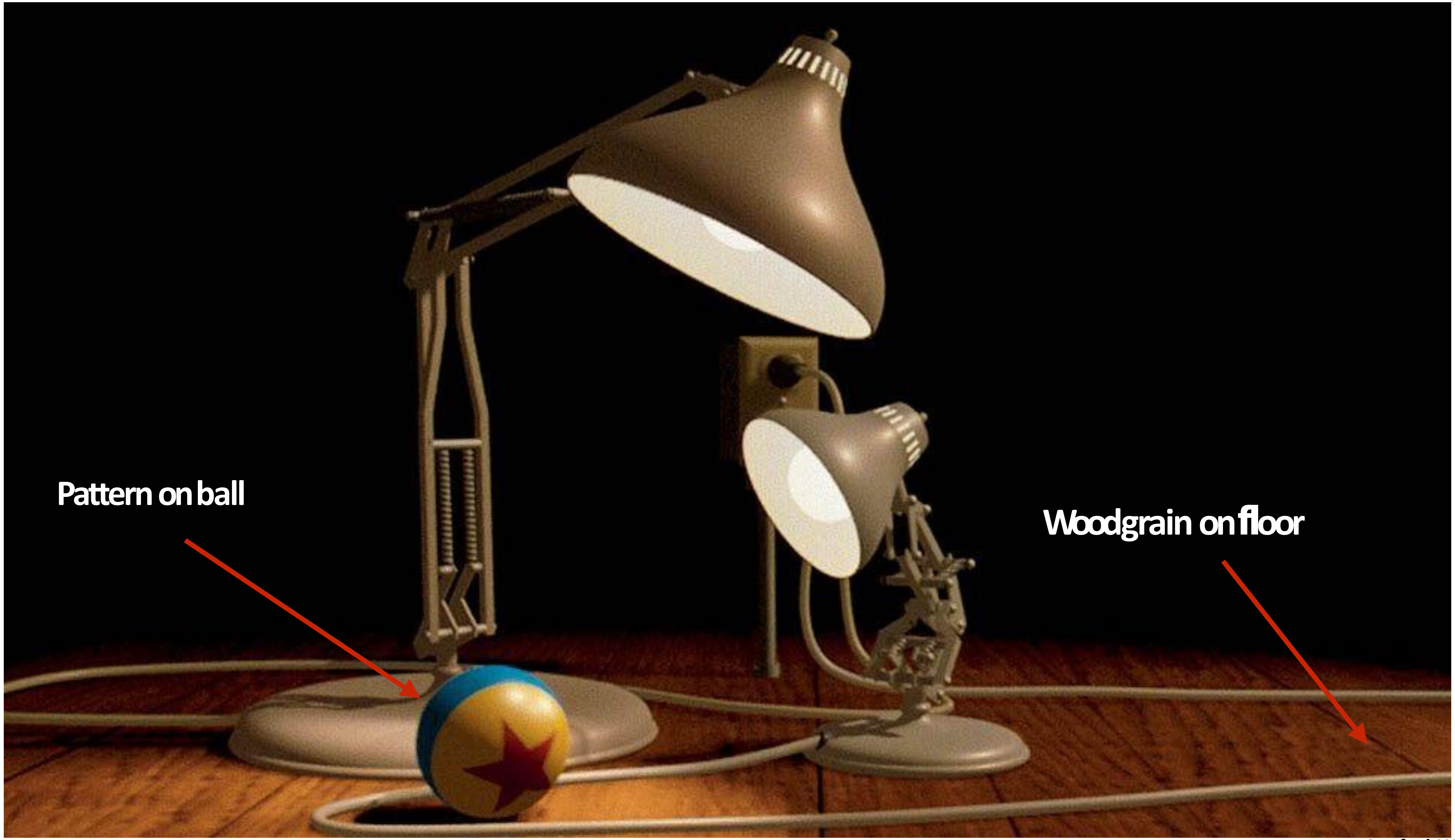
Texture Mapping

Many uses of texture mapping

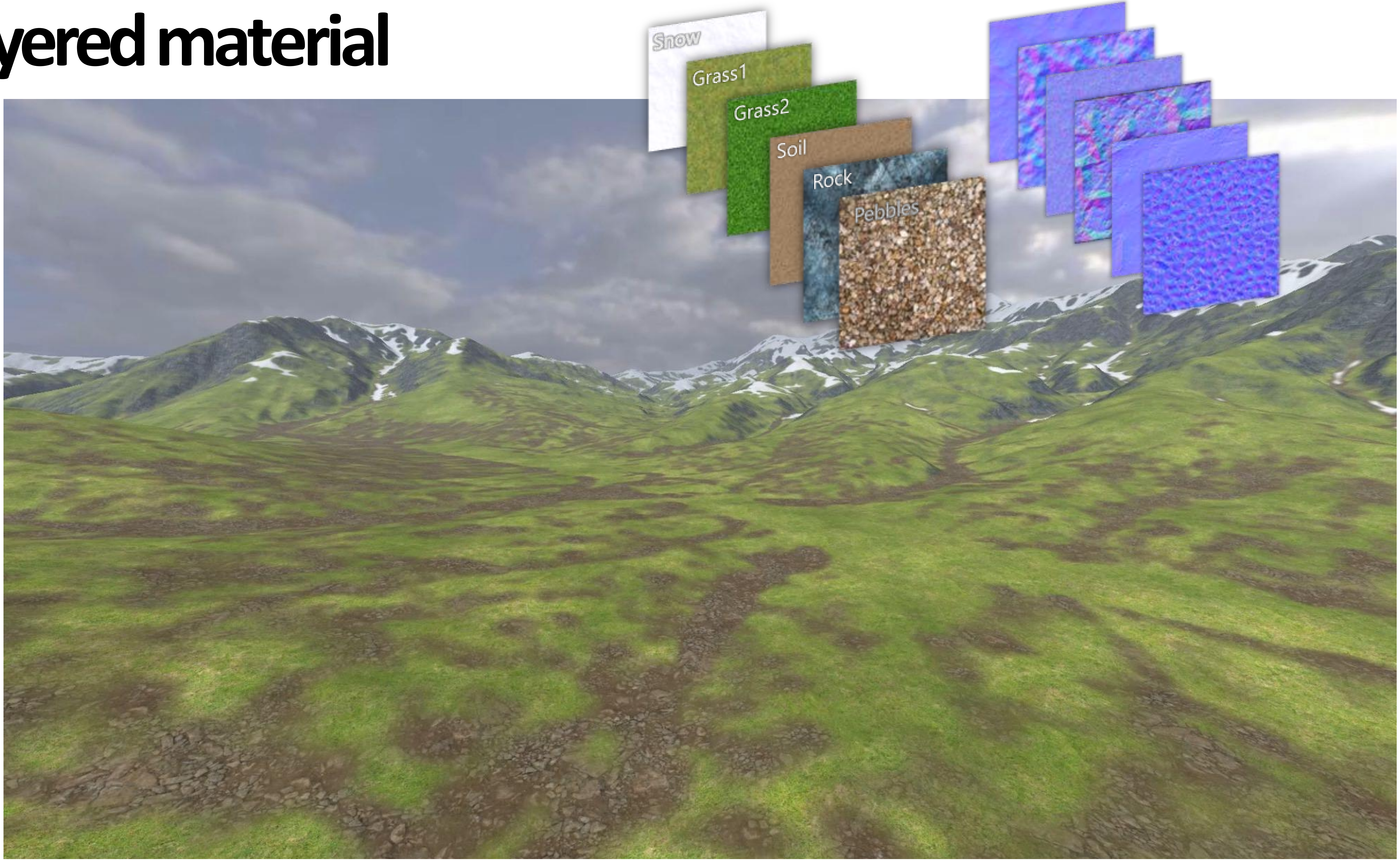
Define variation in surface reflectance



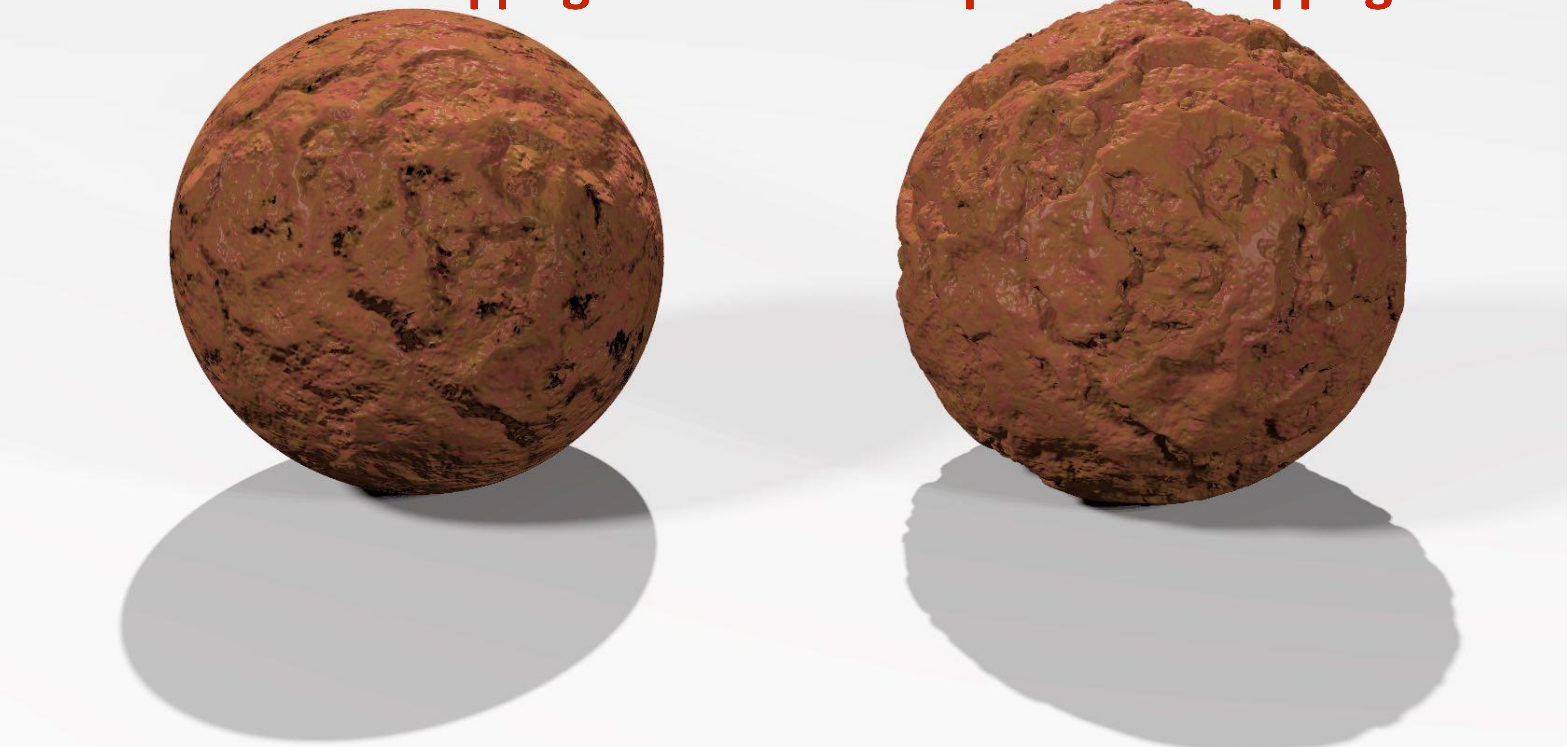
Describe surface material properties

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

Layered material



Normal and displacement mapping



normal mapping

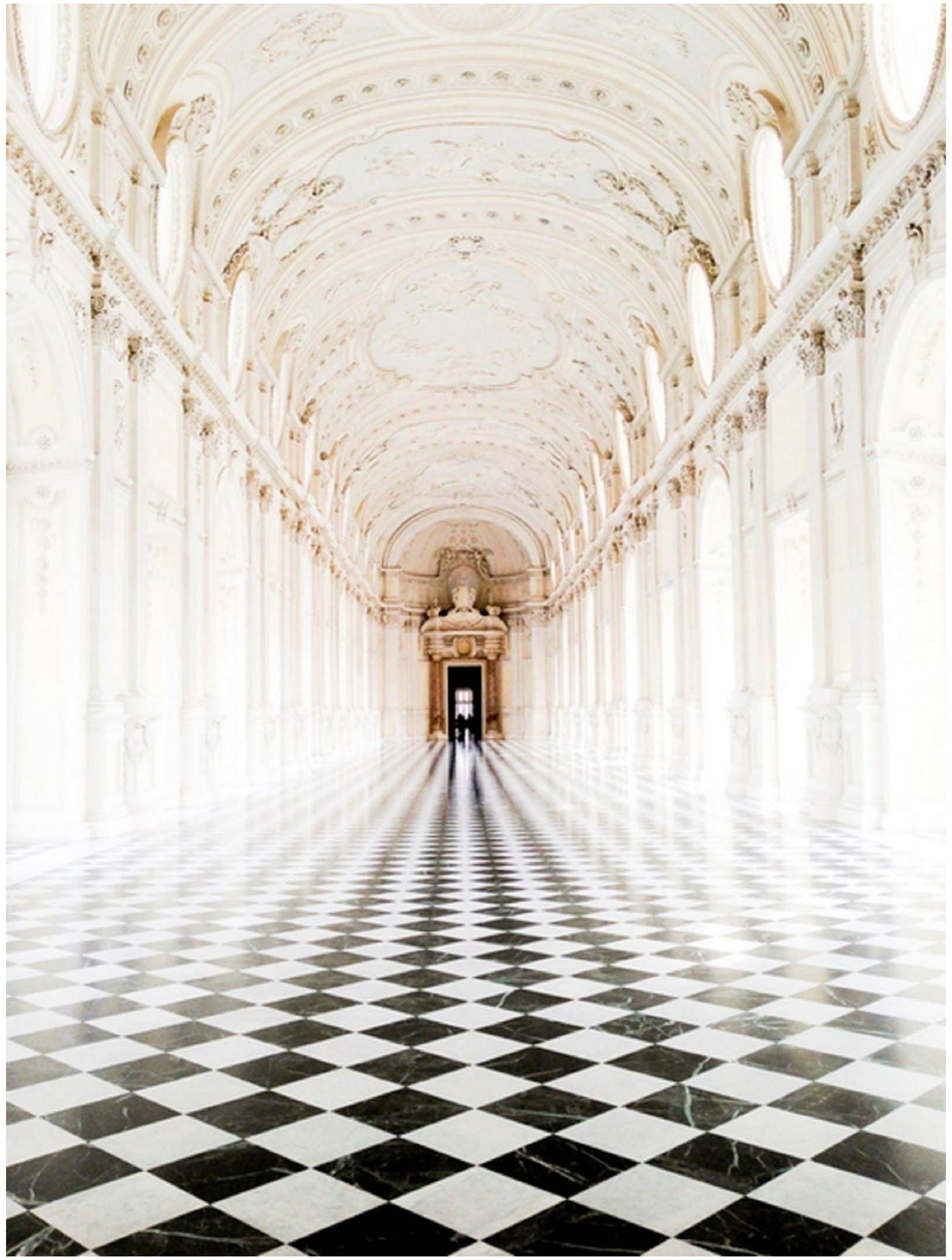
Use texture value to perturb surface normal to “fake” appearance of a bumpy surface
(note smooth silhouette/shadow reveals that surface geometry is not actually bumpy!)

displacement mapping

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

Perspective and texture

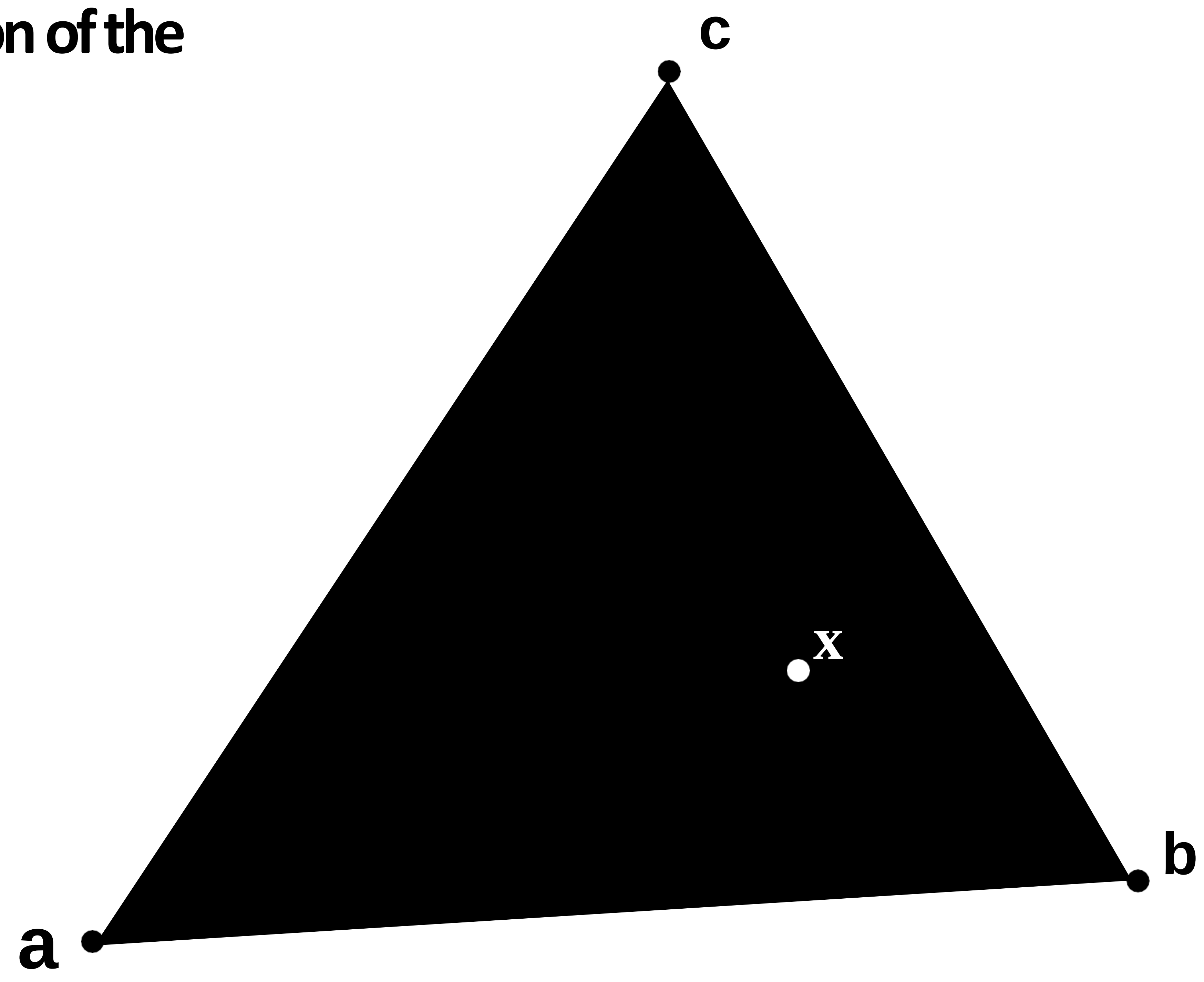
- PREVIOUSLY:
  - *transformation* (how to manipulate primitives in space)
  - *rasterization* (how to turn primitives into colored pixels)
- TODAY:
  - see where these two ideas come crashing together!
  - talk about how to map *texture* onto a primitive to get more detail
  - ...and how perspective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



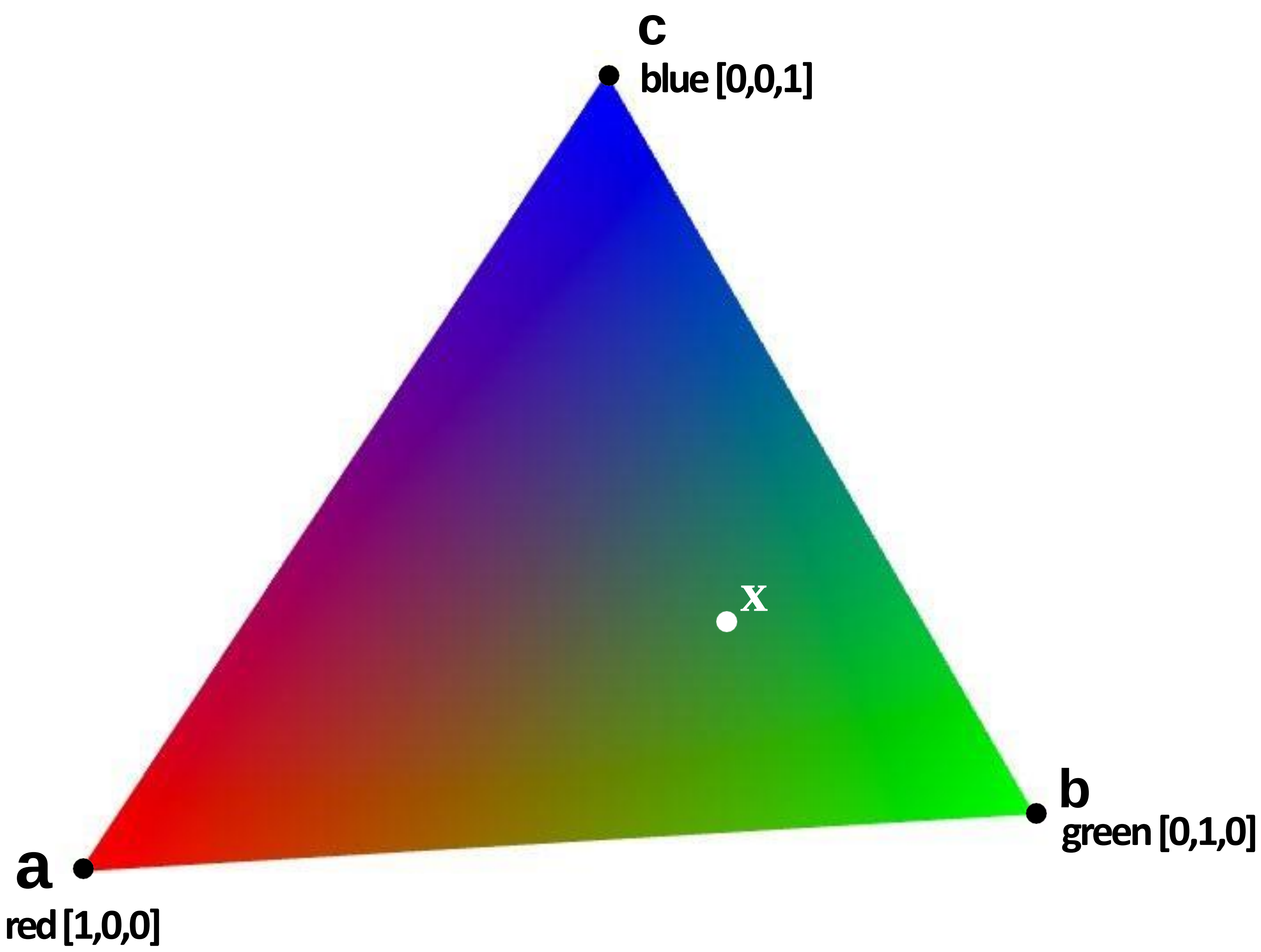
Why is it hard to render an image like this?

Recall the function coverage(x,y) from previous lecture

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



Consider sampling a different signal: color(x,y)



What is the triangle's color at the point x , given its colors at points a, b, c ?

Review: interpolation in 1D

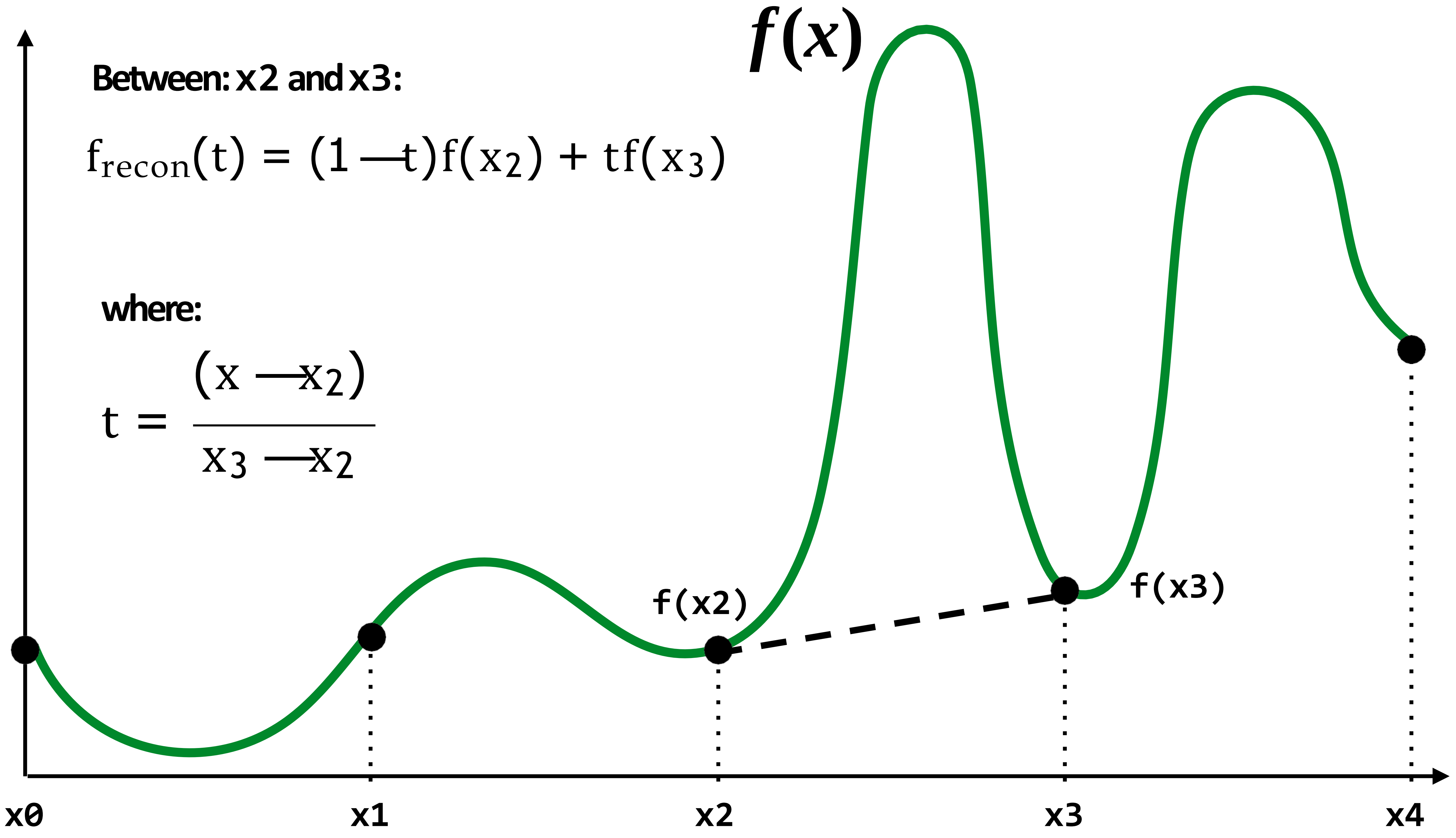
$f_{recon}(x)$ = linear interpolation between values of two closest samples to x

Between: x2 and x3:

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

where:

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

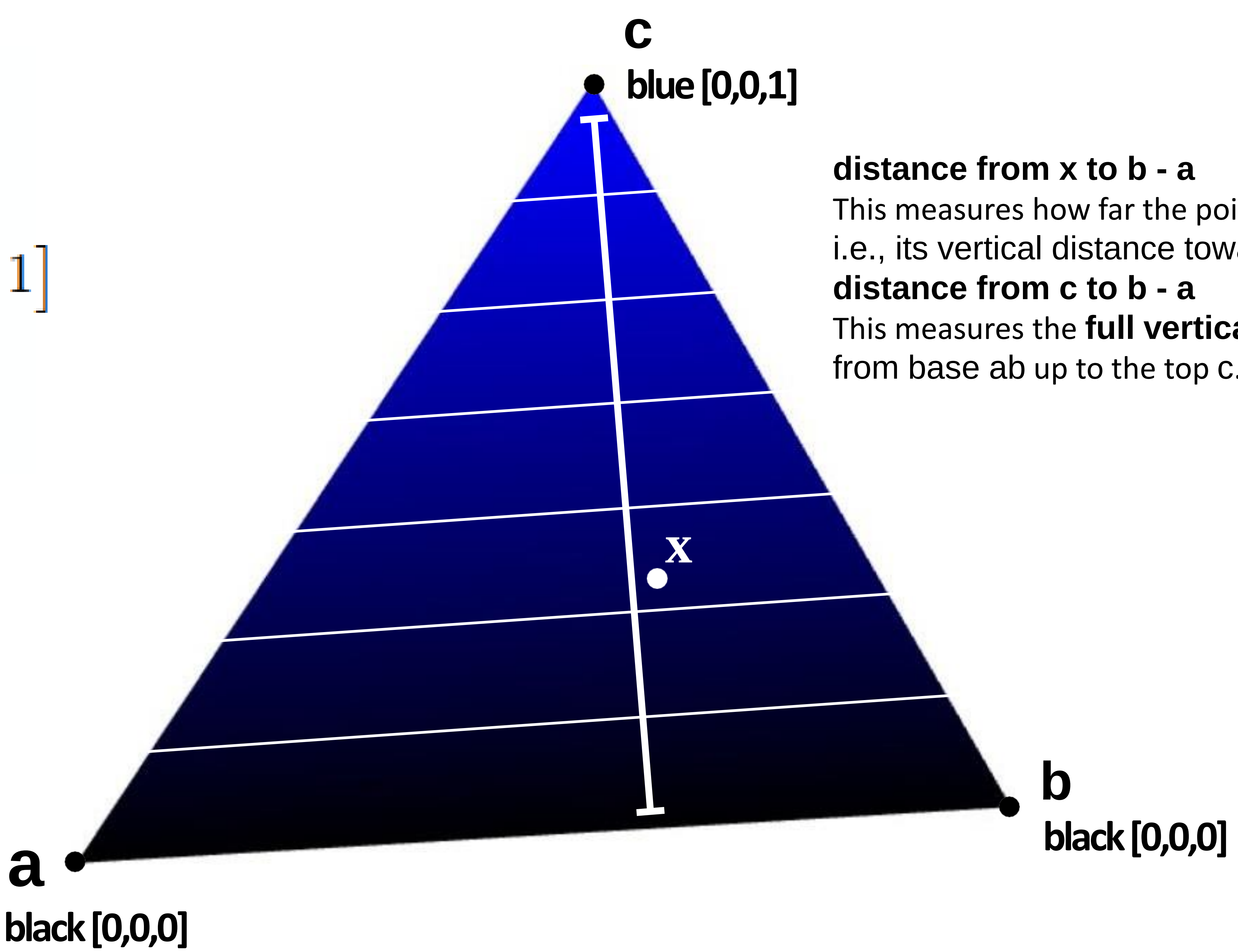


Consider similar behavior on triangle

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

Color = $(1-t)\begin{bmatrix}0 & 0 & 0\end{bmatrix} + t\begin{bmatrix}0 & 0 & 1\end{bmatrix}$

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

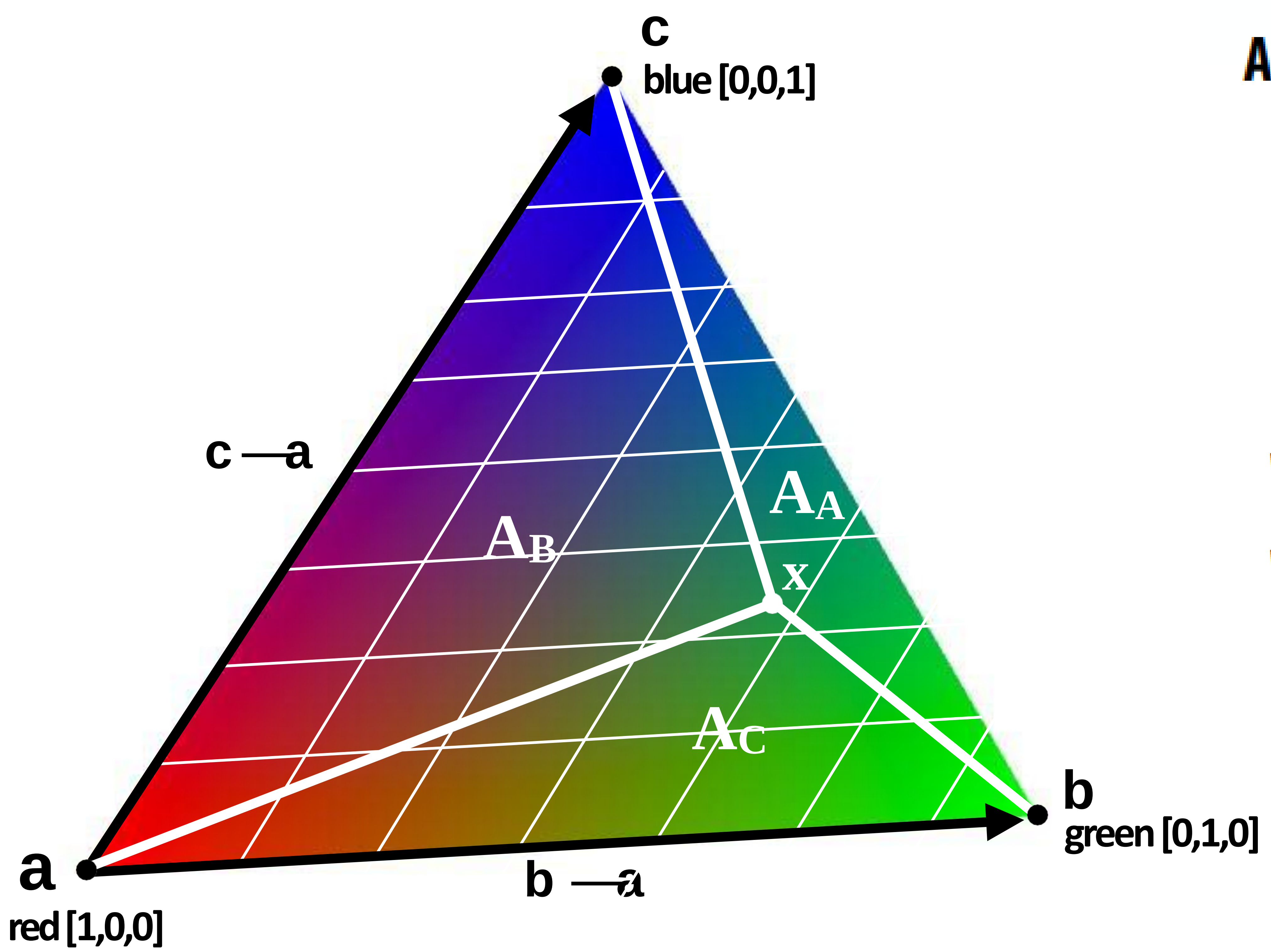


distance from x to b - a
This measures how far the point x is **above the base**, i.e., its vertical distance toward c.
distance from c to b - a
This measures the **full vertical height** of the triangle from base ab up to the top c.

How can we interpolate in 2D between three values?

Why b - a?
Because b - a is a **direction vector** representing the base of the triangle, so the formula is projecting and measuring distance **perpendicular** to this base.

Linear interpolation of quantities over triangle: 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?

