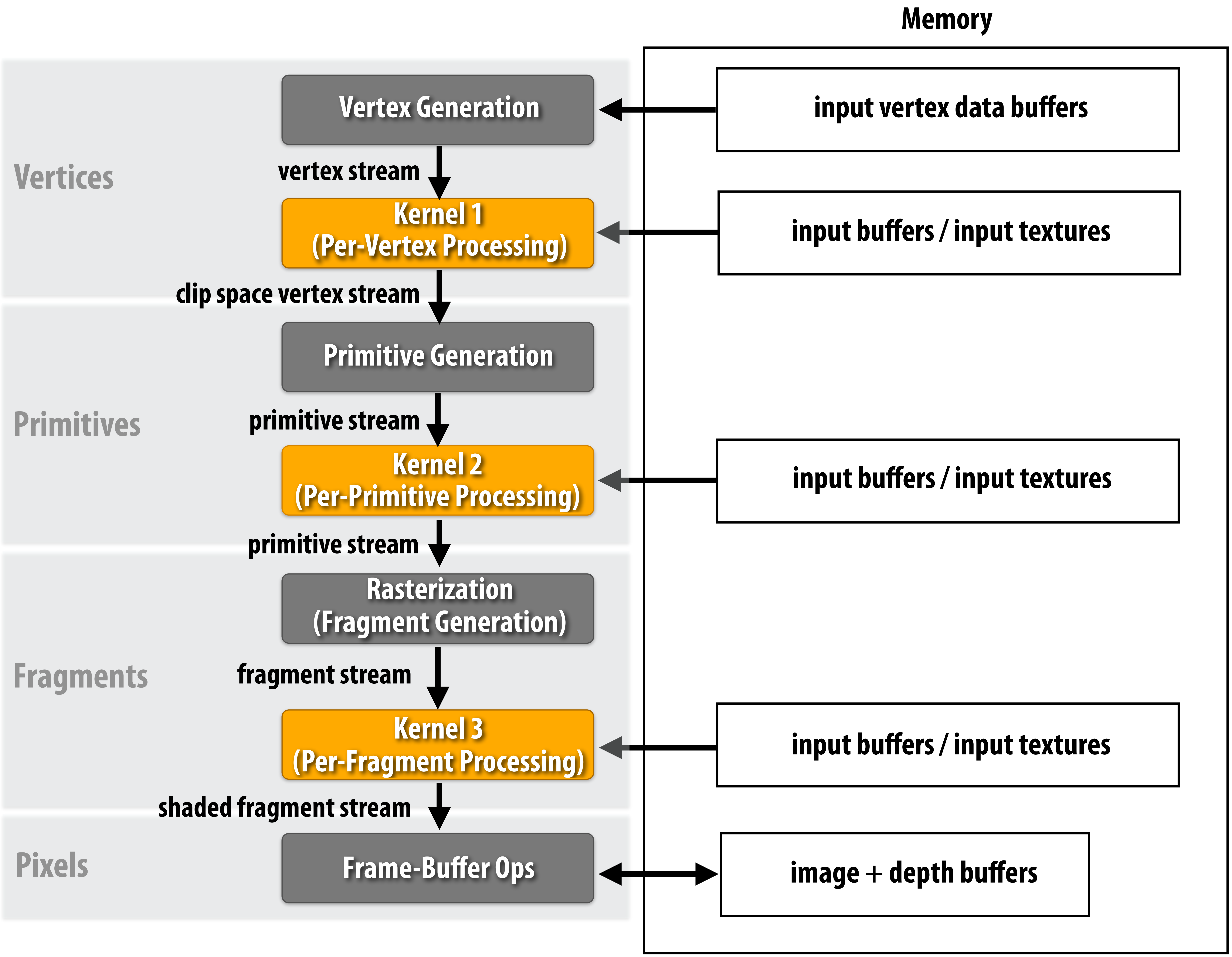
Lecture 19:

Texturing mapping algorithms and hardware

Visual Computing Systems
CMU 15-769, Fall 2016

Graphics pipeline architecture



Today: texturing!

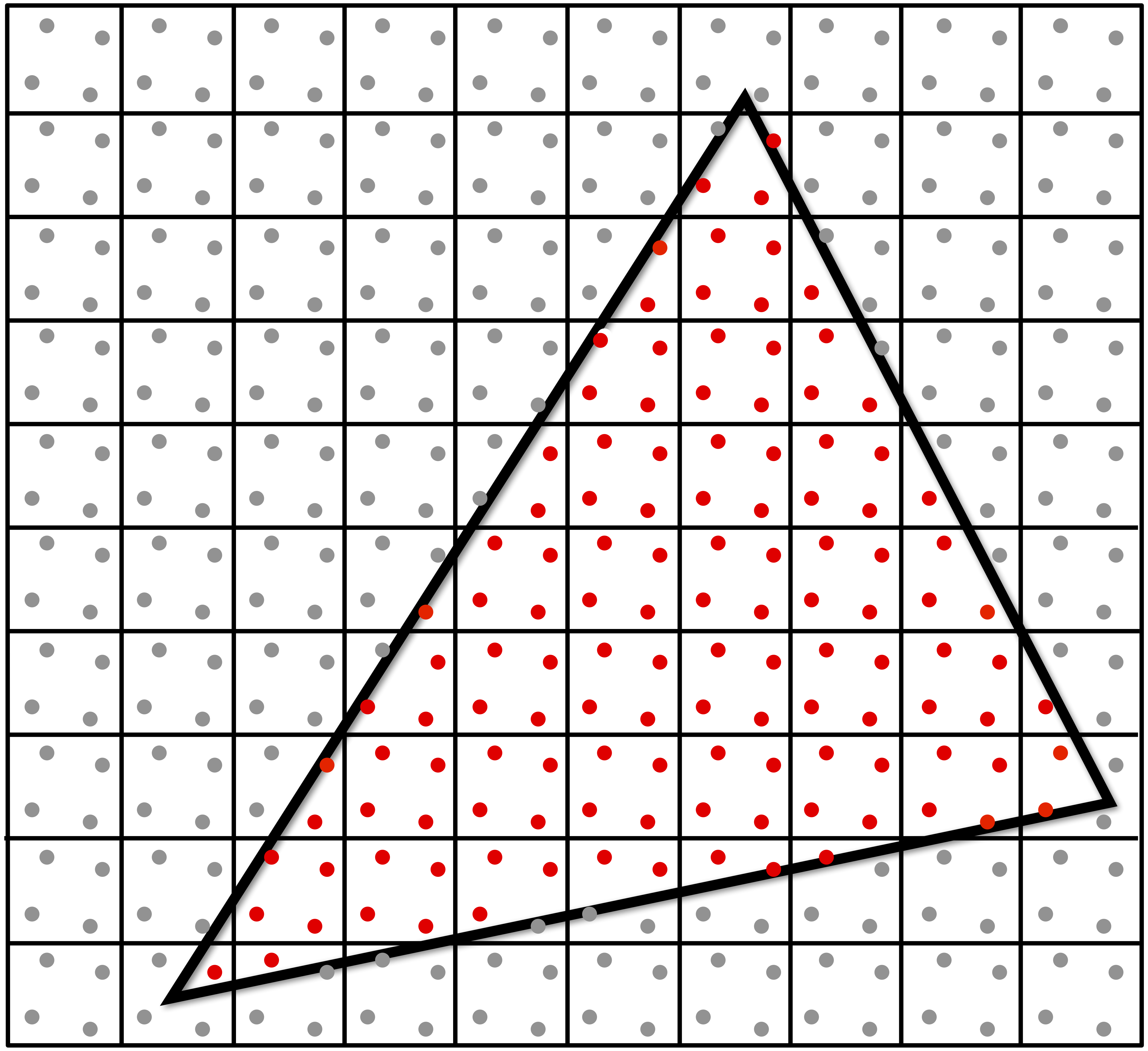
- Texture filtering math
  - At the very least… a texture access is not just a 2D array lookup ;-)
  - Implemented in fixed-function hardware in modern GPUs

- Memory-system implications of texture mapping operations
  - Texture caching
  - Memory layout of texture data
  - Texture compression (decompression support in hardware)
  - Prefetching and multi-threading

Previous lecture (visibility)

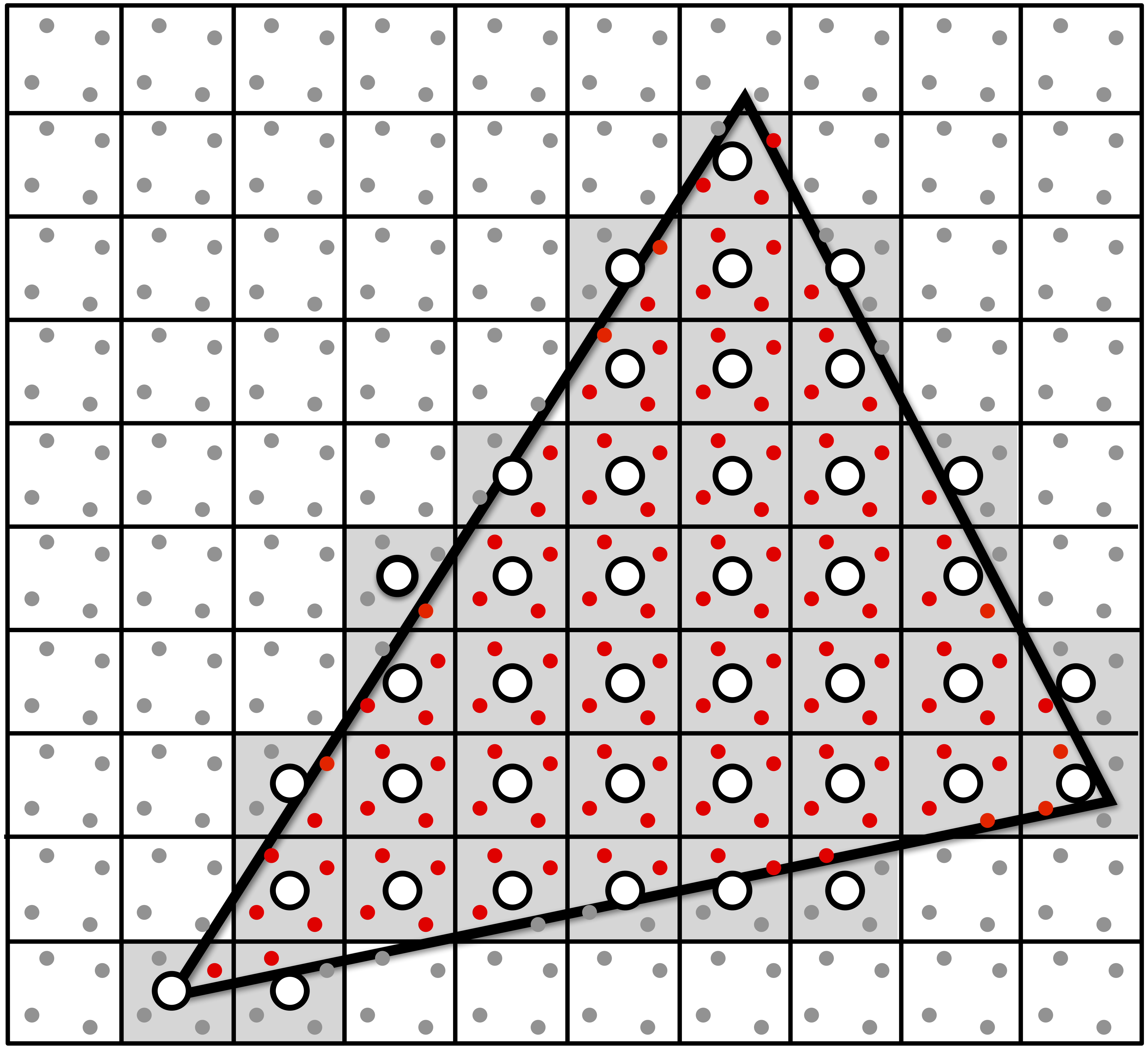
Rasterizer samples triangle-screen coverage (four samples per pixel shown here)

Z-buffer algorithm used to determine occlusion at these sample points



Generating fragments via “multi-sampling”

- Rasterizer samples coverage at N sample points per pixel (small dots in figure)
- If <u>any</u> visibility sample in a pixel is covered, GPU generates fragment for pixel **
- Surface attributes for fragment shading are typically sampled at pixel centers (big dots in figure)



** As we’ll discuss later in this lecture: a GPU actually generates a 2x2 block of fragments if any visibility sample in the 2x2 block is covered

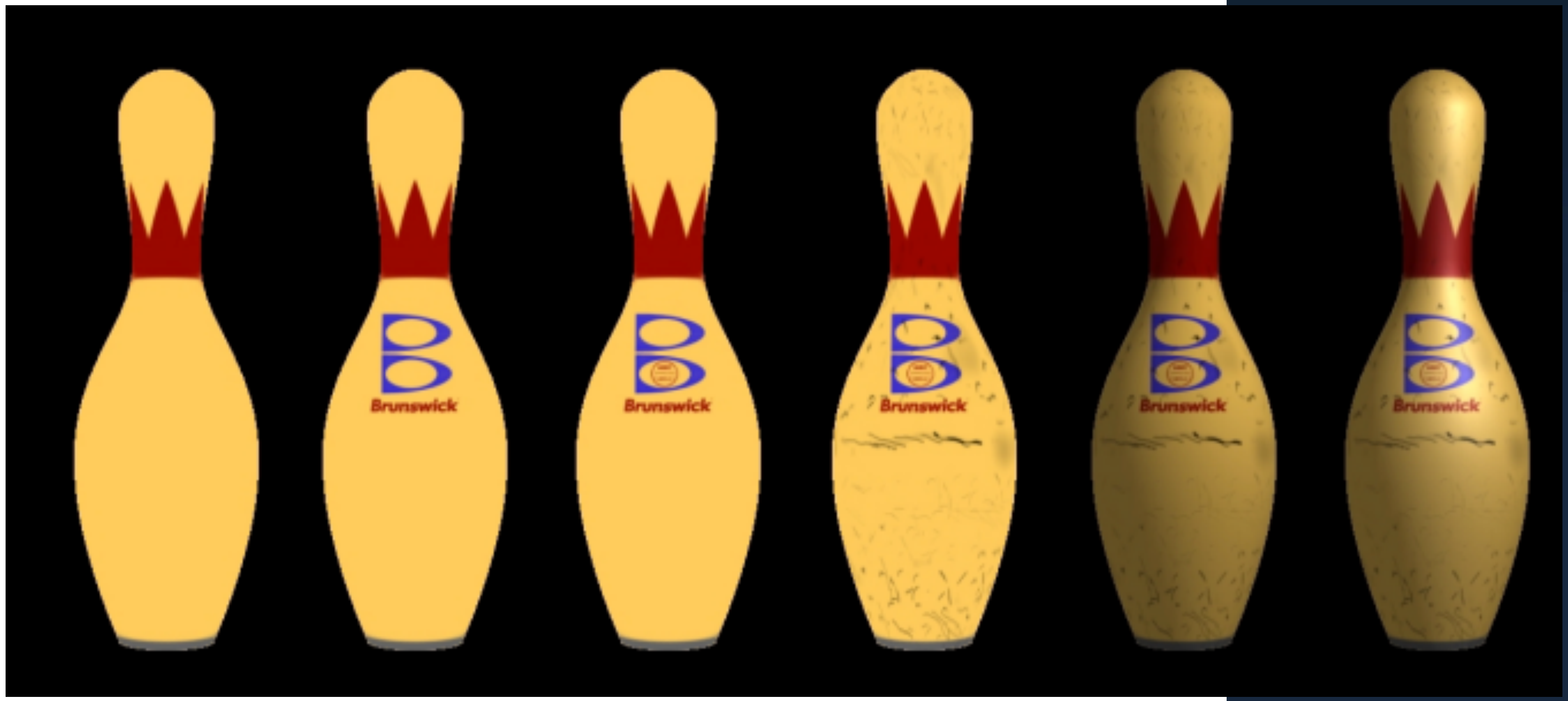
Many uses of texture mapping

Define variation in surface reflectance

Pattern on ball

Wood grain on floor

Describe surface material properties

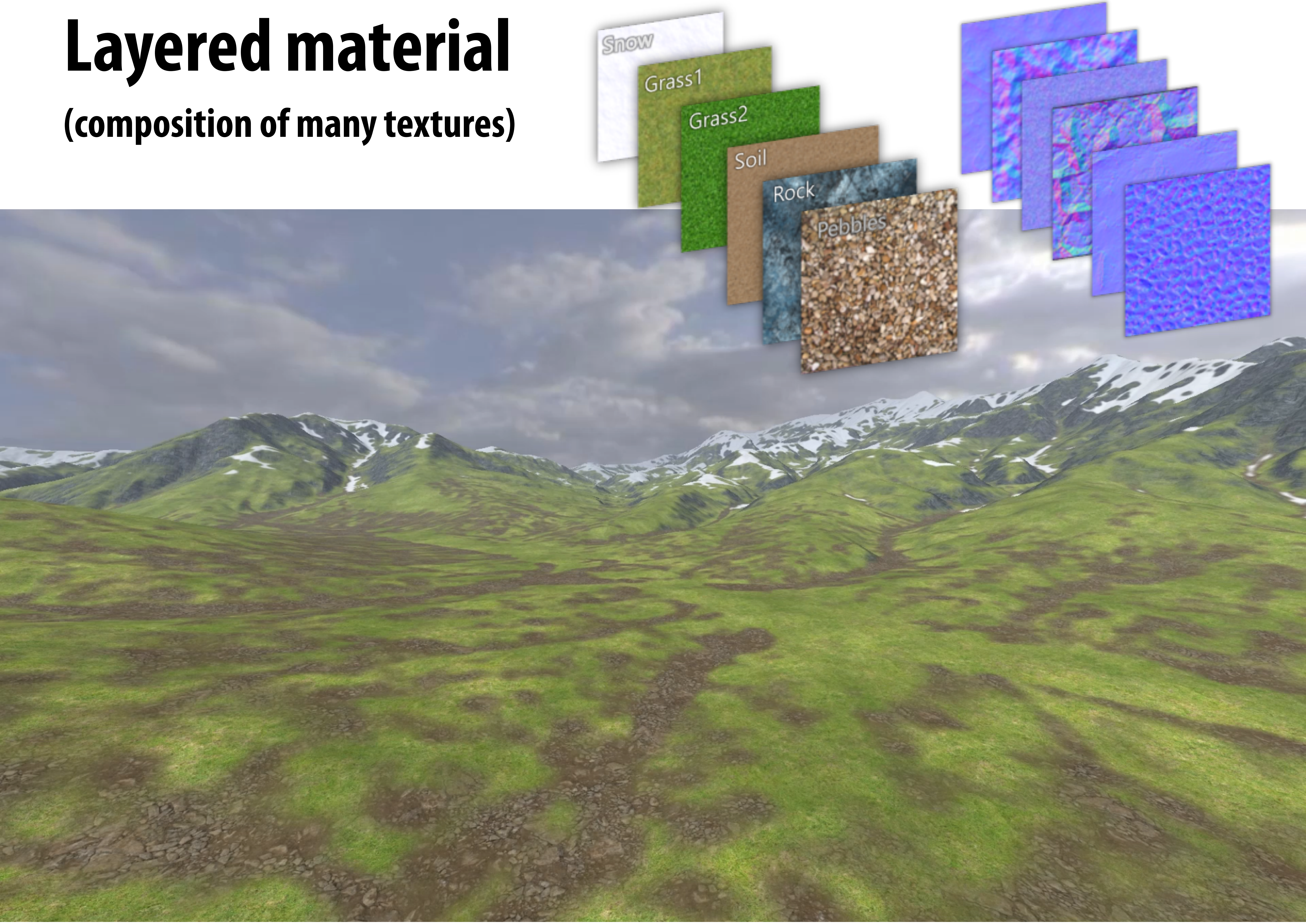


Multiple layers of texture maps for color, logos, scratches, etc.

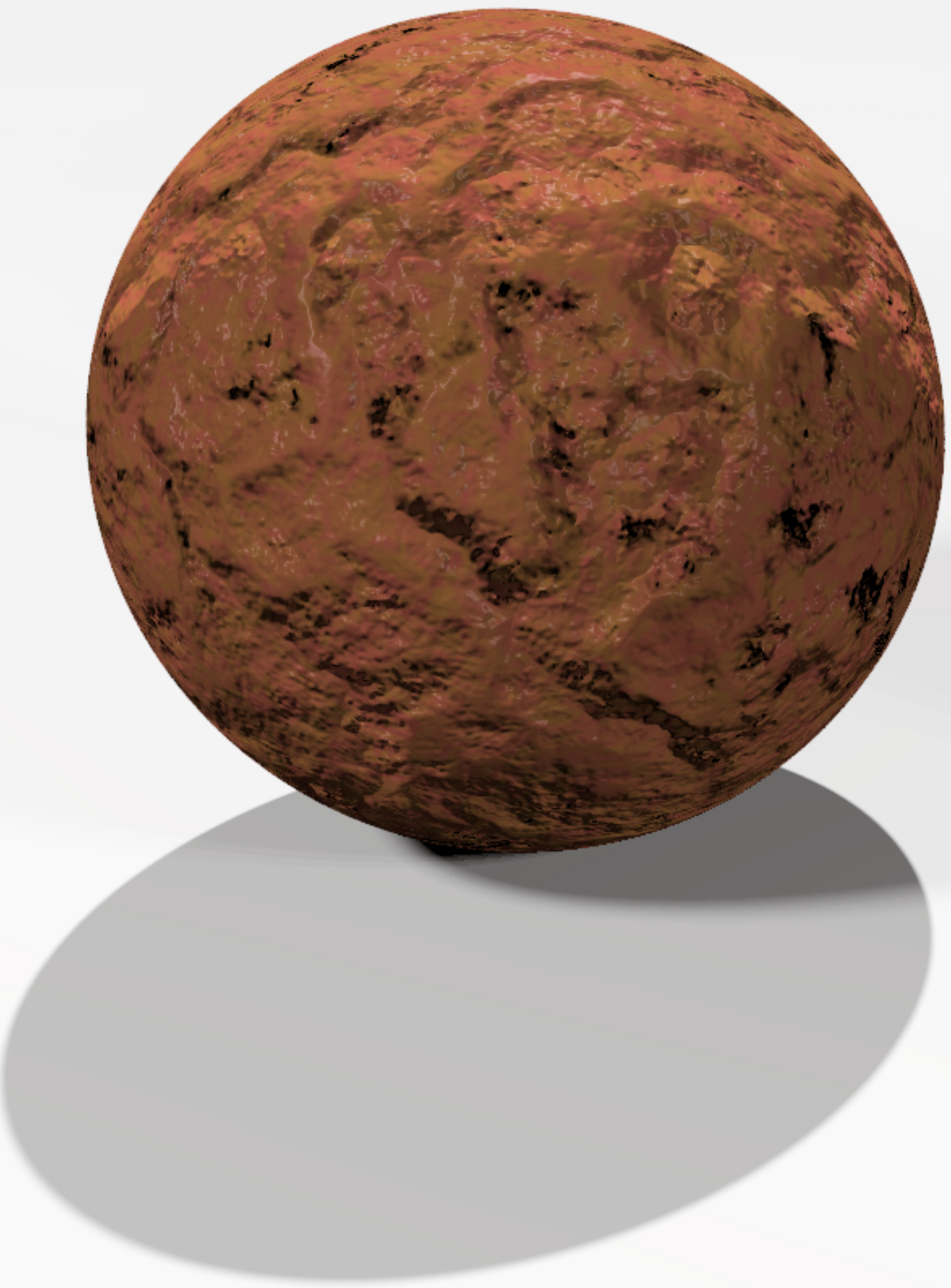
(C)2013 CRYTEK GMBH. ALL RIGHTS RESERVED. RYSE IS A REGISTERED TRADEMARK OF CRYTEK GMBH

Layered material

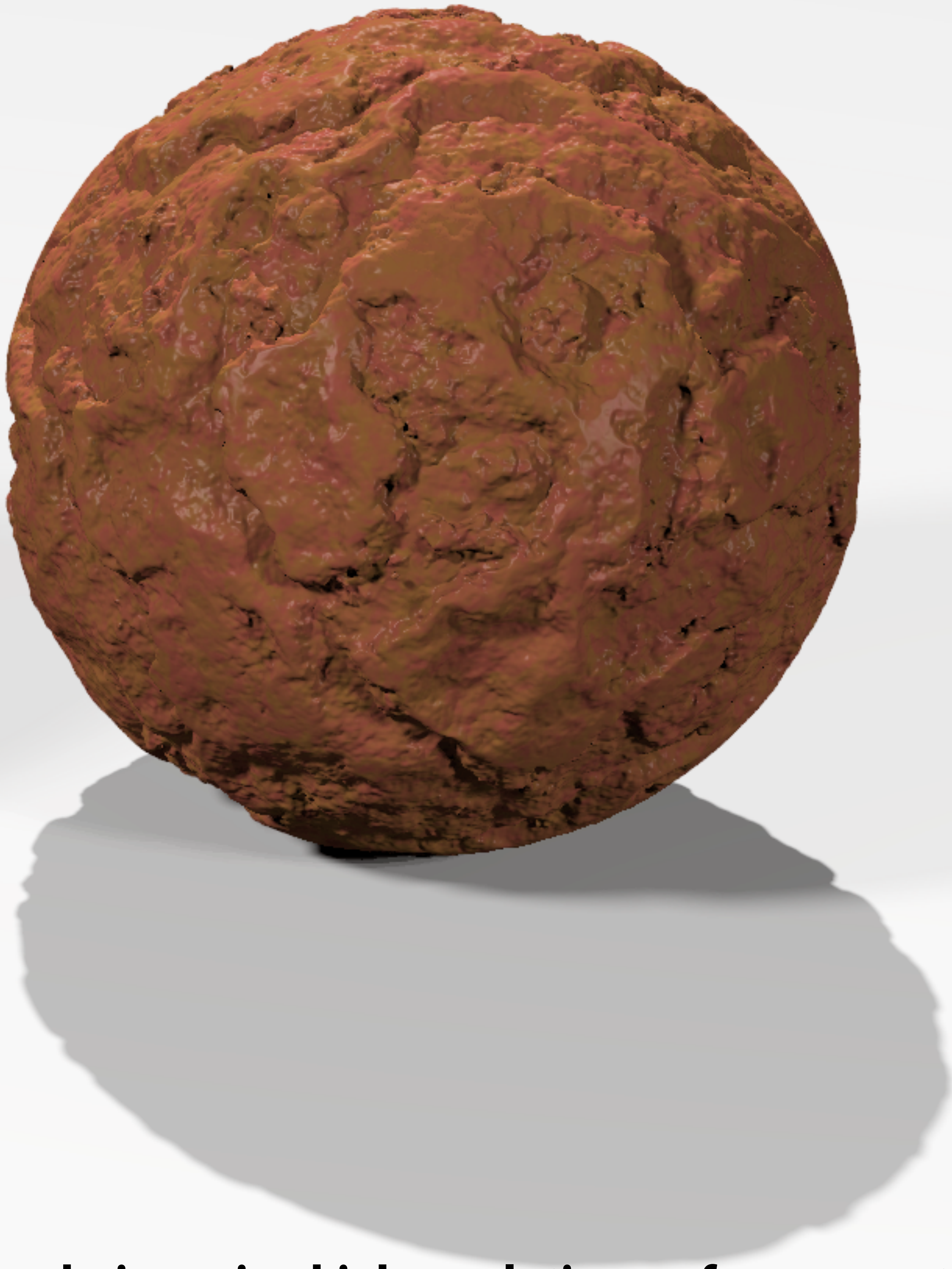
(composition of many textures)



Normal mapping



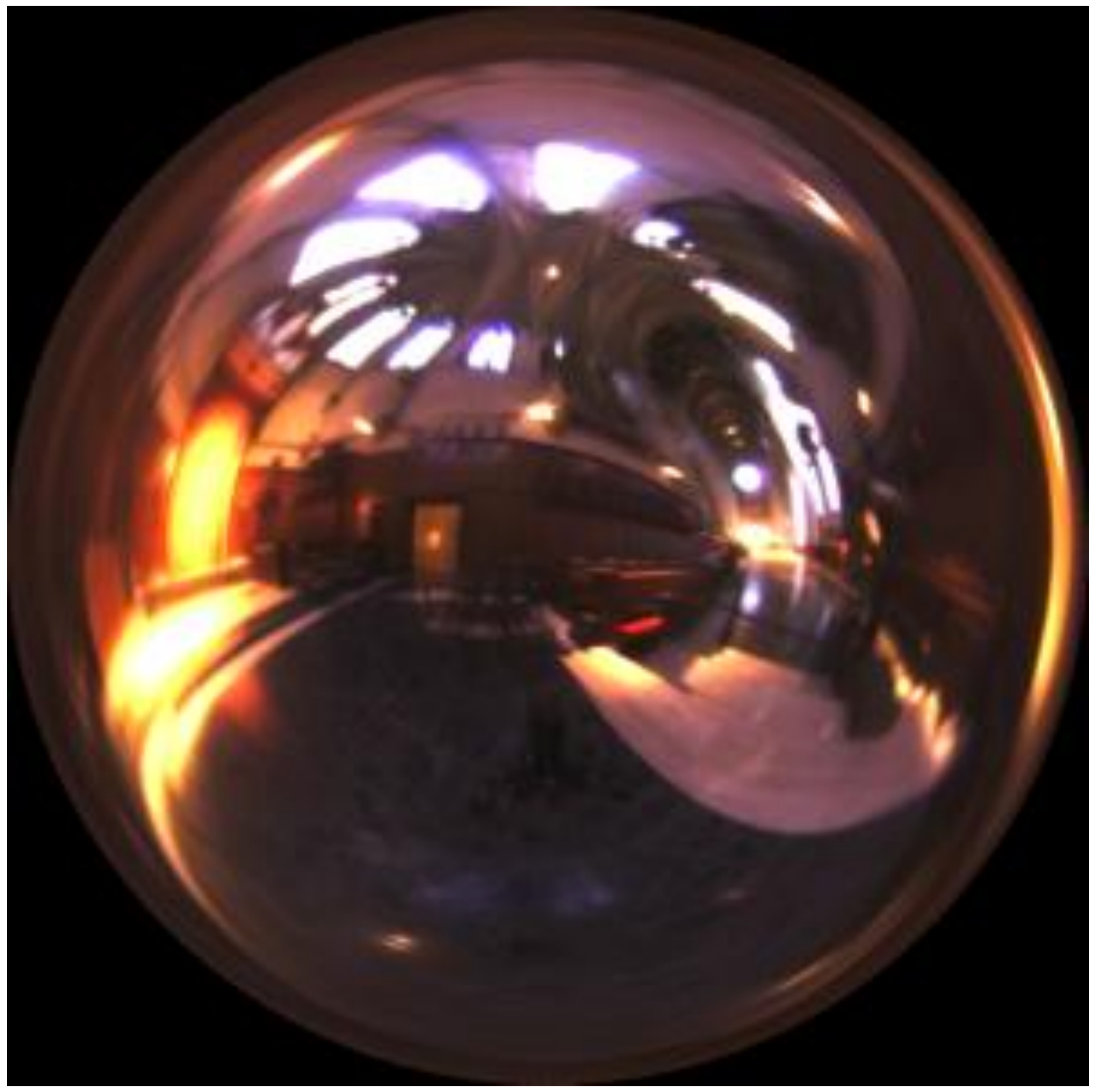
Use texture value to perturb surface normal to give appearance of a bumpy surface
Observe: smooth silhouette and smooth shadow boundary indicates surface geometry is not bumpy



Rendering using high-resolution surface geometry
(note bumpy silhouette and shadow boundary)

Textures used to represent precomputed lighting and shadows

Original model With ambient occlusion Extracted ambient occlusion map



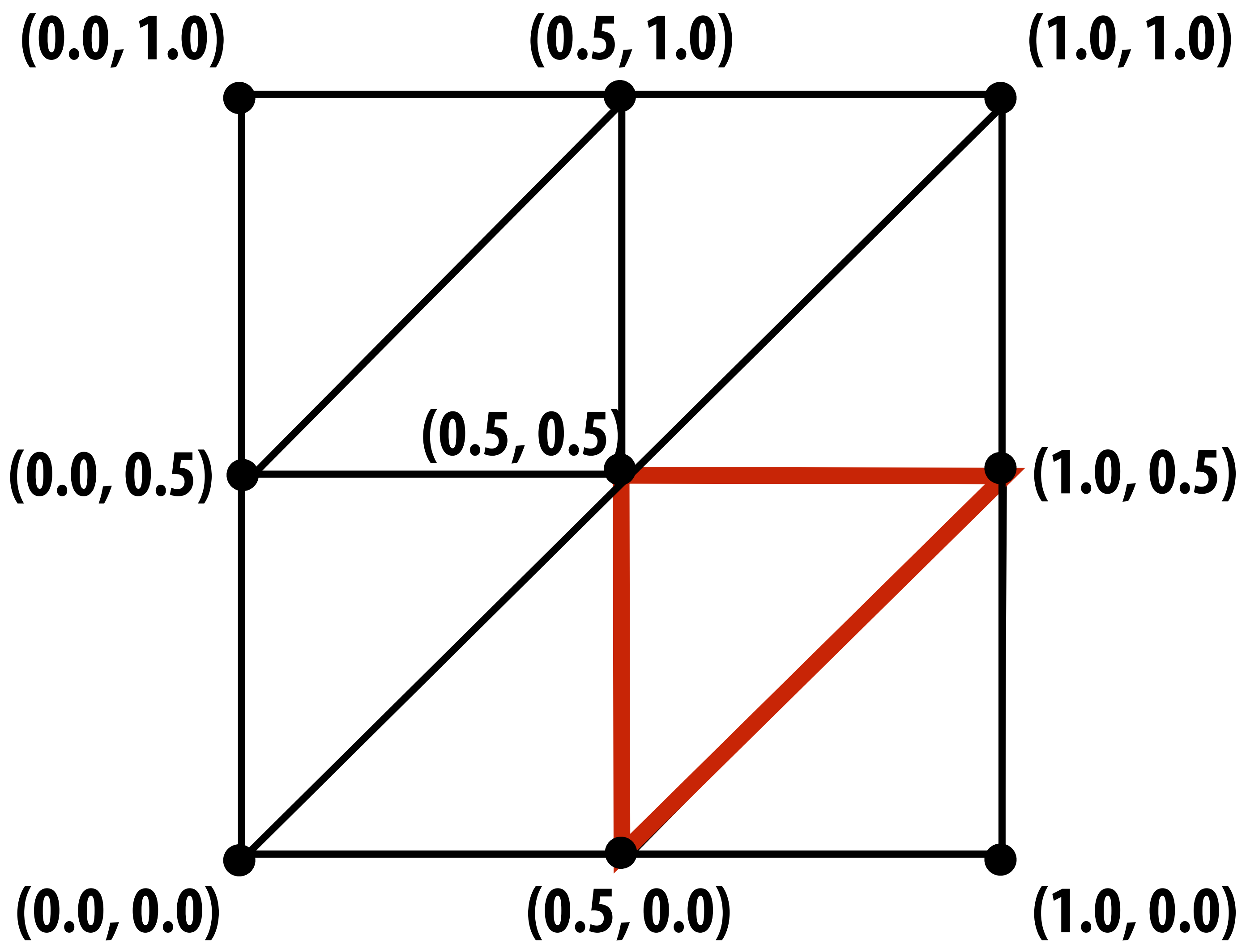
Grace Cathedral environment map

Environment map used in rendering

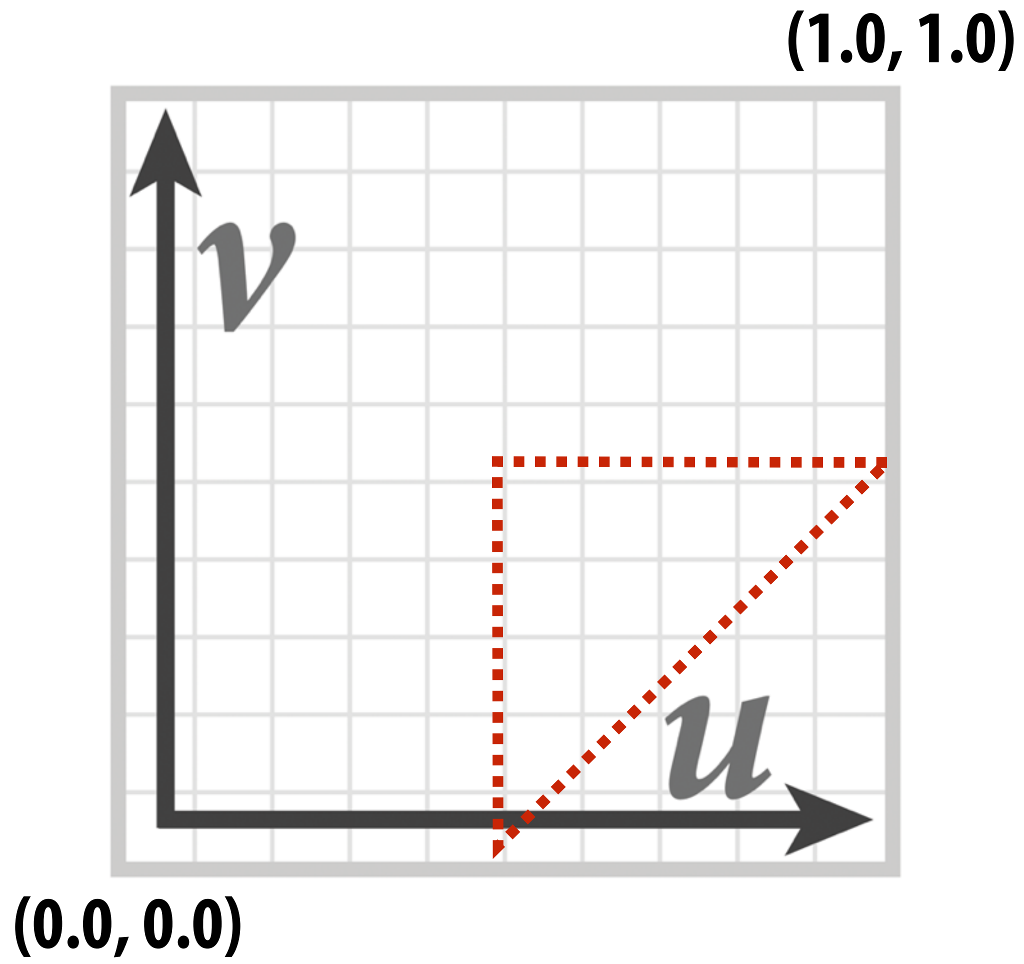
Texture mapping math

Texture coordinates

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



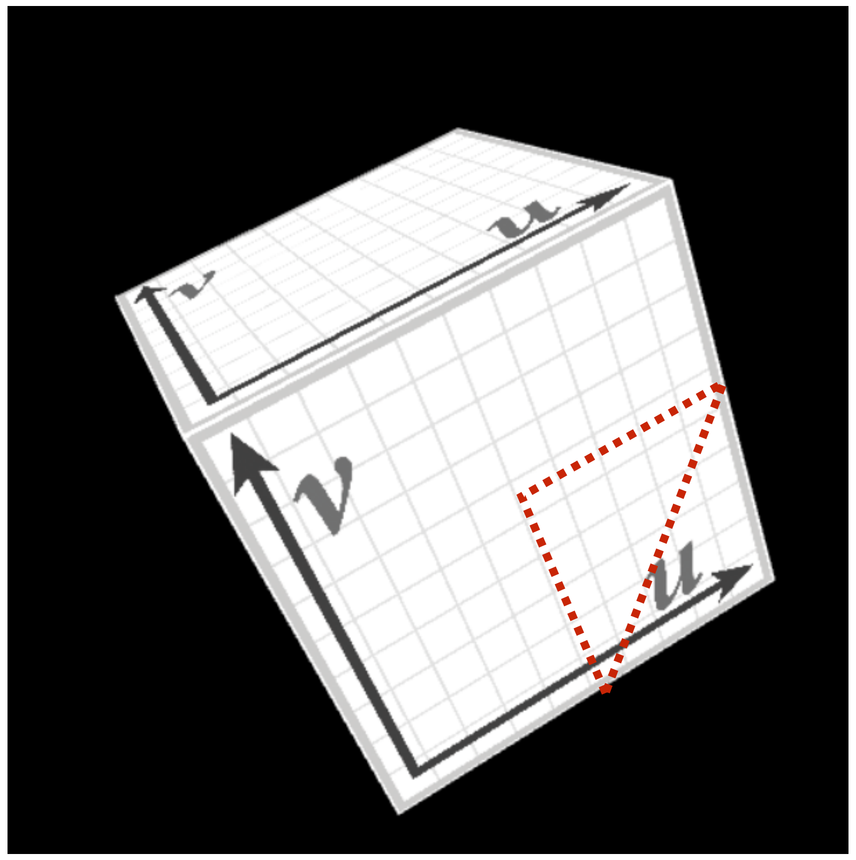
Eight triangles (one face of cube) with surface parameterization provided as per-vertex texture coordinates.



