

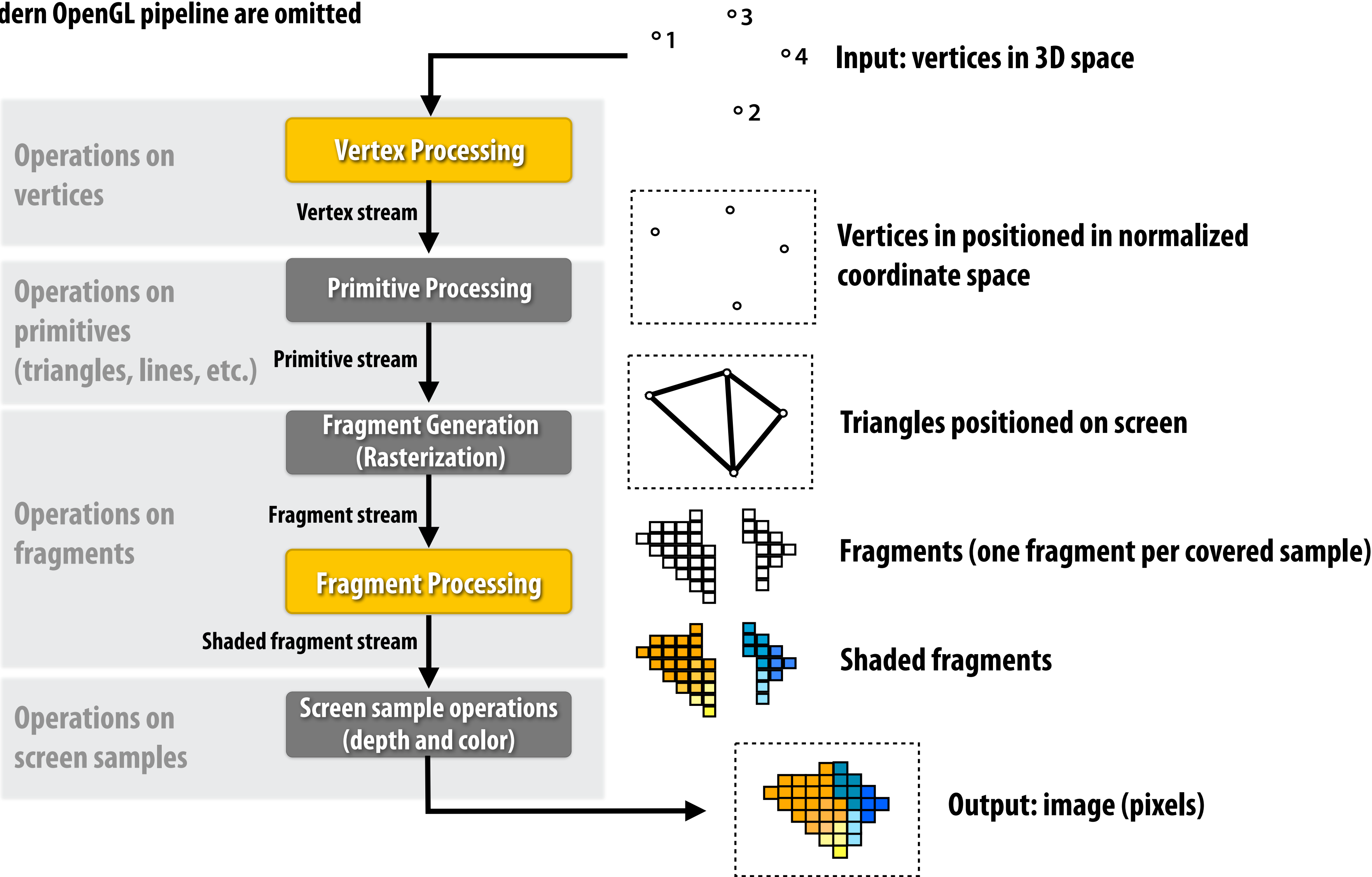
Lecture 14:

Scheduling the Graphics Pipeline on a GPU

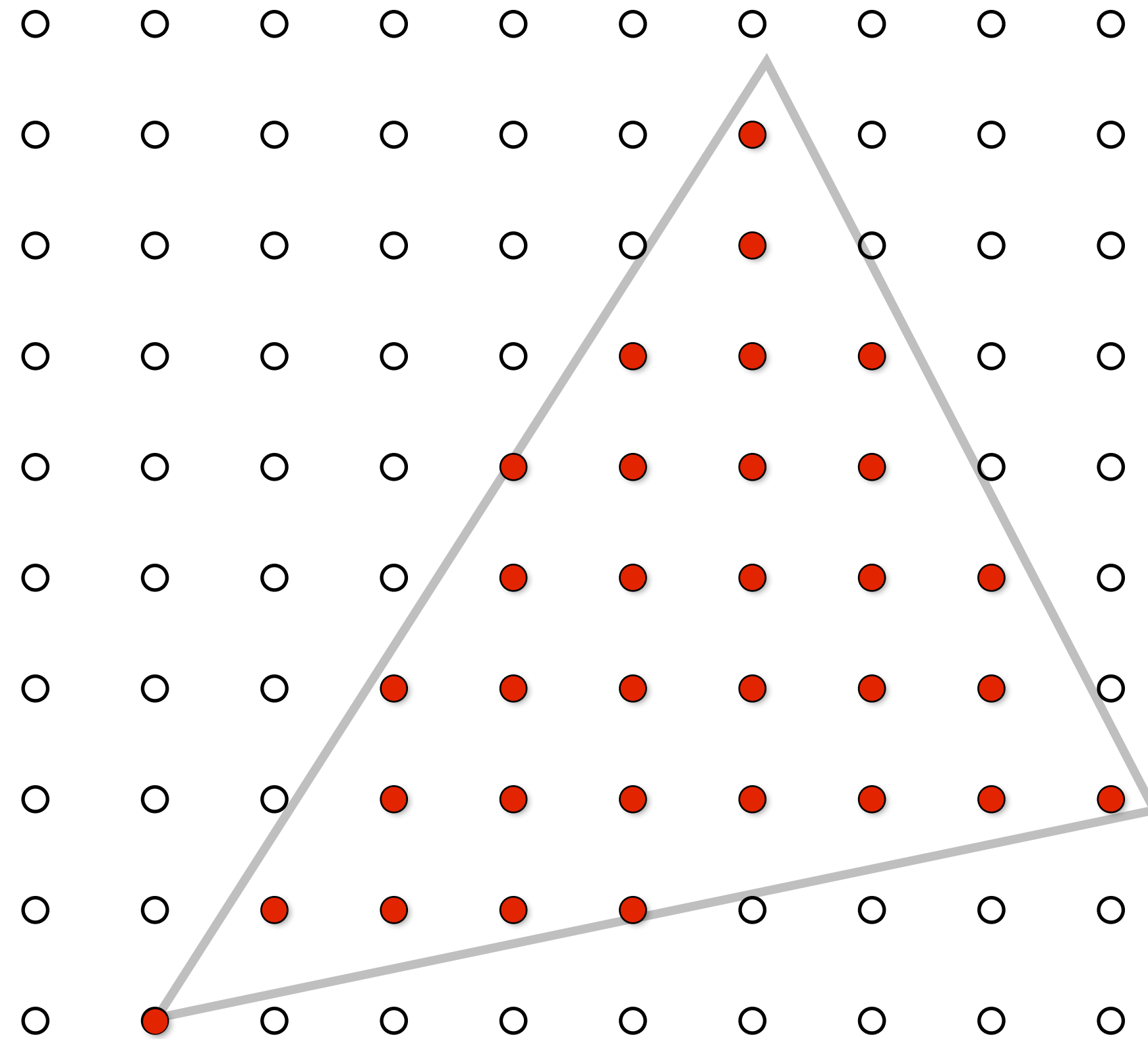
**Visual Computing Systems
Stanford CS348K, Spring 2022**

Simple OpenGL/Direct3D graphics pipeline

* Several stages of the modern OpenGL pipeline are omitted



Sample coverage at pixel centers



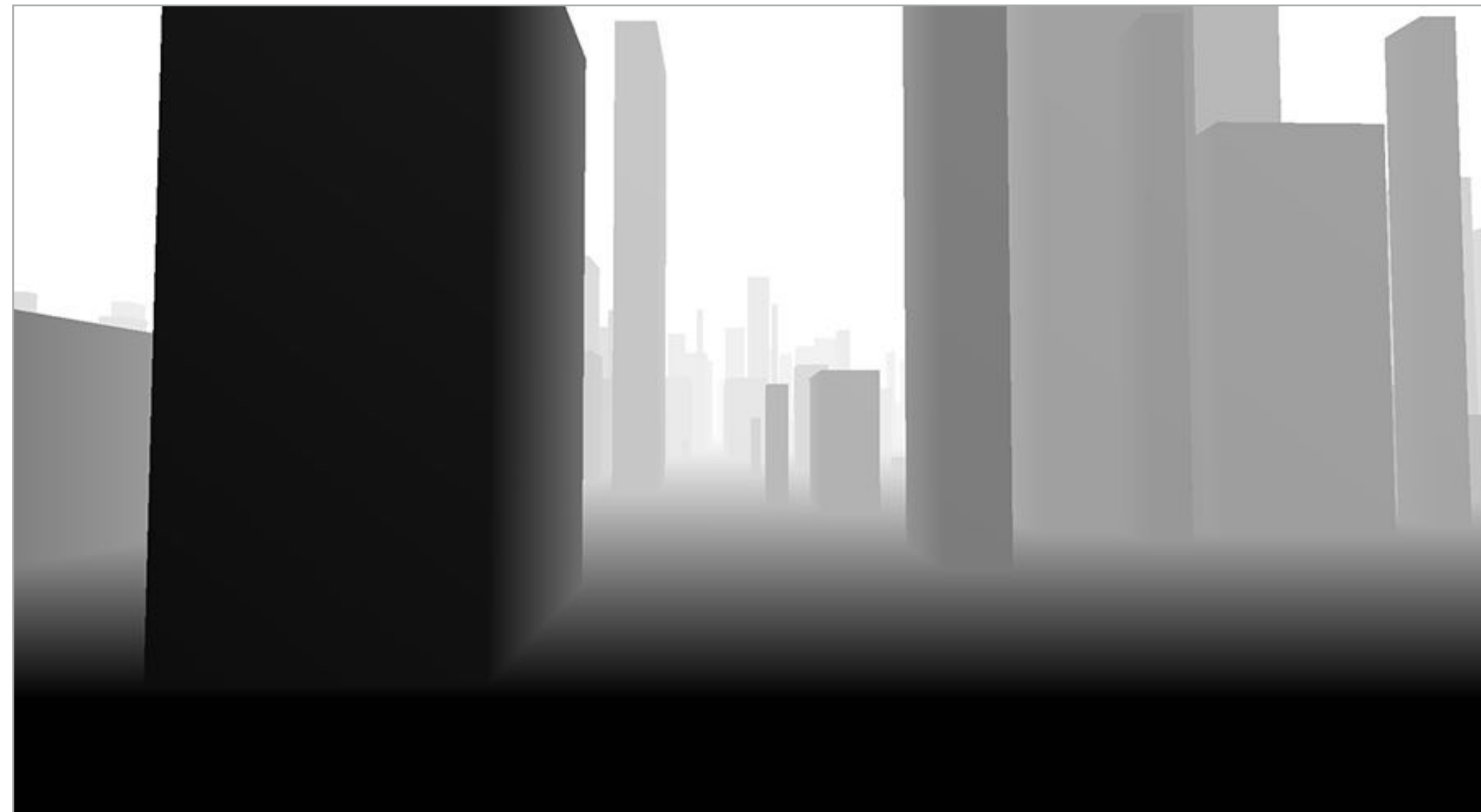
Depth buffer (aka “Z buffer”)

Color buffer:
(stores color per sample... e.g., RGB)

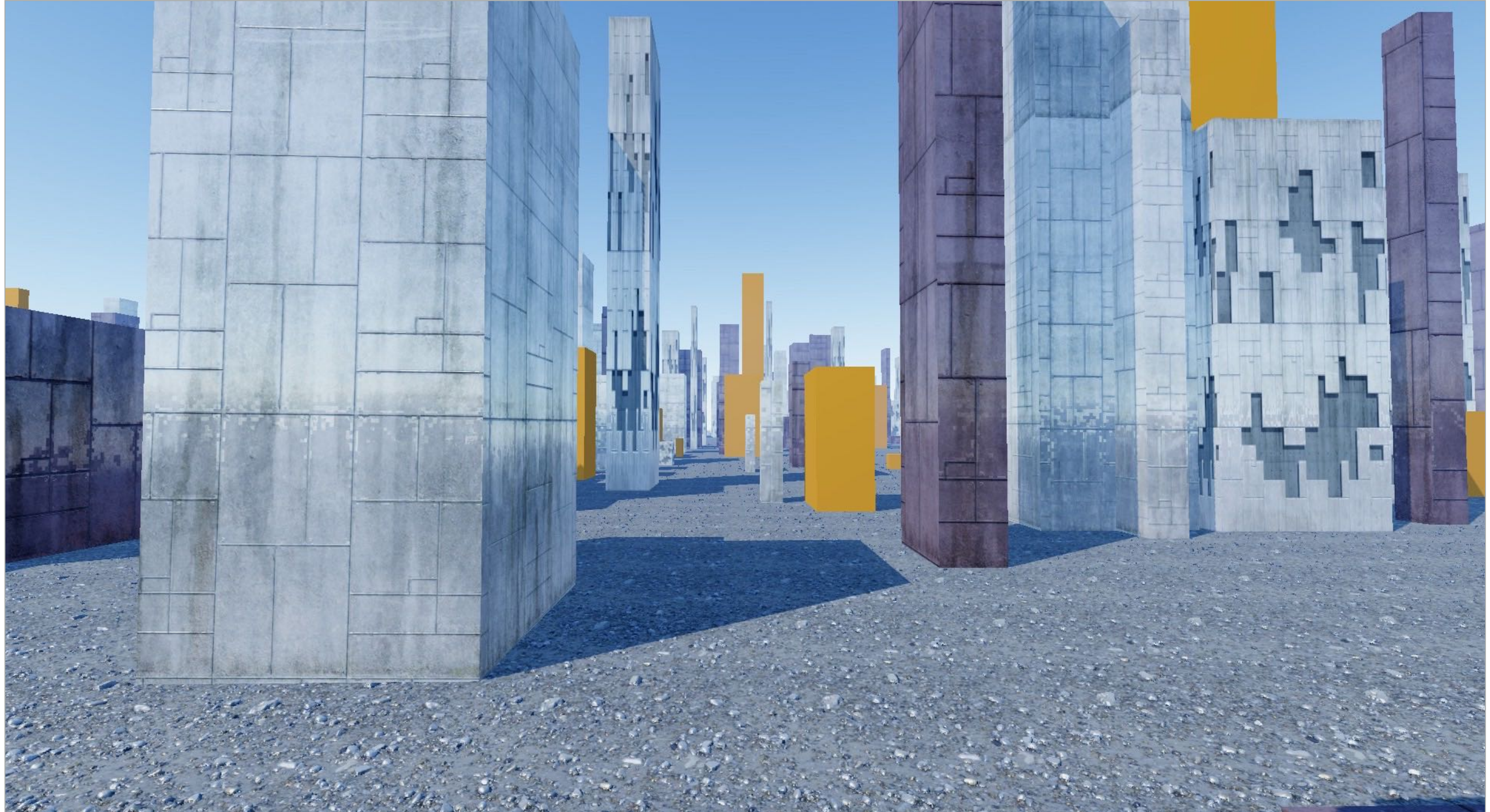


Depth buffer:
(stores depth per sample)

Stores depth of closest surface drawn so far
black = close depth
white = far depth



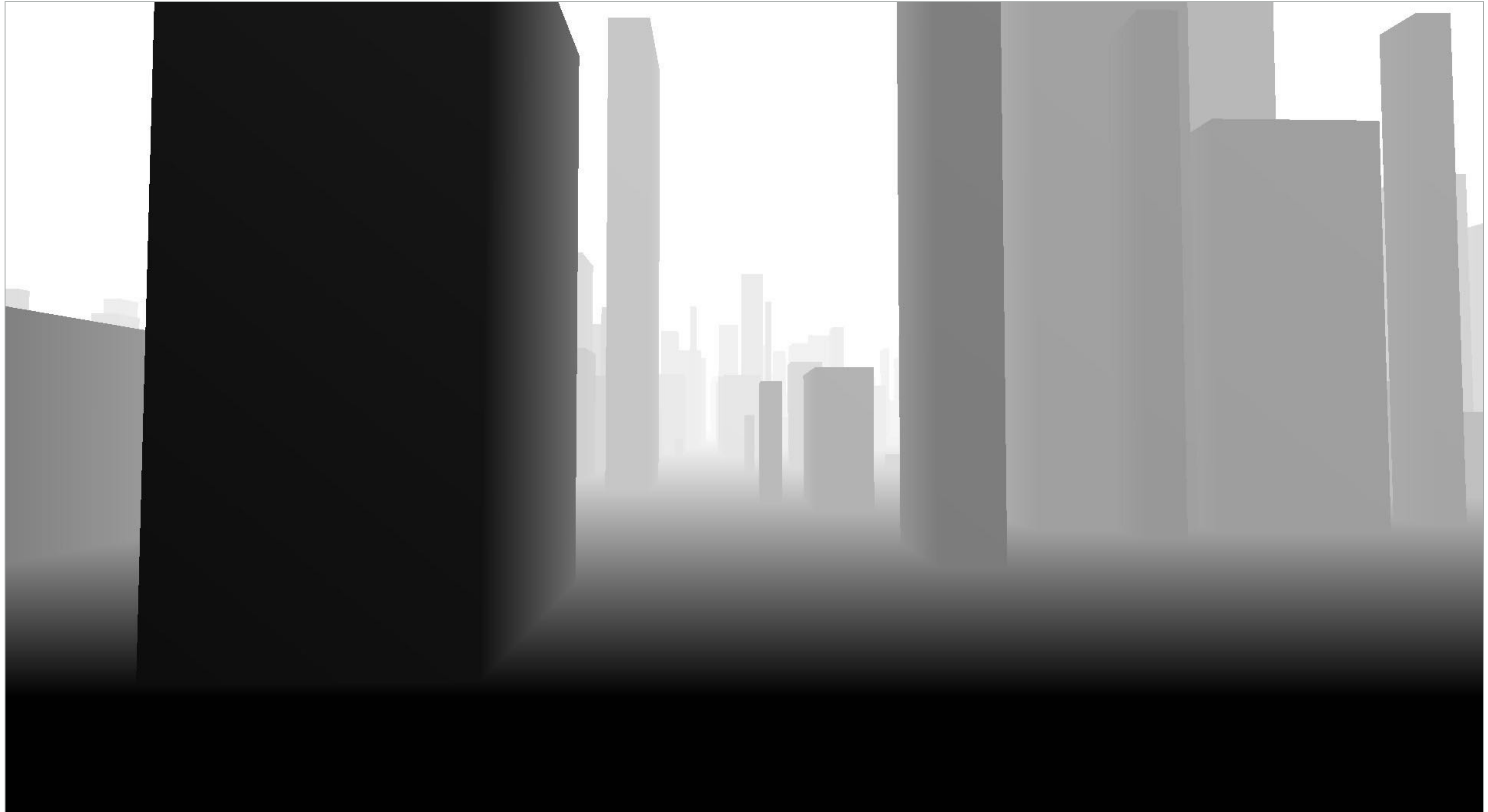
Depth buffer (a better look)



Color buffer (stores color measurement per sample, eg., RGB value per sample)

Depth buffer (a better look)

Visualization: the darker the pixel, the shorter the distance to the closest object



Corresponding depth buffer after rendering all triangles (stores closest scene depth per sample)

Occlusion using the depth buffer (opaque surfaces)

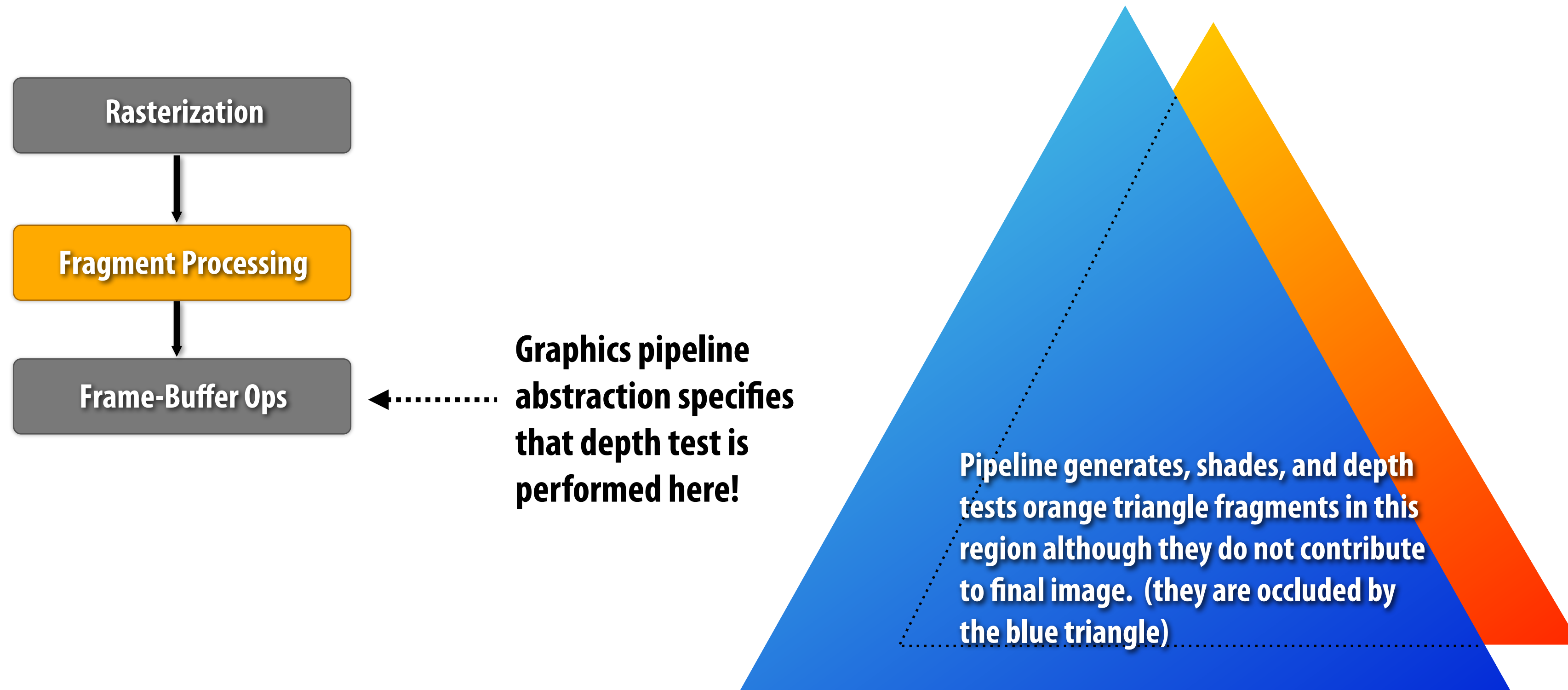
```
bool pass_depth_test(d1, d2) {  
    return d1 < d2;  
}
```

```
depth_test(tri_d, tri_color, x, y) {  
  
    if (pass_depth_test(tri_d, depth_buffer[x][y]) {  
  
        // if triangle is closest object seen so far at this  
        // sample point. Update depth and color buffers.  
  
        depth_buffer[x][y] = tri_d;    // update depth_buffer  
        color[x][y] = tri_color;      // update color buffer  
    }  
}
```

Early Z: one optimization you should know about

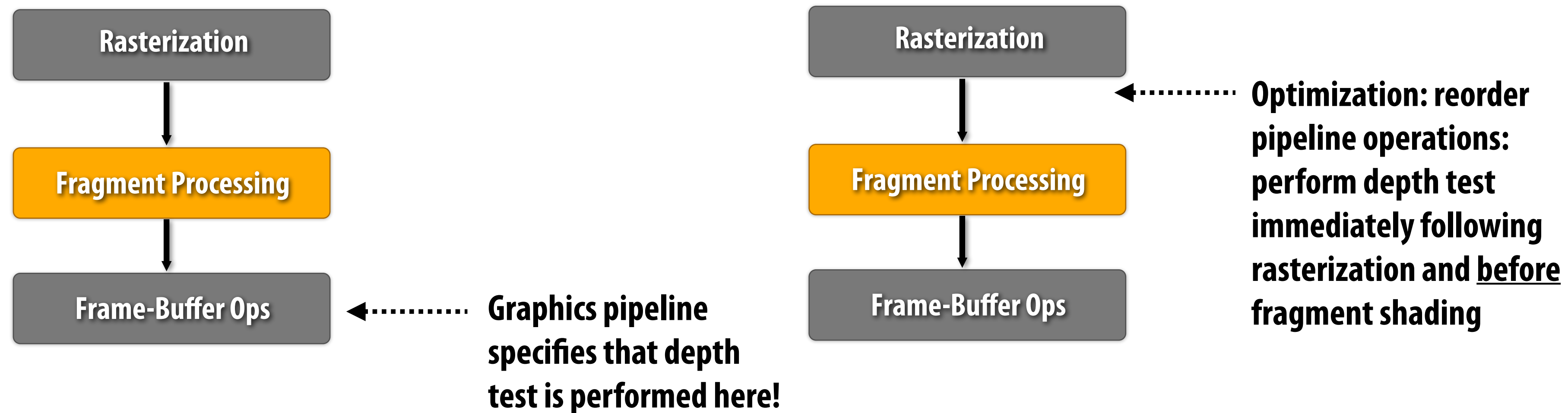
Depth testing as we've described it

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



Early Z culling

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



Key assumption: occlusion results do not depend on fragment shading

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

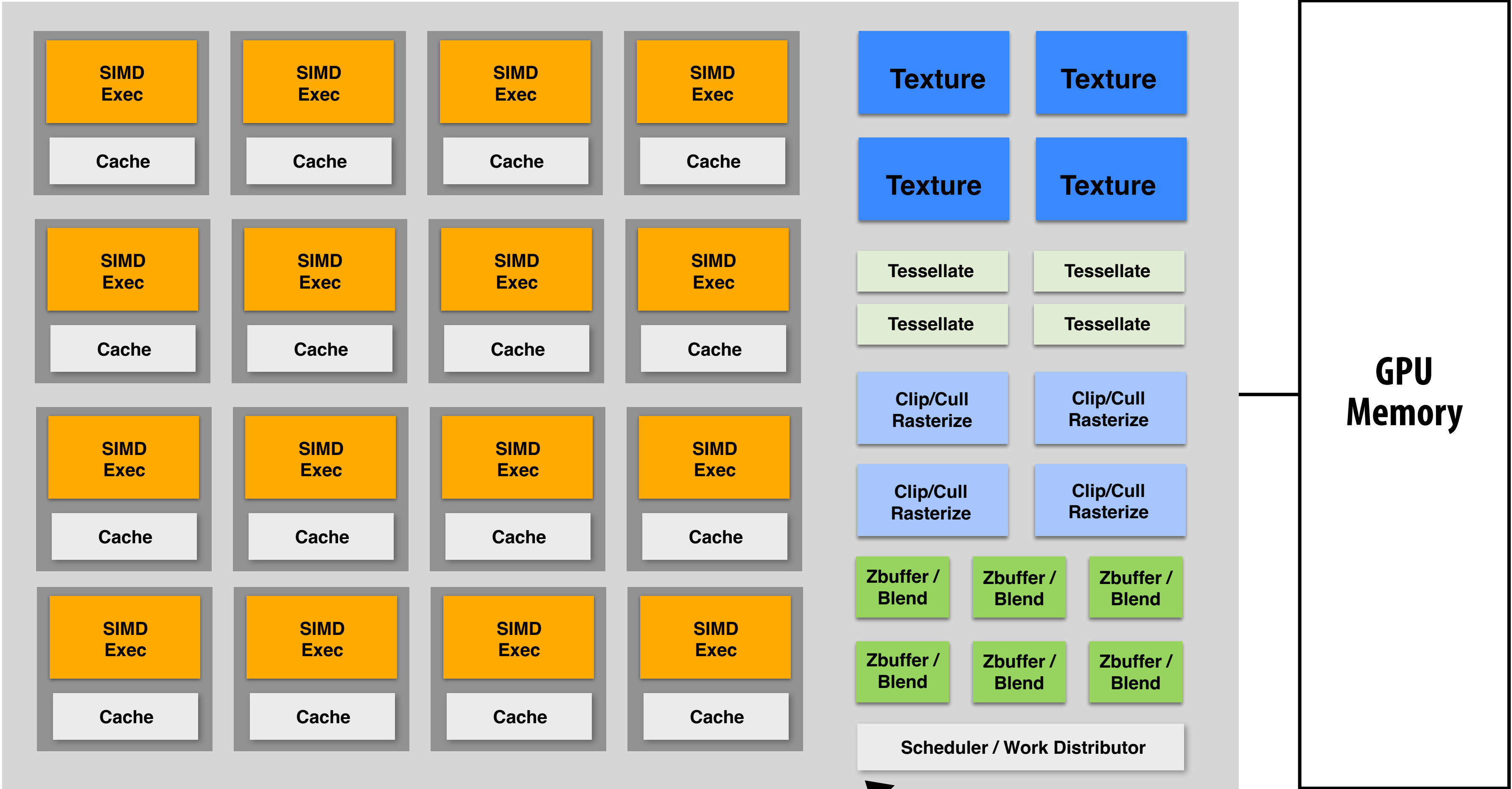
Programming the graphics pipeline

■ Issue draw commands → output image contents change

Command Type	Command
State change	Bind shaders, textures, uniforms
Draw	Draw using vertex buffer for object 1
State change	Bind new uniforms
Draw	Draw using vertex buffer for object 2
State change	Bind new shaders
Draw	Draw using vertex buffer for object 3
State change	Change depth test function
State change	Bind new shader
Draw	Draw using vertex buffer for object 4

Final rendered output should be consistent with the results of executing these commands in the order they are issued to the graphics pipeline.

