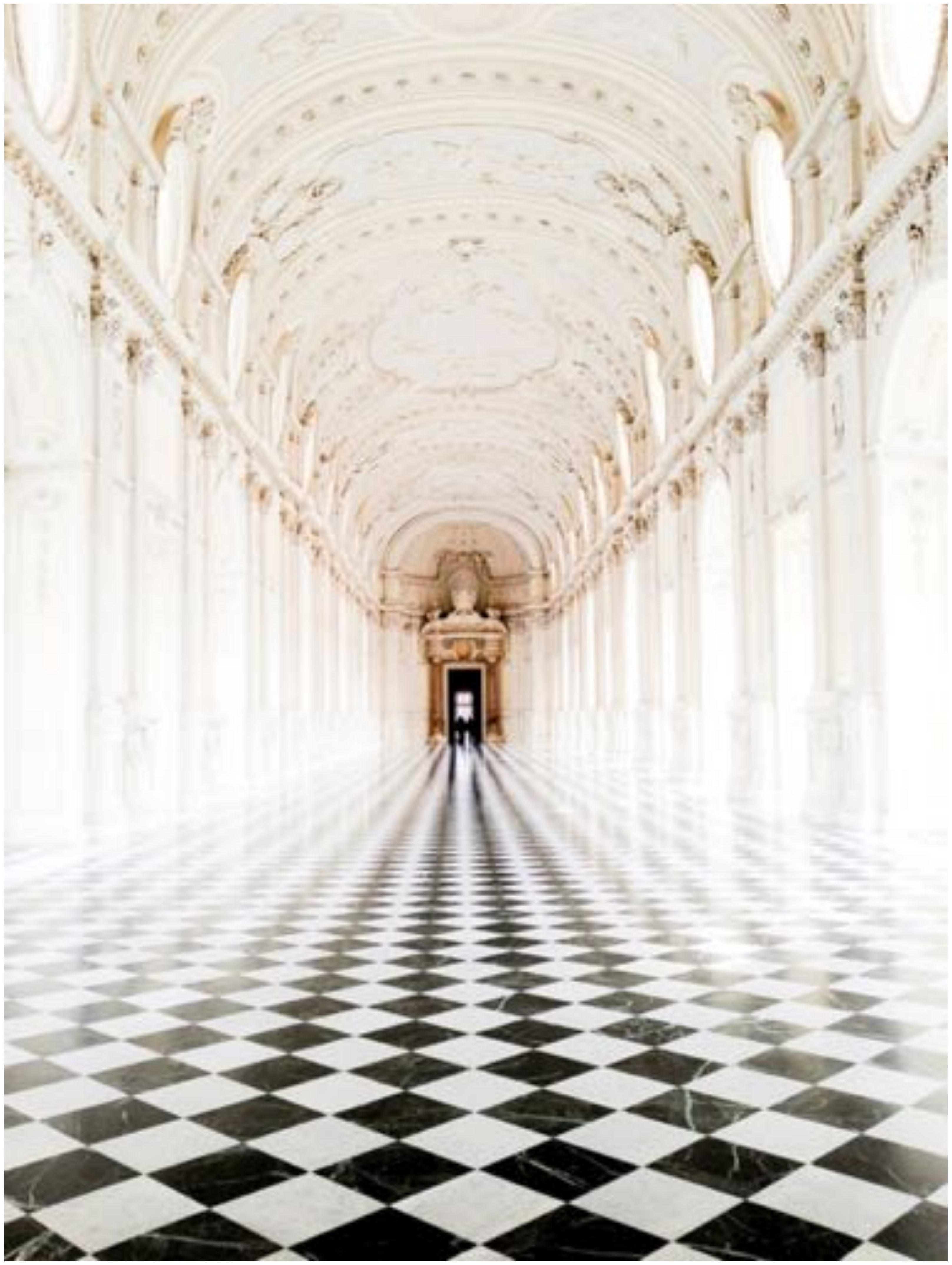
Perspective Projection and Texture Mapping

Computer Graphics
CMU 15-462/15-662

Note: There is a lot of material in these slides. We will likely not work through all of this in one class, but what we don't get to, we'll pick up next week, so let's see how it goes!

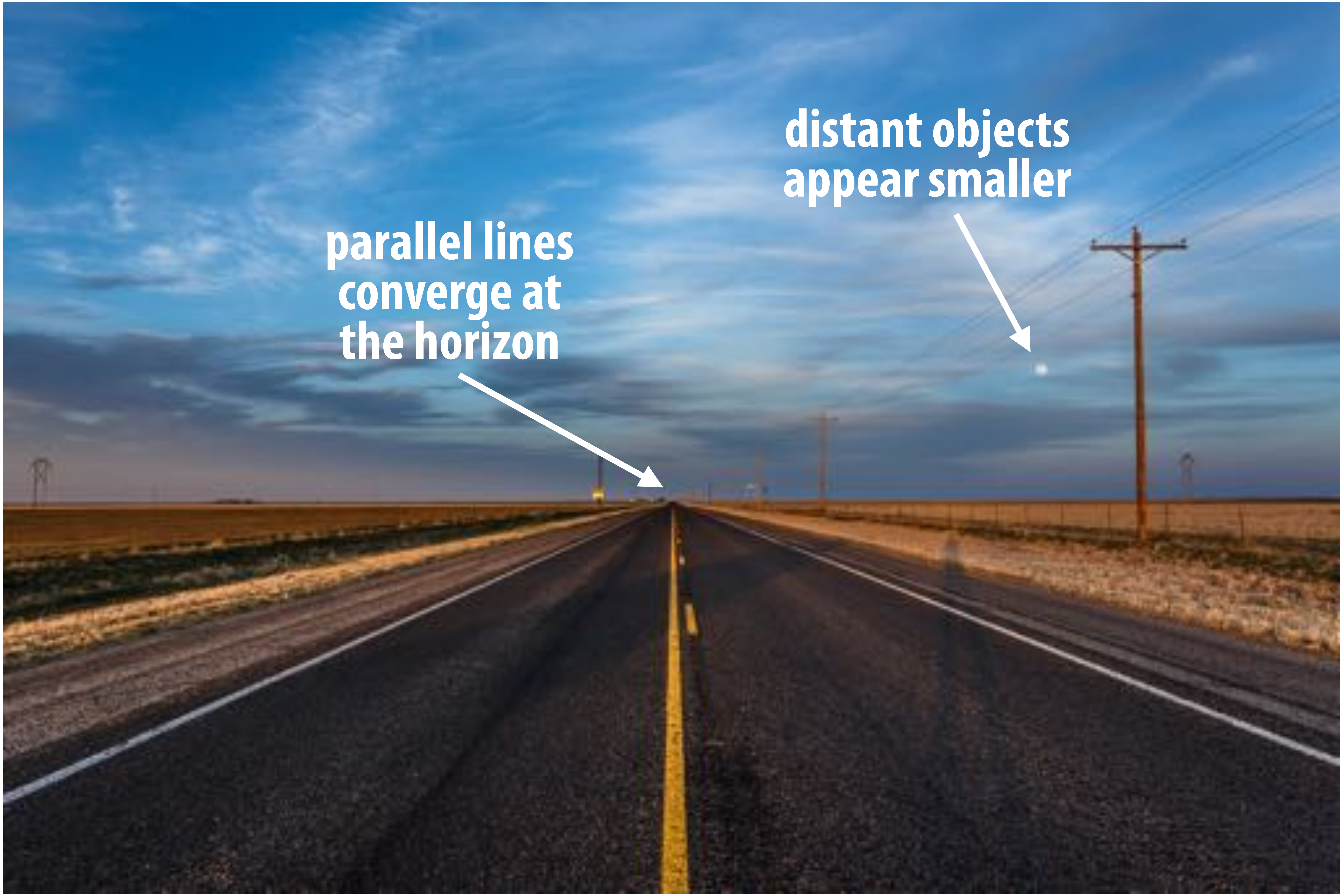
Perspective & Texture

- **PREVIOUSLY:**
  - **rasterization**
    (how to turn primitives into pixels)
  - **transformations**
    (how to manipulate primitives in space)
- **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!**



Perspective Projection

Perspective projection



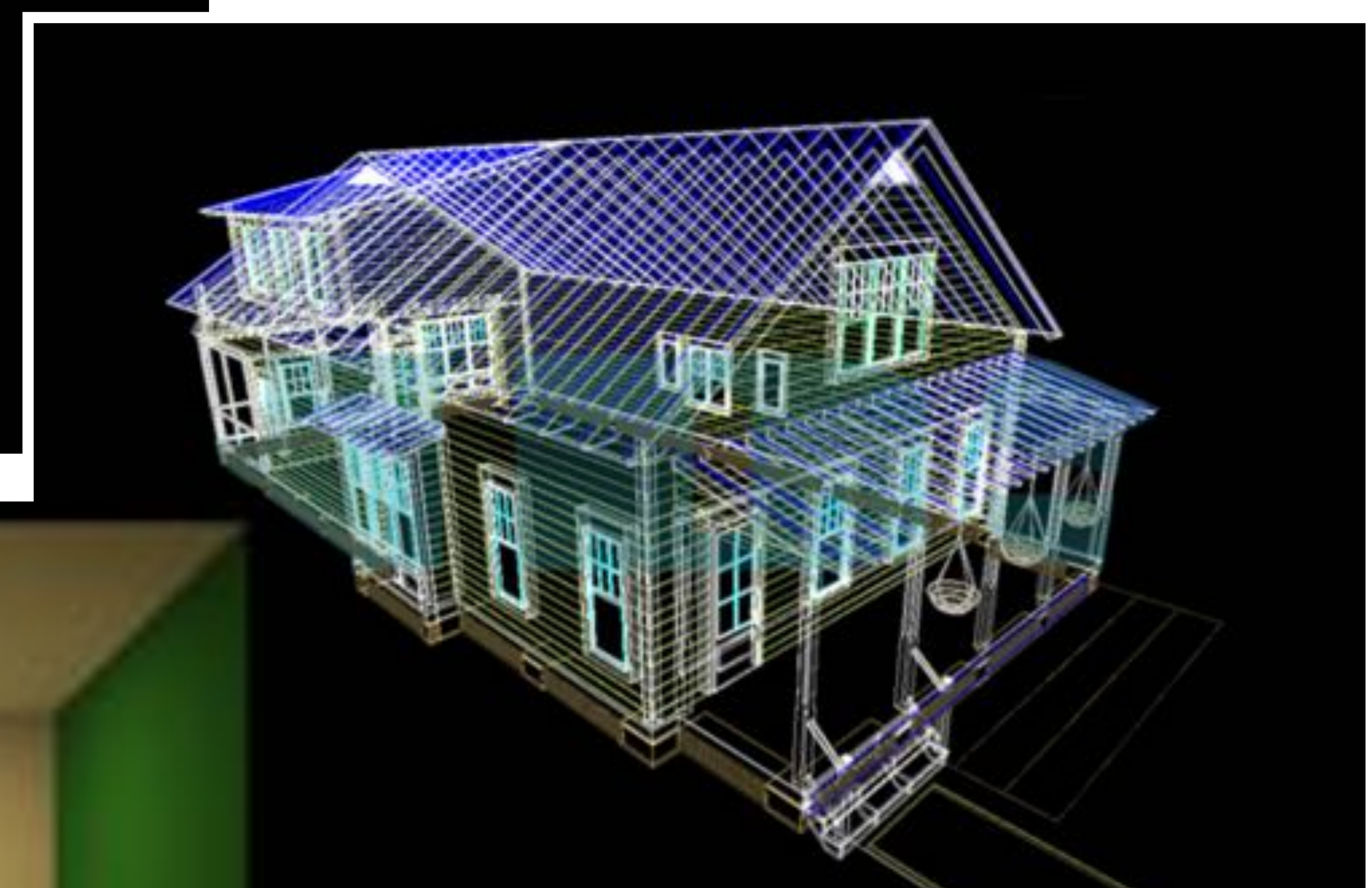
Early painting: incorrect perspective

Carolingian painting, 8-9th century

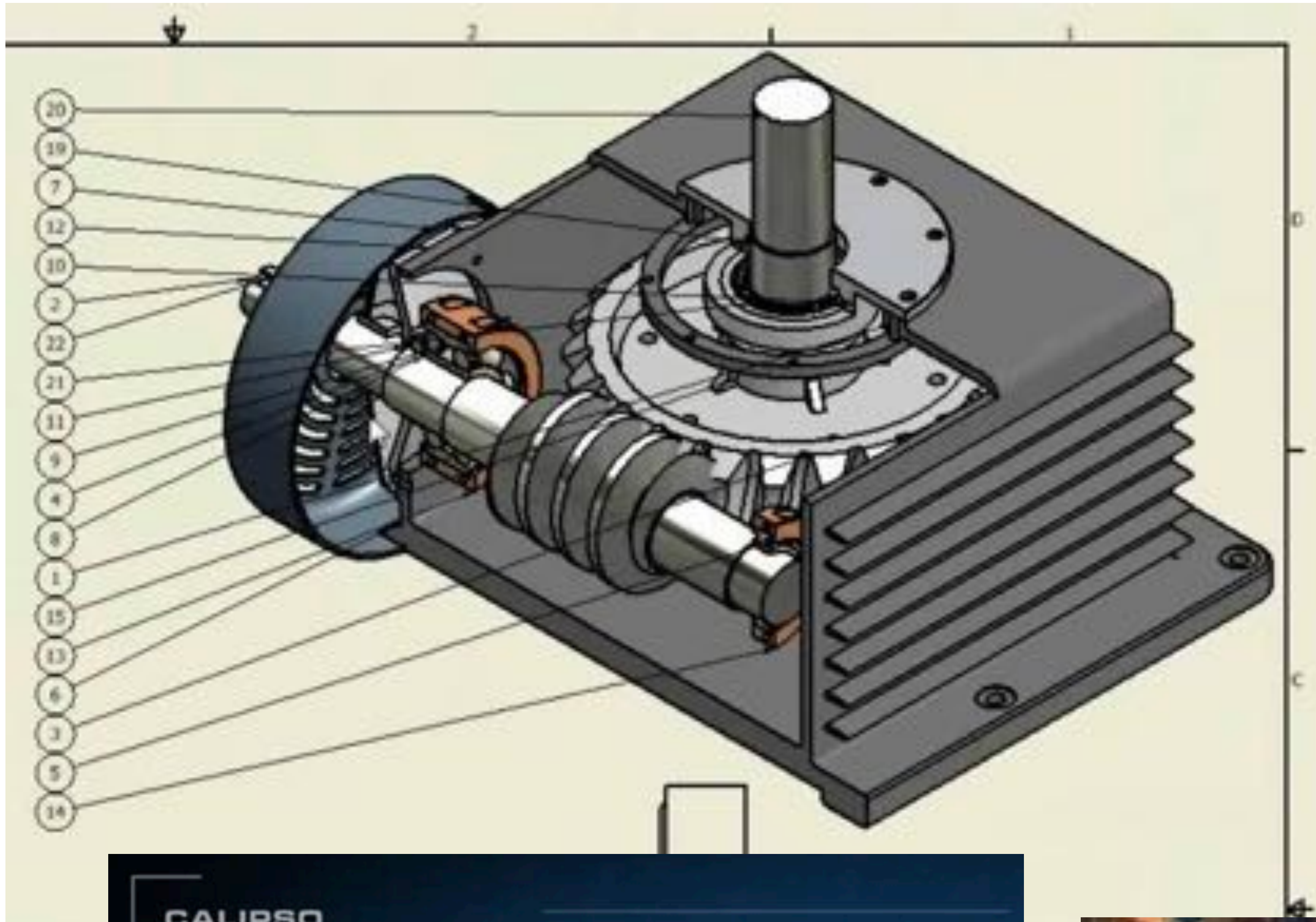
Evolution toward correct perspective

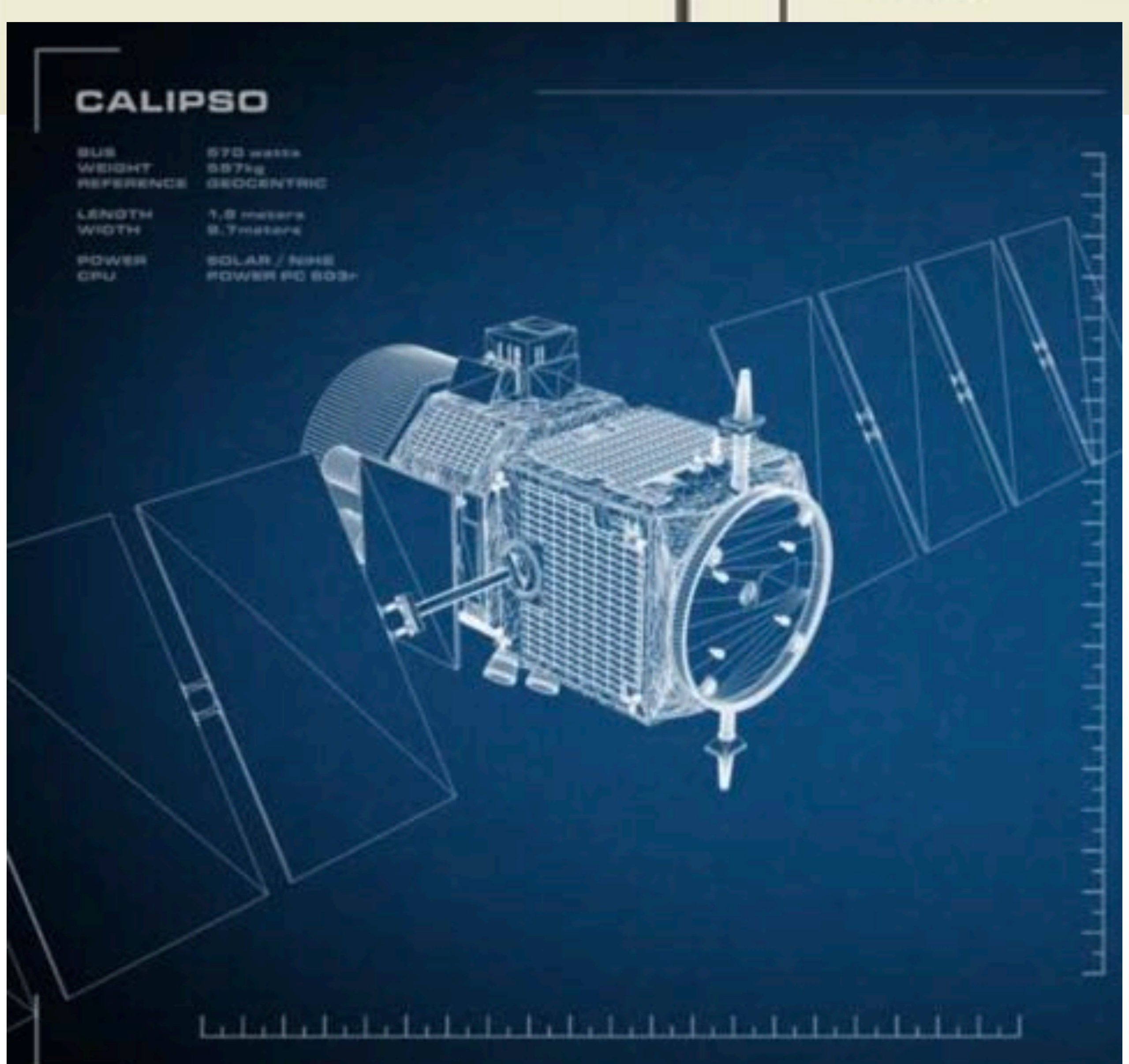
Later… rejection of proper perspective projection

Return of perspective in computer graphics



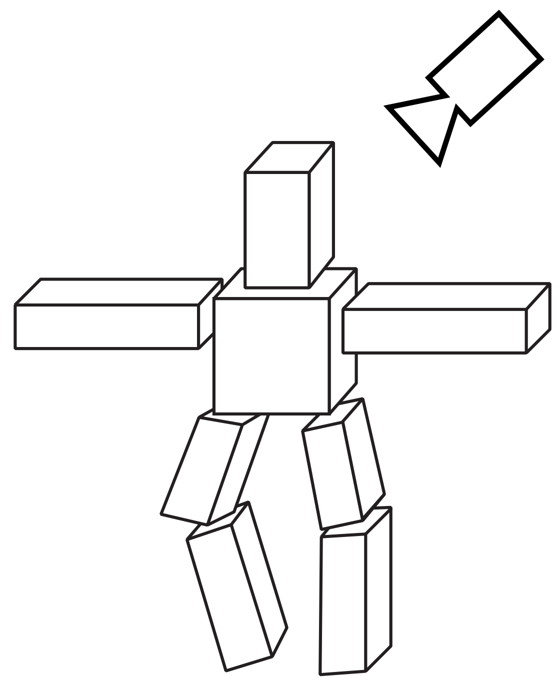
Rejection of perspective in computer graphics





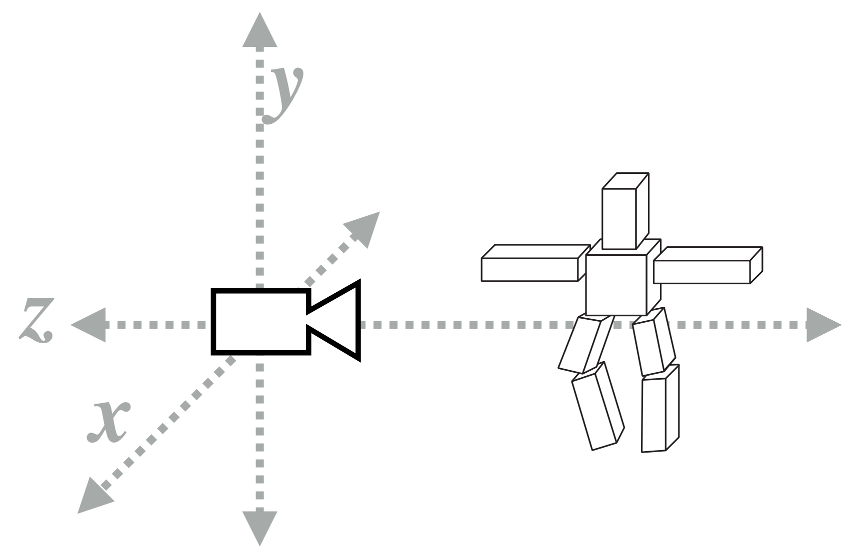
Transformations: From Objects to the Screen

[WORLD COORDINATES]



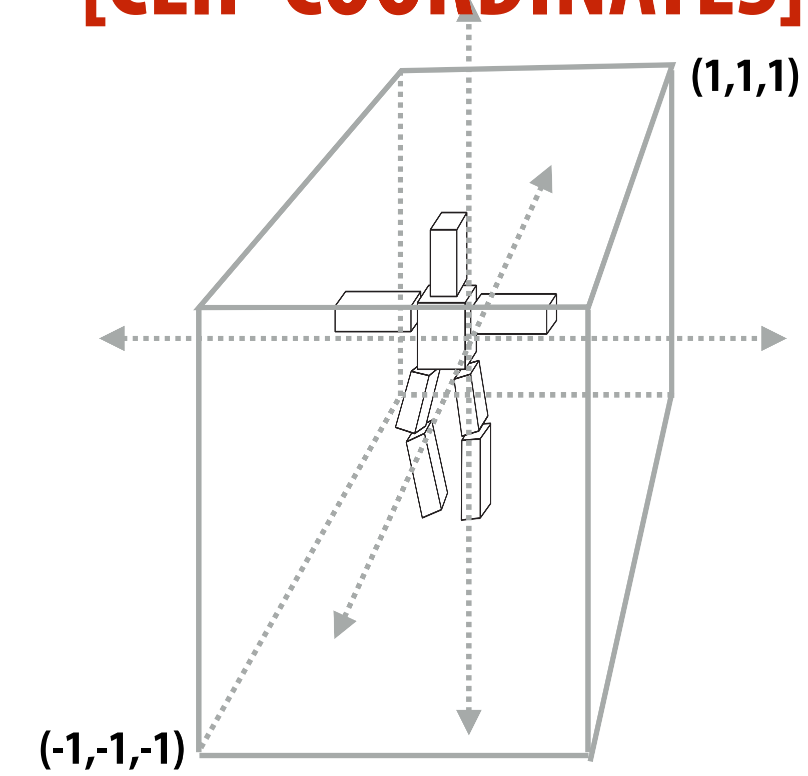
original description of objects

[VIEW COORDINATES]



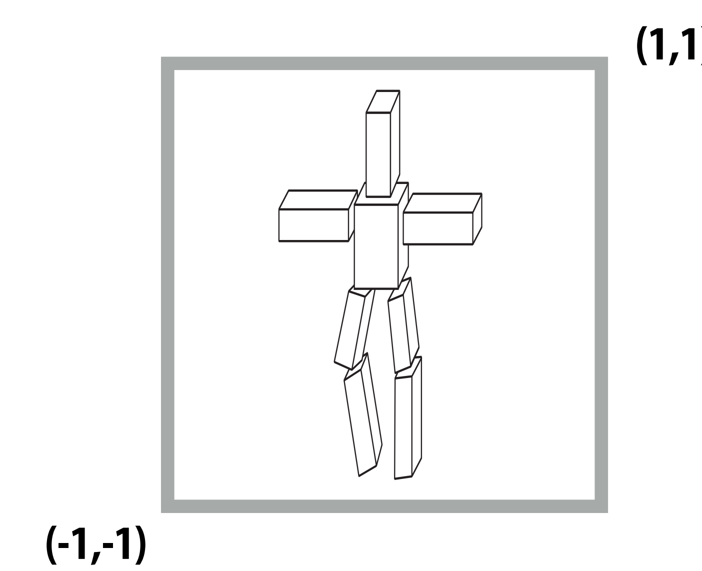
all positions now expressed relative to camera; camera is sitting at origin looking down -z direction

[CLIP COORDINATES]



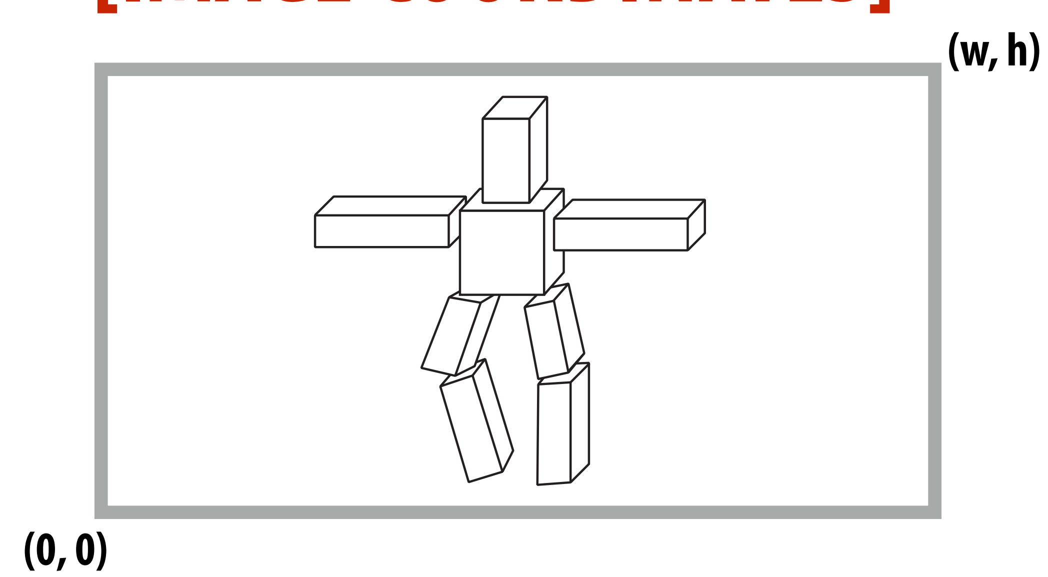
everything visible to the camera is mapped to unit cube for easy “clipping”

[NORMALIZED COORDINATES]



unit cube mapped to unit square via perspective divide

[IMAGE COORDINATES]

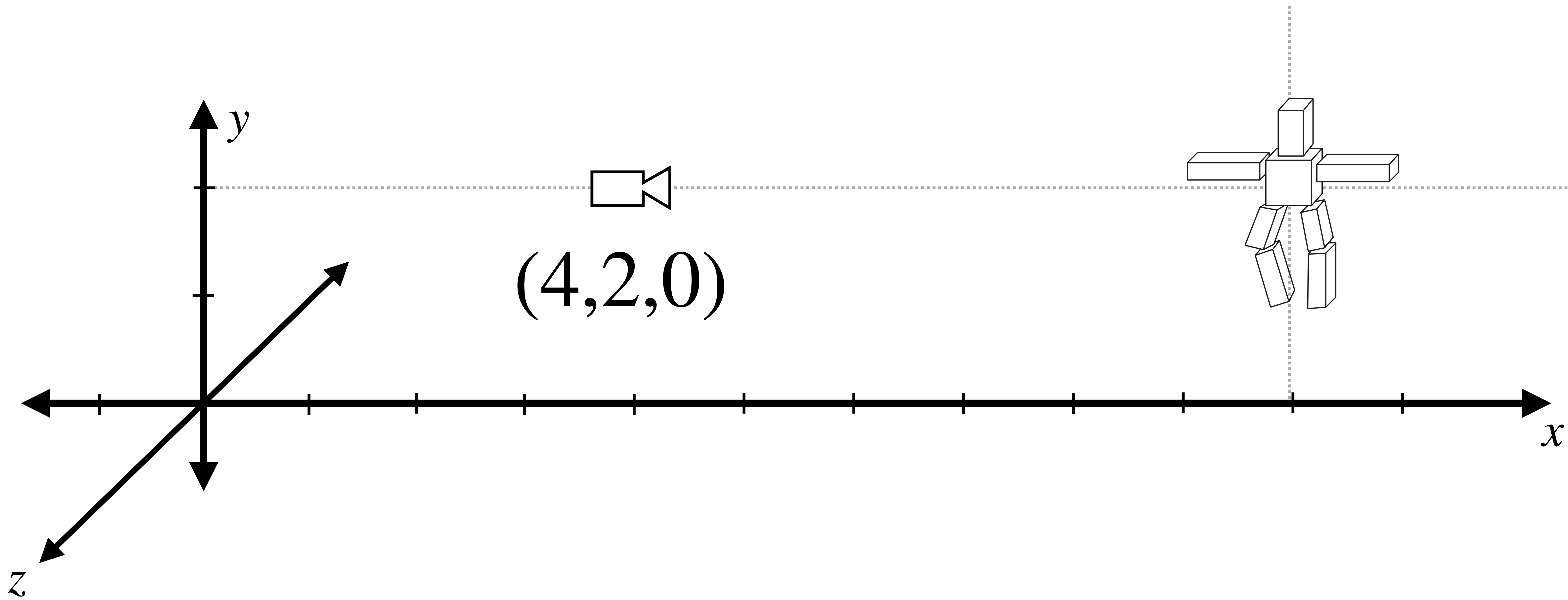


coordinates stretched to match image dimensions (and flipped upside-down)

2D primitives can now be drawn via rasterization

Review: simple camera transform

Consider camera at $(4,2,0)$, looking down x-axis, object given in world coordinates:

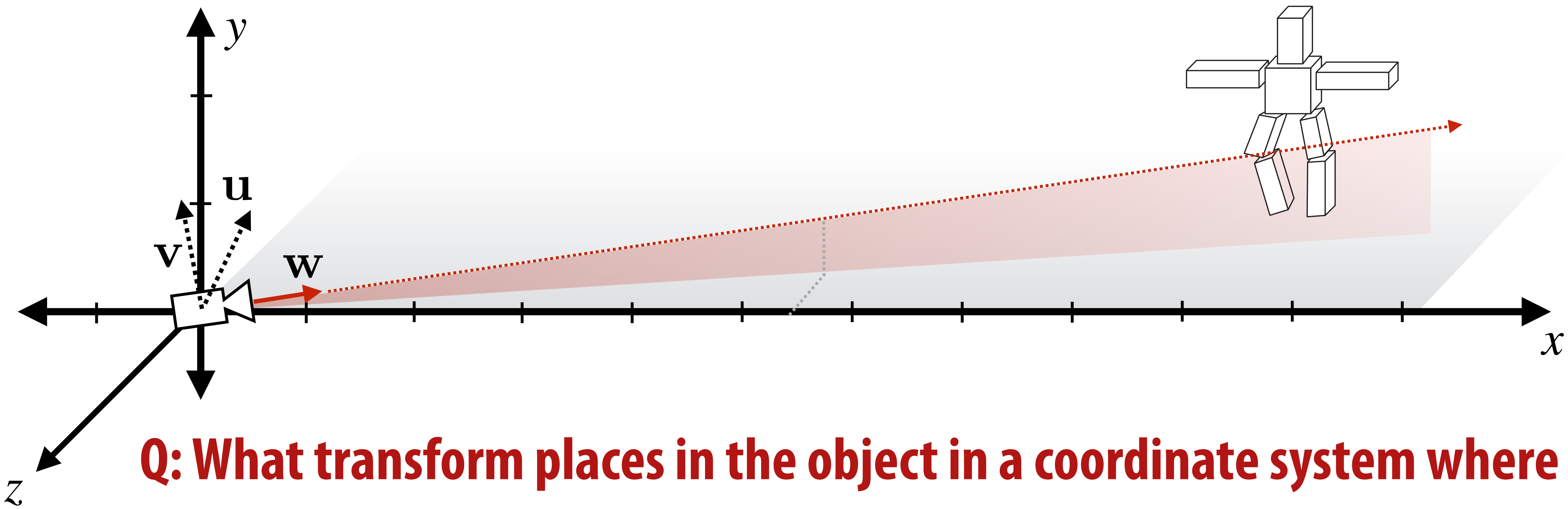


Q: What spatial transformation puts in the object in a coordinate system where the camera is at the origin, looking down the $-z$ axis?

- **Translating object vertex positions by (-4, -2, 0) yields position relative to camera**
- **Rotation about y by $\pi/2$ gives position of object in new coordinate system where camera's view direction is aligned with the $-z$ axis**

Camera looking in a different direction

Now consider a camera looking in a direction $\mathbf{w} \in \mathbb{R}^3$



Q: What transform places in the object in a coordinate system where the camera is at the origin and the camera is looking directly down the z axis?

- Construct vectors $\mathbf{u}$, $\mathbf{v}$ orthogonal to $\mathbf{w}$
  - e.g., with $\mathbf{y}$ as "up vector", let $\mathbf{u} := \mathbf{y} \times \mathbf{w}$
- We need one more basis: $\mathbf{v} := \mathbf{w} \times \mathbf{u}$
- Normalize everything:

$$\hat{\mathbf{u}} := \frac{\mathbf{u}}{||\mathbf{u}||} \quad \hat{\mathbf{v}} := \frac{\mathbf{v}}{||\mathbf{v}||} \quad \hat{\mathbf{w}} := \frac{\mathbf{w}}{||\mathbf{w}||}$$

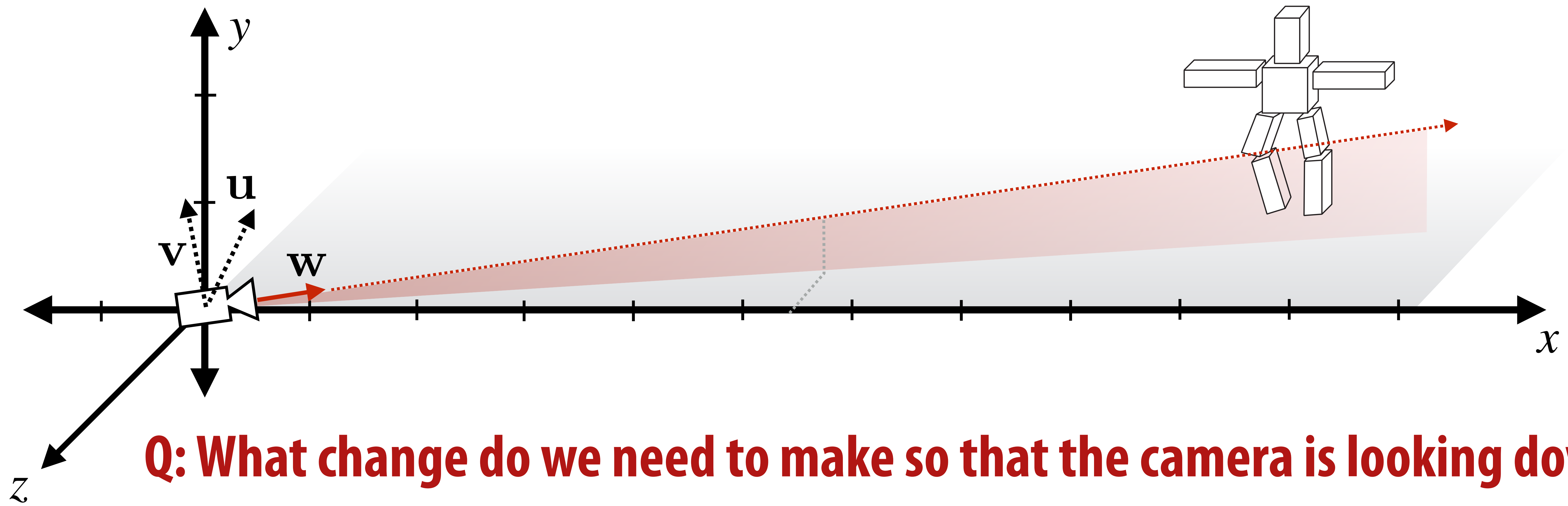
Form a matrix with basis vectors in rows

$$R = \begin{bmatrix} \hat{u}_x & \hat{u}_y & \hat{u}_z \\ \hat{v}_x & \hat{v}_y & \hat{v}_z \\ \hat{w}_x & \hat{w}_y & \hat{w}_z \end{bmatrix}$$

R maps $\hat{\mathbf{u}}$ to $\mathbf{x}$-axis, $\hat{\mathbf{v}}$ to $\mathbf{y}$-axis, $\hat{\mathbf{w}}$ to $\mathbf{z}$-axis

Camera looking in a different direction

Now consider a camera looking in a direction $\mathbf{w} \in \mathbb{R}^3$



Q: What change do we need to make so that the camera is looking down -z?

