Lecture 25:
Parallel Micropolygon Rendering

CMU 15-418: Parallel Computer Architecture and Programming (Spring 2012)

Announcements

- Please fill out course and TA evaluations for us
- Exam 2 will be returned on Thursday
- Parallelism competition
  - Thursday May 10th, 8:30-11:30 AM
  - 5-7 minute presentations per group
  - Judges:
    - Matt Pharr (Intel)
    - Ron Babich (NVIDIA Research)
  - Will make your project pages available to Matt and Ron on Monday May 7th

What you should know

- Pay attention to how I describe the graphics algorithms in this talk
  - How do I describe the algorithm? (inputs, outputs)
  - How do I describe the workload? (type of parallelism, locality, dependencies)
  - What are the challenges in each of the subproblems?
  - How were they overcome?
- Consider the end-to-end system
  - Complex systems have many interesting interactions
  - Component X's behavior also makes life easier in component Y
  - Changed algorithms to get better parallel behavior (obtain different results)
- That graphics is awesome

Reminder: GPU programmable core

NVIDIA Fermi Core

32-wide SIMD

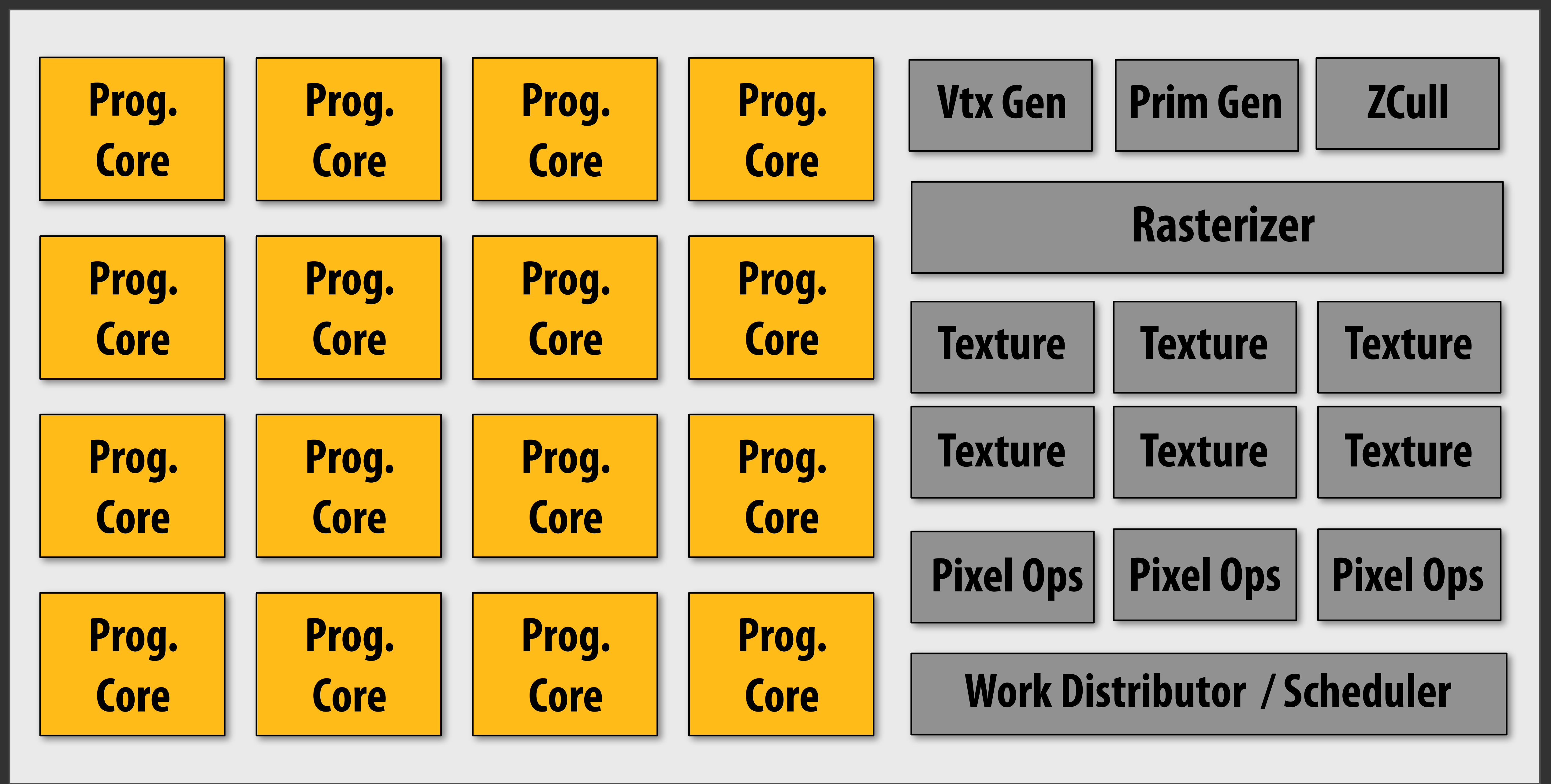
48 interleaved instruction streams

64 KB scratchpad/L1

- Wide SIMD processing
- HW multi-threading
- Small traditional cache + software-managed scratchpad

Needs data-parallelism: more than 1500 elements processed by core at once!

Reminder: heterogeneous, multi-core GPU



NVIDIA Fermi GPU

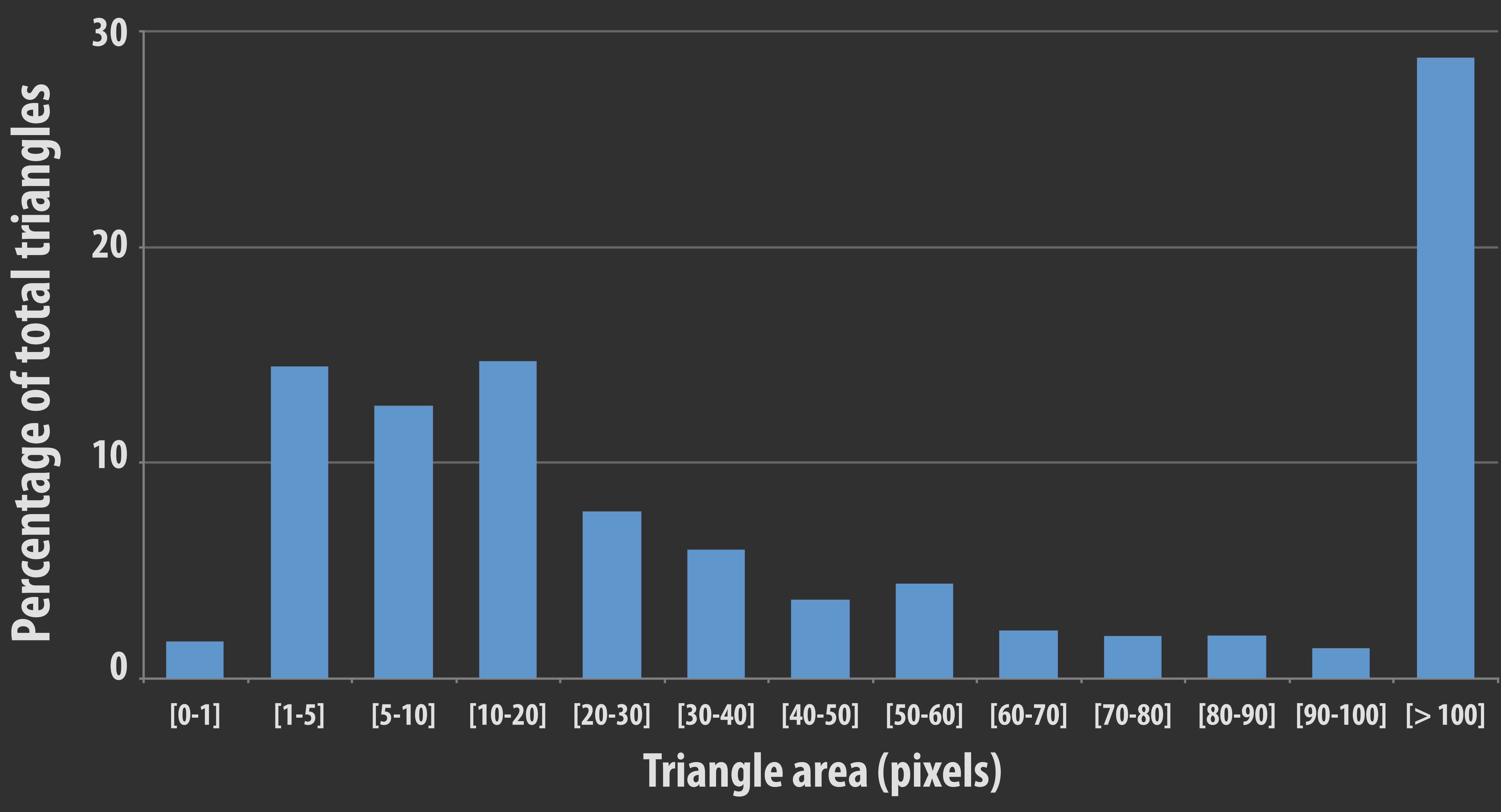
16 programmable cores: ~ 1.5 TFLOPS

+ fixed-function processing specific to graphics

Interactive graphics: low geometric detail

adidas
Credit: Pro Evolution Soccer 2010 (Konami)

Interactive graphics uses large triangles

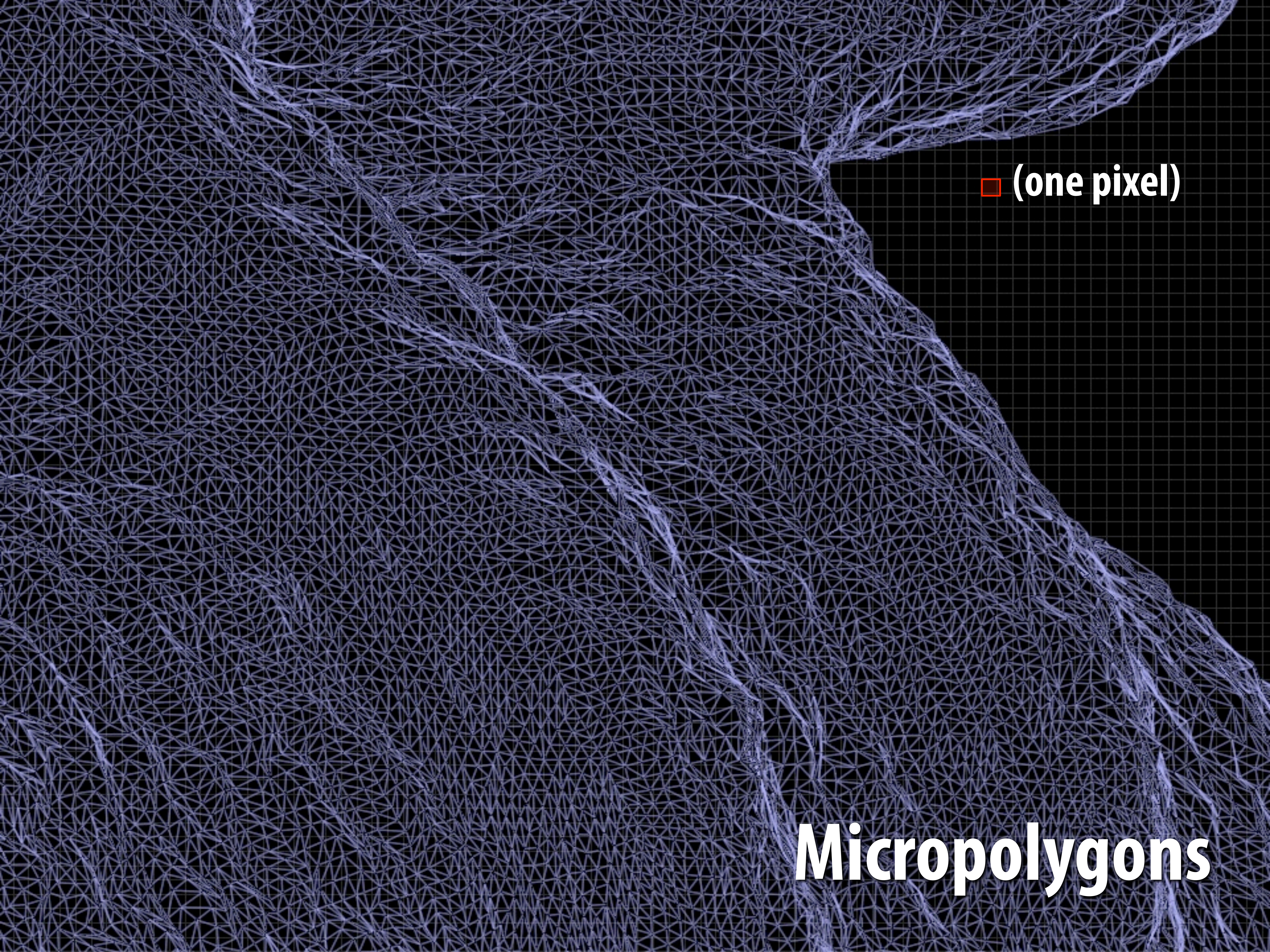


[source NVIDIA]

Highly detailed surfaces

Highly detailed surfaces

Credit: Pixar Animation Studios, UP (2009)


(one pixel)
Micropolygons

Assertion:

It is inefficient to render micropolygons using the OpenGL/Direct3D graphics pipeline implemented by GPUs.

Sources of inefficiency

Tessellation

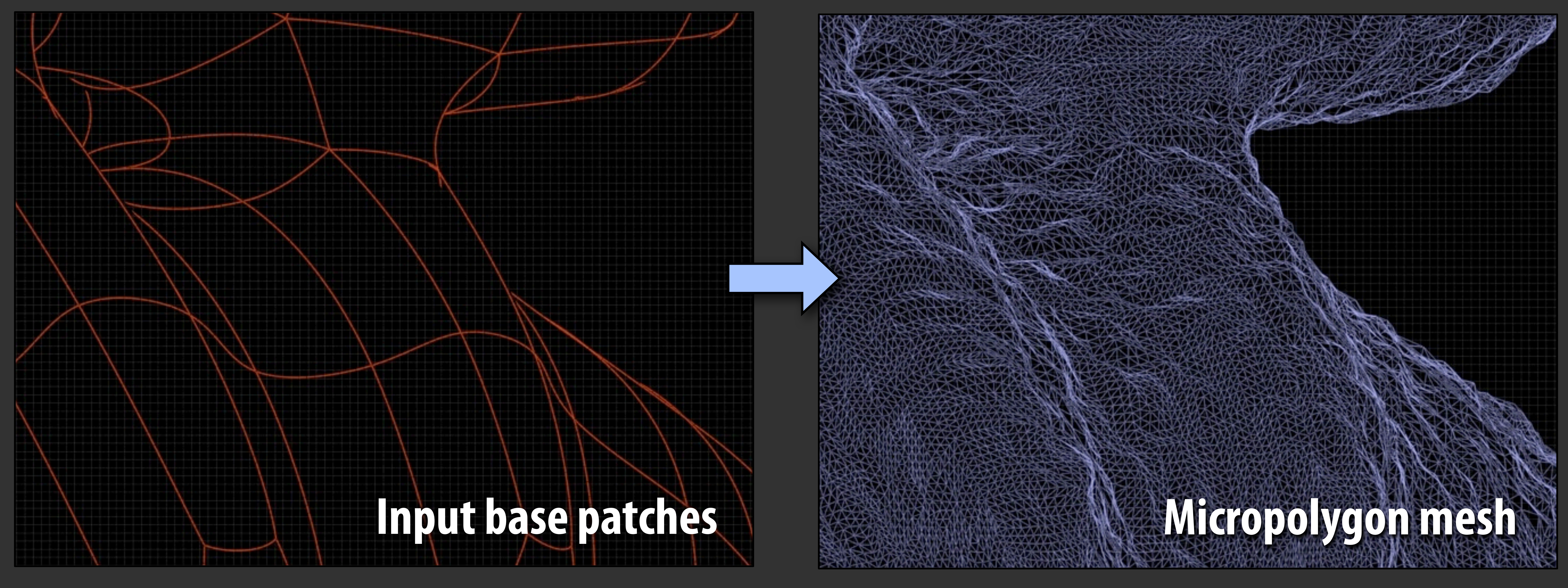
(generating geometry)

Rasterization

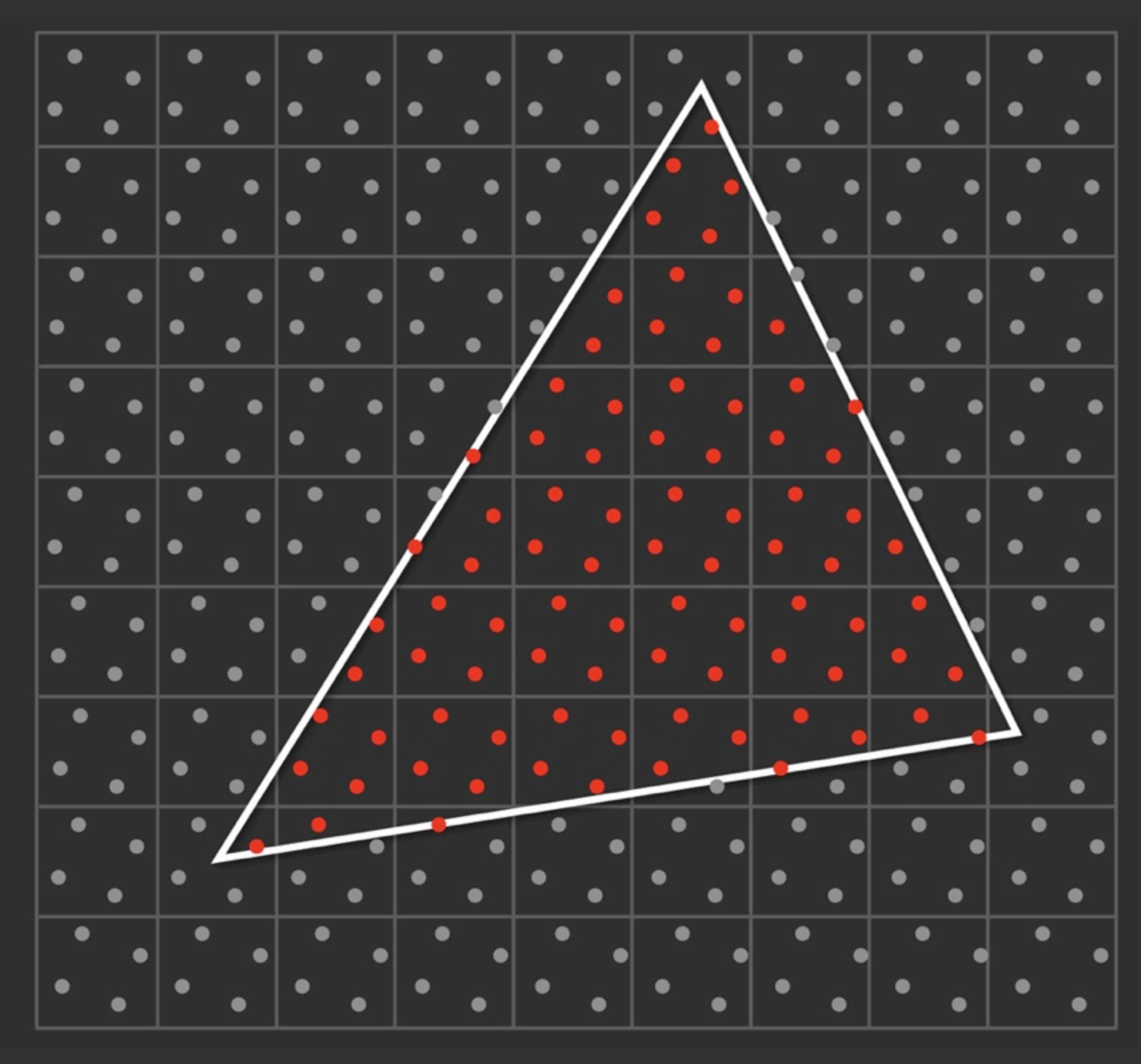
Shading

Missing: adaptive tessellation

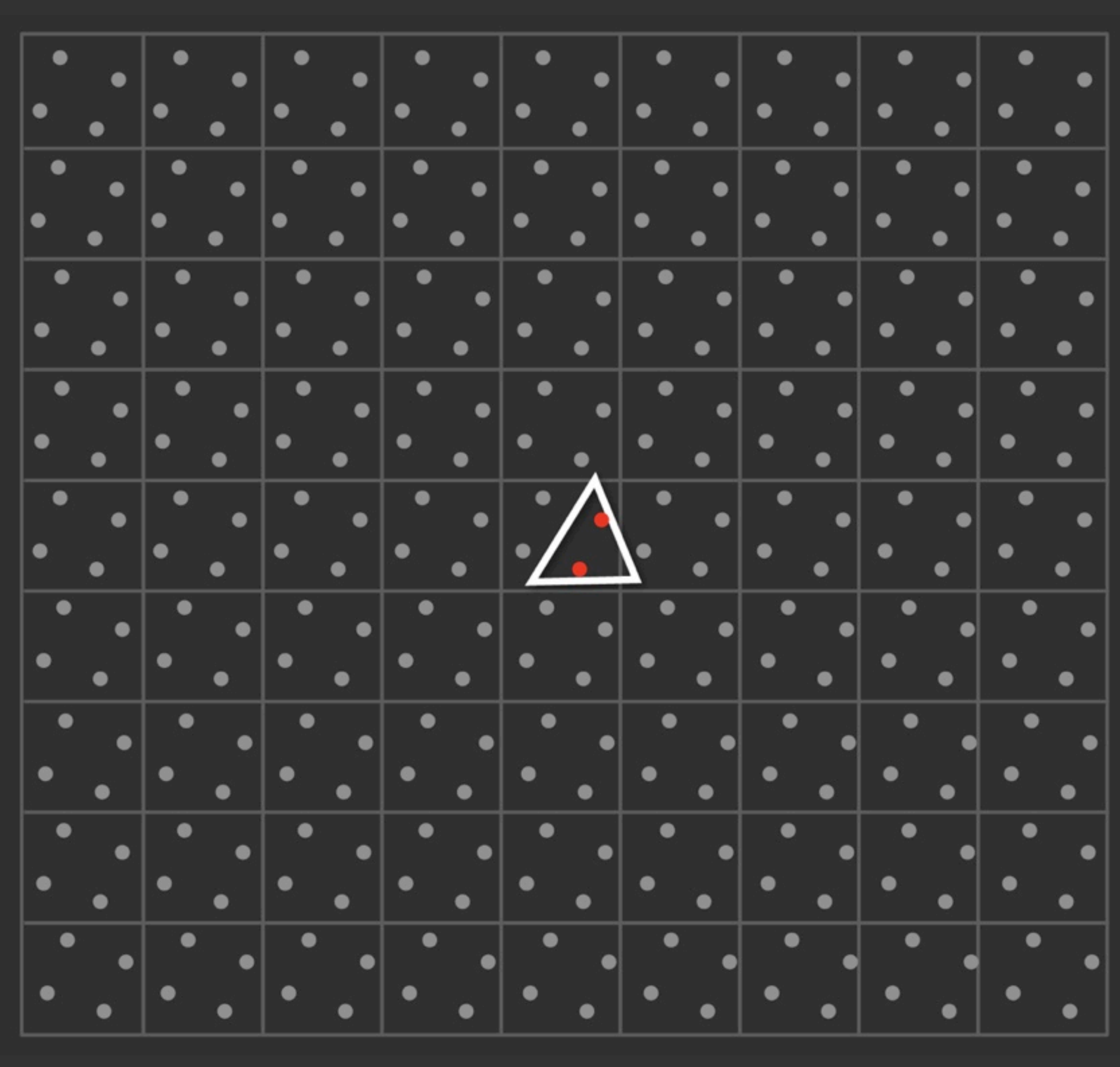
Generate triangles on-demand in the pipeline



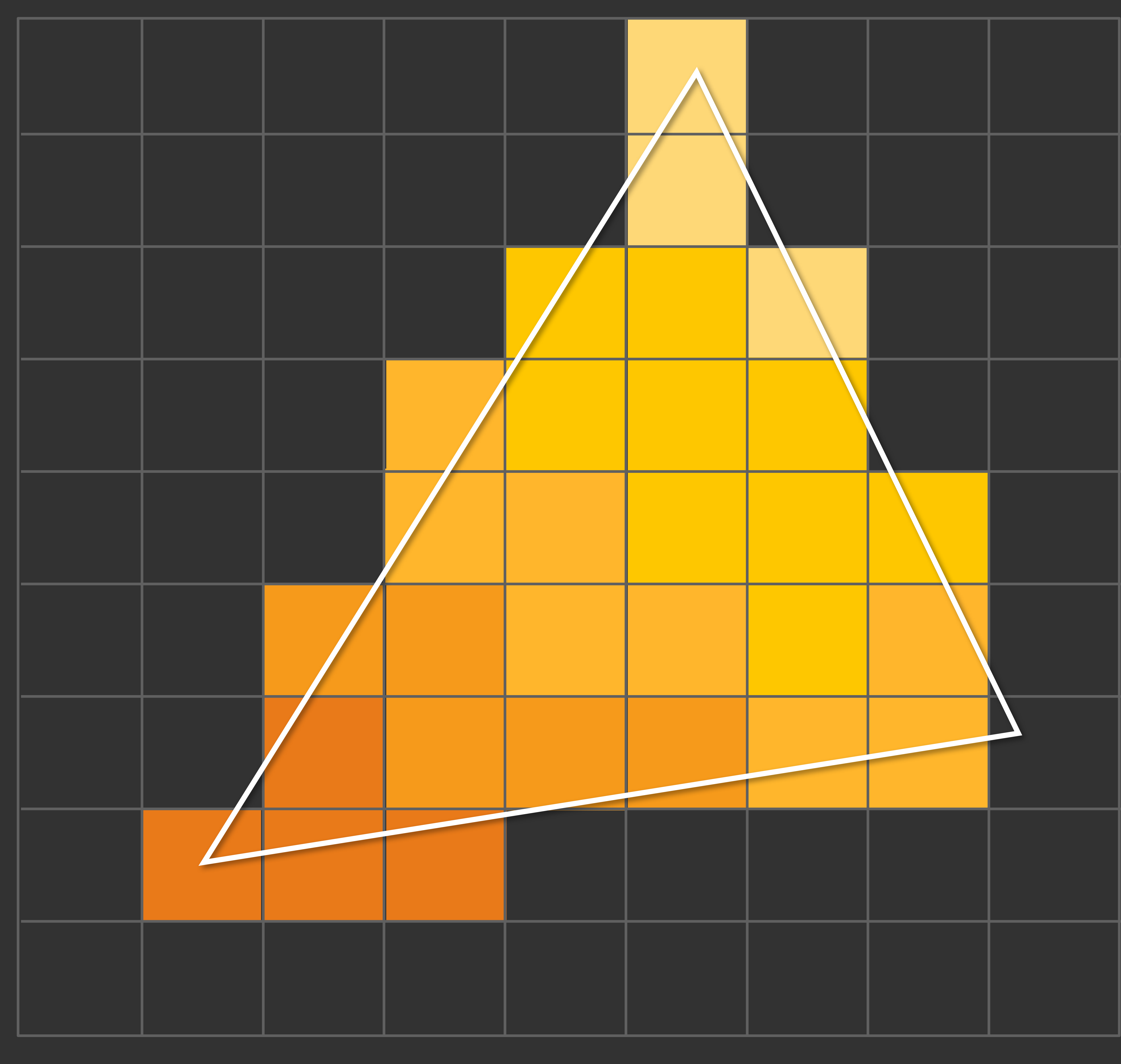
Rasterization: computing covered pixels



Micropolygons too small for pixel-parallelism



Shading: computing surface color



Micropolygons pose three big problems

TESSELLATION

Cannot adaptively tessellate a surface into micropolygons in parallel.

RASTERIZATION

Pixel-parallel coverage tests are inefficient.

SHADING

Pipeline generates over 8x more shading work than needed.

TESSELLATION:

Integrating parallel, adaptive tessellation into the pipeline

Overview: current solutions

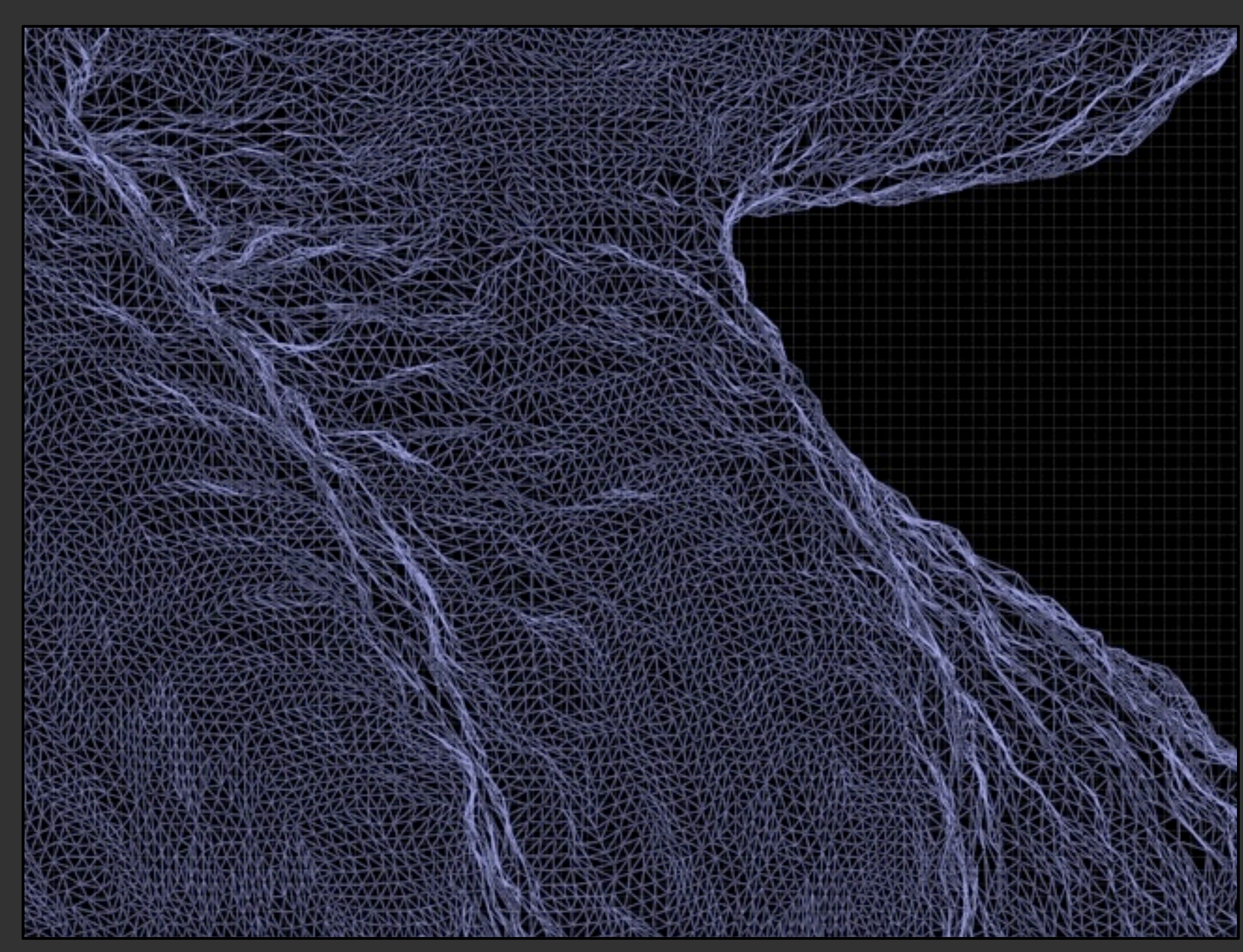
- Lane-Carpenter patch algorithm [Lane 80]
  - High-quality, adapts well to surface complexity
  - Hard to parallelize
- GPU tessellation
  - Low quality, does not adapt well [Moreton 01, Direct3D 11]
  - High performance (parallel, fixed-function)

Tessellation input: parametric patches

Input base patches
(example: bicubic patch)

[Vlachos 01, Loop 08, Loop 09]

