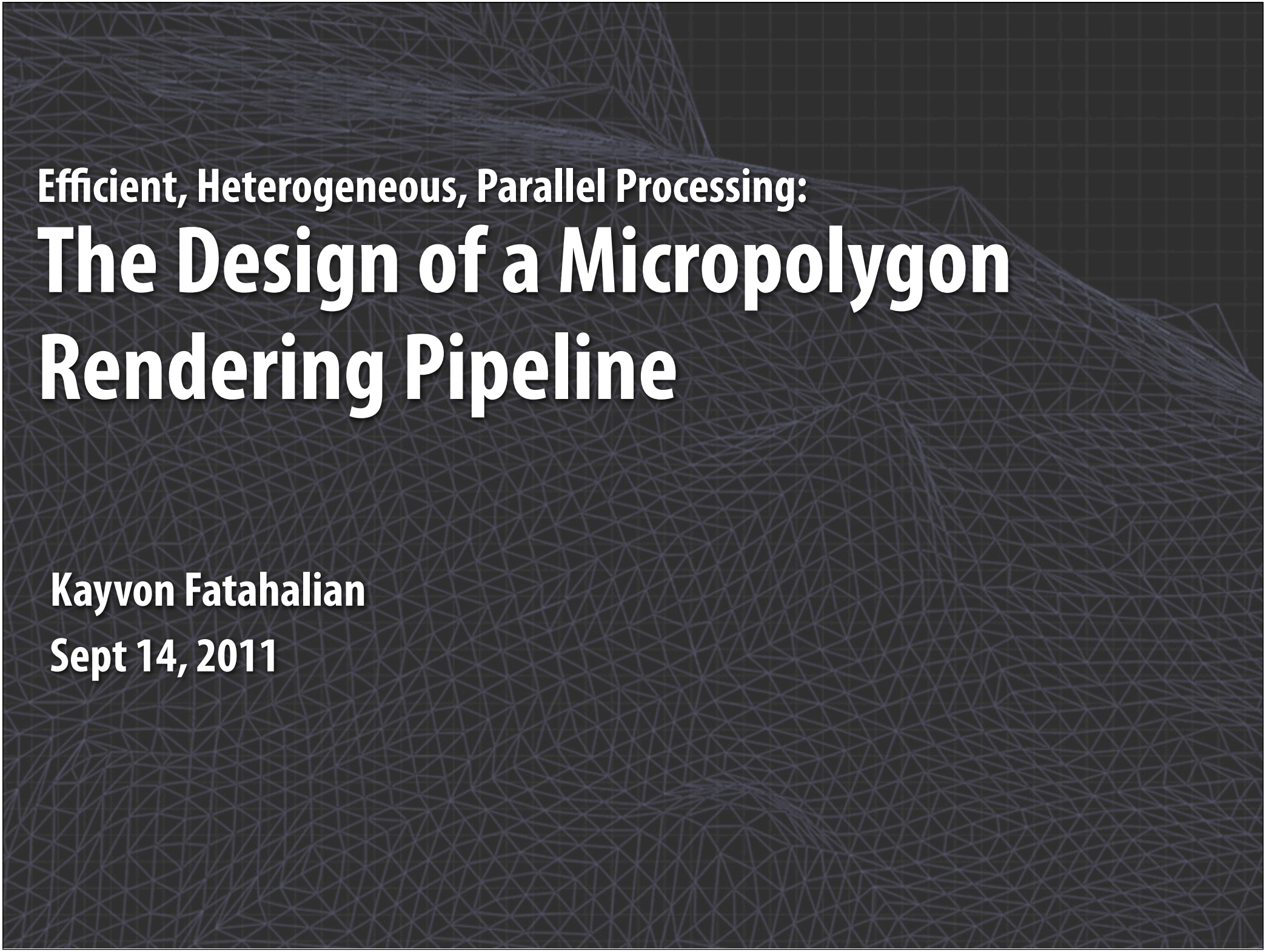


Efficient, Heterogeneous, Parallel Processing:

# The Design of a Micropolygon Rendering Pipeline

Kayvon Fatahalian

Sept 14, 2011

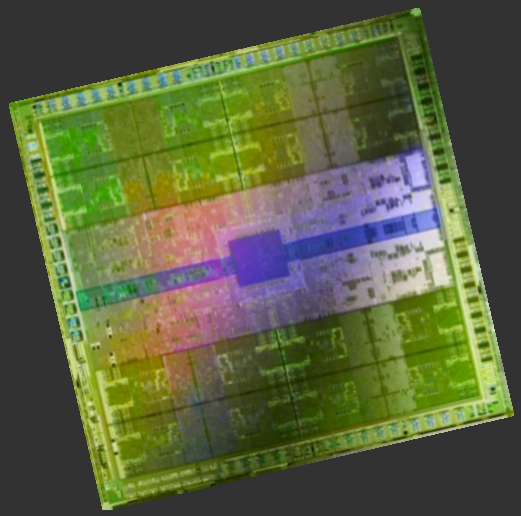
# Real-time graphics systems

## Easy to use

[OpenGL, Direct3D]

## Efficient, parallel, heterogeneous

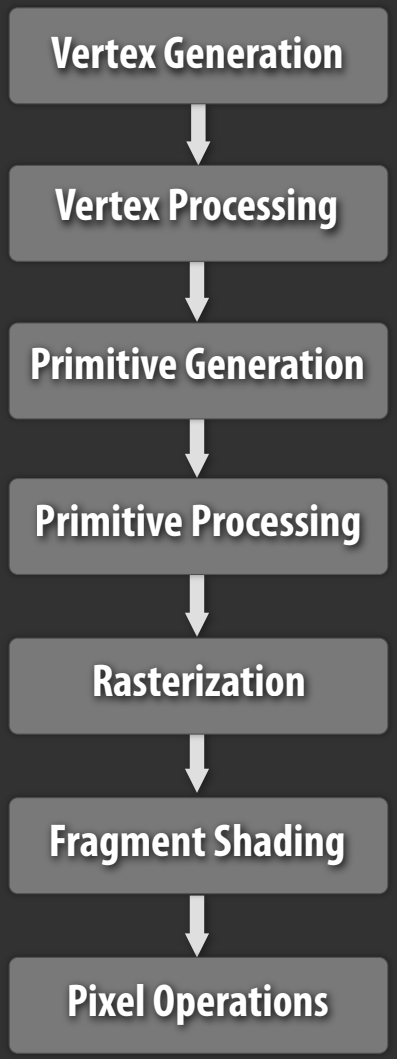
[GPUs]



NVIDIA Fermi GPU:

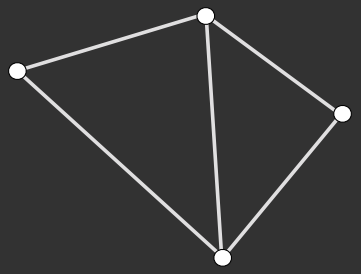
# Graphics: simple programming abstractions

## Real-time graphics pipeline
## OpenGL [Akeley 92], Direct3D [Blythe 05]

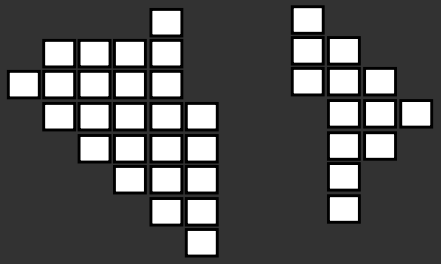




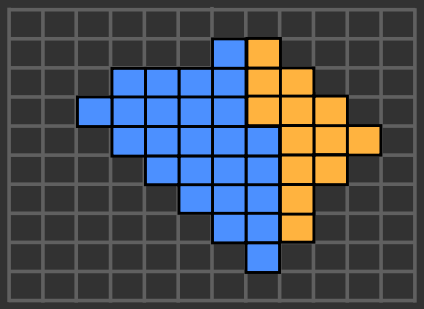
Vertices



Primitives (triangles)

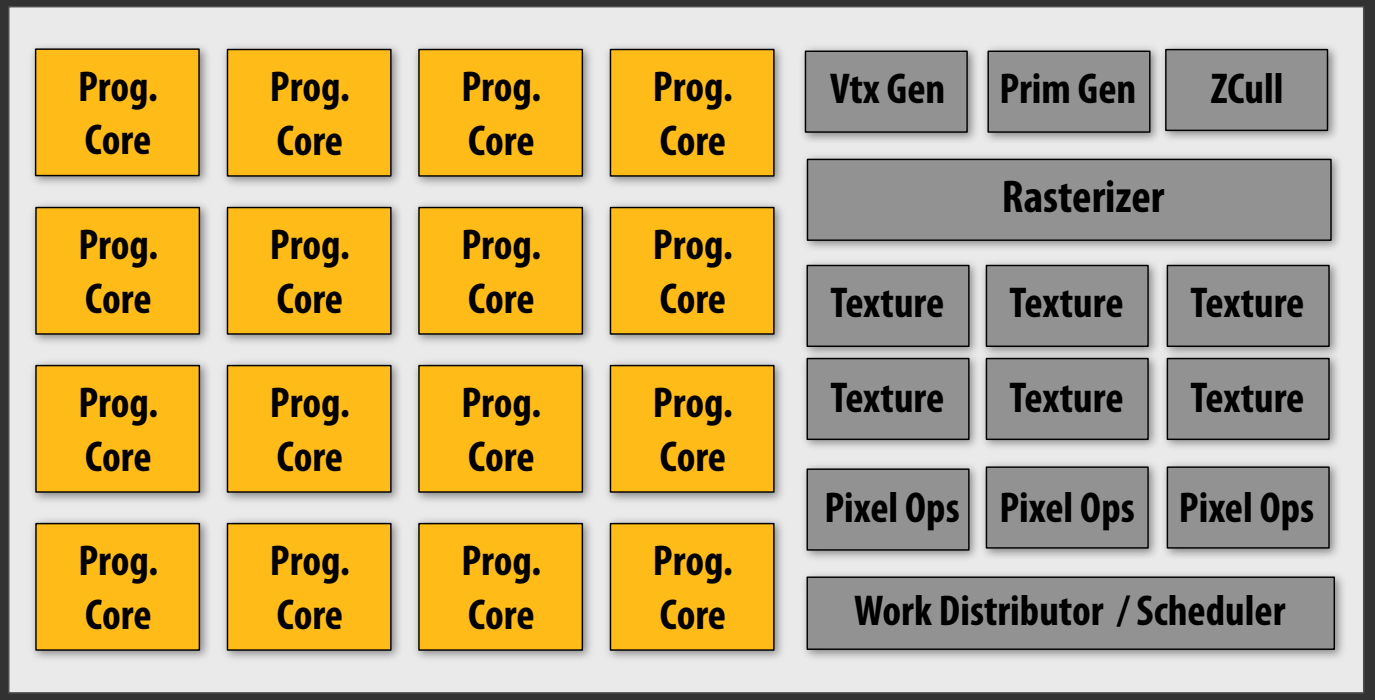


Fragments



Pixels

# Heterogeneous, multi-core GPU



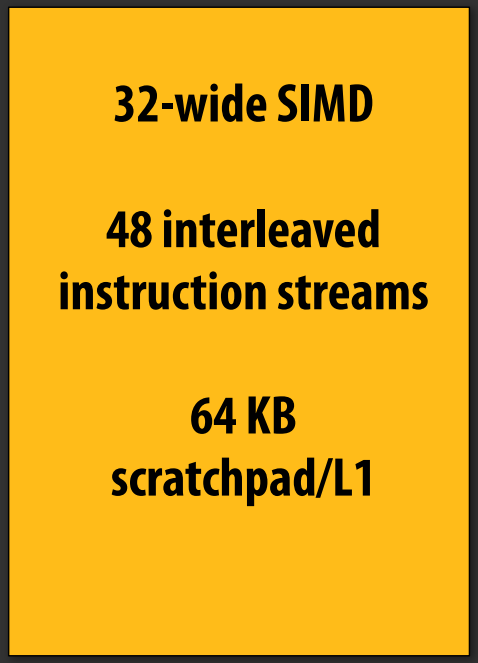

NVIDIA Fermi GPU

16 programmable cores: ~ 1.5 TFLOPS

(15X more flops than quad-core Intel CPU)

# GPU programmable core

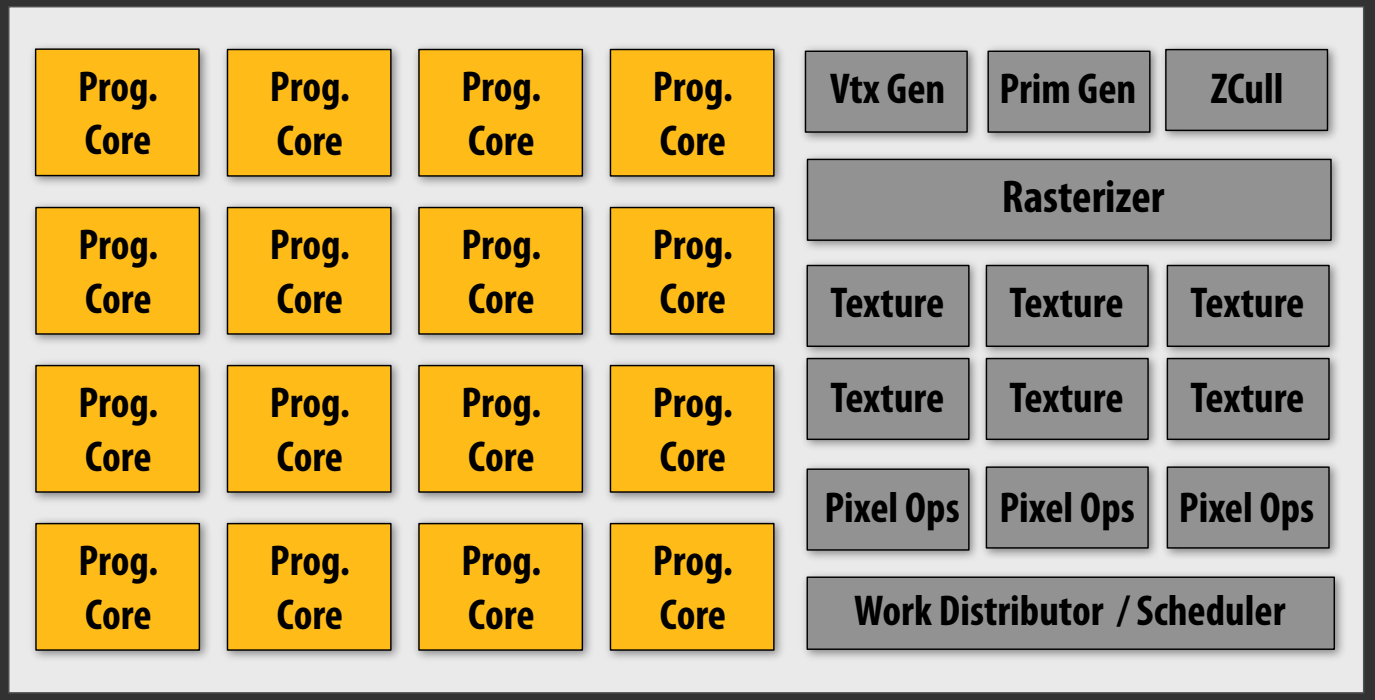
NVIDIA Fermi Core




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

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

# Heterogeneous, multi-core GPU




NVIDIA Fermi GPU

16 programmable cores: ~ 1.5 TFLOPS

+ fixed-function processing specific to graphics

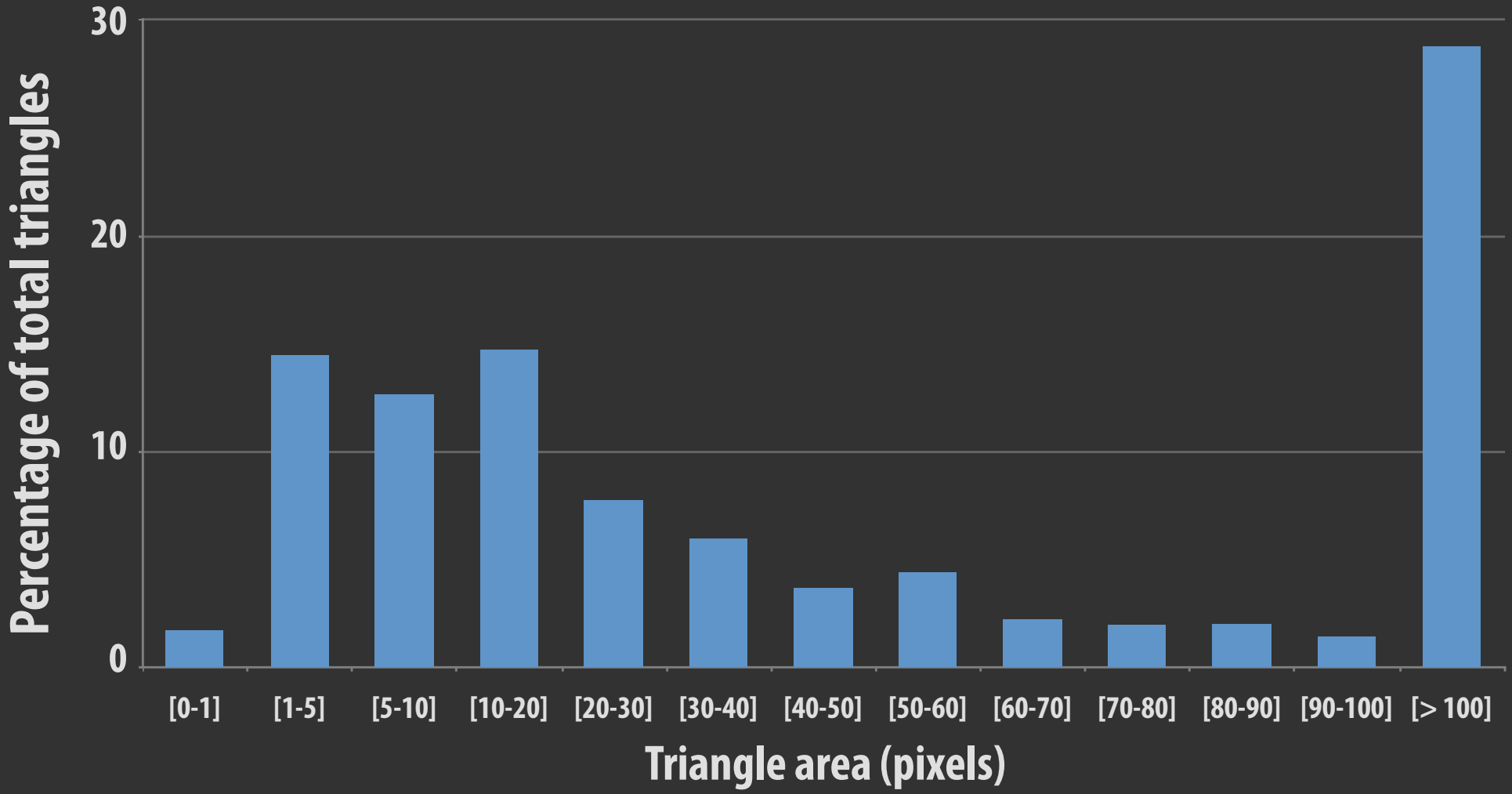
$500

# Interactive graphics: low geometric detail

Credit: "UP" PS3 game (Heavy Iron/Disney)


Credit: Pro Evolution Soccer 2010 (Konami)

# Interactive graphics uses large triangles

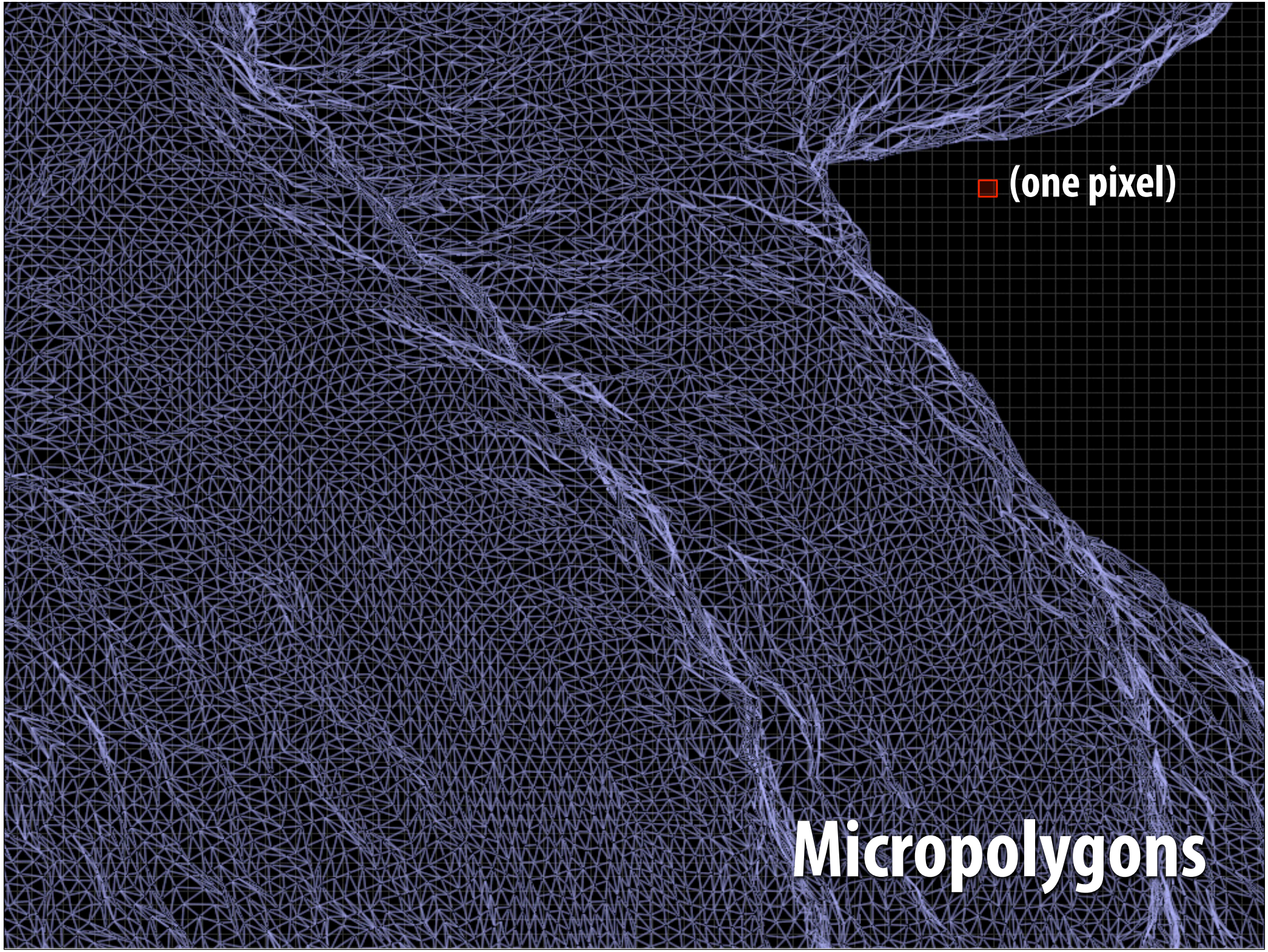



[source NVIDIA]

# Highly detailed surfaces

# Highly detailed surfaces

Credit: Pixar Animation Studios, UP (2009)


