Lecture 15:

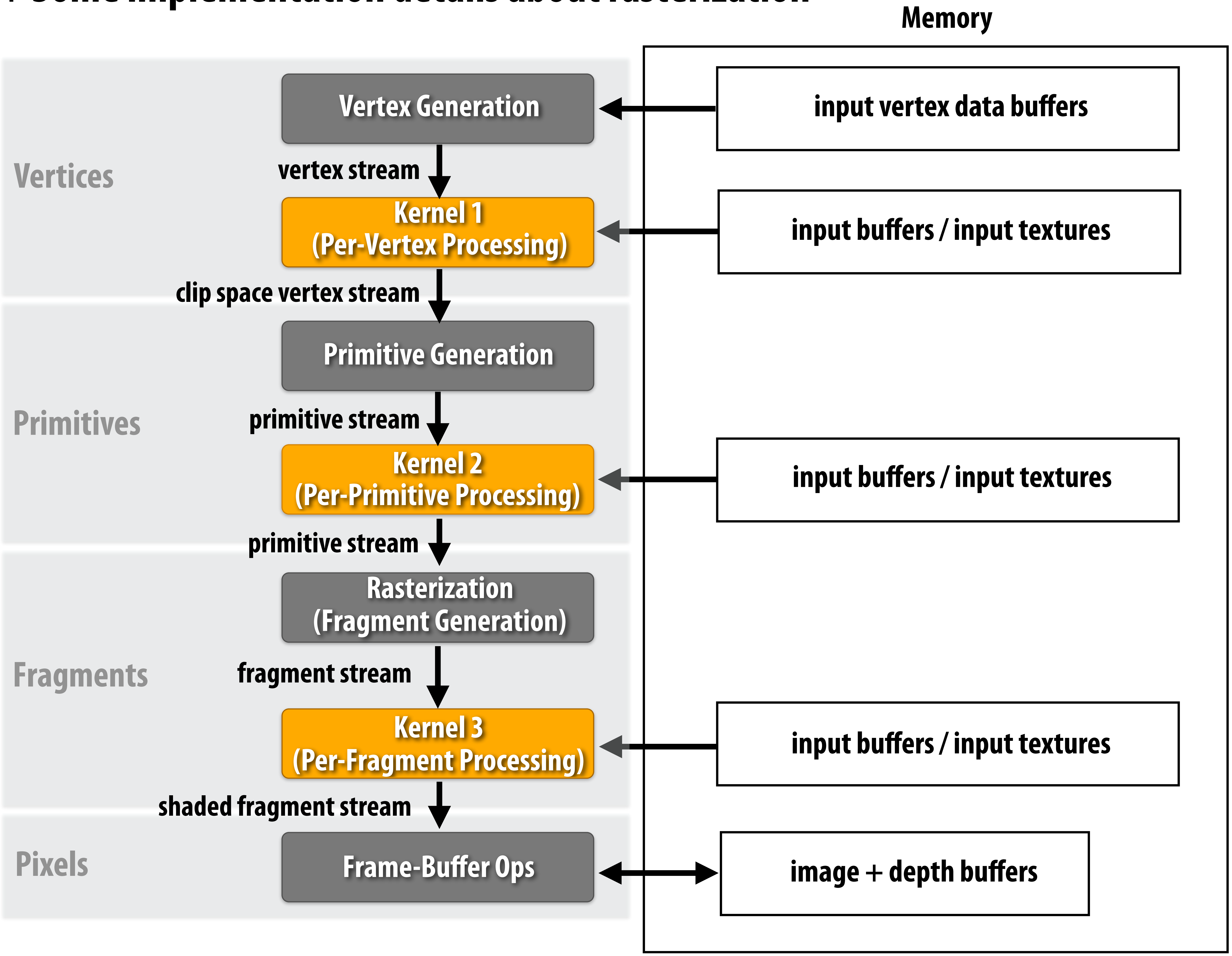
Data Access in the Graphics Pipeline: Efficient Implementations of Texture Mapping and Depth Buffering

---

Visual Computing Systems
Stanford CS348K, Fall 2018

Last time: graphics pipeline architecture

+ Some implementation details about rasterization



Today: revisiting a major course theme: efficiently handling data access

- **Q. Why be efficient with data access?**

- **Answer: performance cost: performance of modern parallel applications is often bandwidth-limited on modern computers**

- **Answer: energy cost: high cost of moving data**

How did Halide help address this problem?

- Scheduling primitives for improving producer-consumer locality

```
blurx(x,y) = (in(x-1, y) + in(x,y) + in(x+1,y)) / 3.0f;
out(x,y)   = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;

out.tile(x, y, xi, yi, 256, 32);
```

```
blurx.compute_at(out, x);
```

Compute necessary elements of blurx within out's x loop nest (all necessary elements for one tile of out)

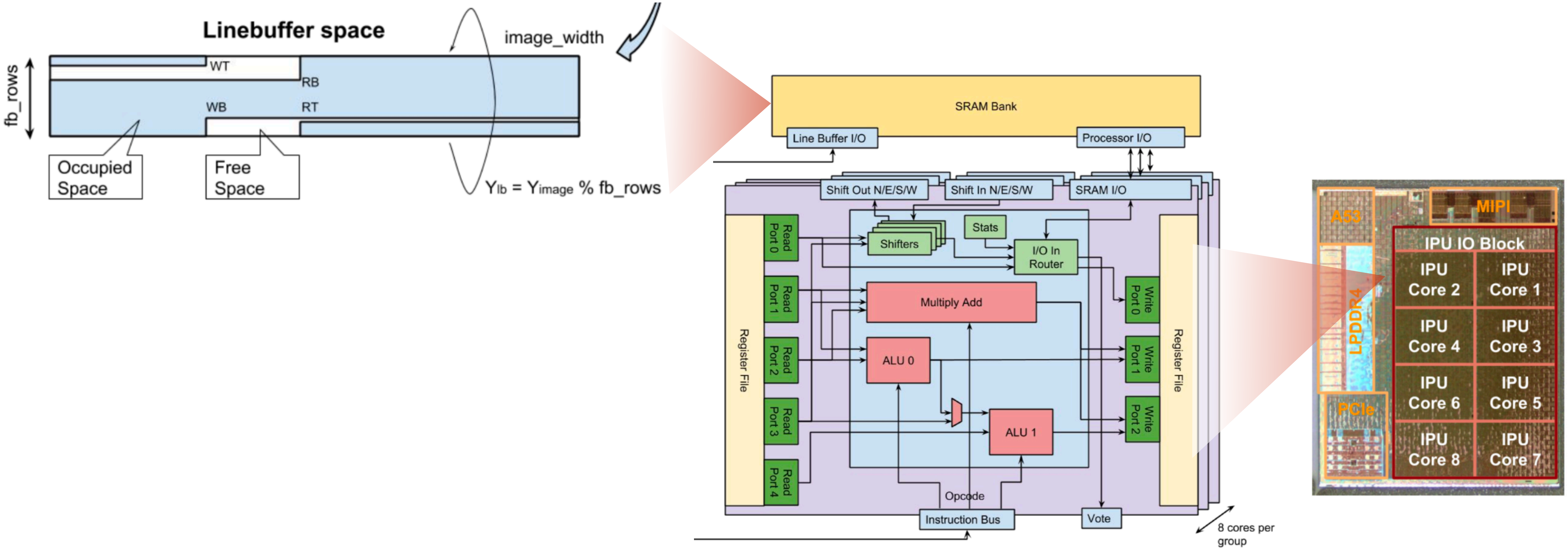
```
for y=0 to num_tiles_y:
   for x=0 to num_tiles_x:

      allocate 258x34 buffer for tile blurx
      for yi=0 to 32+2:
         for xi=0 to 256+2:
            tmp_blurx(xi,yi) = // compute blurx from in

      for yi=0 to 32:
         for xi=0 to 256:
            idx_x = x*256+xi;
            idx_y = y*32+yi
            out(idx_x, idx_y) = …
```

tile of blurx is computed here (and hopefully retained in cache)

tile of blurx is consumed here

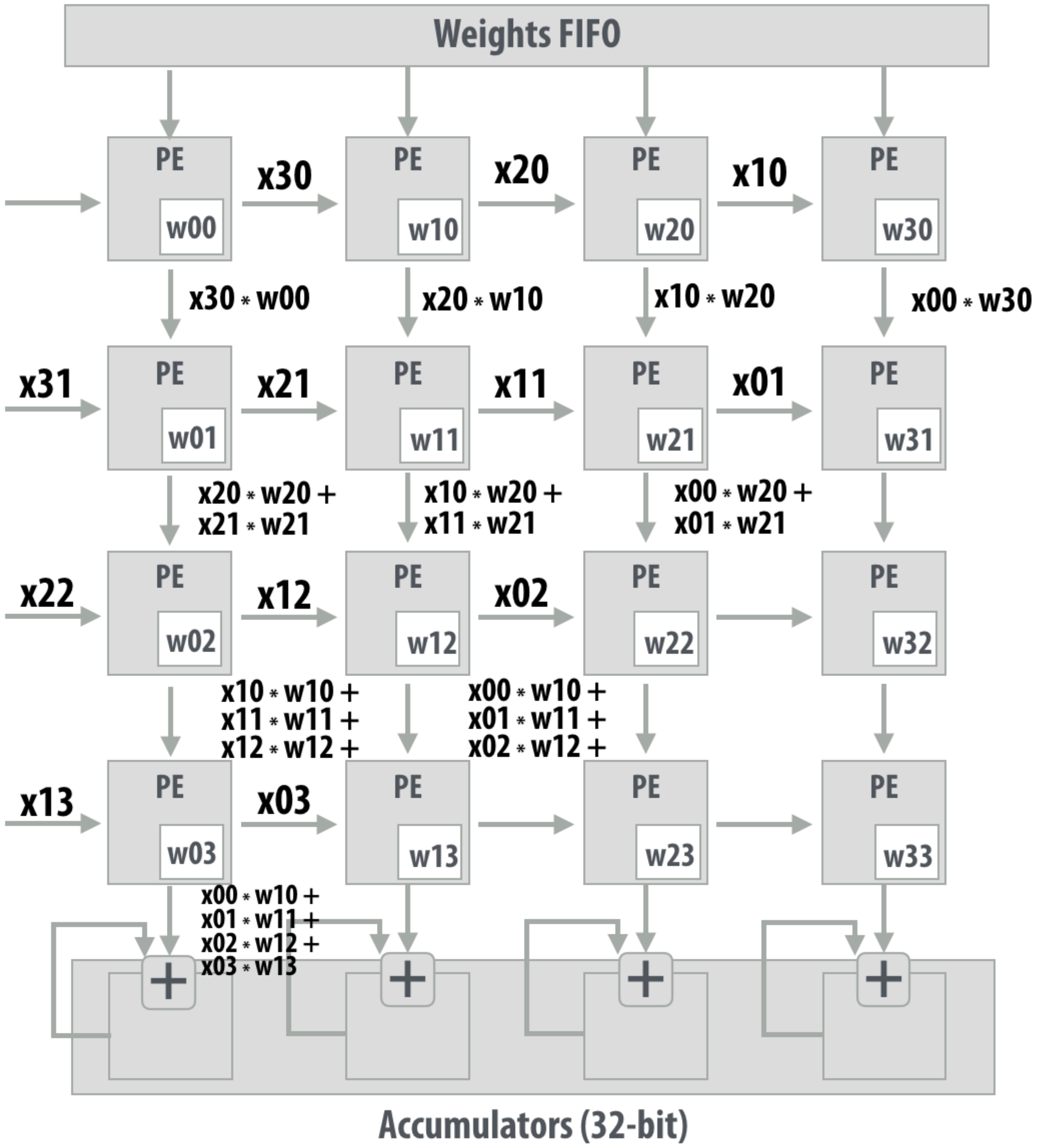
What elements have we seen in image processing hardware architectures?

- On-chip storage for intermediate tiles/lines of an image
  - e.g., Pixel Visual Core

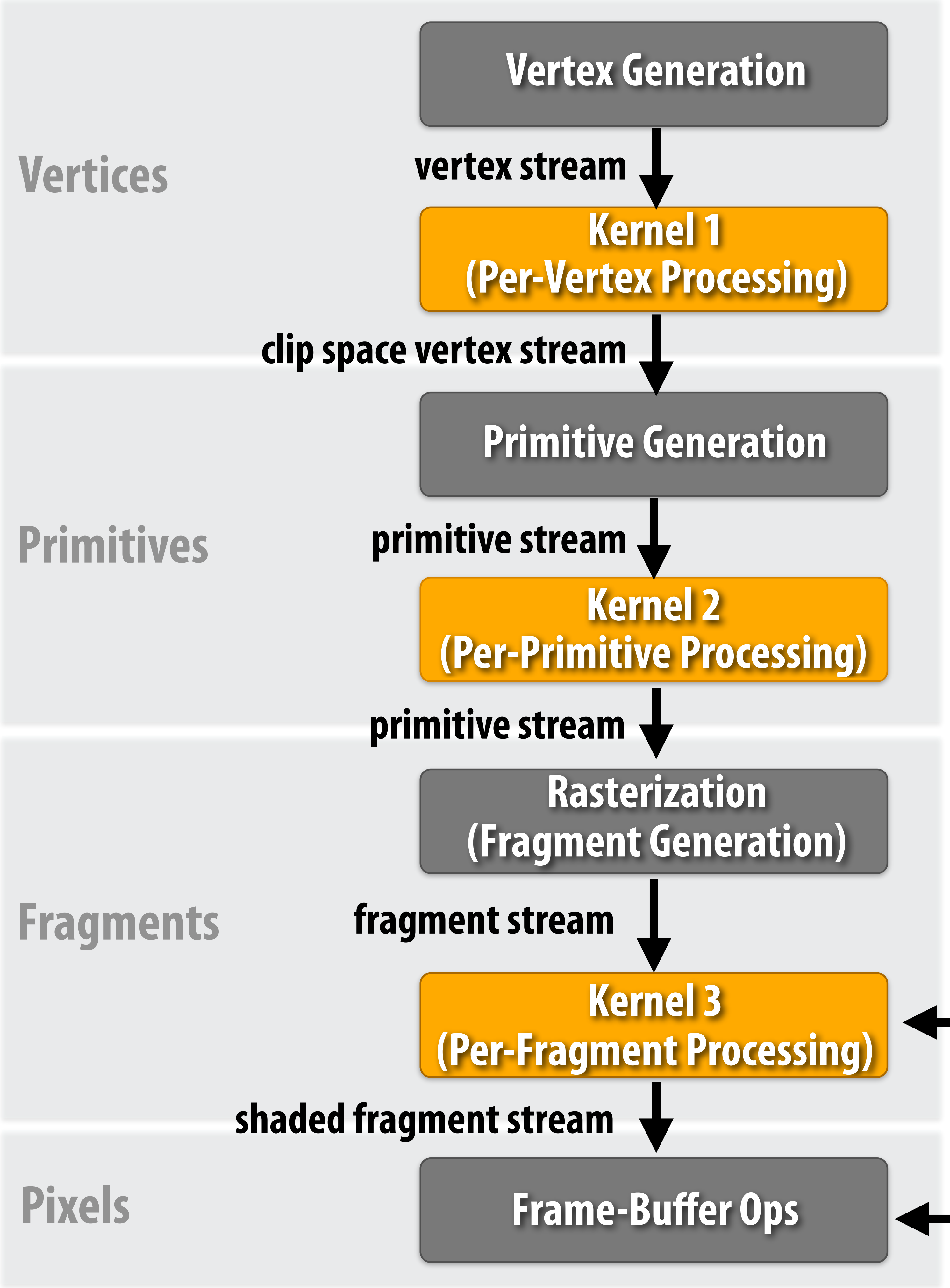


What elements did we see in DNN hardware accelerators?

- **Systolic array architectures: data flows efficiently between processing elements (PE's don't reload data from memory)**



Memory access in the graphics pipeline



Vertices

Vertex Generation

vertex stream

Kernel 1
(Per-Vertex Processing)

clip space vertex stream

Primitives

Primitive Generation

primitive stream

Kernel 2
(Per-Primitive Processing)

primitive stream

Fragments

Rasterization
(Fragment Generation)

fragment stream

Kernel 3
(Per-Fragment Processing)

shaded fragment stream

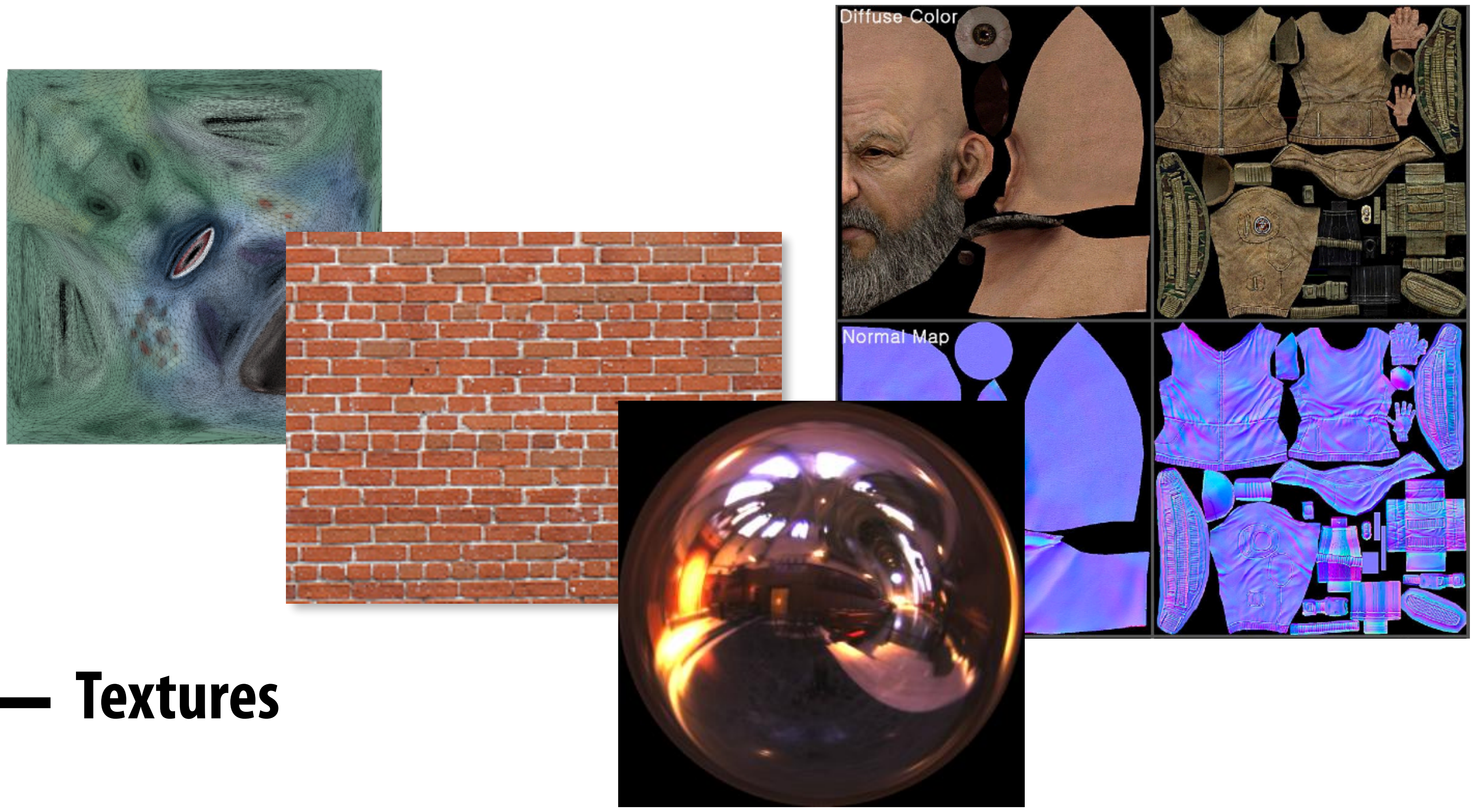
Pixels

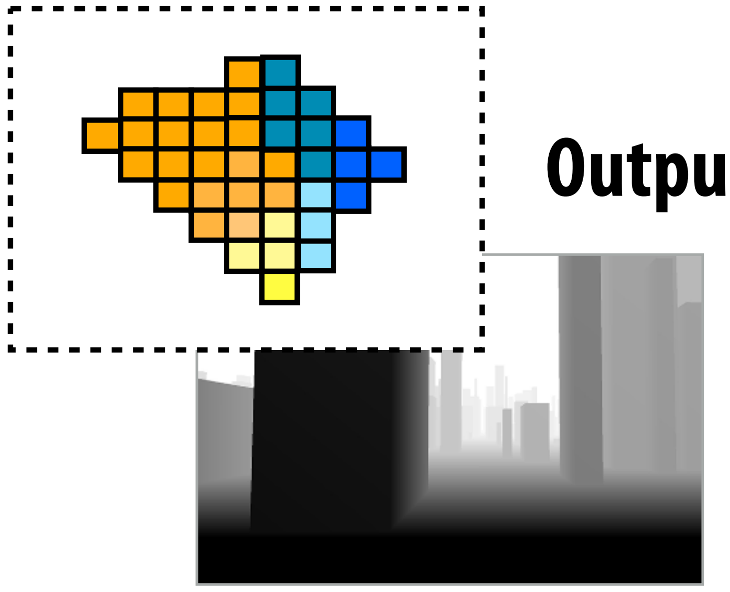
Frame-Buffer Ops

Sources of data access in the graphics pipeline:

- Inter-stage stream buffers
- Texture access (read only access to texture data)
- Frame-buffer access (read/write access to color and depth buffer)

Textures





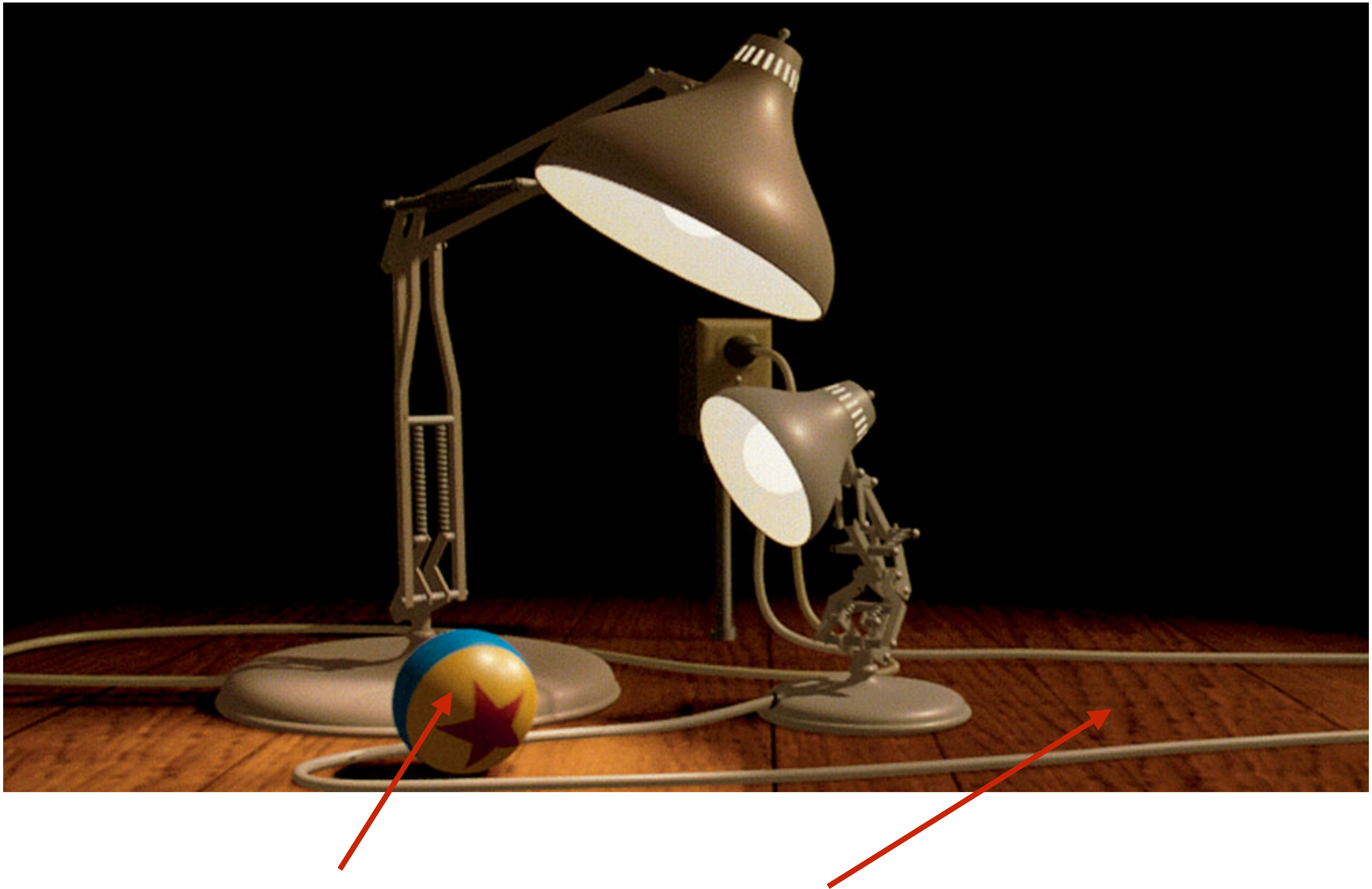
Output image buffer

Depth buffer

Part 1:
efficient implementation of texture mapping

Many uses of texture mapping

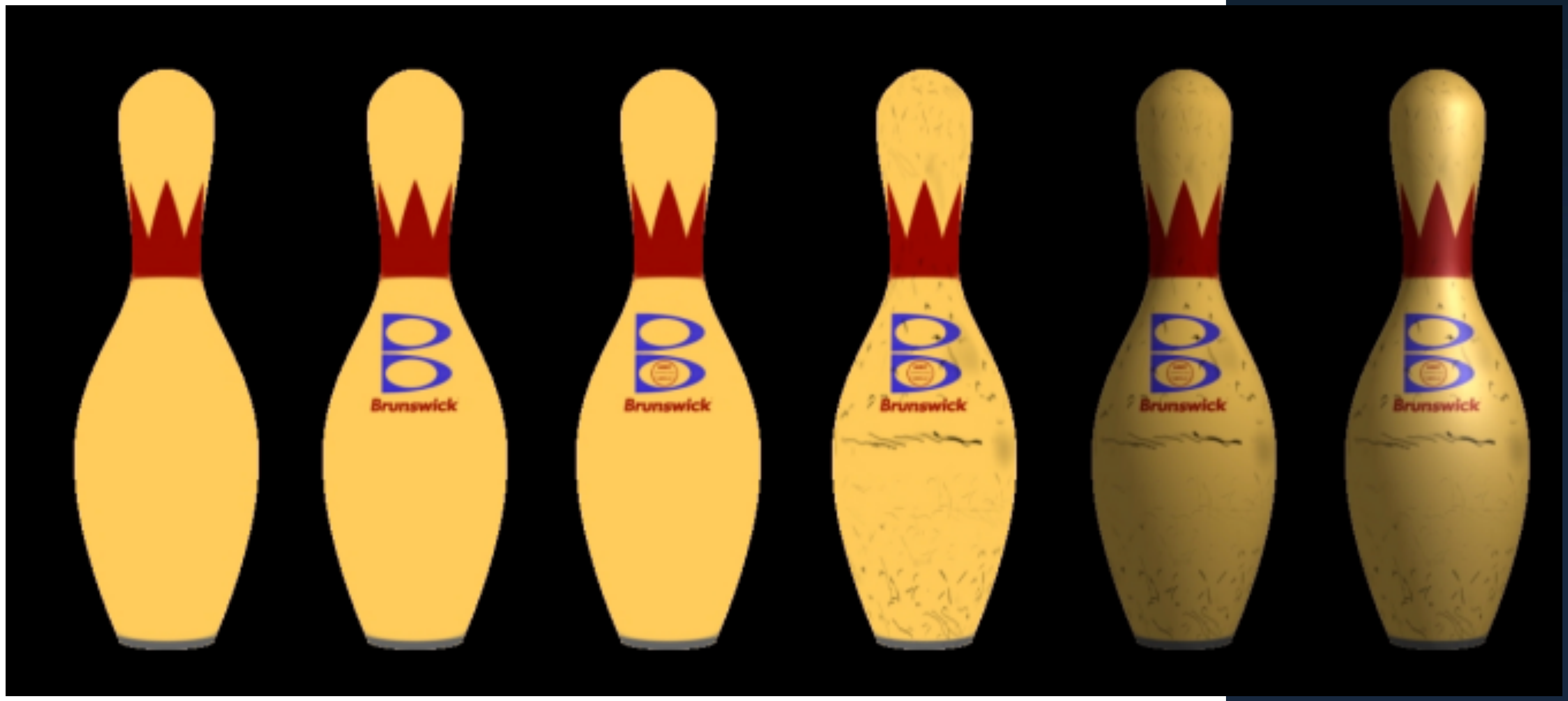
Define spatial variation in surface reflectance



Pattern on ball

Wood grain on floor

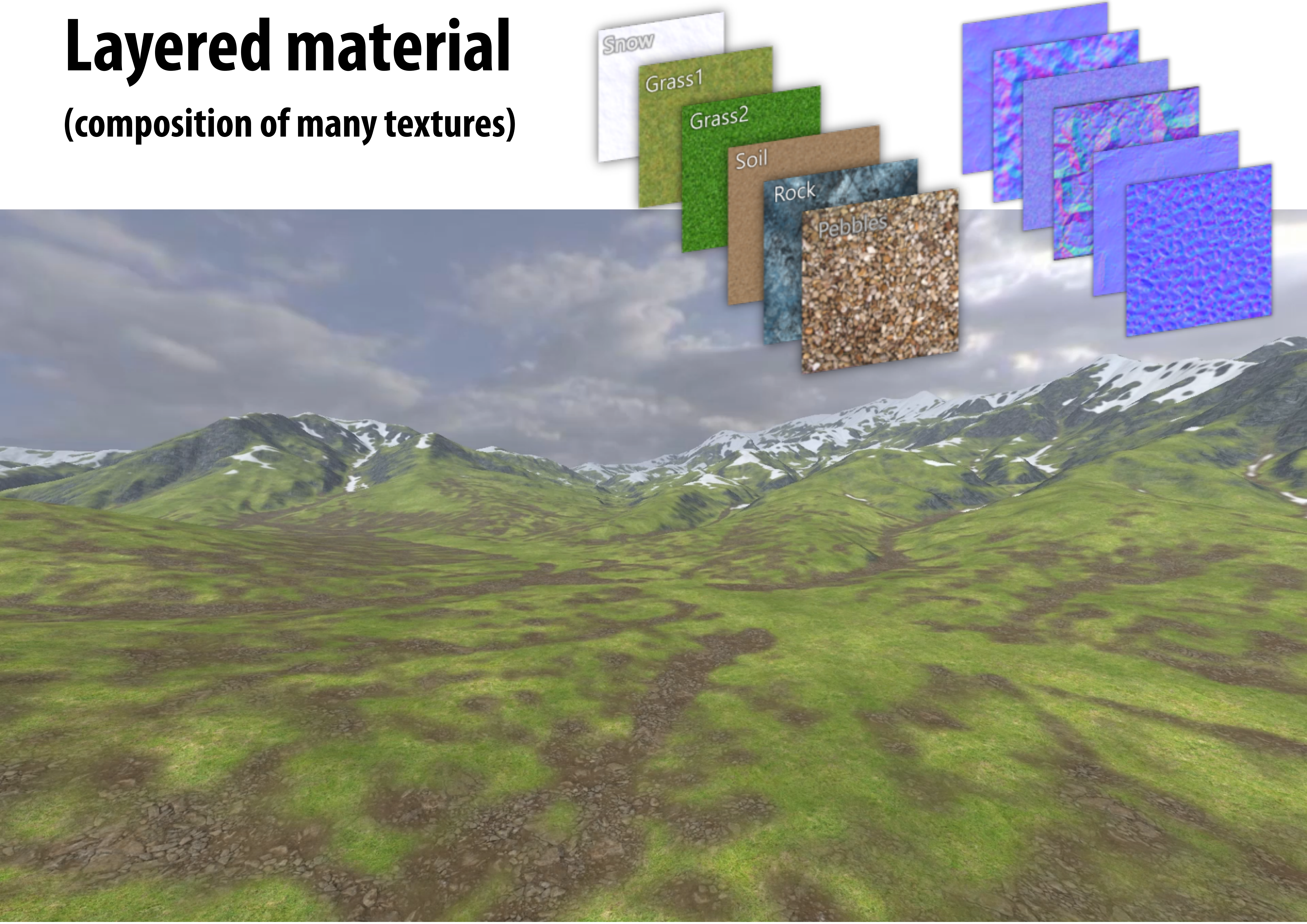
Describe surface material properties



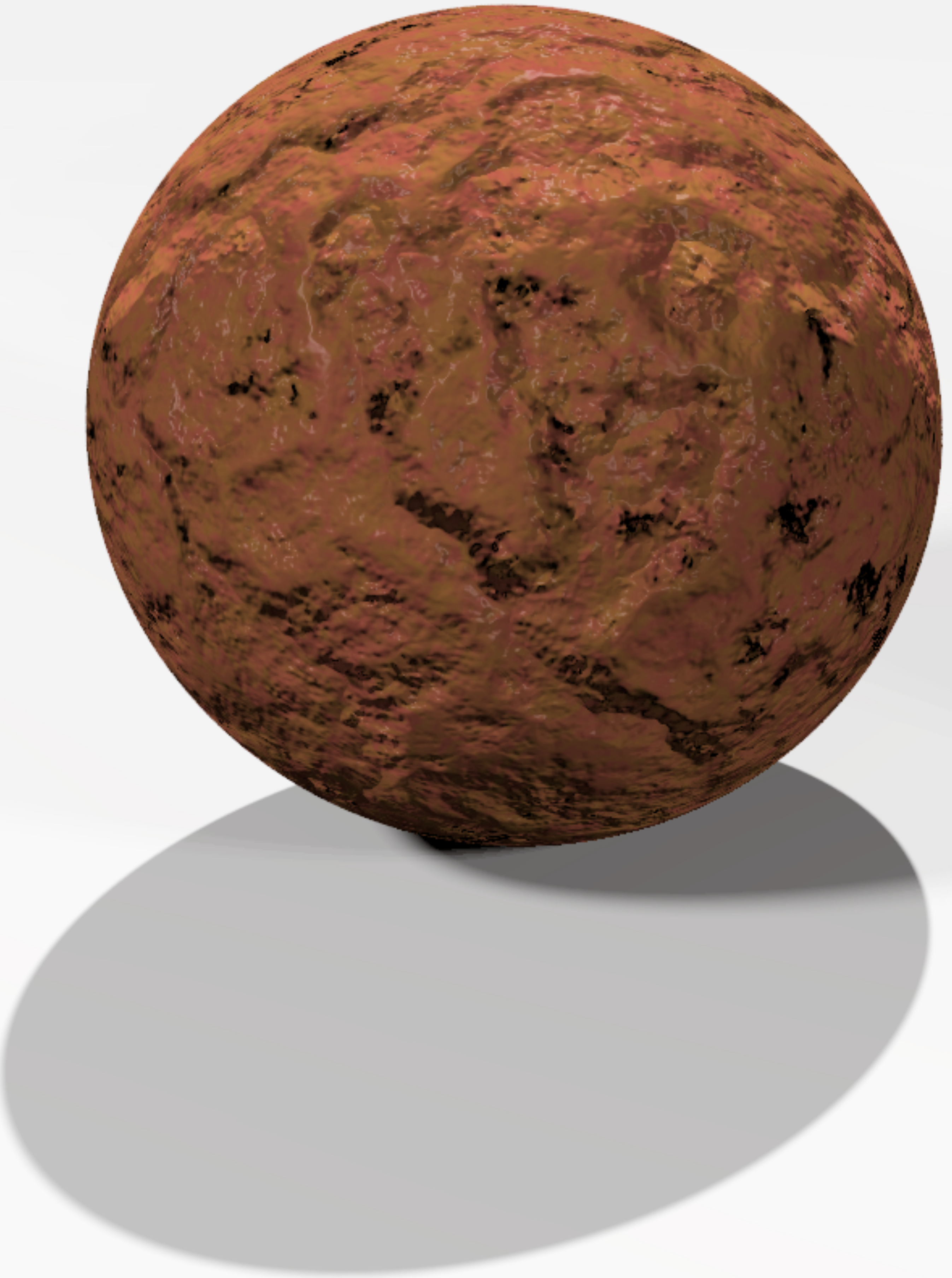
Multiple layers of texture maps for color, logos, scratches, etc.

