Lecture 5:
Texturing

Visual Computing Systems
CMU 15-869, Fall 2014

Today: texturing!

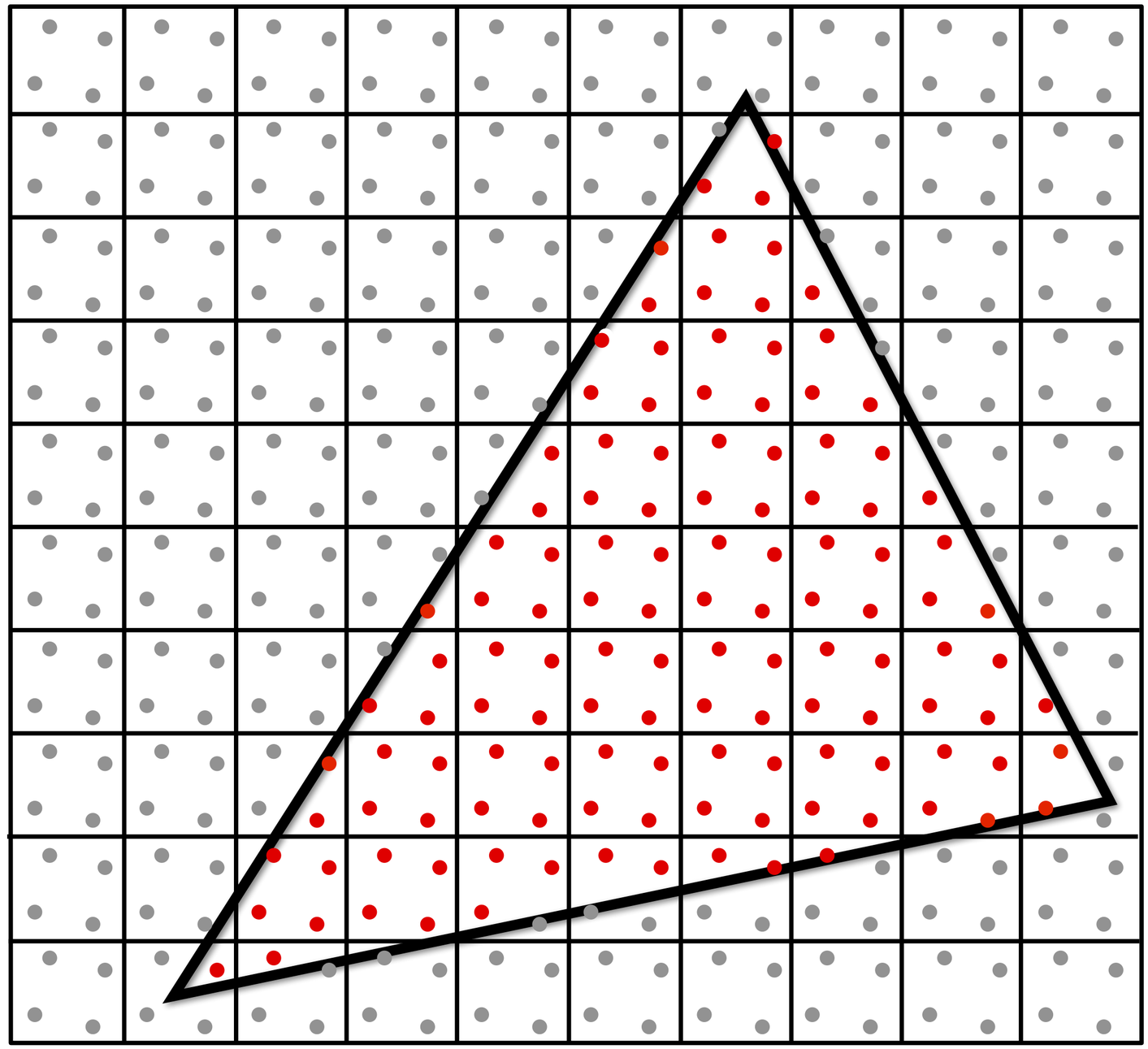
- **Texture filtering math**
  - At the very least… a texture access is not just a 2D array lookup ;-)

- **Memory-system implications of texture mapping operations**
  - Texture caching
  - Memory layout of texture data
  - Prefetching and multi-threading

Previous lecture (visibility)

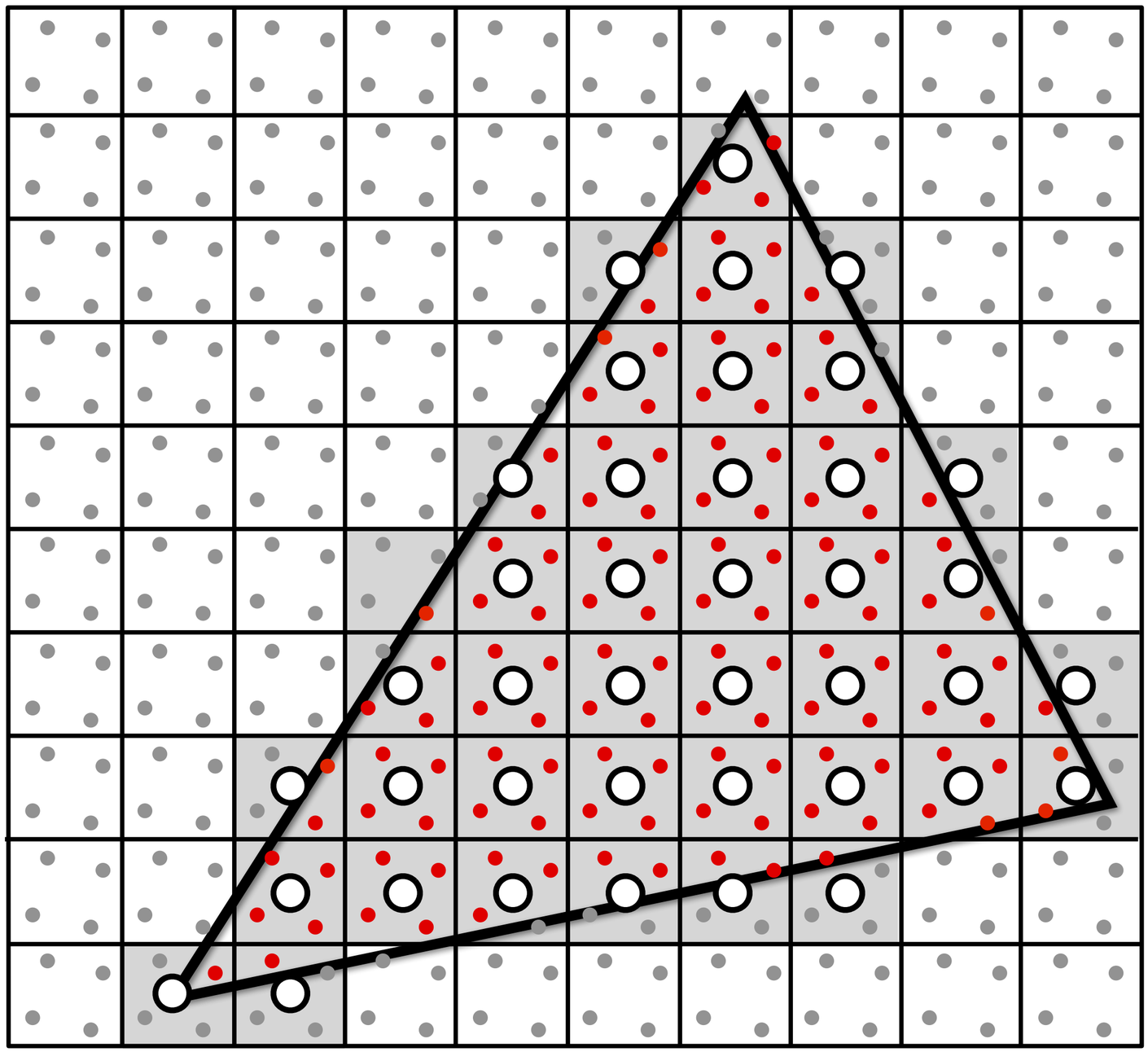
Rasterizer samples triangle-screen coverage (four samples per pixel shown here)

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

Shading a fragment

HLSL shader program: defines logic of fragment processing stage

```
sampler mySampler;
Texture2D<float3> myTex;
float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv)
{
  float3 kd;
  kd = myTex.Sample(mySampler, uv);
  kd *= clamp(dot(-lightDir, norm), 0.0, 1.0);
  return float4(kd, 1.0);
}
```

Sample surface albedo from texture

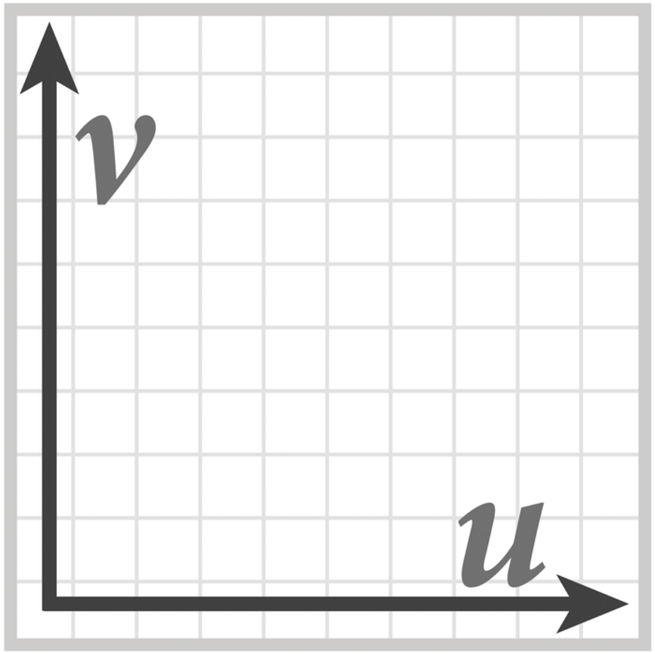
Shader returns surface reflectance

Modulate surface albedo by incident irradiance

Let:

`lightDir = [-1, -1, -1]`

`myTex =`



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

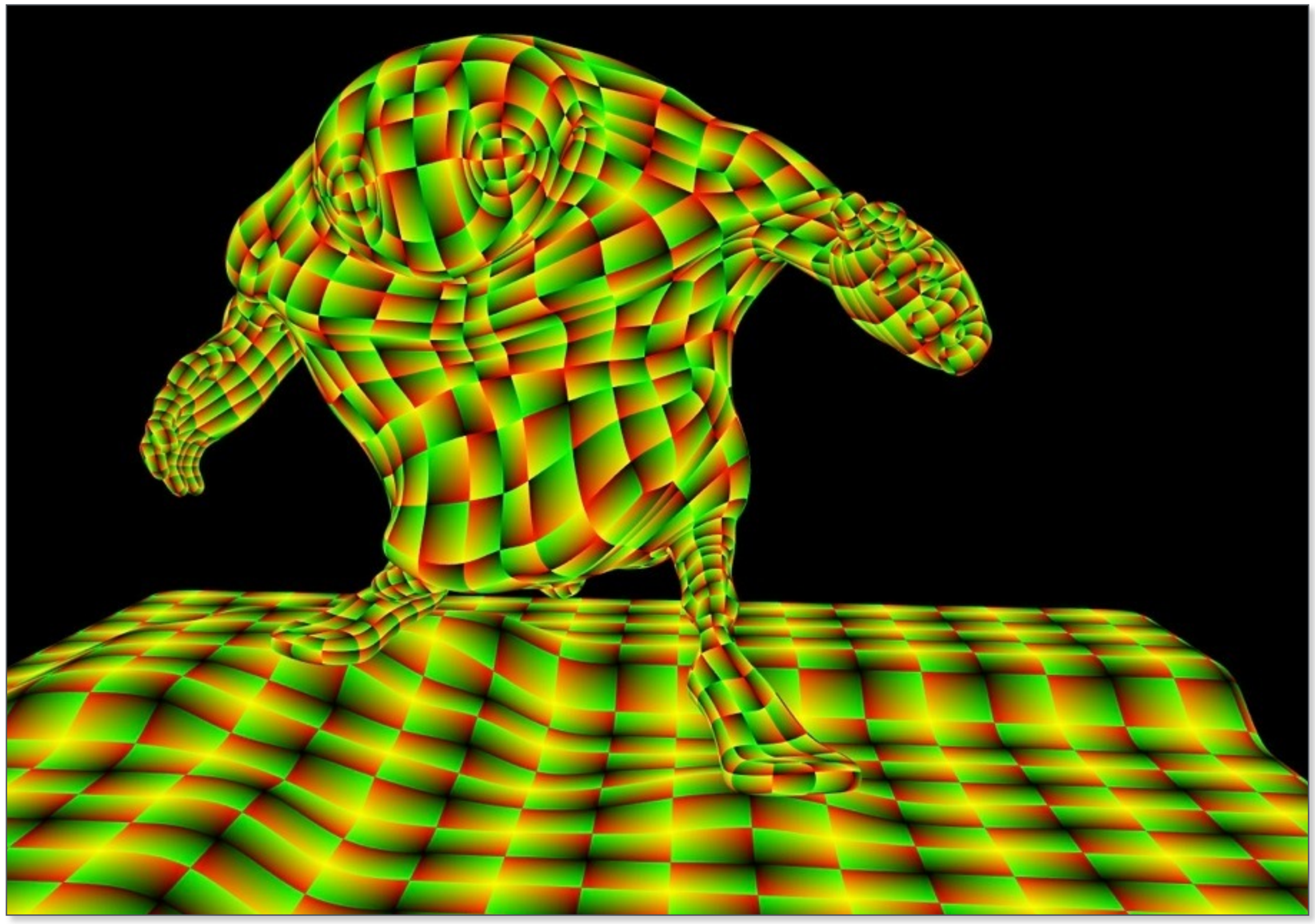
myTex : $[0,1]^2 \rightarrow$ float3

(represented by 2048x2048 image)

`mySampler` defines how to re-sample the texture image to generate sample at (u,v)

Texture coordinate visualization

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



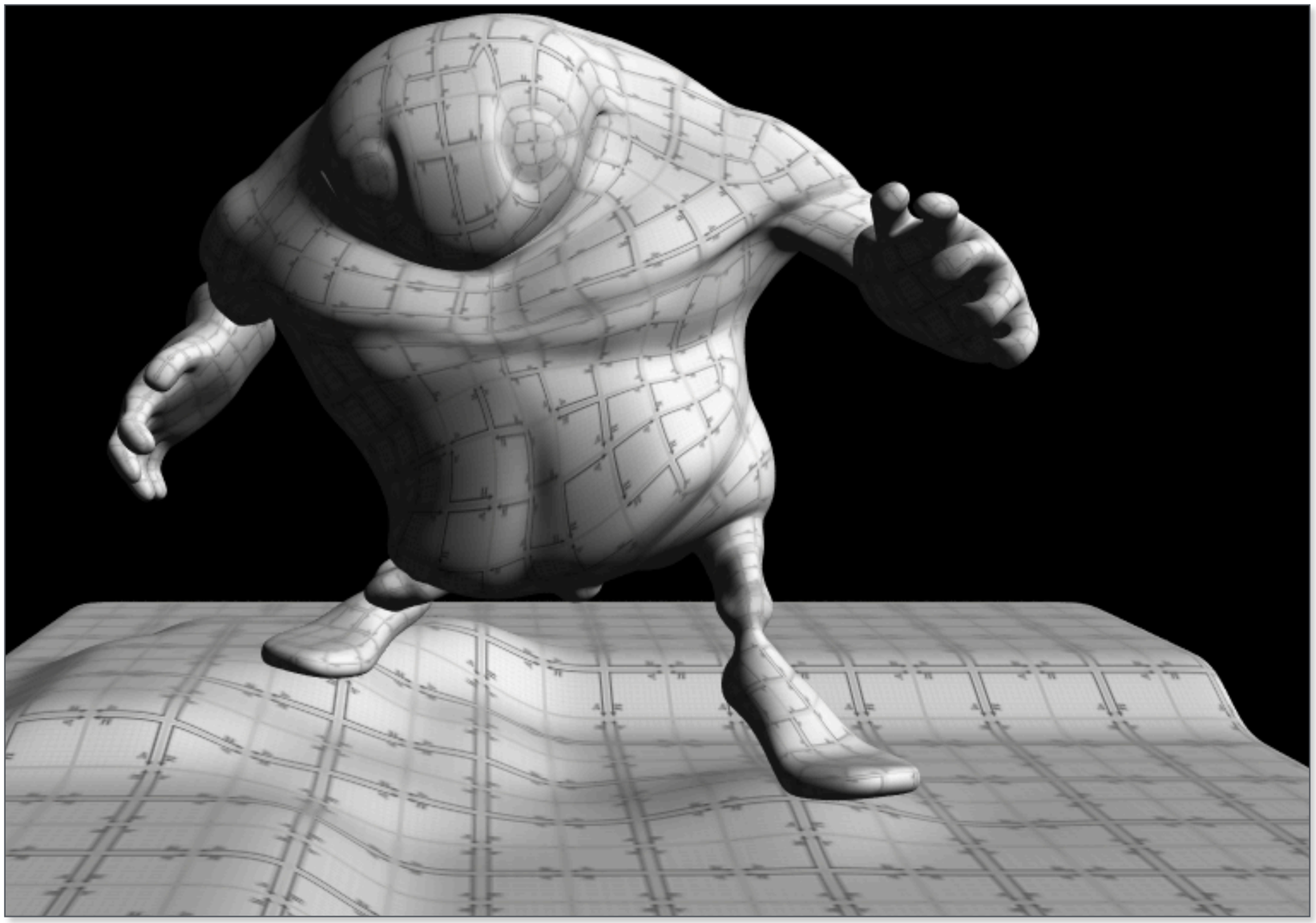
In this example, scene geometry is defined by 2D parametric surface patches