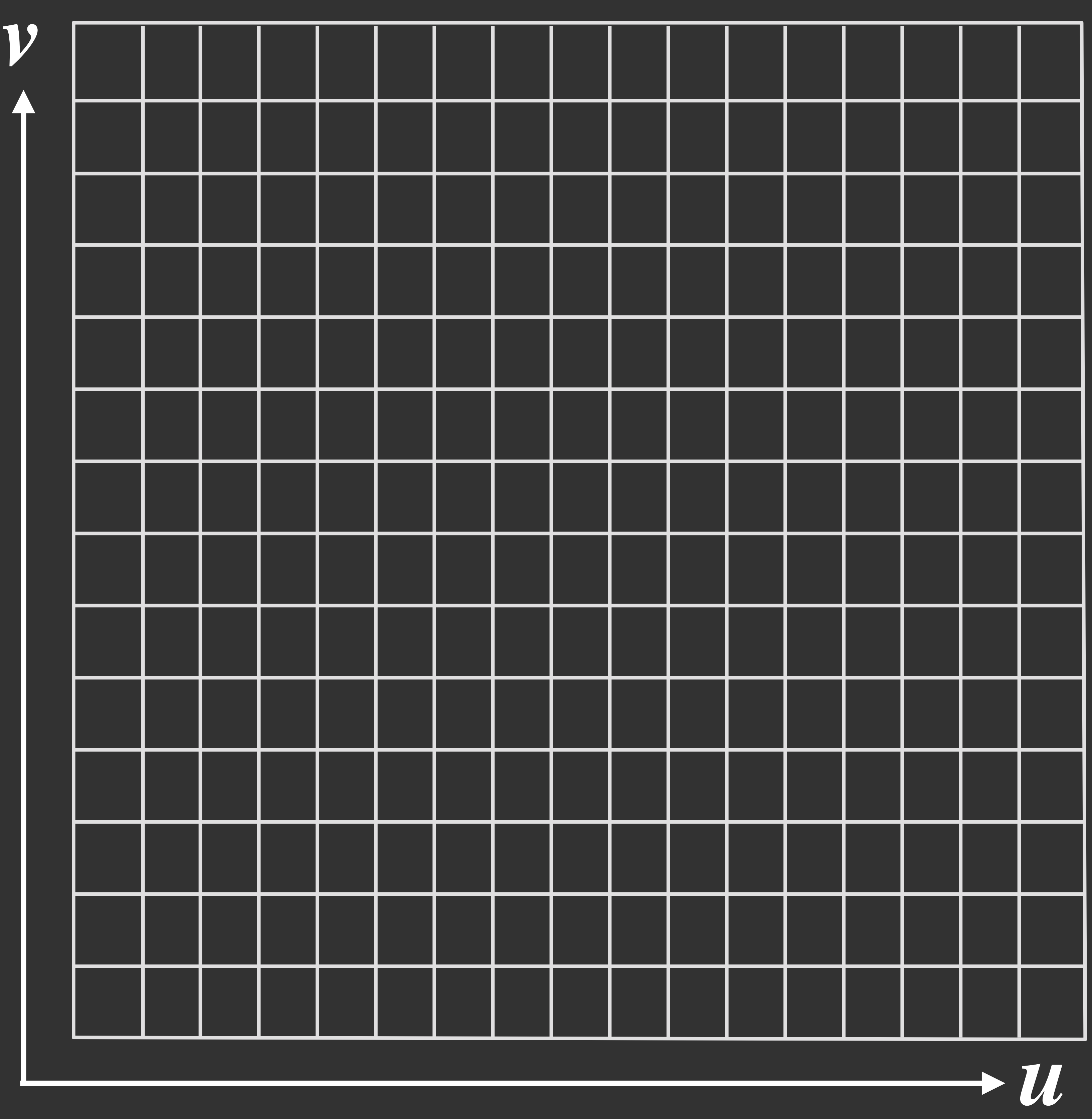
Tessellation output: micropolygon mesh



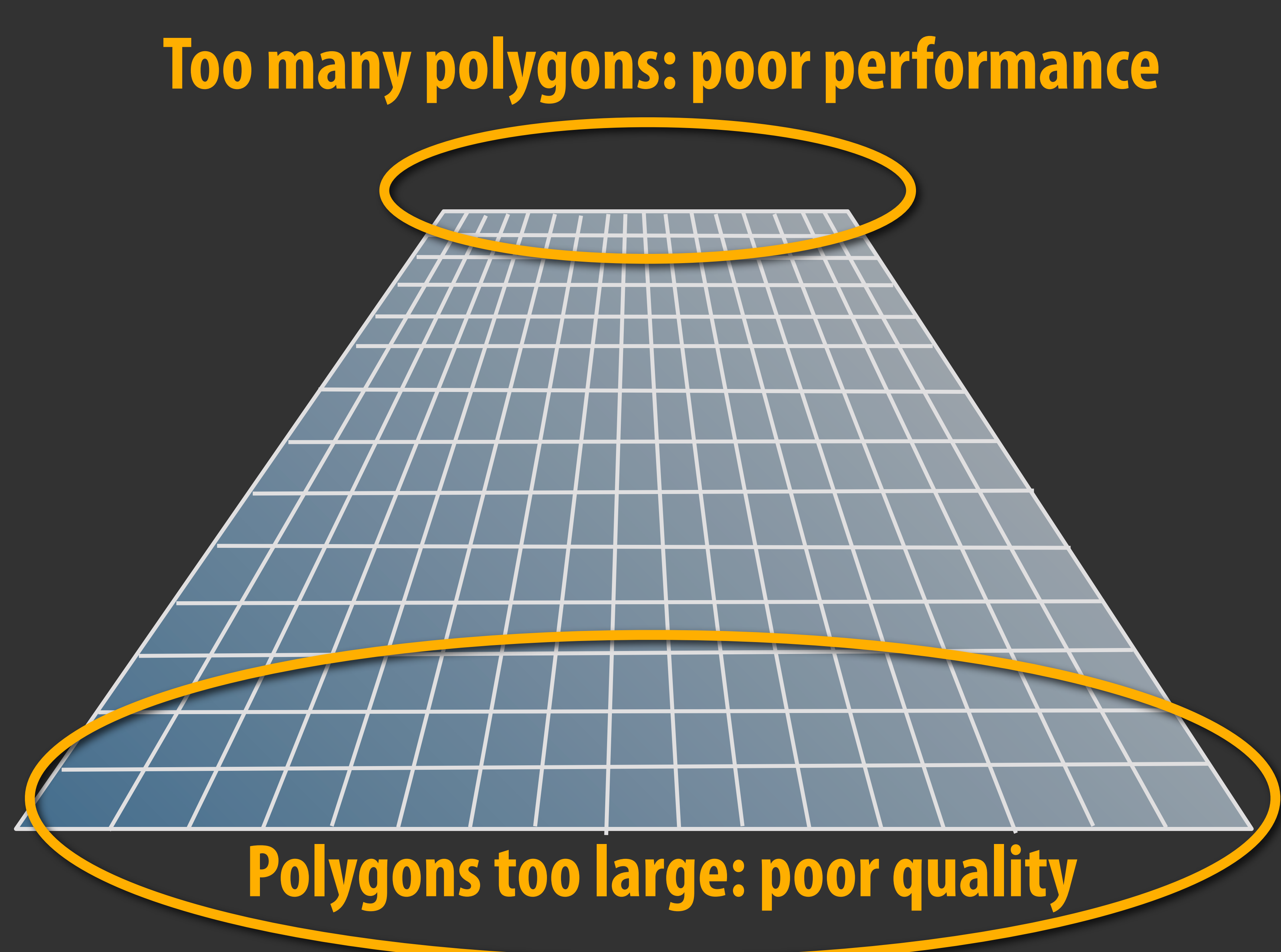
Goal: all triangles are approximately 1/2 pixel in area

(yields about one vertex per pixel)

Uniform patch tessellation is insufficient

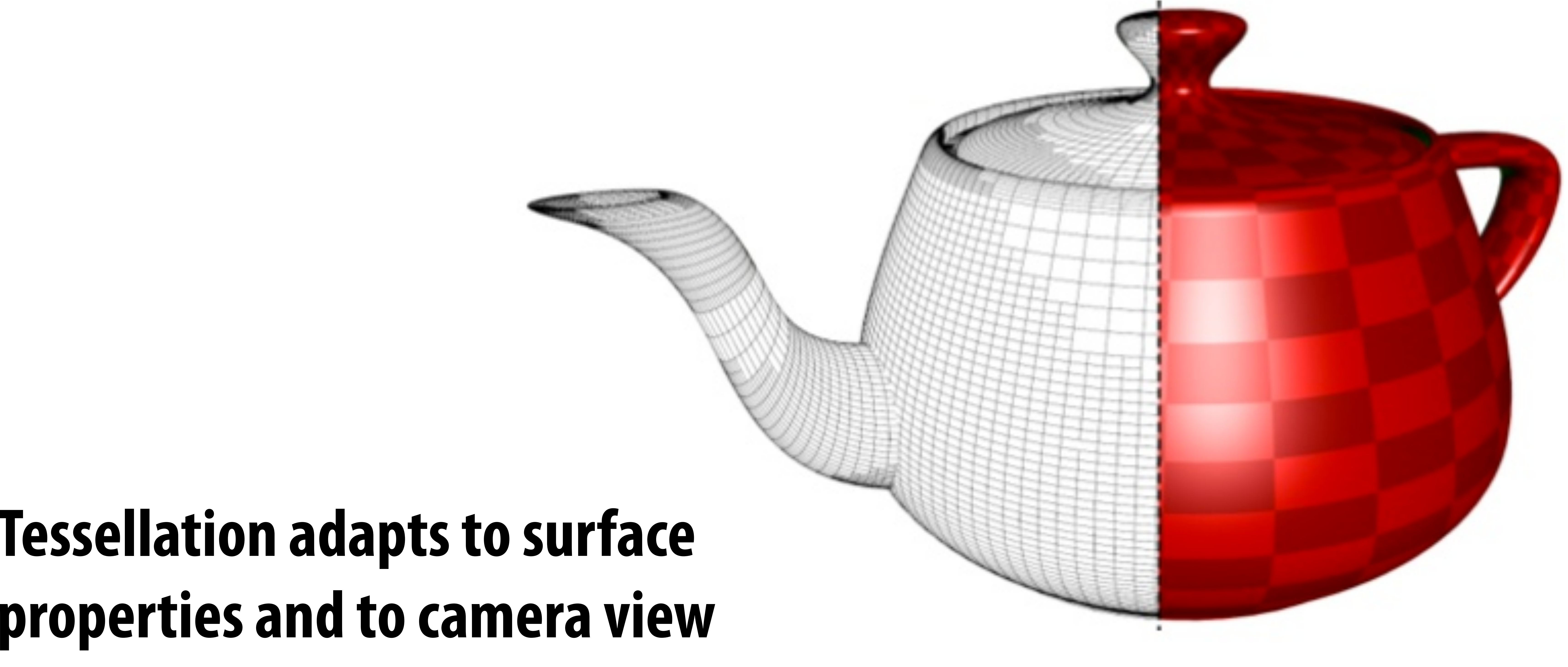


Uniform partitioning of patch (parametric domain)



Patch viewed from camera

Adaptive tessellation

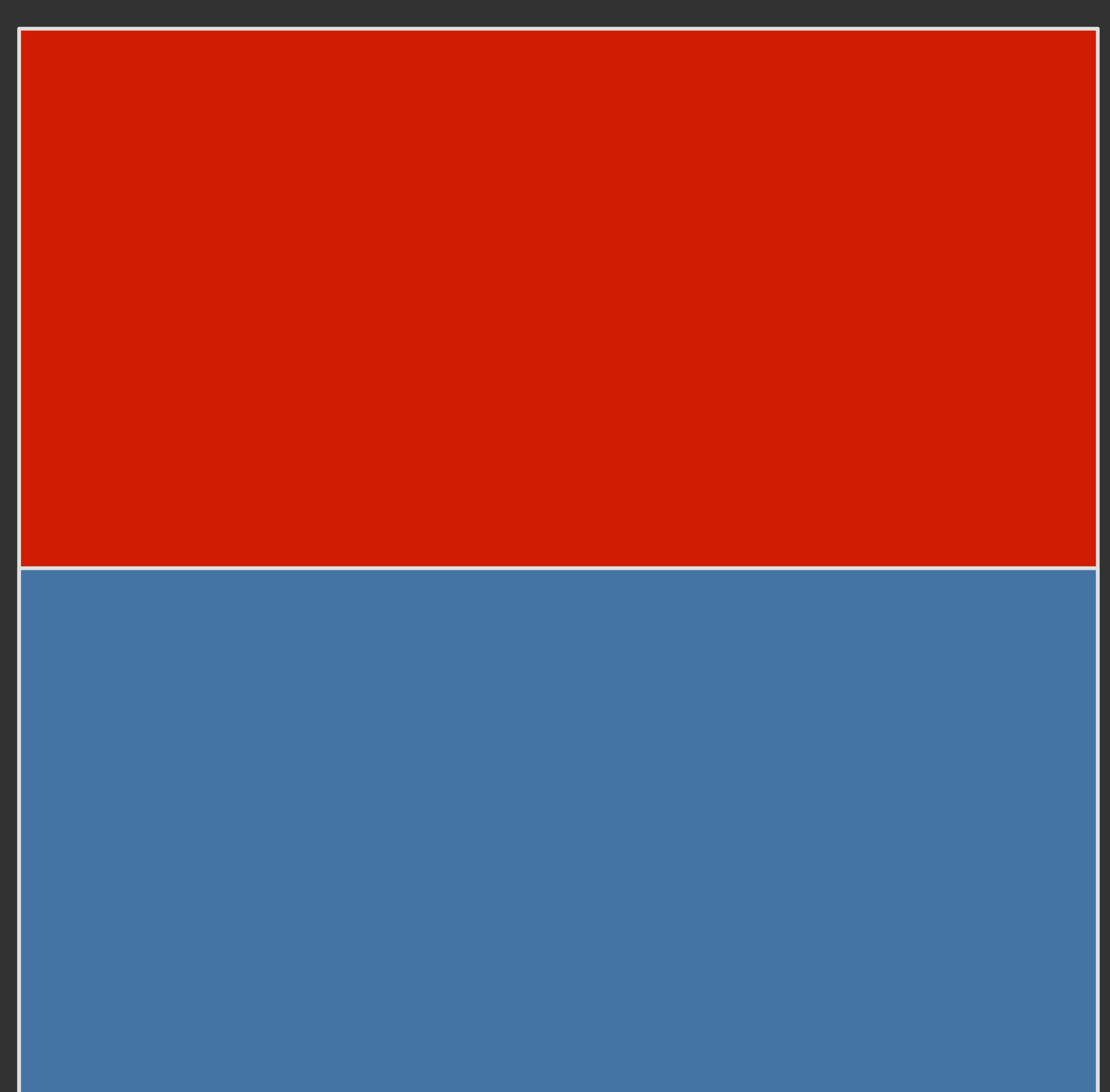


Tessellation adapts to surface properties and to camera view

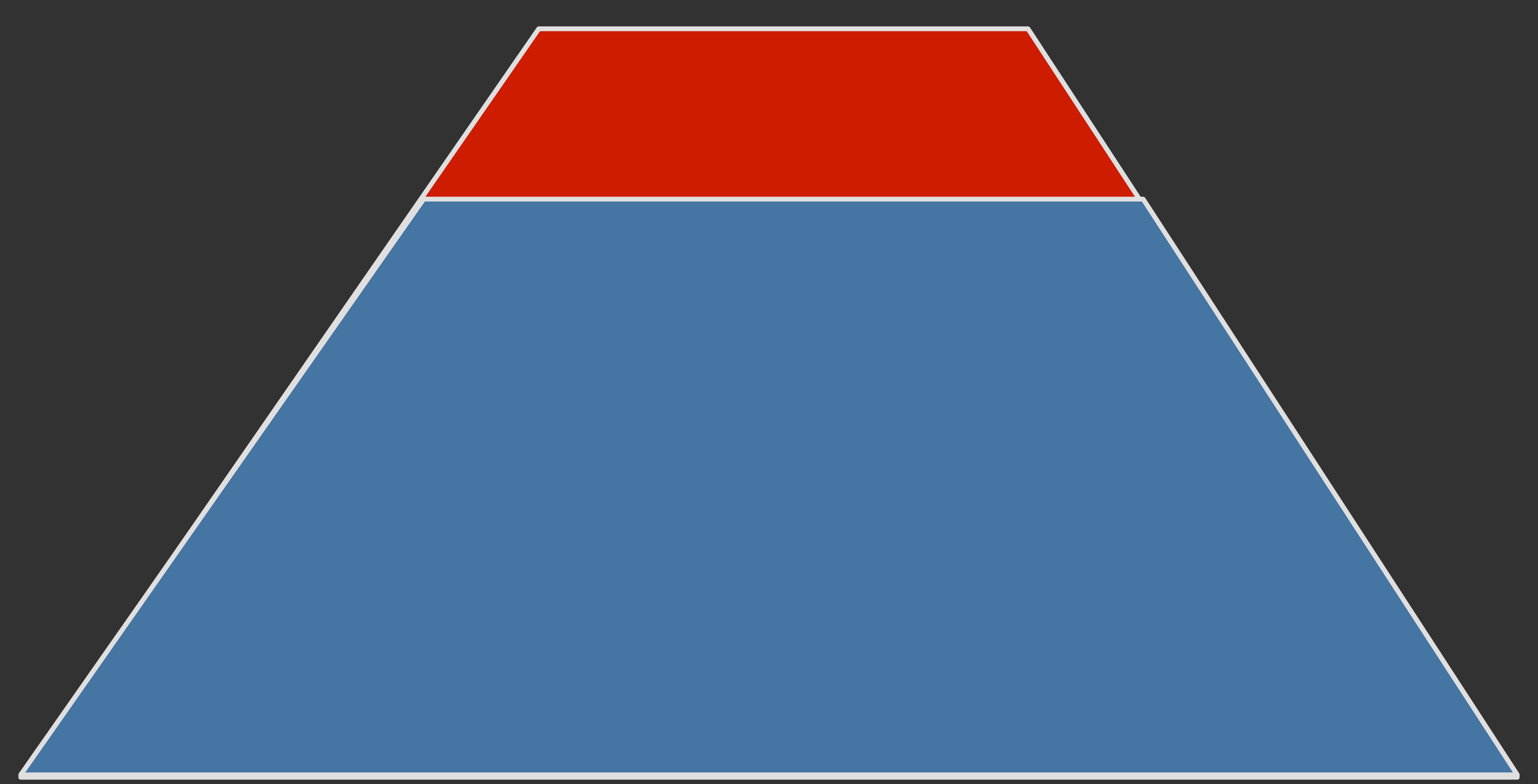
Notice: larger polygons approximate flatter areas of surface well

Adaptive tessellation

(Lane-Carpenter patch algorithm)

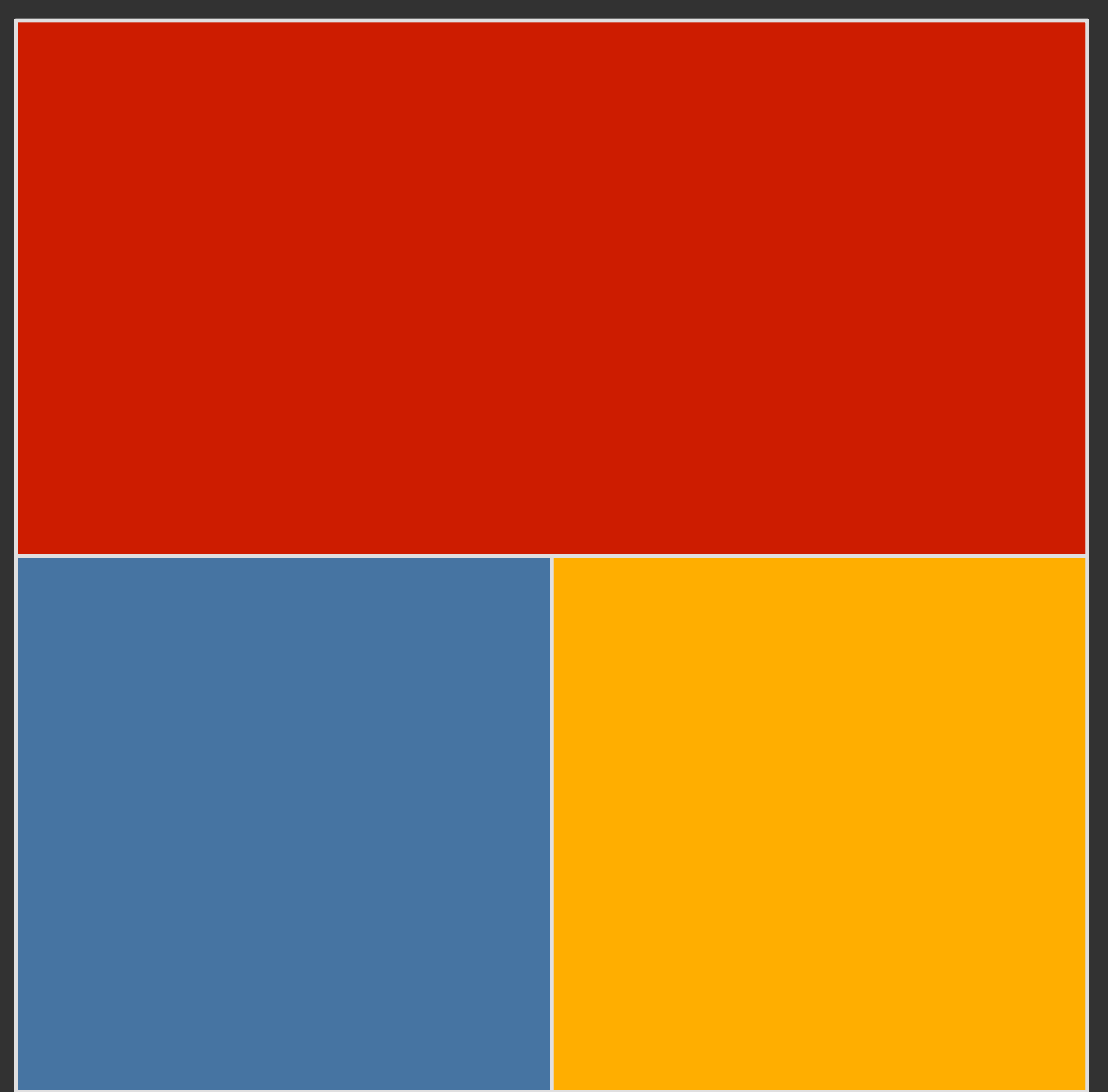


Patch parametric domain

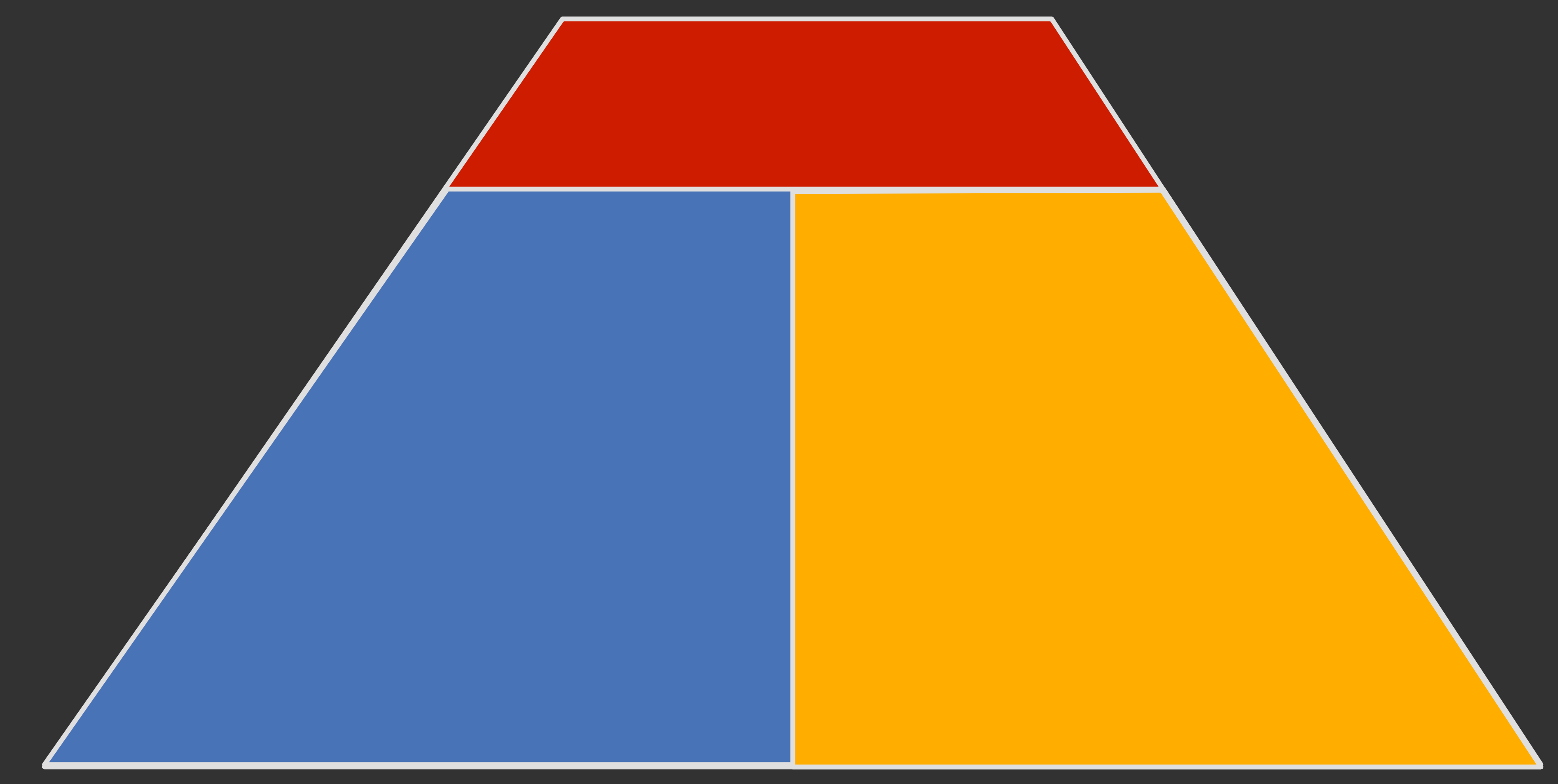


Patch viewed from camera

Adaptive tessellation
(Lane-Carpenter patch algorithm)



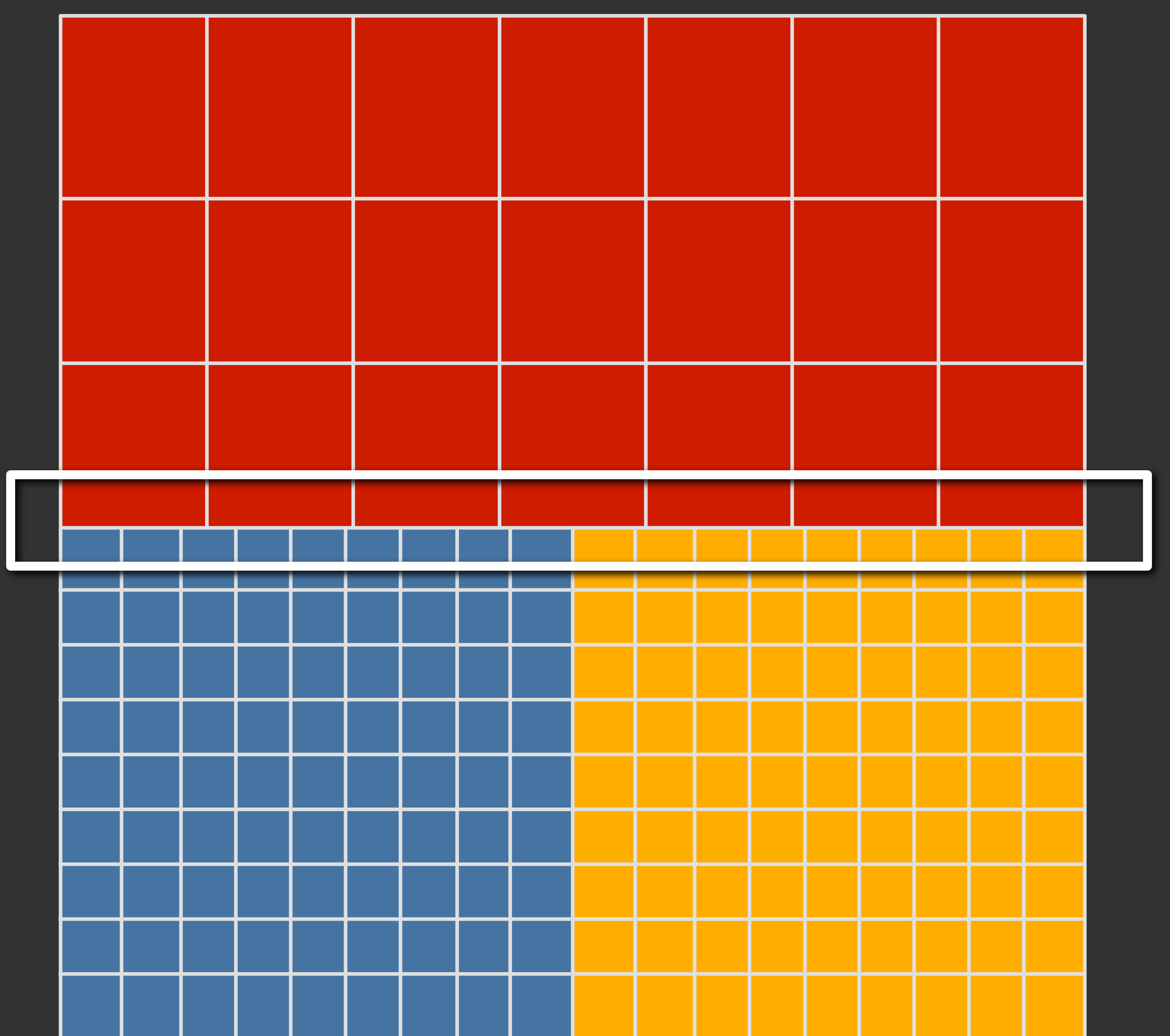
Patch parametric domain



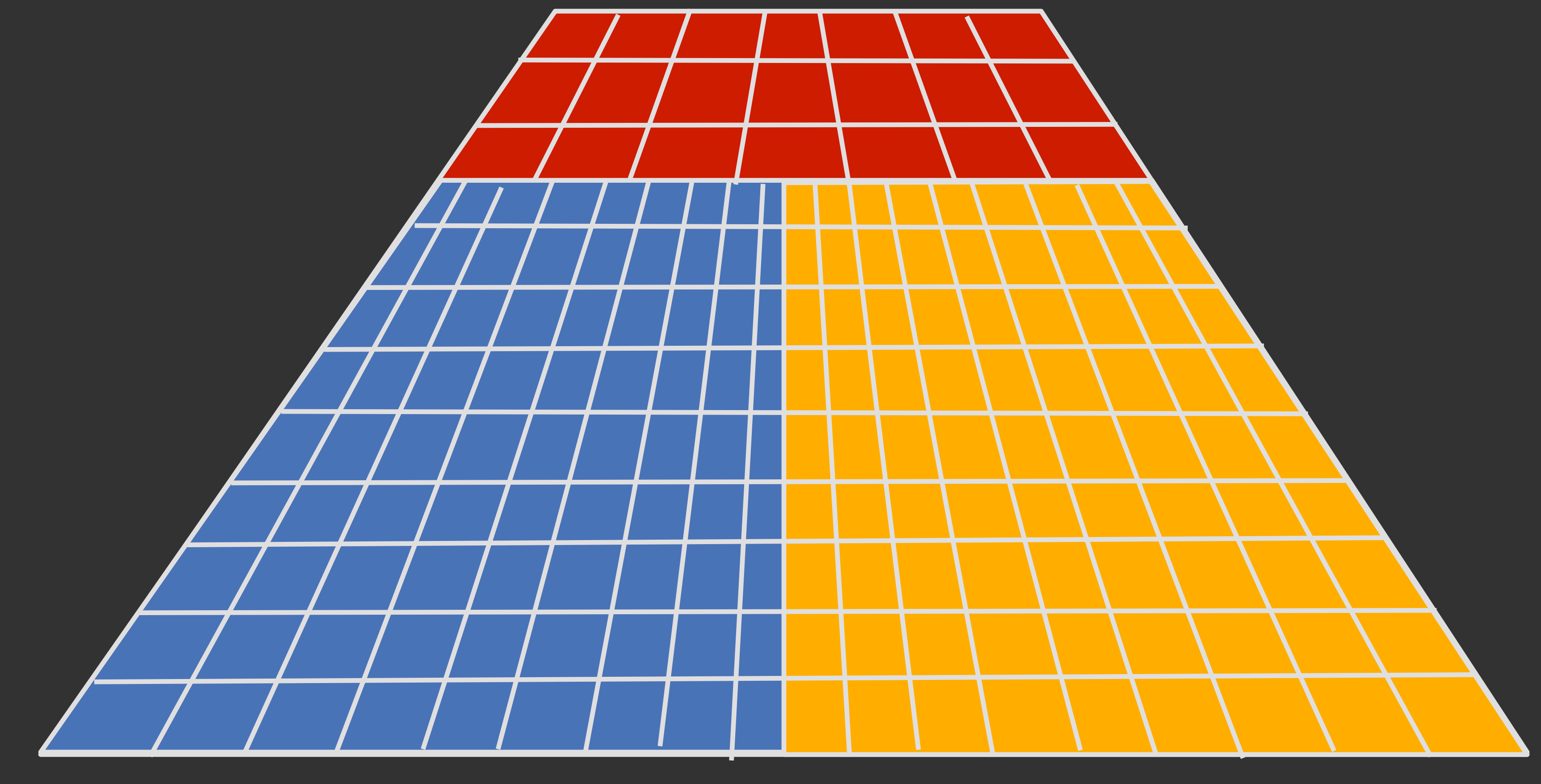
Patch viewed from camera

Adaptive tessellation

(Lane-Carpenter patch algorithm)

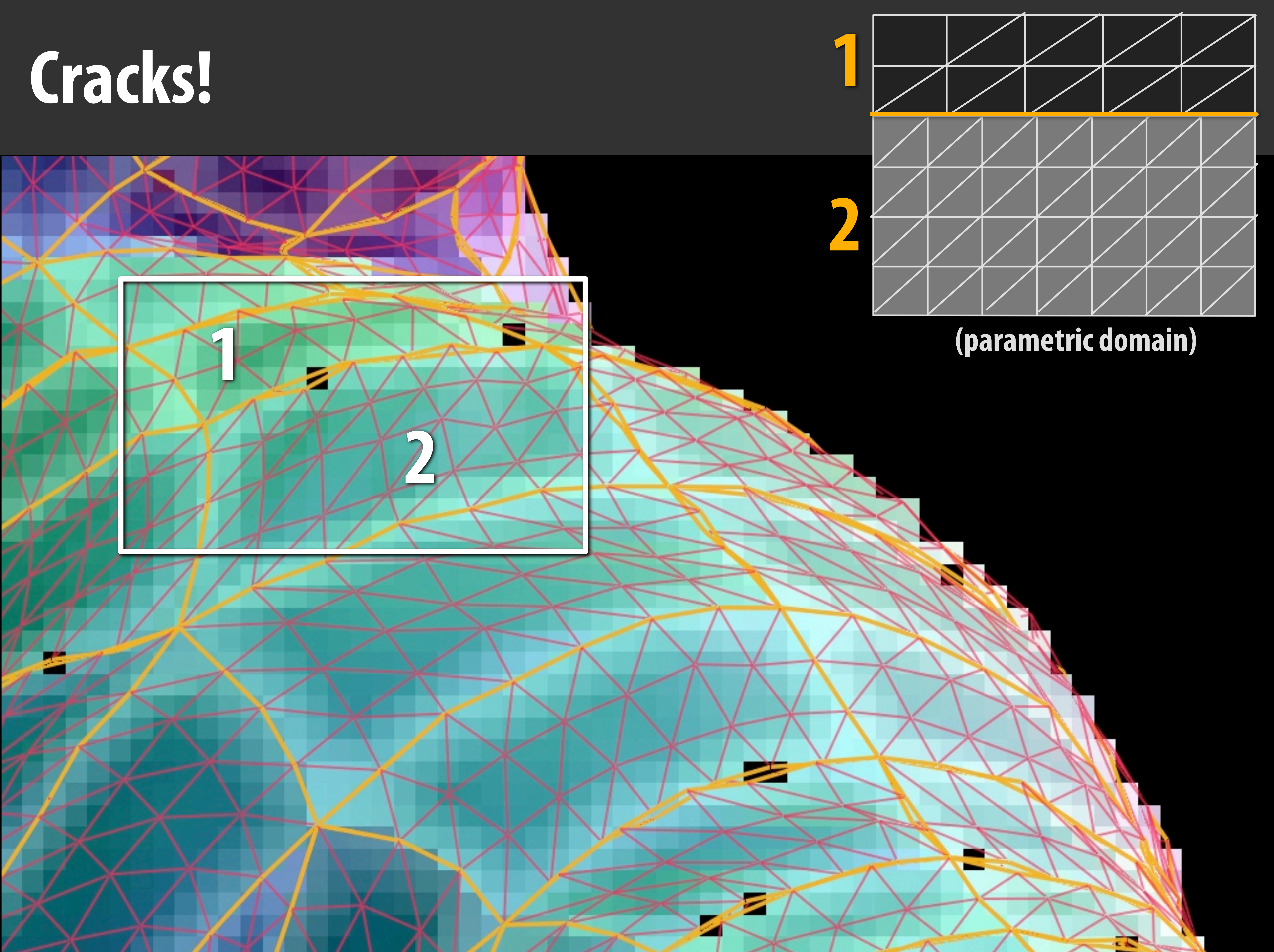


Patch parametric domain



Patch viewed from camera

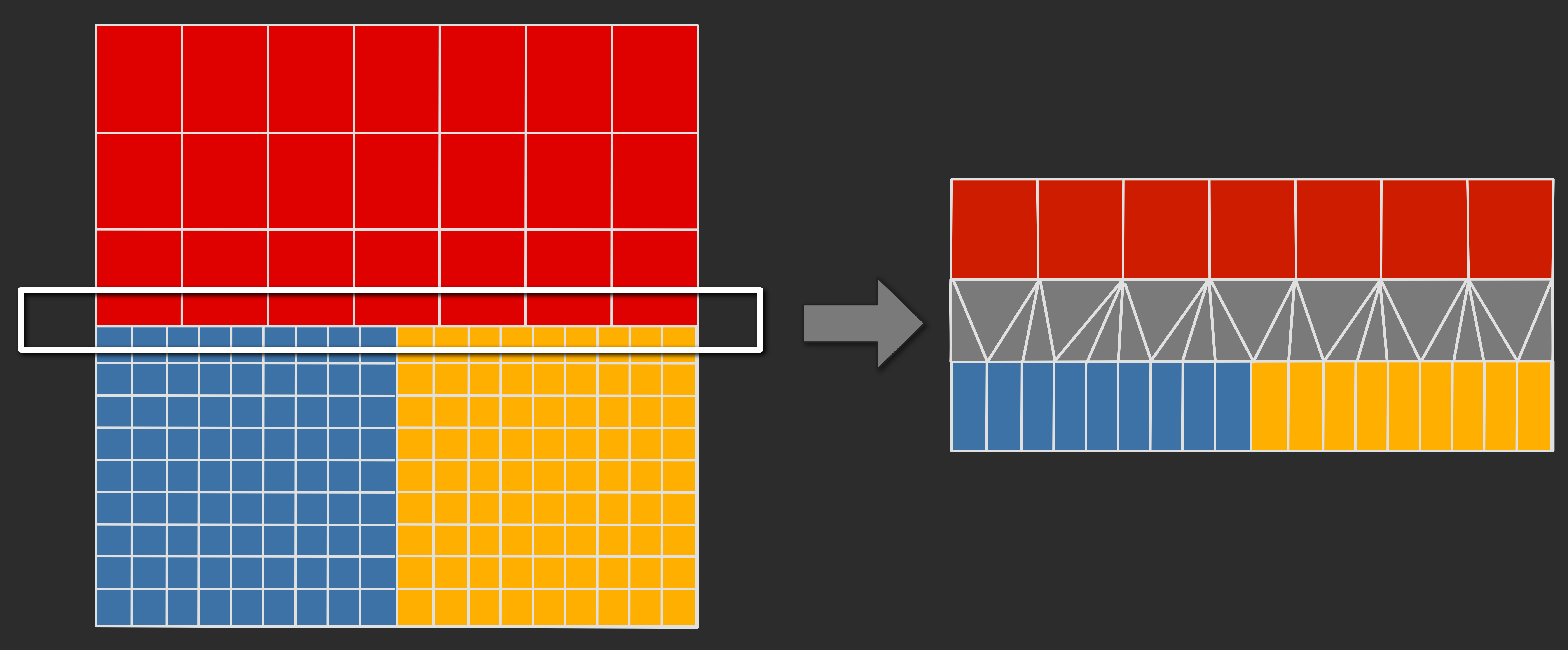
Cracks!



