Lecture 5:

Texturing

Visual Computing Systems
CMU 15-869, Fall 2013

Today: texturing!

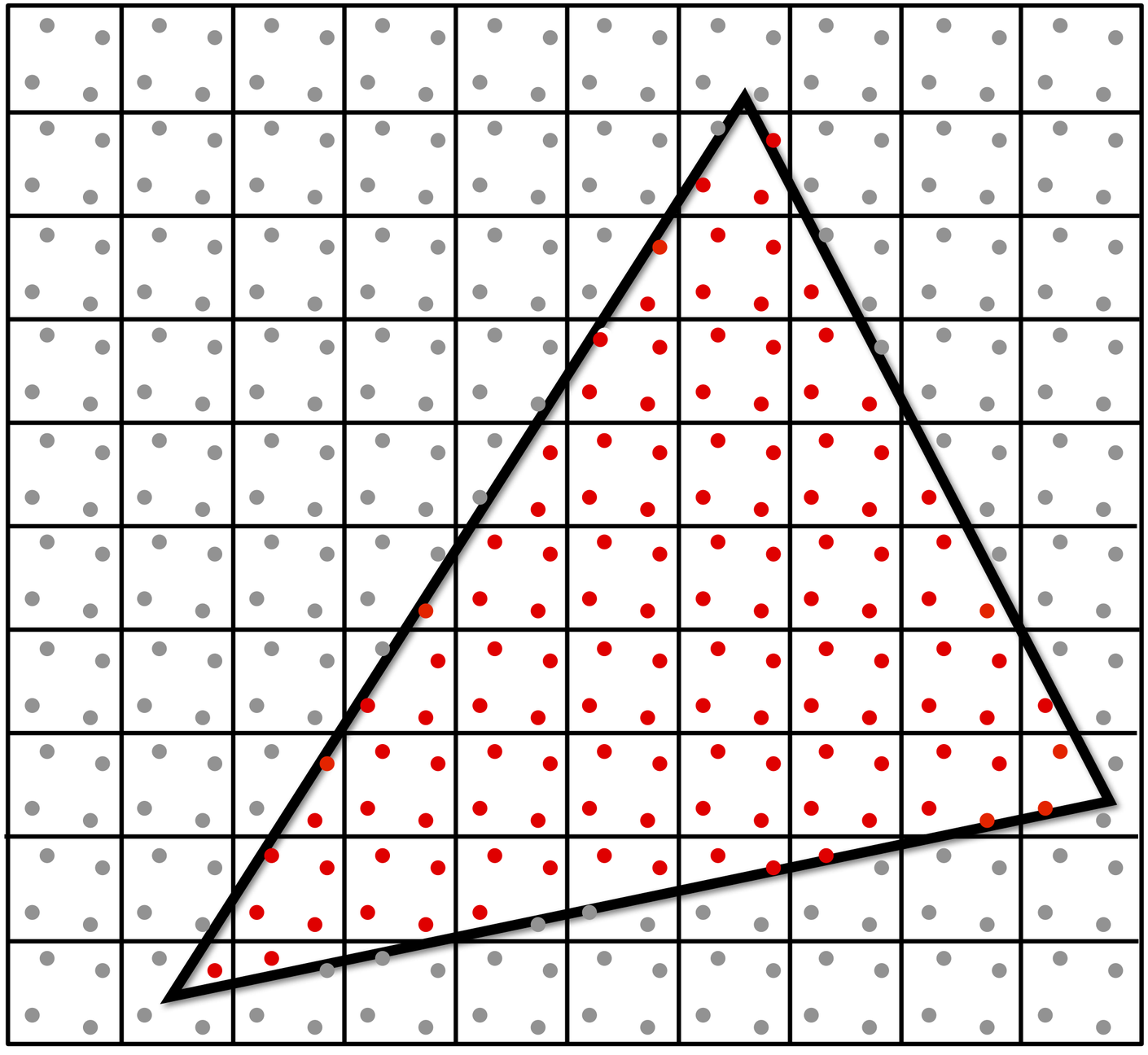
- Texture filtering
  - A texture access is not just a 2D array lookup ;-)

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

Last time

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

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

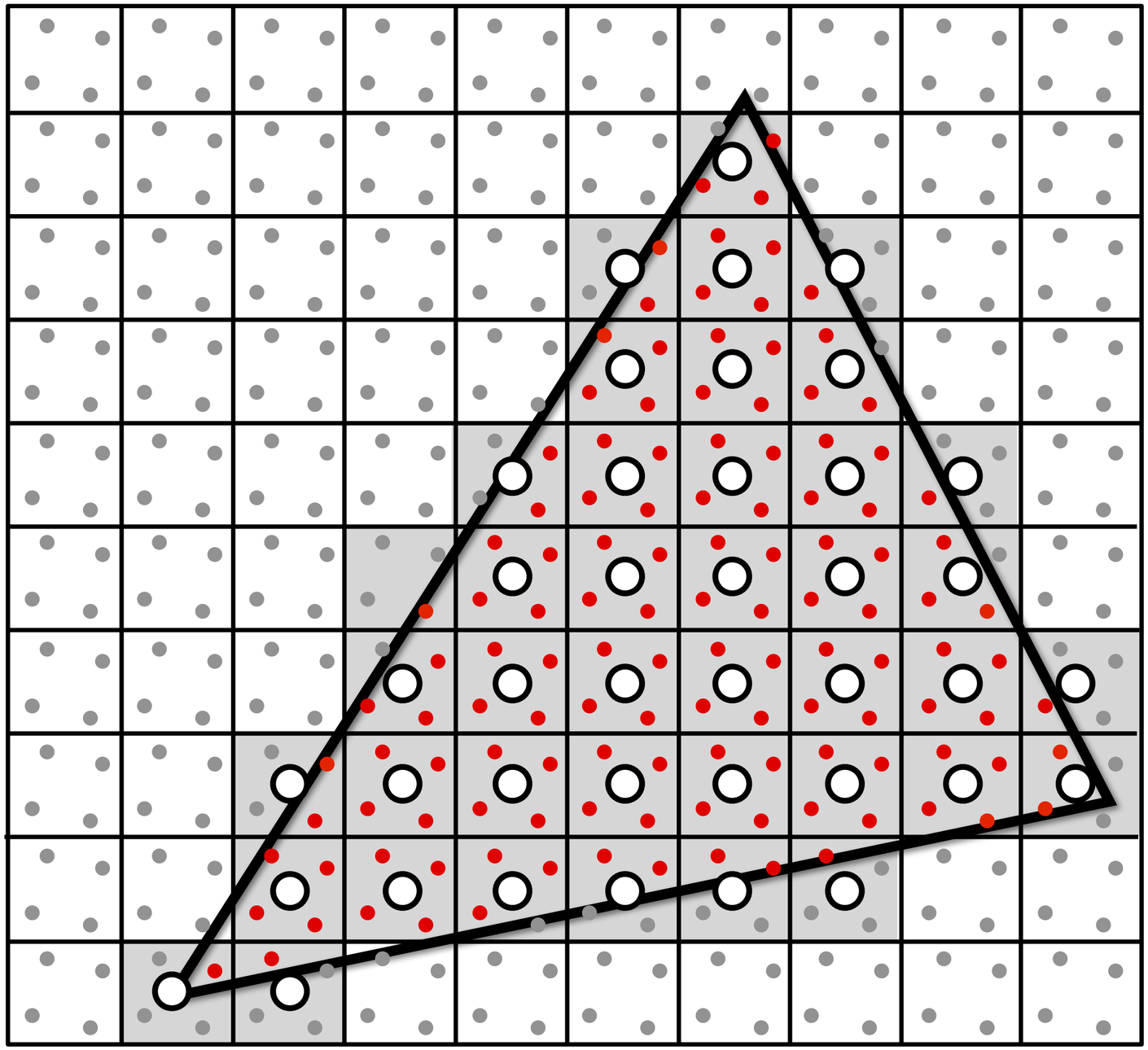


Generating fragments via "multi-sampling"

Last class: one fragment per covered visibility sample (if multiple samples per pixel: "supersampling")

Today:

- **One fragment per covered pixel (if any visibility sample in a pixel is covered, generate fragment) ****
- **Surface attributes for fragment shading [typically] sampled at pixel centers**



**** 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 behavior of fragment processing stage

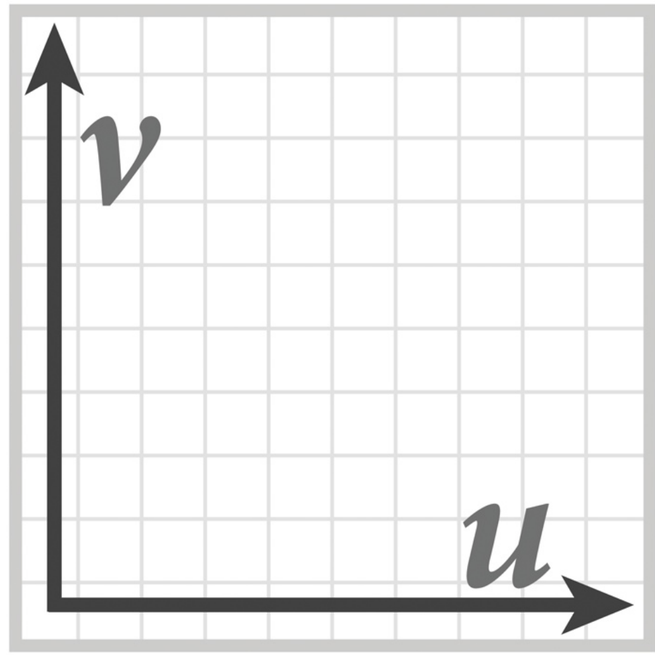
```
sampler mySampler;
Texture2D<float3> myTex;
float3 lightDir;

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);
}
```

Let:

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

`myTex =`

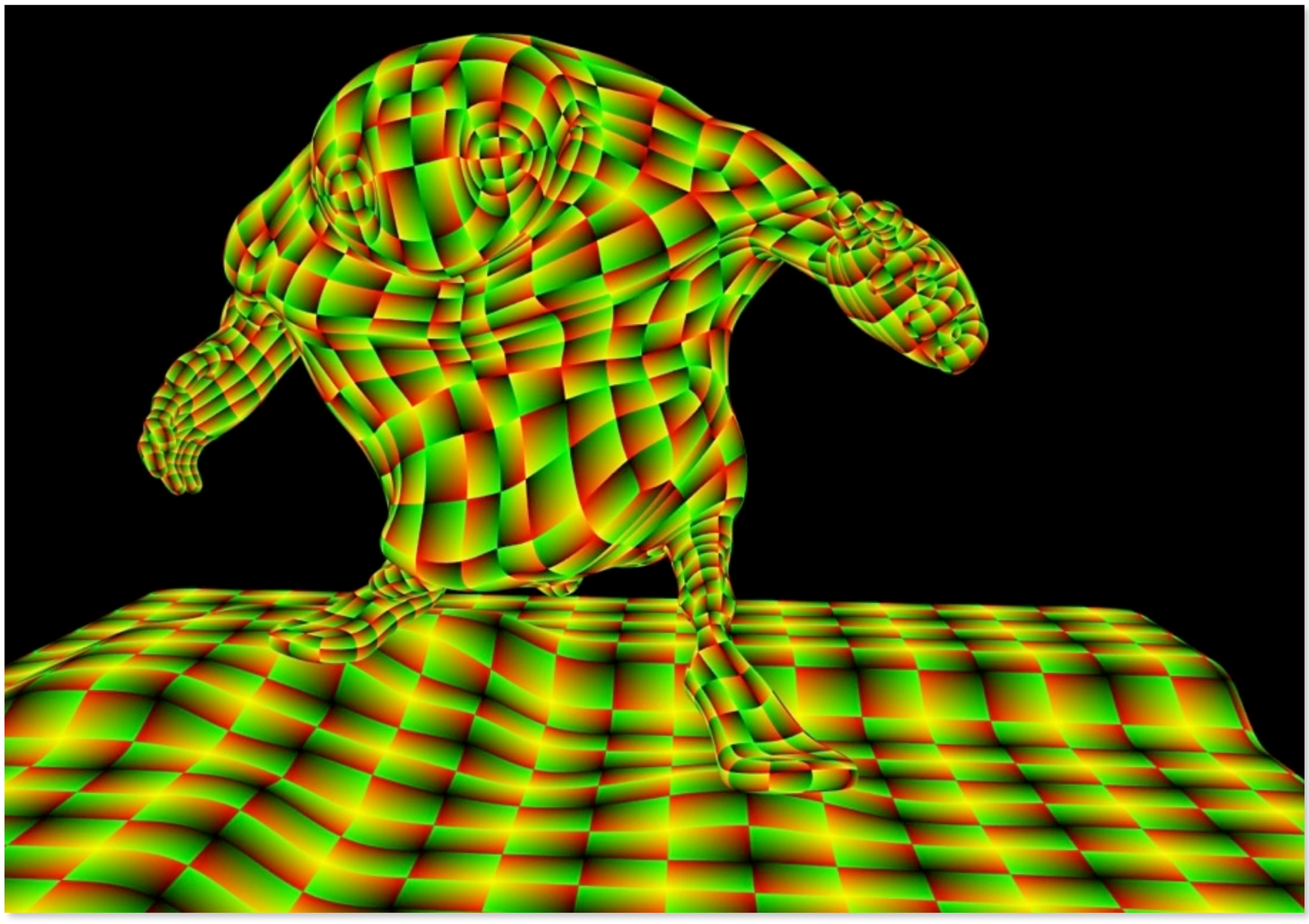


`myTex` is a function defined on $[0,1]^2$ domain:
myTex : $[0,1]^2 \rightarrow$ float3
(represented by 2048x2048 image)

`mySampler` defines how to sample from texture to generate value at (u,v)

Texture coordinates

Mapping from point on surface to point (uv) in texture domain



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

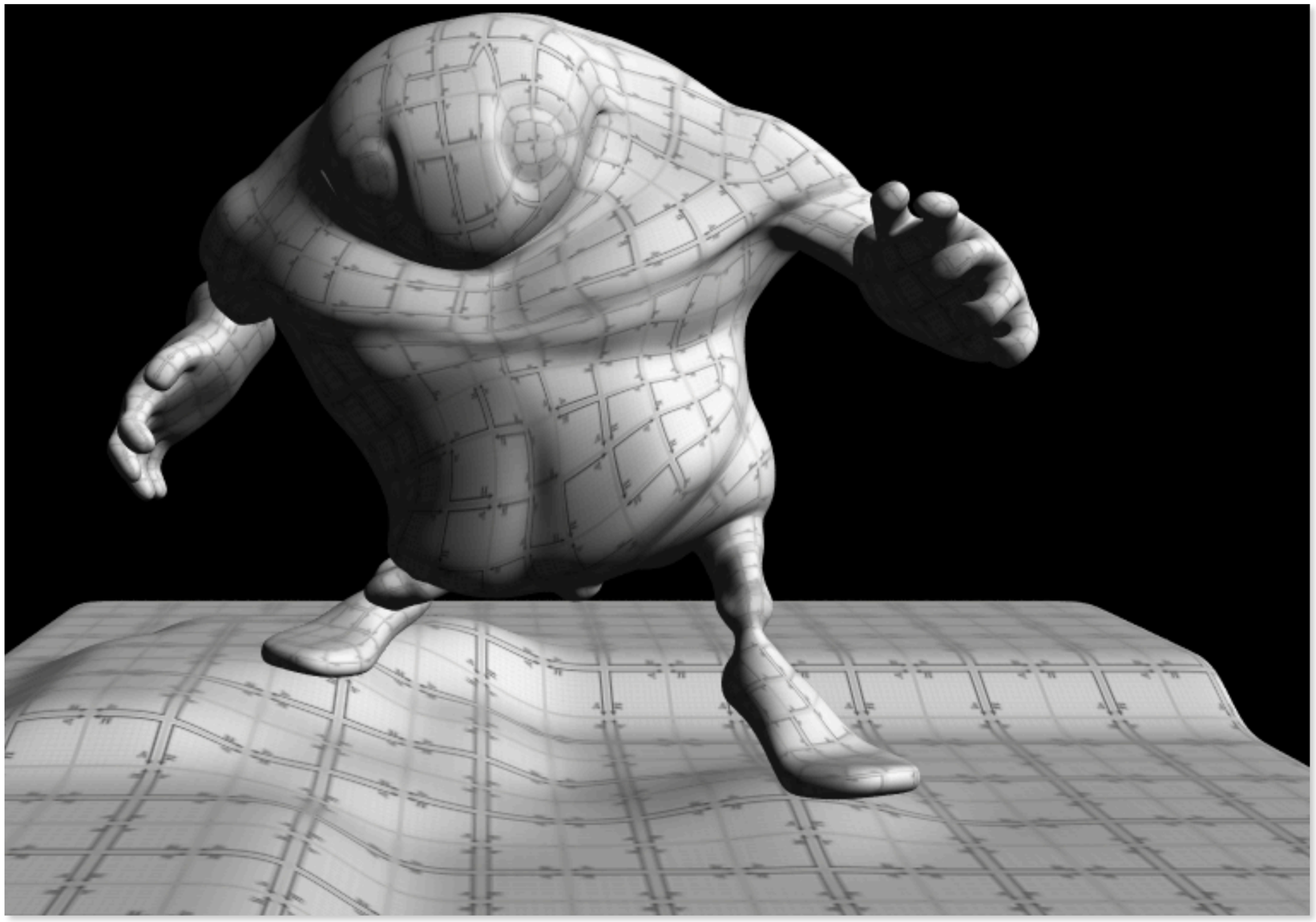
Red channel = u

Green channel = v

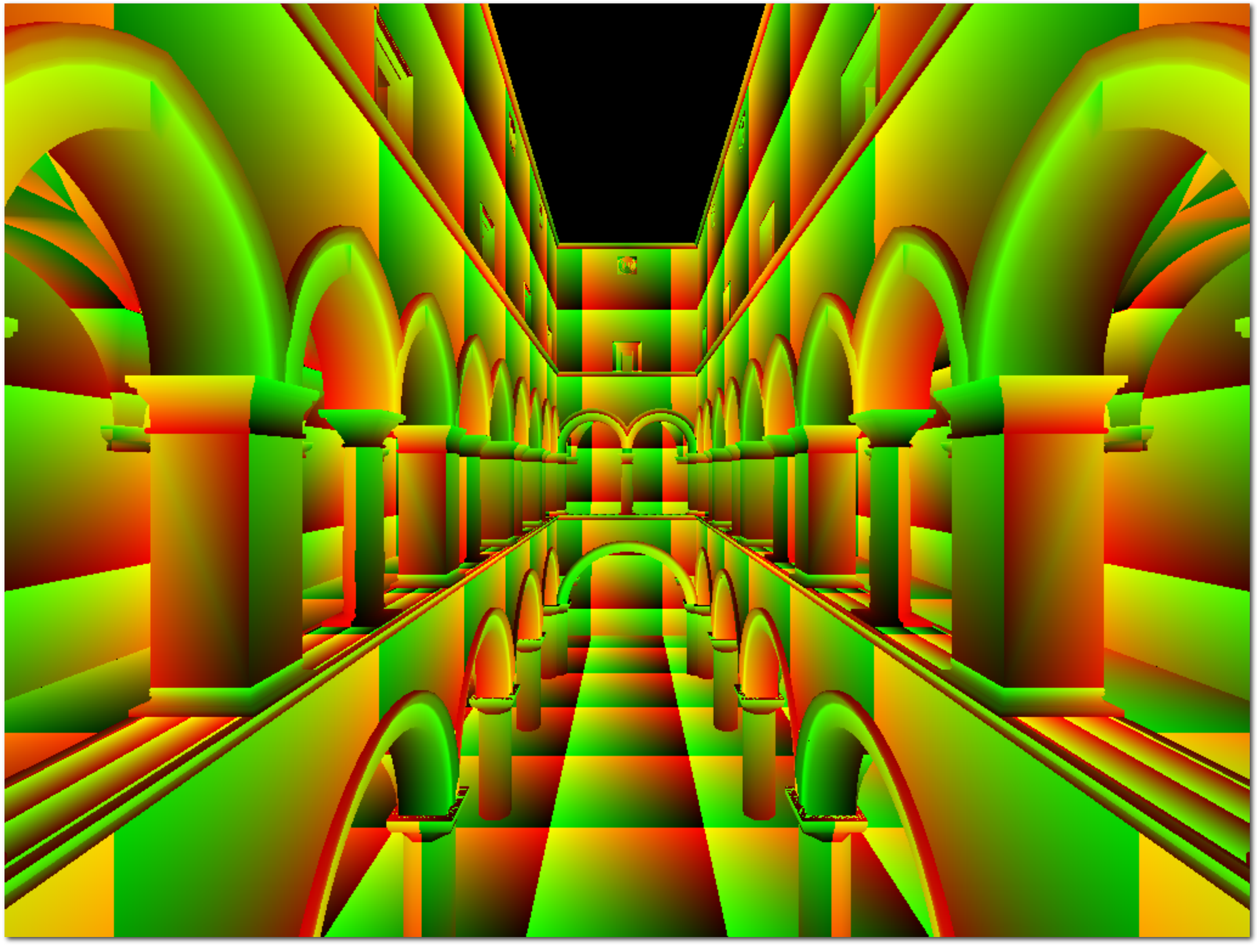
So uv=(0,0) is black, uv=(1,1) is yellow