- Find basis mapping
  - $\hat{\mathbf{u}}$ to the $\mathbf{x}$-axis, $\hat{\mathbf{v}}$ to the $\mathbf{y}$-axis, $-\hat{\mathbf{w}}$ to $\mathbf{z}$-axis
- Construct vectors $\mathbf{u}$, $\mathbf{v}$ orthogonal to $-\mathbf{w}$
  - e.g., with $\mathbf{y}$ as "up vector", let

$$\mathbf{u} := \mathbf{y} \times (-\mathbf{w}) = \mathbf{w} \times \mathbf{y}$$

- We need one more basis:

$$\mathbf{v} := -\mathbf{w} \times \mathbf{u} = \mathbf{u} \times \mathbf{w}$$

- Normalize everything as before

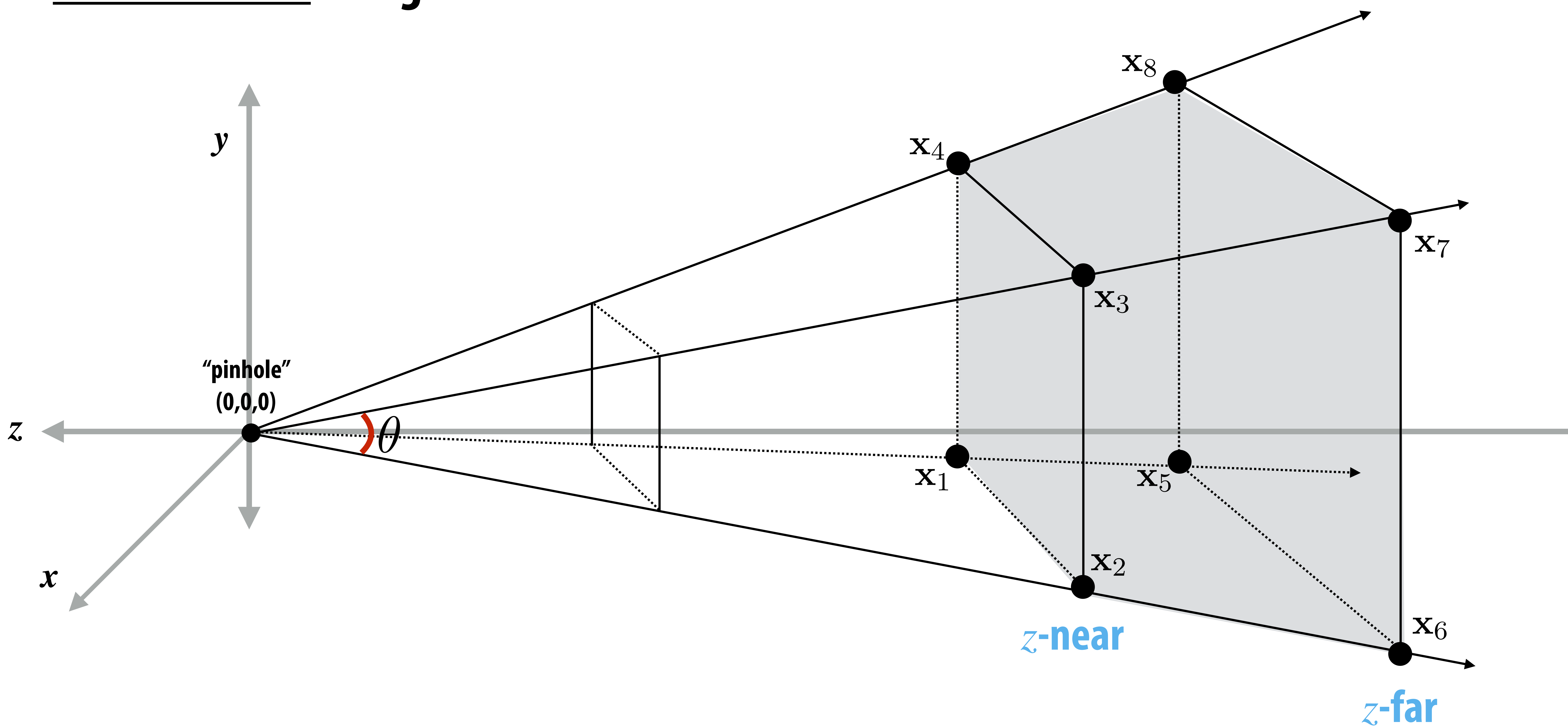
Form a matrix with basis vectors in rows

$$R = \begin{bmatrix} \hat{u}_x & \hat{u}_y & \hat{u}_z \\ \hat{v}_x & \hat{v}_y & \hat{v}_z \\ -\hat{w}_x & -\hat{w}_y & -\hat{w}_z \end{bmatrix}$$

R maps $\hat{\mathbf{u}}$ to $\mathbf{x}$-axis, $\hat{\mathbf{v}}$ to $\mathbf{y}$-axis, $\hat{\mathbf{w}}$ to $-\mathbf{z}$-axis

View frustum

View frustum is region the camera can see:



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

Clipping

- "Clipping" eliminates triangles not visible to the camera / in view frustum
  - Don't waste time rasterizing primitives (e.g., triangles) you can't see!
  - Discarding individual fragments is expensive ("fine granularity")
  - Makes more sense to toss out whole primitives ("coarse granularity")
  - Still need to deal with primitives that are partially clipped…

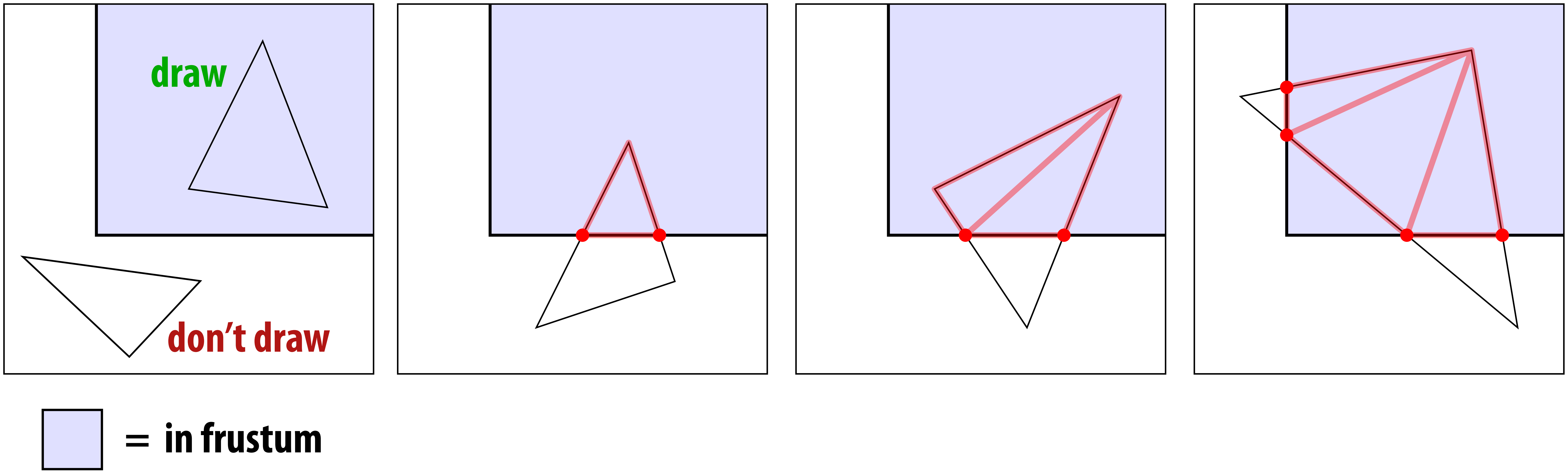
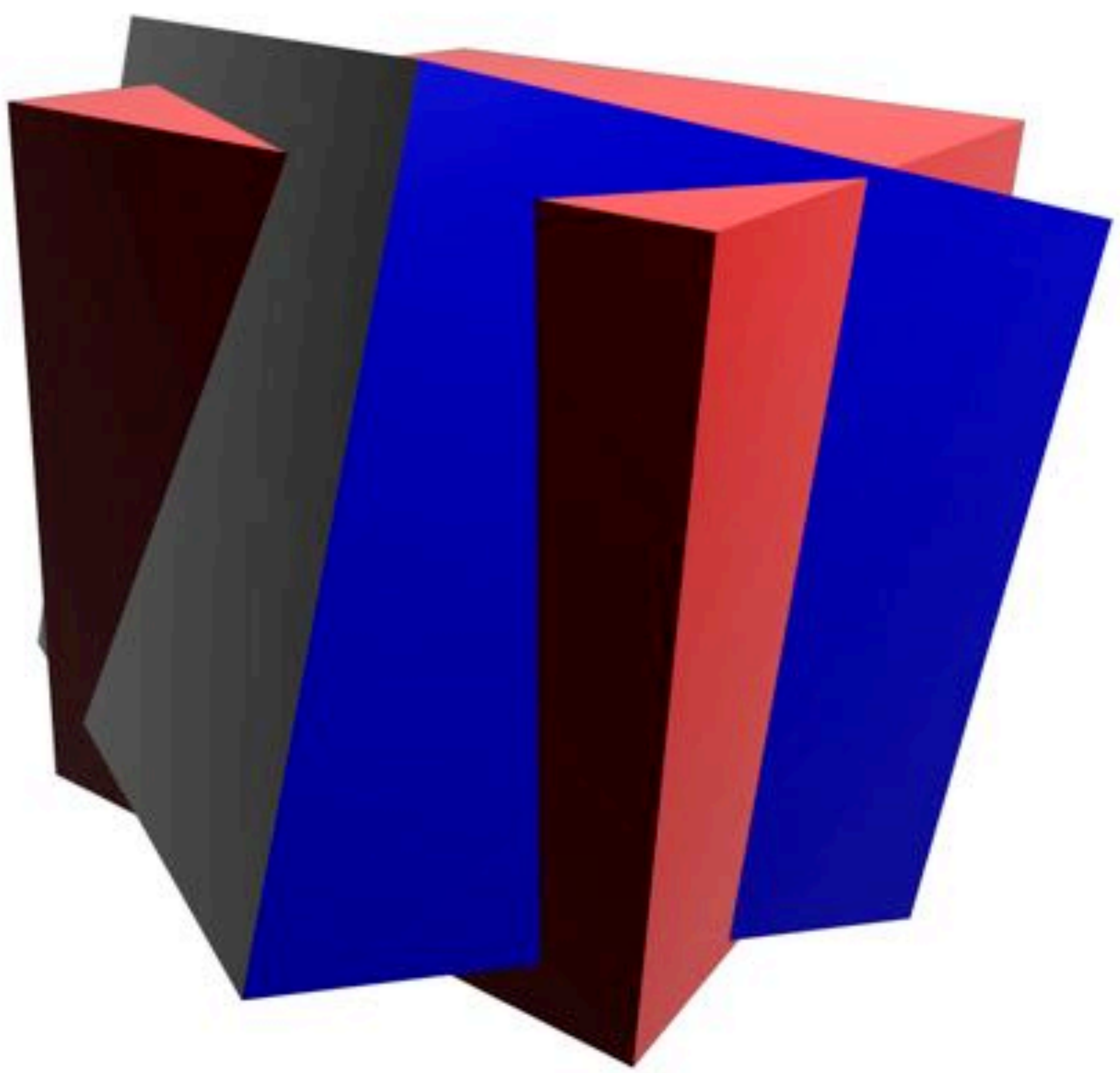


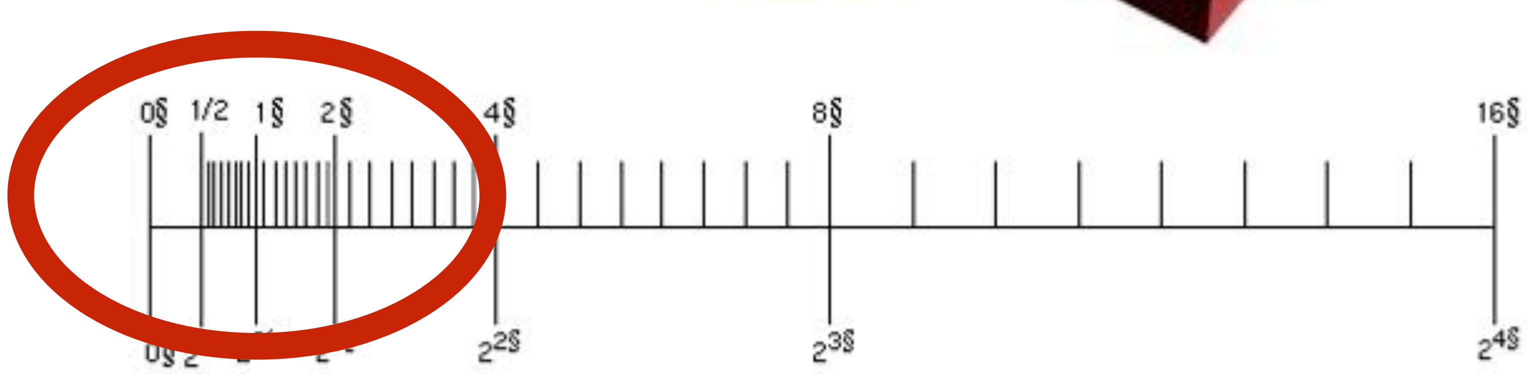
image credit: Jason L. McKesson (https://paroj.github.io/gltut/)

Near/Far Clipping

- Why have near/far clipping planes?
  - Some primitives (e.g., triangles) may have vertices both in front & behind eye! (Causes headaches for rasterization, e.g., checking if fragments are behind eye)
  - Also important for dealing with finite precision of depth buffer / limitations on storing depth as floating point values

near = 10^{-1}
far = 10^{3}

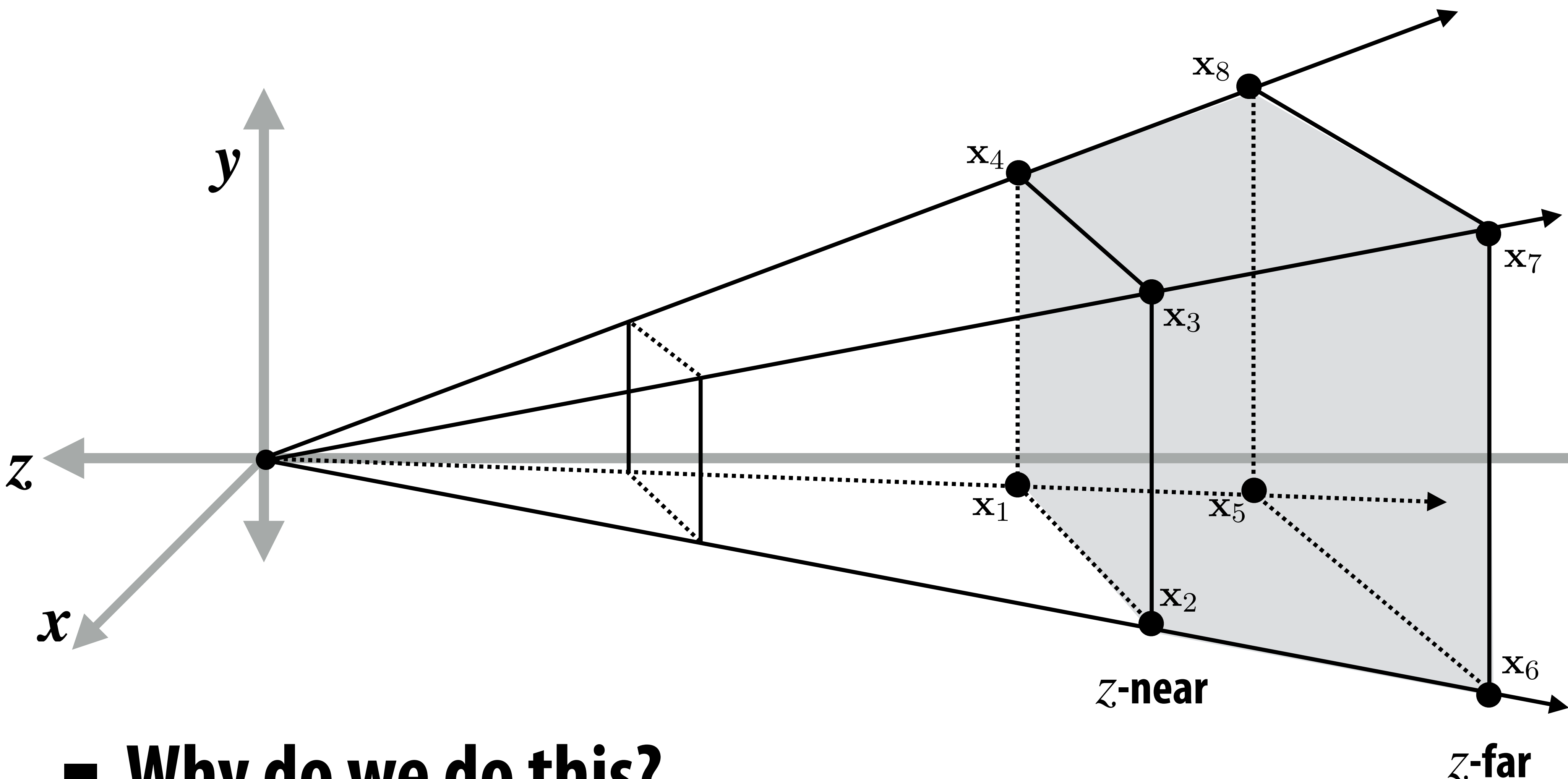


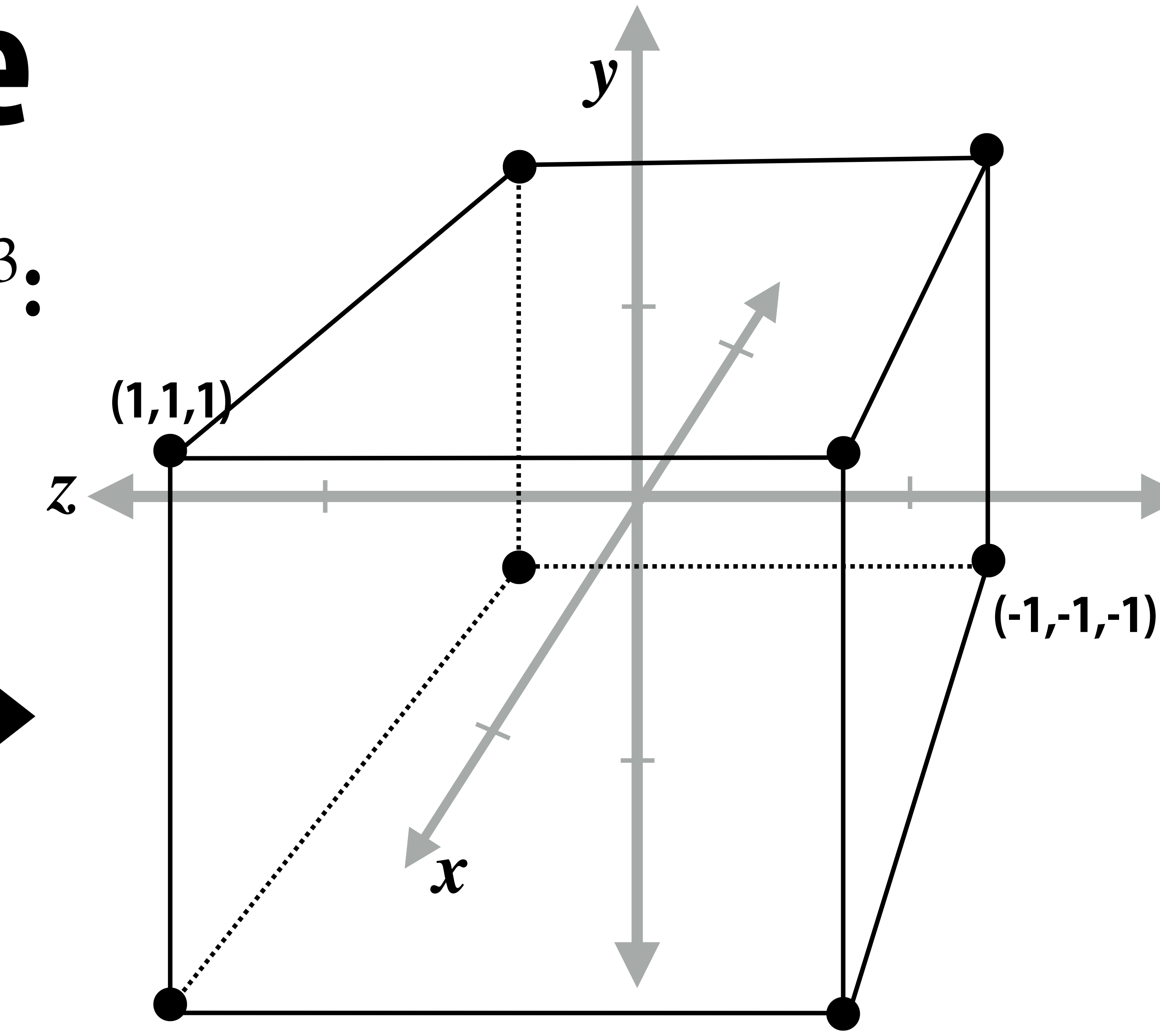


floating point has more "resolution" near zero—hence more precise resolution of primitive-primitive intersection

Mapping frustum to unit cube

Before projecting to 2D, map view frustum to cube $[-1,1]^3$:





- Why do we do this?
- Makes clipping much easier!
  - just discard points outside range [-1,1]
  - need to think about partially-clipped triangles
- Q: How can we express this mapping as a matrix?
- A: Solve $A\mathbf{x}_i = \mathbf{y}_i$ for unknown entries of A

scale to size 2

$$A = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & \frac{l+r}{l-r} \\ 0 & \frac{2}{t-b} & 0 & \frac{b+t}{b-t} \\ 0 & 0 & \frac{2}{n-f} & \frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

translate to origin

$l =$ left $\quad b =$ bottom $\quad n =$ near

$r =$ right $\quad t =$ top $\quad f =$ far

$$\begin{aligned} \mathbf{x}_1 &= \{l,b,n,1\} \\ \mathbf{x}_2 &= \{r,b,n,1\} \\ \mathbf{x}_3 &= \{r,t,n,1\} \\ \mathbf{x}_4 &= \{l,t,n,1\} \\ \mathbf{x}_5 &= \{l,b,f,1\} \\ \mathbf{x}_6 &= \{r,b,f,1\} \\ \mathbf{x}_7 &= \{r,t,f,1\} \\ \mathbf{x}_8 &= \{l,t,f,1\} \end{aligned}$$

$$\begin{aligned} \mathbf{y}_1 &= \{-1,-1,1,1\} \\ \mathbf{y}_2 &= \{1,-1,1,1\} \\ \mathbf{y}_3 &= \{1,1,1,1\} \\ \mathbf{y}_4 &= \{-1,1,1,1\} \\ \mathbf{y}_5 &= \{-1,-1,-1,1\} \\ \mathbf{y}_6 &= \{1,-1,-1,1\} \\ \mathbf{y}_7 &= \{1,1,-1,1\} \\ \mathbf{y}_8 &= \{-1,1,-1,1\} \end{aligned}$$

(orthographic projection)

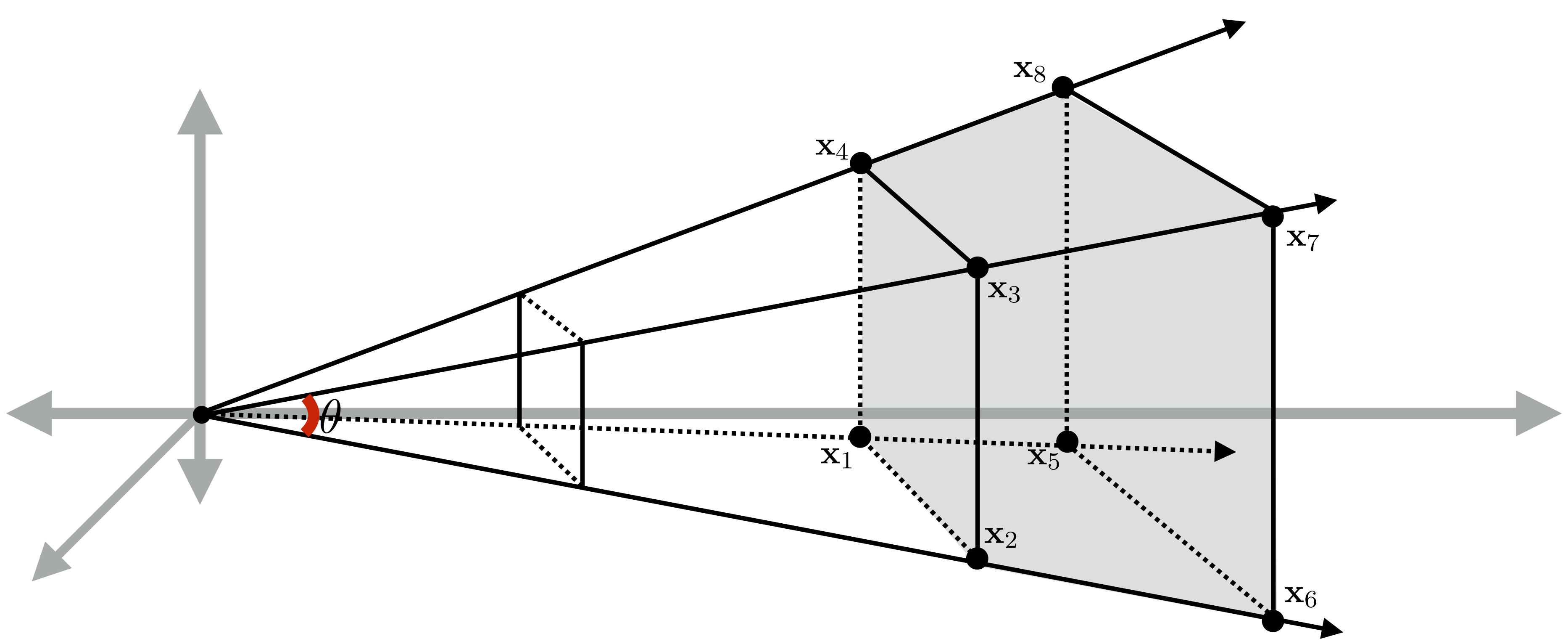
Matrix for Perspective Transform

Recall our basic perspective projection 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

Full perspective matrix takes geometry of view frustum into account:



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

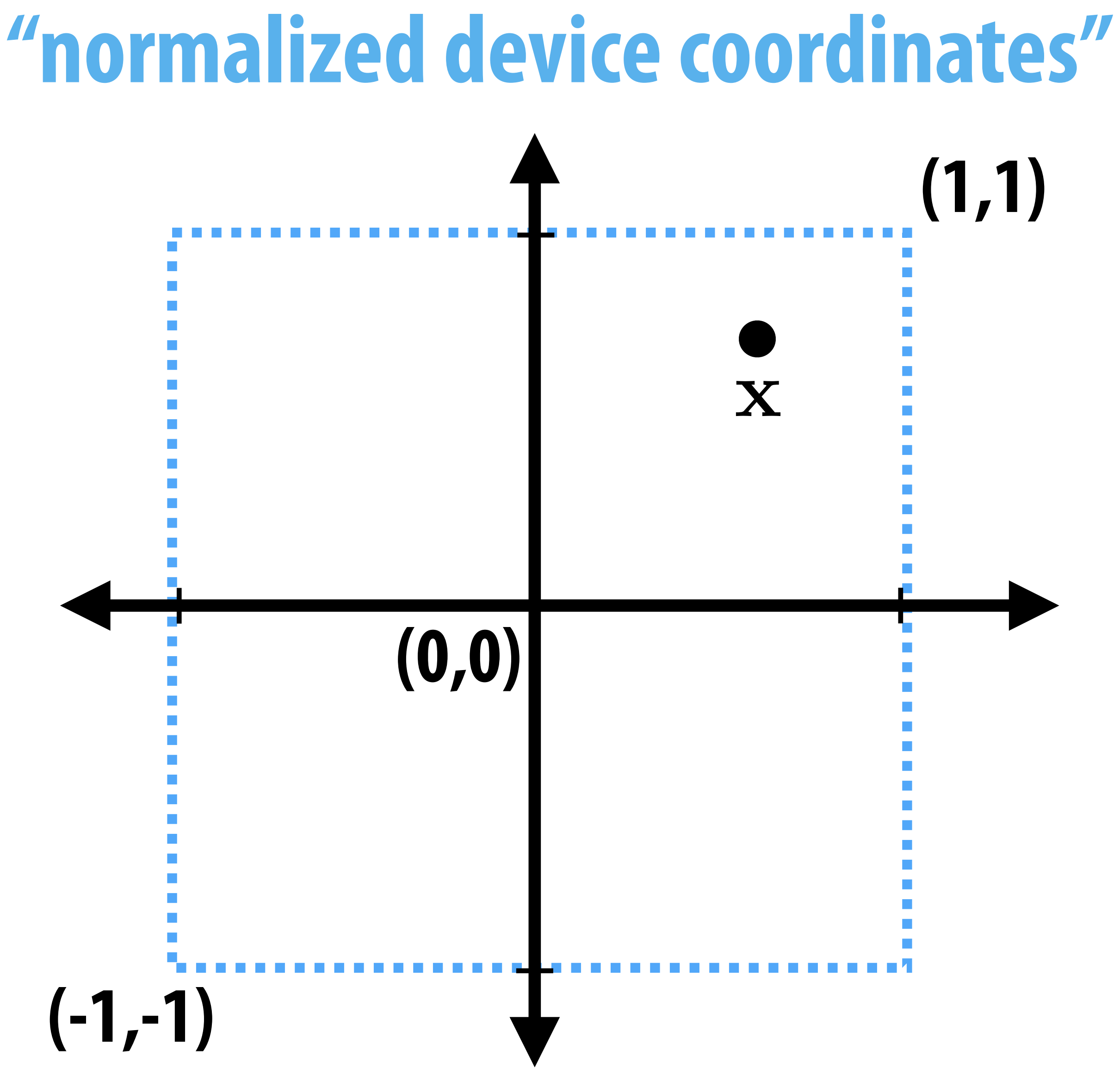
l = left b = bottom n = near

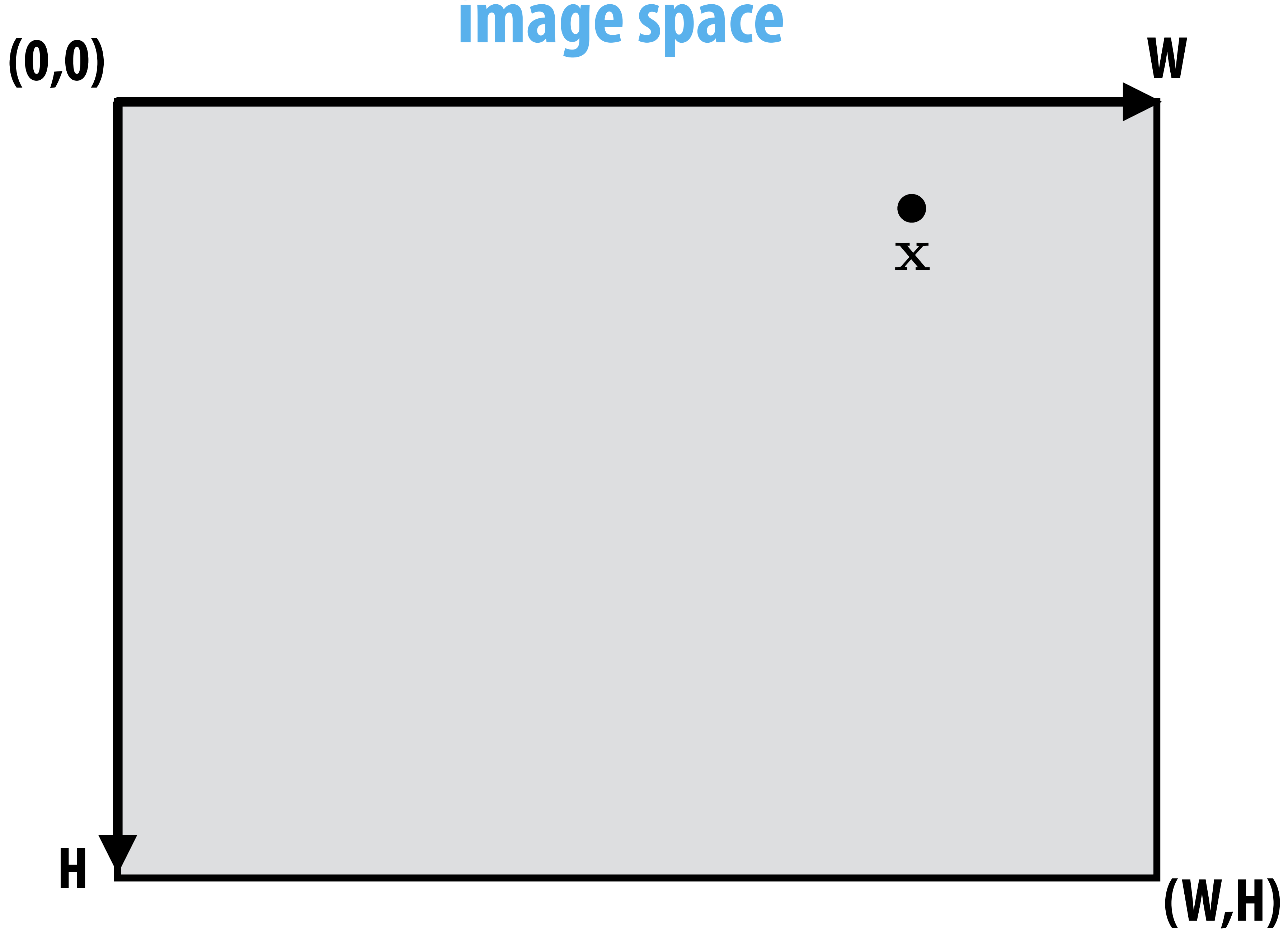
r = right t = top f = far

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

Review: screen transformation

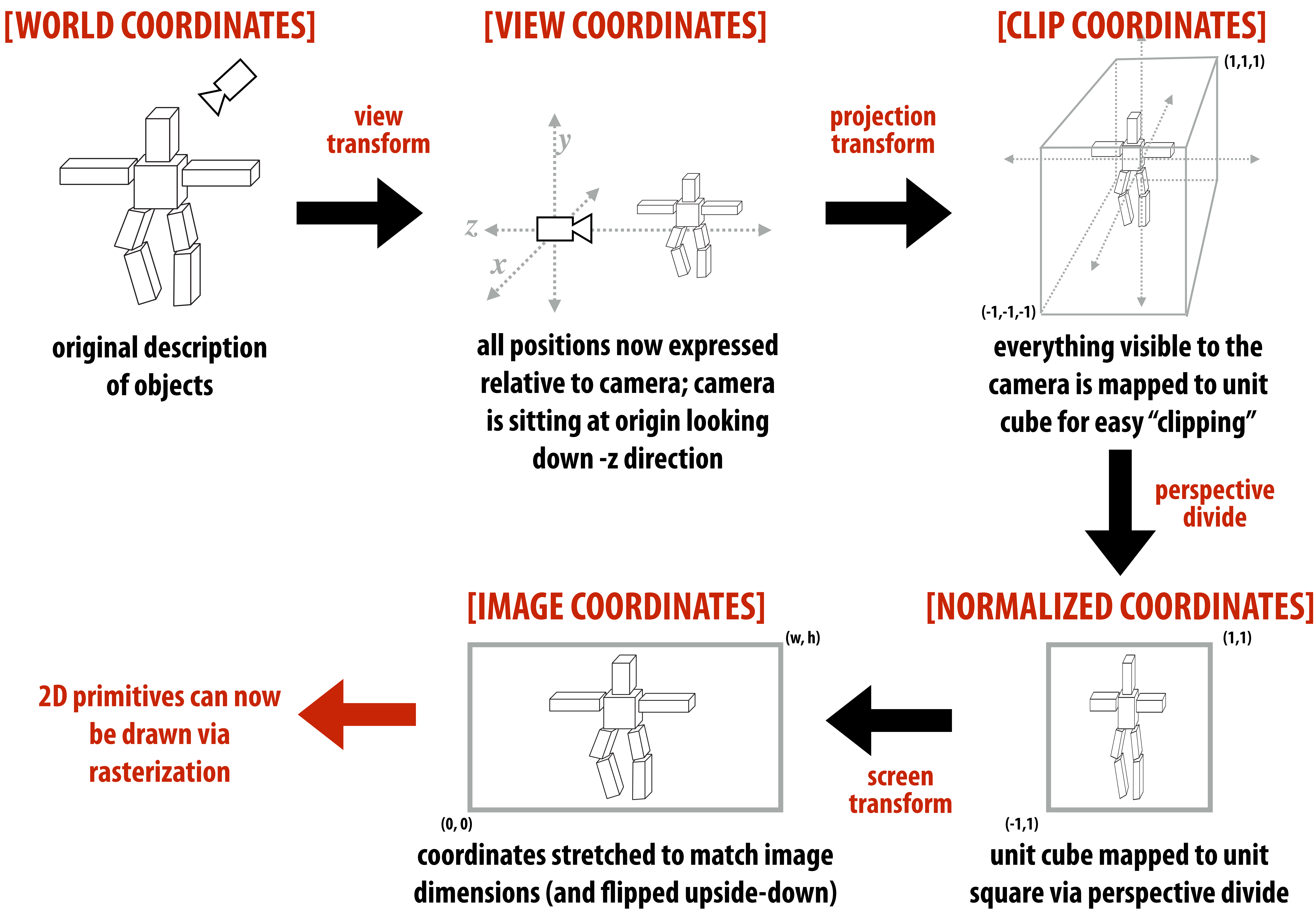
- Had one last transformation in the rasterization pipeline: transform from 2D viewing plane to pixel coordinates
- Projection will take points to [-1,1] x [-1,1] on the z = 1 plane; transform into a W x H pixel image





Step 1: reflect about x-axis
Step 2: translate by (1,1)
Step 3: scale by (W/2,H/2)

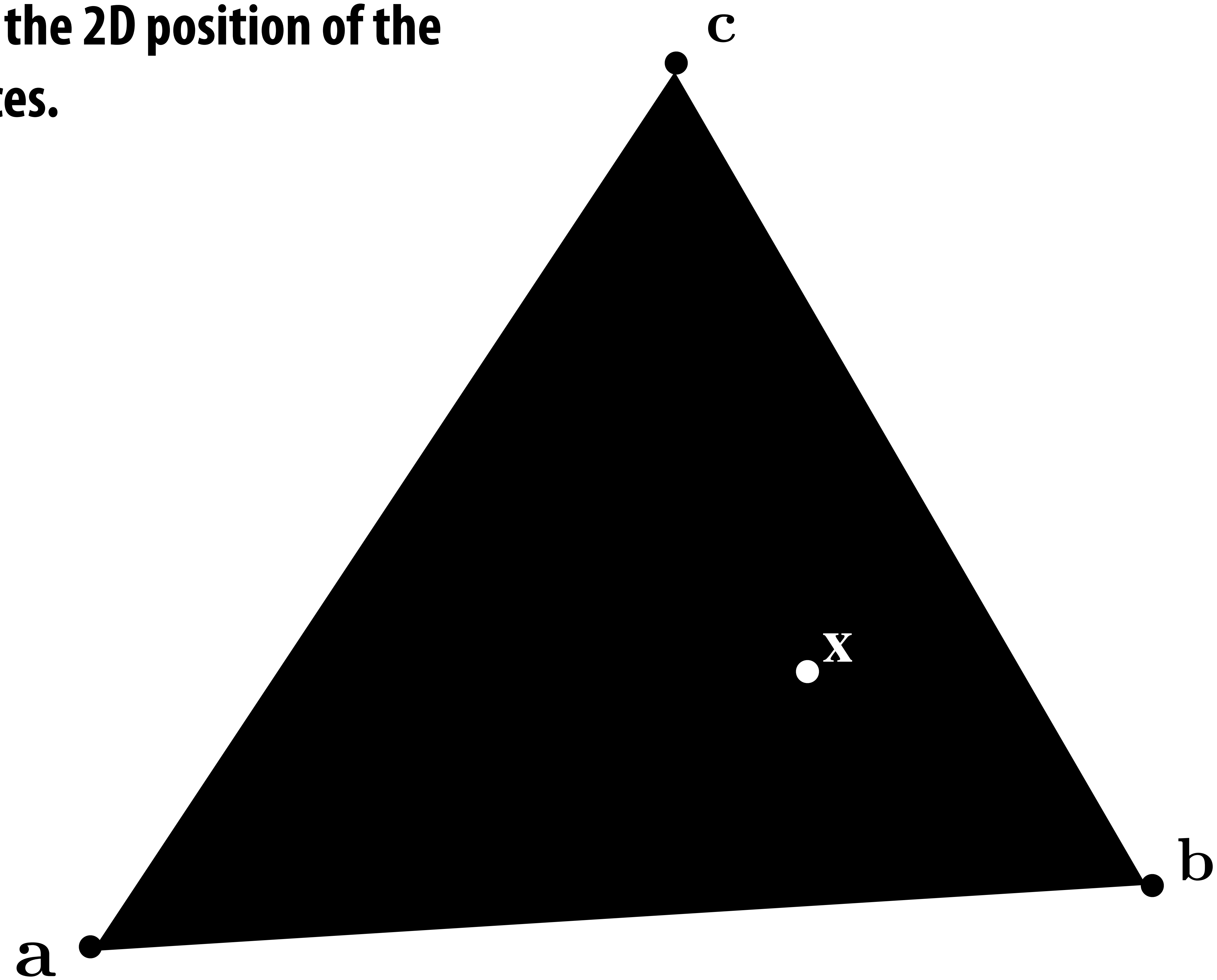
Transformations: From Objects to the Screen



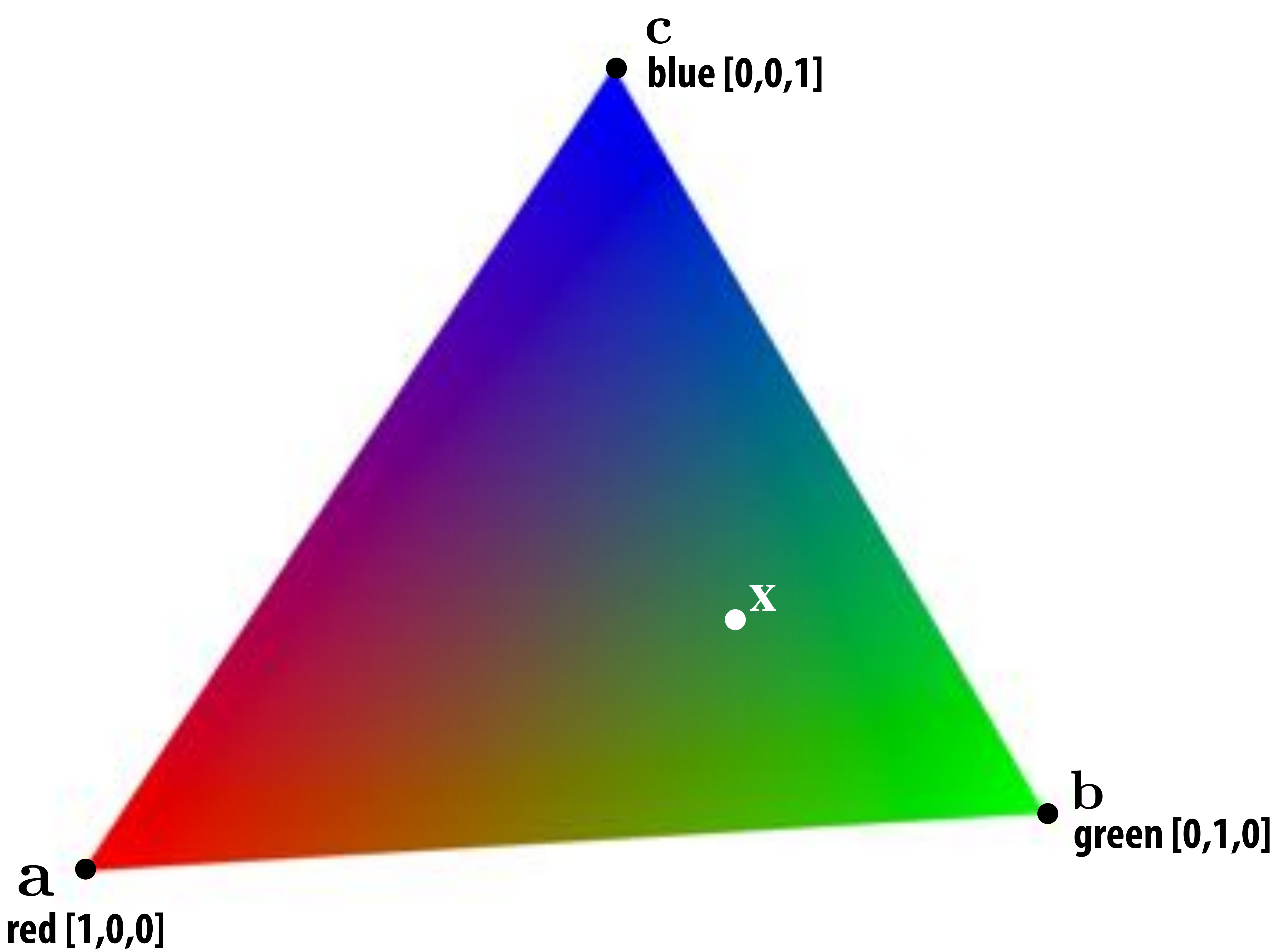
So, how do we draw nice primitives?

