Lecture 6: Texture

Kayvon Fatahalian
CMU 15-869: Graphics and Imaging Architectures (Fall 2011)

Today: texturing!

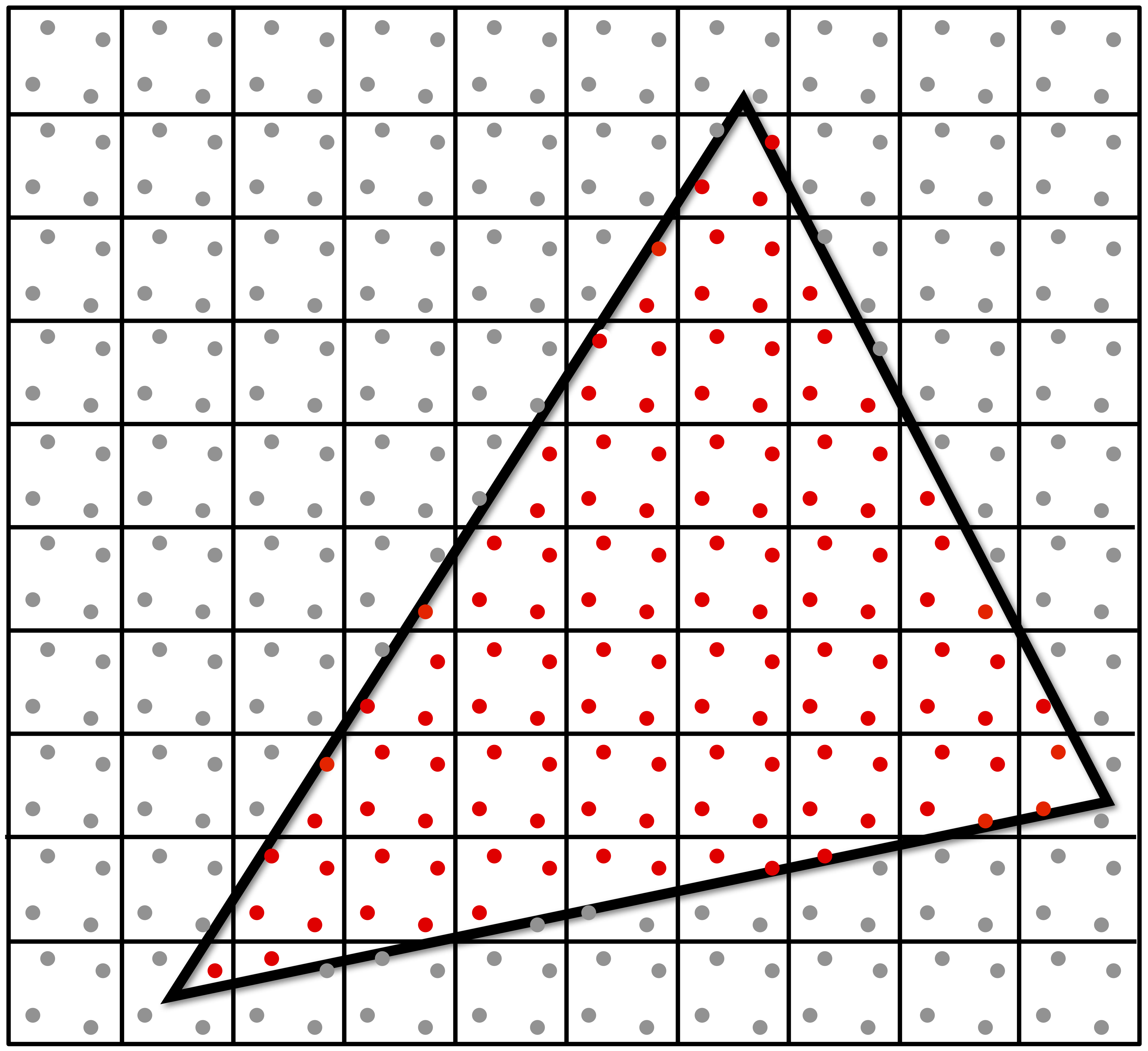
- Texture filtering
  - Texture access is not just a 2D array lookup ;-)

- Memory-system implications
  - Caching
  - Storage layouts
  - Prefetching

Last time

Rasterizer point-samples coverage (4 samples per pixel shown here)

Z-buffer algorithm used to solve for occlusion at these points



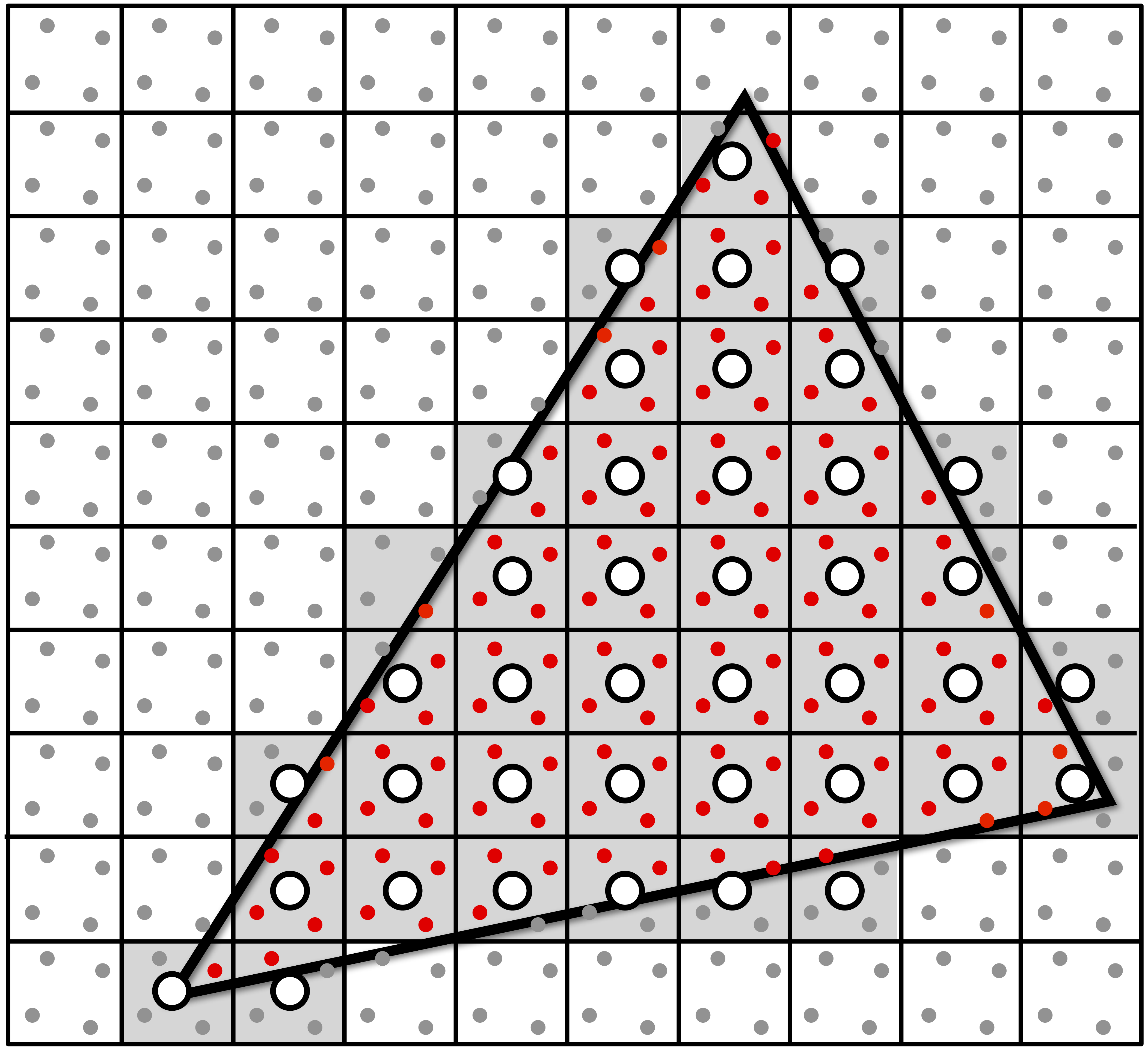
Last time

One fragment per covered pixel

Triangle attributes are interpolated from vertex values

Attributes [typically] sampled at pixel centers to generate values for fragment attributes

(recall modern rasterizers may produce attribute eqns, not values)