Red channel = u

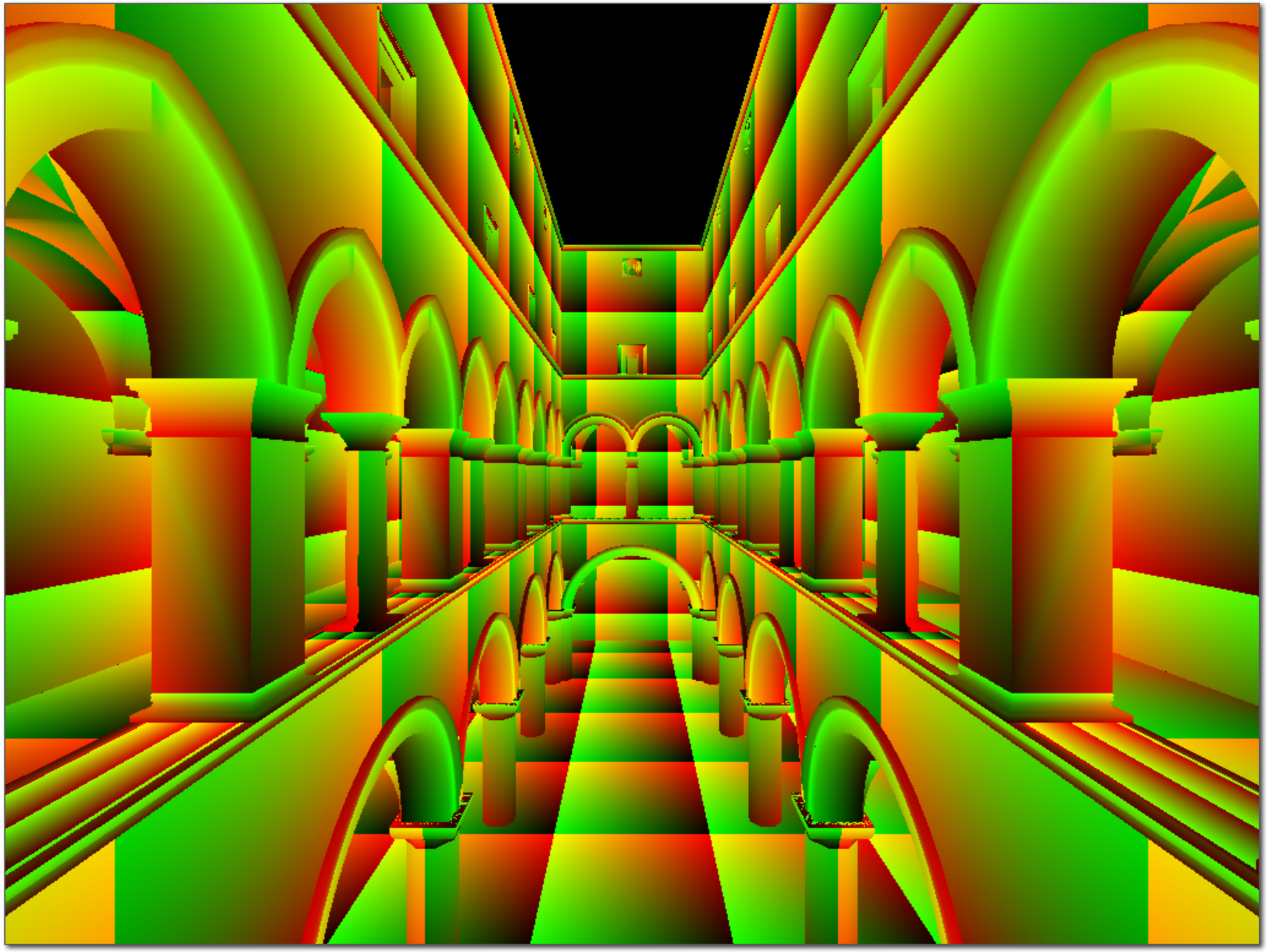
Green channel = v

So uv=(0,0) is black, uv=(1,1) is yellow

Final shaded result

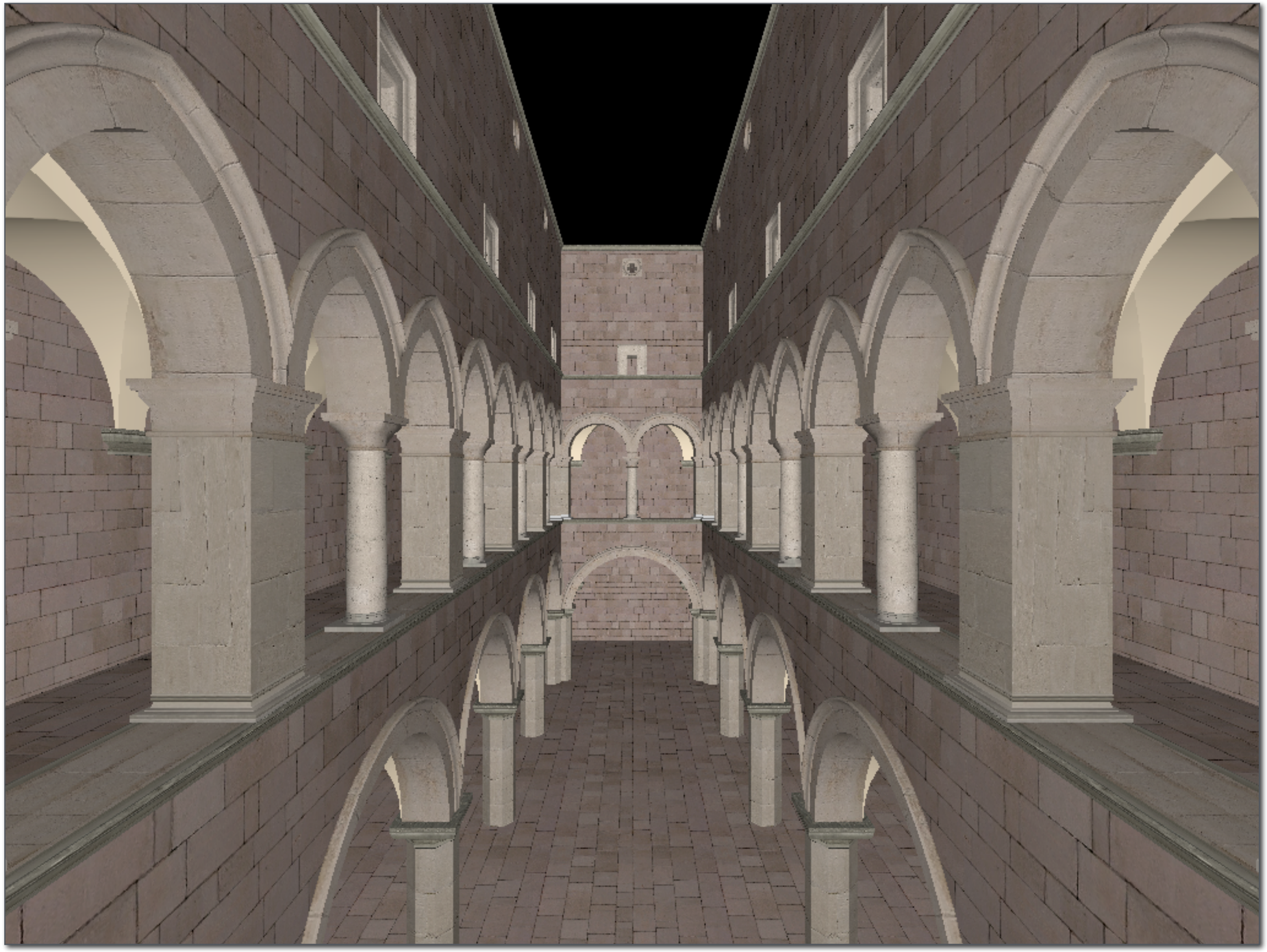


Another example: Sponza

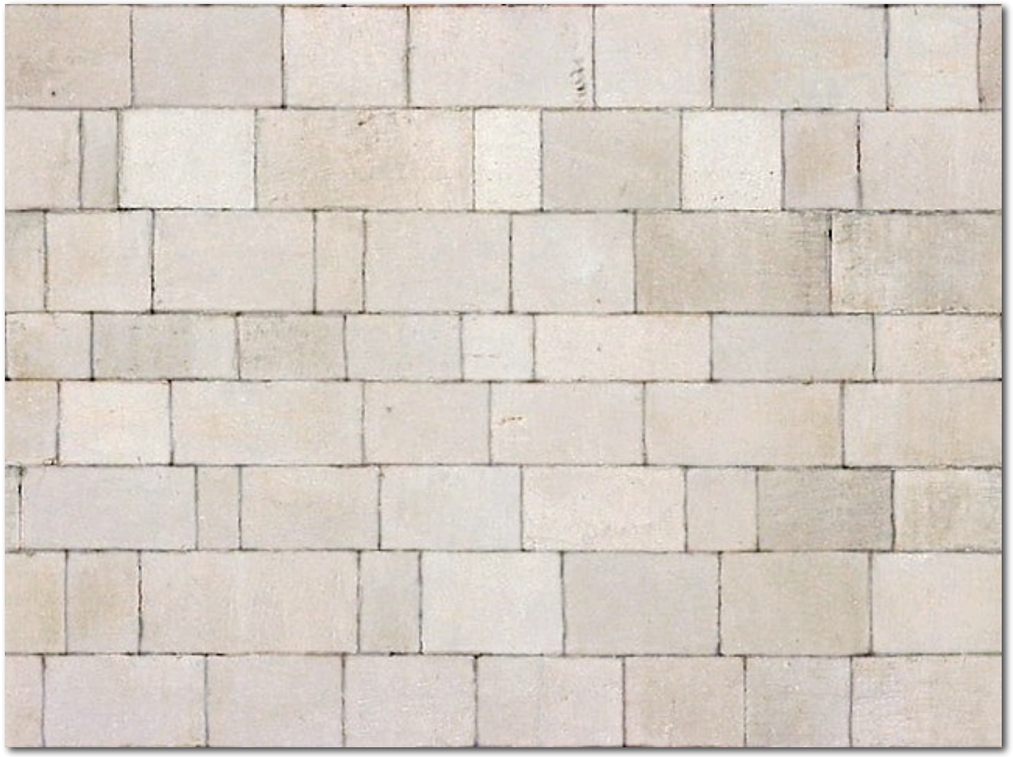


Red channel = u
Green channel = v
So uv=(0,0) is black, uv=(1,1) is yellow

Textured Sponza

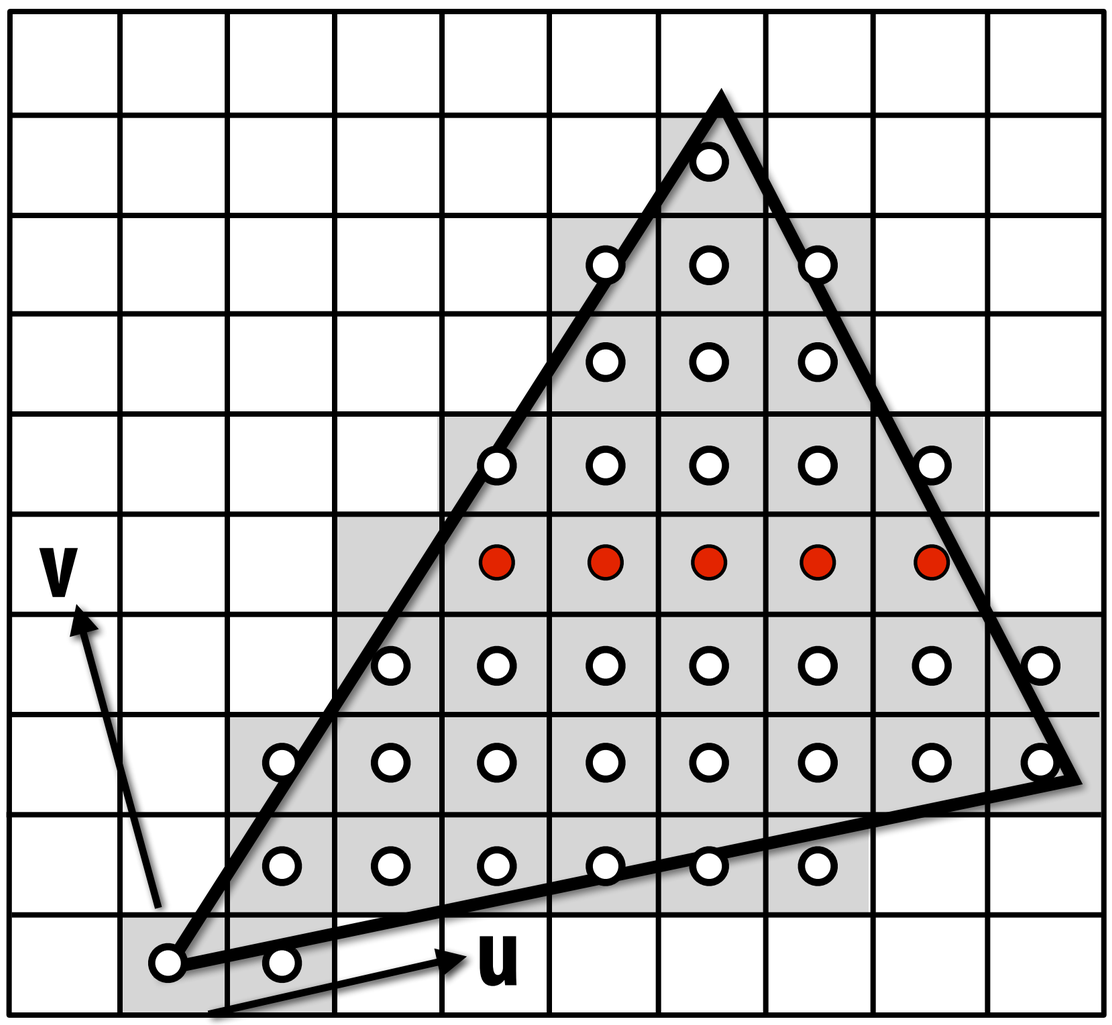


Example textures used in Sponza



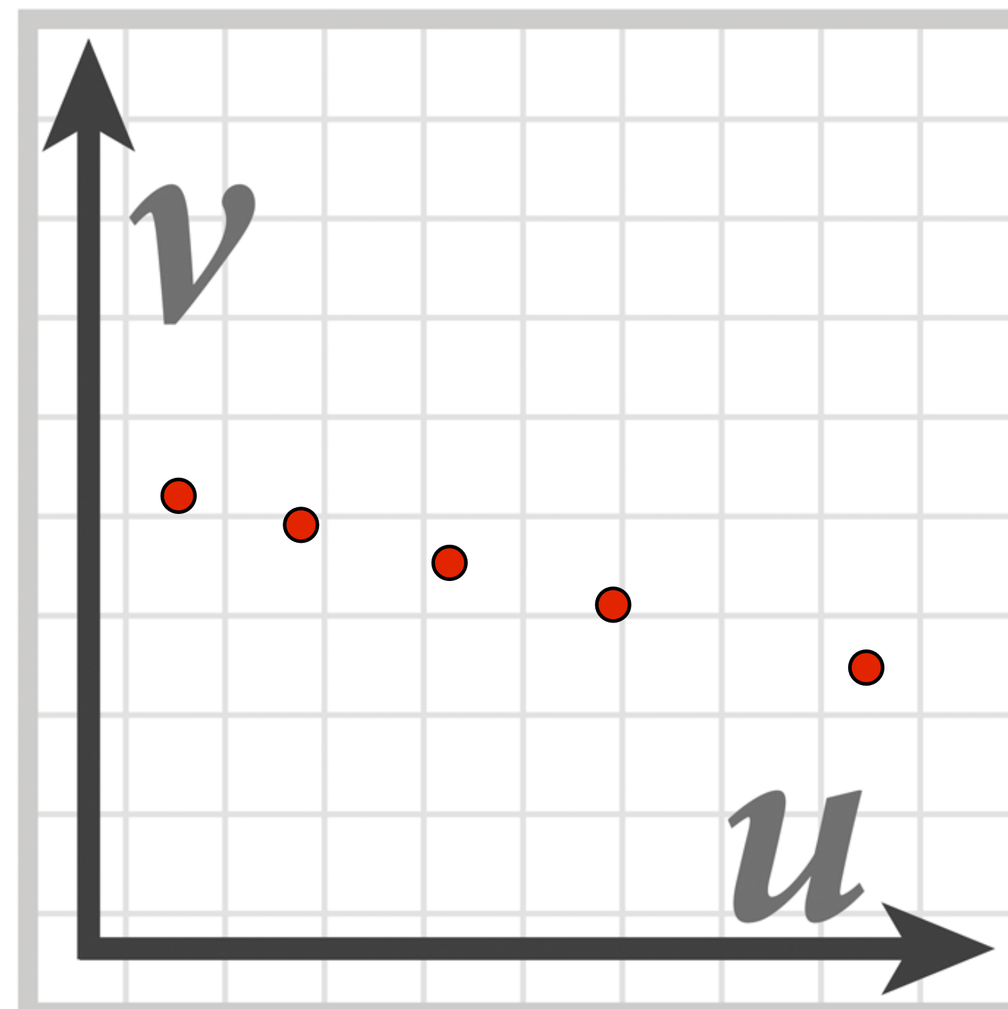
Texture space

Sample positions in XY screen space



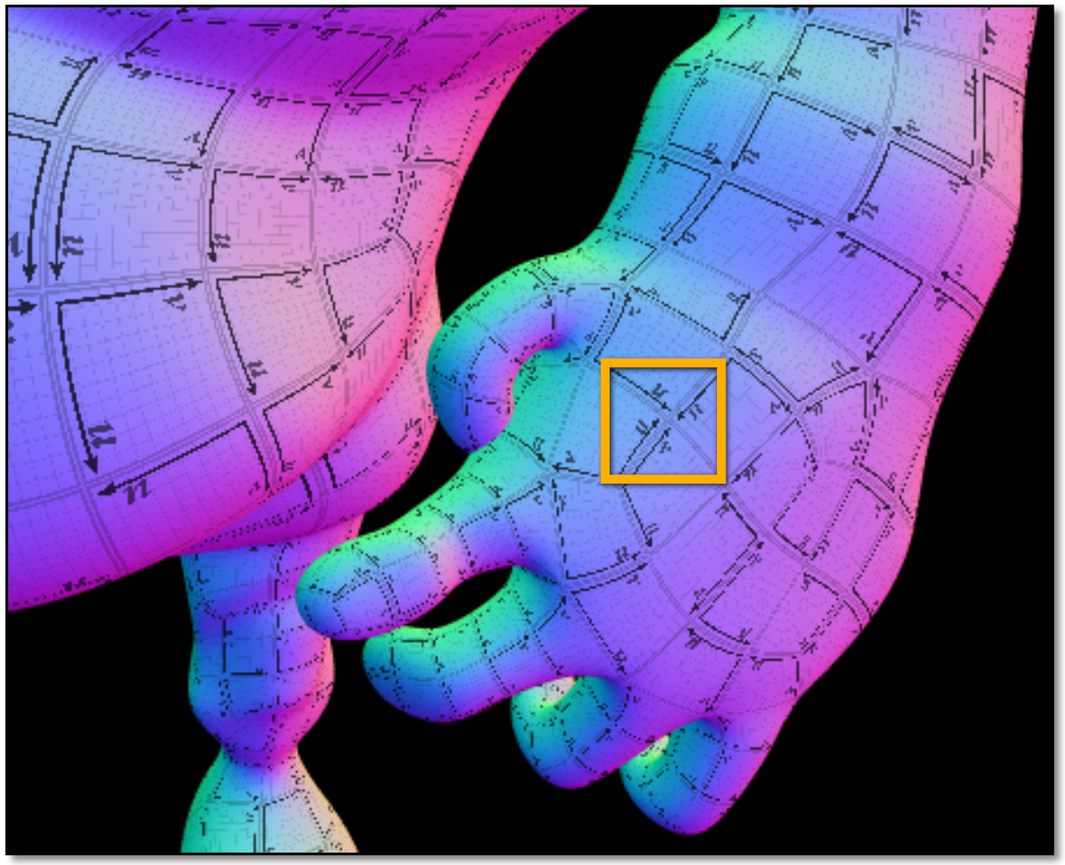
Shading sample positions are uniform in screen space (graphics pipeline samples triangle's appearance — a.k.a. evaluates the fragment shader — at these locations)

Sample positions in UV texture space

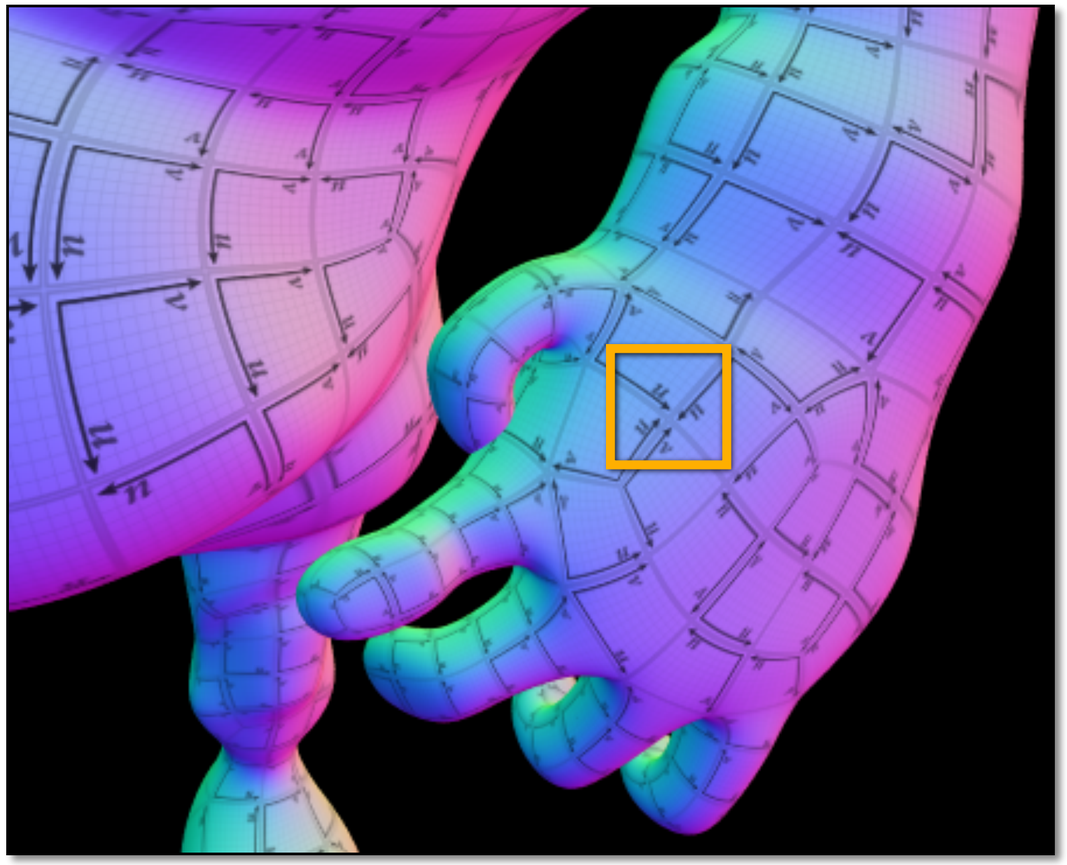


Texture sample positions in texture space (texture is sampled at these locations during shading)

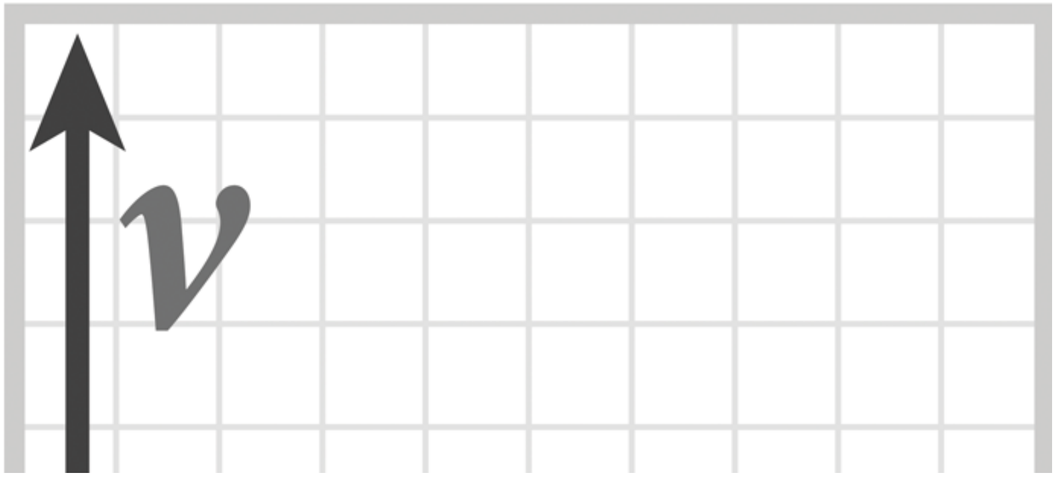
Aliasing due to undersampling texture



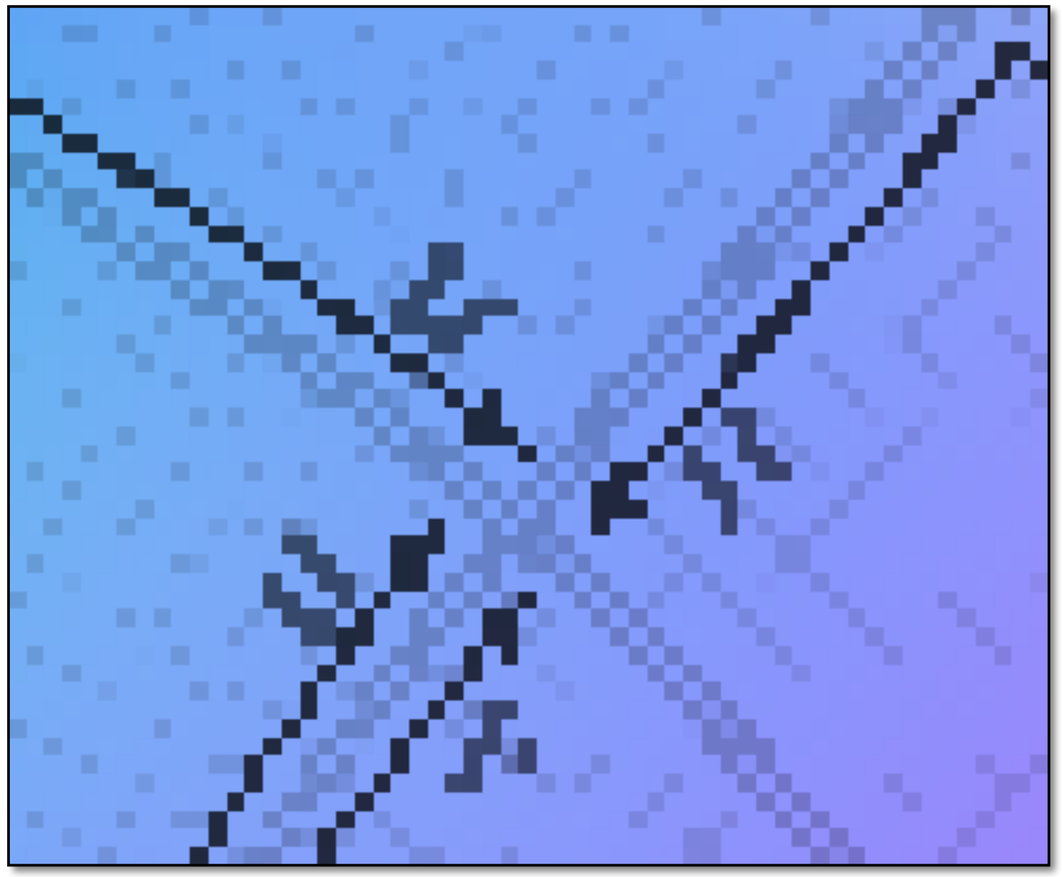
No pre-filtering of texture data
(resulting image exhibits aliasing)



