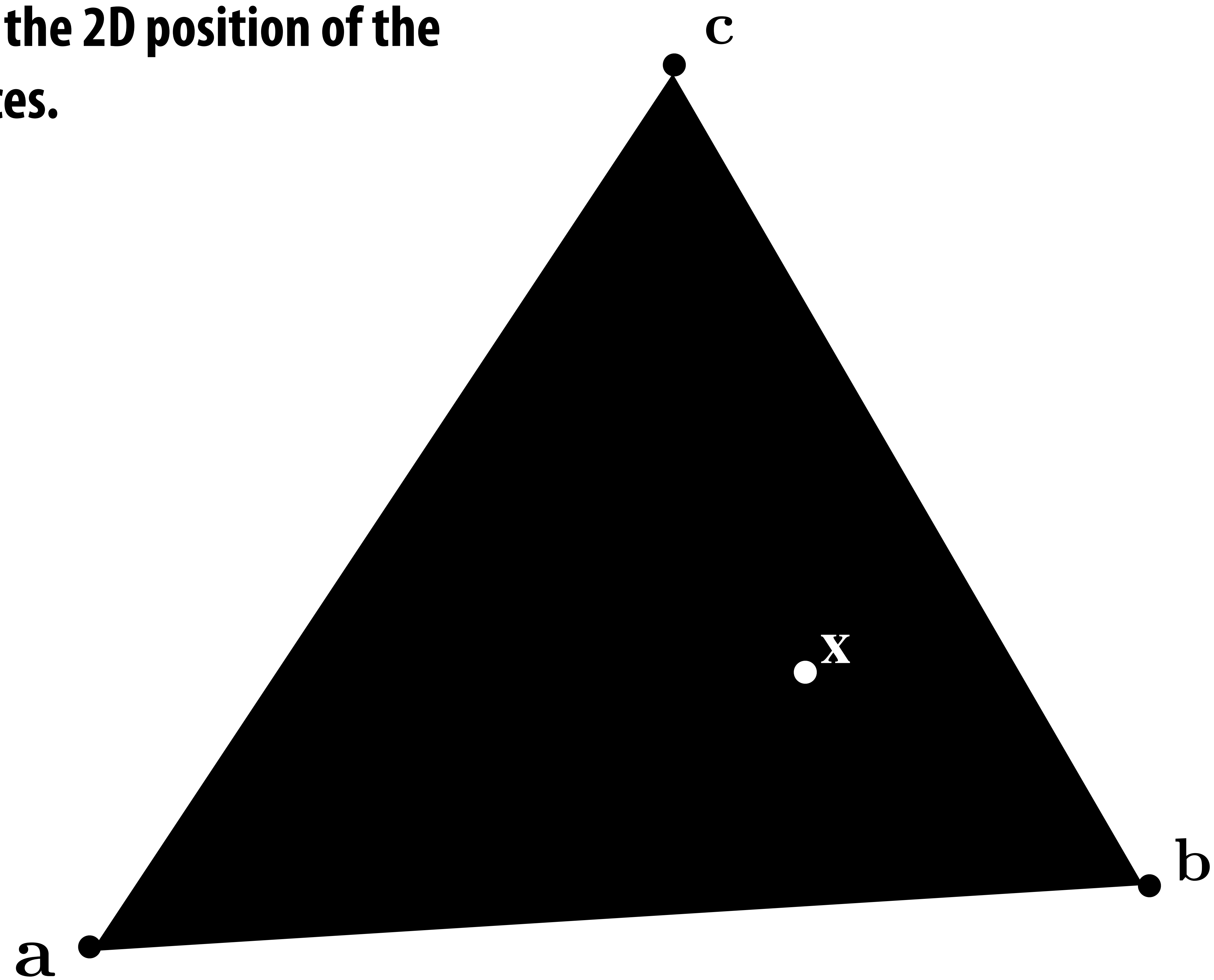
objects now in 2D screen coordinates

primitives are now 2D and can be drawn via rasterization

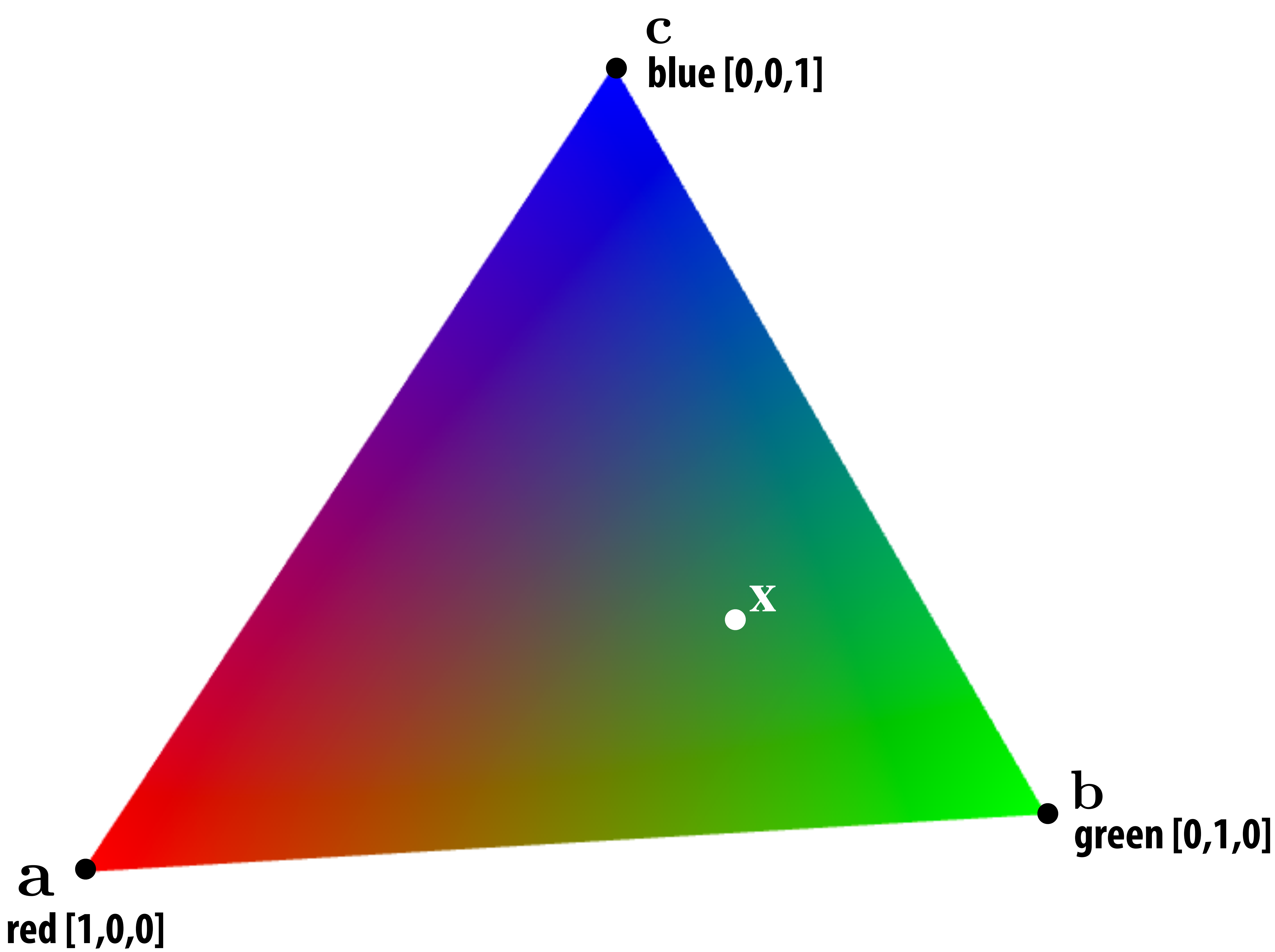
Texture mapping

Coverage(x,y)

In lecture 2 we discussed how to sample coverage given the 2D position of the triangle's vertices.



Consider sampling color(x,y)



What is the triangle's color at the point $\mathbf{x}$?

Review: interpolation in 1D

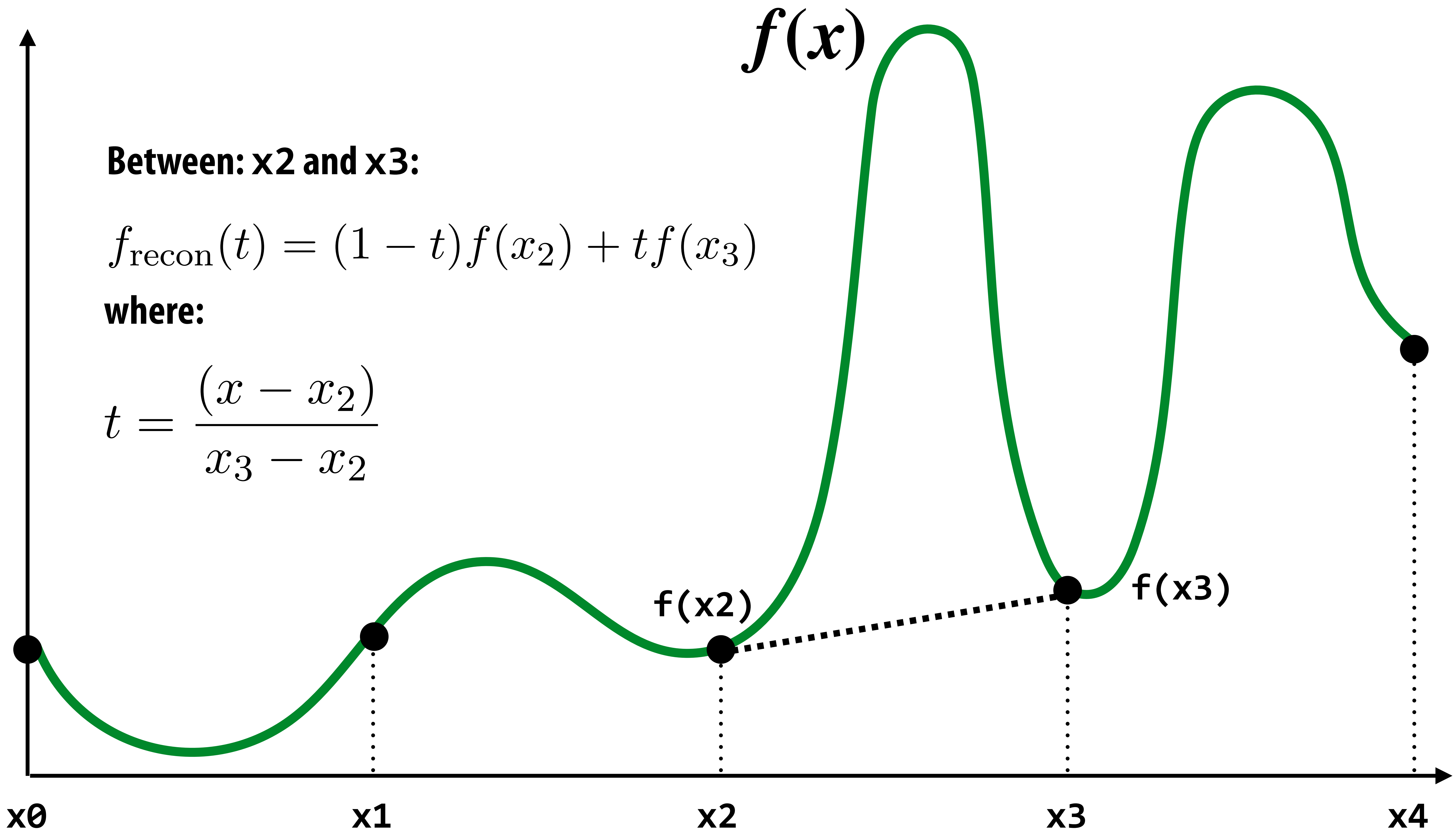
$f_{recon}(x)$ = linear interpolation between values of two closest samples to x

Between: x2 and x3:

$$f_{\text{recon}}(t) = (1-t)f(x_2) + tf(x_3)$$

where:

$$t = \frac{(x - x_2)}{x_3 - x_2}$$

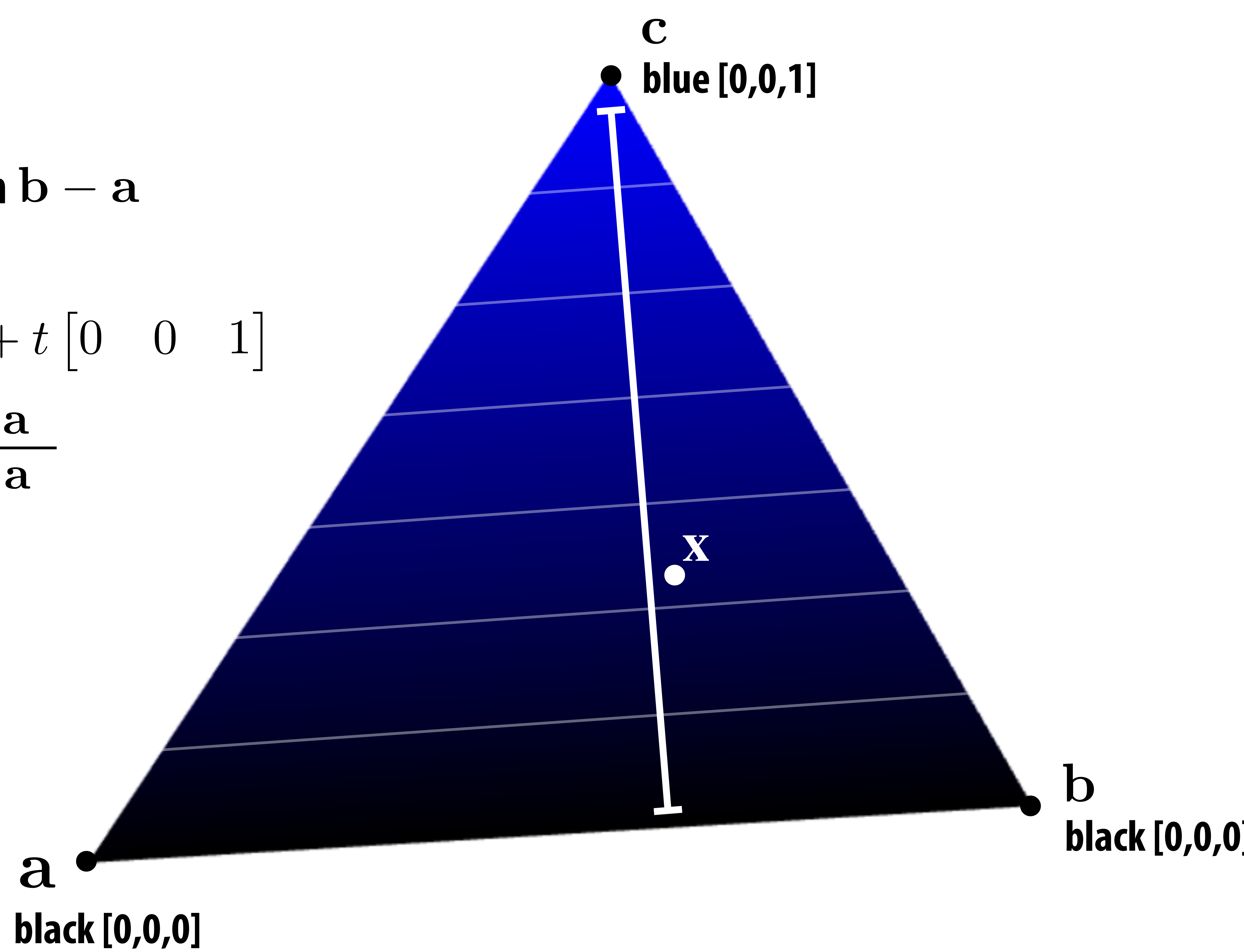


Consider similar behavior on triangle

Color depends on distance from $\mathbf{b} - \mathbf{a}$

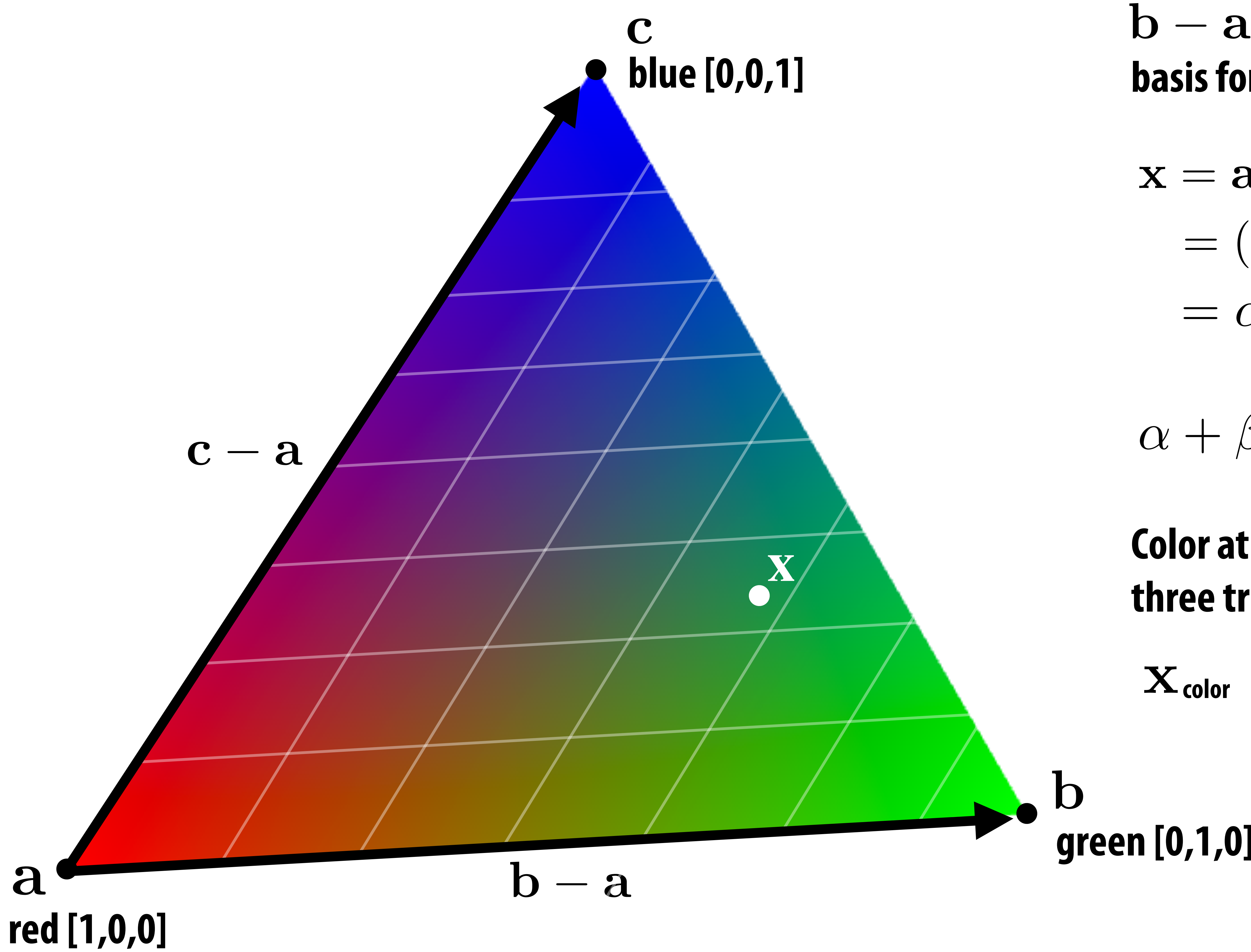
color at $(1-t)\begin{bmatrix}0 & 0 & 0\end{bmatrix} + t\begin{bmatrix}0 & 0 & 1\end{bmatrix}$

$$t = \frac{\text{distance from } \mathbf{x} \text{ to } \mathbf{b} - \mathbf{a}}{\text{distance from } \mathbf{c} \text{ to } \mathbf{b} - \mathbf{a}}$$



How can we interpolate in 2D between three values?

Interpolation via barycentric coordinates



$\mathbf{b} - \mathbf{a}$ and $\mathbf{c} - \mathbf{a}$ form a non-orthogonal basis for points in triangle (origin at $\mathbf{a}$)

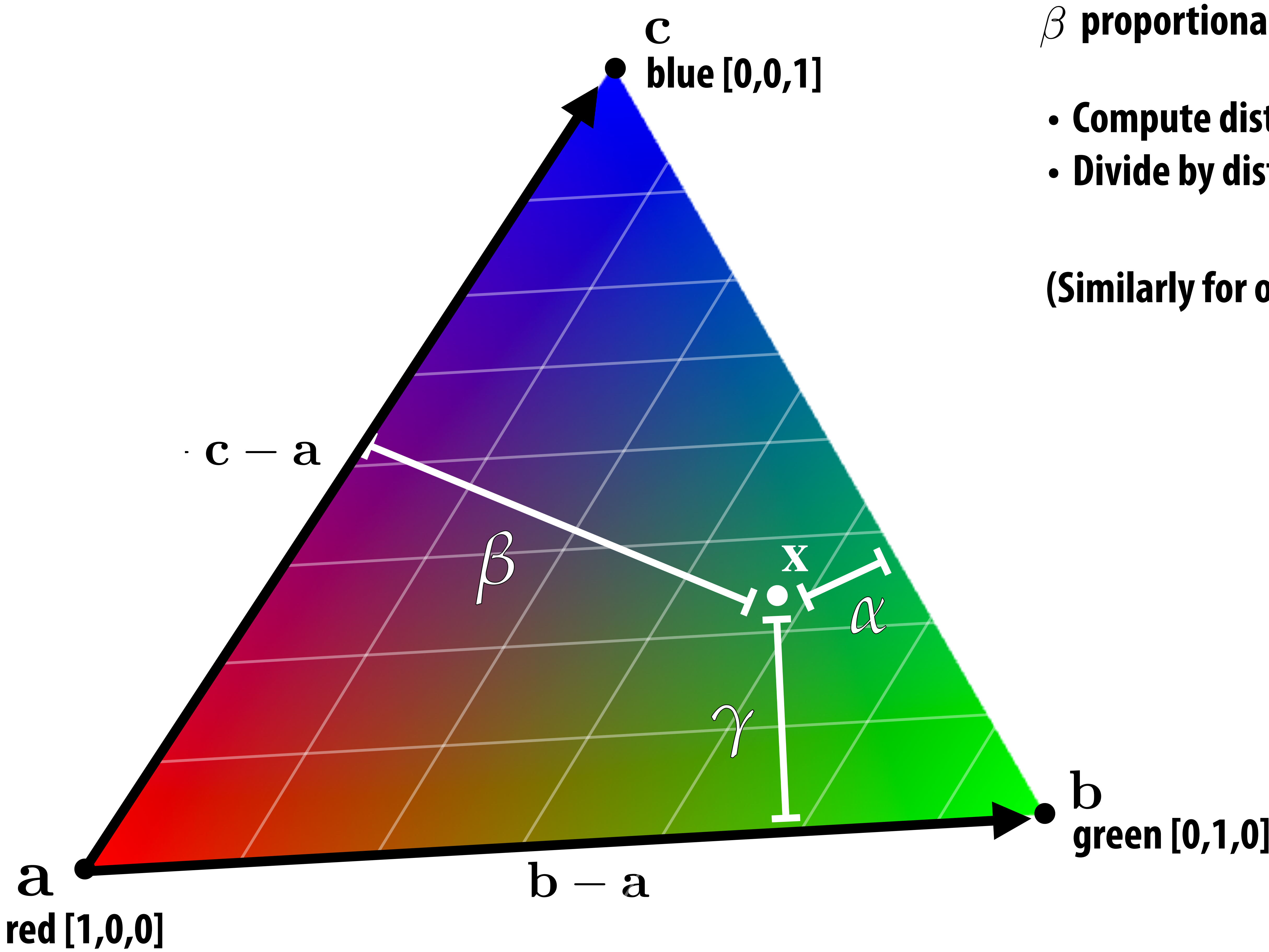
$$\begin{aligned}\mathbf{x} &= \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a}) \\ &= (1 - \beta - \gamma)\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c} \\ &= \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}\end{aligned}$$

$$\alpha + \beta + \gamma = 1$$

Color at $\mathbf{x}$ is linear combination of color at three triangle vertices.

$$\mathbf{x}_{\text{color}} = \alpha\mathbf{a}_{\text{color}} + \beta\,\mathbf{b}_{\text{color}} + \gamma\,\mathbf{c}_{\text{color}}$$

Barycentric coordinates as scaled distances

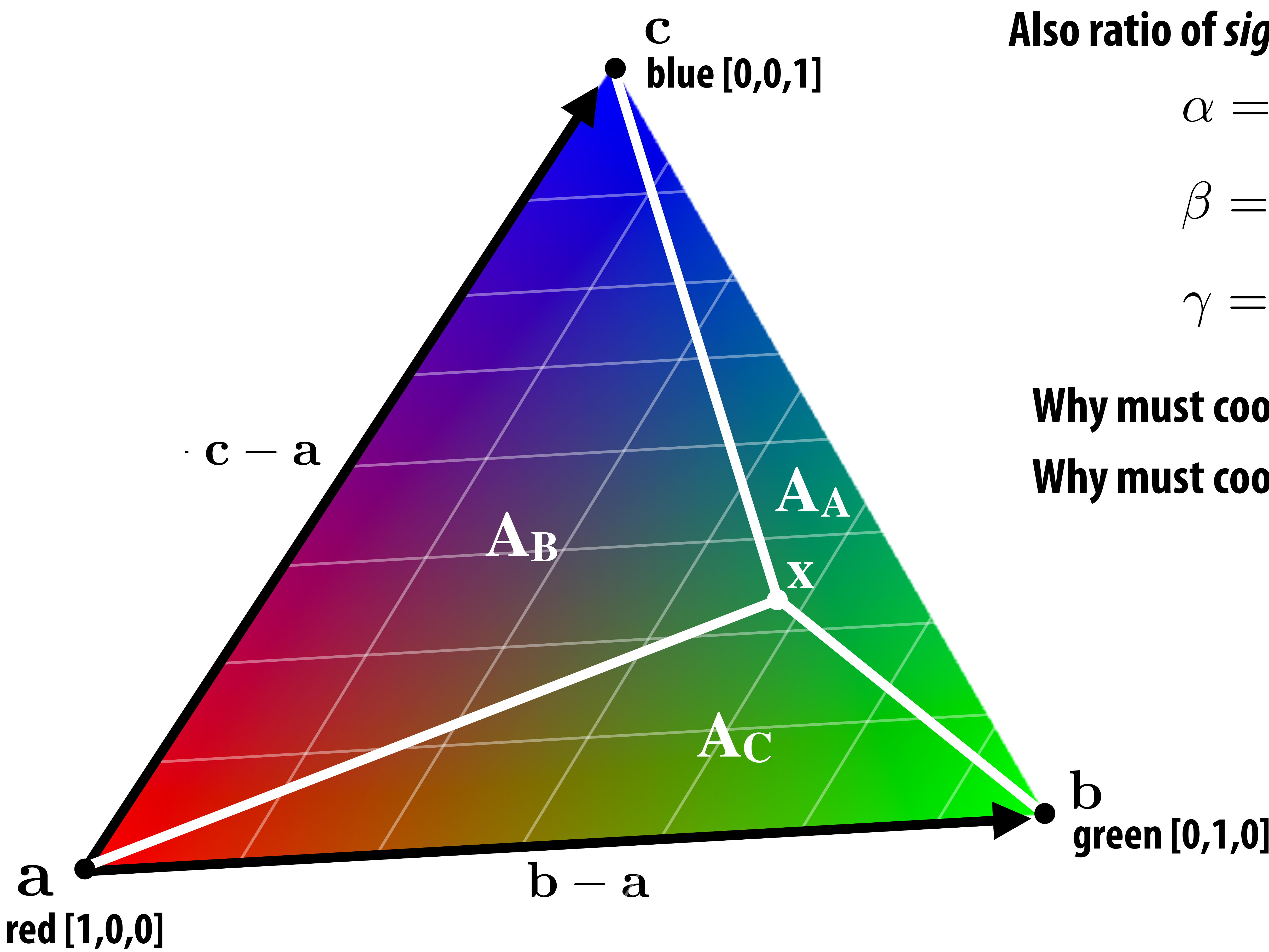


β proportional to distance from $\mathbf{x}$ to edge $\mathbf{c} - \mathbf{a}$

- Compute distance of x from line ca
- Divide by distance of b from line ca (“height”)

(Similarly for other two barycentric coordinates)

Barycentric coordinates as ratio of areas



Also ratio of *signed* areas:

$$\alpha = A_A/A$$

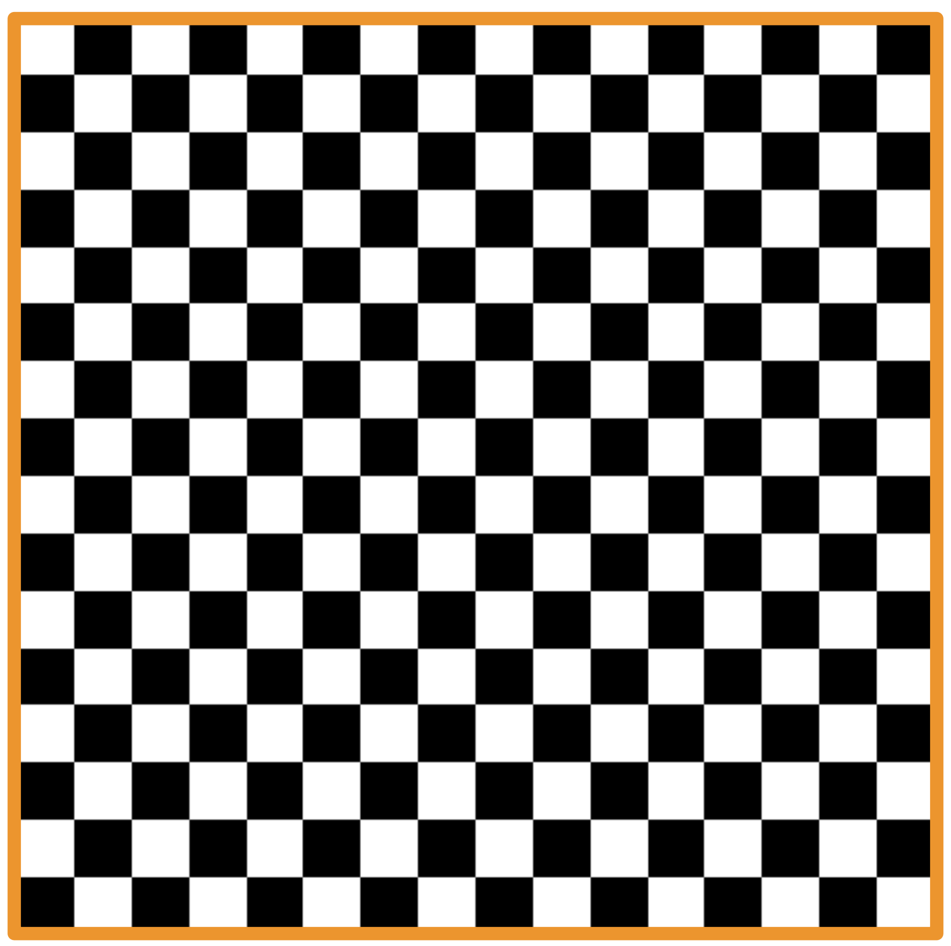
$$\beta = A_B/A$$

$$\gamma = A_C/A$$

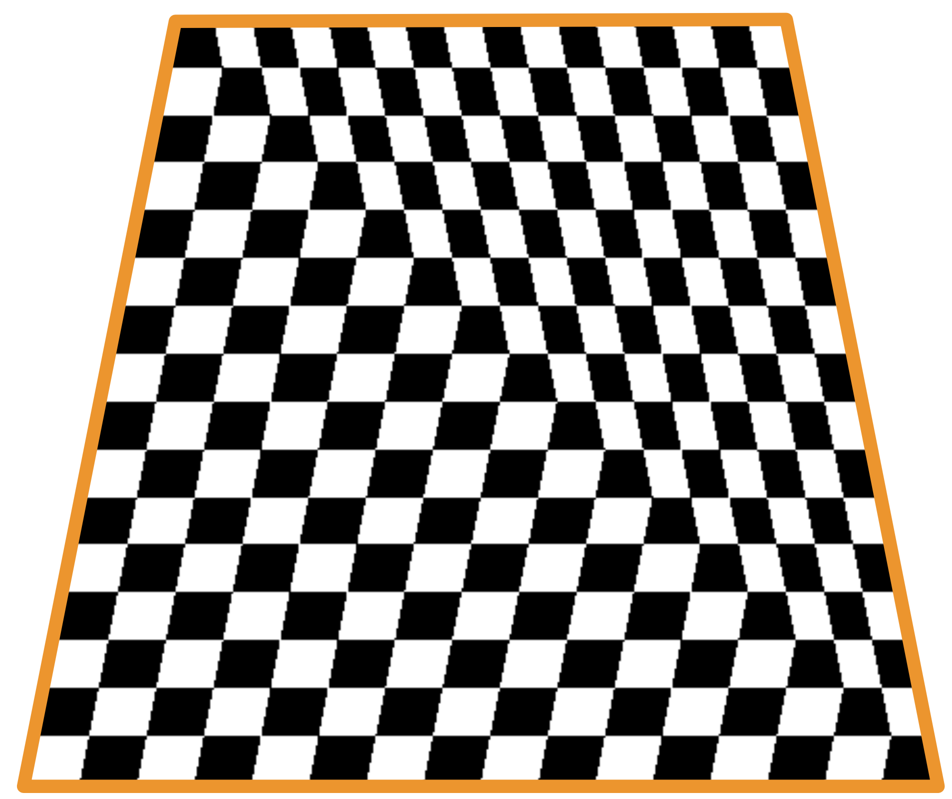
Why must coordinates sum to one?

Why must coordinates be between 0 and 1?

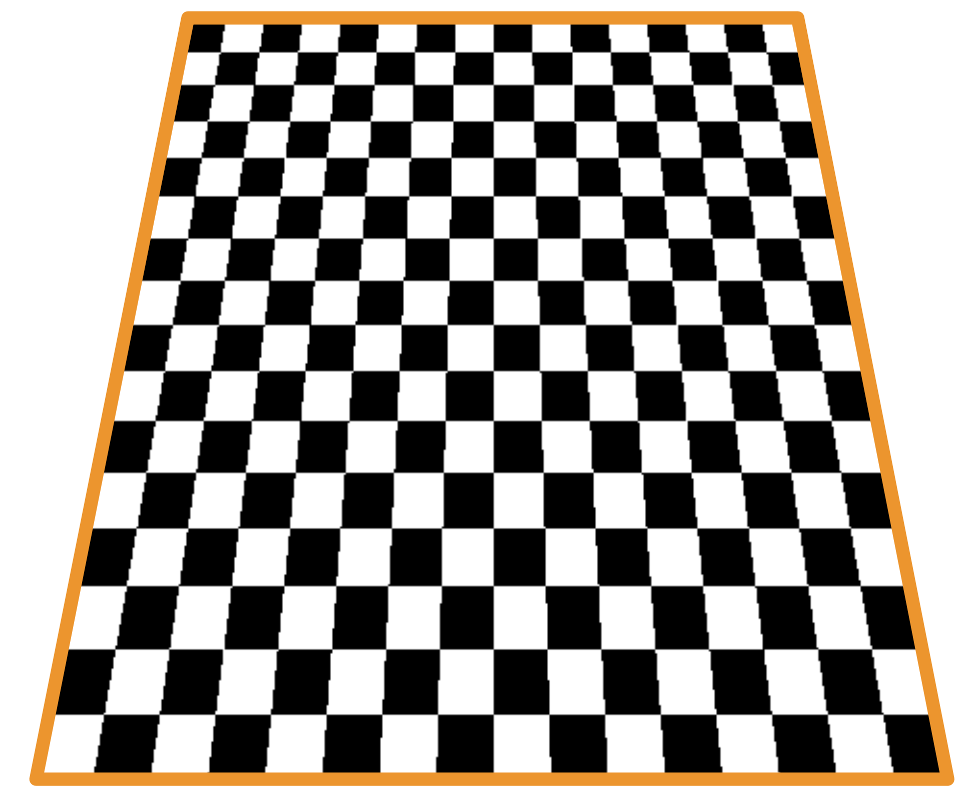
Perspective projection and interpolation



Texture



Plane tilted down with perspective projection — What’s wrong?

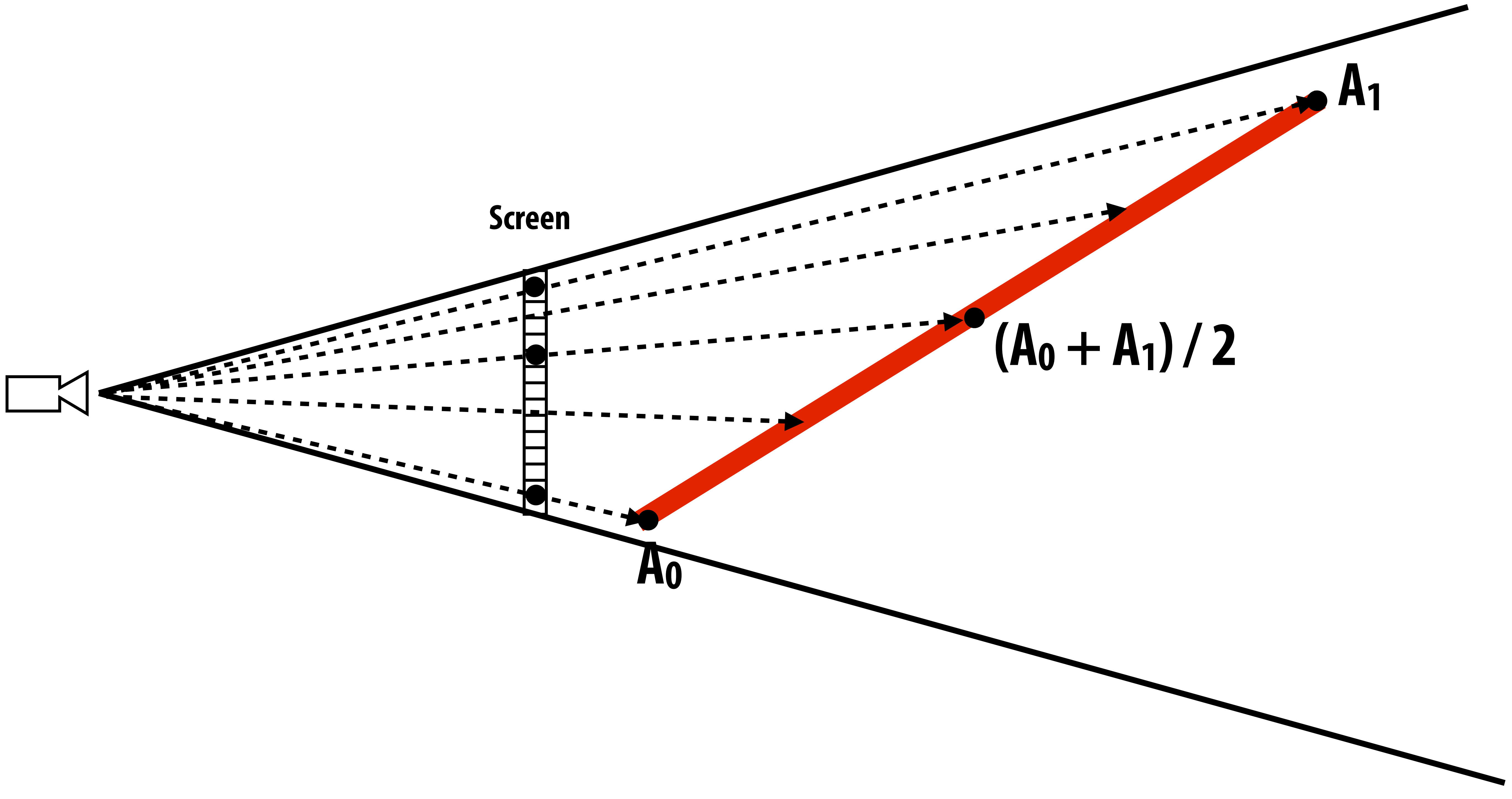


Correct image

Perspective incorrect interpolation

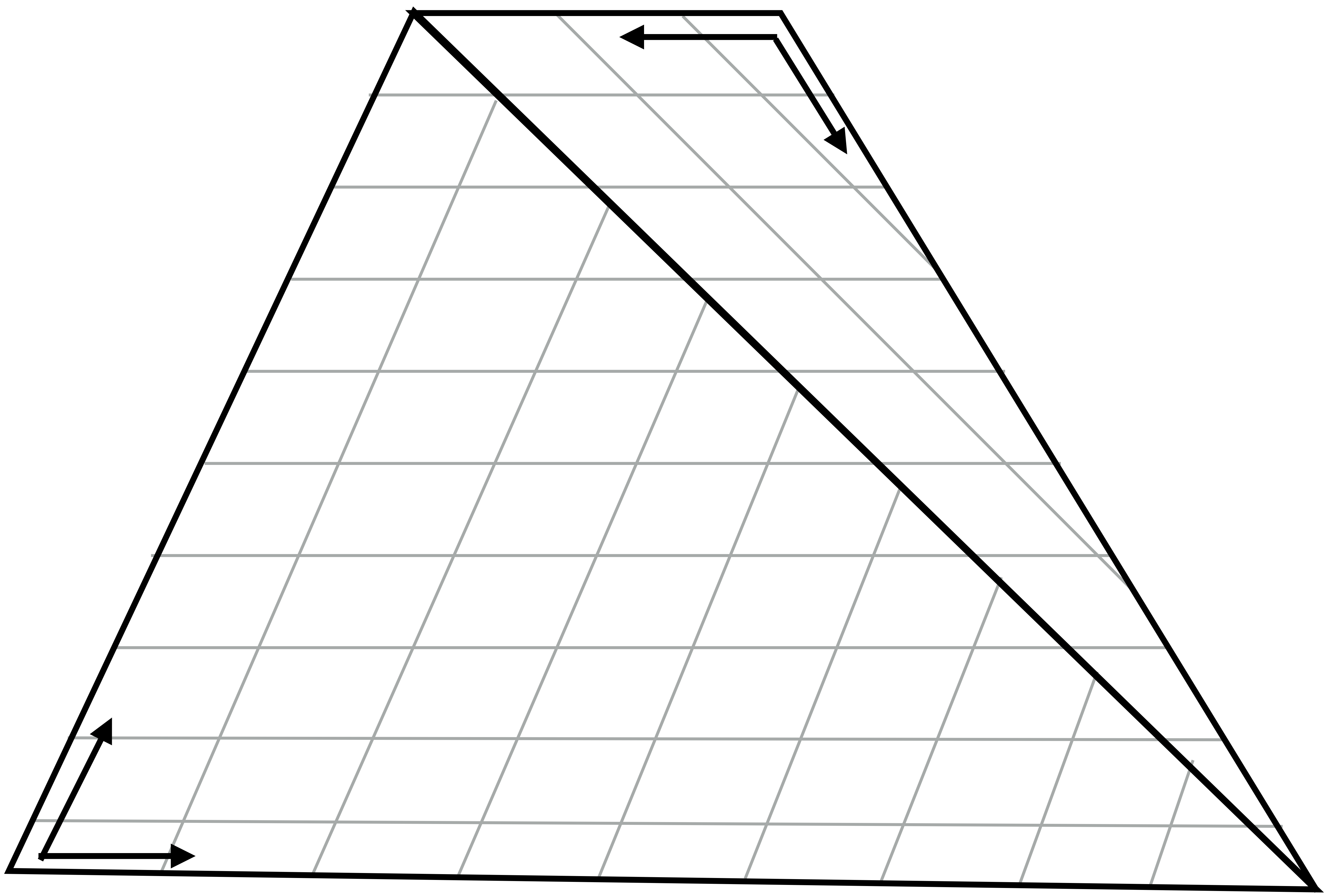
Due to perspective projection (homogeneous divide), barycentric interpolation of values on a triangle with vertices of different depths <u>is not</u> an affine function of screen XY coordinates.