`myTex(u,v)` is a function defined on the $[0,1]^2$ domain:

myTex : $[0,1]^2 \longrightarrow$ float3
(represented by 2048x2048 image)

Location of highlighted triangle in texture space shown in red.



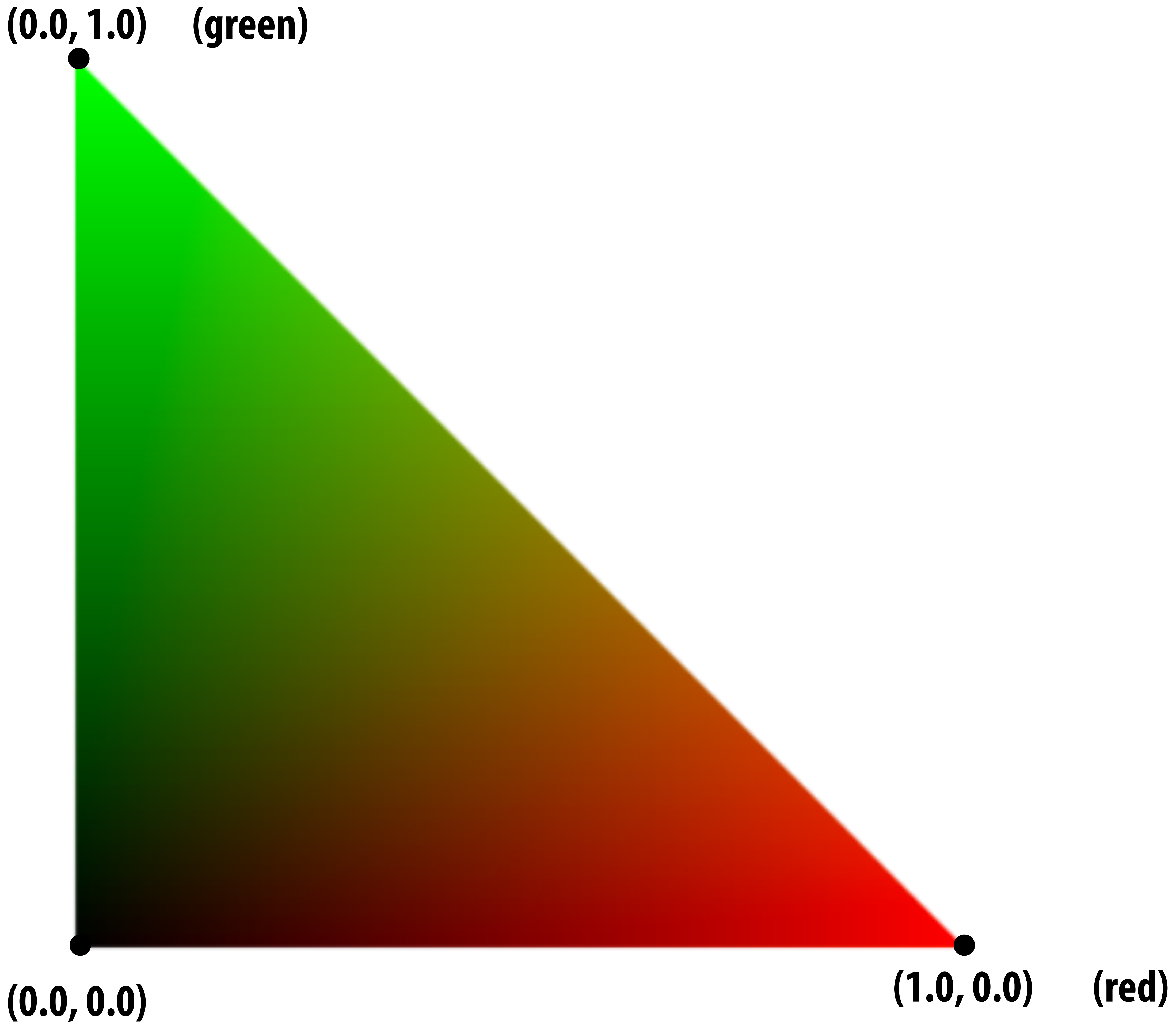
Final rendered result (entire cube shown).

Location of triangle after projection onto screen shown in red.

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

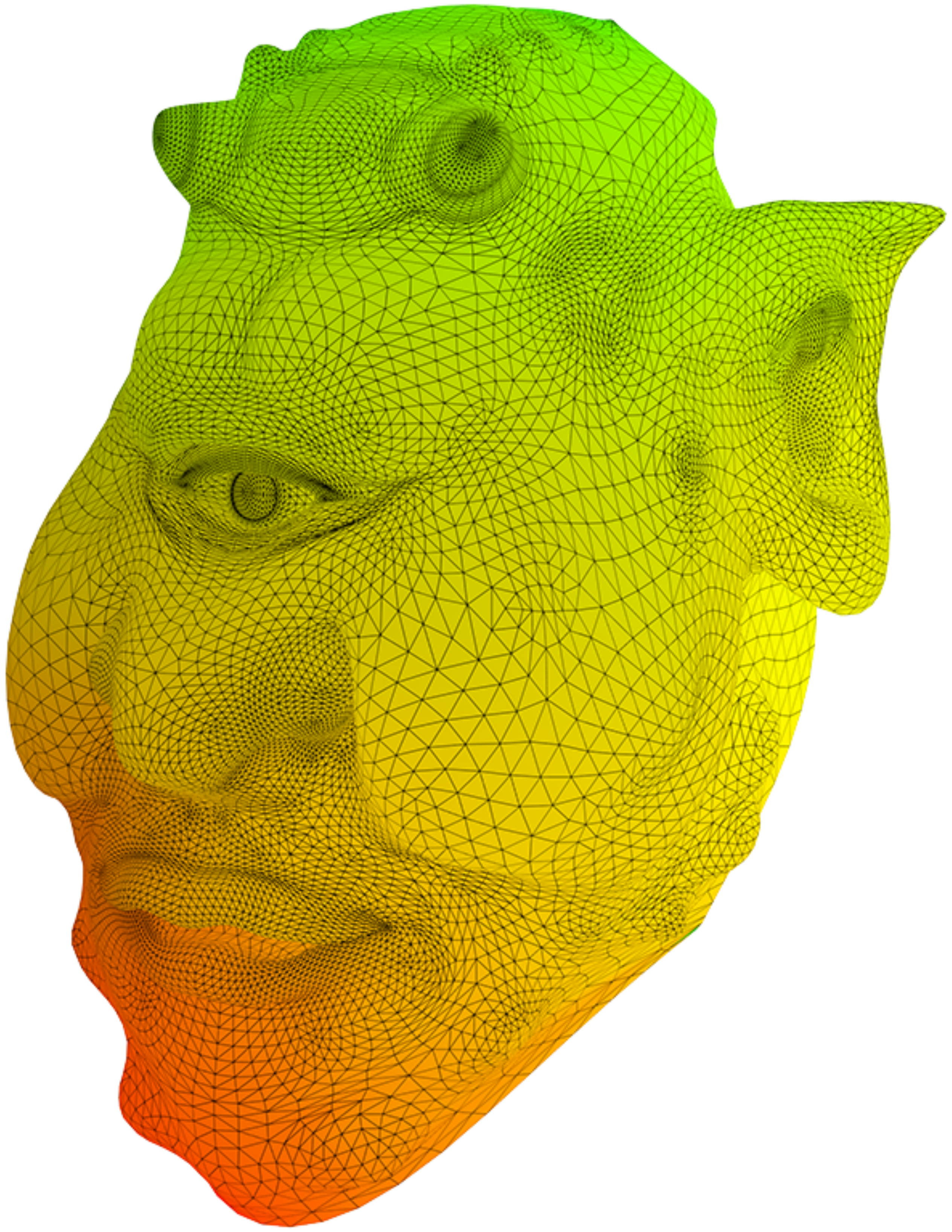
Visualization of texture coordinates

Texture coordinates linearly interpolated over triangle

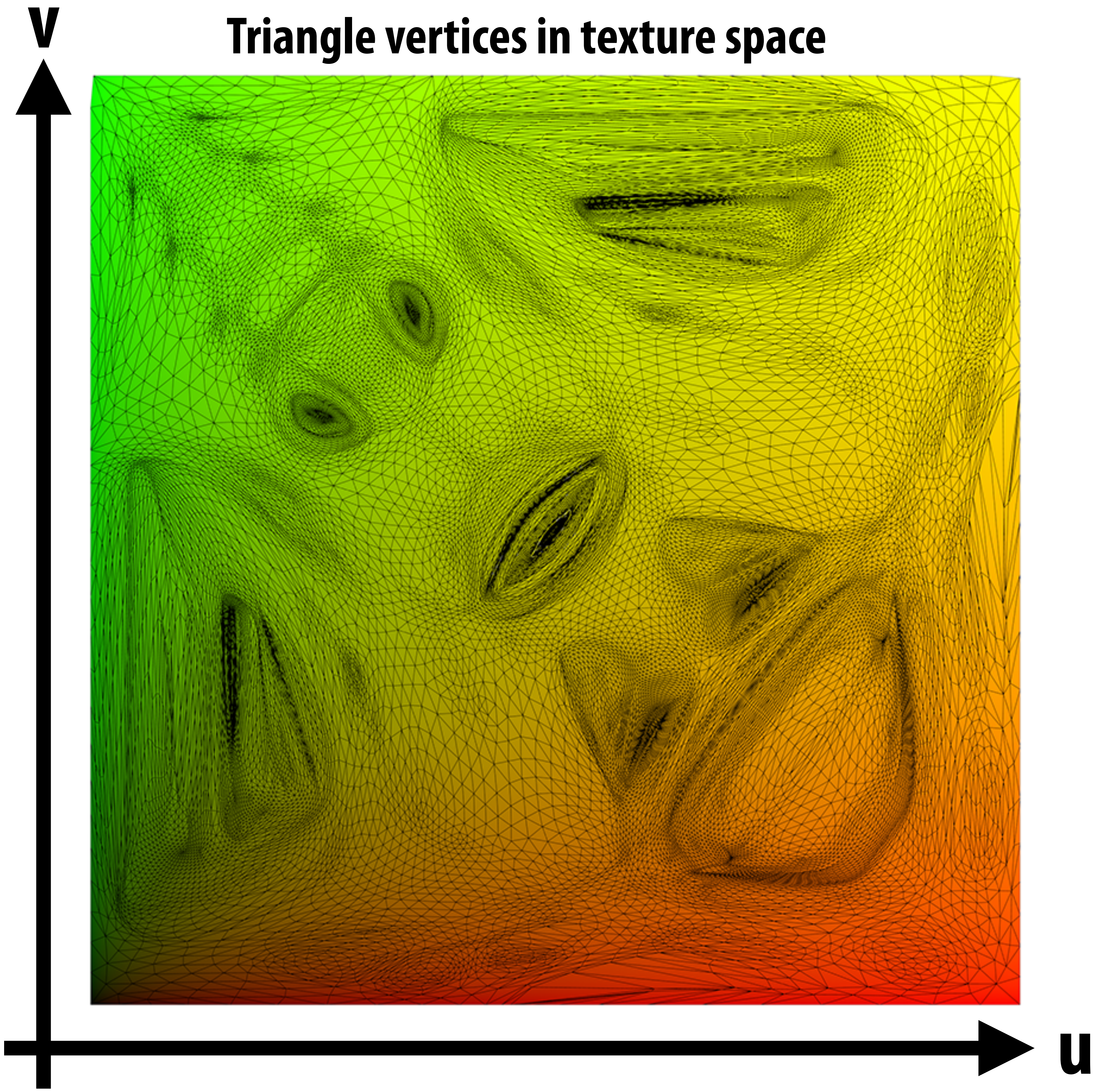


More complex mapping

Visualization of texture coordinates



Triangle vertices in texture space



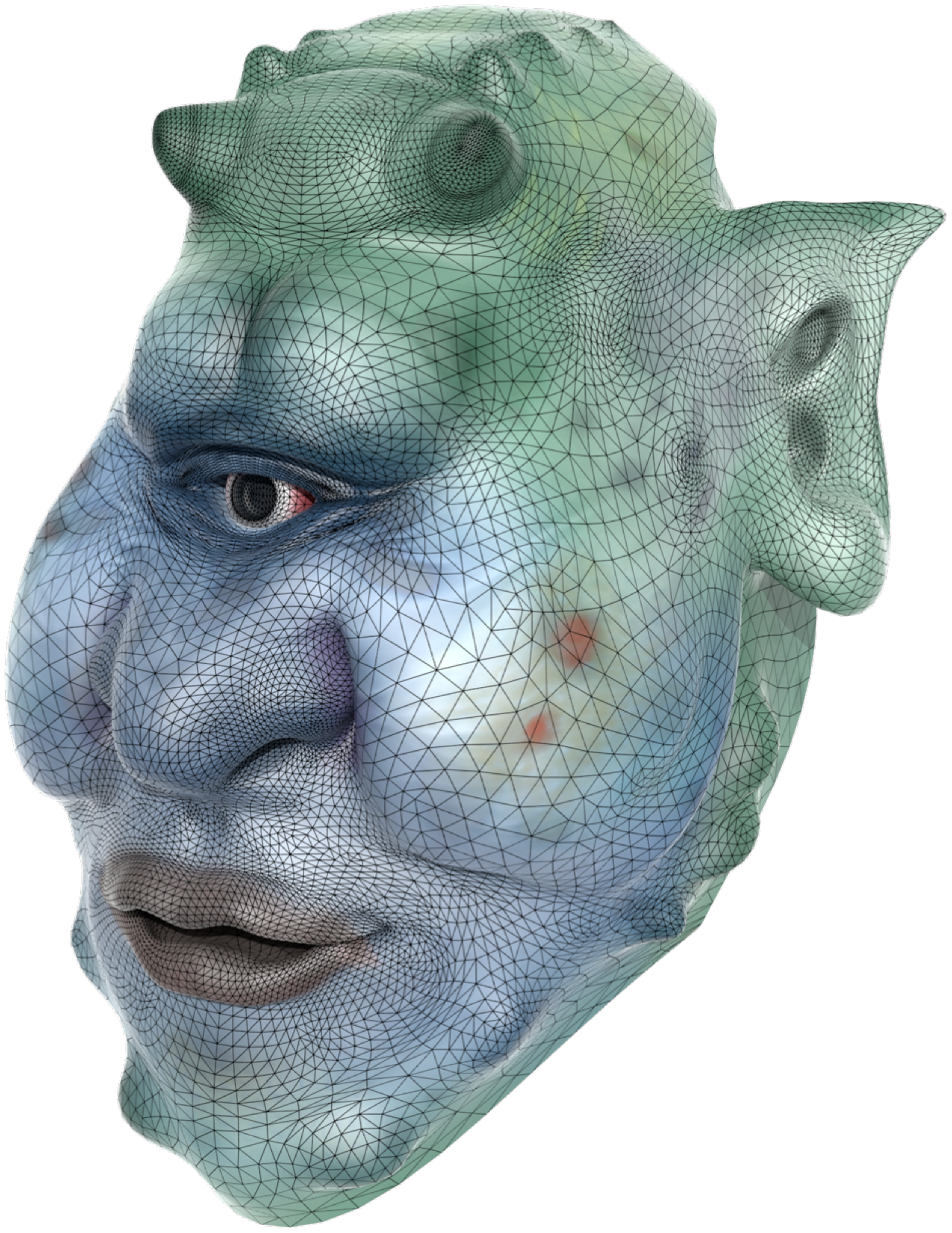
Each vertex has a coordinate (u,v) in texture space.

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

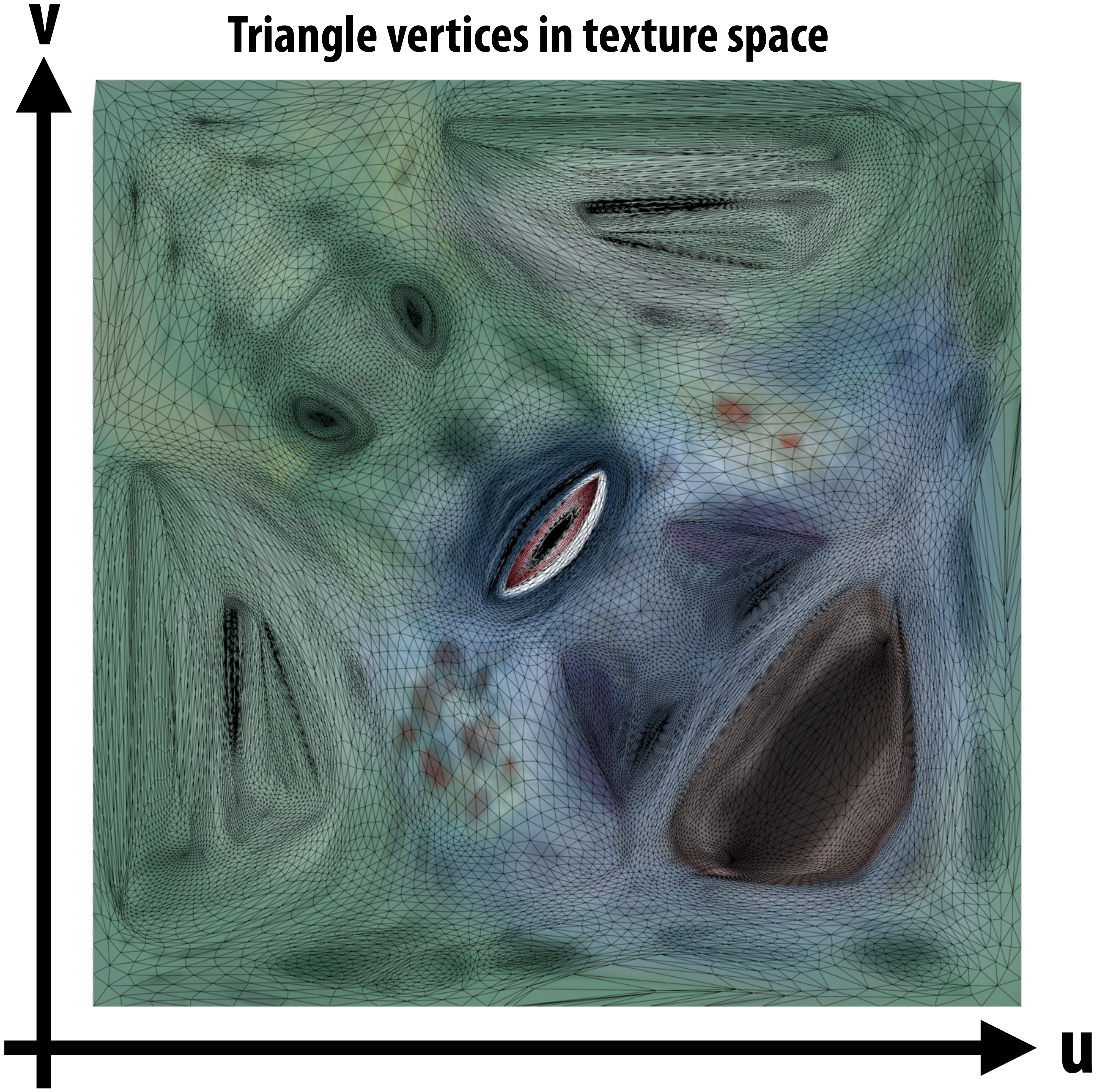
Simple texture mapping operation

```
for each fragment (x,y) in fragment stream:

   // interpolate per-vertex coordinates (eval attribute plane eqn)
   (u,v) = evaluate texcoord value at (x,y);

   float3 texture_color = texture.sample(u,v);
   color of surface at (x,y) = texture_color;
```

Texture mapping adds detail

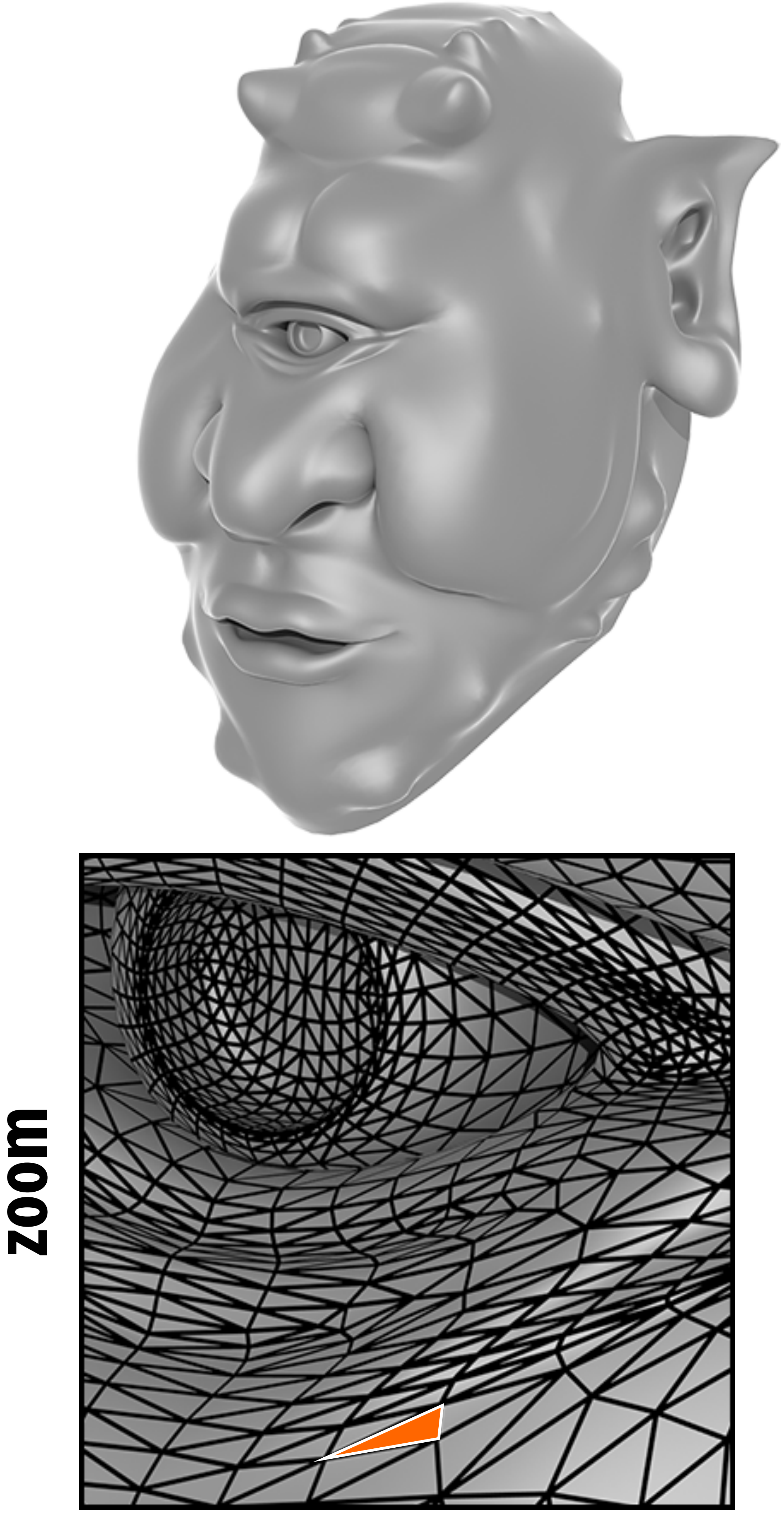


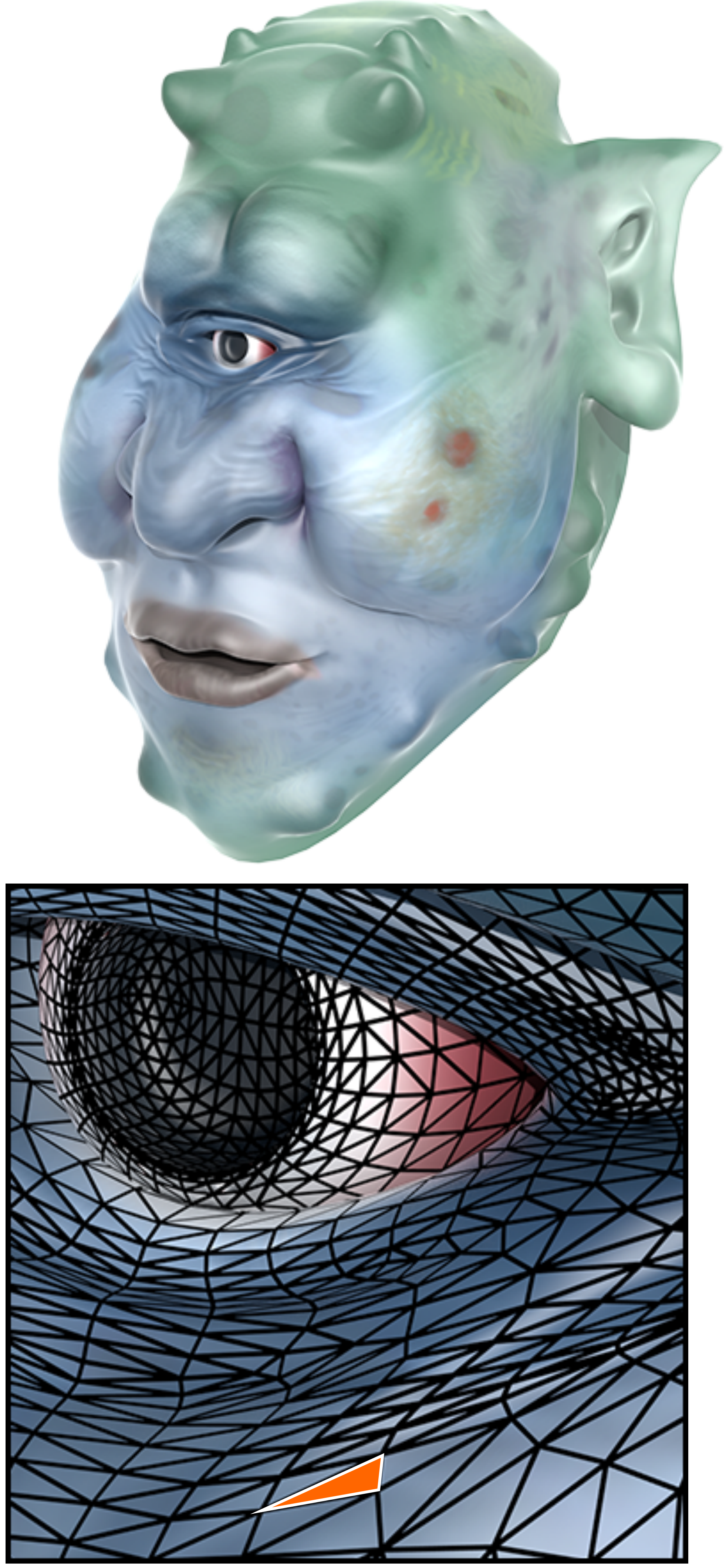
Rendered result

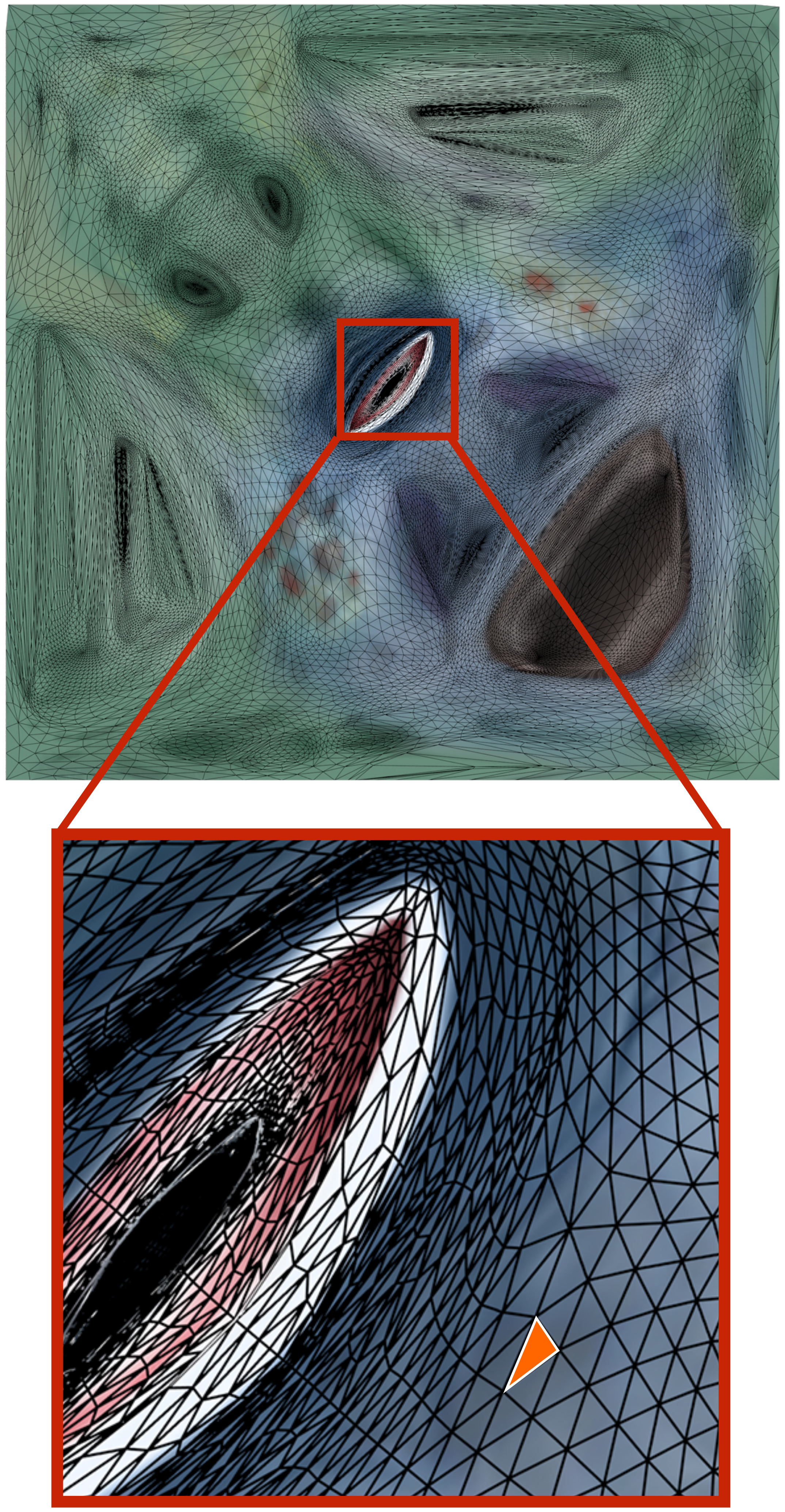


Triangle vertices in texture space

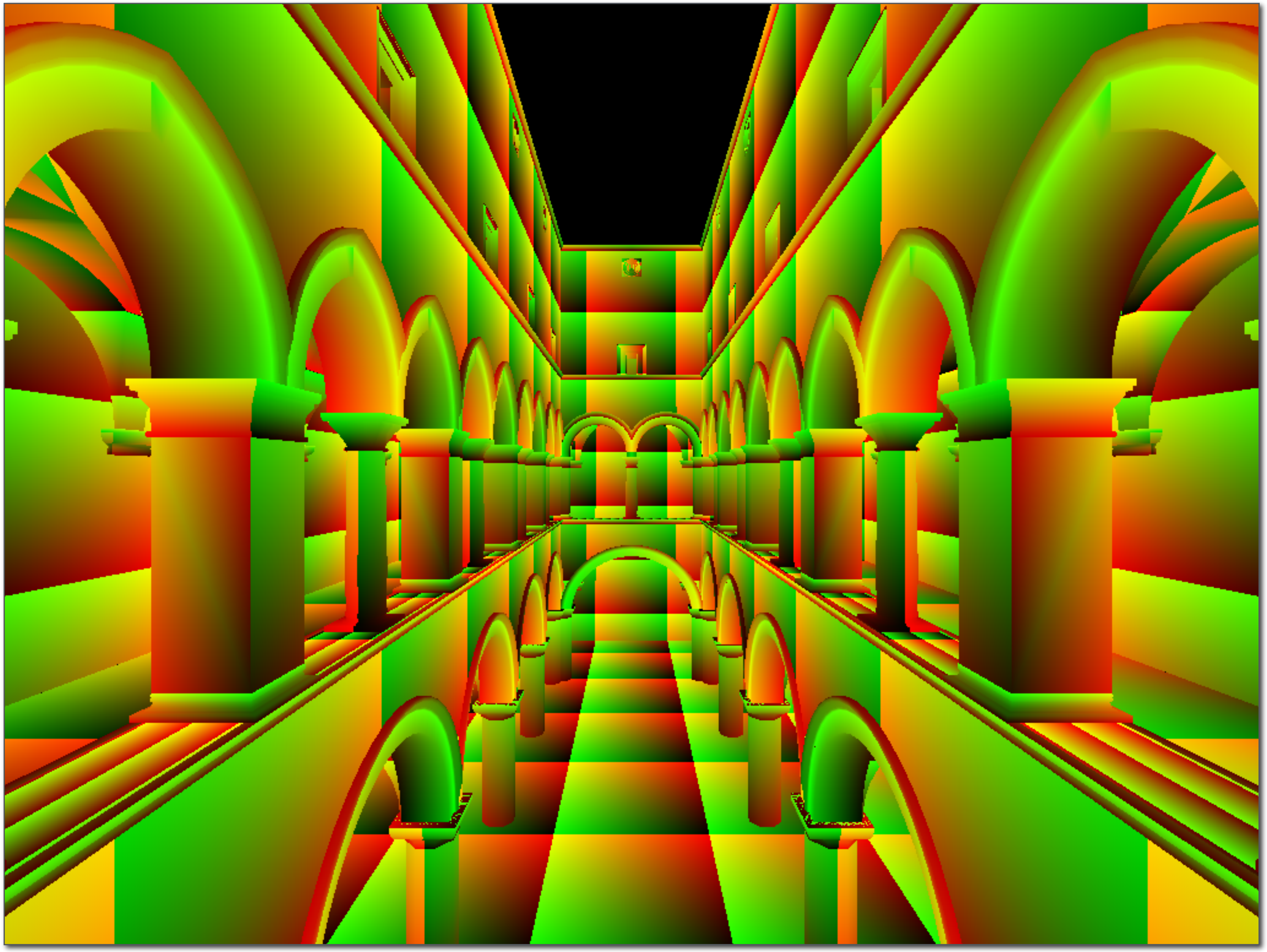
Texture mapping adds detail


rendering without texture


rendering with texture

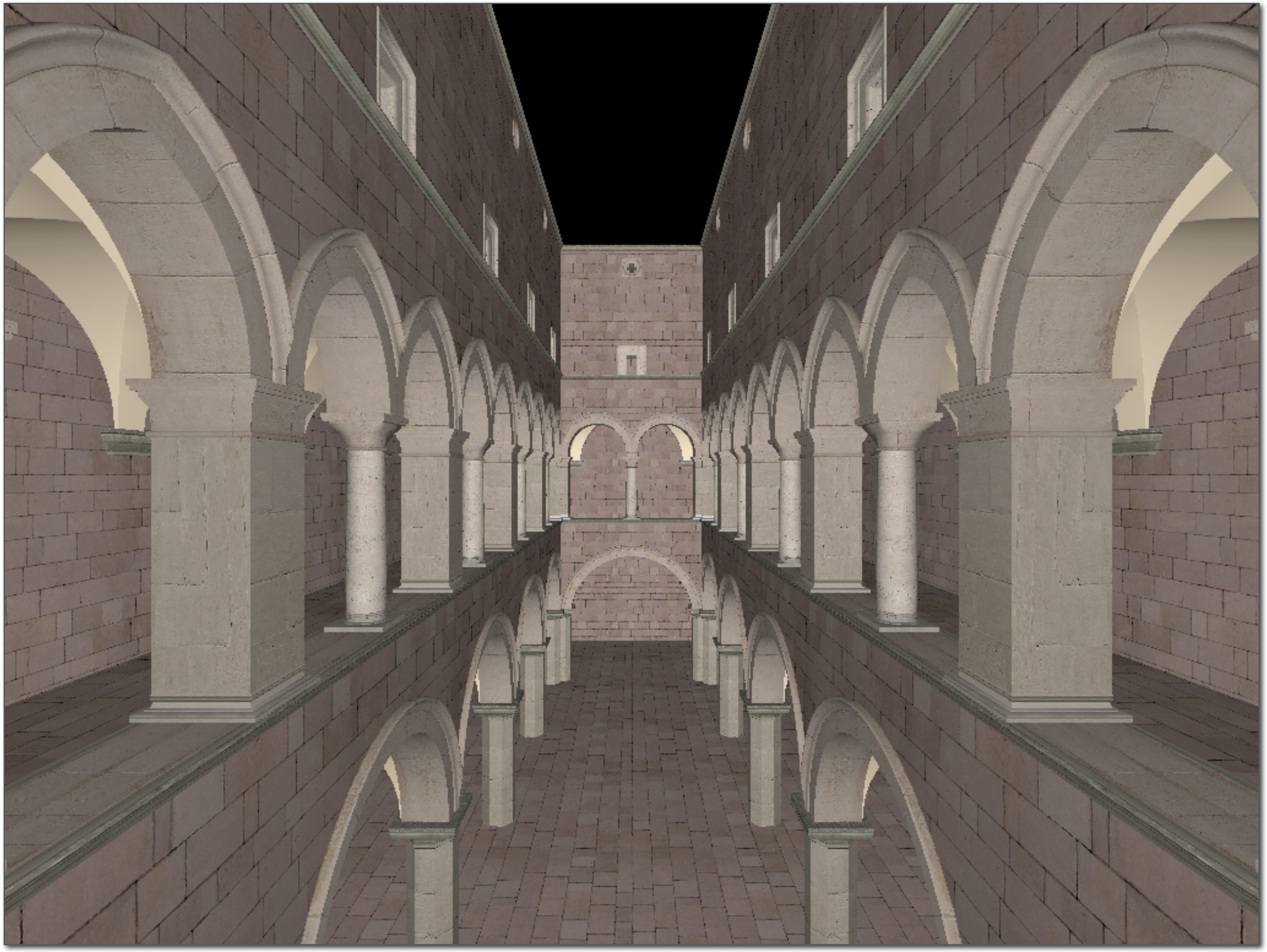

texture image

Another example: Sponza



Notice texture coordinates repeat over surface.