GPU: heterogeneous parallel processor



We're now going to talk
about this scheduler

Graphics workload metrics

Let's consider different workloads

Average triangle size



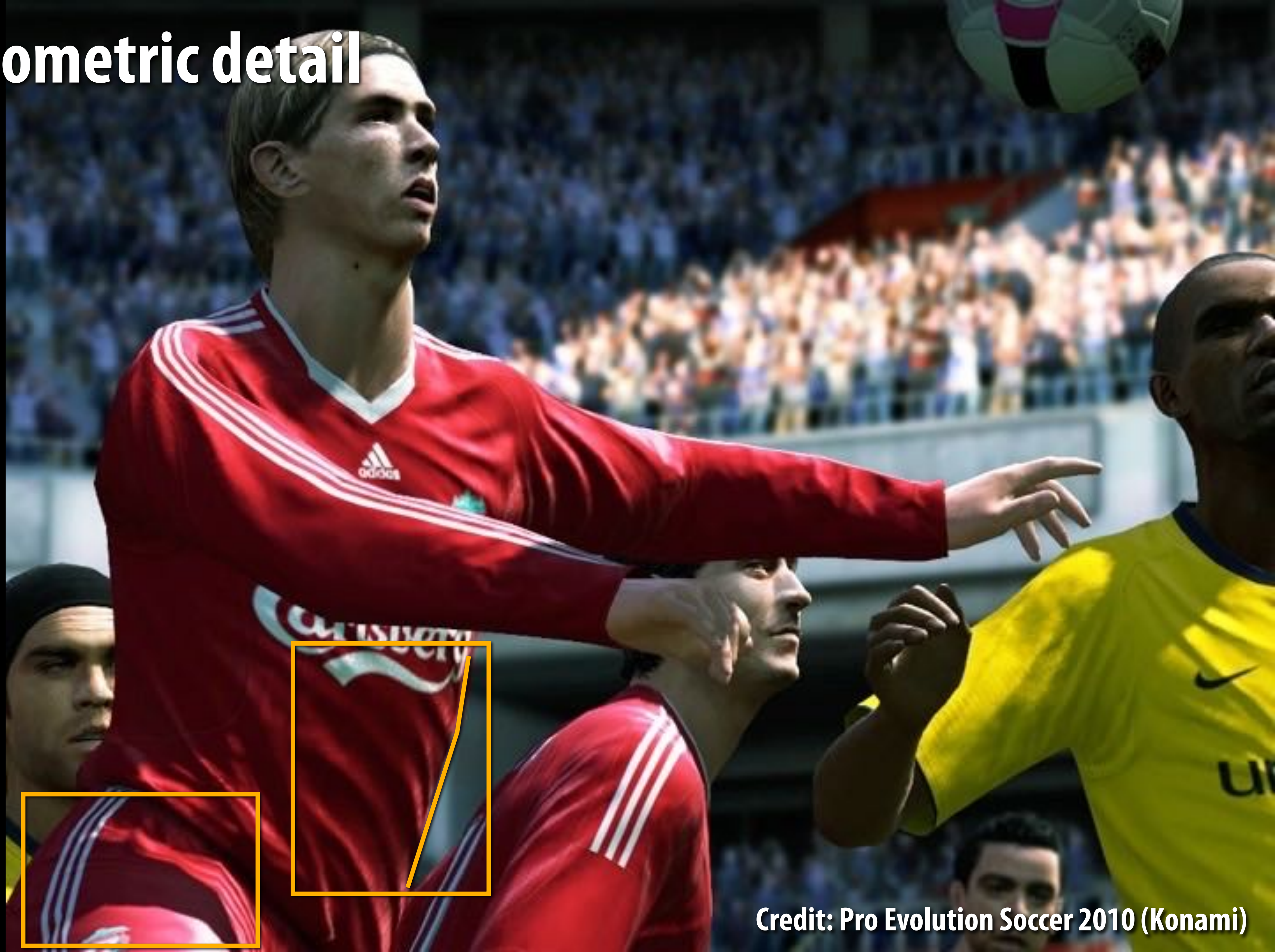
Image credit:

<https://www.theverge.com/2013/11/29/5155726/next-gen-supplementary-piece>

<http://www.mobygames.com/game/android/ghostbusters-slime-city/screenshots/gameShotId,852293/>



Low geometric detail



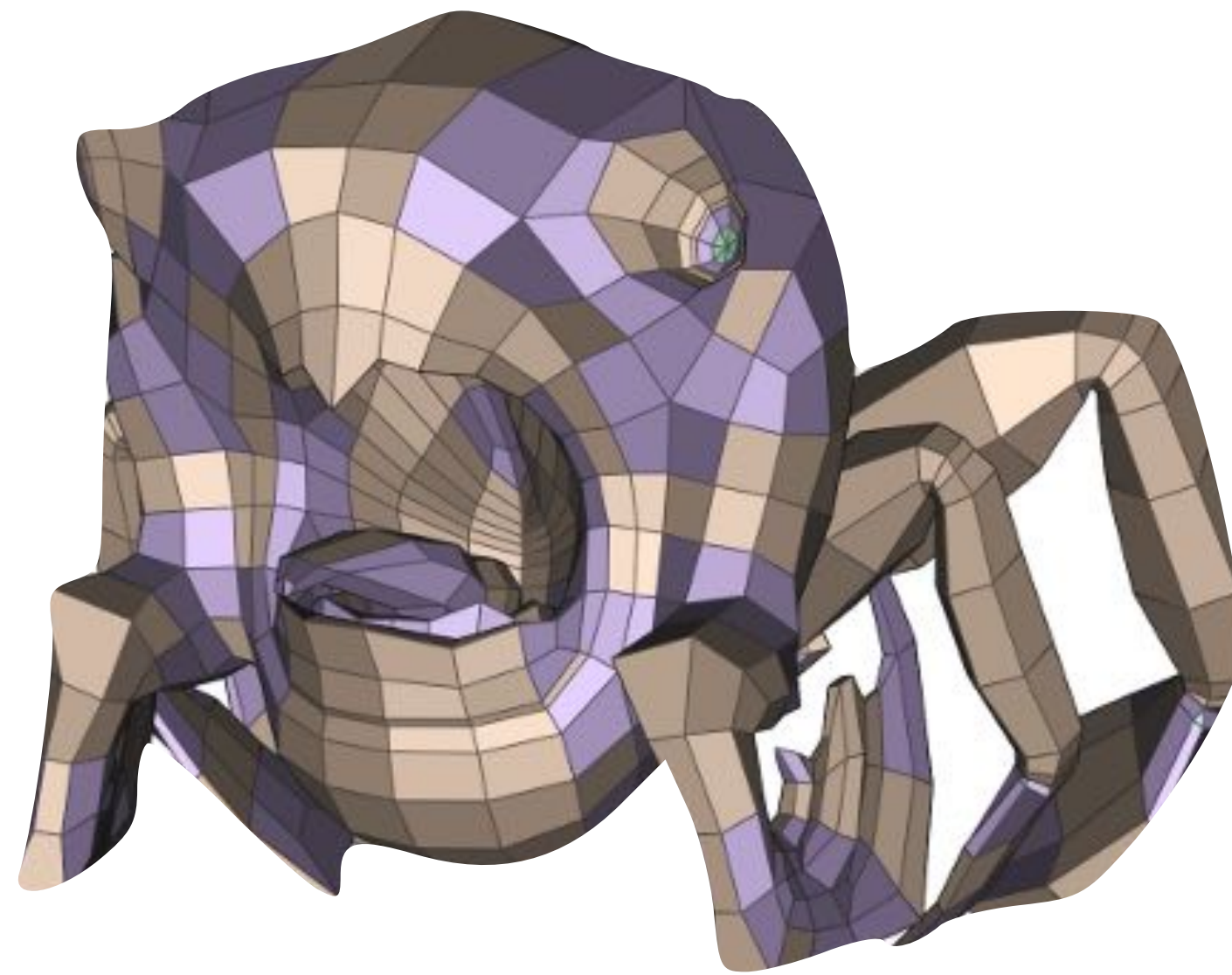
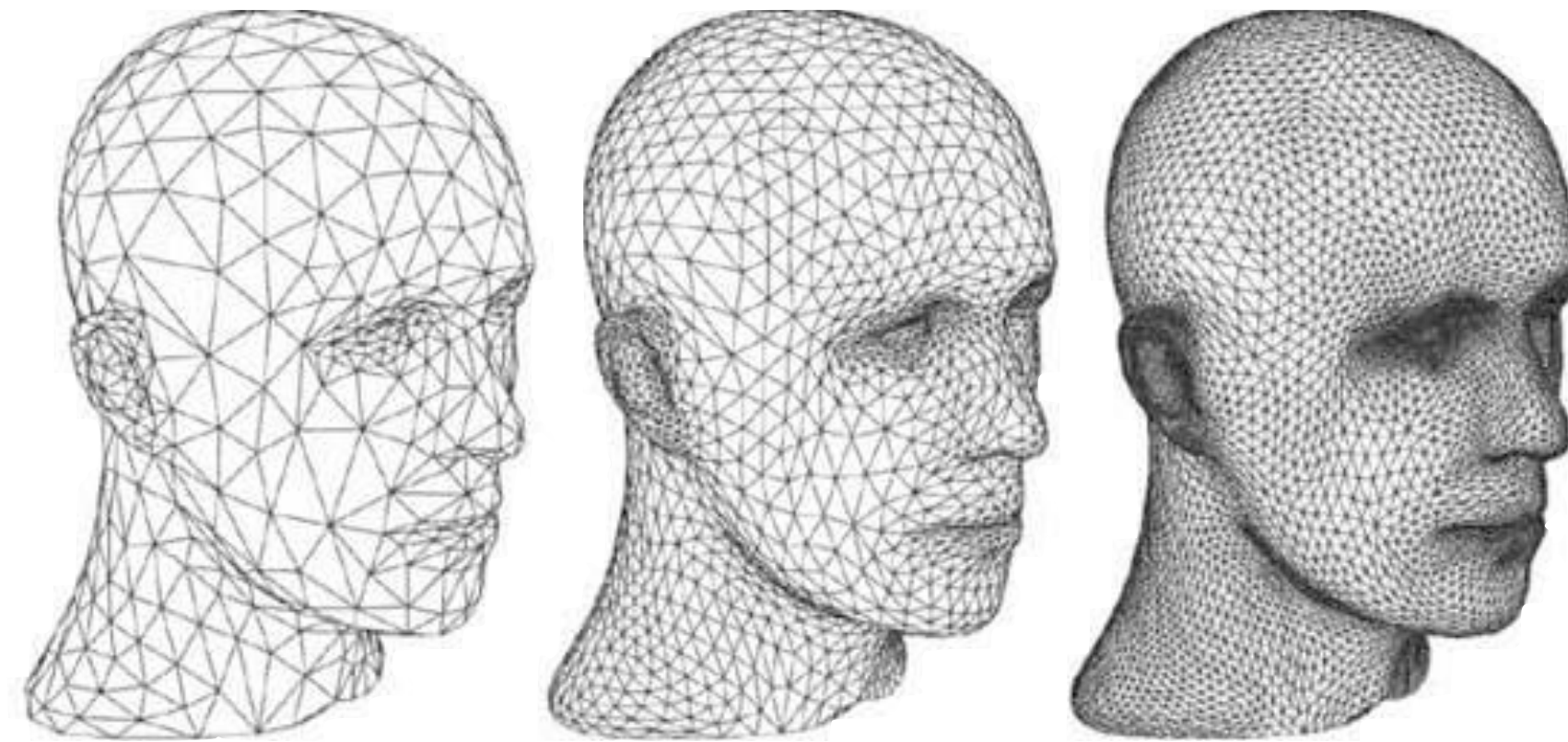
Credit: Pro Evolution Soccer 2010 (Konami)

High geometric detail

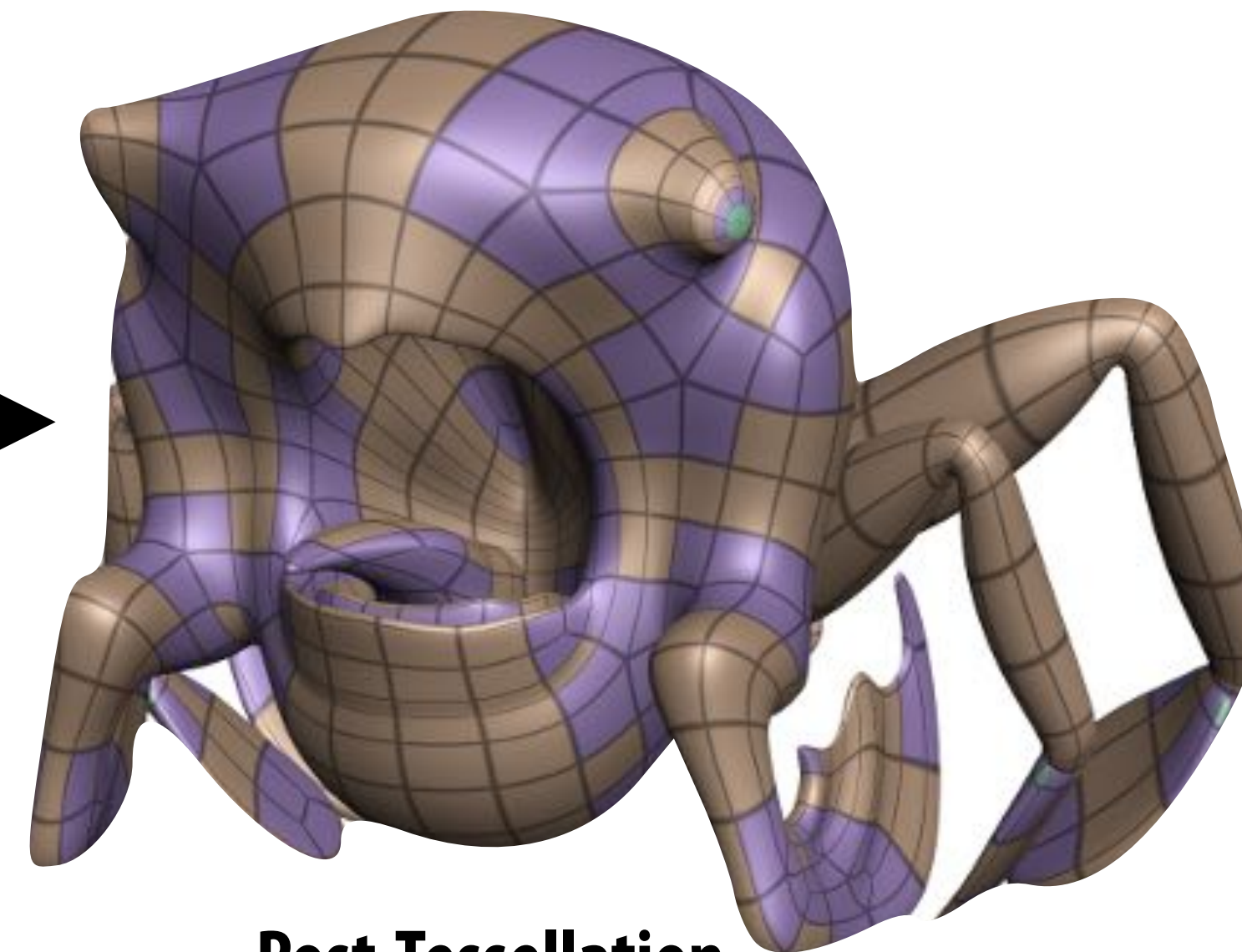


Surface tessellation

Procedurally generate fine triangle mesh from coarse mesh representation



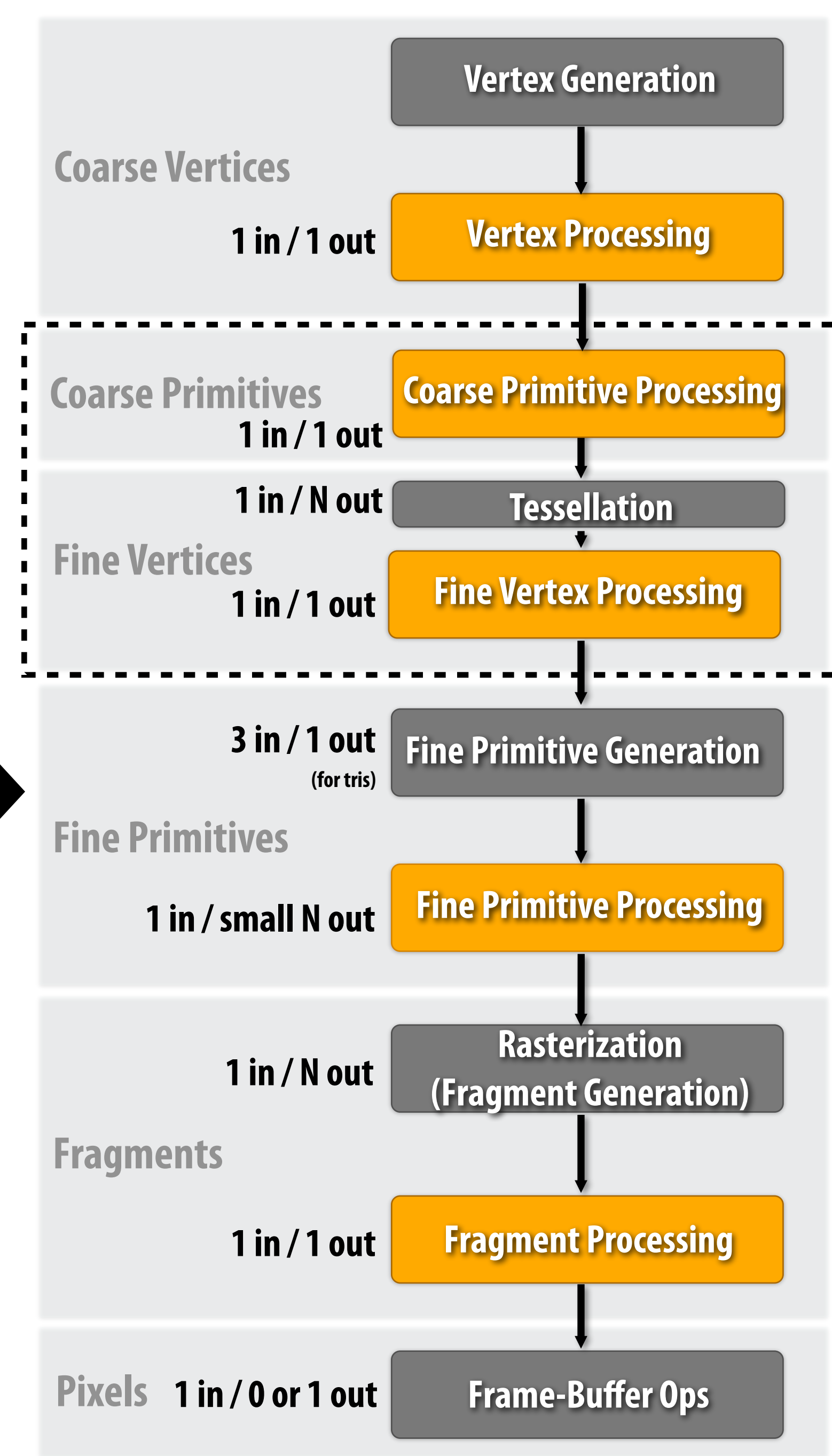
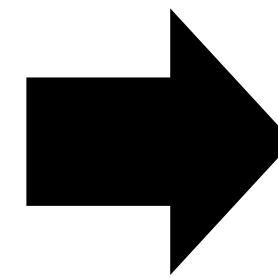
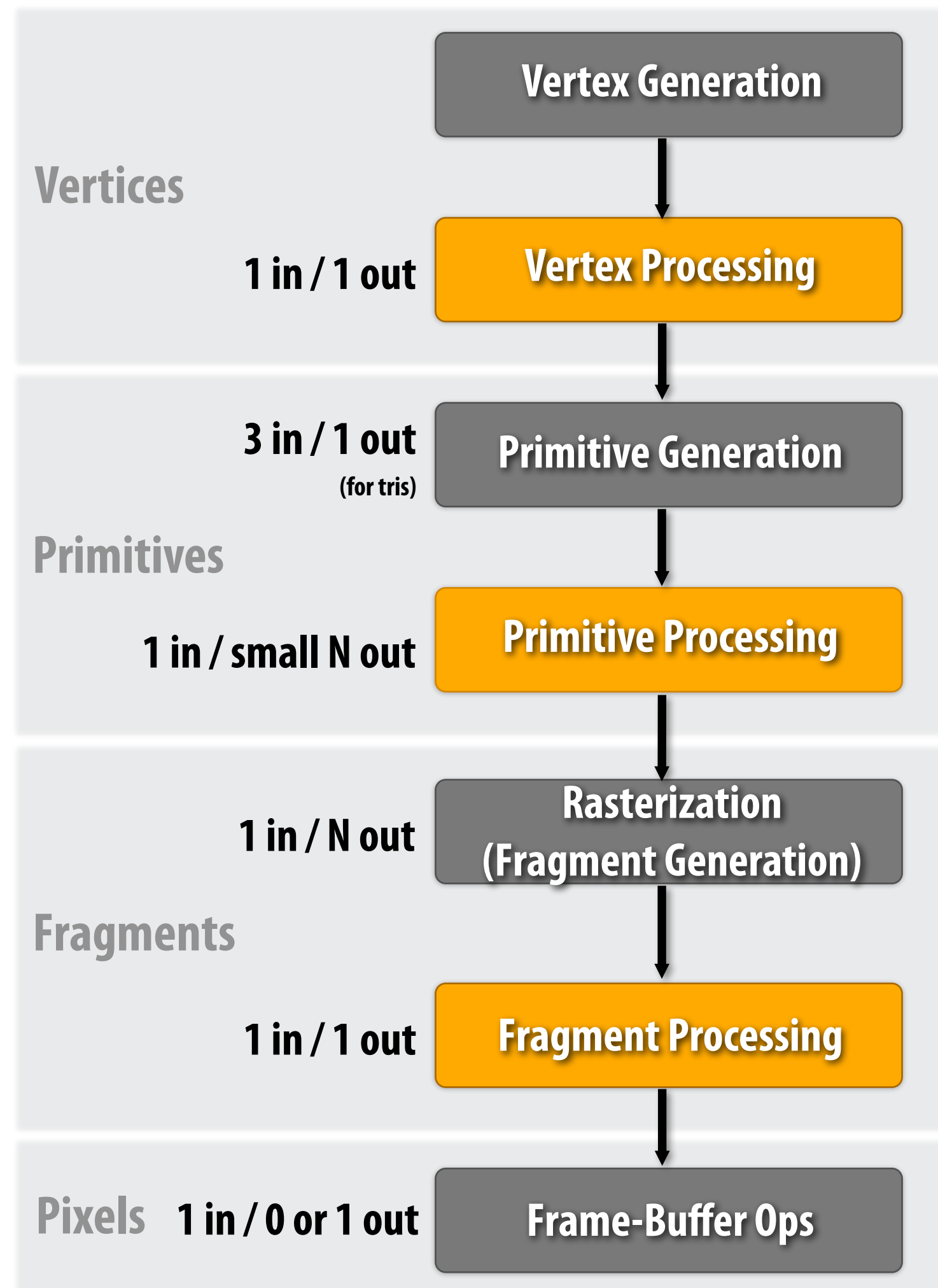
Coarse geometry



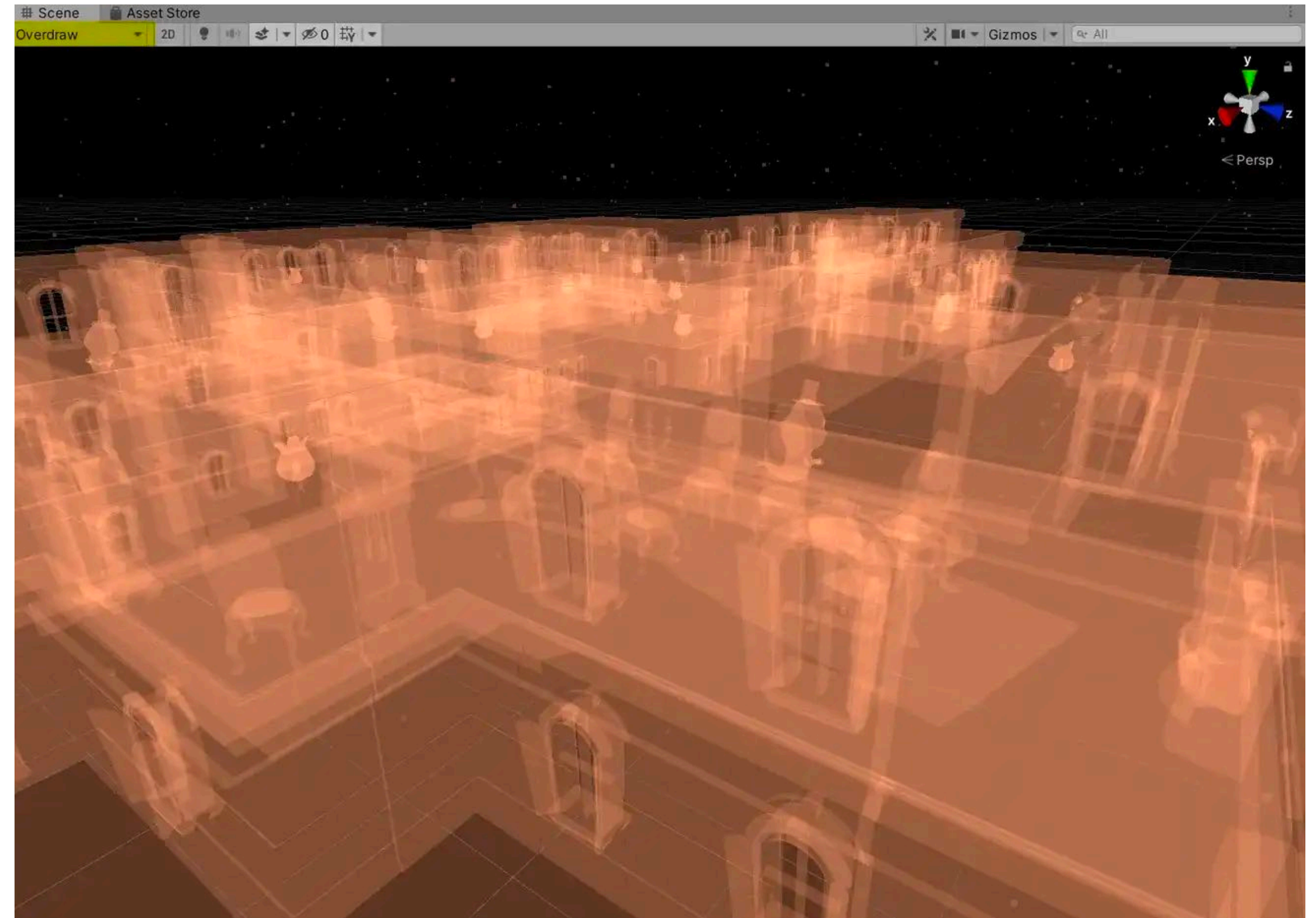
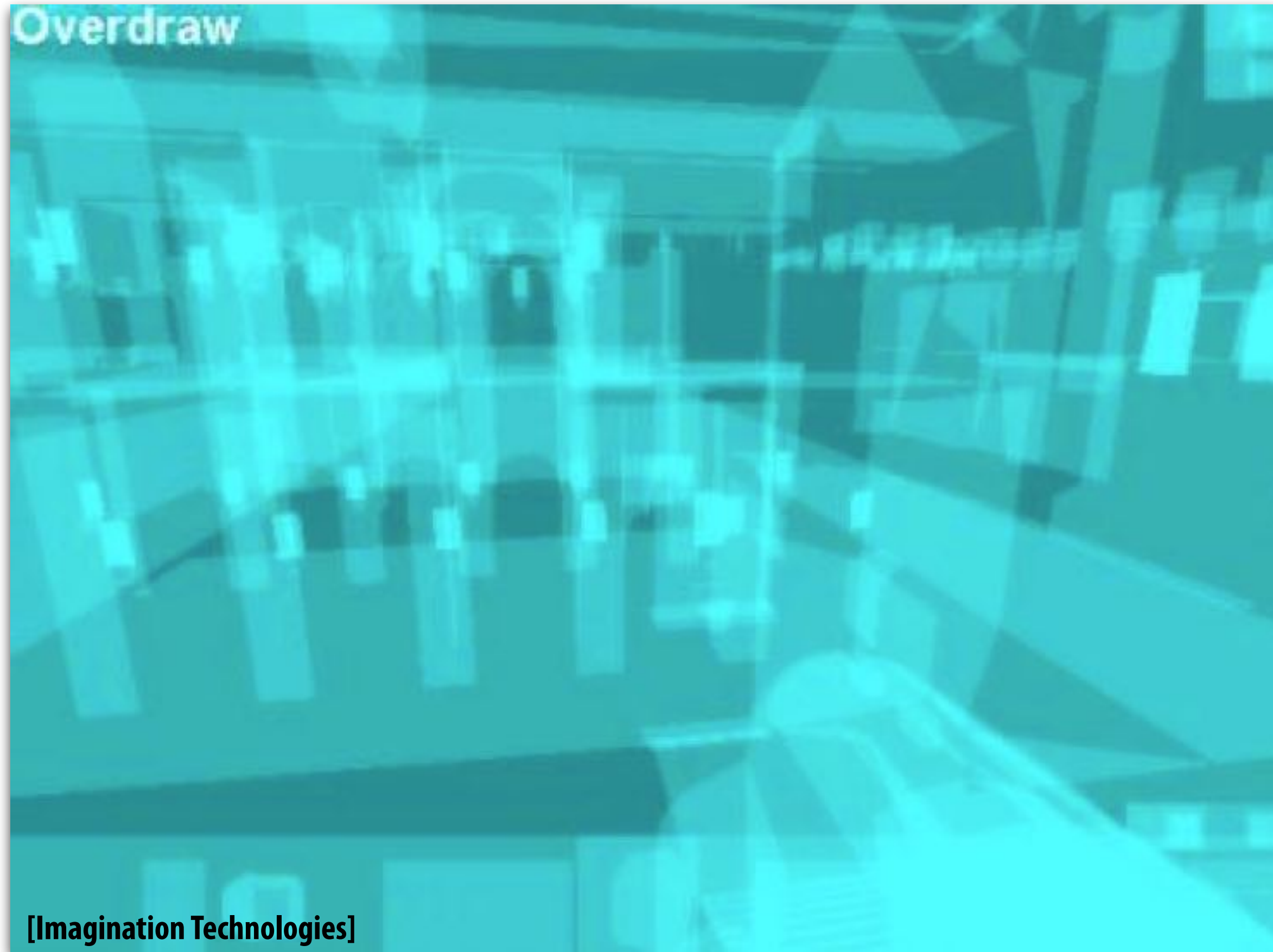
**Post-Tessellation
(fine) geometry**

Graphics pipeline with tessellation

Five programmable stages
(OpenGL 4, Direct3D 11)



Scene depth complexity



<https://thegamedev.guru/unity-gpu-performance/overdraw-optimization/>

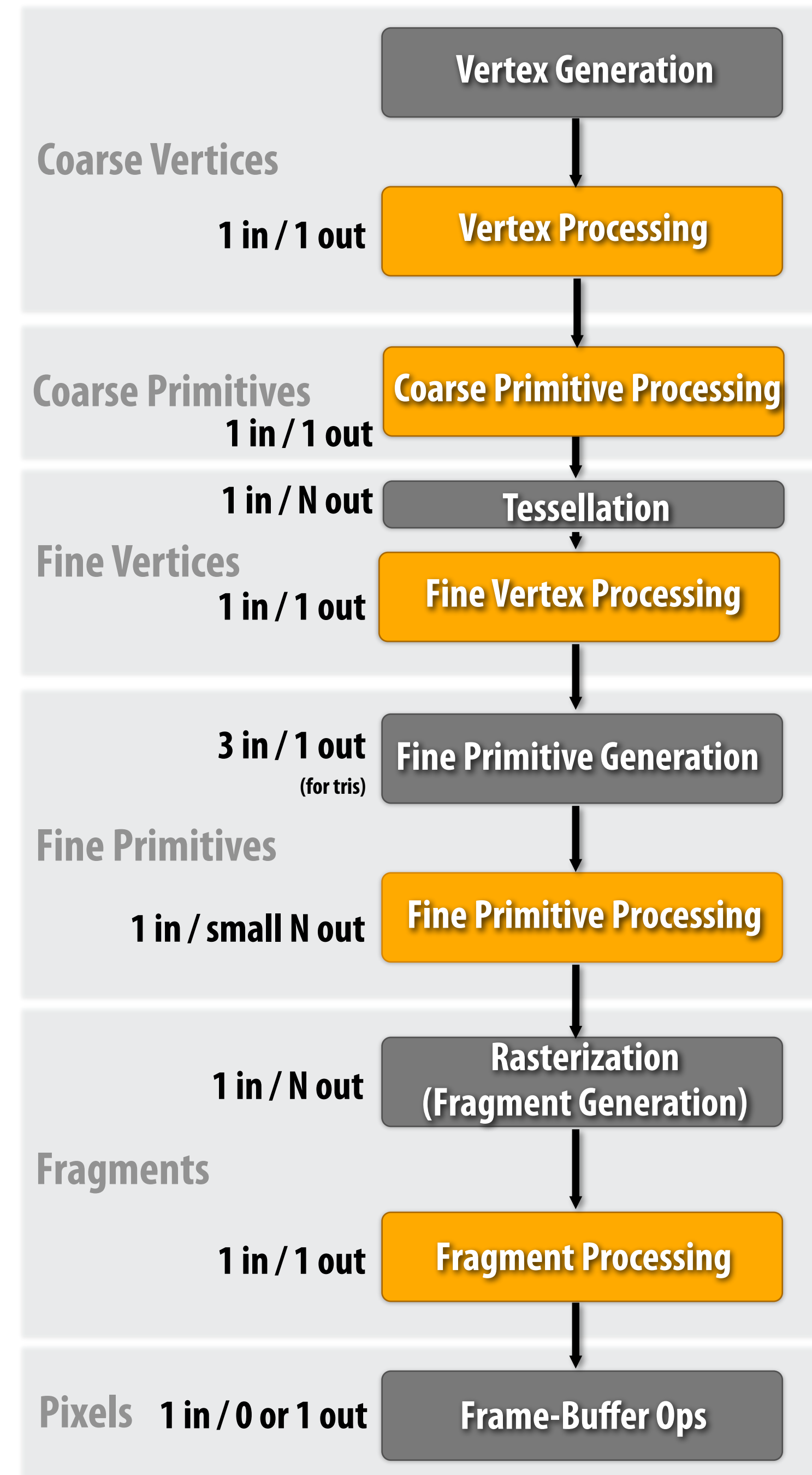
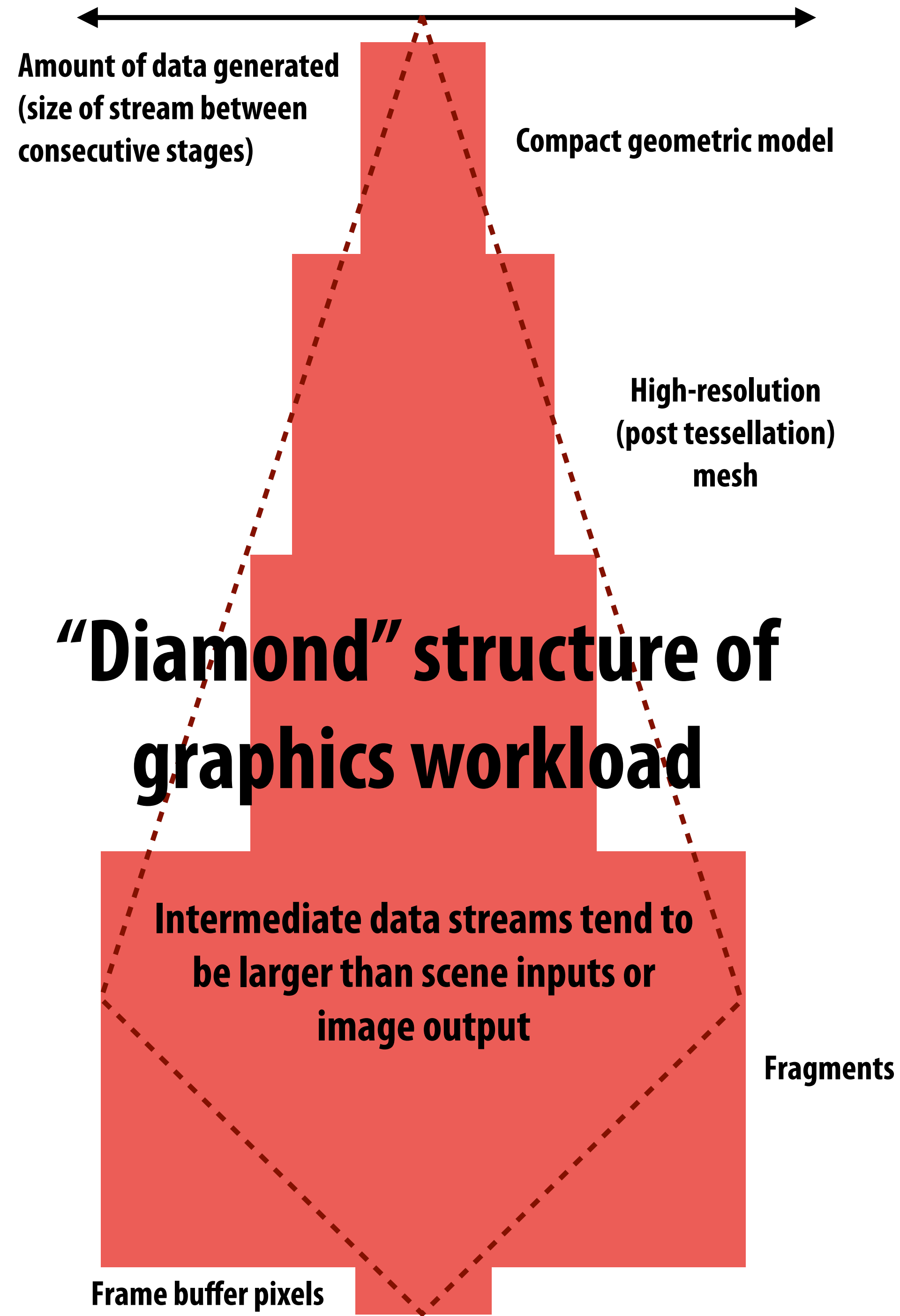
Rough approximation: $TA = SD$

T = # triangles

A = average triangle area

S = pixels on screen

D = average depth complexity



Key 3D graphics workload metrics

- **Data amplification from stage to stage**
 - Average triangle size (amplification in rasterizer: 1 triangle \rightarrow N pixels)
 - Expansion during primitive processing (if enabled)
 - Tessellation factor (if tessellation enabled)
- **[Vertex/fragment/geometry] shader cost**
 - How many instructions?
 - Ratio of math to data access instructions?
- **Scene depth complexity**
 - Determines number of depth and color buffer writes

Graphics pipeline workload changes dramatically across draw commands

- **Triangle size is scene and frame dependent**
 - Move far away from an object, triangles get smaller
 - Vary within a frame (characters are usually higher resolution meshes than buildings)
- **Varying complexity of materials, different number of lights illuminating surfaces**
 - Tens to a few hundreds of instructions per shader
- **Stages can be disabled**
 - Depth-only rendering = NULL fragment shader
 - Post-processing effects = no vertex work
- **Thousands of state changes and draw commands per frame**

Parallelizing the graphics pipeline

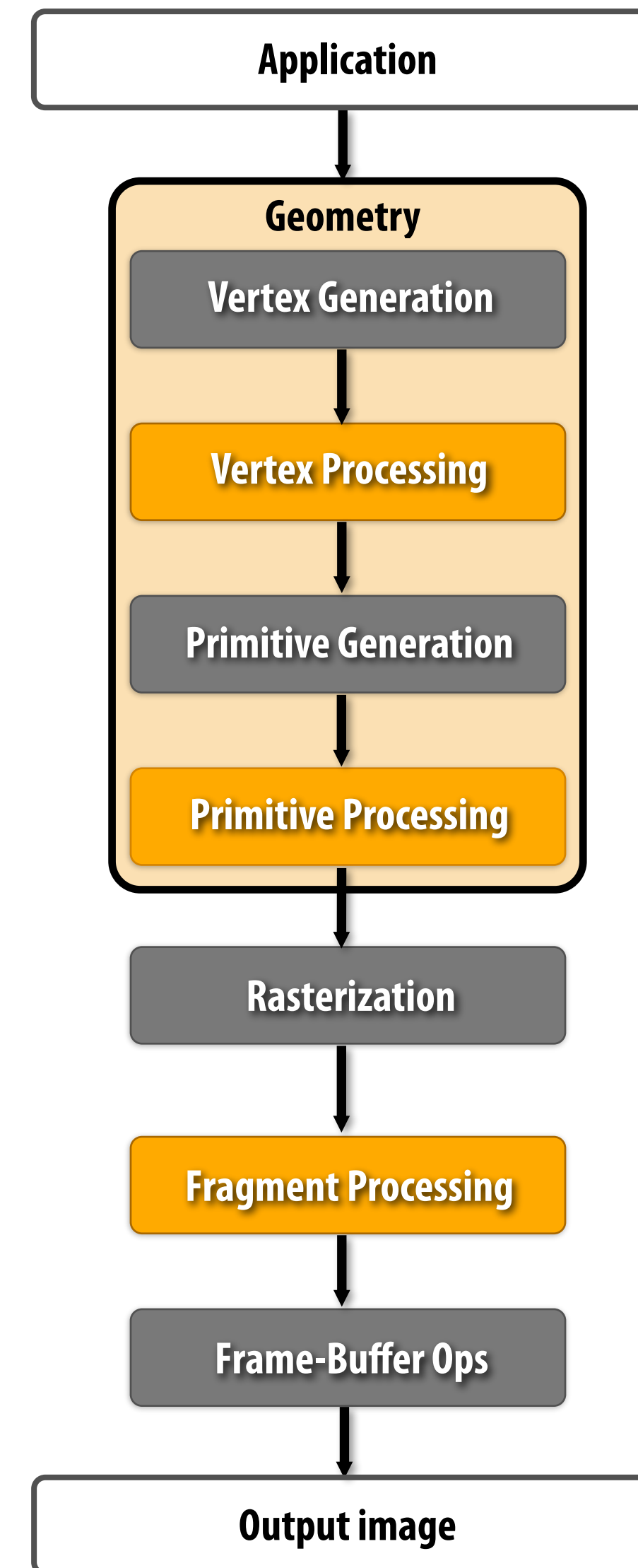
Requirements + workload challenges

- Pipeline accepts sequence of commands
 - Draw commands
 - State modification commands
- Processing commands has sequential semantics
 - Effects of command A must be visible before those of command B
- Relative cost of pipeline stages changes frequently and unpredictably (e.g., due to changing triangle size, rendering mode)
- Ample opportunities for parallelism
 - Many triangles, vertices, fragments, etc.