*** Texture coordinates are also often written as (s,t) (uv is typically the parametric coordinates of the surface)**

Shaded result

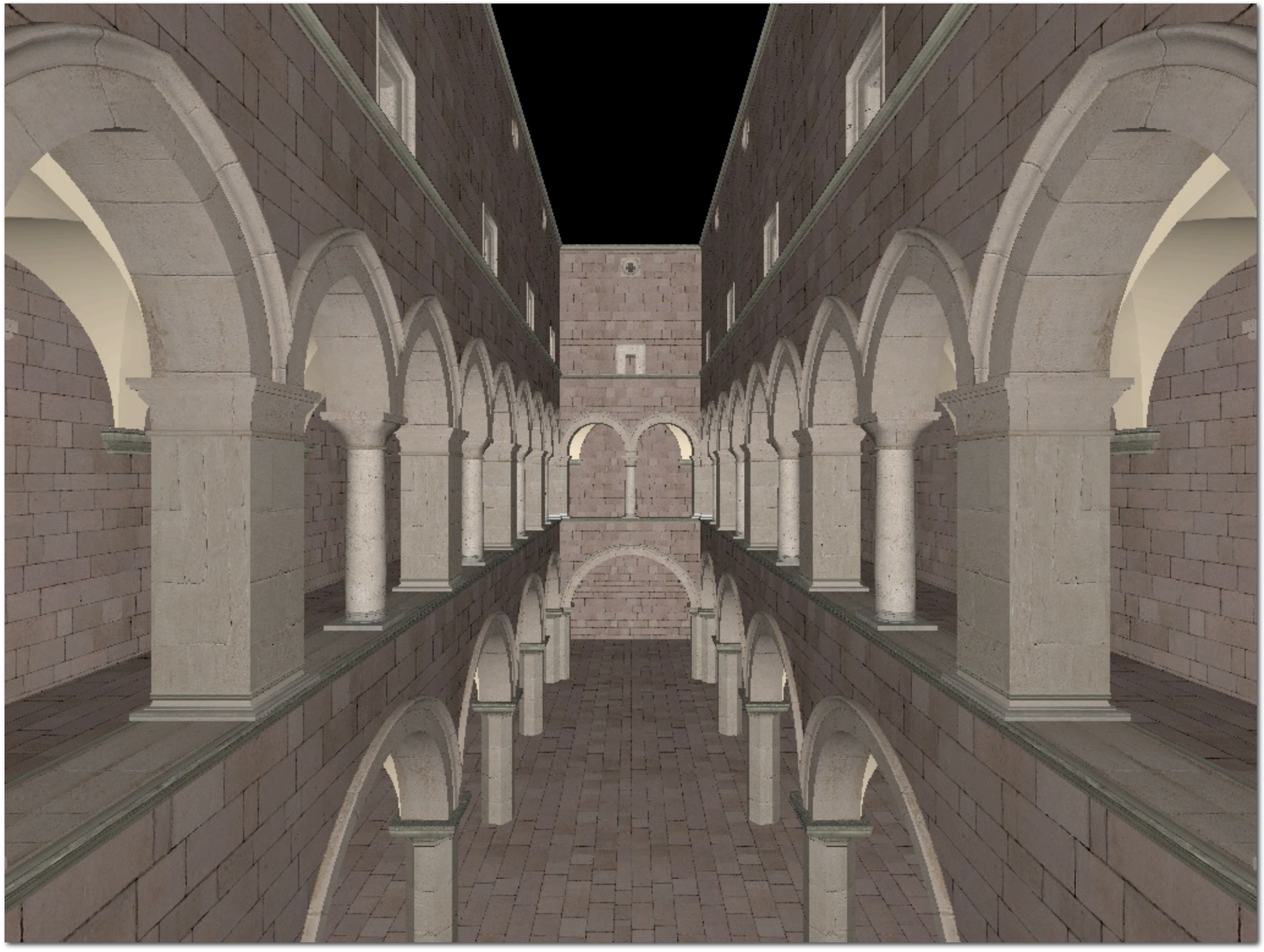


Another example: Sponza

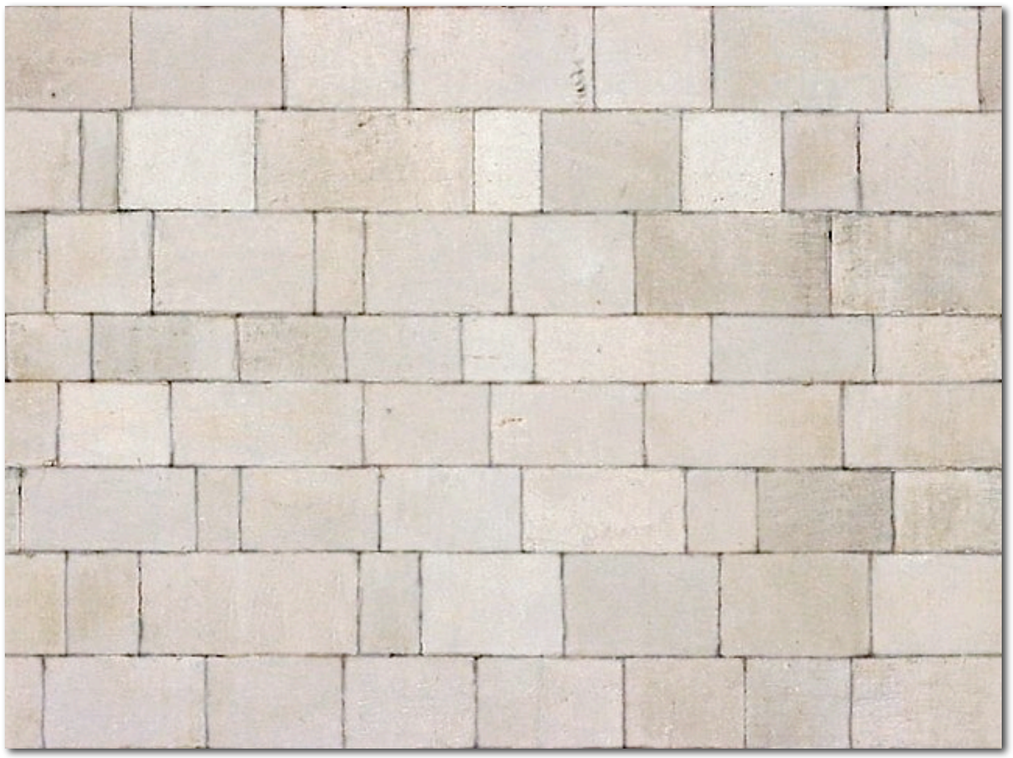


Red channel = u
Green channel = v
So uv=(0,0) is black, uv=(1,1) is yellow

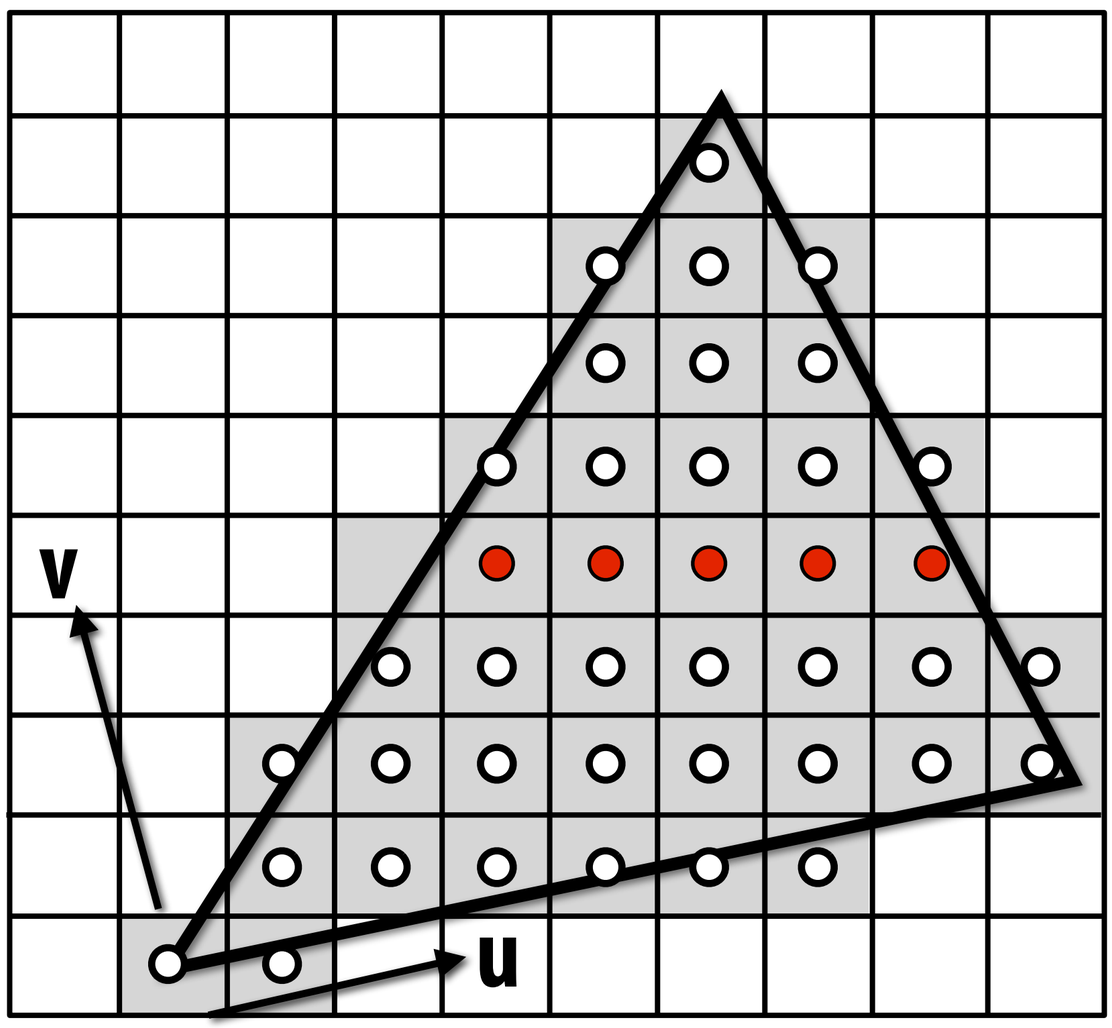
Textured Sponza



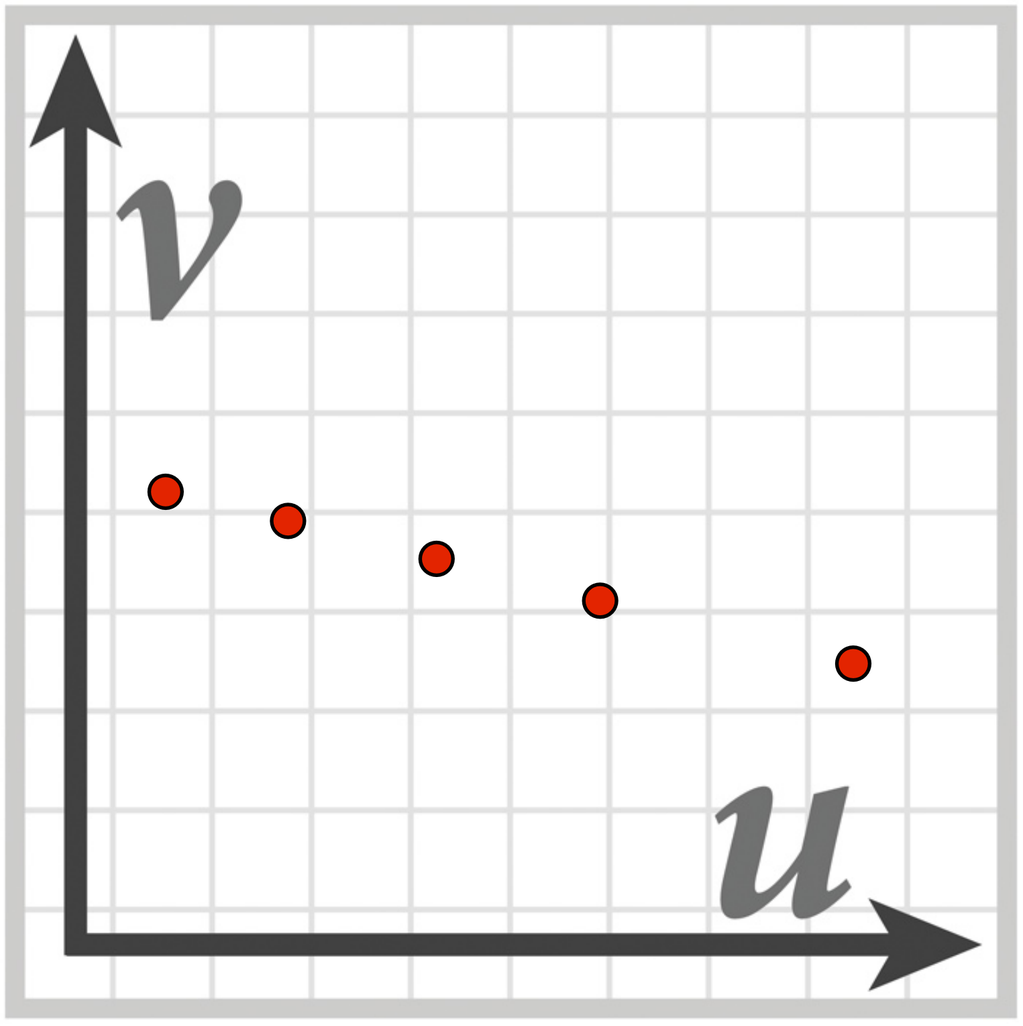
Examples of textures used in Sponza



Texture space

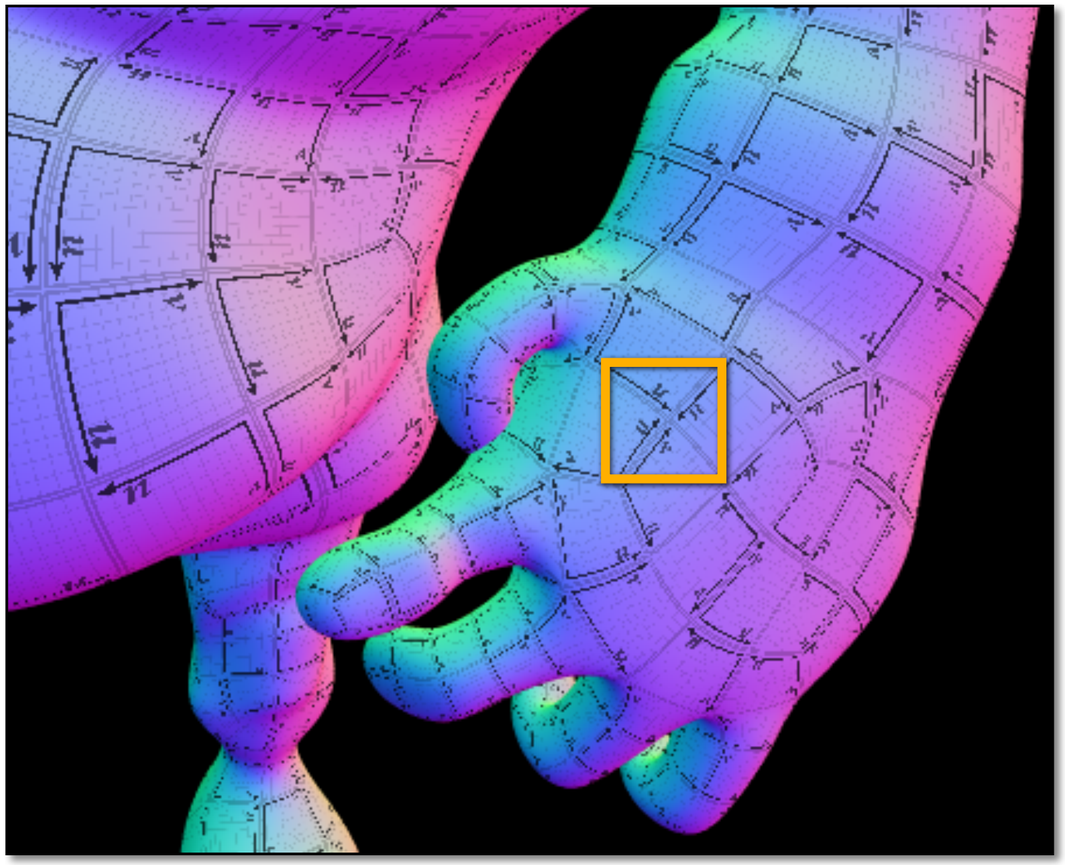


Positions of surface appearance samples in screen space (graphics pipeline samples triangle's appearance at these locations)

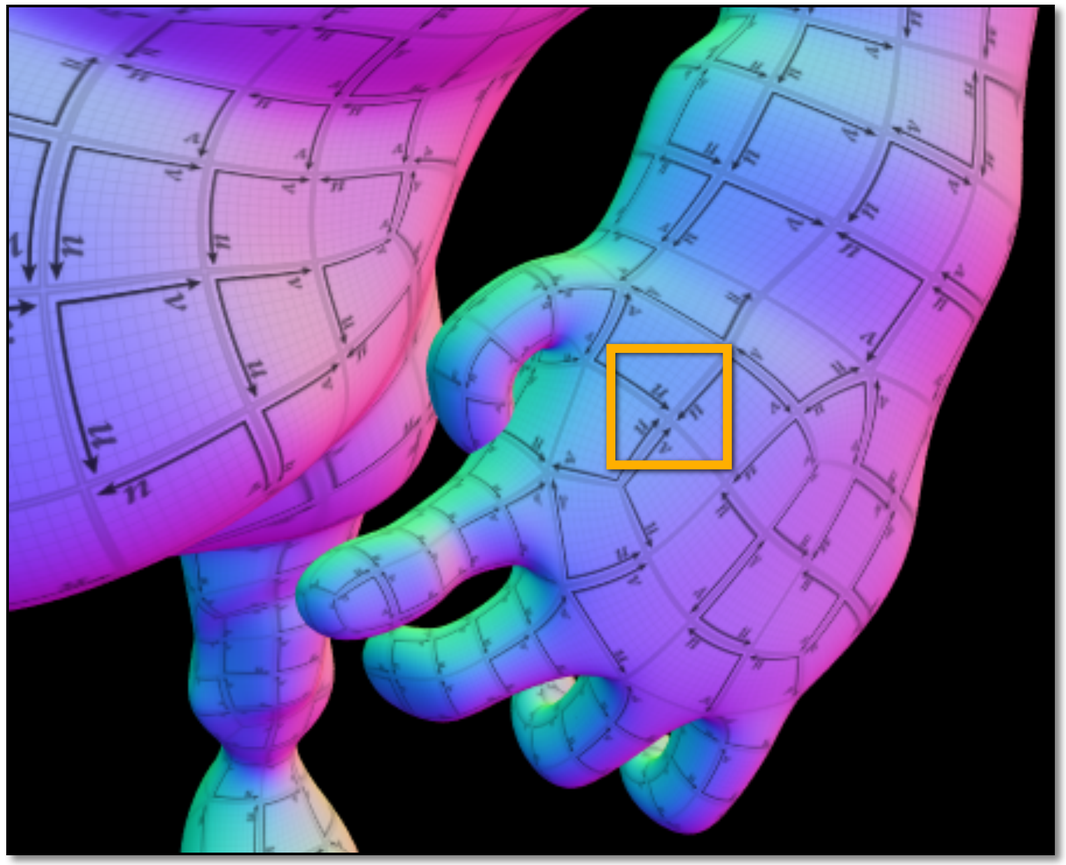


Positions of corresponding texture samples in texture space (texture is sampled at these locations as part of shading)

