Lecture 27:

Parallel 3D graphics

(implementing the real-time graphics pipeline)

Parallel Computer Architecture and Programming

CMU 15-418/15-618, Spring 2016

Tunes

Ellie Goulding

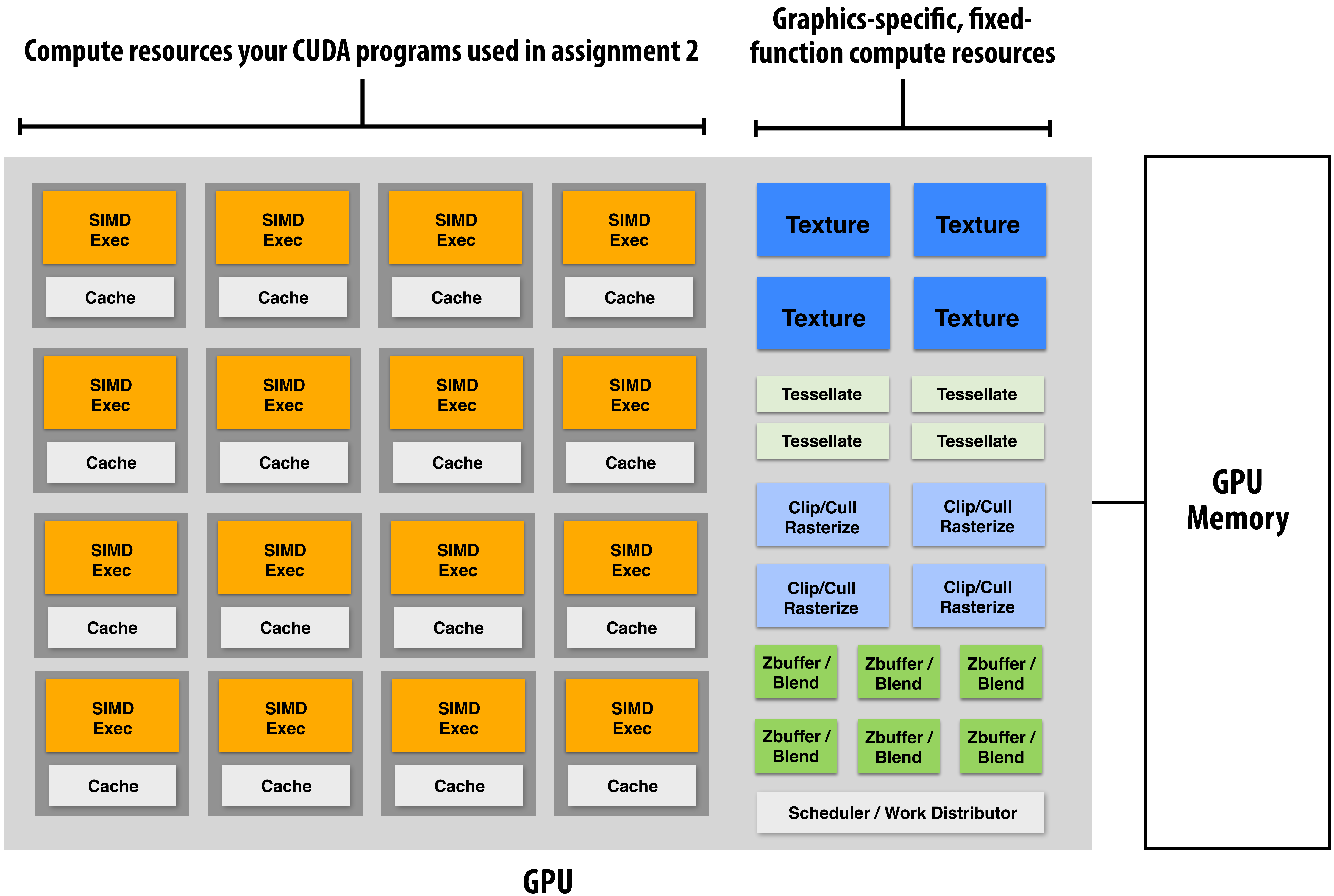
Tessellate

(Halcyon Days)

"How else do we get all the triangles to render in parallel?"

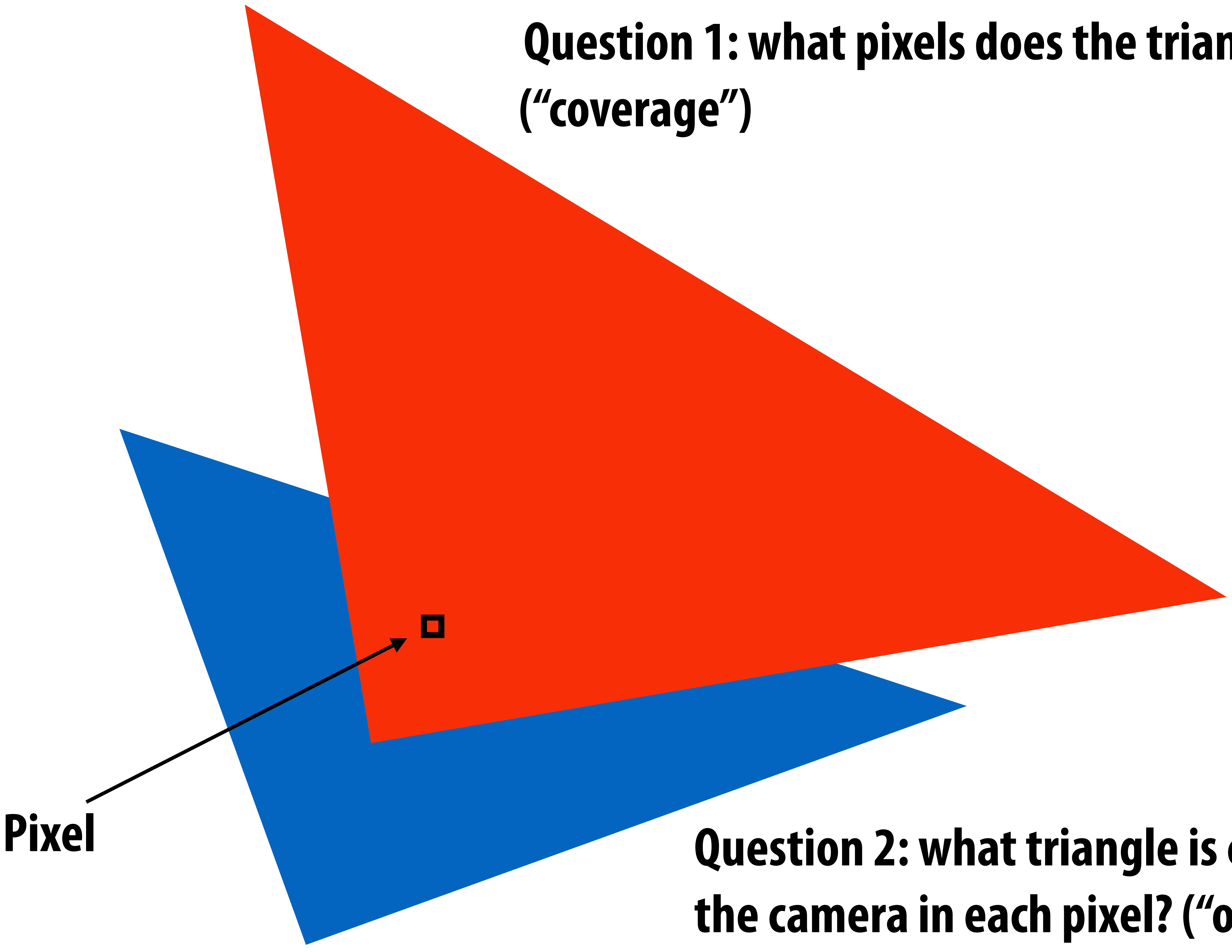
- Ellie

GPU's are heterogeneous multi-core processors



Let’s draw some triangles on the screen

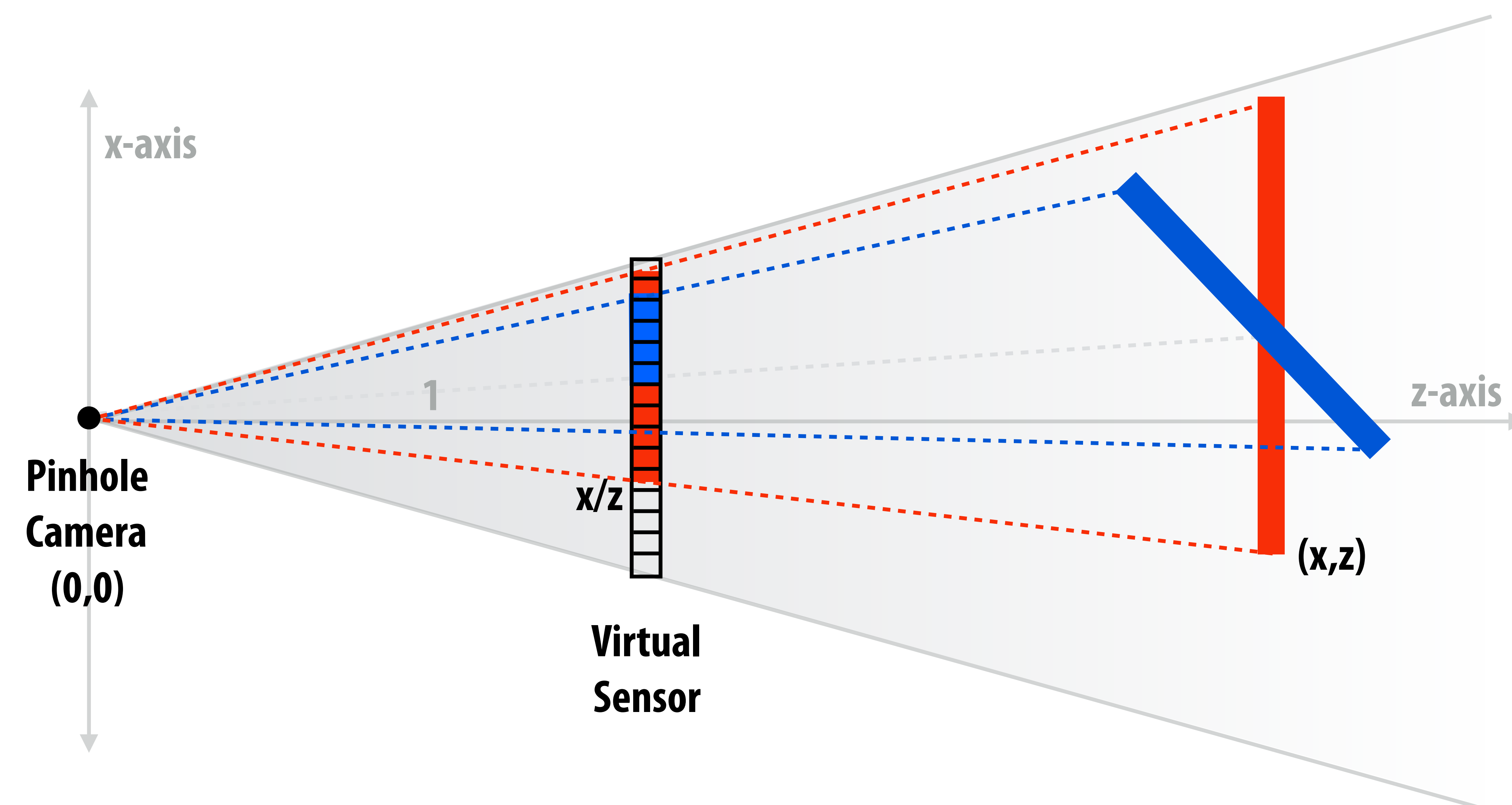
Question 1: what pixels does the triangle overlap? (“coverage”)



Question 2: what triangle is closest to the camera in each pixel? (“occlusion”)

The visibility problem

- An informal definition: what scene geometry is visible within each screen pixel?
  - What scene geometry projects into a screen pixel? (coverage)
  - Which geometry is visible from the camera at that pixel? (occlusion)



The visibility problem (said differently)

- **In terms of rays:**
  - What scene geometry is hit by a ray from a pixel through the pinhole? (coverage)
  - What object is the first hit along that ray? (occlusion)

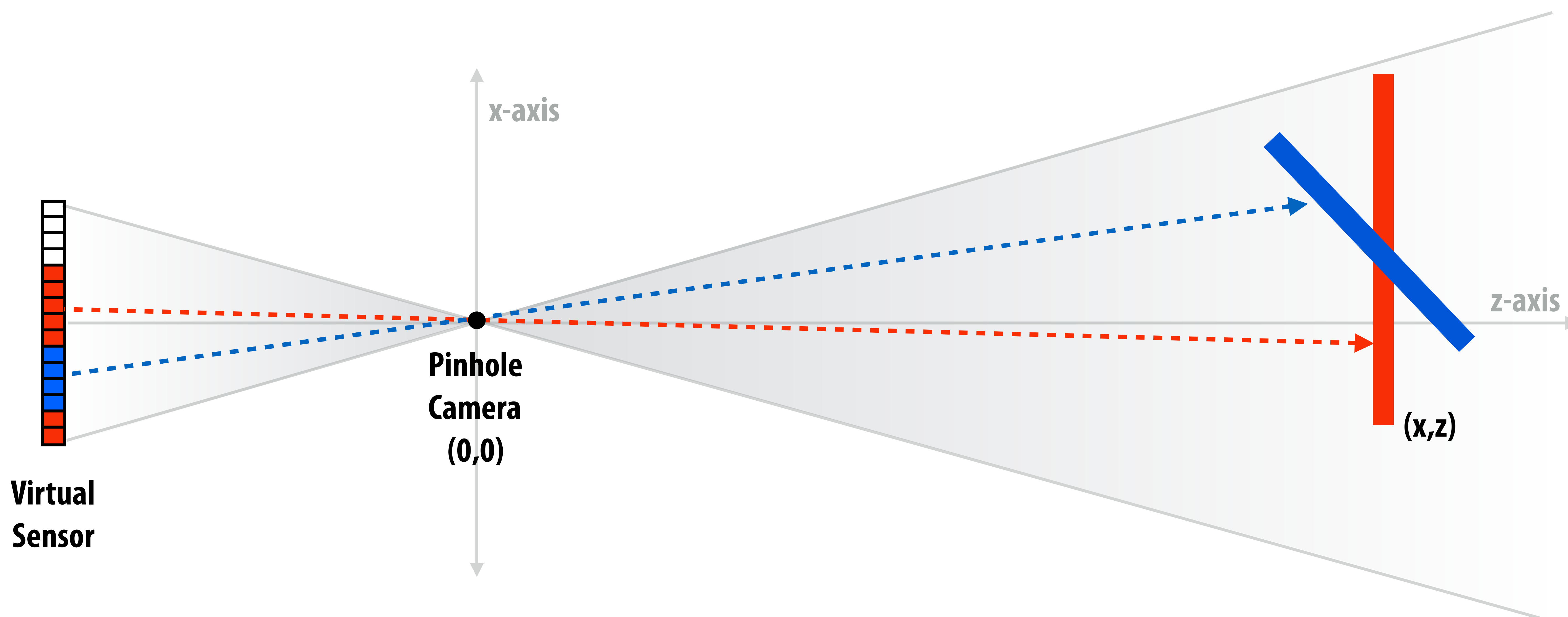
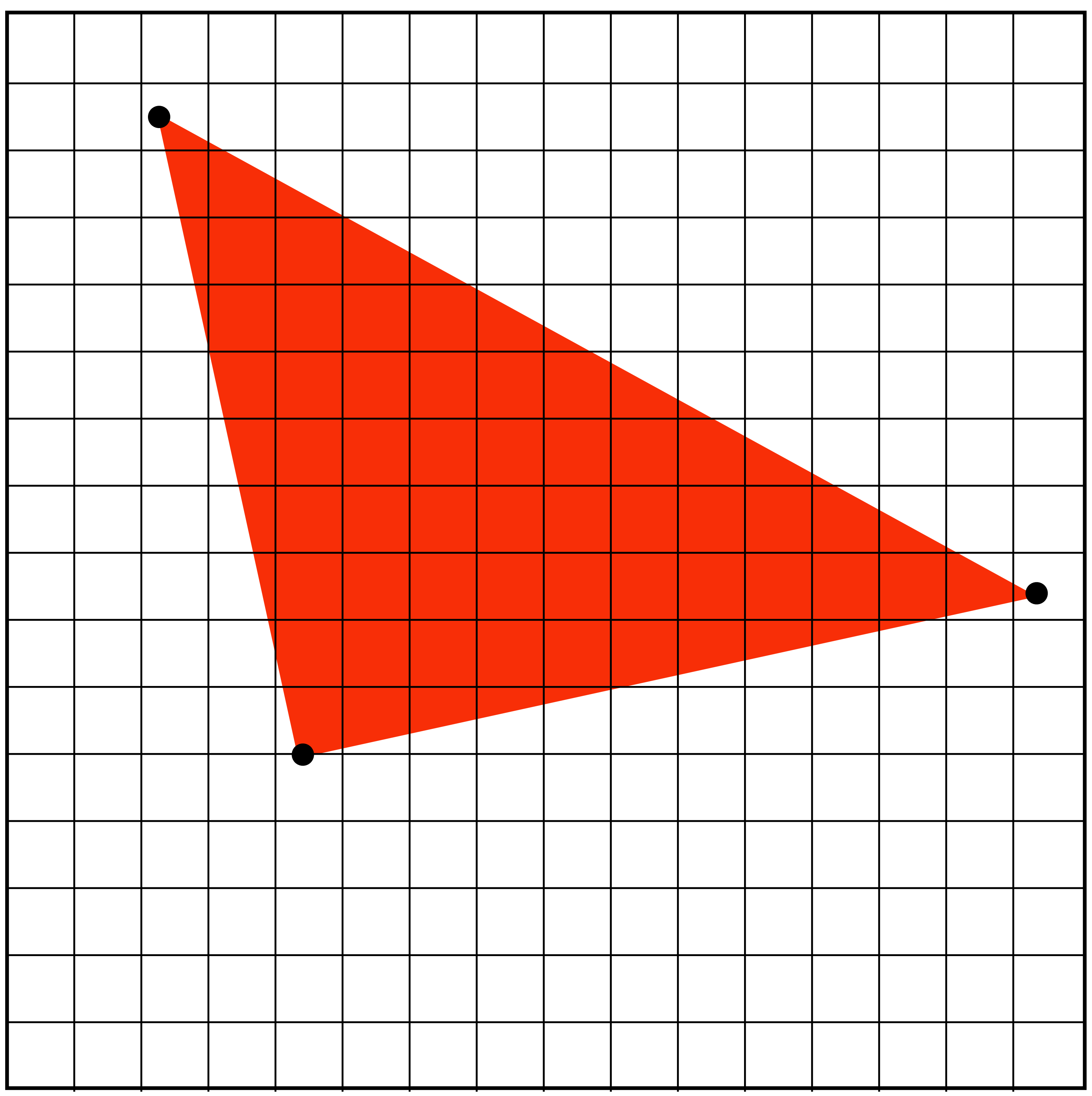
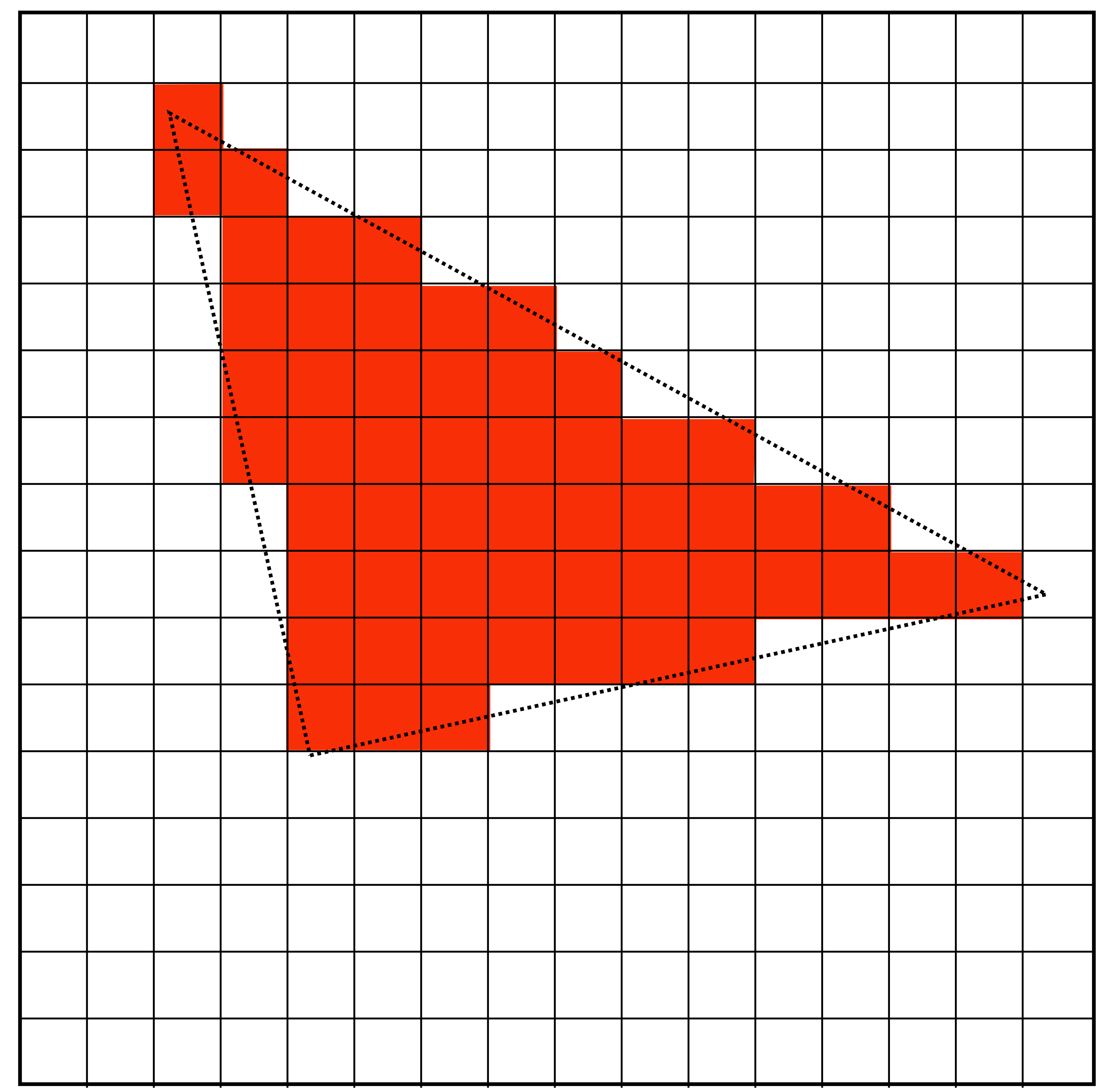


Image synthesis by the graphics pipeline

Input:
projected position of triangle vertices: P_0, P_1, P_2

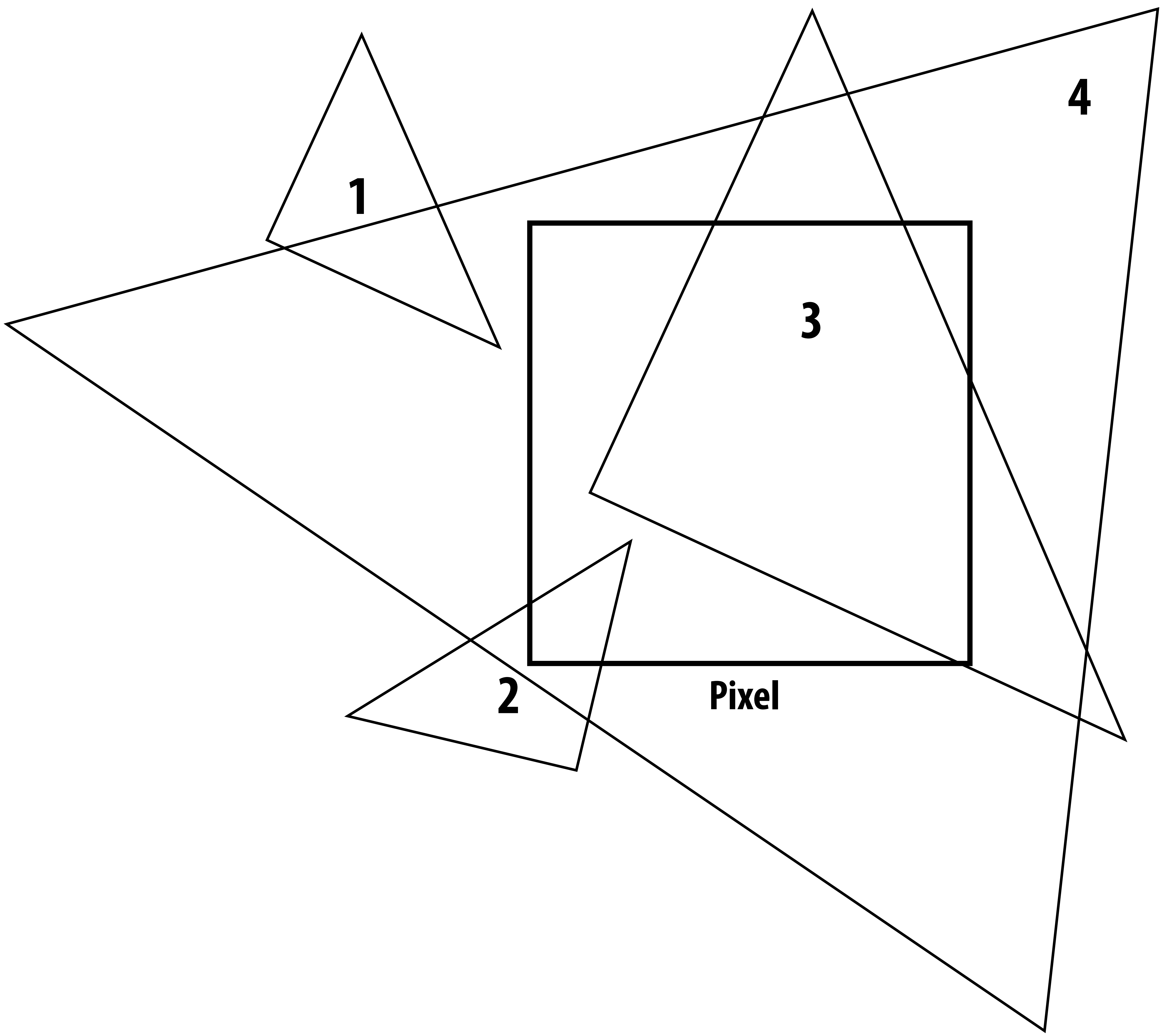


Output:
set of pixels "covered" by the triangle

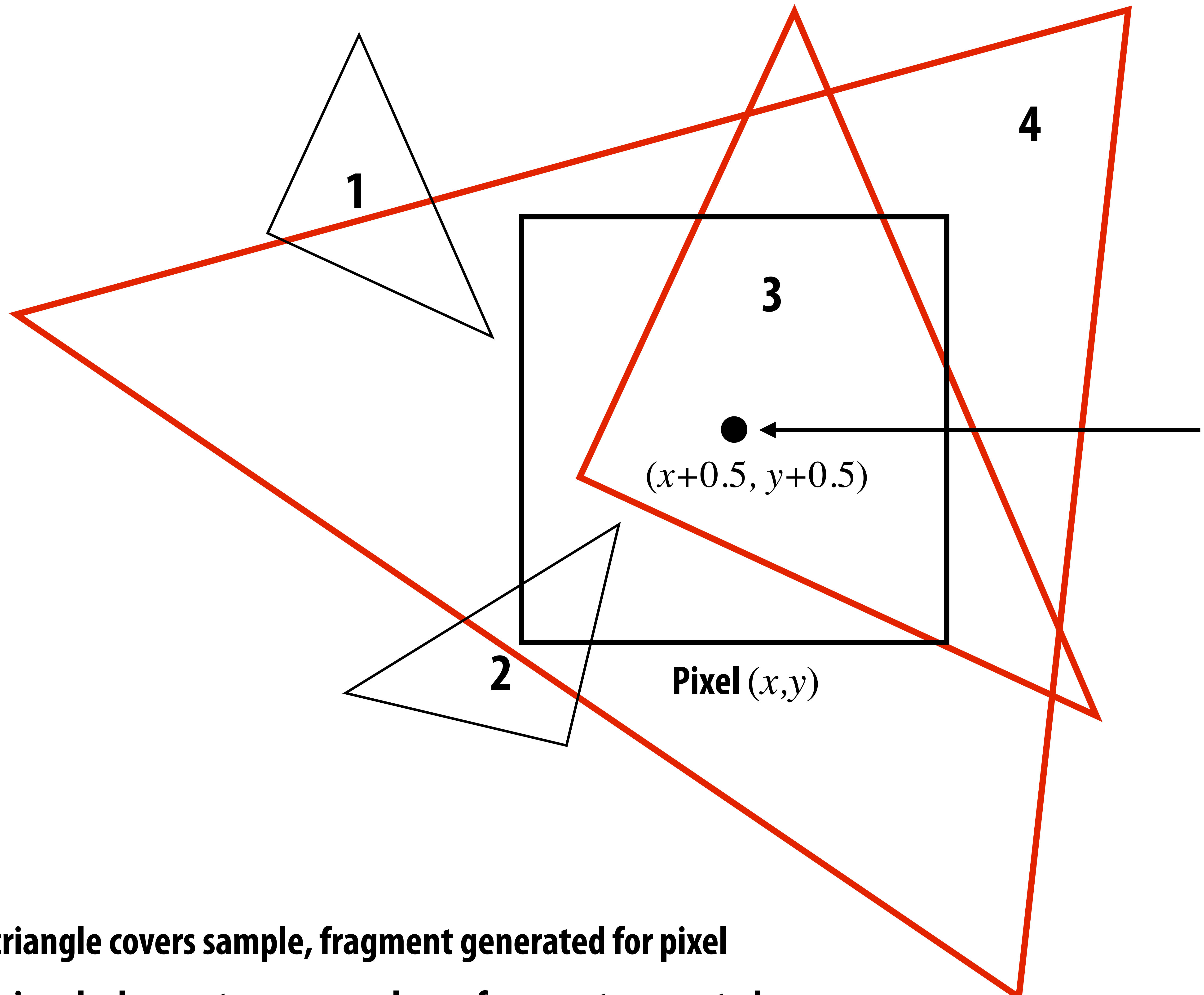


What does it mean for a pixel to be covered by a triangle?

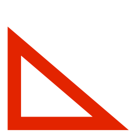
Question: which triangles "cover" this pixel?

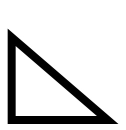


Estimate triangle-screen coverage by sampling the binary function: `coverage(x,y)`



Example:
Here I chose the coverage sample point to be at a point corresponding to the pixel center.


= triangle covers sample, fragment generated for pixel


= triangle does not cover sample, no fragment generated

For this lecture

The graphics pipeline

Geometry:
Compute vertex positions on screen

↓

Rasterization:
compute covered samples

↓

Shading:
compute color of colored pixels

↓

Pixel Ops:
Depth Test and Depth/Color Write

For this lecture

- Assume a triangle is represented as 3 points in 2D screen coordinates + depth from camera

$$P_0 = \begin{bmatrix} x_0 & y_0 & d_0 \end{bmatrix}^T$$
$$P_1 = \begin{bmatrix} x_1 & y_1 & d_1 \end{bmatrix}^T$$
$$P_2 = \begin{bmatrix} x_2 & y_2 & d_2 \end{bmatrix}^T$$

- Assume, for a given triangle we can evaluate the binary function coverage at any point on screen

```
coverage(x, y, p0, p1, p2)
```

- We can also evaluate depth at any point on screen

```
depth(x, y, p0, p1, p2)
```

Computing coverage(x,y): point-in-triangle test

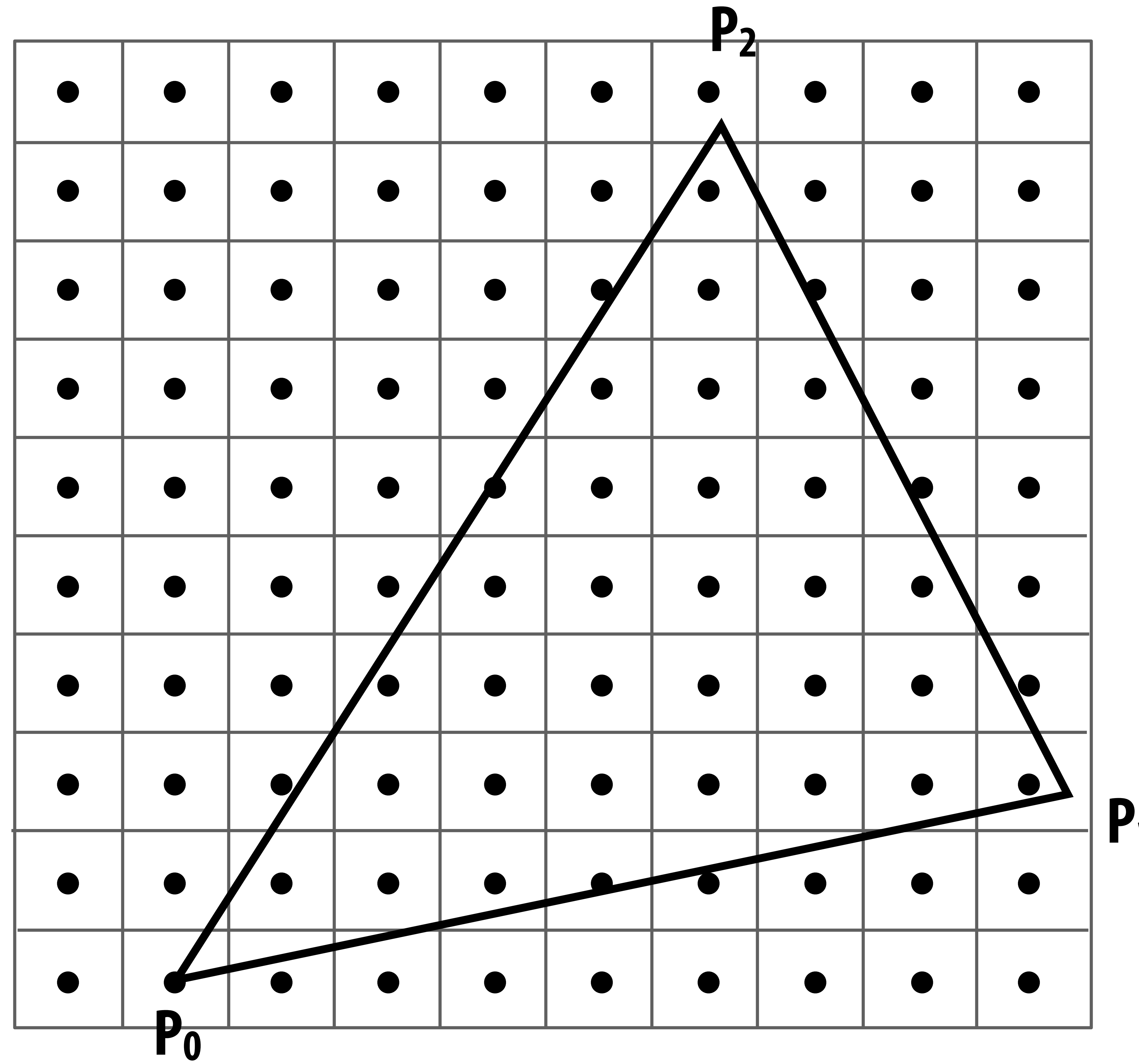
Compute triangle edge equations from projected positions of vertices

$P_i = (X_i, Y_i)$

$dX_i = X_{i+1} - X_i$
$dY_i = Y_{i+1} - Y_i$

$$E_i(x, y) = (x - X_i)\, dY_i - (y - Y_i)\, dX_i$$
$$= A_i x + B_i y + C_i$$

$E_i(x, y) = 0$: point on edge
> 0 : outside edge
< 0 : inside edge



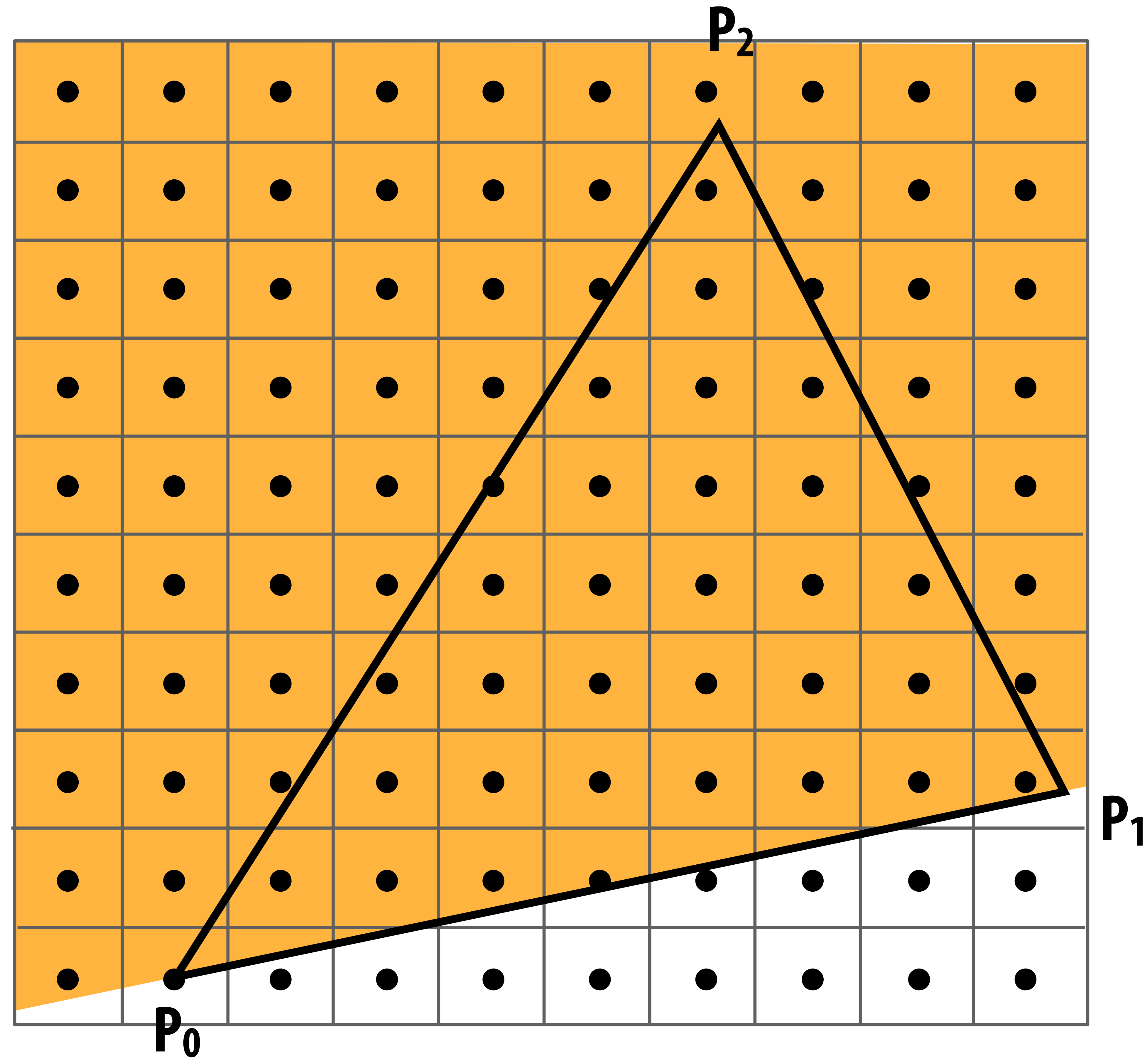
Point-in-triangle test

$P_i = (X_i, Y_i)$

$dX_i = X_{i+1} - X_i$
$dY_i = Y_{i+1} - Y_i$

$$E_i(x, y) = (x - X_i)\, dY_i - (y - Y_i)\, dX_i$$
$$= A_i x + B_i y + C_i$$

$E_i(x, y) = 0$: point on edge
> 0 : outside edge
< 0 : inside edge



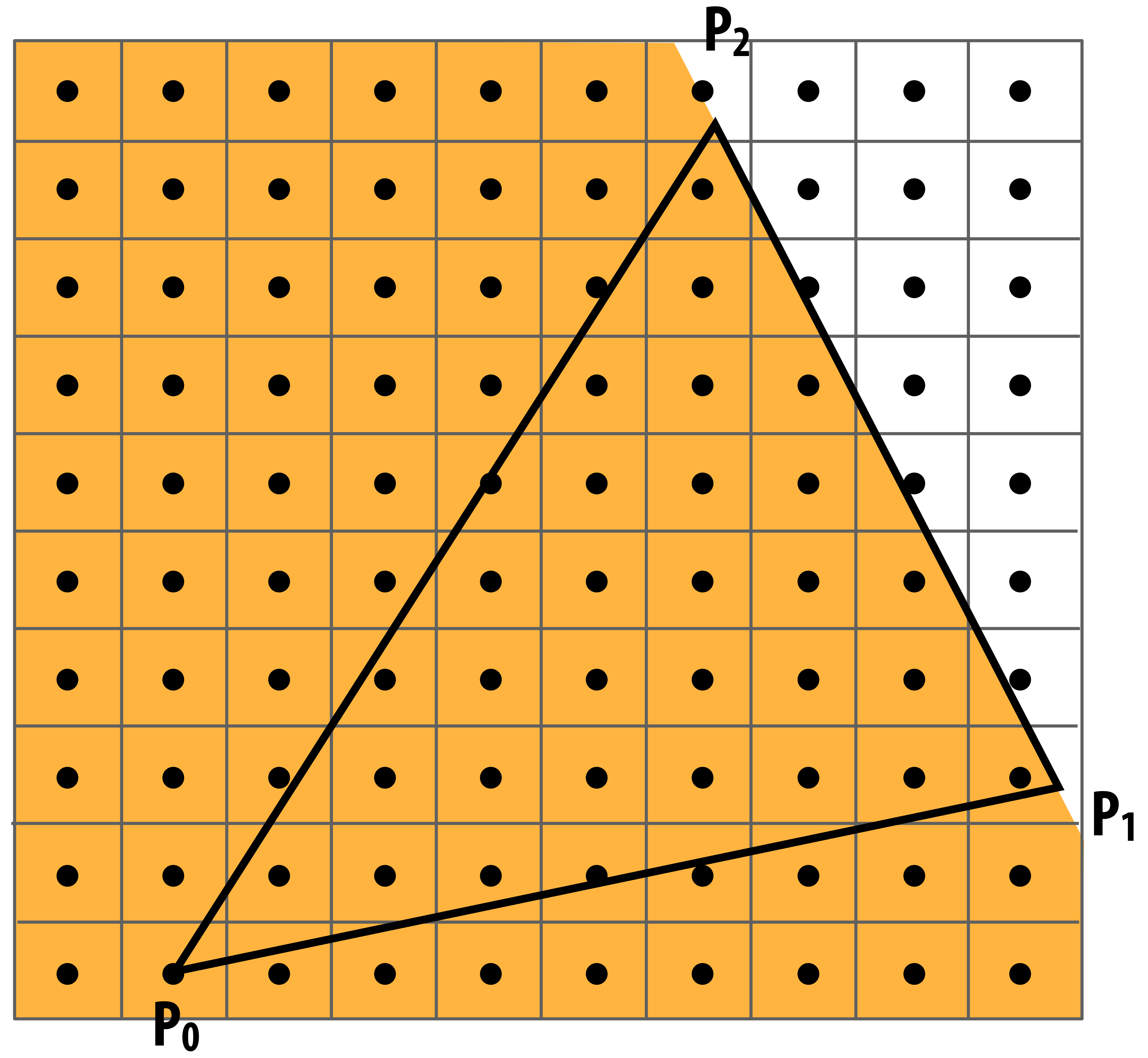
Point-in-triangle test

$P_i = (X_i, Y_i)$

$dX_i = X_{i+1} - X_i$
$dY_i = Y_{i+1} - Y_i$

$$E_i(x, y) = (x - X_i)\, dY_i - (y - Y_i)\, dX_i$$
$$= A_i\, x + B_i\, y + C_i$$