Coverage(x,y)

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



Consider sampling color(x,y)



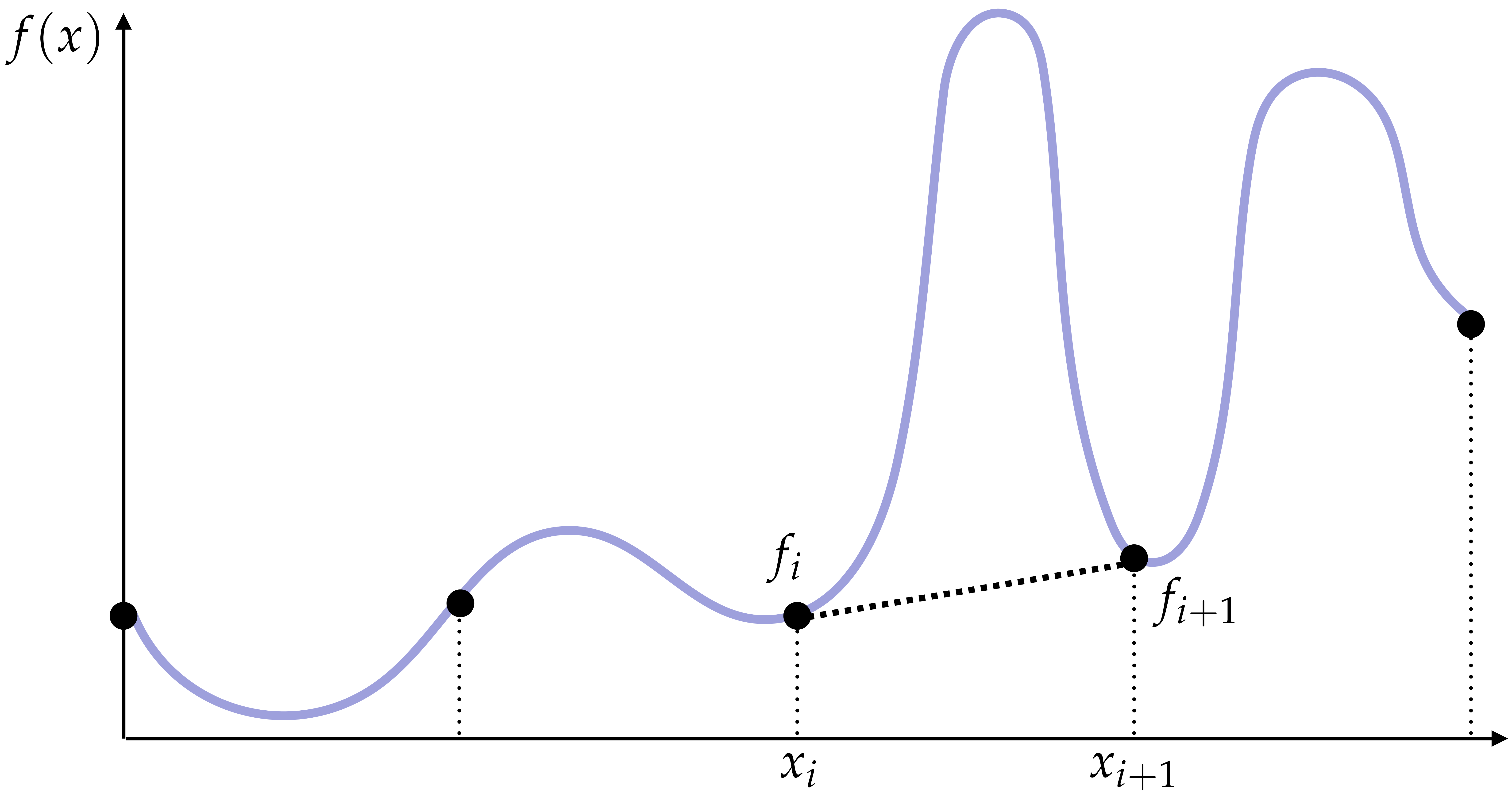
What is the triangle's color at the point $\mathbf{x}$?

Standard strategy: <u>interpolate</u> color values at vertices.

Linear interpolation in 1D

Suppose we've sampled values of a function f(x) at points x_i, i.e., $f_i := f(x_i)$

Q: How do we construct a function that "connects the dots" between x_i and x_{i+1}?



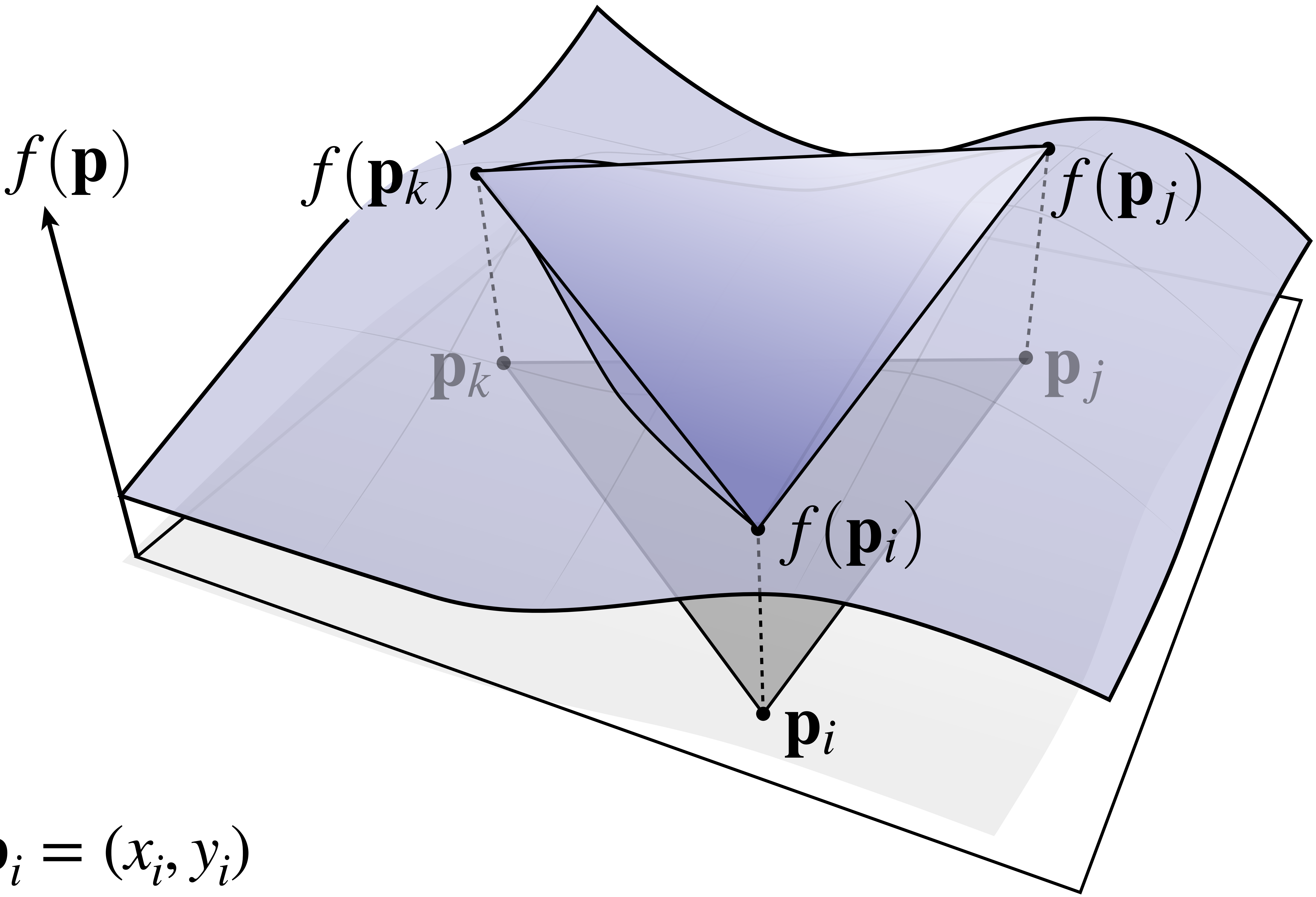
$$t := (x - x_i)/(x_{i+1} - x_i) \in [0,1]$$

$$\hat{f}(t) = f_i + t(f_{i+1} - f_i) = (1-t)f_i + tf_{i+1}$$

Linear interpolation in 2D

Suppose we've likewise sampled values of a function $f(\mathbf{p})$ at points $\mathbf{p}_i$, $\mathbf{p}_j$, $\mathbf{p}_k$ in 2D

Q: How do we "connect the dots" this time? E.g., how do we fit a plane?



$\mathbf{p}_i = (x_i, y_i)$

Linear interpolation in 2D

- **Want to fit a linear (really, affine) function to three values**
- **Any such function has three unknown coefficients a, b, and c:**

$$\hat{f}(x,y) = ax + by + c$$

- **To interpolate, we need to find coefficients such that the function matches the sample values at the sample points:**

$$\hat{f}(x_n, y_n) = f_n,\ n \in \{i, j, k\}$$

- **Yields three linear equations in three unknowns. Solution?**

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \frac{1}{(x_j y_i - x_i y_j) + (x_k y_j - x_j y_k) + (x_i y_k - x_k y_i)} \begin{bmatrix} f_i(y_k - y_j) + f_j(y_i - y_k) + f_k(y_j - y_i) \\ f_i(x_j - x_k) + f_j(x_k - x_i) + f_k(x_i - x_j) \\ f_i(x_k y_j - x_j y_k) + f_j(x_i y_k - x_k y_i) + f_k(x_j y_i - x_i y_j) \end{bmatrix}$$

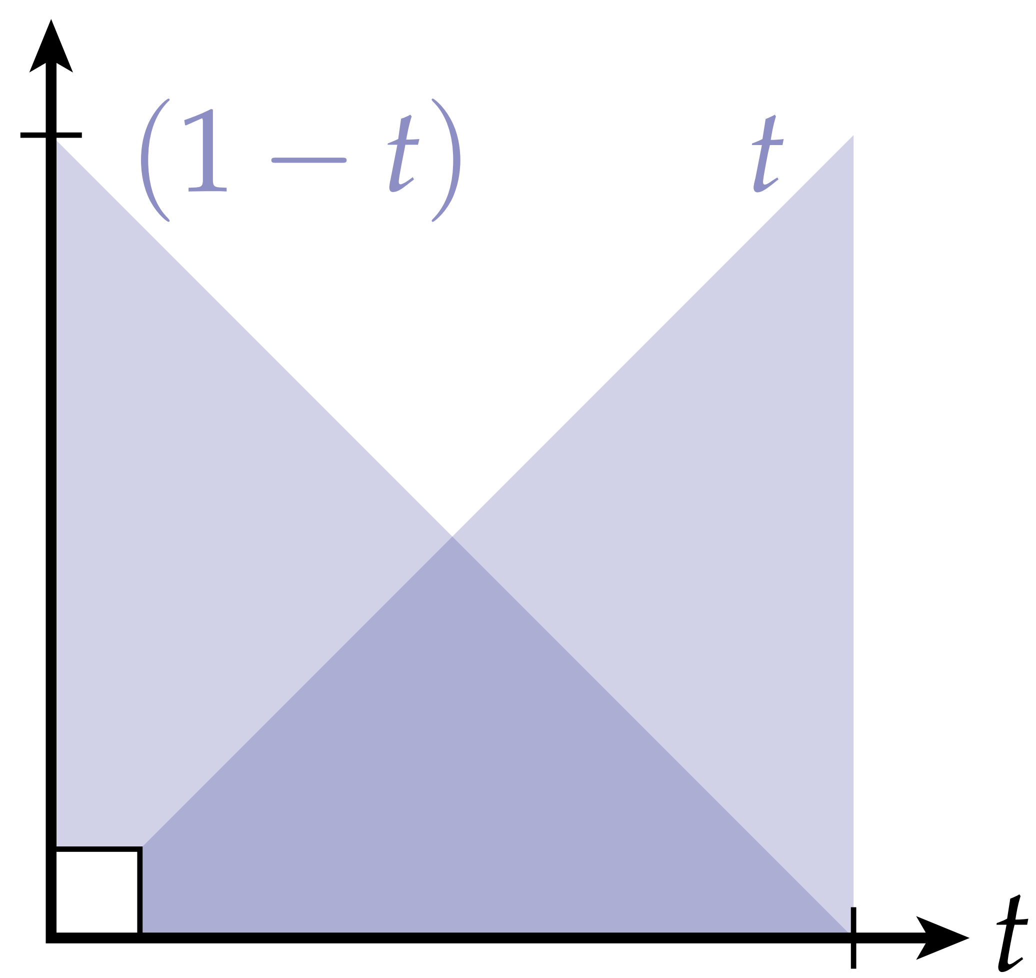
This is ugly. There <u>has</u> to be a better way to think about this…

1D Linear Interpolation, revisited

- **Let's think about how we did linear interpolation in 1D:**

$$\hat{f}(t) = (1-t)f_i + tf_j$$

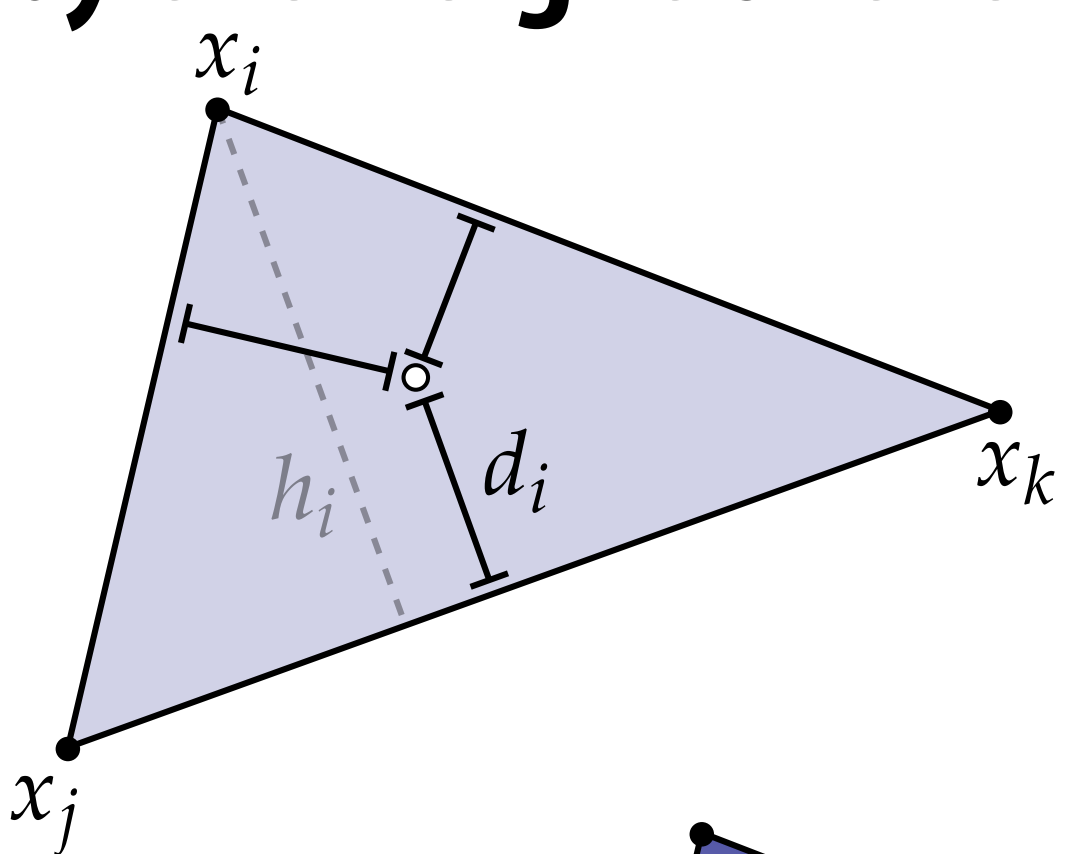
- **Can think of this as a linear combination of two functions:**



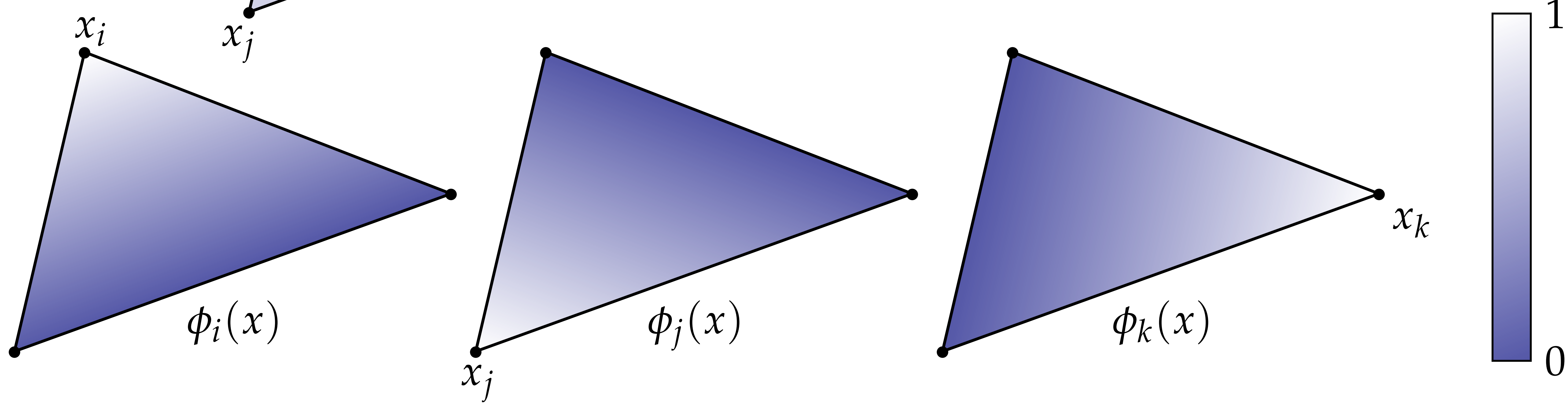
- **As we move closer to t=0, we approach the value of f at x_i**
- **As we move closer to t=1, we approach the value of f at x_j**

2D Linear Interpolation, revisited

- **We can construct analogous functions for a triangle**
- **For a given point x, measure the distance to each edge; then divide by the height of the triangle:**



$$\phi_i(x) = d_i(x)/h_i$$

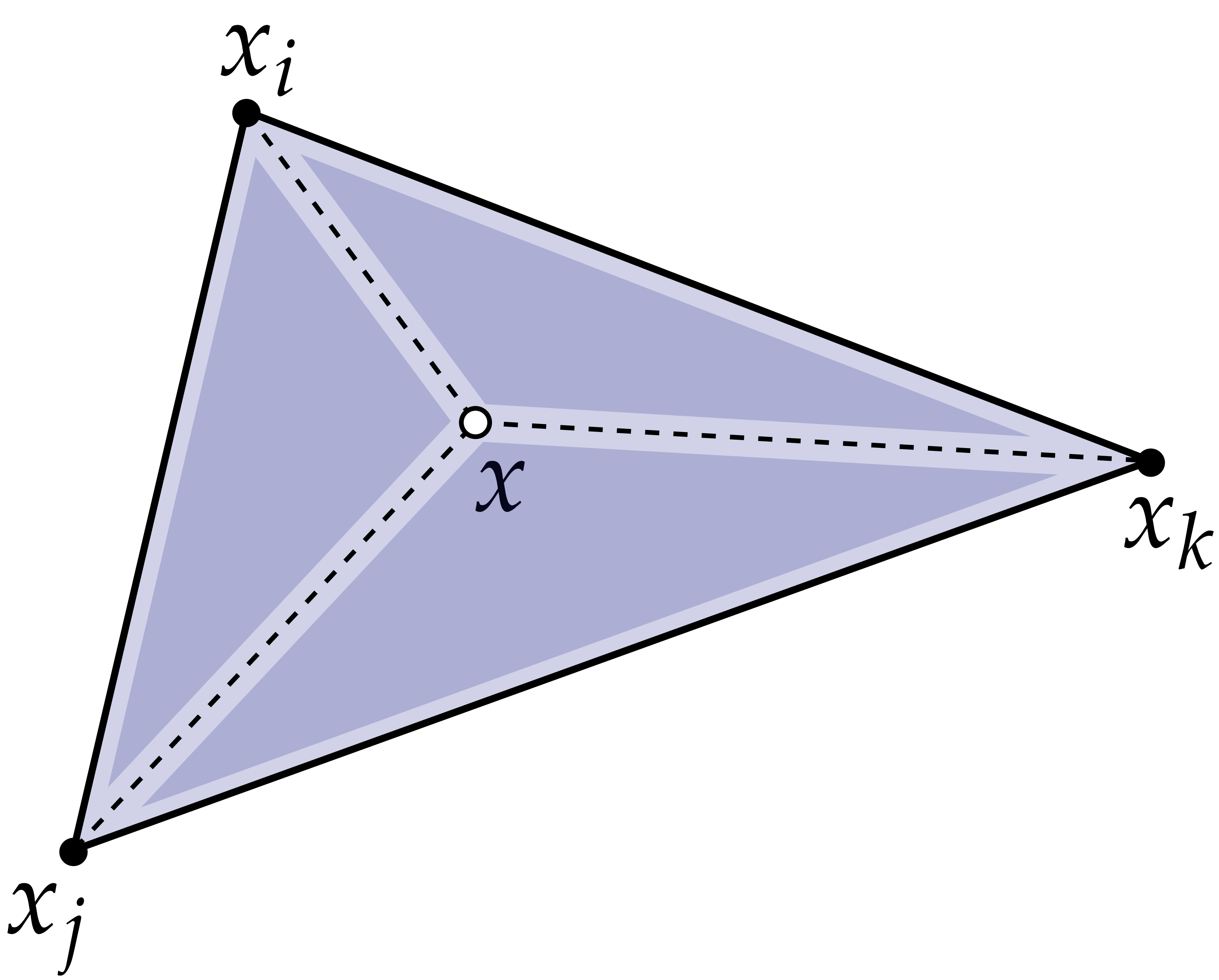


Interpolate by taking linear combination: $\hat{f}(x) = f_i\phi_i + f_j\phi_j + f_k\phi_k$

Q: Is this the same as the (ugly) function we found before?

2D Interpolation, another way

- **I claim we can also get the same three basis functions as a ratio of triangle areas:**



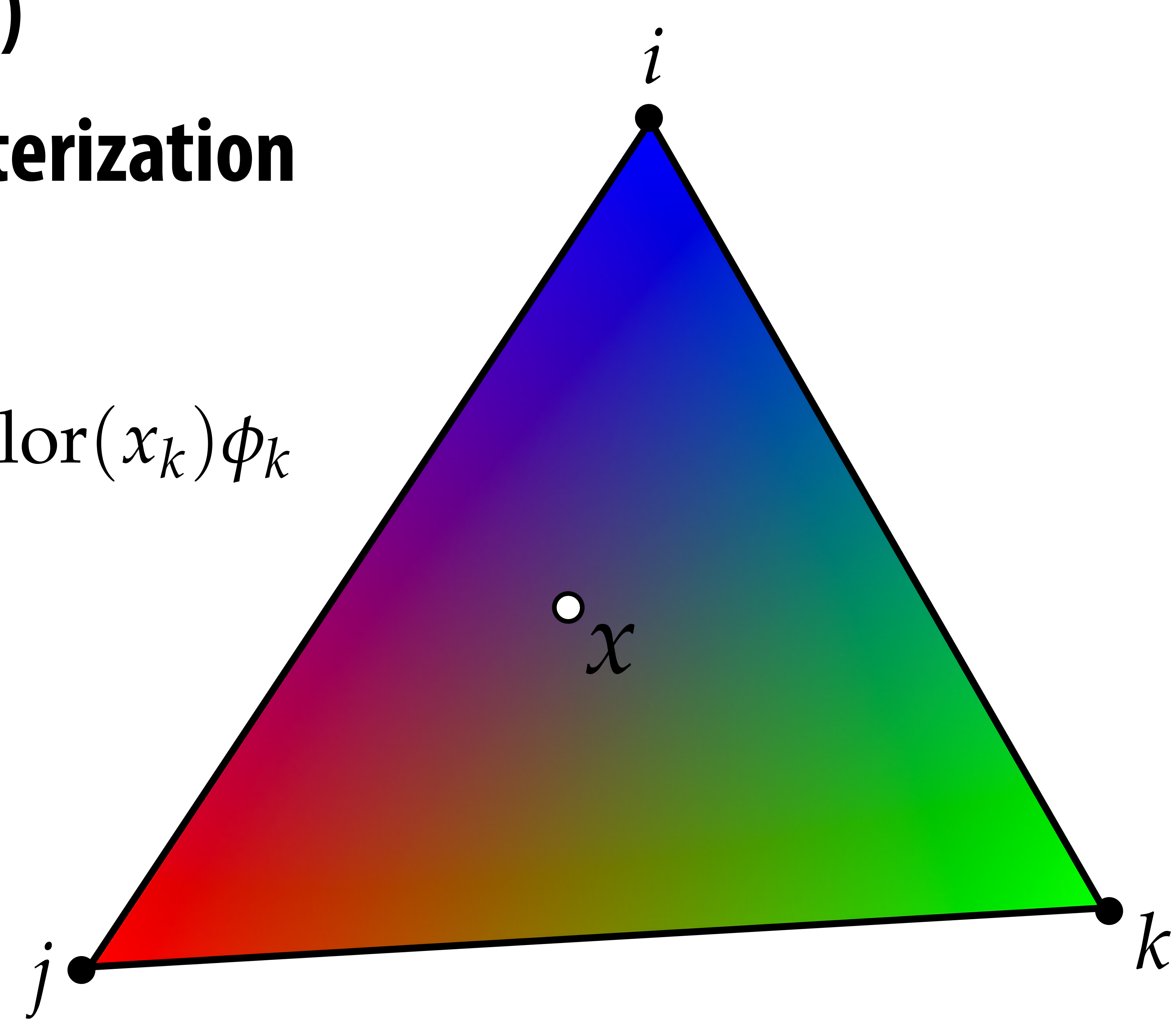
$$\phi_i(x) = \frac{\text{area}(x, x_j, x_k)}{\text{area}(x_i, x_j, x_k)}$$

Q: Do you buy it? (Why or why not?)

Barycentric Coordinates

- **No matter how you compute them, the values of the three functions $\phi_i(\mathbf{x}), \phi_j(\mathbf{x}), \phi_k(\mathbf{x})$ for a given point are called <u>barycentric coordinates</u>**
- **Can be used to interpolate any attribute associated with vertices. (color*, texture coordinates, etc.)**
- **Importantly, these same three values fall out of the half-plane tests used for triangle rasterization! (Why?)**
- **Hence, get them for "free" during rasterization**

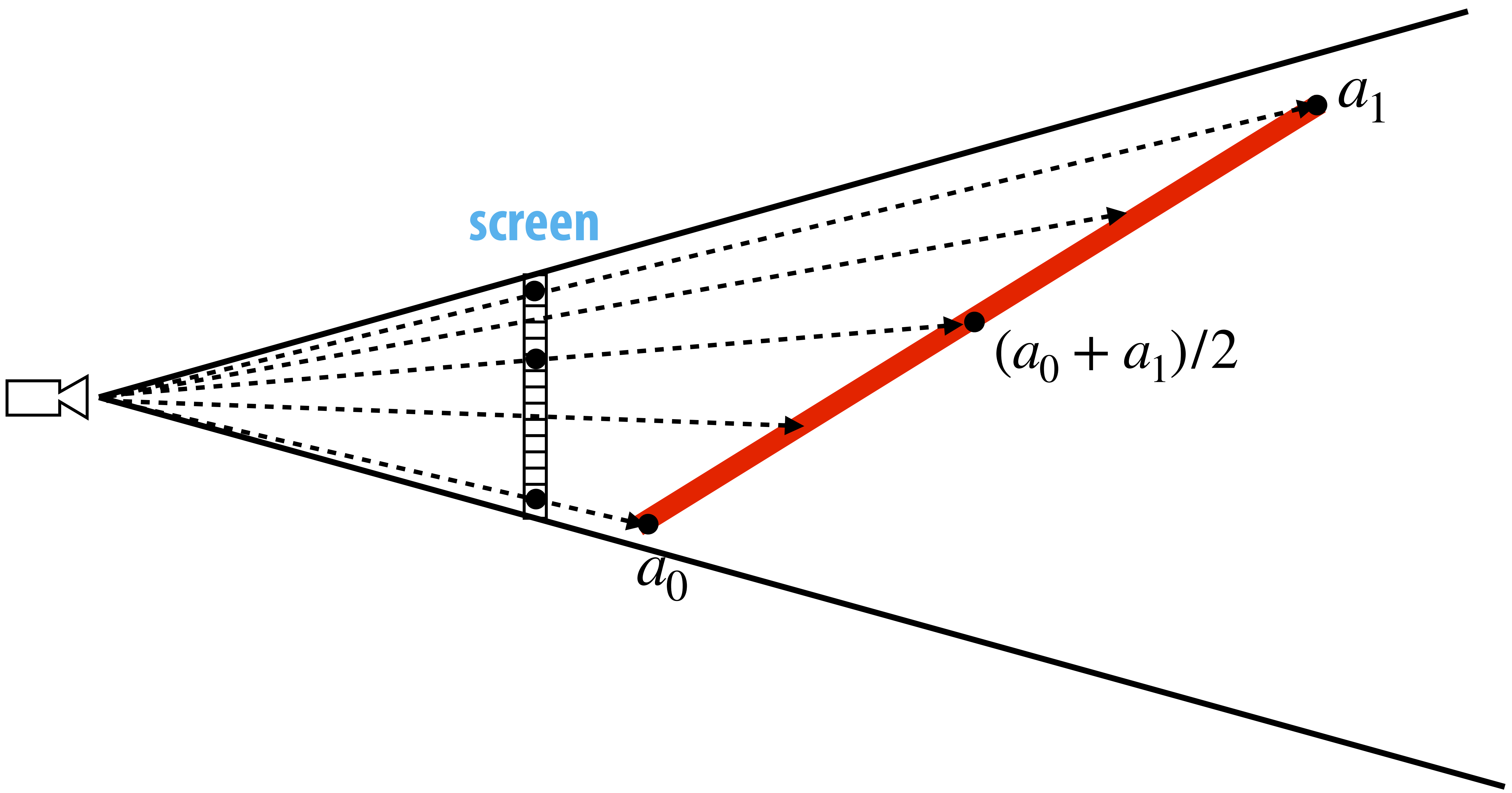
$$\text{color}(x) = \text{color}(x_i)\phi_i + \text{color}(x_j)\phi_j + \text{color}(x_k)\phi_k$$



*Note: we haven't explained yet how to encode colors as numbers! We'll talk about that in a later lecture…

Perspective-incorrect interpolation

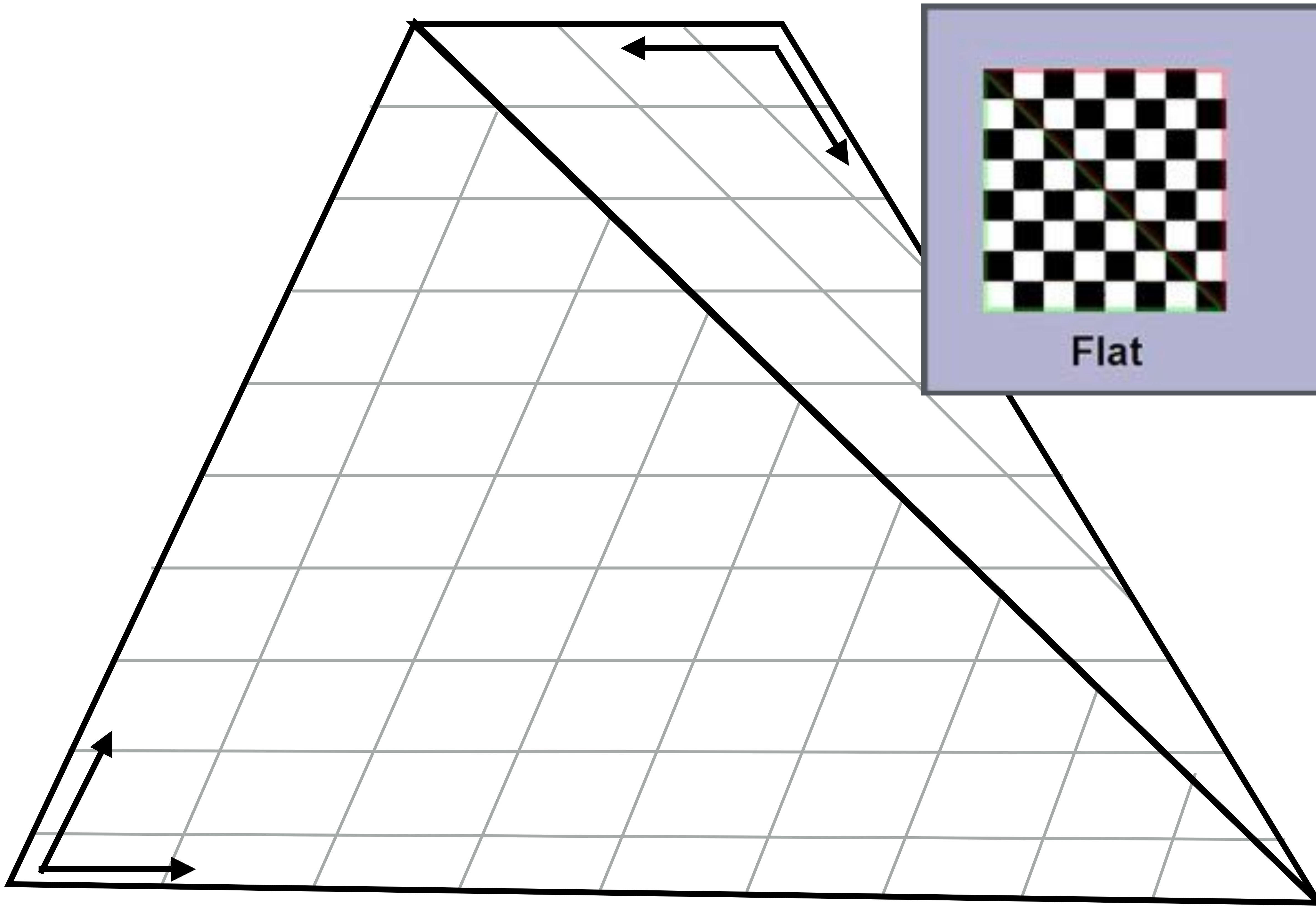
Due to perspective projection (homogeneous divide), barycentric interpolation of values on a triangle with different depths is not an affine function of screen XY coordinates

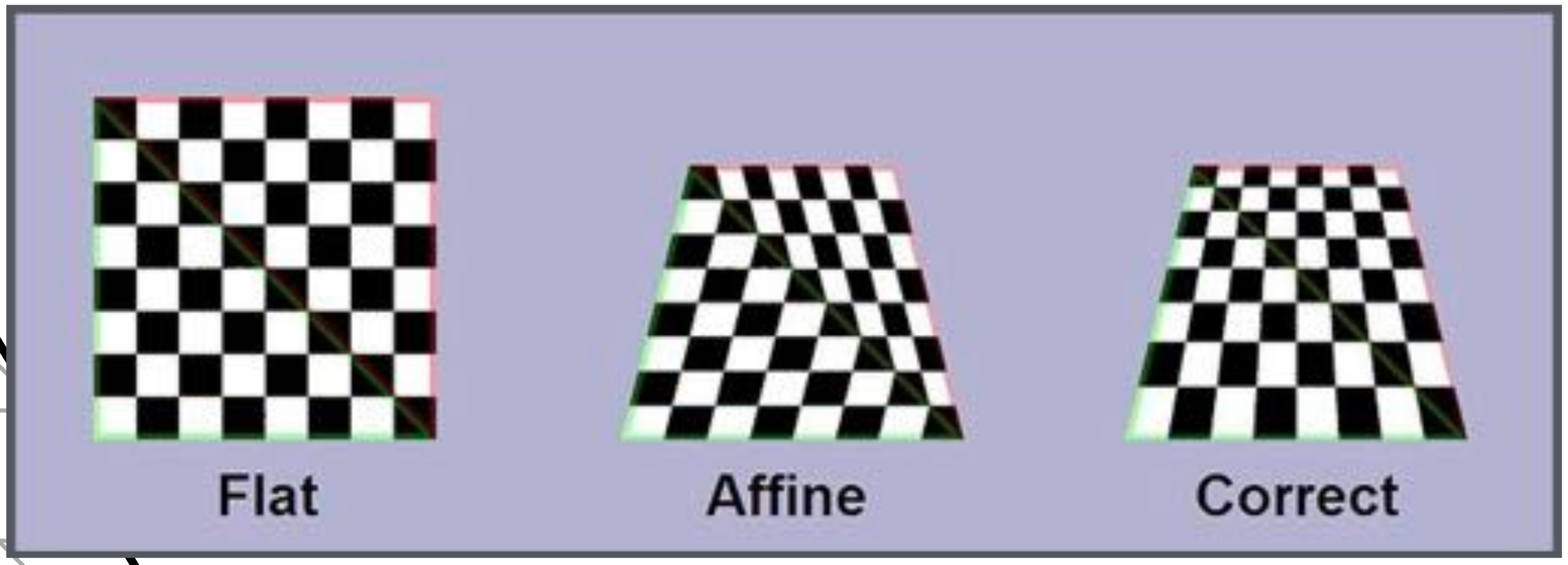


Want to interpolate attribute values linearly in 3D object space, not image space.