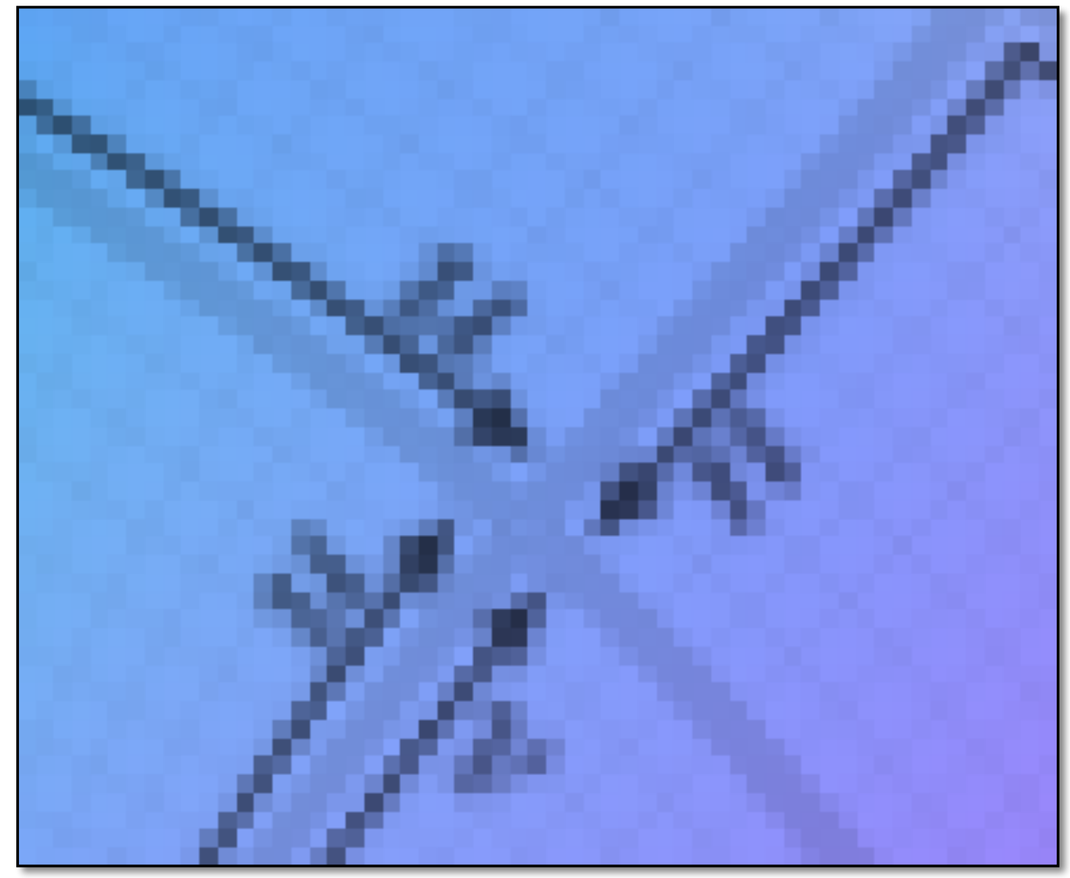
Rendering using pre-filtered texture data



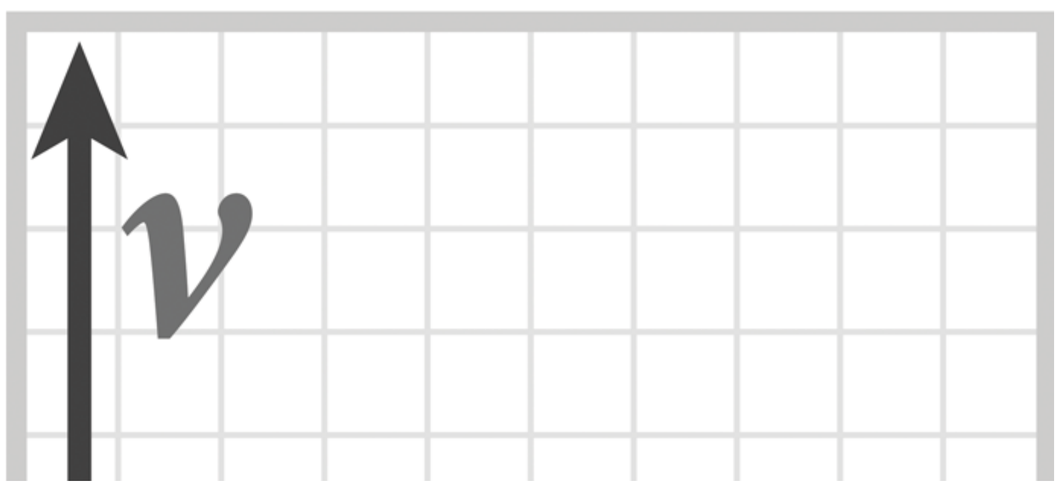
Aliasing due to undersampling



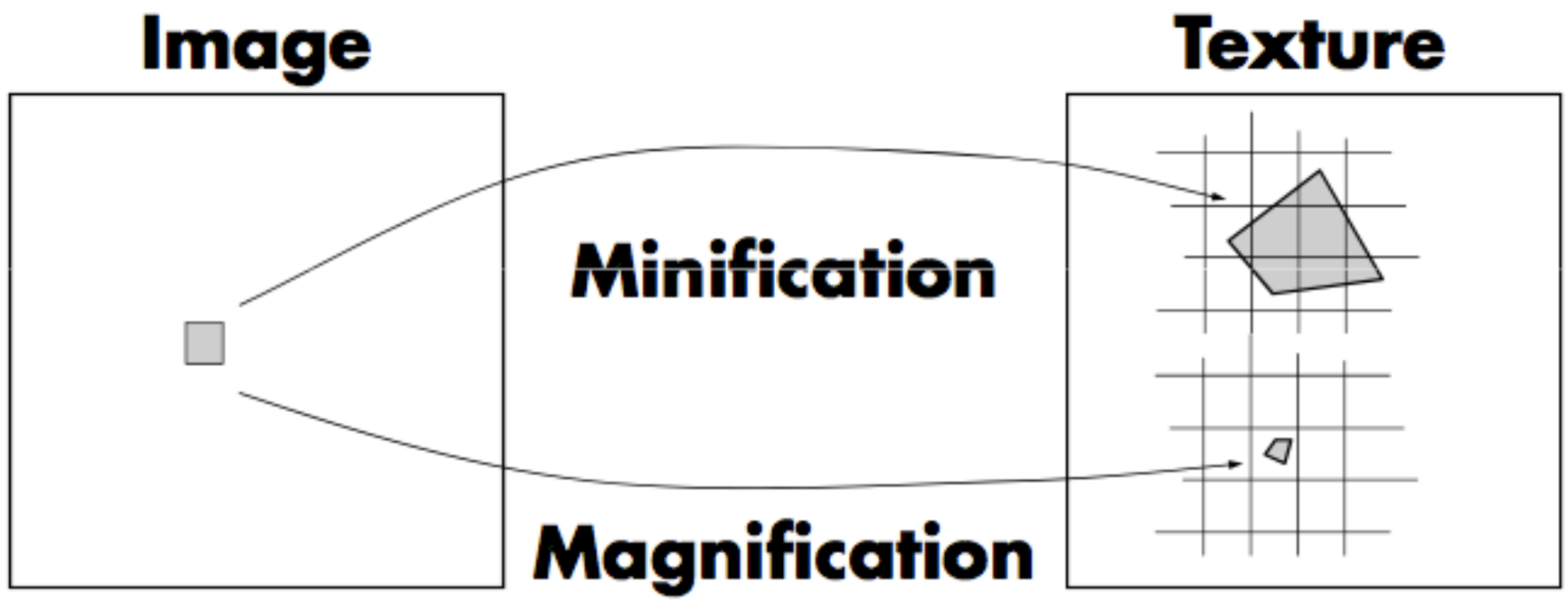
No pre-filtering of texture data
(resulting image exhibits aliasing)



Rendering using pre-filtered texture data



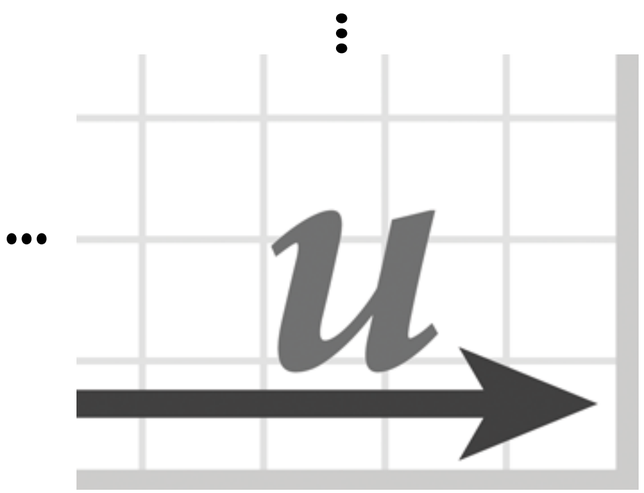
Filtering textures



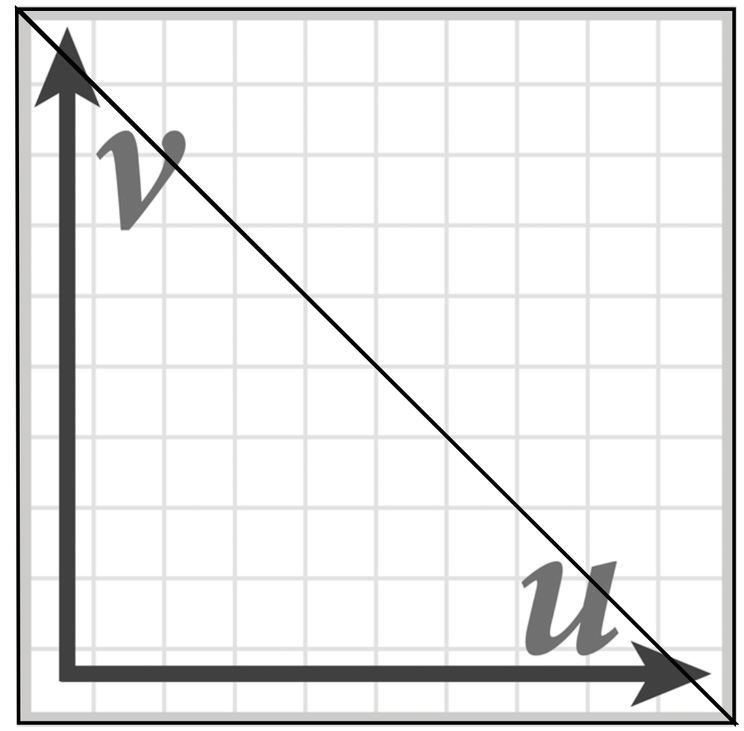
- **Minification:**
  - Area of screen pixel maps to large region of texture (filtering required -- averaging)
  - One texel corresponds to far less than a pixel on screen
  - Example: when scene object is very far away
- **Magnification:**
  - Area of screen pixel maps to tiny region of texture (interpolation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

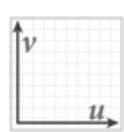
Filtering textures



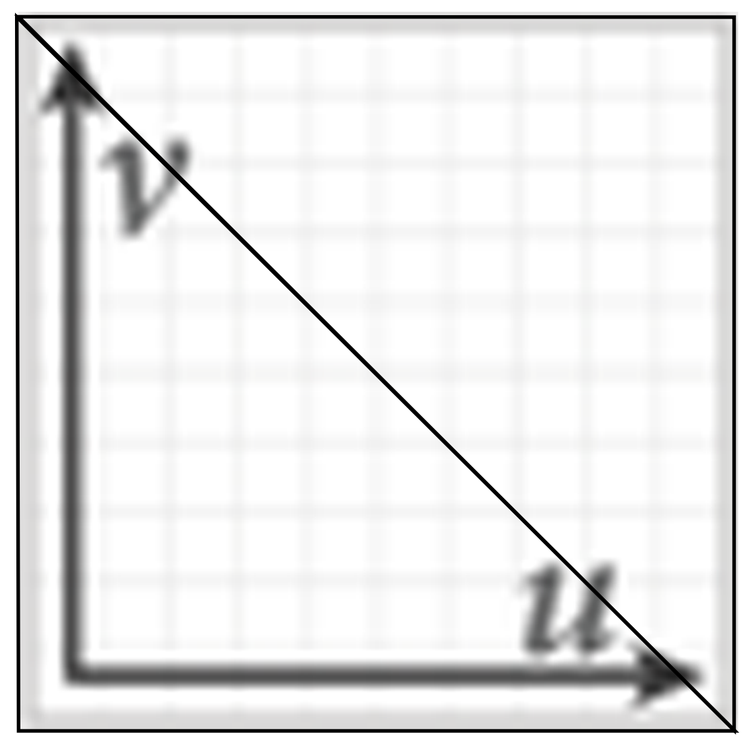
Actual texture: 700x700
(only a crop is shown)



Texture minification

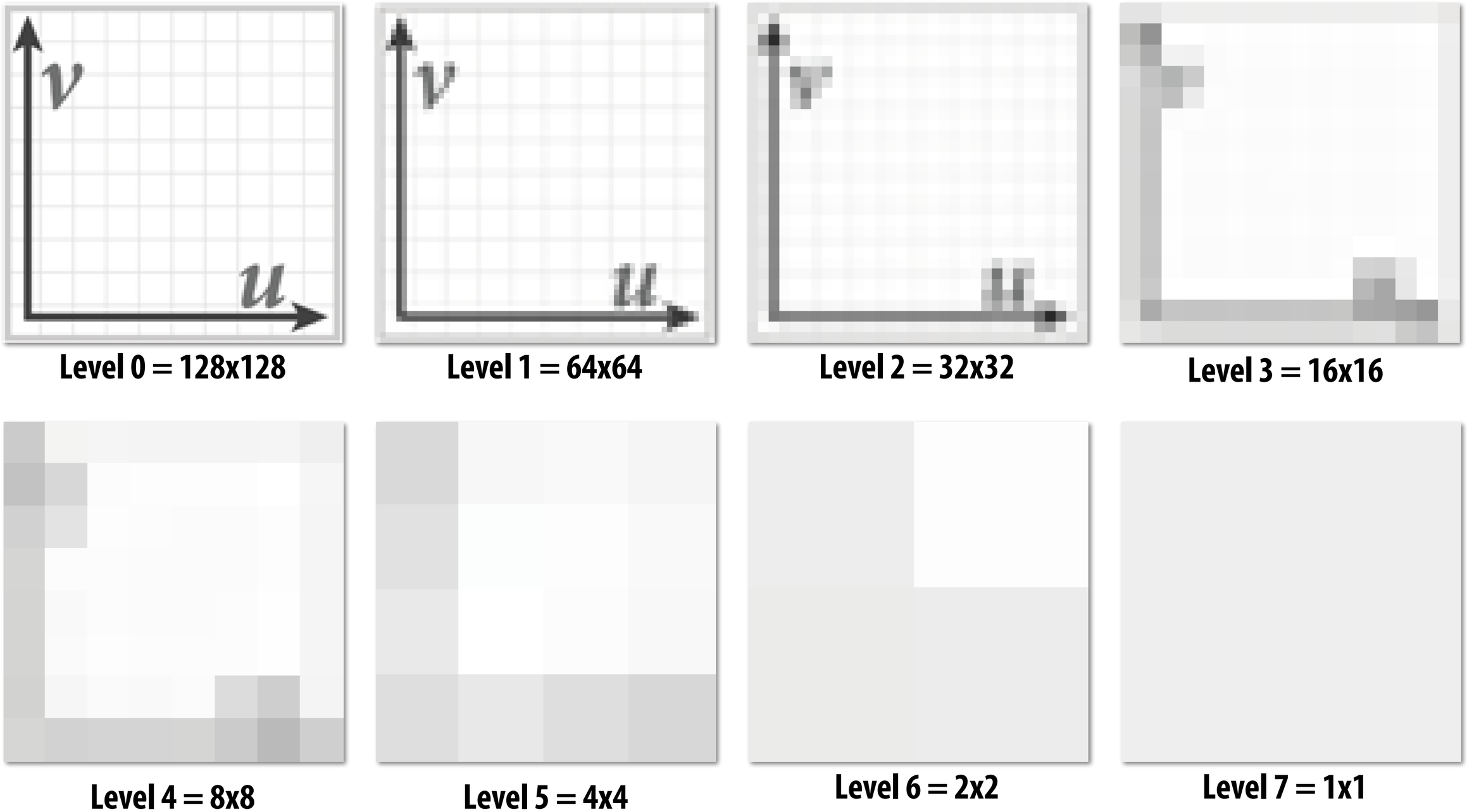


Actual texture: 64x64



Texture magnification

Mipmap (L. Williams 83)

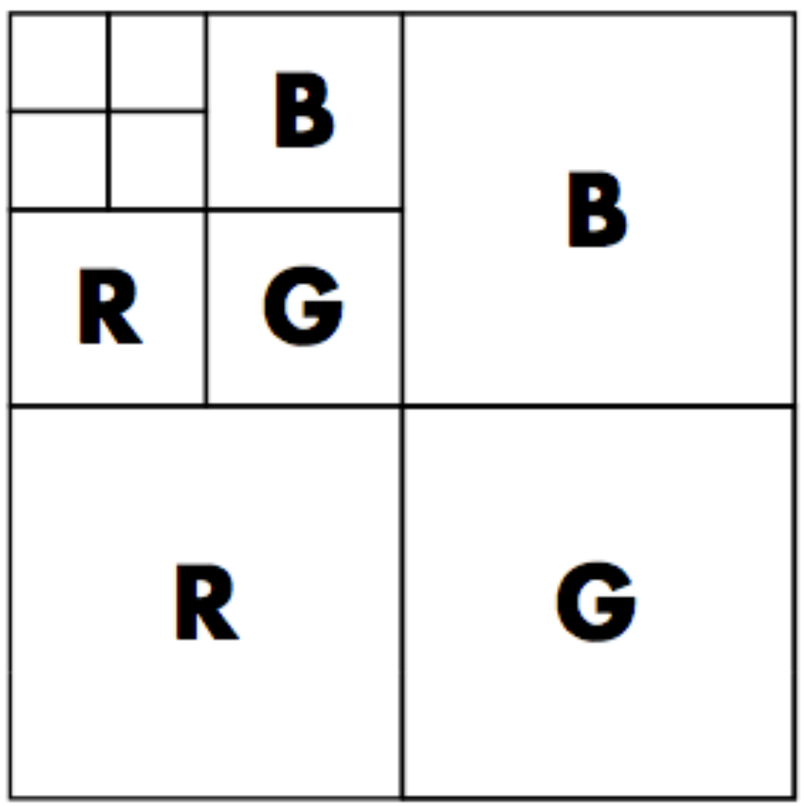


Idea: prefilter texture data to remove high frequencies

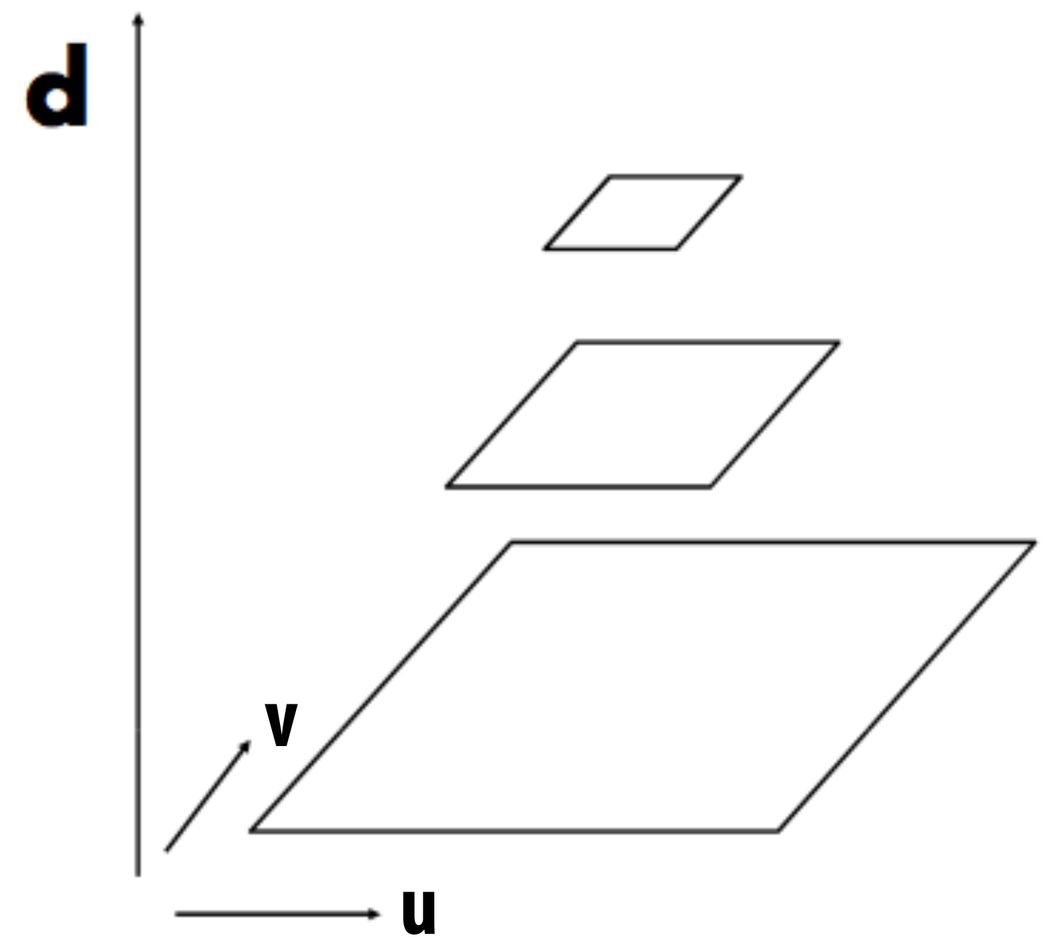
Texels at higher levels store integral of the texture function over a region of texture space (downsampled images)