$E_i(x, y) = 0$: point on edge
> 0 : outside edge
< 0 : inside edge



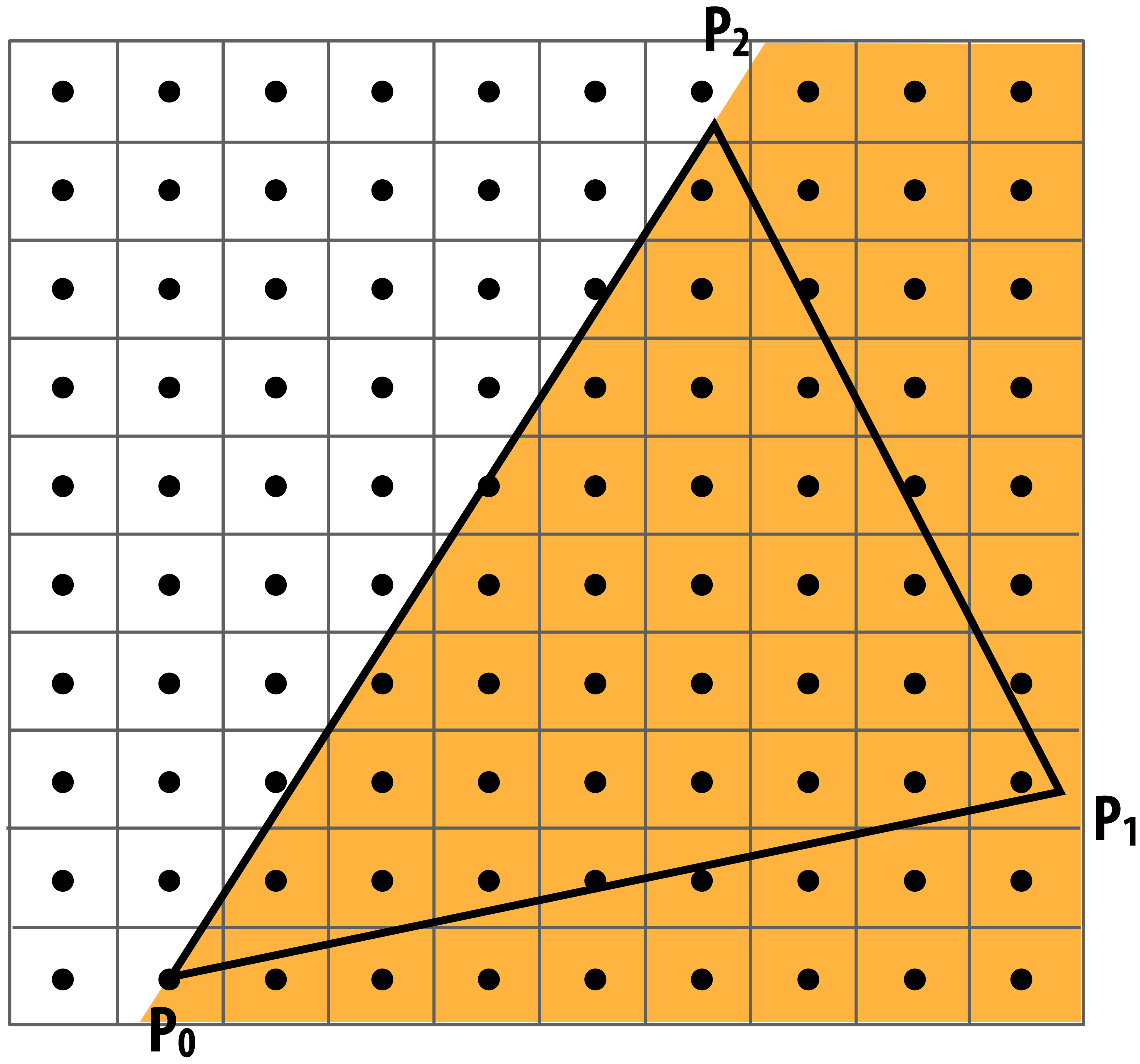
Point-in-triangle test

$P_i = (X_i, Y_i)$

$dX_i = X_{i+1} - X_i$
$dY_i = Y_{i+1} - Y_i$

$$E_i(x, y) = (x - X_i)\, dY_i - (y - Y_i)\, dX_i$$
$$= A_i\, x + B_i\, y + C_i$$

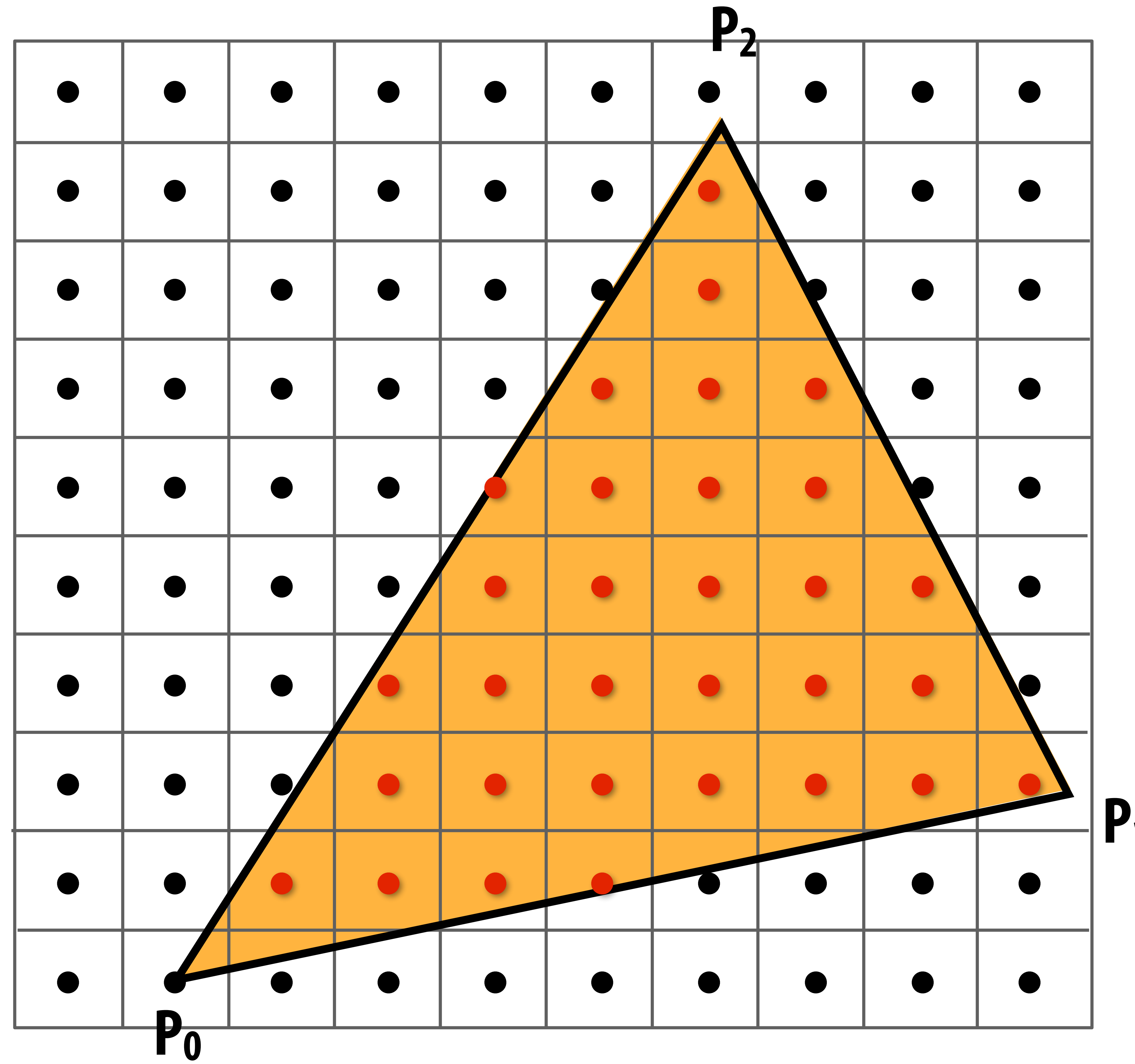
$E_i(x, y) = 0$: point on edge
> 0 : outside edge
< 0 : inside edge



Point-in-triangle test

Sample point $s = (sx, sy)$ is inside the triangle if it is inside all three edges.

$$inside(sx, sy) =$$
$$E_0\,(sx, sy) < 0 \;\&\&$$
$$E_1\,(sx, sy) < 0 \;\&\&$$
$$E_2\,(sx, sy) < 0;$$



Sample points inside triangle are highlighted red.

One option: incremental triangle traversal (work efficient!)

$P_i = (X_i, Y_i)$

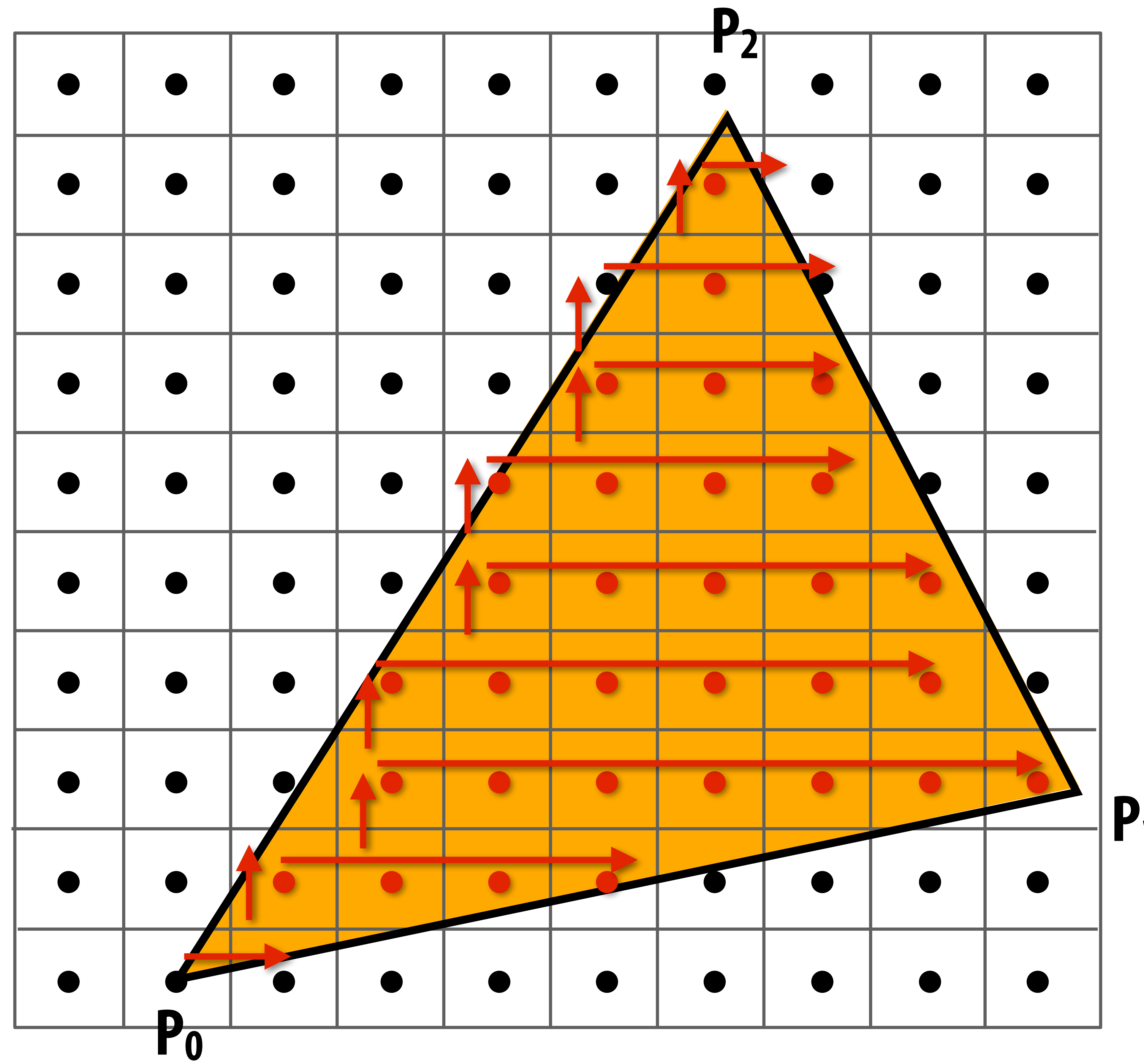
$dX_i = X_{i+1} - X_i$
$dY_i = Y_{i+1} - Y_i$

$$E_i(x, y) = (x - X_i)\, dY_i - (y - Y_i)\, dX_i = A_i x + B_i y + C_i$$

$E_i(x, y) = 0$: point on edge
> 0 : outside edge
< 0 : inside edge

Efficient incremental update:

$$dE_i(x+1,y) = E_i(x,y) + dY_i = E_i(x,y) + A_i$$
$$dE_i(x,y+1) = E_i(x,y) + dX_i = E_i(x,y) + B_i$$



Incremental update saves computation:
Only one addition per edge, per sample test

Many traversal orders are possible: backtrack, zig-zag, Hilbert/Morton curves (locality maximizing)

Modern approach: tiled triangle traversal

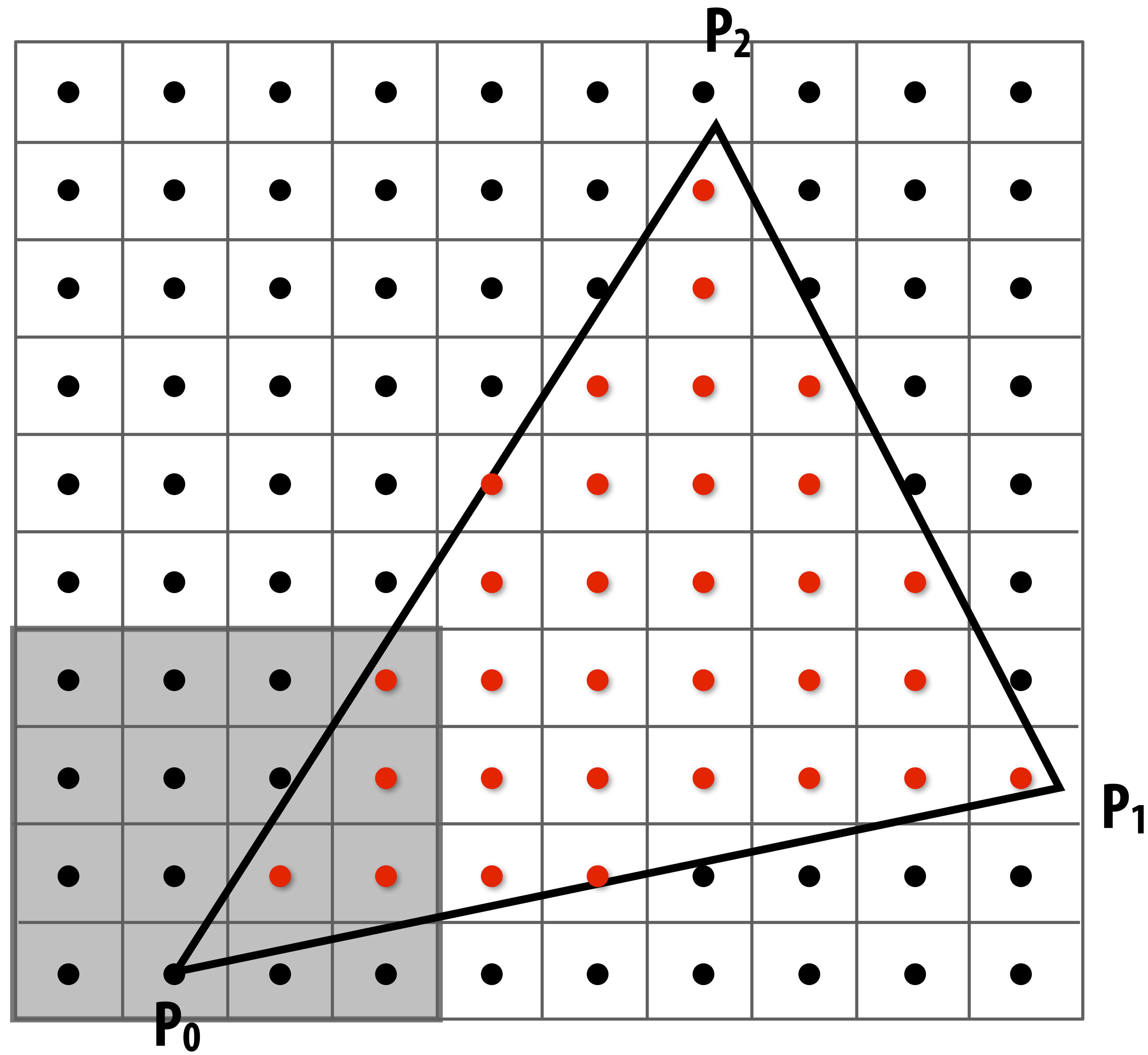
All modern GPUs have fixed-function hardware for efficiently performing data-parallel point-in-triangle tests

Traverse triangle in blocks

Test all samples in block against triangle in parallel

Advantages:

- Simplicity of wide parallel execution overcomes cost of extra point-in-triangle tests (most triangles cover many samples, especially when super-sampling coverage)
- Can skip sample testing work: use triangle-box test to classify entire block not in triangle ("early out test"), or entire block entirely within triangle ("early in")



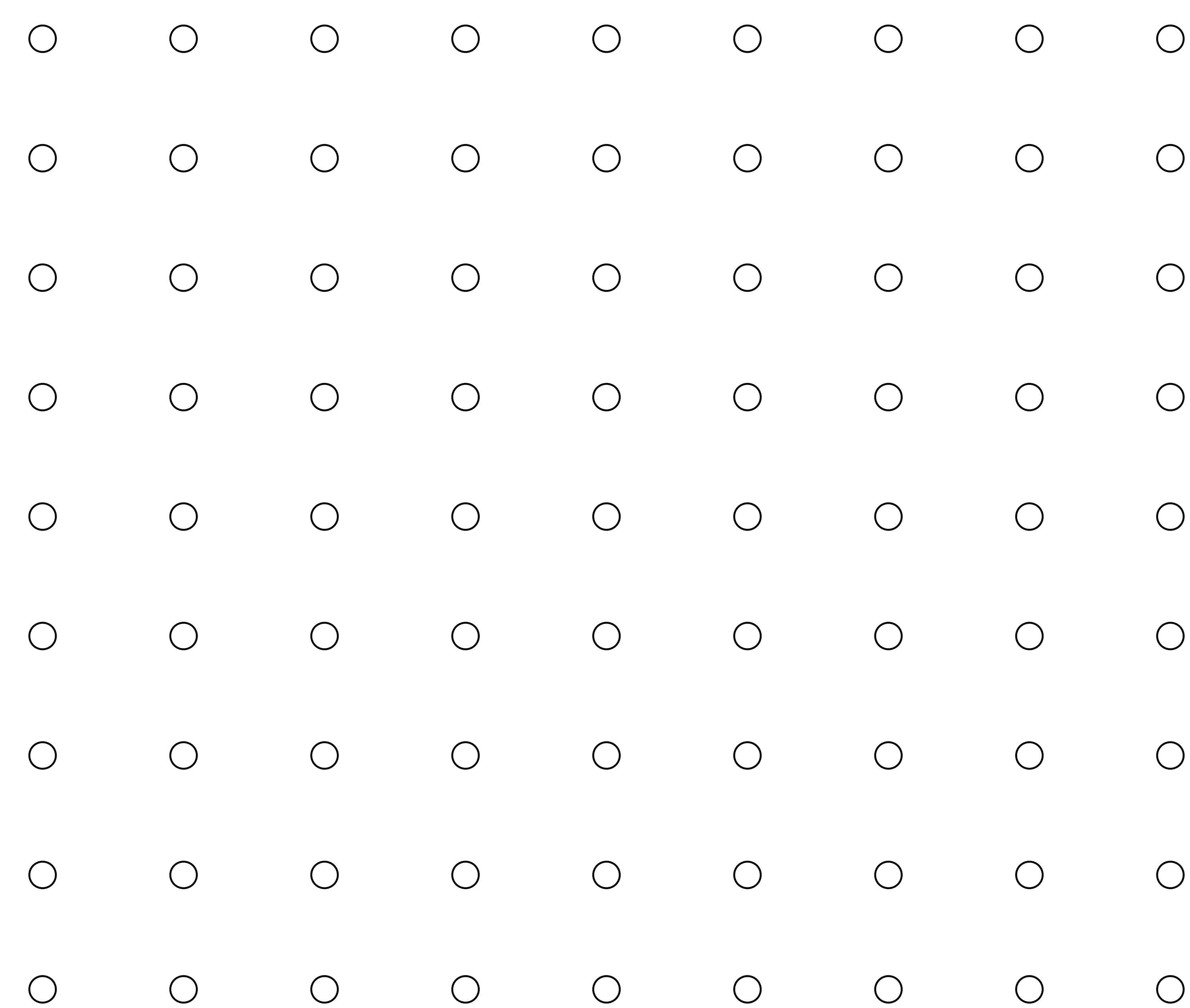
Occlusion

Occlusion using the depth-buffer (Z-buffer)

For each coverage sample point, depth-buffer stores depth of closest triangle at this sample point that has been processed by the renderer so far.

Closest triangle at sample point (x,y) is triangle with minimum depth at (x,y)

Initial state of depth buffer before rendering any triangles (all samples store farthest distance)



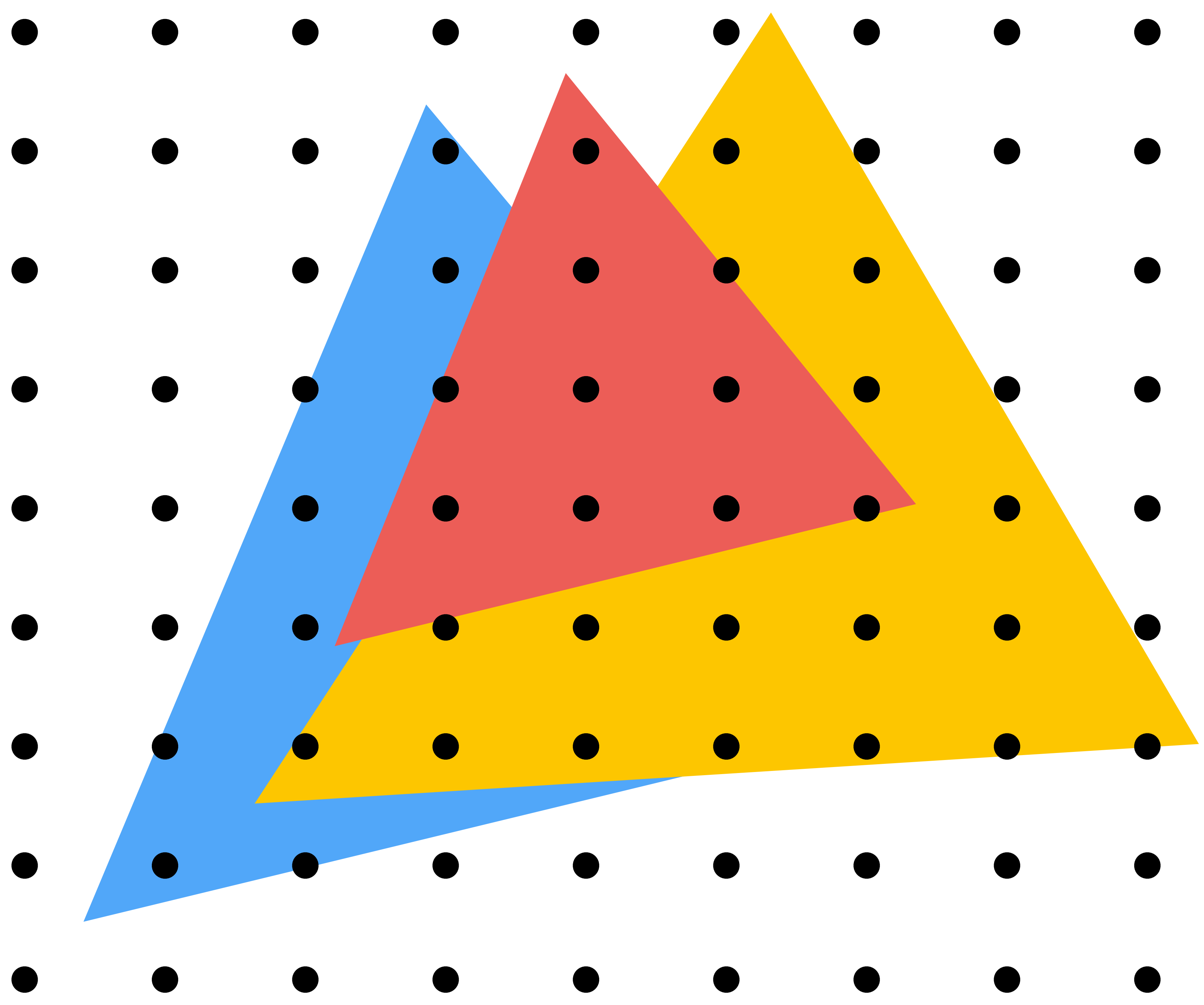
Grayscale value of sample point used to indicate distance

Black = small distance

White = large distance

Depth buffer example

Example: rendering three opaque triangles



Occlusion using the depth-buffer (Z-buffer)

Processing yellow triangle:

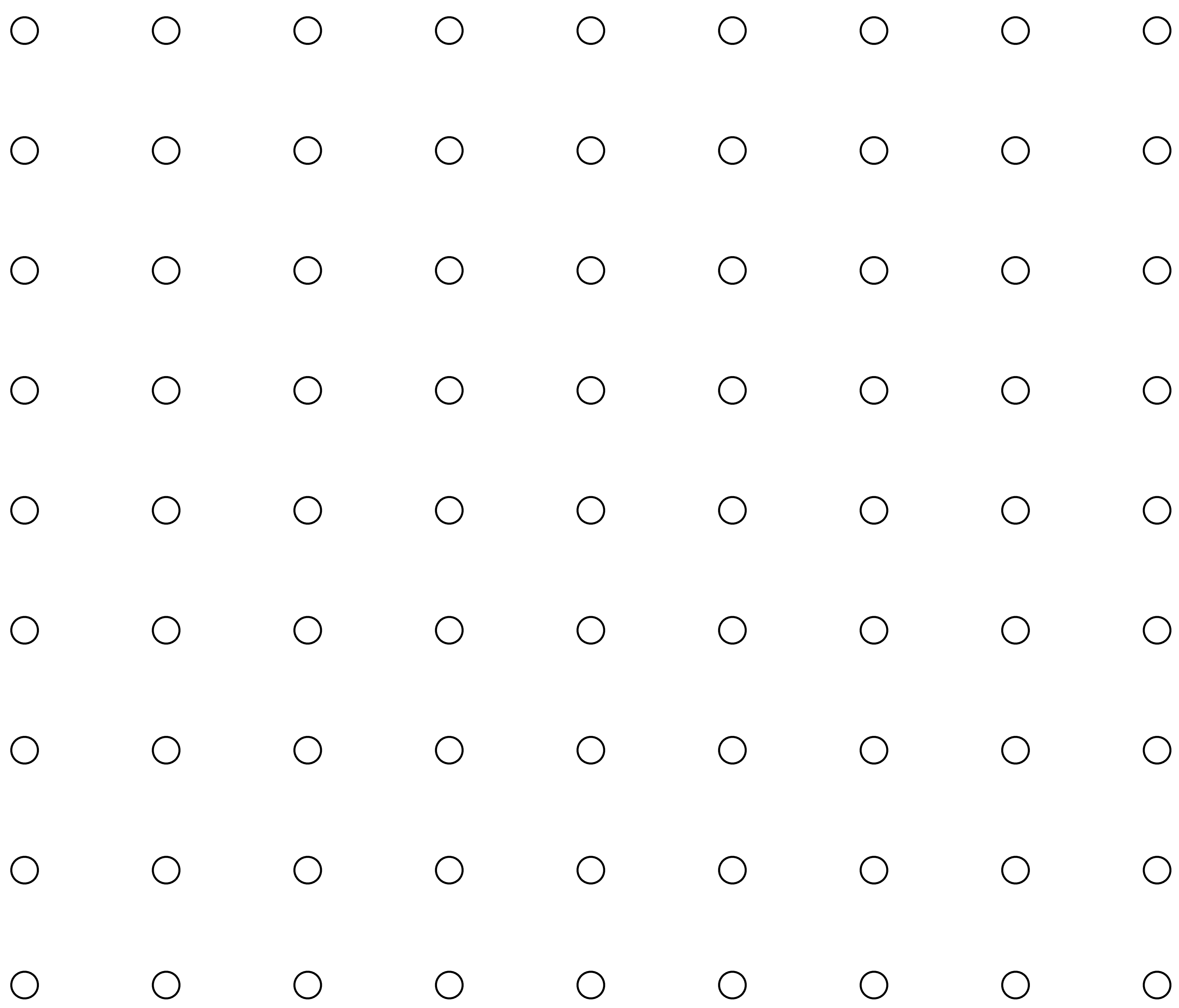
depth = 0.5

Grayscale value of sample point used to indicate distance

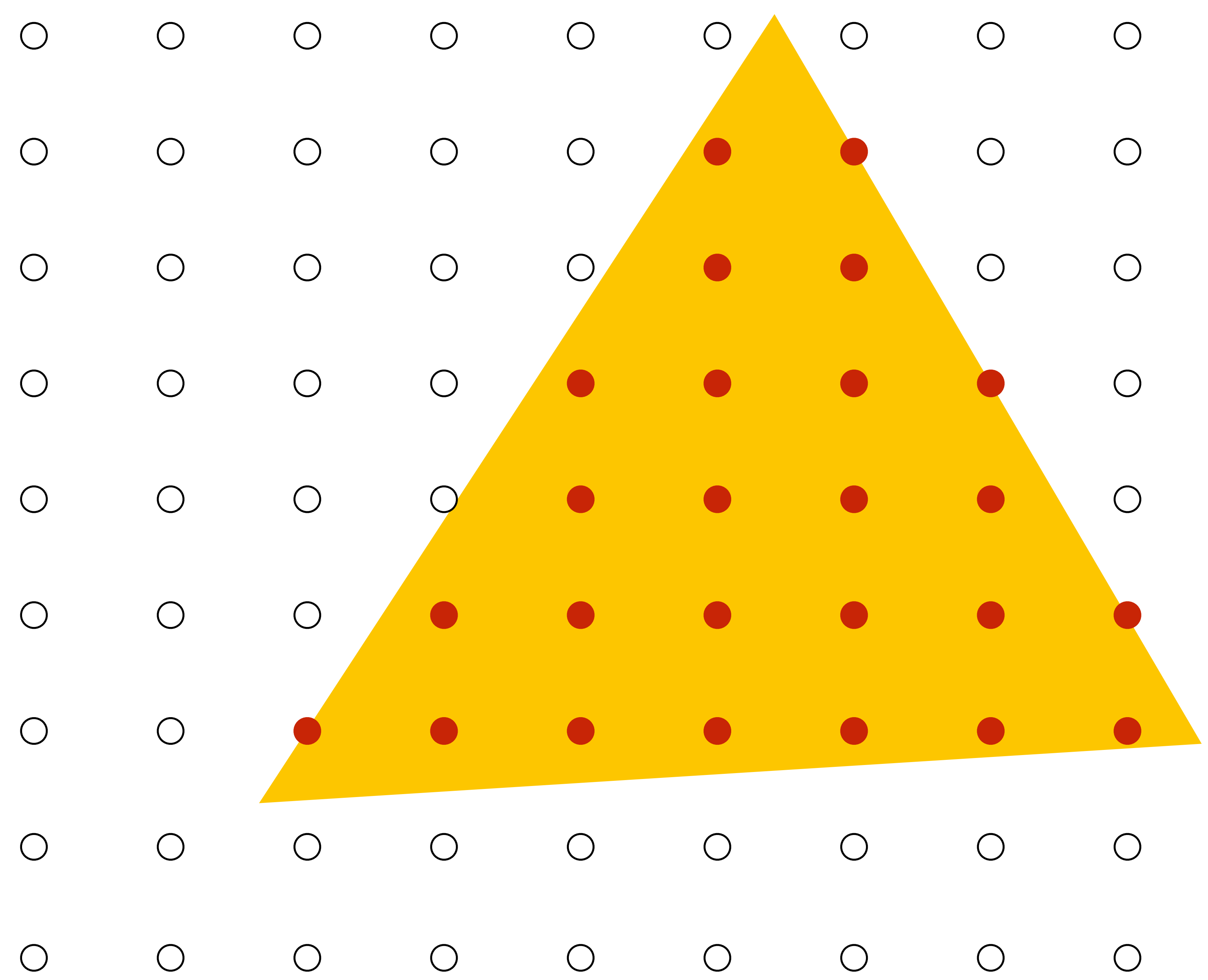
White = large distance

Black = small distance

Red = sample passed depth test



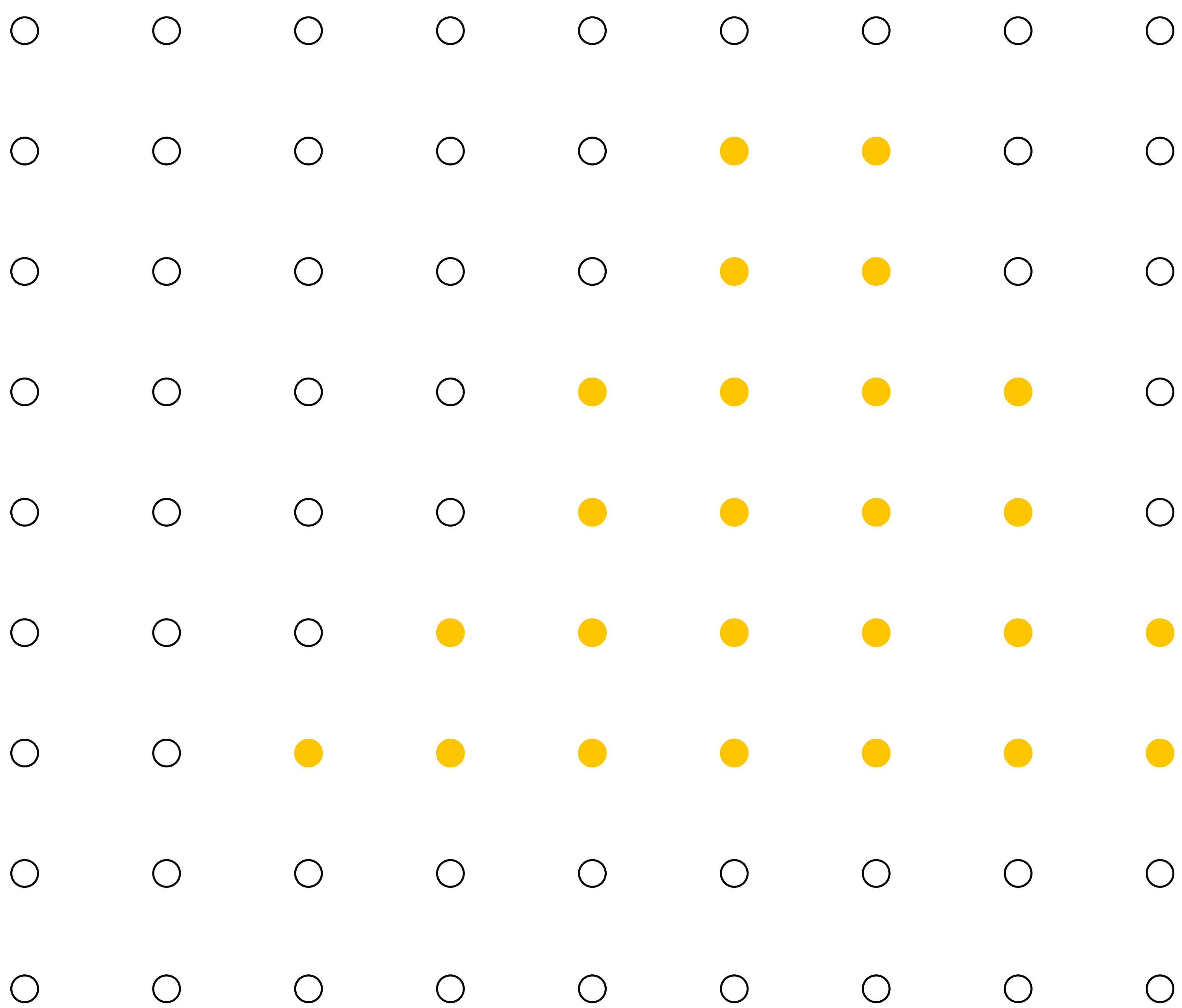
Color buffer contents



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

After processing yellow triangle:



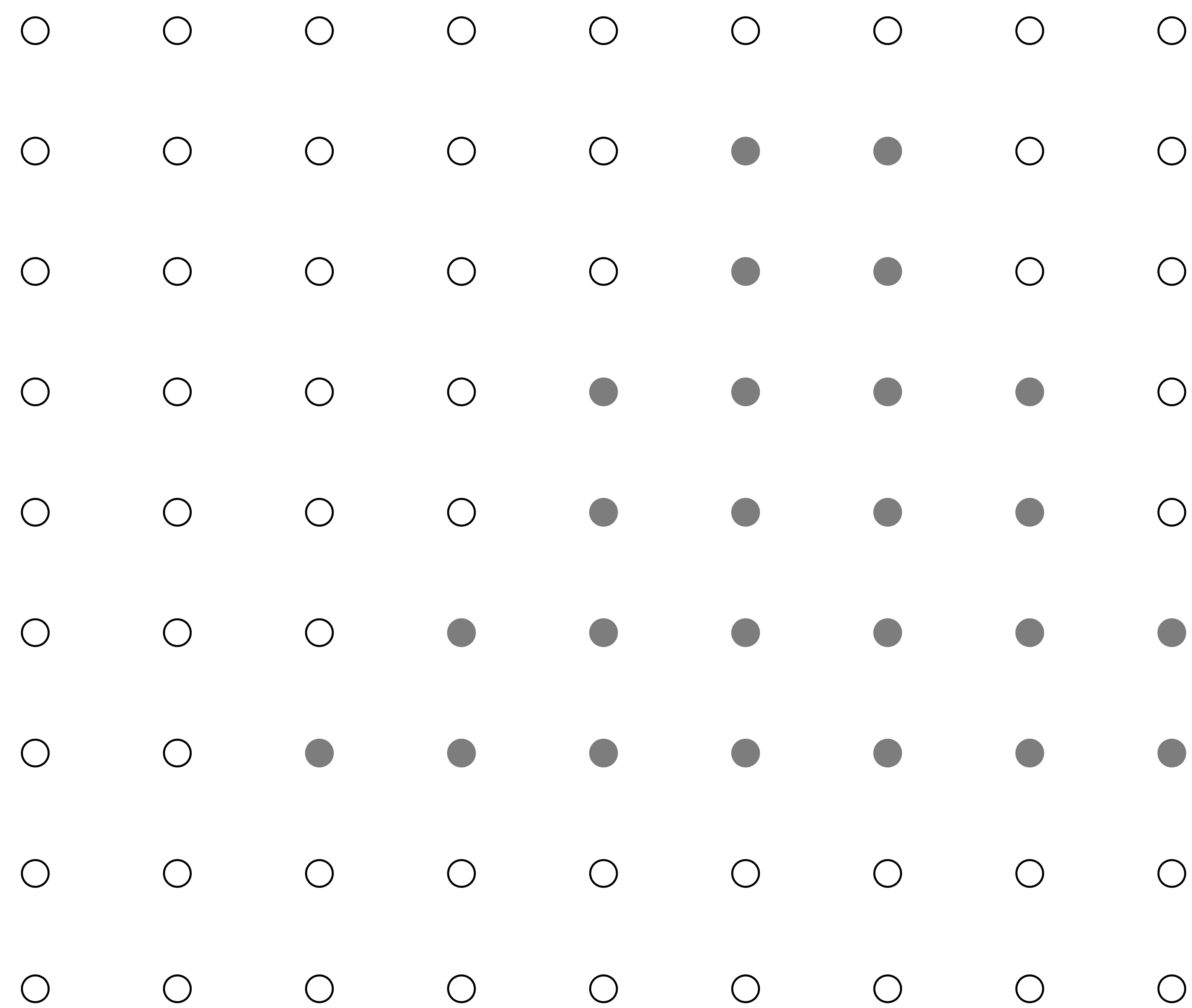
Color buffer contents

Grayscale value of sample point used to indicate distance

White = large distance

Black = small distance

Red = sample passed depth test



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

Processing blue triangle:

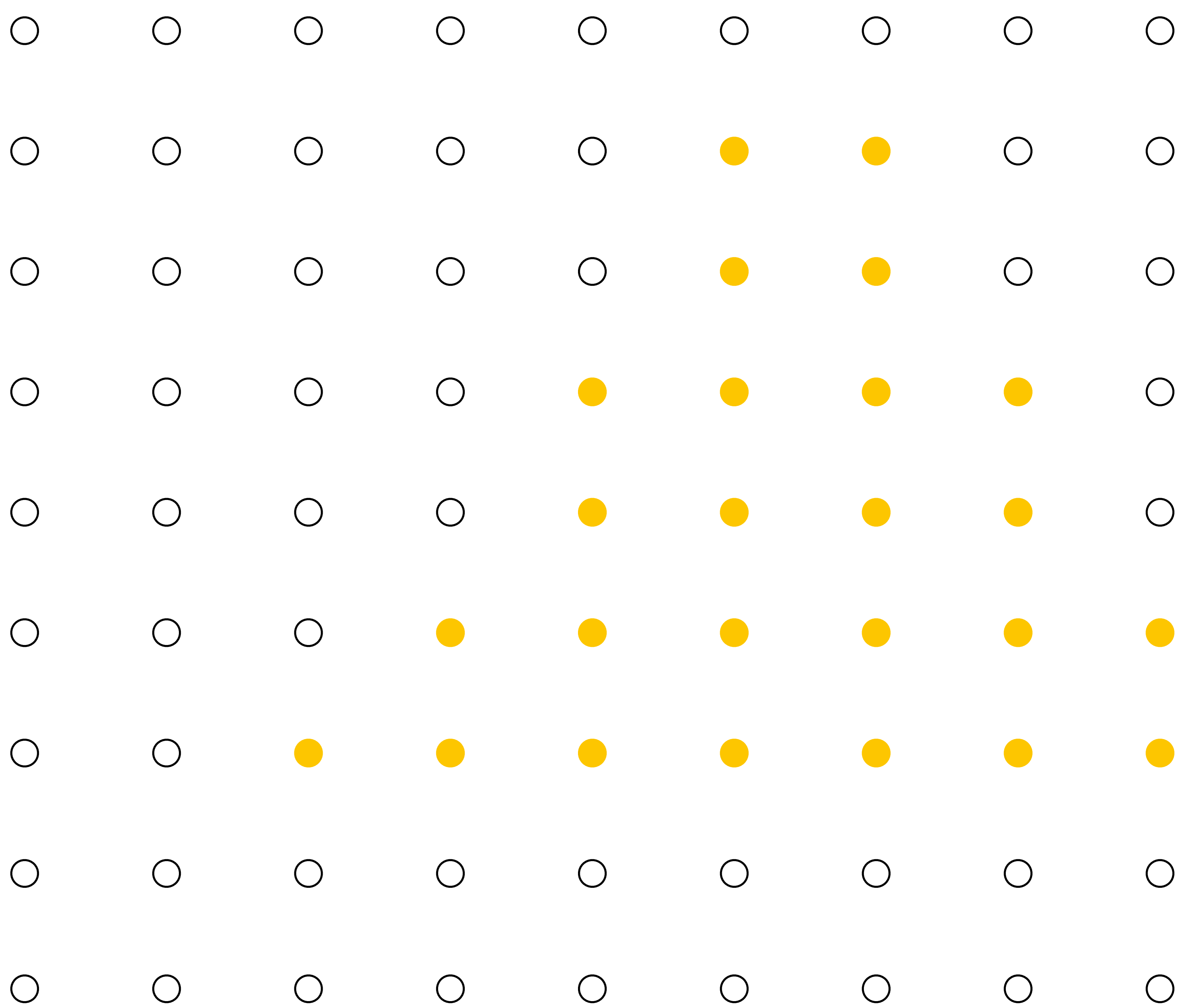
depth = 0.75

Grayscale value of sample point used to indicate distance

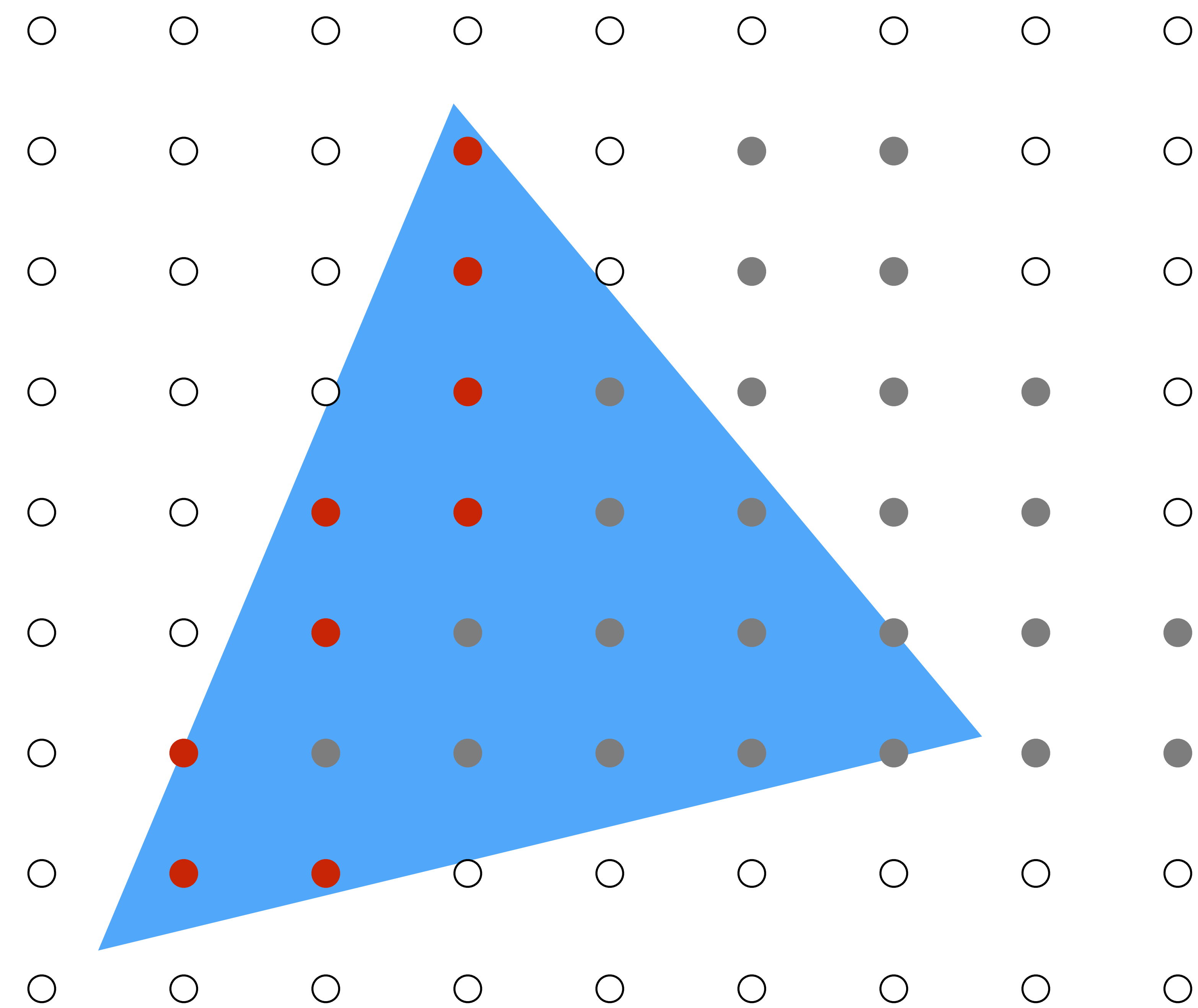
White = large distance

Black = small distance

Red = sample passed depth test



Color buffer contents



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

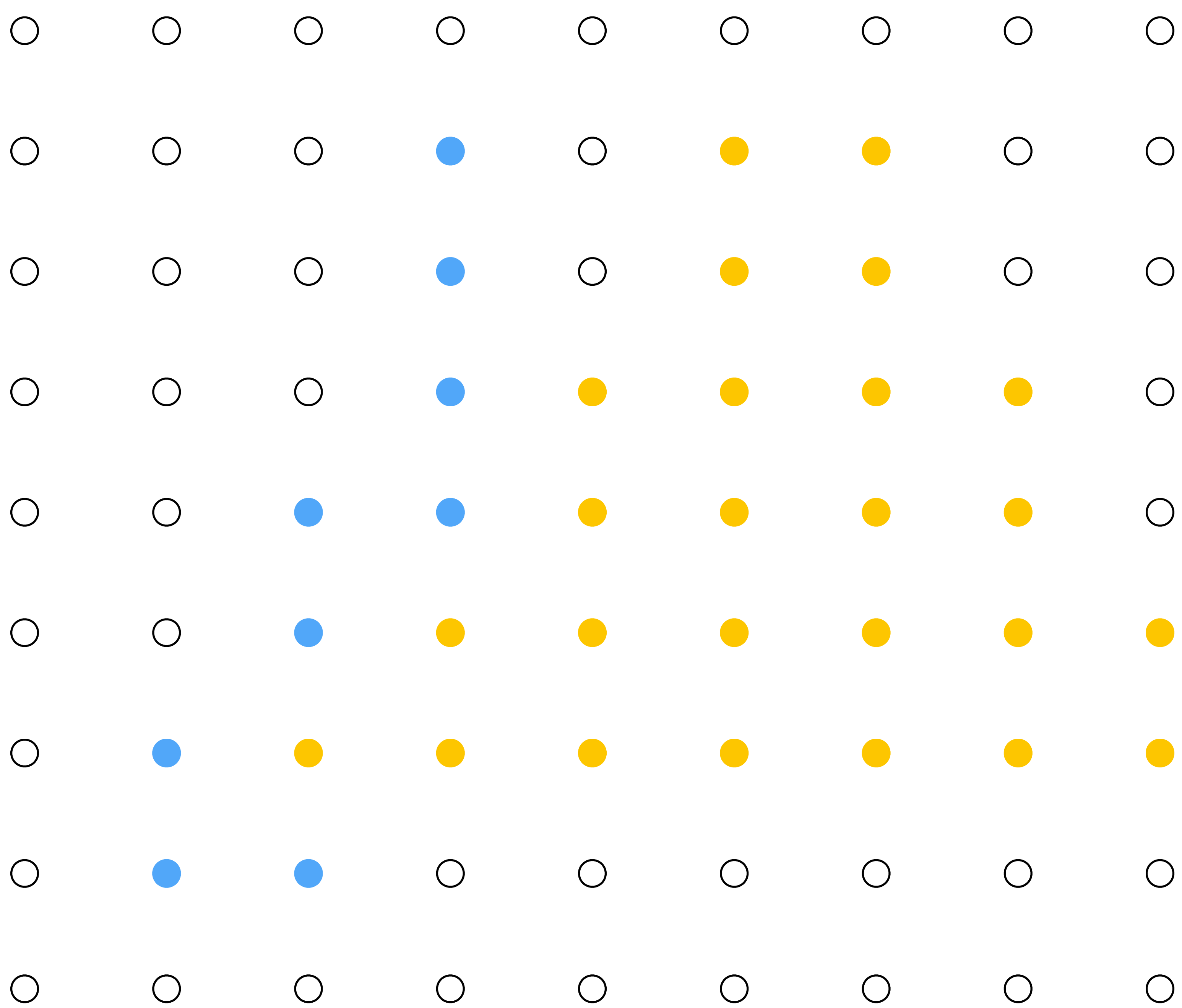
After processing blue triangle:

Grayscale value of sample point used to indicate distance

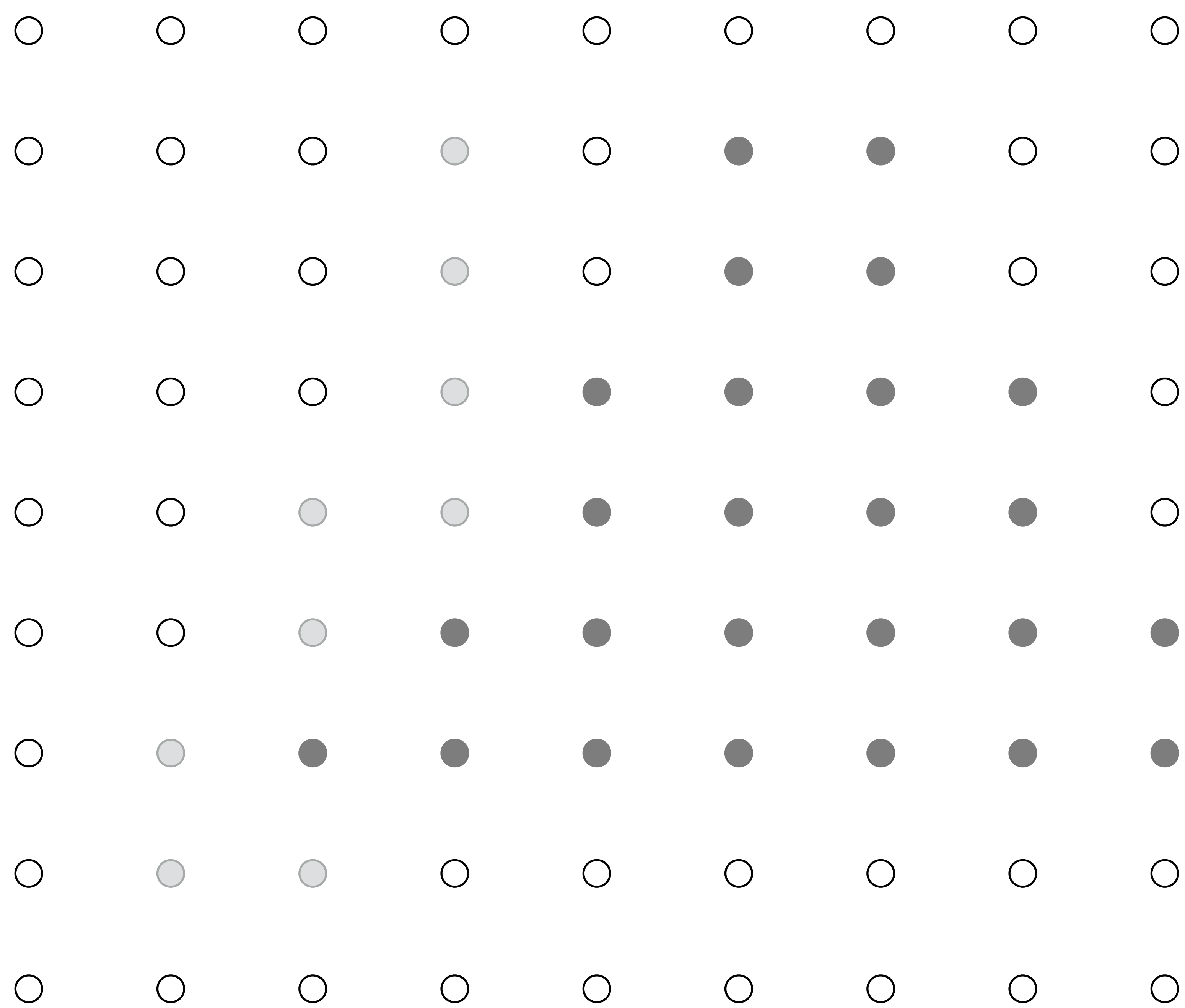
White = large distance

Black = small distance

Red = sample passed depth test



Color buffer contents

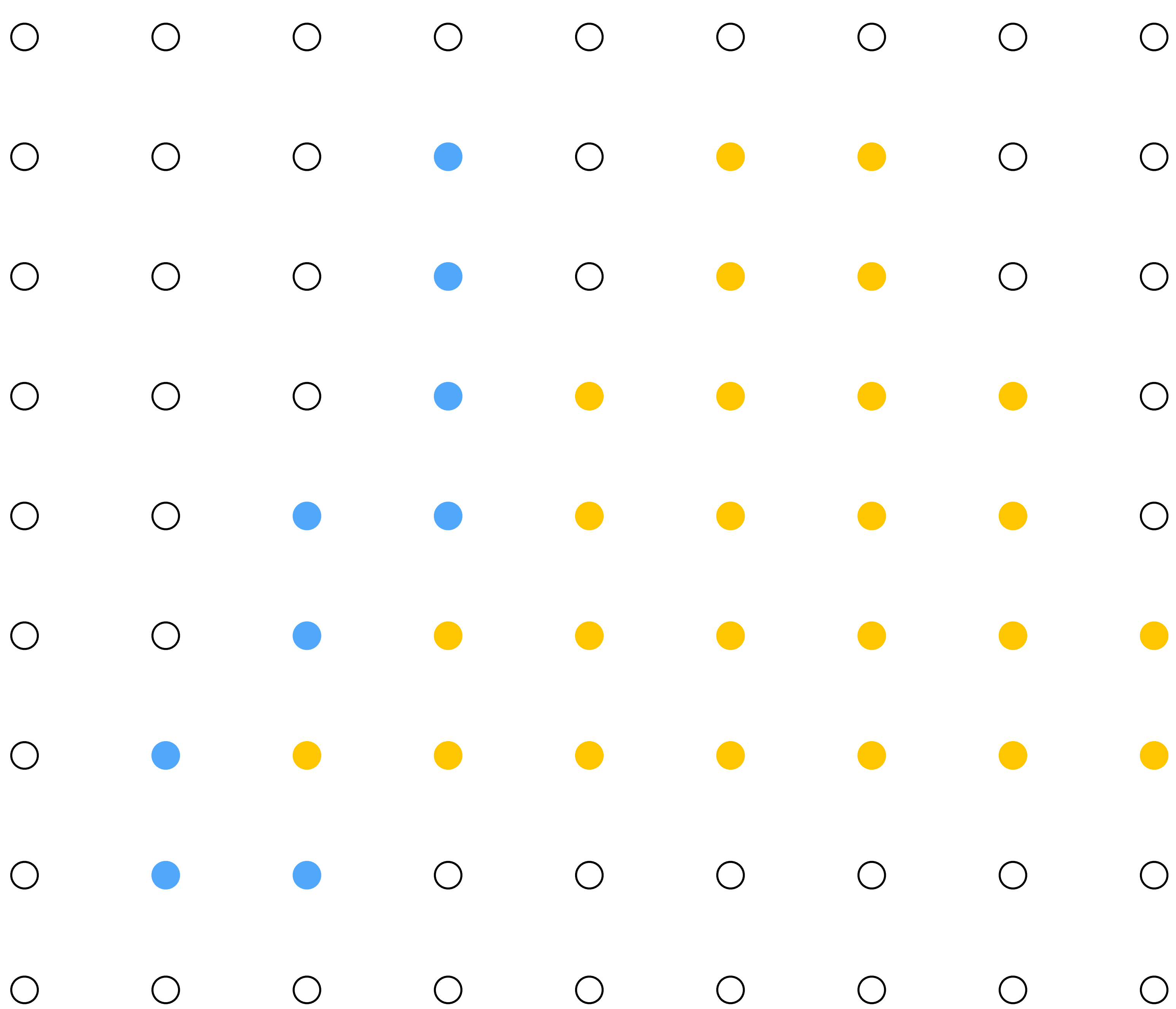


Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

Processing red triangle:

depth = 0.25



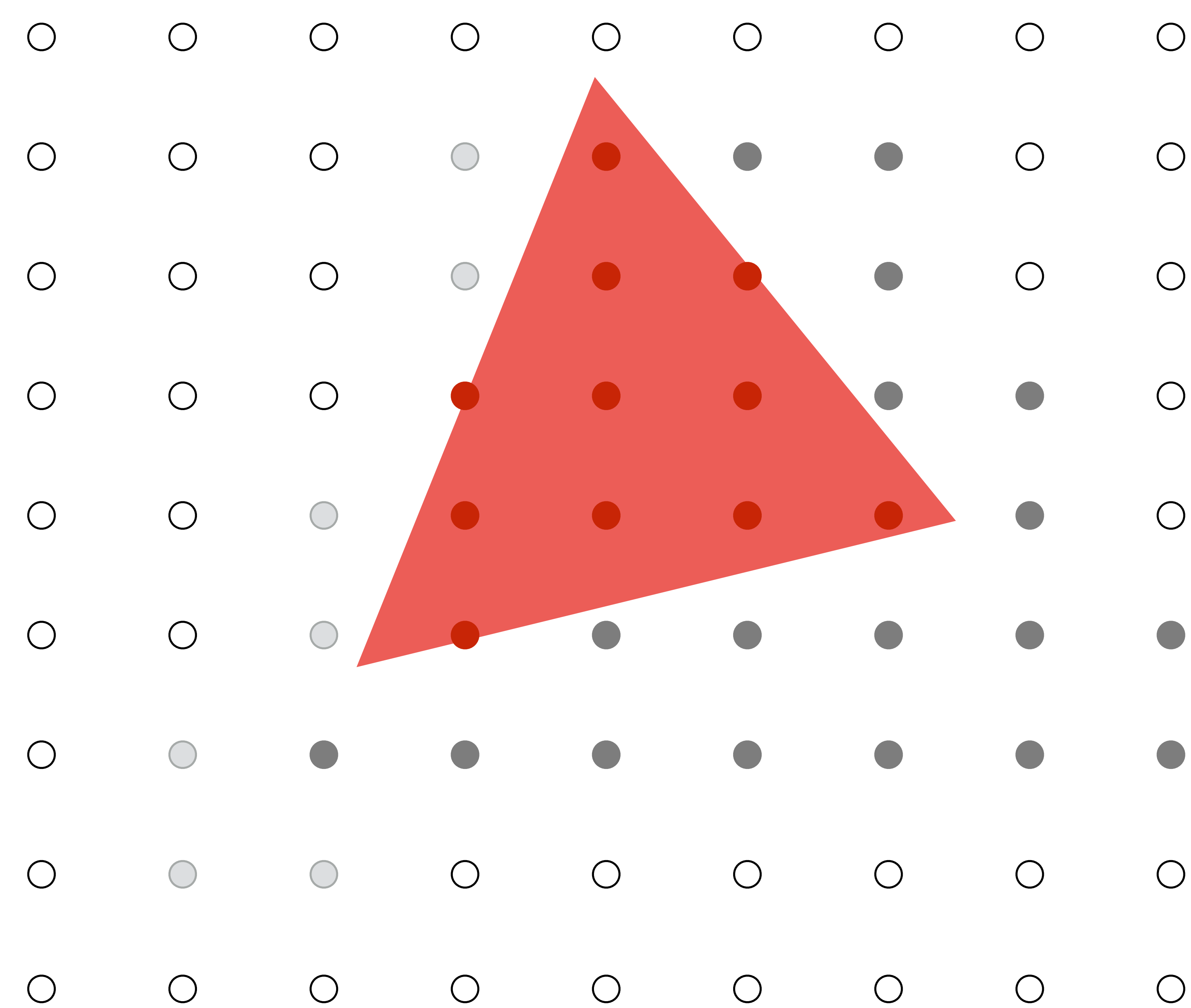
Color buffer contents

Grayscale value of sample point used to indicate distance

White = large distance

Black = small distance

Red = sample passed depth test



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

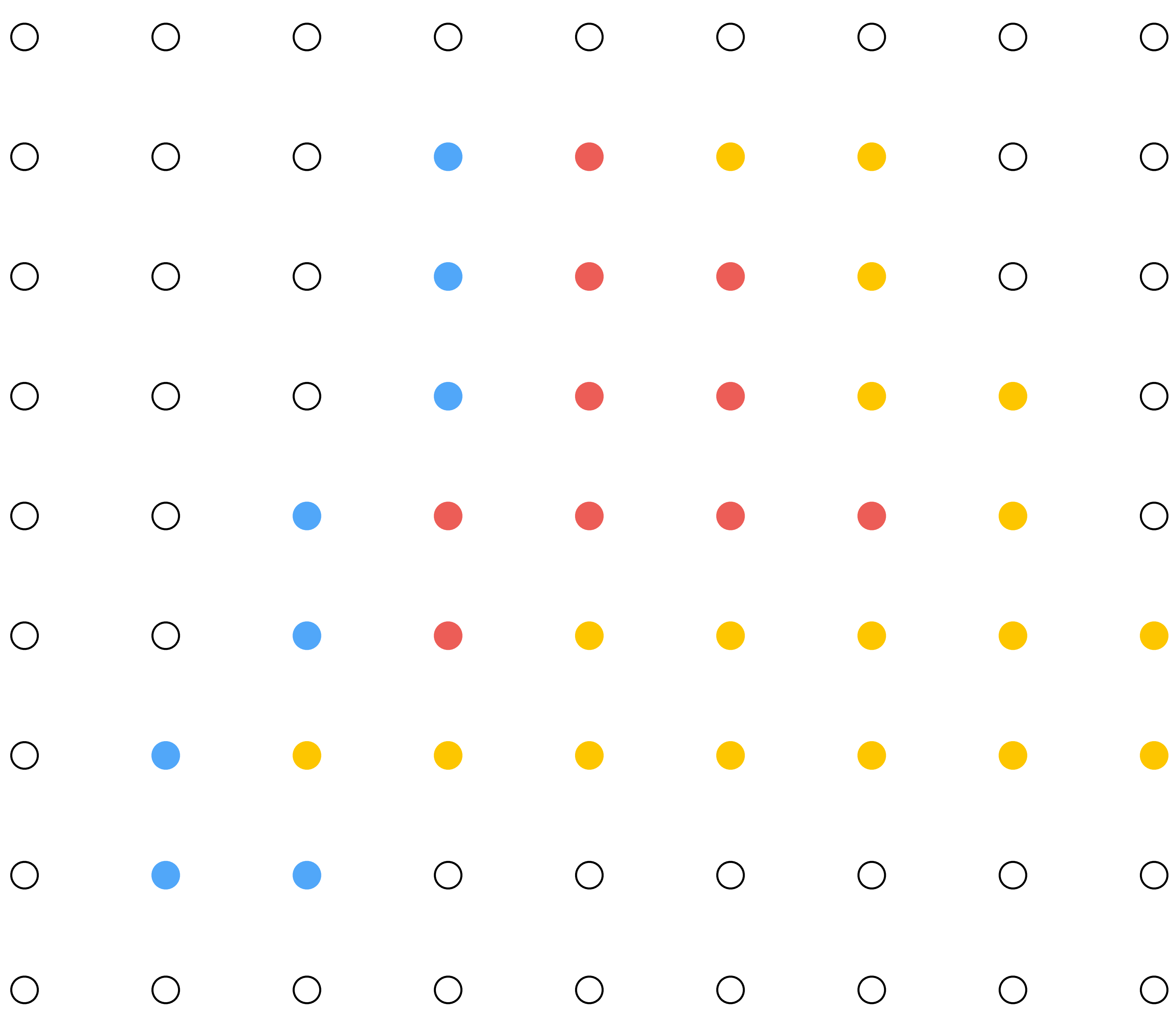
After processing red triangle:

Grayscale value of sample point used to indicate distance

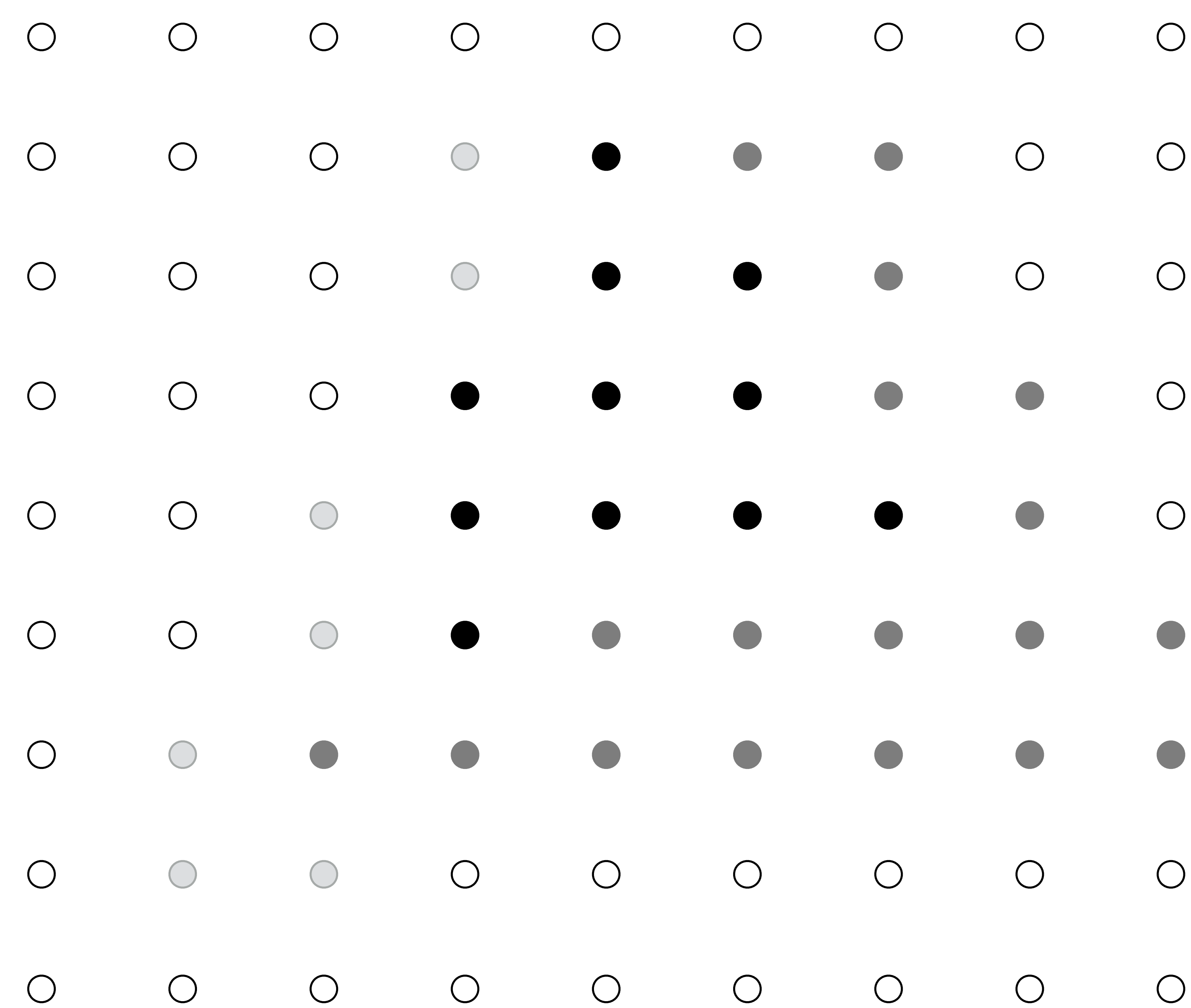
White = large distance

Black = small distance

Red = sample passed depth test



Color buffer contents



Depth buffer contents

Occlusion using the depth buffer

```
bool pass_depth_test(d1, d2) {
   return d1 < d2;
}


depth_test(tri_d, tri_color, x, y) {

  if (pass_depth_test(tri_d, zbuffer[x][y]) {

    // triangle is closest object seen so far at this
    // sample point. Update depth and color buffers.

    zbuffer[x][y] = tri_d;     // update zbuffer
    color[x][y] = tri_color;   // update color buffer
  }
}
```

Depth buffer for occlusion

- **Z-buffer algorithm has high bandwidth requirements**
  - Number of Z-buffer reads/writes for a frame depends on:
    - Depth complexity of the scene: how many triangles cover each pixel on average
    - The order triangles are provided to the graphics pipeline
      (if depth test fails, don't write to depth buffer or rgba)

- **Bandwidth estimate:**
  - 60 Hz x 4 MPixel image x avg. scene depth complexity 4 (assume: replace 50% of time)
    x 32-bit Z = ~6 GB/s
  - GPUs often sample coverage multiple times per pixel for quality: multiply by 4
  - Scene is often rendered multiple times per frame: multiply by 2-3
  - Note: this is just depth buffer accesses. It does not include color-buffer bandwidth.

- **Modern GPUs have fixed-function hardware to implement caching and lossless compression of both color and depth buffers to reduce bandwidth**

Frame-buffer compression

Depth-buffer compression

- **Motivation: reduce bandwidth required for depth-buffer accesses**
  - Worst-case (uncompressed) buffer allocated in DRAM
  - Conserving memory <u>footprint</u> is a non-goal

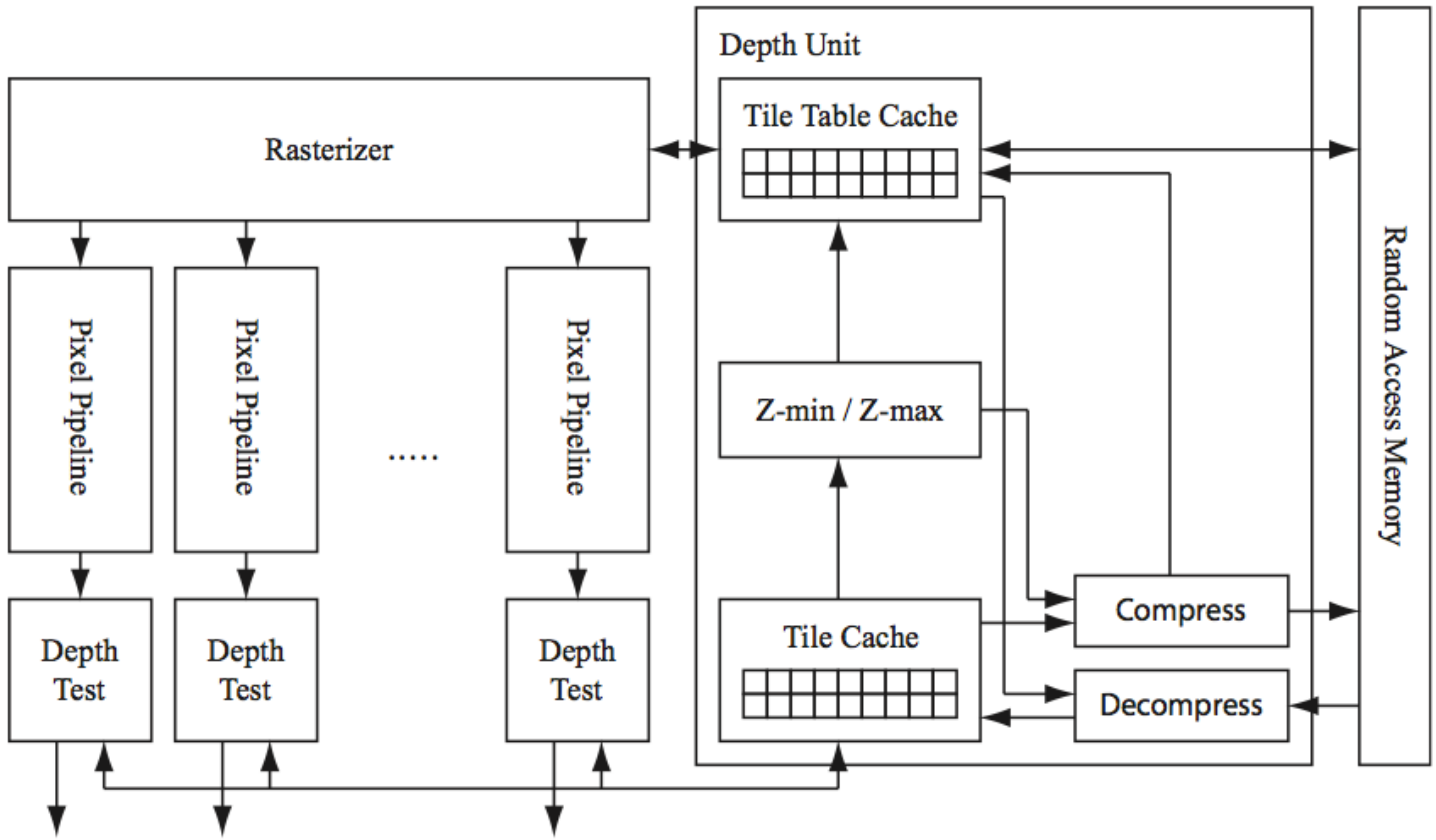
    (Need for real-time guarantees in graphics applications requires application to plan for worst case anyway)

- **Lossless compression**
  - Q. Why not lossy?

