Lecture 18:

Parallelizing and Optimizing Rasterization on Modern (Mobile) GPUs

---

Interactive Computer Graphics
Stanford CS248, Winter 2020

Q. What is a big concern in ~~mobile~~ all computing?

A. Power

Two reasons to save power

Run at *higher performance* for a *fixed* amount of time. ← Power = heat
If a chip gets too hot, it must be clocked down to cool off

Run at *sufficient performance* for a *longer* amount of time. ← Power = battery
Long battery life is a desirable feature in mobile devices

Mobile phone examples

Samsung Galaxy s9

11.5 Watt hours

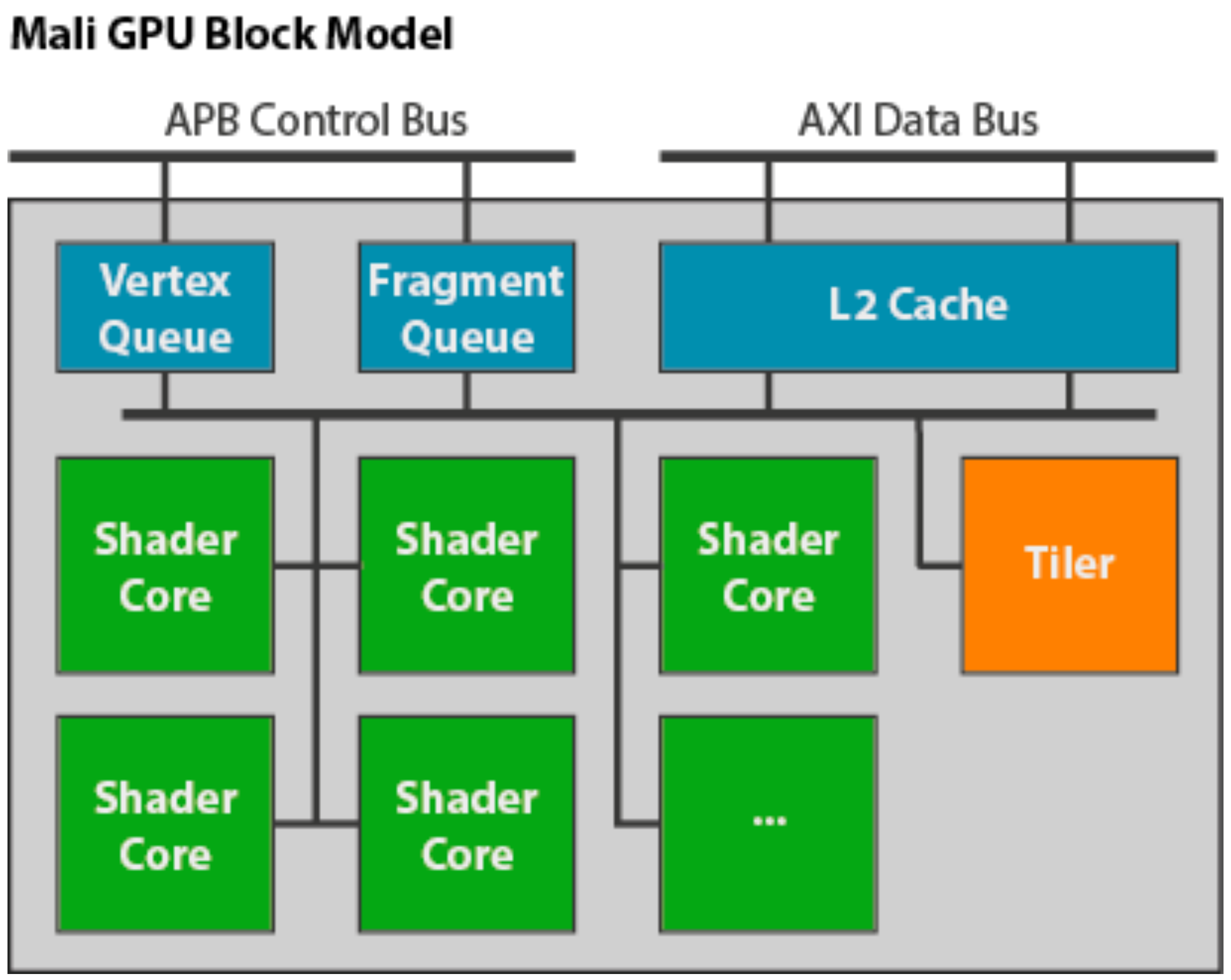
Apple iPhone 8

7 Watt hours

Graphics processors (GPUs) in these mobile phones

Samsung Galaxy s9
(non US version)

ARM Mali G72MP18



Apple iPhone 8

Custom Apple GPU in A11 Processor

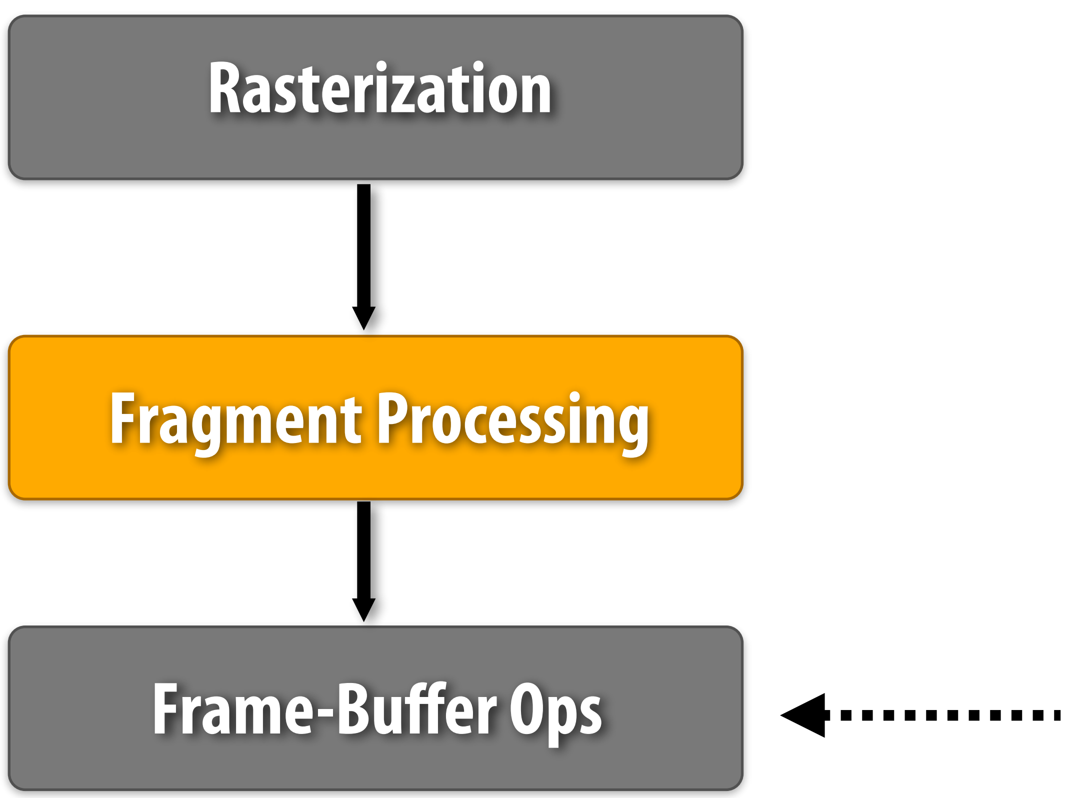
Ways to conserve power

- **Compute less**
  - **Reduce the amount of work required to render a picture**
  - **Less computation = less power**

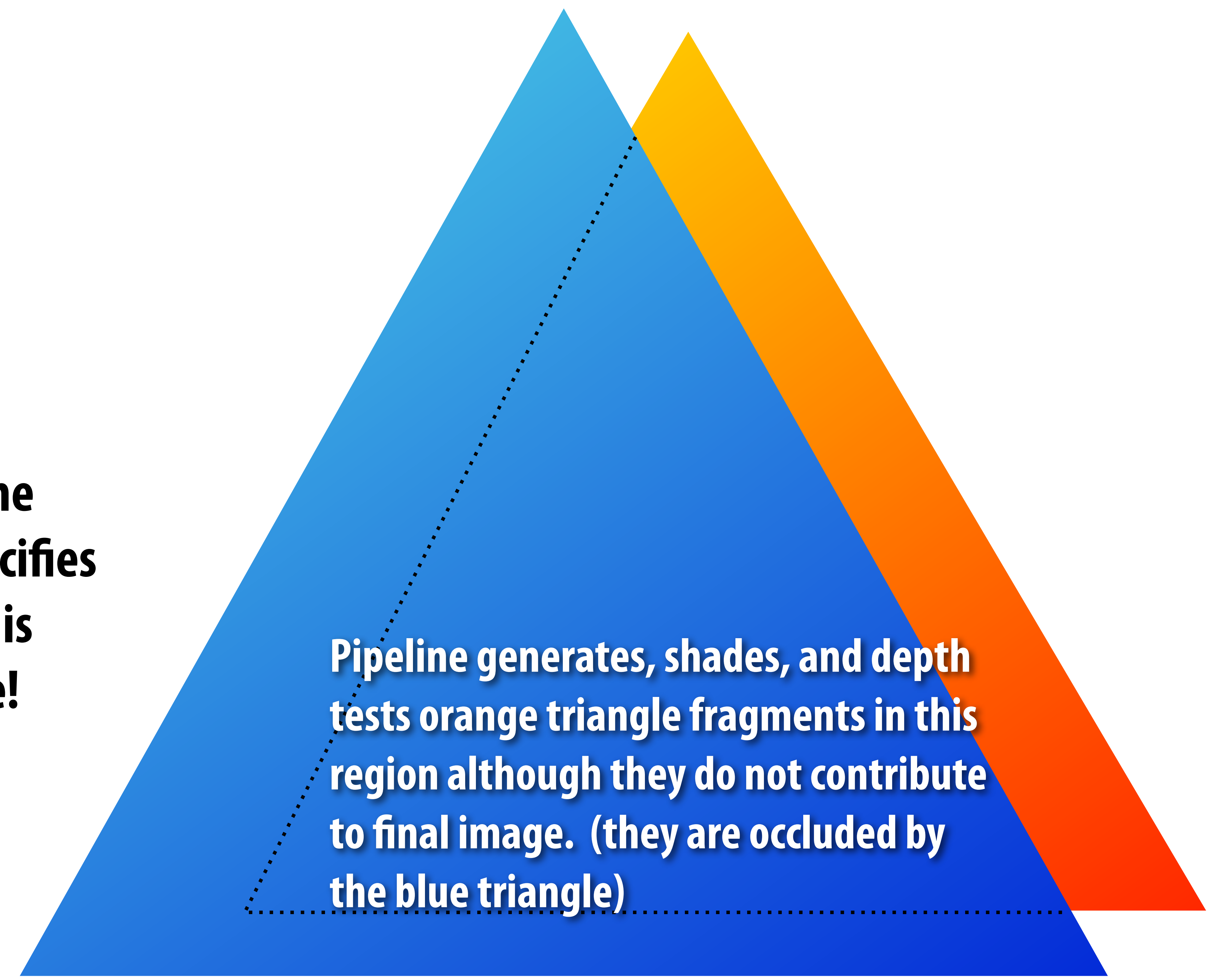
- **Read less data**
  - **Data movement has high energy cost**

Early depth culling (“Early Z”)

Depth testing as we've described it

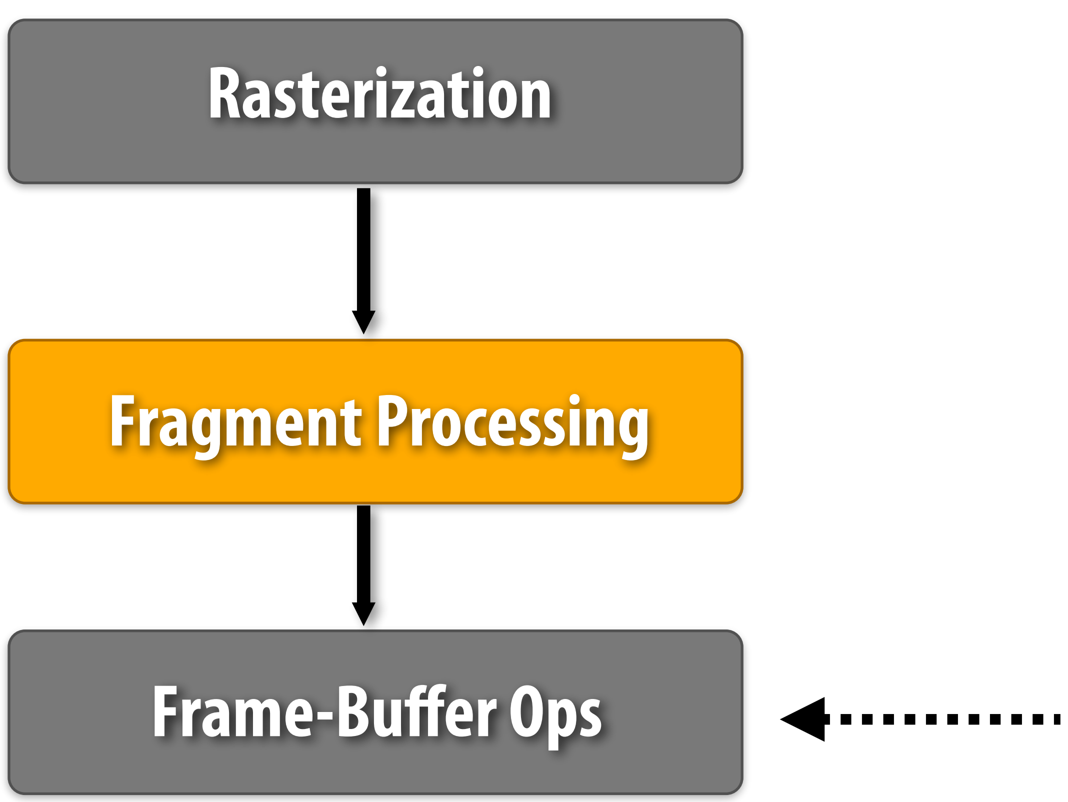


Graphics pipeline abstraction specifies that depth test is performed here!

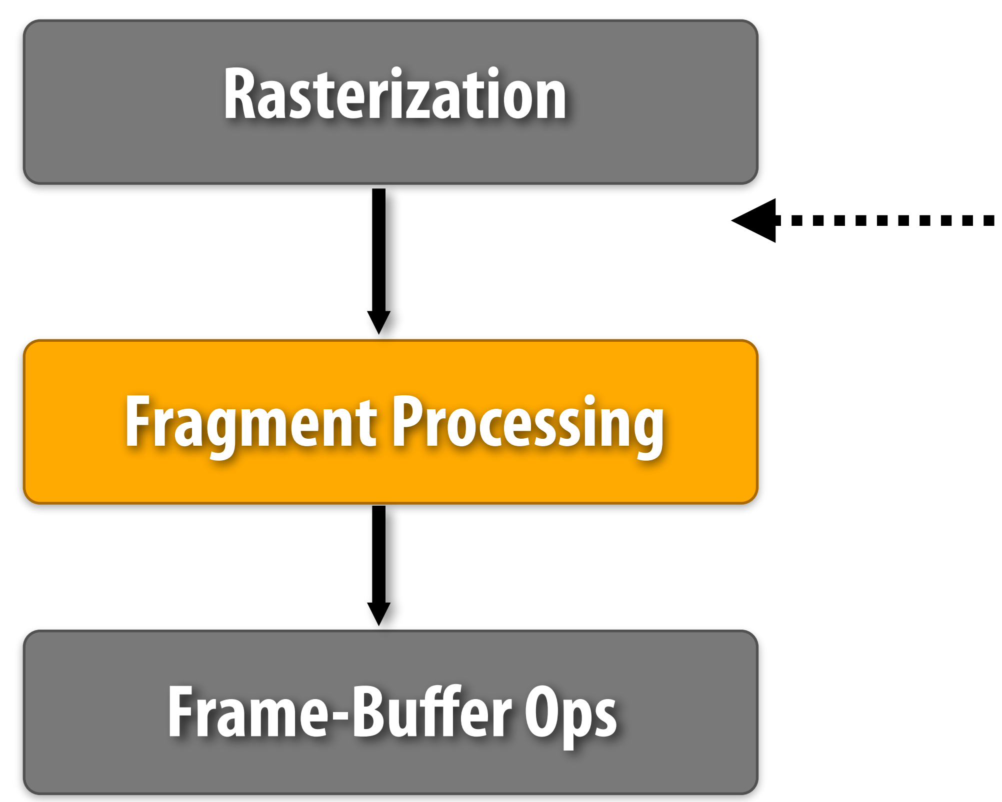


Early Z culling

- Implemented by all modern GPUs, not just mobile GPUs
- Application needs to sort geometry to make early Z most effective. *Why?*



Graphics pipeline specifies that depth test is performed here!



