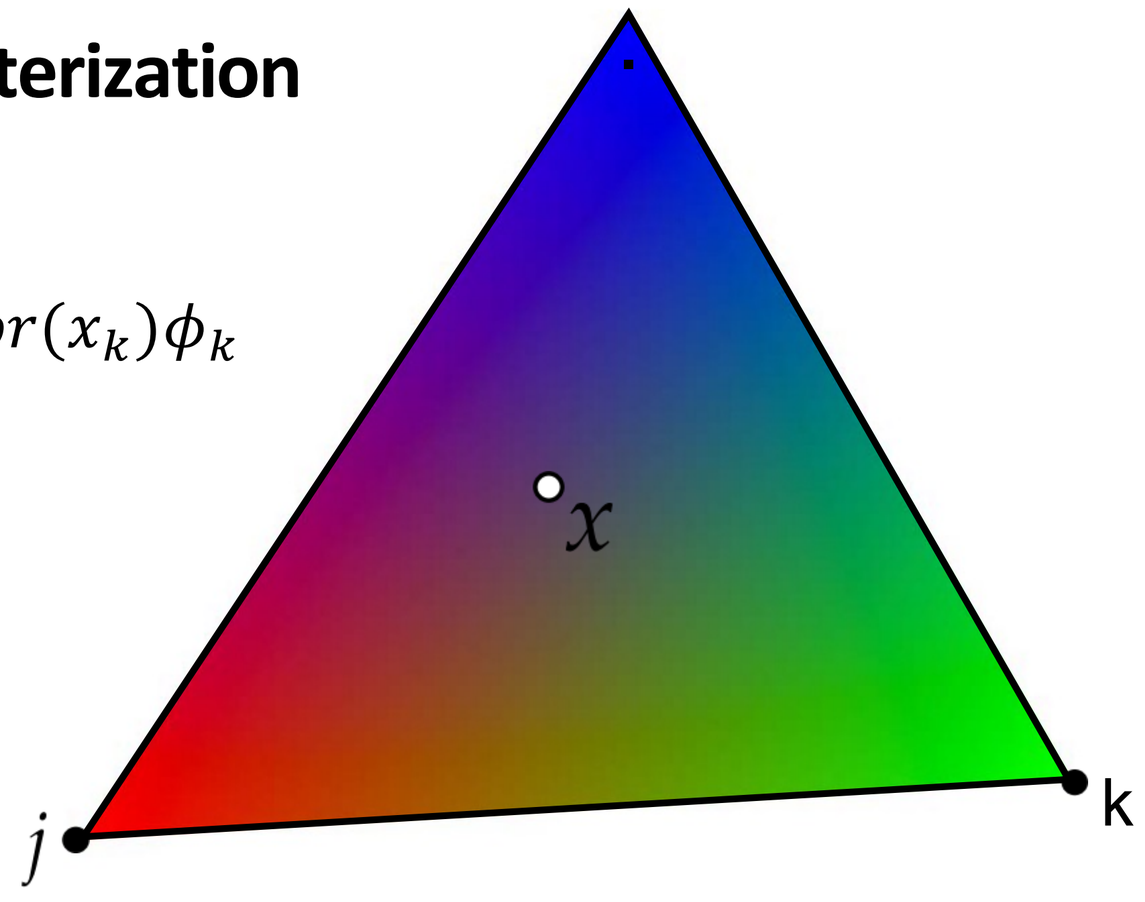
Texture Mapping

Recall: 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

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



Texture Mapping

Use of texture mapping

Define variation in surface reflectance

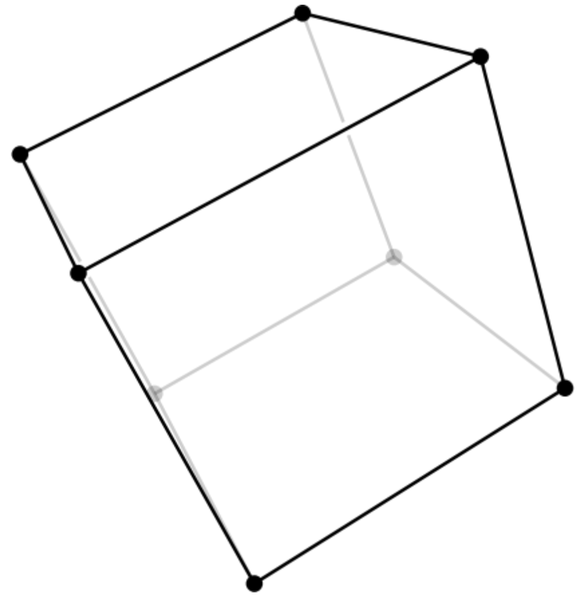
Pattern on ball

Wood grain on floor

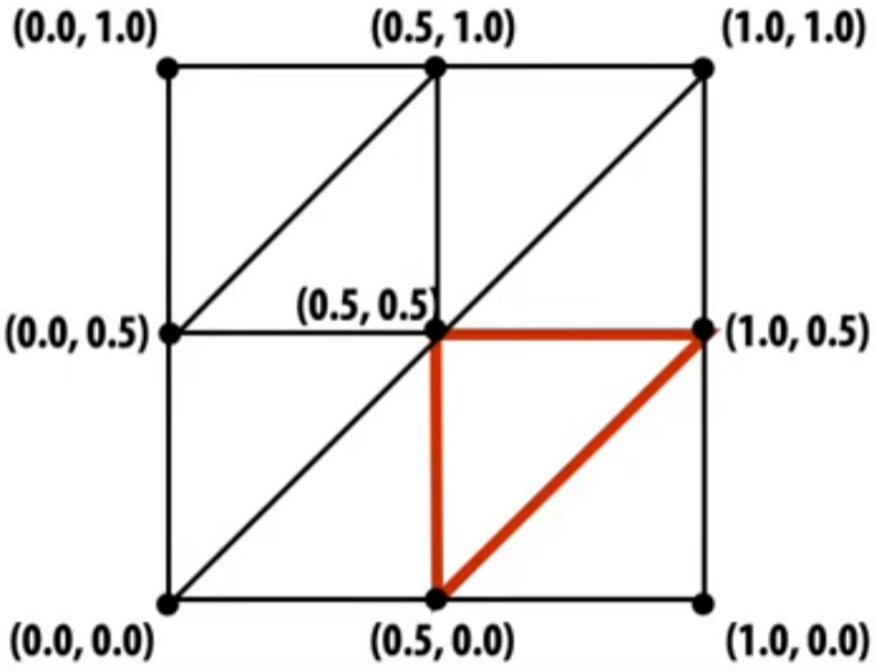
Texture coordinates

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

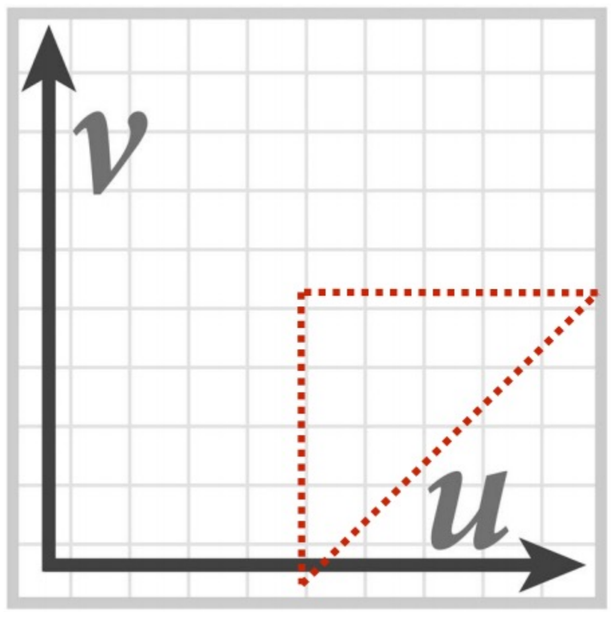
example: texture this cube



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

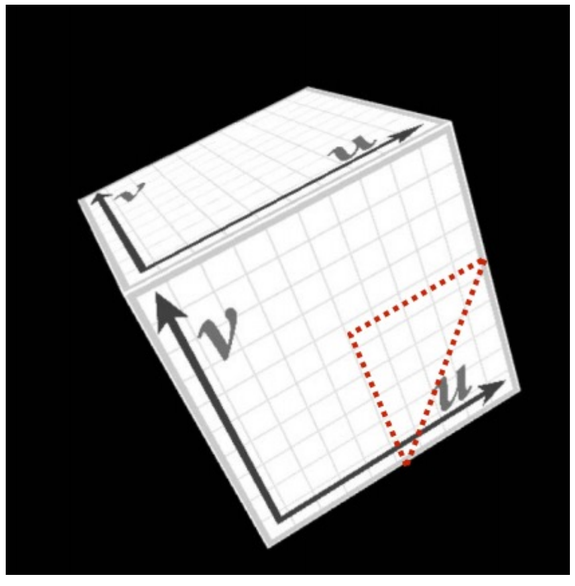


A texture on the $[0, 1]^2$ domain can be specified by a 2048x2048 image.



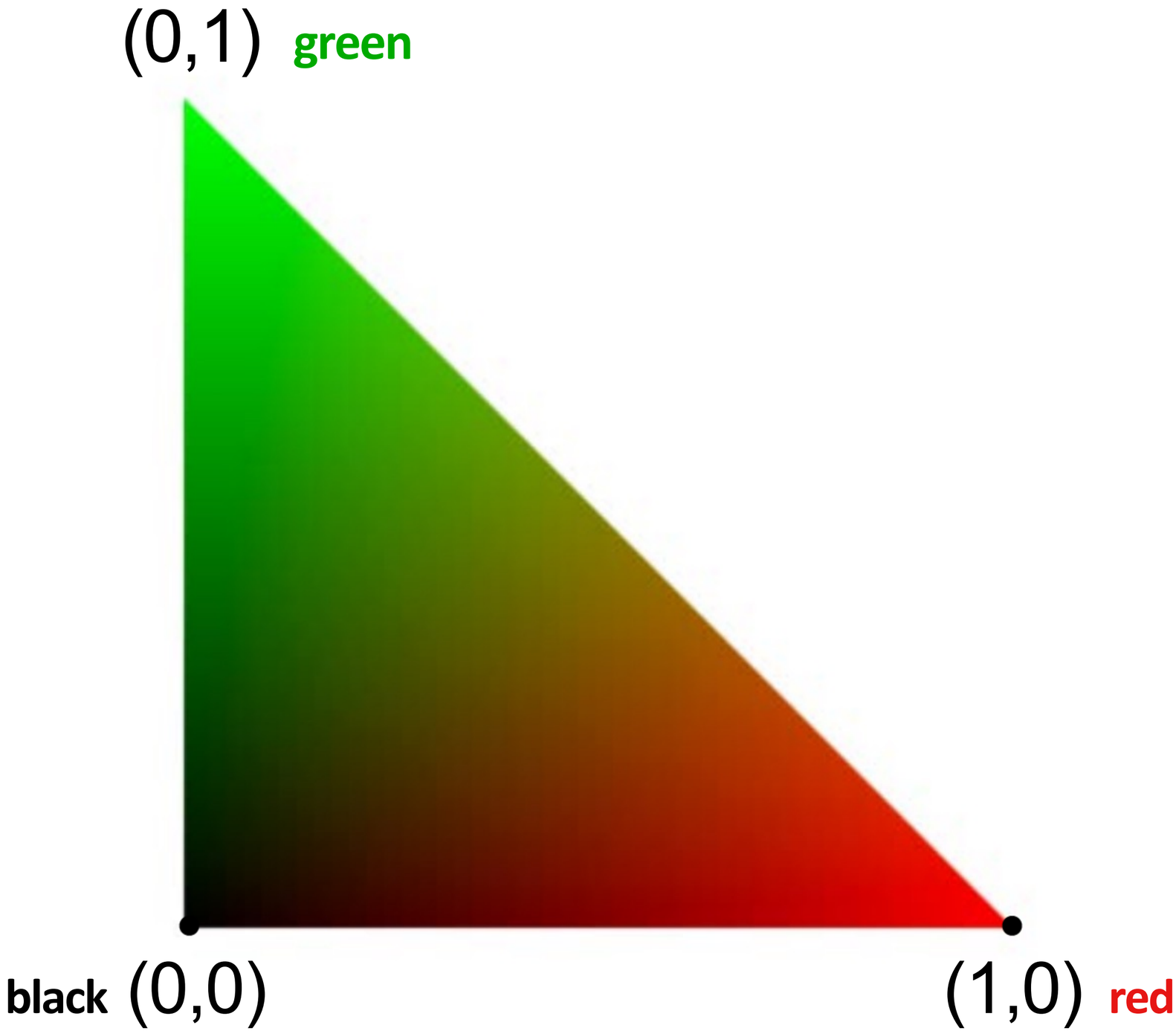
(location of highlighted triangle in texture space shown in red)