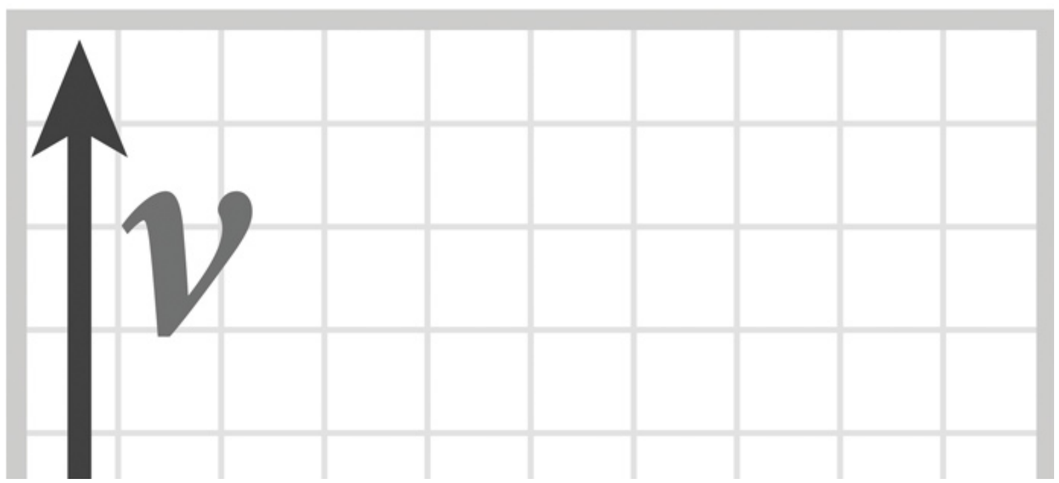
Aliasing due to undersampling



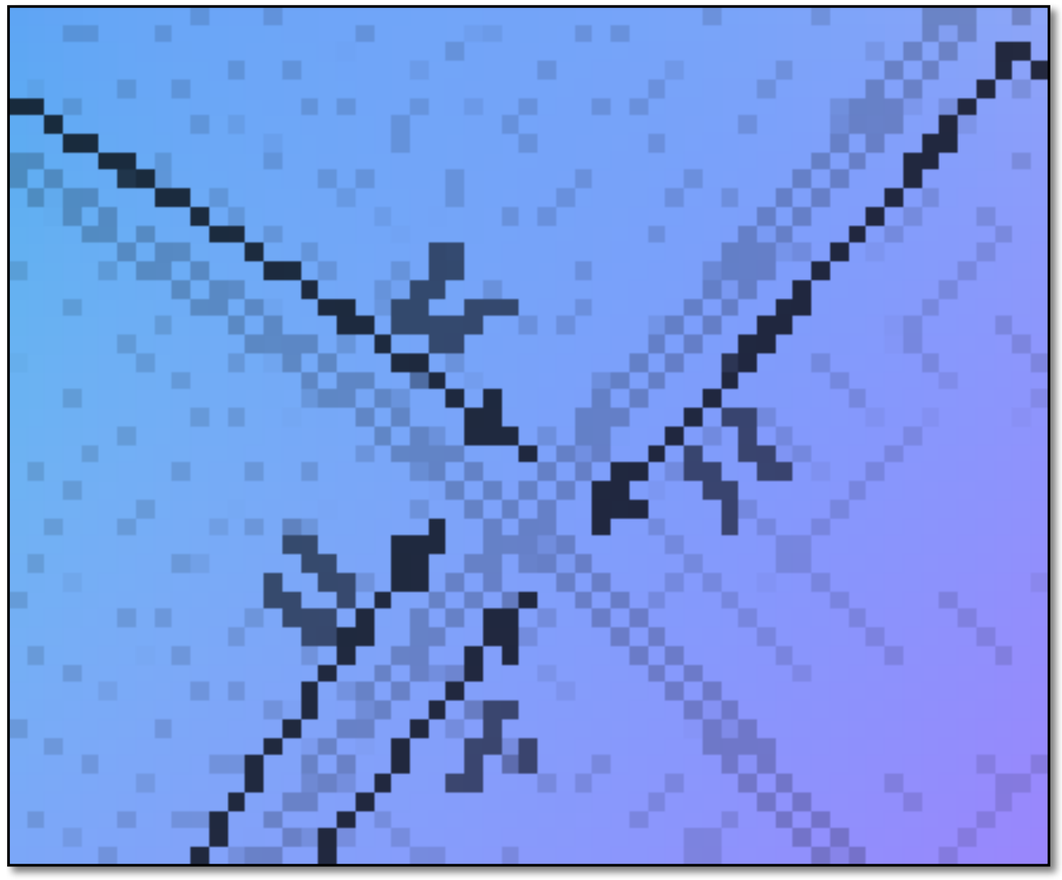
No pre-filtering of texture data
(resulting image exhibits aliasing)



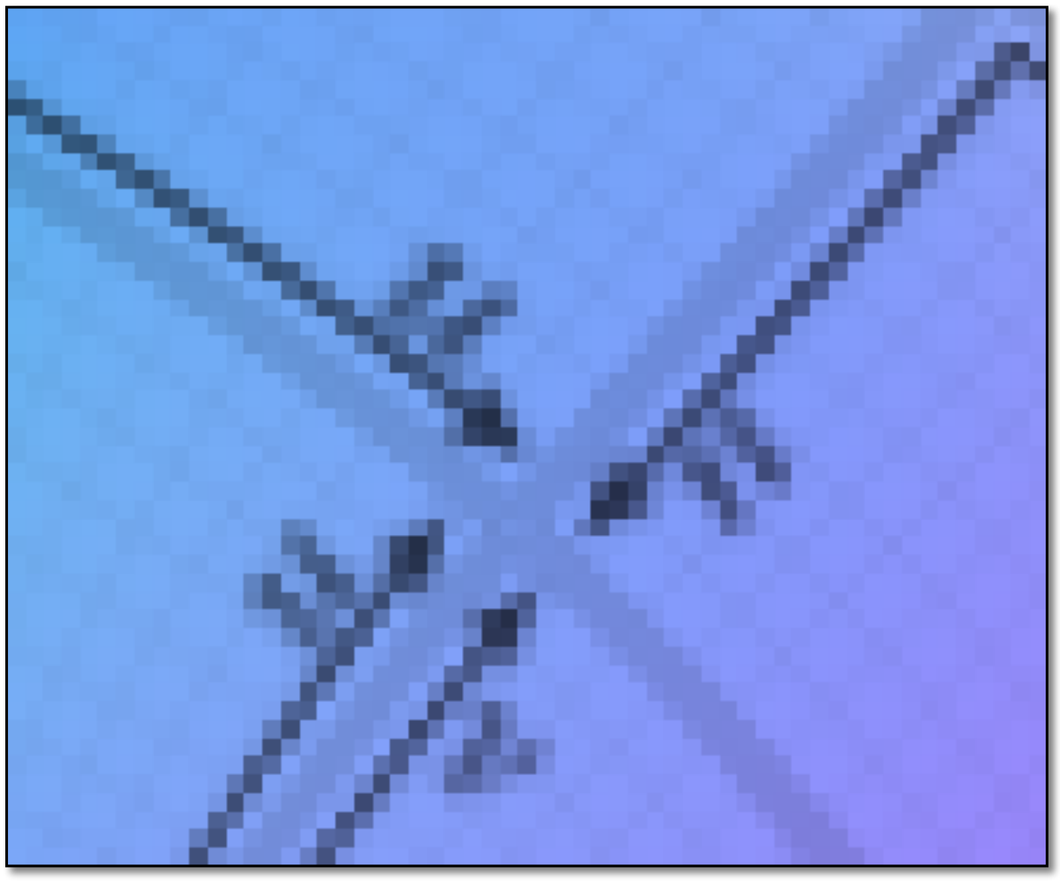
Rendering using pre-filtered texture data



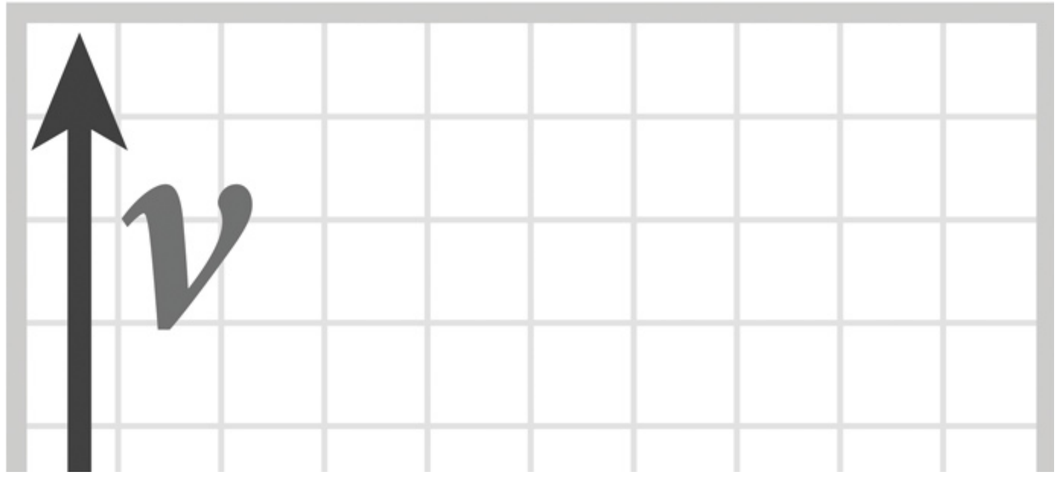
Aliasing due to undersampling



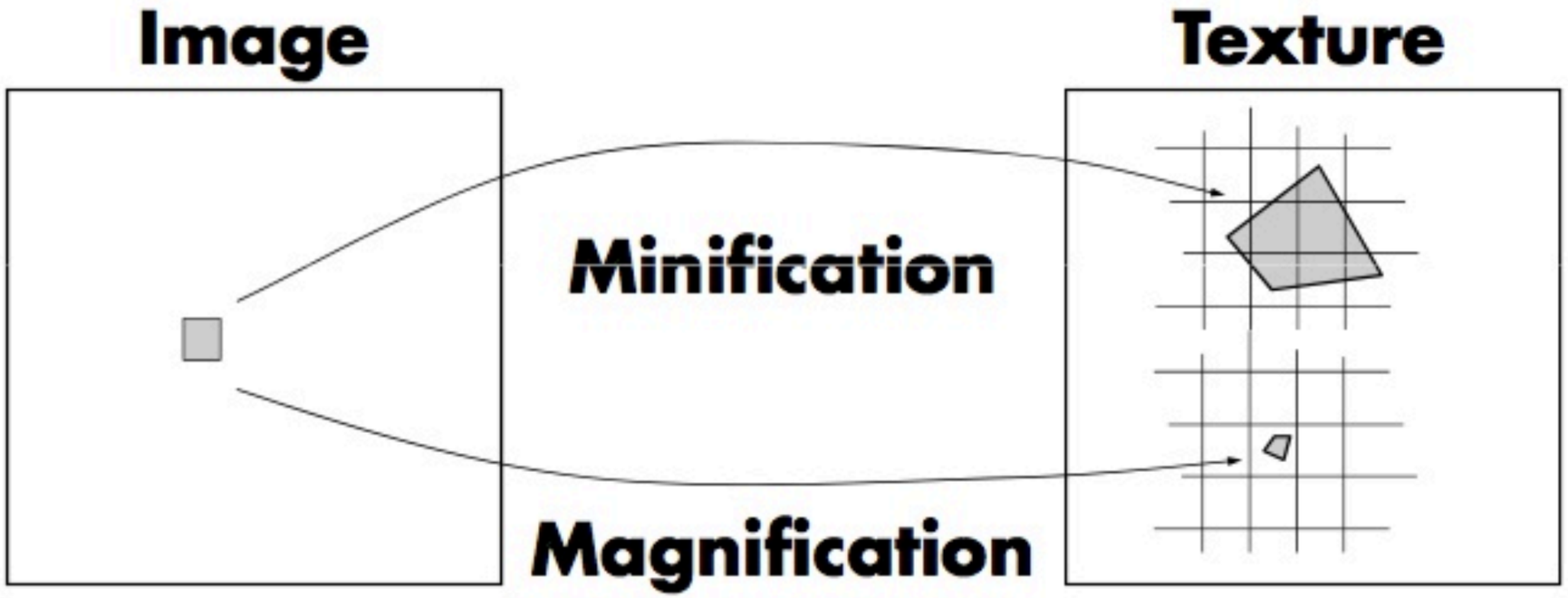
No pre-filtering of texture data
(resulting image exhibits aliasing)



Rendering using pre-filtered texture data



Filtering textures

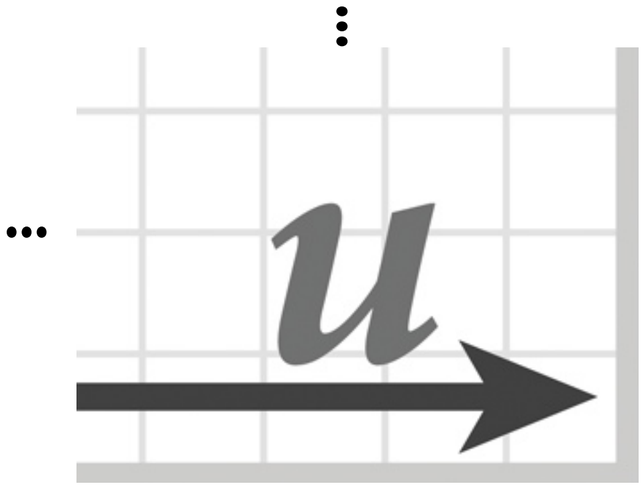


- **Minification:**
  - Area of texel corresponds to far less than a pixel on screen
  - Area of screen pixel maps to large region of texture (filtering required -- averaging)
  - Example: when scene object is very far away

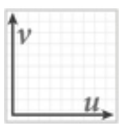
- **Magnification:**
  - Area of texel maps to many screen pixels
  - Area of screen pixel maps to tiny region of texture (interpolation required)
  - Example: when camera is very close to scene object (need higher resolution texture map)

Figure credit: Akeley and Hanrahan

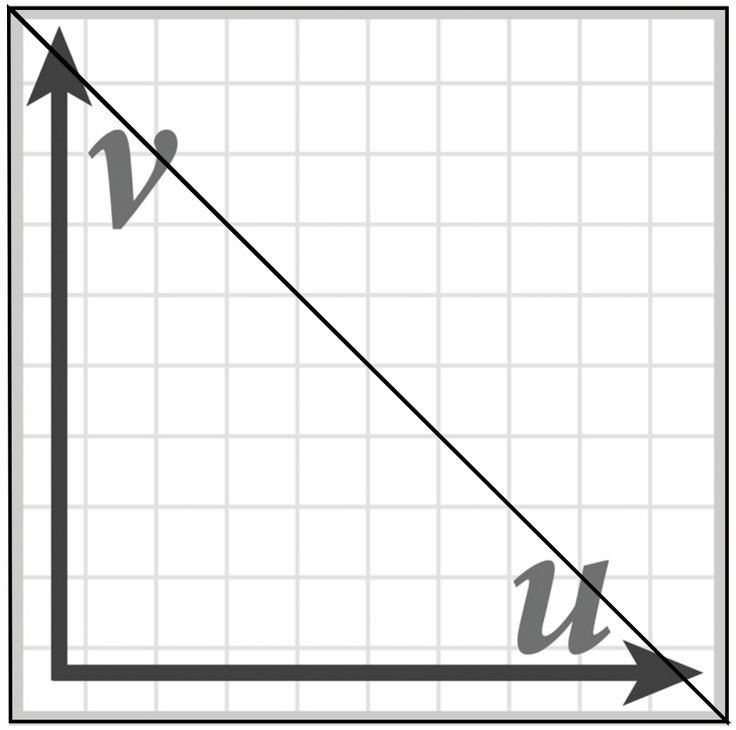
Filtering textures



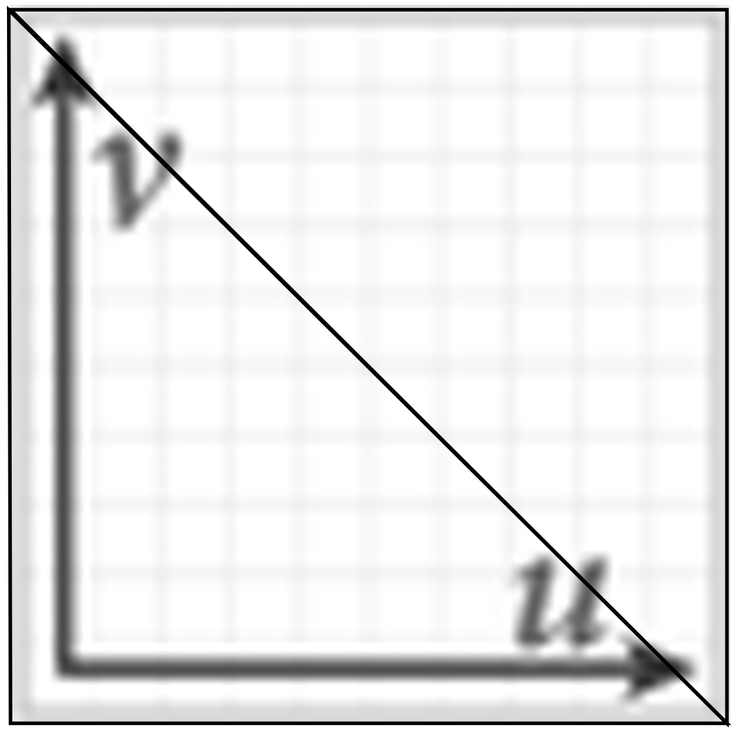
Actual texture: 700x700



Actual texture: 64x64

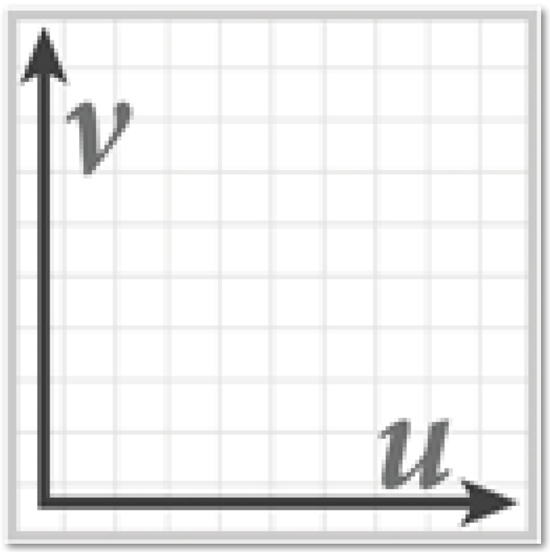


Texture minification

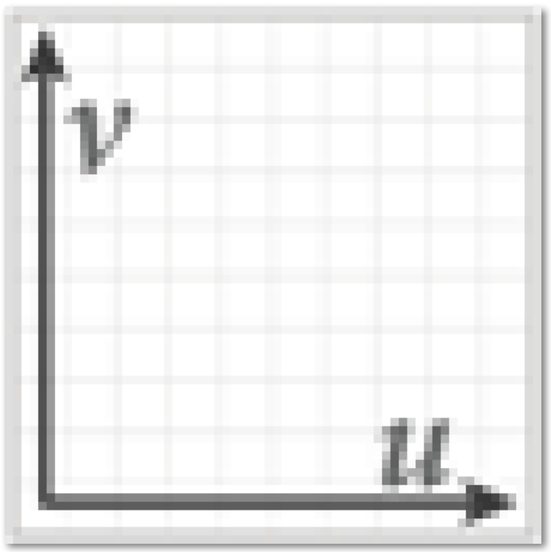


Texture magnification

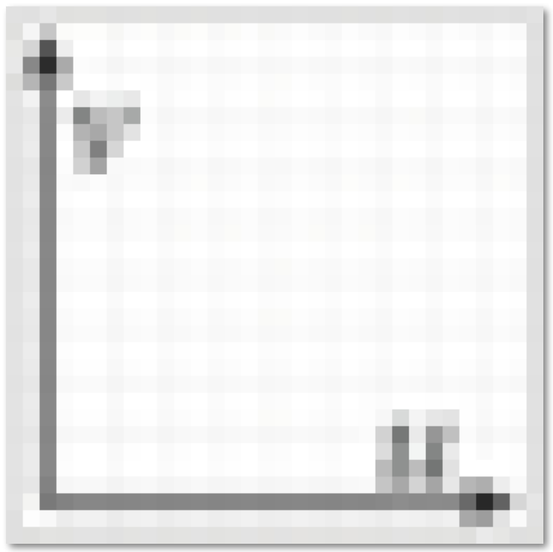
Mipmap (L. Williams 83)