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

Mipmap (L. Williams 83)

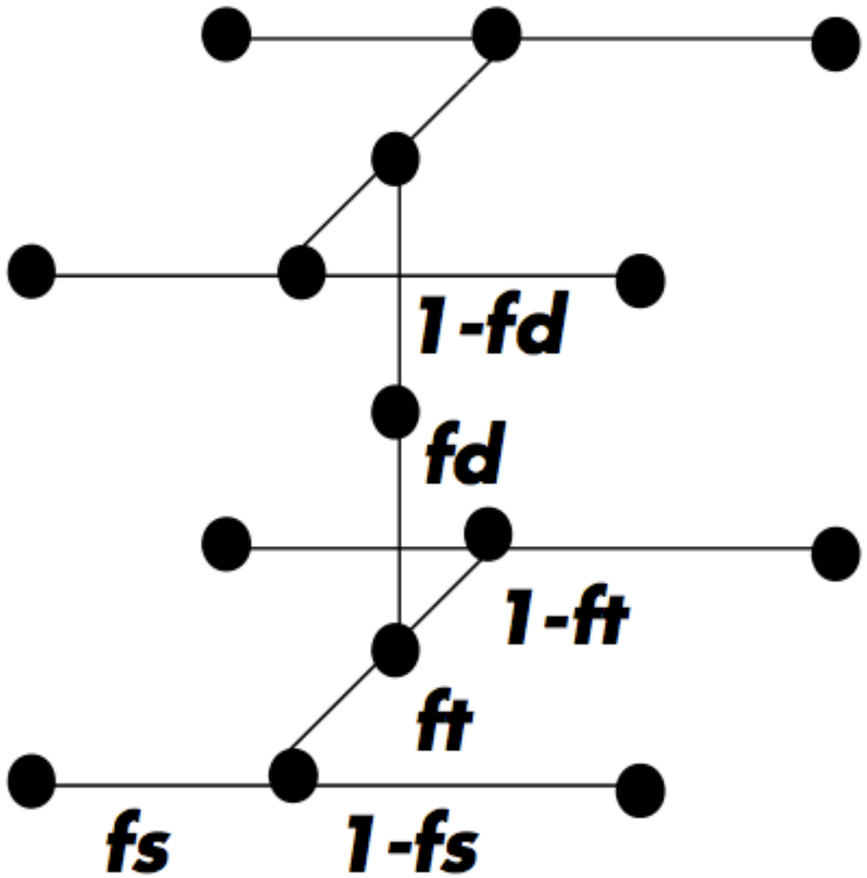


Williams' original proposed mip-map layout



"Mip hierarchy"
level = d

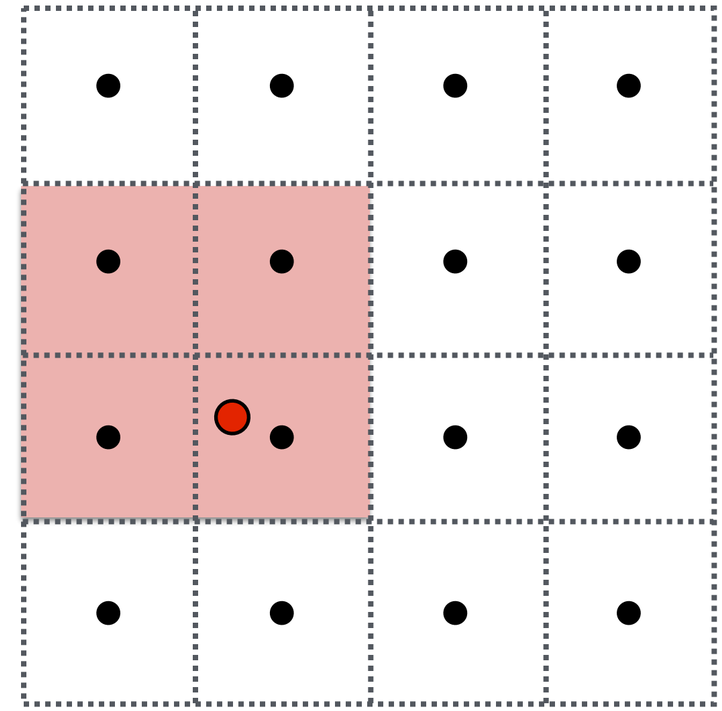
Constant-time filtering



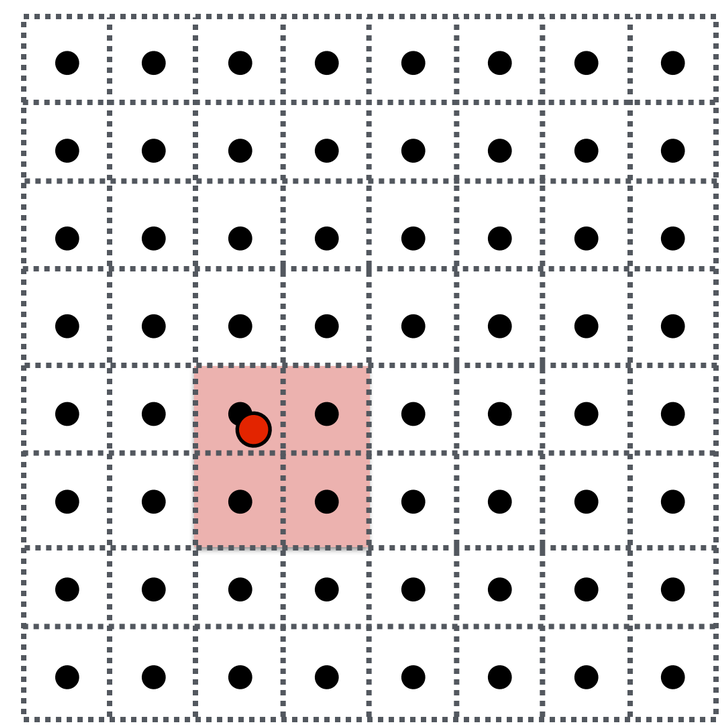
$$lerp(t, v_1, v_2) = v_1 + t(v_2 - v_1)$$