Shading the fragment

HLSL shader program: defines behavior of fragment processing stage

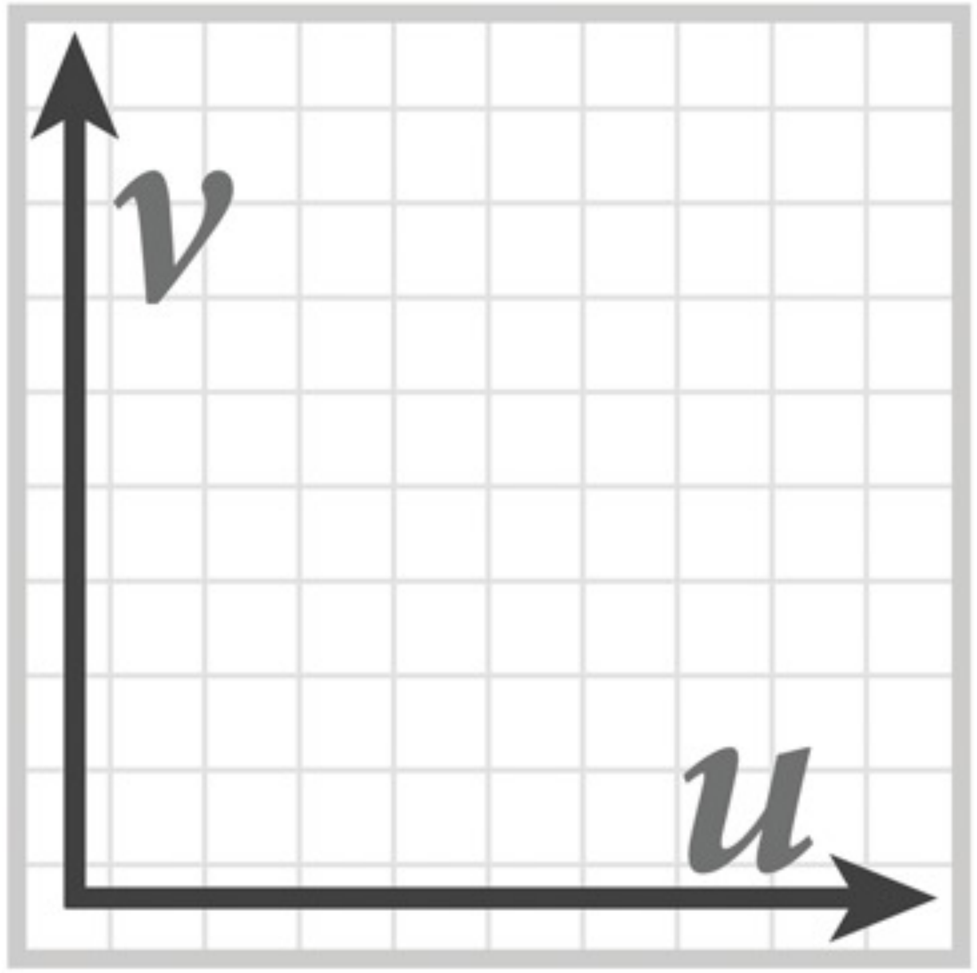
```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

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

Let:

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

`myTex =`



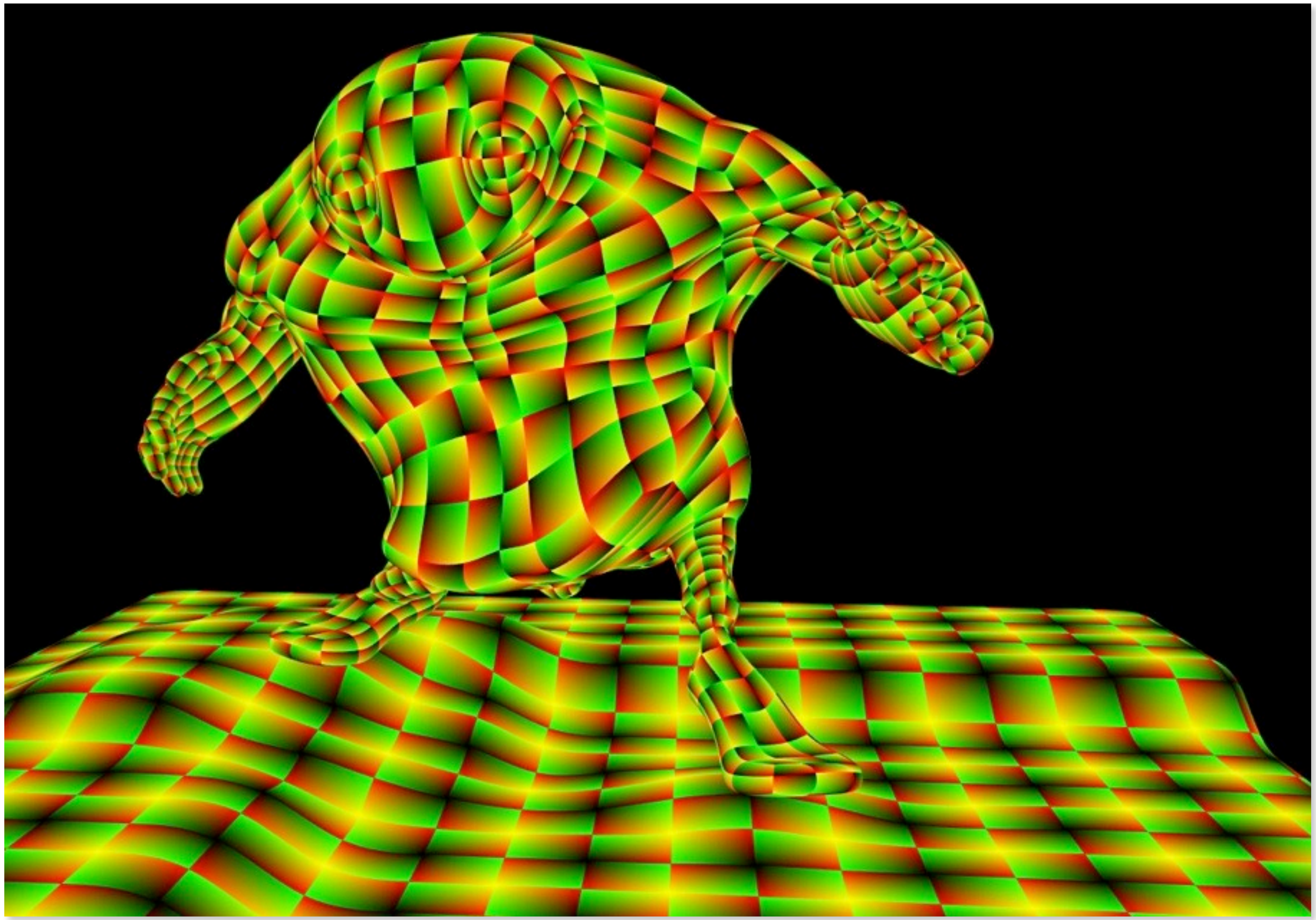
function defined on $[0,1]^2$ domain:

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

(represented by 2048x2048 image)

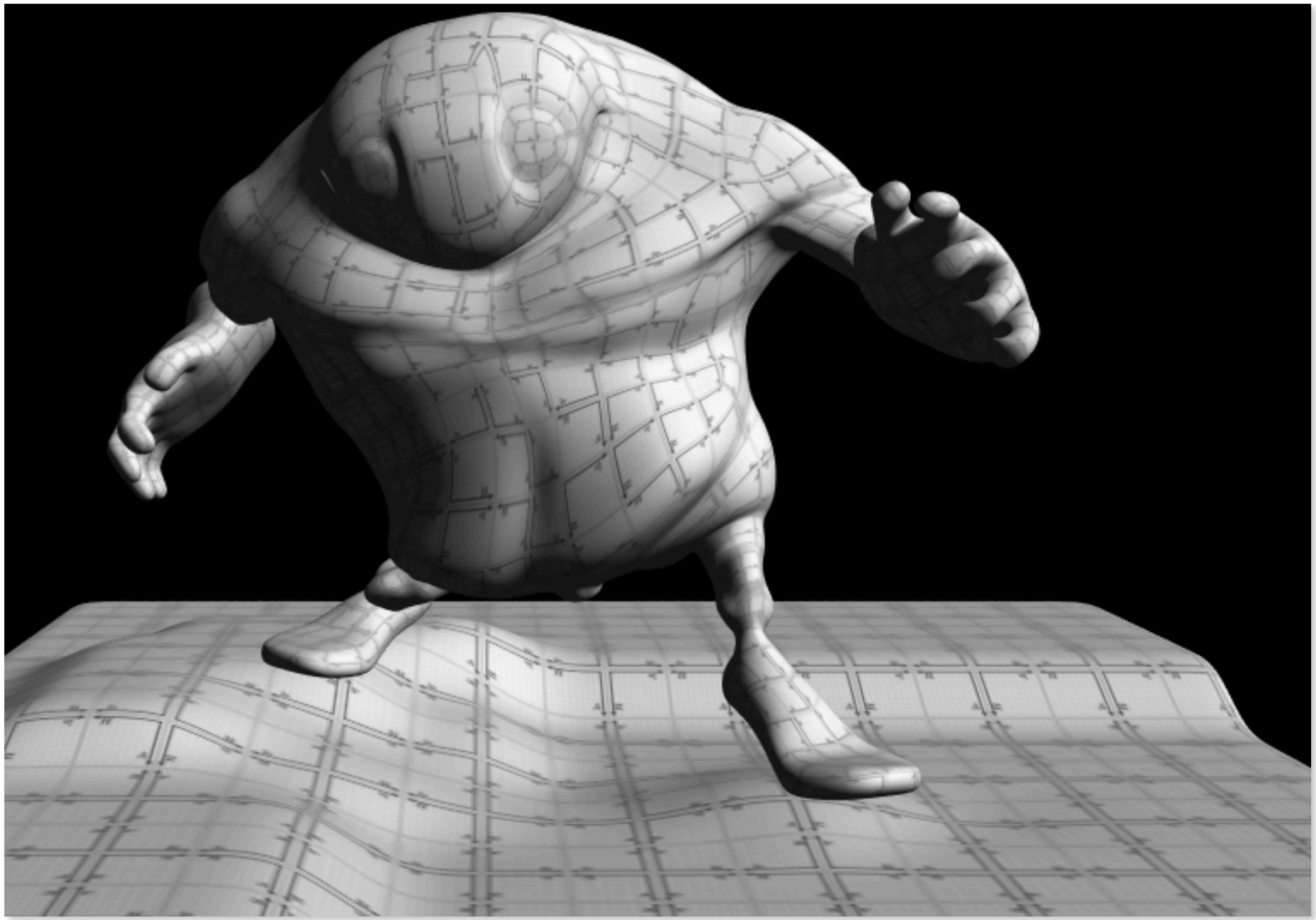
`mySamp` **defines how to resample function to generate value at (u,v)**

Texture coordinates (UV)

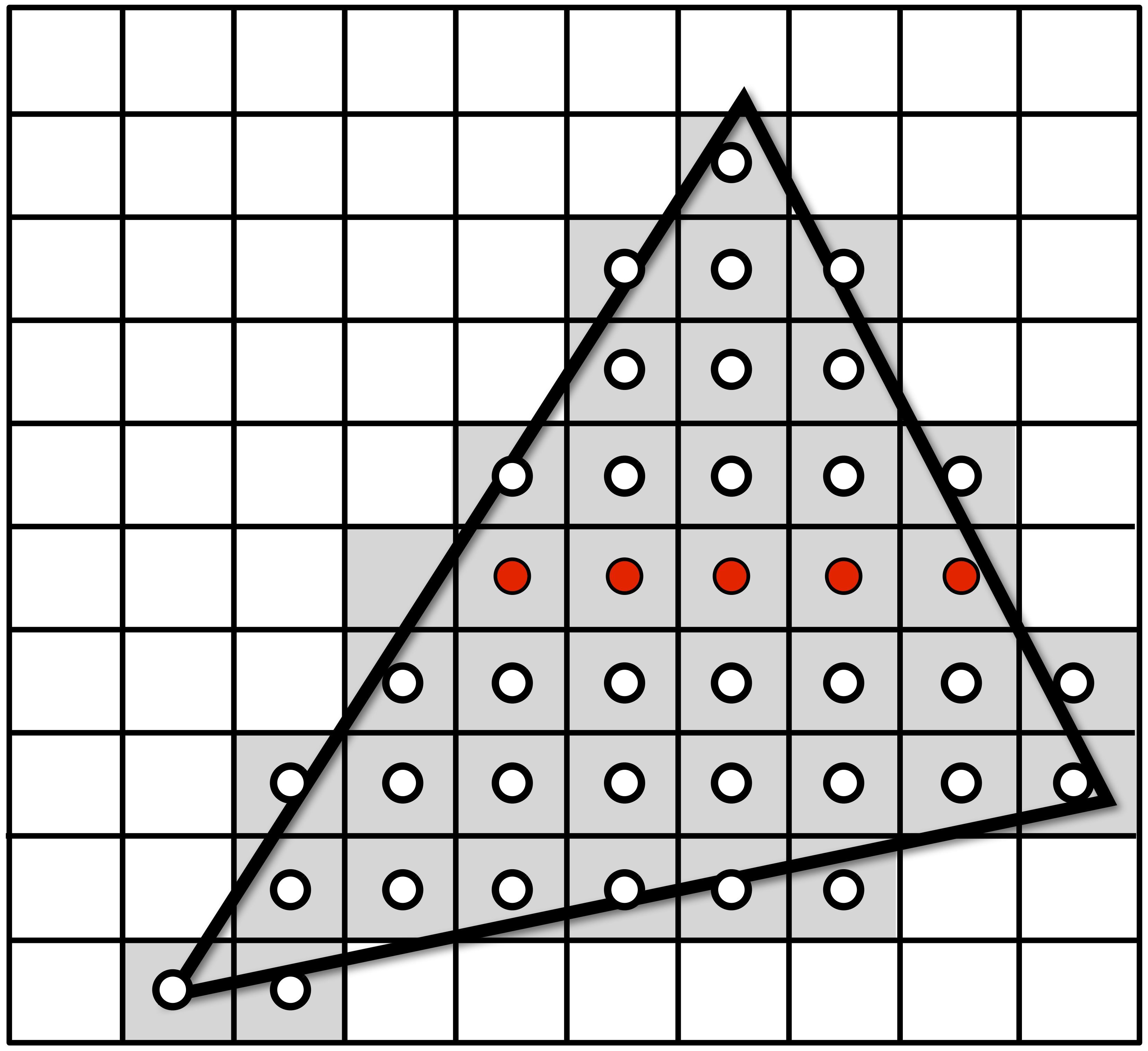


Red channel = u
Green channel = v

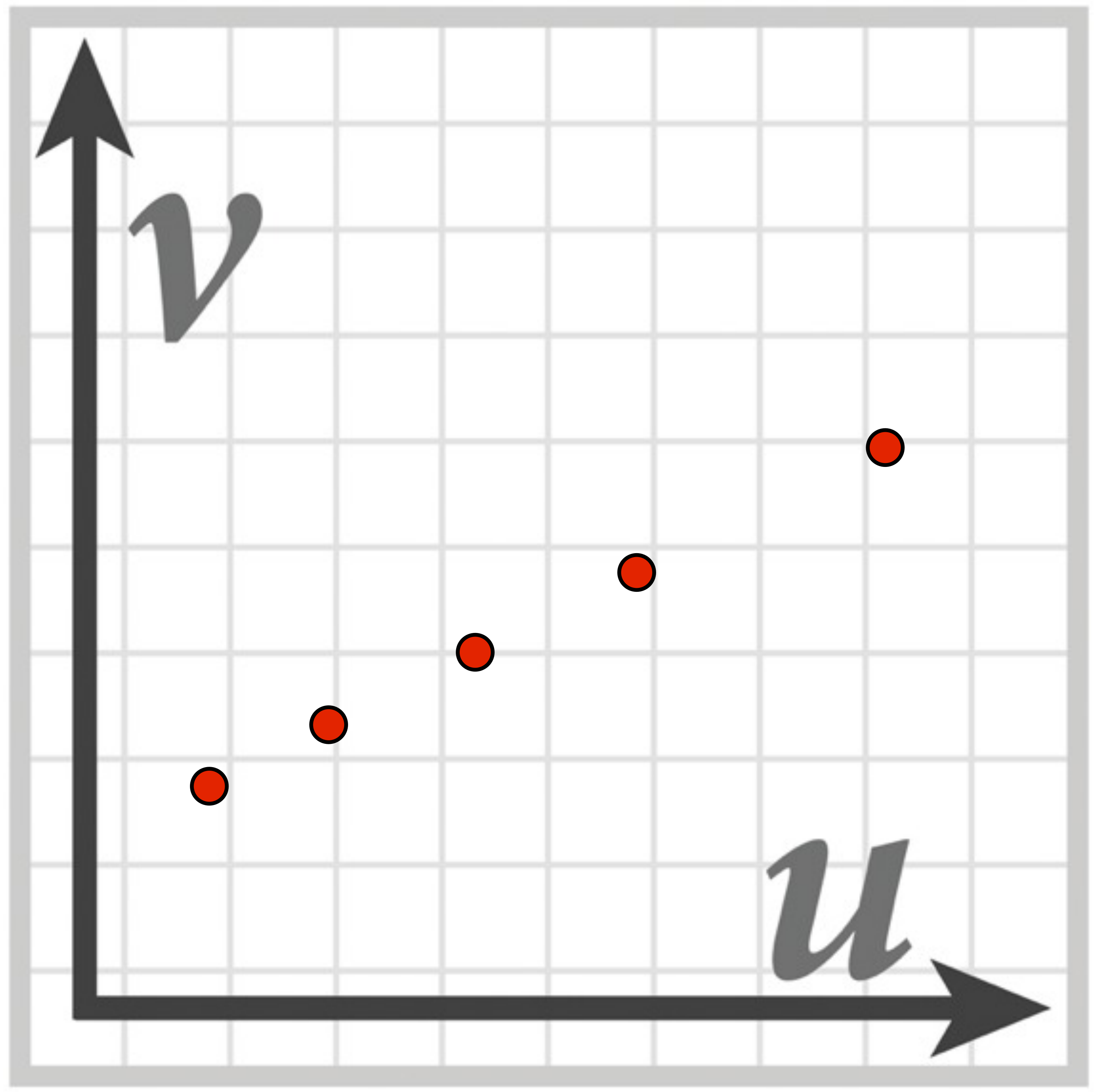
Shaded result



Texture space

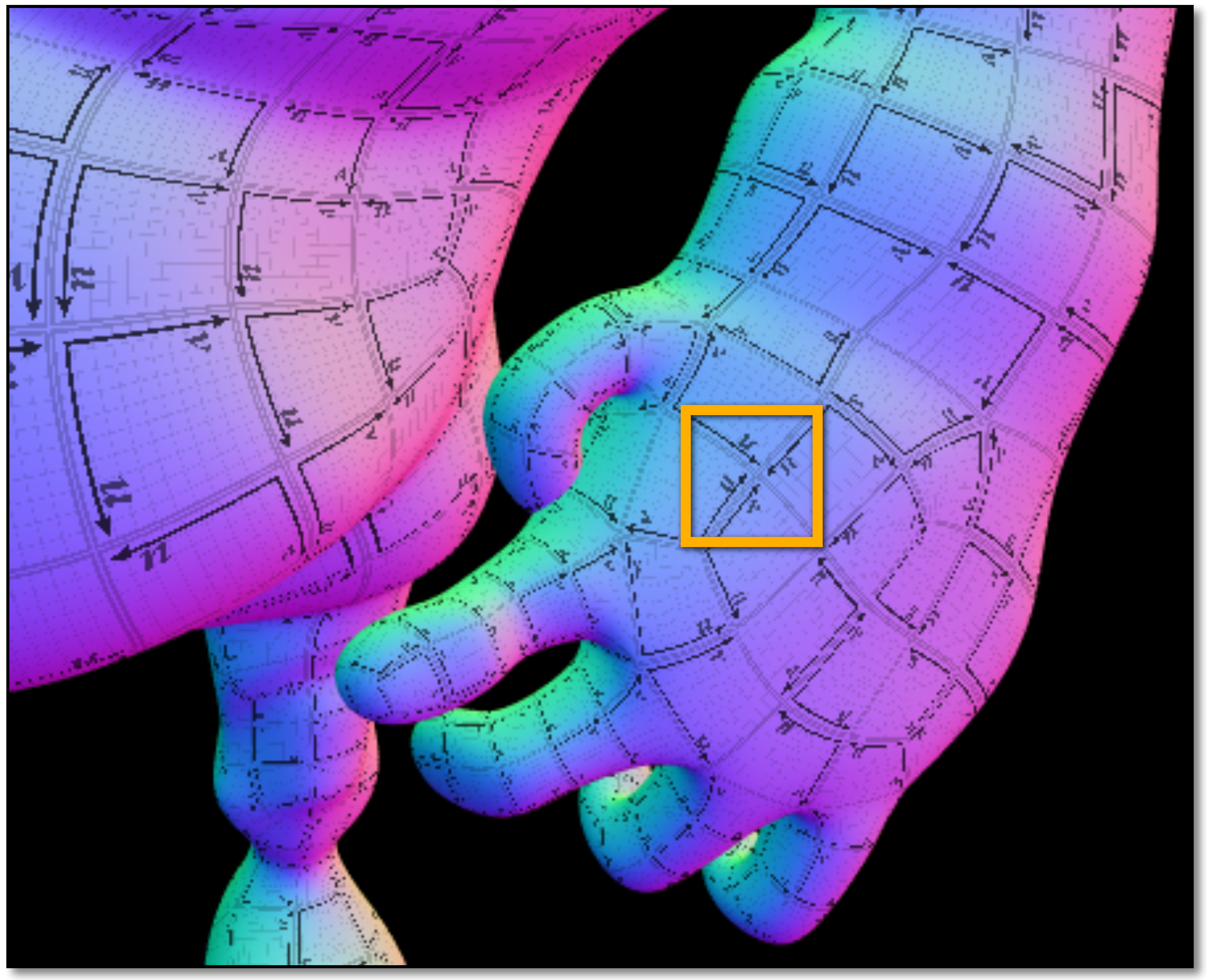


Screen space

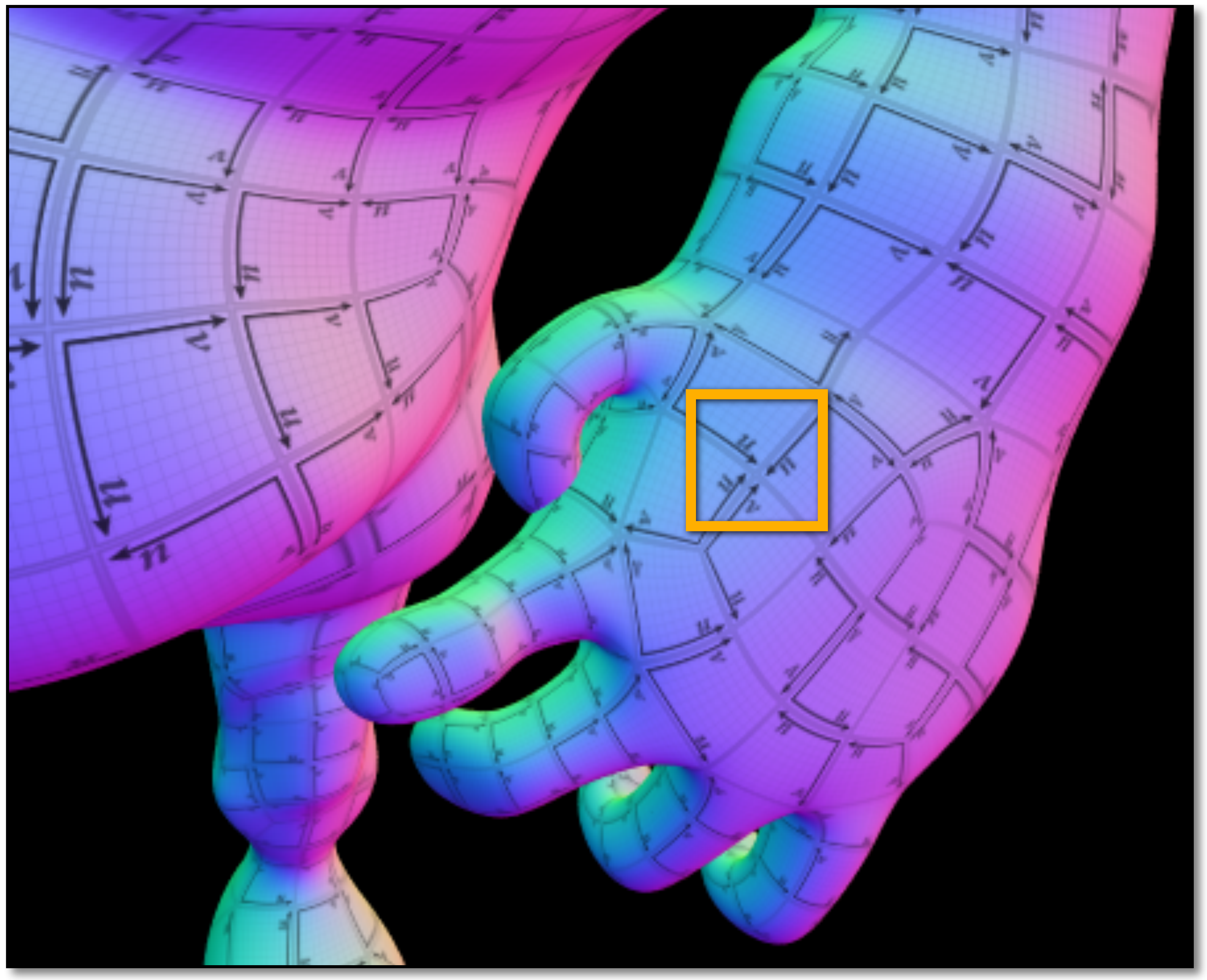


Texture space

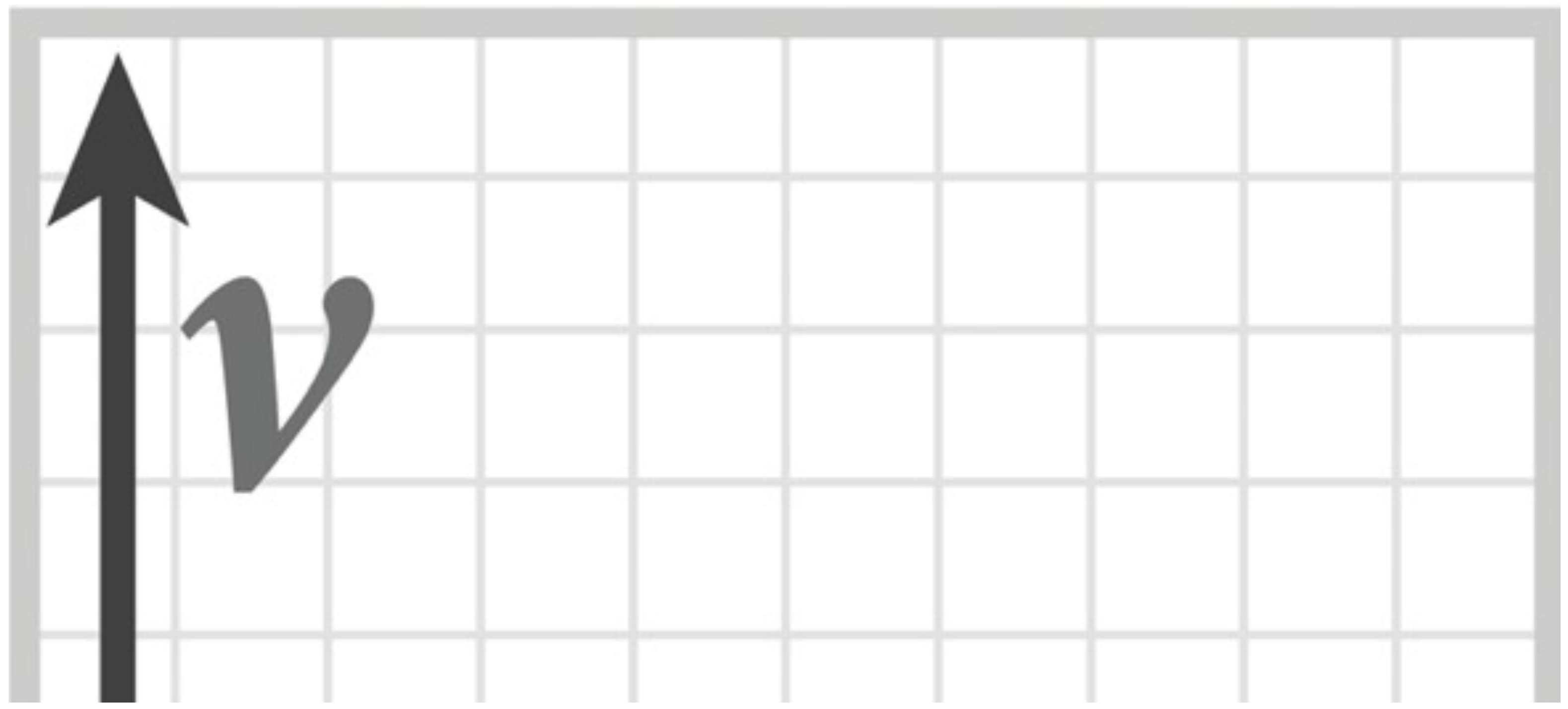
Aliasing due to undersampling



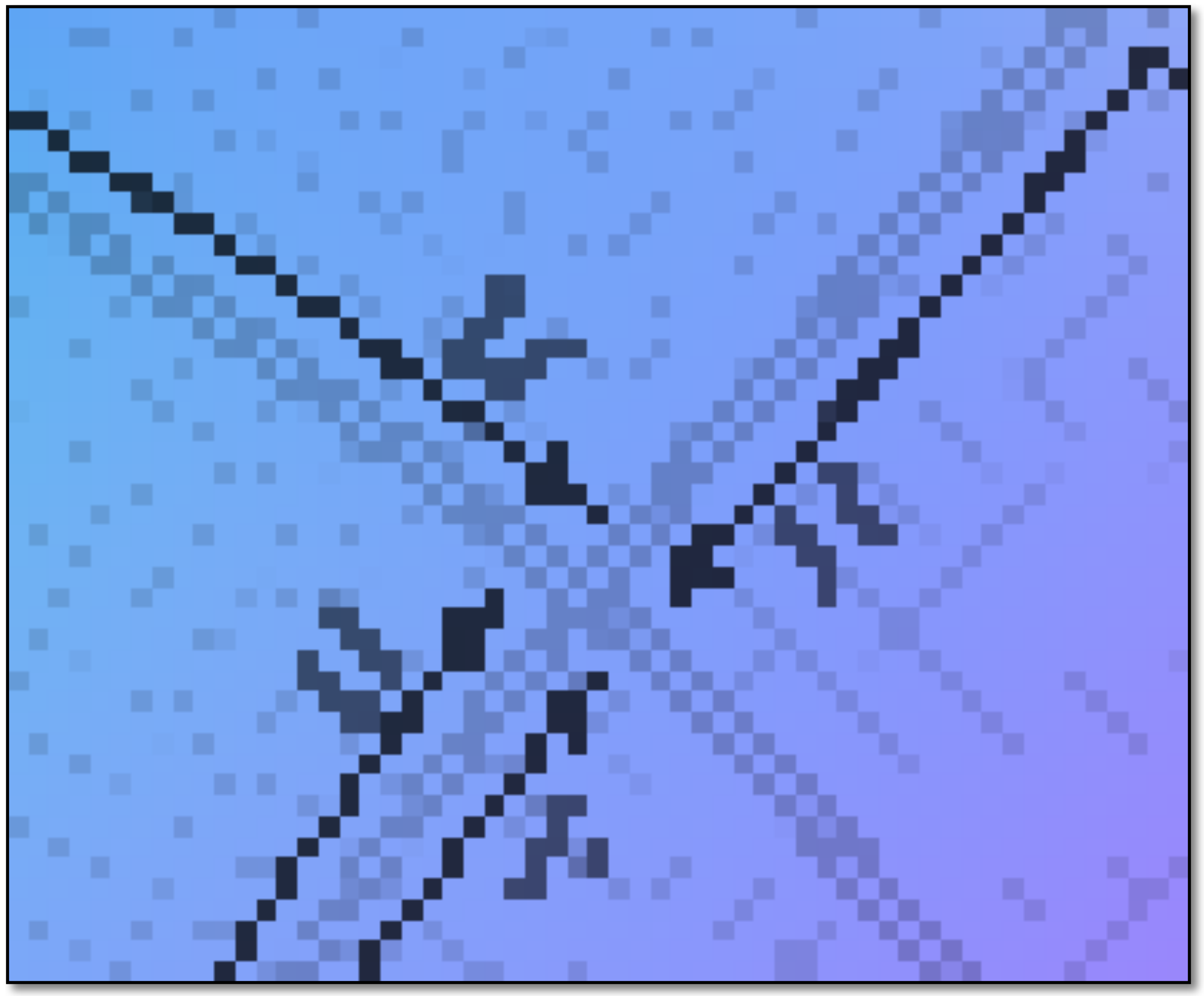
No pre-filtering
(aliased result)



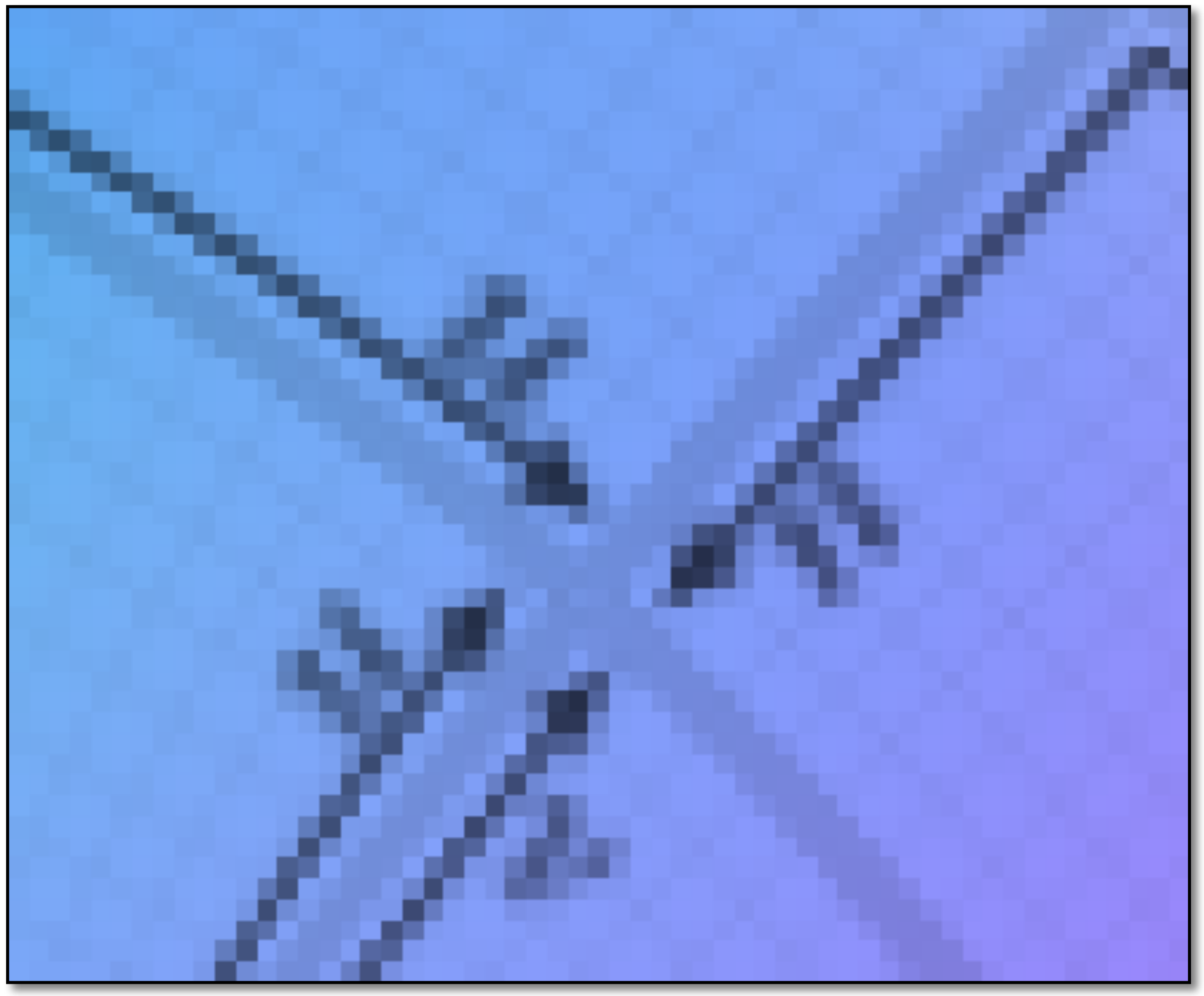
Pre-filtered texture



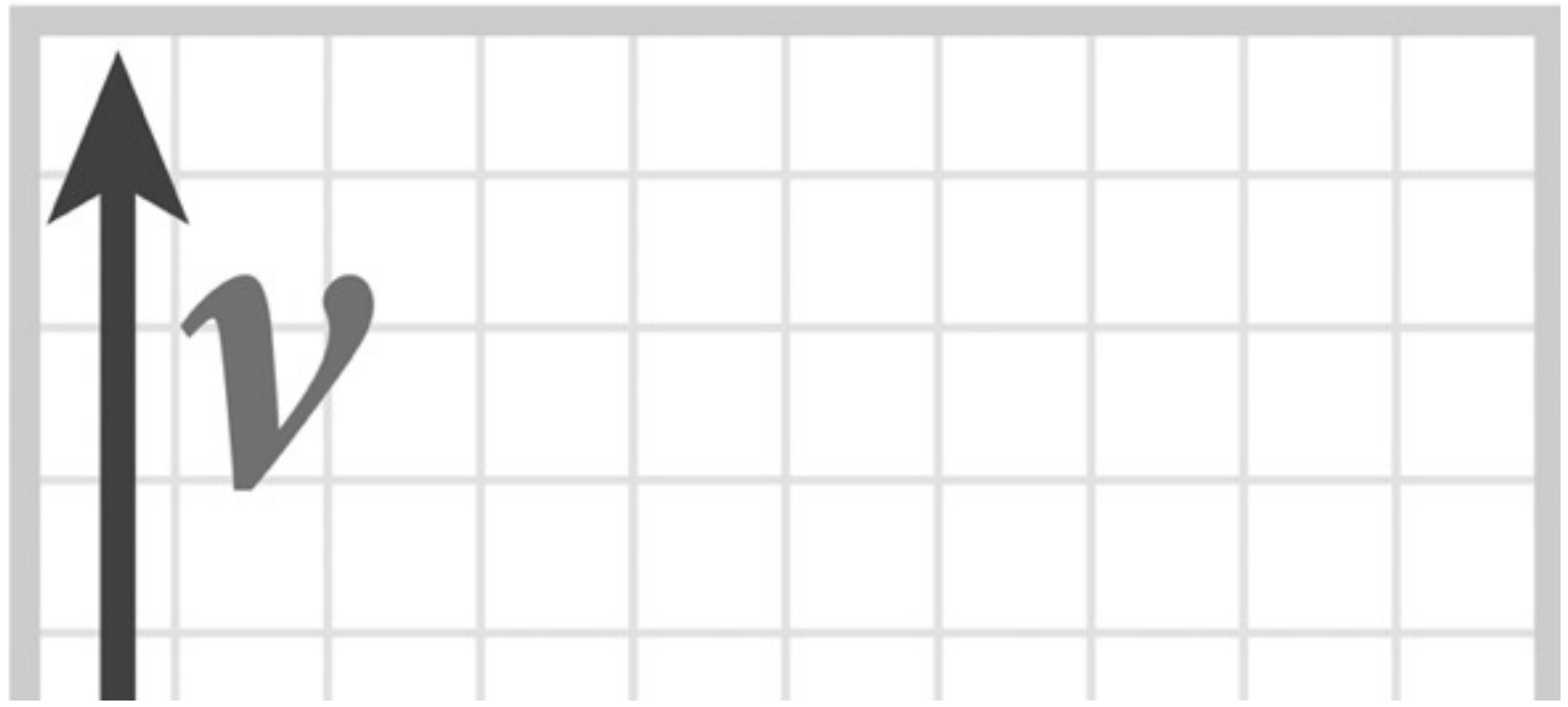