(one pixel)
Micropolygons

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

# Sources of inefficiency

**Tessellation**

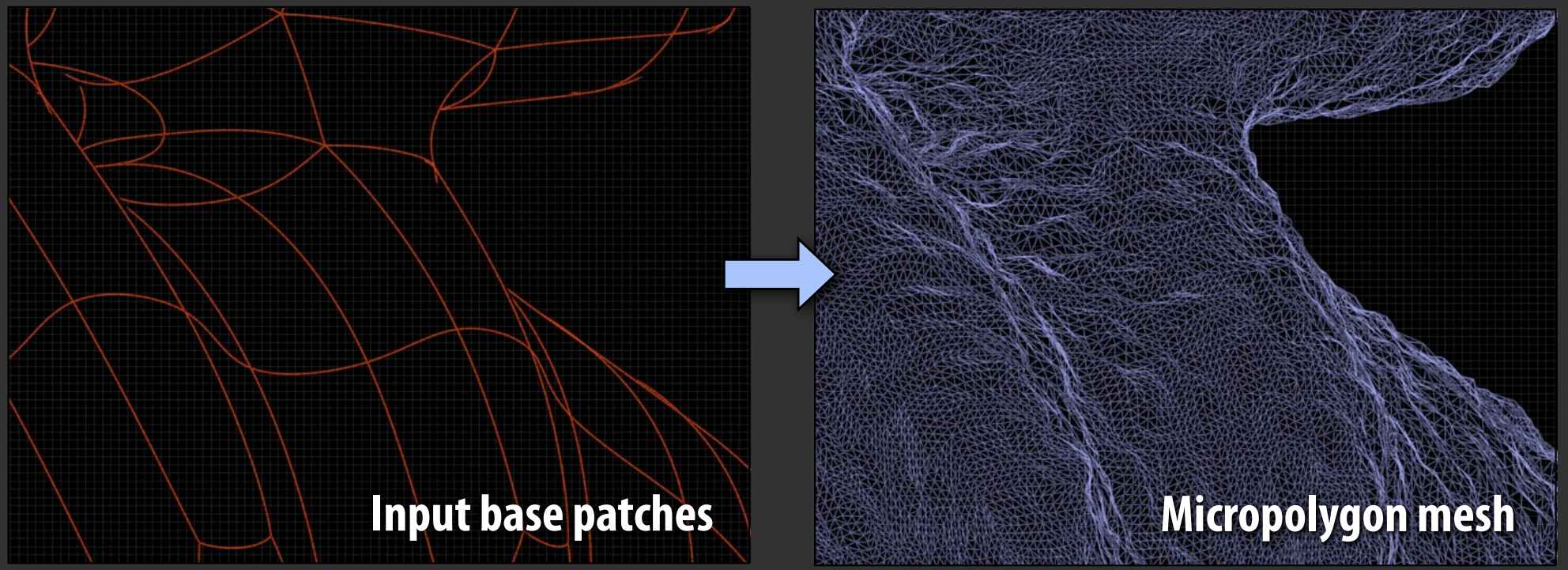
**(generating geometry)**

**Rasterization**

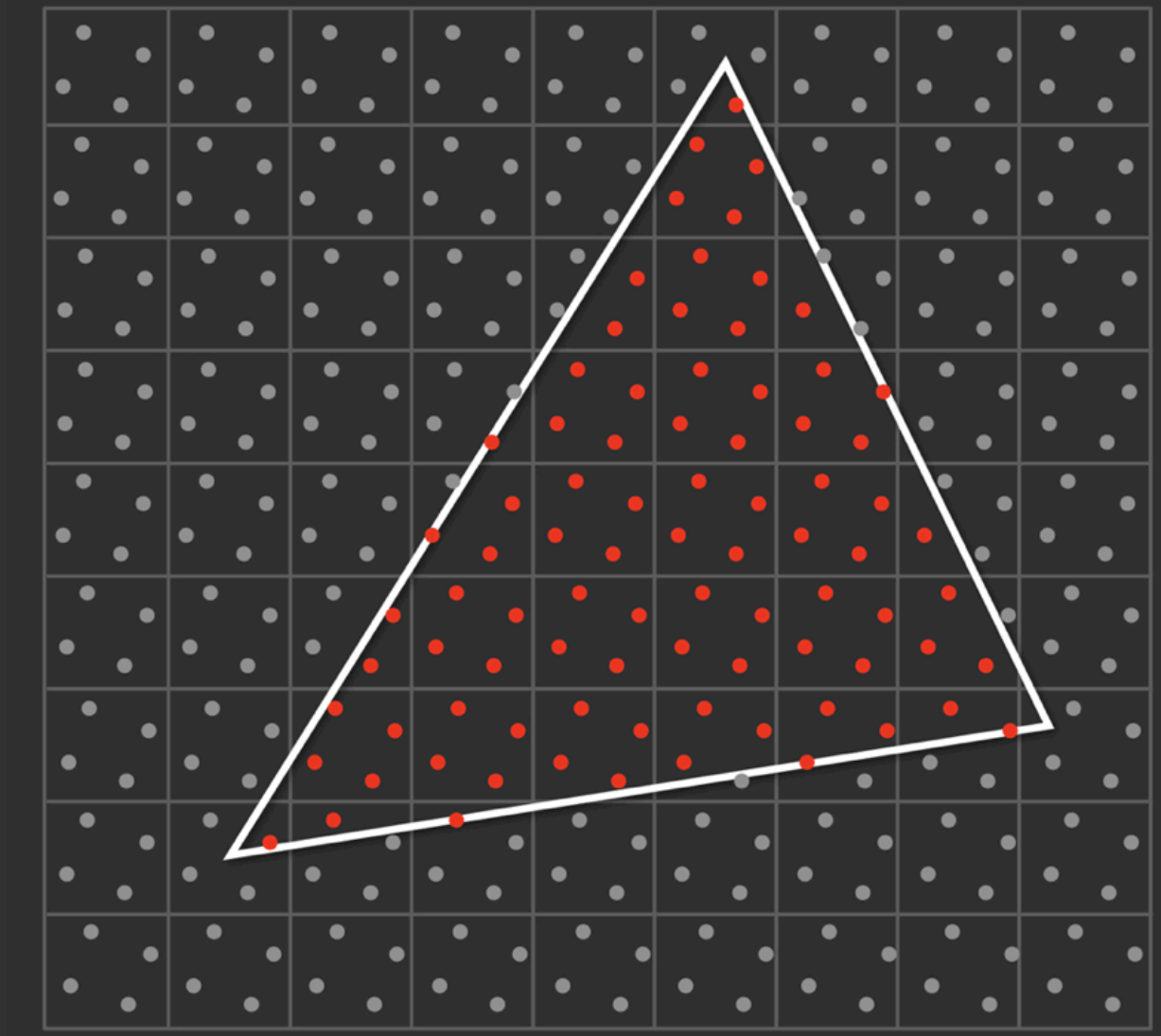
**Shading**

# Missing: adaptive tessellation

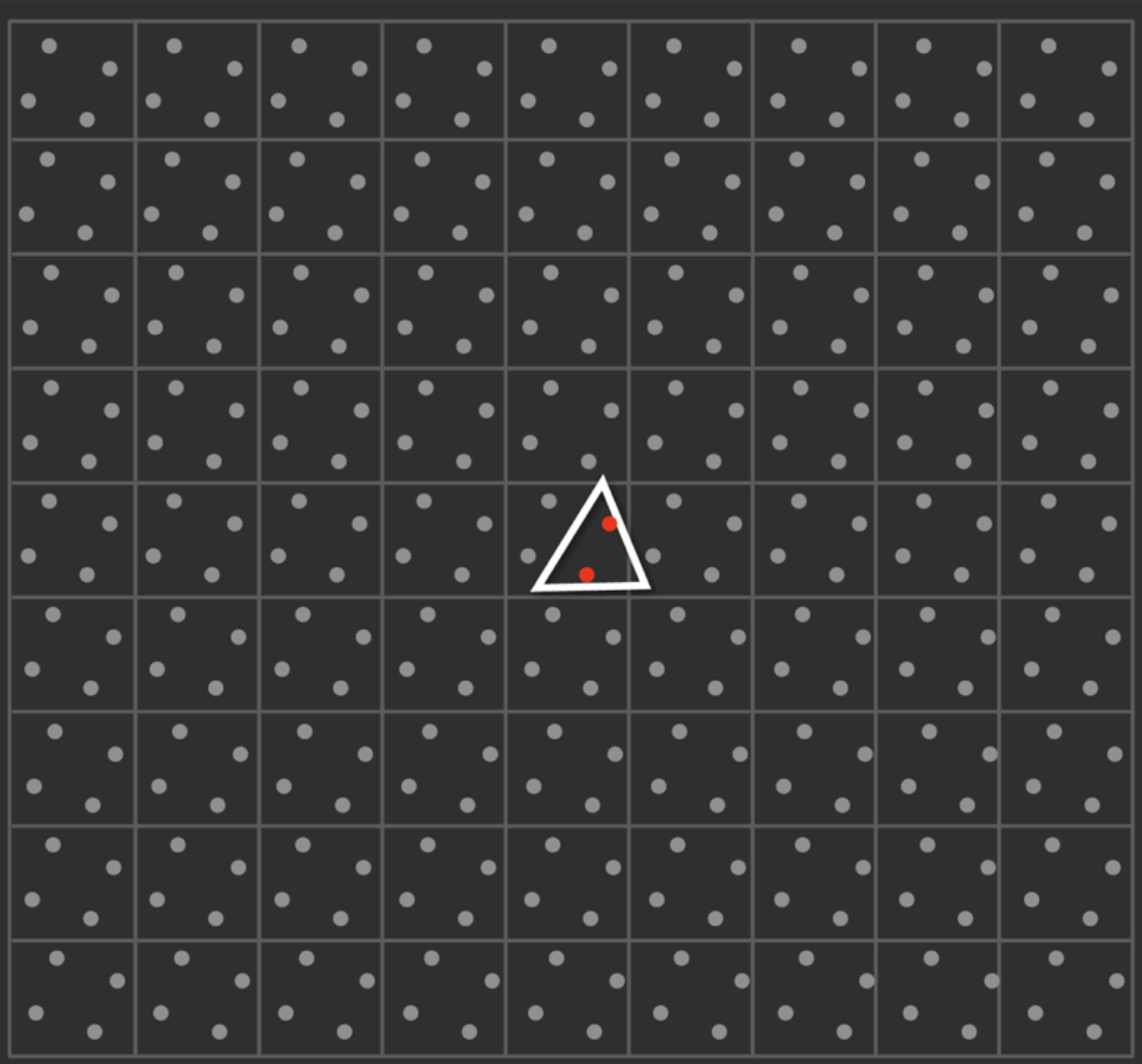
## Generate triangles on-demand in the pipeline



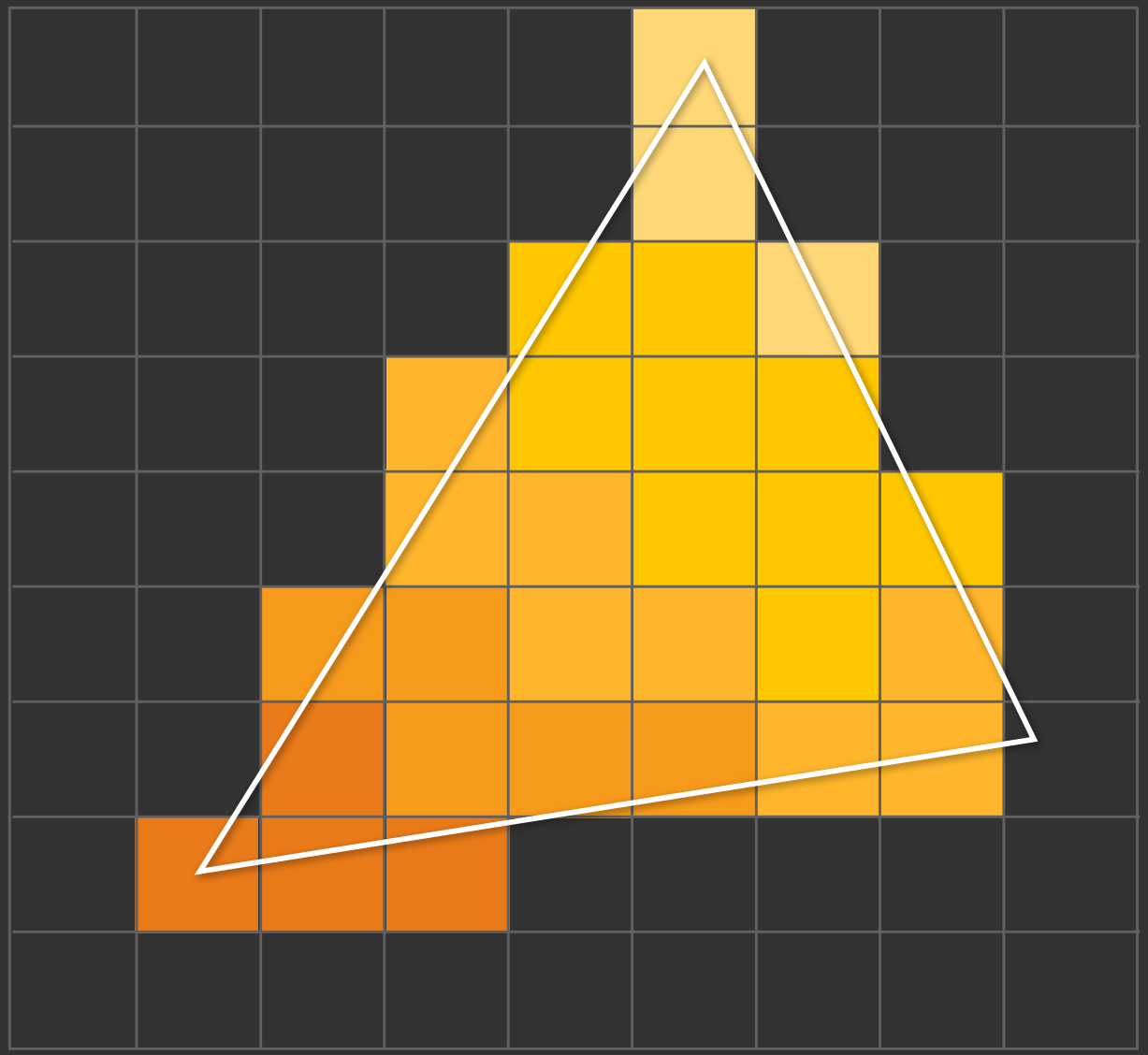
# Rasterization: computing covered pixels



# 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 8× more shading work than needed.**

# Goal: influence design of future GPUs

- Non-goal: use current GPUs to accelerate implementation of advanced rendering pipelines

  [RenderAnts] [Loop/Eisenacher 09]
  [Gelato] [Patney 08] [many, many others]

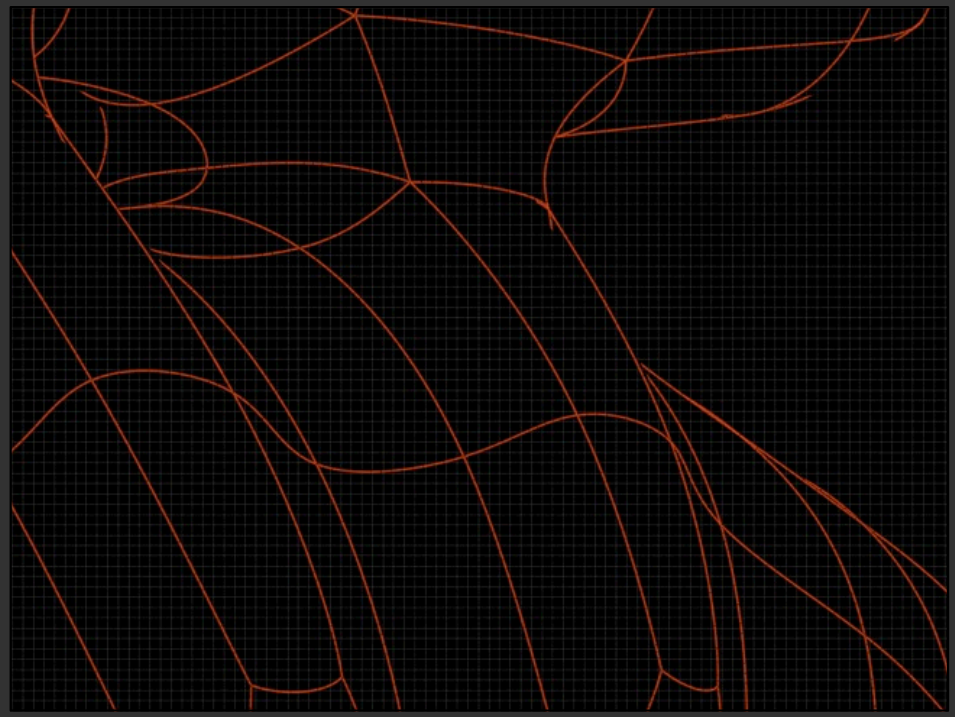
# TESSELLATION:

## 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)

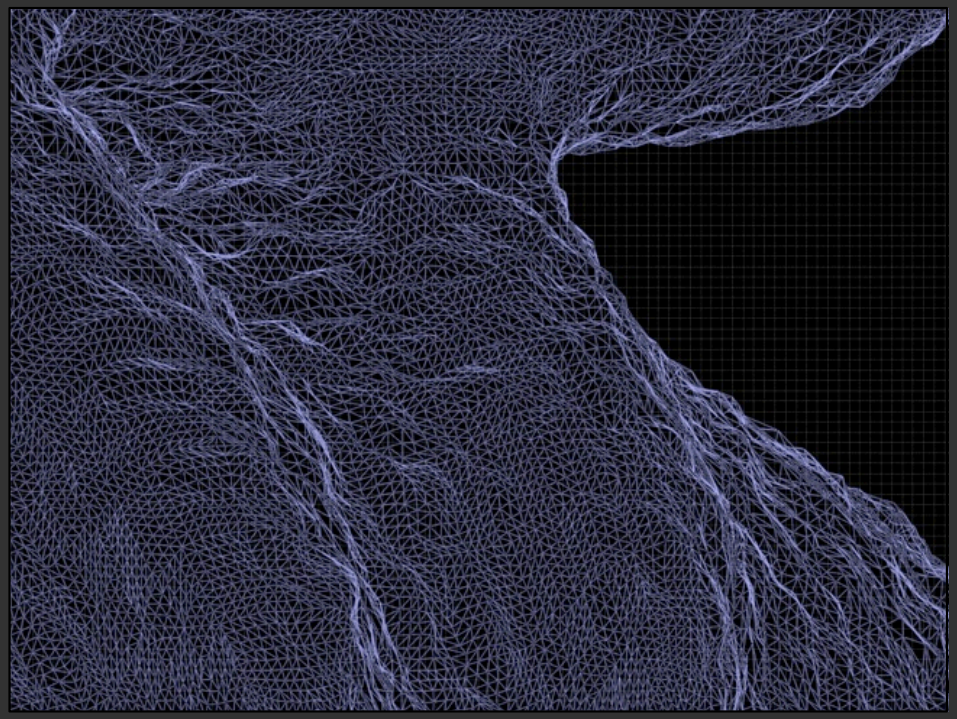
# Tessellation input: parametric patches



Input base patches
(example: bicubic patch)

[Vlachos 01, Loop 08, Loop 09]

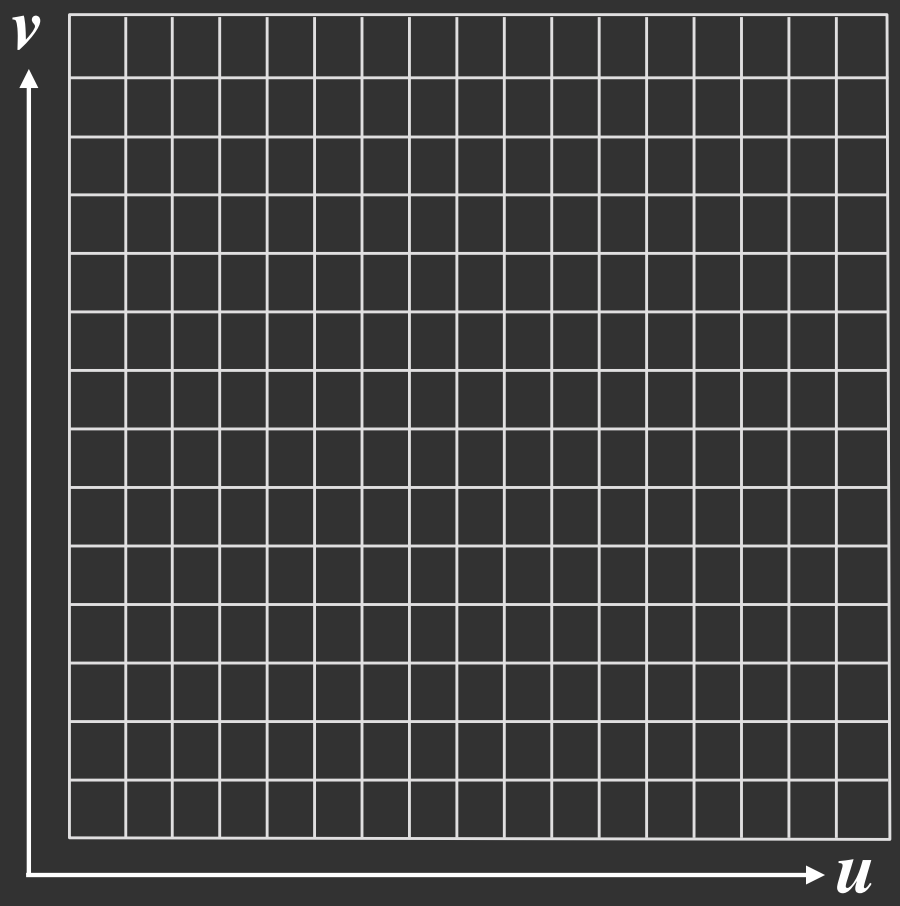
# Tessellation output: micropolygon mesh



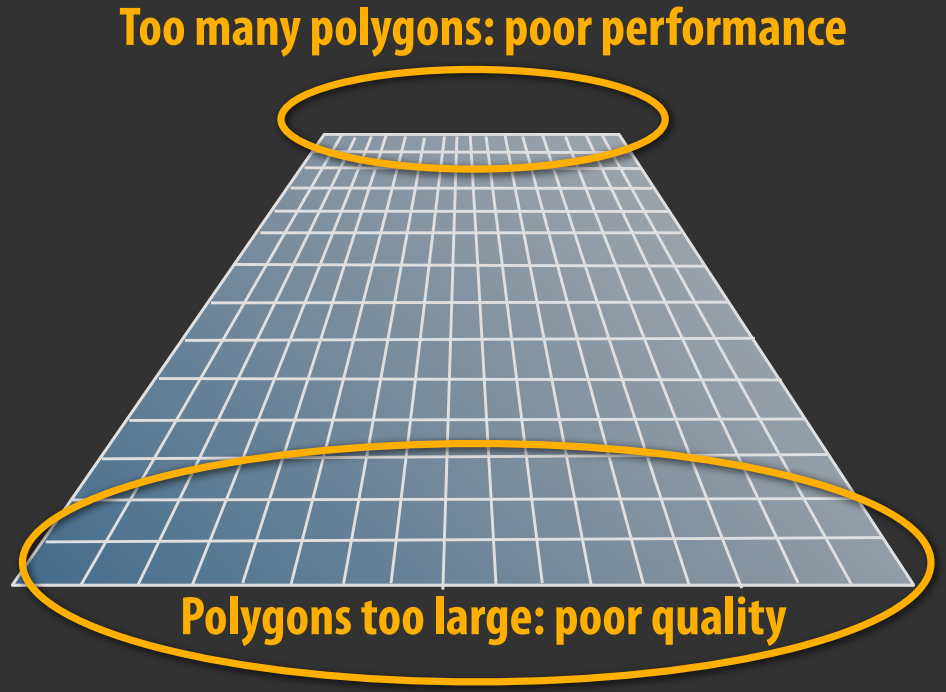
**Goal: all triangles are approximately 1/2 pixel in area**

**(yields about one vertex per pixel)**

# Uniform patch tessellation is insufficient




Uniform partitioning of patch (parametric domain)



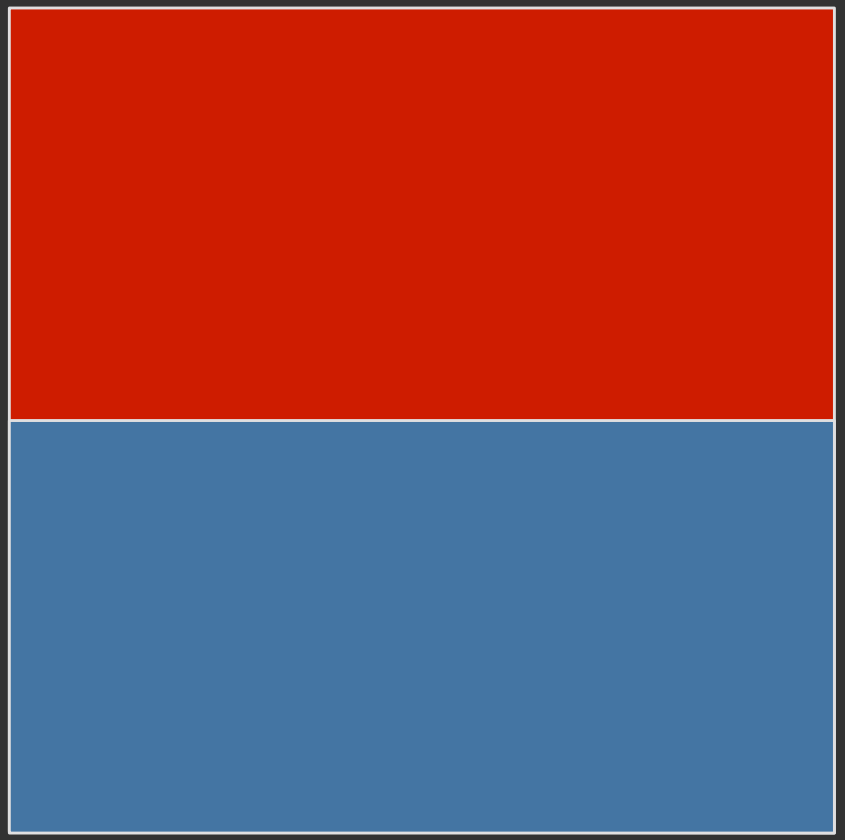

Patch viewed from camera

## Adaptive tessellation:

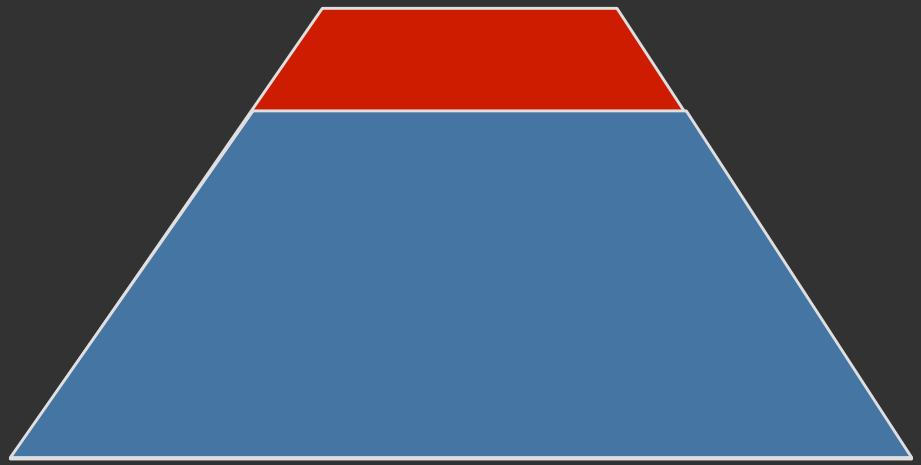
# Lane-Carpenter patch algorithm

[Lane 80]