Example: perspective incorrect interpolation

Consider a quadrilateral split into two triangles:





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

Perspective Correct Interpolation

- Goal: interpolate some attribute ϕ at vertices
- Basic recipe:
  - Compute depth z at each vertex
  - Evaluate Z := 1/z and P := 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

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

Texture Mapping

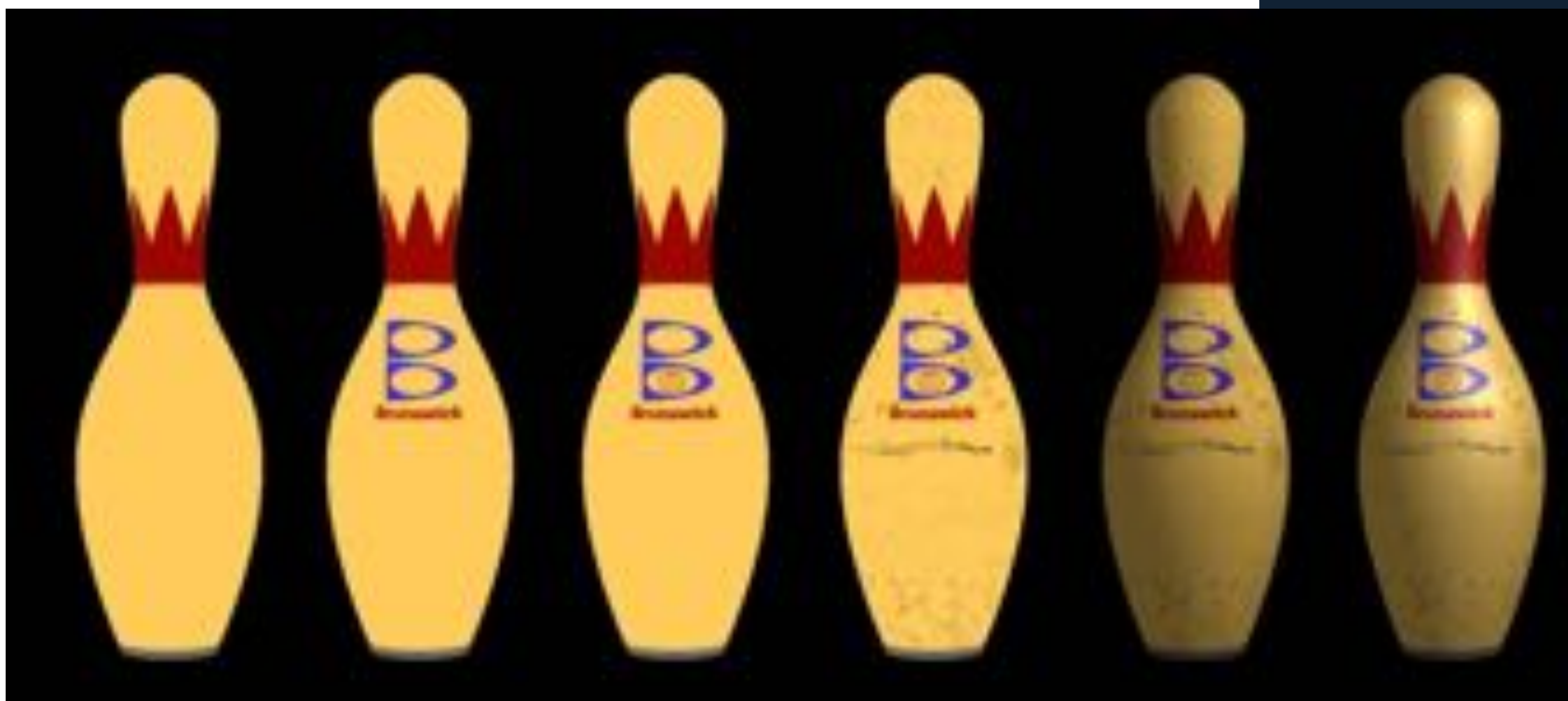
Many uses of texture mapping

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.

Normal & Displacement Mapping

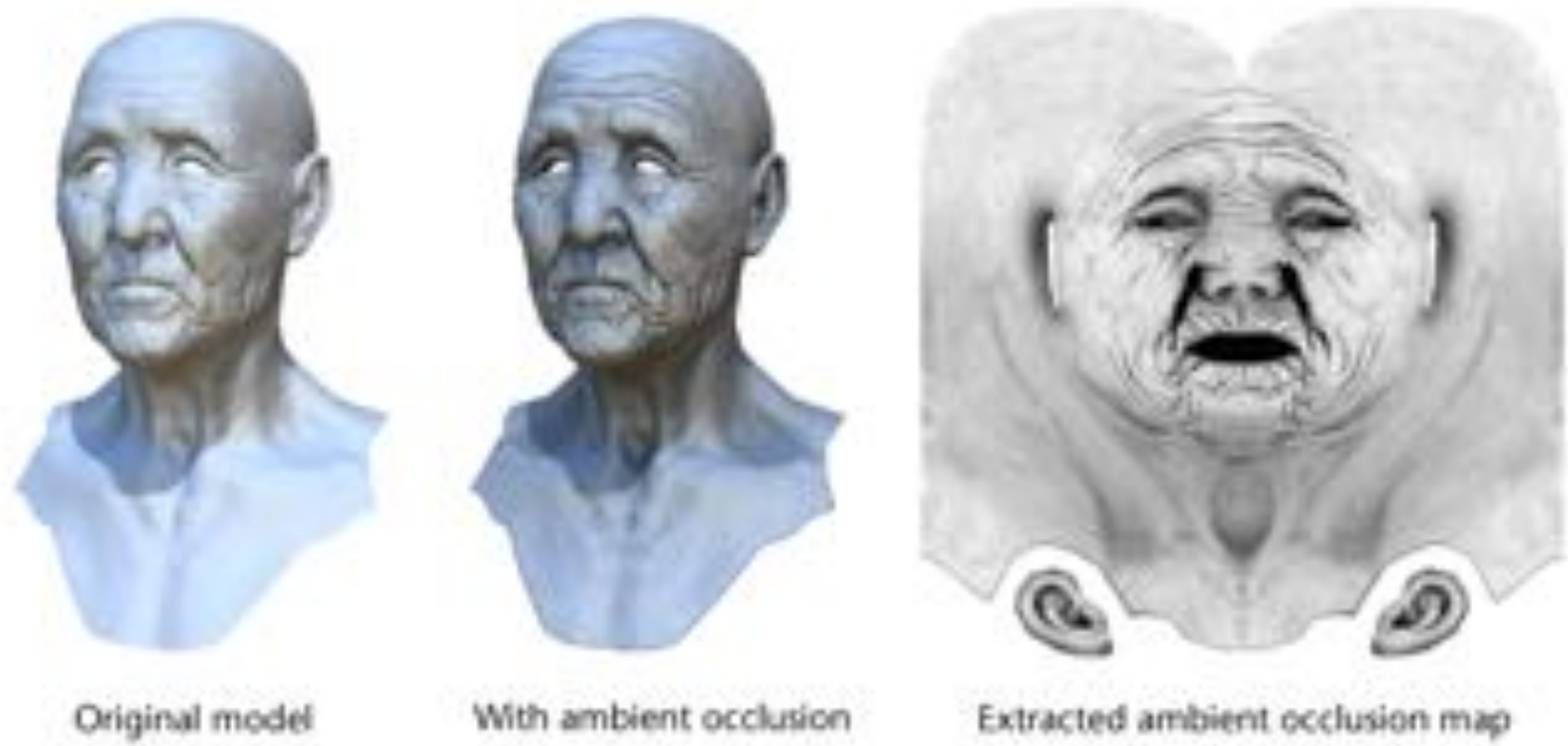
normal mapping

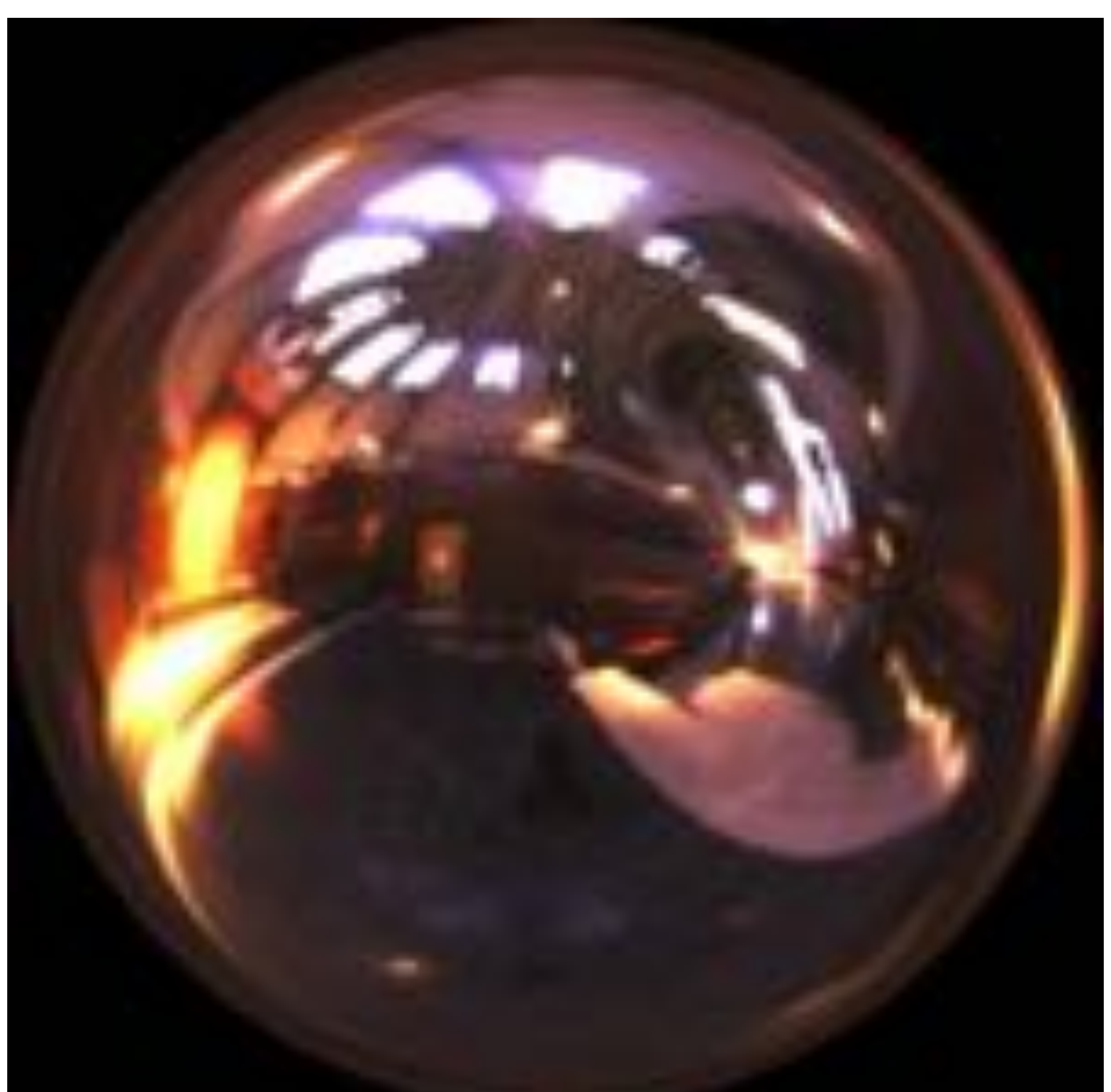
Use texture value to perturb surface normal to
"fake" appearance of a bumpy surface

displacement mapping

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

Represent precomputed lighting and shadows



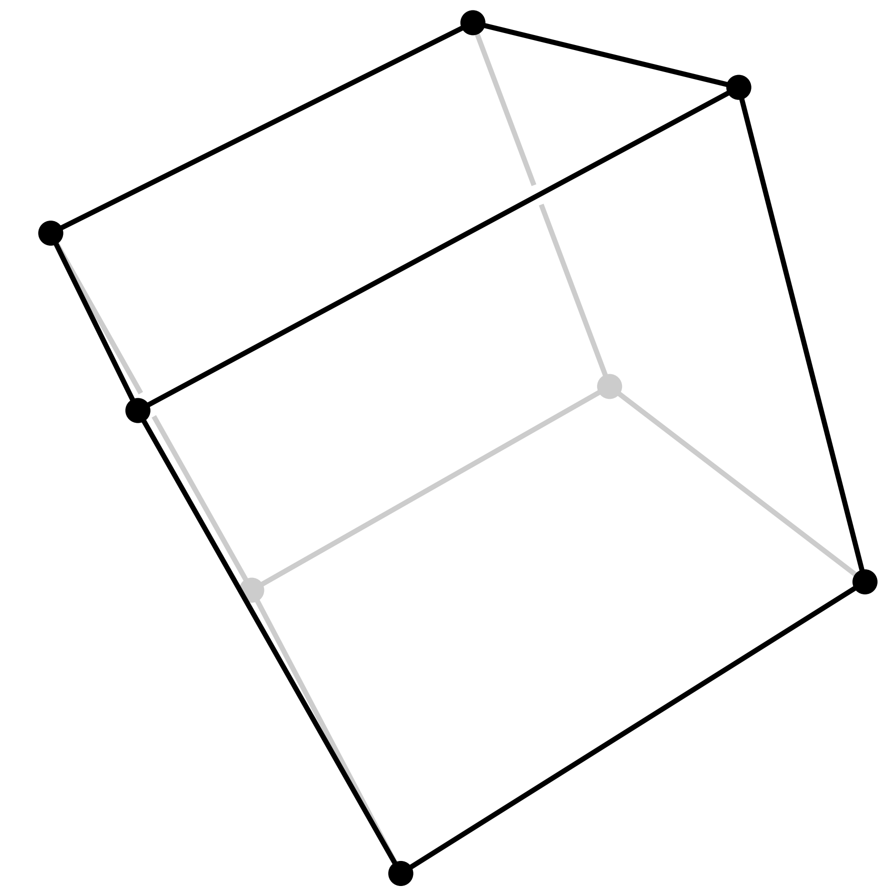


Grace Cathedral environment map

Environment map used in rendering

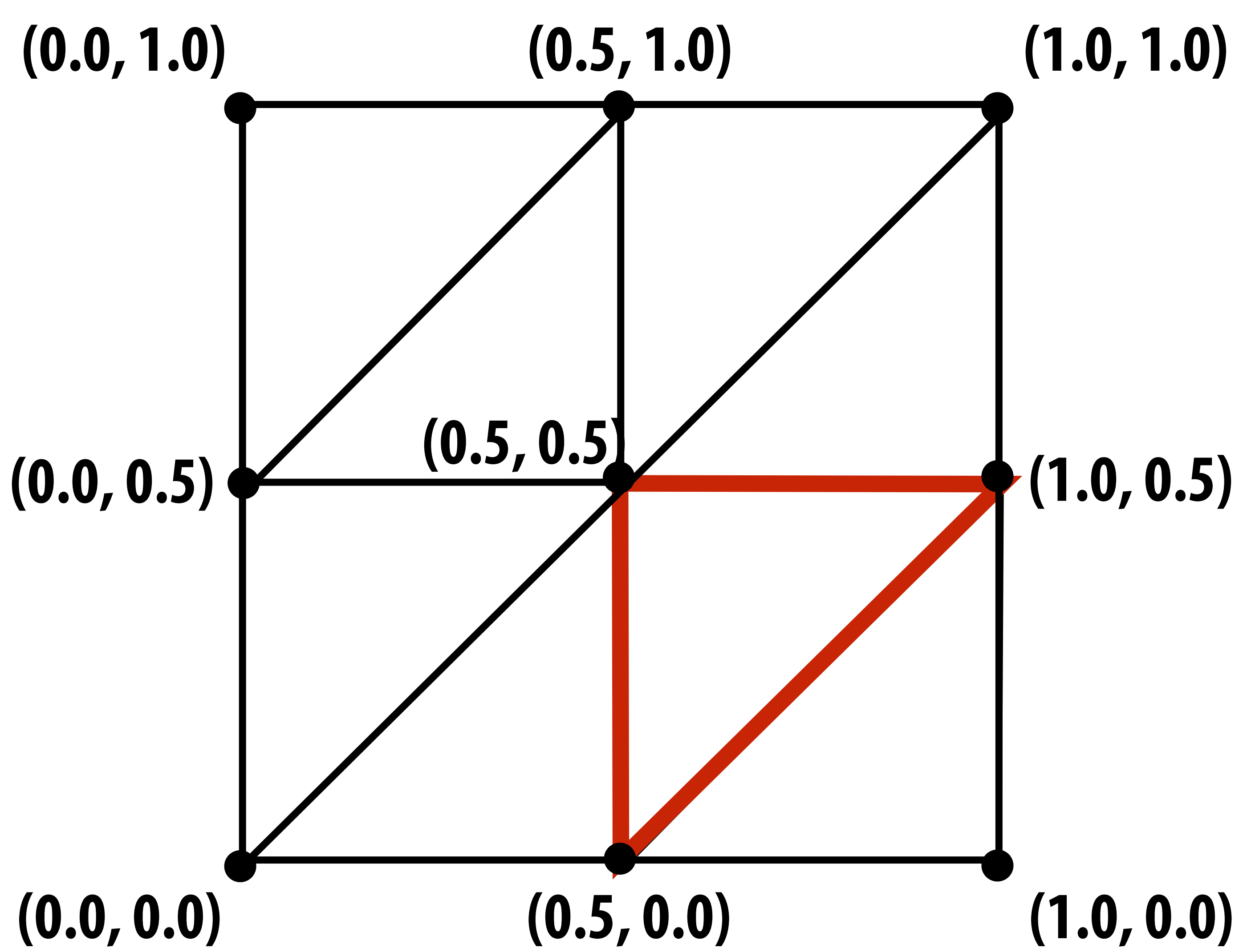
Texture coordinates

example: texture this cube

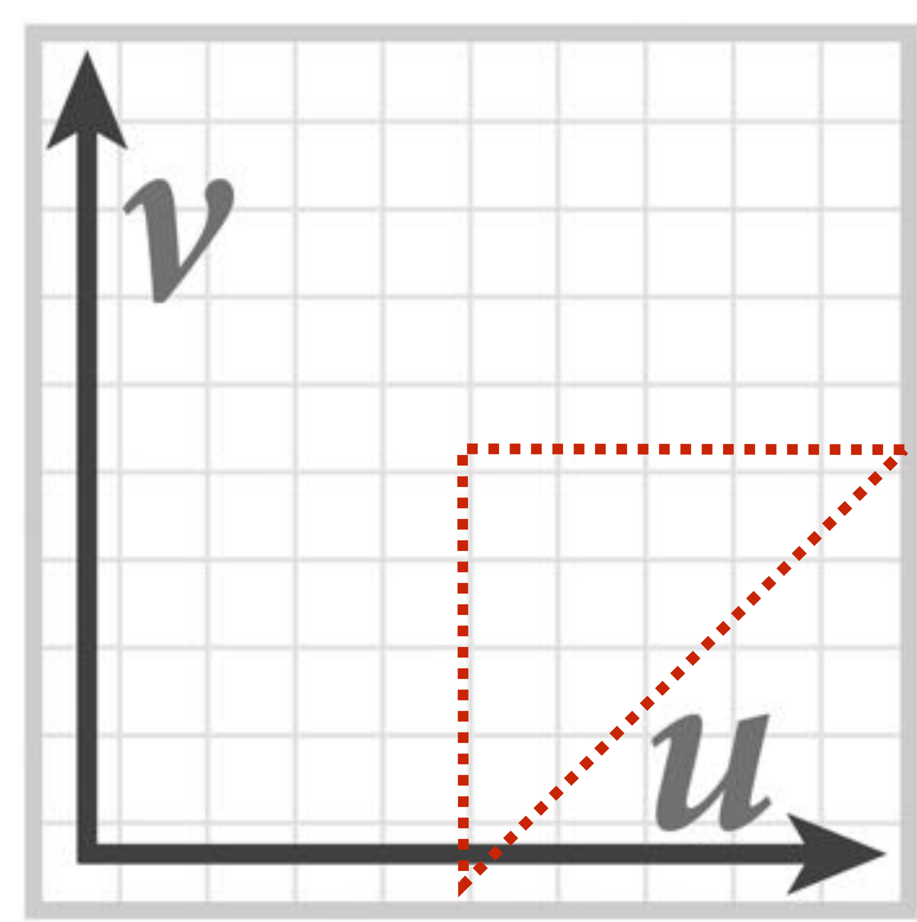


- "Texture coordinates" define a mapping from surface coordinates to points in texture domain
- Often defined by linearly interpolating texture coordinates at triangle vertices

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

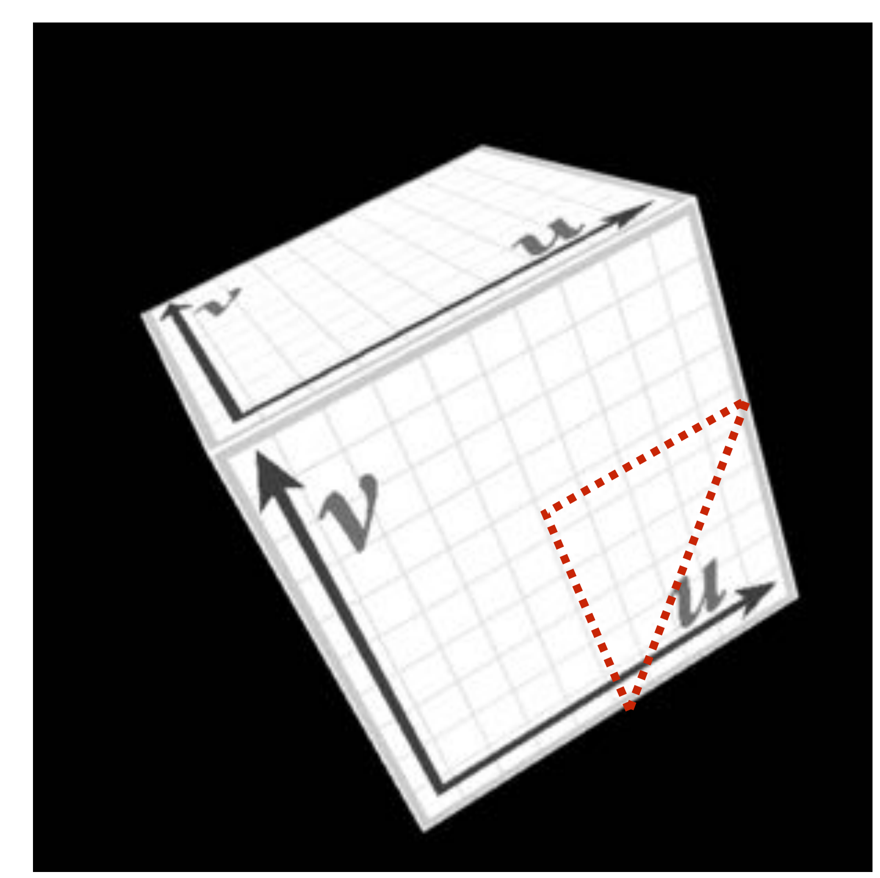


A texture on the $[0,1]^2$ domain can be specified by a 2048x2048 image



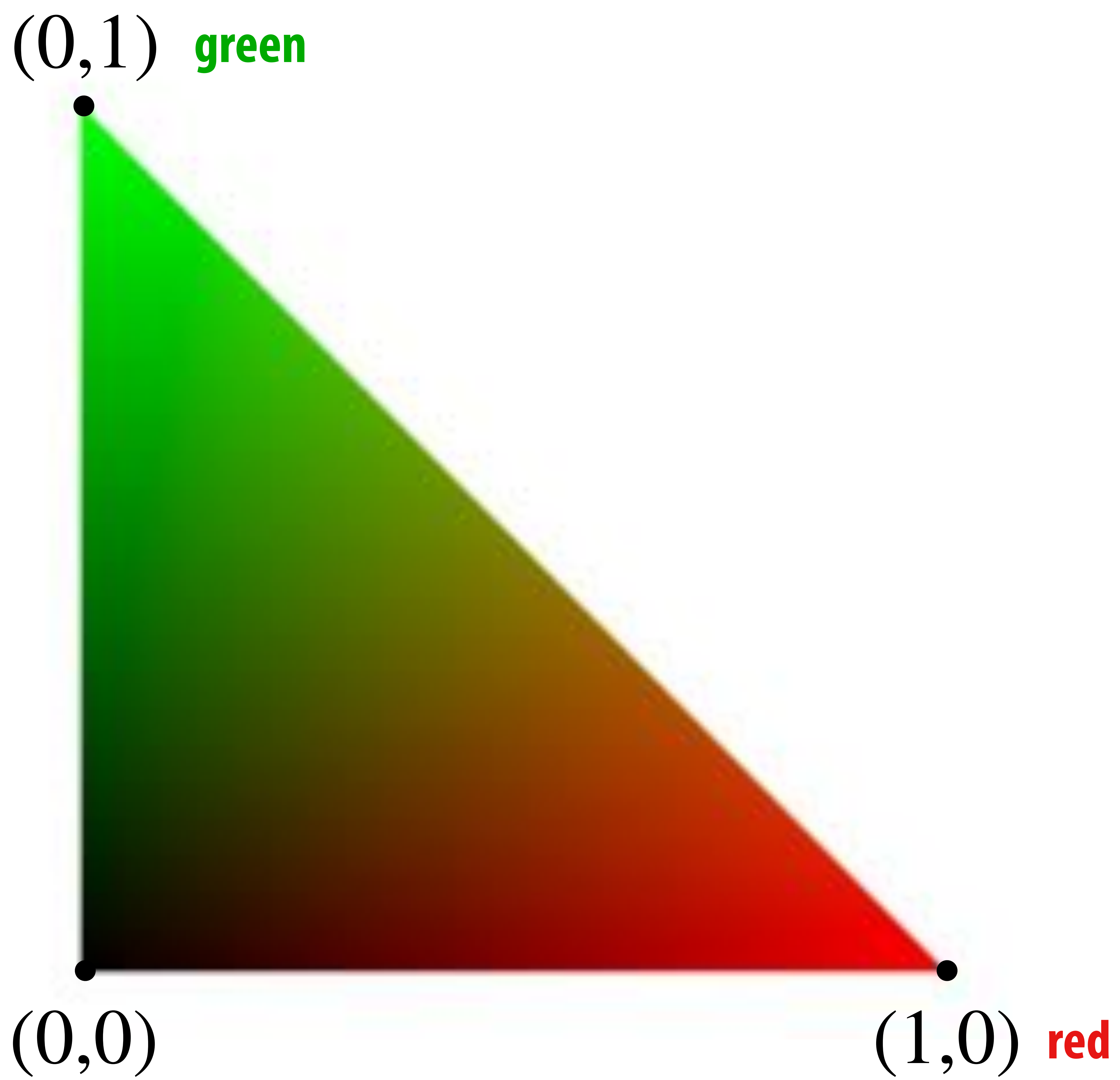
(location of highlighted triangle in texture space shown in red)