- **Designed for fixed-point numbers (fixed-point math in rasterizer)**

Depth-buffer compression is screen tile based

- **Main idea: exploit similarity of values within a screen tile**



On tile evict:

1. Compute zmin/zmax (needed for hierarchical culling and/or compression)
2. Attempt to compress
3. Update tile table
4. Store tile to memory

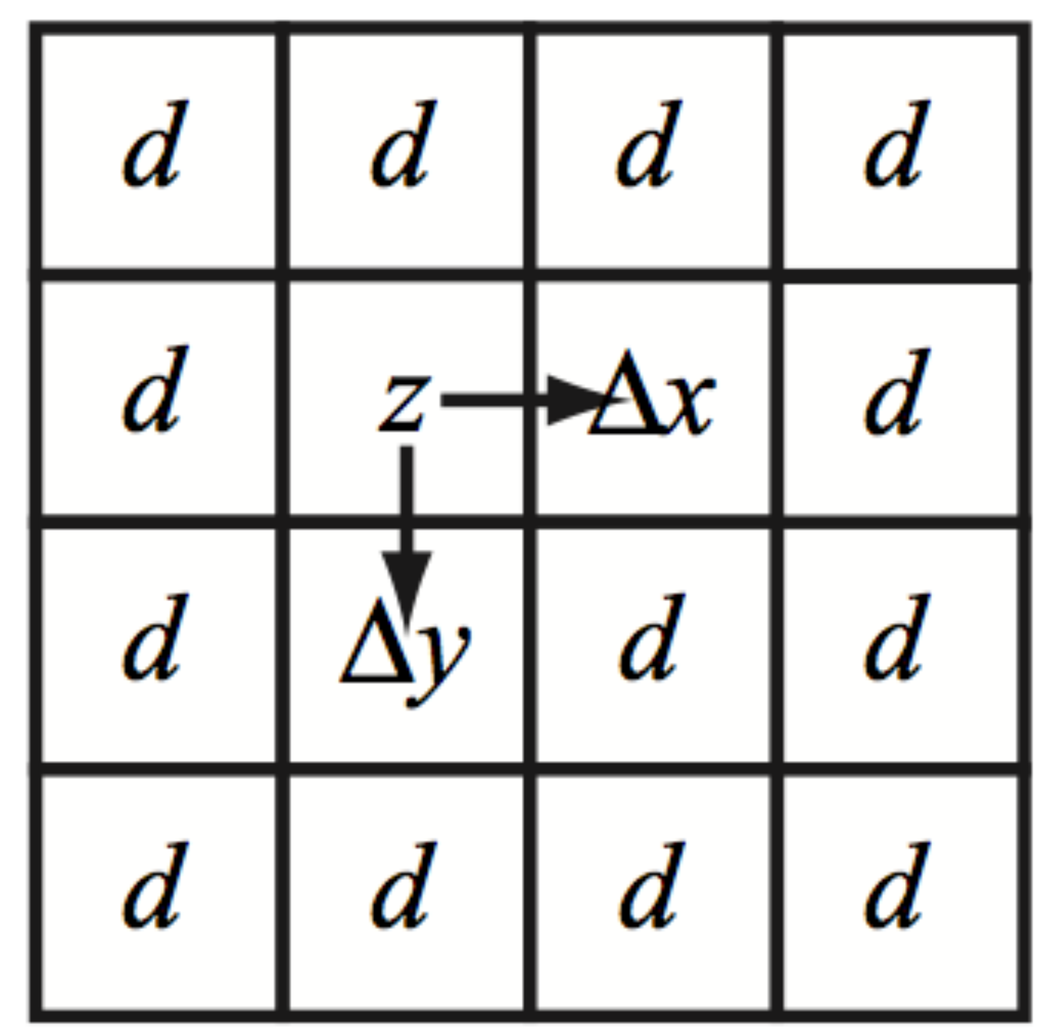
On tile load:

1. Check tile table for compression scheme
2. Load required bits from memory
3. Decompress into tile cache

Figure credit: [Hasselgren et al. 2006]

Anchor encoding

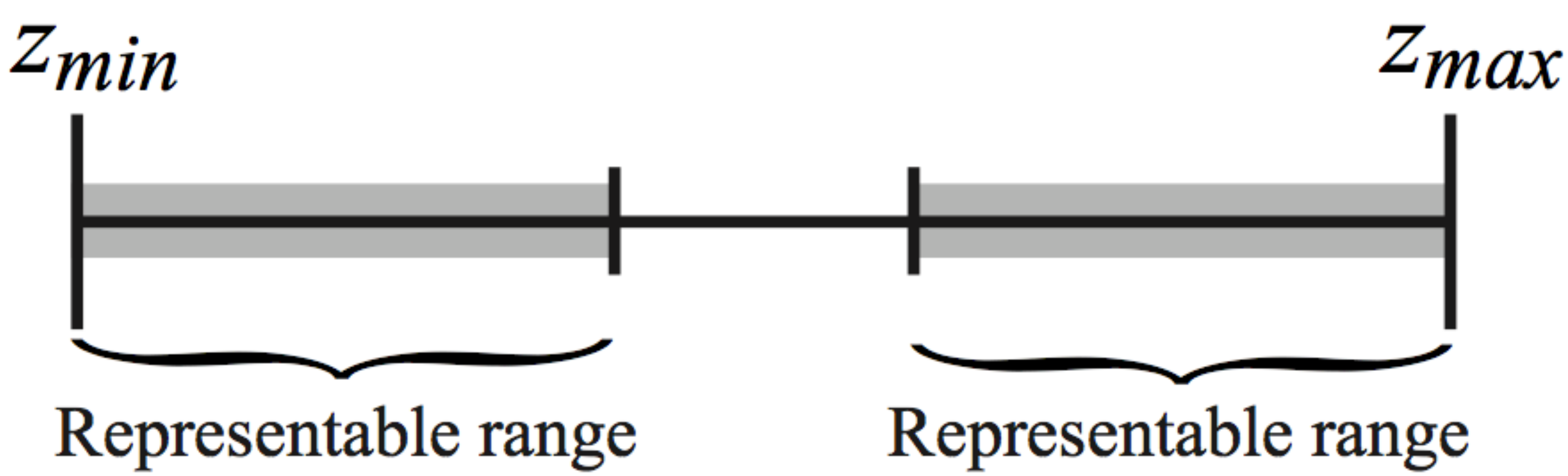
- Choose anchor value and compute DX, DY from adjacent pixels (fits a plane to the data)
- Use plane to predict depths at other pixels, store offset d from prediction at each pixel
- Scheme (for 24-bit depth buffer)
  - Anchor: 24 bits (full resolution)
  - DX, DY: 15 bits
  - Per-sample offsets: 5 bits



[Van Dyke and Margeson]

Depth-offset compression

- Assume depth values have low dynamic range relative to tile's zmin and zmaz (assume two surfaces)
- Store zmin/zmax
- Store low-precision (8-12 bits) offset value for each sample
  - MSB encodes if offset is from zmin or zmax



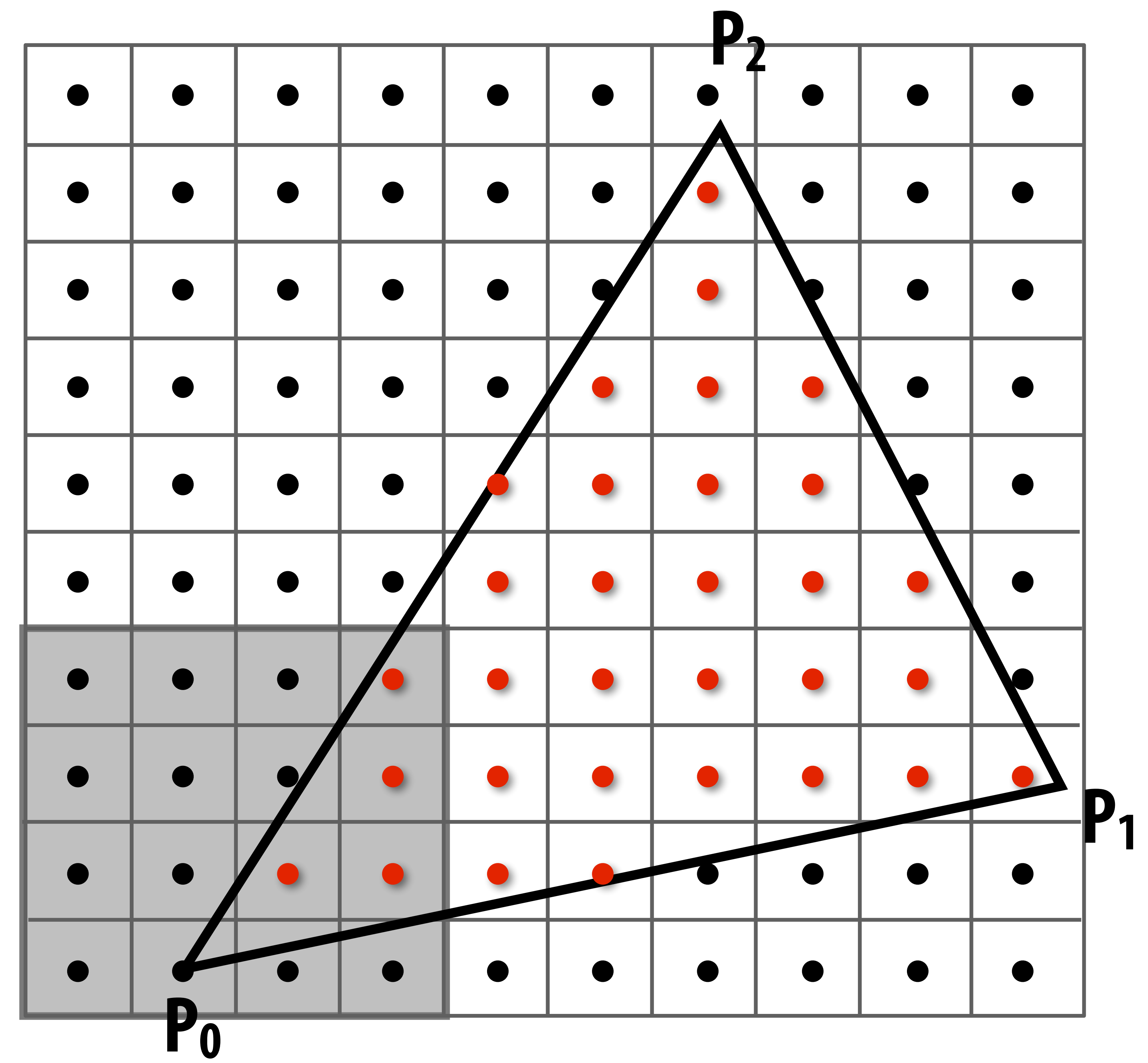
[Morein and Natali]

Explicit plane encoding

- Do not attempt to infer prediction plane, just get the plane equation of the triangle directly from the rasterizer
  - Store plane equation in tile (values must be stored with high precision: to match exact math performed by rasterizer)
  - Store bit per sample indicating coverage
- Simple extension to multiple triangles per tile:
  - Store up to N plane equations in tile
  - Store $\log_2(N)$ bit id per depth sample indicating which triangle it belongs to
- When new triangle contributes coverage to tile:
  - Add new plane equation if storage is available, else decompress
- To decompress:
  - For each sample, evaluate Z(x,y) for appropriate plane

0	0	0	0	0	1	1	1
0	0	0	0	0	1	1	1
0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
0	0	0	1	1	1	1	1
0	0	0	1	1	1	1	1
0	0	0	1	1	1	1	1

Aside: hierarchical occlusion culling: “hi-Z”



Z-Max culling:

For each screen tile, compute farthest value in the depth buffer (often needed for compression): `z_max`

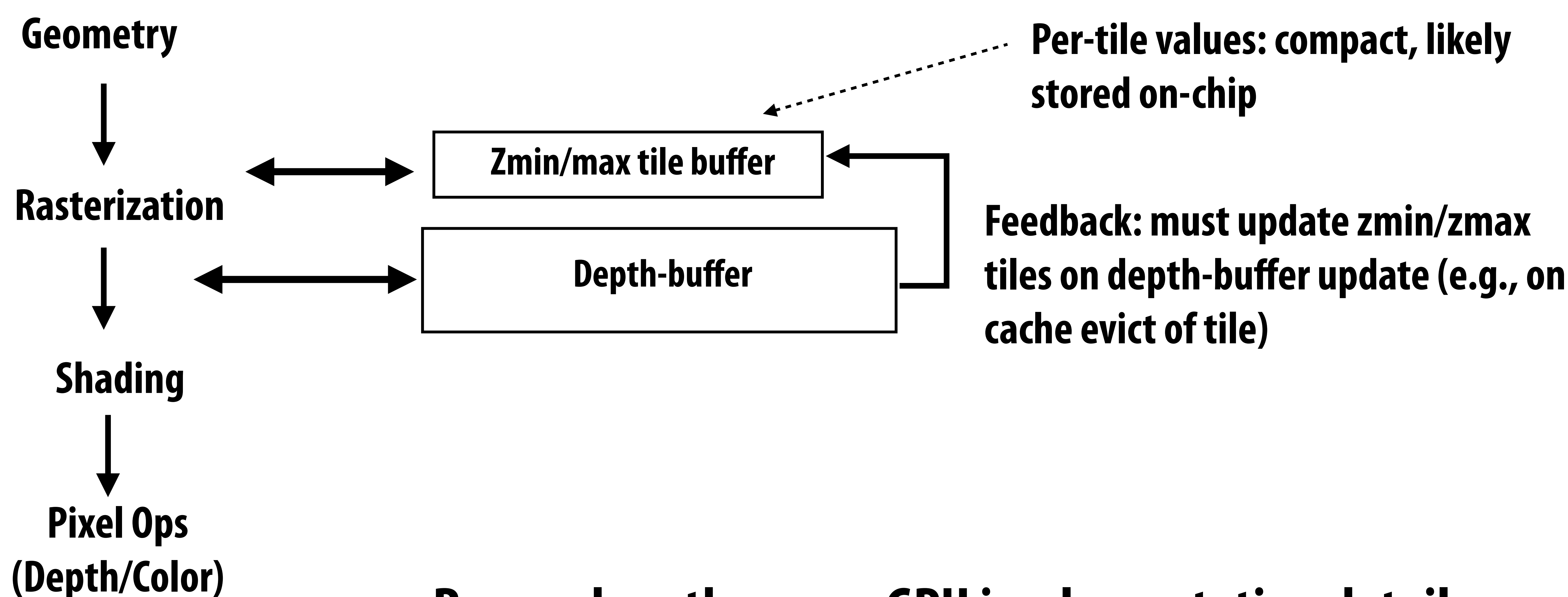
During traversal, for each tile:

1. **Compute closest point on triangle in tile: `tri_min`**
2. **If `tri_min > z_max`, then triangle is completely occluded in this tile. (The depth test will fail for all samples in the tile.) Proceed to next tile without performing coverage tests for individual samples in tile.**

Z-min optimization:

Depth-buffer also stores `z_min` for each tile.
If `tri_max < z_min`, then all depth tests for fragments in tile will pass. (No need to perform depth test on individual fragments.)

Hierarchical Z + early Z-culling



Remember: these are GPU implementation details (common optimizations performed by most GPUs in fixed-function hardware) They are invisible to the programmer and not reflected in the graphics pipeline abstraction

Summary: reducing the bandwidth requirements of depth testing

- Caching: access DRAM less often (by caching depth buffer data)
- Data compression: reduce number of bits that must be read from memory
- Hierarchical Z techniques (zmin/zmax culling): "early outs" result in accesses individual sample data less often

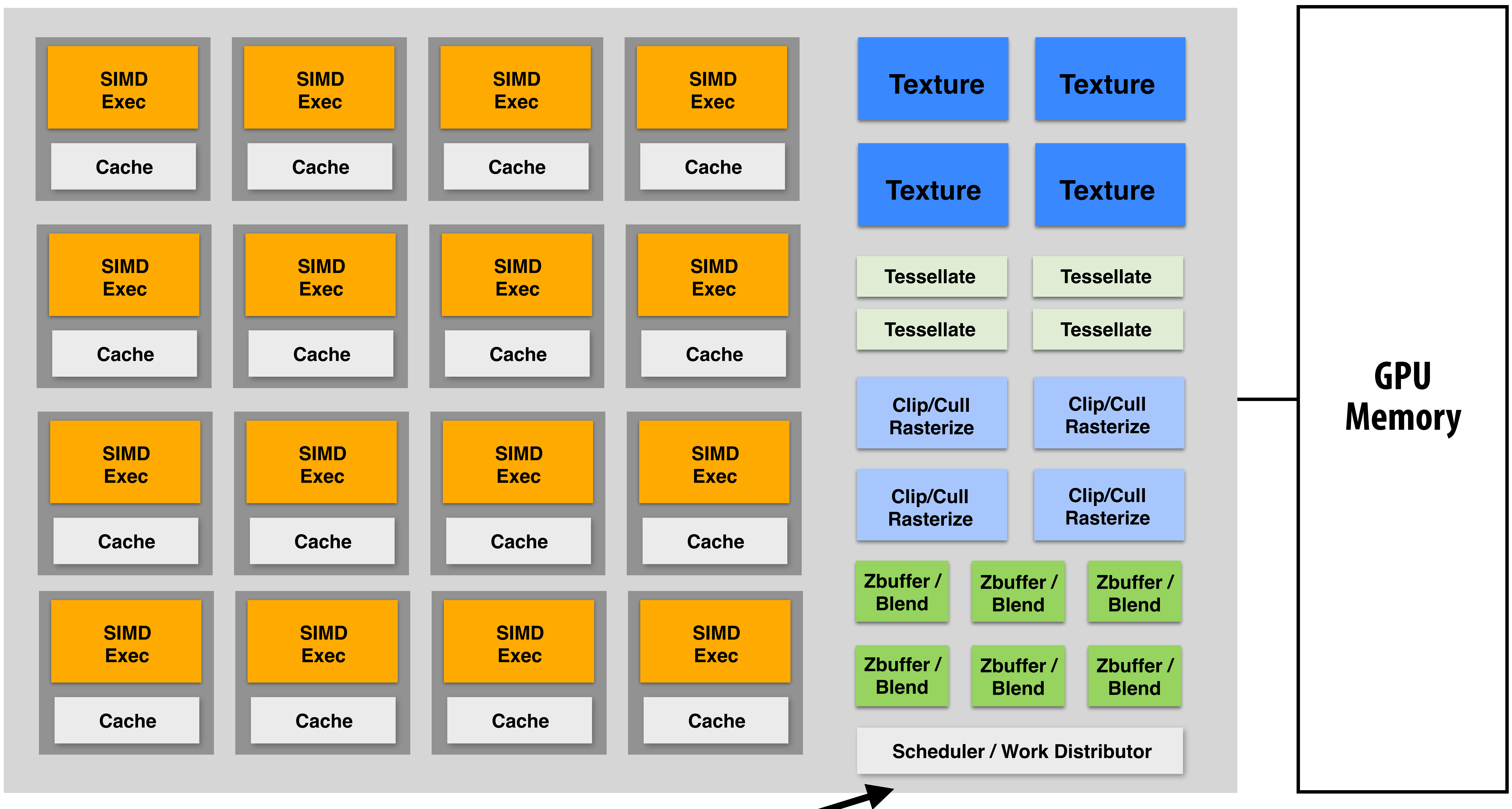
---

- Color buffer is also compressed using similar techniques
  - Depth buffer typically achieves higher compression ratios than color buffer. Why?

The story so far

- Parallelizing rasterization with data-parallel hardware for point-in-triangle tests
- Accelerate depth testing with fixed-function hardware for data compression (also hardware for the math of the depth test)
- But what about parallelizing the computation over many triangles?

GPU



We're now going to talk about this scheduler

The graphics pipeline

For each input triangle, in input command order…

Geometry:
Compute vertex positions on screen *

↓

Rasterization:
compute covered samples

↓

Shading:
compute color of colored pixels

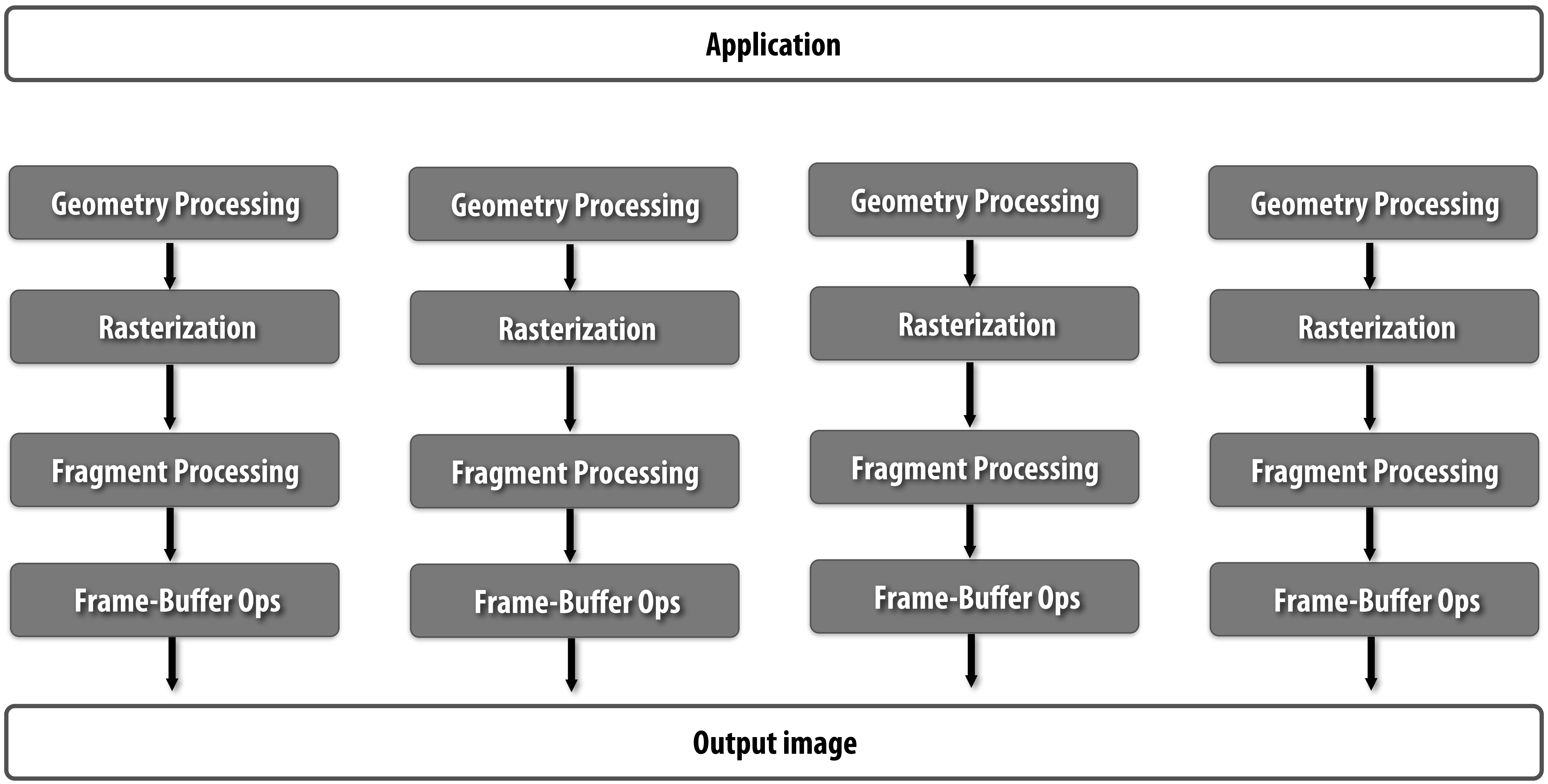
↓

Pixel Ops:
Depth Test and Depth/Color Write

*** In practice, there's more done here than just projection: animation, etc.**

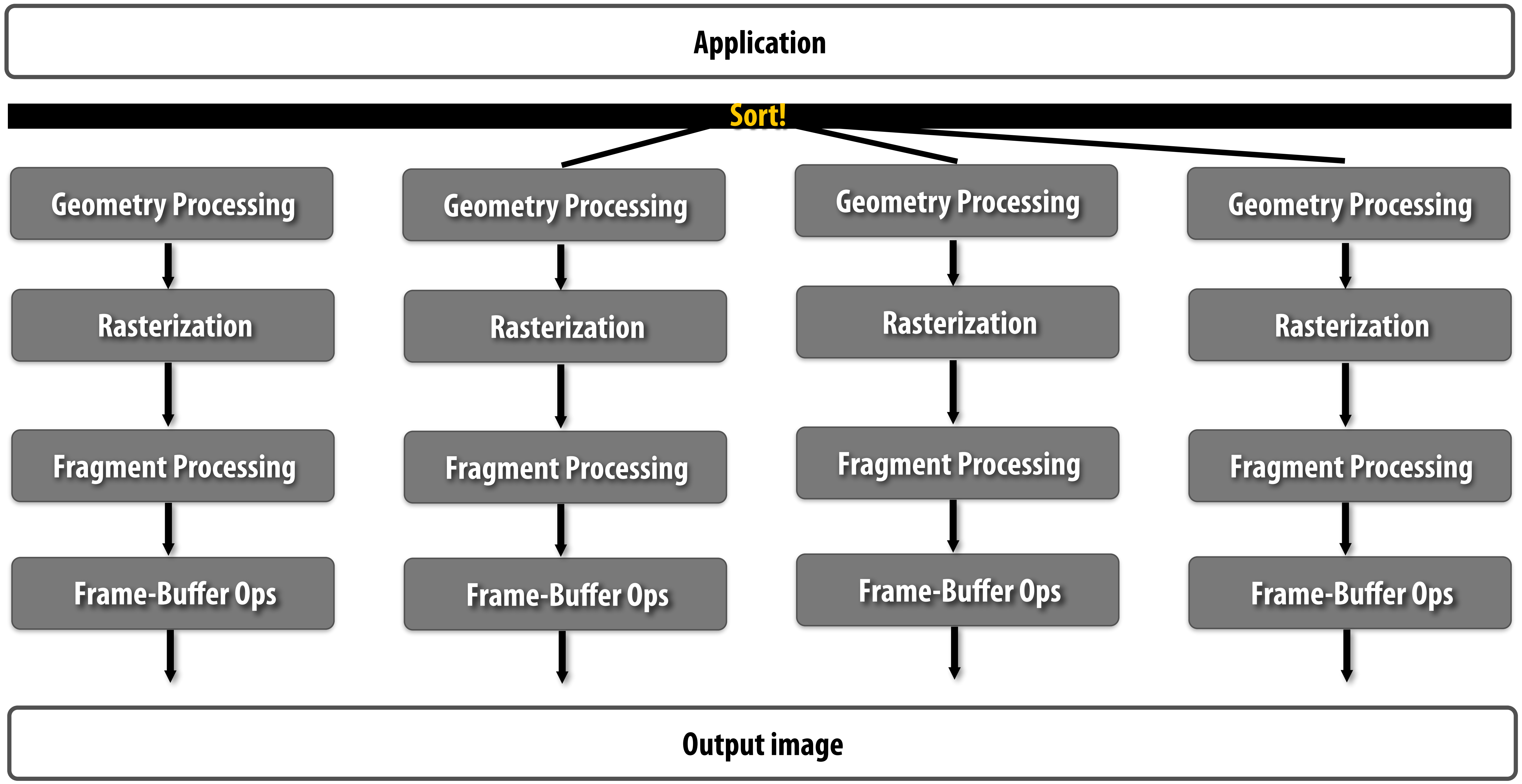
A cartoon GPU:

Assume we have four separate processing pipelines



Sort first

Sort first

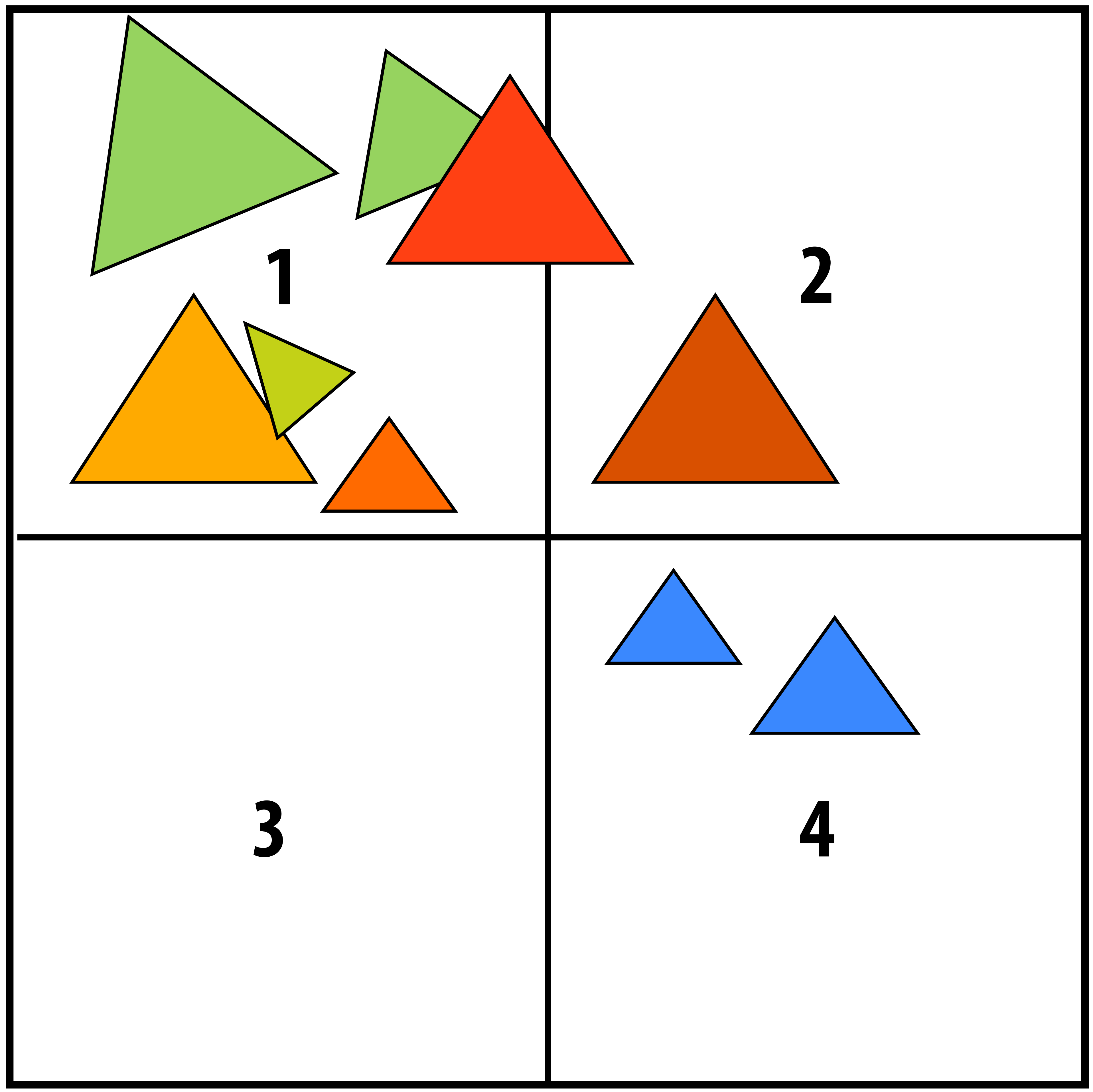


Assign each replicated pipeline responsibility for a region of the output image

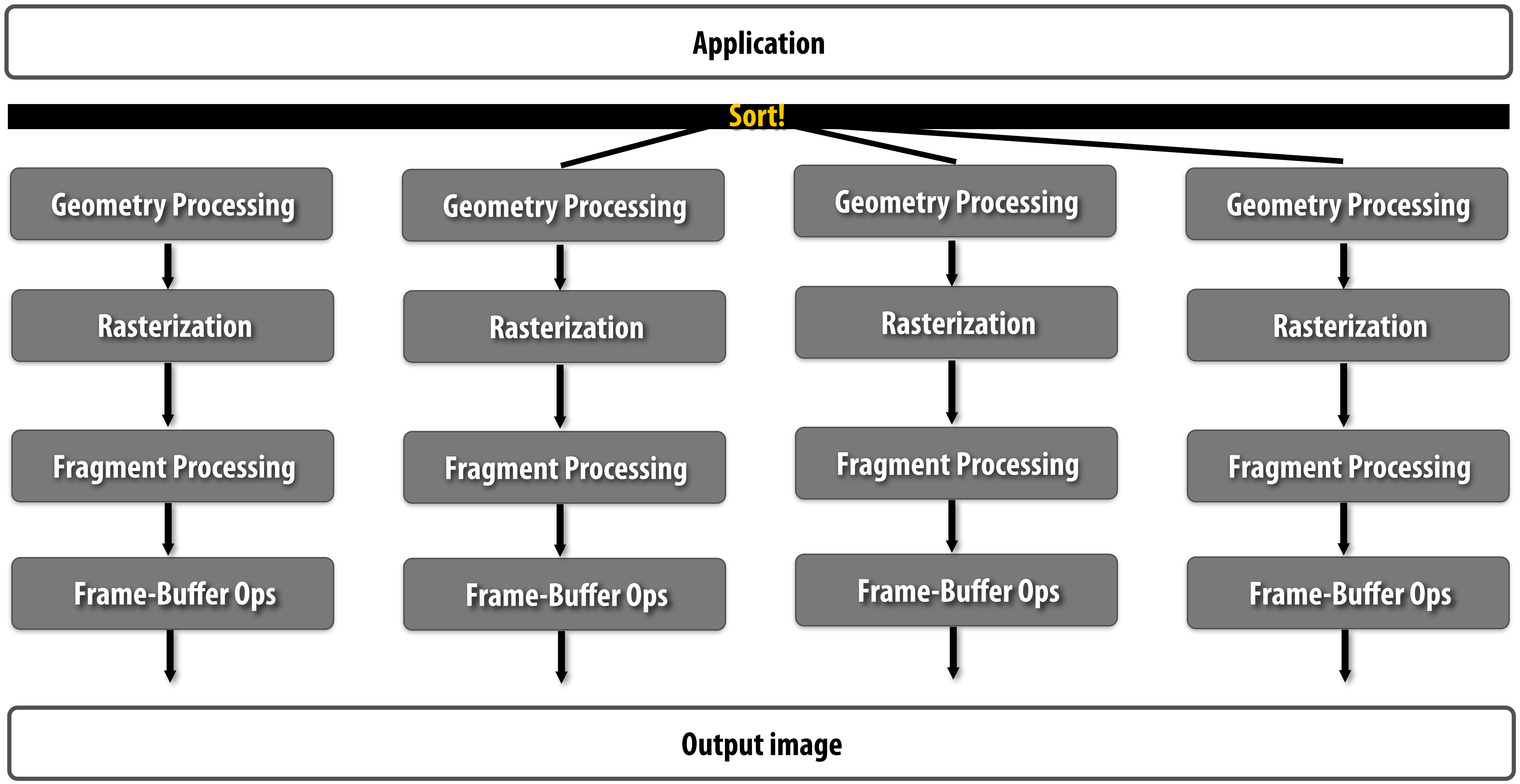
Do minimal amount of work to determine which region(s) each input primitive overlaps

Sort first work partitioning

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

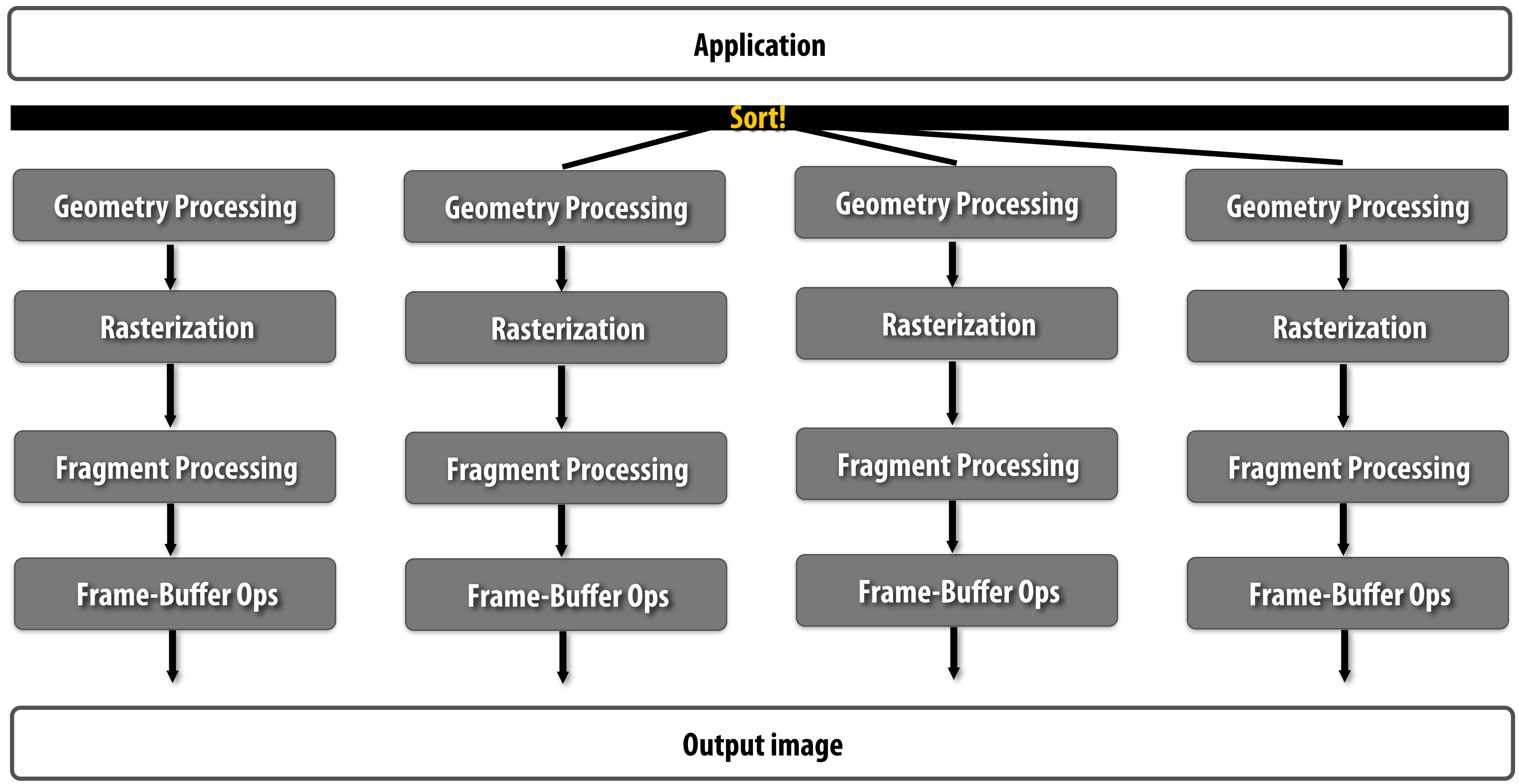


Sort first



- **Good:**
  - **Bandwidth scaling (small amount of sync/communication, simple point-to-point)**
  - **Computation scaling (more parallelism = more performance)**
  - **Simple: just replicate rendering pipeline (order maintained within each)**

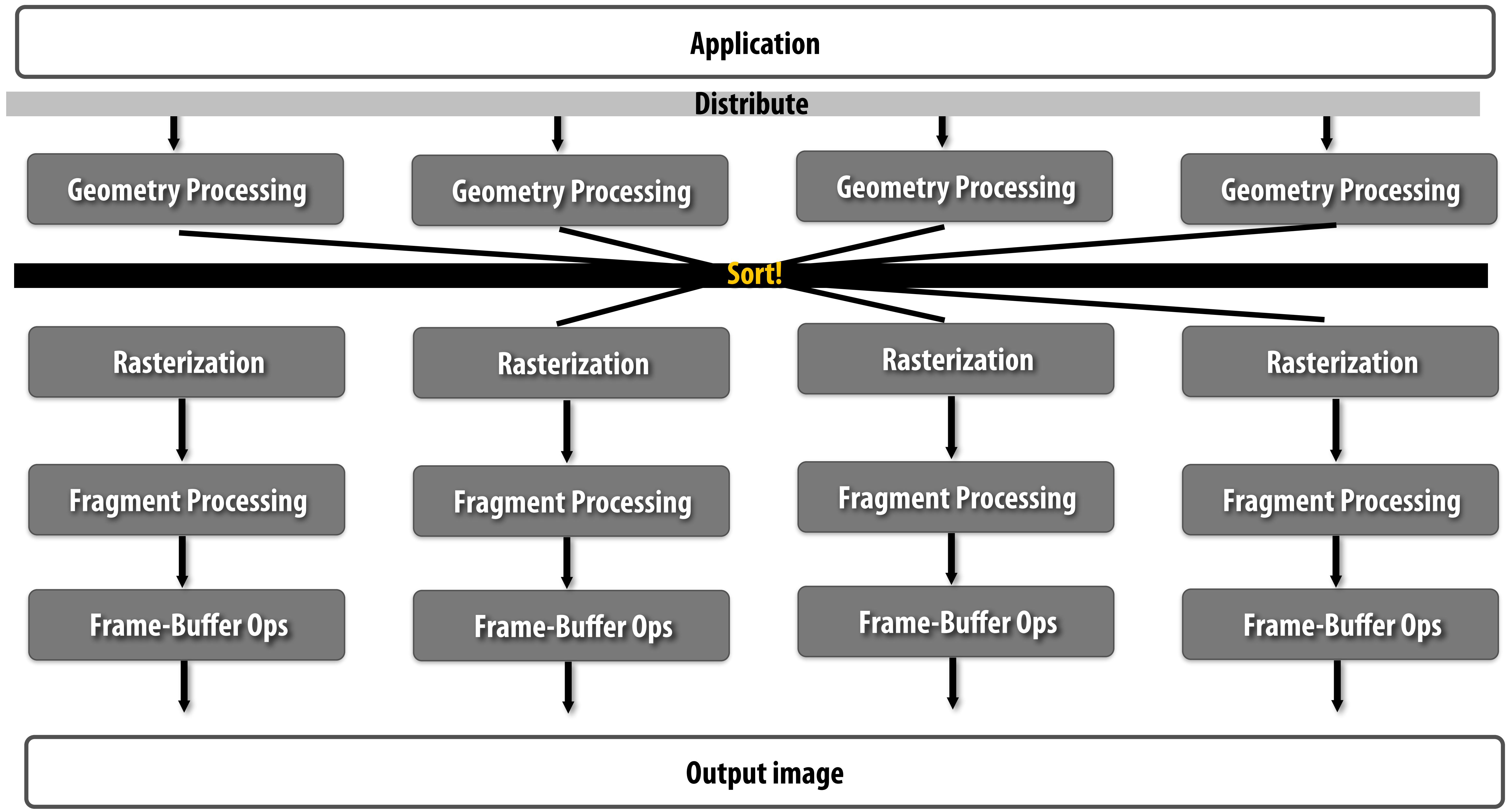
Sort first



- **Bad:**
  - **Potential for workload imbalance (one part of screen contains most of scene)**
  - **"Tile spread": as screen tiles get smaller, primitives cover more tiles (duplicate geometry processing across the parallel pipelines)**