Attribute values must be interpolated linearly in 3D space.



Example: perspective incorrect interpolation

Good example is quadrilateral split into two triangles:



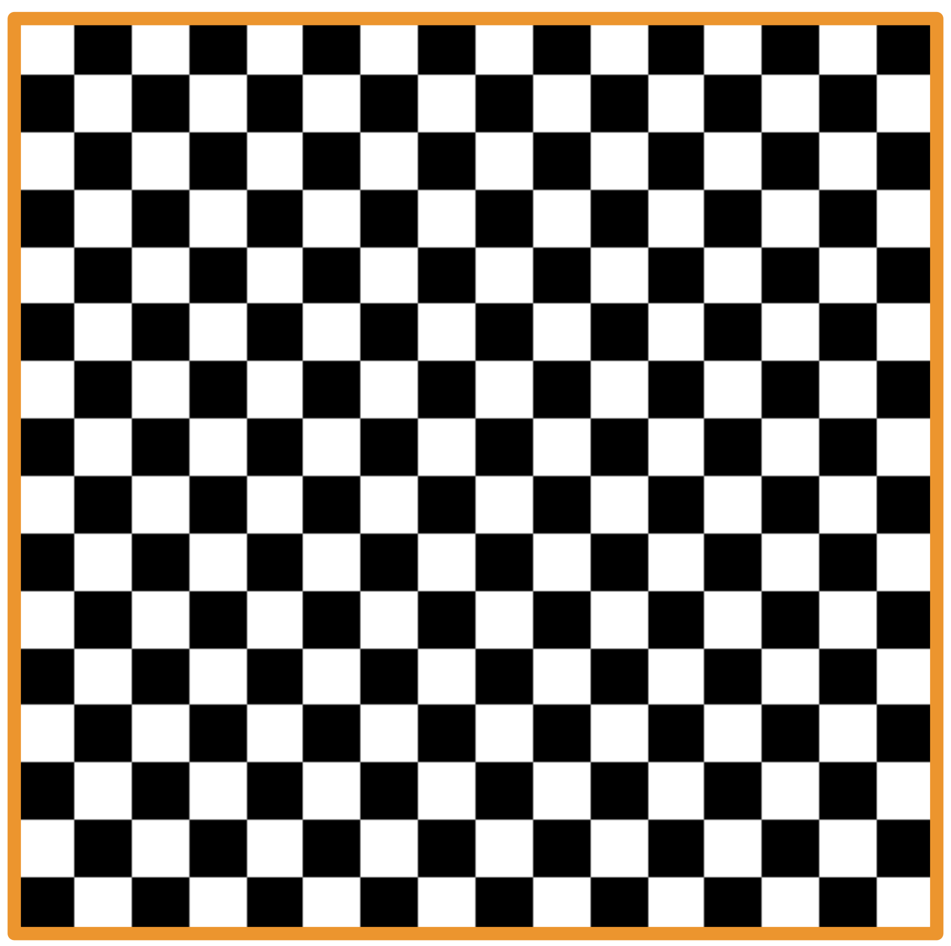
If we compute barycentric coordinates using 2D (projected) vertex positions, can lead to (derivative) discontinuity in interpolation where quad was split.

Perspective correct interpolation

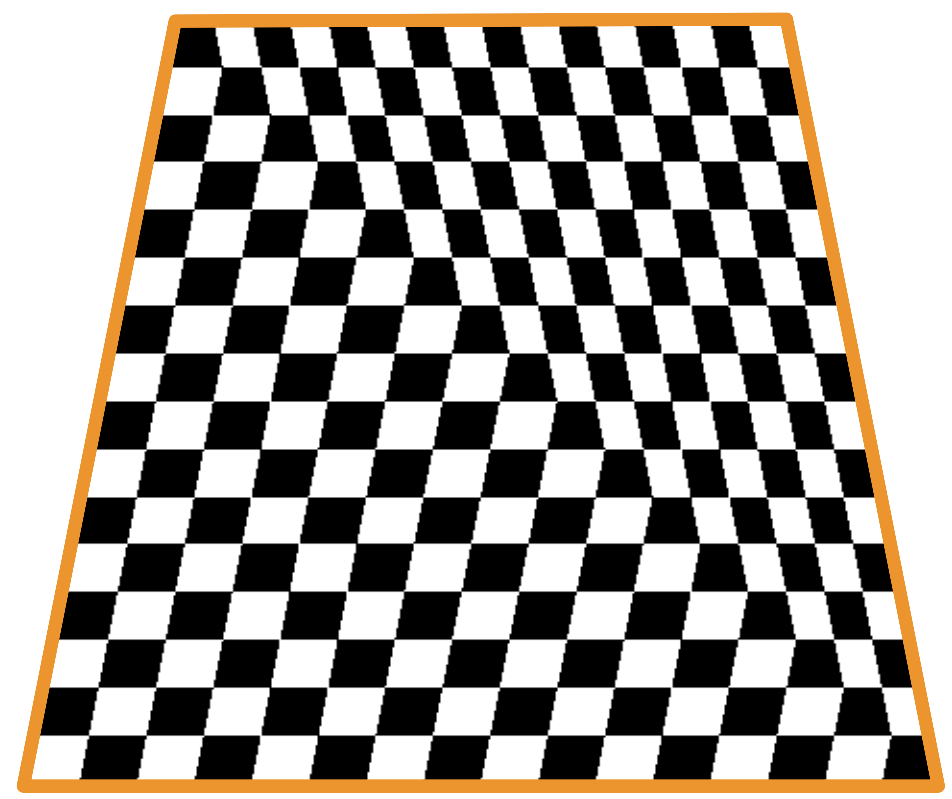
- **Basic recipe:**
  - To interpolate some attribute ϕ…
  - Compute depth z at each vertex
  - Evaluate Z := 1/z and P := ϕ/z at each vertex
  - Interpolate Z and P using standard (2D) barycentric coords
  - At each *fragment*, divide interpolated P by interpolated Z to get final value of ϕ

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

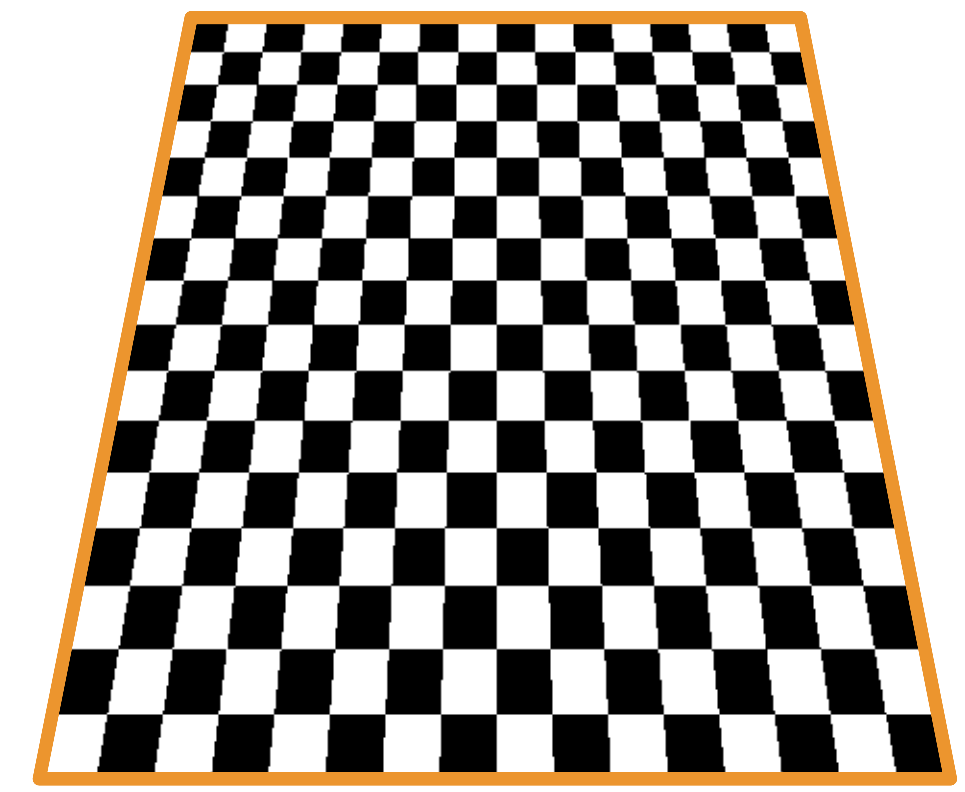
Perspective correct interpolation



Texture



Affine screen-space interpolation



Perspective world-space interpolation

Texture mapping

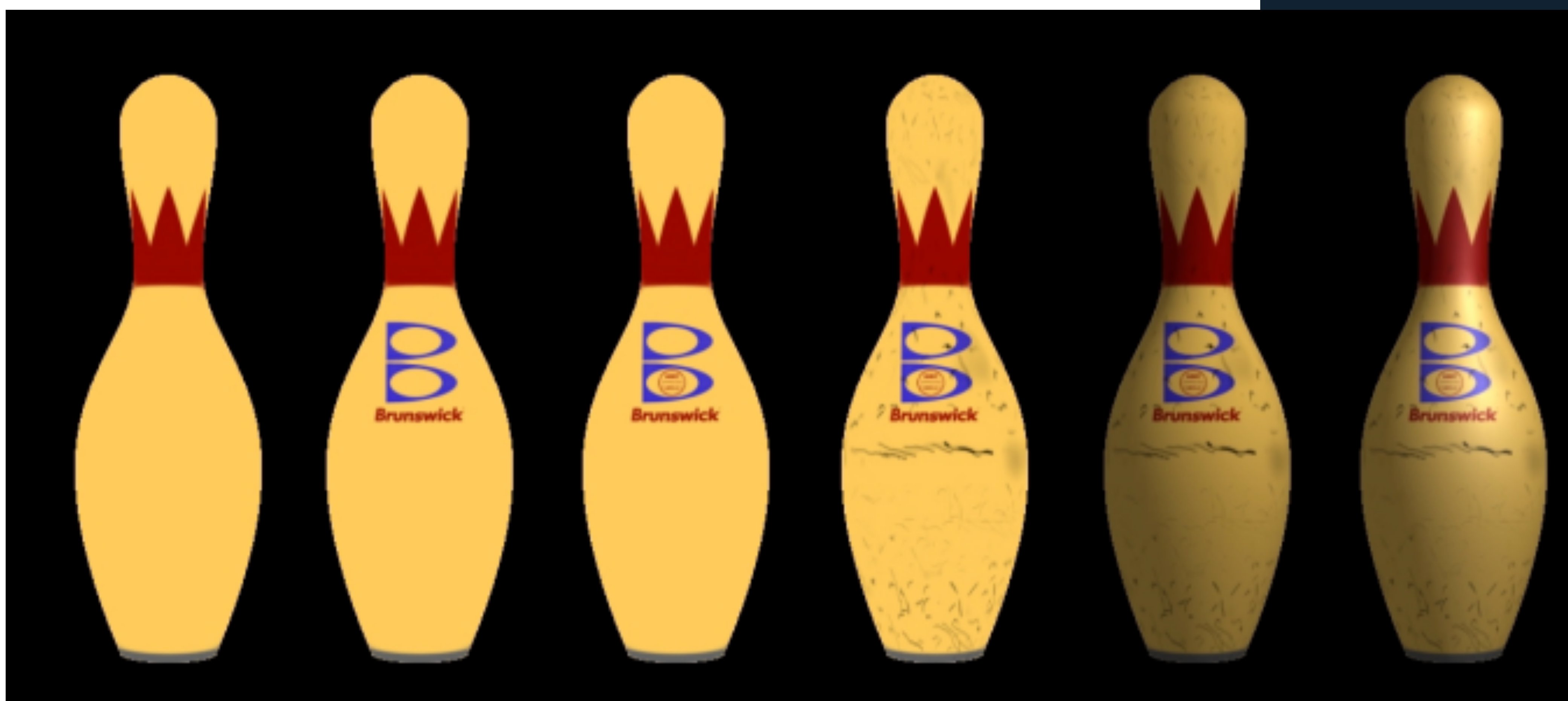
Many uses of texture mapping

Define variation in surface reflectance

Pattern on ball

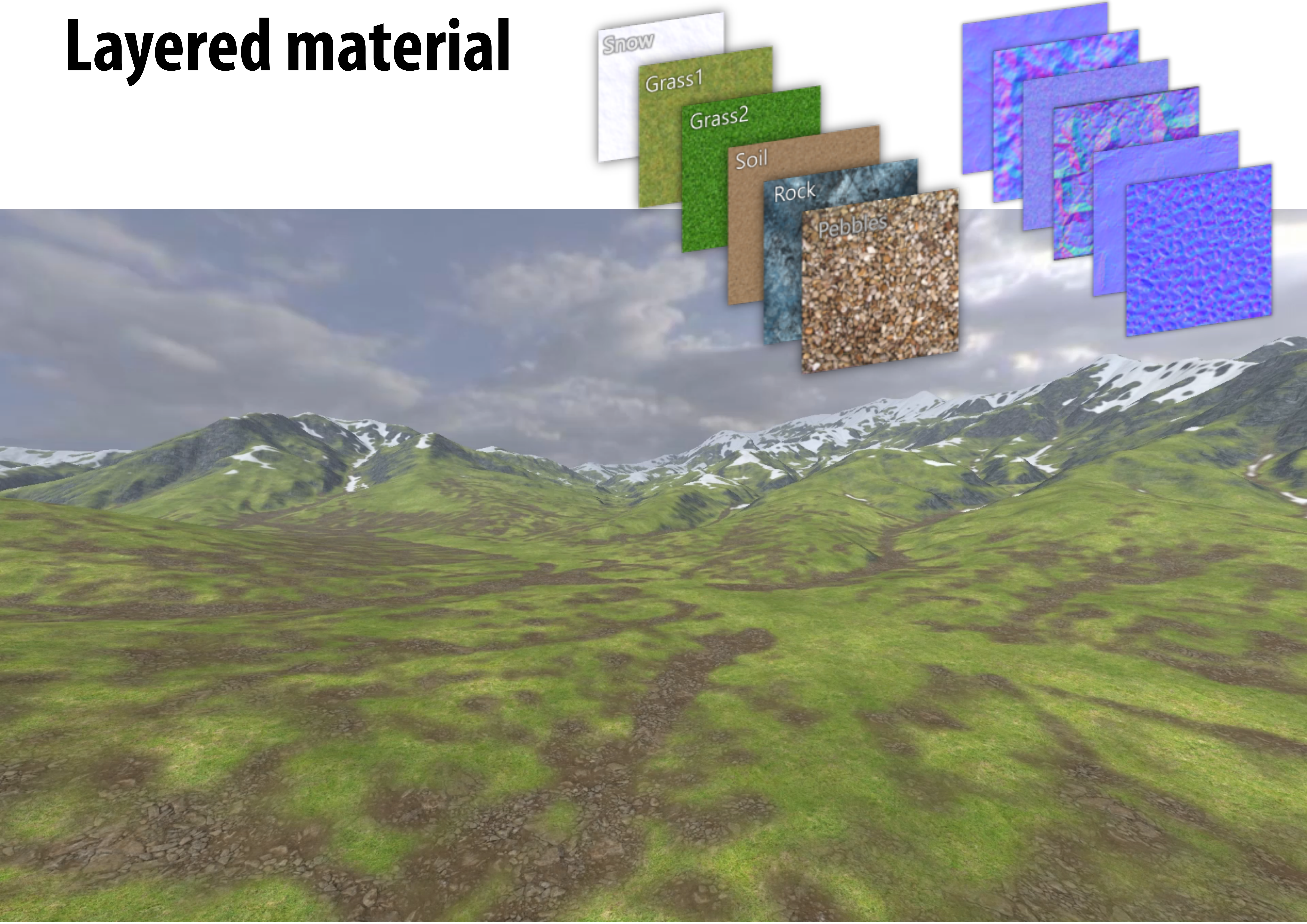
Wood grain on floor

Describe surface material properties



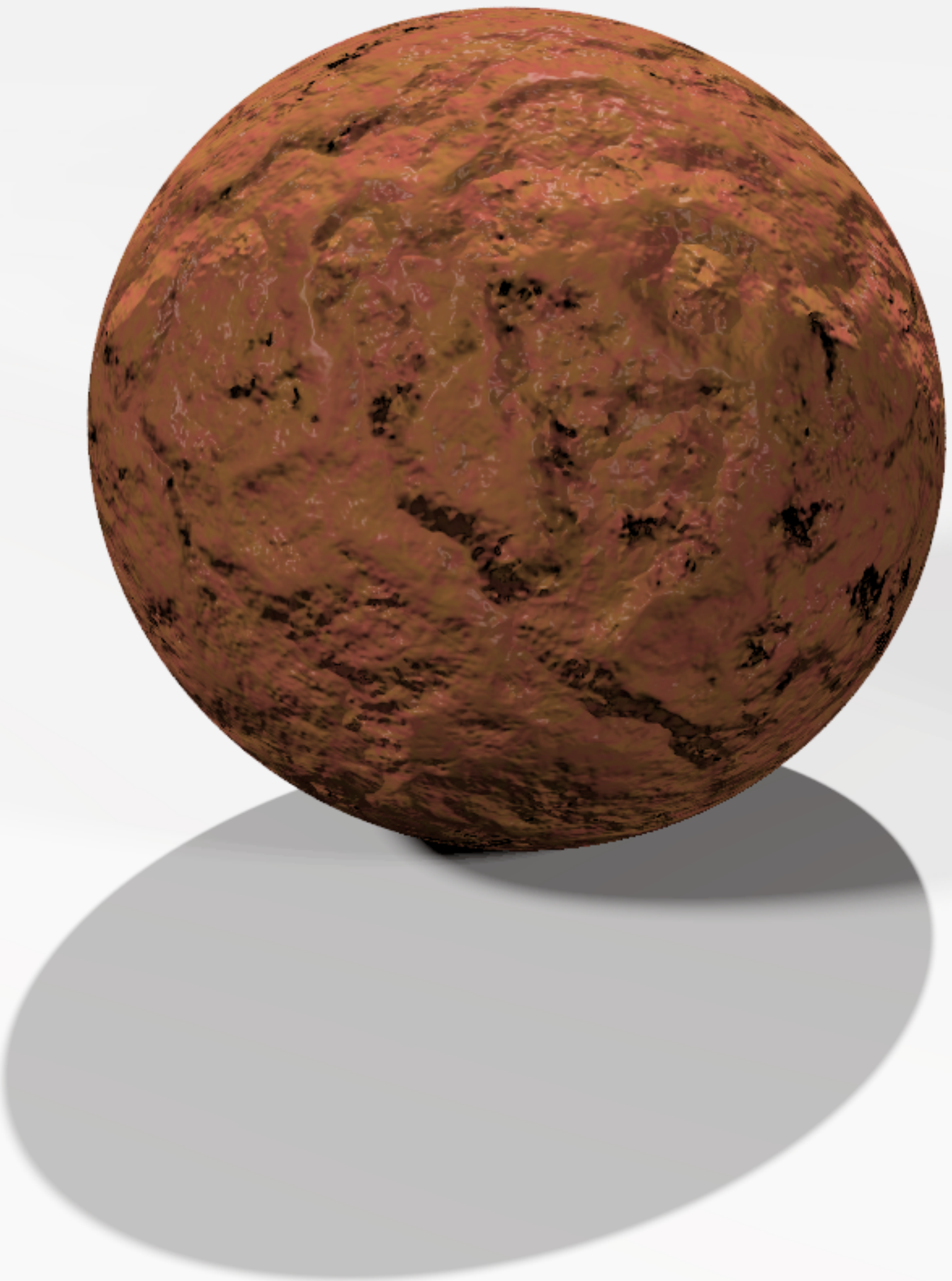
Multiple layers of texture maps for color, logos, scratches, etc.

Layered material



Normal and displacement mapping

normal mapping



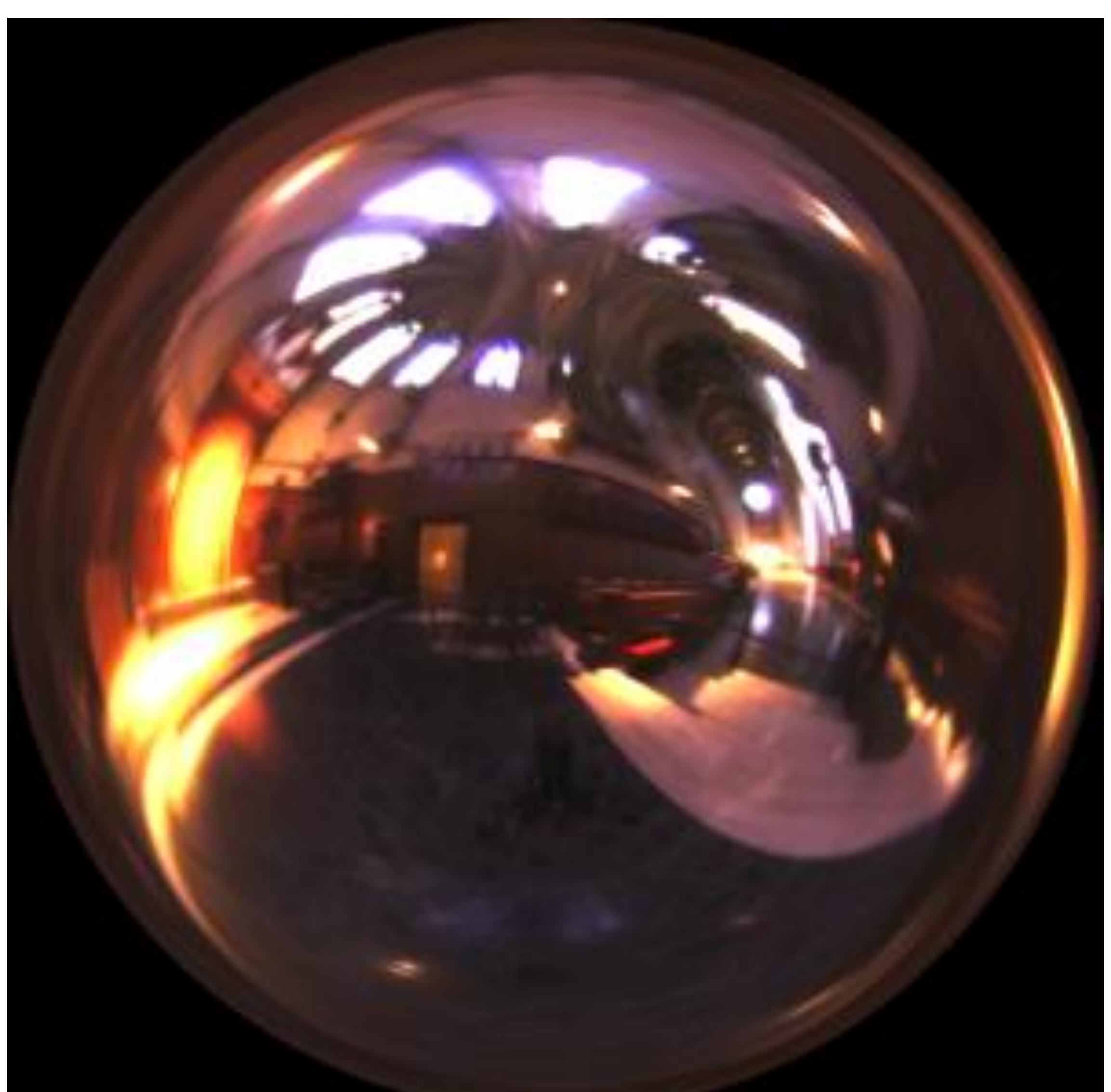
Use texture value to perturb surface normal to "fake" appearance of a bumpy surface (note smooth silhouette/shadow reveals that surface geometry is not actually bumpy!)

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

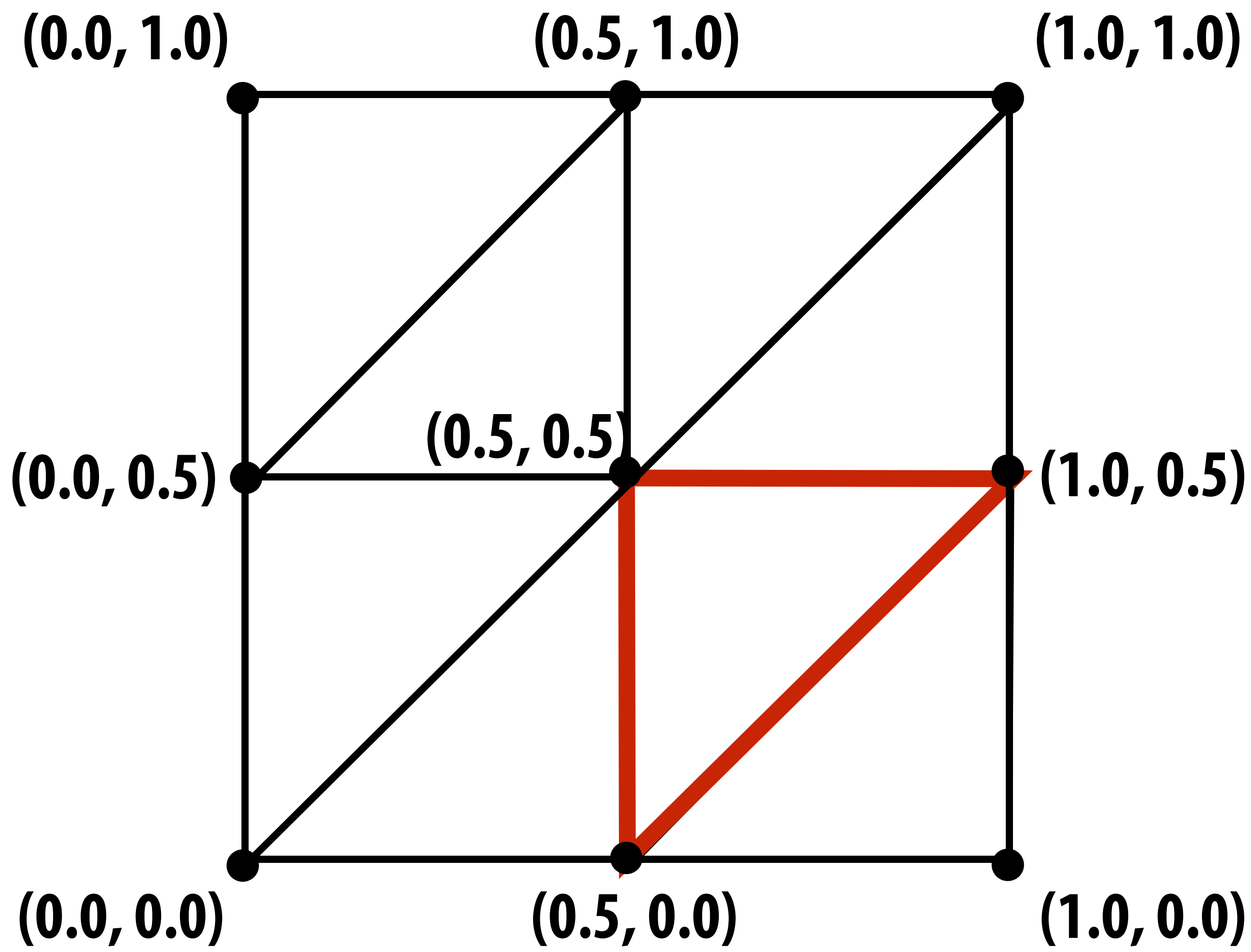


Grace Cathedral environment map

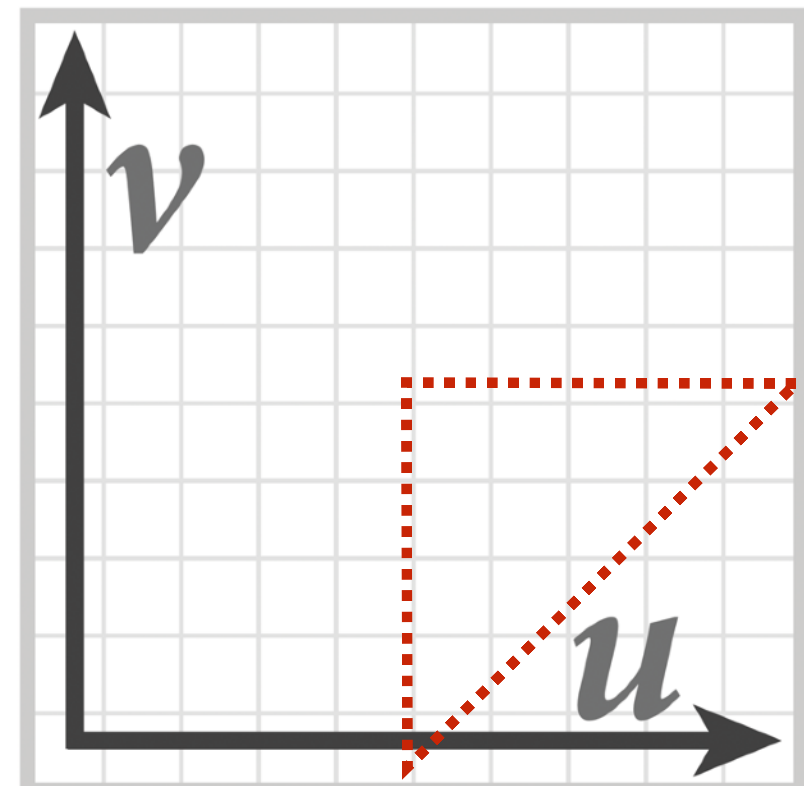
Environment map used in rendering

Texture coordinates

"Texture coordinates" define a mapping from surface coordinates (points on triangle) to points in texture domain.

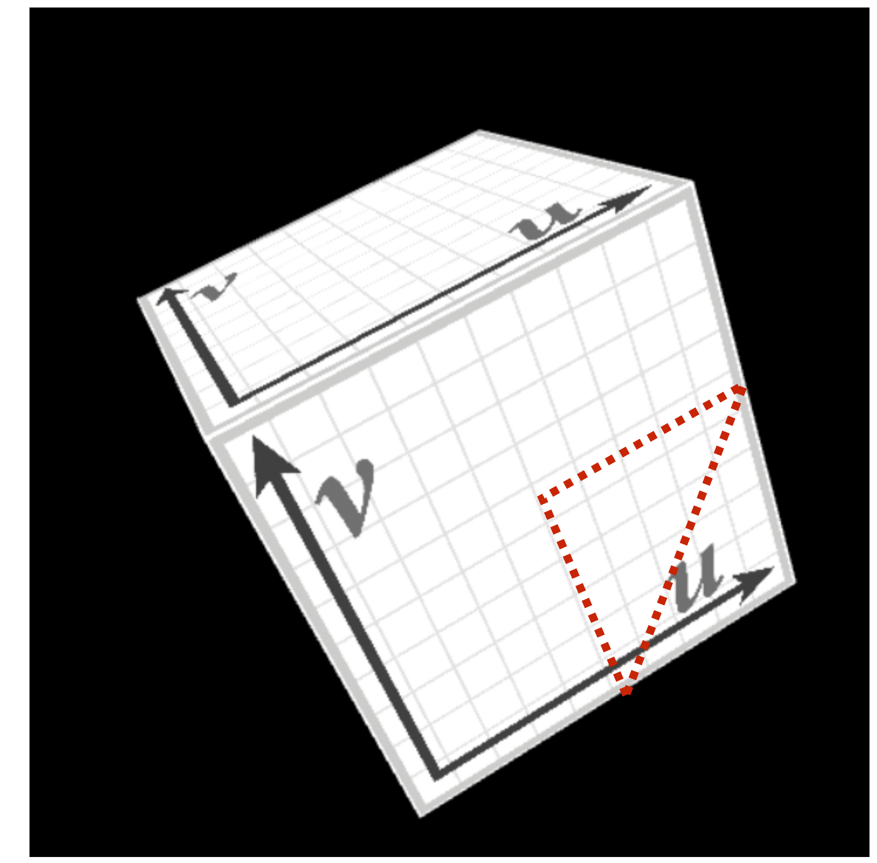


Eight triangles (one face of cube) with surface parameterization provided as per-vertex texture coordinates.



`myTex(u,v)` is a function defined on the $[0,1]^2$ domain (represented by 2048x2048 image)

Location of highlighted triangle in texture space shown in red.



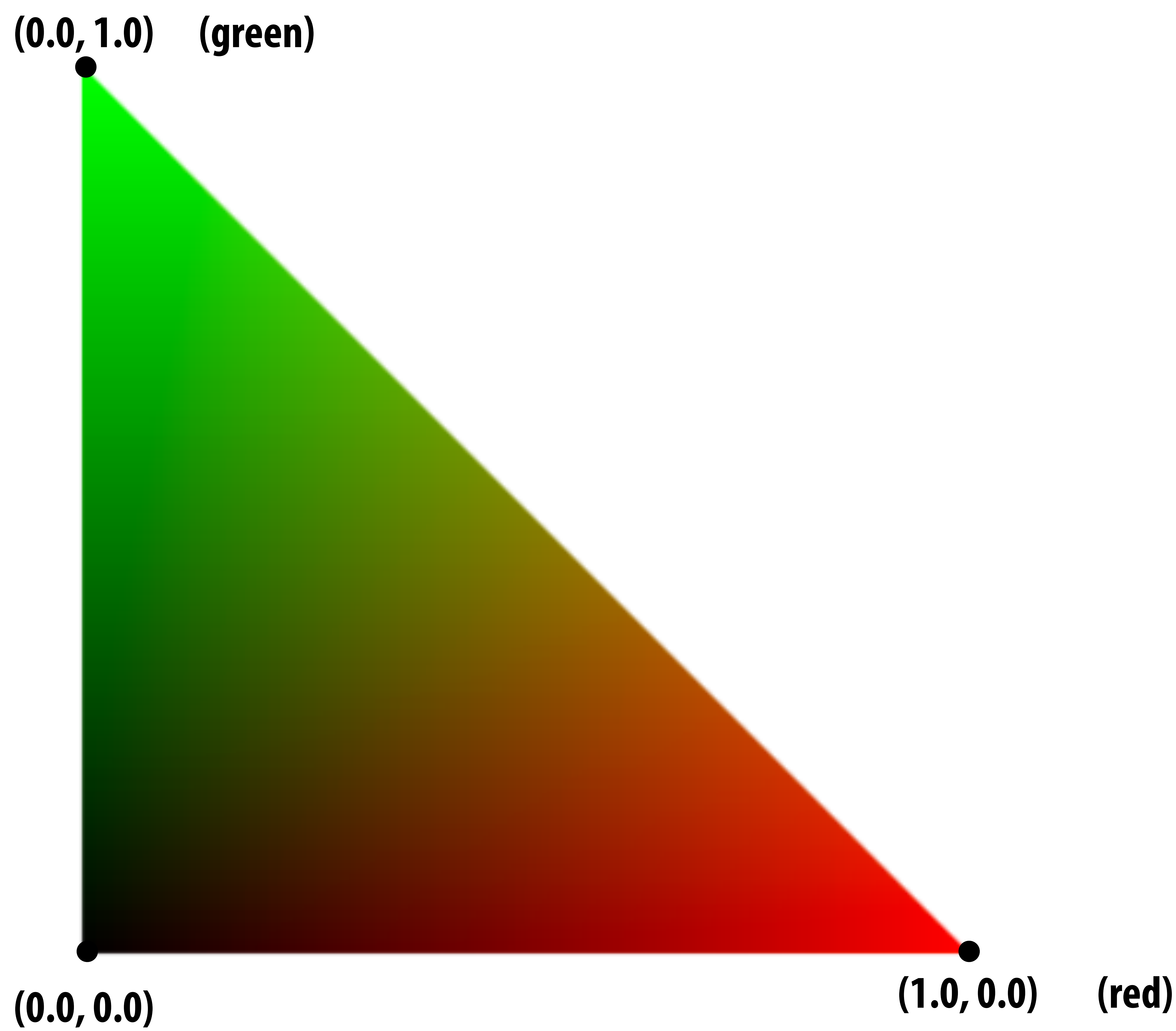
Final rendered result (entire cube shown).

Location of triangle after projection onto screen shown in red.