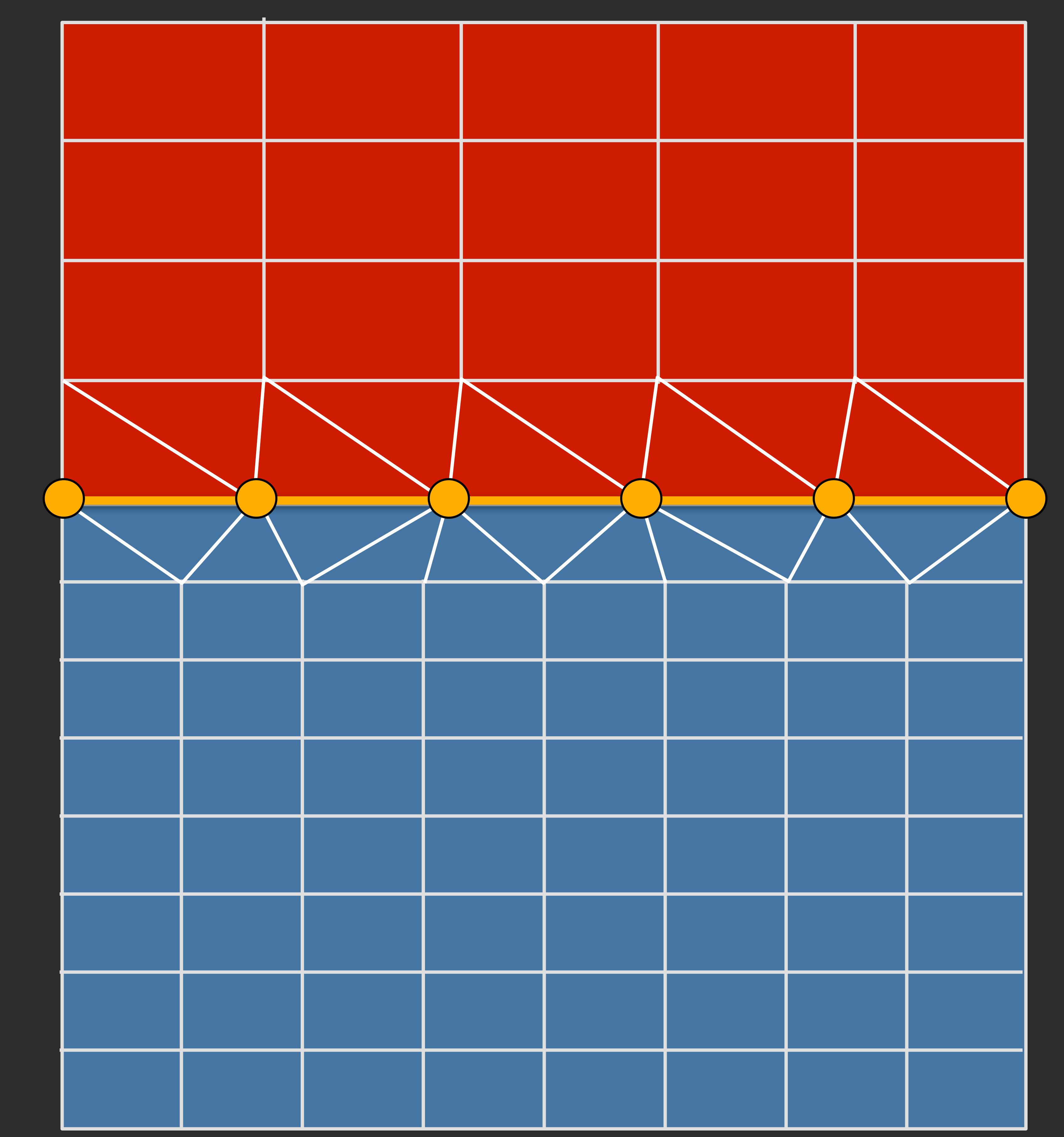
Off-line status quo: “stitching” fixes cracks

Use a strip of polygons to connect adjacent sub-patches

Creates dependency: cannot process sub-patches in parallel



Parallel crack fixing

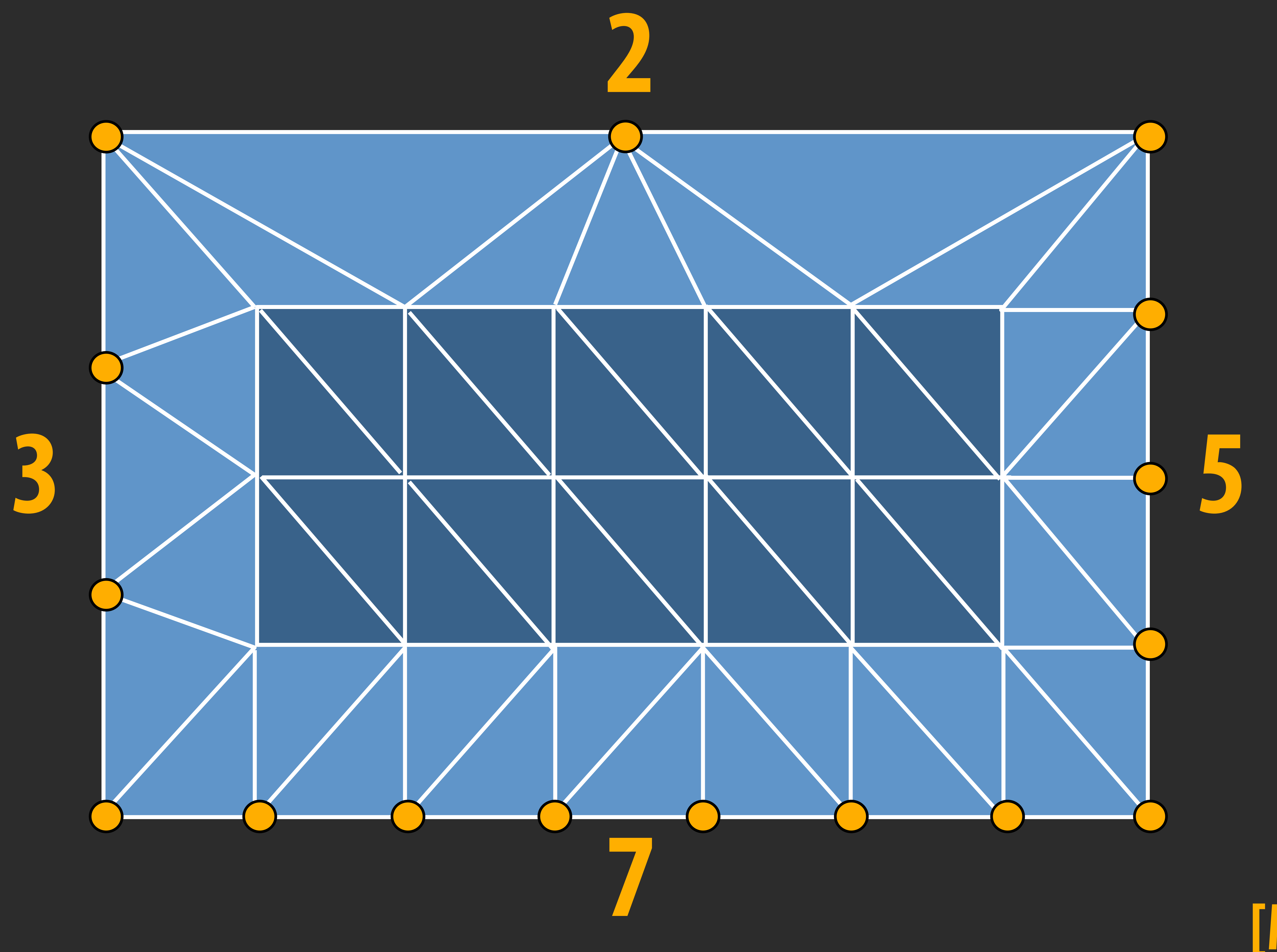


Adjacent regions agree on tessellation along edge (in this case: 5 segments)

Crack-free, uniform tessellation

Input: edge tessellation constraints for a patch

Output: (almost) uniform mesh that meets these constraints



[Moreton 01]

GPU tessellation

[Direct3D 11]

Crack-free, uniform patch tessellation
But no adaptive partitioning of patches!

Base patch data +
edge constraints

Uniform tessellation
(mesh generation)

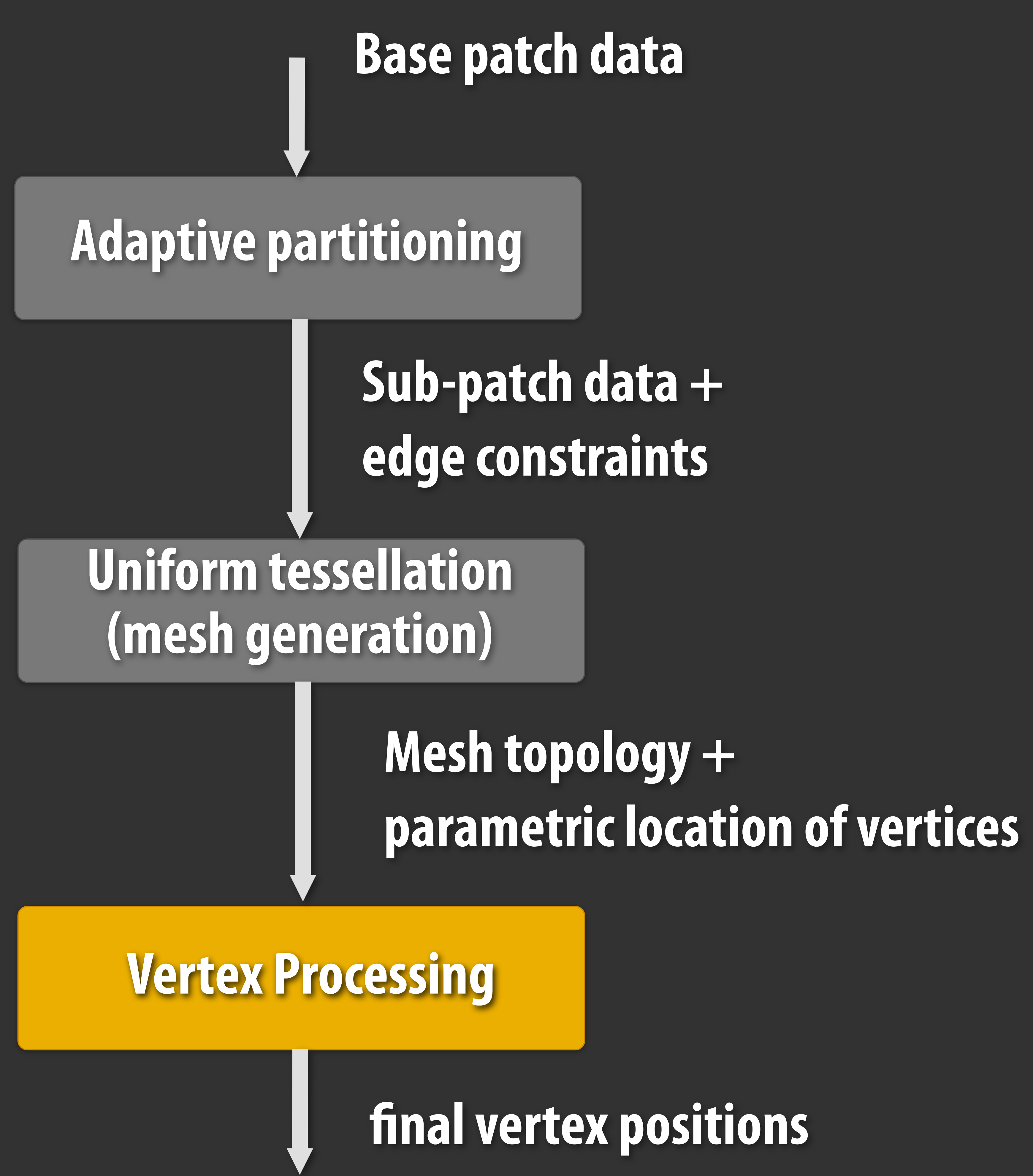
Mesh topology +
parametric location of vertices

Vertex Processing

final vertex positions

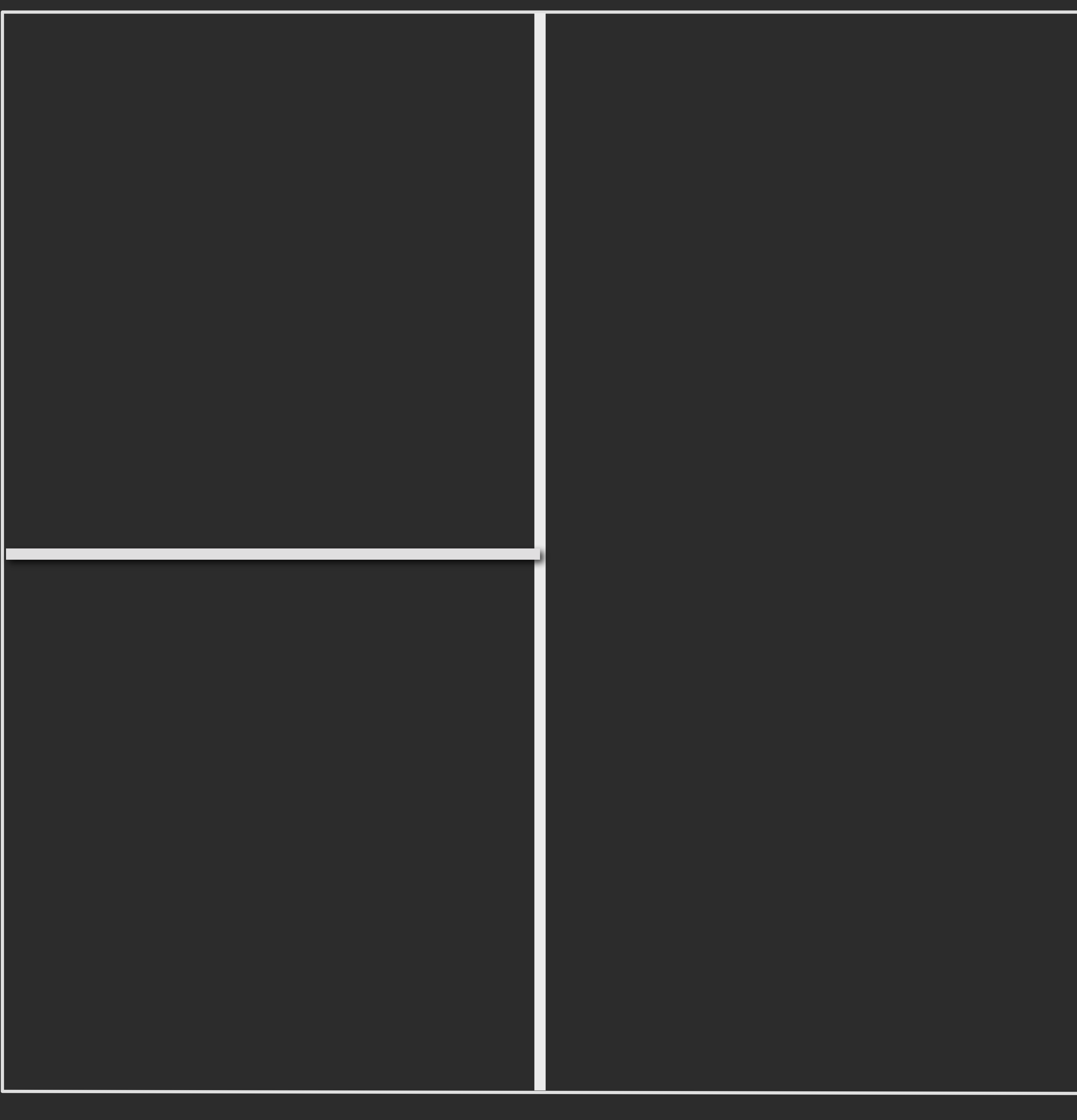
Fixed-function
Programmable

Want: adaptive tessellation pipeline

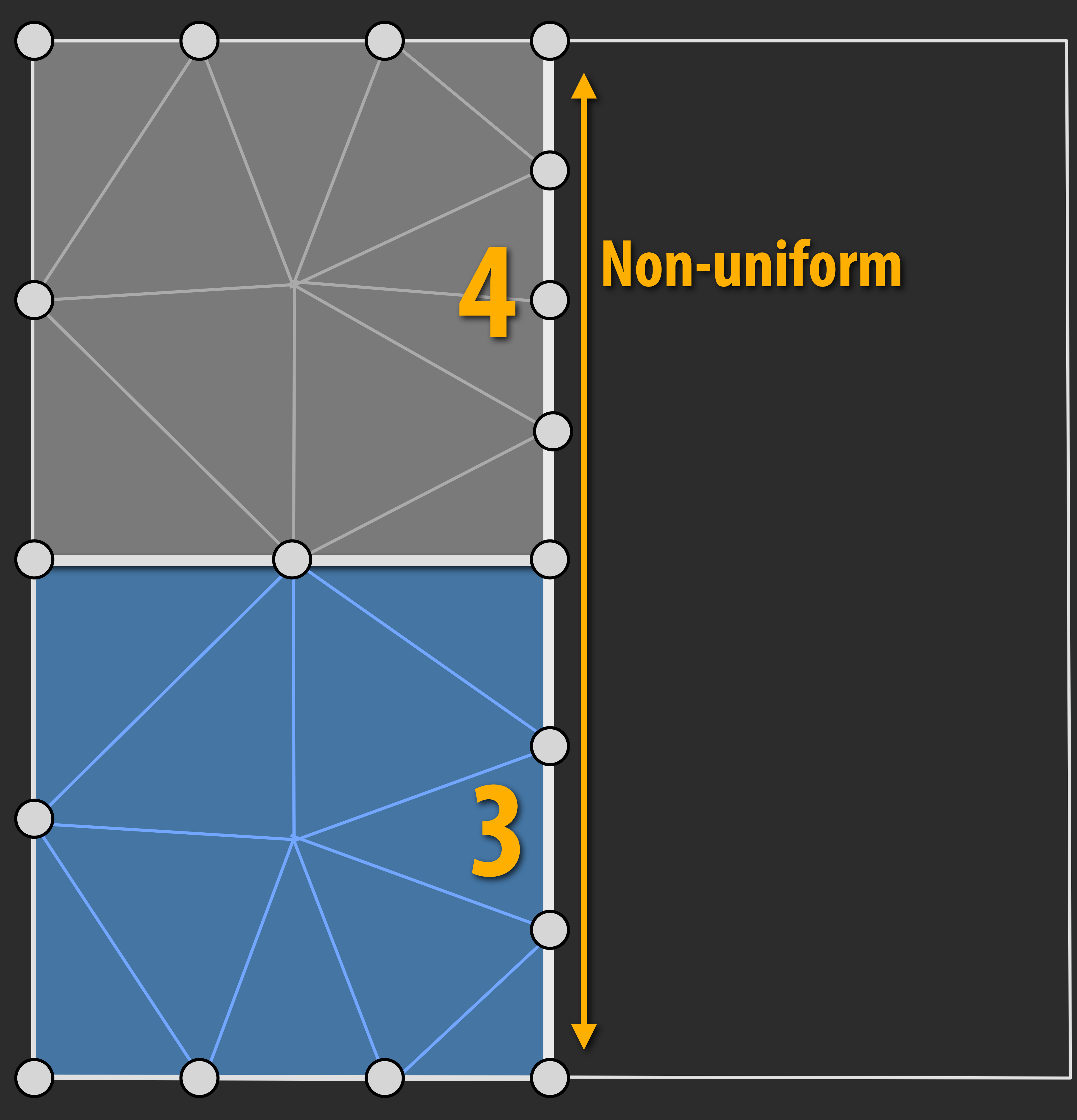


Fixed-function
Programmable

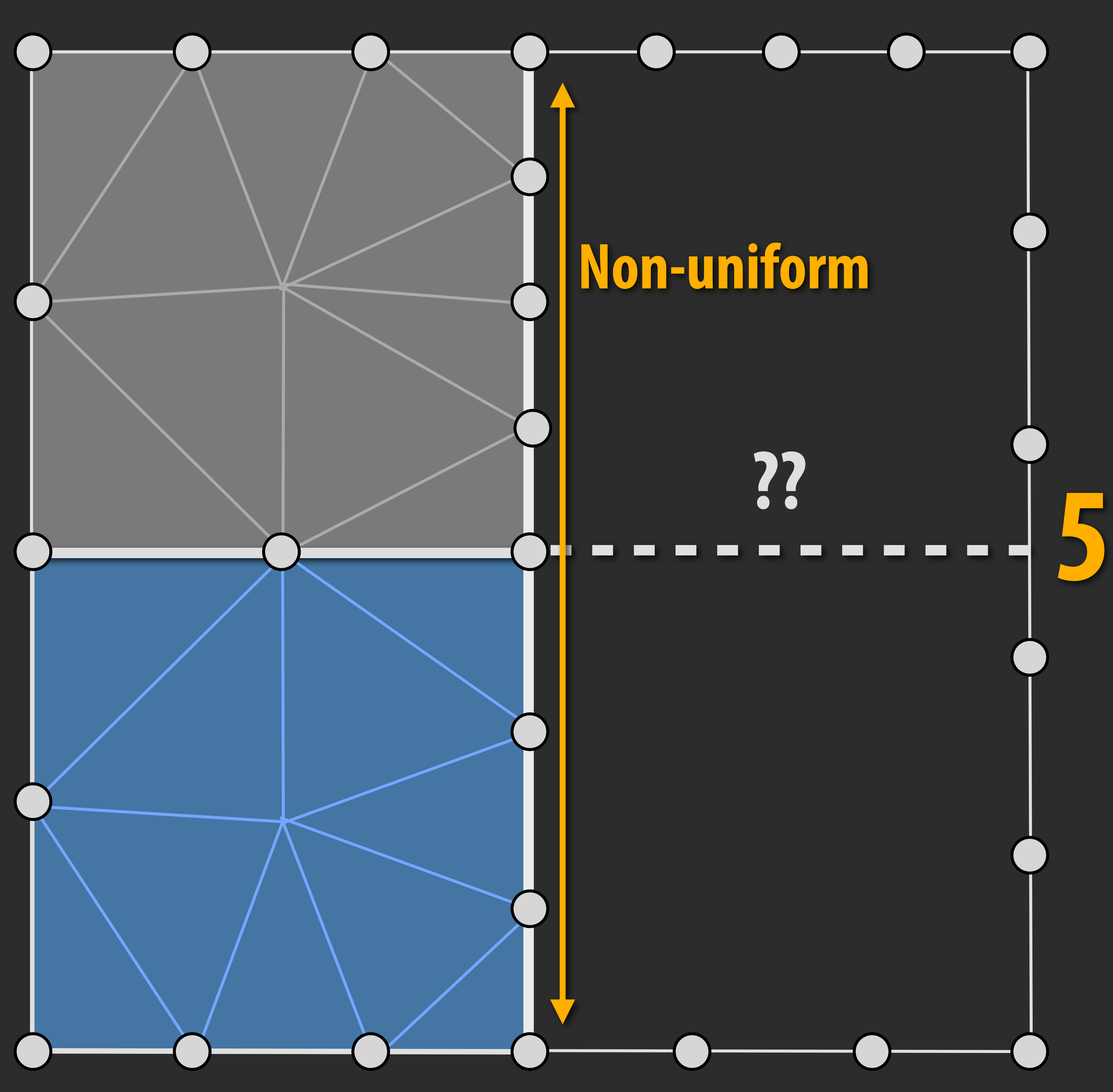
Making Lane-Carpenter match edges



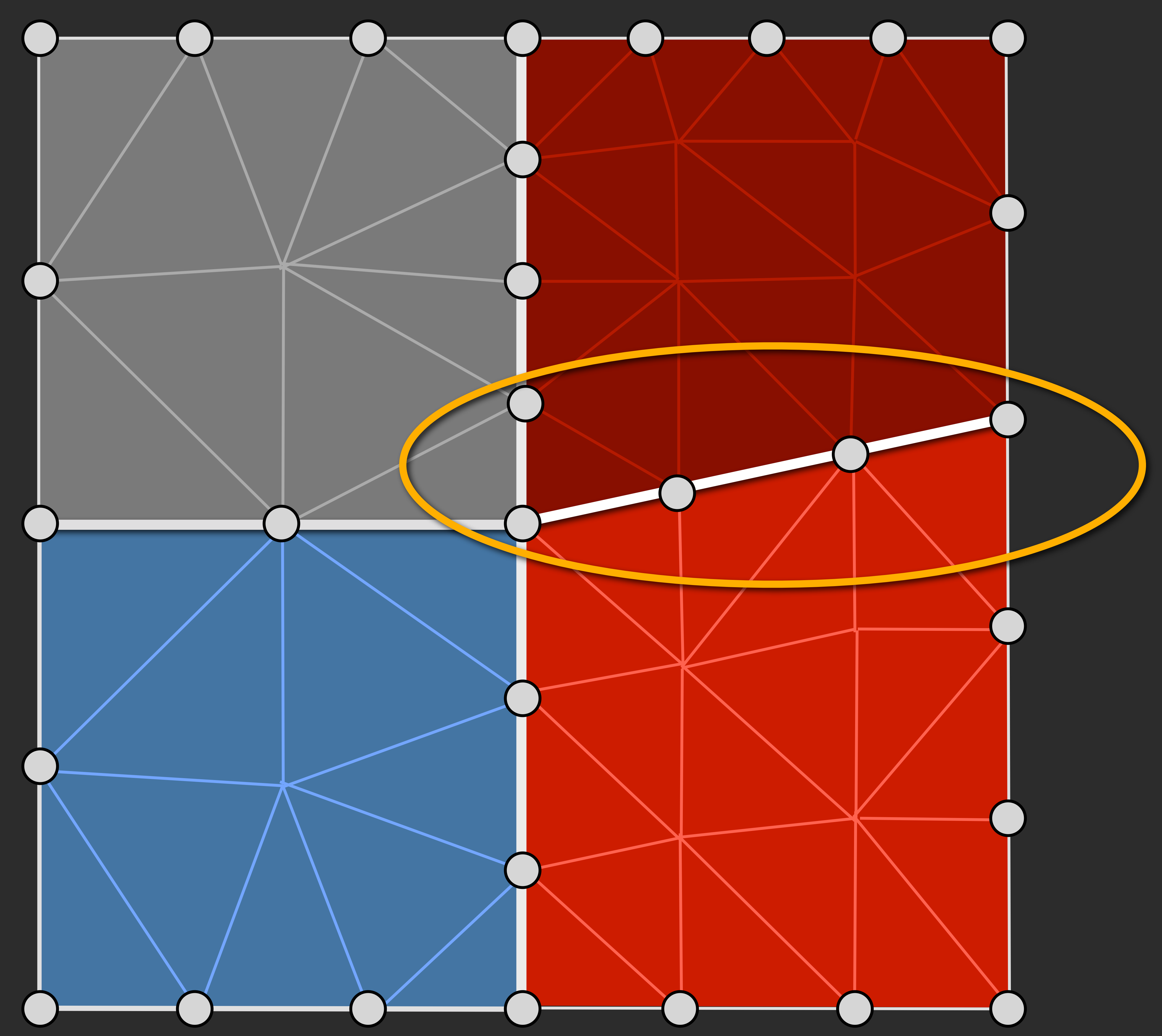
Making Lane-Carpenter match edges



Making Lane-Carpenter match edges



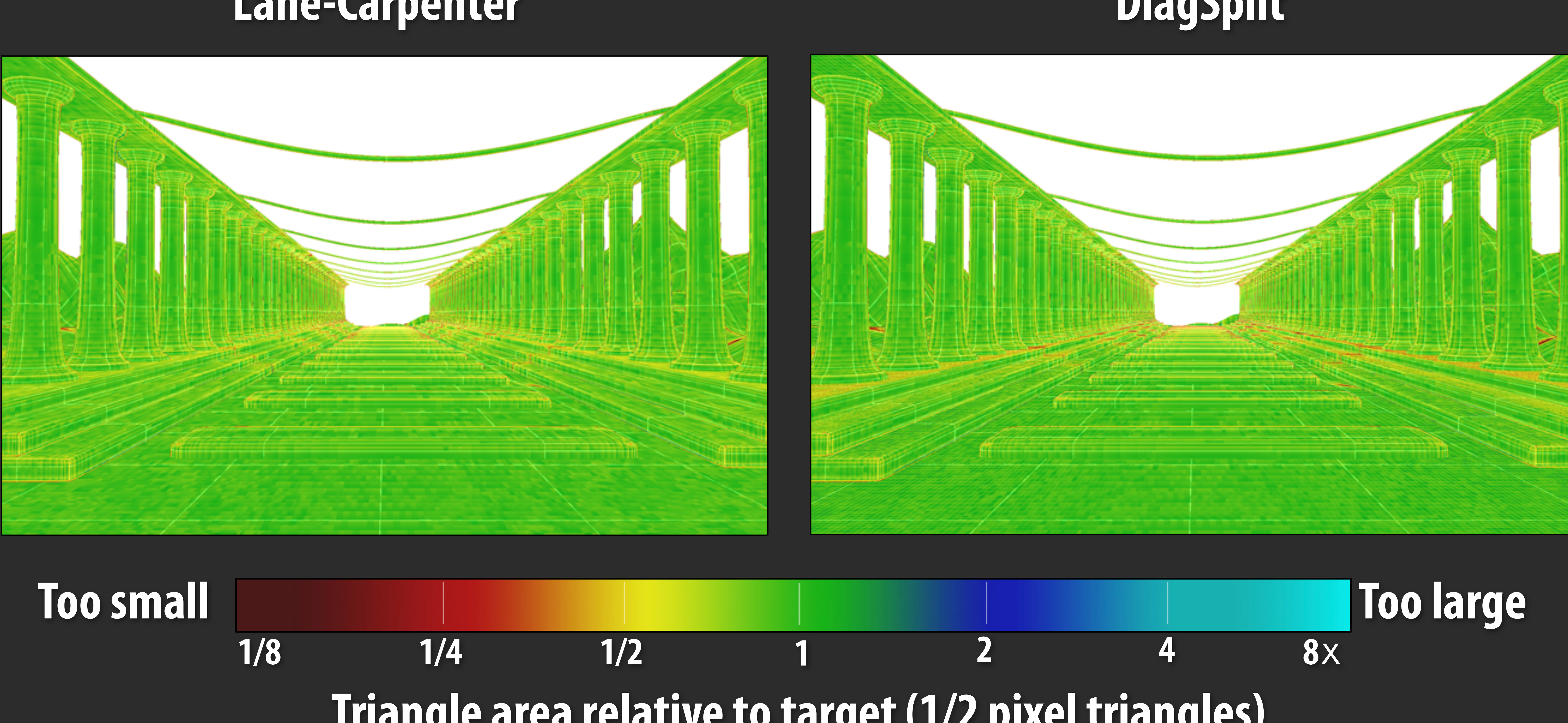
Non-isoparametric splits



DiagSplit: adaptive, crack-free, sub-patch parallel

DiagSplit adapts as well as Lane-Carpenter

7% more vertices



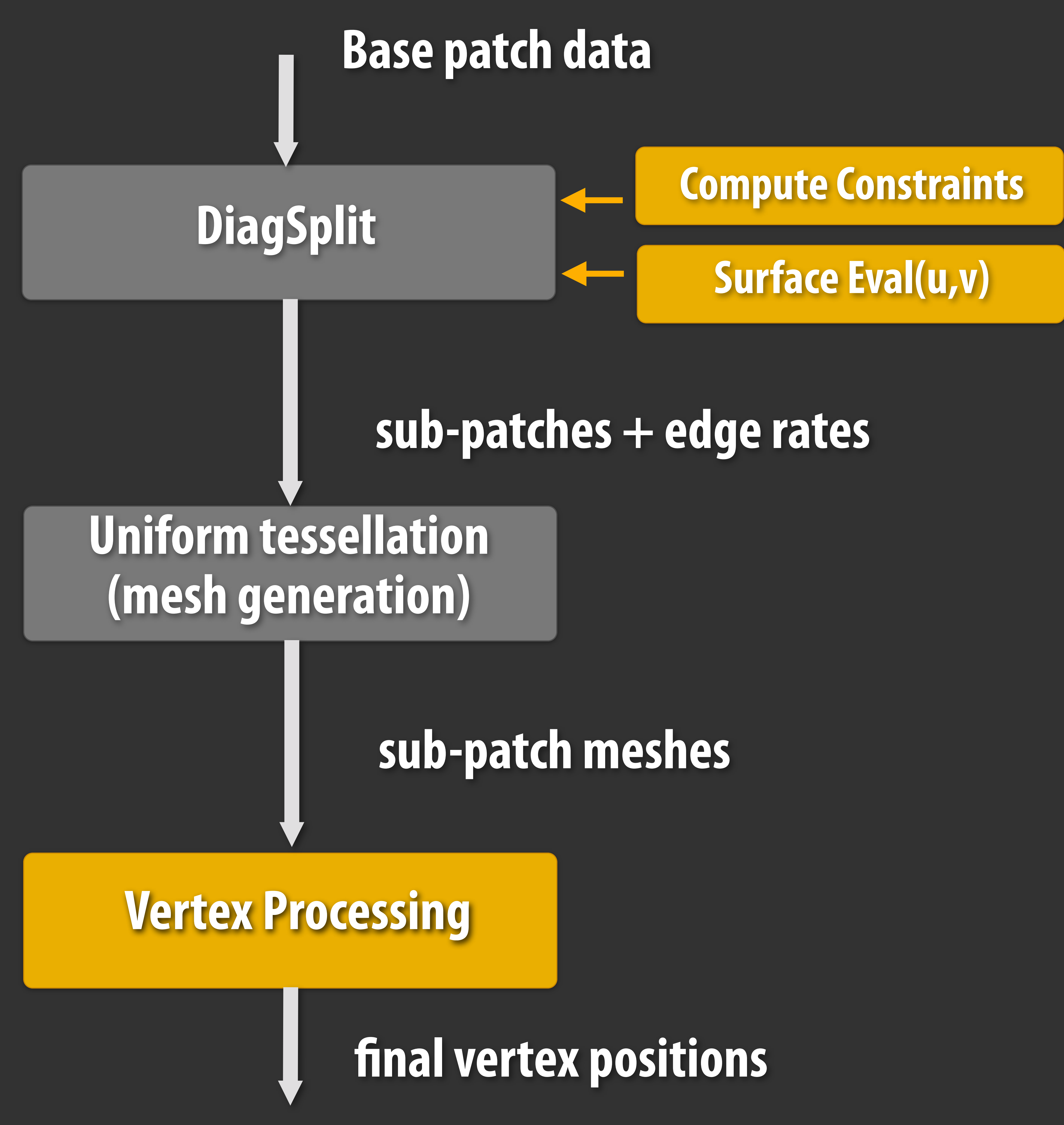
[Fisher 09]