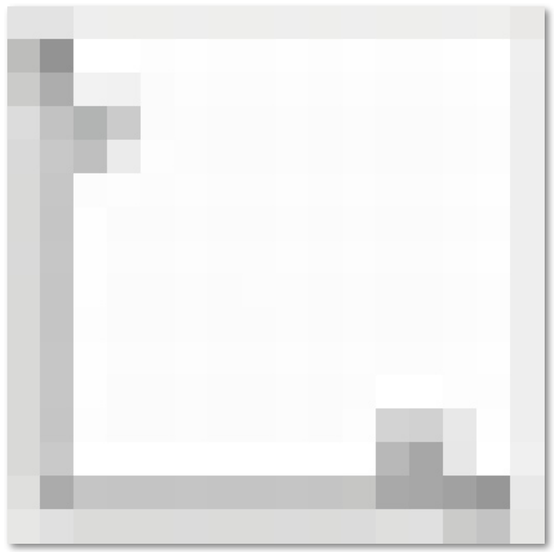
Level 0 = 128x128



Level 1 = 64x64

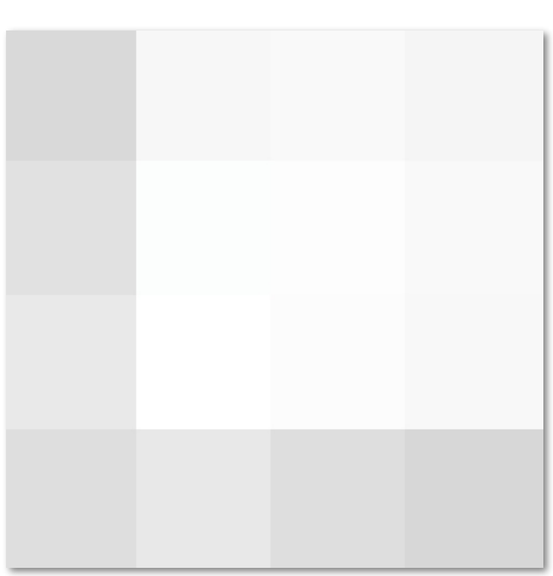


Level 2 = 32x32

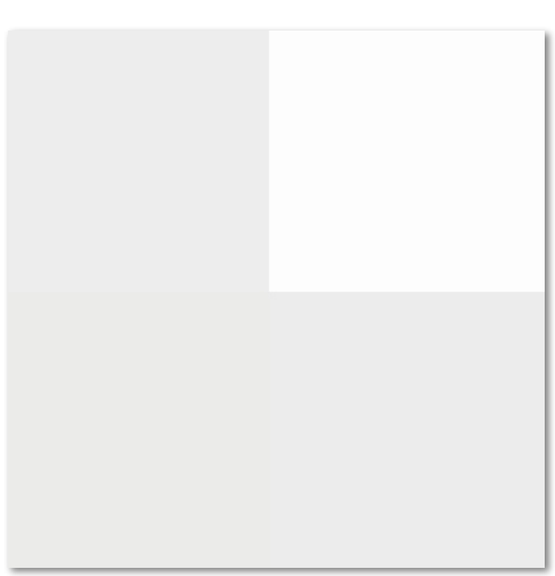


Level 3 = 16x16

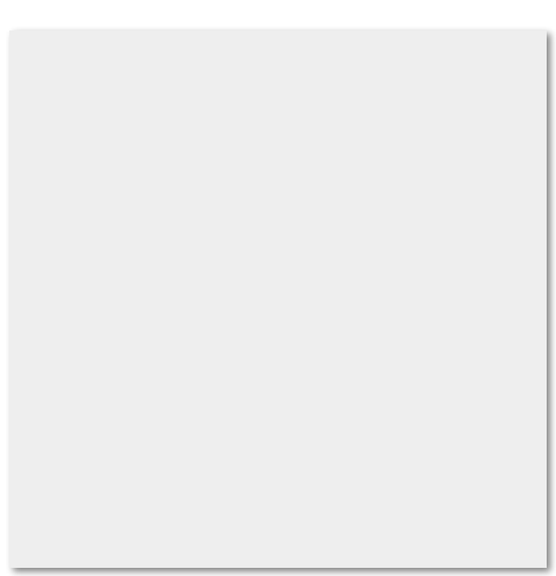
Level 4 = 8x8



Level 5 = 4x4



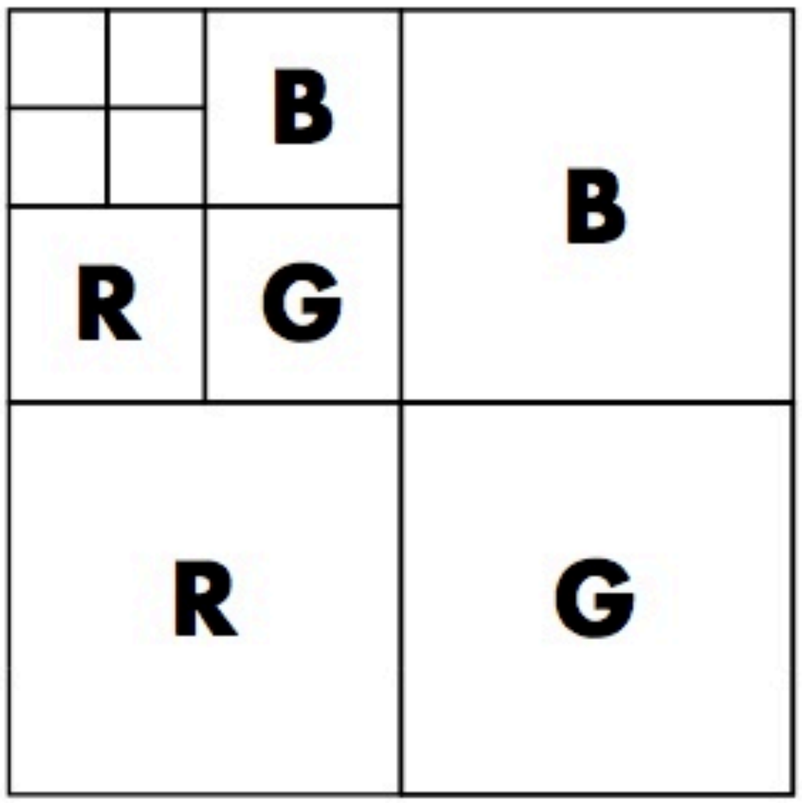
Level 6 = 2x2



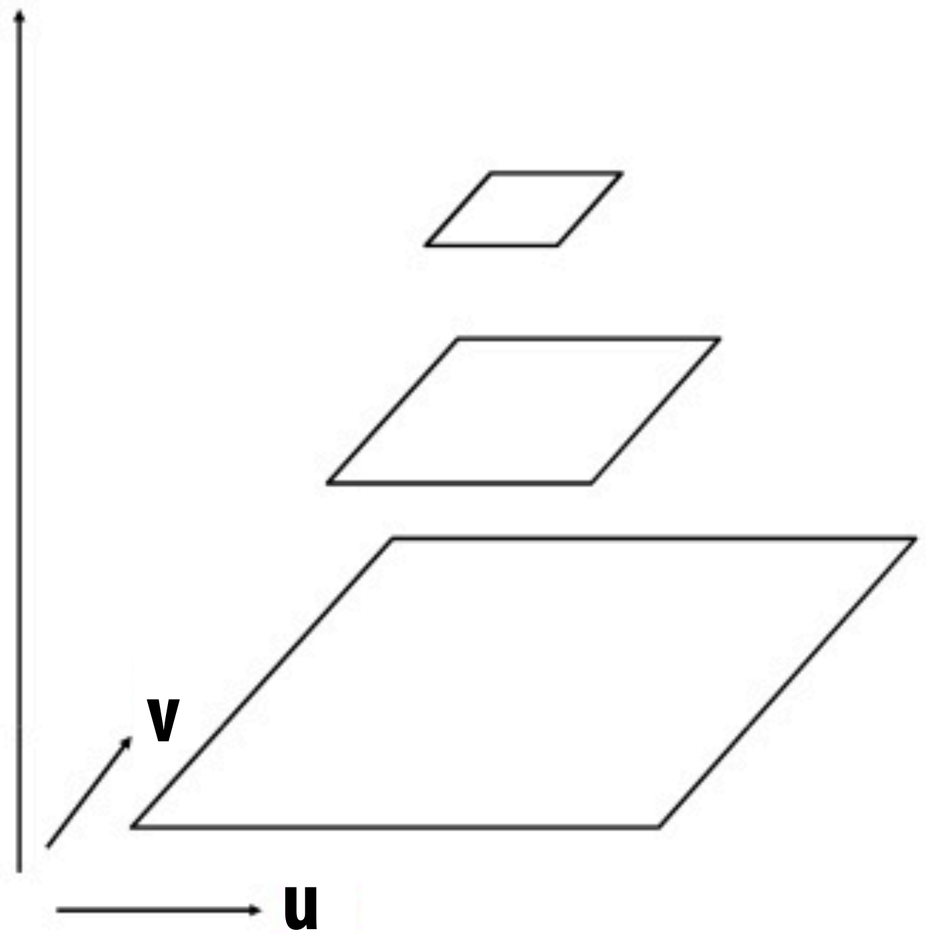
Level 7 = 1x1

Idea: prefilter texture data to removal high frequencies
(texels at higher levels store integral of the texture function over a region of texture space)

Mipmap (L. Williams 83)

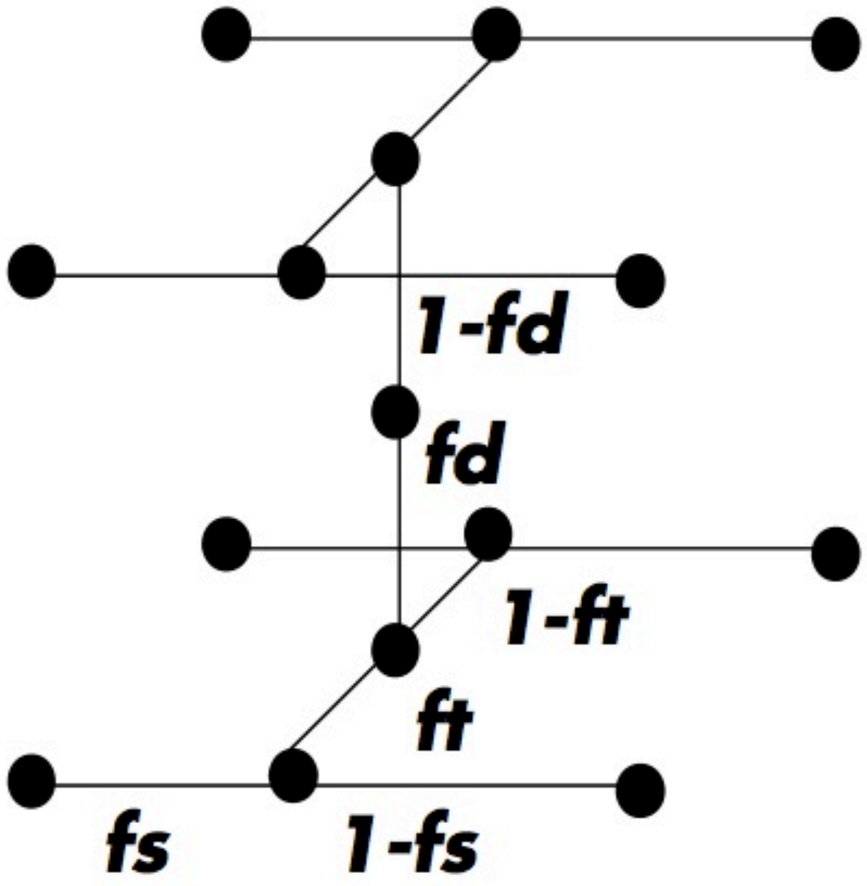


Williams' original proposed mip-map layout



"Mip hierarchy" level = d

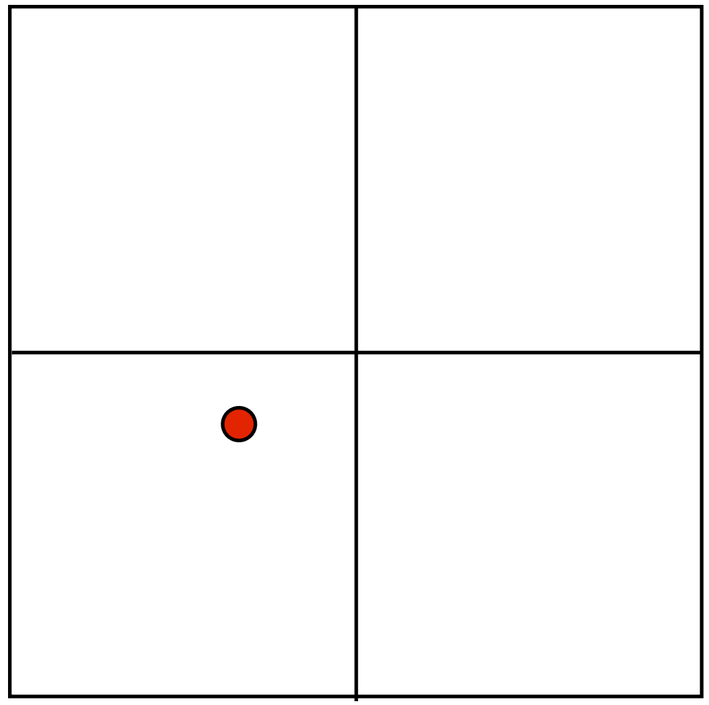
Constant-time filtering



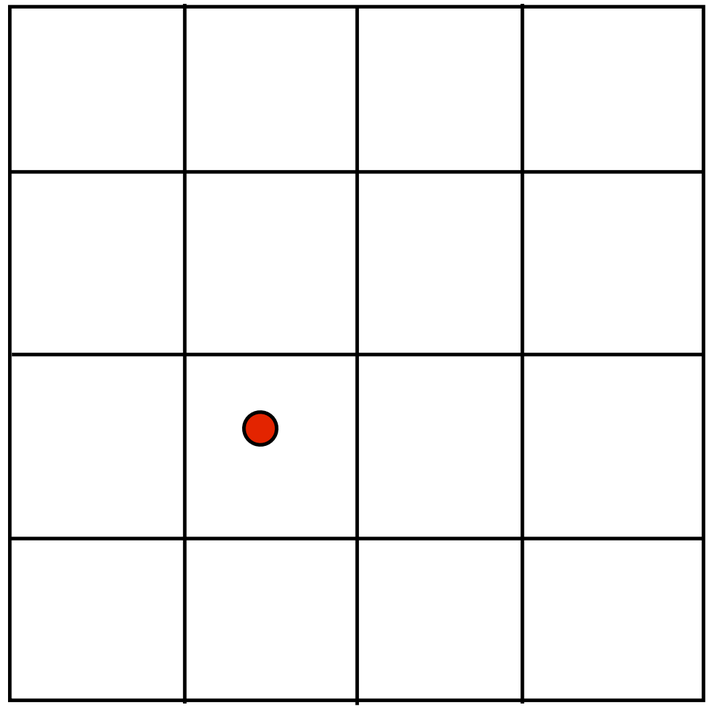
$$lerp(t, v_1, v_2) = v_1 + t(v_2 - v_1)$$