Simplified pipeline

For now: just consider all geometry processing work (vertex/primitive processing, tessellation, etc.) as “geometry” processing.

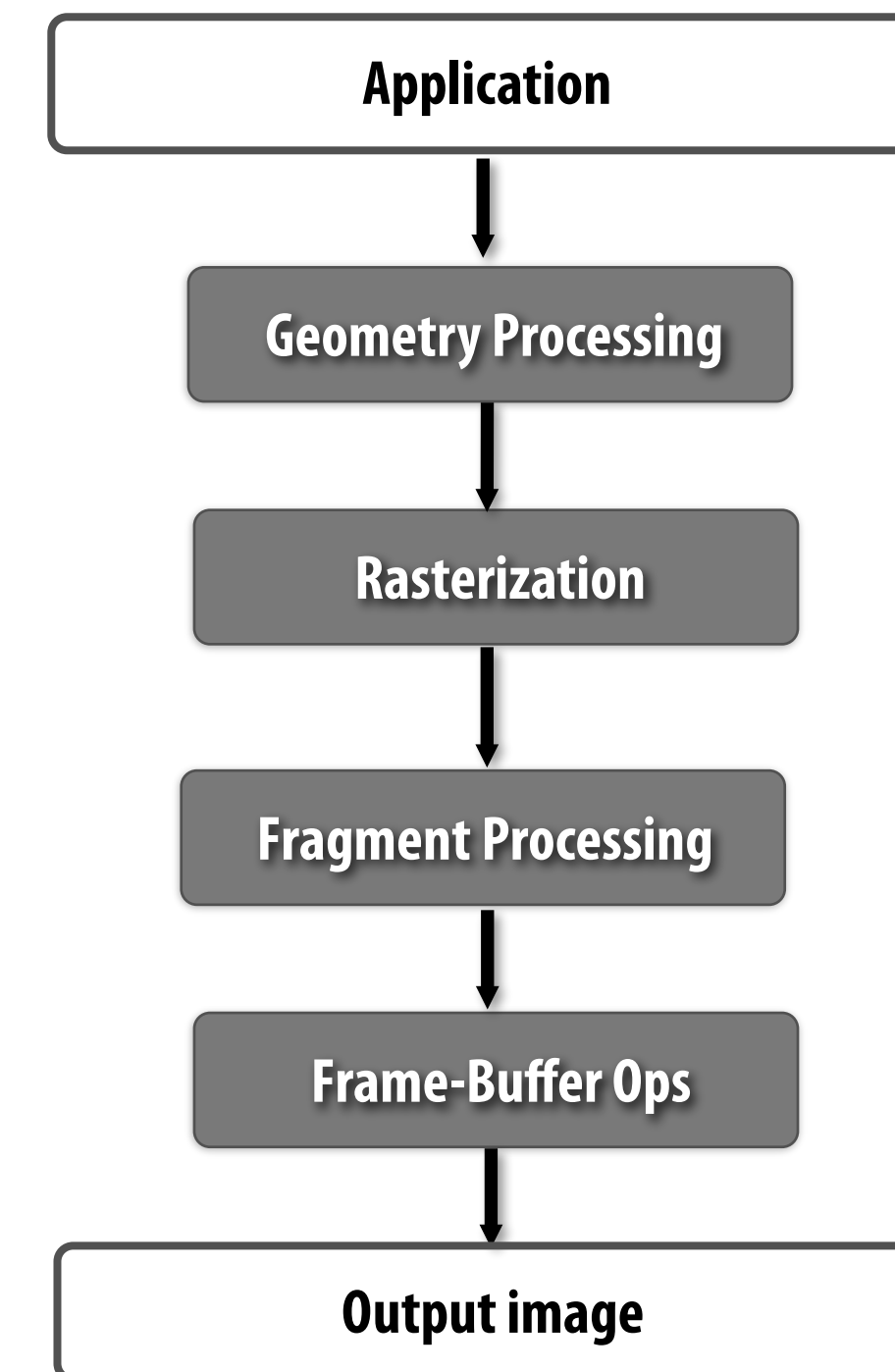
(I’m drawing the pipeline this way to match the suggested readings under this lecture)



Simple parallelization (pipeline parallelism)

Separate hardware unit is responsible for executing work in each stage

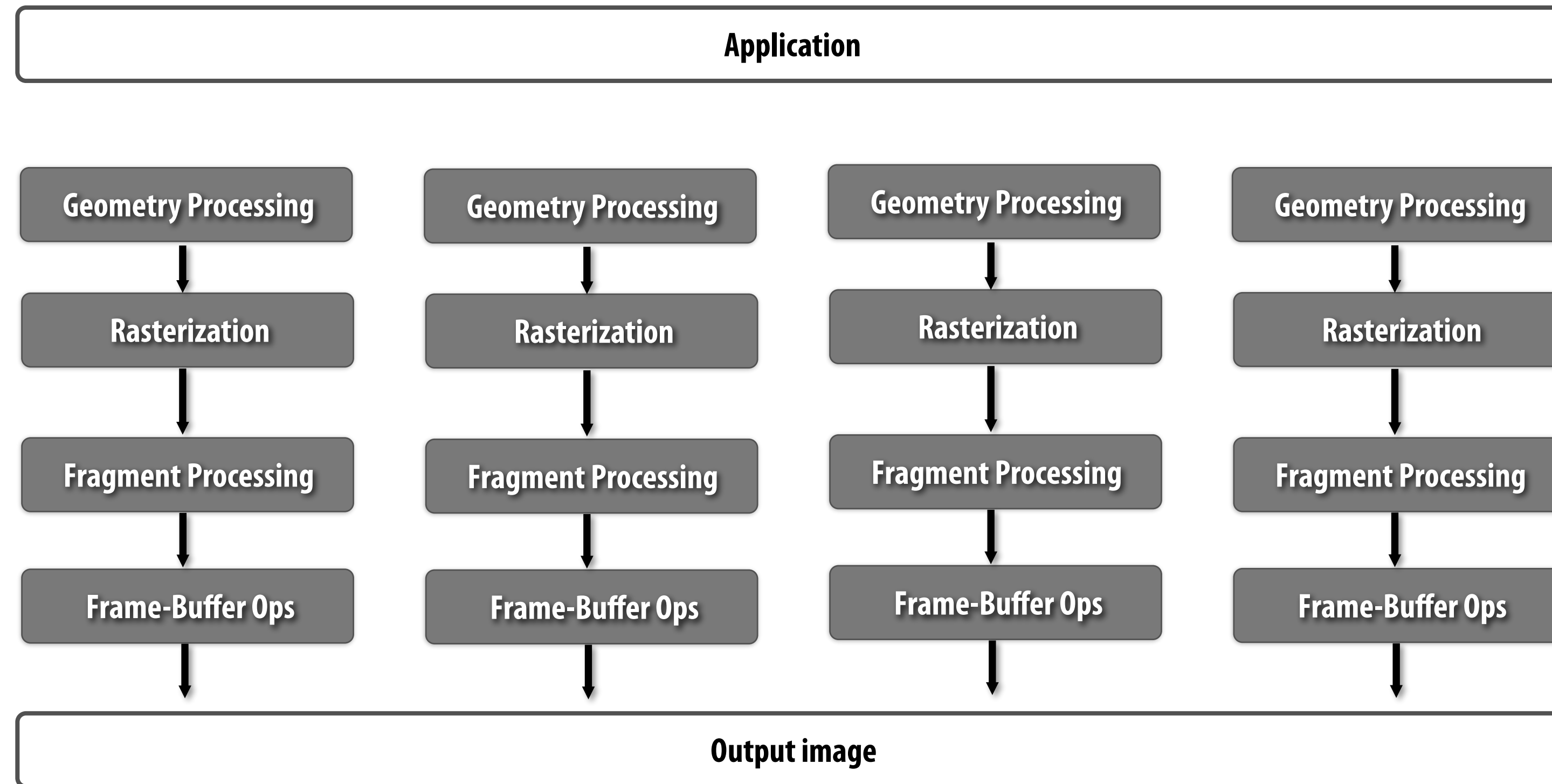
What is my maximum speedup?



A cartoon GPU:

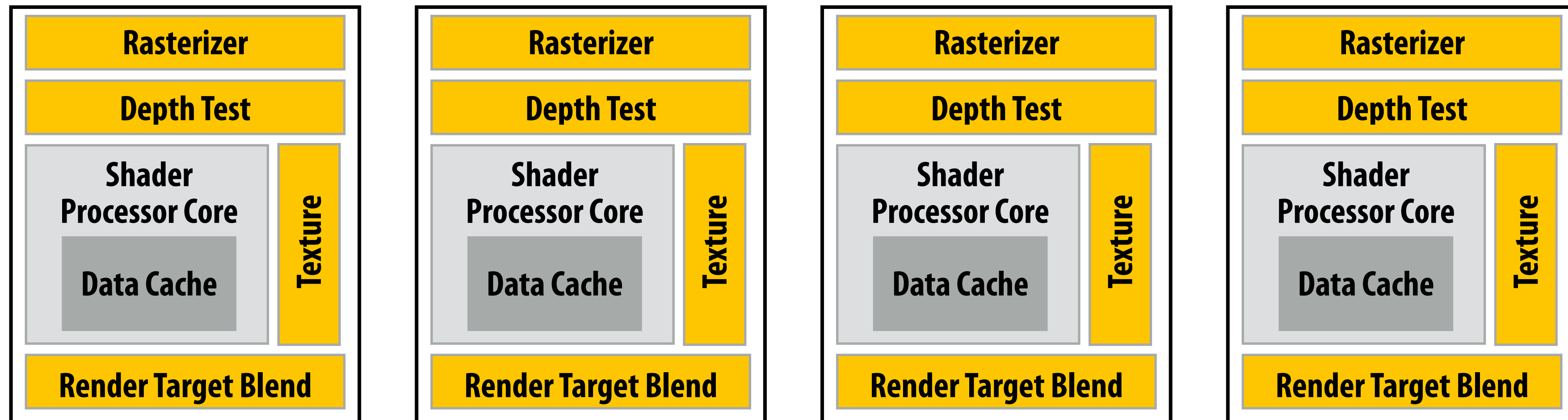
Assume we have four separate processing pipelines

Leverages data-parallelism present in rendering computation



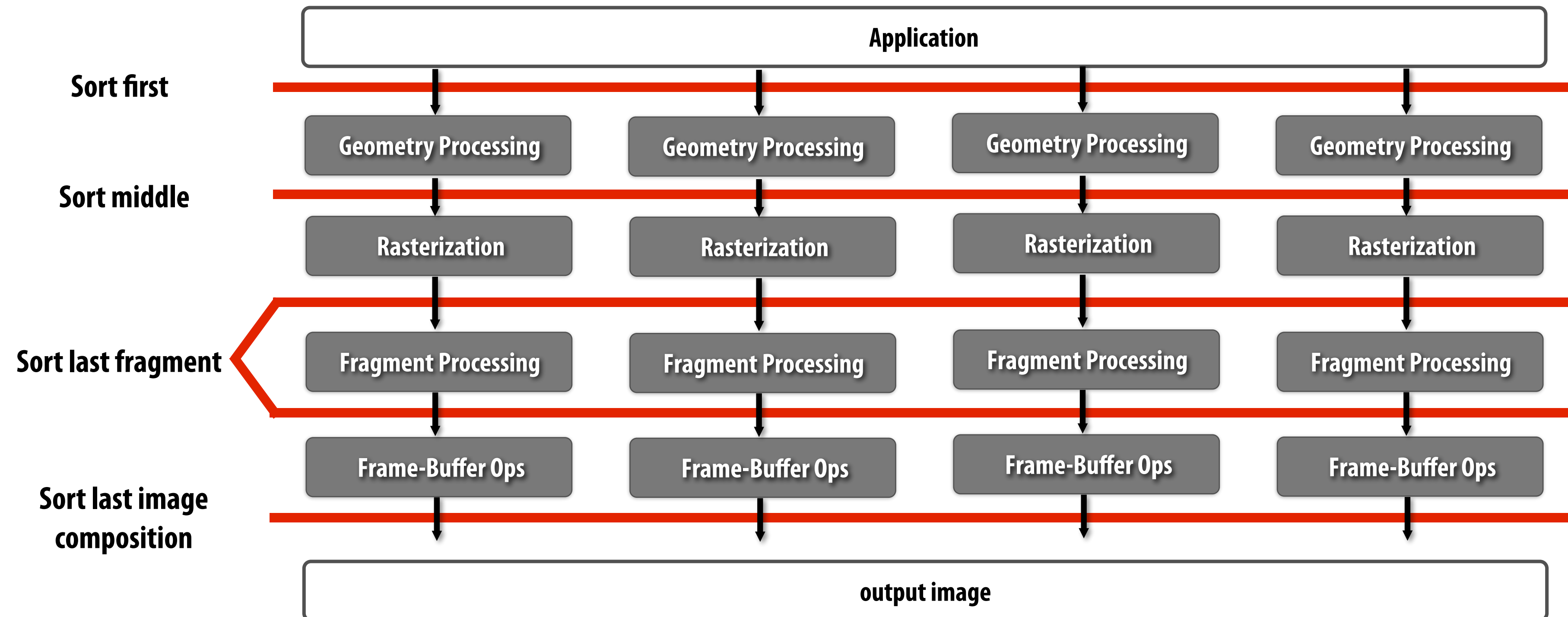
More realistic GPU

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



Molnar's “sorting” taxonomy

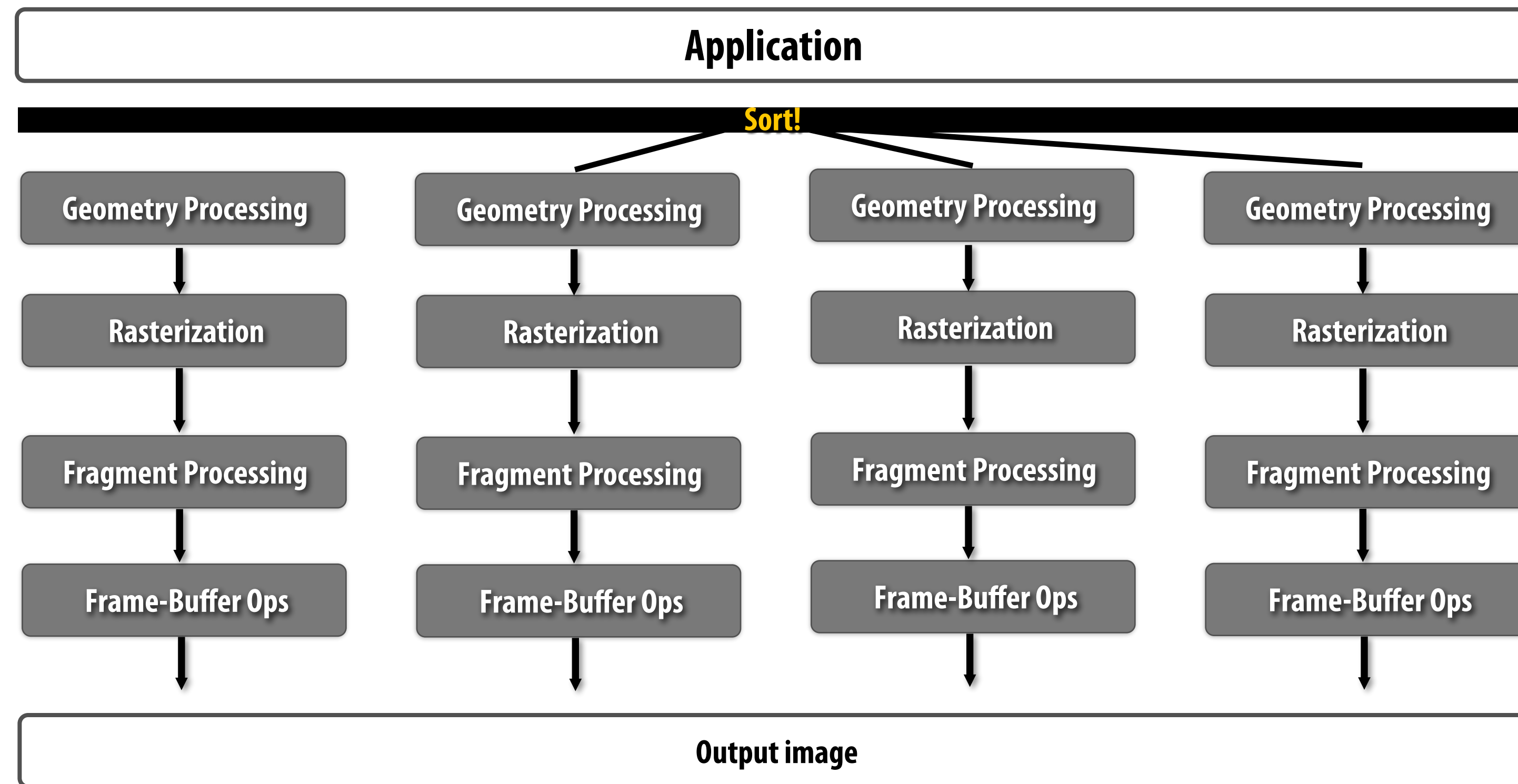
Implementations characterized by where inter-processor communication occurs in pipeline



Note: The term “sort” can be misleading for some. It may be helpful to instead consider the term “distribution” rather than sort. The implementations are characterized by how and when they redistribute work onto processors. *

Sort first

Sort first

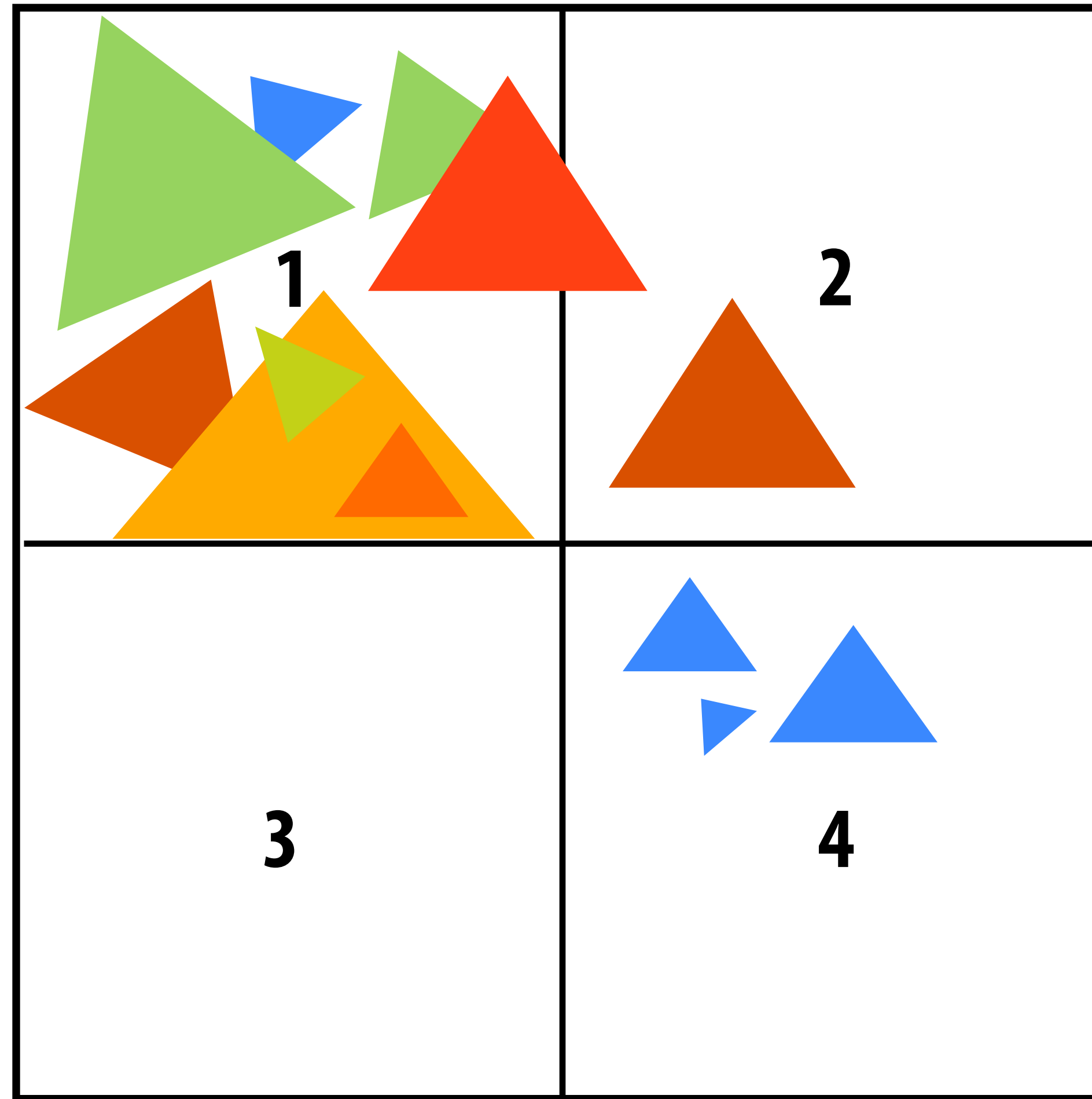


Assign each replicated pipeline responsibility for a region of the output image

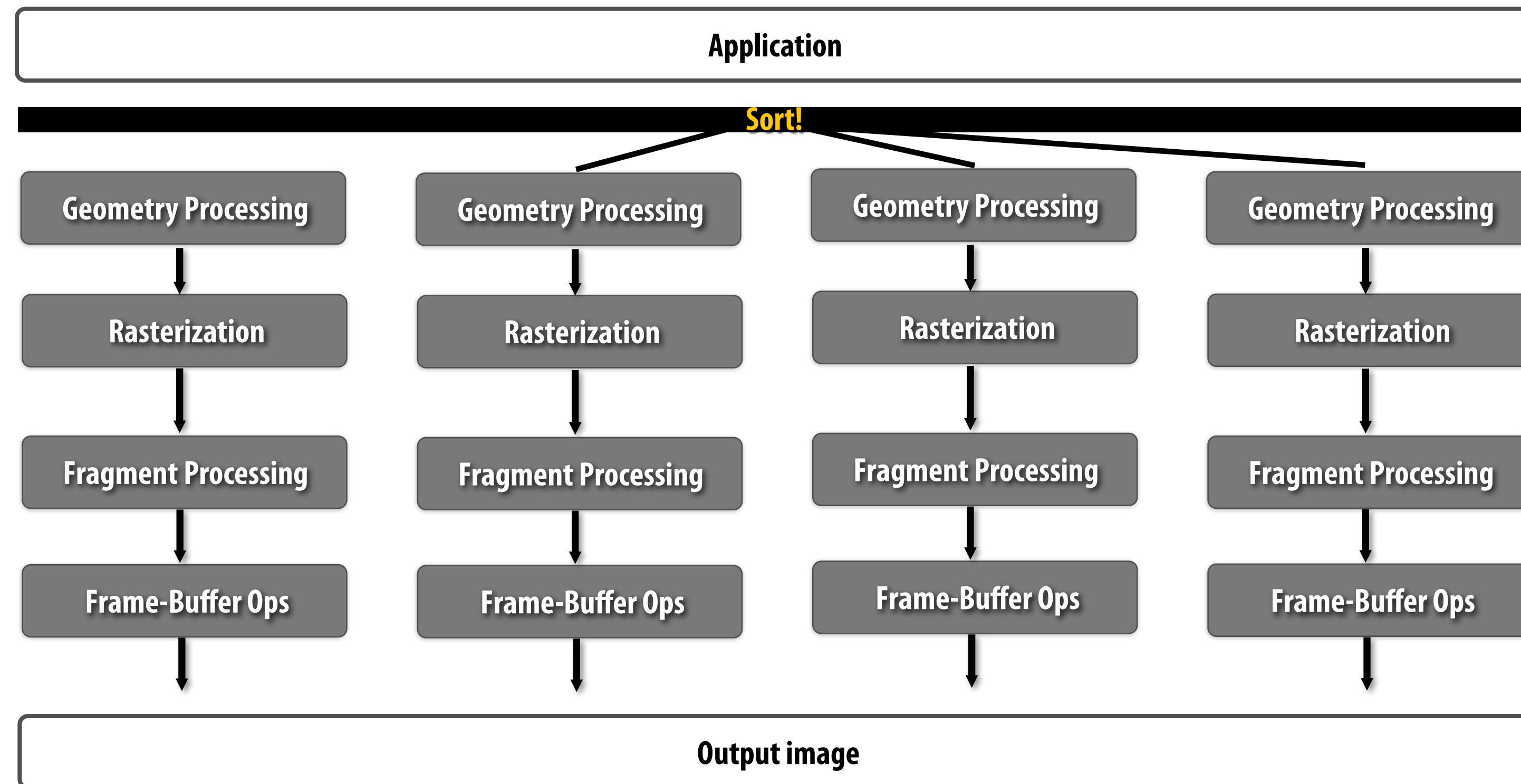
First do minimal amount of work (compute screen-space vertex positions of triangle) to determine which region(s) each input primitive overlaps

Sort first work partitioning

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



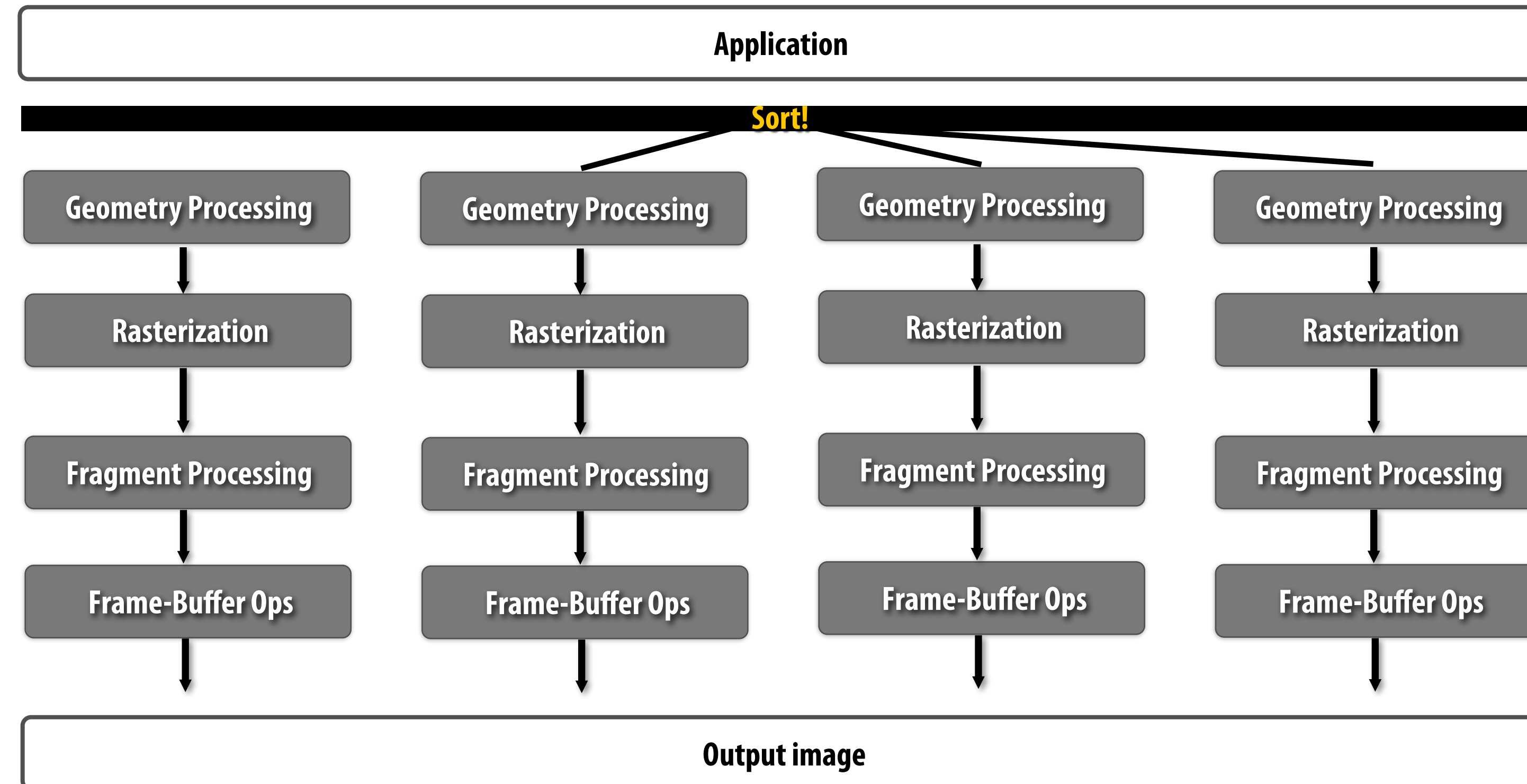
Sort first



■ Good:

- Simple parallelization: just replicate rendering pipeline and operate independently in screen regions (order maintained in each)
- More parallelism = more performance
- Small amount of sync/communication (communicate original triangles)
- Early fine occlusion cull ("early z") just as easy as single pipeline

Sort first



■ Bad:

- Potential for workload imbalance (one part of screen contains most of scene)
- Extra cost of triangle “pre-transformation” (needed to sort)
- “Tile spread”: as screen tiles get smaller with more parallelism, primitives cover more tiles (duplicate geometry processing across multiple parallel pipelines)