DiagSplit tessellation pipeline

Divide and conquer
(not programmable, just provide edge function)

Irregular (data-amplification)
Fixed-function implementations exist

data-parallel, application programmable



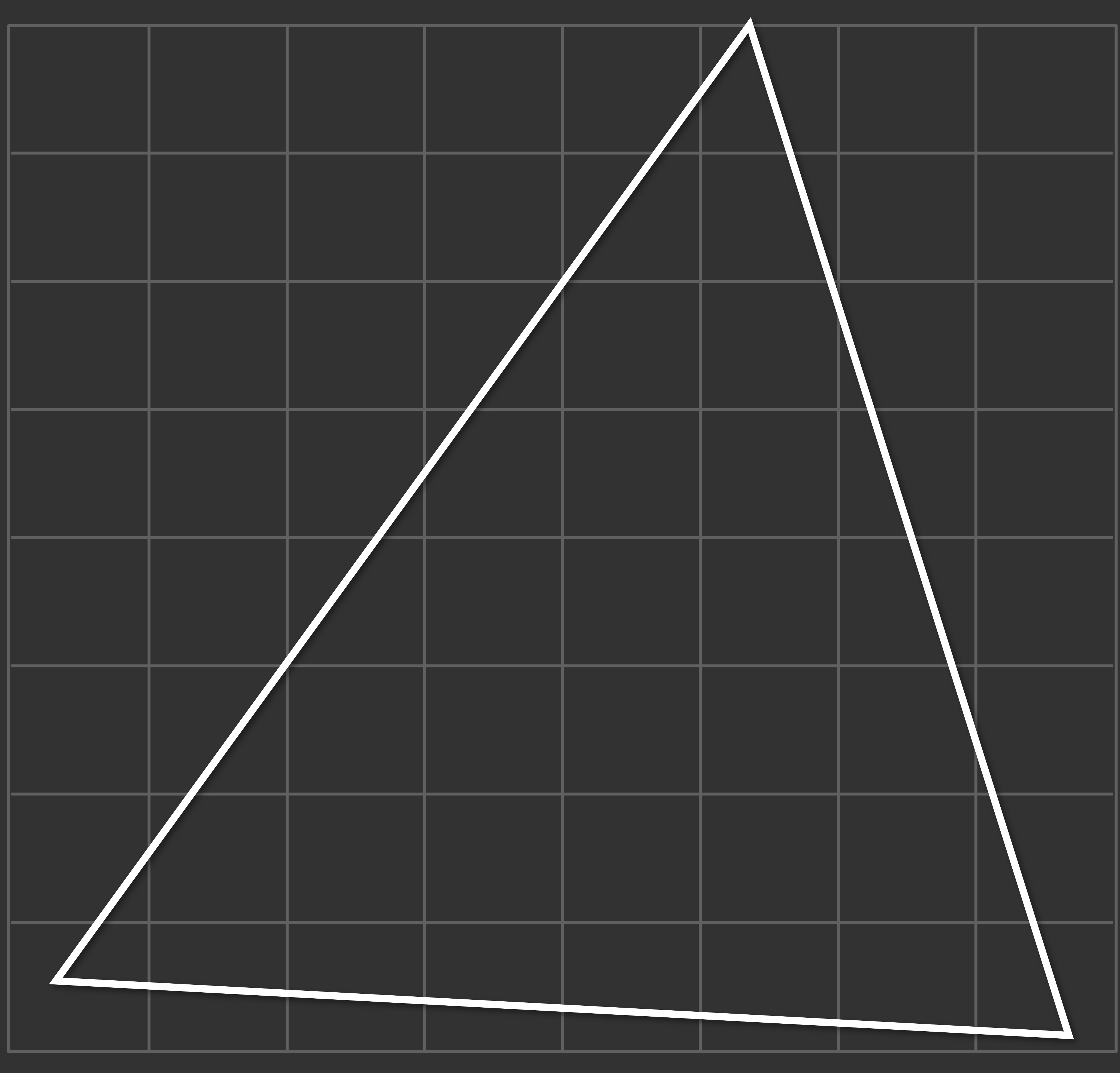
Fixed-function
Programmable

Recap

- DiagSplit: new algorithm designed to fit parallel system
  - Output triangles not equivalent to Lane-Carpenter (but very close)
- 1.4x - 8.2x reduction in vertex count compared to uniform [Fisher 09]
- Heterogeneous implementation
  - Programmable data-parallel component (supports all parametric surfaces)
  - Fixed-function components irregular, but parallelizable

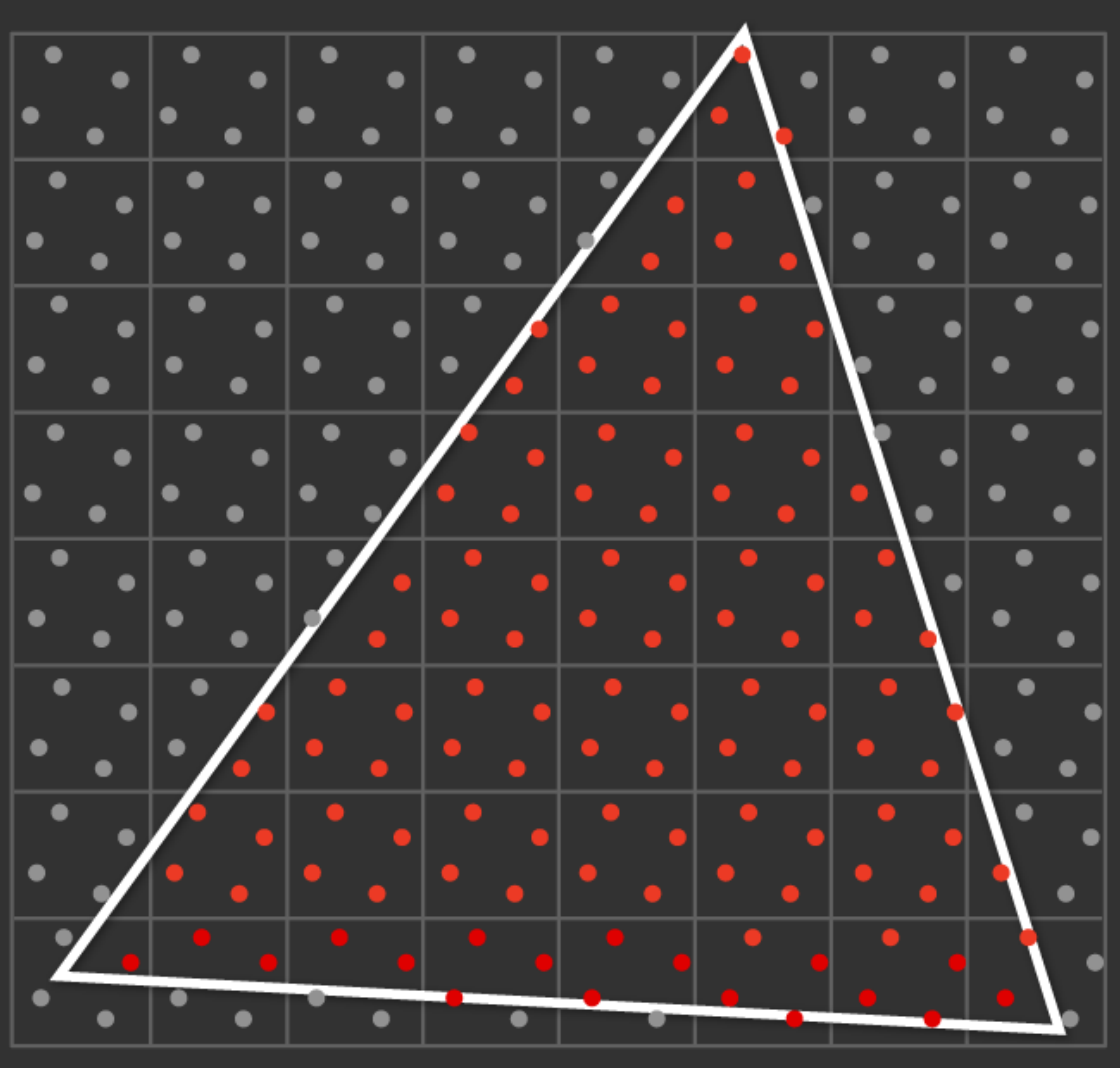
RASTERIZATION

Rasterization



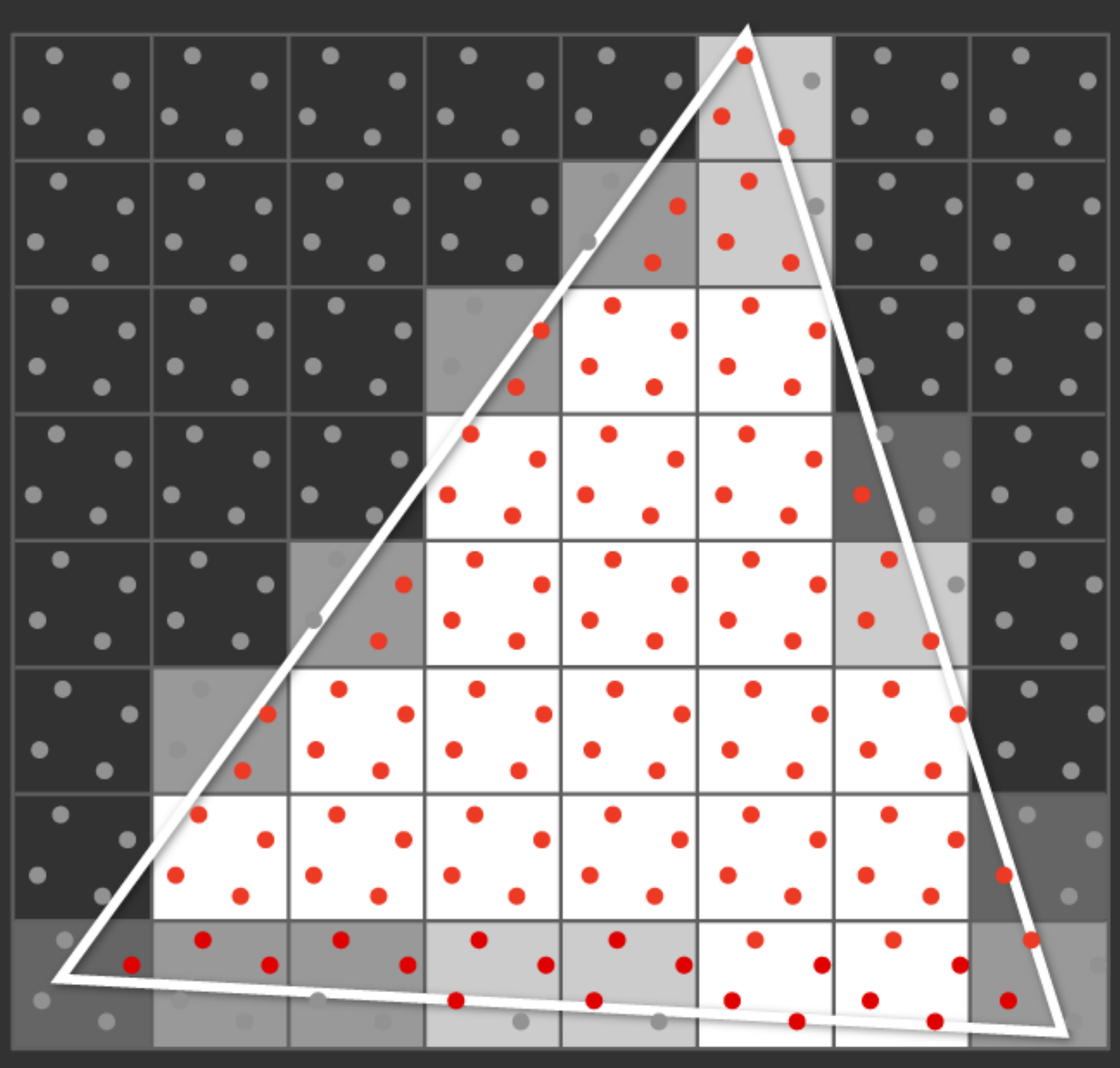
Rasterization

Compute coverage using point-in-triangle tests

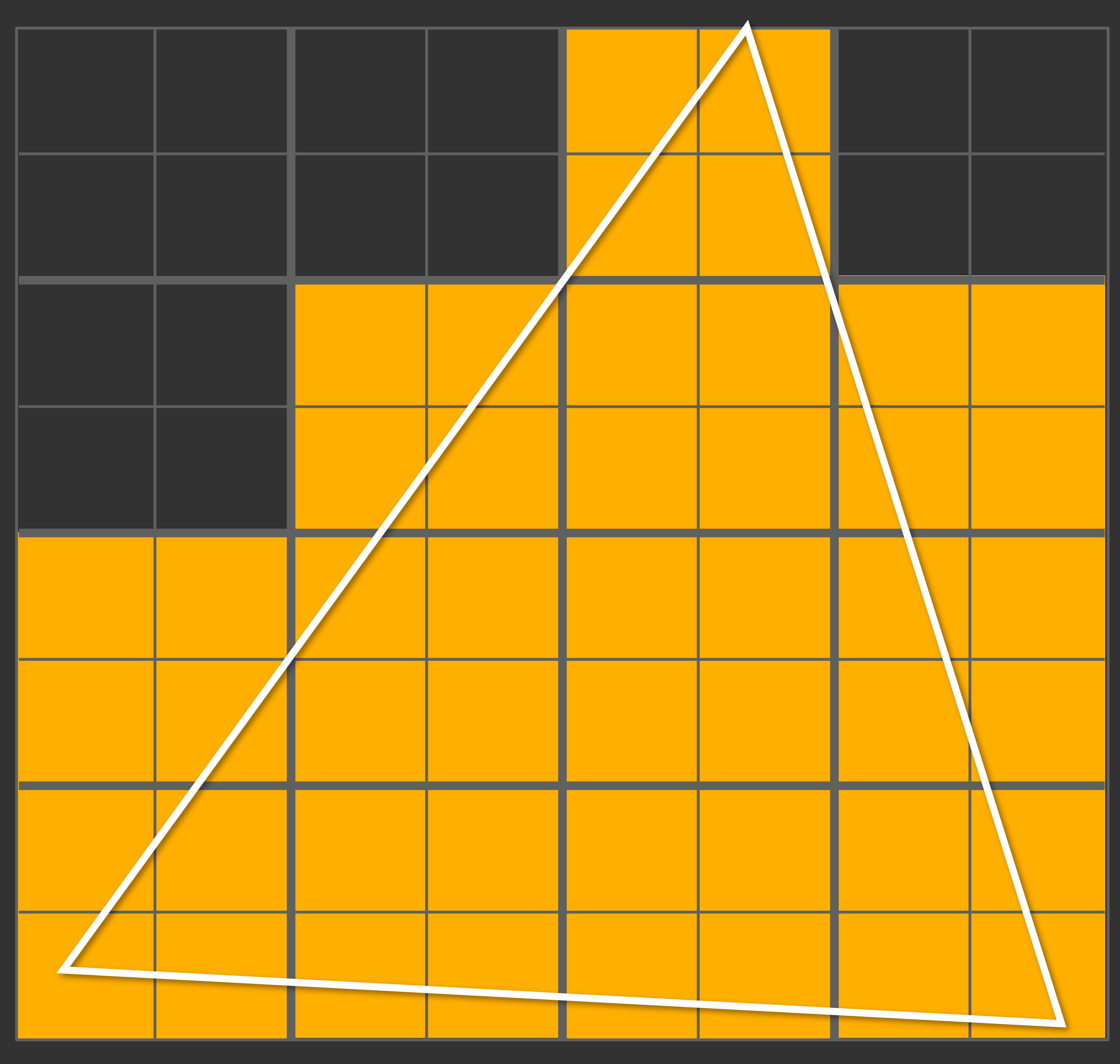


Rasterization

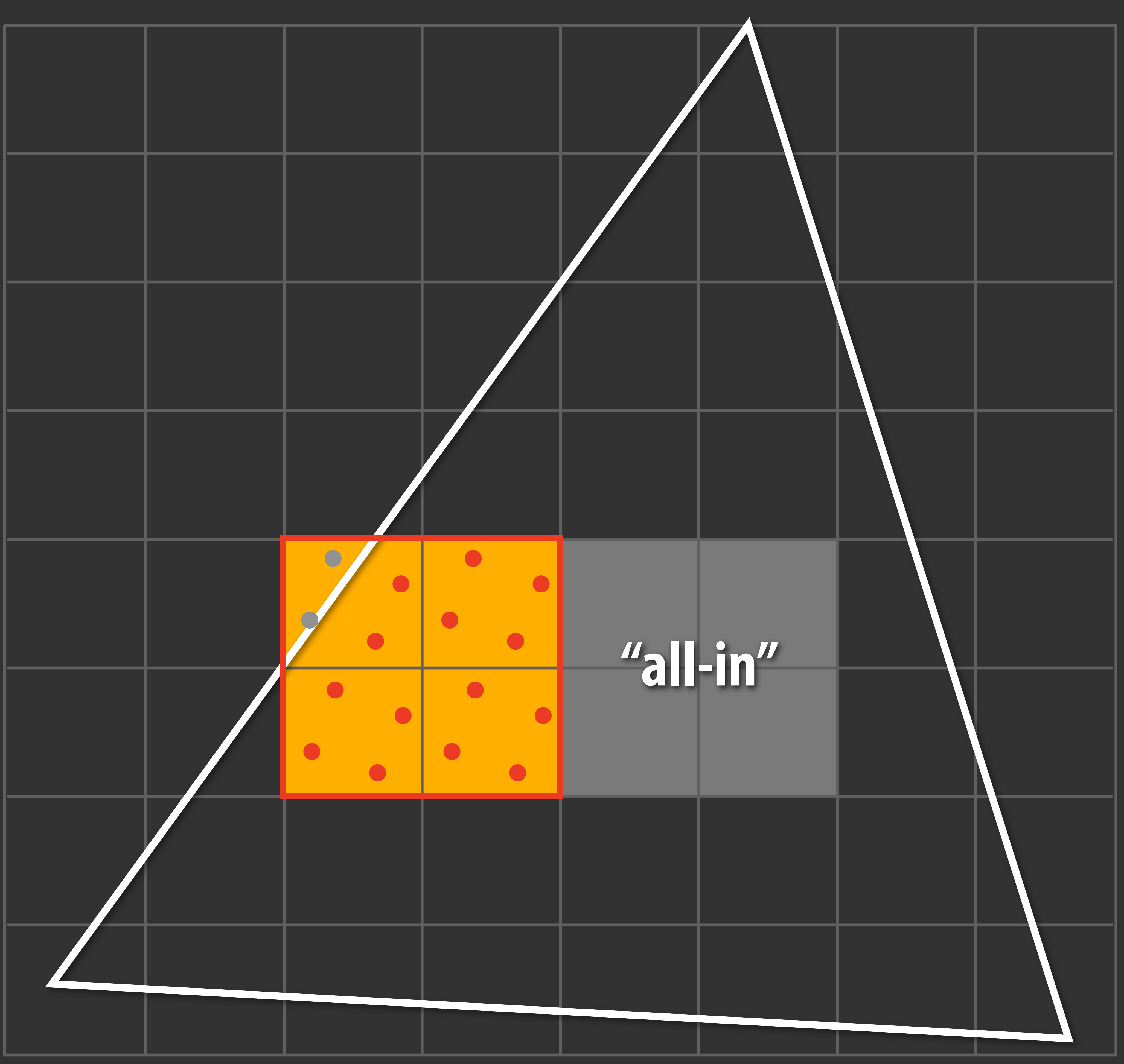
Compute coverage using point-in-triangle tests



Compute "possibly covered" pixels

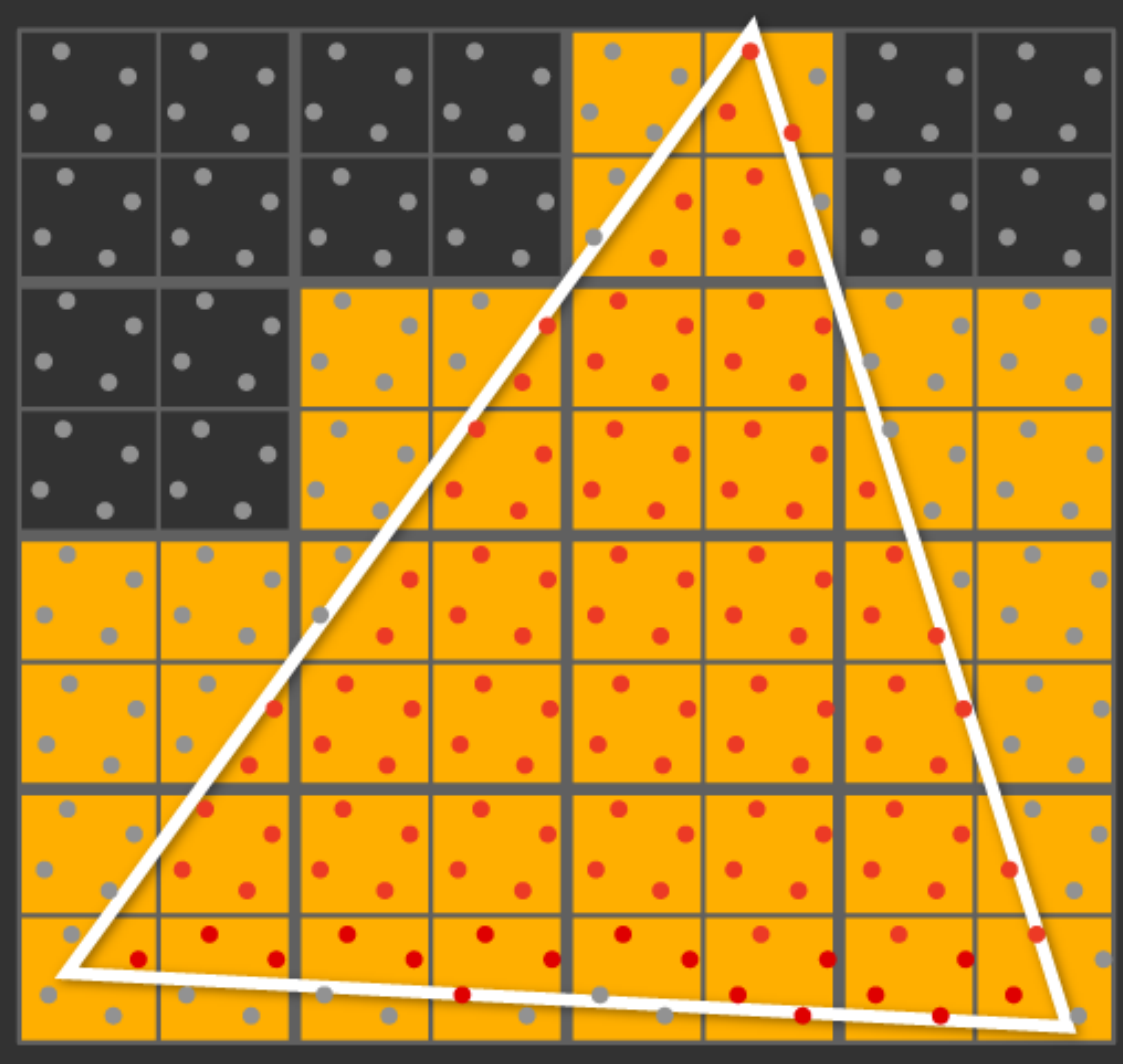


Data-parallel sample tests

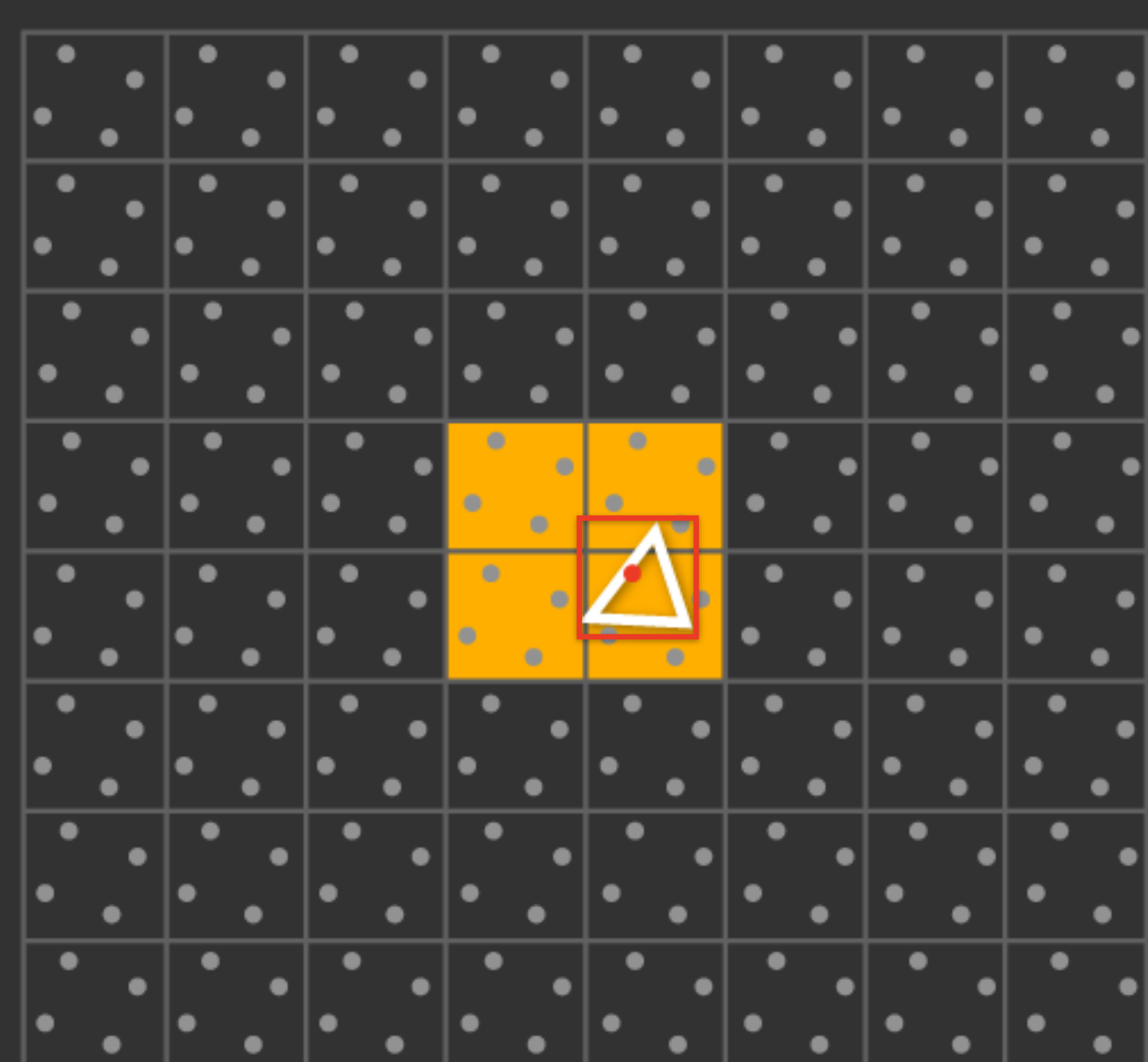


[Pineda 88]
[Fuchs 89]
[Greene 96]
[Seiler 08]

Micropolygons: most point-in-polygon tests fail



61% of candidate samples inside triangle



6% of candidate samples inside triangle

Low sample test efficiency!