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

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



mip-map texels: level d+1

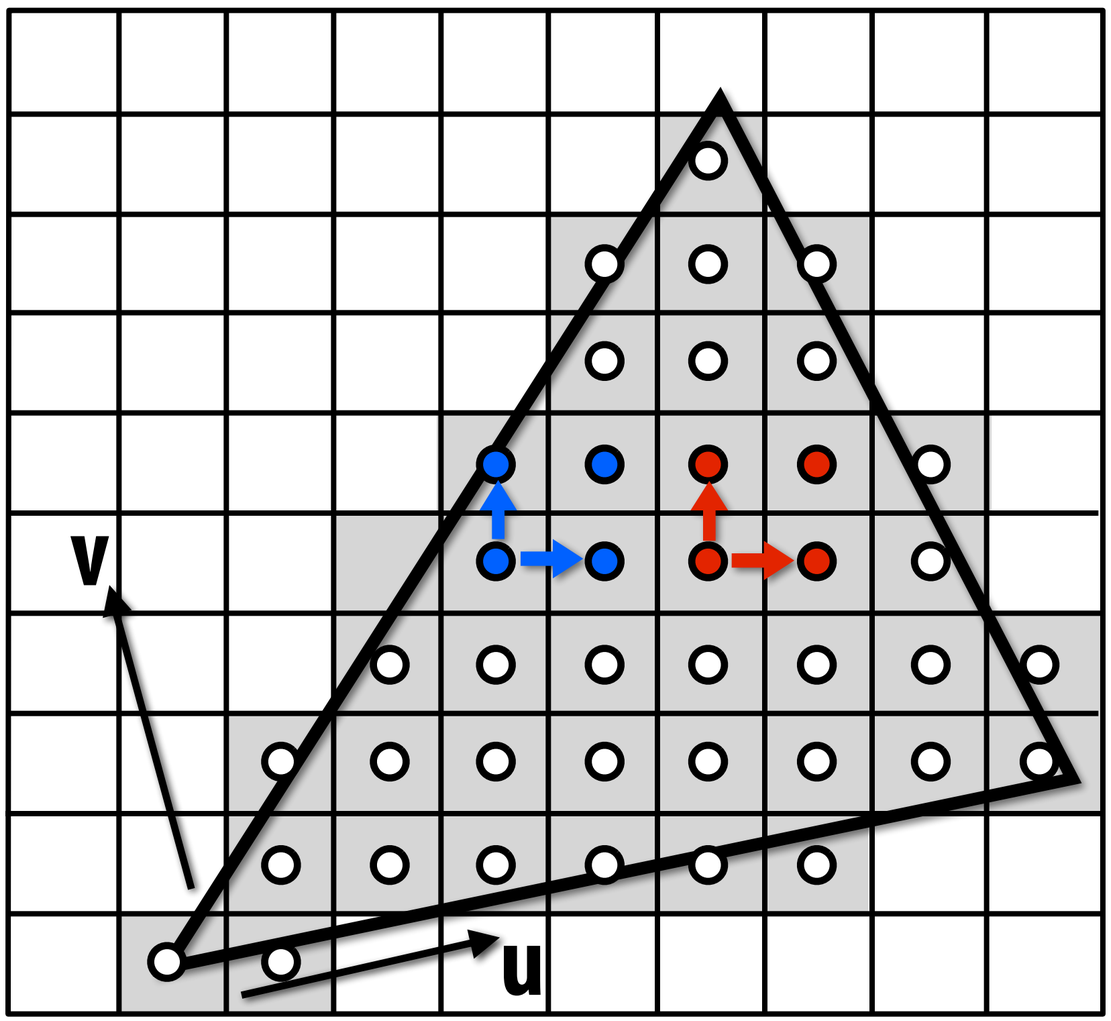


mip-map texels: level d

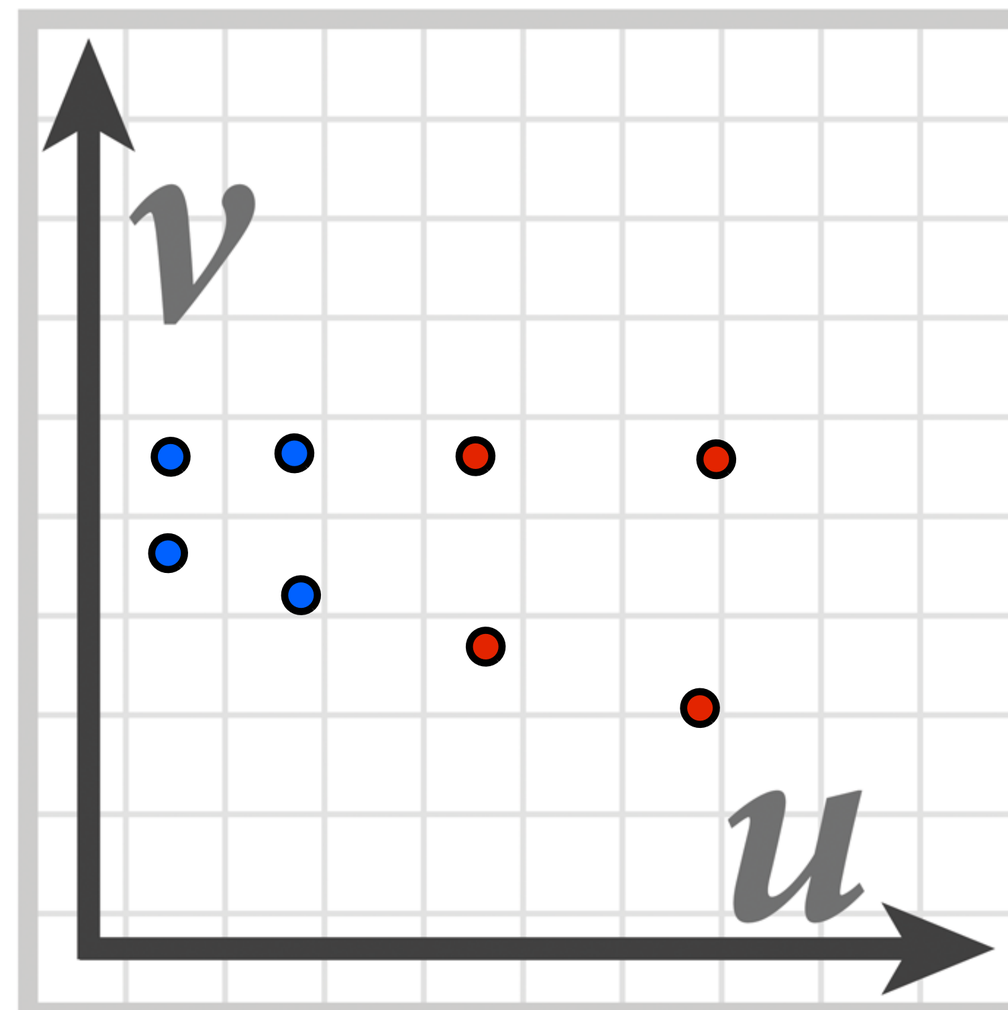
Figure credit: Akeley and Hanrahan

Computing *d*

Take differences between texture coordinate values of neighboring fragments



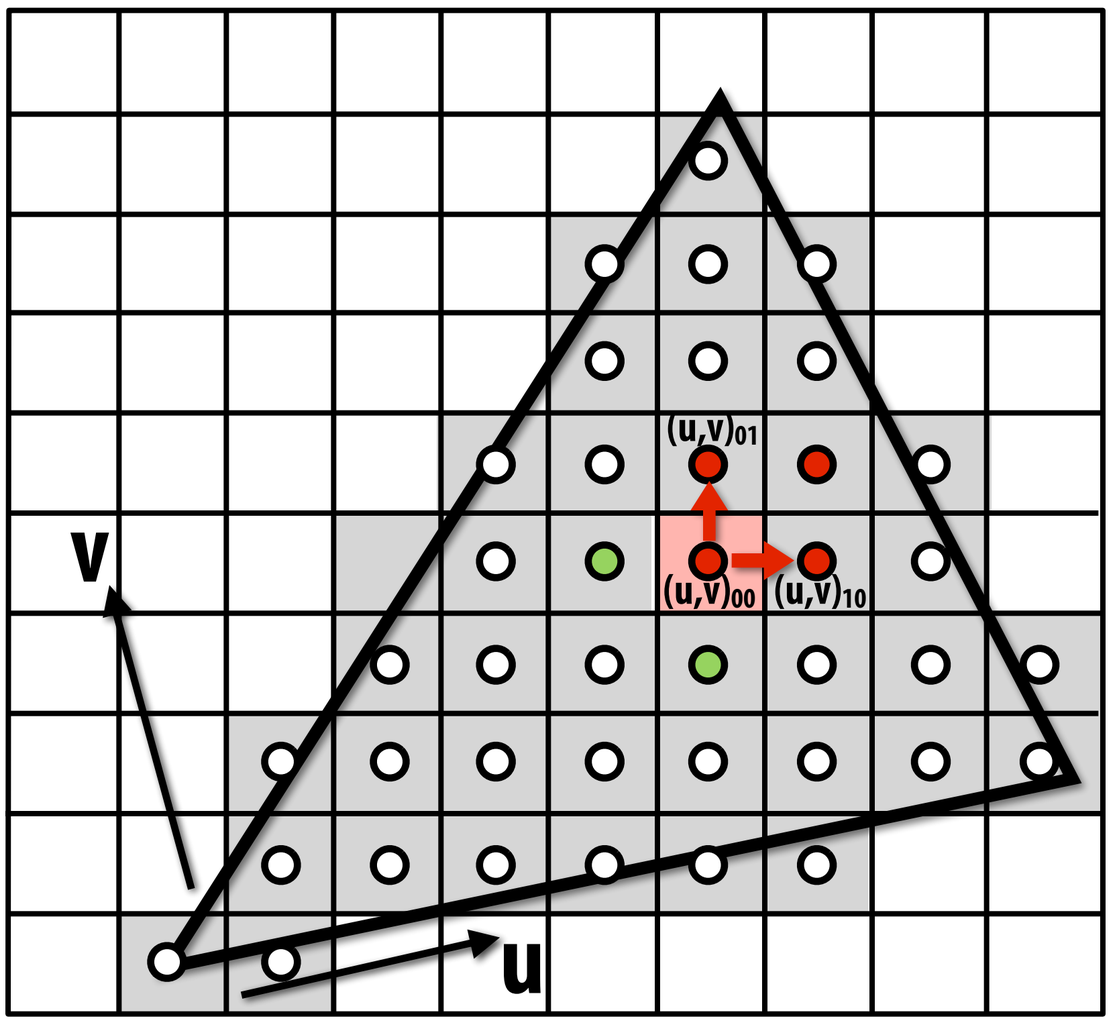
Screen space

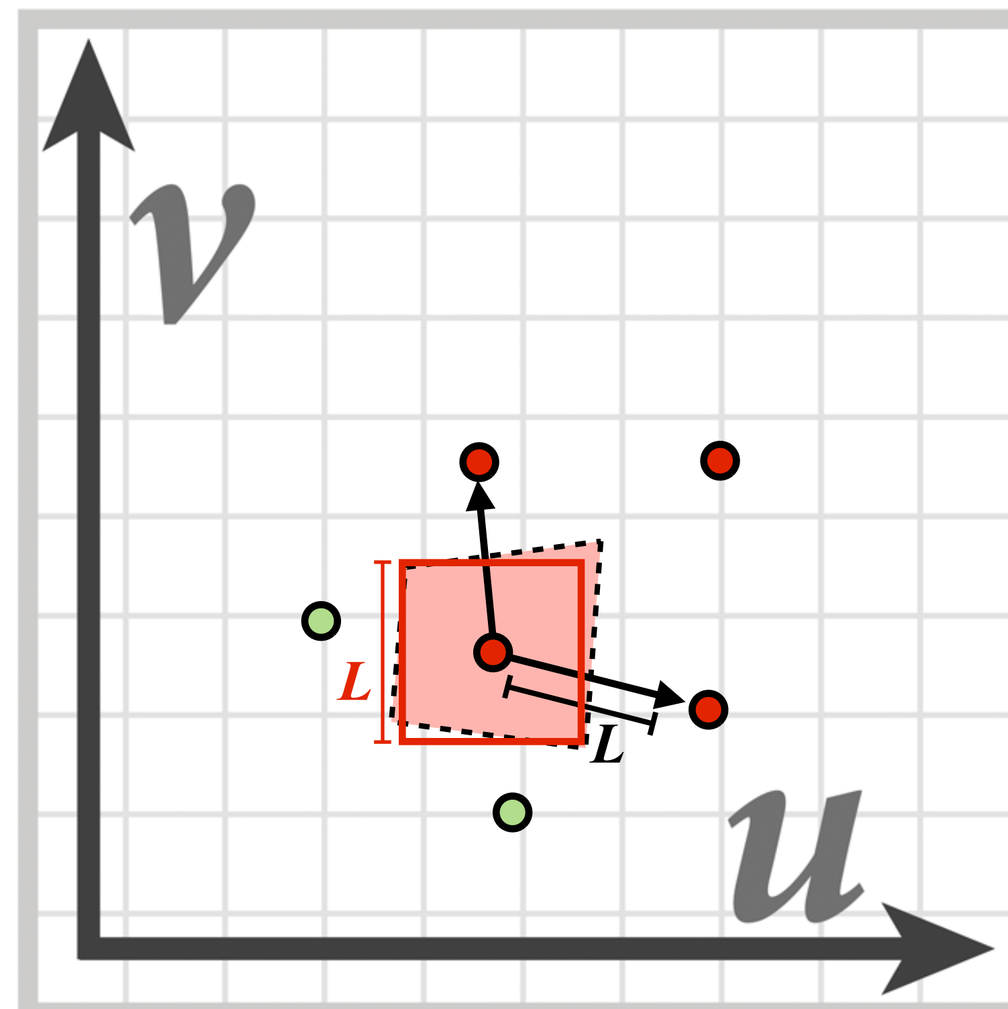


Texture space

Computing *d*

Take differences between texture coordinate values of neighboring fragments





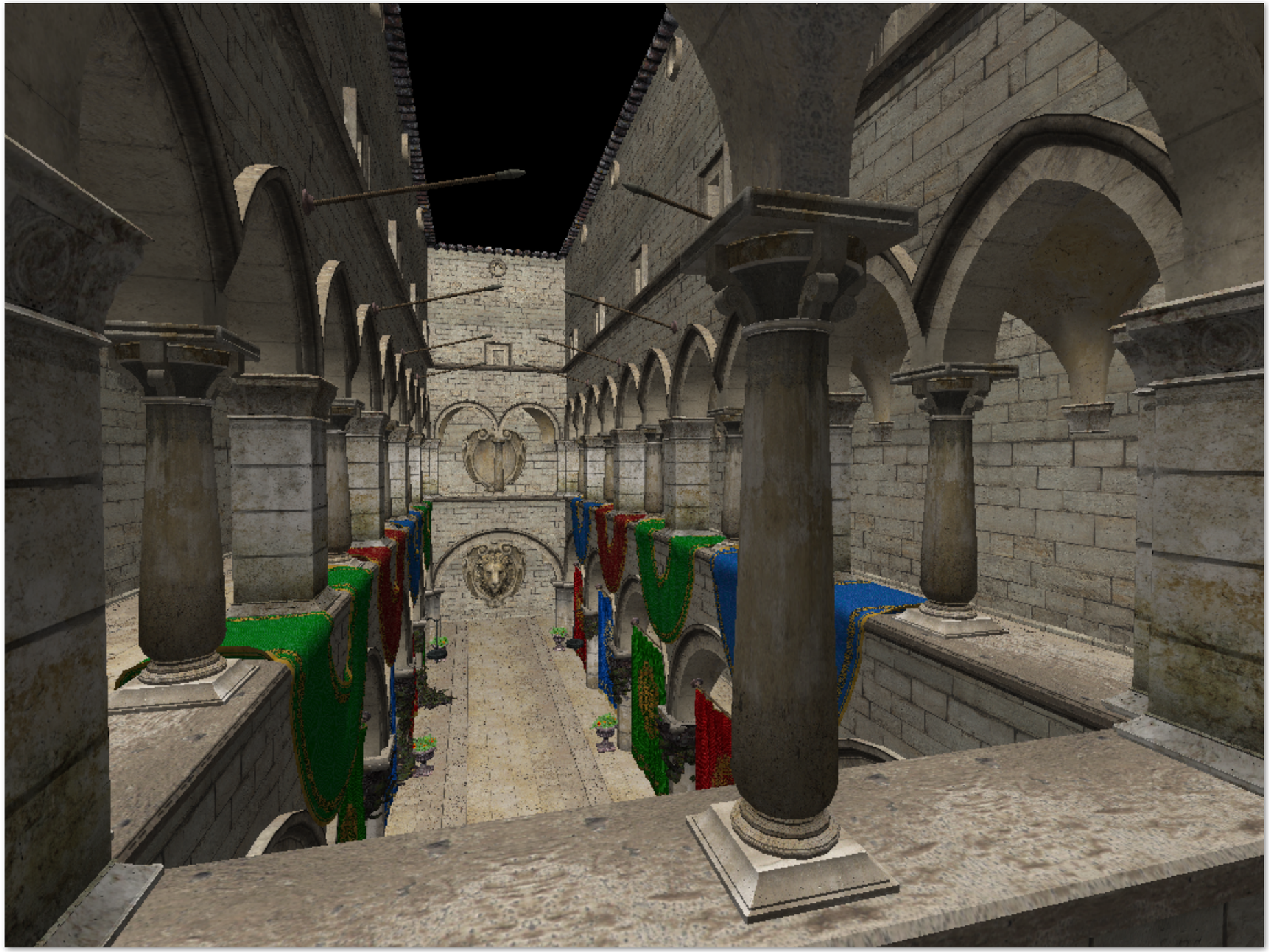
$du/dx = u_{10} - u_{00}$ $\quad$ $dv/dx = v_{10} - v_{00}$

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

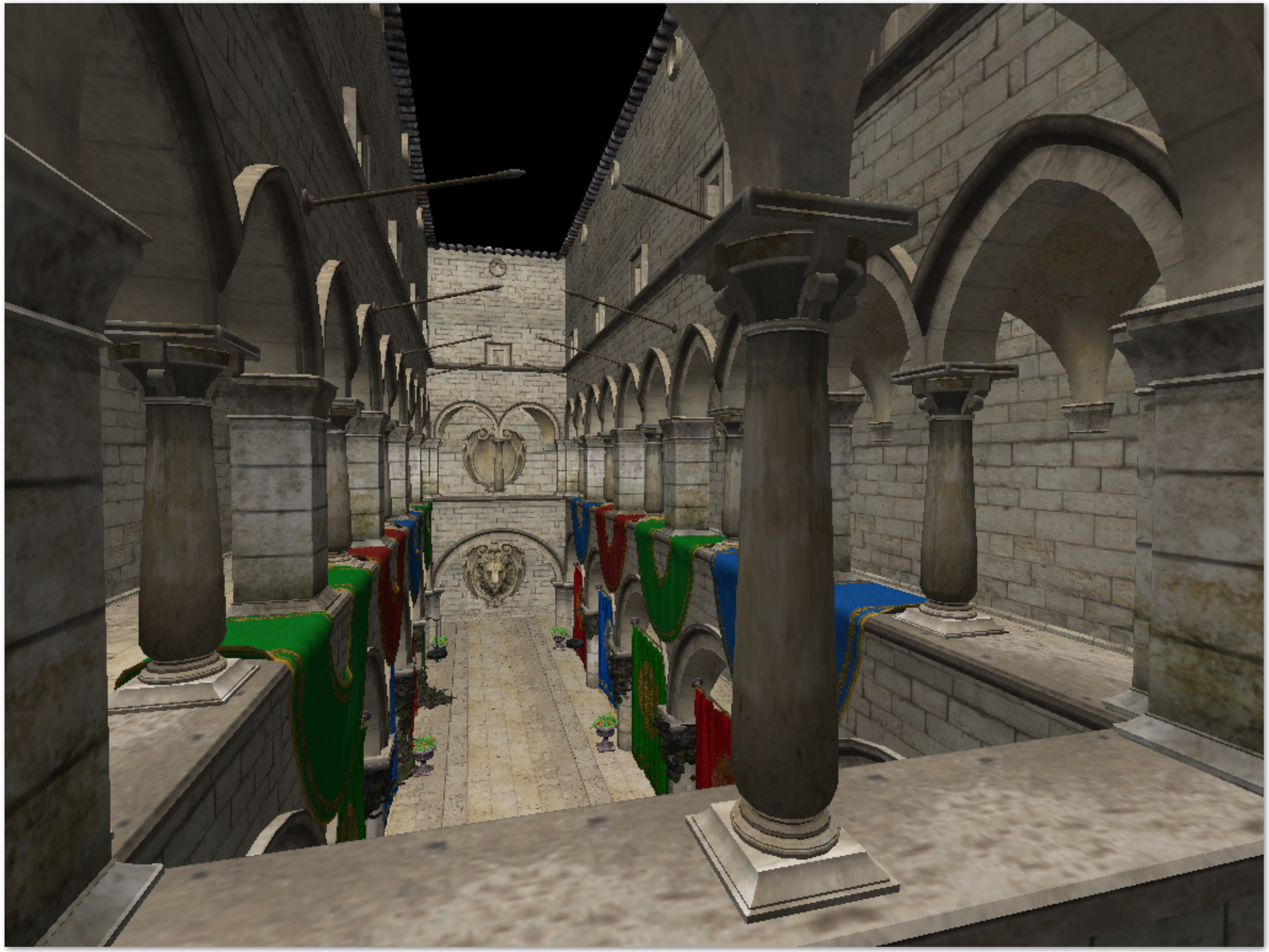
$$L = \max\left(\sqrt{\left(\frac{du}{dx}\right)^2 + \left(\frac{dv}{dx}\right)^2}, \sqrt{\left(\frac{du}{dy}\right)^2 + \left(\frac{dv}{dy}\right)^2}\right)$$

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

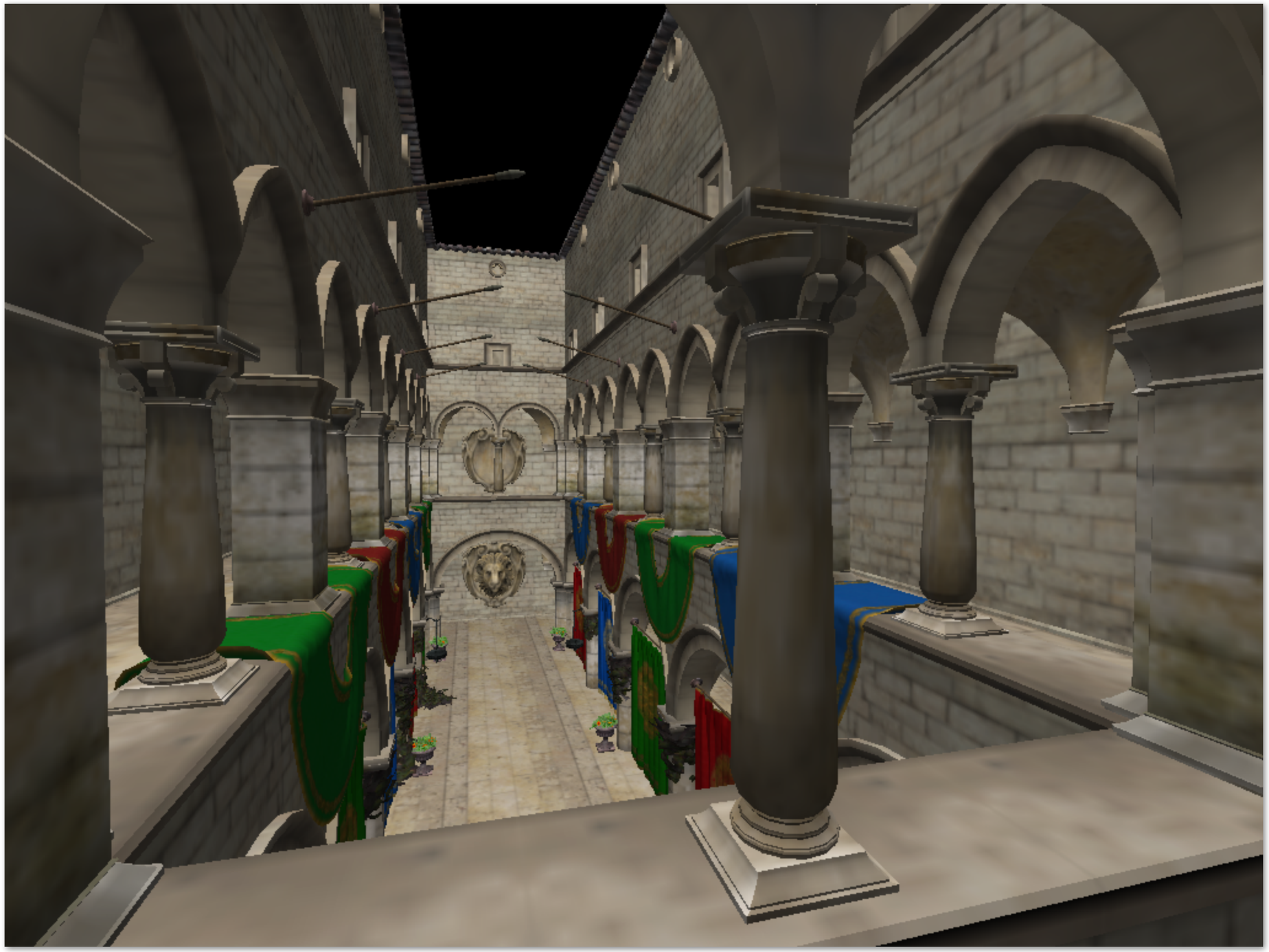
Sponza (bilinear resampling at level 0)



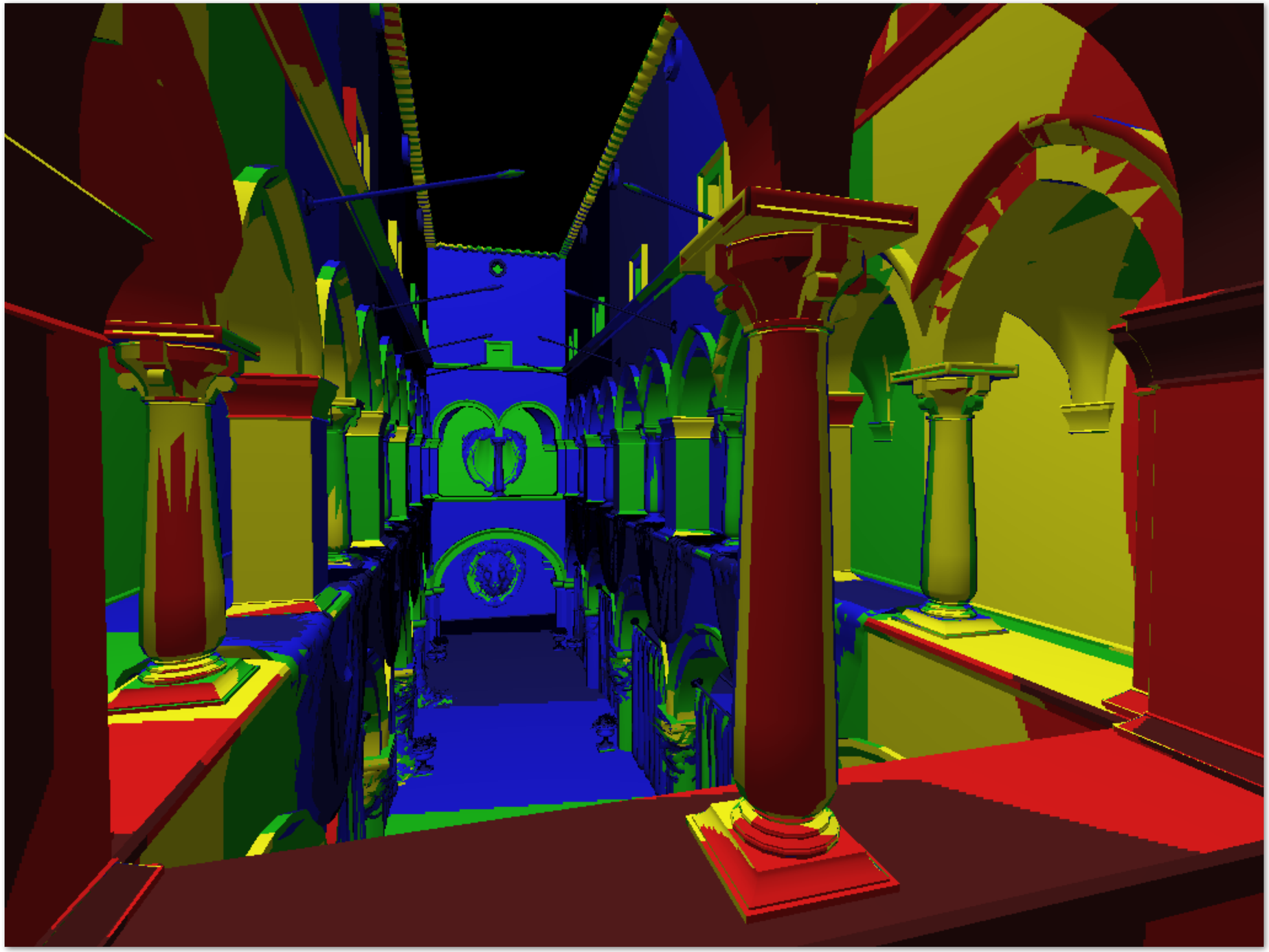
Sponza (bilinear resampling at level 2)



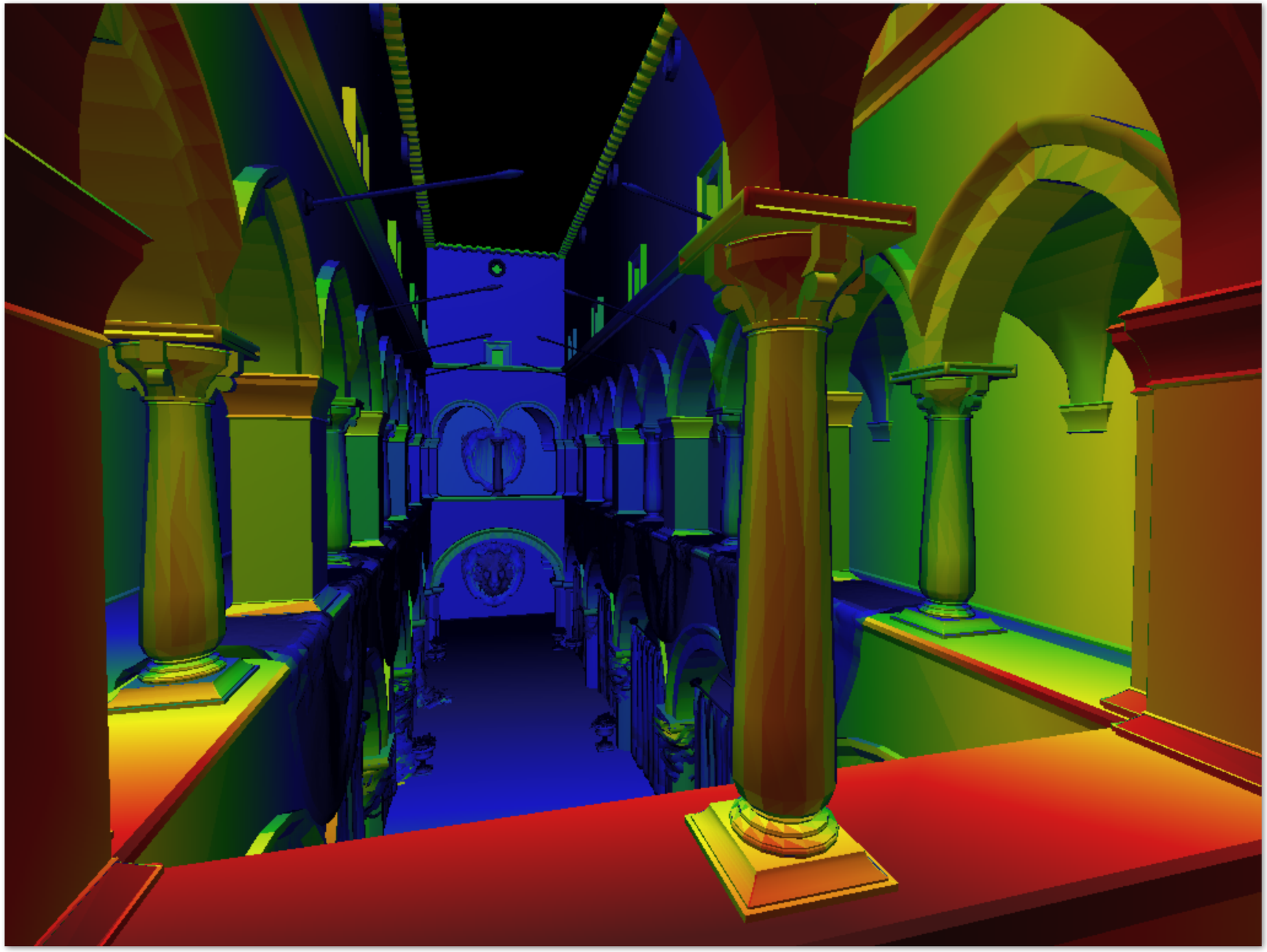
Sponza (bilinear resampling at level 4)



Mip-map level visualization
(bilinear filtering only: *d* clamped to nearest level)