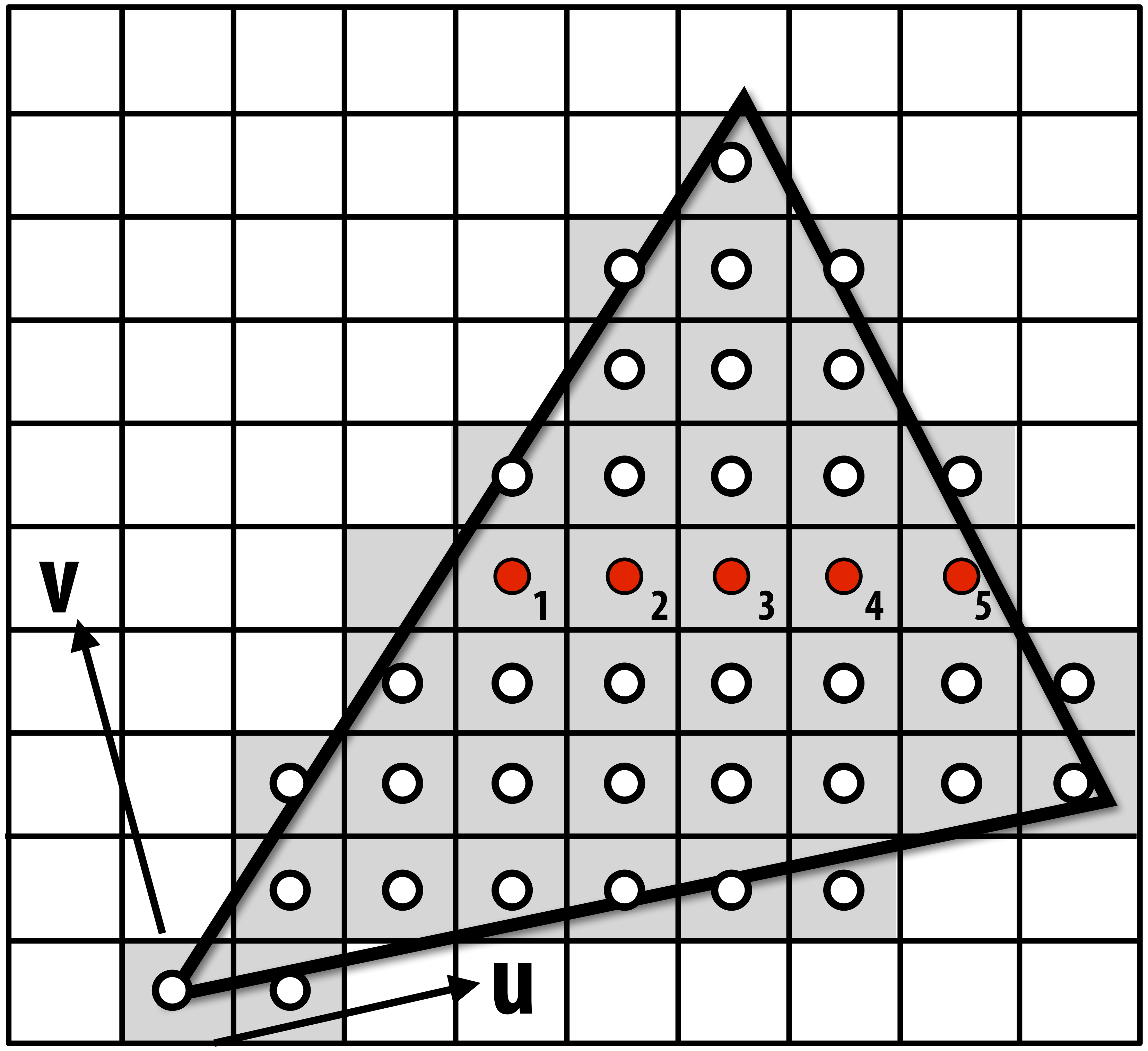
Textured Sponza



Example textures used in Sponza

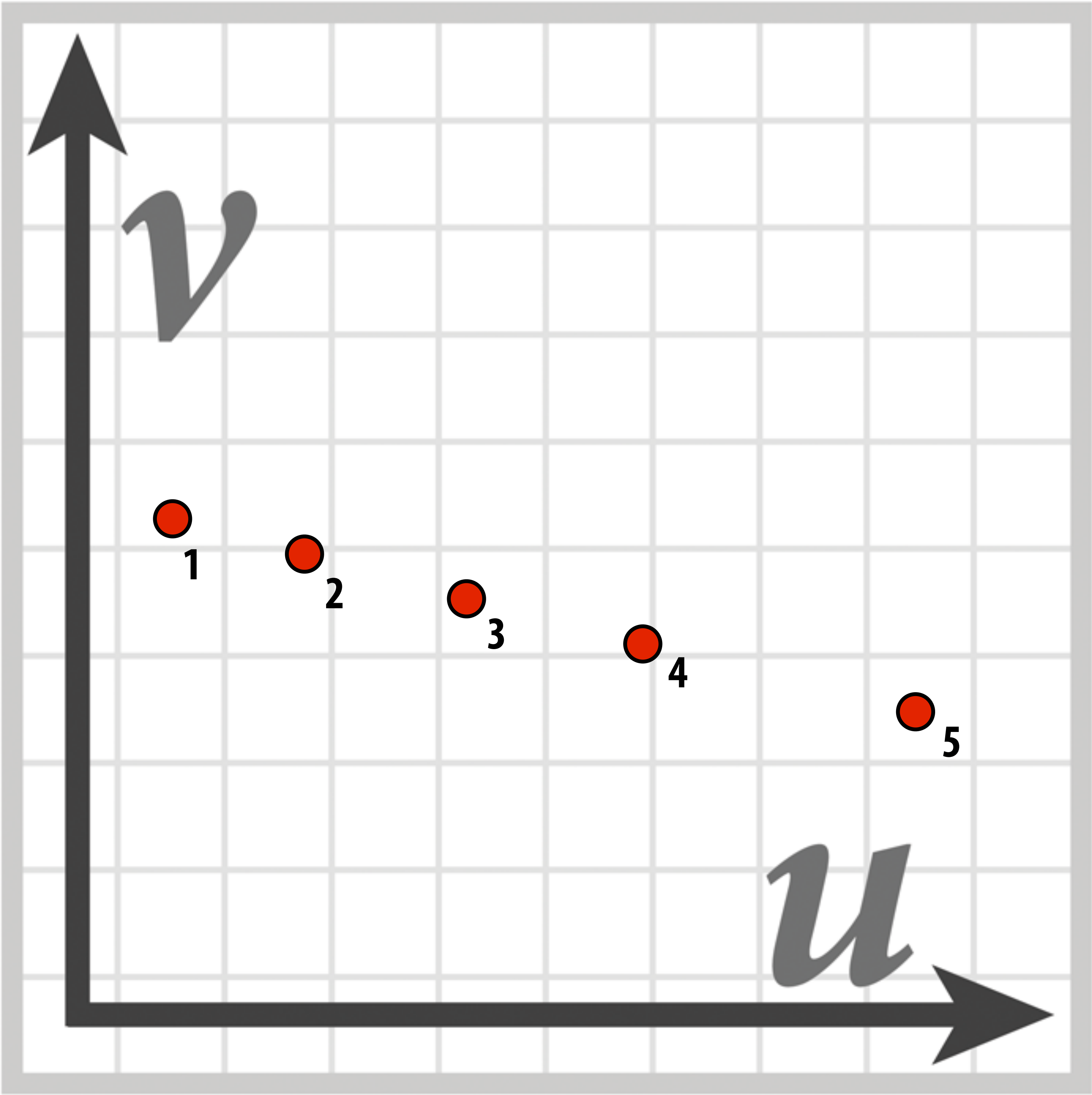
Texture space samples

Sample positions in XY screen space



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

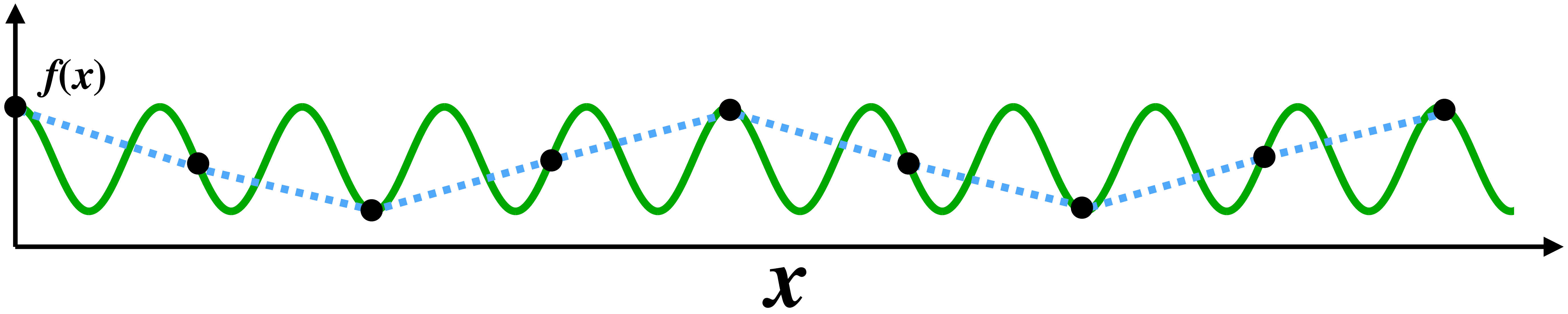
Sample positions in texture space



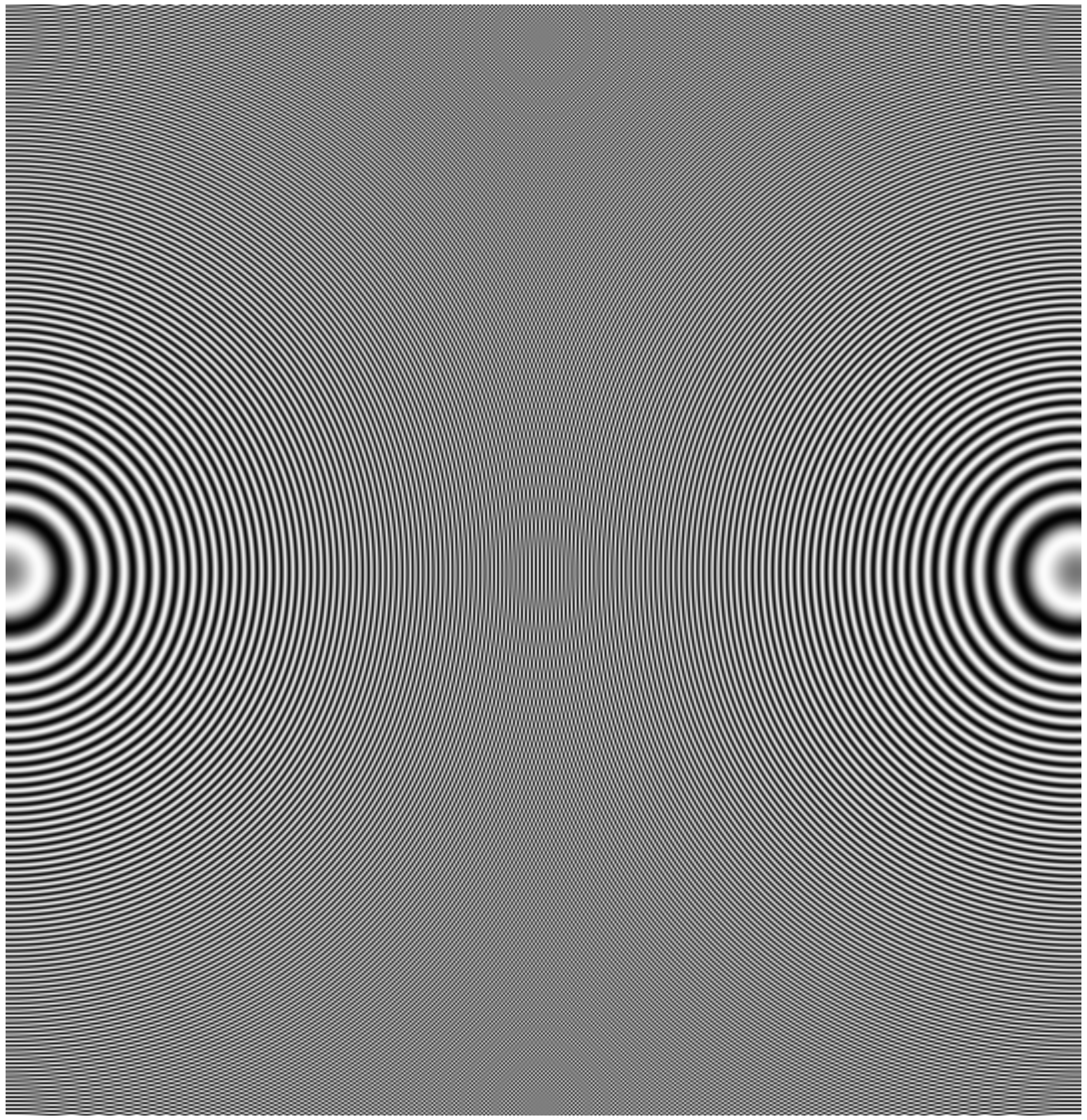
Texture sample positions in texture space (texture function is sampled at these locations)

Recall: aliasing

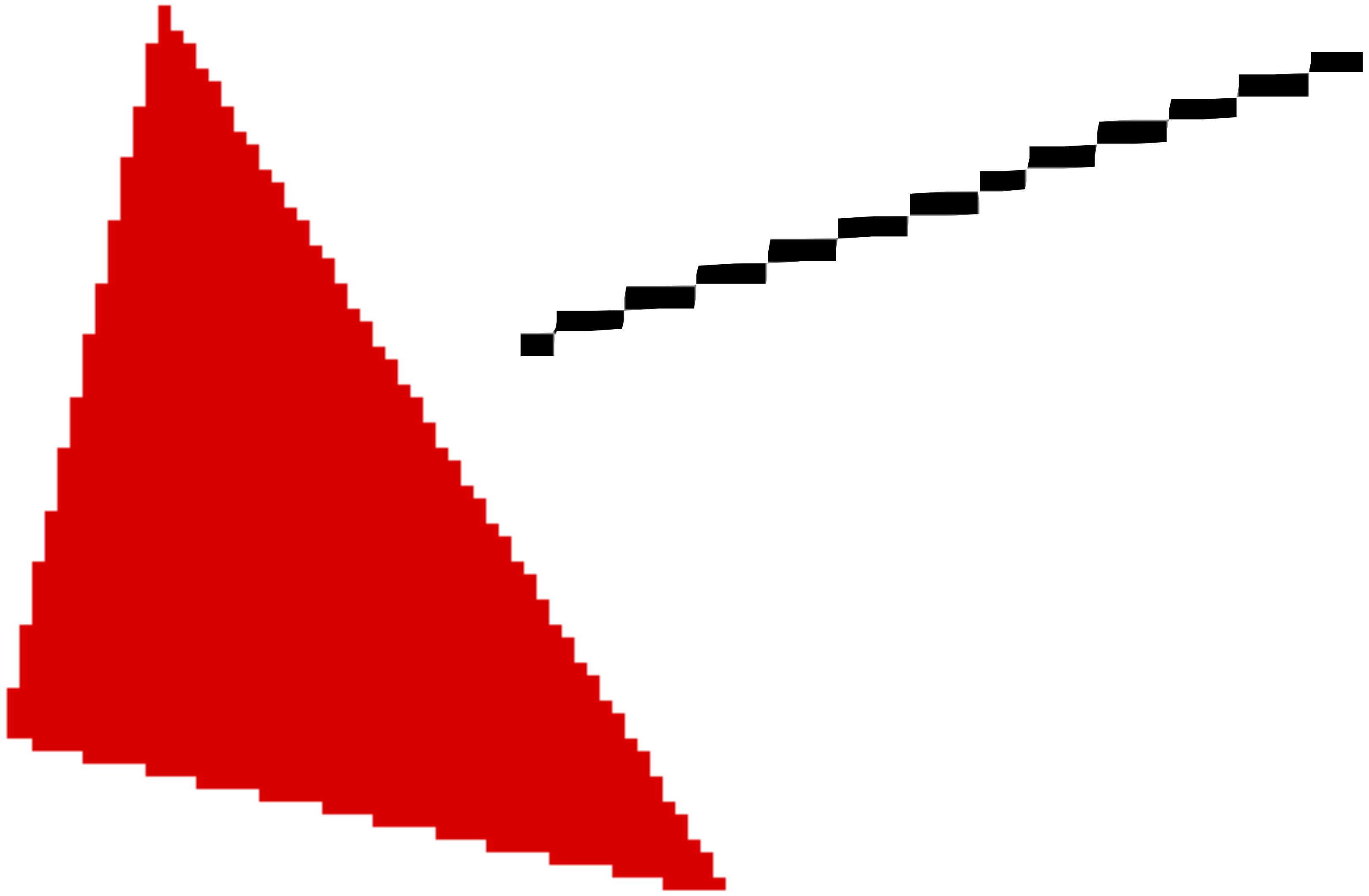
Undersampling a high-frequency signal can result in aliasing



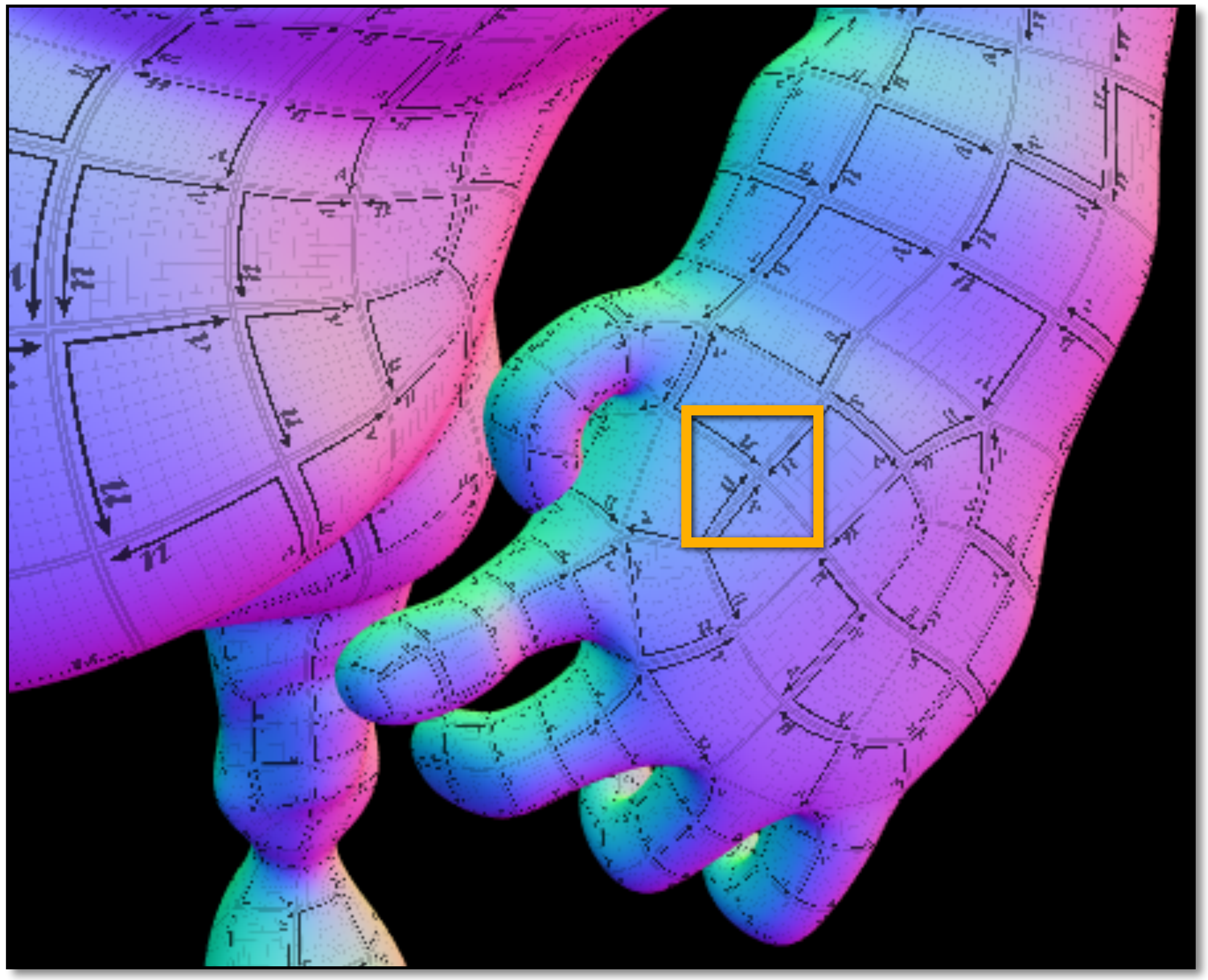
1D example



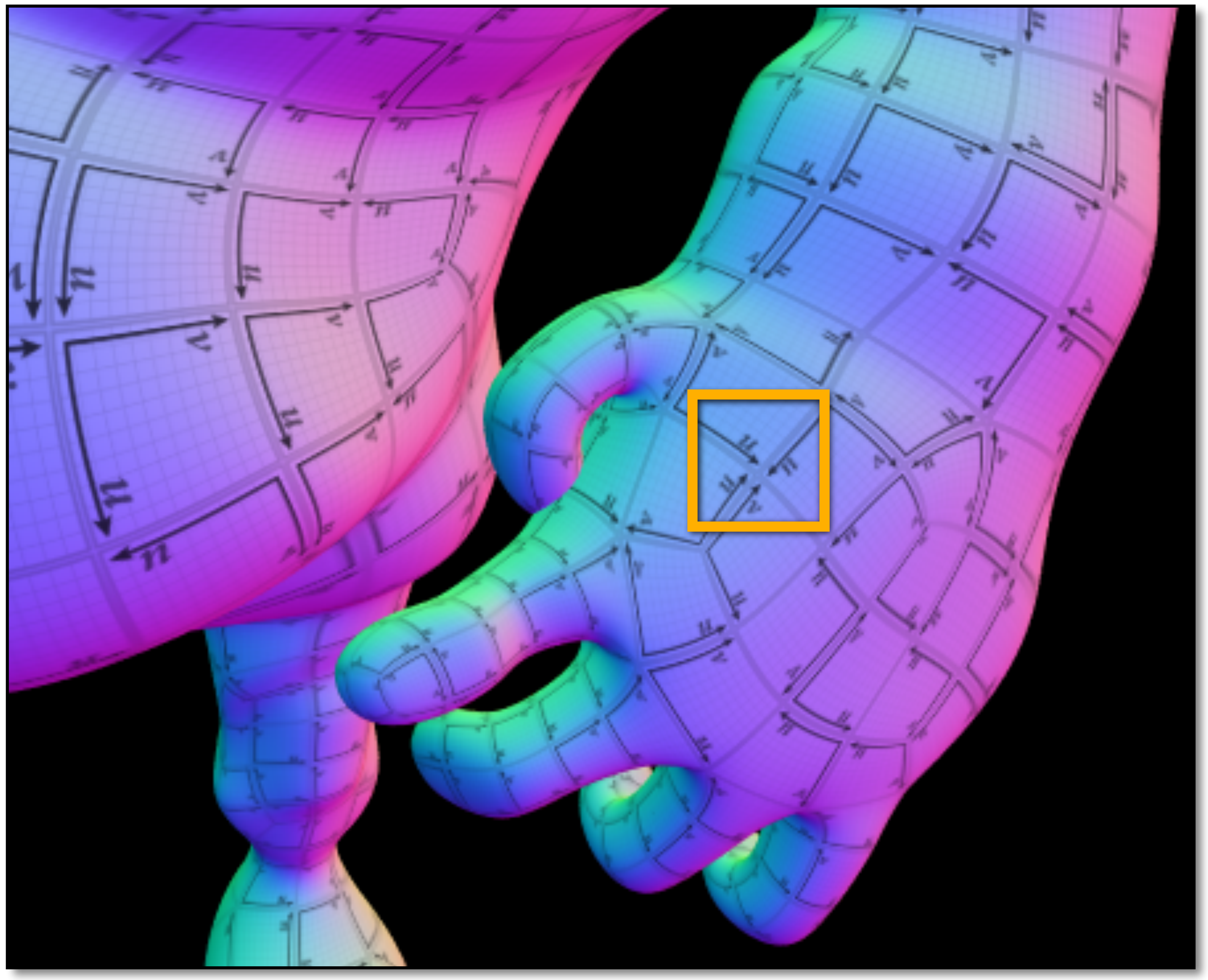
2D examples:
Moiré patterns, jaggies



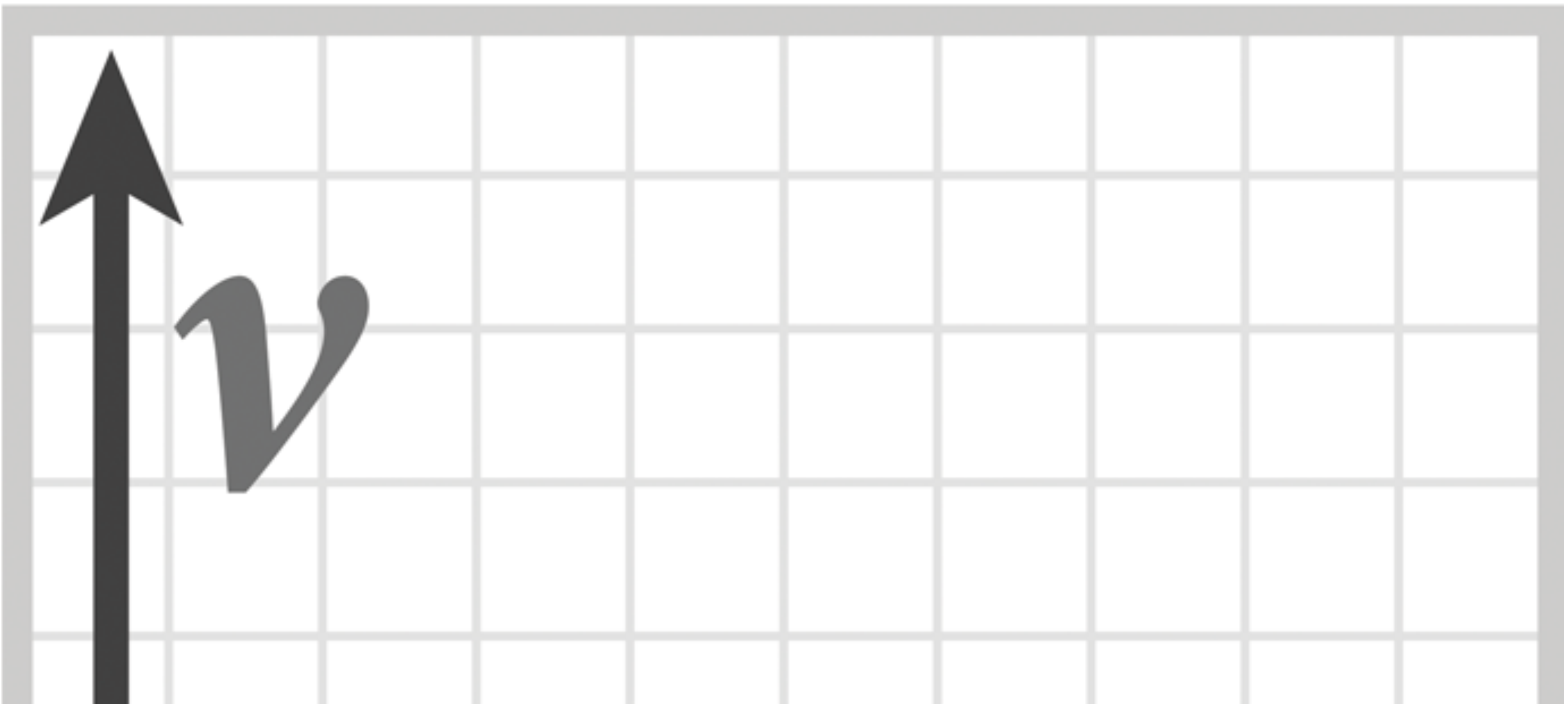
Aliasing due to undersampling texture



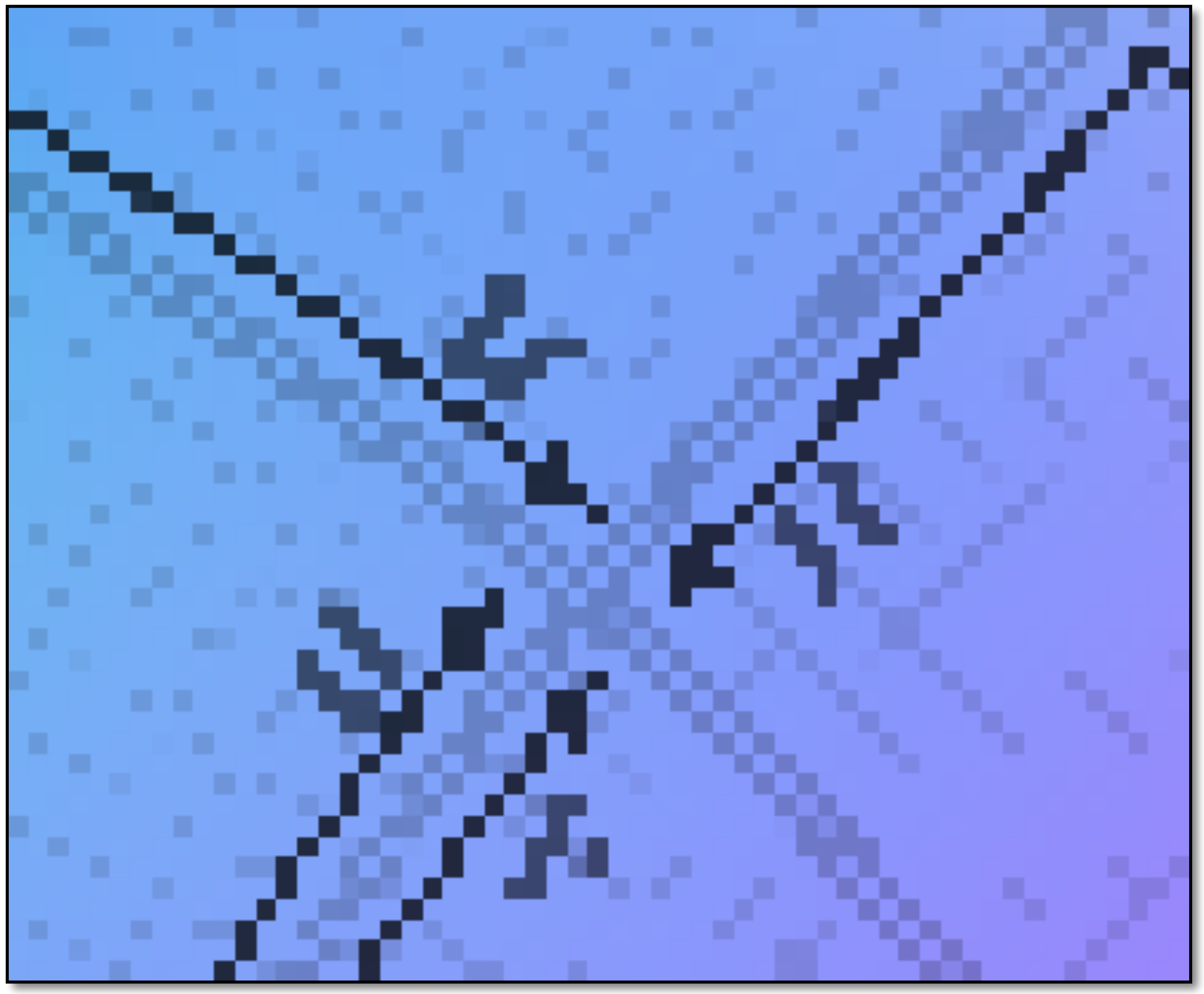
No pre-filtering of texture data
(resulting image exhibits aliasing)



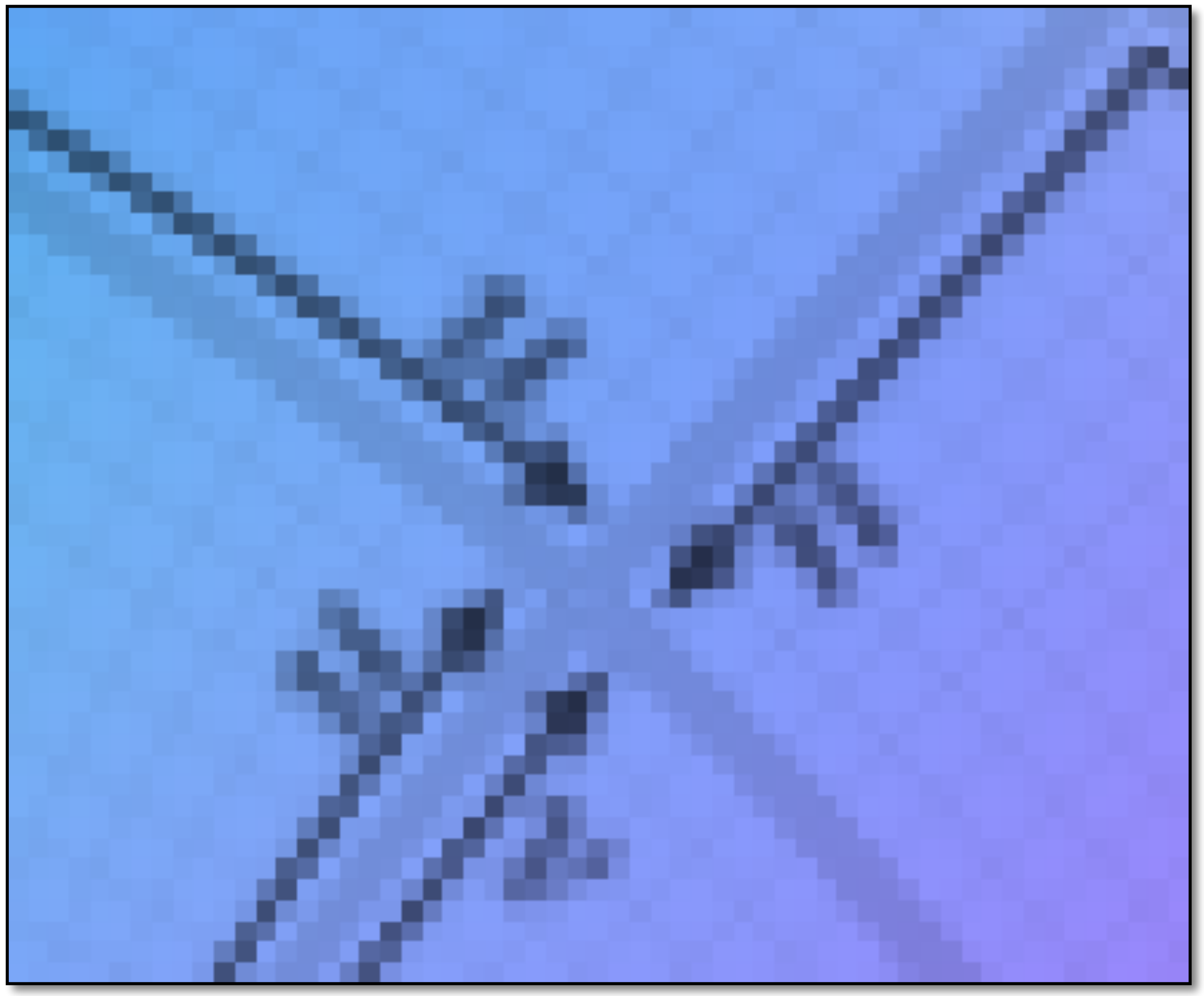
Rendering using pre-filtered texture data



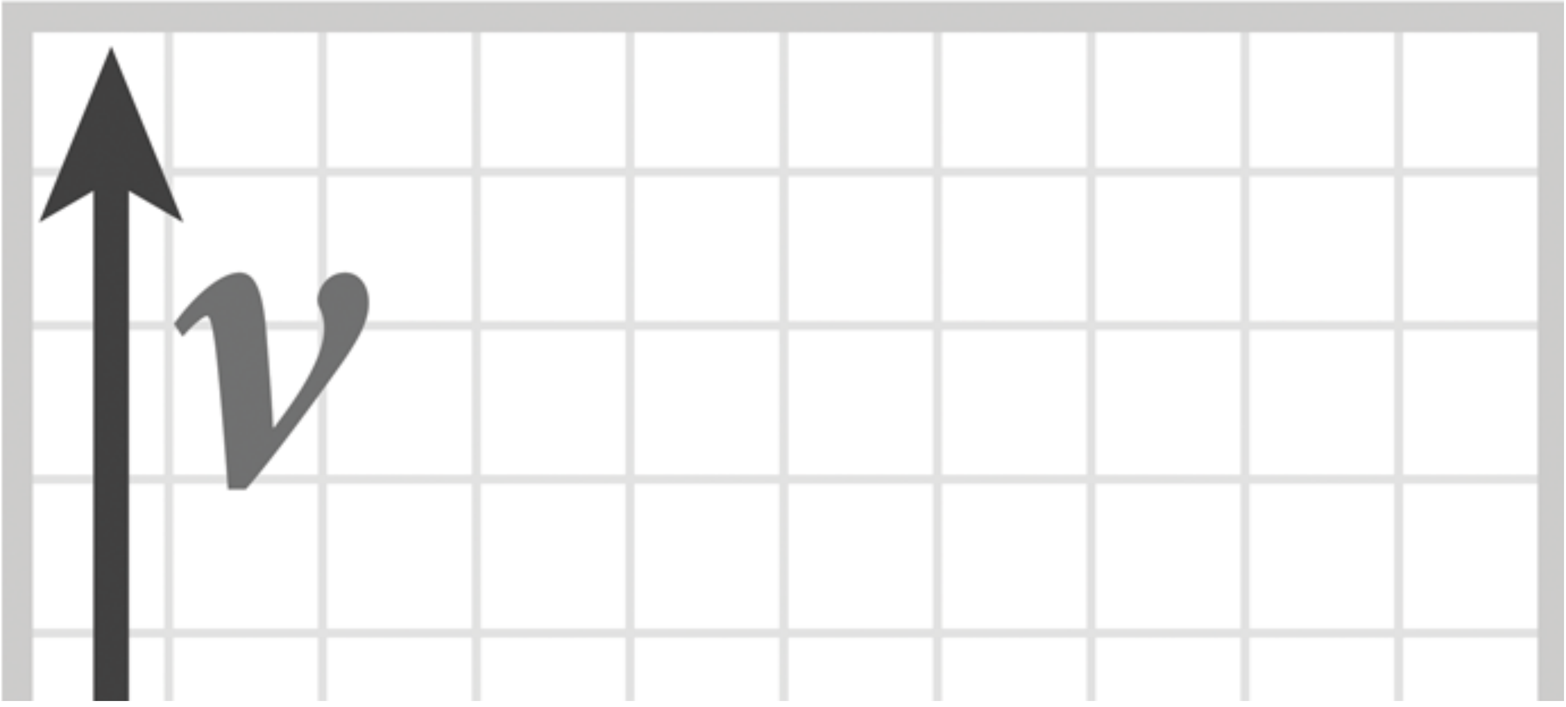
Aliasing due to undersampling (zoom)



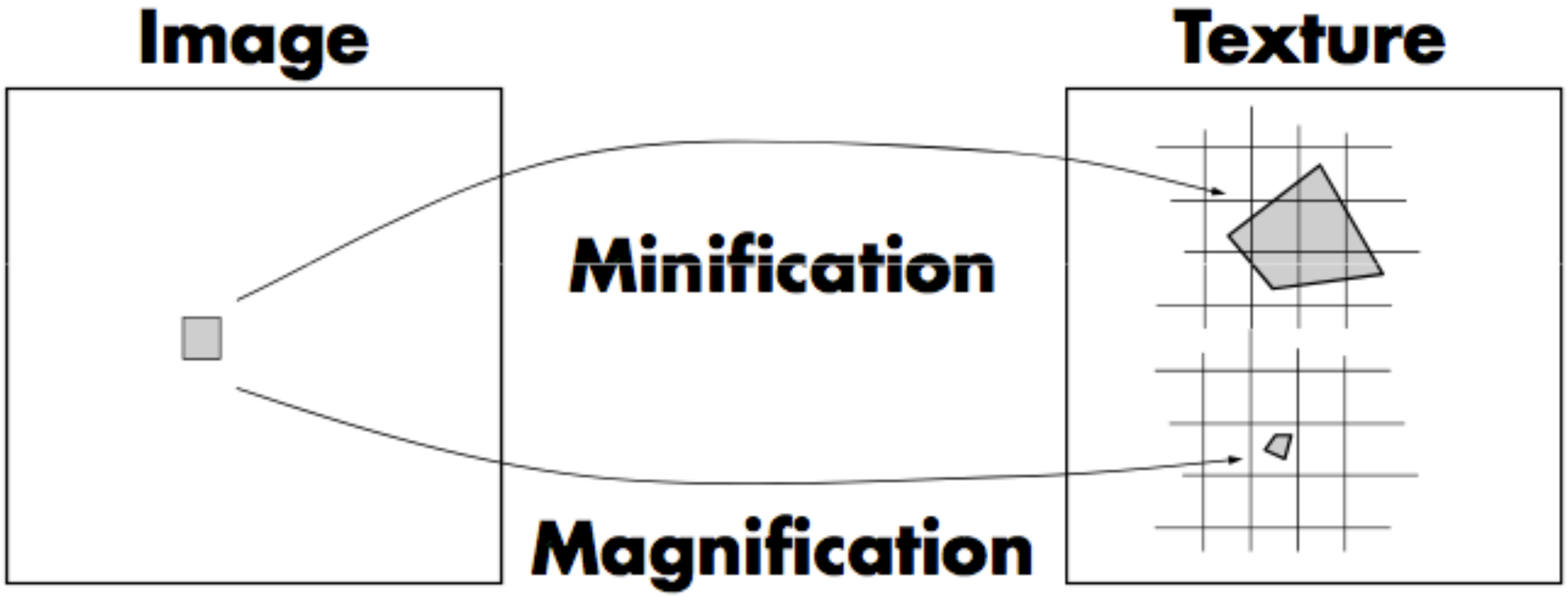
No pre-filtering of texture data
(resulting image exhibits aliasing)



Rendering using pre-filtered texture data



Filtering textures

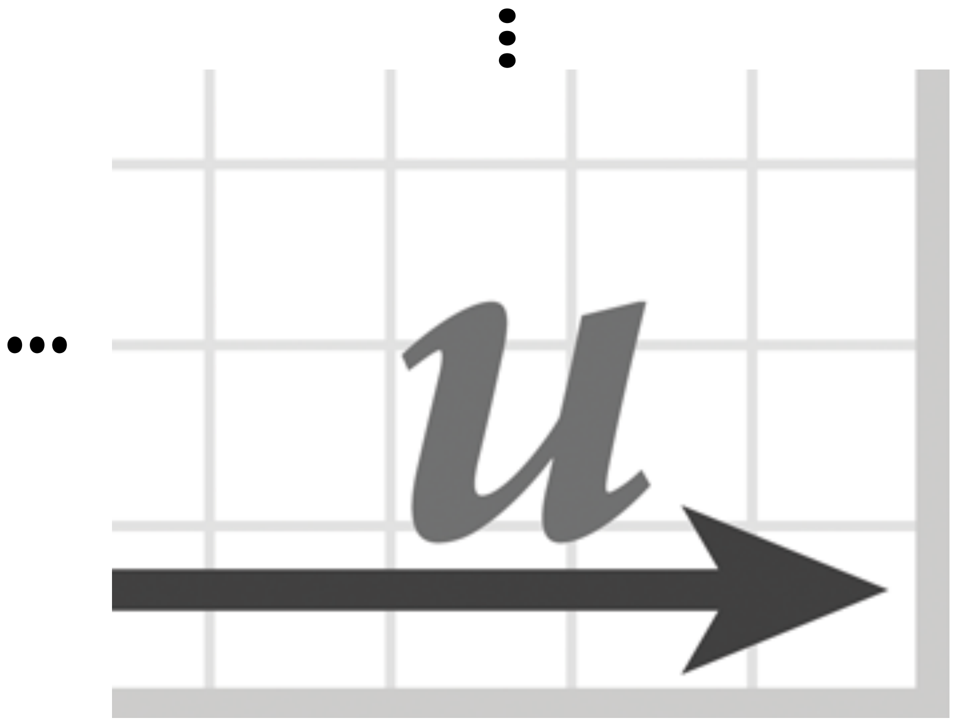


- **Minification:**
  - Area of screen pixel maps to large region of texture (filtering required -- averaging)
  - One texel corresponds to far less than a pixel on screen
  - Example: when scene object is very far away