Aliasing due to undersampling

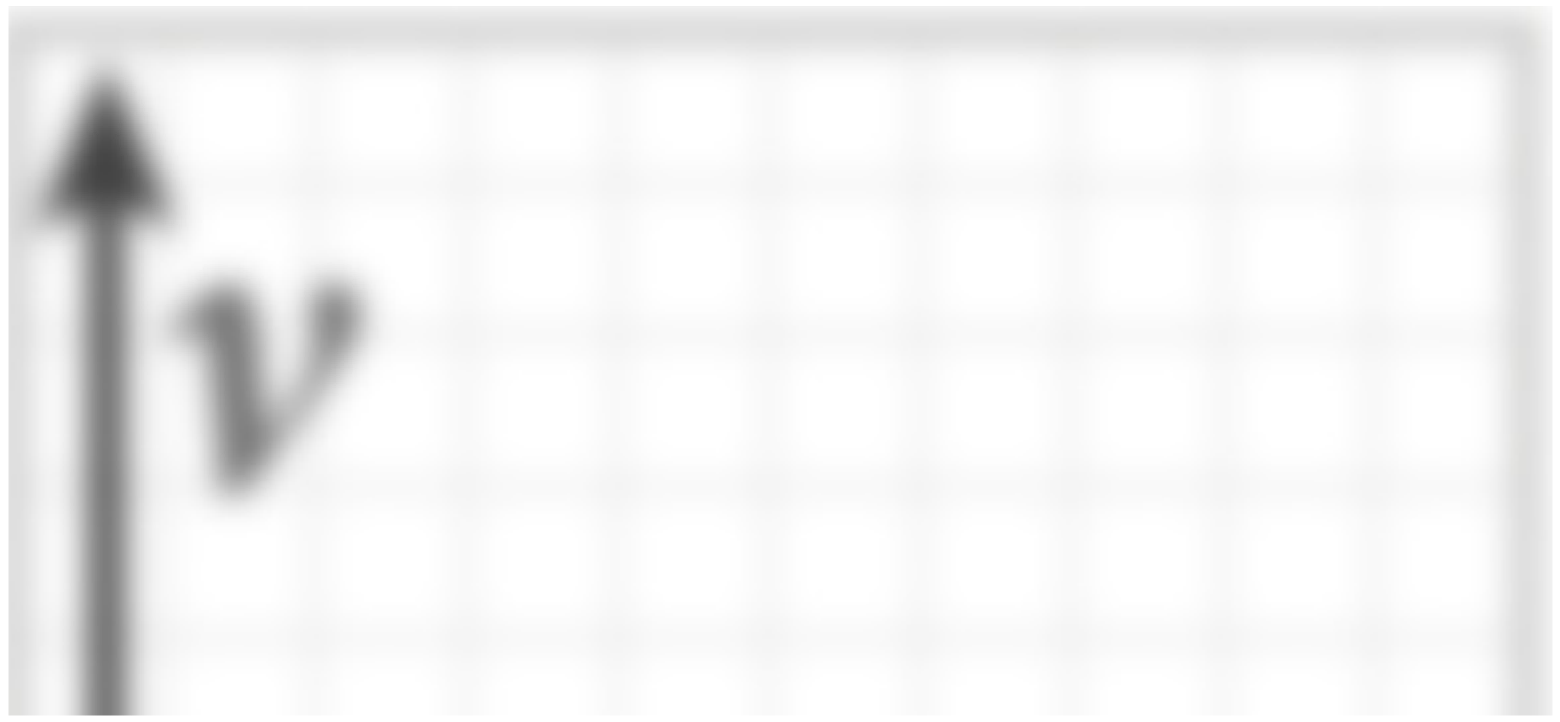


No pre-filtering
(aliased result)

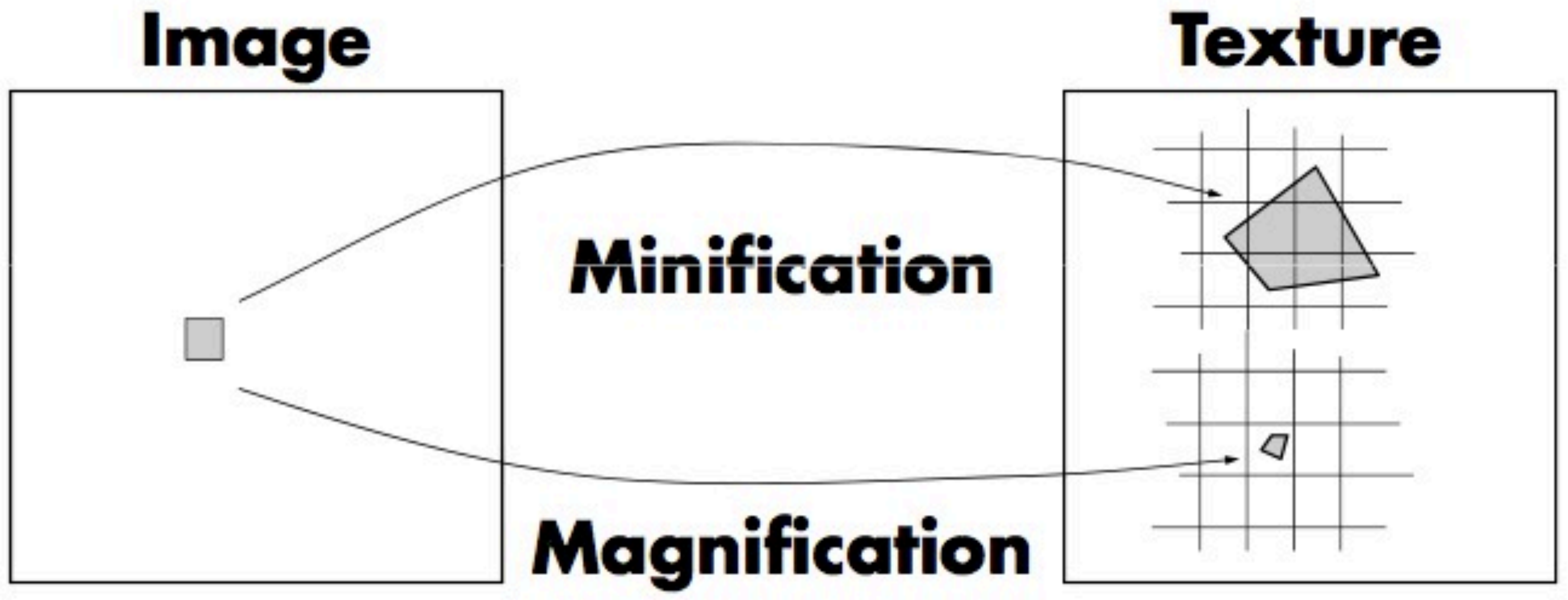


Pre-filtered texture





Filtering textures



- Texture footprint

  Footprint changes from pixel to pixel

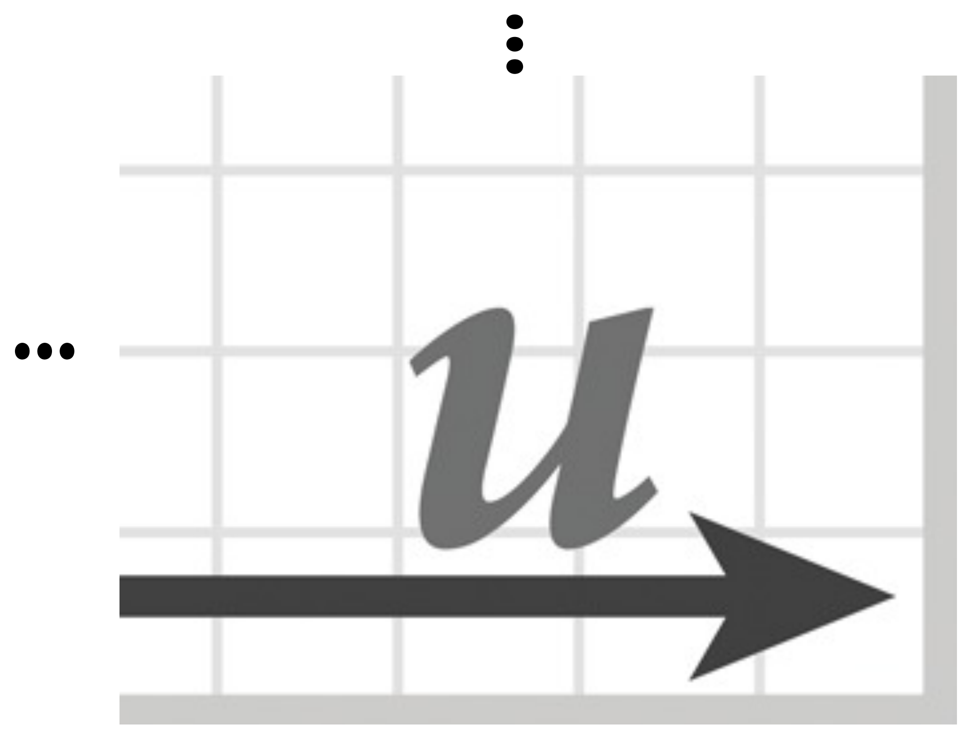
  i.e. not shift-invariant

- Resampling theory: two cases

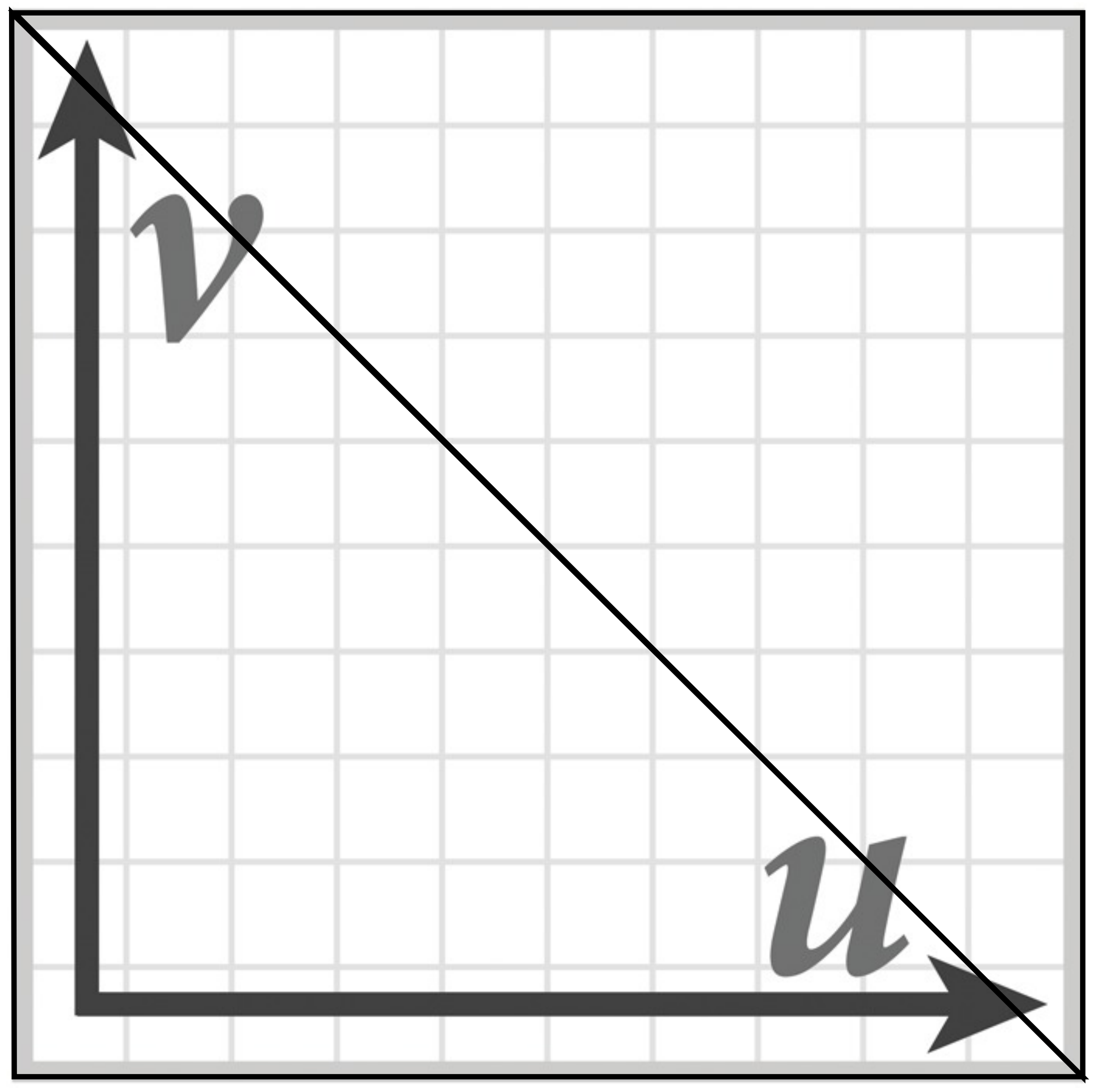
  1. Magnification => Interpolation

  2. Minification => Filter (averaging)

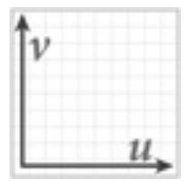
Filtering textures



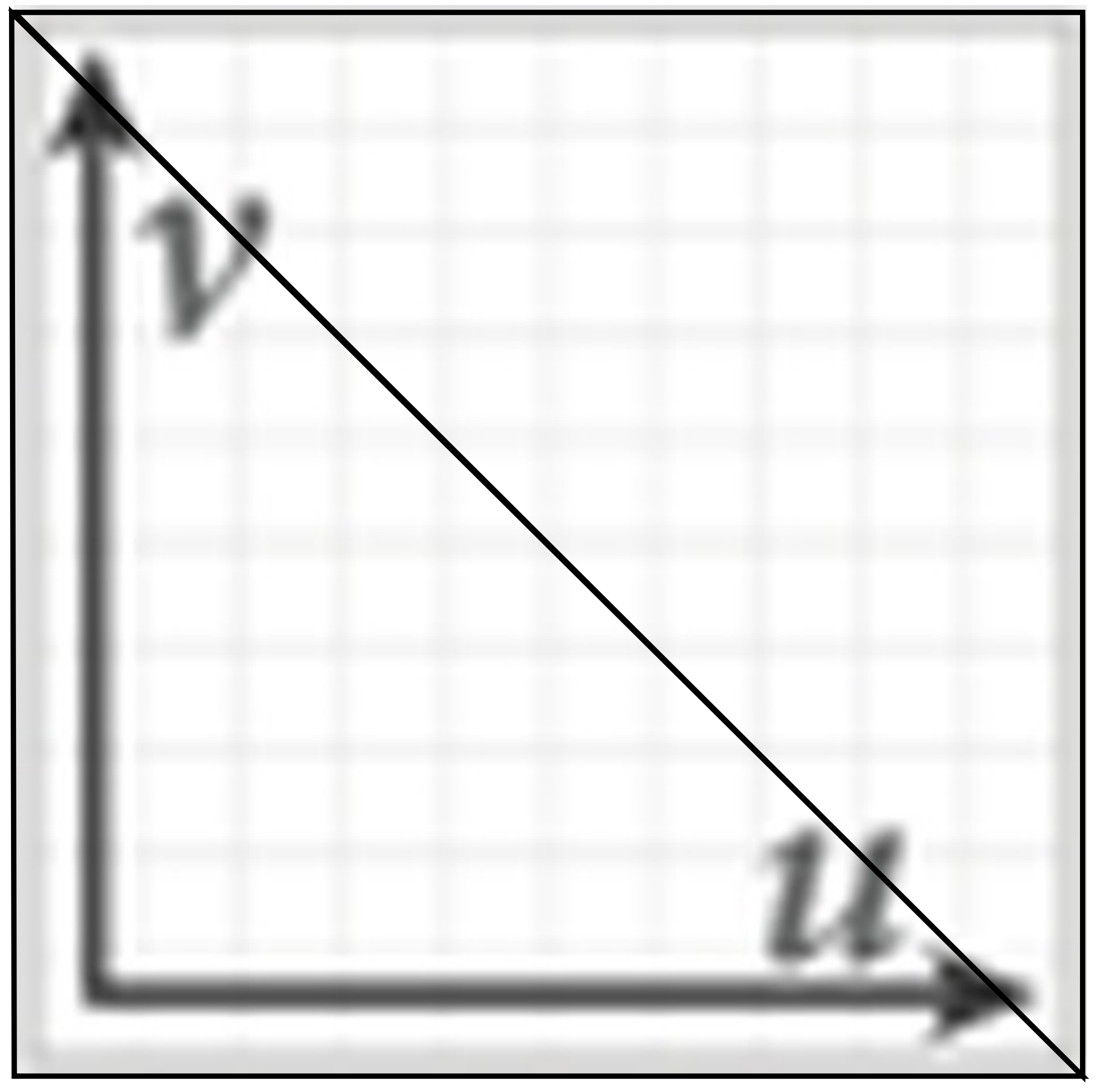
Actual texture: 700x700



Texture minification

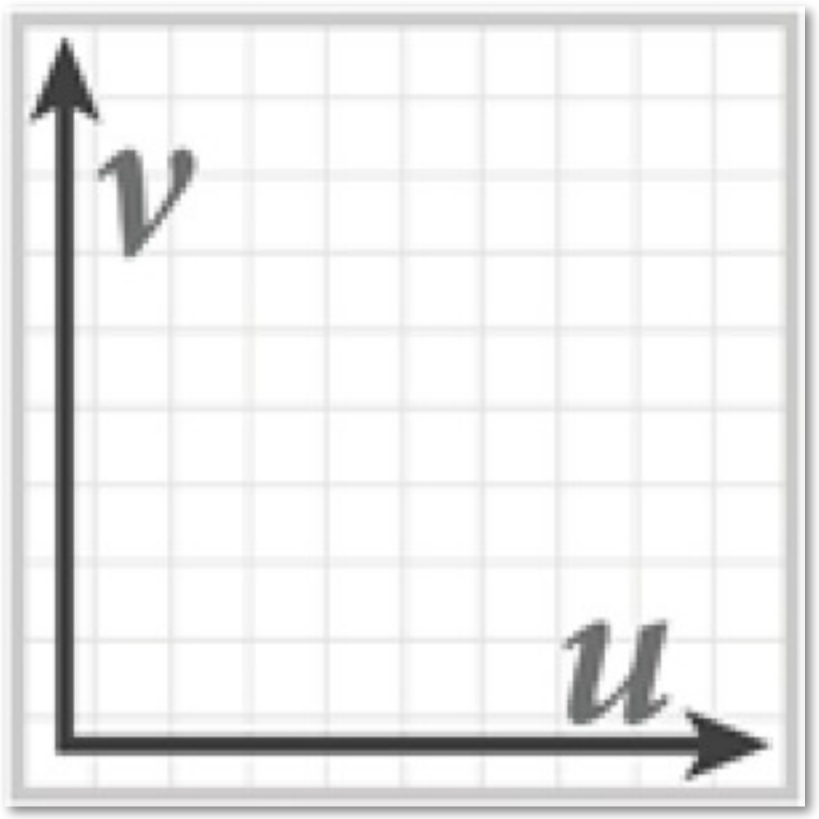


Actual texture: 64x64

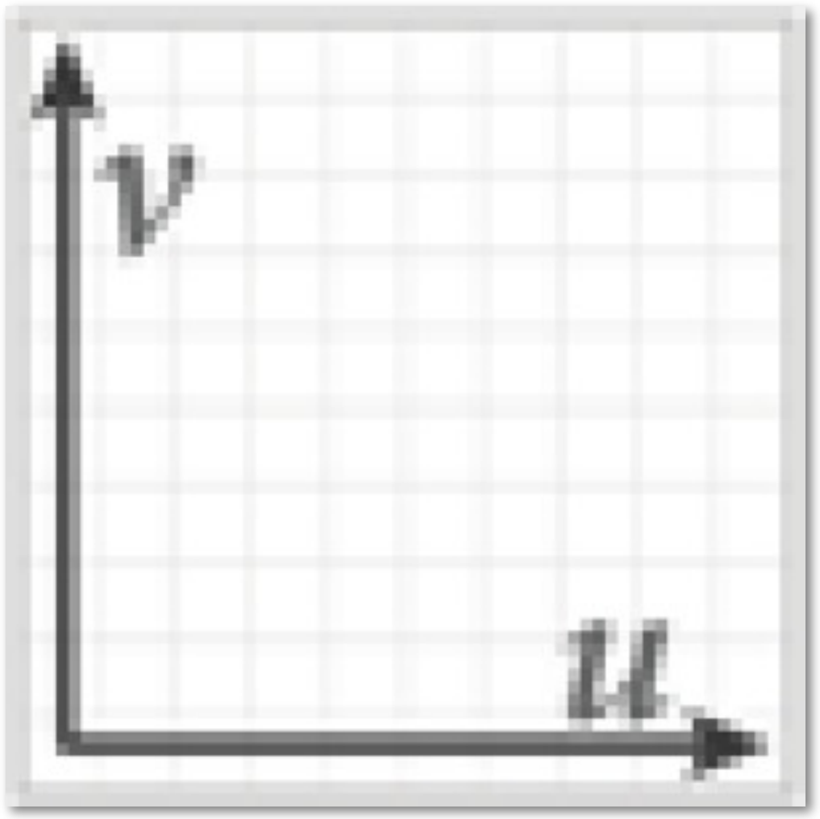


Texture magnification

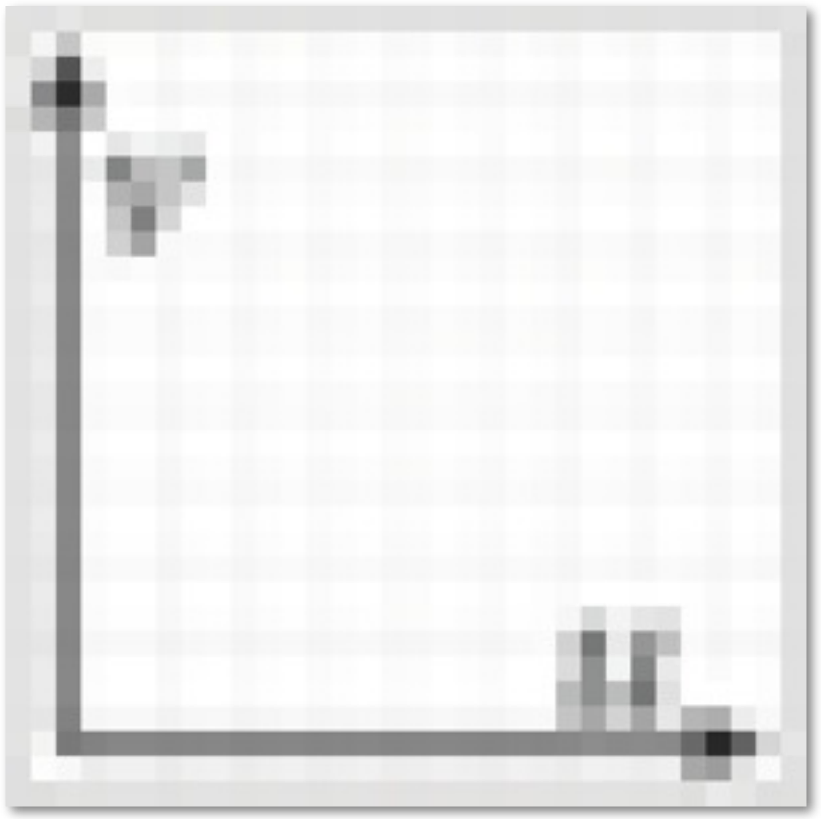
Mipmap (L. Williams 83)



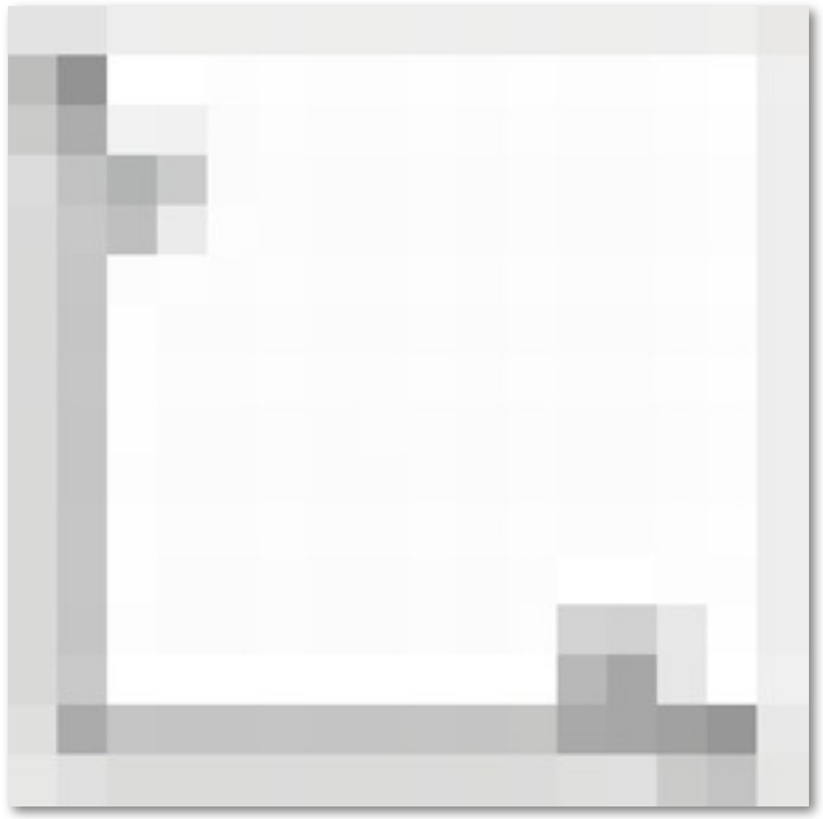
Level 0 = 128x128



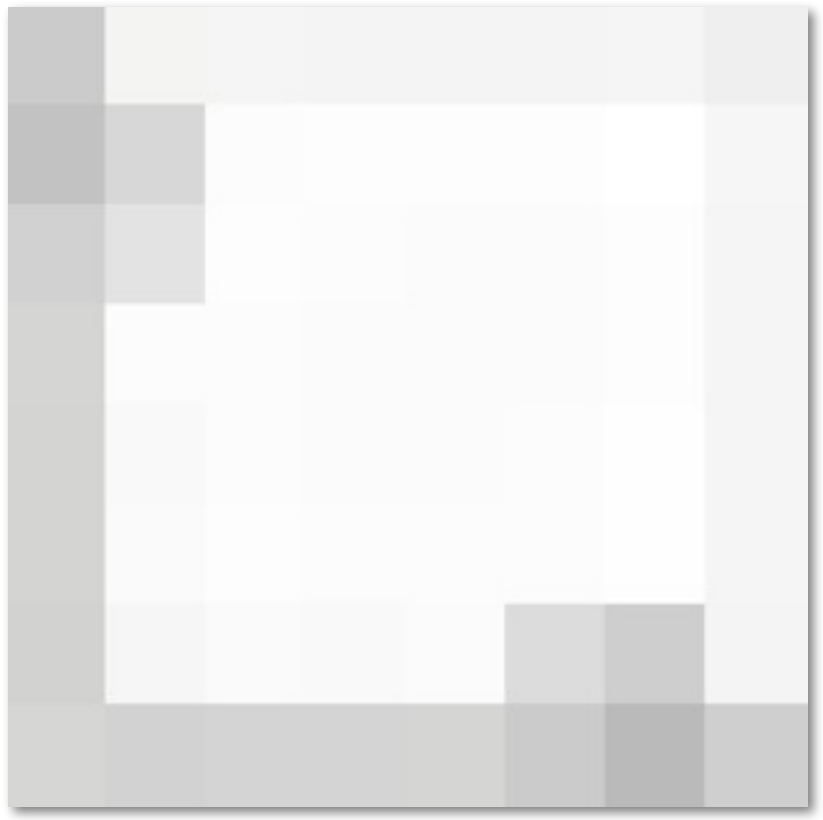
Level 1 = 64x64



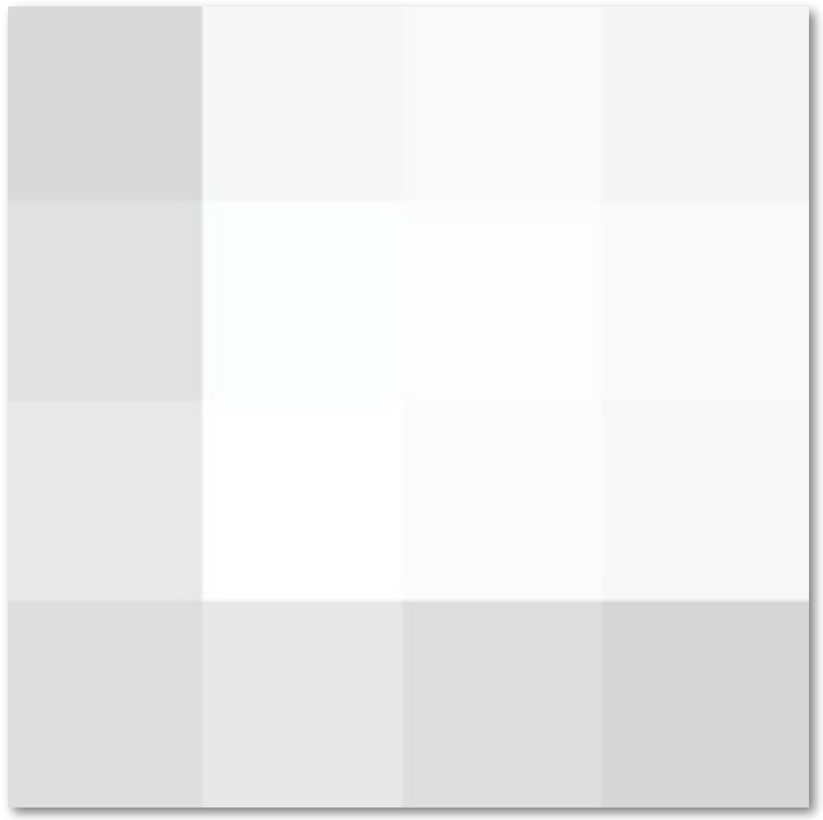
Level 2 = 32x32



Level 3 = 16x16



Level 4 = 8x8



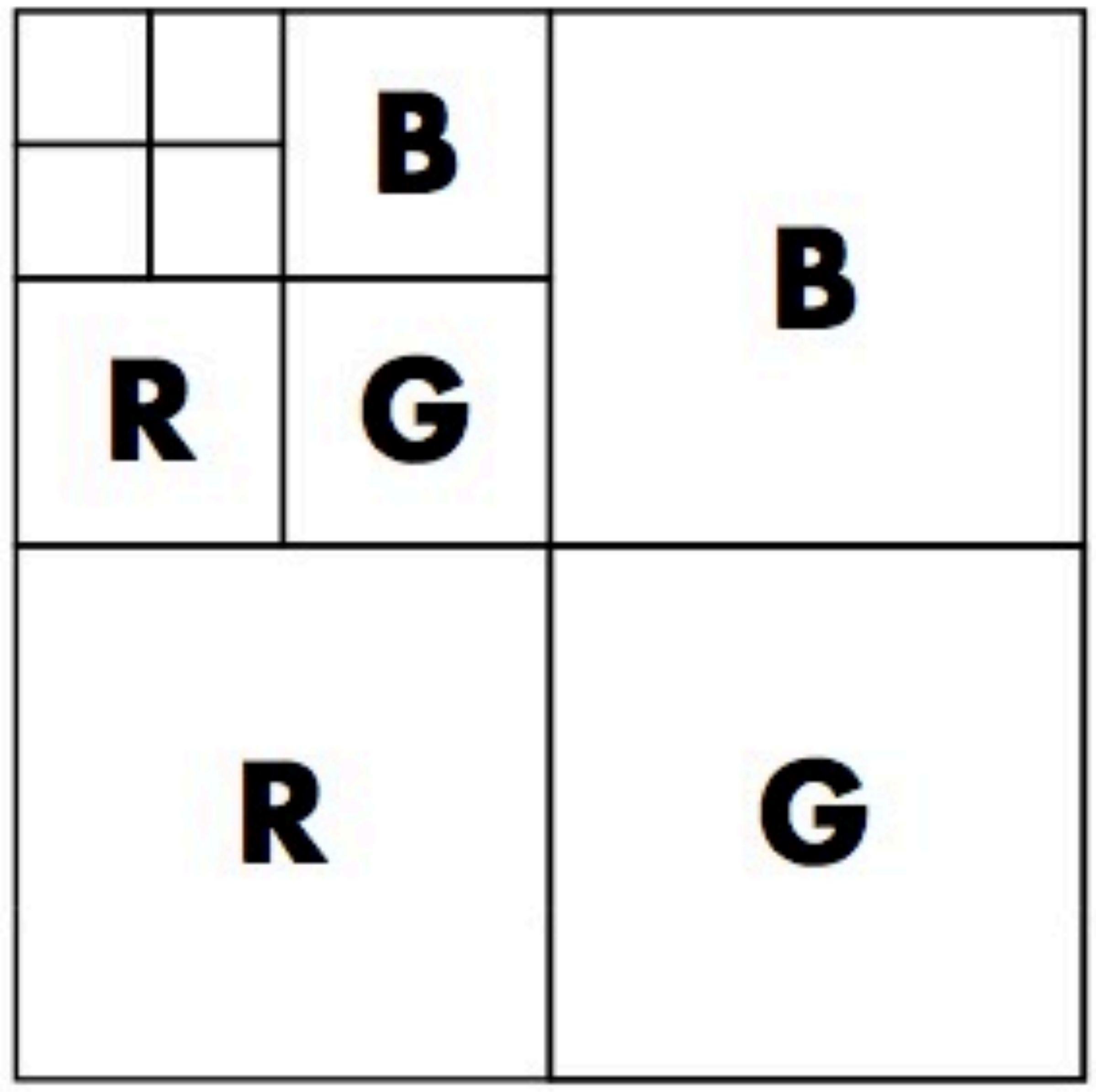
Level 5 = 4x4

Level 6 = 2x2