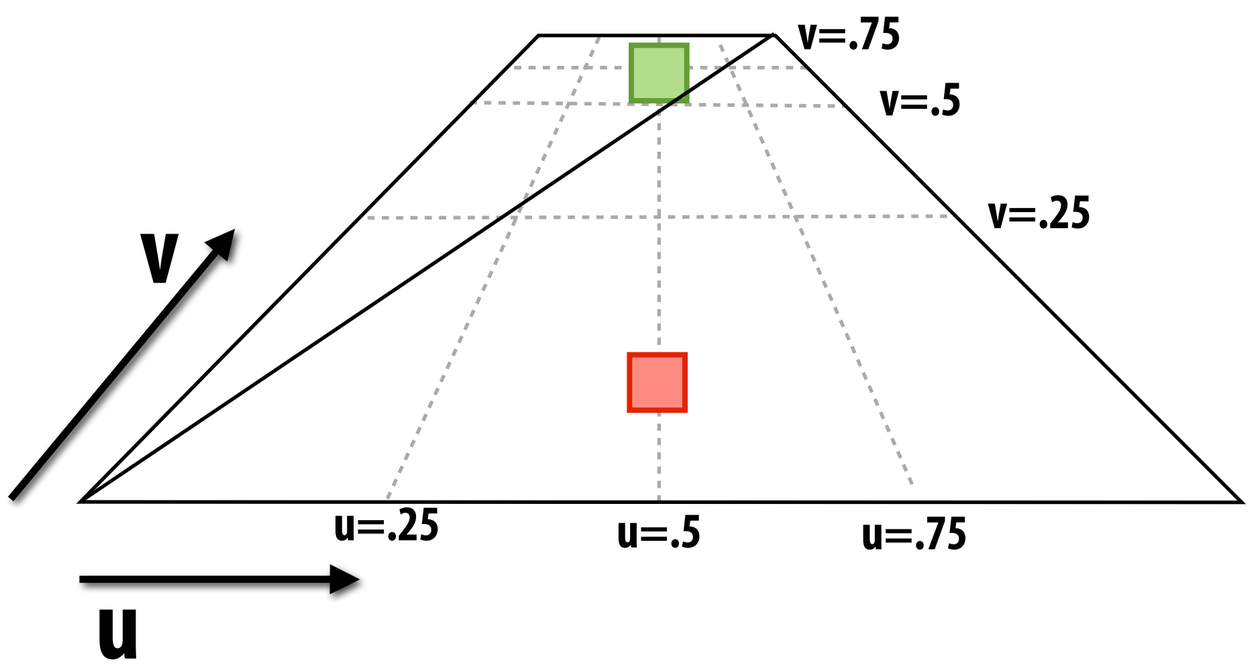


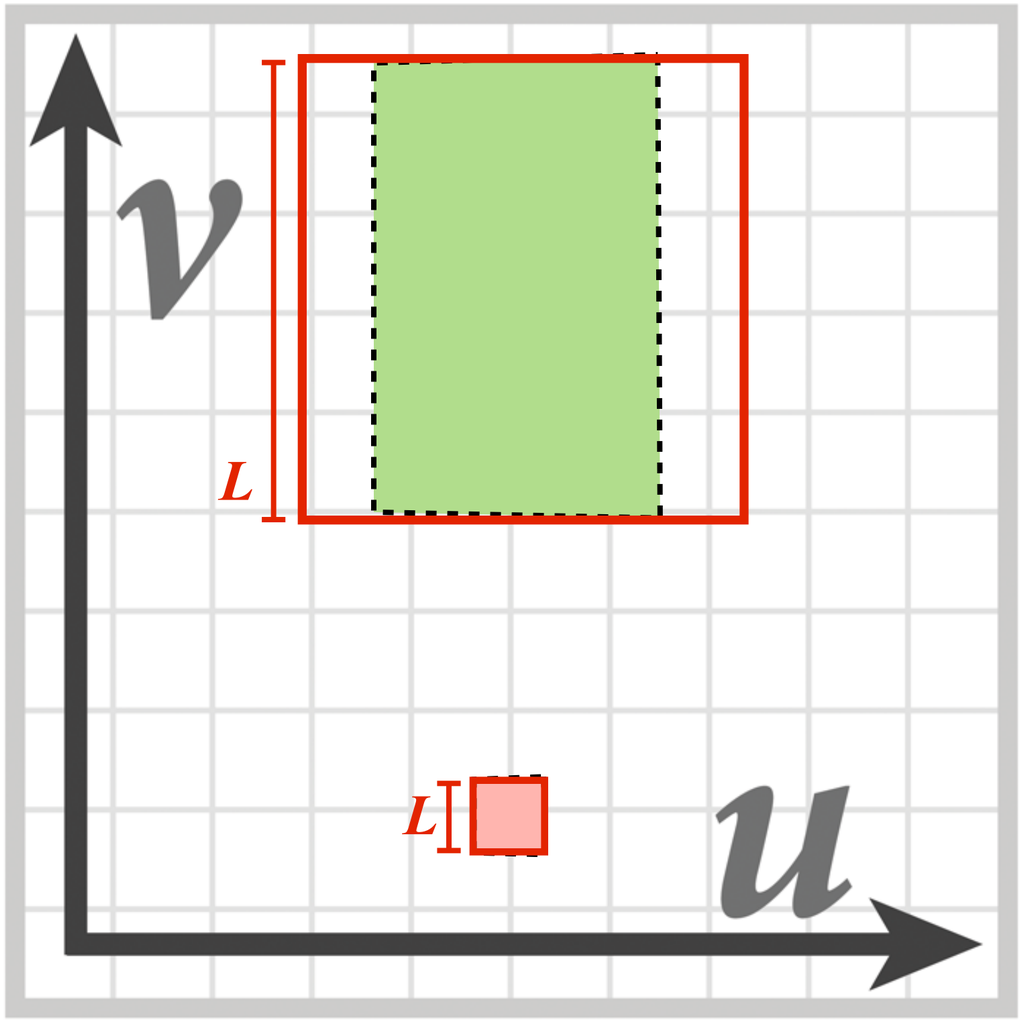
Mip-map level visualization
(trilinear filtering: visualization of continuous *d*)



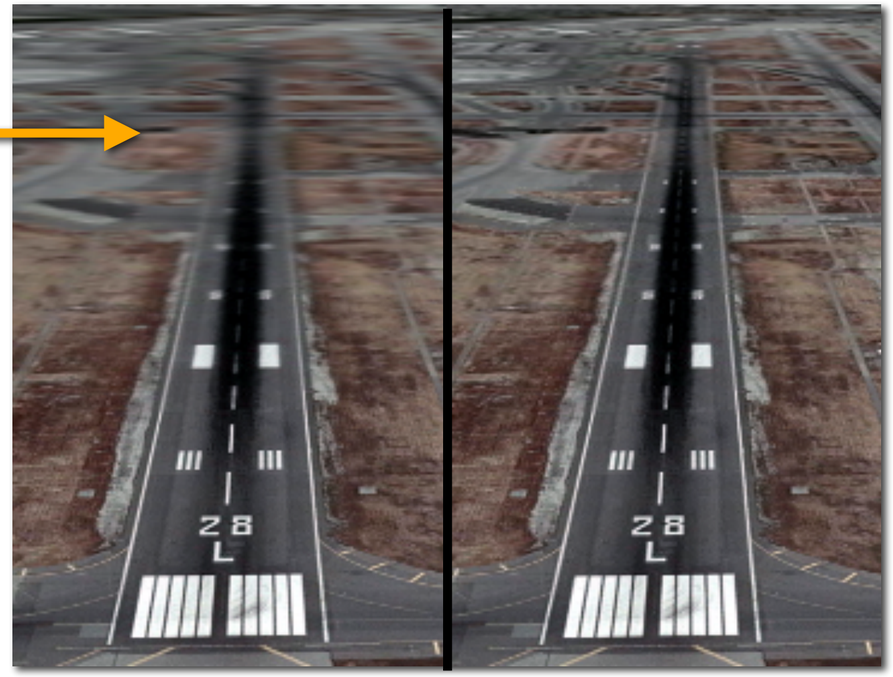
Pixel area may not map to isotropic region in texture space

Proper filtering requires anisotropic filter footprint





Overblurring in *u* direction



Trilinear (Isotropic) Filtering

Anisotropic Filtering

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

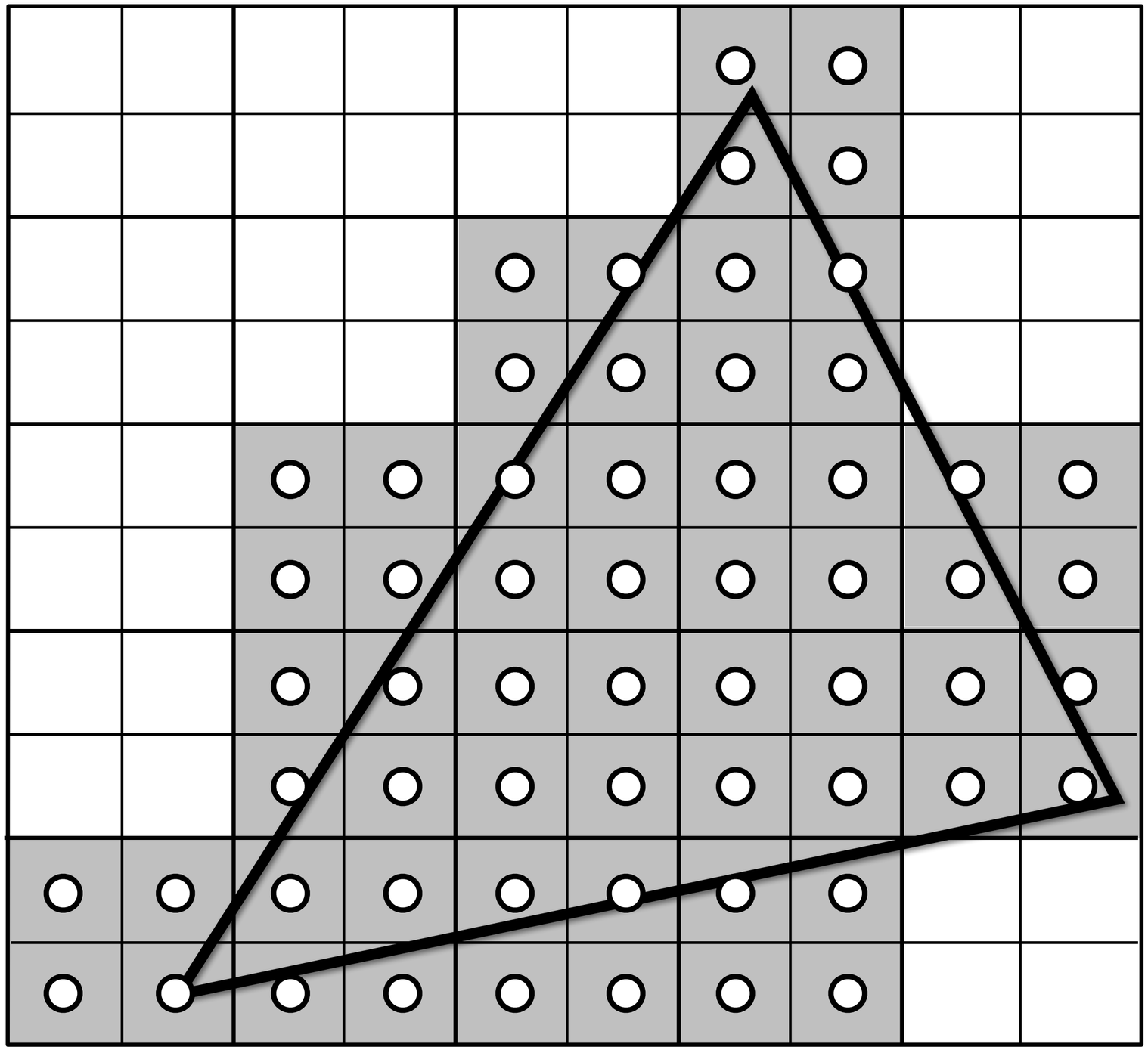
$$mip\text{-}map\ d = \log_2(L)$$

GPUs shade at the granularity of quad fragments

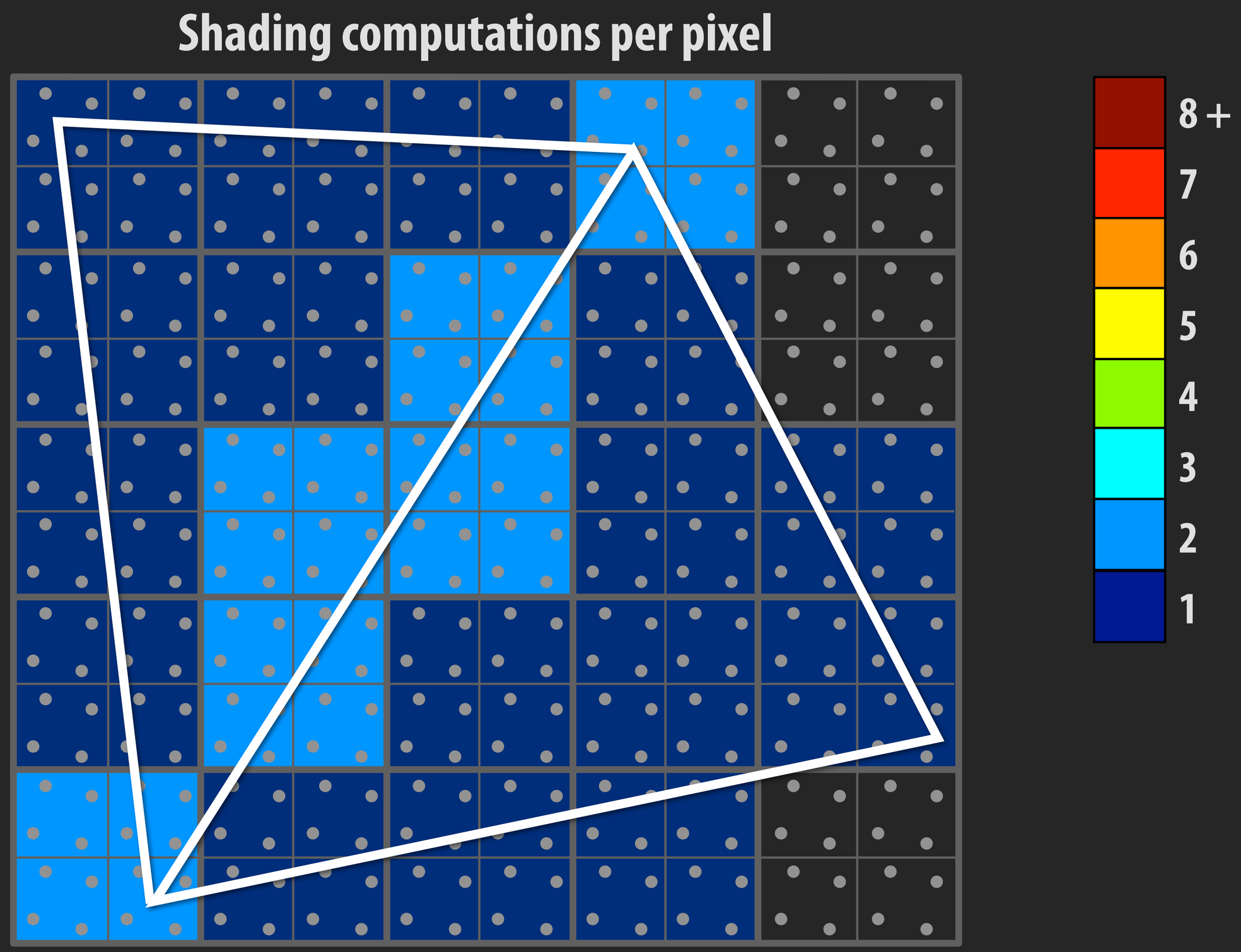
(2x2 fragment block is the minimum granularity of rasterization output and shading)

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

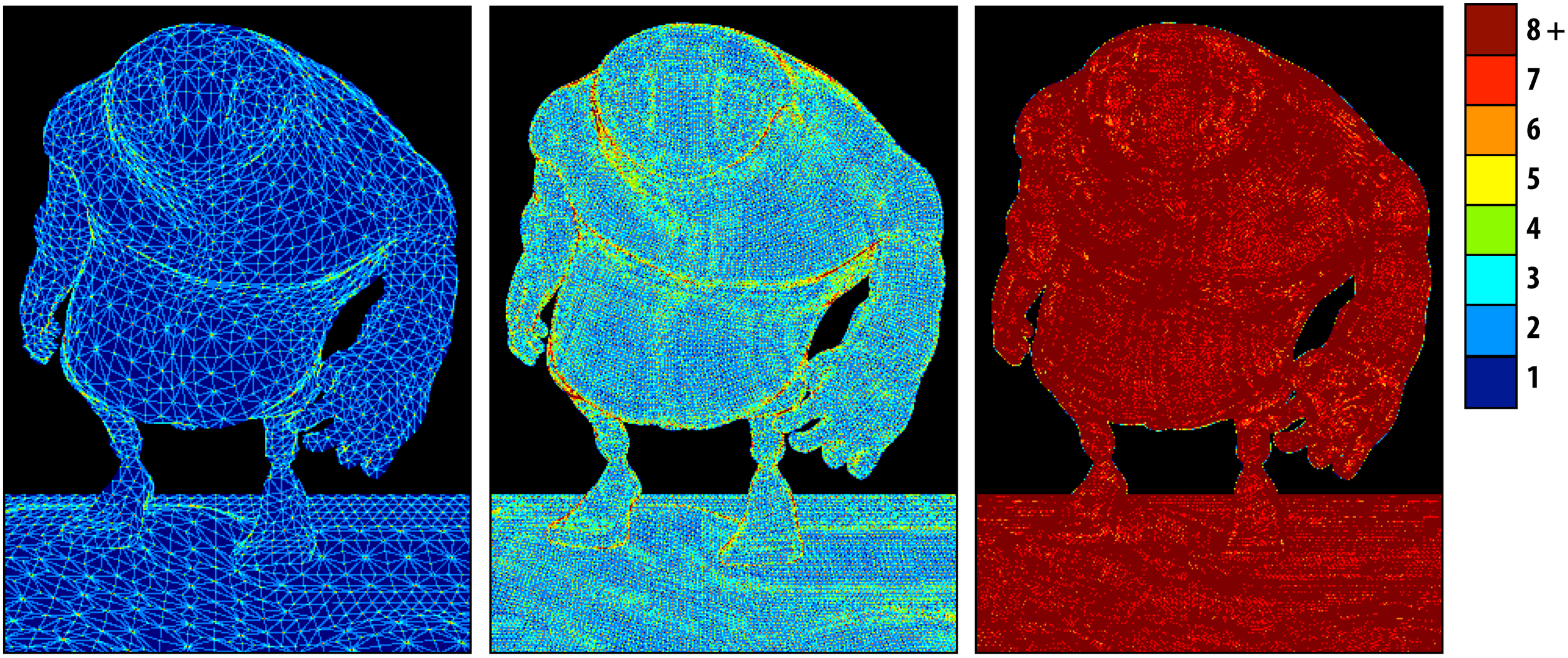


Small triangles result in extra shading

Shaded quad fragments per pixel

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

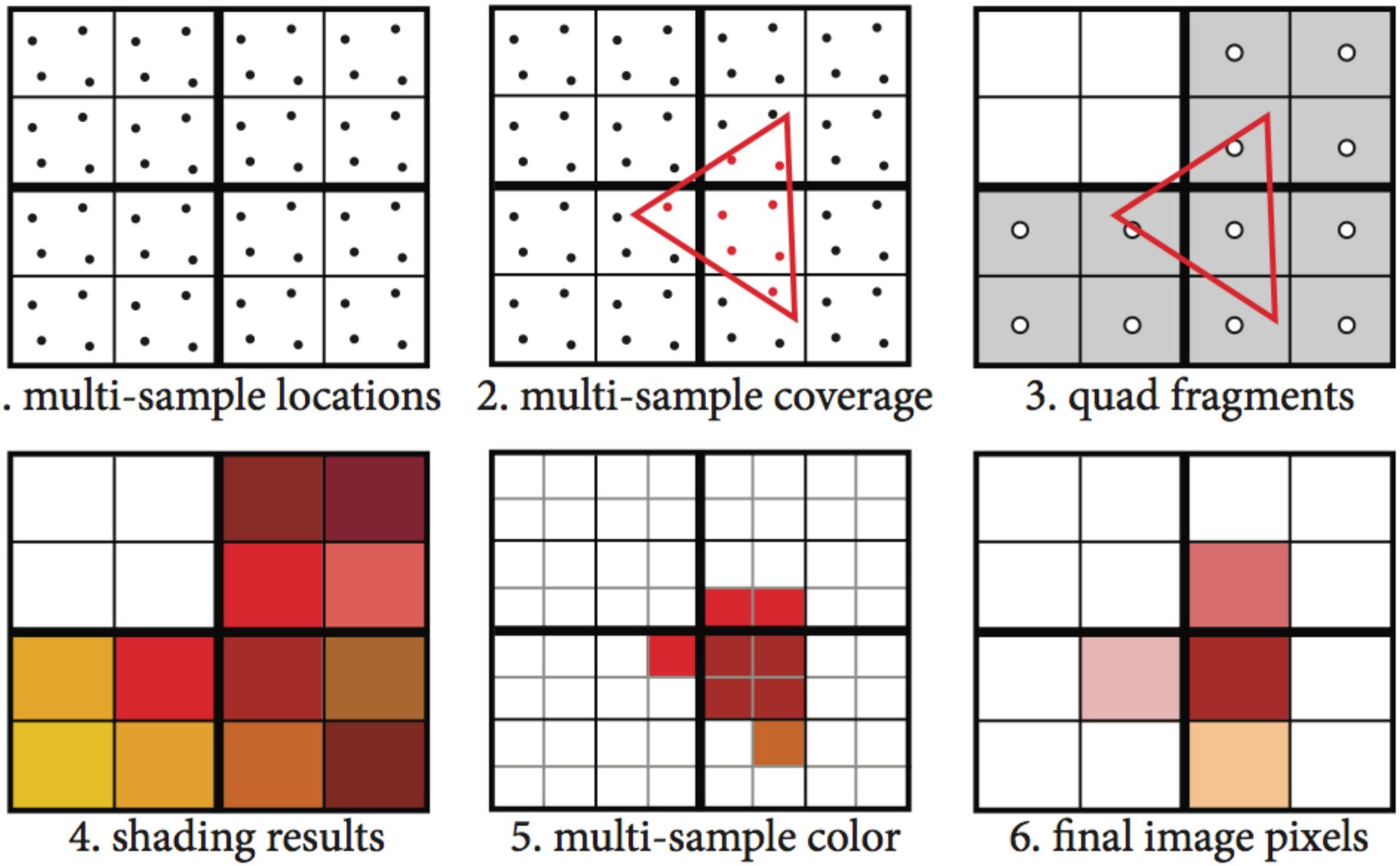
100 pixel-area triangles | **10 pixel-area triangles** | **1 pixel-area triangles**



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

But graphics pipeline generates at least one appearance sample <u>per triangle</u> 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 texture request**
  - Constant filtering cost (independent of *d*)
  - Constant number of texels accessed (independent of *d*)
- **Bilinear/trilinear filtering is isotropic: will "overblur" to avoid aliasing**
  - Anisotropic texture filtering provides higher image quality at higher compute and memory bandwidth cost

Summary: a texture fetch operation

For each texture fetch in a shader program:

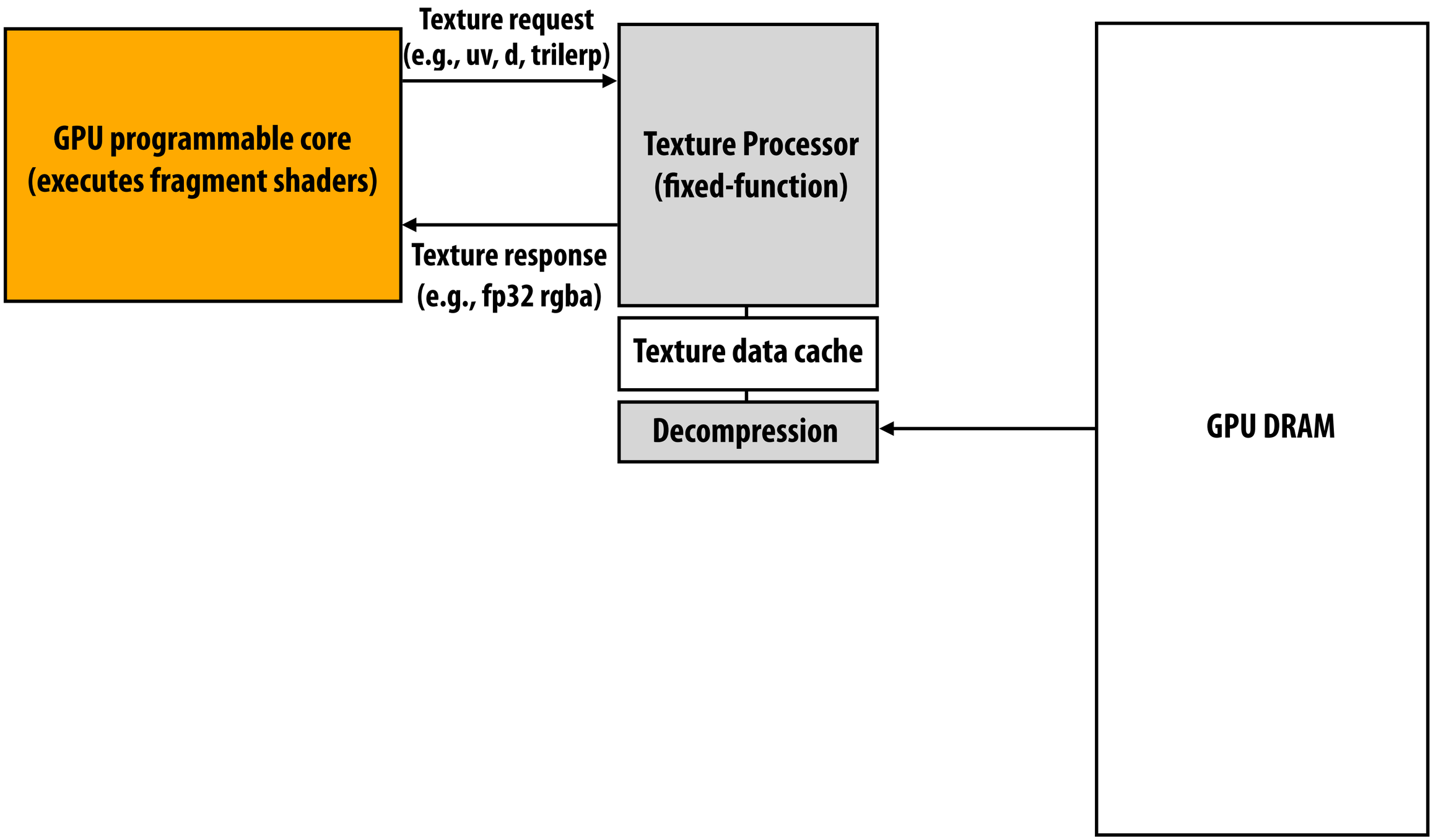
1. Compute du/dx, du/dy, dv/dx, dv/dy differentials from quad fragment texture sampling locations.
2. Compute *d*
3. Convert normalized texture coordinate uv to texel coordinates tu,tv
4. Compute required texels **
5. Load texture data in filter footprint (need eight texels for trilinear) ****
6. Perform tri-linear interpolation according to (tu,tv,d)

A texture fetch involves a significant amount of math: all modern GPUs have dedicated fixed-function hardware support for texture sampling

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

**** May involve memory fetch and decompression of texture data into texture cache

Texture system block diagram



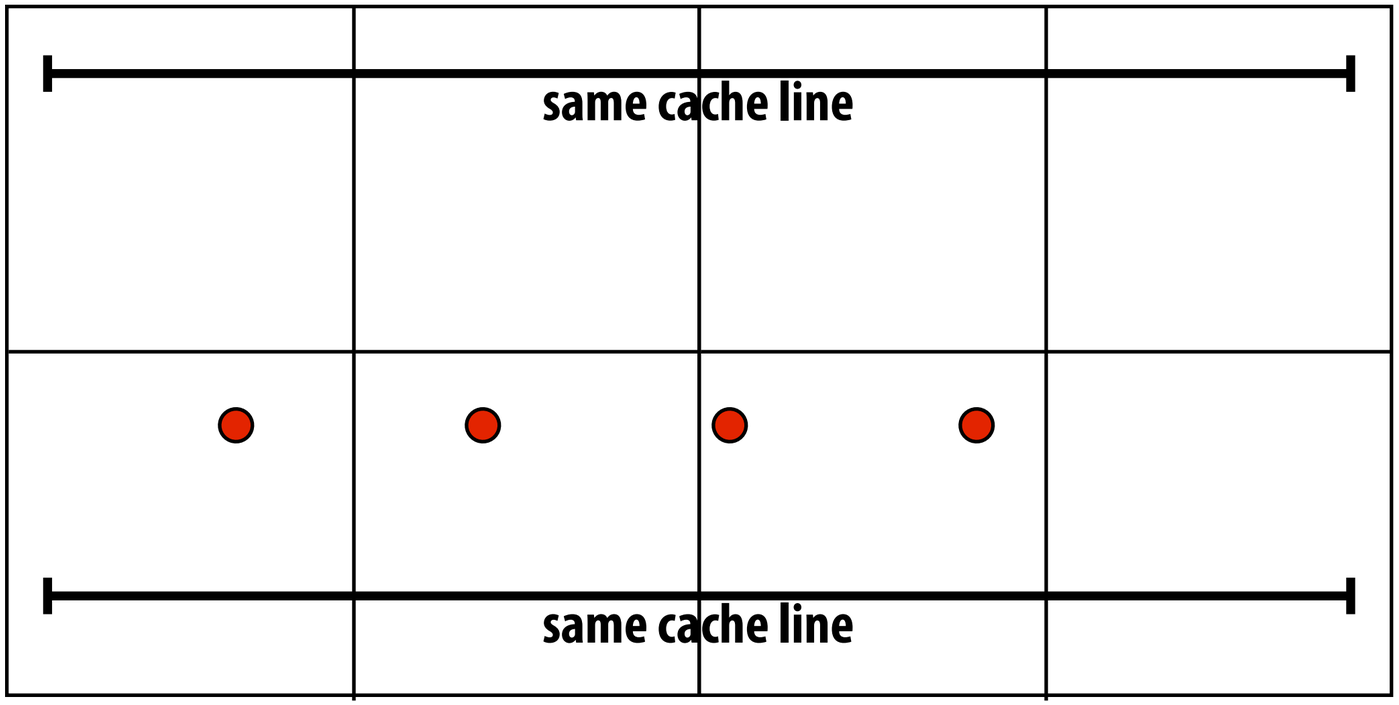
Consider memory implications of texturing

- **Texture data footprint**
  - Modern games: large textures: 10s-100s of MB
  - Film rendering: GBs to TBs of textures in scene DB
- **Texture bandwidth**
  - 8 texels per tri-linear fetch (assume 4 bytes/texel)
  - Modern GPU: billions of fragments/sec
    (NVIDIA GTX 580: ~40 billion/sec)
- **A performant graphics system needs:**
  - High memory bandwidth
  - Texture caching
  - Texture data compression
  - Latency hiding solution to avoid stalls during texture data access

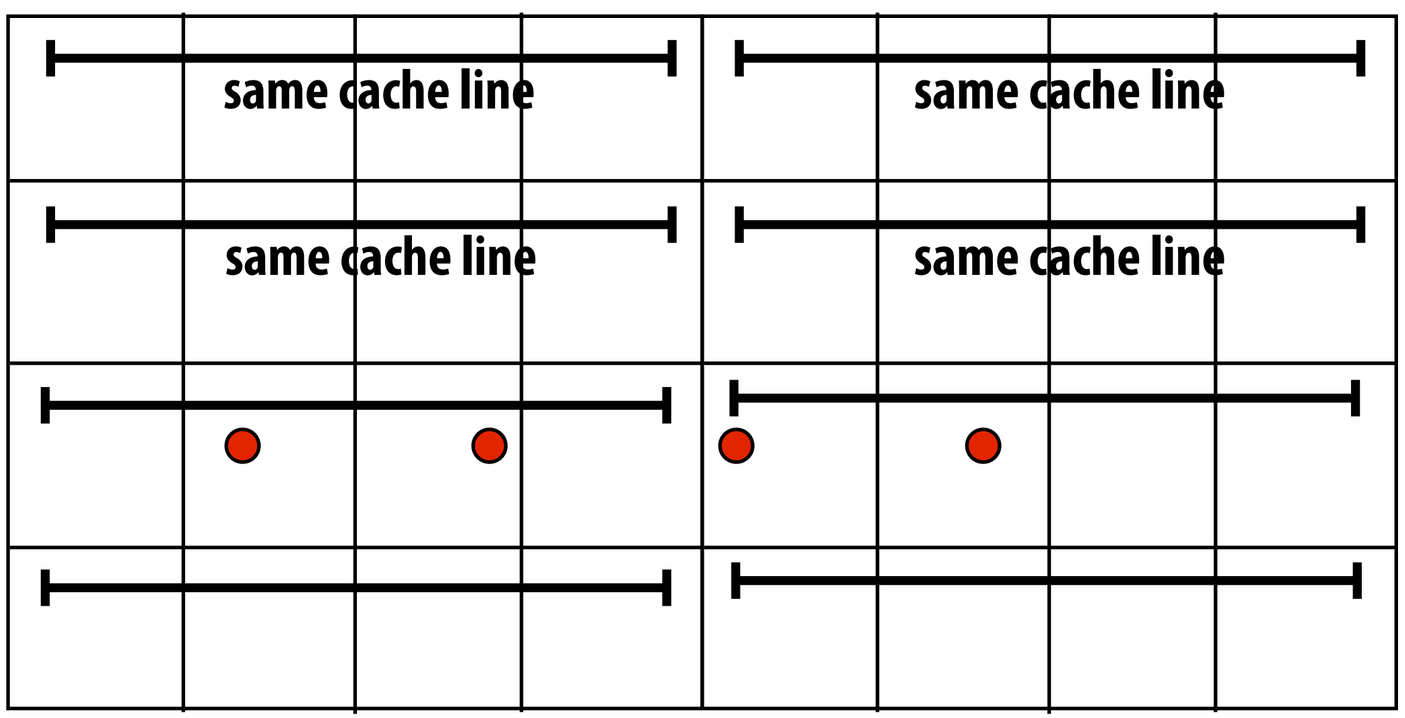
Review: the role of caches in CPUs

- 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

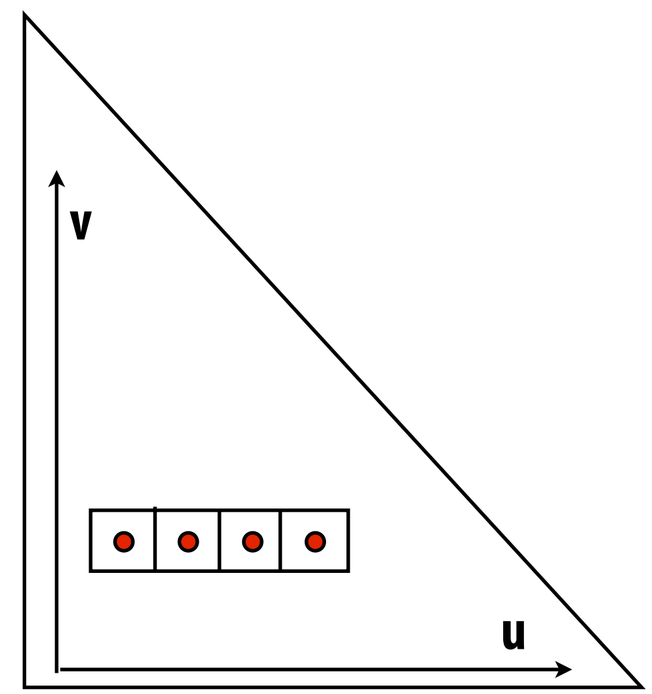


mip-map: level *d+1* texels



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)

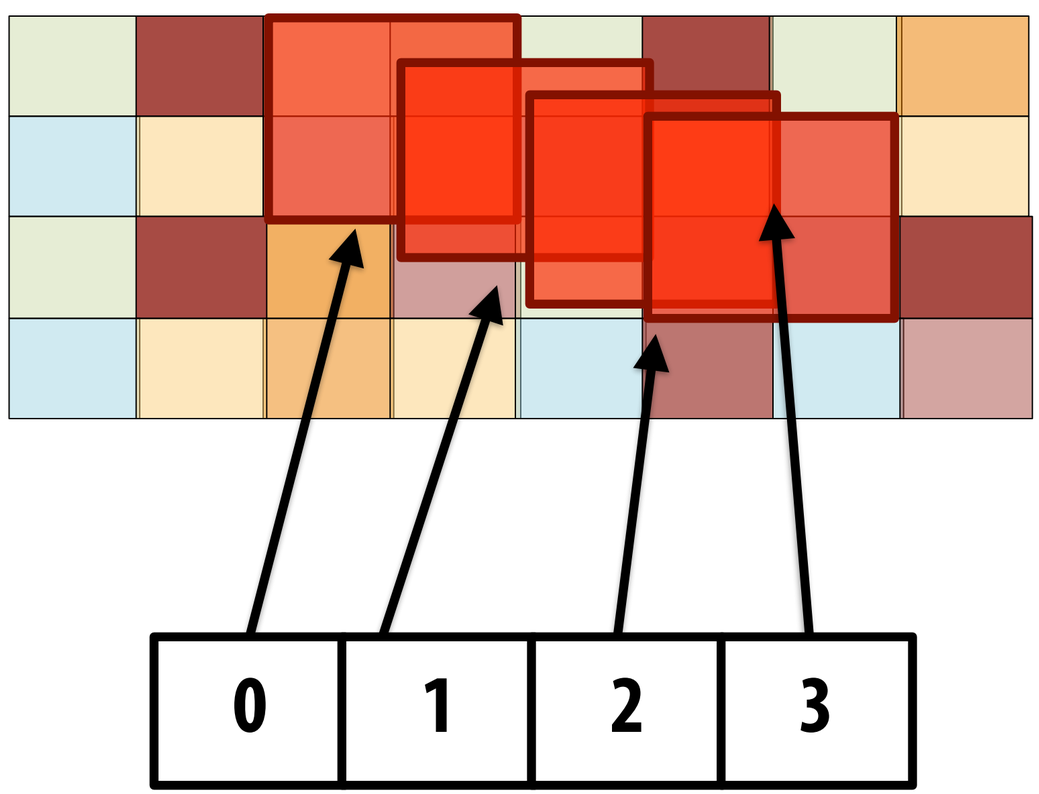
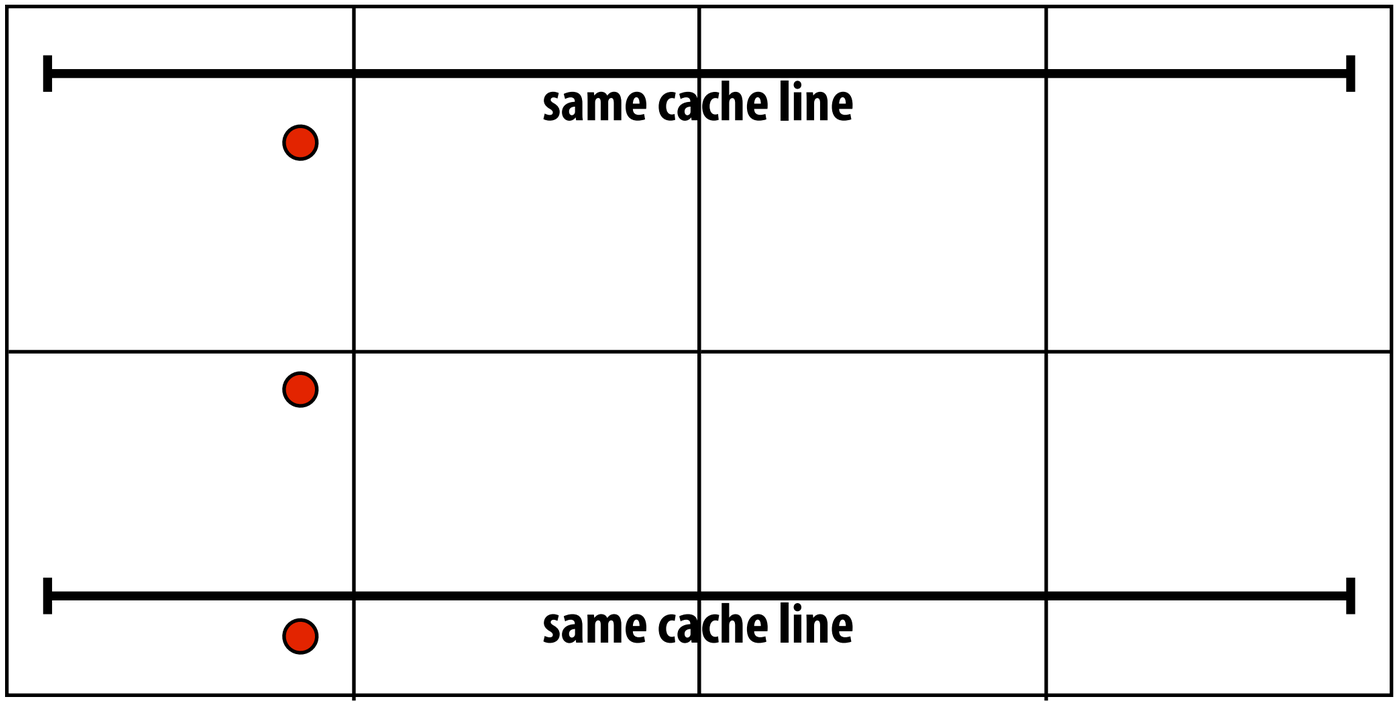
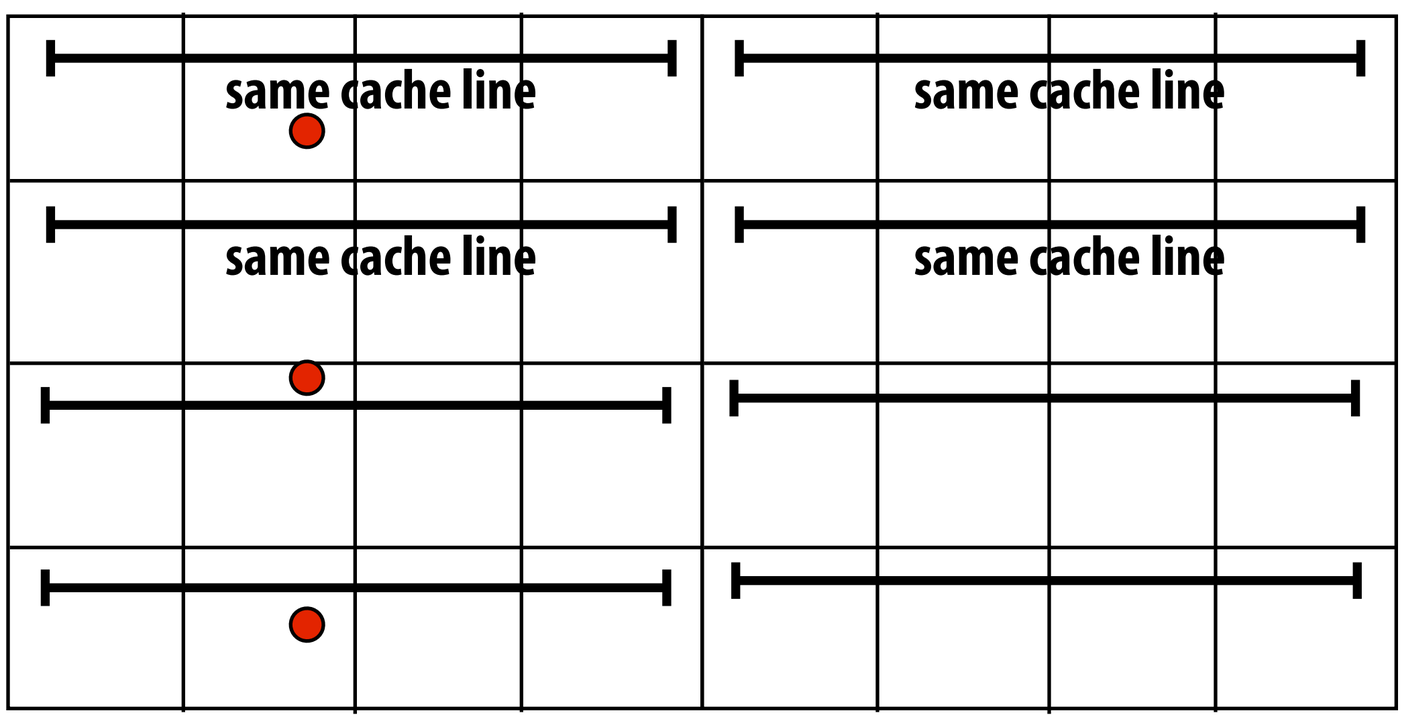


Figure illustrates filter support regions from texture fetches from four adjacent fragments