Sort middle

Sort middle



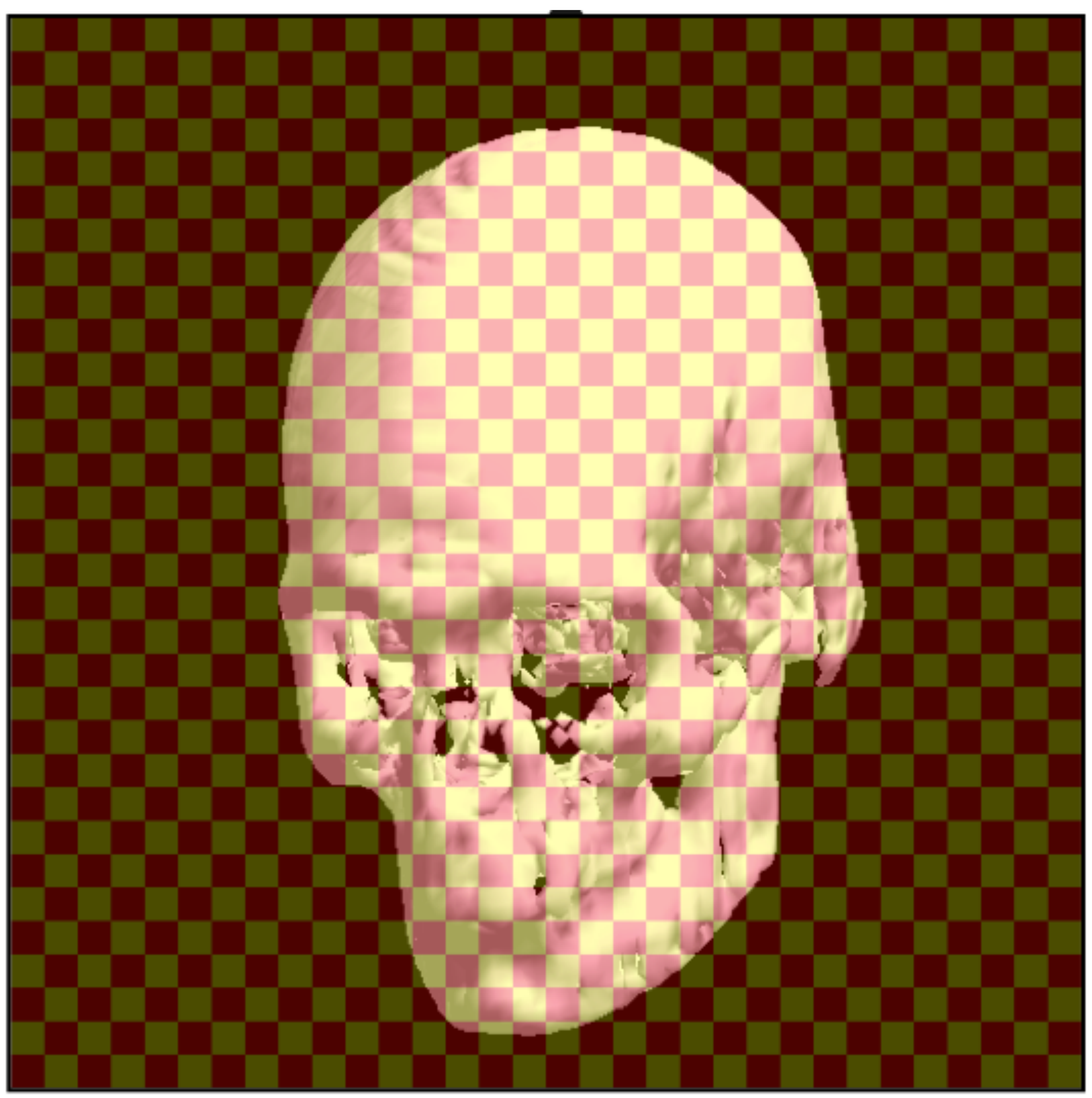
Assign each <u>rasterizer</u> a region of the render target

Distribute primitives to pipelines (e.g., round-robin distribution)

Sort after geometry processing based on screen space projection of primitive vertices

Interleaved mapping of screen

- Decrease chance of one rasterizer processing most of scene
- Most triangles overlap multiple screen regions (often overlap all)

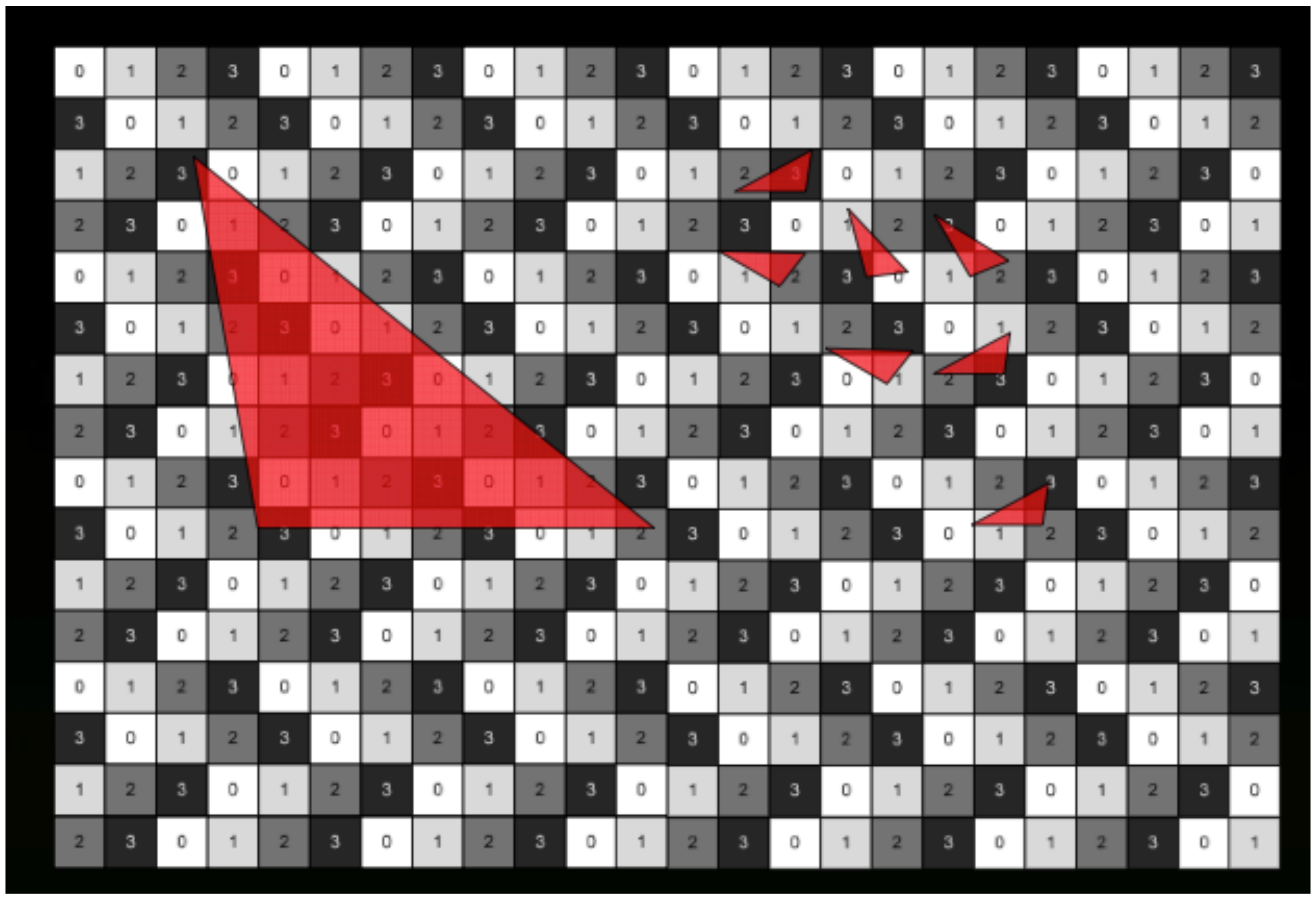


Interleaved mapping

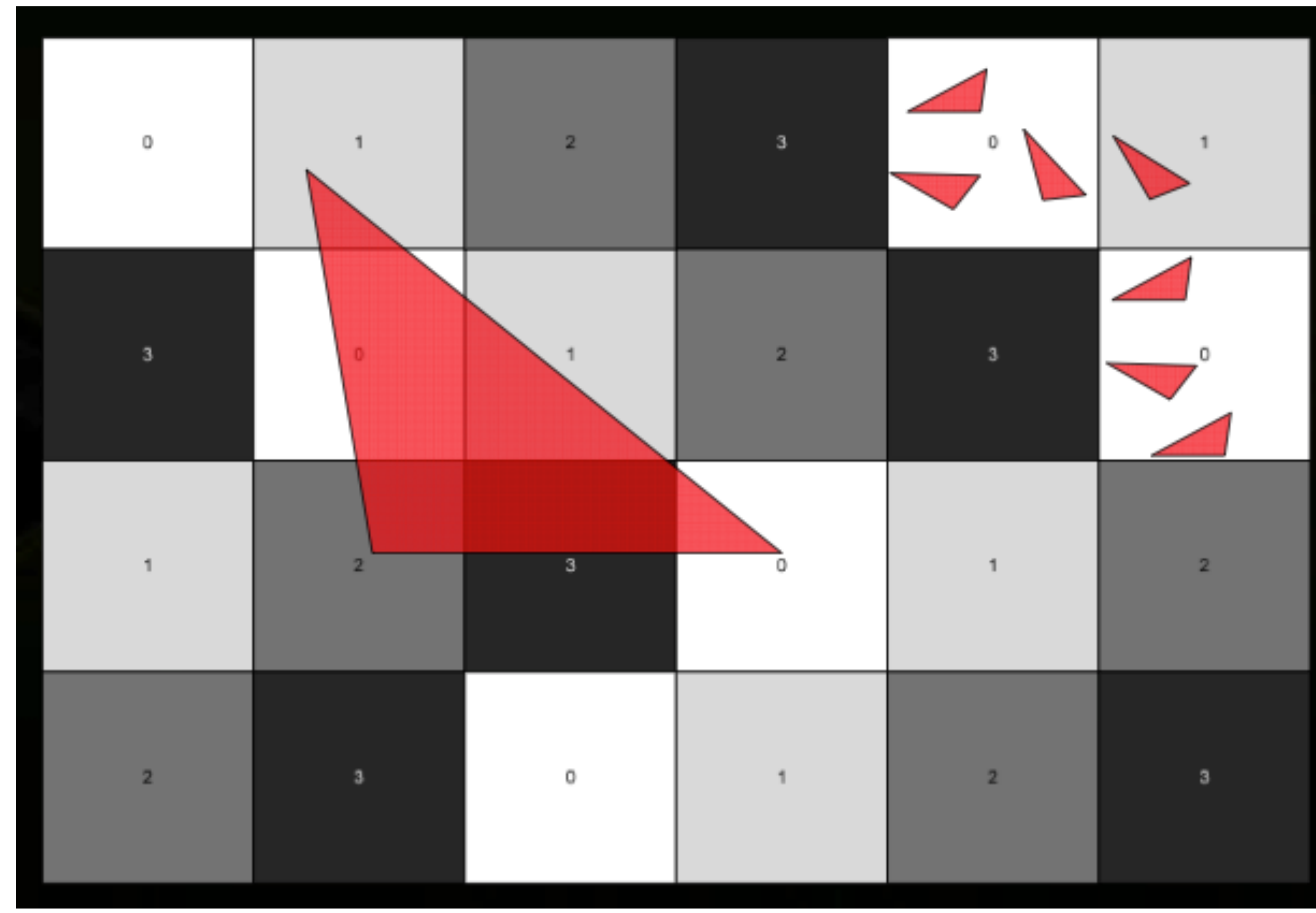
Tiled mapping

Fragment interleaving in NVIDIA Fermi

Fine granularity interleaving



Coarse granularity interleaving

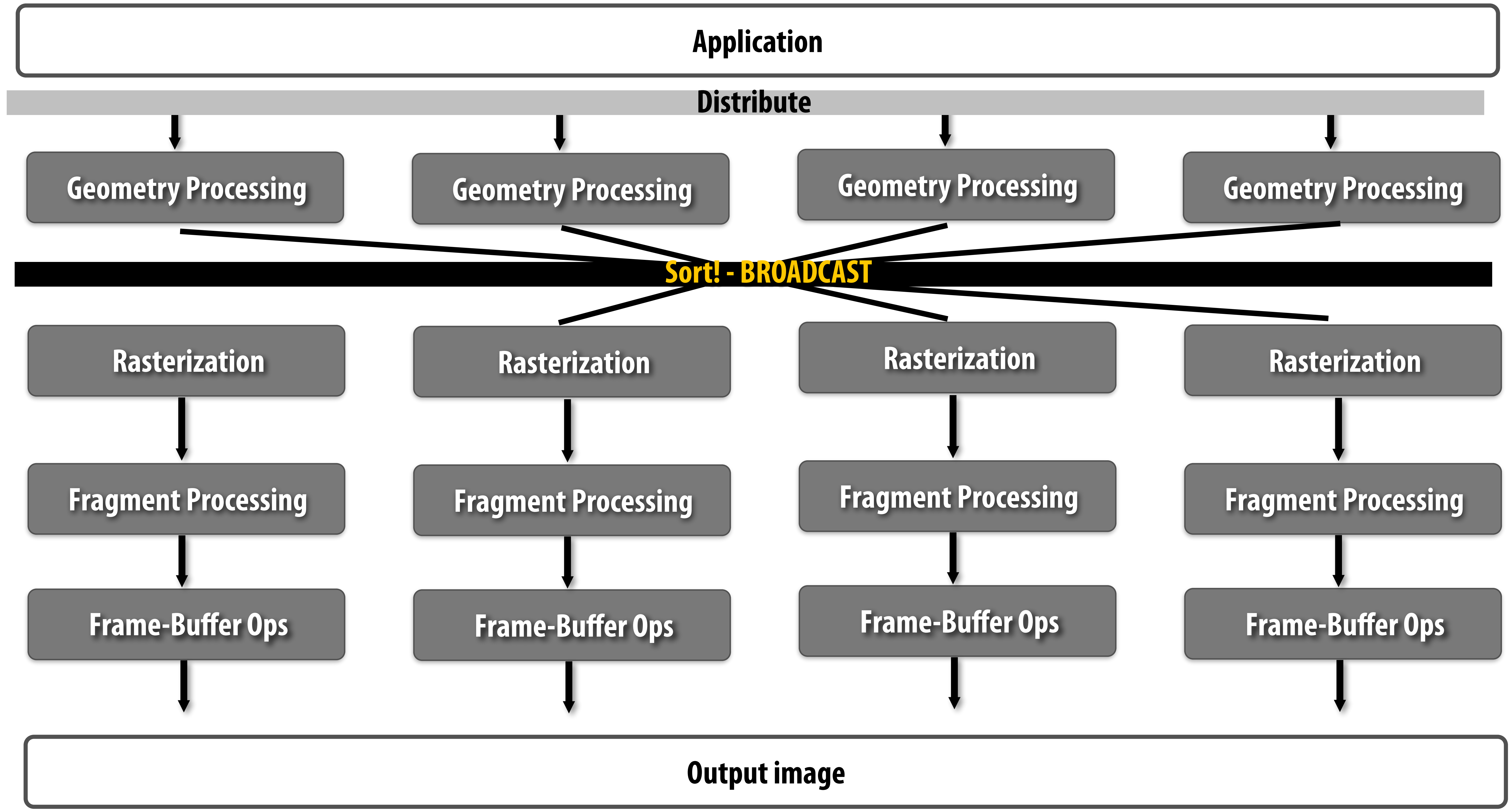


Question 1: what are the benefits/weaknesses of each interleaving?

Question 2: notice anything interesting about these patterns?

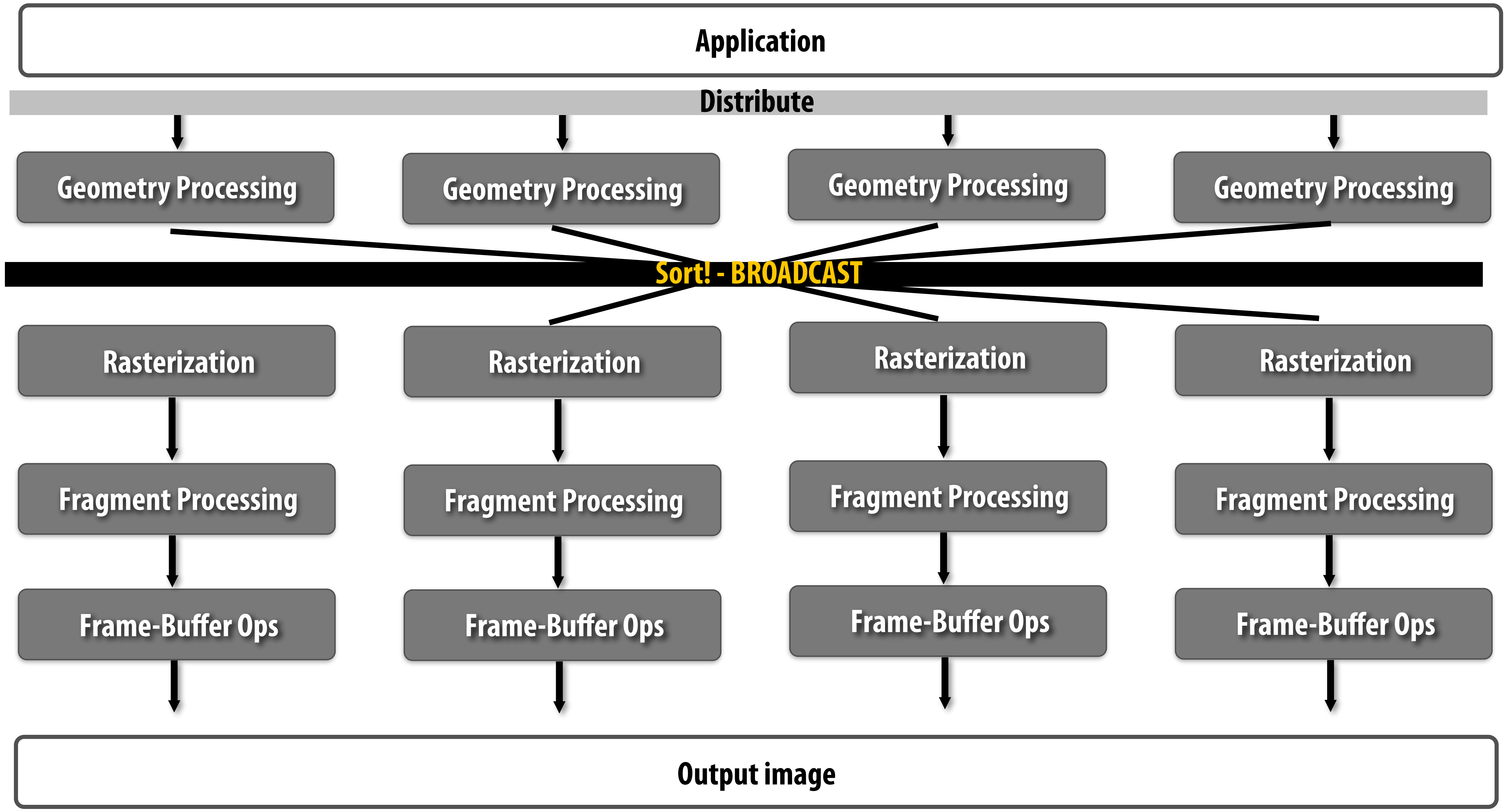
[Image source: NVIDIA]

Sort middle interleaved



- **Good:**
  - Workload balance: both for geometry work AND onto rasterizers (due to interleaving)
  - Does not duplicate geometry processing for each overlapped screen region

Sort middle interleaved

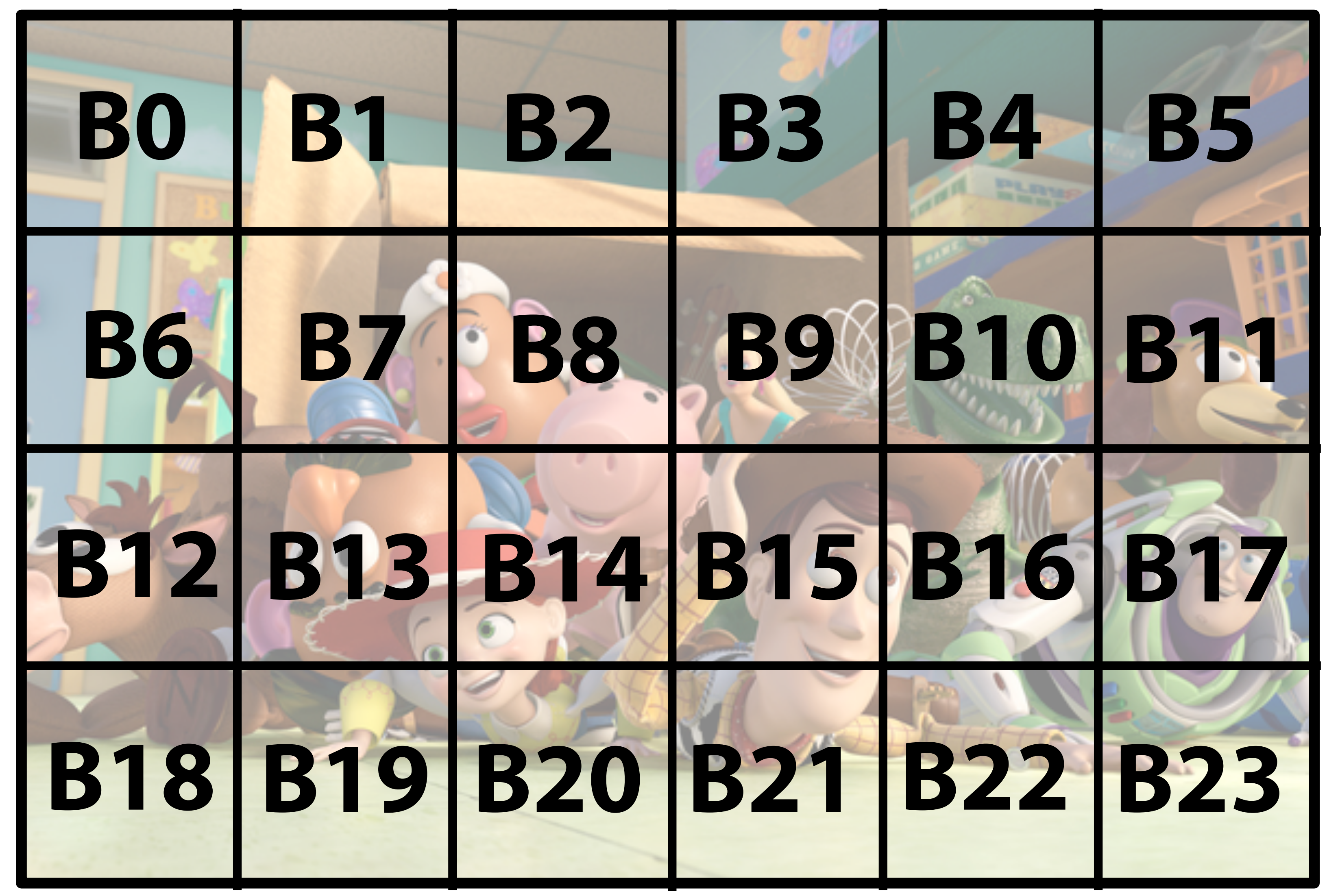


- **Bad:**
  - **Bandwidth scaling: sort is implemented as a broadcast (each triangle goes to many/all rasterizers)**
  - **If tessellation is enabled, must communicate many more primitives than sort first**
  - **Duplicated per triangle setup work across rasterizers**

Tiling (a.k.a. "chunking", "bucketing")

Processor 1 | Processor 2
Processor 3 | Processor 4

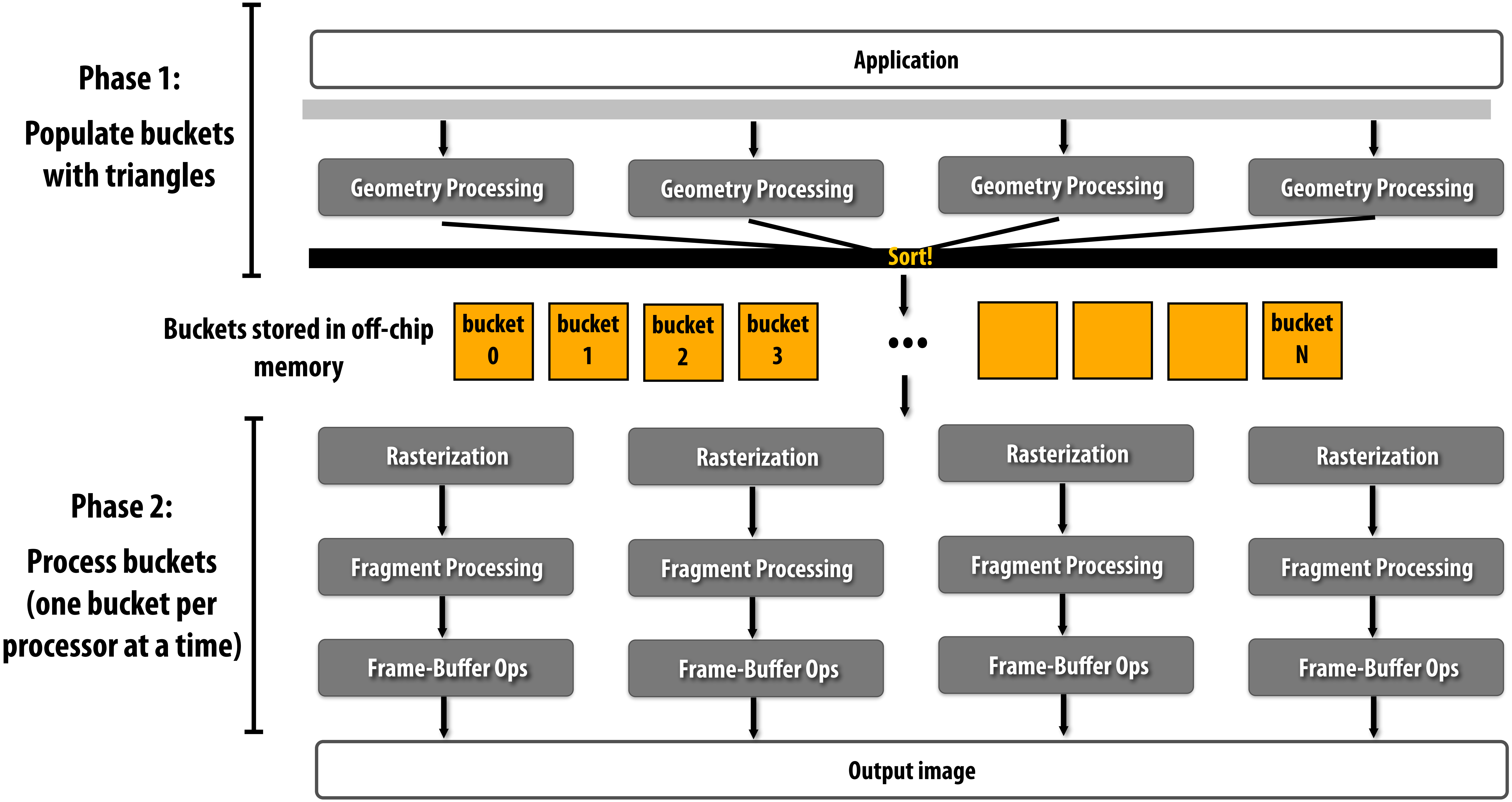
Interleaved (static) assignment to processors



Assignment to buckets

List of buckets is a work queue. Buckets are dynamically assigned to processors.

Sort middle tiled (chunked)



Partition screen into many small tiles (many more tiles than physical rasterizers)

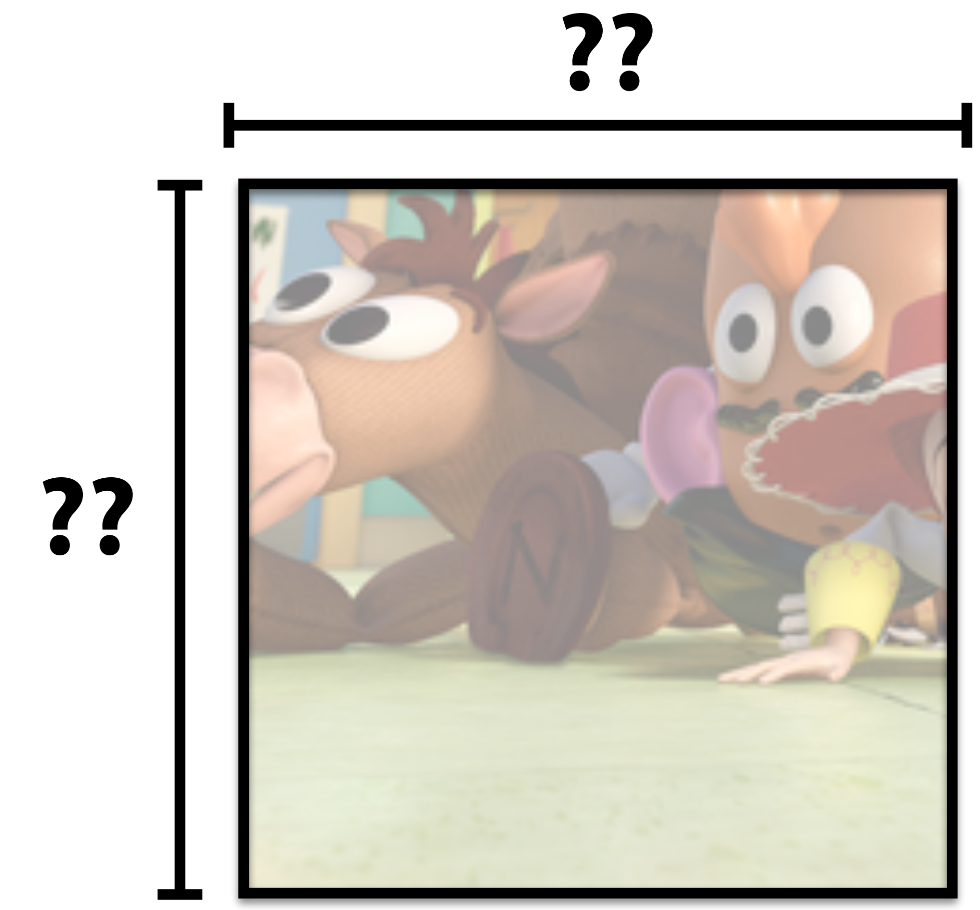
Sort geometry by tile into buckets (one bucket per tile of screen)

After all geometry is bucketed, rasterizers process buckets in parallel

Sort middle tiled (chunked)

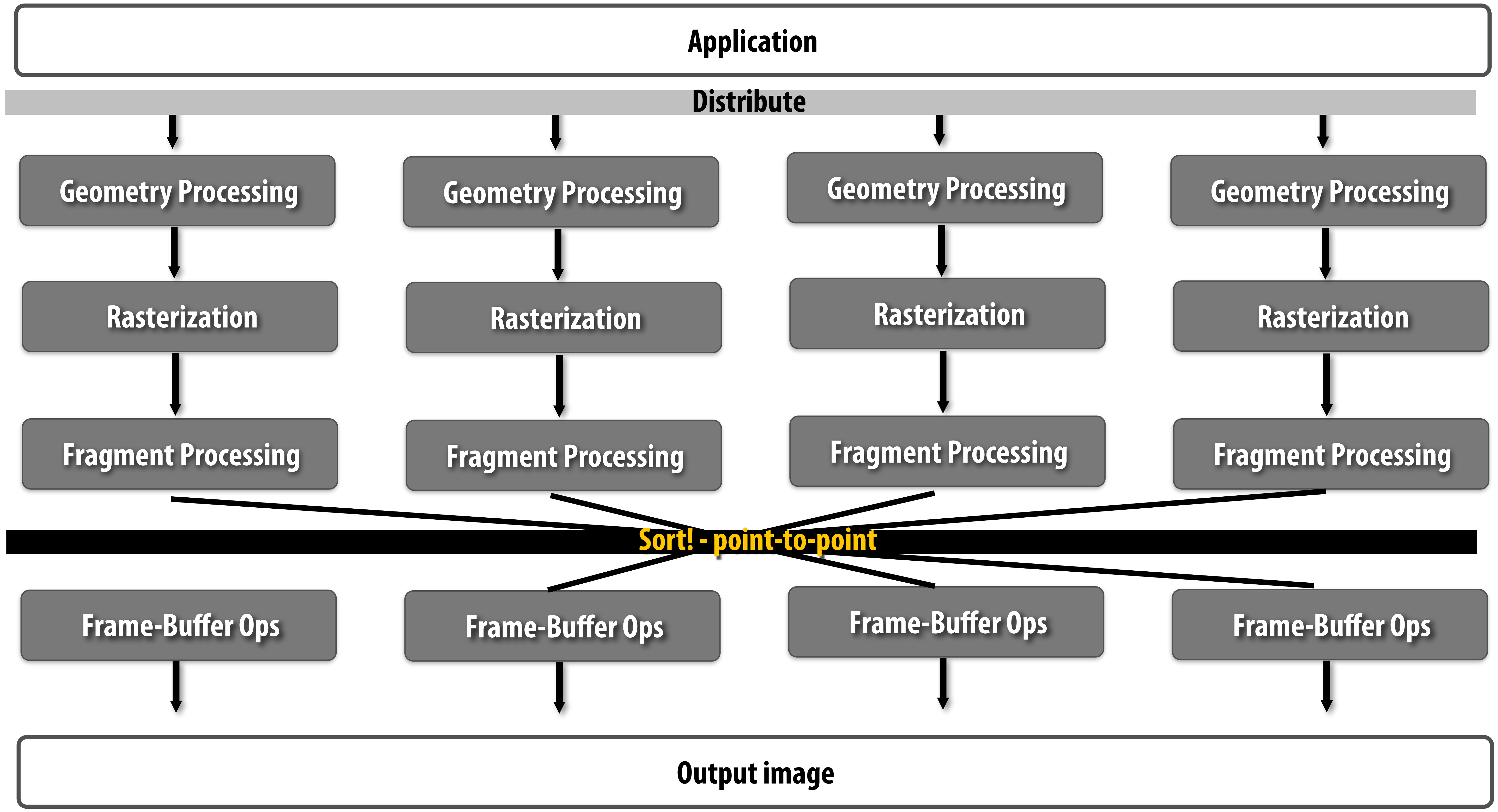
- **Good:**
  - Sort requires point-to-point traffic (assuming each triangle only touches a few buckets)
  - Good load balance (distribute many buckets onto rasterizers)
  - Potentially low bandwidth requirements (why? when?)
    - Question: What should the size of tiles be for maximum BW savings?

- **Recent examples:**
  - Mobile GPUs: Imagination PowerVR, ARM Mali, etc.
  - Parallel software rasterizers
    - Intel Larrabee software rasterizer
    - NVIDIA CUDA software rasterizer
    - 15-418/618 Assignment 2



Sort last

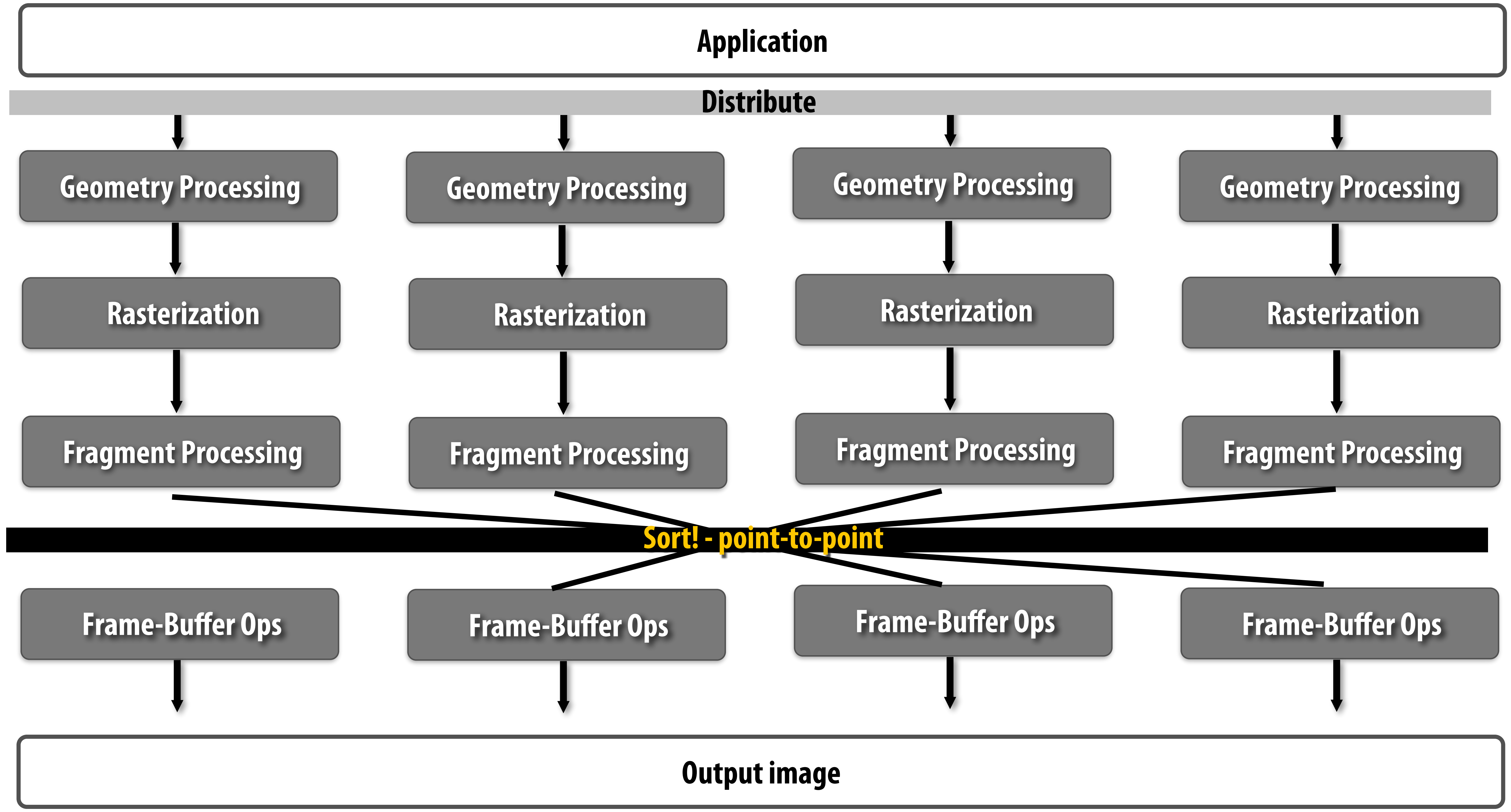
Sort last fragment



Distribute primitives to top of pipelines (e.g., round robin)

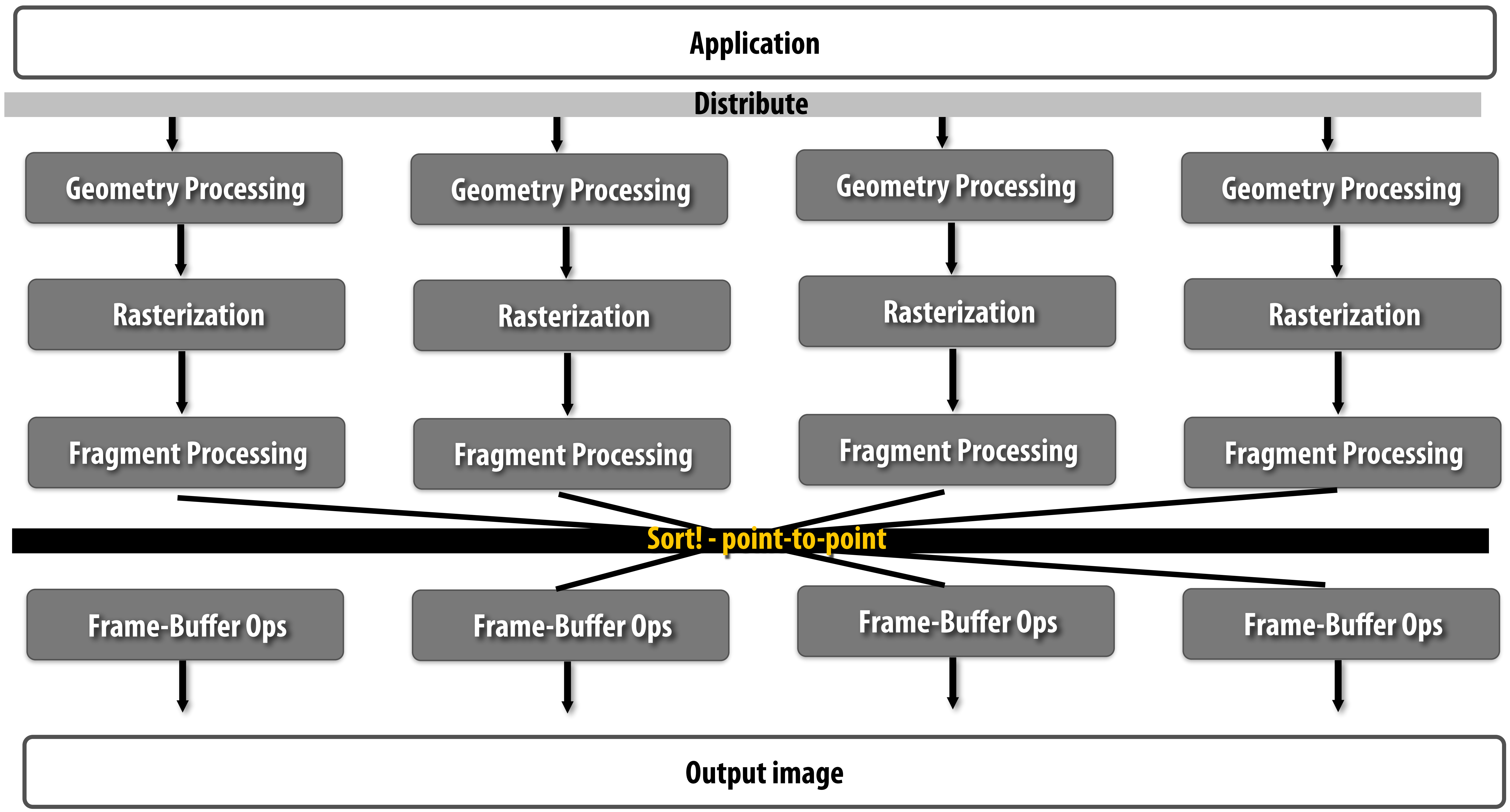
Sort after fragment processing based on (x,y) position of fragment

Sort last fragment



- **Good:**
  - No redundant geometry processing or in rasterizers (but z-cull is a problem)
  - Point-to-point communication during sort
  - Interleaved pixel mapping results in good workload balance for frame-buffer ops