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



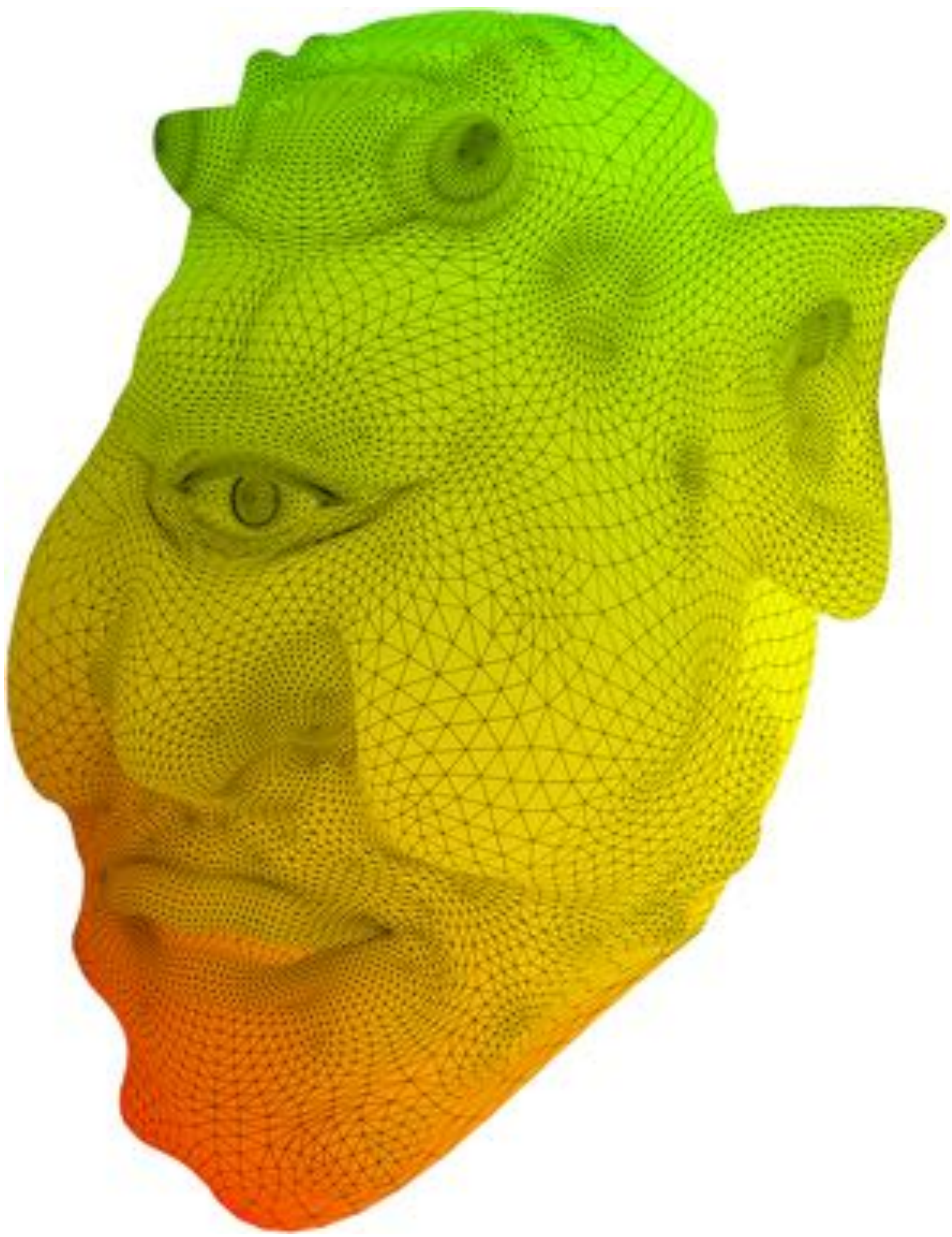
Visualization of texture coordinates

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

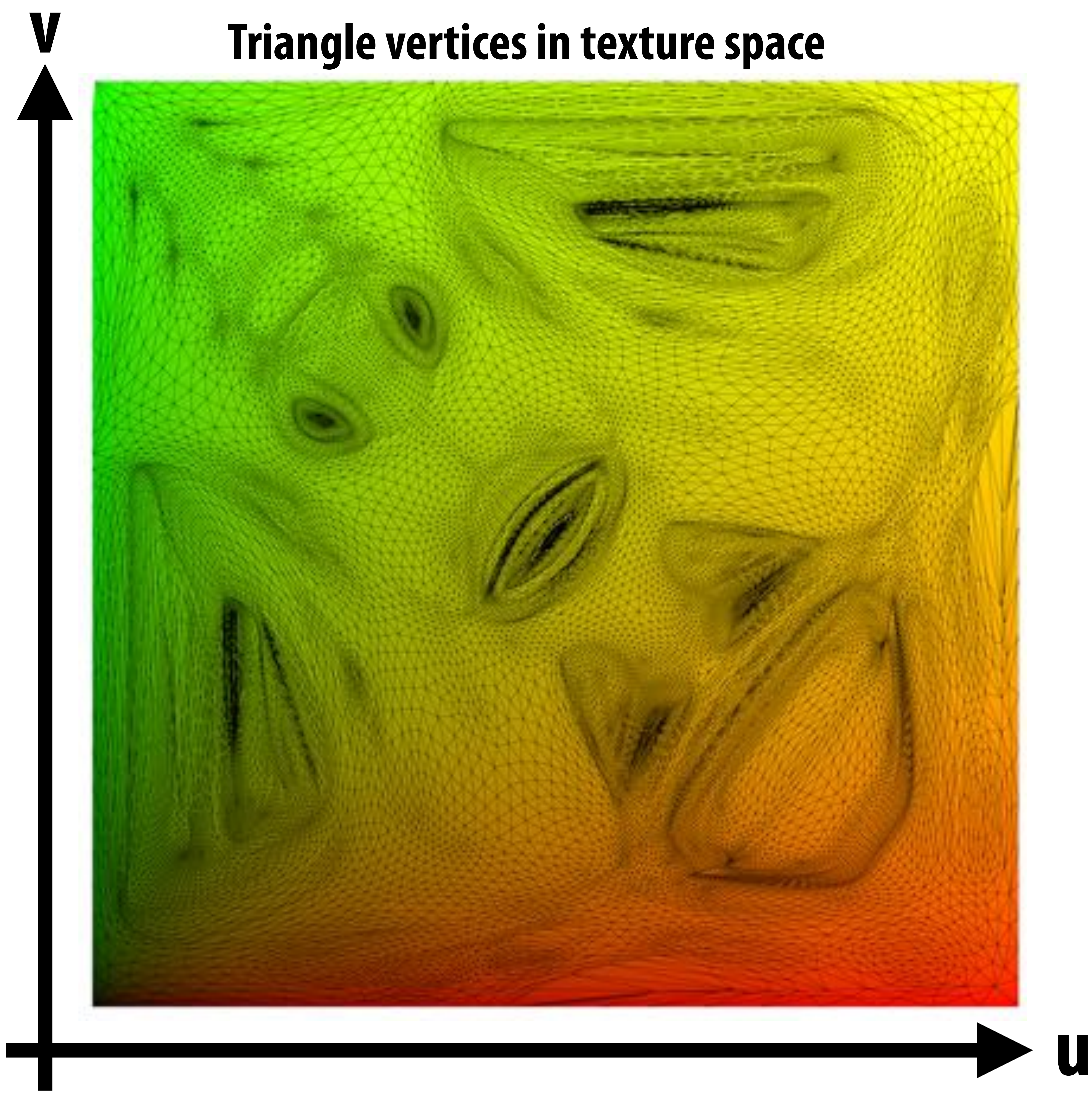


More complex mapping

Visualization of texture coordinates



Triangle vertices in texture space

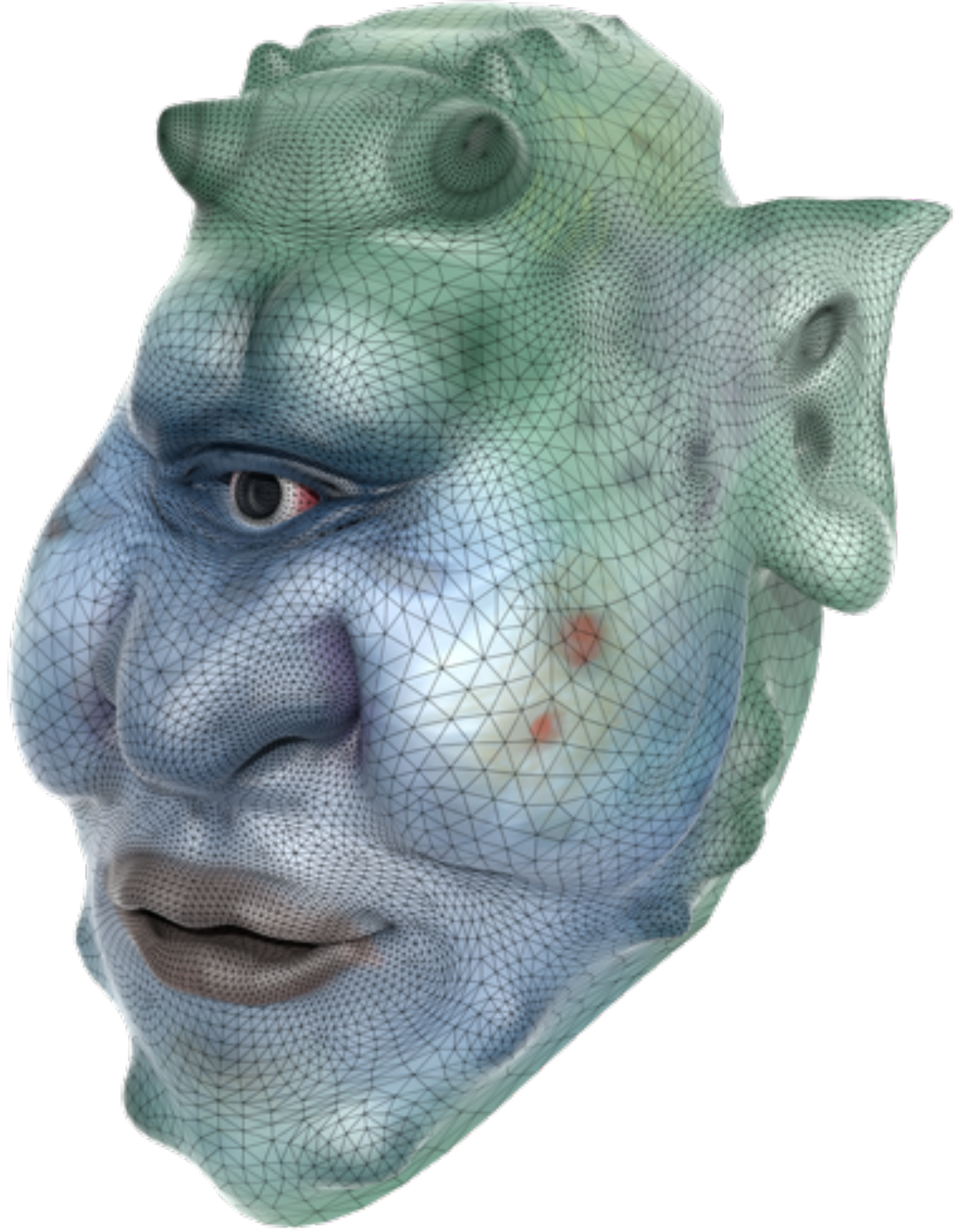


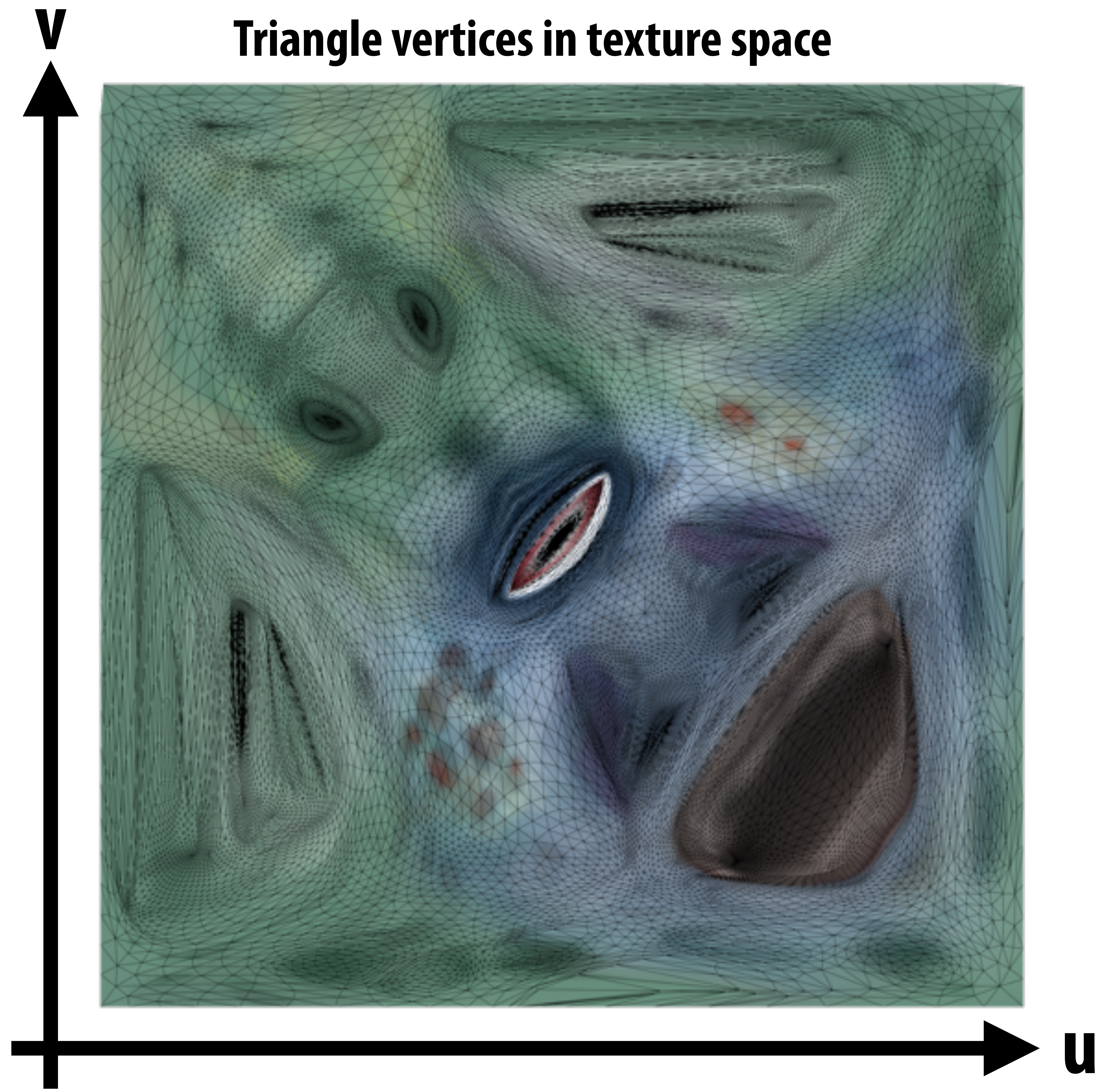
Each vertex has a coordinate (u,v) in texture space

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

Texture mapping adds detail

Rendered result

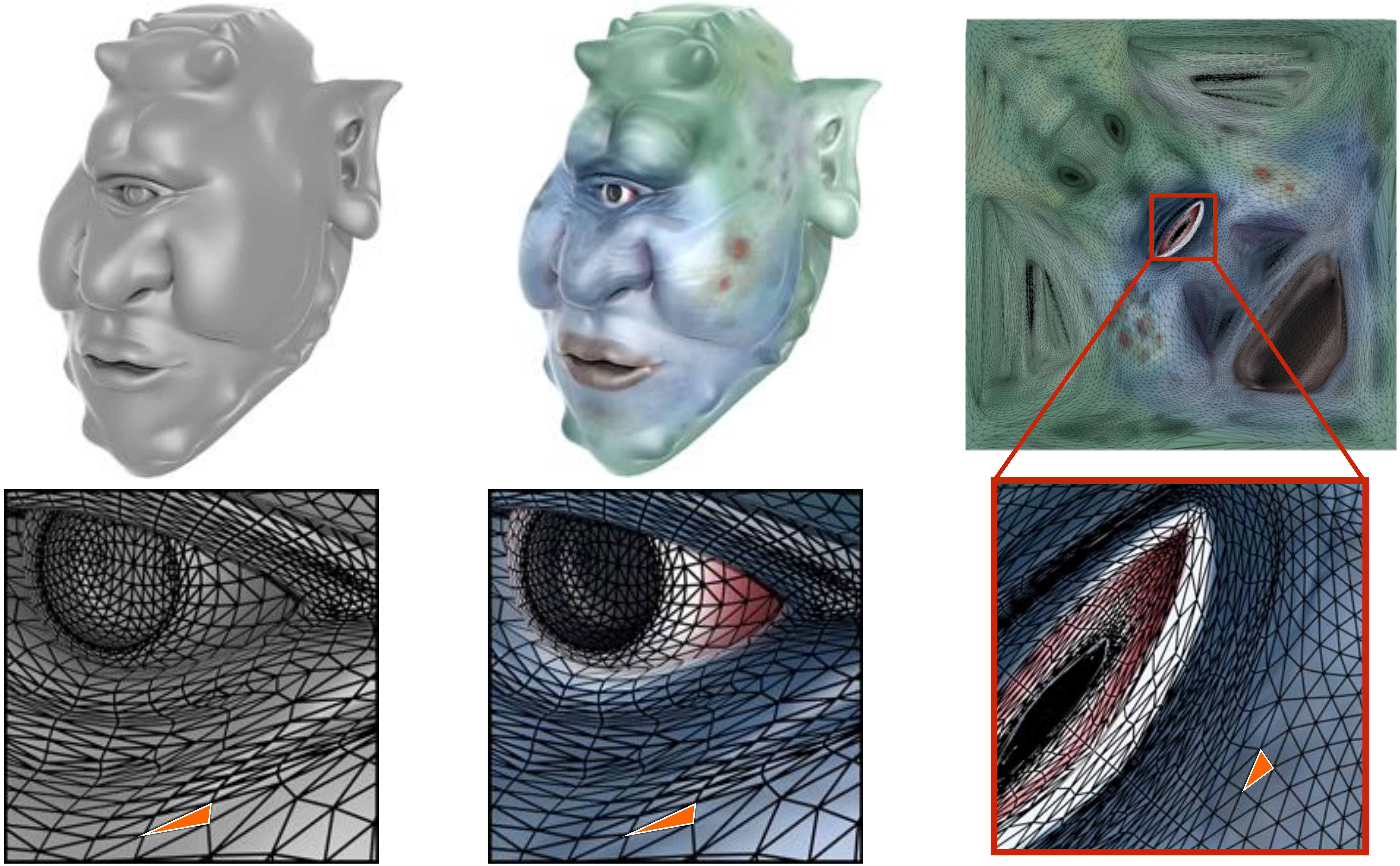




Each triangle "copies" a piece of the image back to the surface

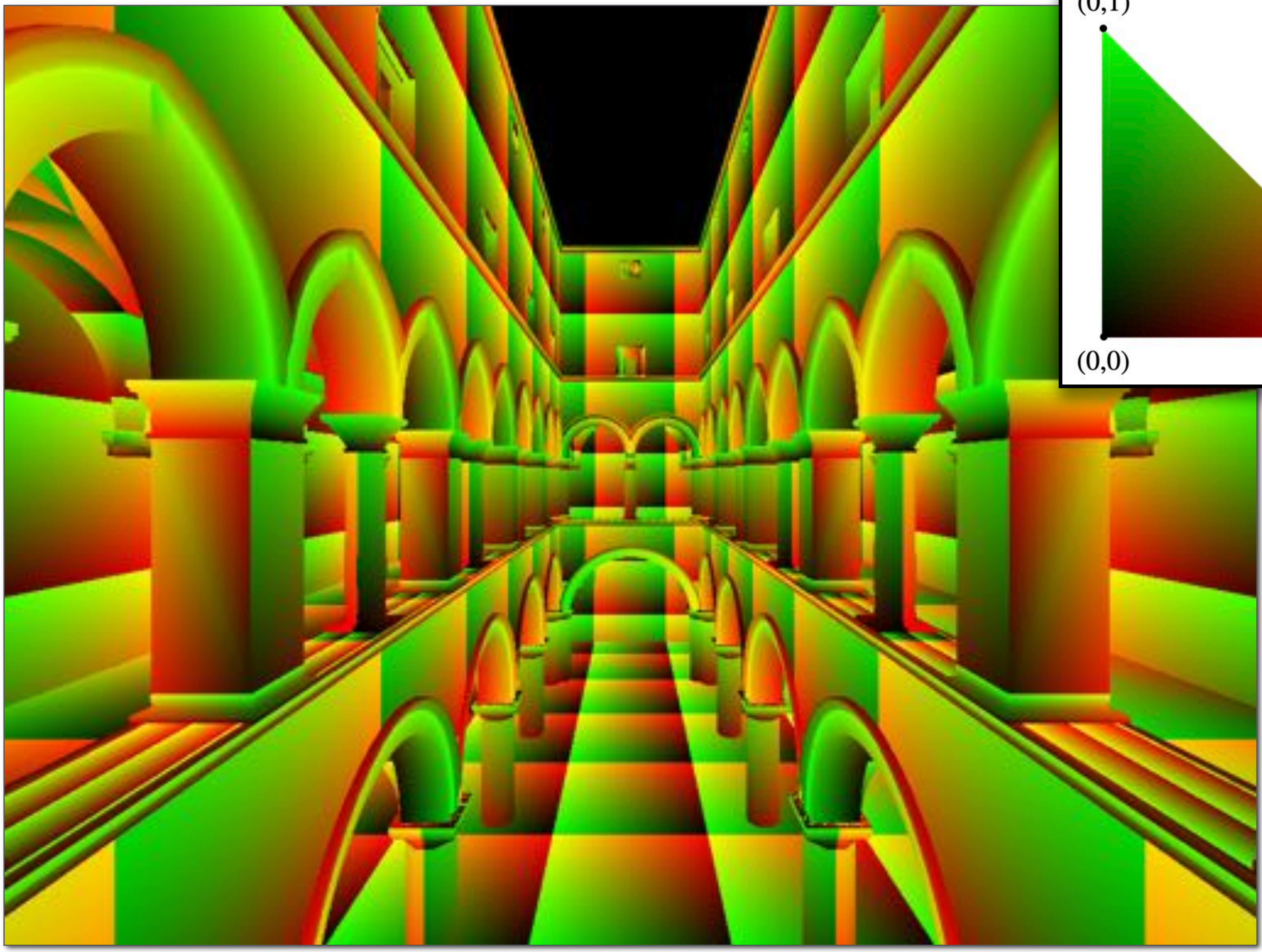
Texture mapping adds detail

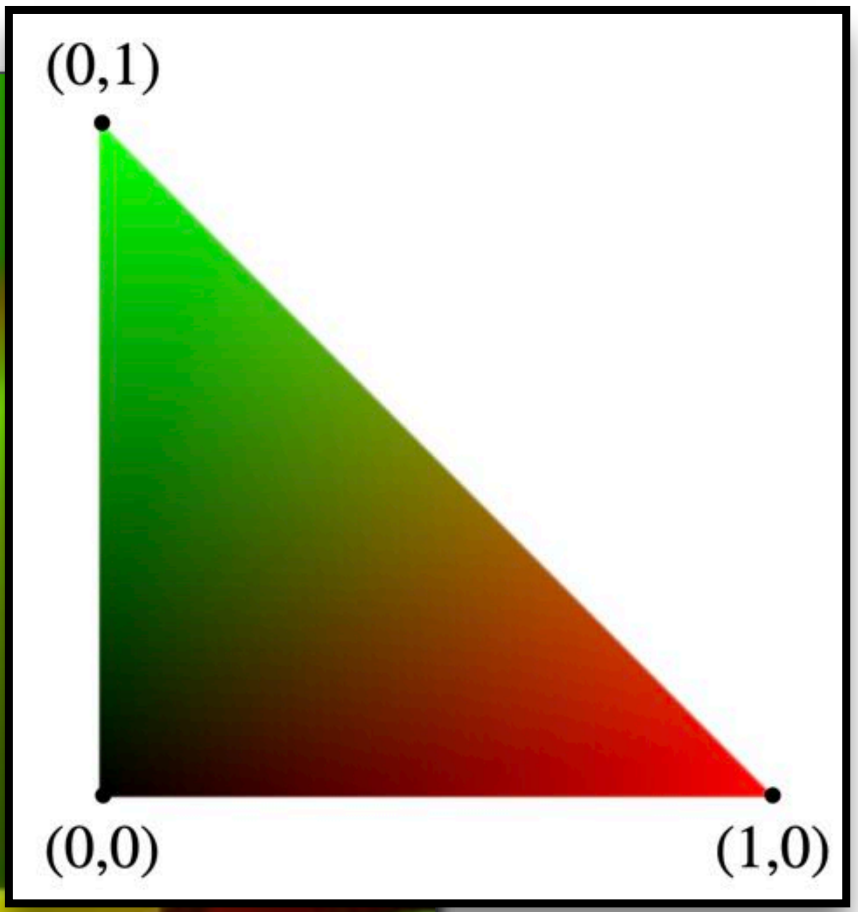
rendering without texture | **rendering with texture** | **texture image**



zoom

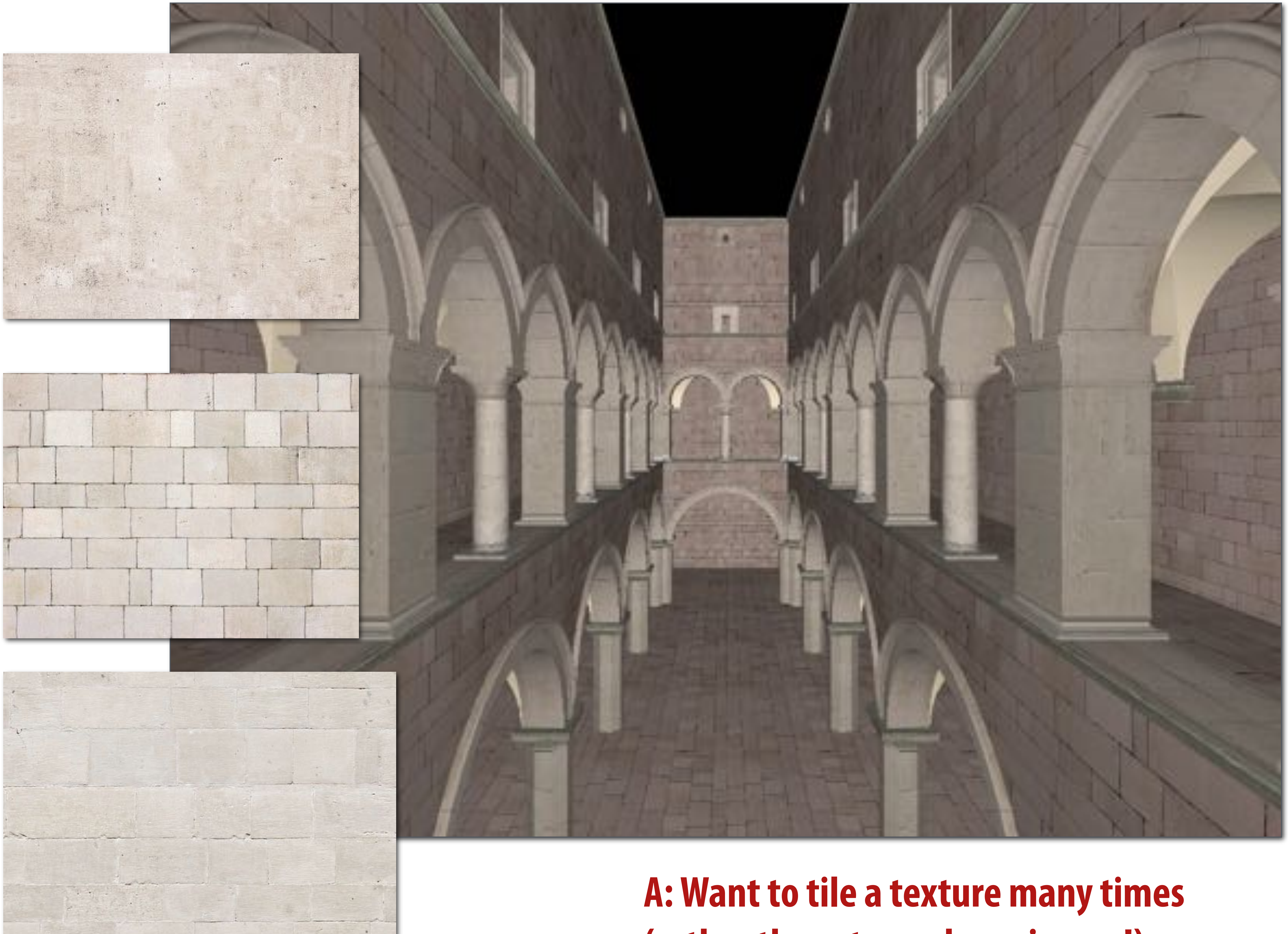
Another example: periodic coordinates





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

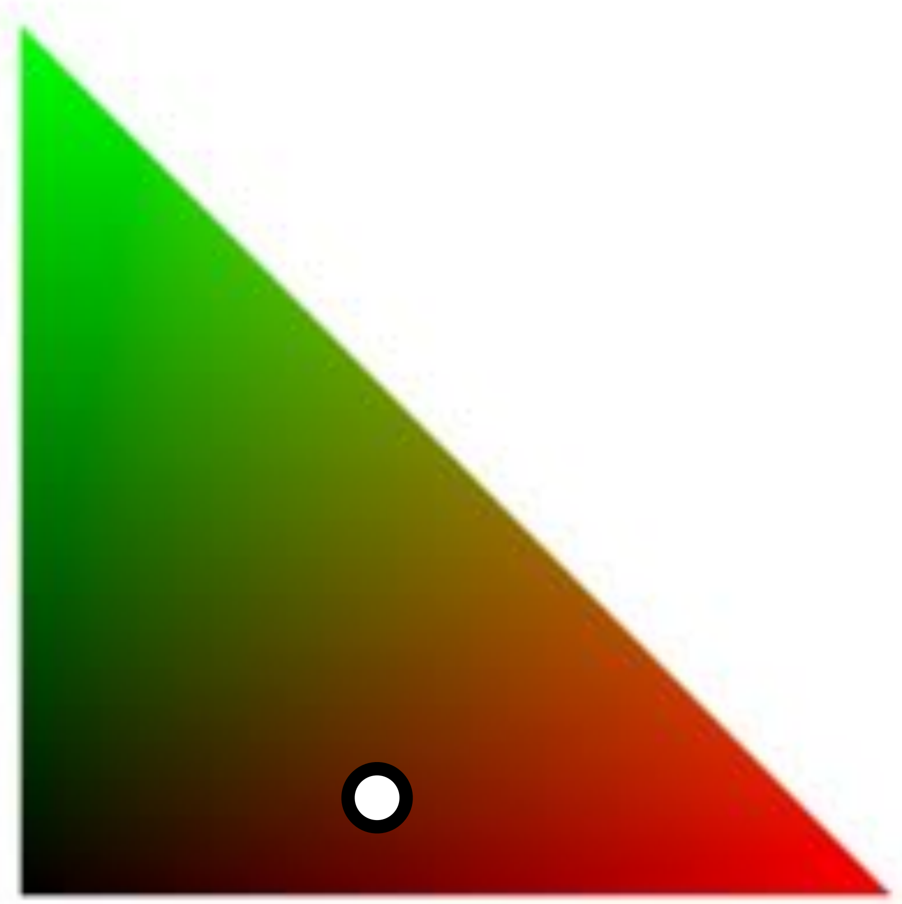
Textured Sponza

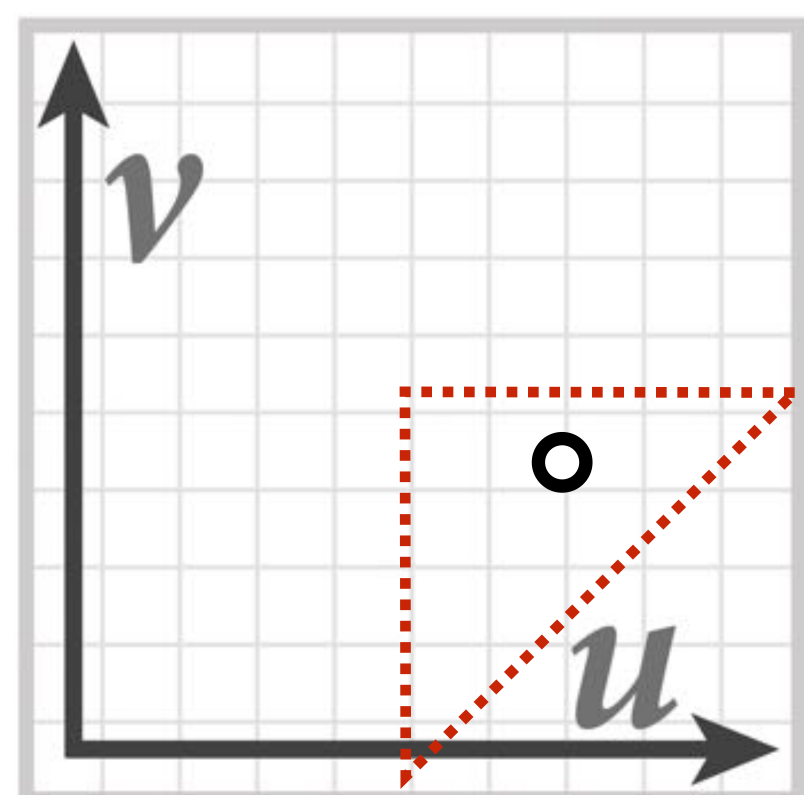


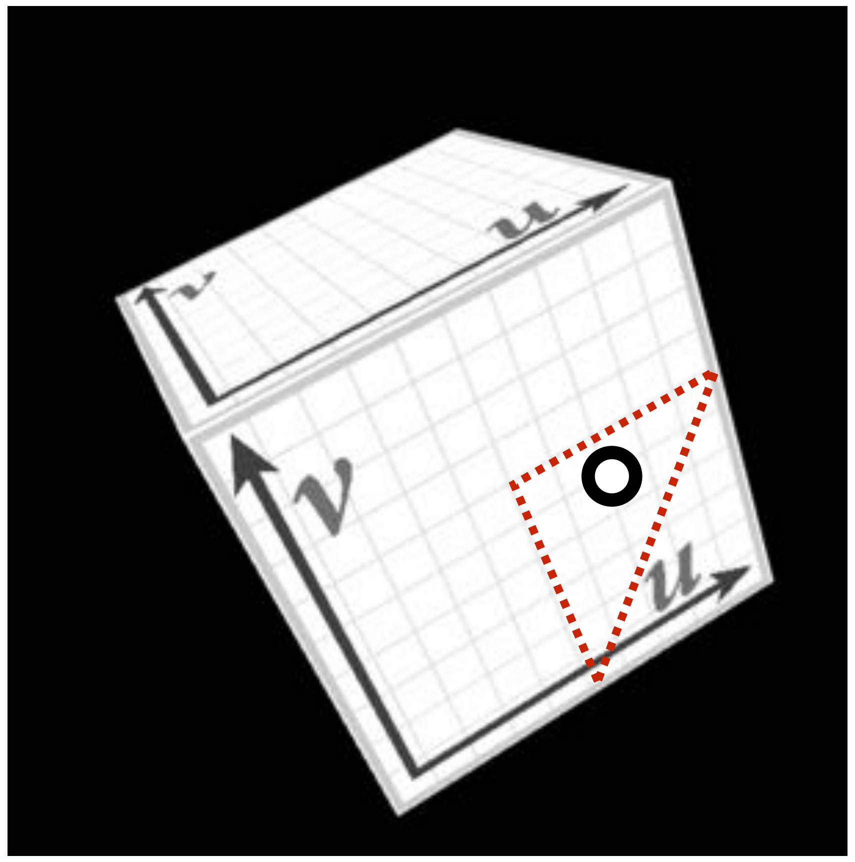
A: Want to tile a texture many times (rather than store a huge image!)

Texture Sampling 101

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



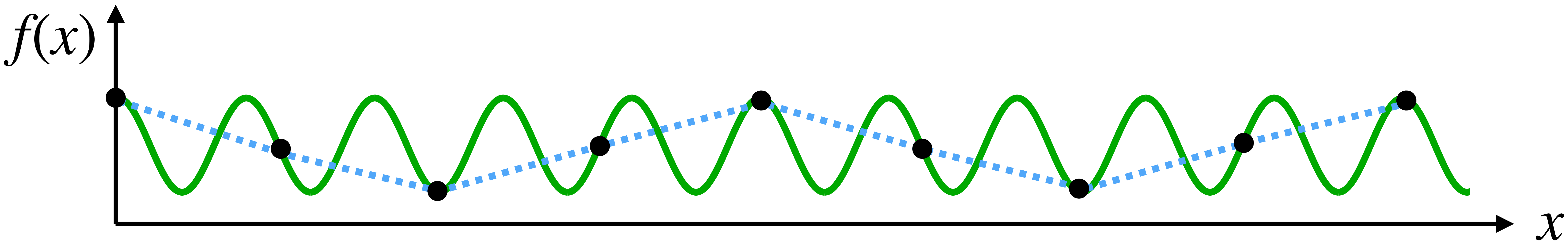


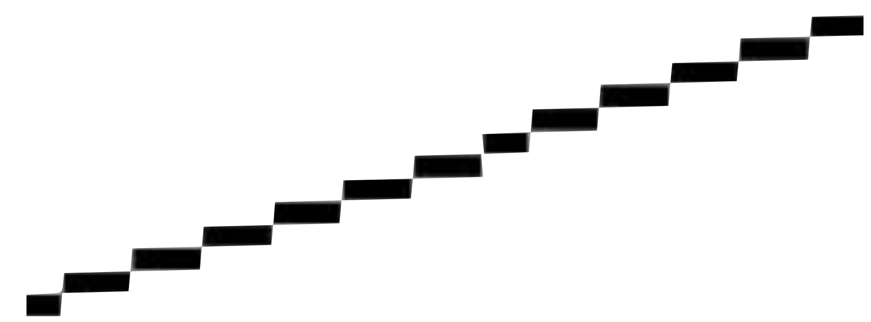


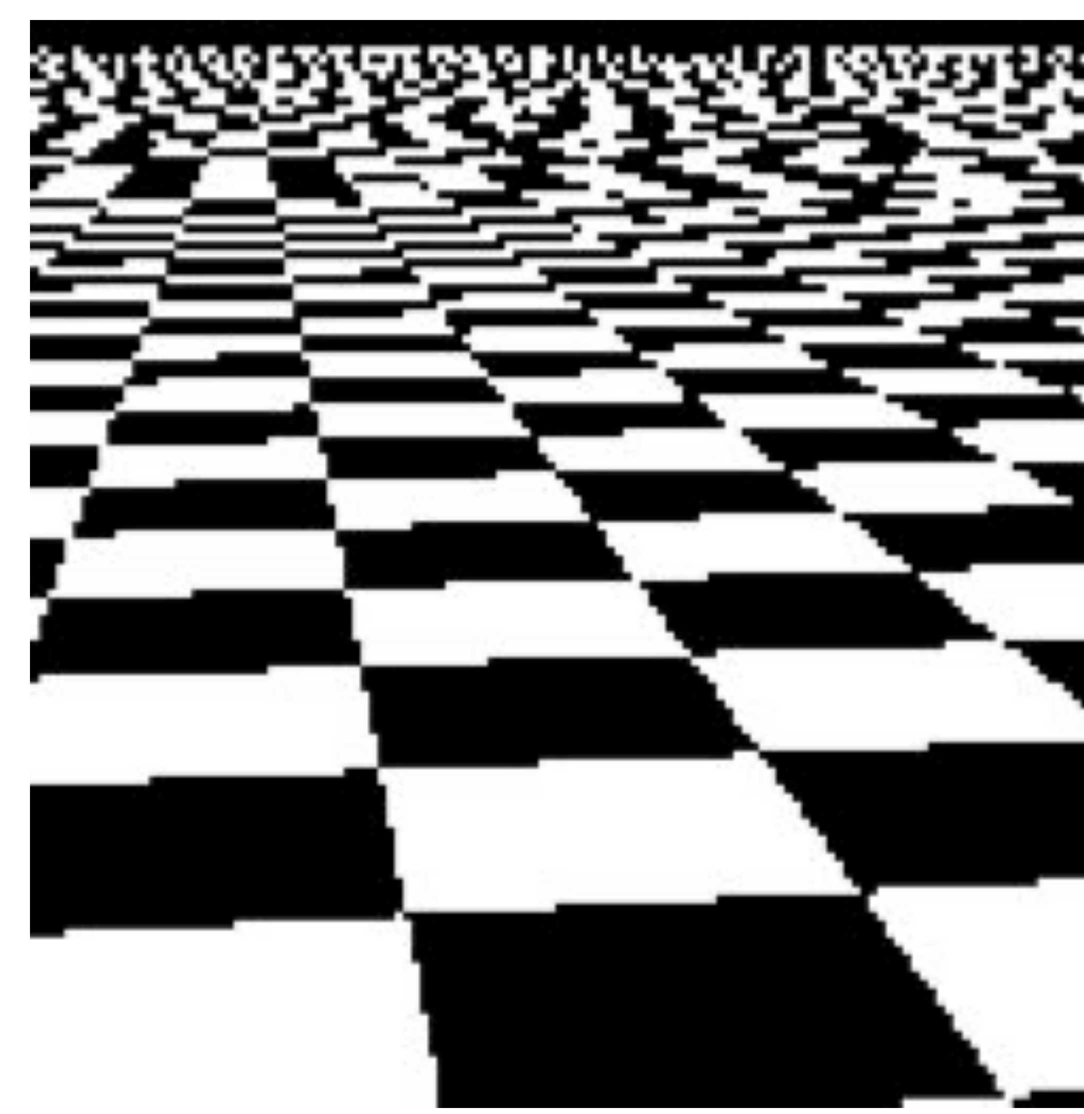
...sadly not this easy in general!