Sort first examples

- **WireGL/Chromium*** (parallel rendering with a cluster of GPUs)
 - “Front-end” node sorts primitives to machines
 - Each GPU is a full rendering pipeline (responsible for part of screen)



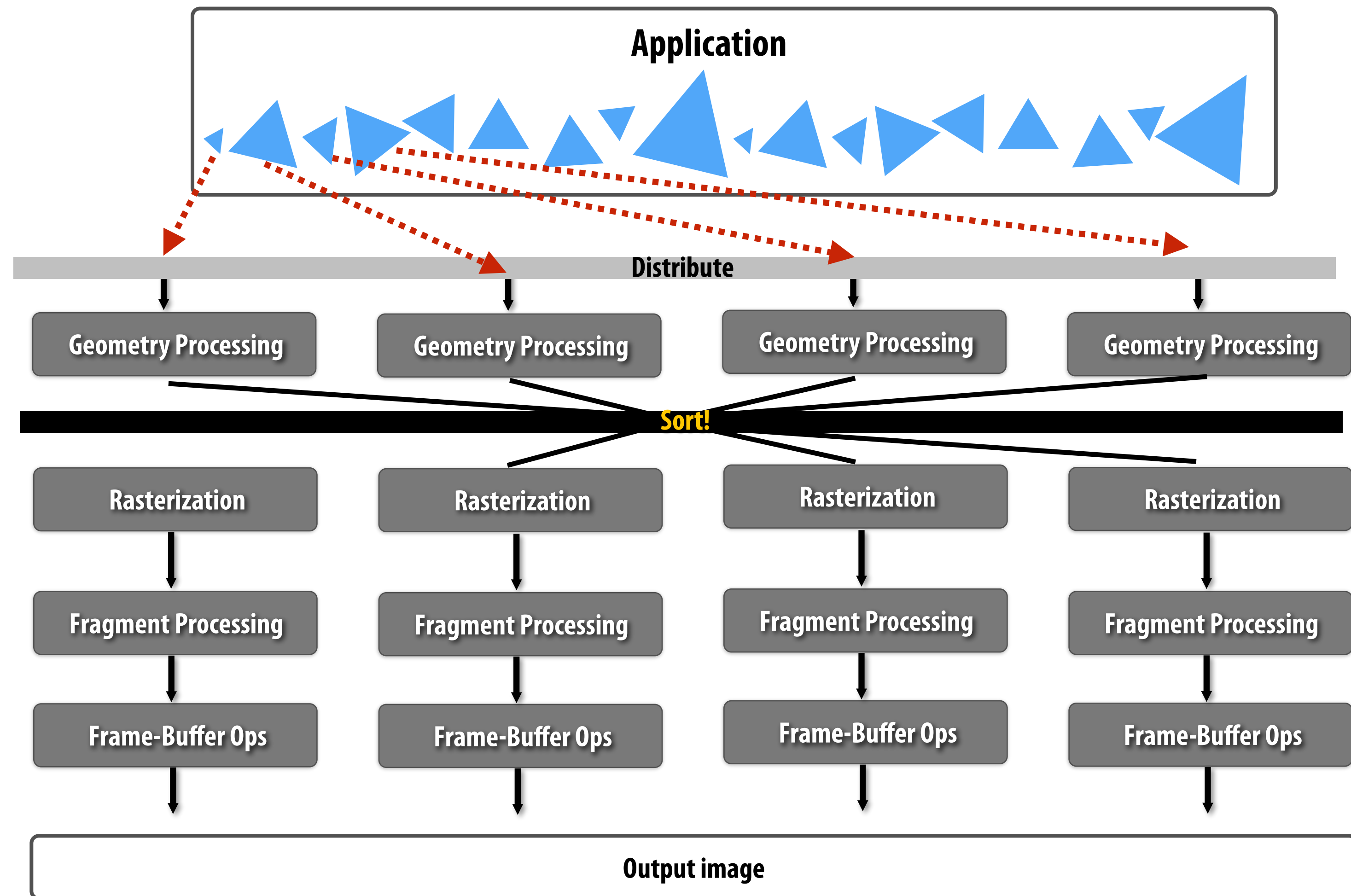
- **Initial parallel versions of Pixar's RenderMan**
 - Multi-core software renderer
 - Sort surfaces into screen tiles prior to tessellation



* Chromium can also be configured as a sort-last image composition system

Sort middle

Sort middle



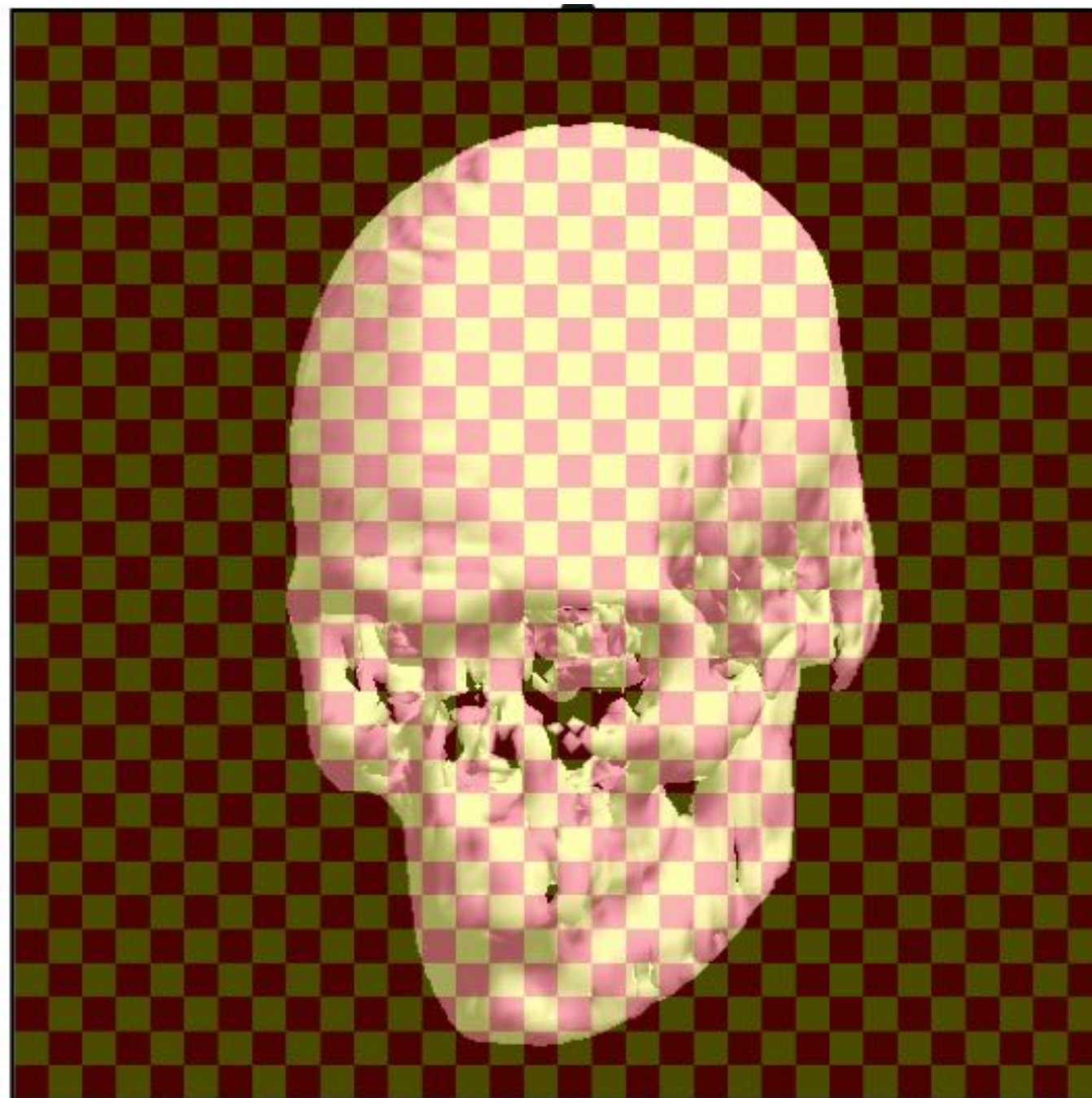
Distribute primitives to pipelines (e.g., round-robin distribution)

Assign each rasterizer a region of the render target (aka the output image)

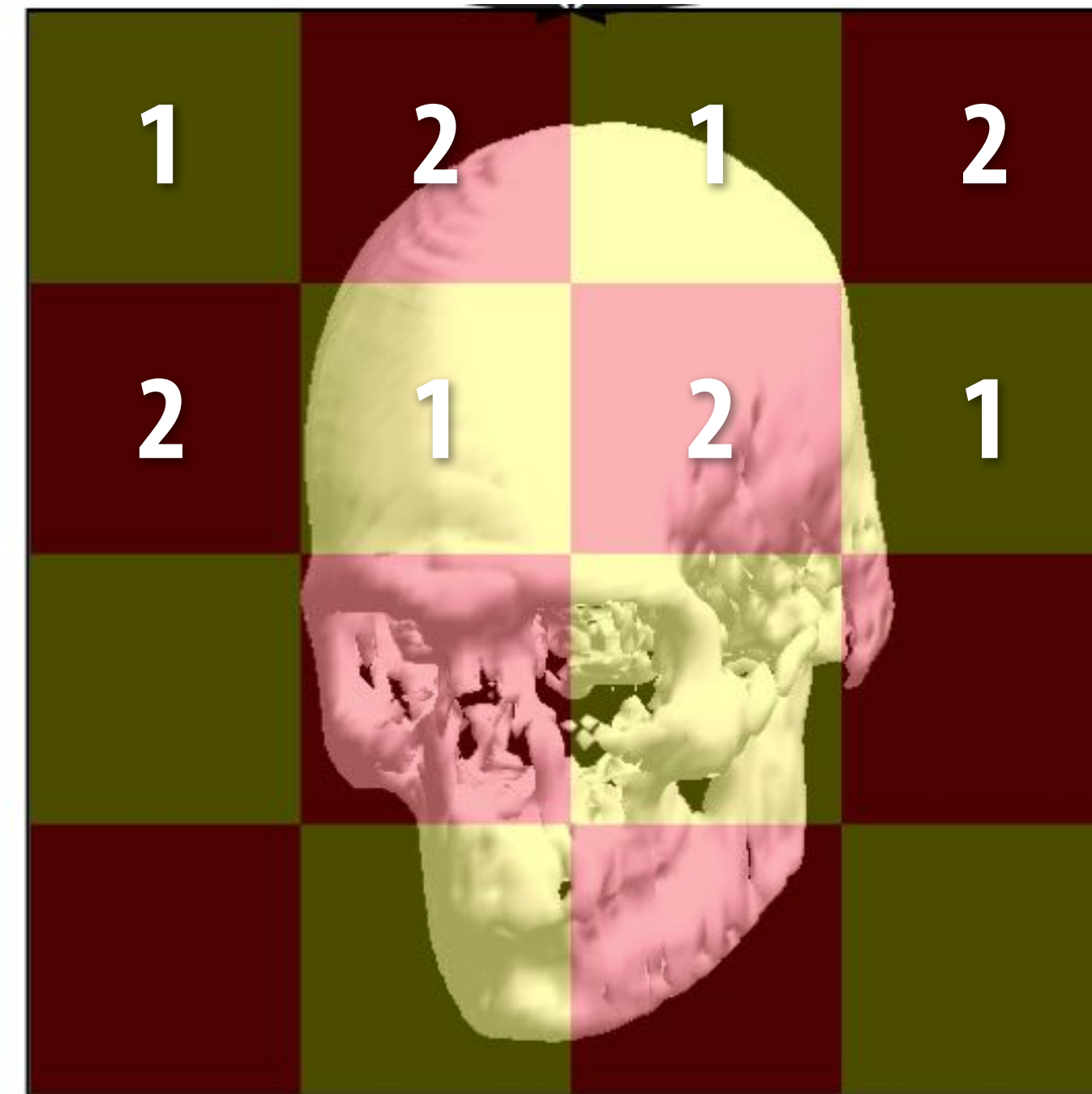
Sort after geometry processing based on screen space projection of primitive vertices

Interleaved mapping of screen

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



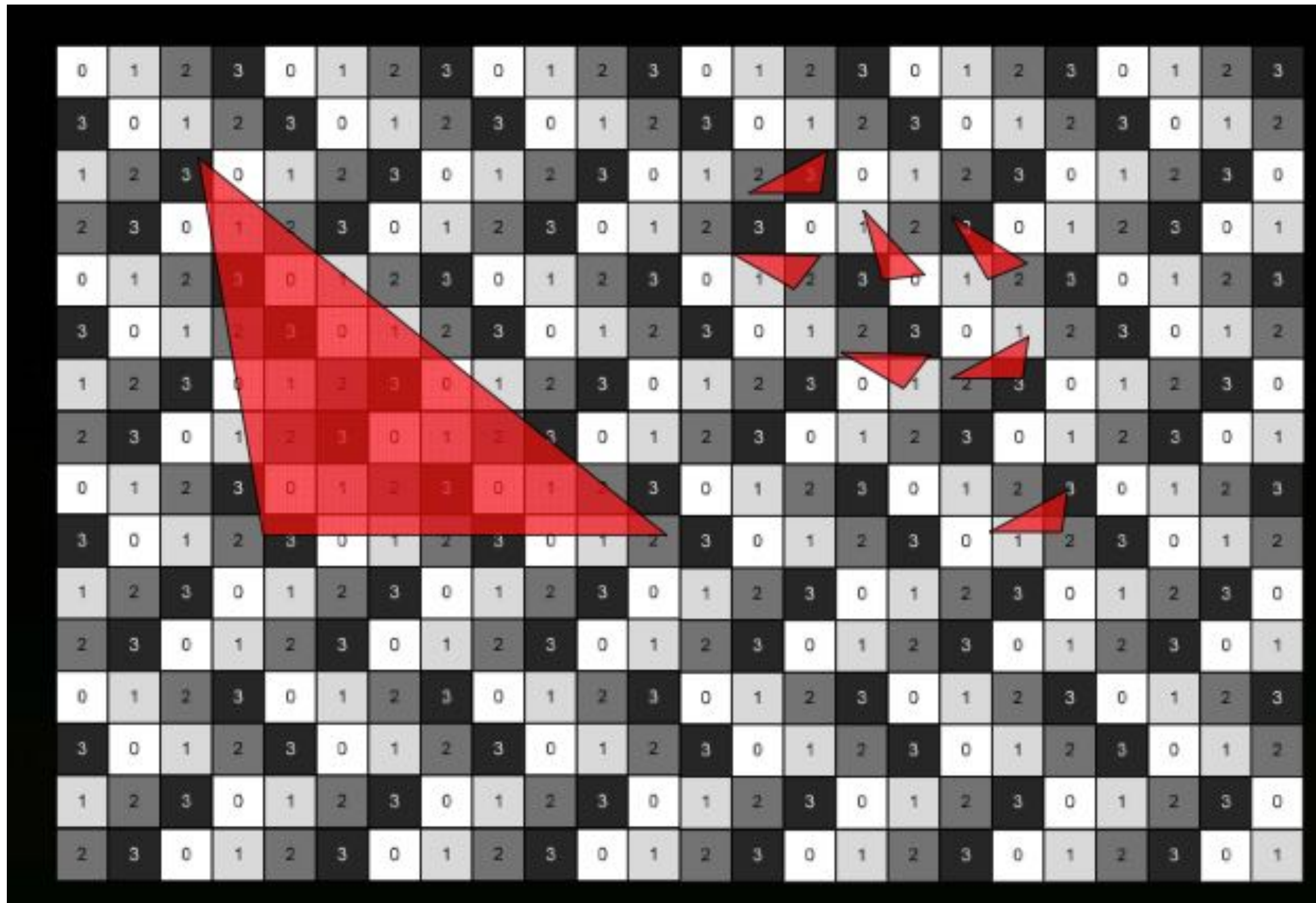
Interleaved mapping



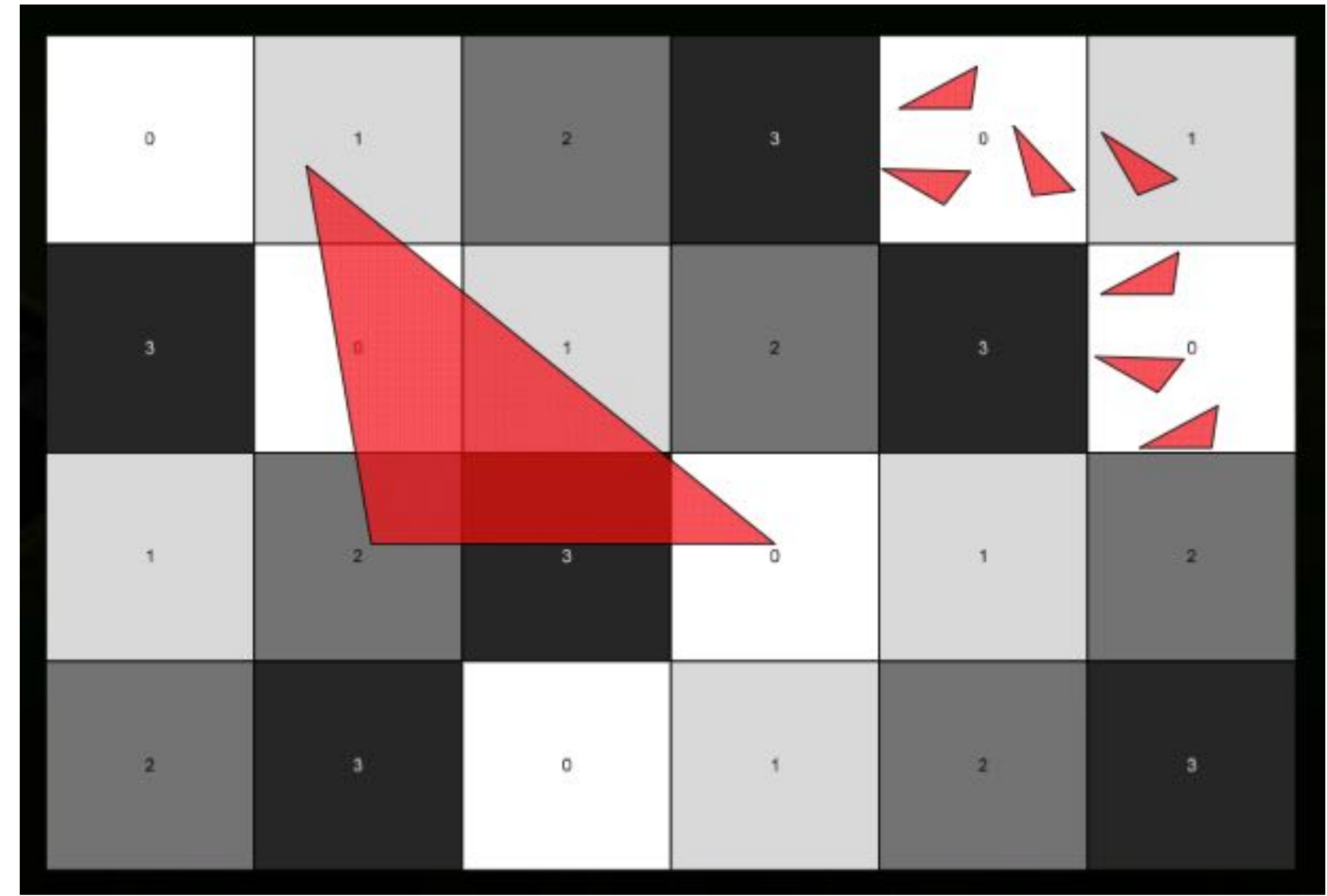
Tiled mapping

Fragment interleaving in NVIDIA Fermi

Fine granularity interleaving



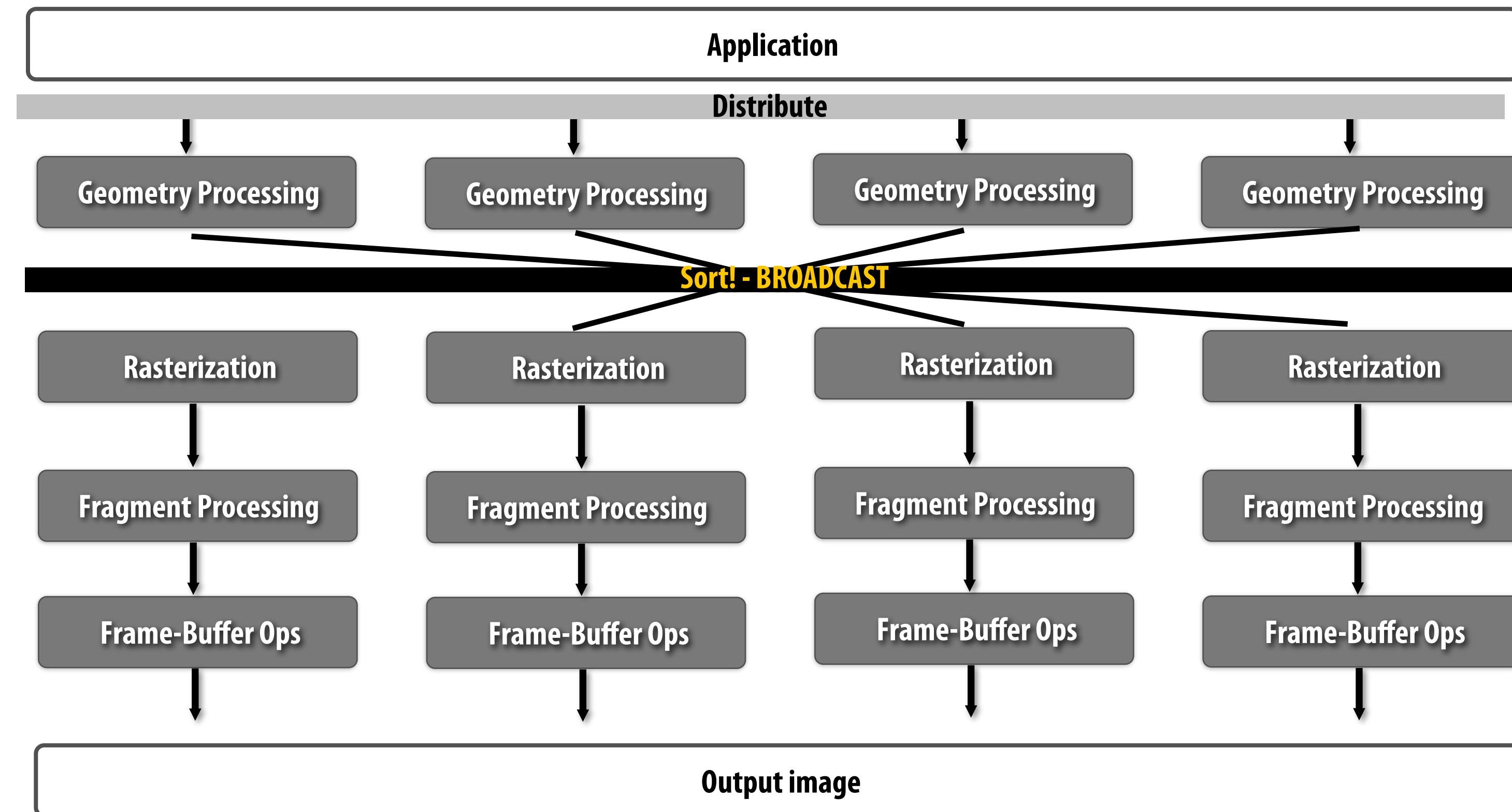
Coarse granularity interleaving



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

Question 2: notice anything interesting about these patterns?

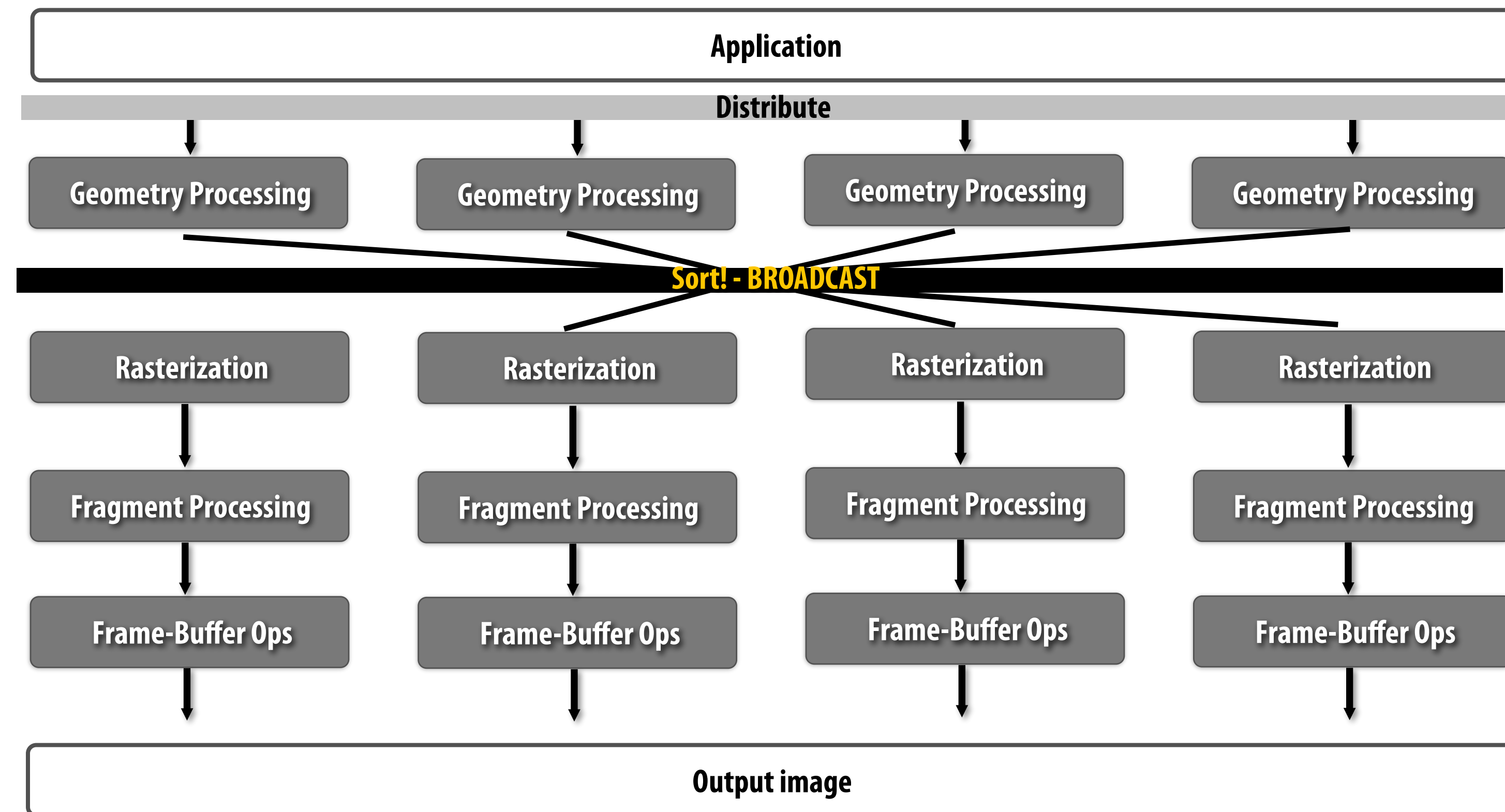
Sort middle interleaved



■ 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



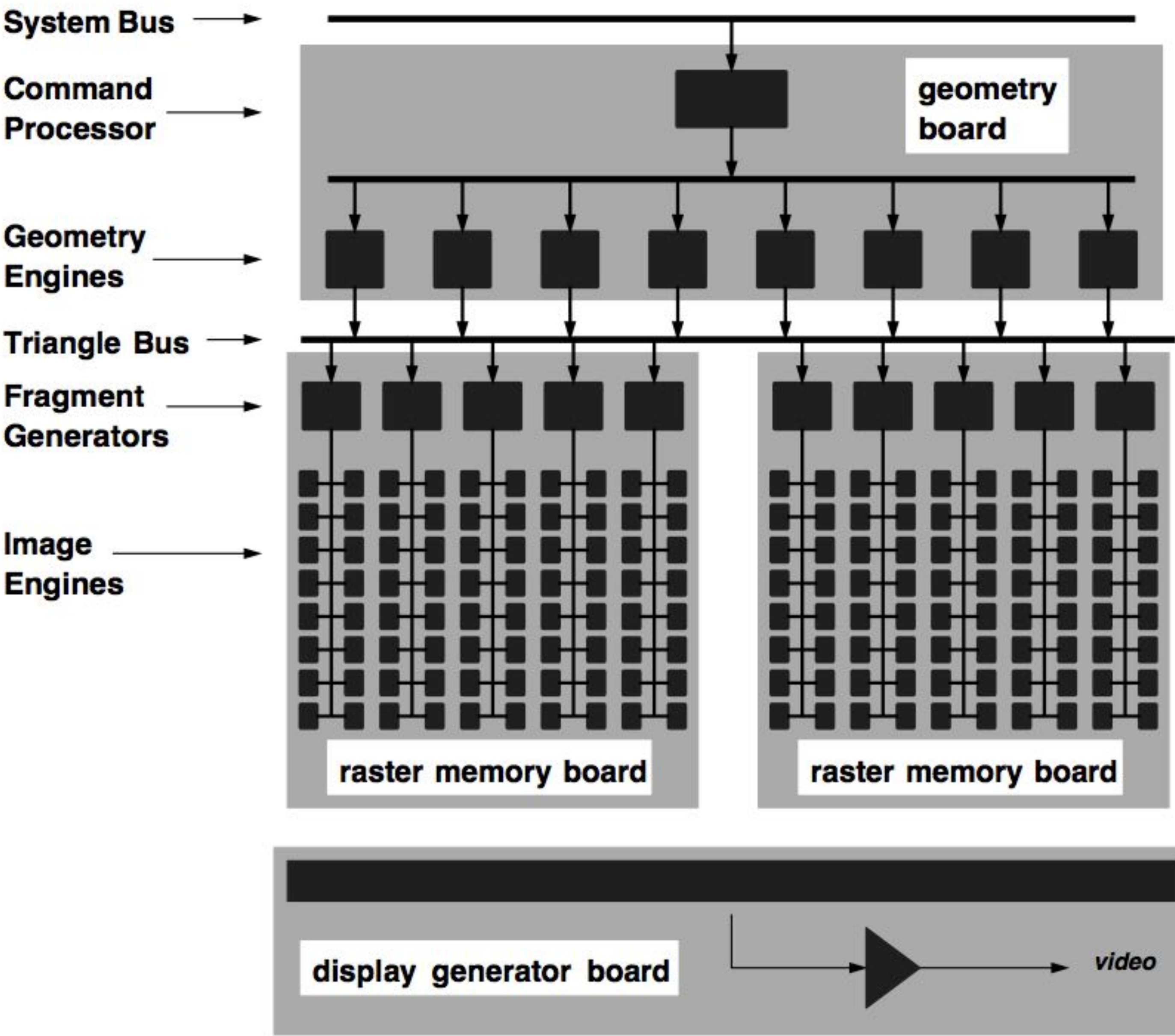
■ Bad:

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

SGI RealityEngine

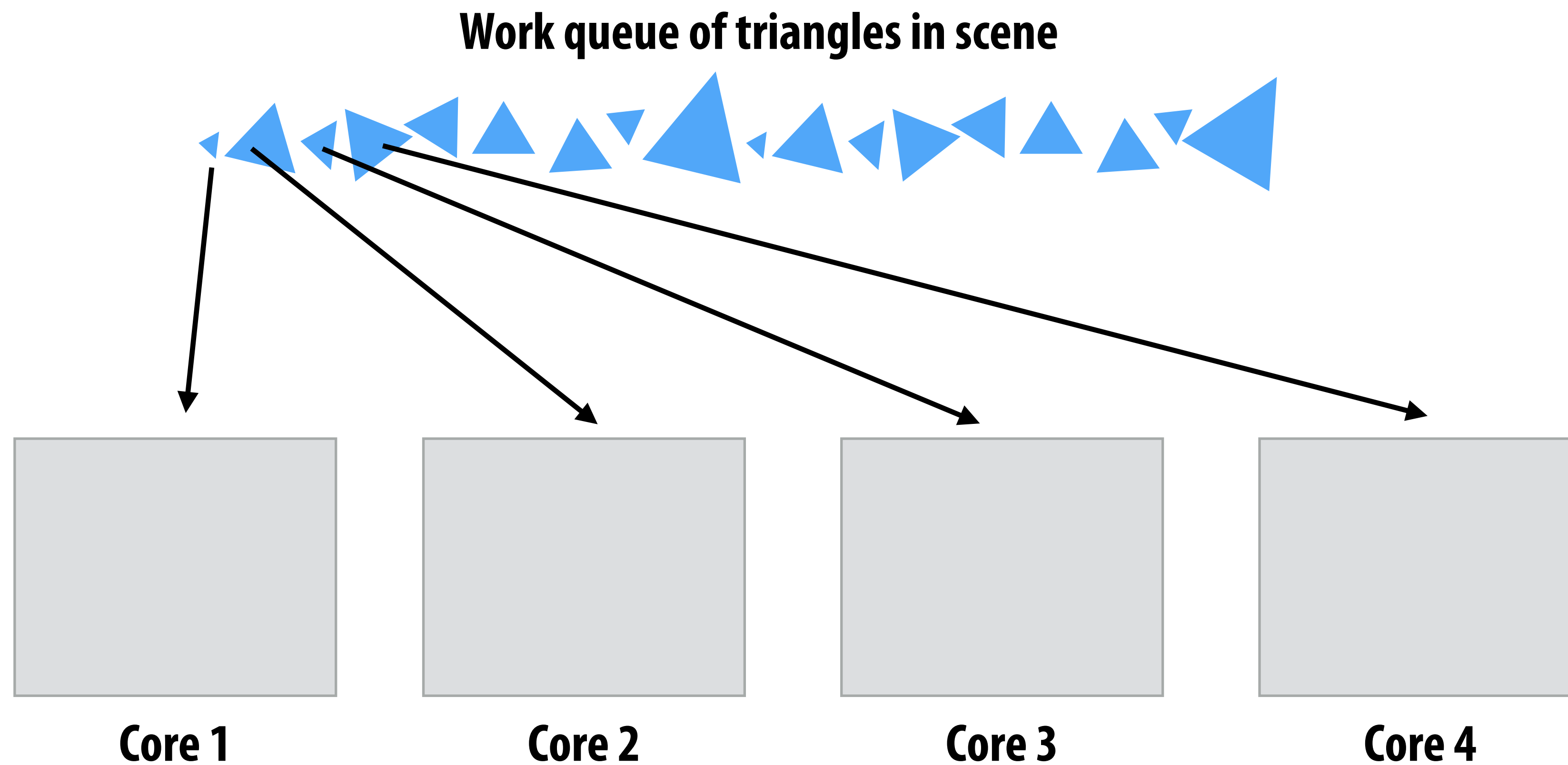
[Akeley 93]