Triangle vertices (2D coordinates):

- $a = (0, 0)$
- $b = (4, 0)$
- $c = (0, 4)$

Color at each vertex:

- $\text{Color}_a = \text{Red} = [1, 0, 0]$
- $\text{Color}_b = \text{Green} = [0, 1, 0]$
- $\text{Color}_c = \text{Blue} = [0, 0, 1]$

Pick a point **x** inside the triangle:

- $x = (1, 1)$

Compute Area of triangle

Use the **shoelace formula** or cross product for area:

$$A = \frac{1}{2} |(b_x - a_x)(c_y - a_y) - (c_x - a_x)(b_y - a_y)| = \frac{1}{2} |(4)(4) - (0)(0)| = \frac{1}{2} \cdot 16 = 8$$

Compute Area of sub triangle

A_α = area of triangle formed by points `x`, `b`, `c`

$$A_\alpha = \frac{1}{2} |(x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1)|$$

$$x = (x_1, y_1)$$

$$b = (x_2, y_2)$$

$$c = (x_3, y_3)$$

$$A = \frac{1}{2} |(4 - 1)(4 - 1) - (0 - 1)(0 - 1)| = \frac{1}{2} \left|3 \cdot 3 - (-1)^2\right| = \frac{1}{2}(9 - 1) = 4$$

Area of sub triangle

- A_α = area of triangle formed by points `x`, `b`, `c`
- A_β = area of triangle formed by `x`, `c`, `a`
- A_γ = area of triangle formed by `x`, `a`, `b`

To compute area of triangle for x, c, a

$x = (1, 1) \qquad x_1 = 1, y_1 = 1$

$c = (0, 4) \qquad x_2 = 0, y_2 = 4$

$a = (0, 0) \qquad x_3 = 0, y_3 = 0$

$$A(x, c, a) = \frac{1}{2}\left|(x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1)\right|$$

$$A_\beta = \frac{1}{2}\left|(0-1)(0-1) - (0-1)(4-1)\right|$$

$$= \frac{1}{2}\left|(-1)(-1) - (-1)(3)\right| = \frac{1}{2}\left|1 - (-3)\right| = \frac{1}{2} \cdot 4 = 2$$

To compute area of triangle for x, a, b

$A_\gamma = \text{area}(x, a, b)$

$$A_\gamma = \frac{1}{2}|(0-1)(0-1) - (4-1)(0-1)| = \frac{1}{2}(1 - (-3)) = \frac{1}{2} \cdot 4 = 2$$

Calculate the barycentric coordinates

$A = 8 \qquad \alpha = 4/8 = 0.5$

$A_\alpha = 4 \qquad \beta = 2/8 = 0.25$

$A_\beta = 2 \qquad \gamma = 2/8 = 0.25$

$A_\gamma = 2$

Interpolate the color

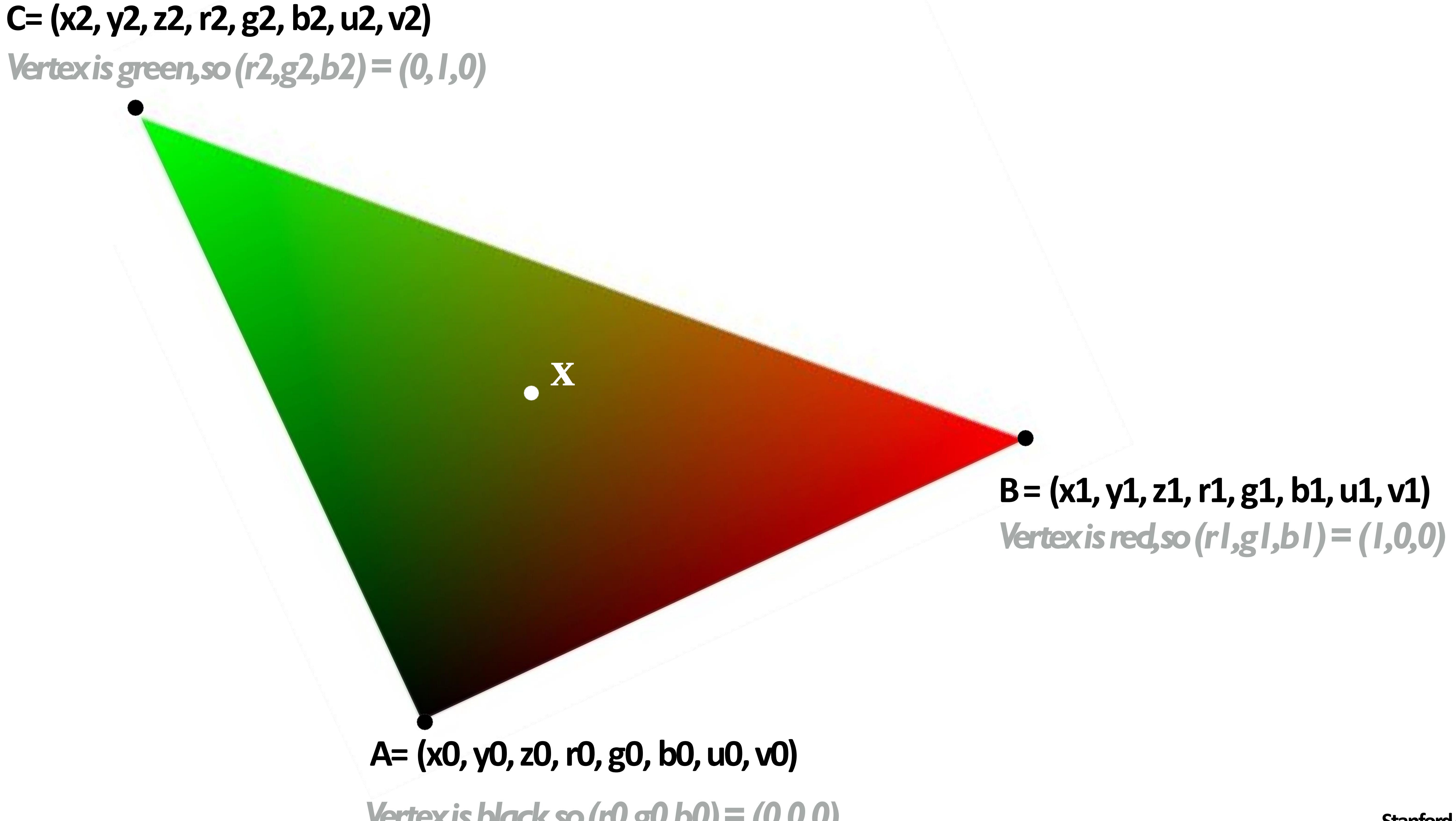
$$\text{Color}_x = \alpha \cdot \text{Color}_a + \beta \cdot \text{Color}_b + \gamma \cdot \text{Color}_c$$

$$= 0.5 \cdot [1, 0, 0] + 0.25 \cdot [0, 1, 0] + 0.25 \cdot [0, 0, 1]$$

$$= [0.5, 0.25, 0.25]$$

You can can linearly interpolate any values (defined at vertices) over the triangle this way

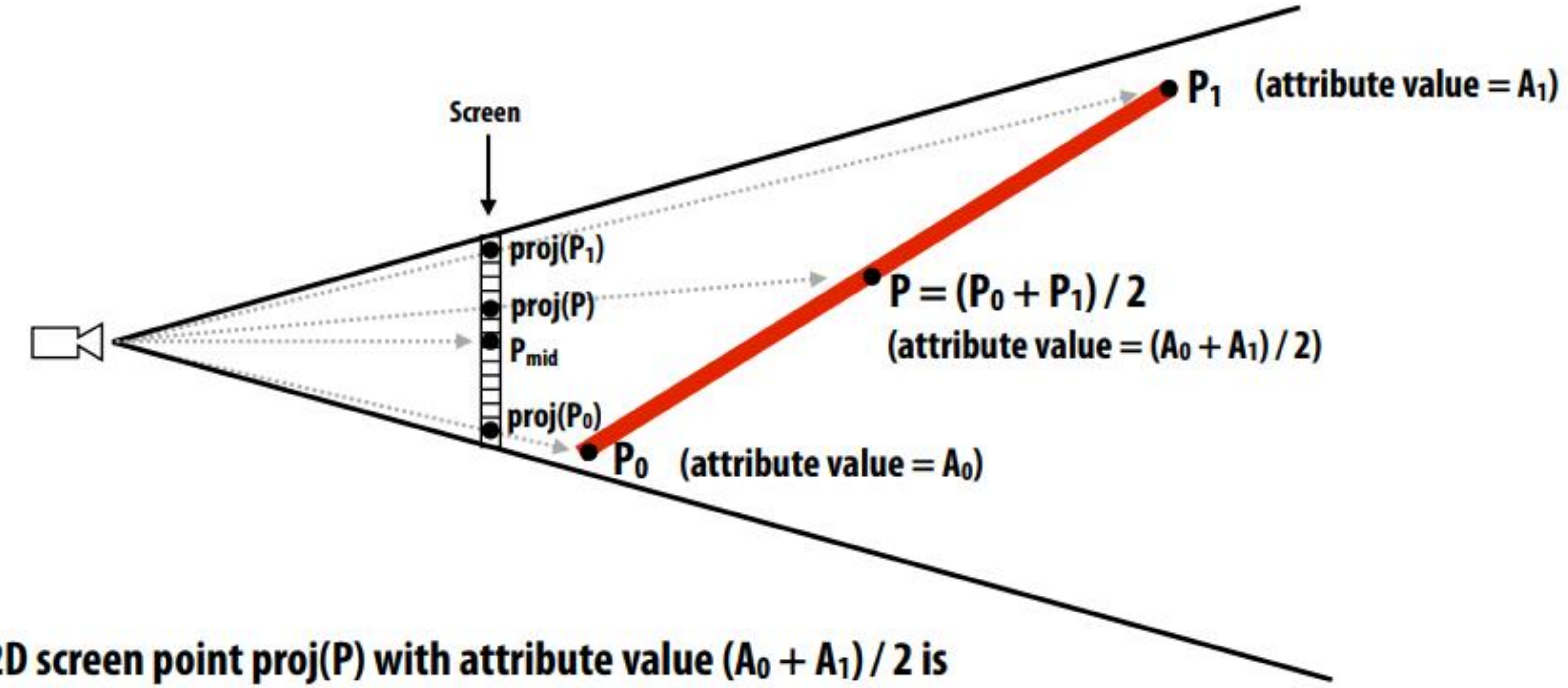
Here, I'm interpolating position (x,y,z), color (r,g,b), and extra values (u,v)



Not so fast... perspective incorrect interpolation

The value of an attribute at the 3D point P on a triangle is a linear combination of attribute values at vertices.

But due to perspective projection, barycentric interpolation of values on a triangle with vertices of different depths in 3D <u>is not</u> linear in 2D screen XY coordinates (vertex coordinates *after* projection)

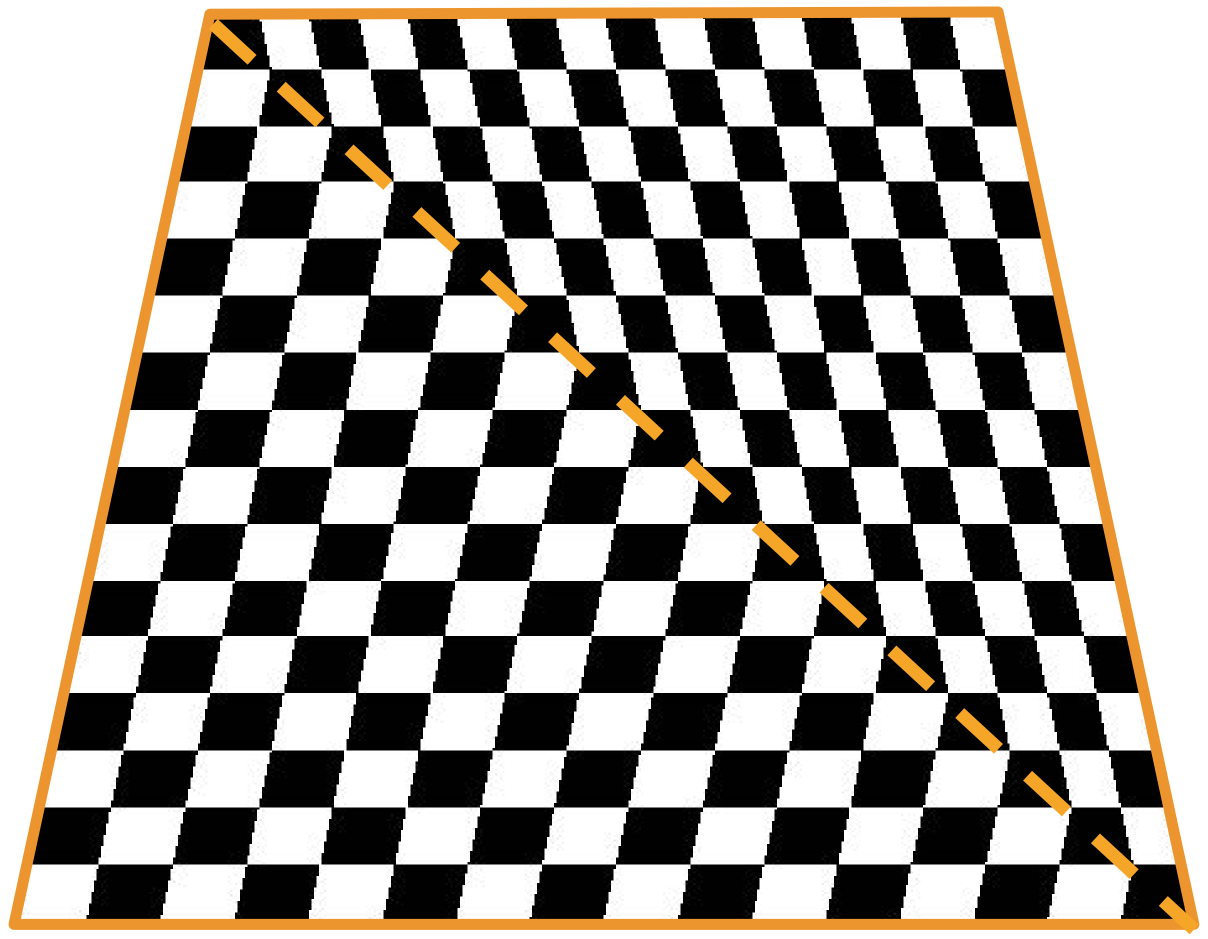


In this example, the 2D screen point proj(P) with attribute value $(A_0 + A_1) / 2$ is not halfway between the 2D screen points proj(P_0) and proj(P_1).

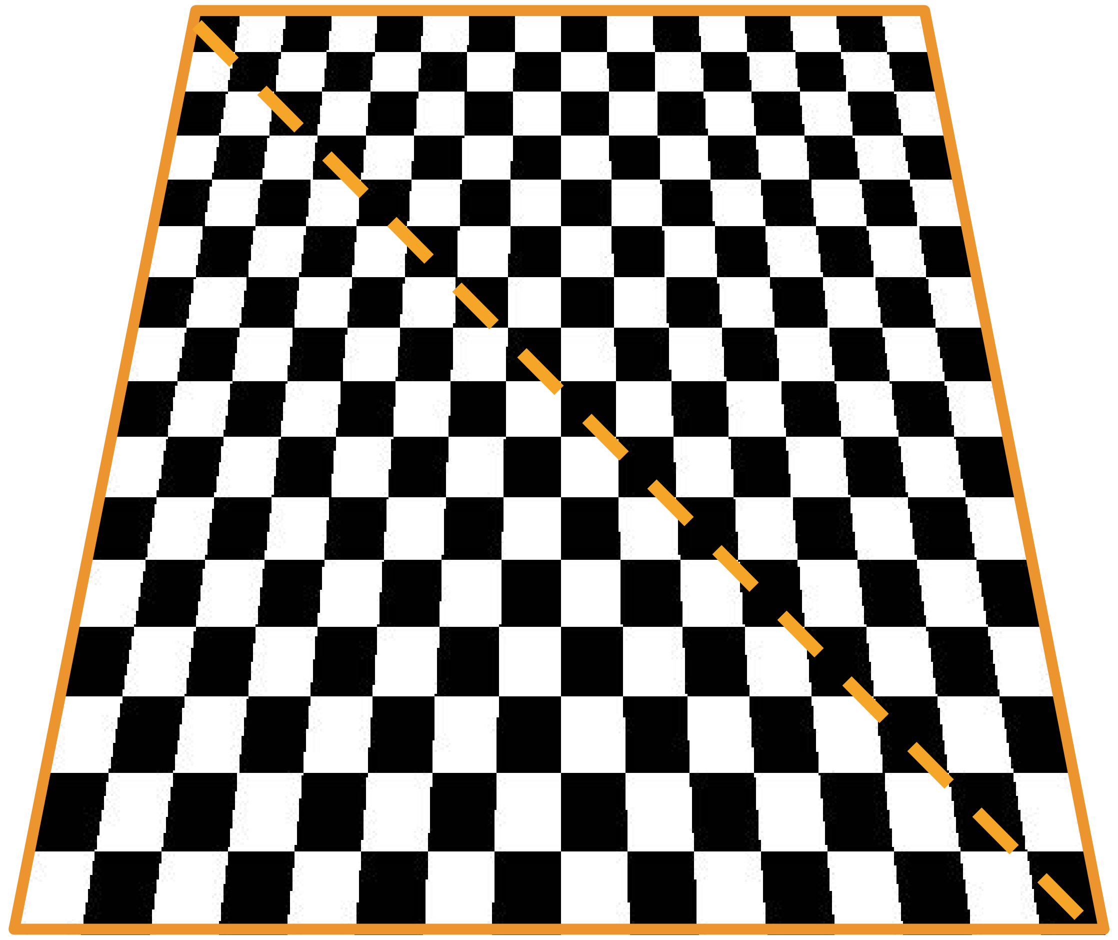
Similarly, the attribute's value at P_{mid} = (proj(P_0) + proj(P_1)) /2 is not $(A_0 + A_1)$ /2.

Perspective correct interpolation

This is a plane (two triangles), tilted down and rendered under perspective.



2D screen-space interpolation



3D world-space interpolation

Perspective correct interpolation on a projected triangle (in 2D)

- **Problem:**
  - Given some value f_i at each of a 3D triangle's vertices, that is linearly interpolated across the triangle in 3D
  - And the 2D screen coordinates $P_i=(x_i,y_i)$ of each of a triangles vertices after projection
  - As well as the homogenous coordinate w_i for each vertex

Sample the value of f(x,y) for the projected triangle at a given 2D screen space location (x,y)

Perspective-correct interpolation

Assume a triangle attribute varies linearly across the triangle (in 3D)

Attribute's value at 3D point on triangle $P = \begin{bmatrix} x & y & z \end{bmatrix}^T$ **is given by:**

$$f(x, y, z) = ax + by + cz$$

Perspective project P, get 2D homogeneous representation:

$$\begin{bmatrix} x_{2D-H} \\ y_{2D-H} \\ w \end{bmatrix} \leftarrow \begin{bmatrix} x \\ y \\ z \\ z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

projection of *P* in 2D-H — **Drop z to move to 2D-H** — **perspective projection of *P* in 3D-H** — **Simple perspective projection matrix *** — **point *P* in 3D-H**

*** Note: using a more general perspective projection matrix only changes the coefficient in front of x_{2d} and y_{2d} . (property that f/w is affine still holds)**

Then plug back in to equation for *f* at top of slide…

$$f(x_{2D-H}, y_{2D-H}) = ax_{2D-H} + by_{2D-H} + cw$$

$$\frac{f(x_{2D-H}, y_{2D-H})}{w} = \frac{a}{w}x_{2D-H} + \frac{b}{w}y_{2D-H} + c$$

$$\frac{f(x_{2D}, y_{2D})}{w} = \frac{a}{w}x_{2D} + \frac{b}{w}y_{2D} + c$$

So … $\frac{f}{w}$ **is affine function of 2D screen coordinates:** $\begin{bmatrix} x_{2D} & y_{2D} \end{bmatrix}^T$

Direct evaluation of surface attributes from 2D-H vertices

For any surface attribute (with value defined at triangle vertices as: f_a, f_b, f_c)

w coordinate of vertex *a* after perspective projection transform

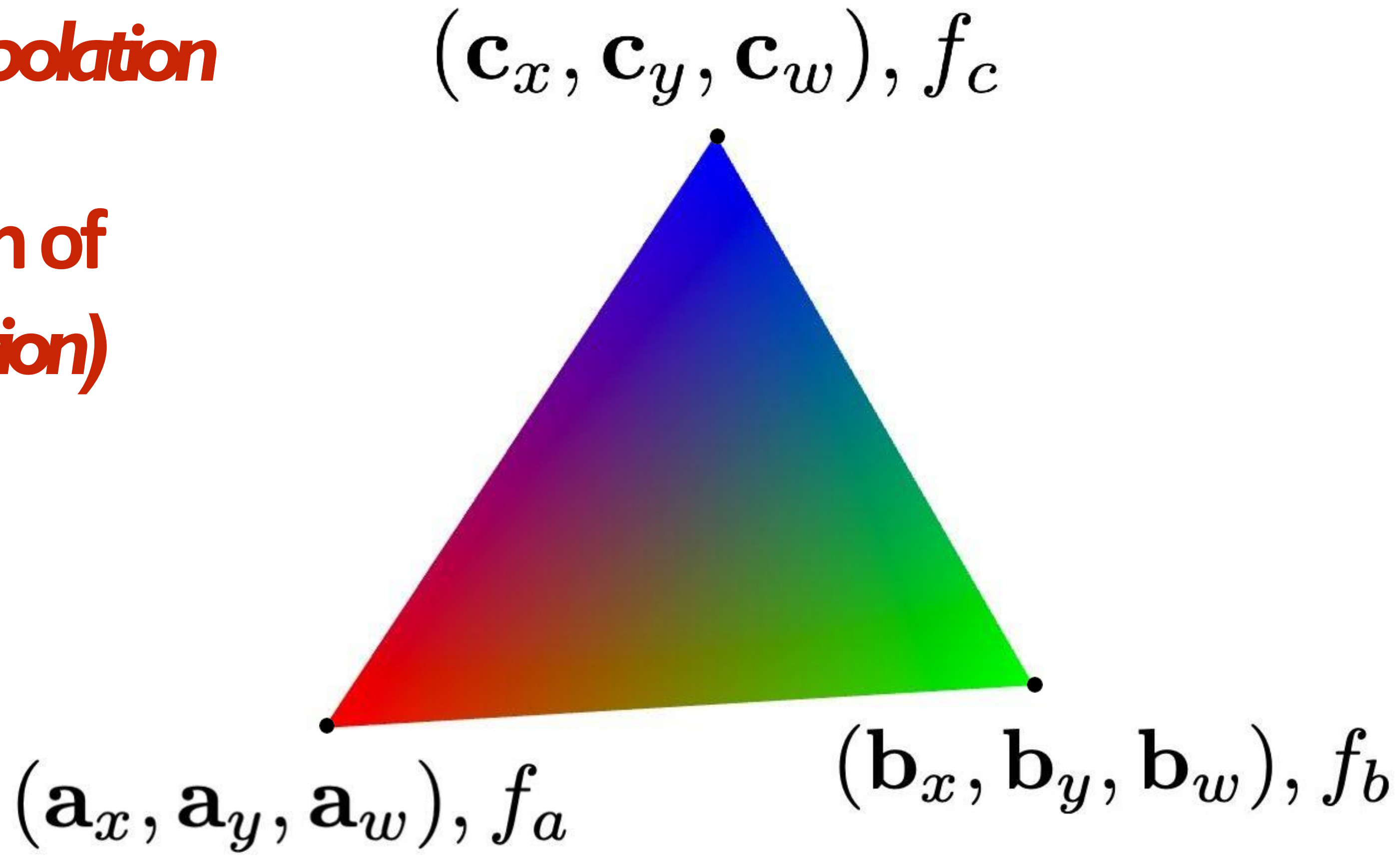
$$\frac{f_a}{\mathbf{a}_w} = A\mathbf{a}_x + B\mathbf{a}_y + C$$

$$\frac{f_b}{\mathbf{b}_w} = A\mathbf{b}_x + B\mathbf{b}_y + C$$

$$\frac{f_c}{\mathbf{c}_w} = A\mathbf{c}_x + B\mathbf{c}_y + C$$

Original value of attribute at vertex *a* ***before interpolation***

projected 2D position of vertex *a* *(after projection)*



3 equations, solve for 3 unknowns (A, B, C)

This is done as a per triangle "setup" computation prior to sampling, just like you computed edge equations for evaluating coverage.

Efficient perspective-correct interpolation

GOAL:
We want to compute the value of an attribute f(x,y)f(x, y)f(x,y) (e.g., color, depth, texture coordinate) at any point (x, y) inside a triangle, while correctly accounting for perspective distortion (i.e., to make sure textures and colors look right in 3D space projected to 2D).

Setup:

Given f_a, f_b, f_c and w_a, w_b, w_c... compute A, B, C for $f/w(x,y) = Ax + By + C$

Also compute equation for $1/w(x,y)$

To evaluate surface attribute f(x,y) at every covered sample (x,y):

Evaluate $^1/_w\,(x,y)$ (from precomputed equation for value $^1/_w$)

Reciprocate $^1/_w\,(x,y)$ to get $w(x,y)$

For each triangle attribute:

Evaluate $^f/_w\,(x,y)$ (from precomputed equation for value $^f/_w$)

Multiply $^f/_w\,(x,y)$ by $w(x,y)$ to get $f(x,y)$

Works for any surface attribute f that varies linearly across triangle: e.g., color, depth, texture coordinates

Three triangle vertices: a, b, c

Each has:

- A weight w_a, w_b, w_c (this is typically the **perspective depth** or **homogeneous w**)
- An attribute value f_a, f_b, f_c (e.g., red component, depth, u texture coordinate, etc.)

We precompute:

- $\frac{f}{w}$ at each vertex:
  - $\frac{f_a}{w_a}, \frac{f_b}{w_b}, \frac{f_c}{w_c}$
- $\frac{1}{w}$ at each vertex:
  - $\frac{1}{w_a}, \frac{1}{w_b}, \frac{1}{w_c}$

$\frac{f}{w}$ varies linearly over the triangle.

$\frac{1}{w}$ also varies linearly.

$$\frac{f}{w}(x,y) = A_f x + B_f y + C_f$$

$$\frac{1}{w}(x,y) = A_w x + B_w y + C_w$$

Using barycentric coordinates

For each pixel (x,y) inside the triangle

a. Evaluate:

$$\frac{1}{w}(x,y) = A_w x + B_w y + C_w$$

b. Invert:

$$w(x,y) = \frac{1}{\frac{1}{w}(x,y)} = \frac{1}{A_w x + B_w y + C_w}$$

c. Evaluate:

$$\frac{f}{w}(x,y) = A_f x + B_f y + C_f$$

d. Final attribute:

$$f(x,y) = \left(\frac{f}{w}(x,y)\right) \cdot w(x,y)$$

To get perspective-correct attribute $f(x,y)$:

$$f(x,y) = (A_f x + B_f y + C_f) \cdot \left(\frac{1}{A_w x + B_w y + C_w}\right)$$

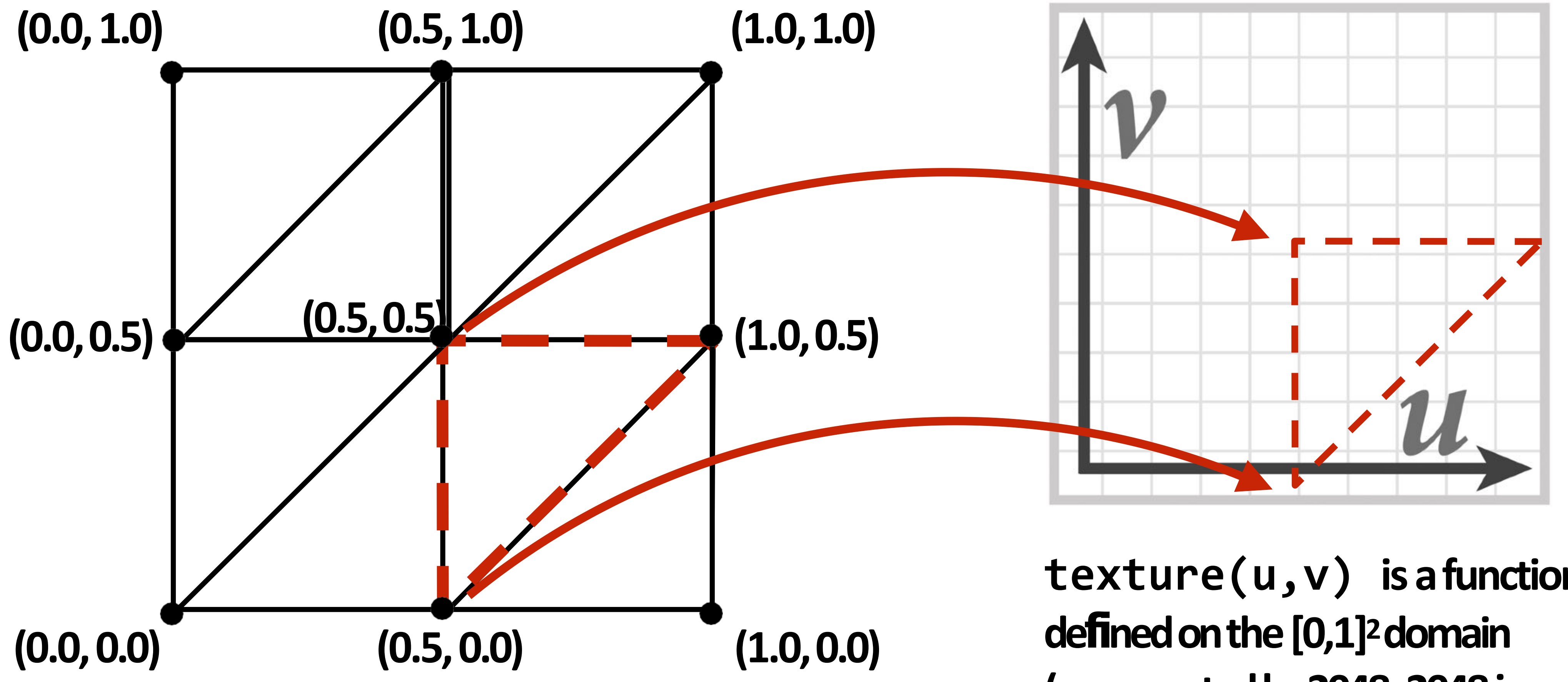
Texture coordinates

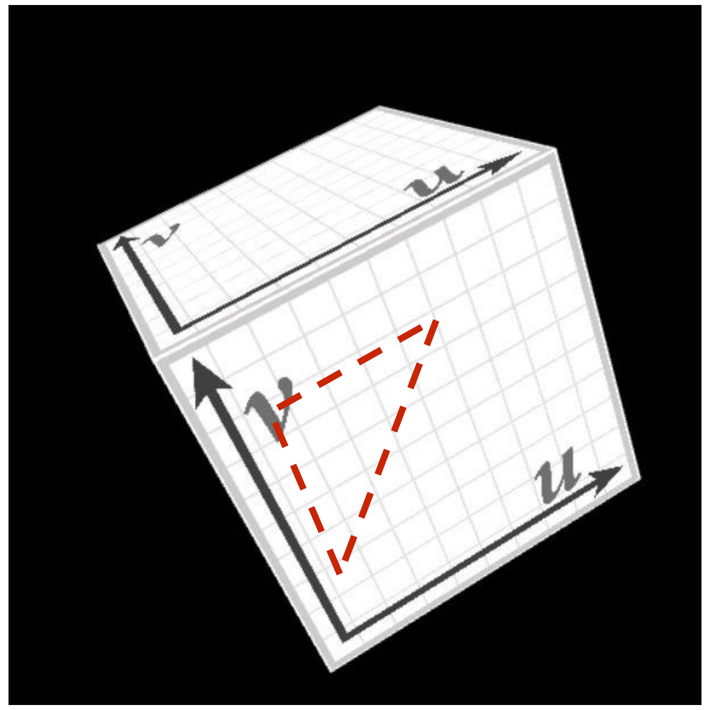
"Texture coordinates" define a mapping from surface coordinates (e.g., points on triangle) to points in the domain of a texture image

Surface (one face of cube)

Texture function (represented by an image)

Rendered image of texture mapped onto surface





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

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

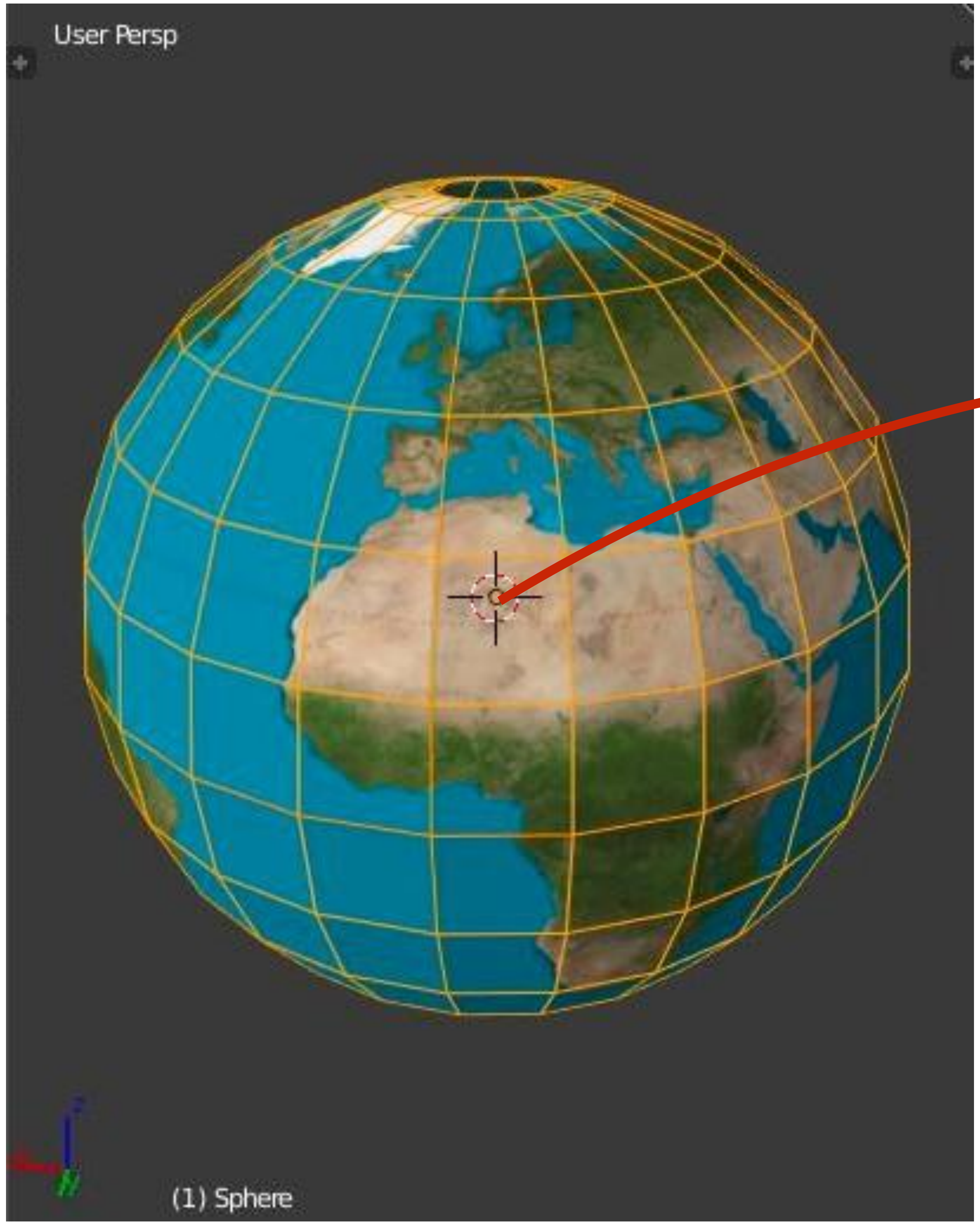
Location of highlighted triangle in texture space shown in red.

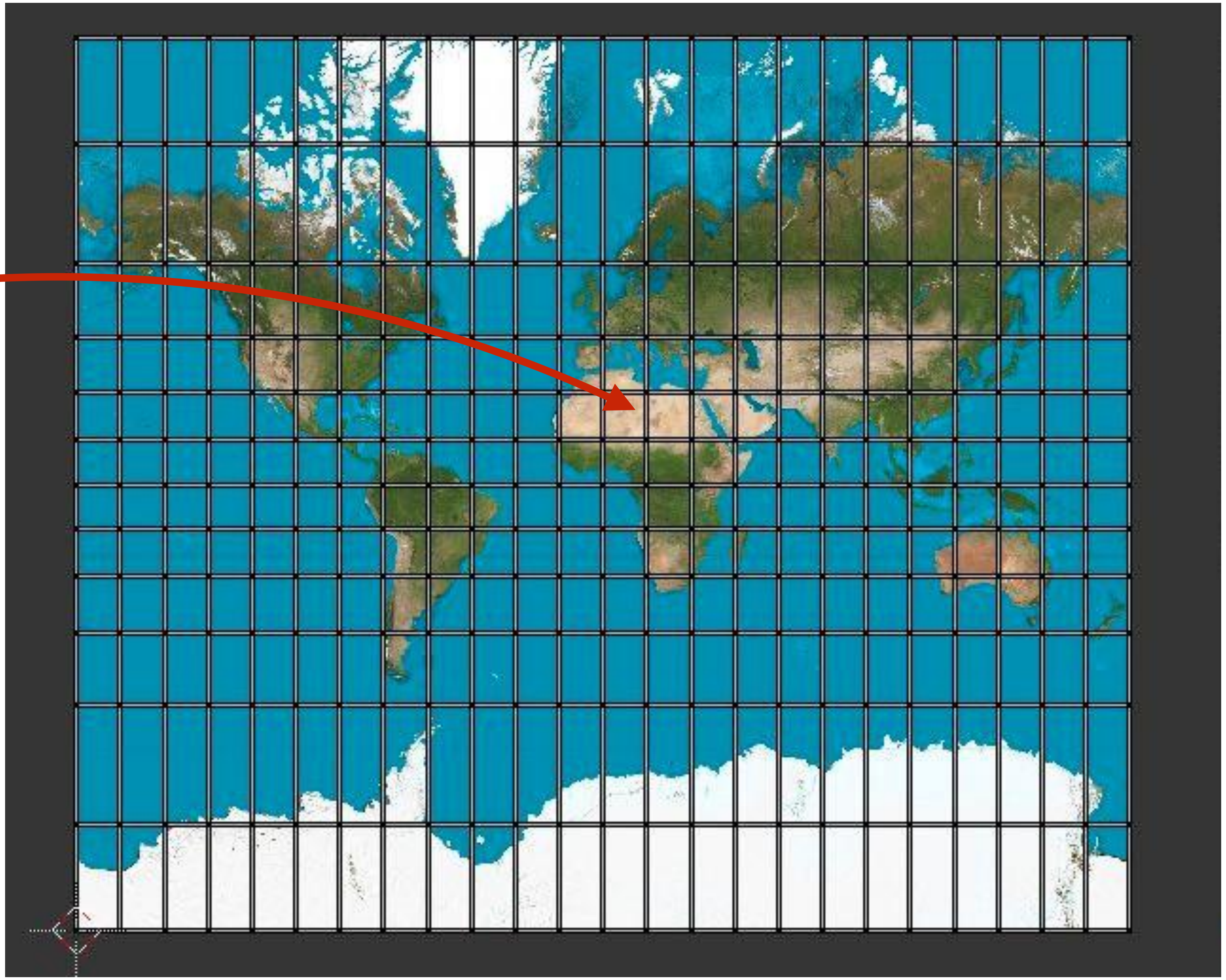
Final rendered result (entire cube shown).