Sort last fragment

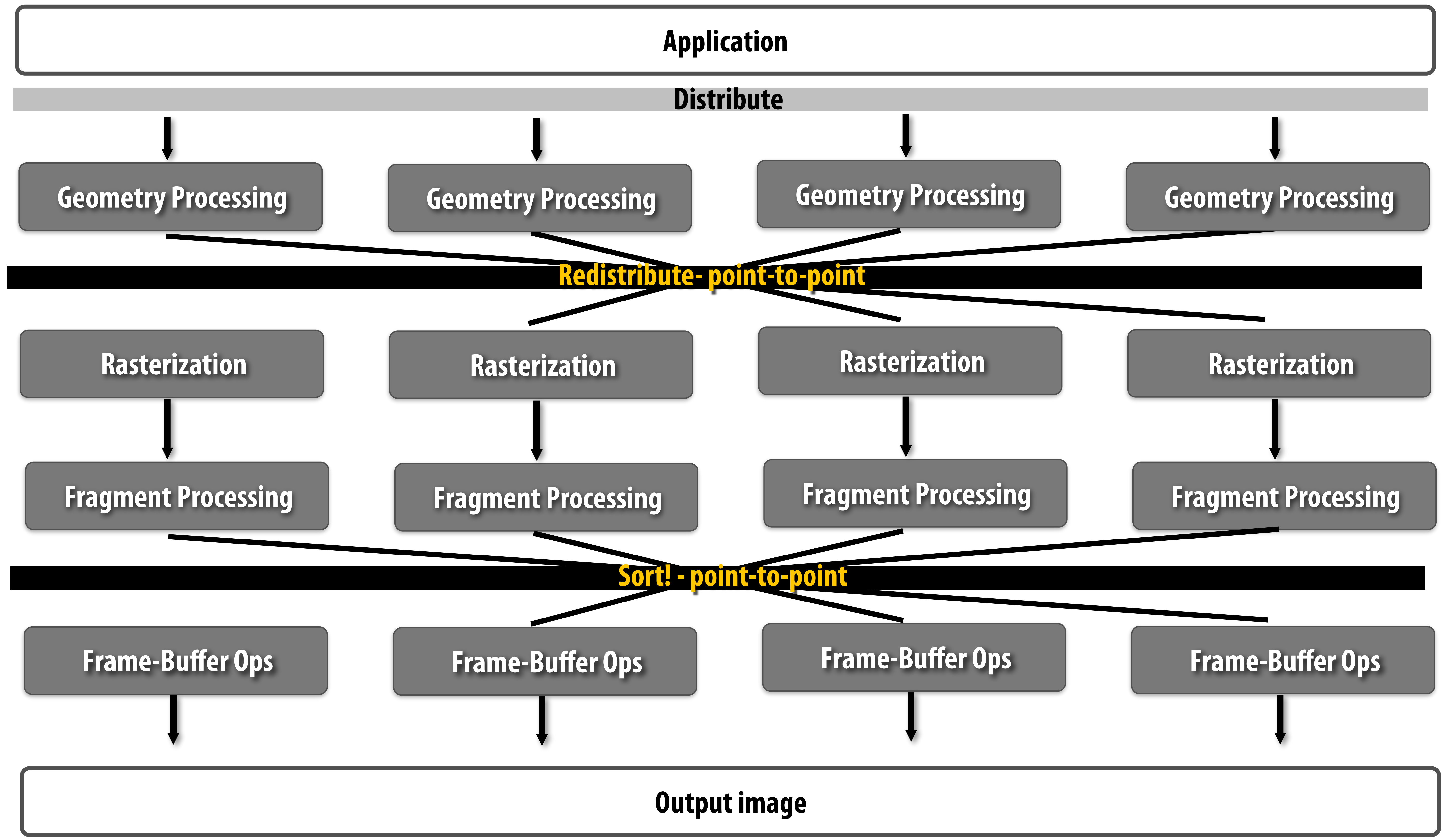


- Bad:
  - Workload imbalance due to primitives of varying size (due to order)
  - Bandwidth scaling: many more fragments than triangles
  - Early z cull is difficult

Sort everywhere

Sort everywhere

[Eldridge 00]



Distribute primitives to top of pipelines

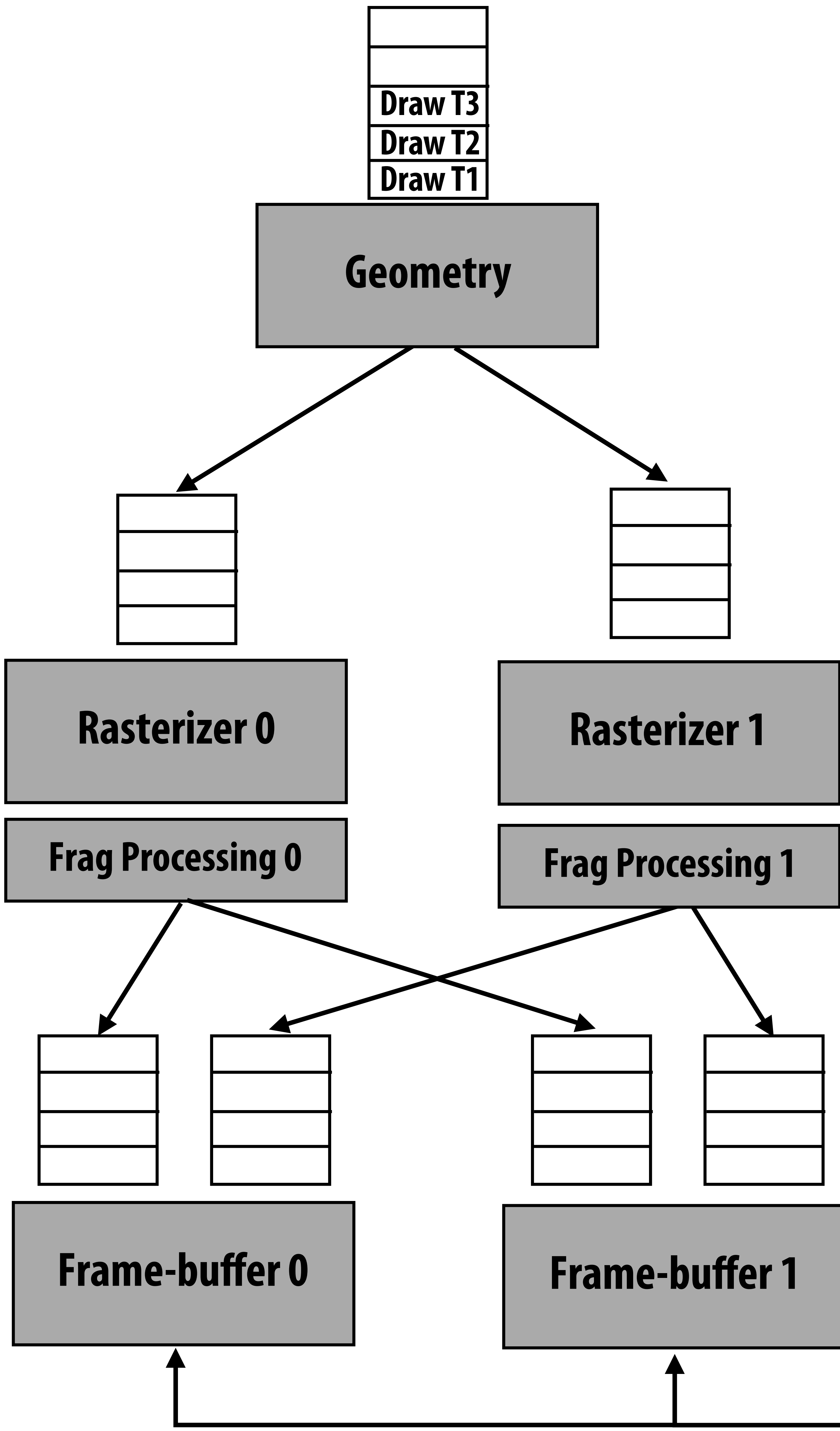
Redistribute after geometry processing (e.g, round robin)

Sort after fragment processing based on (x,y) position of fragment

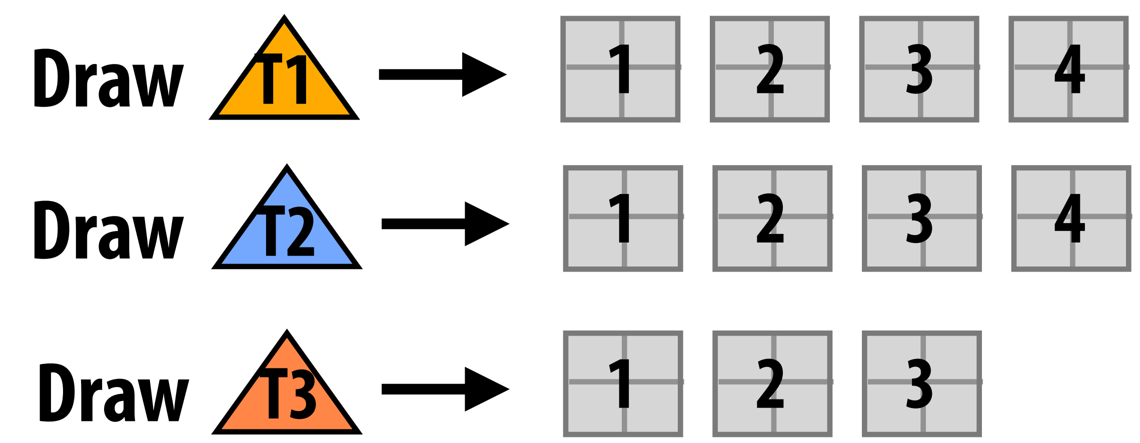
Implementing sort everywhere

(Challenge: rebalancing work at multiple places in the graphics pipeline to achieve efficient parallel execution, while maintaining triangle draw order)

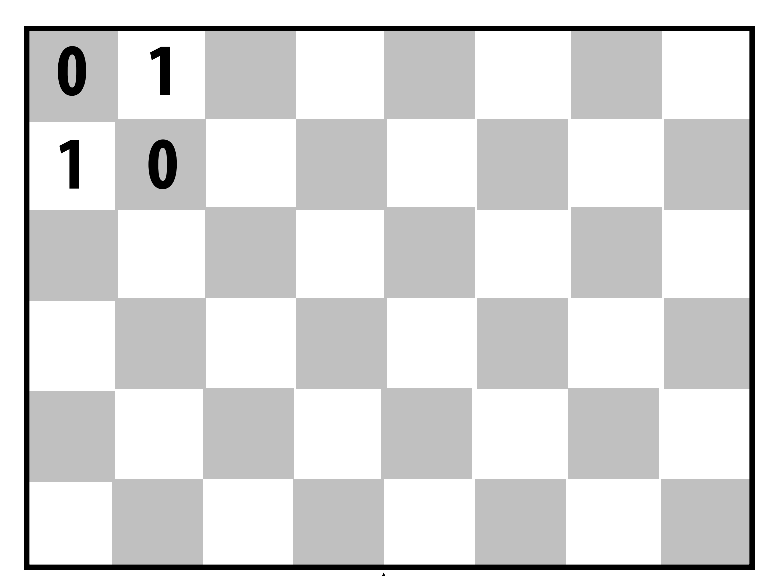
Starting state: draw commands enqueued for pipeline



Input: three triangles to draw
(covered pixels for each triangle by rasterization are shown below)

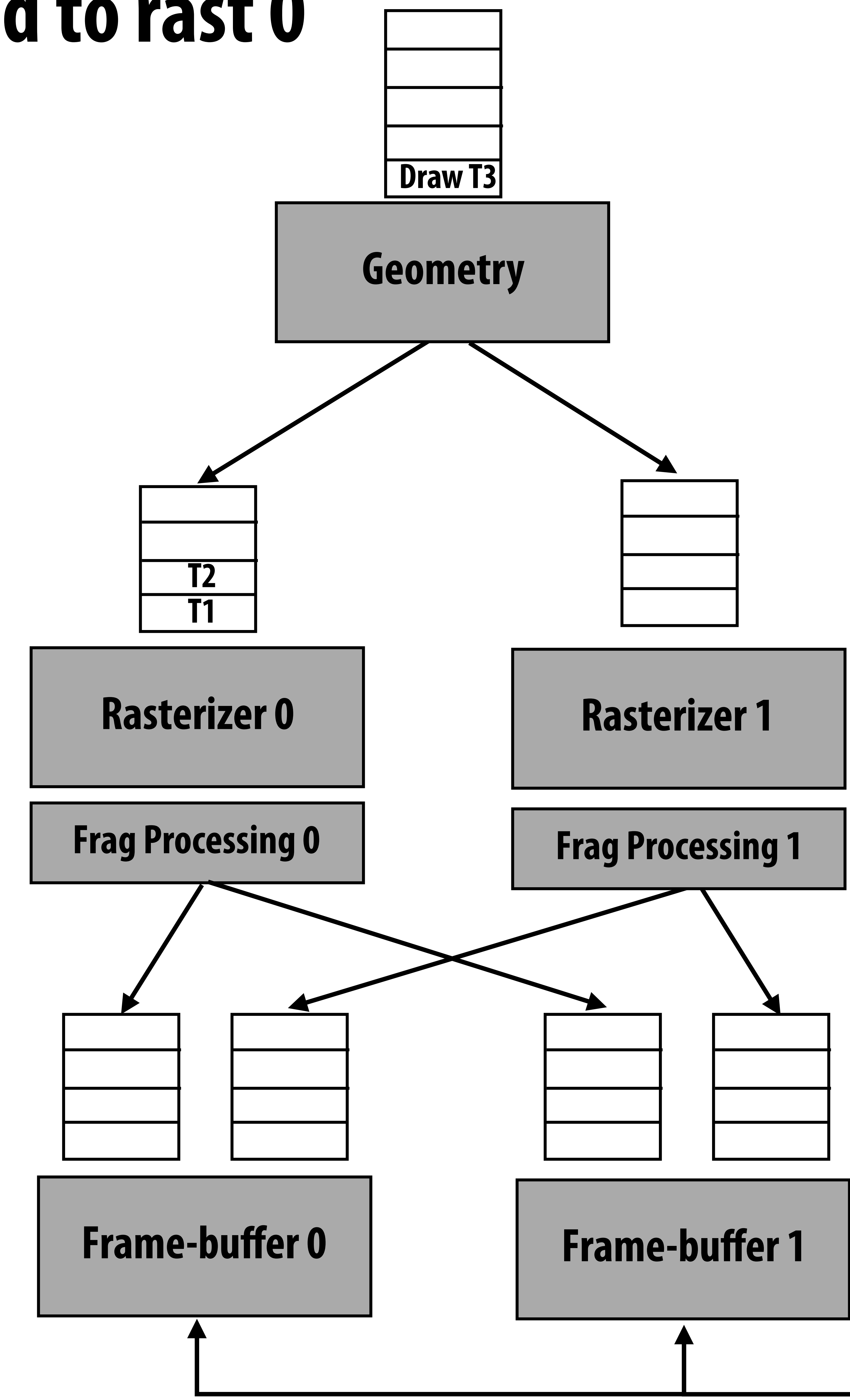


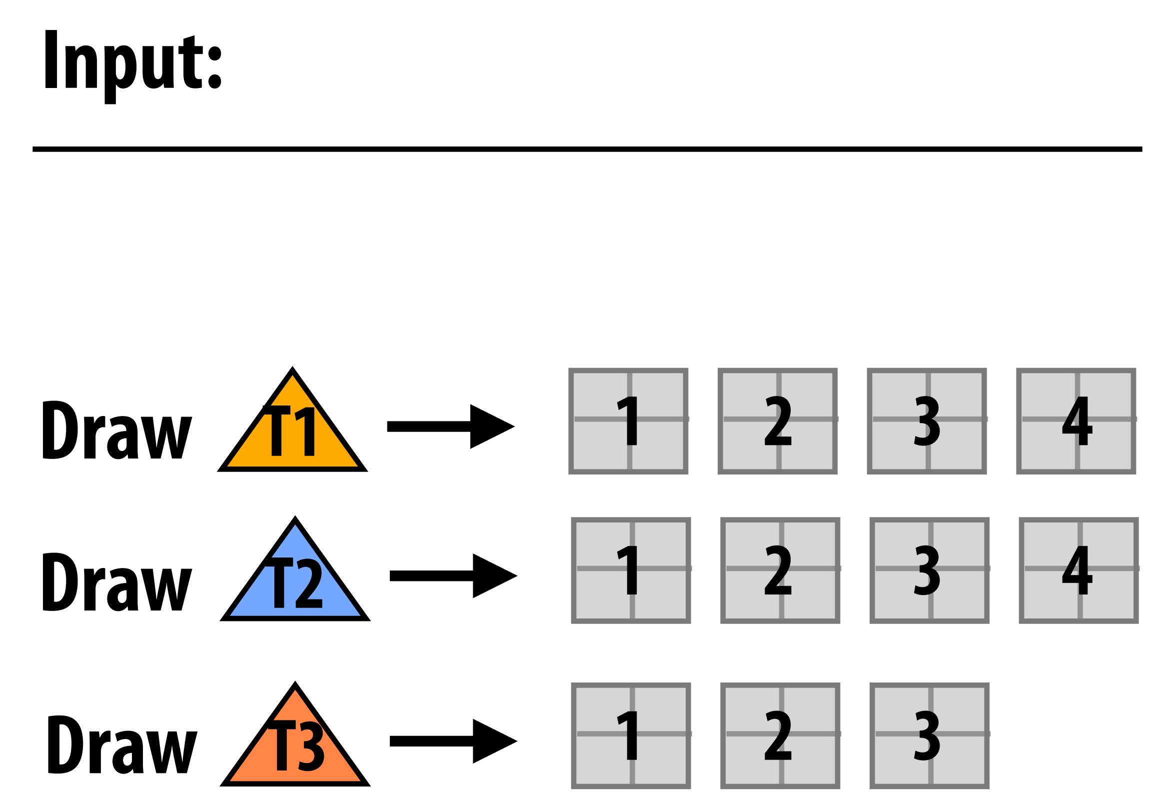
Assume batch size is 2 for assignment to rasterizers.



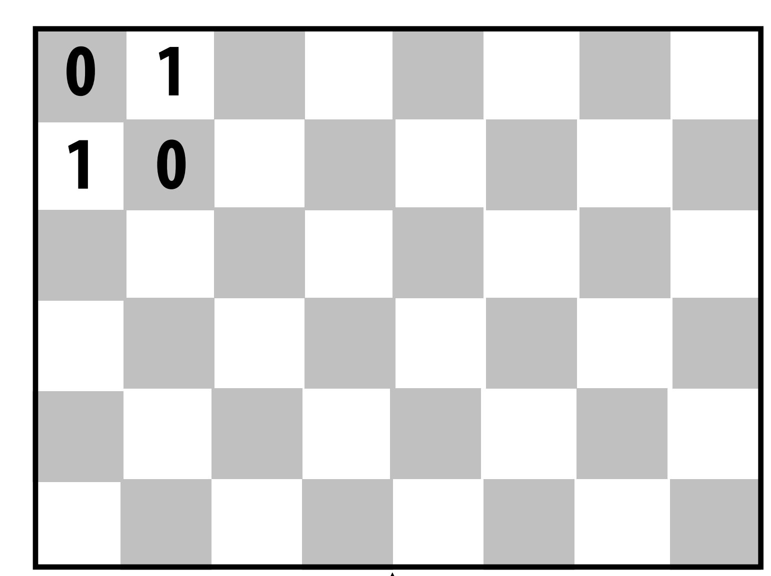
Interleaved render target

After geometry processing, first two processed triangles assigned to rast 0





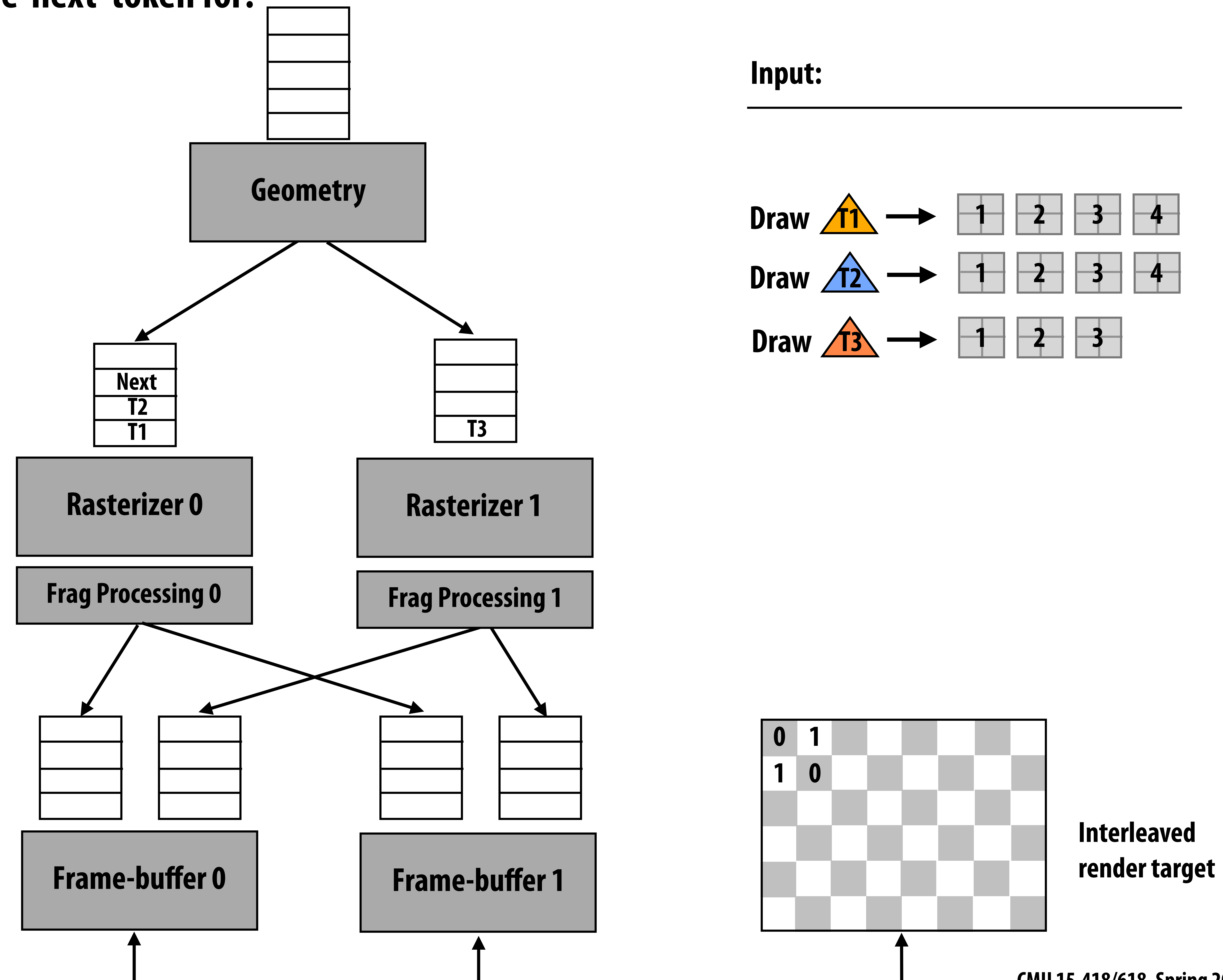
Assume batch size is 2 for assignment to rasterizers.



Interleaved render target

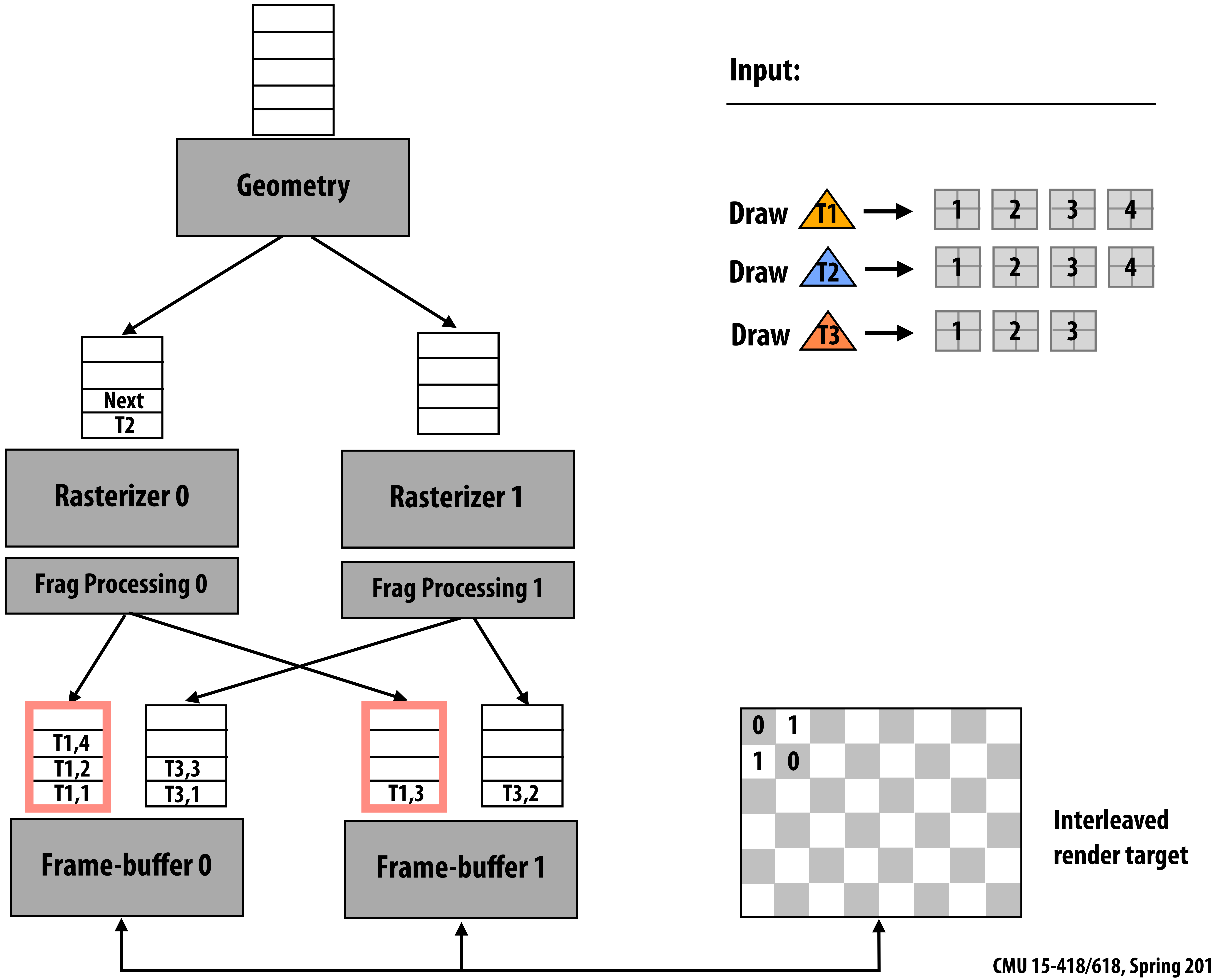
Assign next triangle to rast 1 (round robin policy, batch size = 2)

Q. What is the 'next' token for?



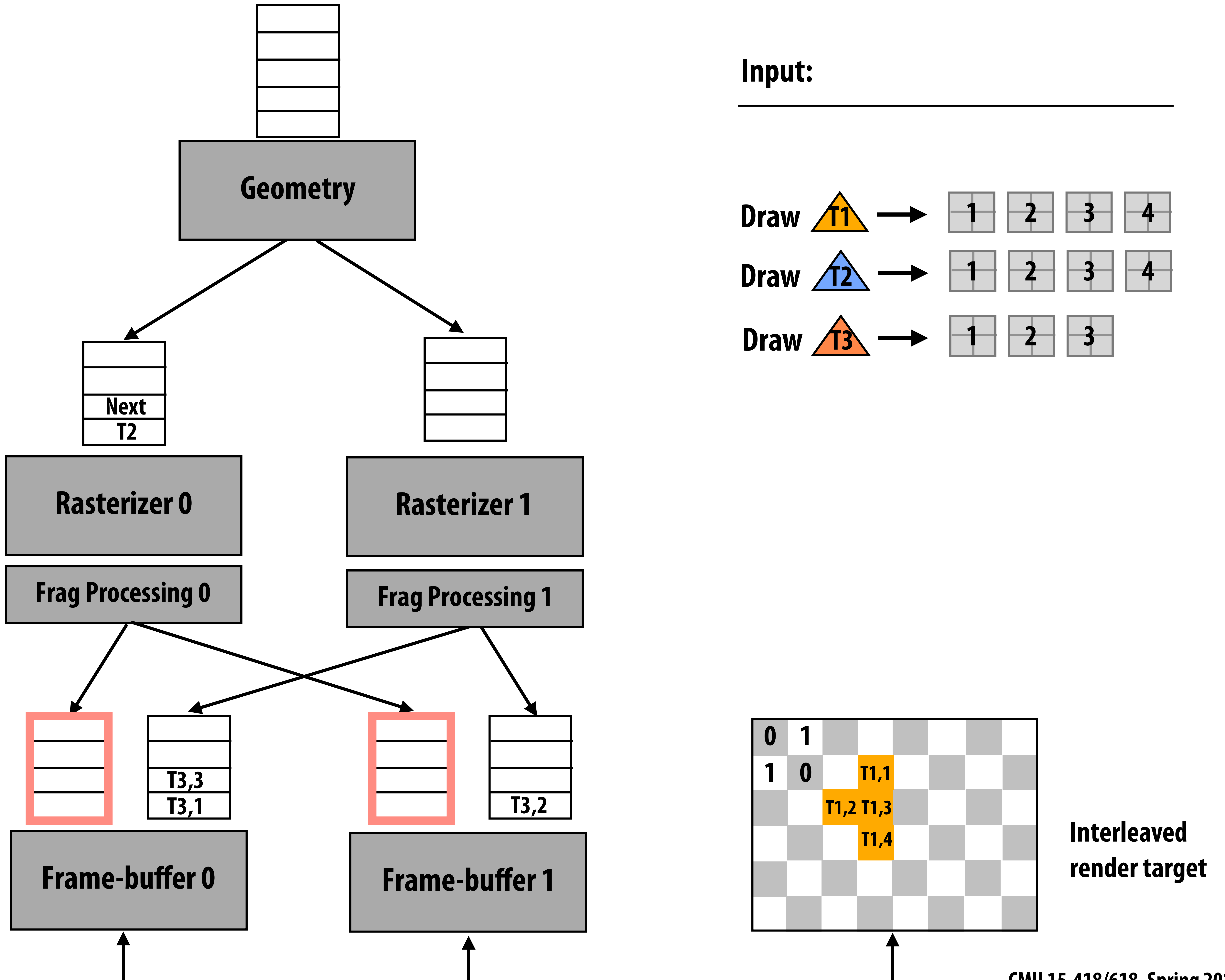
Rast 0 and rast 1 can process T1 and T3 simultaneously

(Shaded fragments enqueued in frame-buffer unit input queues)

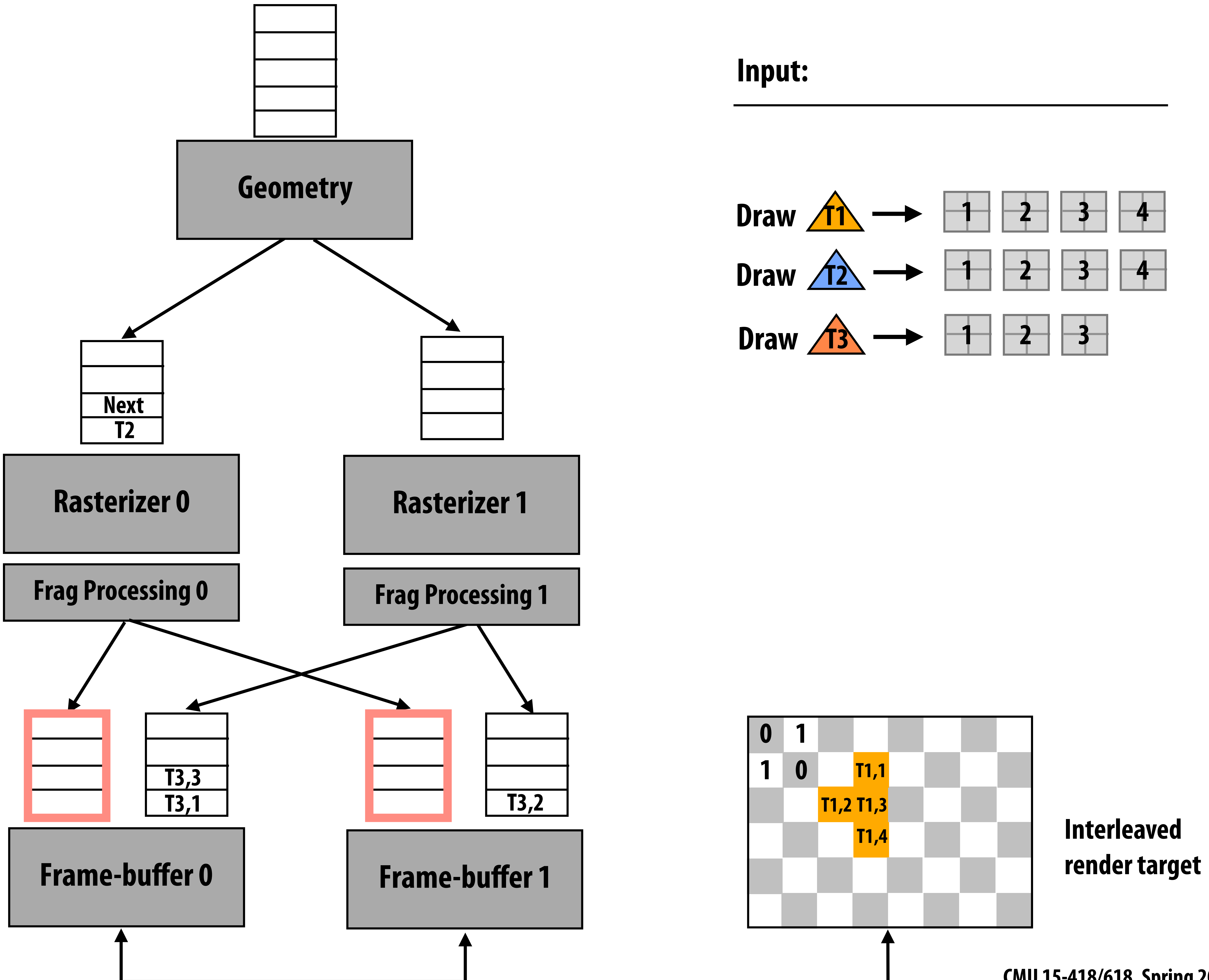


FB 0 and FB 1 can simultaneously process fragments from rast 0

(Notice updates to frame buffer)

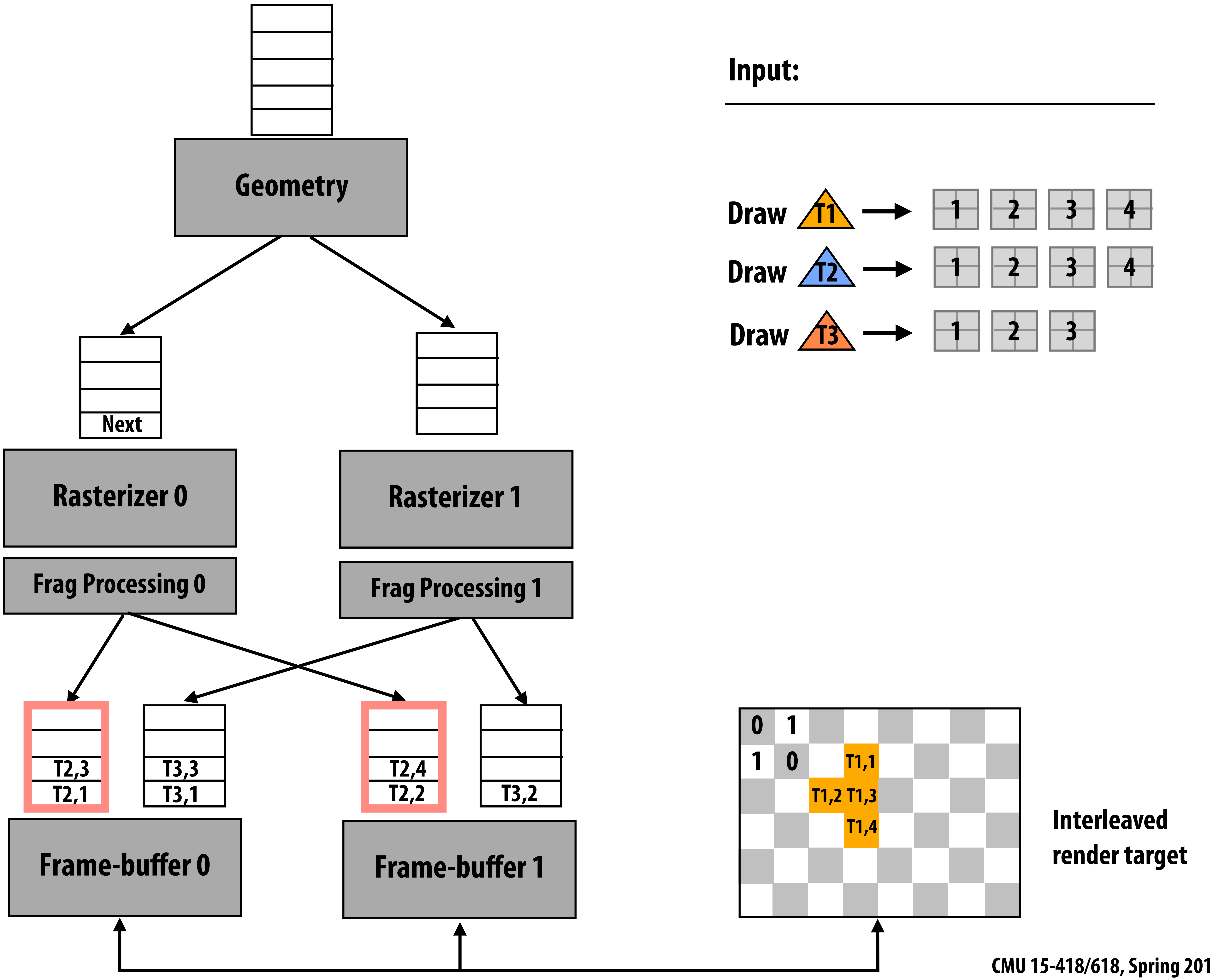


Fragments from T3 cannot be processed yet. Why?