Layered material

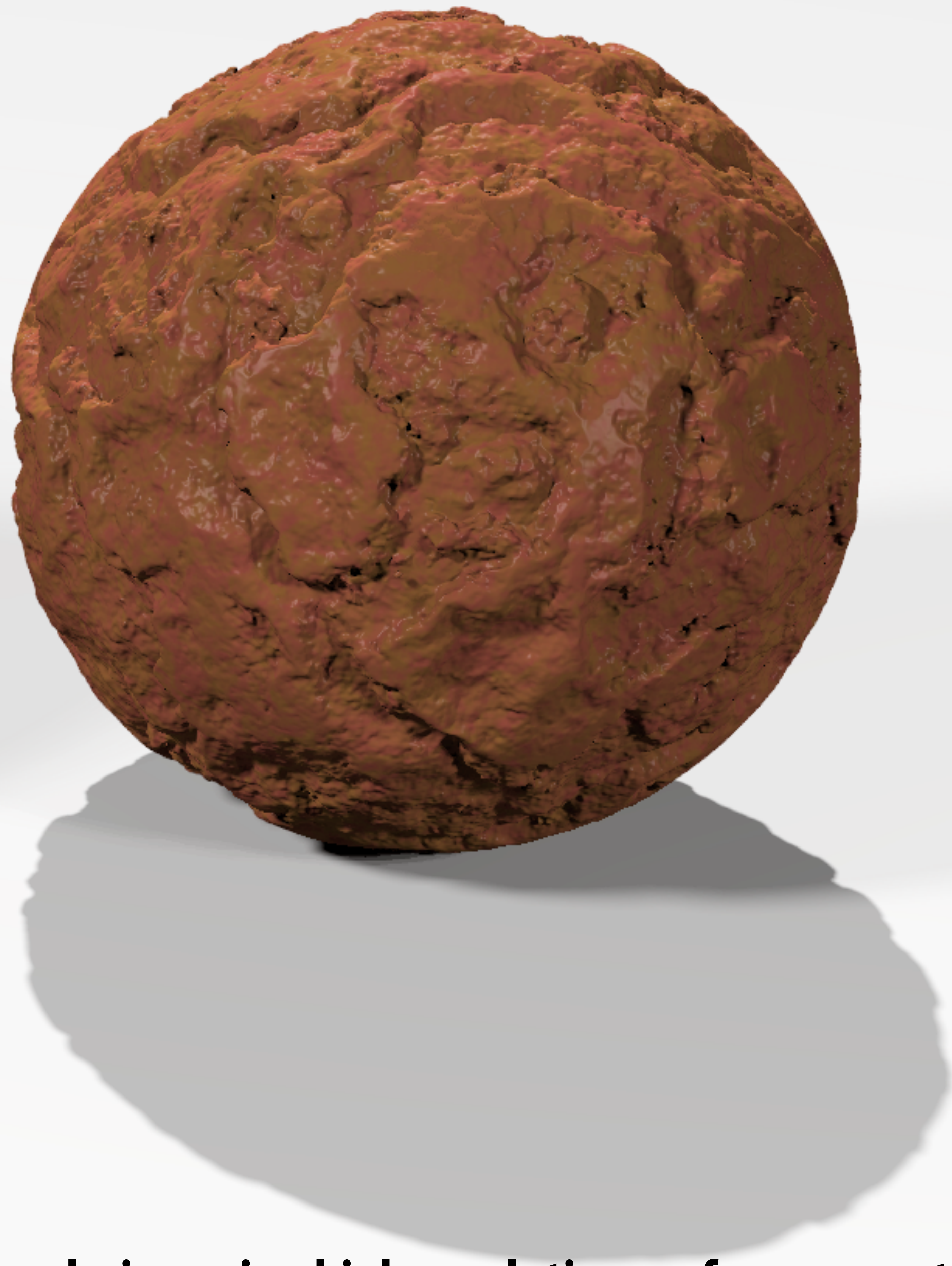
(composition of many textures)



Normal mapping: texture encodes perturbation of surface normal



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



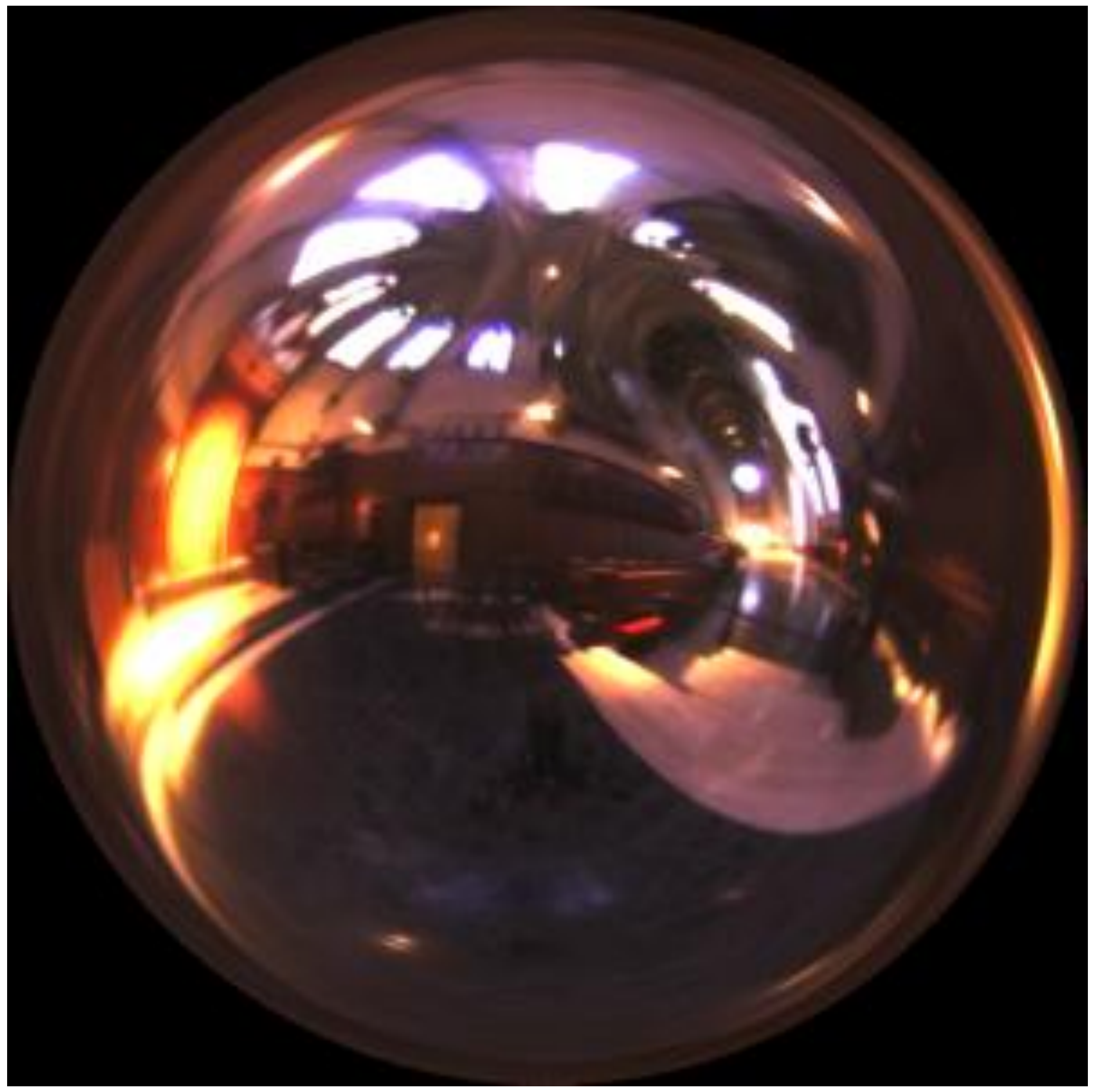
Rendering using high-resolution surface geometry
(note bumpy silhouette and shadow boundary)

Textures encode precomputed lighting and shadows


Original model

Extracted ambient occlusion map

With ambient occlusion

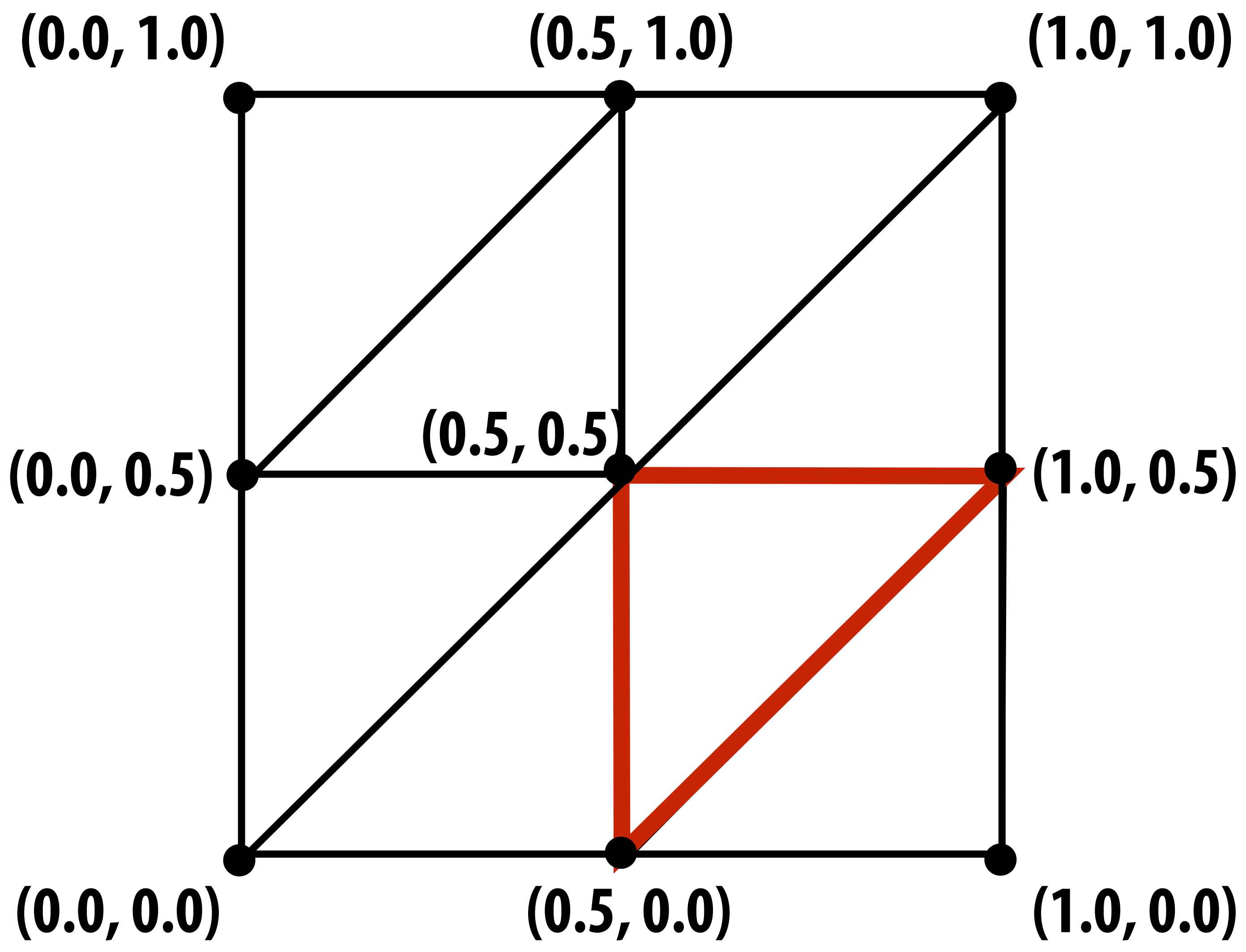

Grace Cathedral environment map


Environment map used in rendering

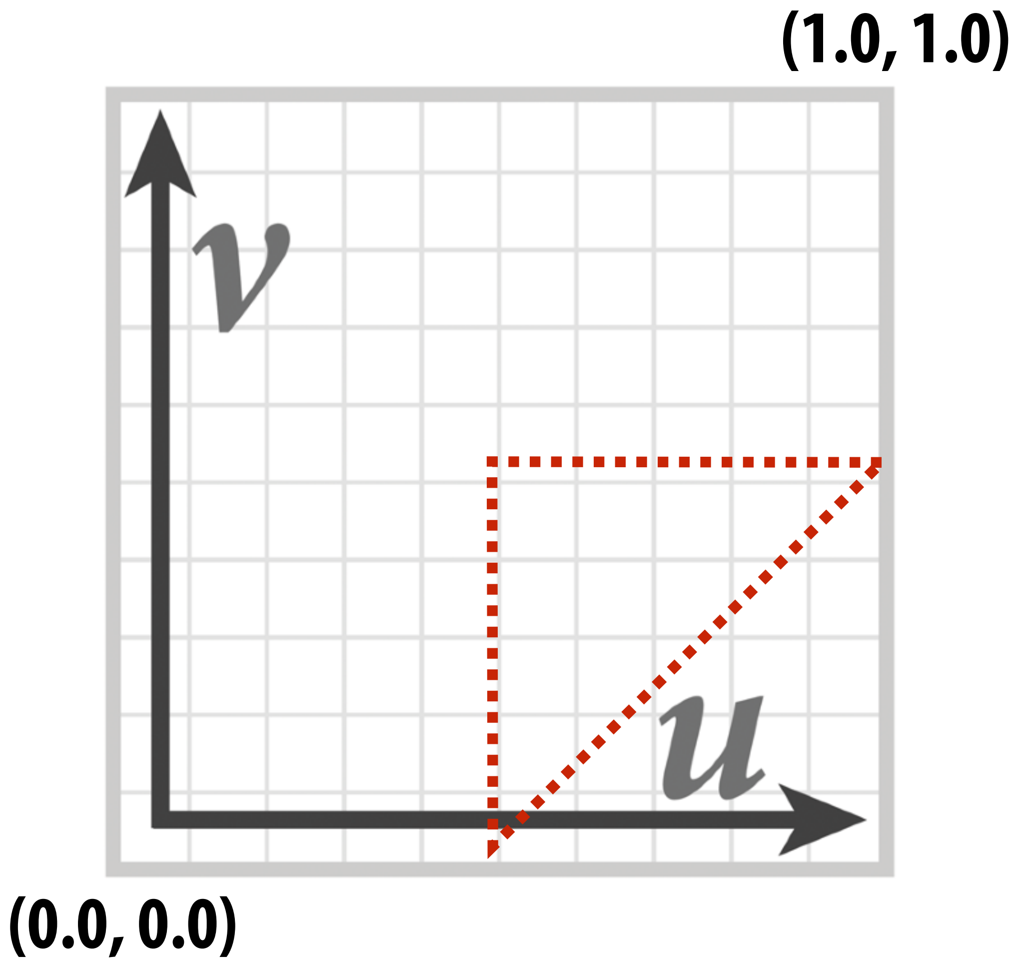
Background:
Texture mapping math

Texture coordinates

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



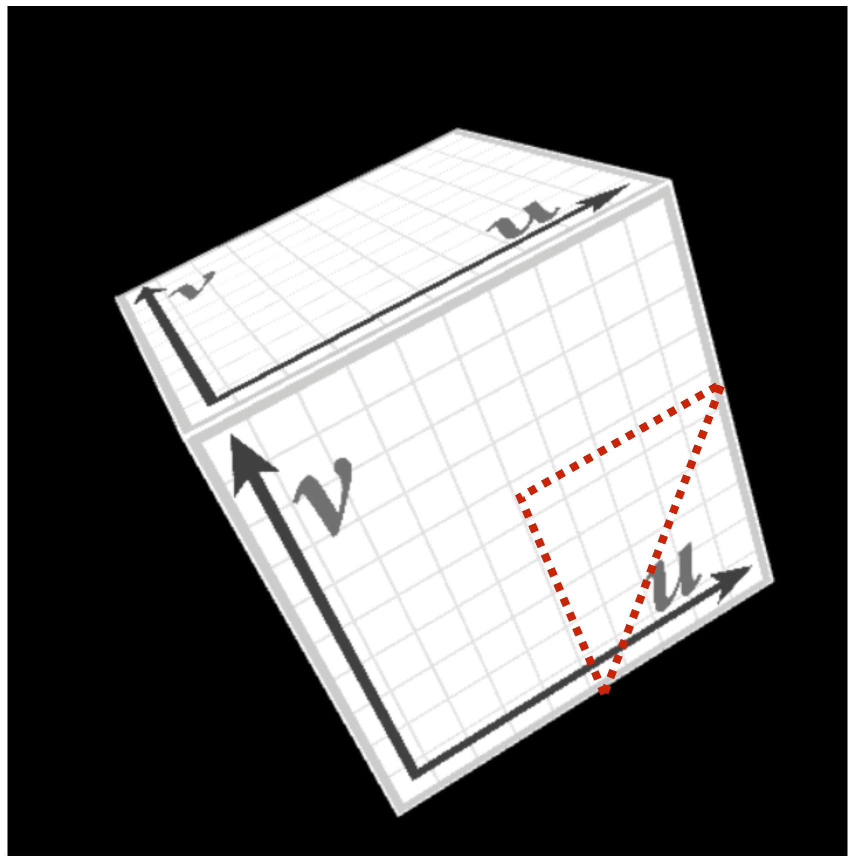
Eight triangles (one face of cube) with surface parameterization provided as per-vertex texture coordinates.



`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.



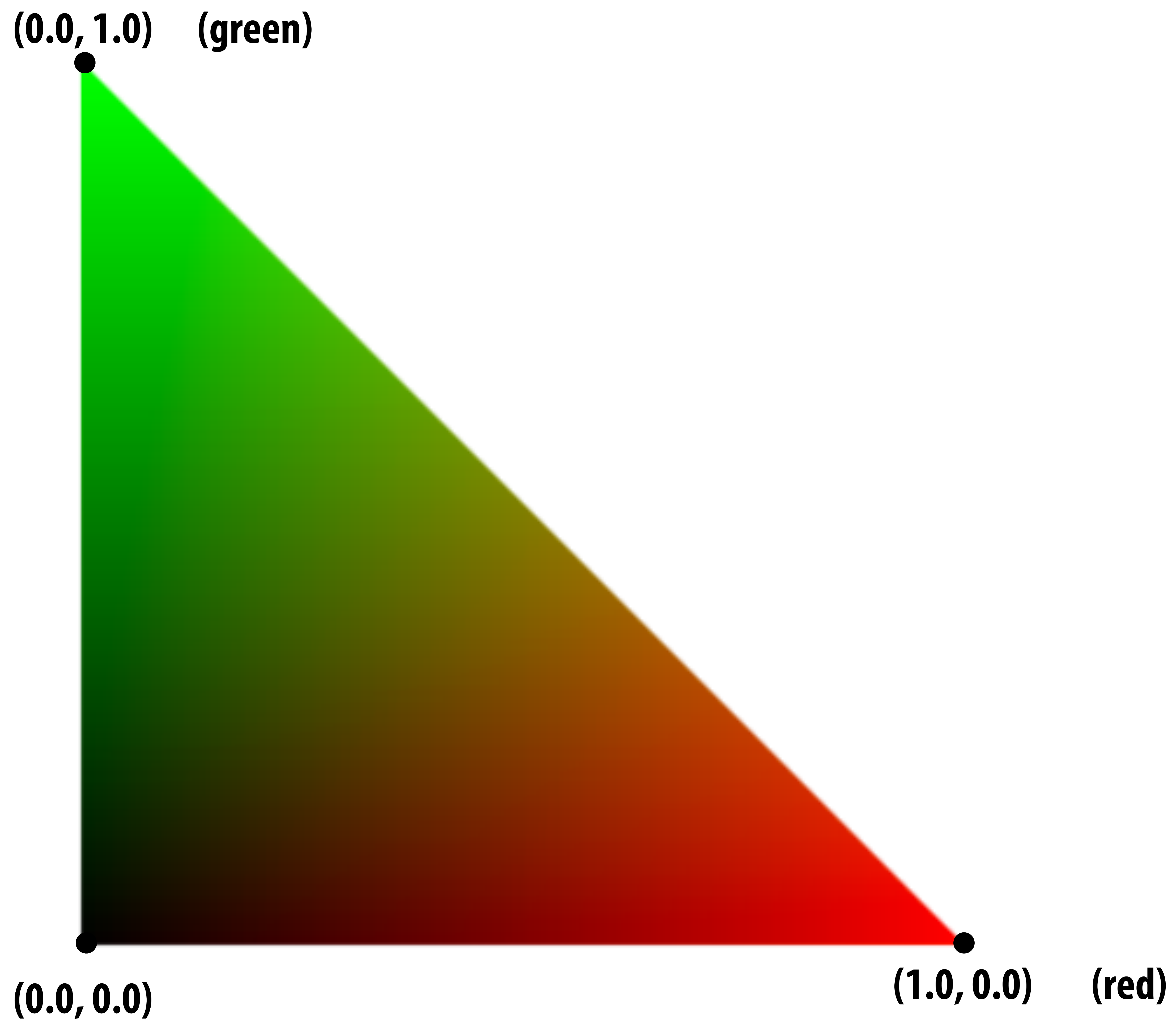
Final rendered result (entire cube shown).

Location of triangle after projection onto screen is 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)

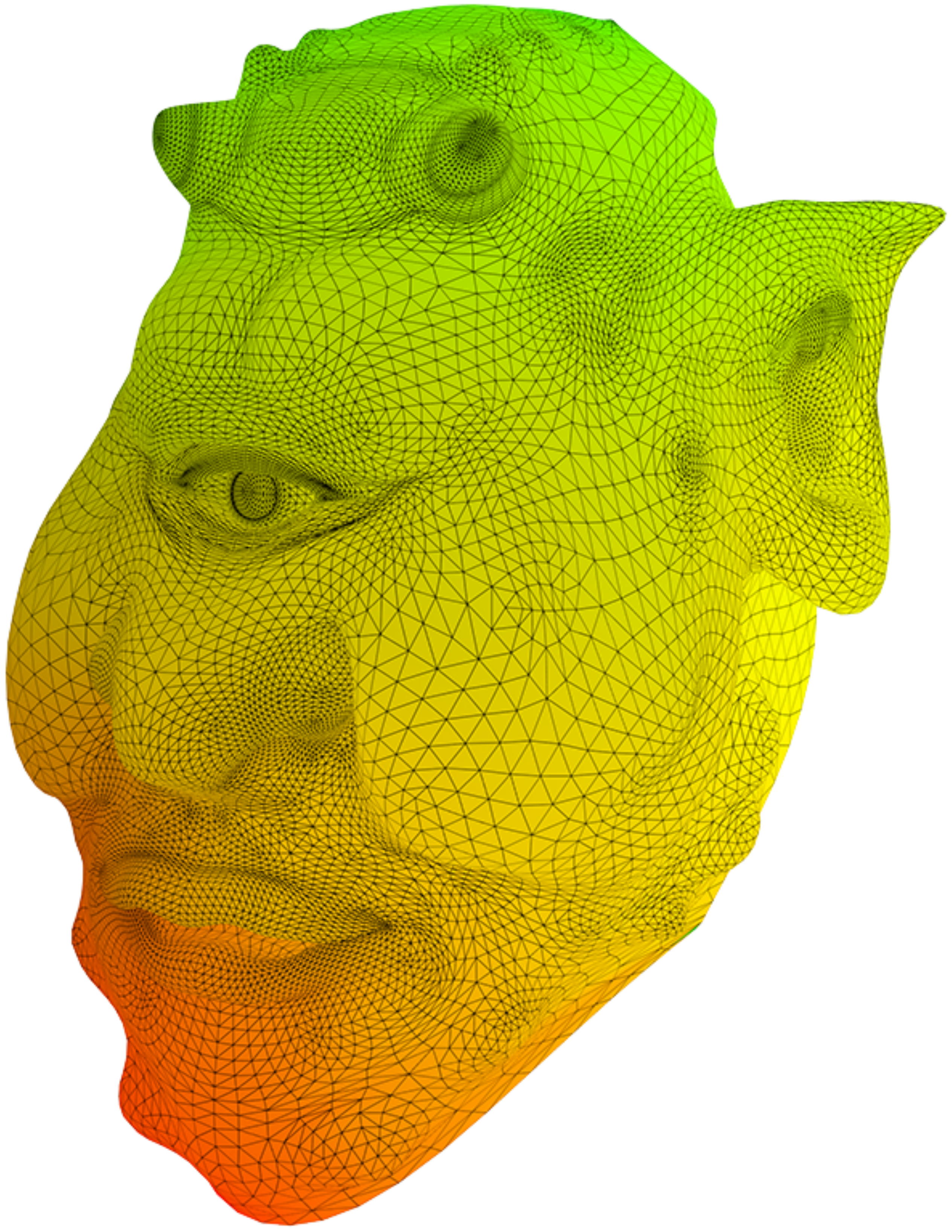
Visualization of texture coordinates

Texture coordinates linearly interpolated over triangle

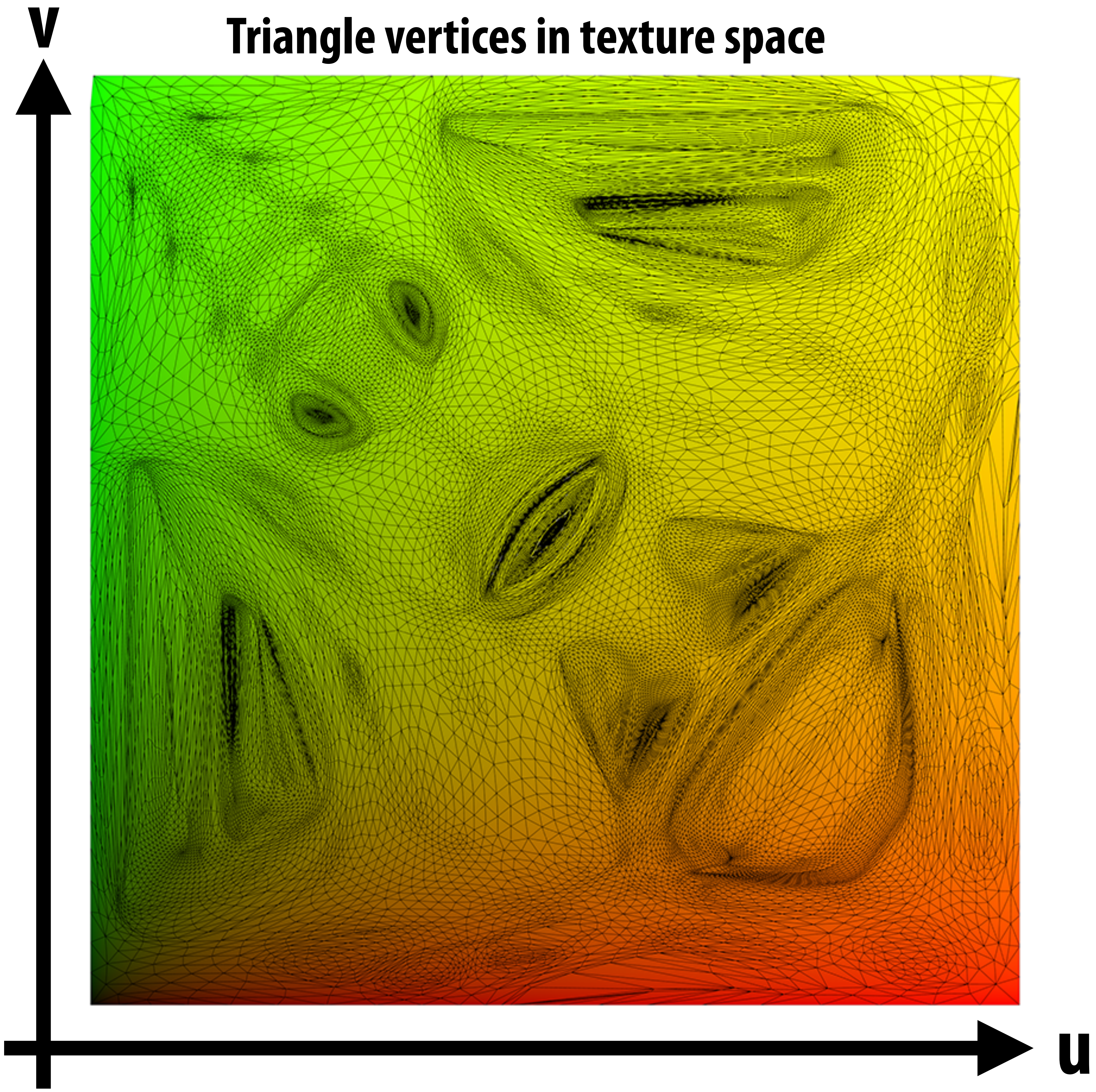


More complex mapping

Visualization of texture coordinates



Triangle vertices in texture space



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

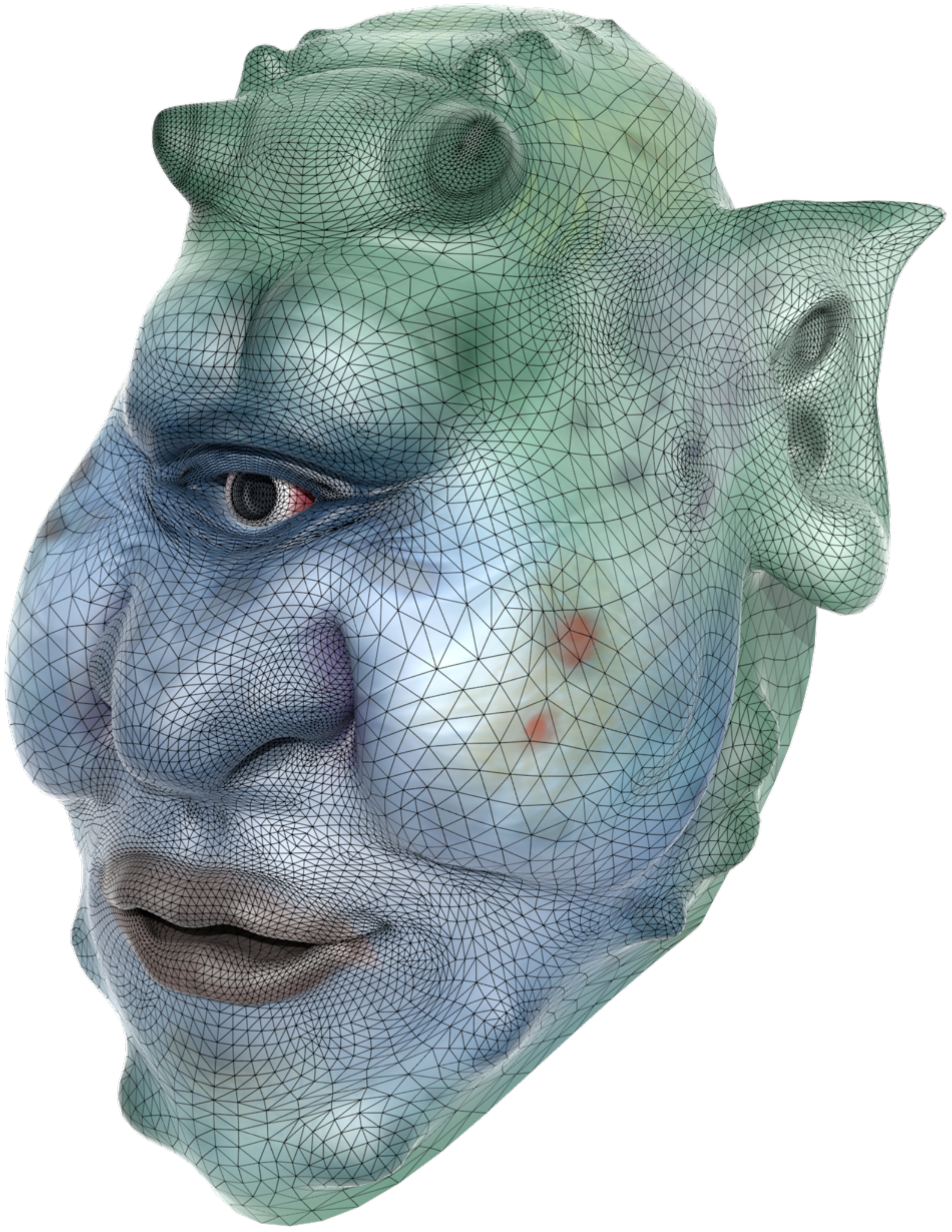
(Coming up with vertex coordinate values is topic of a graphics class)