Optimization: reorder pipeline operations: perform depth test immediately following rasterization and <u>before</u> fragment shading

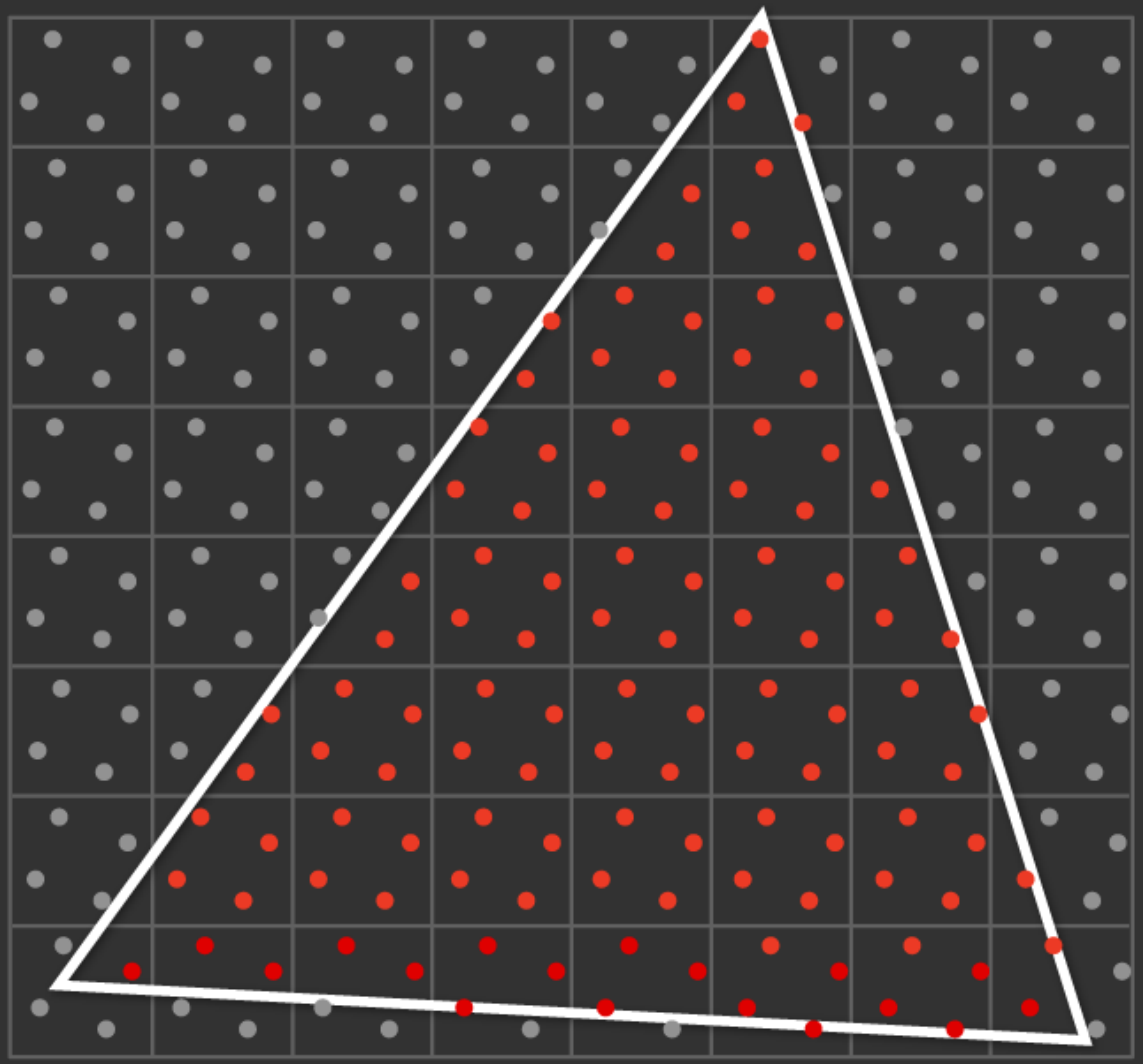
Key assumption: occlusion results do not depend on fragment shading

- Example operations that prevent use of this early Z optimization: enabling alpha test, fragment shader modifies fragment's Z value

Multi-sample anti-aliasing

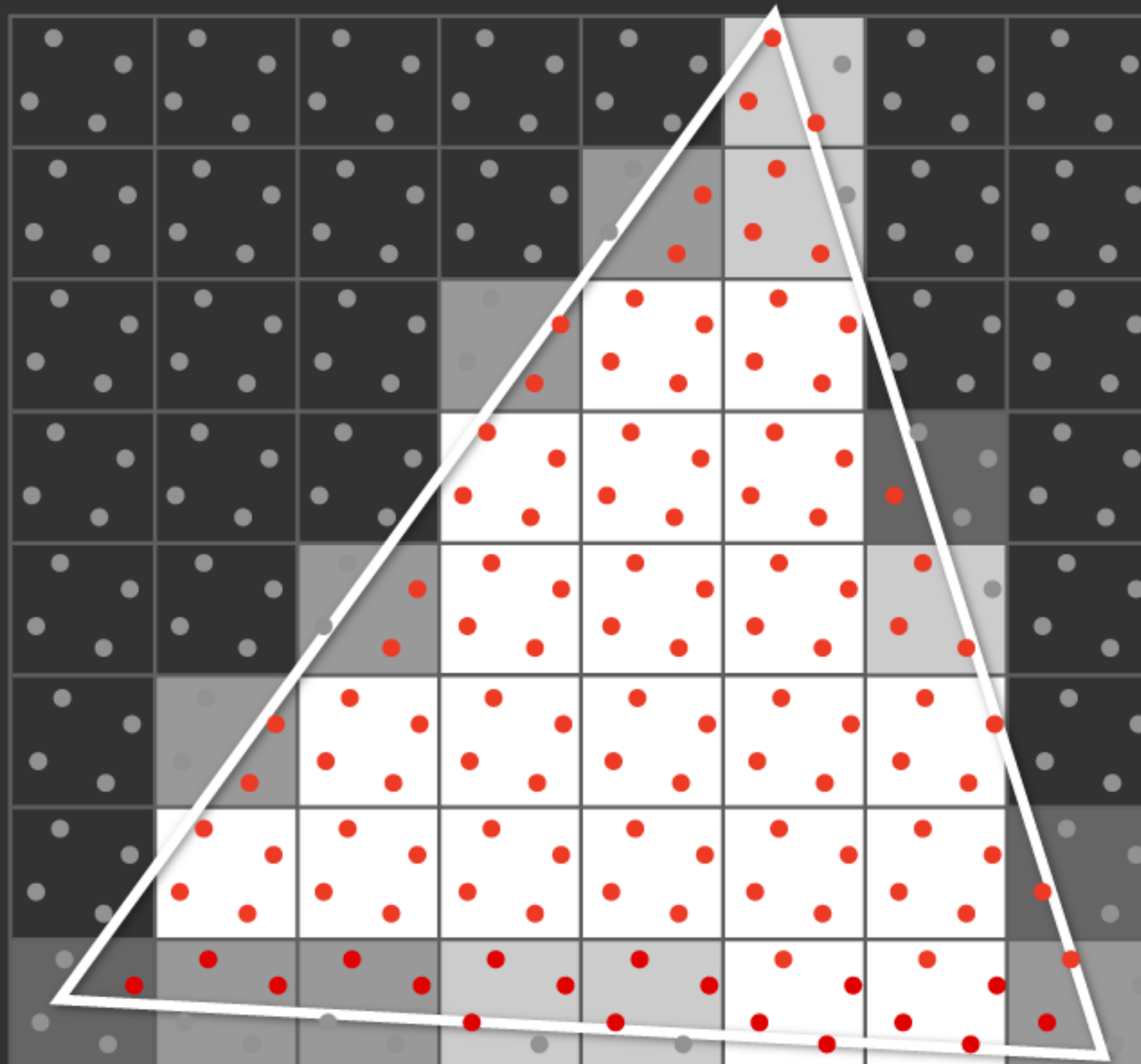
Supersampling triangle coverage

Multiple point in triangle tests per pixel. Why?



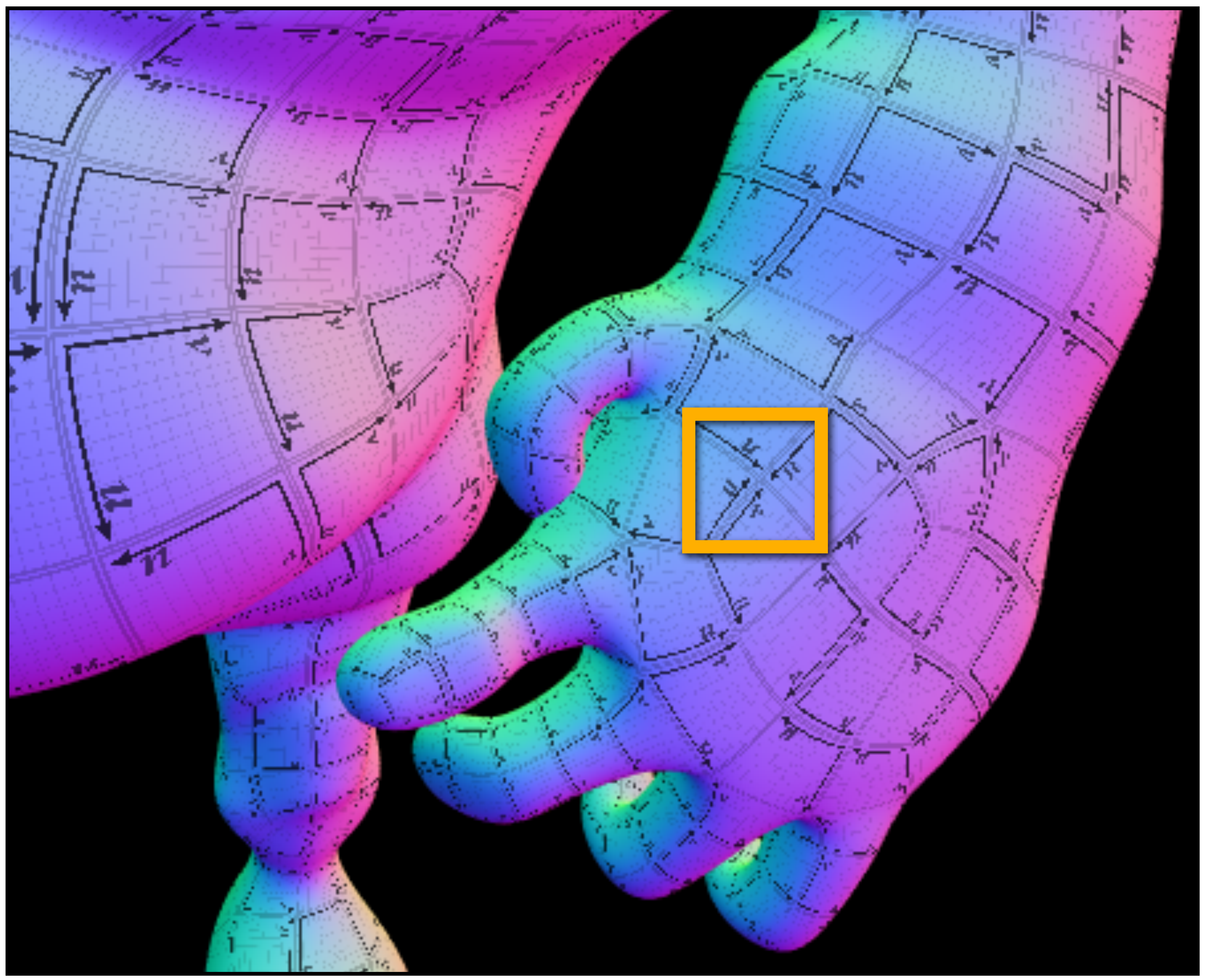
Supersampling to anti-alias triangle edges

Compute coverage using point-in-triangle tests

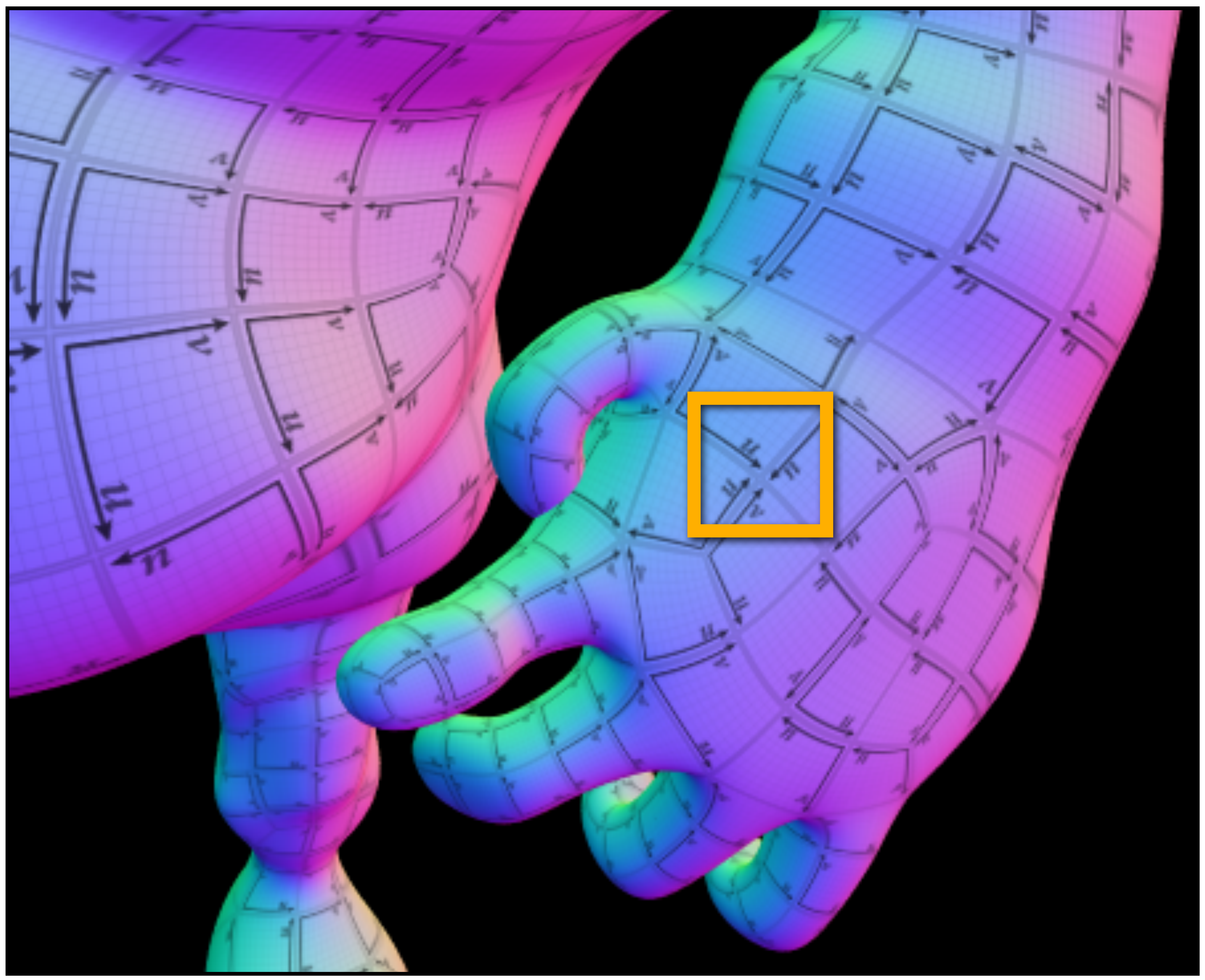


Texture data can be pre-filtered to avoid aliasing

Implication: ~ one shade per pixel is sufficient



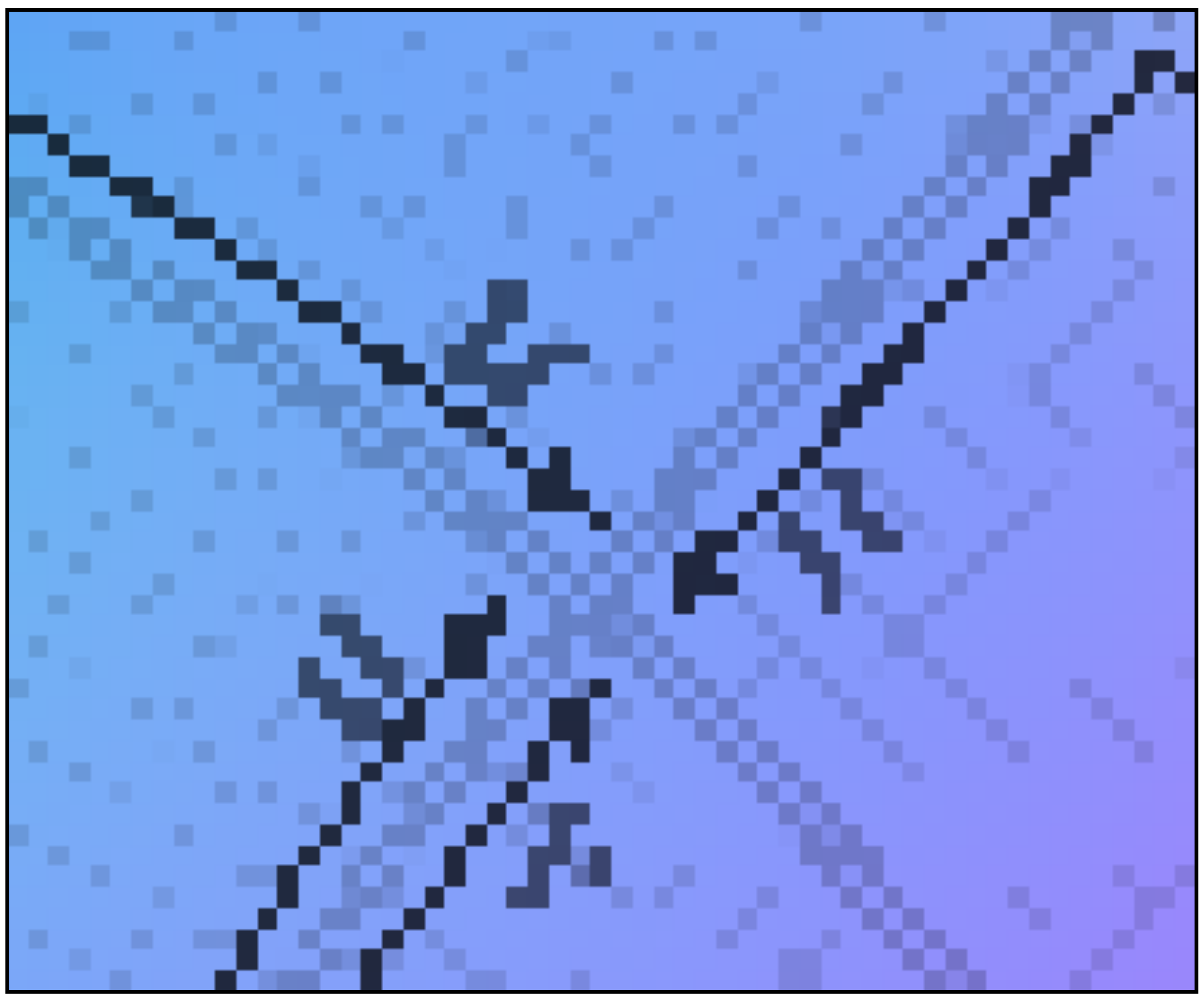
No pre-filtering
(aliased result)