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



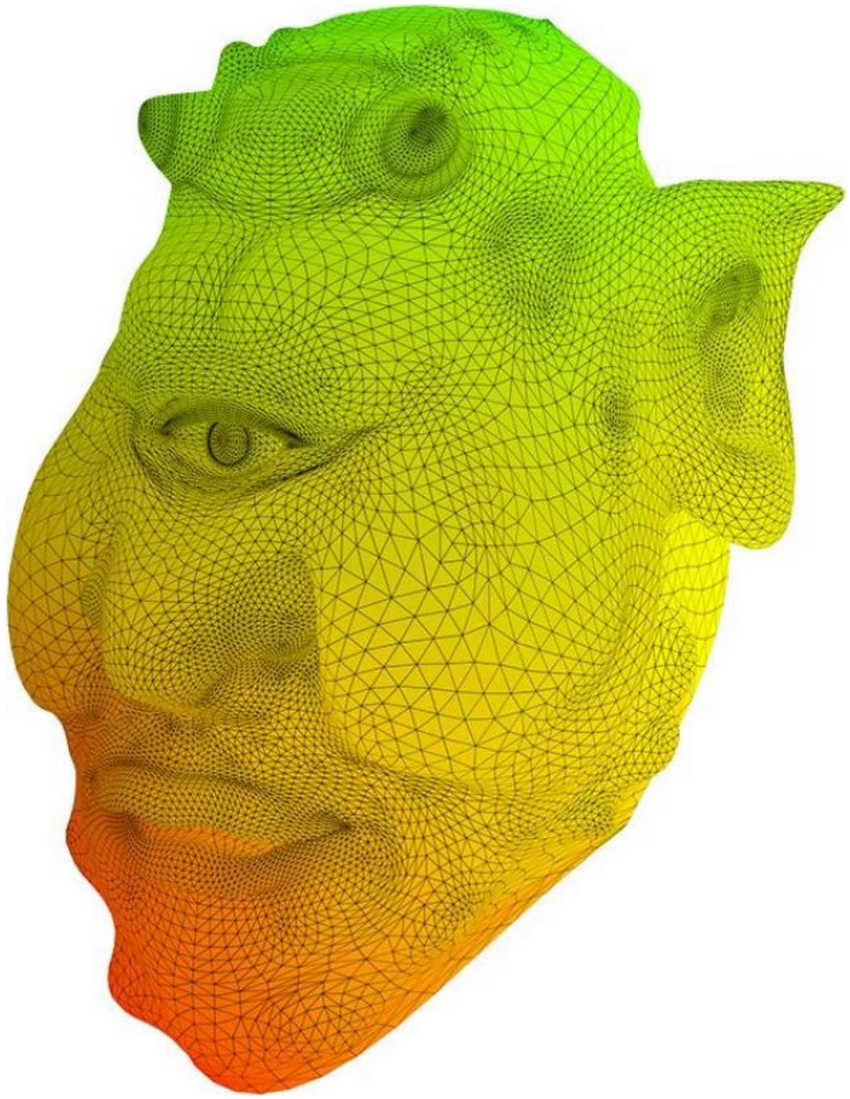
Visualization of texture coordinates

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

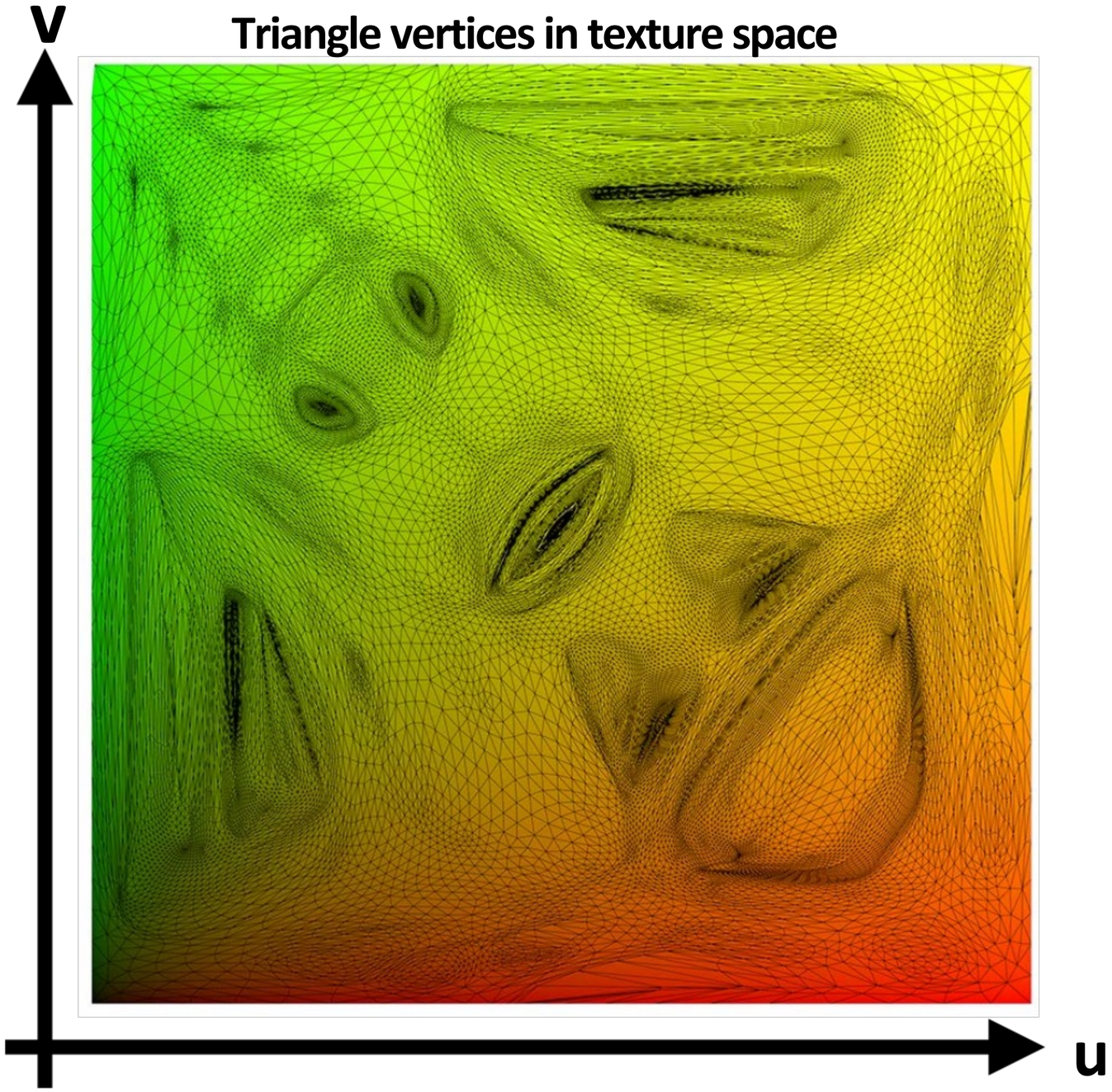


More complex mapping

Visualization of texture coordinates



Triangle vertices in texture space

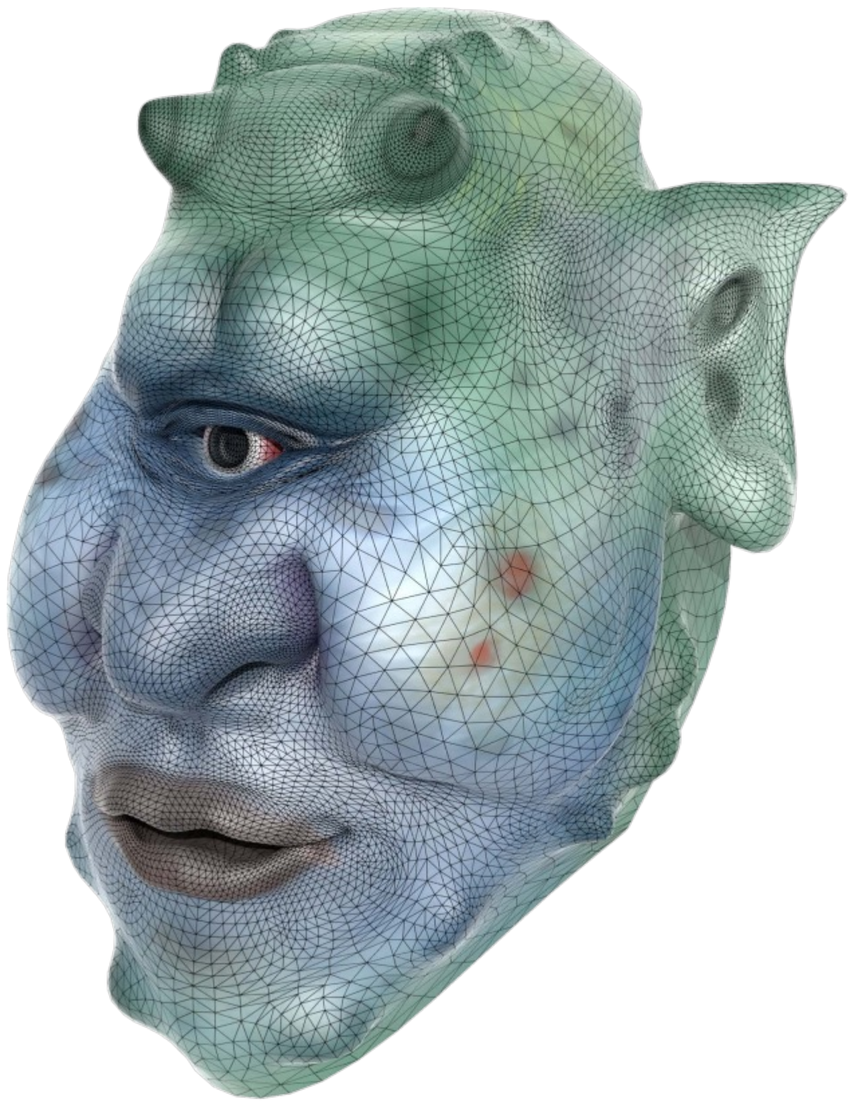


Each vertex has a coordinate (u,v) in texture space

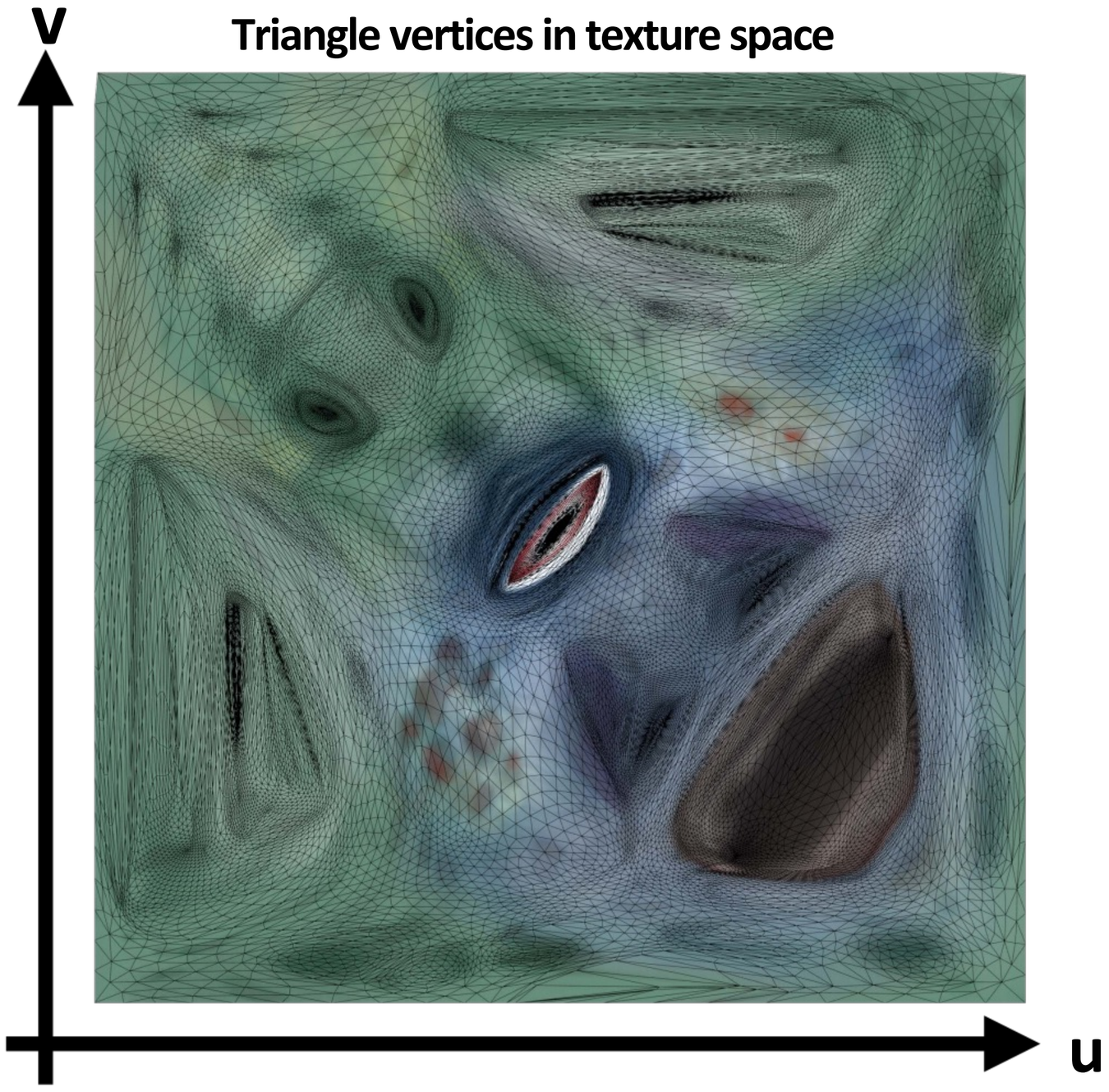
(Actually coming up with these coordinates is another story!)

Texture mapping adds detail

Rendered result



Triangle vertices in texture space



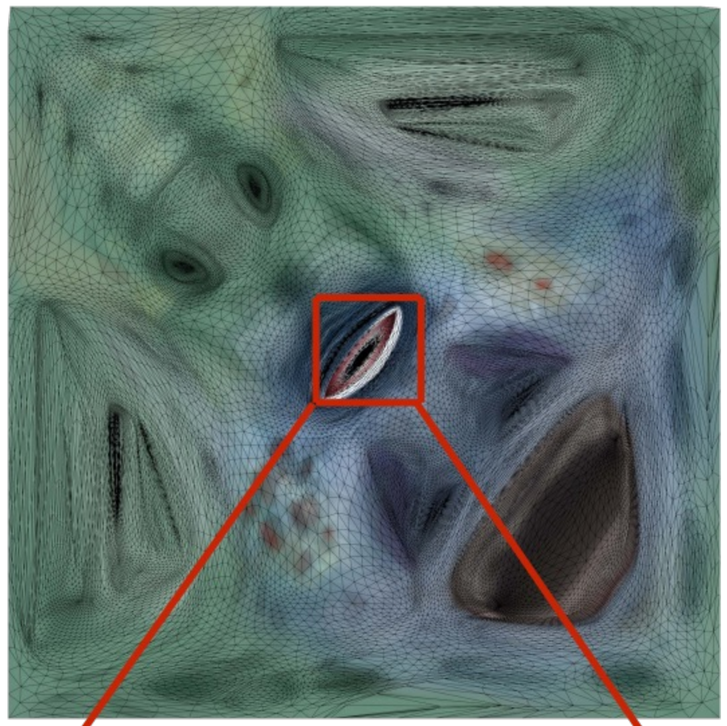
Each triangle "copies" a piece of the image back to the surface