Today we'll assume surface-to-texture space mapping is provided as per vertex attribute
(Not discussing methods for generating surface texture parameterizations)

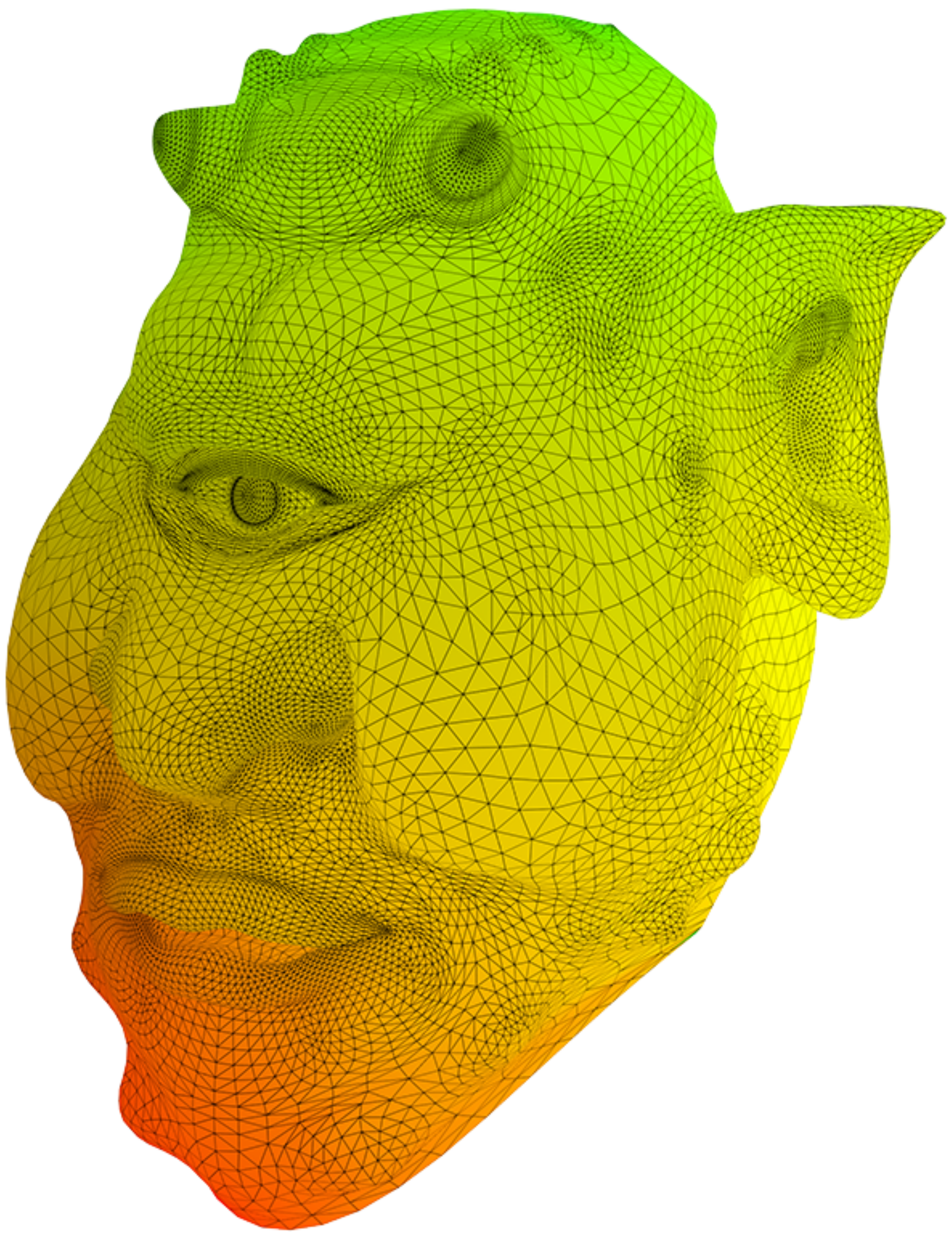
Visualization of texture coordinates

Texture coordinates linearly interpolated over triangle

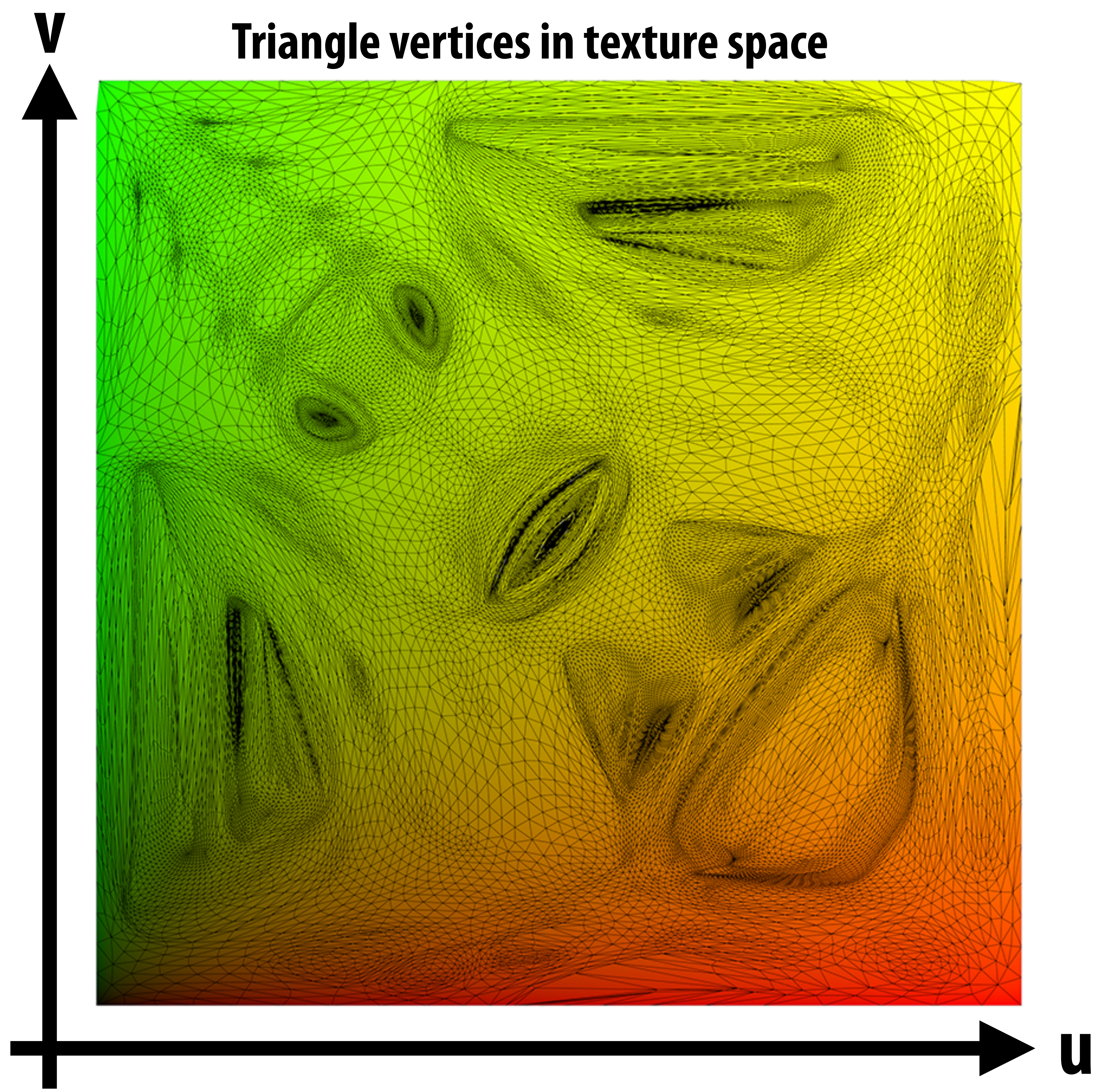


More complex mapping

Visualization of texture coordinates



Triangle vertices in texture space

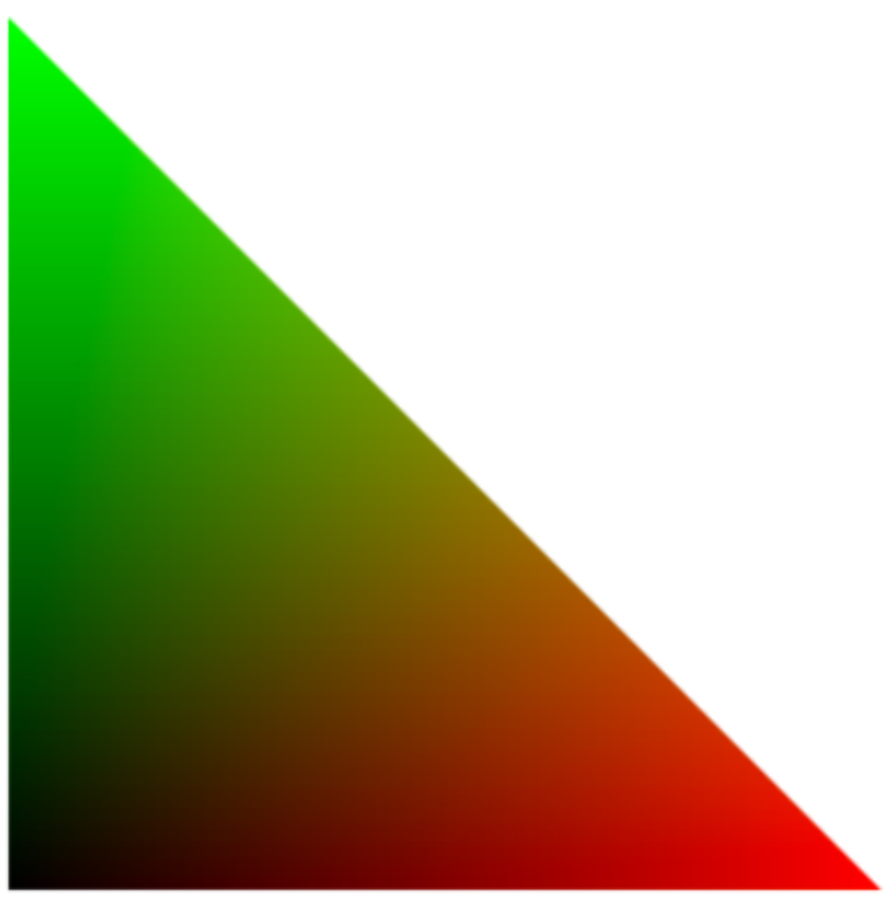


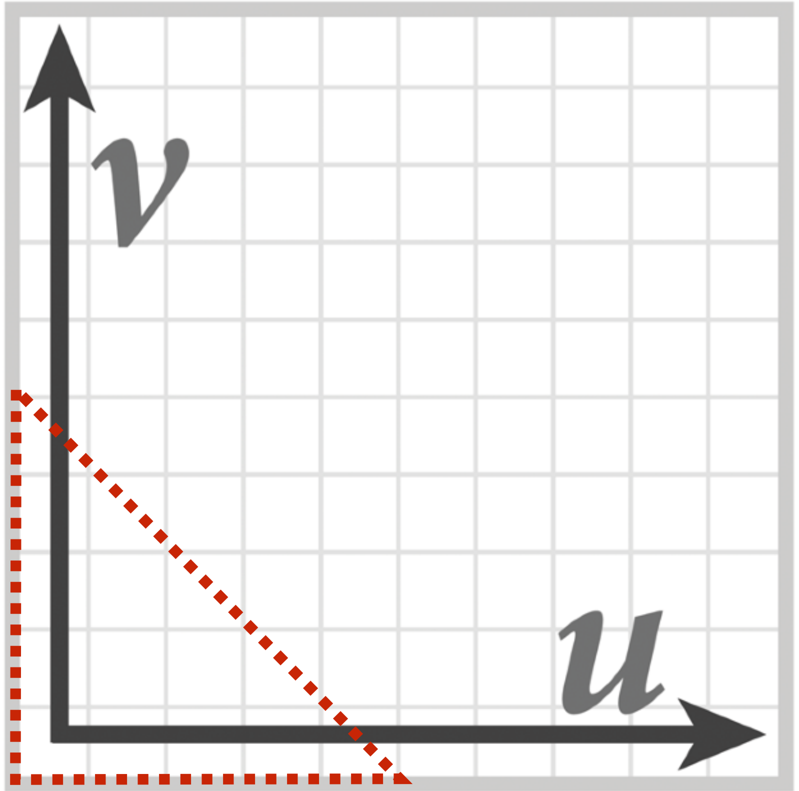
Each vertex has a coordinate (u,v) in texture space.

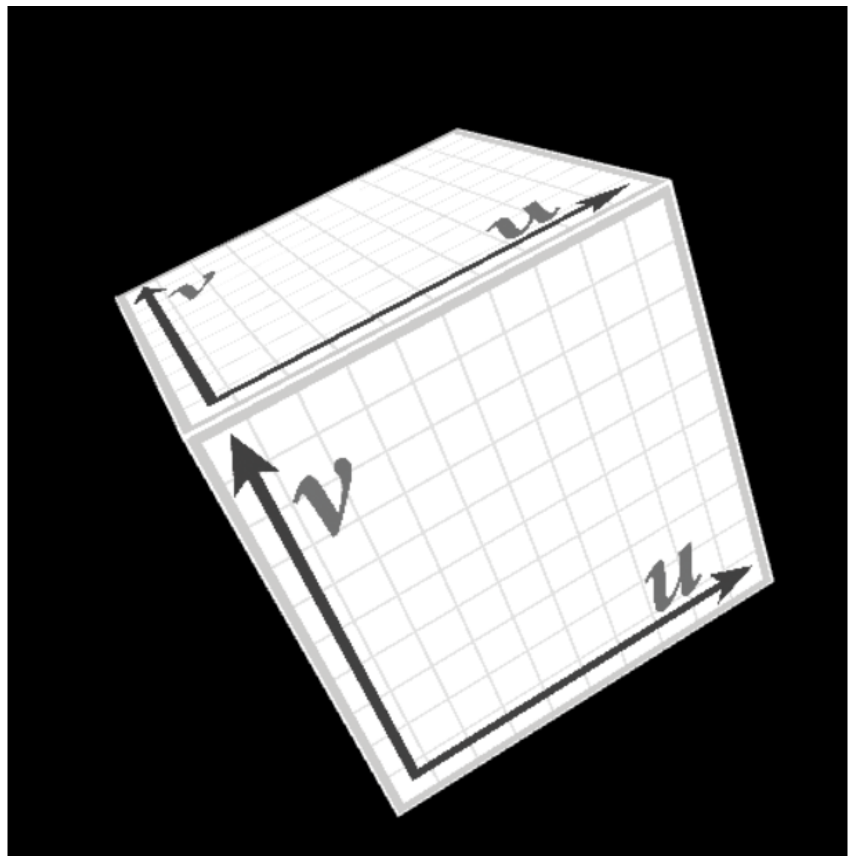
(Actually coming up with these coordinates is another story!)

Texture sampling 101

- Basic algorithm for mapping texture to surface:
  - For each color sample location (X,Y)
    - Interpolate U and V coordinates across triangle to get value at (X,Y)
    - Sample (evaluate) texture at (U,V)
    - Set color of fragment to sampled texture value

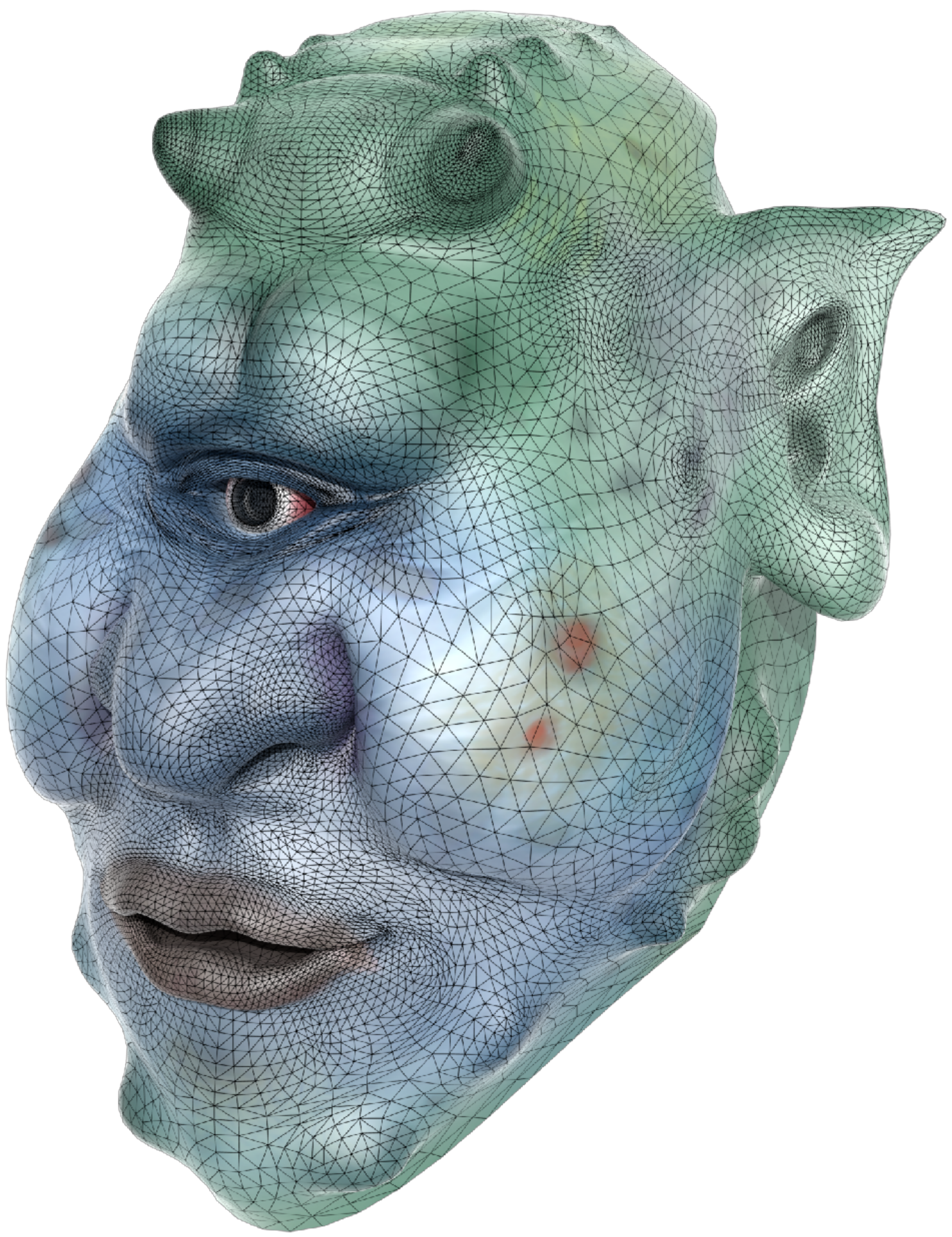




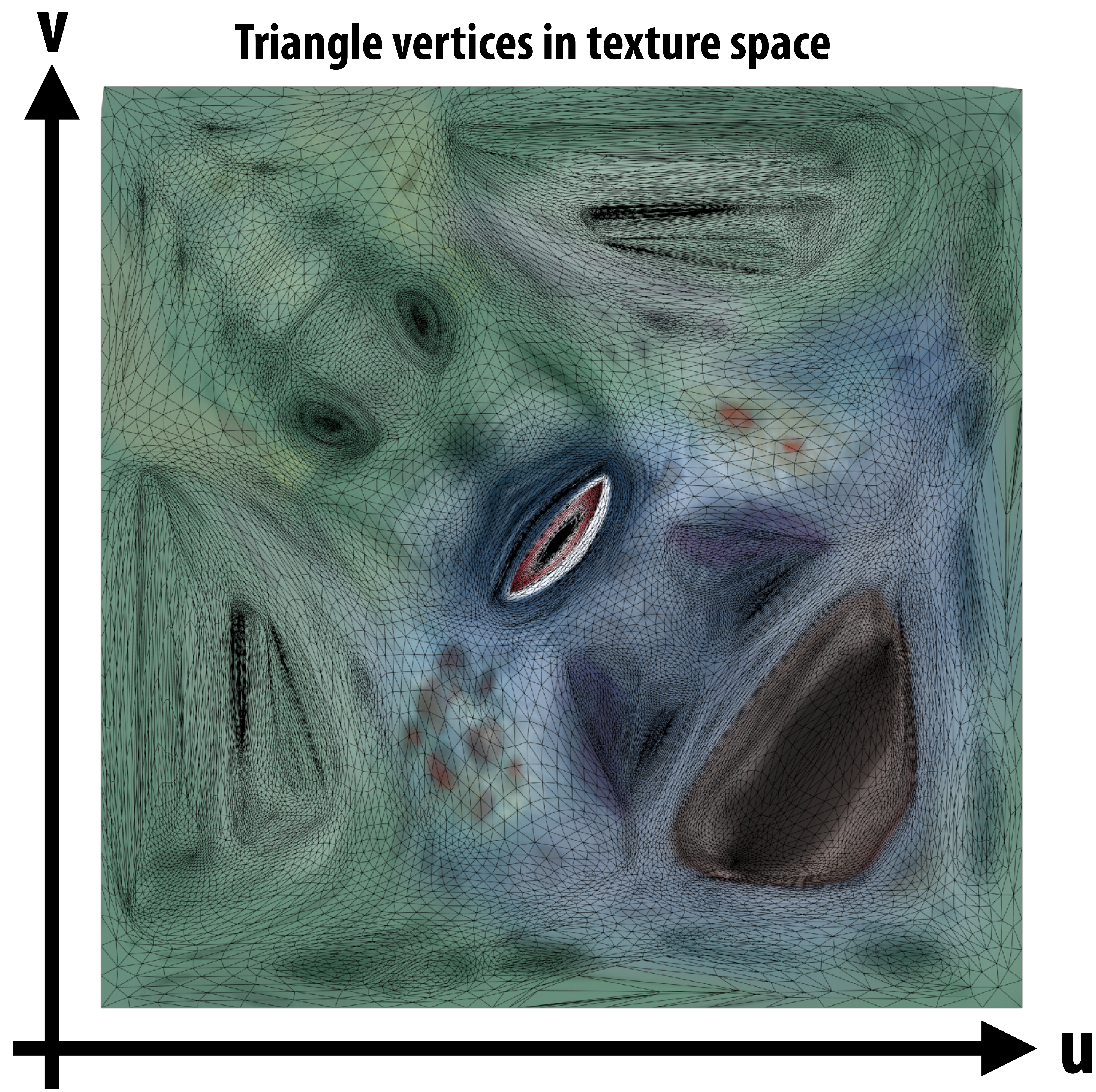


Texture mapping adds detail

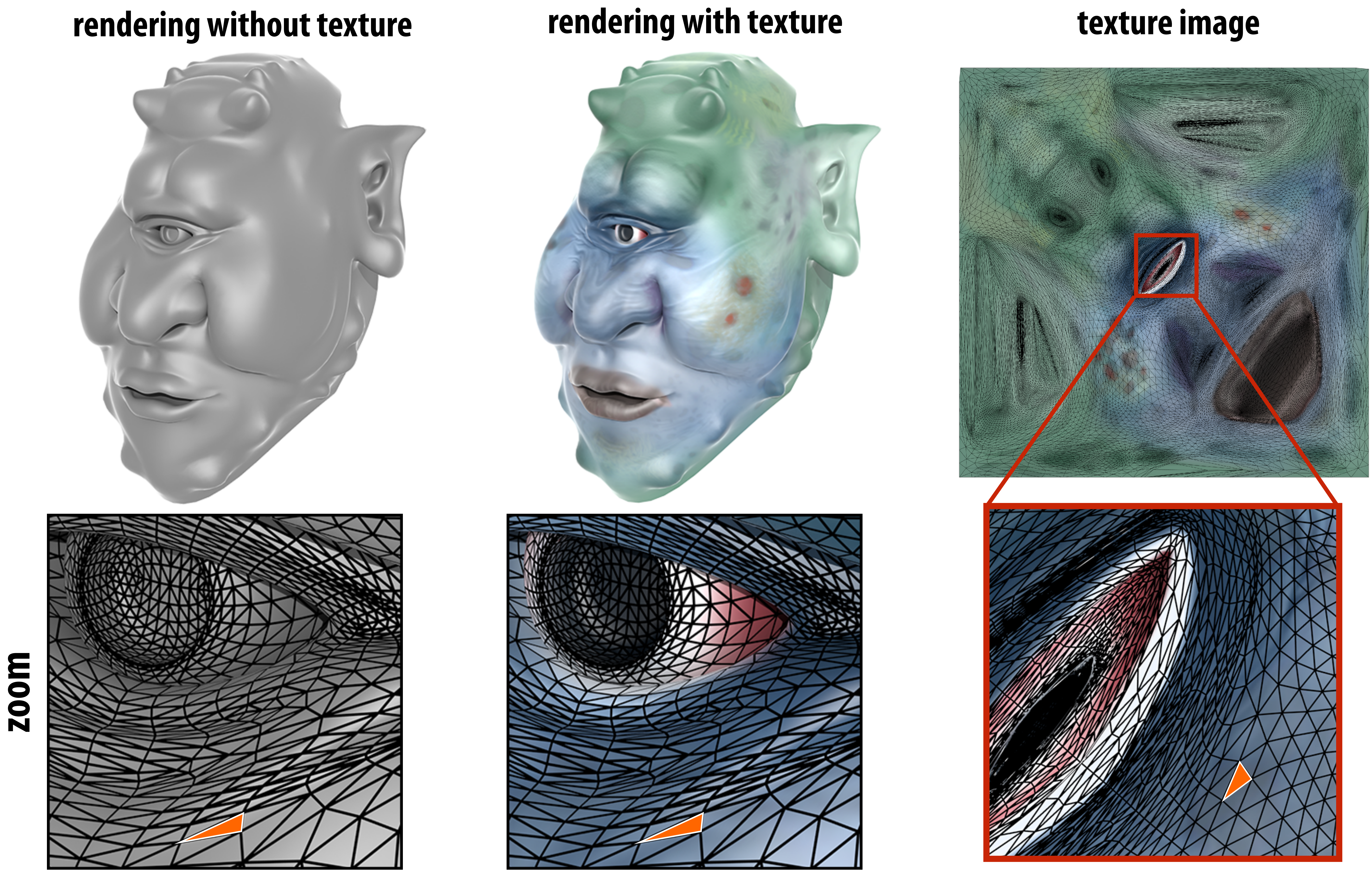
Rendered result



Triangle vertices in texture space

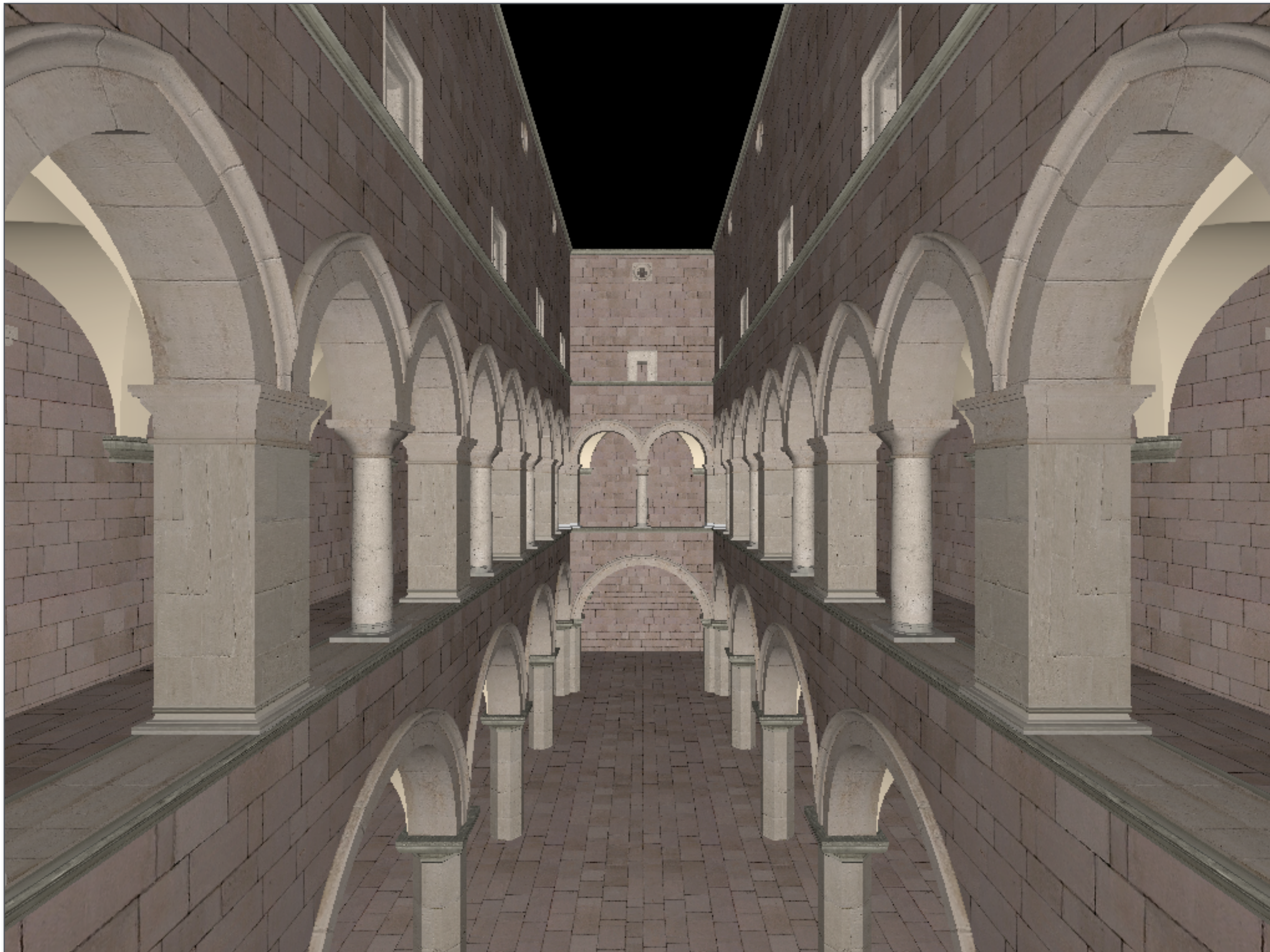


Texture mapping adds detail

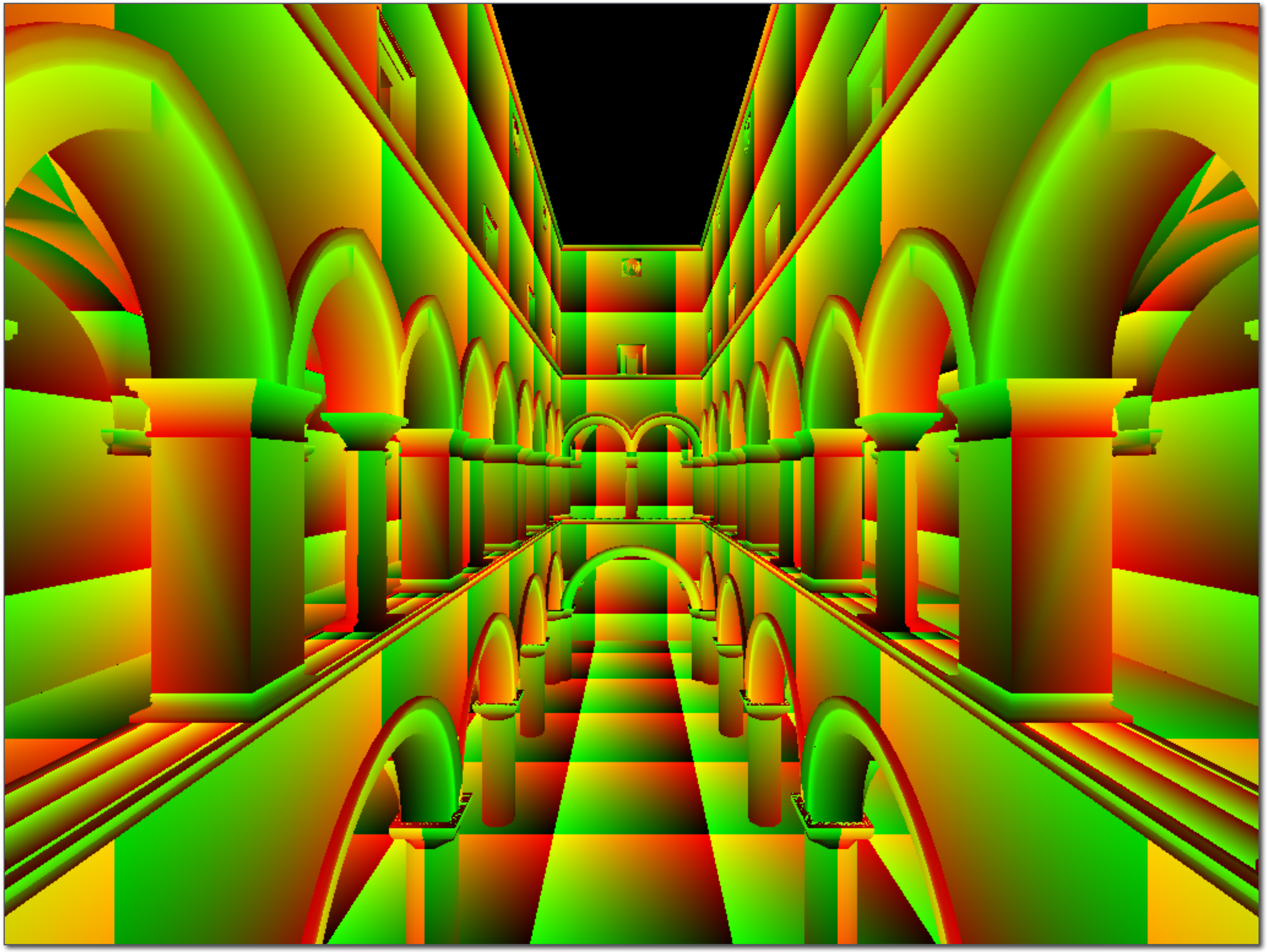


Each triangle "copies" a piece of the image back to the surface.

Textured Sponza

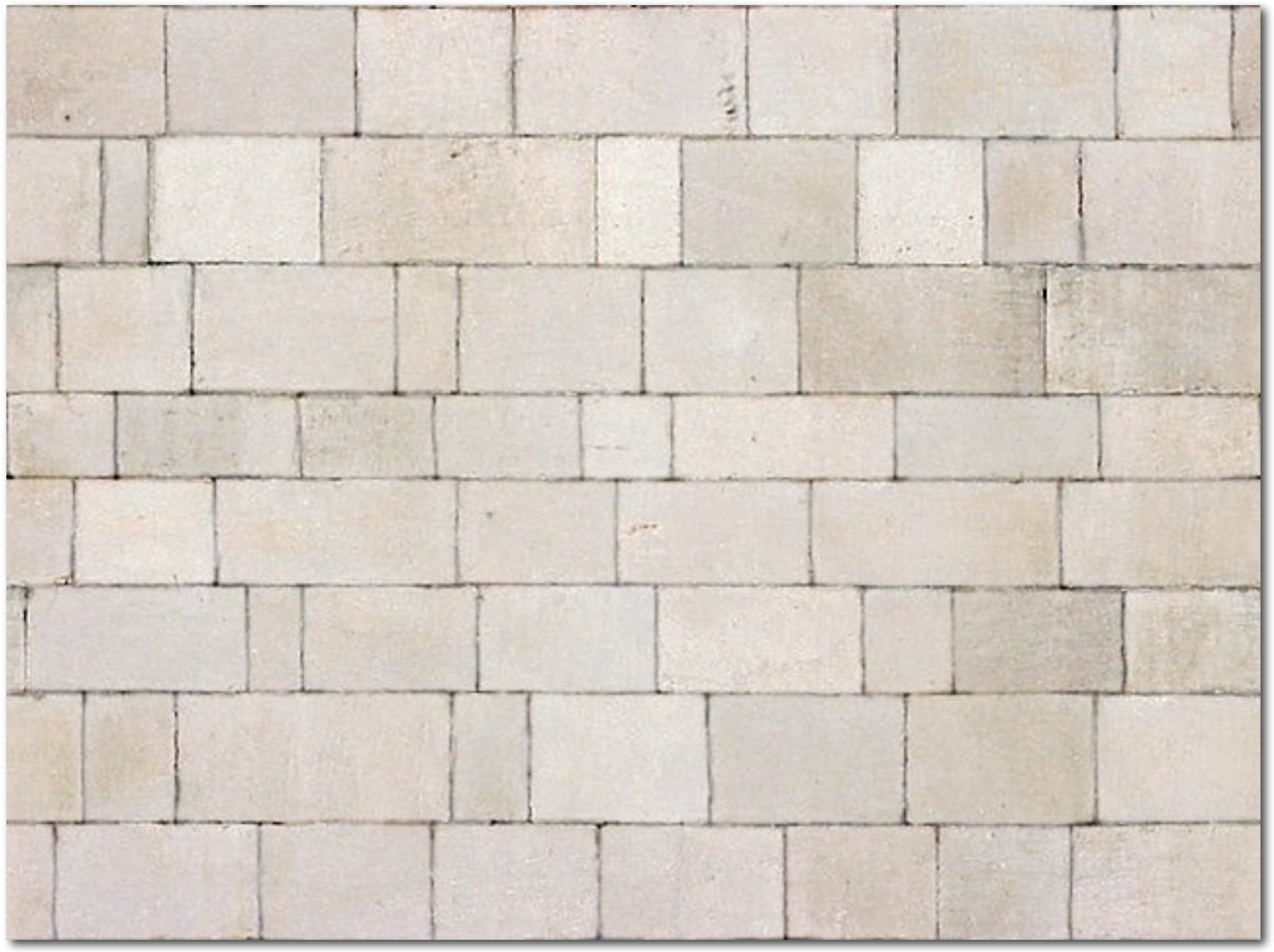


Another example: Sponza



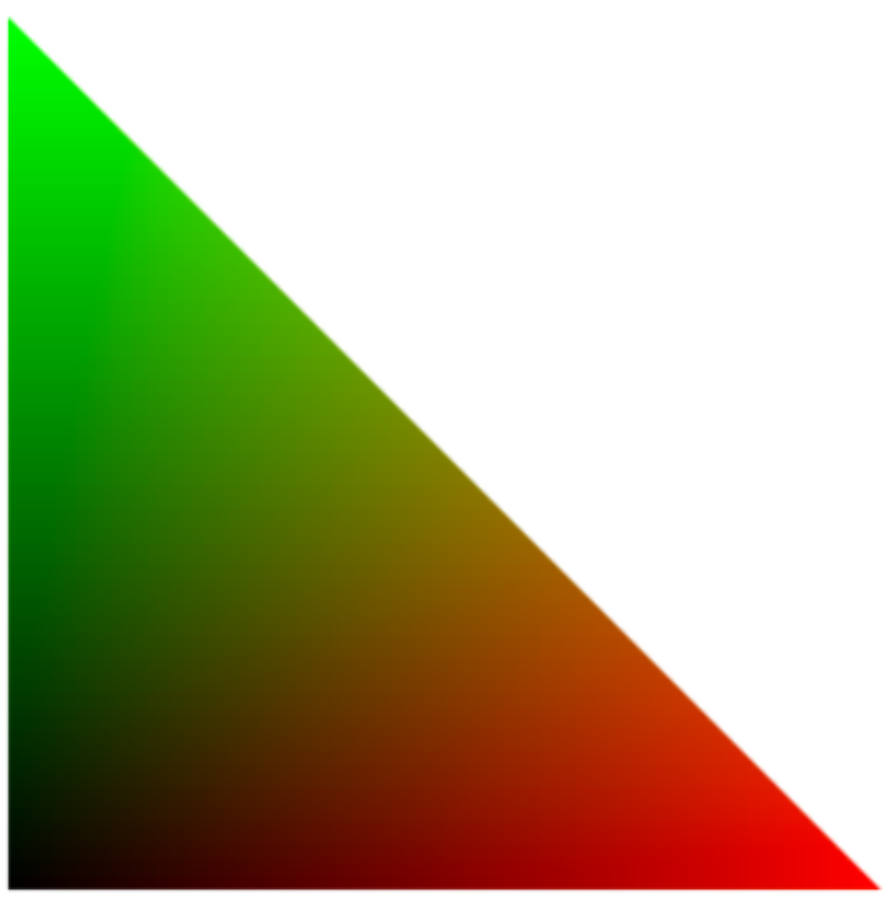
Notice texture coordinates repeat over surface.