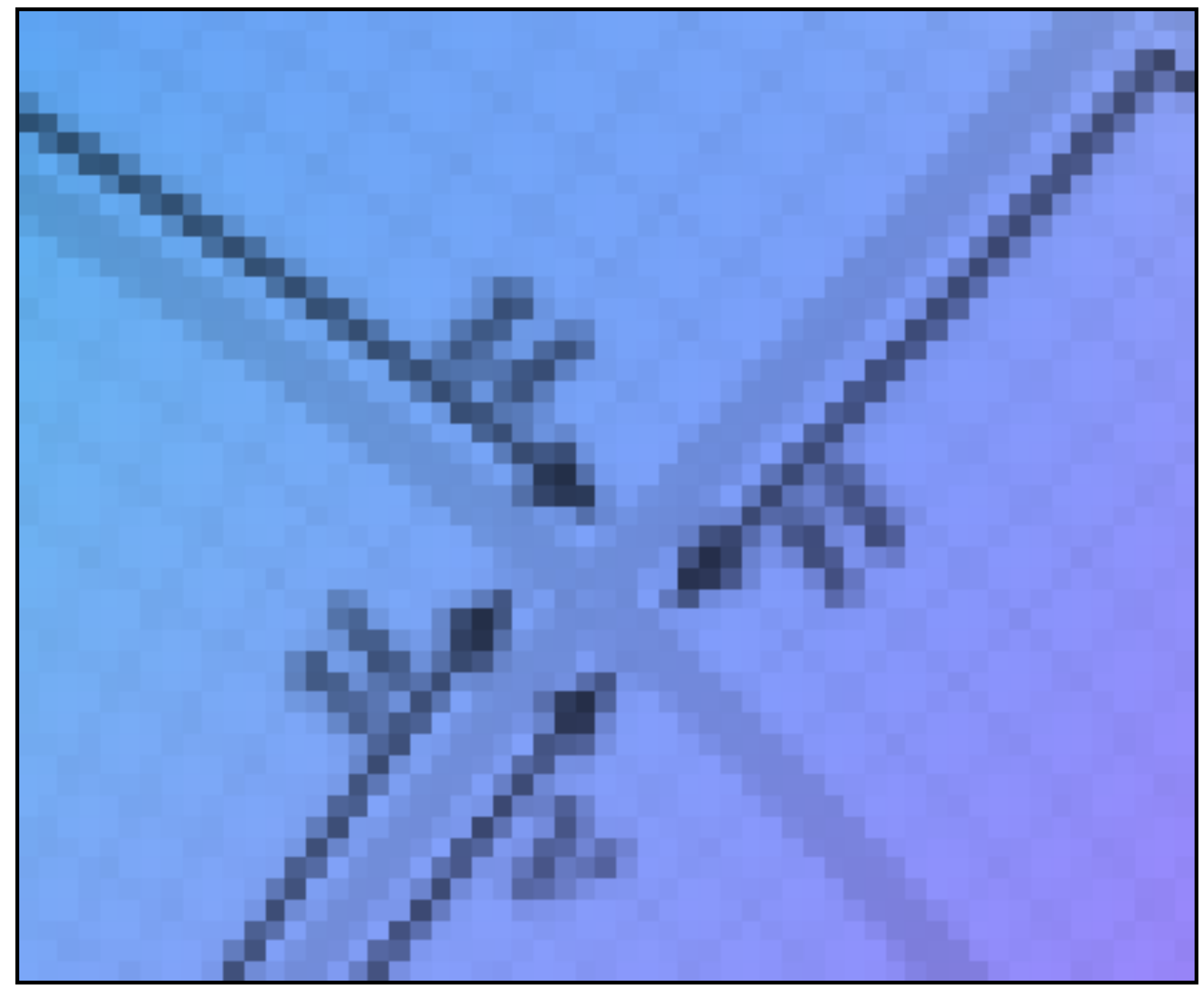
Pre-filtered texture

Texture data can be pre-filtered to avoid aliasing

Implication: ~ one shade per pixel is sufficient

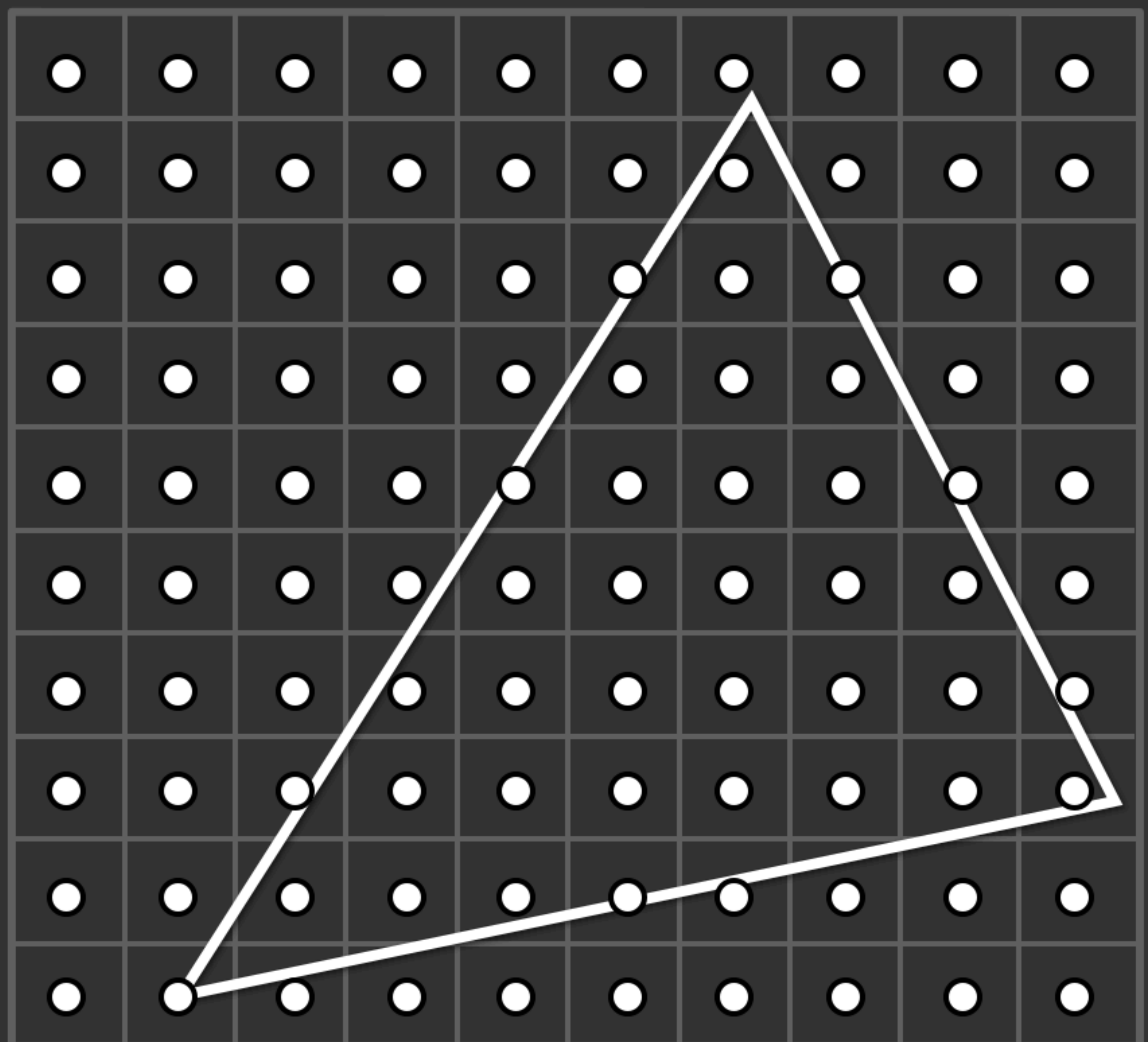


No pre-filtering
(aliased result)



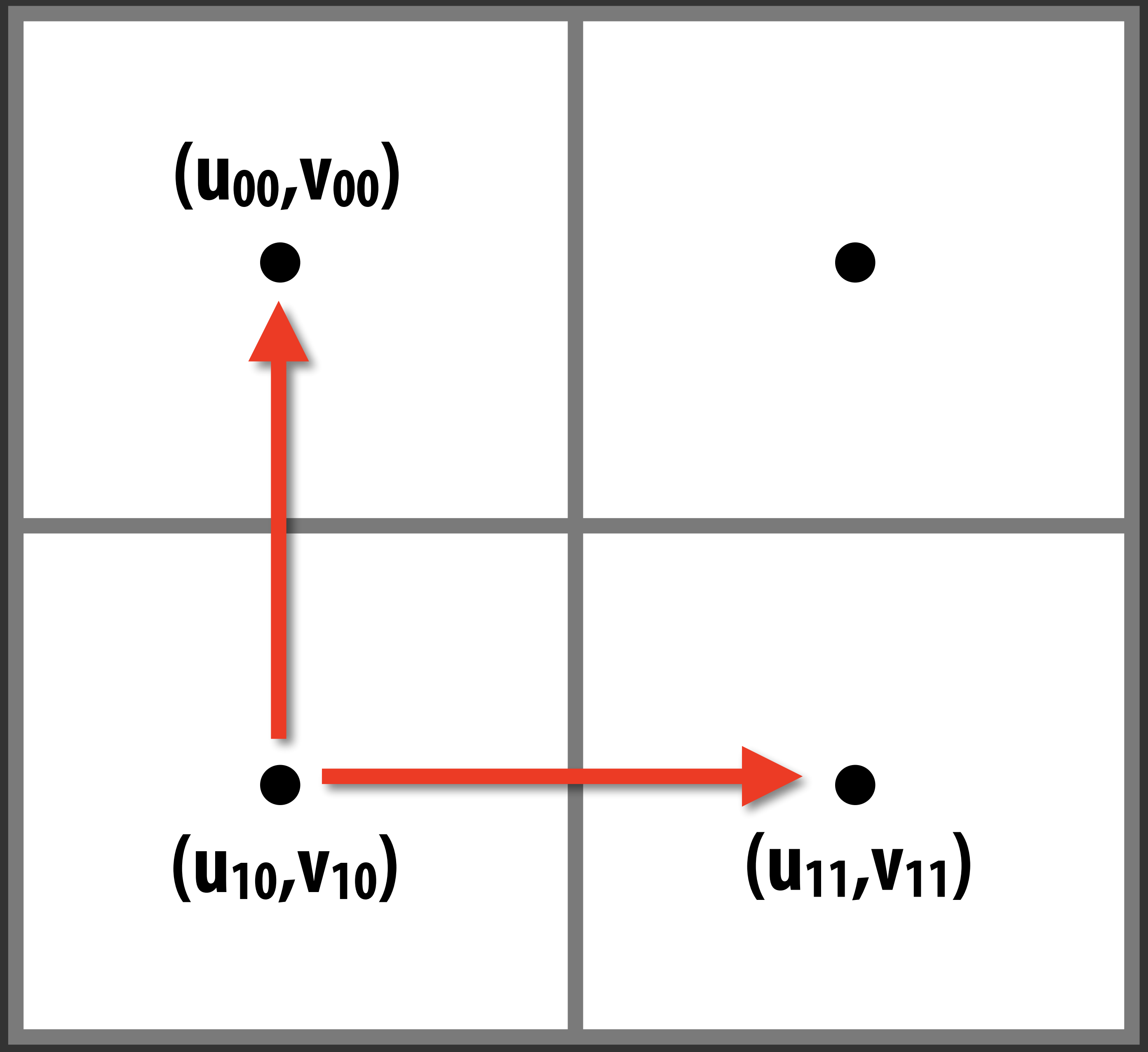
Pre-filtered texture

Shading sample locations

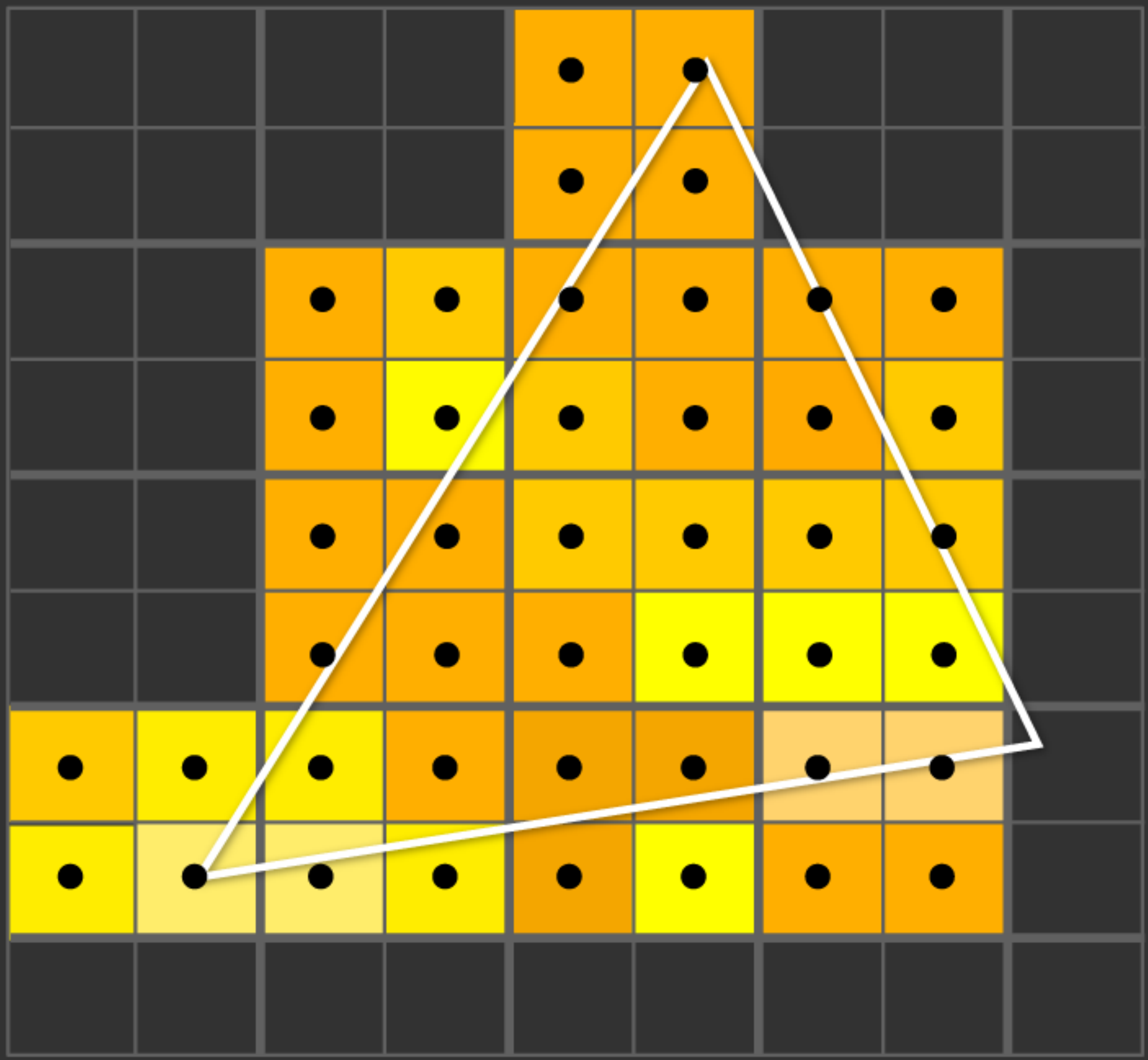


Quad fragments (2x2 pixel blocks)

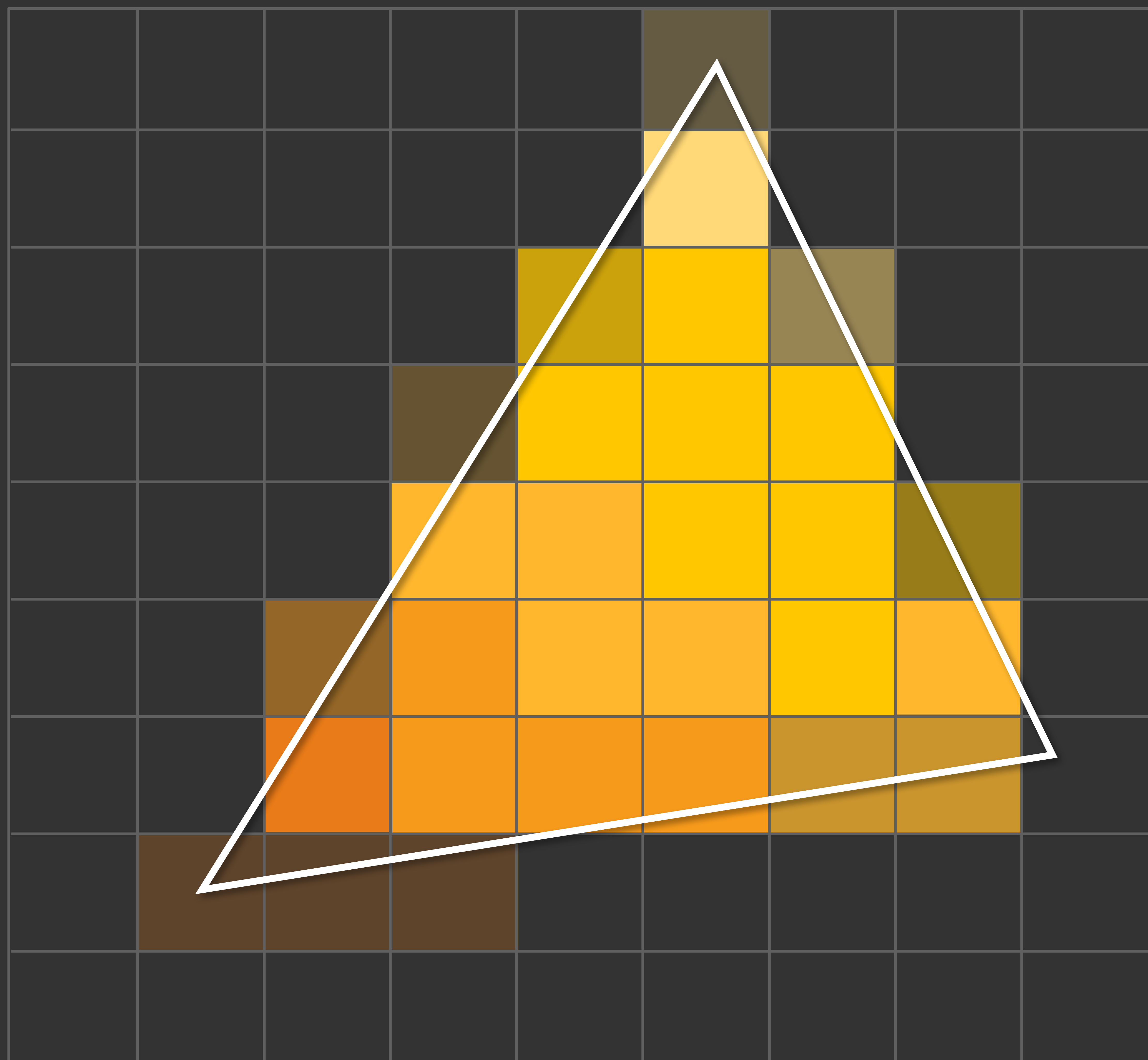
Difference neighboring texture coordinates to approximate derivatives