Sort-middle interleaved design



Step 1: parallel geometry processing

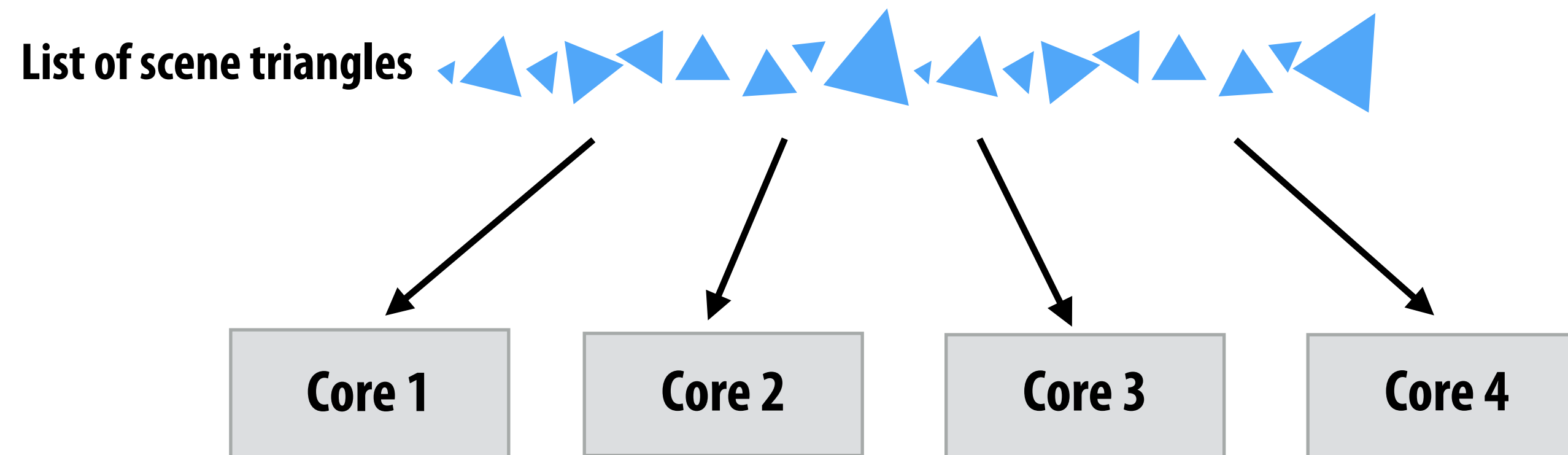
- Distribute triangles to the four processors (e.g., round robin)
- In parallel, processors perform vertex processing



Sort-middle tiled (a.k.a. “chunking”, “bucketing”, “binning”)

Step 1: sort triangles into bins







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



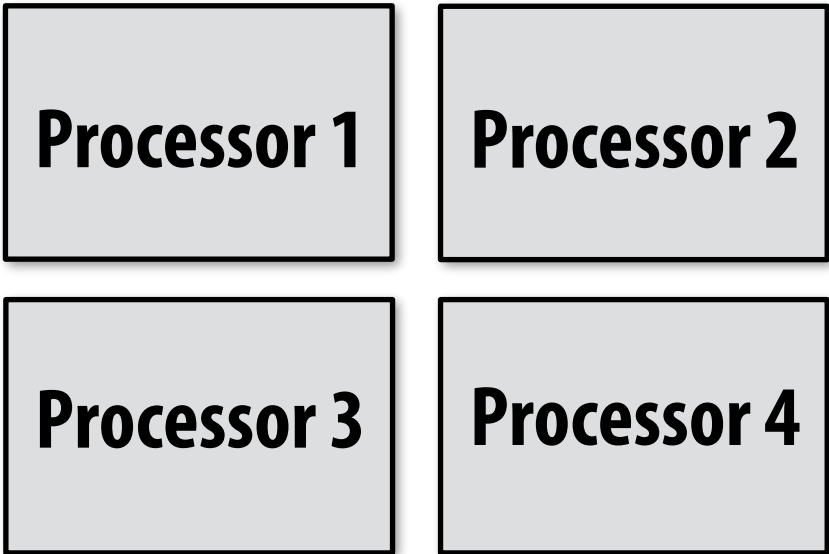
After processing first five triangles:

Bin 1 list: 1,2,3,4

Bin 2 list: 4,5

    Bin 1	  Bin 2	Bin 3	Bin 4
Bin 5	Bin 6	Bin 7	Bin 8
Bin 9	Bin 10	Bin 11	Bin 12

Sort-middle interleaved vs. binning



0	1	2	3	0	1
2	3	0	1	2	3
0	1	2	3	0	1
2	3	0	1	2	3

**Interleaved (static) assignment
of screen tiles to processors**
(Number above is the processor id that
processes each screen region)

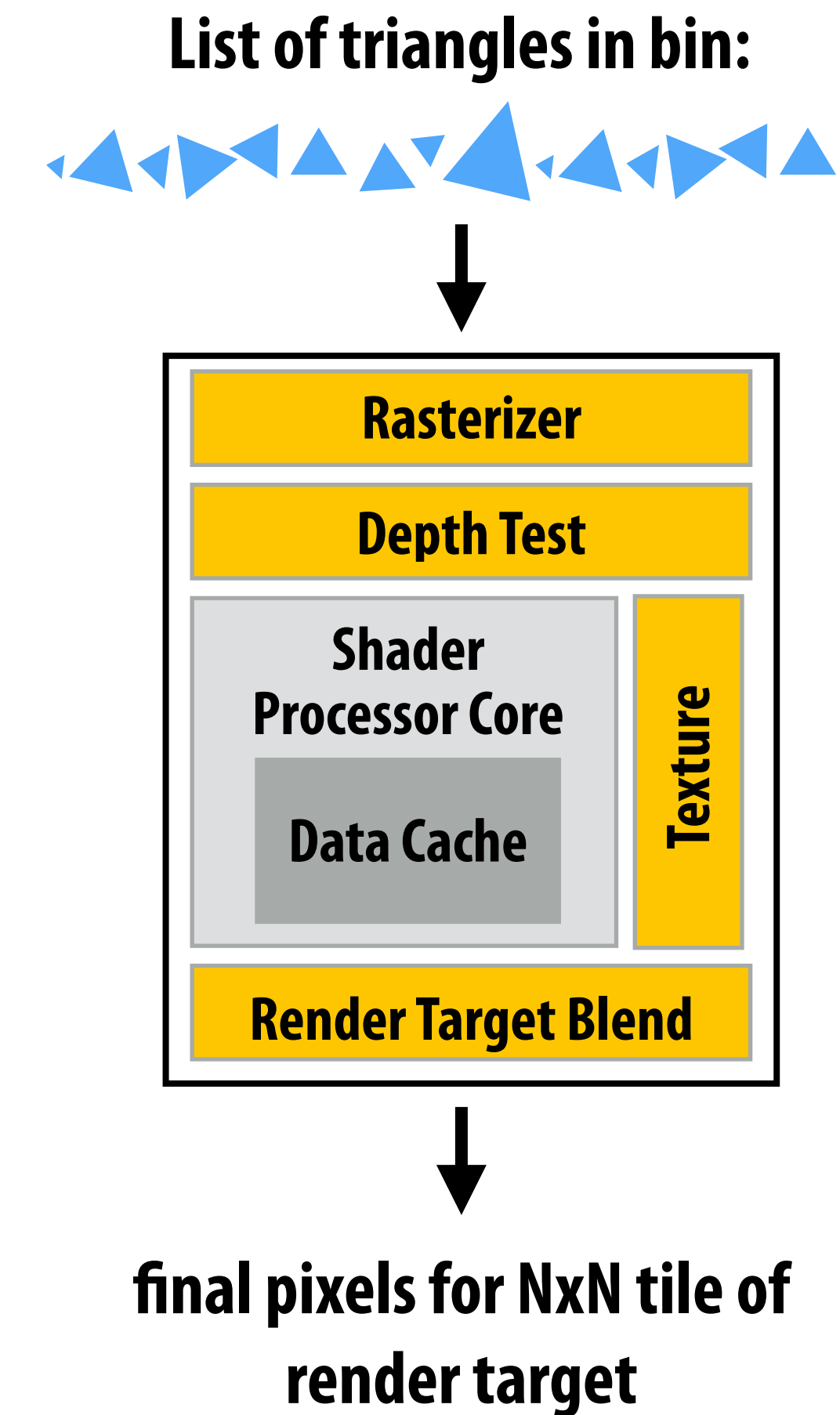
B0	B1	B2	B3	B4	B5
B6	B7	B8	B9	B10	B11
B12	B13	B14	B15	B16	B17
B18	B19	B20	B21	B22	B23

Assignment to bins

List of bins is a work queue. Bins are
dynamically assigned to processors.

Step 2: per-tile processing

- Cores process bins in parallel, performing rasterization fragment shading and frame buffer update
- While there are more bins to process:
 - Assign bin to available core
 - For all triangles:
 - Rasterize
 - Fragment shade
 - Depth test
 - Update frame buffer



Reminder: reading less data conserves power

- **Goal: redesign algorithms so that they make good use of on-chip memory or processor caches**
 - **And therefore transfer less data from memory**
- **A fact you might not have heard:**
 - It is *far more* costly (in energy) to load/store data from memory, than it is to perform an arithmetic operation

“Ballpark” numbers

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

Implications

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

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

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

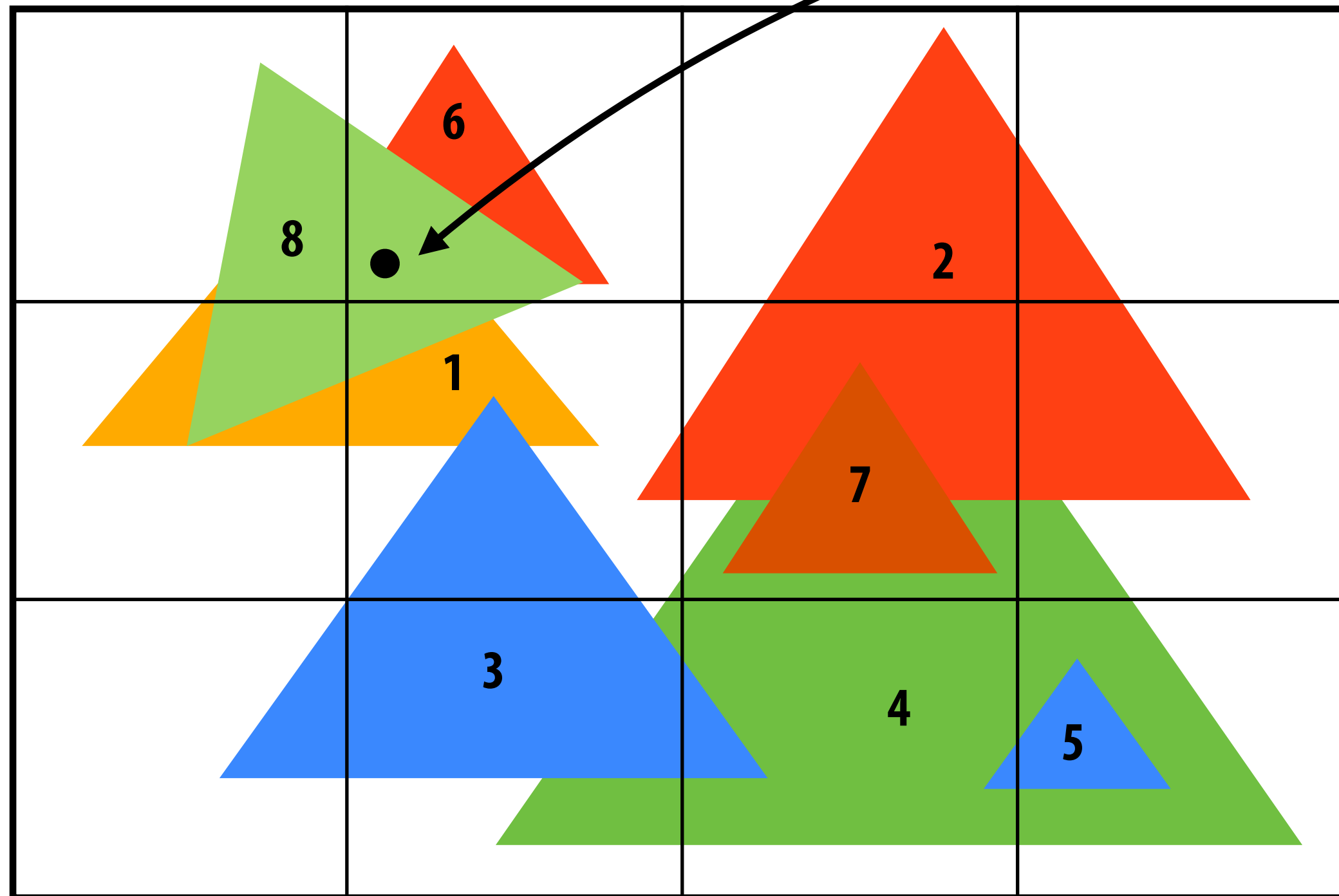
What should the screen size of the bins be?

- Small enough for a tile of the color buffer and depth buffer (potentially supersampled) to fit in a shader processor core's on-chip storage (i.e., cache)
- Tile sizes in range 16x16 to 64x64 pixels are common
- ARM Mali GPU: commonly uses 16x16 pixel tiles



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

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



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

Now consider step 2 of a tiled
renderer:

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

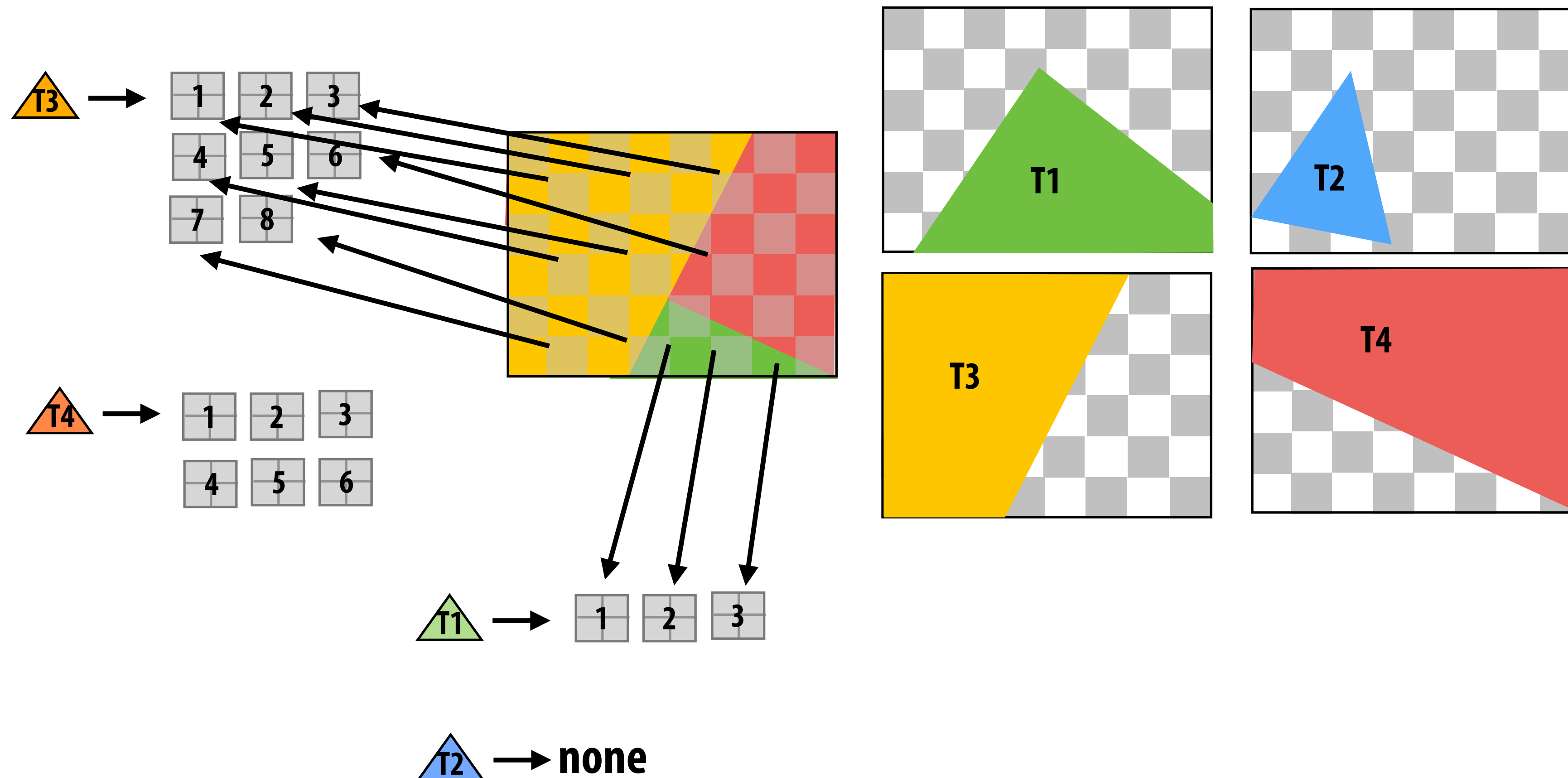
Q. Why doesn't the renderer need to read color or depth buffer from memory?

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

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

Tile-based deferred rendering (TBDR)

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



Sort middle tiled (chunked)

■ Good:

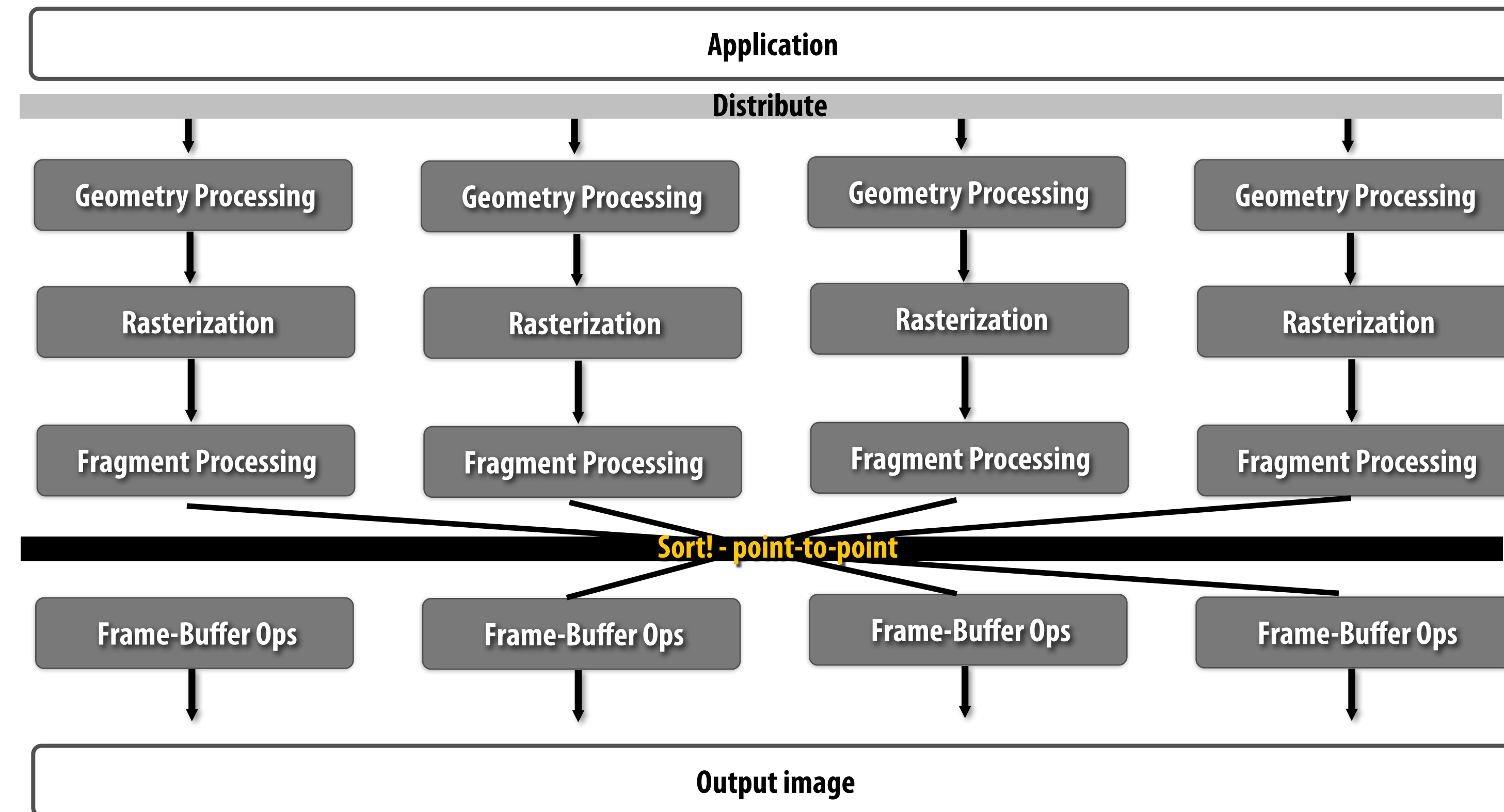
- Good load balance (distribute many buckets onto rasterizers)
- **Low bandwidth requirements (why? when?)**
- Challenge: “bucketing” sort has low contention (assuming each triangle only touches a small number of buckets), but there still is contention

■ Recent examples:

- Many mobile GPUs: Imagination PowerVR, ARM Mali, Qualcomm Adreno
- Parallel software rasterizers
 - Intel Larrabee software rasterizer
 - NVIDIA CUDA software rasterizer

Sort last

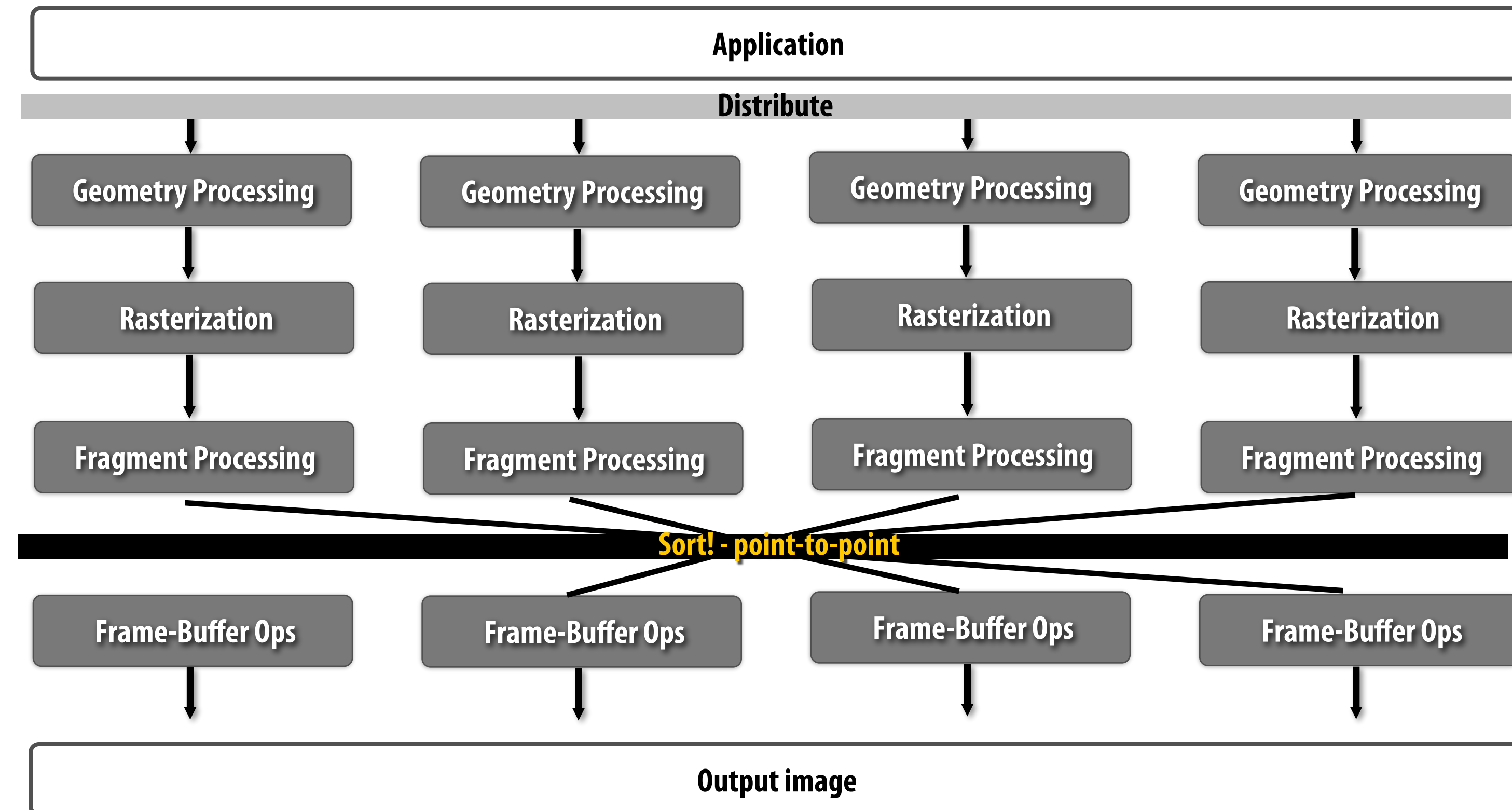
Sort last fragment



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

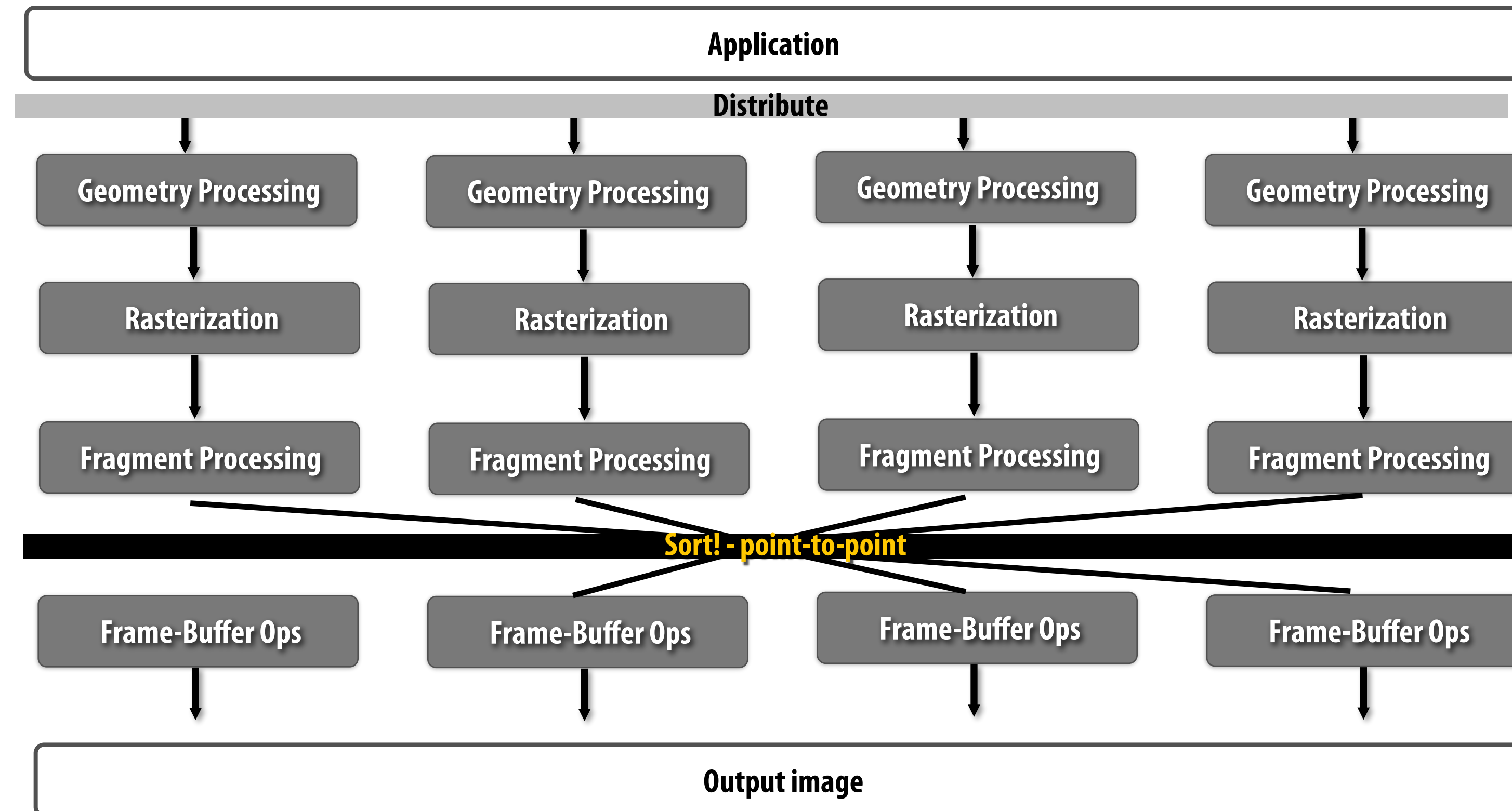
Sort after fragment processing based on (x,y) position of fragment

Sort last fragment



- **Good:**
 - No redundant geometry processing or rasterization (but early 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

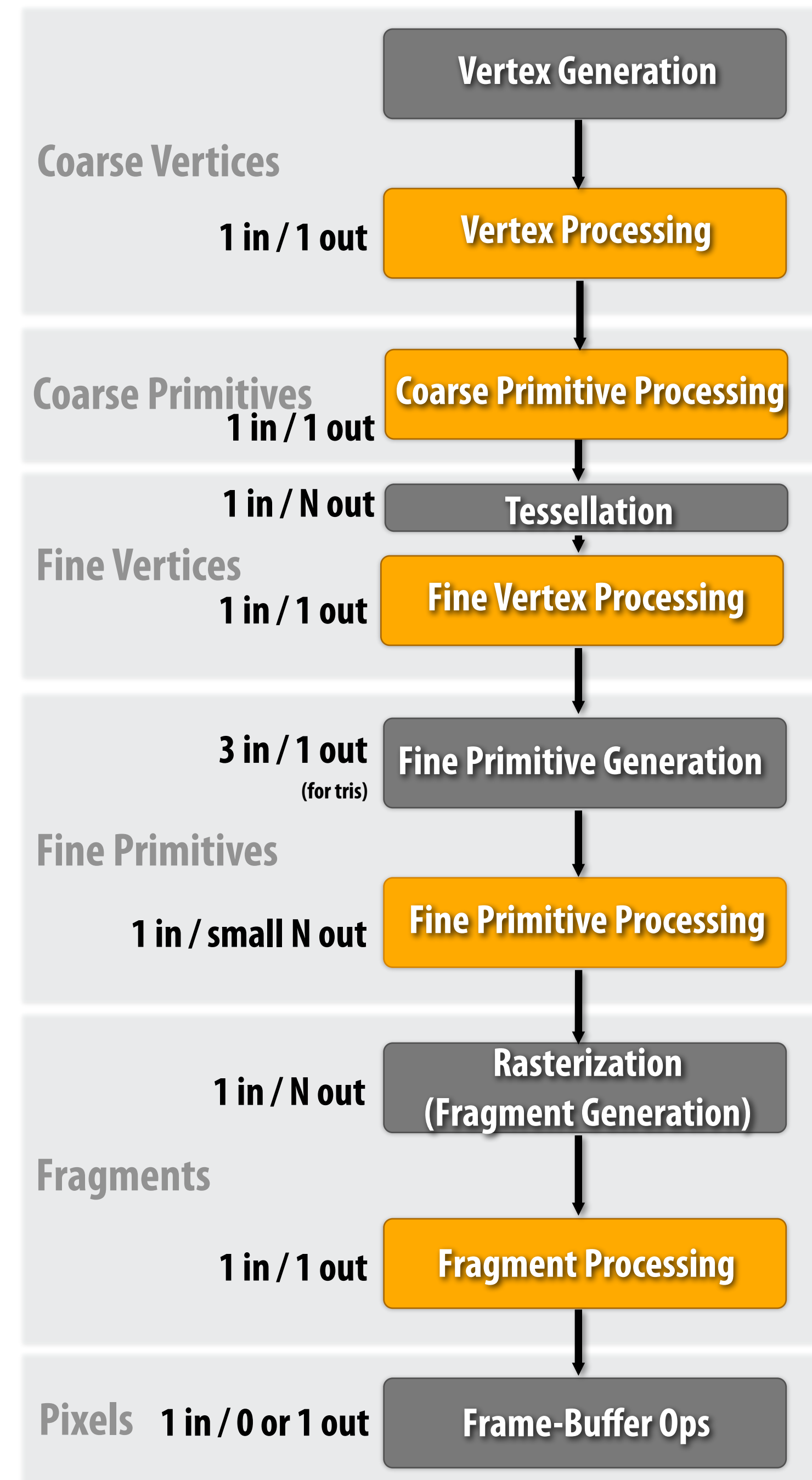


■ Bad:

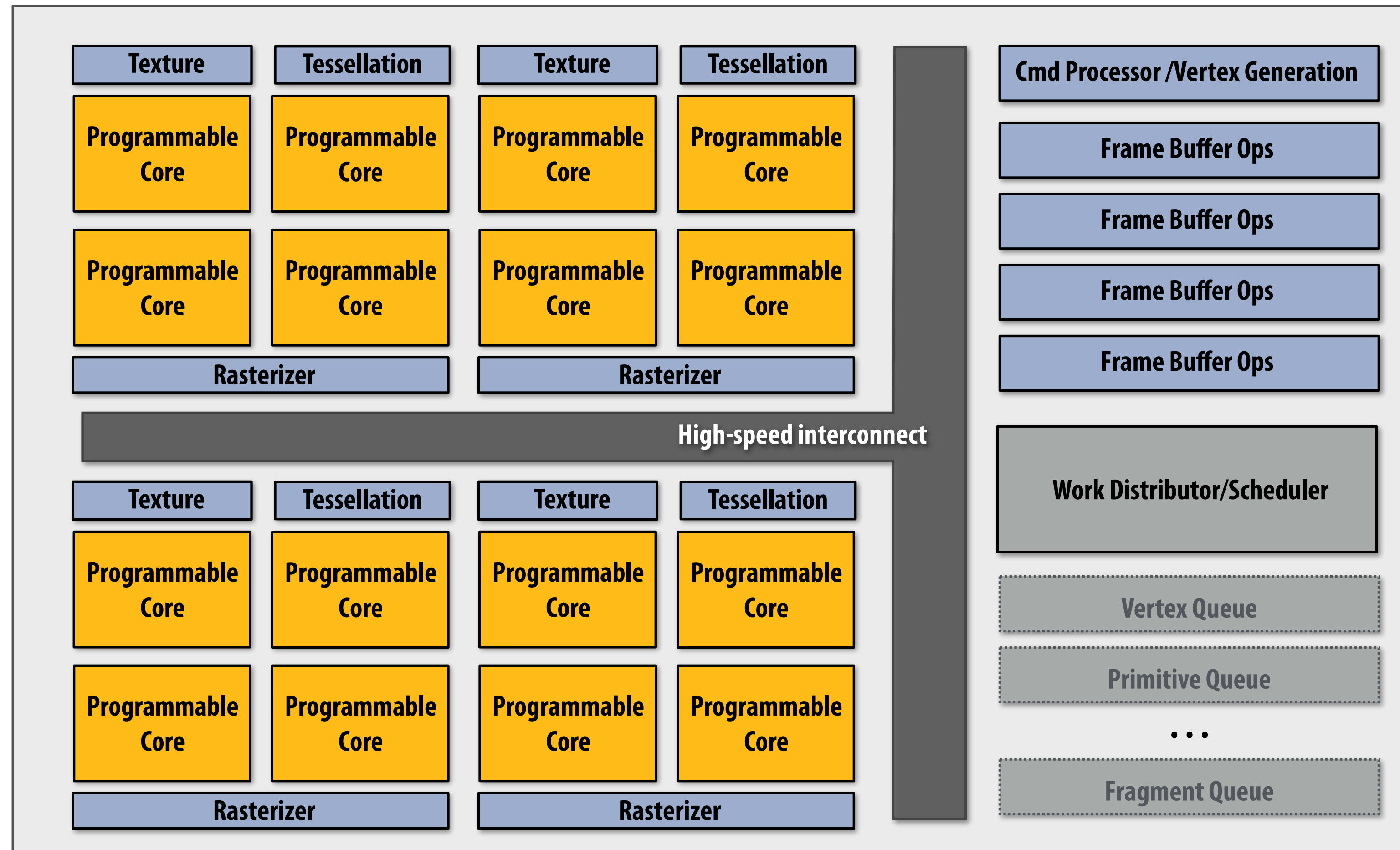
- Pipelines may stall due to primitives of varying size (due to order requirement)
- Bandwidth scaling: many more fragments than triangles
- Hard to implement early occlusion cull (more bandwidth challenges)

OpenGL 4 / Direct3D 11 pipeline

Five programmable stages



Modern GPU: programmable parts of pipeline virtualized on pool of programmable cores



Hardware is a heterogeneous collection of resources (programmable and non-programmable)

Programmable resources are time-shared by vertex/primitive/fragment processing work

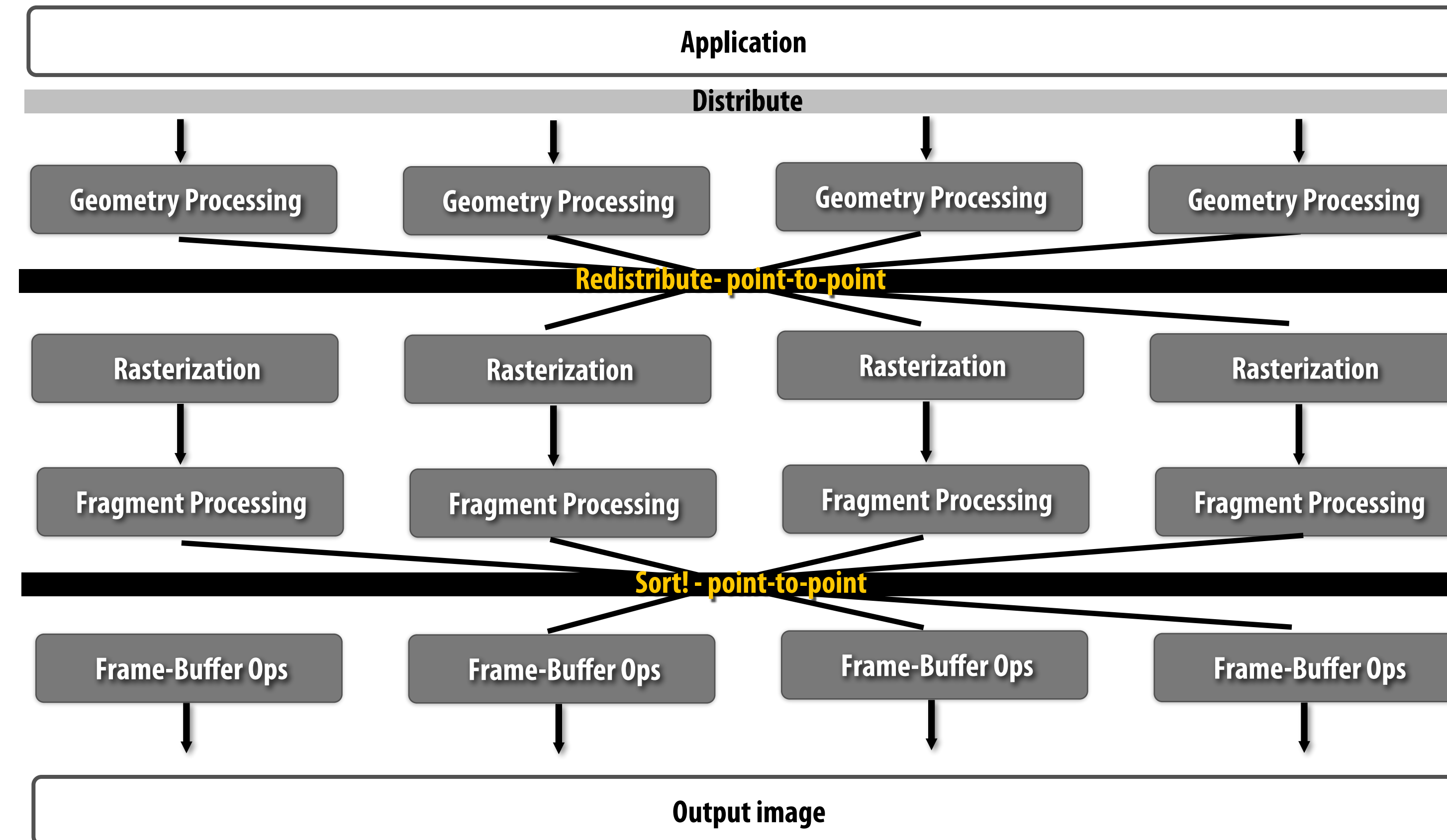
Must keep programmable cores busy: sort everywhere

Hardware work distributor assigns work to cores (based on contents of inter-stage queues)

Sort everywhere

(How modern high-end GPUs are scheduled)

Sort everywhere



Distribute primitives to top of pipelines

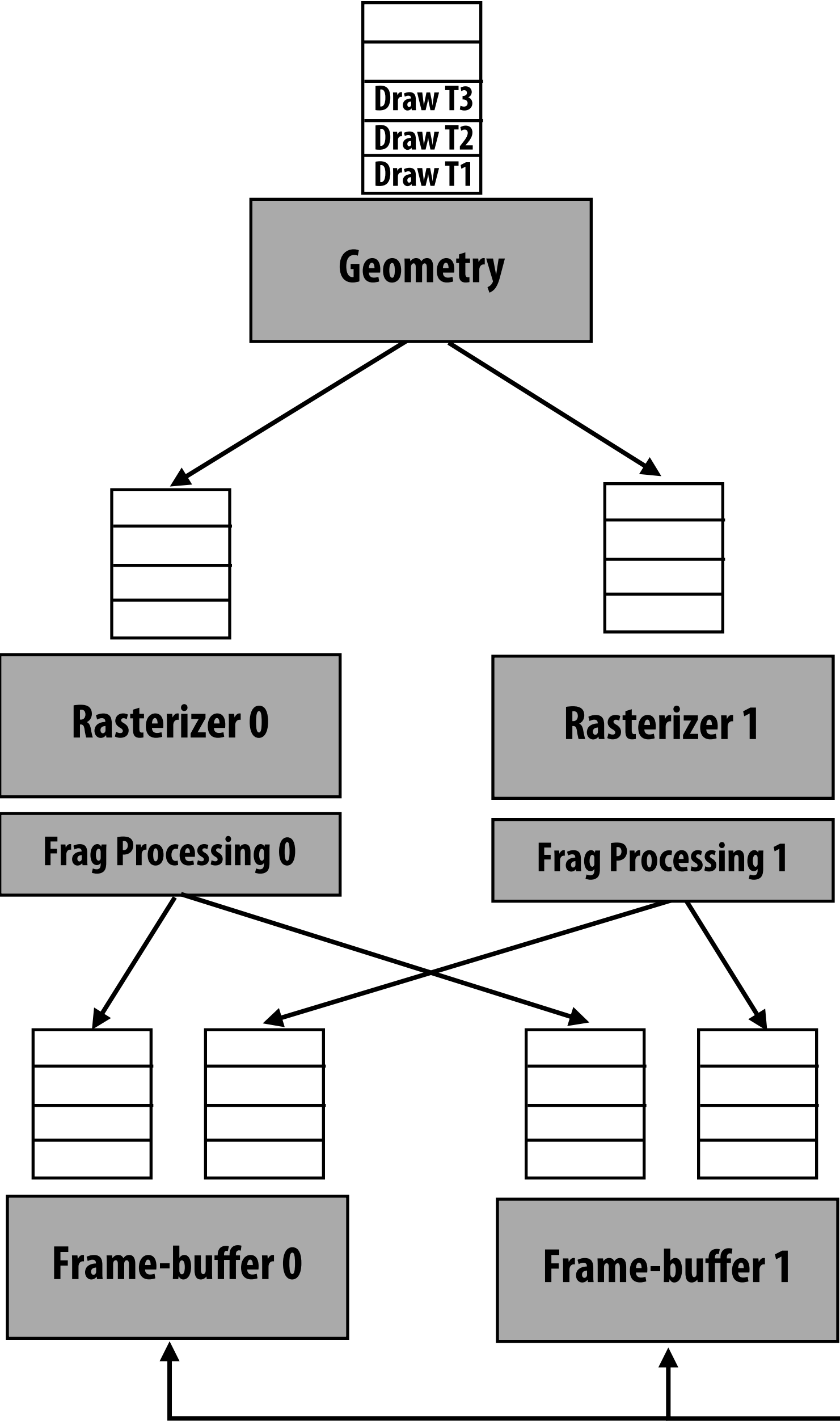
Redistribute after geometry processing (e.g, round robin)

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

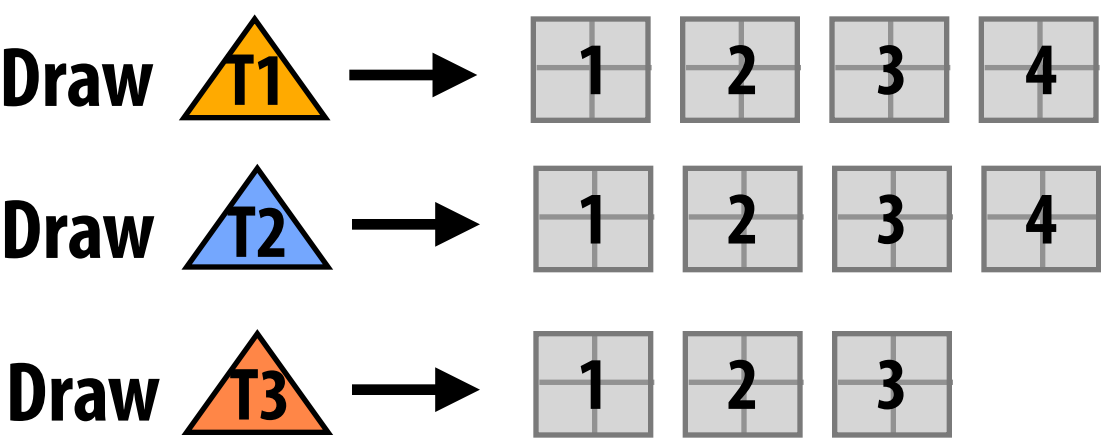
Implementing sort everywhere

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

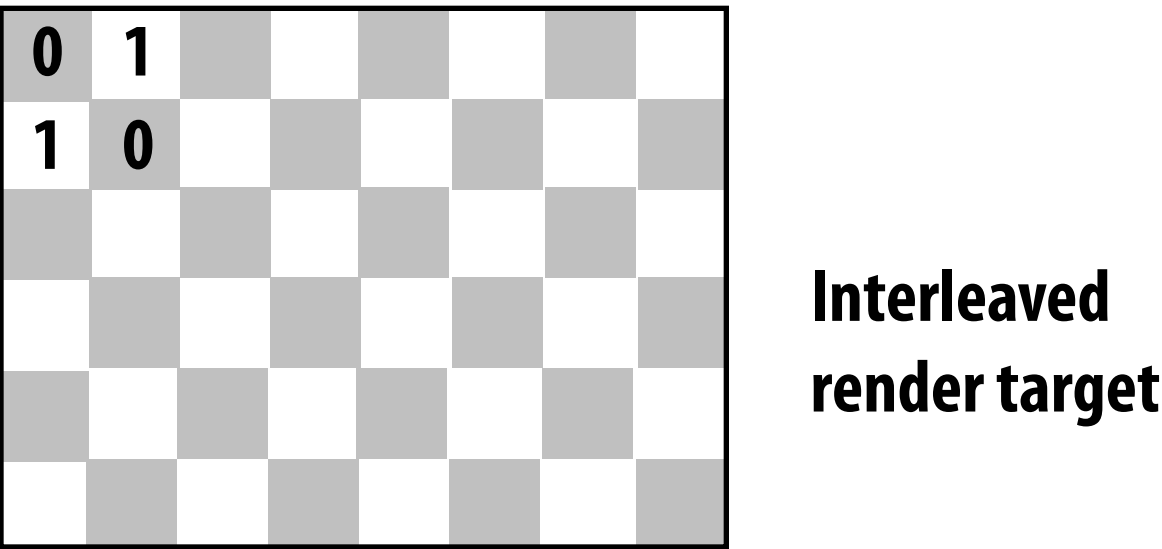
Starting state: draw commands enqueued for pipeline



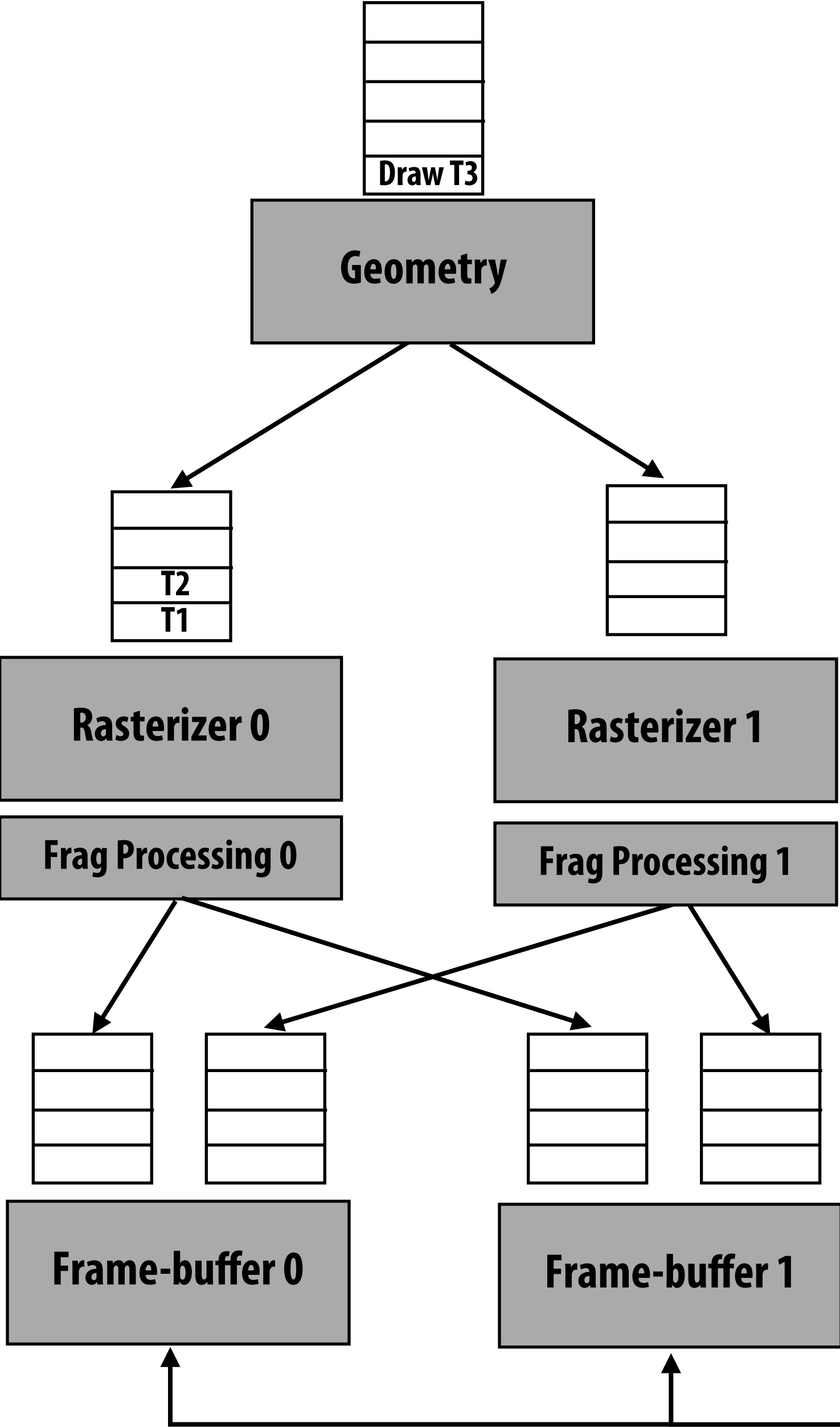
Input: three triangles to draw
(fragments to be generated for each triangle by rasterization are shown below)



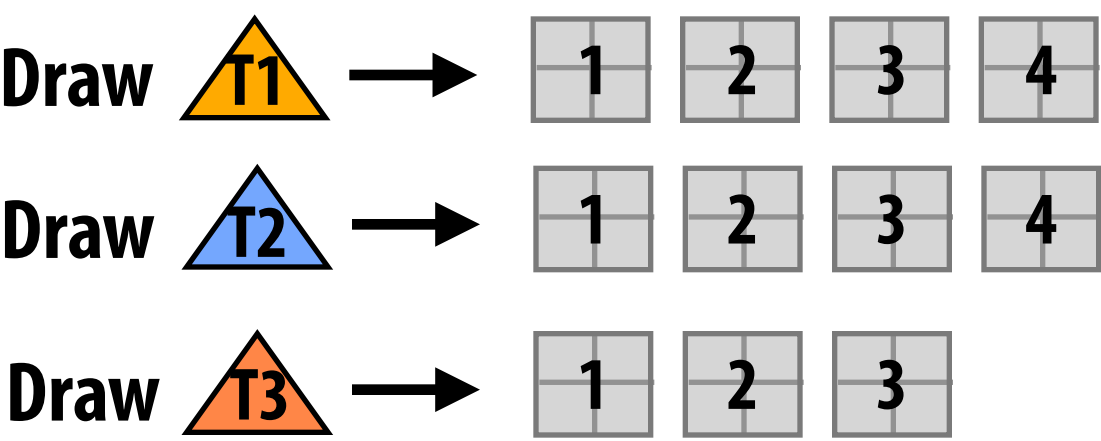
Assume batch size is 2 for assignment to rasterizers.



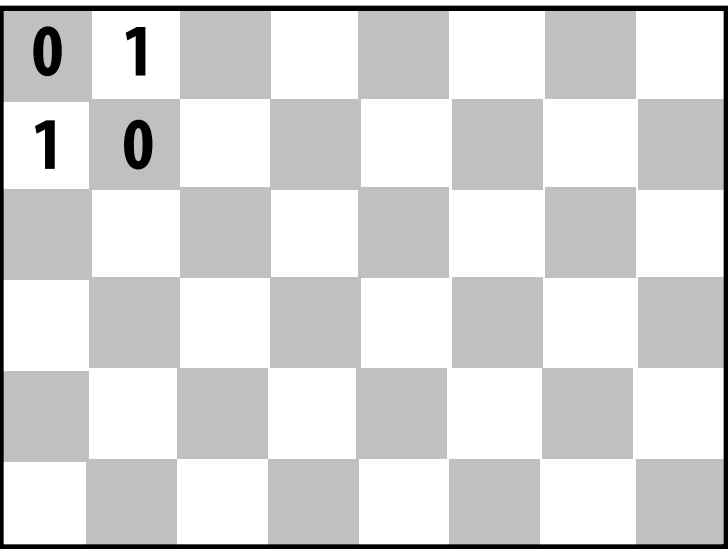
After geometry processing, first two processed triangles assigned to rast 0



Input:



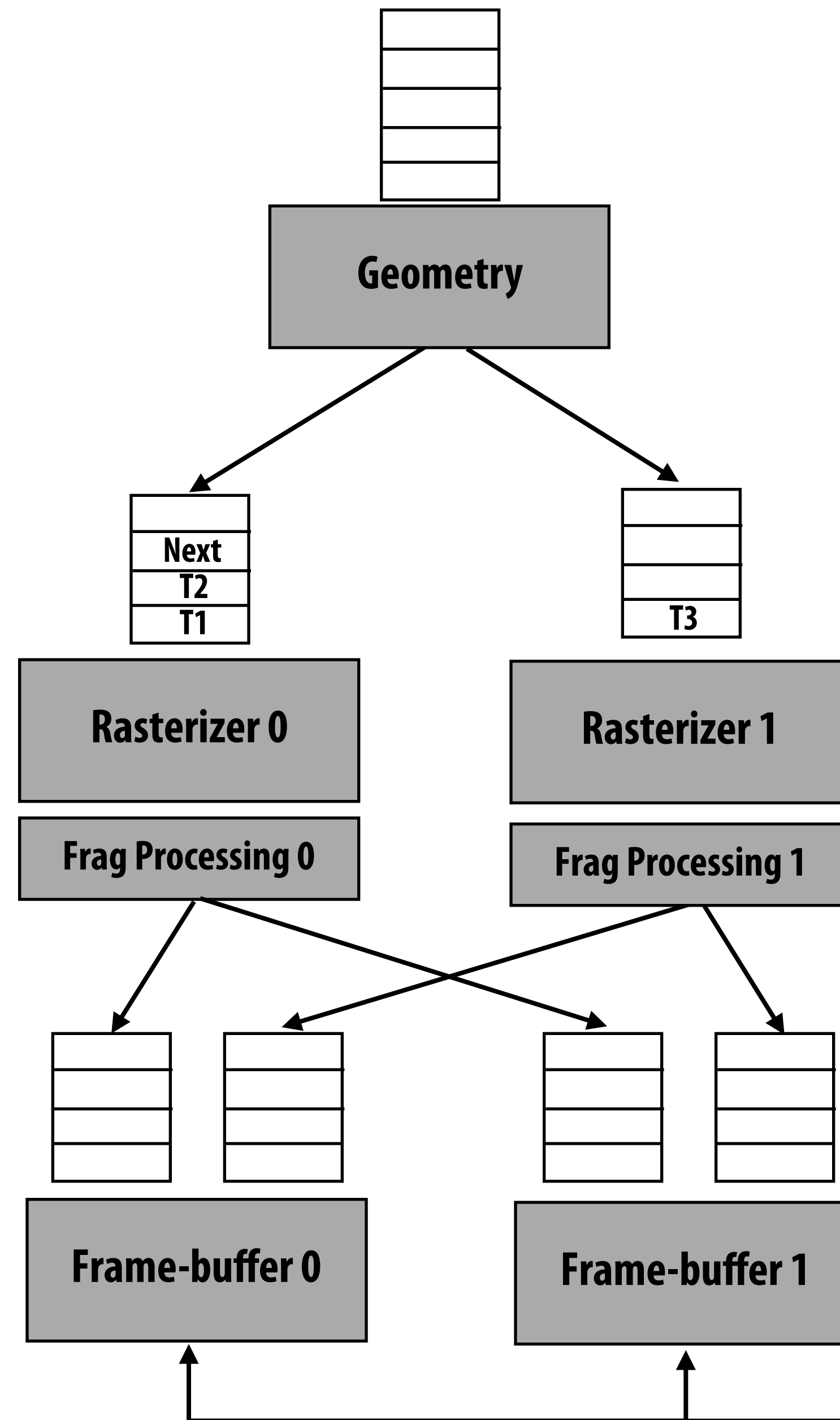
Assume batch size is 2 for assignment to rasterizers.



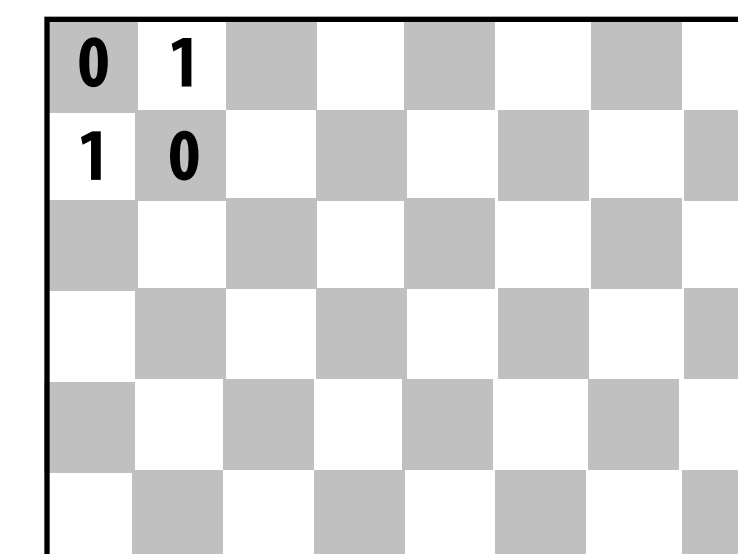
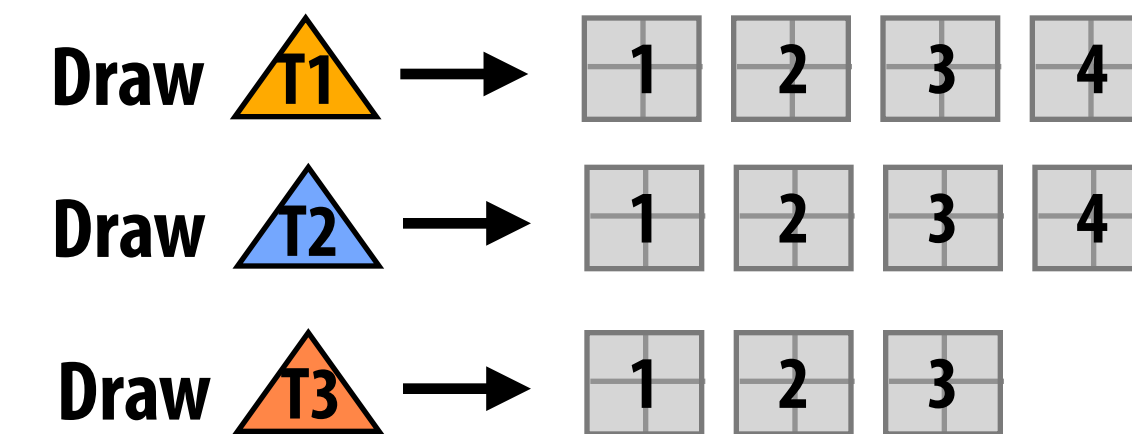
Interleaved render target

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

Q. What is the 'next' token for?



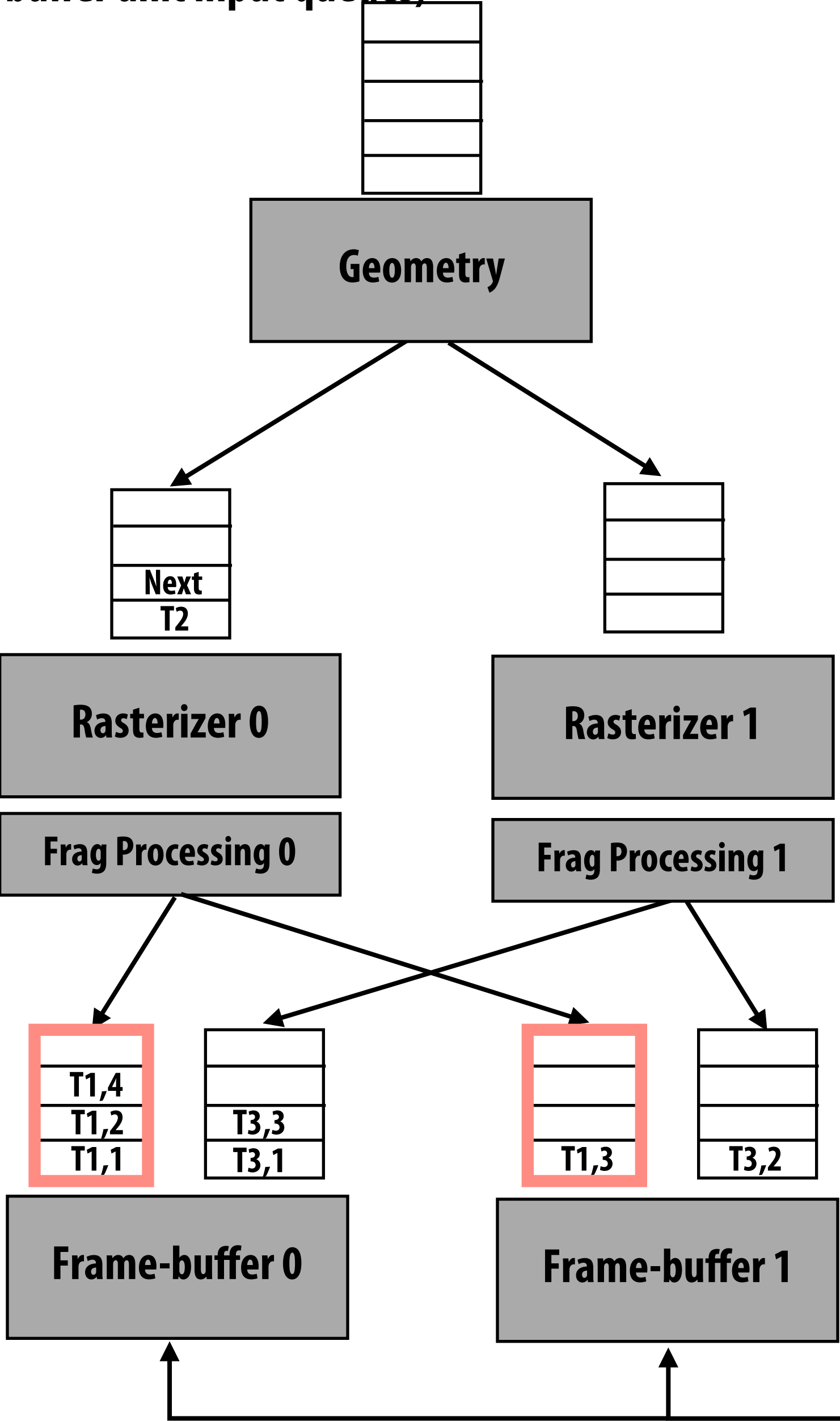
Input:



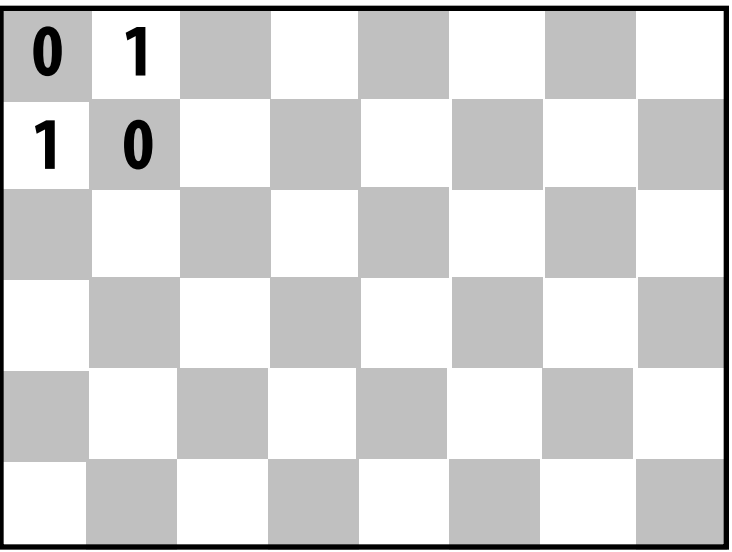
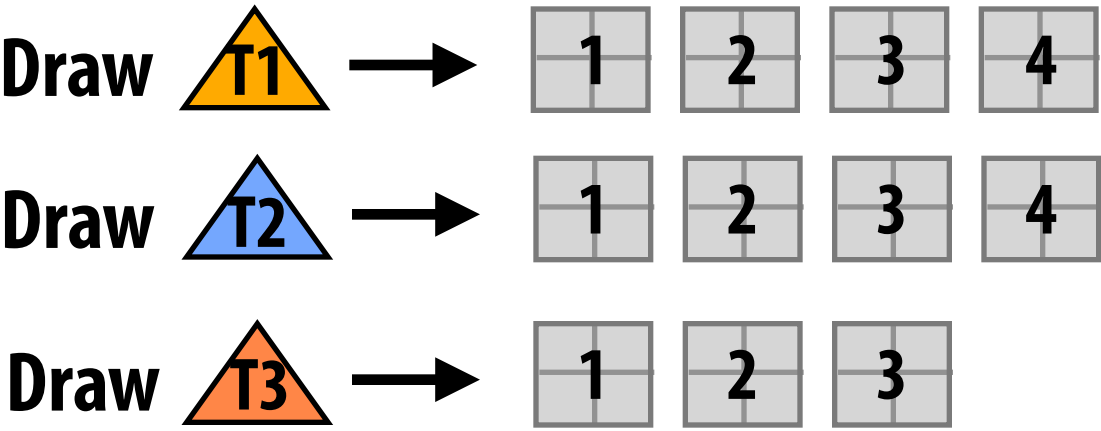
Interleaved
render target