Texture mapping adds detail

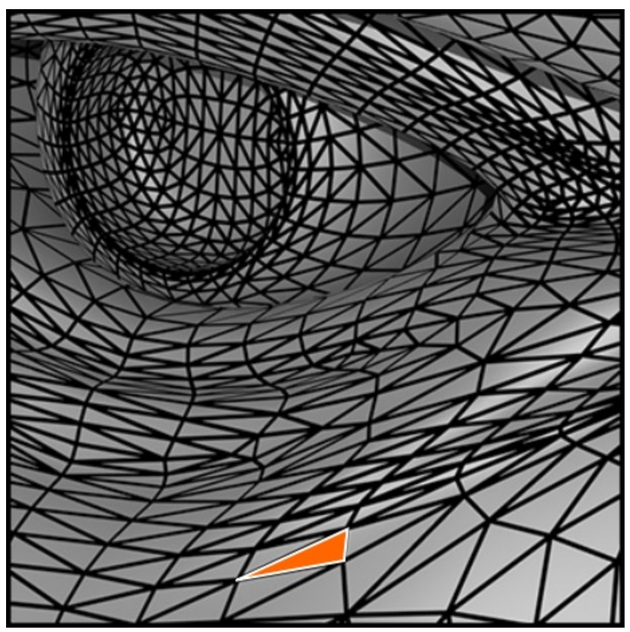
rendering without texture

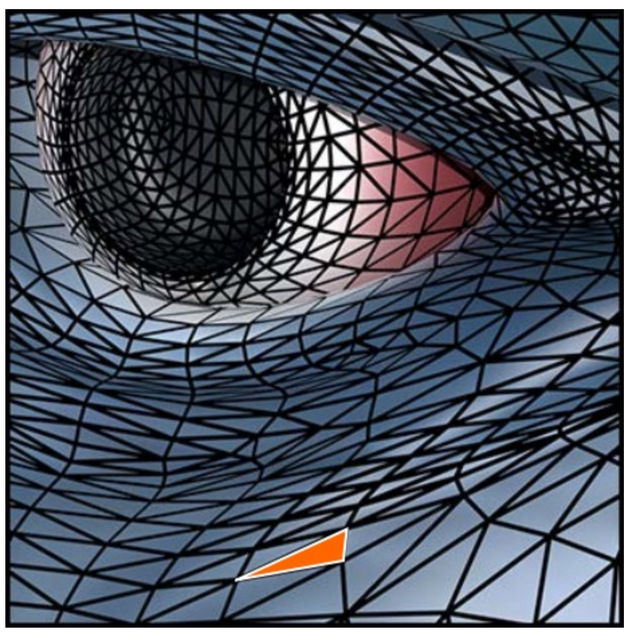
rendering with texture

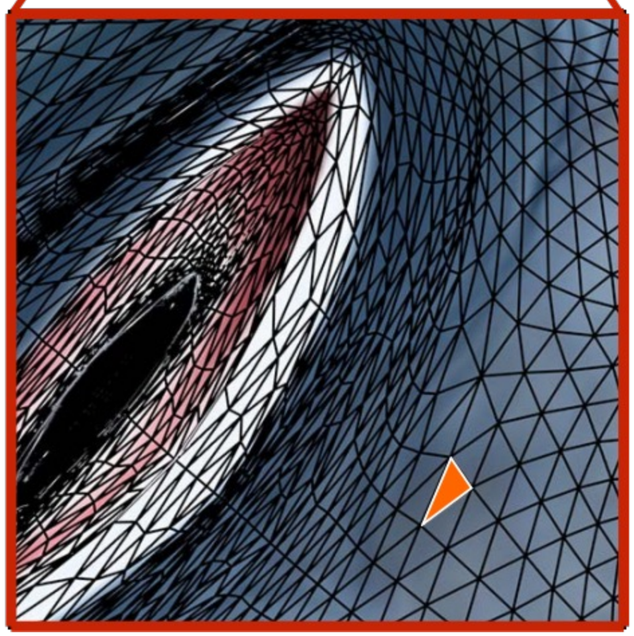
texture image



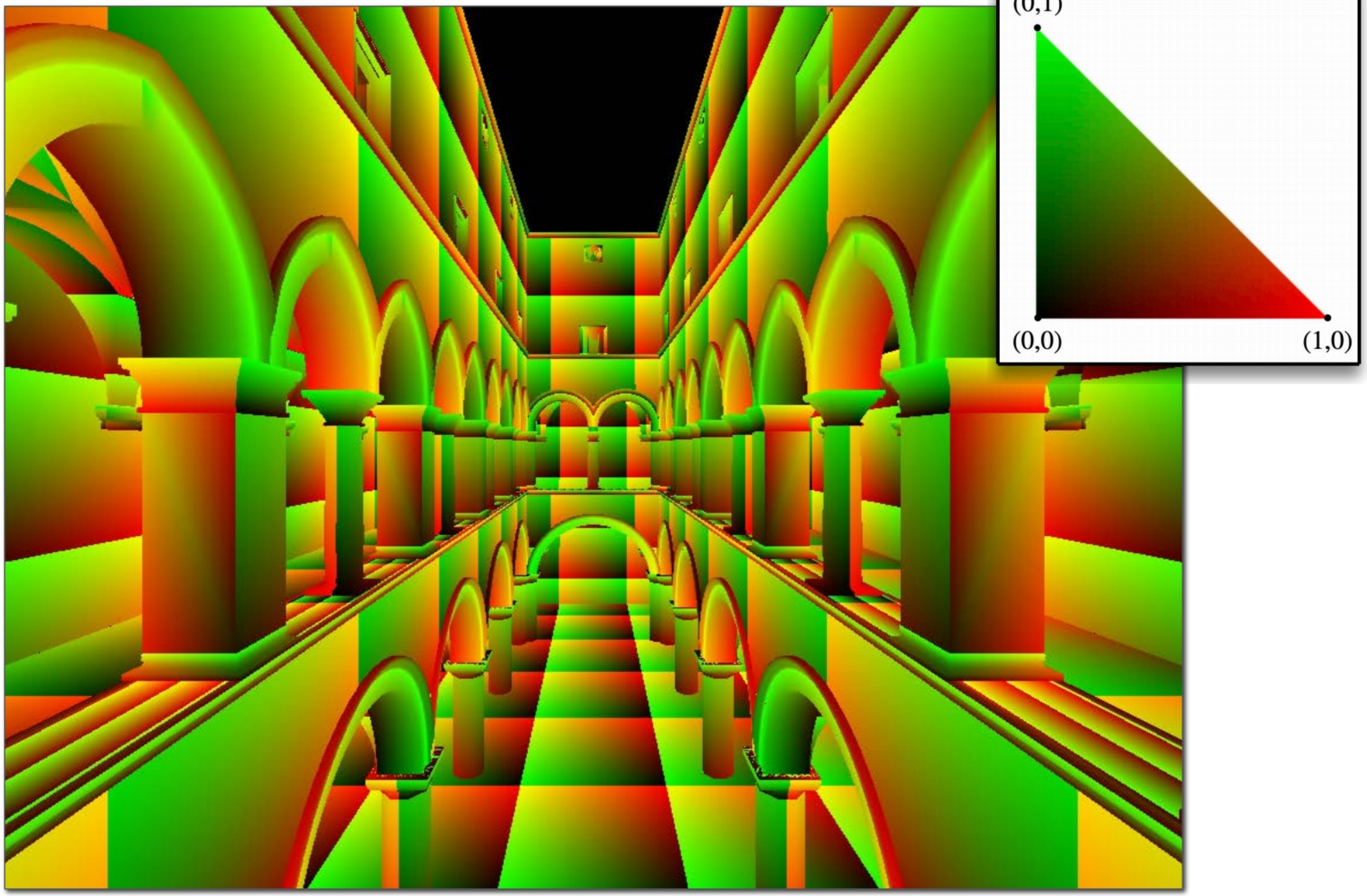
zoom





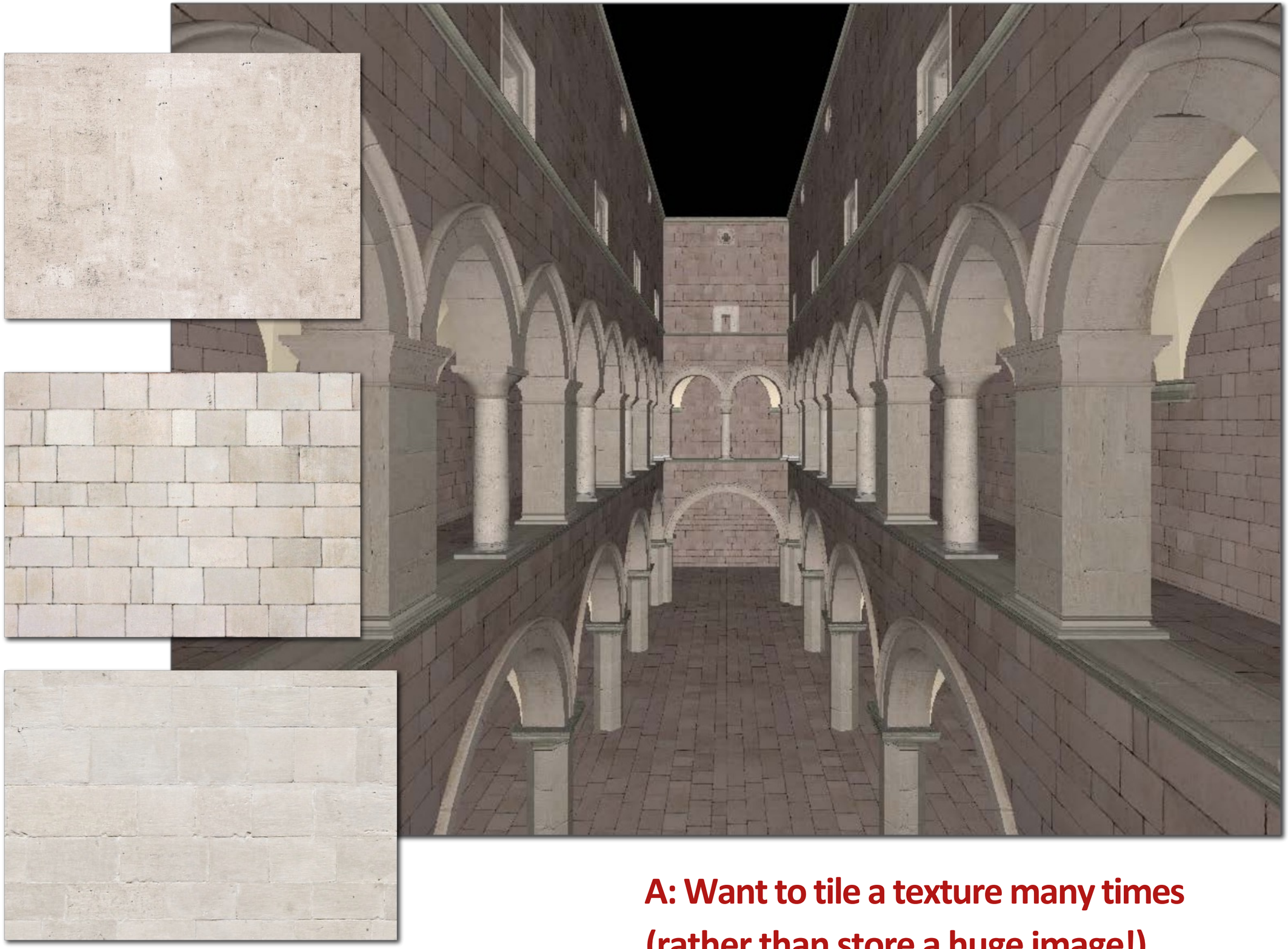


Another example: periodic coordinates



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

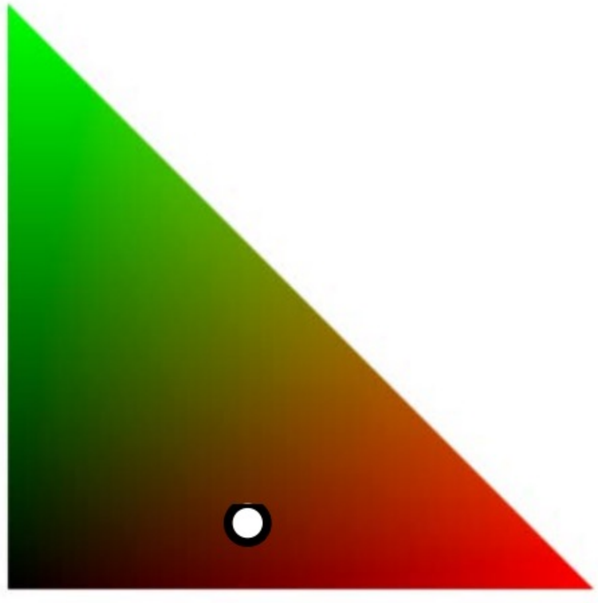
Textured Sponza

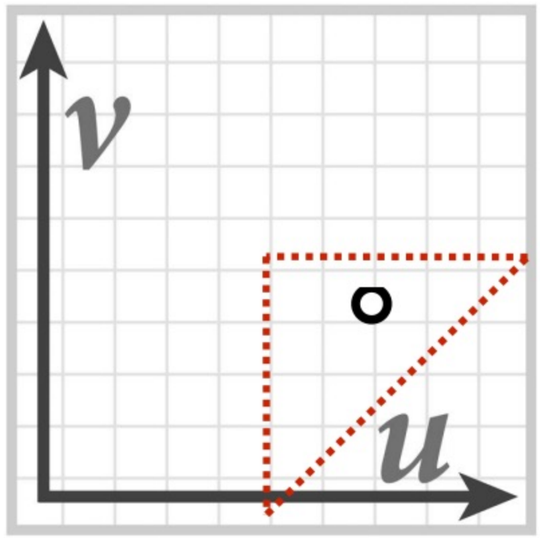


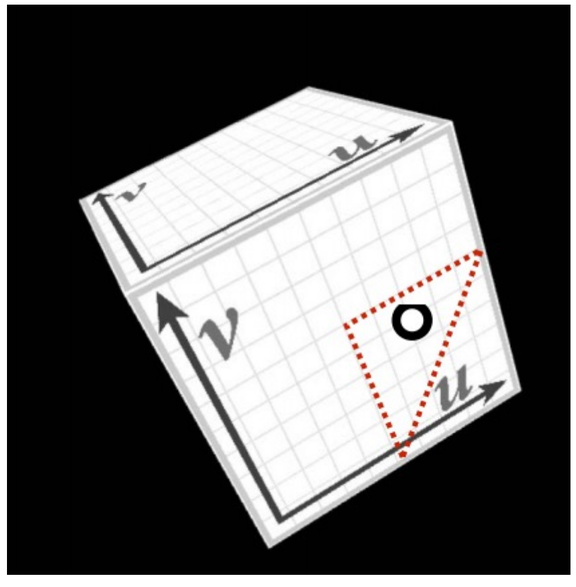
A: Want to tile a texture many times
(rather than store a huge image!)

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`



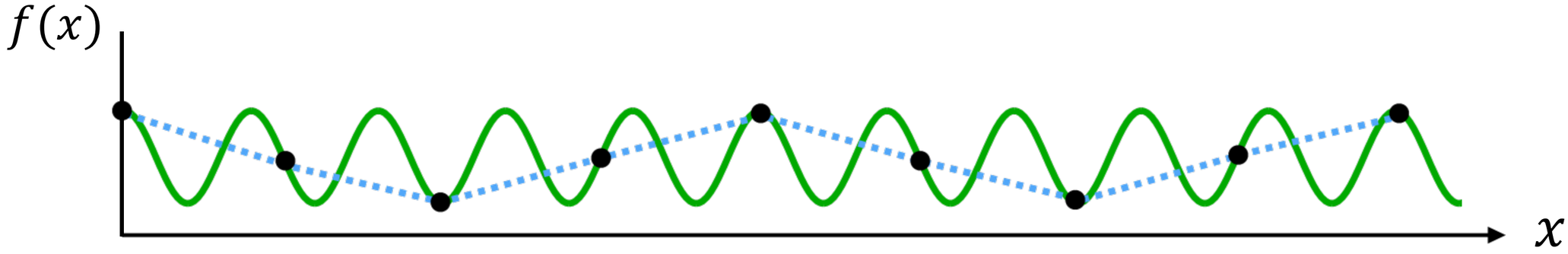


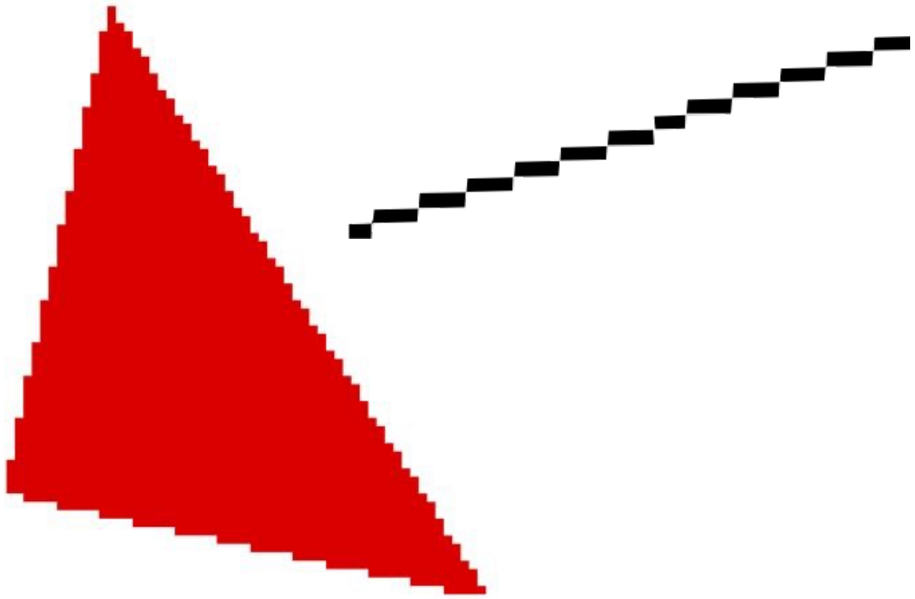


…sadly not this easy in general!