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)

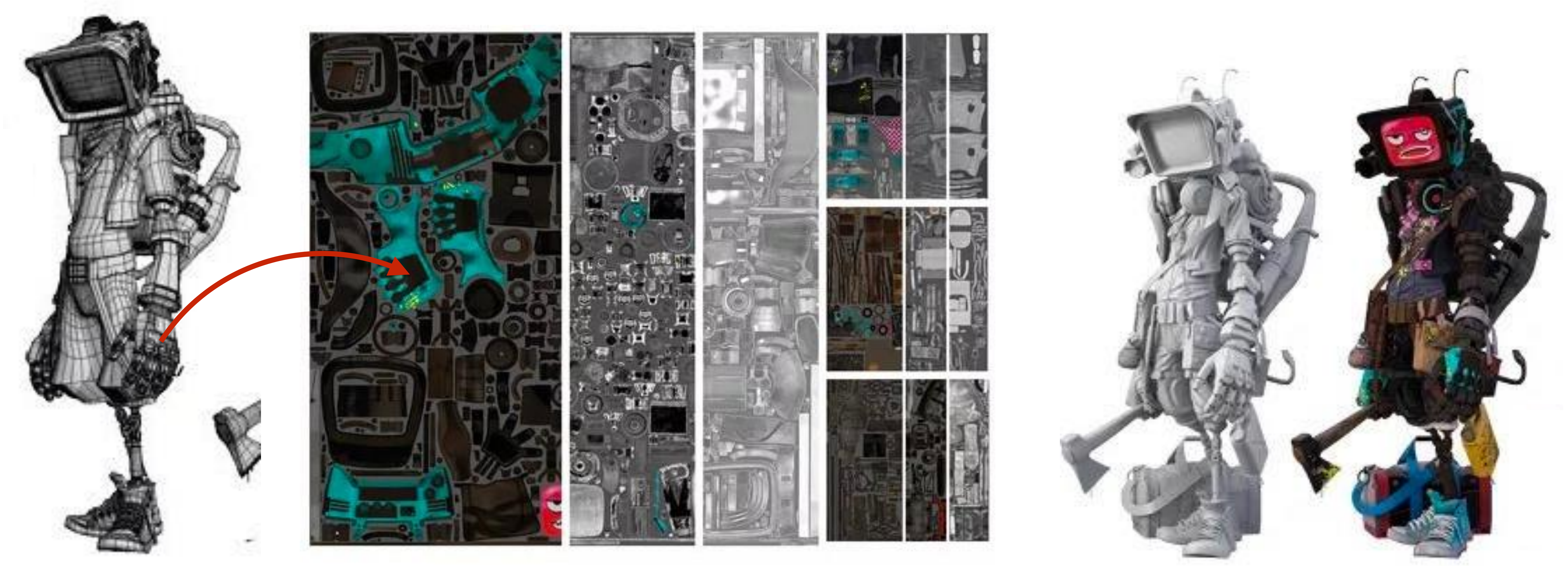
Many different mappings of surface position to texture space





Example: mercator projection onto sphere

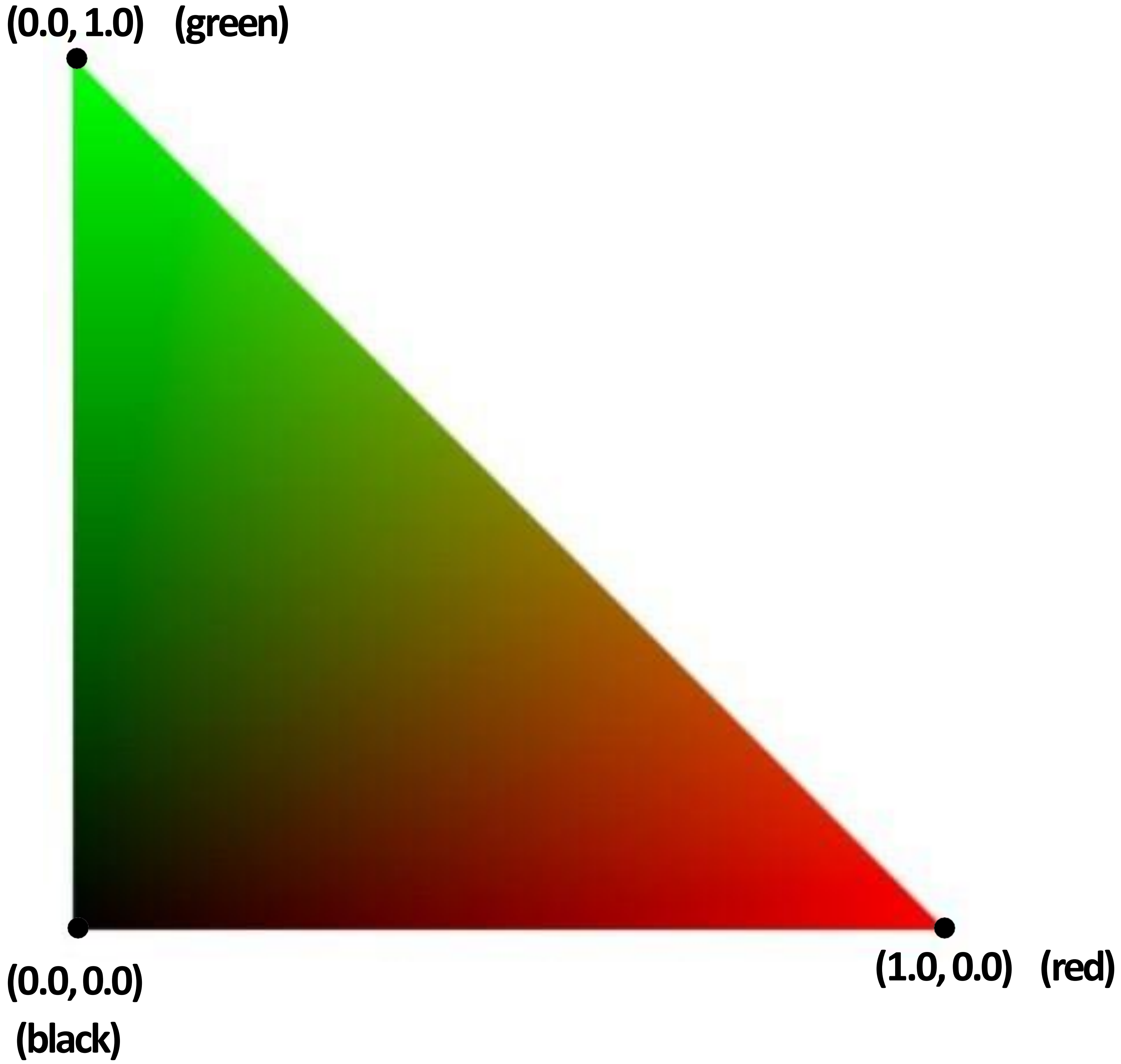
Texture “atlas”



https://www.creativebloq.com/3d/how-create-killer-3d-robot-21410645

Visualization of texture coordinates

Texture coordinates linearly interpolated over triangle

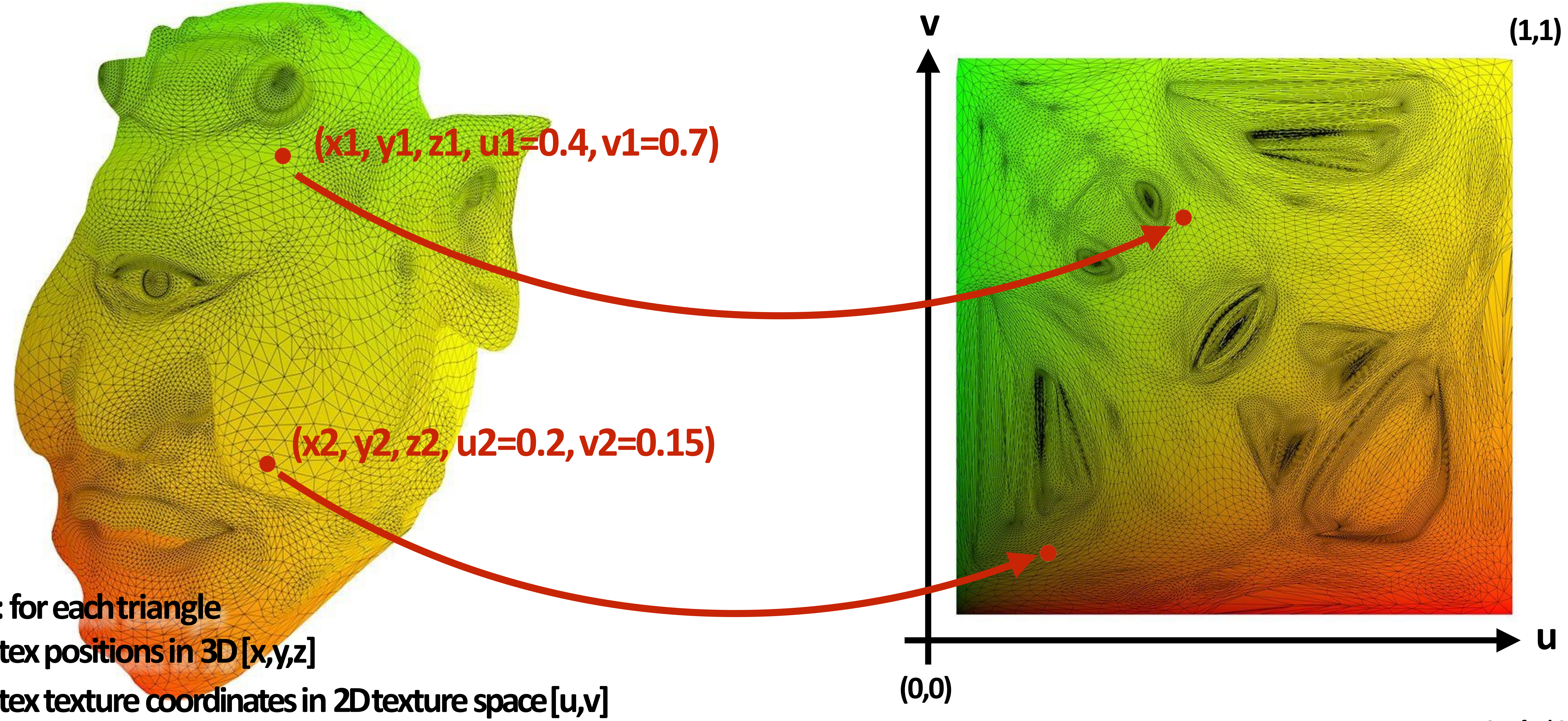


Texture coordinate values provided at triangle vertices

(Just like 3D positions are provided at vertices)

Visualization of texture coordinate value on mesh
(texture coordinate = color)

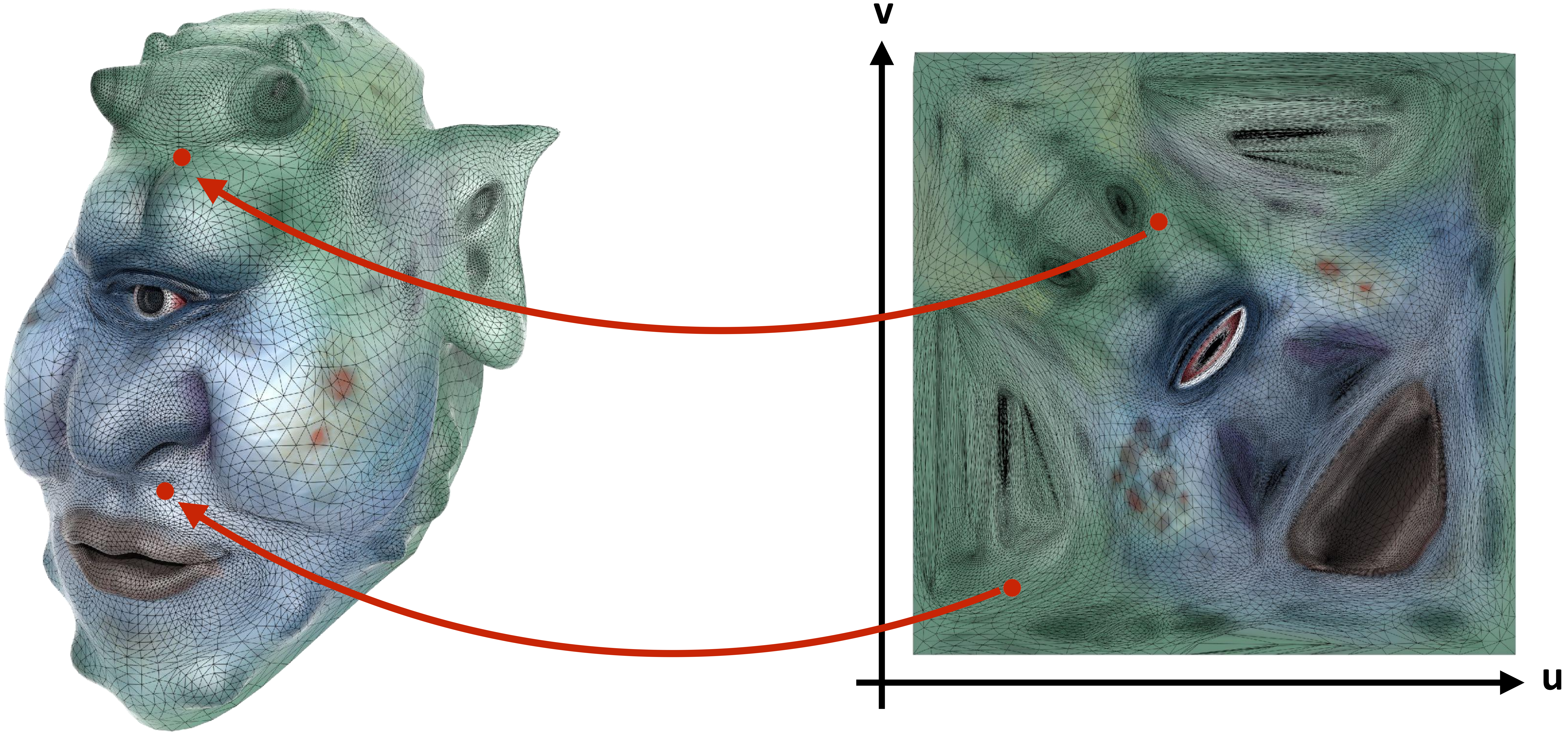
Visualization of location of triangle vertices
in texture space



Mesh inputs: for each triangle
- Per-vertex positions in 3D [x,y,z]
- Per-vertex texture coordinates in 2D texture space [u,v]

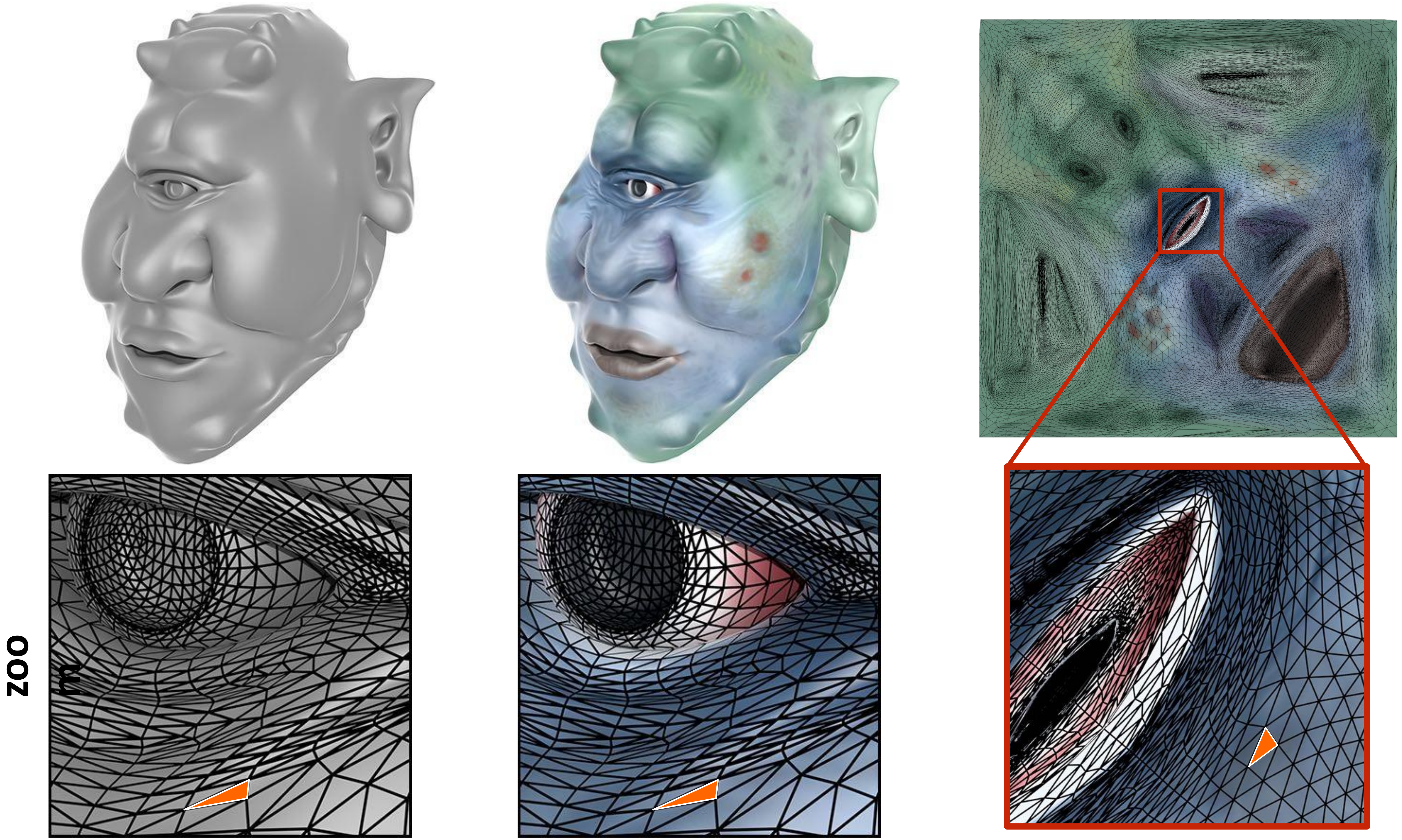
Texture mapping adds detail

Sample texture map at specified location in *texture coordinate space* to determine the surface's color at the corresponding point on surface.



Texture mapping adds detail

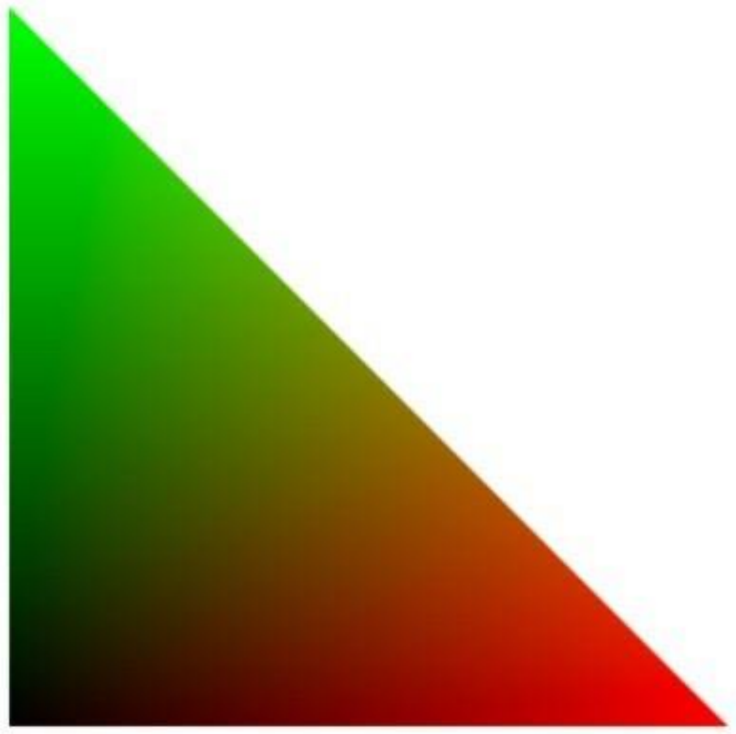
rendering without texture | rendering with texture | texture image

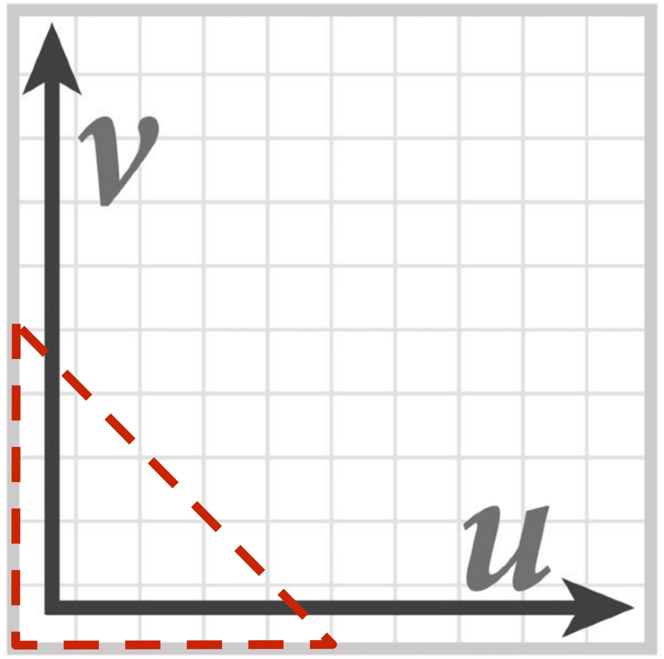


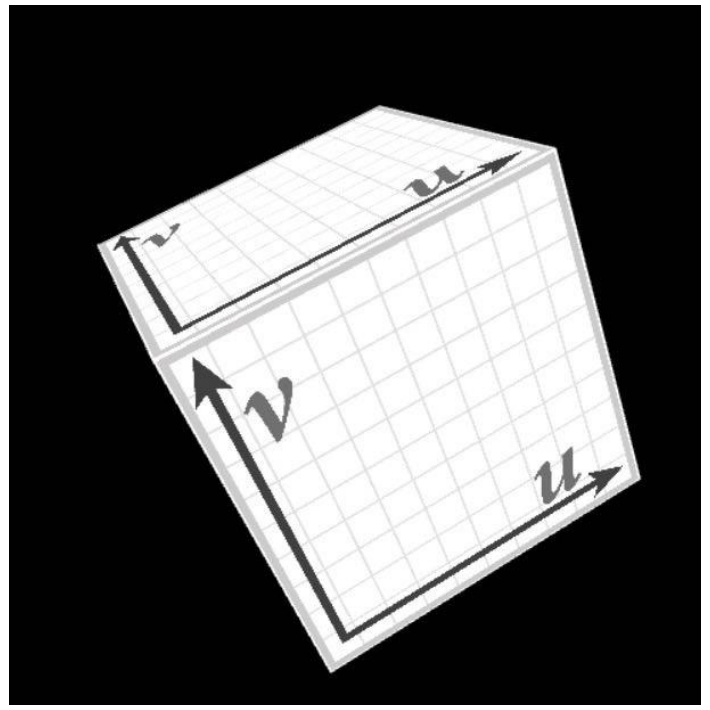
Each triangle "copies" a piece of the image back to the surface.

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 location given by (U,V)
    - Set color of surface point to sampled texture value

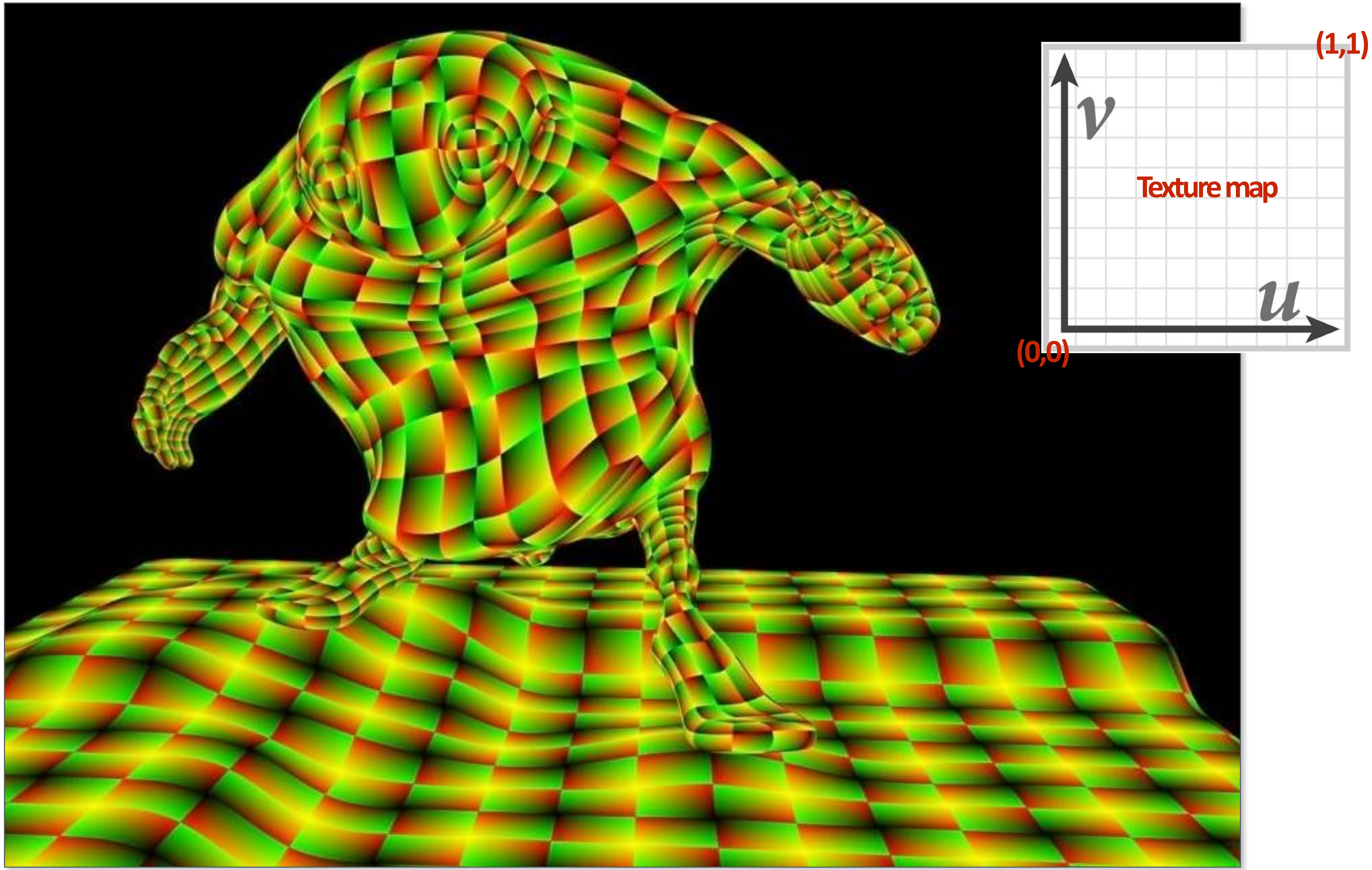






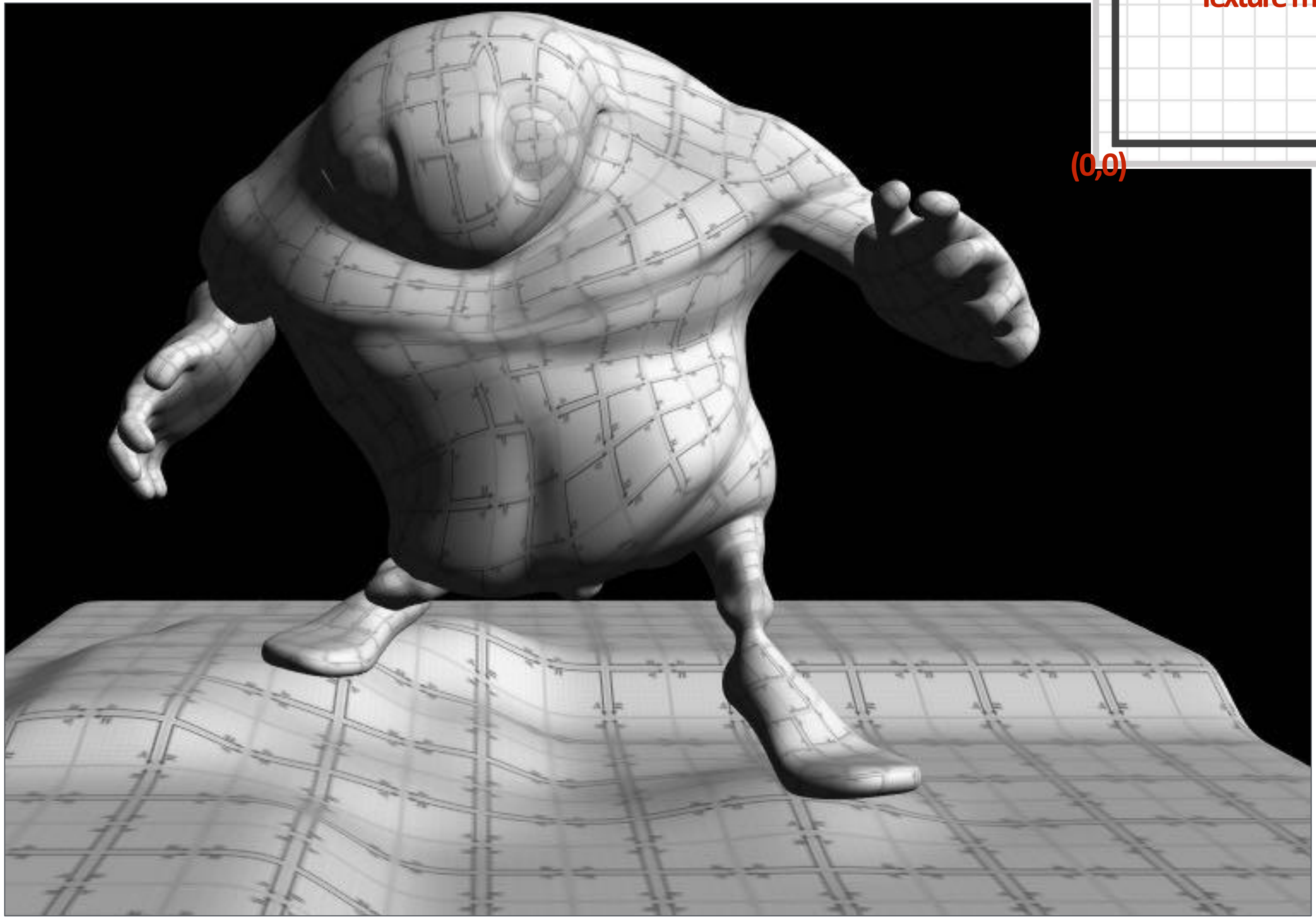
Texture coordinate visualization

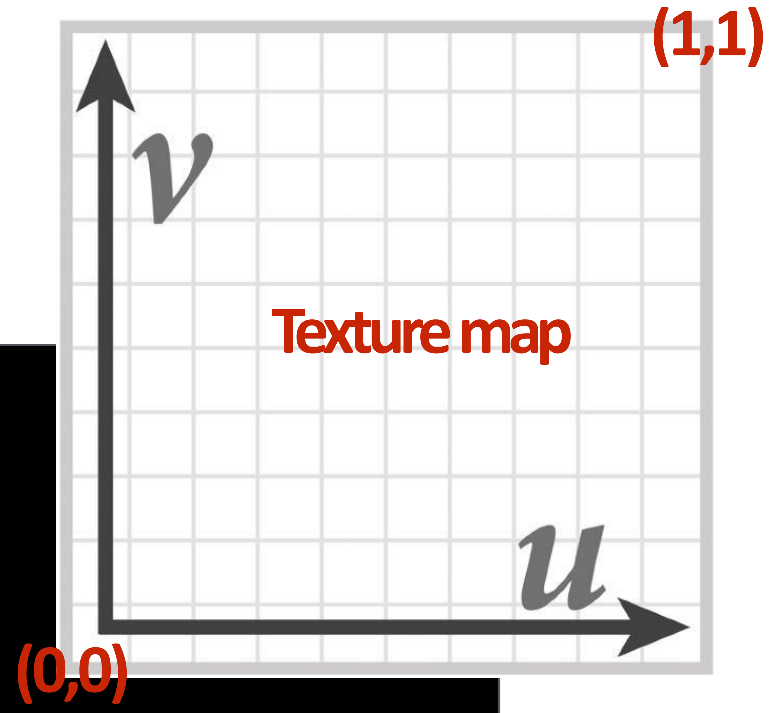
Defines mapping from point on surface to point (uv) in texture domain



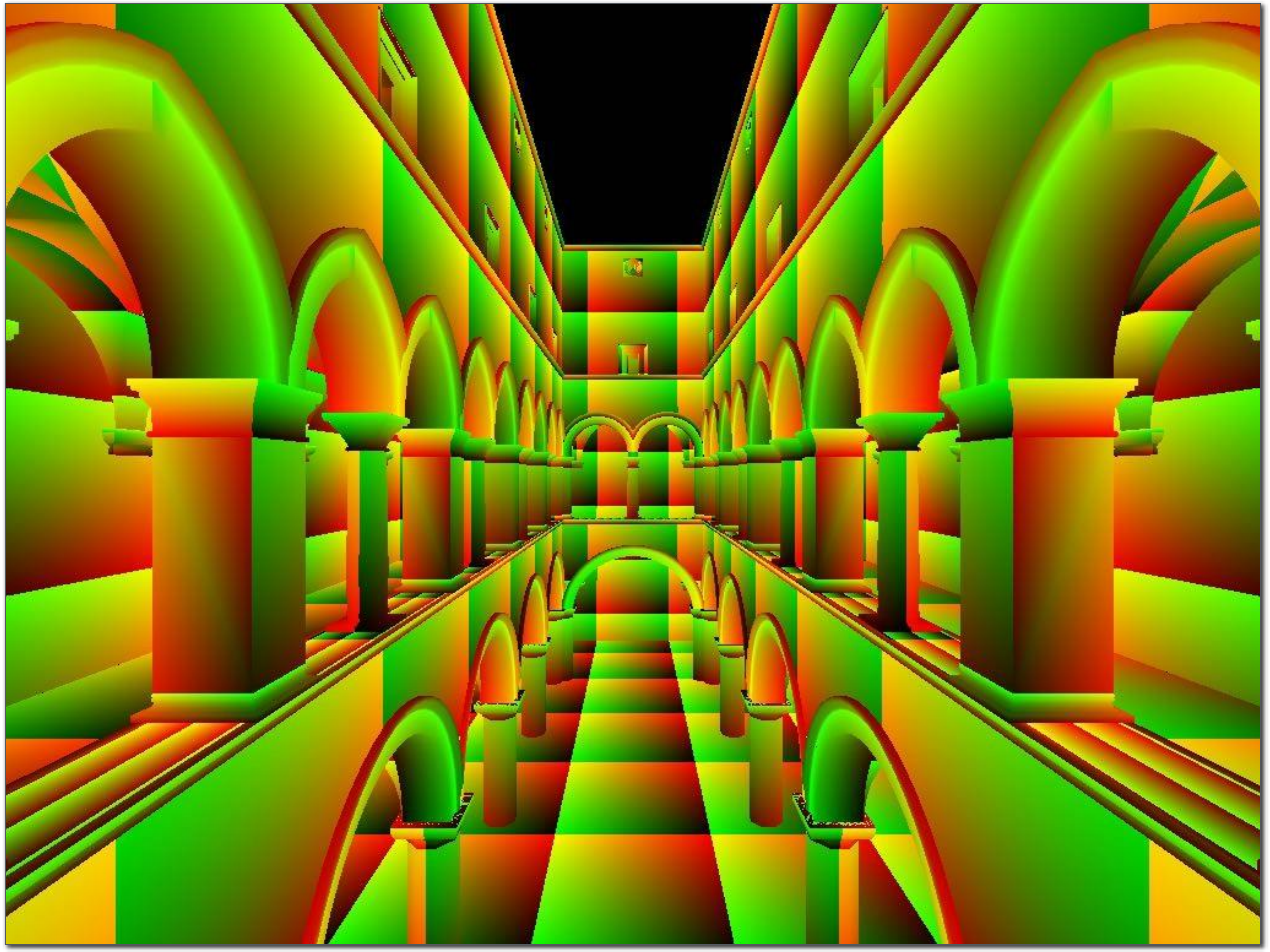
Red channel = u, Green channel = v
So uv=(0,0) is black, uv=(1,1) is yellow

Rendered result



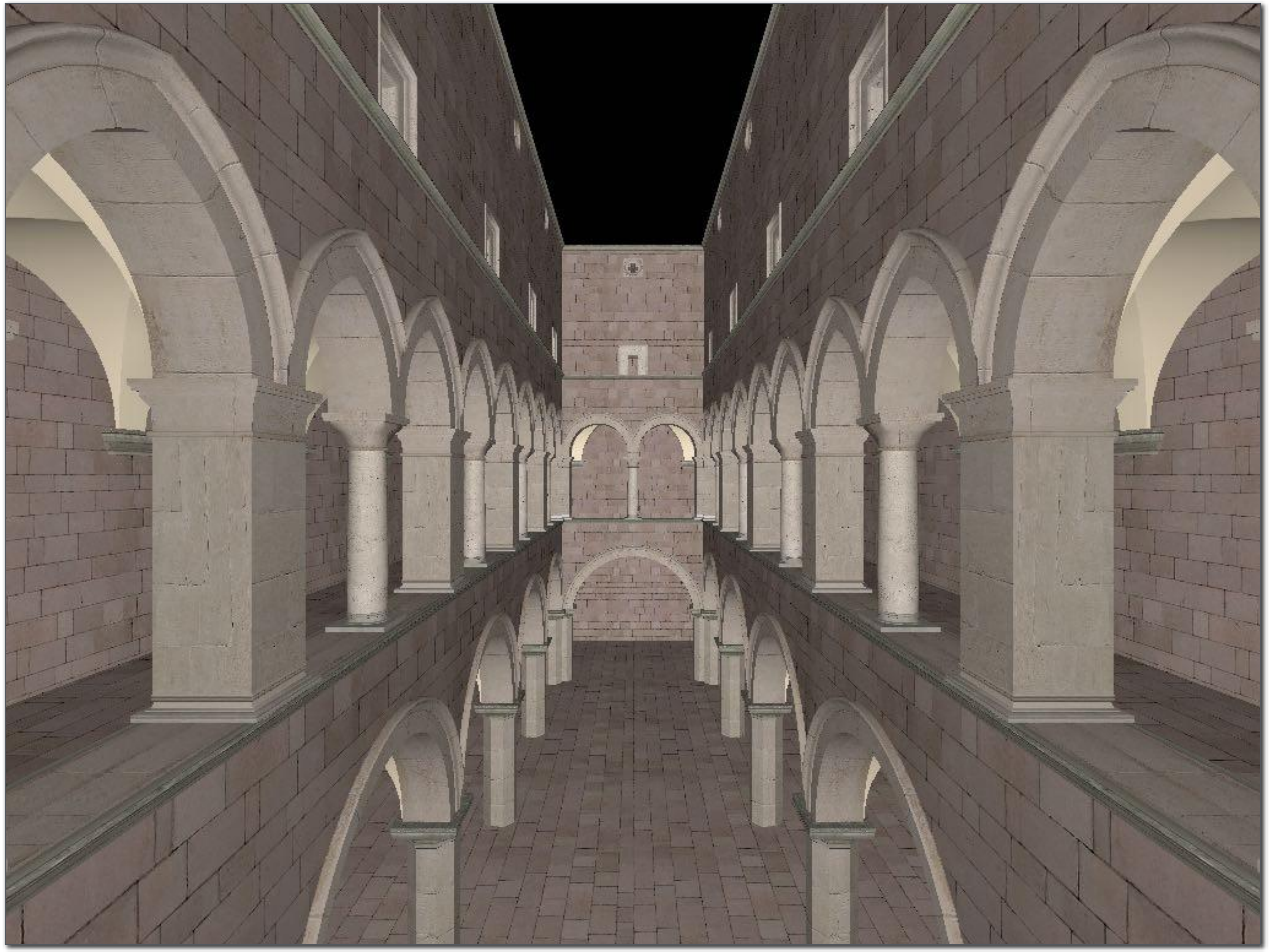


Visualization of texture coordinates



Notice texture coordinates repeat over surface.