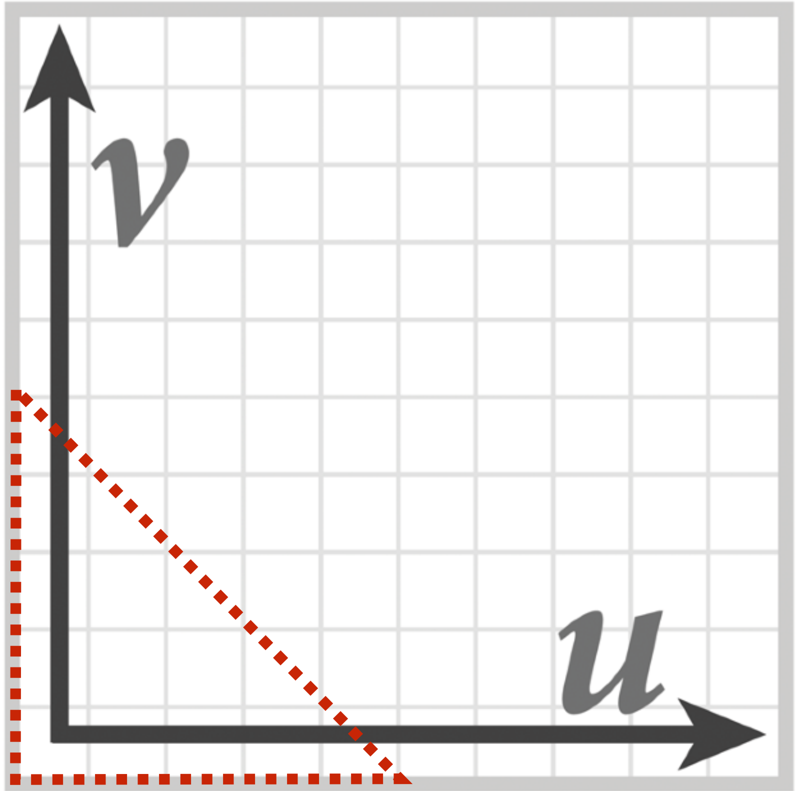
Example textures used in Sponza

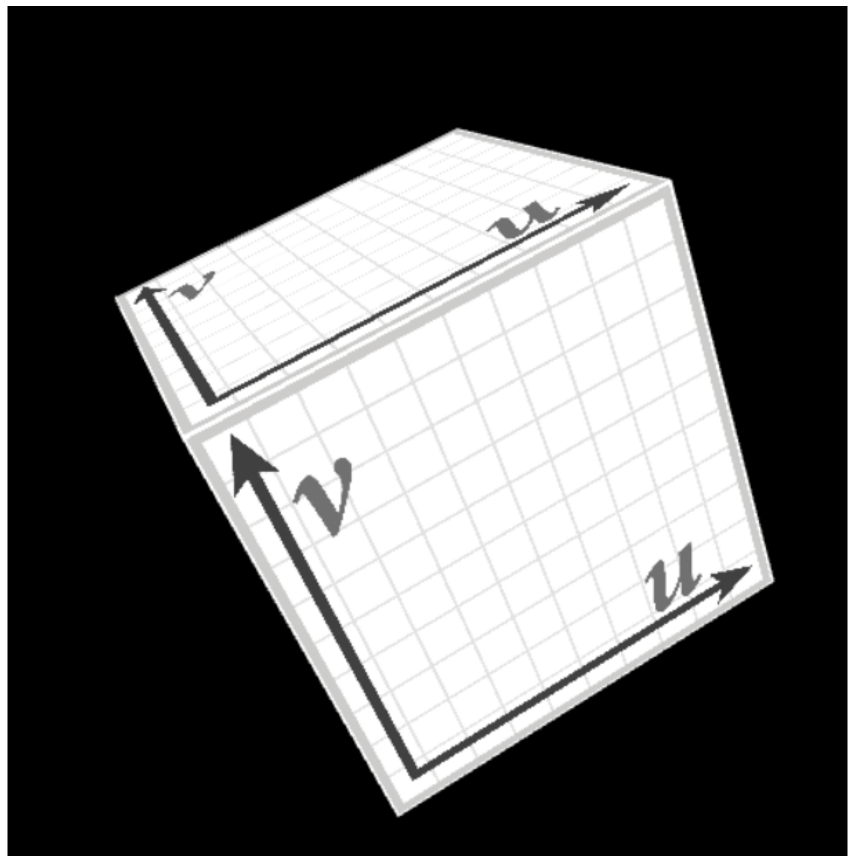


Summary

- Basic algorithm for mapping texture to surface:
  - For each color sample location (X,Y)
    - Interpolate U and V coordinates across triangle to get value at (X,Y)
    - Sample (evaluate) texture at (U,V)
    - Set color of fragment to sampled texture value



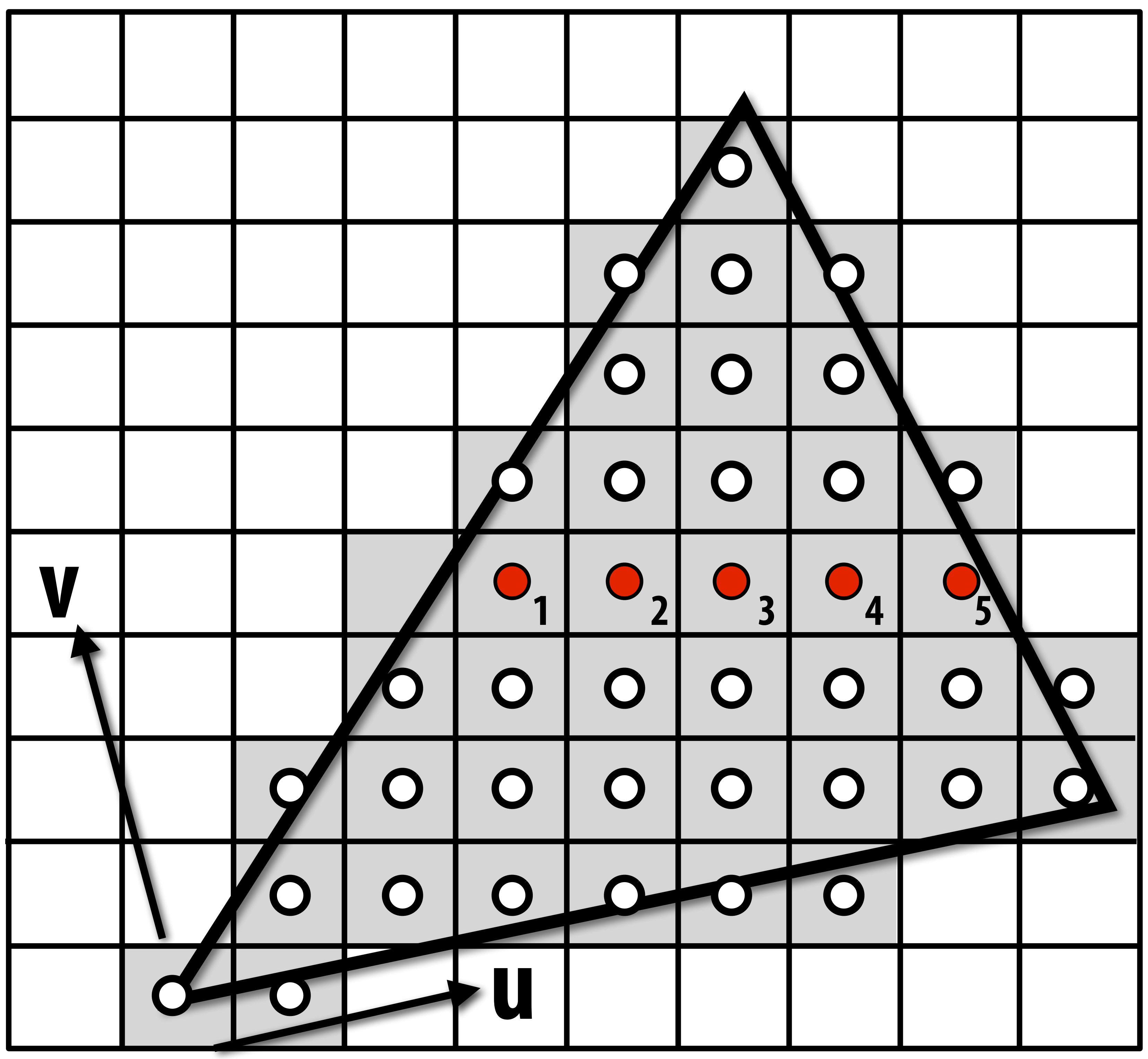




. . .sadly not this easy in general!

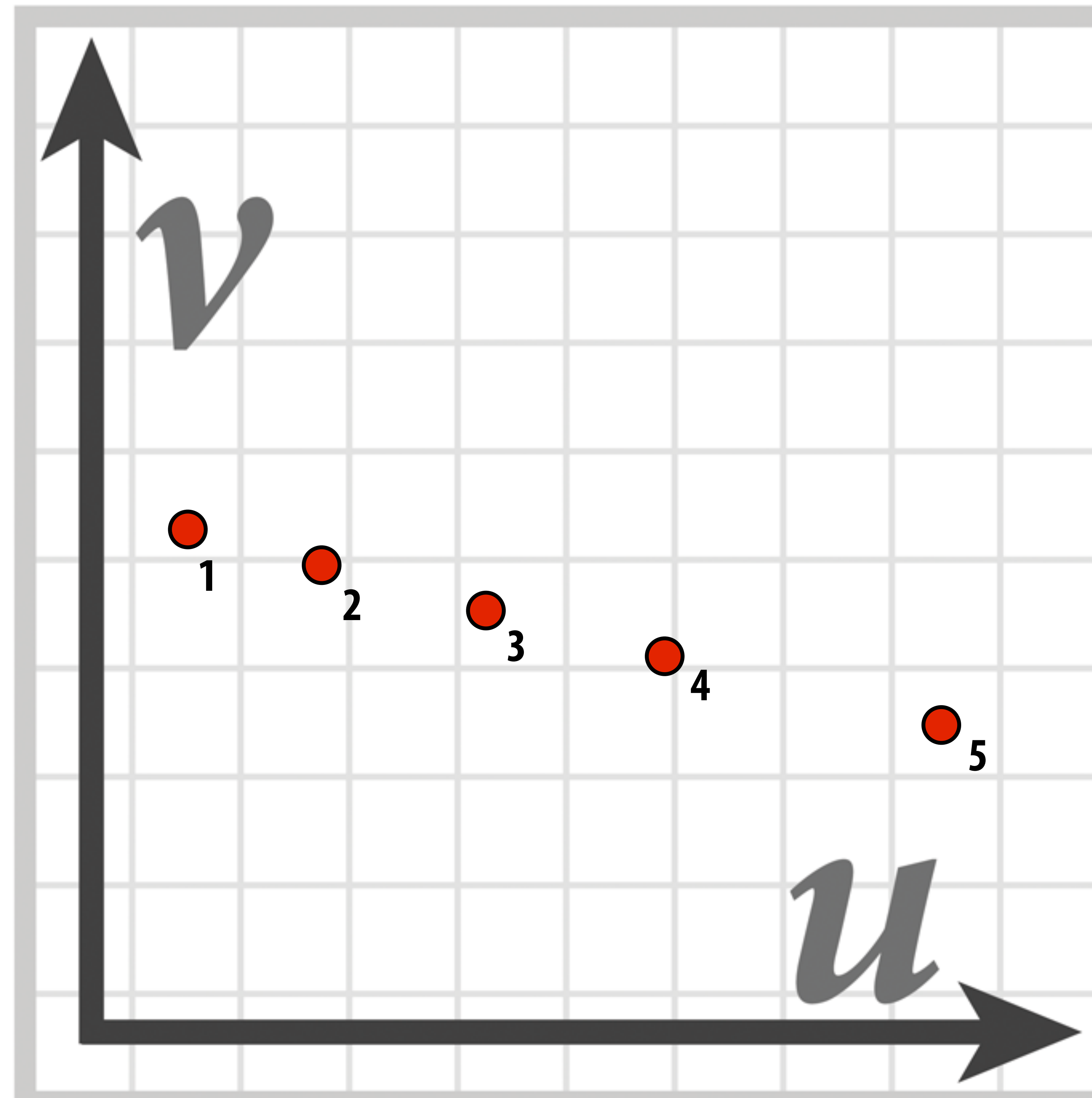
Texture space samples

Sample positions in XY screen space



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

Sample positions in texture space



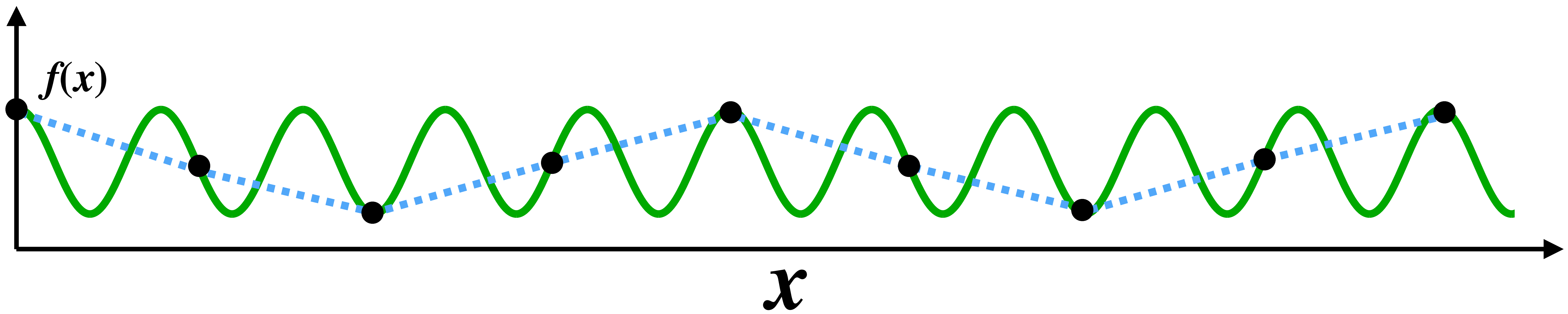
Texture sample positions in texture space (texture function is sampled at these locations)

Applying textures is a form of sampling!

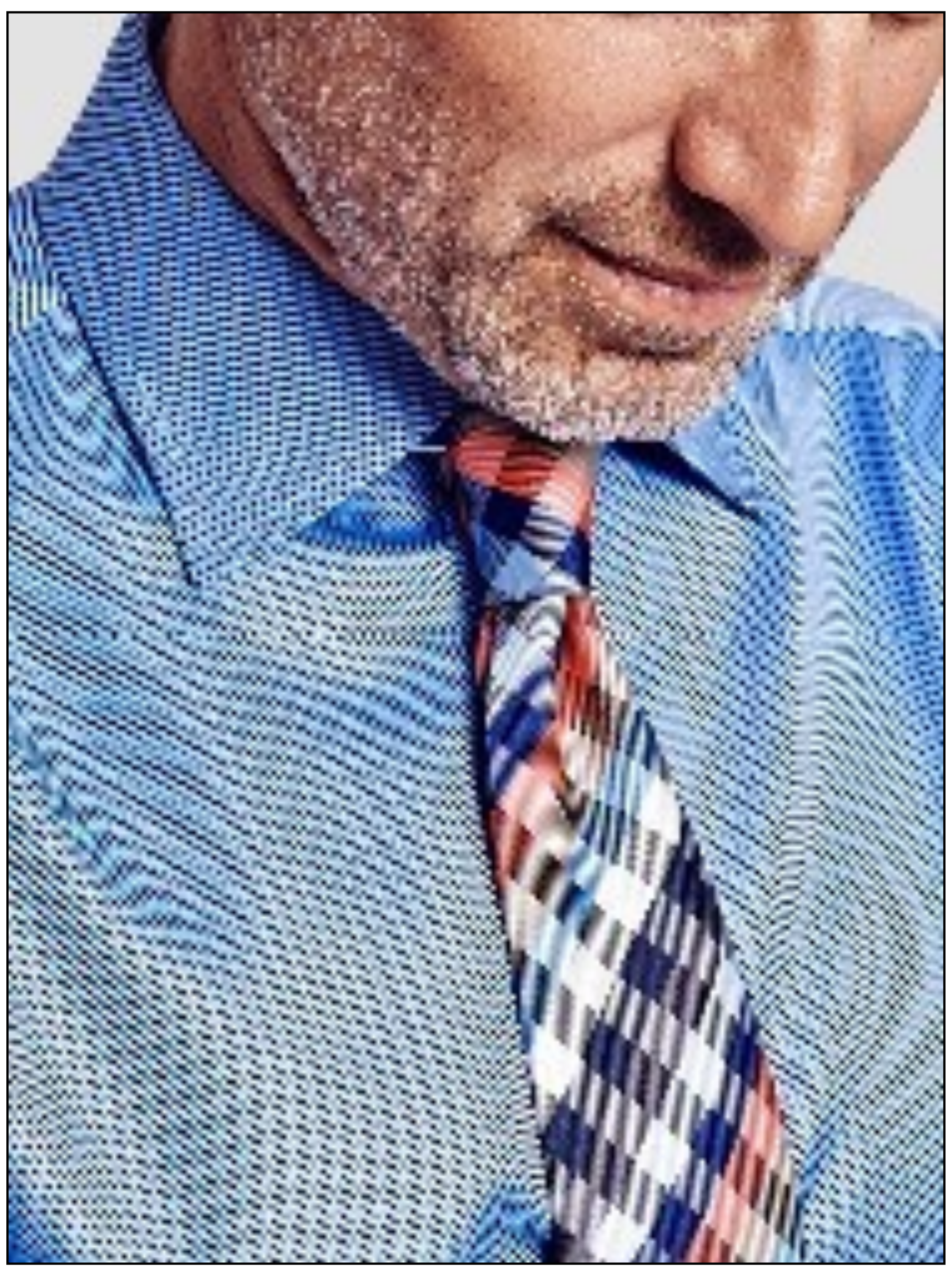
t(u,v)

Recall: aliasing

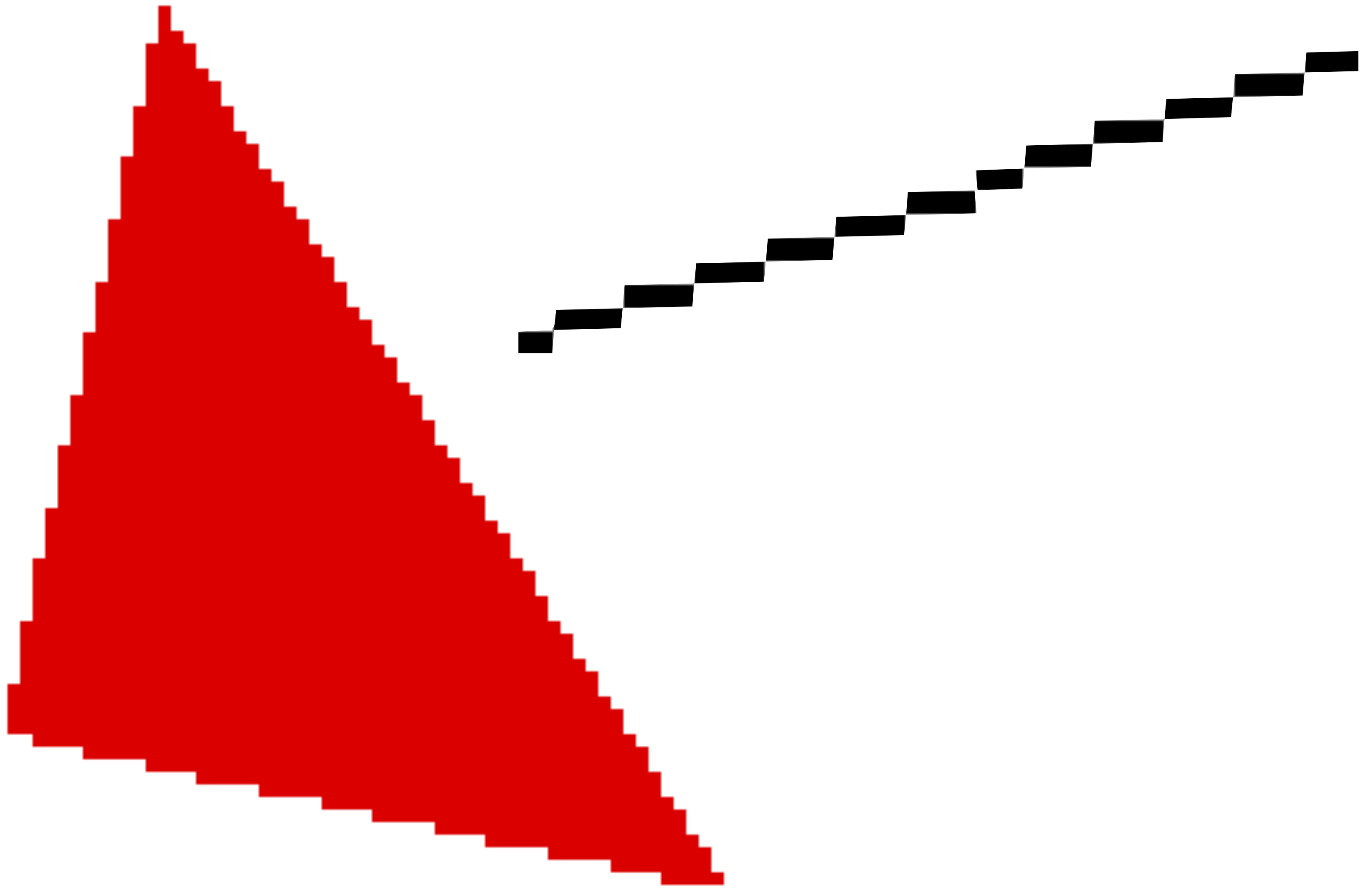
Undersampling a high-frequency signal can result in aliasing



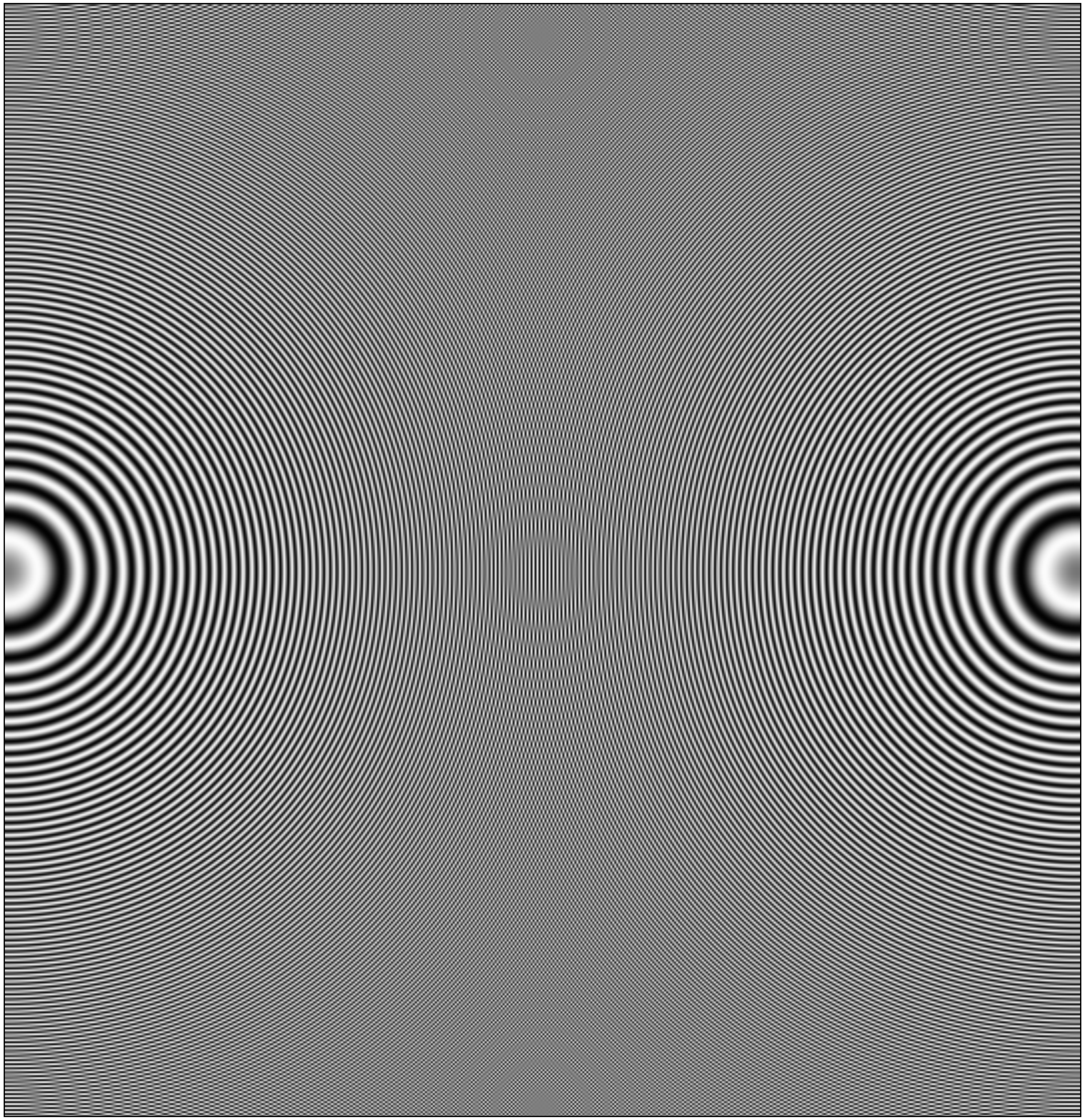
1D example



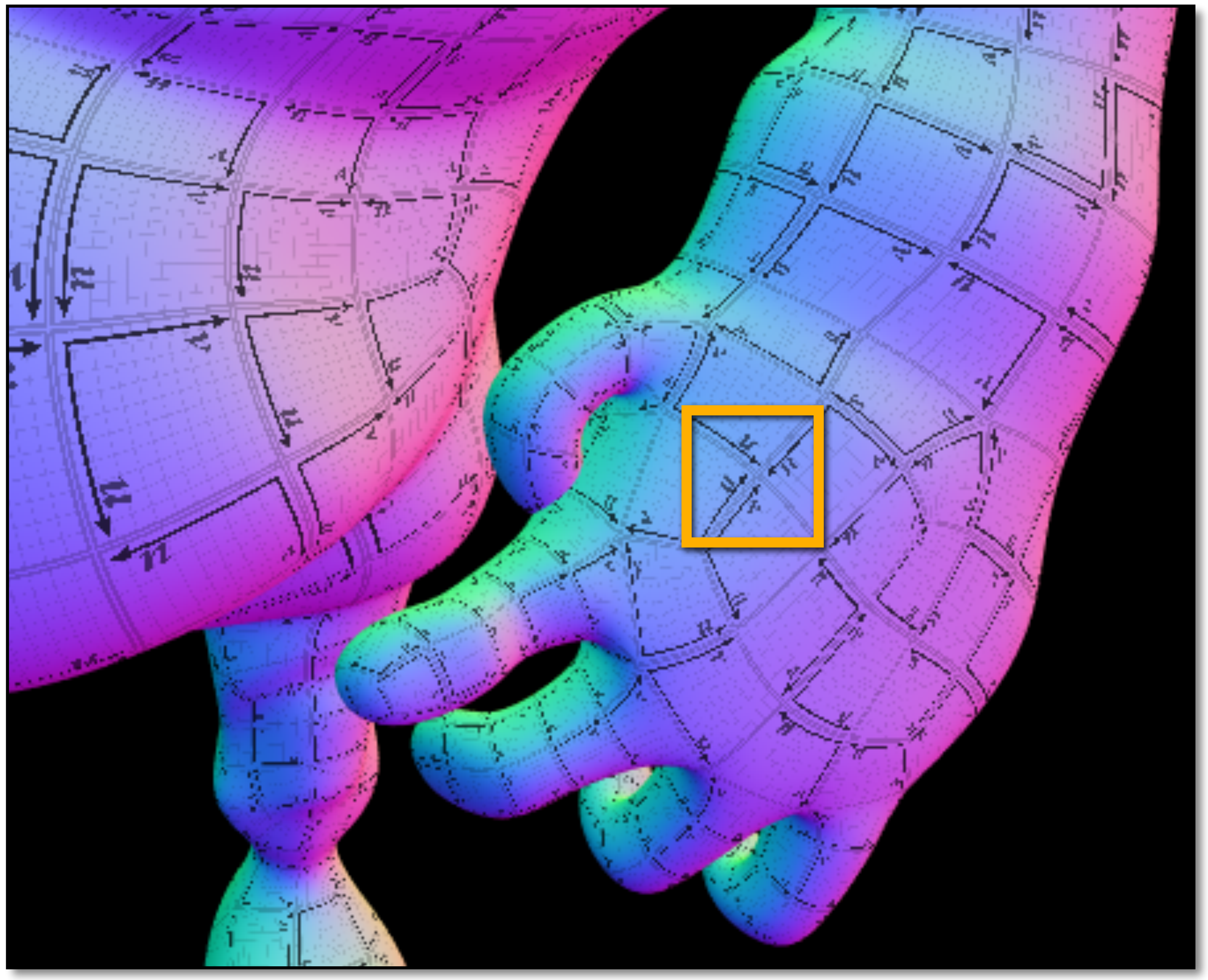
2D examples:
Moiré patterns, jaggies

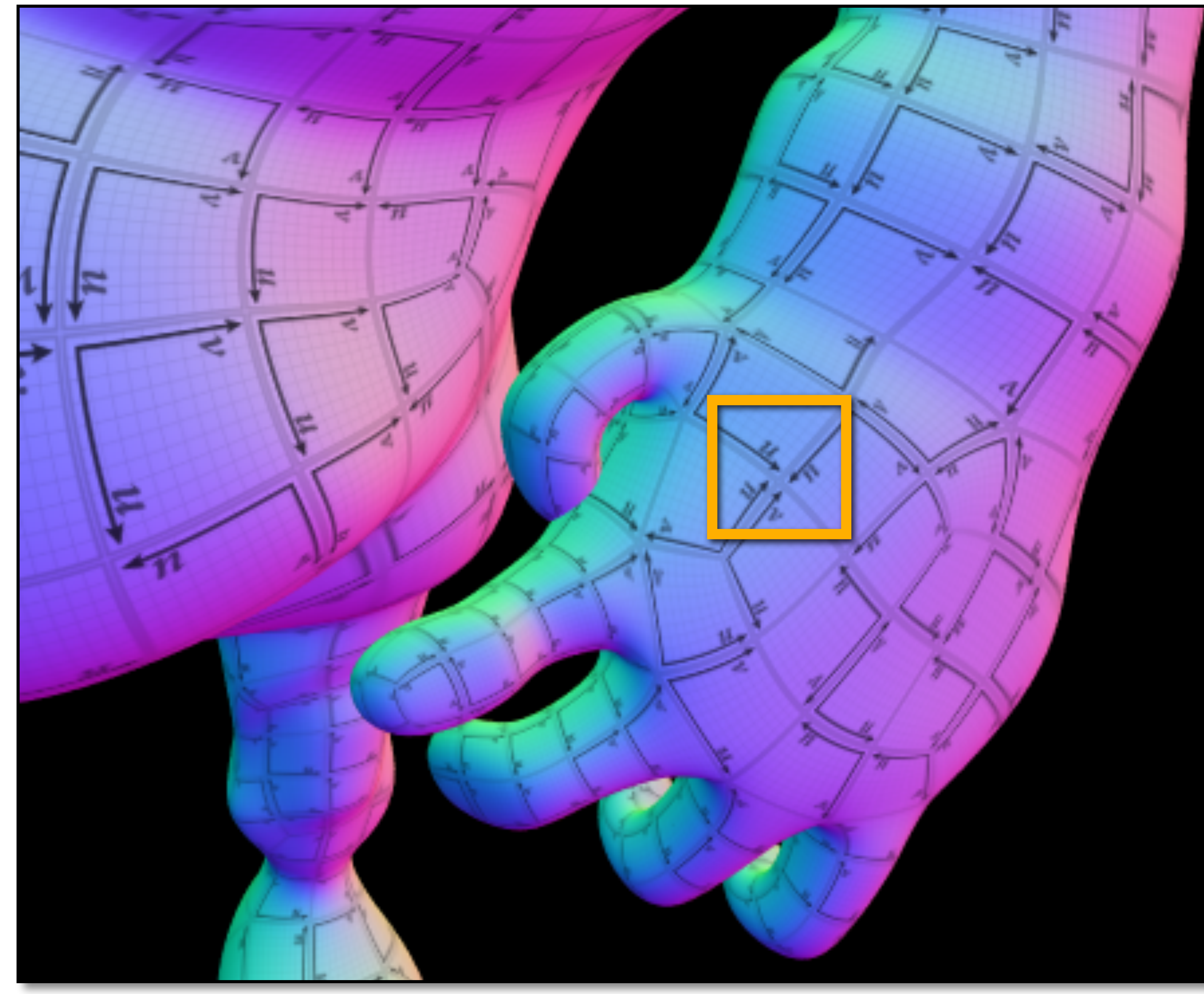


Aside: what is happening here?



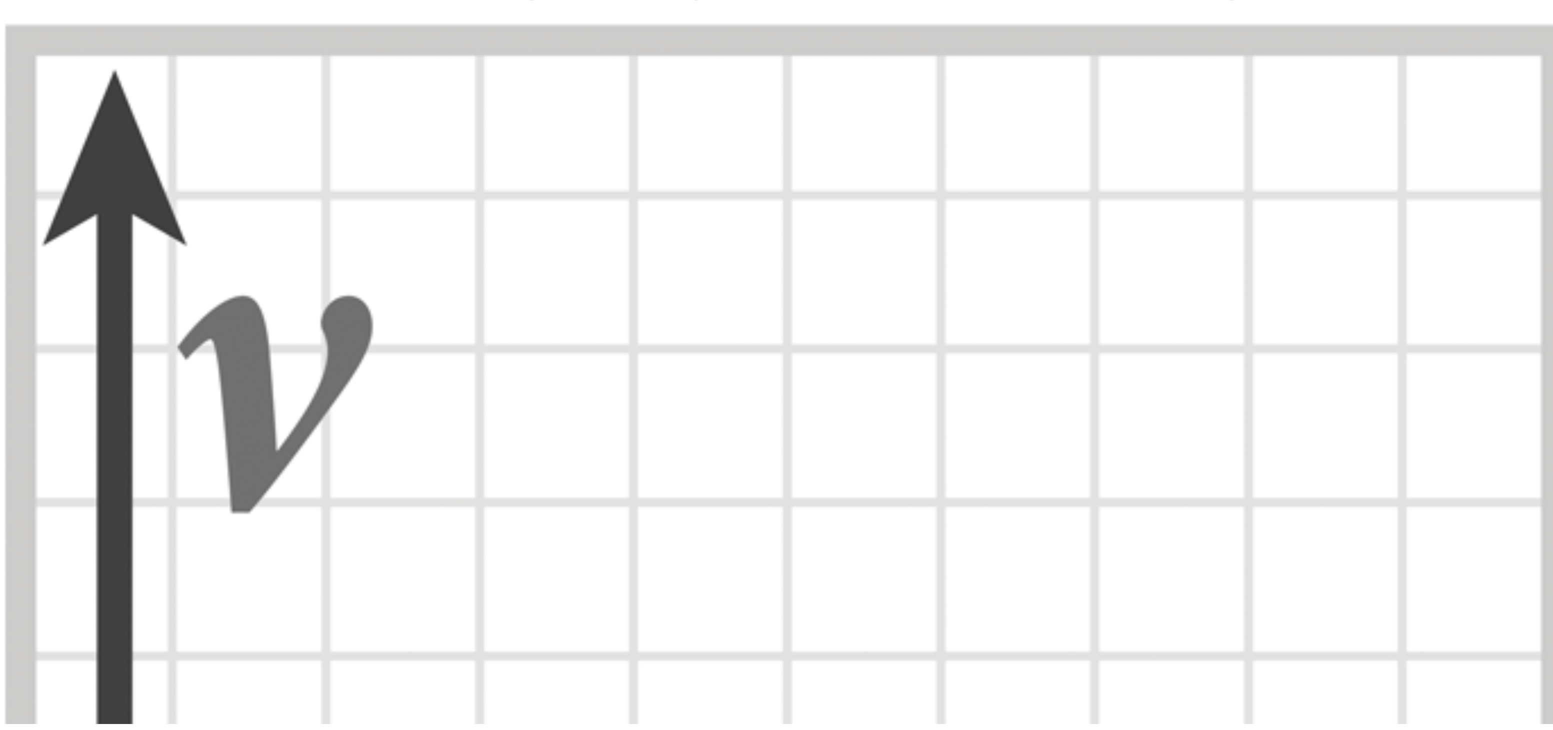
Aliasing due to undersampling texture



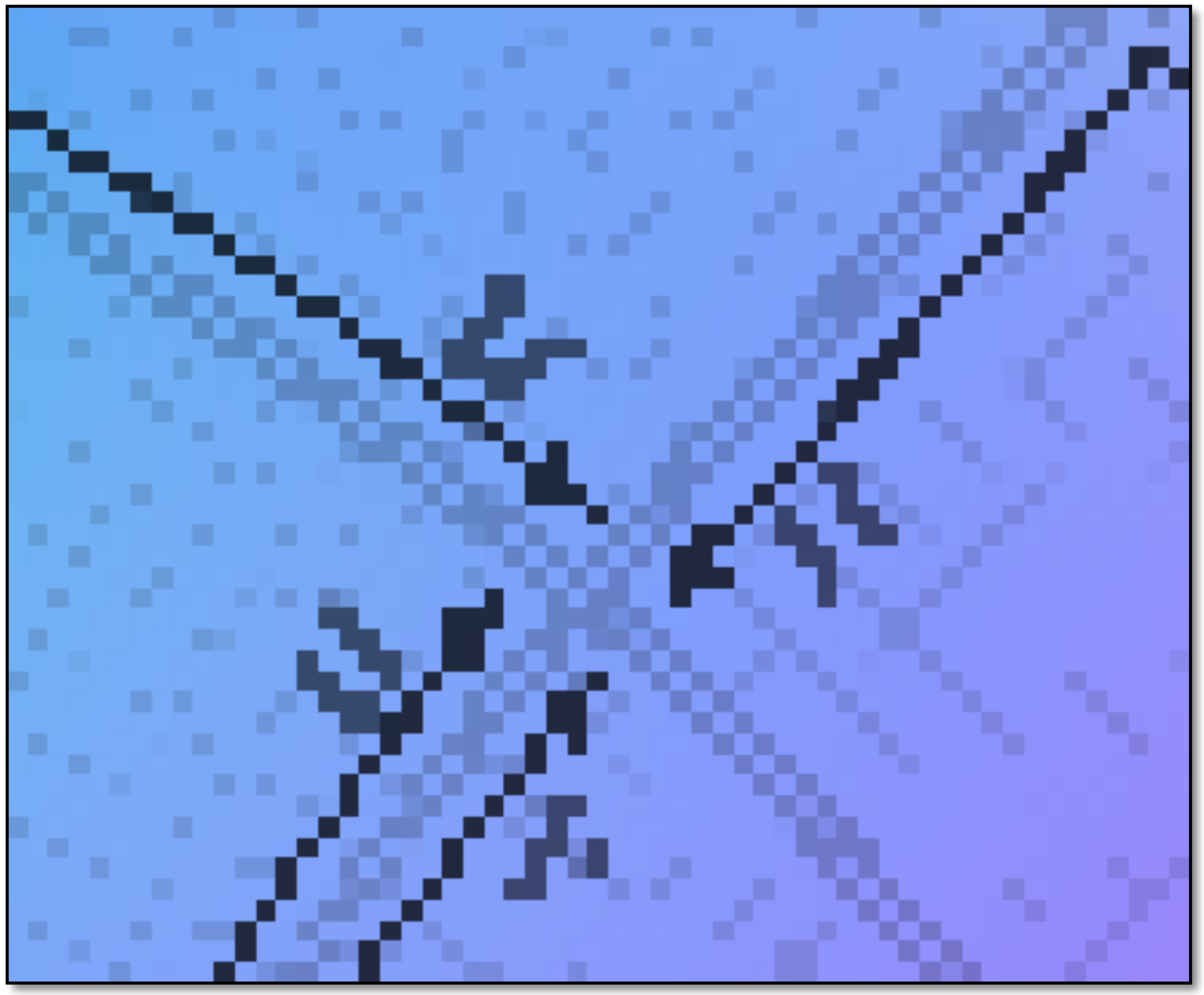


No pre-filtering of texture data
(resulting image exhibits aliasing)

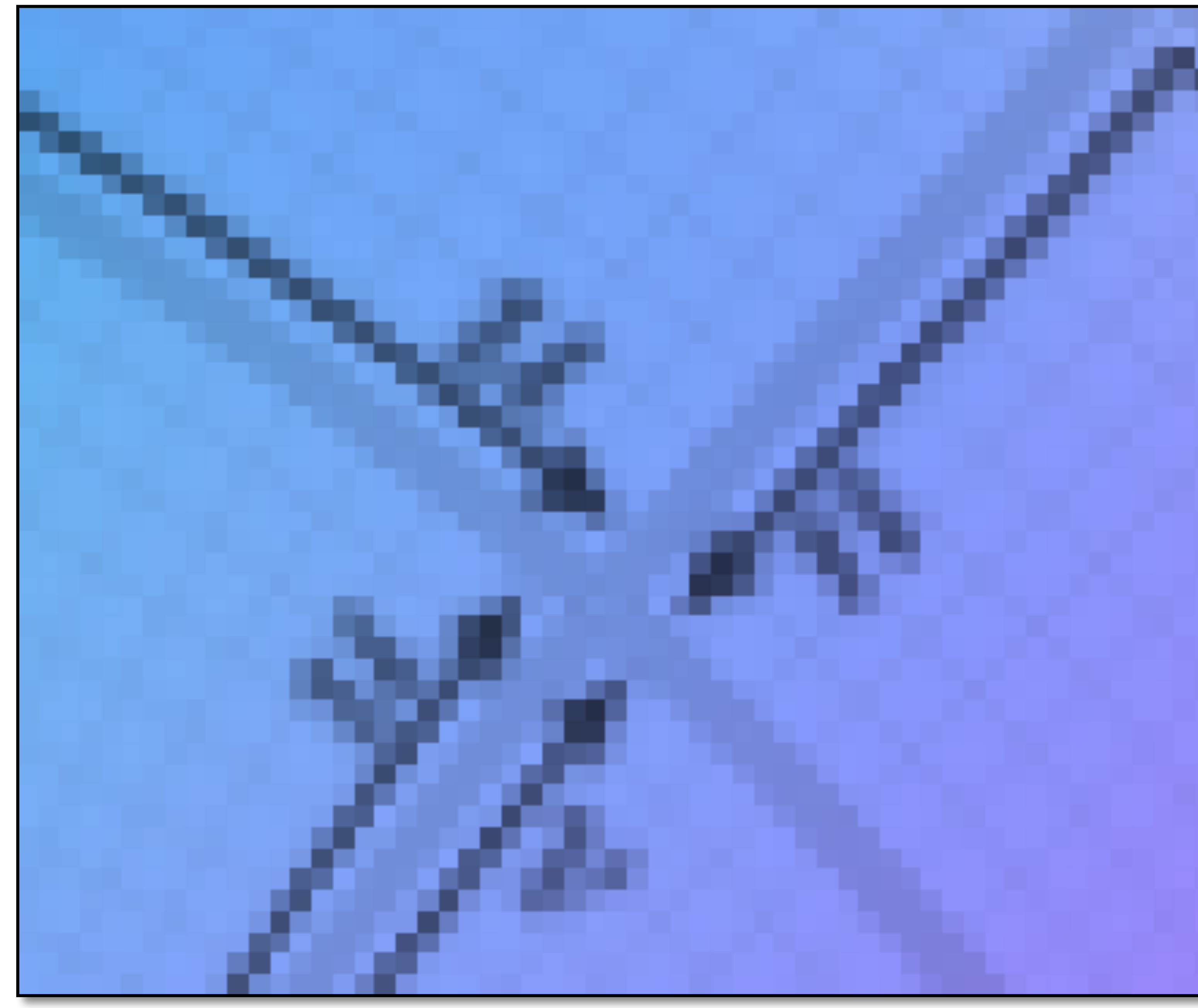
Rendering using pre-filtered texture data