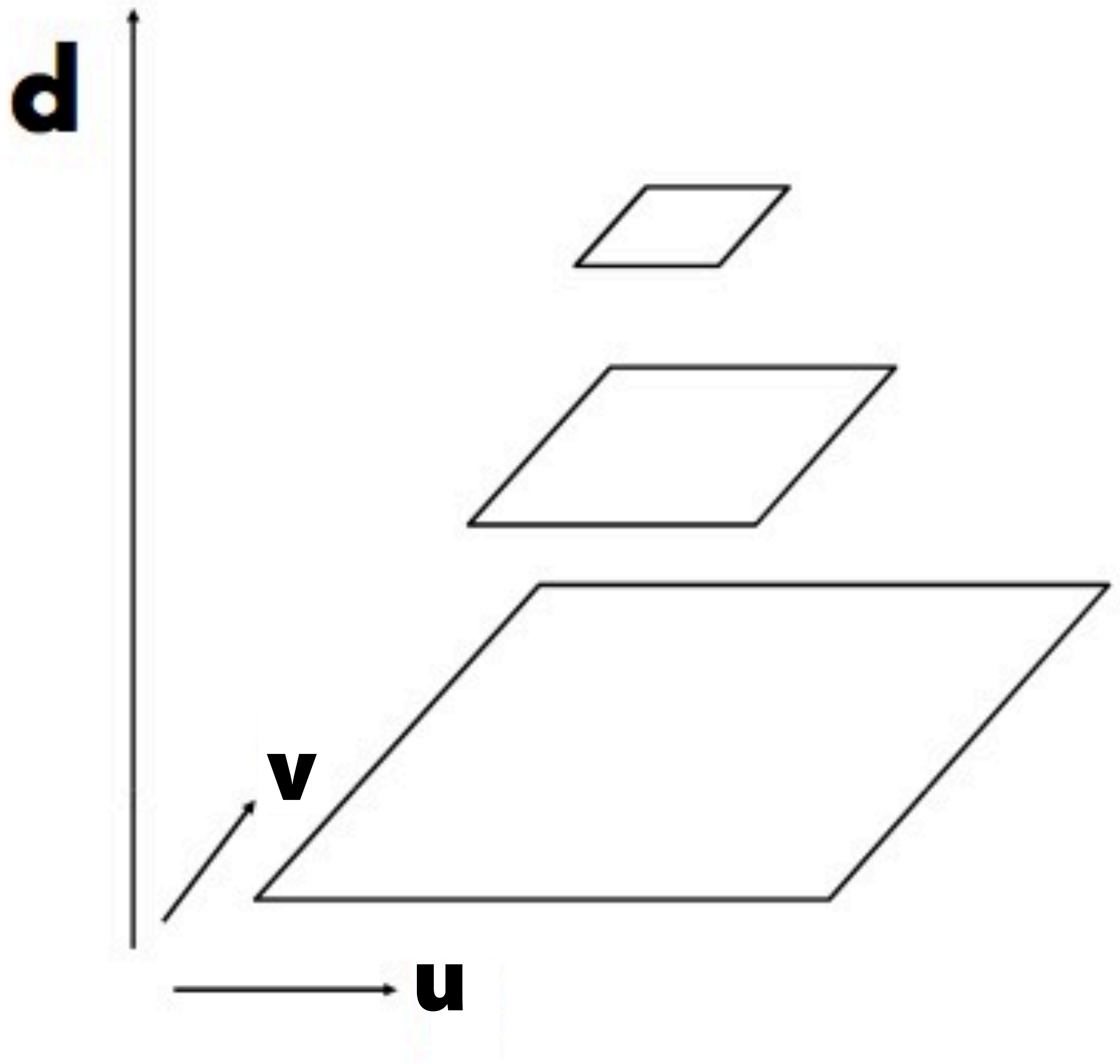
Level 7 = 1x1

Pre-filter texture data

Mipmap (L. Williams 83)

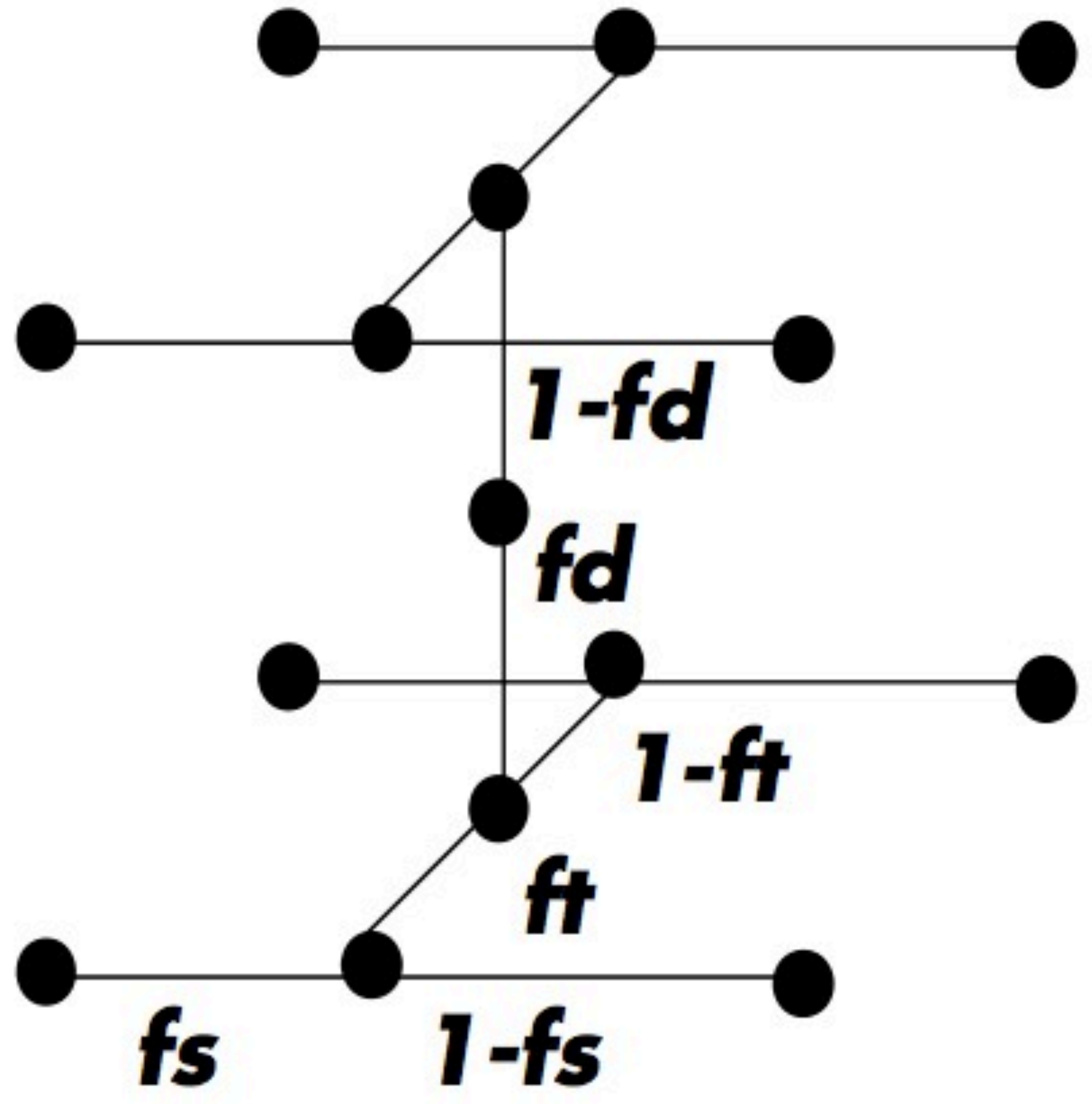


Williams' original proposed layout



"Mip hierarchy"
level = d

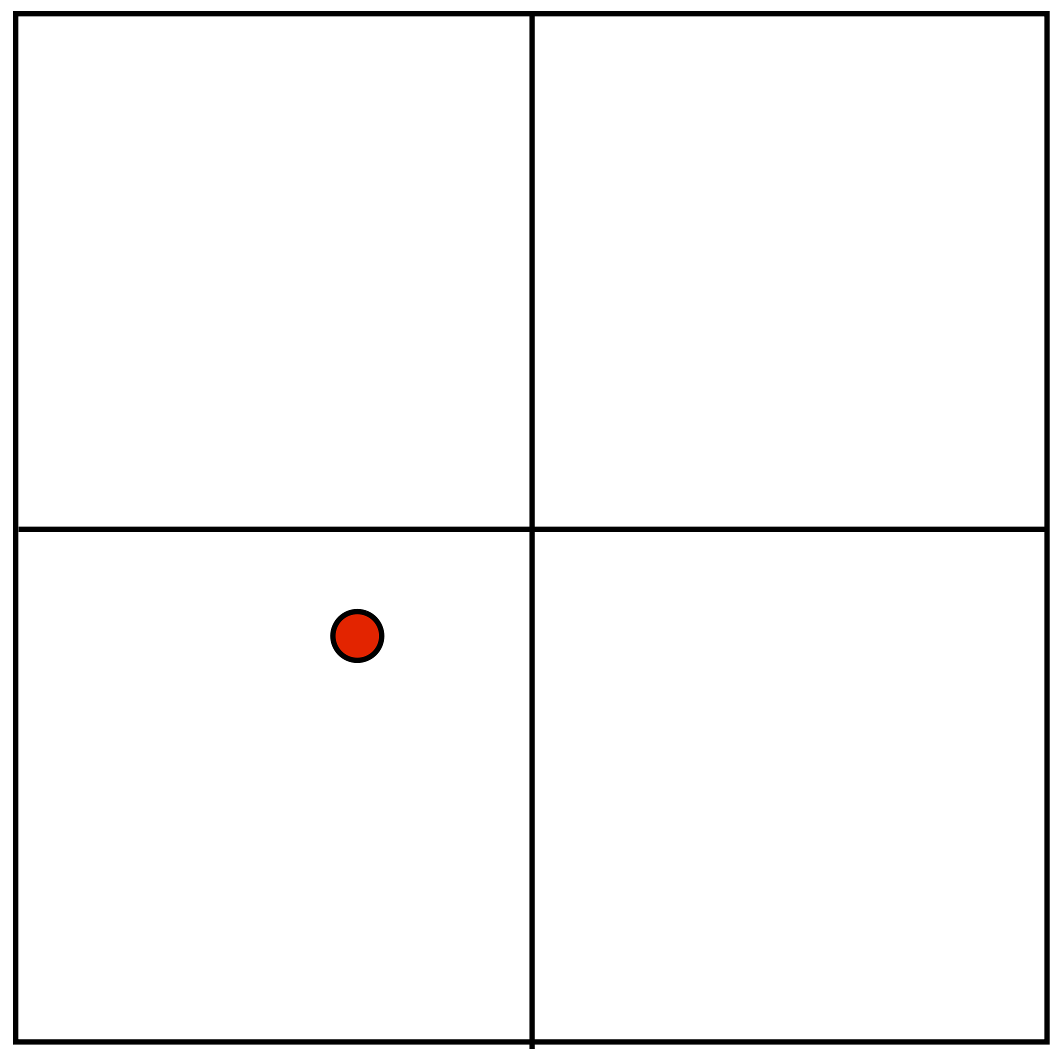
Constant-time filtering



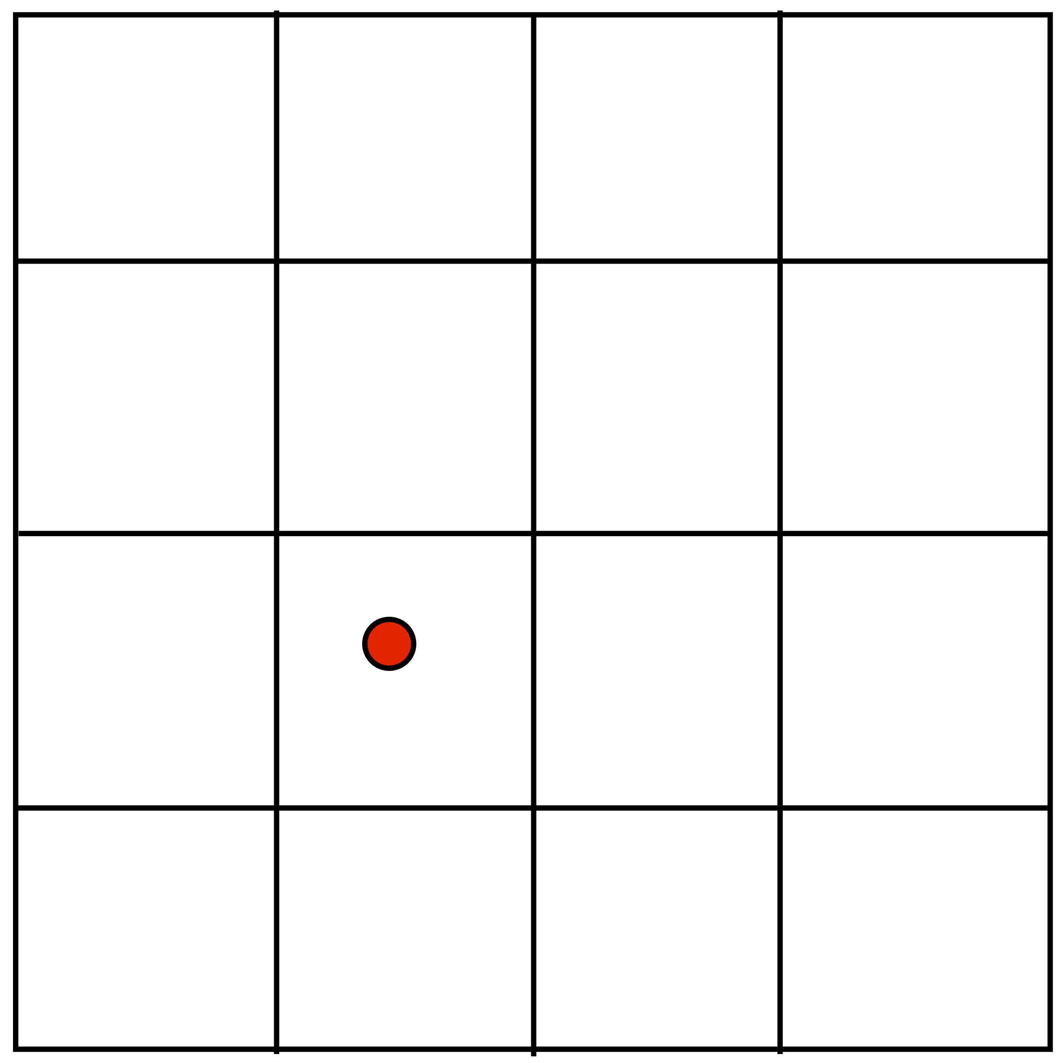
$$lerp(t, v_1, v_2) = v_1 + t(v_2 - v_1)$$

Bilinear: 3 lerps (3 mul + 6 add)

Trilinear: 7 lerps (7 mul + 14 add)



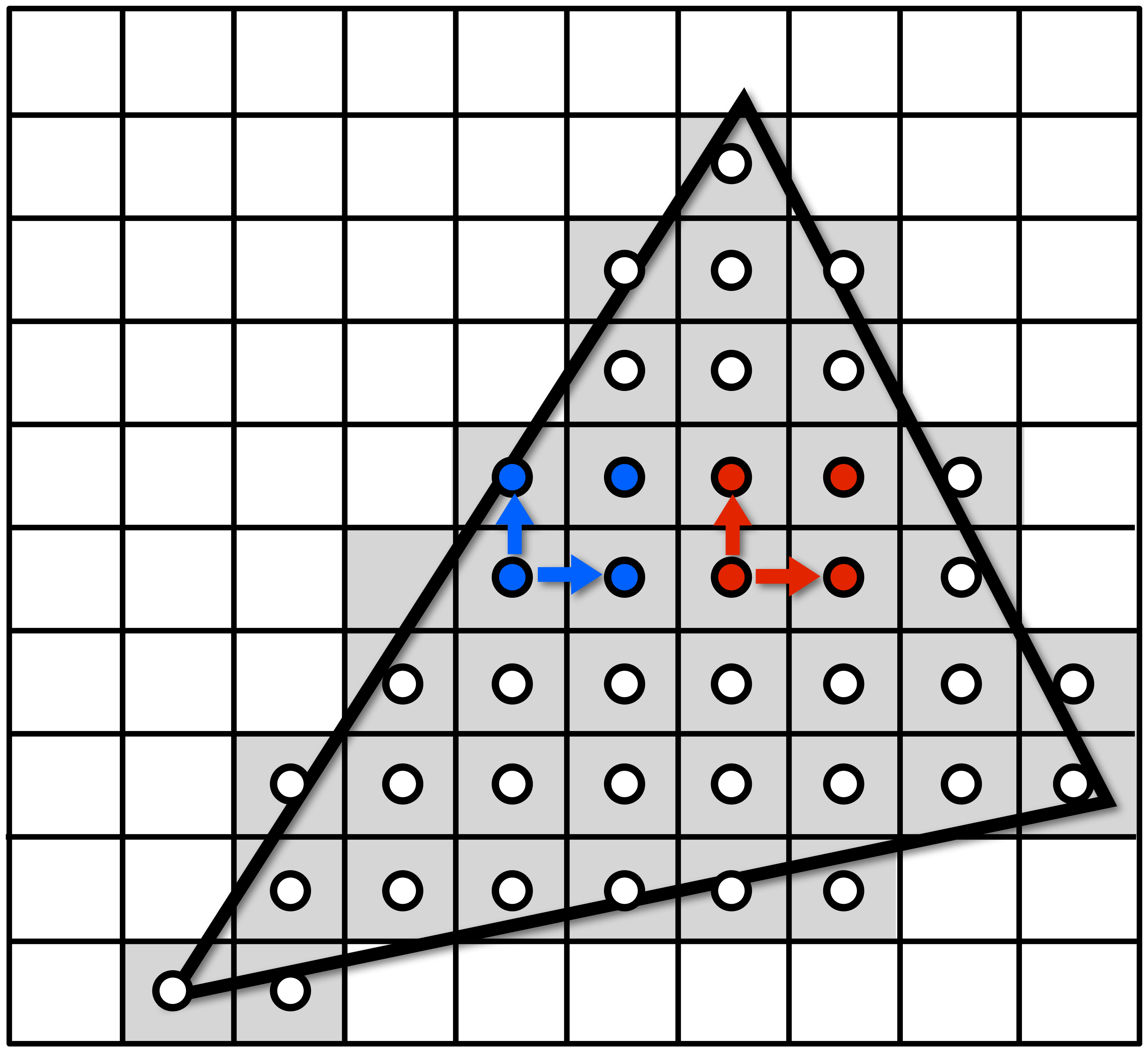
mip-map level *d+1* texels



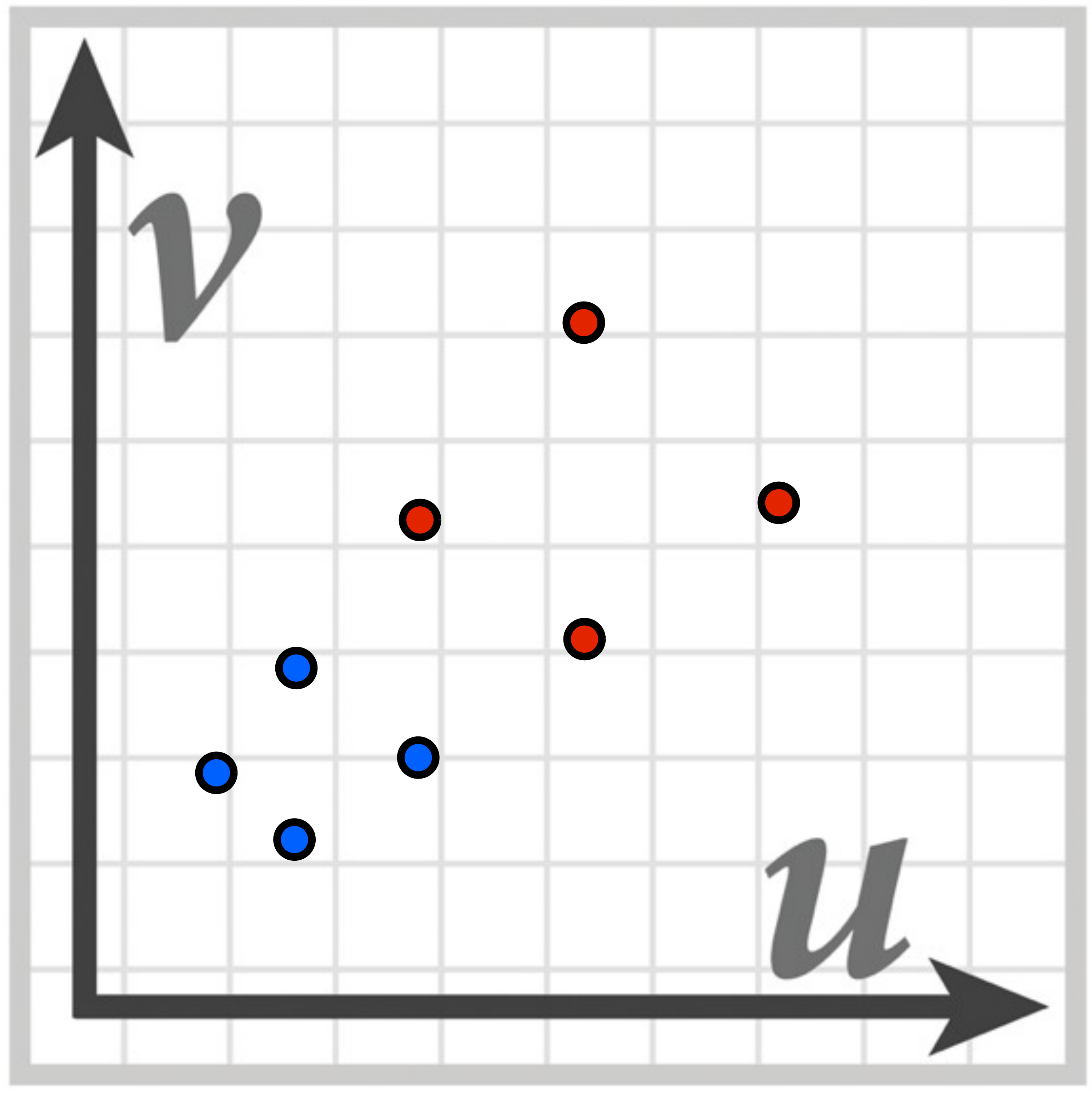
mip-map level *d* texels

Computing d

Take differences between texture coordinate values of neighboring fragments



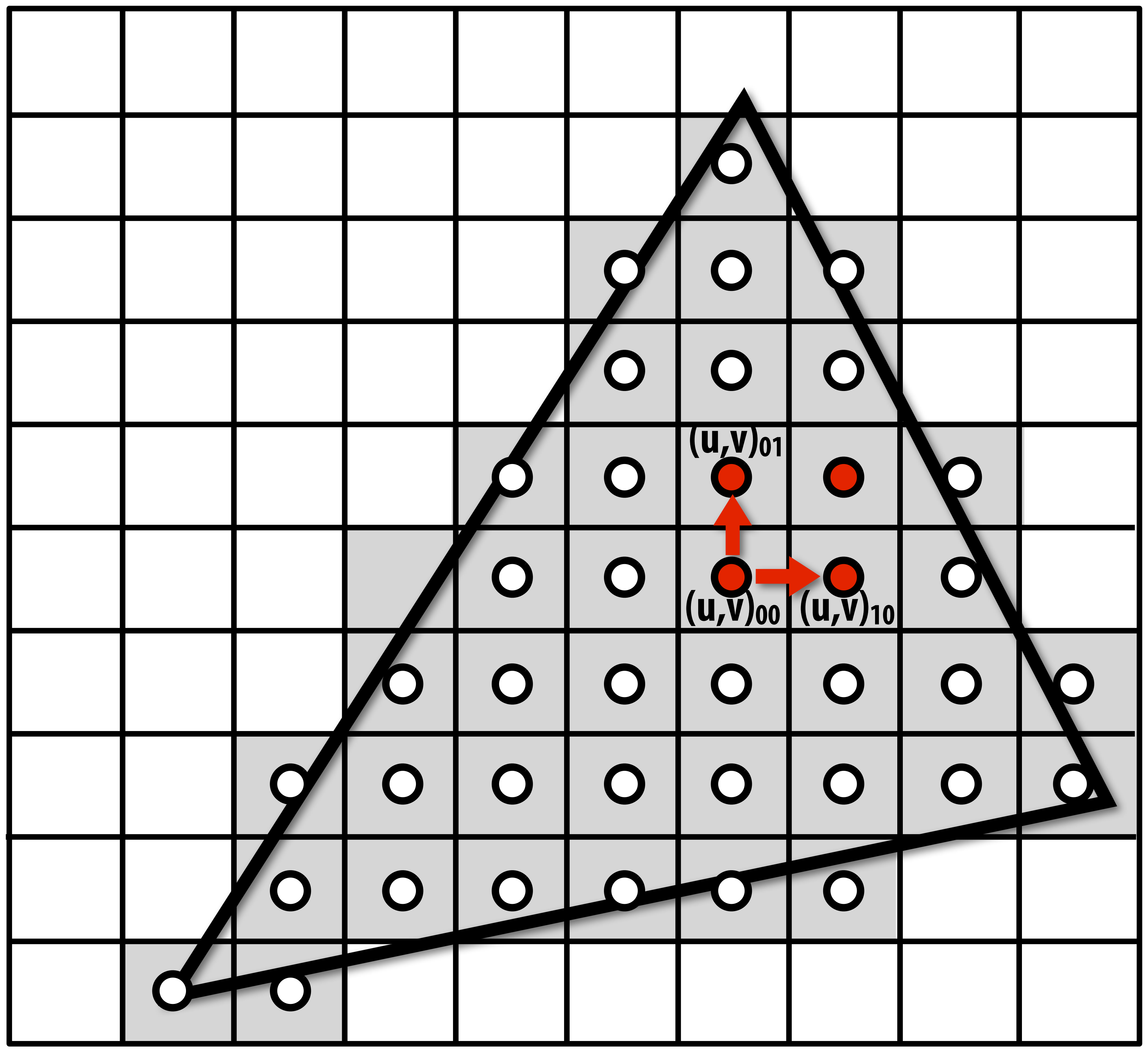
Screen space

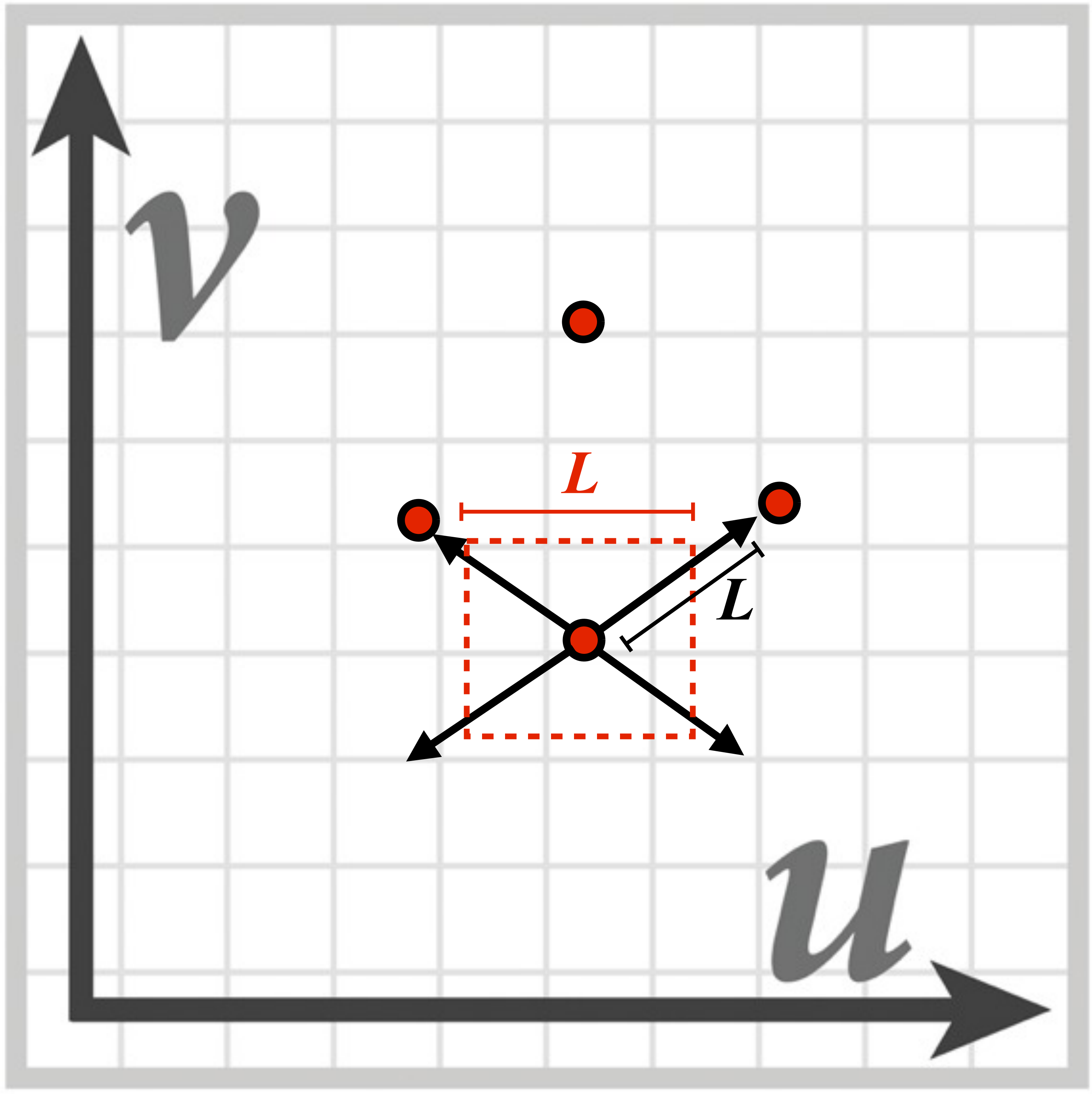


Texture space

Computing d

Take differences between texture coordinate values of neighboring fragments





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

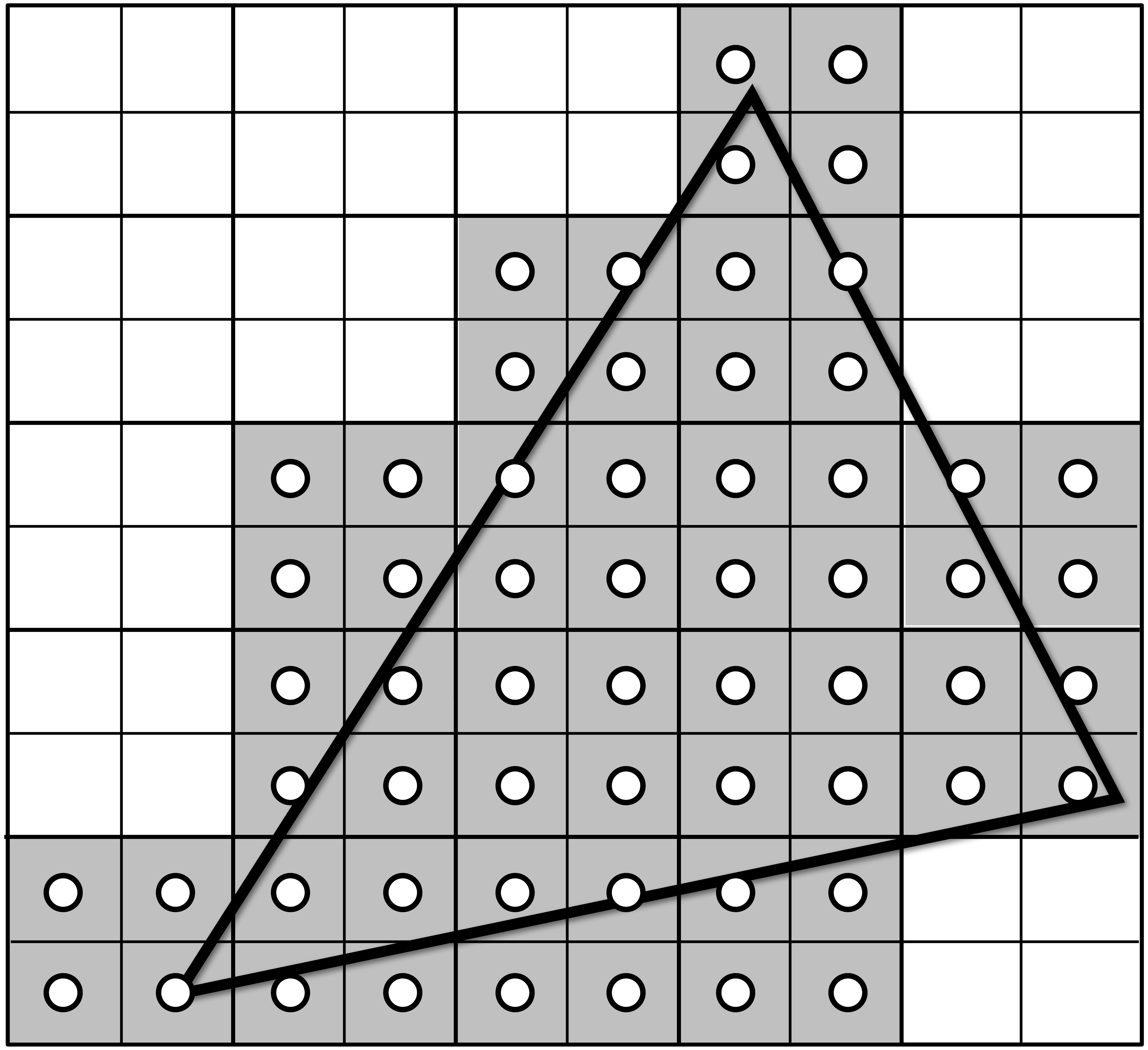
$du/dy = u_{01} - u_{00}$ $dv/dy = v_{01} - v_{00}$

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

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