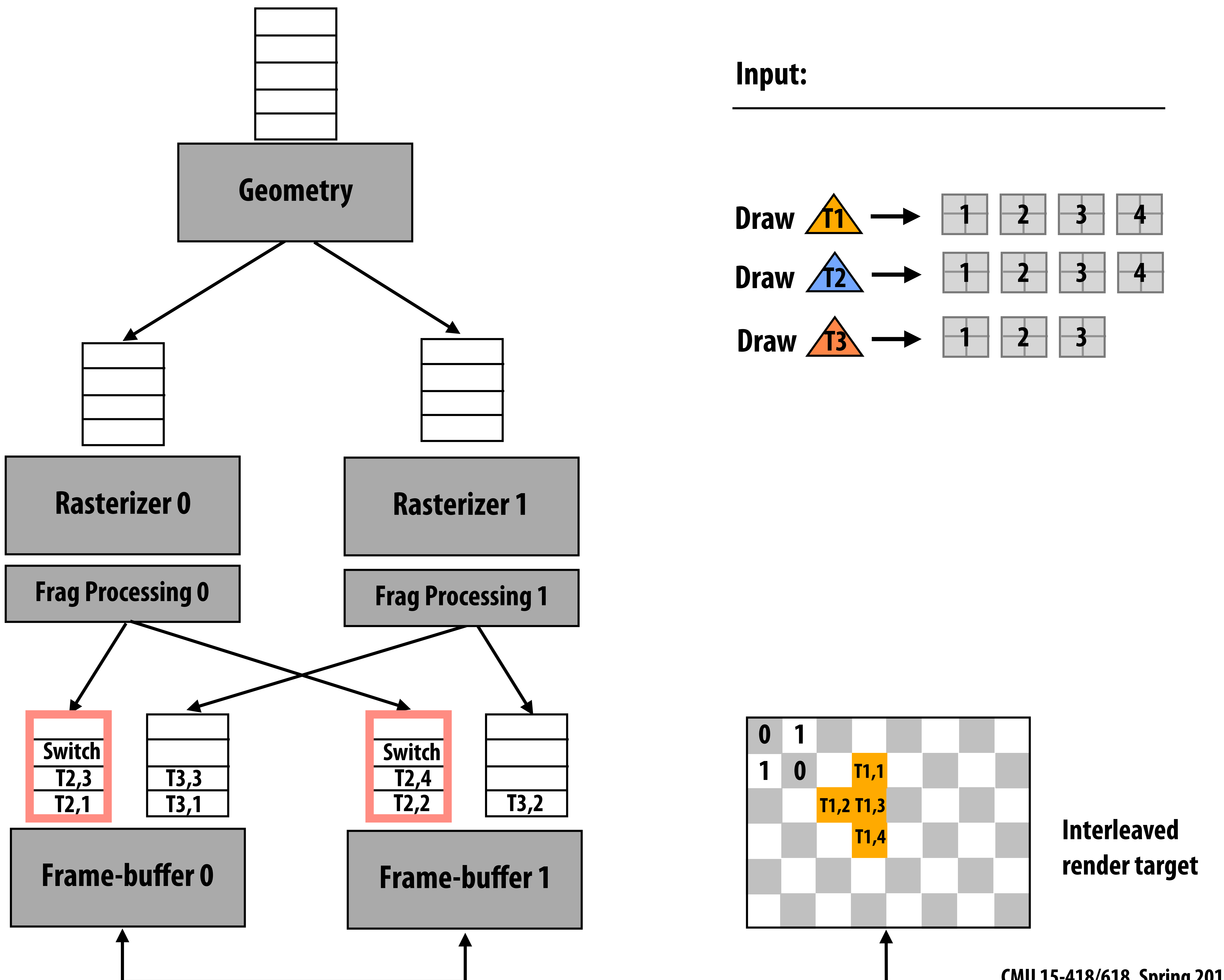


Rast 0 processes T2

(Shaded fragments enqueued in frame-buffer unit input queues)

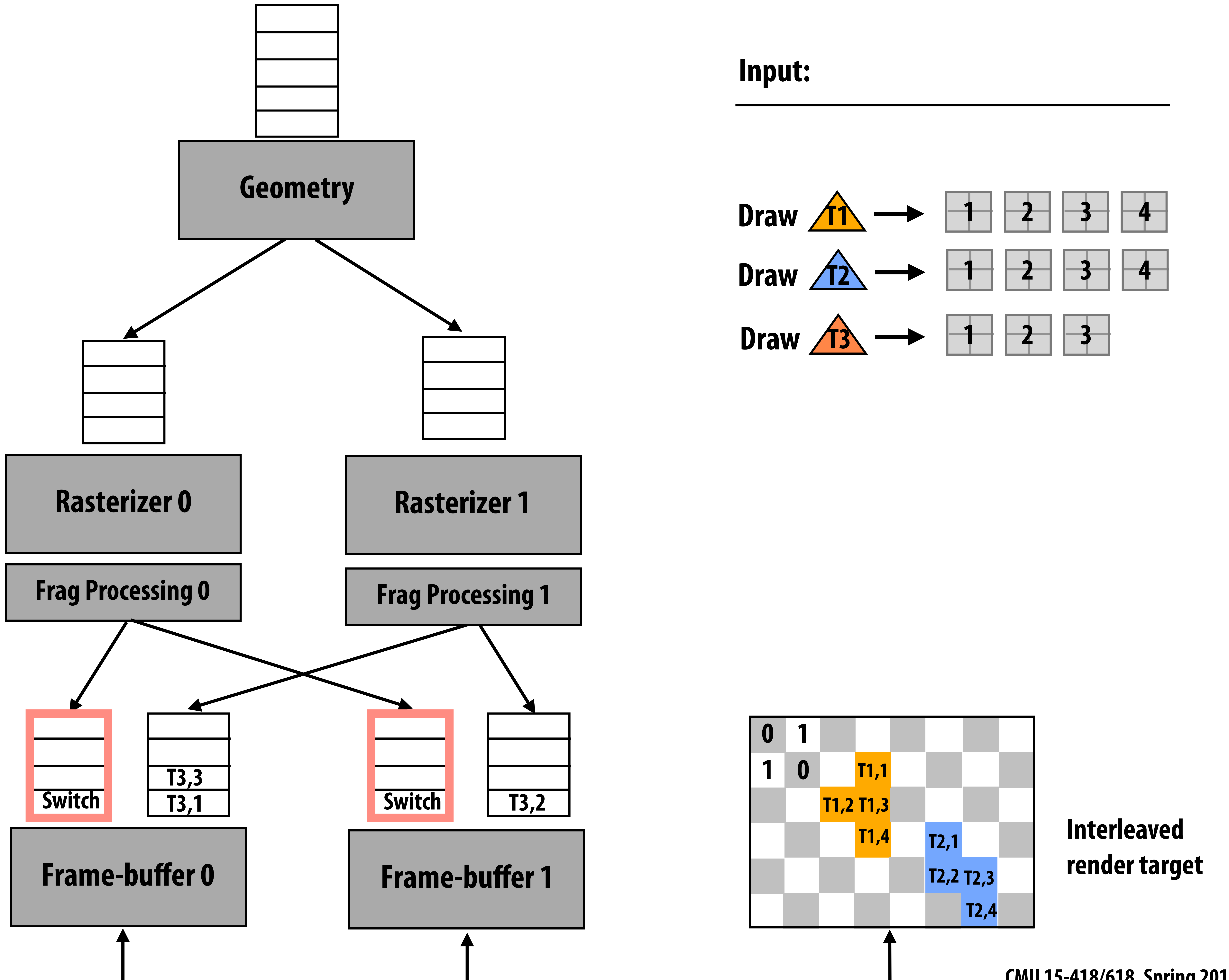


Rast 0 broadcasts 'next' token to all frame-buffer units

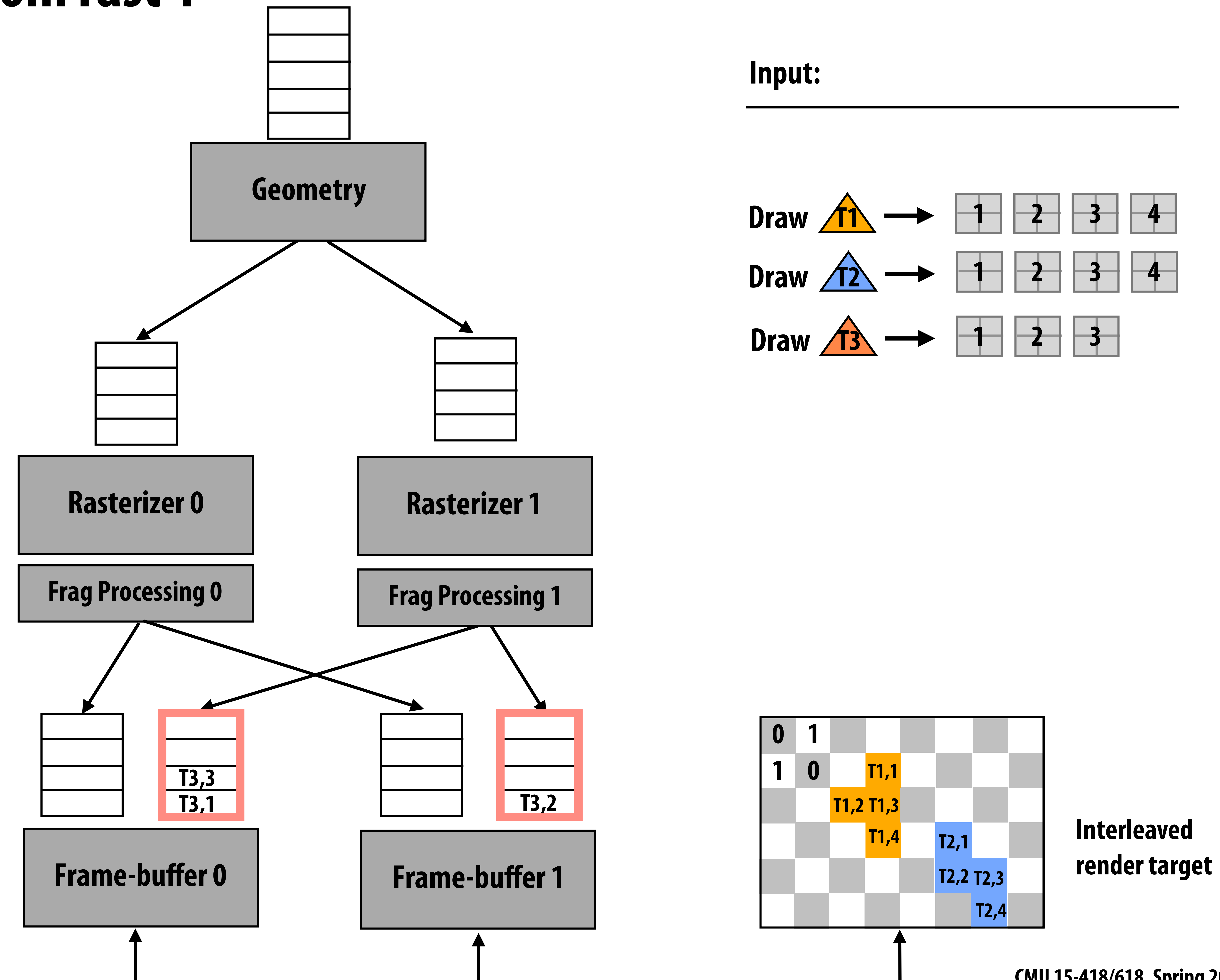


FB 0 and FB 1 can simultaneously process fragments from rast 0

(Notice updates to frame buffer)

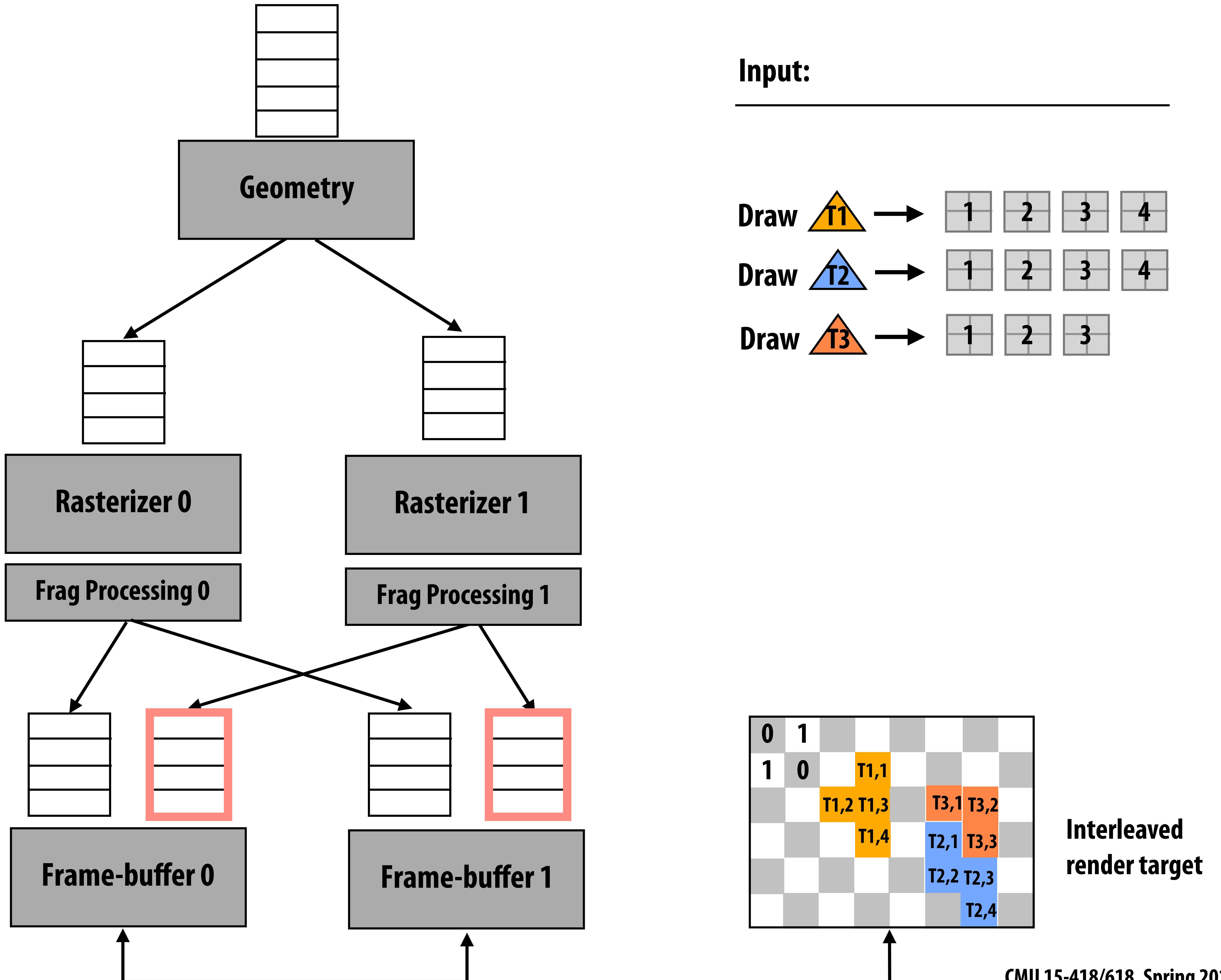


Switch token reached: frame-buffer units start processing input from rast 1



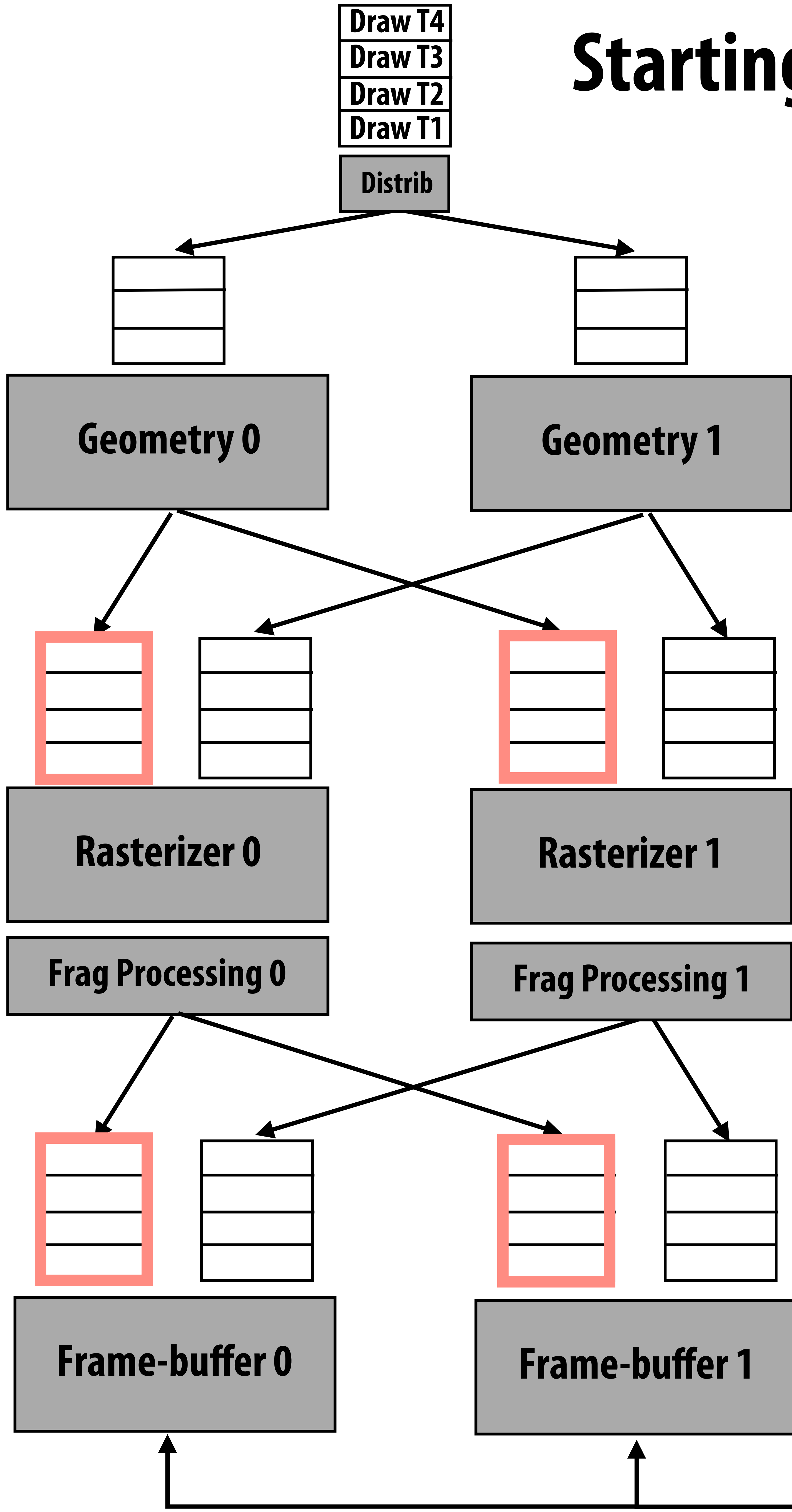
FB 0 and FB 1 can simultaneously process fragments from rast 1

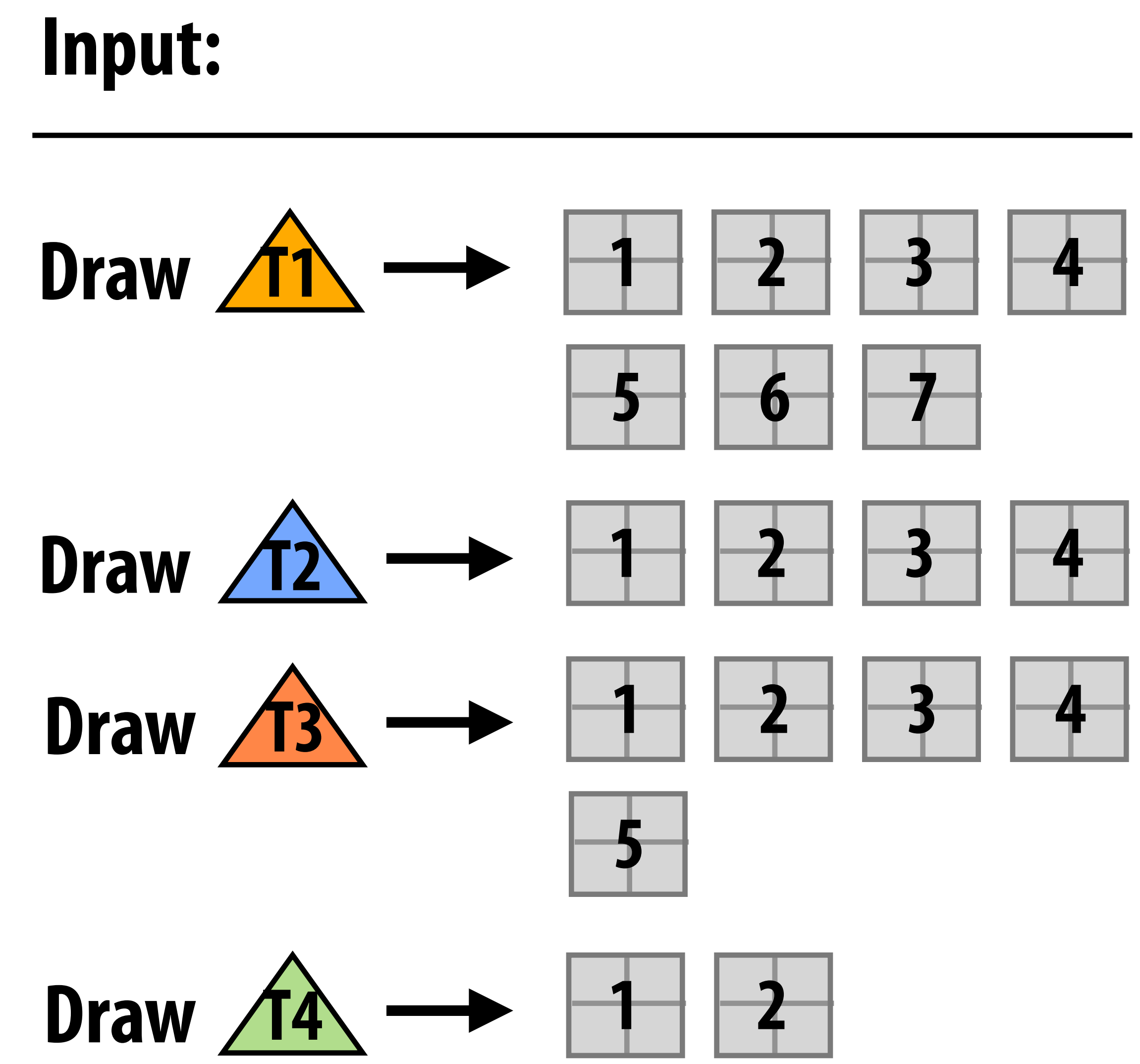
(Notice updates to frame buffer)



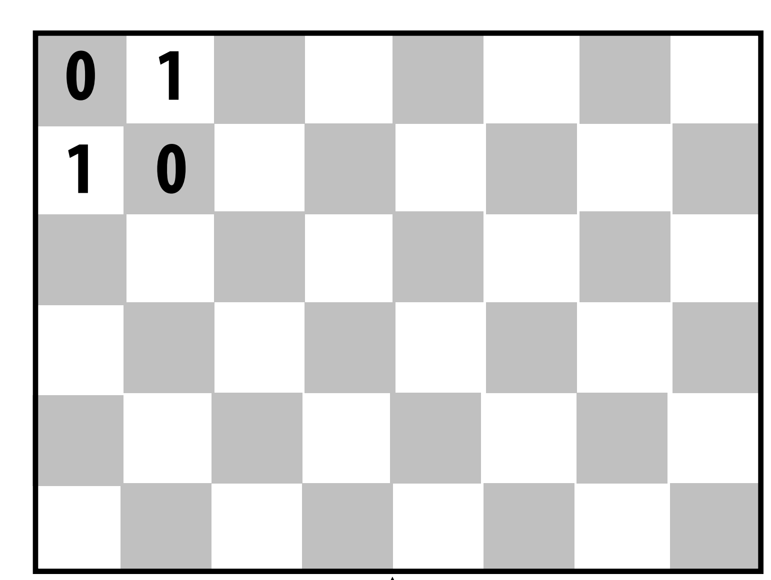
Extending to parallel geometry units

Starting state: commands enqueued



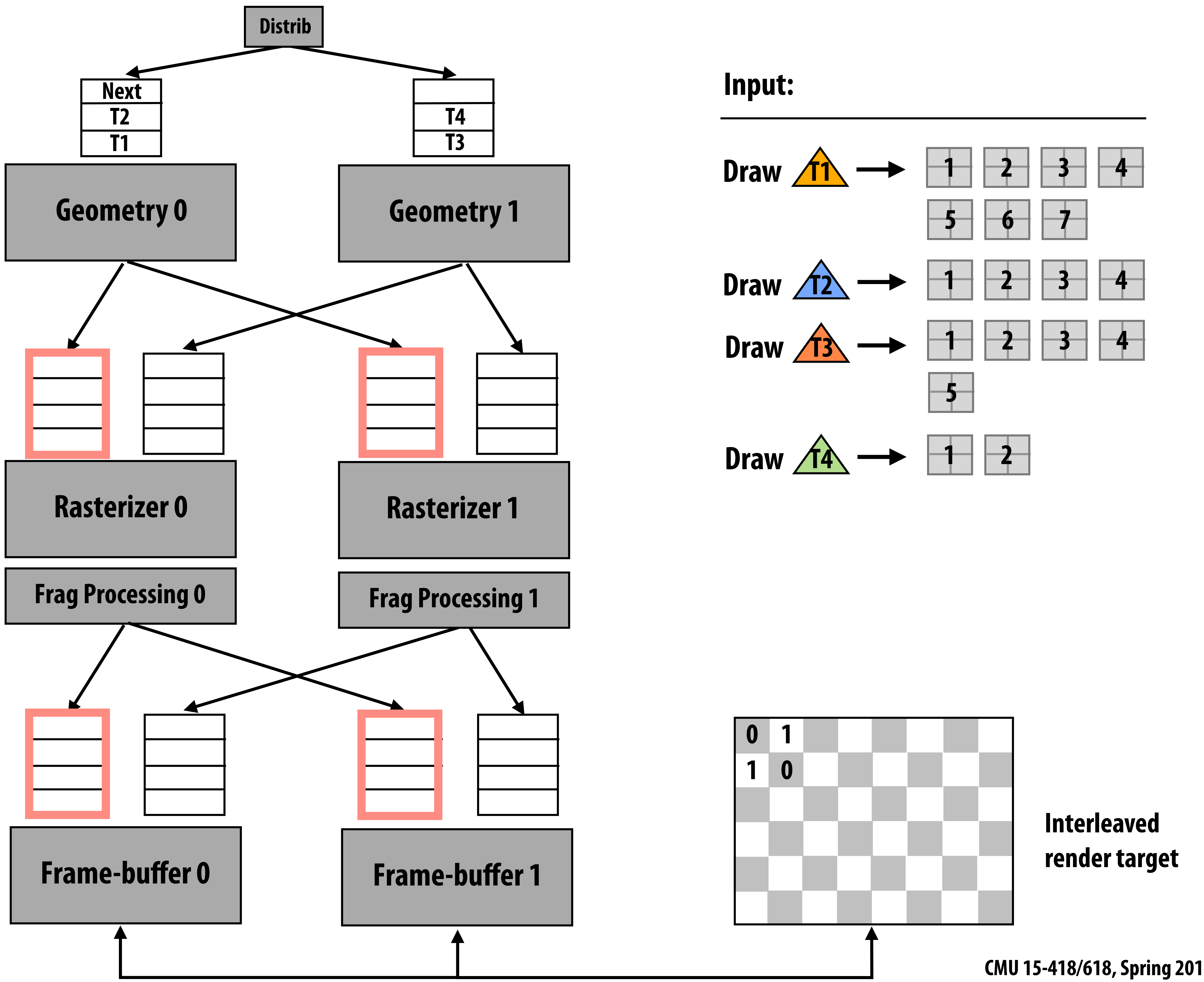


Assume batch size is 2 for assignment to geom units and to rasterizers.



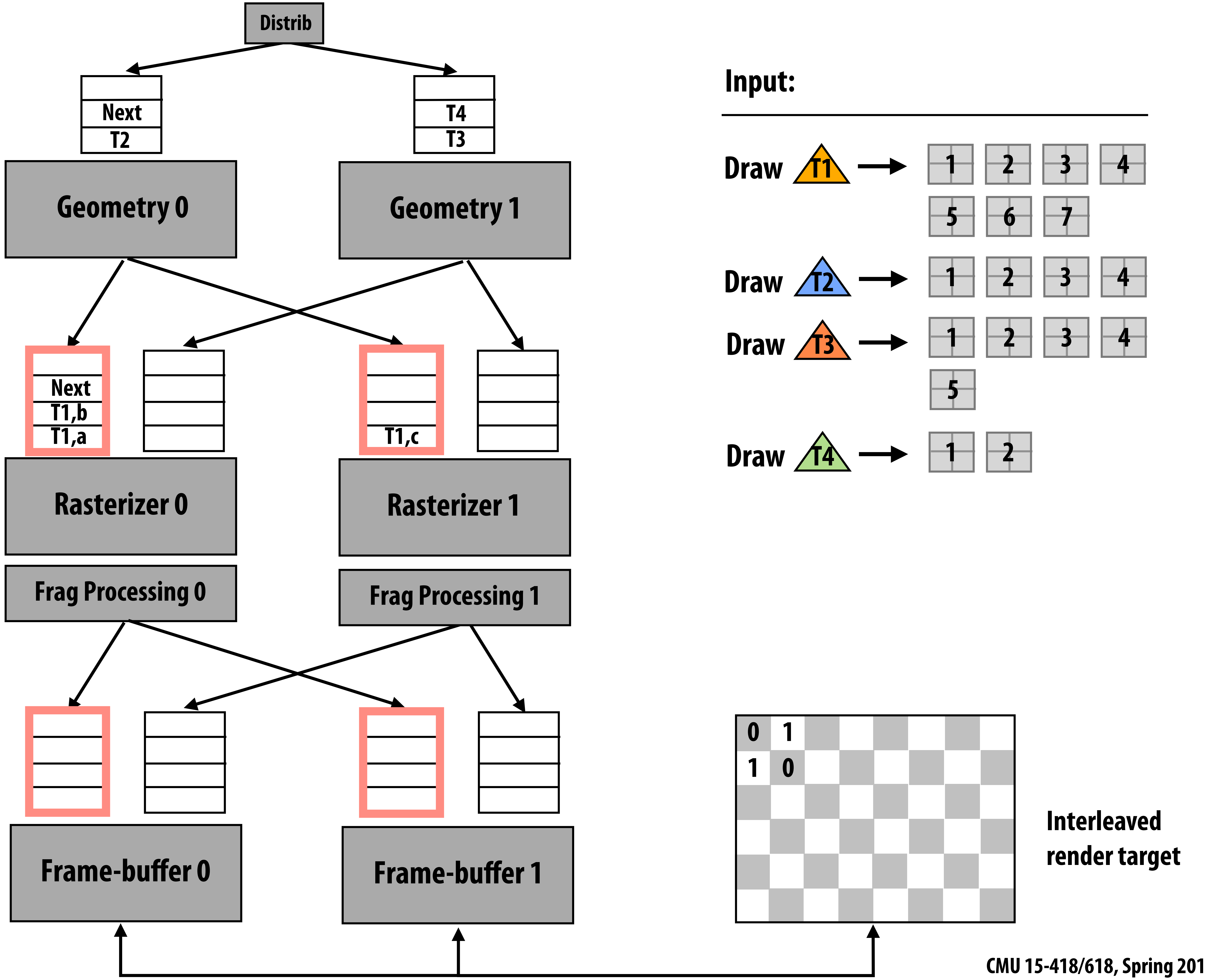
Interleaved render target

Distribute triangles to geom units round-robin (batches of 2)



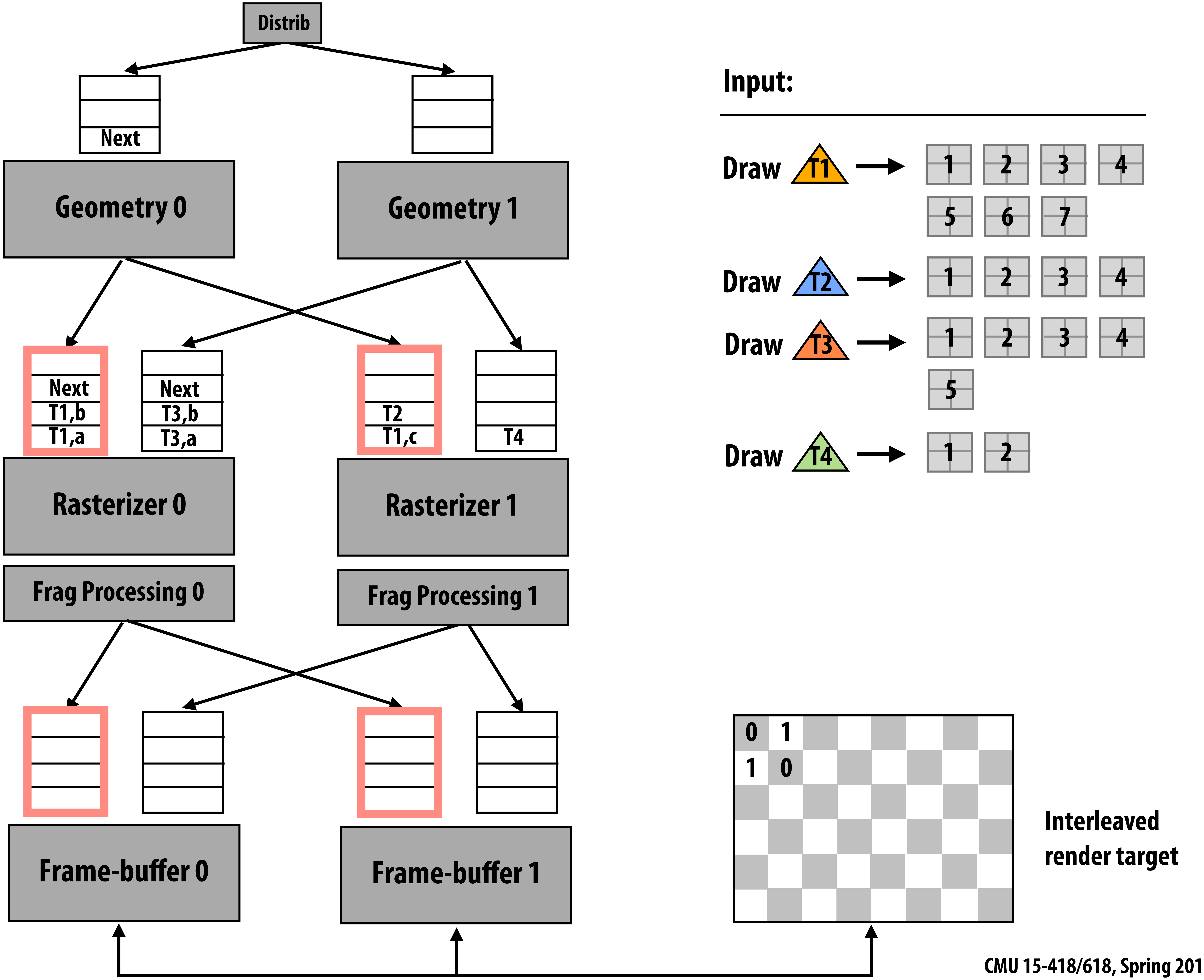
Geom 0 and geom 1 process triangles in parallel

(Results after T1 processed are shown. Note big triangle T1 broken into multiple work items. [Eldridge et al.])

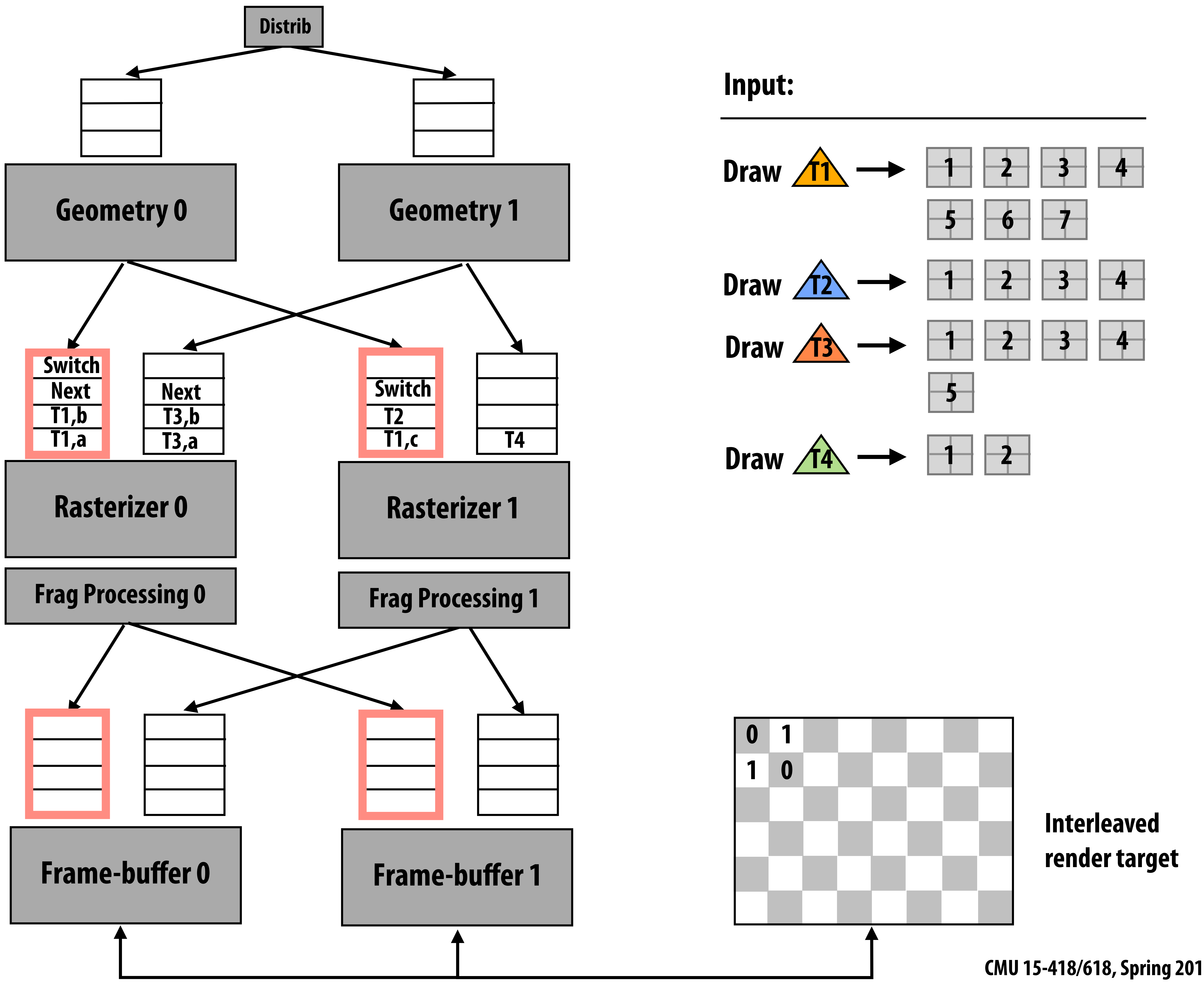


Geom 0 and geom 1 process triangles in parallel

(Triangles enqueued in rast input queues. Note big triangles broken into multiple work items. [Eldridge et al.])