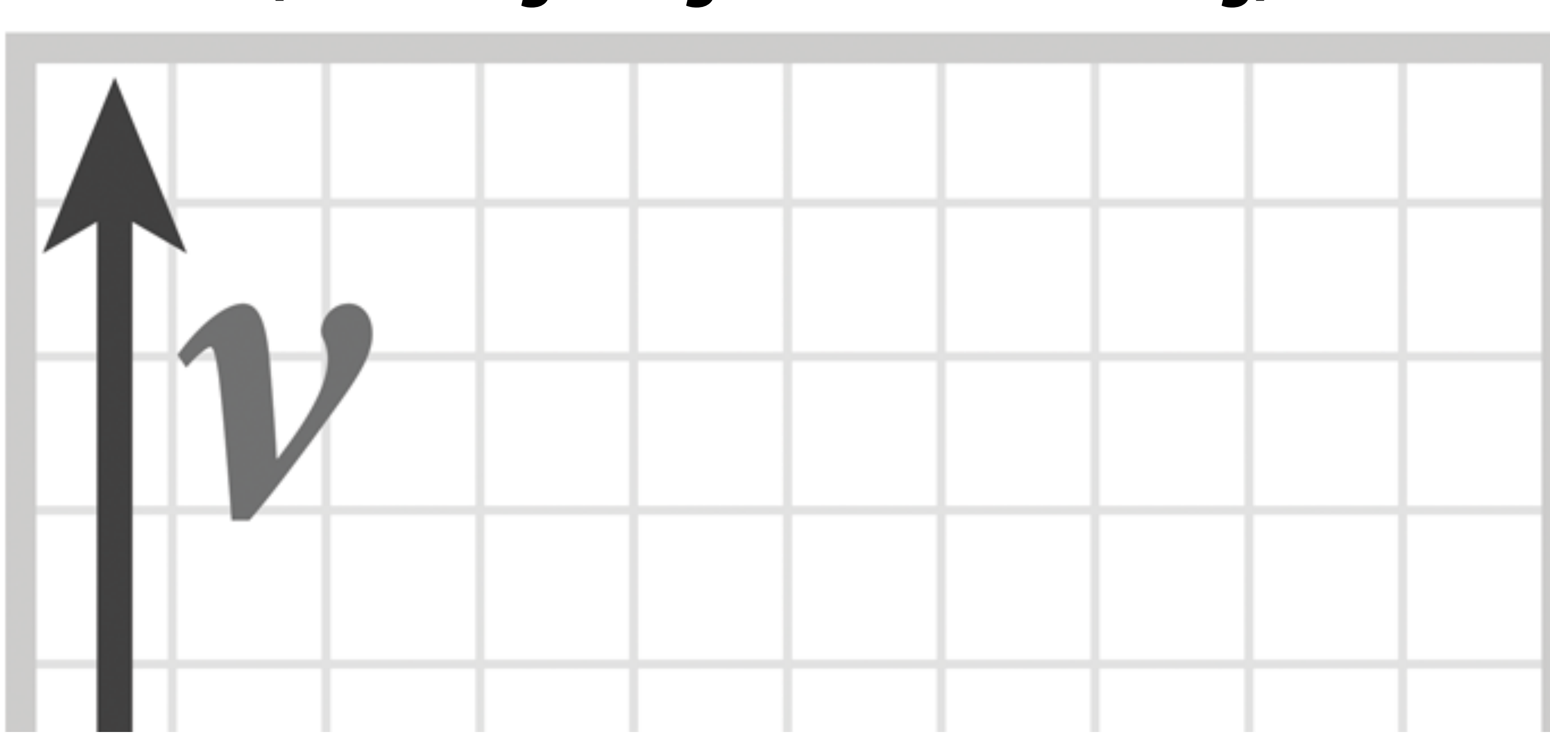
Aliasing due to undersampling (zoom)



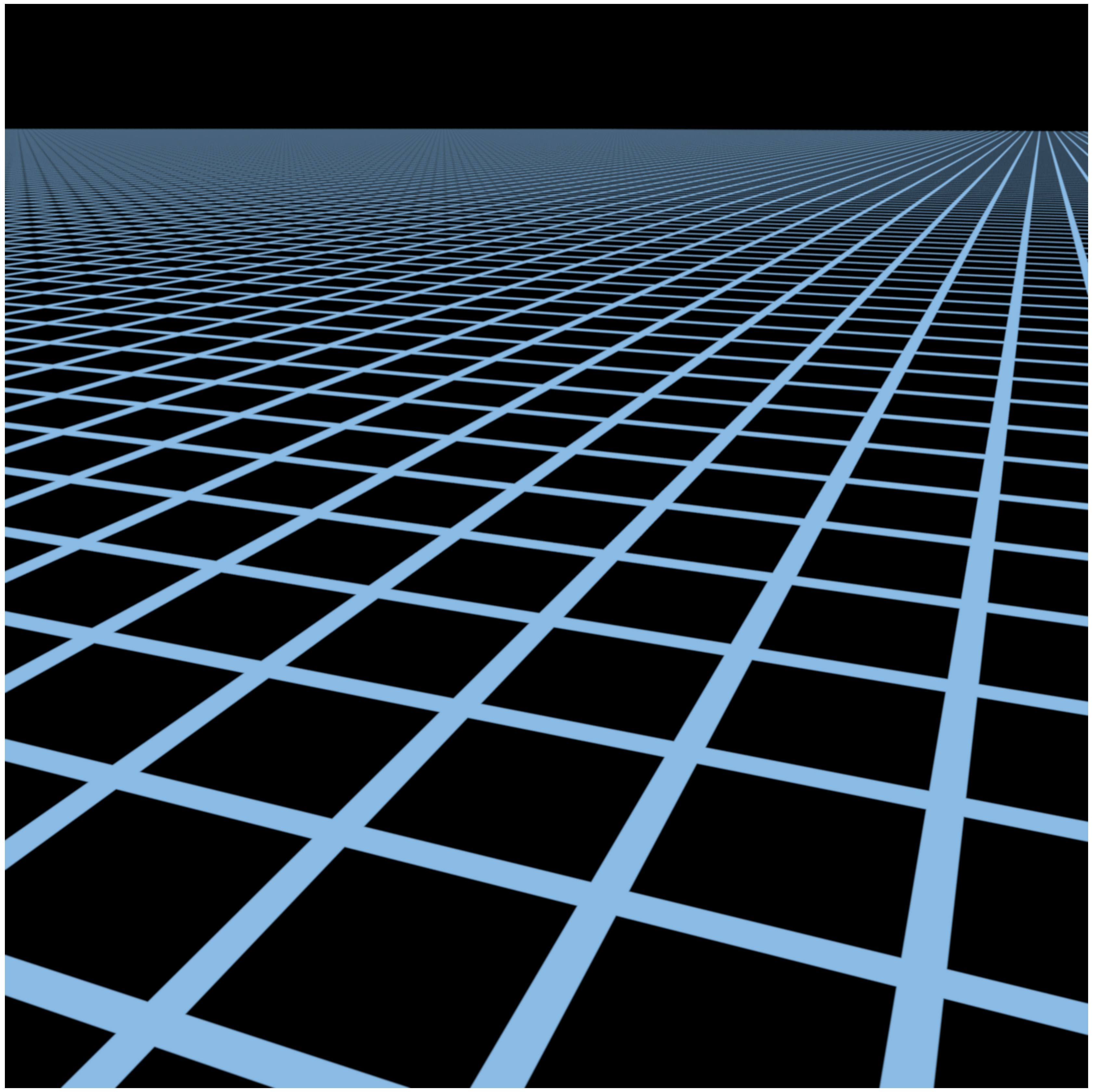
No pre-filtering of texture data
(resulting image exhibits aliasing)



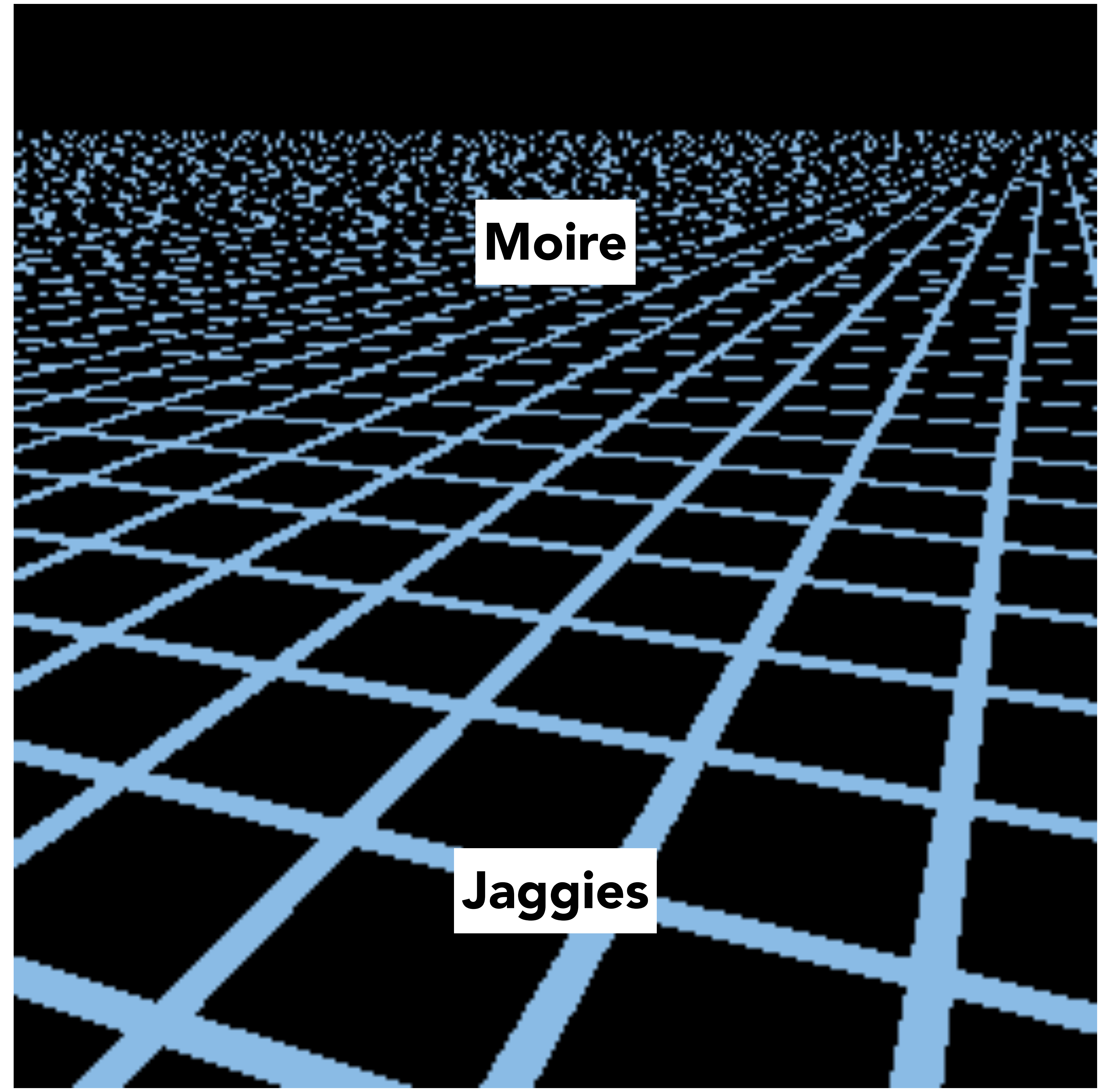
Rendering using pre-filtered texture data



Another example:



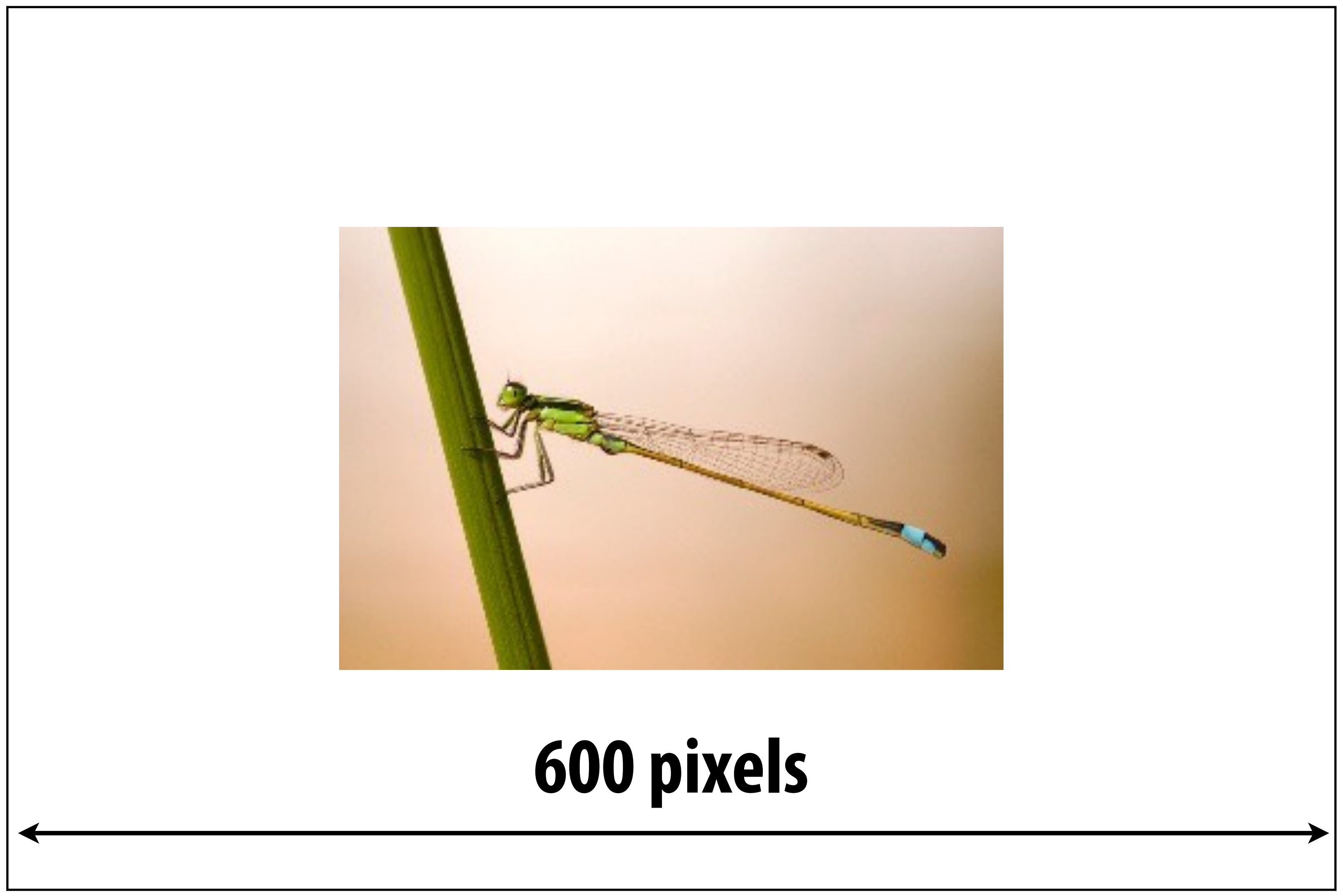
Source image: 1280x1280 pixels



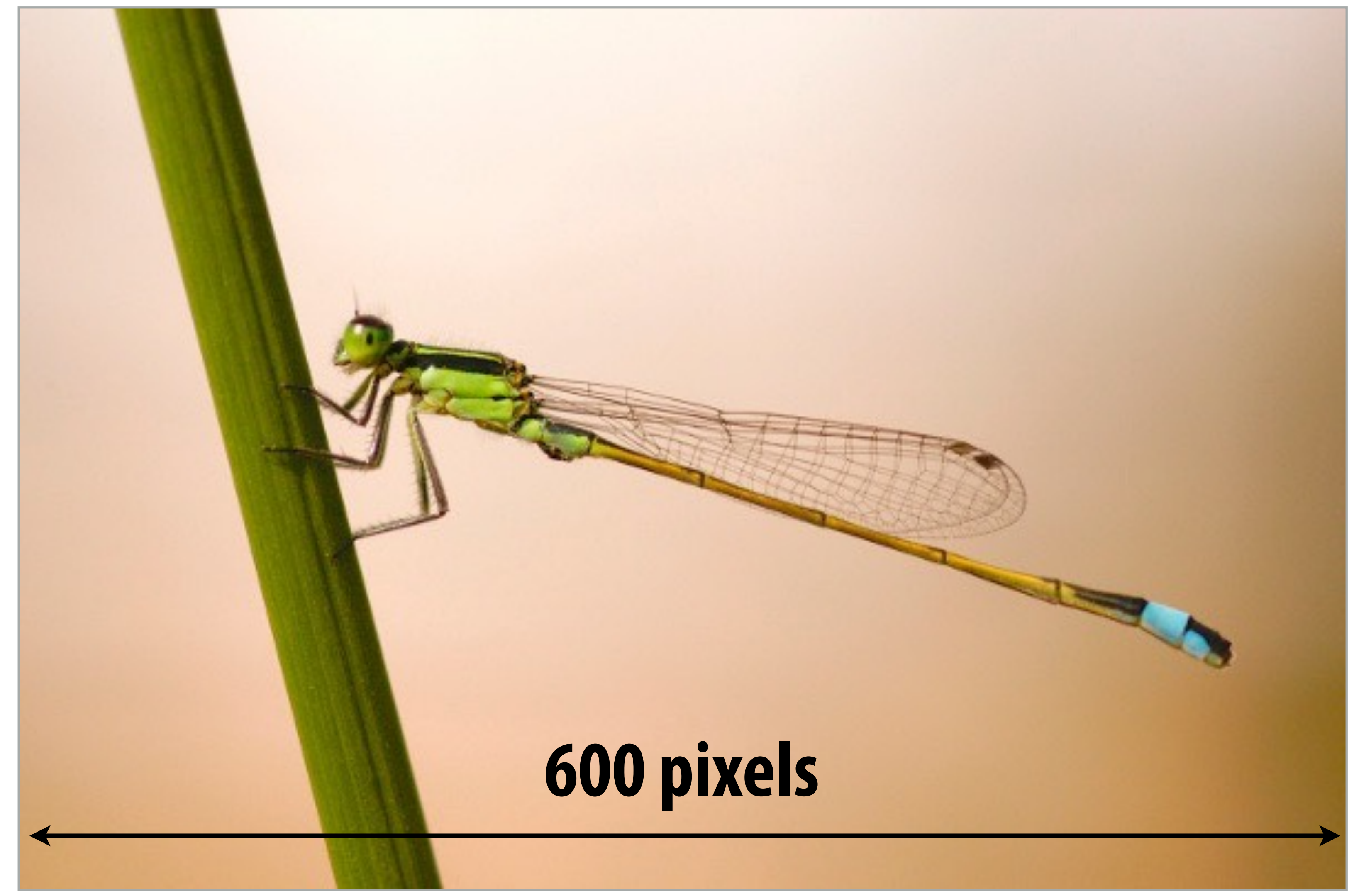
Rendered image: 256x256 pixels

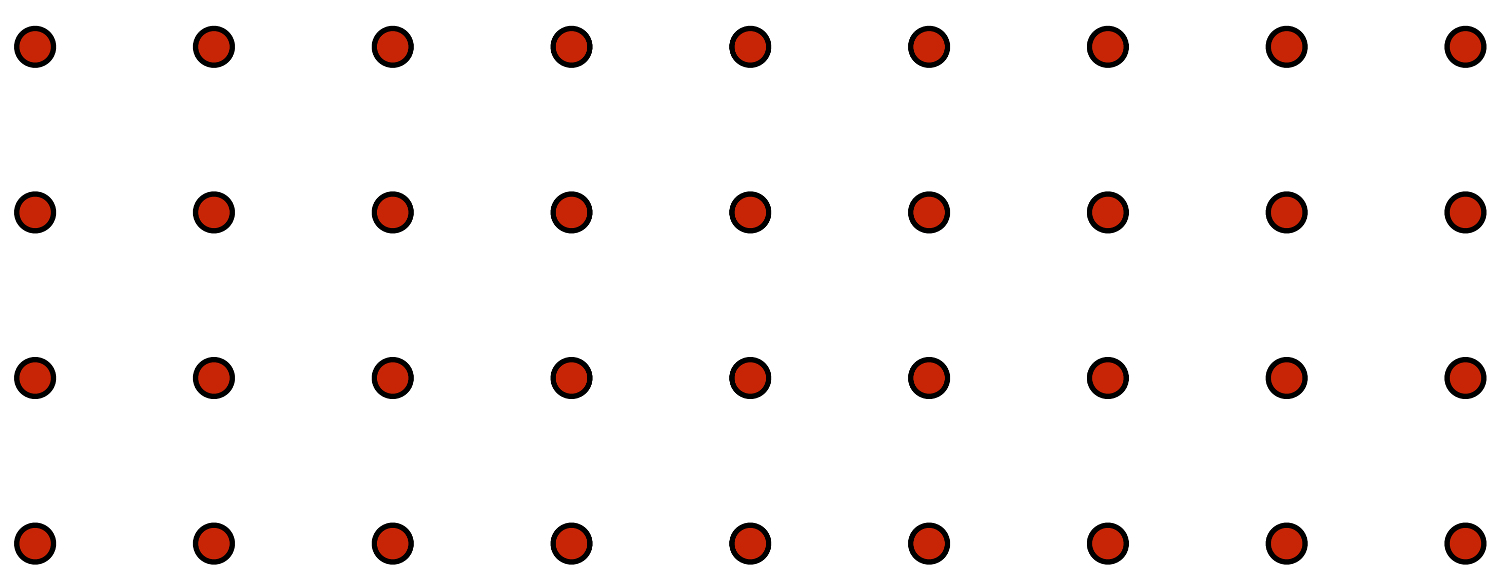
Sampling rate on screen vs texture

Rendered image (object zoomed out)

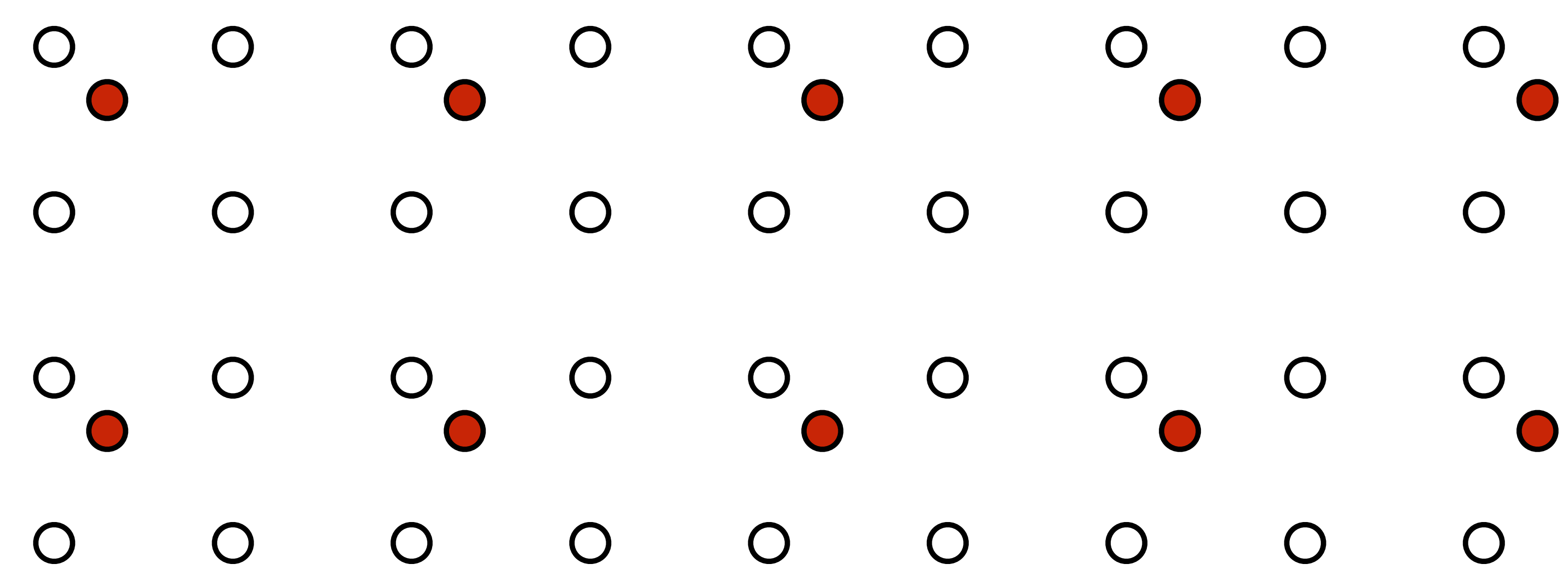


Texture Image





Screen space (x,y)



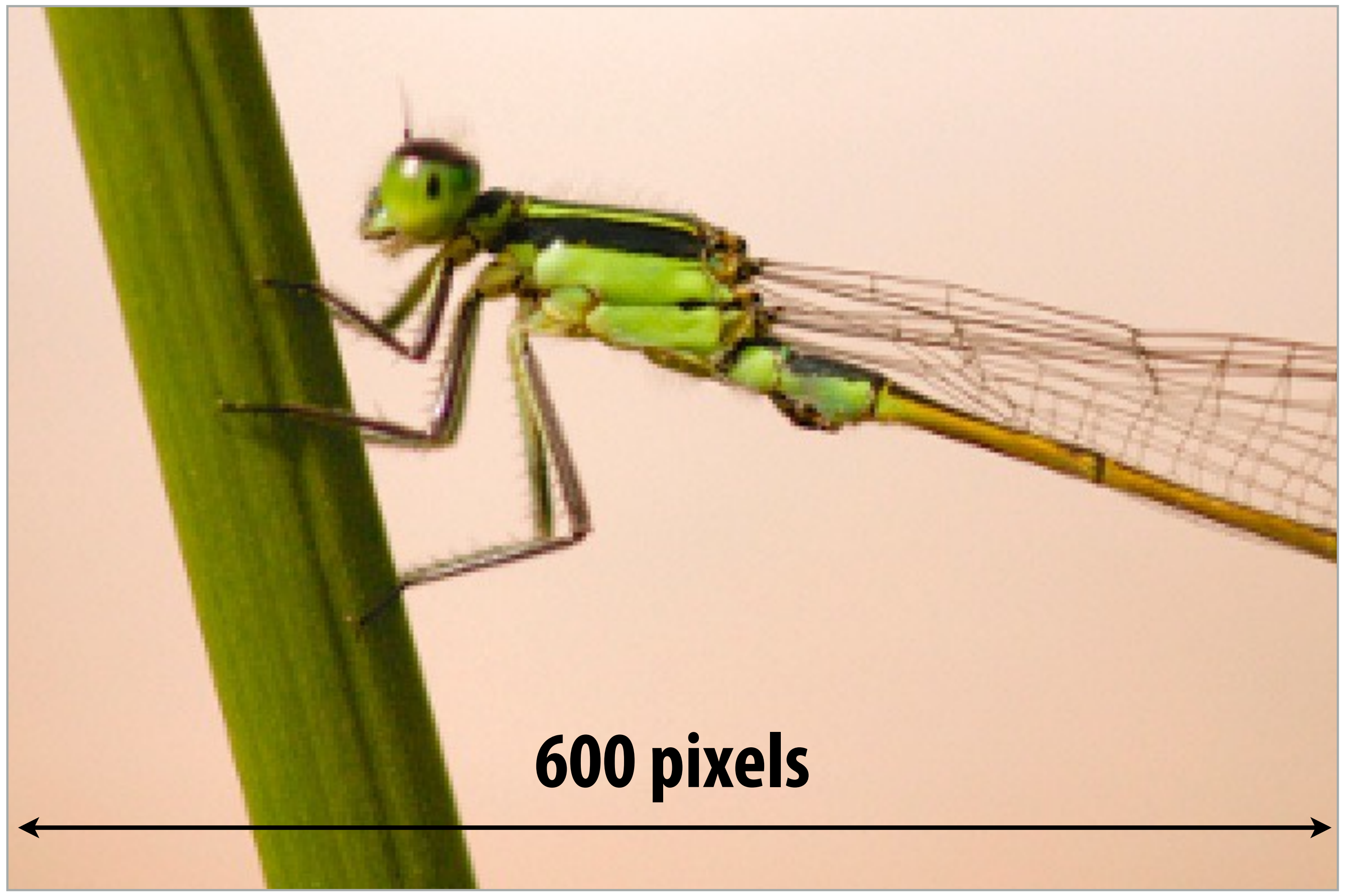
Texture space (u,v)

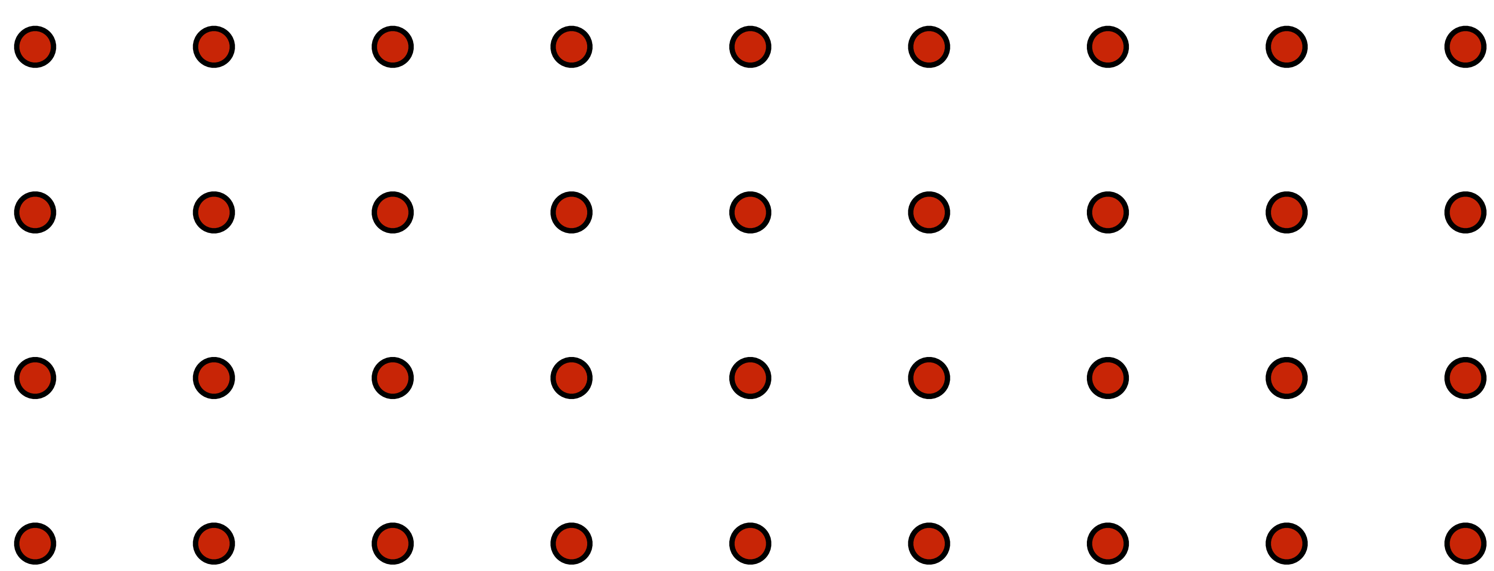
Texture is "minified"

Red dots = samples needed to render
White dots = samples existing in texture map

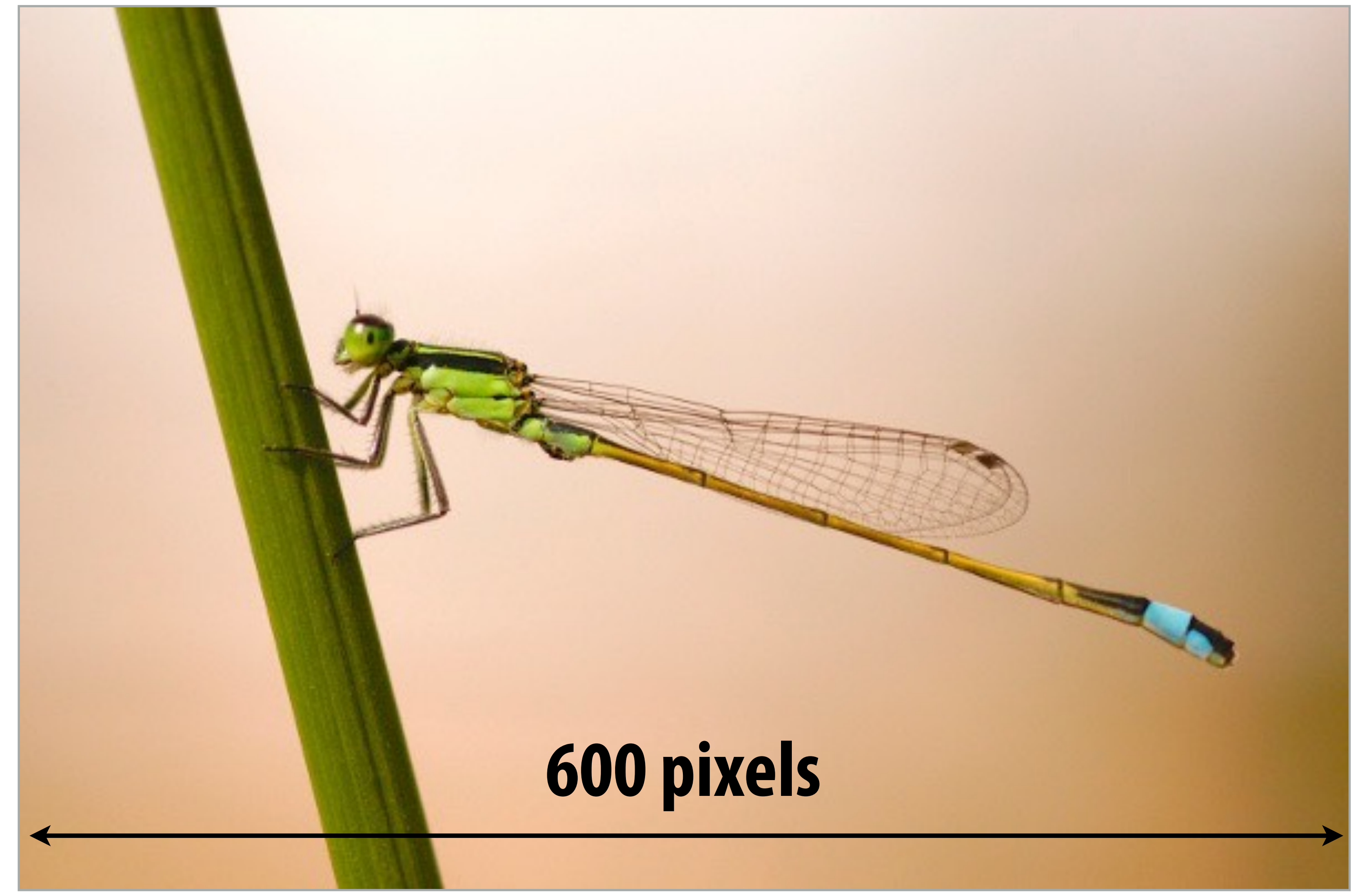
Sampling rate on screen vs texture

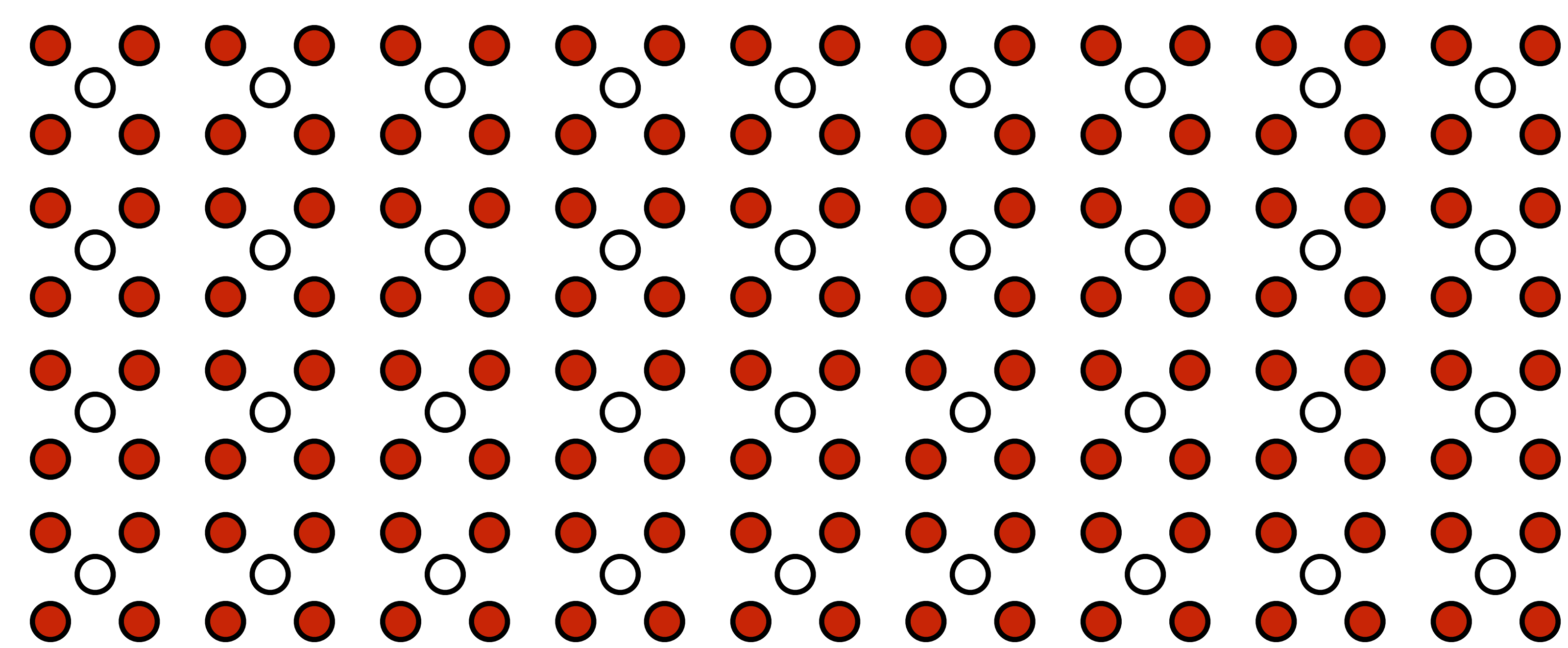
Rendered image (zoomed in)





Texture Image





Screen space (x,y)

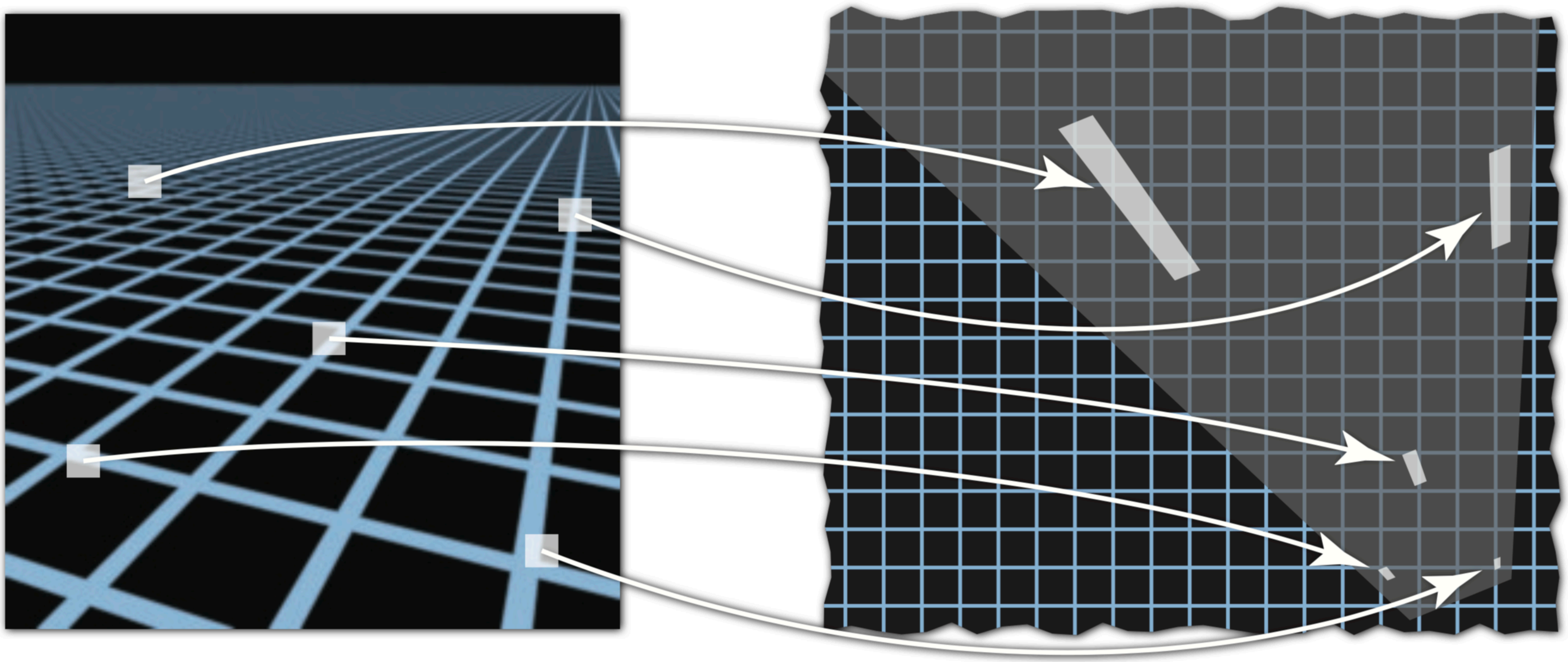
Texture space (u,v)