Recall: aliasing

Undersampling a high-frequency signal can result in aliasing

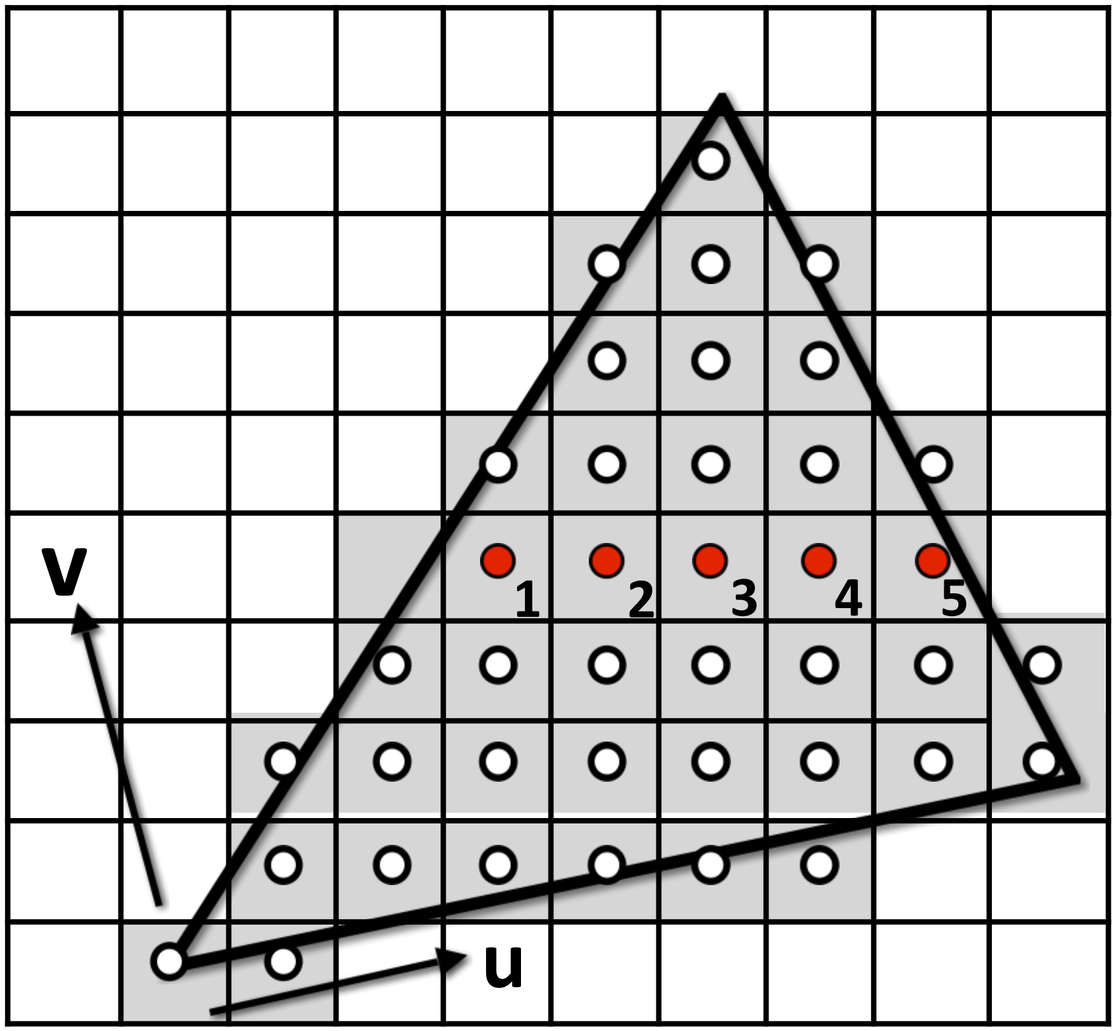




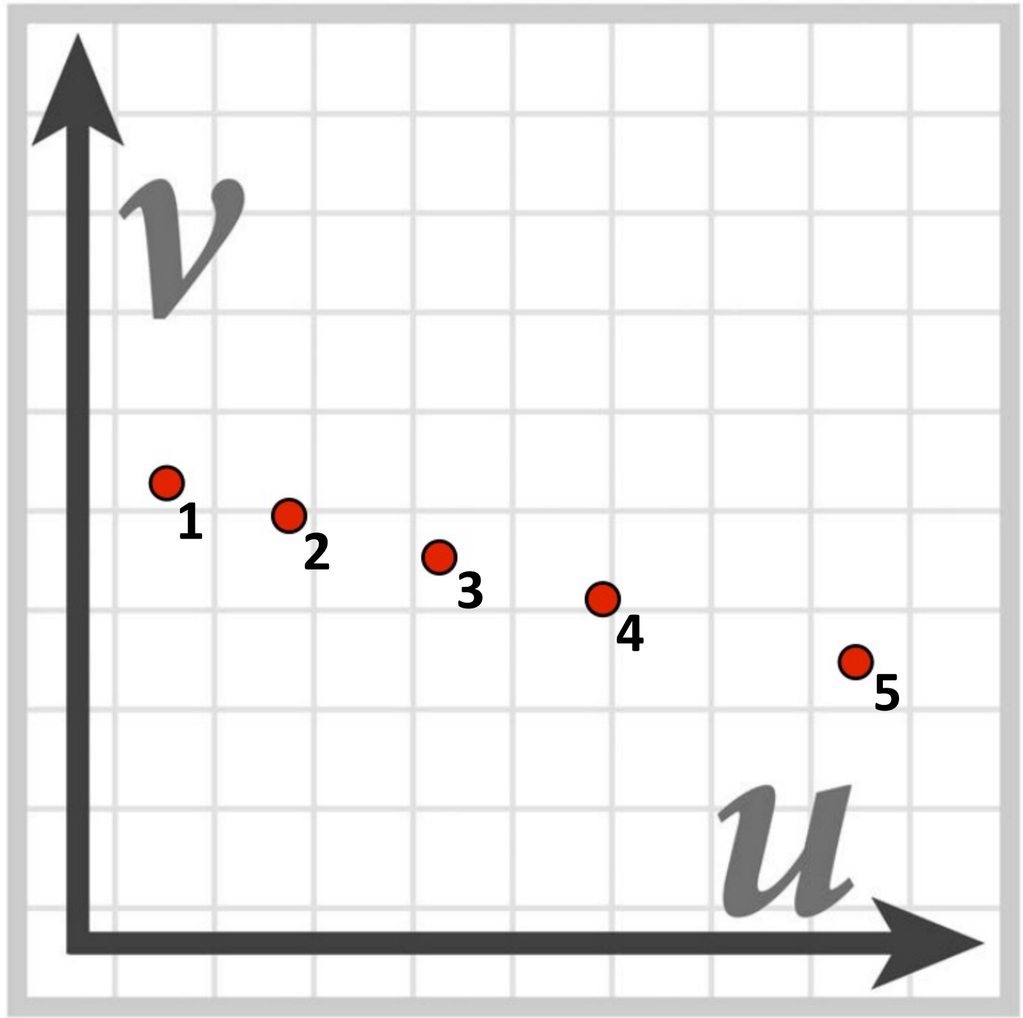
Visualizing texture samples

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 in texture space

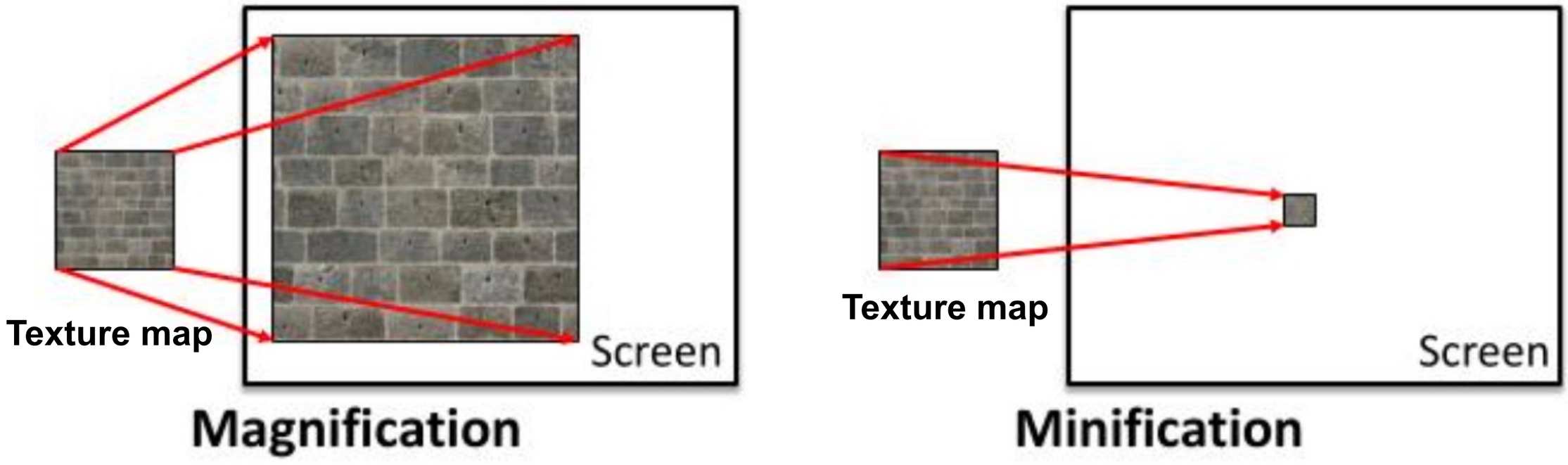


Sample positions are uniformly distributed in screen space (rasterizer samples triangle's appearance at these locations)

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

Irregular sampling pattern makes it hard to avoid aliasing!

Texture Magnification vs. Minification



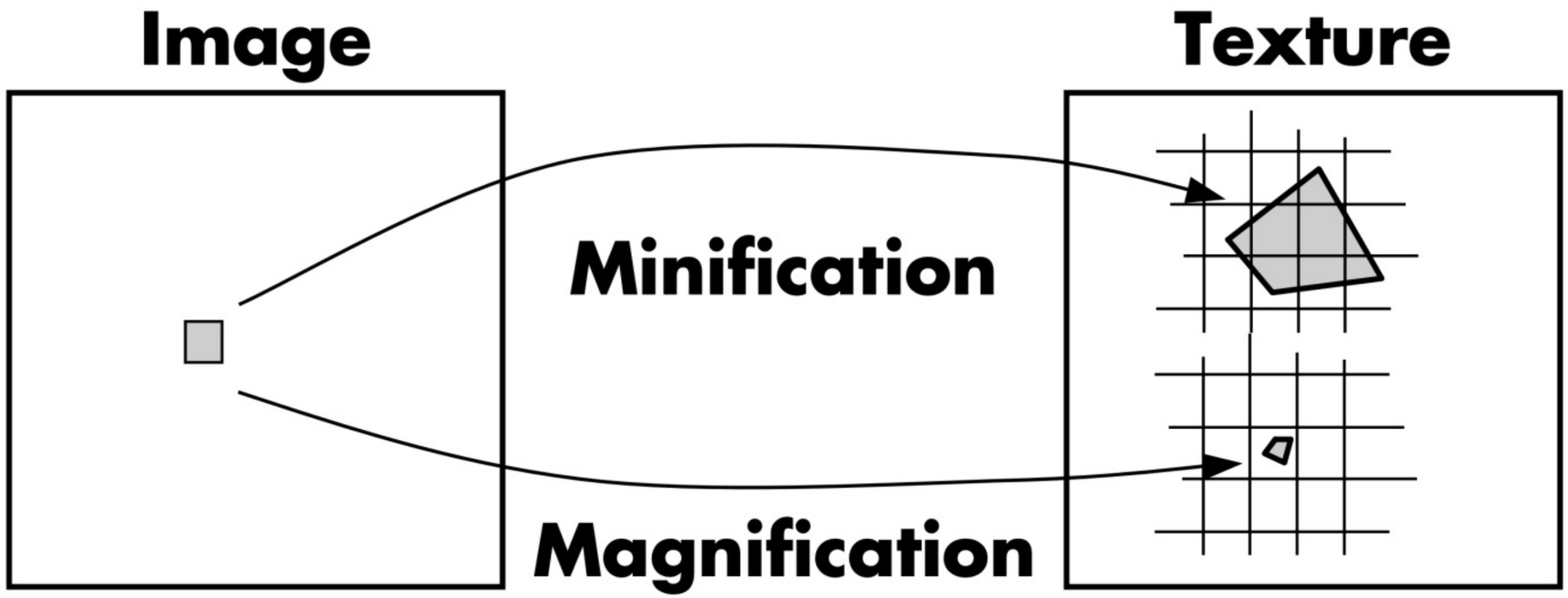
. Texture Magnification (easier):

- Example: camera is very close to scene object
- Single screen pixel maps to tiny region of texture
- Can just interpolate value at screen pixel center

. Texture Minification (harder):

- Example: scene object is very far away
- Single screen pixel maps to large region of texture
- Need to compute average texture value over pixel to avoid aliasing

Texture Magnification vs. Minification



. Texture Magnification (easier):

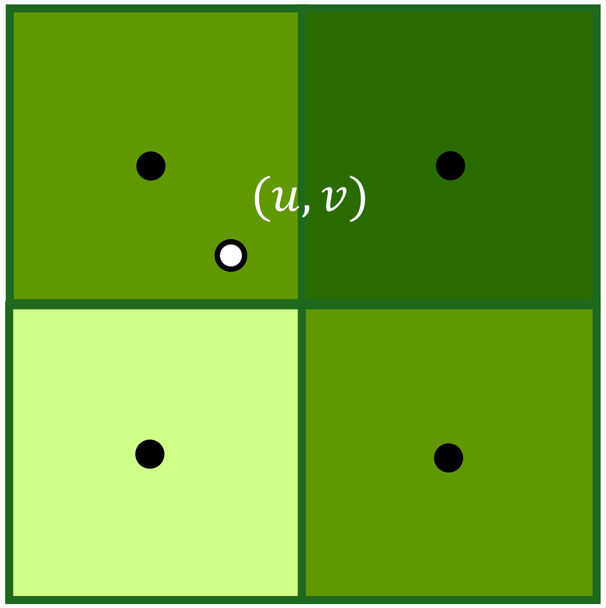
- Example: camera is very close to scene object
- Single screen pixel maps to tiny region of texture
- Can just interpolate value at screen pixel center

. Texture Minification (harder):

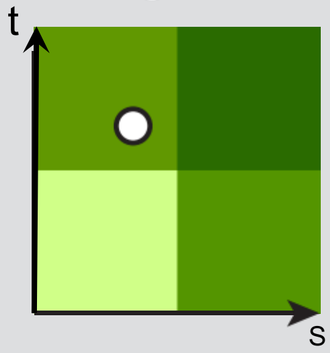
- Example: scene object is very far away
- Single screen pixel maps to large region of texture
- Need to compute average texture value over pixel to avoid aliasing

Bilinear interpolation (magnification)

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



nearest neighbor

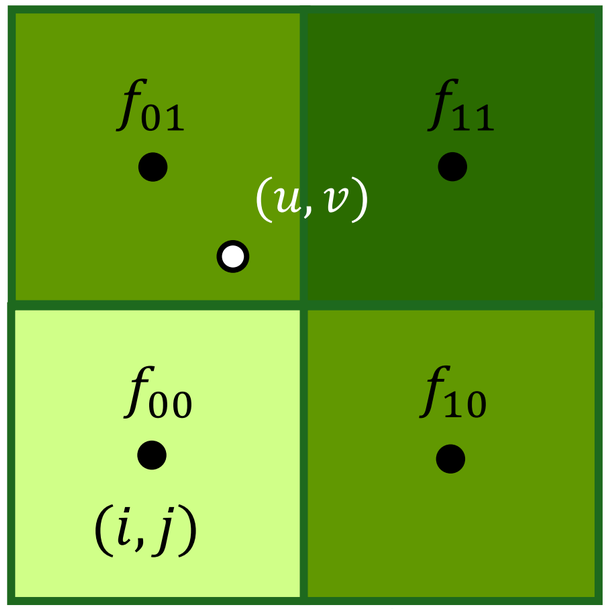


fast but ugly:

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

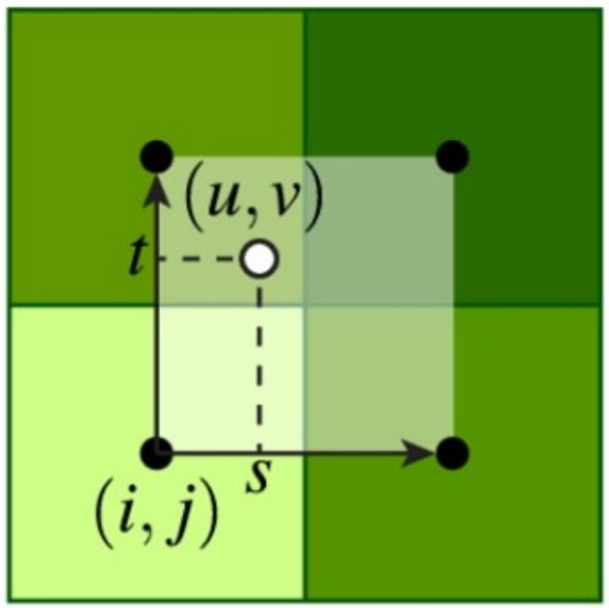
Bilinear interpolation (magnification)

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



$$i = \left\lfloor u - \frac{1}{2} \right\rfloor$$

$$j = \left\lfloor v - \frac{1}{2} \right\rfloor$$



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

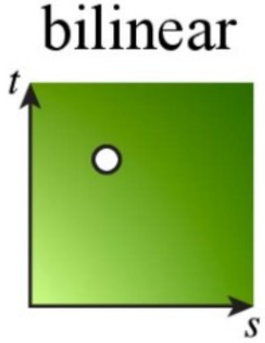
$$t = v - \left(j + \frac{1}{2}\right) \in [0,1]$$

linear (each row)