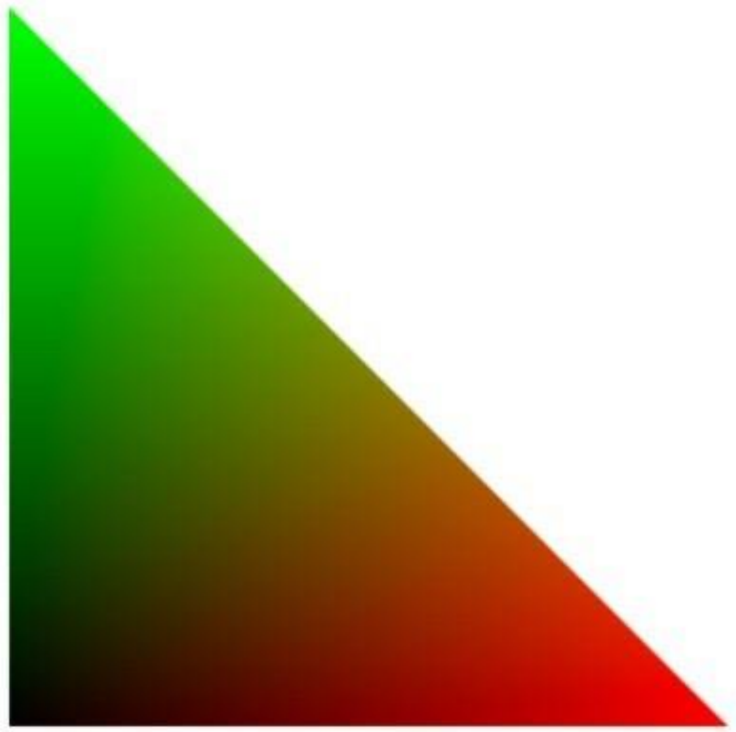
Example textured scene

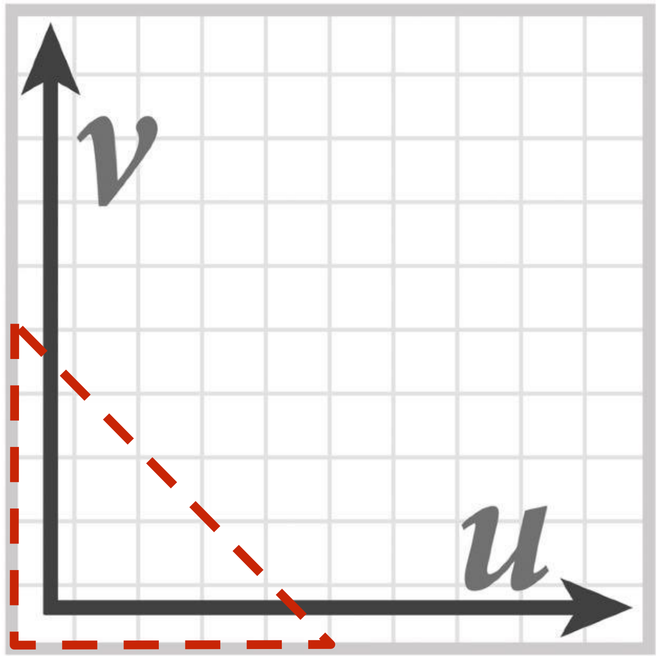


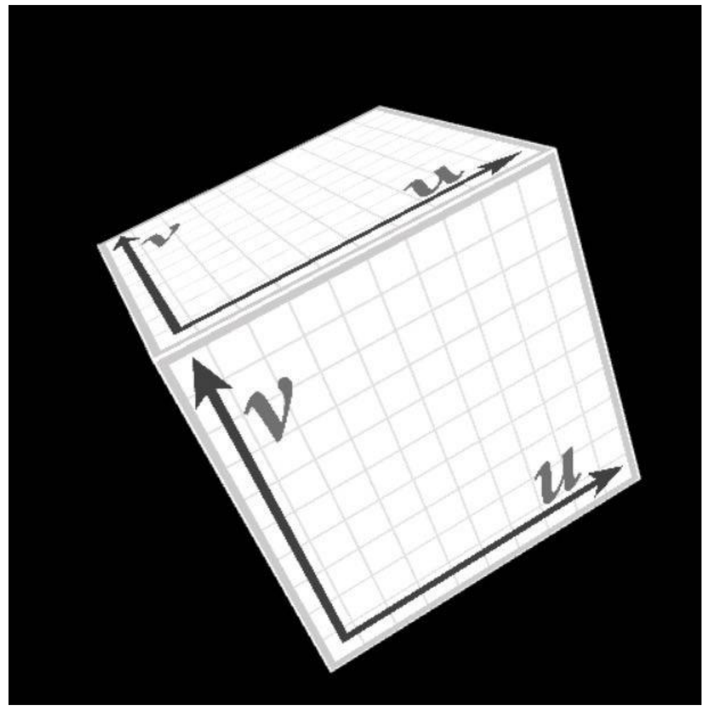
Example textures used in previous scene

Texture mapping: basic algorithm

- Basic algorithm for mapping texture image onto a surface:
  - For each color sample location (X,Y) in the image
    - Interpolate U and V texture coordinates across triangle to get texture coordinate value at (X,Y)
    - Sample texture map at location (U,V)
    - Set output image sample color to sampled texture value

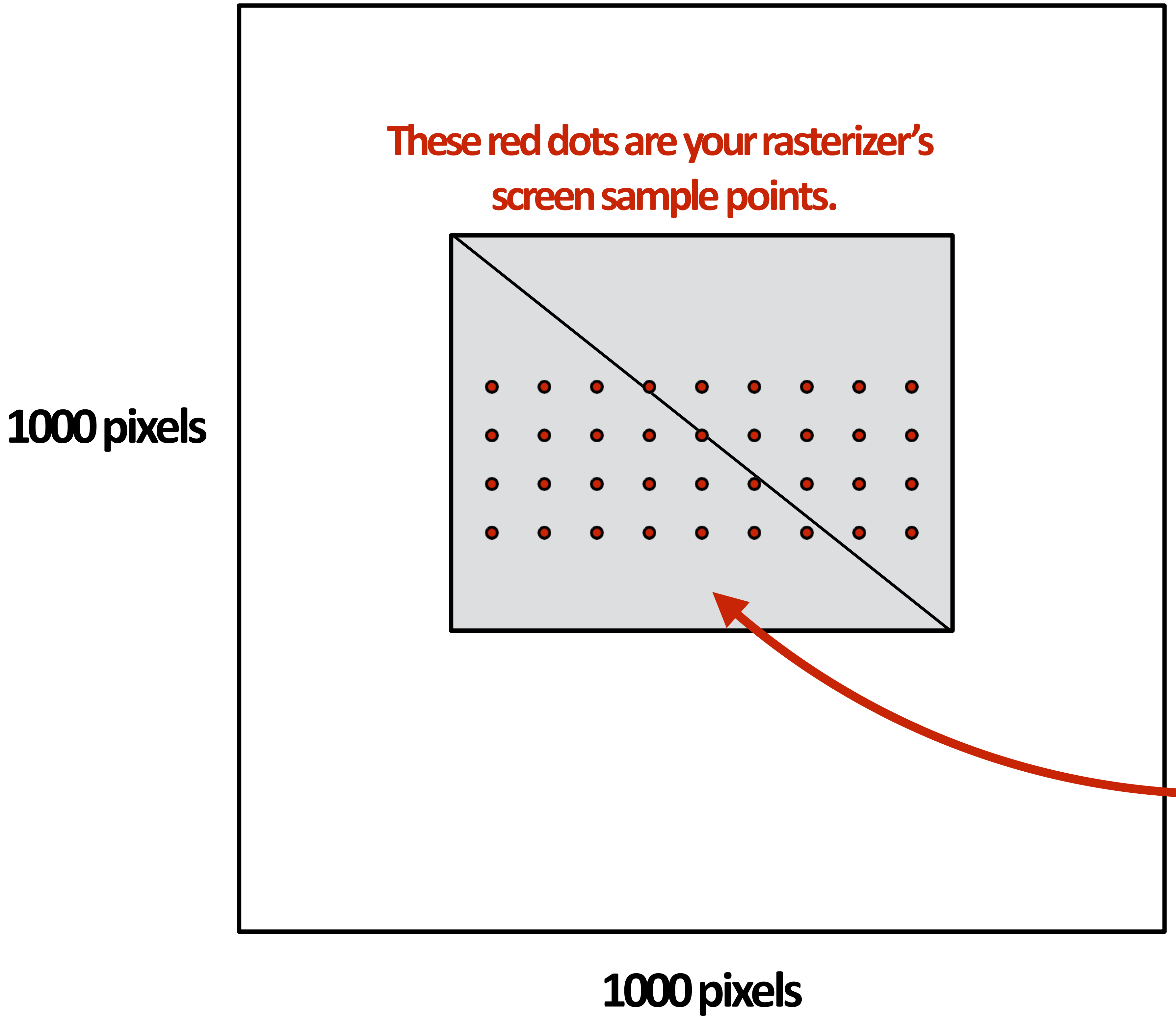






Thought experiment

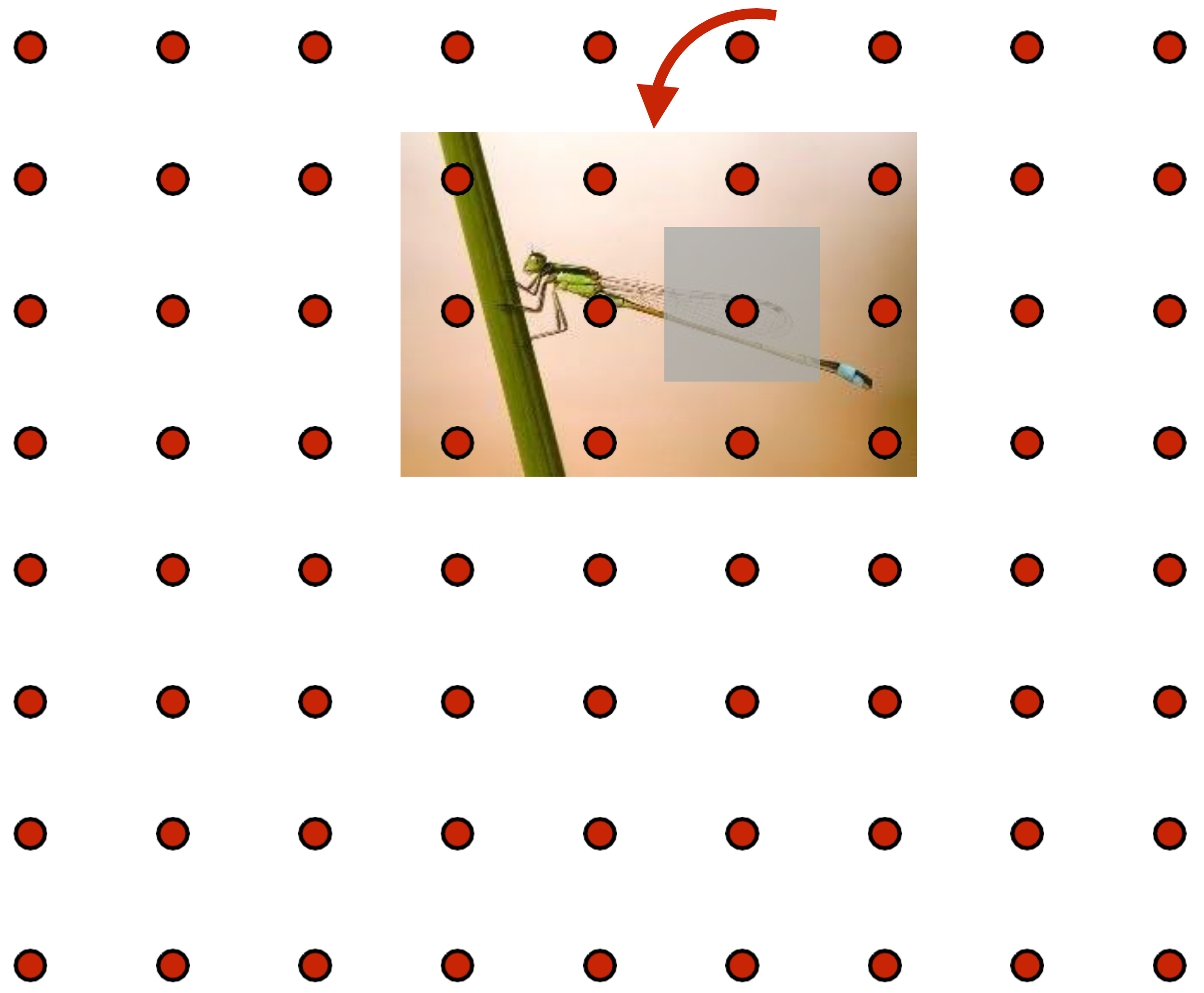
Imagine rendering a texture-mapped quadrilateral onto a 1000x1000 pixel output image



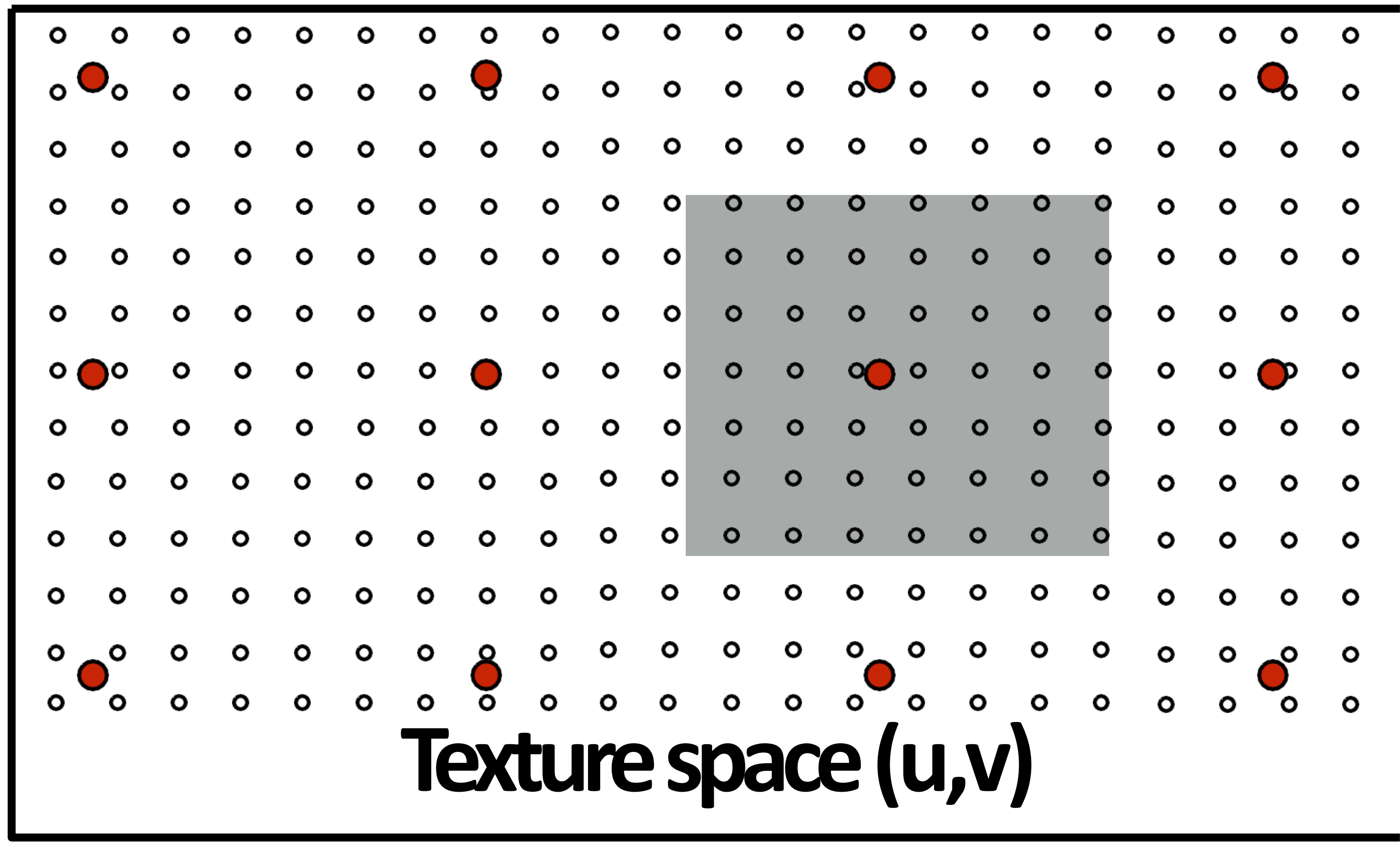
Let's also say the texture image is 1000x1000 as well.

Sampling rate on screen vs in texture: object zoomed out

The entire 1000x1000 texture is rendered into a small region of the screen.



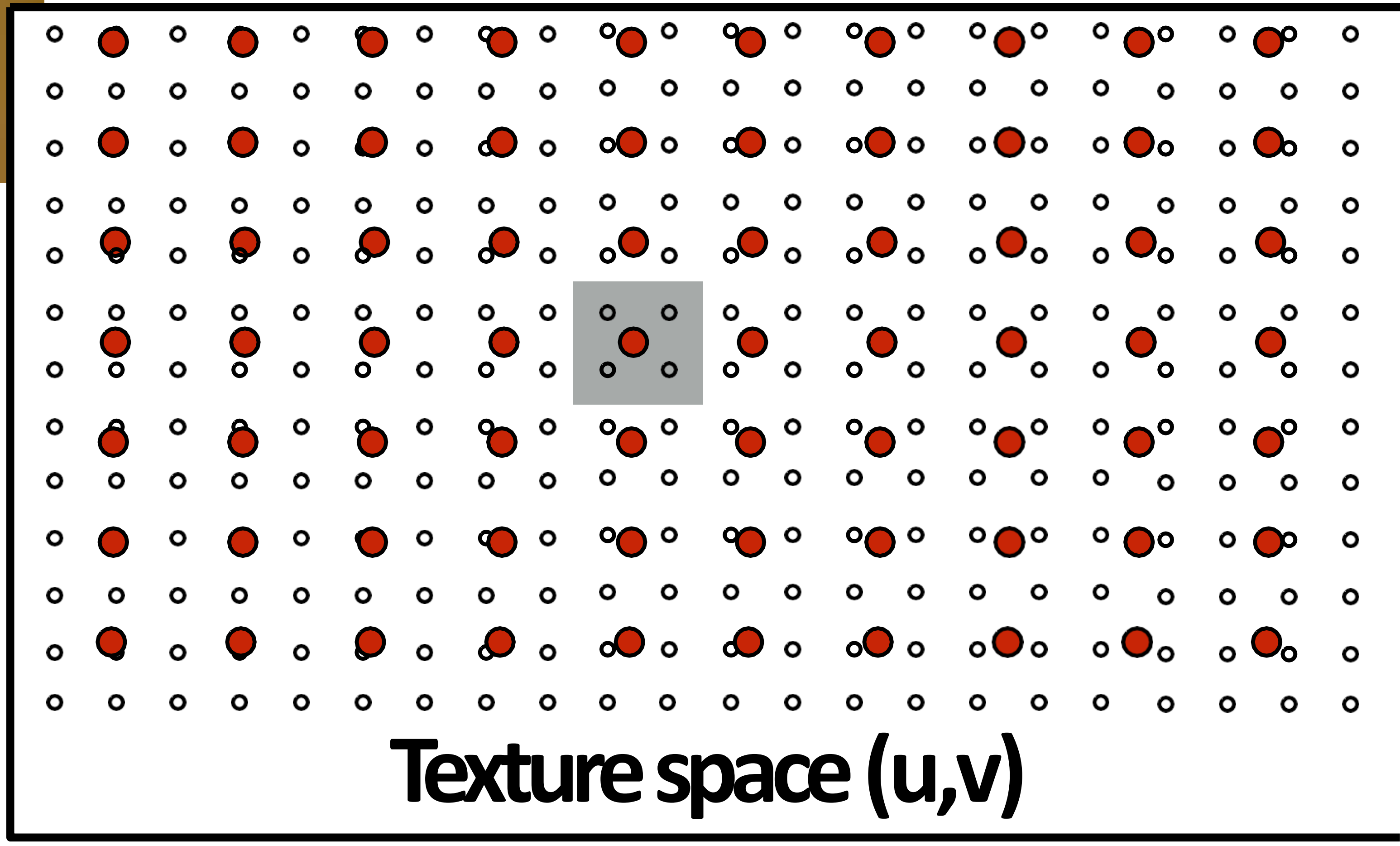
Texture is “minified” on screen



Red dots = samples on screen

White dots = texture map samples in texture space

Zooming in ...

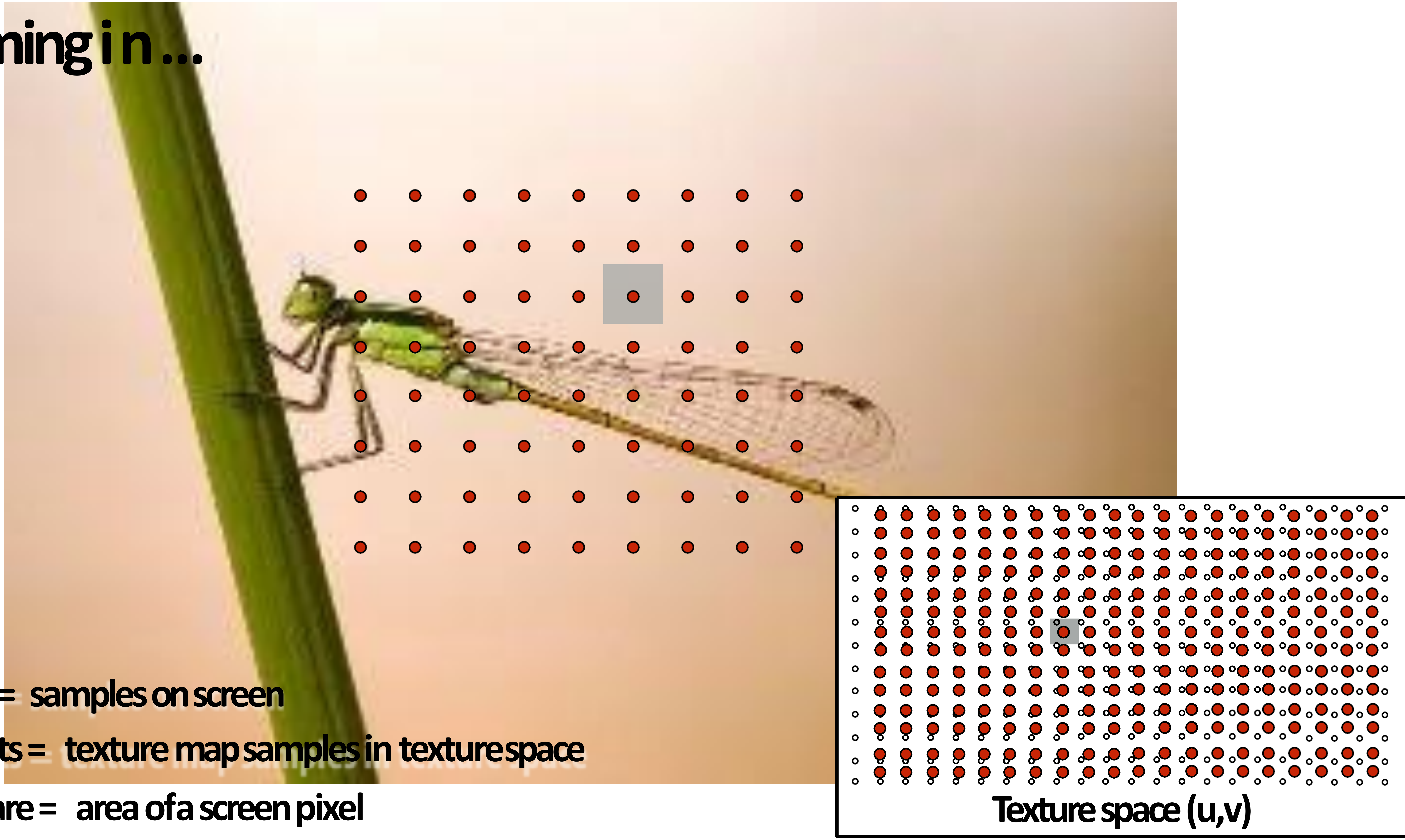


Red dots = samples on screen

White dots = texture map samples in texture space

Gray square = area of a screen pixel

Zooming in ...



Red dots = samples on screen

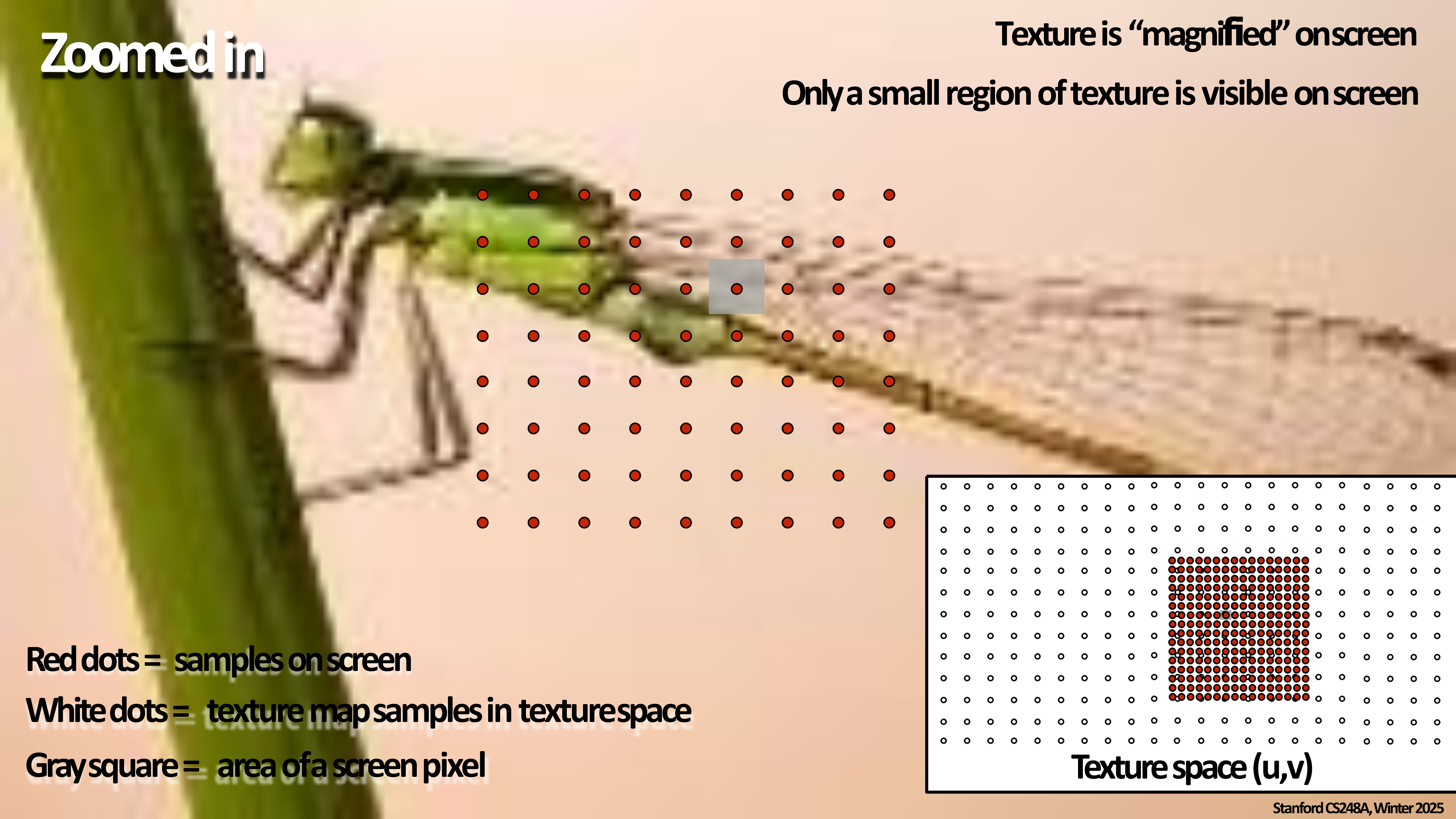
White dots = texture map samples in texture space

Gray square = area of a screen pixel

Zoomed in

Texture is “magnified” on screen

Only a small region of texture is visible on screen

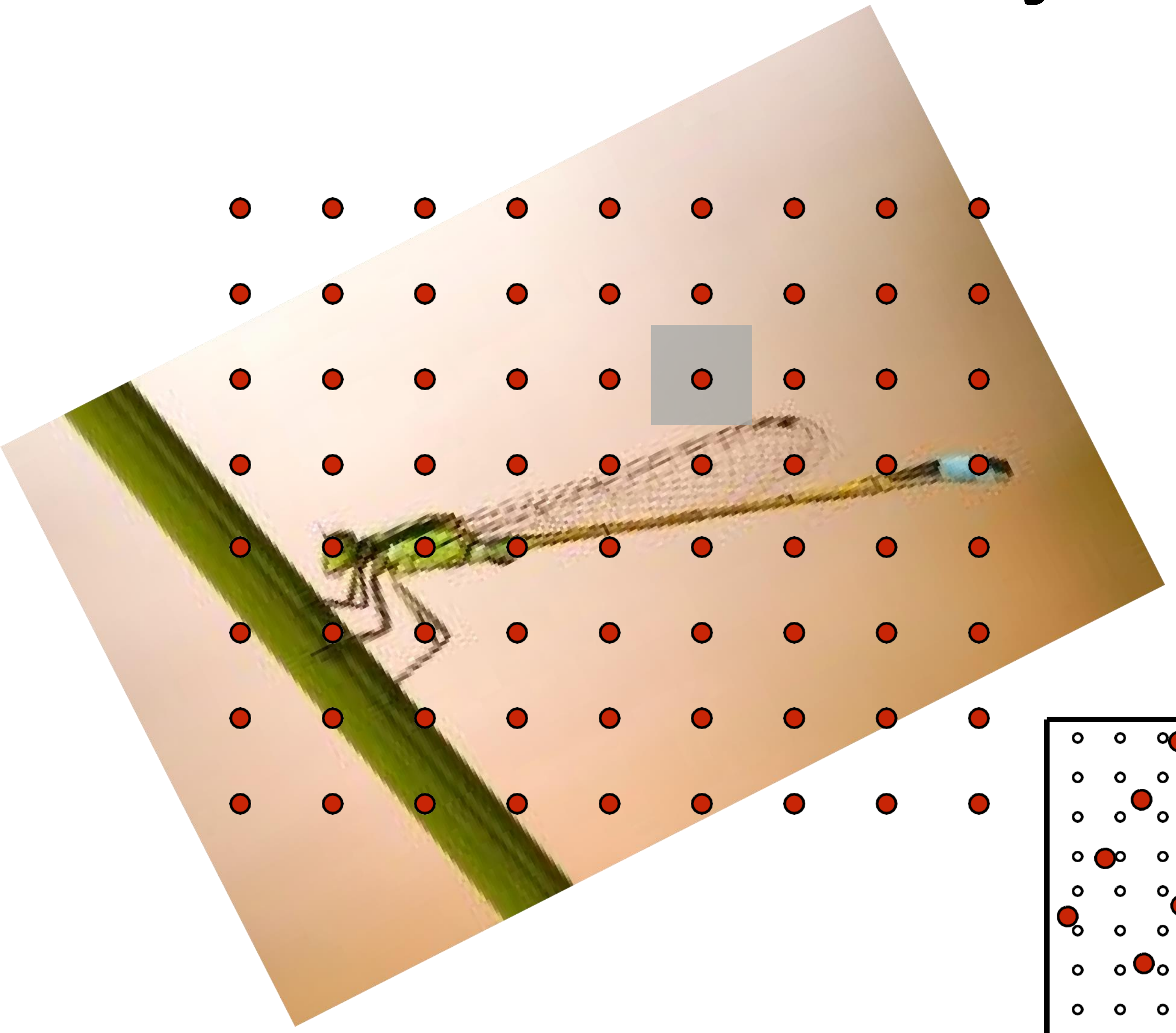


Red dots = samples on screen

White dots = texture map samples in texture space

Gray square = area of a screen pixel

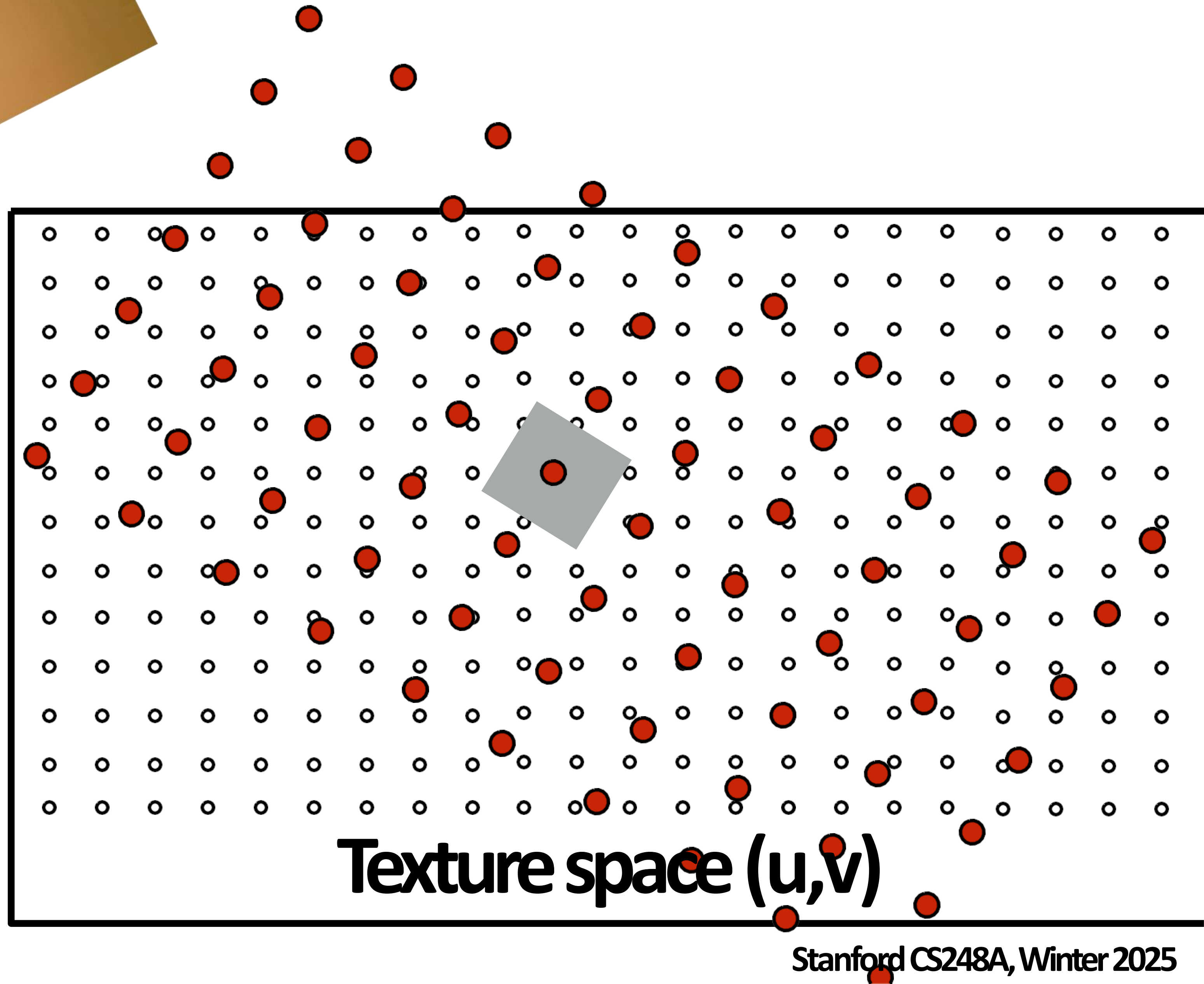
Sampling rate on screen vs in texture: object rotation



Red dots = samples on screen

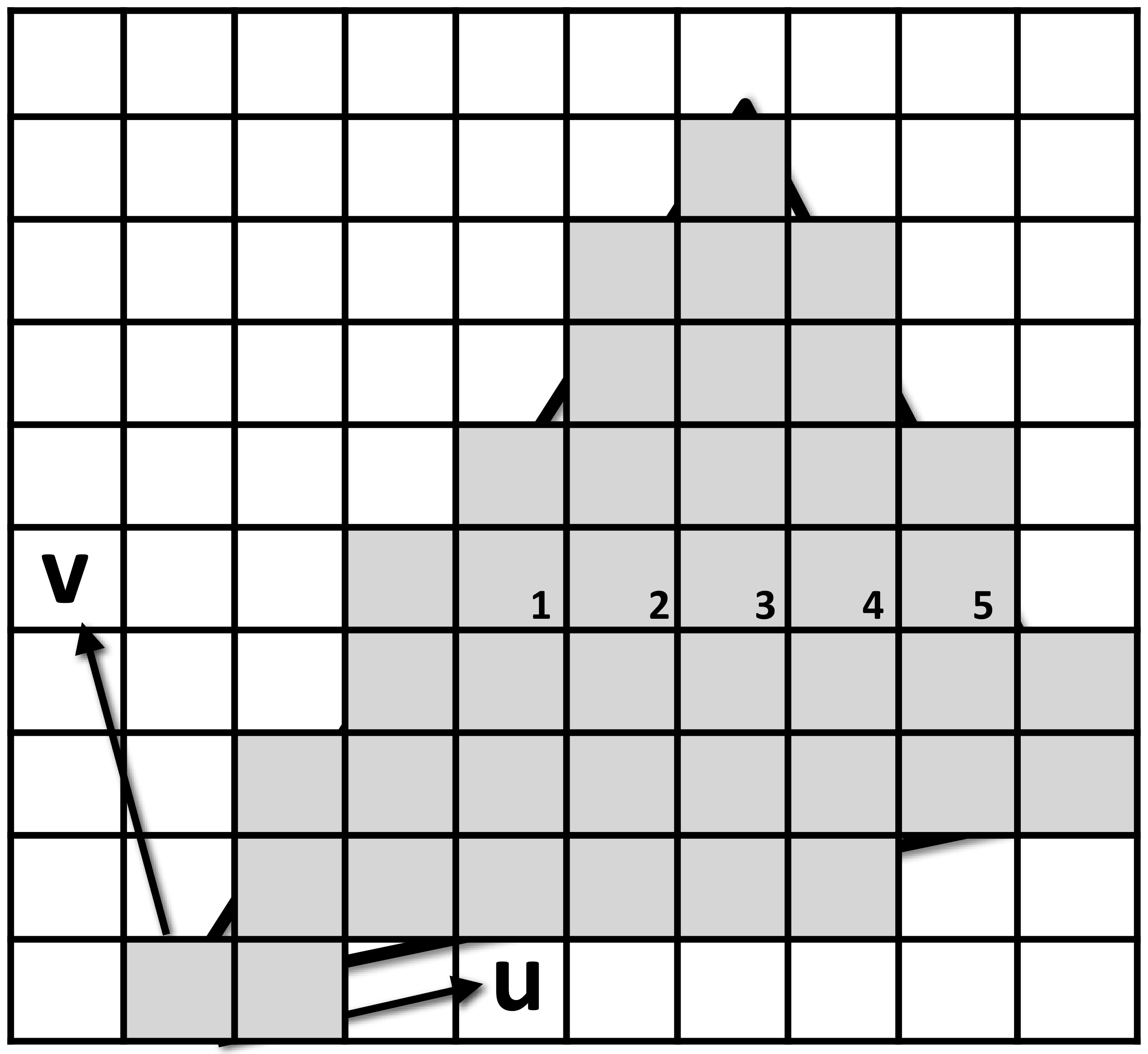
White dots = texture map samples in texture space

Gray square = area of a screen pixel



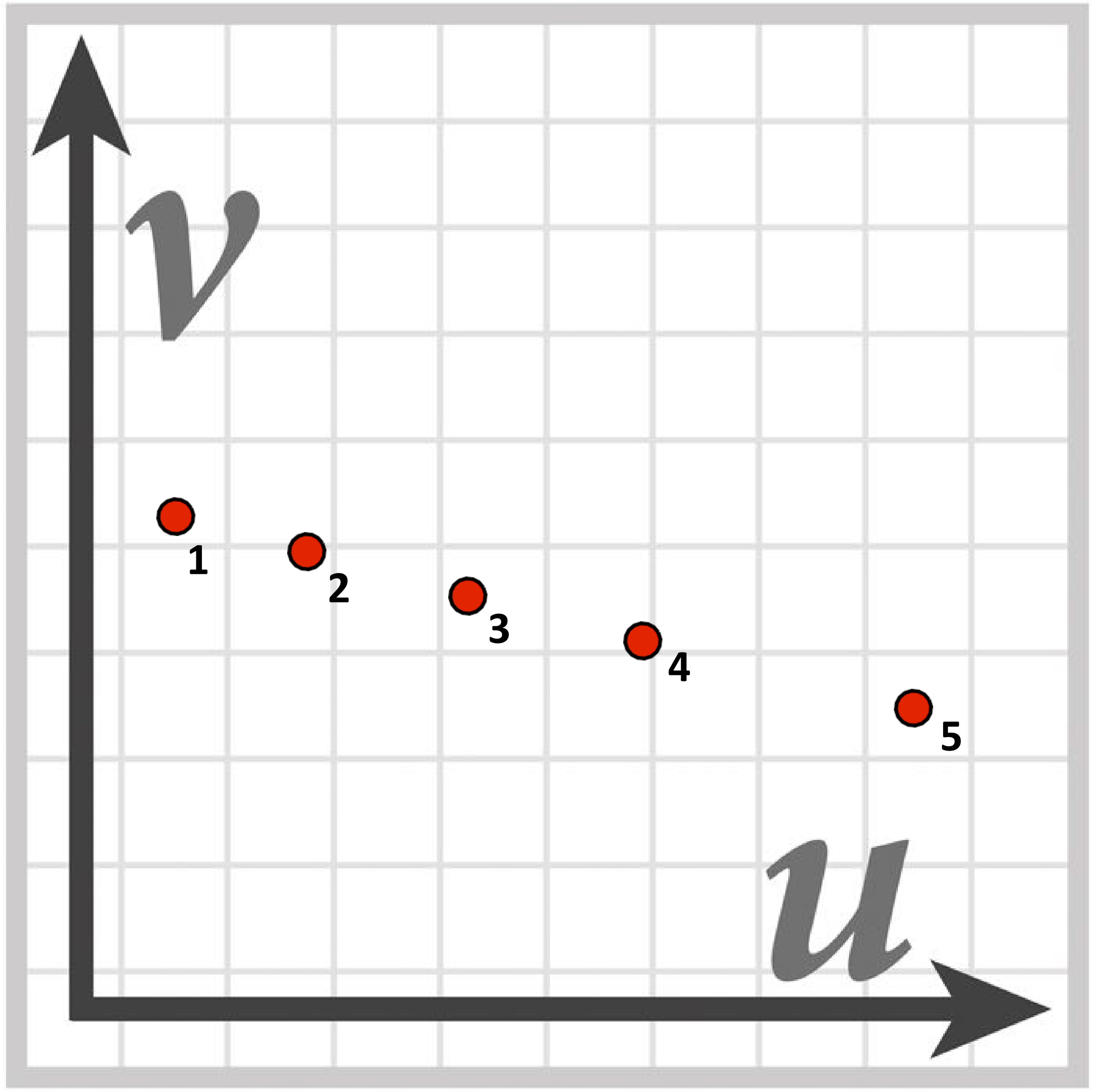
Equally spaced samples on screen != equally spaced samples in texture space