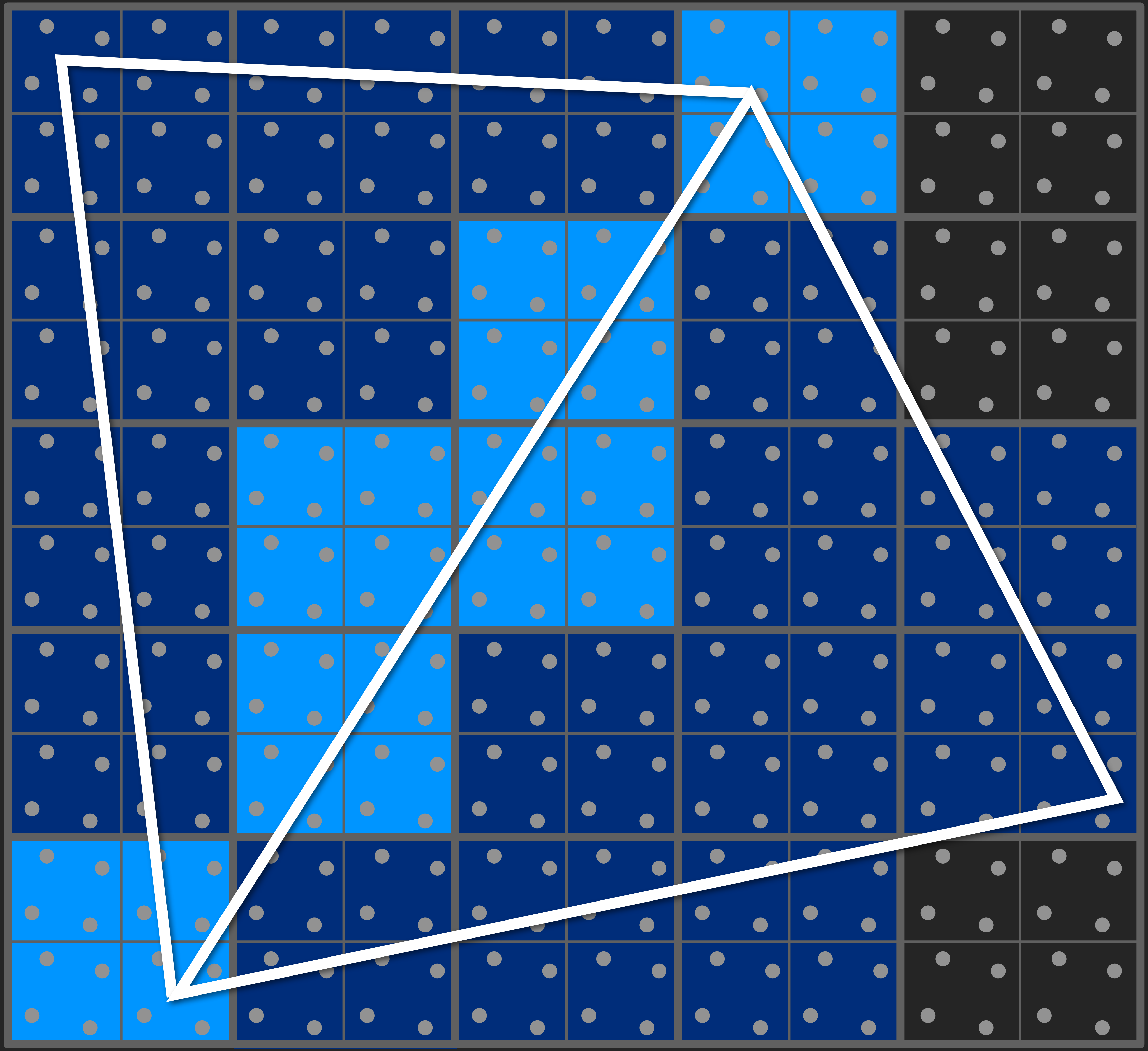
GPUs shade quad fragments (2x2 fragment blocks)

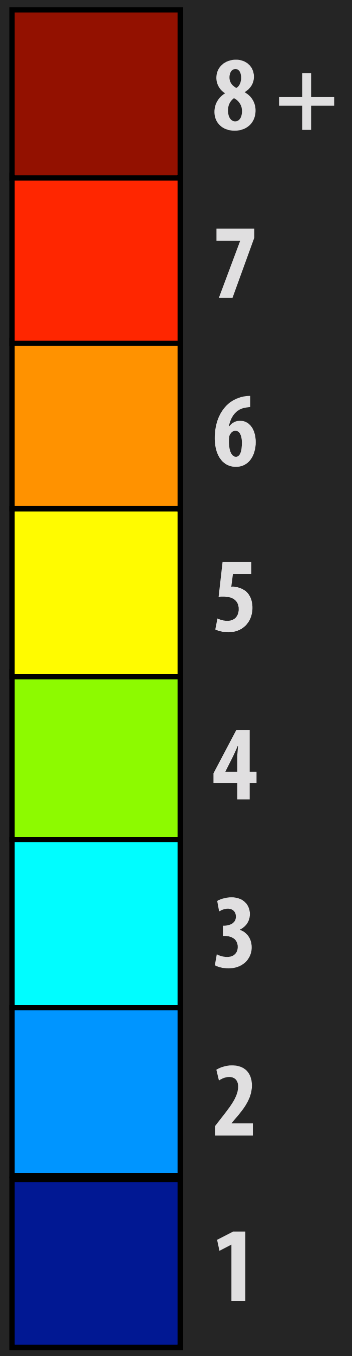
Cheap, simple texture coordinate differentials + avoids communication



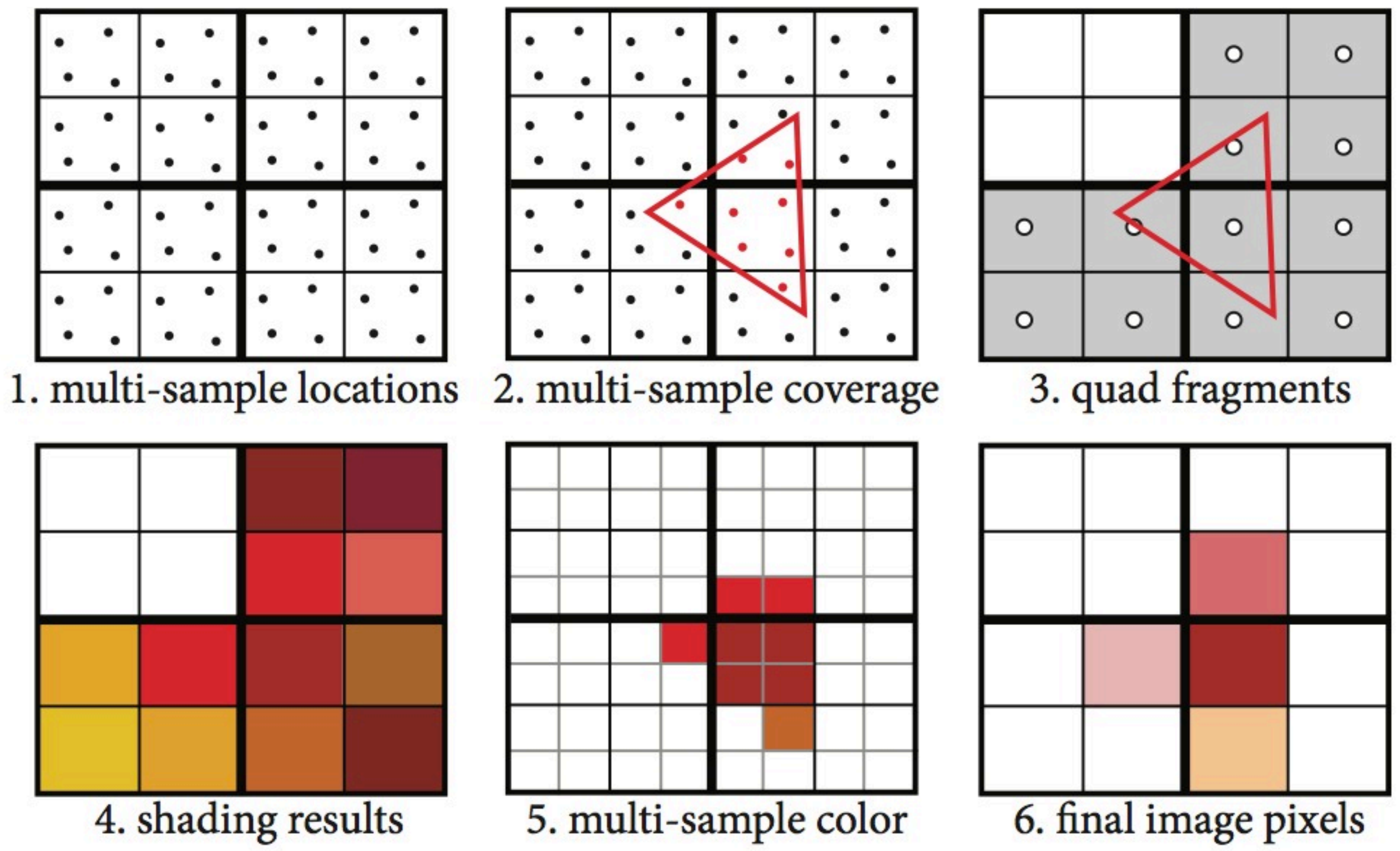
Multiple fragments shaded for pixels at triangle boundaries

Shading computations per pixel





Quick aside: multi-sample AA (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**

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 ...

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.

Filtering with mip mapping summary

- Extra storage: 1/3 of original texture image
- For each texture filtering request
  - Constant filtering cost (independent of d)
  - Constant # texels accessed (independent of d)
- Pretty good quality
  - Assumption: isotropic filtering of texture function
  - Many improvements to handle anisotropic filtering (exist in current GPUs)
    - Higher quality, but greater compute and memory bandwidth cost

Texture mechanics

For each texture fetch in a shader program:

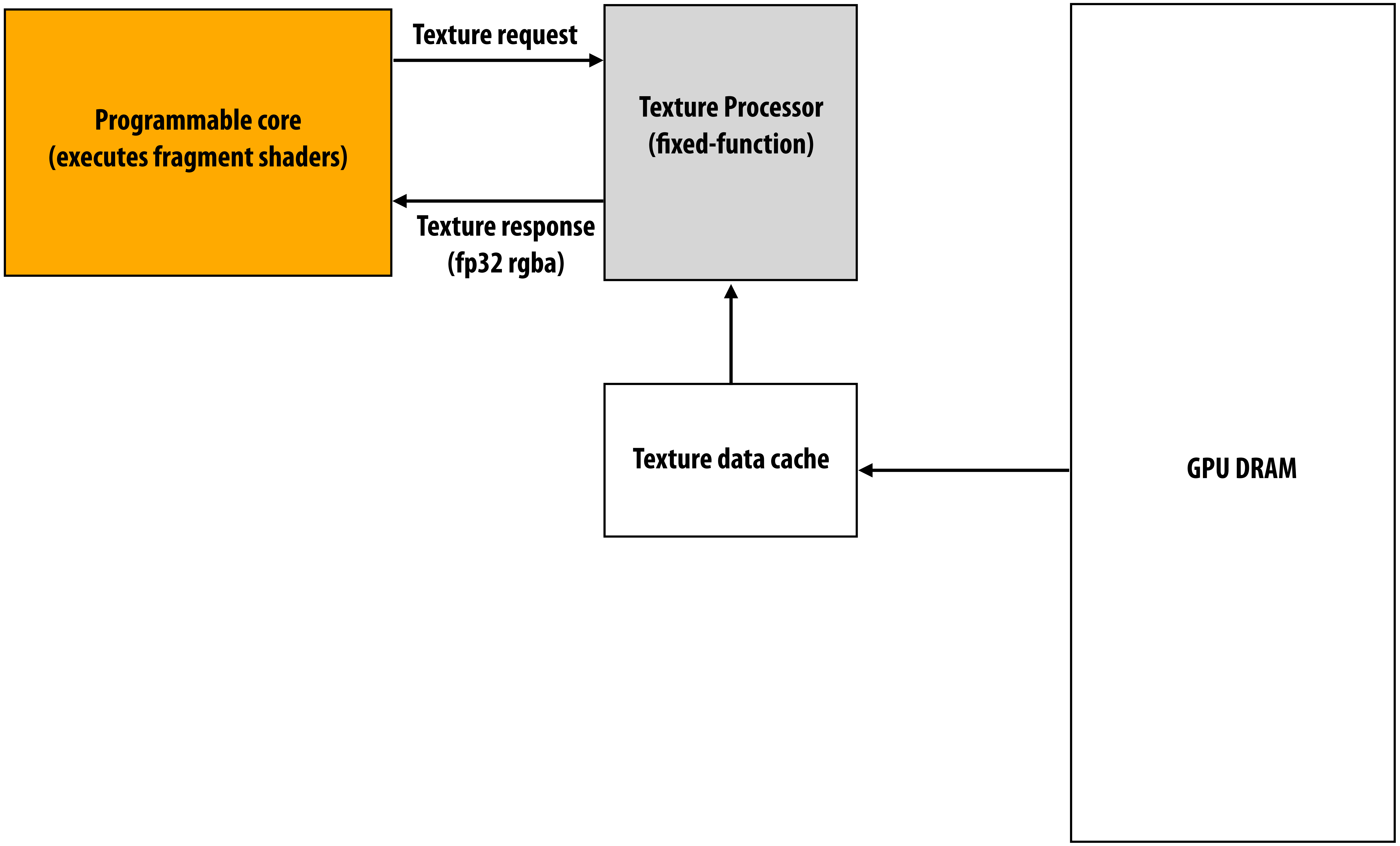
1. **Compute du/dx, du/dy, dv/dx, dv/dy differentials from quad fragment**