Shaded quad fragments

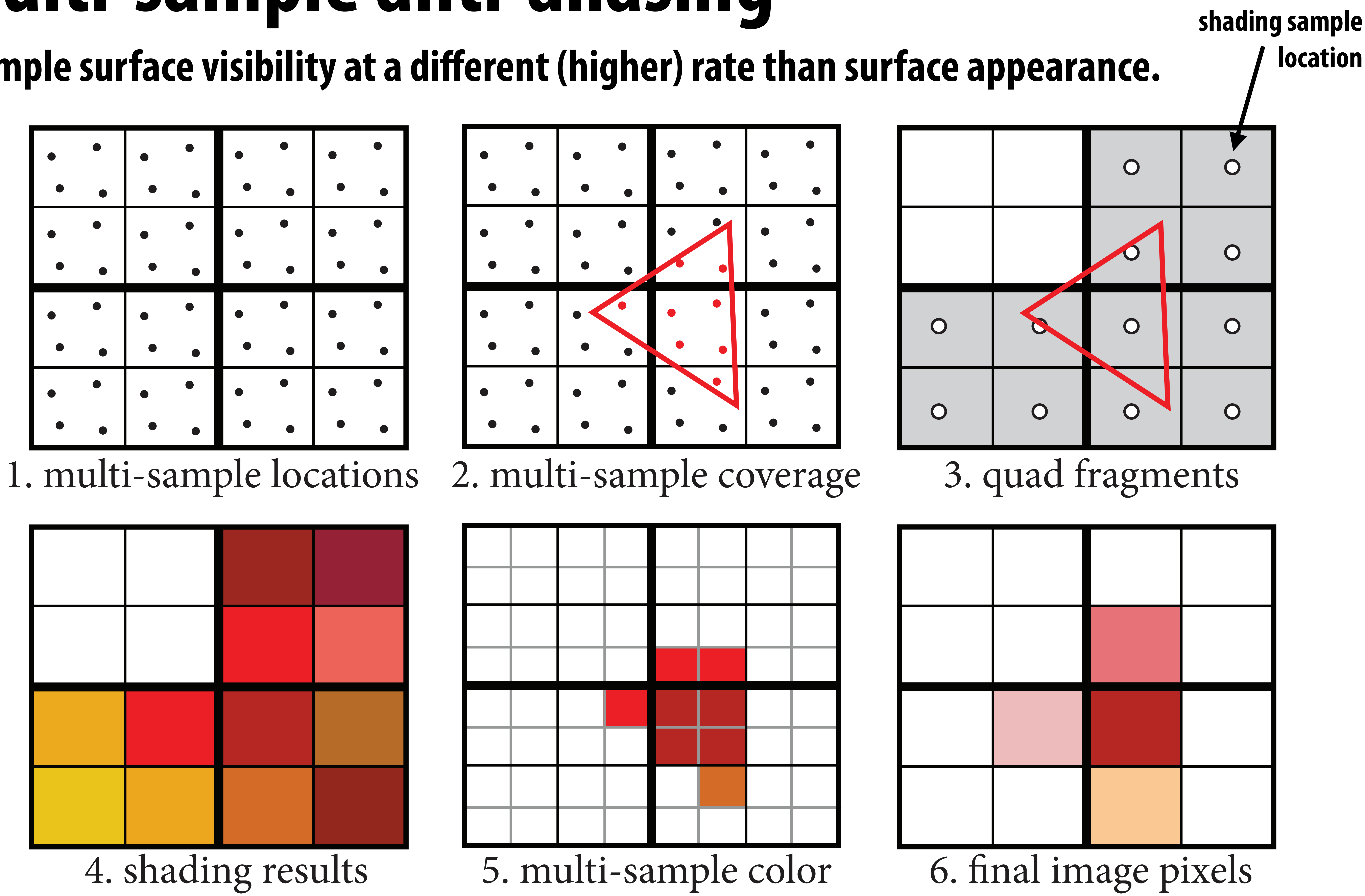


Final result: involving coverage



Multi-sample anti-aliasing

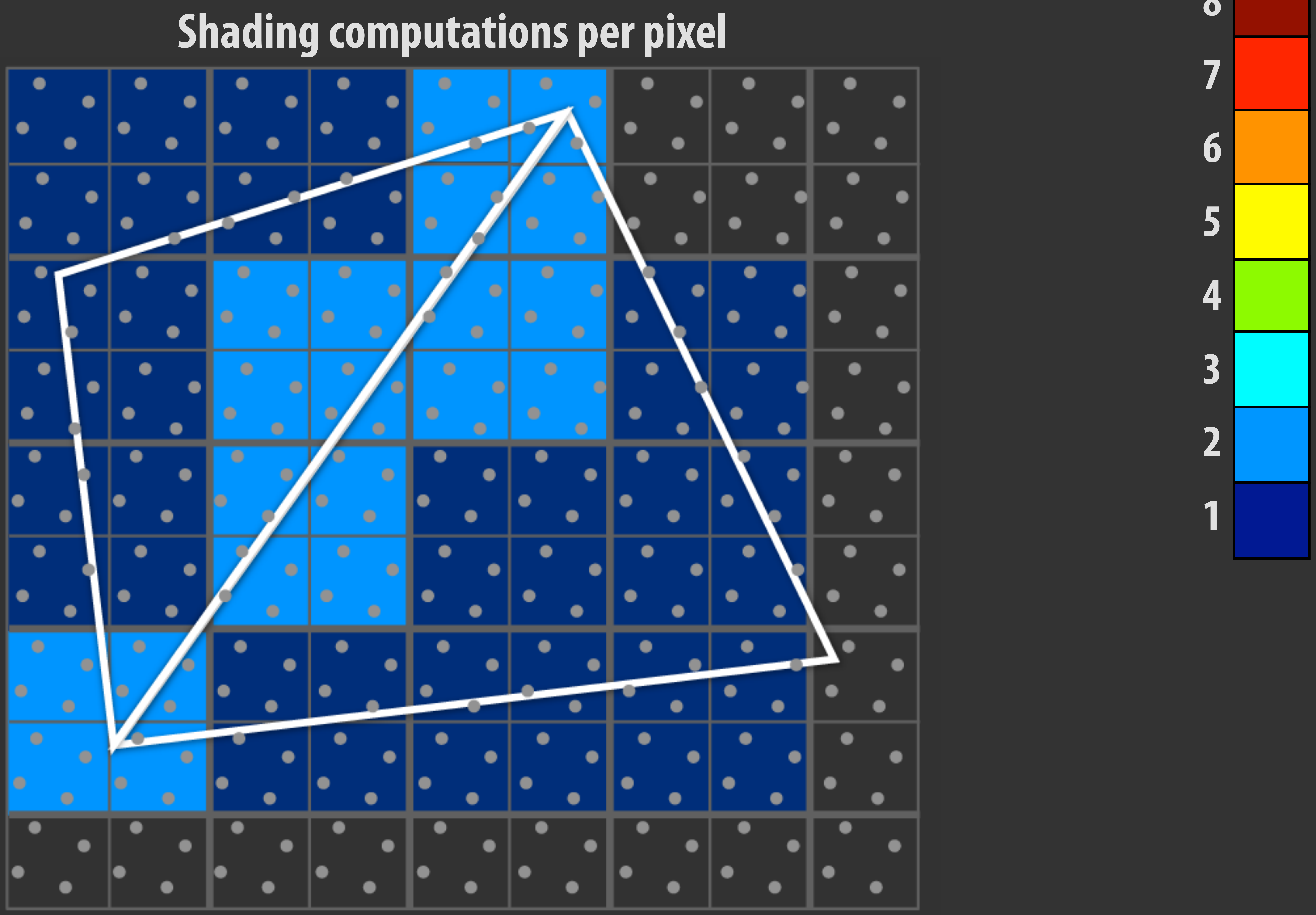
Sample surface visibility at a different (higher) rate than surface appearance.



Idea: use supersampling to anti-alias detail due to geometric visibility, use texture prefiltering (mipmapped texture access) to anti-alias detail to texture

Problem: pixels along edges shaded multiple times

Ug... technique designed to reduce shading in large triangle case actually increases shading when triangles get smaller (higher detailed scenes)



Read data less often

Reading less data conserves power

- Goal: redesign algorithms so that they make good use of on-chip memory or processor caches
  - And therefore transfer less data from memory

- A fact you might not have heard:
  - It is *far more* costly (in energy) to load/store data from memory, than it is to perform an arithmetic operation

"Ballpark" numbers [Sources: Bill Dally (NVIDIA), Tom Olson (ARM)]

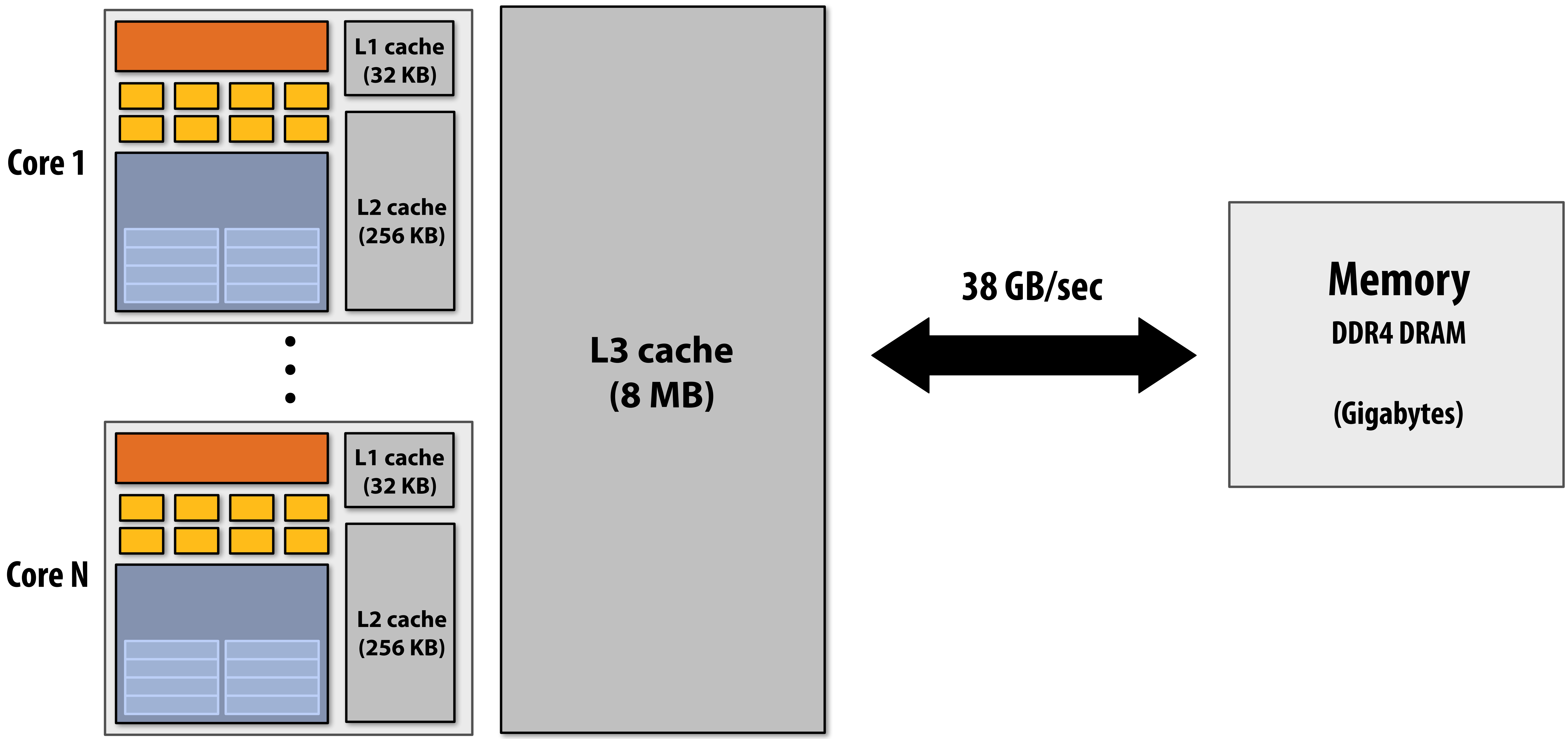
- Integer op: ~ 1 pJ *
- Floating point op: ~20 pJ *
- Reading 64 bits from small local SRAM (1mm away on chip): ~ 26 pJ
- Reading 64 bits from low power mobile DRAM (LPDDR): ~1200 pJ

Implications

- Reading 10 GB/sec from memory: ~1.6 watts

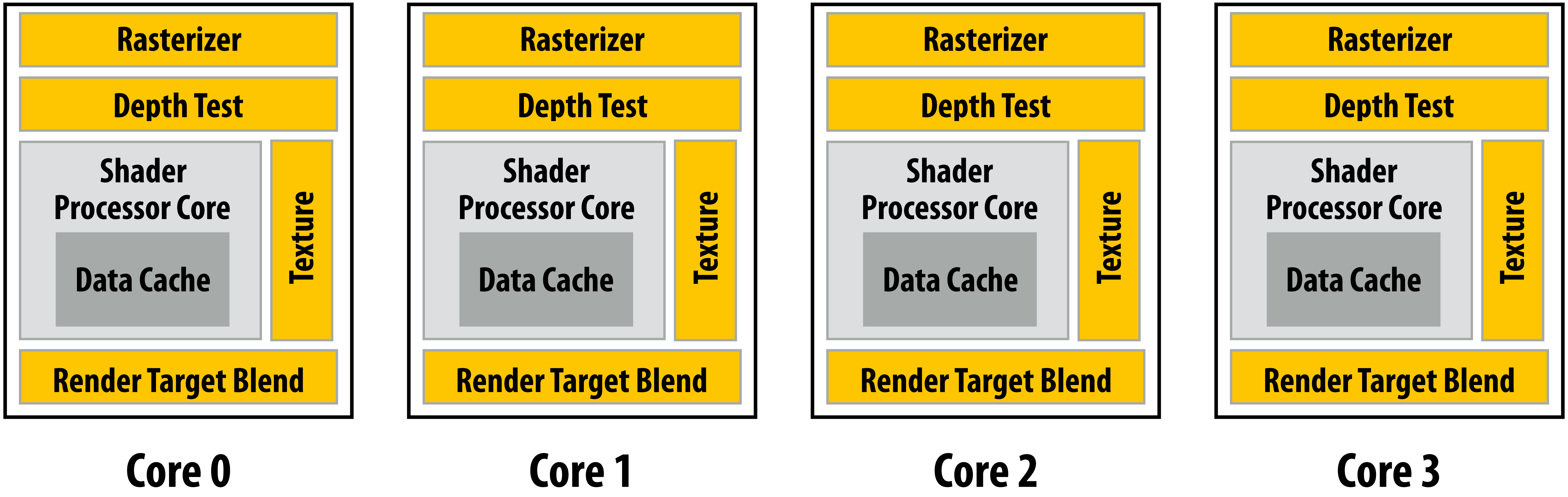
* Cost to just perform the logical operation, not counting overhead of instruction decode, load data from registers, etc.

What does a data cache do in a processor?



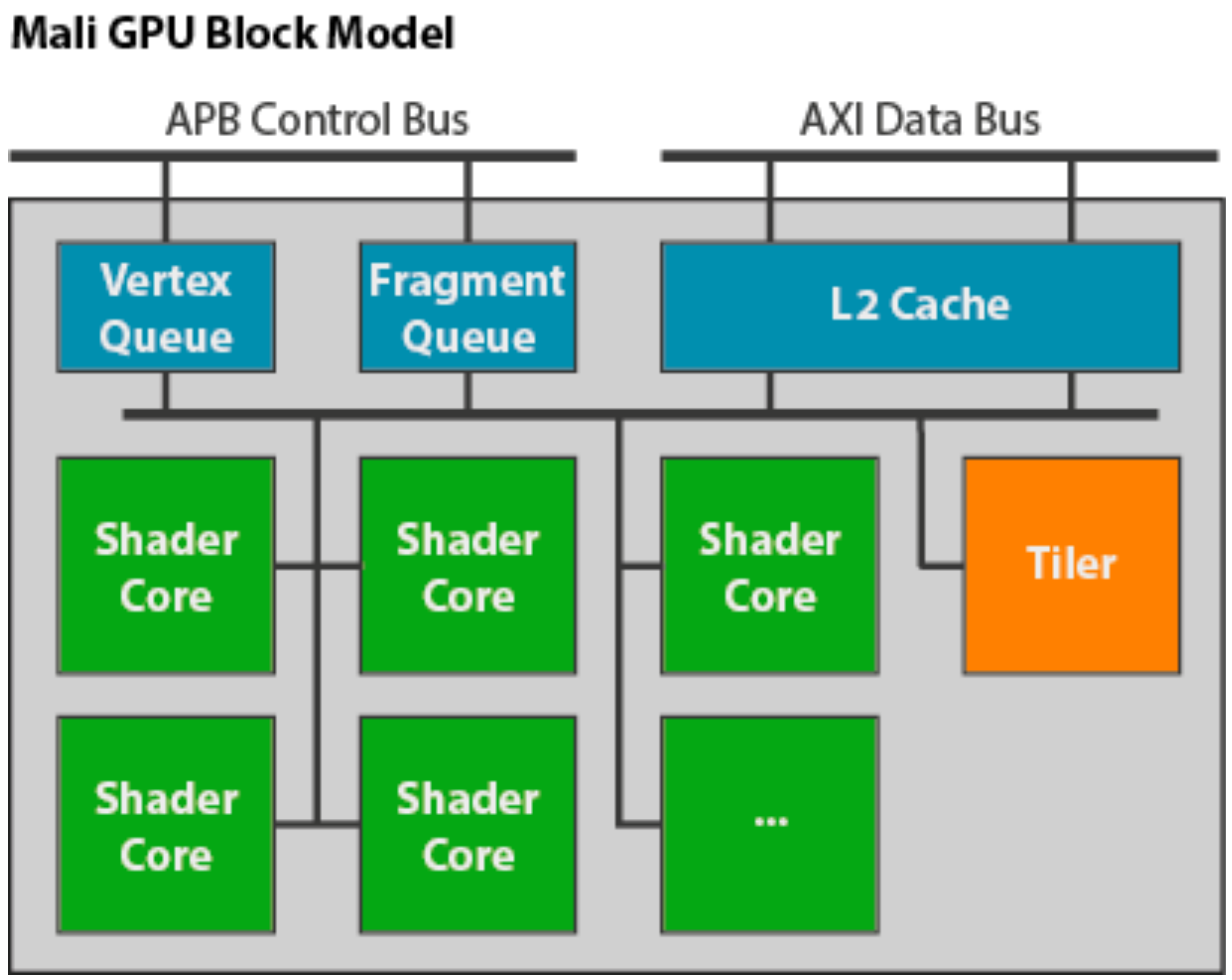
Today: a simple mobile GPU

- A set of programmable cores (run vertex and fragment shader programs)
- Hardware for rasterization, texture mapping, and frame-buffer access

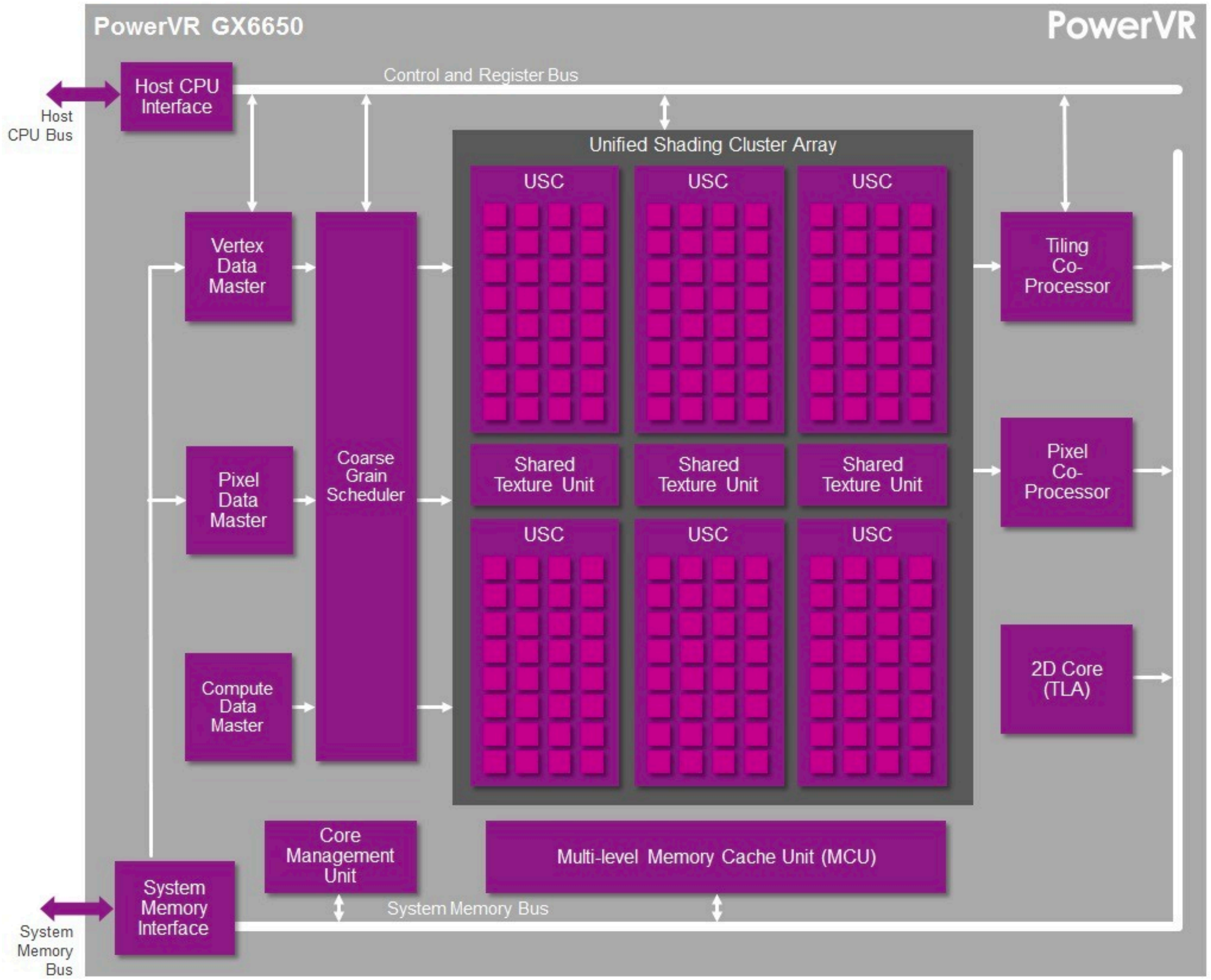


Block diagrams from vendors

ARM Mali G72MP18



Imagination PowerVR (in earlier iPhones)