Texture is "magnified" on screen

Red dots = samples needed to render

White dots = samples existing in texture map

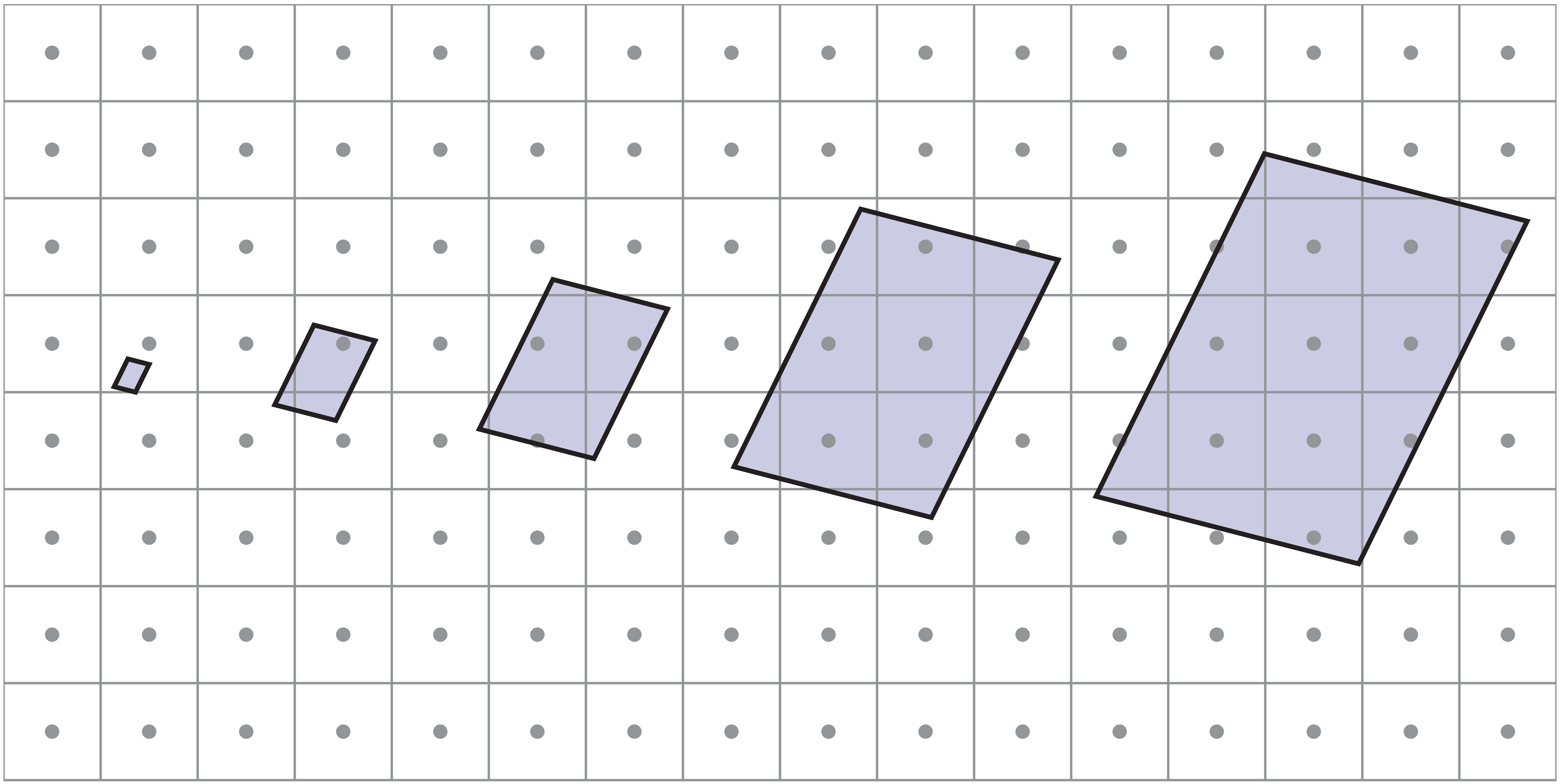
Screen pixel footprint in texture space



Screen space **Texture space**

Texture sampling pattern not rectilinear or isotropic

Screen pixel footprint in texture space



Upsampling (Magnification) ⟷ **Downsampling (Minification)**

Camera zoomed in close to object

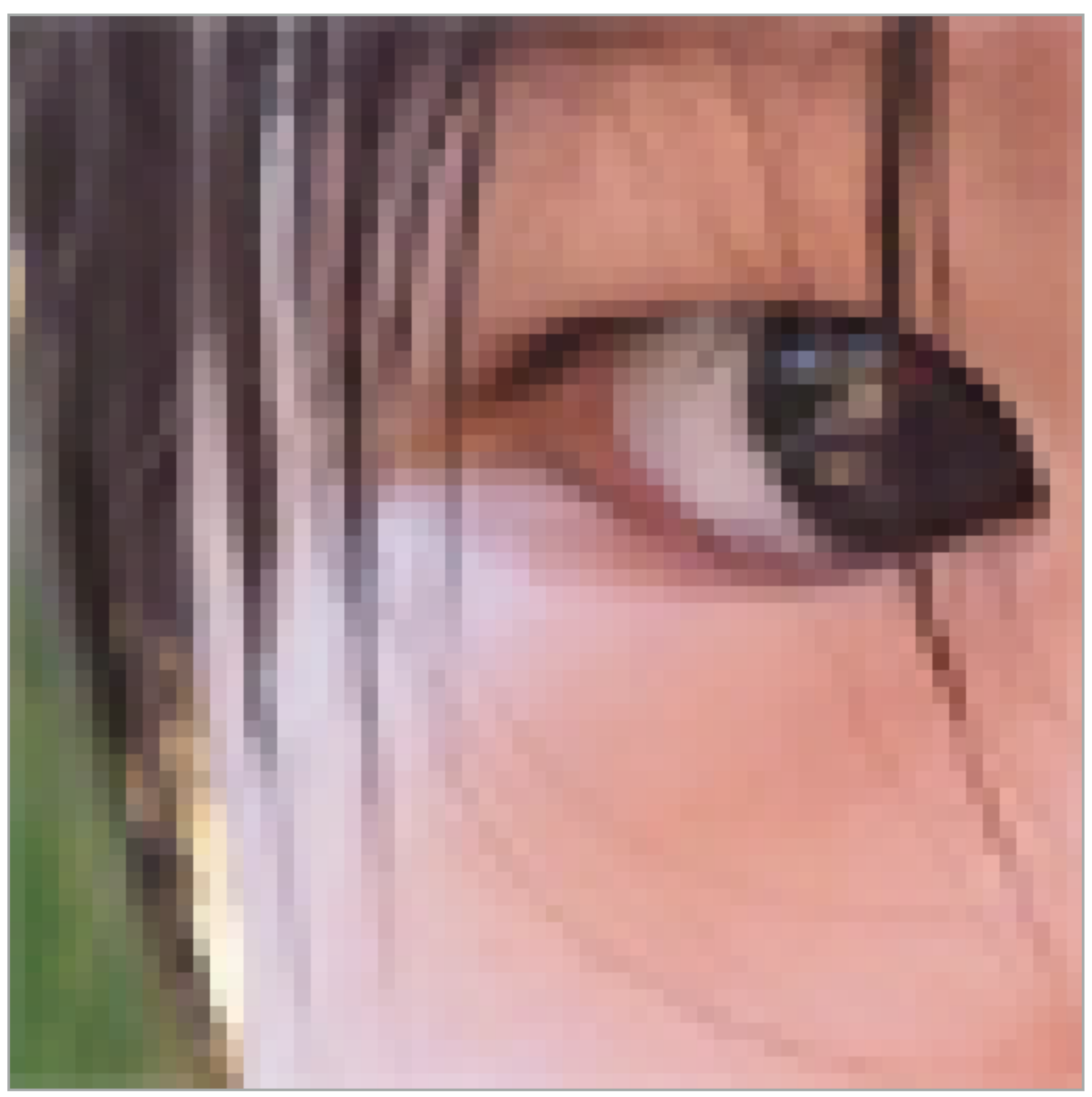
Camera far away from object

Screen pixel area vs texel area

- At optimal viewing size:
  - 1:1 mapping between pixel sampling rate and texel sampling rate
  - Dependent on screen and texture resolution!
- When pixel area is larger than texel area (texture minification)
  - Think: zoom far out from object
  - One pixel sample per multiple texel samples
- When pixel area is smaller than texel area (texture magnification)
  - Think: zoom in on an object
  - Multiple pixel samples per texel sample

Texture magnification - easy case

- Generally don't want this situation — it means we have insufficient texture resolution)
- This is image interpolation (below: three different kernel functions)

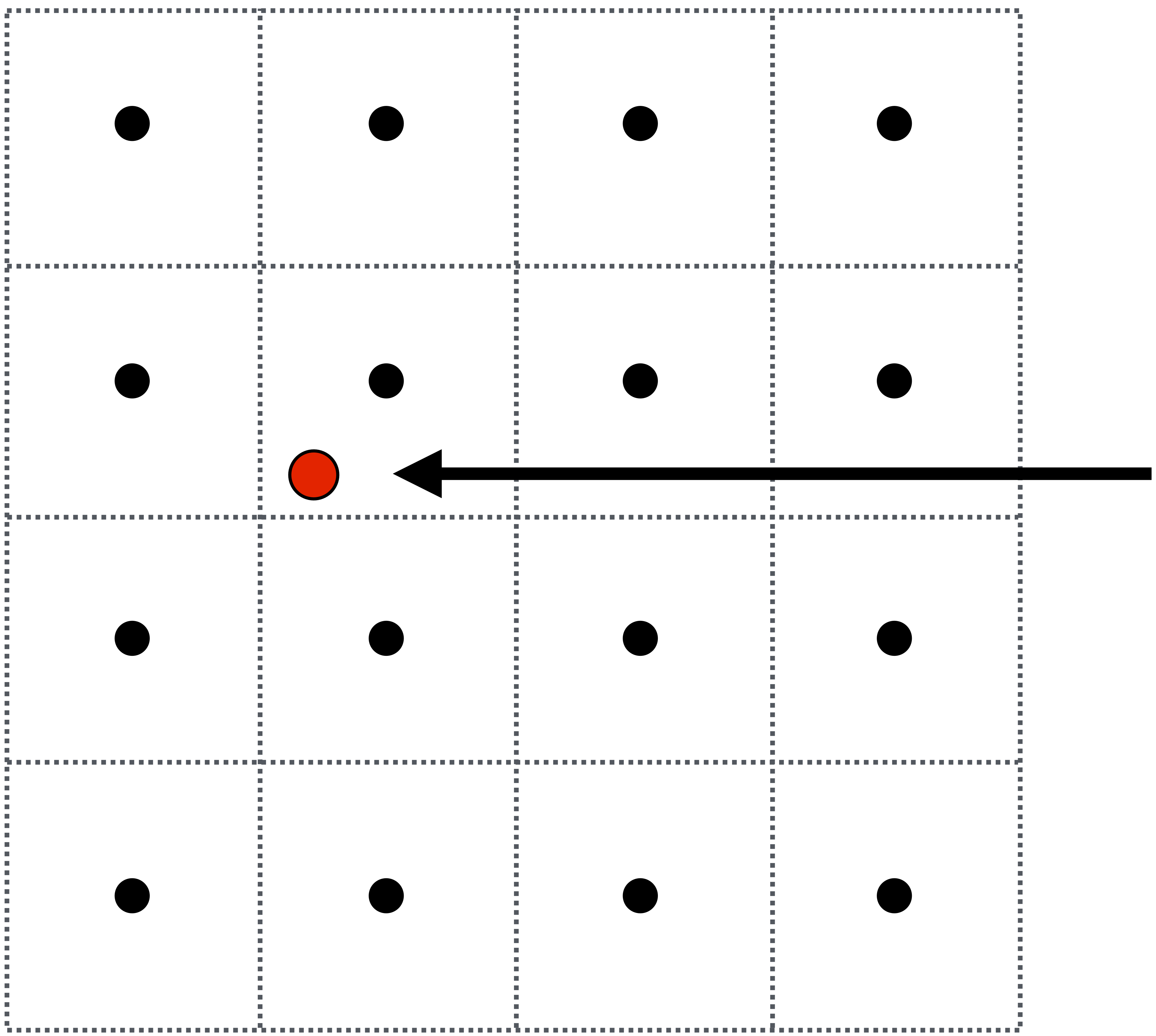


Nearest

Bilinear

Bicubic

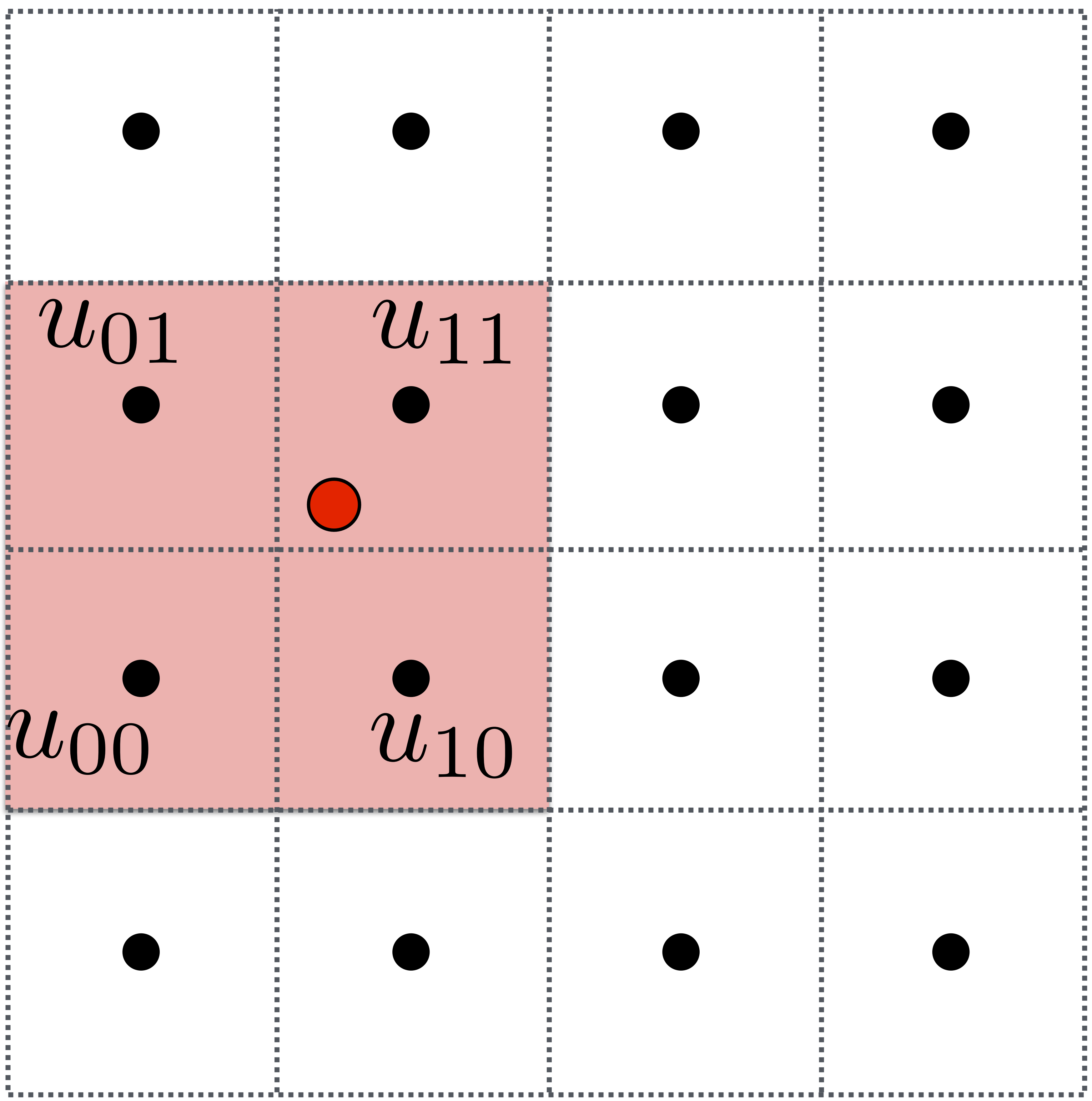
Bilinear filtering



Want to sample texture value f(x,y) at red point

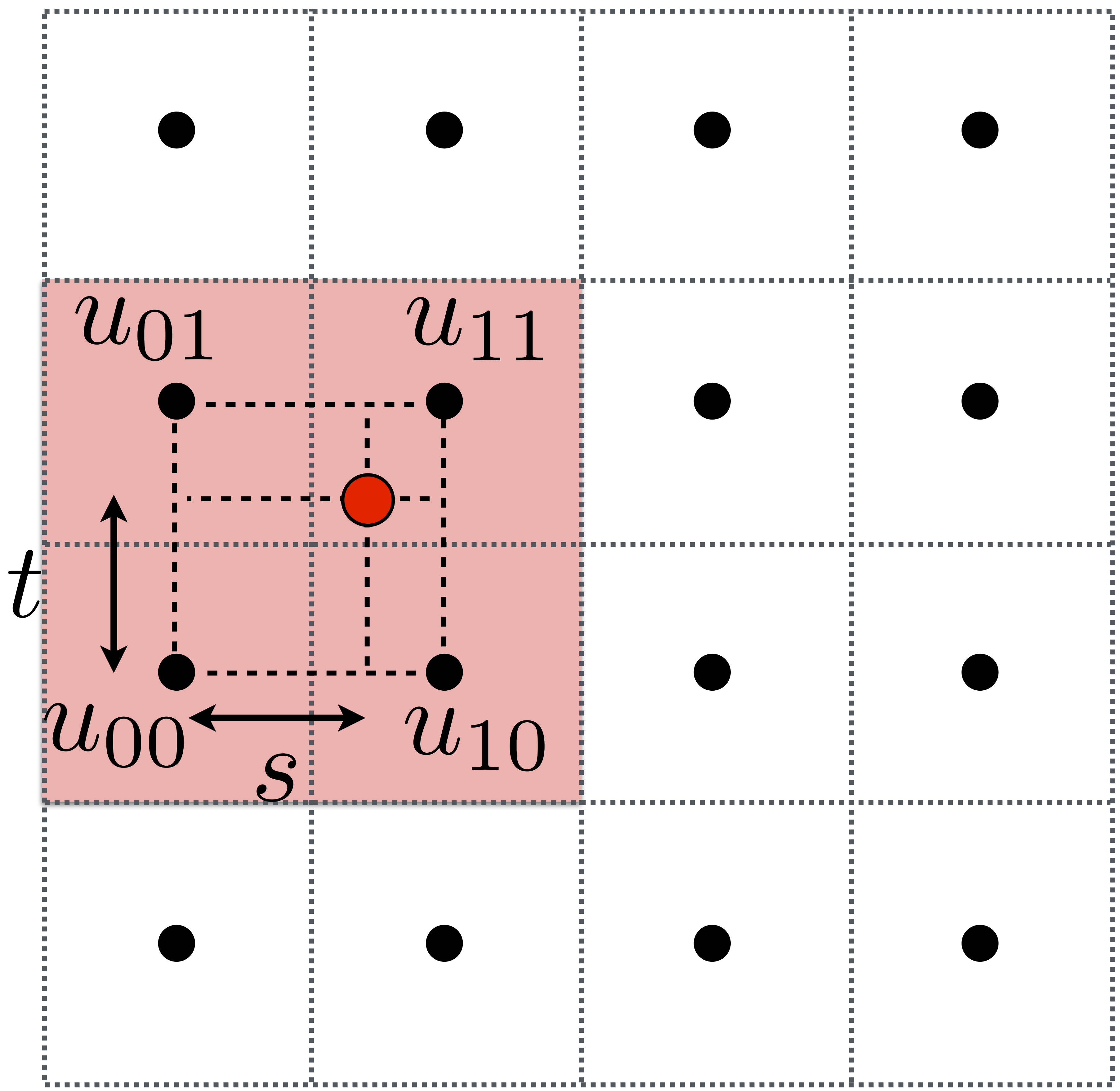
Black points indicate texture sample locations

Bilinear filtering



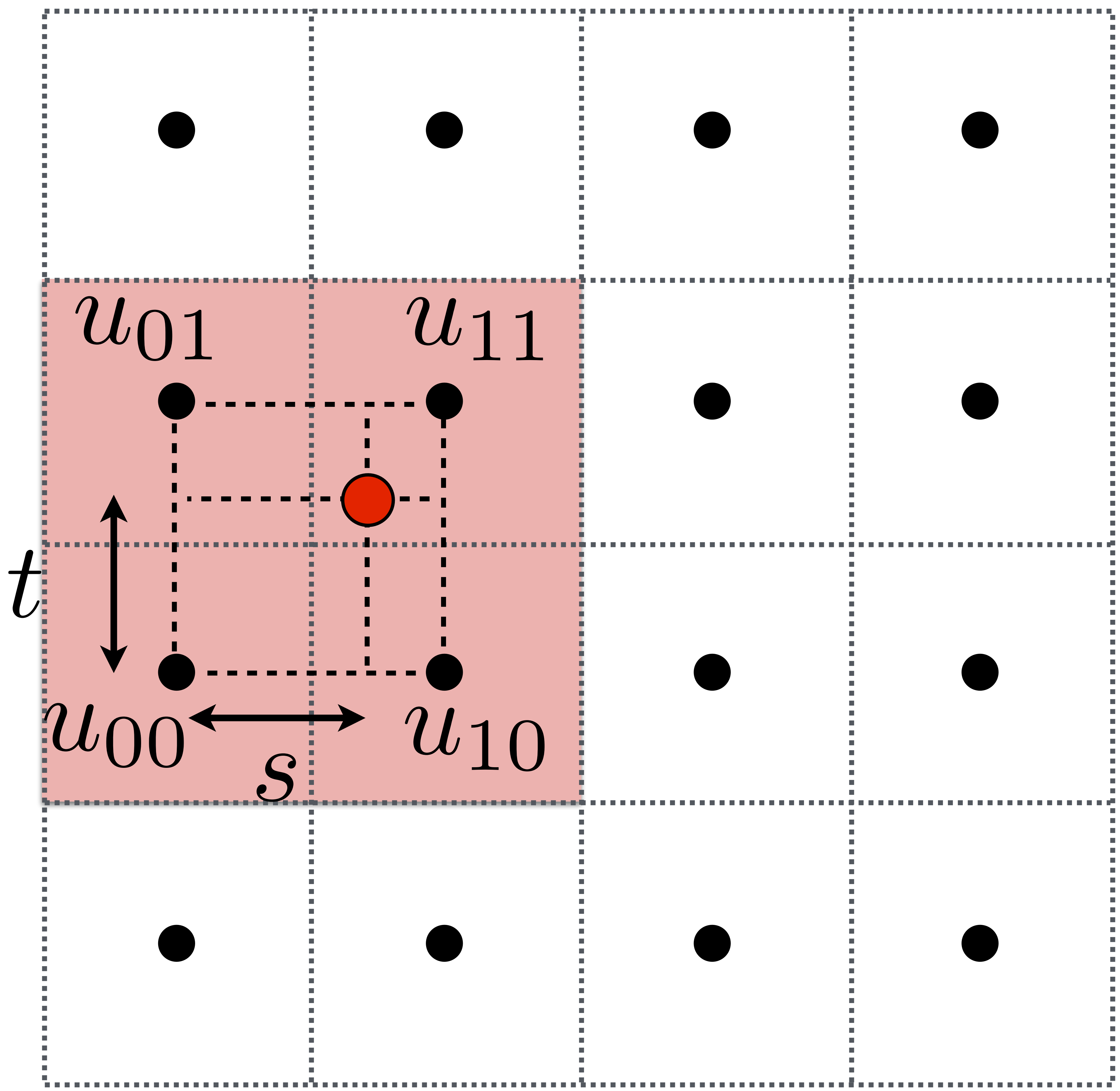
Take 4 nearest sample locations, with texture values as labeled.

Bilinear filtering



And fractional offsets, (s,t) as shown

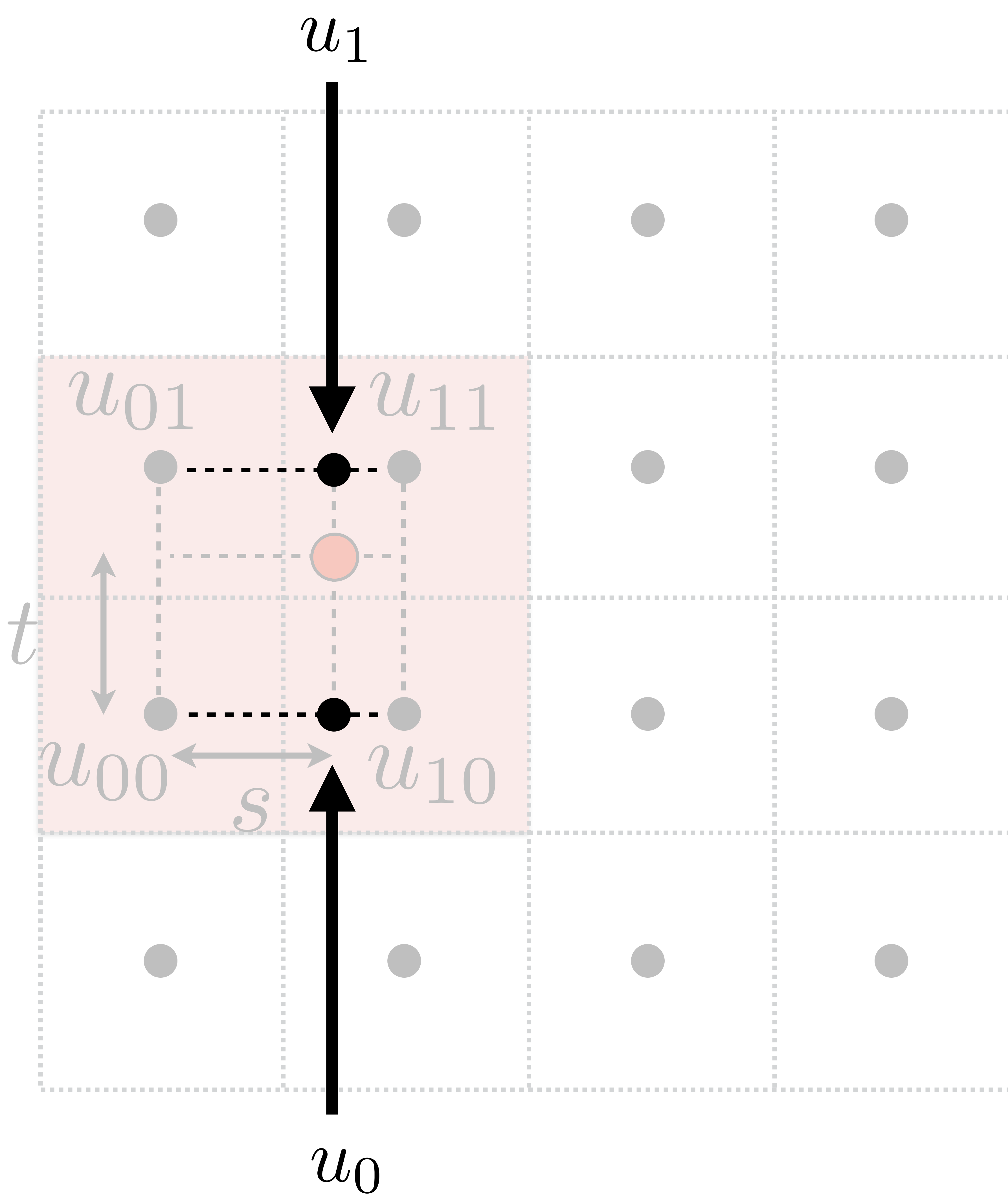
Bilinear filtering



Linear interpolation (1D)

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

Bilinear filtering



Linear interpolation (1D)

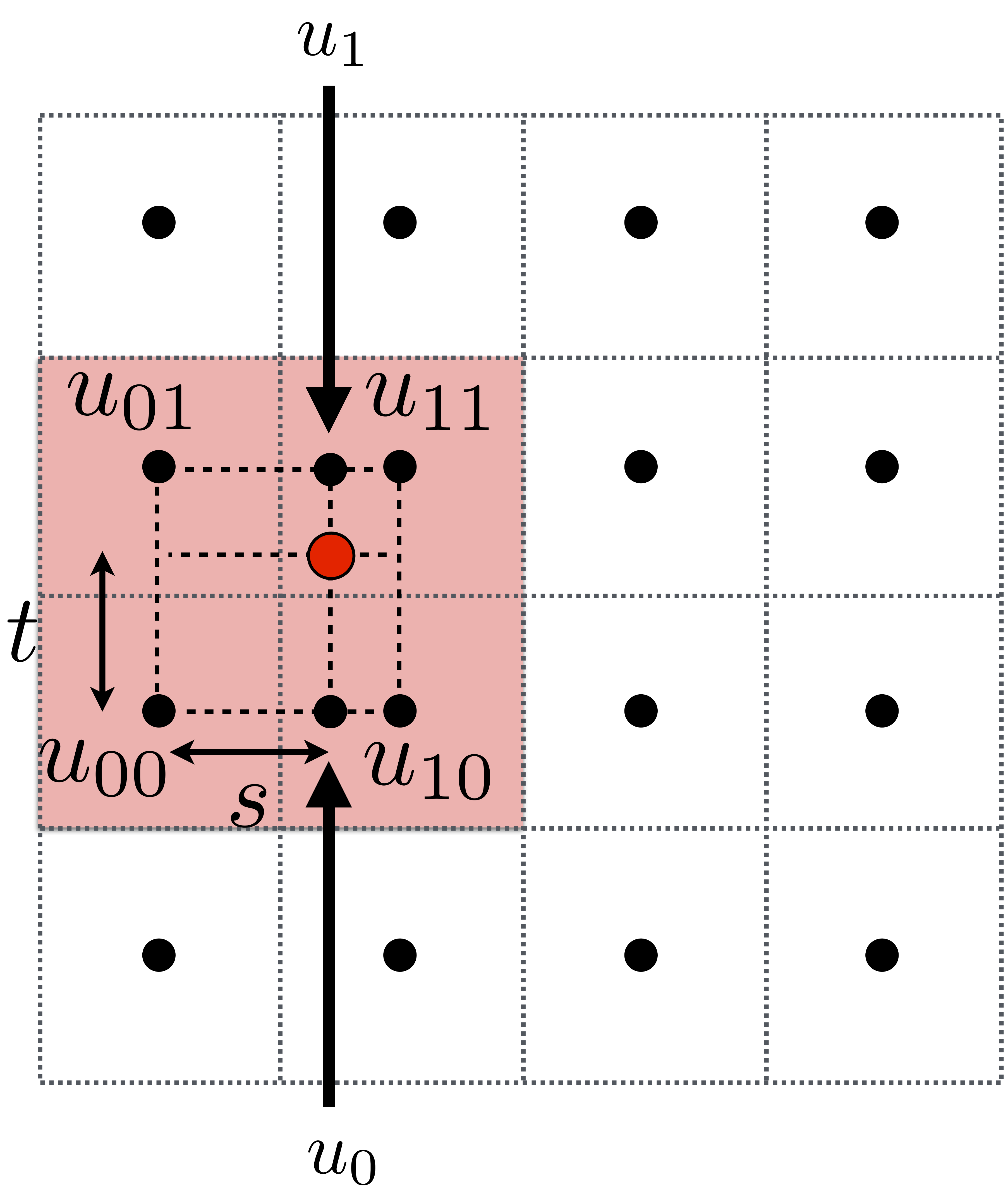
$$\text{lerp}(x, v_0, v_1) = v_0 + x(v_1 - v_0)$$

Two helper lerps (horizontal)

$$u_0 = \text{lerp}(s, u_{00}, u_{10})$$

$$u_1 = \text{lerp}(s, u_{01}, u_{11})$$

Bilinear filtering



Linear interpolation (1D)

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

Two helper lerps

$$u_0 = \text{lerp}(s, u_{00}, u_{10})$$

$$u_1 = \text{lerp}(s, u_{01}, u_{11})$$

Final vertical lerp, to get result:

$$f(x, y) = \text{lerp}(t, u_0, u_1)$$

Reconstruction filter function

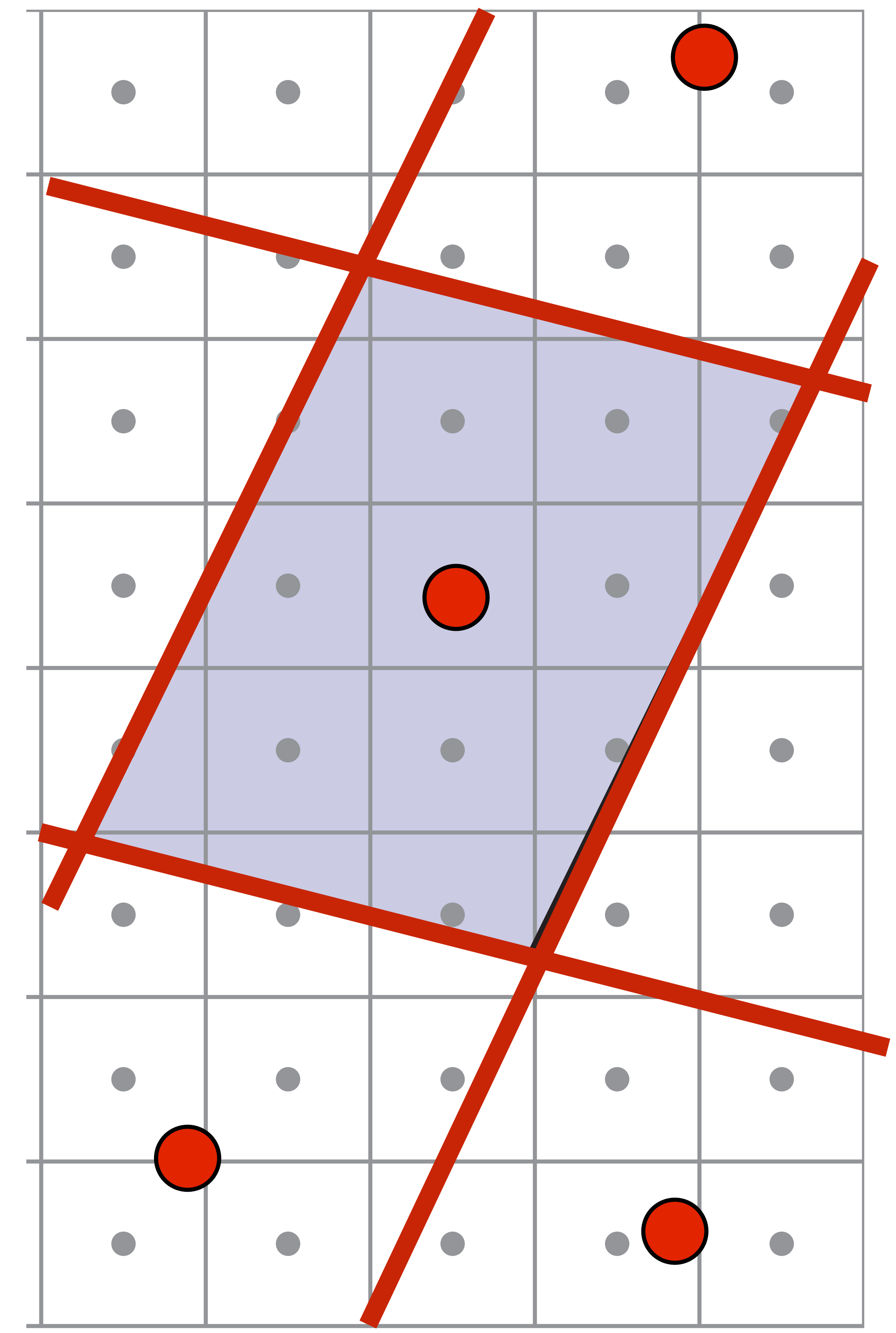
- **Test your understanding:**
  - What is the reconstruction filter k(x,y) for bilinear interpolation? Nearest?

Nearest | Bilinear | Bicubic

Texture minification - hard case

- Challenging
  - Many texels can contribute to pixel (only sampling one of them could yield aliasing)
  - Shape of pixel footprint can be complex

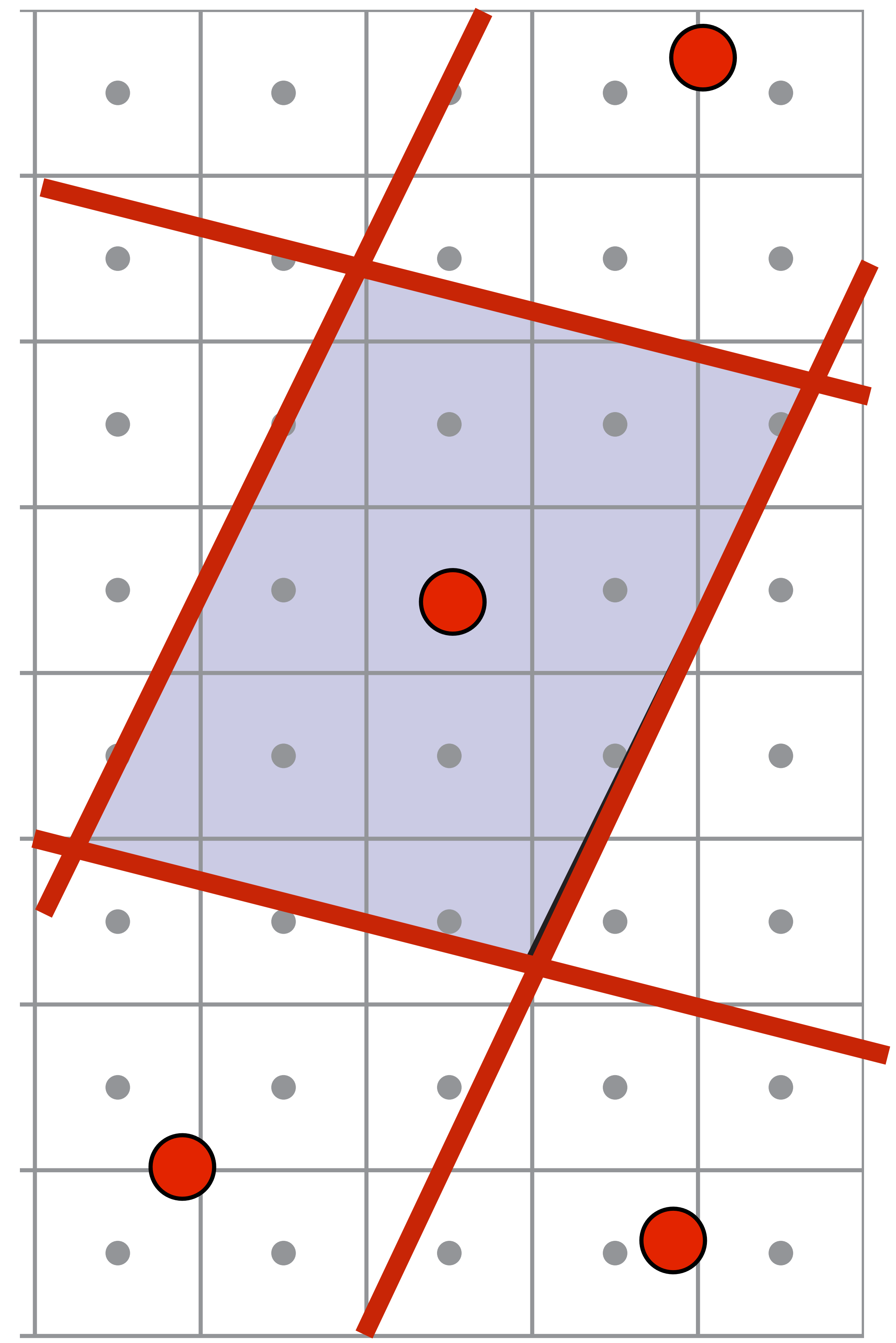
- One solution that you already know: supersampling
  - Averaging many texture samples per pixel can approximate result of convolving texture map with pixel-area sized filter
  - Problem?