# Adaptive tessellation

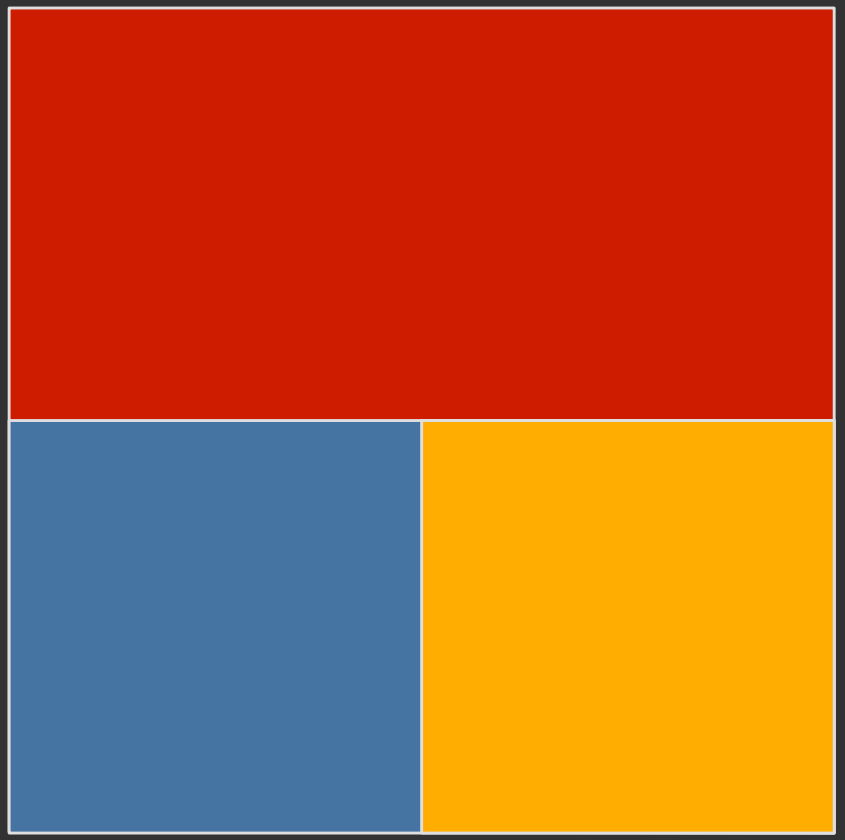


Patch parametric domain

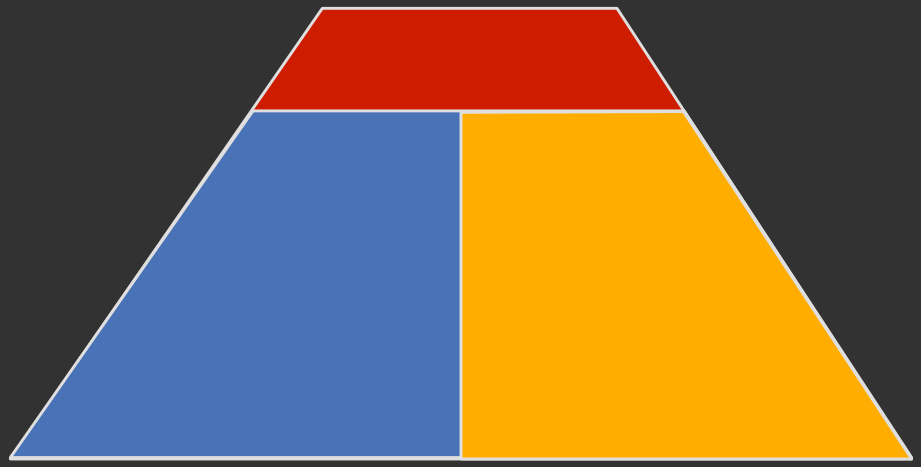


Patch viewed from camera

# Adaptive tessellation

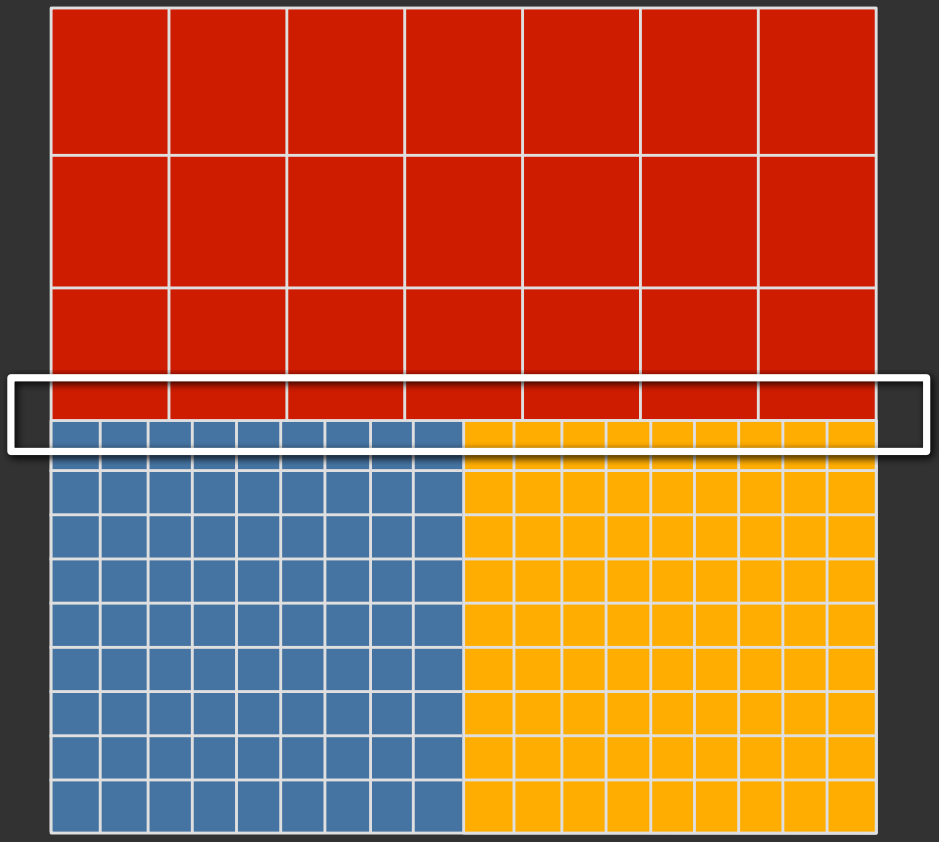


Patch parametric domain

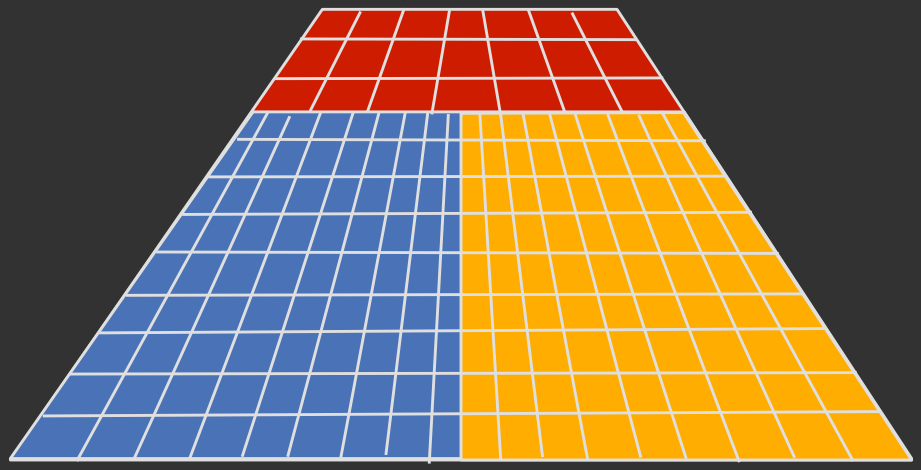


Patch viewed from camera

# Adaptive tessellation

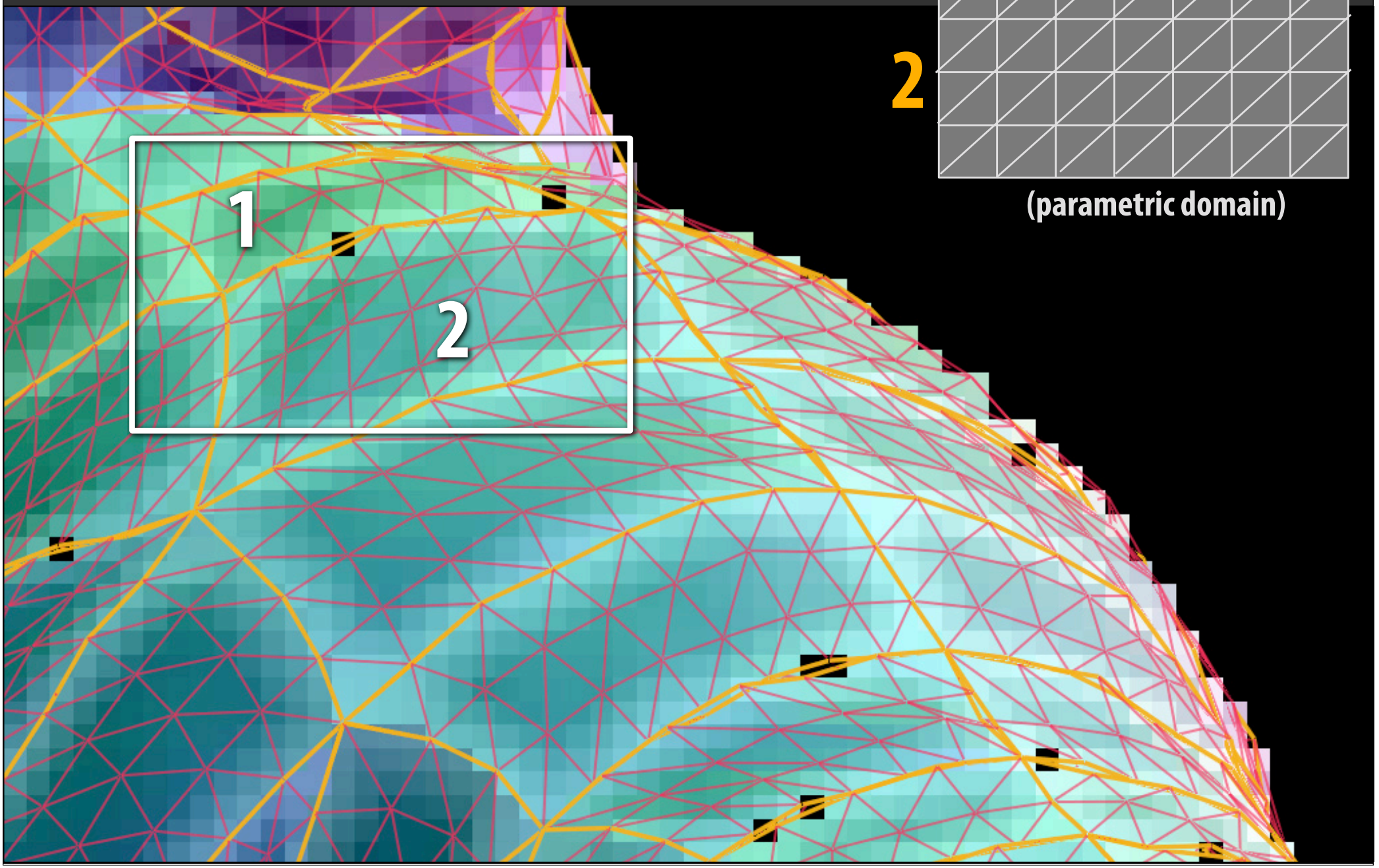


Patch parametric domain



Patch viewed from camera

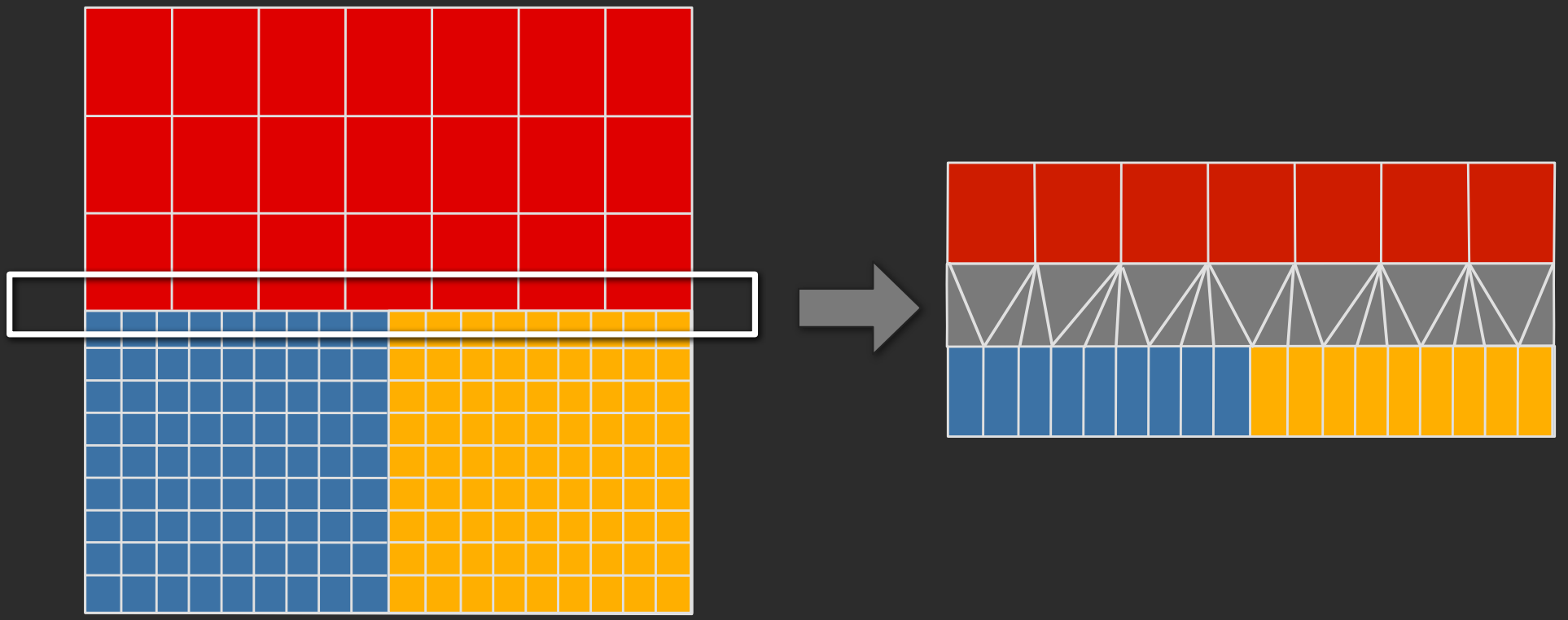
# Cracks!



# Off-line status quo: "stitching" fixes cracks

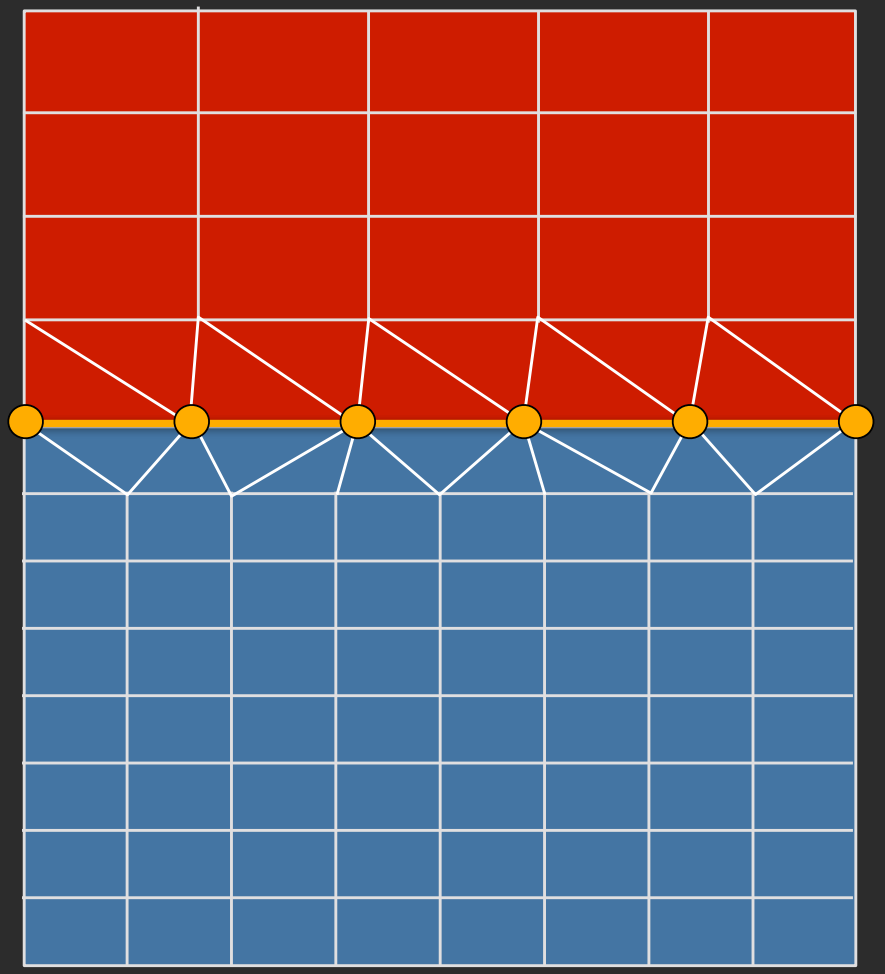
Use a strip of polygons to connect adjacent sub-patches

Creates dependency: cannot process sub-patches in parallel



# Parallel crack fixing

T(edge) = 5

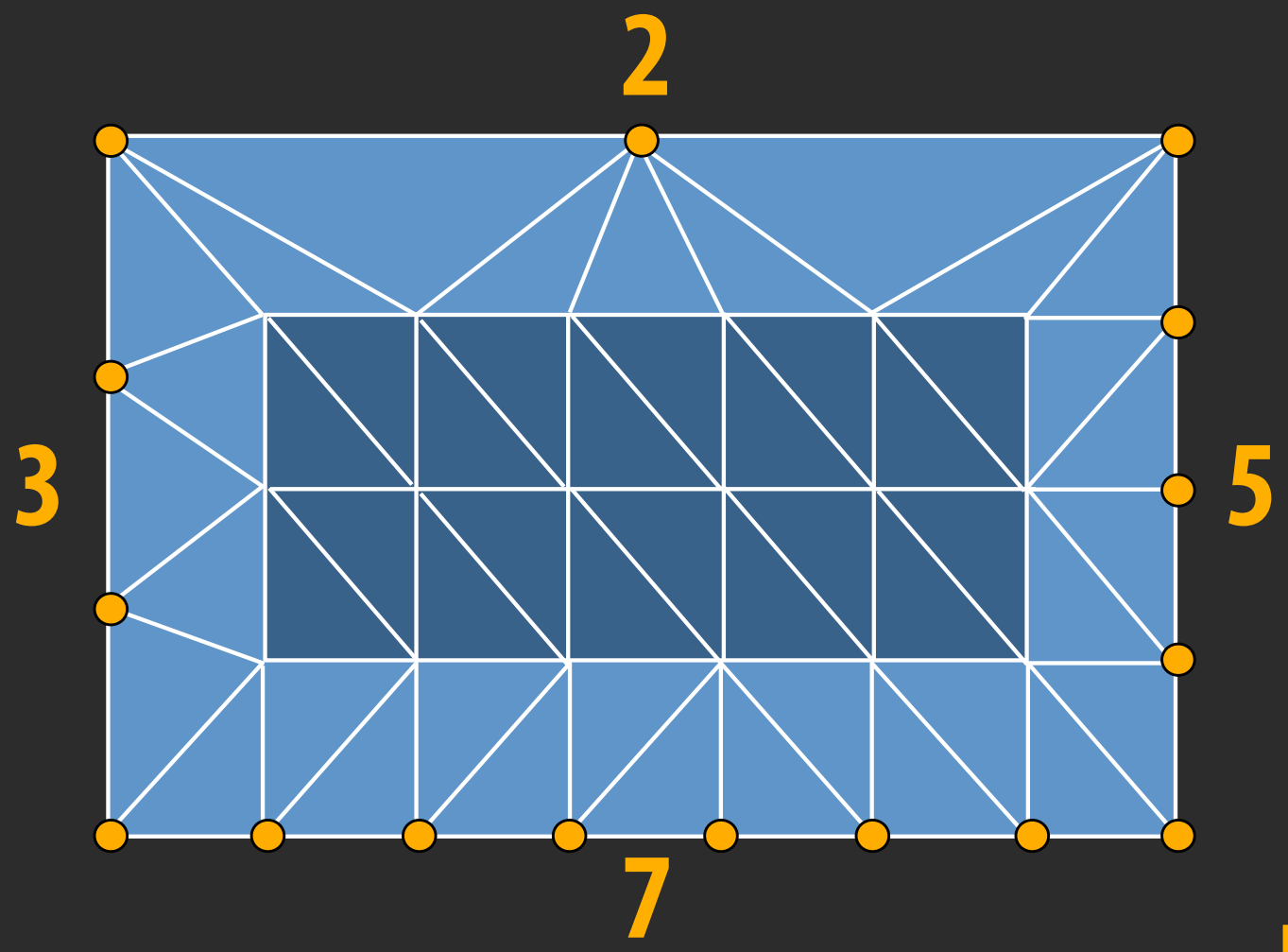


Adjacent regions agree on tessellation along edge
(in this case: 5 segments)

# Crack-free, uniform tessellation

**Input: edge tessellation constraints for a patch**

**Output: (almost) uniform mesh that meets these constraints**

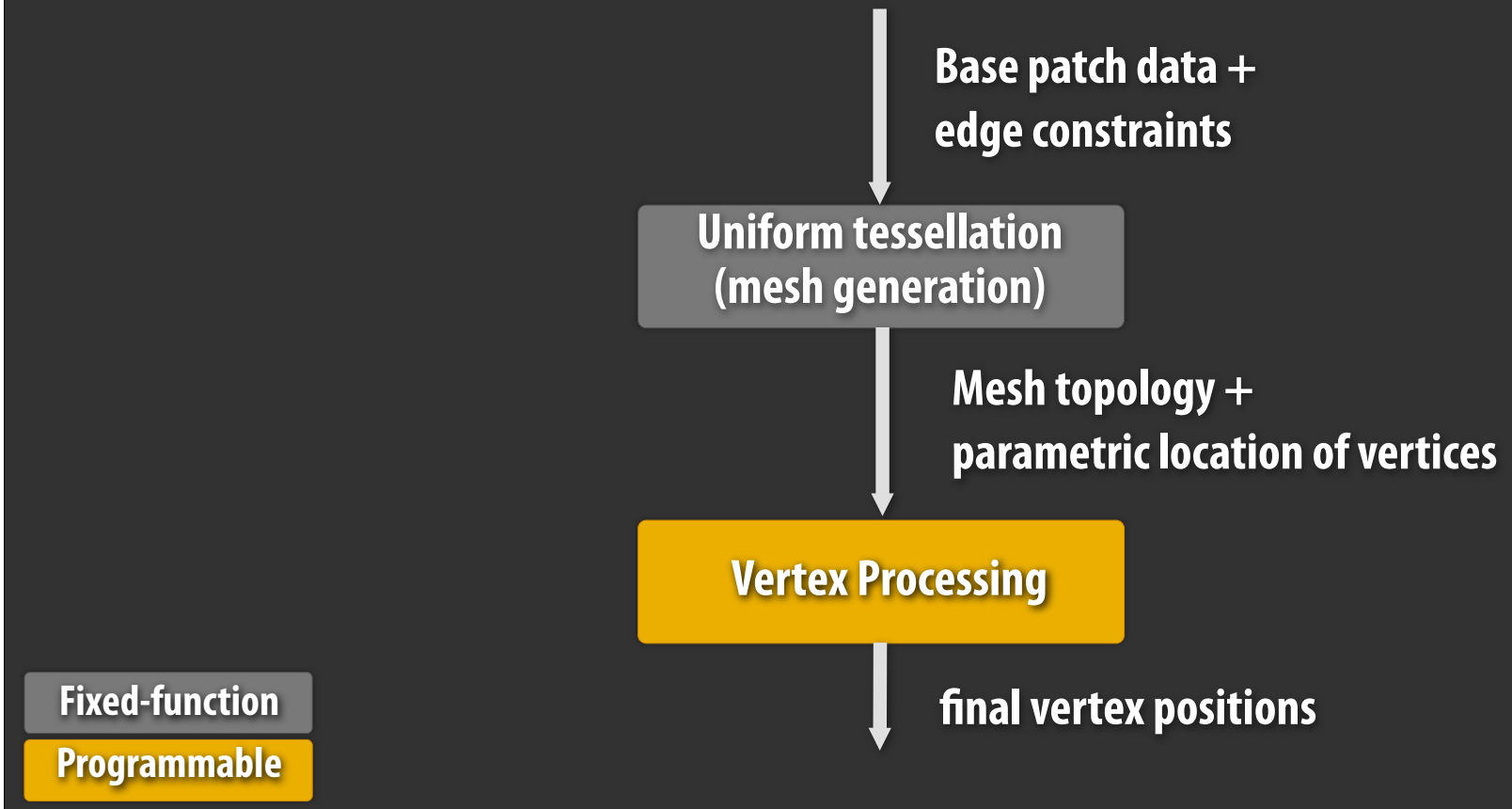



[Moreton 01]

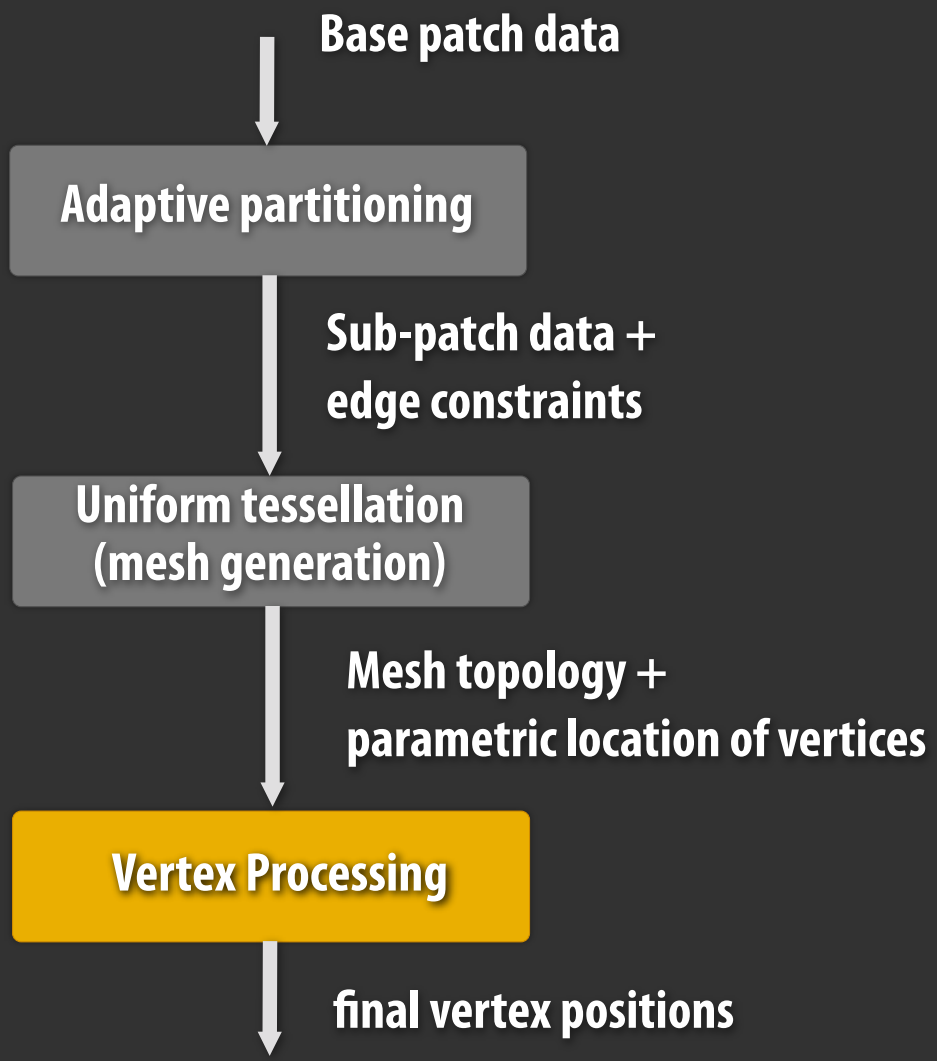
# GPU tessellation

[Direct3D 11]