Recall: aliasing

Undersampling a high-frequency signal can result in aliasing



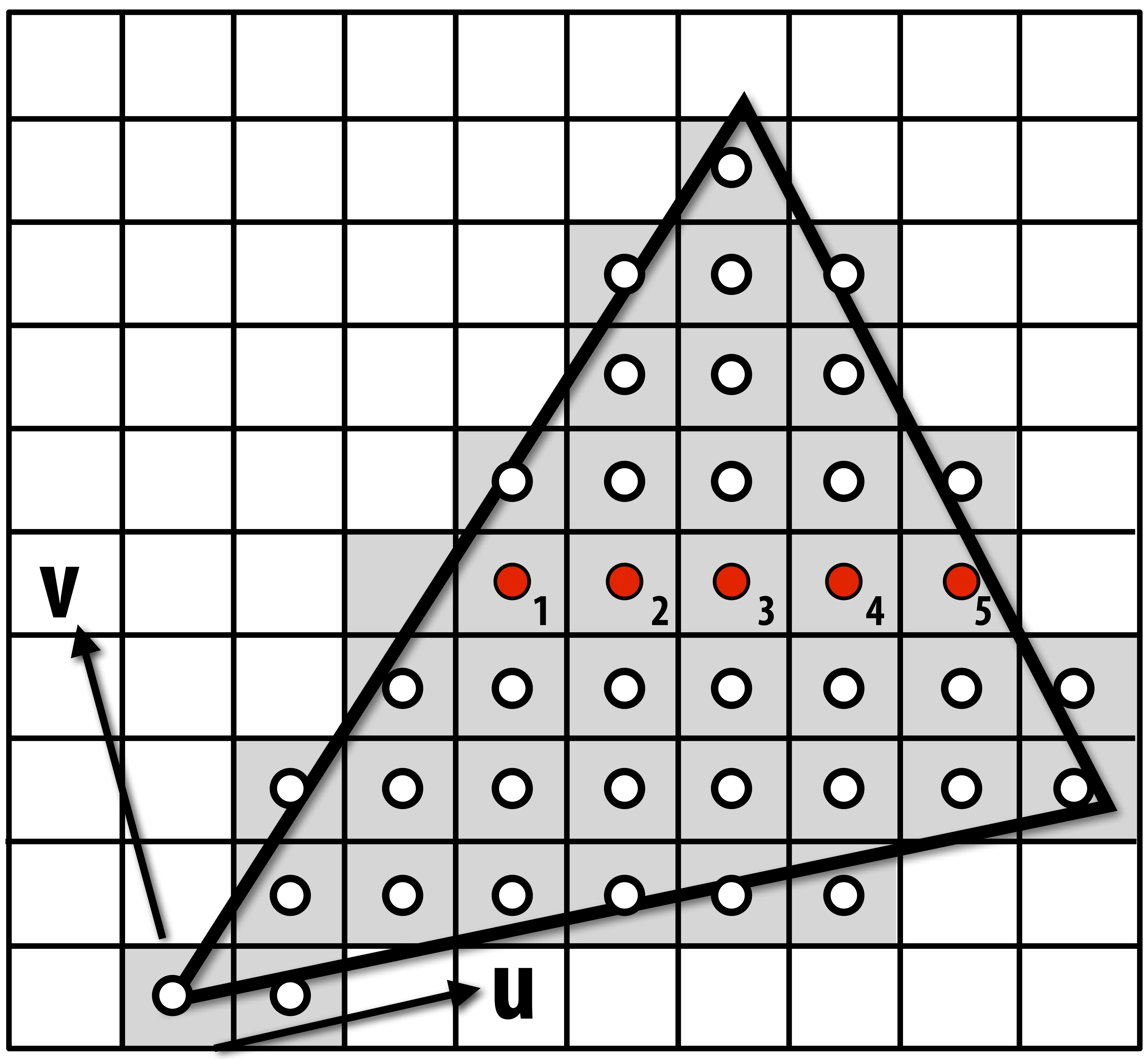




Visualizing texture samples

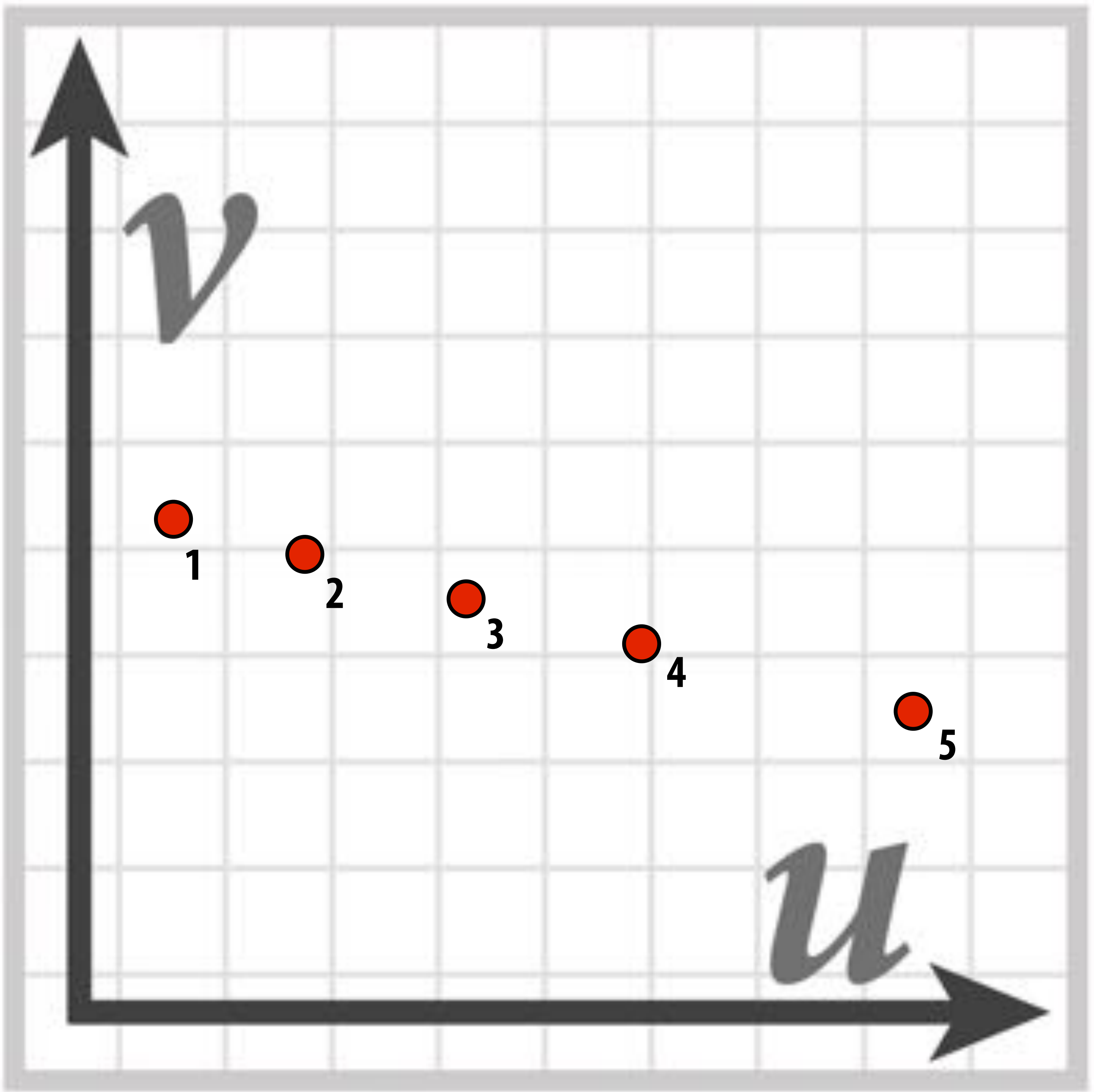
Since triangles are projected from 3D to 2D, pixels in screen space will correspond to regions of varying size & location in texture

sample positions in screen space



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

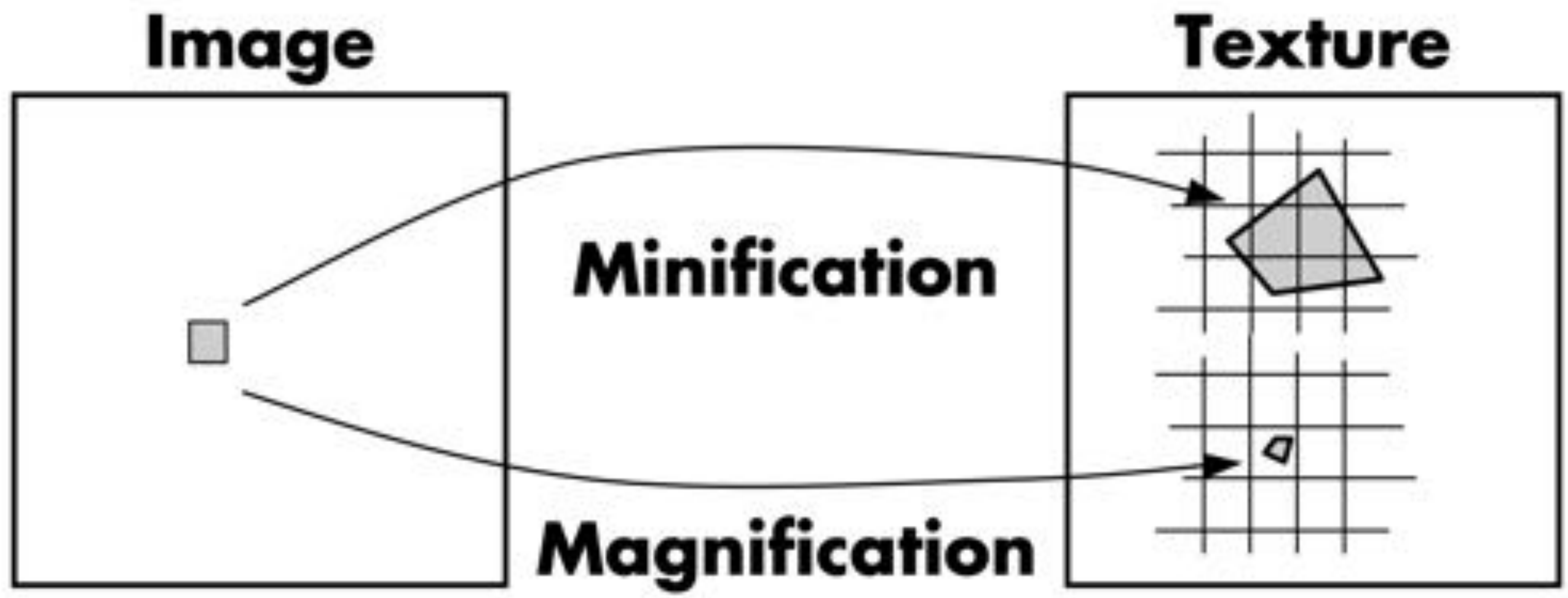
sample positions in texture space



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

Irregular sampling pattern makes it hard to avoid aliasing!

Magnification vs. Minification

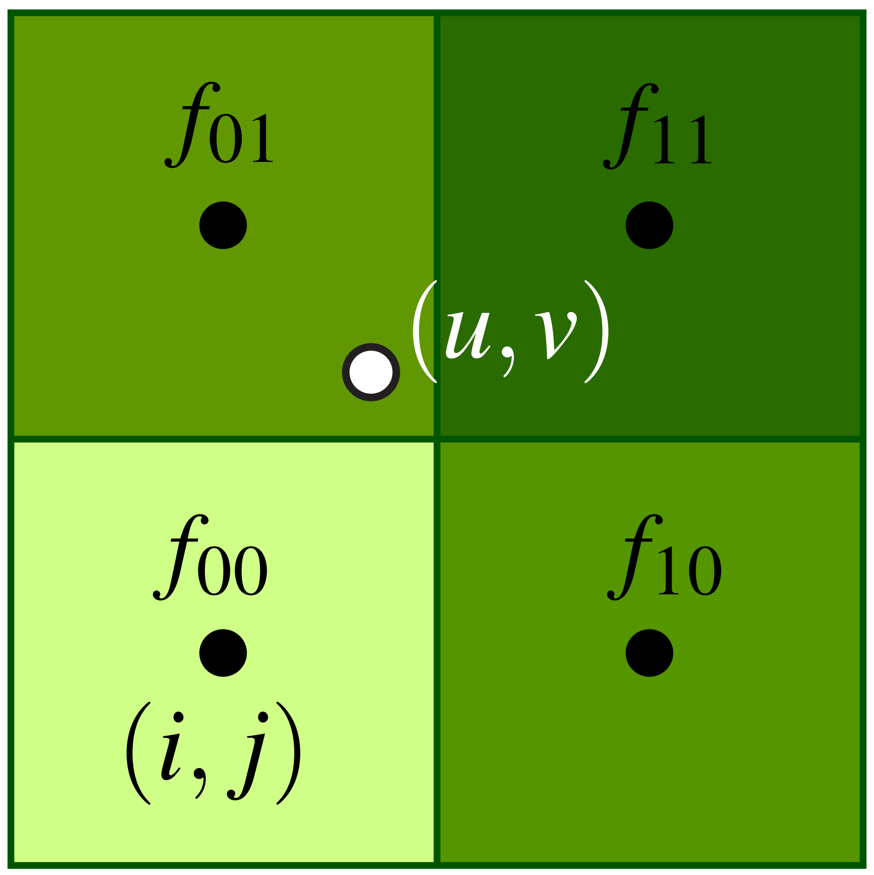


- **Magnification (easier):**
  - Example: camera is <u>very</u> close to scene object
  - Single screen pixel maps to tiny region of texture
  - Can just interpolate value at screen pixel center
- **Minification (harder):**
  - Example: scene object is <u>very</u> far away
  - Single screen pixel maps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

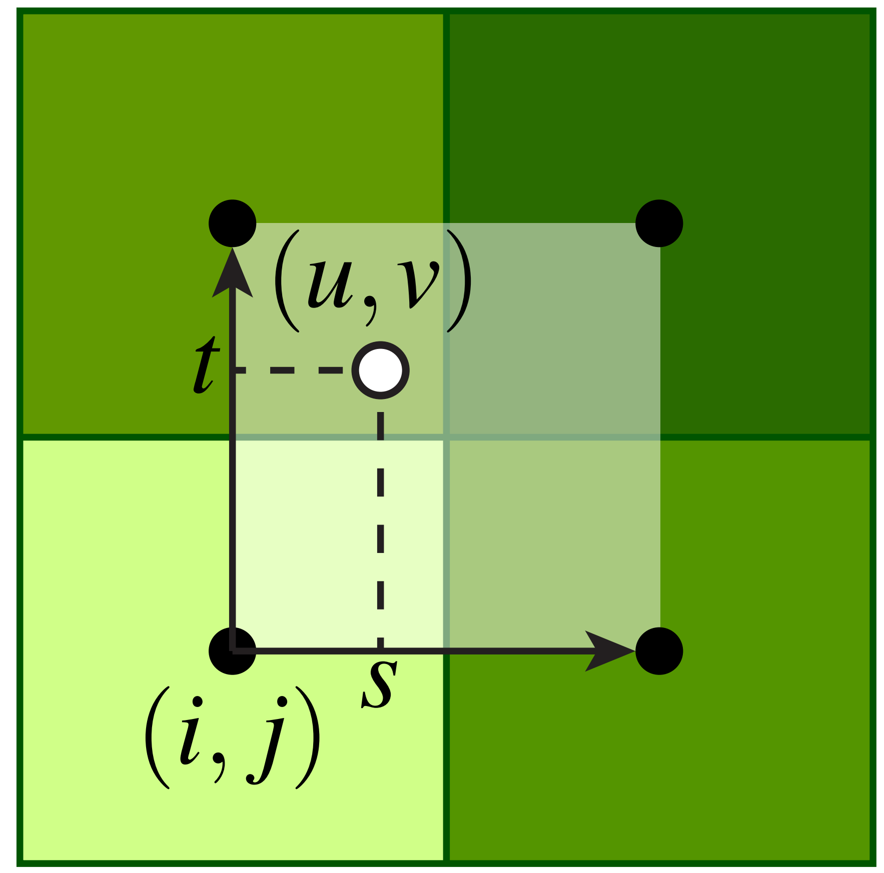
Figure credit: Akeley and Hanrahan

Bilinear interpolation (magnification)

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

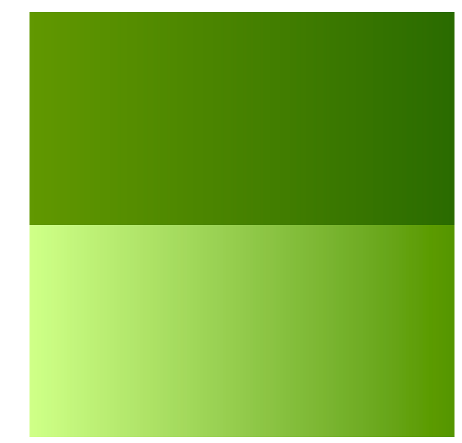


$$i = \left\lfloor u - \tfrac{1}{2} \right\rfloor$$
$$j = \left\lfloor v - \tfrac{1}{2} \right\rfloor$$



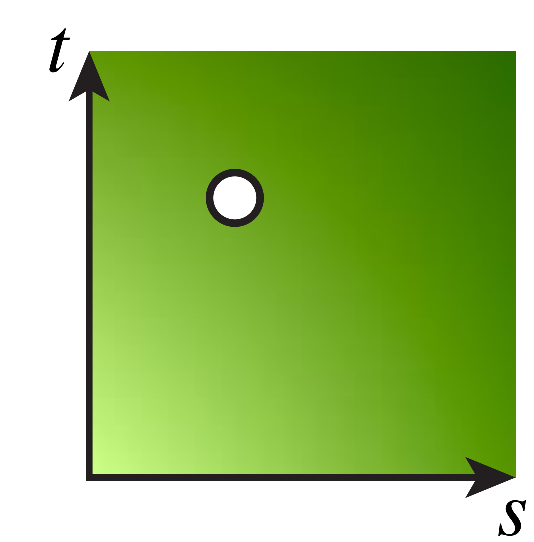
$$s = u - \left(i + \tfrac{1}{2}\right) \in [0, 1]$$
$$t = v - \left(j + \tfrac{1}{2}\right) \in [0, 1]$$

linear (each row)



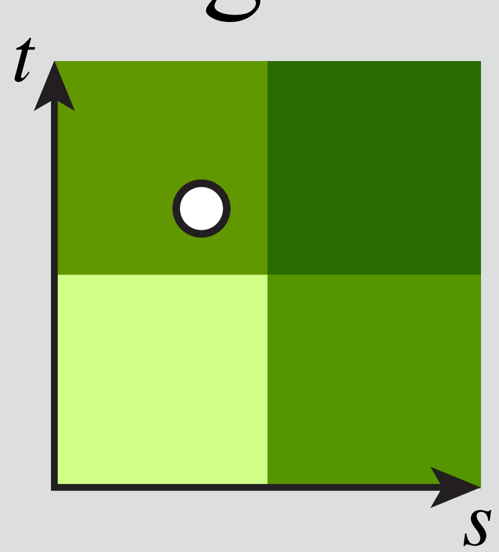
$$(1-s)f_{01} + s f_{11}$$
$$(1-s)f_{00} + s f_{10}$$

bilinear



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

nearest neighbor

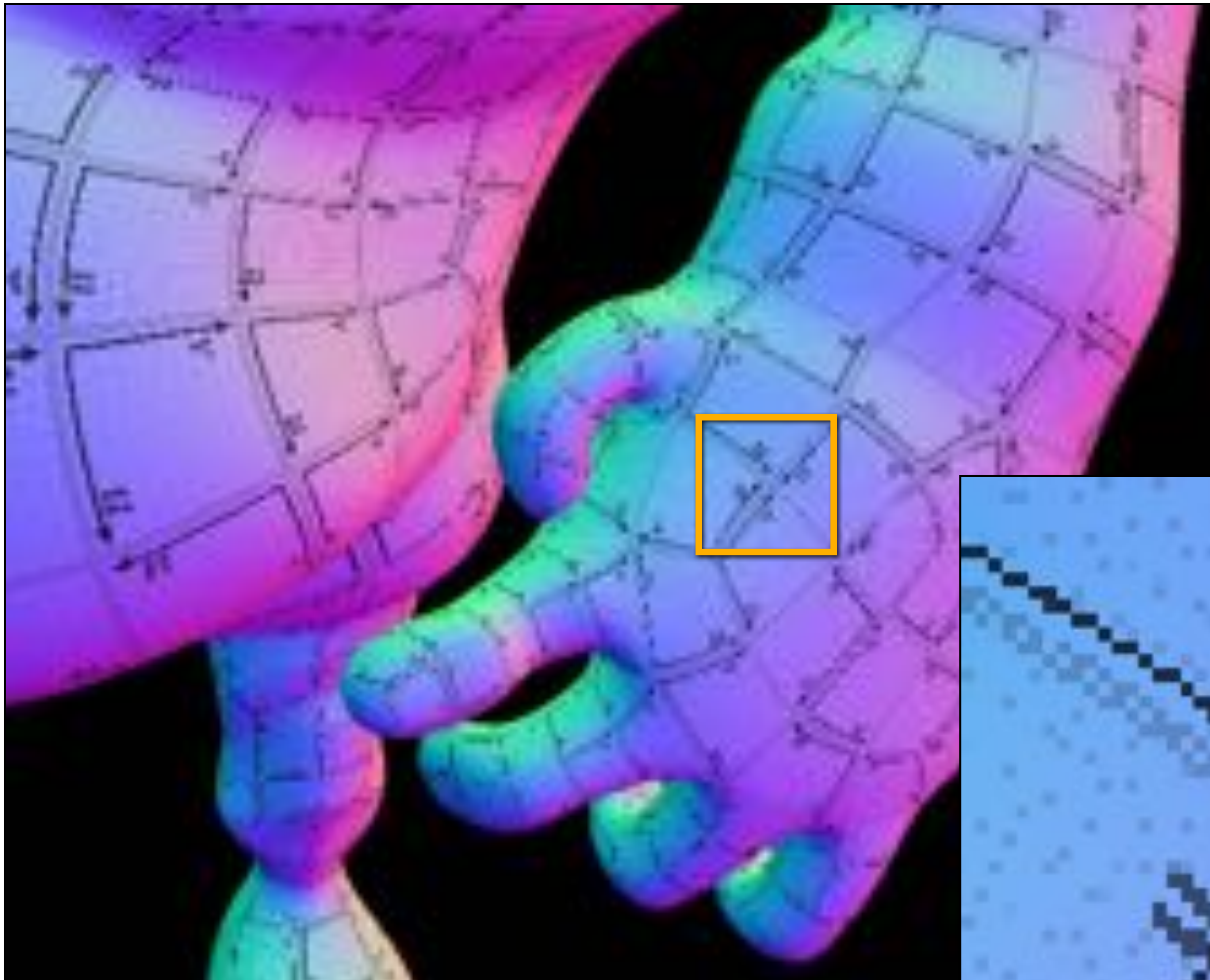


fast but ugly:

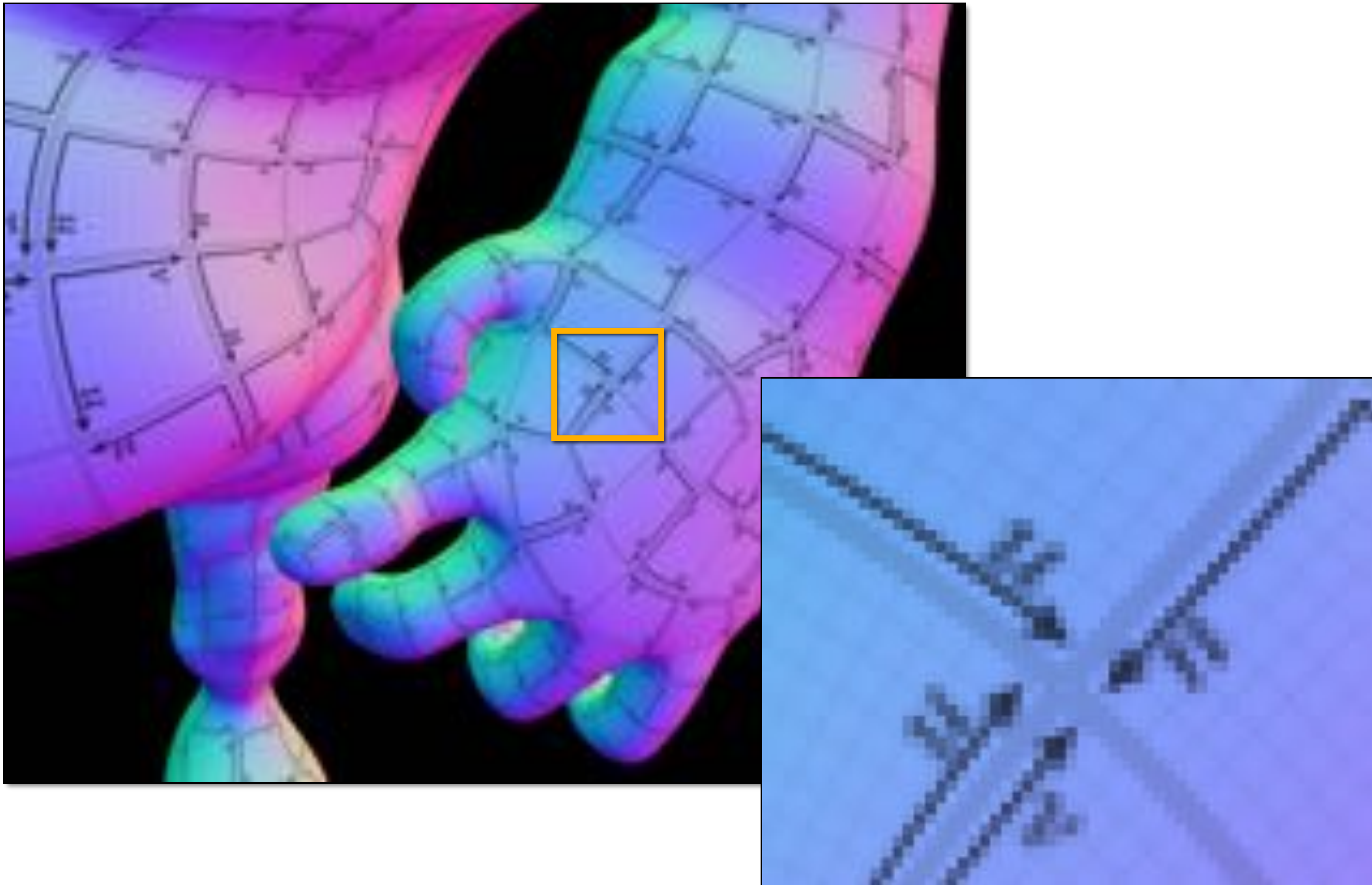
just grab value of nearest "texel" (texture pixel)

Q: What happens if we interpolate vertically <u>first</u>?

Aliasing due to minification

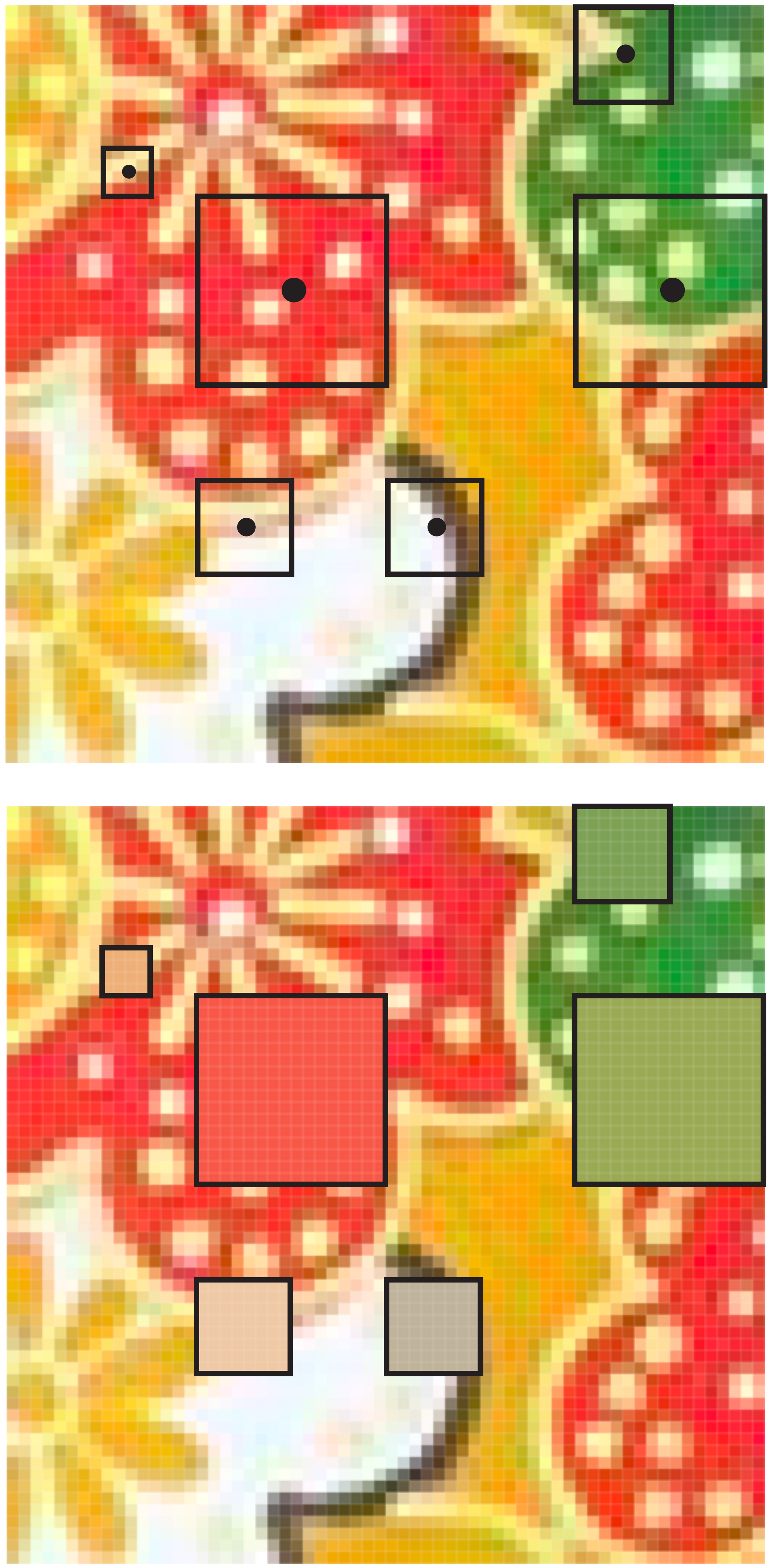


"Pre-filtering" texture (minification)



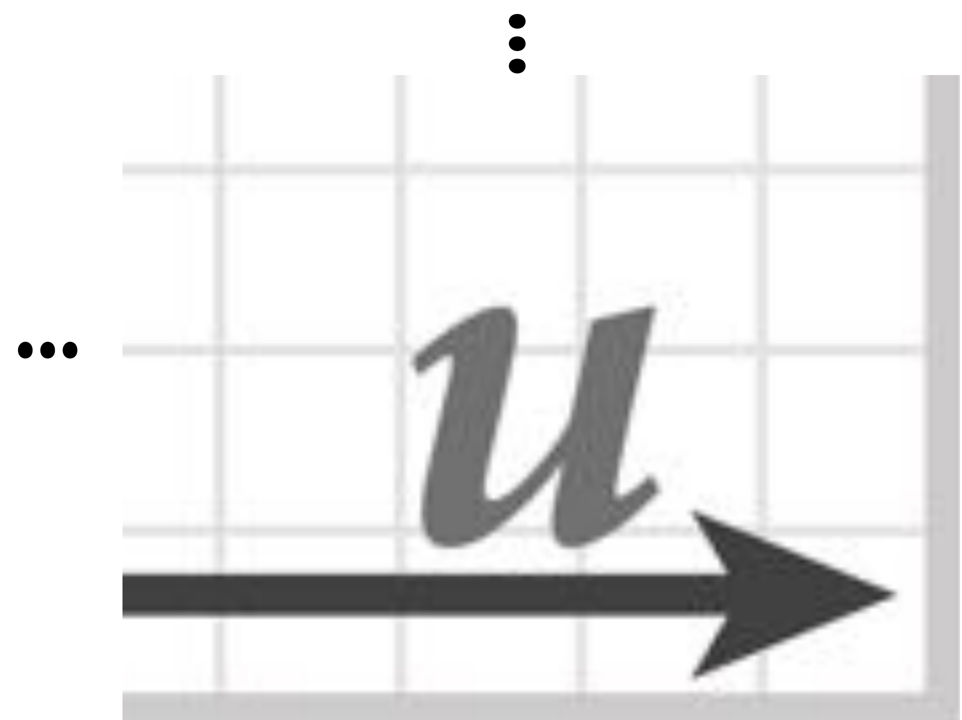
Texture prefiltering — basic idea

- Texture aliasing often occurs because a <u>single pixel</u> on the screen covers <u>many pixels</u> of the texture
- If we just grab the texture value at the center of the pixel, we get aliasing (get a "random" color that changes if the sample moves even very slightly)
- Ideally, would use the average texture value—but this is expensive to compute
- Instead, we can <u>pre-compute</u> the averages (once) and just look up these averages (many times) at run-time

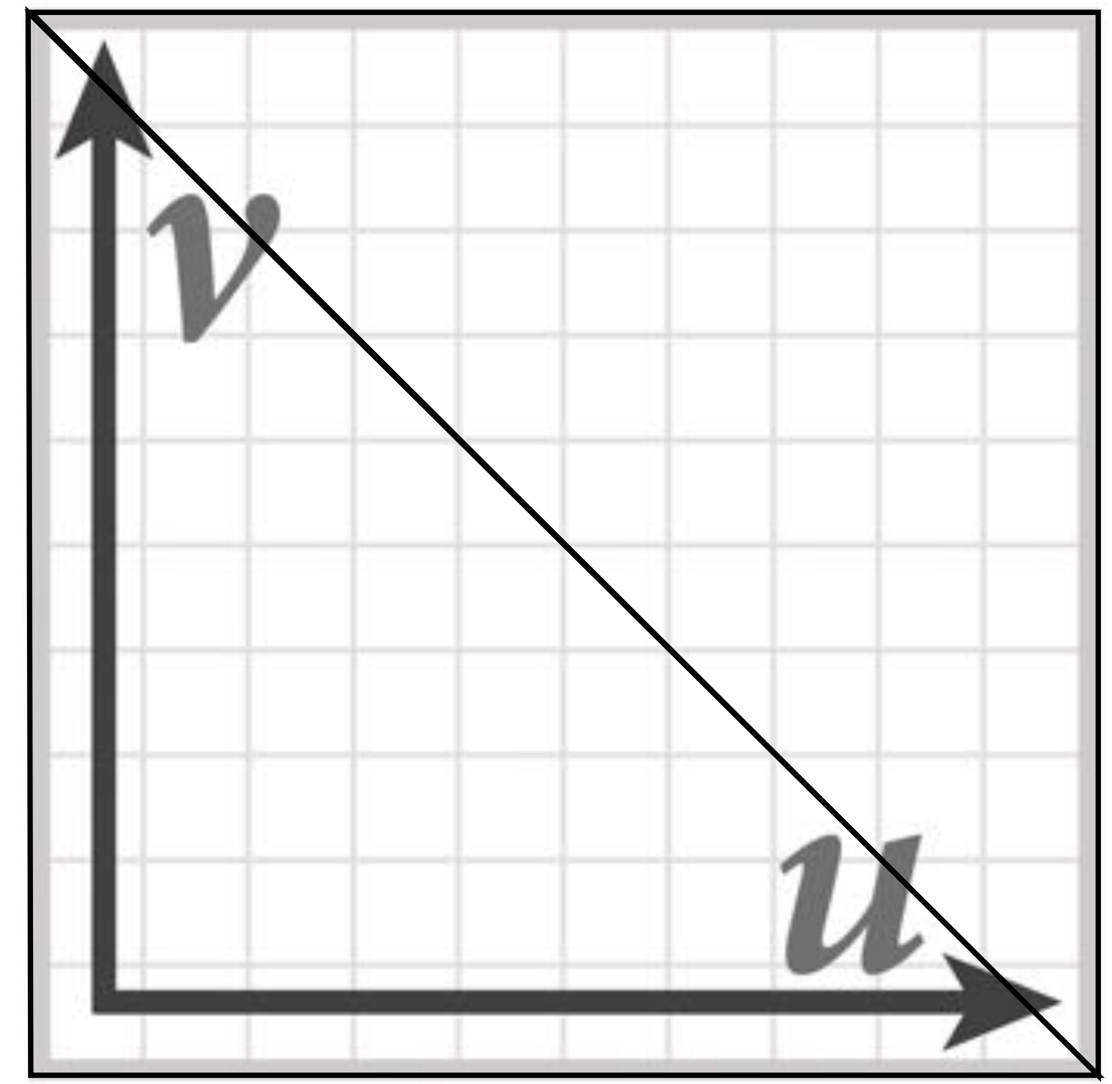


But which averages should we store? Can't precompute them all!

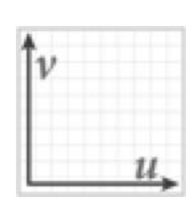
Prefiltered textures



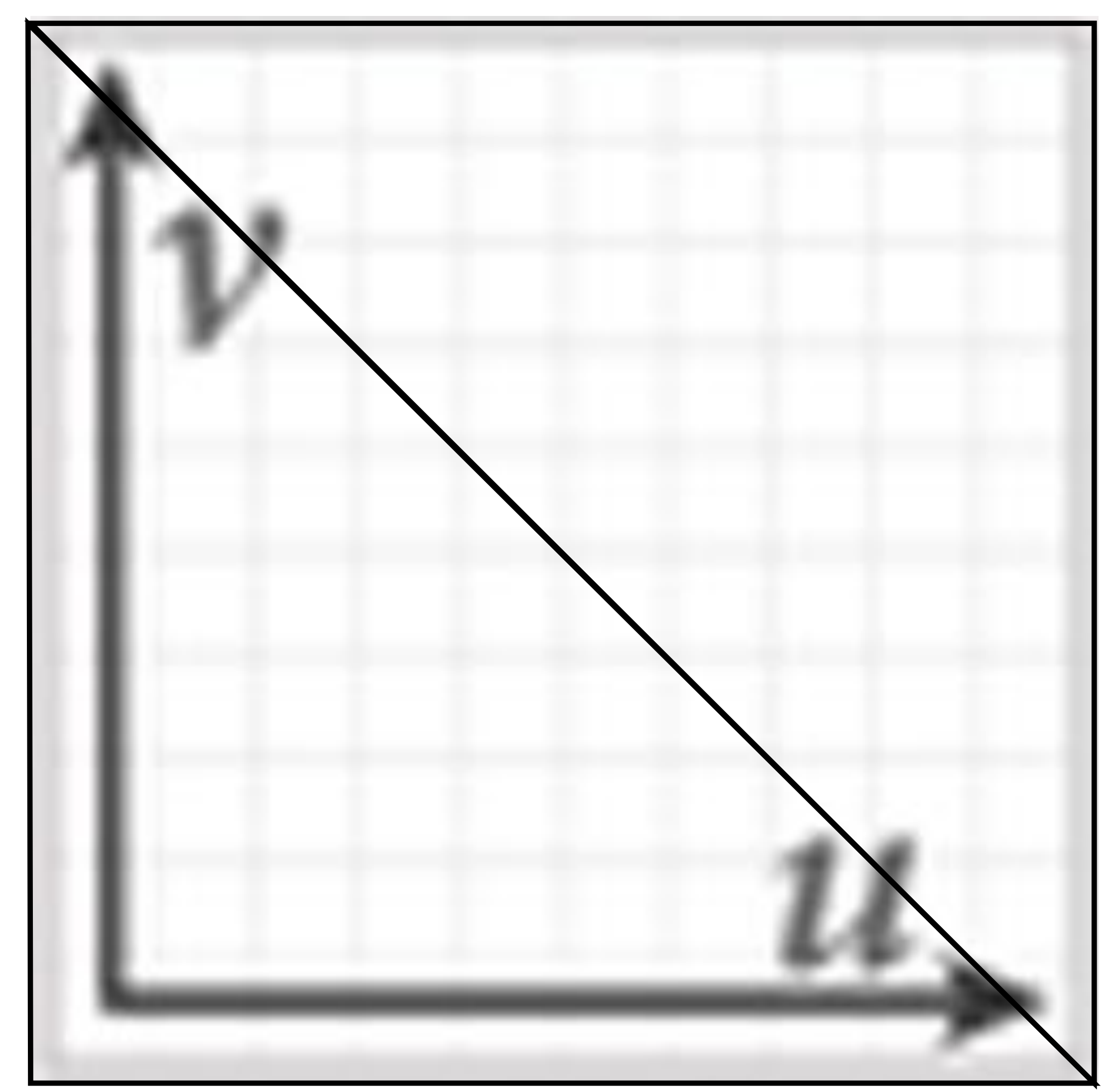
Actual texture: 700x700 image
(only a crop is shown)