Shader region = pixel area
Red lines = screen pixel boundaries
Red dots = texture space sample points for adjacent pixels

Texture minification - hard case

- Challenging
  - Many texels can contribute to pixel (only sampling one of them could yield aliasing)
  - Shape of pixel footprint can be complex
- Idea:
  - Low-pass filter and downsample texture file, and store successively lower resolutions
  - For each sample, use the texture file whose resolution approximates the screen sampling rate

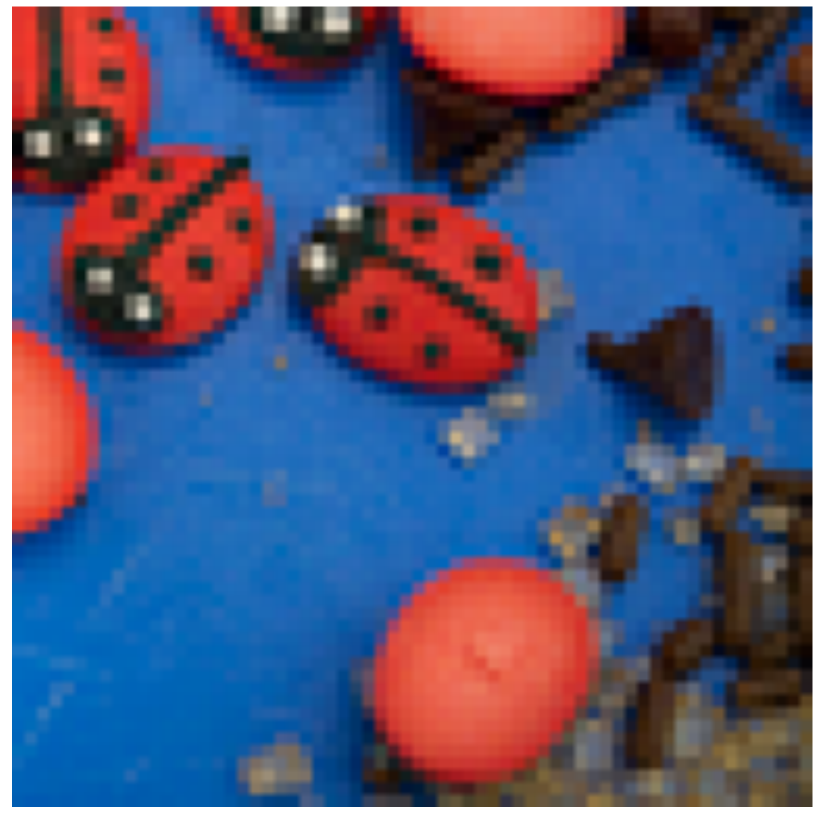


Shader region = pixel area
Red lines = screen pixel boundaries
Red dots = texture space sample points for adjacent pixels

Mipmap (L. Williams 83)

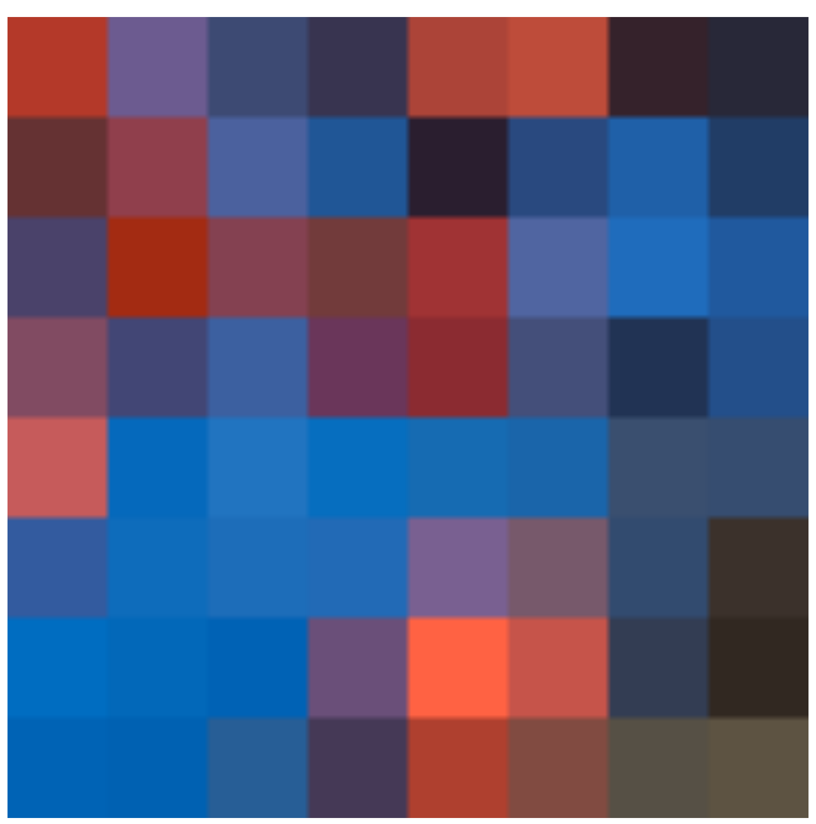
Each mipmap level is downsampled (low-pass filtered) version of the previous

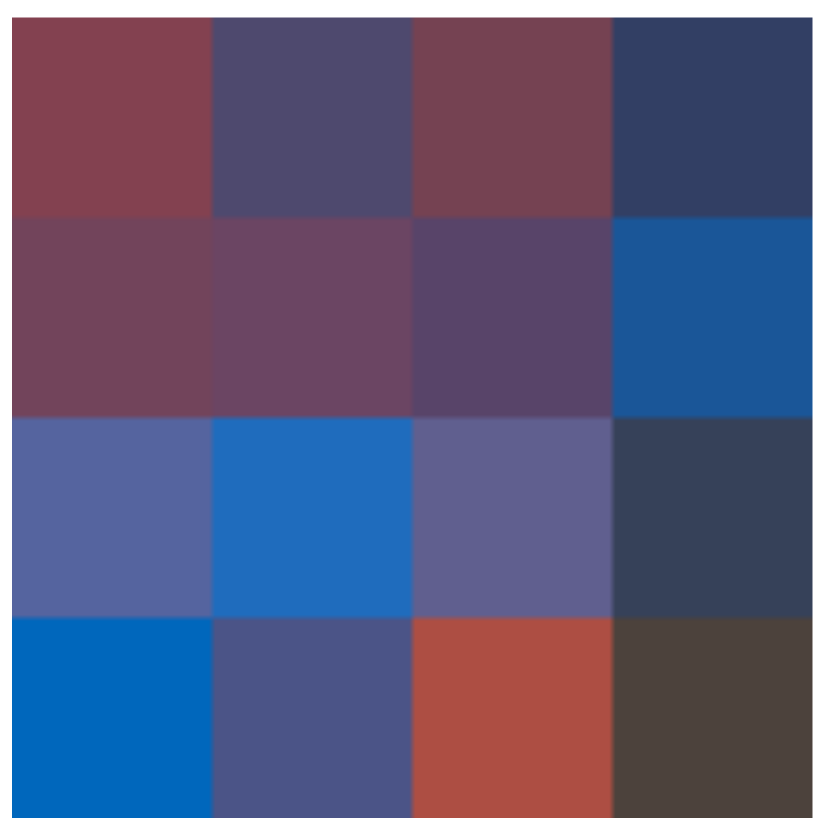

Level 0 = 128x128

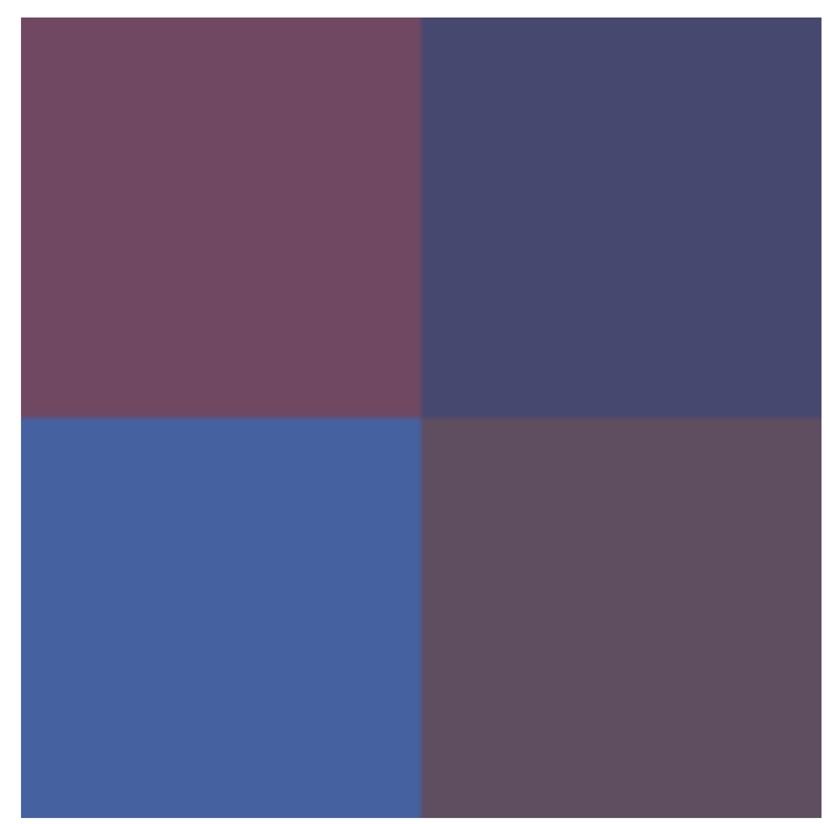

Level 1 = 64x64


Level 2 = 32x32


Level 3 = 16x16


Level 4 = 8x8

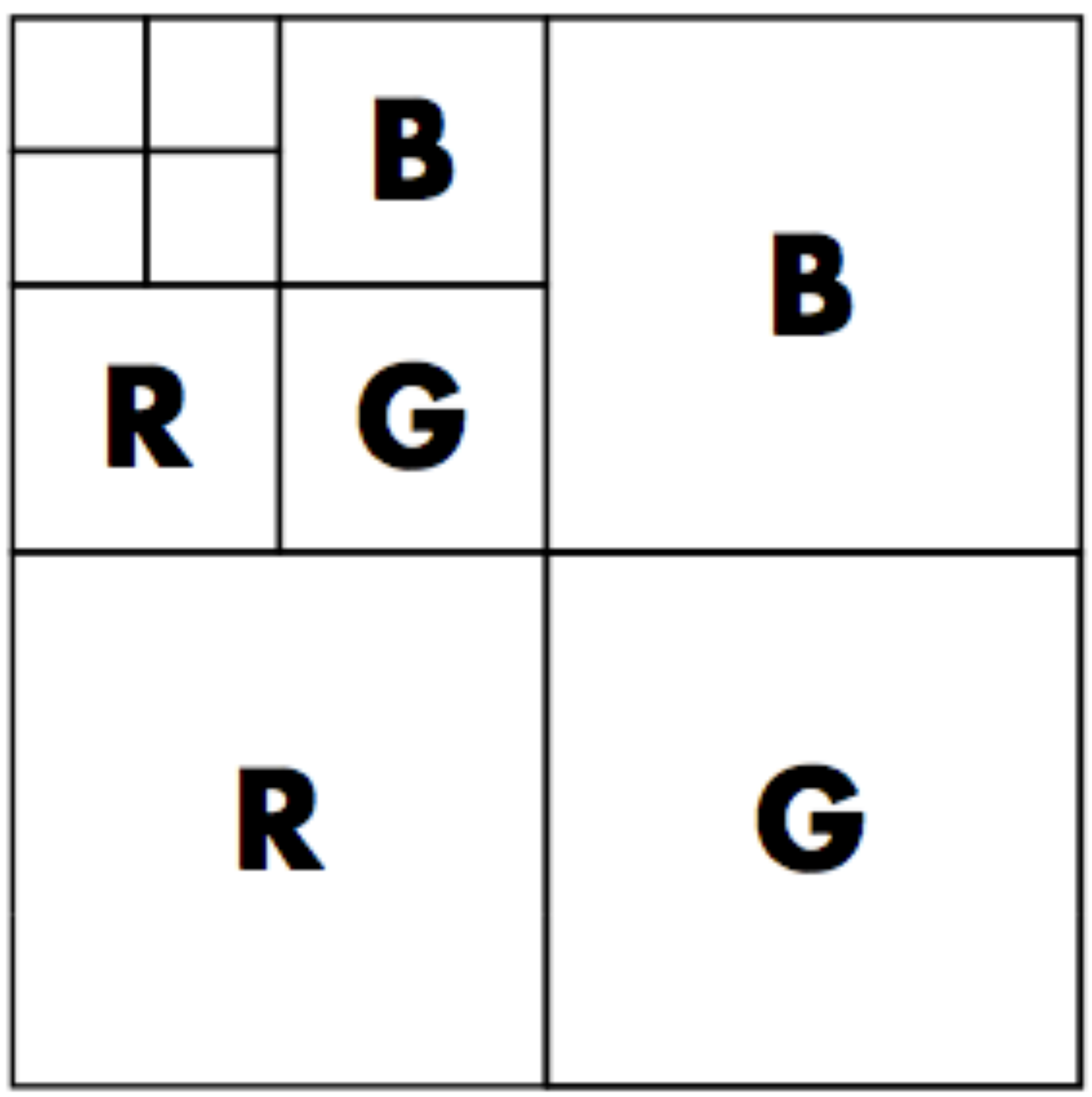

Level 5 = 4x4


Level 6 = 2x2

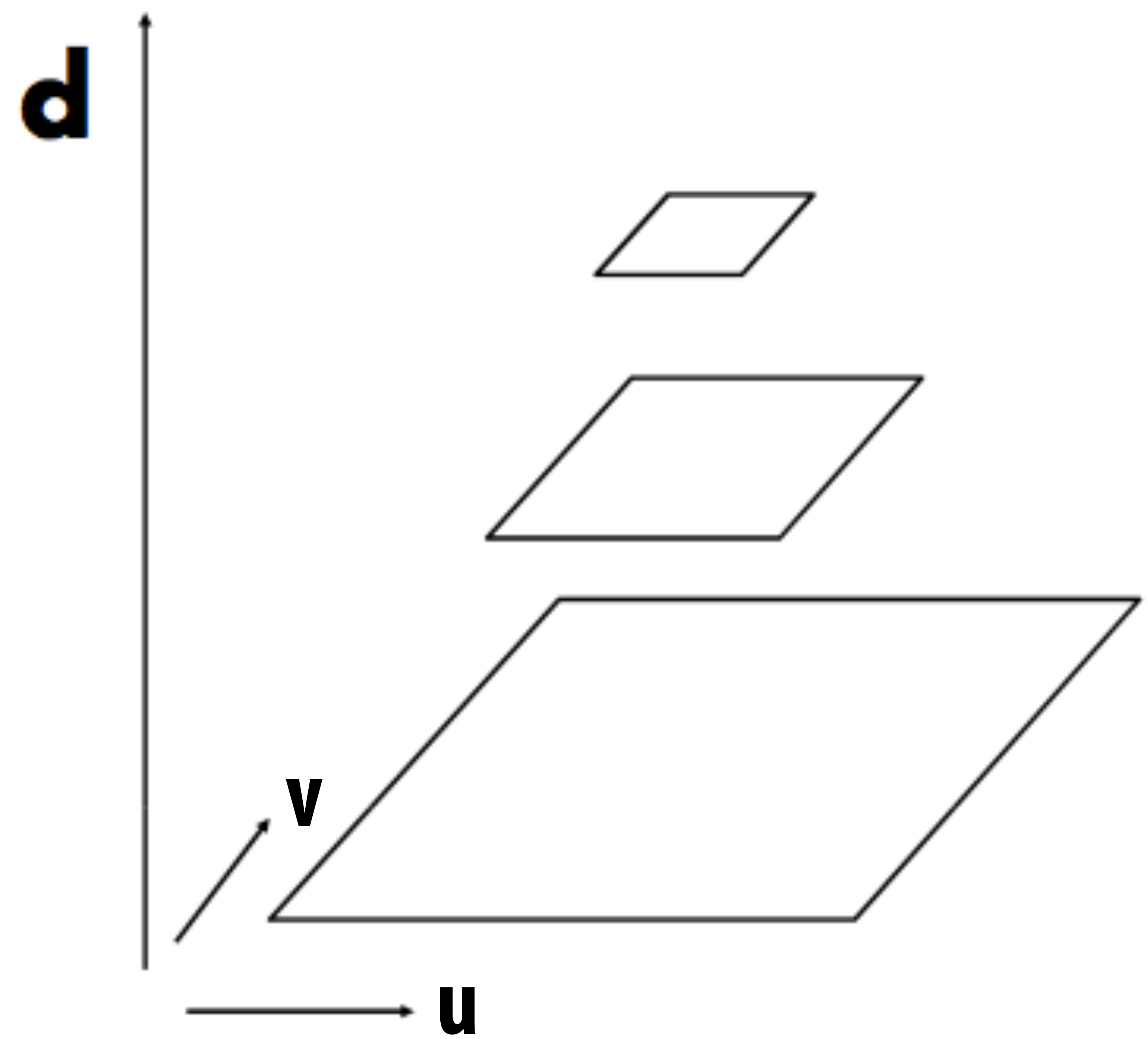

Level 7 = 1x1

“Mip” comes from the Latin “multum in parvo", meaning a multitude in a small space

Mipmap (L. Williams 83)



Williams' original proposed
mip-map layout

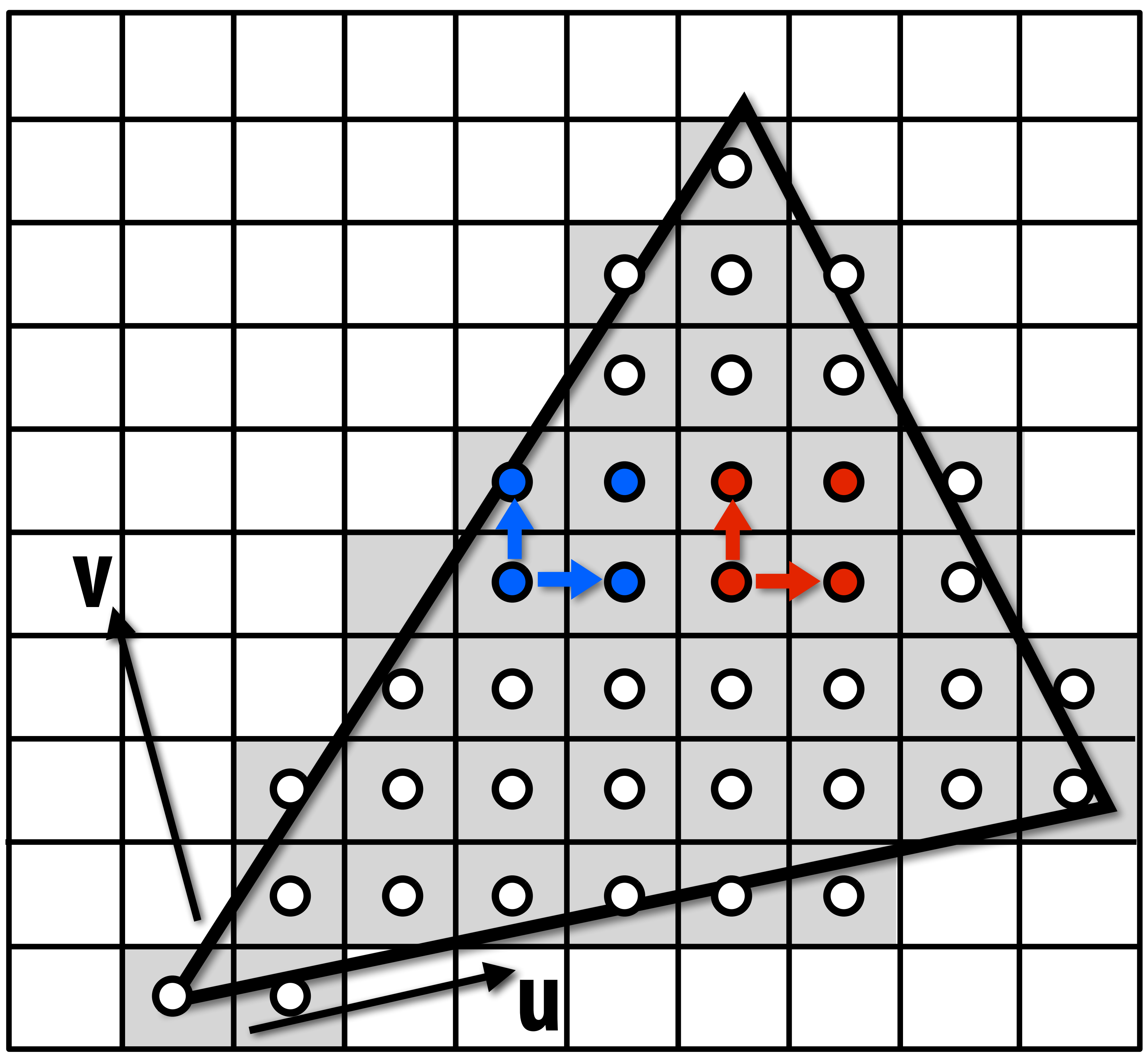


"Mip hierarchy"
level = *d*

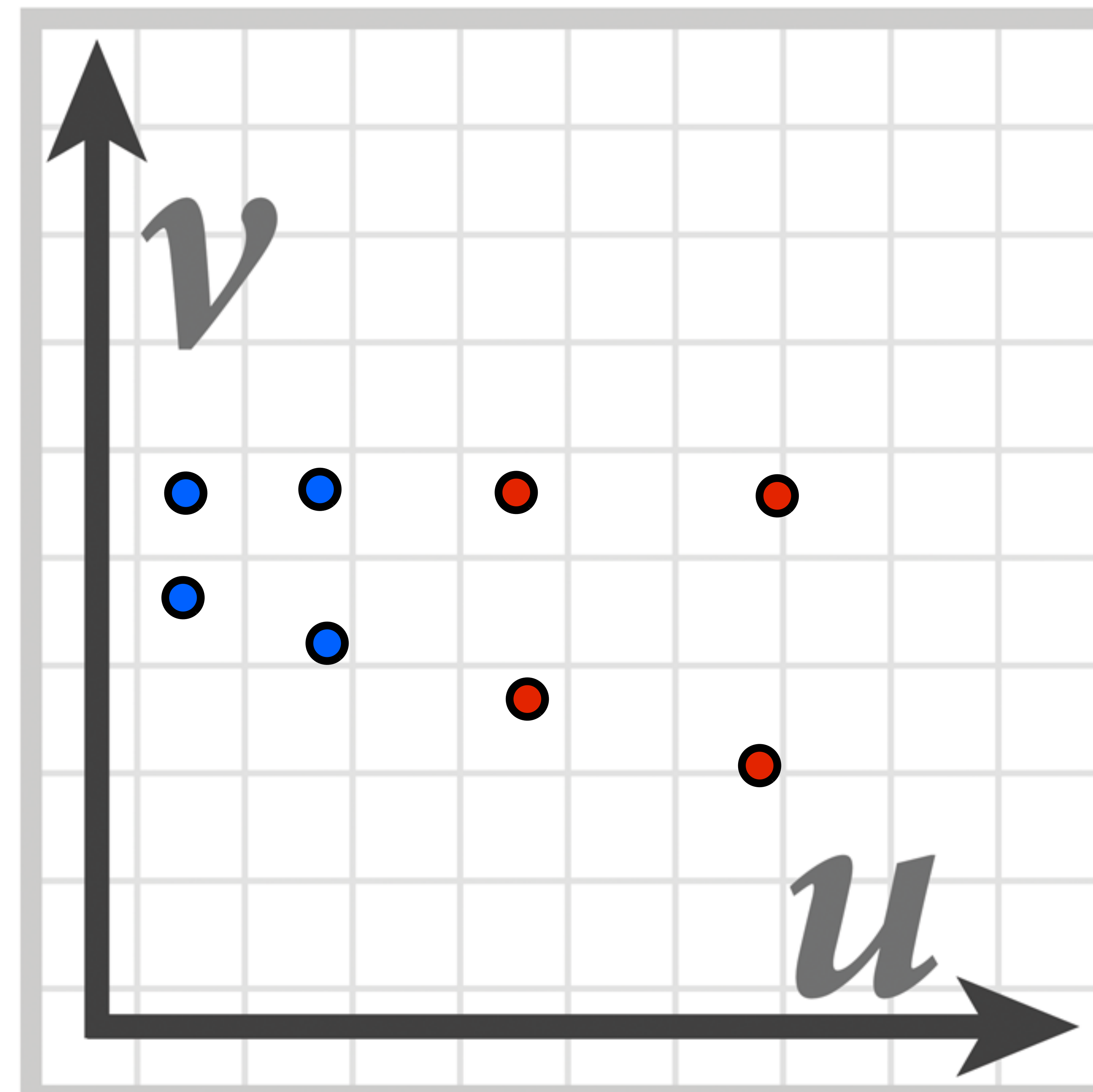
What is the storage overhead of a mipmap?

Computing mipmap level

Compute differences between texture coordinate values of neighboring screen samples



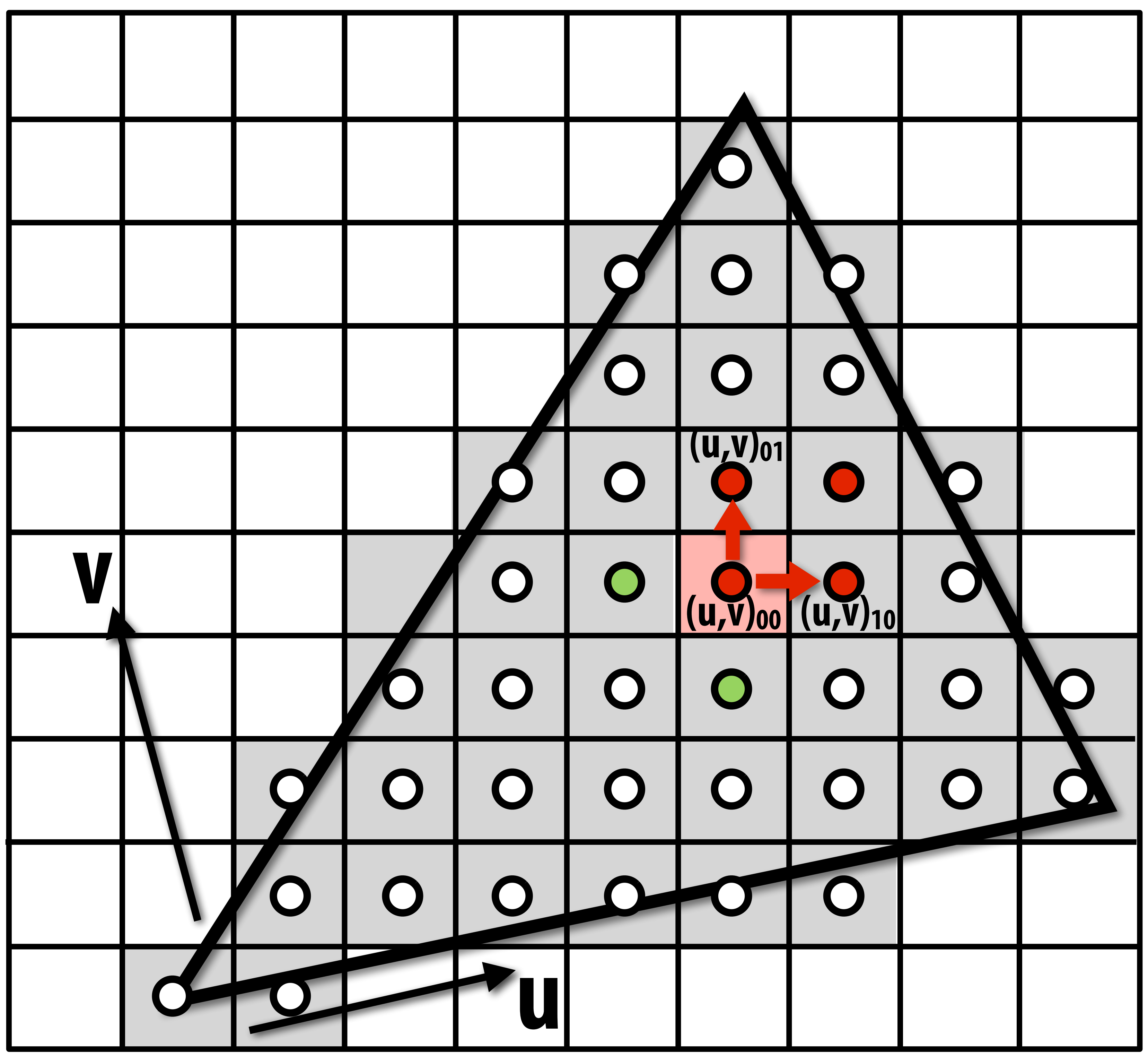
Screen space

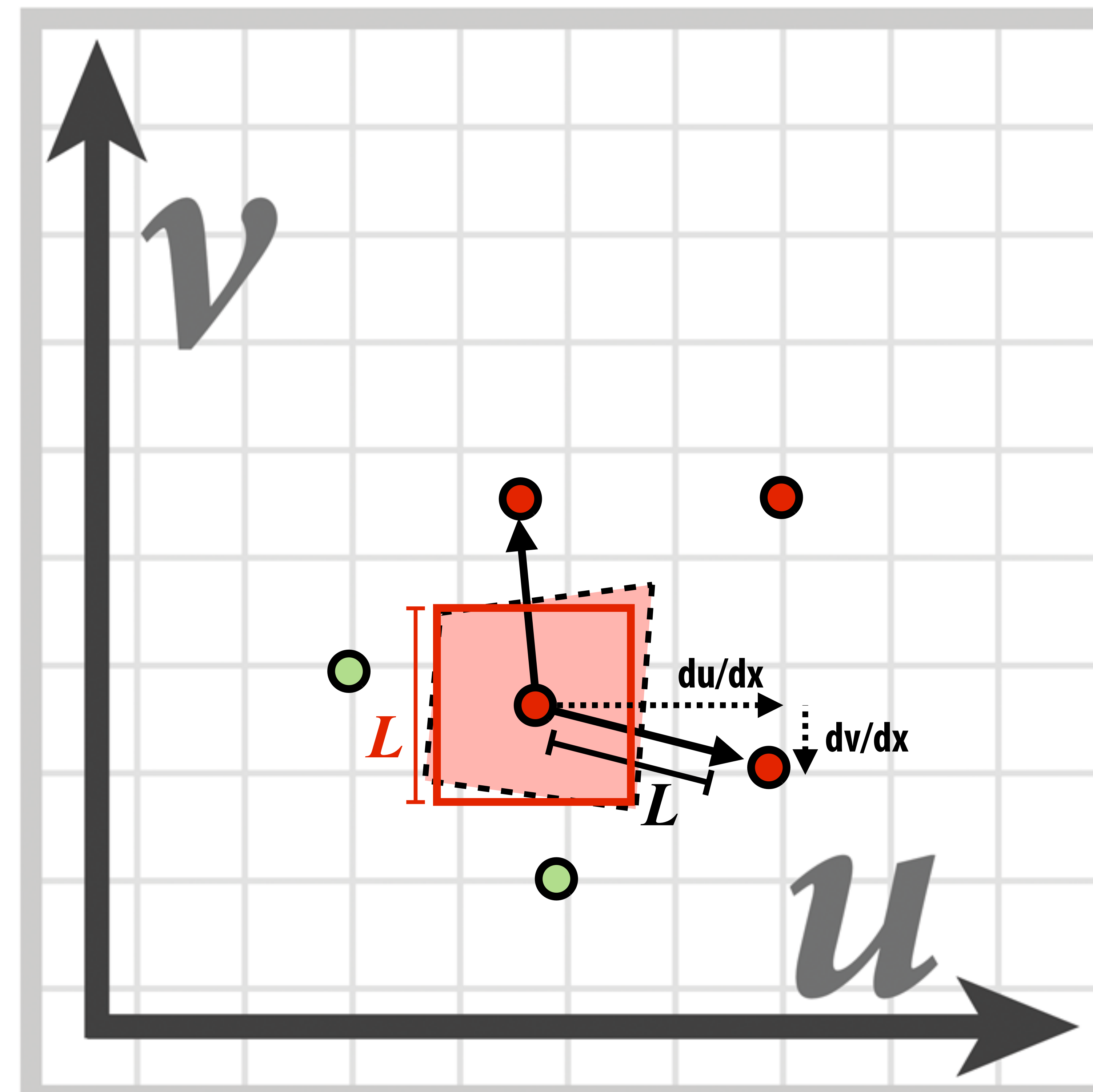


Texture space

Computing mipmap level

Compute differences between texture coordinate values of neighboring screen samples





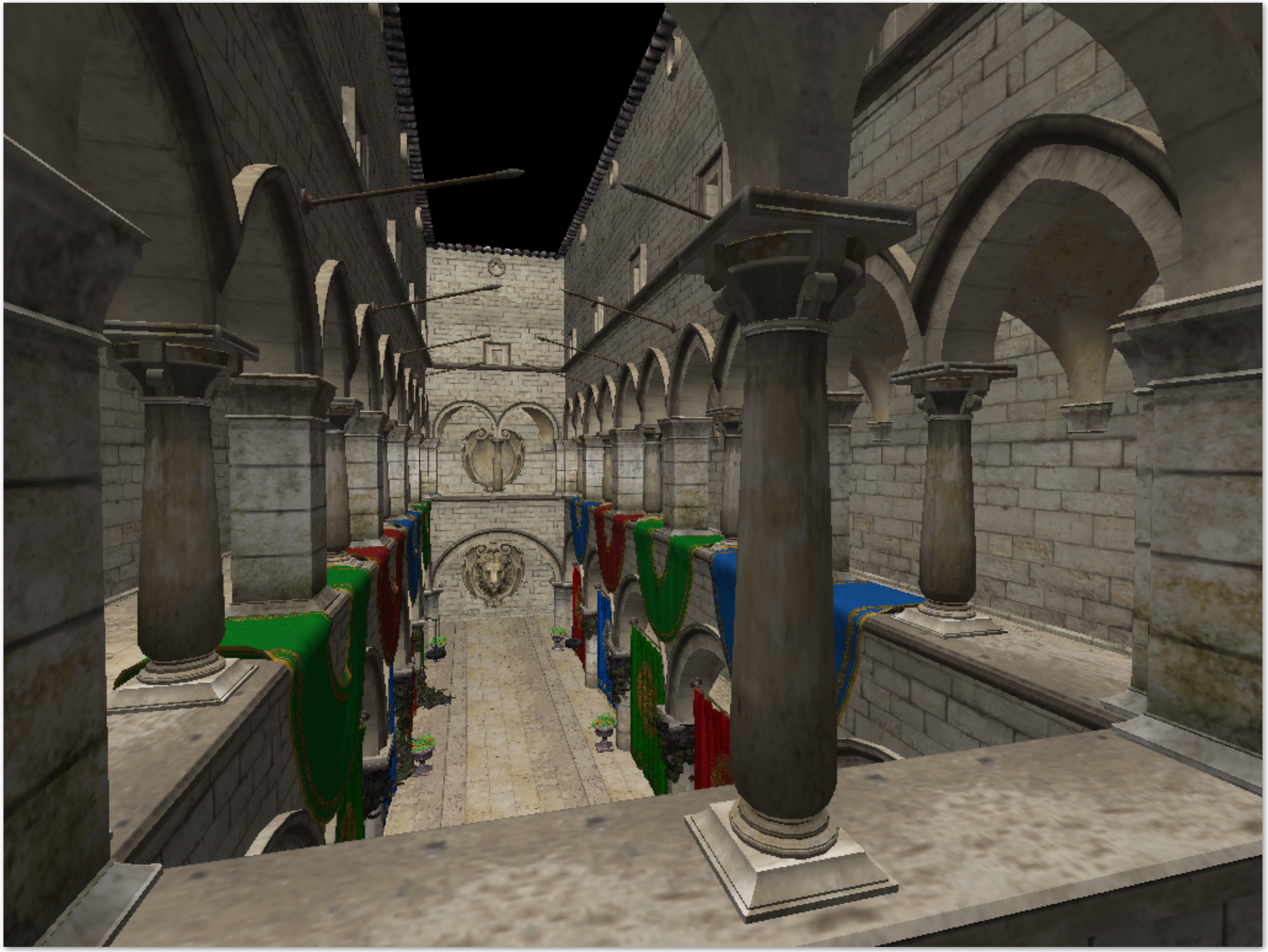
$$du/dx = u_{10} - u_{00} \qquad dv/dx = v_{10} - v_{00}$$
$$du/dy = u_{01} - u_{00} \qquad dv/dy = v_{01} - v_{00}$$

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

$$mip\text{-}map\ d = \log_2 L$$

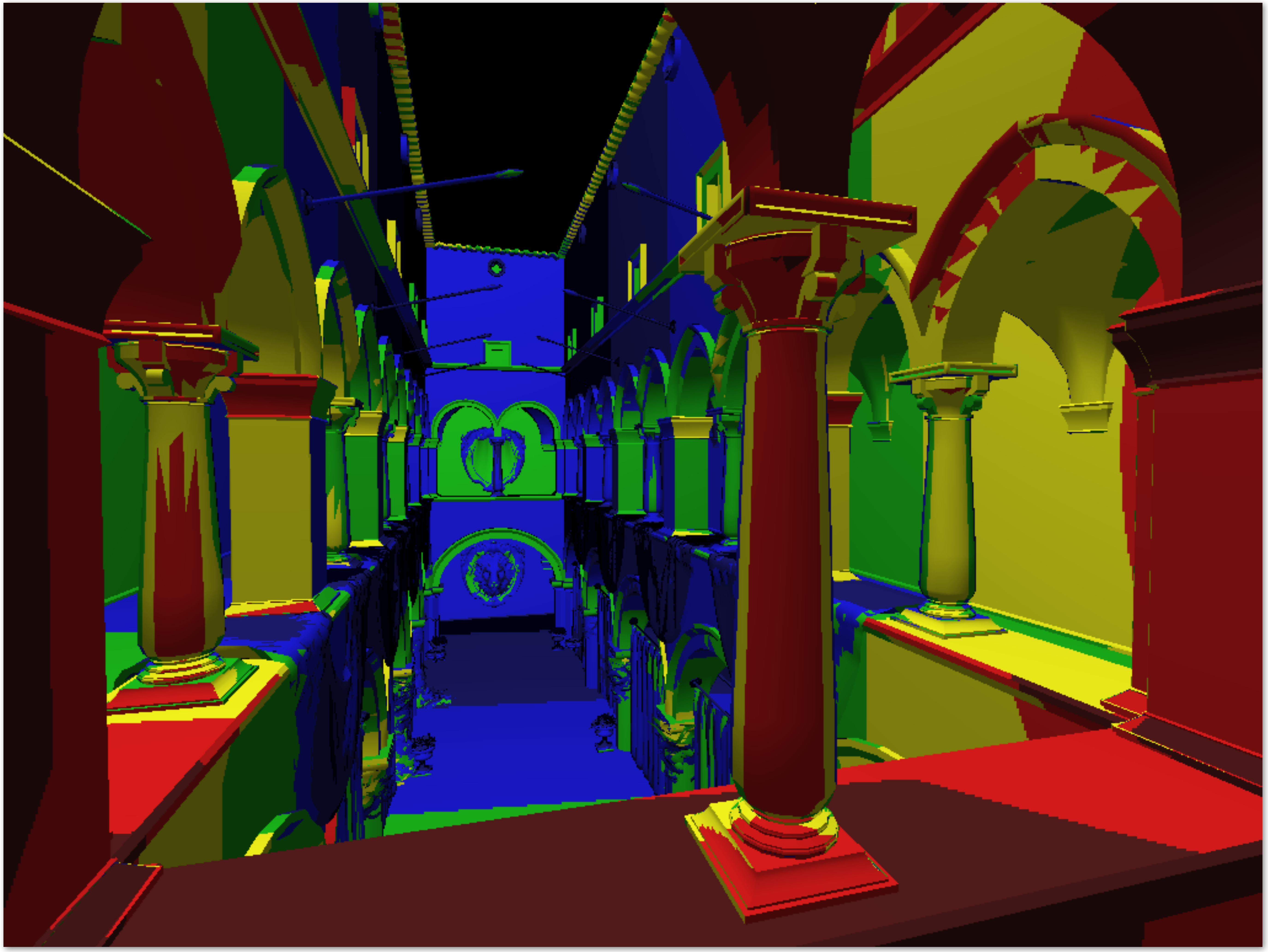
Bilinear resampling at level 0

Bilinear resampling at level 2

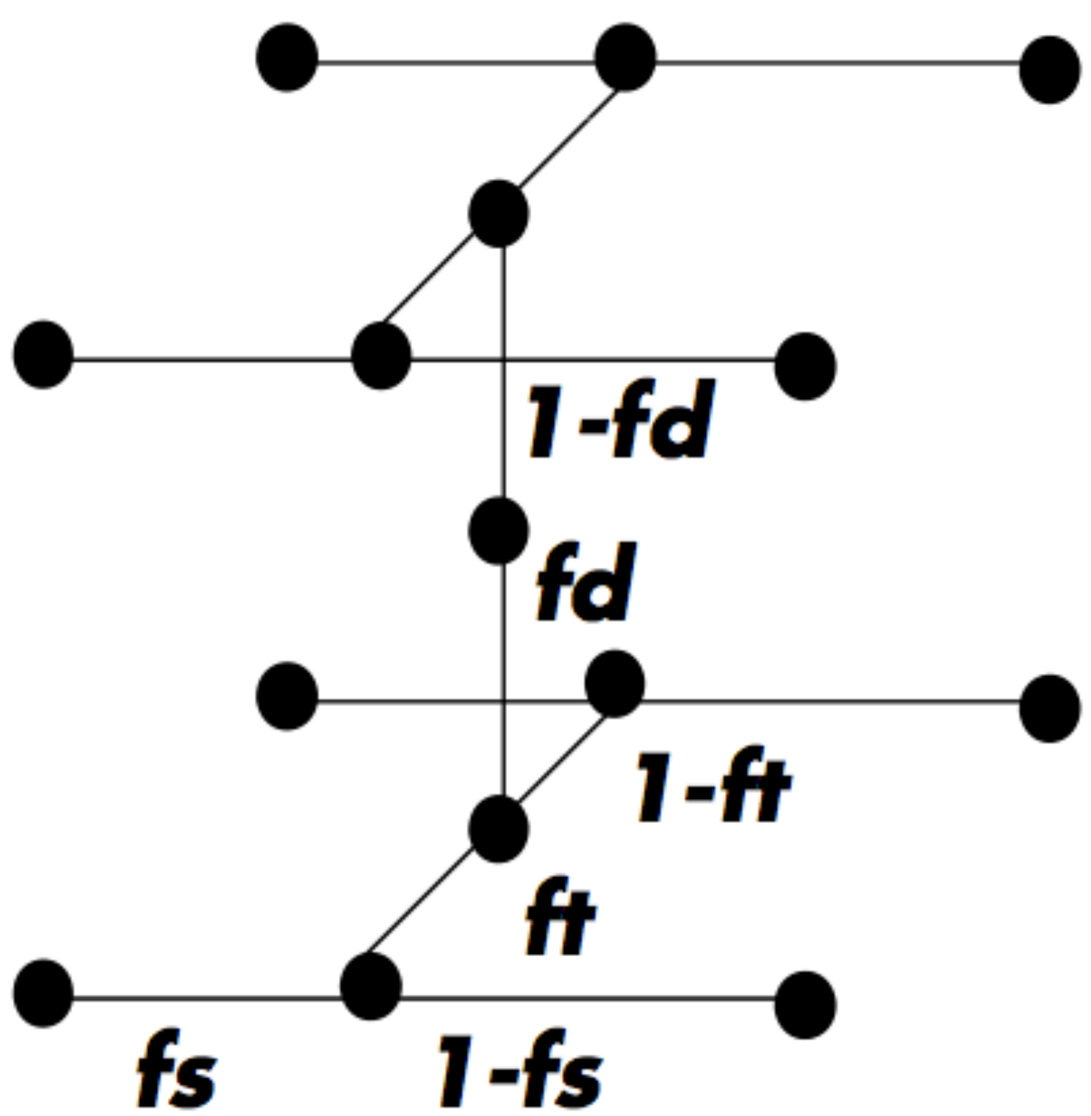


Bilinear resampling at level 4

Visualization of mipmap level
(bilinear filtering only: *d* clamped to nearest level)