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



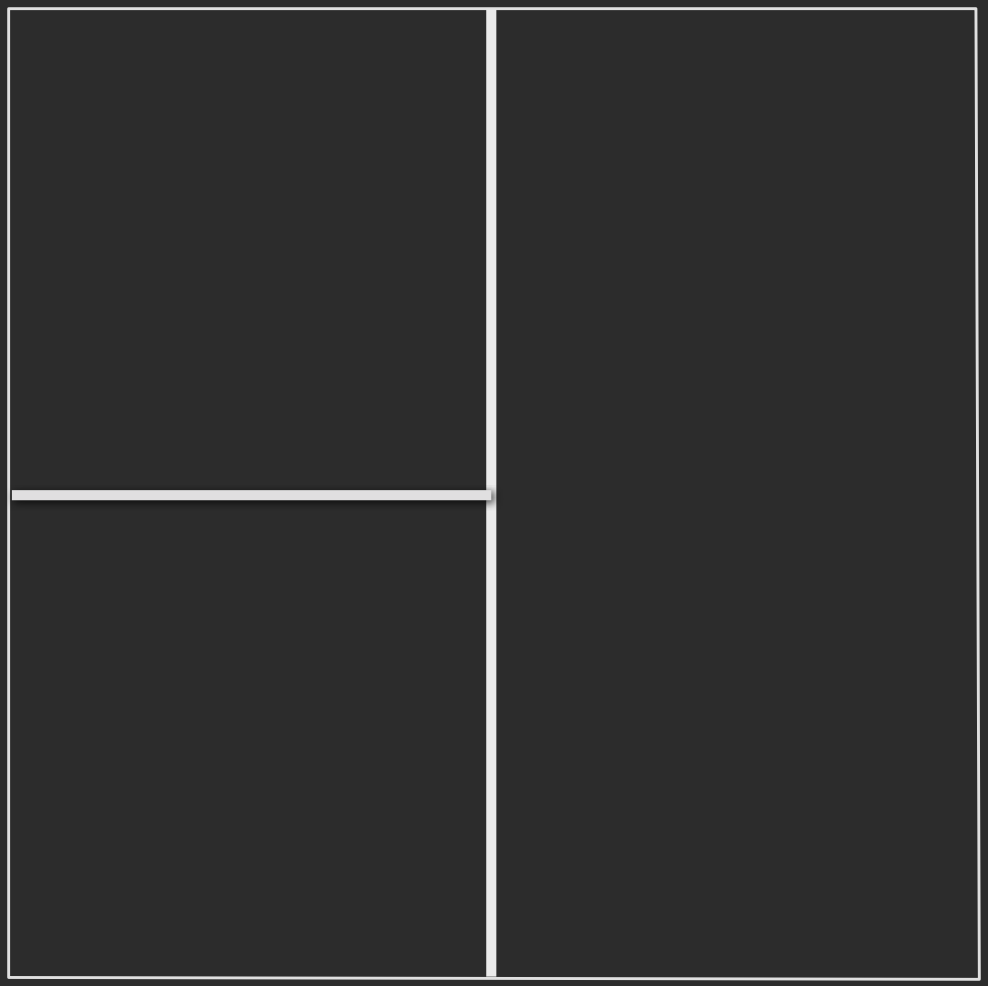
# Want: adaptive tessellation pipeline



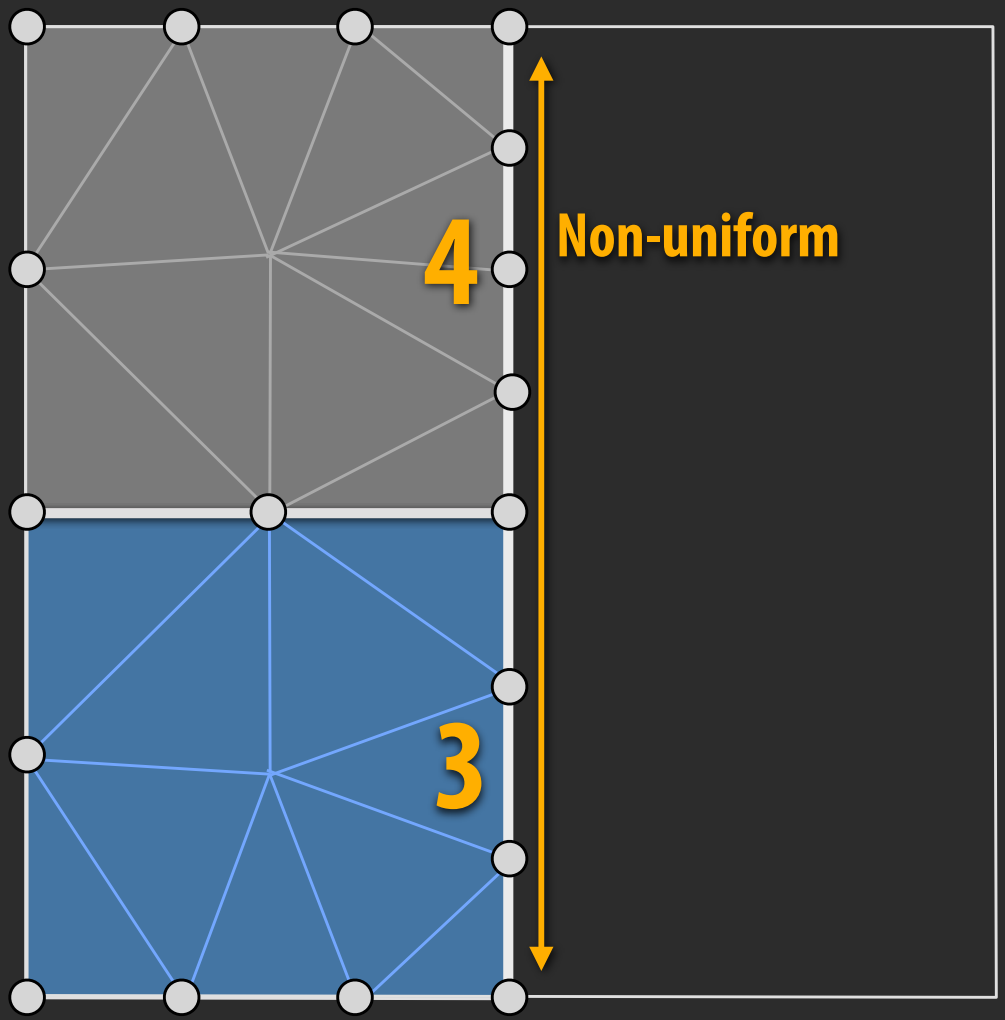

Fixed-function

Programmable

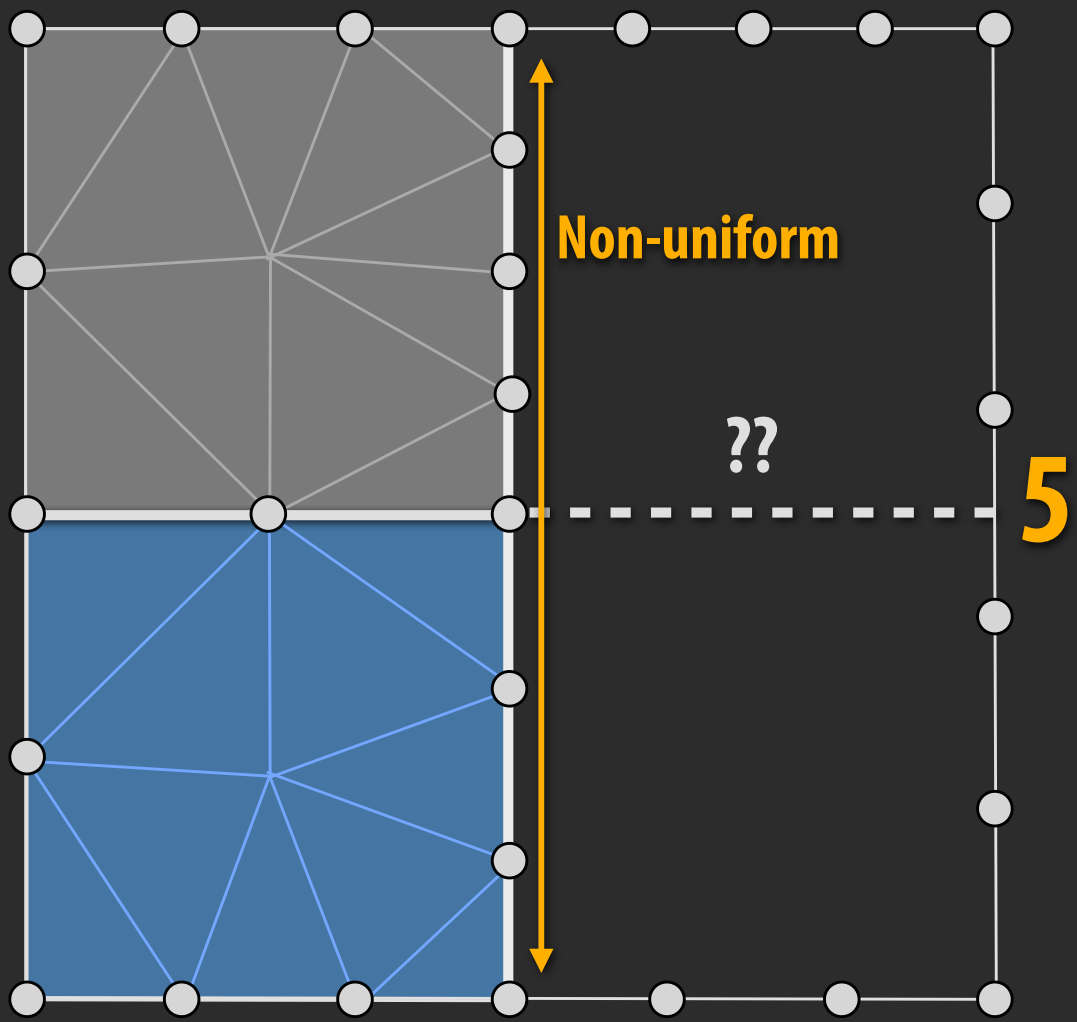
# Making Lane-Carpenter match edges



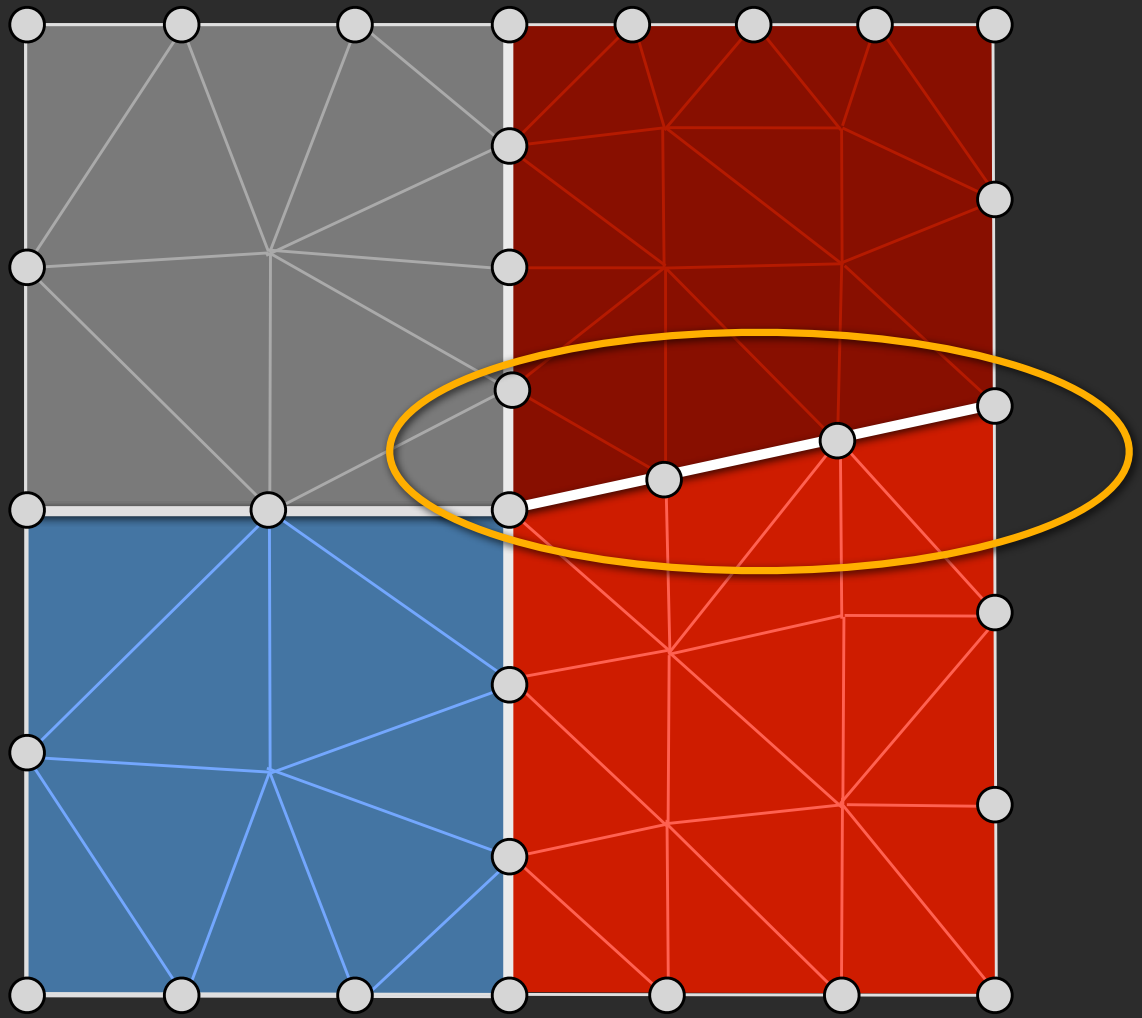
# Making Lane-Carpenter match edges



# Making Lane-Carpenter match edges



# Non-isoparametric splits



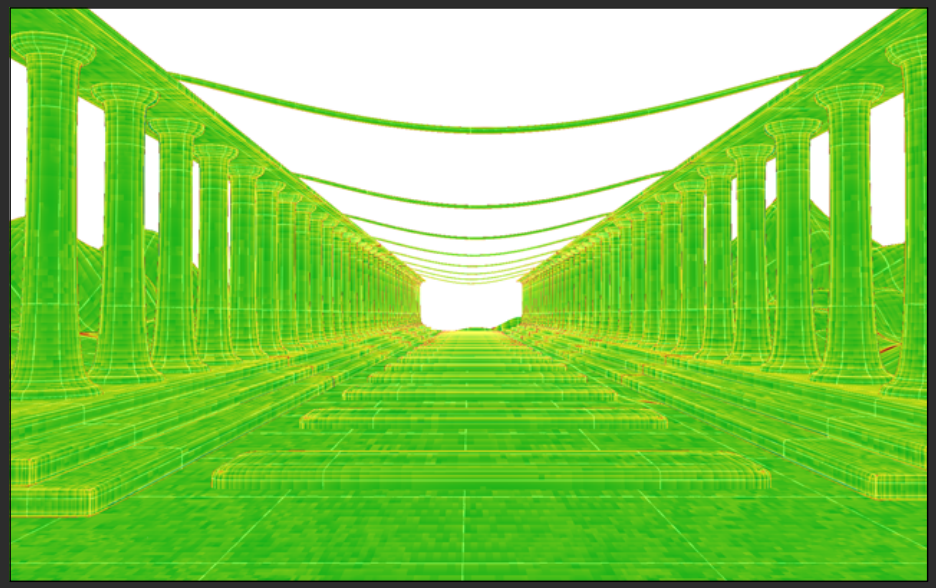
DiagSplit: adaptive, crack-free, sub-patch parallel

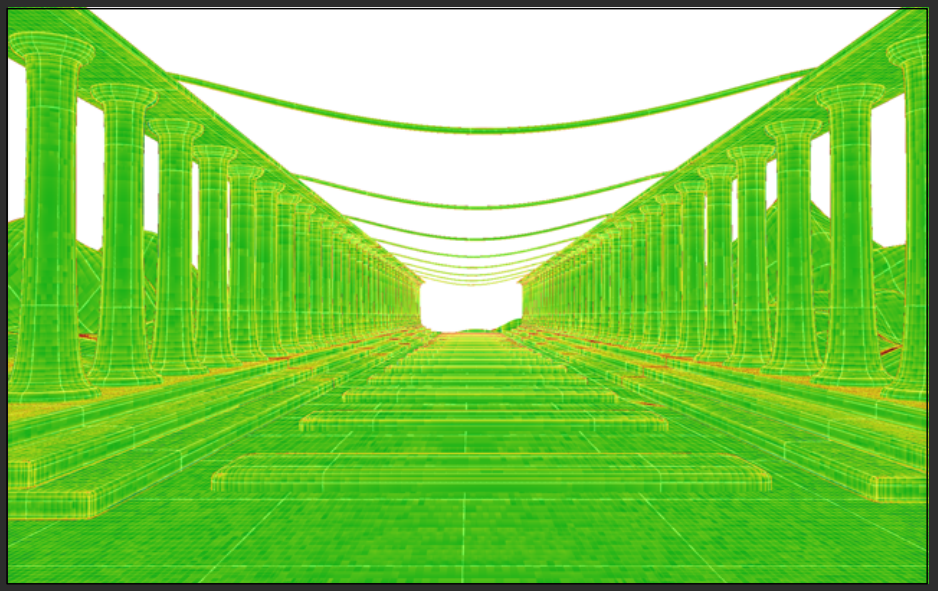
# DiagSplit adapts as well as Lane-Carpenter

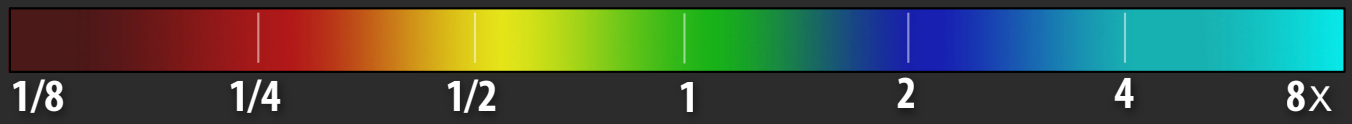
7% more vertices

Lane-Carpenter

DiagSplit







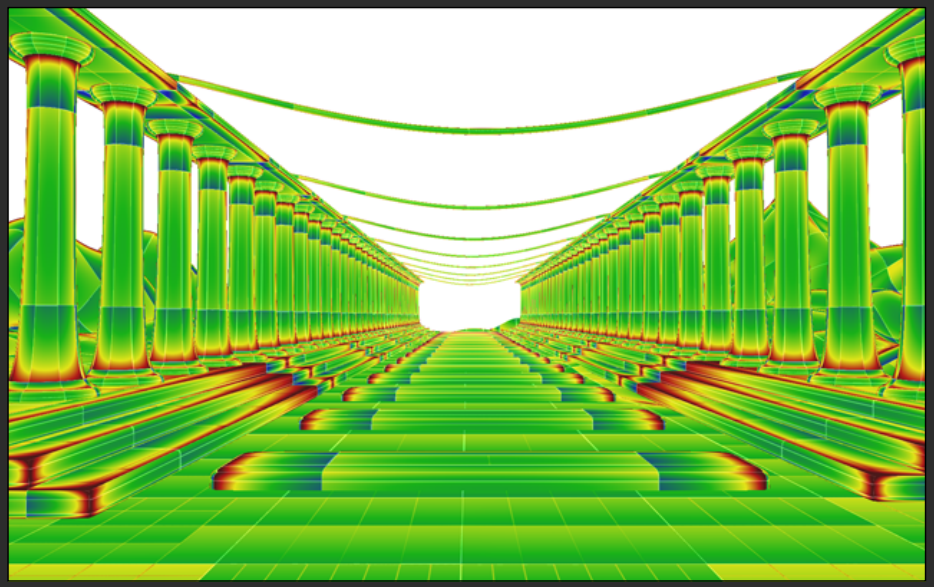

Triangle area relative to target (1/2 pixel triangles)

[Fisher 09]

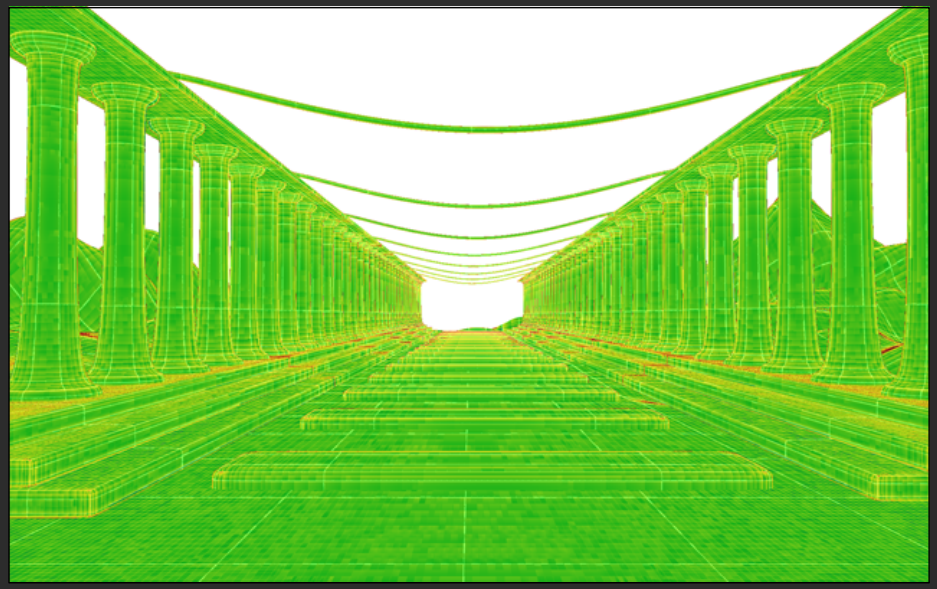
# DiagSplit: produces better meshes using fewer vertices

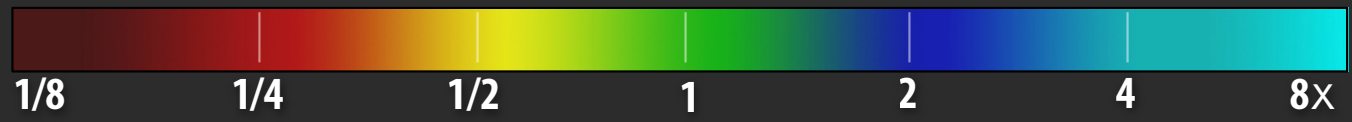
40% fewer vertices

Direct3D 11 Uniform



DiagSplit






Triangle area relative to target (1/2 pixel triangles)

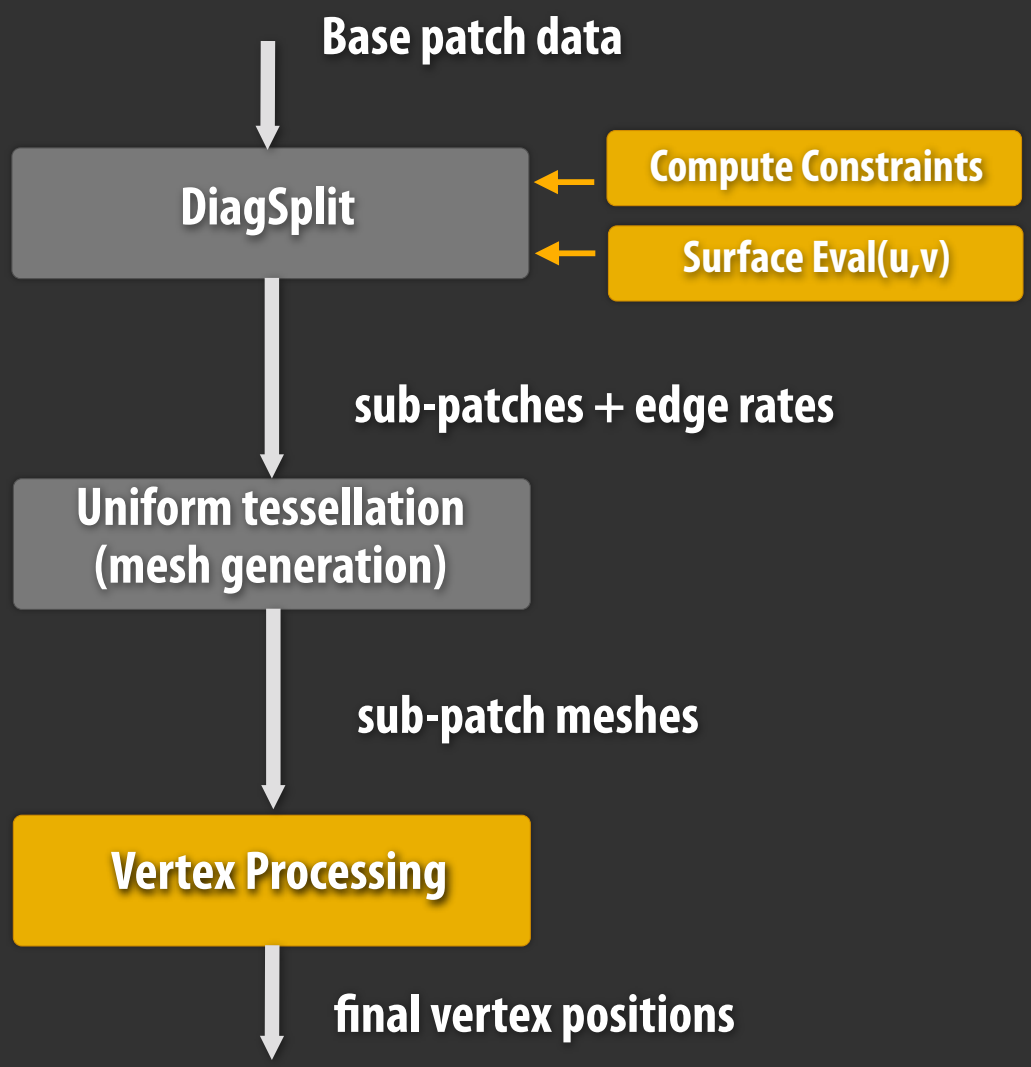
[Fisher 09]

# DiagSplit tessellation pipeline

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

Irregular (data-amplification)
Fixed-function implementations exist

data-parallel, application programmable



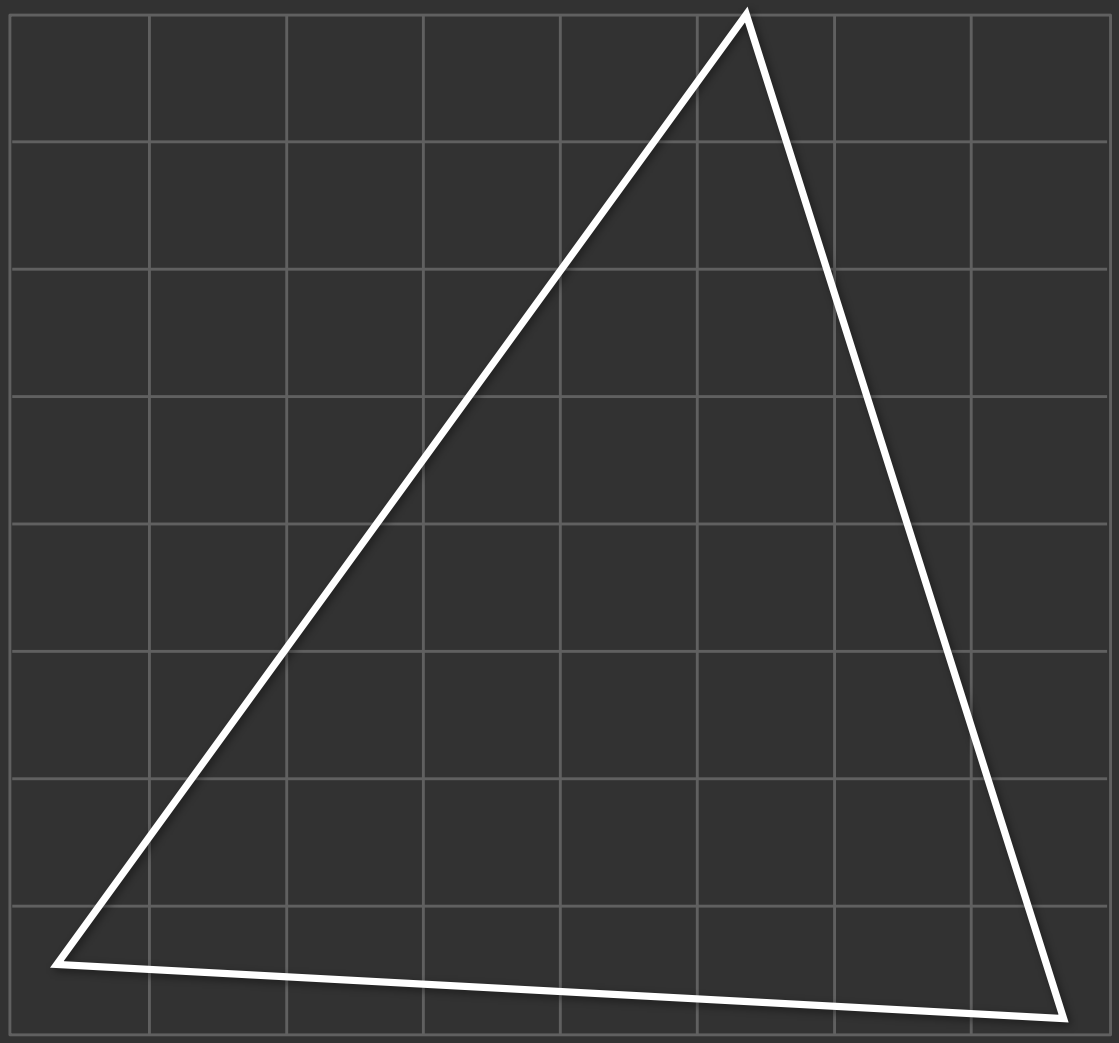

Fixed-function
Programmable

# Recap

- DiagSplit: new algorithm designed to fit 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