Simple texture mapping operation

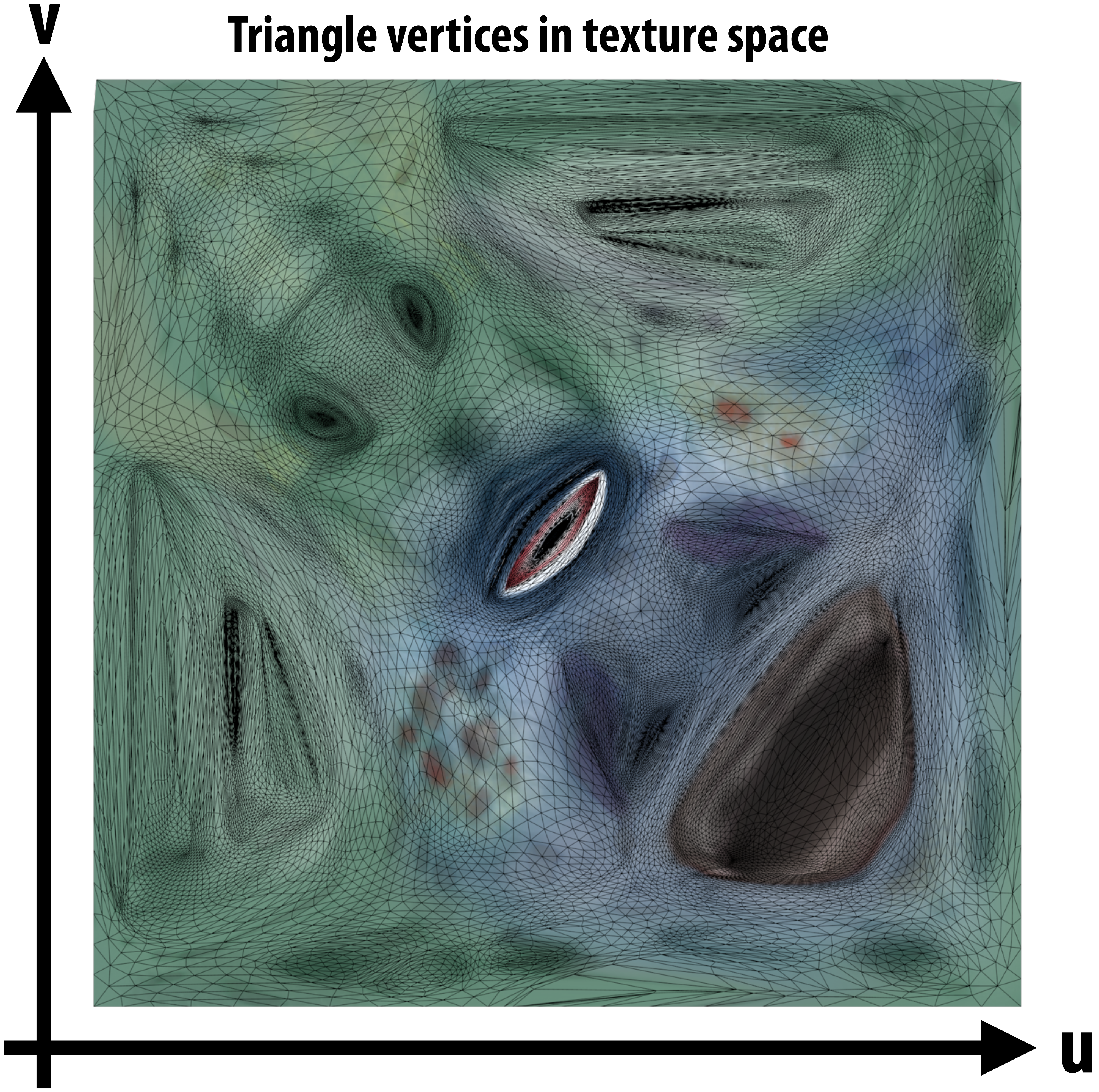
```
for each fragment (x,y) in fragment stream:

   // interpolate per-vertex coordinates
   (u,v) = evaluate texcoord value of surface at screen point (x,y);

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

Texture mapping adds detail

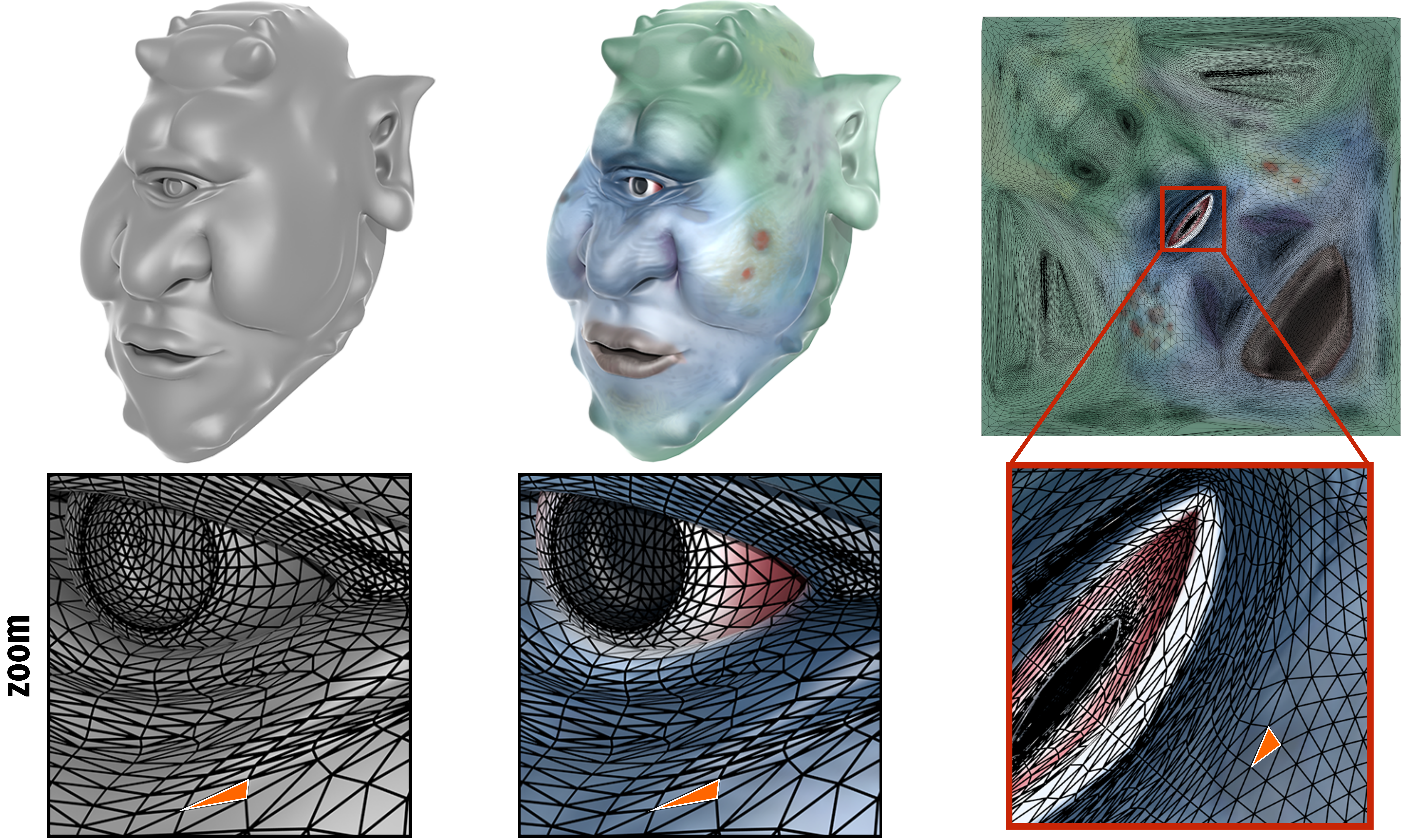
Rendered result



Triangle vertices in texture space



Texture mapping adds detail



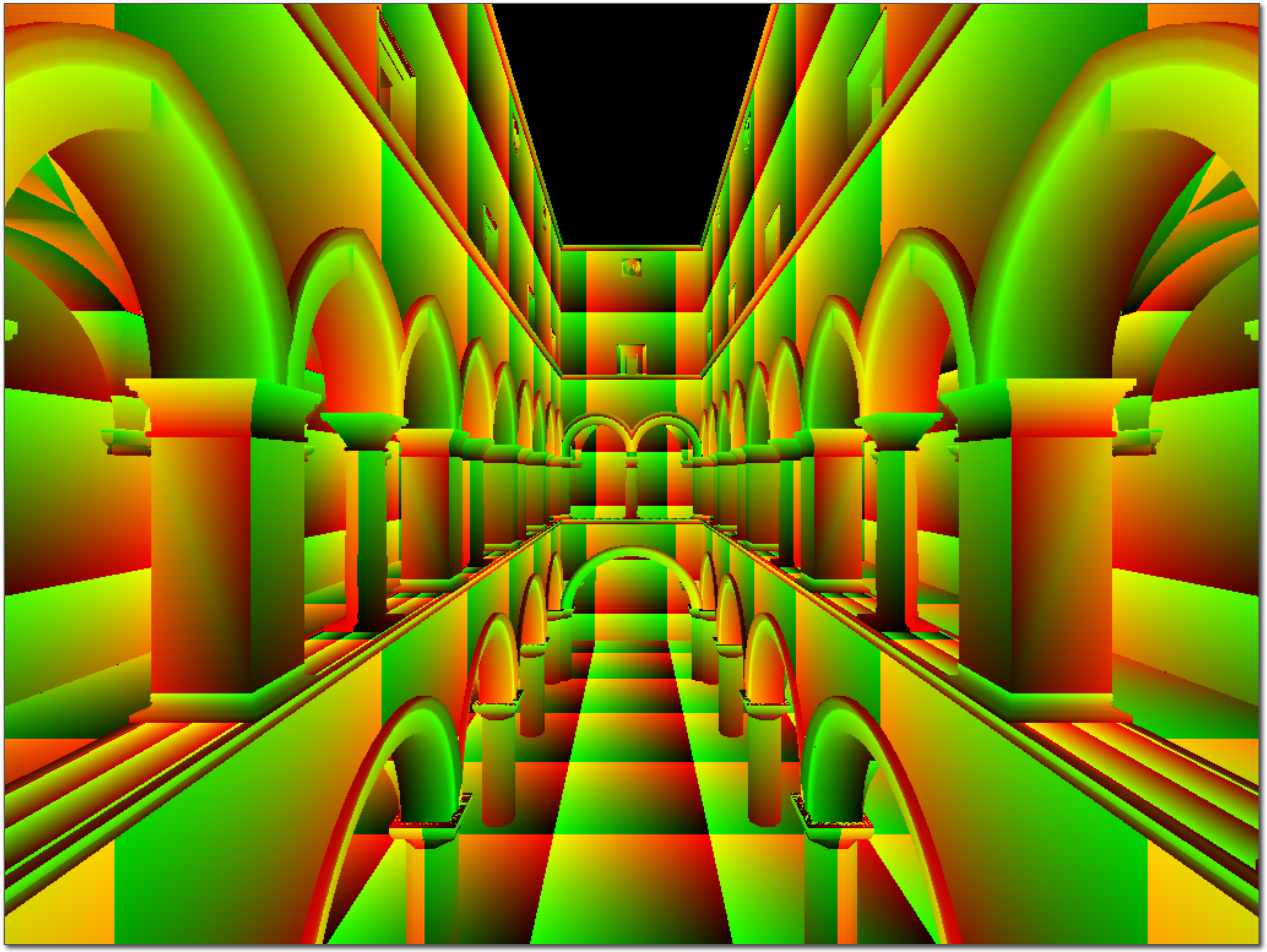
rendering without texture

rendering with texture

texture image

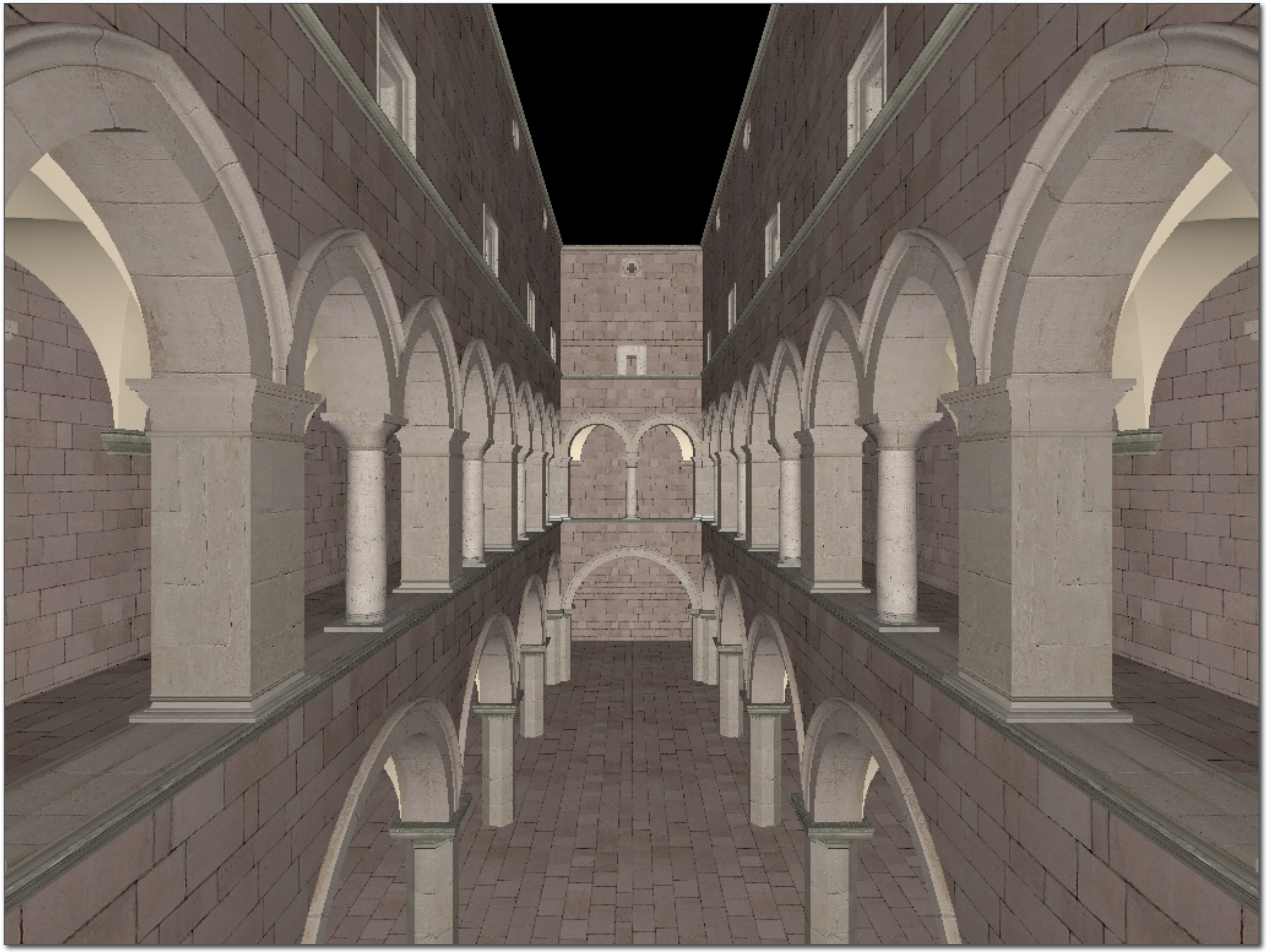
zoom

Another example: Sponza



Notice texture coordinates repeat over surface.

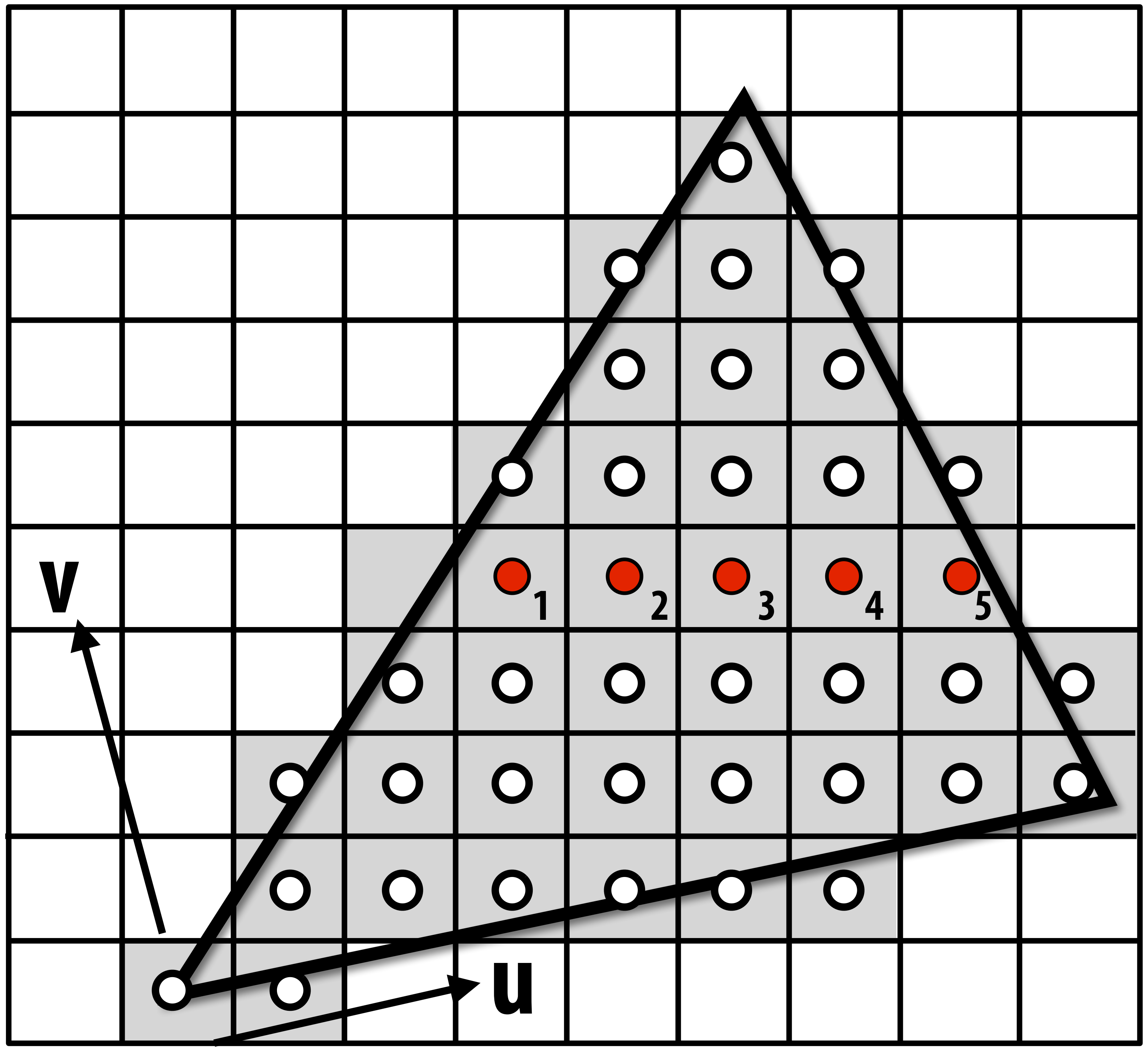
Textured Sponza



Example texture images used in Sponza

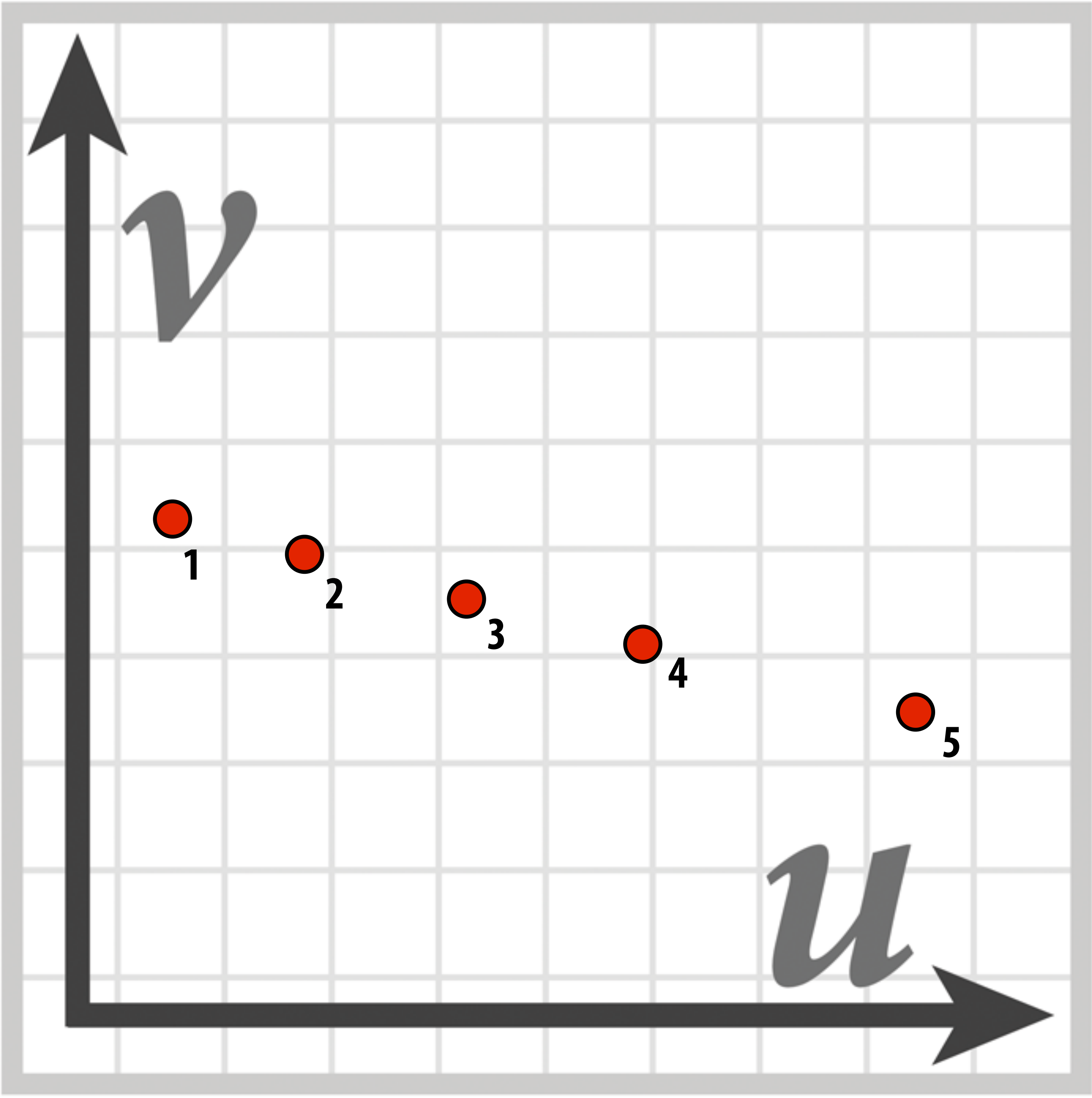
Texture space samples

Sample positions in XY screen space



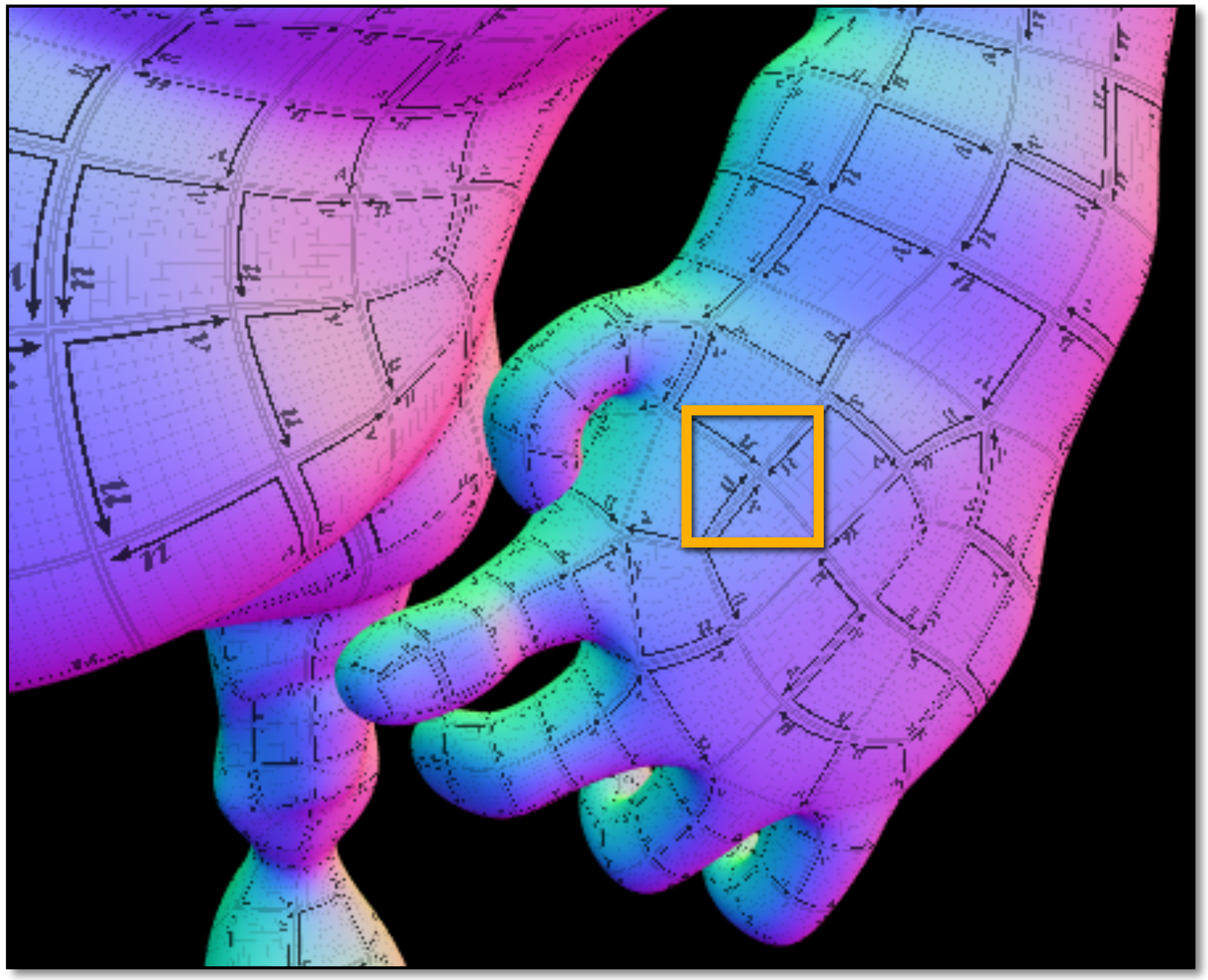
Sample positions are uniformly distributed in screen space (rasterizer samples triangle's appearance at these locations)

Sample positions in texture space

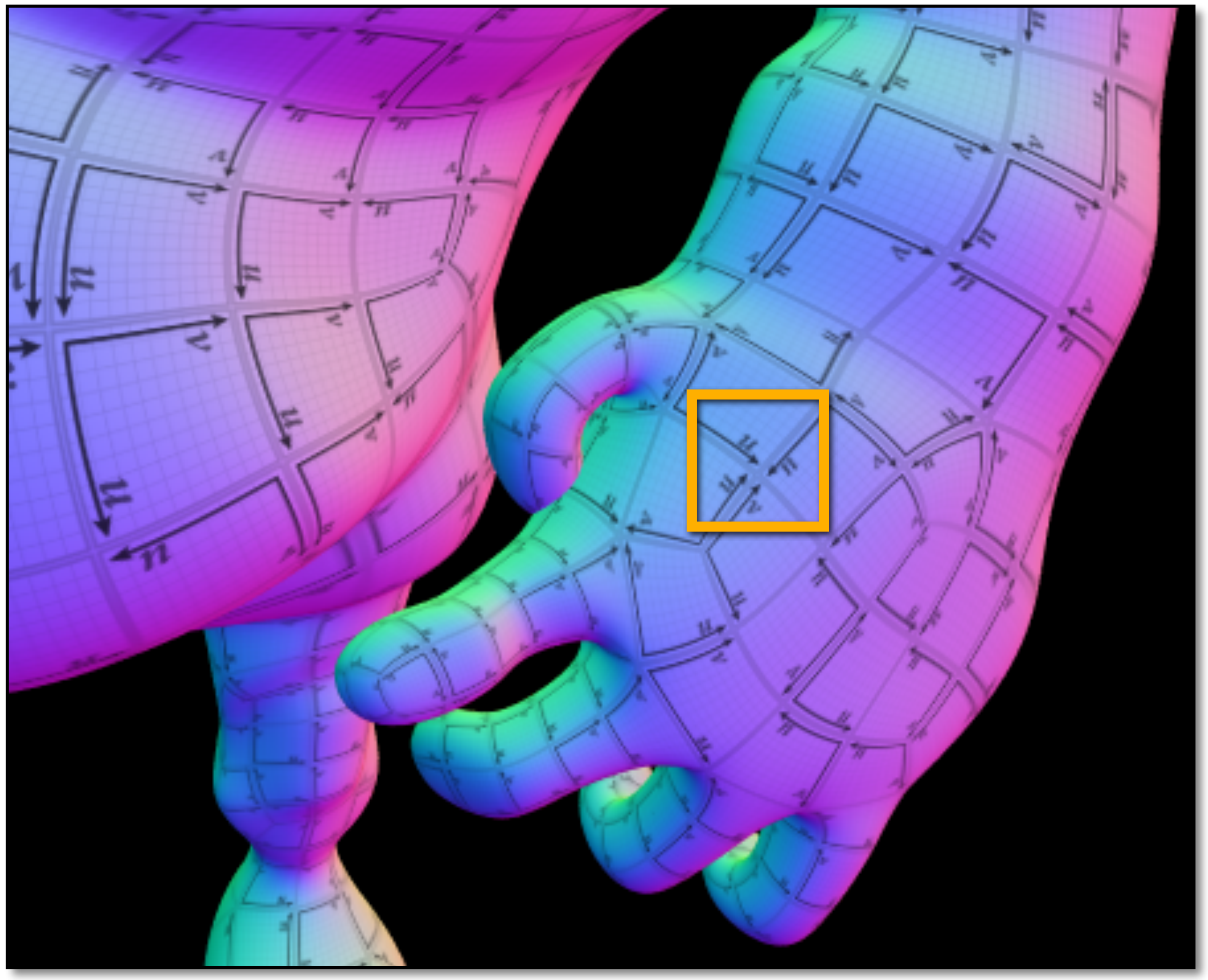


Texture sample positions in texture space (texture function is sampled at these locations)

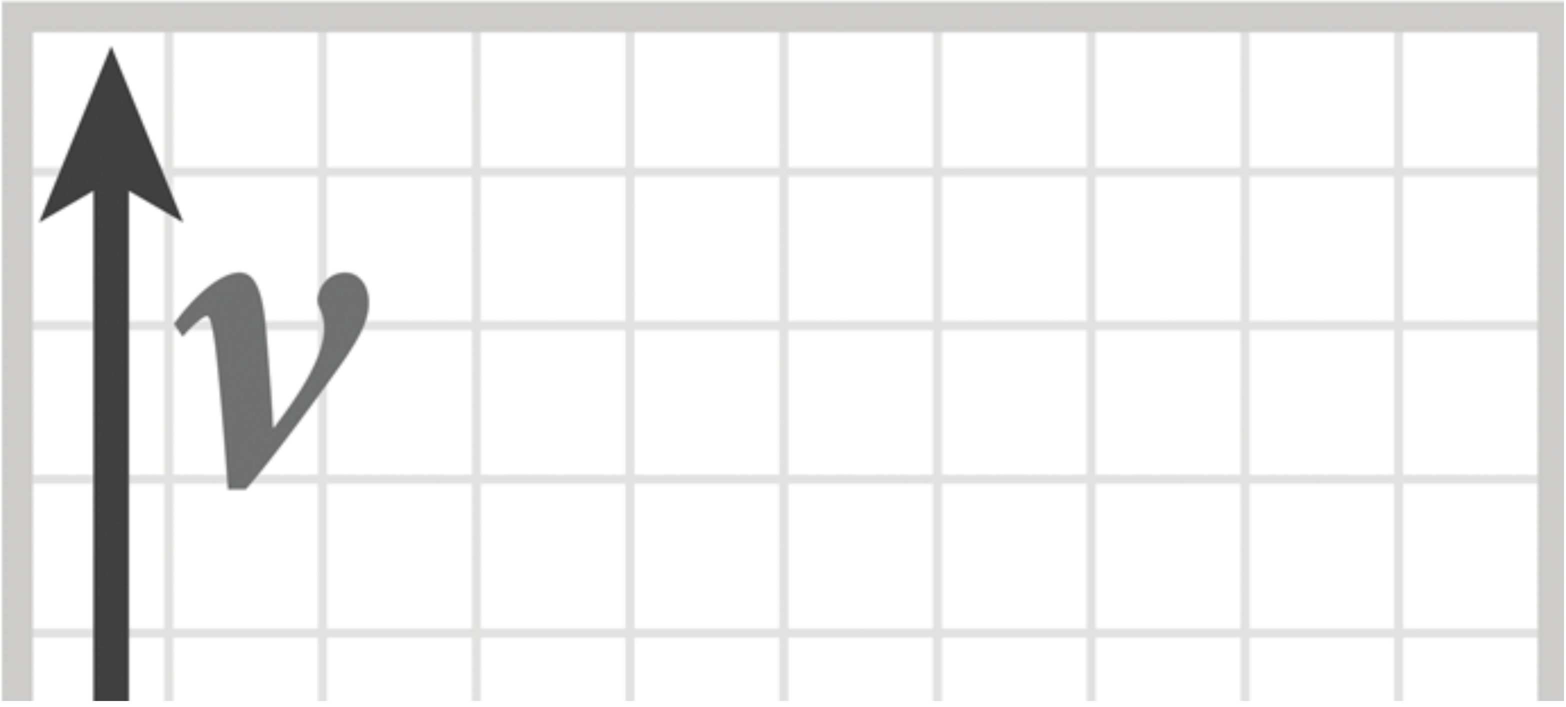
Aliasing due to undersampling texture



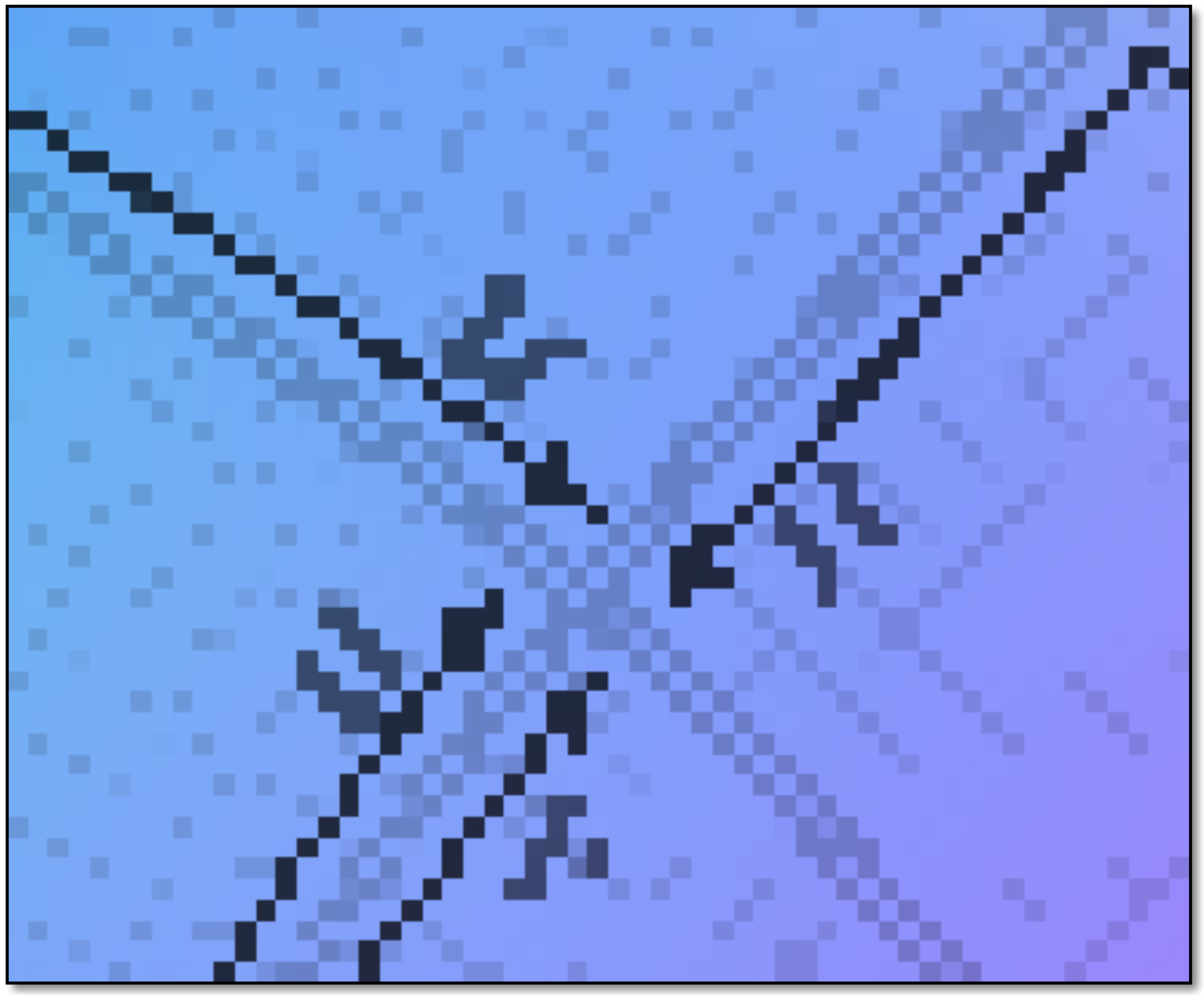
No pre-filtering of texture data
(resulting image exhibits aliasing)



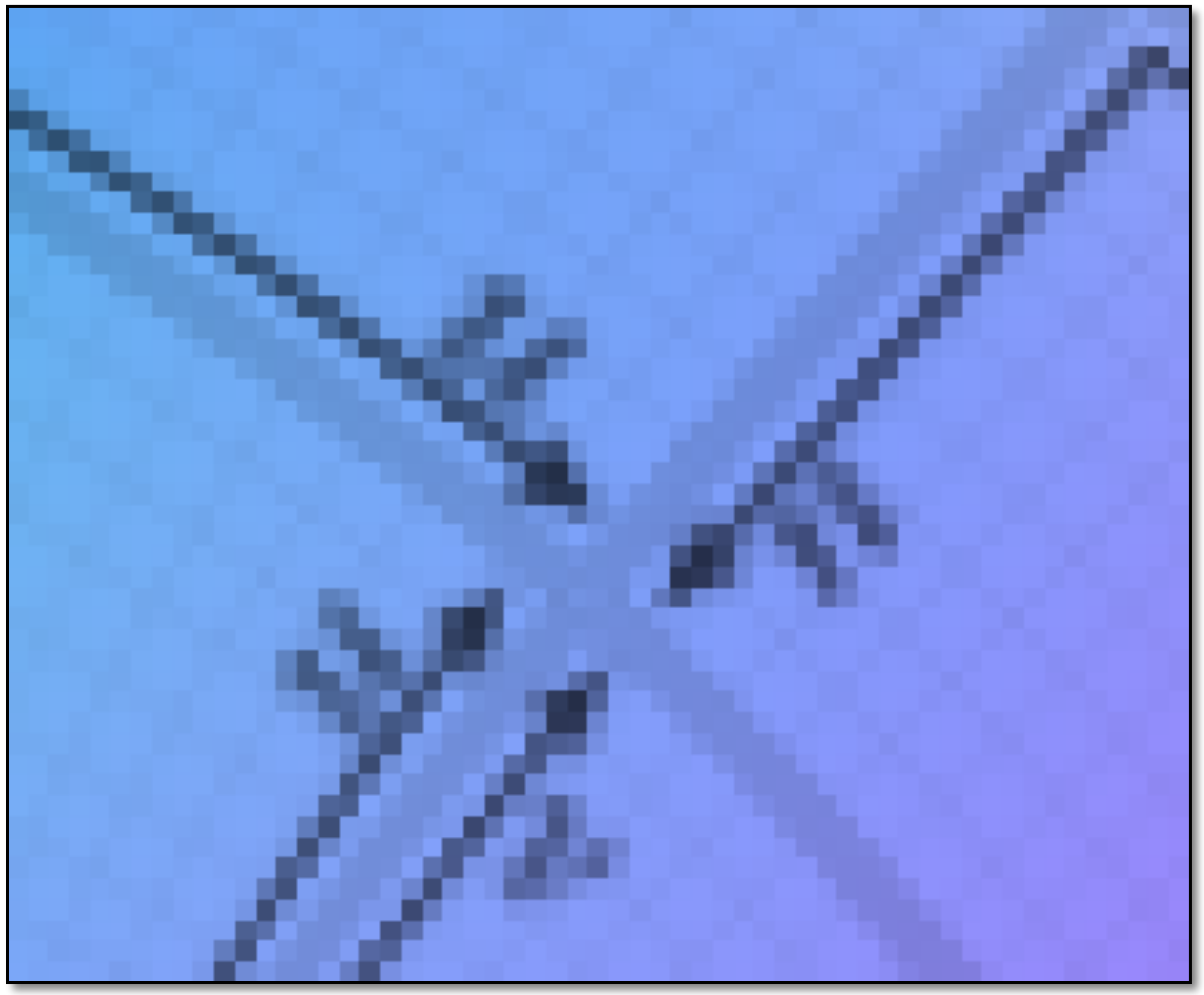
Rendering using pre-filtered texture data



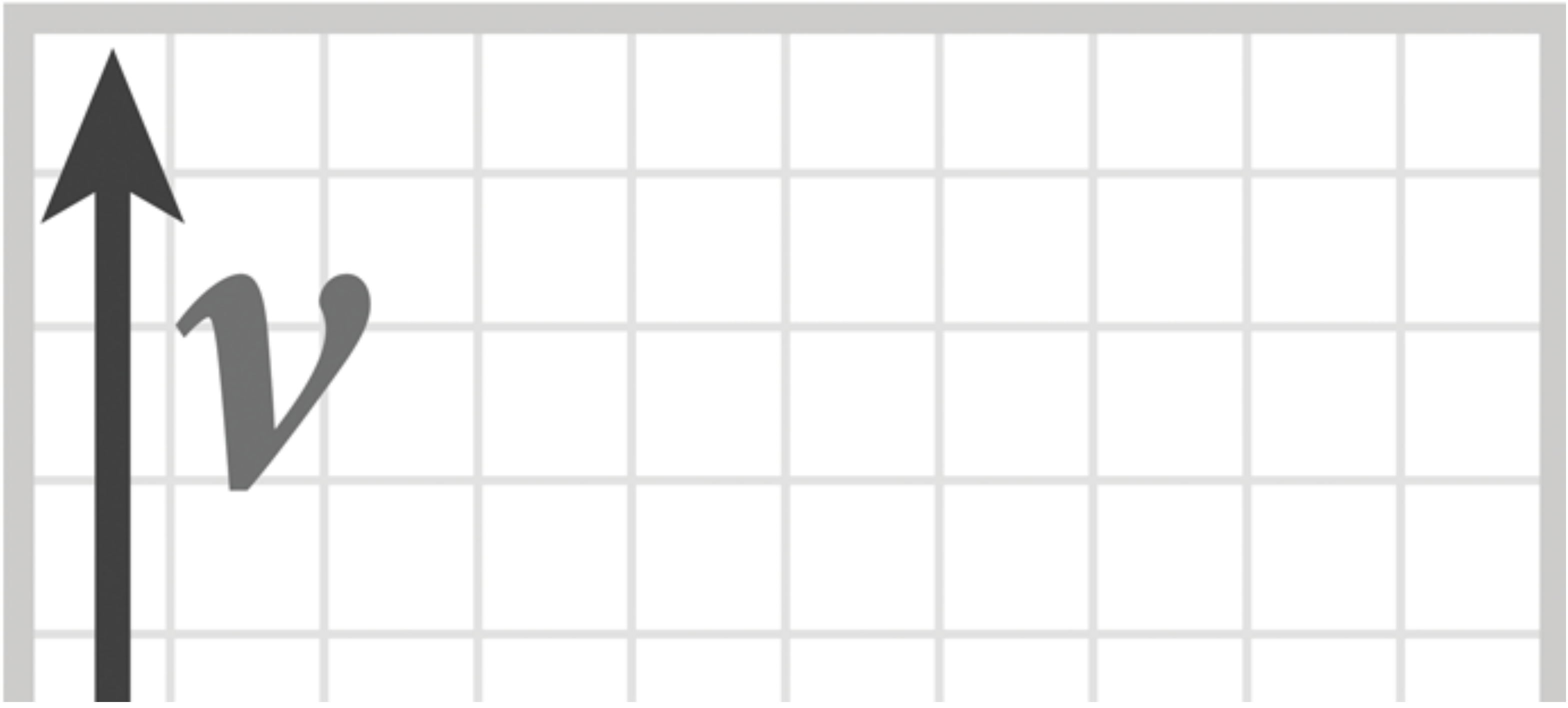
Aliasing due to undersampling (zoom)



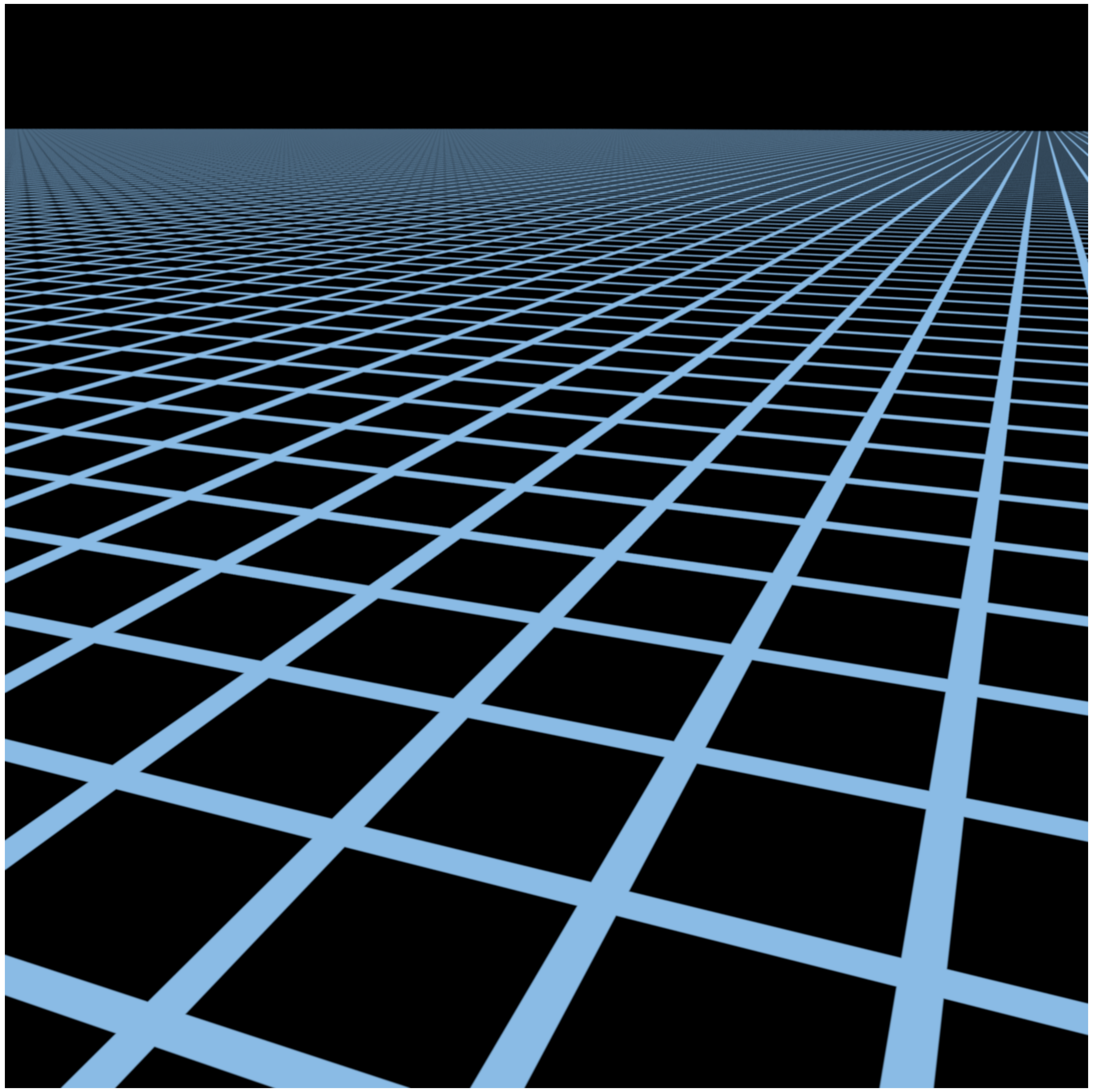
No pre-filtering of texture data
(resulting image exhibits aliasing)



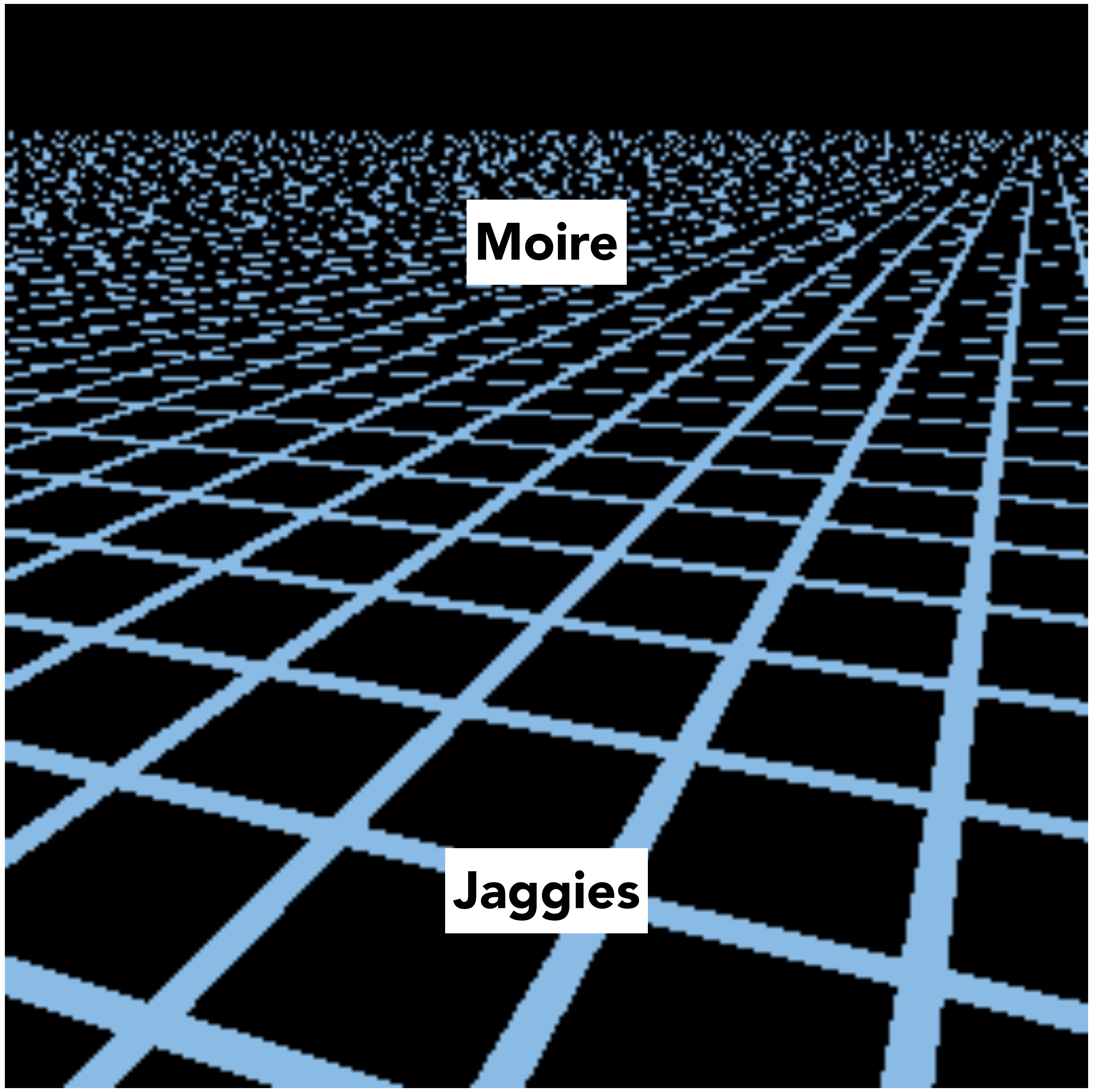
Rendering using pre-filtered texture data



Another example:



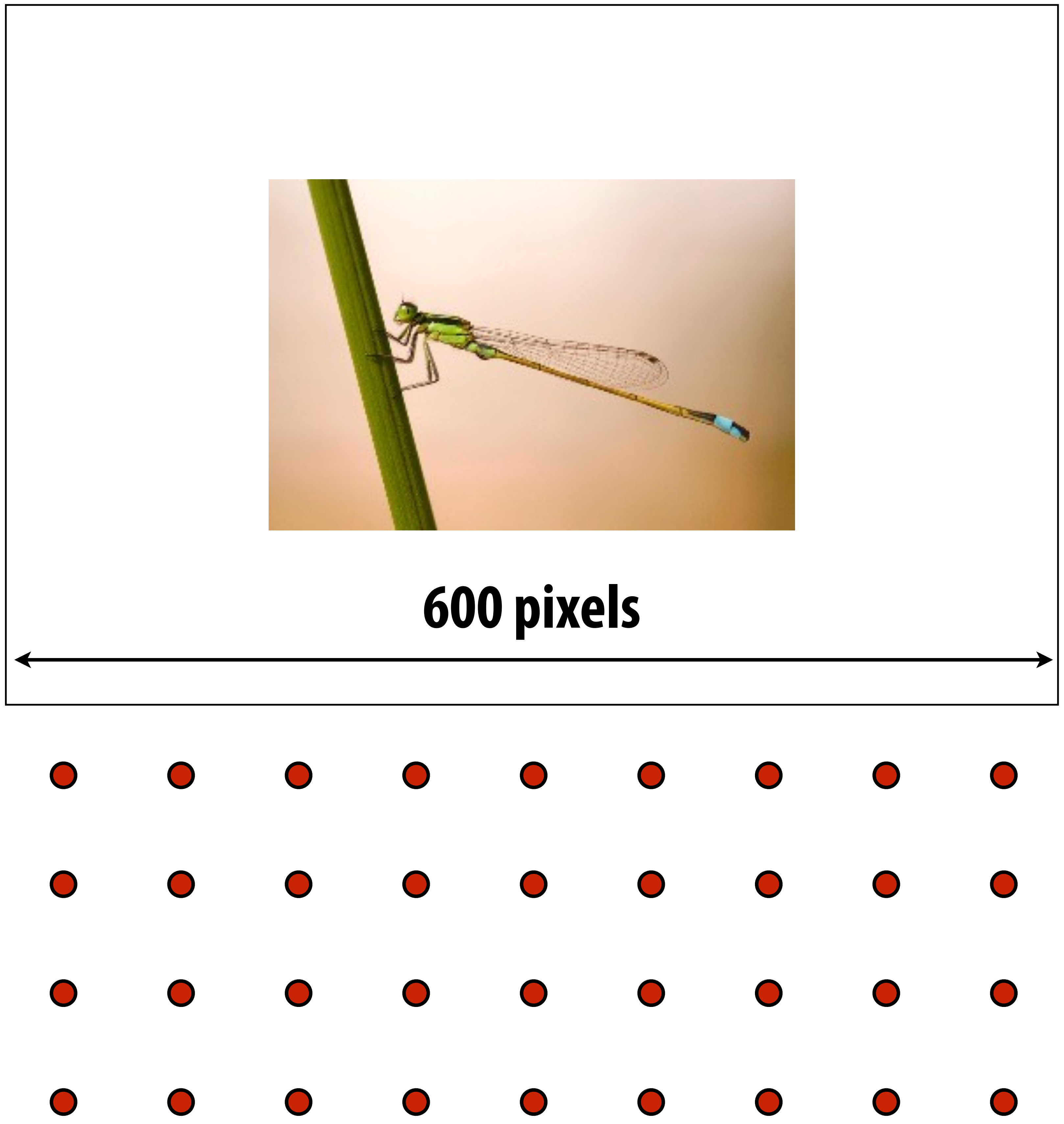
Source image: 1280x1280 pixels



Rendered image: 256x256 pixels

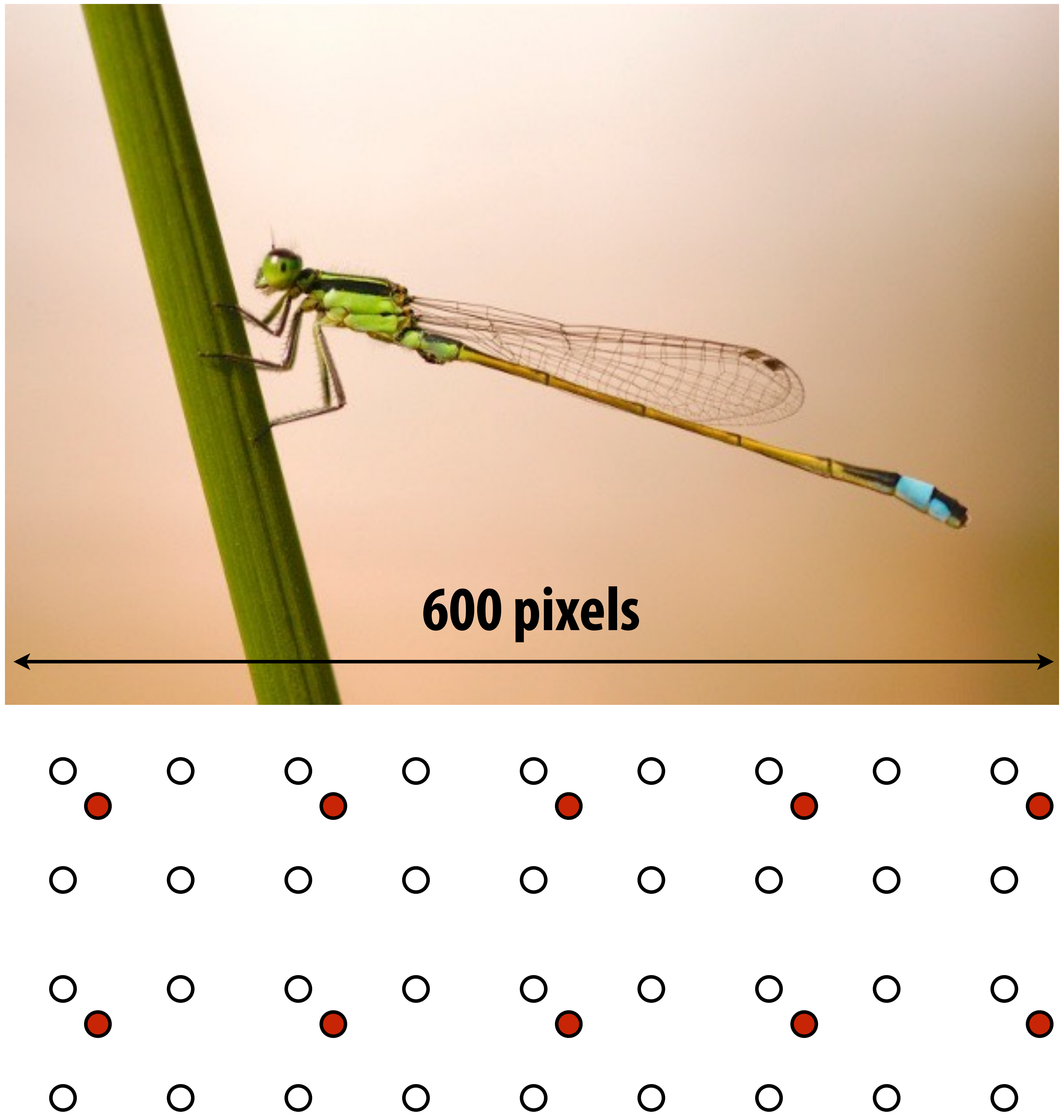
Sampling rate on screen vs texture

Rendered image (object zoomed out)



Screen space (x,y)

Texture Image



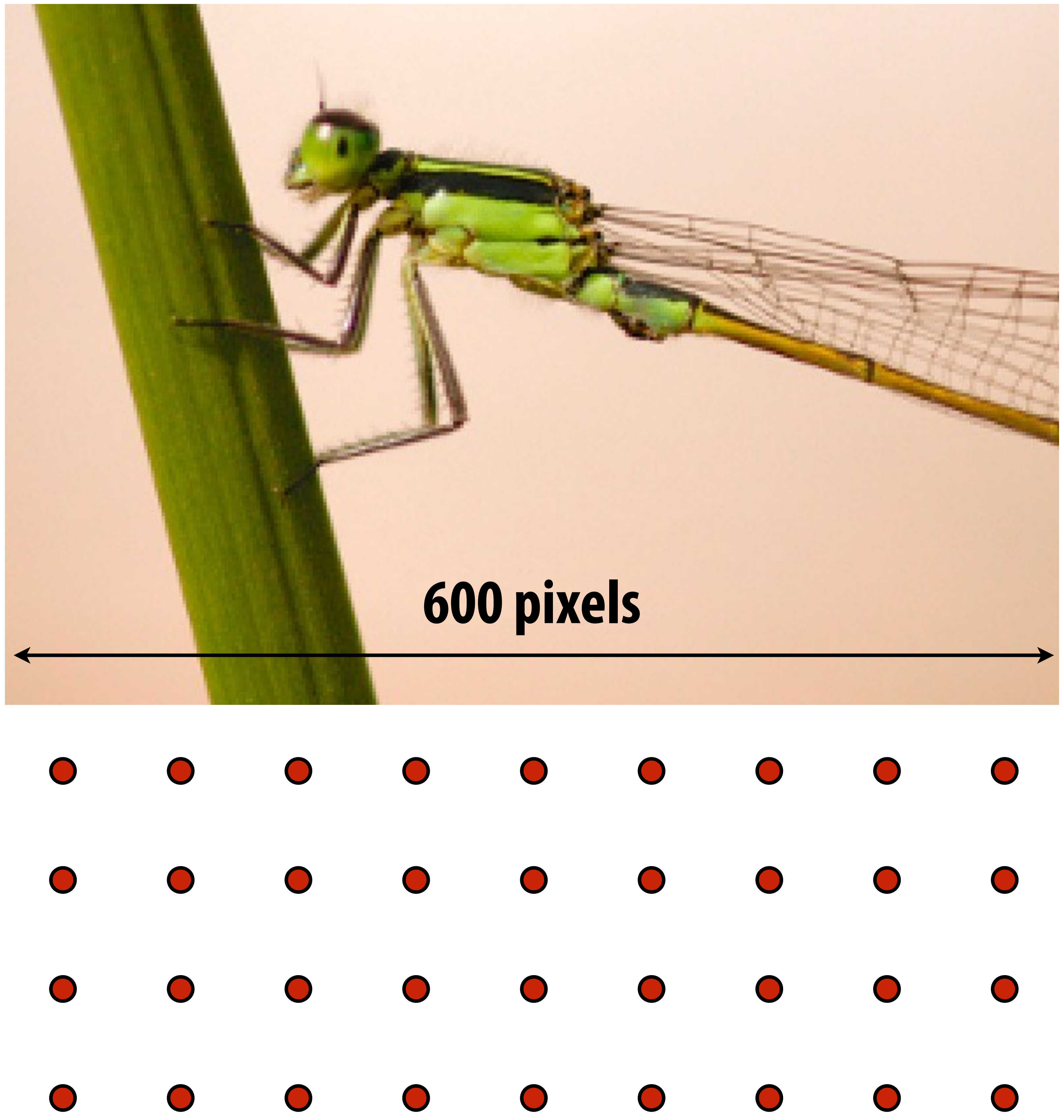
Texture space (u,v)

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

Texture is "minified"

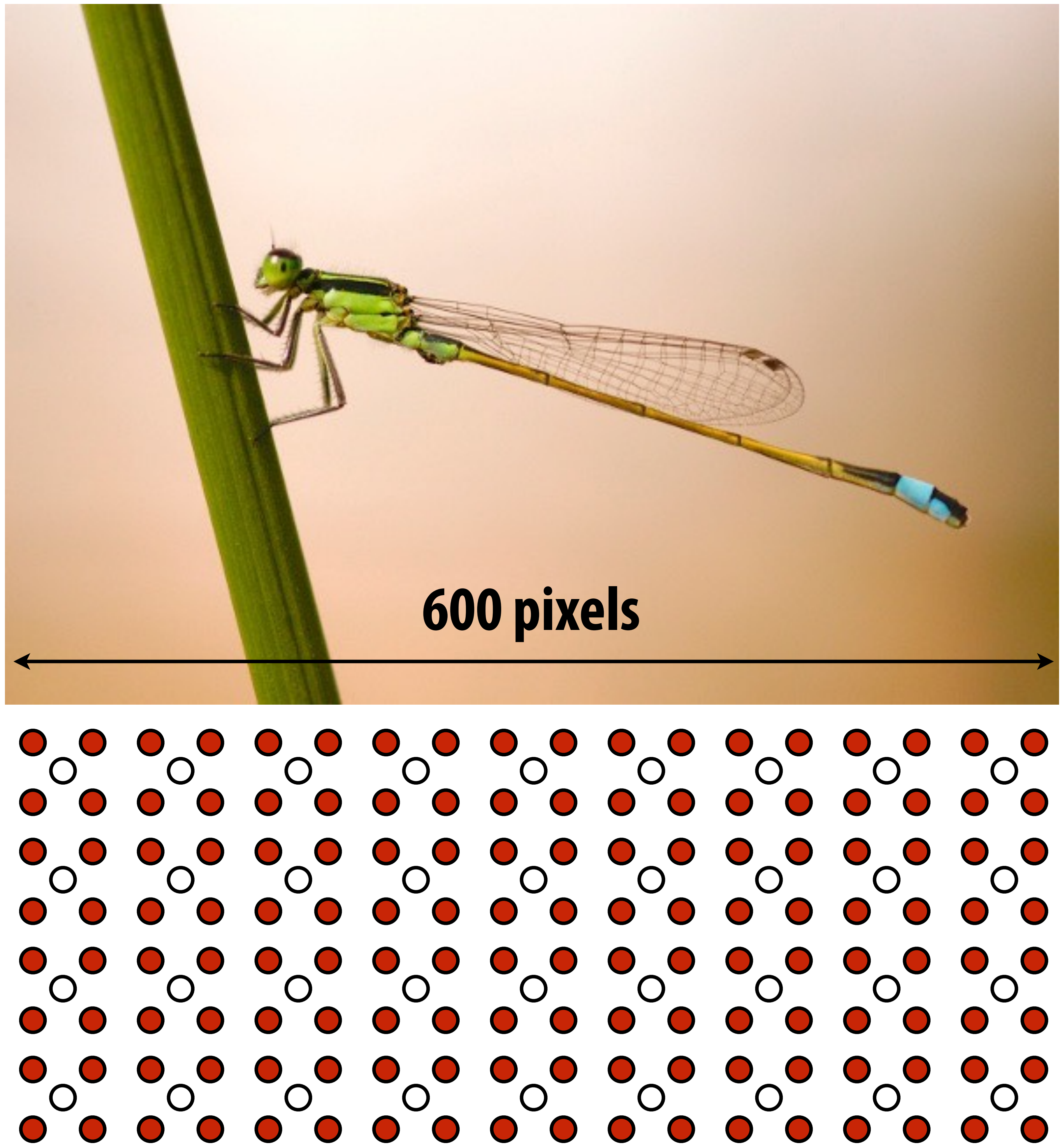
Sampling rate on screen vs texture

Rendered image (zoomed in)



Screen space (x,y)

Texture Image

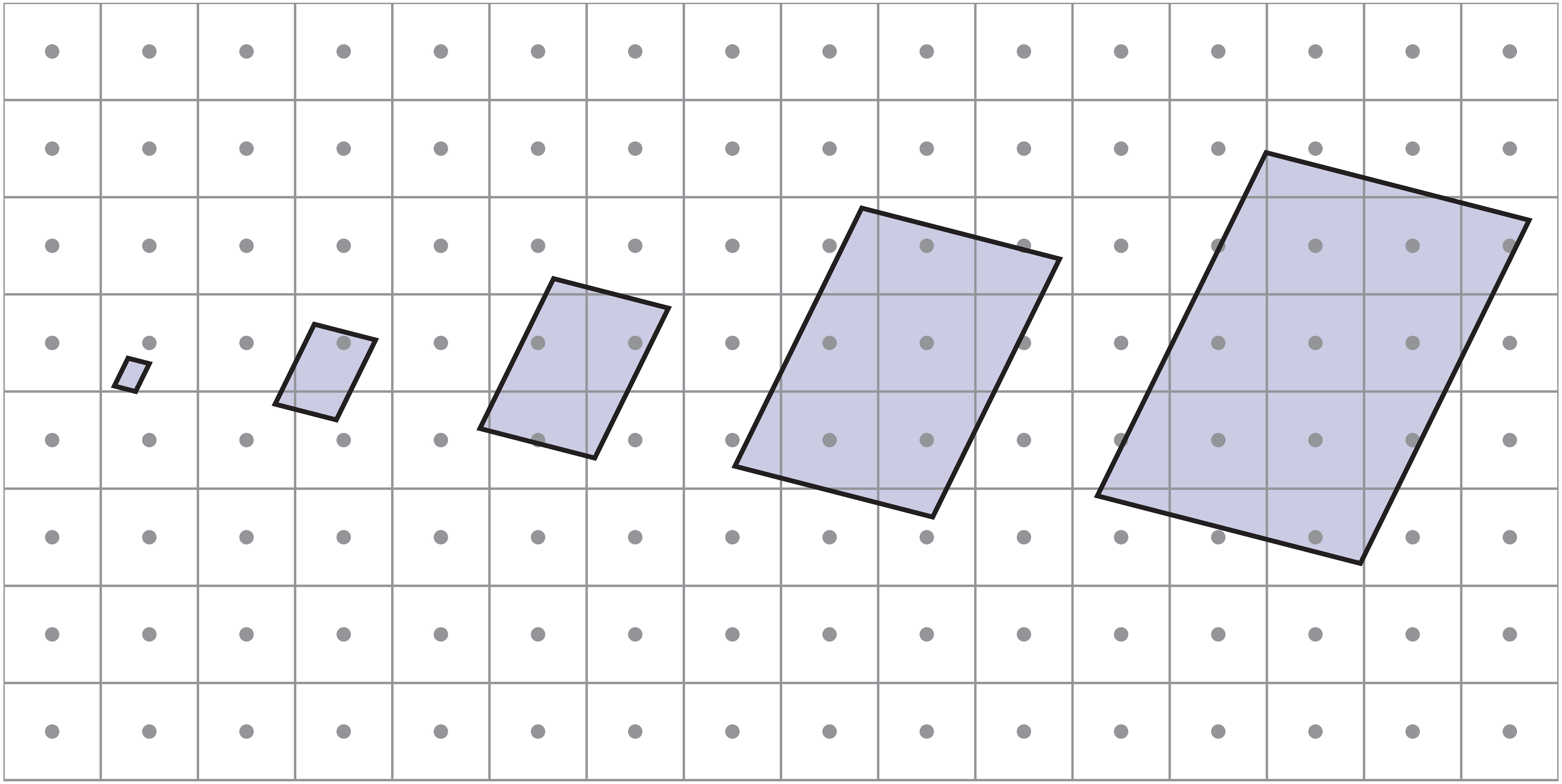


Texture space (u,v)