Micropolygon rasterization

```
For each MP
```

Setup

```
Cull polygon if back-facing
```

Bound

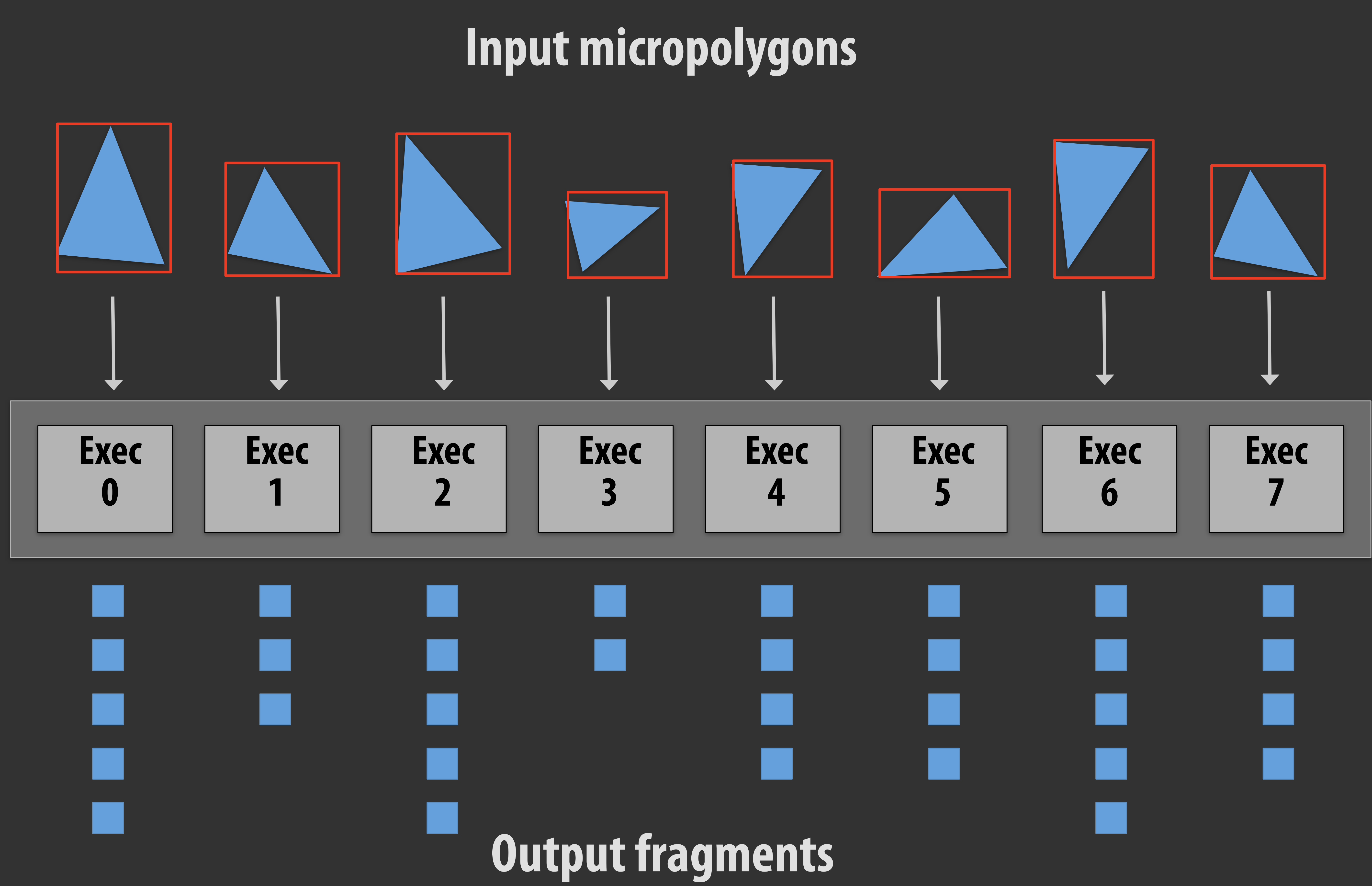
```
Compute subpixel bbox of MP
```

Test

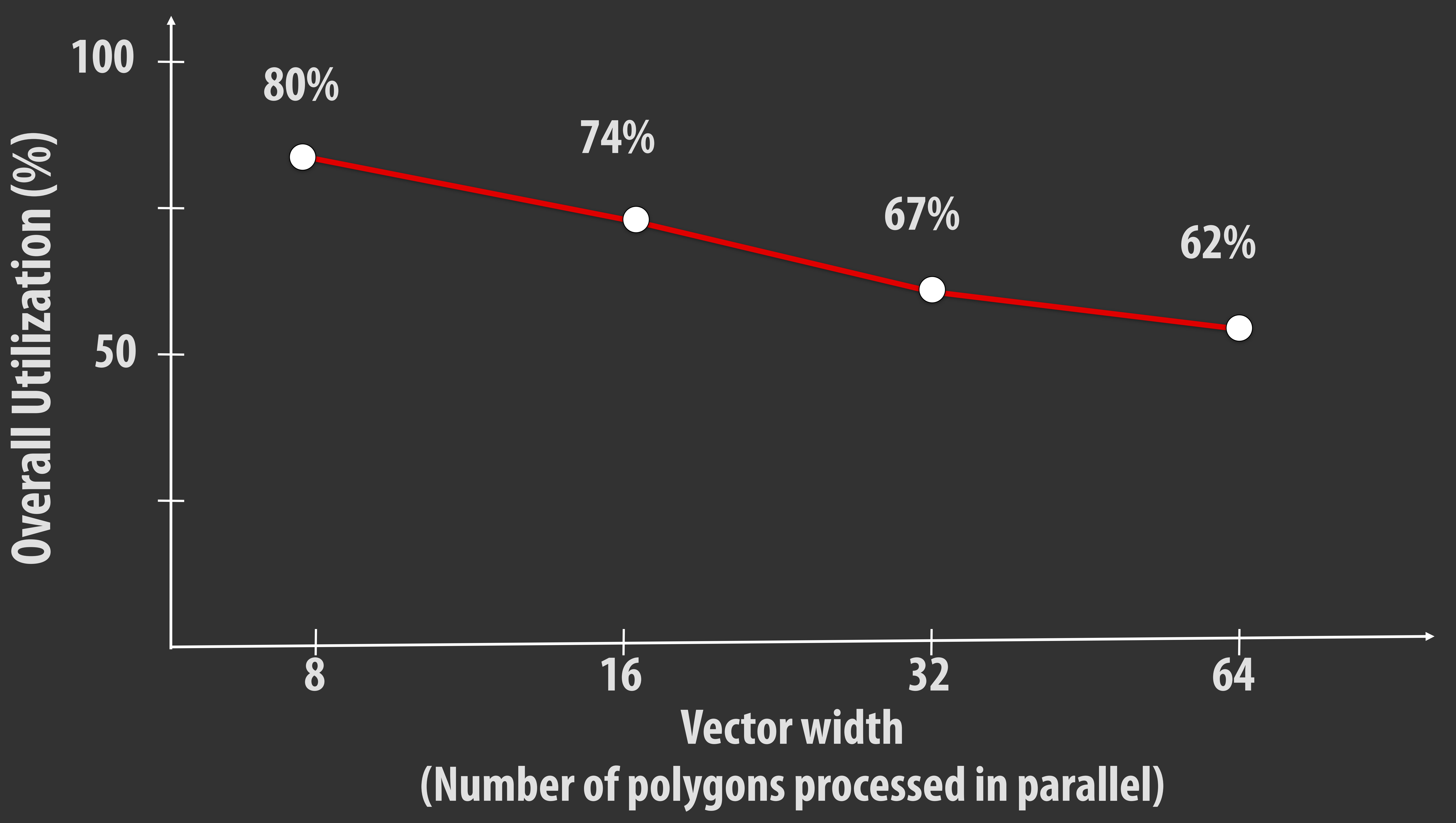
```
For each sample in bbox
   Test MP-sample coverage
```

Parallel micropolygon rasterization

Process multiple micropolygons simultaneously



MP parallel rast sustains high vector utilization



Micropolygon rasterization is simple, but expensive

- 28% of tested samples fall within the triangle
  - + Good: Up from 11% from a 16-sample-stamp algorithm
  - - Bad: Still much lower than stamp-based algorithms on large triangles
- No cheap “all-in” cases
- Can’t amortize setup across many sample tests

1 billion micropolygons/sec at 16 samples per pixel

(~15 million polygon scene at 60 Hz)

Estimated cost of GPU software implementation in CUDA:

Several high-end NVIDIA GPUs

See [Brunhaver et al. HPG 2010]: A Hardware Implementation of Micropolygon Rasterization...

See [Lane et al. HPG 2011]: High-performance Software Rasterization on GPUs

Lesson learned:

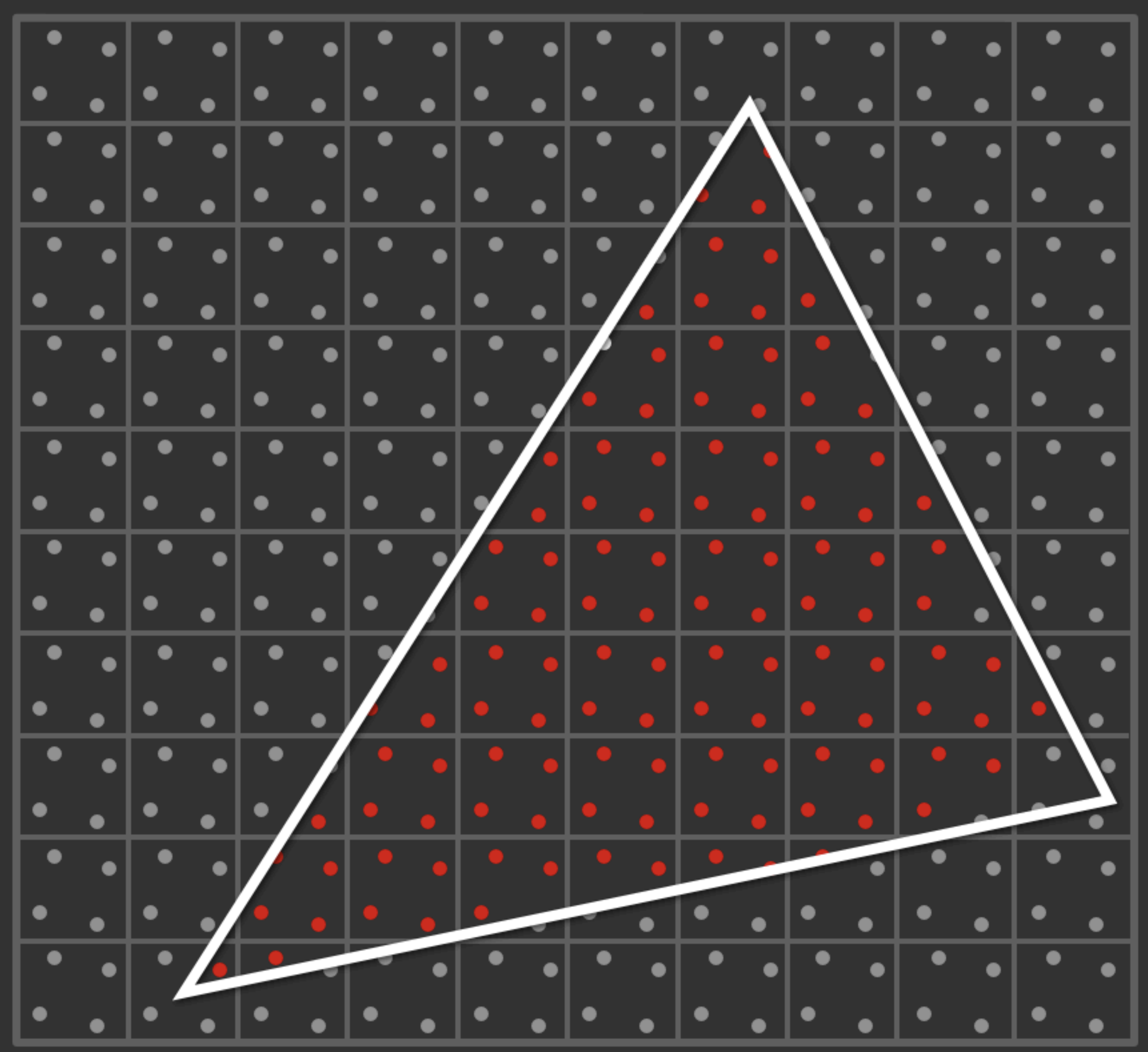
Despite the speed of the programmable parts of a GPU, I expect to see hardware rasterization around for awhile

SHADING:

Current GPUs shade small triangles inefficiently

Multi-sample locations

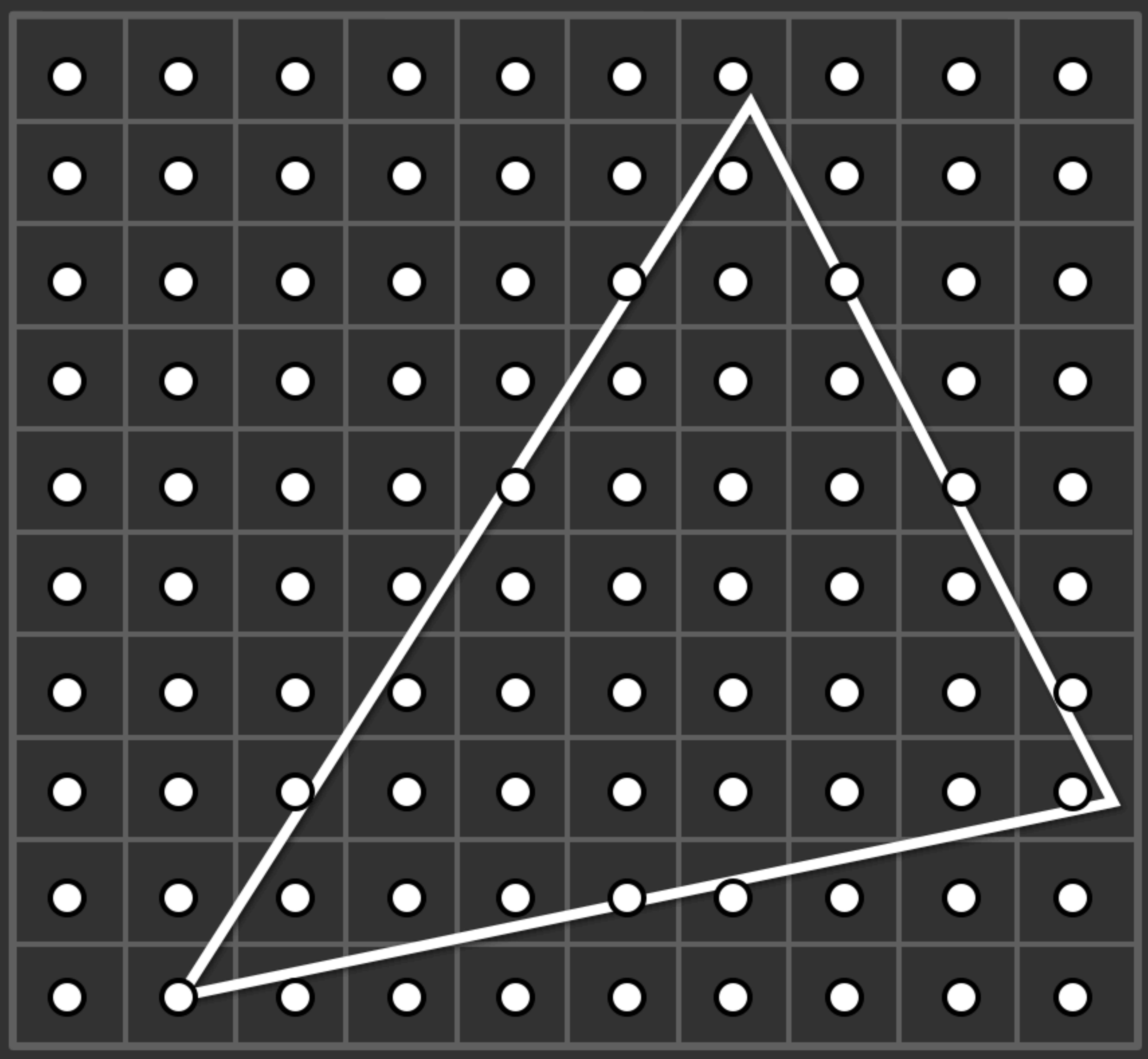
[Akeley 93]



Sample coverage multiple times per pixel (for anti-aliased edges)

Shading sample locations

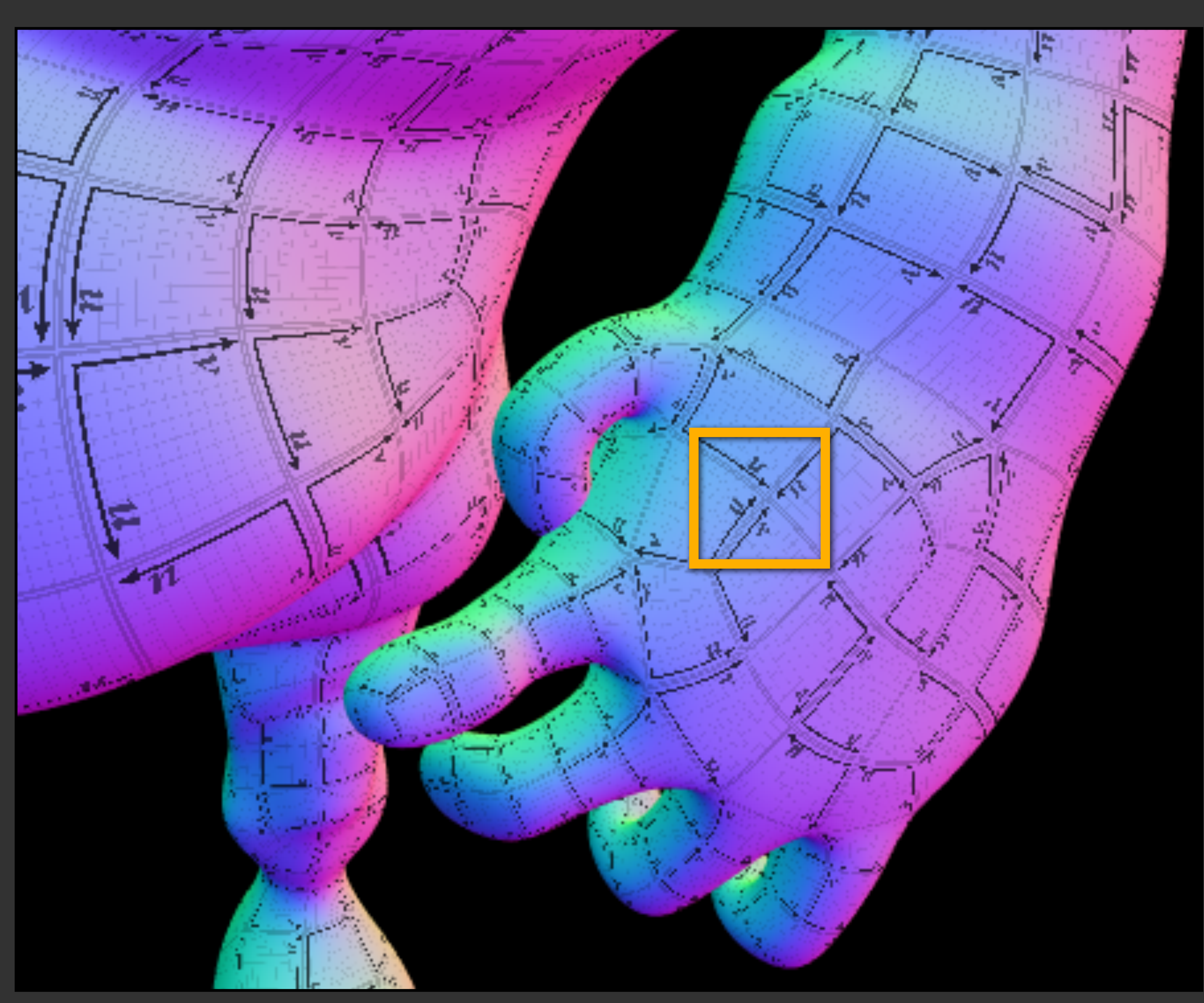
[Akeley 93]



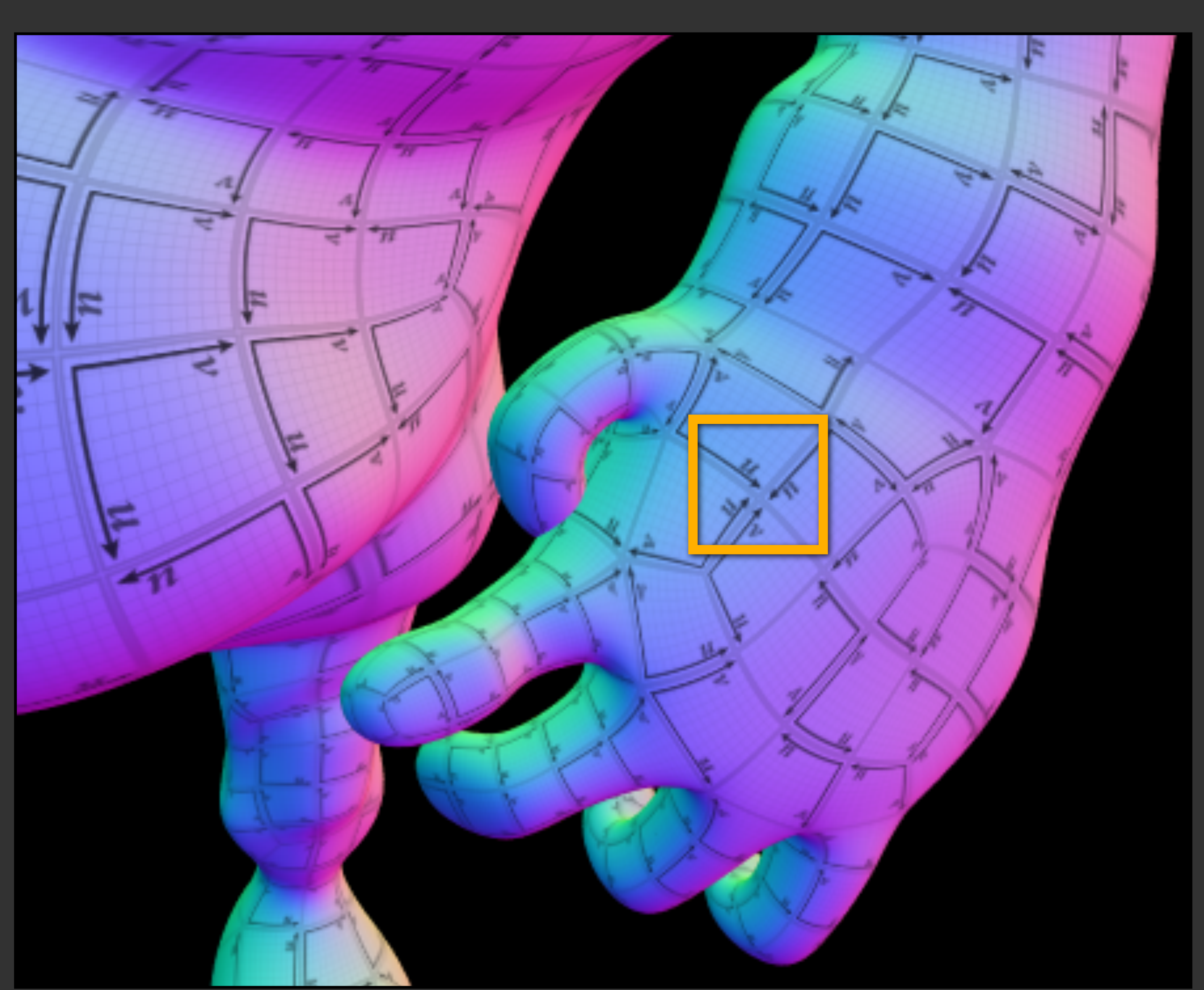
Sample shading once per pixel

Texture data is pre-filtered to avoid aliasing

(one shade per pixel is sufficient)

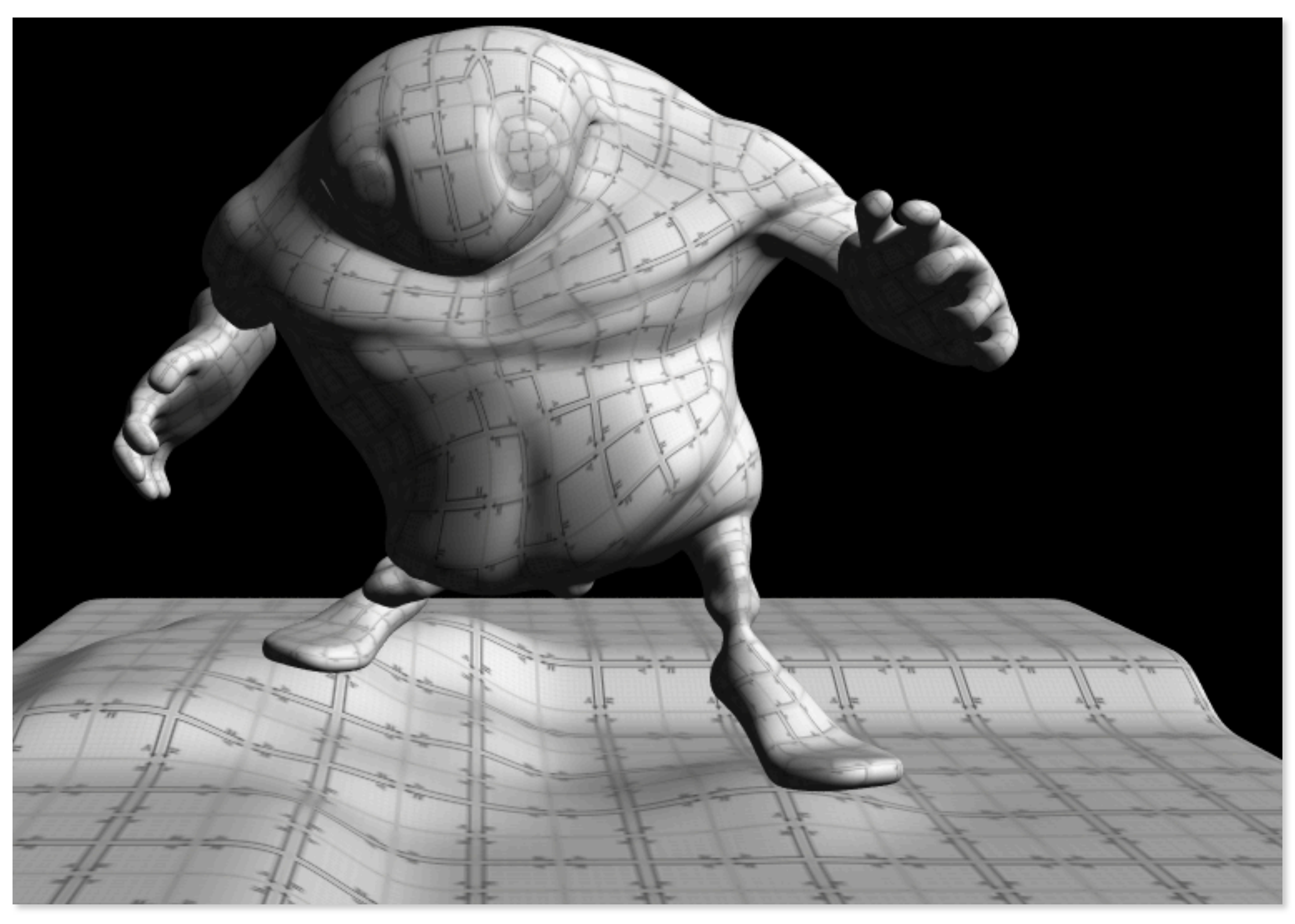


No pre-filtering
(aliased result)

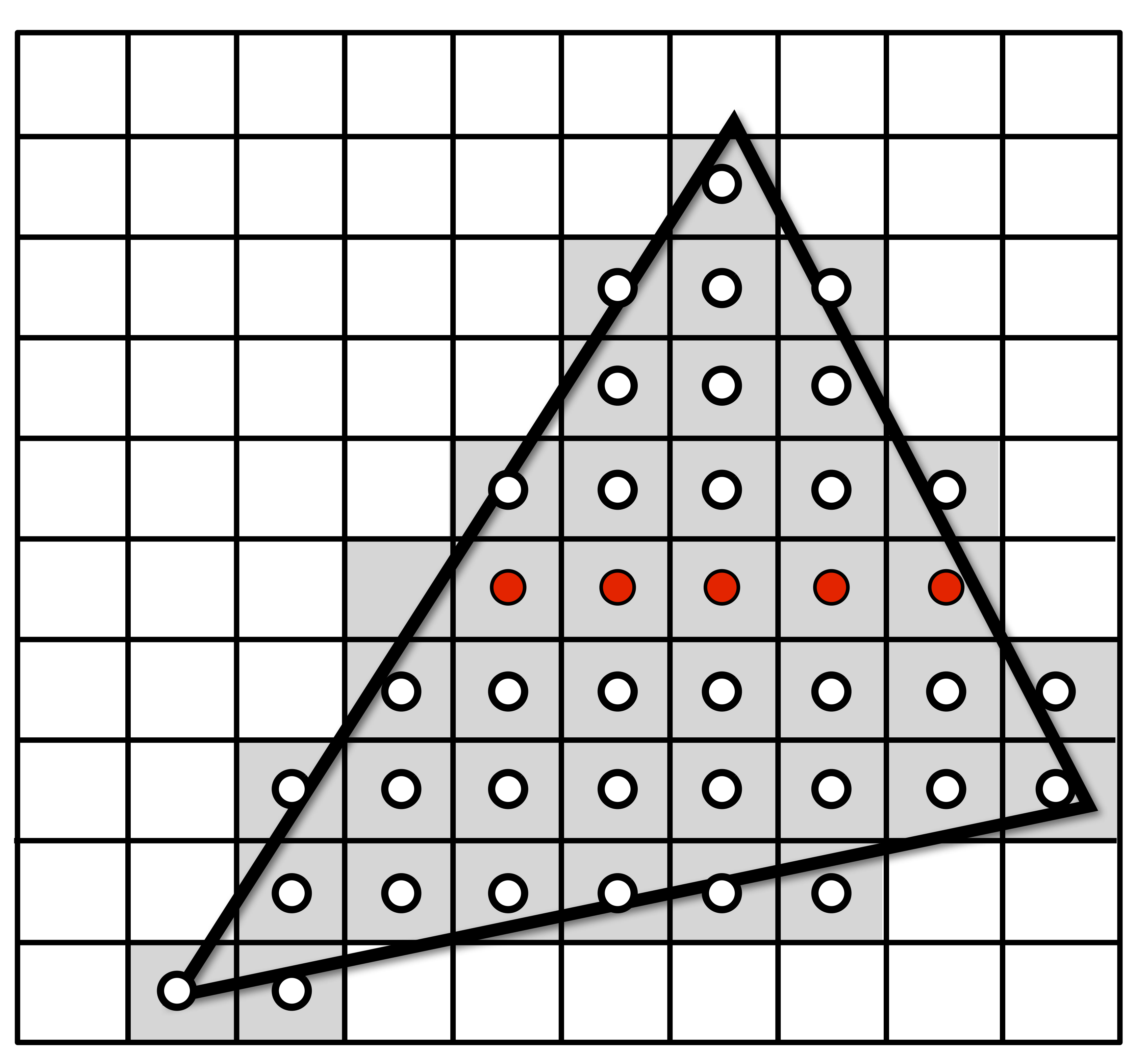


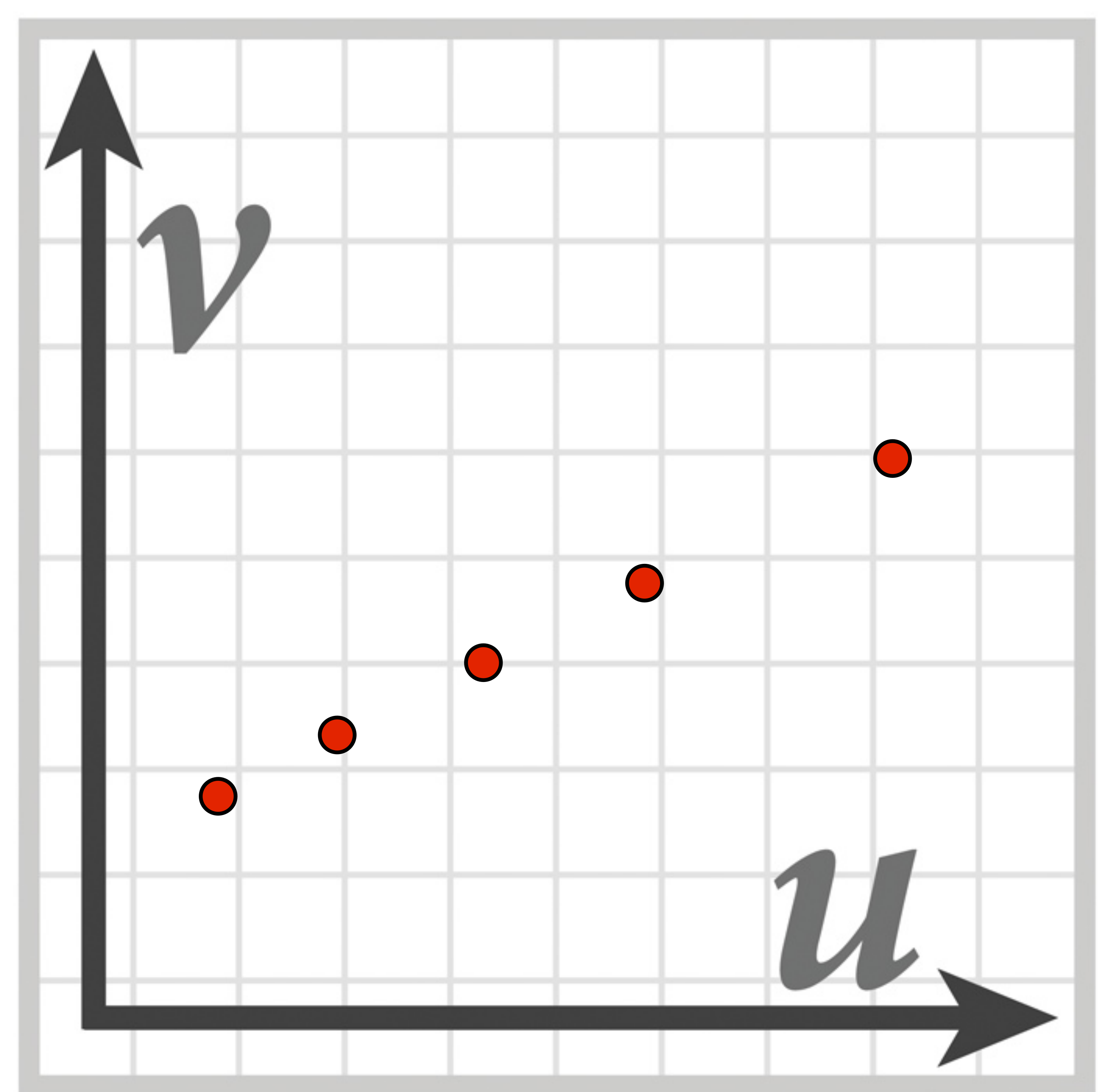
Pre-filtered texture

Recall this image?