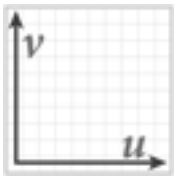
- **Magnification:**
  - Area of screen pixel maps to tiny region of texture (interpolation required)
  - One texel maps to many screen pixels
  - Example: when camera is very close to scene object (need higher resolution texture map)

Figure credit: Akeley and Hanrahan

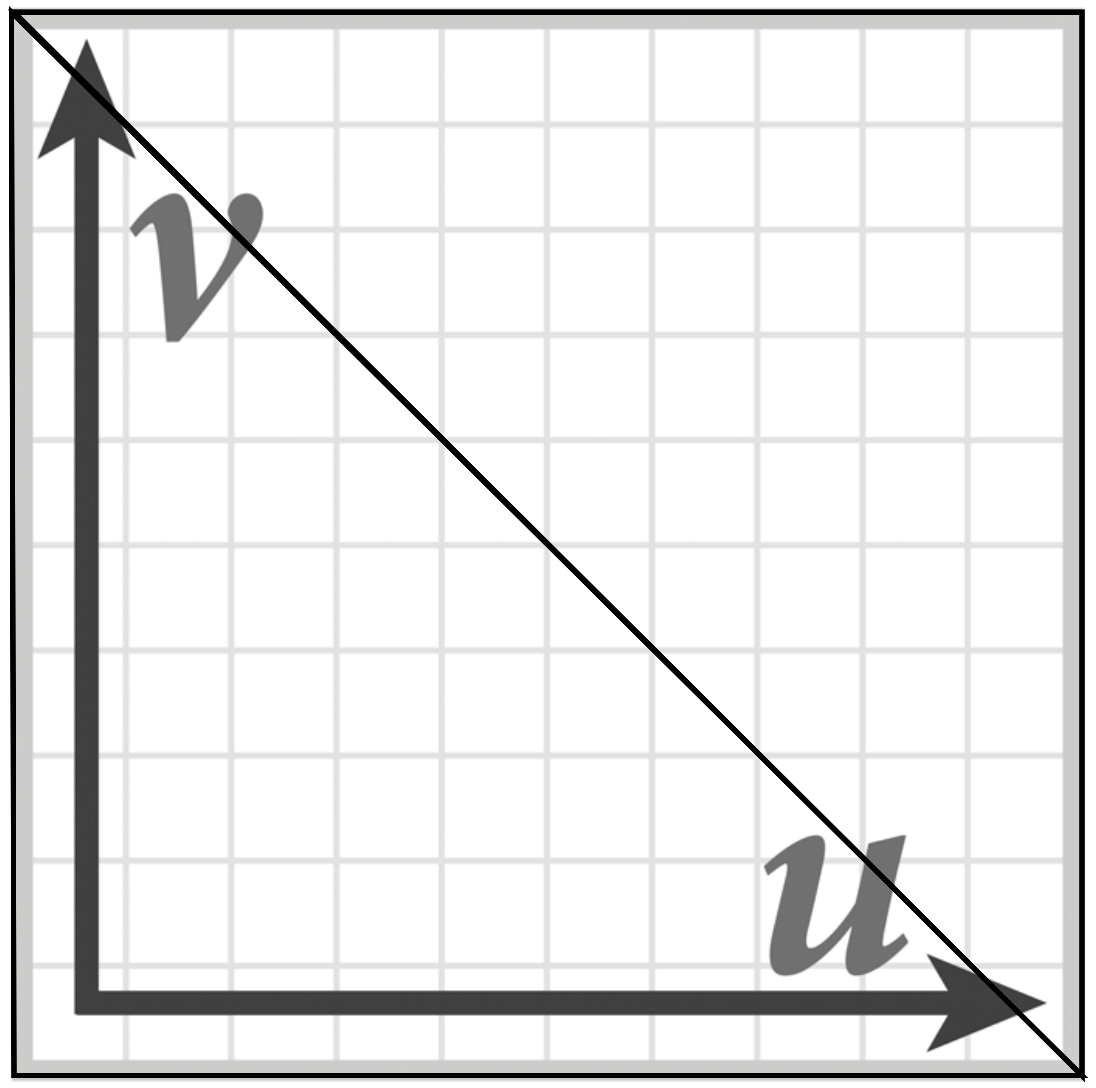
Filtering textures



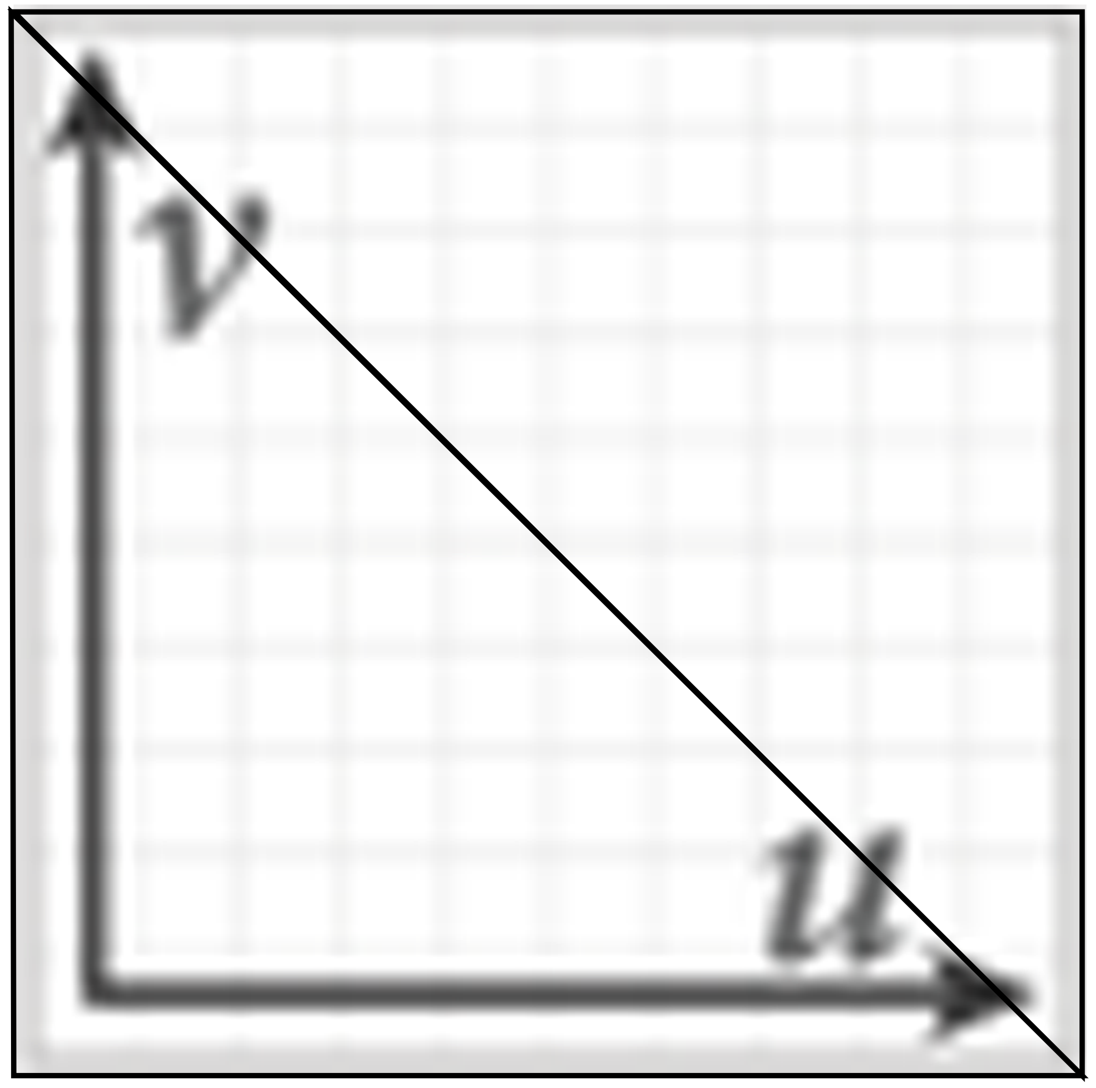
Actual texture: 700x700 image
(only a crop is shown)



Actual texture: 64x64 image

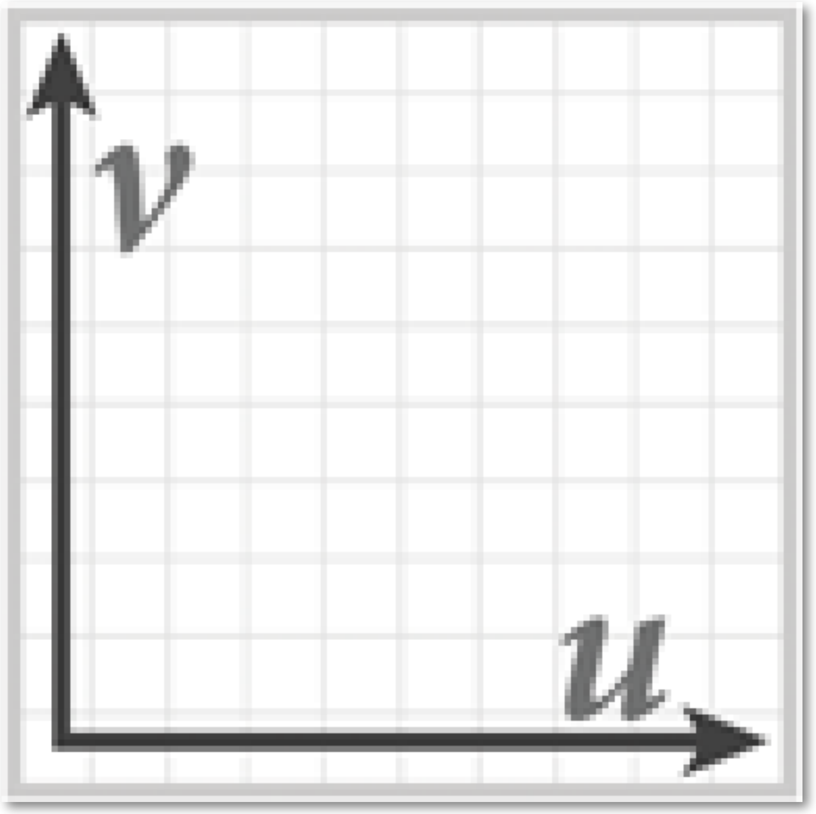


Texture minification

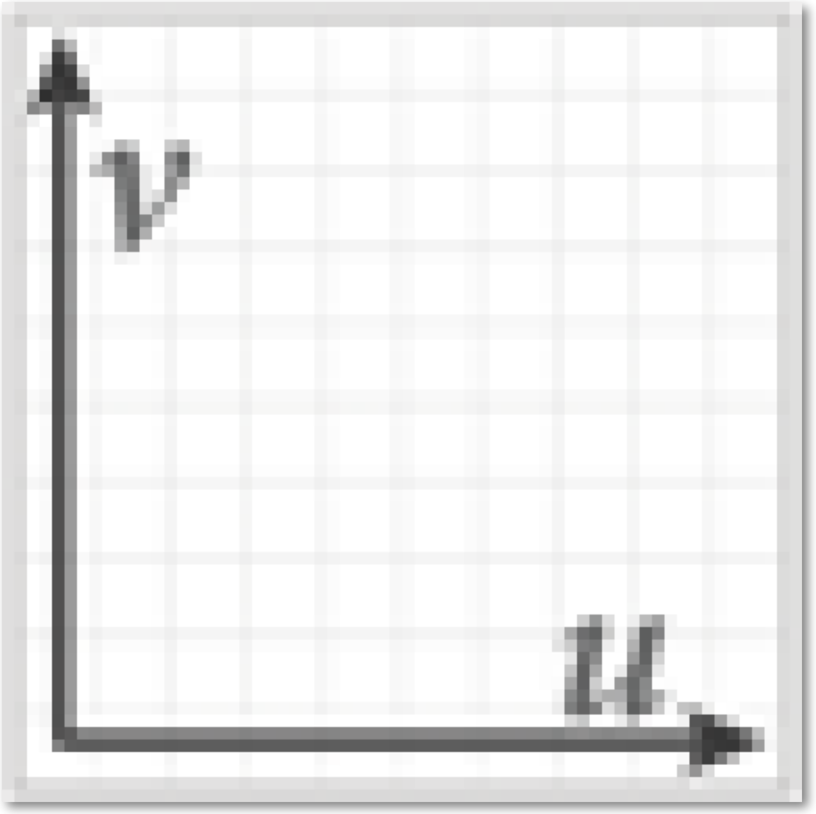


Texture magnification

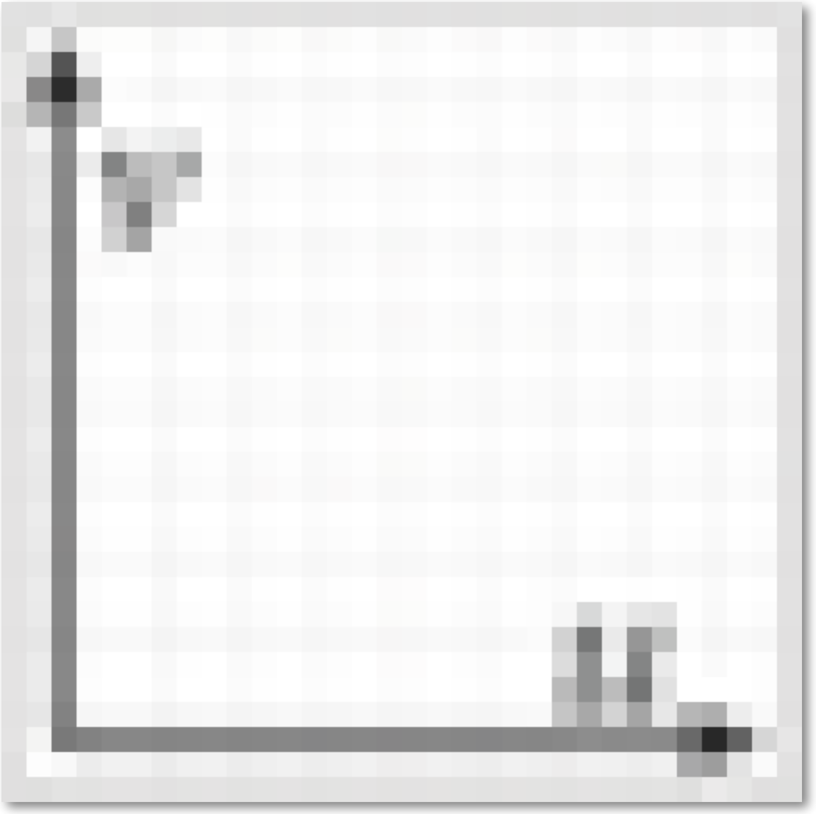
Mipmap (L. Williams 83)



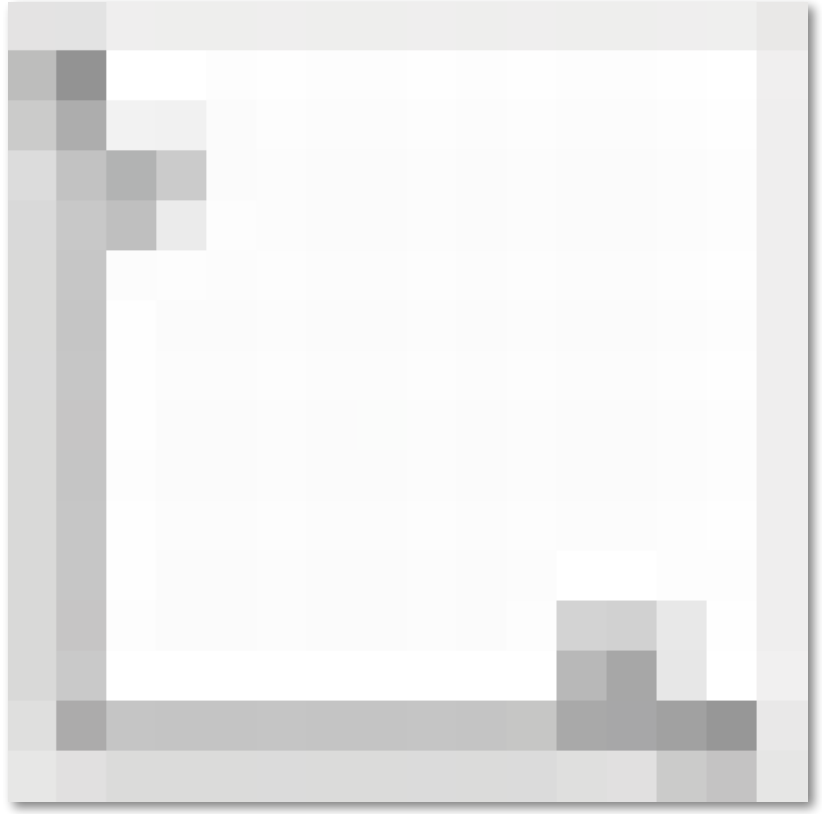
Level 0 = 128x128



Level 1 = 64x64

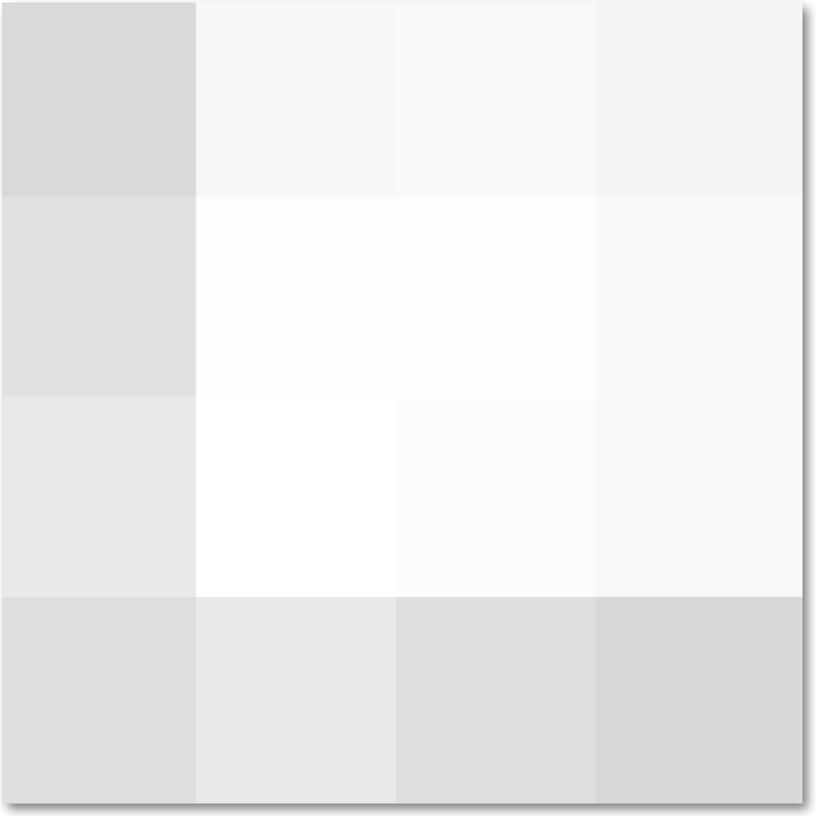


Level 2 = 32x32

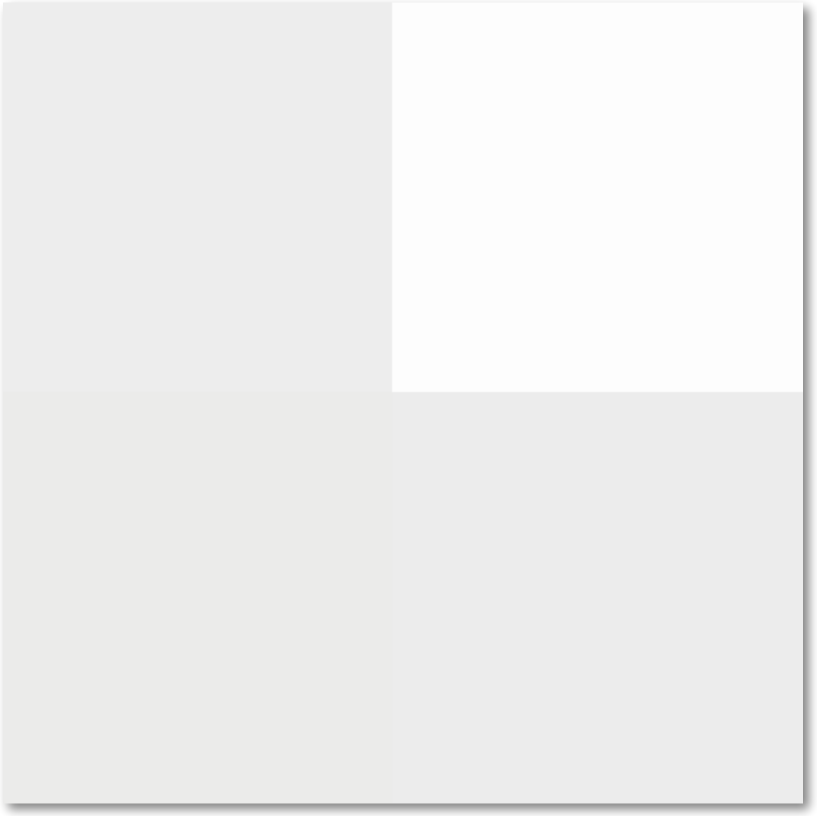


Level 3 = 16x16

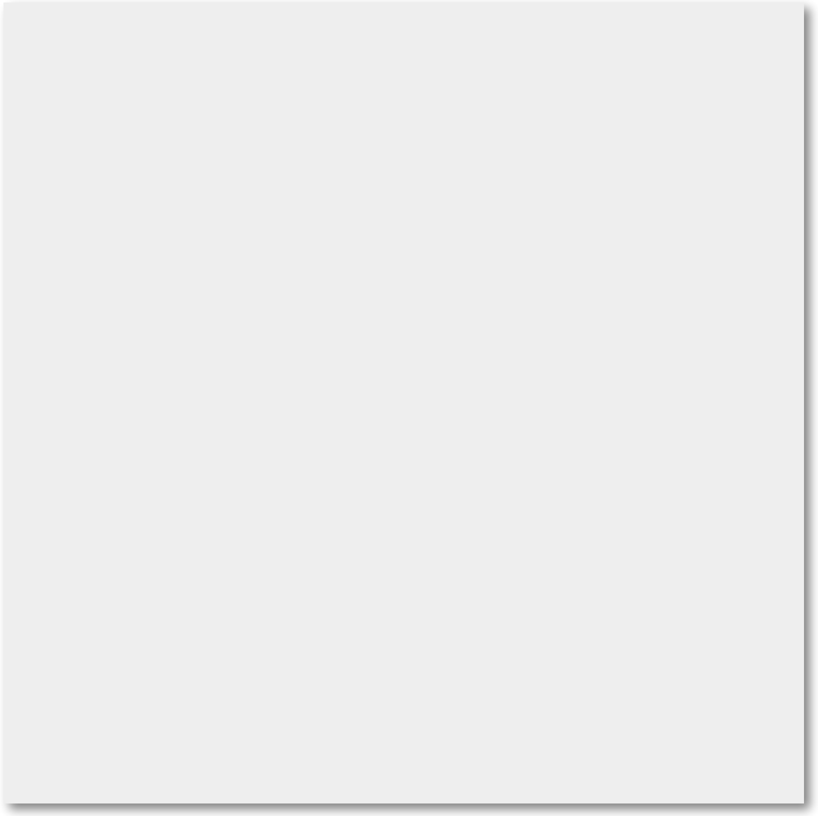
Level 4 = 8x8



Level 5 = 4x4



Level 6 = 2x2



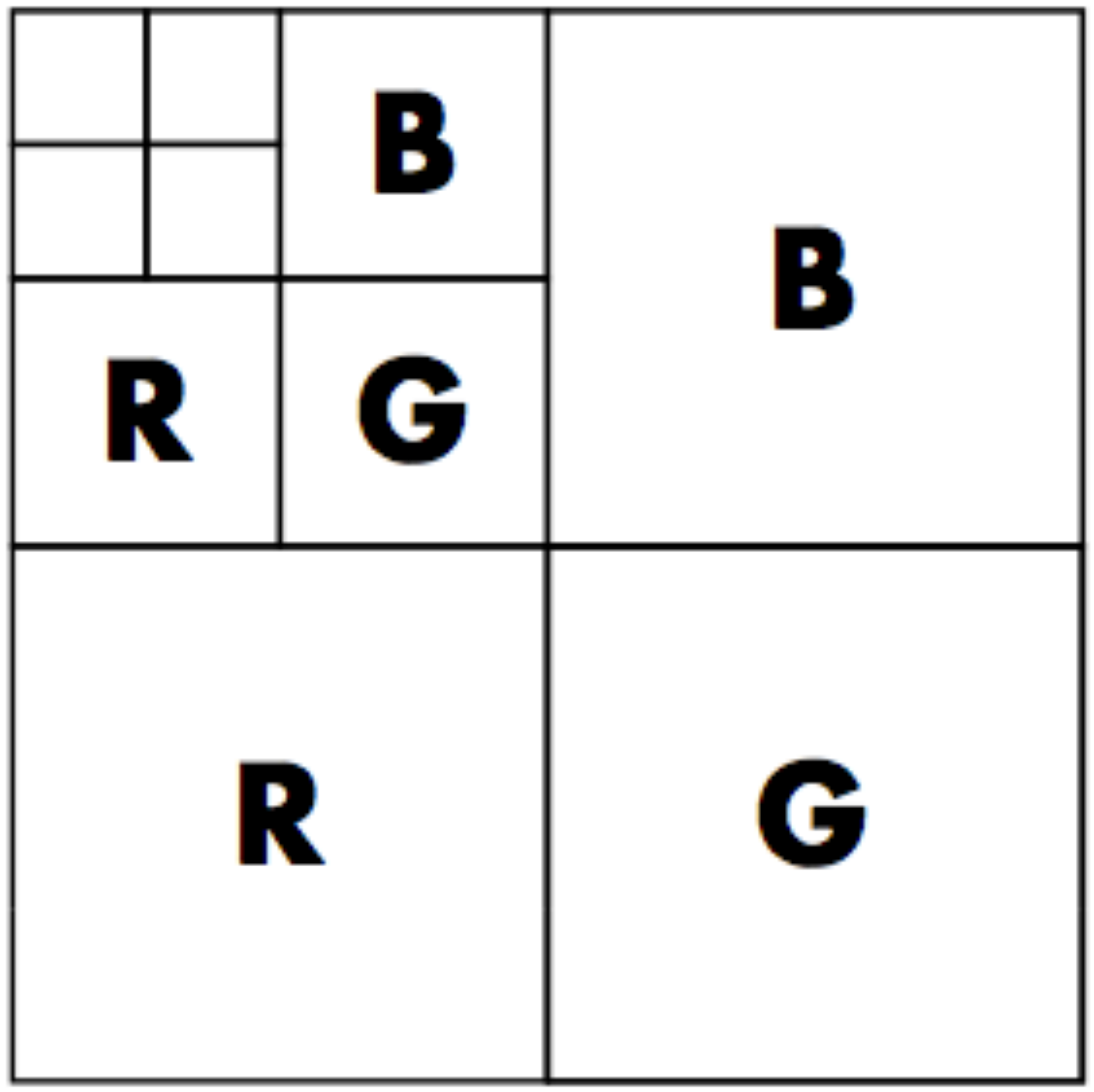
Level 7 = 1x1

Idea: prefilter texture data to remove high frequencies

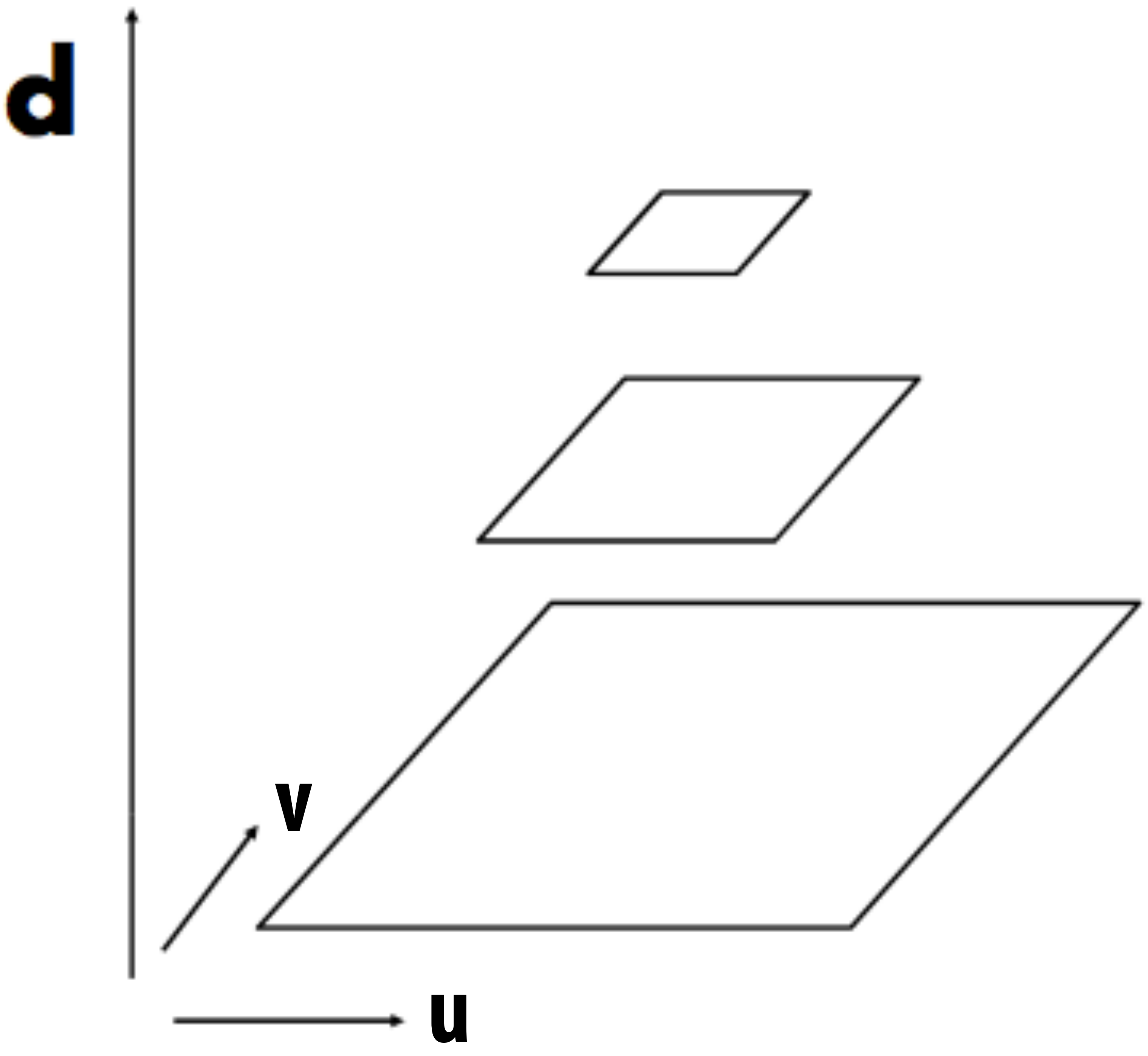
Texels at higher levels store integral of the texture function over a region of texture space (downsampled images)

Texels at higher levels represent low-pass filtered version of original texture signal

Mipmap (L. Williams 83)