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

Texture is "magnified"

Screen pixel footprint in texture space



Upsampling
(Magnification)

⟷

Downsampling
(Minification)

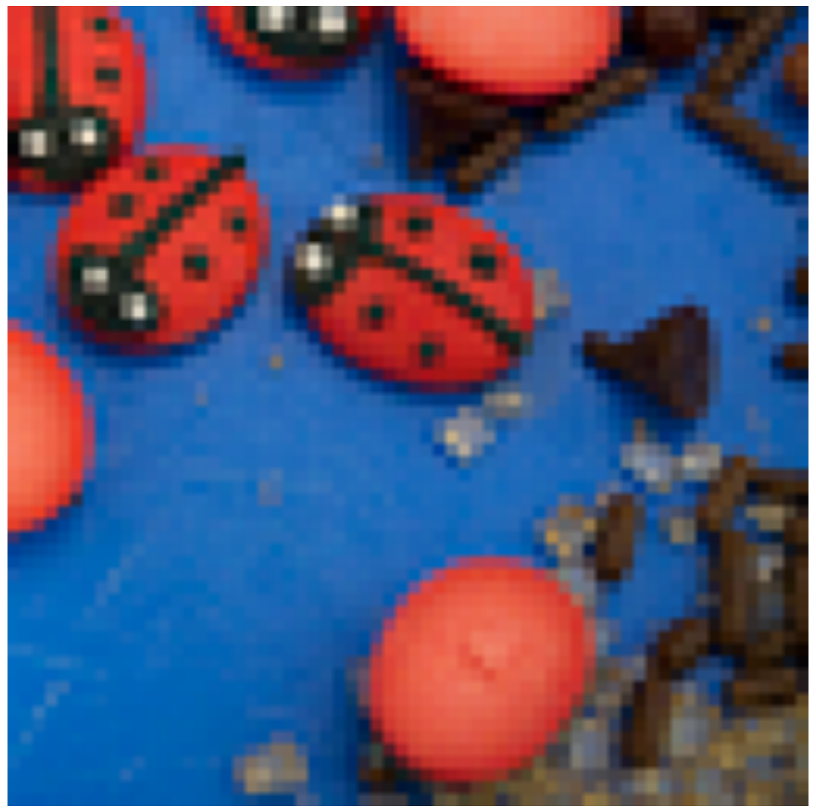
Screen pixel area vs texel area

- At optimal viewing size:
  - 1:1 mapping between pixel sampling rate and texel sampling rate
  - Dependent on screen and texture resolution! e.g. 512x512
- When larger (magnification)
  - Multiple pixel samples per texel sample
- When smaller (minification)
  - One pixel sample per multiple texel samples

Mipmap (L. Williams 83)

(You have seen this earlier in class as a Gaussian pyramid)

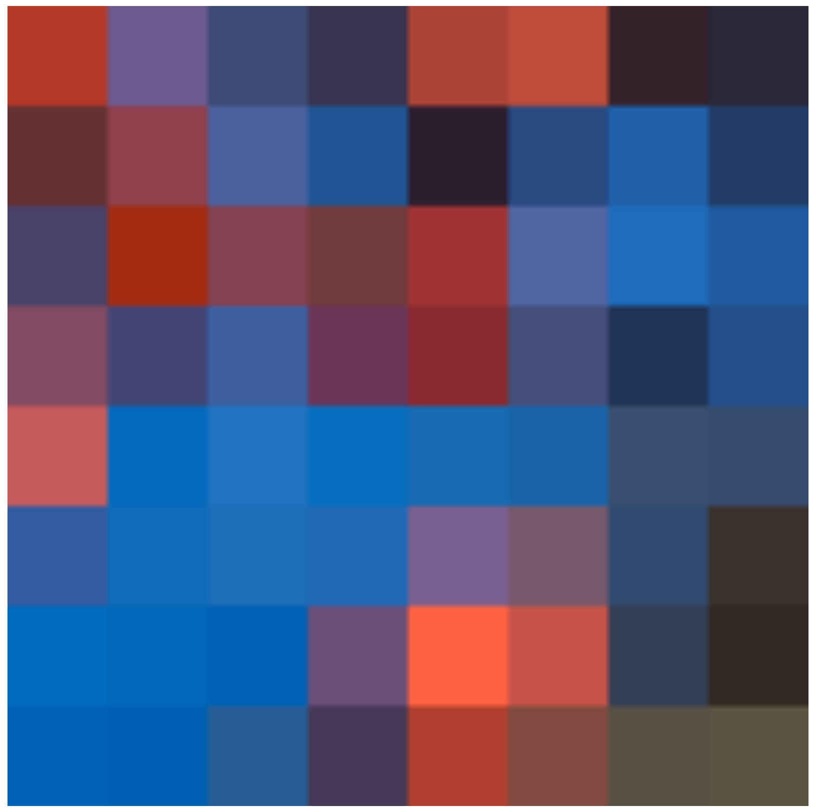
Level 0 = 128x128



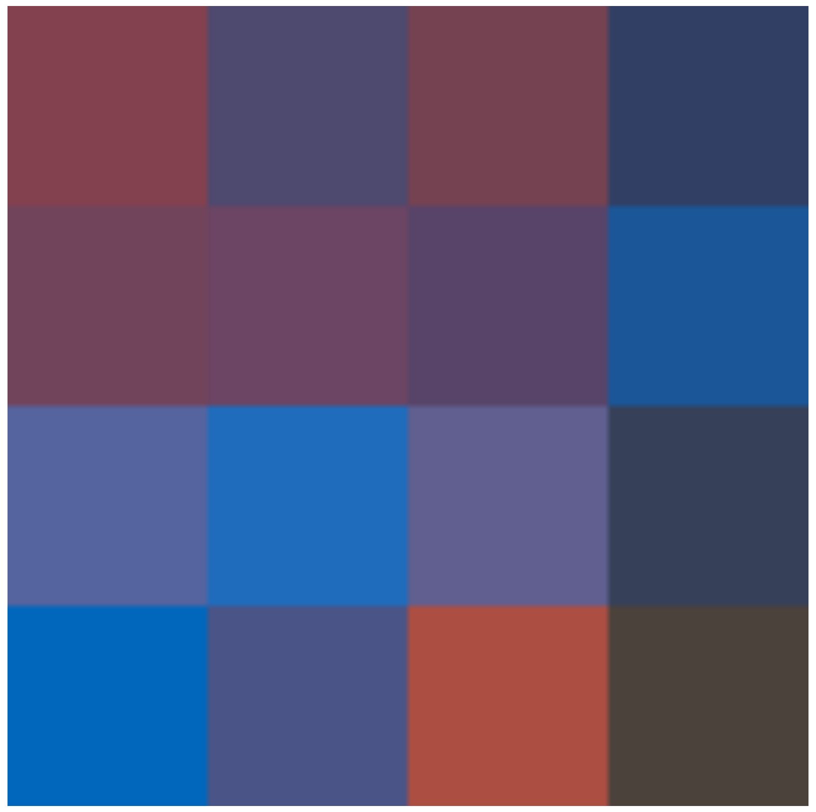
Level 1 = 64x64

Level 2 = 32x32

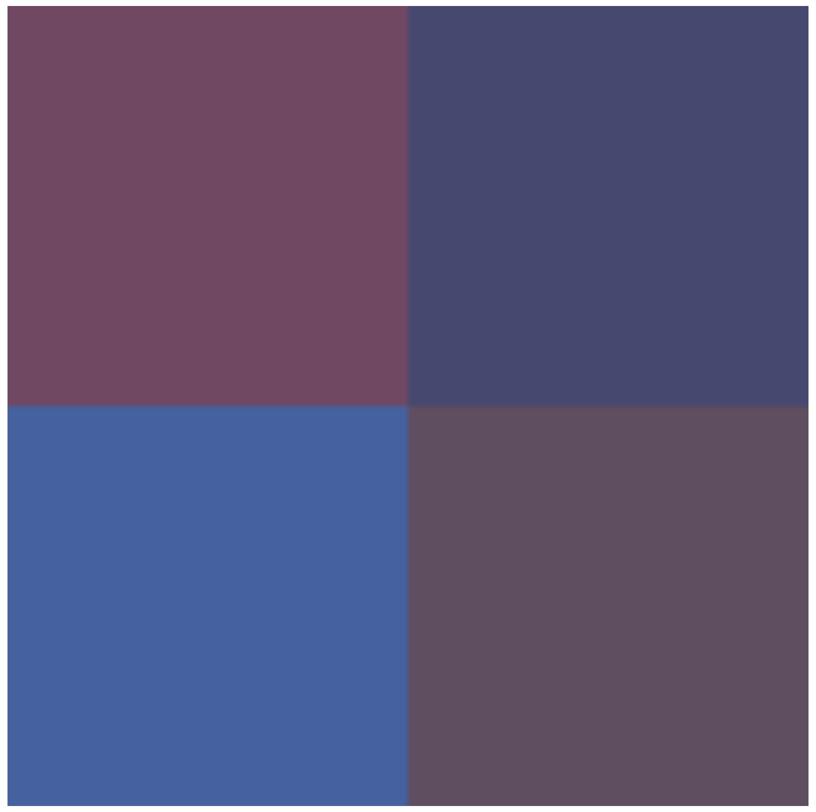
Level 3 = 16x16



Level 4 = 8x8



Level 5 = 4x4

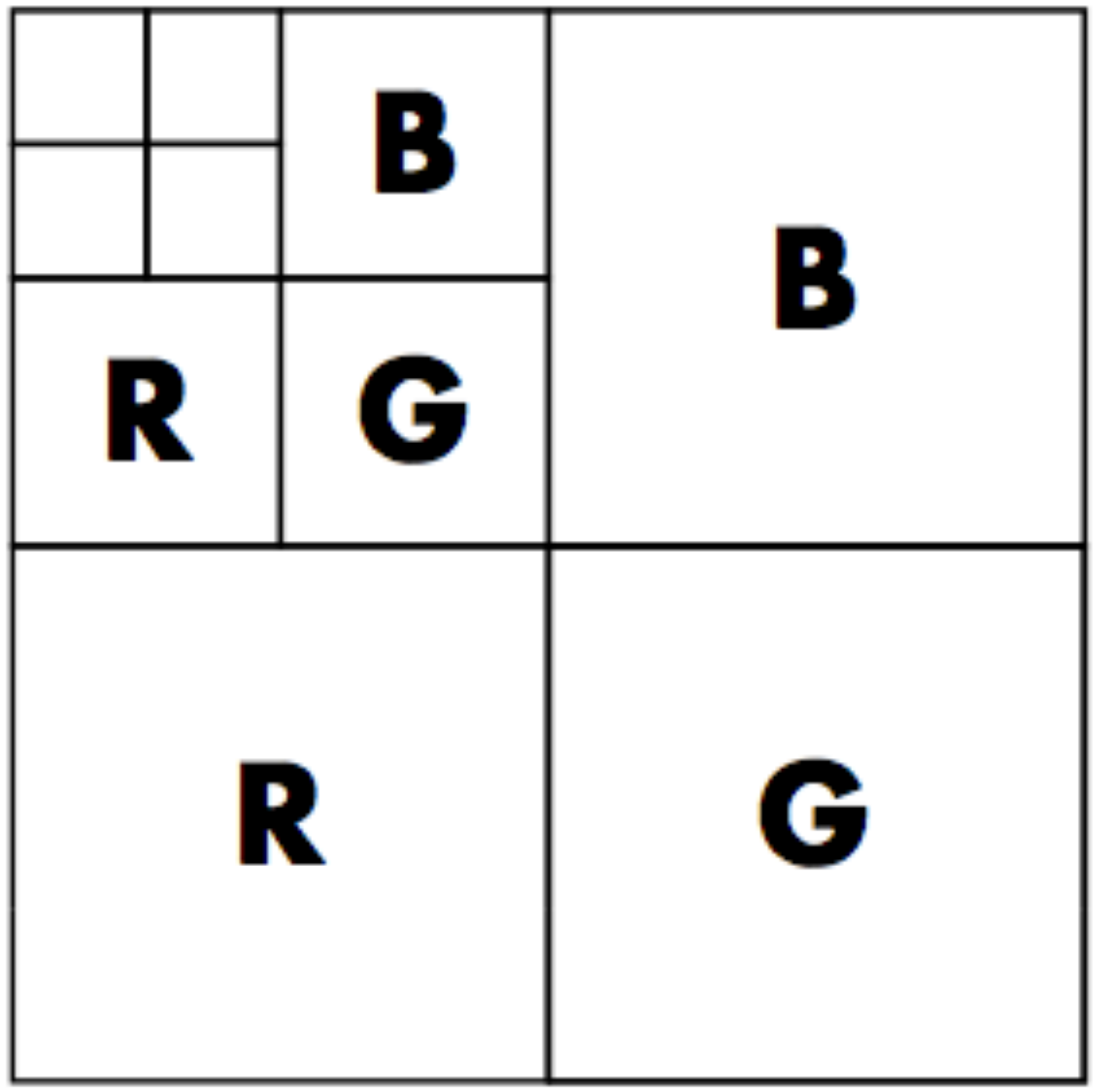


Level 6 = 2x2

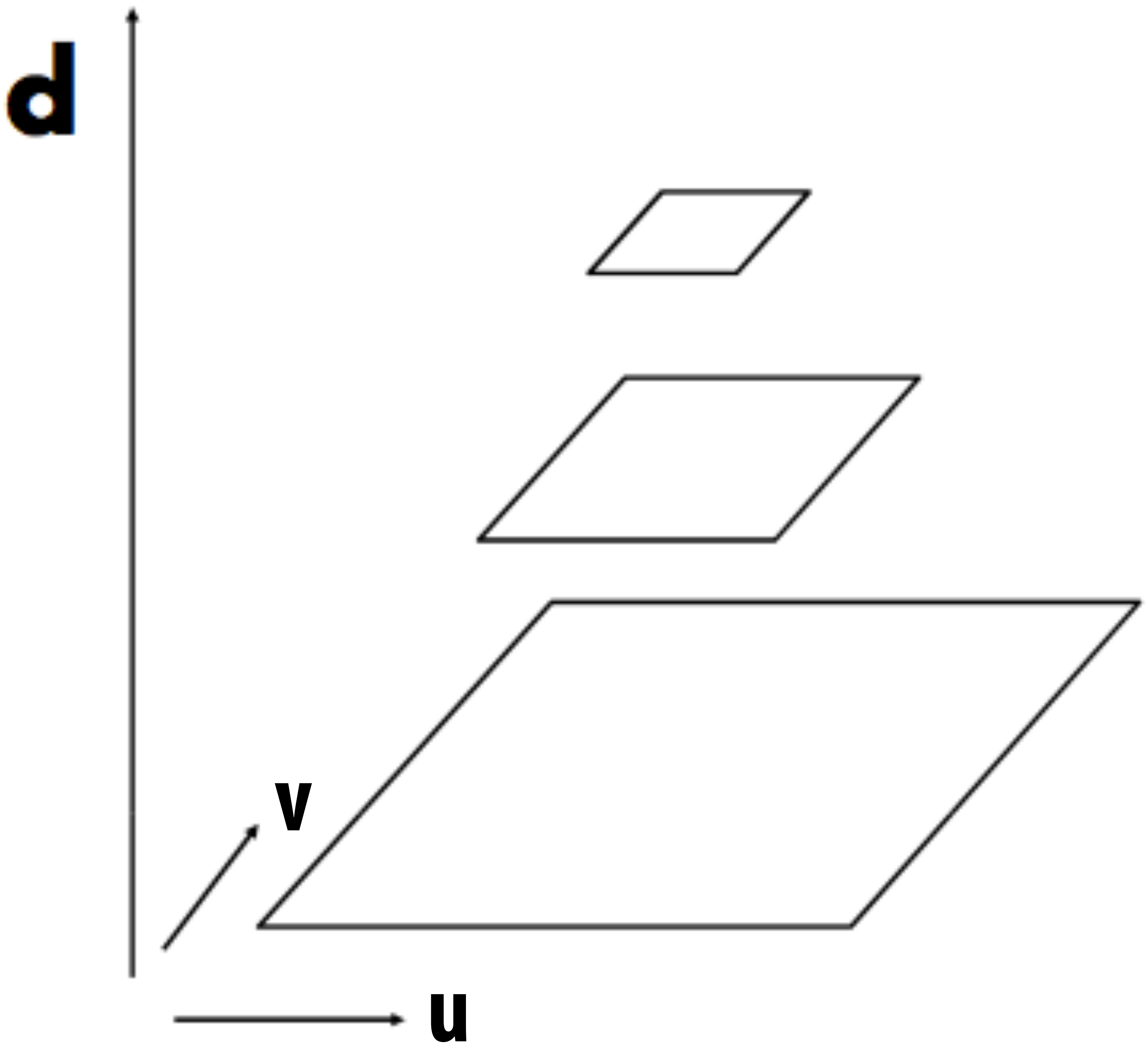
Level 7 = 1x1

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

Mipmap (L. Williams 83)



Williams' original proposed mip-map layout

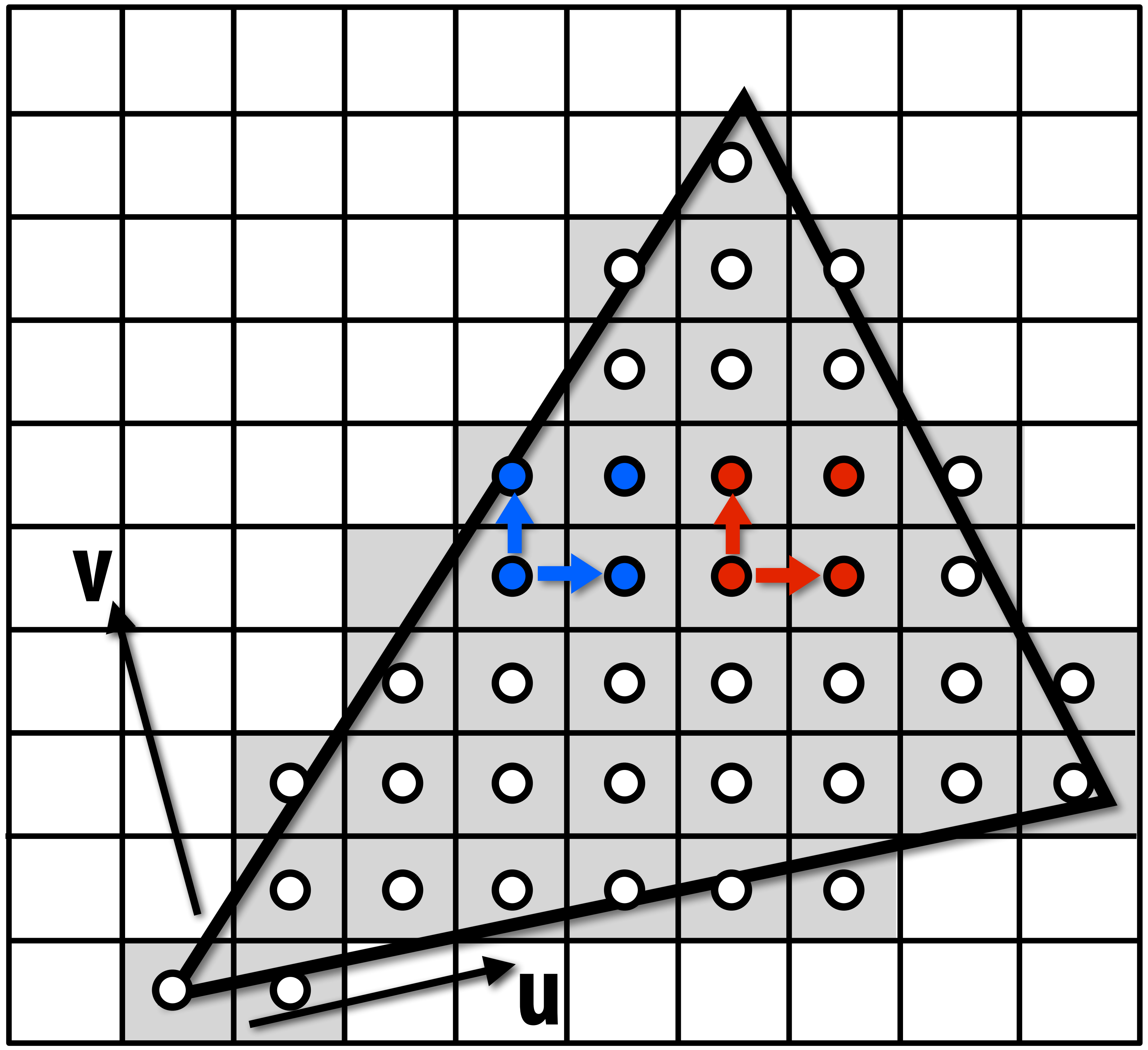


"Mip hierarchy"
level = *d*

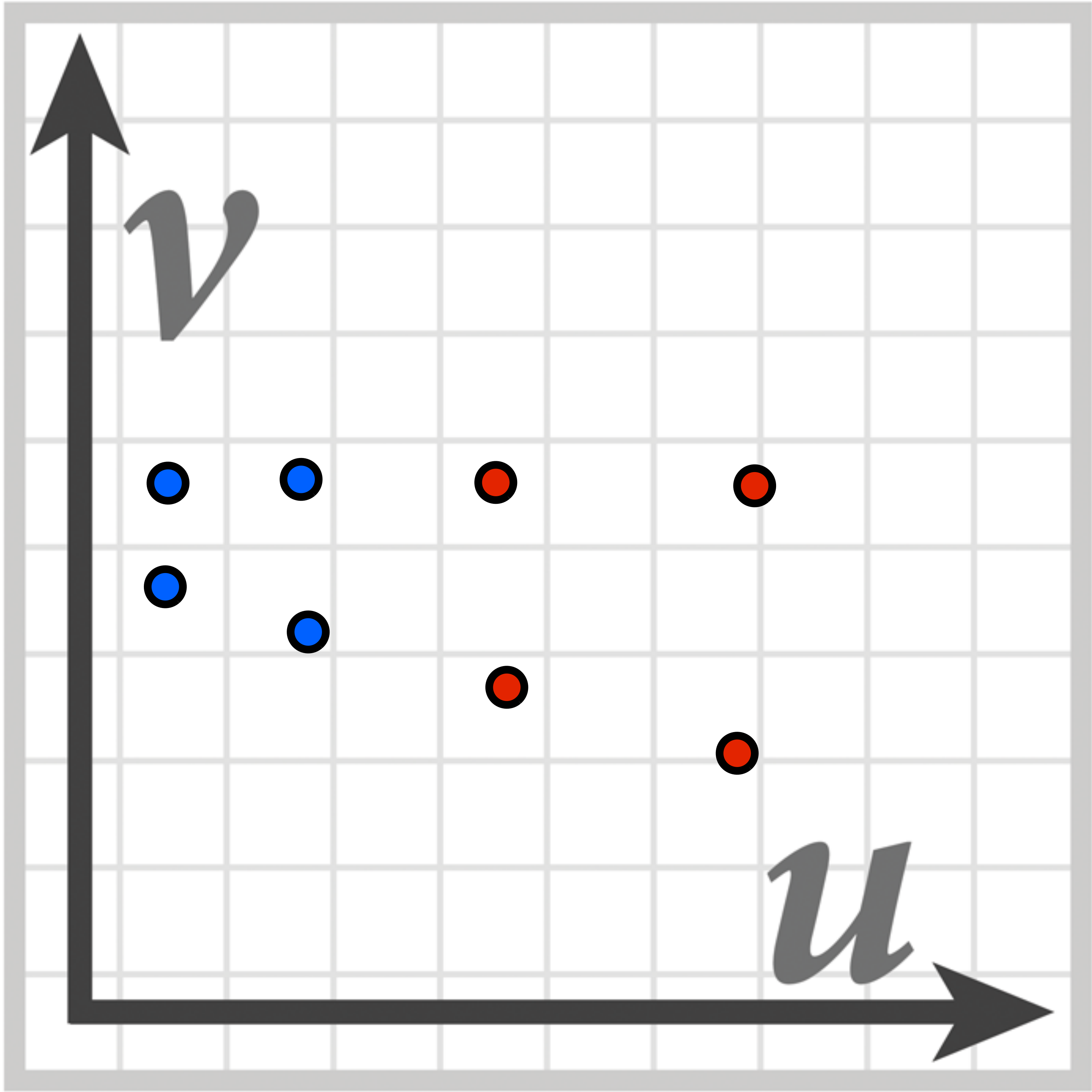
Slide credit: Akeley and Hanrahan

Computing *d*

Compute differences between texture coordinate values of neighboring fragments



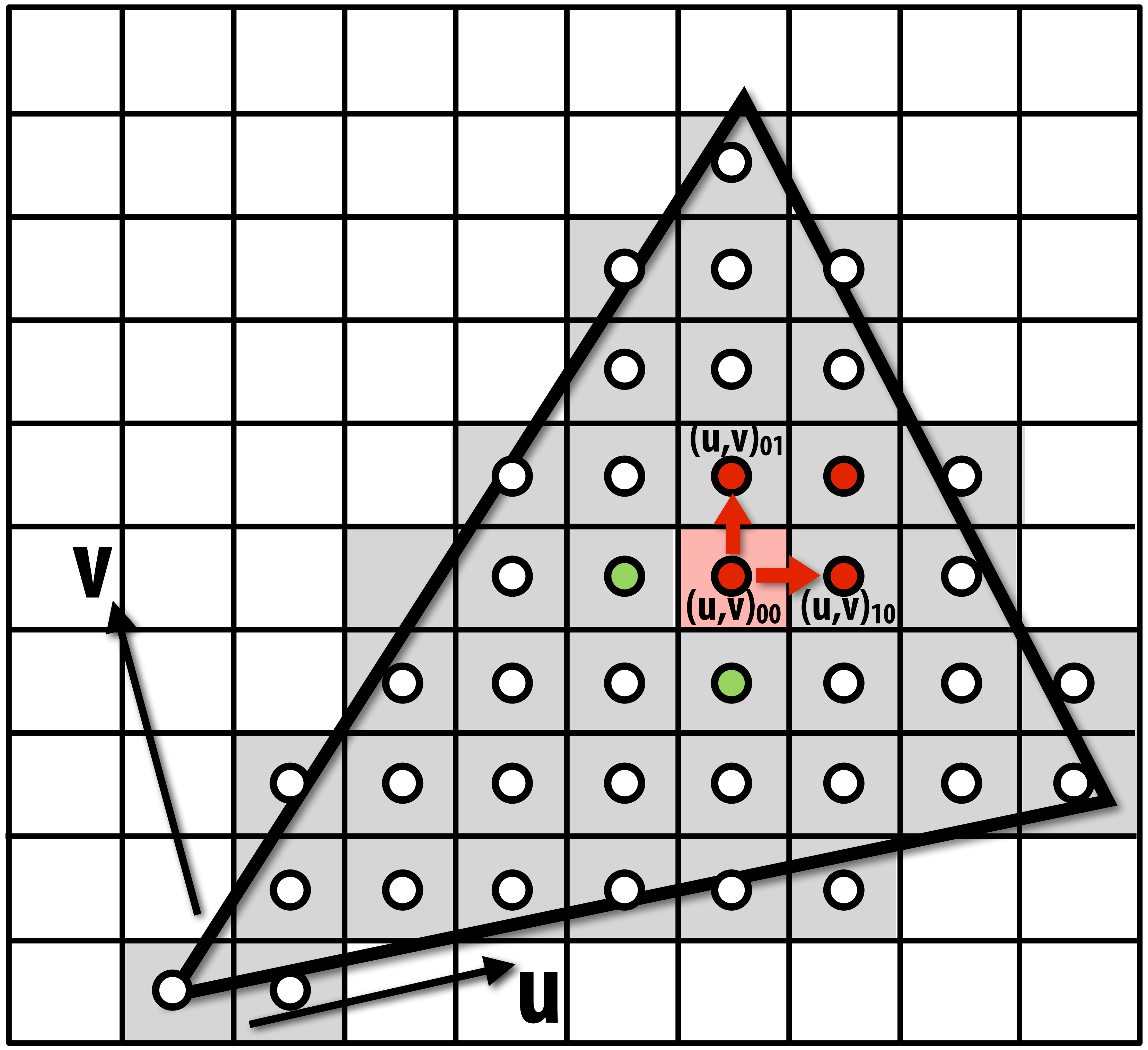
Screen space

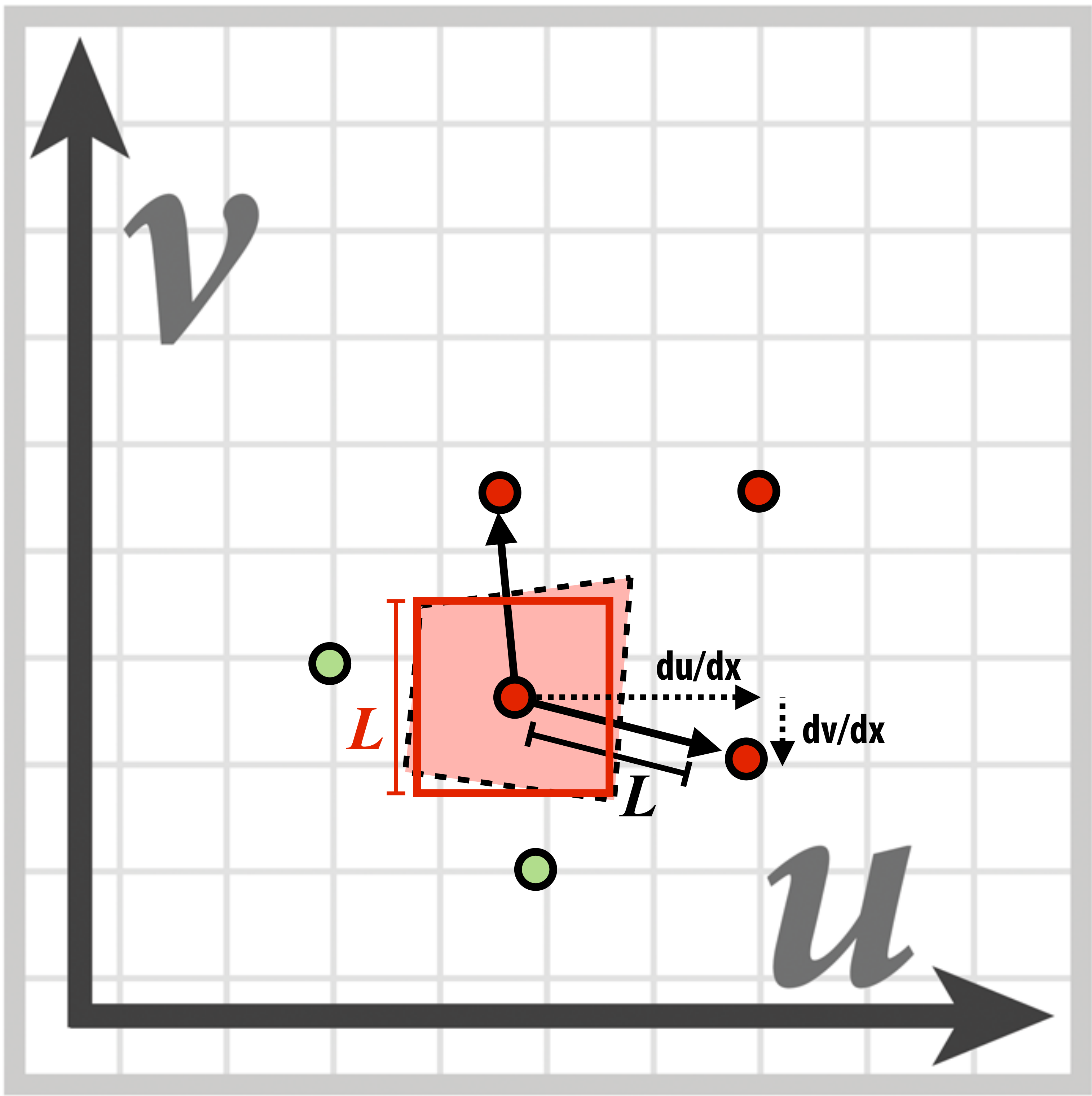


Texture space

Computing *d*

Compute differences between texture coordinate values of neighboring fragments





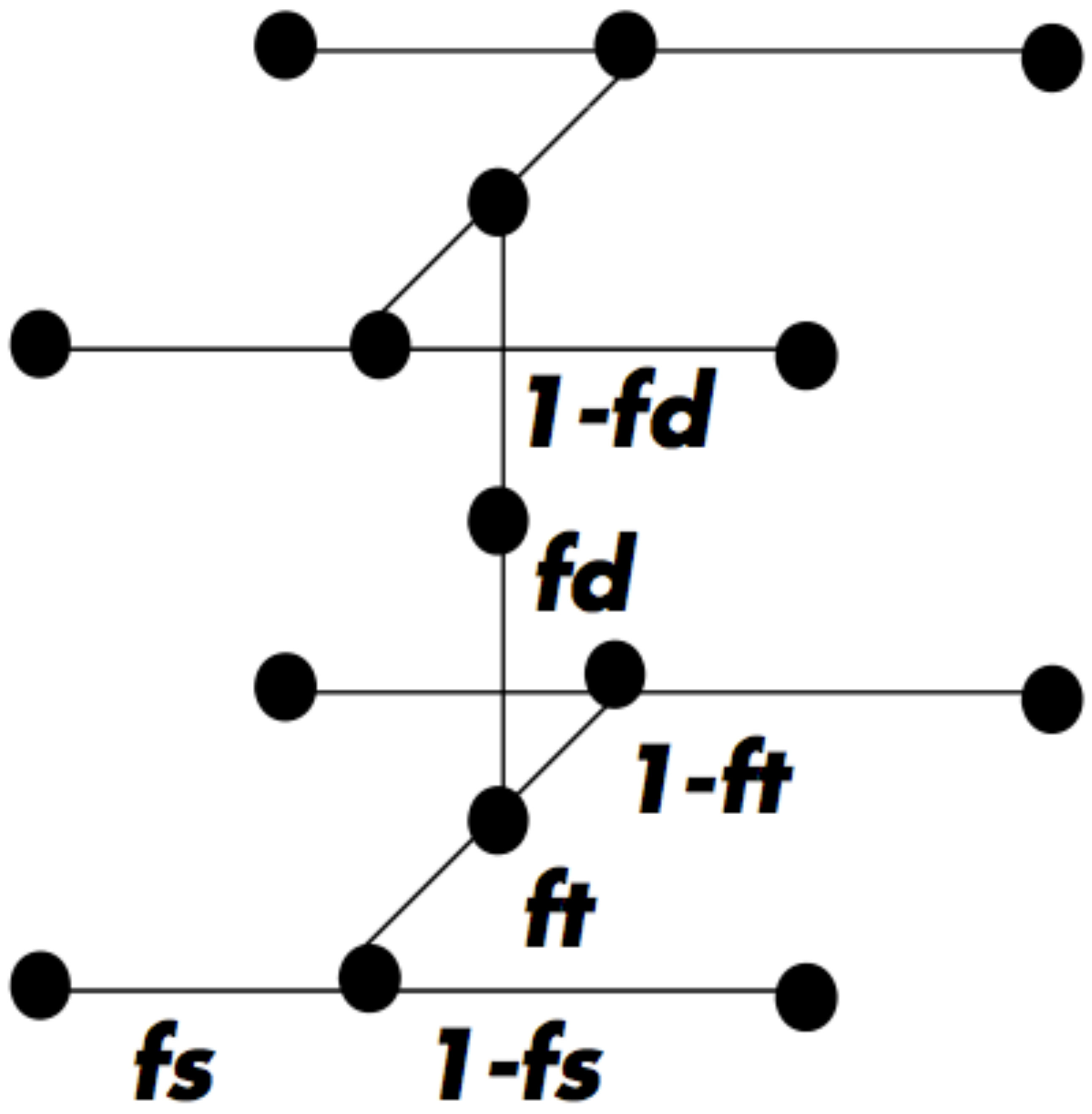
$du/dx = u_{10} - u_{00}$ $dv/dx = v_{10} - v_{00}$

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

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

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

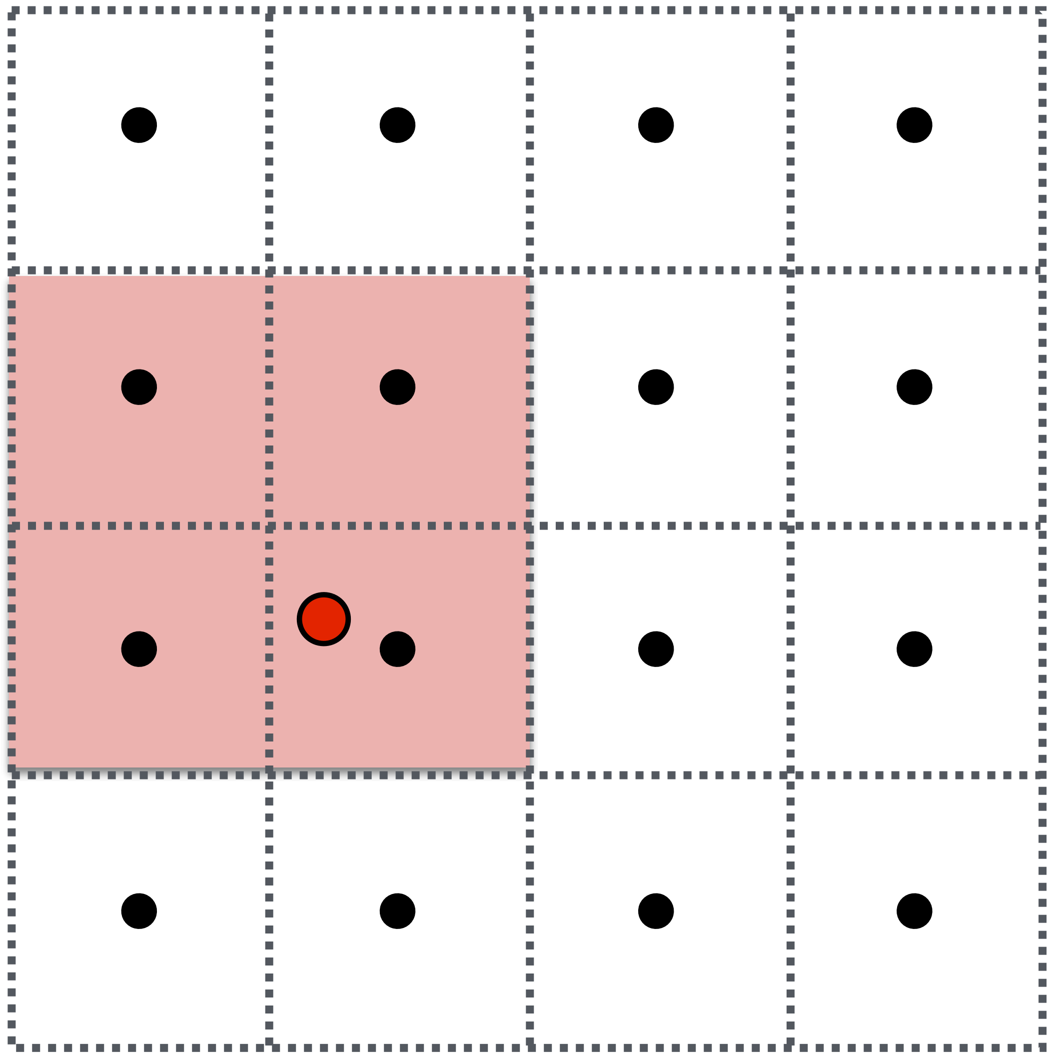
"Tri-linear" filtering



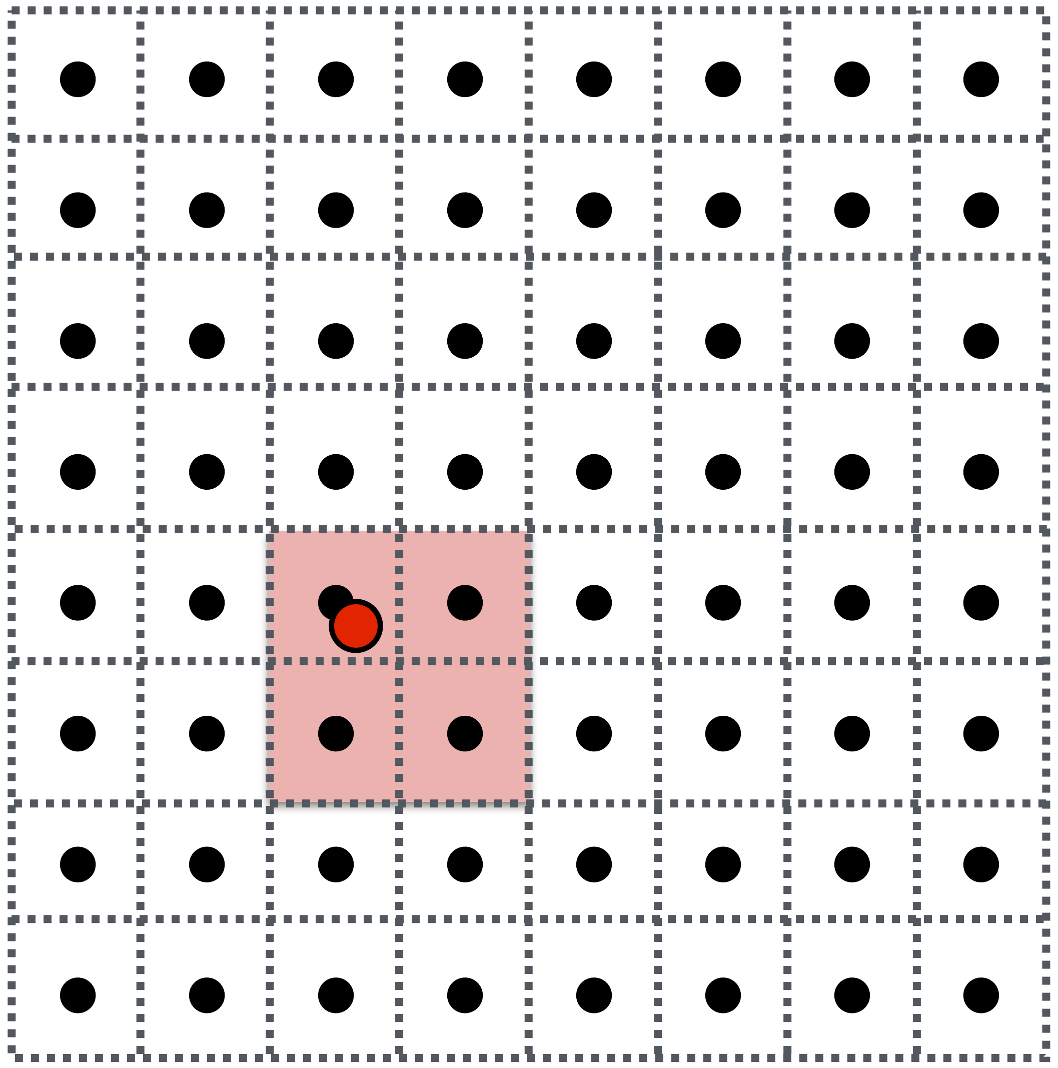
$$lerp(t, v_1, v_2) = v_1 + t(v_2 - v_1)$$

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

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



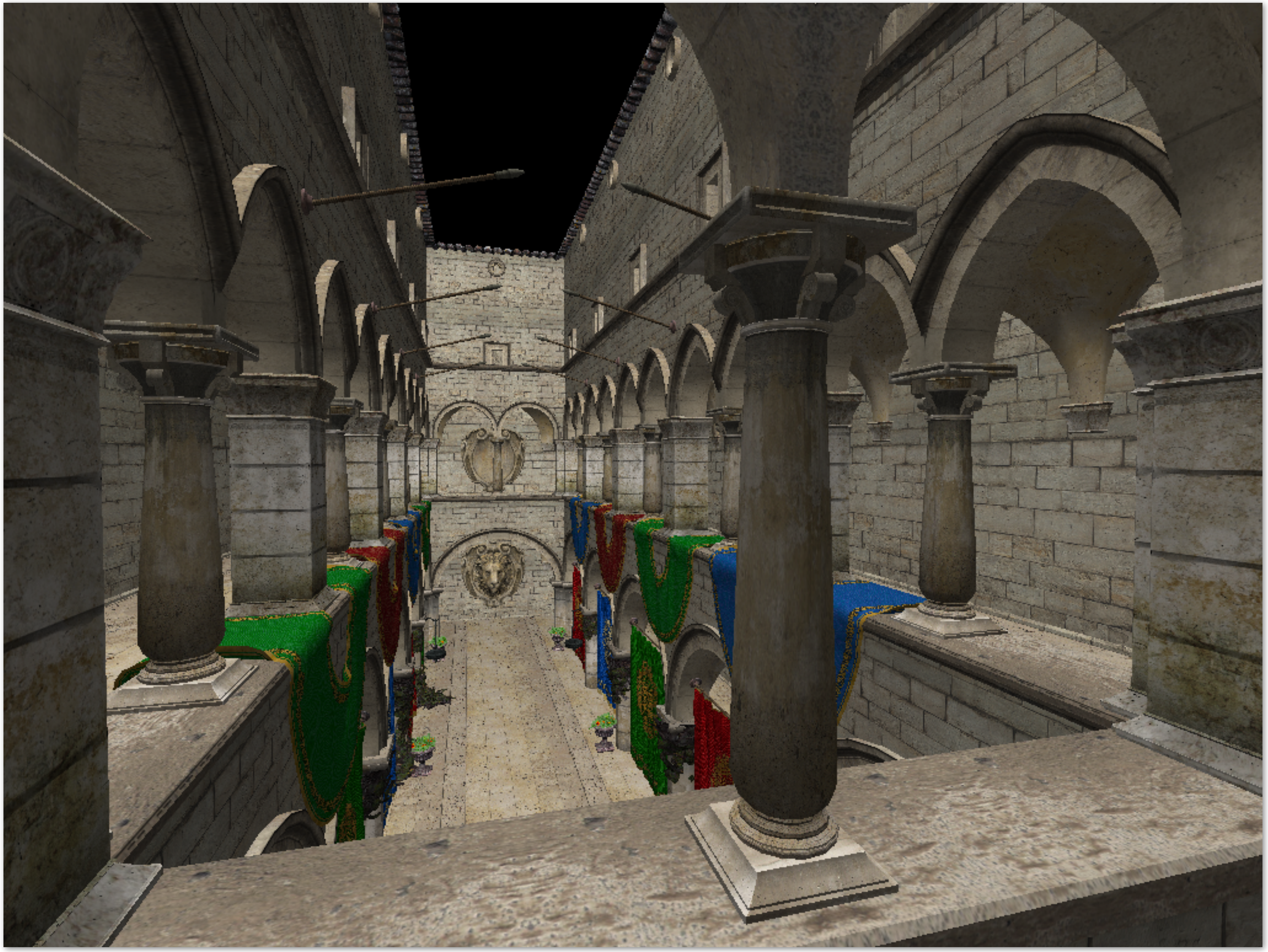
mip-map texels: level d+1



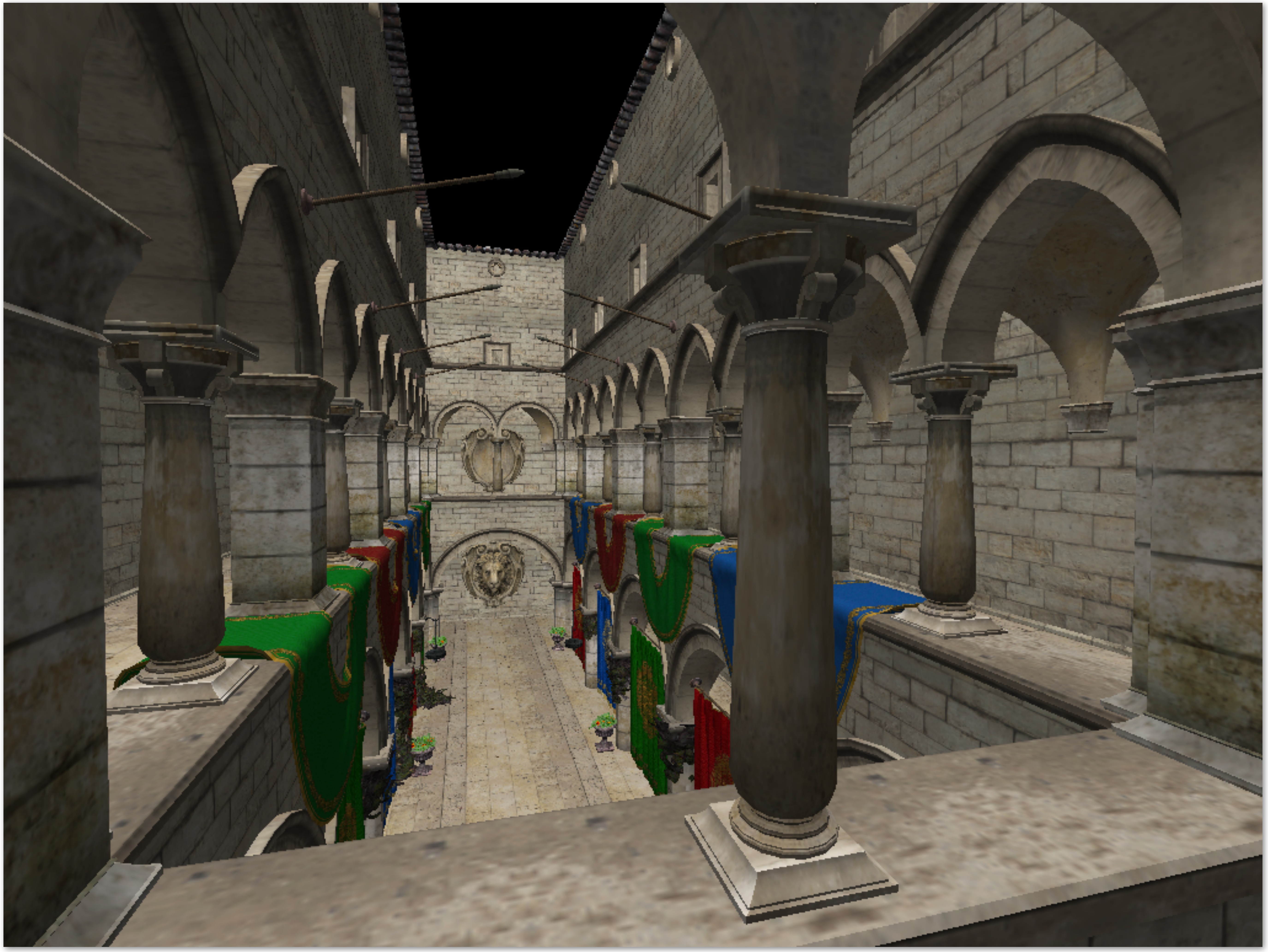
mip-map texels: level d

Figure credit: Akeley and Hanrahan

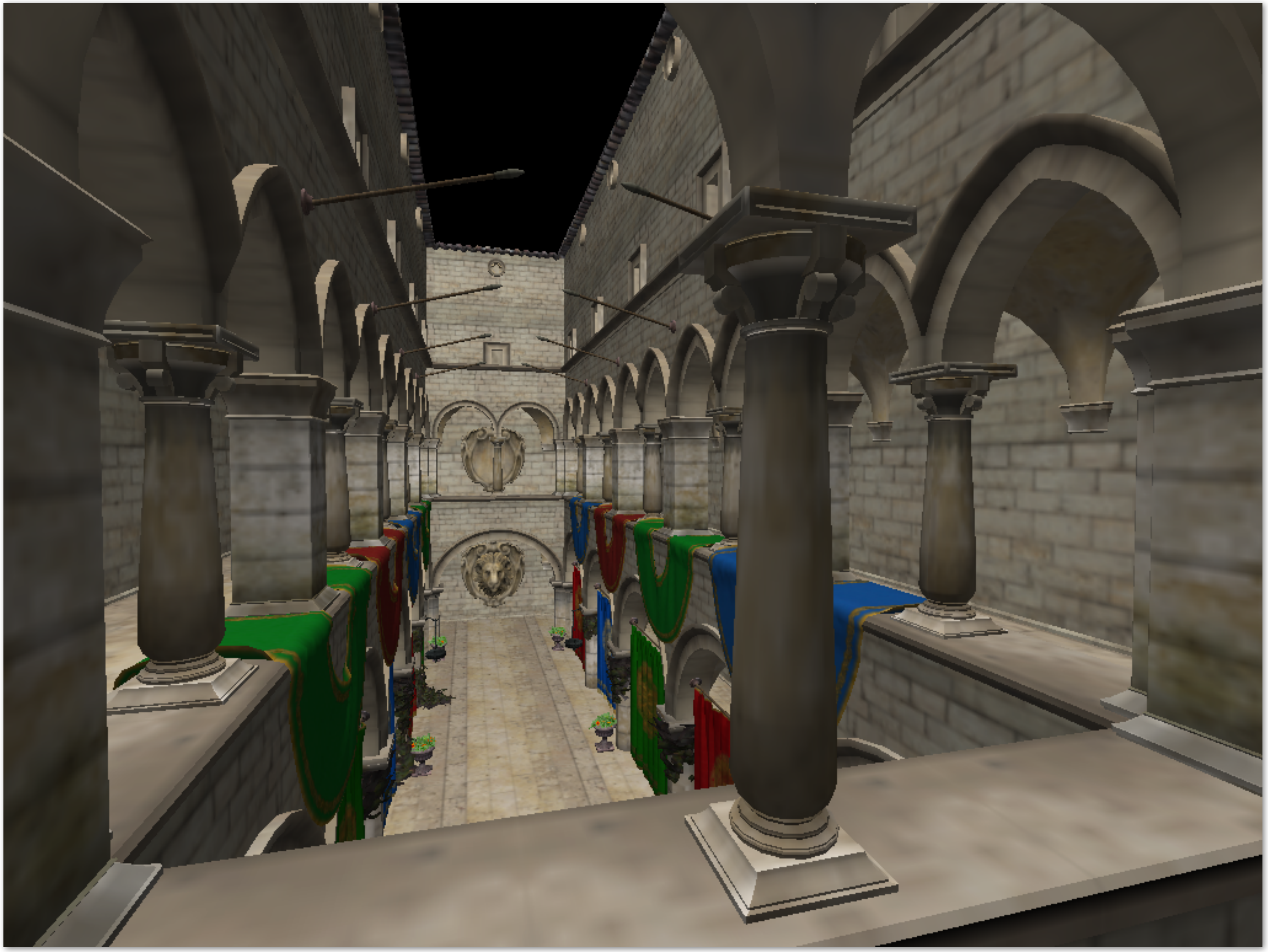
Sponza (bilinear resampling at level 0)



Sponza (bilinear resampling at level 2)