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

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



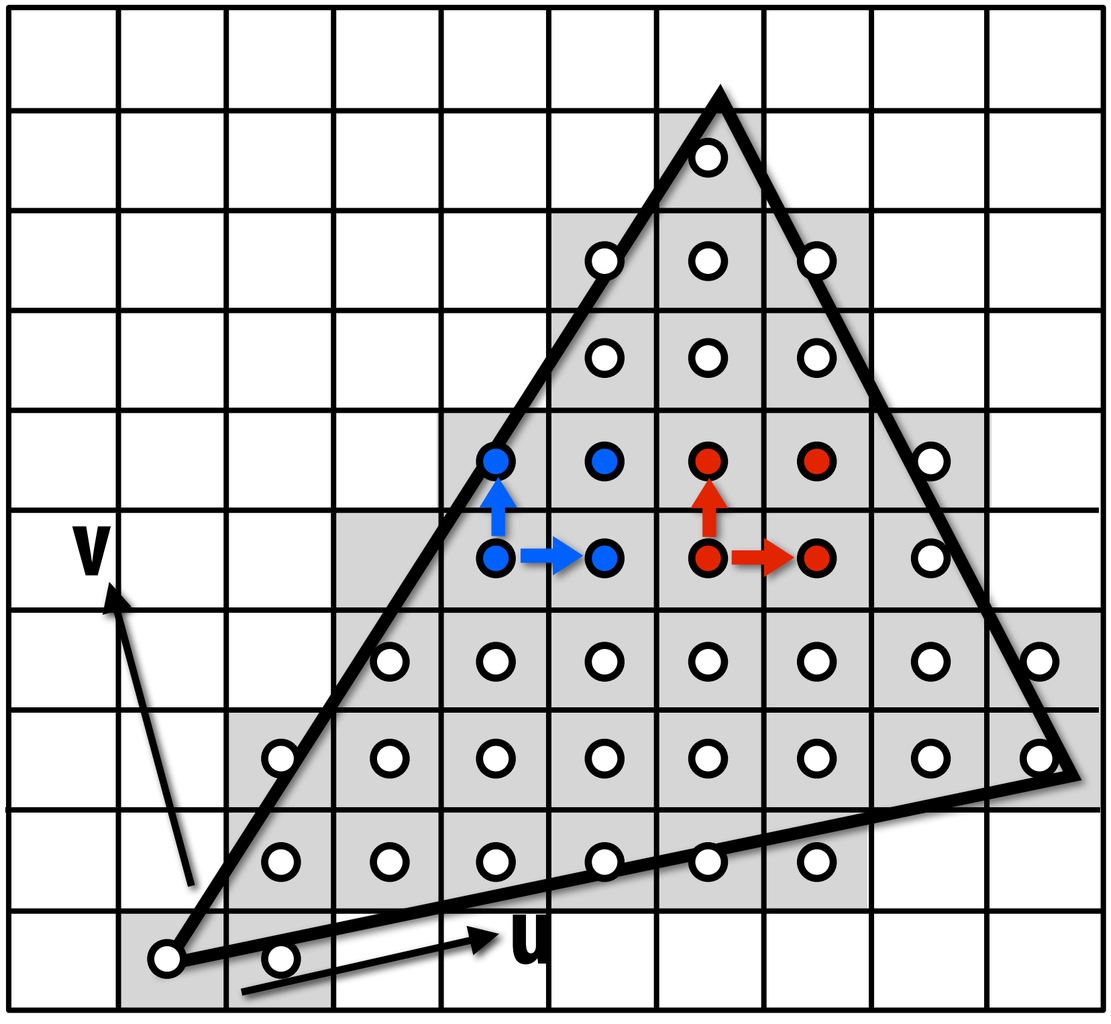
mip-map level d+1 texels



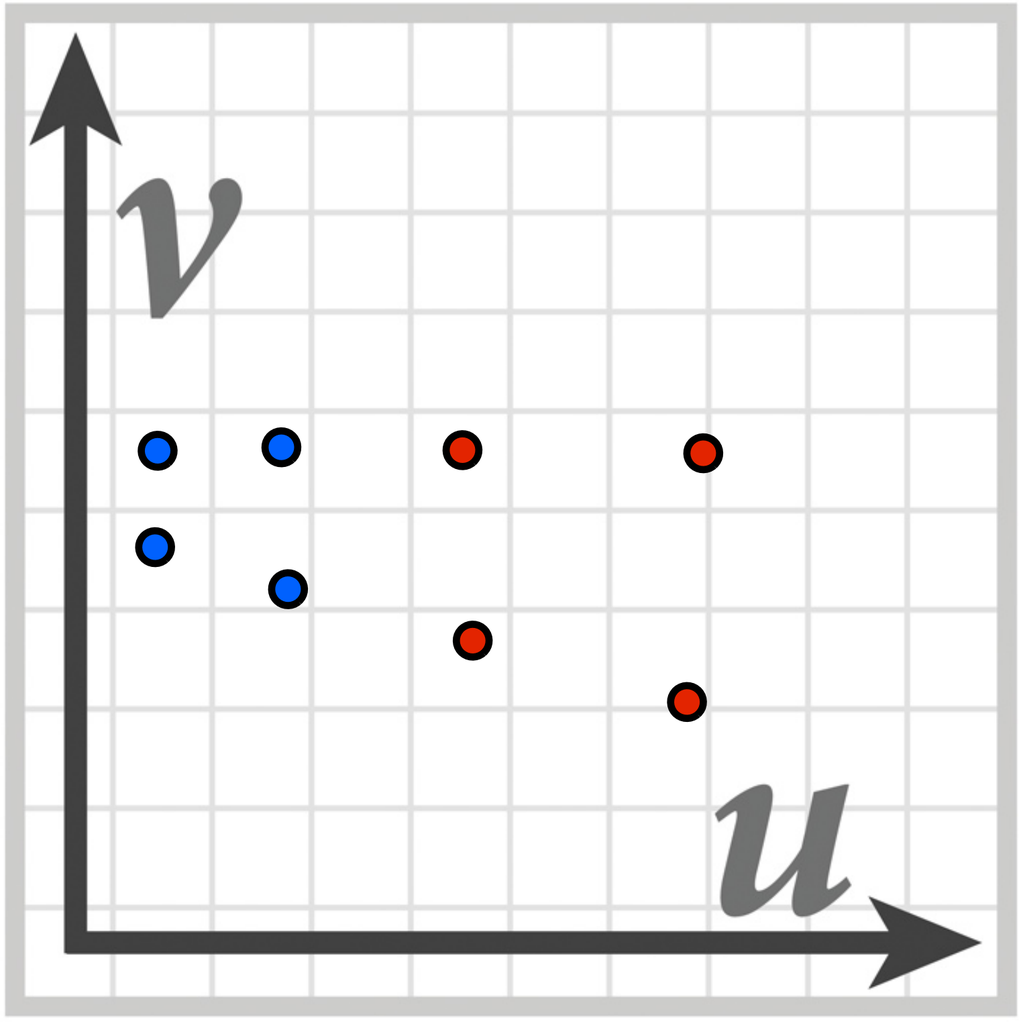
mip-map level d texels

Computing d

Take differences between texture coordinate values for neighboring fragments



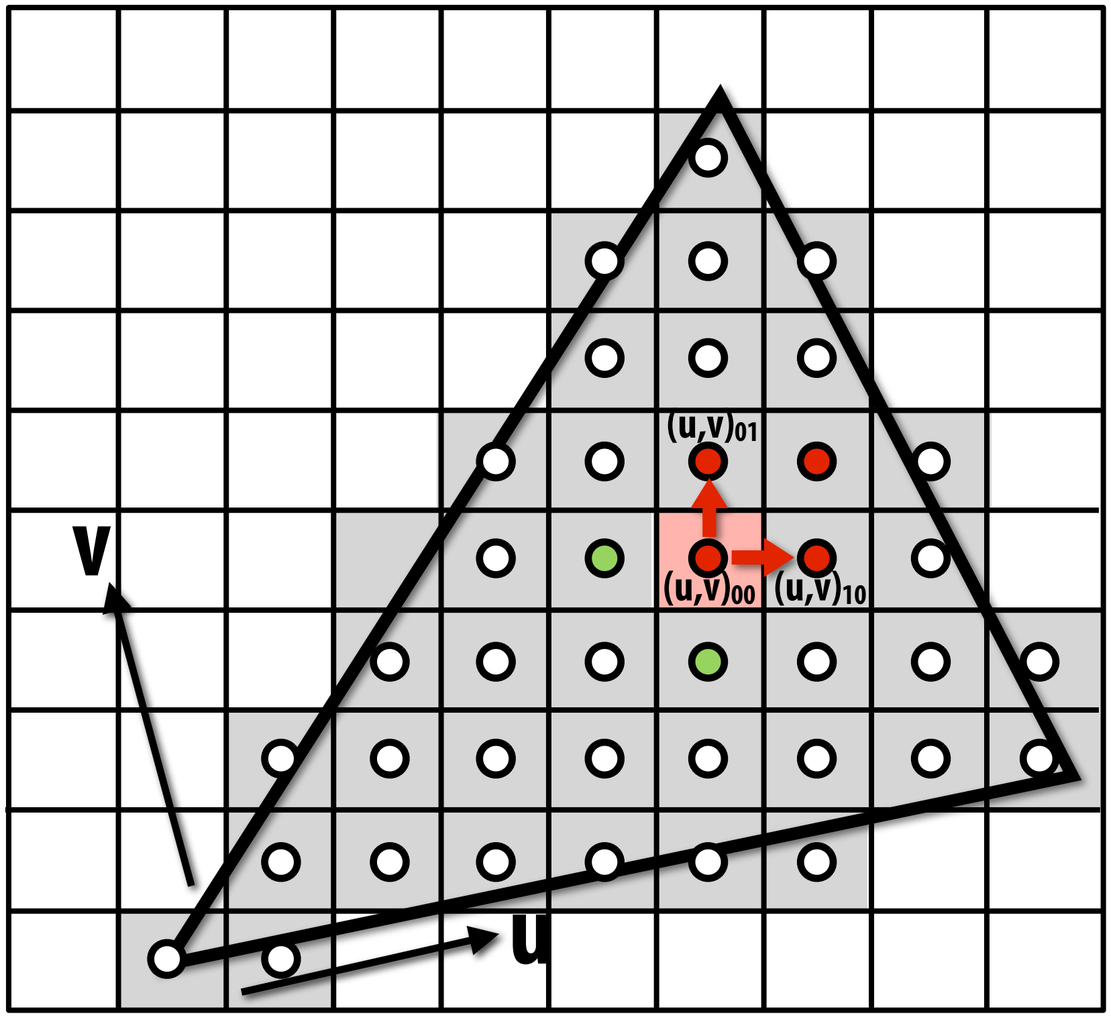
Screen space



Texture space

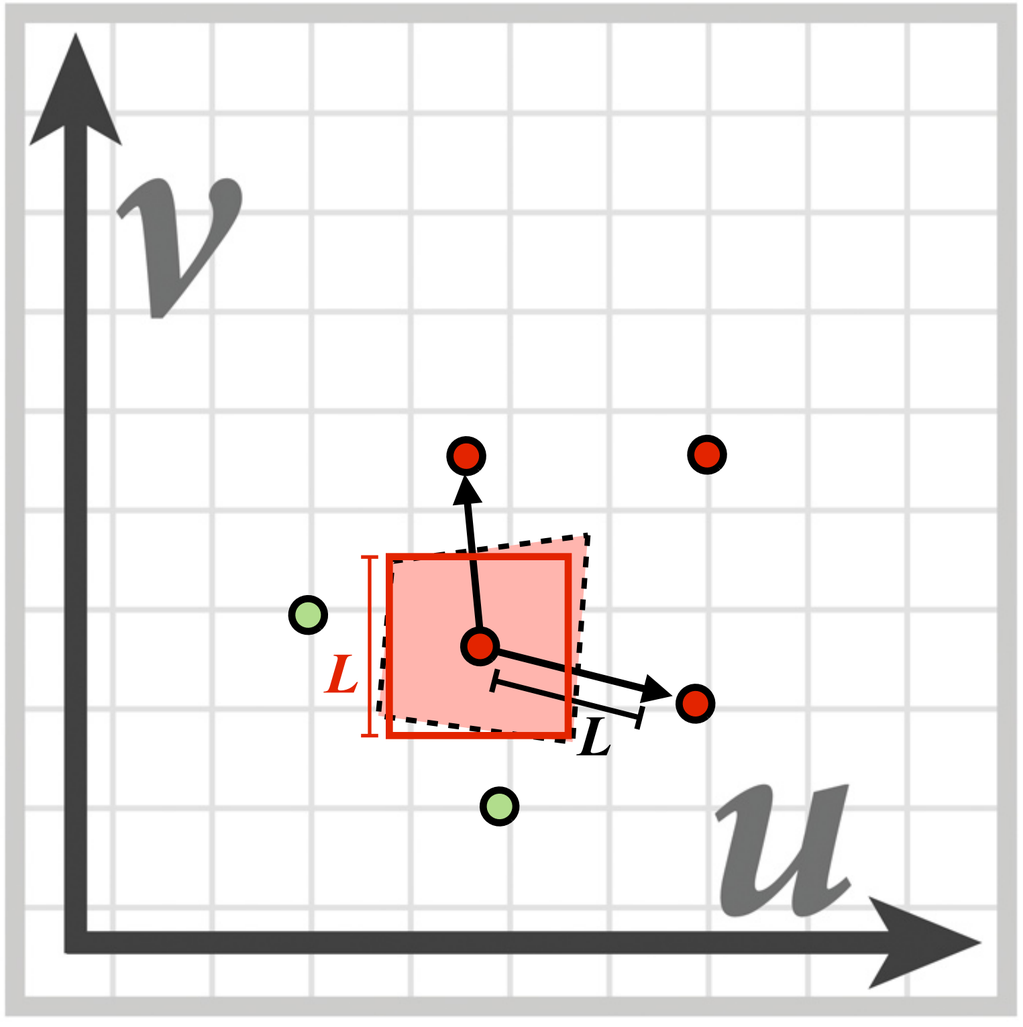
Computing d

Take differences between texture coordinate values of neighboring fragments



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

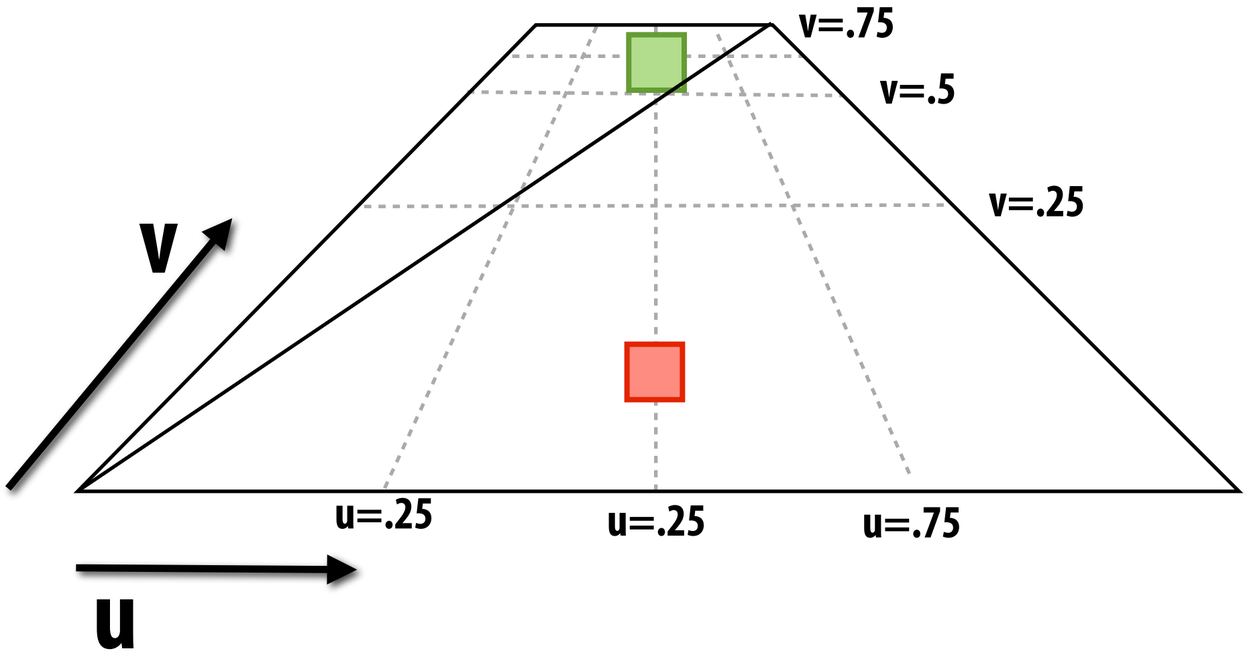
$du/dy = u_{01}-u_{00}$ $dv/dy = v_{01}-v_{00}$

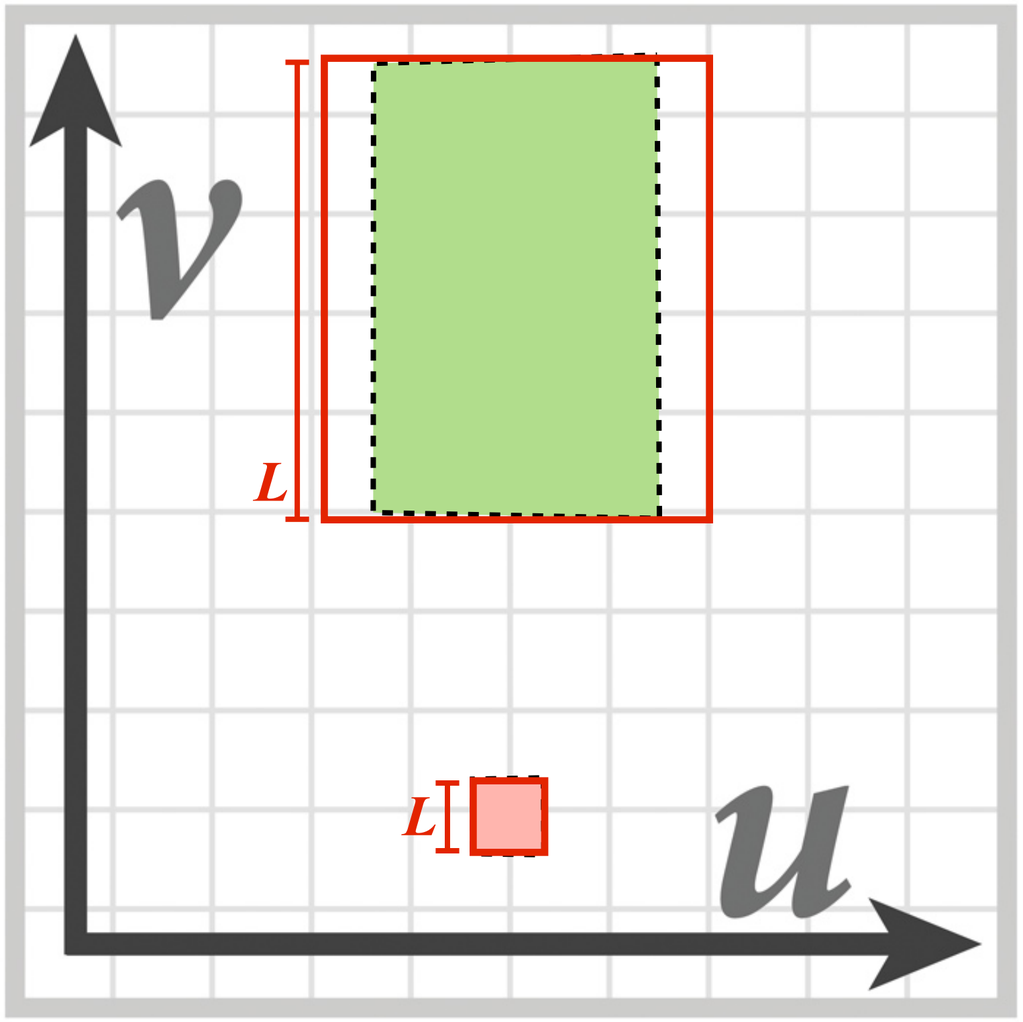


$$L = \max\left(\sqrt{\left(\frac{du}{dx}\right)^2+\left(\frac{dv}{dx}\right)^2}, \sqrt{\left(\frac{du}{dy}\right)^2+\left(\frac{dv}{dy}\right)^2}\right)$$