Williams' original proposed
mip-map layout

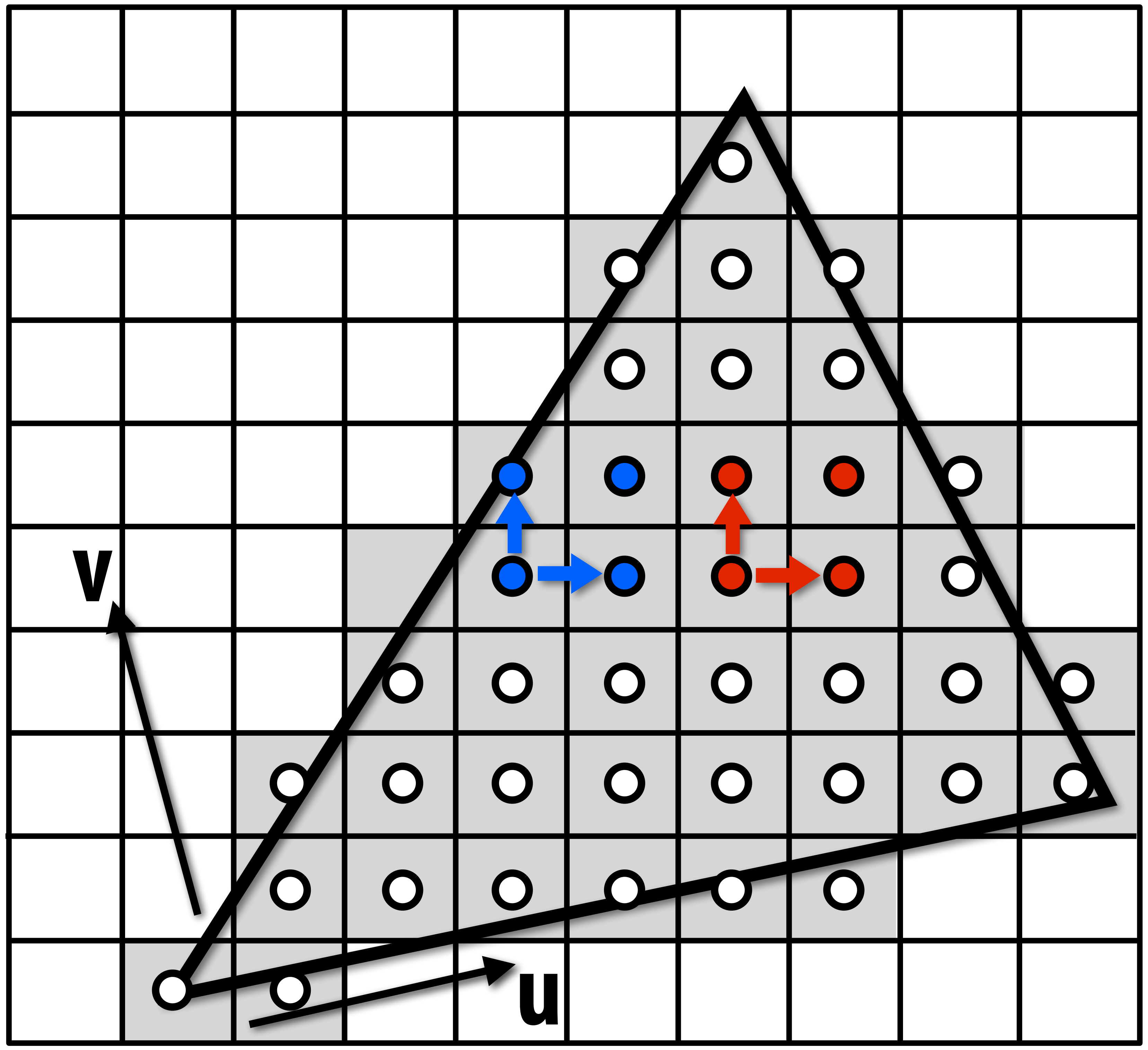


"Mip hierarchy"
level = *d*

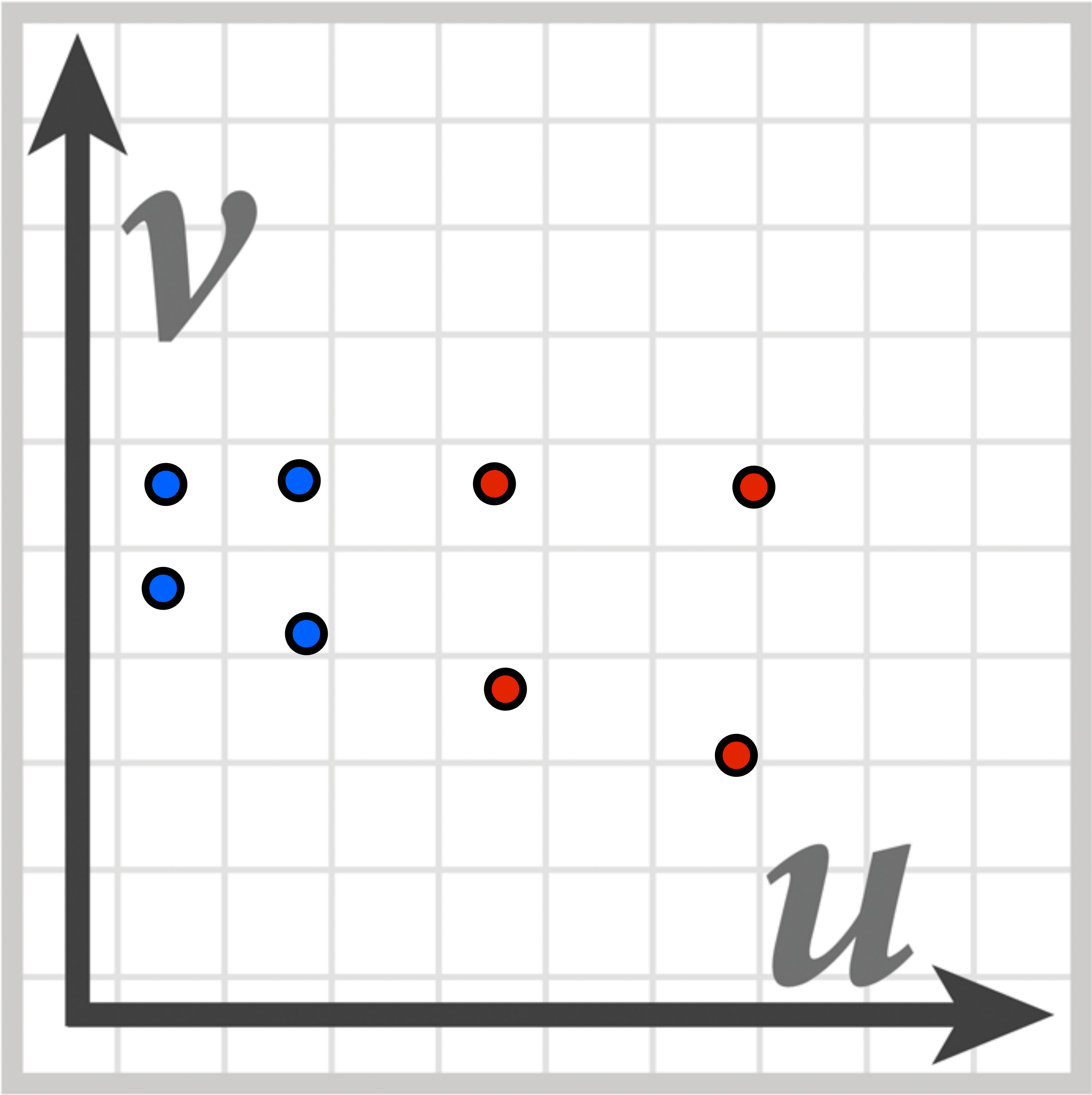
Slide credit: Akeley and Hanrahan

Computing *d*

Compute differences between texture coordinate values of neighboring fragments



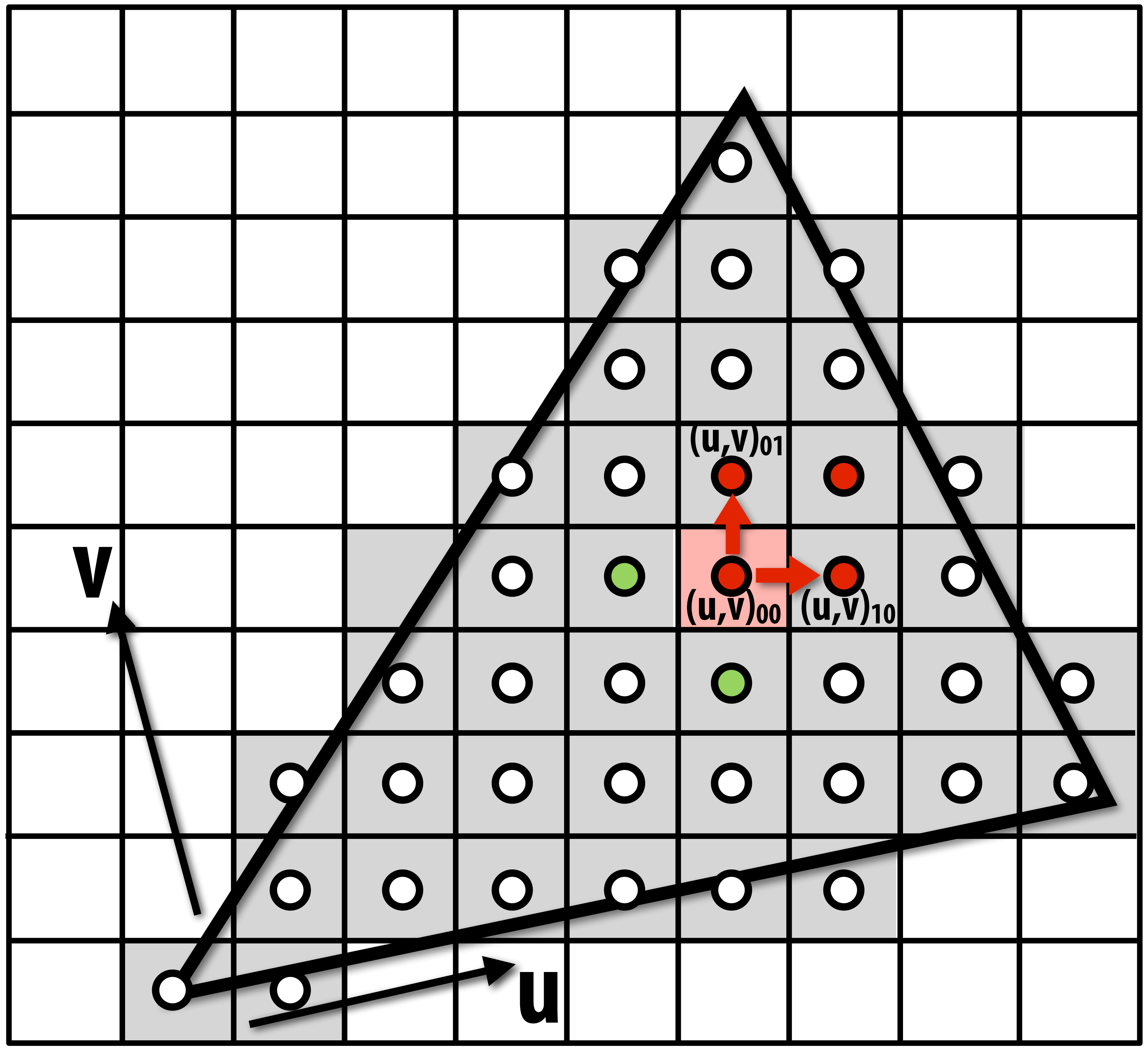
Screen space



Texture space

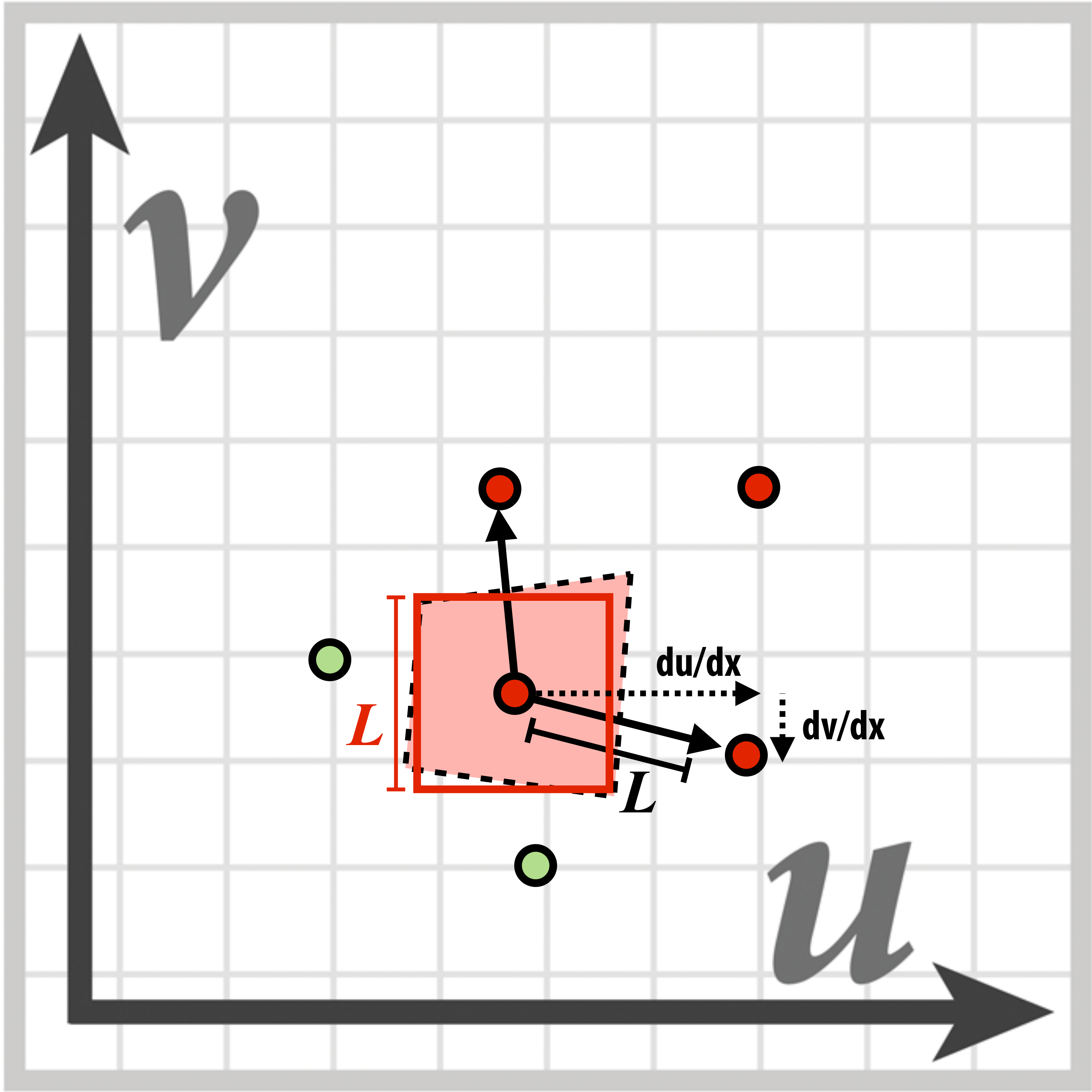
Computing *d*

Compute differences between texture coordinate values of neighboring fragments



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

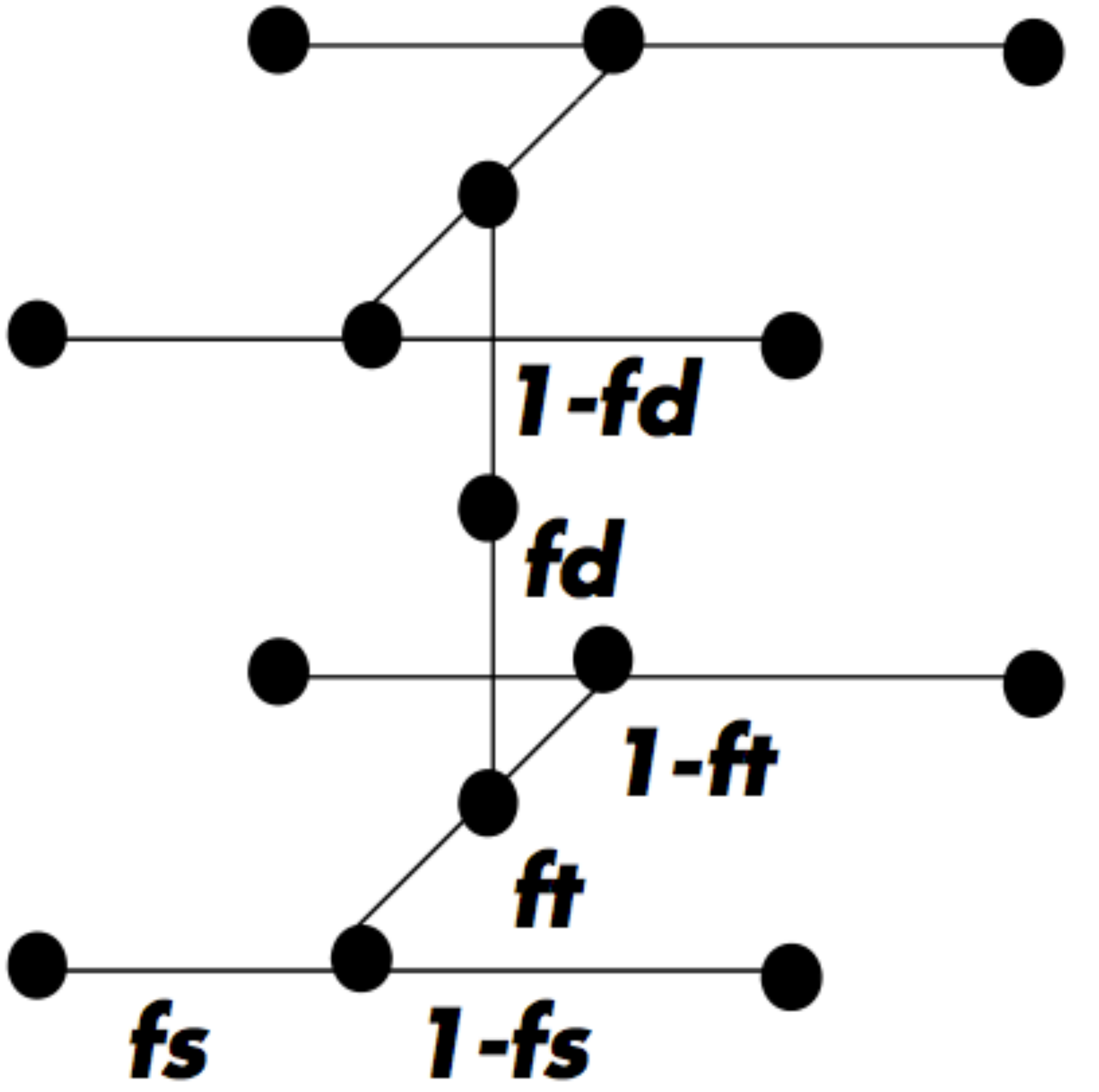
$du/dy = u_{01}-u_{00}$ $dv/dy = v_{01}-v_{00}$



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

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

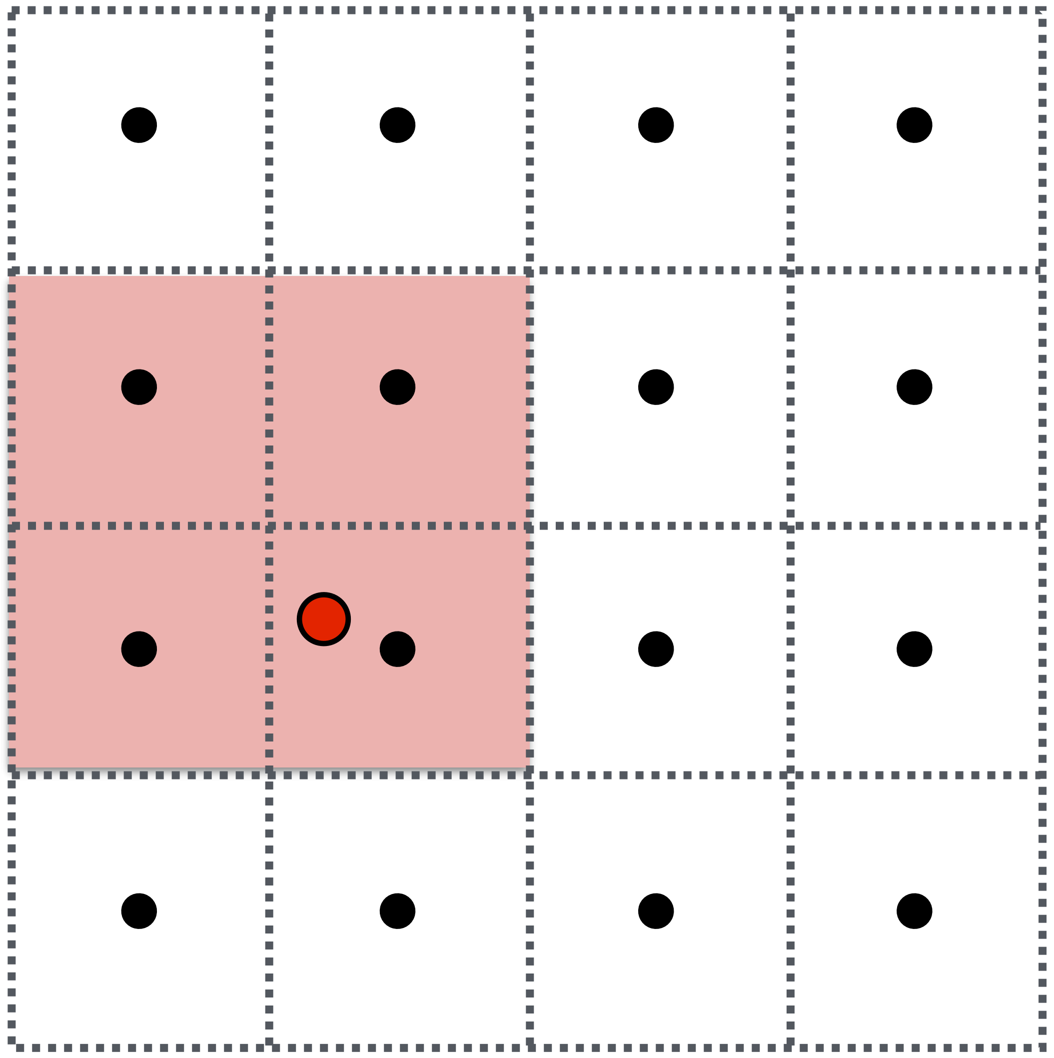
"Tri-linear" filtering



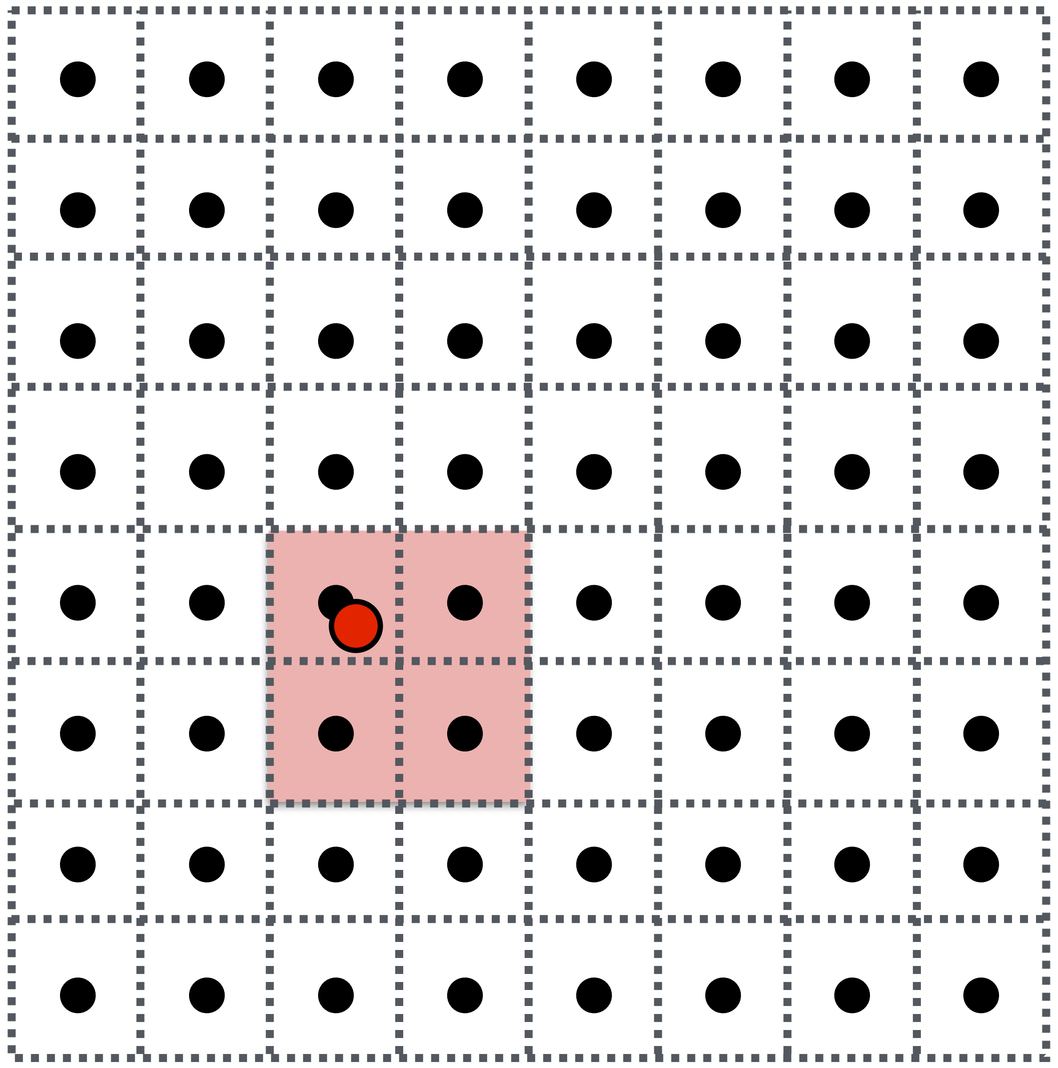
$$lerp(t, v_1, v_2) = v_1 + t(v_2 - v_1)$$

Bilinear resampling: 3 lerps (3 mul + 6 add)

Trilinear resampling: 7 lerps (7 mul + 14 add)



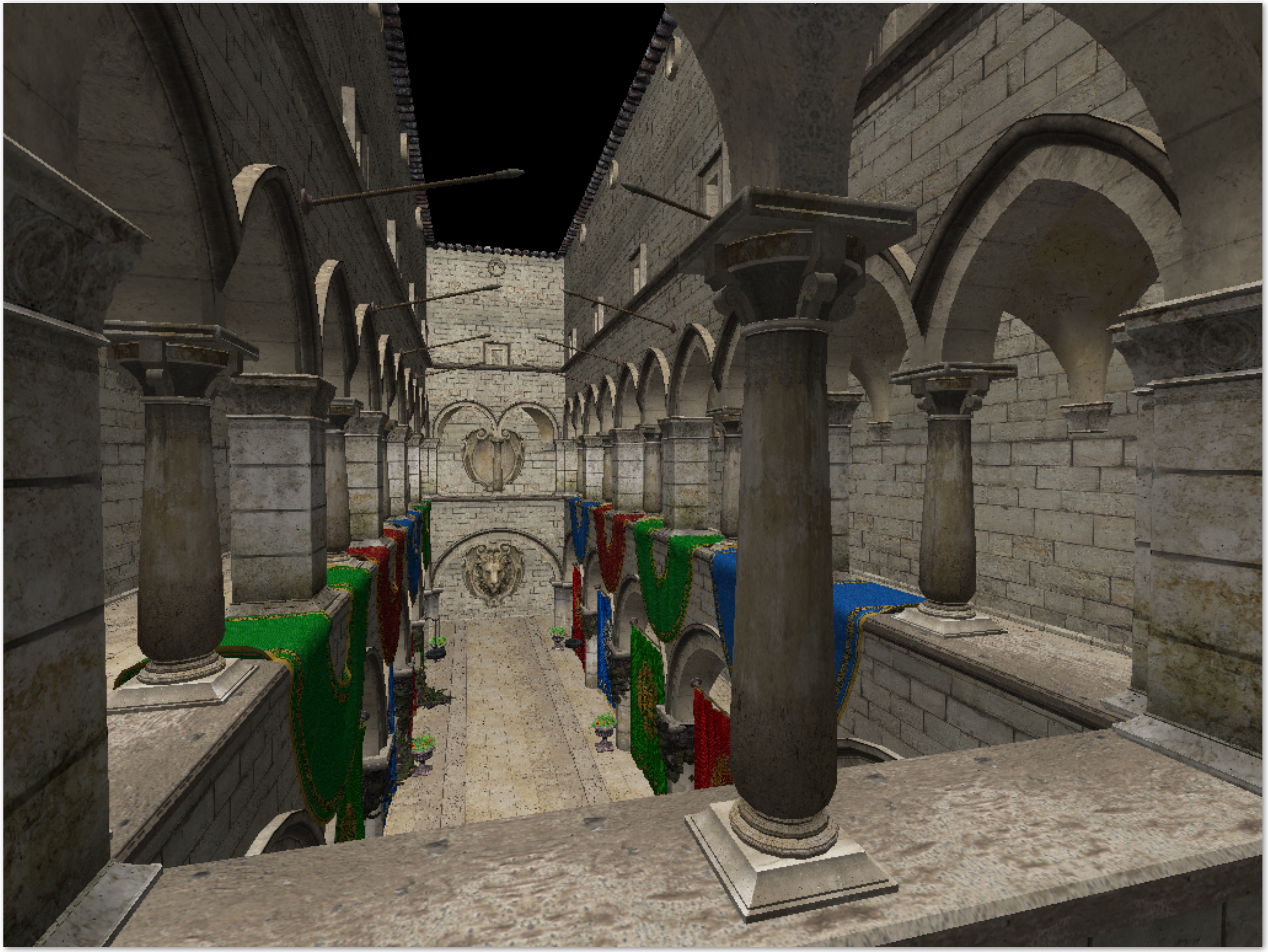
mip-map texels: level d+1



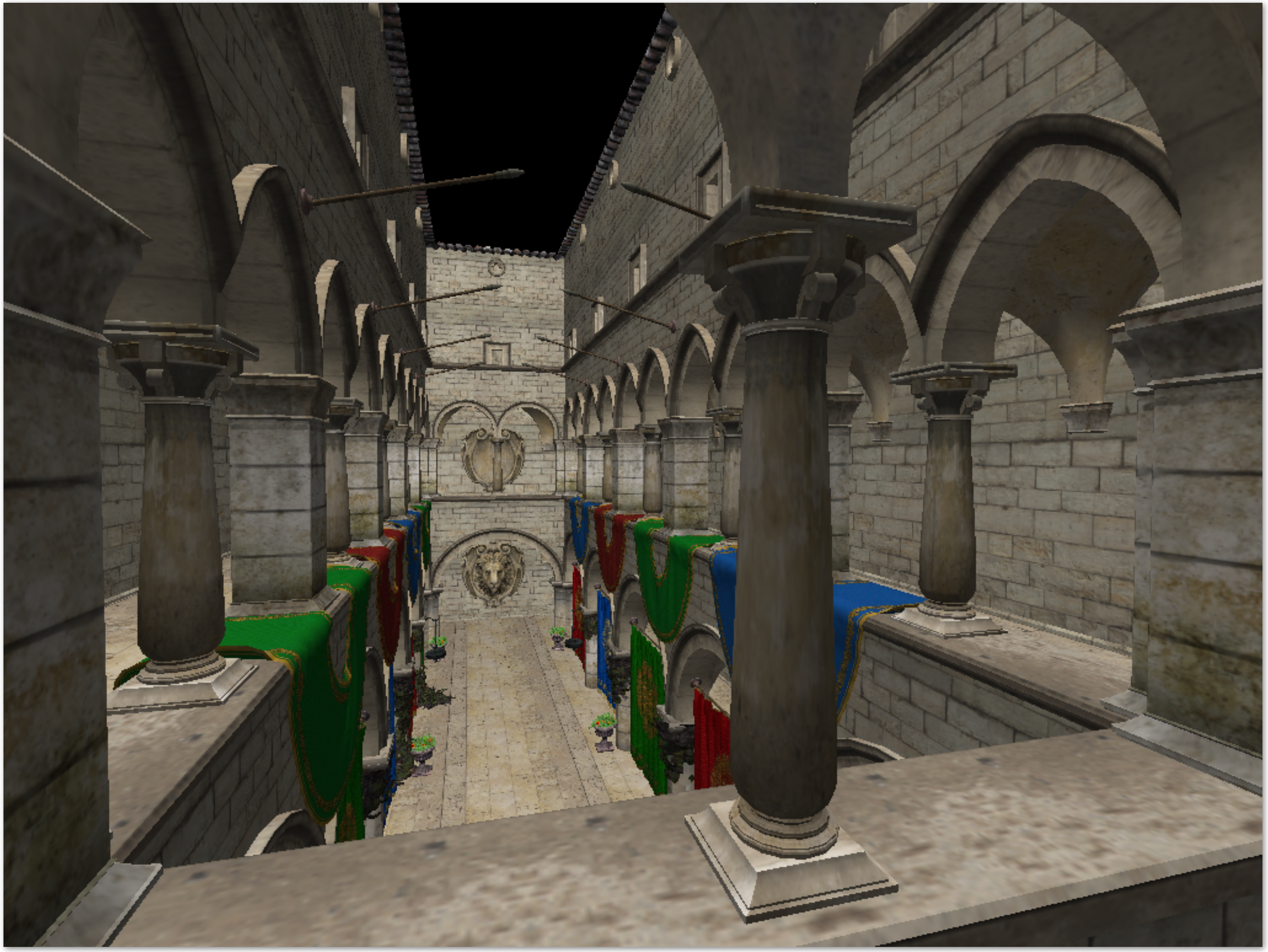
mip-map texels: level d

Figure credit: Akeley and Hanrahan

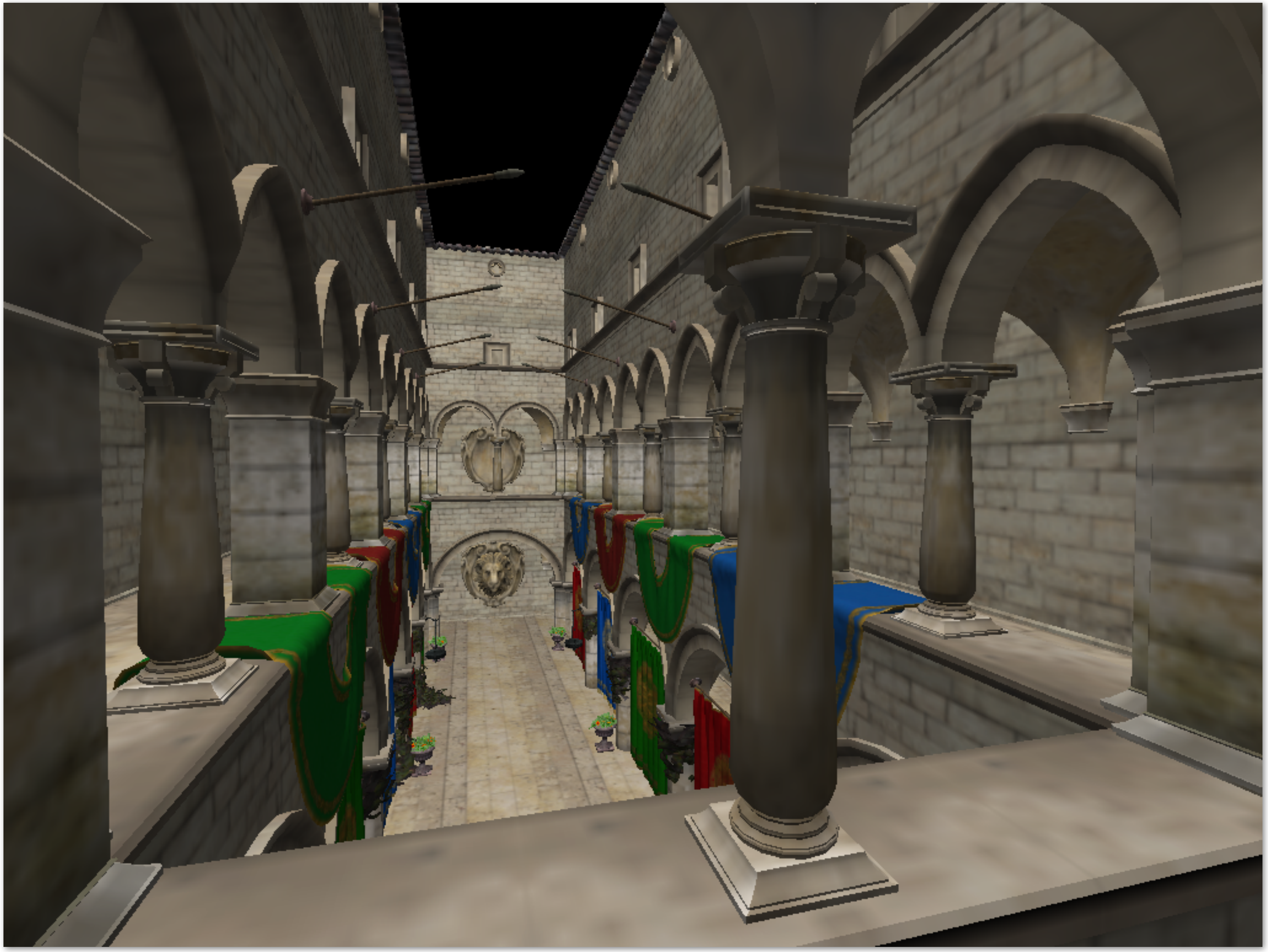
Sponza (bilinear resampling at level 0)



Sponza (bilinear resampling at level 2)

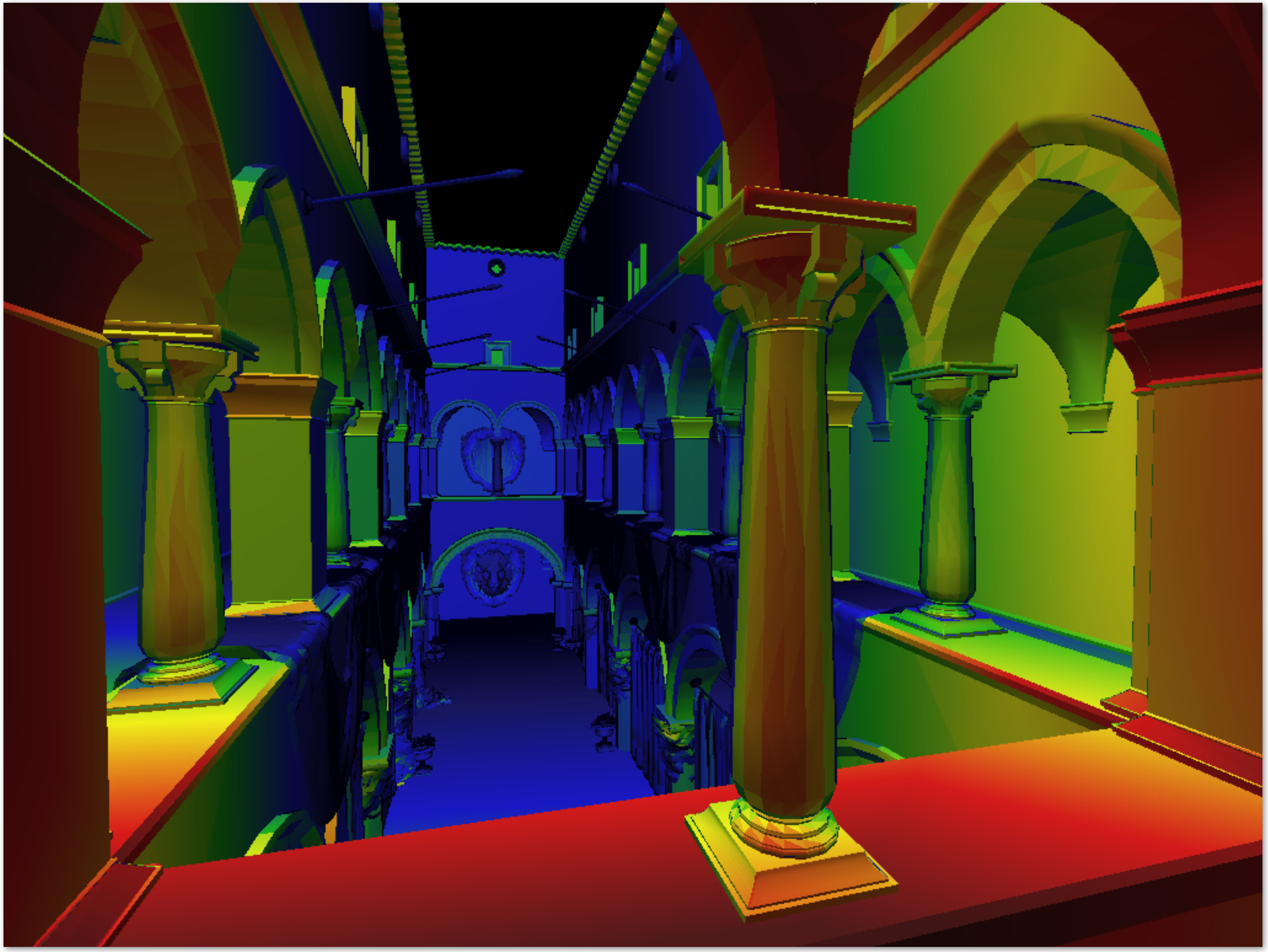


Sponza (bilinear resampling at level 4)



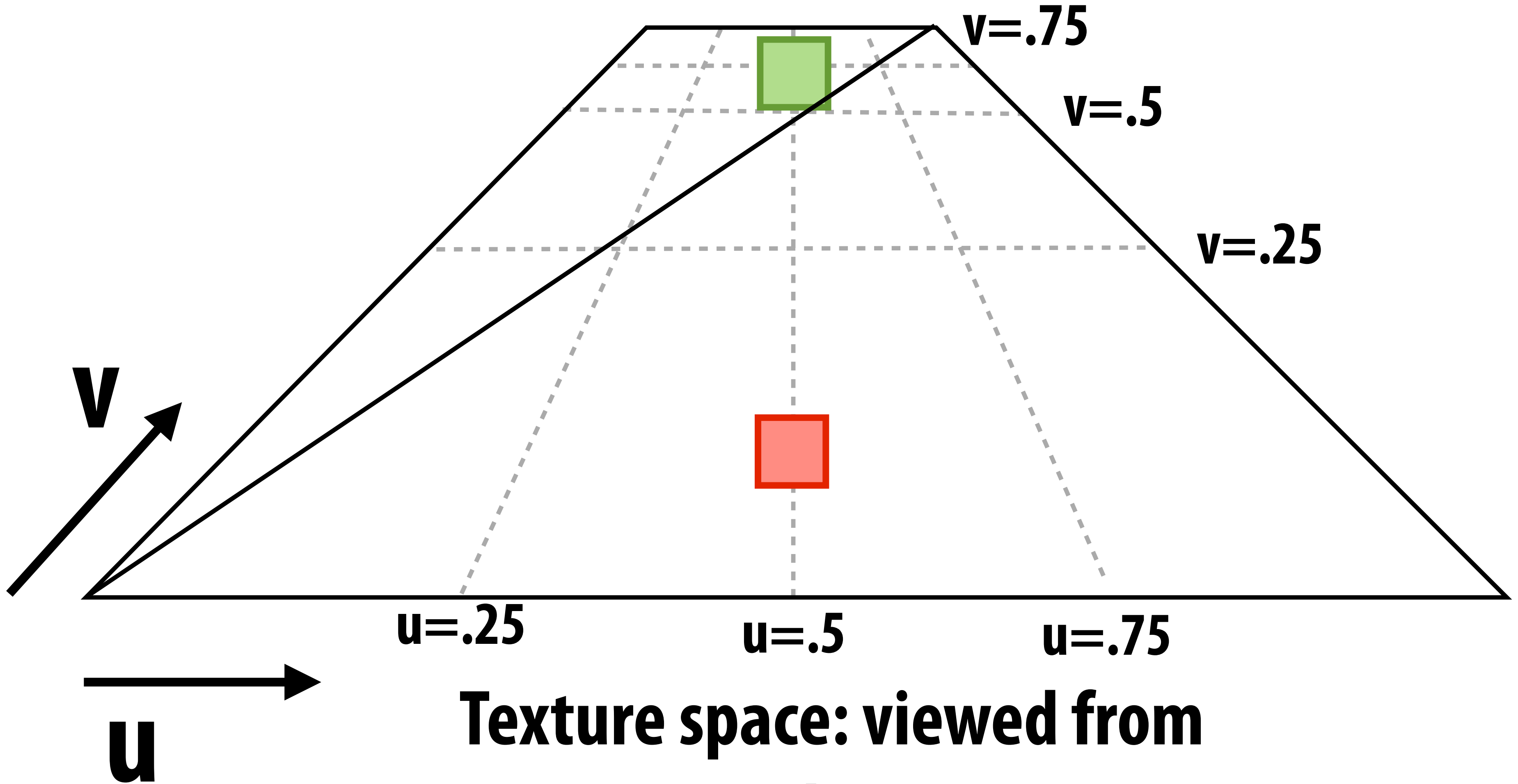
Mip-map level visualization

(trilinear filtering: visualization of continuous *d*)

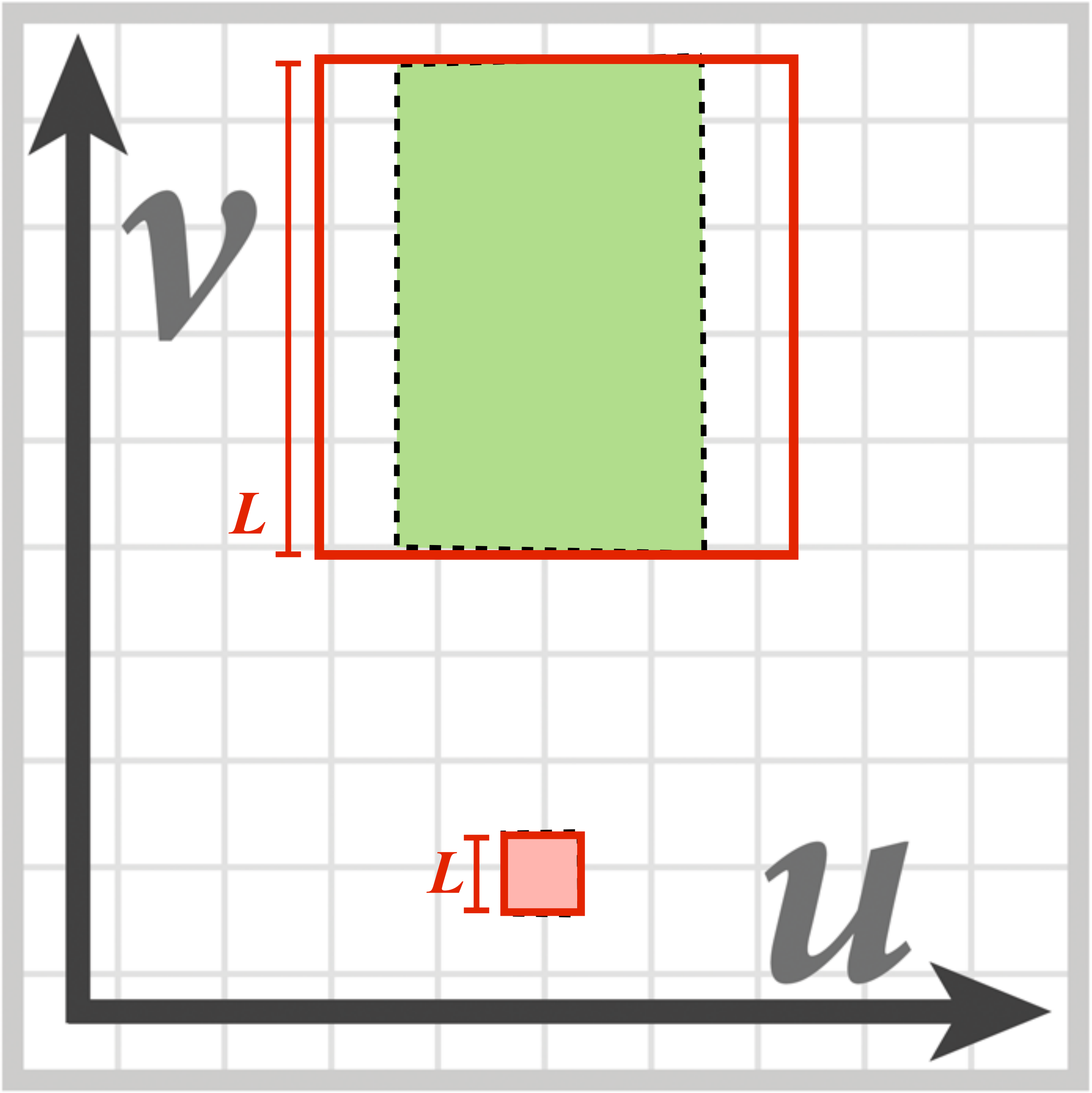


Pixel area may not map to isotropic region in texture space

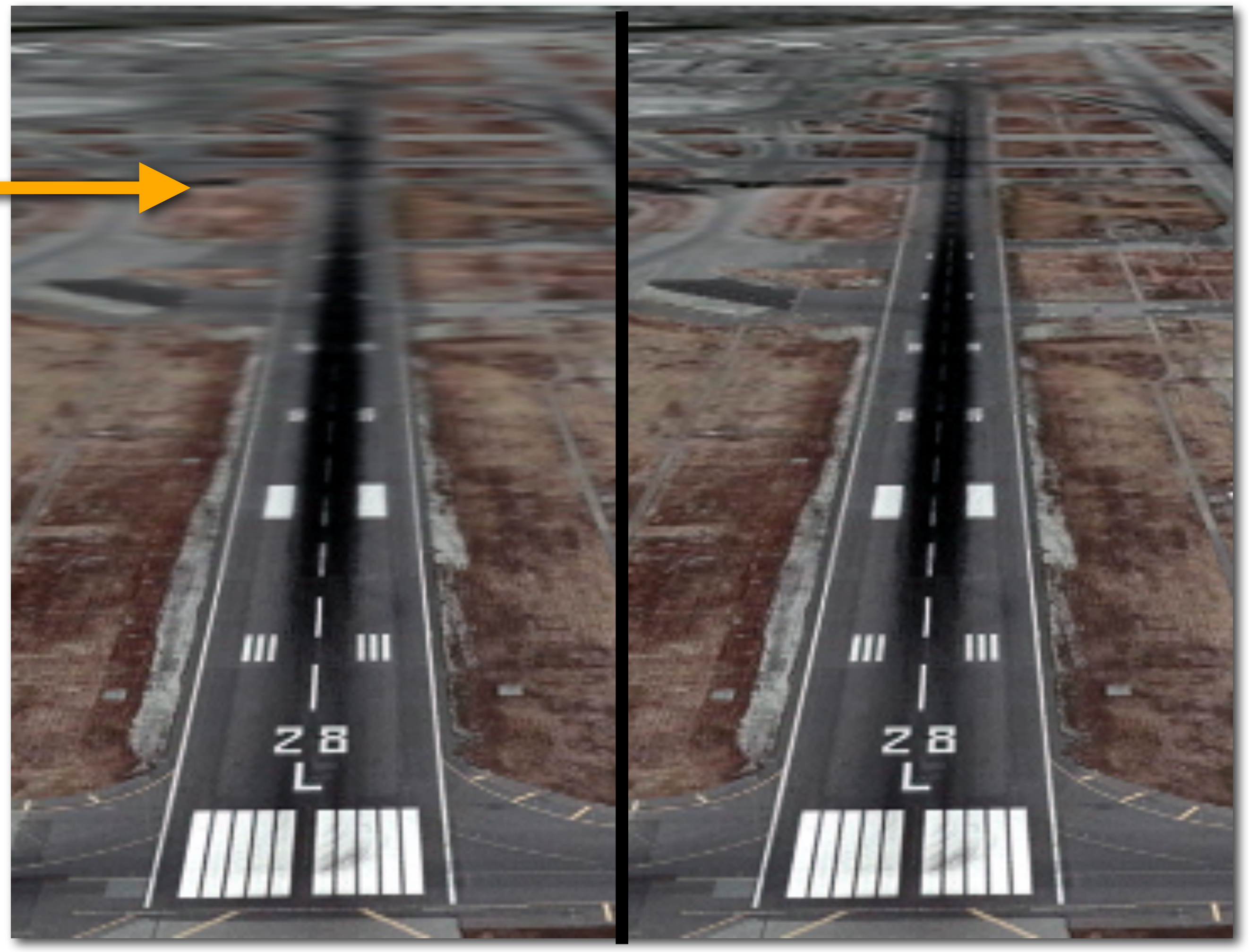
Proper filtering requires anisotropic (in texture space) filter footprint