Let's consider different workloads

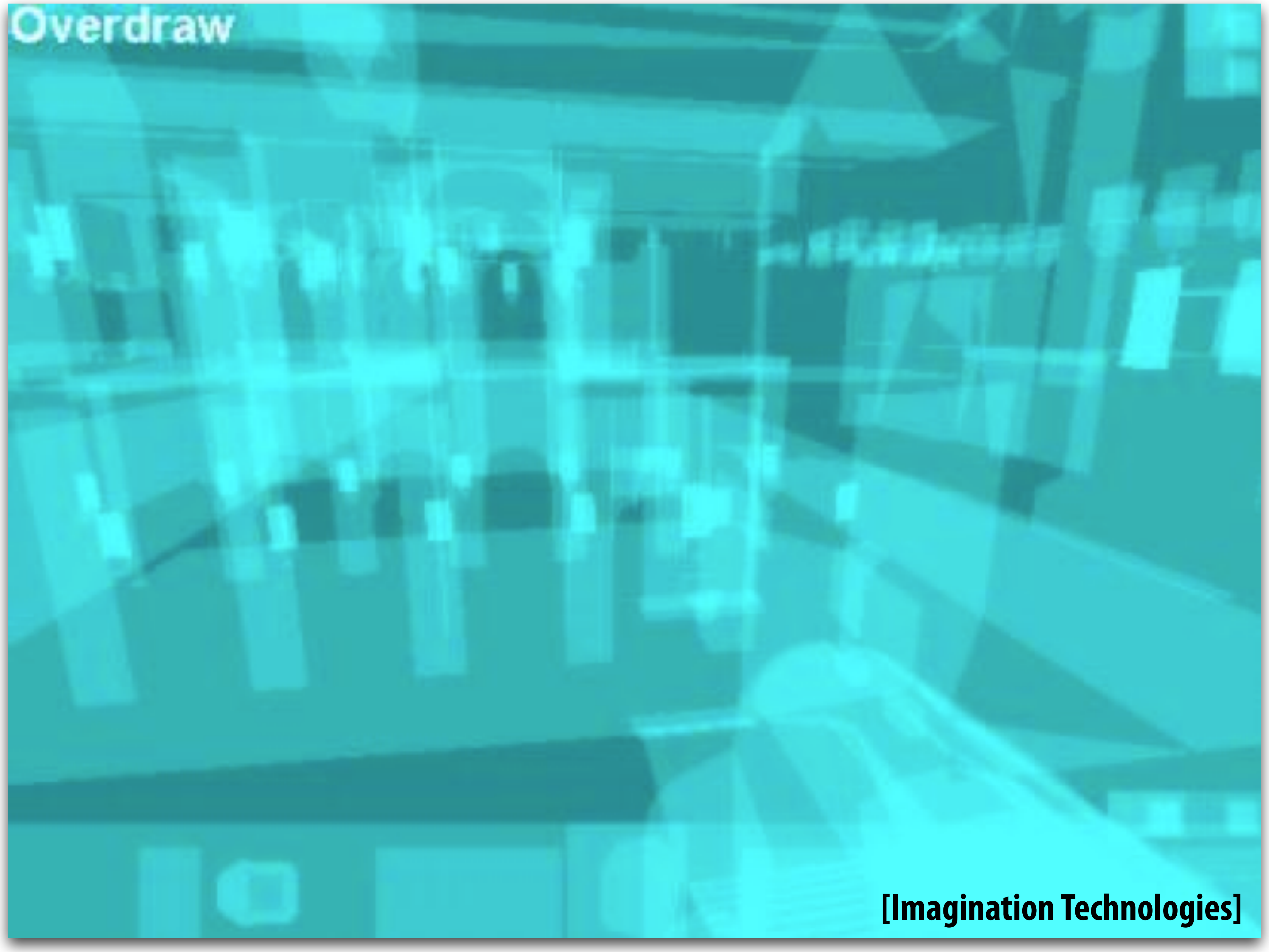
Average triangle size

Image credit:
https://www.theverge.com/2013/11/29/5155726/next-gen-supplementary-piece
http://www.mobygames.com/game/android/ghostbusters-slime-city/screenshots/gameShotId,852293/

Let's consider different workloads

Scene depth complexity

Average number of overlapping triangles per pixel



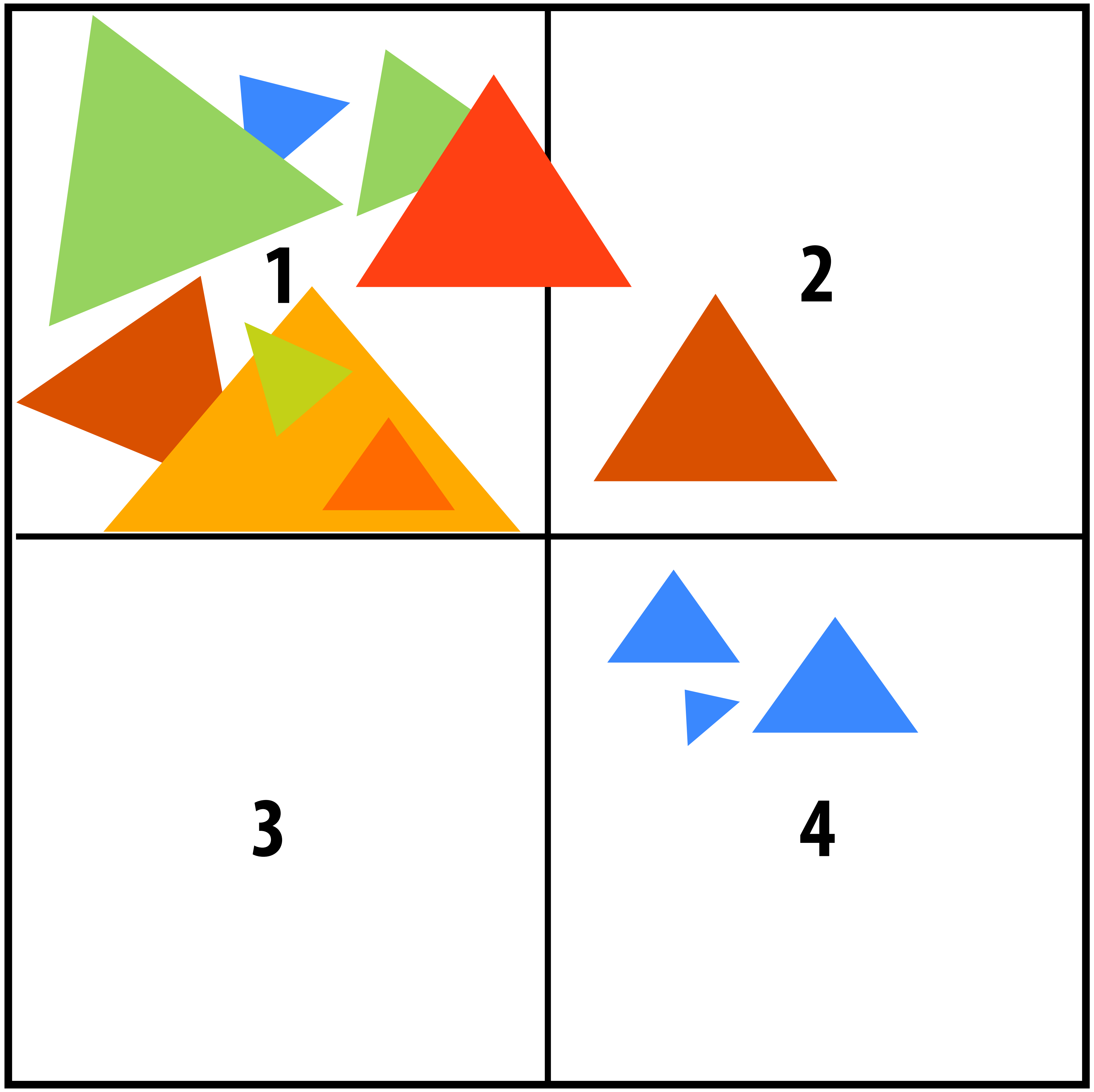
In this visualization: bright colors = more overlap

One very simple solution

- Let's assume four GPU cores
- Divide screen into four quadrants, each processor processes all triangles, but only renders triangles that overlap quadrant
- *Problems?*

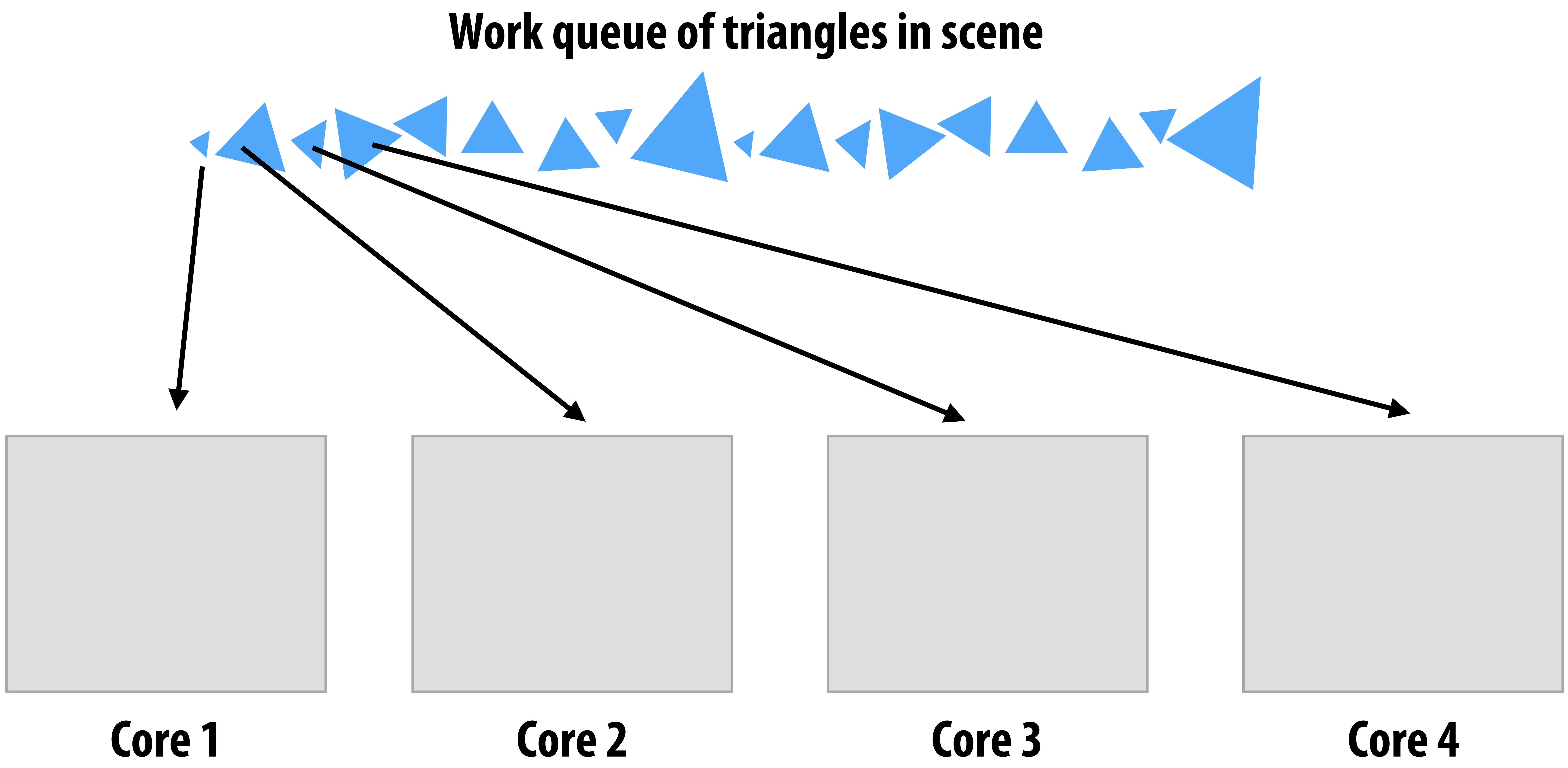
Problem: unequal work partitioning

(partition the primitives to parallel units based on screen overlap)



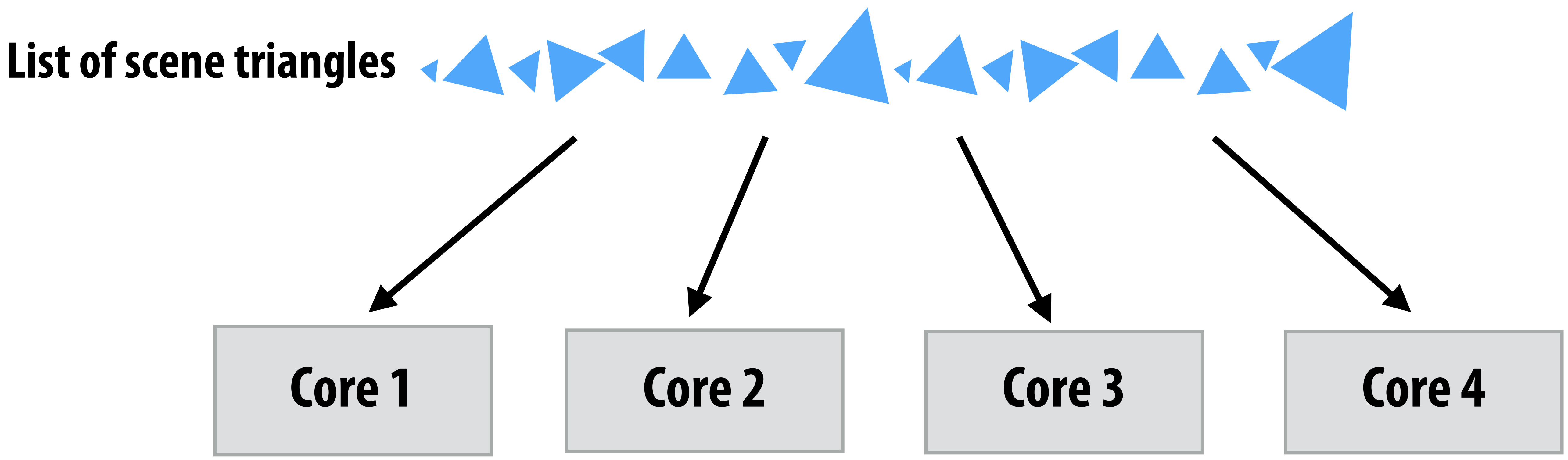
Step 1: parallel geometry processing

- Distribute triangles to the four processors (e.g., round robin)
- In parallel, processors perform vertex processing



Step 2: sort triangles into per-tile lists

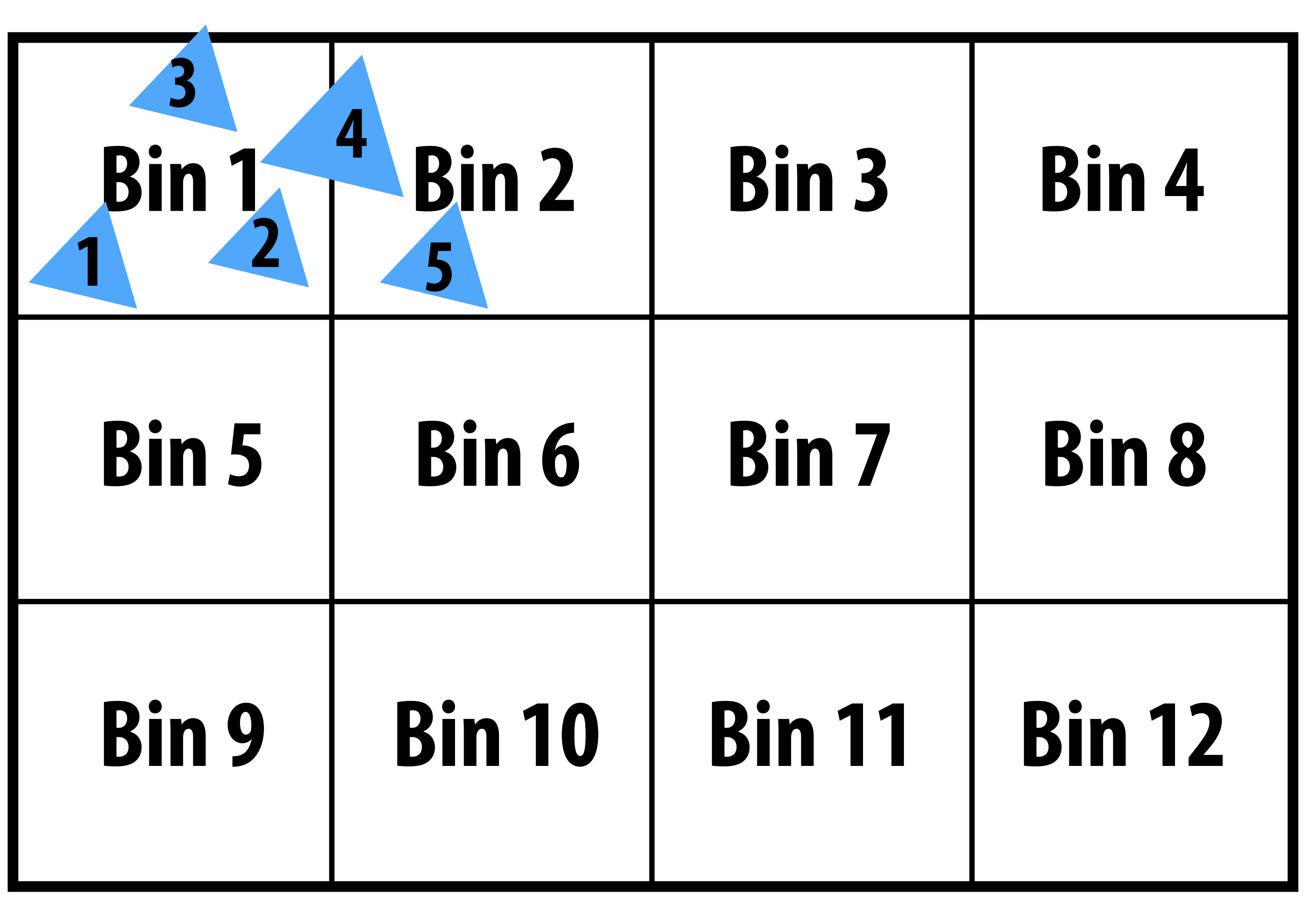
- Divide screen into tiles, one triangle list per "tile" of screen (called a "bin")
- Core runs vertex processing, computes 2D triangle/screen-tile overlap, inserts triangle into appropriate bin(s)



After processing first five triangles:

Bin 1 list: 1,2,3,4

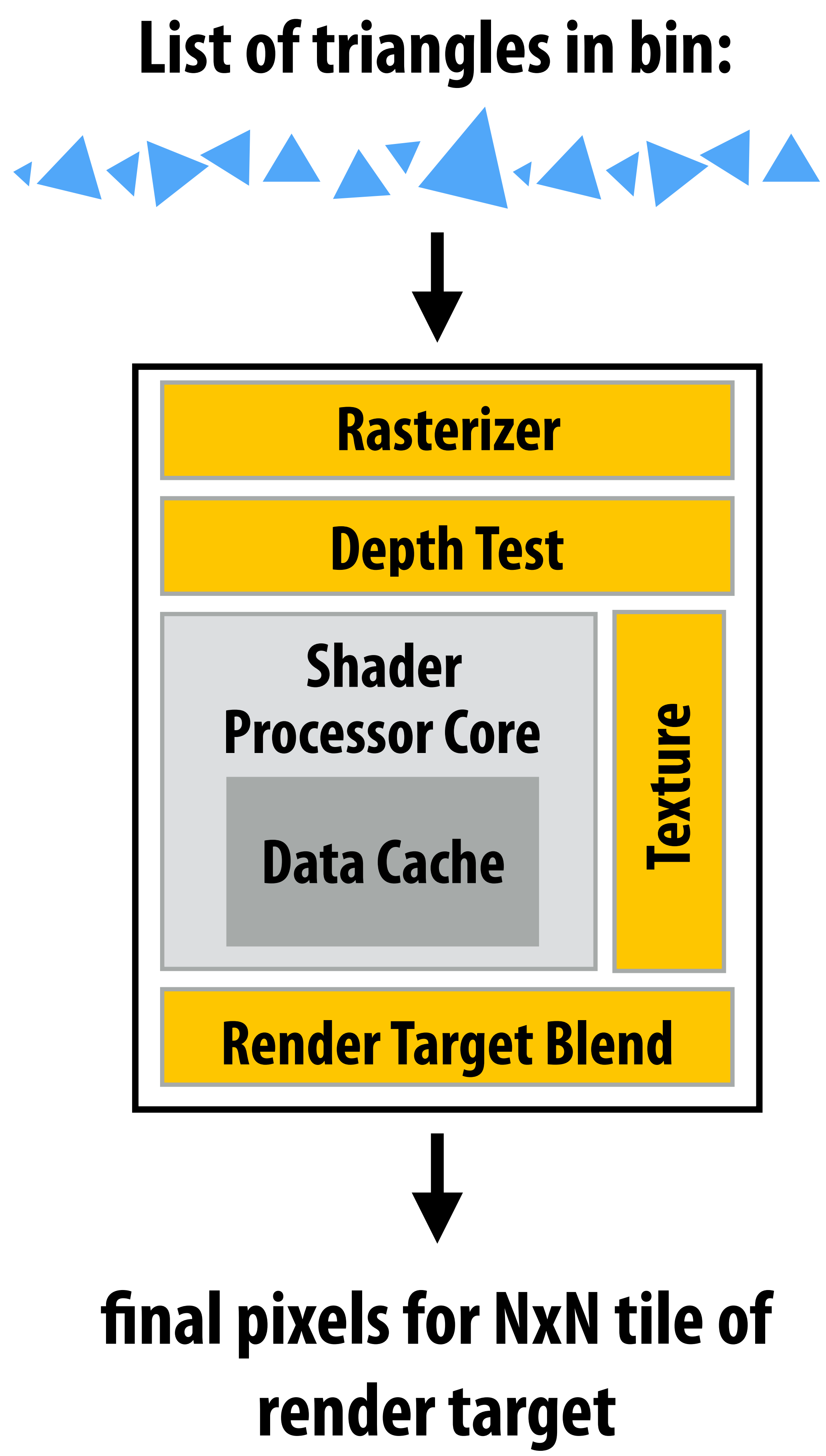
Bin 2 list: 4,5



Step 3: per-tile processing

- In parallel, the cores process the bins: performing rasterization, fragment shading, and frame buffer update

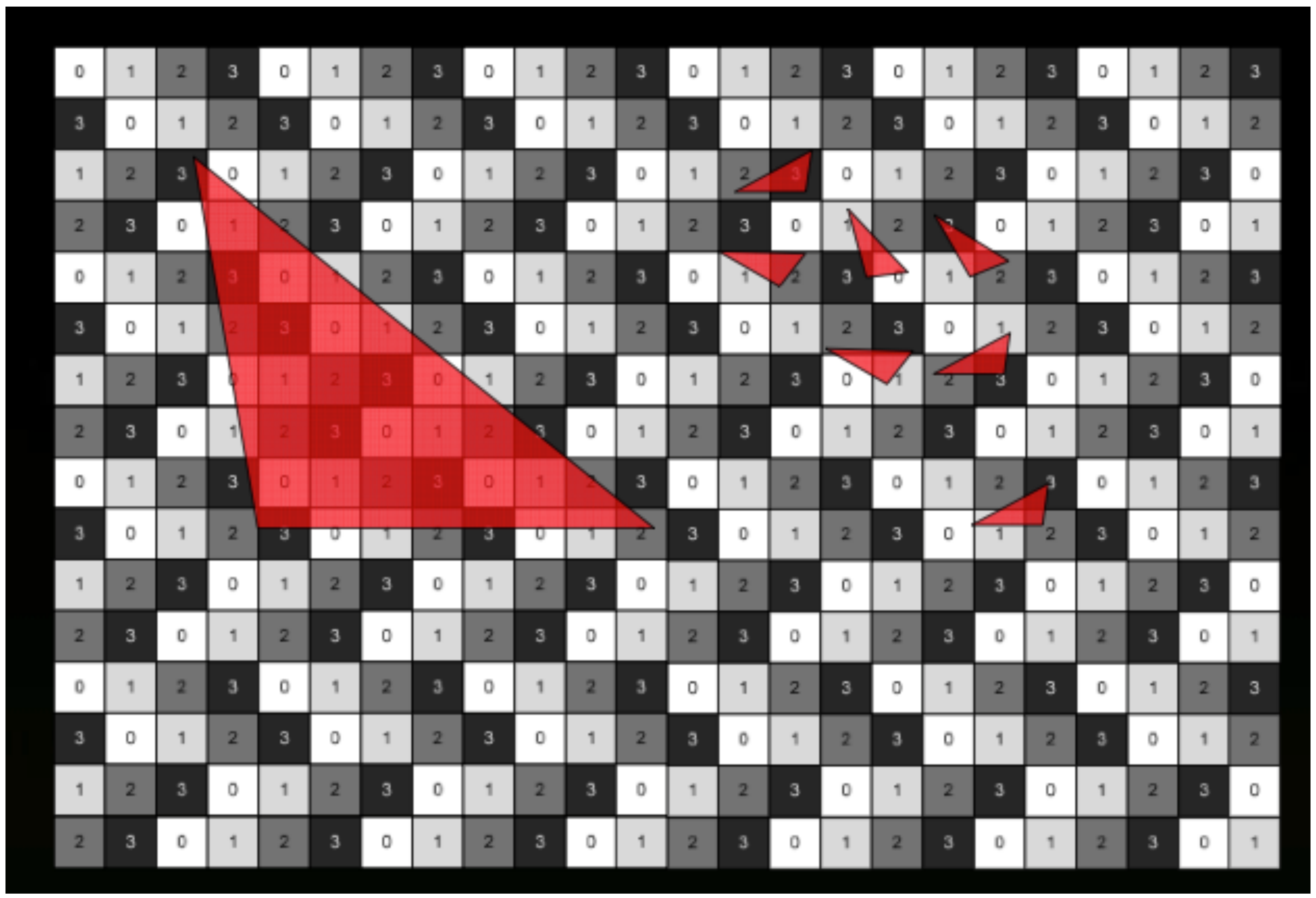
- While (more bin's left to process):
  - Assign bin to available core
  - For all triangles in bin:
    - Rasterize
    - Fragment shade
    - Depth test
    - Render target blend



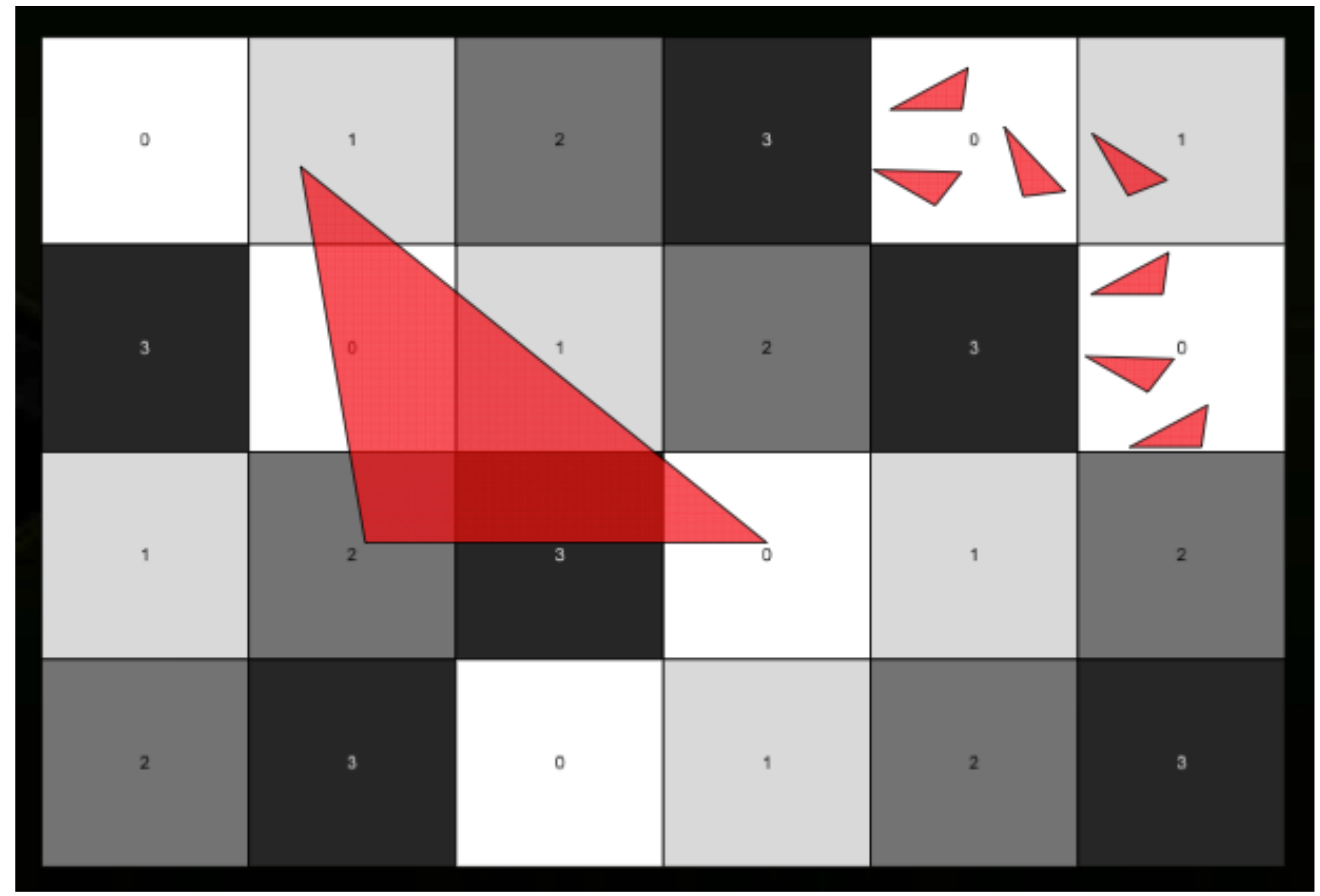
What should the size of tiles be?

What should the size of the bins be?

Fine granularity



Coarse granularity



[Image source: NVIDIA]

What size should the tiles be?

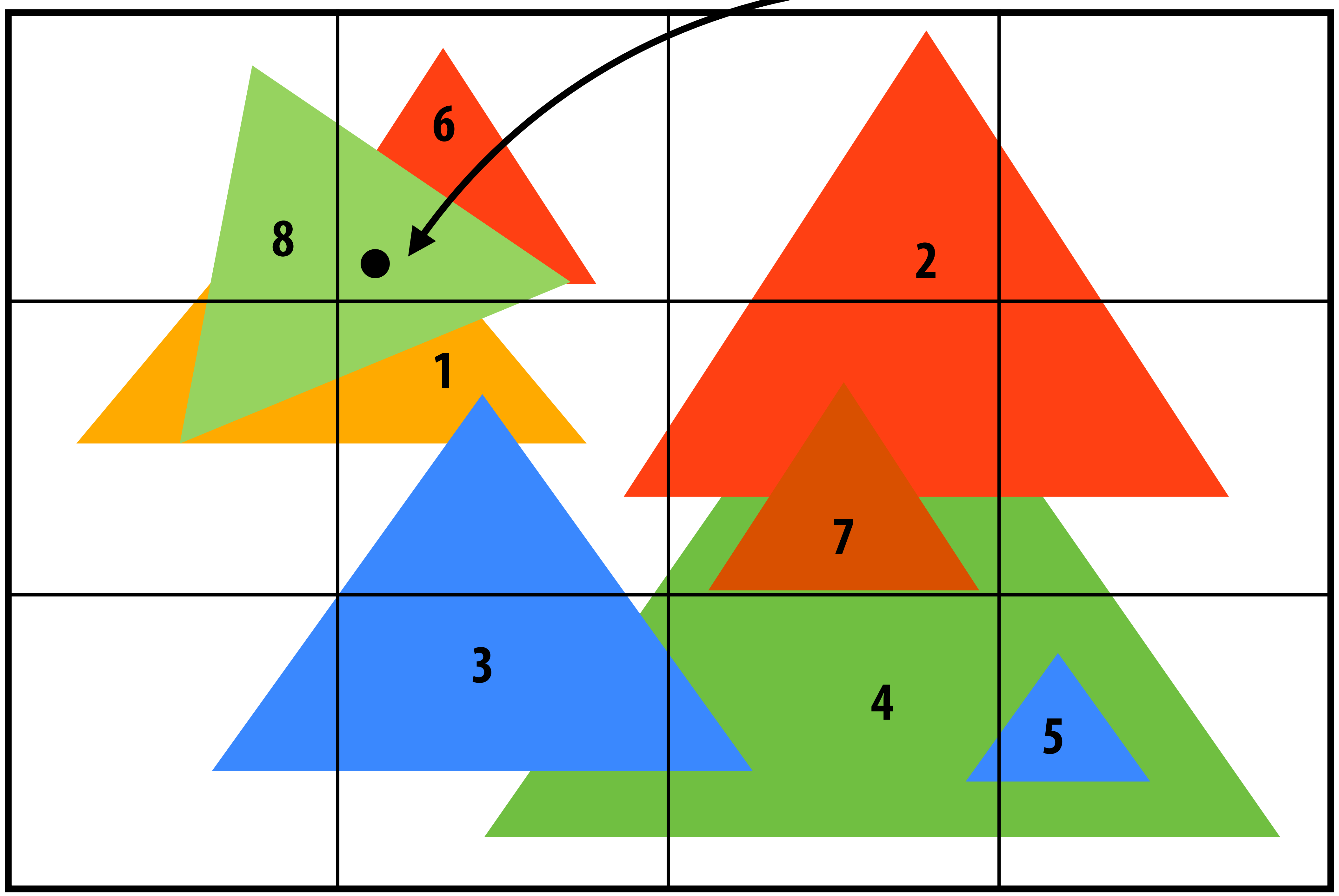
- Small enough for a tile of the color buffer and depth buffer (potentially supersampled) to fit in a shader processor core's on-chip storage (i.e., cache)

- Tile sizes in range 16x16 to 64x64 pixels are common

- ARM Mali GPU: commonly uses 16x16 pixel tiles

Tiled rendering "sorts" the scene in 2D space to enable efficient color/depth buffer access

Consider rendering without a sort:
(process triangles in order given by application)



This sample is updated three times during rendering, but it may have fallen out of cache in between accesses

Now consider step 3 of a tiled renderer:

```
Initialize Z and color buffer for tile
for all triangles in tile:
  for all each fragment:
    shade fragment
    update depth/color
write color tile to final image buffer
```

Q. Why doesn't the renderer need to read color or depth buffer from memory?

Q. Why doesn't the renderer need to write depth buffer in memory? *

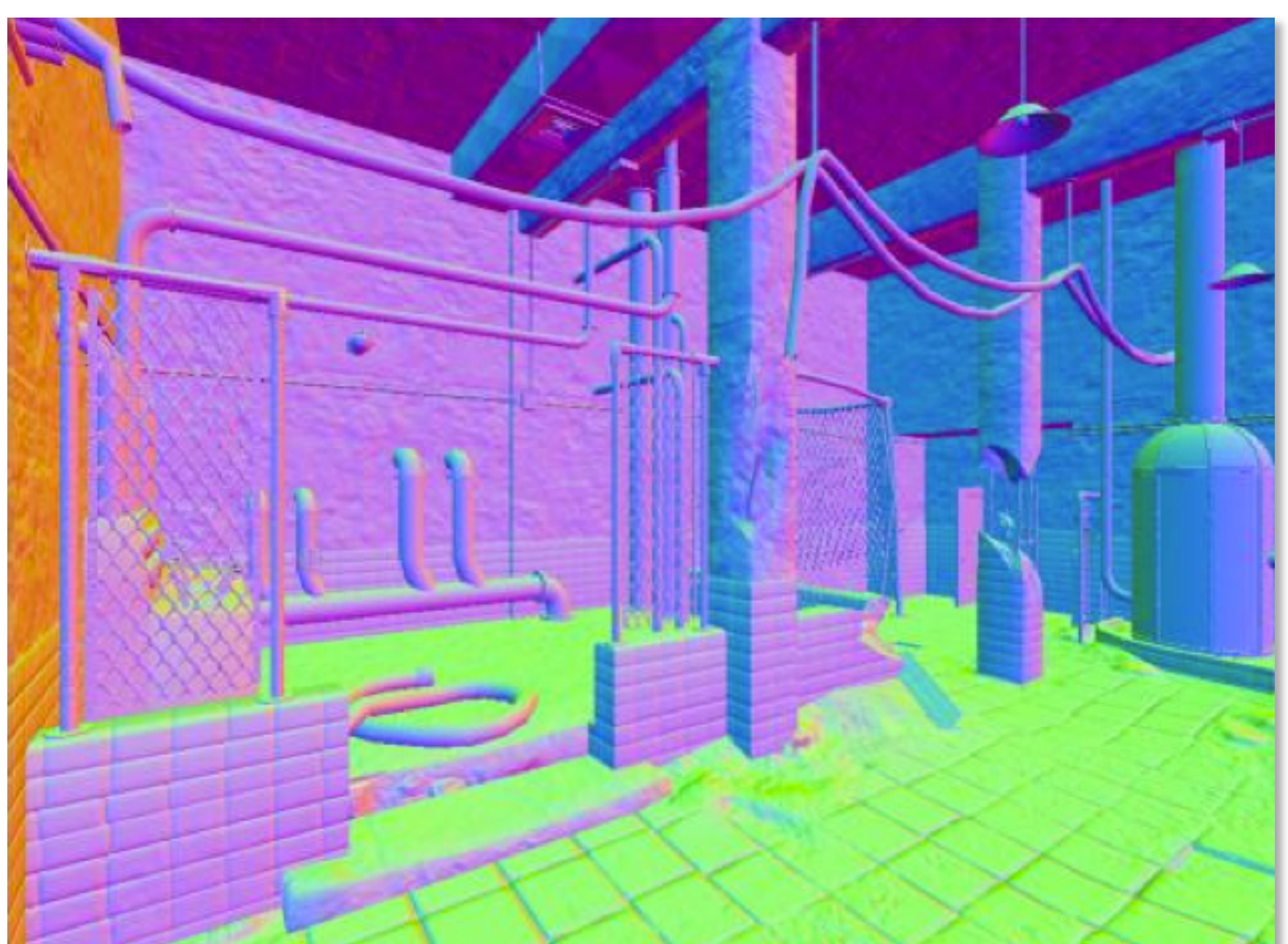
* Assuming application does not need depth buffer for other purposes.