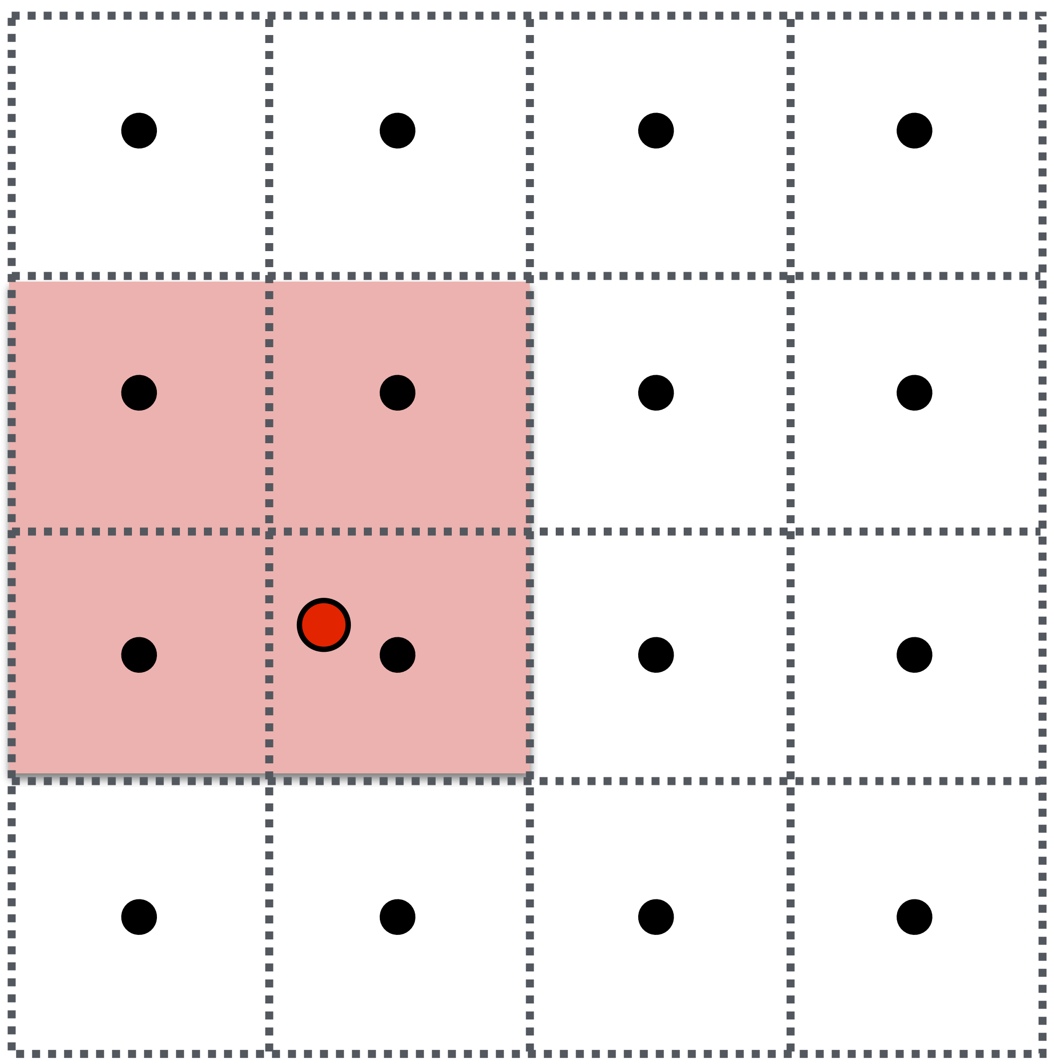
"Tri-linear" filtering



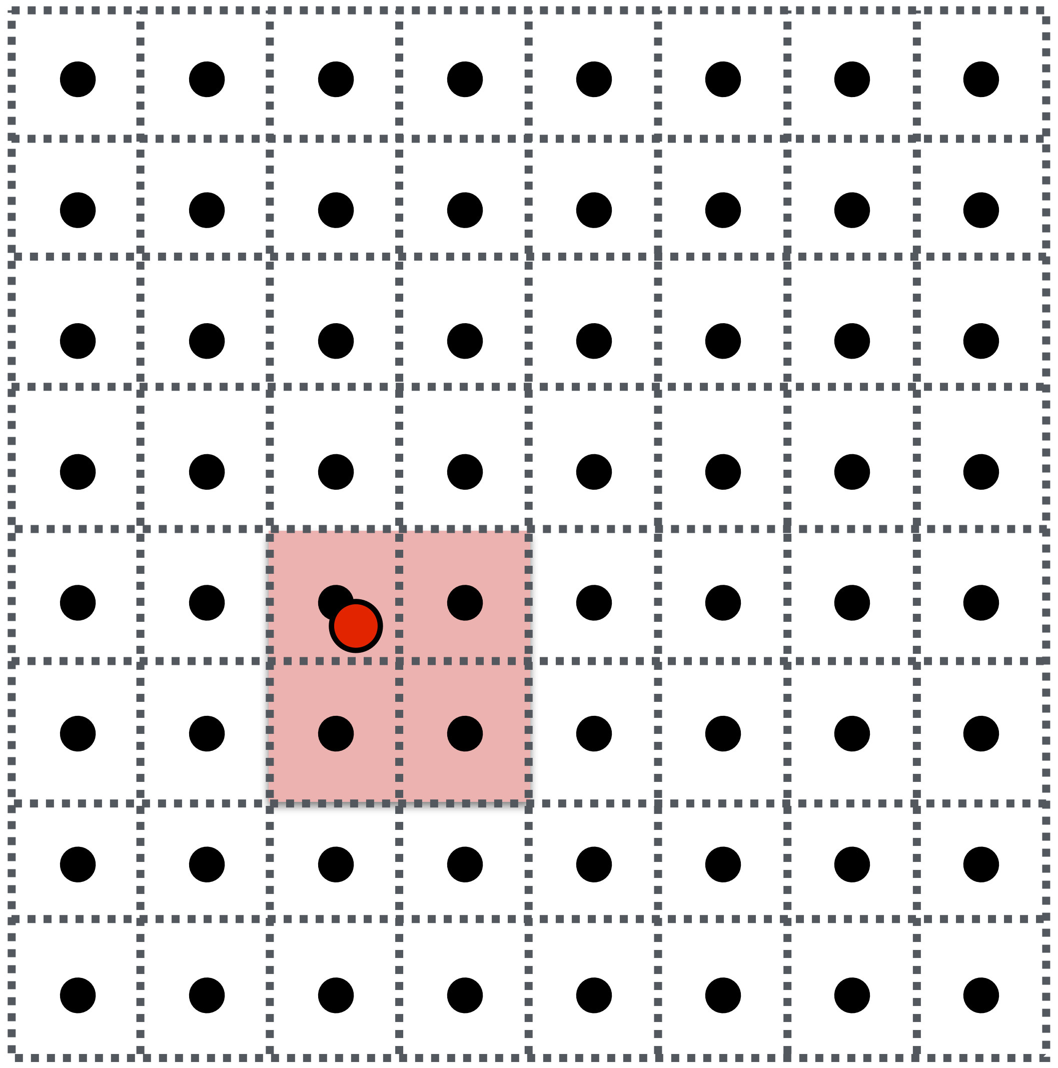
$$lerp(t, v_1, v_2) = v_1 + t(v_2 - v_1)$$

Bilinear resampling:
- four texel reads
- 3 lerps (3 mul + 6 add)

Trilinear resampling:
- eight texel reads
- 7 lerps (7 mul + 14 add)



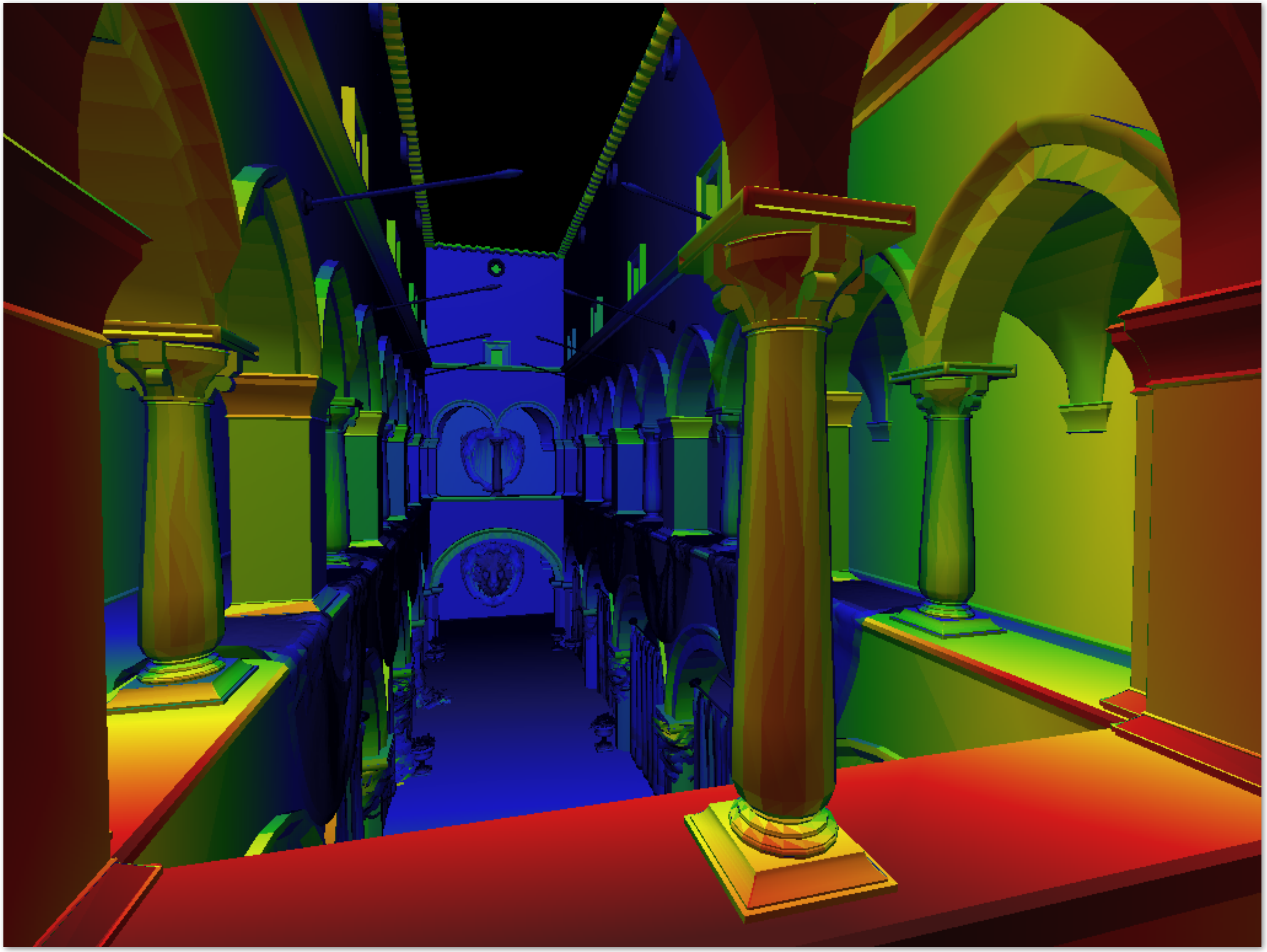
mip-map texels: level d+1



mip-map texels: level d

Figure credit: Akeley and Hanrahan

Visualization of mipmap level
(trilinear filtering: visualization of continuous *d*)

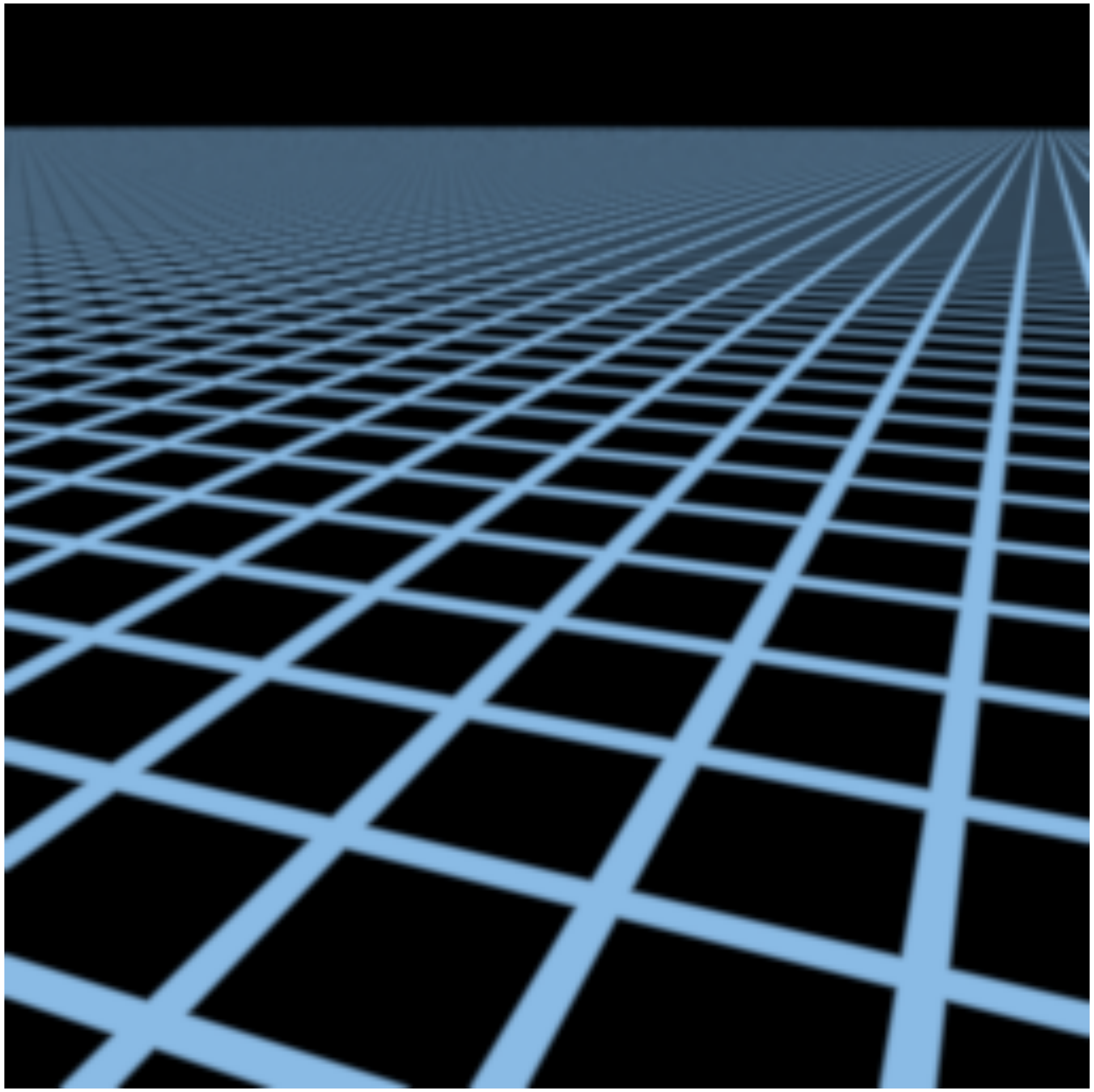


Bilinear vs trilinear filtering cost

- **Bilinear resampling:**
  - 4 texel reads
  - 3 lerps (3 mul + 6 add)

- **Trilinear resampling:**
  - 8 texel reads
  - 7 lerps (7 mul + 14 add)

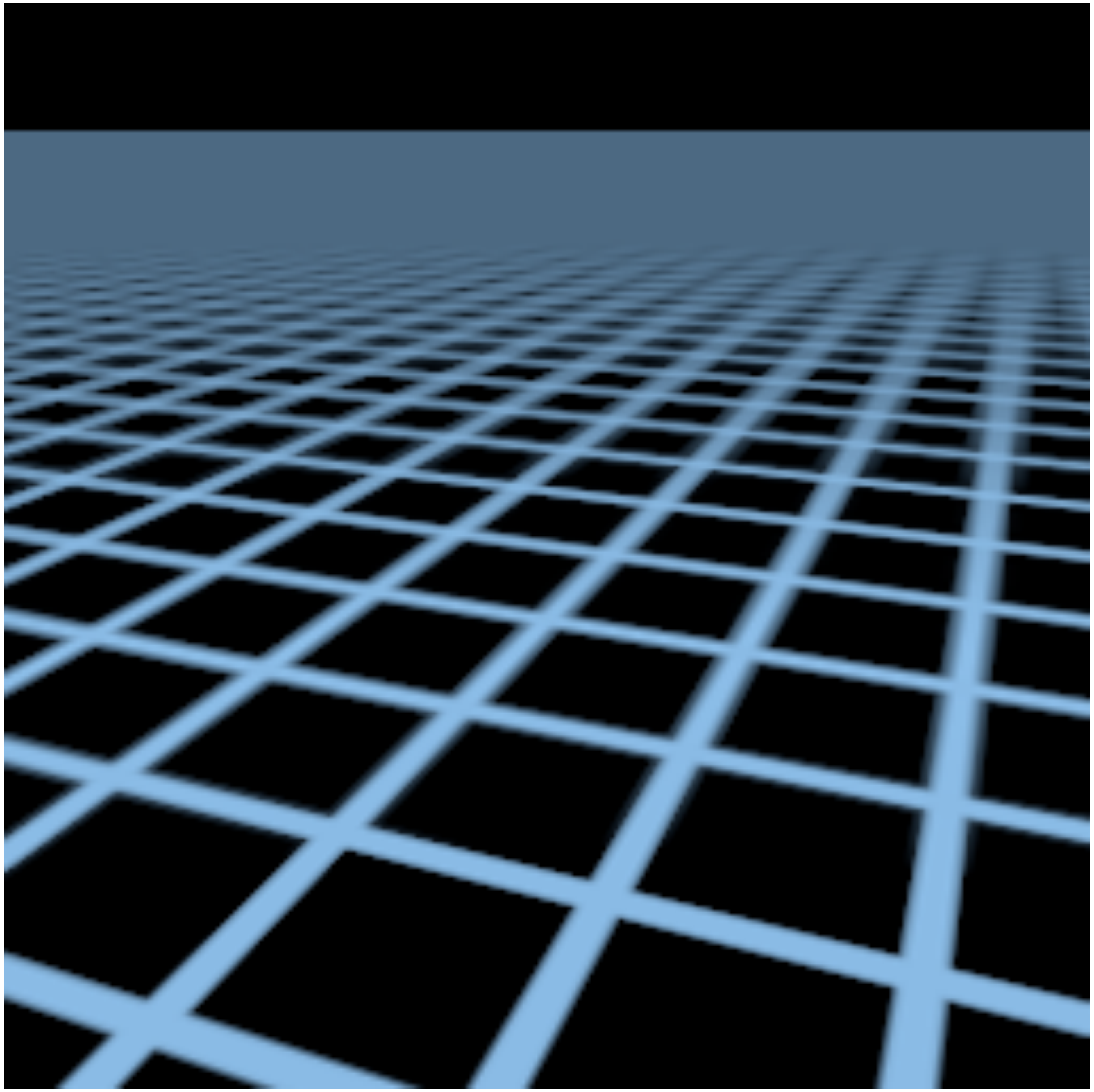
Example: mipmap limitations



Supersampling 512x
(desired answer)

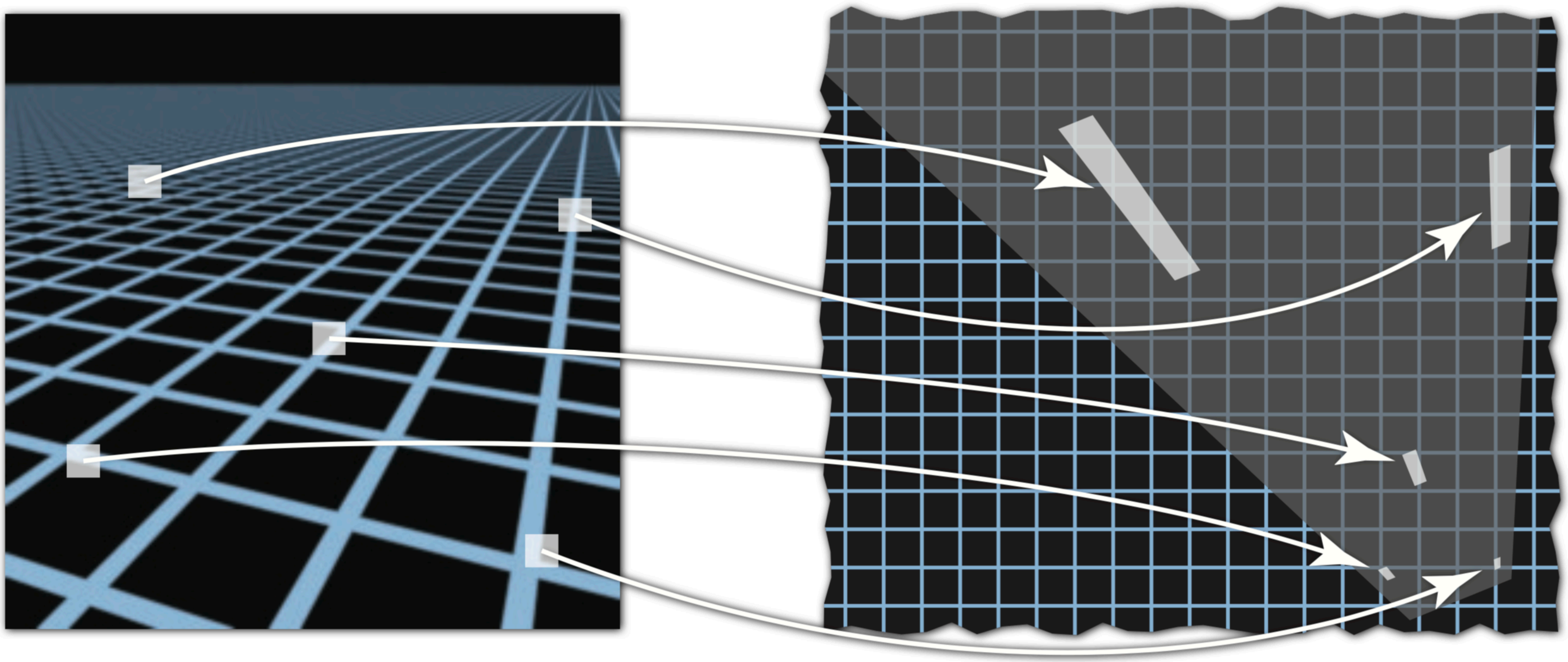
Example: mipmap limitations

Overblur
Why?



Mipmap trilinear sampling

Screen pixel footprint in texture space



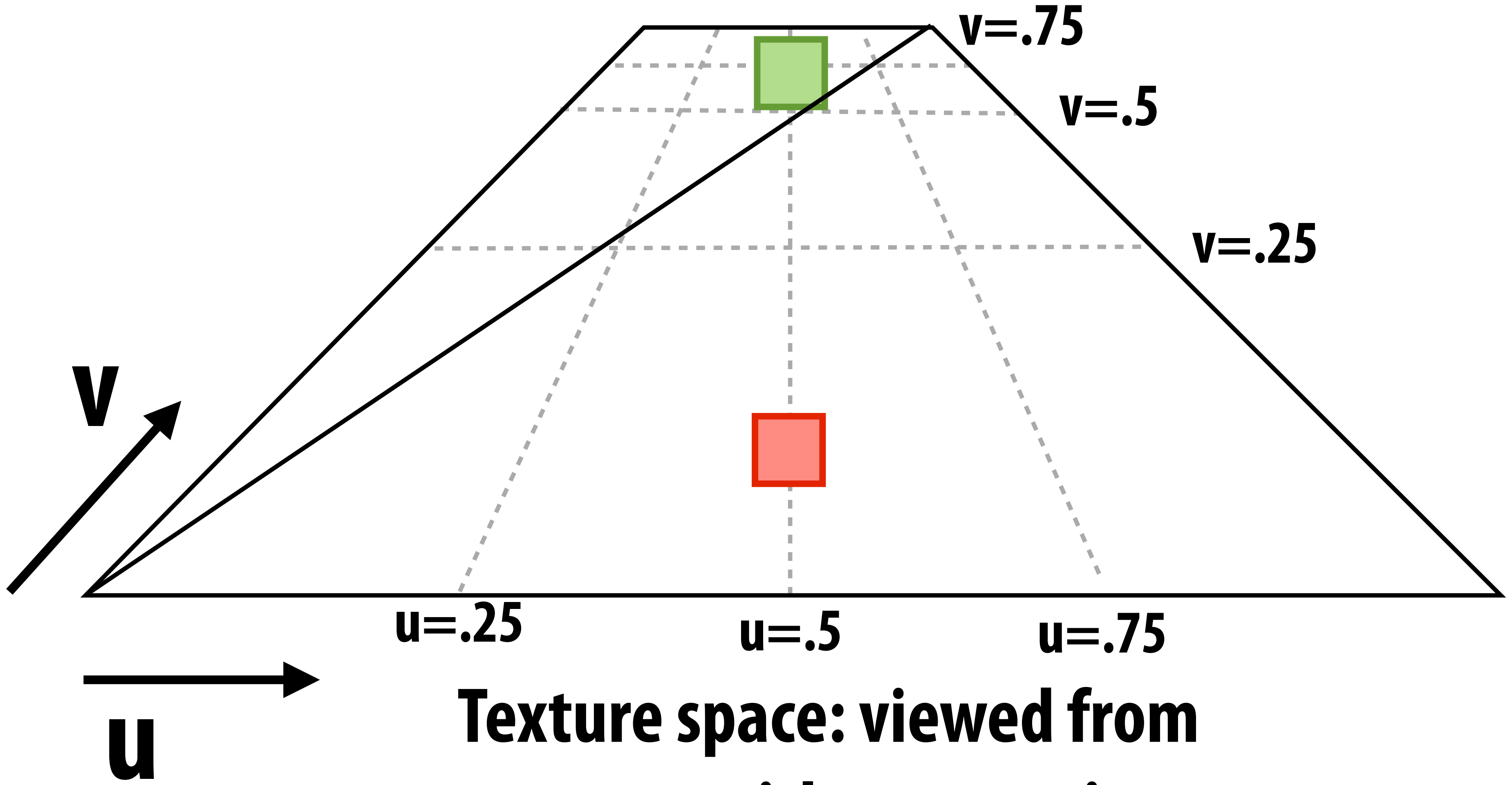
Screen space

Texture space

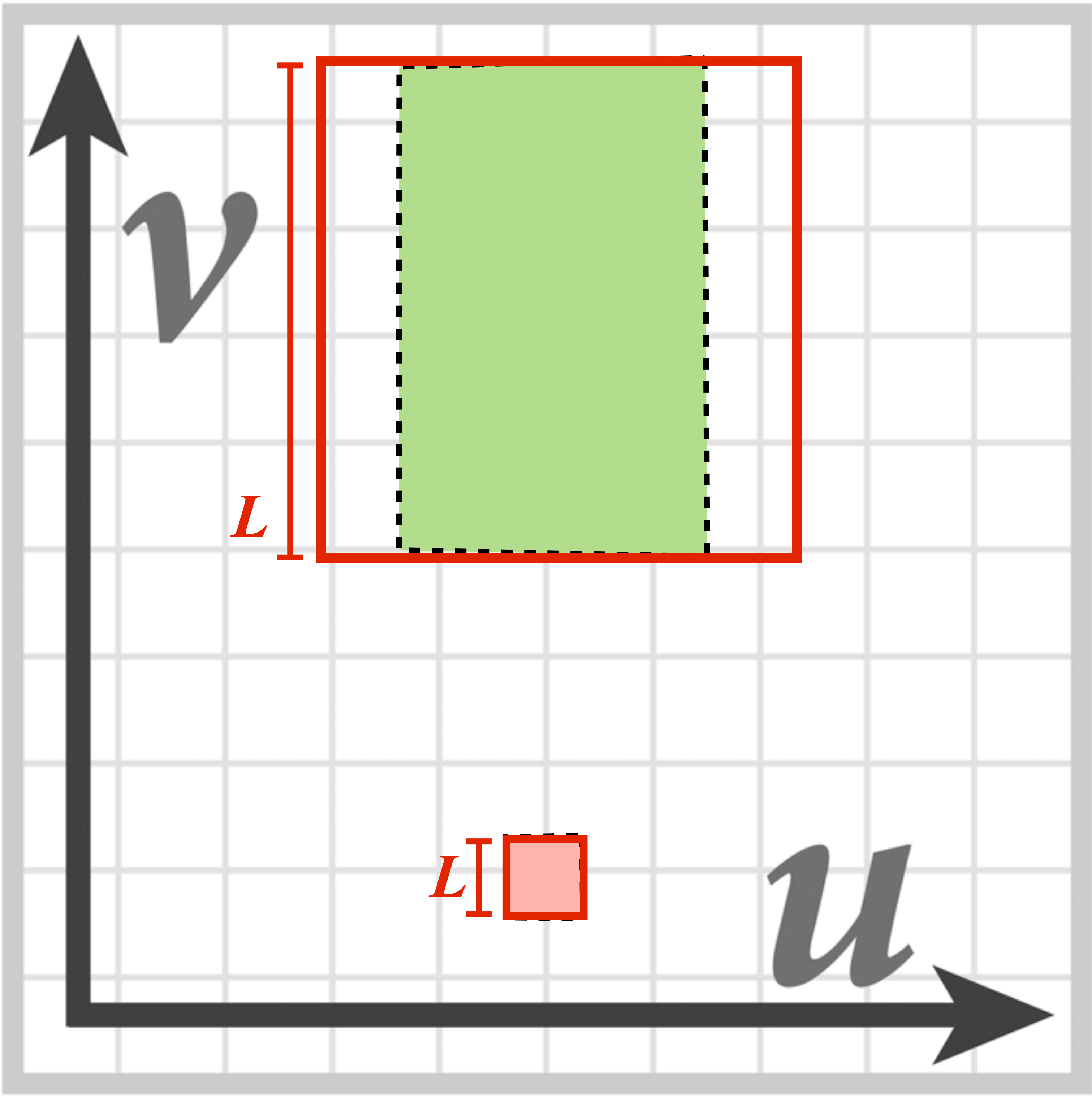
Texture sampling pattern not rectilinear or isotropic

Pixel area may not map to isotropic region in texture space

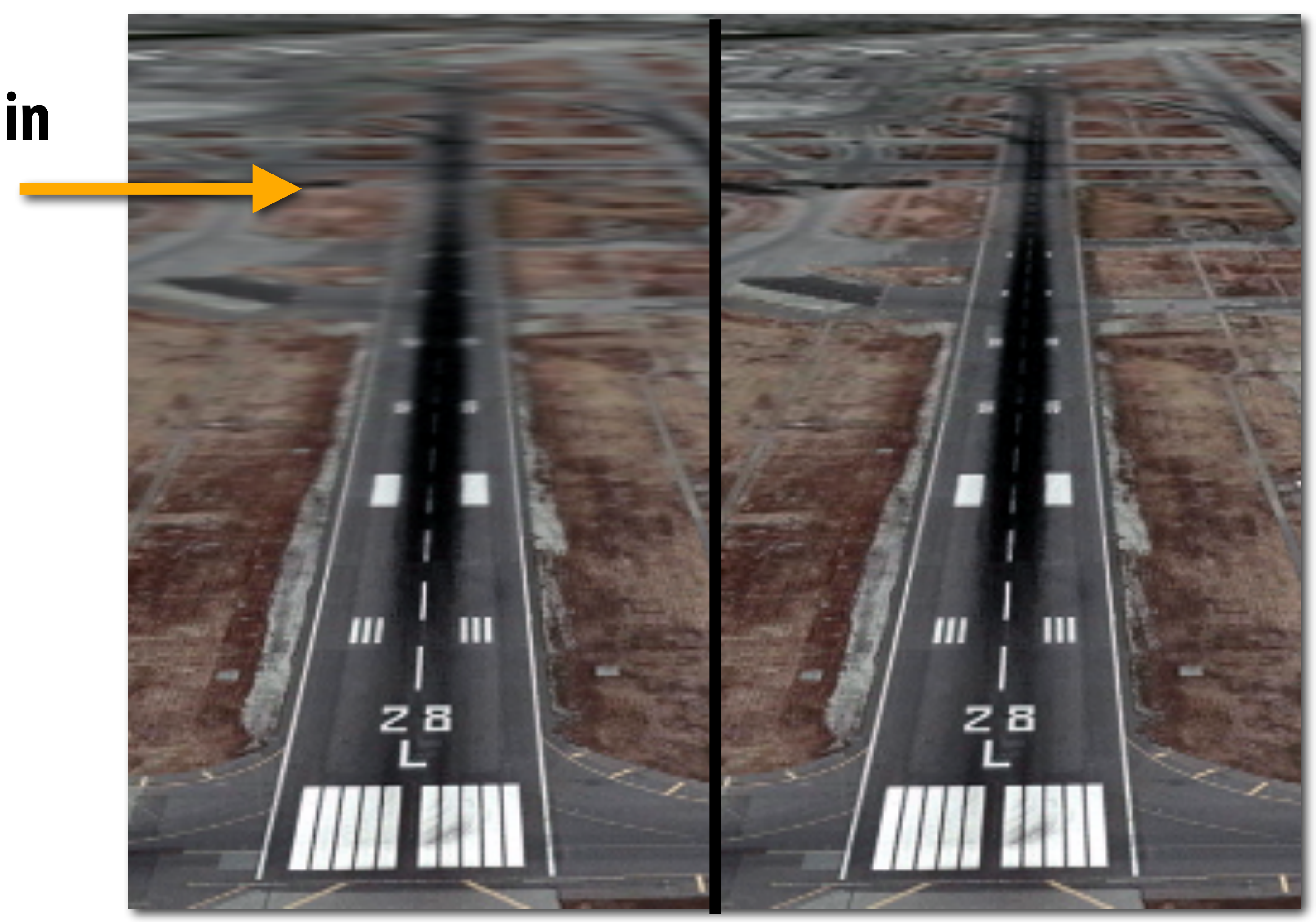
Proper filtering requires anisotropic filter footprint



Texture space: viewed from camera with perspective



Overblurring in *u* direction

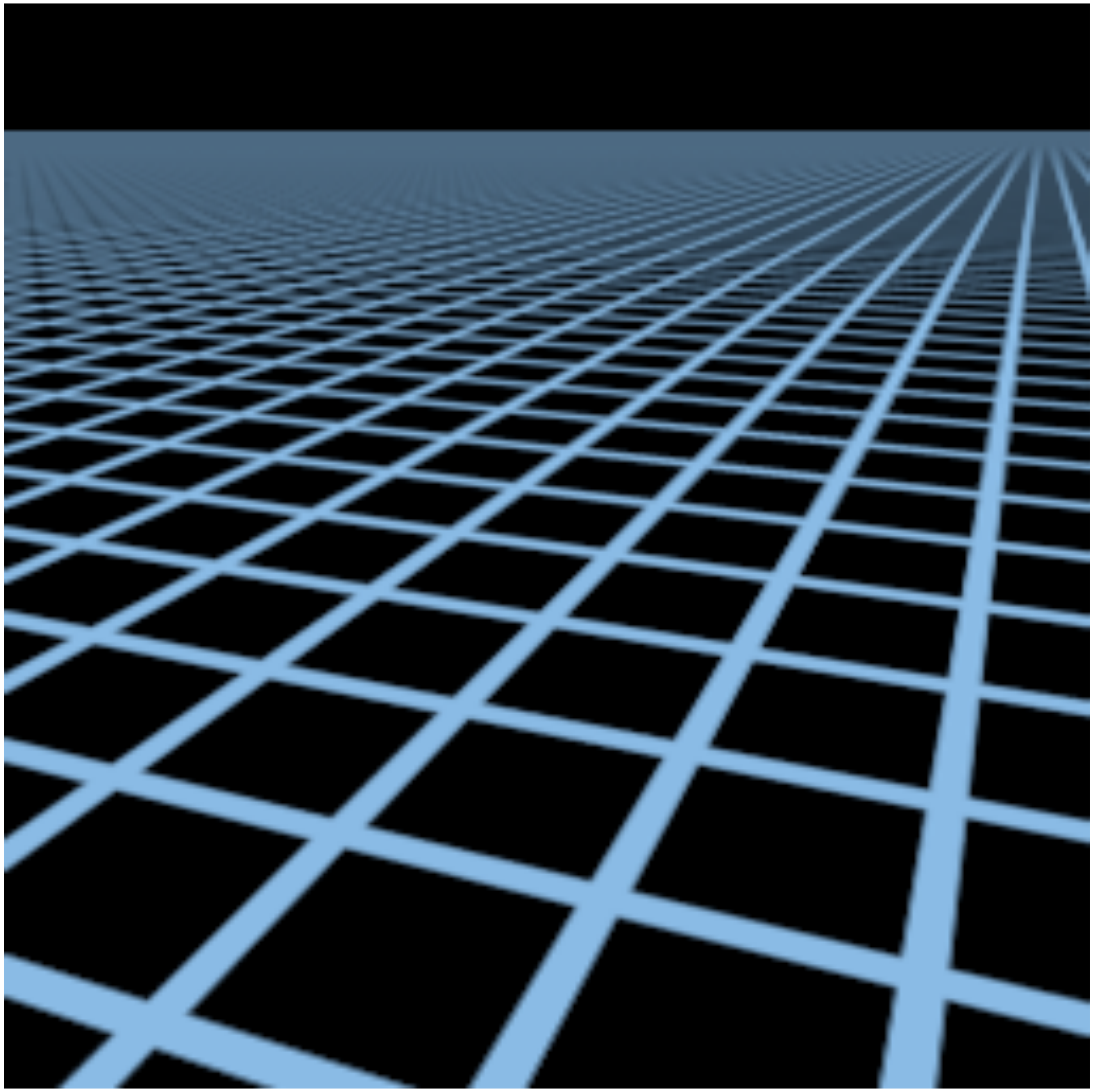


Trilinear (Isotropic) Filtering

Anisotropic Filtering

(Modern anisotropic texture filtering solutions combine multiple mip map samples)

Anisotropic filtering



Elliptical weighted average (EWA) filtering
(uses multiple lookups into mip-map to approximate filter region)

Summary: texture filtering using the mip map

- **Small storage overhead (33%)**
  - Mipmap is 4/3 the size of original texture image
- **For each isotropically-filtered sampling operation**
  - Constant filtering cost (independent of mip map level)
  - Constant number of texels accessed (independent of mip map level)
- **Combat aliasing with *prefiltering*, rather than supersampling**
  - Recall: we used supersampling to address aliasing problem when sampling coverage
- **Bilinear/trilinear filtering is isotropic and thus will "overblur" to avoid aliasing**
  - Anisotropic texture filtering provides higher image quality at higher compute and memory bandwidth cost (in practice: multiple mip map samples)

A full texture sampling operation

1. Compute u and v from screen sample x,y (via evaluation of attribute equations)
2. Compute du/dx, du/dy, dv/dx, dv/dy differentials from screen-adjacent samples.
3. Compute mip map level *d*
4. Convert normalized [0,1] texture coordinate (u,v) to texture coordinates U,V in [W,H]
5. Compute required texels in window of filter
6. Load required texels from memory (need eight texels for trilinear)
7. Perform tri-linear interpolation according to (U, V, d)

Takeaway: a texture sampling operation is not just an image pixel lookup! It involves a significant amount of math.

For this reason, modern GPUs have dedicated fixed-function hardware support for performing texture sampling operations.

Summary: texture mapping

- Texturing: used to add visual detail to surfaces that is not captured in geometry
- Texture coordinates: define mapping between points on triangle's surface (object coordinate space) to points in texture coordinate space
- Texture mapping is a sampling operation and is prone to aliasing
  - Solution: precompute and store multiple multiple resampled versions of the texture image (each with different amounts of low-pass filtering to remove increasing amounts of high frequency detail)
  - During rendering: dynamically select how much low-pass filtering is required based on distance between neighboring screen samples in texture space
    - Goal is to retain as much high-frequency content (detail) in the texture as possible, while avoiding aliasing

Acknowledgements

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