Rast 0 and rast 1 can process T1 and T3 simultaneously

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



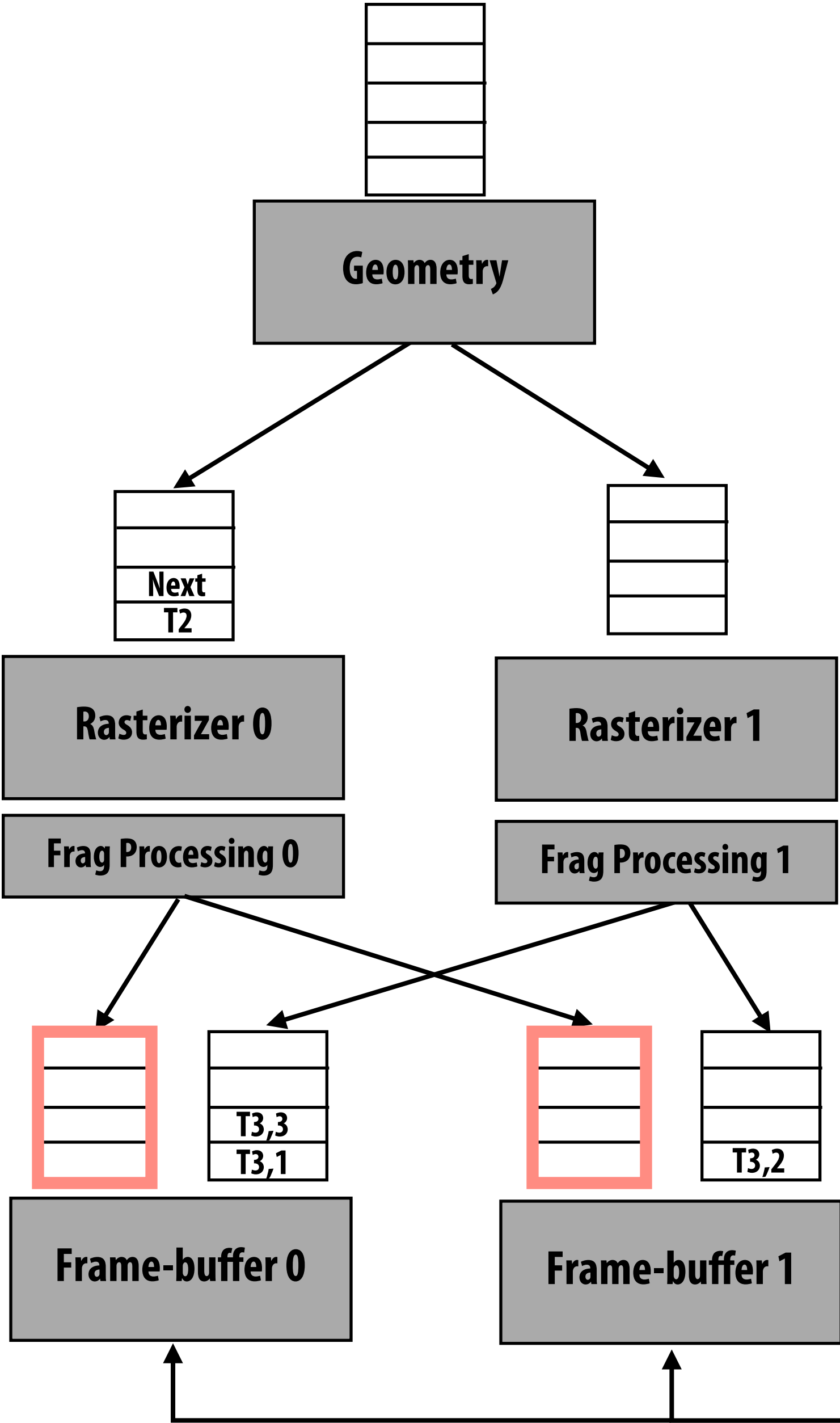
Input:



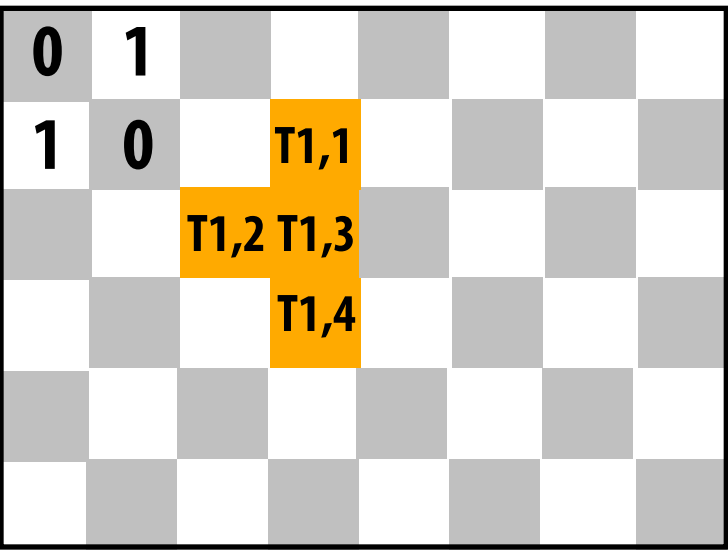
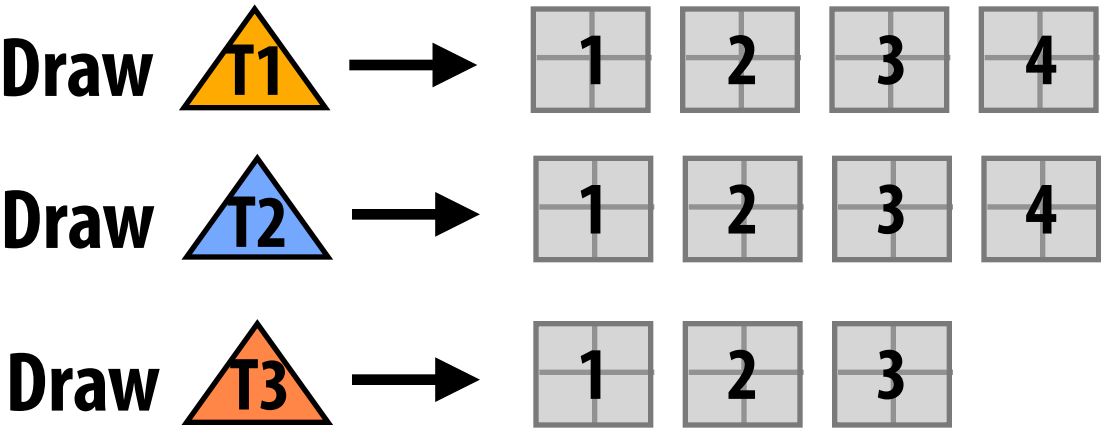
Interleaved
render target

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

(Notice updates to frame buffer)

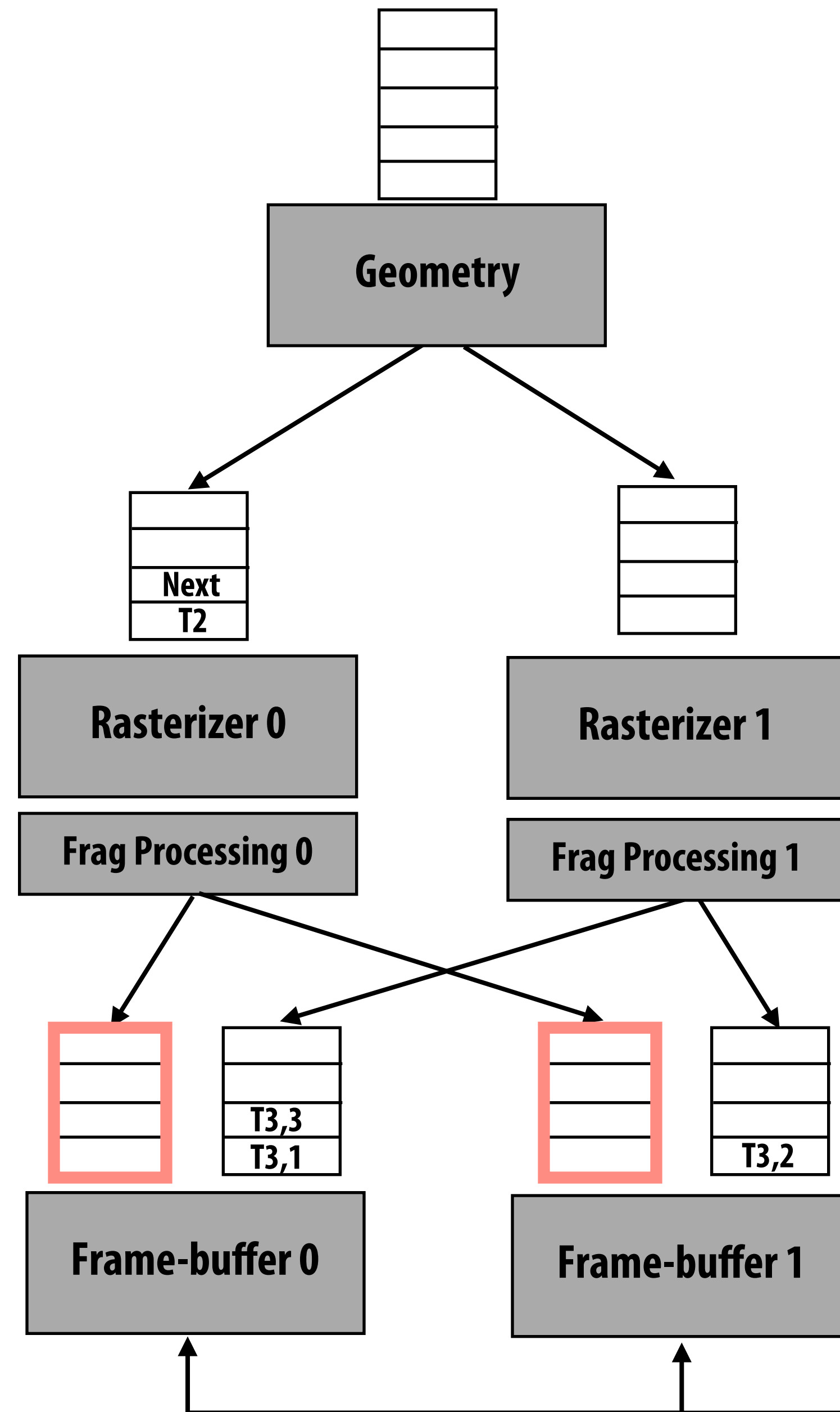


Input:

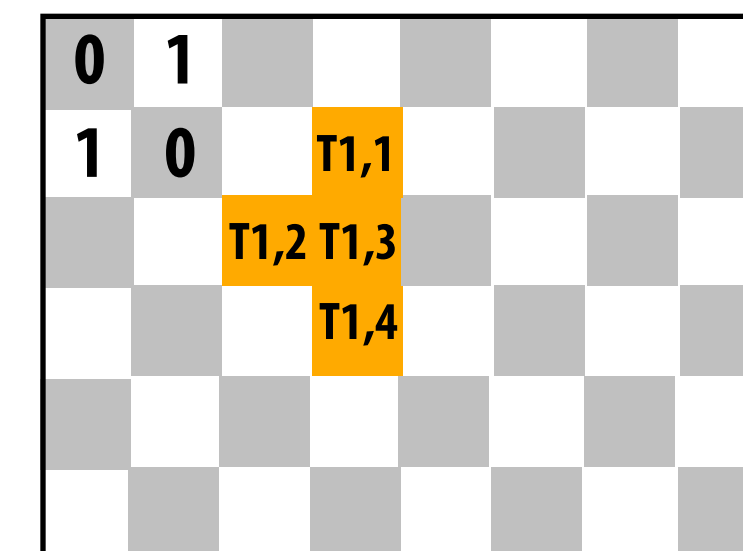
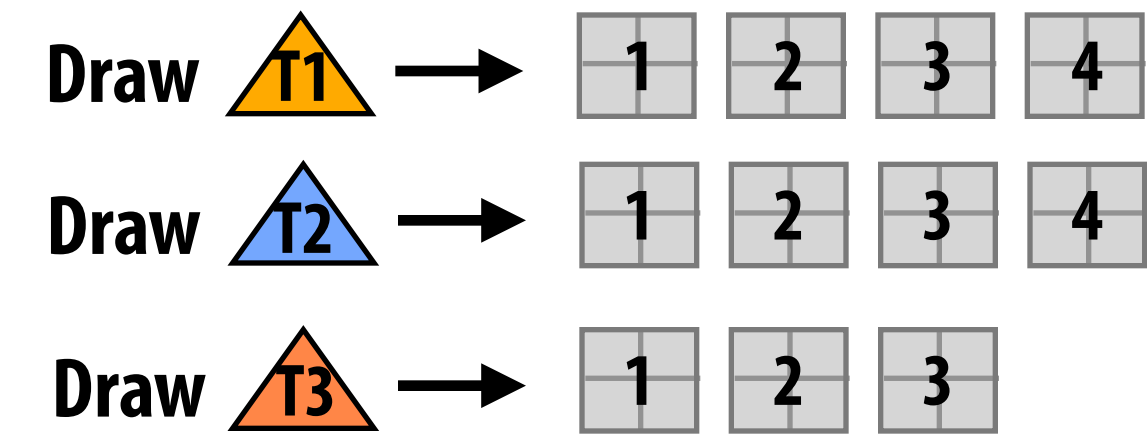


Interleaved
render target

Fragments from T3 cannot be processed yet. Why?



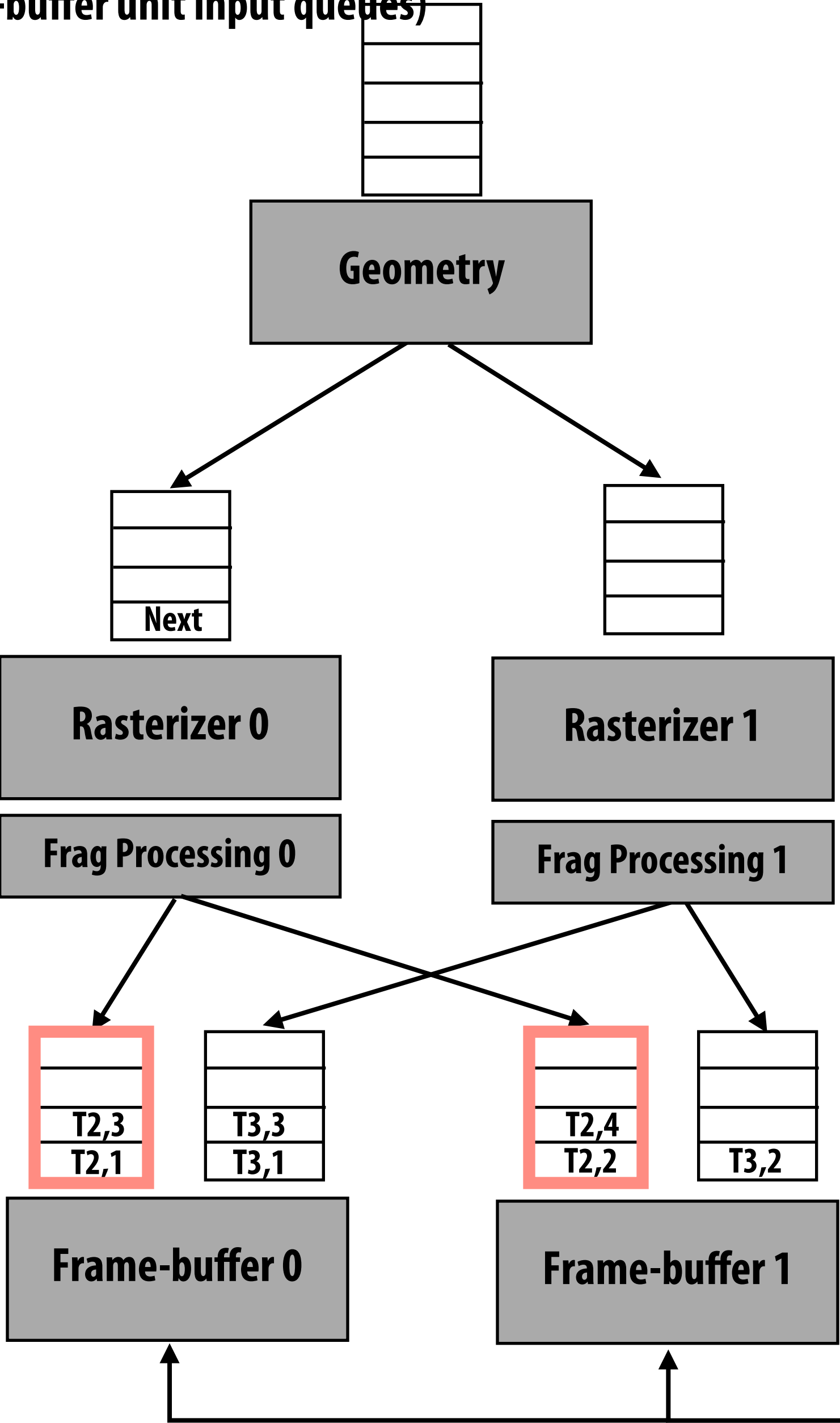
Input:



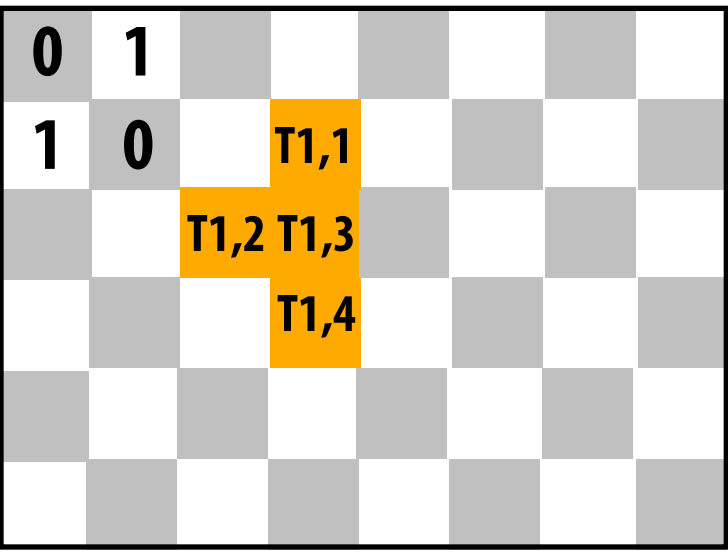
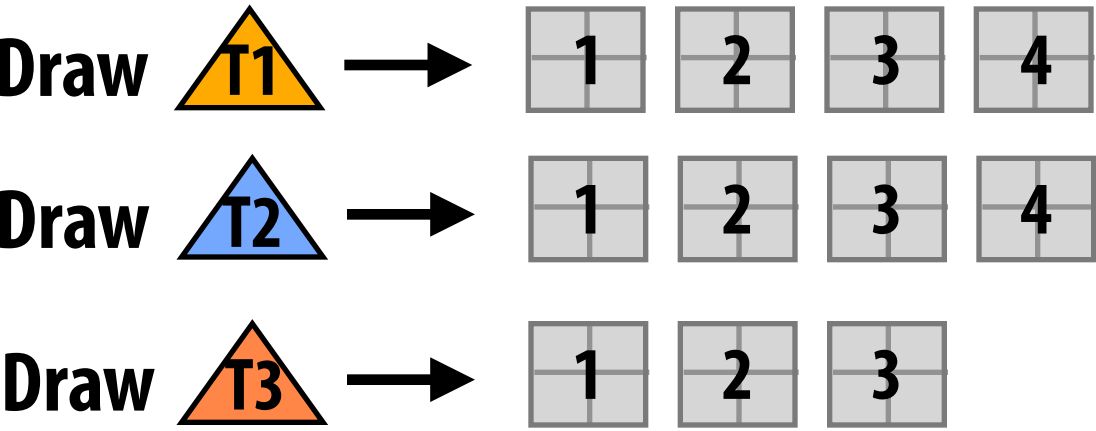
Interleaved
render target

Rast 0 processes T2

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

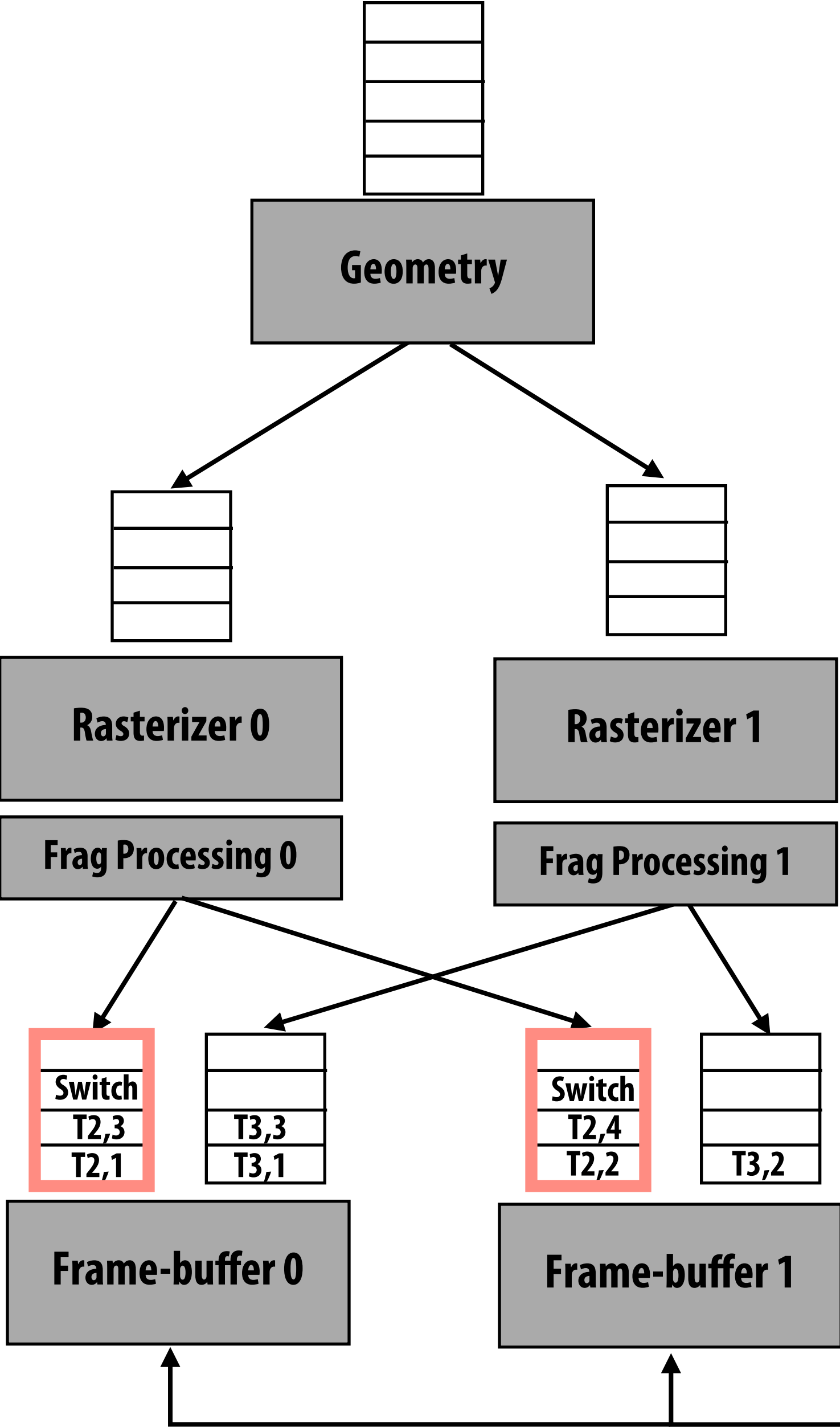


Input:

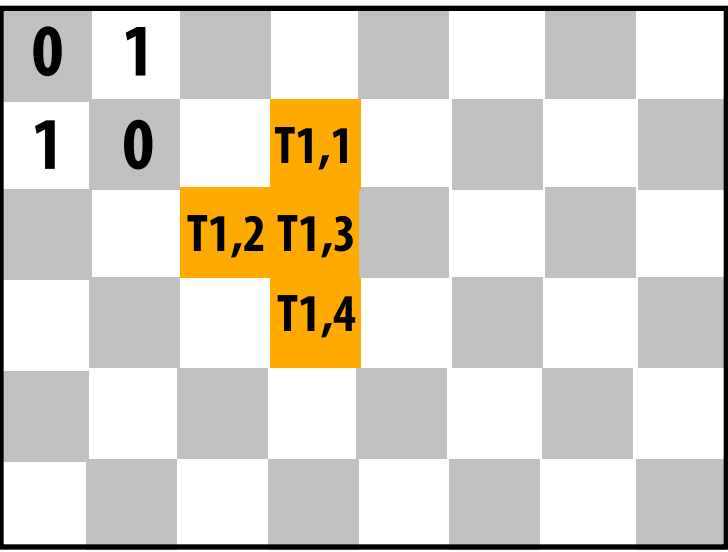
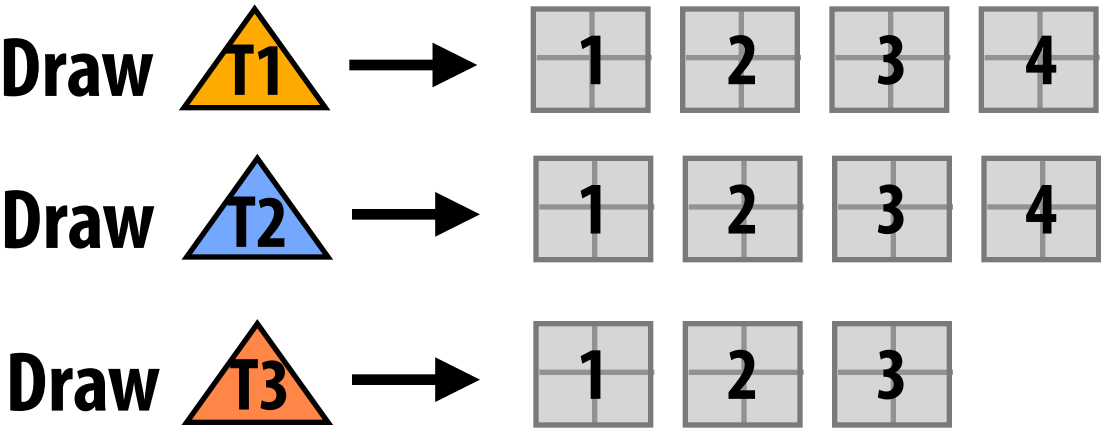


Interleaved
render target

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

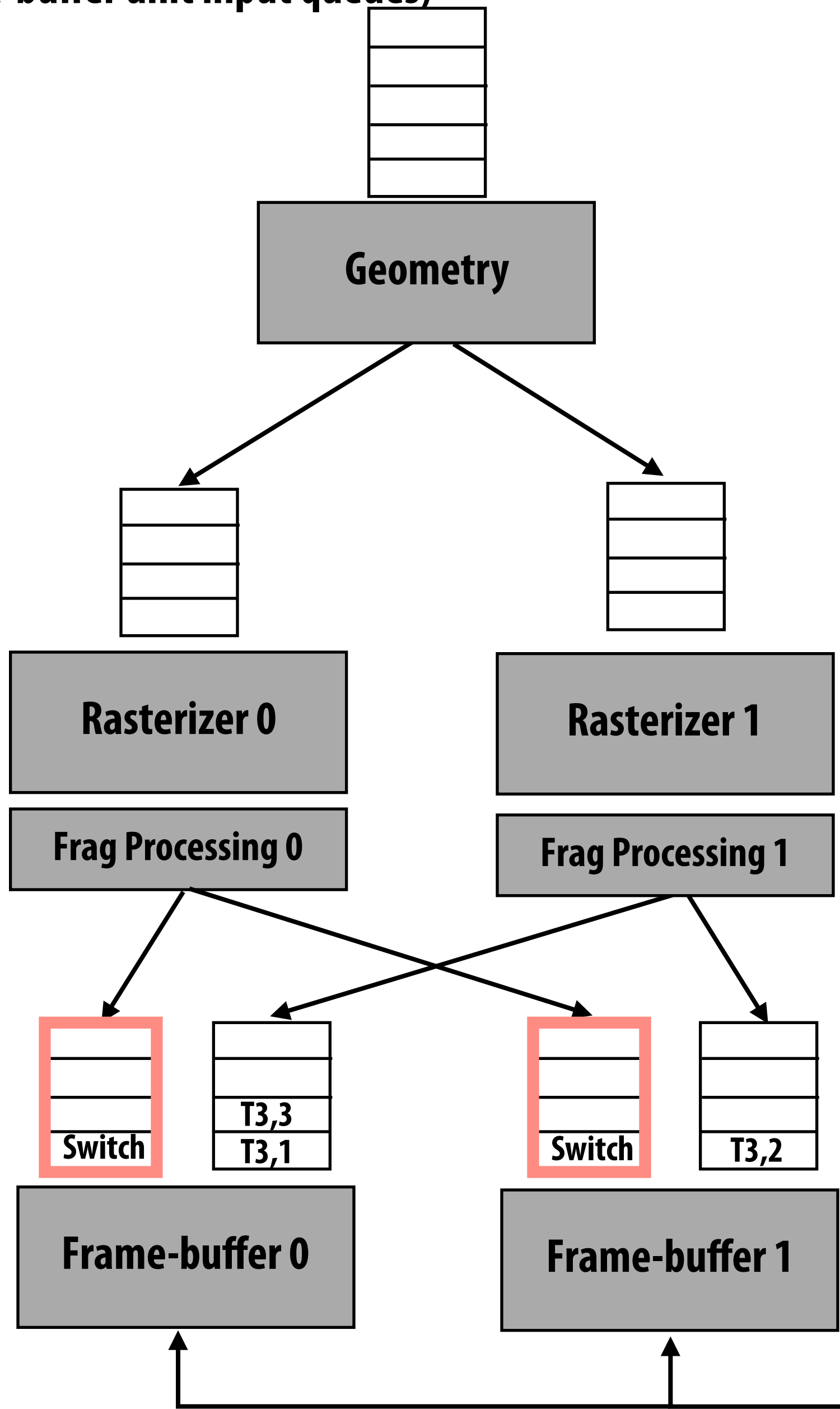


Input:

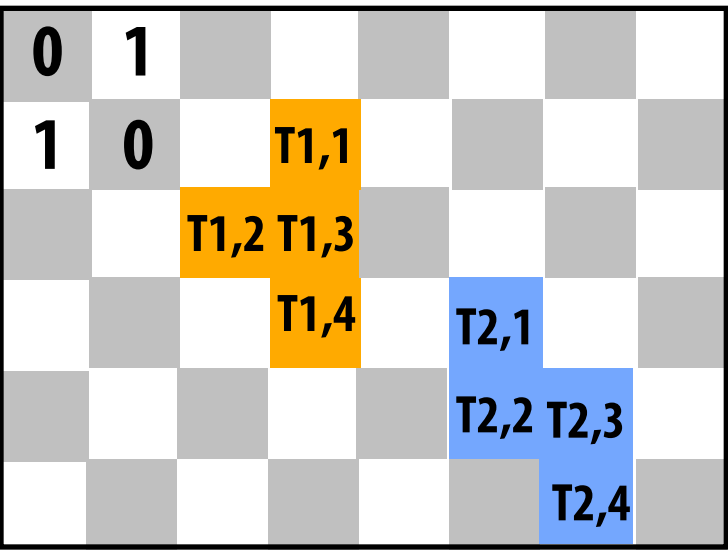
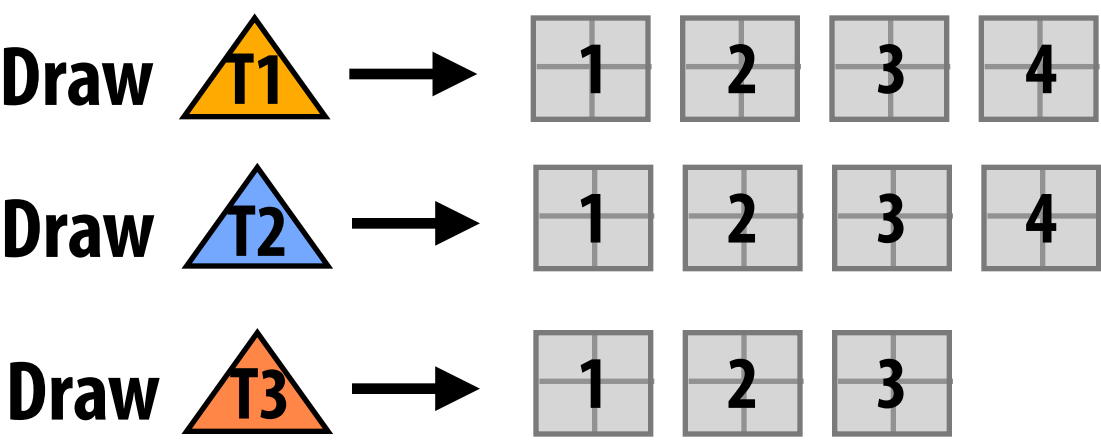