Recall: deferred shading using a G-buffer

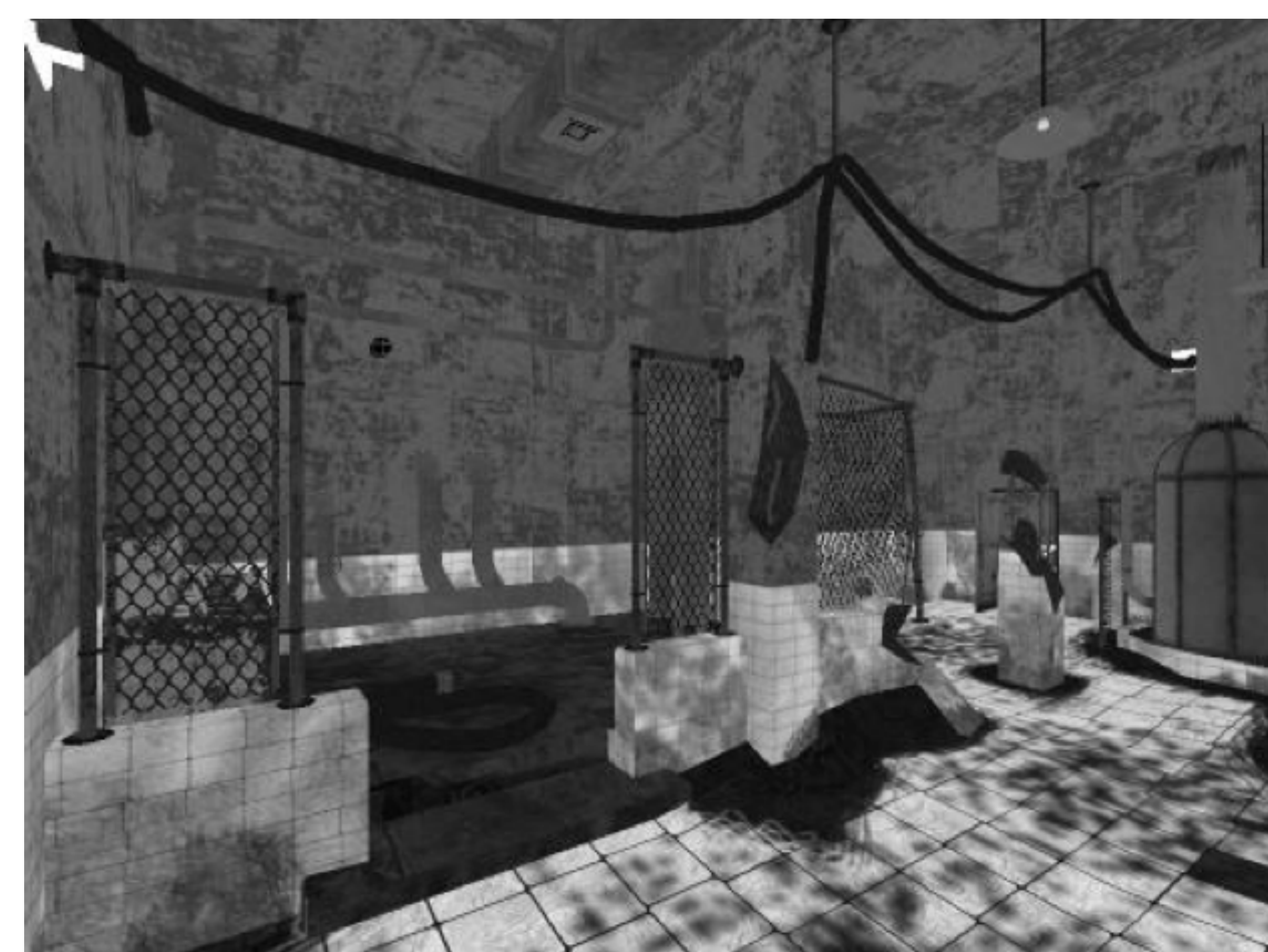
Key benefit: shade each sample *exactly* once.

Albedo (Reflectance)

Depth



Normal

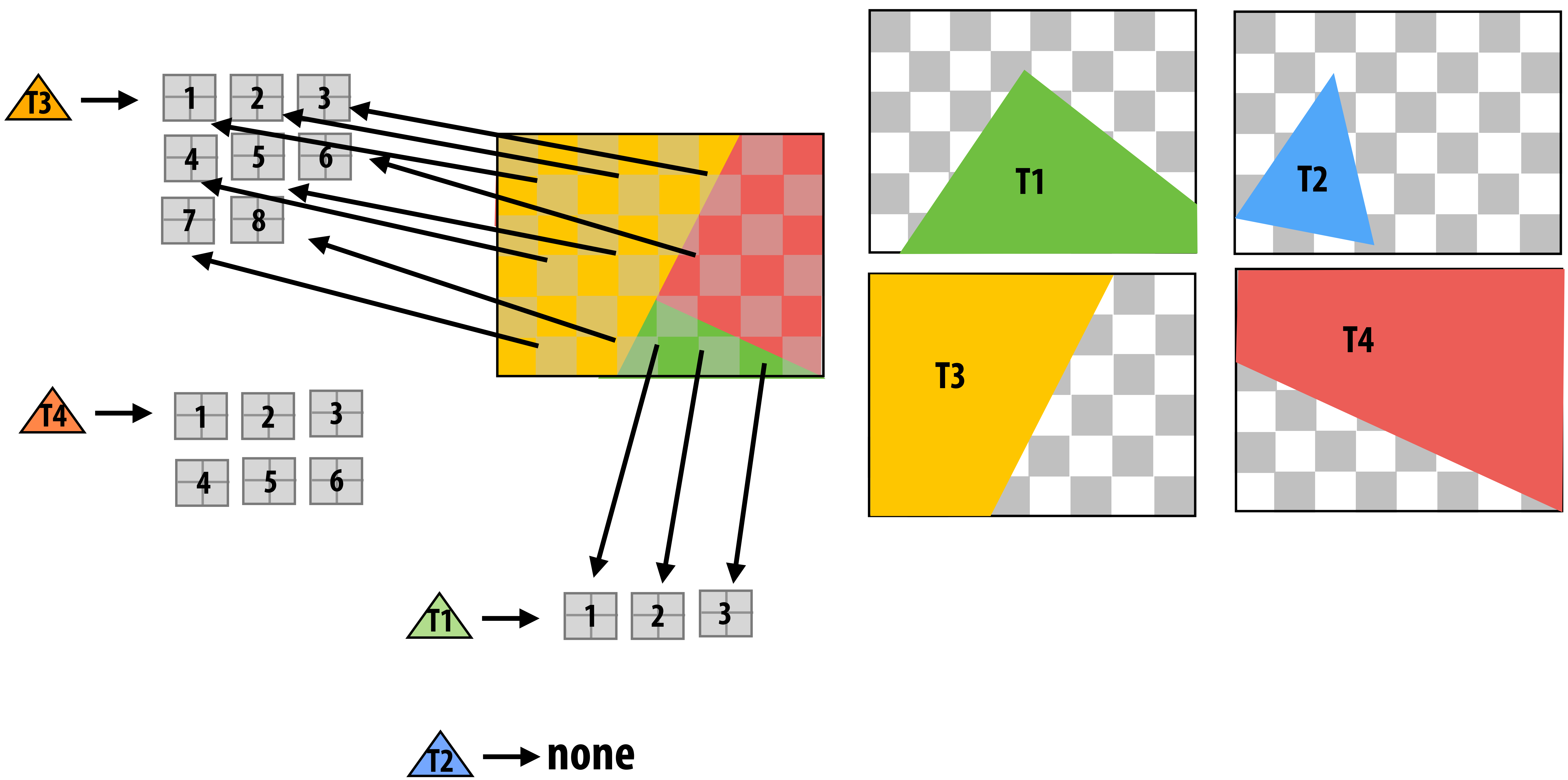


Specular

Image Credit: J. Klint, "Deferred Rendering in Leadworks Engine"

Tile-based deferred rendering (TBDR)

- Many mobile GPUs implement deferred shading in the hardware!
- Divide step 3 of tiled pipeline into two phases:
- Phase 1: compute what triangle/quad fragment is visible at every sample
- Phase 2: perform shading of only the visible quad fragments



The story so far

- **Computation-saving optimizations (shade less)**
  - multi-sample anti-aliasing
  - early Z cull
  - tile-based deferred shading

- **Bandwidth-saving optimizations**
  - tile-based rendering
  - many more…

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 mipmap level *L*
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, L)

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

Texture compression

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

Simple scheme: color palette (indexed color)

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

Color palette (eight colors)

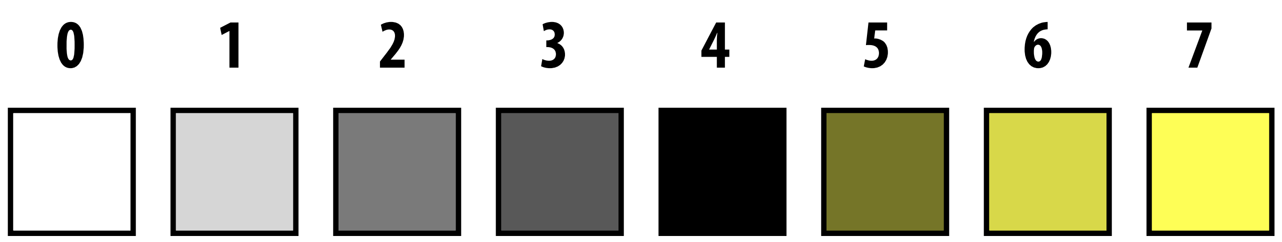
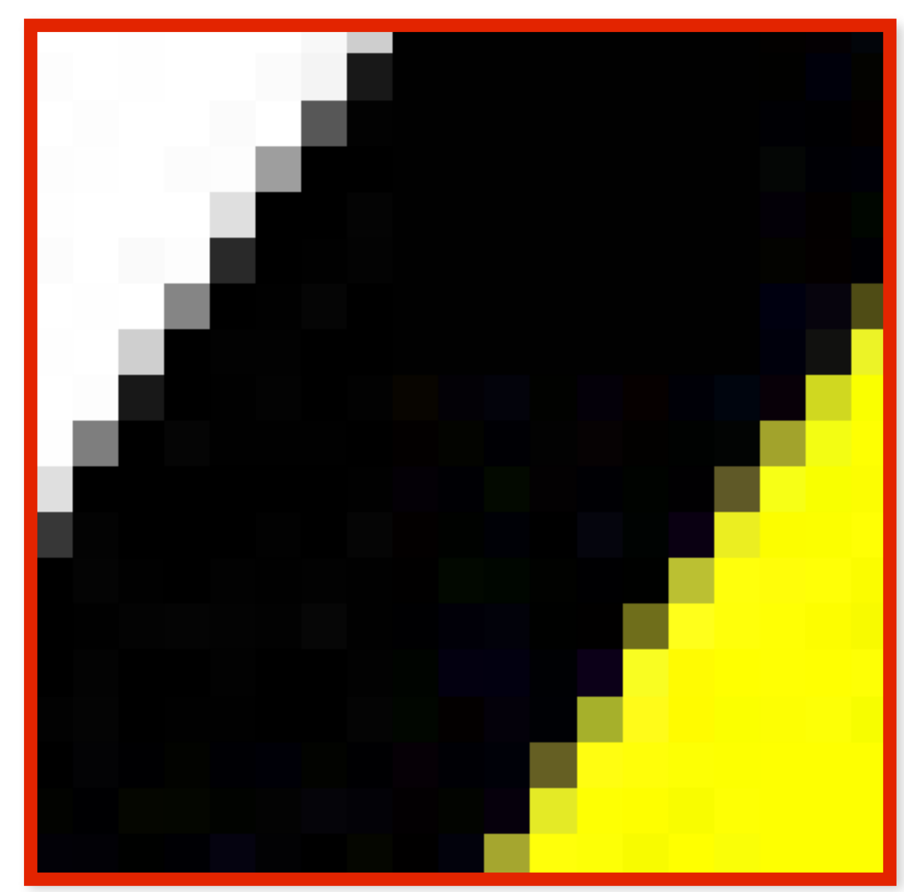


Image encoding in this example:

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

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

What is the compression ratio?



Per-block palette

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

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

- Can we do better?

S3TC (called BC1 or DXTC by Direct3D)

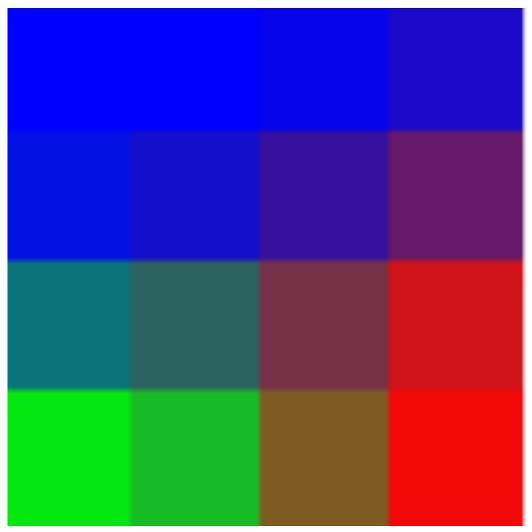
- **Palette of four colors encoded in four bytes:**
  - Two low-precision base colors: C_0 and C_1 (2 bytes each: RGB 5-6-5 format)
  - Other two colors computed from base values
    - $\frac{1}{3}C_0 + \frac{2}{3}C_1$
    - $\frac{2}{3}C_0 + \frac{1}{3}C_1$

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

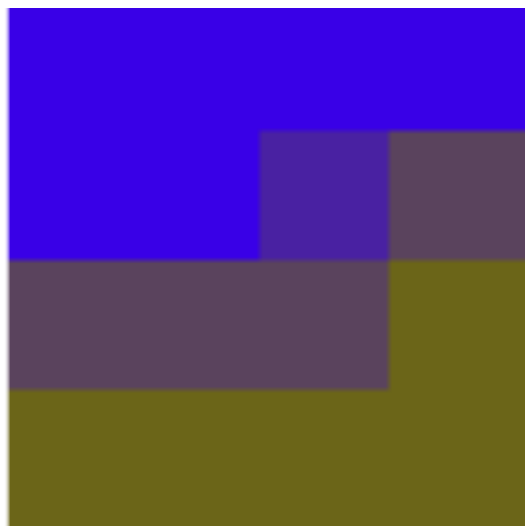
- **S3TC assumption:**
  - All texels in a 4x4 block lie on a line in RGB color space

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

S3TC artifacts



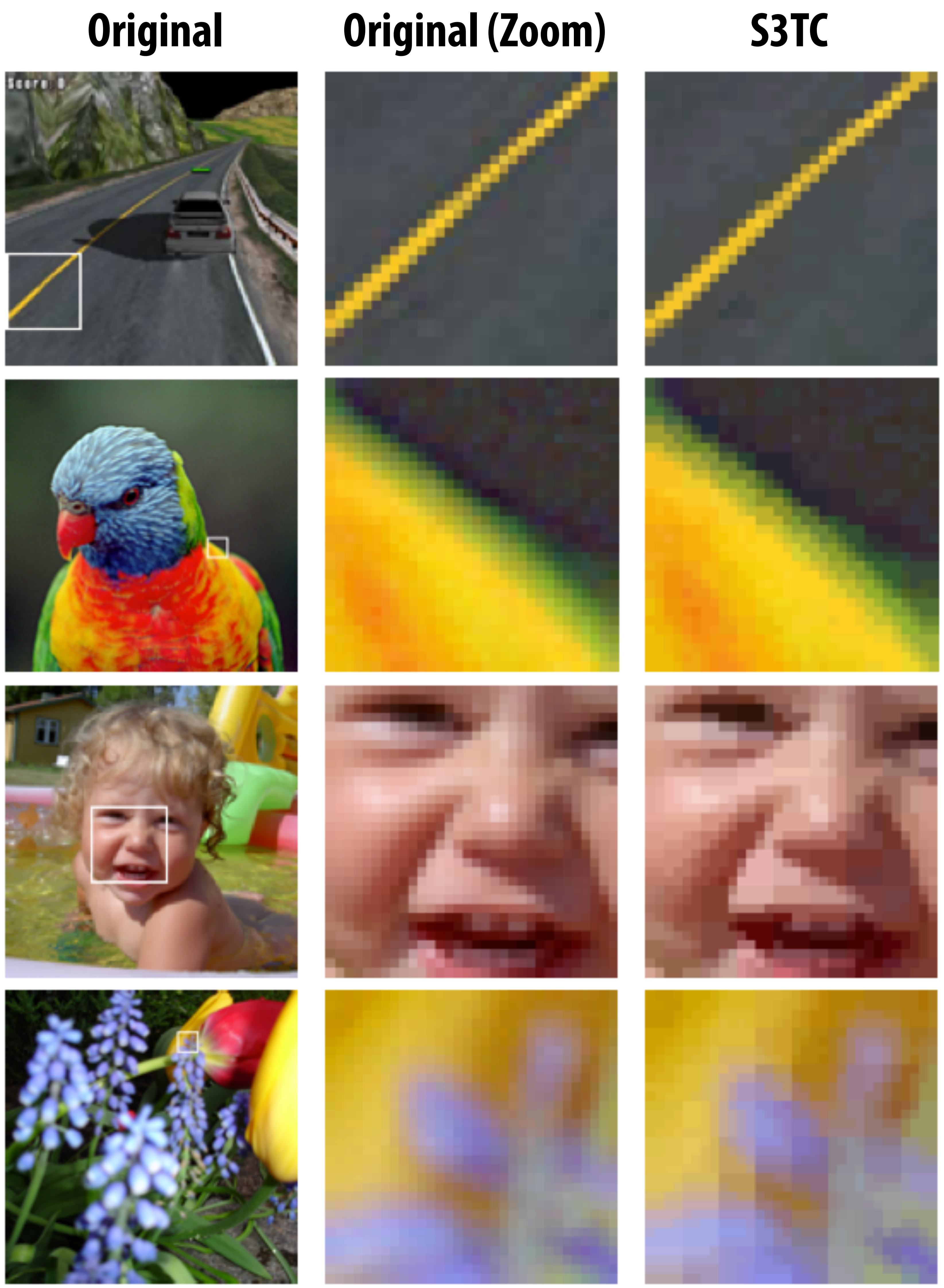
Original data



Compressed result

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

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



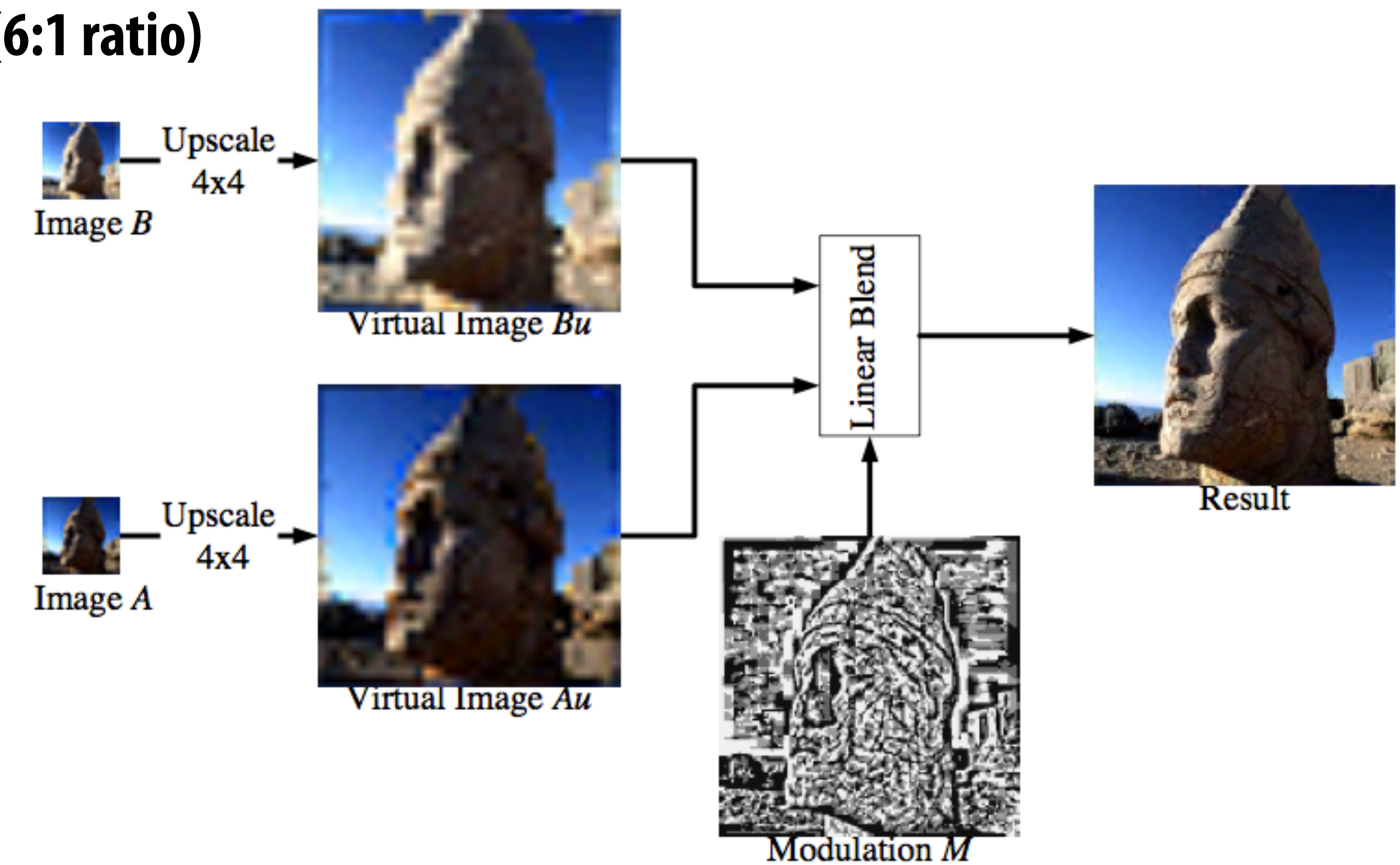
[Strom et al. 2007]