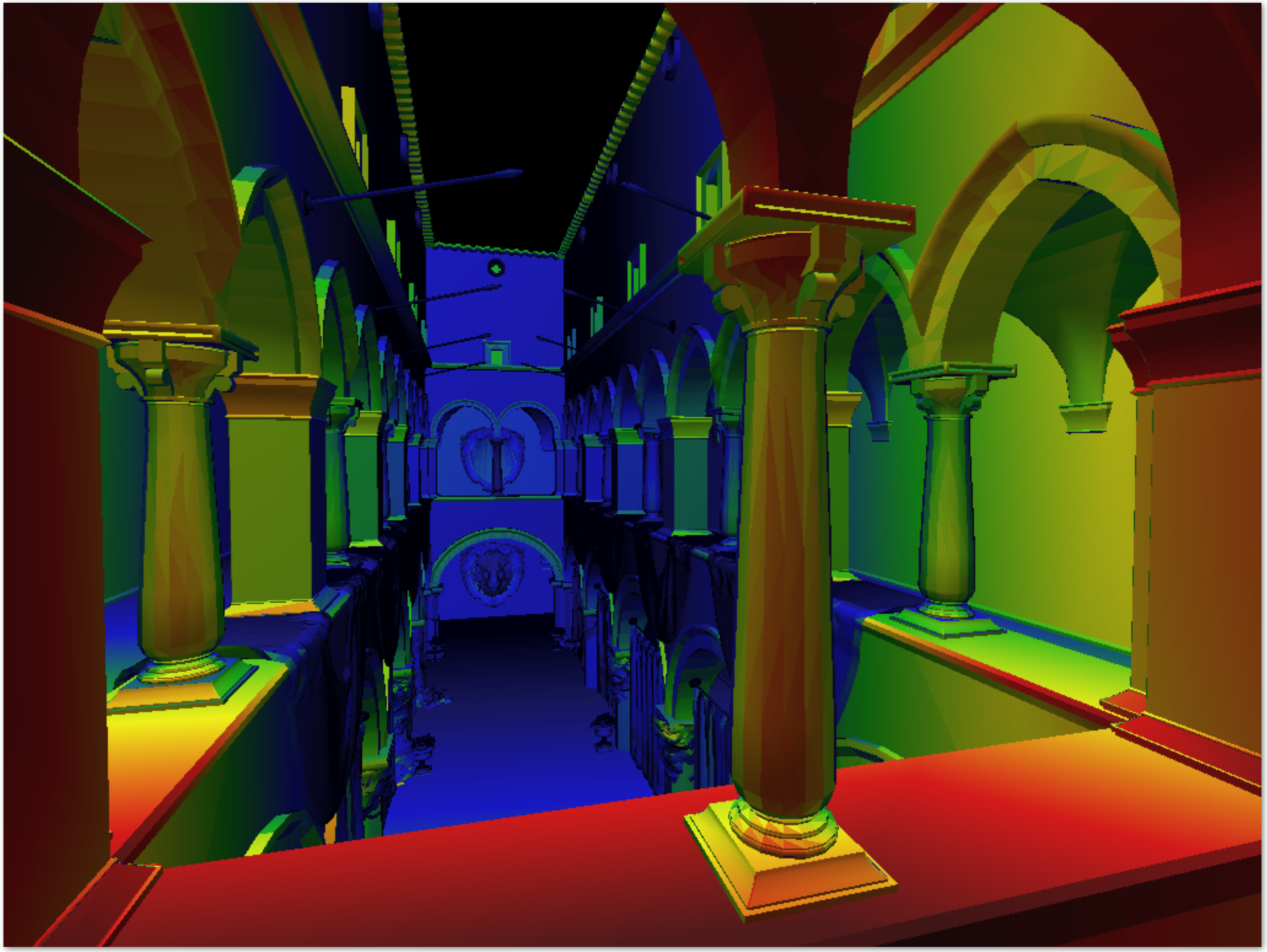


Sponza (bilinear resampling at level 4)



Mip-map level visualization

(trilinear filtering: visualization of continuous *d*)



GPUs shade at the granularity of 2x2 fragments

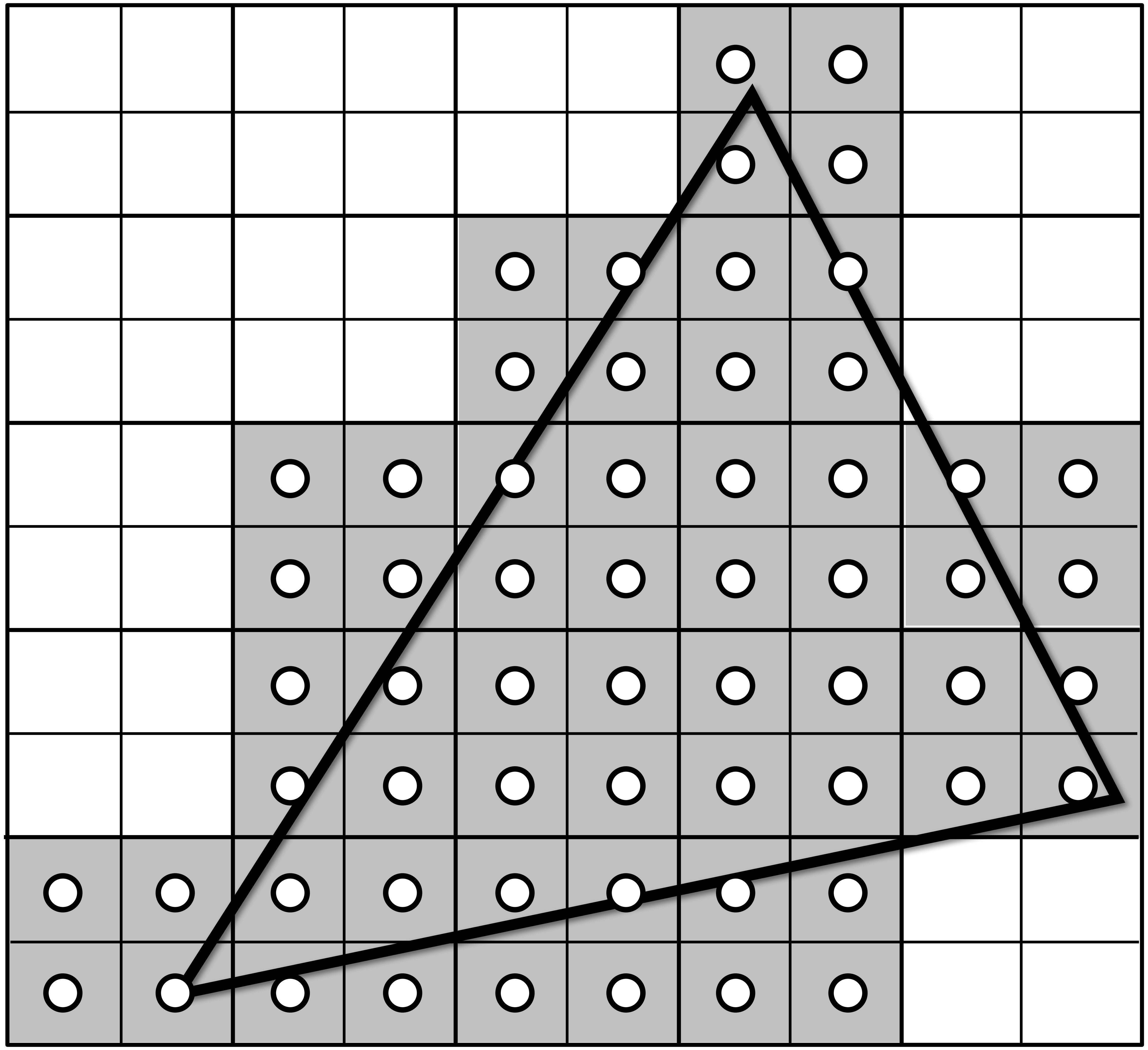
(“quad fragment” is the minimum granularity of rasterization output and shading)

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)



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: 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.

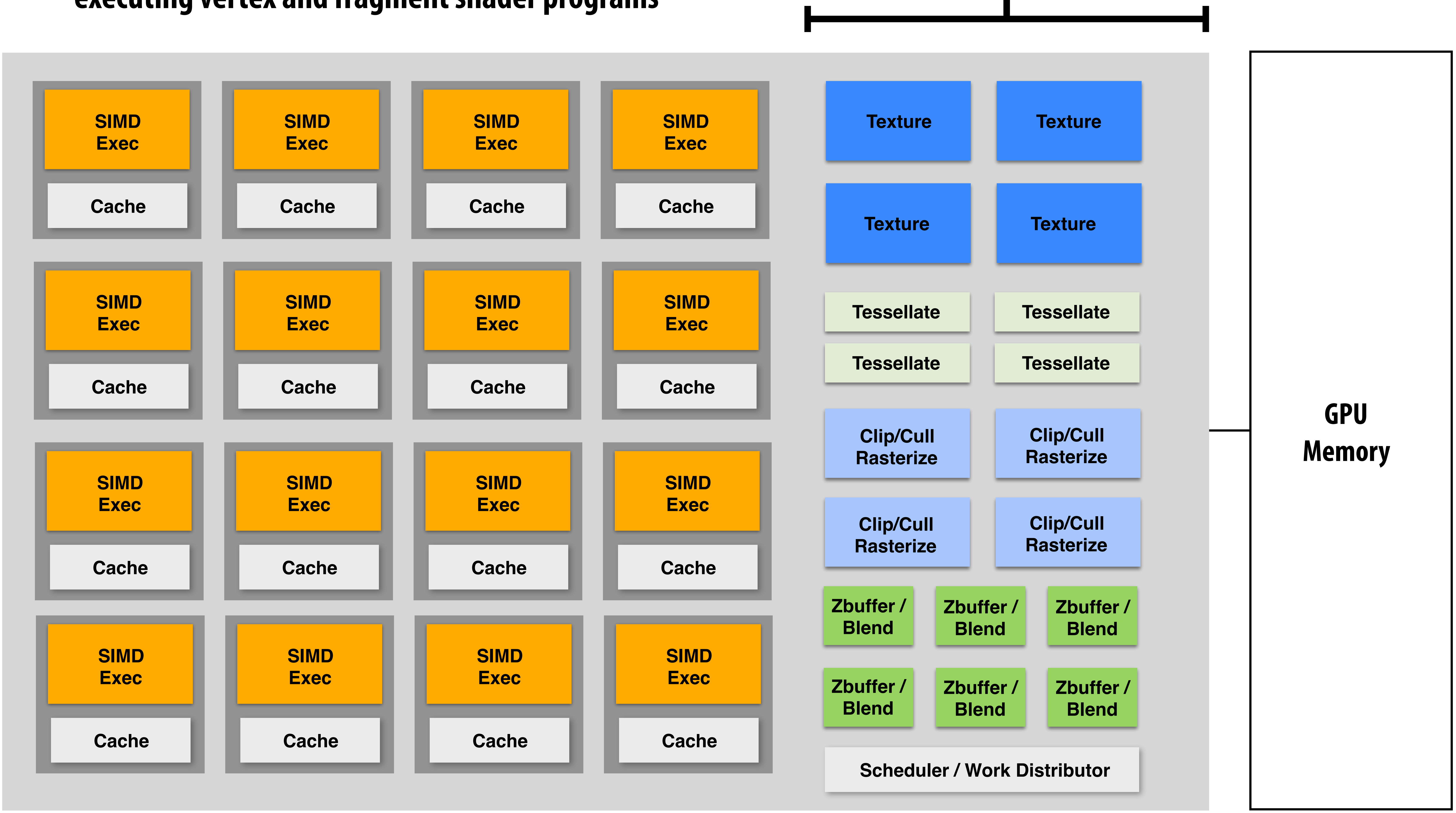
All modern GPUs have dedicated fixed-function hardware support for performing texture sampling operations.

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

GPU: heterogeneous, multi-core processor

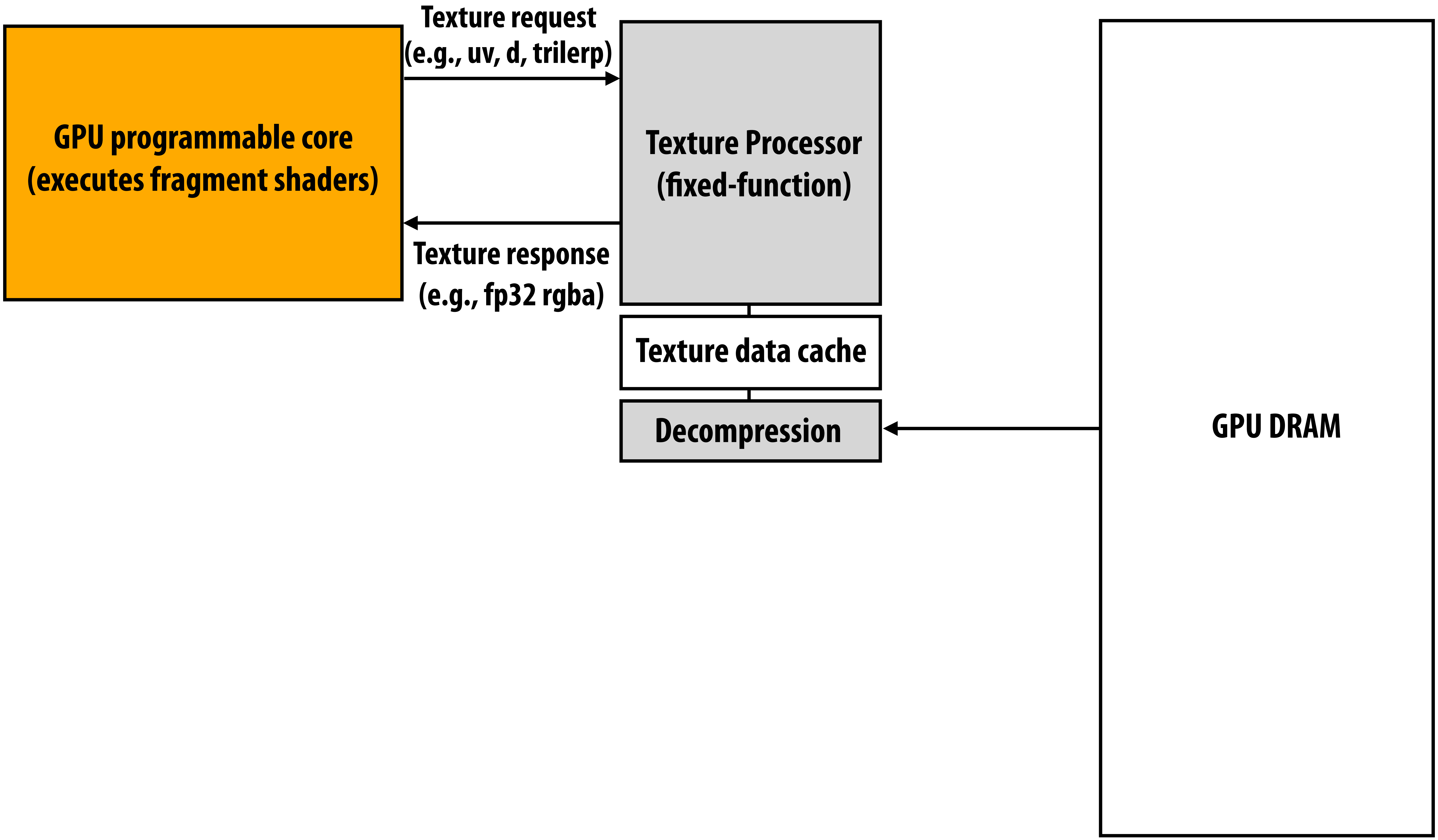
Modern GPUs offer ~ TFLOPs of performance for executing vertex and fragment shader programs

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



Texture caching

Texture system block diagram



Texture request
(e.g., uv, d, trilerp)

GPU programmable core
(executes fragment shaders)

Texture Processor
(fixed-function)

Texture response
(e.g., fp32 rgba)

Texture data cache

Decompression

GPU DRAM

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)
- **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**

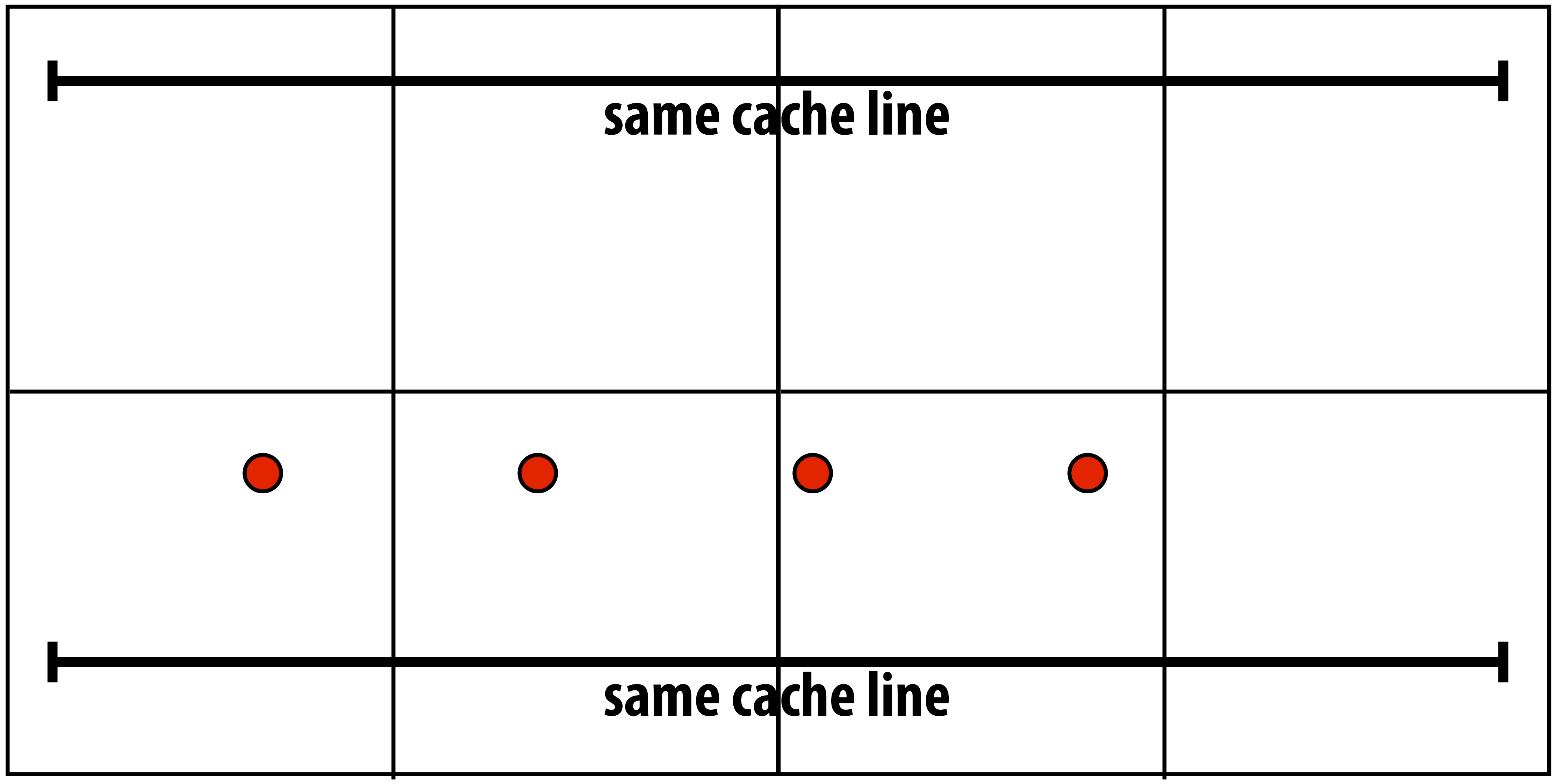
Texture caching thought experiment

Assume:

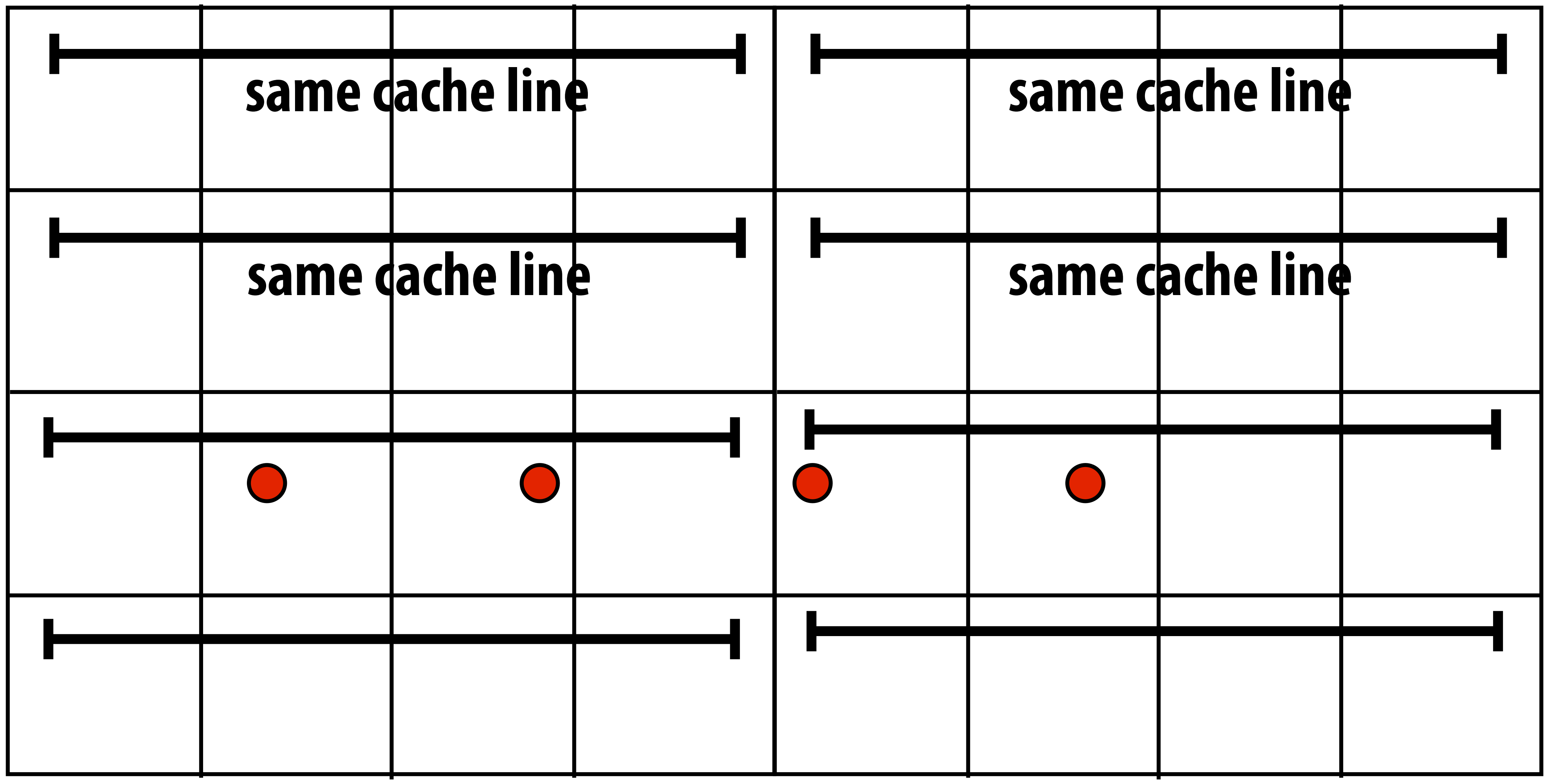
Row-major rasterization order

Horizontal texels contiguous in memory

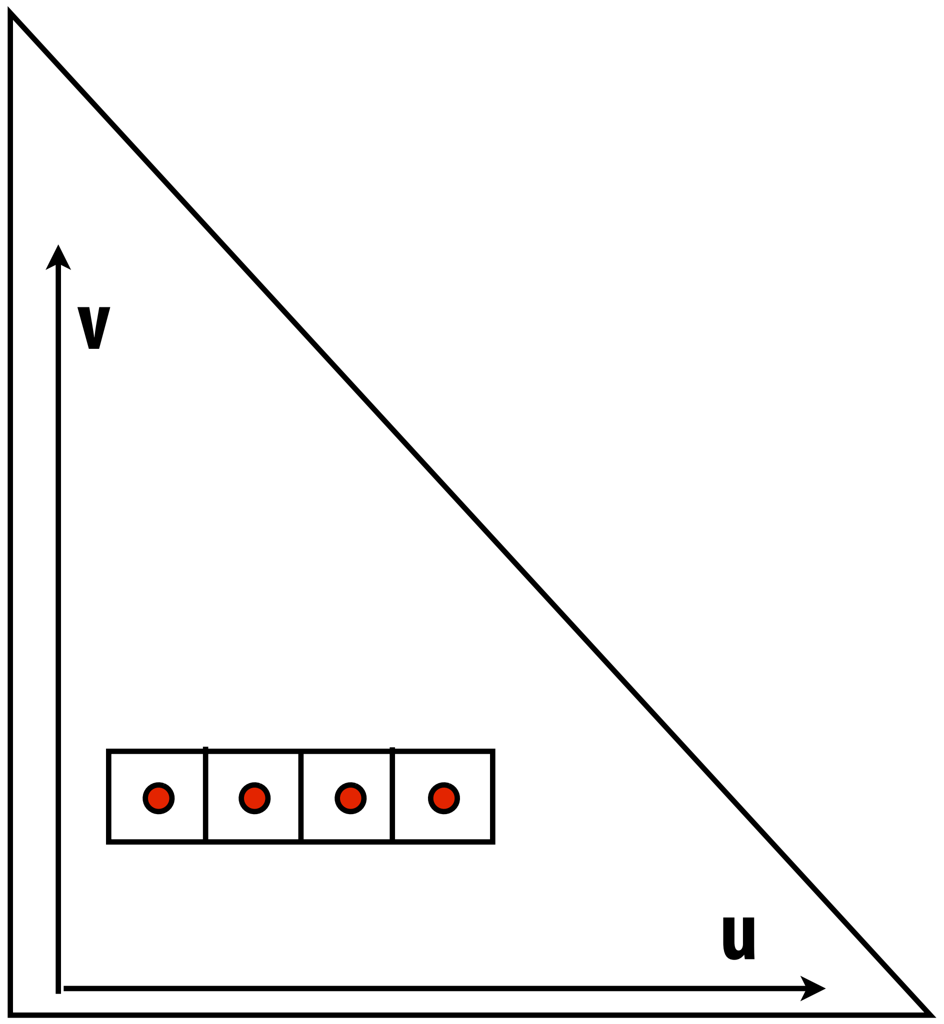
Texture cache line = 4 texels



mip-map: level *d+1* texels



mip-map: level *d* 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 (there is little reason for a shader to access the same part of the texture map twice when computing surface appearance)

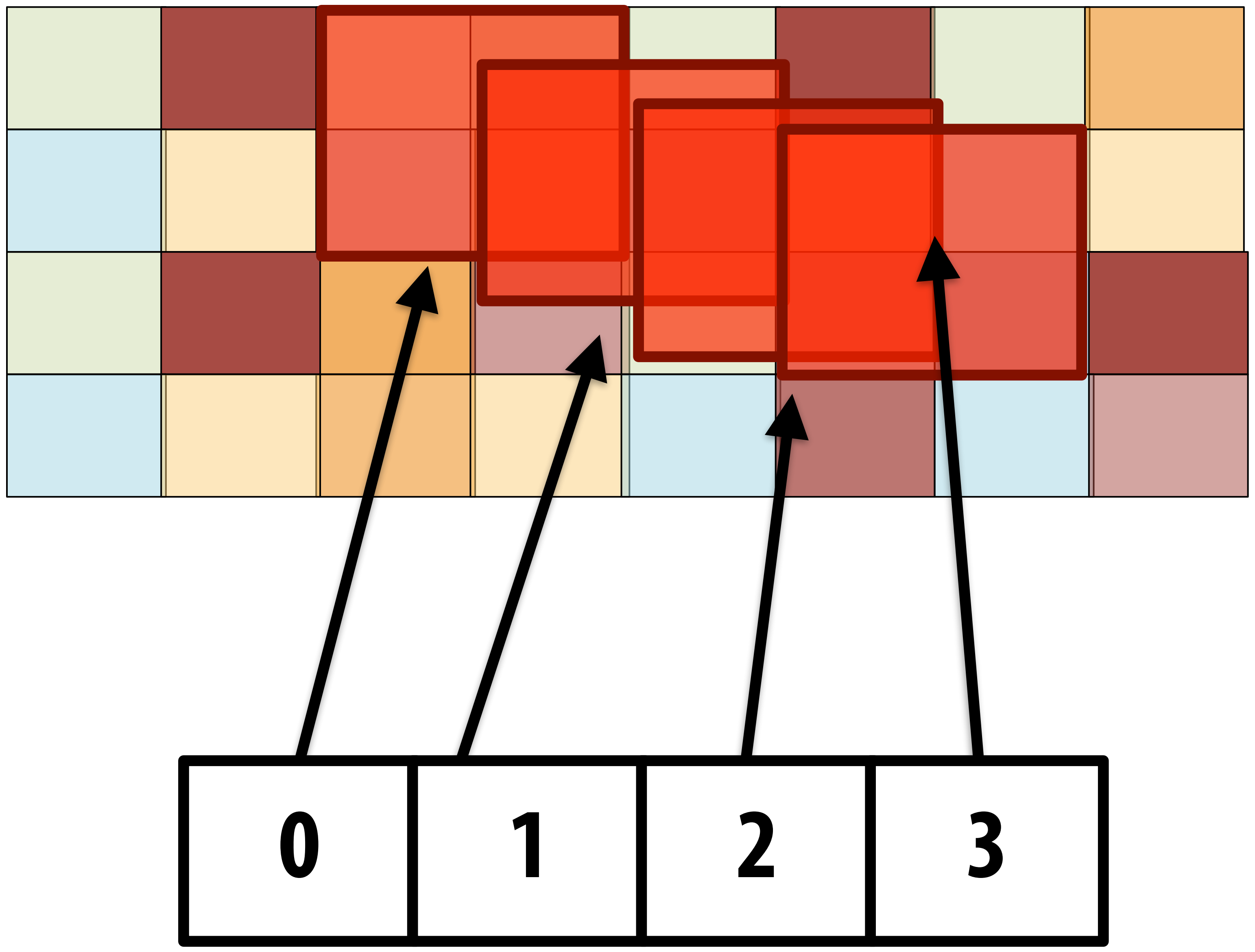
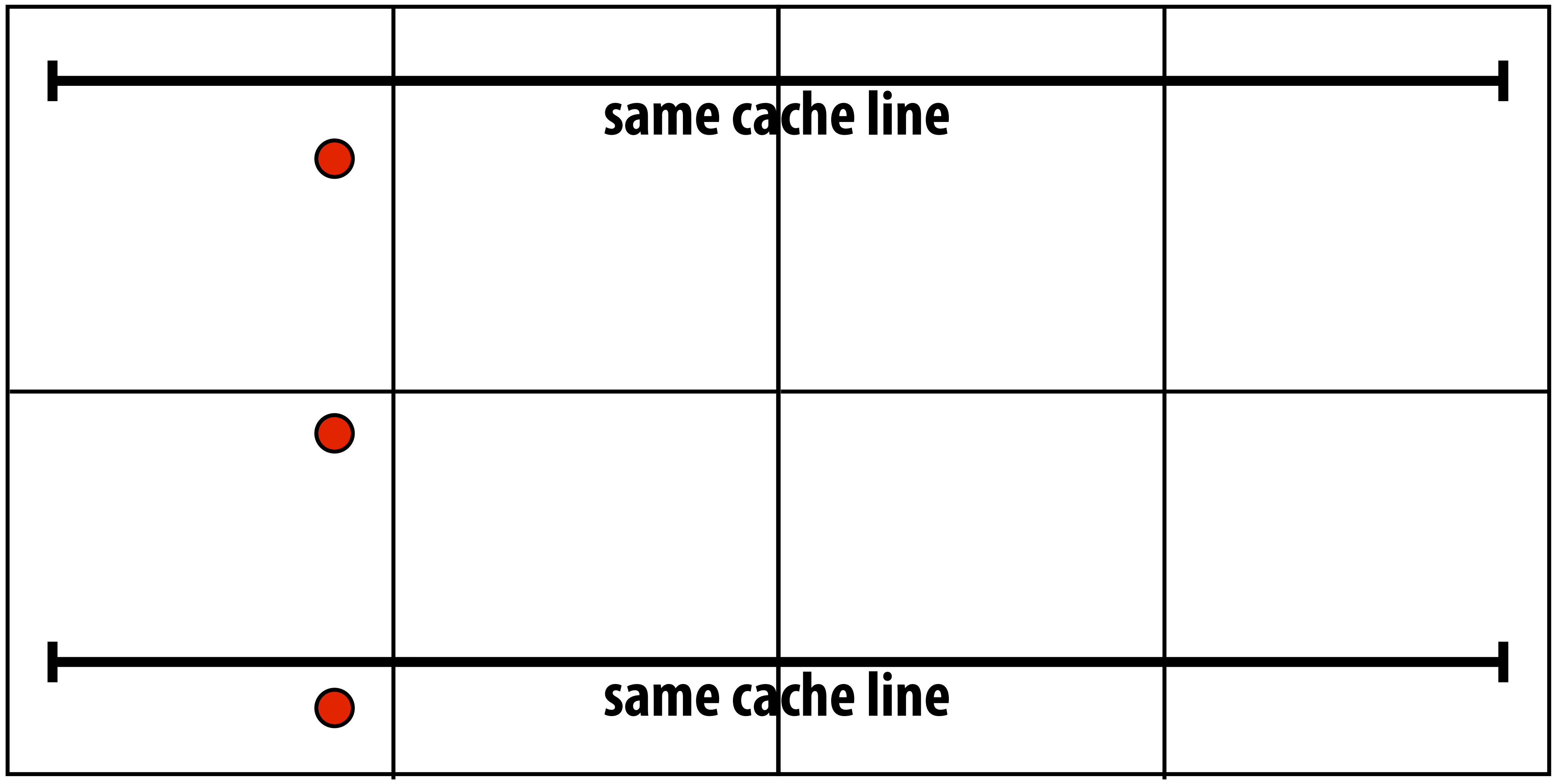
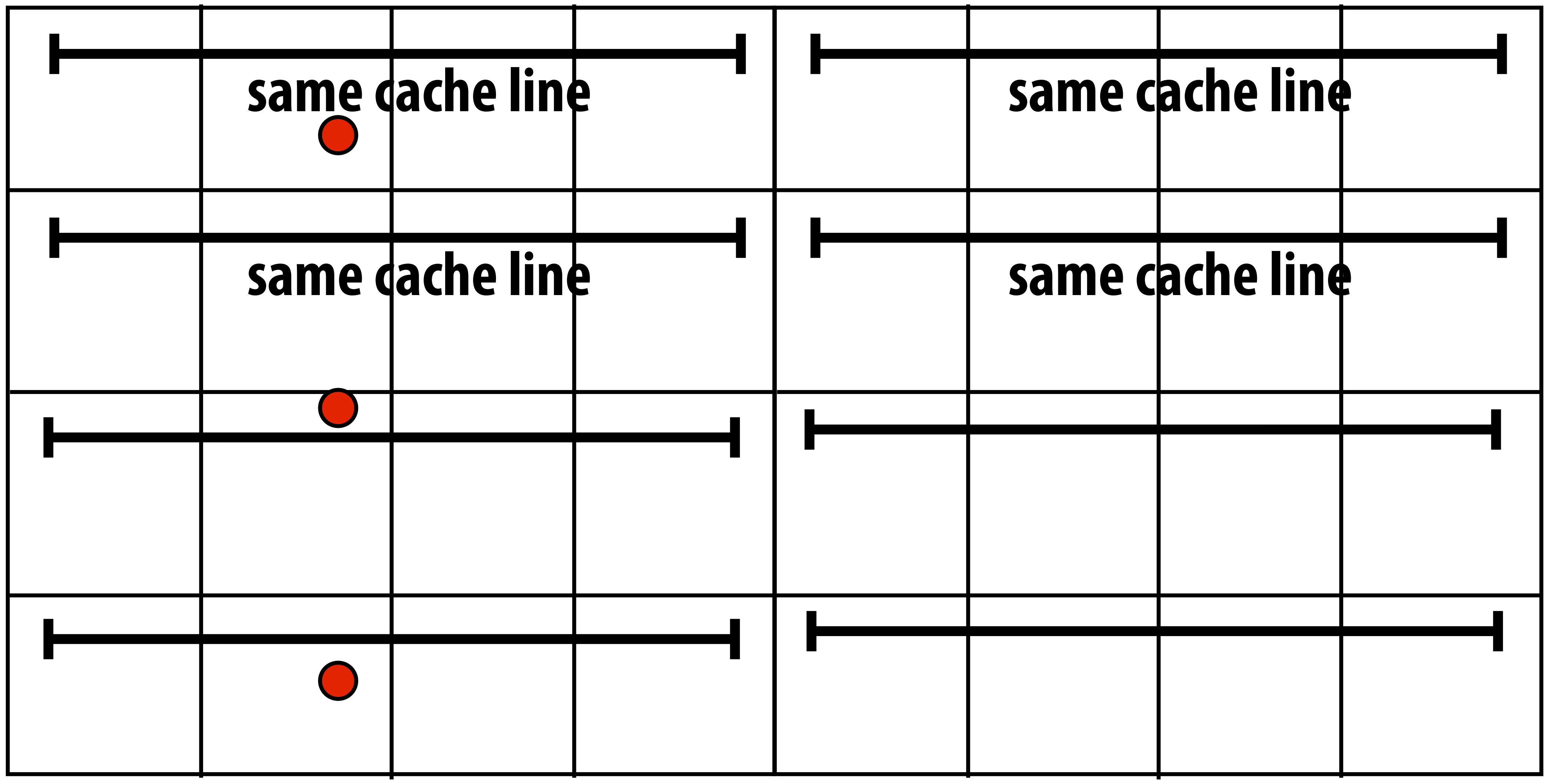


Figure illustrates filter support regions from texture fetches from four adjacent fragments

Now rotate triangle on screen



mip-map: level *d+1* texels



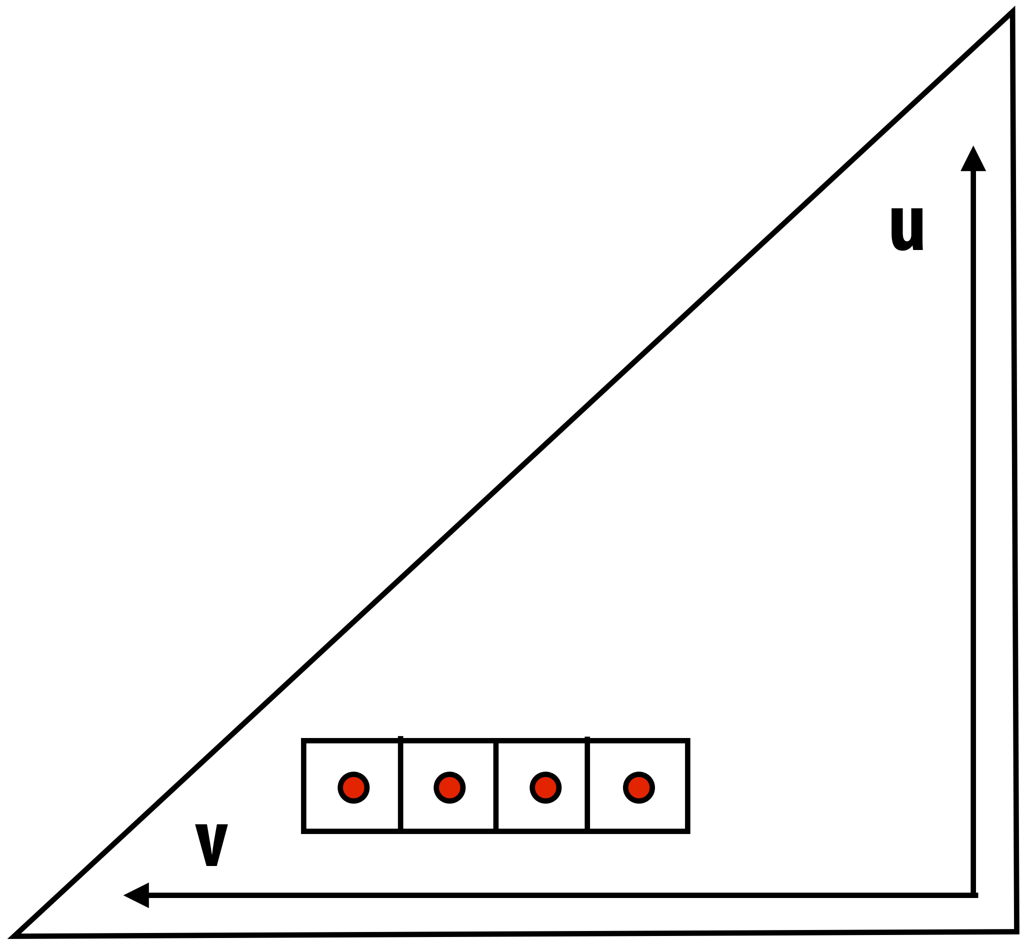
mip-map: level *d* texels

Assume:

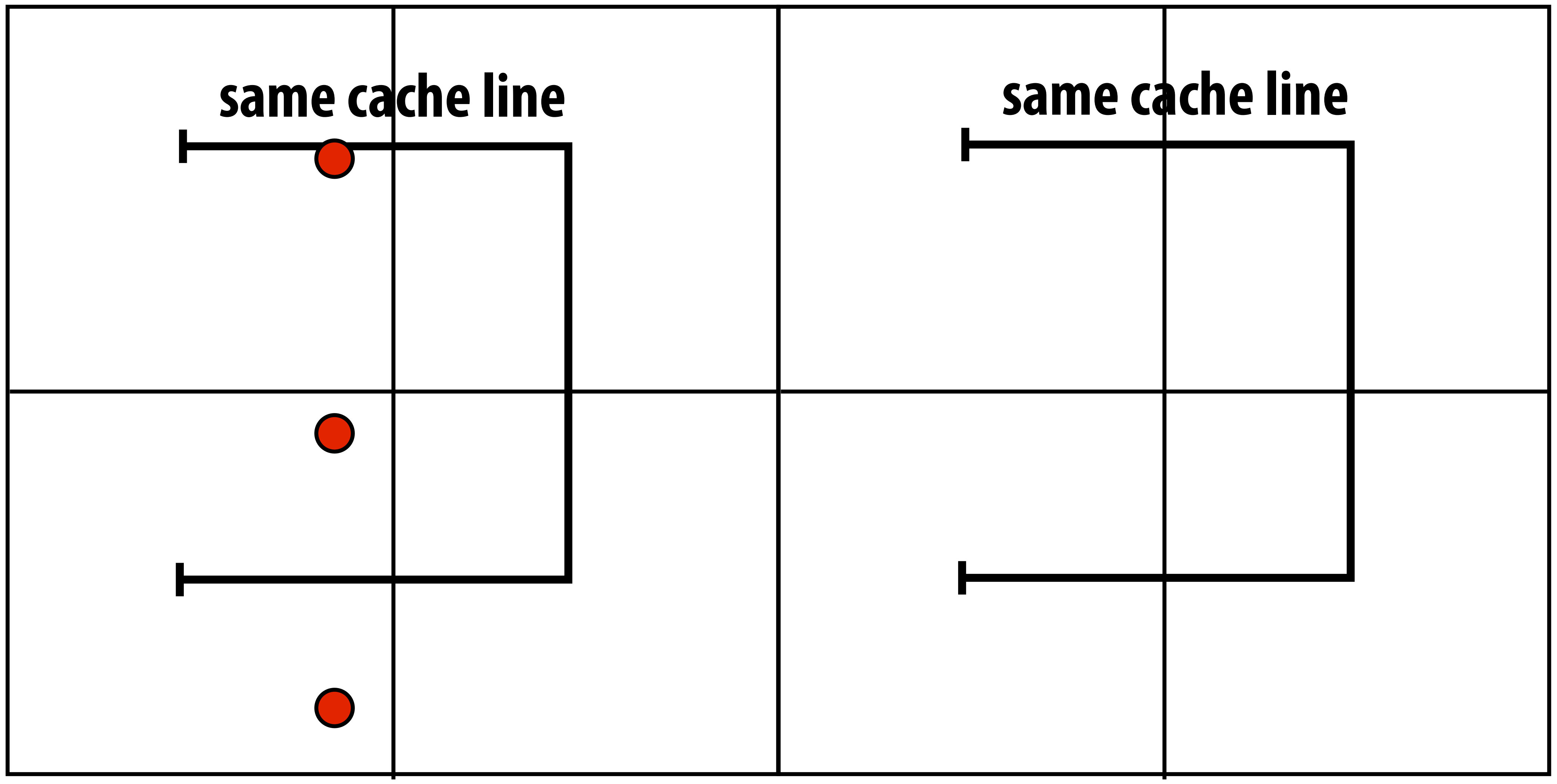
Row-major rasterization order

Horizontal texels contiguous in memory

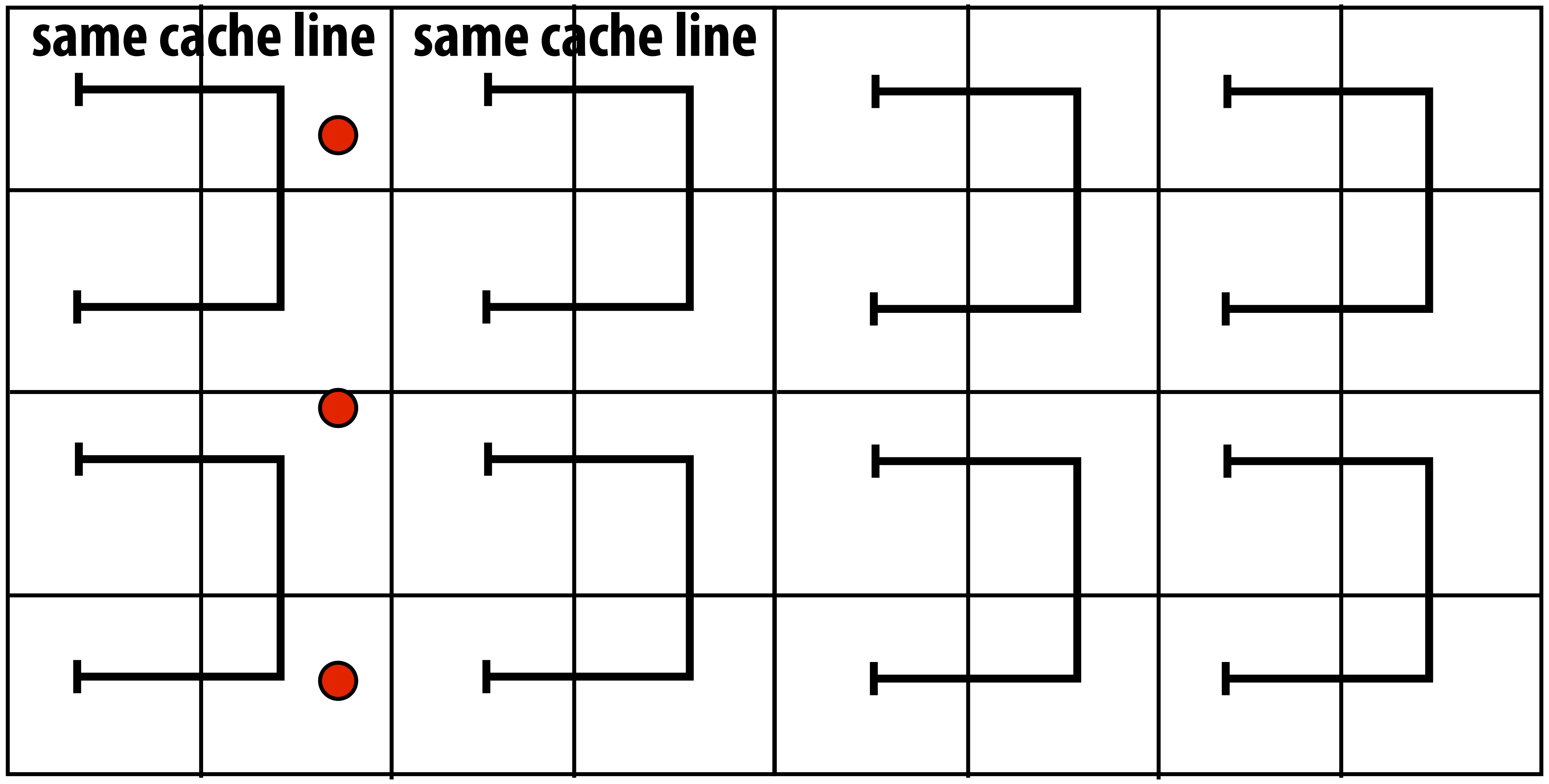
Cache line = 4 texels