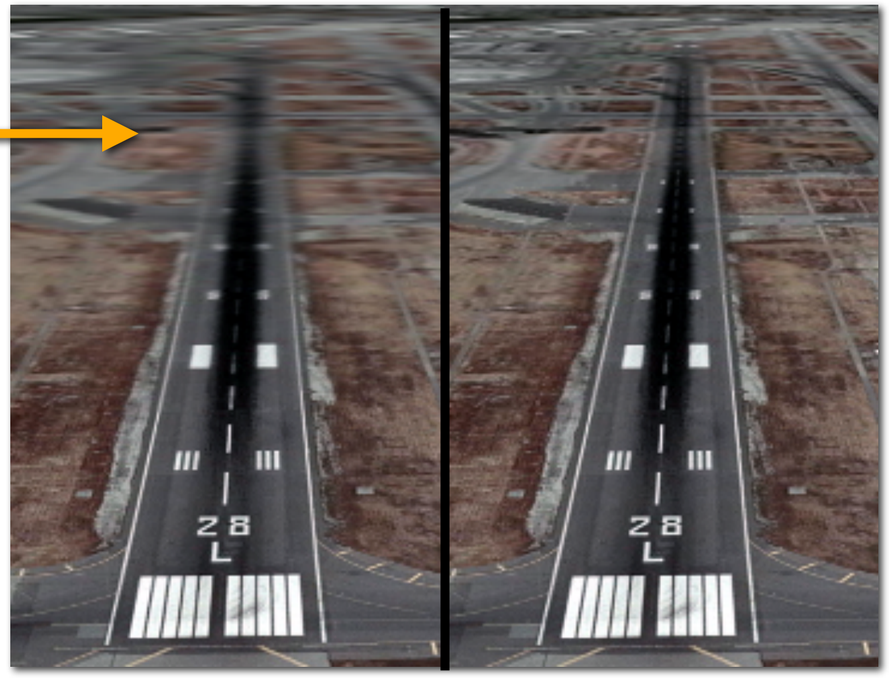
$$mip\text{-}map\ d = \log_2(L)$$

Pixel area may not map to isotropic region in texture space





Notice overblurring in u direction



Trilinear (Isotropic) Filtering Anisotropic Filtering

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

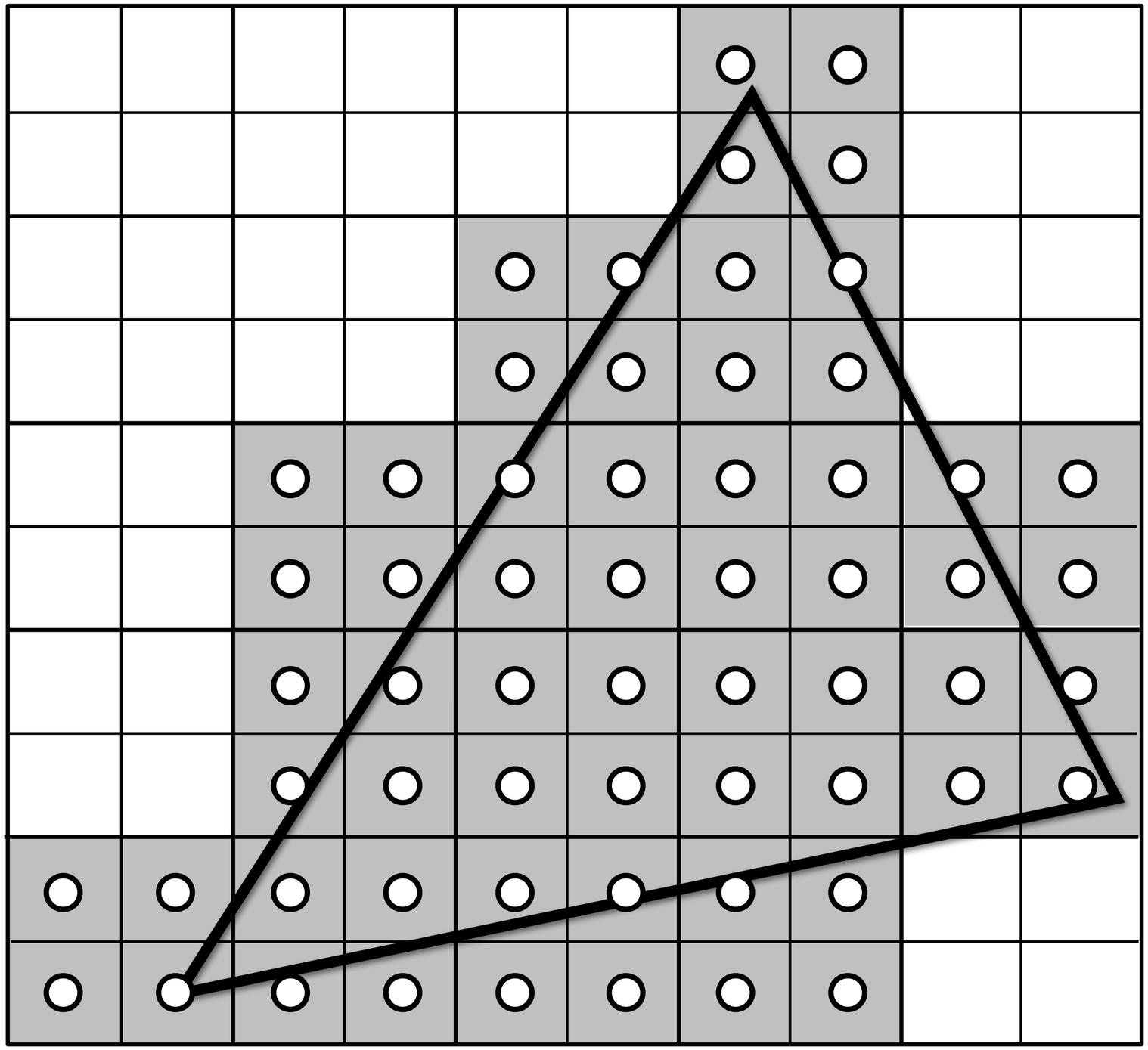
$$mip\text{-}map\ d = \log_2(L)$$

GPUs shade quad fragments

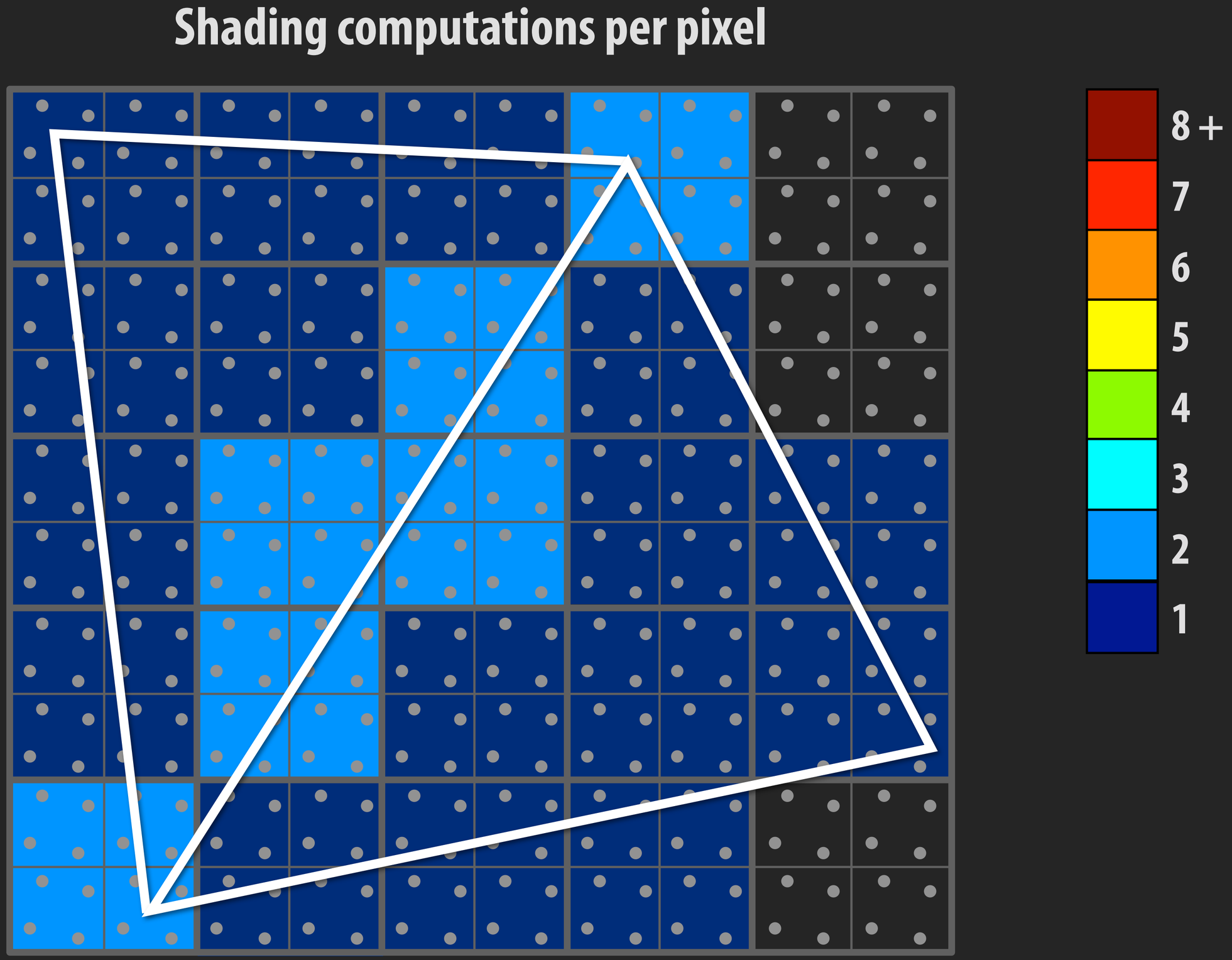
(2x2 fragment block is the minimum granularity of rasterization output and shading)

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)



Multiple fragments shaded for pixels near triangle boundaries



Small triangles result in extra shading

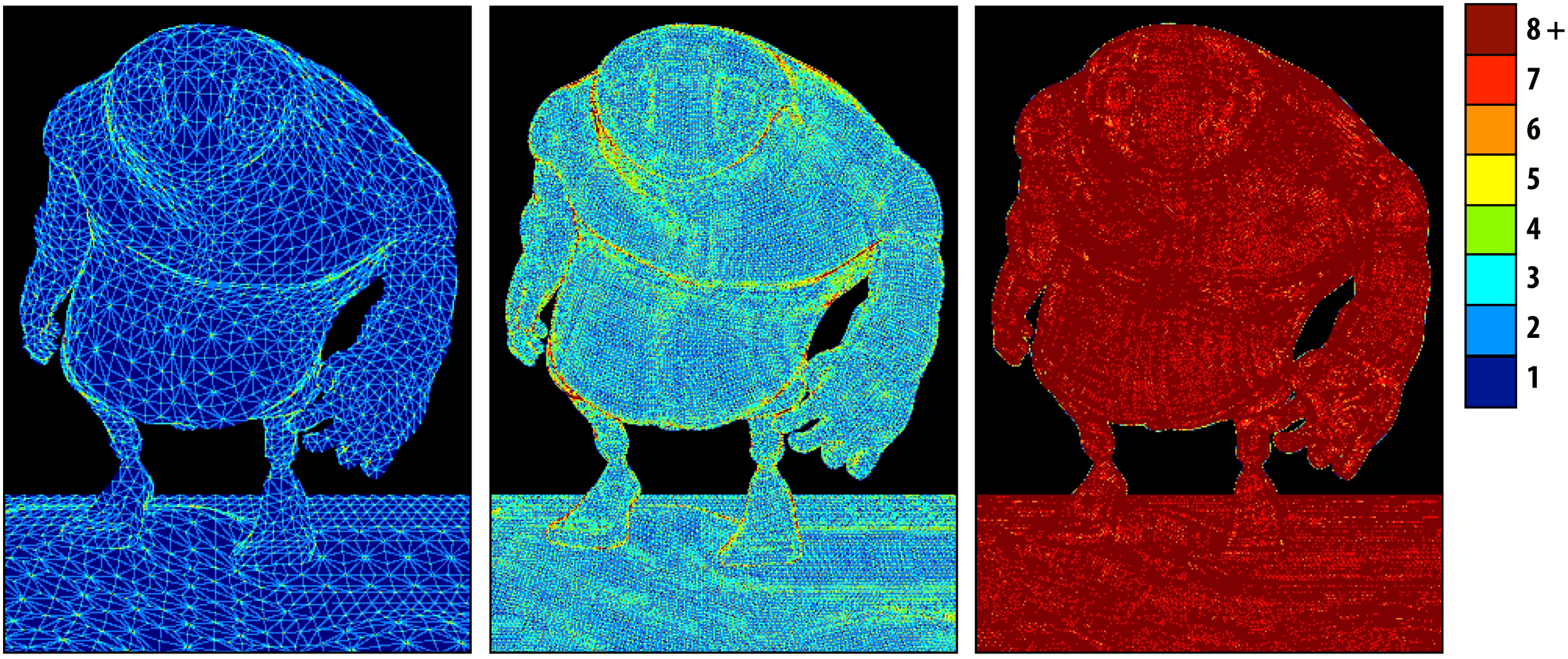
Shaded quad-fragments per pixel

(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

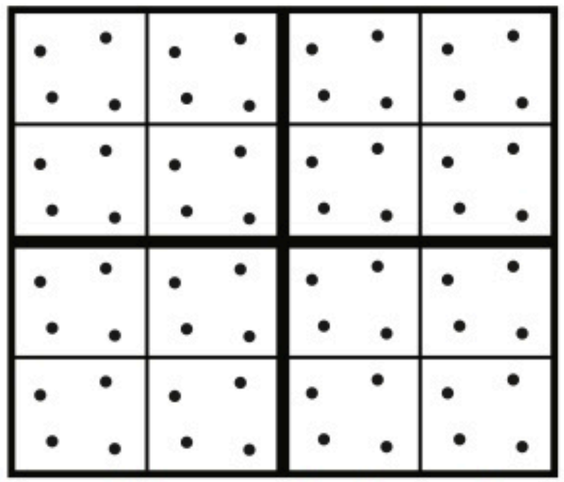
1 pixel-area triangles



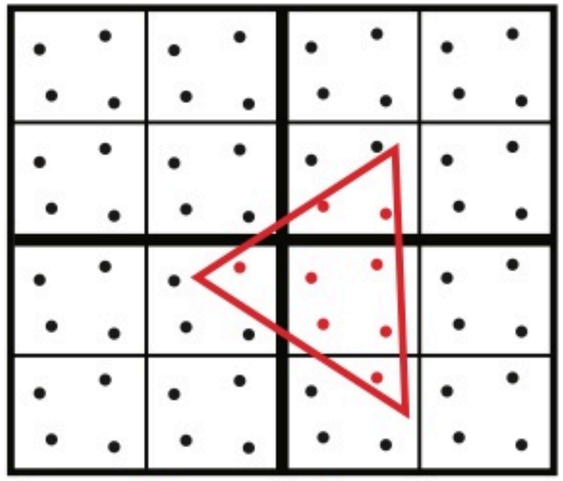
Want to sample appearance approximately once per surface per pixel (assuming correct texture filtering)

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

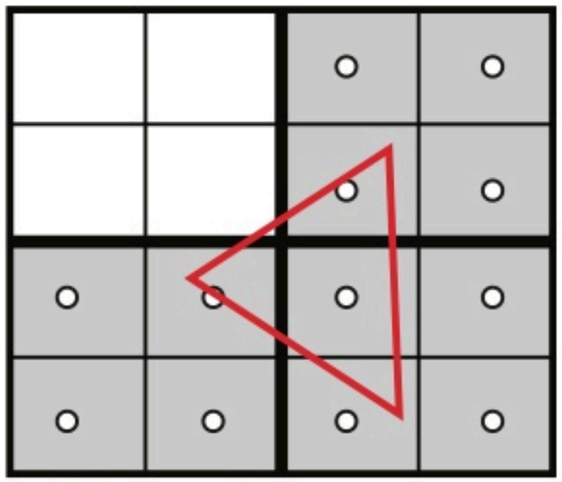
Multi-sample anti-aliasing (MSAA)



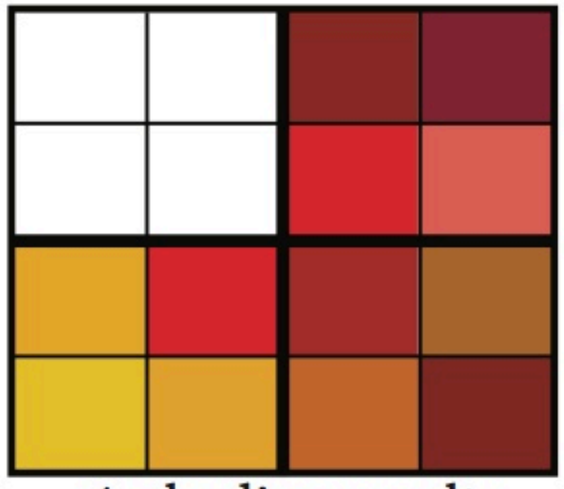
1. multi-sample locations



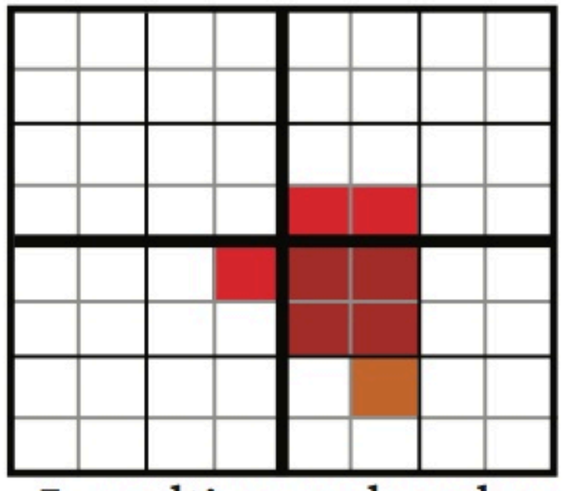
2. multi-sample coverage



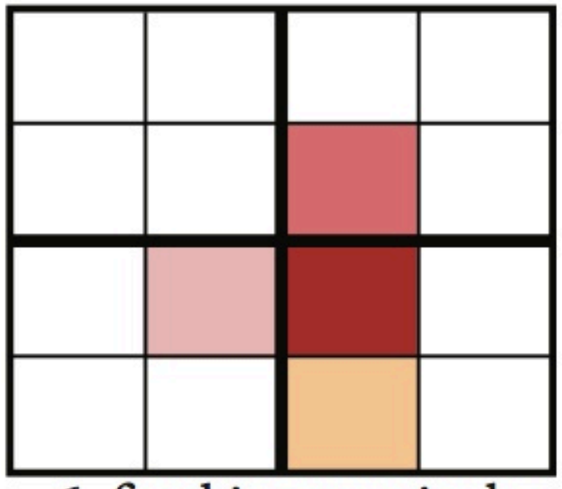
3. quad fragments



4. shading results



5. multi-sample color



6. final image pixels

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**
  - Mipmap is 4/3 the size of original texture image
- **For each texture filtering request**
  - Constant filtering cost (independent of d)
  - Constant number of texels accessed (independent of d)
- **Bilinear/trilinear filtering is isotropic: must "overblur" to avoid aliasing**
  - Anisotropic texture filtering provides higher image quality, but also higher compute and memory bandwidth cost

Summary: a texture fetch

For each texture fetch in a shader program:

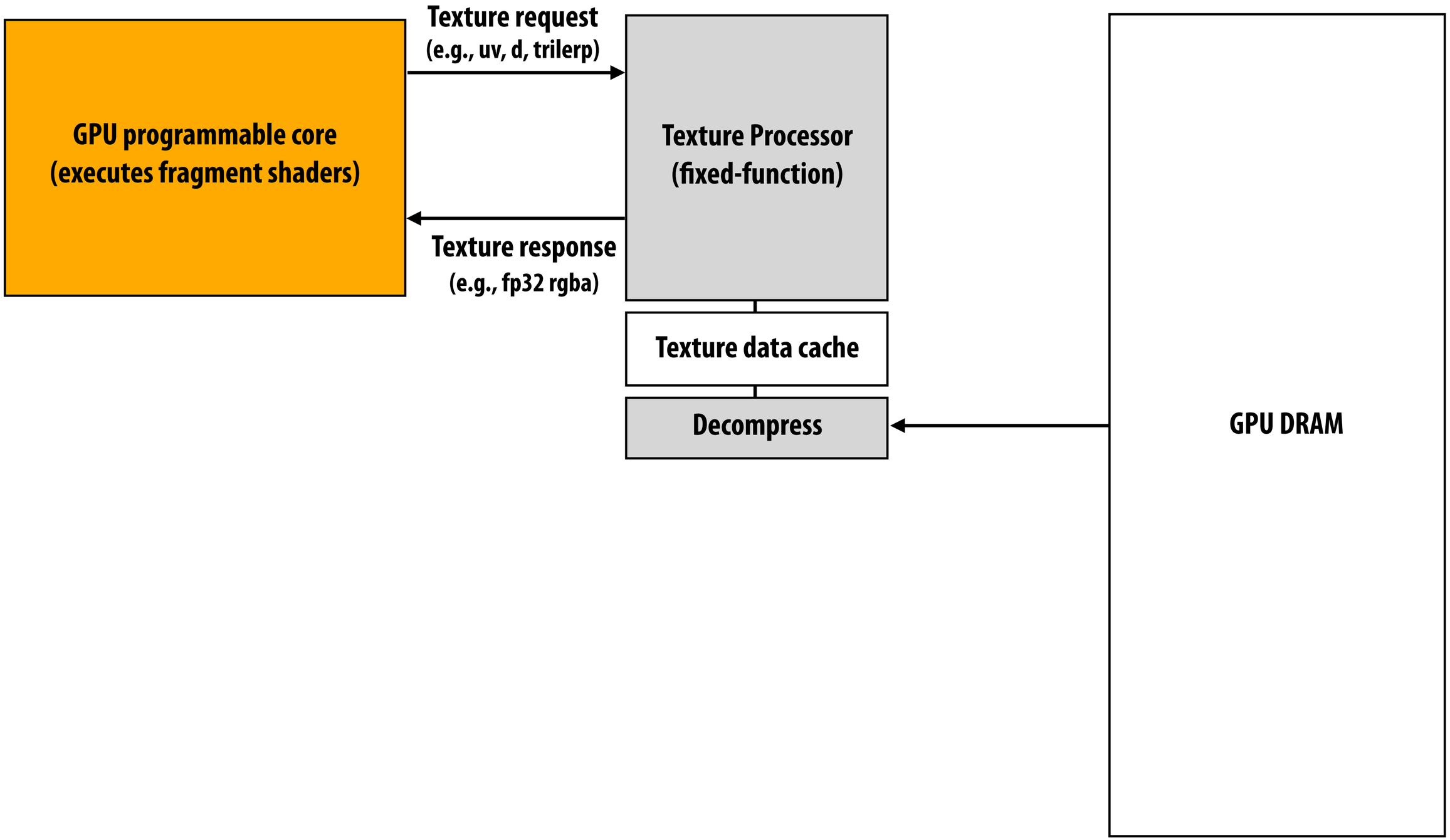
1. Compute du/dx, du/dy, dv/dx, dv/dy differentials from quad fragment
2. Compute d
3. Convert normalized texture coordinate uv to texel coordinates tu,tv
4. Compute required texels **
5. Load texture data in filter footprint (need eight texels for trilinear) ****
6. Result = perform tri-linear interpolation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