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

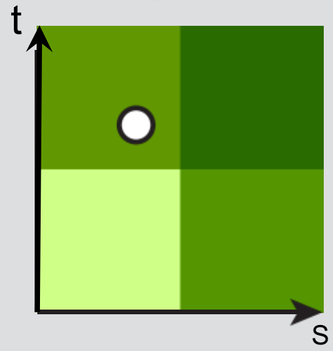
$$(1-s)f_{00} + sf_{10}$$

bilinear



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

nearest neighbor

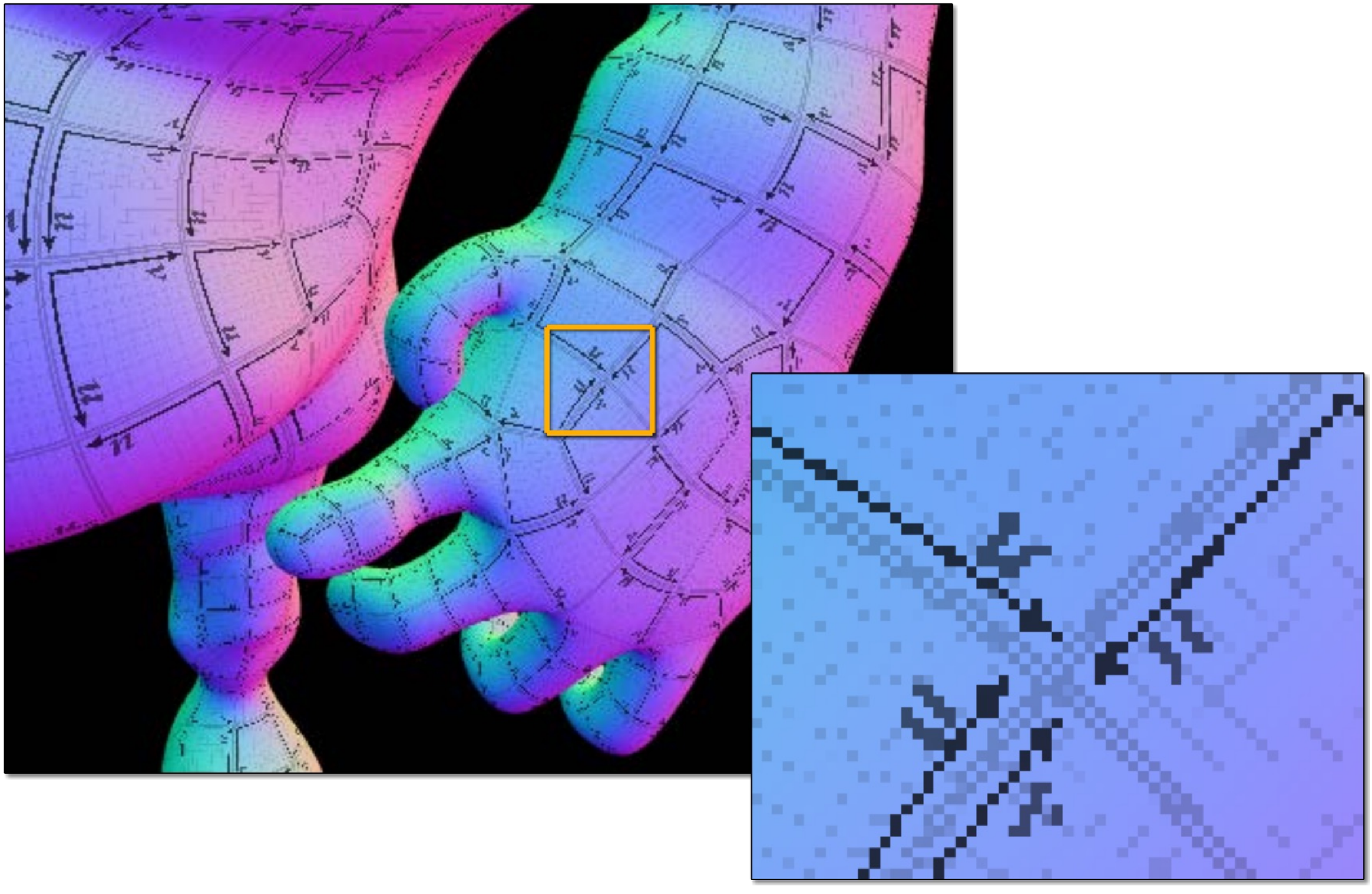


fast but ugly:

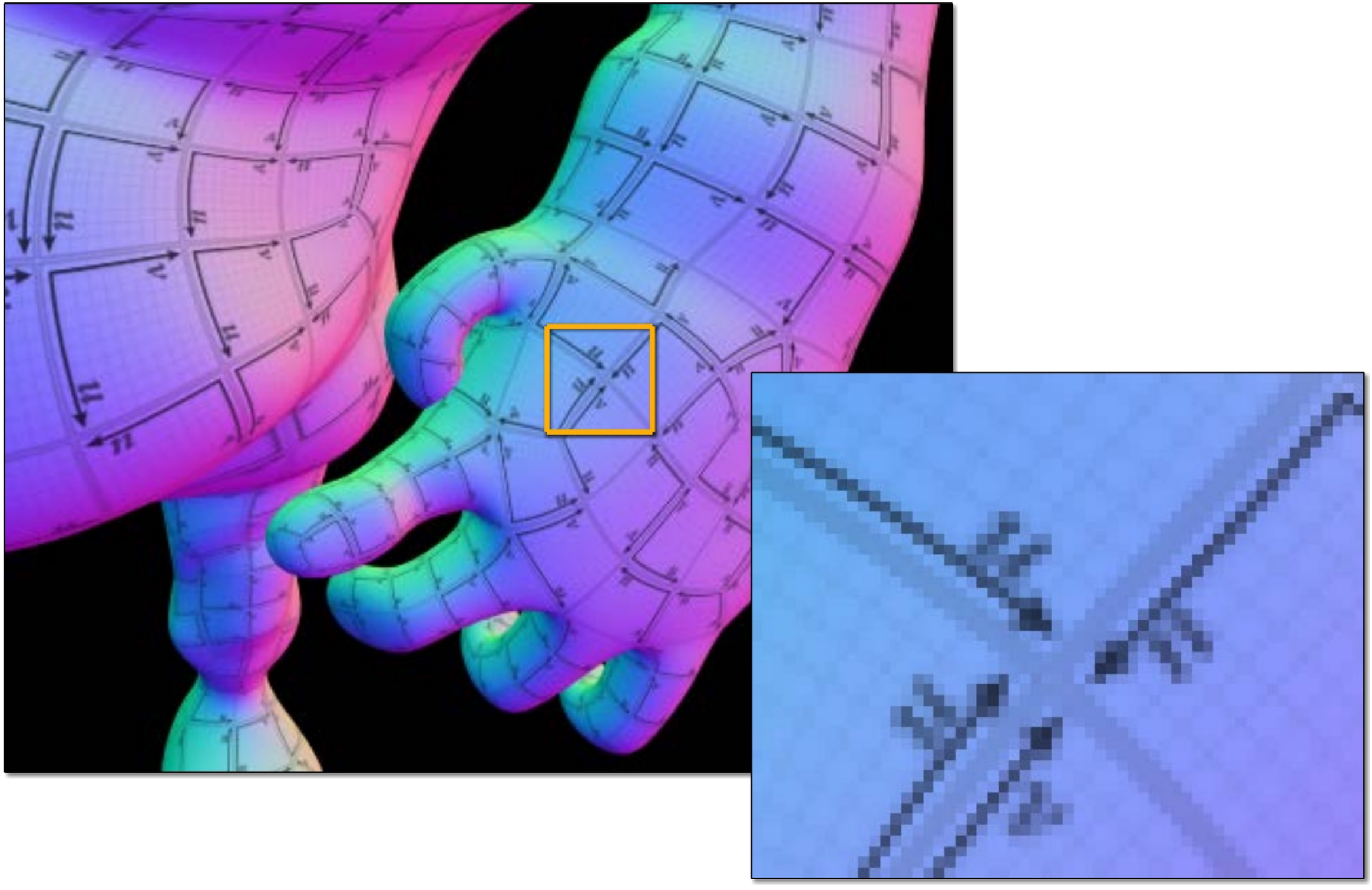
just grab value of nearest "texel" (texture pixel)

Q: What happens if we interpolate vertically first?

Aliasing due to minification



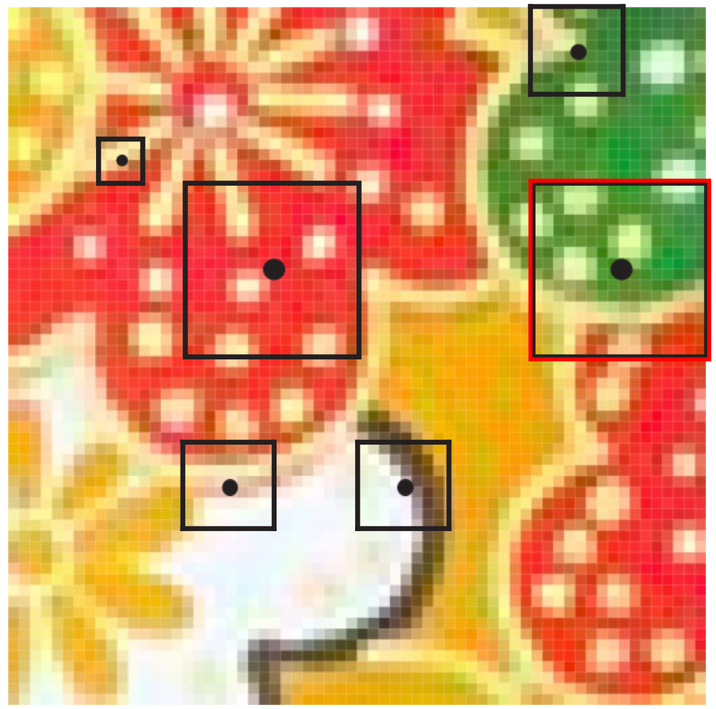
“Pre-filtering” texture (minification)



Texture prefiltering — basic idea

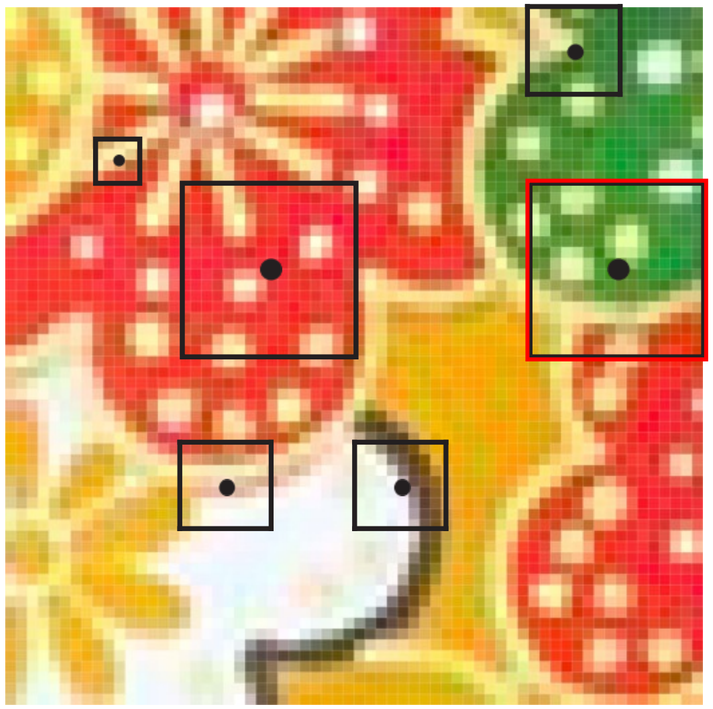
- Texture aliasing often occurs because a single pixel on the screen covers many pixels of the texture

- 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)

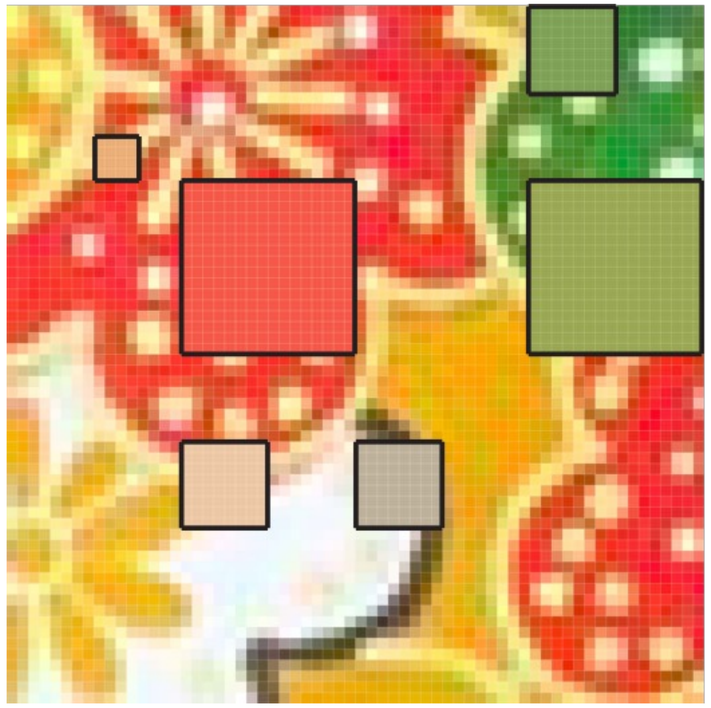


Texture prefiltering — basic idea

- Texture aliasing often occurs because a single pixel on the screen covers many pixels of the texture

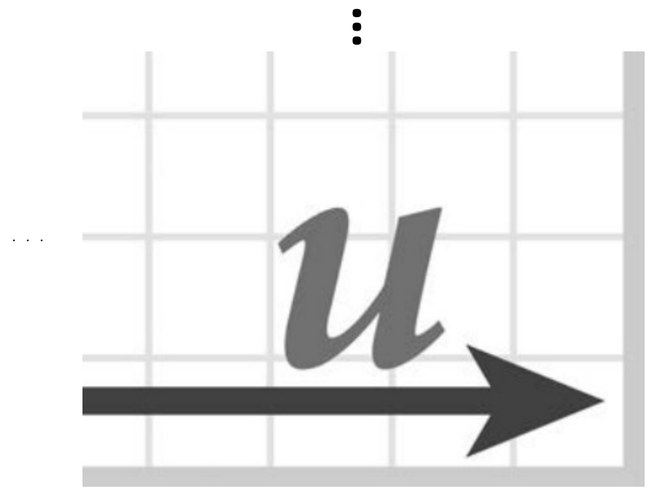


- If we just grab the texture value at the center of the pixel, we get aliasing (get a "random" color that changes if the sample moves even very slightly)
- Ideally, would use the *average* texture value—but this is expensive to compute
- Instead, we can pre-compute the averages (once) and just lookup these averages (many times) at run-time



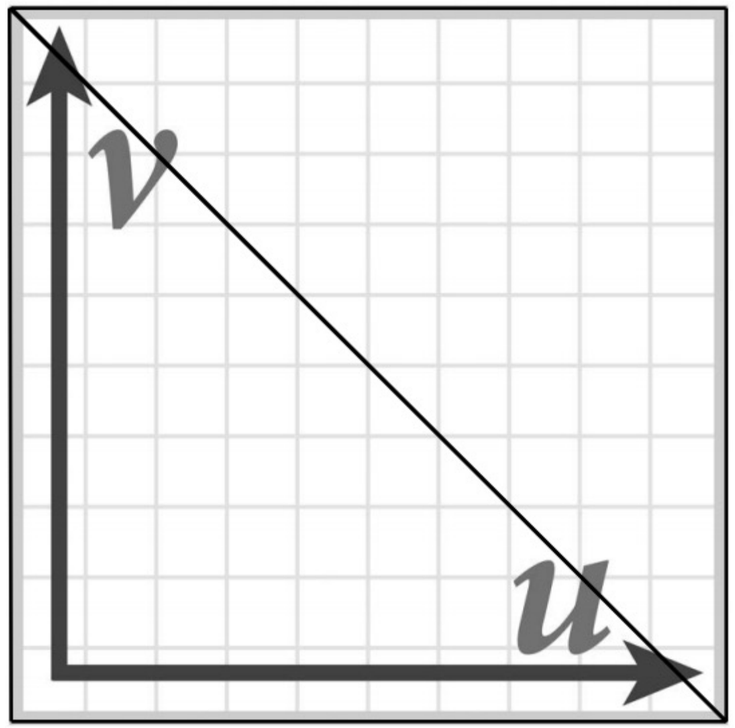
But which averages should we store? Can't precompute them all!