Now rotate triangle on screen

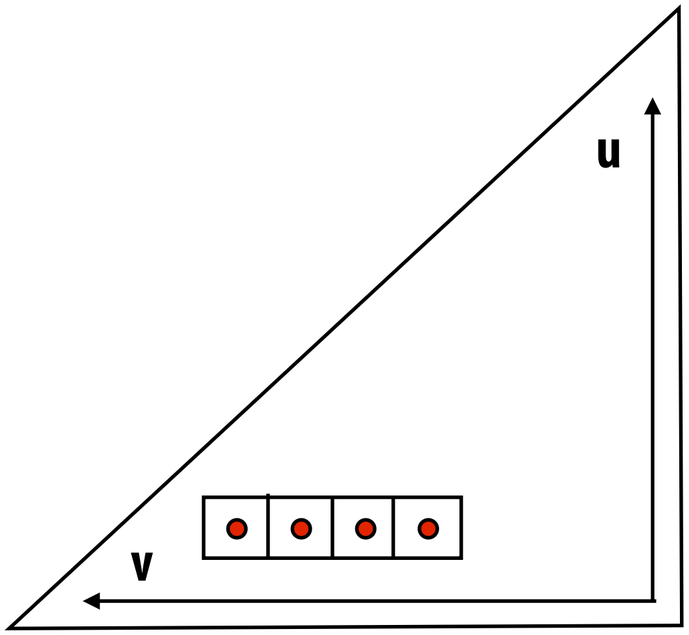


mip-map: level *d+1* texels

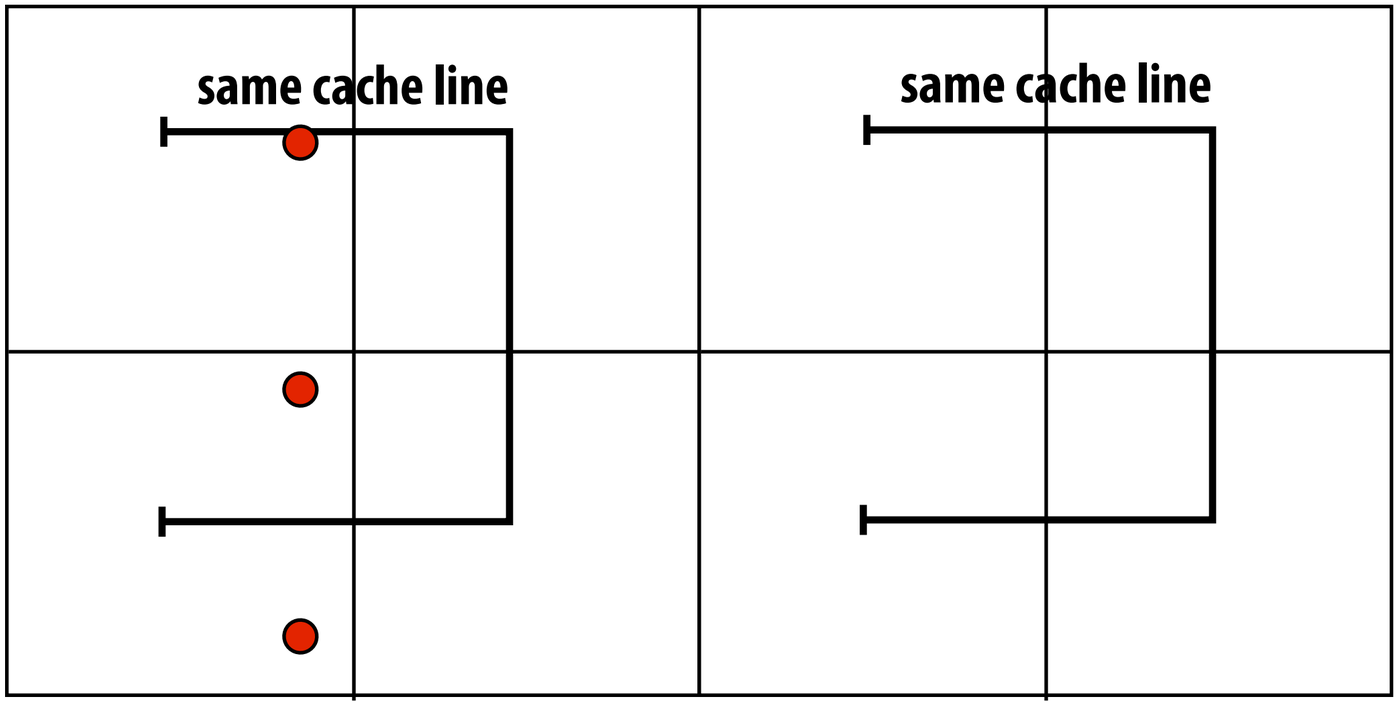


mip-map: level *d* texels

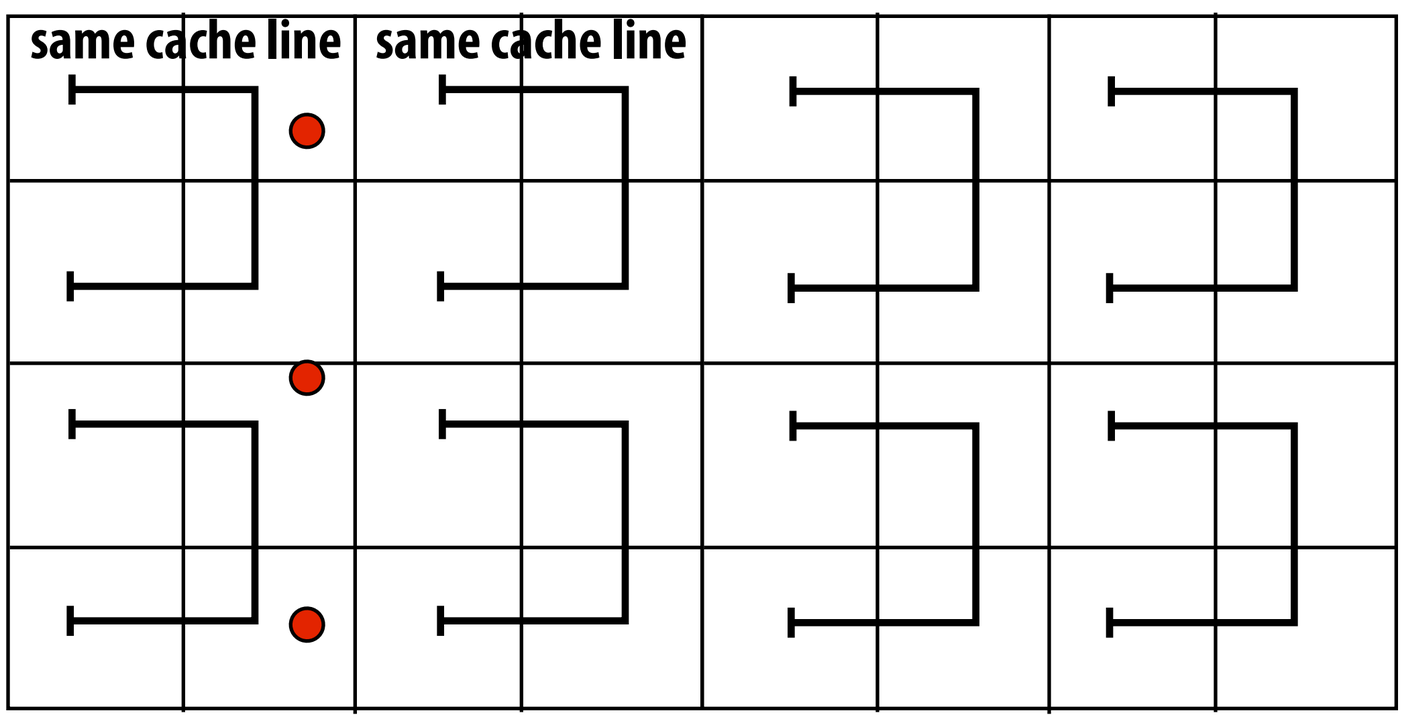
Assume:
Row-major rasterization order
Horizontal texels contiguous in memory
Cache line = 4 texels



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



mip-map: level *d+1* texels



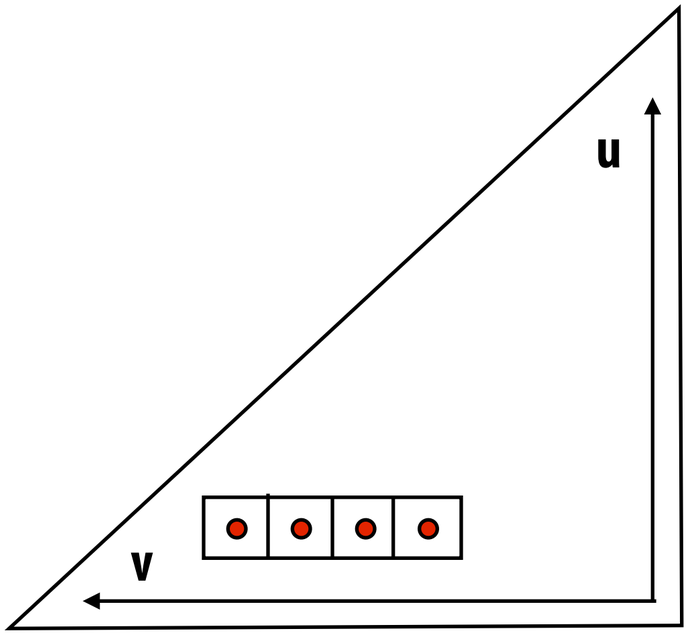
mip-map: level *d* texels

Assume:

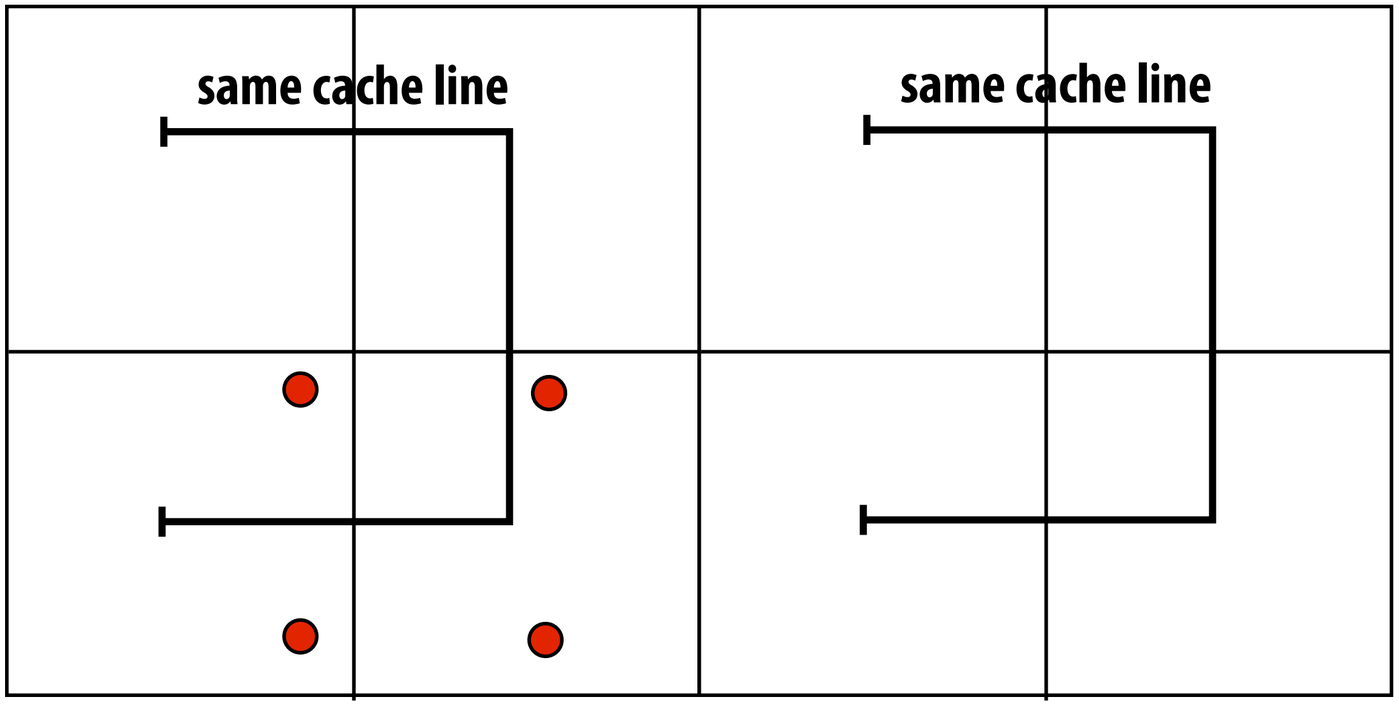
Row-major rasterization order

2D blocks of texels contiguous in memory

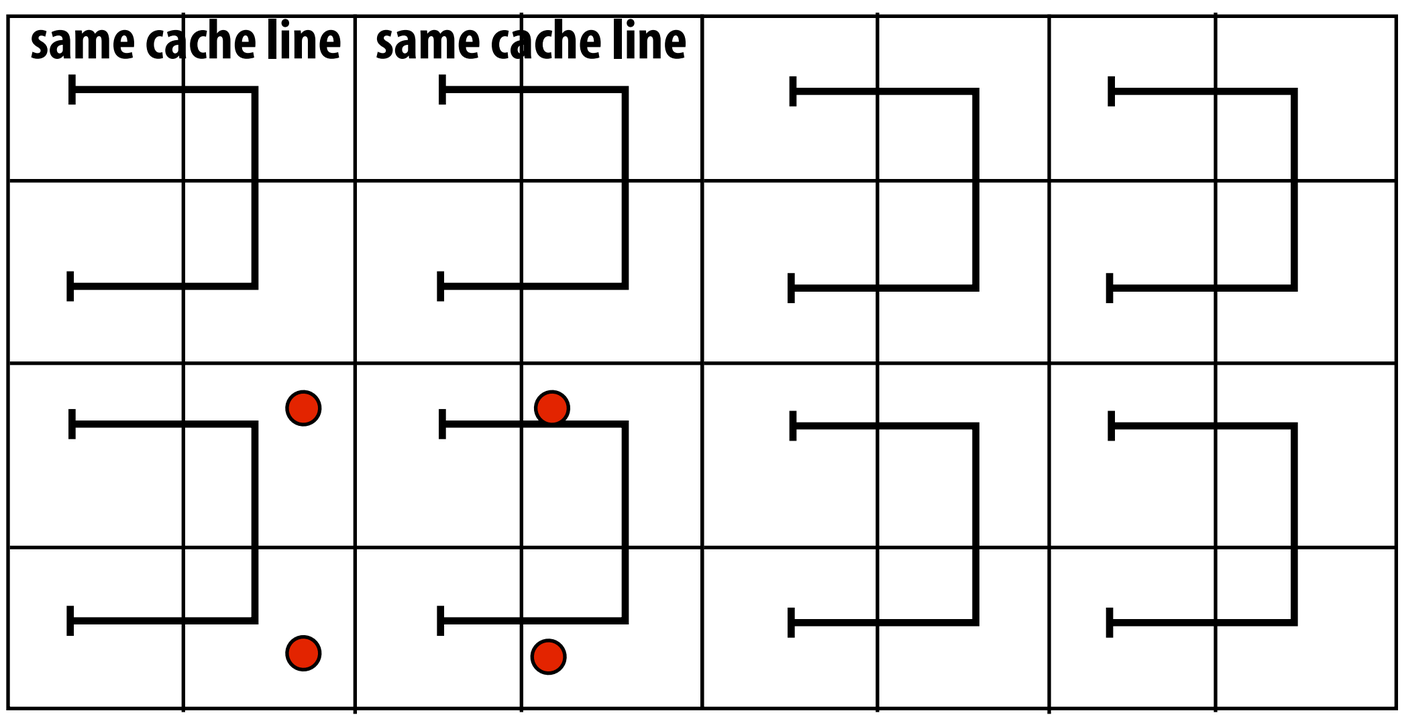
Cache line = 4 texels



Tiled rasterization increases reuse



mip-map: level *d+1* texels



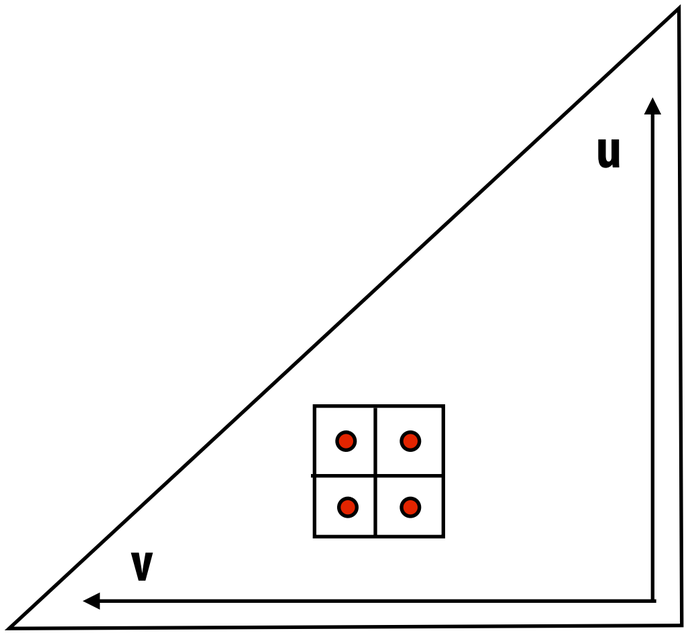
mip-map: level *d* texels

Assume:

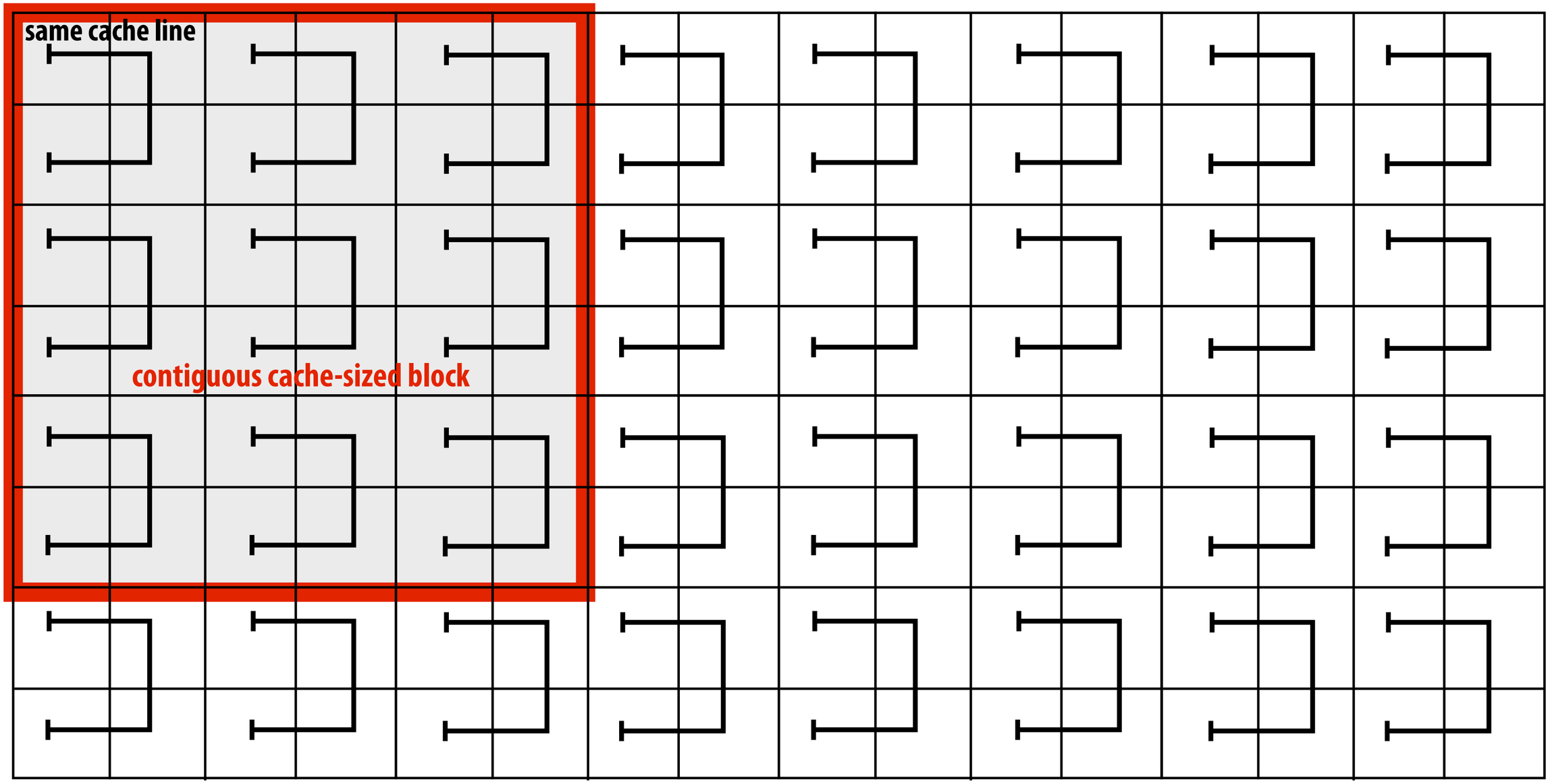
Blocked rasterization order

2D blocks of texels contiguous in memory

Cache line = 4 texels



6D blocking further reduces conflicts



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 summary

- **Pre-filtering texture data reduces aliasing**
  - Mip-mapping algorithm is fundamental to texture system design
  - Used to avoid aliasing under minification
  - Also improves cache behavior under minification

- **A texture lookup is a lot more than a 2D array access**
  - It is a resampling operation, with an adaptive filter footprint
  - Significant computational expense, implemented in specialized fixed-function hardware on GPUs

- **GPU texture caches:**
  - Primarily serve to decrease bandwidth requirements of DRAM
  - Not to reduce average data access latency
  - Small in size, multi-ported (e.g., need to access 8 texels simultaneously for trilinear filtering)

- **Tiled rasterization order, tiled texture layouts serve to increase cache hits**

- **Next lecture: texture compression and latency hiding**

Readings

- H. Igehy et al., *Prefetching in a Texture Cache Architecture*. Graphics Hardware 1998