Sample positions in XY screen space



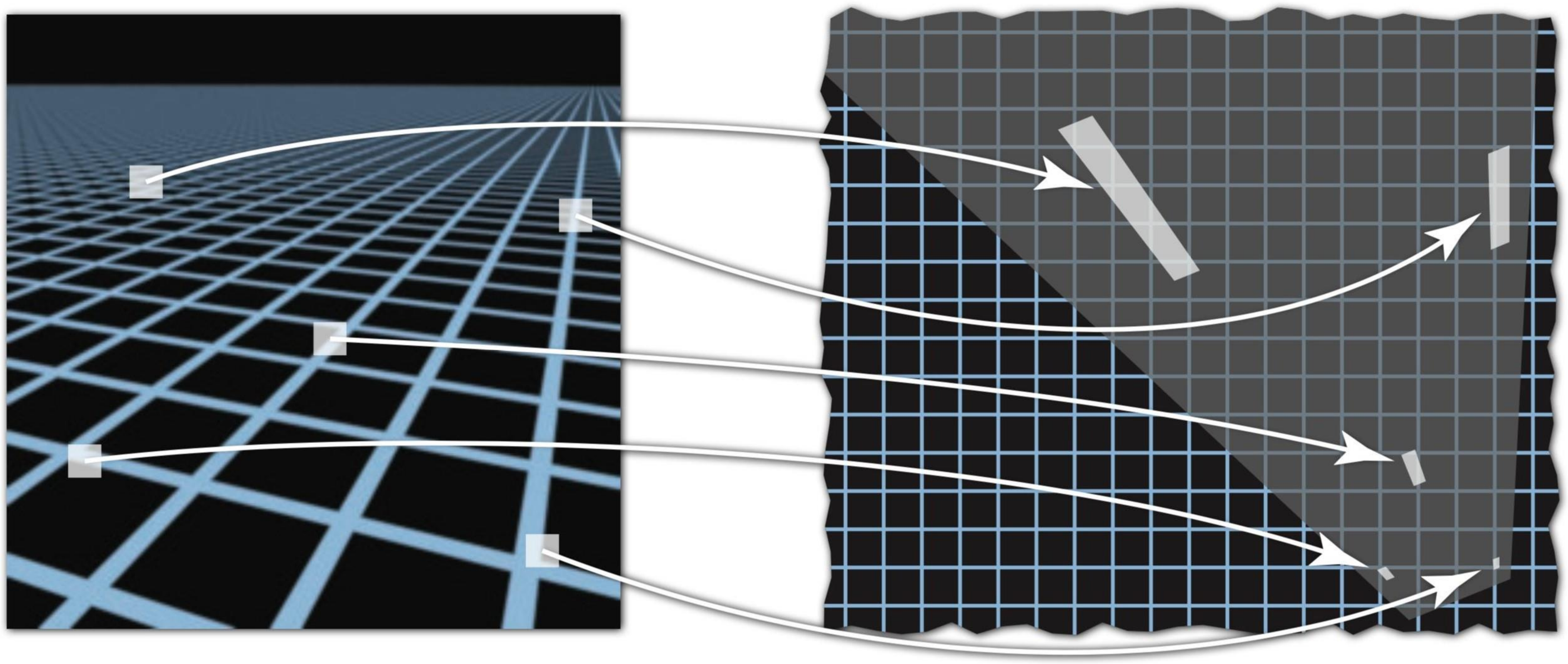
Sample positions are uniformly distributed in screen space
(rasterizer samples triangle's appearance at these locations)

Sample positions in texture space



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

Screen pixel footprint in texture space

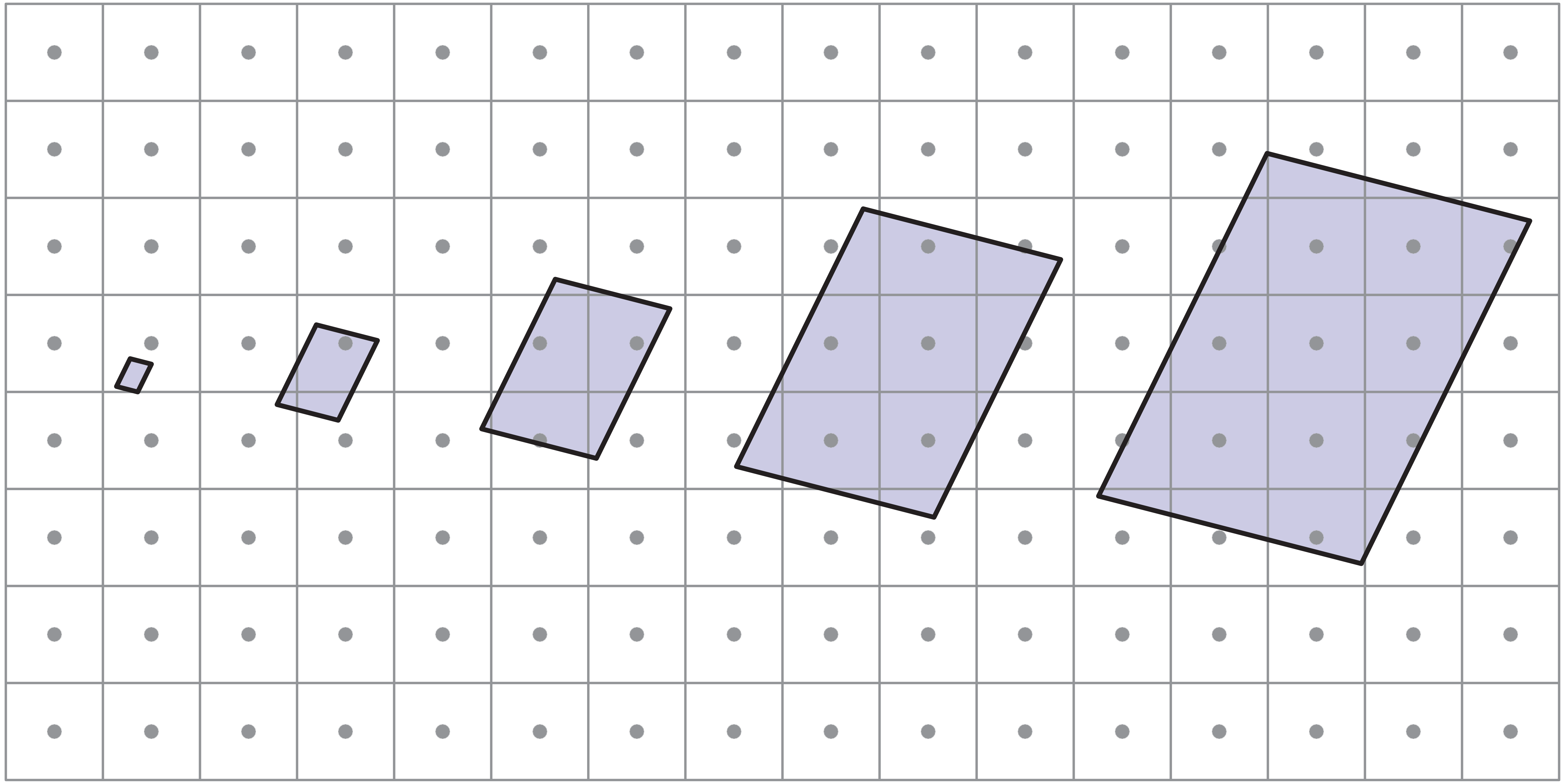


Screen space

Texture space

Texture sampling pattern not rectilinear or isotropic

Screen pixel footprint in texture space



Upsampling
(Magnification)

Camera zoomed in close to object

Downsampling
(Minification)

Camera far away from object

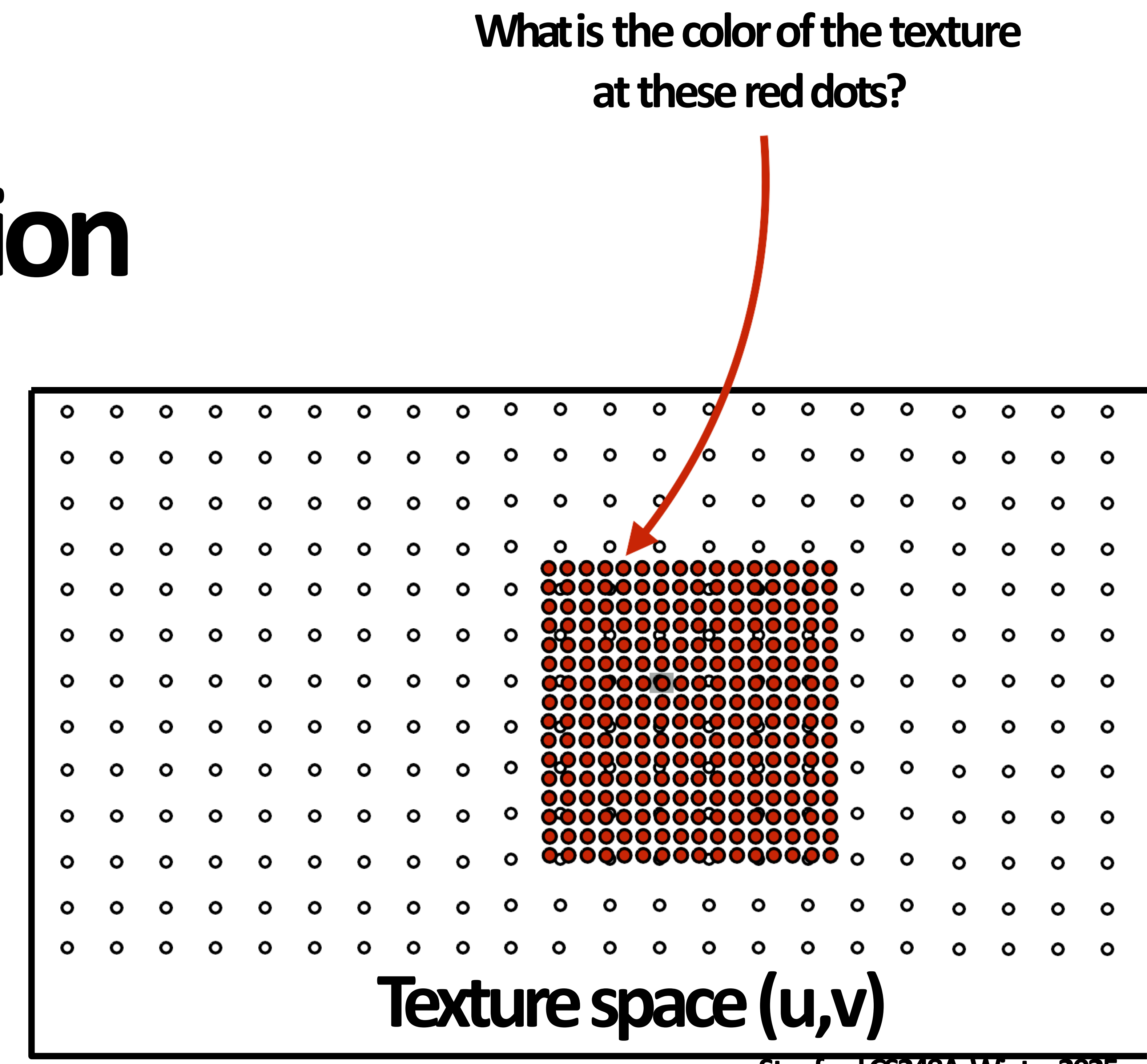
Screen pixel area vs texel area

- 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

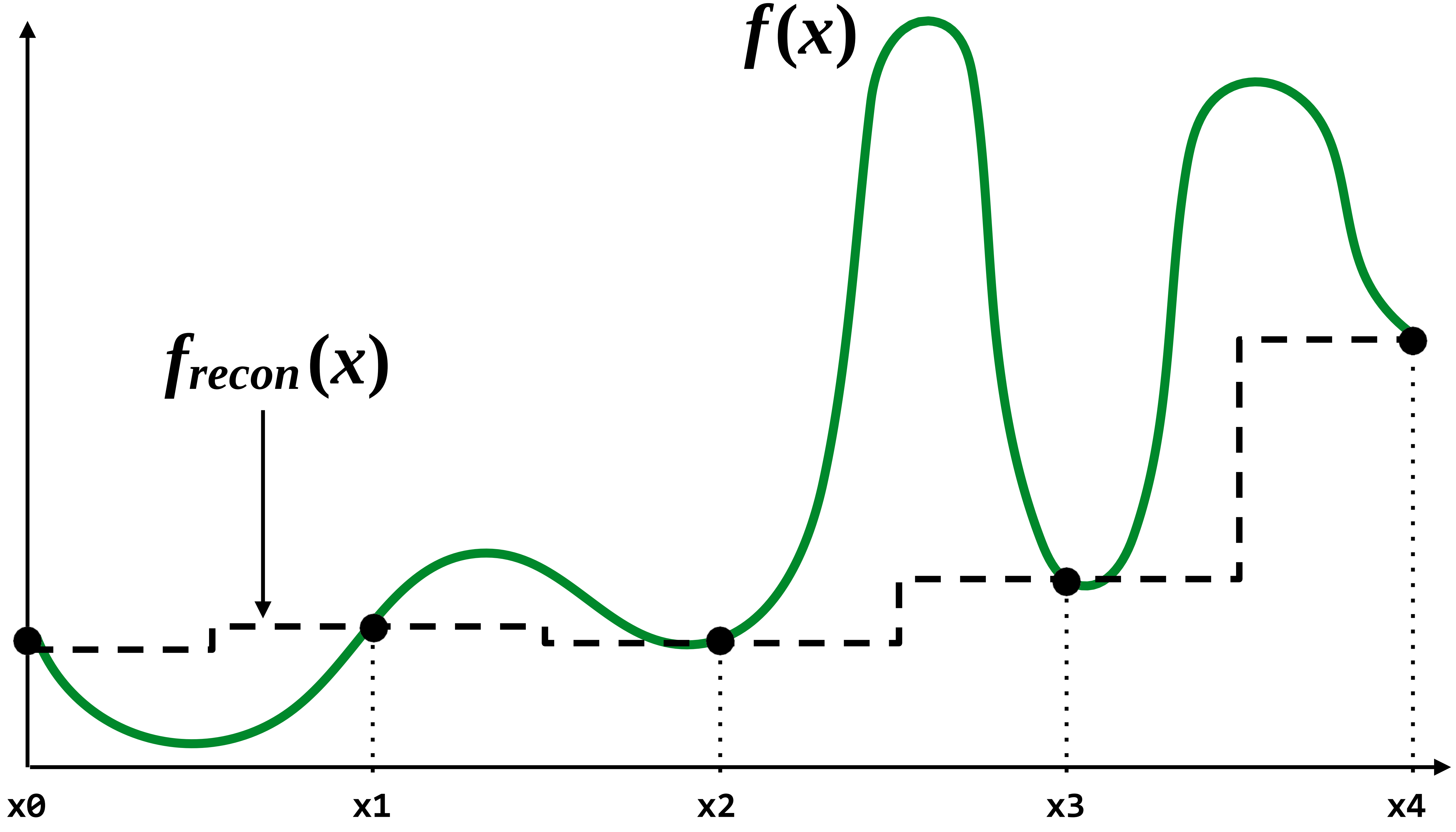
Texture magnification



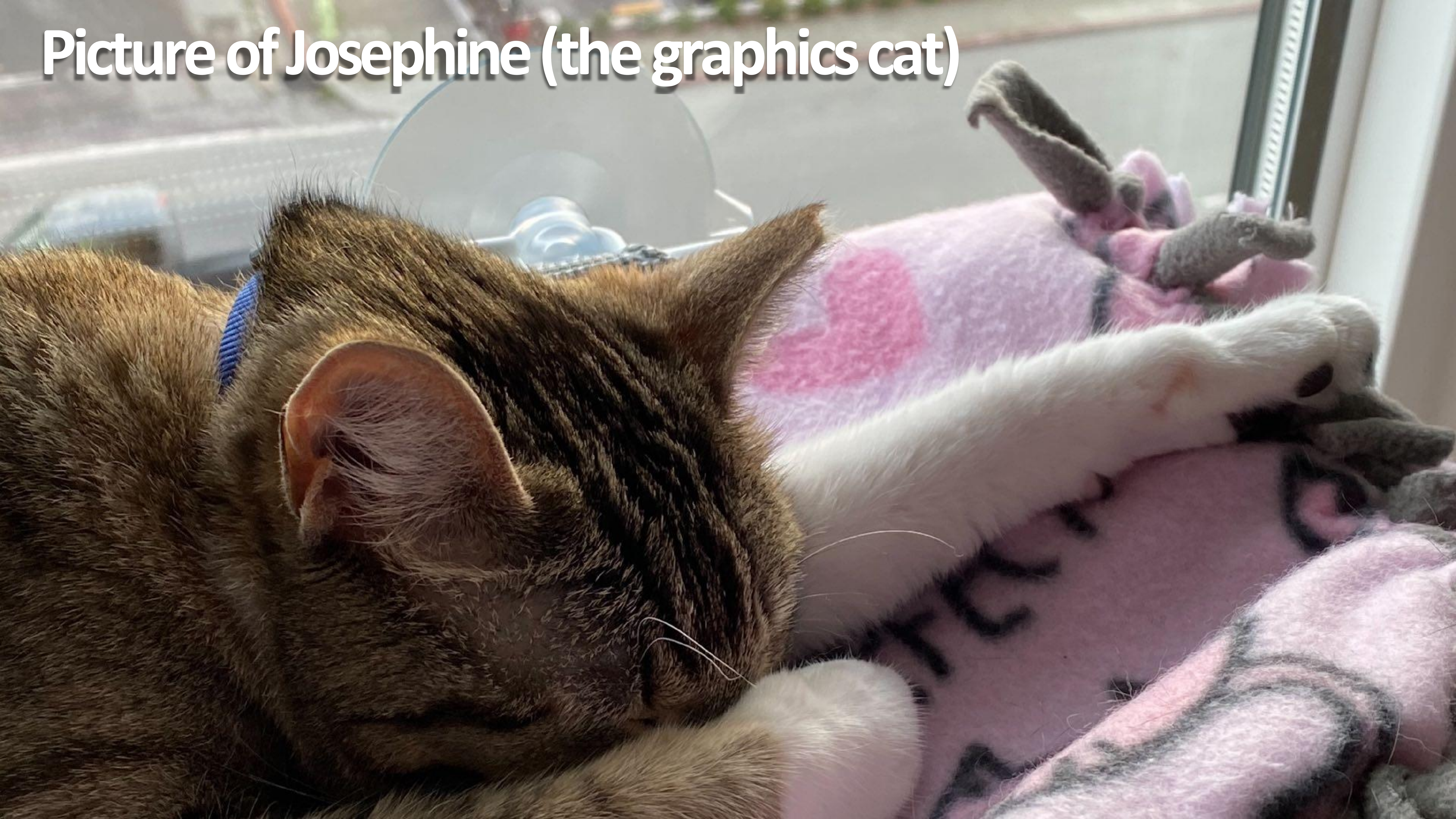
Review: piecewise constant approximation

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

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



Picture of Josephine (the graphics cat)



Texture magnification (nearest)

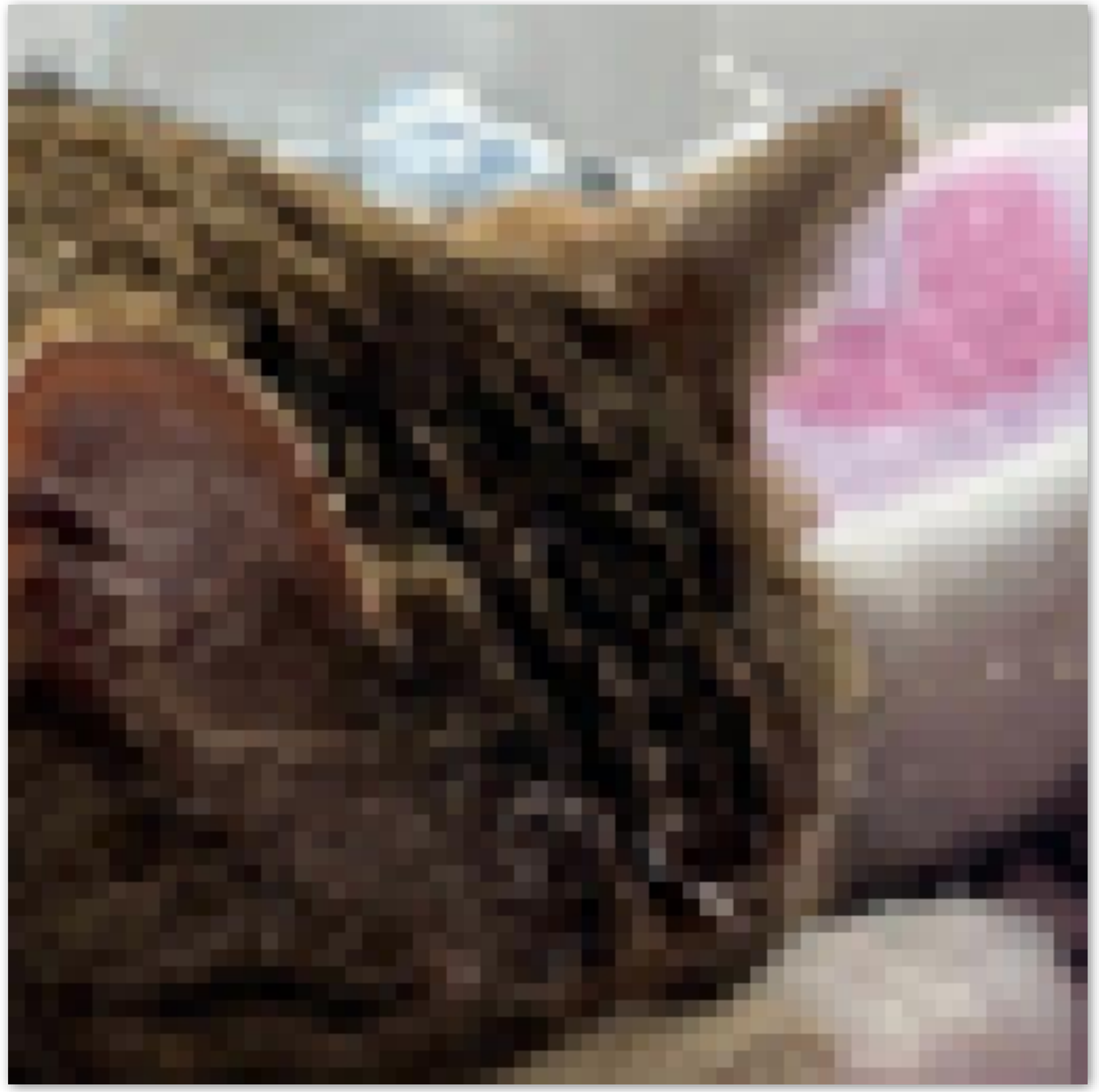
Texture magnification (nearest)

Texture magnification (nearest)

Texture magnification (nearest)

Texture magnification

- Generally don't want this situation — it means we have insufficient texture resolution
- Magnification involves interpolation of values in texture map (below: three different interpolation kernel functions)



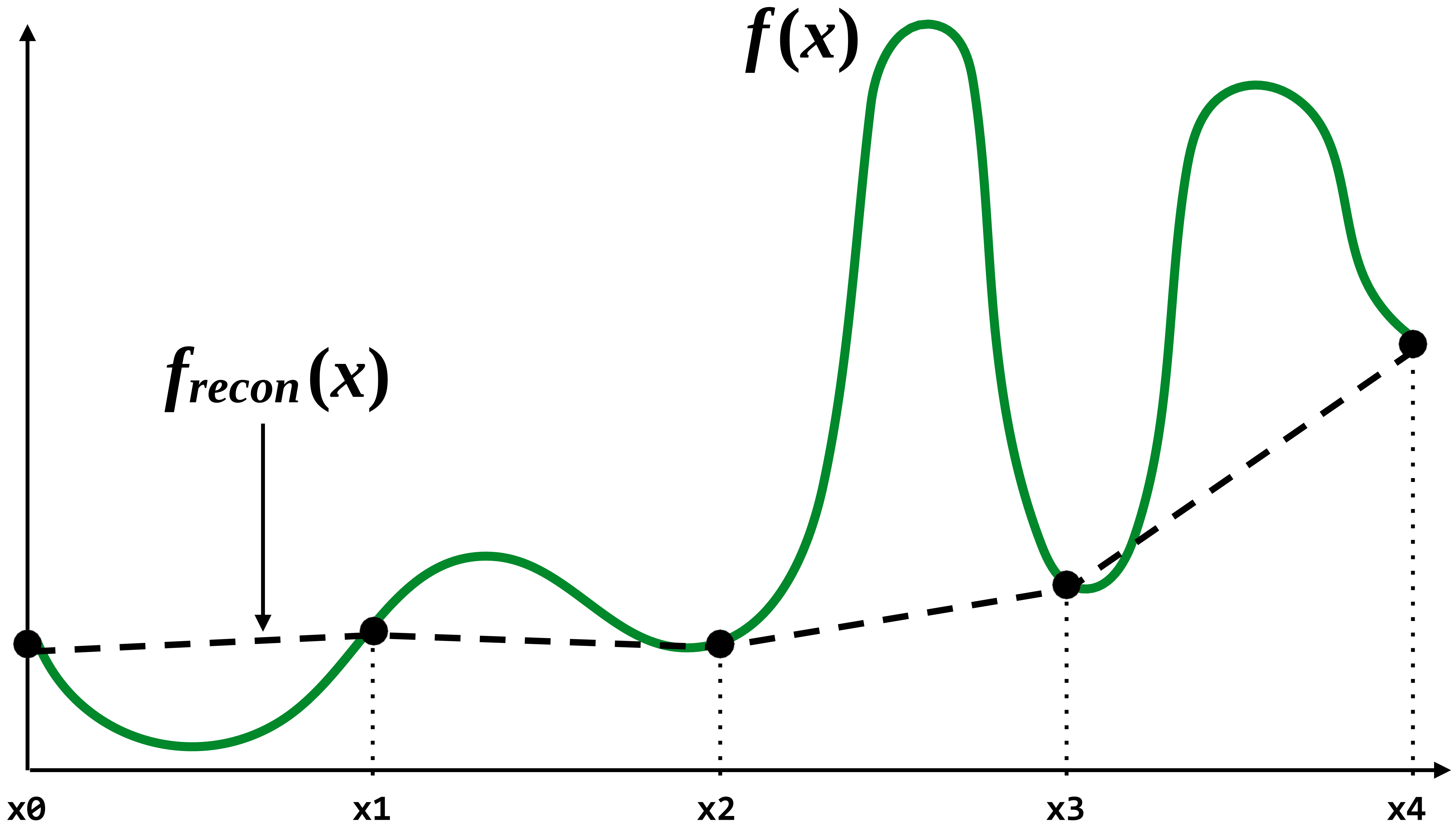
Nearest sample

Bilinear

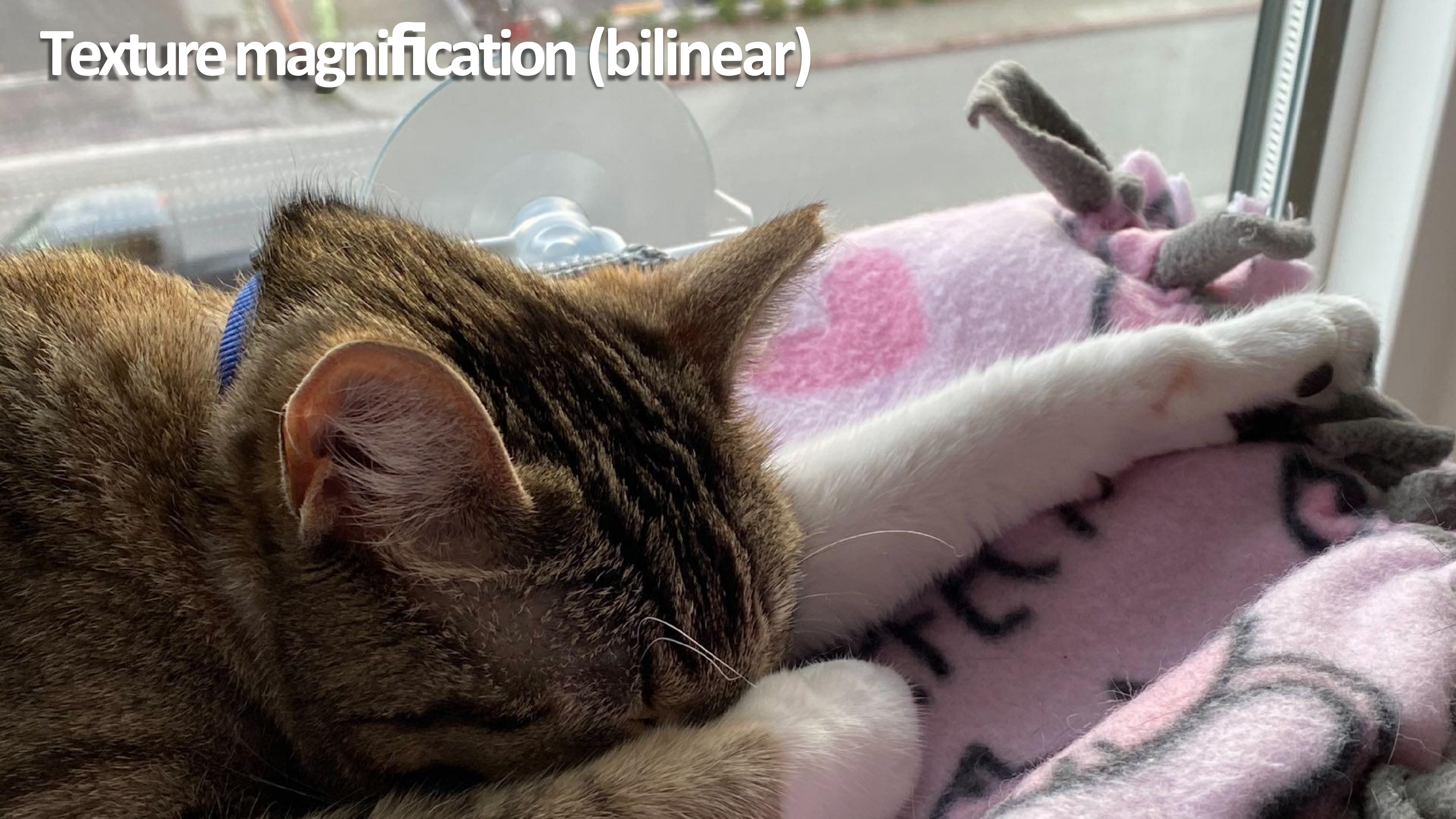
Bicubic

Review: piecewise linear approximation

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



Texture magnification (bilinear)

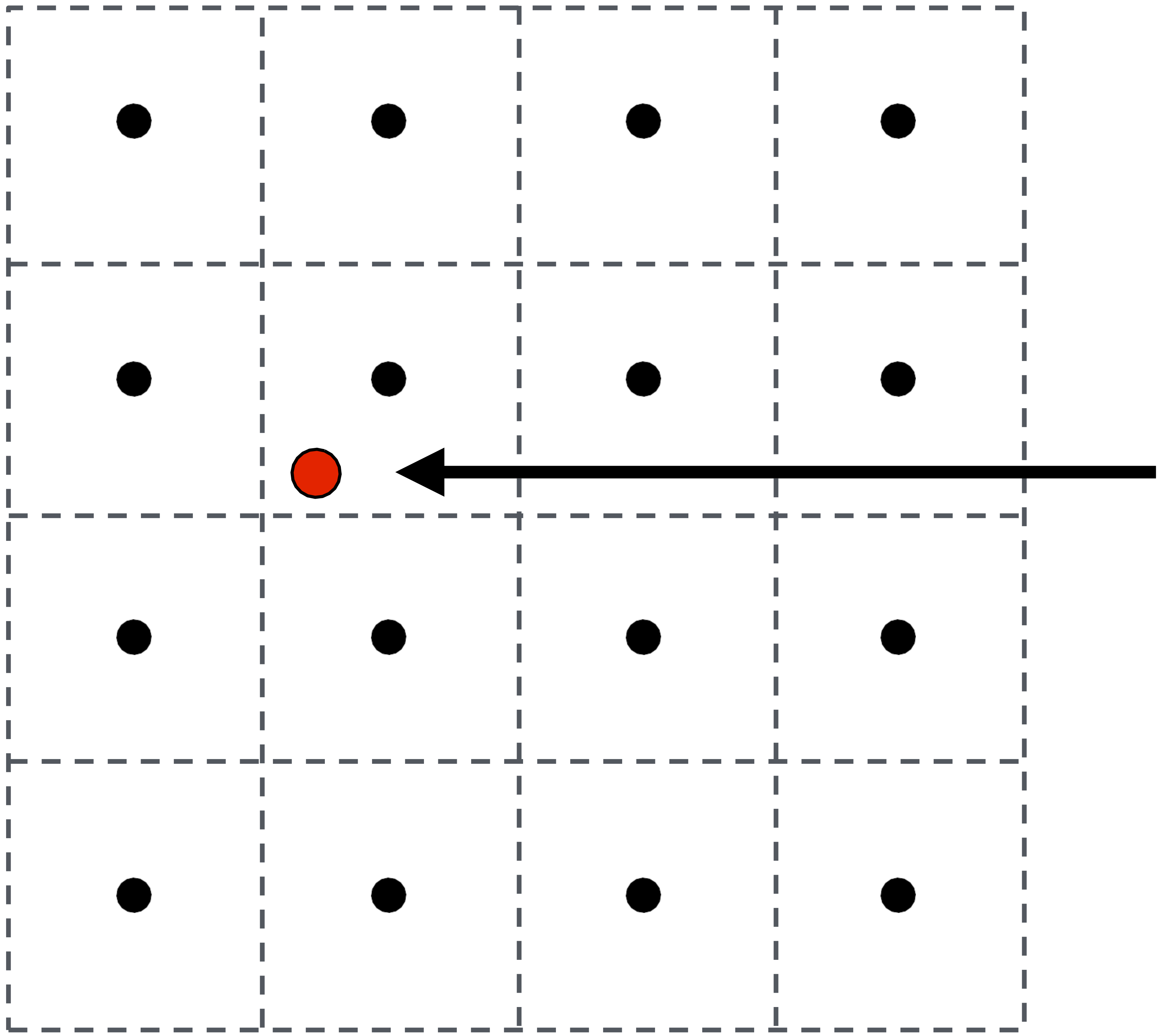


Texture magnification (bilinear)

Texture magnification (bilinear)

Texture magnification (bilinear)

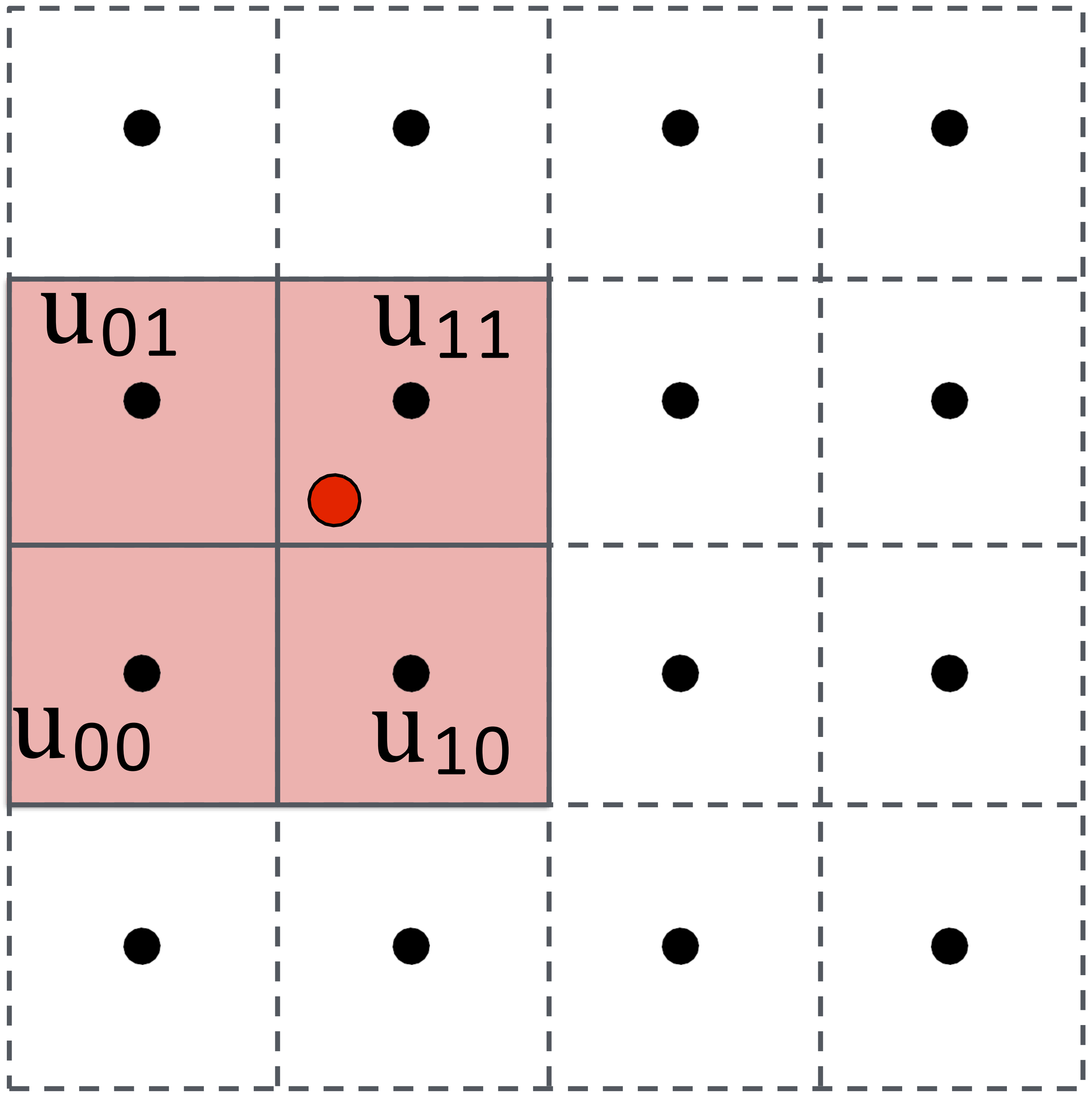
Bilinear interpolation



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

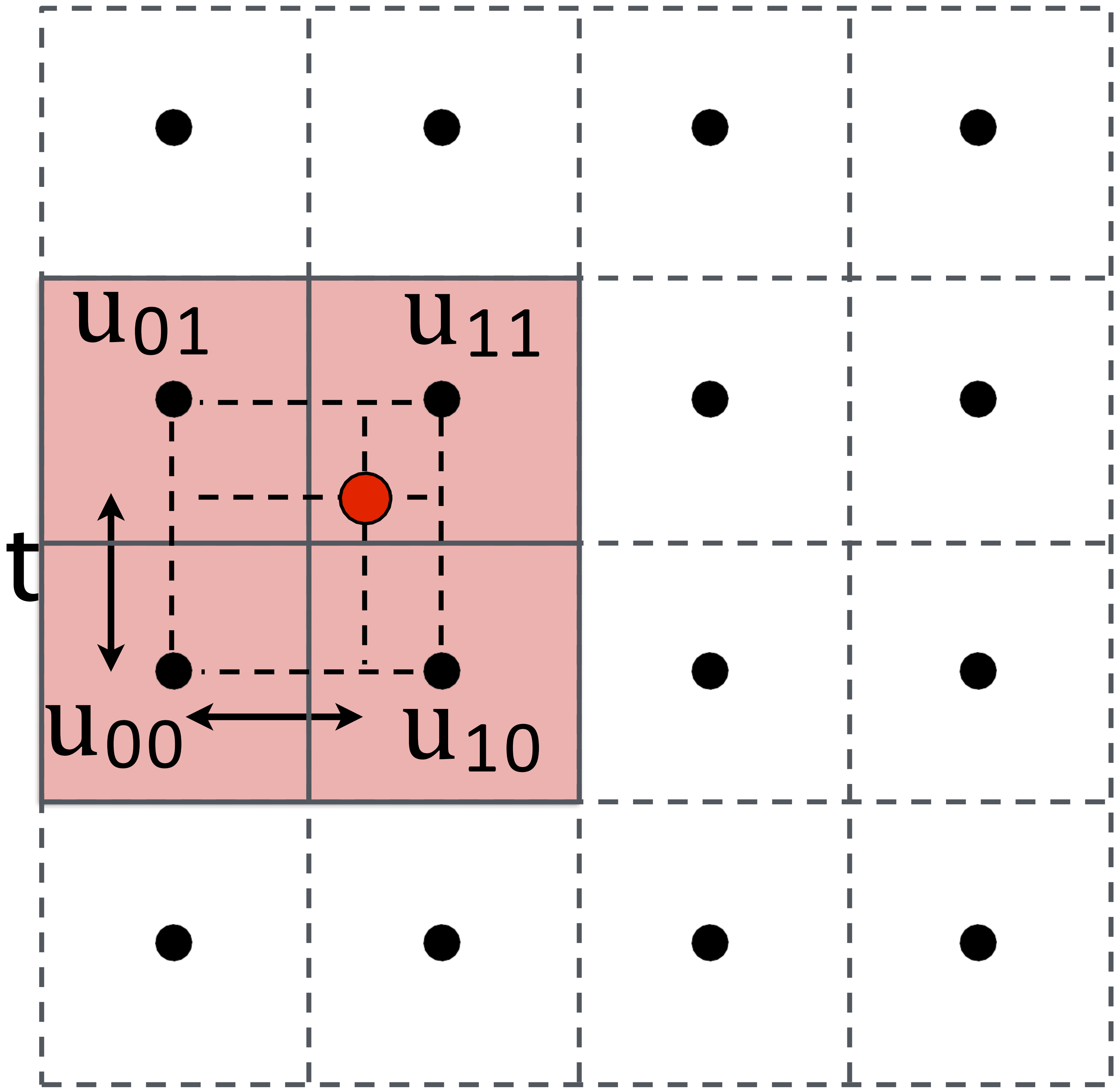
Black points indicate texture sample locations

Bilinear interpolation



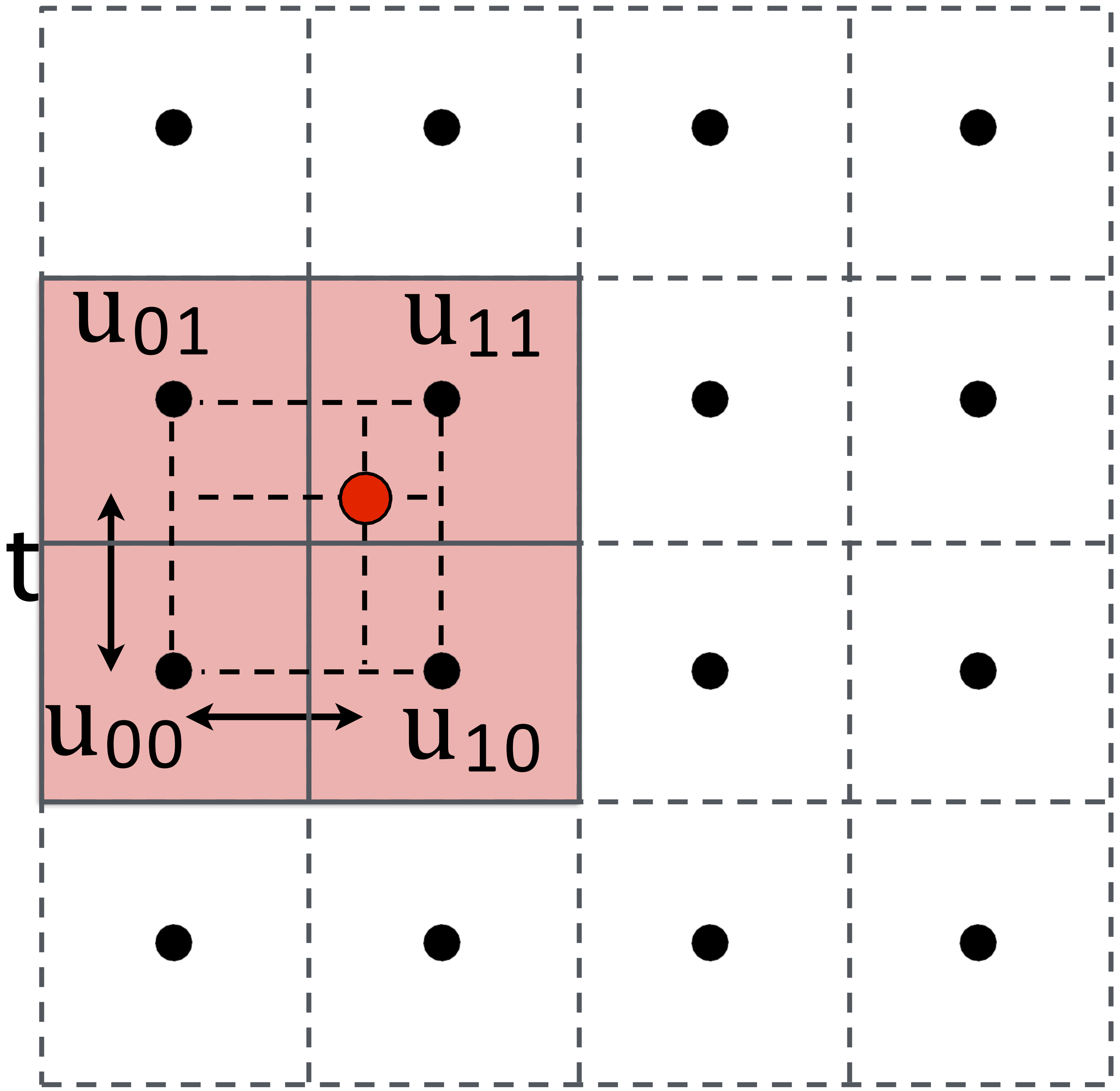
Take 4 nearest sample locations, with texture values as labeled.