Texture minification



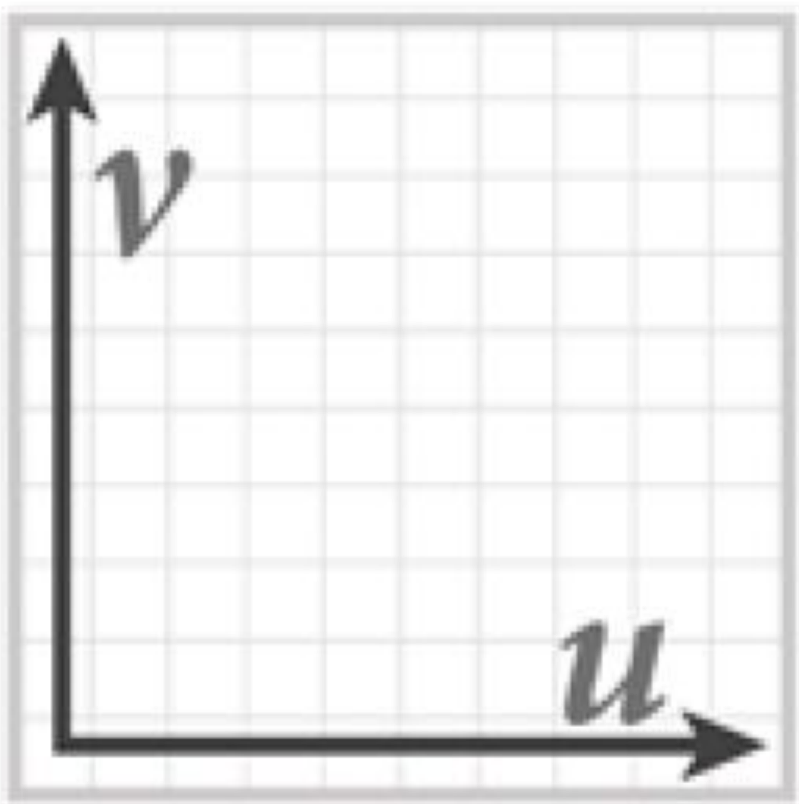
Actual texture: 64x64 image



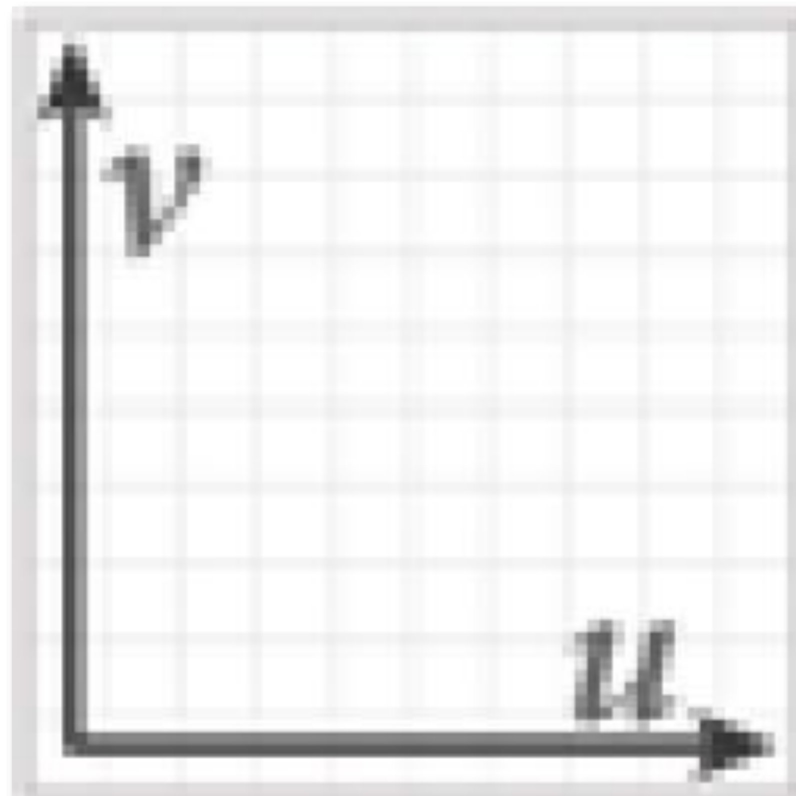
Texture magnification

Q: Are two resolutions enough? A: No…

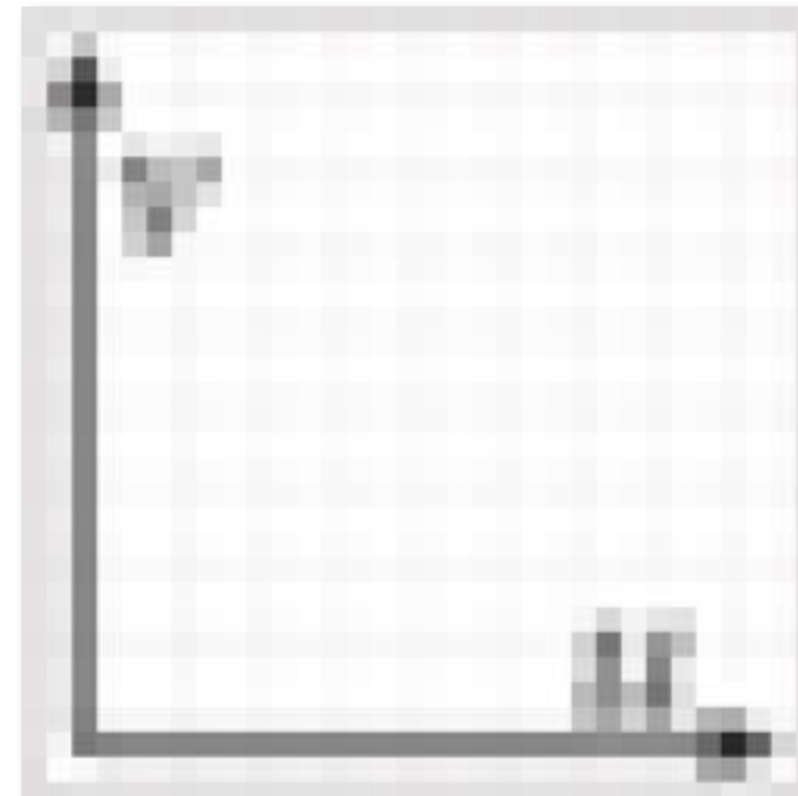
MIP map (L. Williams 83)



Level 0 = 128x128

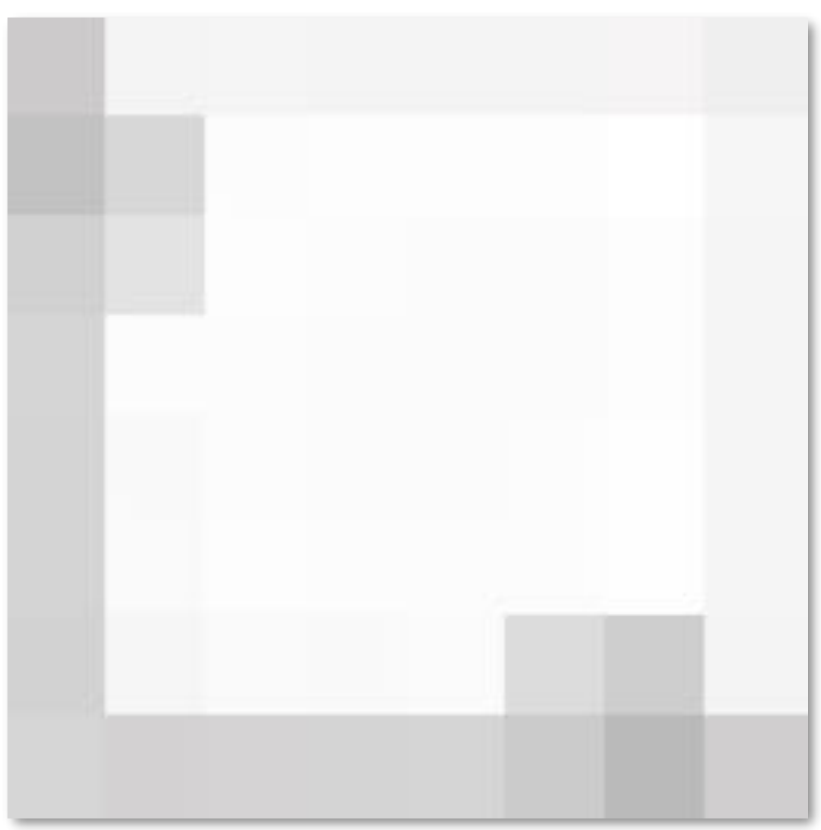


Level 1 = 64x64



Level 2 = 32x32

Level 3 = 16x16



Level 4 = 8x8

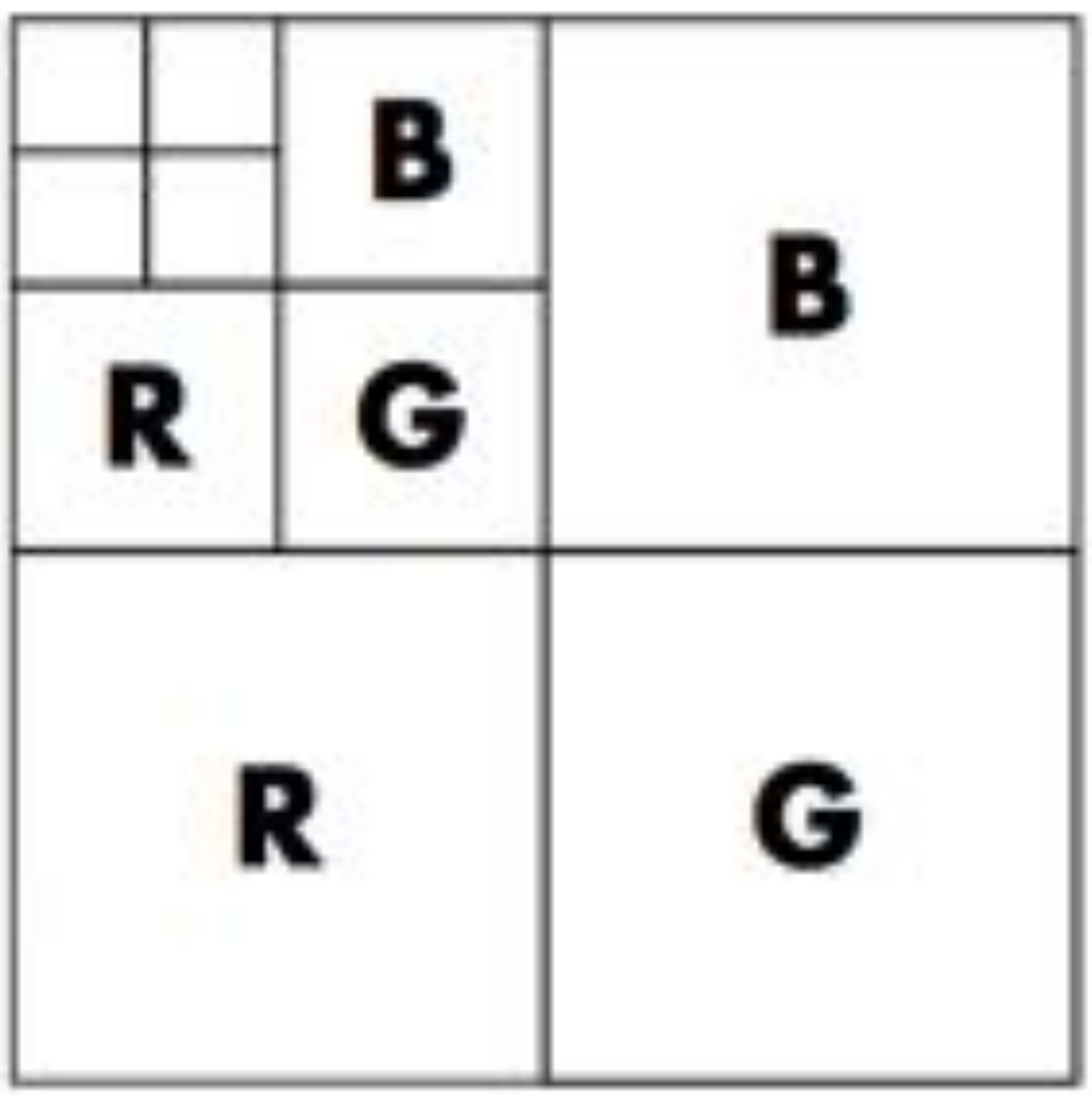
Level 5 = 4x4

Level 6 = 2x2

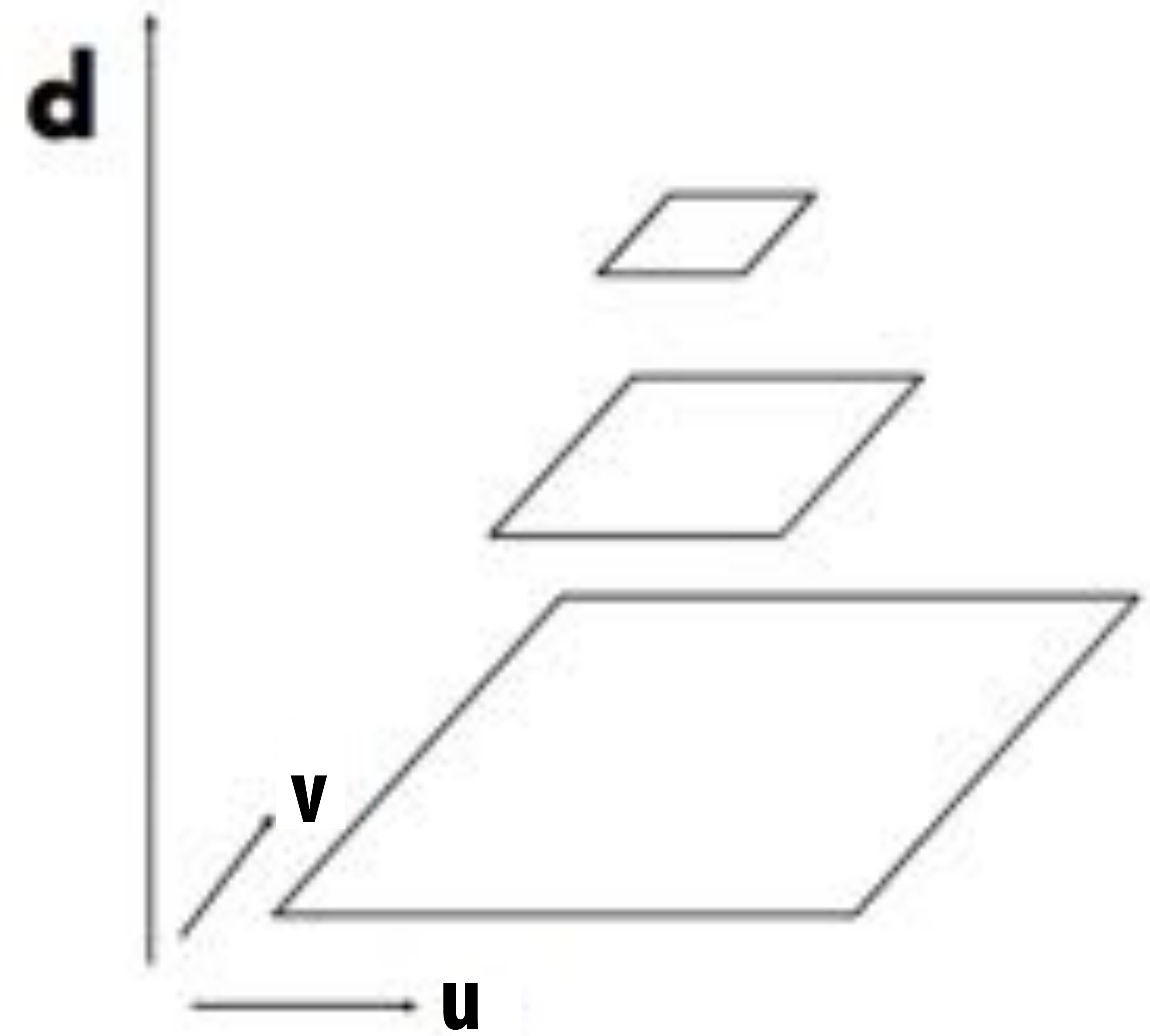
Level 7 = 1x1

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

Mipmap (L. Williams 83)



Williams' original proposed
mip-map layout



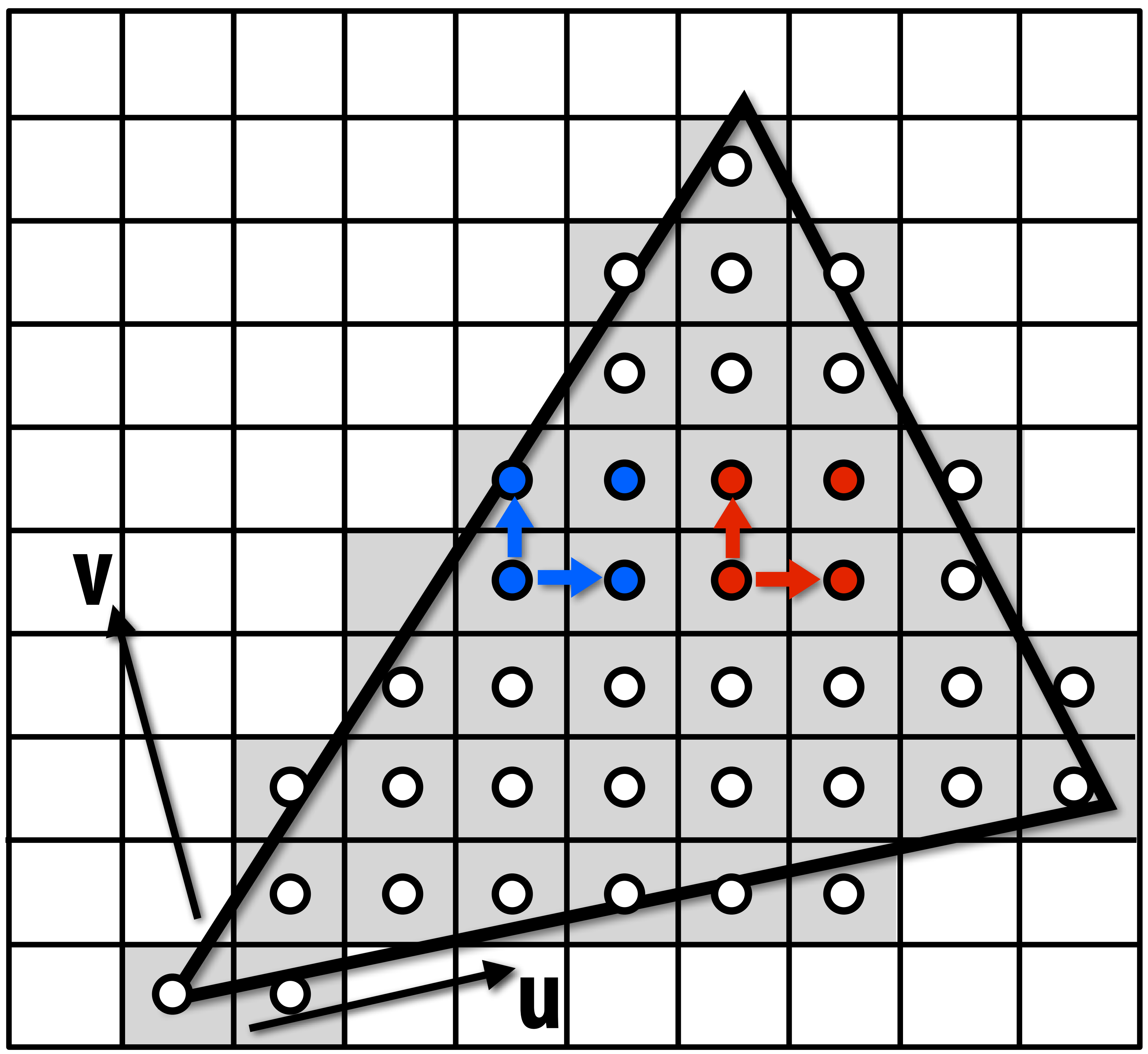
"Mip hierarchy"
level = d

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

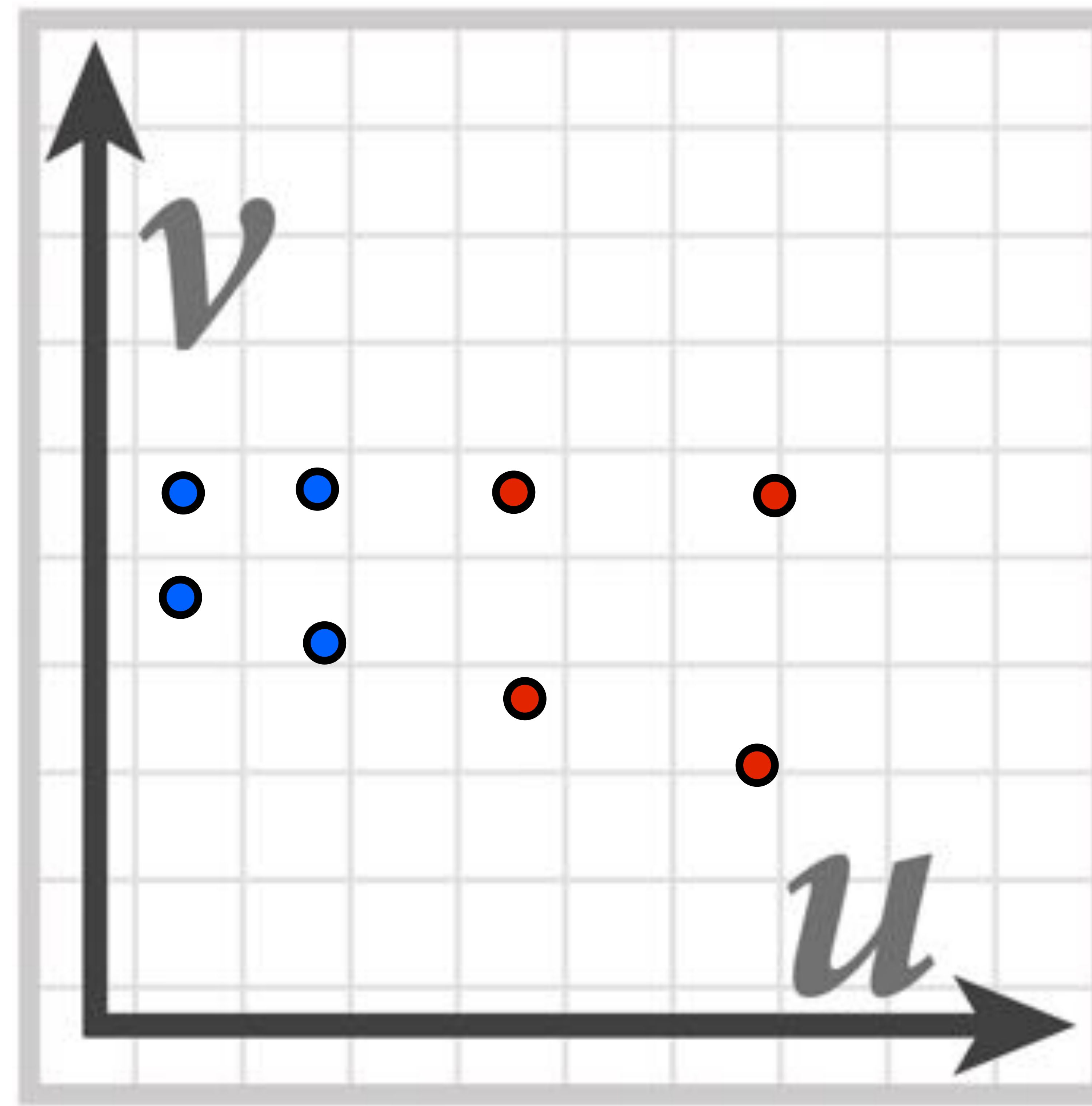
Slide credit: Akeley and Hanrahan

Computing MIP Map Level

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



Screen space

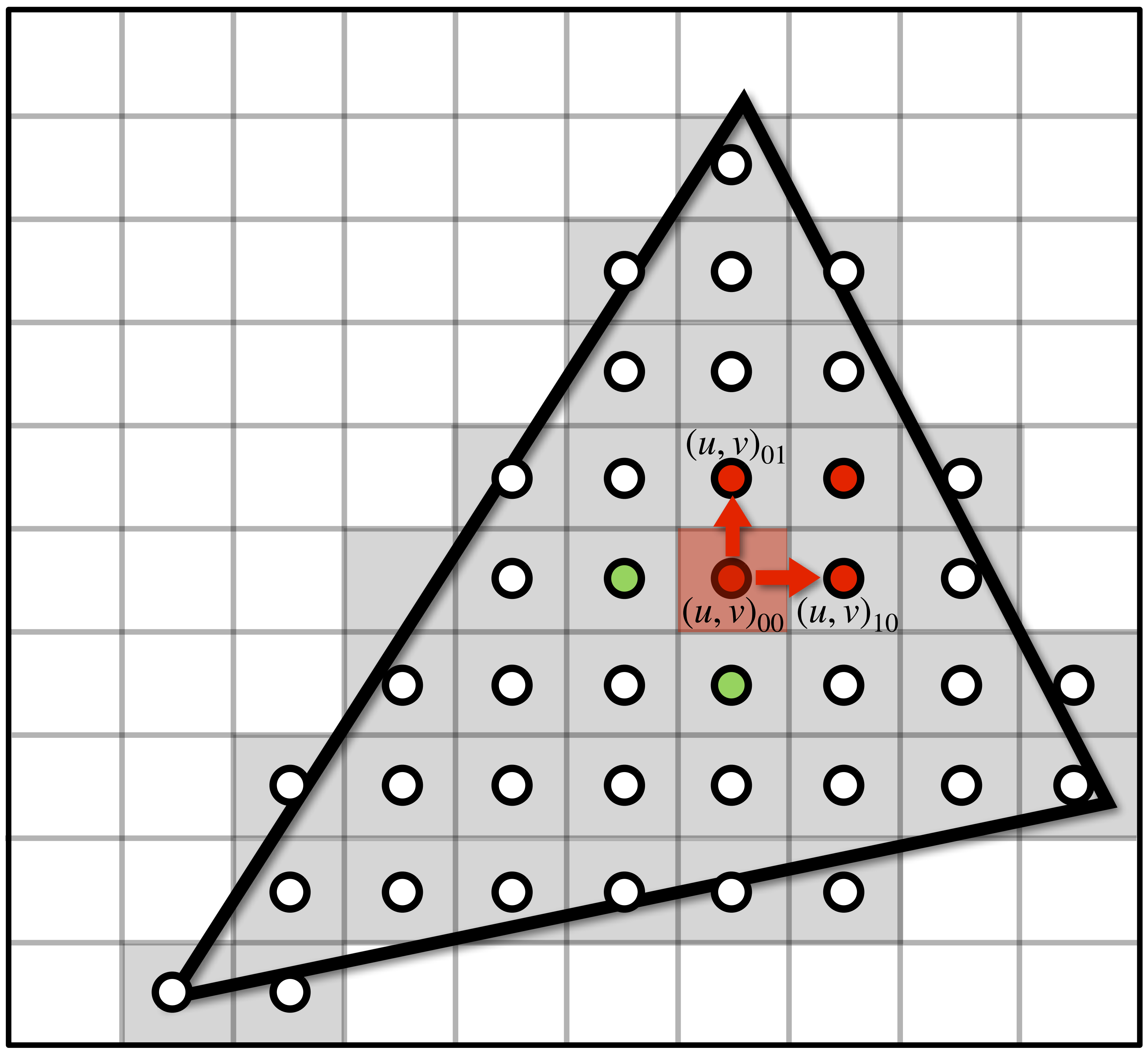


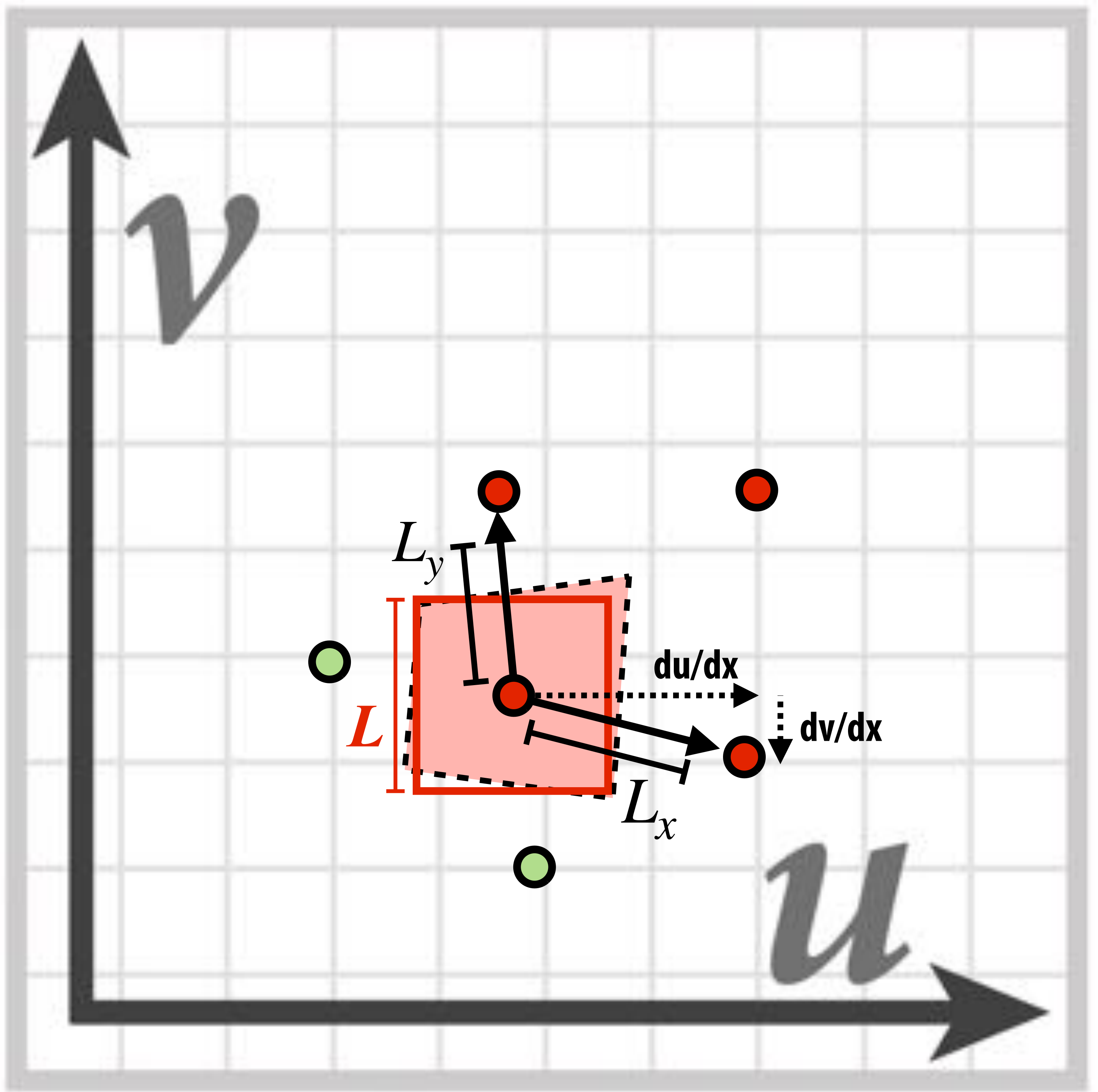
Texture space

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

Computing Mip Map Level

Compute differences between texture coordinate values at neighboring samples





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

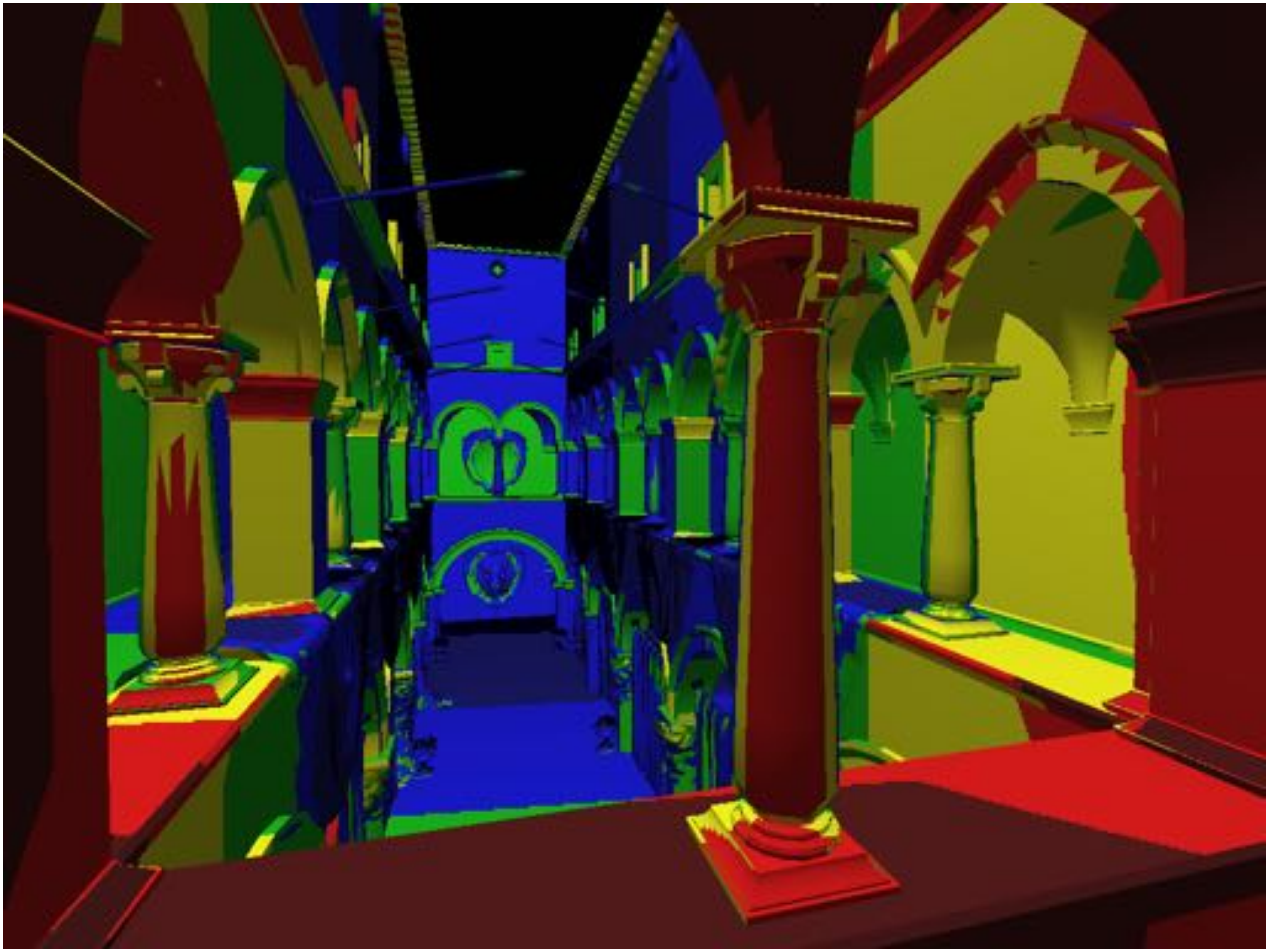
$$\frac{du}{dy} = u_{01} - u_{00} \qquad \frac{dv}{dy} = v_{01} - v_{00}$$

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

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

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

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

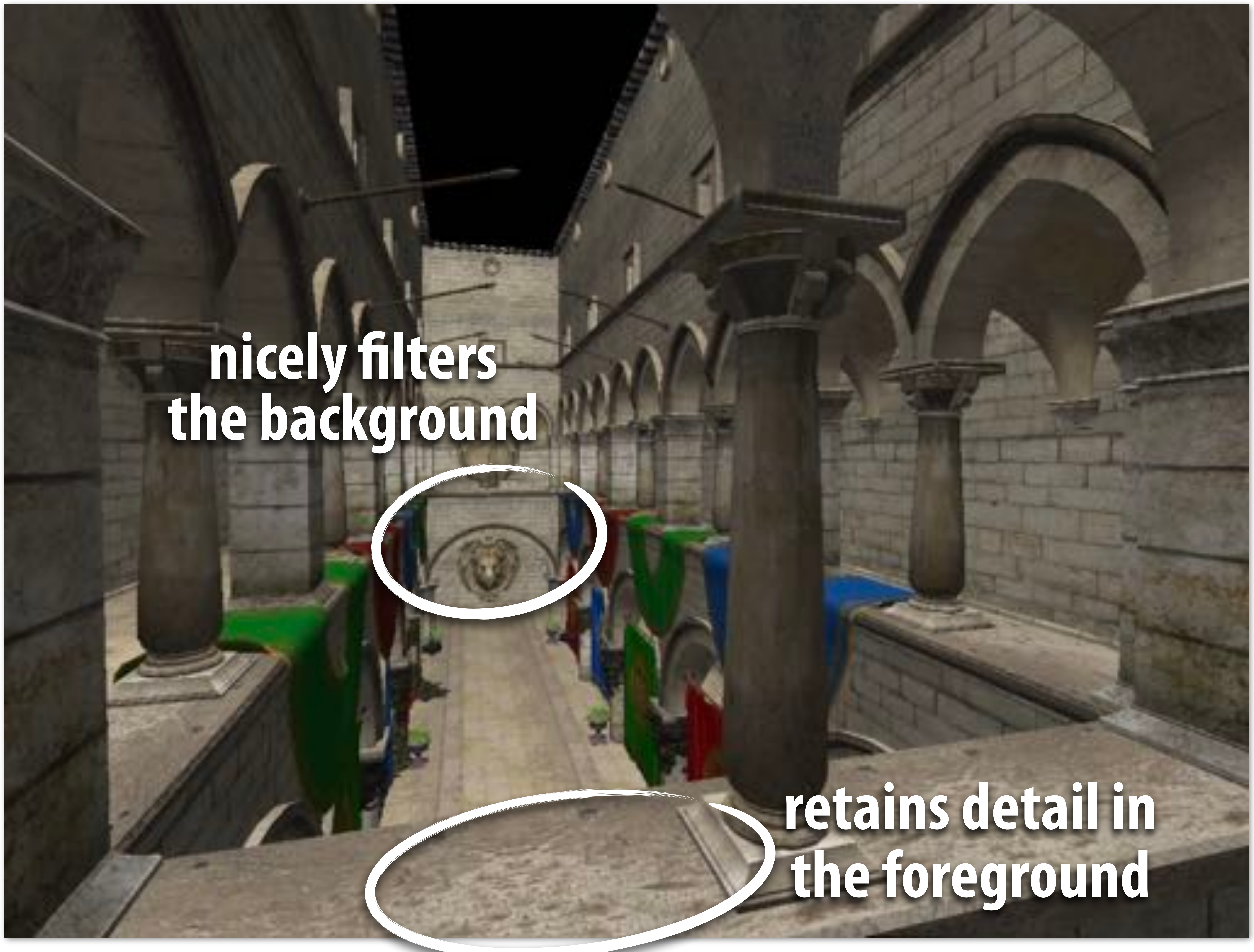


Sponza (bilinear resampling at level 0)

Sponza (bilinear resampling at level 2)

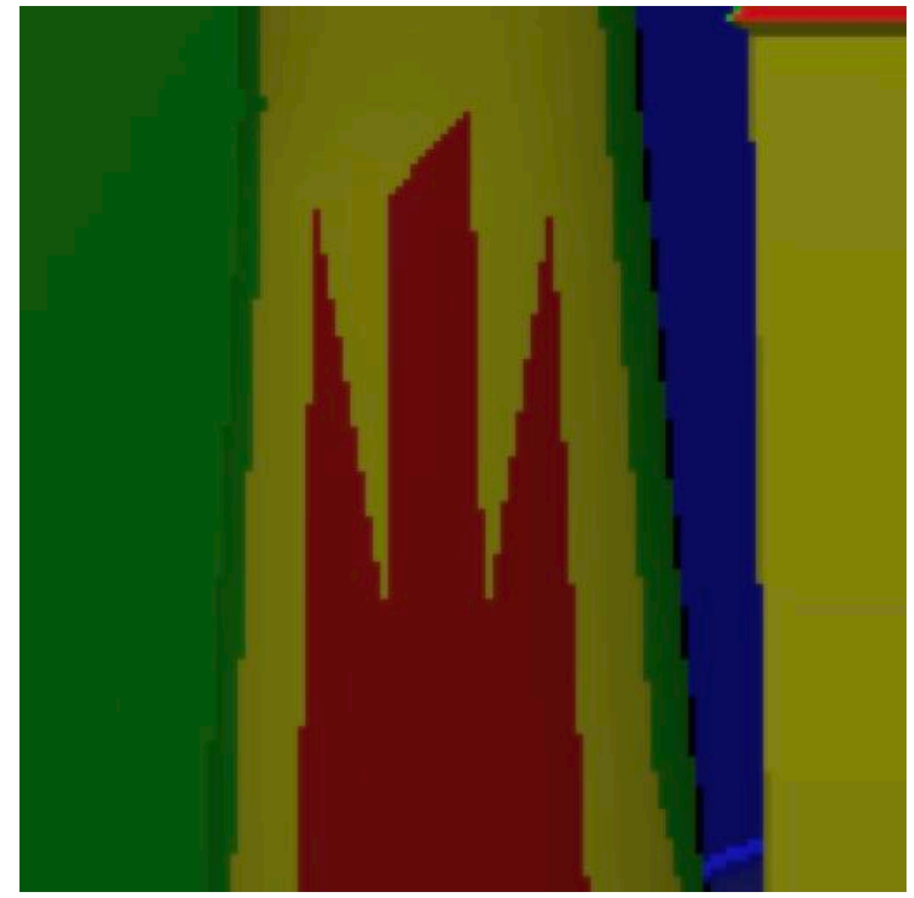
Sponza (bilinear resampling at level 4)