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

PVRTC (Power VR texture compression)

[Fenney et al. 2003]

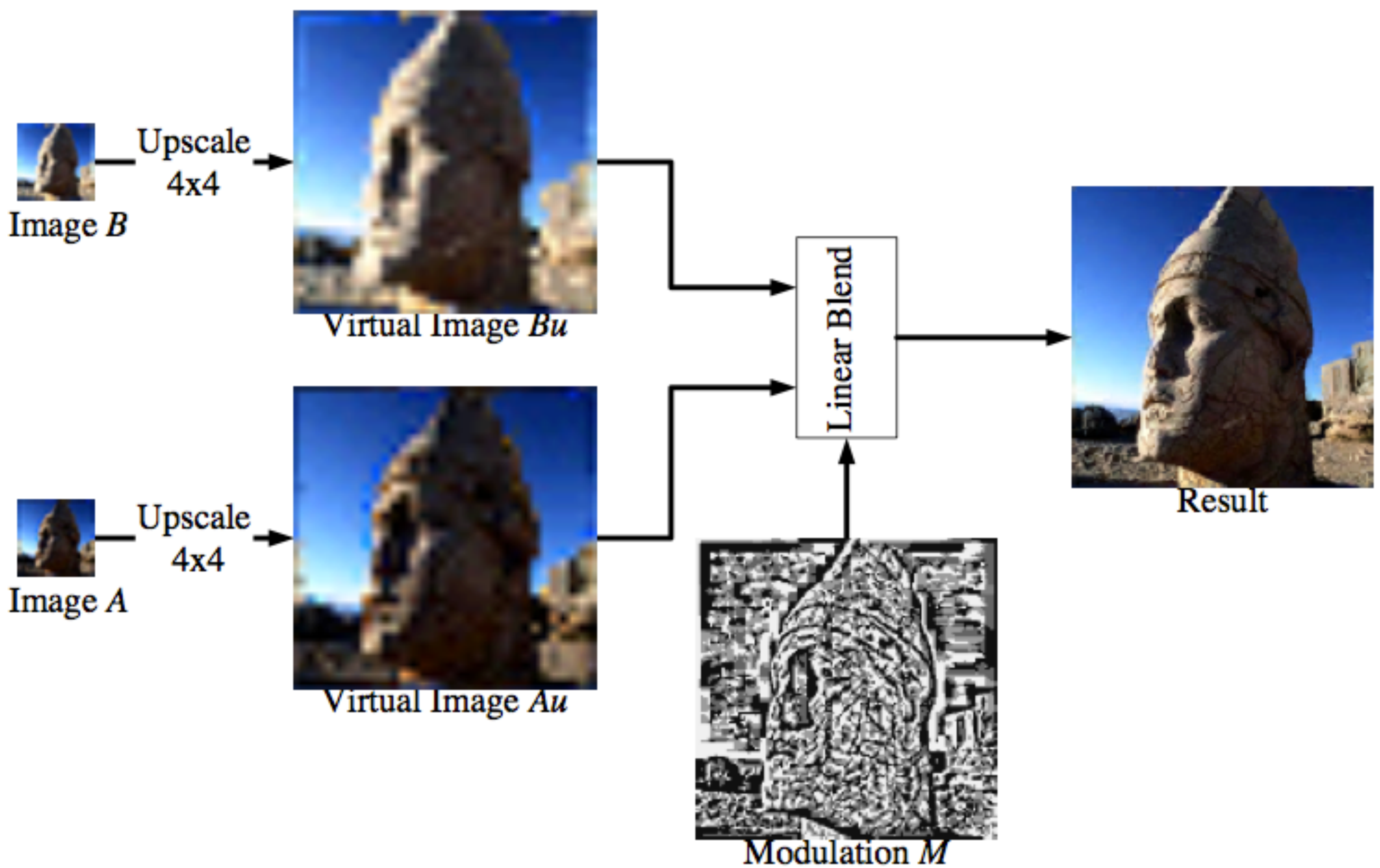
- Not a block-based format
  - Used in Imagination PowerVR GPUs
- Store low-frequency base images A and B
  - Base images downsampled by factor of 4 in each dimension ($^1/_{16}$ fewer texels)
  - Store base image pixels in RGB 5:5:5 format (+ 1 bit alpha)
- Store 2-bit modulation factor per texel
-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

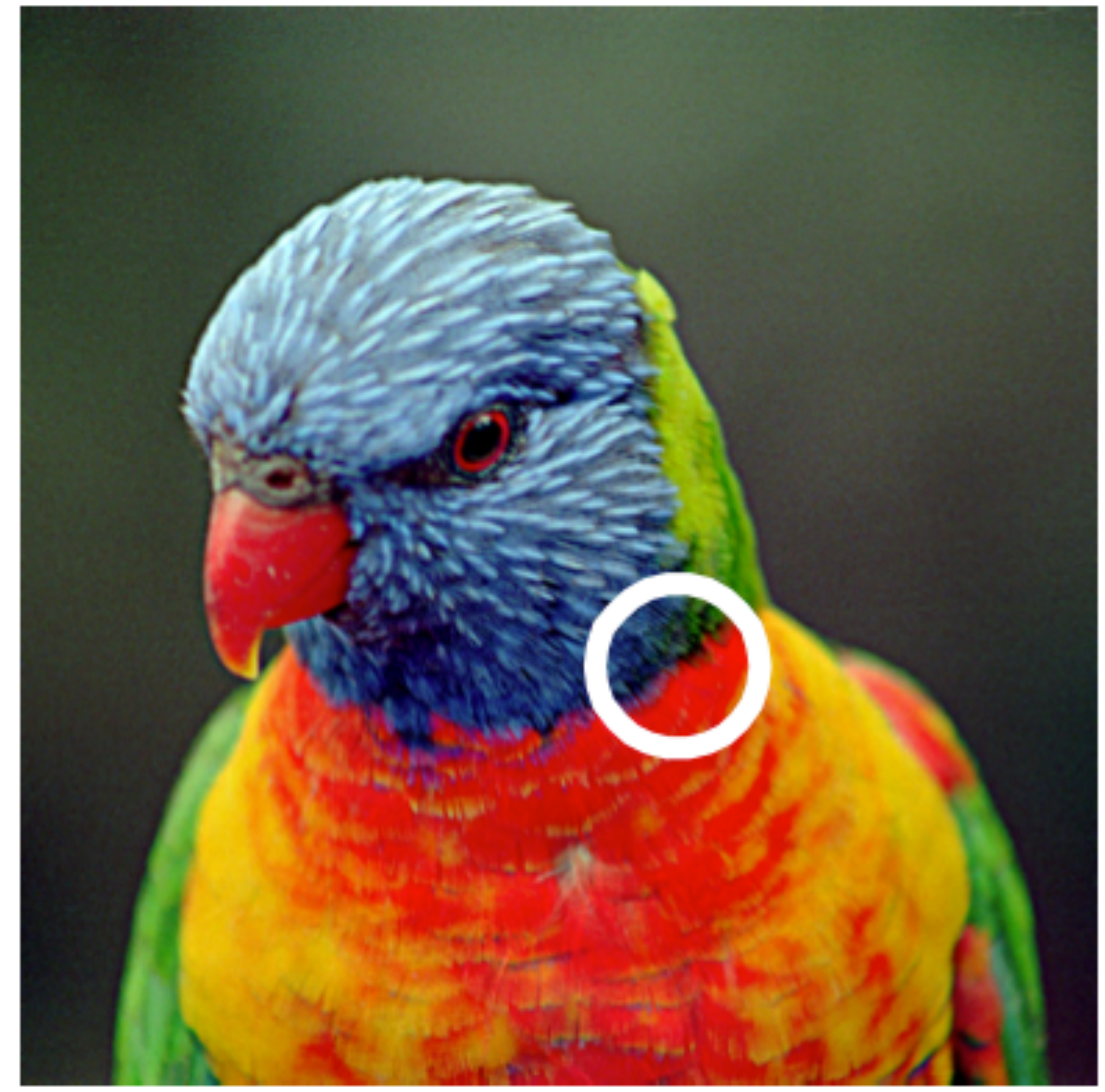


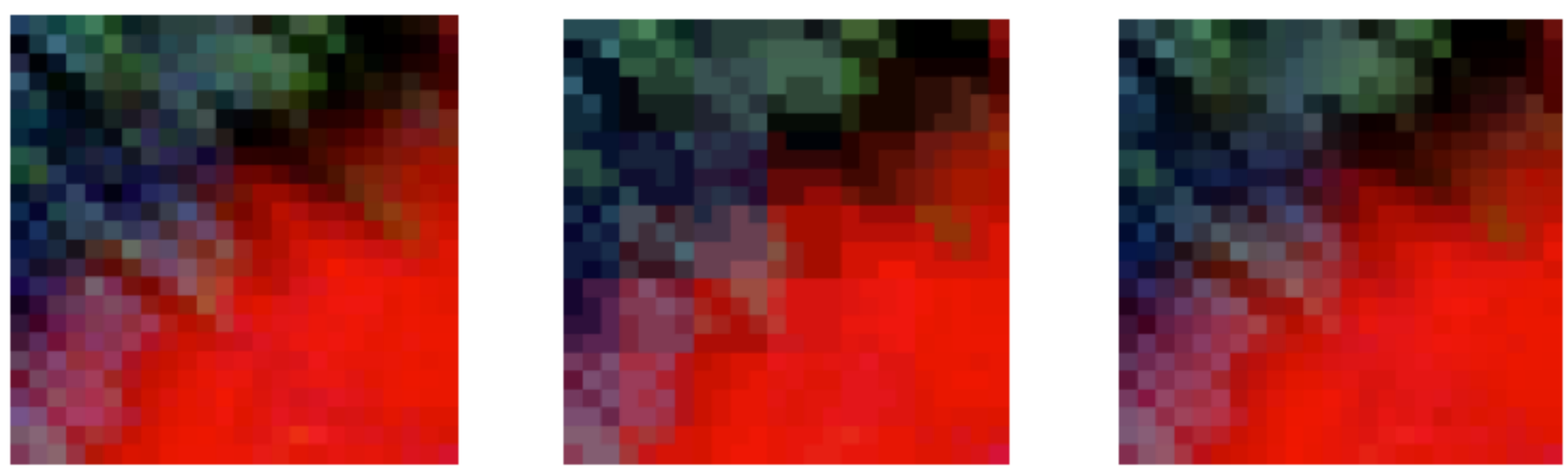
PVRTC avoids blocking artifacts

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)





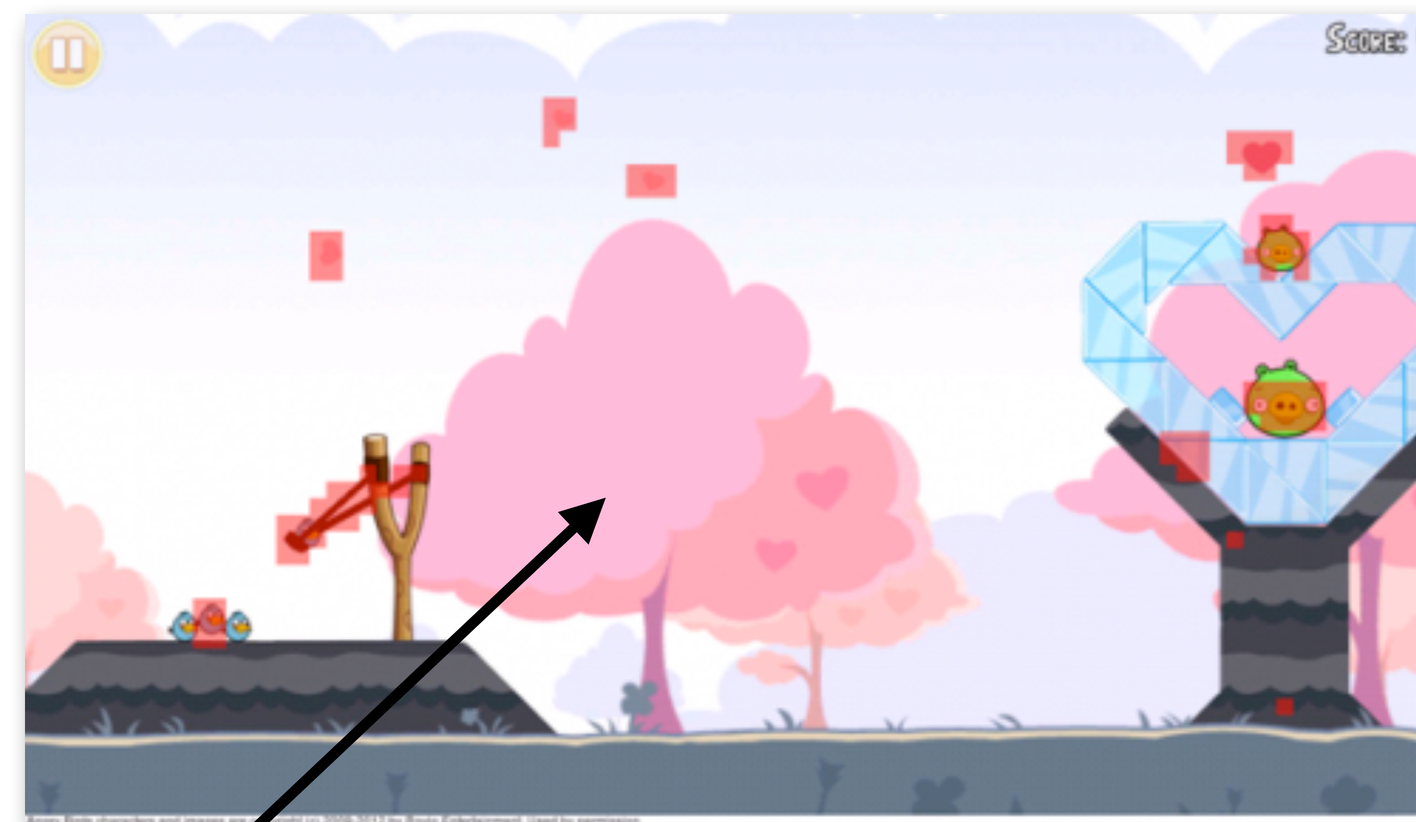
Original   S3TC   4bpp PVRTC

Image credit: Fenney et al. 2003

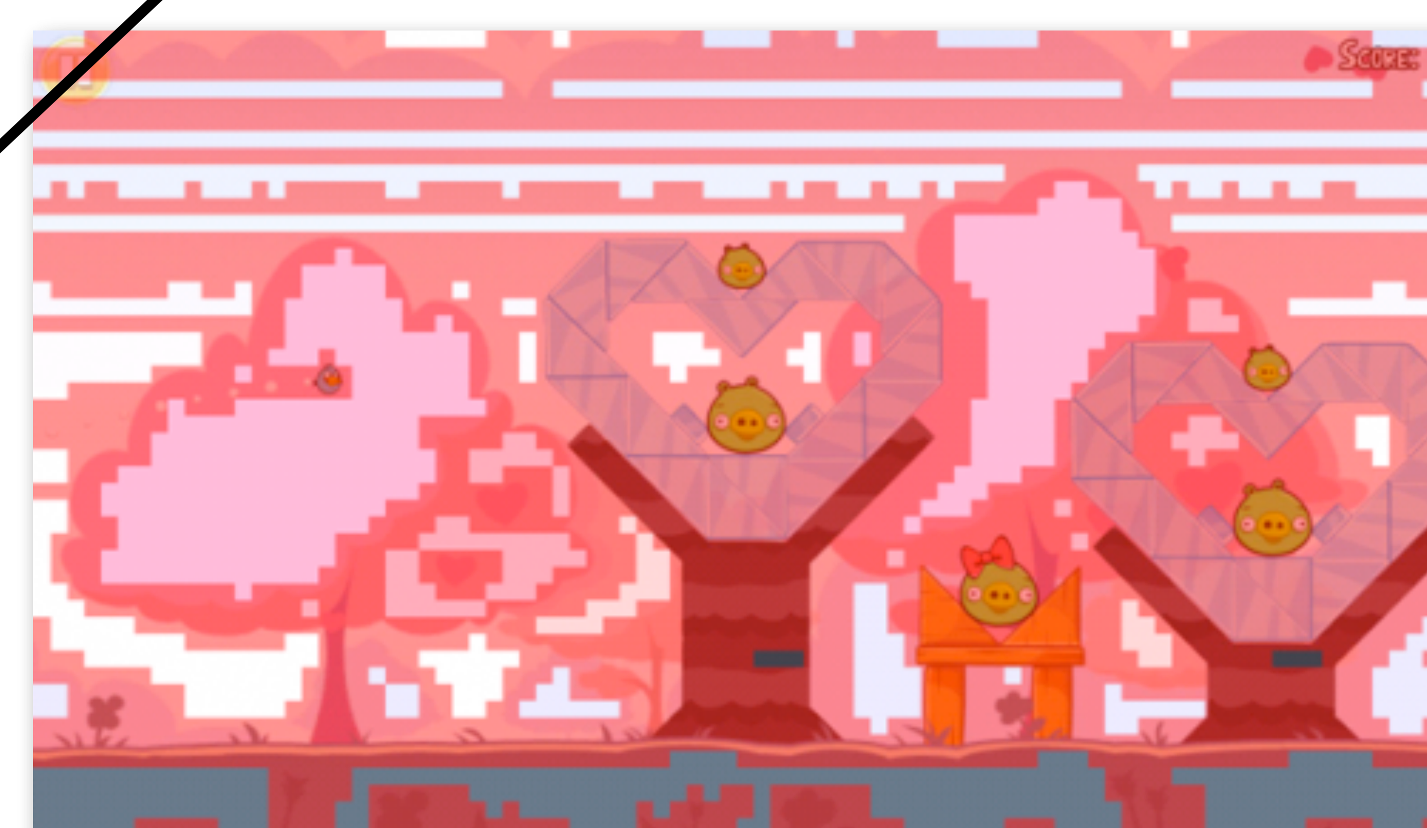
Mobile GPU architects go to many steps to reduce bandwidth to save power

- Compress texture data
- Compress frame buffer
- Eliminate unnecessary memory writes!

- Frame 1:
  - Render frame as normal
  - 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's contents are the same as in the last frame
    - If yes, skip memory write



Slow camera motion: 96% of writes avoided
Fast camera motion: ~50% of writes avoided
(red tile = required a memory write)



[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

- 3D graphics implementations are highly optimized for power efficiency
  - Tiled rendering for bandwidth efficiency *
  - Deferred rendering to reduce shading costs
  - Many additional optimizations such as buffer compression, eliminating unnecessary memory ops, etc.

- If you enjoy these topics, consider CS348K (Visual Computing Systems)

* Not all mobile GPUs use tiled rendering as described in this lecture.