2. **Convert normalized values to texel coordinates**
3. **Compute d**
4. **Compute required texels ****
5. **Load texture data ******
6. **Result = tri-linear filter according to (u,v,d)**

Lots of math: All modern GPUs have sophisticated fixed-function hardware for texture processing! (note: even Larrabee design had texturing hardware)

**** may involve wrap, clamp, etc. of u,v values according to sampling mode configuration**

****** may require decompression**

Texture block diagram



Consider memory

- **Texture 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
  - Latency hiding solution
  - Texture compression

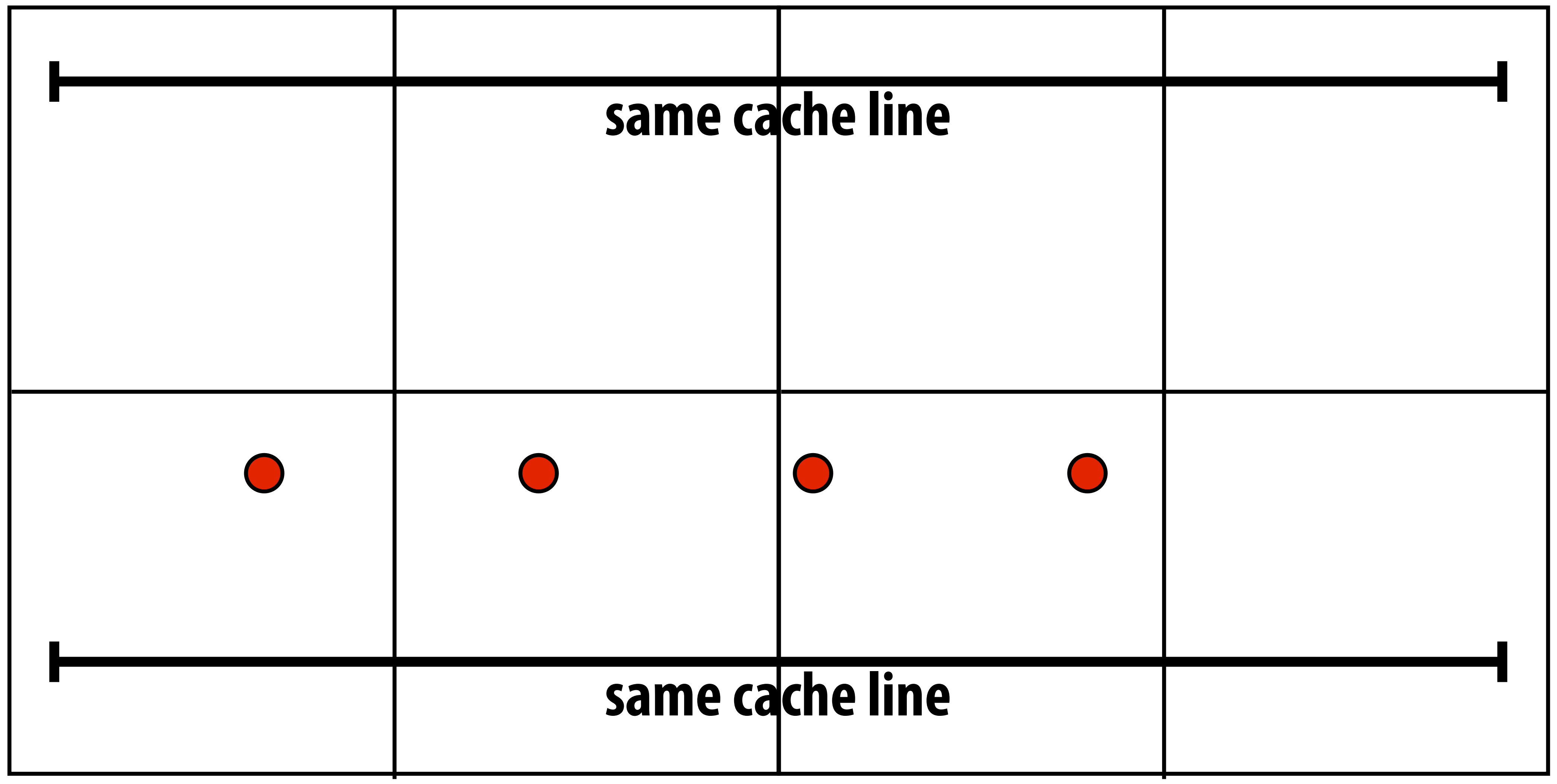
Review: the role of caches in CPUs

- 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)

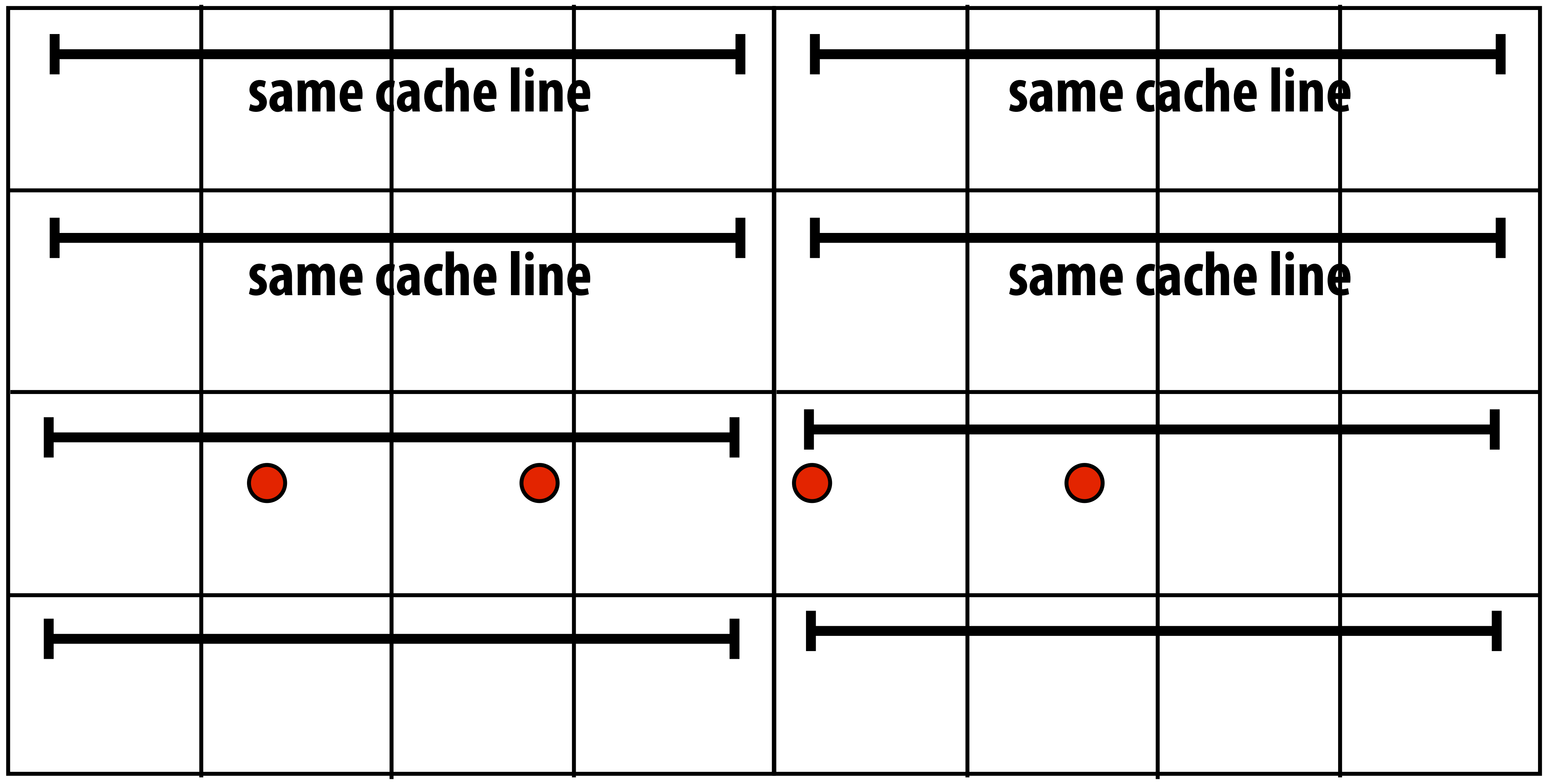
- Reduce latency of data access

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

Texture caching thought experiment



mip-map level *d+1* texels



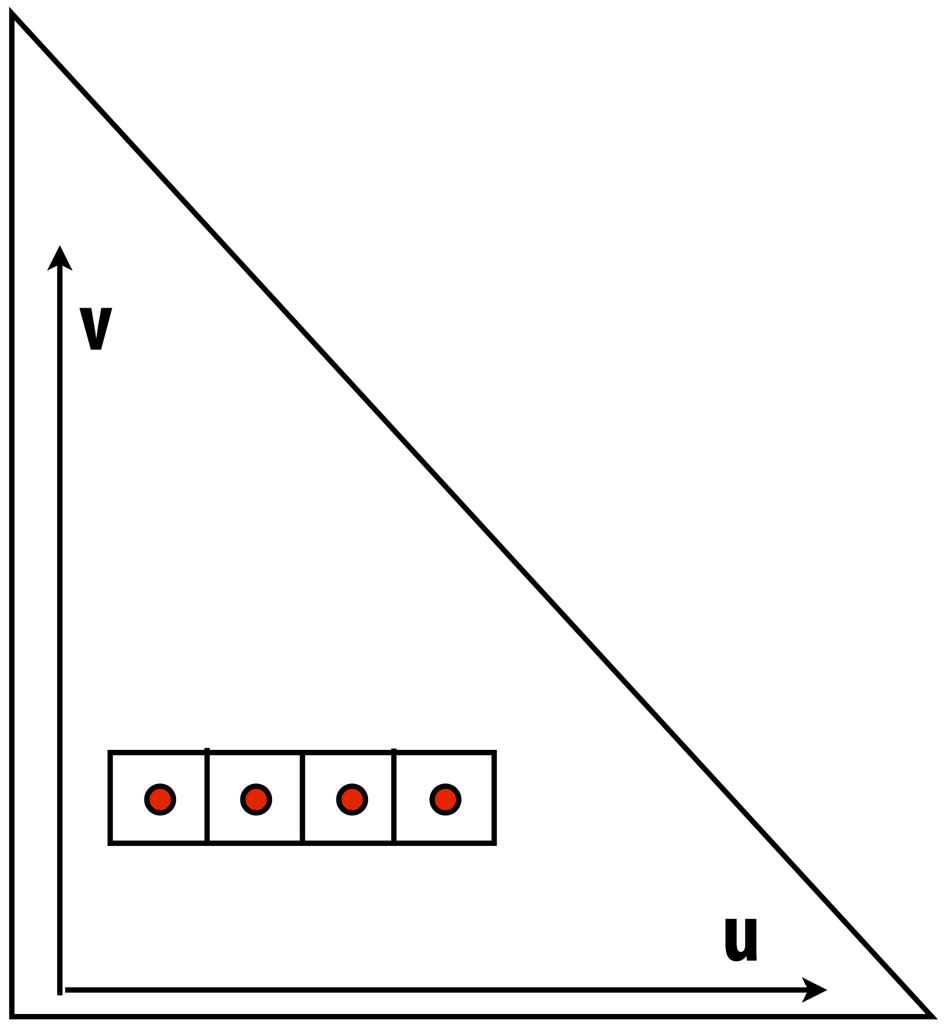
mip-map level *d* texels

Assume:

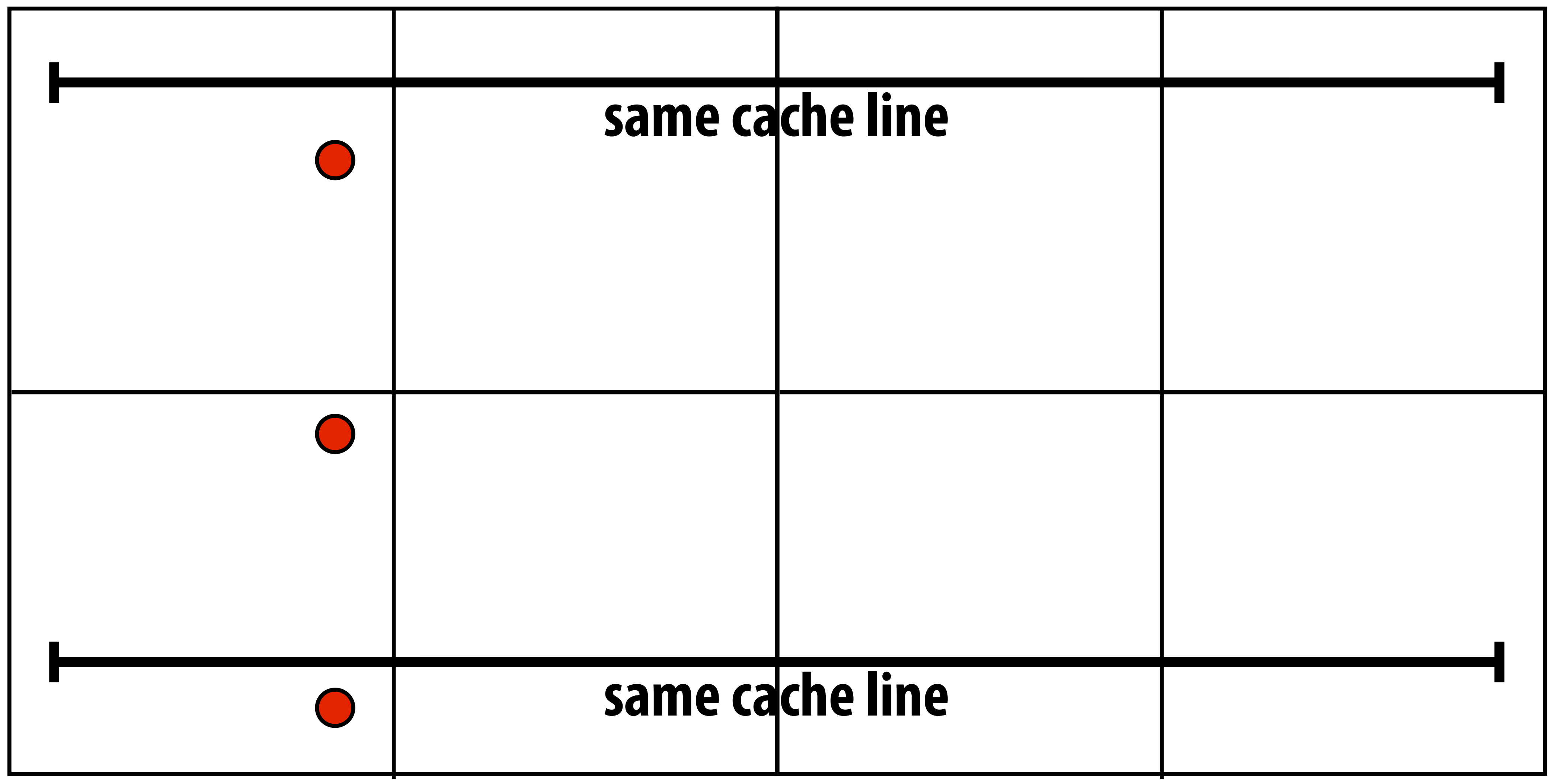
Row-major raster order