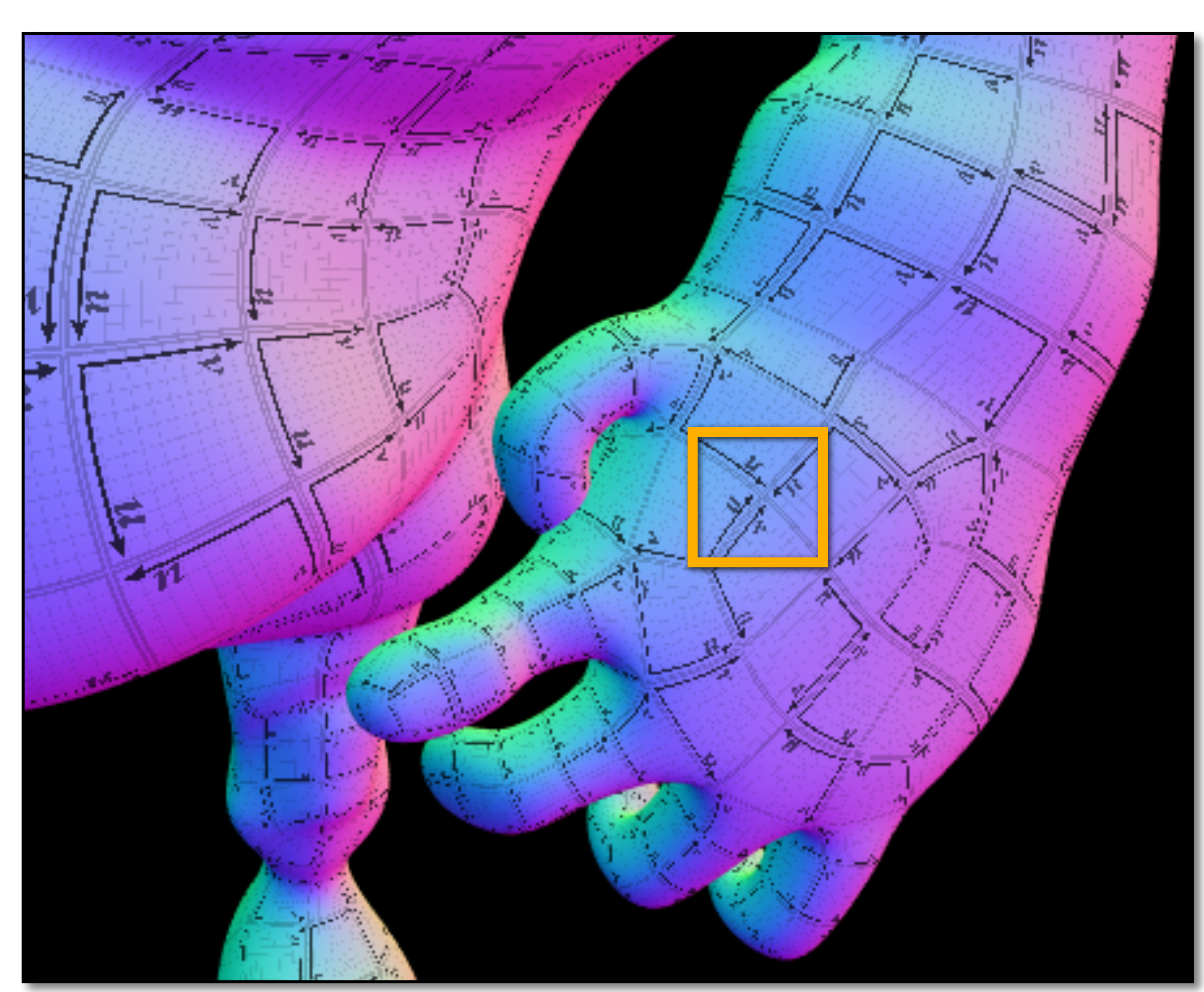
Texture space


Screen space



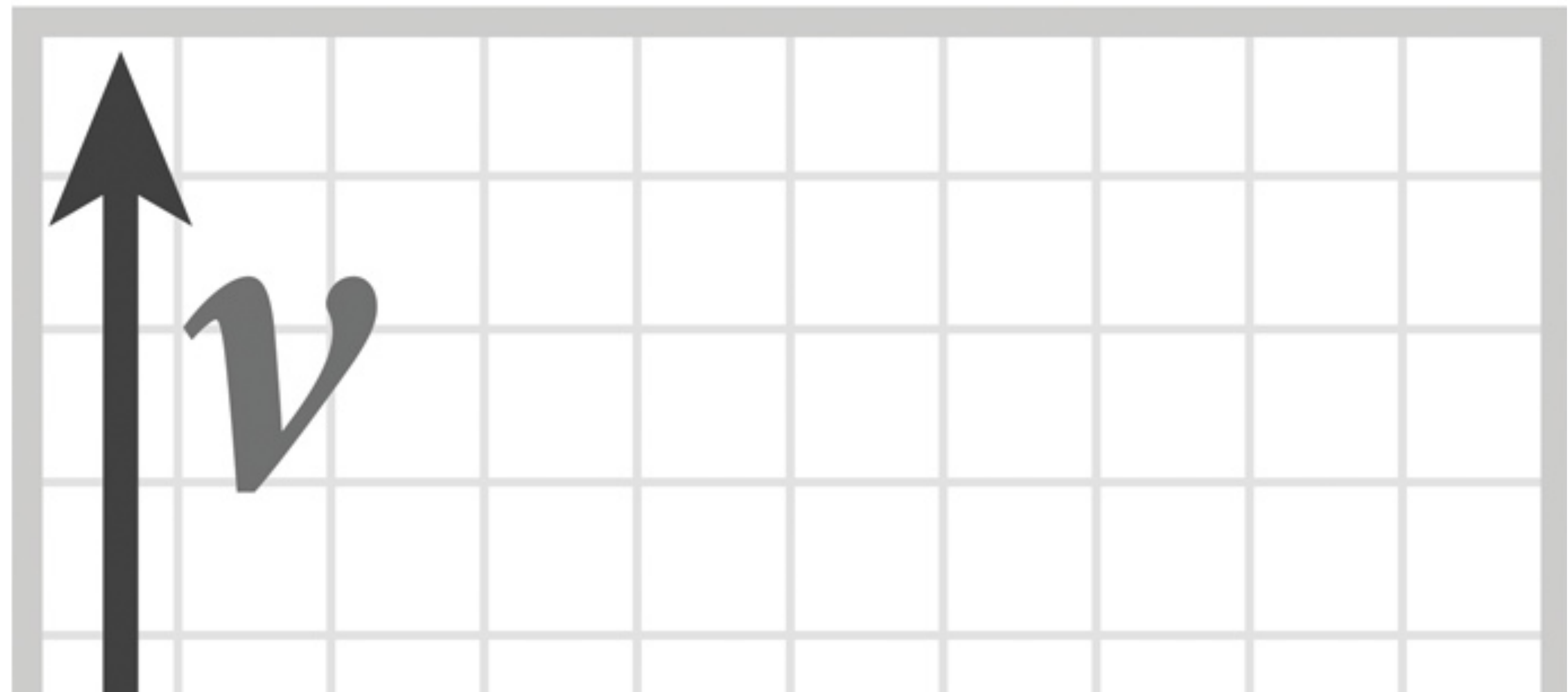
Texture space

Aliasing due to undersampling

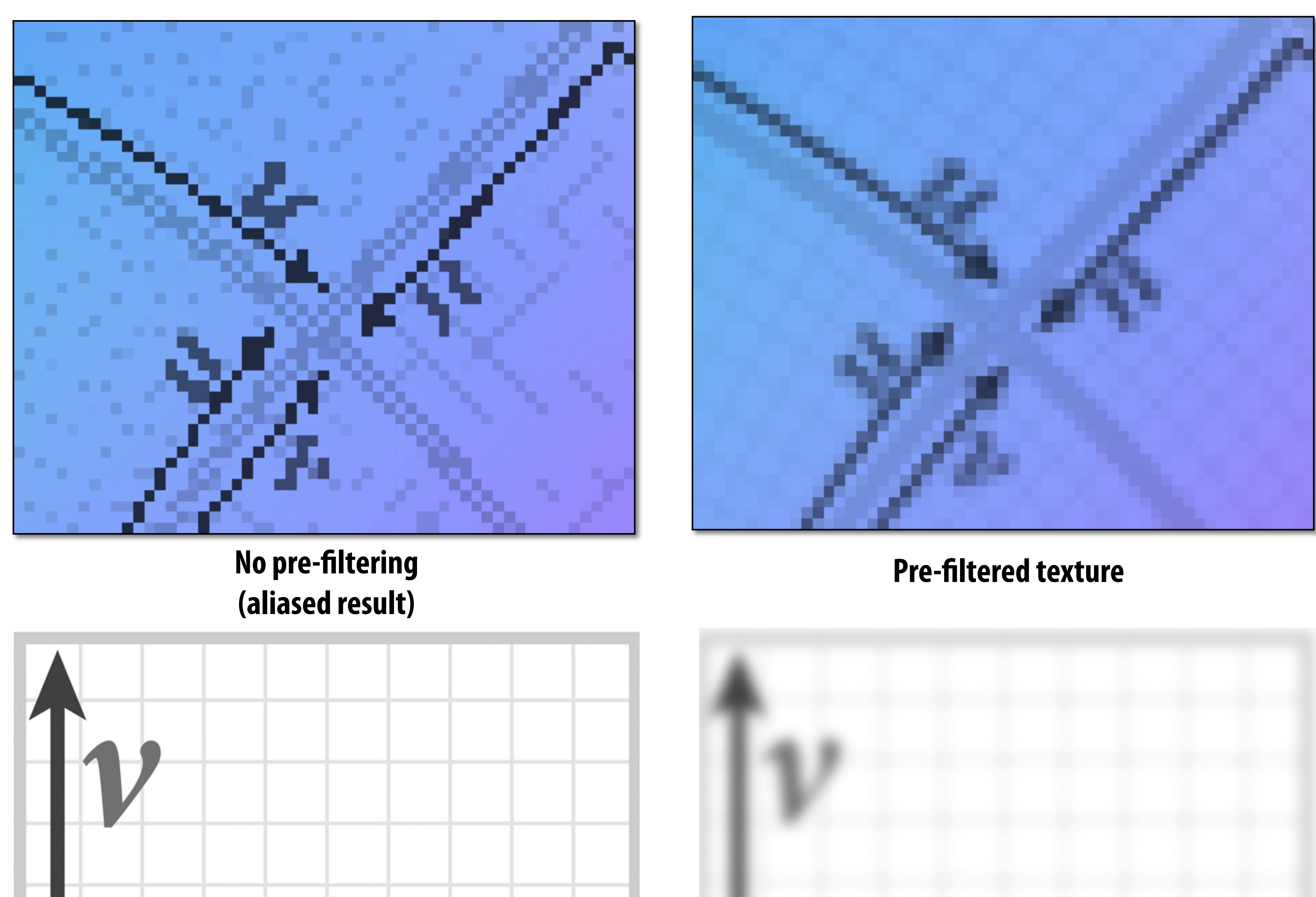


No pre-filtering
(aliased result)

Pre-filtered texture



Aliasing due to undersampling

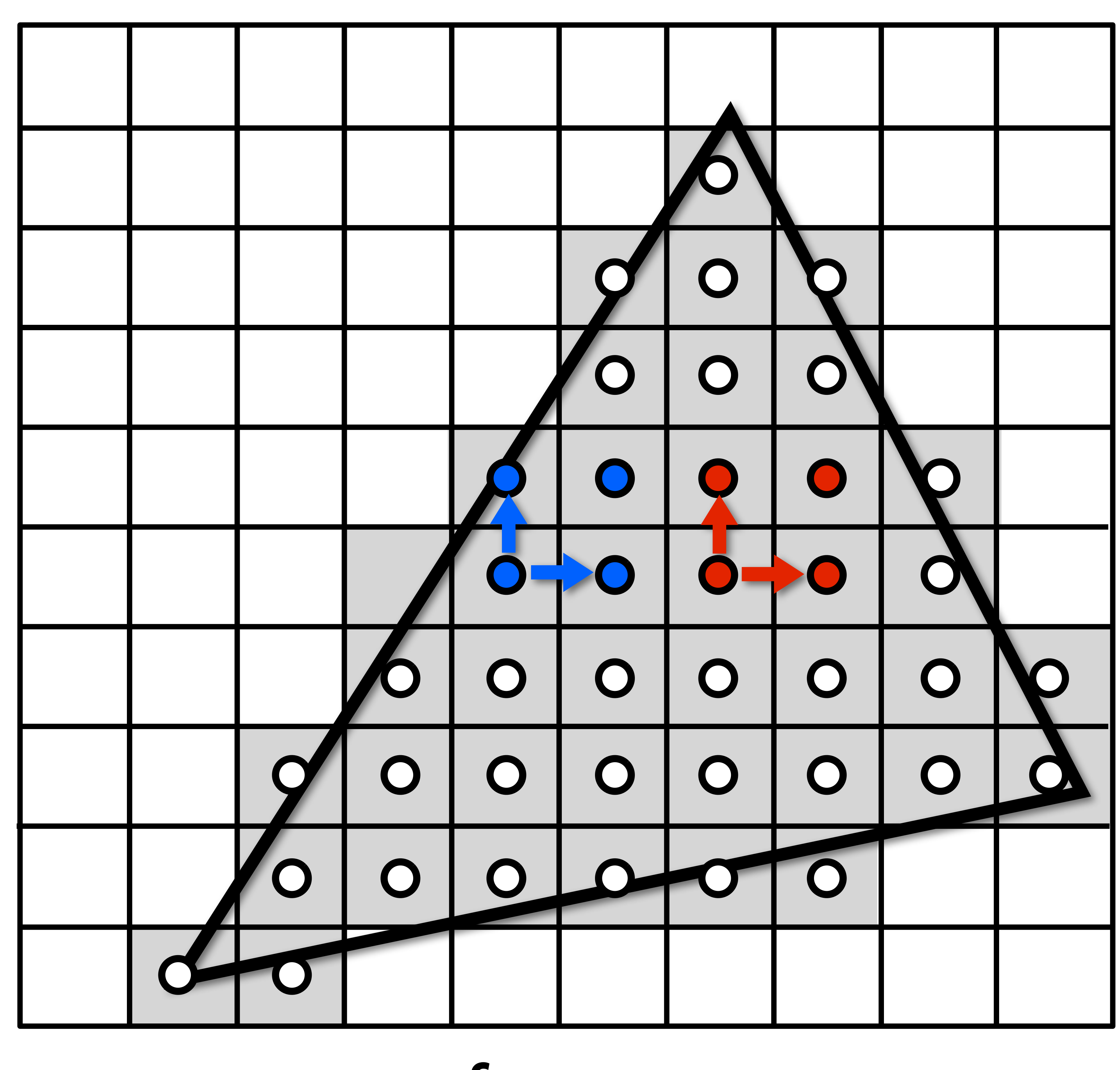


No pre-filtering (aliased result)

Pre-filtered texture

Computing amount of filtering

Take differences between texture coordinate values of neighboring fragments



Screen space

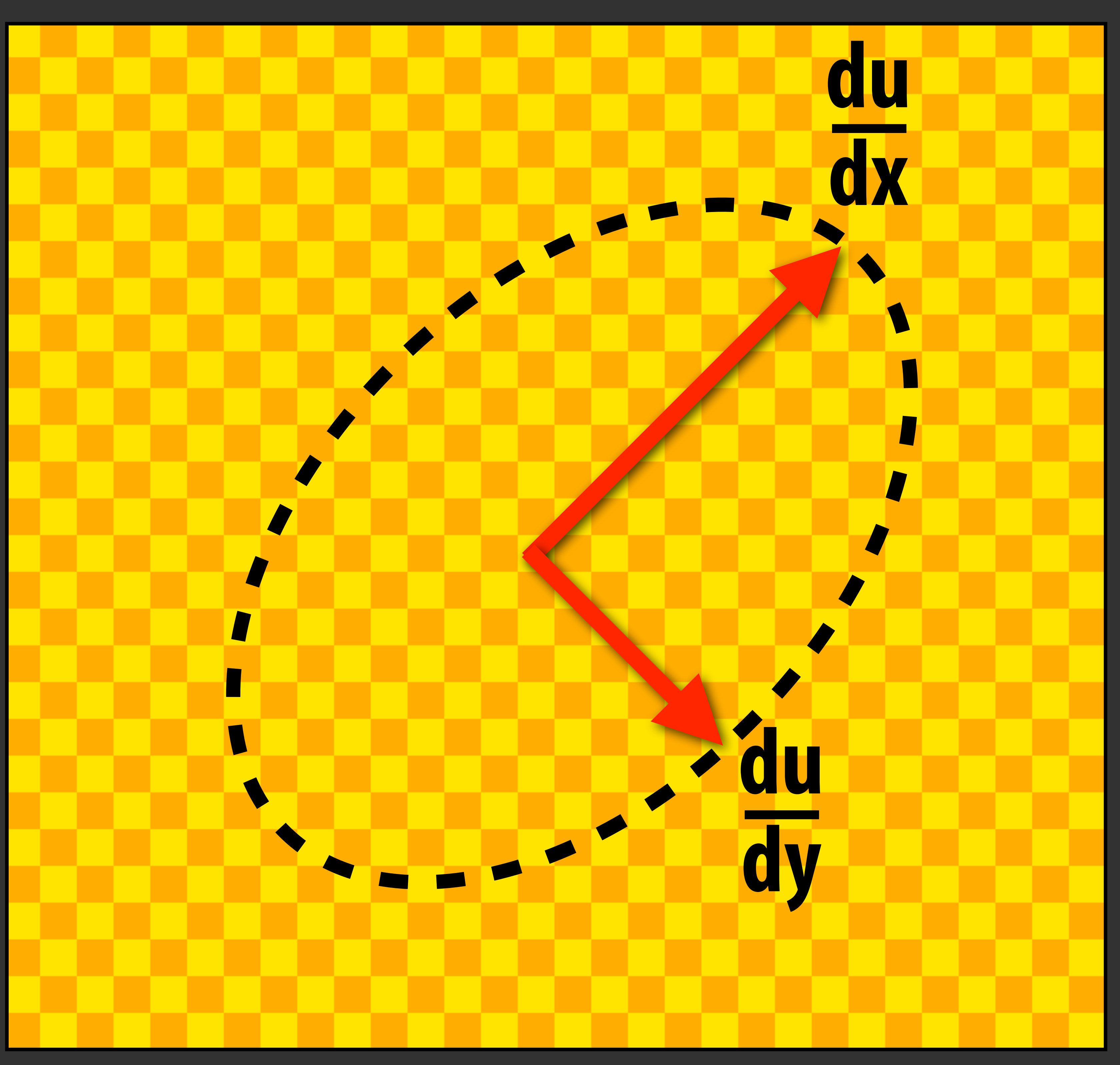
v

u

Texture space

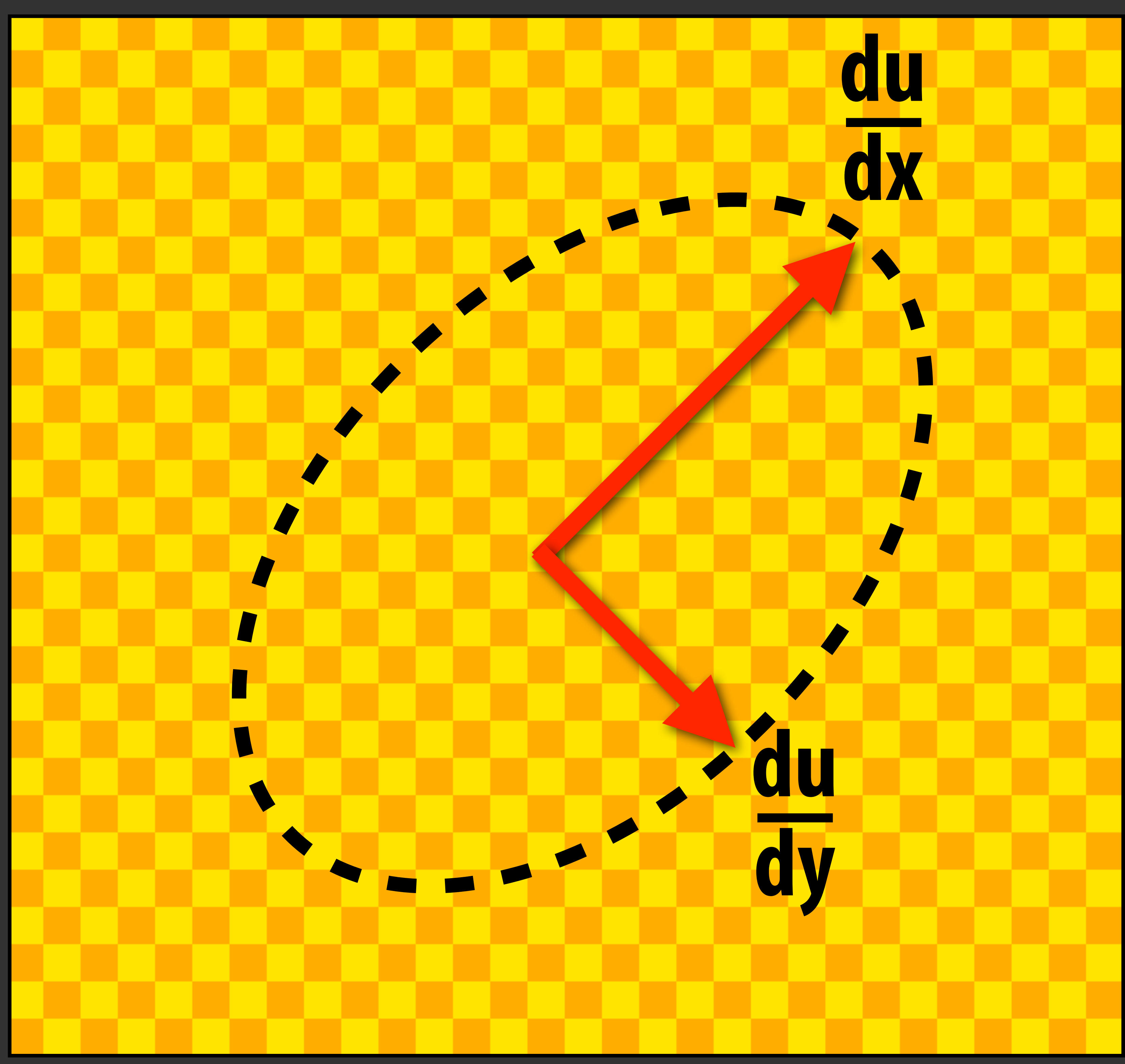
Surface derivatives are needed for texture filtering

Texture data

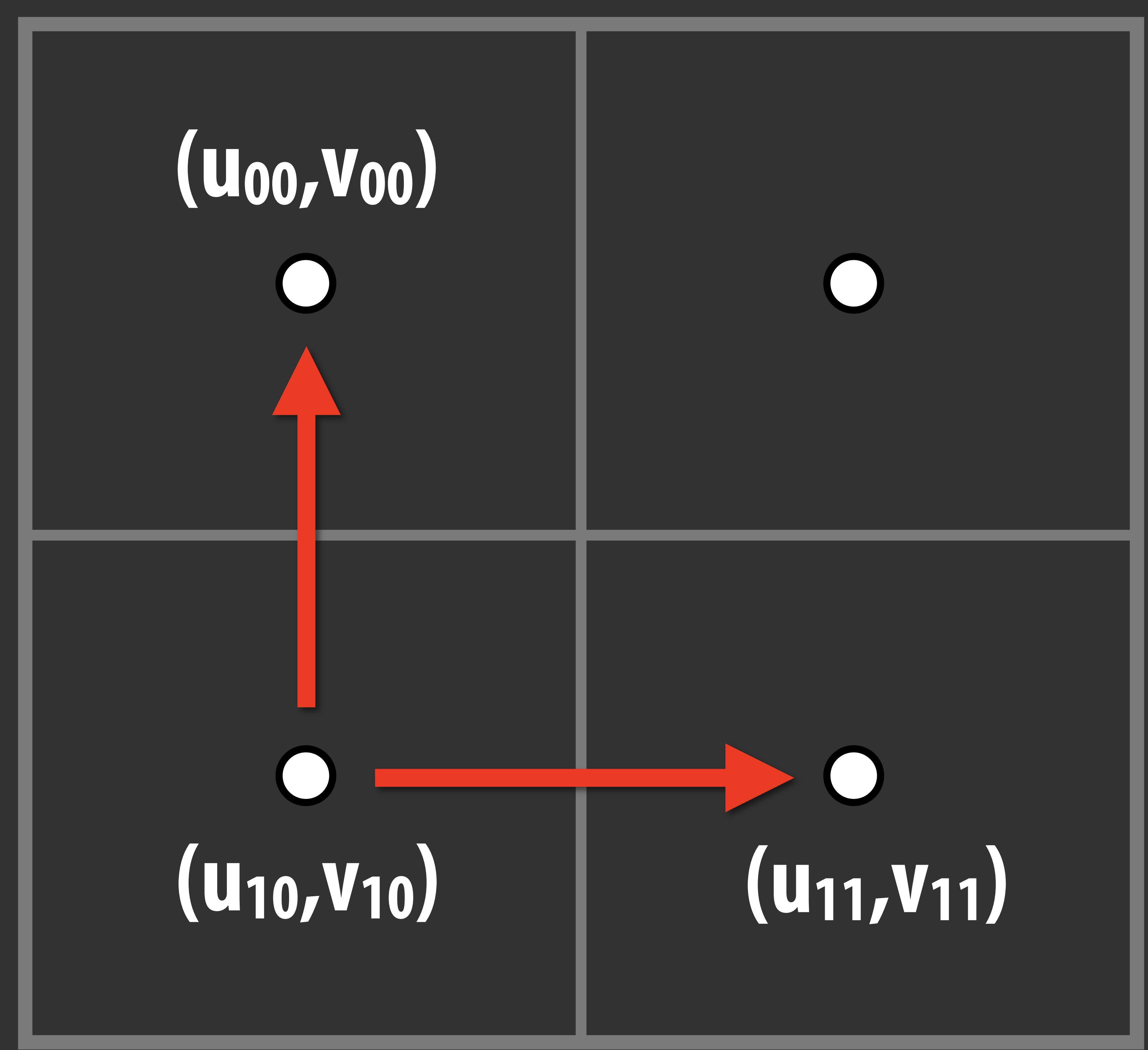


GPUs shade quad fragments (2x2 pixel blocks)