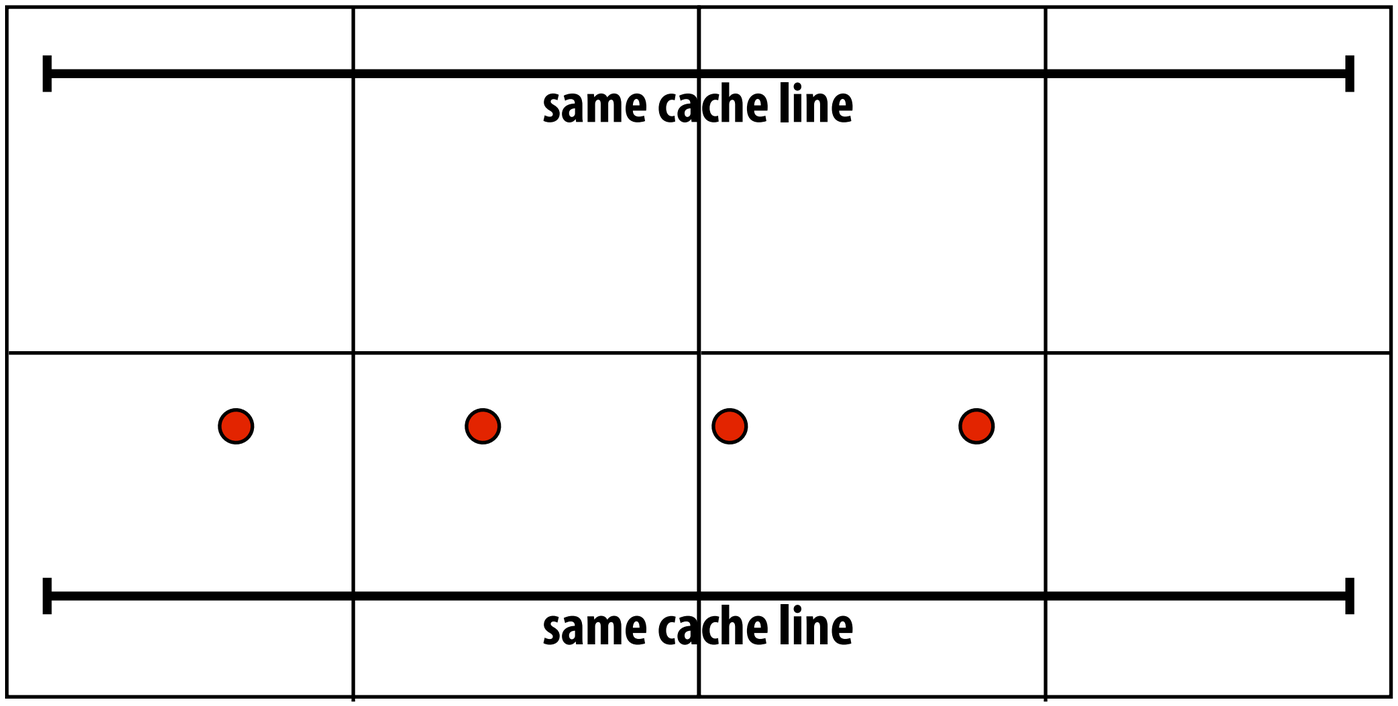
- **Texture data footprint**
  - Modern games: large textures: 10s-100s of MB
  - Film rendering: GBs to TBs of textures in scene DB
- **Texture bandwidth**
  - 8 texels per tri-linear fetch (assume 4 bytes/texel)
  - Modern GPU: billions of fragments/sec

    (NVIDIA GTX 580: ~40 billion/sec)
- **Performant graphics systems need:**
  - Texture caching
  - Texture compression
  - Latency hiding solution

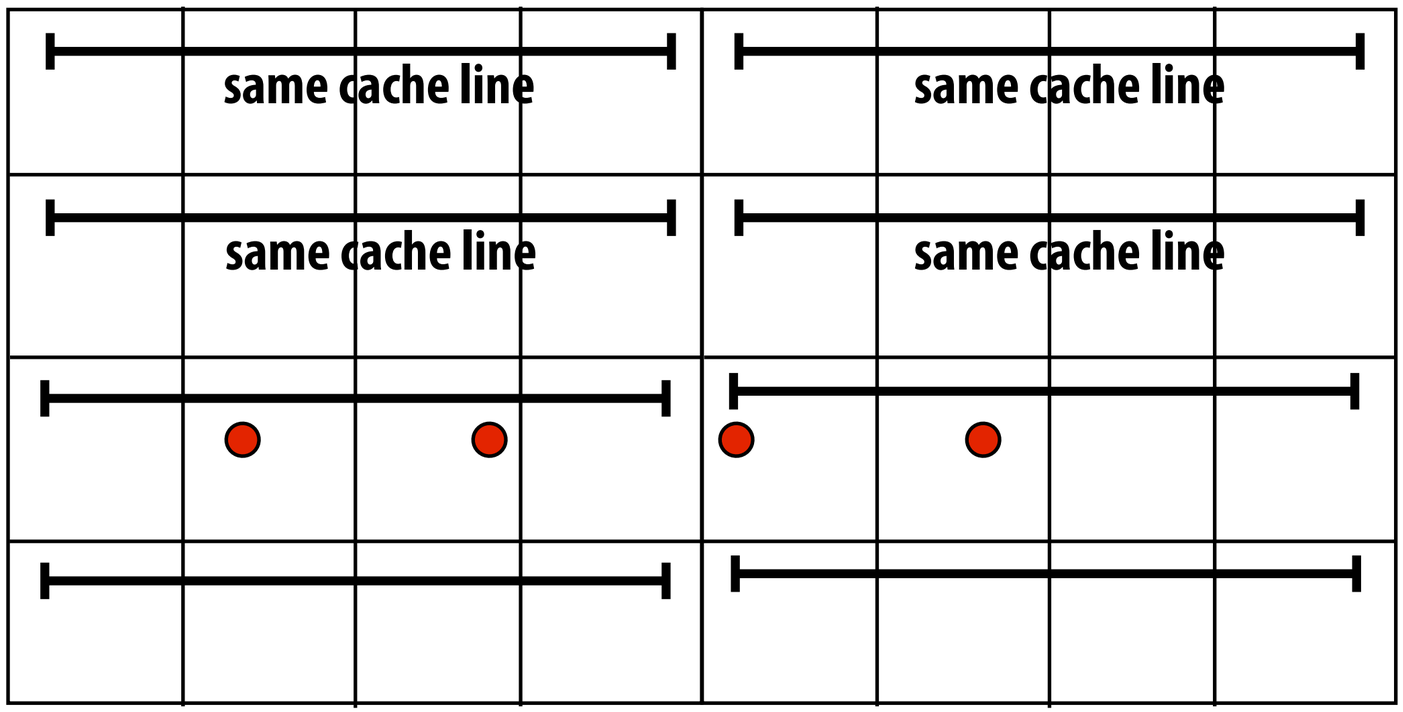
Review: the role of caches in CPUs

- 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 bandwidth amplifiers (data path between cache and ALUs is usually wider than that to DRAM)
- Convert fine-grained 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



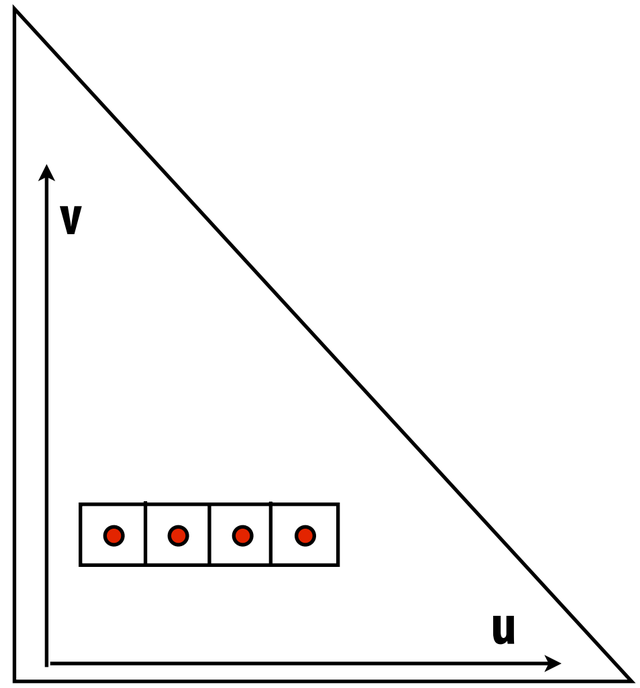
mip-map level *d* texels

Assume:

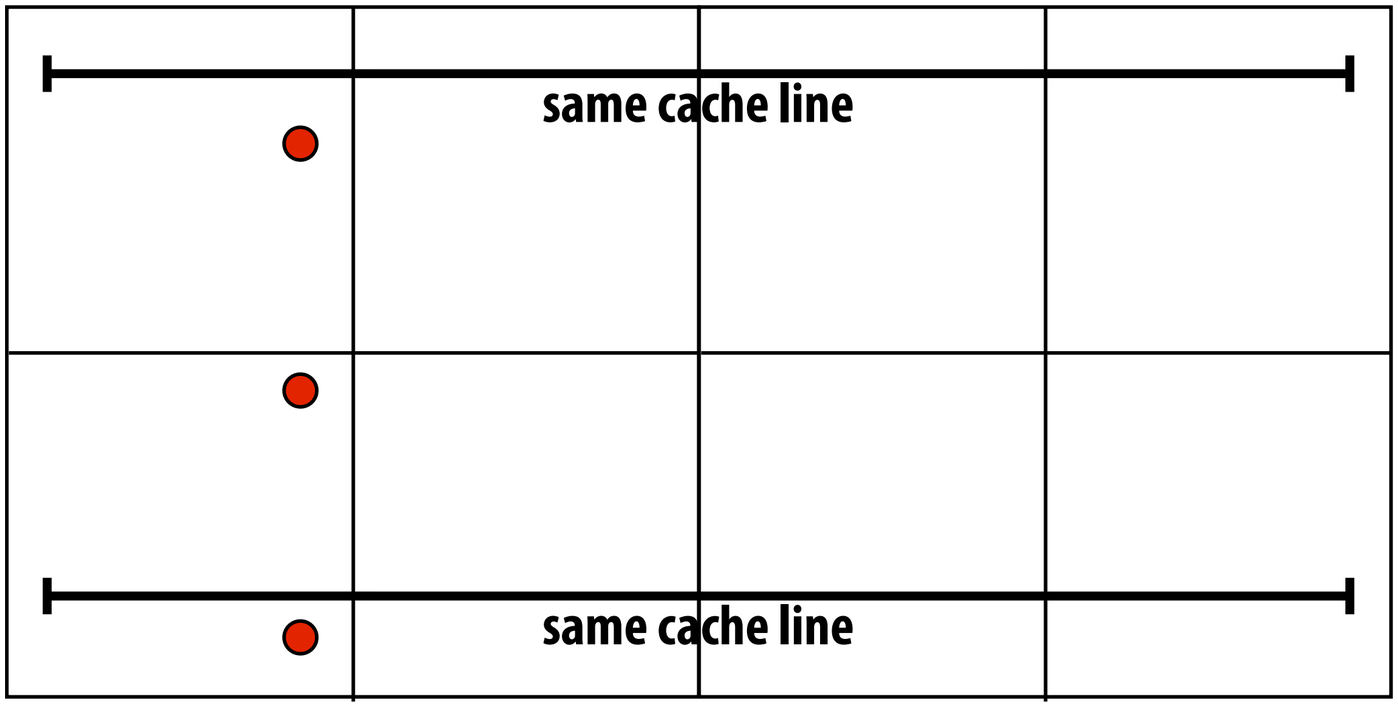
Row-major raster order

Horizontal texels contiguous in memory

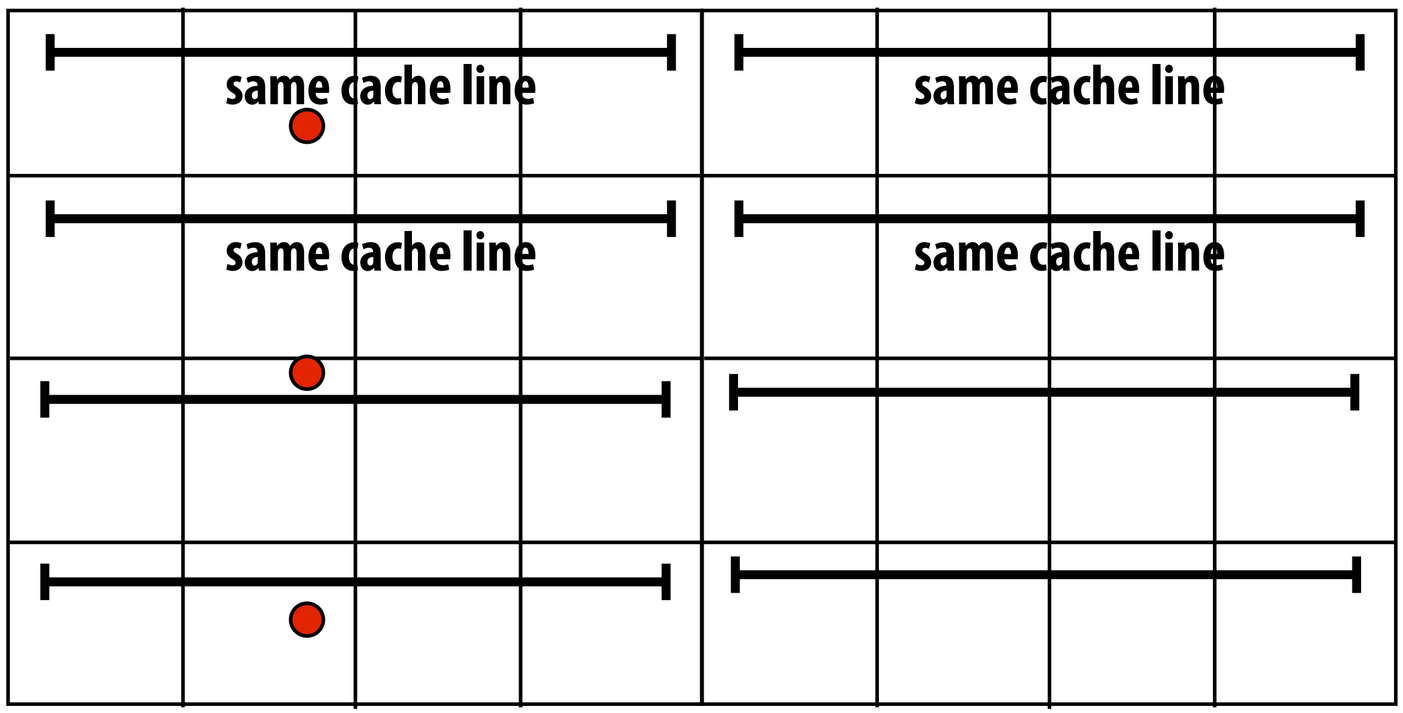
Texture cache line = 4 texels



Now rotate triangle on screen



mip-map level *d+1* texels



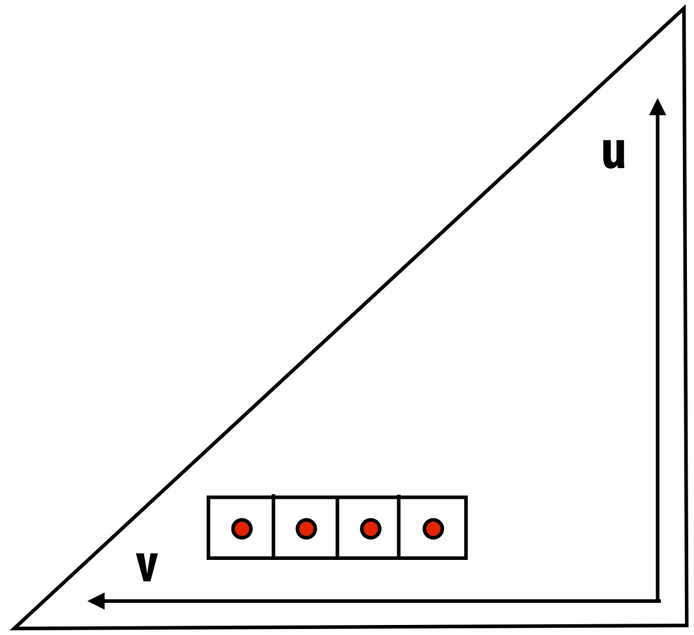
mip-map level *d* texels

Assume:

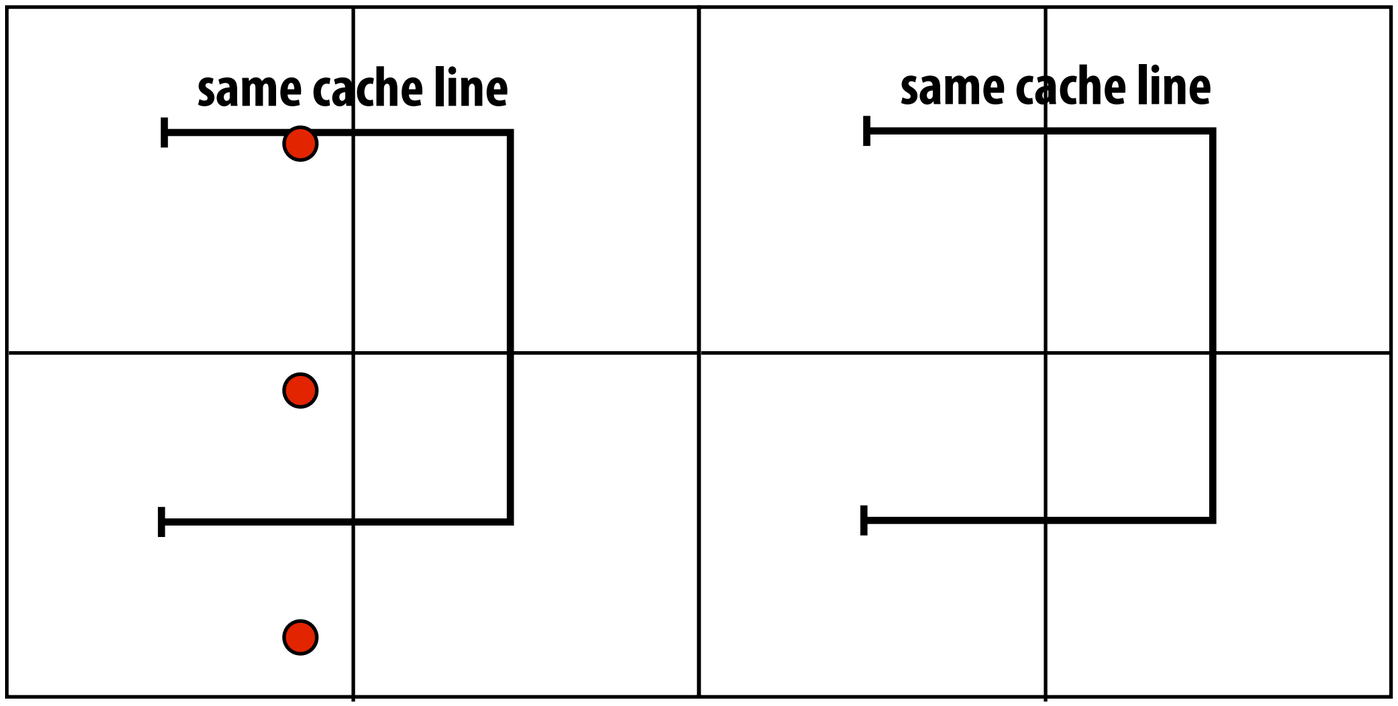
Row-major raster order

Horizontal texels contiguous in memory

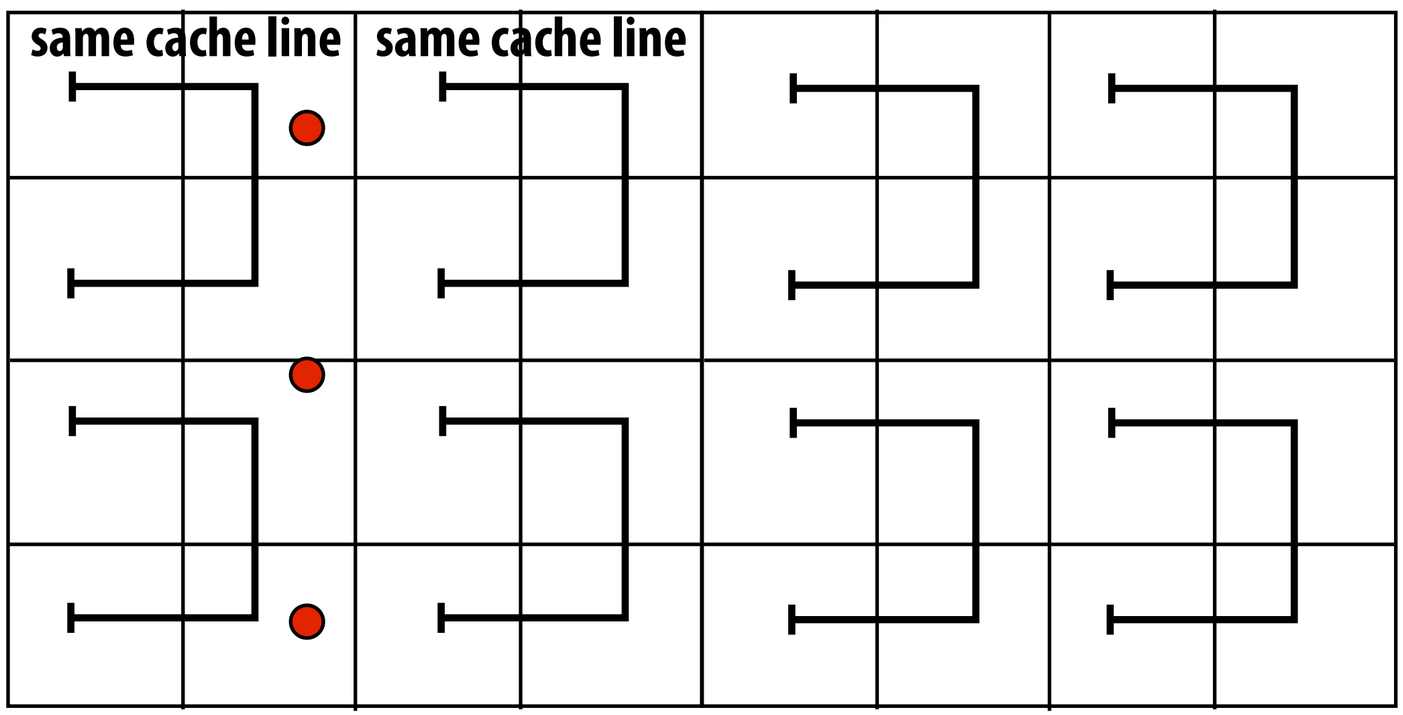
Cache line = 4 texels



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



mip-map level *d+1* texels



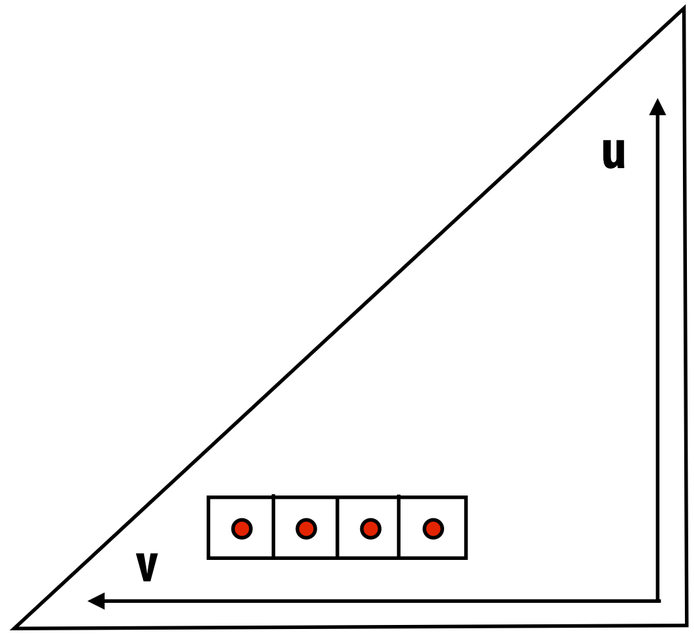
mip-map level *d* texels

Assume:

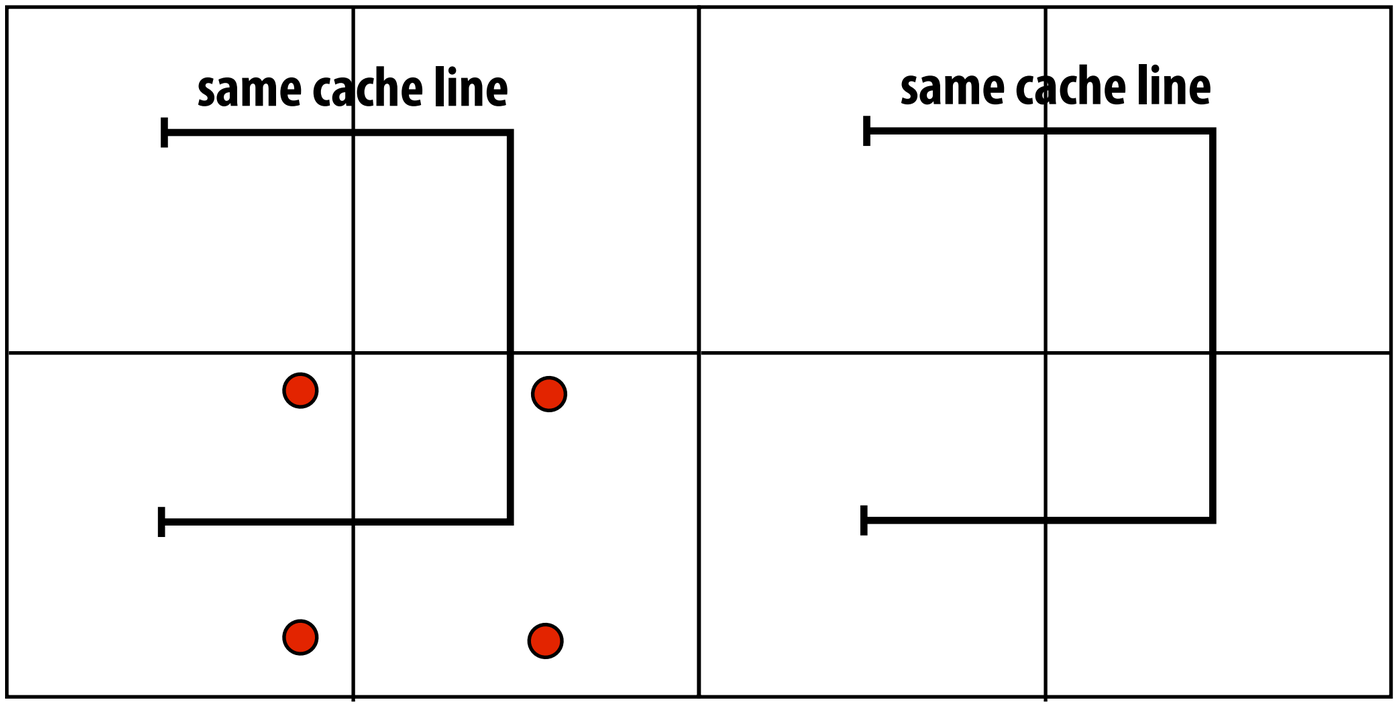
Row-major raster order

2D blocks of texels contiguous in memory

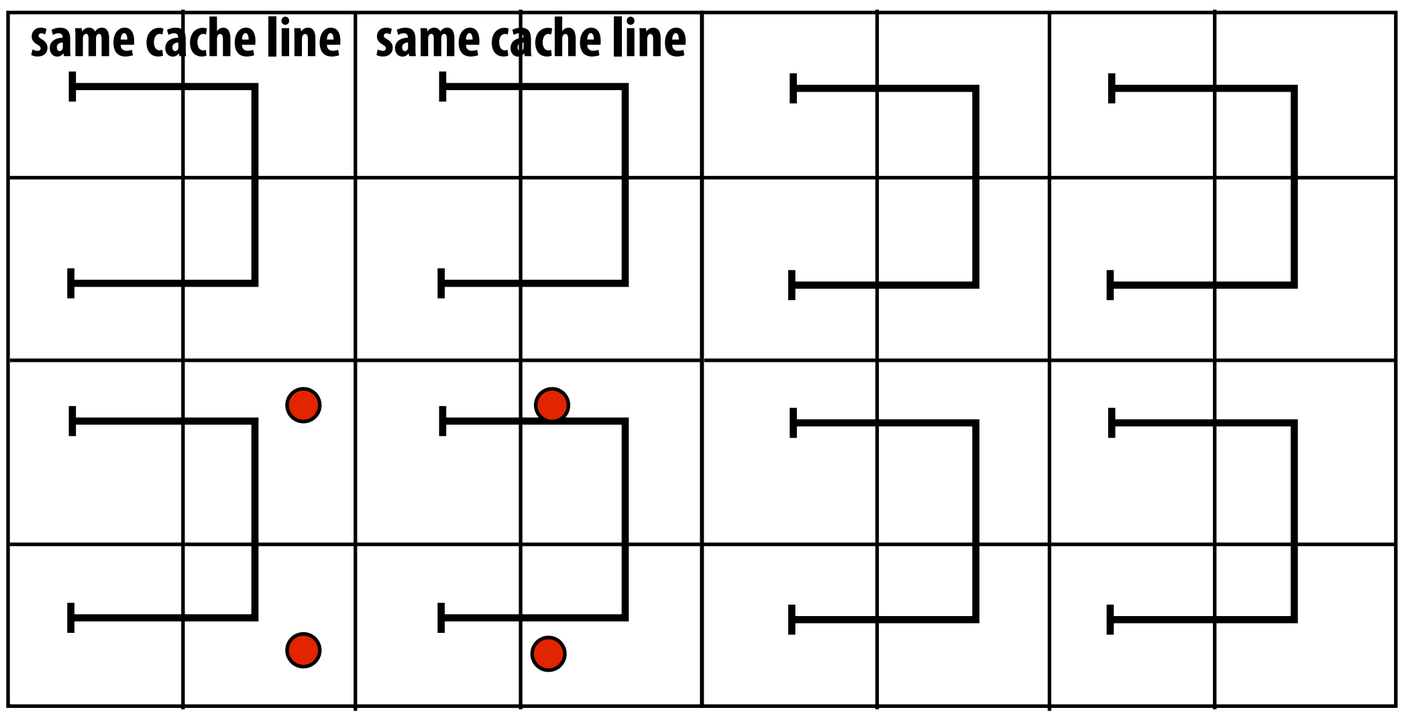
Cache line = 4 texels



Tiled rasterization increases reuse



mip-map level *d+1* texels



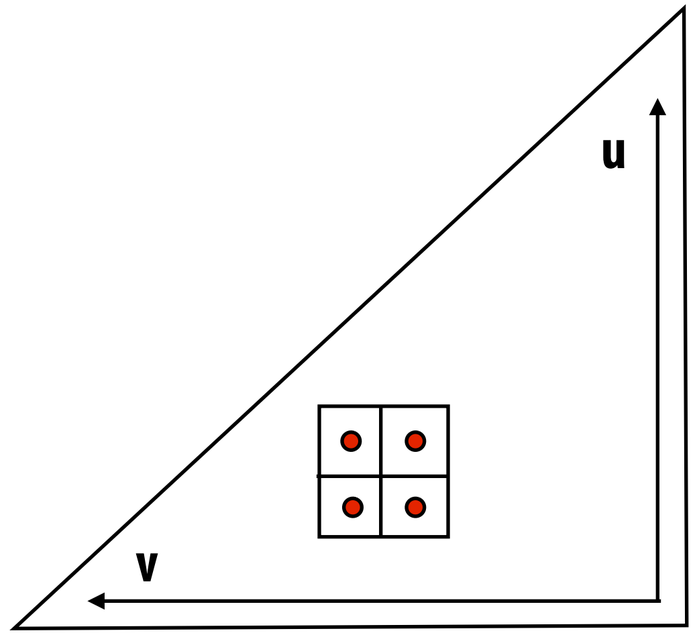
mip-map level *d* texels

Assume:

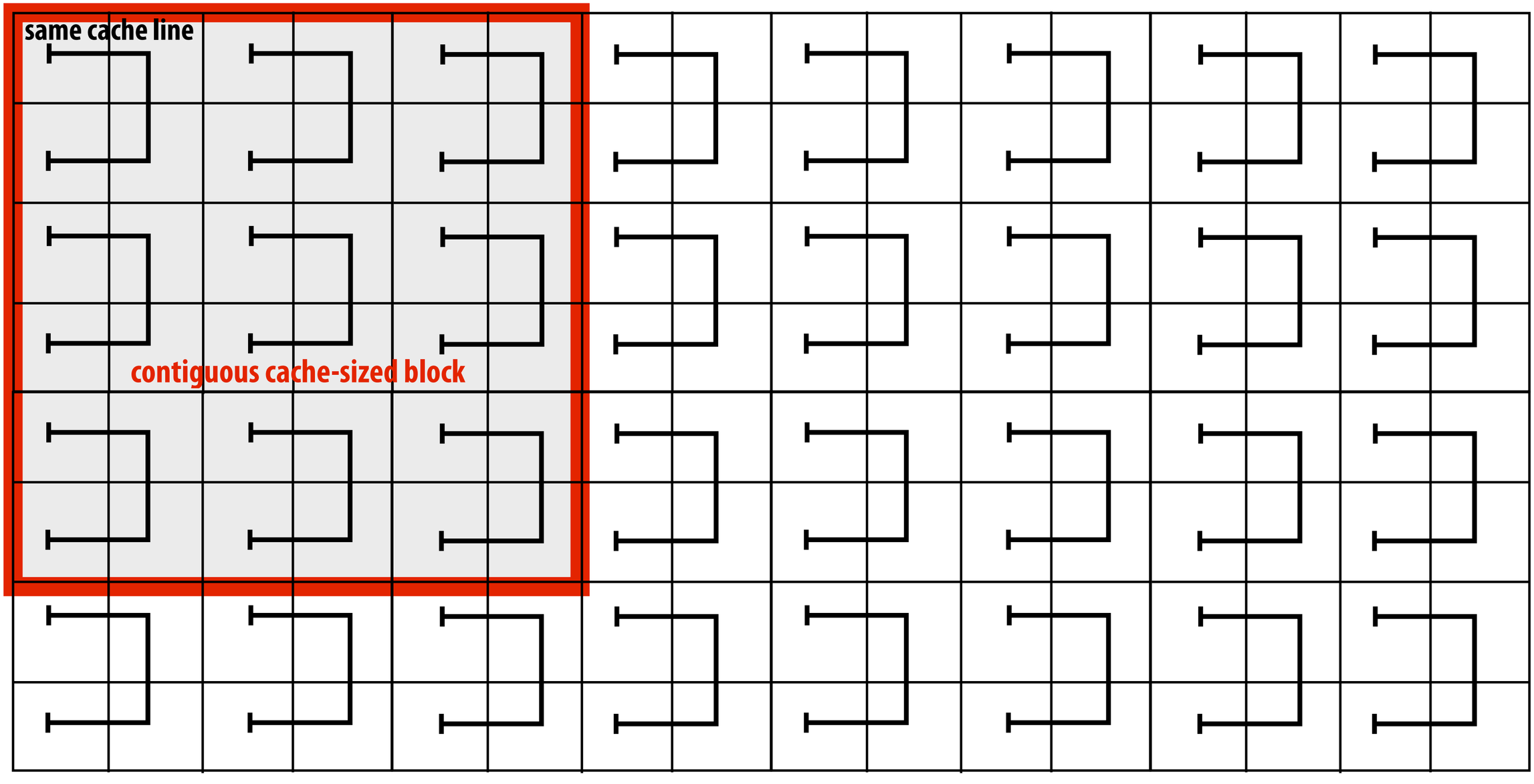
Blocked raster order

2D blocks of texels contiguous in memory

Cache line = 4 texels



6D blocking further reduces conflicts



Blocked texture formats

- **Render-to-texture challenge:**
  - Frame-buffer has a preferred memory layout
  - Texture buffers has a preferred memory layout
  - Costly to convert buffers between formats when render target is subsequently bound as a texture for a later rendering pass

- **Modern graphics APIs:**
  - Declare usage for buffers at allocation in API
  - In general, standard blocking schemes across the board

What type of data reuse does a texture cache capture?

- **Spatial locality, not temporal locality**
  - **Many of the same texels are accessed by texture fetches from adjacent fragments (due to overlap of filter support regions)**
  - **There is essentially no temporal locality within a fragment shader (little reason for a shader does to access the same part of the texture map twice)**

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 assuming trilinear filtering?
  - How can incorrect computation of texture miplevel (d) affect this?

- **In reality, caching behavior is good, but not CPU workload good**
  - [Montrym & Moreton 95] design for 90% hits
  - Why? only so much spatial locality
  - Graphics pipeline requires high memory bandwidth for texture

Memory latency

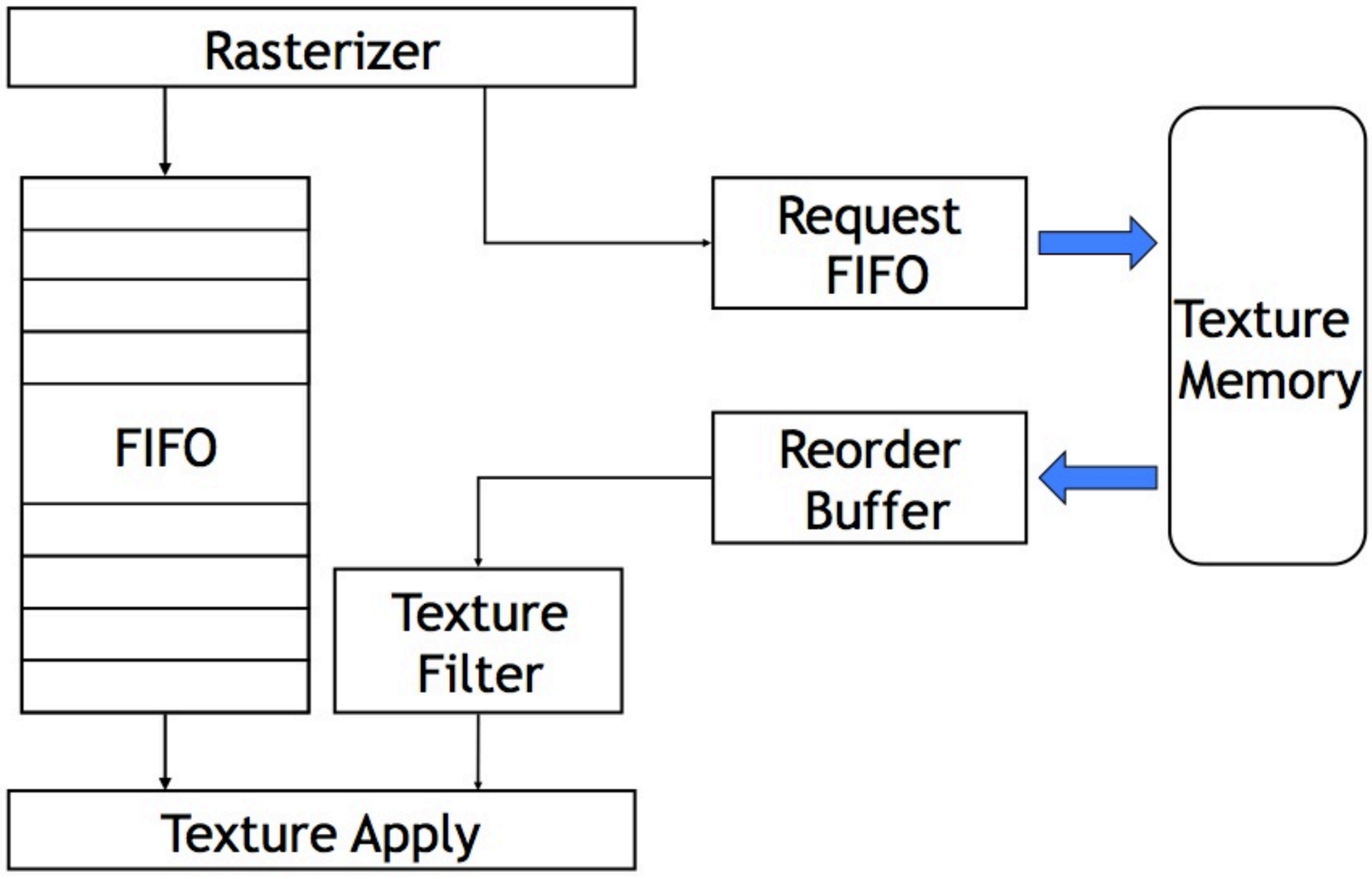
- Processor requests texture data → processor waits for hundreds of cycles (Very bad)

- Recall: GPUs will miss the cache more often than CPUs (fundamental to the streaming workload)

- Solution prior to programmable cores on GPUs: texture prefetching
  - See today's reading: Igehy et al. *Prefetching in a Texture Cache Architecture*

- Solution in modern GPUs: multi-threaded programmable cores
  - Subject of a later lecture

Large fragment FIFOs



Note: This diagram does not contain a texture cache. See reading for implementation of prefetching with caching.

Slide credit: Akeley and Hanrahan

Texture summary

- Pre-filtering texture data reduces aliasing
  - Mip-mapping fundamental to texture system design
  - Avoid aliasing under minification
  - Improve cache behavior under minification

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

- GPU texture caches:
  - Primarily serve to amplify limited DRAM bandwidth
  - Not to reduce latency to off-chip memory
  - Small in size, multi-ported (e.g., need to access 8 texels simultaneously)

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

- Texture access latency is hidden by prefetching (in the old days) and multi-threading (in modern GPUs)
  - The design of a modern GPU processing core is influenced heavily by the need to hide texture access latency

Readings

- Z. Hakura and a. Gupta, *The Design and Analysis of a Cache Architecture for Texture Mapping*. ISCA 97
- H. Igehy et al., *Prefetching in a Texture Cache Architecture*. Graphics Hardware 1998