Horizontal texels contiguous in memory

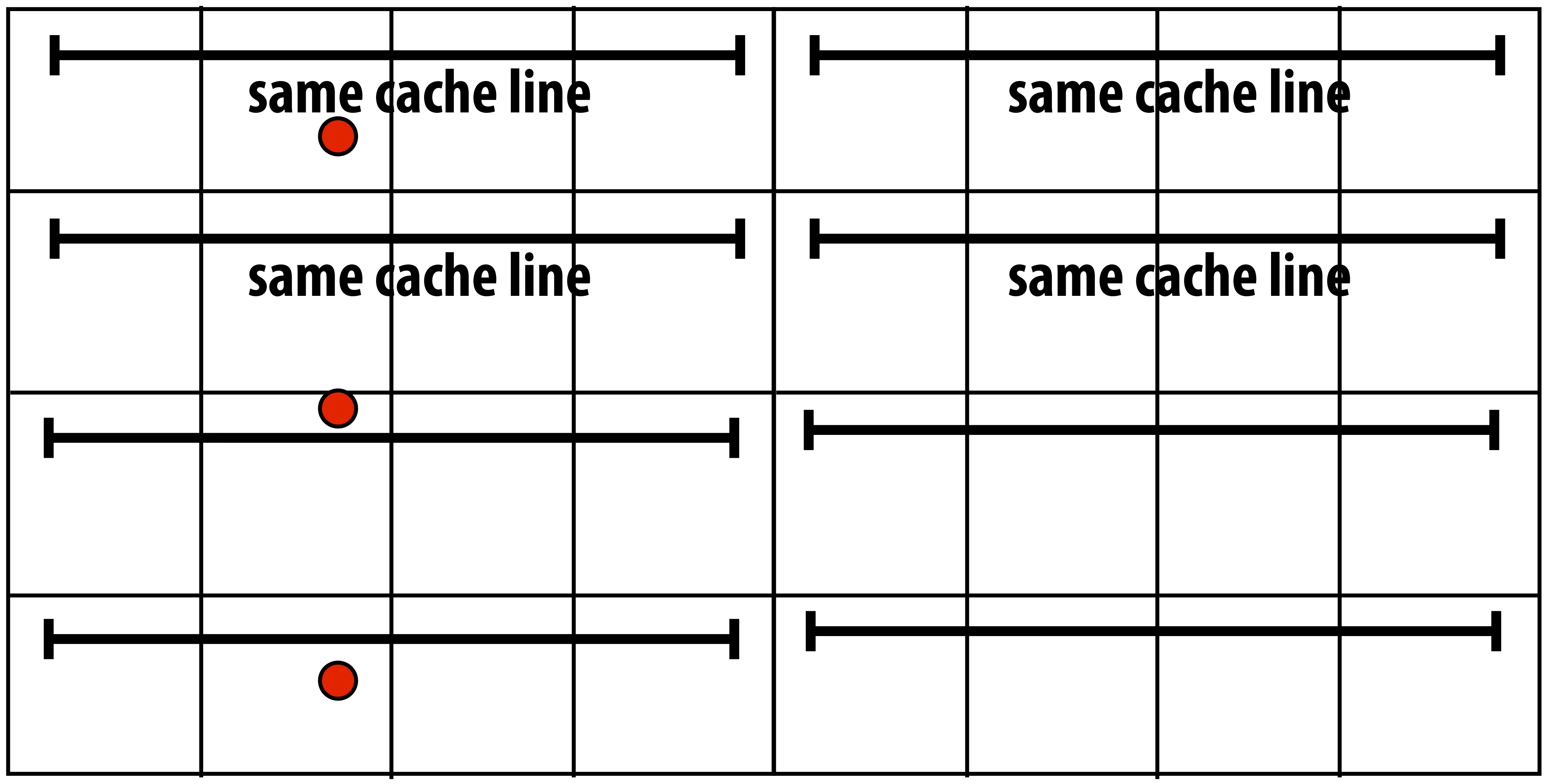
Cache line = 4 texels



Now rotate triangle on screen



mip-map level *d+1* texels



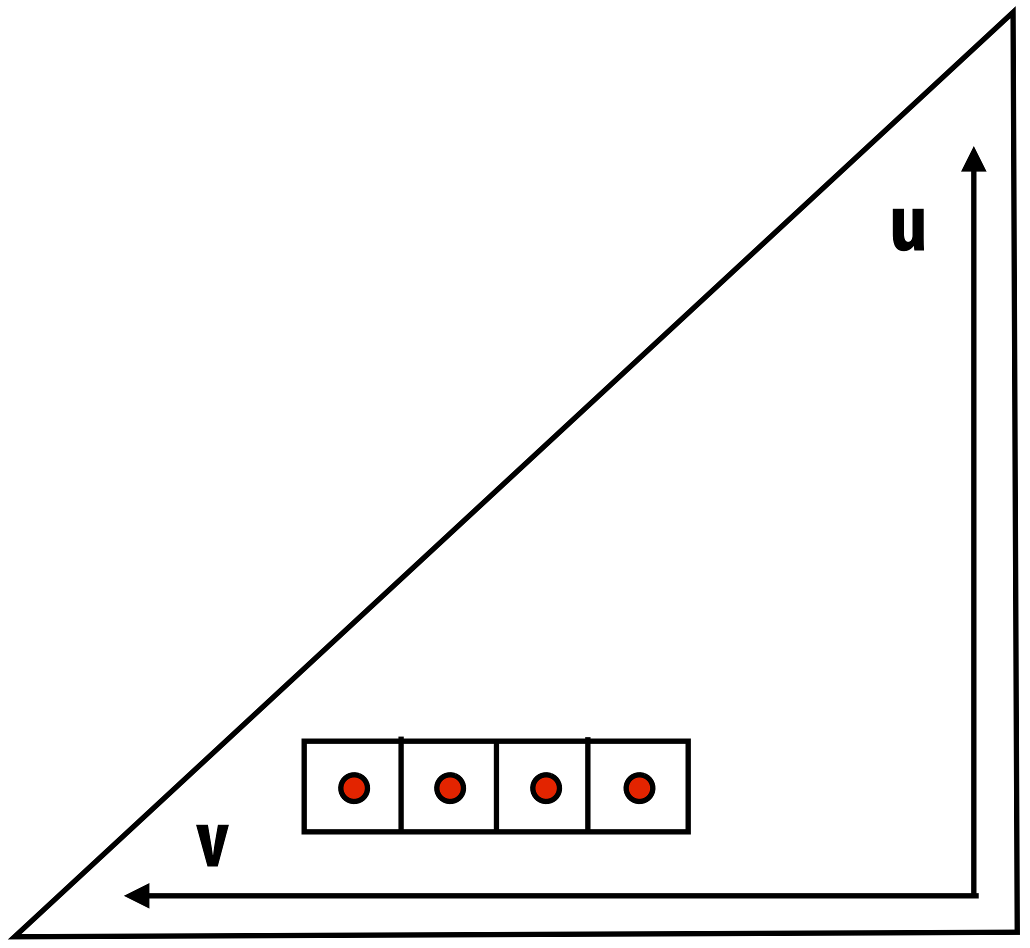
mip-map level *d* texels

Assume:

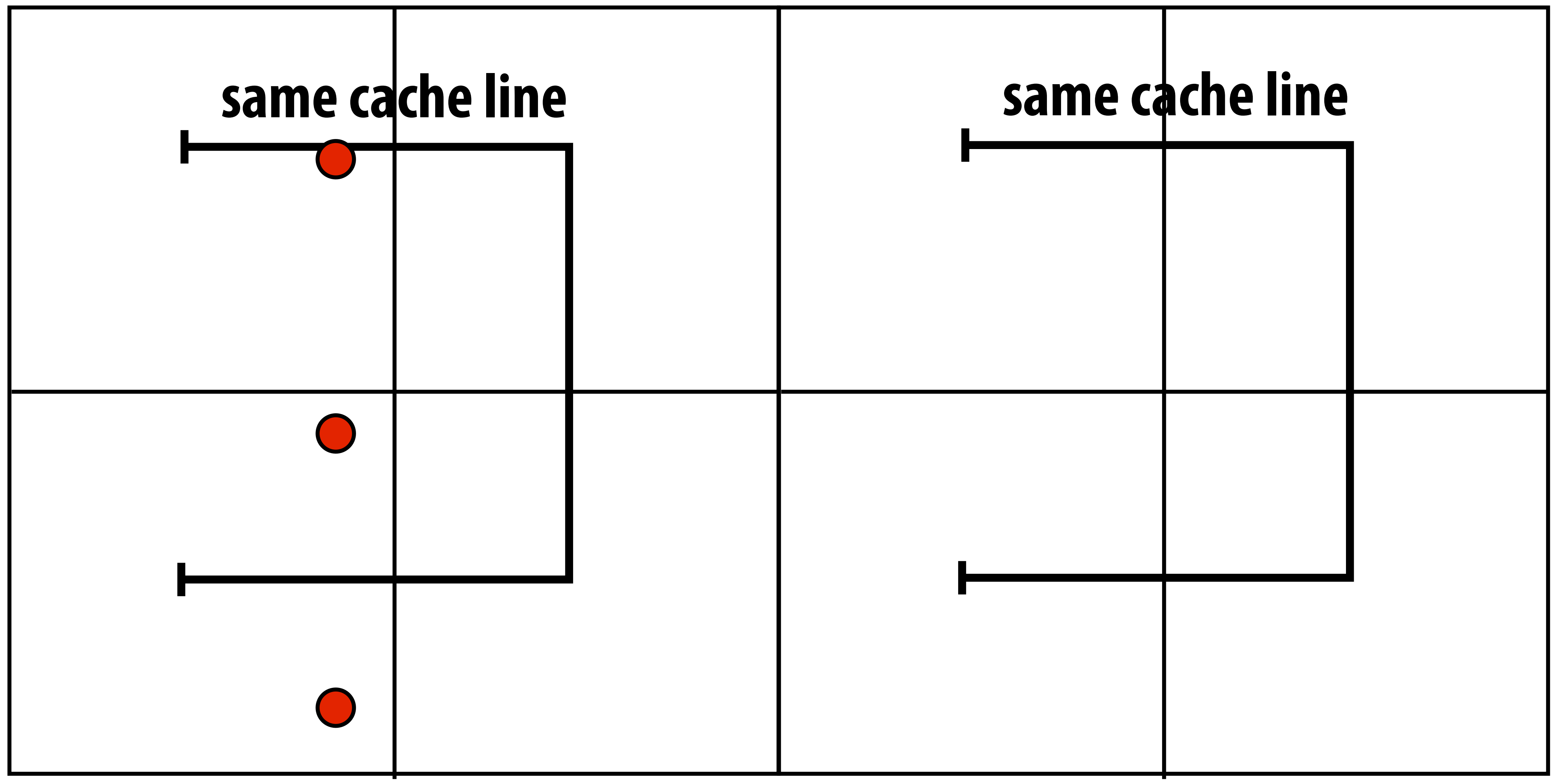
Row-major raster order

Horizontal texels contiguous in memory

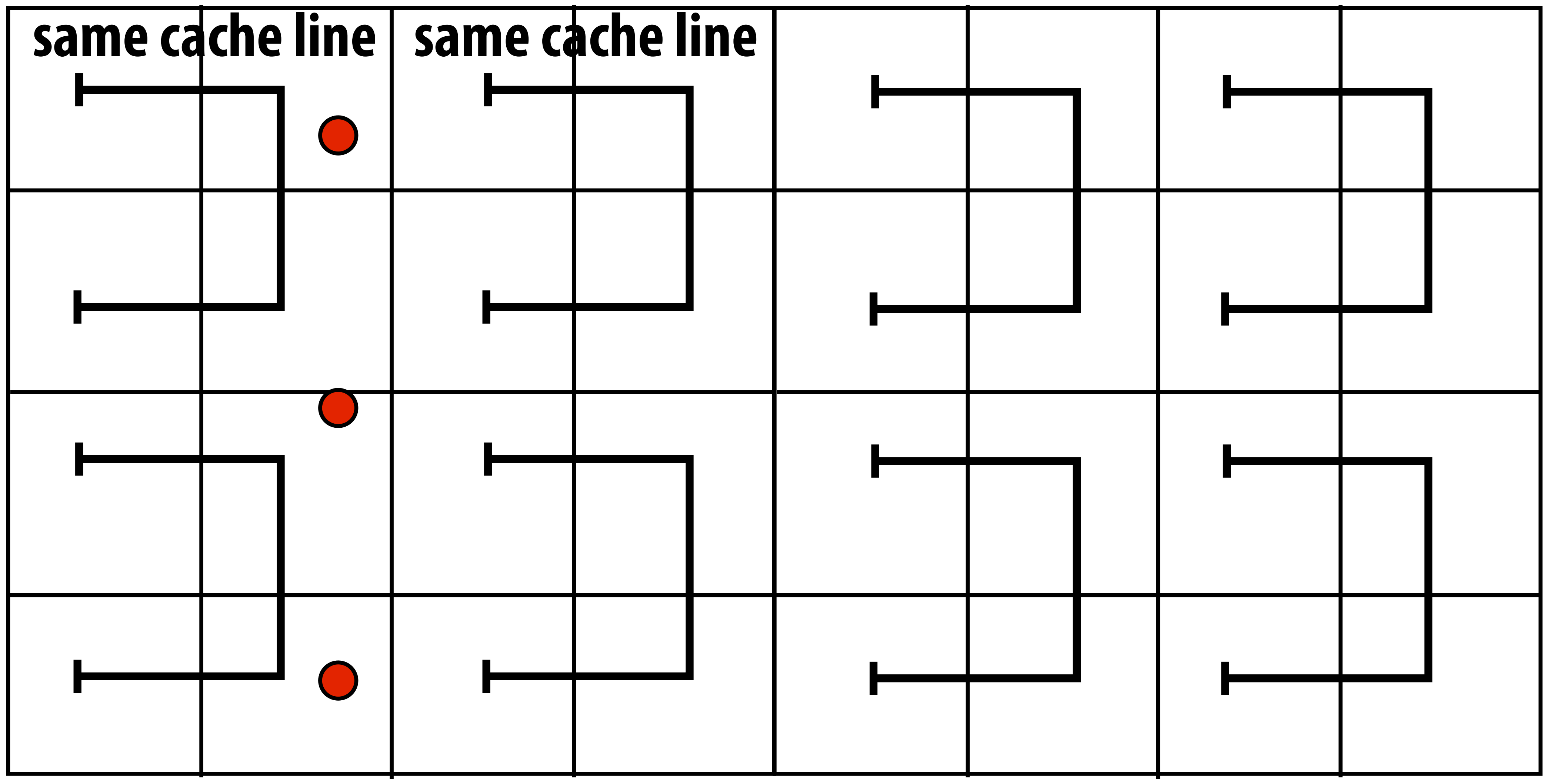
Cache line = 4 texels



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



mip-map level *d+1* texels



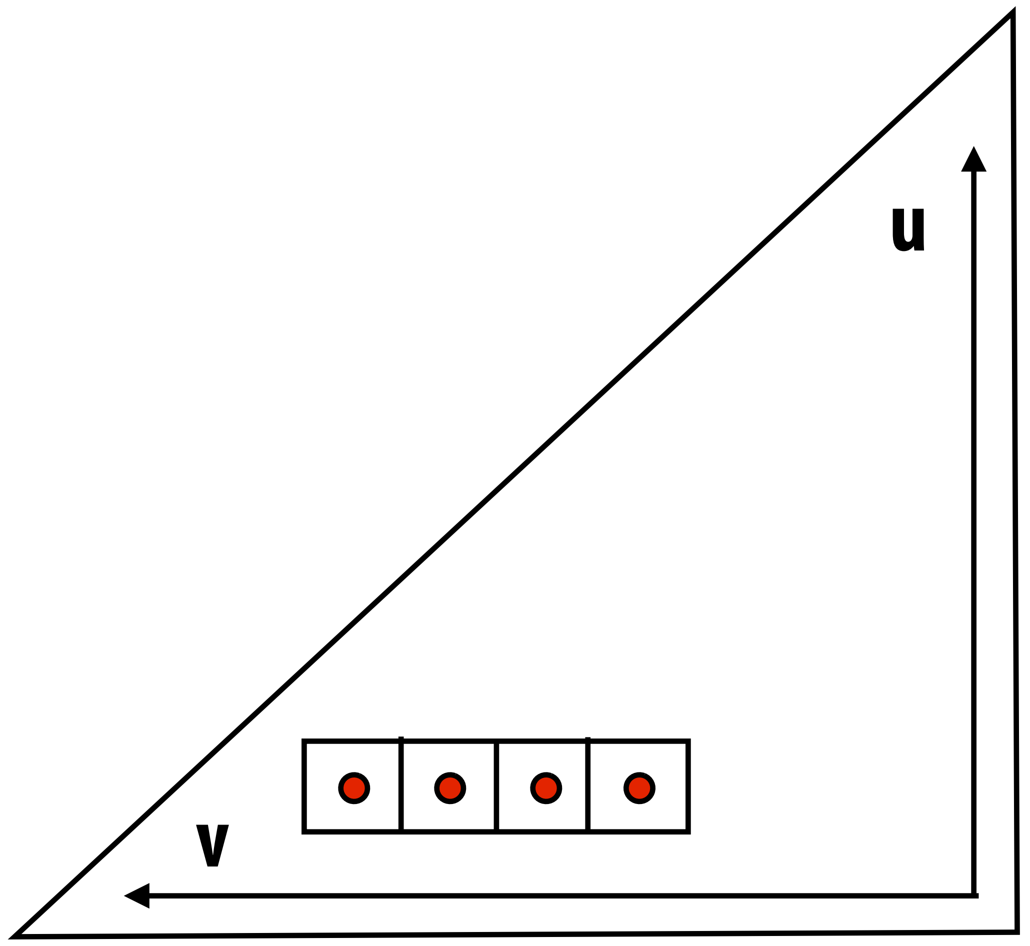
mip-map level *d* texels

Assume:

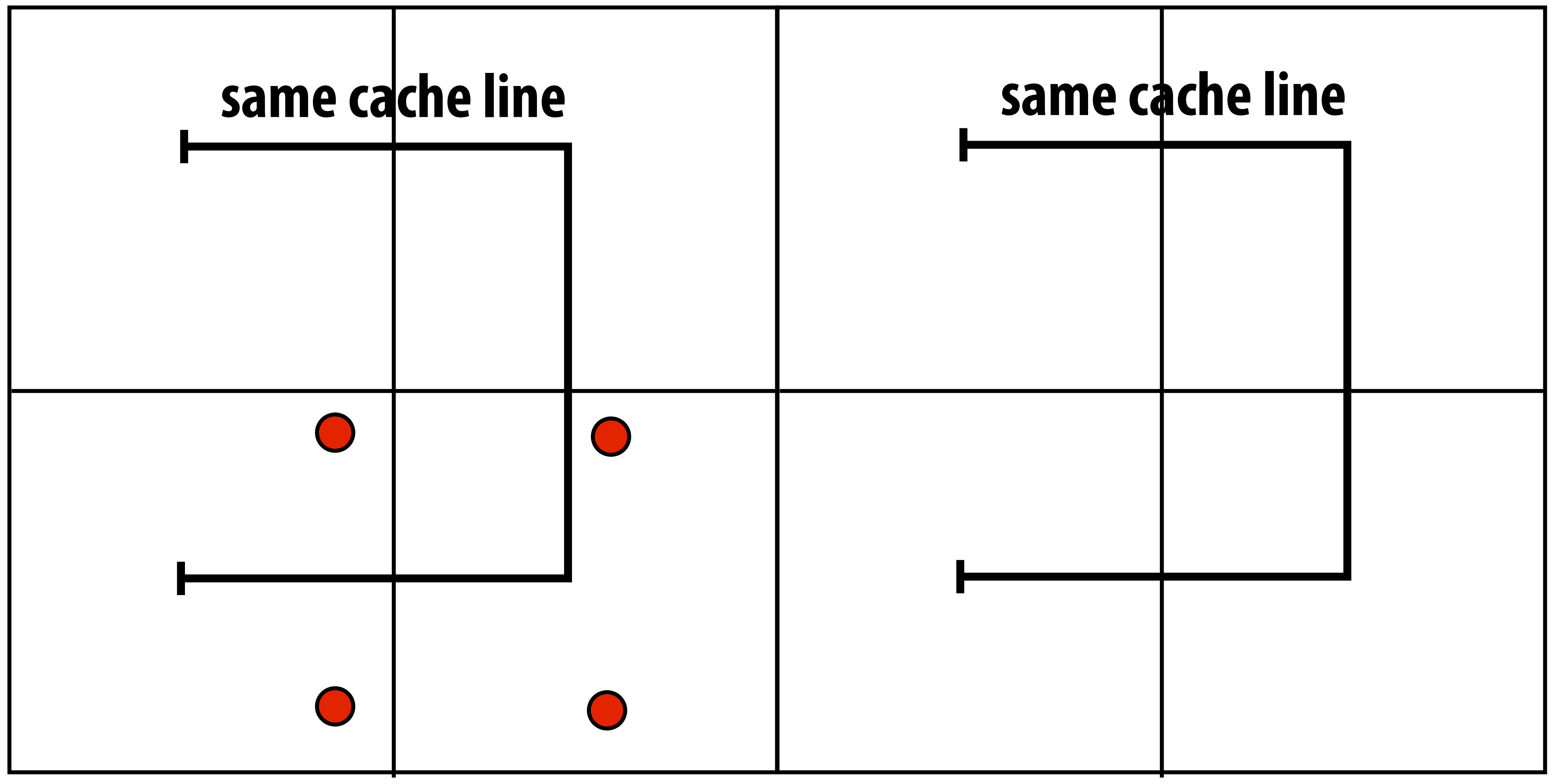
Row-major raster order

2D blocks of texels contiguous in memory

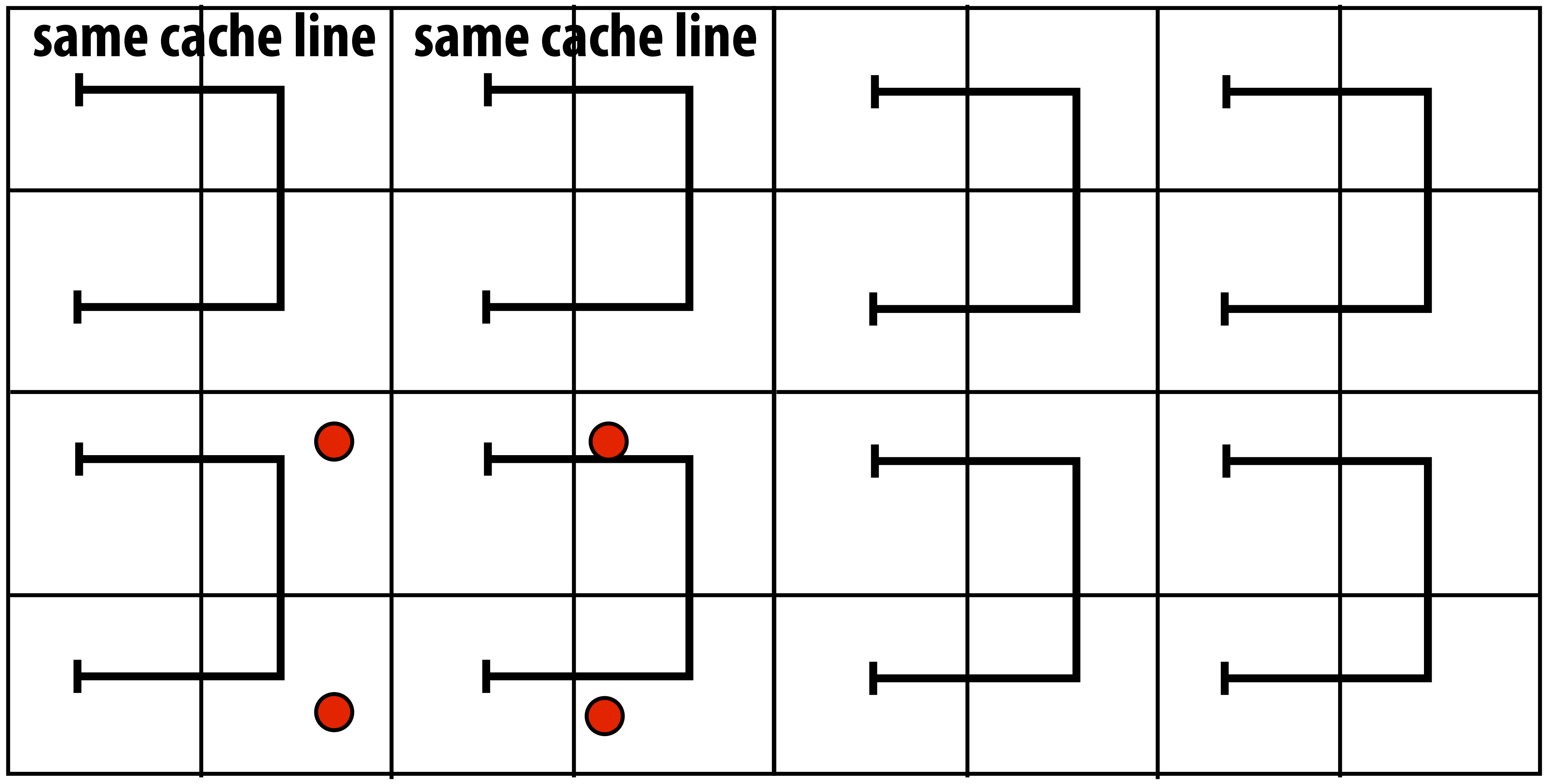
Cache line = 4 texels



Tiled rasterization increases reuse



mip-map level *d+1* texels



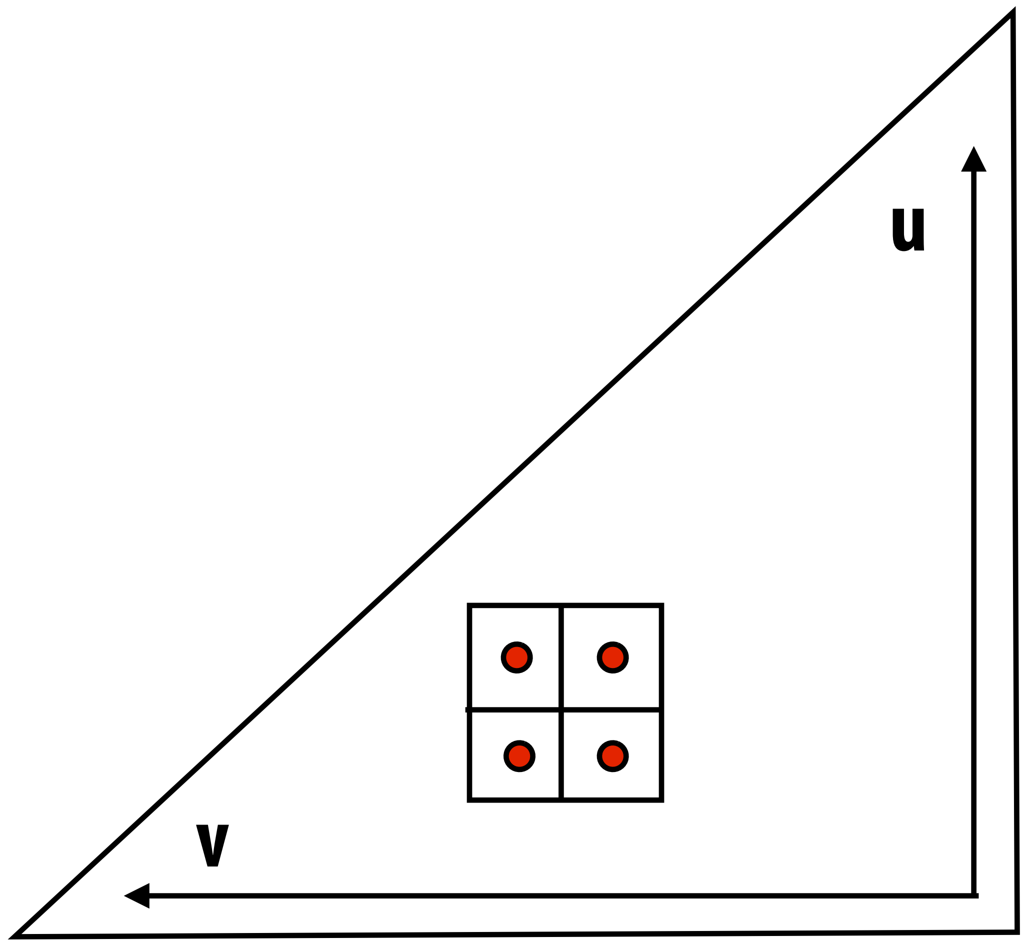
mip-map level *d* texels

Assume:

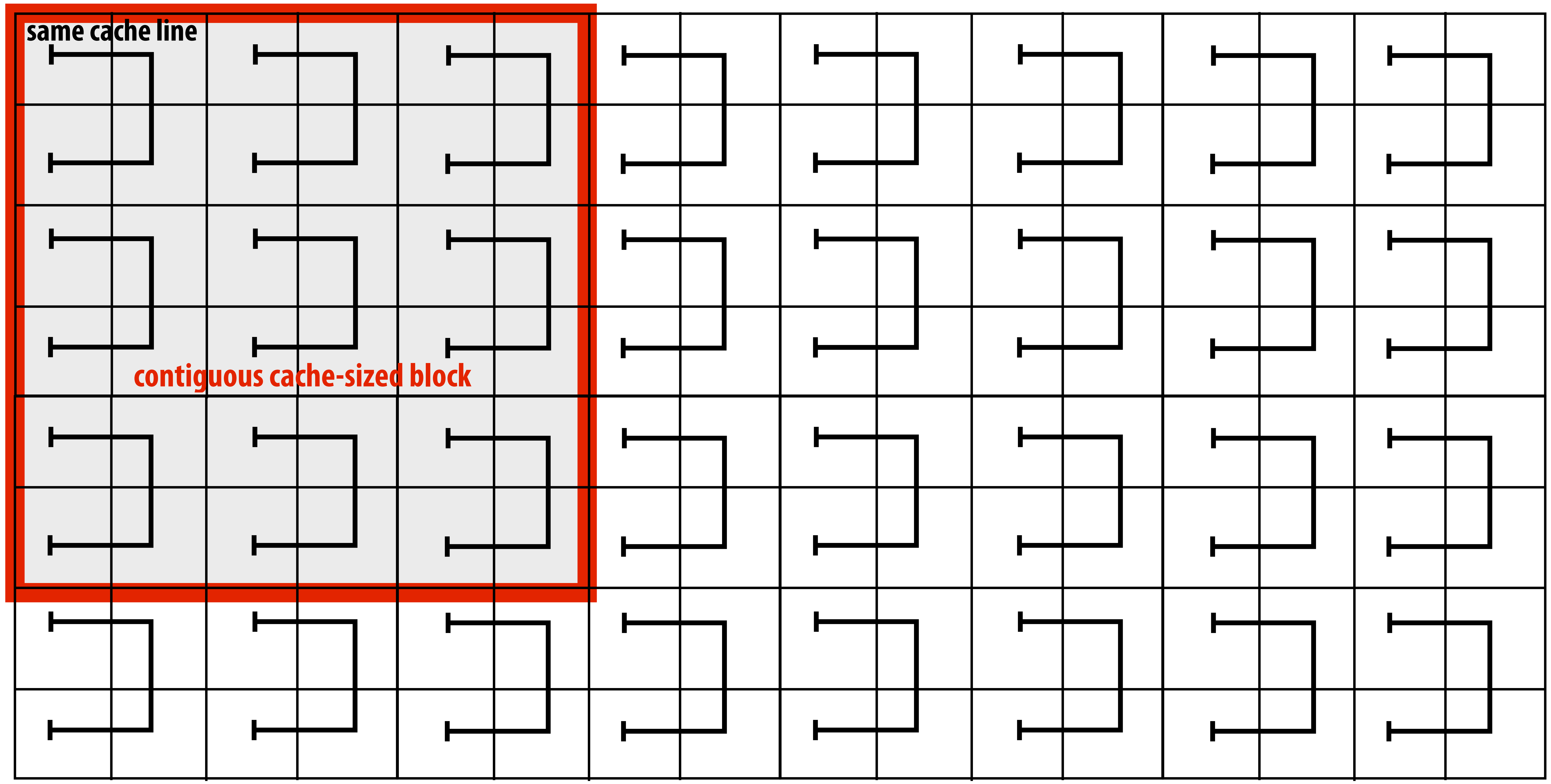
Blocked raster order

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 had a preferred format
  - Textures had a preferred format
  - Costly to convert between the two

- These days:
  - Declare usage for buffers at allocation in API