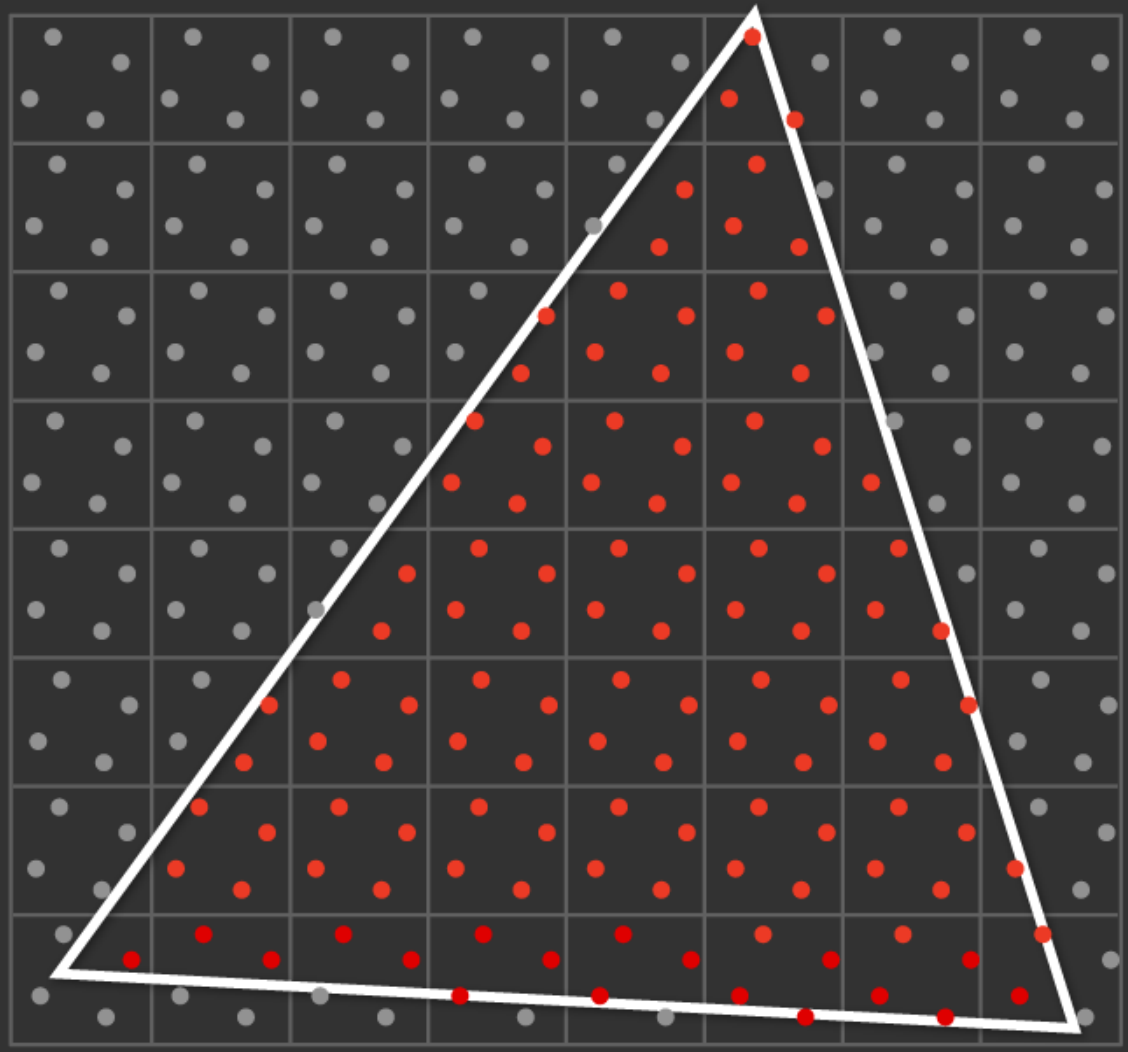
# RASTERIZATION

# Rasterization



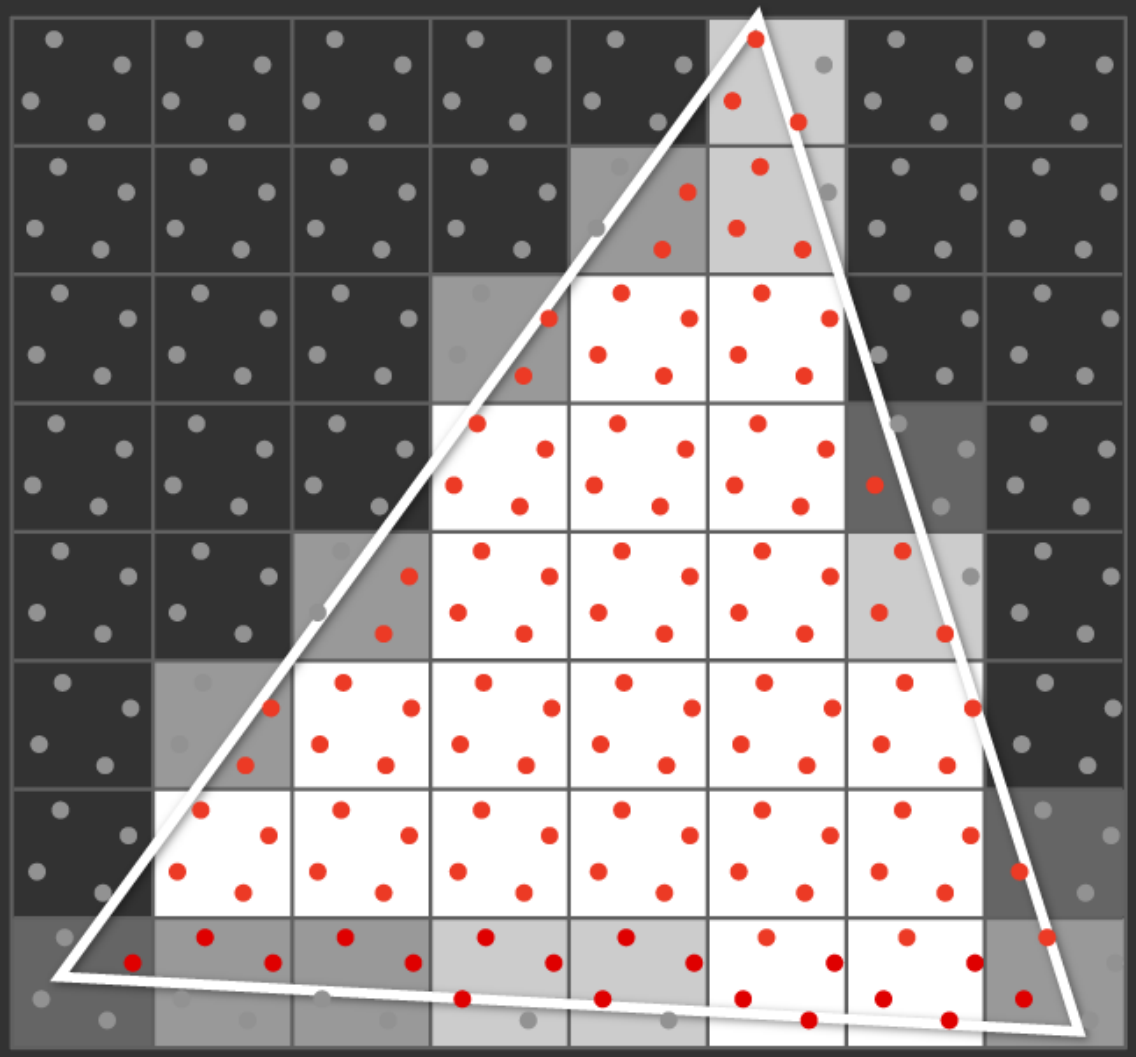
# Rasterization

## Compute coverage using point-in-triangle tests

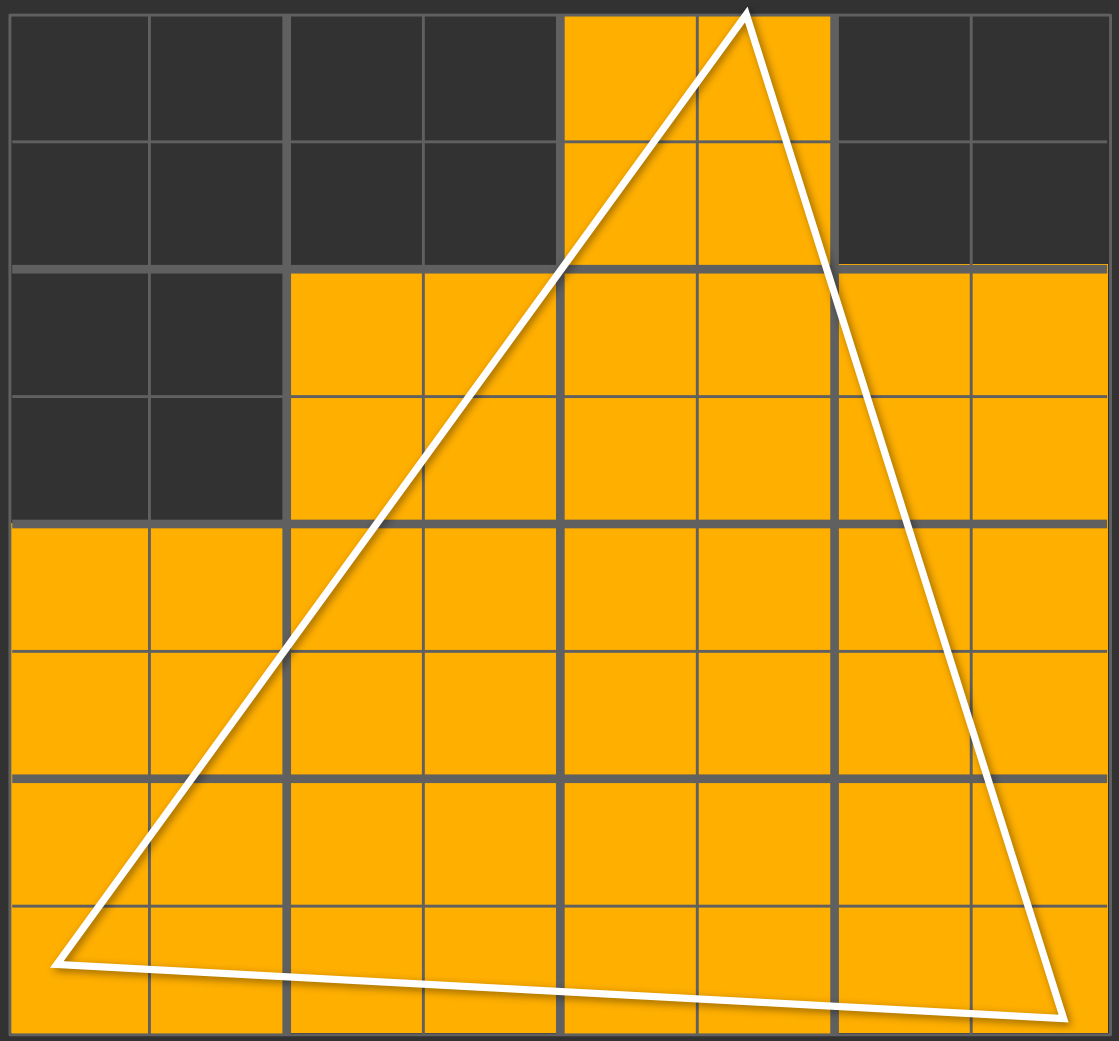


# Rasterization

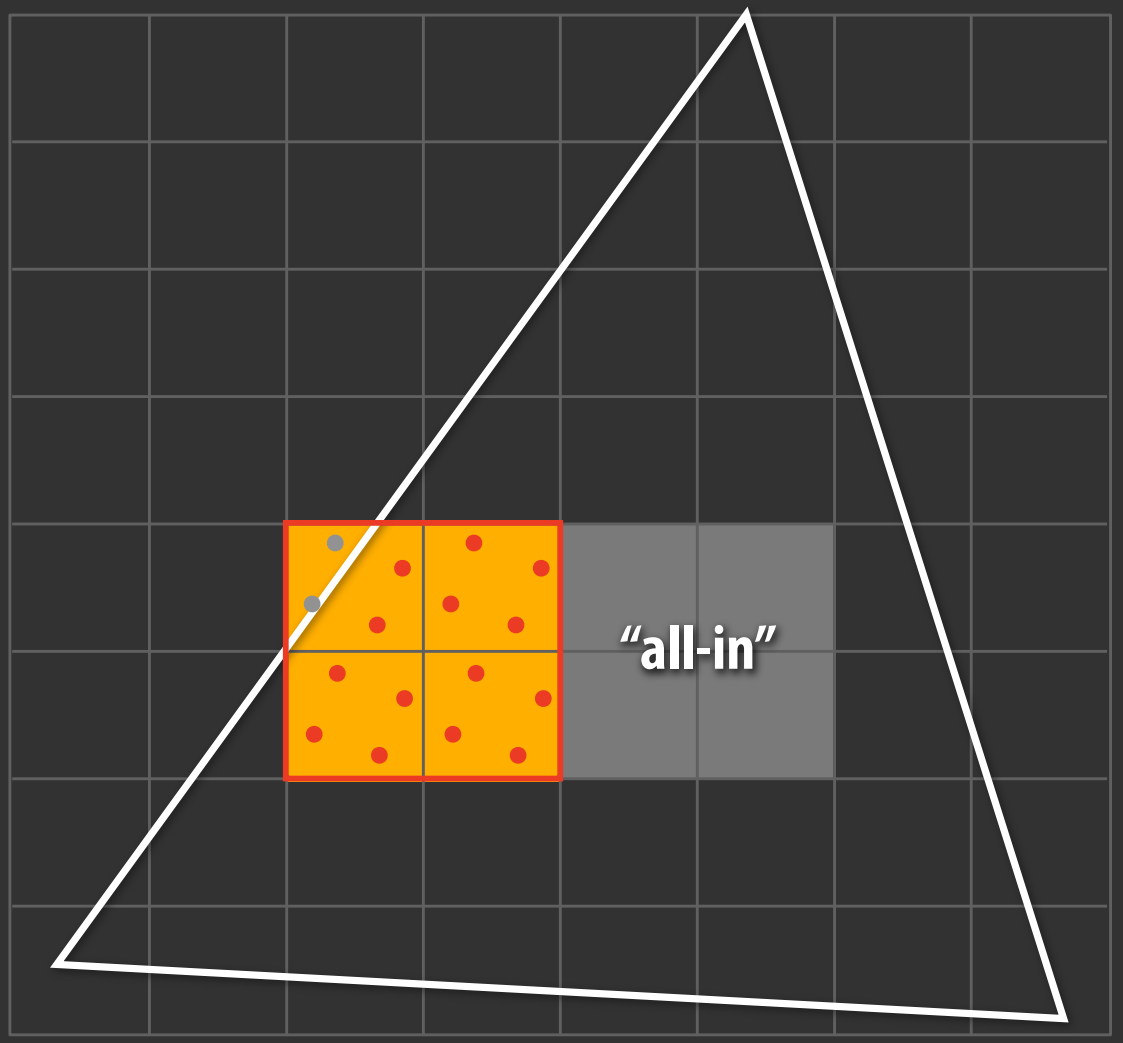
## Compute coverage using point-in-triangle tests



# Compute “possibly covered” pixels

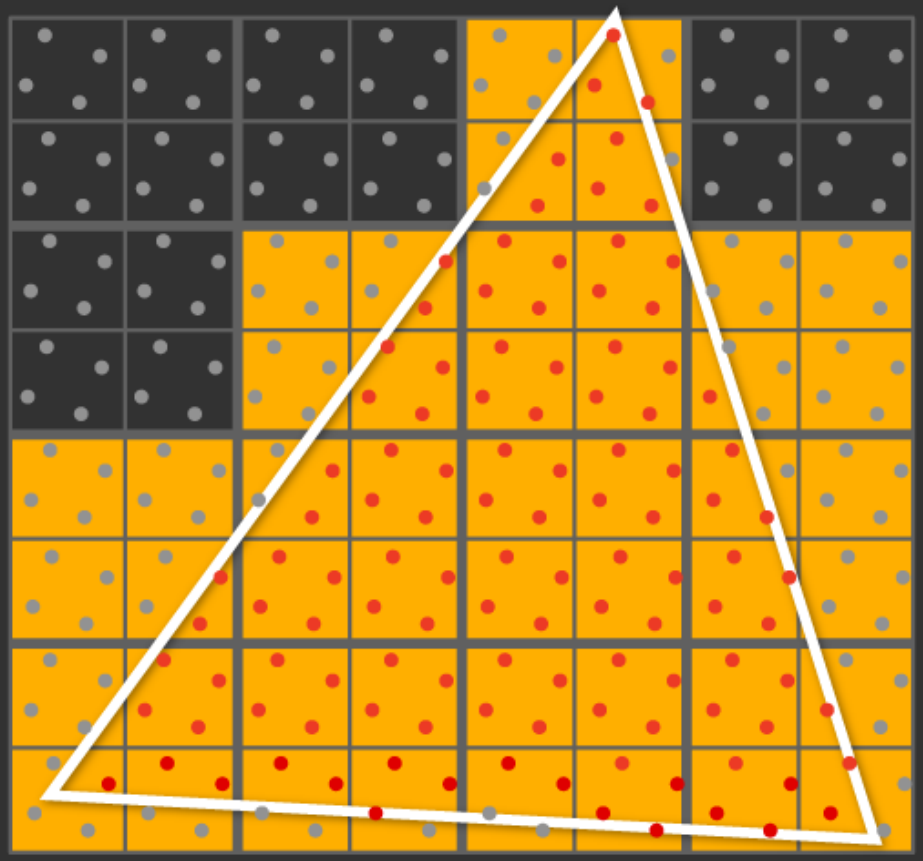


# Data-parallel sample tests

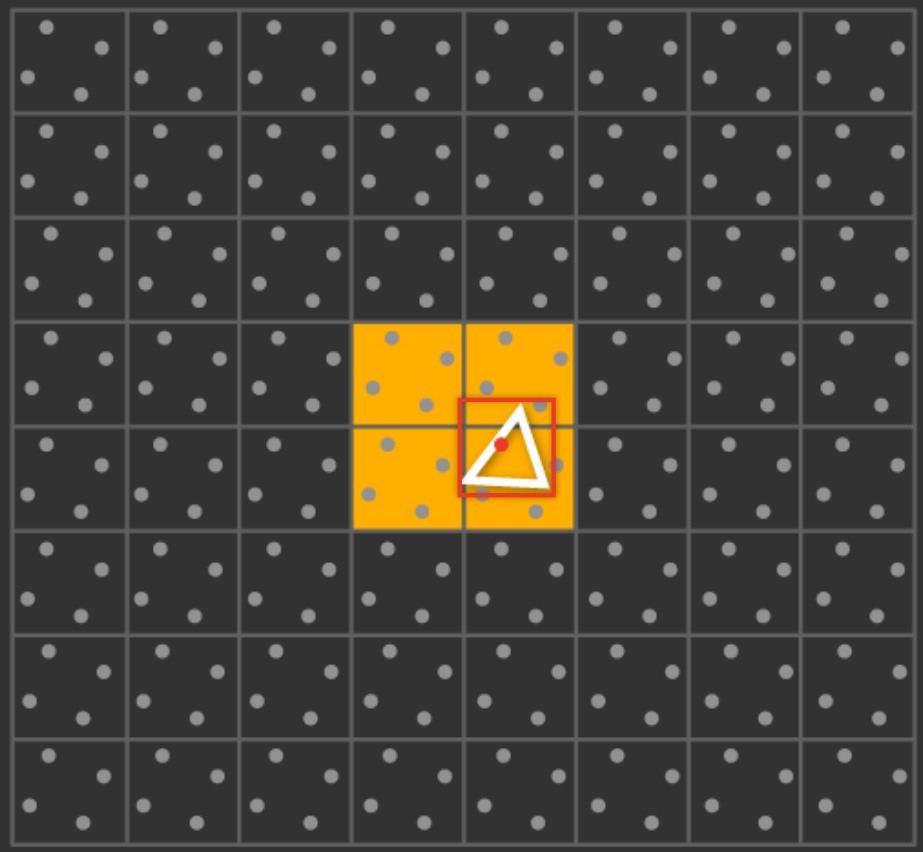



[Pineda 88]
[Fuchs 89]
[Greene 96]
[Seiler 08]

# 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**

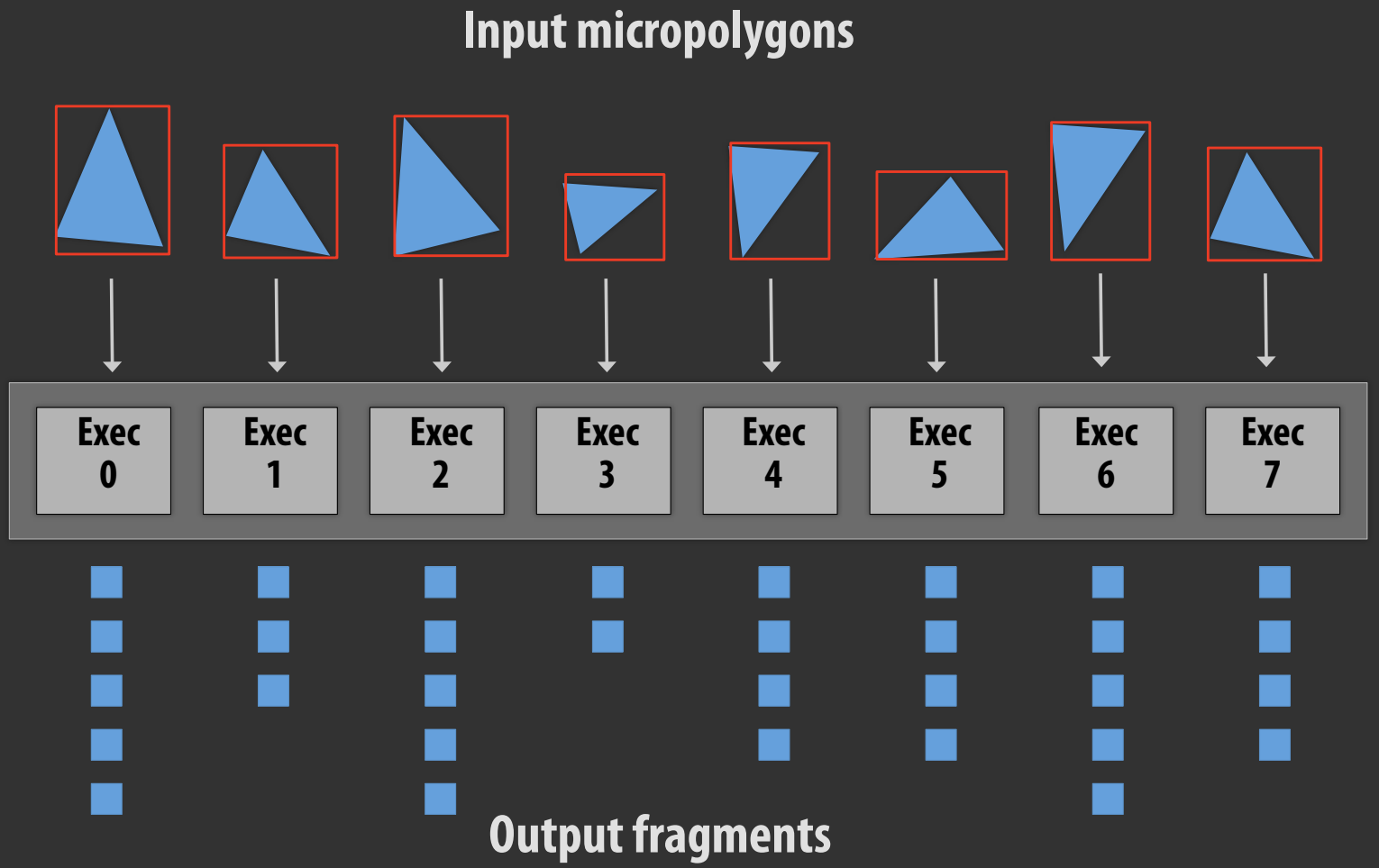
```
Compute subpixel bbox of MP
```

**Test**

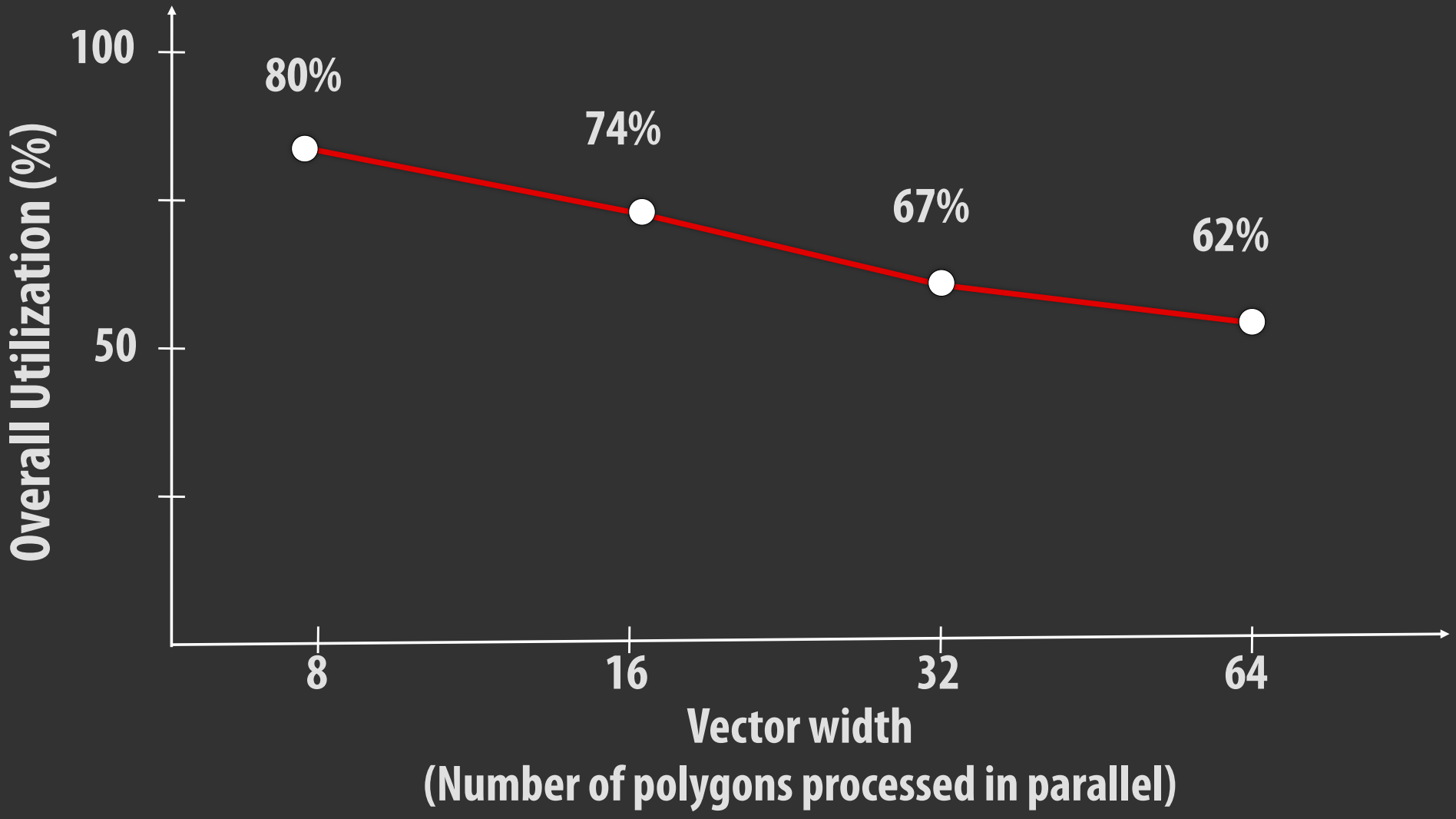
```
For each sample in bbox
   Test MP-sample coverage
```