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



mip-map: level $d+1$ texels



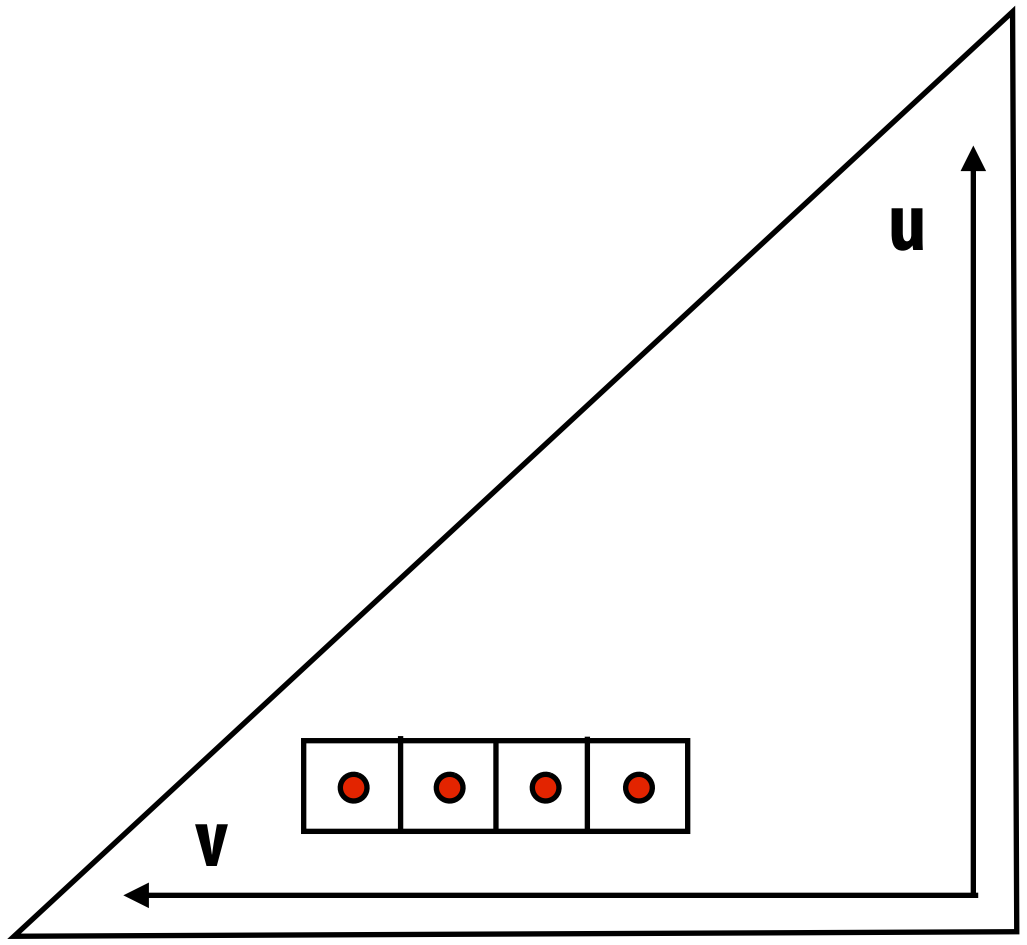
mip-map: level d texels

Assume:

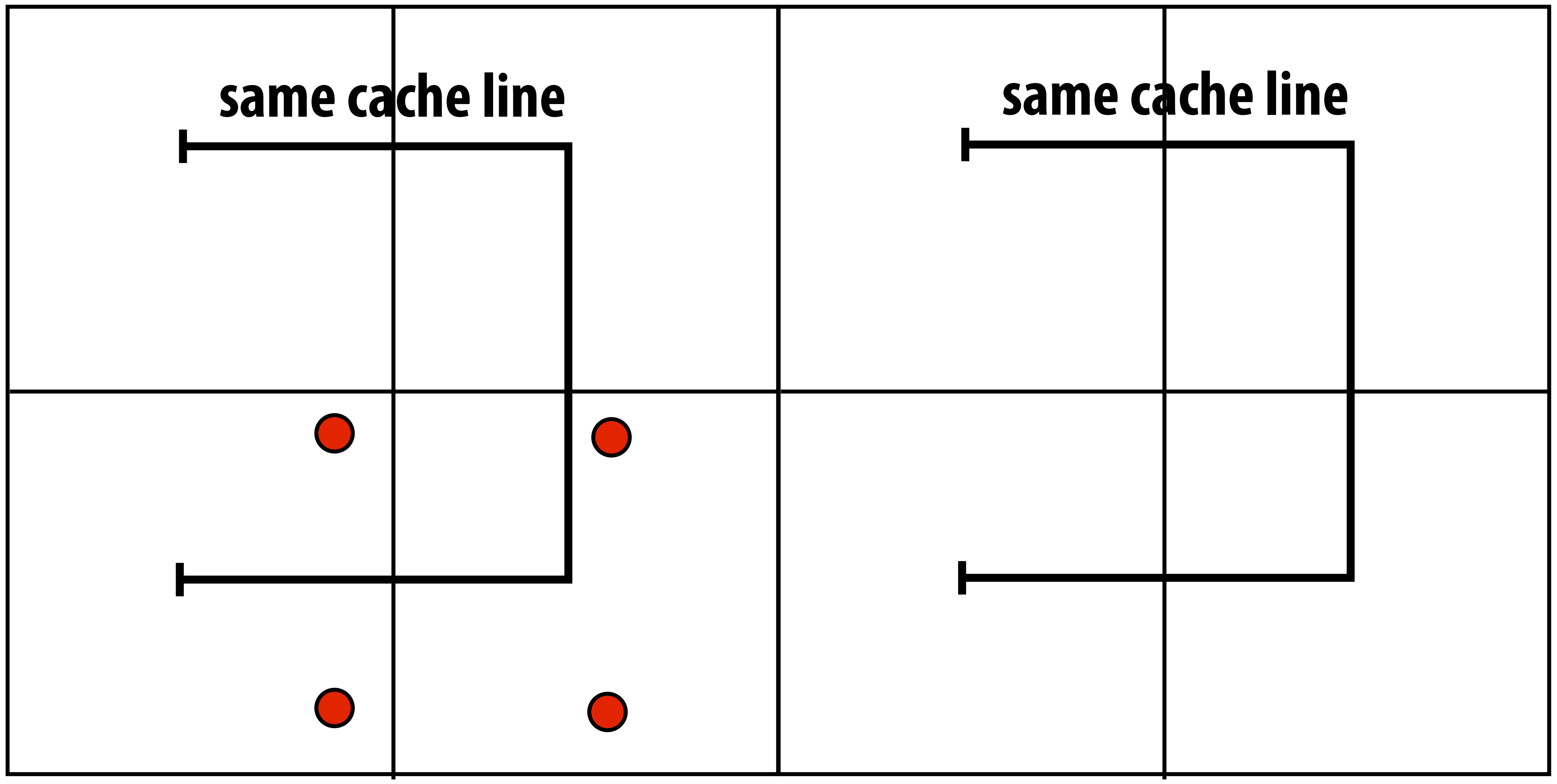
Row-major rasterization order

2D blocks of texels contiguous in memory

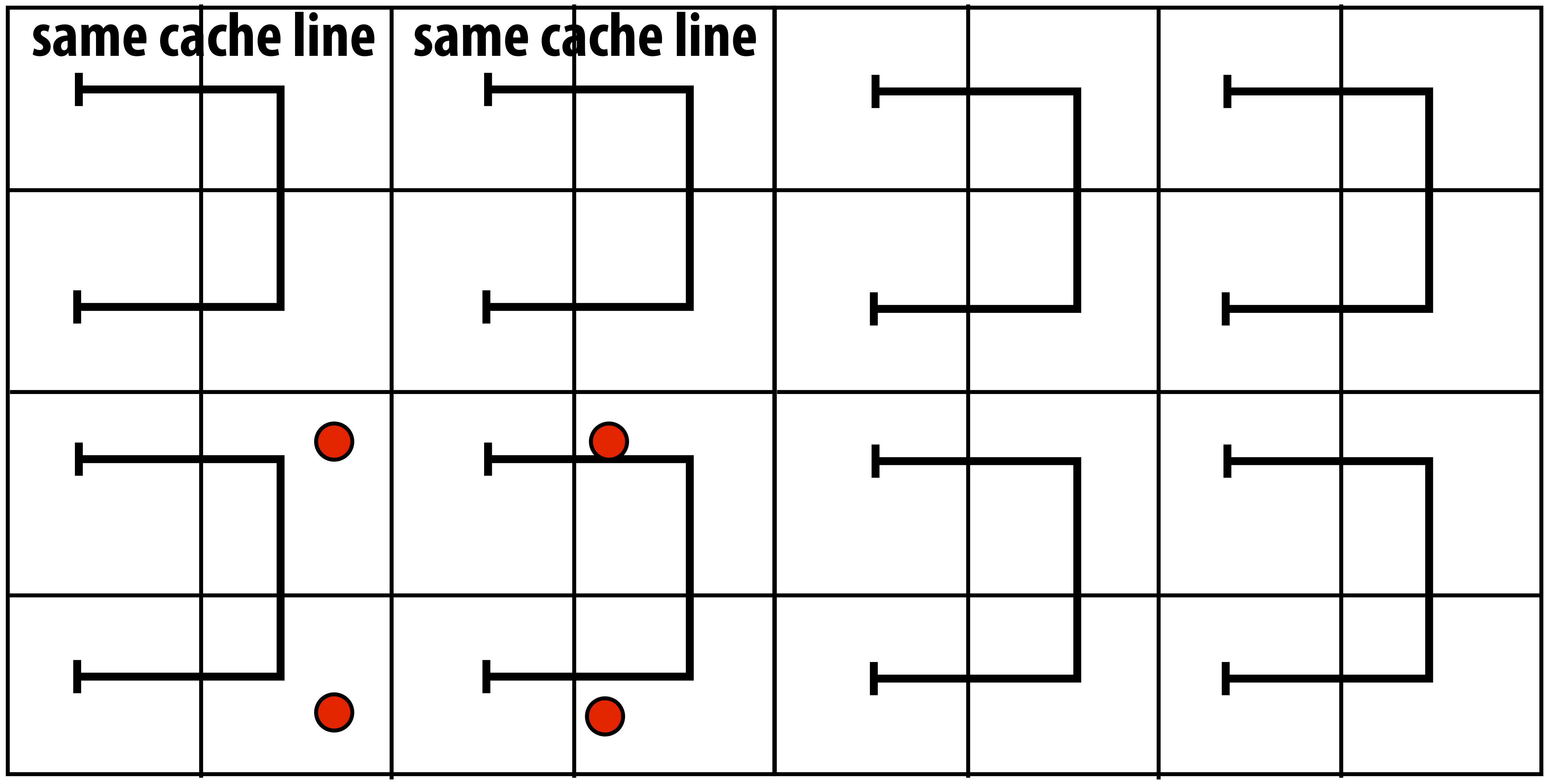
Cache line = 4 texels



Tiled rasterization increases reuse



mip-map: level *d+1* texels



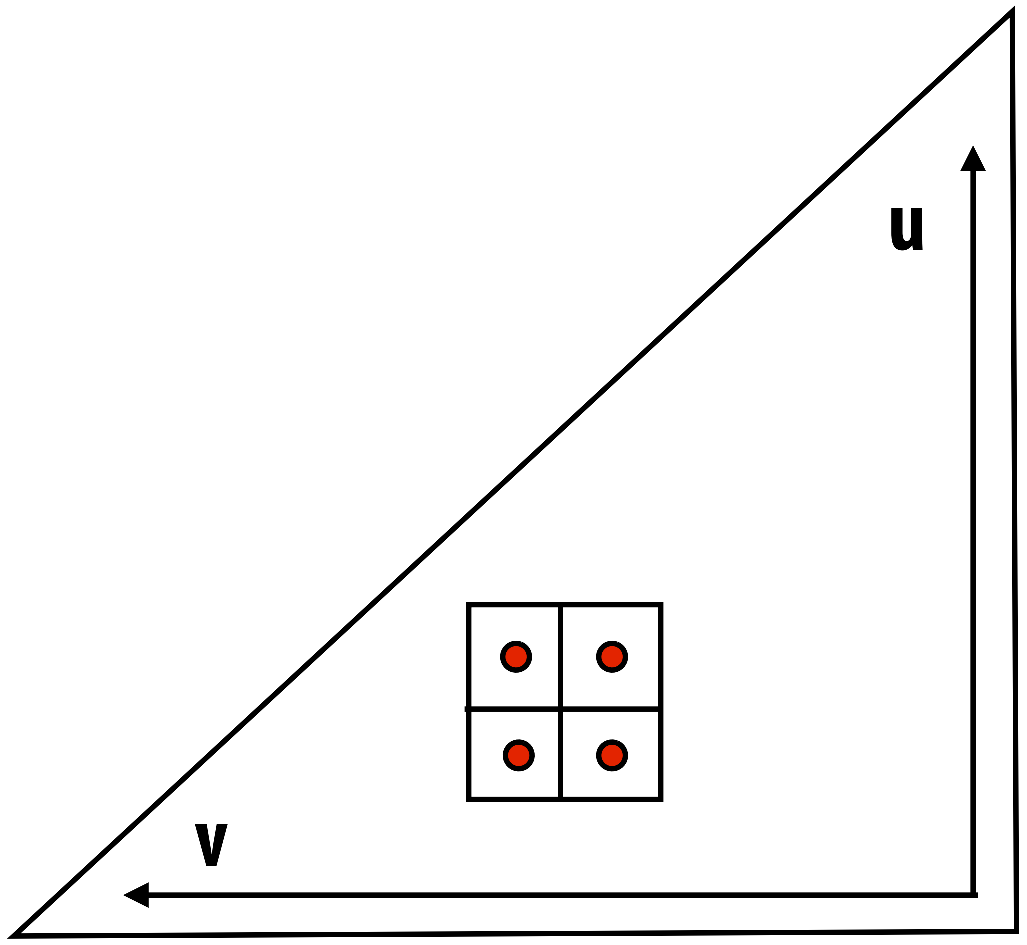
mip-map: level *d* texels

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

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 includes the 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

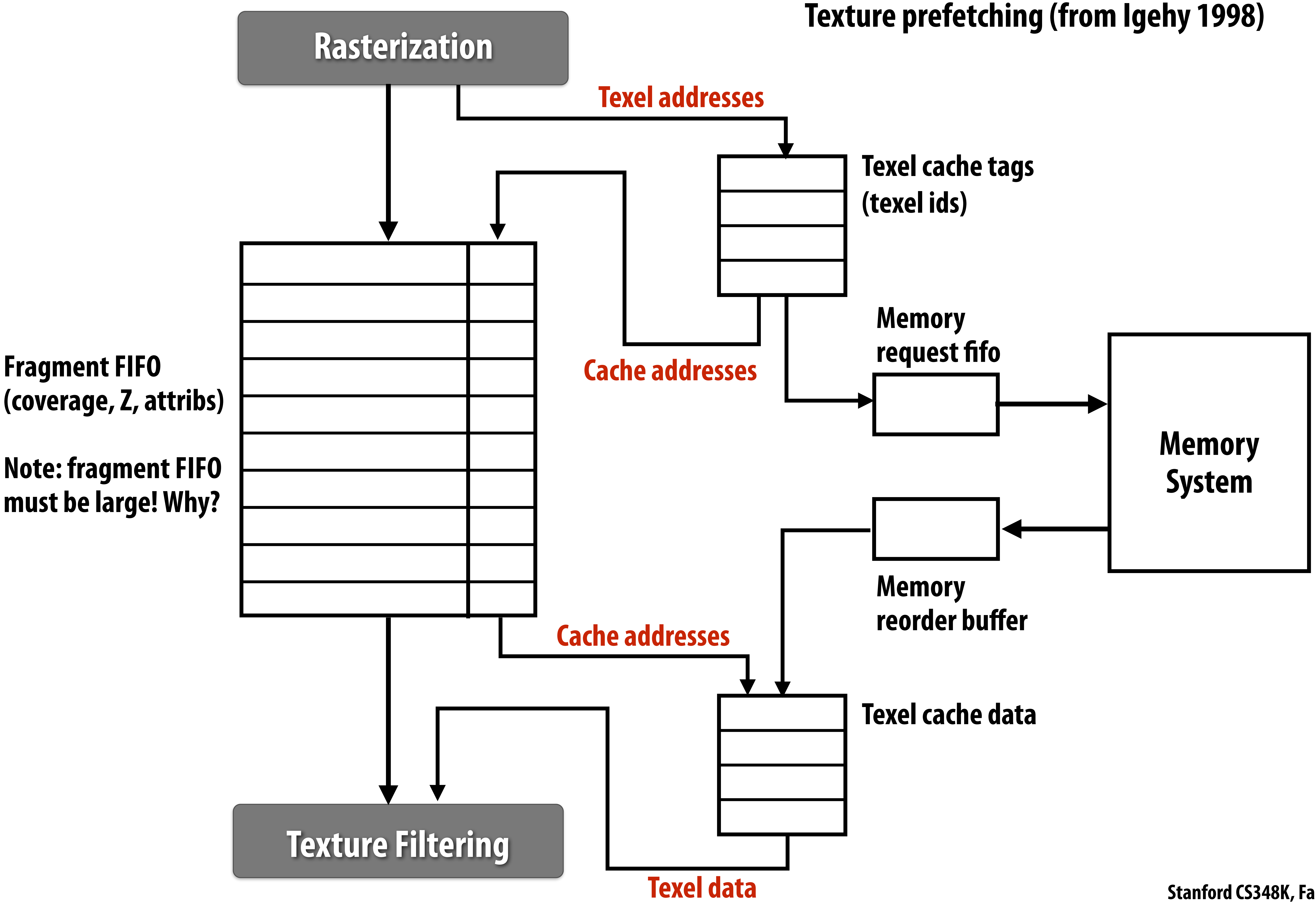
- Processor requests filtered texture data → processor waits hundreds of cycles (significant loss of performance)

- 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

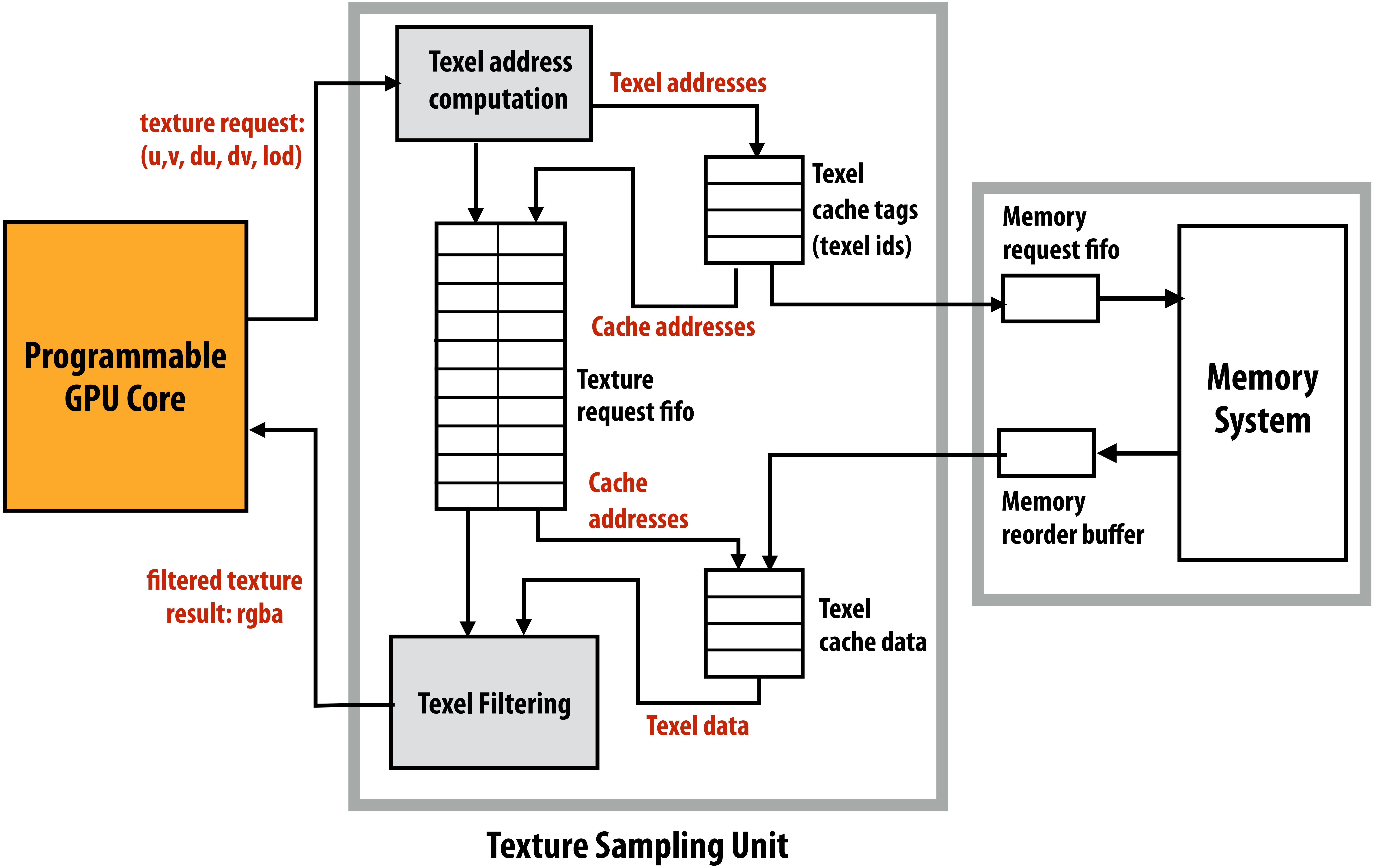
Texture prefetching (from Igehy 1998)



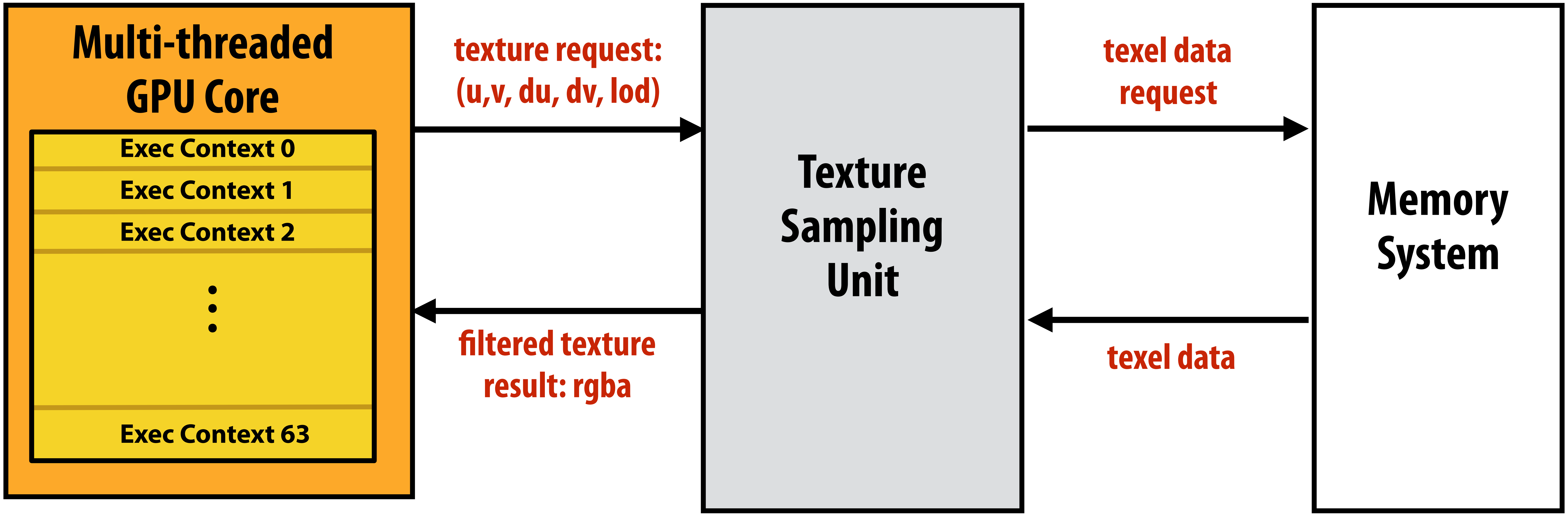
Fragment FIFO
(coverage, Z, attribs)

Note: fragment FIFO
must be large! Why?

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's prefetching design is now represented by live fragment state in the programmable core (a GPU "thread")

Texture compression (reducing bandwidth cost)

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)

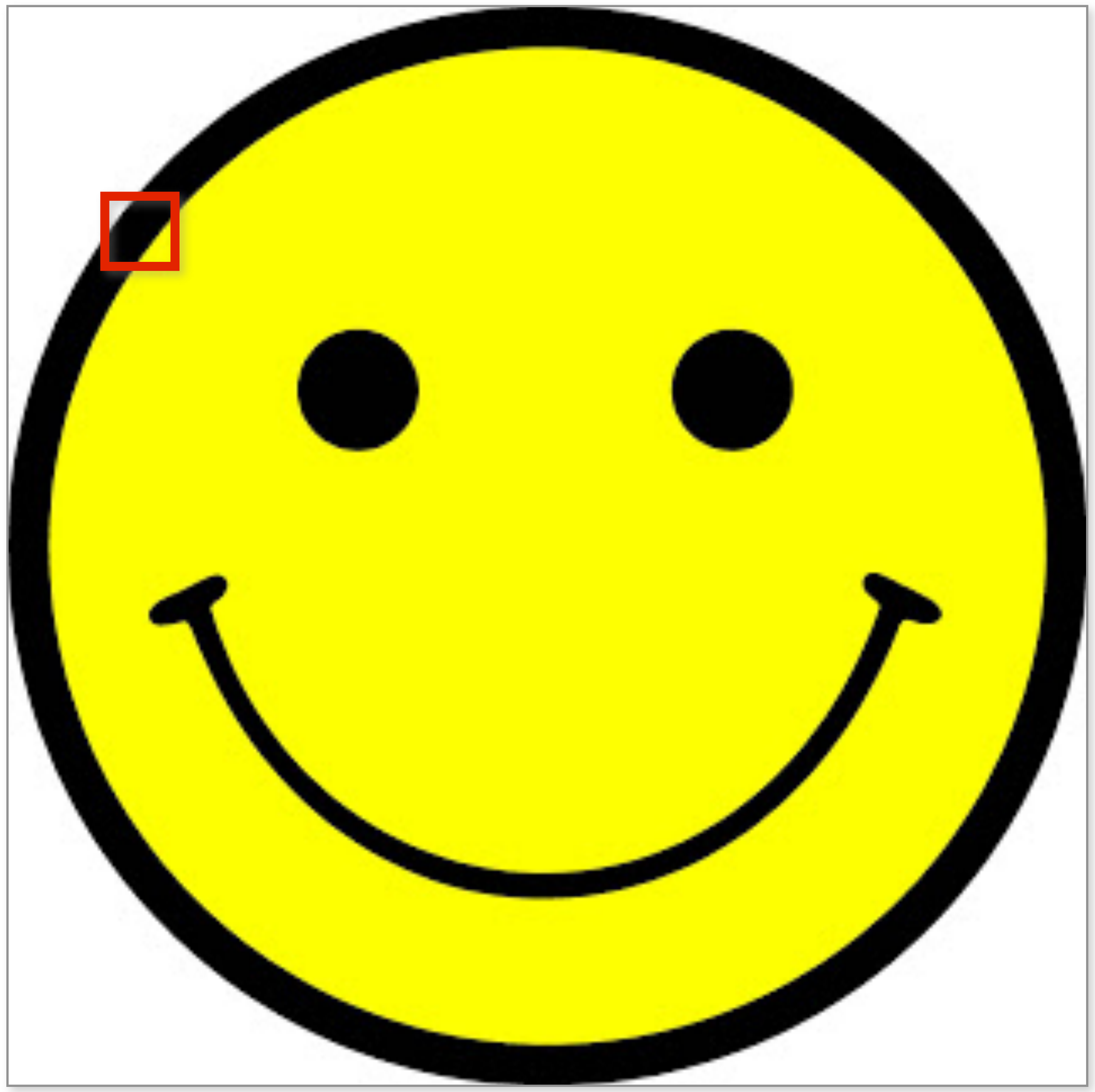
** May involve wrap, clamp, etc. of texel coordinates according to sampling mode configuration

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)

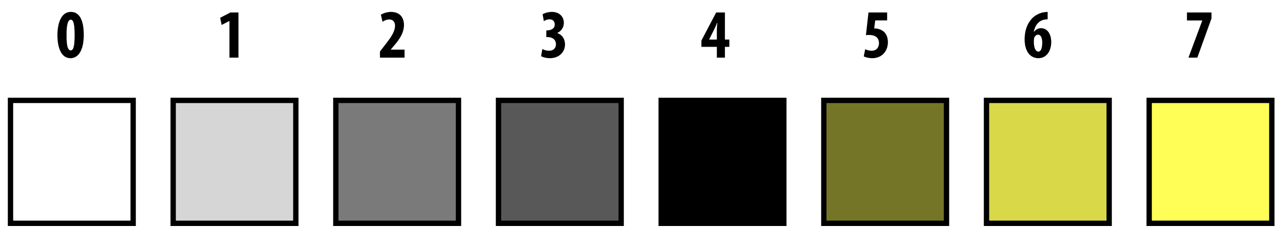
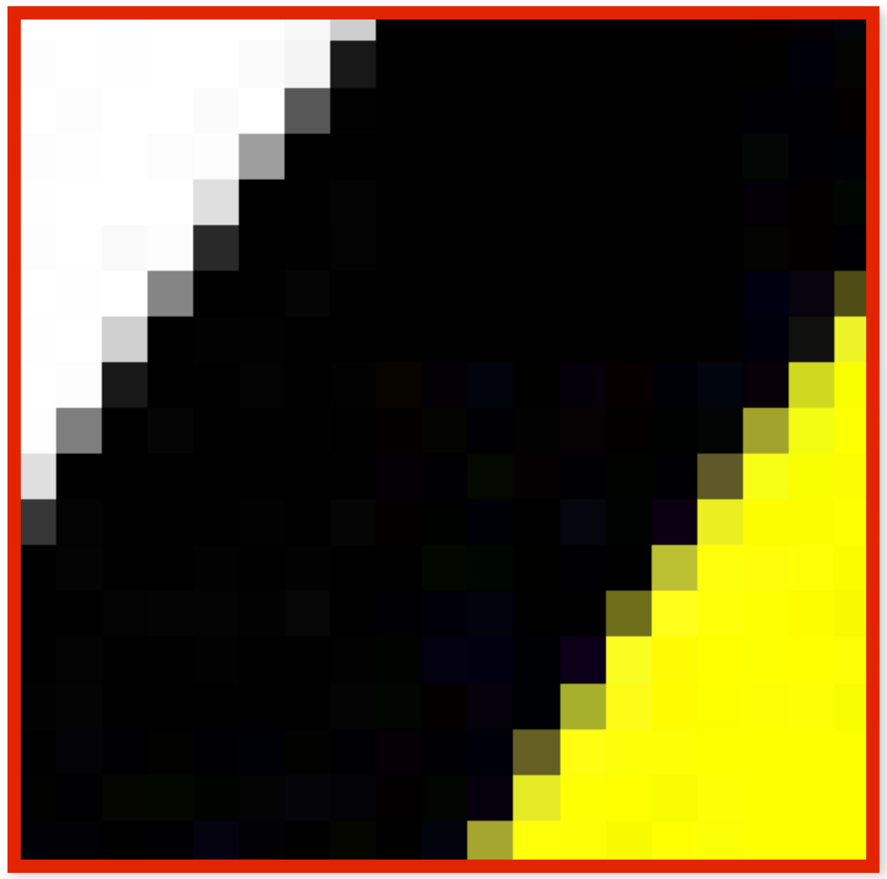


Image encoding in this example:

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)

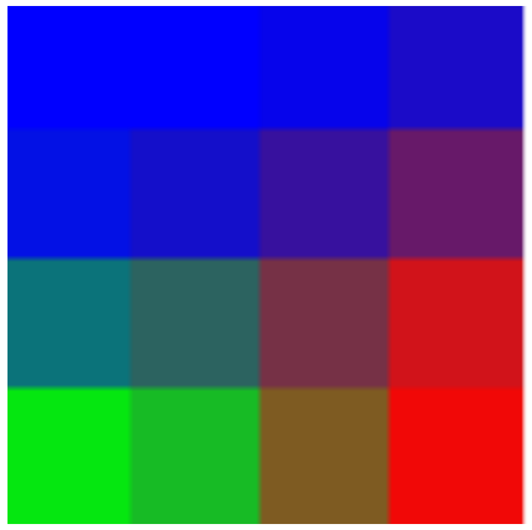
- **Palette of four colors encoded in four bytes:**
  - Two low-precision base colors: C_0 and C_1 (2 bytes each: RGB 5-6-5 format)
  - Other two colors computed from base values
    - $^1/_3C_0 + ^2/_3C_1$
    - $^2/_3C_0 + ^1/_3C_1$

- **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 $^1/_2C_0 + ^1/_2C_1$ and fourth color is transparent black

S3TC artifacts

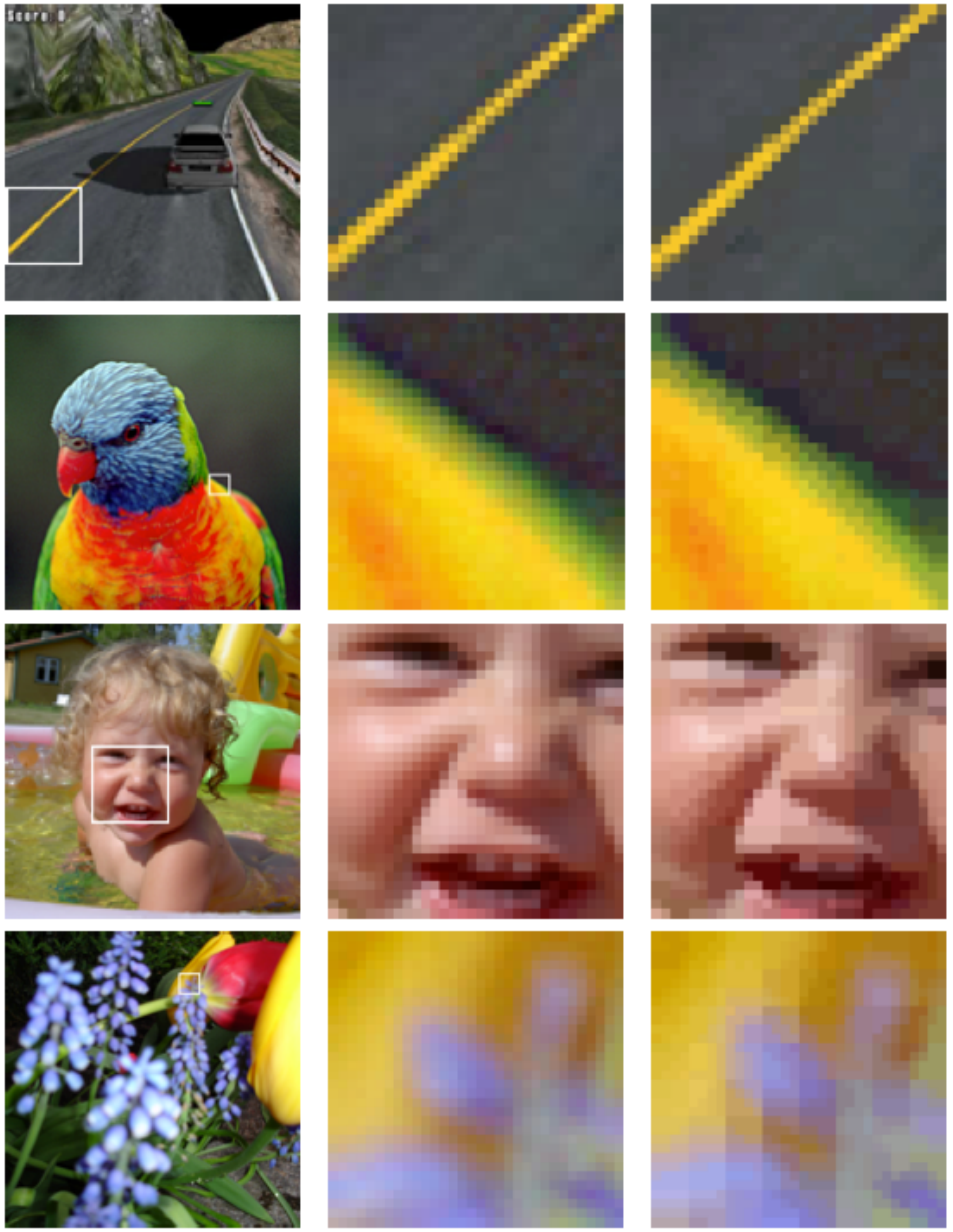


Original data

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)



Original

Original (Zoom)

S3TC

[Strom et al. 2007]

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

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]];
```