Texture data

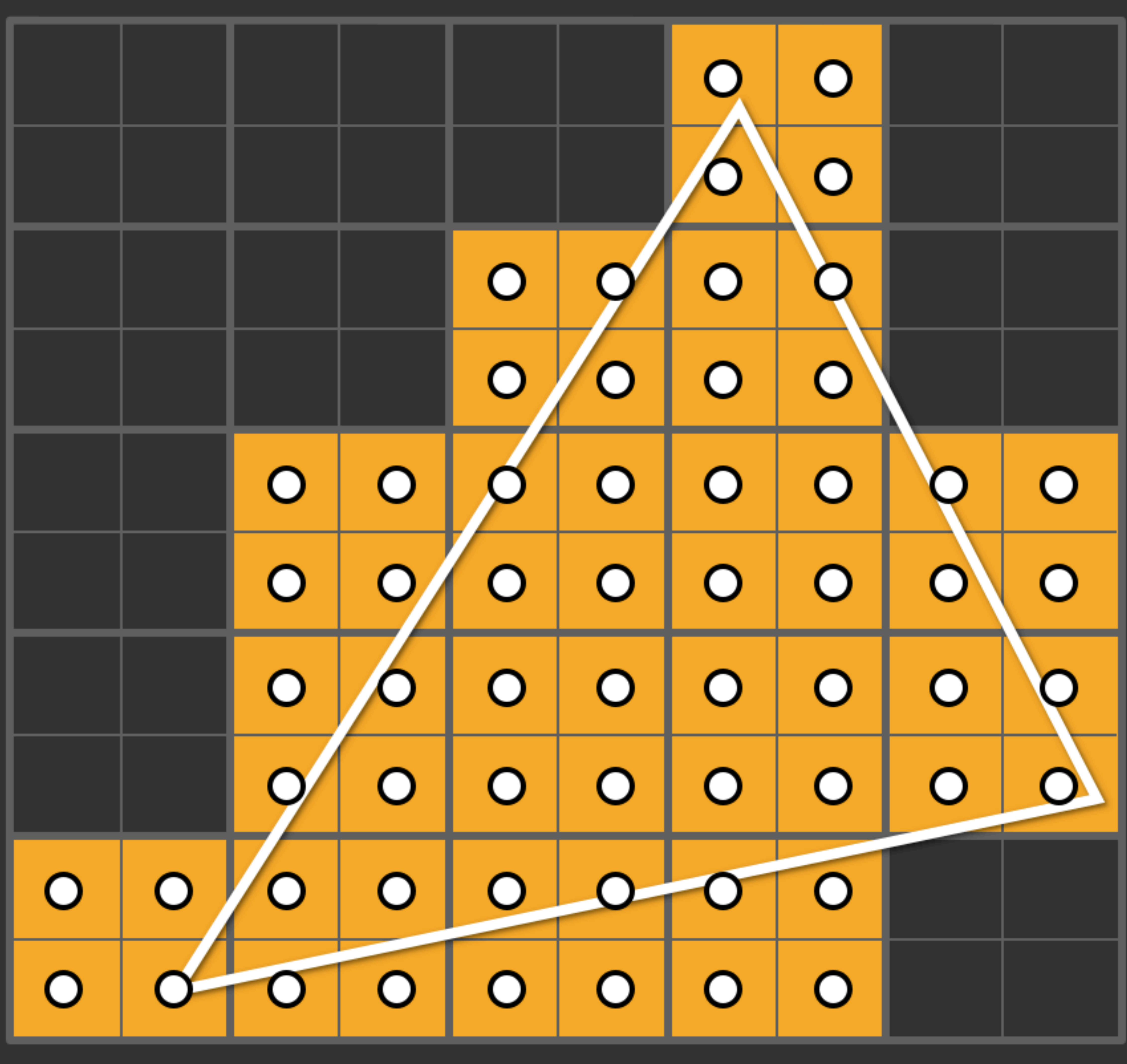


Quad fragment

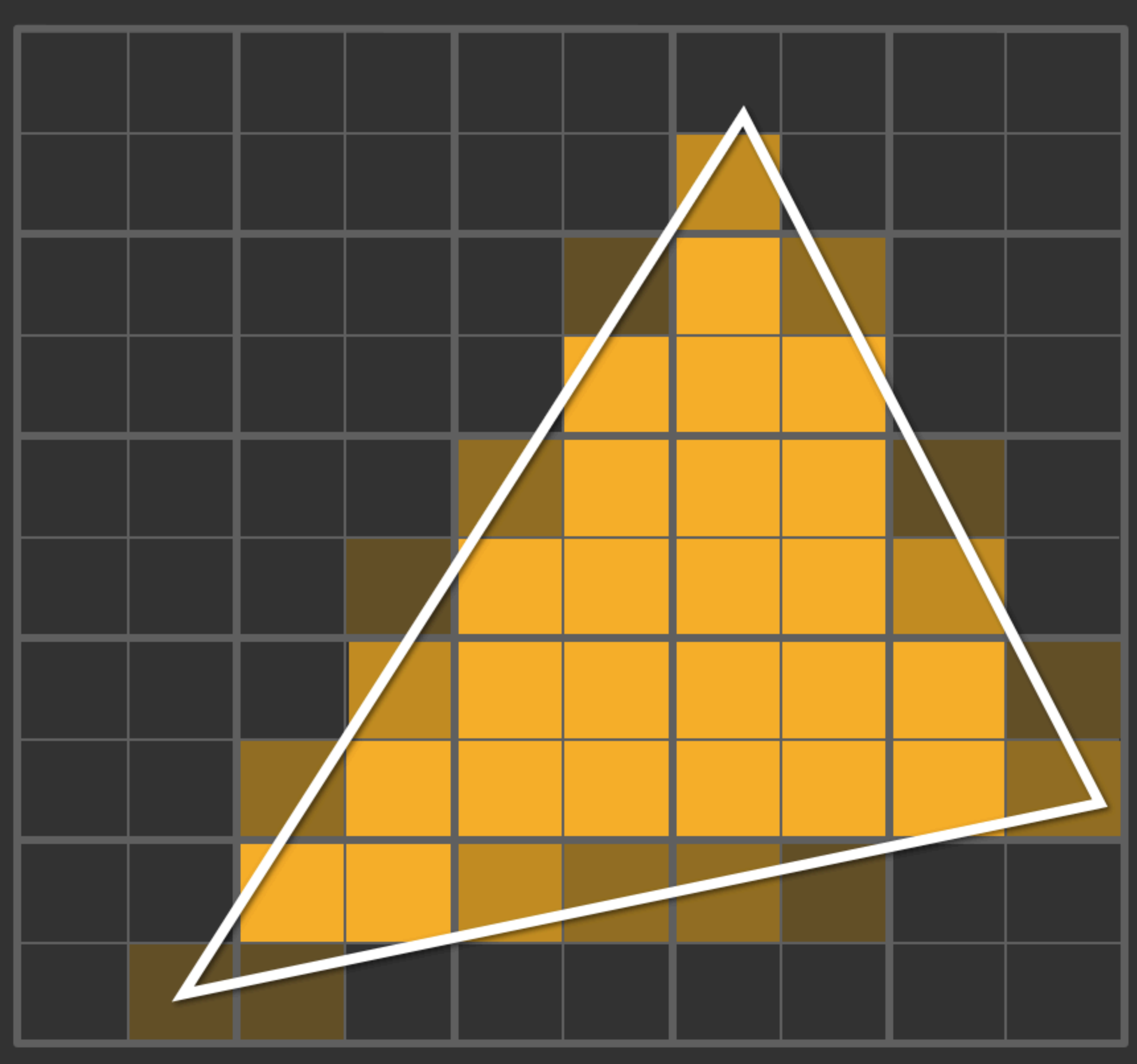


use differences between neighboring texture coordinates to estimate derivatives

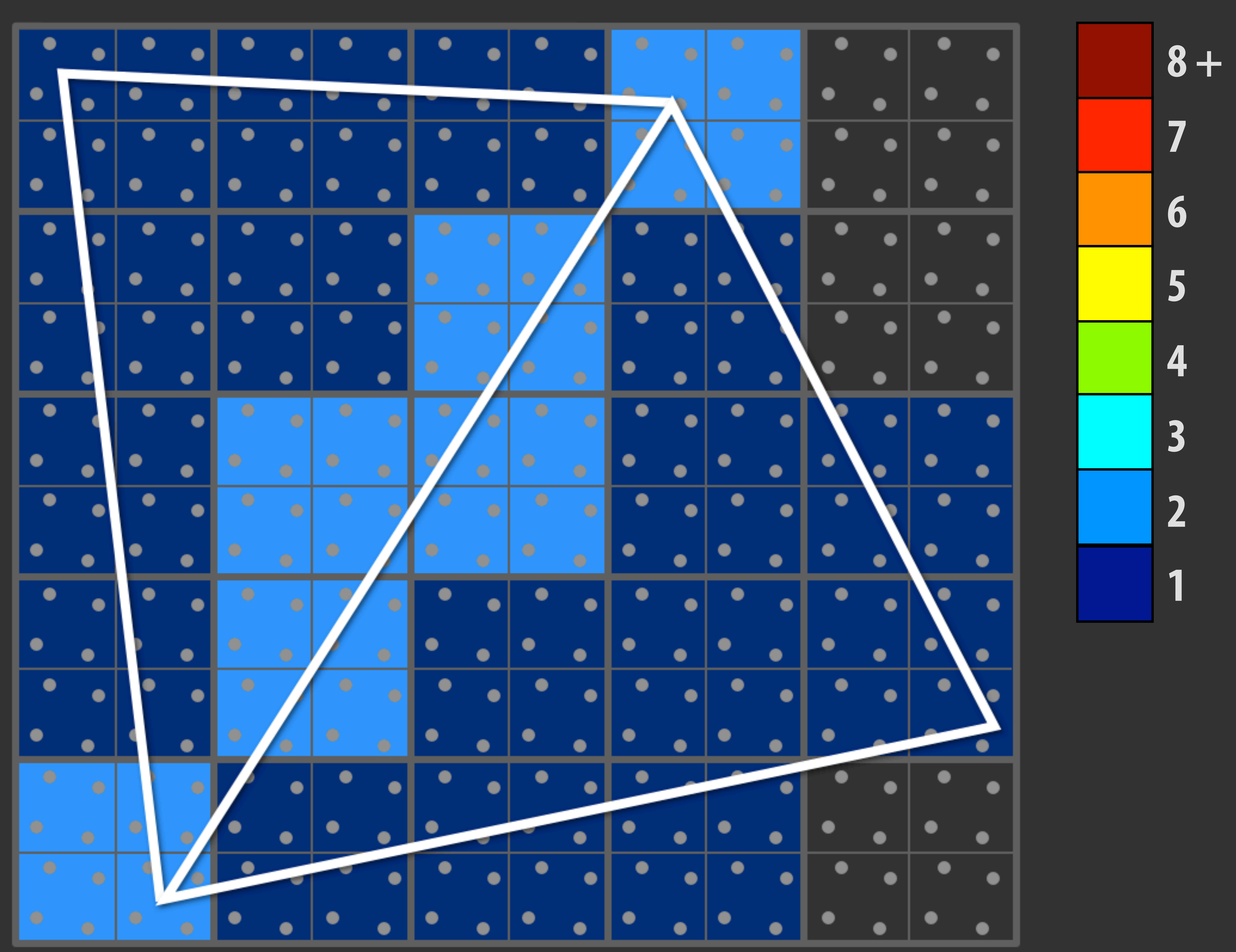
Shaded quad fragments



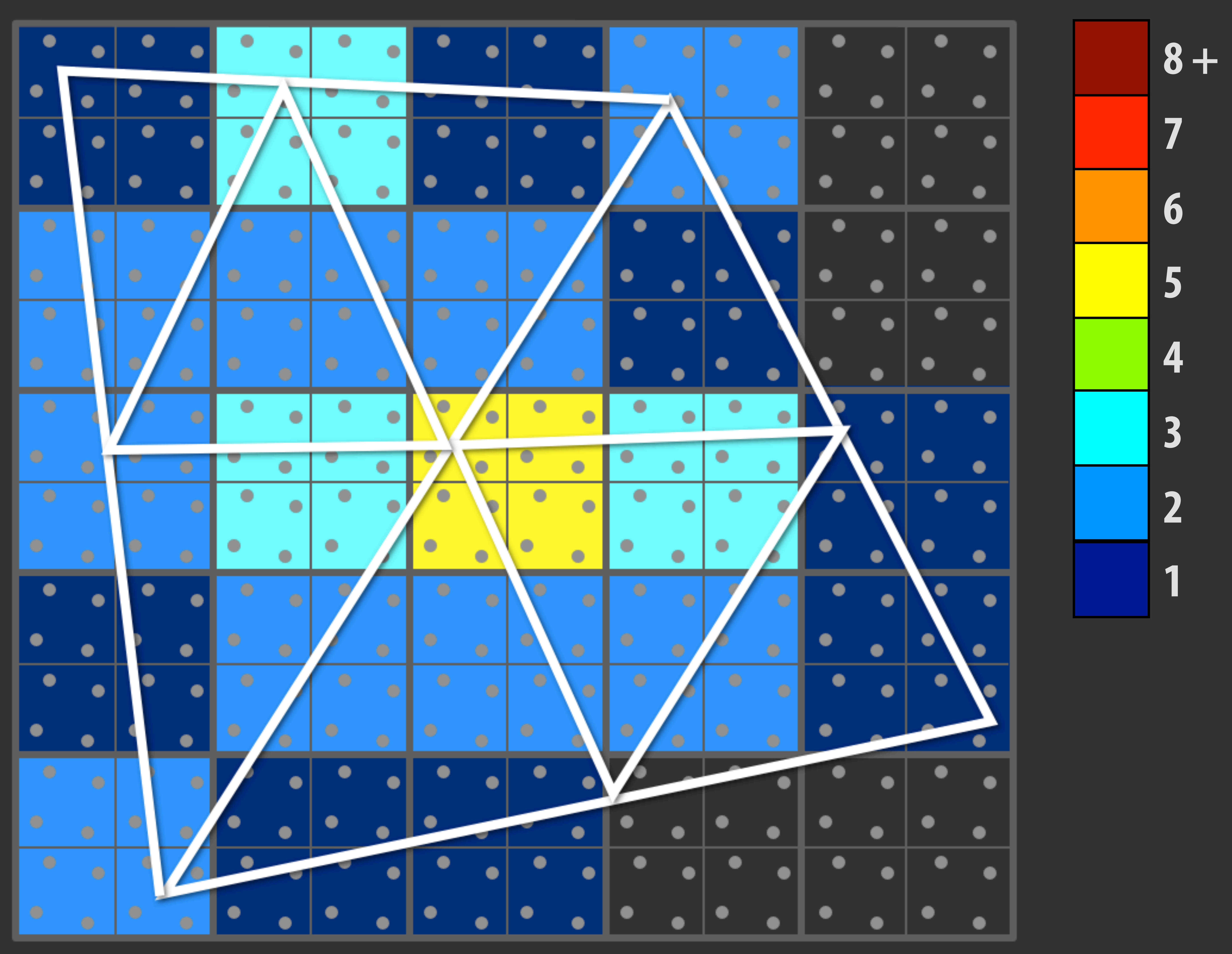
Final pixel values



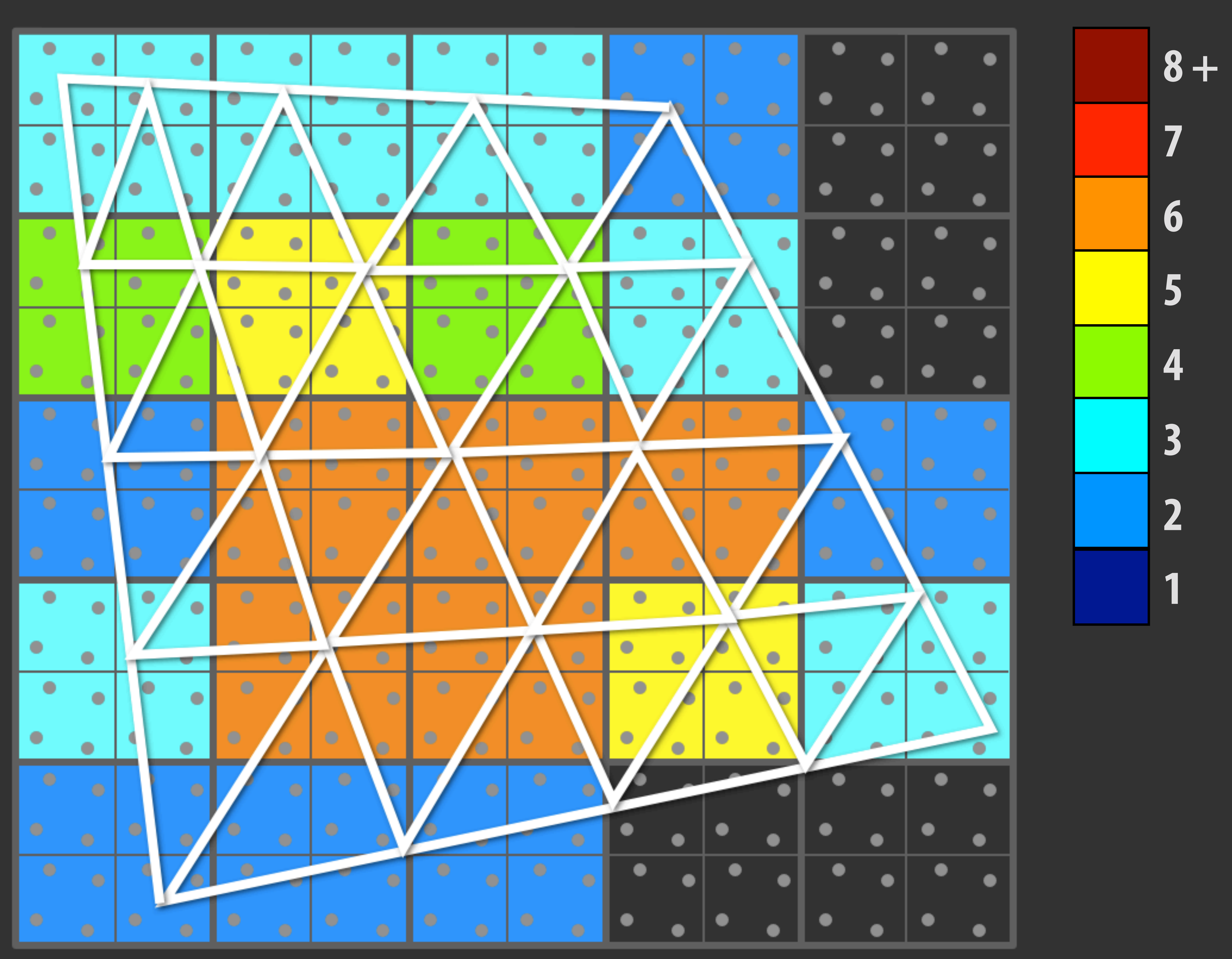
Pixels at triangle boundaries are shaded multiple times



Pixels at triangle boundaries are shaded multiple times

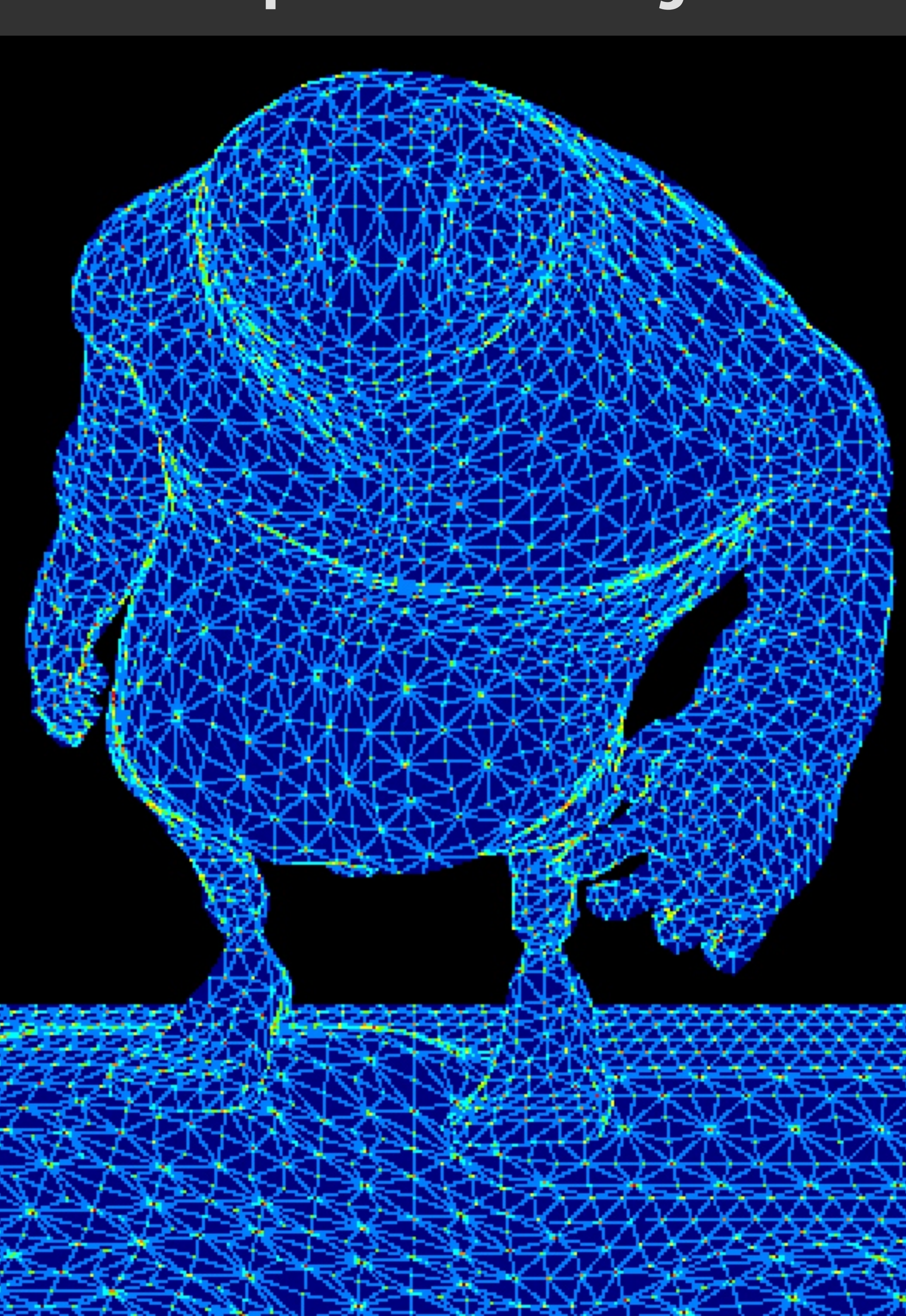


Pixels at triangle boundaries are shaded multiple times

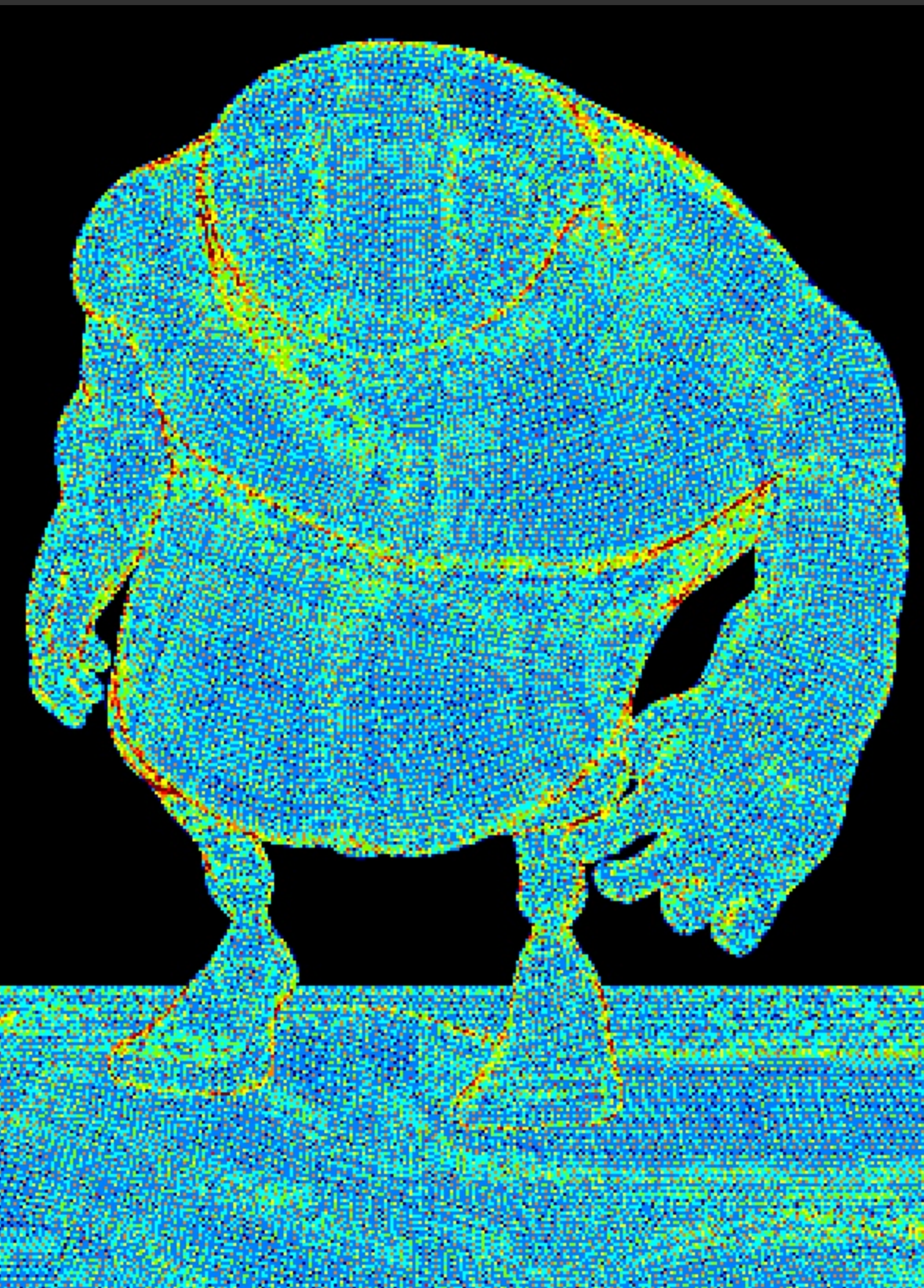


Small triangles result in extra shading

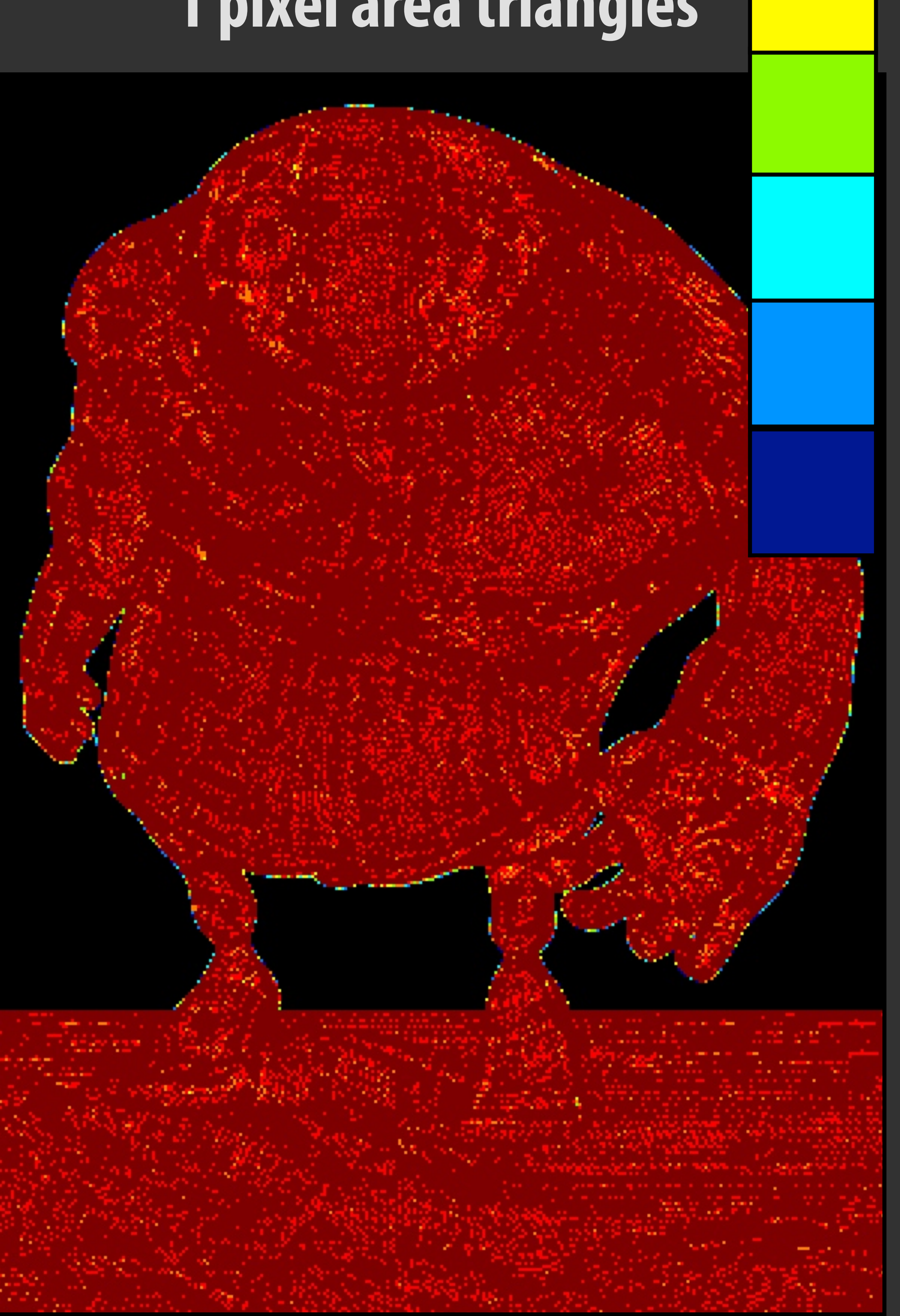
Shading computations per pixel



10 pixel area triangles



1 pixel area triangles



8 +
7
6
5
4
3
2
1

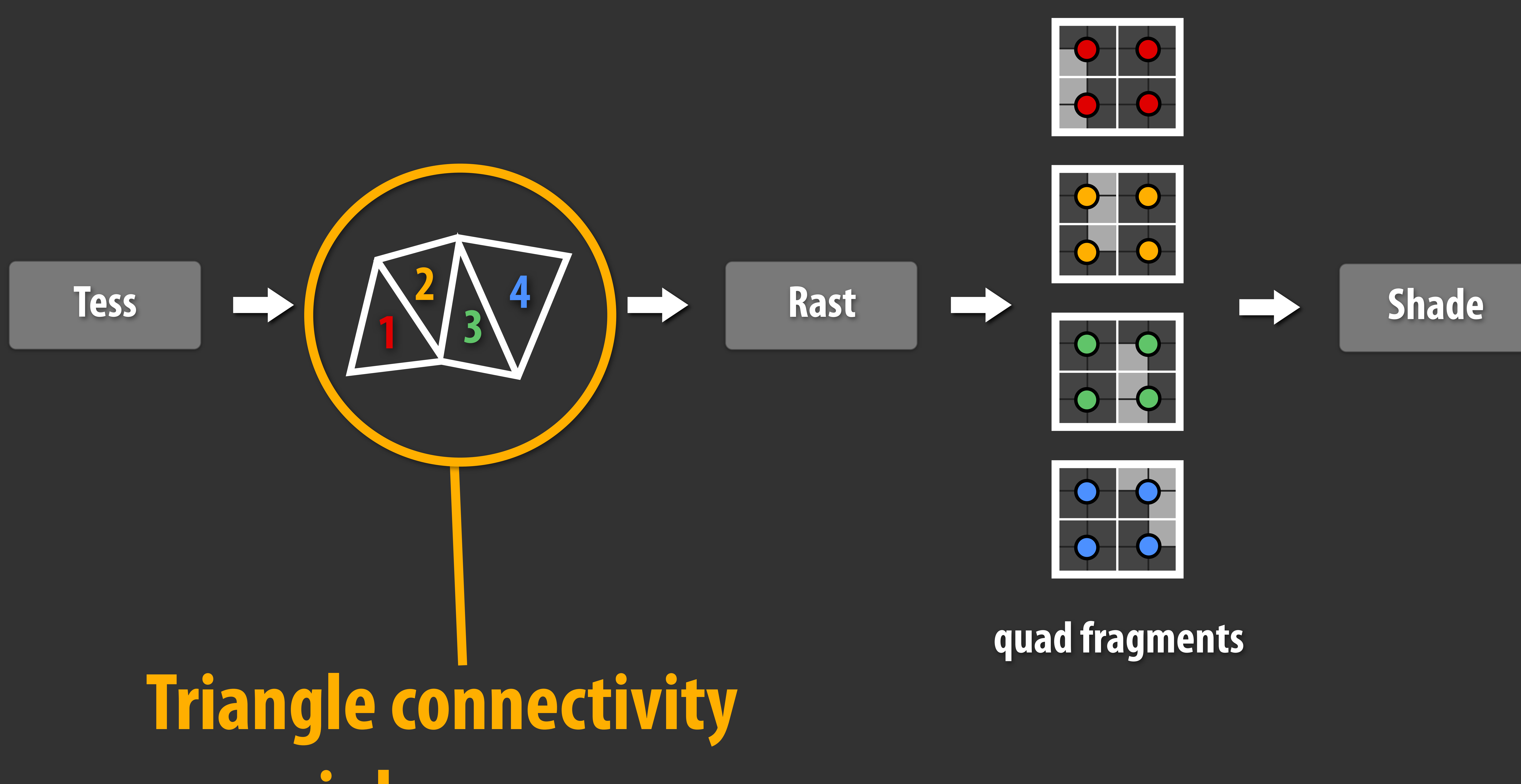
Goal:

Shade high-resolution meshes (not individual triangles) approximately once per pixel

Solution:

Quad-fragment merging

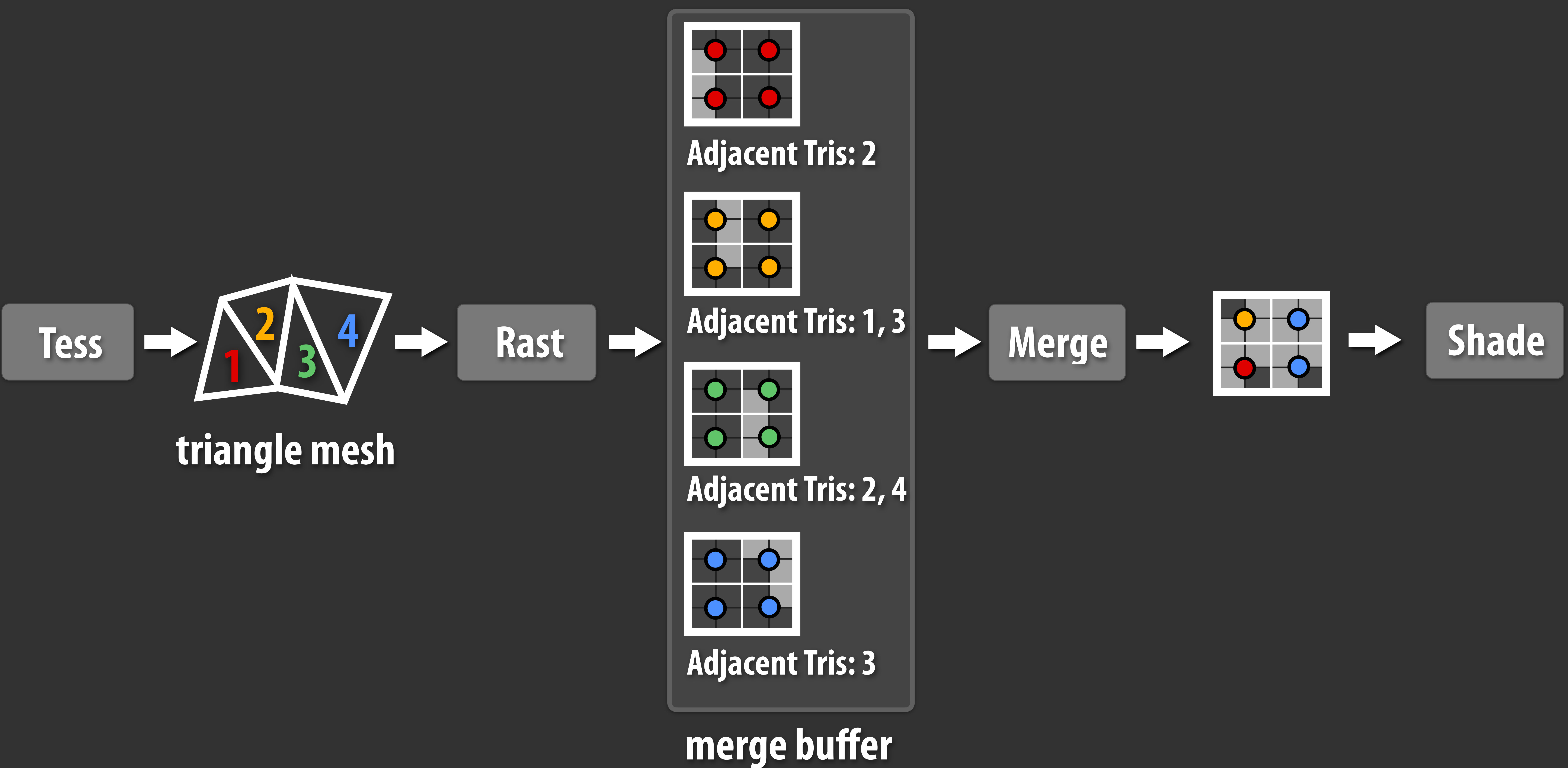
GPU pipeline: triangle connectivity is known



Pipeline with quad-fragment merging

[Fatahalian et al. SIGGRAPH 2010]

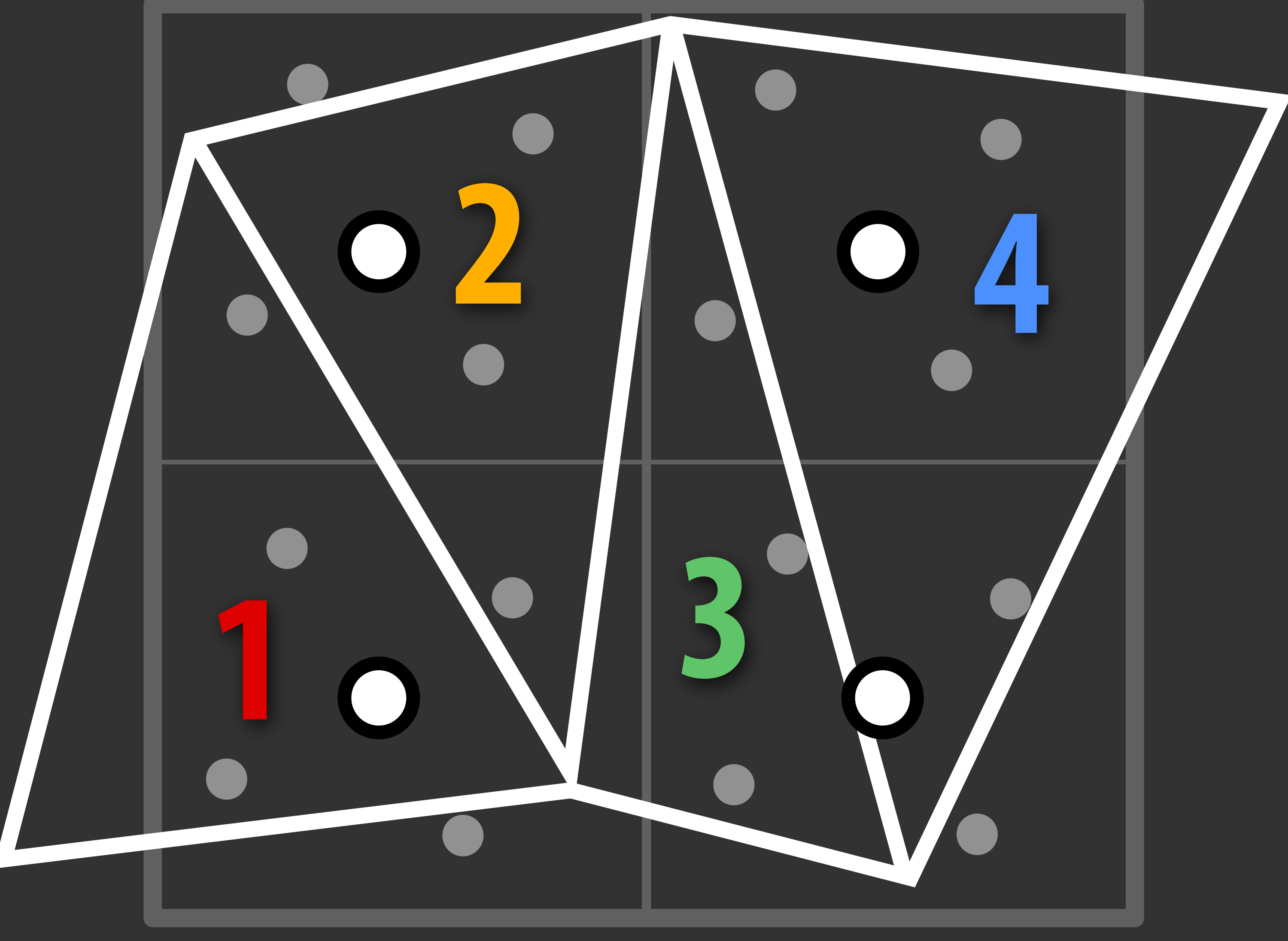
Pipeline with quad-fragment merging



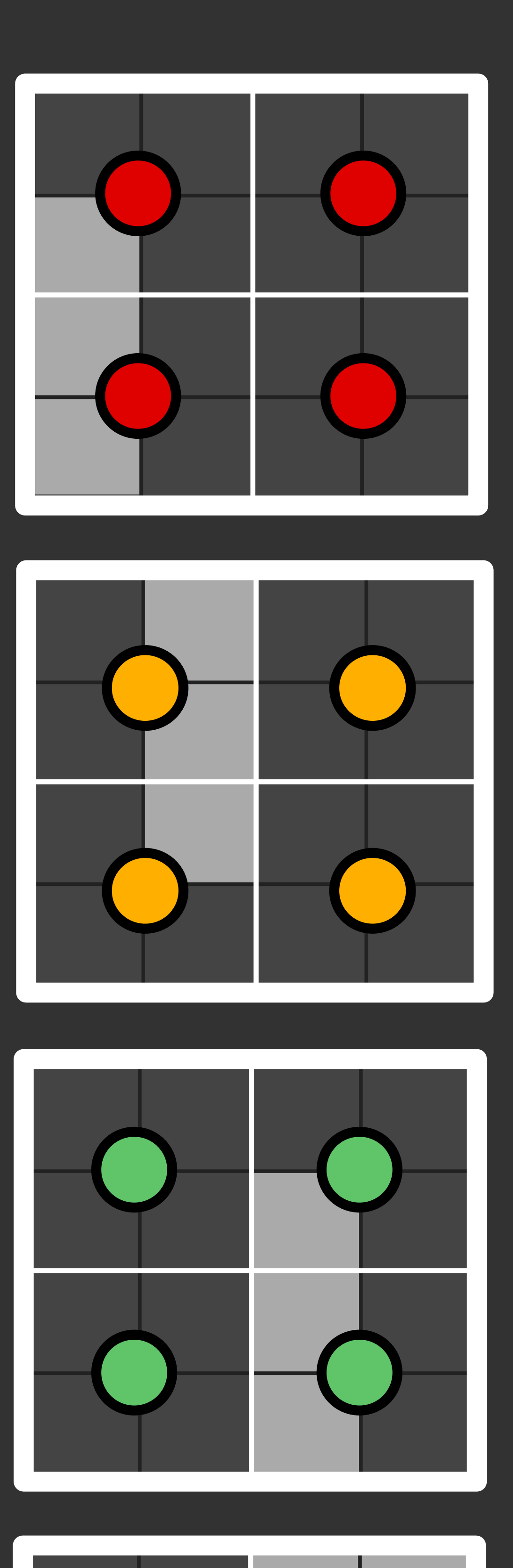
[Fatahalian et al. SIGGRAPH 2010]