  - In general, standard blocking schemes across the board

Key metric

- Unique texel to fragment ratio
  - Number of unique texels accessed when rendering a scene / number of fragments processed [see Igeny reading for stats: often less than < 1]
  - Worse case?

- In reality, caching behavior is good, but not CPU workload good
  - Design for 90% hits [Montrym & Moreton 95]
  - GPUs require high memory bandwidth

Memory latency

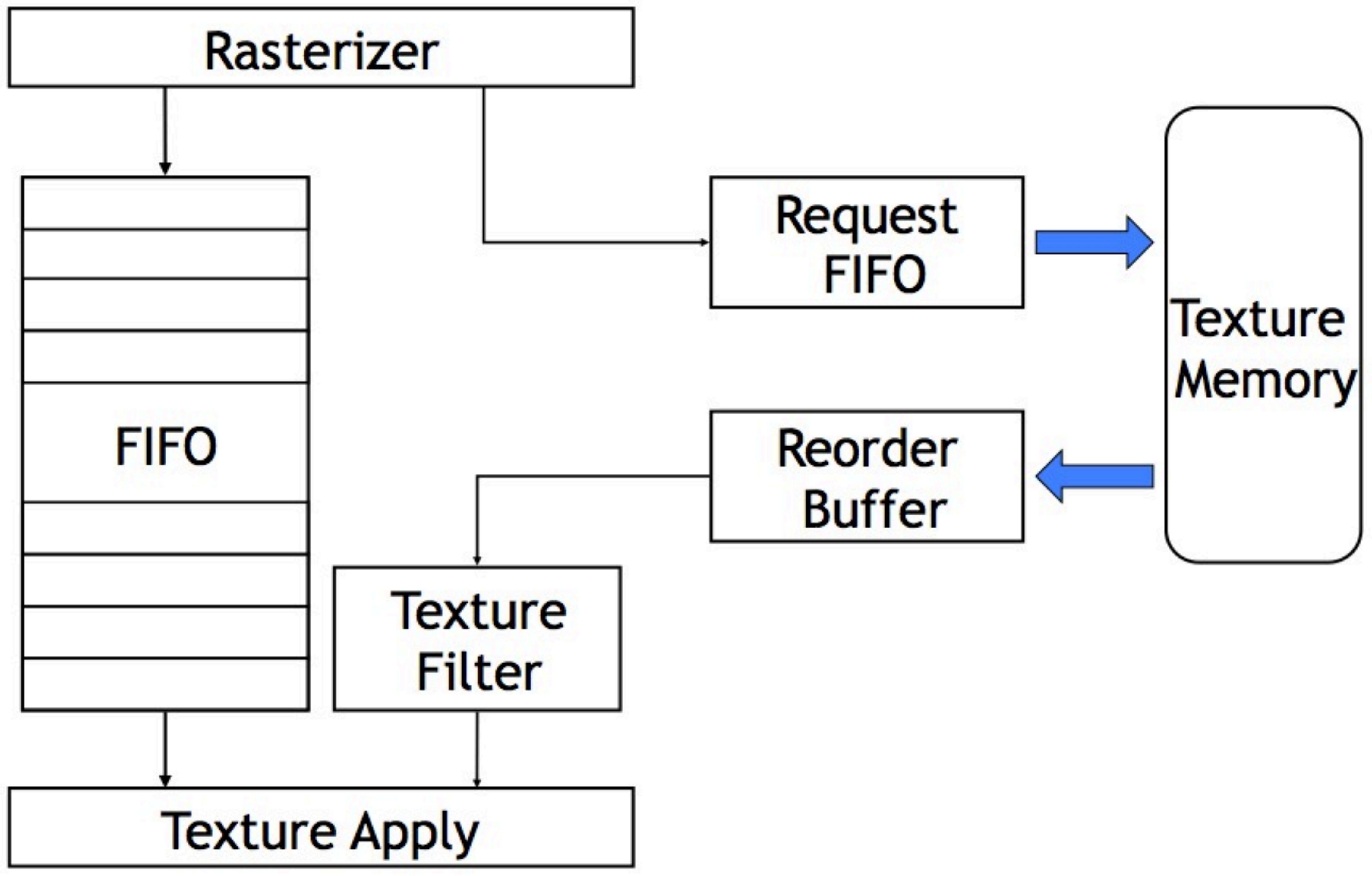
- Request texture data. Processor waits for hundreds of cycles. (Very bad)

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

- Solution prior to programmable shading: prefetch
  - Today's reading: Igehy et al. *Prefetching in a Texture Cache Architecture*

- Solution in modern programmable GPUs: thread
  - Next time

Large fragment FIFOs



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

Texture summary

- Pre-filtering texture data reduces aliasing
  - Mip-mapping fundamental to texture system design
- 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 (next time)

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