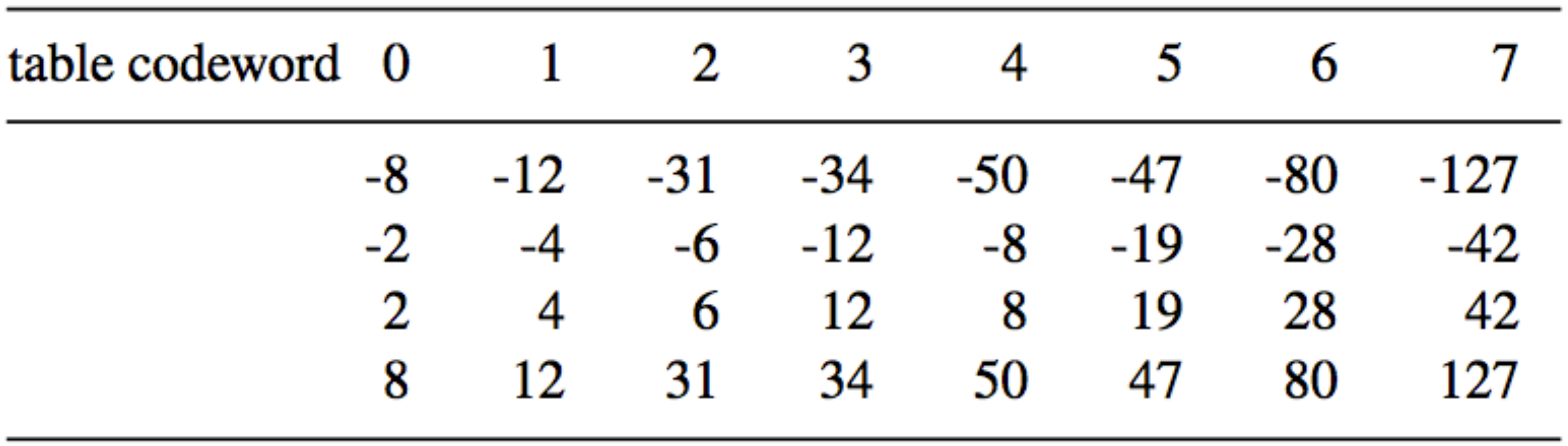


table codeword	0	1	2	3	4	5	6	7
	-8	-12	-31	-34	-50	-47	-80	-127
	-2	-4	-6	-12	-8	-19	-28	-42
	2	4	6	12	8	19	28	42
	8	12	31	34	50	47	80	127