Sponza (MIP mapped)



Problem with basic MIP mapping

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



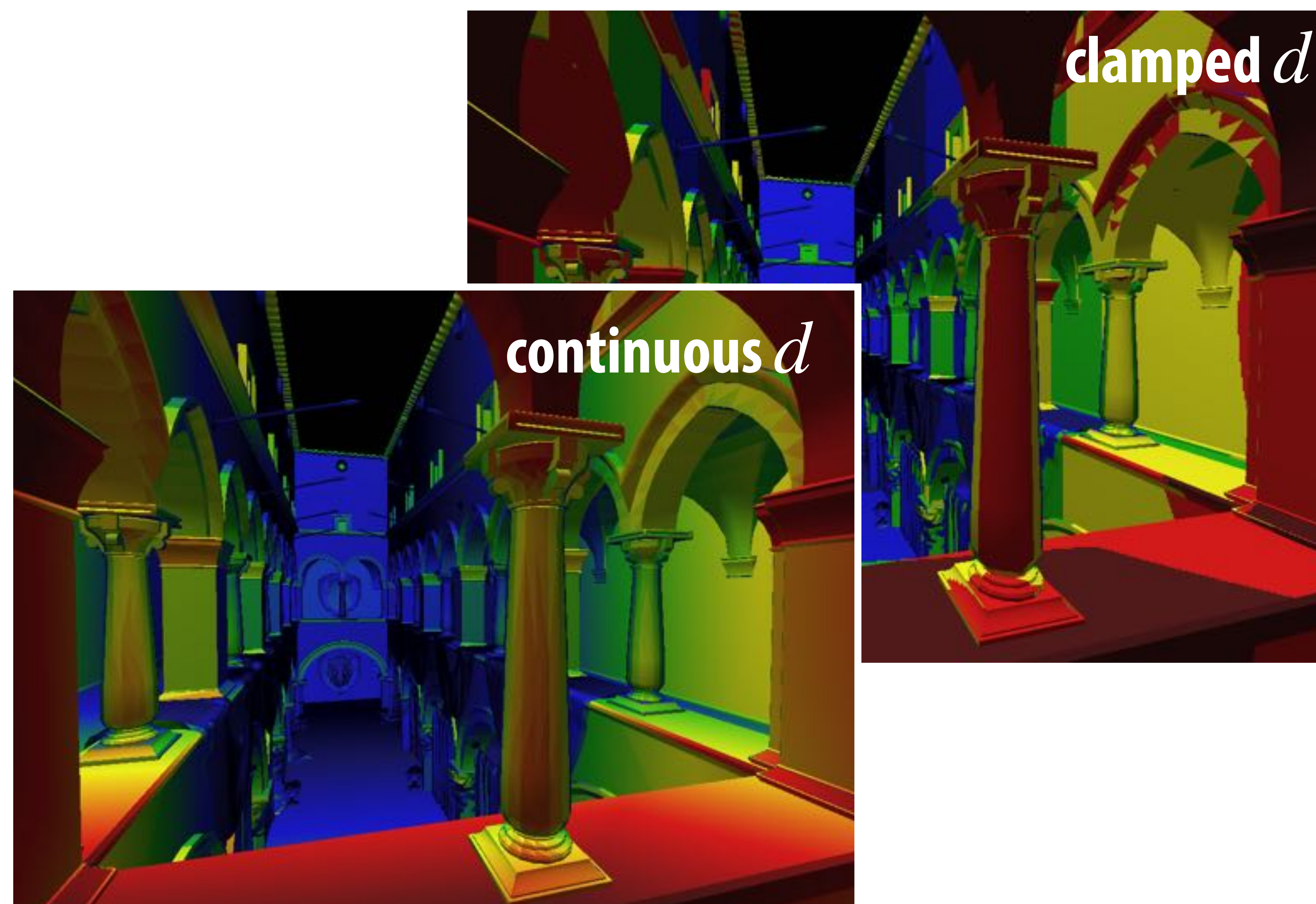
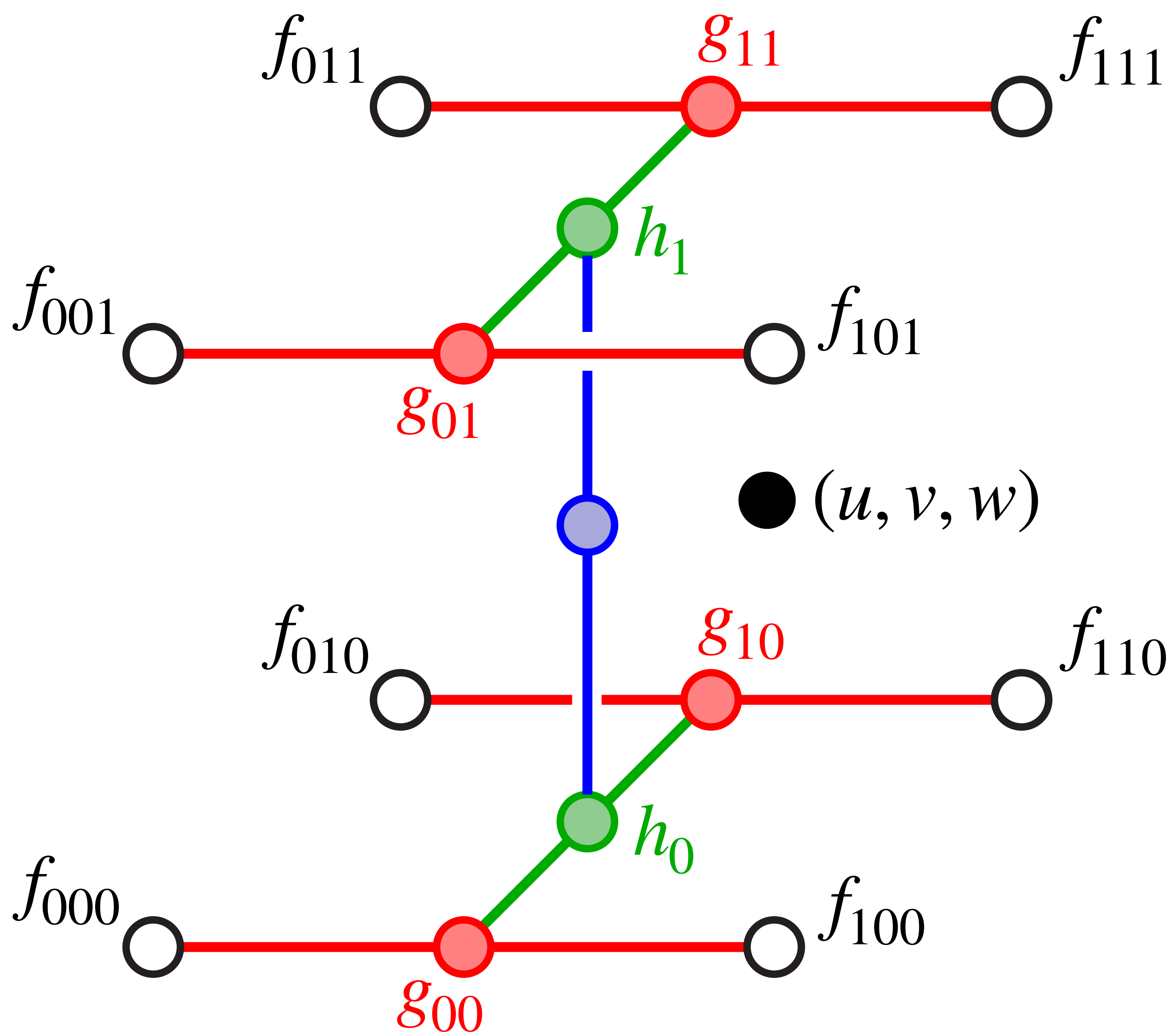


image adapted from: Akeley and Hanrahan

Trilinear Filtering

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



$g_{00} = (1-u)f_{000} + uf_{100}$

$g_{10} = (1-u)f_{010} + uf_{110}$

$g_{01} = (1-u)f_{001} + uf_{101}$

$g_{11} = (1-u)f_{011} + uf_{111}$

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

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

$(1-w)h_0 + wh_1$

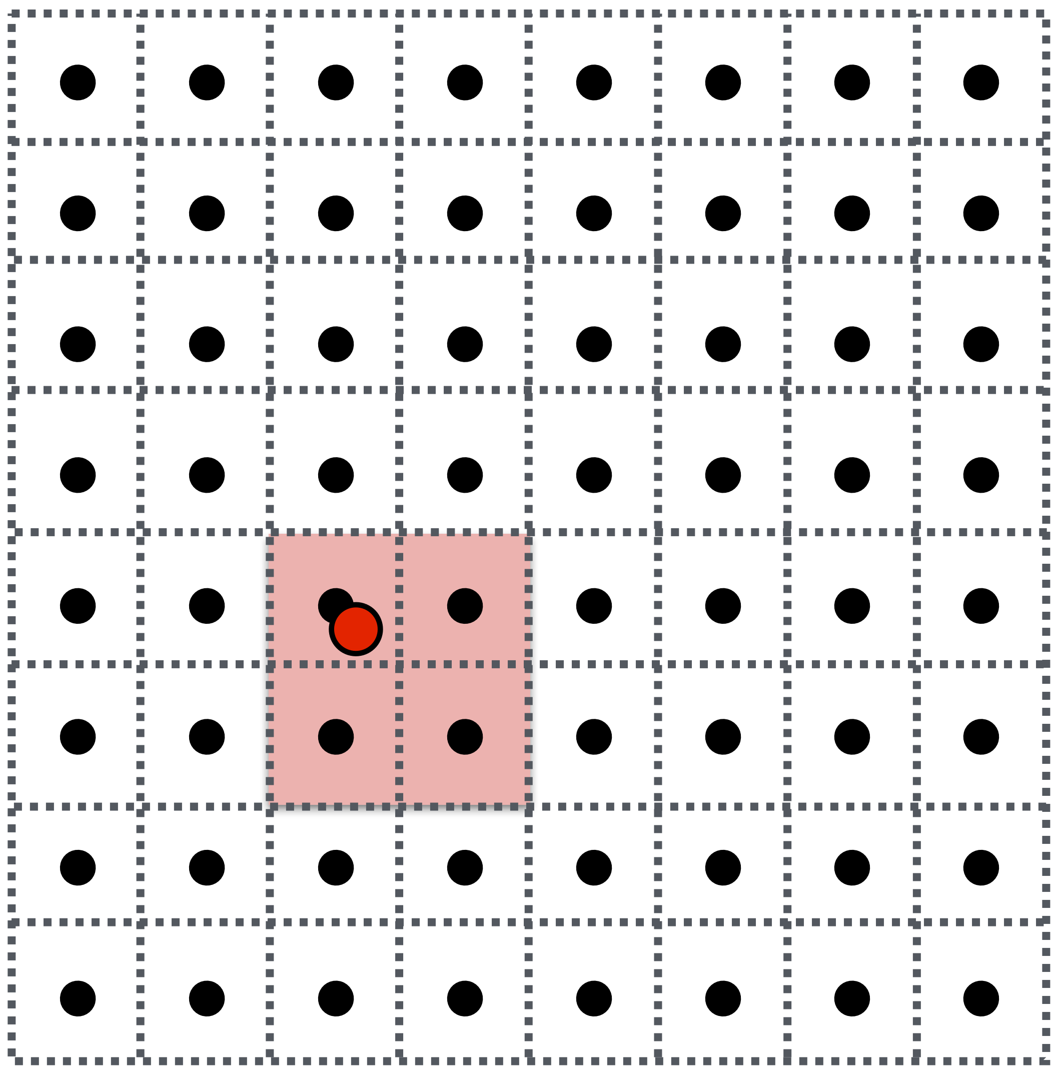
MIP Map Lookup

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

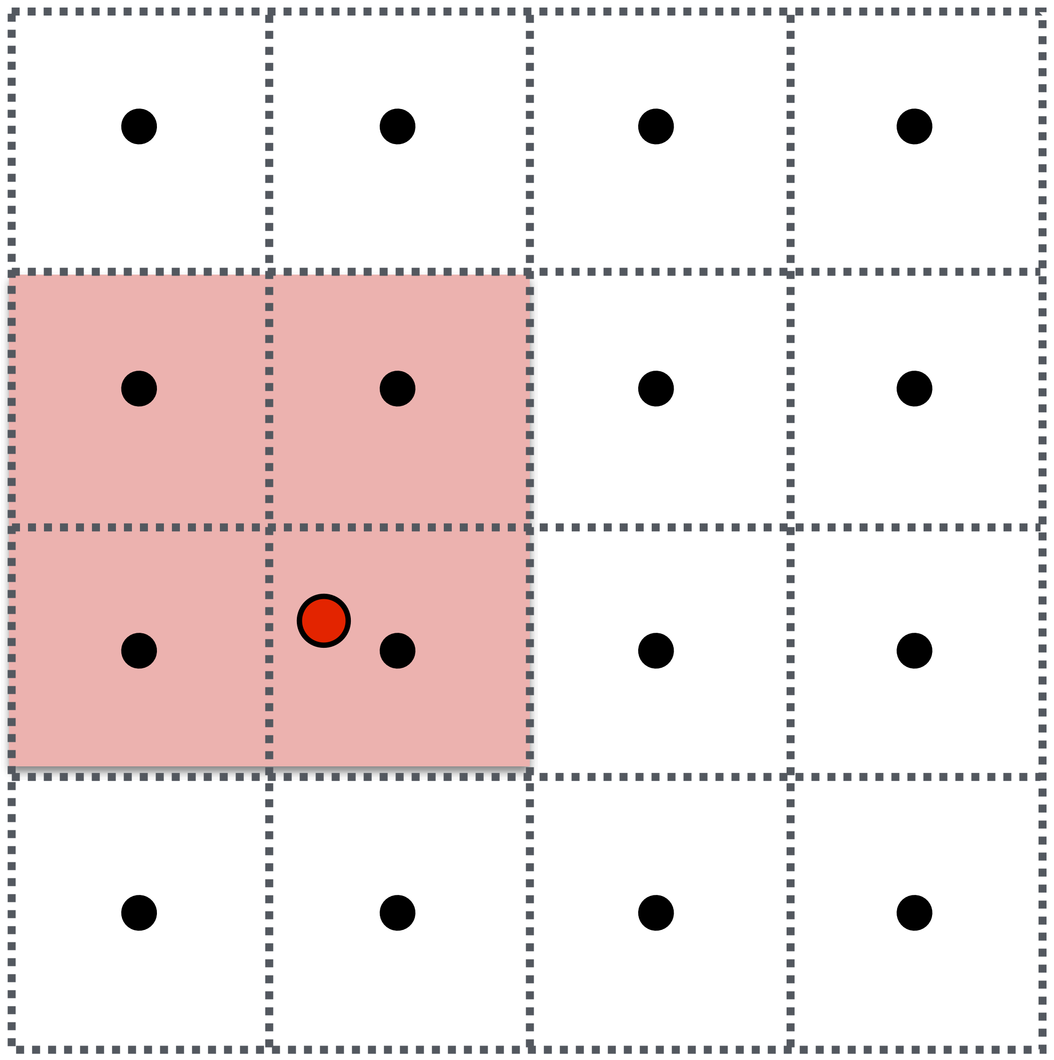
Starts getting expensive! (➟ specialized hardware)

<u>Bilinear interpolation:</u>
four texel reads
3 linear interpolations (3 mul + 6 add)

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



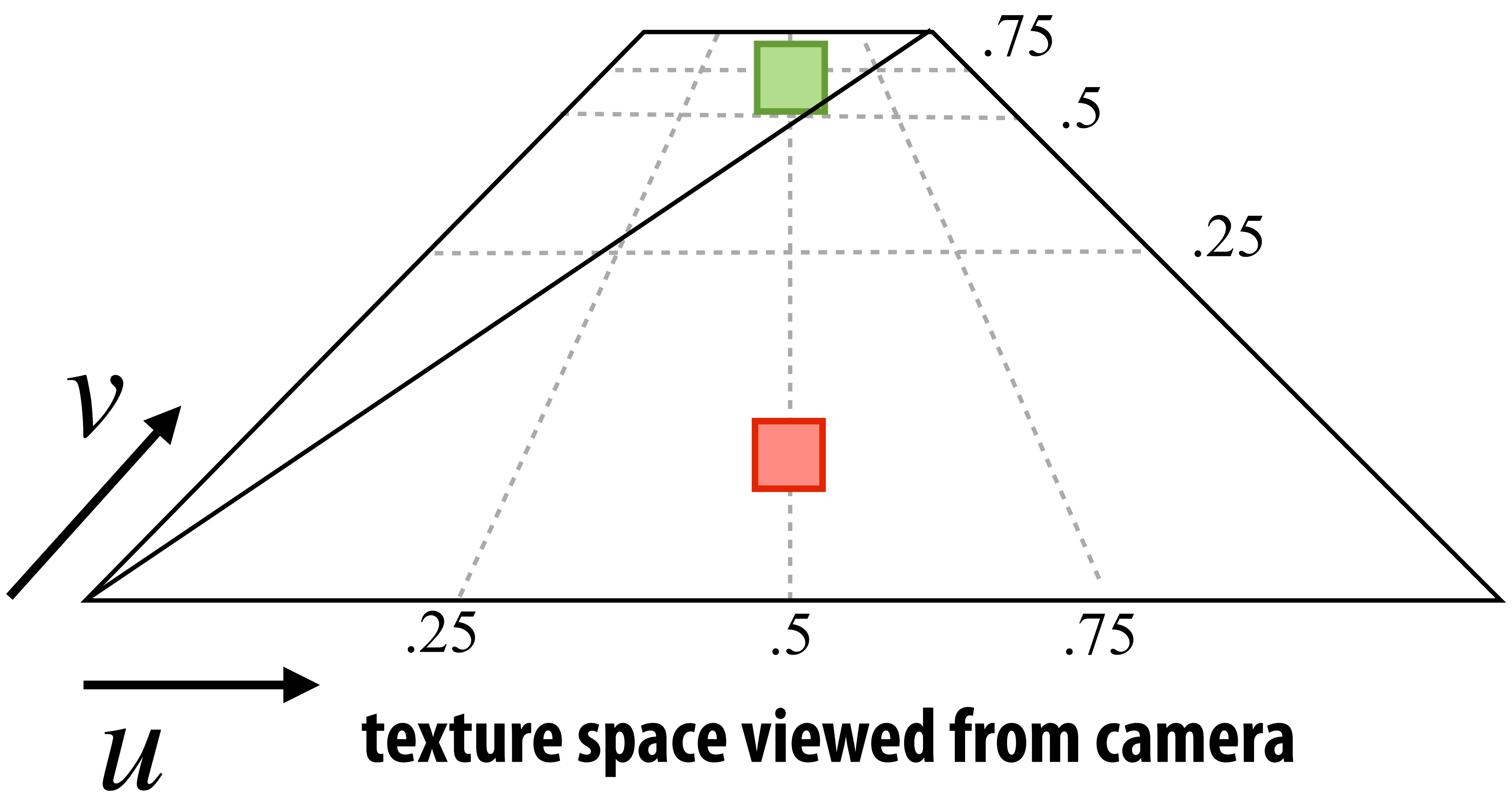
mip-map texels: level $\lfloor d \rfloor$



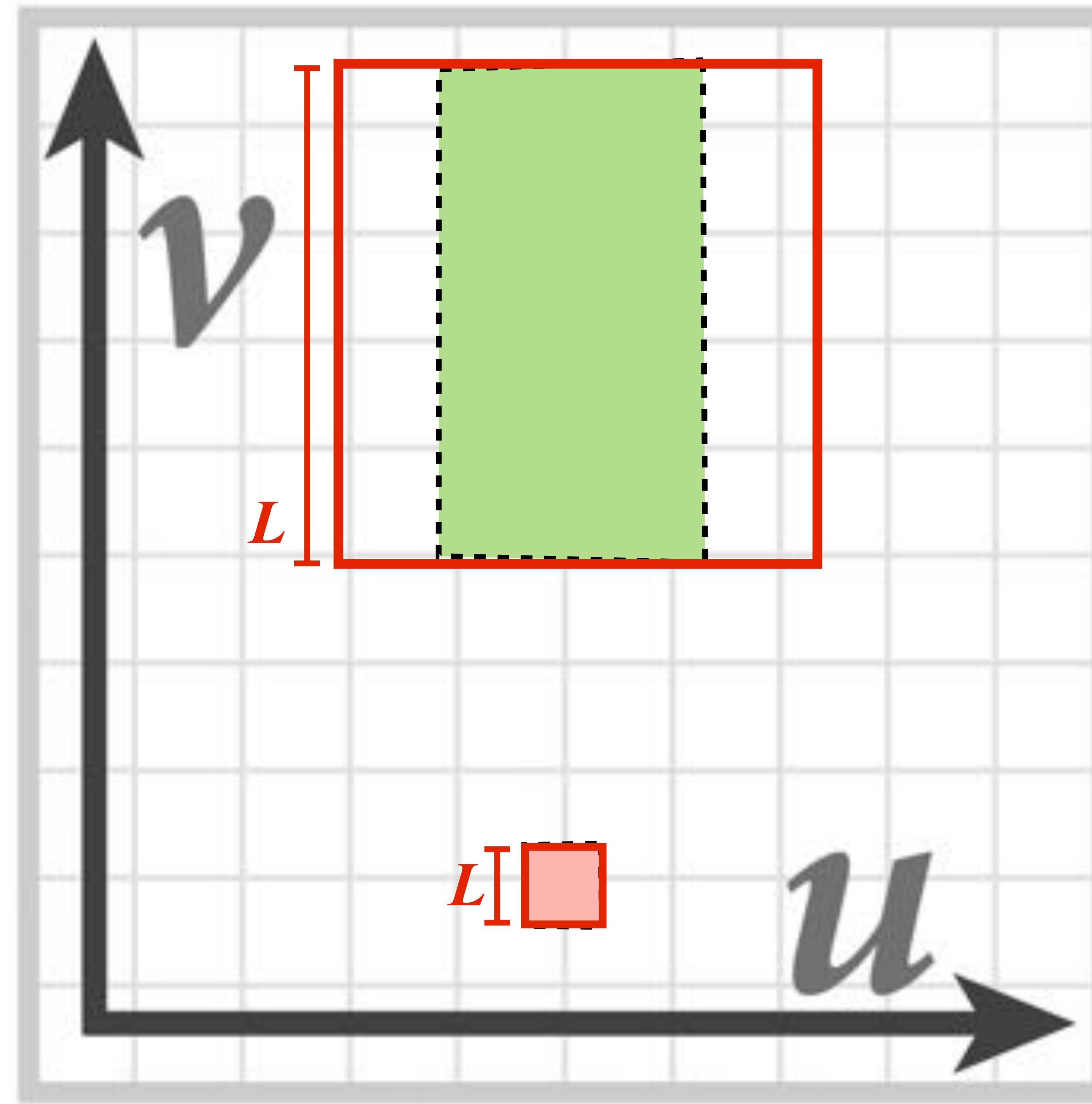
mip-map texels: level $\lfloor d \rfloor + 1$

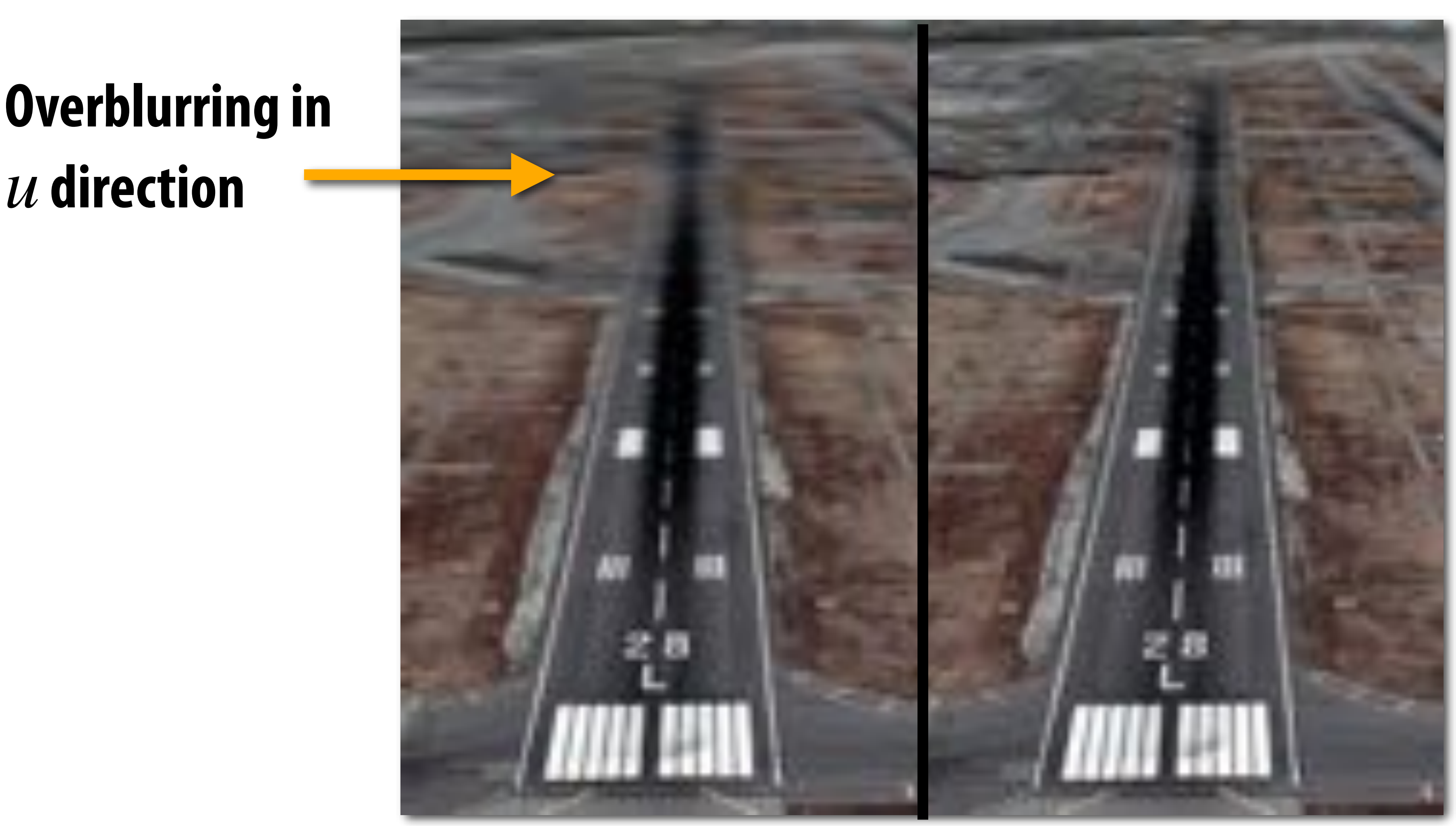
Anisotropic Filtering

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



texture space viewed from camera w/ perspective projection





isotropic Filtering (trilinear) | **anisotropic Filtering**

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

Texture Sampling Pipeline

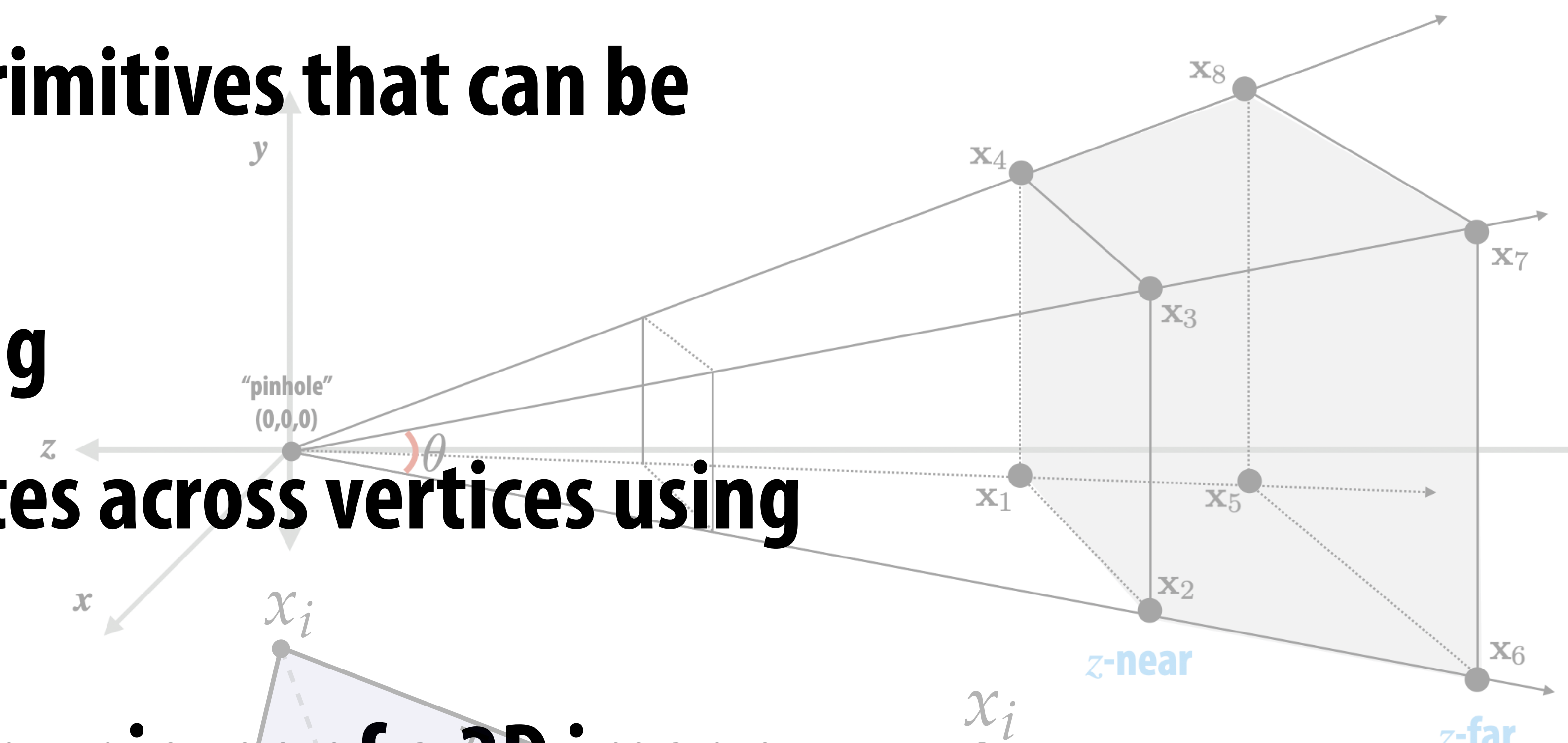
1. Compute u and v from screen sample (x, y) via barycentric interpolation
2. Approximate $\frac{du}{dx}, \frac{du}{dy}, \frac{dv}{dx}, \frac{dv}{dy}$ by taking differences of screen-adjacent samples
3. Compute mip map level d
4. Convert normalized $[0,1]$ texture coordinate (u, v) to pixel locations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

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

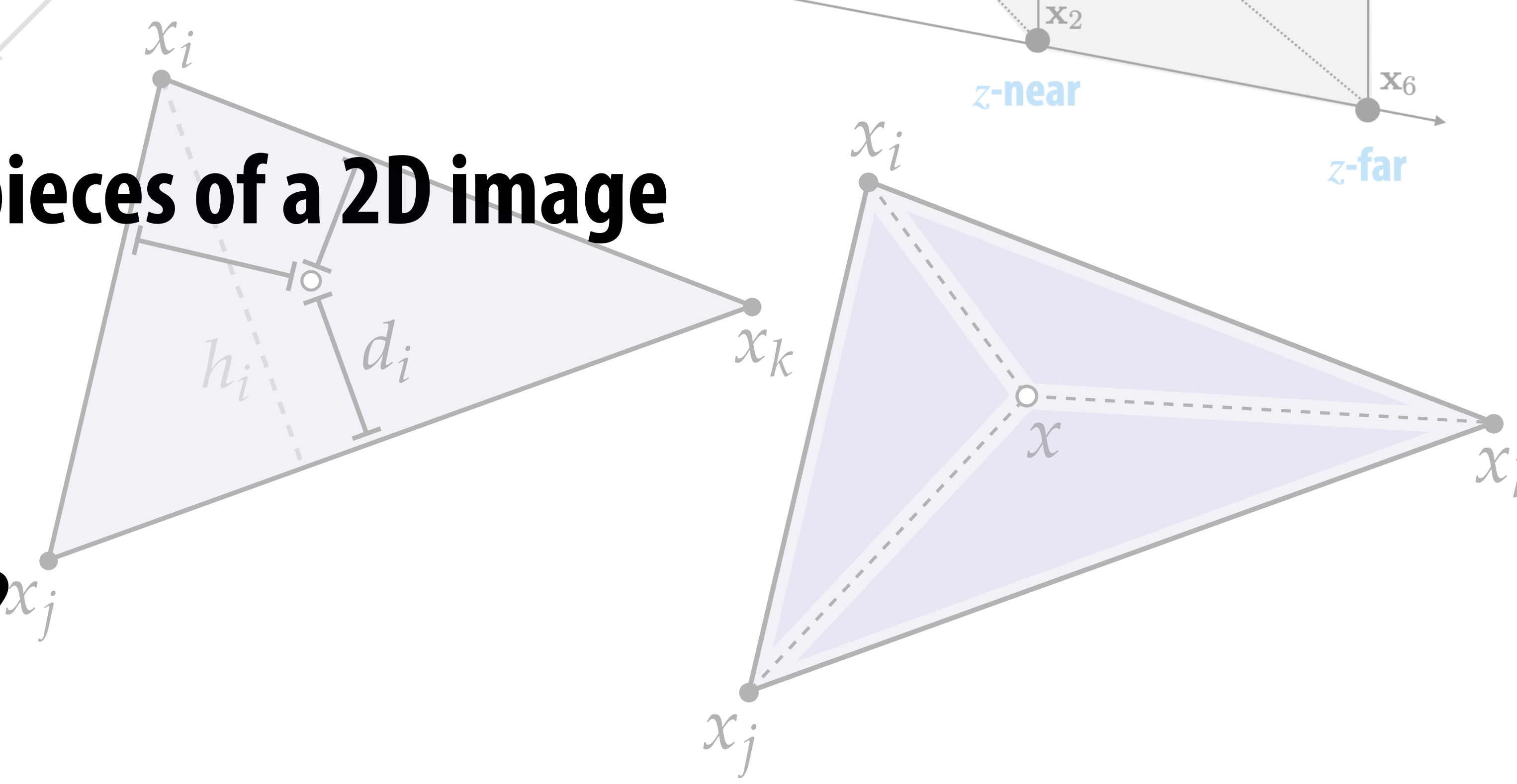
For this reason, graphics processing units (GPUs) have dedicated, fixed-function hardware support to perform texture sampling operations

Perspective & Texture Mapping—Summary

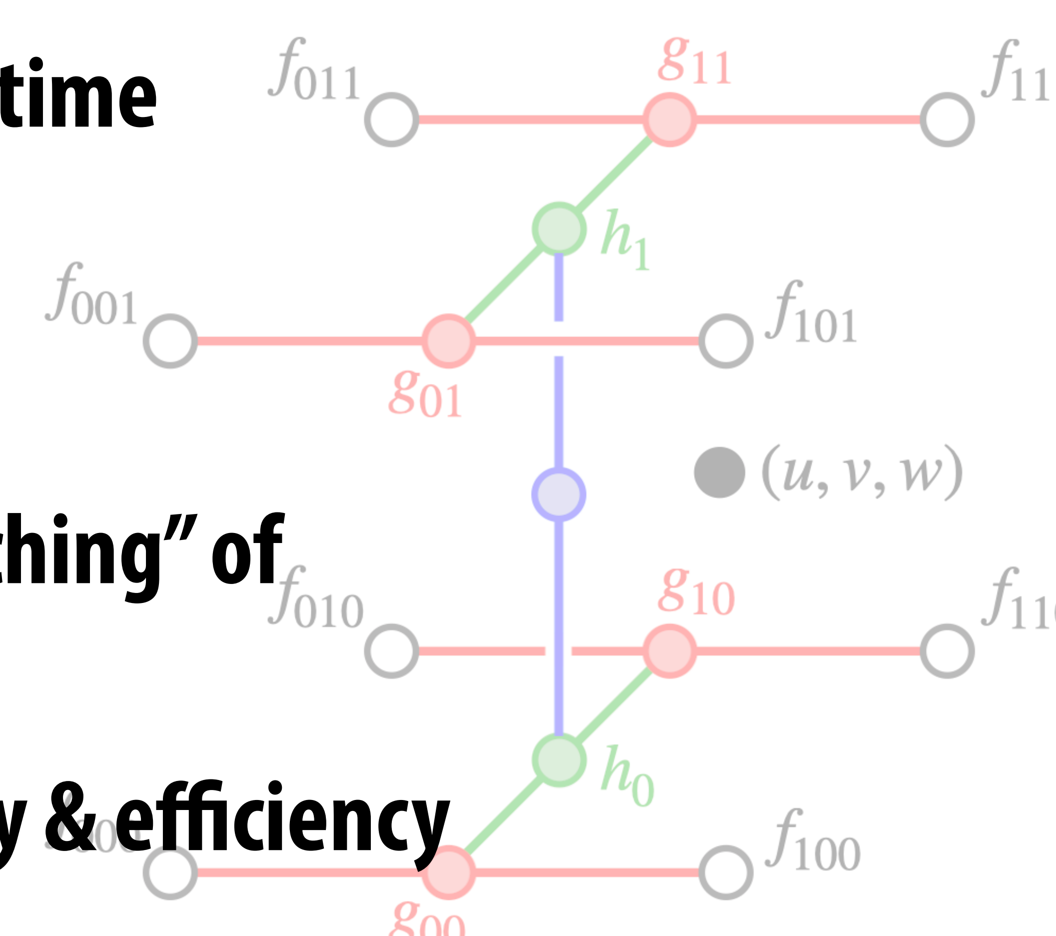
- Perspective projection turns 3D primitives into 2D primitives that can be rasterized
  - View frustum used to manage clipping, Z-fighting



- Once we have 2D primitives, can interpolate attributes across vertices using barycentric coordinates



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



Next Time: Depth & Transparency