# Parallel micropolygon rasterization

## Process multiple micropolygons simultaneously



# MP 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 16x MSAA
## (~15 million polygon scene at 60 Hz)

## Estimated cost of GPU software implementation in CUDA:

## About seven 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

# Temporal anti-aliasing (motion blur)

- Increases rasterization costs further (3-7x)
  - More point-in-triangle tests (5% of tested samples lie in polygon)
  - Individual tests are more expensive

# Lesson learned:

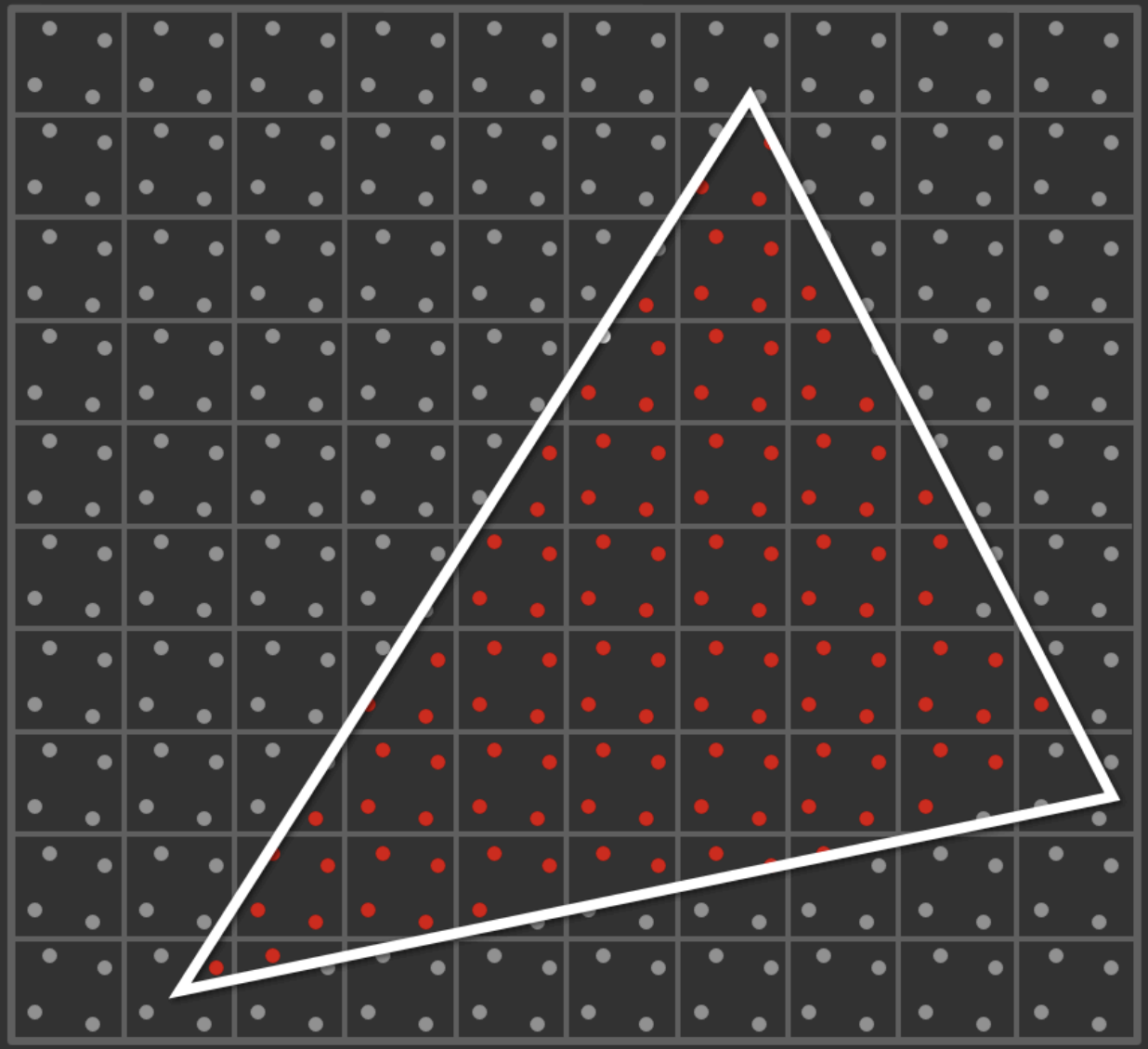
**Despite the speed of the programmable parts of a GPU, I expect to see hardware rasterization around for awhile**

# SHADING:

## Current GPUs shade small triangles inefficiently

# Multi-sample locations

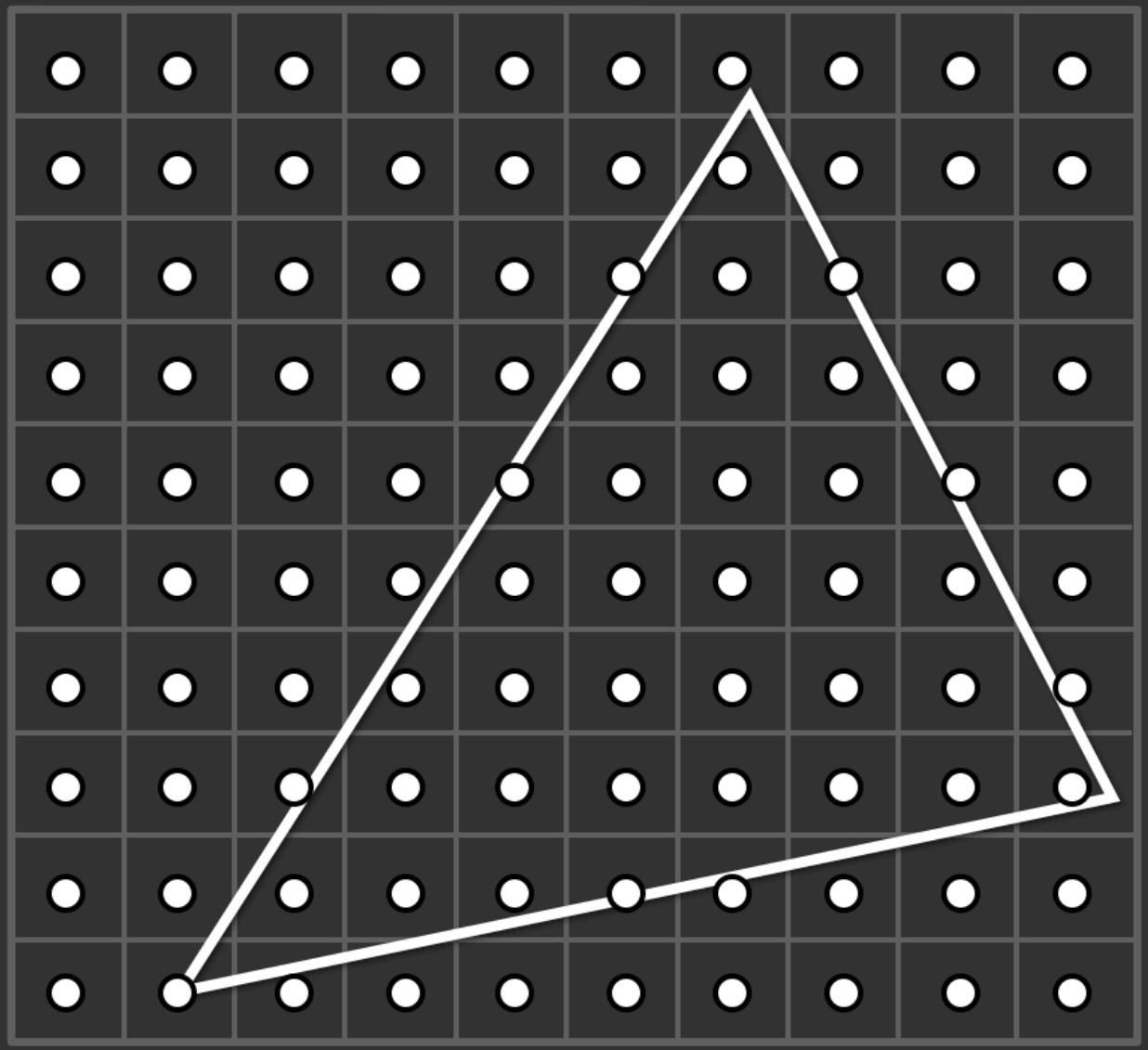
[Akeley 93]



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

# Shading sample locations

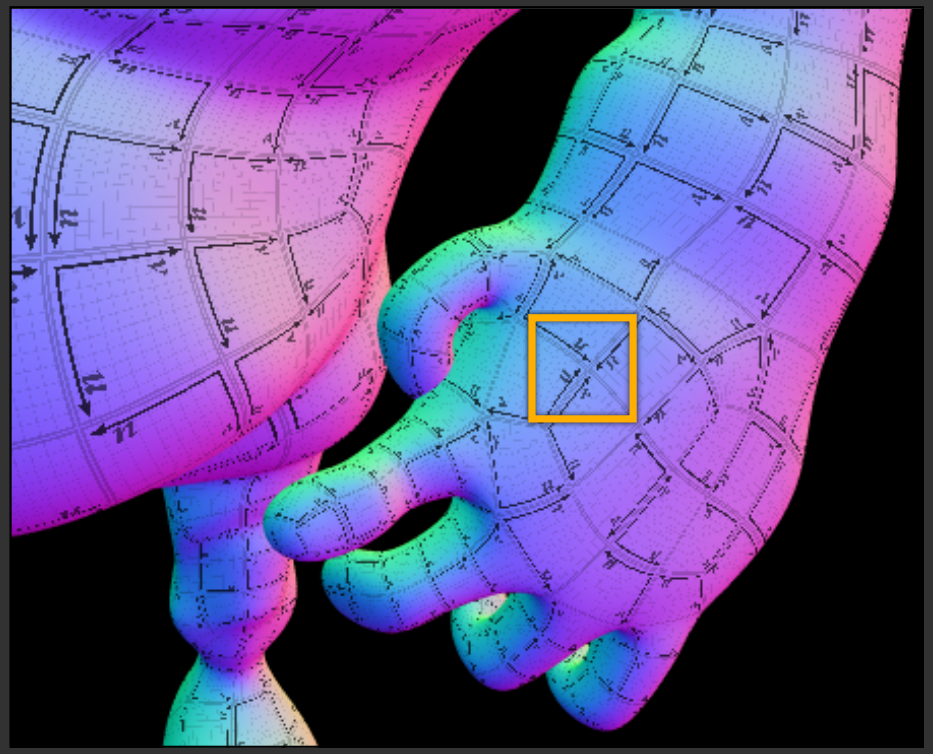
[Akeley 93]



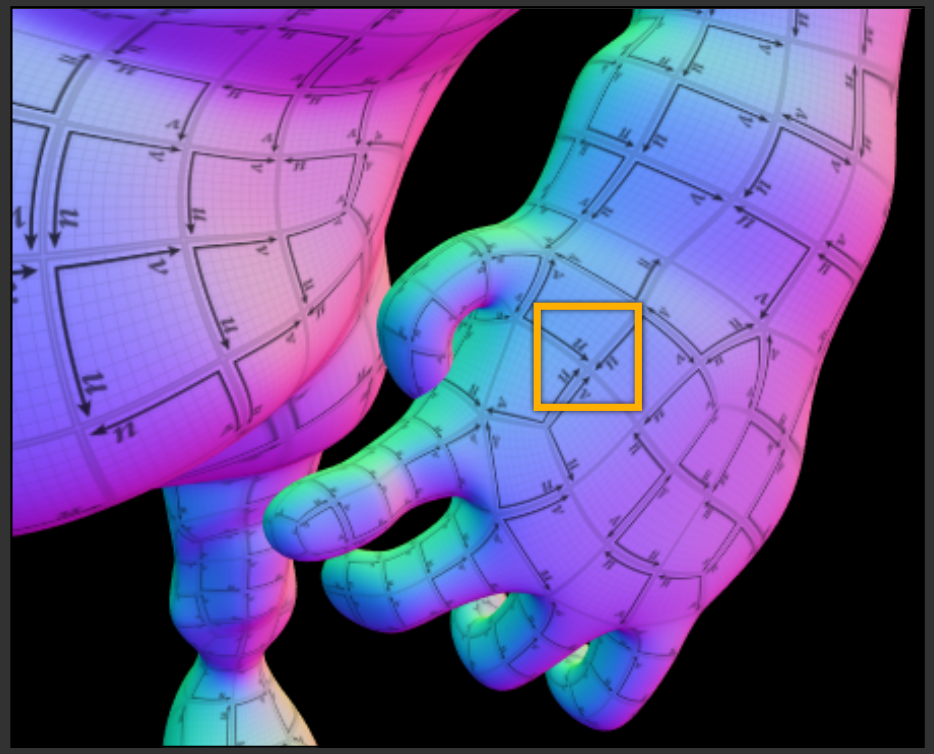
Sample shading once per pixel

# Texture data is pre-filtered to avoid aliasing

## (one shade per pixel is sufficient)



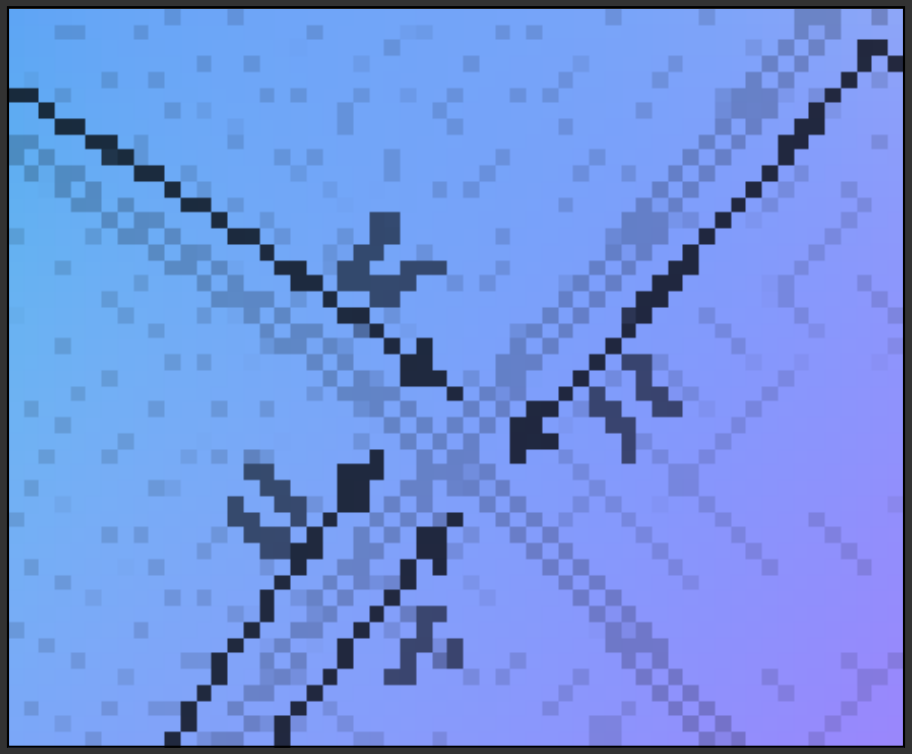
No pre-filtering
(aliased result)



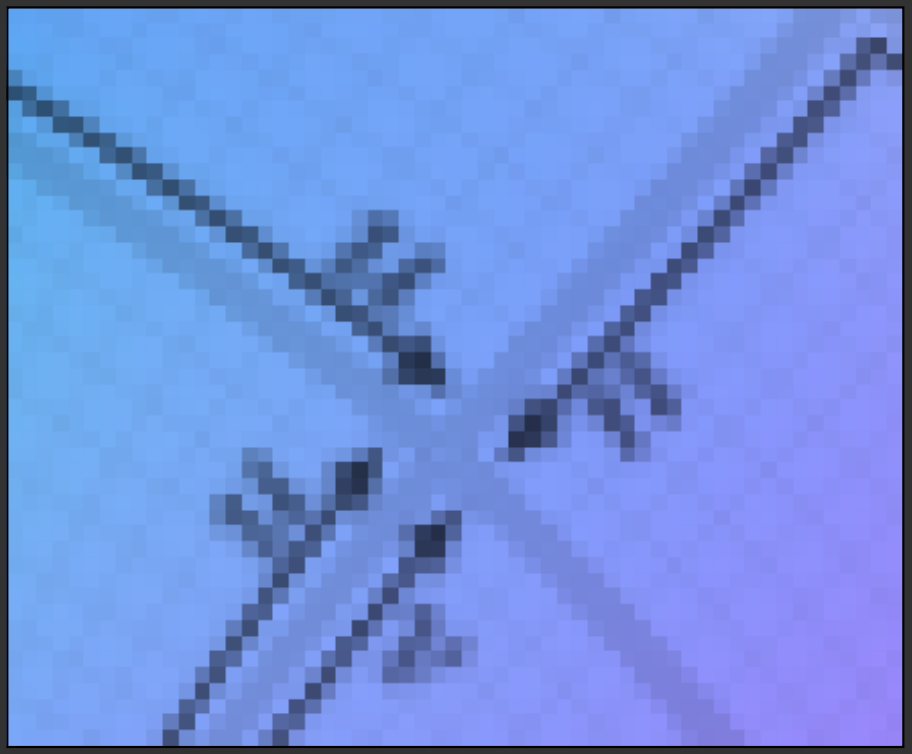
Pre-filtered texture

# Texture data is pre-filtered to avoid aliasing

## (one shade per pixel is sufficient)



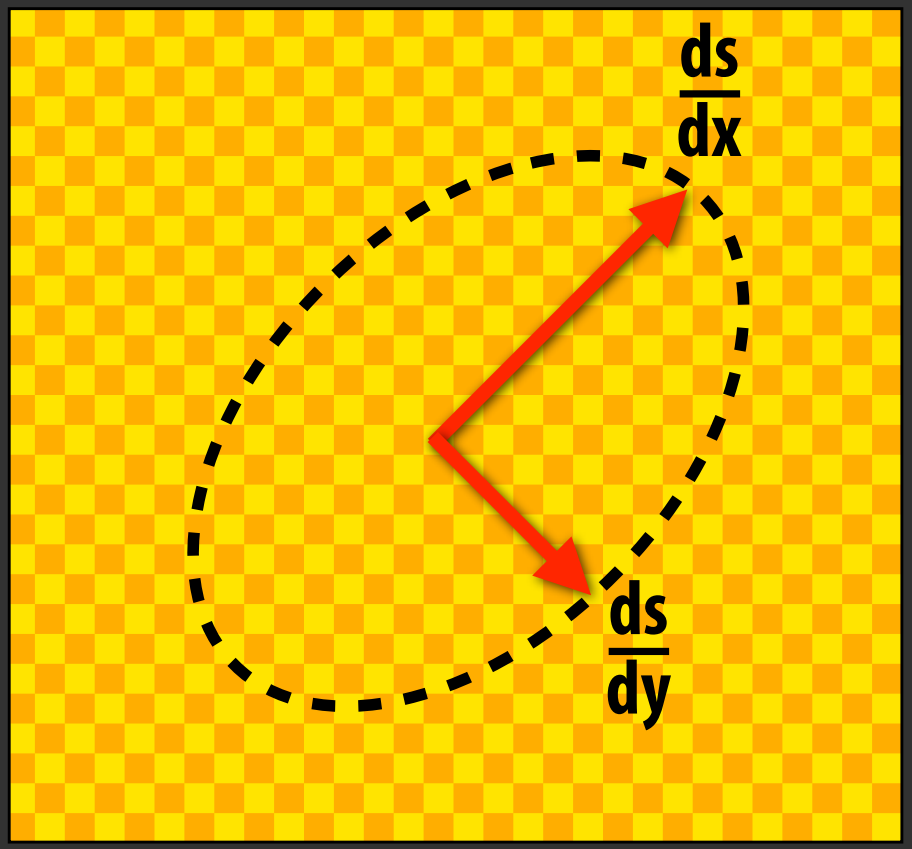
No pre-filtering
(aliased result)



Pre-filtered texture

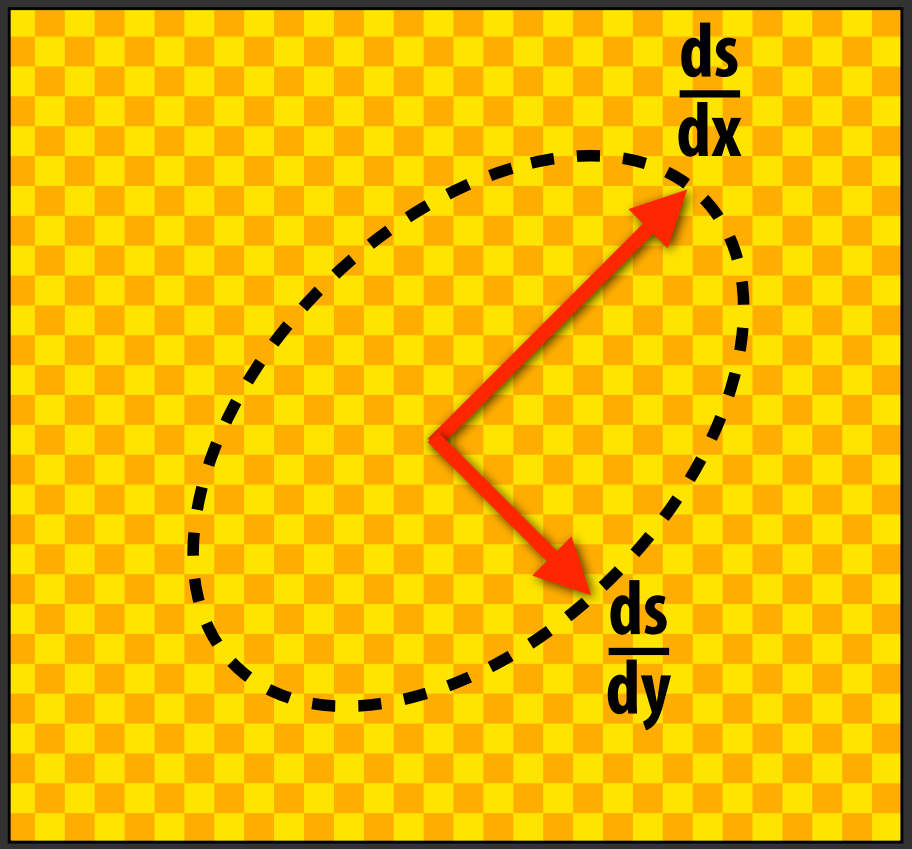
# Surface derivatives are needed for texture filtering