Prefiltered textures

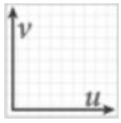


Actual texture: 700x700 image

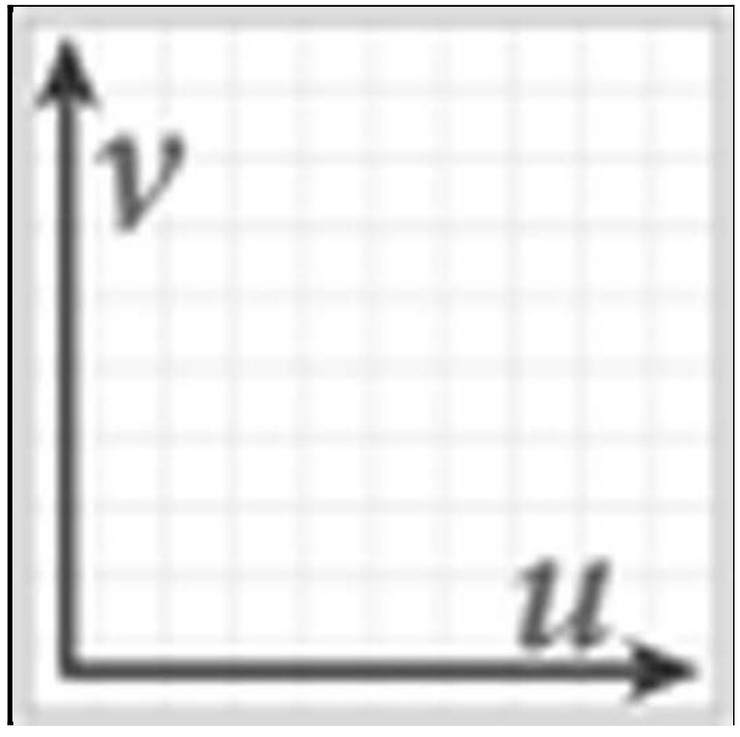
(only a crop is shown)



Texture minification



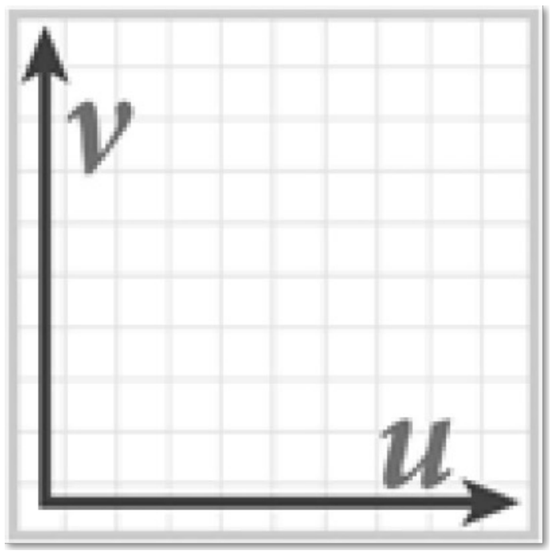
Actual texture: 64x64 image



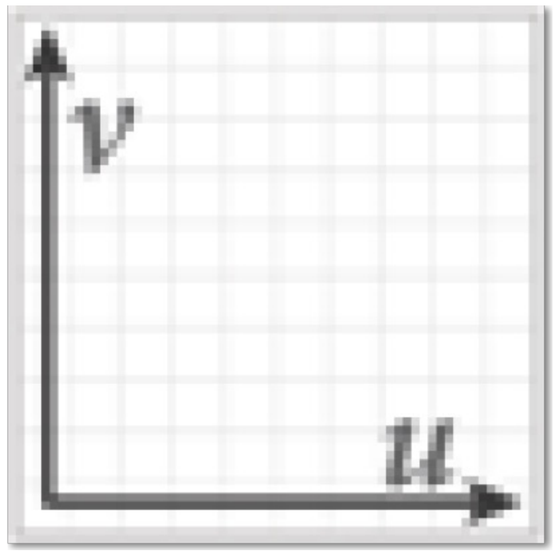
Texture magnification

Q: Are two resolutions enough? A:No ….

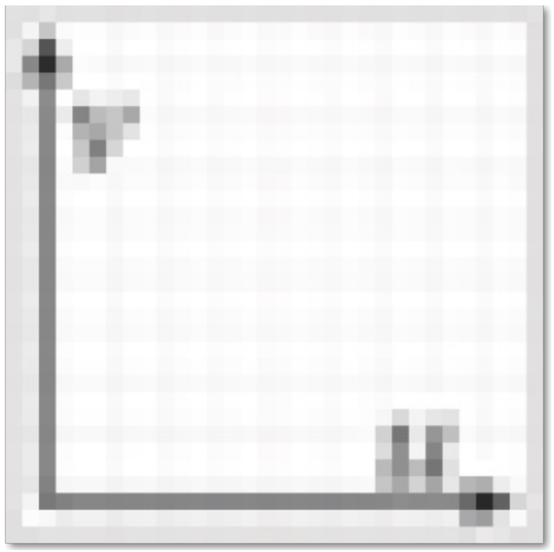
MIP map (L. Williams 83)



Level 0 = 128x128



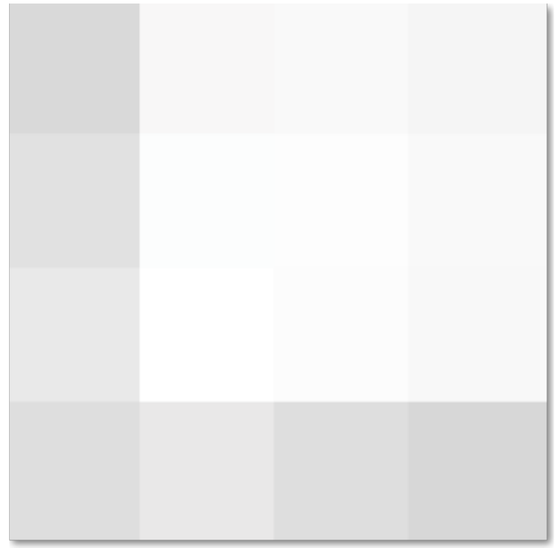
Level 1 = 64x64



Level 2 = 32x32

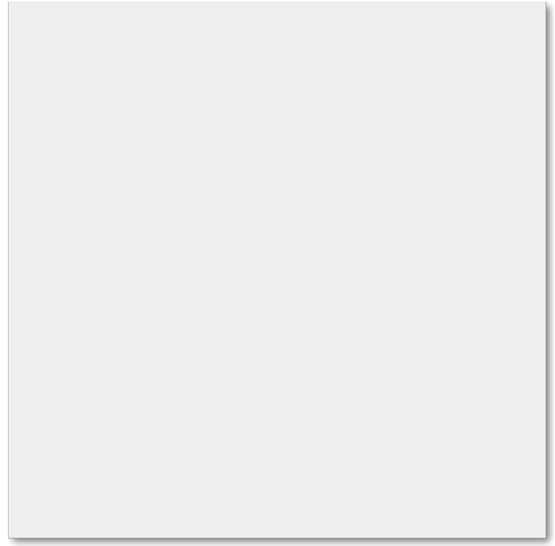
Level 3 = 16x16

Level 4 = 8x8



Level 5 = 4x4

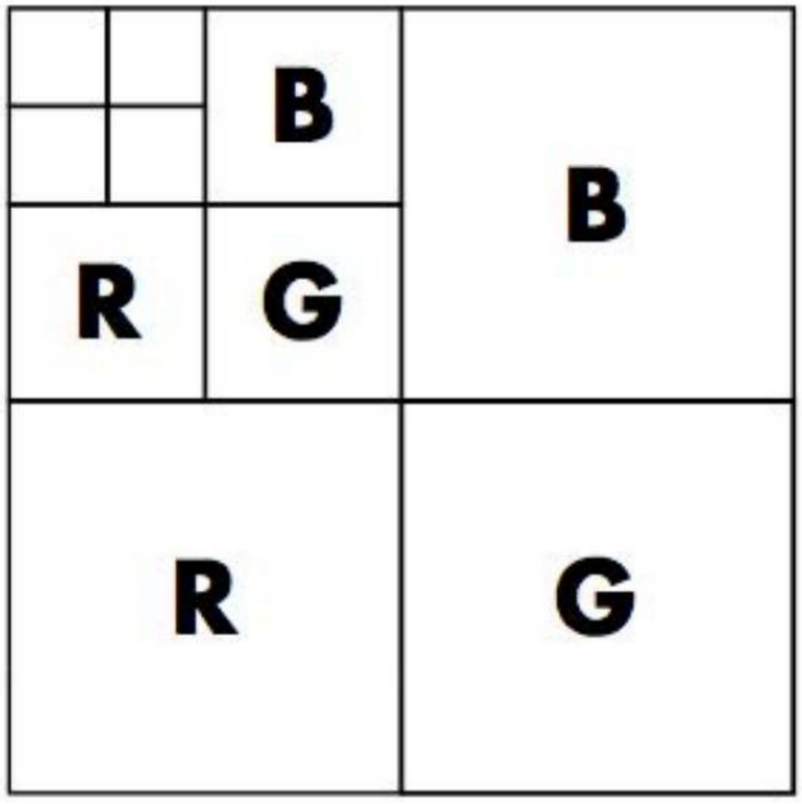
Level 6 = 2x2



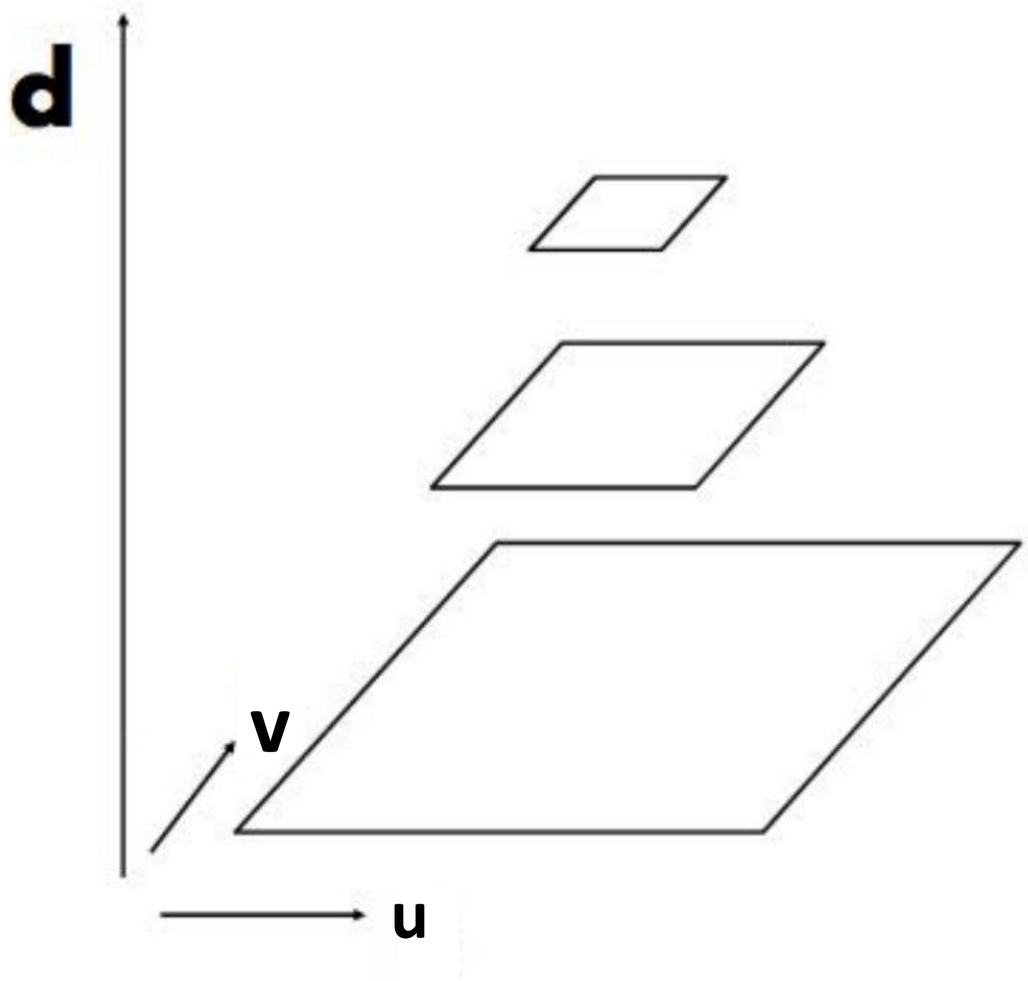
Level 7 = 1x1

- 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: lookup a single pixel from MIP map of appropriate size

Mipmap (L. Williams 83)



Williams' original proposed
mip-map layout



"Mip hierarchy"
level = d

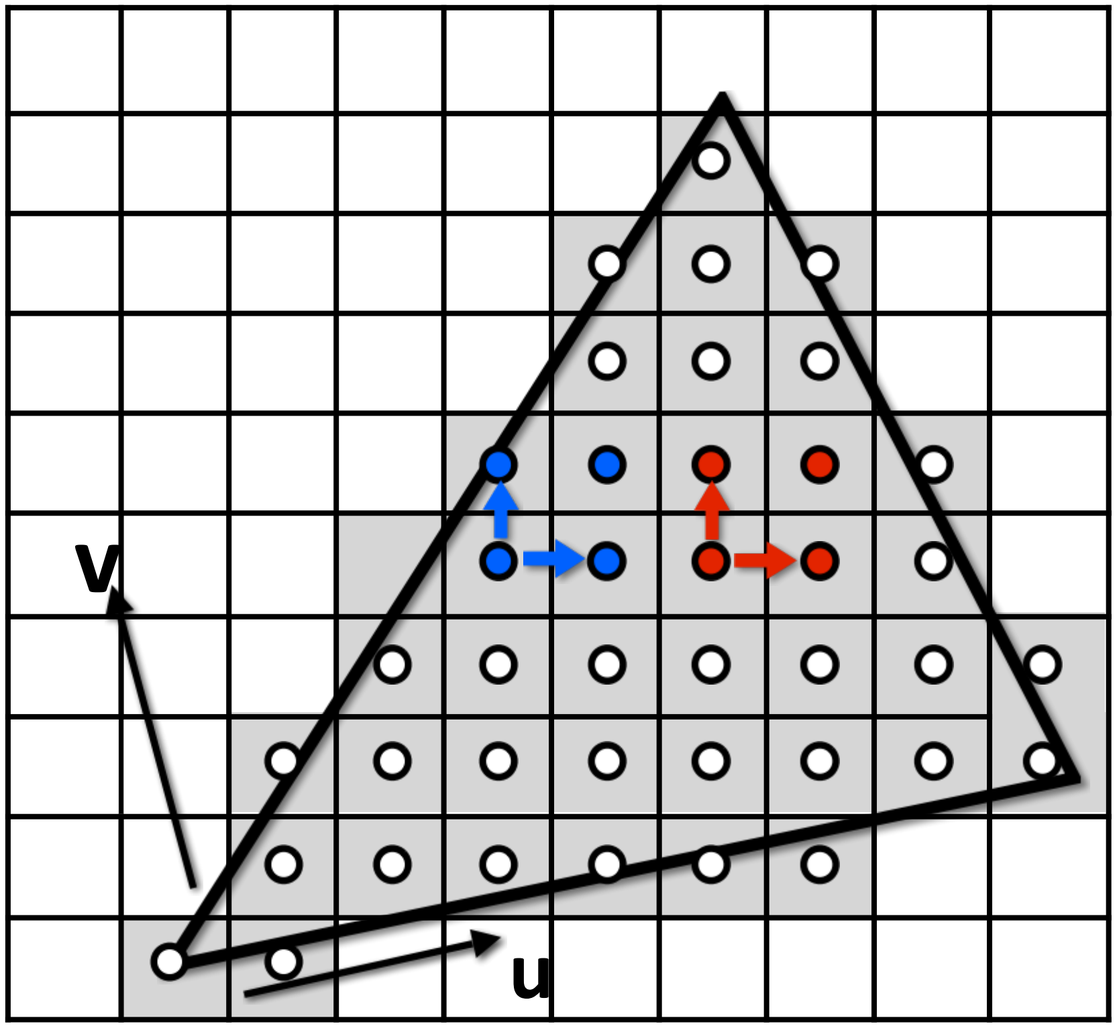
Q: What's the storage overhead of a mipmap?