Bilinear interpolation



And fractional offsets, (s,t) as shown

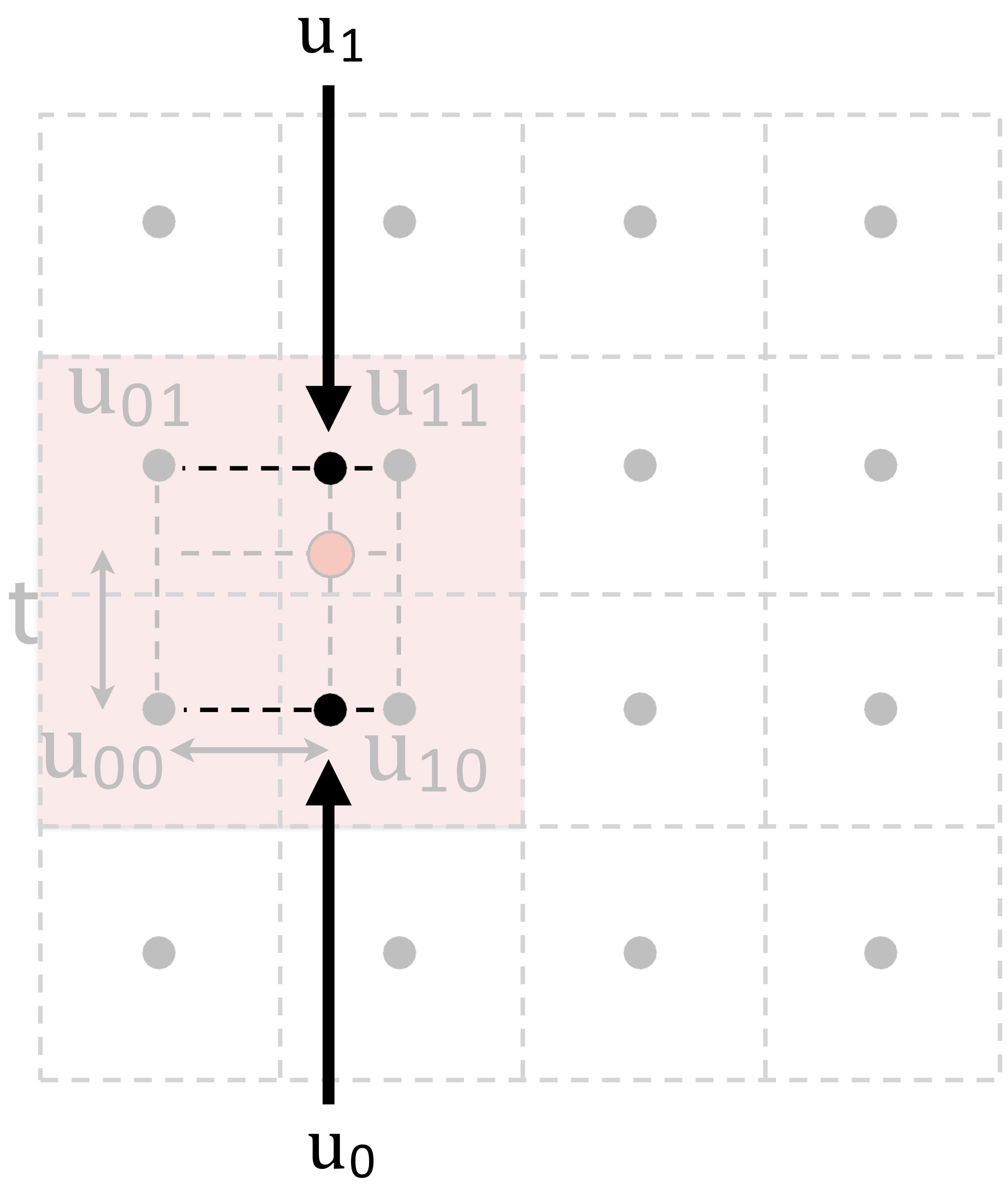
Bilinear interpolation



Linear interpolation (1D)

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

Bilinear interpolation



Linear interpolation (1D)

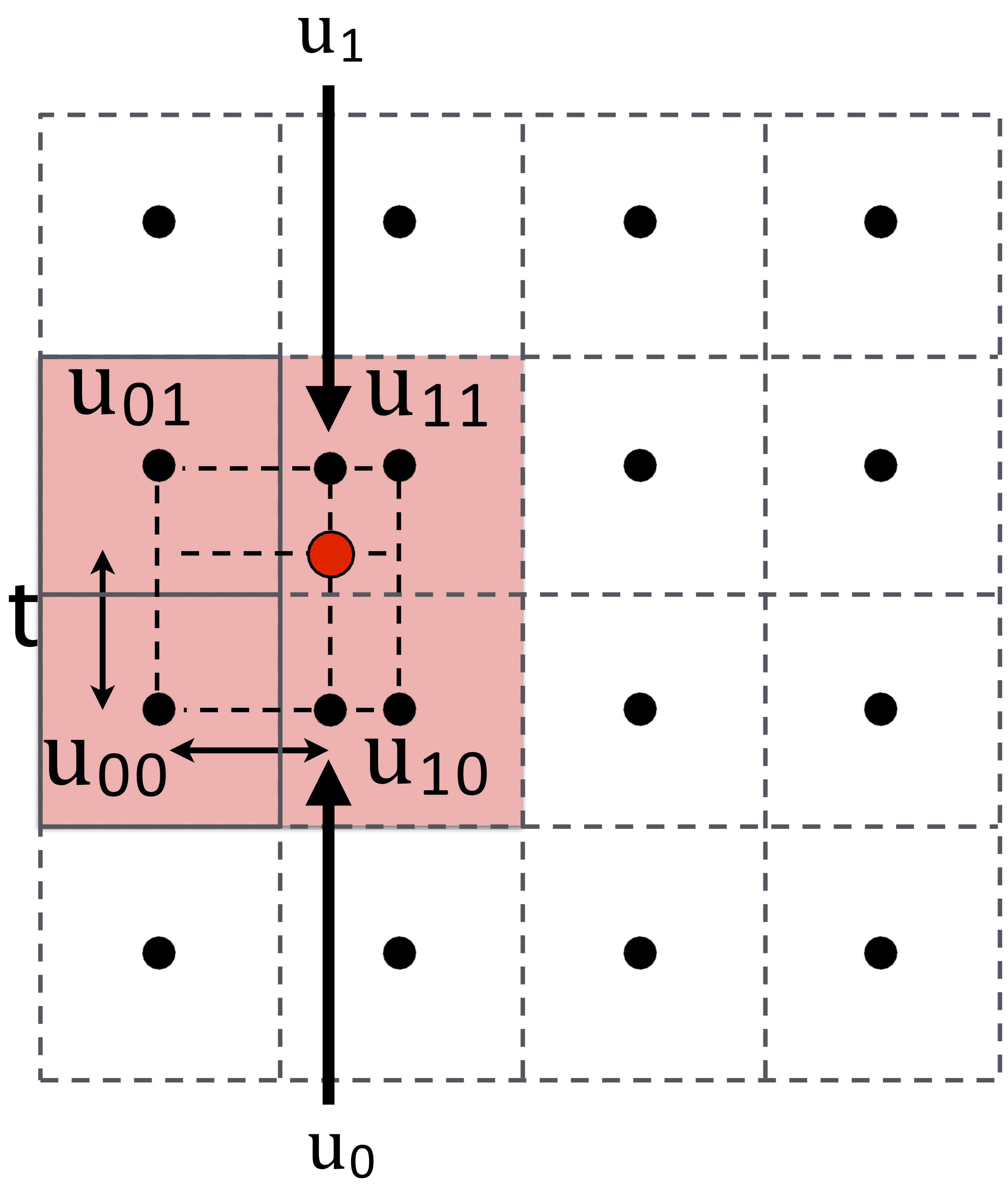
$$\mathrm{lerp}(x, v_0, v_1) = v_0 + x(v_1 - v_0)$$

Two helper lerps (horizontal)

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

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

Bilinear interpolation



Linear interpolation (1D)

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

Two helper lerps

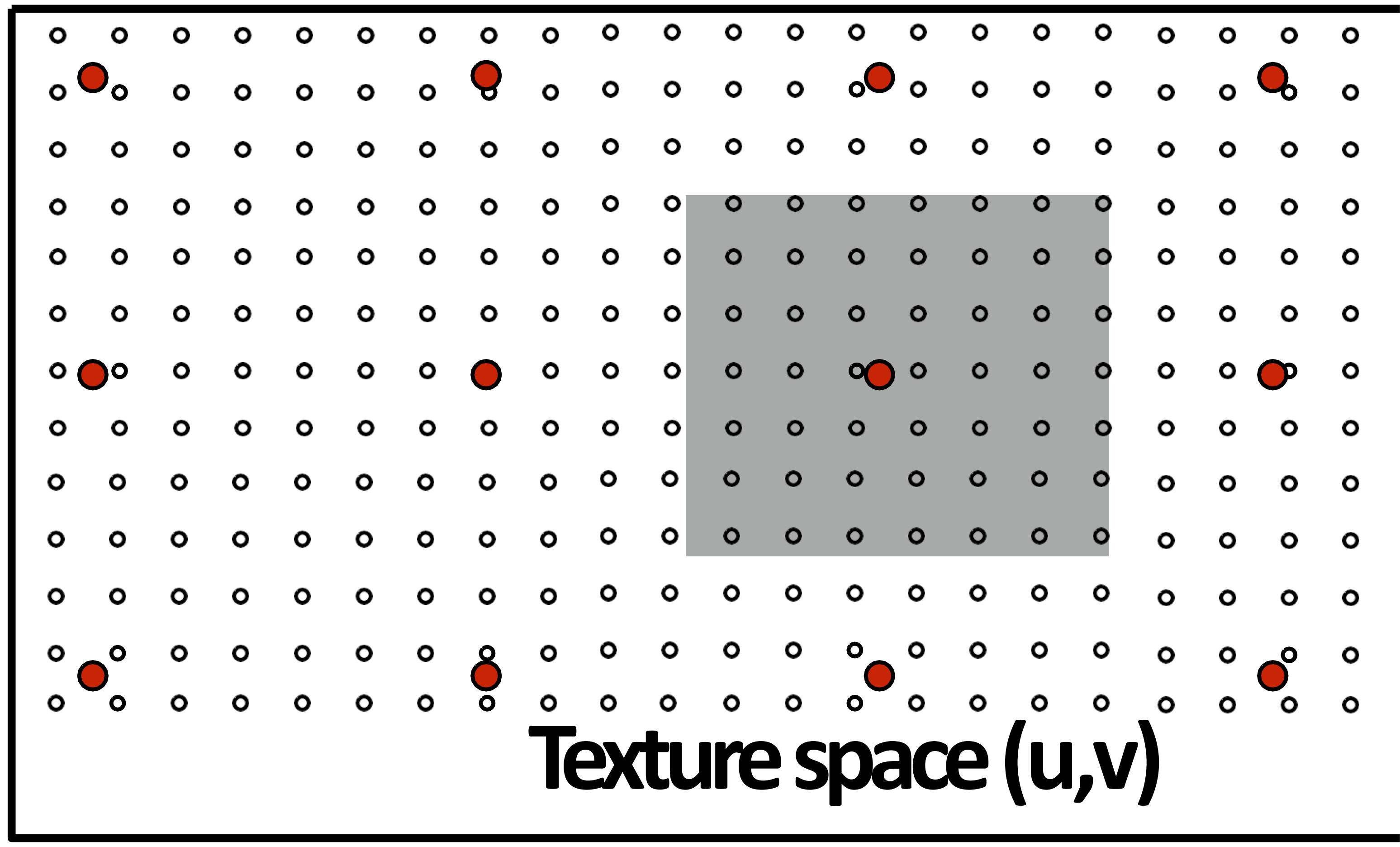
$$u_0 = \mathrm{lerp}(s, u_{00}, u_{10})$$

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

Final vertical lerp, to get result:

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

Texture minification



Applying textures is a form of sampling!

t(u,v)

Minification of Josephine

Imagine the texture map is 9x9

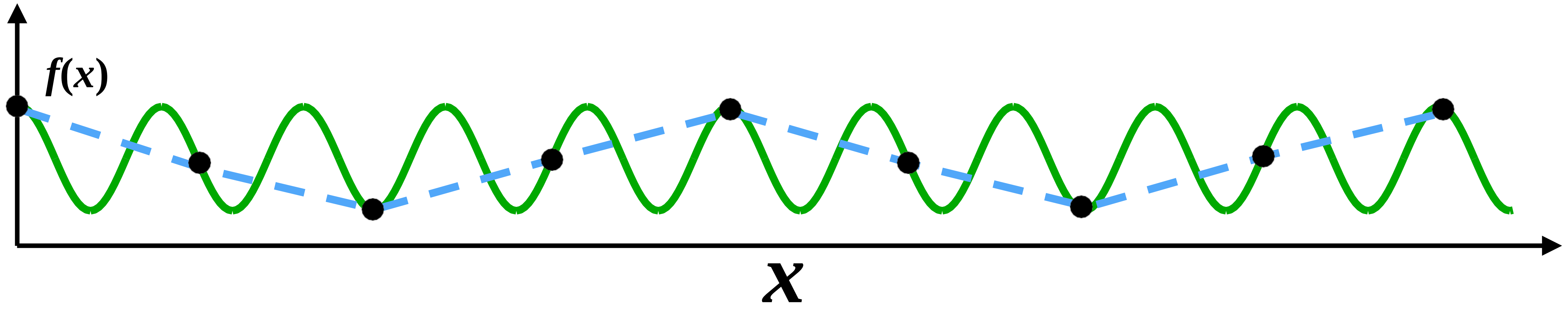
And is applied to a quad that spans a 3x3 pixel region of screen.

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

When a texture is minimized, the texture map is sampled sparsely!

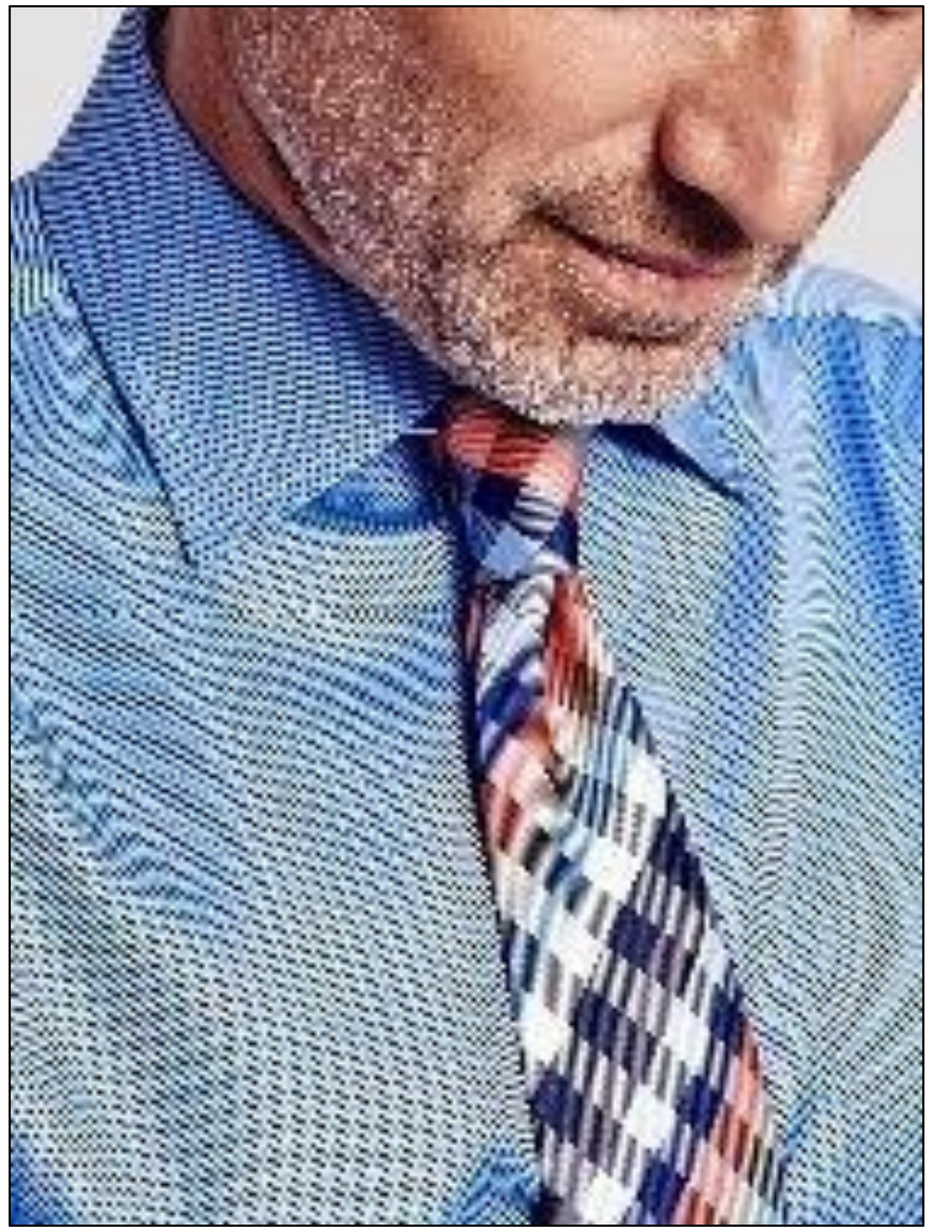
Recall: aliasing

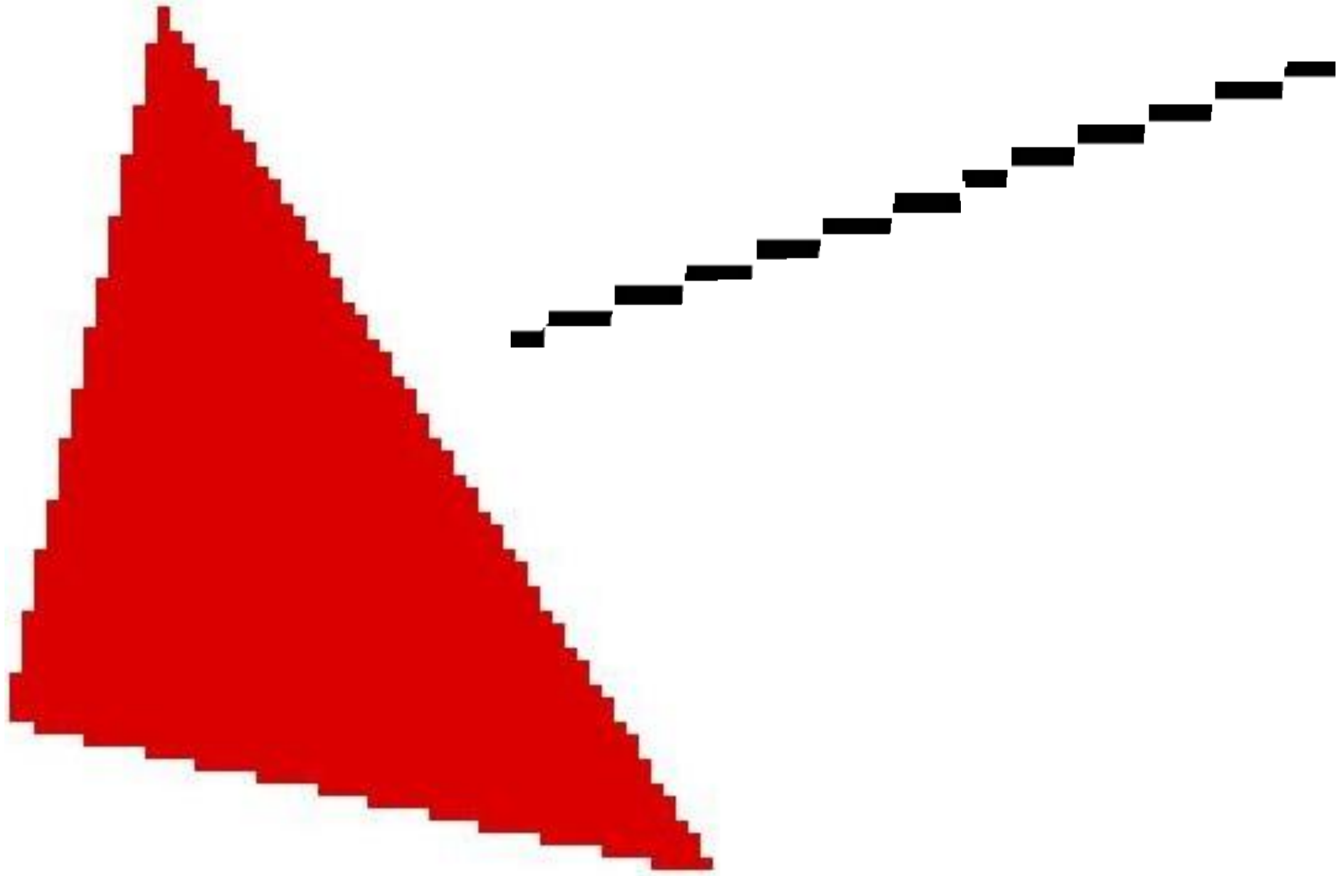
Undersampling a high-frequency signal can result in aliasing



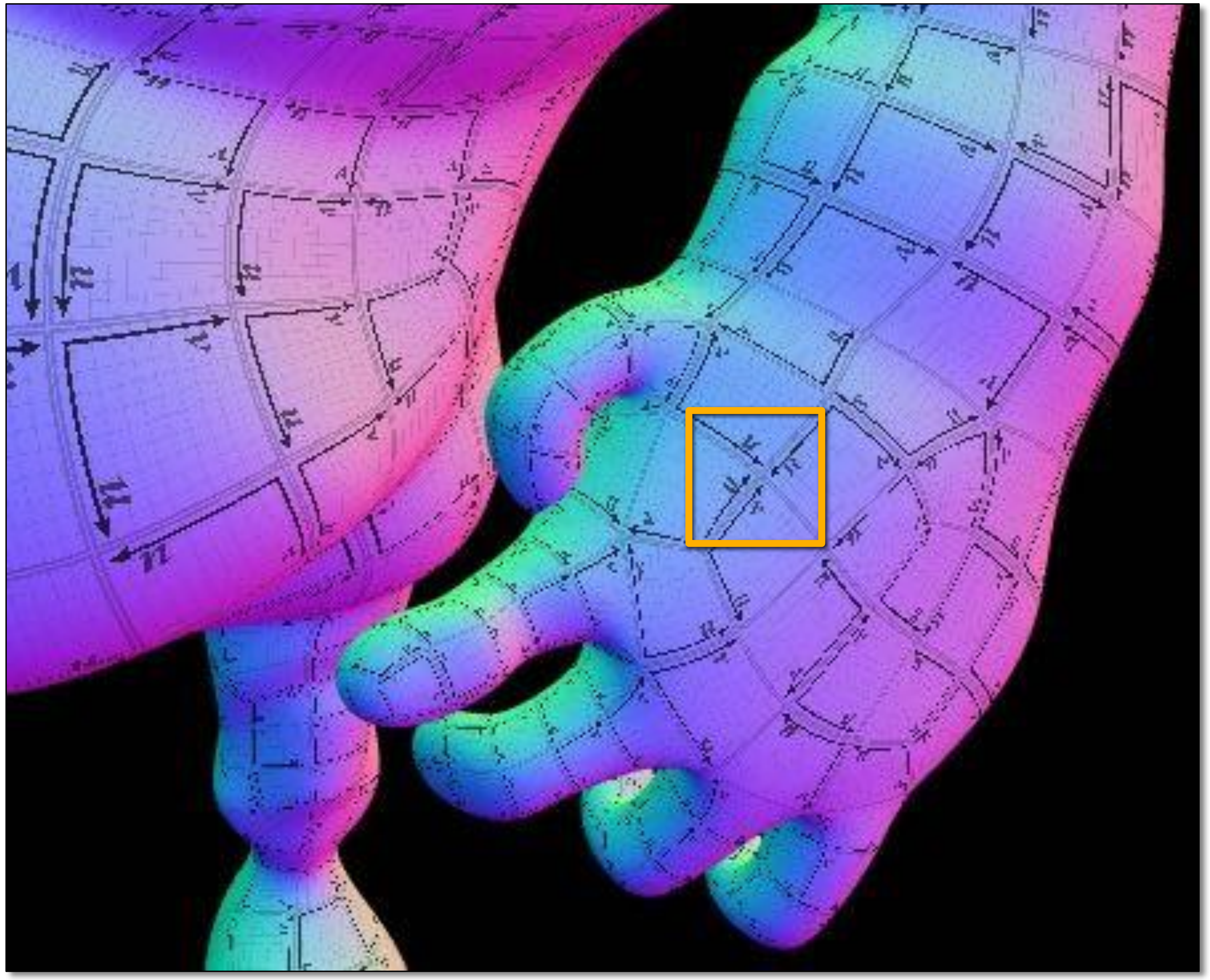
1D example

2D examples:
Moiré patterns, jaggies

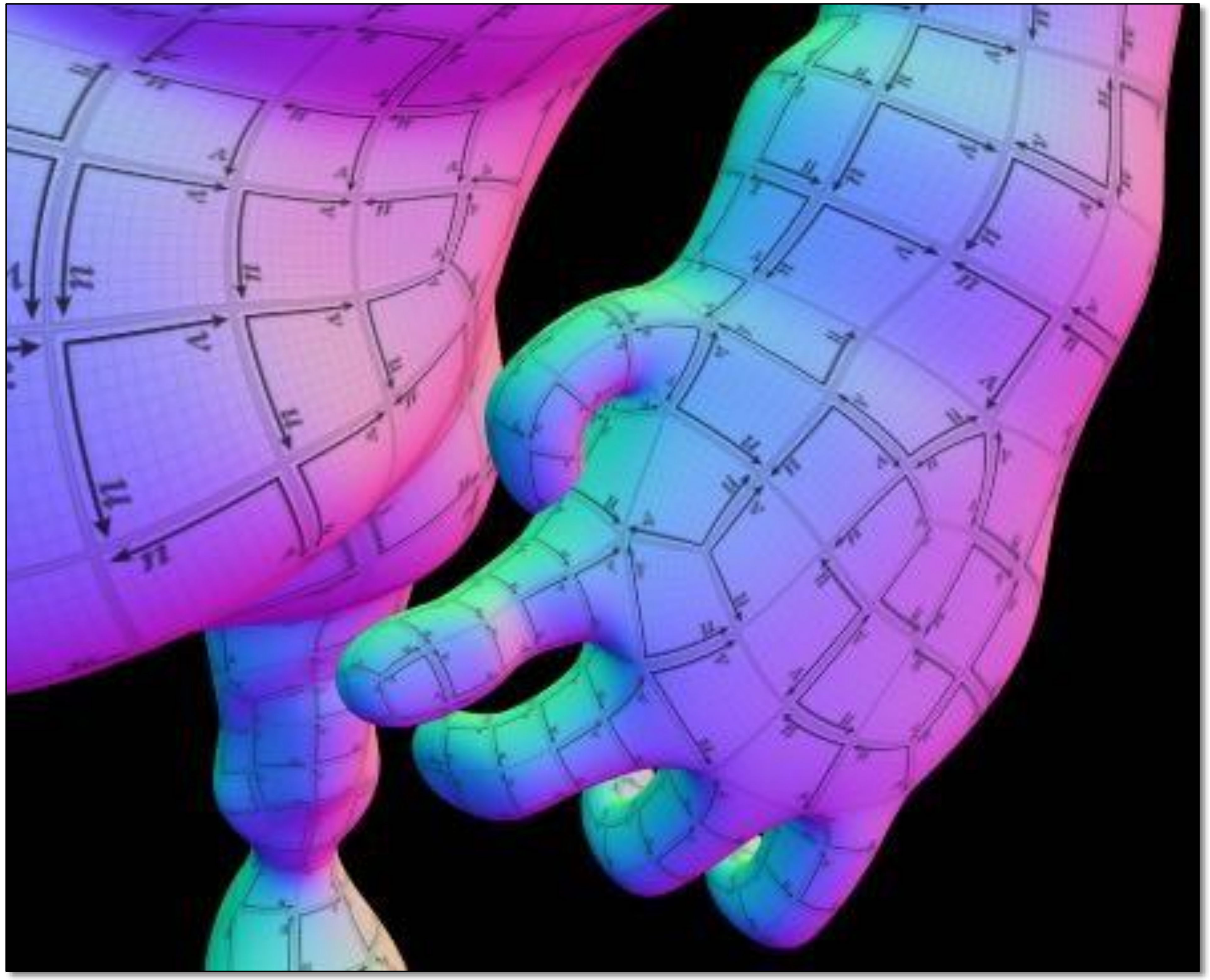




Aliasing due to undersampling texture

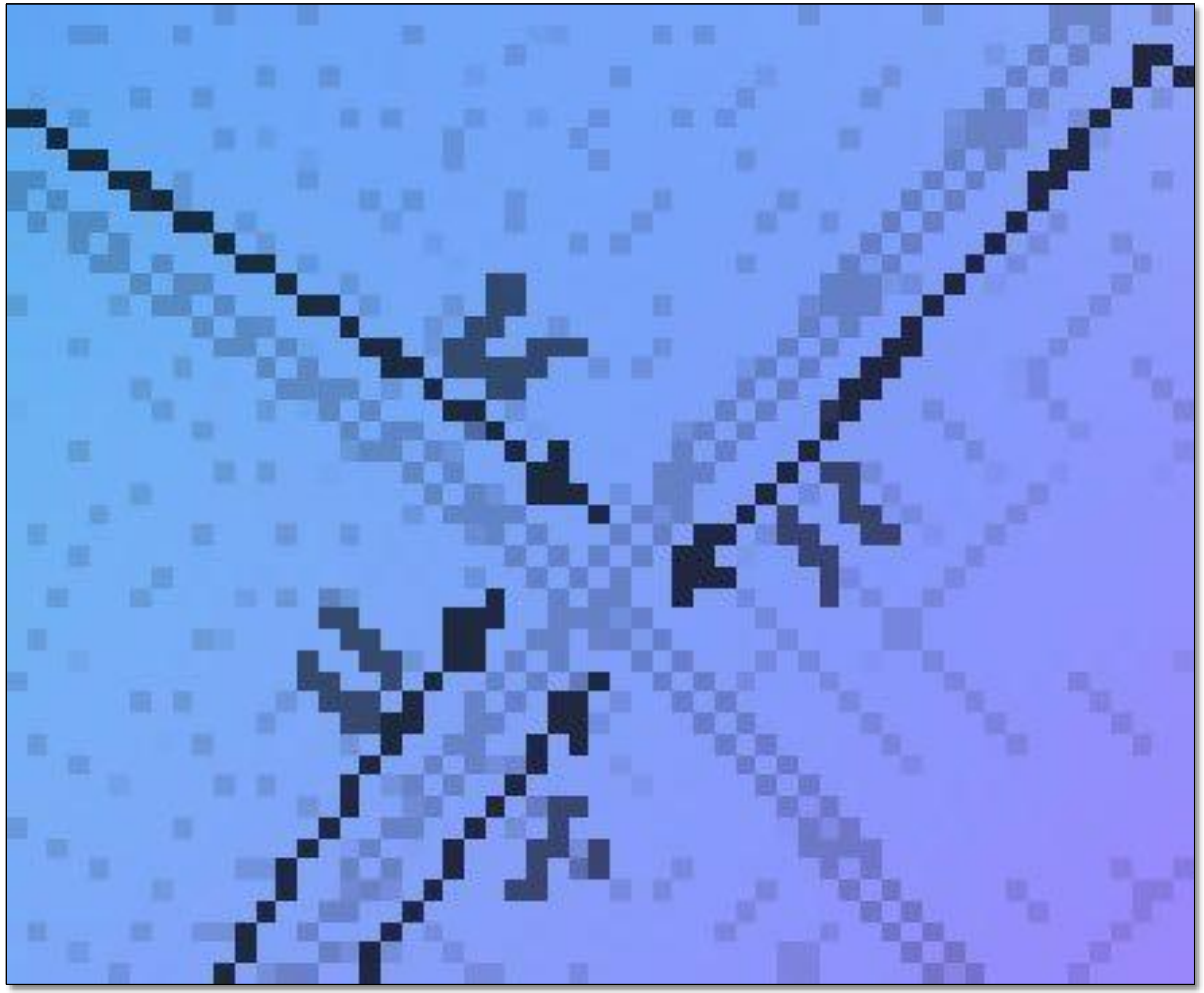


One texture sample per pixel
(aliasing!)