Texture data

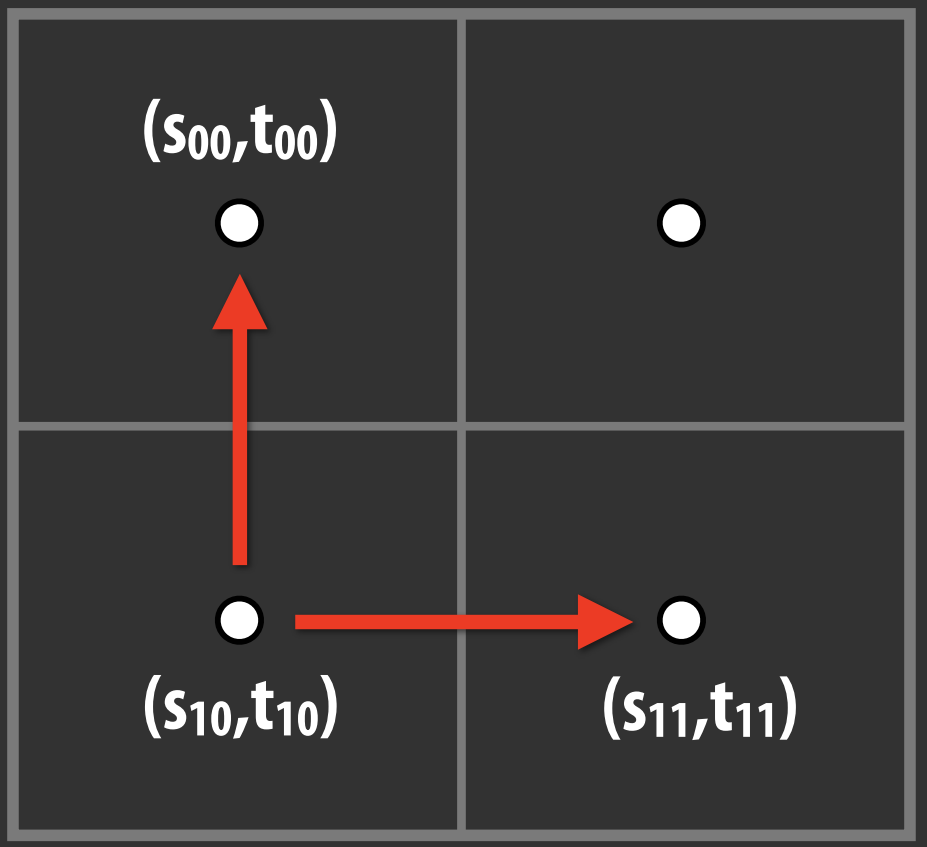


# GPUs shade quad fragments (2x2 pixel blocks)

Texture data

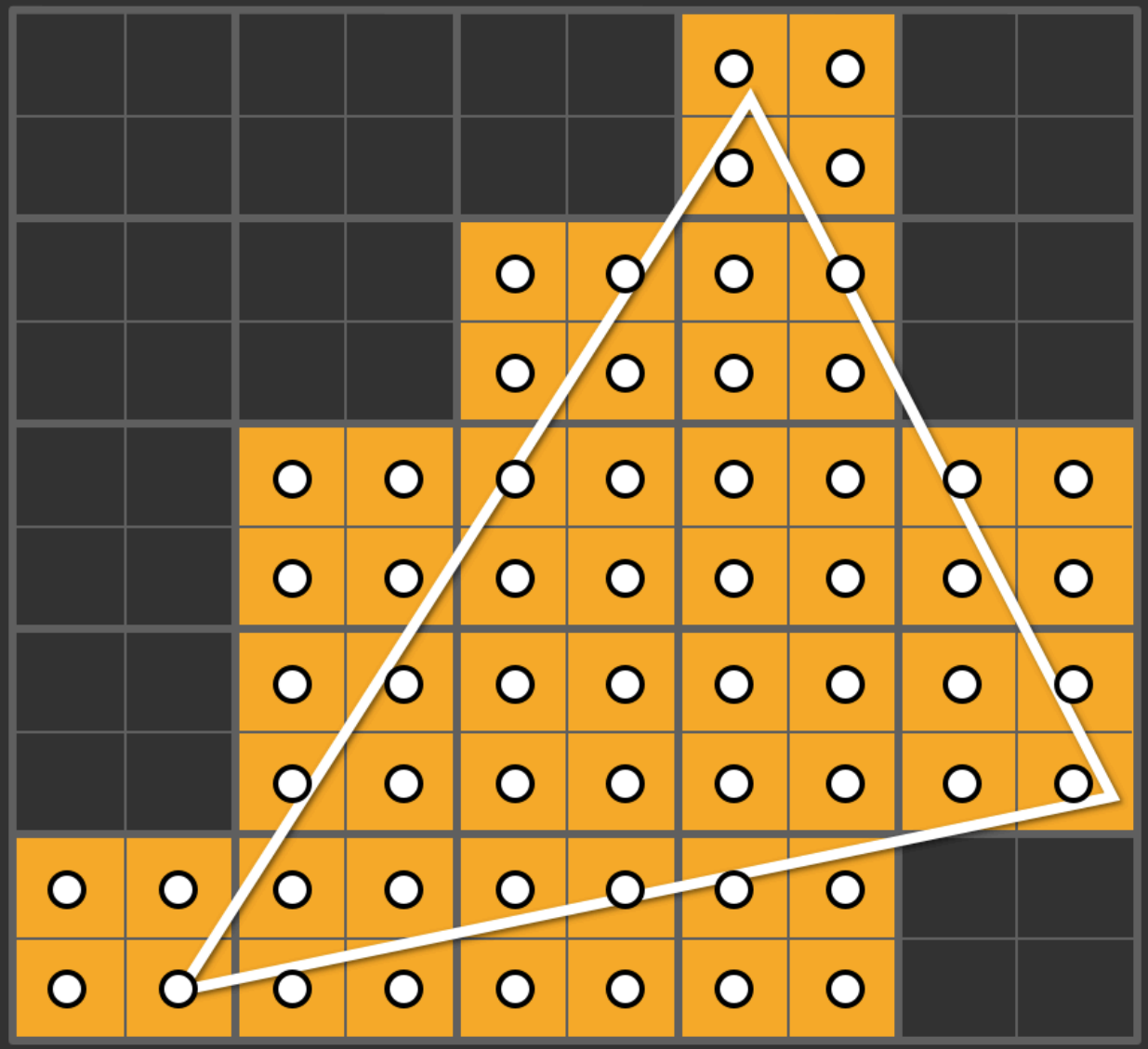



Quad fragment

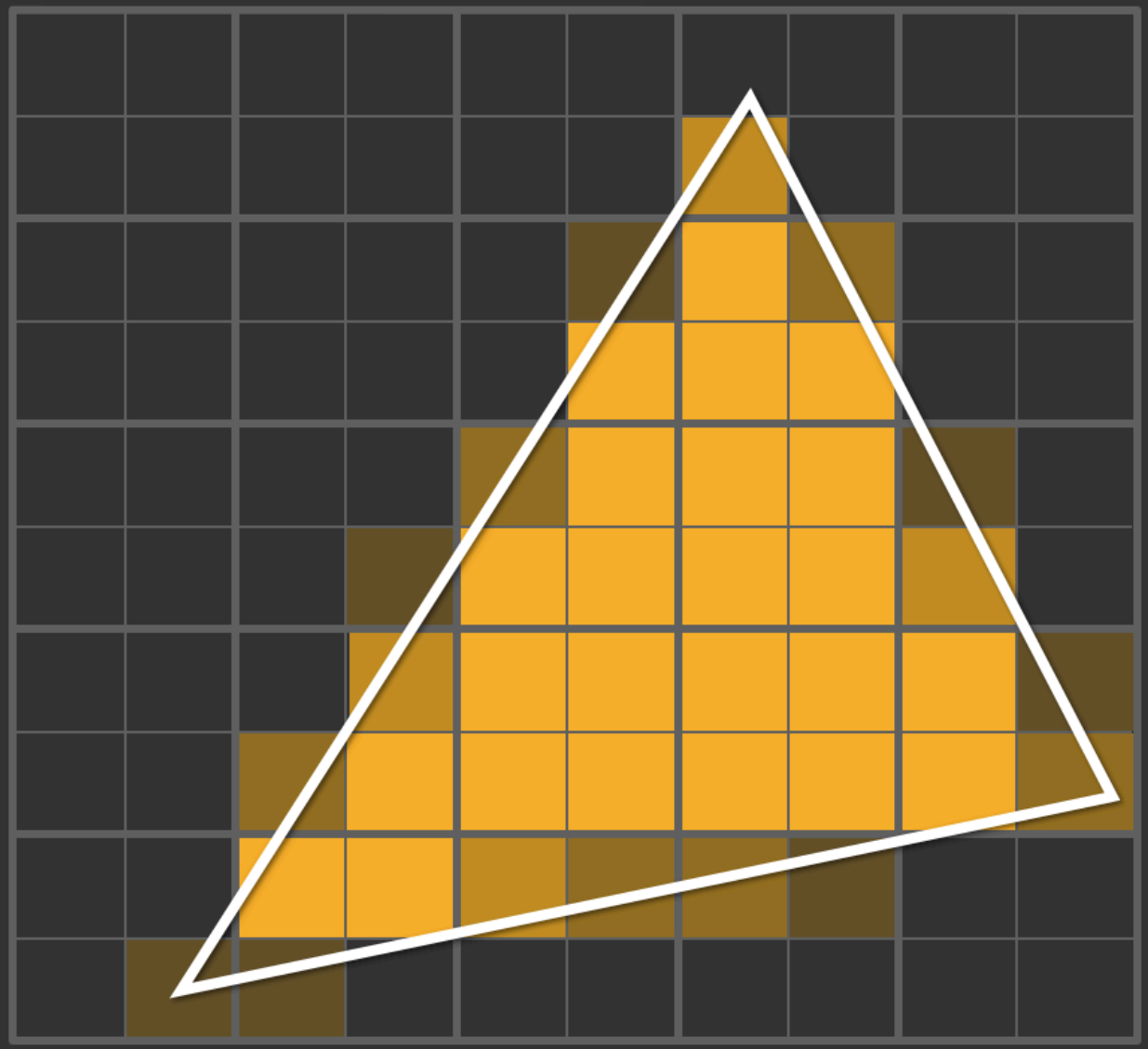



use differences between neighboring texture coordinates to estimate derivatives

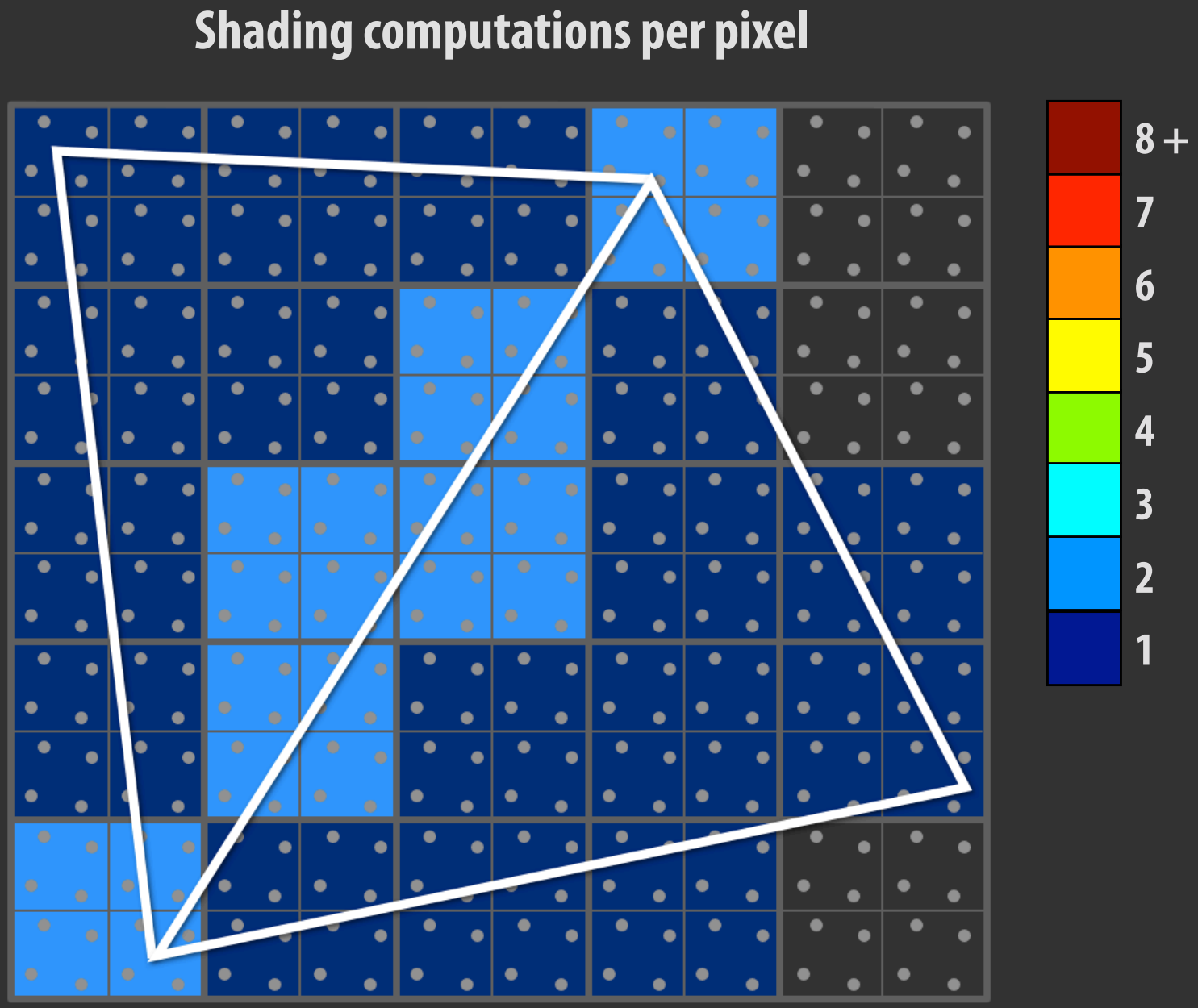
# Shaded quad fragments



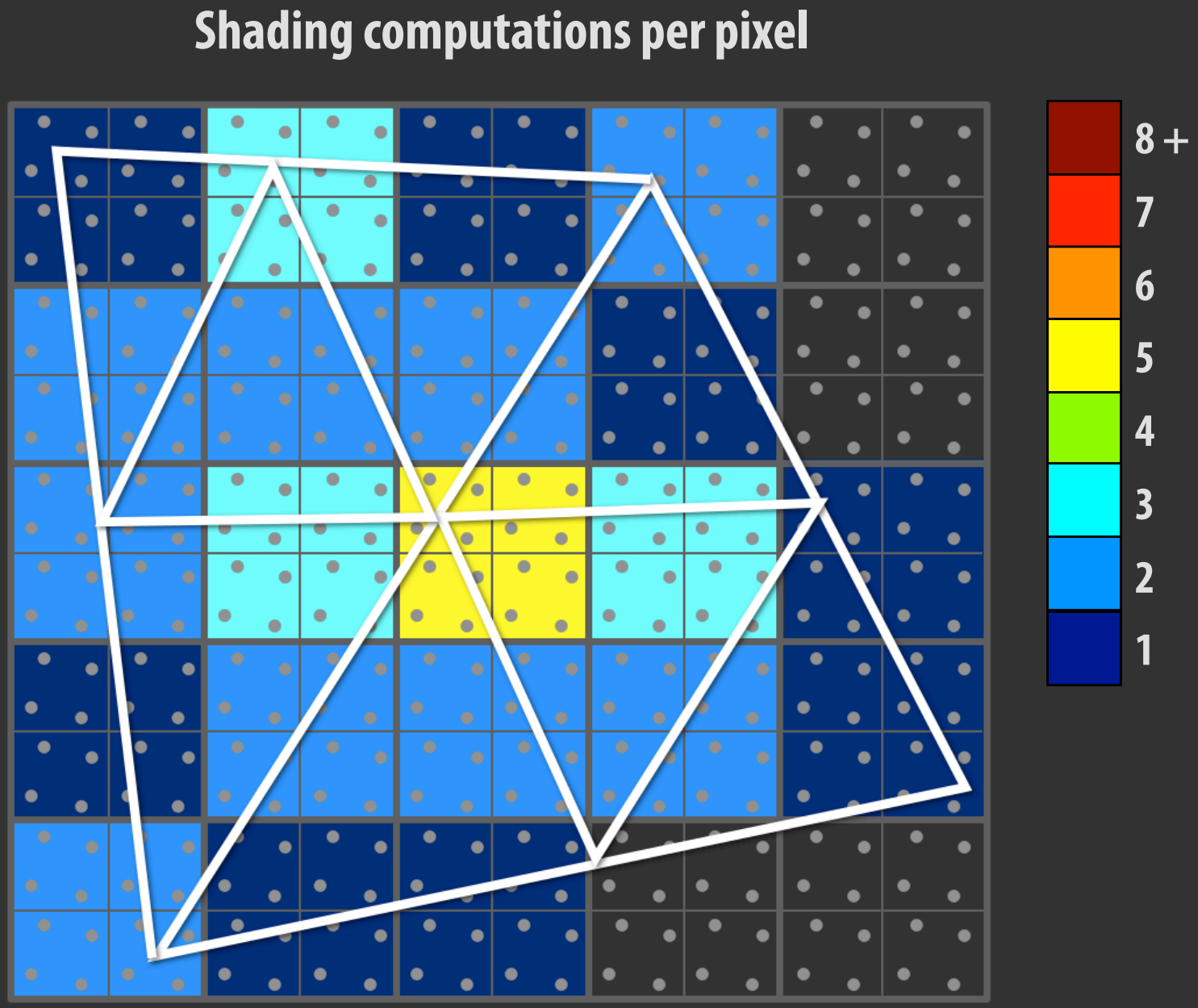
# Final pixel values



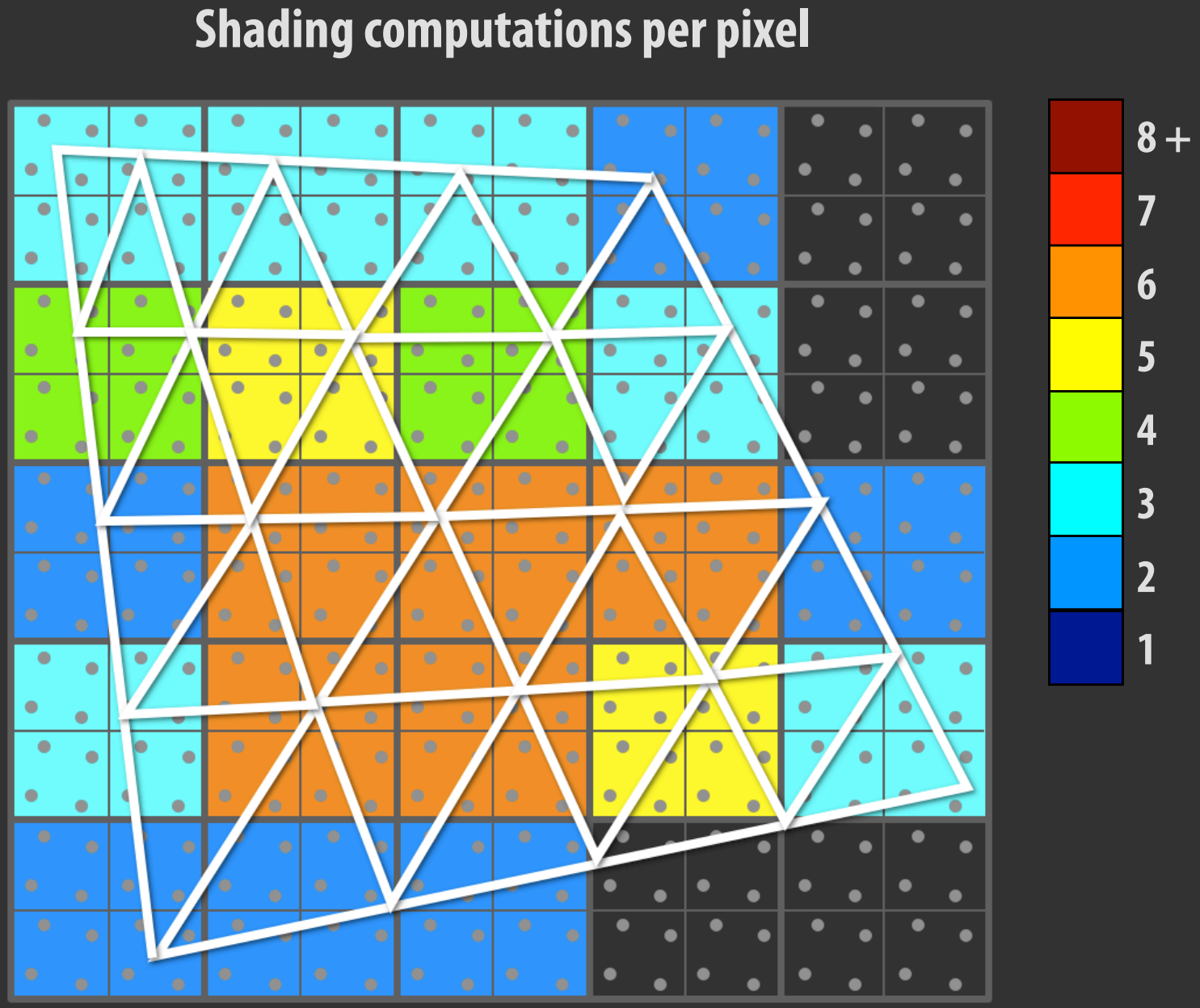
# Pixels at triangle boundaries are shaded multiple times



# Pixels at triangle boundaries are shaded multiple times

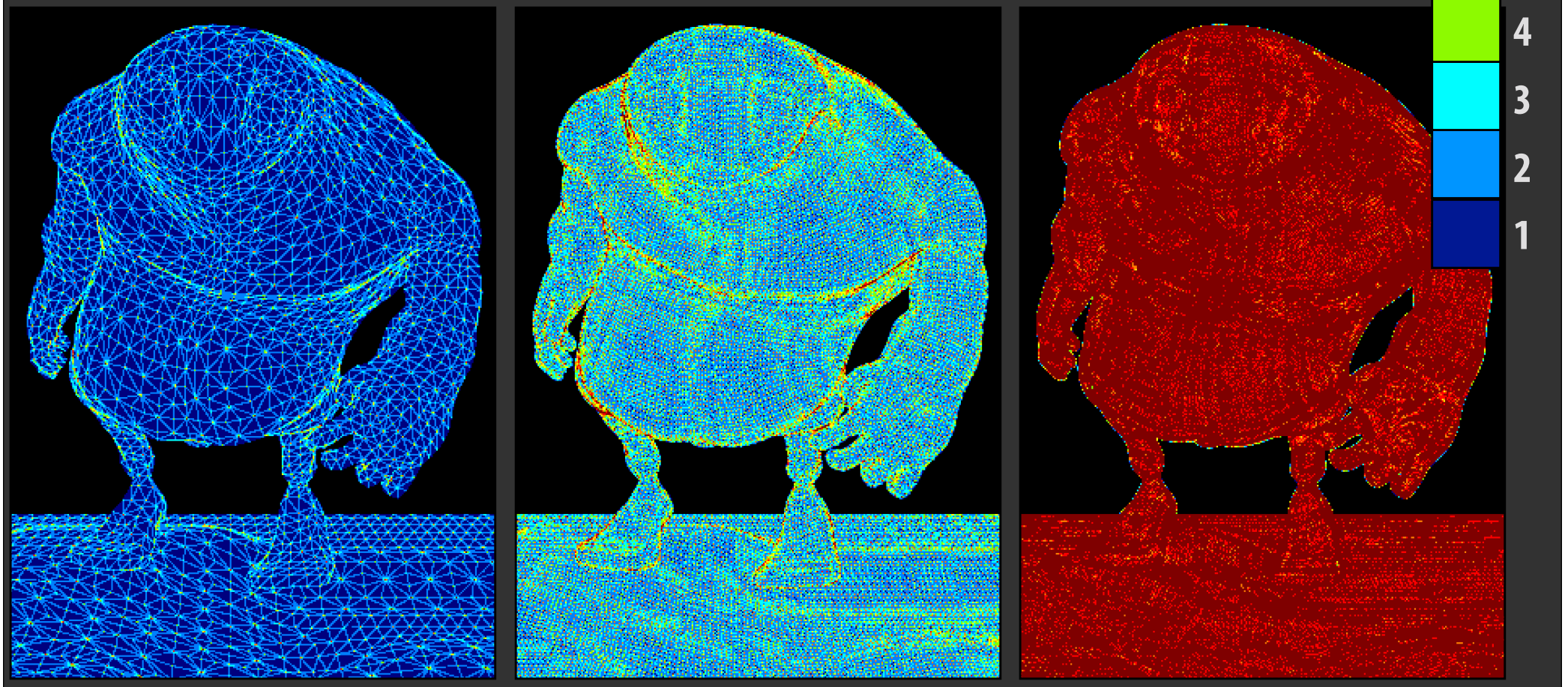


# Pixels at triangle boundaries are shaded multiple times



# Small triangles result in extra shading

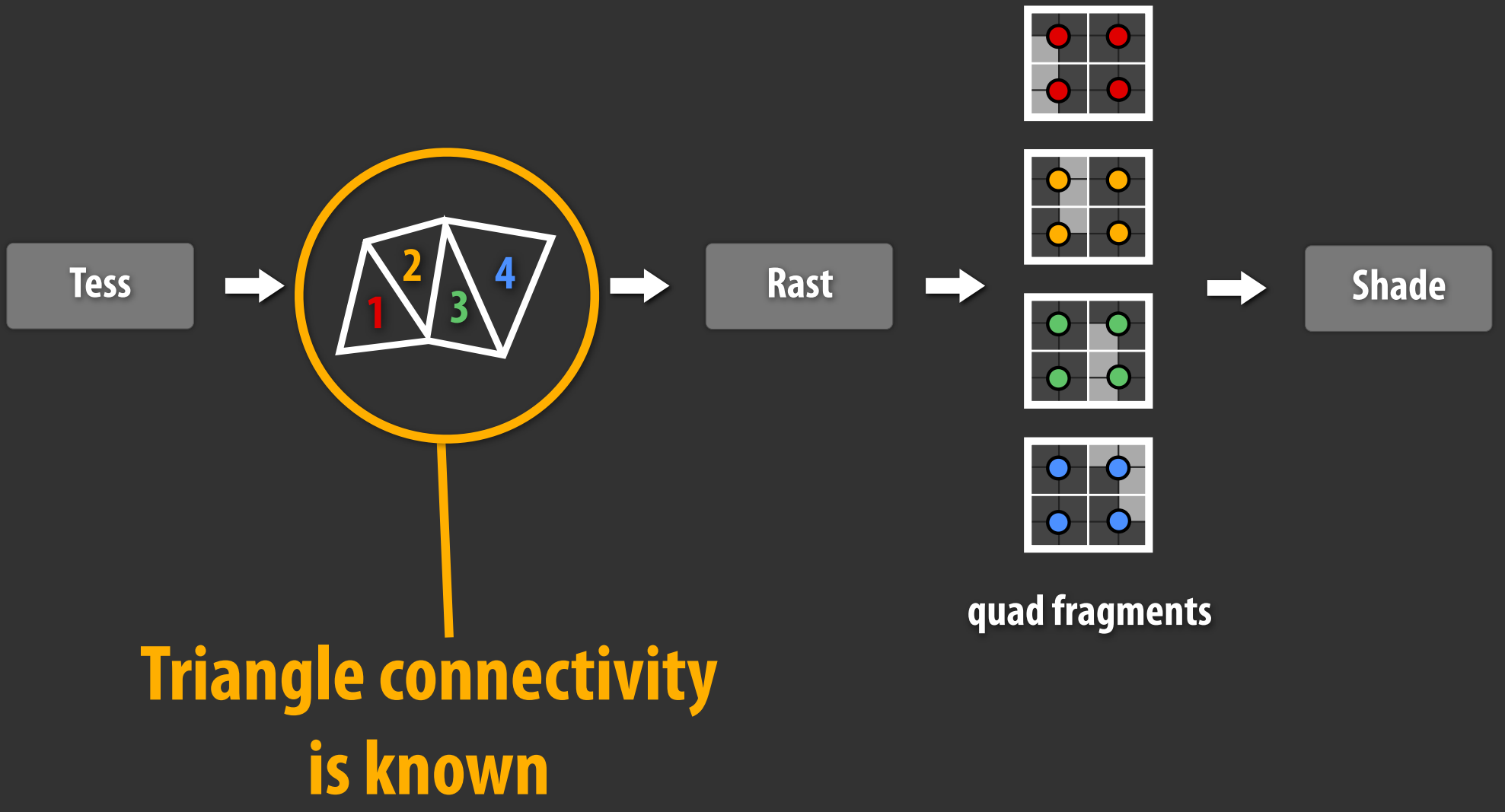
## Shading computations per pixel



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

**Solution:**
**Quad-fragment merging**

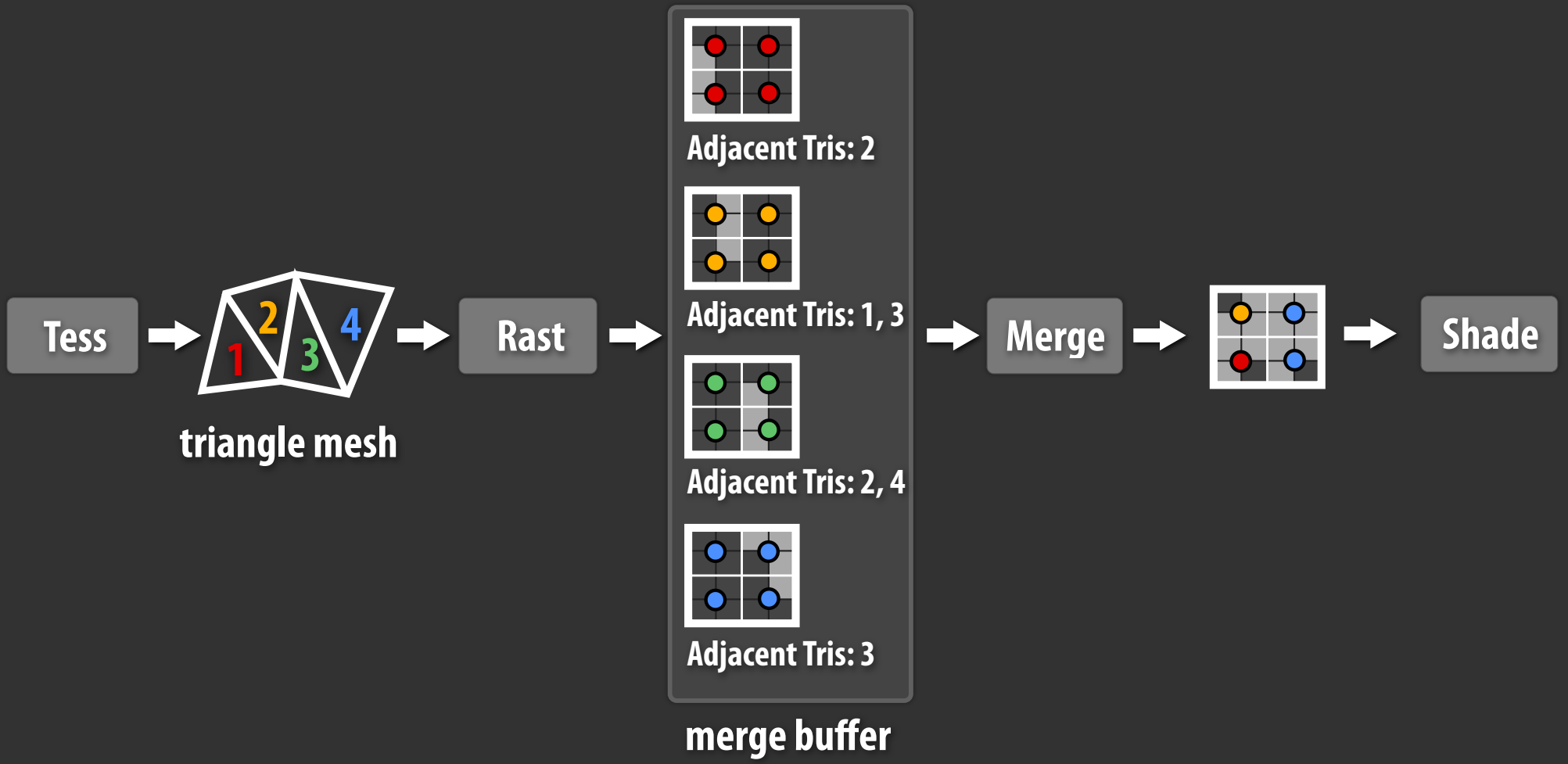
# GPU pipeline: triangle connectivity is known