A: 1/3 of the original texture map storage

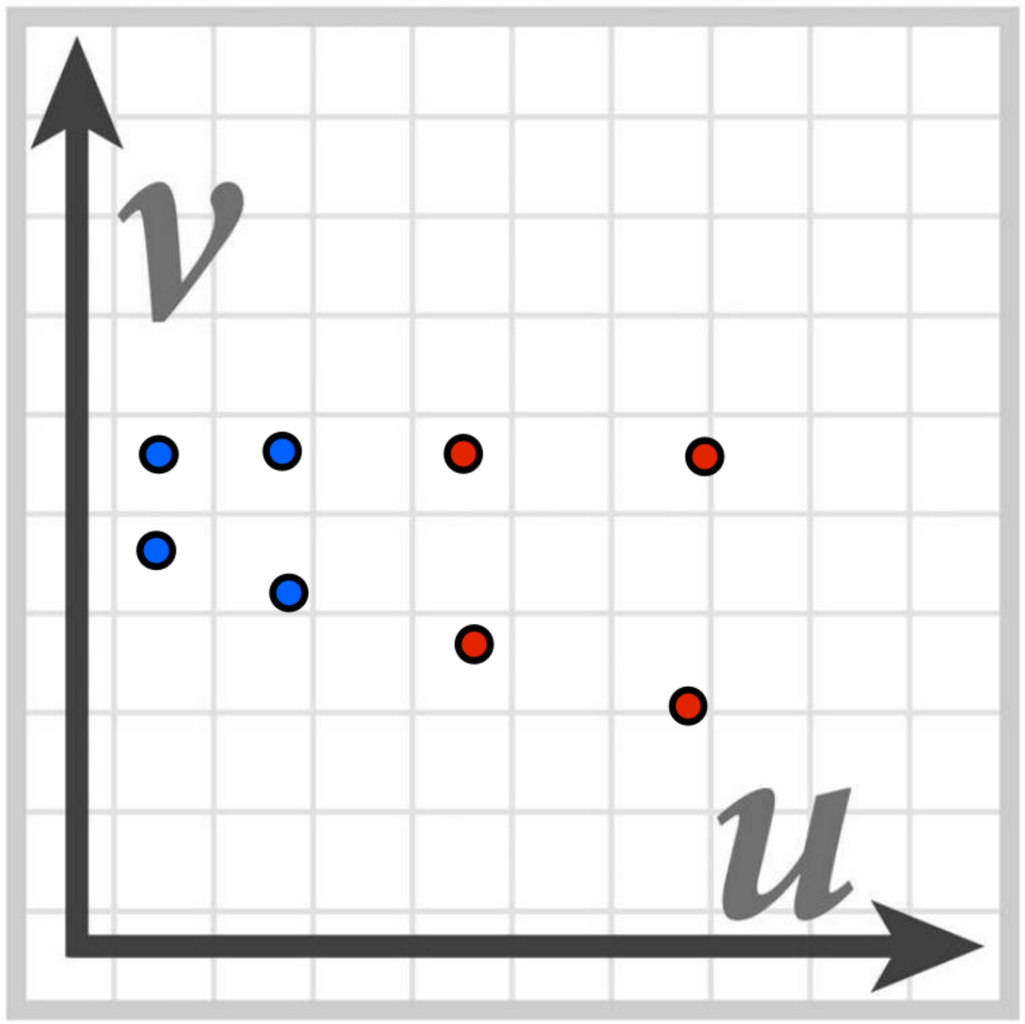
Slide credit: Akeley and Hanrahan

Computing MIP Map Level

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



Screen space

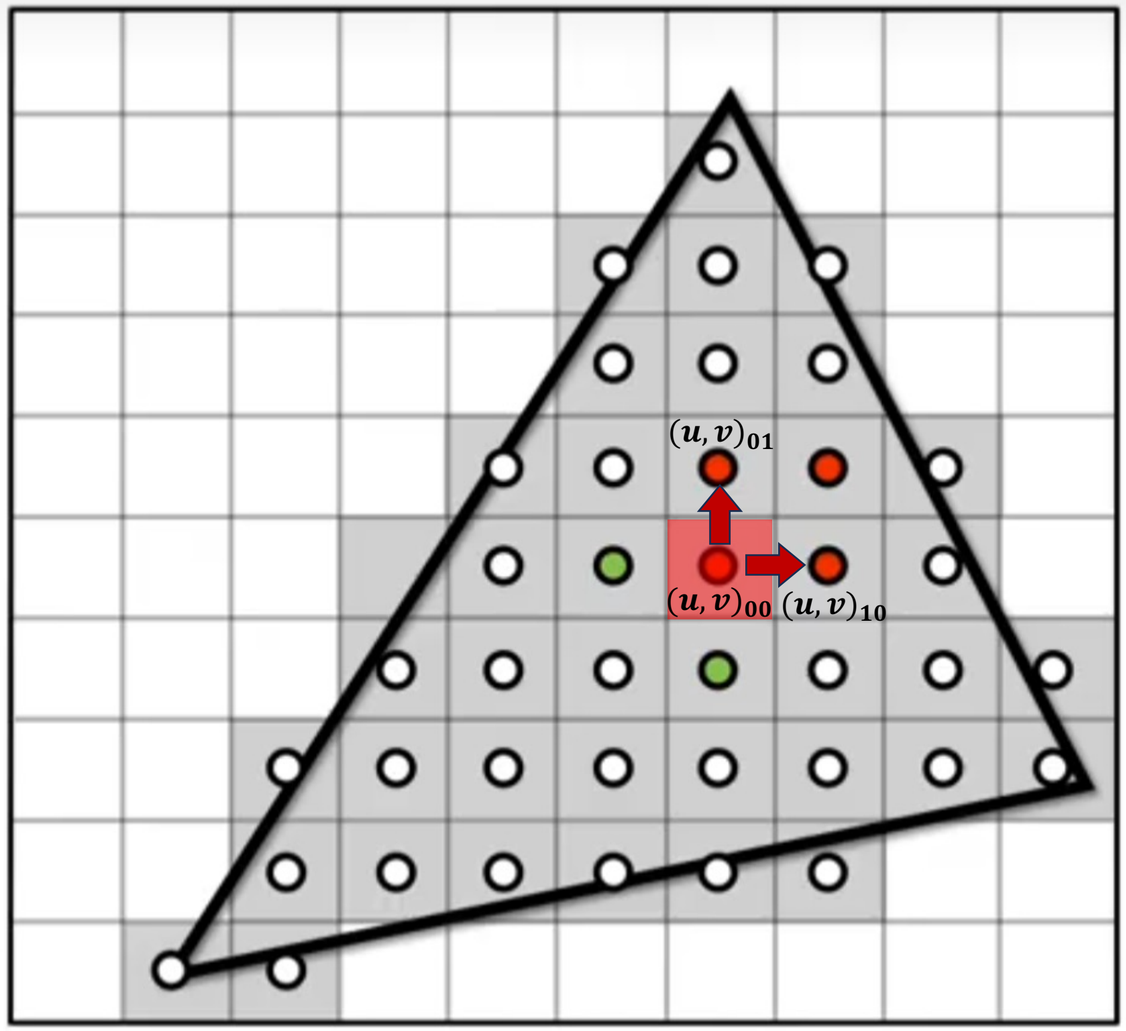


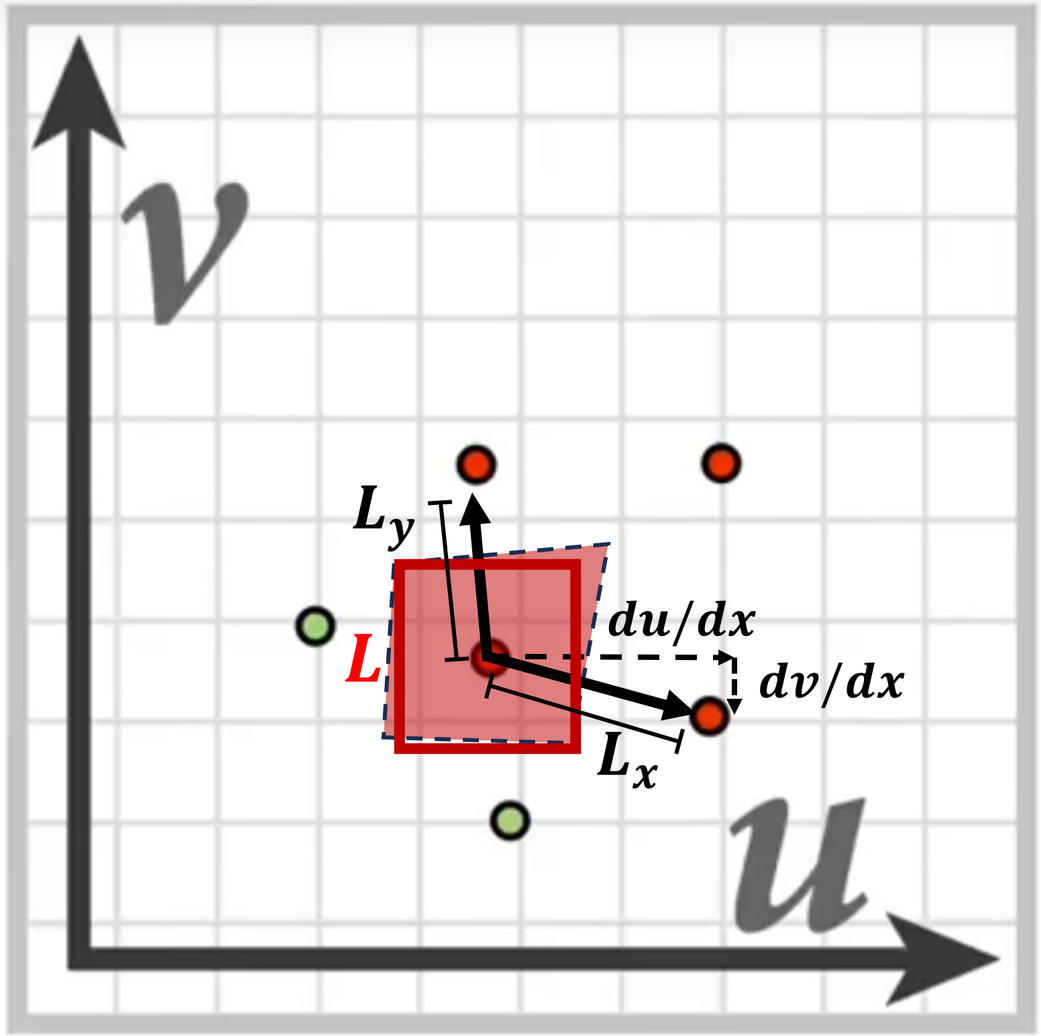
Texture space

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

Computing Mip Map Level

Compute differences between texture coordinate values at neighboring samples





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

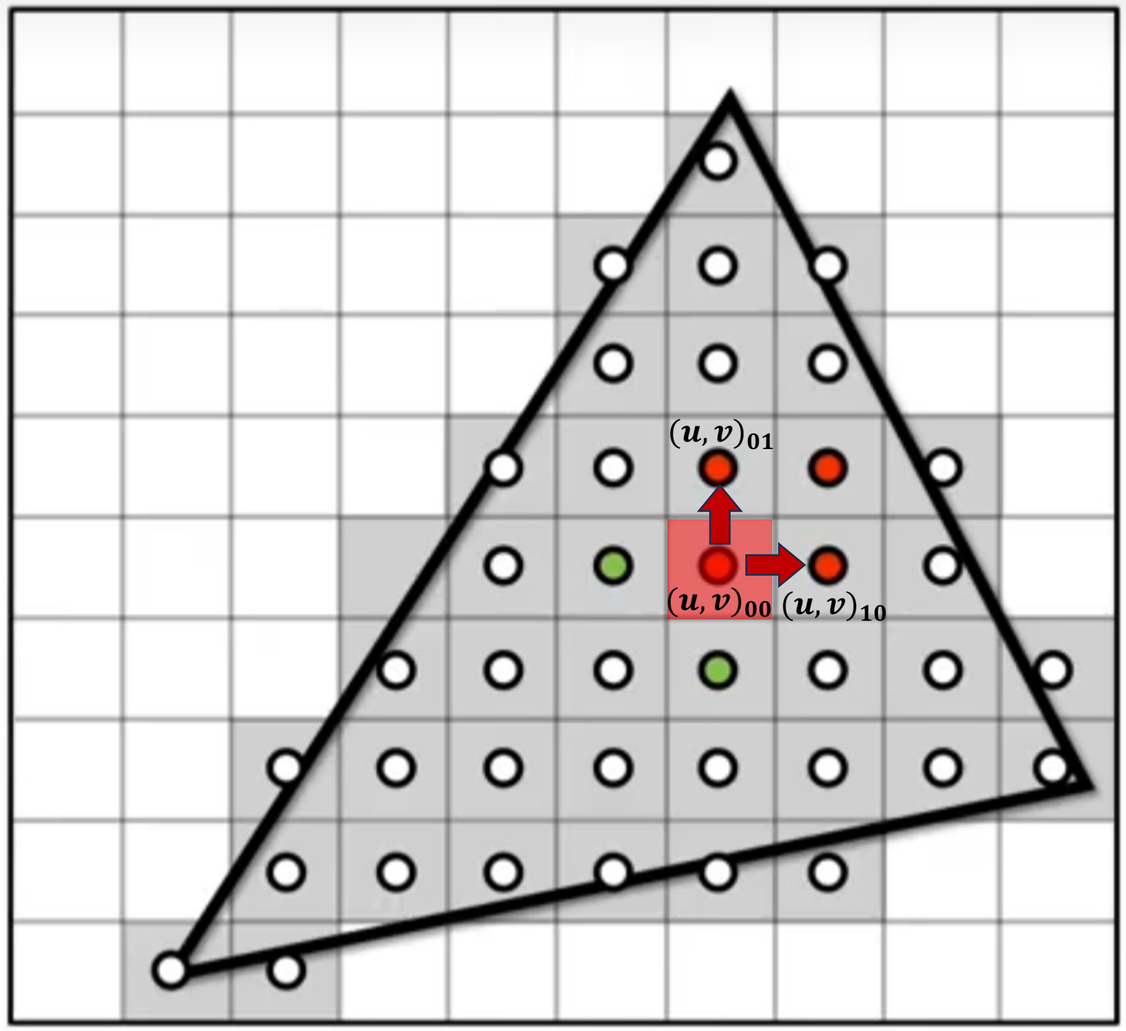
$$\frac{du}{dy} = u_{01} - u_{00} \qquad \frac{dv}{dy} = v_{01} - v_{00}$$

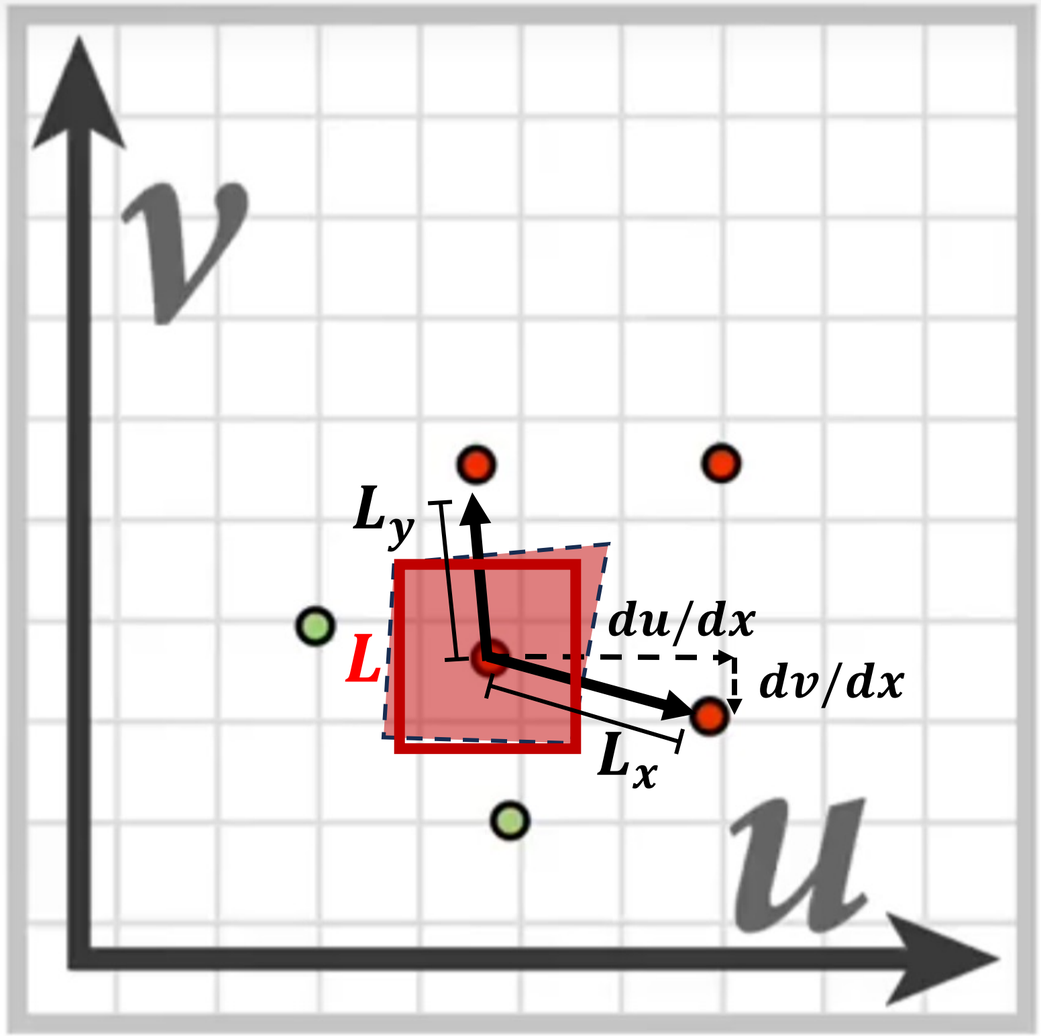
$$L_x^2 = (\frac{du}{dx})^2 + (\frac{dv}{dx})^2 \qquad L_y^2 = (\frac{du}{dy})^2 + (\frac{dv}{dy})^2$$