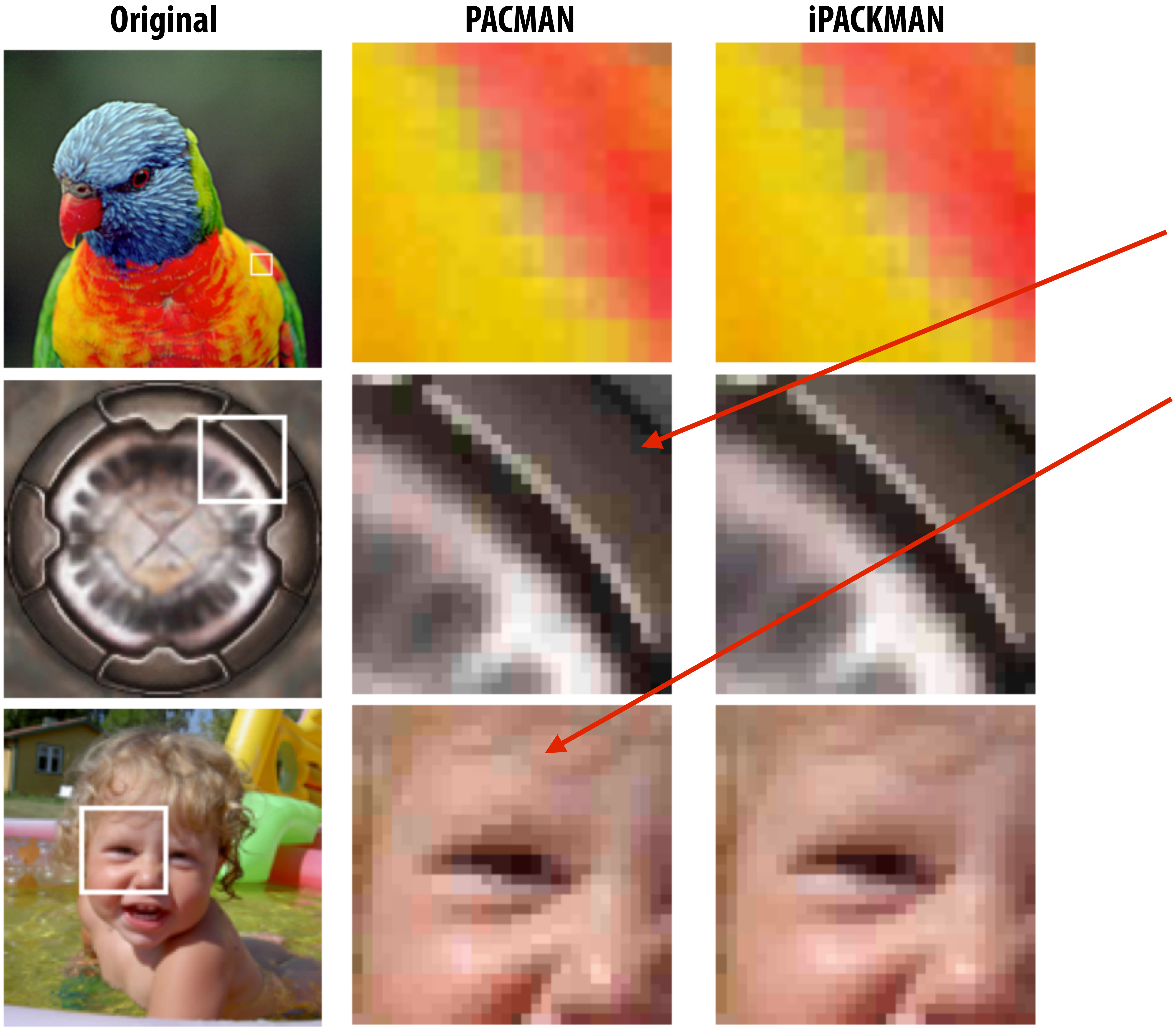
Example codebook for modulation tables (8 of 16 tables shown)

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



Original

PACMAN

iPACKMAN

Chrominance banding

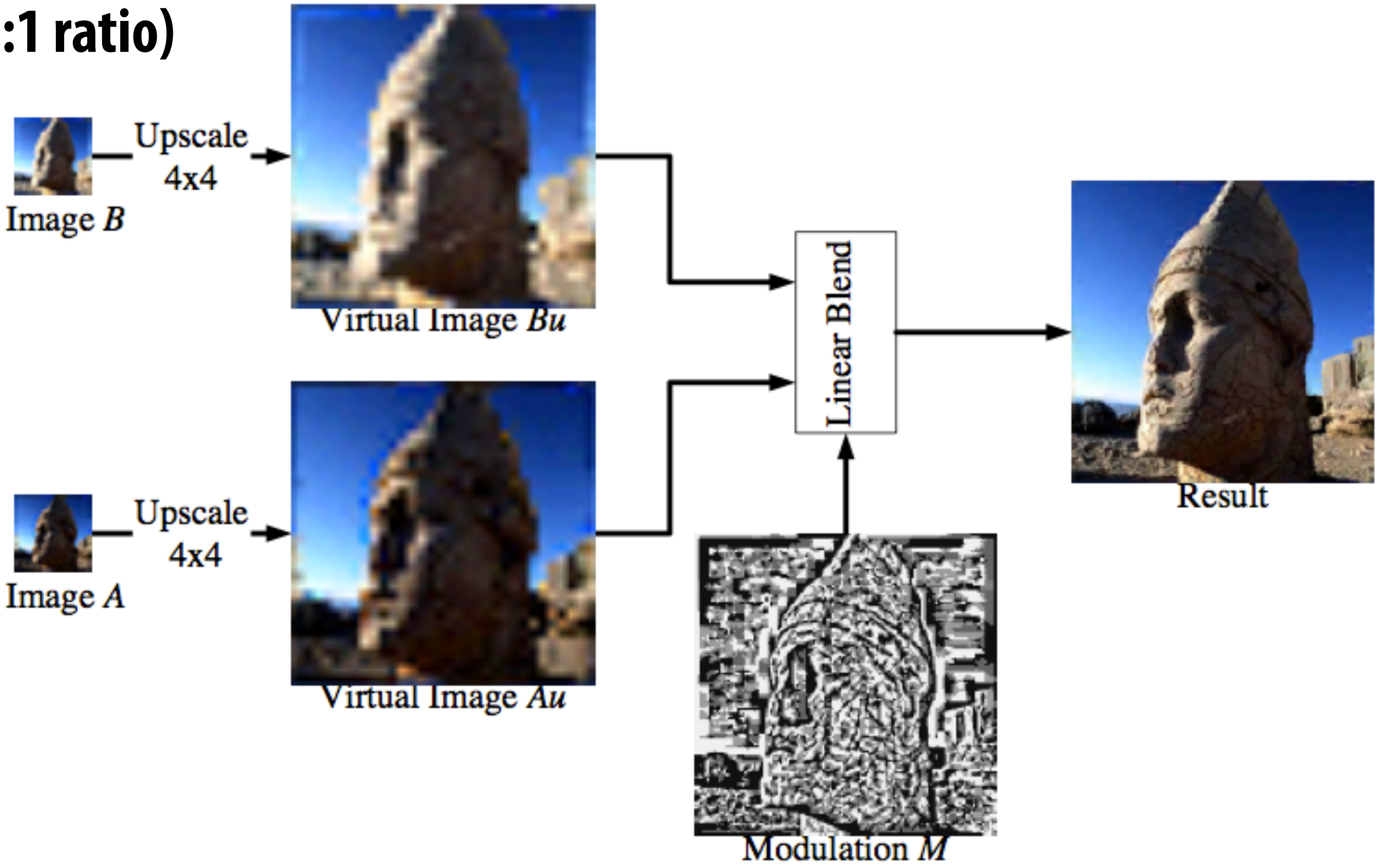
Chrominance block artifact

Image credit: Strom et al. 2005

PVRTC (Power VR texture compression)

[Fenney et al. 2003]

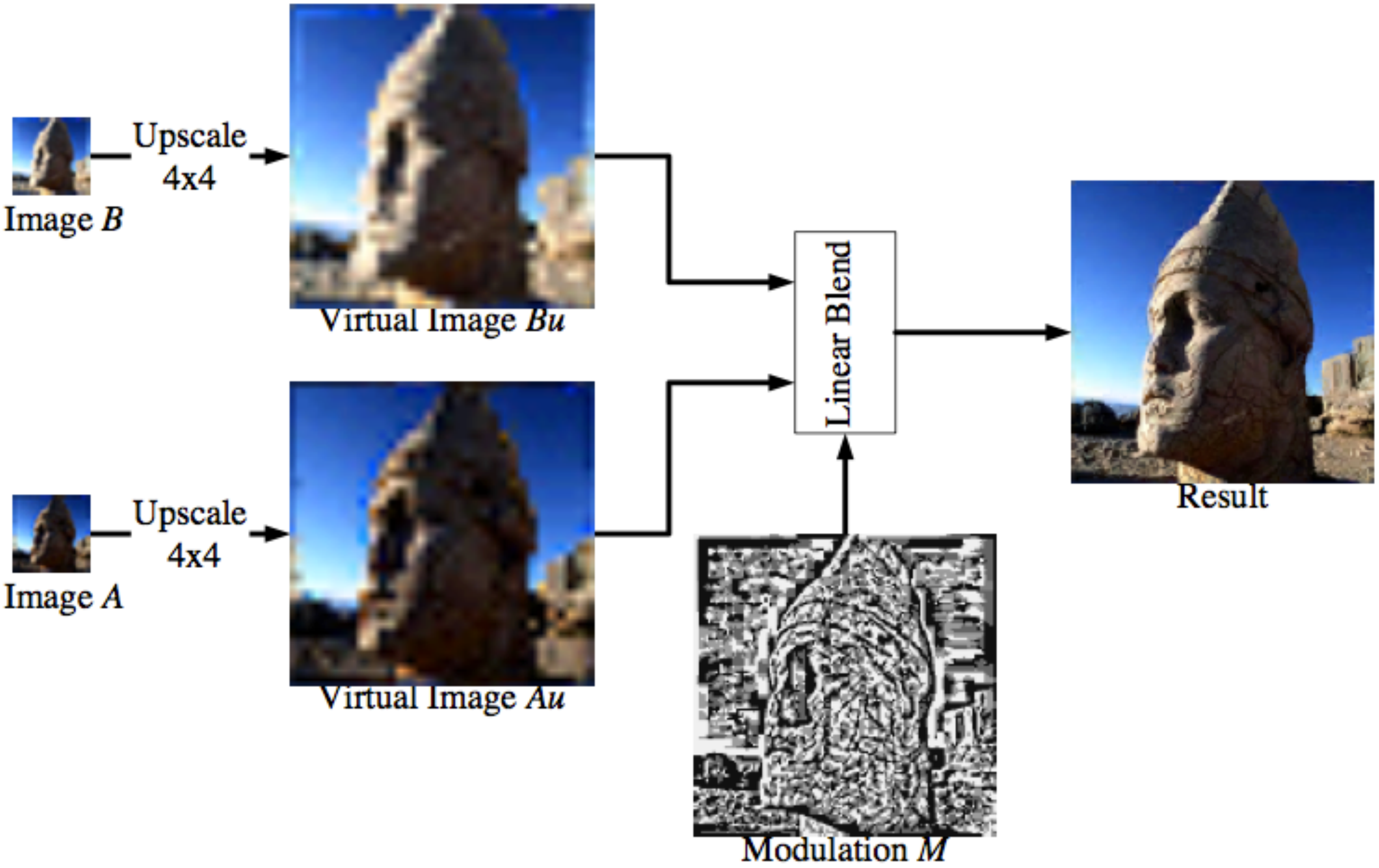
- Not a block-based format
  - Used in Imagination PowerVR GPUs
- Store low-frequency base images A and B
  - Base images downsampled by factor of 4 in each dimension ($^{1}/_{16}$ fewer texels)
  - Store base image pixels in RGB 5:5:5 format (+ 1 bit alpha)
- Store 2-bit modulation factor per texel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

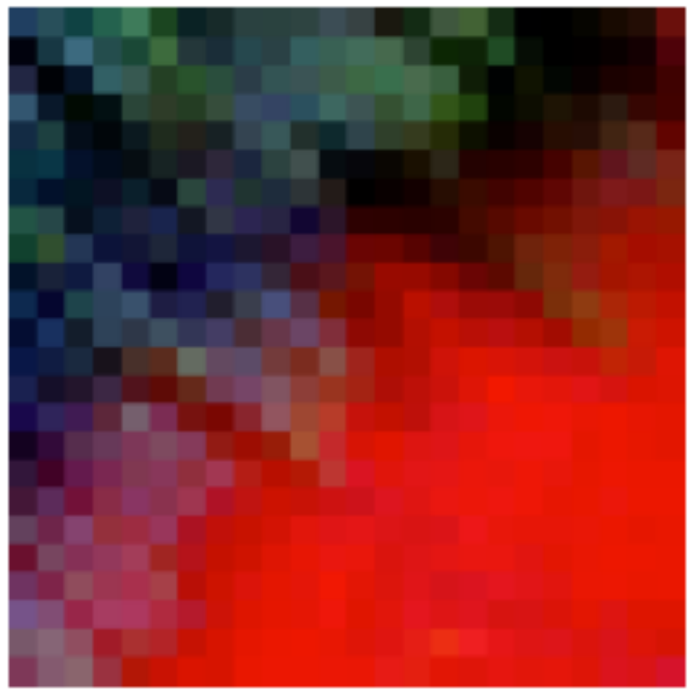


PVRTC avoids blocking artifacts

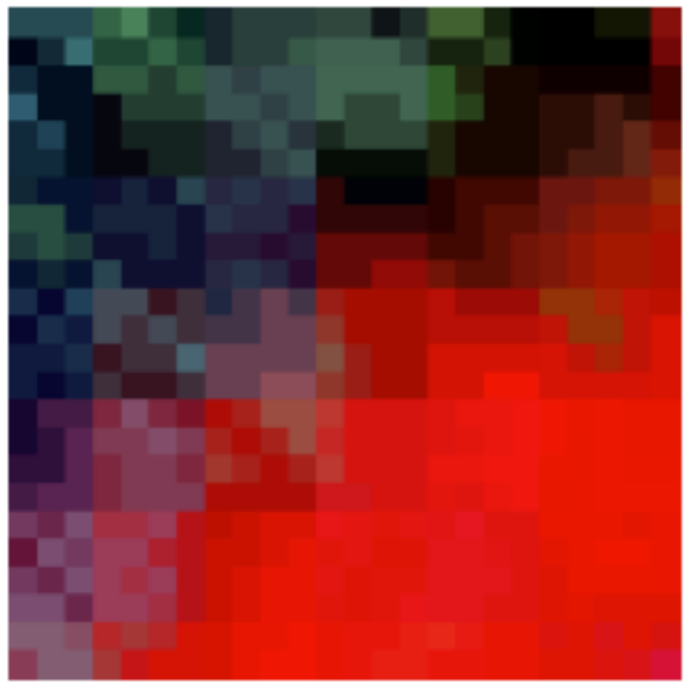
Because it is not block-based

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

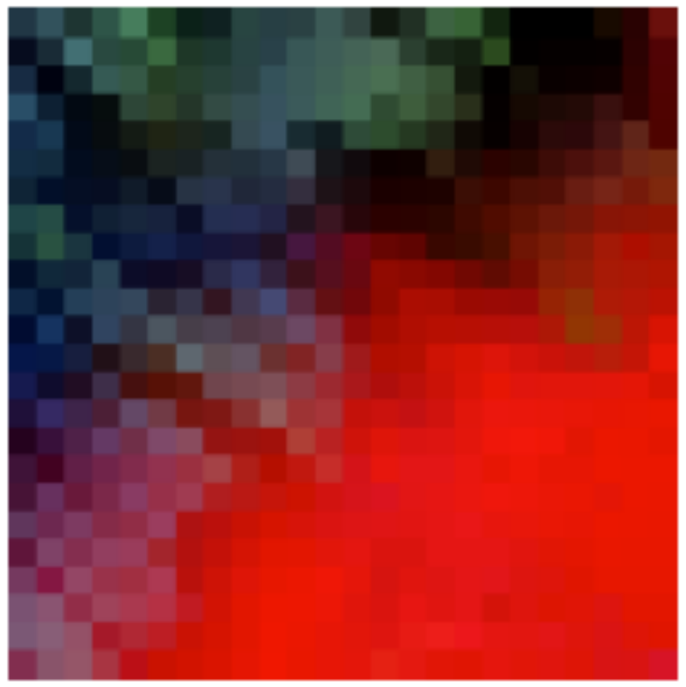
(Followed by a weighted combination of the two images)



Original



S3TC



4bpp PVRTC

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)

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)

Bandwidth reduction techniques for frame-buffer access

From last time: occlusion via the depth buffer

```
bool pass_depth_test(d1, d2) {
   return d1 < d2;
}

depth_test(tri_d, tri_color, x, y) {

  if (pass_depth_test(tri_d, zbuffer[x][y]) {

    zbuffer[x][y] = tri_d;      // update zbuffer
    color[x][y] = tri_color;    // update color buffer
  }
}
```

Z-buffer algorithm has high bandwidth requirements

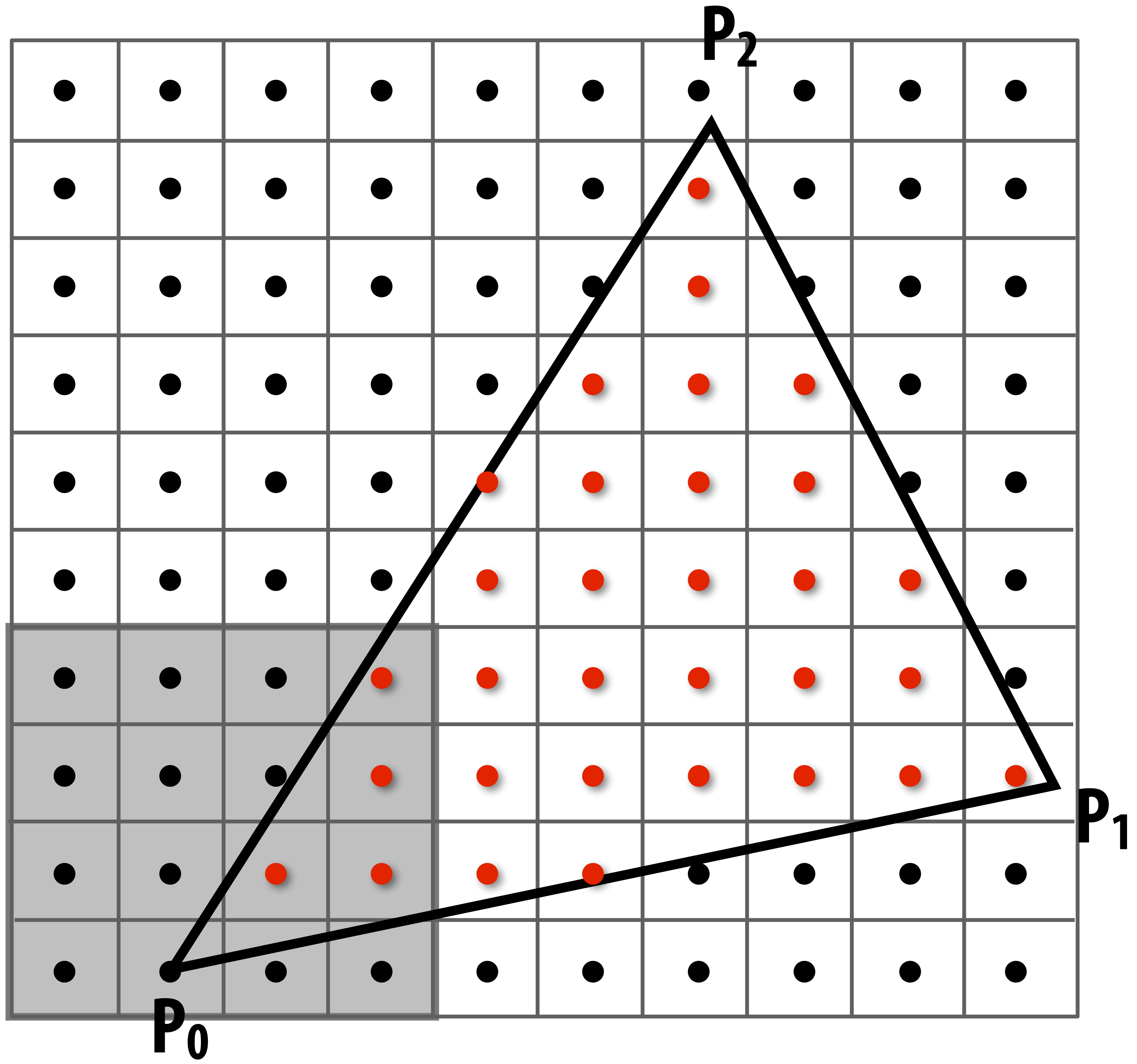
- **Particularly when super-sampling triangle coverage)**
  - Number of Z-buffer reads/writes for a frame depends on:
    - Depth complexity of the scene
    - The order triangles are provided to the graphics pipeline
      (if depth test fails, don't write to depth buffer or rgba)

- **Bandwidth estimate:**
  - 60 Hz x 2 MPixel image x avg. depth complexity 4 (assume: replace 50% of time) x 32-bit Z = 2.8 GB/s
  - If super-sampling at 4 times per pixel, multiply by 4
  - Consider five shadow maps per frame (1 MPixel, not super-sampled): additional 8.6 GB/s
  - Note: this is just depth accesses. It does not include color-buffer bandwidth

- **Modern GPUs implement caching and lossless compression of both color and depth buffers to reduce bandwidth (coming slides)**

Hierarchical early occlusion culling: "hi-Z"

Rasterize triangles in tiles (recall benefit to texture caching)



Z-Max culling:

For each screen tile, compute farthest value in the depth buffer: `z_max`

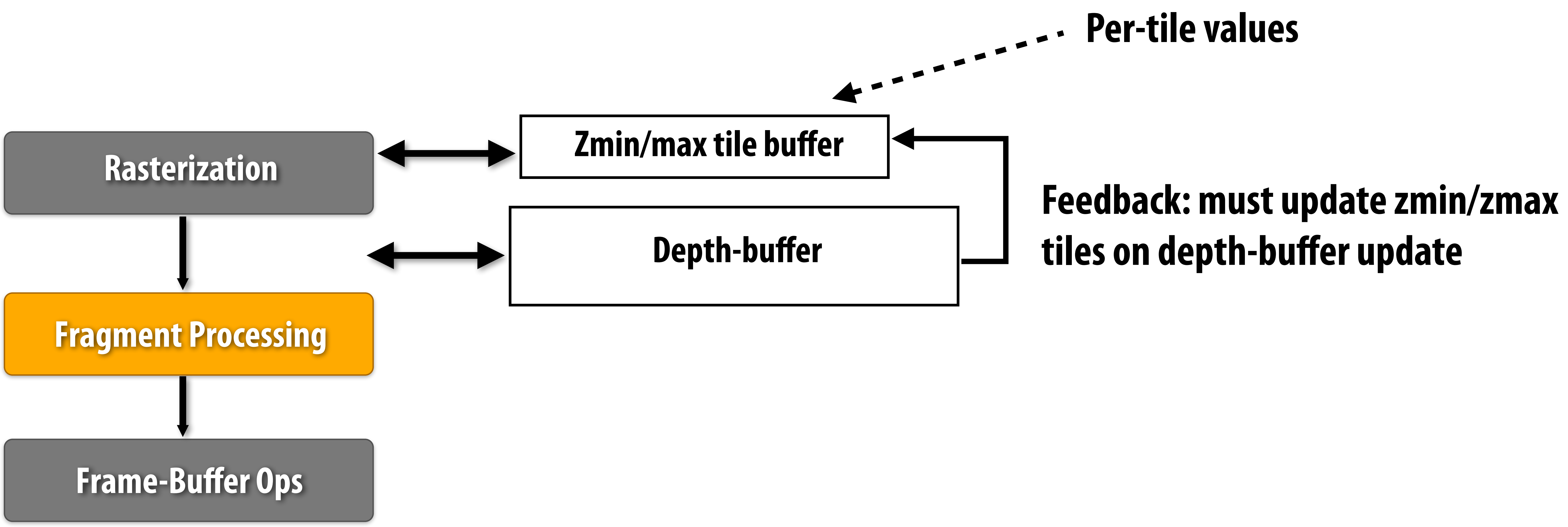
During traversal, for each tile:

1. Compute closest point on triangle in tile within screen region of tile: `tri_min`
2. If `tri_min > z_max`, then triangle is completely occluded in this tile. (The depth test will fail for all samples in the tile.) Proceed to next tile without performing coverage tests for individual samples in tile.

Z-min optimization:

Depth-buffer also stores `z_min` for each tile.
If `tri_max < z_min`, then all depth tests for fragments in tile will pass. (No need to perform depth test on individual fragments!)

Hierarchical Z + early Z-culling



Remember: these are GPU implementation details (common optimizations performed by most GPUs). They are invisible to the programmer and not reflected in the graphics pipeline abstraction

Depth-buffer compression

Depth-buffer compression

- **Motivation: reduce bandwidth required for depth-buffer accesses**
  - **Worst-case (uncompressed) buffer allocated in DRAM**
  - **Conserving memory <u>footprint</u> is a non-goal**

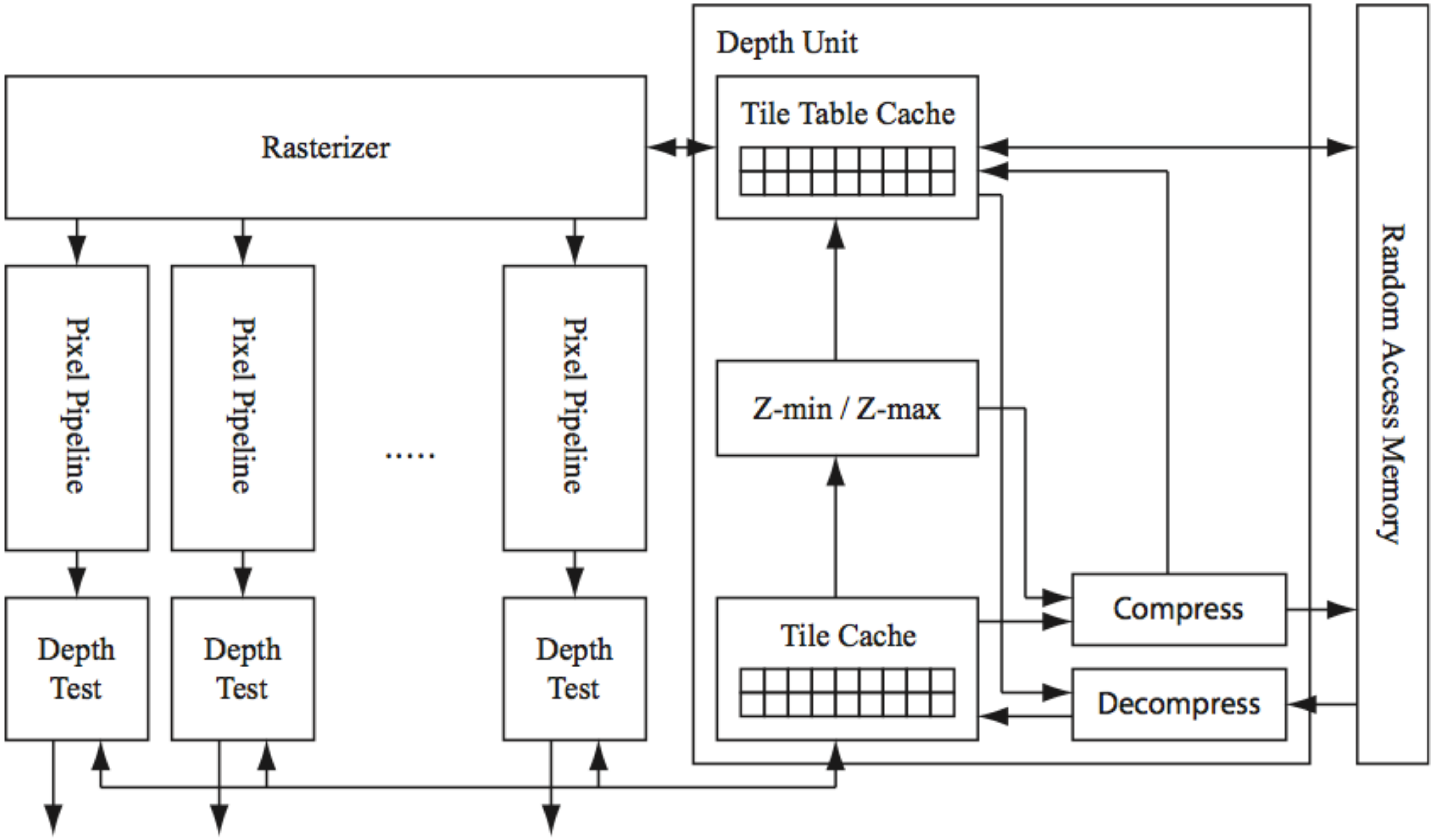
    (Need for real-time guarantees in graphics applications requires application to plan for worst case anyway)

- **Requires lossless compression**
  - **Question: why not lossy?**

- **Designed for fixed-point numbers (fixed-point math in rasterizer)**

Depth-buffer compression is tile based

Main idea: exploit similarity of values within a screen tile



On tile evict:

1. Compute zmin/zmax (needed for hierarchical culling and/or compression)
2. Attempt to compress
3. Update tile table
4. Store tile to memory

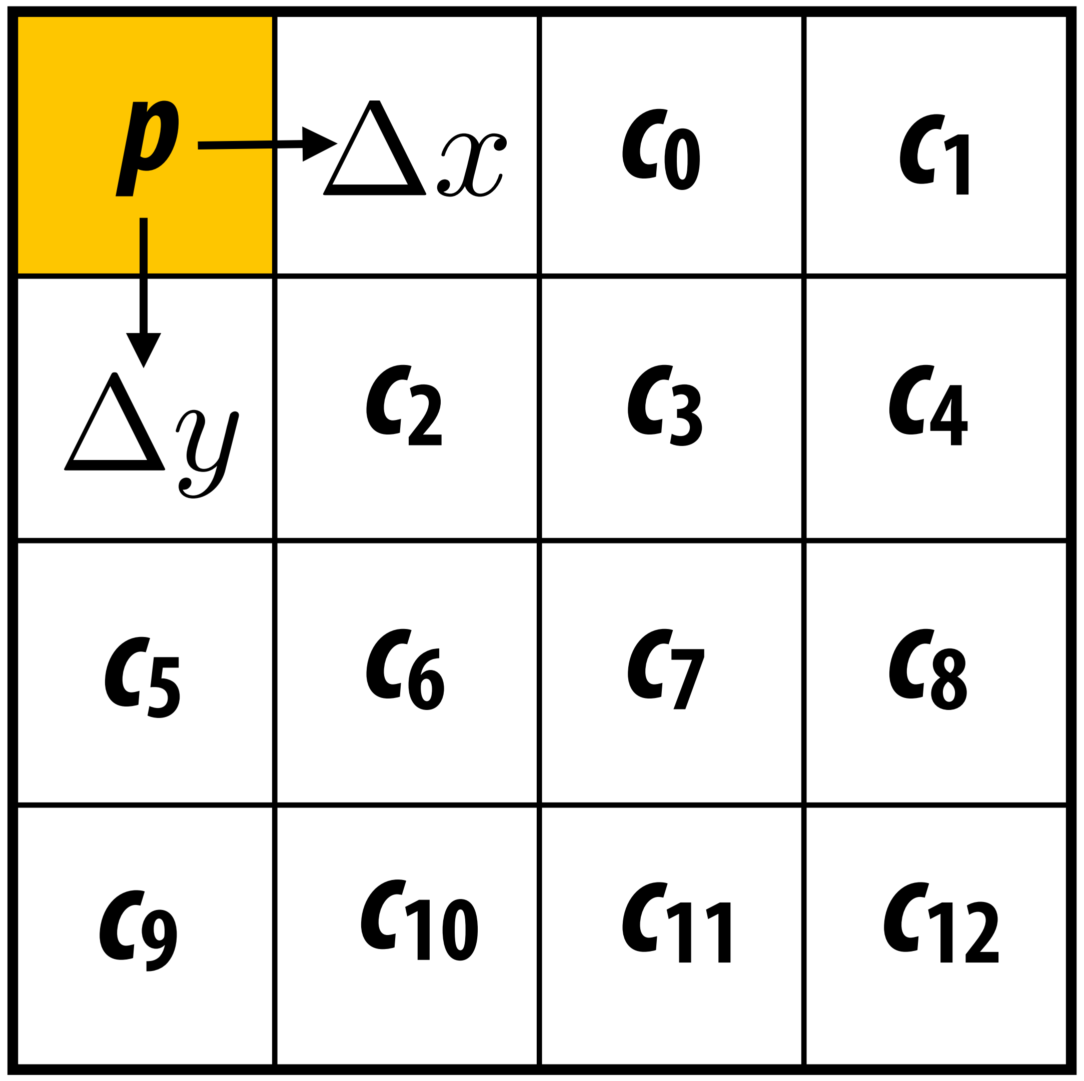
On tile load:

1. Check tile table for compression scheme
2. Load required bits from memory
3. Decompress into tile cache

Figure credit: [Hasselgren et al. 2006]

Anchor encoding

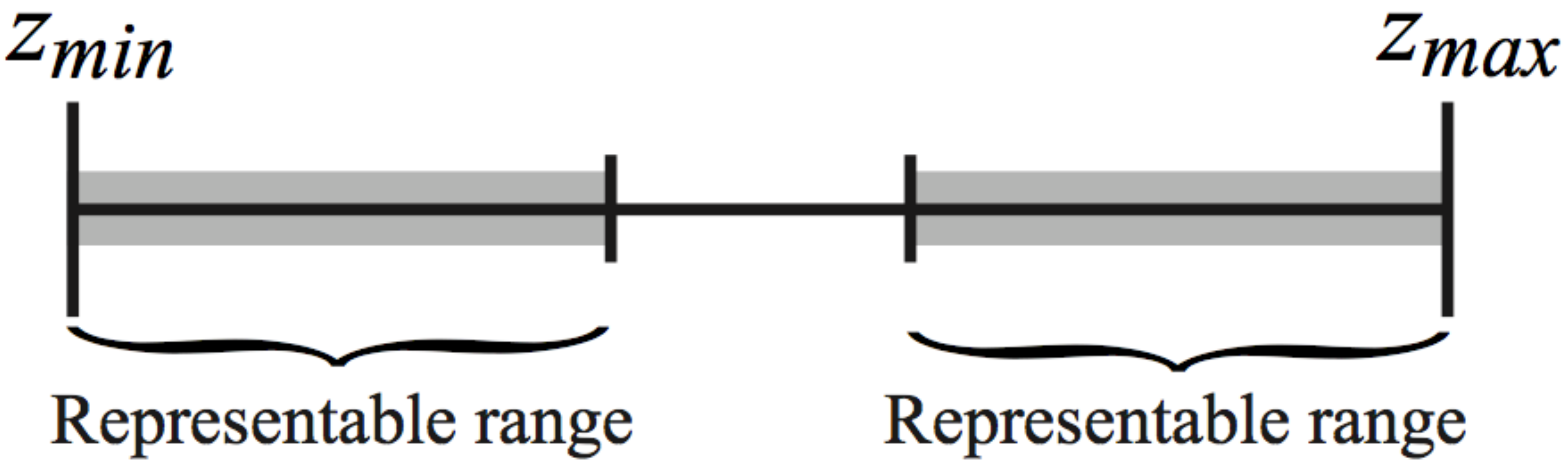
- Store value of "anchor pixel" *p* and compute Δx and Δy of adjacent pixels (fit a plane to the data)
- Predict color of other pixels in tile based on offset from anchor
  - $\text{value}(i,j) = p + i\Delta x + j\Delta y$
- Store "correction" c_i on prediction at each pixel
- Scheme (for 24-bit depth buffer)
  - Anchor: 24 bits (full resolution)
  - DX, DY: 15 bits
  - Per-sample offsets: 5 bits



[Van Dyke and Margeson]

Depth-offset compression

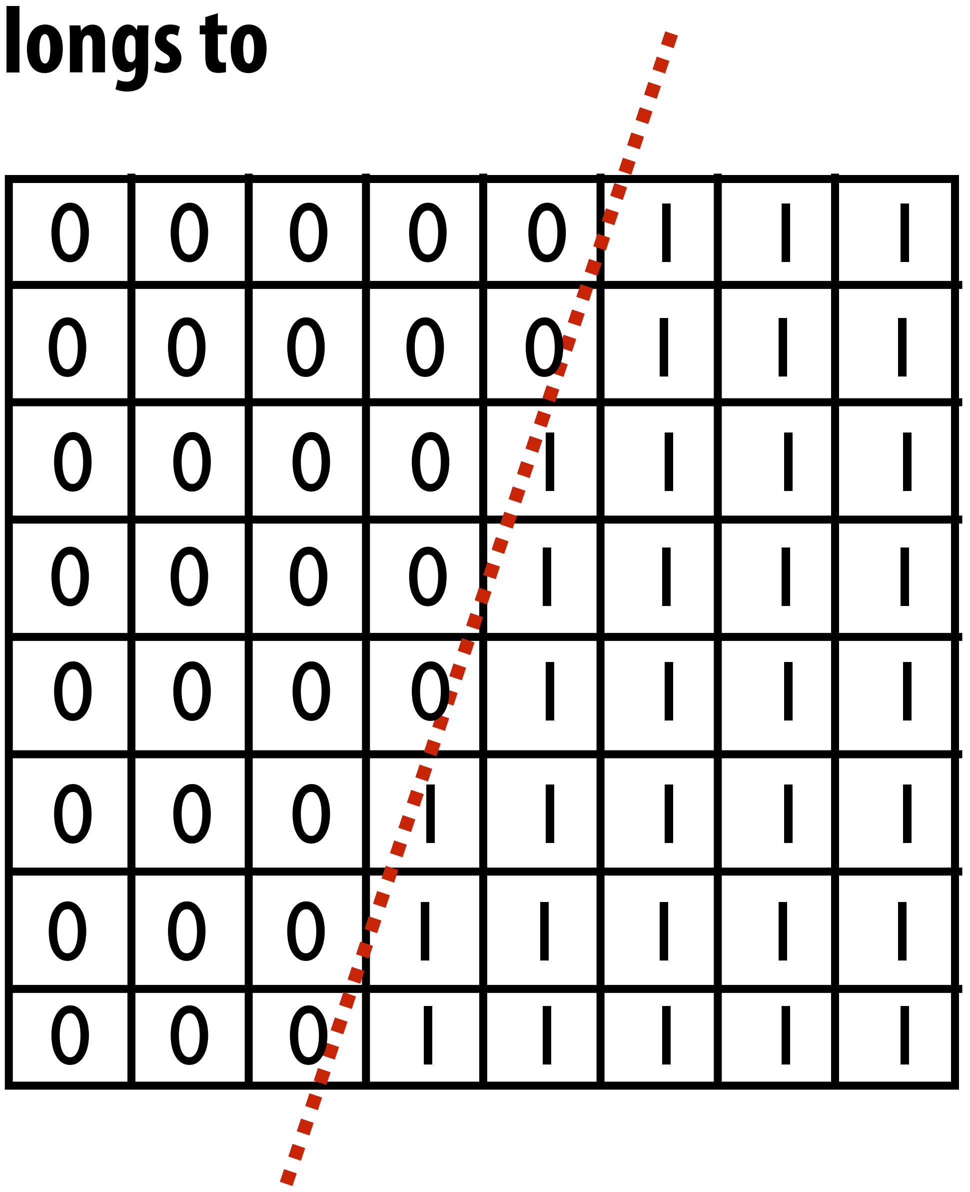
- Assume depth values have low dynamic range relative to tile's zmin and zmax (assume two surfaces)
- Store zmin/zmax (need to anyway for hierarchical Z)
- Store low-precision (8-12 bits) offset value for each sample
  - MSB encodes if offset is from zmin or zmax



[Morein and Natali]

Explicit plane encoding

- Do not attempt to learn prediction plane from depths, just store the plane equation for the triangle directly
  - Store triangle plane equation in tile
  - Store bit per sample indicating whether sample is covered by triangle
  - Evaluate plane equation as necessary to “decompress”
- Simple extension to multiple triangles per tile:
  - Store up to N plane equations in tile
  - Store $\log_2(N)$ bit id per depth sample indicating which triangle it belongs to
- When new triangle contributes coverage to tile:
  - Add new plane equation if storage is available, else decompress
- To decompress:
  - For each sample, evaluate Z(x,y) for appropriate plane



0	0	0	0	0	1	1	1
0	0	0	0	0	1	1	1
0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
0	0	0	1	1	1	1	1
0	0	0	1	1	1	1	1
0	0	0	1	1	1	1	1

"Memory transaction elimination" in ARM GPUs

- Writing pixels in output image is a bandwidth-heavy operation
- Idea: skip output image write if it is unnecessary (color buffer compression!)
  - Frame 1:
    - Render frame tile at a time
    - Compute hash of pixels in each tile on screen
  - Frame 2:
    - Render frame tile at a time
    - Before storing pixel values for tile to memory, compute hash and see if tile is the same as last frame
      - If yes, skip memory write

Slow camera motion: 96% of writes avoided

Fast camera motion: ~50% of writes avoided

[Source: Tom Olson http://community.arm.com/groups/arm-mali-graphics/blog/2012/08/17/how-low-can-you-go-building-low-power-low-bandwidth-arm-mali-gpus]

Summary: reducing the bandwidth requirements of depth testing

- Caching: access DRAM less often (by caching depth buffer data)
- Hierarchical Z techniques (zmin/zmax culling): "early outs" result in accessing individual sample data less often
- Data compression: reduce number of bits that must be transferred from memory to read/write a depth sample

---

- The pipeline's output color buffer (output image) is also compressed using similar techniques
  - Depth buffer typically achieves higher compression ratios than color buffer. Why?

Cross-cutting issues

- **Hierarchical traversal during rasterization**
  - Leveraged to reduce number of coverage tests and depth buffer accesses
  - Tile size often coupled to hierarchical Z granularity
  - May also be coupled to compression tile granularity
  - Useful for improving texture cache hit rate

- **Hierarchical culling and plane-based buffer compression are most effective when triangles are reasonably large**
  - Modern GPU implementations are still optimized for triangles of area ~ tens of pixels