How to merge quad fragments

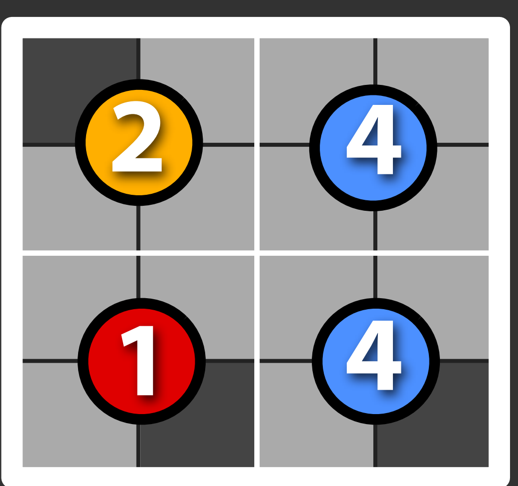
Mesh triangles



Rasterized quad fragments



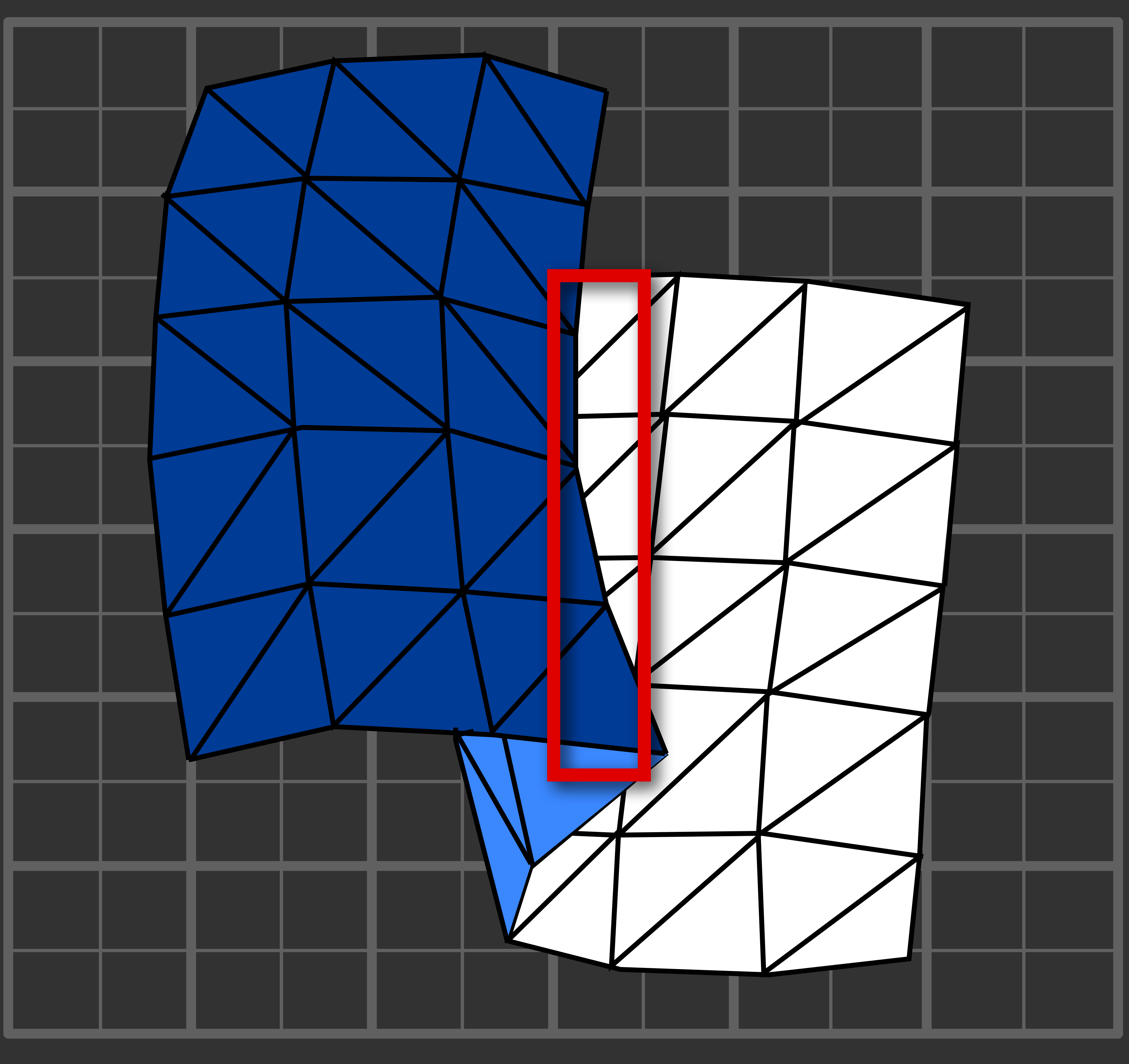
Merged quad fragment

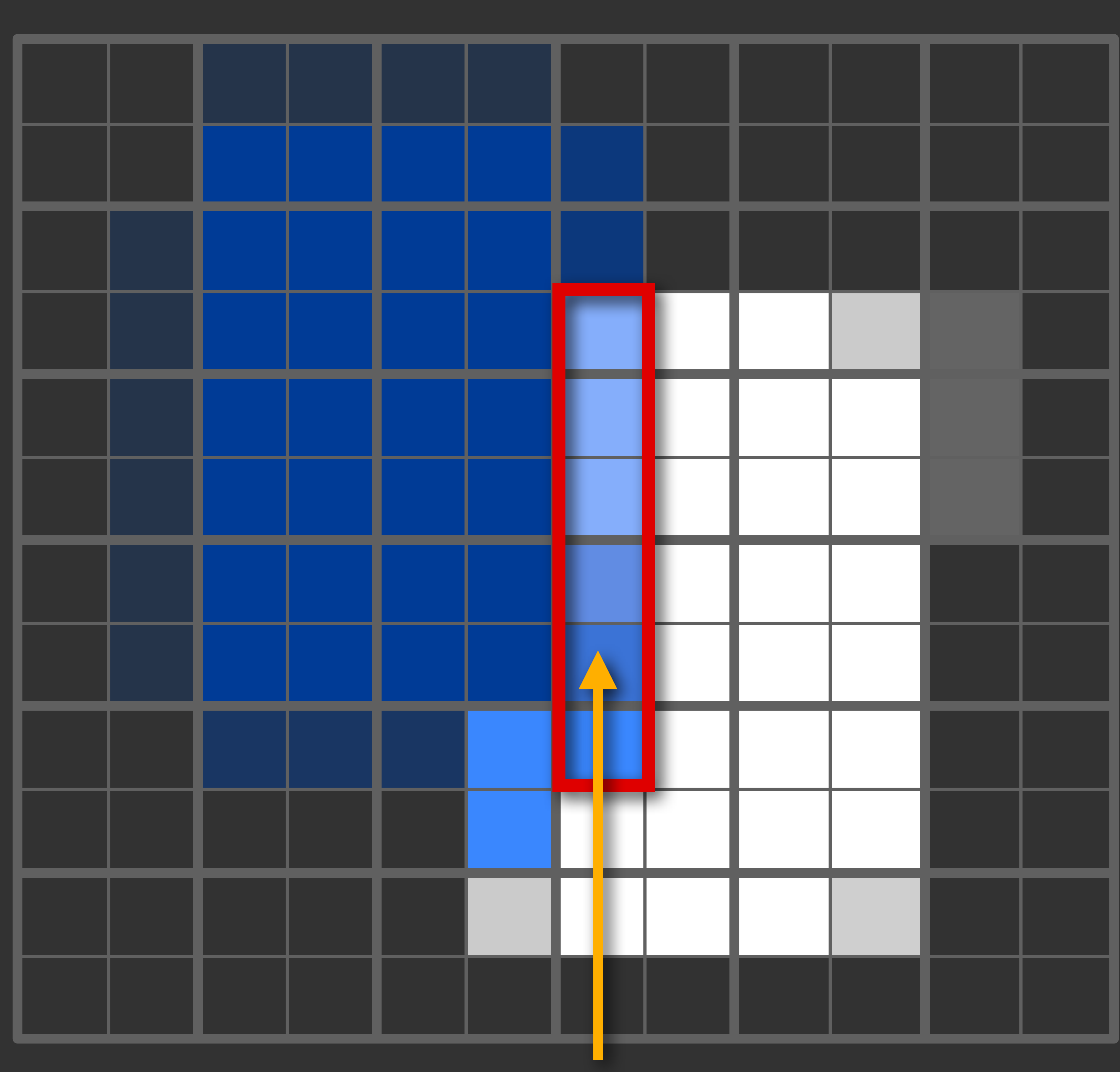


When to merge quad fragments

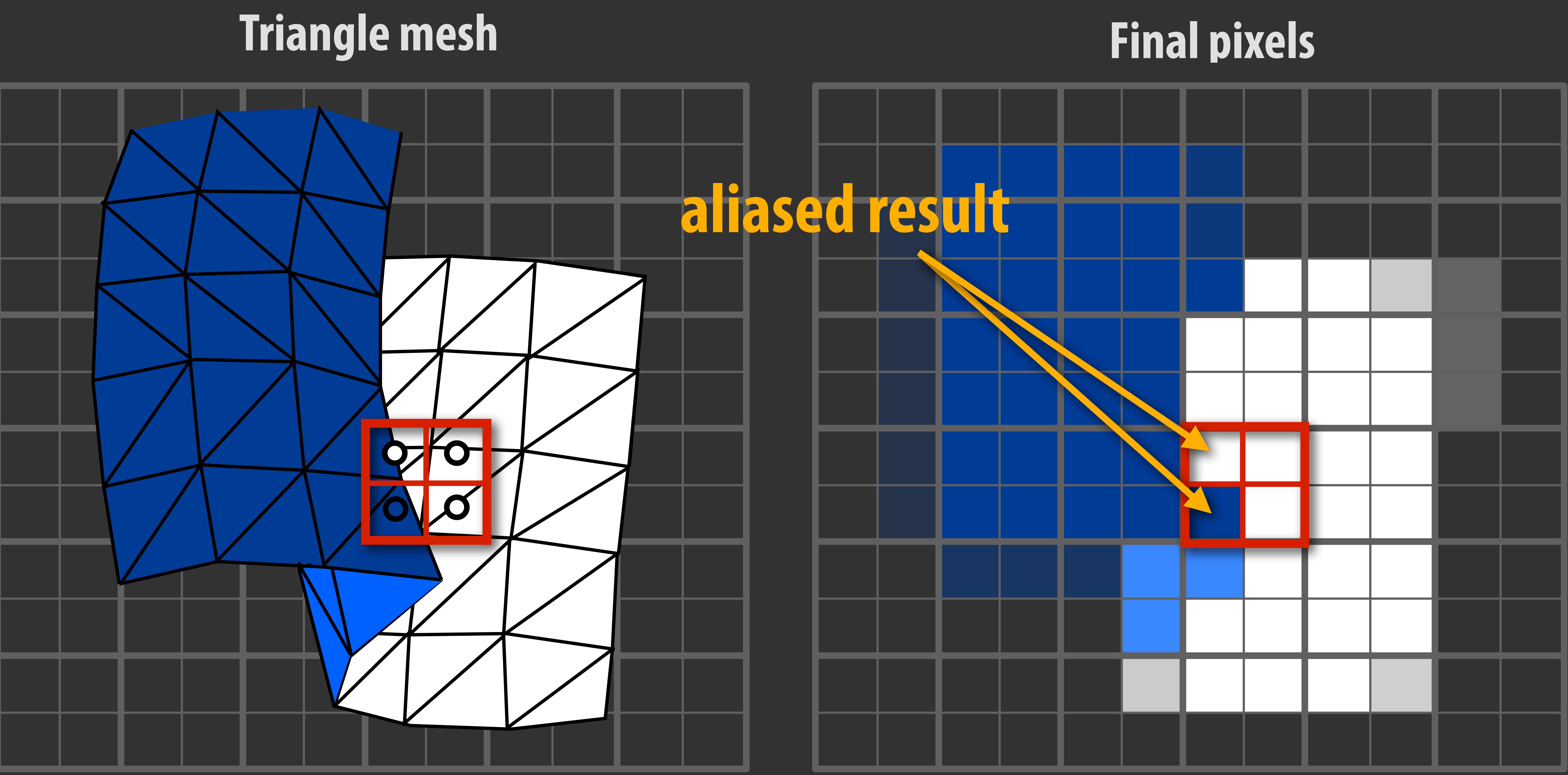
Challenge: avoiding merges that introduce visual artifacts

Example: surface with a silhouette





Naive merging results in aliasing



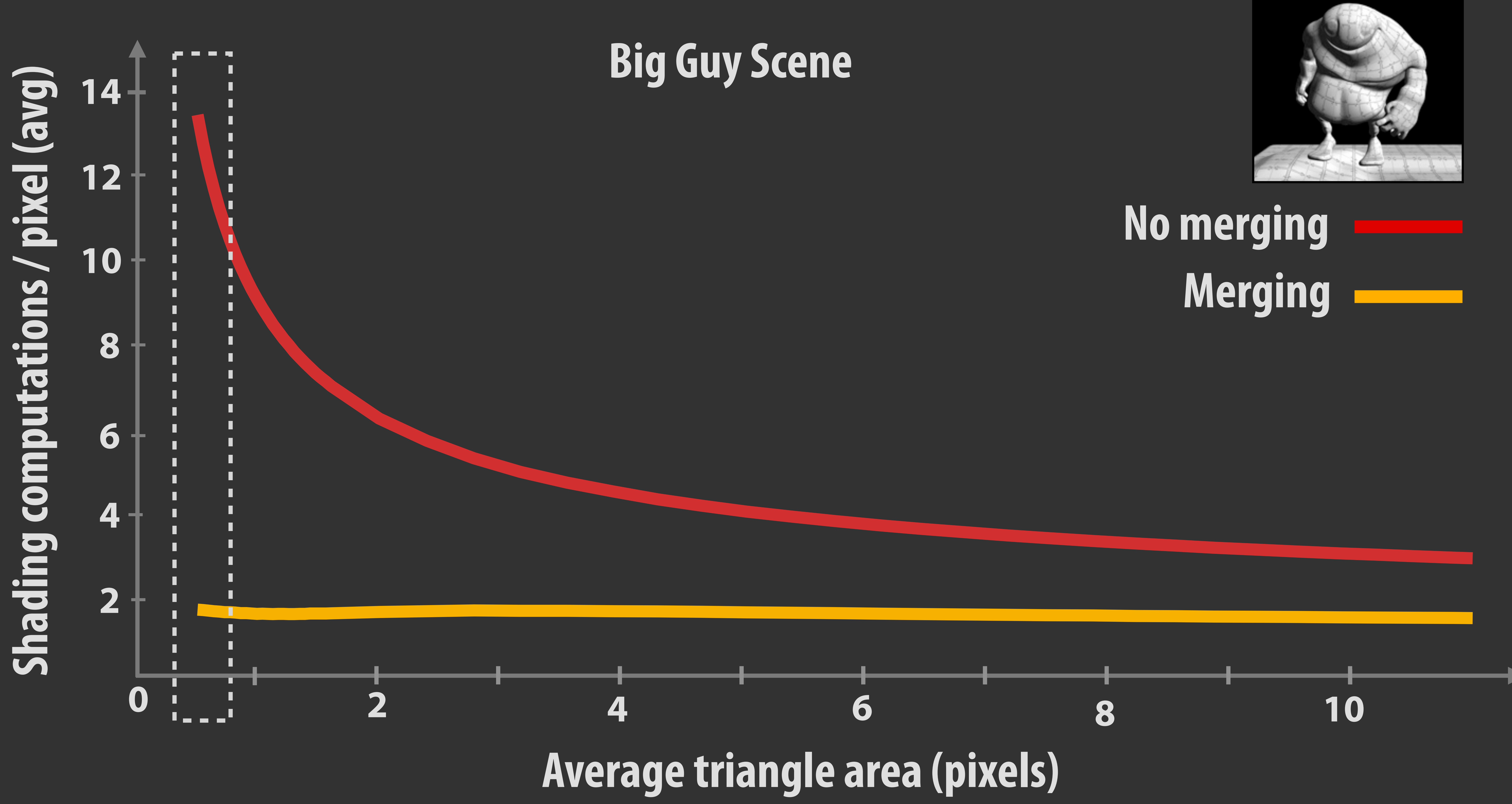
Only merge quad-fragments from adjacent triangles in mesh

Implementation: the cost of merging is low

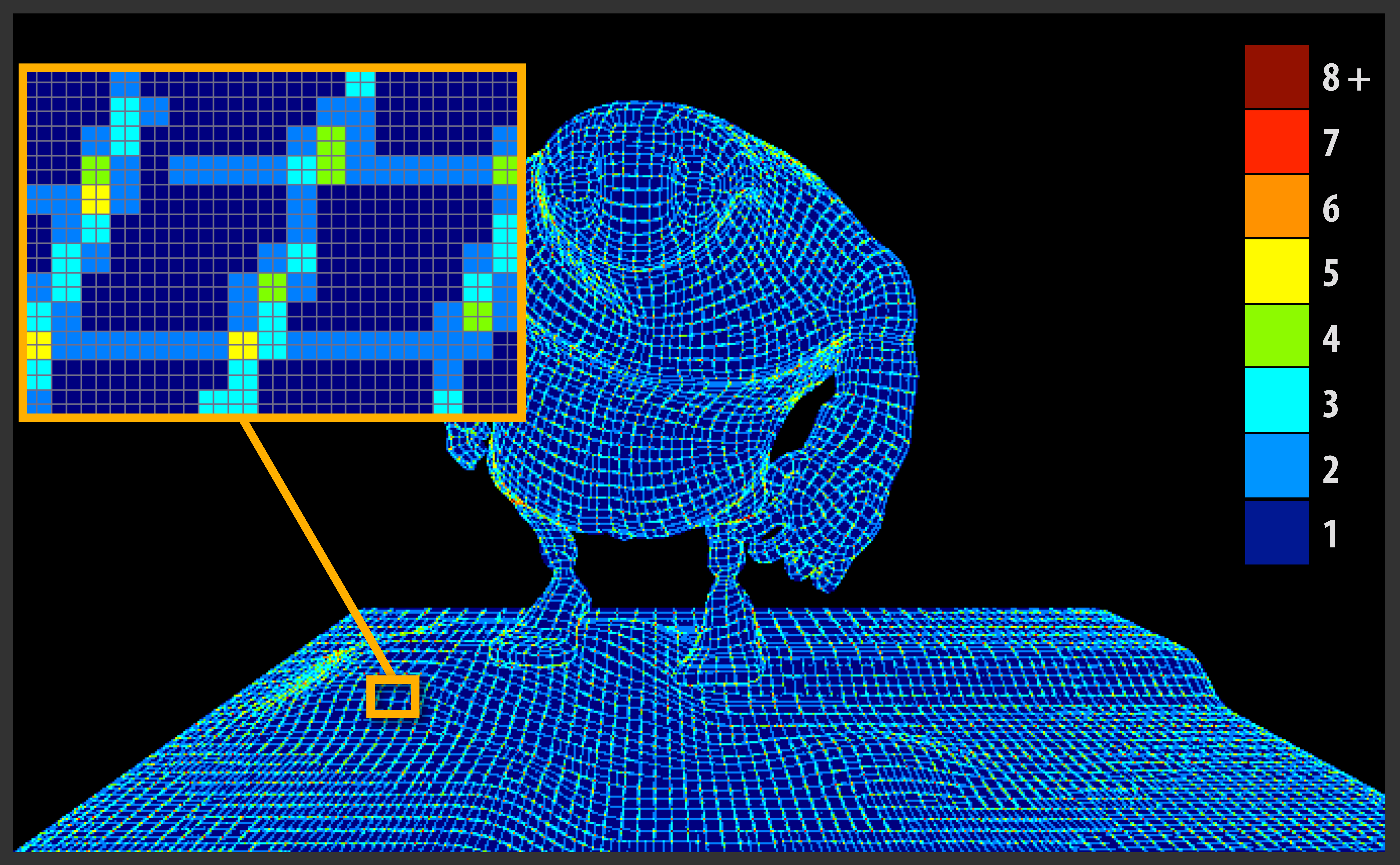
- Merging operations are cheap
  - testing merging rules requires only bitwise operations
  - each triangle carries a bit mask with adjacent triangle ids set
- Merge buffer is small
  - 32 quad fragment merge buffer is very effective
  - 90% of all possible merges
- Expectation: quad-fragment merging can be encapsulated in fixed-function hardware

Merging reduces total shaded quad fragments

1/2-pixel-area triangles: 8x reduction



Extra shading occurs at merging window boundaries



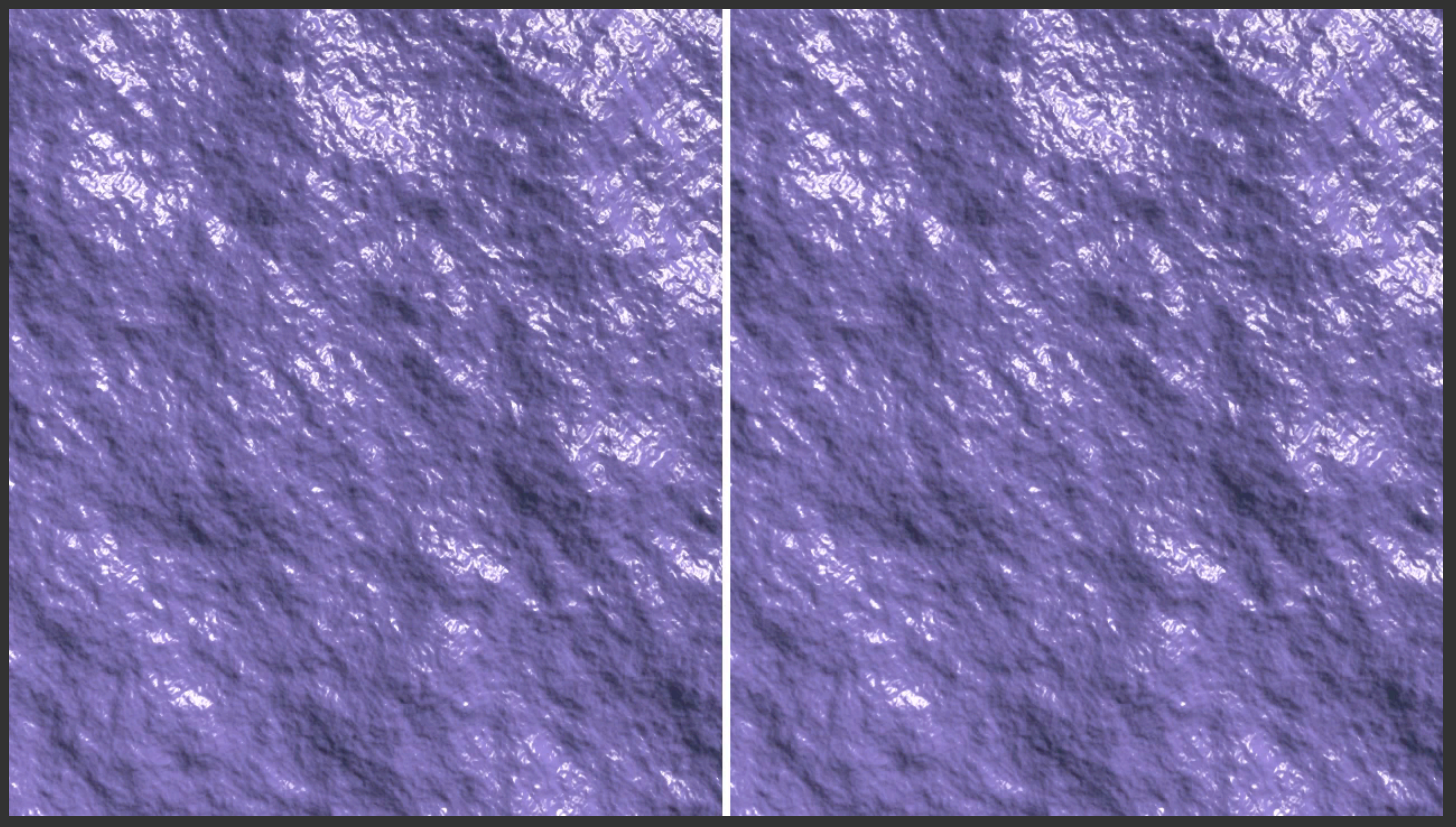
Nearly identical visual quality

Quad-fragment merging

Current GPU (no merging)

Nearly identical visual quality

Quad-fragment merging | **Current GPU (no merging)**



Quad-fragment merging summary

- Reduces shading costs for high-res meshes
  - shade surfaces (not triangles) at a density of once per pixel

- Images not identical, but maintains high visual quality
  - Requires triangle connectivity

- Evolutionary: not a radical change to rasterization or shading
  - isolates dynamic communication/control in merge step, maintains data-parallel shading
  - uses quad fragments for derivatives (still efficient for big triangles)
  - compatible with edge anti-aliasing
  - supports shading large triangles

SYSTEM-WIDE INTERACTIONS

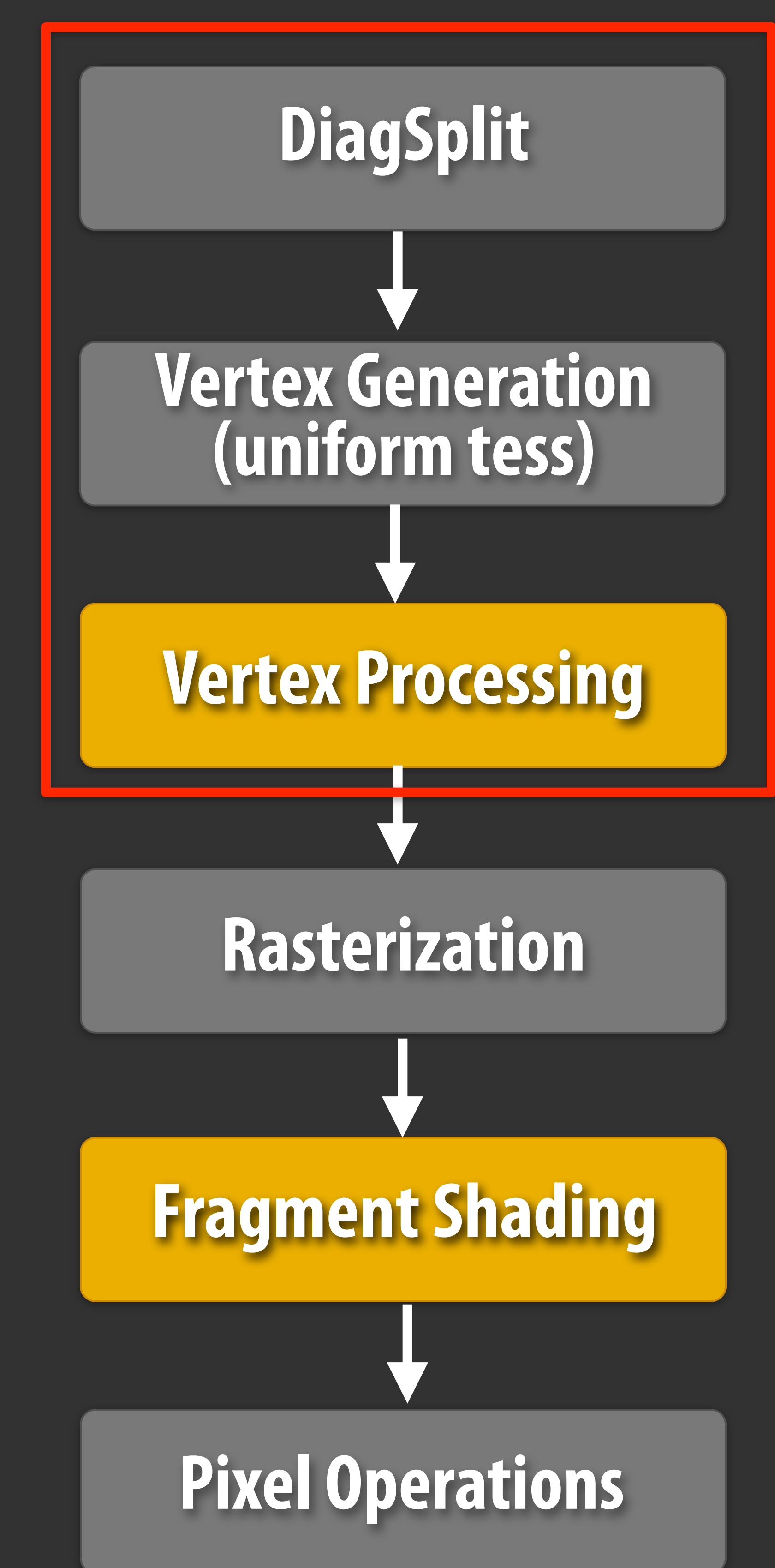
A micropolygon rendering pipeline

DiagSplit adaptive tessellation:

Reduces rendered vertex count

Simplifies micropolygon-parallel rasterization

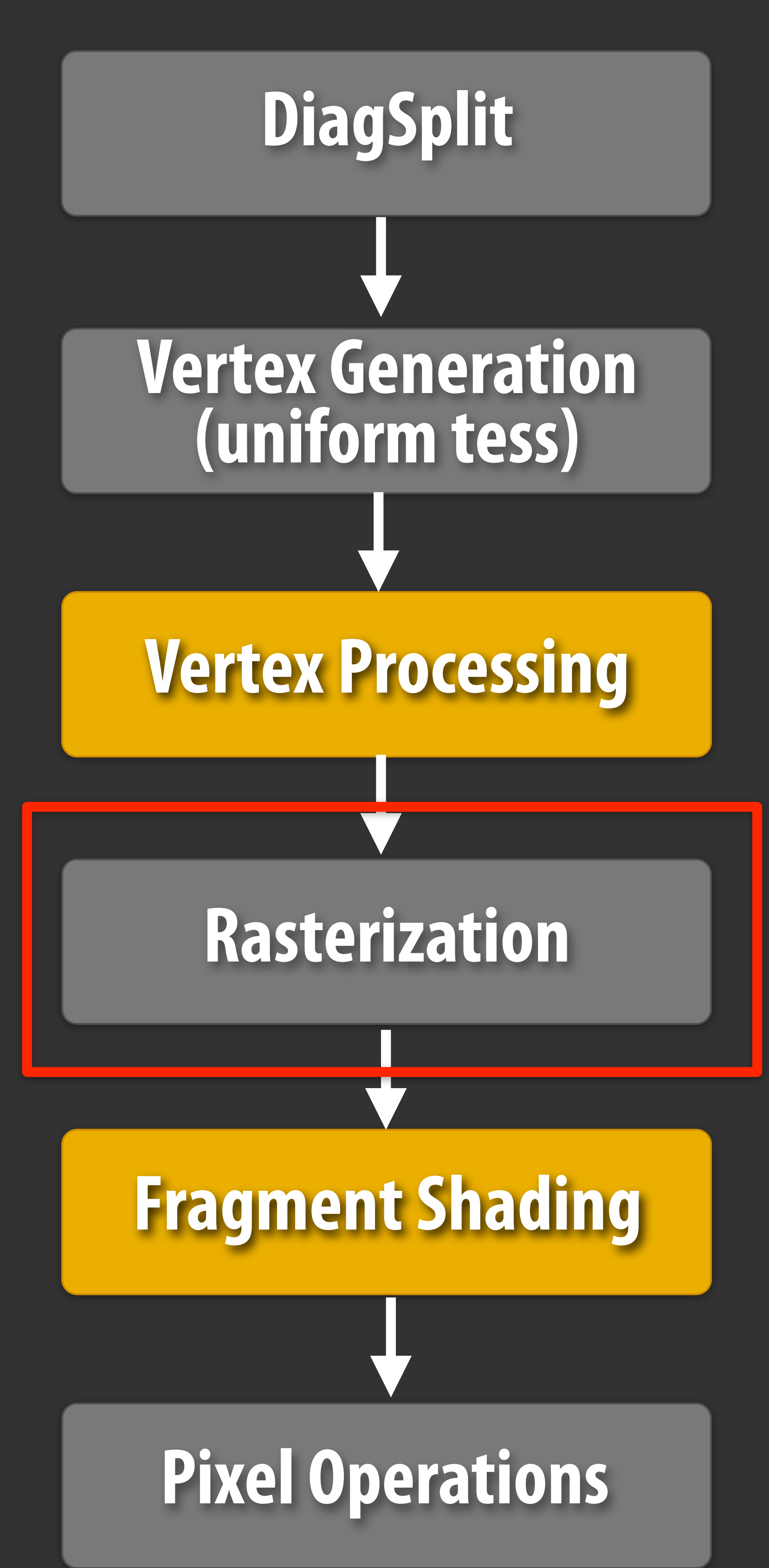
Makes quad-fragment merging practical
(provides topology, sets triangle order)



A micropolygon rendering pipeline

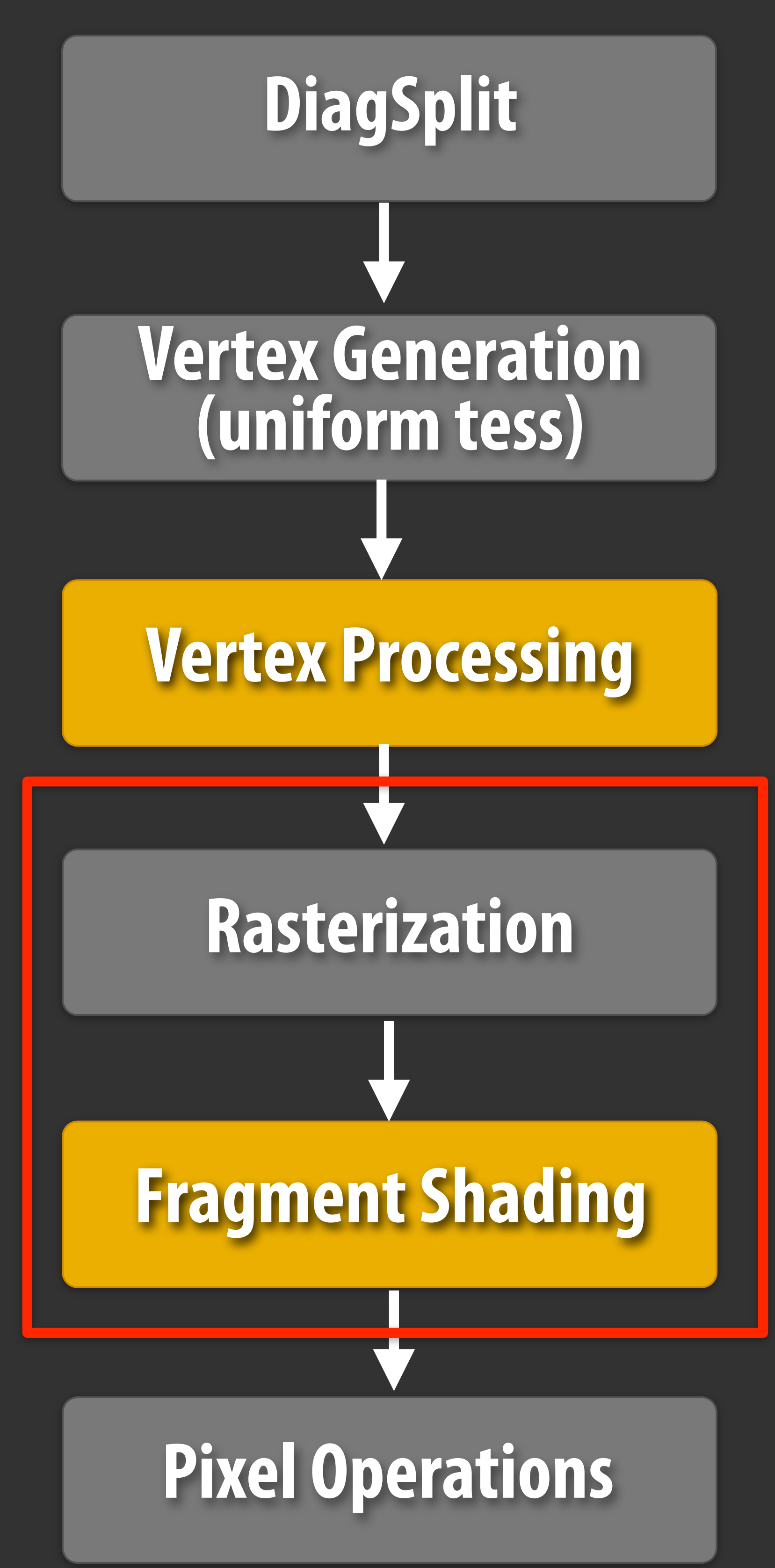
Rasterization:

Simple, but expensive: fixed-function hardware highly desirable



A micropolygon rendering pipeline

Quad-fragment merging:



Reduces shaded fragments by 8x

Not a radical change to existing rasterization and shading systems

Output quality very similar to that of current GPUs

Domain knowledge in graphics system design

1. **Willingness to change algorithms to fit the system**
   - Natural for a field where output simply must "look good"

2. **Unique approach to exploiting heterogeneity**
   - Isolate irregularity, synchronization in non-programmable regions
   - Keep programmable stuff regular (and easy to code)
   - Programmable "stuff" forms the inner loops! (admittedly odd)

Hot questions

What is the future of the real-time graphics pipeline?

(continue to evolve? or replace?)

How can graphics systems continue to leverage fixed-function processing, but place it under software control?

Plug

- Real-time computer graphics presents some really challenging parallel systems problems
- Ditto for computational photography and computer vision