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

Computing Mip Map Level

Compute differences between texture coordinate values at neighboring samples





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

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

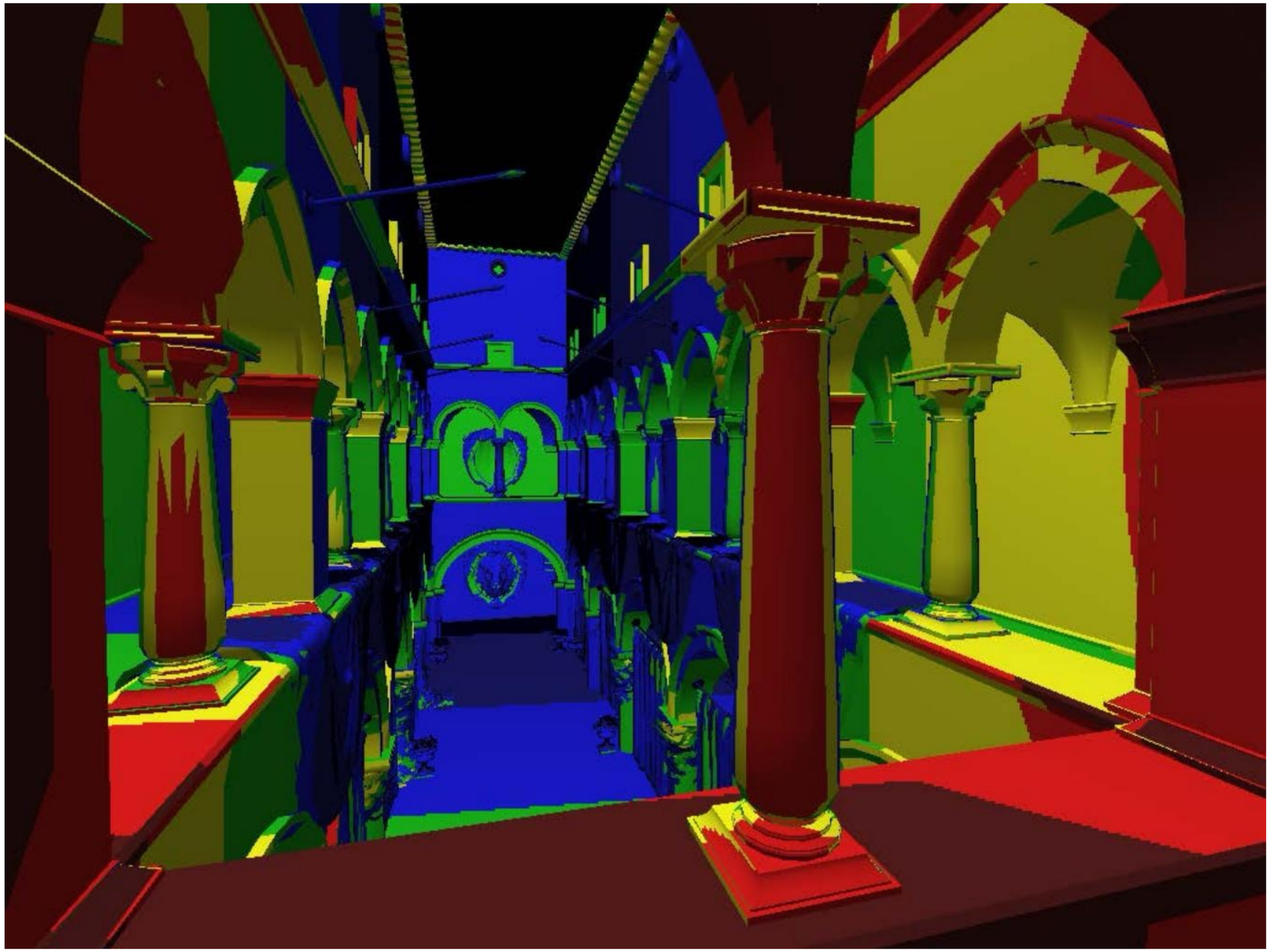
$$L_x^2 = (\frac{du}{dx})^2 + (\frac{dv}{dx})^2 \qquad L_y^2 = (\frac{du}{dy})^2 + (\frac{dv}{dy})^2$$

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

Mip-map level $d = log_2 L$

Visualization of mip-map level

(d clamped to nearest level)



Sponza (bilinear resampling at level 0)

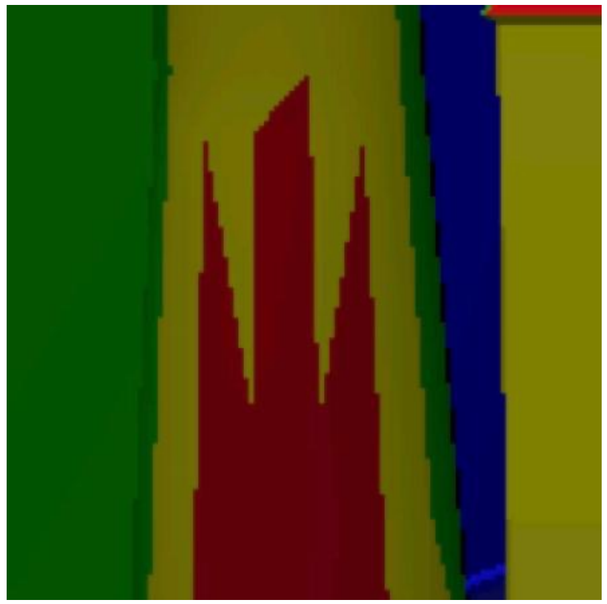
Sponza (bilinear resampling at level 2)

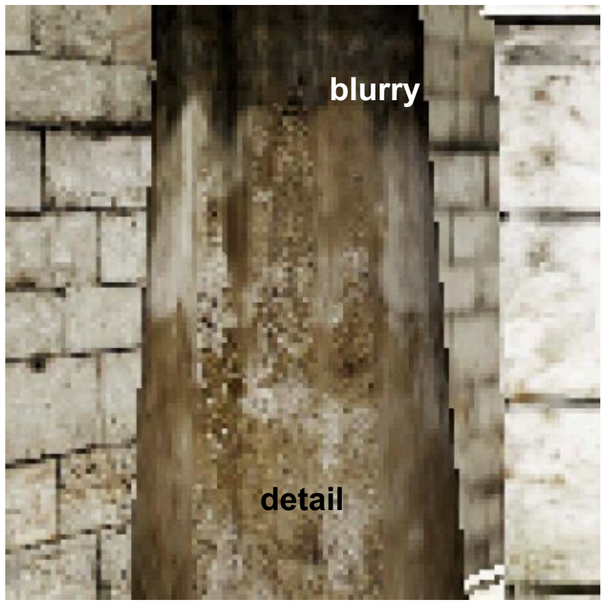
Sponza (bilinear resampling at level 4)

Sponza (MIP mapped)

Problem with basic MIP mapping

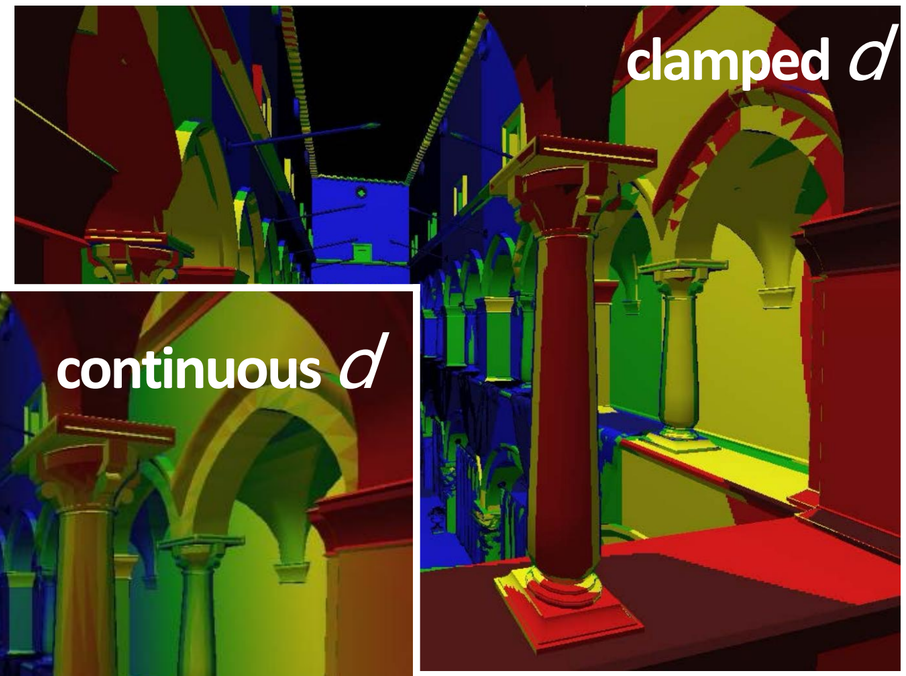
- If we just use the nearest level, can get artifacts where level "jumps"—appearance sharply transitions from detailed to blurry texture





- IDEA: rather than clamping the MIP map level to the closest integer, use the original (continuous) MIP map level d

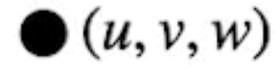
- PROBLEM: we only computed a fixed number of MIP map levels. How do we interpolate between levels?



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

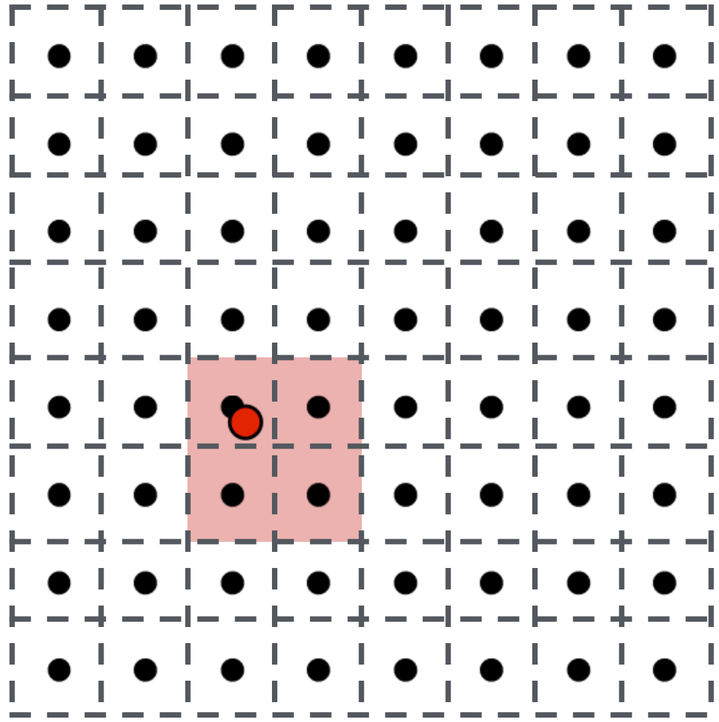
$$g_{00} = (1-u)f_{000} + uf_{100}$$
$$g_{10} = (1-u)f_{010} + uf_{110}$$
$$g_{01} = (1-u)f_{001} + uf_{101}$$
$$g_{11} = (1-u)f_{011} + uf_{111}$$

$$h_0 = (1-v)g_{00} + vg_{10}$$
$$h_1 = (1-v)g_{01} + vg_{11}$$

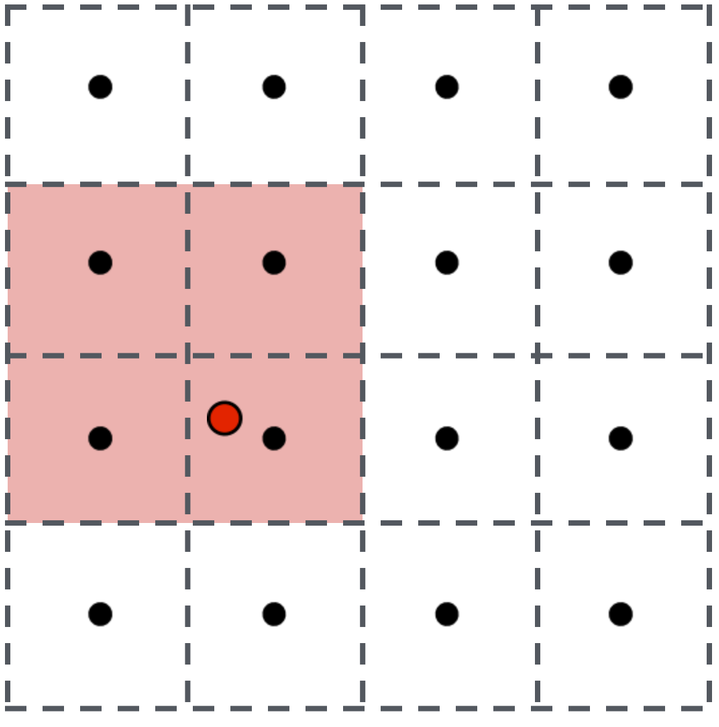
$$(1-w)h_0 + wh_1$$

MIP Map Lookup

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



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



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

MIP Map Lookup

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

Starts getting expensive! (➟specialized hardware)

Bilinear interpolation:

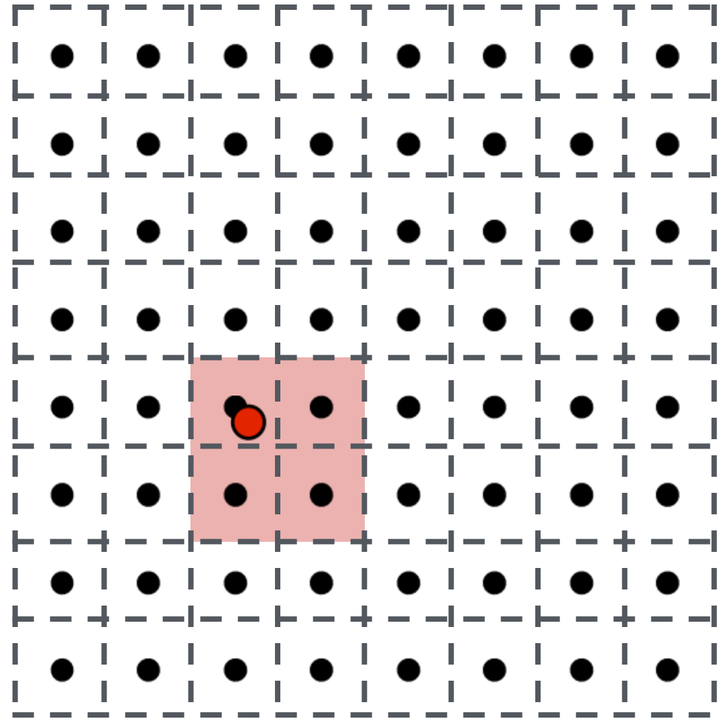
four texel reads

3 linear interpolations (3 mul + 6 add)

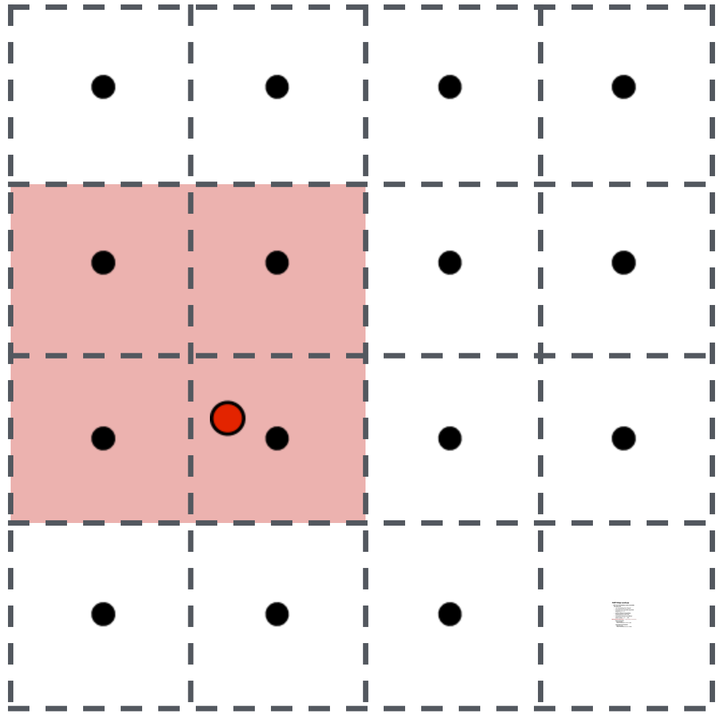
Trilinear/MIP map interpolation:

eight texel reads

7 linear interpolations (7 mul + 14 add)



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



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

Texture Sampling Pipeline

1. Compute u and v from screen sample (x, y) via barycentric interpolation
2. Approximate $\frac{\mathrm{du}}{dx}, \frac{\mathrm{du}}{dy}, \frac{\mathrm{d}v}{dx}, \frac{\mathrm{d}v}{dy}$ 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 $(U, V) = [W, H]$ 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)

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

Texture Mapping—Summary

- 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 average 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
- In general, no perfect solution to aliasing! Try to balance quality & efficiency