Texture space: viewed from camera with perspective



Overblurring in *u* direction



Trilinear (Isotropic) Filtering

Anisotropic Filtering

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

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

GPUs shade at the granularity of 2x2 fragments

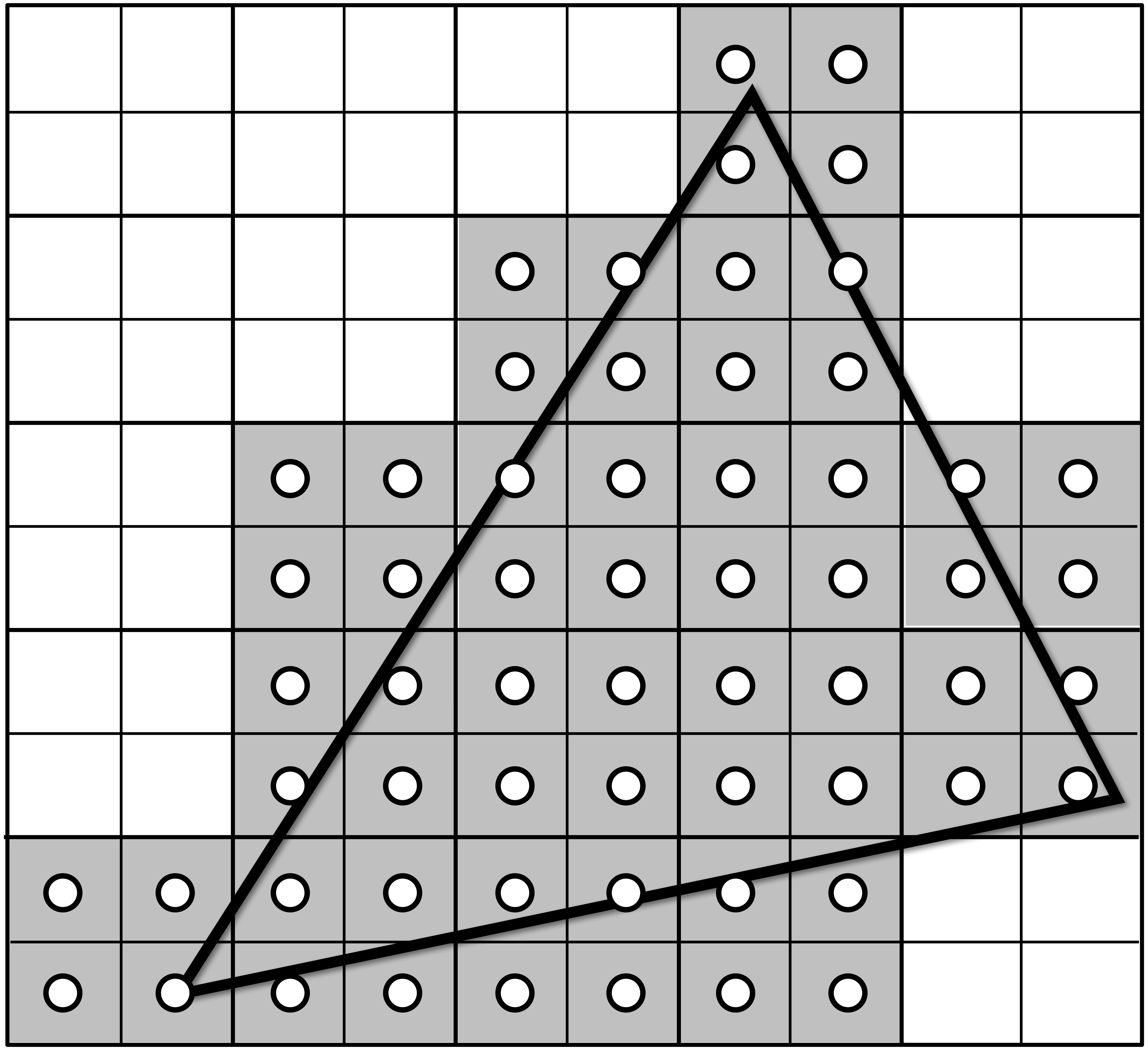
("quad fragment" is the minimum granularity of rasterization output and shading)

Enables cheap computation of texture coordinate differentials

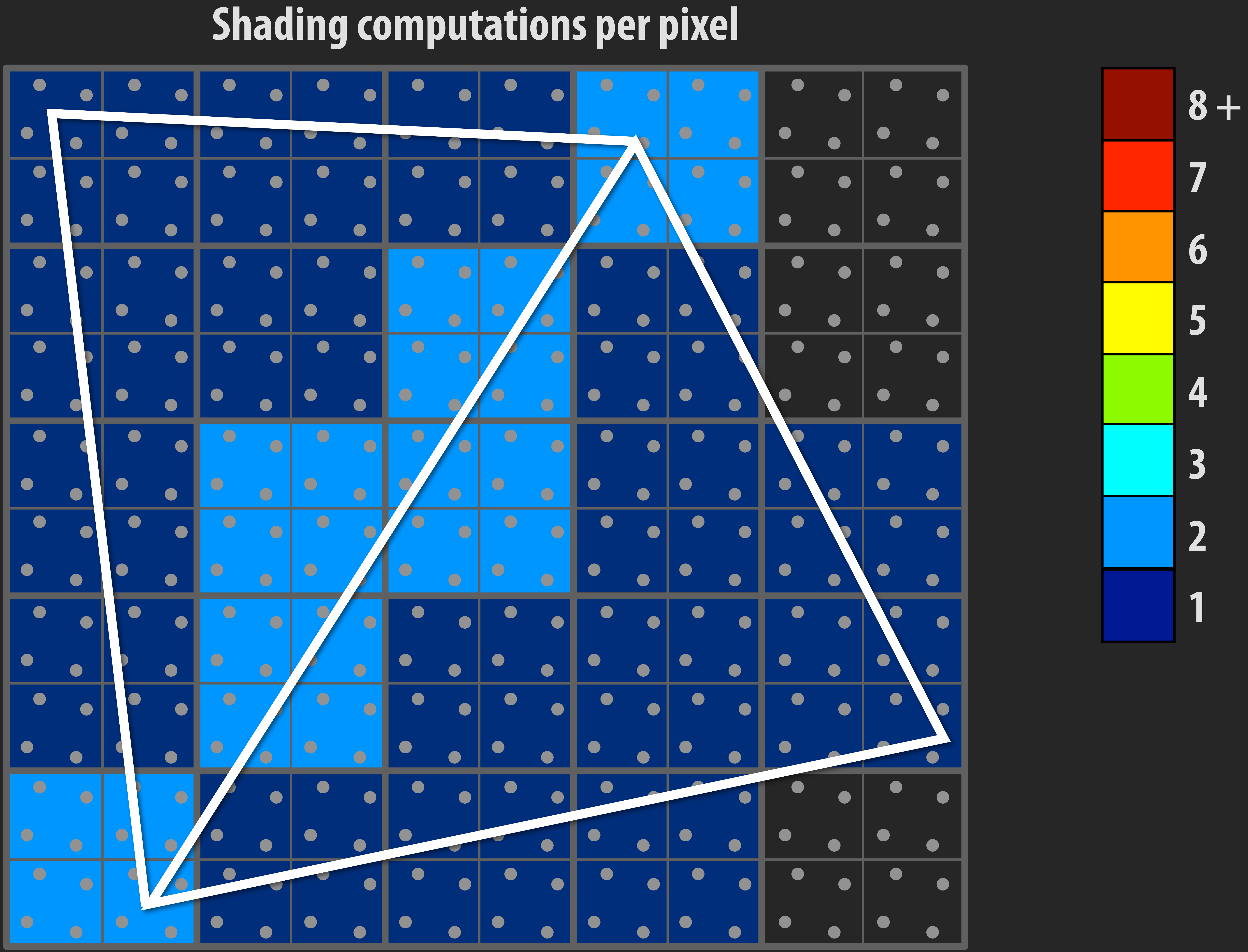
(cheap: derivative computation leverages shading work that must be done by adjacent fragment anyway)

All quad-fragments are shaded independently

(communication is between fragments in a quad fragment, no communication required between quad fragments)



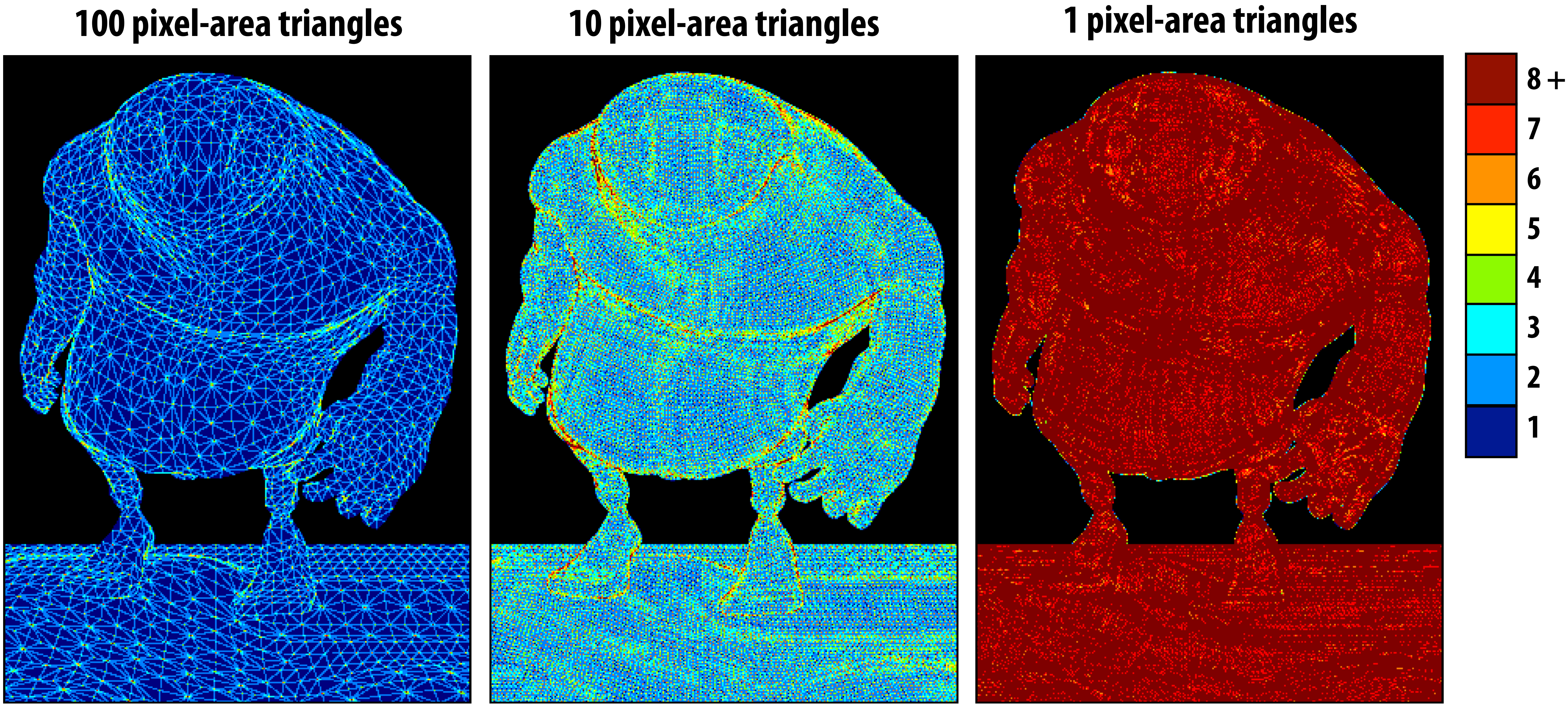
Implication: multiple fragments get shaded for pixels near triangle boundaries



Small triangles result in extra shading

Shaded quad fragments per pixel

(early-z is enabled + scene rendered in approximate front-to-back order to minimize extra shading due to overdraw)



100 pixel-area triangles

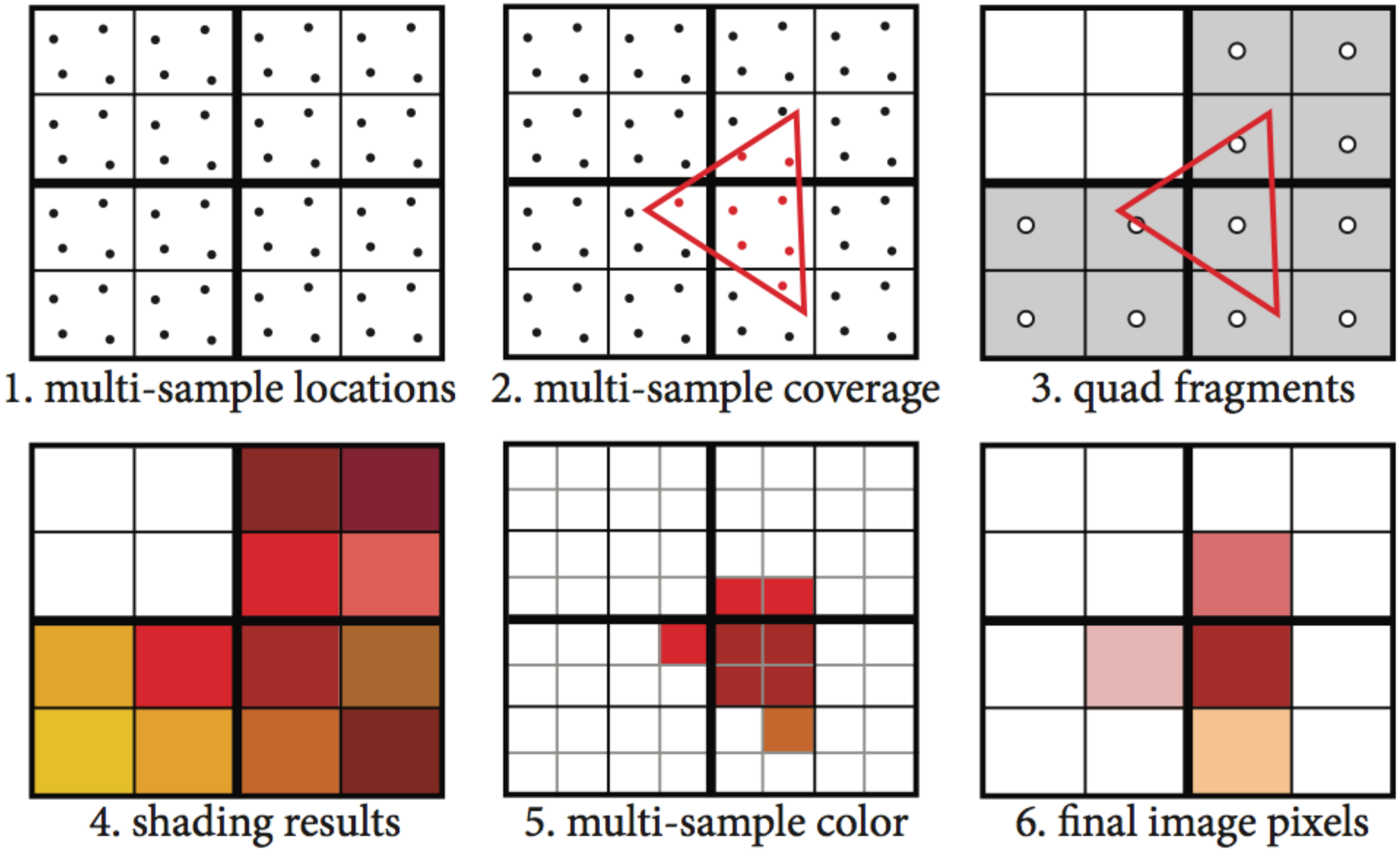
10 pixel-area triangles

1 pixel-area triangles

Want to sample appearance approximately once per surface per pixel (assuming correct texture filtering)

But graphics pipeline generates at least one appearance sample per triangle per pixel (actually more, considering quad fragments)

Multi-sample anti-aliasing (MSAA)



Main idea: decouple shading sampling rate from visibility sampling rate

- Depth buffer: stores depth per sample
- Color buffer: stores color per sample
- Resample color buffer to get final image pixel values (need one sample per display pixel)

Principle of texture thrift

[Peachey 90]

Given a scene consisting of textured 3D surfaces, the amount of texture information minimally required to render an image of the scene is proportional to the resolution of the image and is independent of the number of surfaces and the size of the textures.

Summary: texture filtering using the mip map

- **Small storage overhead (33%)**
  - Mipmap is 4/3 the size of original texture image

- **For each isotropically-filtered sampling operation**
  - Constant filtering cost (independent of *d*)
  - Constant number of texels accessed (independent of *d*)

- **Combat aliasing with prefiltering, rather than supersampling**
  - Recall: we used supersampling to address aliasing problem when sampling coverage

- **Bilinear/trilinear filtering is isotropic and thus will "overblur" to avoid aliasing**
  - Anisotropic texture filtering provides higher image quality at higher compute and memory bandwidth cost (use more texture samples to better approximate non-square footprint in texture space)

Summary: a 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 quad-fragment samples
3. Compute *d*
4. Convert normalized texture coordinate (u,v) to texture coordinates texel_u, texel_v
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 (texel_u, texel_v, d)

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

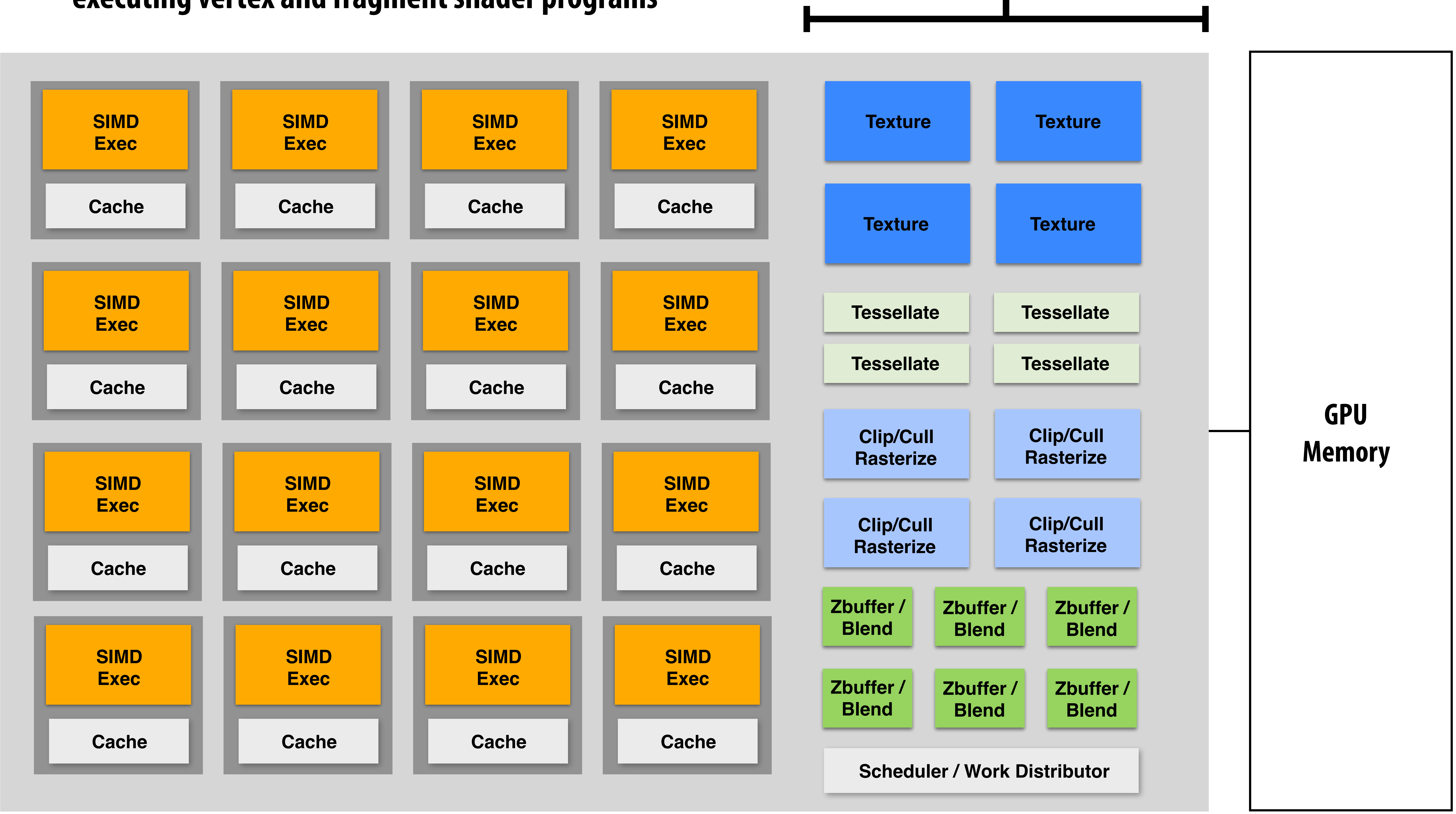
All modern GPUs have dedicated fixed-function hardware support for performing texture sampling operations.

** May involve wrap, clamp, etc. of texel coordinates according to sampling mode configuration

GPU: heterogeneous, multi-core processor

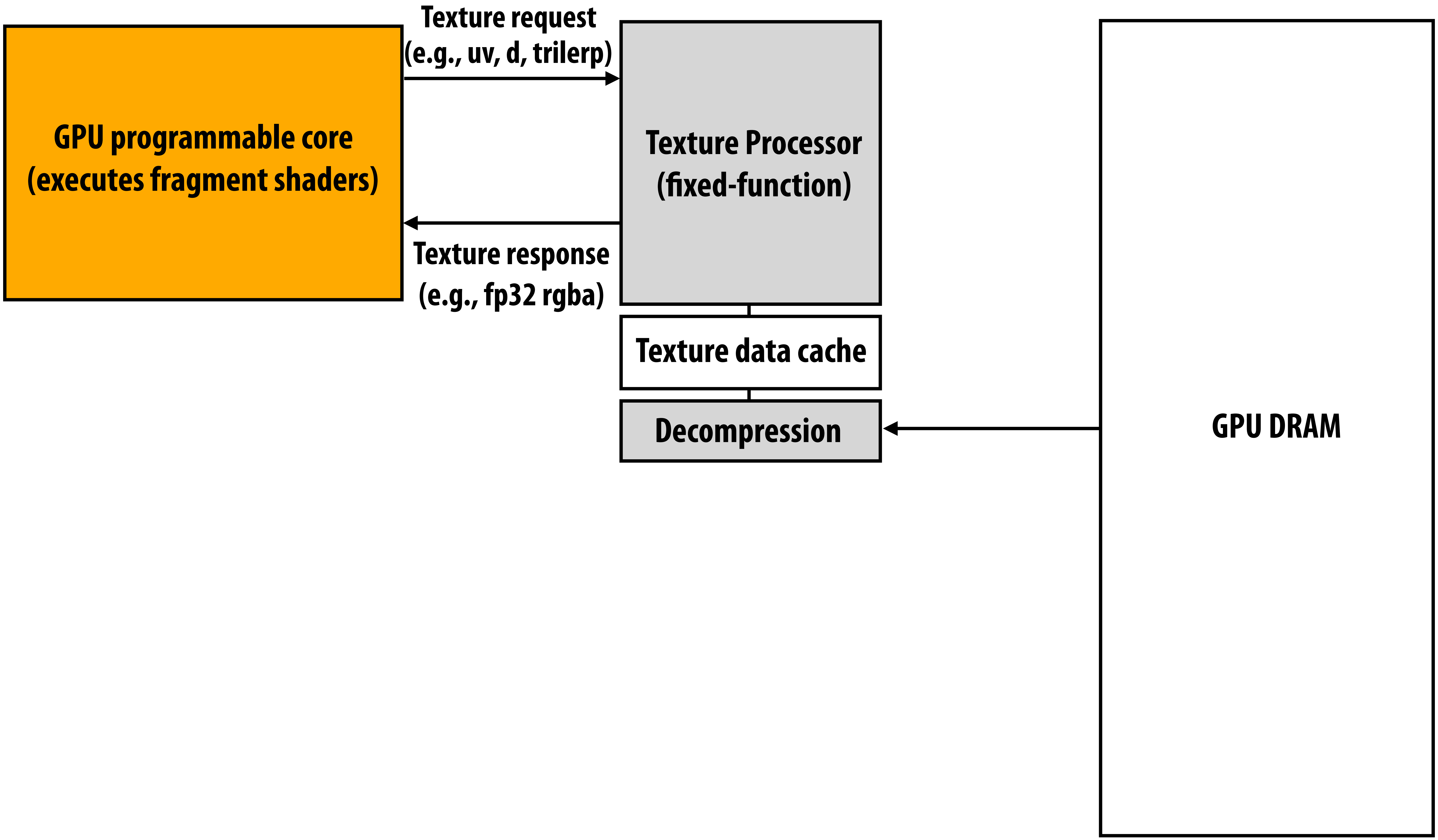
Modern GPUs offer ~ TFLOPs of performance for executing vertex and fragment shader programs

T-OP's of fixed-function compute capability over here



Texture caching

Texture system block diagram



Consider memory implications of texturing

- **Texture data footprint**
  - Modern game scenes = many large textures
    - GBs of texture data in a scene (uncompressed 2K x 2K RGB is 12MB)
  - Film rendering: GBs to TBs of textures in scene DB

- **Texture bandwidth**
  - 8 texels per tri-linear fetch
  - Modern GPU: billions of fragments/sec
    (NVIDIA GTX 1080: ~300 billion filtered texture values/sec)

- **A performant graphics system needs:**
  - High memory bandwidth
  - Texture caching
  - Texture data compression
  - Latency hiding solution to avoid stalls during texture data access

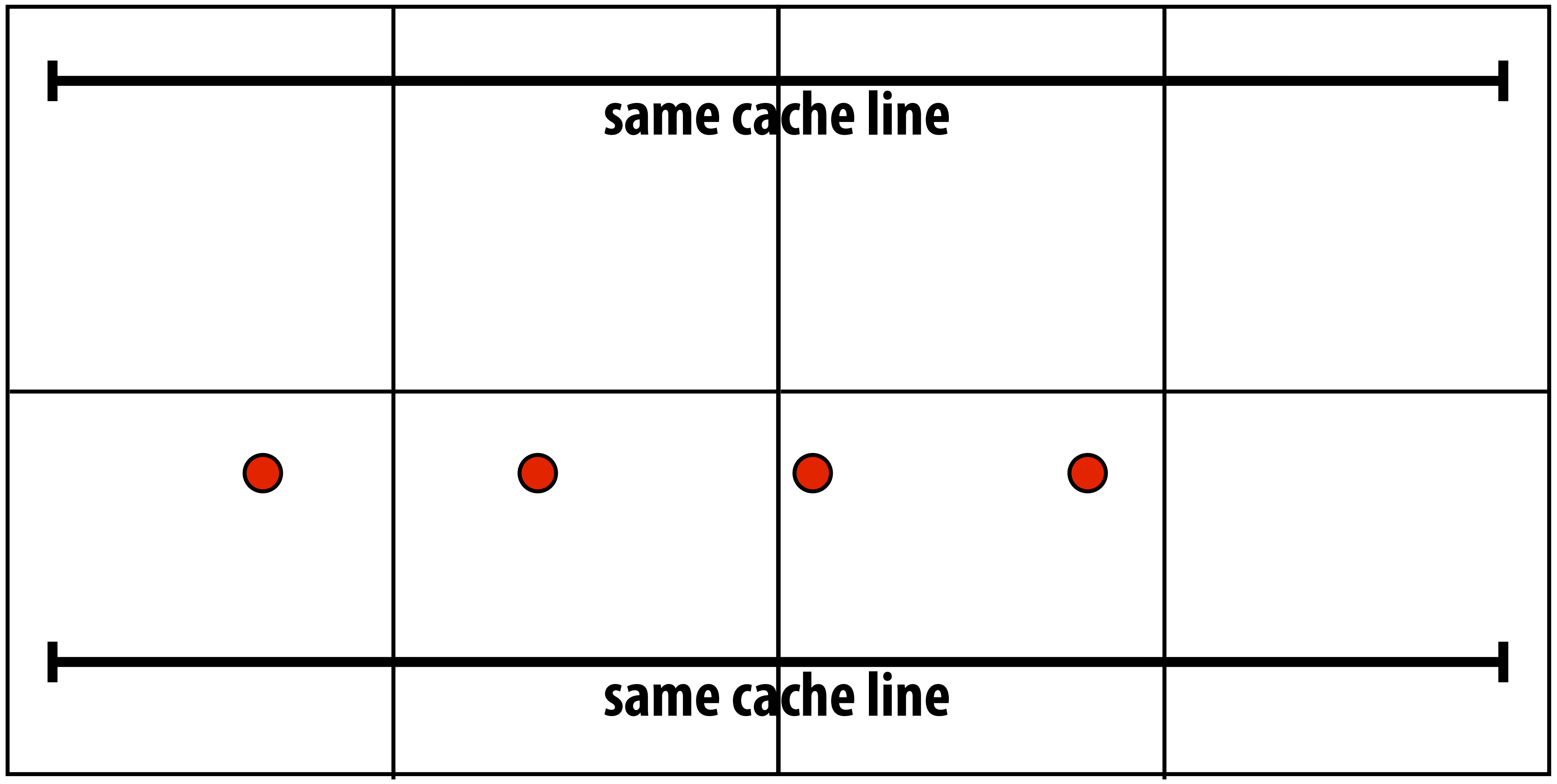
Review: the role of caches in CPUs

- Reduce latency of data access

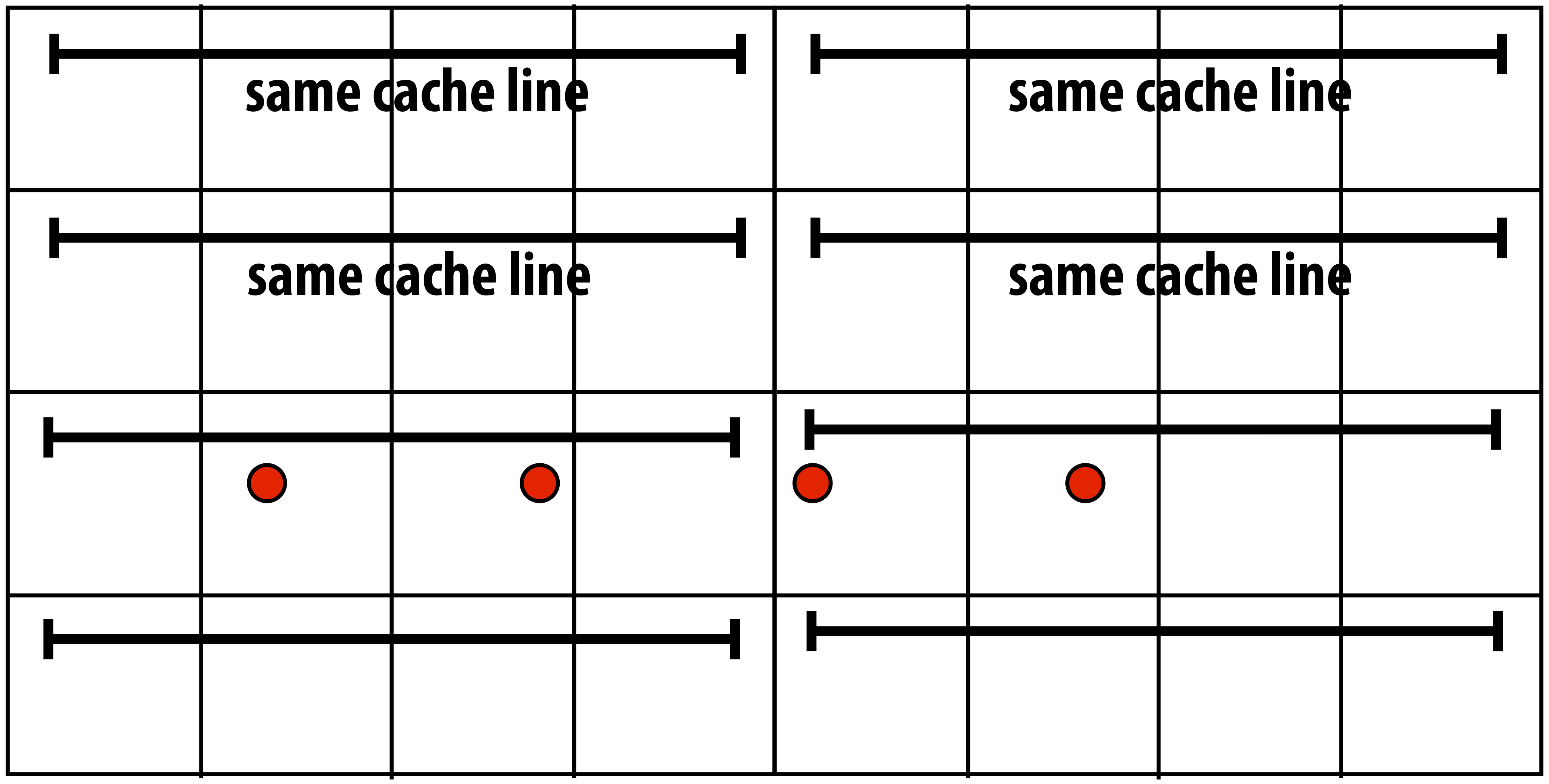
- Reduce off-chip bandwidth requirements (caches service requests that would require DRAM access)
  - Note: alternatively, you can think about caches as <u>bandwidth amplifiers</u> (data path between cache and ALUs is usually wider than that to DRAM)

- Convert fine-grained (word-sized) memory requests from processors into large (cache-line sized) requests than can be serviced efficiently by wide memory bus and DRAM

Texture caching thought experiment



mip-map: level *d+1* texels



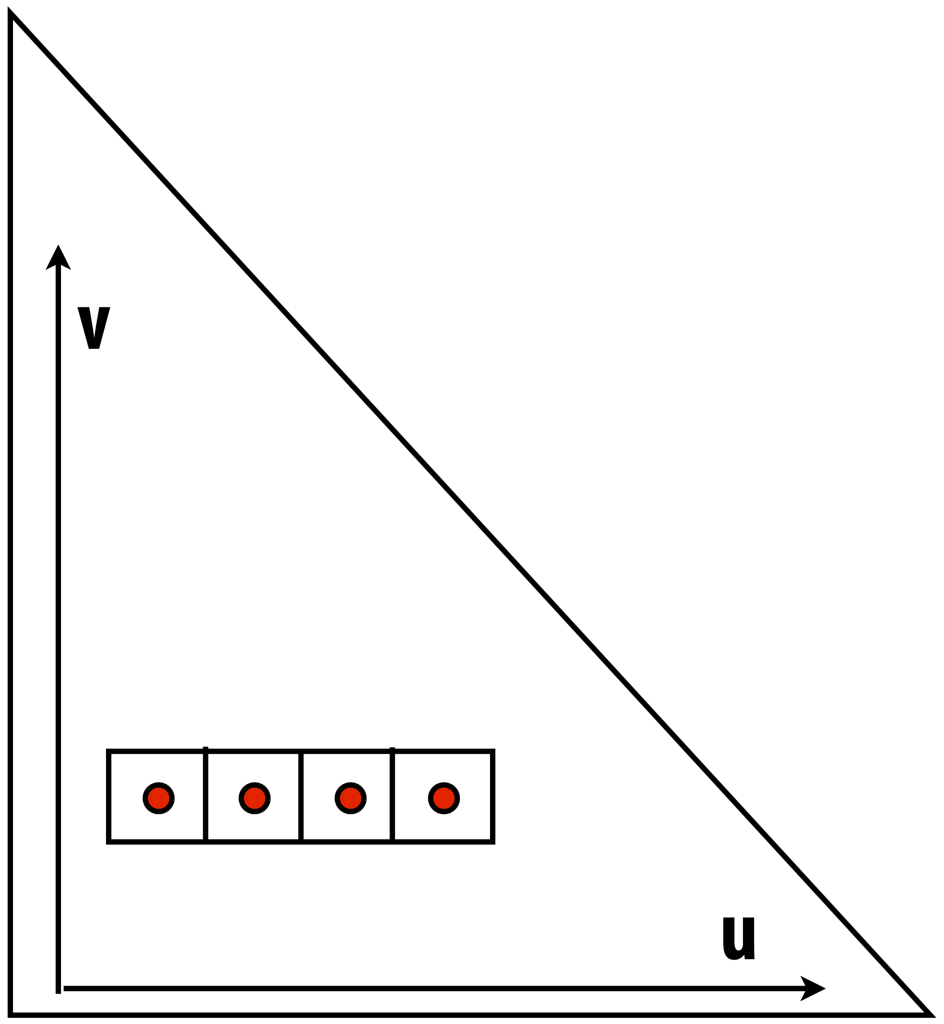
mip-map: level *d* texels

Assume:

Row-major rasterization order

Horizontal texels contiguous in memory

Texture cache line = 4 texels



What type of data reuse does a texture cache designed to capture?