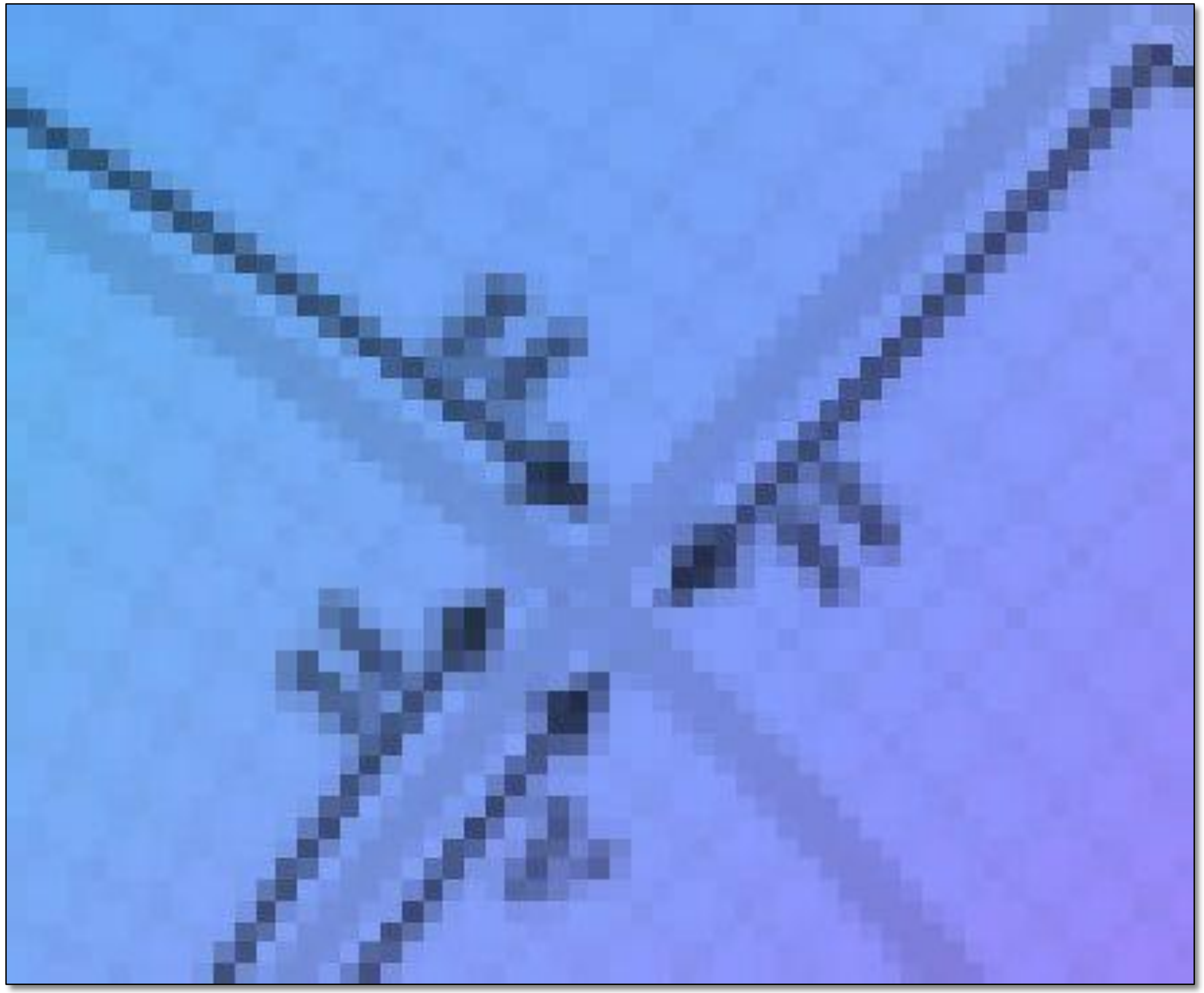


Anti-aliased texture sampling

Aliasing due to undersampling (zoom)

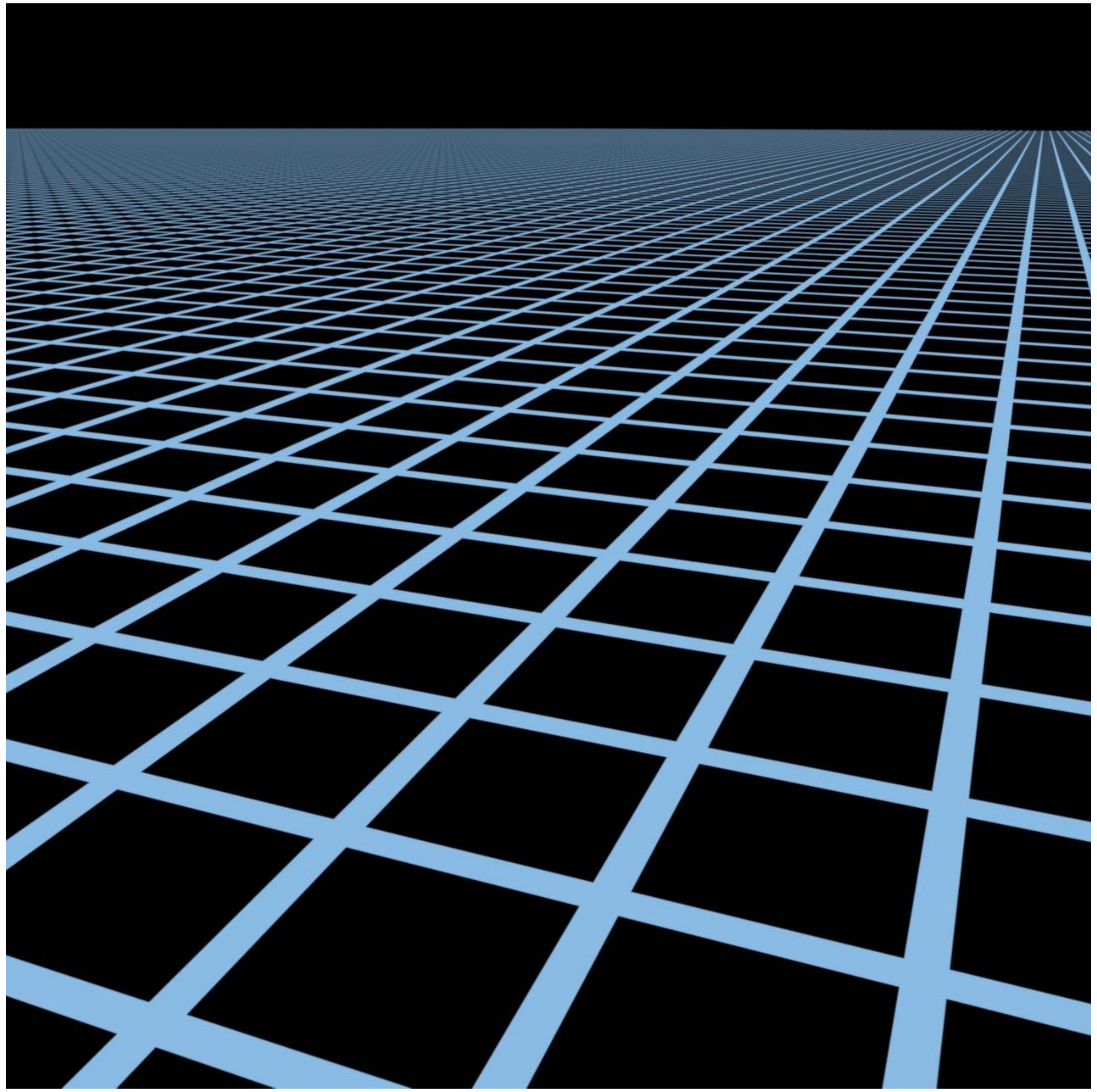


One texture sample per pixel
(aliasing!)

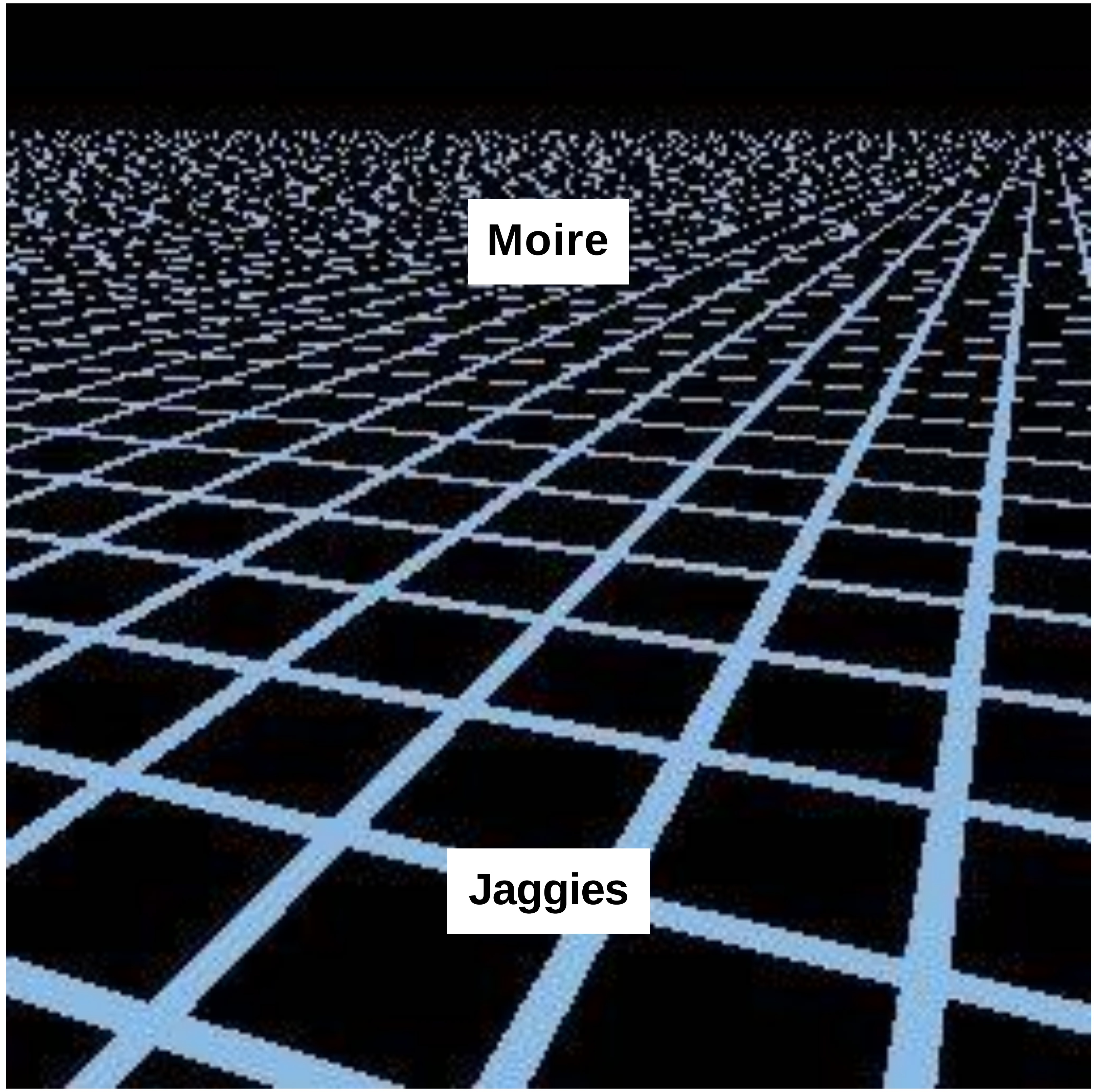


Anti-aliased texture sampling

Another example



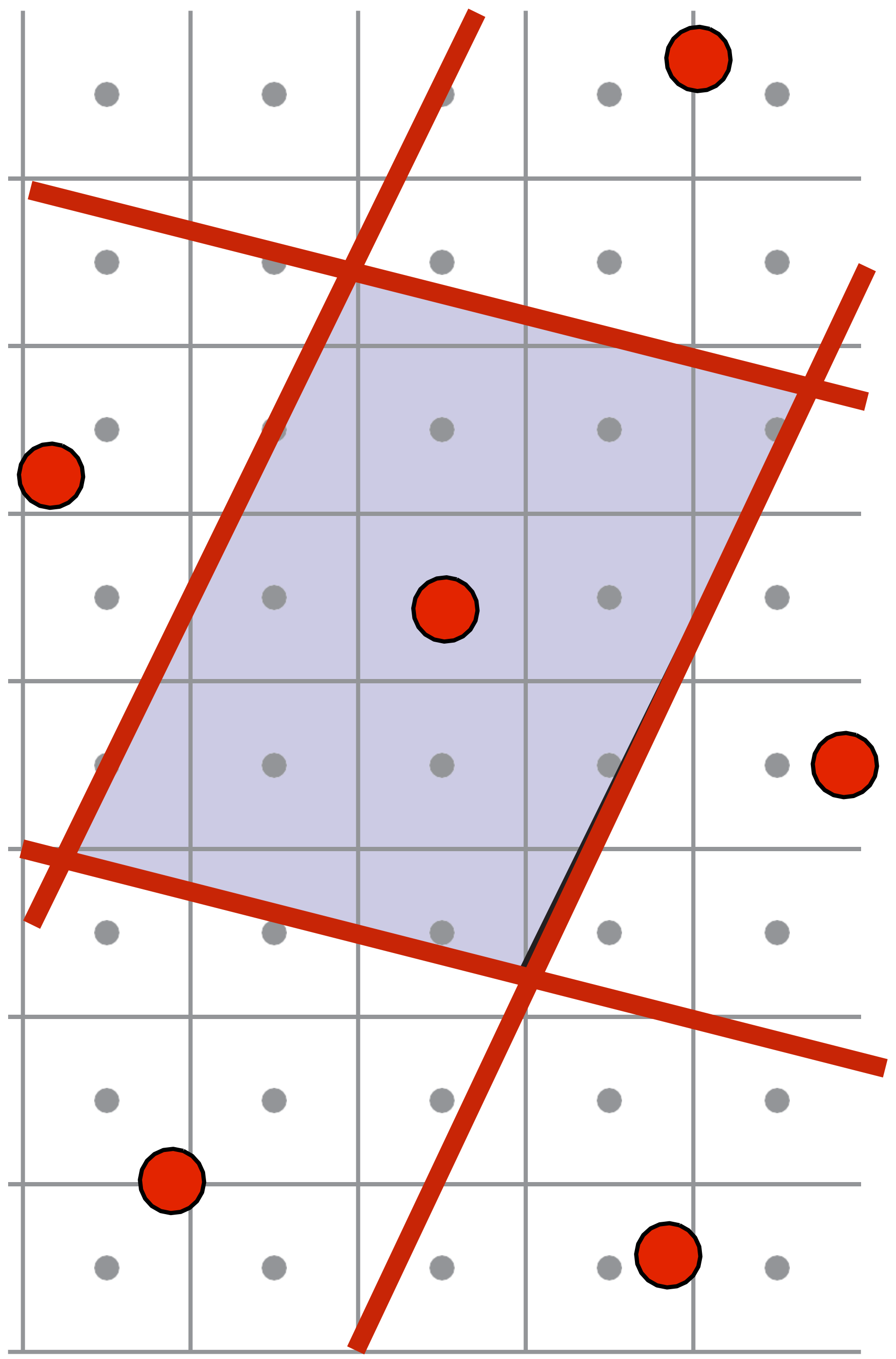
Anti-aliased result



Rendered image: 256x256 pixels

Texture minification - hard case

- **Challenge:**
  - Many texels contribute to color of an output image pixel (sampling only one of them could yield aliasing)
  - Shape of pixel footprint can be complex



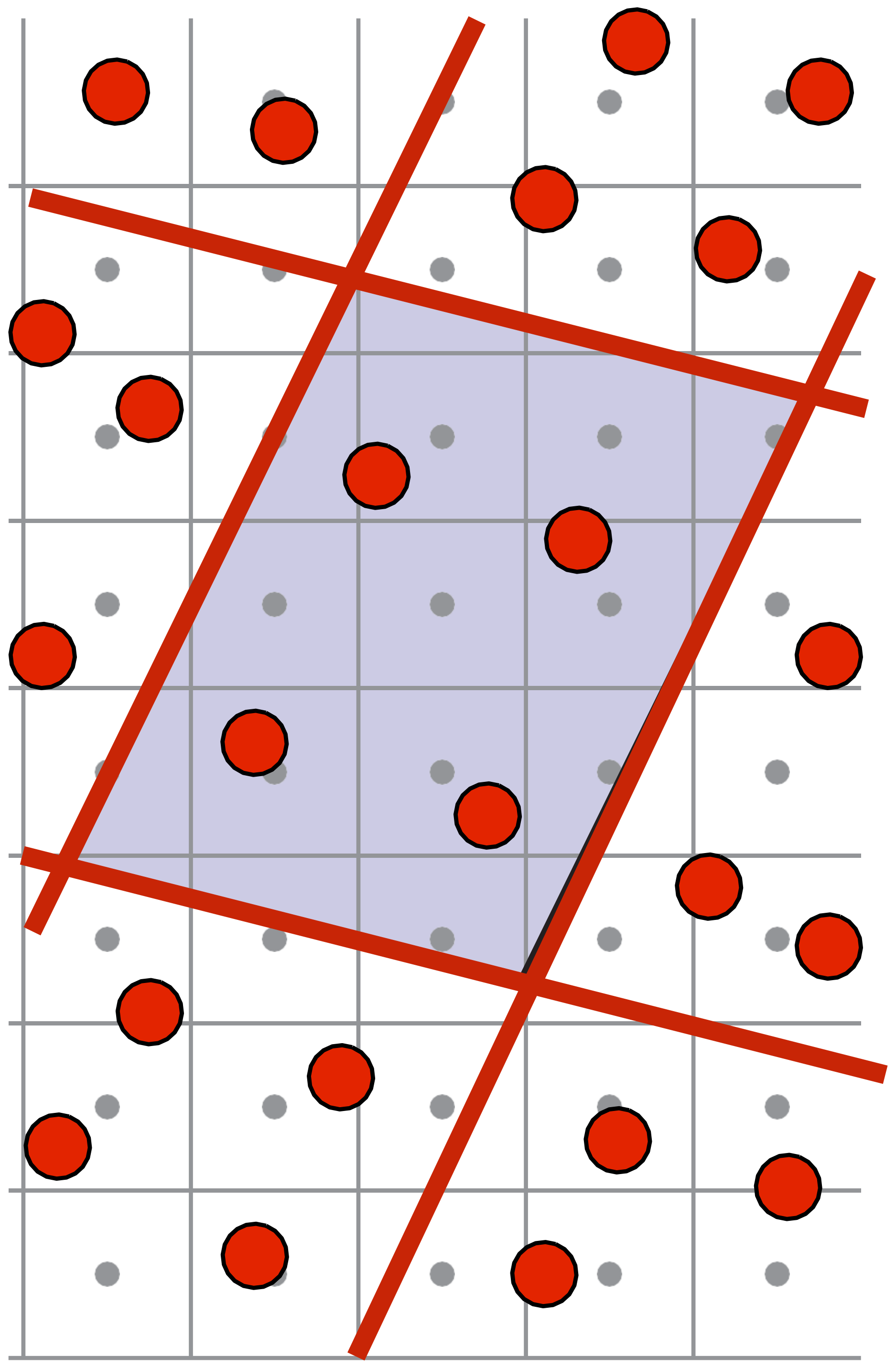
Shaded region = pixel area
Red lines = screen pixel boundaries
Red dots = texture space sample points for adjacent pixels

Texture minification - hard case

- Challenge:
  - Many texels contribute to color of an output image pixel (sampling only 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?

Alternative solution: remove high frequency from texture to reduce aliasing!

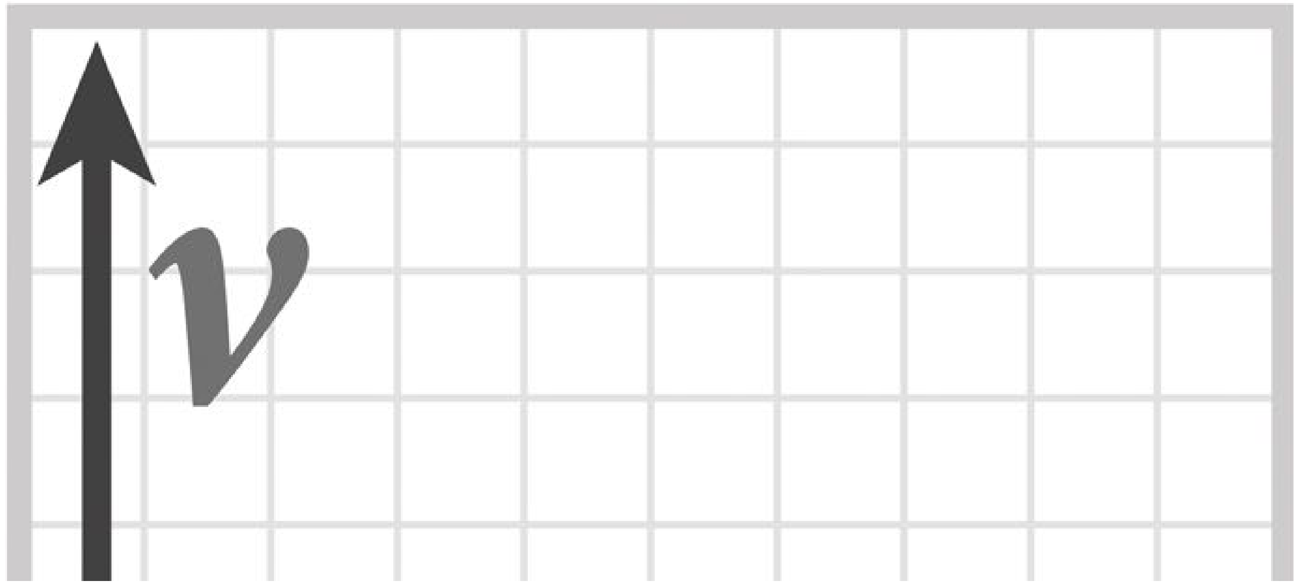


Shaded region = pixel area
Red lines = screen pixel boundaries
Red dots = texture space sample points for adjacent pixels

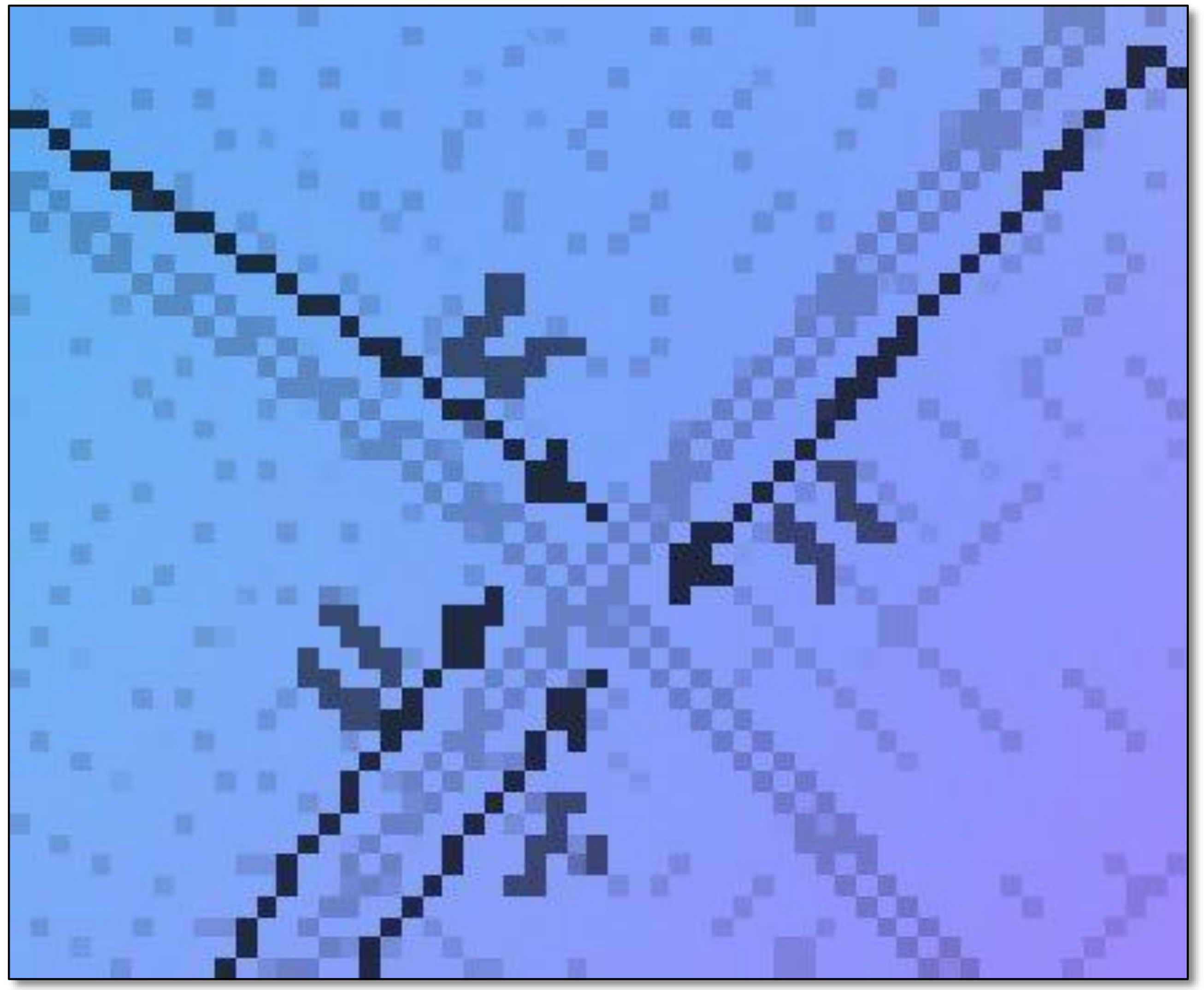
Pre-filtering texture map reduces aliasing

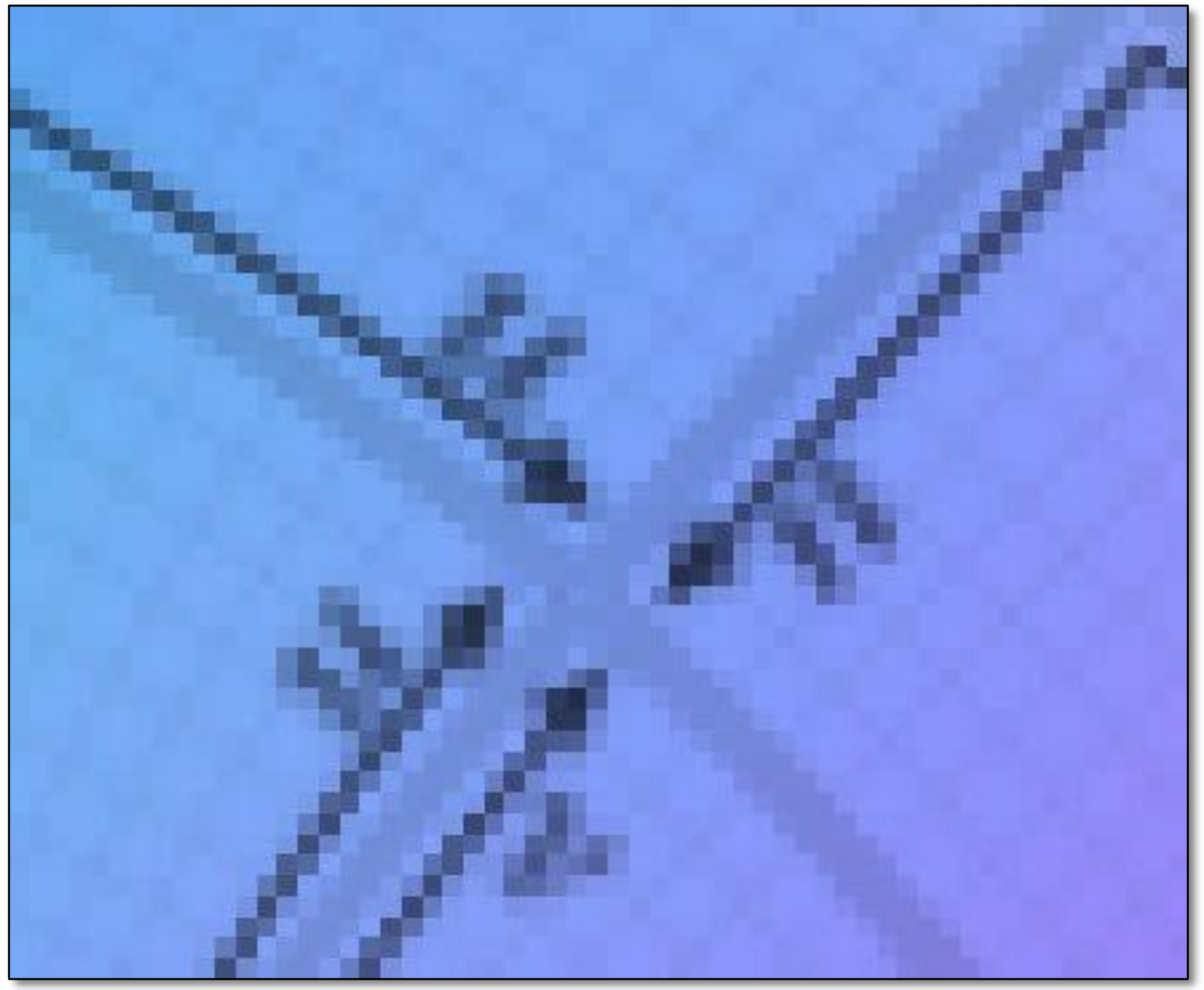
One texture sample per pixel
(aliasing!)

Pre-filtered texture map
(high frequencies removed)



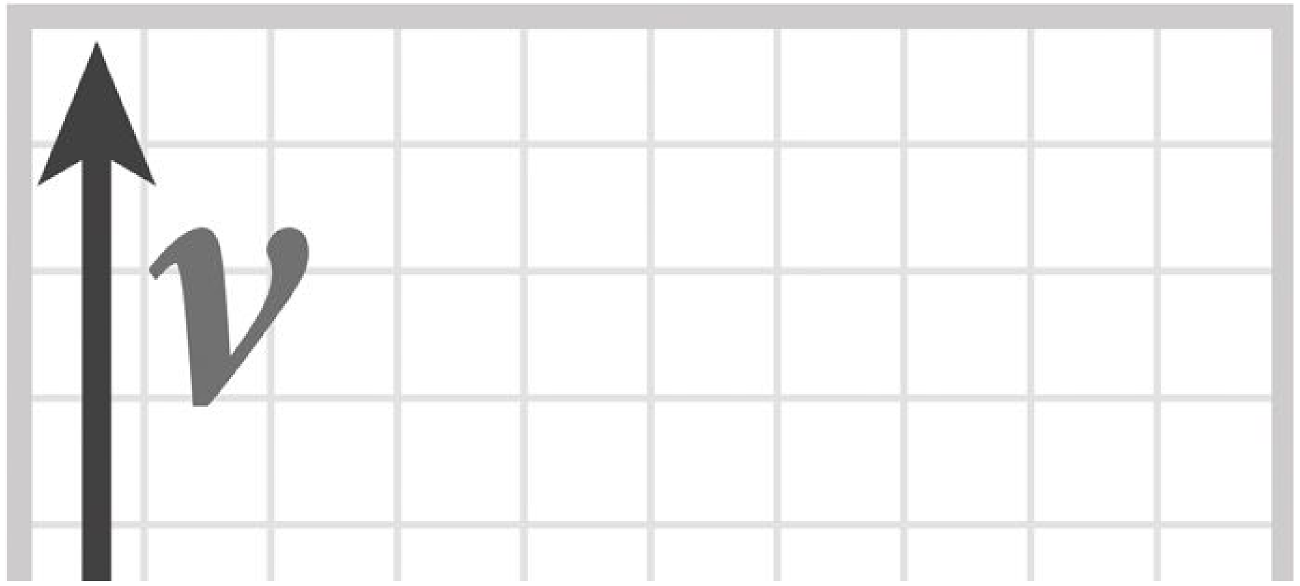
Pre-filtering texture map reduces aliasing





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

Pre-filtered texture map
(high frequencies removed)

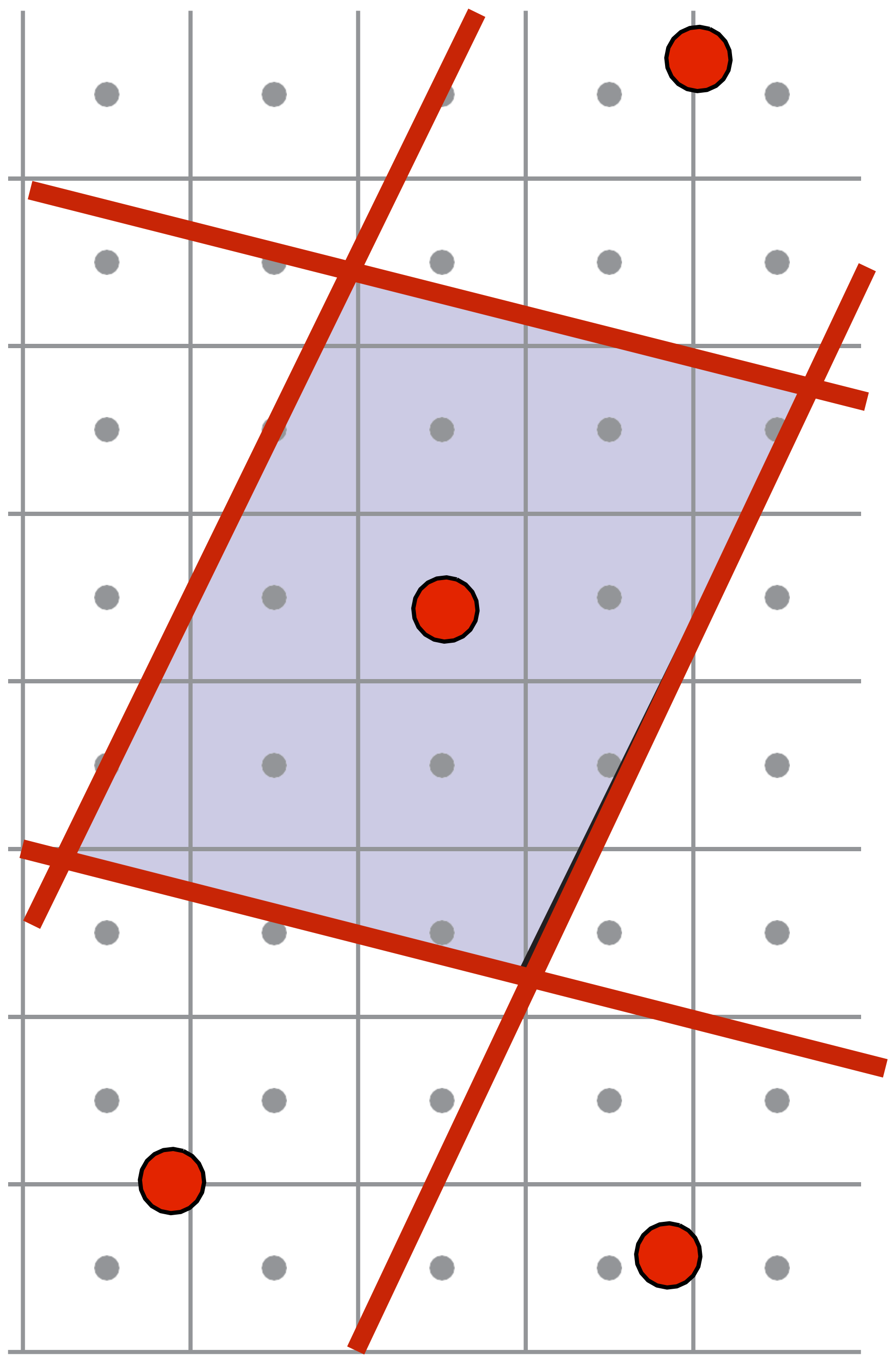


But how much should we pre-filter?

- Amount of pre-filtering depends on how far away the object is:

  - minor minification: image pixel extreme
    magnification: image pixel spans large region of texture

- 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

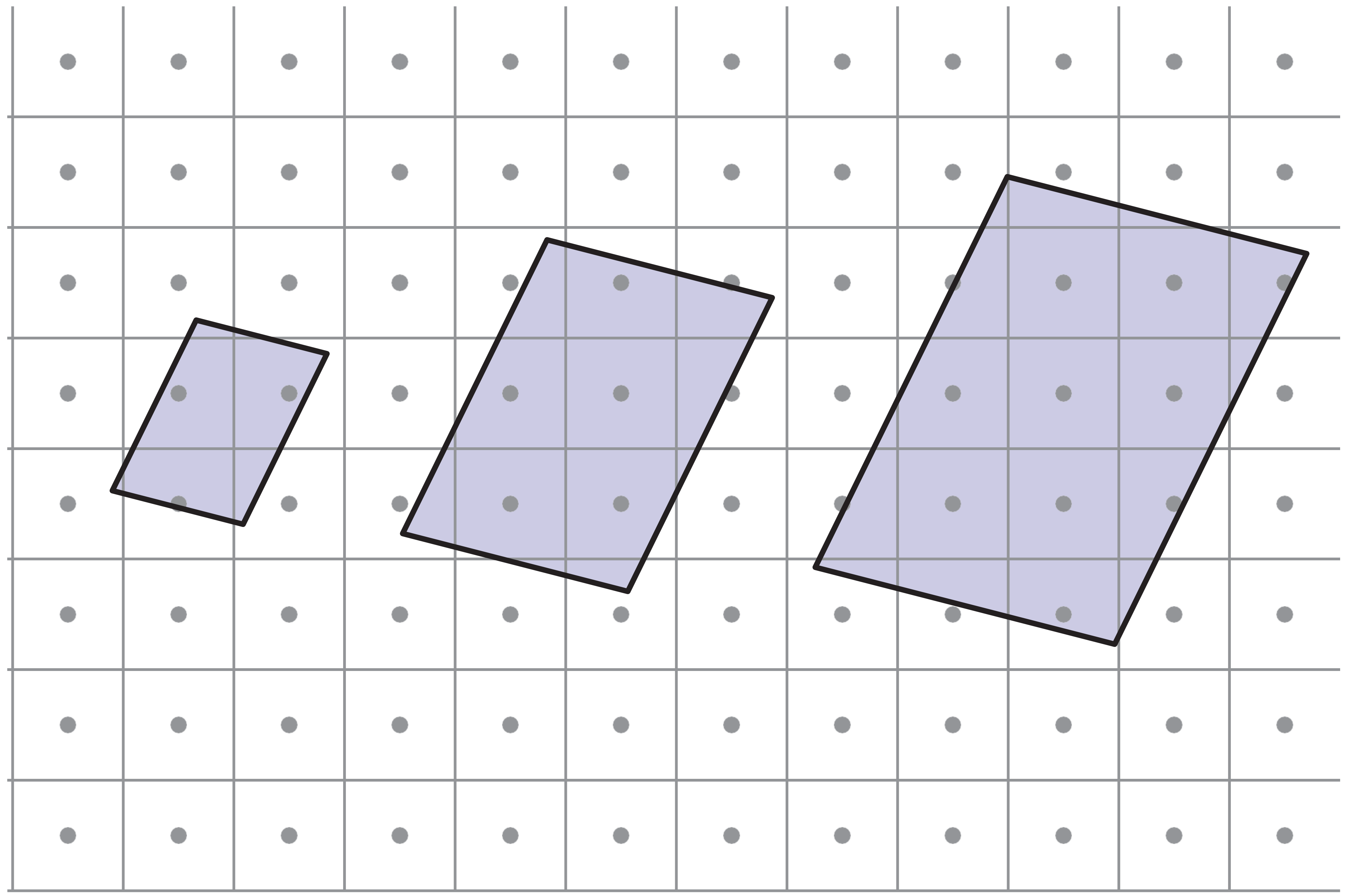


Shader region = pixel area
Red lines = screen pixel boundaries
Red dots = texture space sample points for adjacent pixels

But how much should we pre-filter?