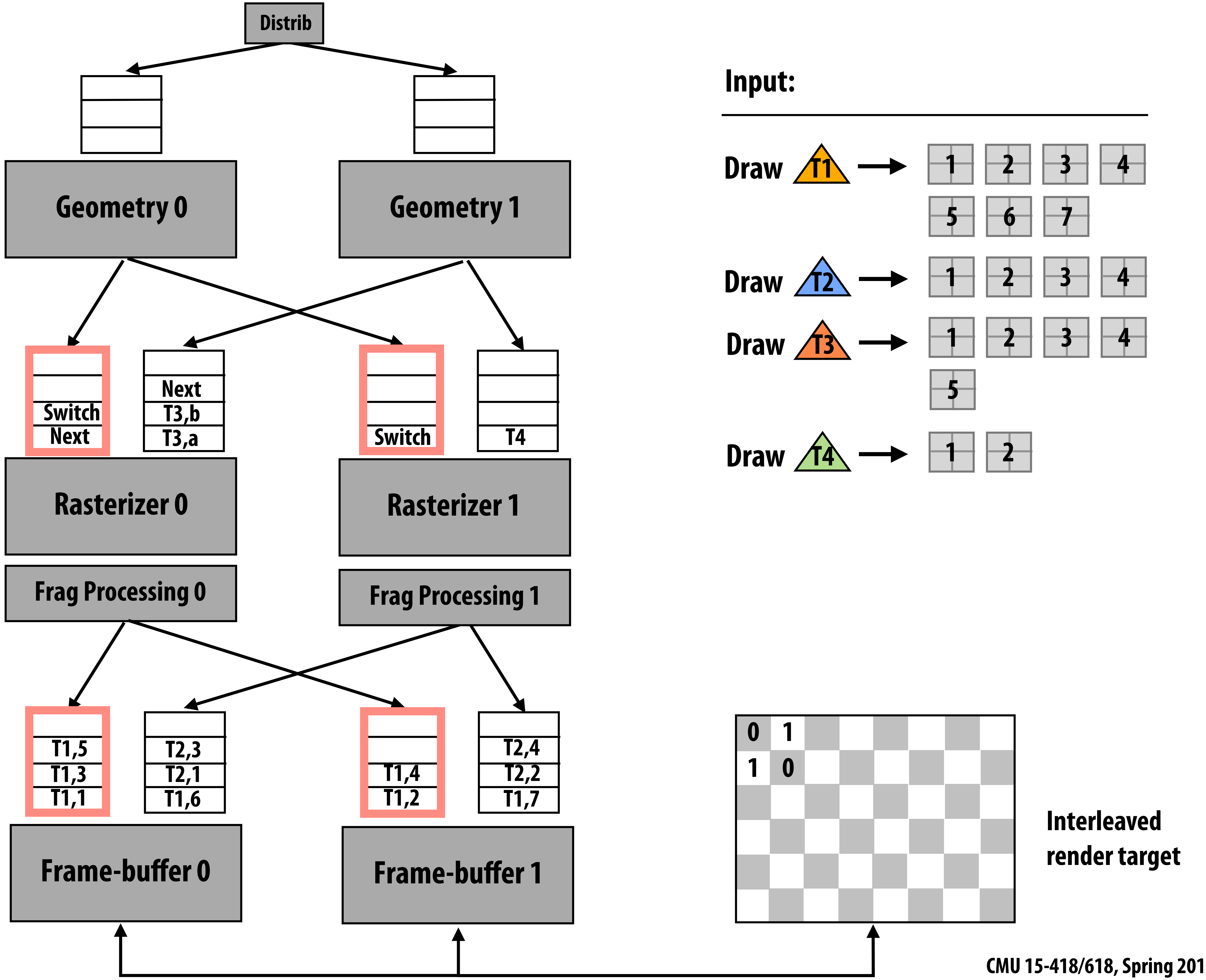


Geom 0 broadcasts 'next' token to rasterizers

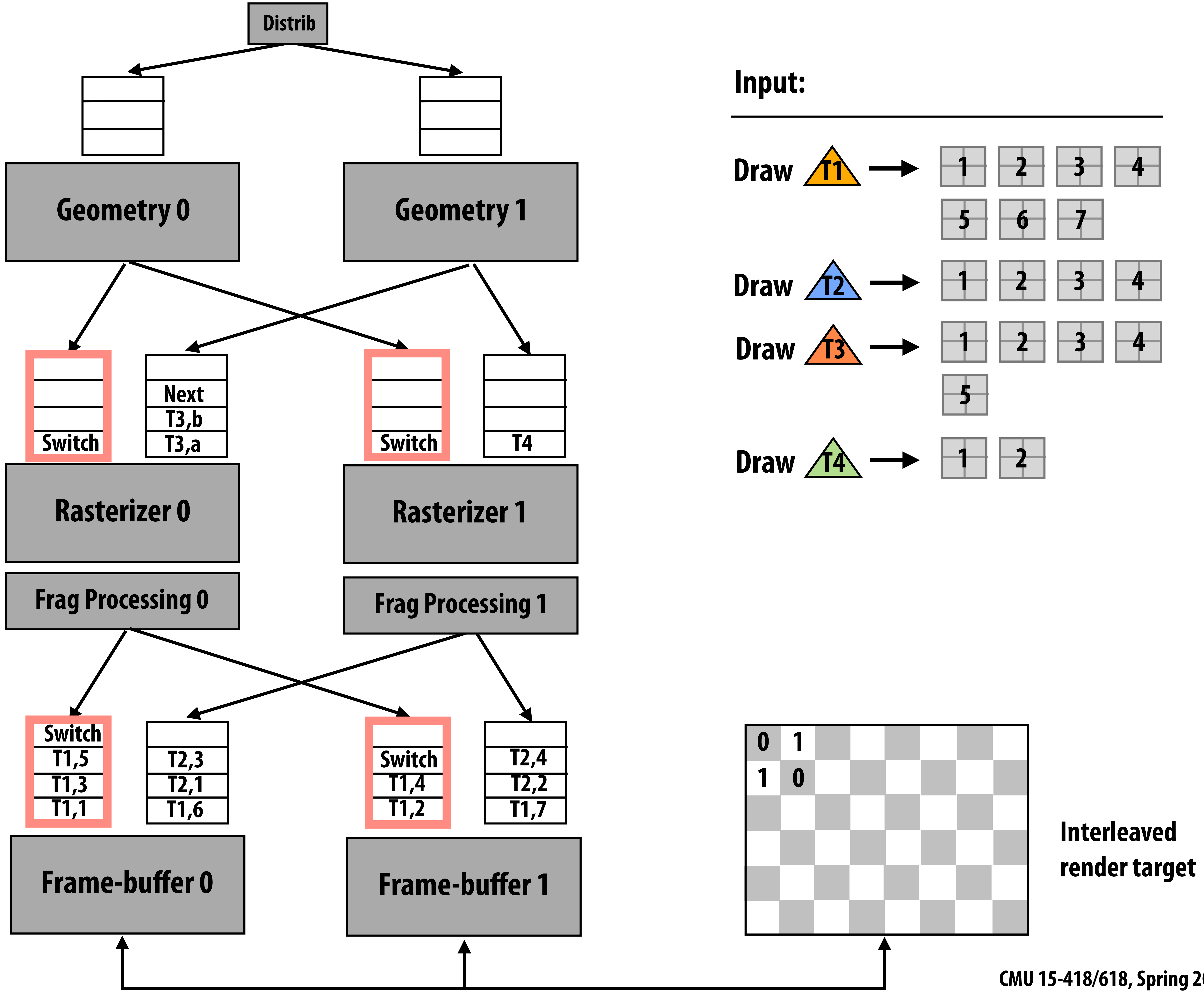


Rast 0 and rast 1 process triangles from geom 0 in parallel

(Shaded fragments enqueued in frame-buffer unit input queues)

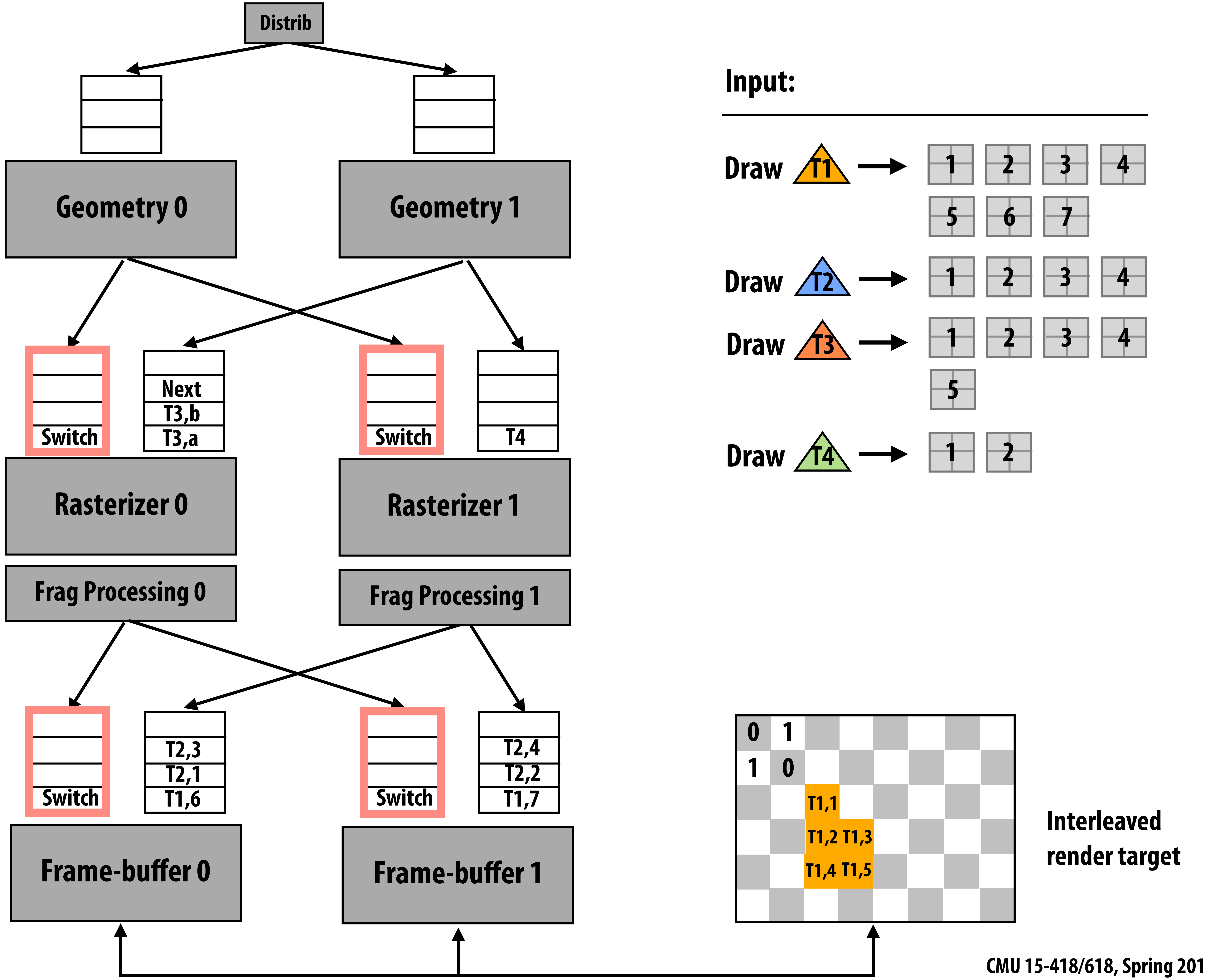


Rast 0 broadcasts 'next' token to FB units (end of geom 0, rast 0)

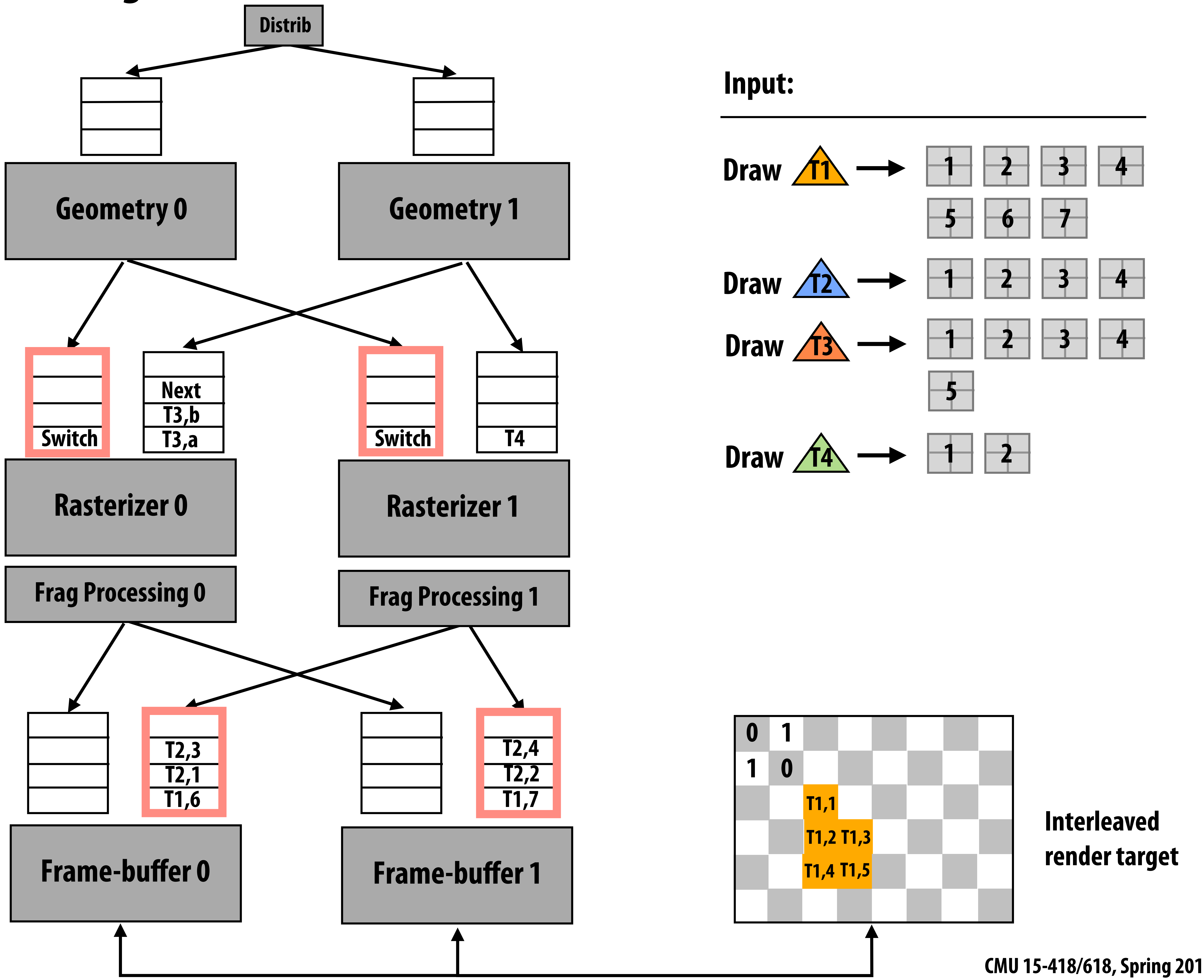


Frame-buffer units process frags from (geom 0, rast 0) in parallel

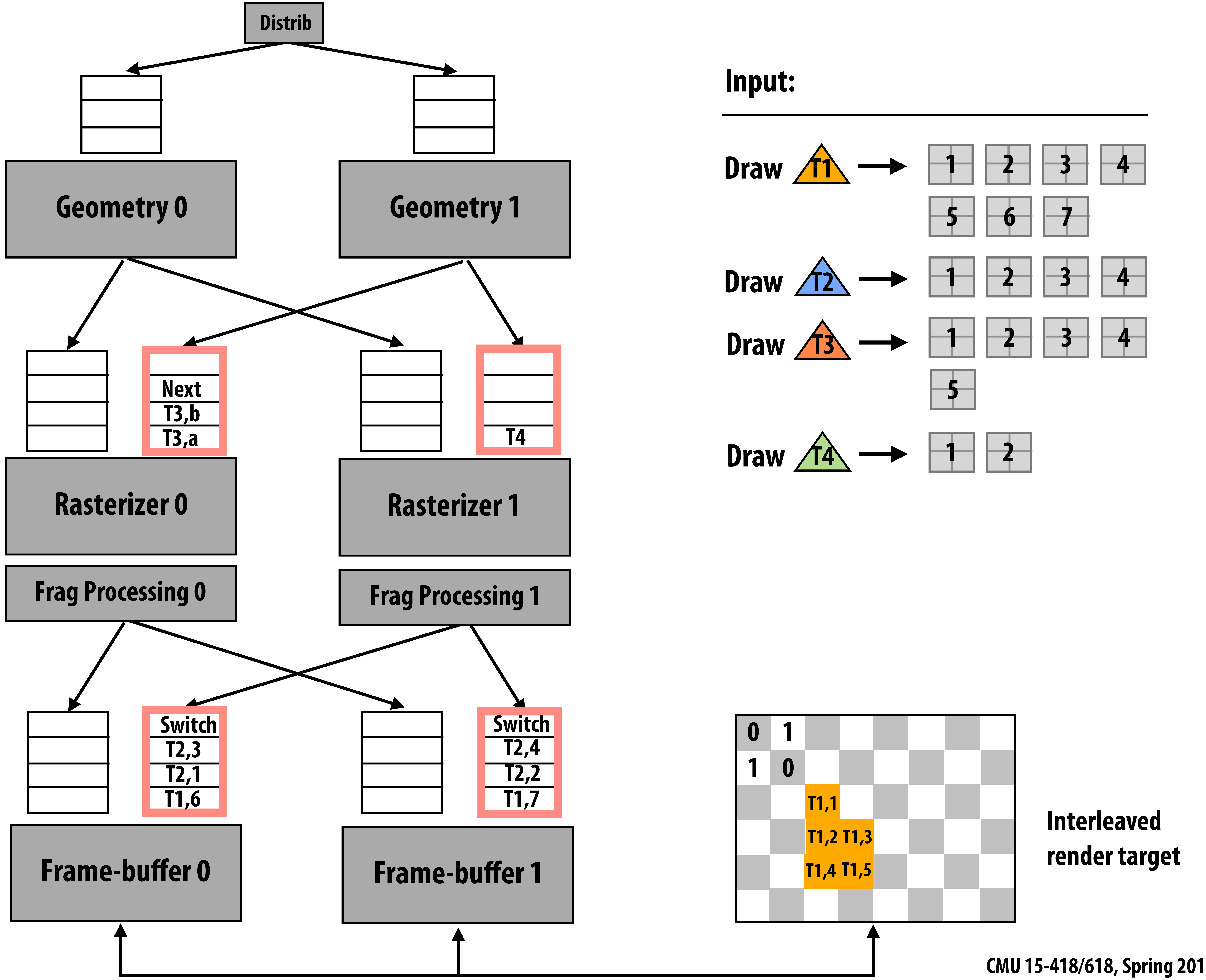
(Notice updates to frame buffer)



“End of rast 0” token reached by FB: FB units start processing input from rast 1 (fragments from geom 0, rast 1)

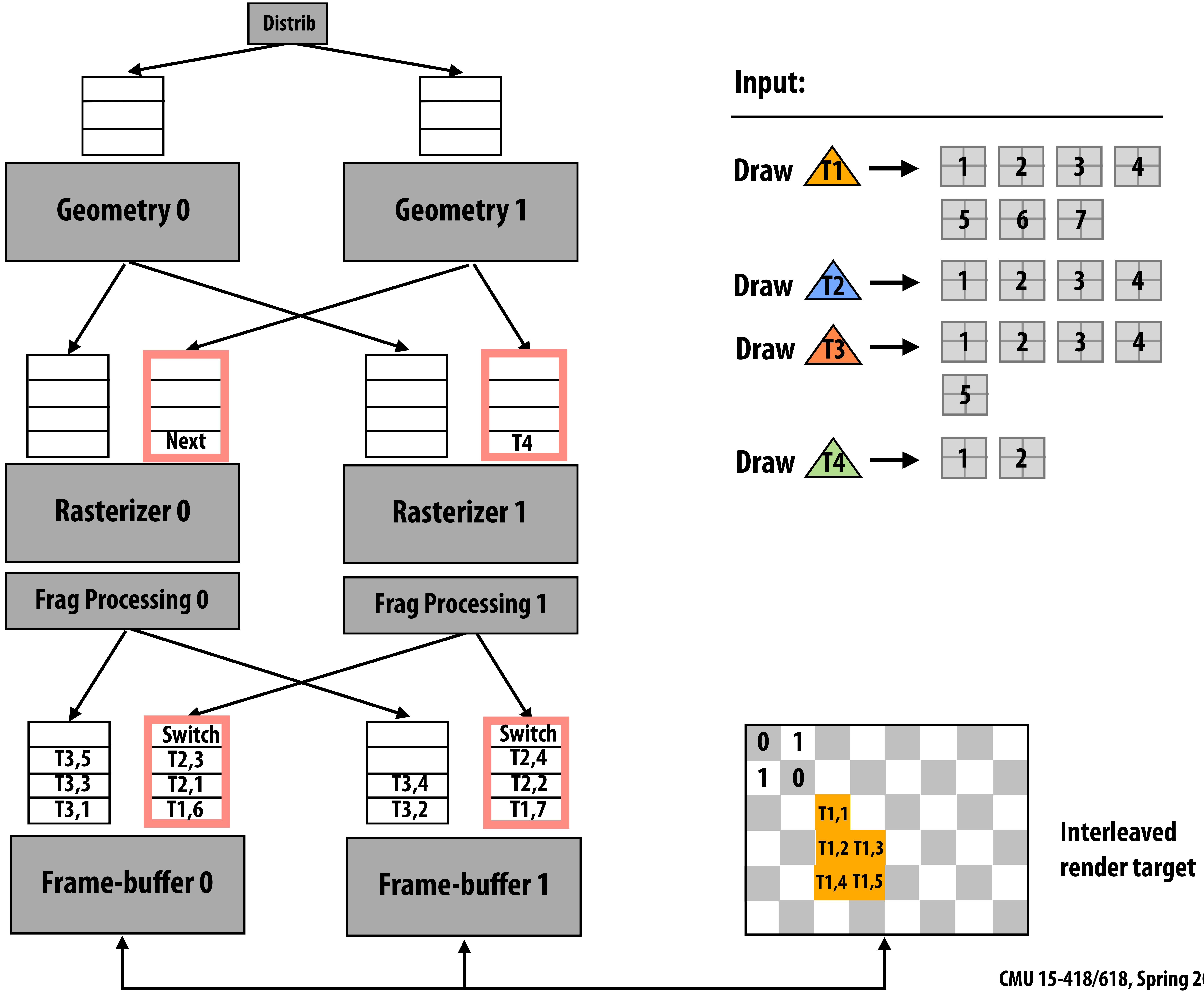


“End of geom 0” token reached by rast units: rast units start processing input from geom 1 (note “end of geom 0, rast 1” token sent to rast input queues)

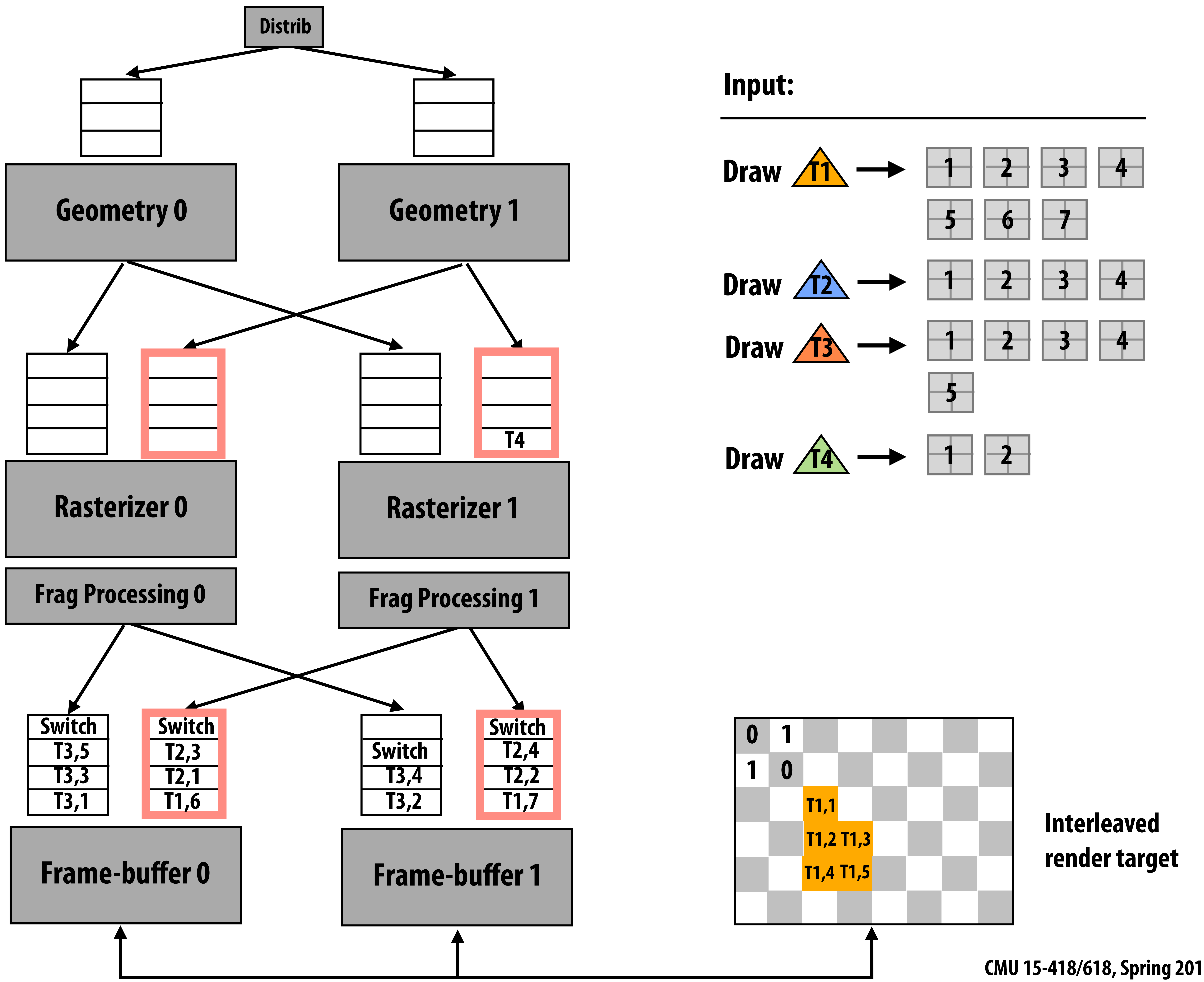


Rast 0 processes triangles from geom 1

(Note Rast 1 has work to do, but cannot make progress because its output queues are full)

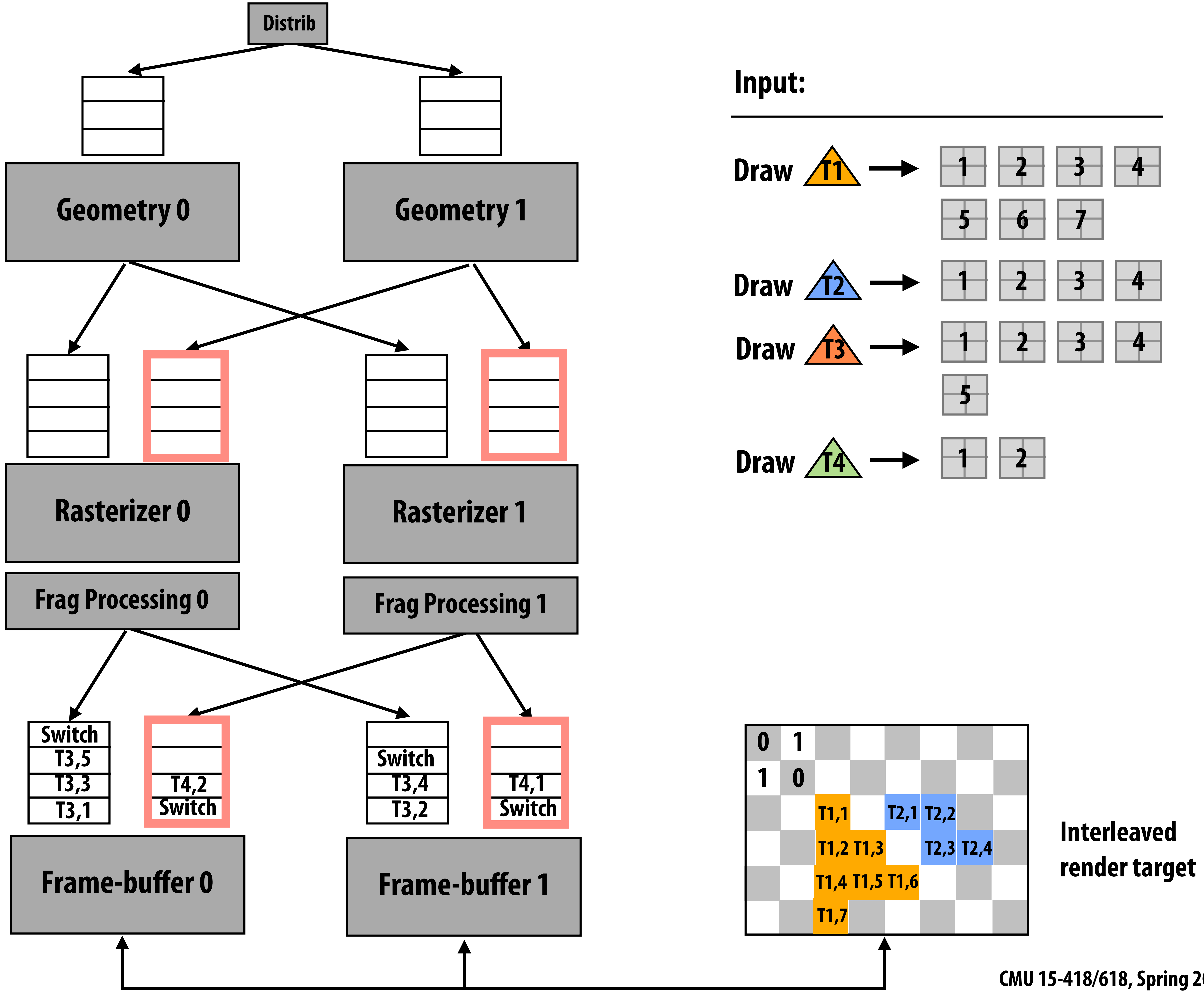


Rast 0 broadcasts "end of geom 1, rast 0" token to frame-buffer units

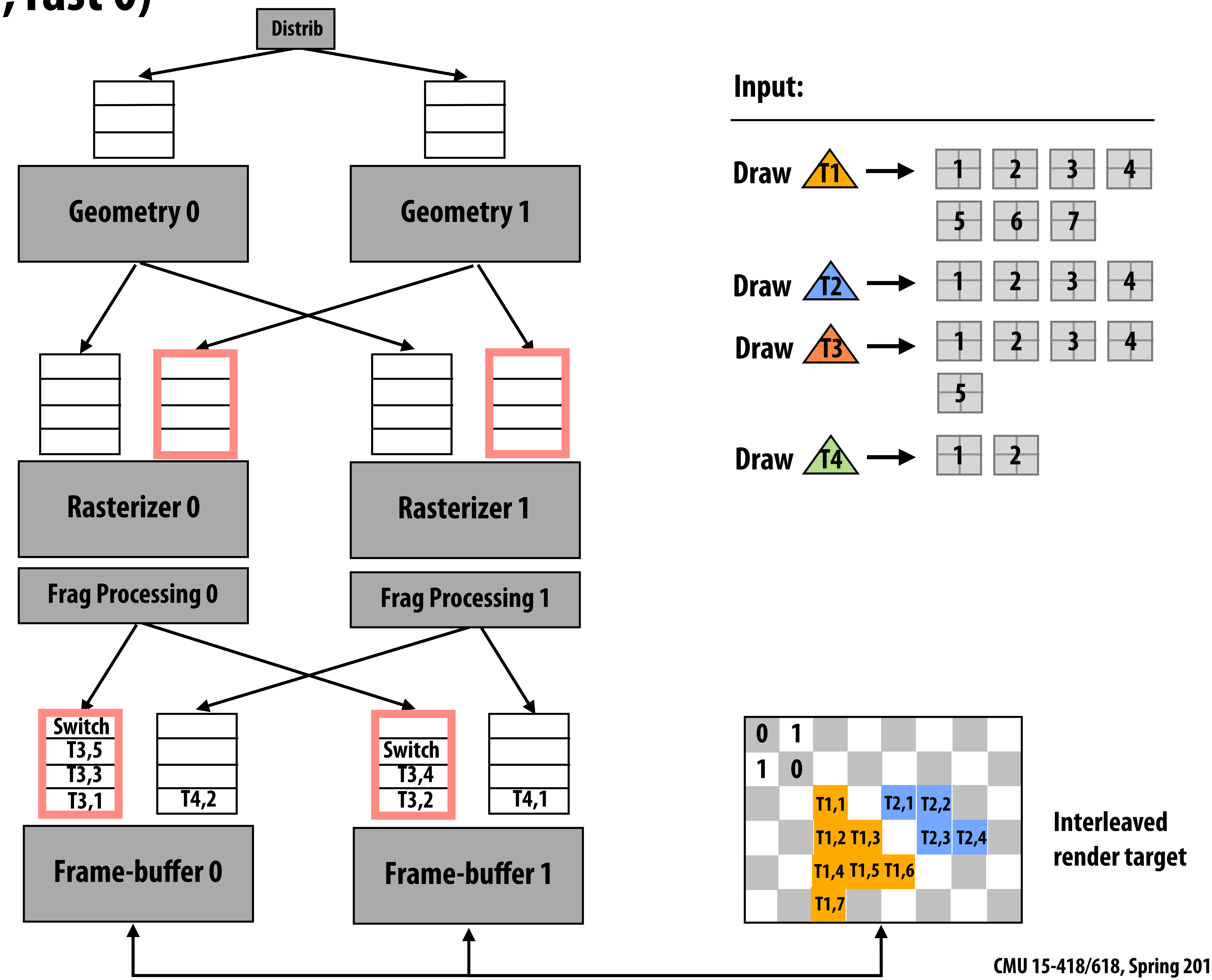


Frame-buffer units process frags from (geom 0, rast 1) in parallel

(Notice updates to frame buffer. Also notice rast 1 can now make progress since space has become available)

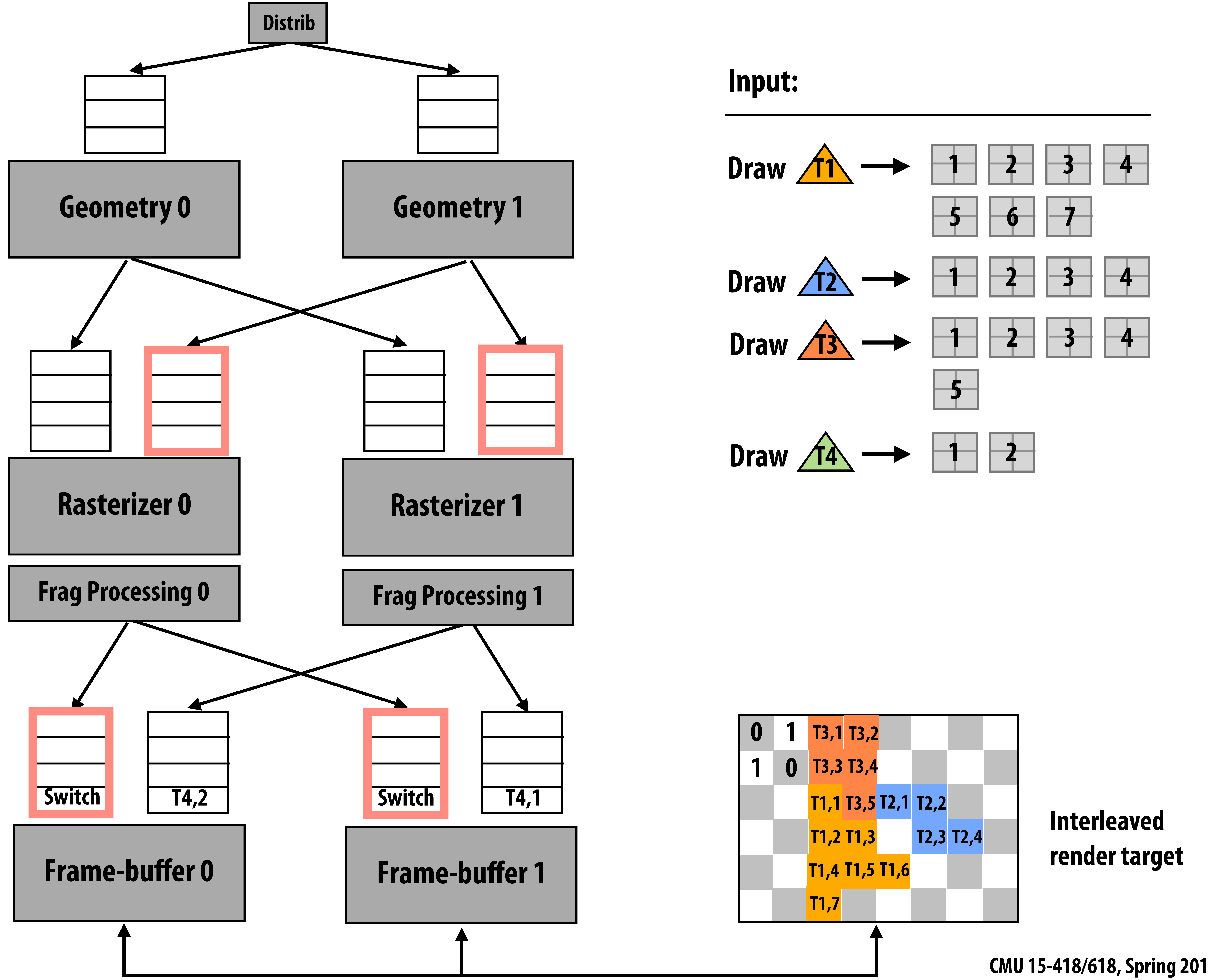


Switch token reached by FB: FB units start processing input from (geom 1, rast 0)

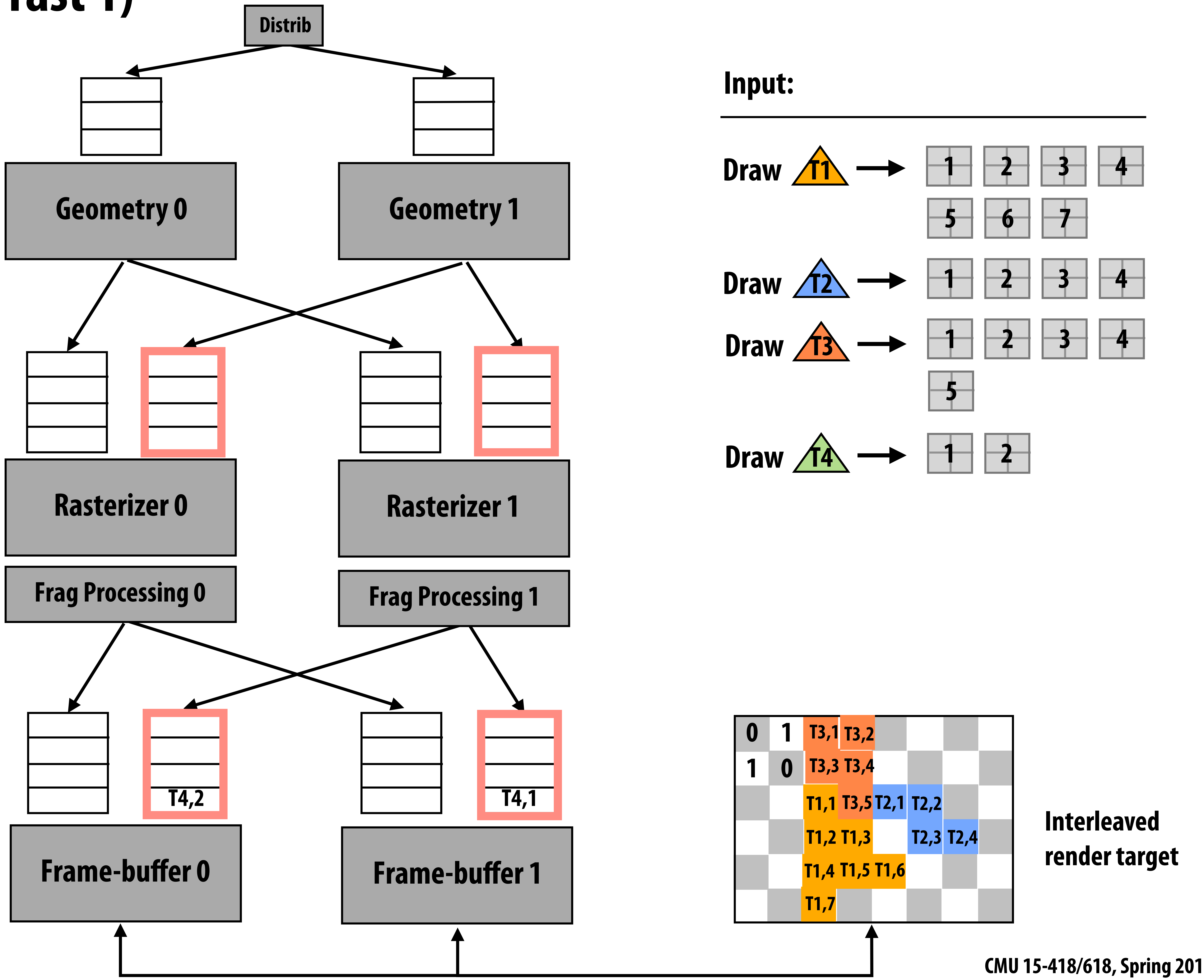


Frame-buffer units process frags from (geom 1, rast 0) in parallel

(Notice updates to frame buffer)

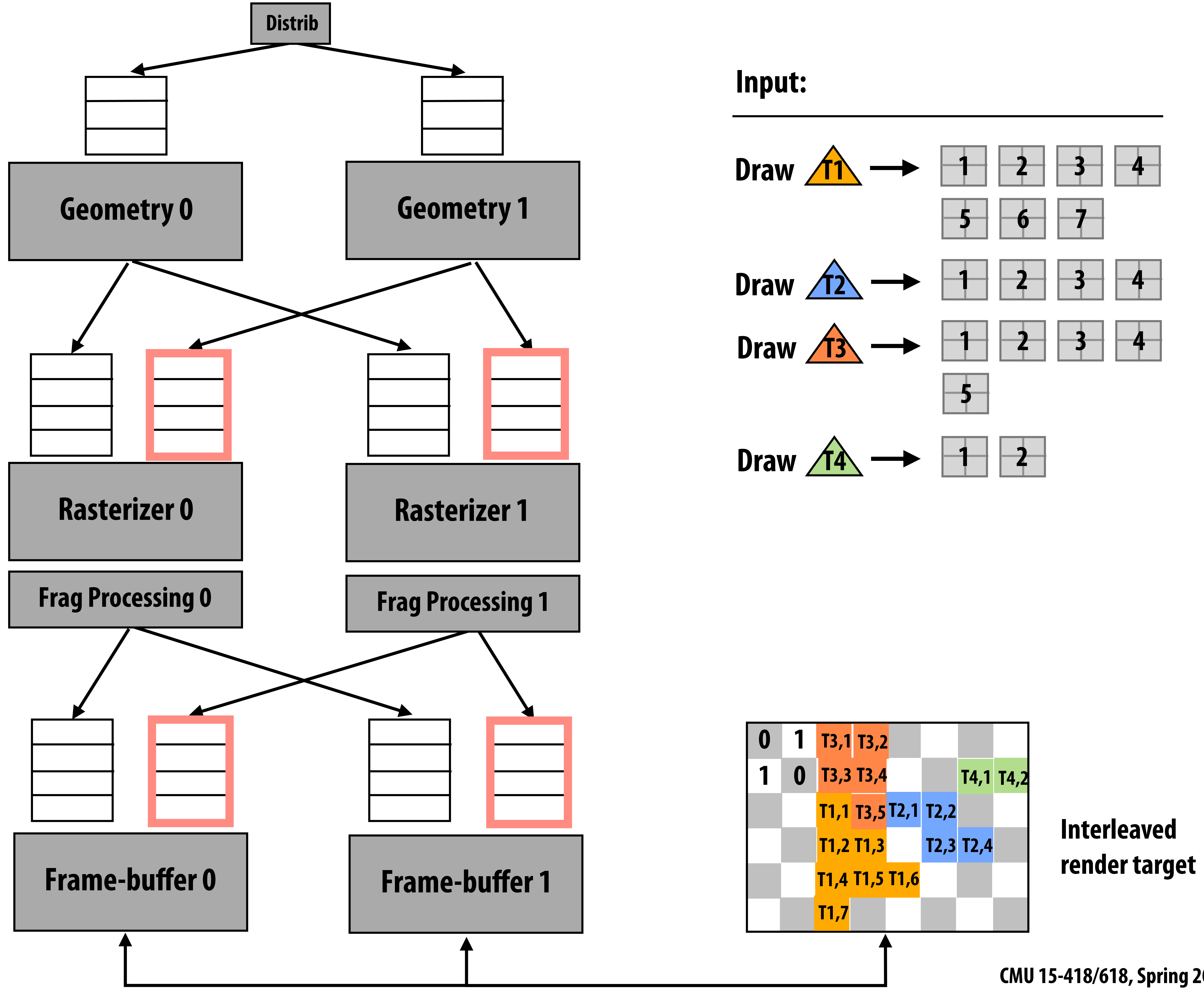


Switch token reached by FB: FB units start processing input from (geom 1, rast 1)



Frame-buffer units process frags from (geom 1, rast 1) in parallel

(Notice updates to frame buffer)



Summary: GPU accelerated 3D graphics

- Leverages parallel hardware units
- Leverages combination of programmable and fixed-function hardware units
- Fixed-function not just used for expensive arithmetic!
  - Data-compression
  - Computation scheduling
- Modern GPU's provide an extremely efficient implementation of the graphics pipeline abstraction