- **Spatial locality across fragments, not temporal locality within a fragment!**
  - The same texels are required to filter texture fetches from adjacent fragments (due to overlap of filter support regions)
  - Little-to-no temporal locality within a fragment shader (little reason for a shader to access the same part of the texture map twice)

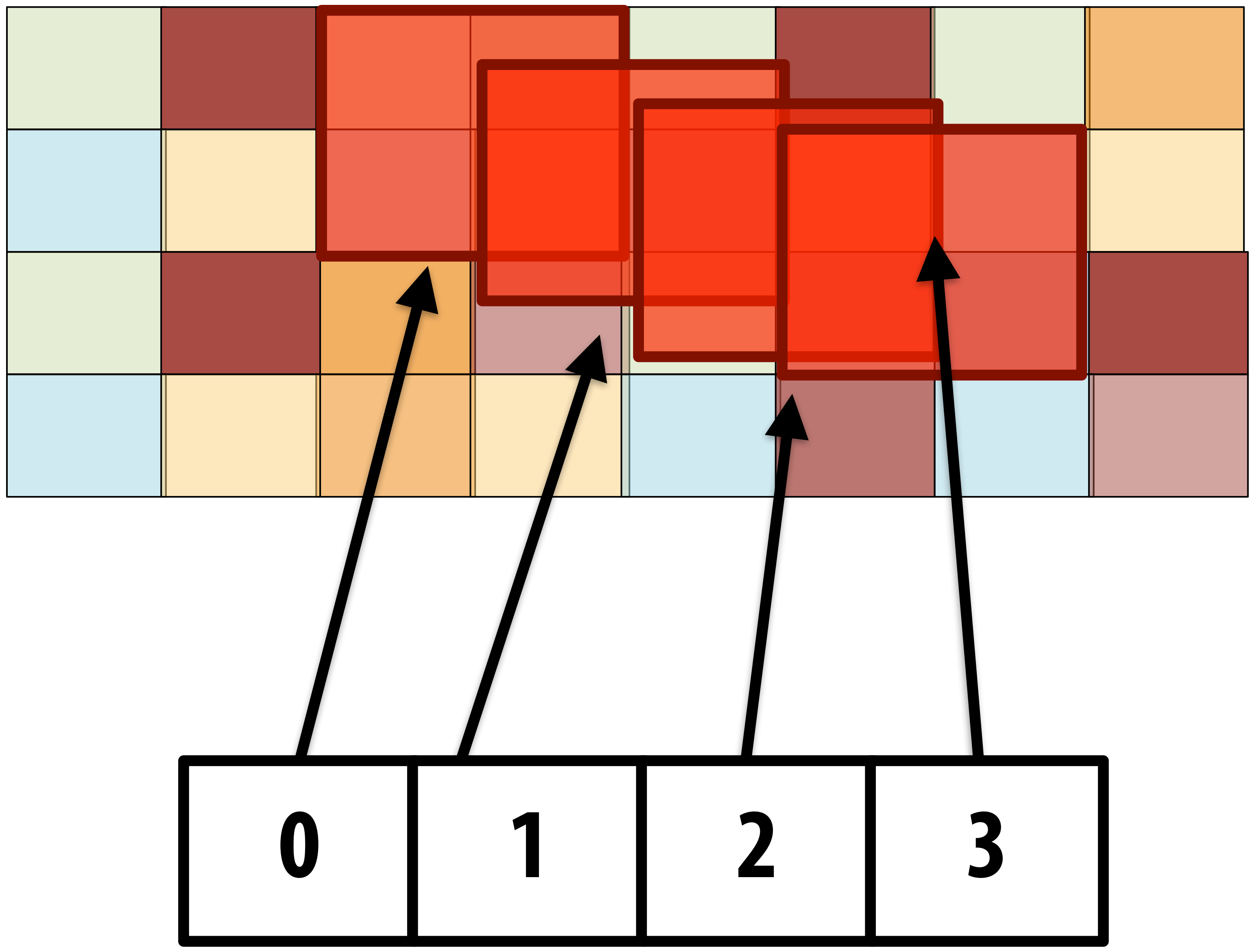
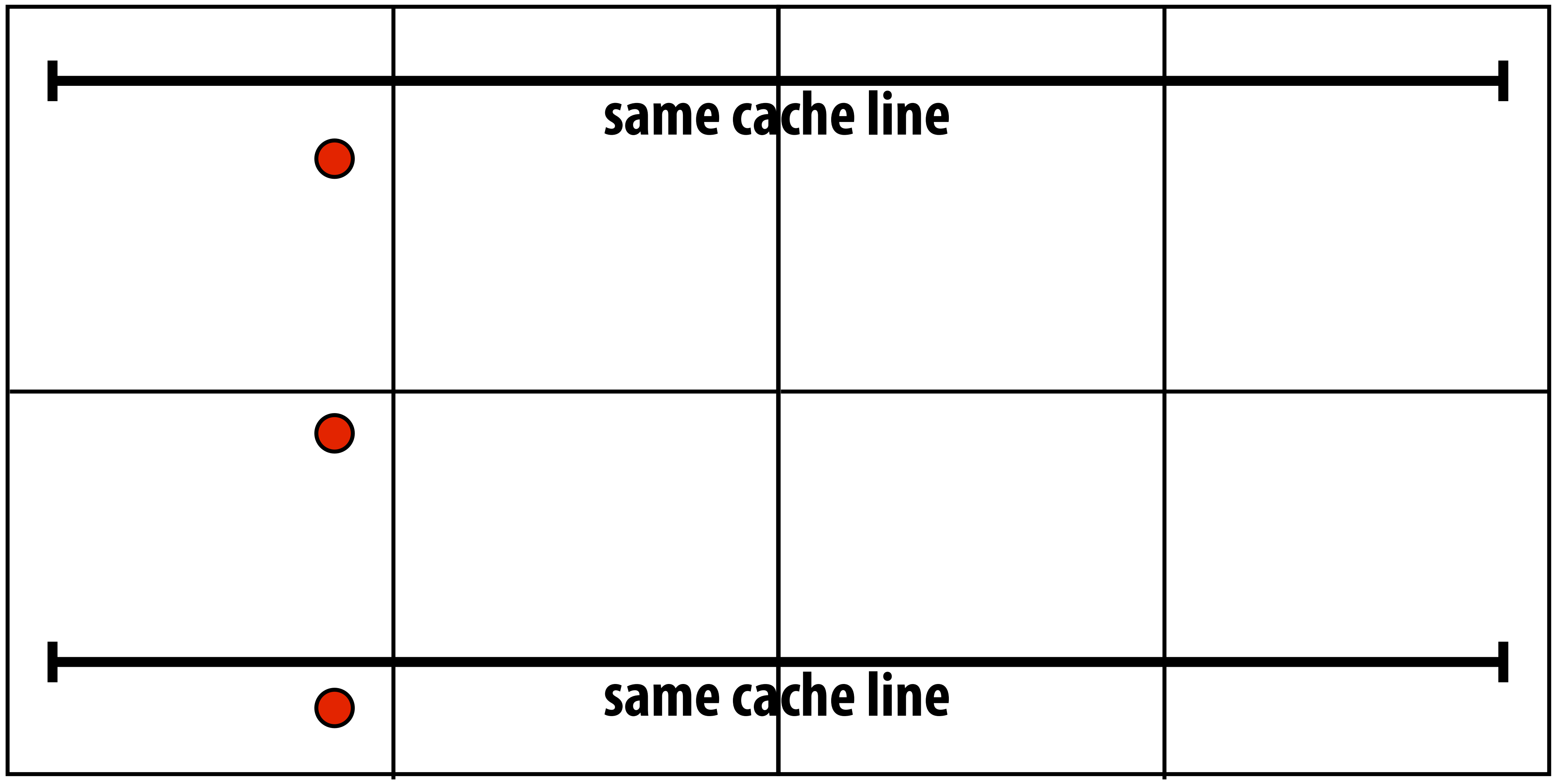
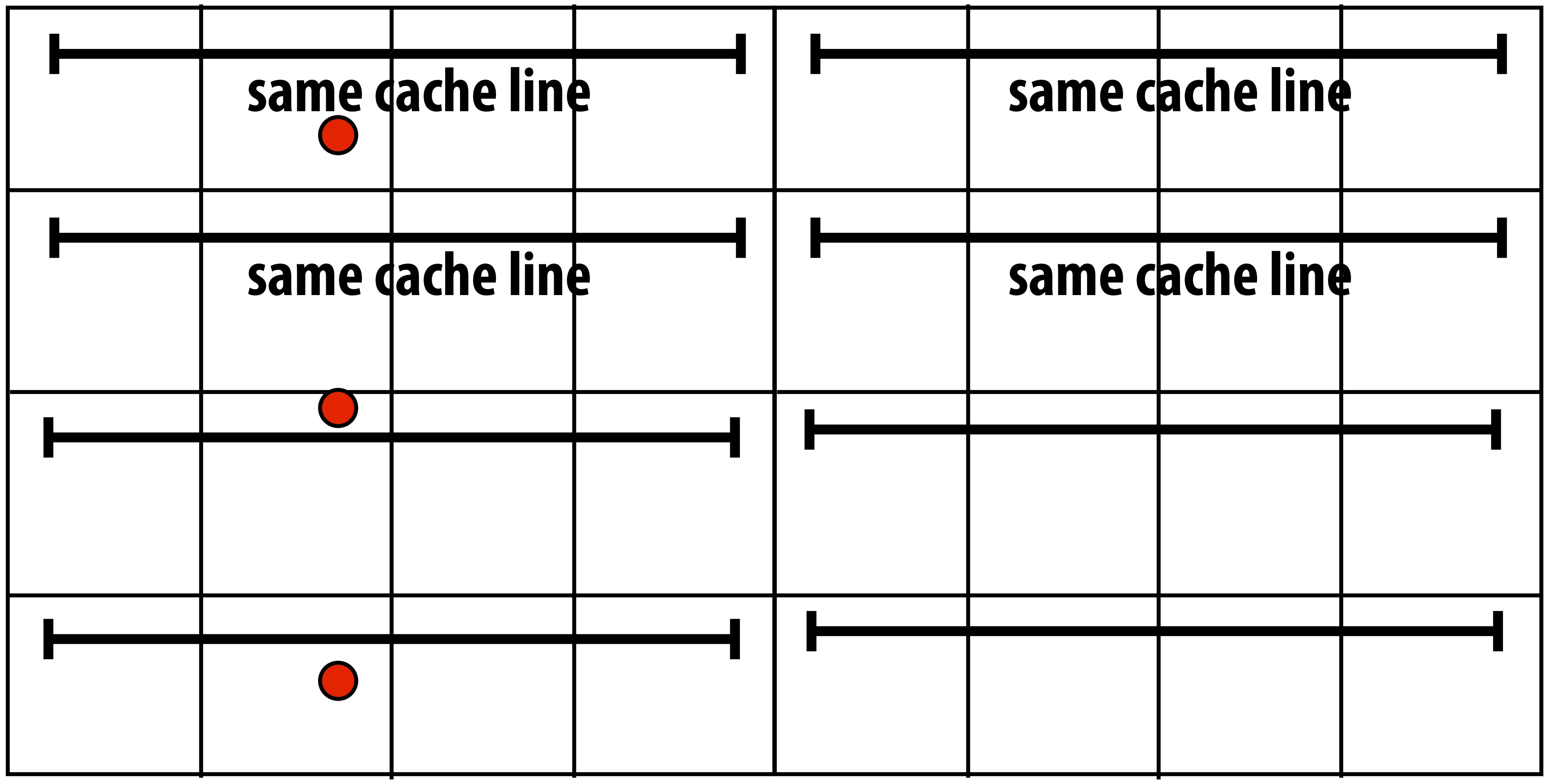


Figure illustrates filter support regions from texture fetches from four adjacent fragments

Now rotate triangle on screen



mip-map: level $d+1$ texels



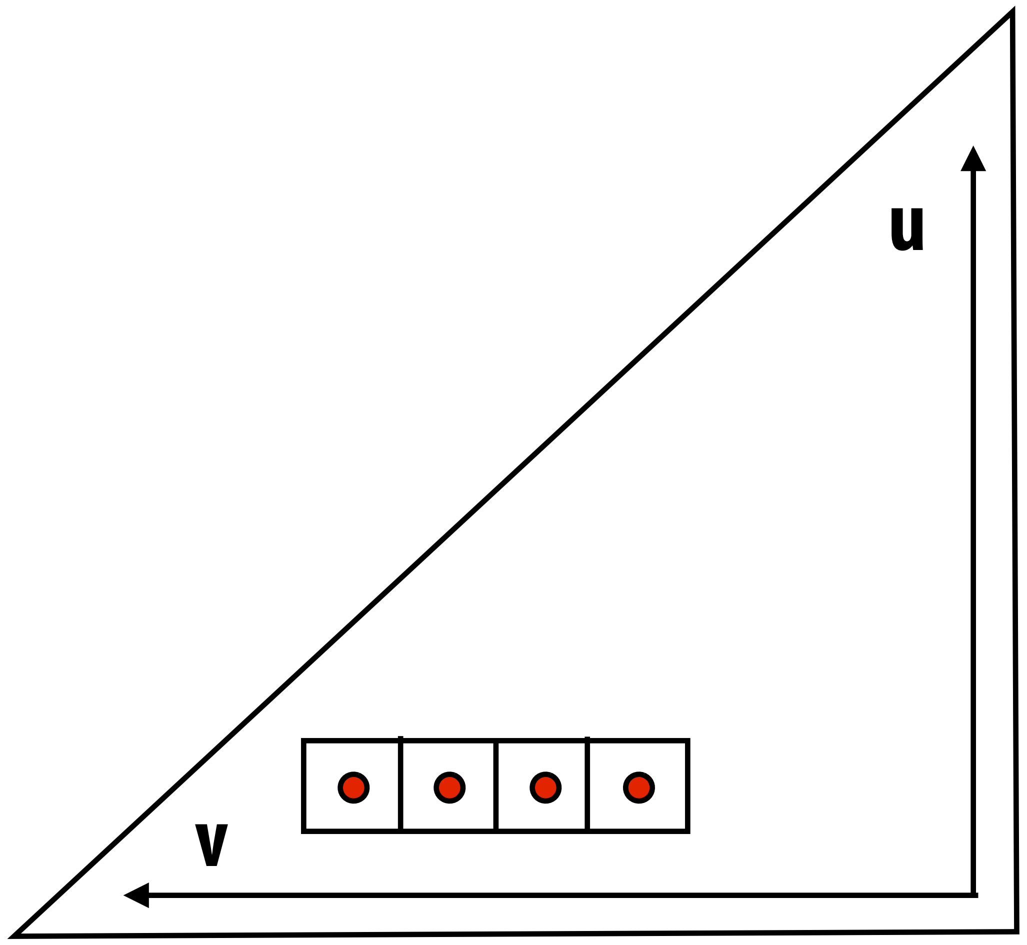
mip-map: level d texels

Assume:

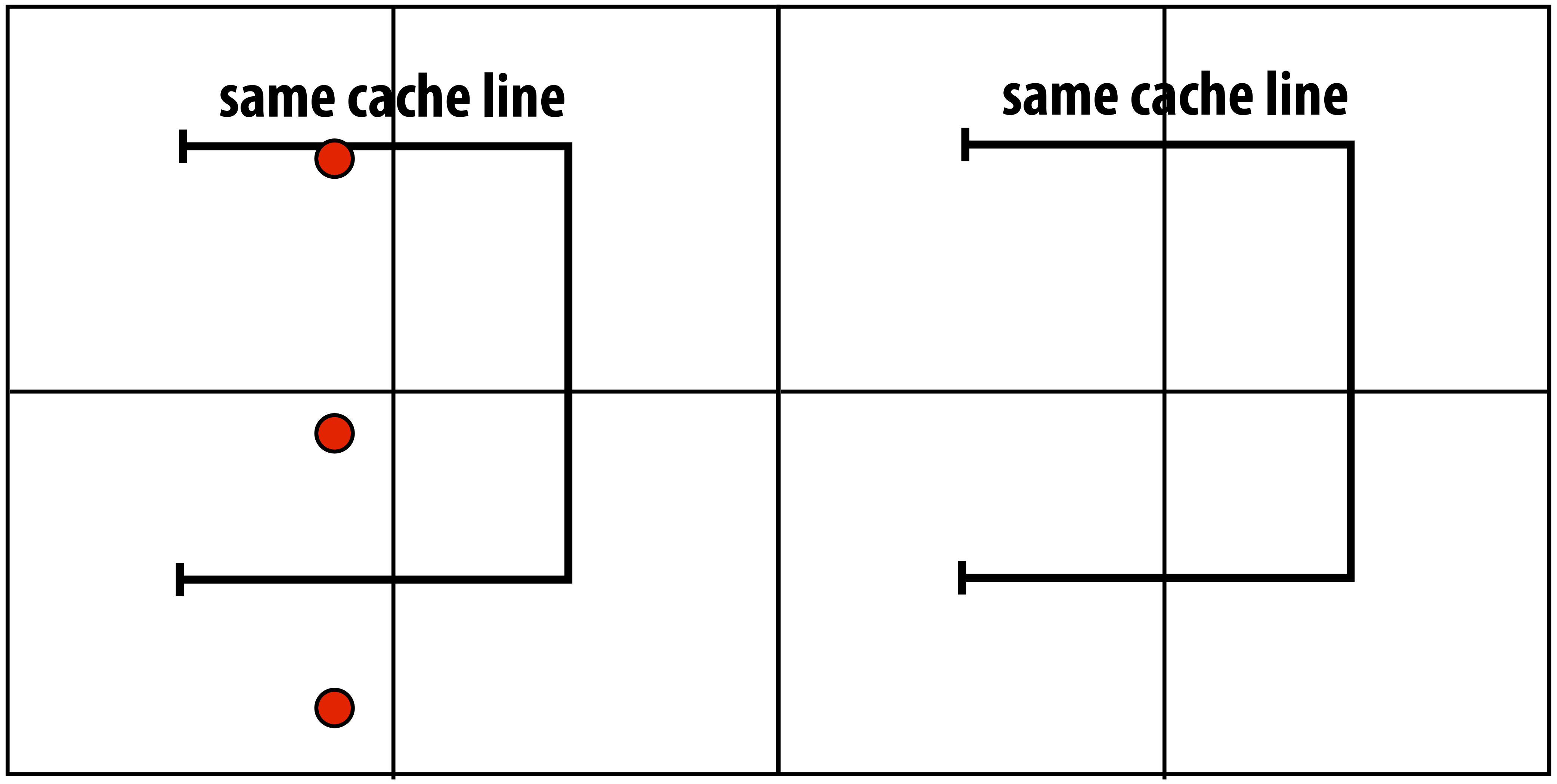
Row-major rasterization order

Horizontal texels contiguous in memory

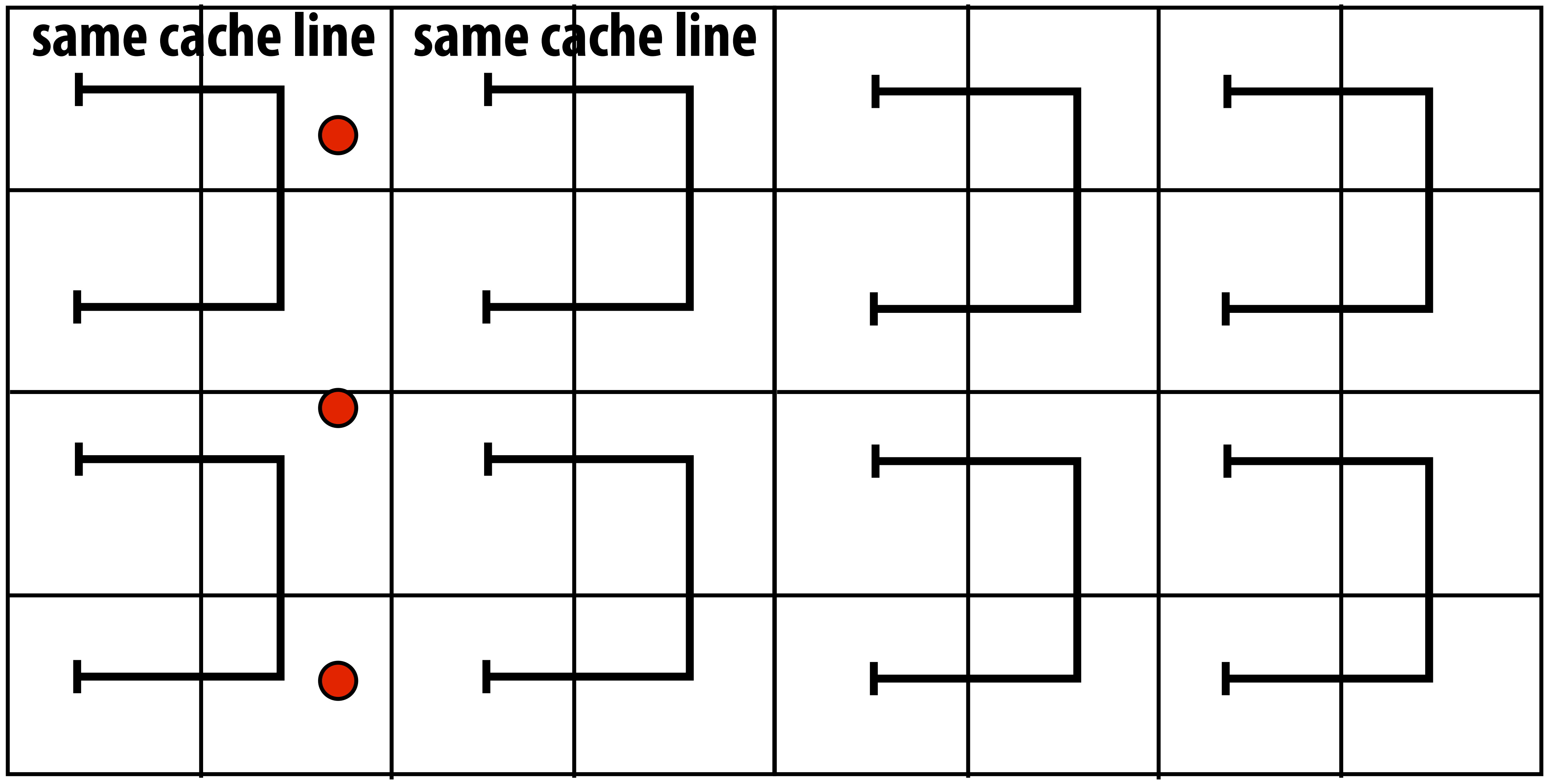
Cache line = 4 texels



4D blocking (texture is 2D array of 2D blocks: robust to triangle orientation)



mip-map: level *d+1* texels



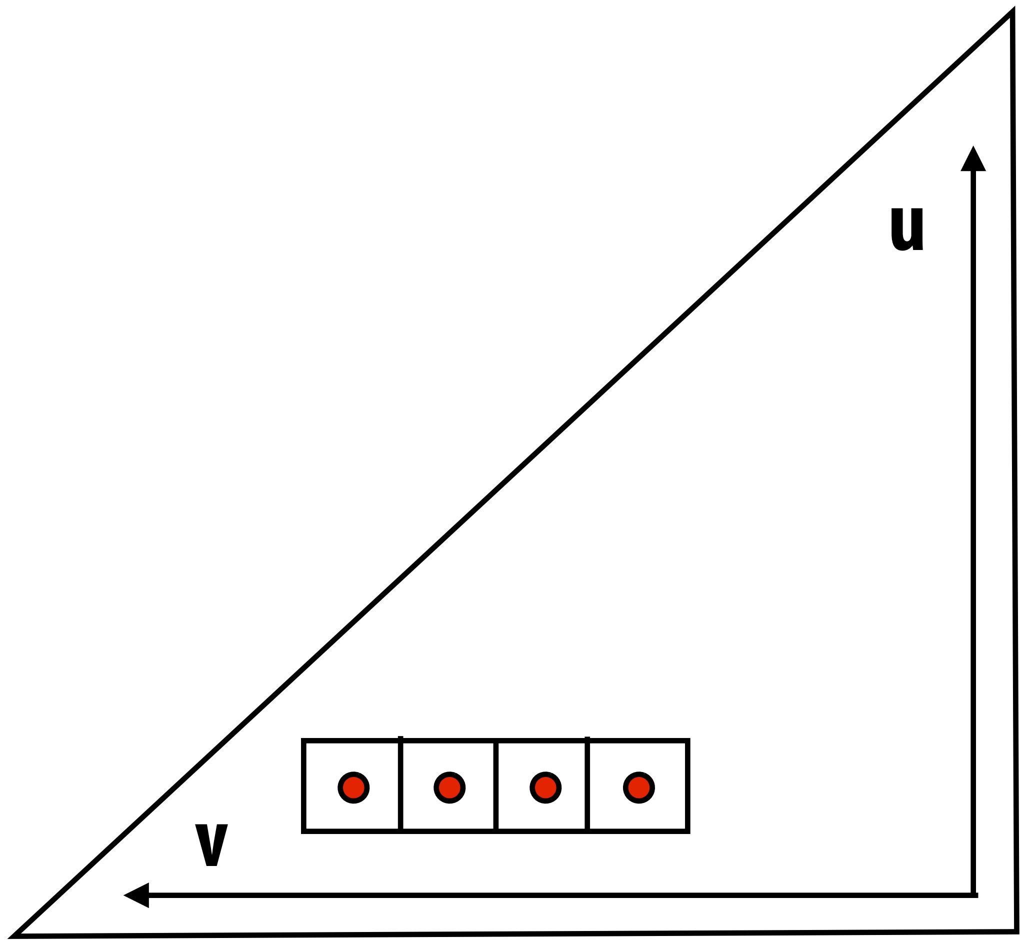
mip-map: level *d* texels

Assume:

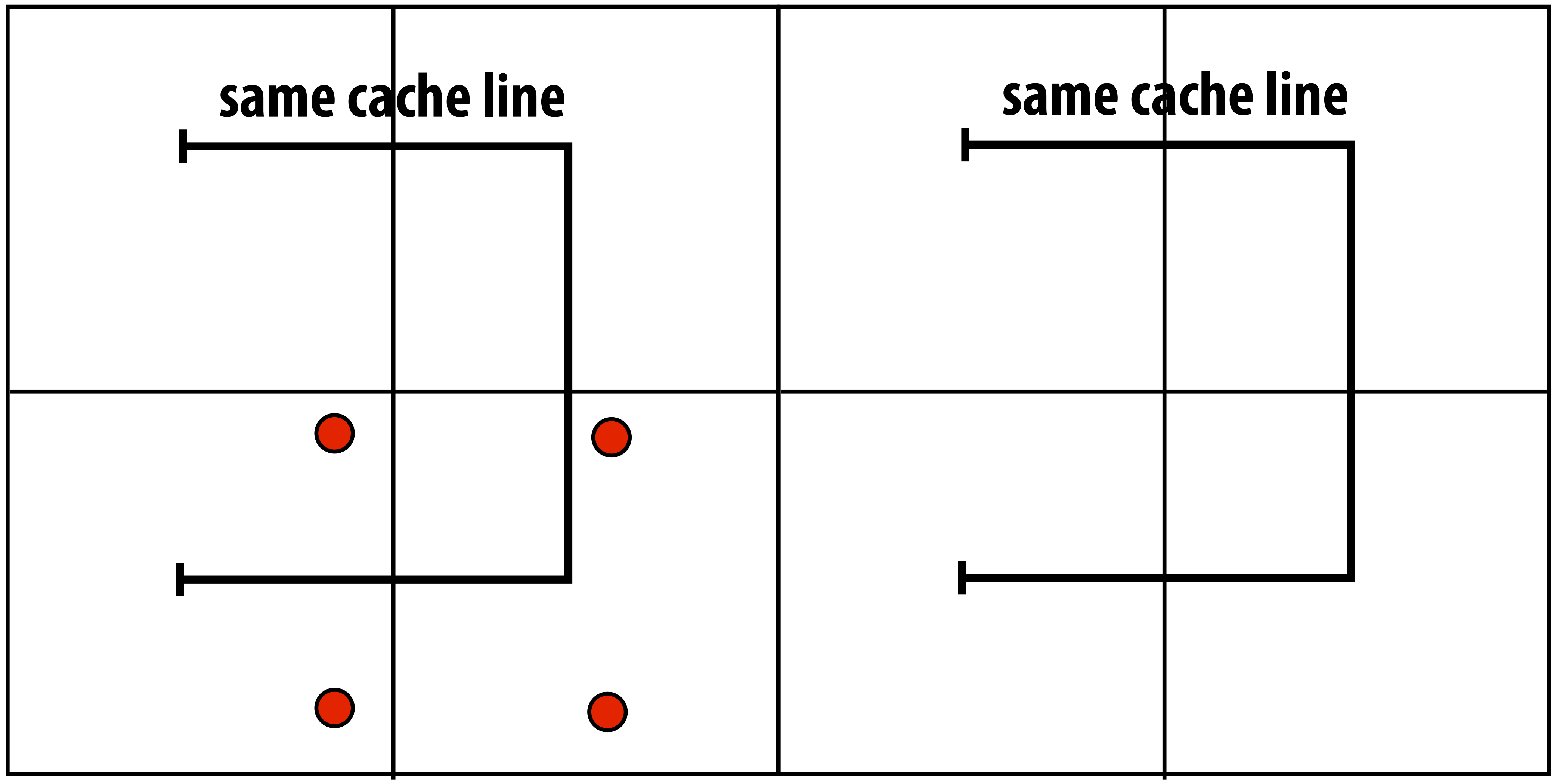
Row-major rasterization order

2D blocks of texels contiguous in memory

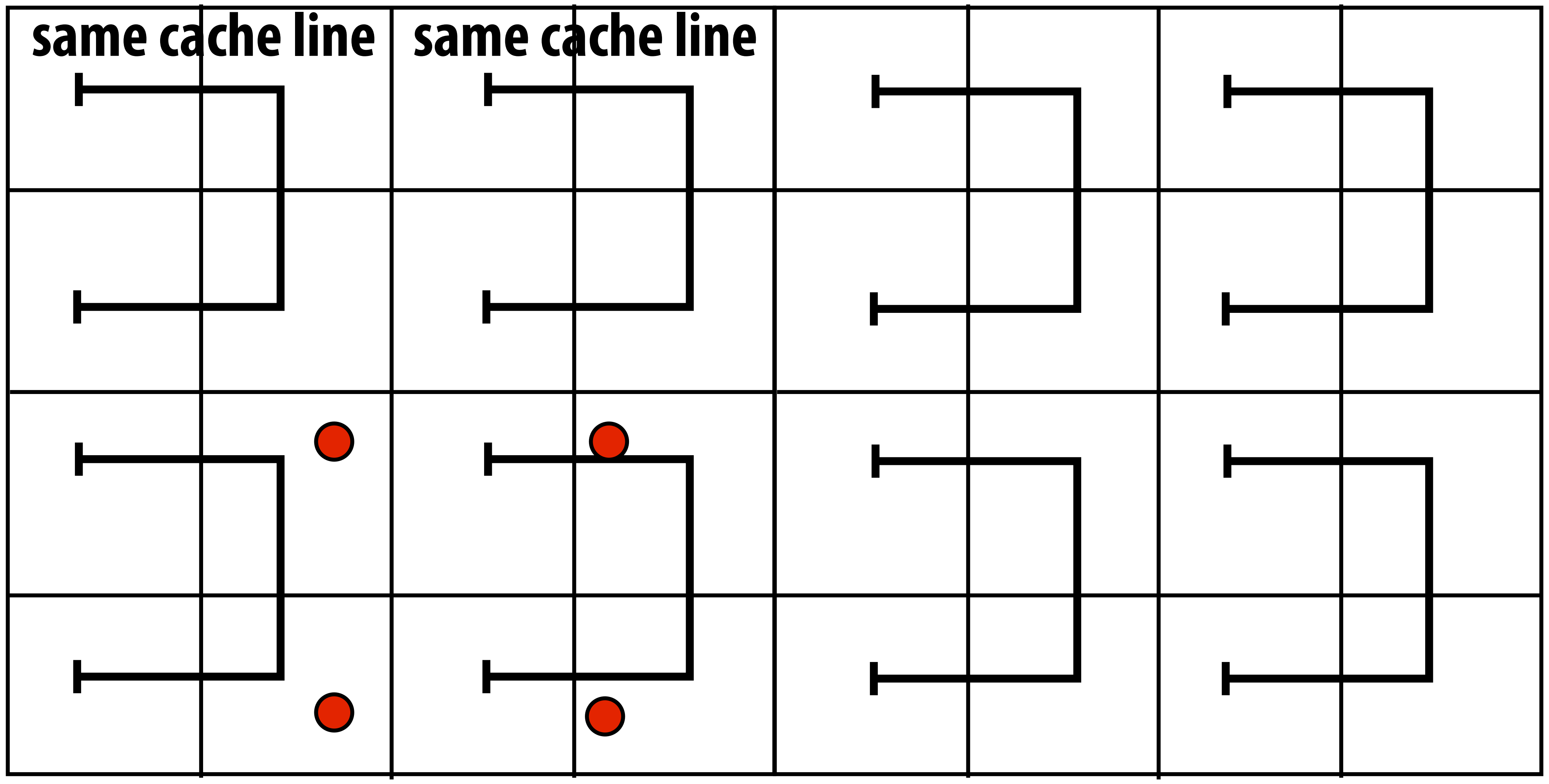
Cache line = 4 texels



Tiled rasterization increases reuse



mip-map: level $d+1$ texels



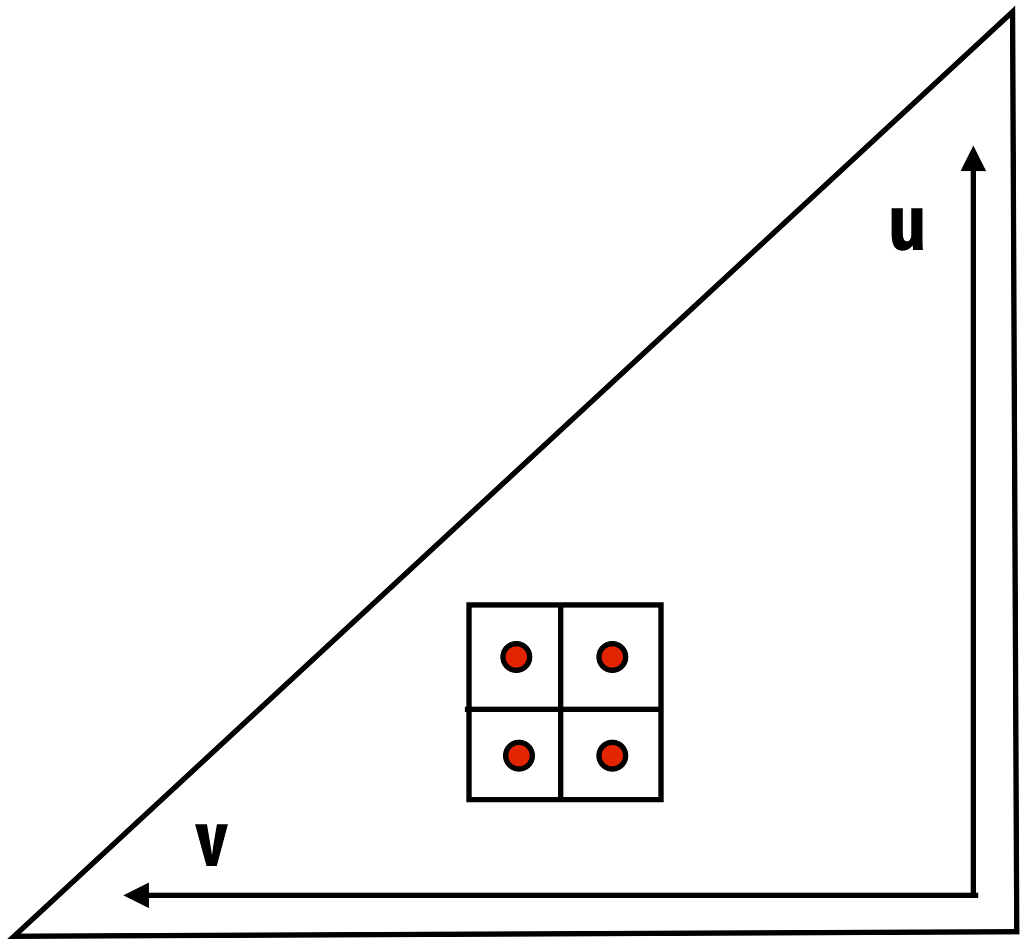
mip-map: level d texels

Assume:

Blocked rasterization order!

2D blocks of texels contiguous in memory

Cache line = 4 texels



Key metric: unique texel-to-fragment ratio

- **Unique texel-to-fragment ratio**
  - Number of unique texels accessed when rendering a scene divided by number of fragments processed [see Igeny reading for stats: can be less than < 1]
  - What is the worst case ratio assuming trilinear filtering?
  - How can inaccurate computation of texture mip level (*d*) affect this?

- **In reality, texture caching behavior is good, but not CPU workload good**
  - [Montrym & Moreton 95] design for 90% hits
  - Only so much spatial locality to exploit (no high temporal locality like CPU workloads)

Texture data access characteristics

- Key metric: unique texel-to-fragment ratio
  - Number of unique texels accessed when rendering a scene divided by number of fragments processed [see Igeny reading for stats: often less than < 1]
  - What is the worst-case ratio? (assuming trilinear filtering)
  - How can incorrect computation of texture miplevel (*d*) affect this?

- In practice, caching behavior is good, but not CPU workload good
  - [Montrym & Moreton 95] design for 90% hits
  - Why? (only so much spatial locality)

- Implications
  - GPU must provide high memory bandwidth for texture data access
  - GPU must have solution for hiding memory access latency
  - GPU must reduce its bandwidth requirements using caching and texture compression

Texture compression

A 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 quad-fragment samples
3. Compute *d*
4. Convert normalized texture coordinate (u,v) to texture coordinates texel_u, texel_v
5. Compute required texels in window of filter **
6. If texture data in filter footprint (eight texels for trilinear filtering) is not in cache:
    - Load required texels (in compressed form) from memory
    - Decompress texture data
7. Perform tri-linear interpolation according to (texel_u, texel_v, d)

** May involve wrap, clamp, etc. of texel coordinates according to sampling mode configuration

Texture compression

- **Goal: reduce bandwidth requirements of texture access**
- **Texture is read-only data**
    - Compression can be performed off-line, so compression algorithms can take significantly longer than decompression (decompression must be fast!)
    - Lossy compression schemes are permissible
- **Design requirements**
    - Support random texel access into texture map (constant time access to any texel)
    - High-performance decompression
    - Simple algorithms (low-cost hardware implementation)
    - High compression ratio
    - High visual quality (lossy is okay, but cannot lose too much!)

Simple scheme: color palette (indexed color)

- Lossless (if image contains a small number of unique colors)

Color palette (eight colors)

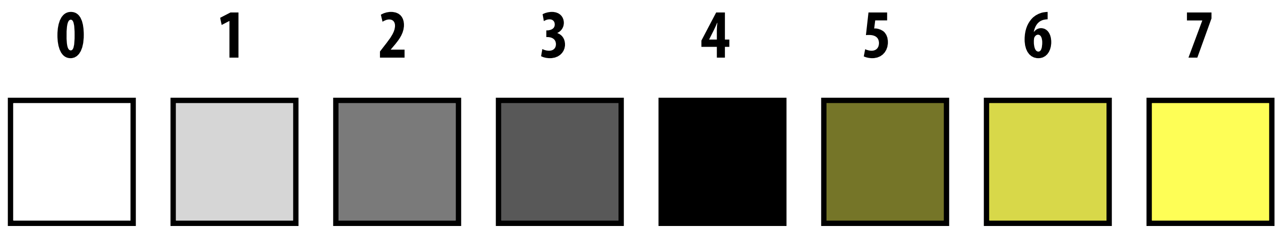
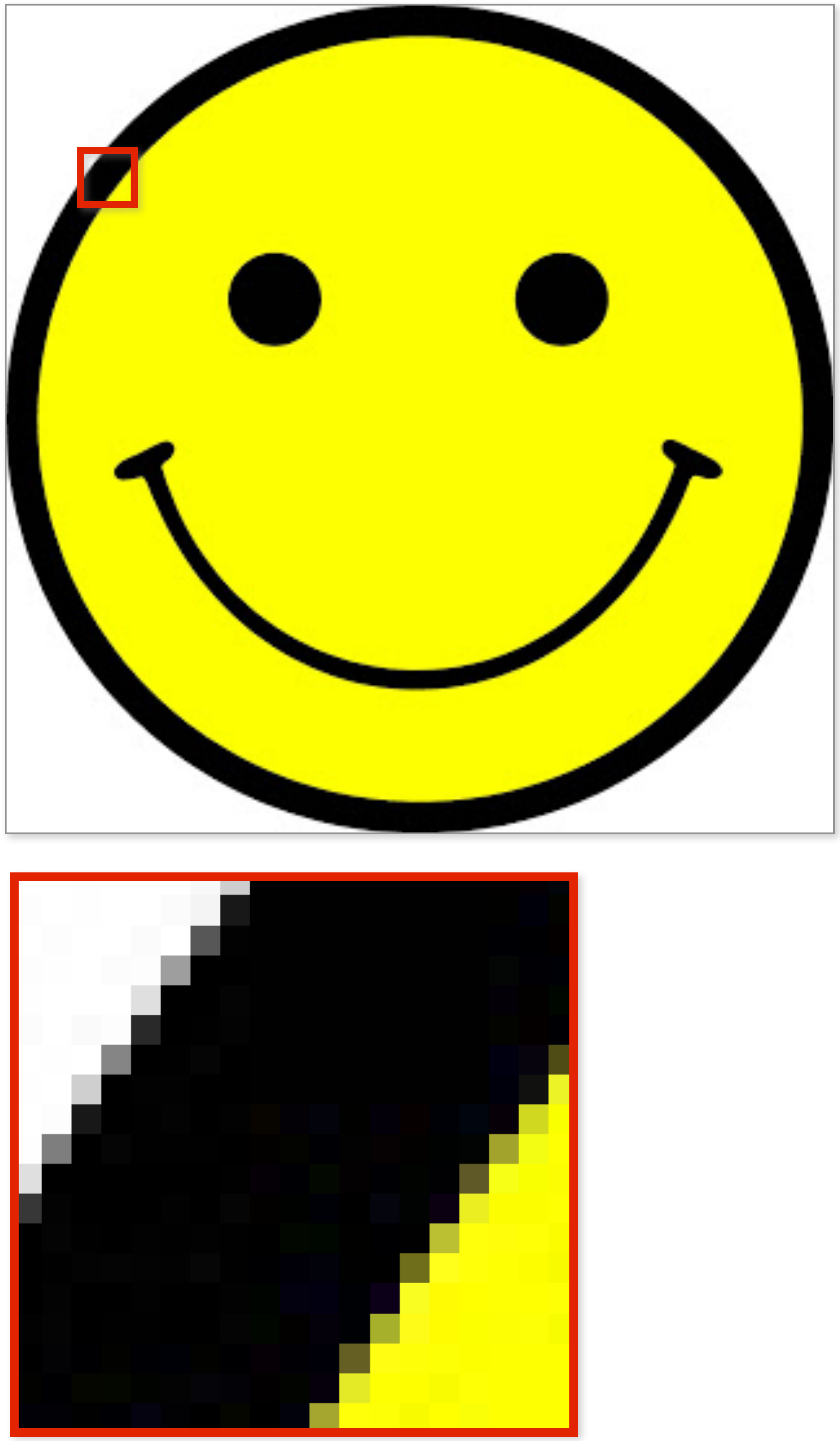


Image encoding in this example:

3 bits per texel + eight RGB values in palette (8x24 bits)

0	1	3	6
0	2	6	7
1	4	6	7
4	5	6	7

What is the compression ratio?