- Amount of pre-filtering necessary depends on how far away the object is

- Idea: pre-compute and store different versions of the texture with different amounts of prefiltering
  - Low-pass filter and downsample texture file, and store successively lower resolutions
  - When sampling texture, use the texture file whose prefiltering amount matches the desired sampling rate



Mipmap (L. Williams 83)

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

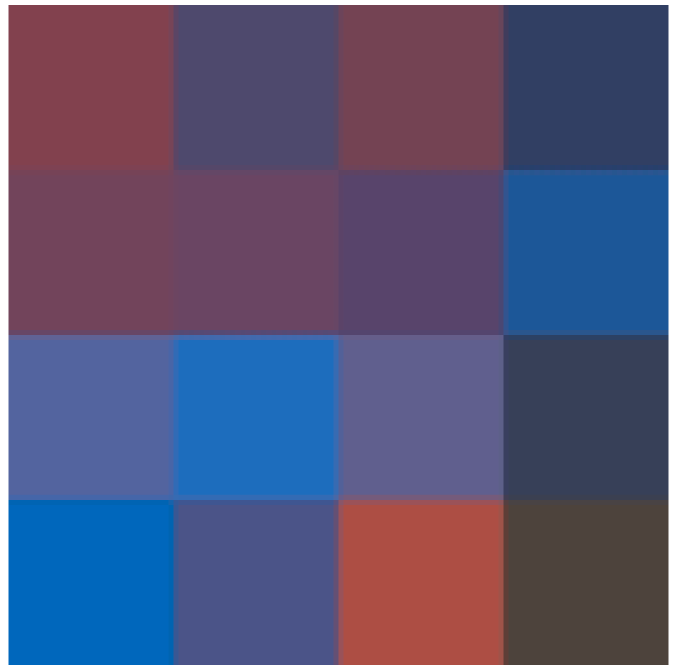

Level 0 = 128x128

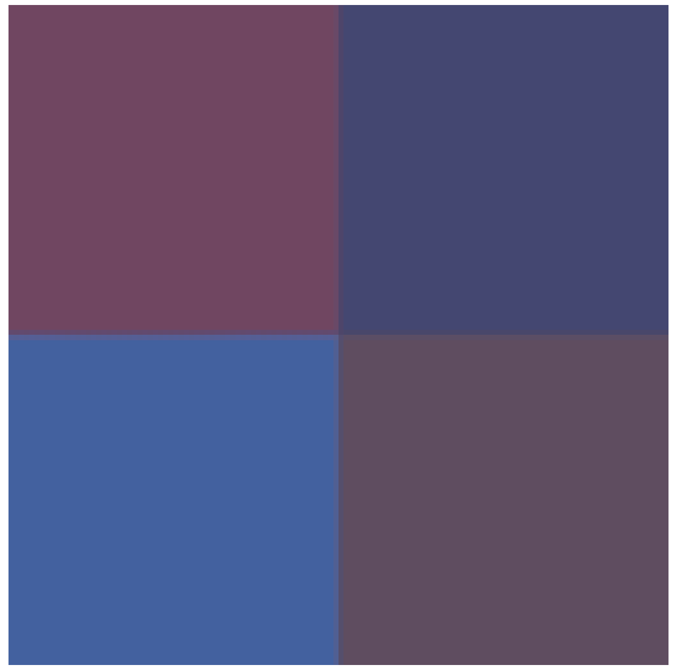

Level 1 = 64x64


Level 2 = 32x32


Level 3 = 16x16


Level 4 = 8x8

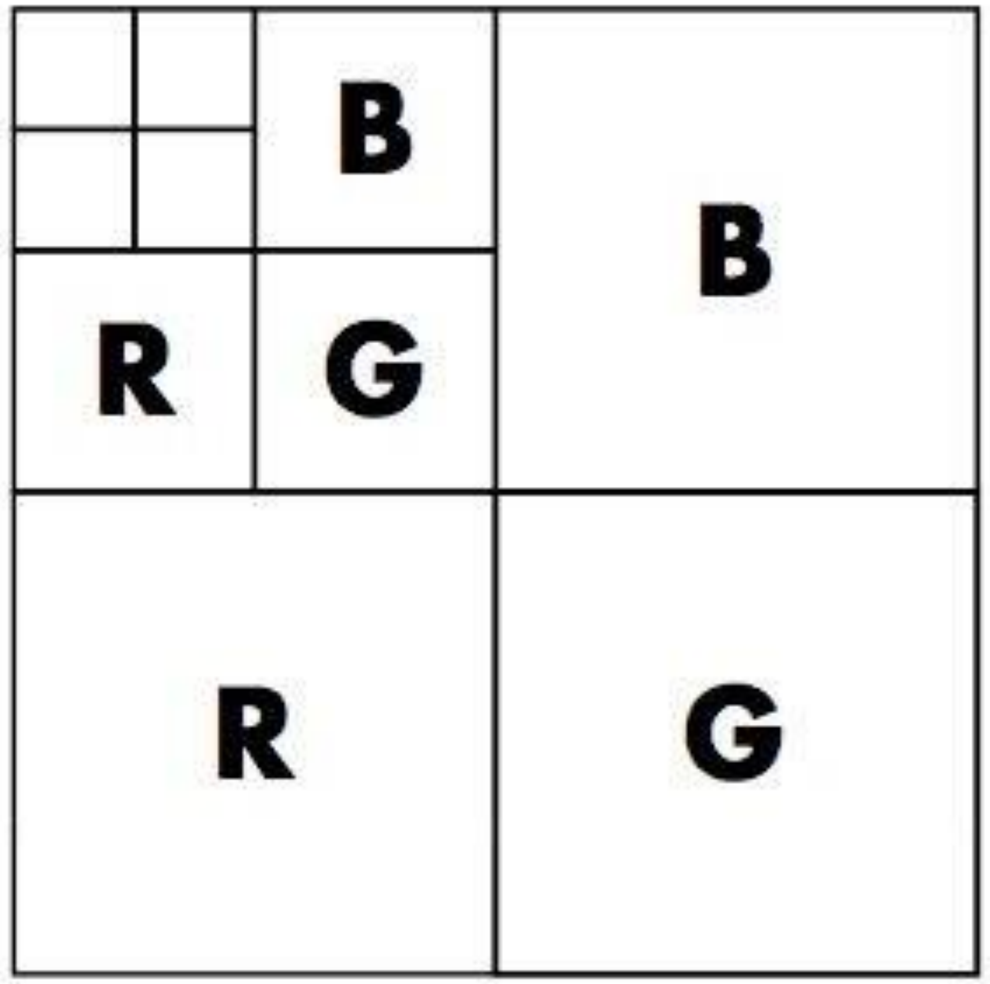

Level 5 = 4x4


Level 6 = 2x2

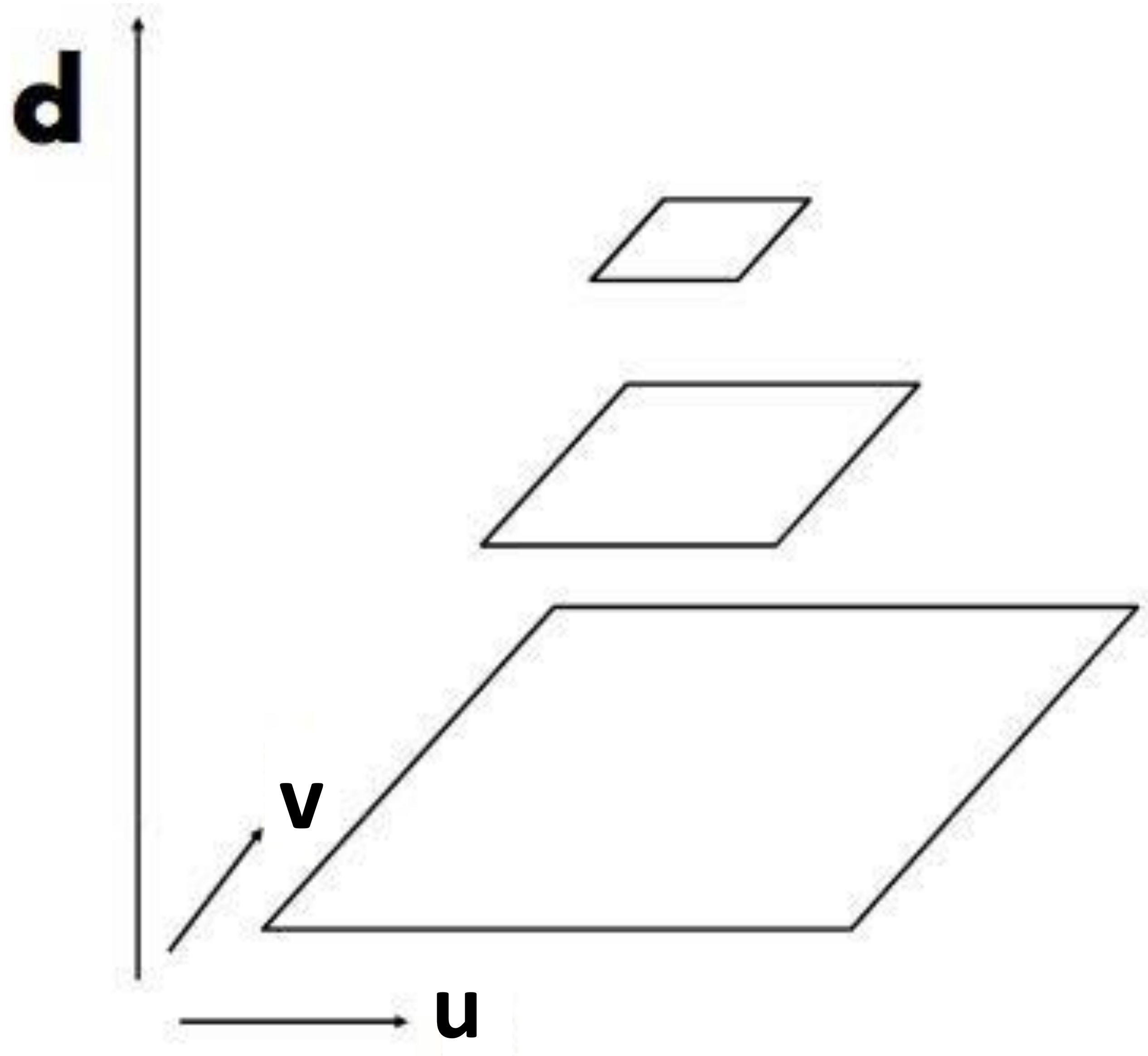

Level 7 = 1x1

"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



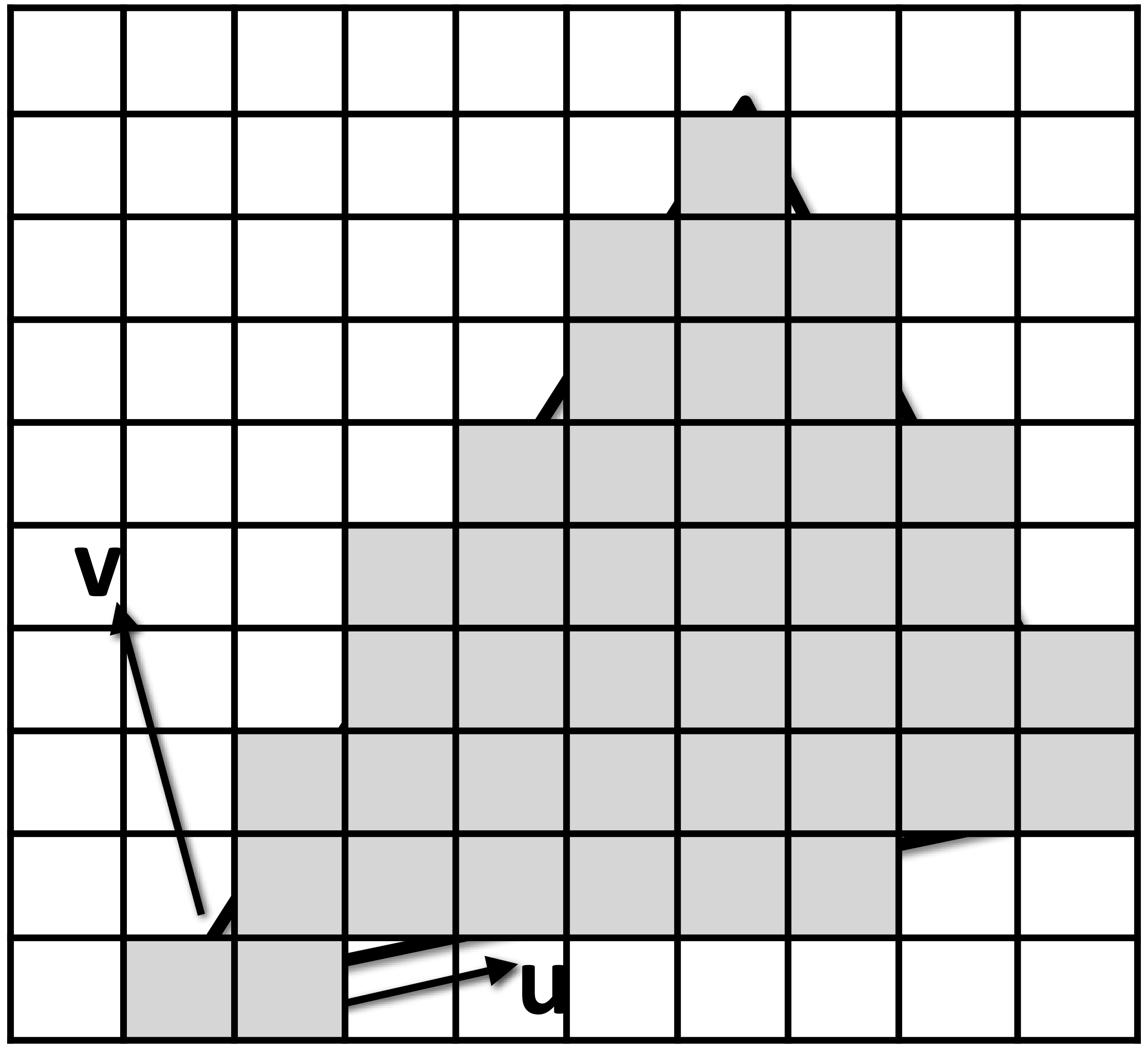
"Mip hierarchy"
level = *d*

What is the storage overhead of a mipmap?

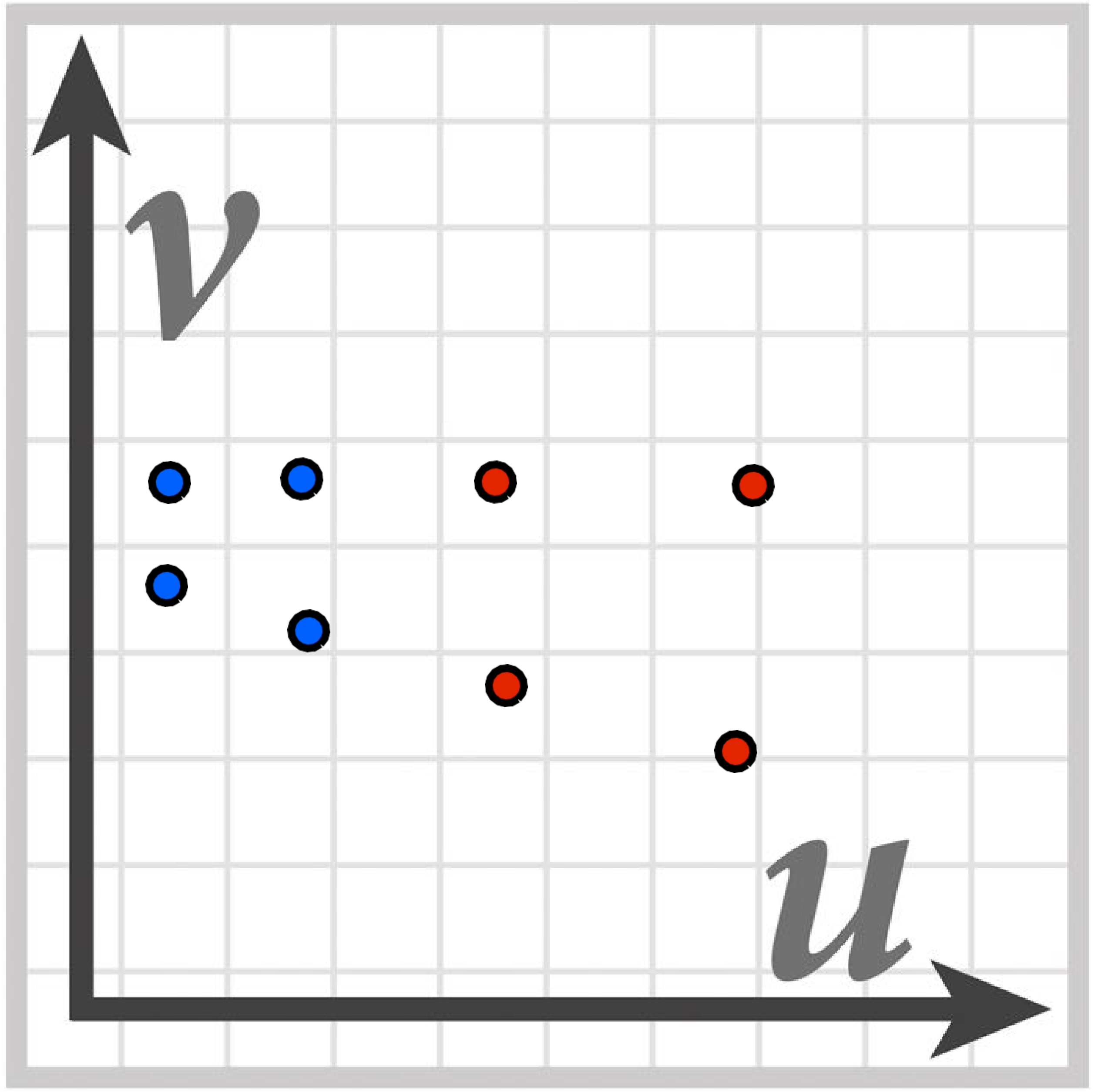
Slide credit: Akeley and Hanrahan

Computing mipmap level

Compute differences between texture coordinate values of neighboring screen samples



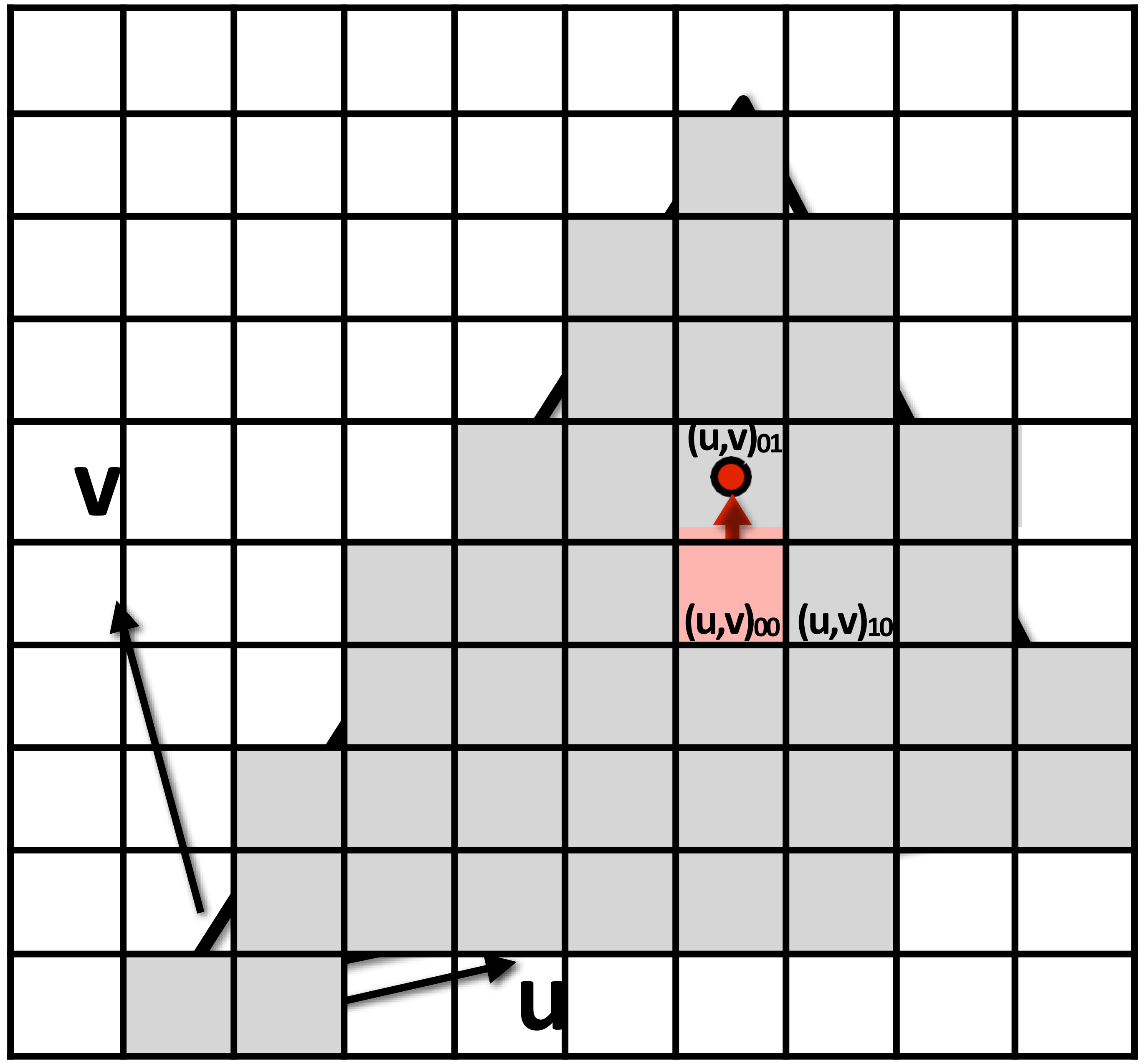
Screen space



Texture space

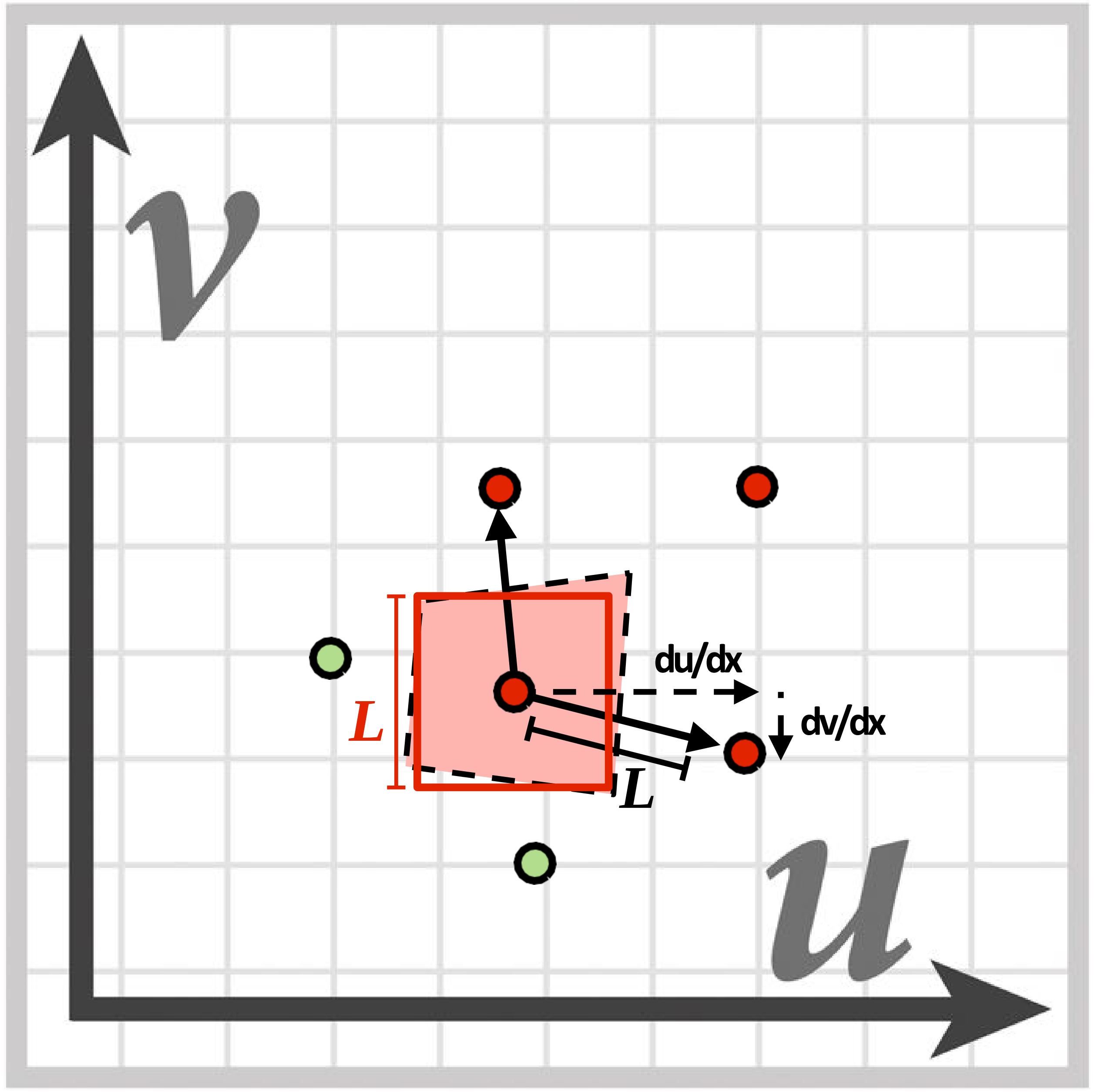
Computing mipmap level

Compute differences between texture coordinate values of neighboring screen samples



$du/dx = u_{10} - u_{00}$ $dv/dx = v_{10} - v_{00}$

$du/dy = u_{01} - u_{00}$ $dv/dy = v_{01} - v_{00}$



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

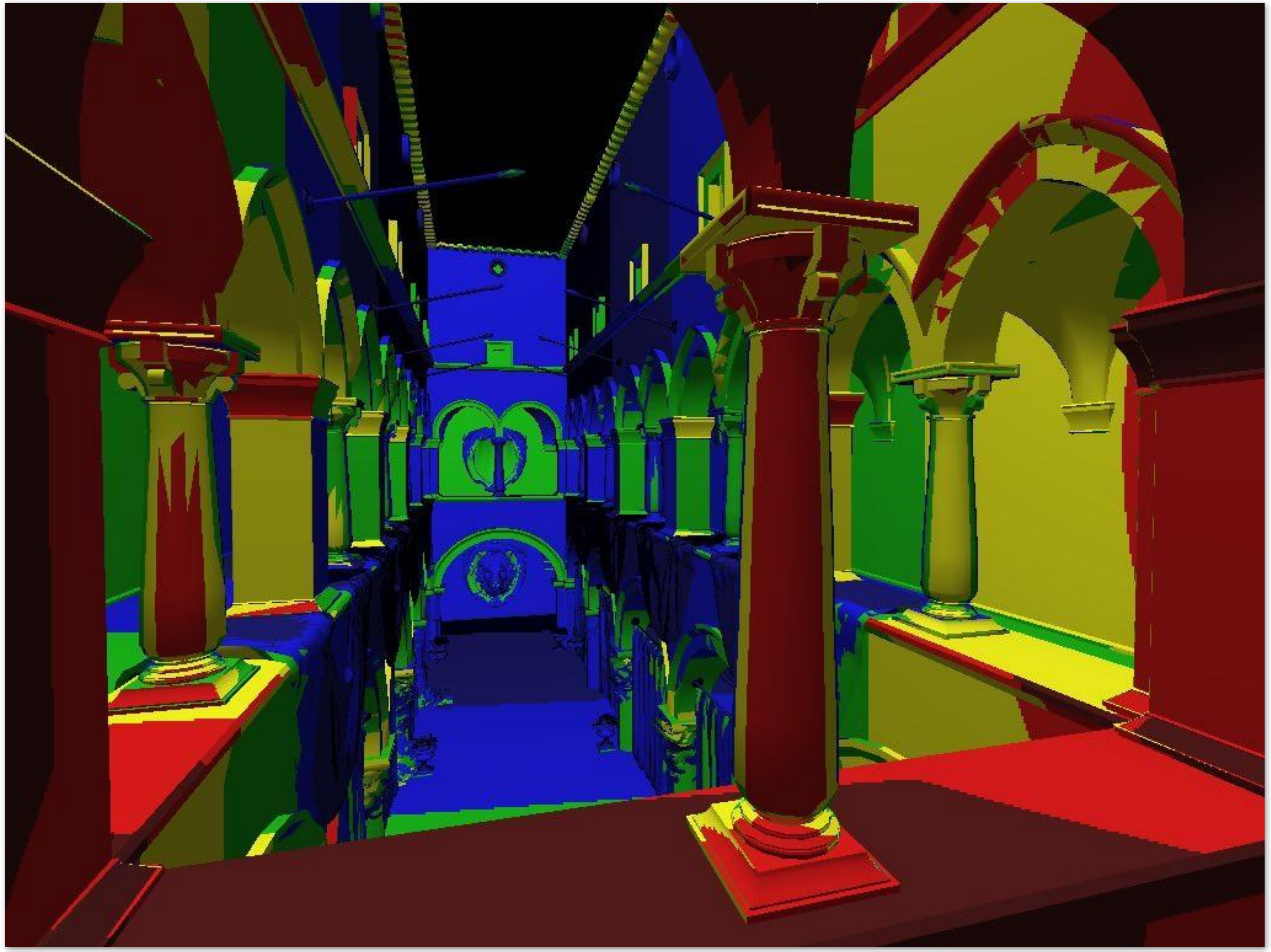
$$\textit{mip-map } d = \log_2 L$$

Bilinear resampling at level 0

Bilinear resampling at level 2

Bilinear resampling at level 4

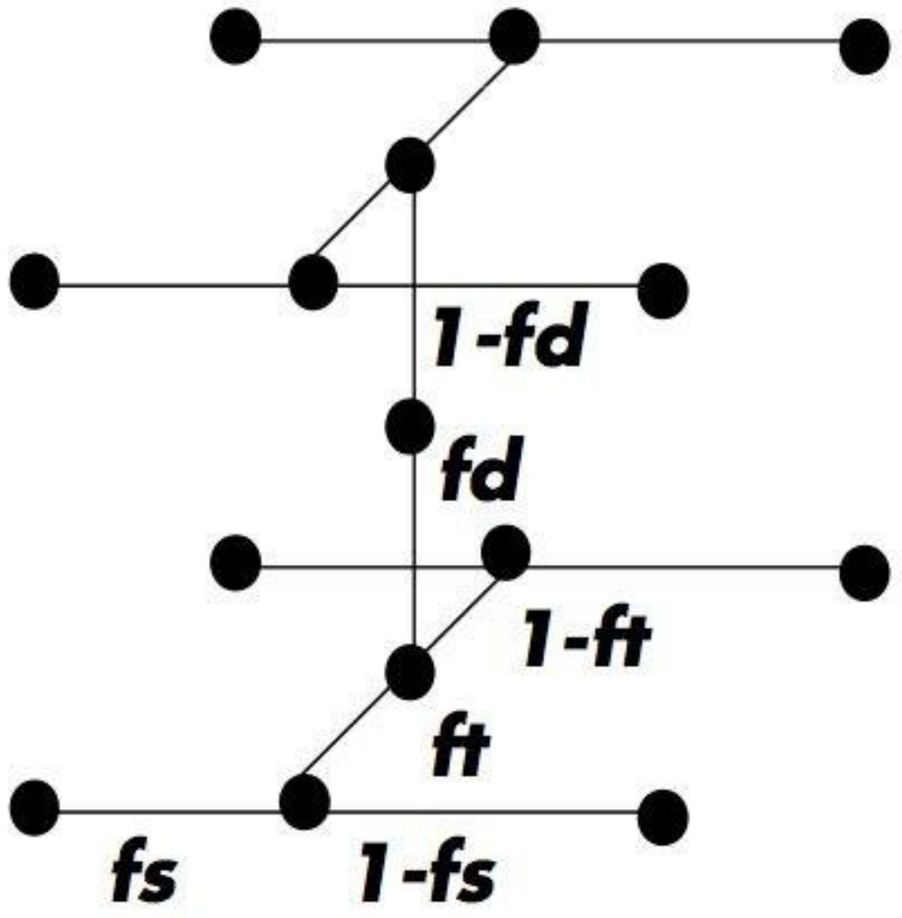
Visualization of mipmap level
(bilinear filtering only: *d* clamped to nearest level)



"Tri-linear" filtering

Linearly interpolate the bilinear interpolation results from two adjacent levels of the mip map.

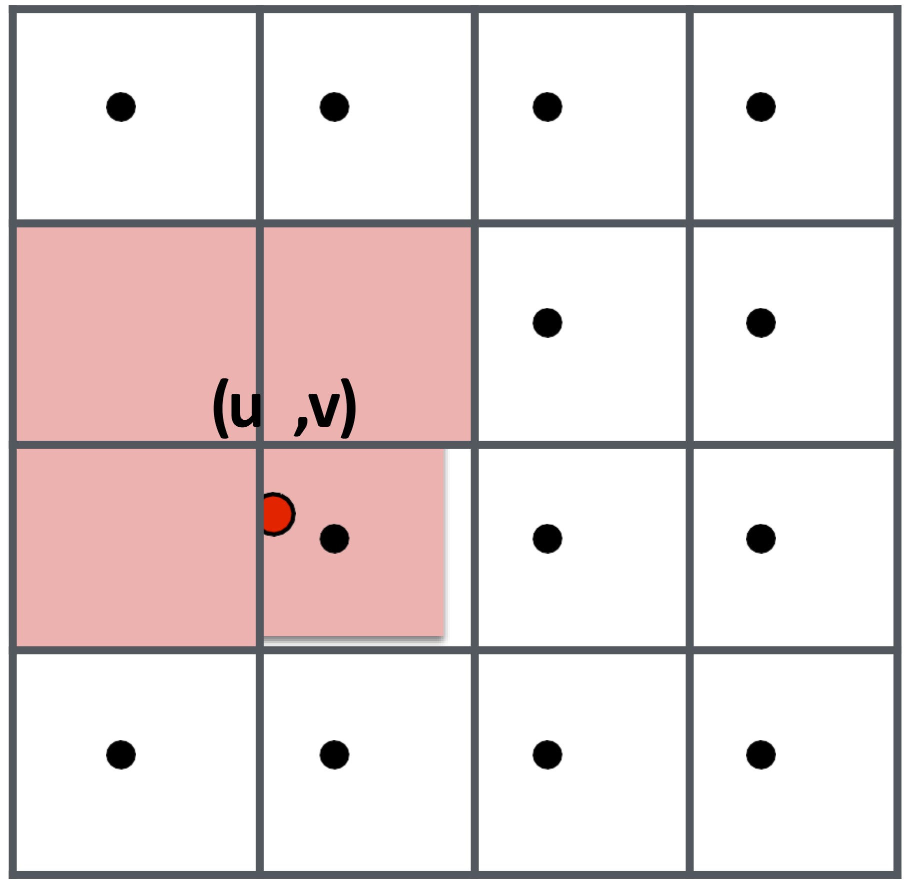
(smoothly transition between different levels of prefiltering)



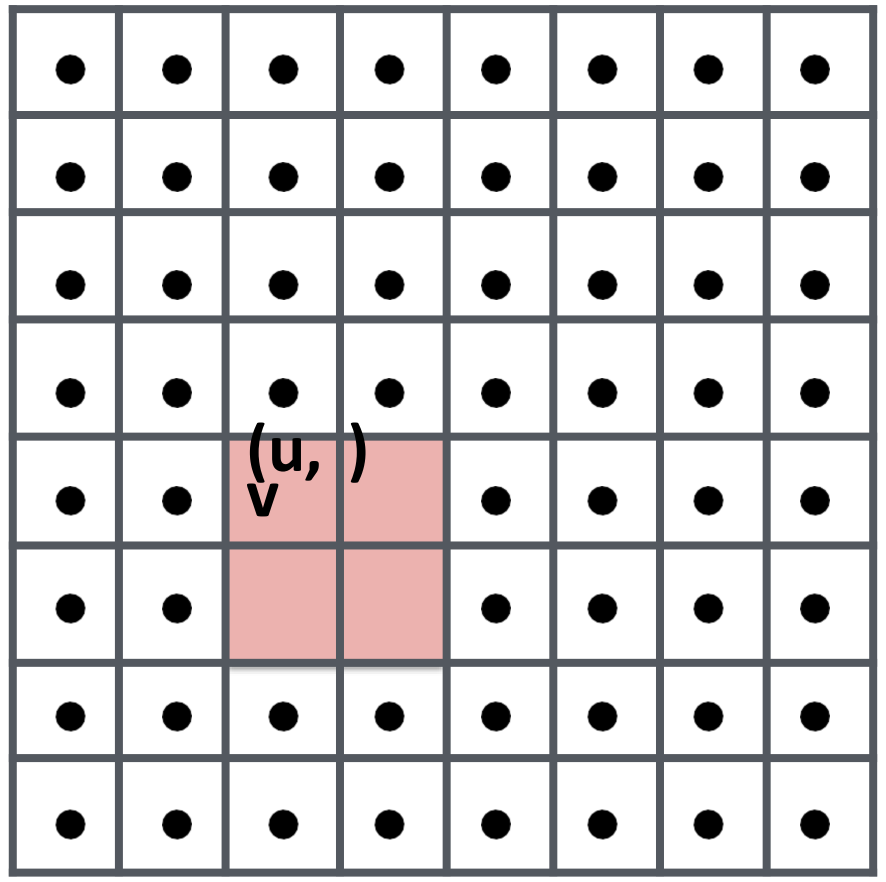
$$lerp(t, v_1, v_2) = v_1 + t(v_2 - v_1)$$

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

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



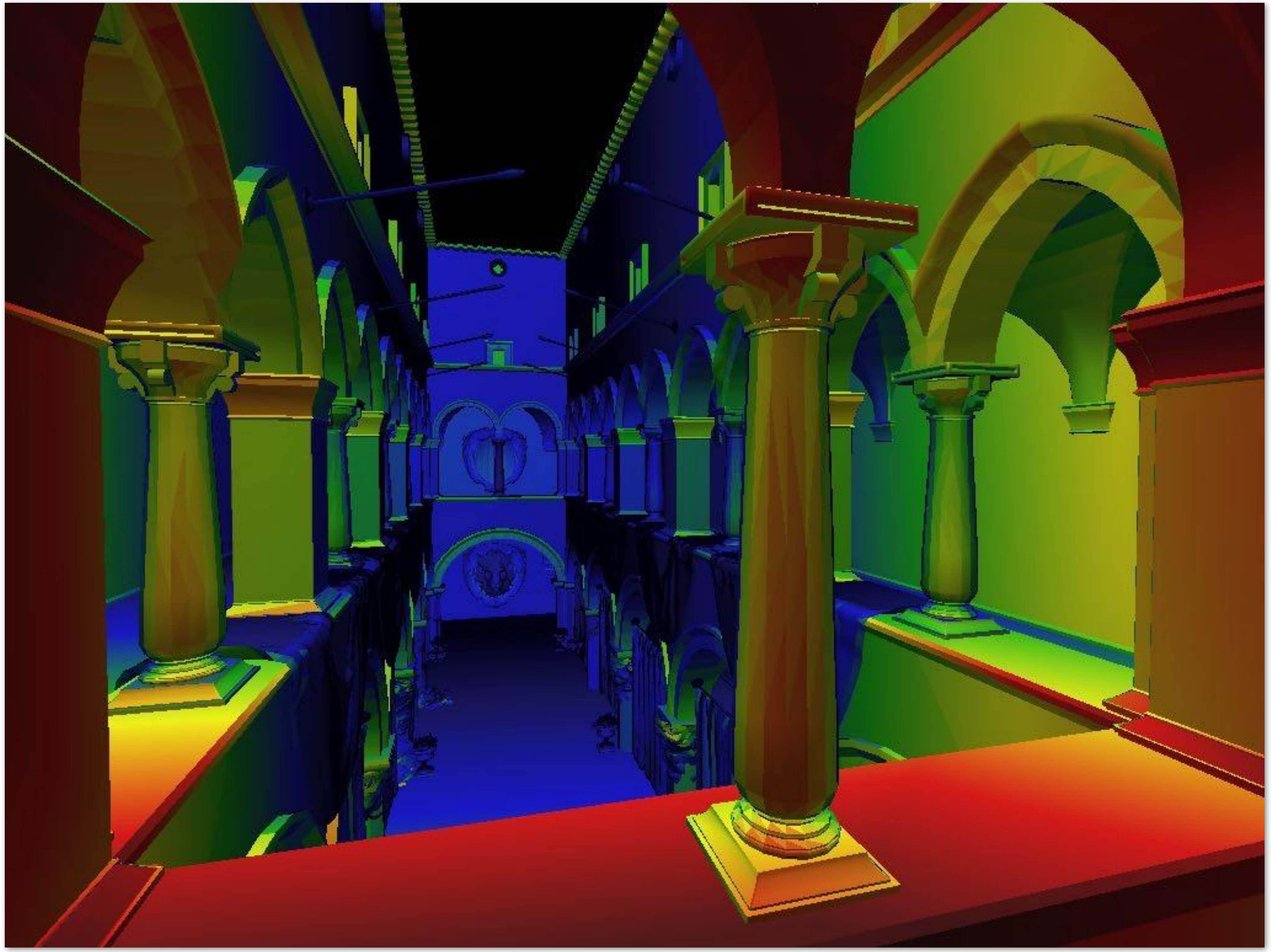
mip-map texels: level d+1



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)

Afull 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.