# Pipeline with quad-fragment merging


[Fatahalian et al. SIGGRAPH 2010]

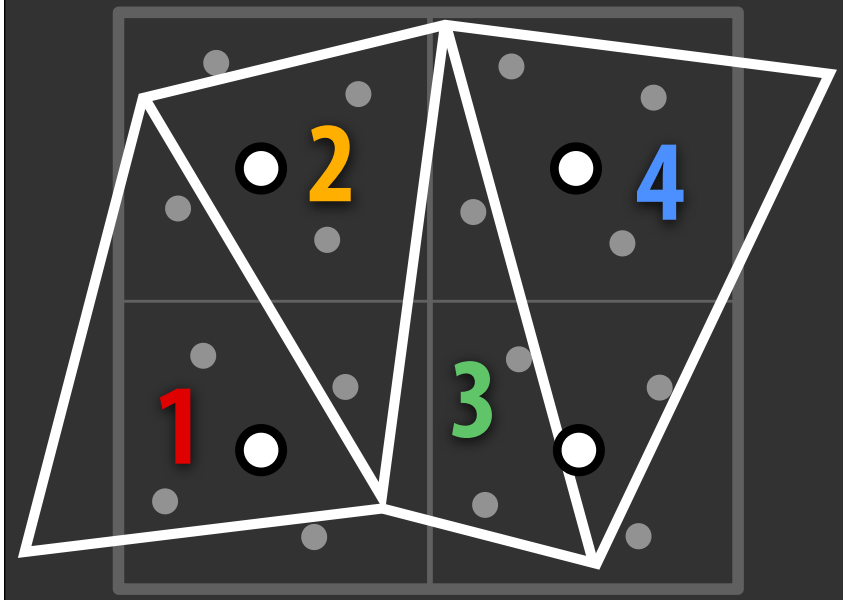
# Pipeline with quad-fragment merging



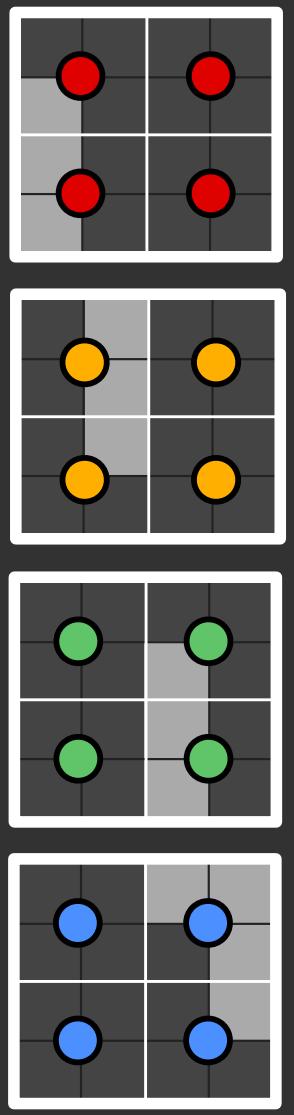

[Fatahalian et al. SIGGRAPH 2010]

# How to merge quad fragments

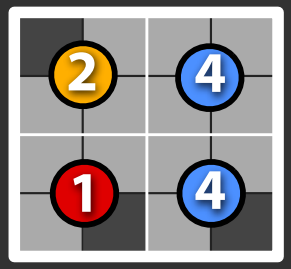
Mesh triangles




Rasterized quad fragments



Merged quad fragment



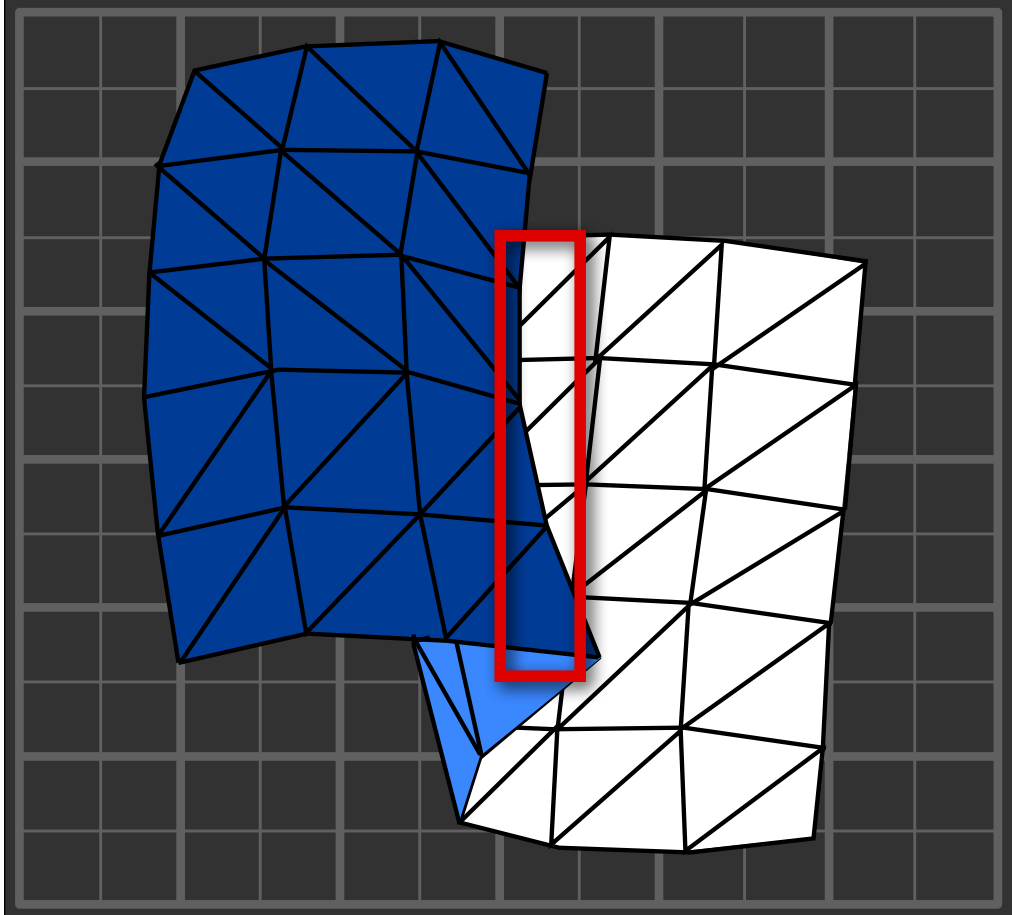
# When to merge quad fragments

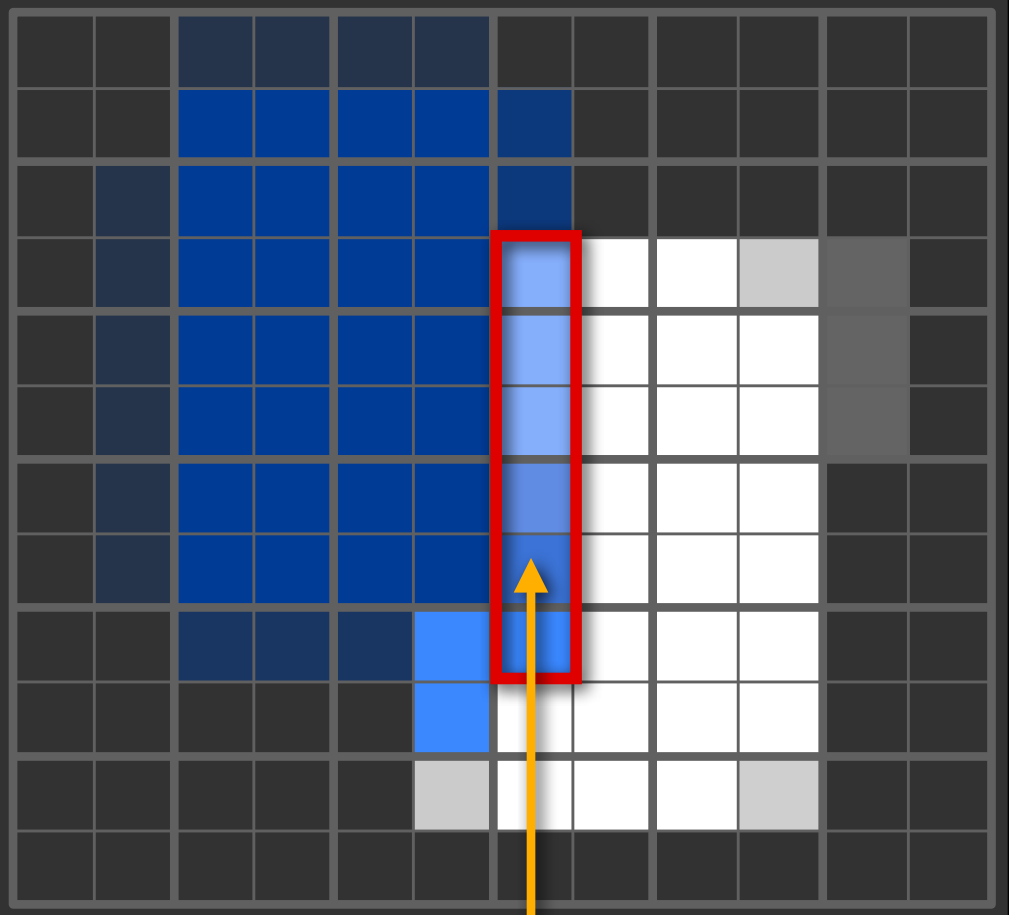
## Challenge: avoiding merges that introduce visual artifacts

# Example: surface with a silhouette

Triangle mesh

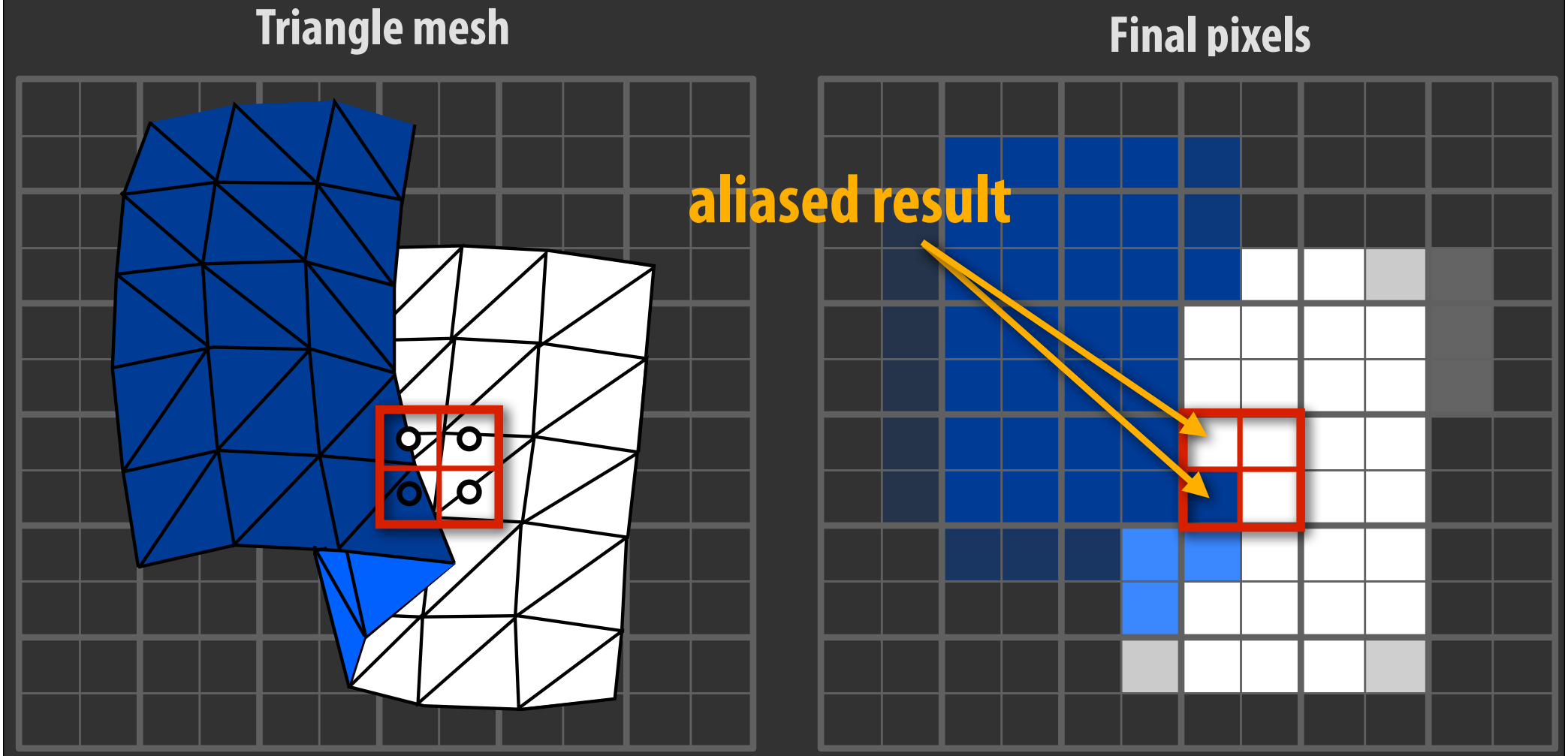
Final pixels





anti-aliased 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
- 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

Big Guy Scene

No merging

Merging

Shading computations / pixel (avg)

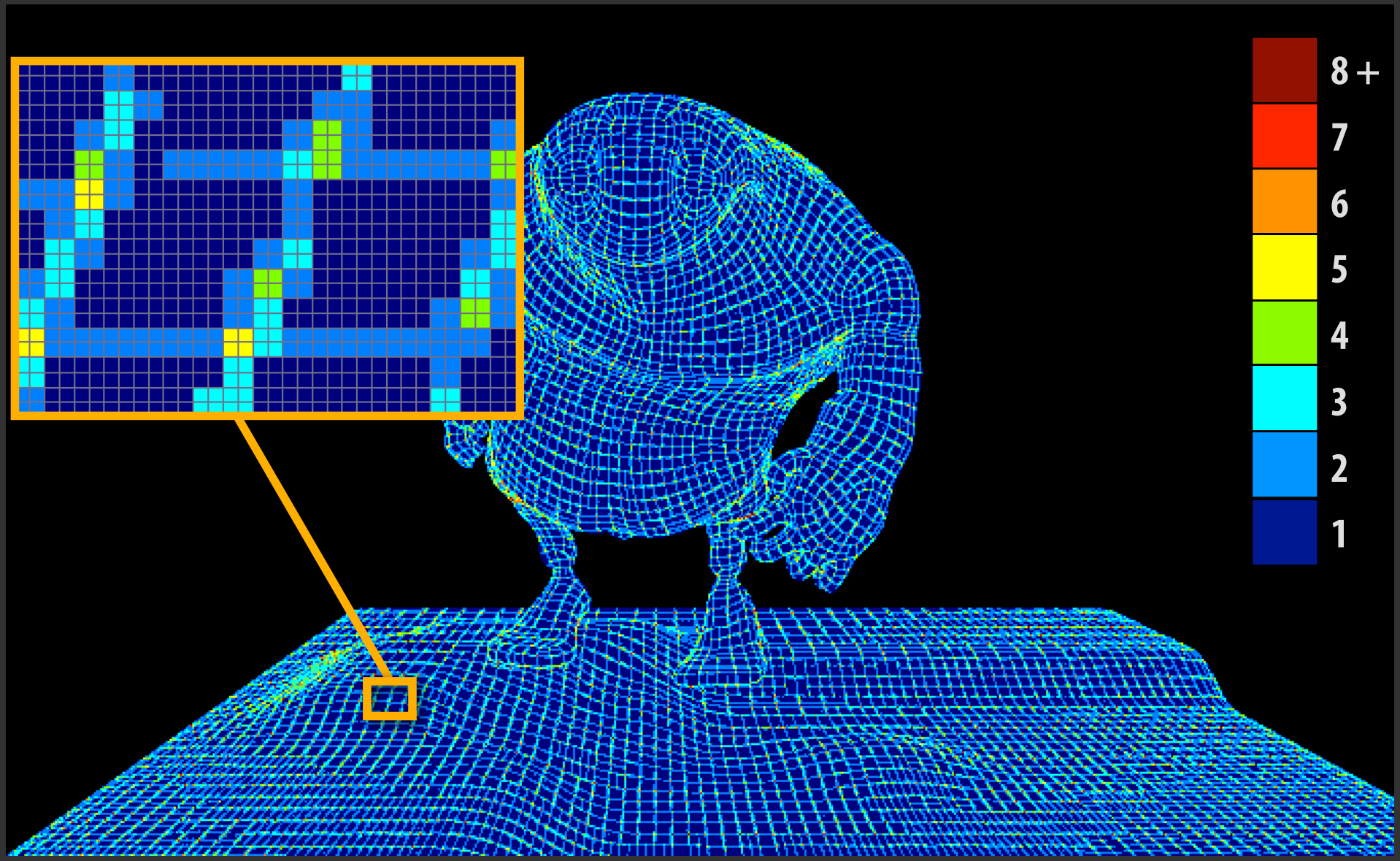
14
12
10
8
6
4
2
0

2
4
6
8
10

Average triangle area (pixels)

# Extra shading occurs at merging window boundaries

## 1/2 pixel area triangles



# Nearly identical visual quality *

Quad-fragment merging

Current GPU (no merging)

* see SIGGRAPH 2010 paper for more detail on possible artifacts

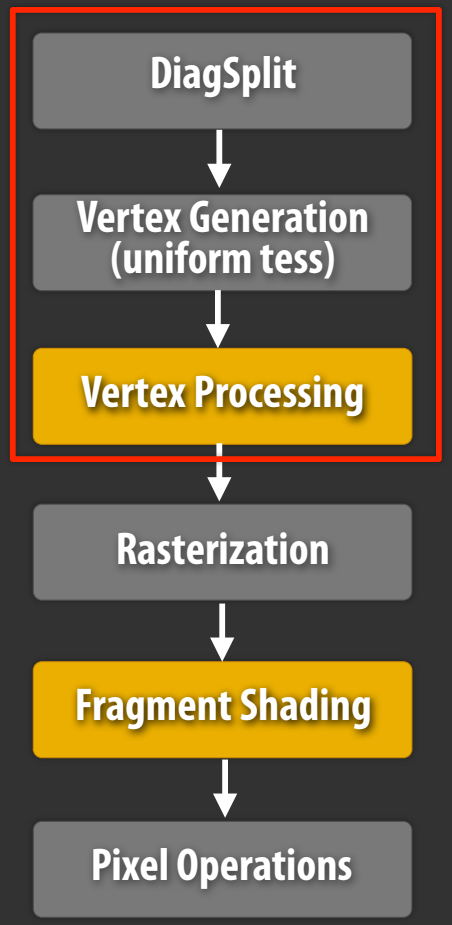
# Quad-fragment merging summary

- Reduces shading costs for high-res meshes
  - shade surfaces (not triangles) at a density of once per pixel
- Maintains high visual quality
  - Requires triangle connectivity
- Evolutionary: not a radical change to rasterization or shading
  - isolates dynamic communication/control, maintains data-parallel shading
  - uses quad fragments for derivatives
  - compatible with edge anti-aliasing
  - supports shading large triangles

# SUMMARY

# A micropolygon rendering pipeline

## DiagSplit adaptive tessellation:




---

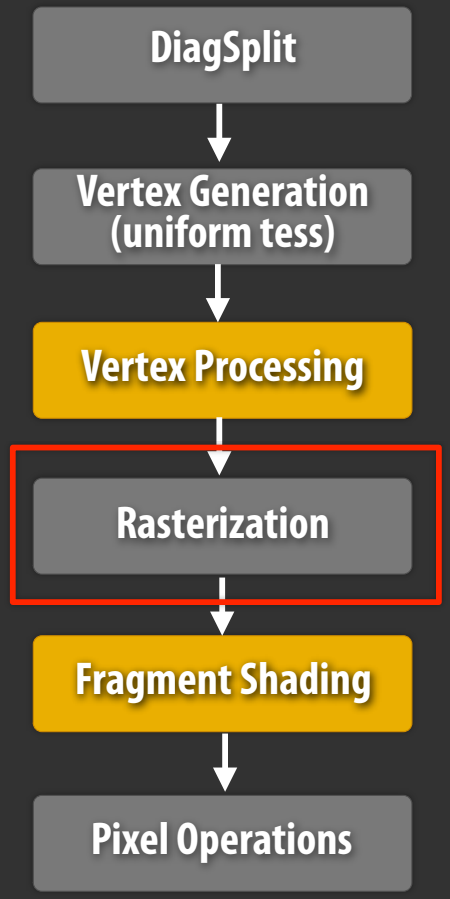
Reduces rendered vertex count

Simplifies micropolygon-parallel rasterization

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

# A micropolygon rendering pipeline

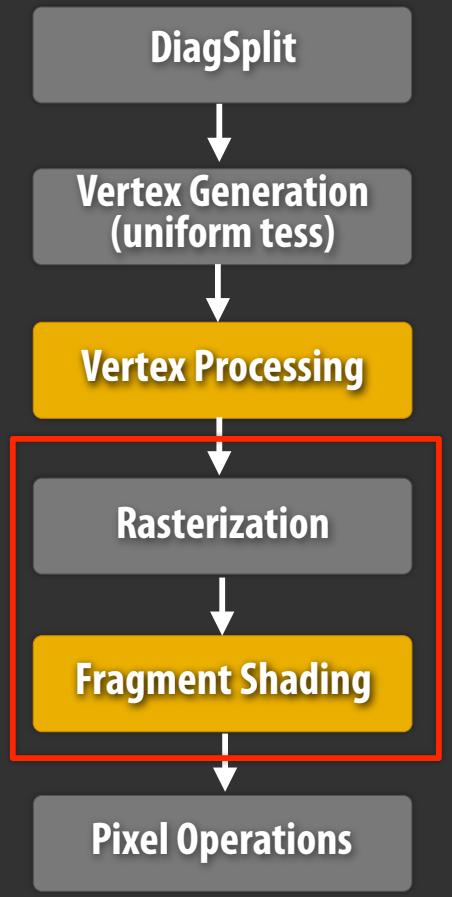
## Rasterization:




Simple, but expensive: fixed-function hardware highly desirable

# A micropolygon rendering pipeline

## Quad-fragment merging:



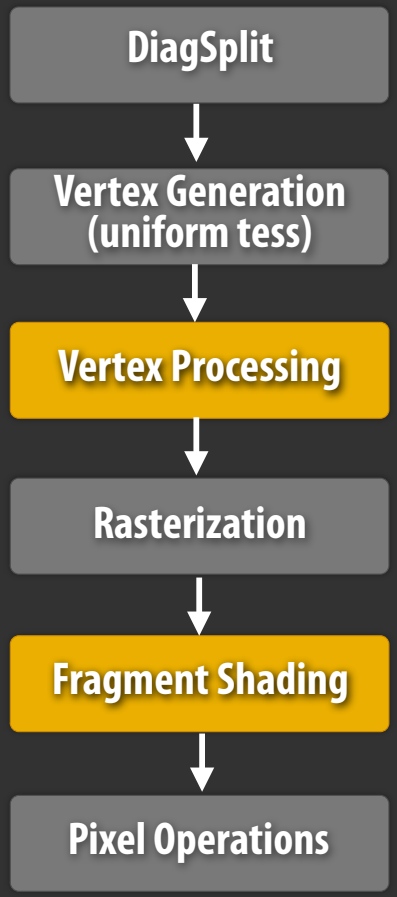

Reduces shaded fragments by 8x

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

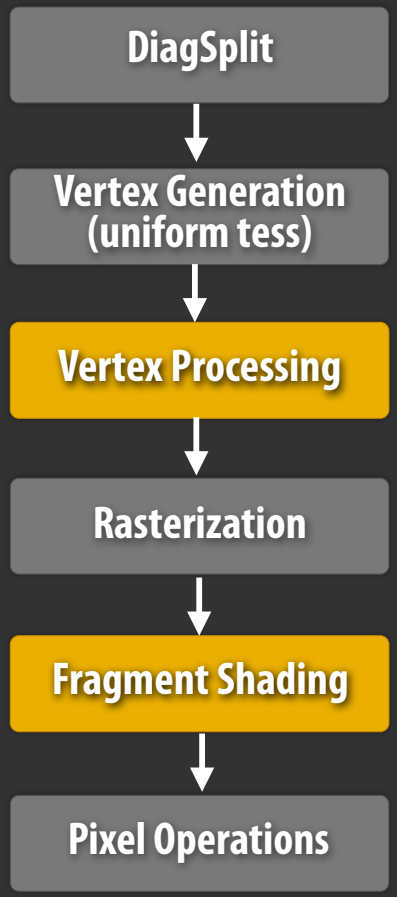
1. Willingness to change algorithms to fit the system



# Domain knowledge in graphics system design

1. Willingness to change algorithms to fit the system

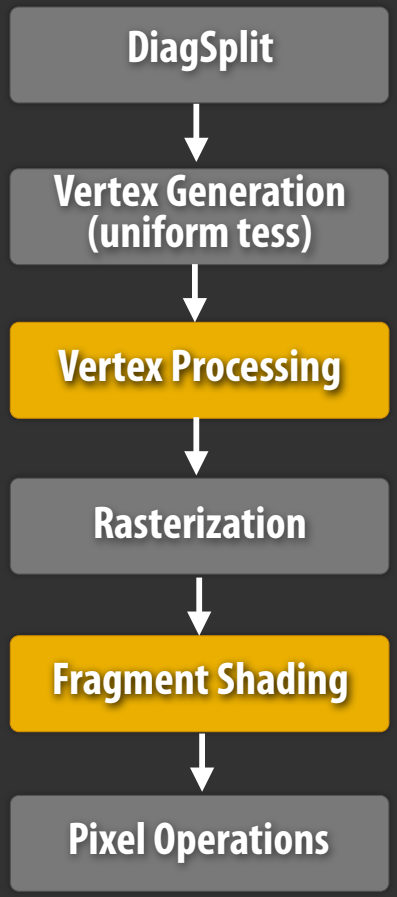
2. Unique approach to exploiting heterogeneity
   - isolate irregularity, sync
   - keep programmable stuff regular
   - programmable "stuff" forms the inner loops!



# Hot questions

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

(continue to evolve structure? or replace?)



# 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?