Per-block palette

- **Block-based compression scheme on 4x4 texel blocks**
  - Idea: there might be many unique colors across an entire image, but can approximate all values in any 4x4 texel region using only a few unique colors

- **Per-block palette (e.g., four colors in palette)**
  - 12 bytes for palette (assume 24 bits per RGB color: 8-8-8)
  - 2 bits per texel (4 bytes for per-texel indices)
  - 16 bytes (3x compression on original data: 16x3=48 bytes)

- **Can we do better?**

S3TC

(Called BC1 or DXTC by Direct3D)

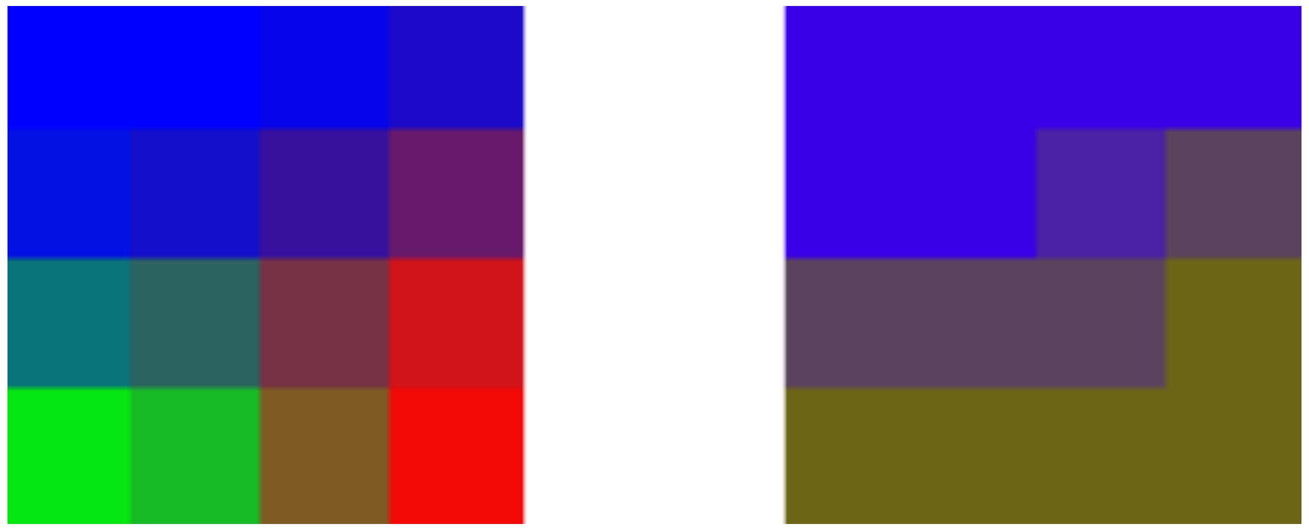
- **Palette of four colors encoded in four bytes:**
  - Two low-precision base colors: C_0 and C_1 (2 bytes each: RGB 5-6-5 format)
  - Other two colors computed from base values
    - $\frac{1}{3}C_0 + \frac{2}{3}C_1$
    - $\frac{2}{3}C_0 + \frac{1}{3}C_1$

- **Total footprint of 4x4 texel block: 8 bytes**
  - 4 bytes for palette, 4 bytes of color ids (16 texels, 2 bits per texel)
  - 4 bpp effective rate, 6:1 compression ratio (fixed ratio: independent of data values)

- **S3TC assumption:**
  - All texels in a 4x4 block lie on a line in RGB color space

- **Additional mode:**
  - If $C0 < C1$, then third color is $\frac{1}{2}C_0 + \frac{1}{2}C_1$ and fourth color is transparent black

S3TC artifacts

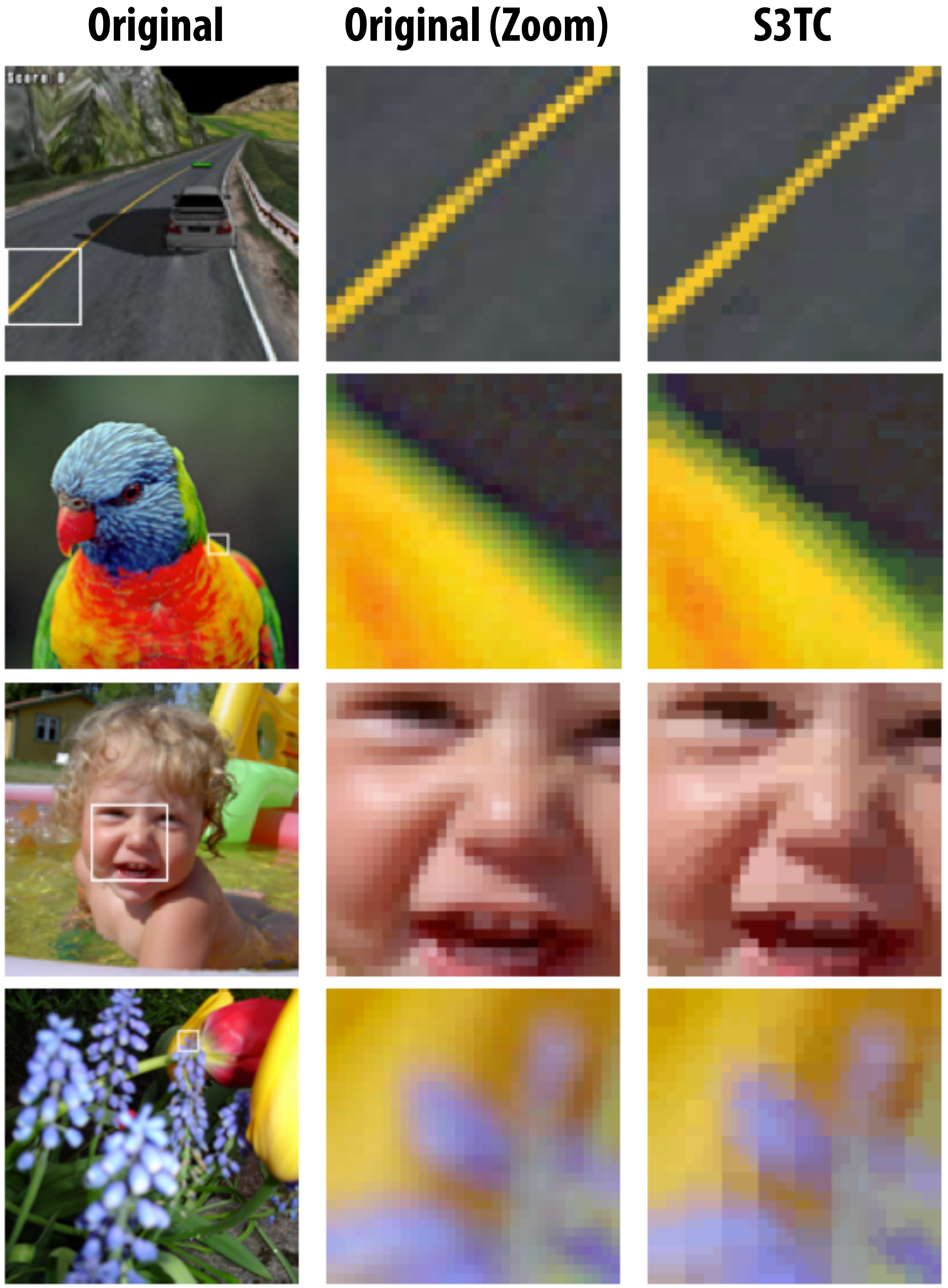


Original data　　　Compressed result

Cannot interpolate red and blue to get green (here compressor chose blue and yellow as base colors to minimize overall error)

But scheme works well in practice on "real-world" images. (see images at right)

Image credit:
http://renderingpipeline.com/2012/07/texture-compression/



[Strom et al. 2007]

PACKMAN

[Strom et al. 2004]

- Block-based compression on 2x4 texel blocks
  - Idea: vary luminance per texel, but specify single chrominance per block (similar idea as YUV 4:0:0)
- Each block encoded as:
  - A single base color per block (12 bits: RGB 4-4-4)
  - 4-bit index identifying one of 16 predefined luminance modulation tables
  - Per-texel 2-bit index into luminance modulation table (8x2=16 bits)
  - Total block size = 12 + 4 + 16 = 32 bits (6:1 compression ratio)
- Decompression:

```
texel[i] =  base_color + table[table_id][table_index[i]];
```

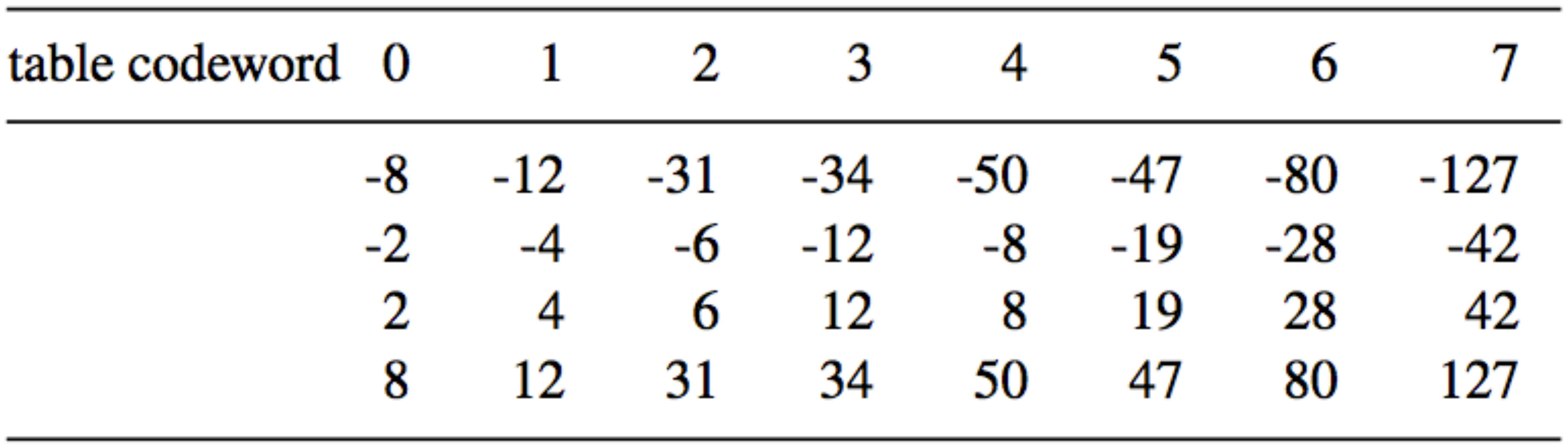


table codeword	0	1	2	3	4	5	6	7
	-8	-12	-31	-34	-50	-47	-80	-127
	-2	-4	-6	-12	-8	-19	-28	-42
	2	4	6	12	8	19	28	42
	8	12	31	34	50	47	80	127

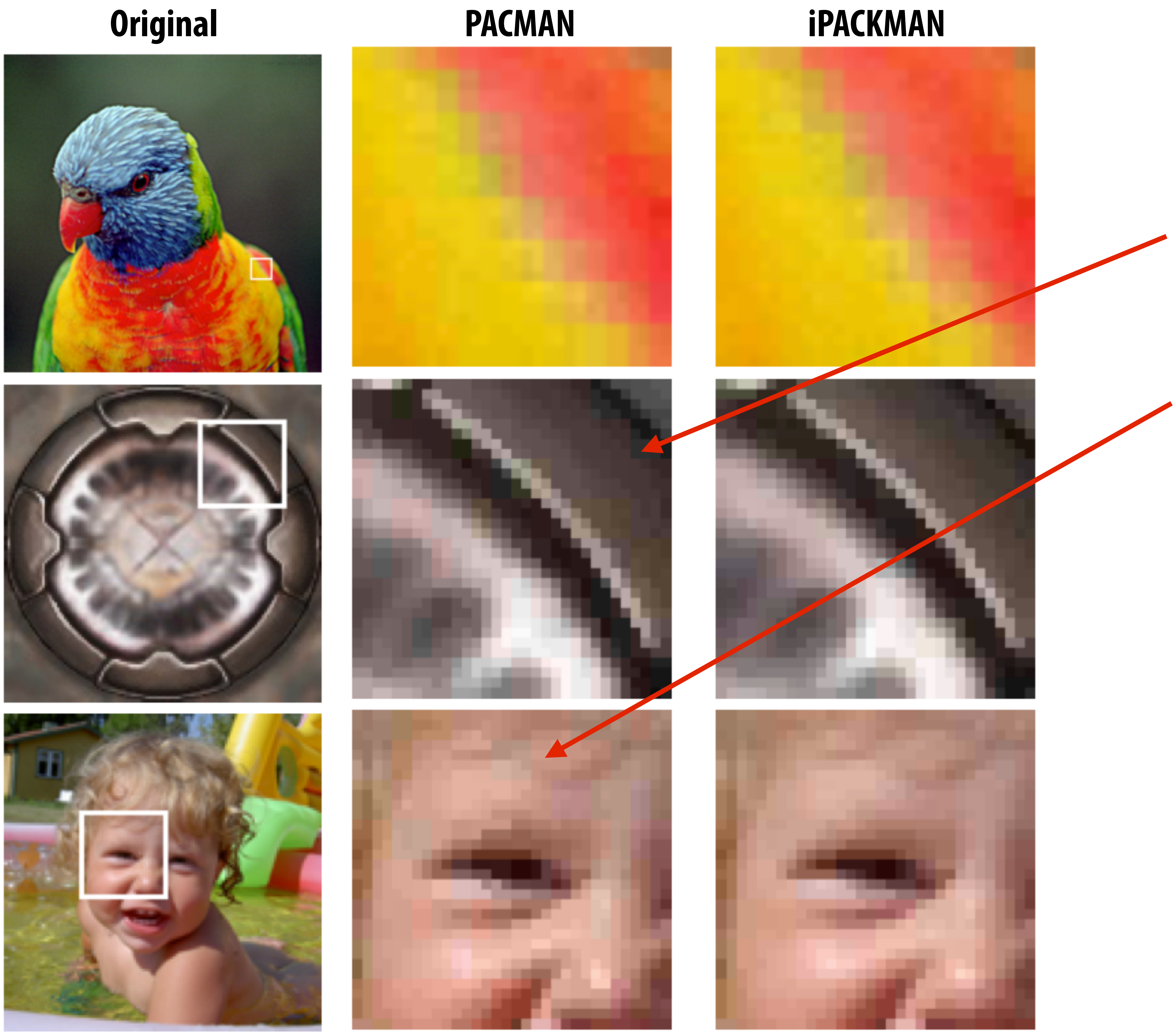
Example codebook for modulation tables (8 of 16 tables shown)

iPackman (ETC)

[Strom et al. 2005]

- Improves on problems of heavily quantized and sparsely represented chrominance in PACKMAN
  - Higher resolution base color + differential color represents color more accurately
- Operates on 4x4 texel blocks
  - Optionally represent 4x4 block as two eight-texel subblocks with differentials (else use PACKMAN for two subblocks)
    - 1 bit designates whether differential scheme is in use
  - Base color for first block (RGB 5-5-5: 15 bits)
  - Color differential for second block (RGB 3-3-3: 9 bits)
  - 1 bit designating if subblocks are 4x2 or 2x4
  - 3-bit index identifying modulation table per subblock (2x3 bits)
  - Per-texel modulation table index (2x16 bits)
  - Total compressed block size: 1 + 15 + 9 + 1 + 6 + 32 = 64 bits (6:1 ratio)

PACKMAN vs. iPACKMAN quality comparison



Chrominance banding

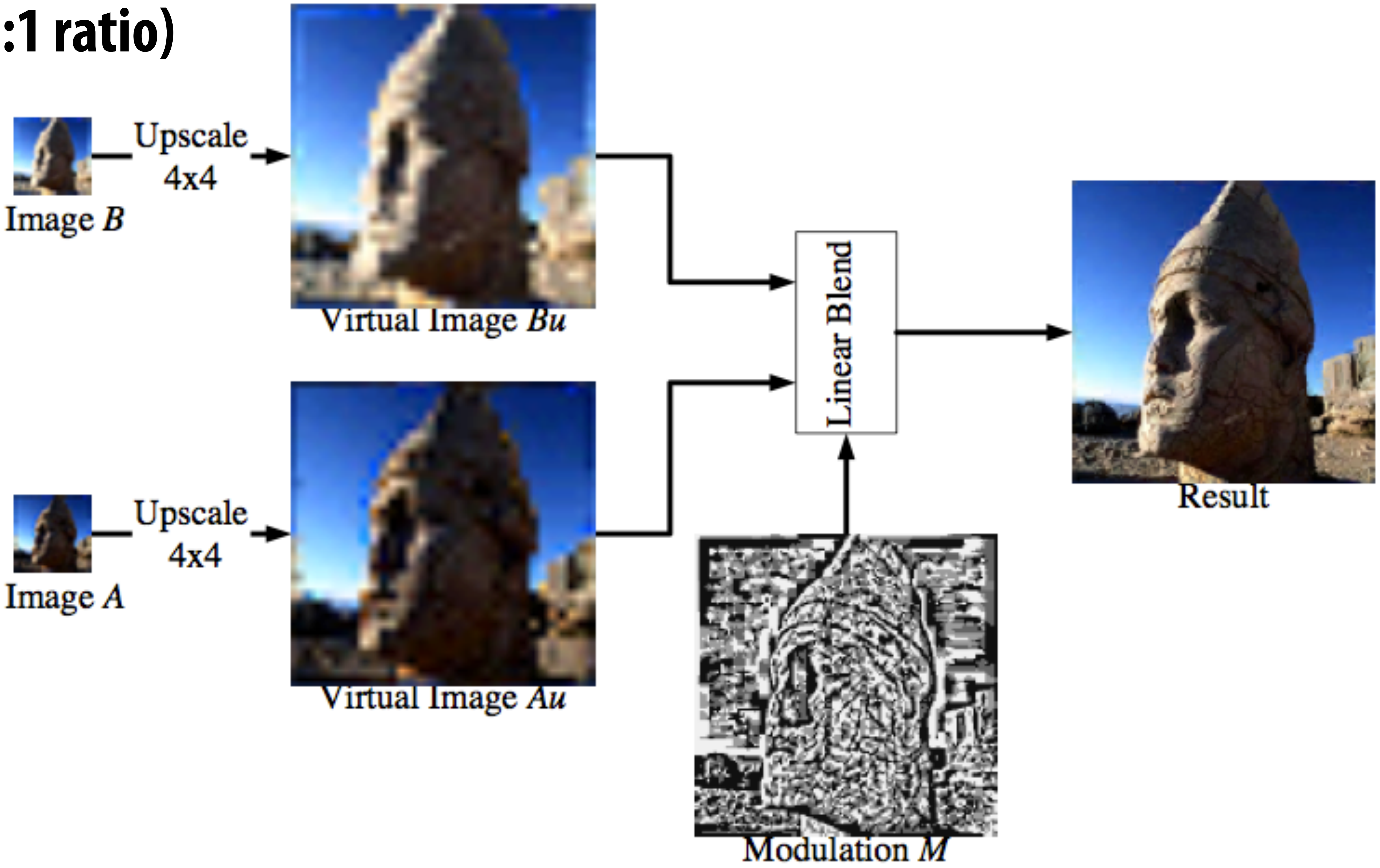
Chrominance block artifact

Image credit: Strom et al. 2005

PVRTC (Power VR texture compression)

[Fenney et al. 2003]

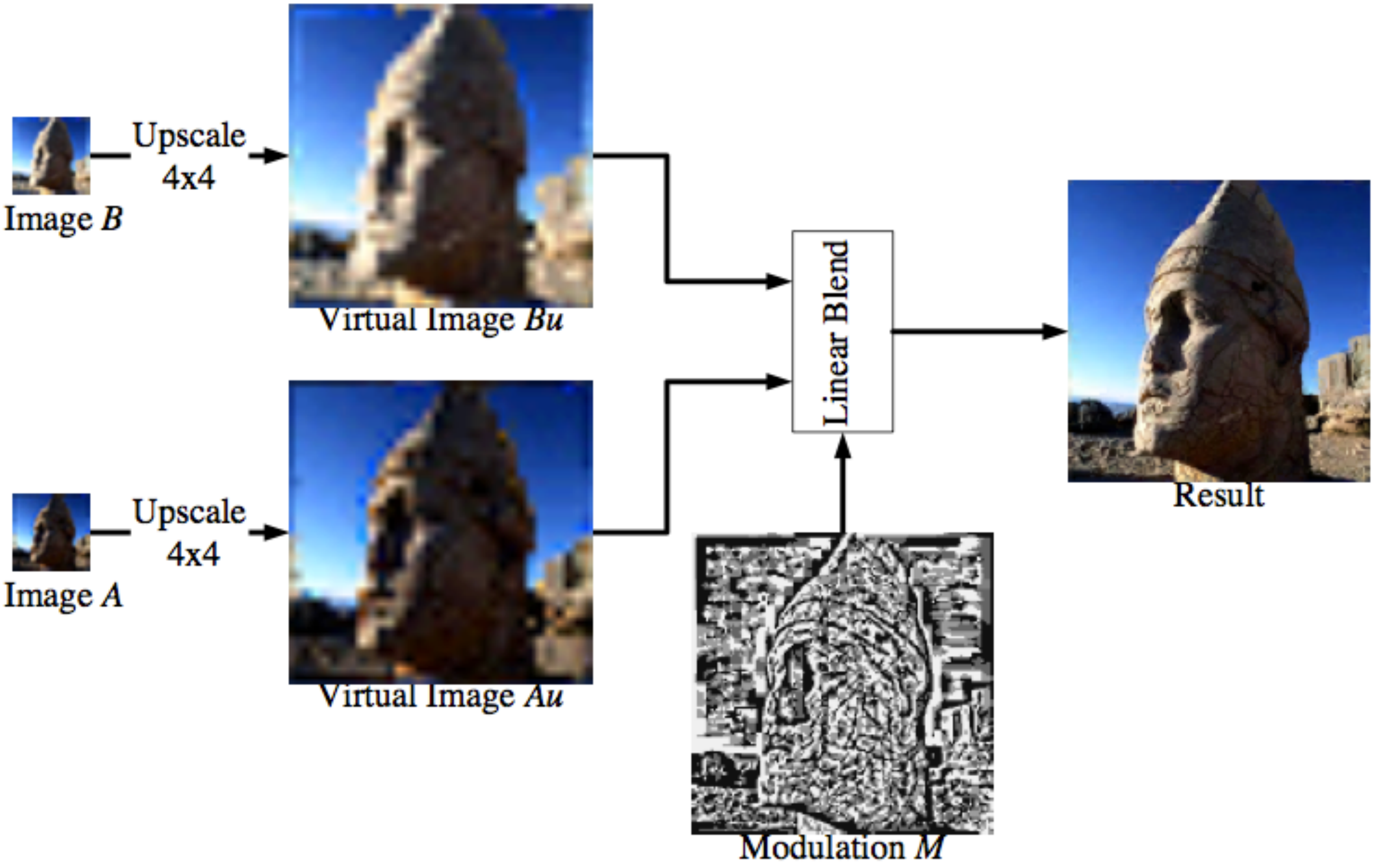
- Not a block-based format
  - Used in Imagination PowerVR GPUs
- Store low-frequency base images A and B
  - Base images downsampled by factor of 4 in each dimension ($^1/_{16}$ fewer texels)
  - Store base image pixels in RGB 5:5:5 format (+ 1 bit alpha)
- Store 2-bit modulation factor per texel
- Total footprint: 4 bpp (6:1 ratio)



PVRTC

[Fenney et al. 2003]

- **Decompression algorithm:**
  - Bilinear interpolate samples from A and B (upsample) to get value at desired texel
  - Interpolate upsampled values according to 2-bit modulation factor



PVRTC avoids blocking artifacts

Because it is not block-based

Recall: decompression algorithm involves bilinear upsampling of low-resolution base images

(Followed by a weighted combination of the two images)

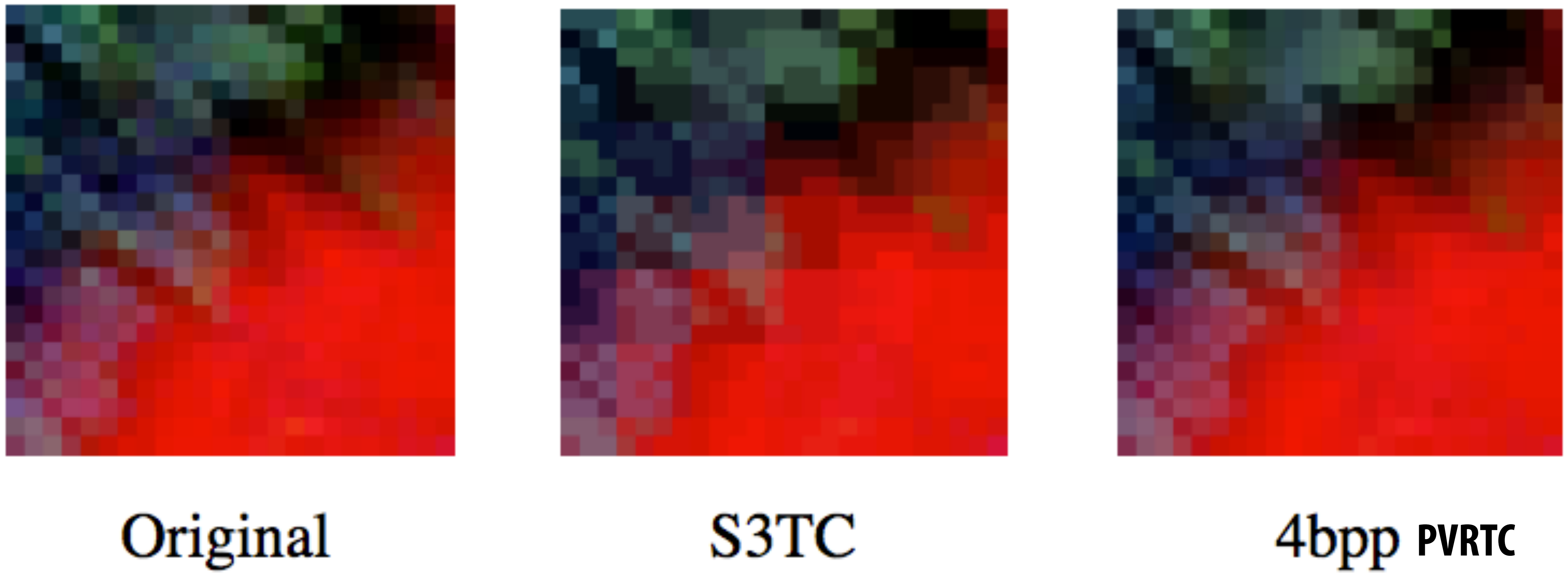


Image credit: Fenney et al. 2003

Summary: texture compression

- Many schemes target 6:1 fixed compression ratio (4 bpp)
  - Predictable performance
  - 8 bytes per 4x4-texel block is desirable for memory transfers

- Lossy compression techniques
  - Exploit characteristics of the human visual system to minimize <u>perceived</u> error
  - Texture data is read only, so “drift” due to multiple reads/writes is not a concern

- Block-based vs. not-block based
  - Block-based: S3TC/DXTC/BC1, iPACKMAN/ETC/ETC2, ASTC (not discussed today)
  - Not-block-based: PVRTC

- We only discussed decompression today:
  - Compression can be performed off-line (except when textures are generated at runtime… e.g., reflectance maps)

Hiding the latency of texture sampling and texture data access

Texture sampling is a high-latency 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 quad-fragment samples
3. Compute *d*
4. Convert normalized texture coordinate (u,v) to texture coordinates texel_u, texel_v
5. Compute required texels in window of filter **
6. If texture data in filter footprint (eight texels for trilinear filtering) is not in cache:
    - Load required texels (in compressed form) from memory
    - Decompress texture data
7. Perform tri-linear interpolation according to (texel_u, texel_v, d)

Latency of texture fetch involves time to perform math for texel address computation, decompression, and filtering (not just latency of fetching data from memory)

** May involve wrap, clamp, etc. of texel coordinates according to sampling mode configuration

Addressing texture sampling latency

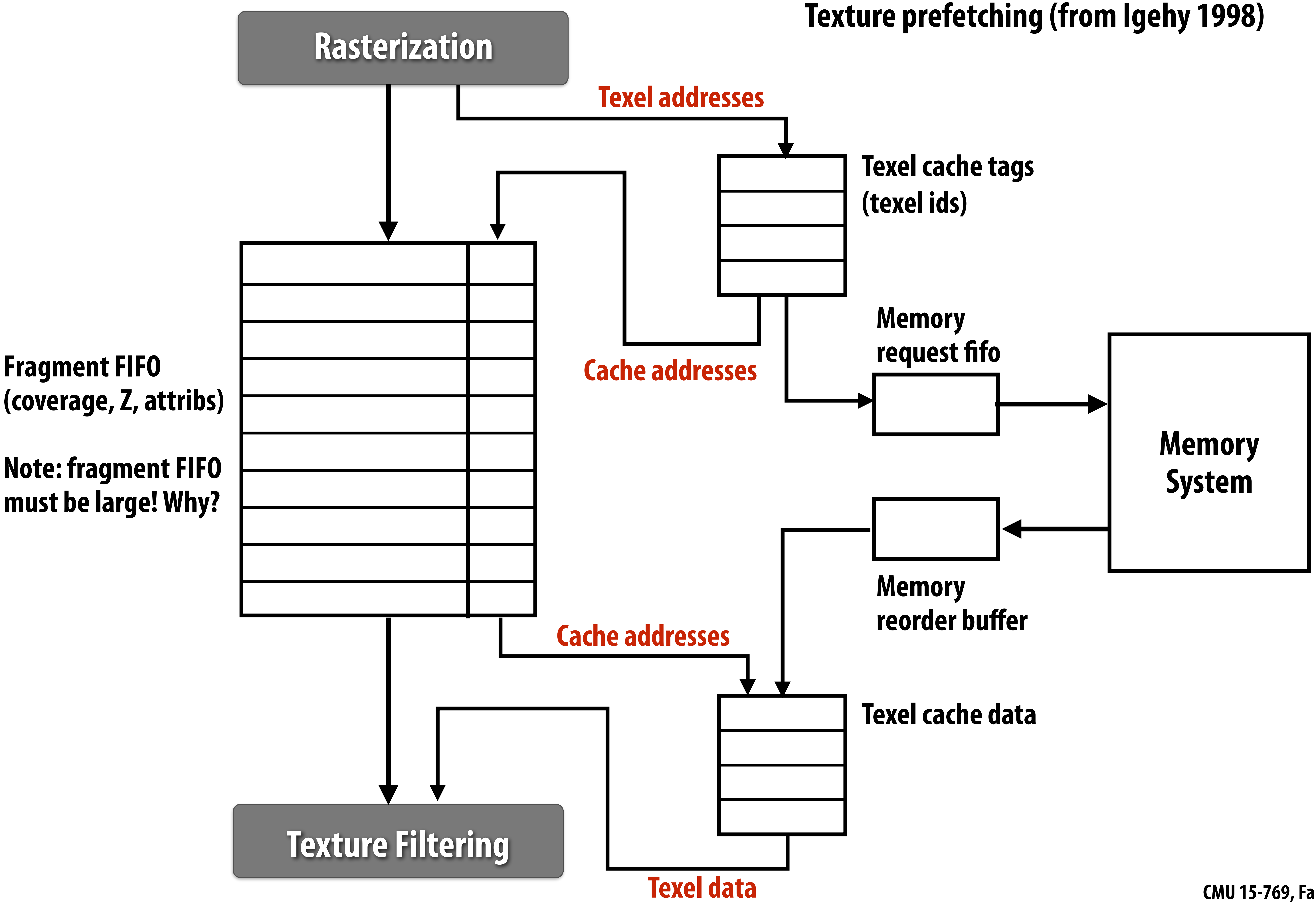
- Processor requests filtered texture data → processor waits hundreds of cycles (significant loss of performance)

- Solution prior to programmable GPU cores: texture data prefetching
  - Igehy et al. *Prefetching in a Texture Cache Architecture*

- Solution in all modern GPUs: multi-threaded processor cores

Prefetching example: large fragment FIFOs

Texture prefetching (from Igehy 1998)



Rasterization

Texel addresses

Texel cache tags
(texel ids)

Memory
request fifo

Memory
System

Cache addresses

Fragment FIFO
(coverage, Z, attribs)

Note: fragment FIFO
must be large! Why?

Memory
reorder buffer

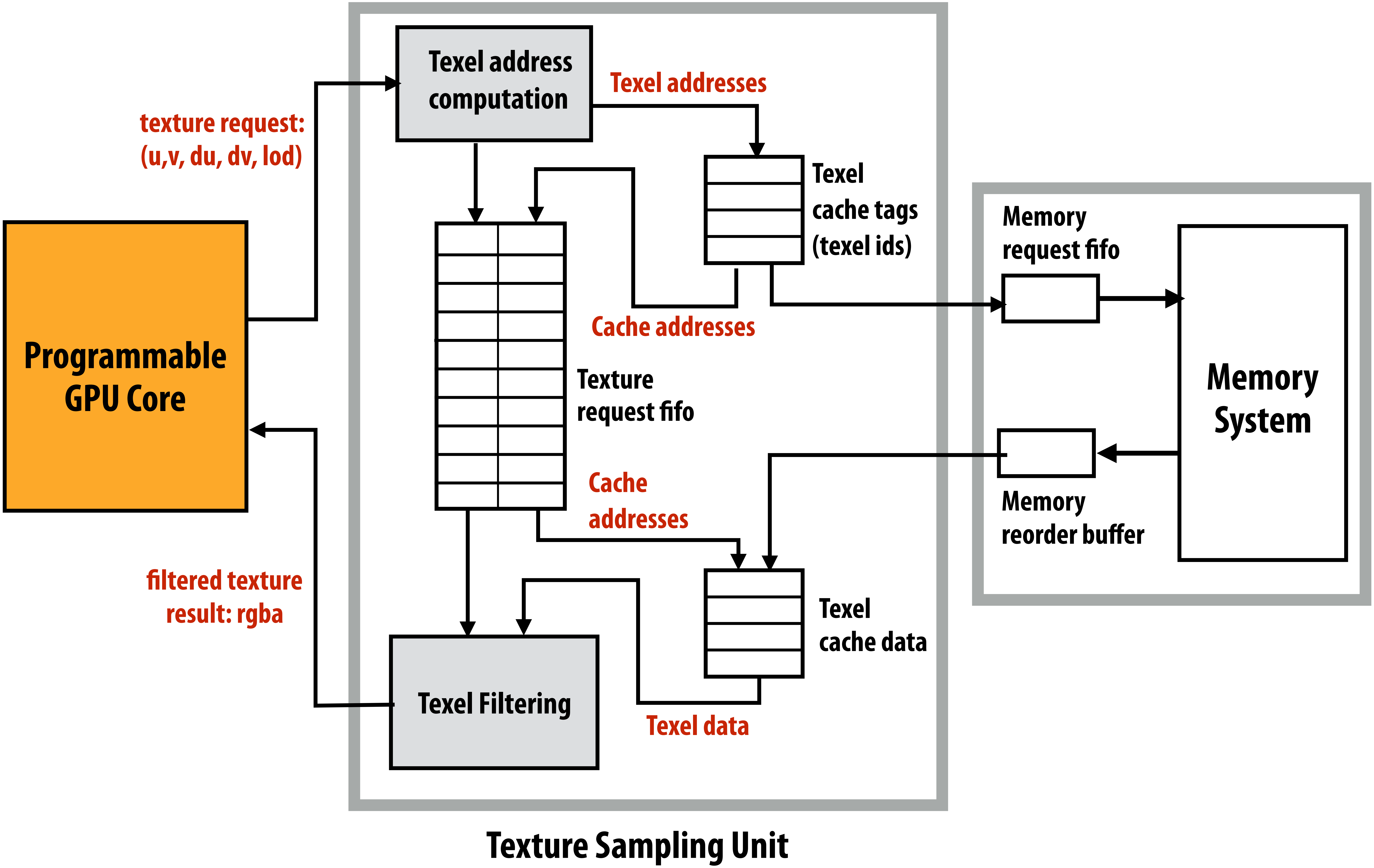
Cache addresses

Texel cache data

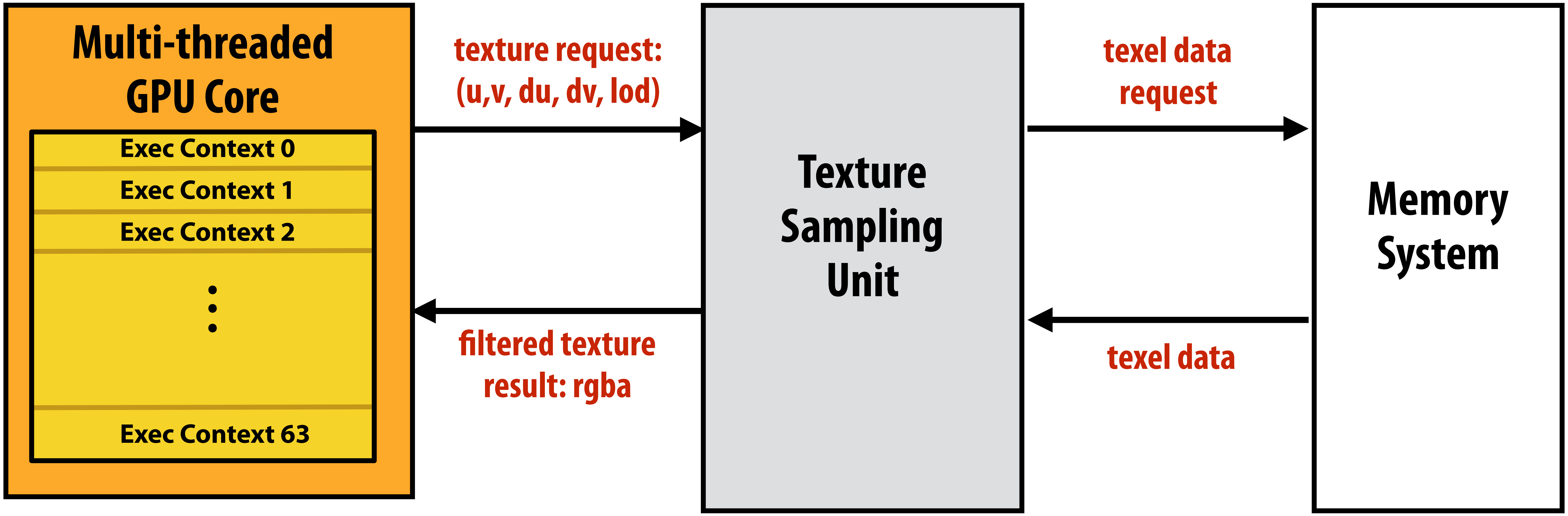
Texture Filtering

Texel data

A more modern design



Modern GPUs: texture latency is hidden via hardware multi-threading



GPU executes instructions from runnable fragments when other fragments are waiting on texture sampling responses.

Fragment FIFO from Igehy prefetching design is now represented by live fragment state in the programmable core.

GPU texture system summary

- A texture lookup is a lot more than a 2D array access
  - Significant computational and bandwidth expense
  - Implemented in specialized fixed-function hardware

- Bandwidth reduction mechanism: GPU texture caches
  - Primarily serve to amplify limited DRAM bandwidth, not reduce latency to off-chip memory
  - Small capacity compared to CPU caches, but high BW (need eight texels at once)
  - Tiled rasterization order + tiled texture layout optimizations increase cache hits

- Bandwidth reduction mechanism: texture compression
  - Lossy compression schemes
  - Fixed-compression ratio encodings (e.g, 6:1 ratio, 4 bpp is common for RGB data)
  - Schemes permit random access into compressed representation

- Latency avoidance/hiding mechanisms:
  - Prefetching (in the old days)
  - Multi-threading (in modern GPUs)