Rast 0 processes T2

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

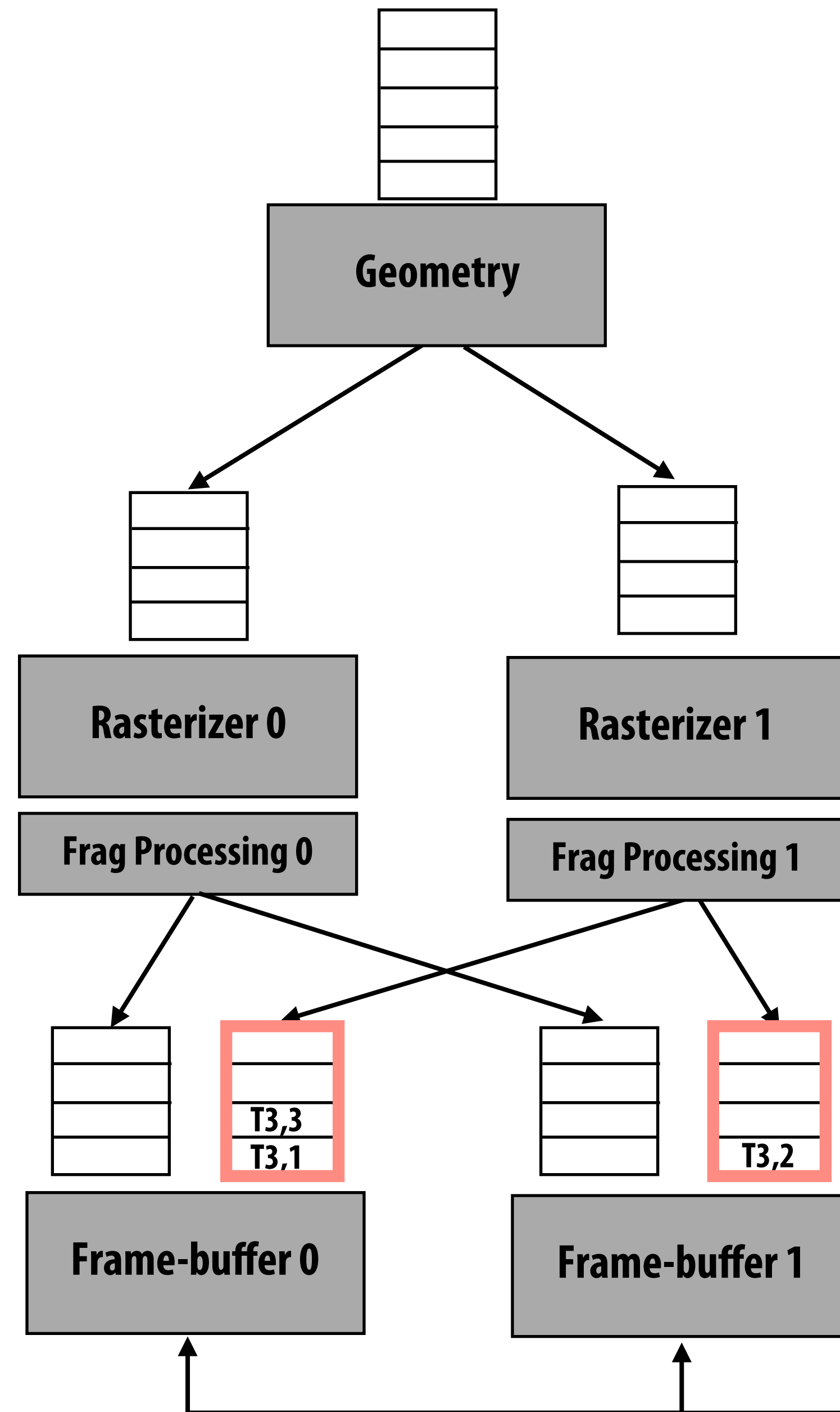


Input:

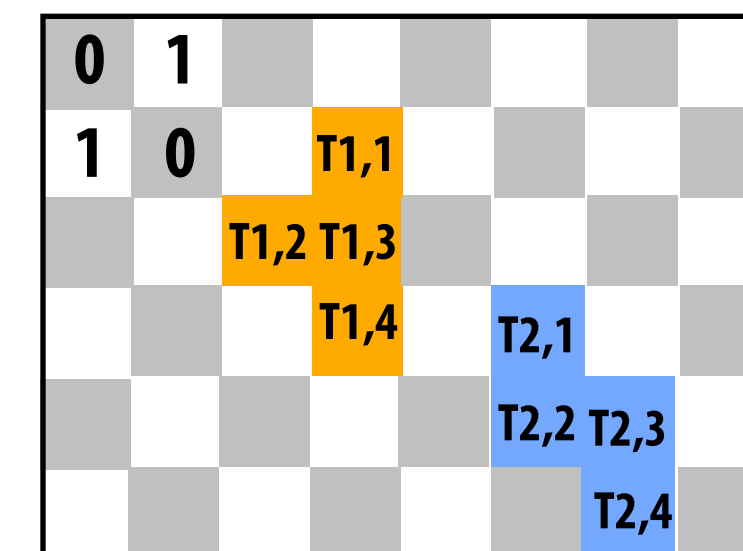
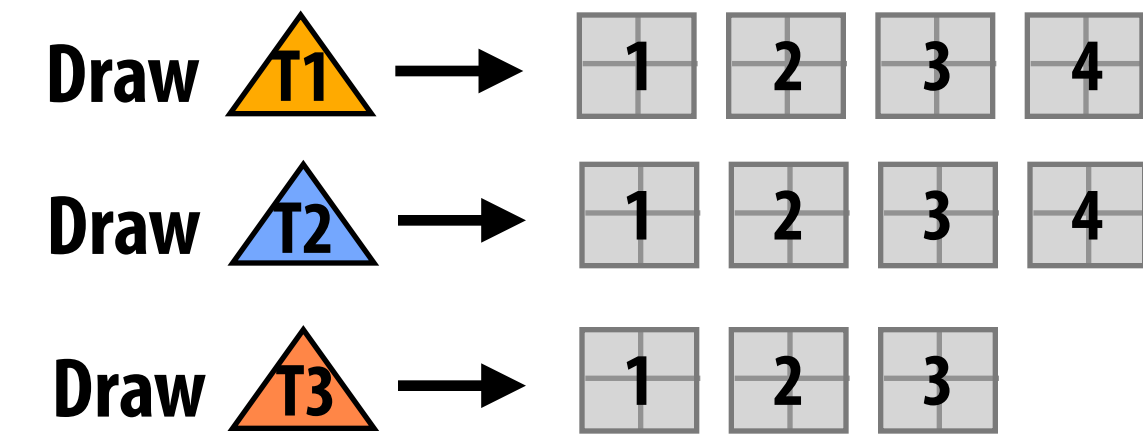


Interleaved
render target

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



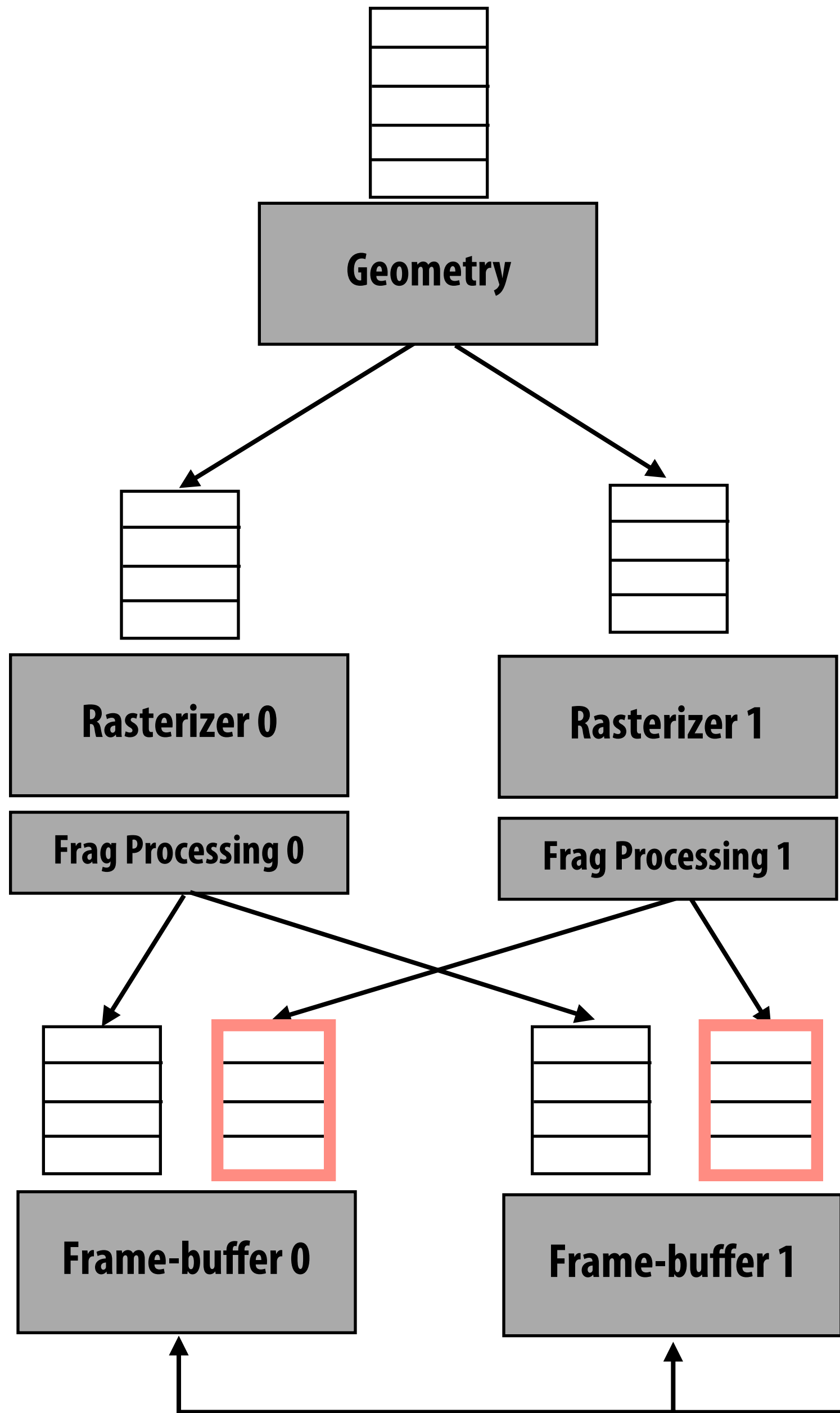
Input:



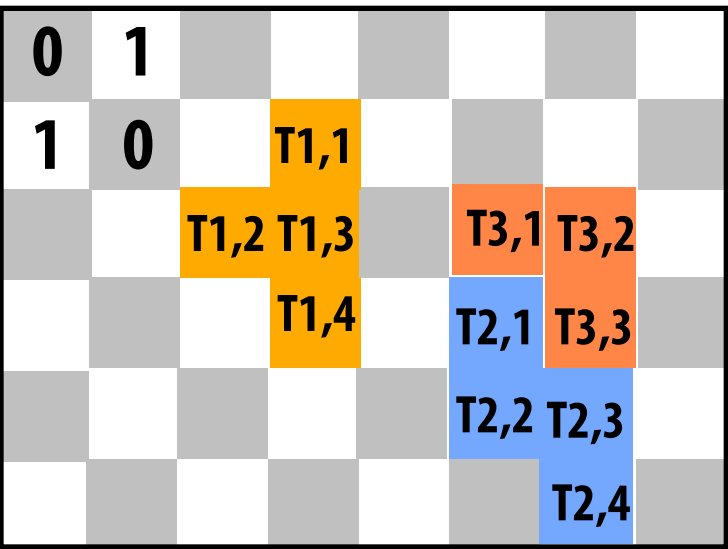
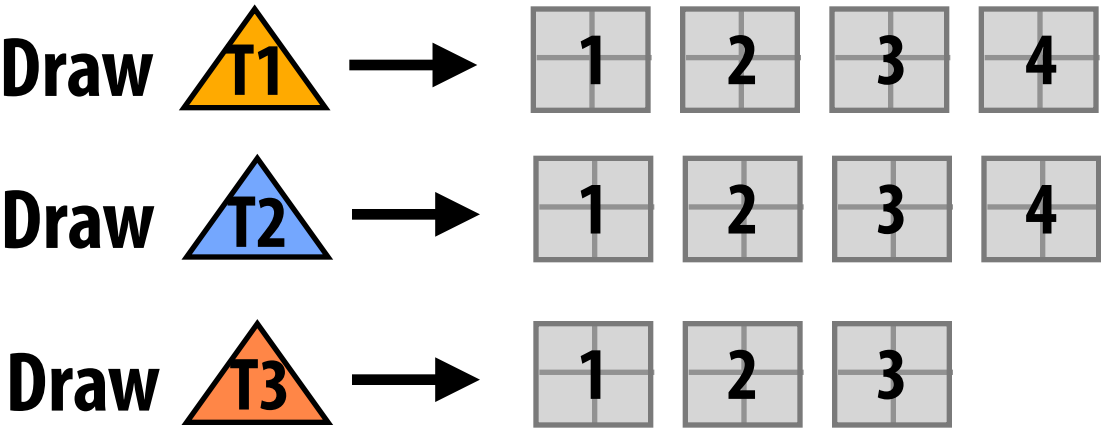
Interleaved
render target

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

(Notice updates to frame buffer)



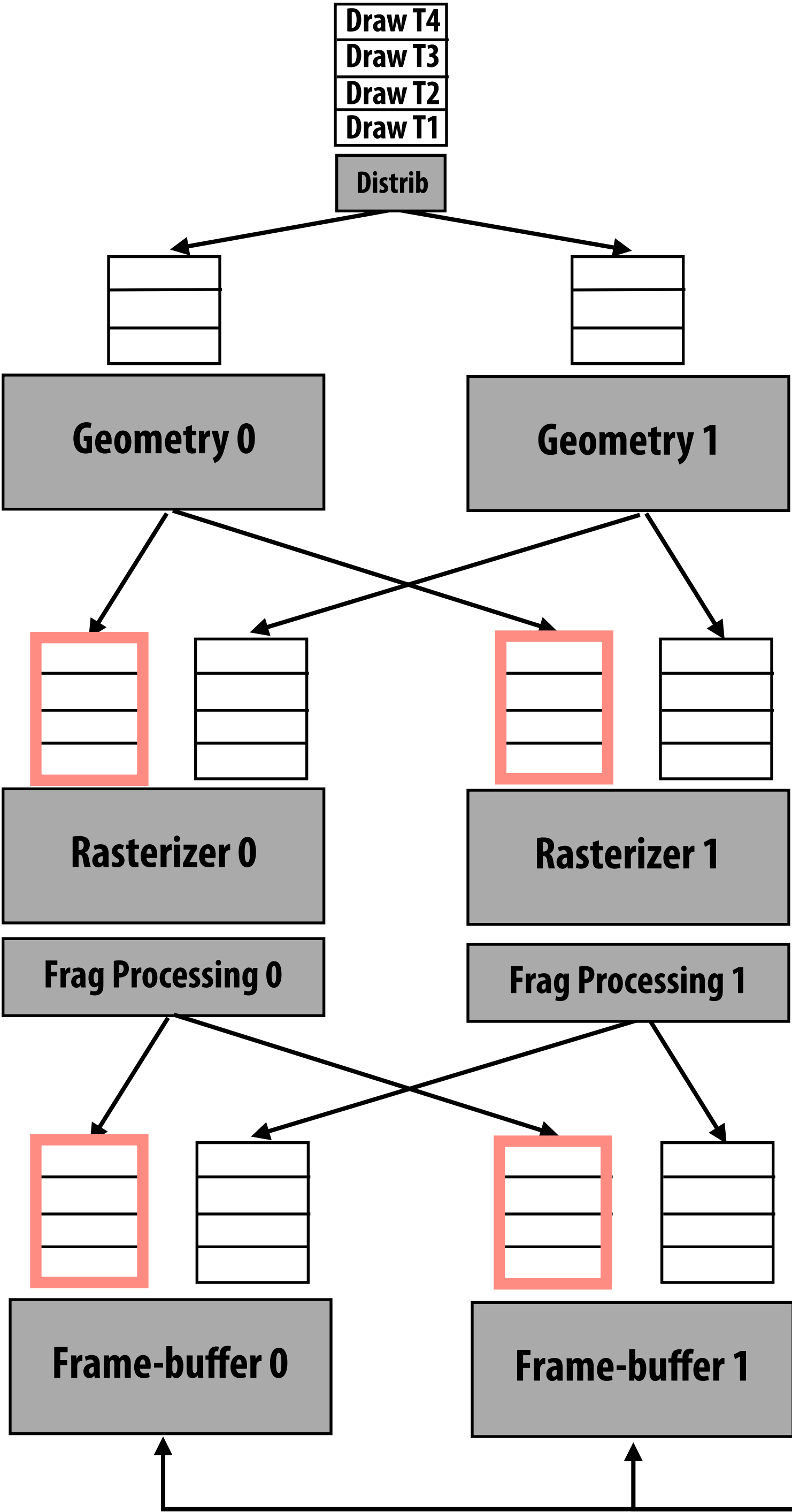
Input:



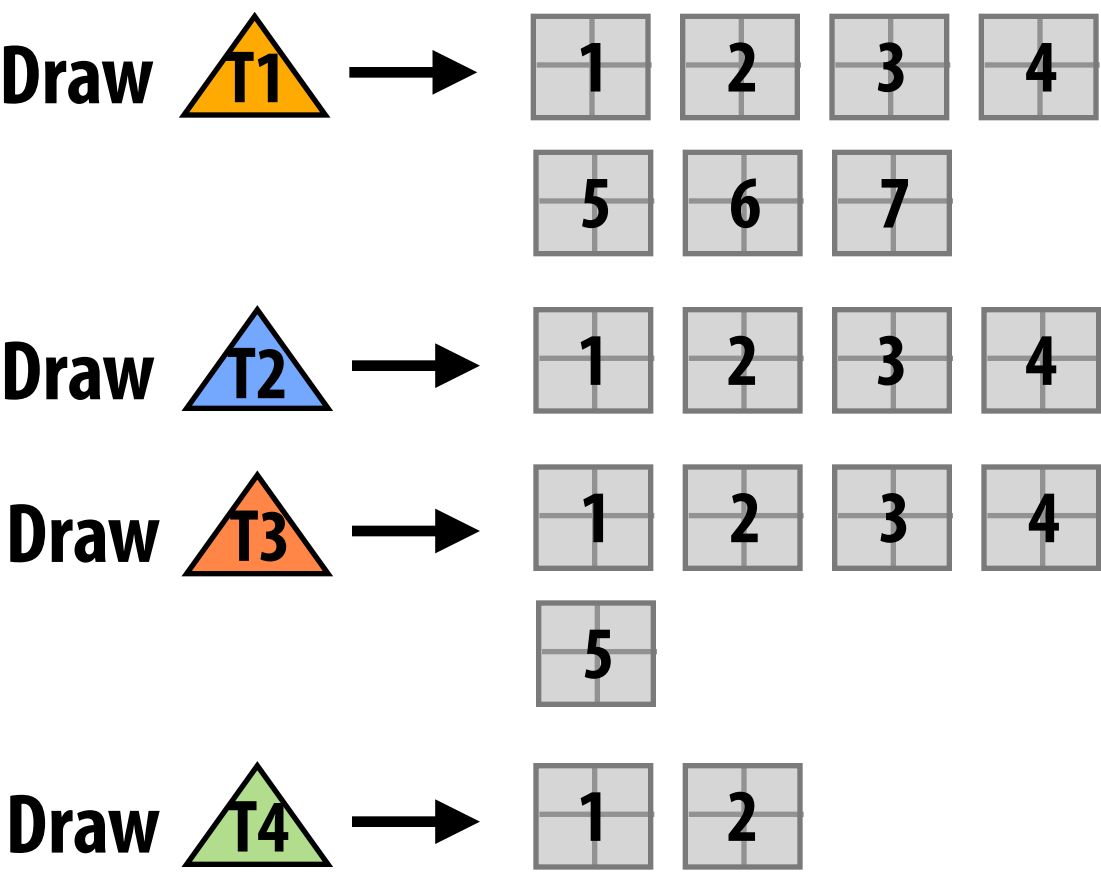
Interleaved
render target

Extending to parallel geometry units

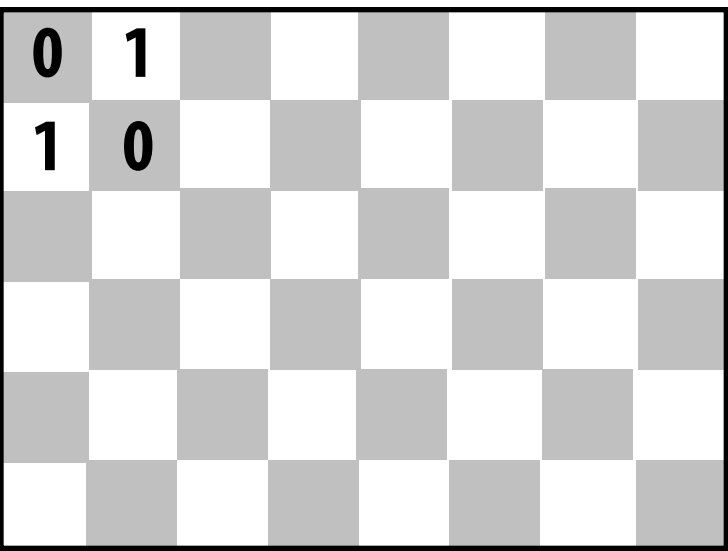
Starting state:
commands enqueued



Input:

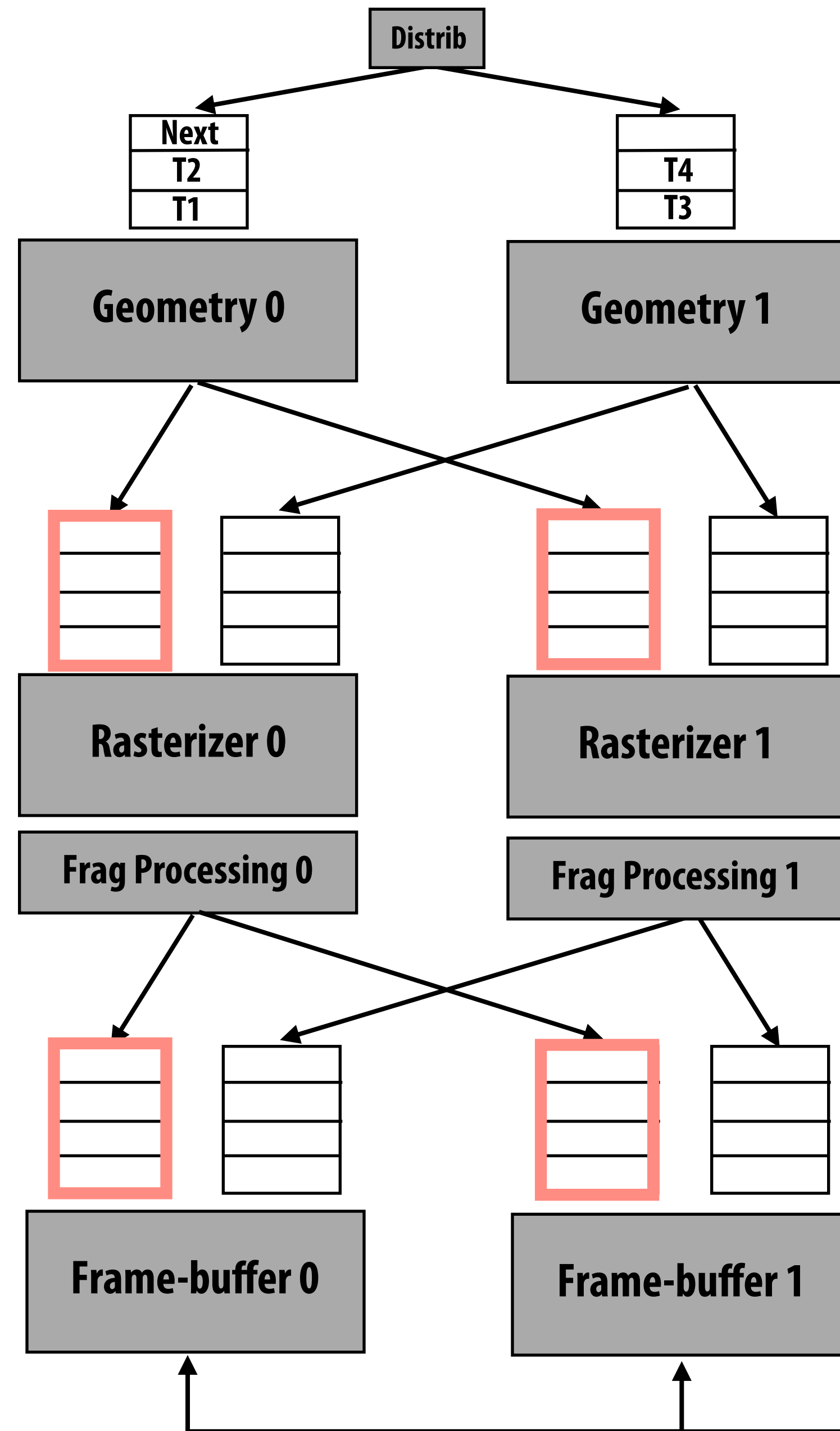


Assume batch size is 2 for
assignment to geom units
and to rasterizers.

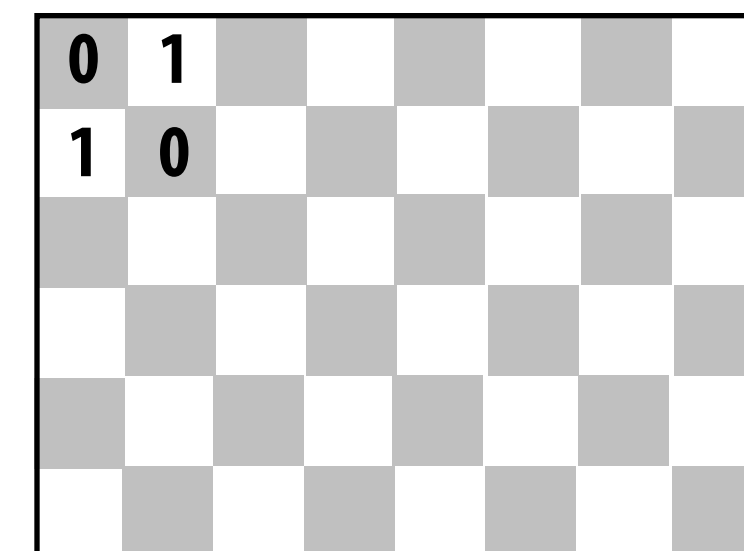
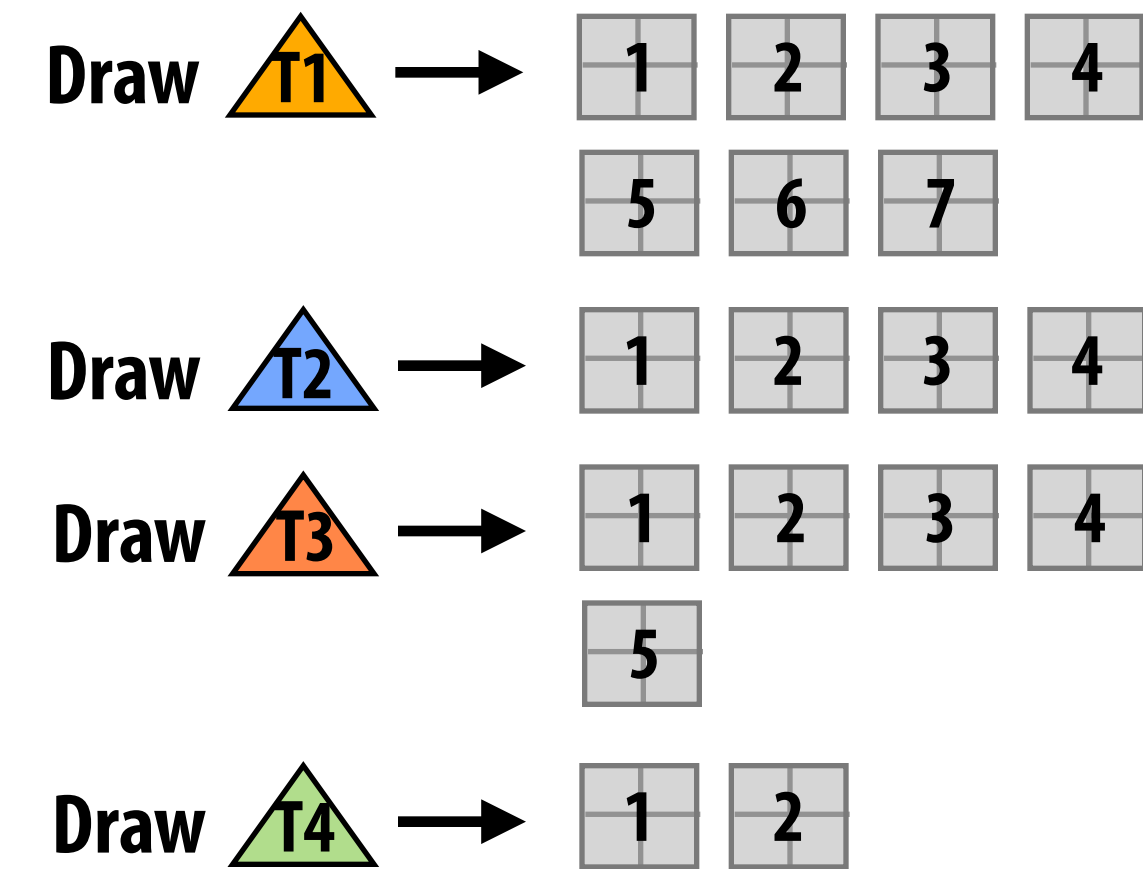


Interleaved
render target

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



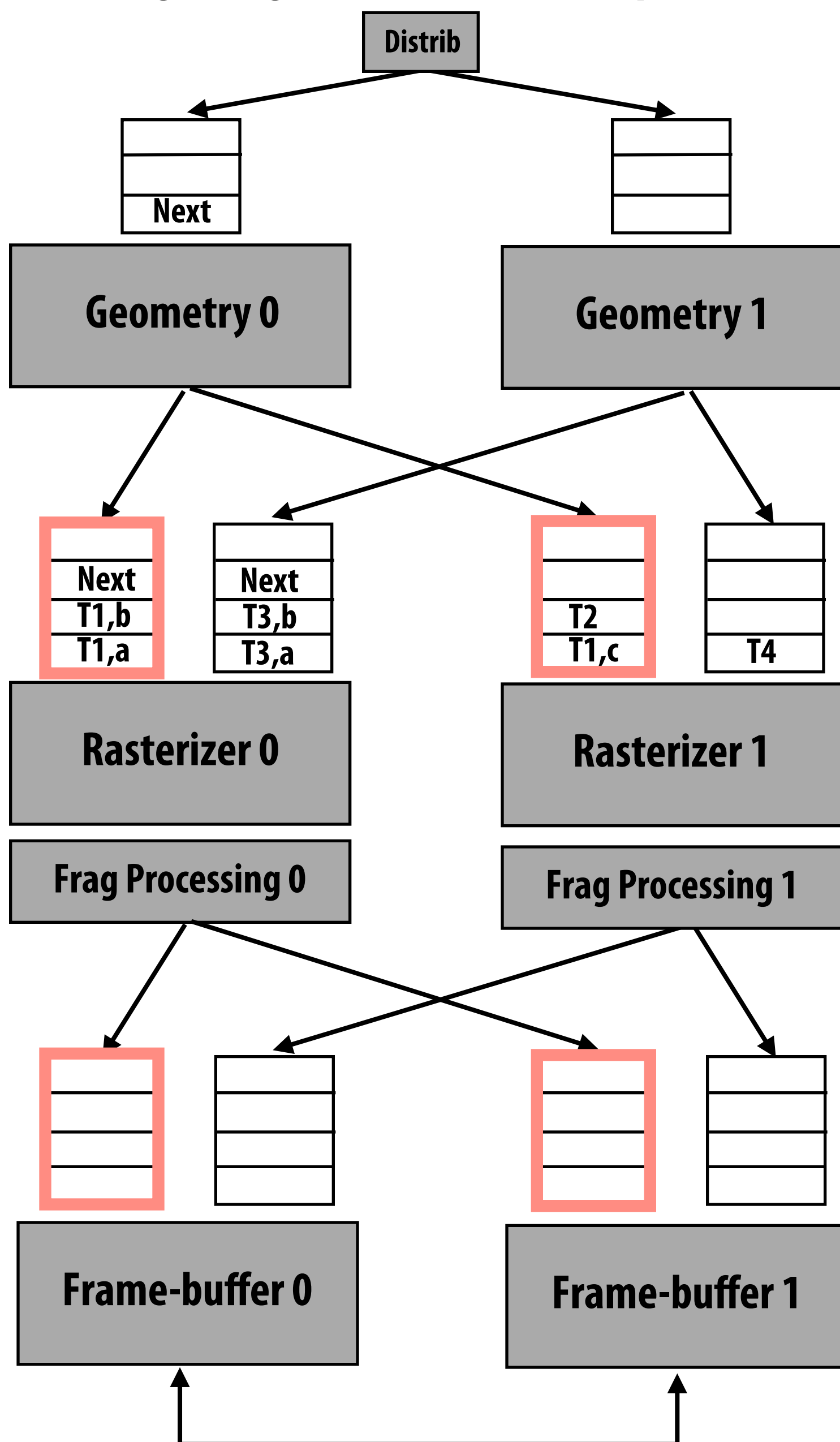
Input:



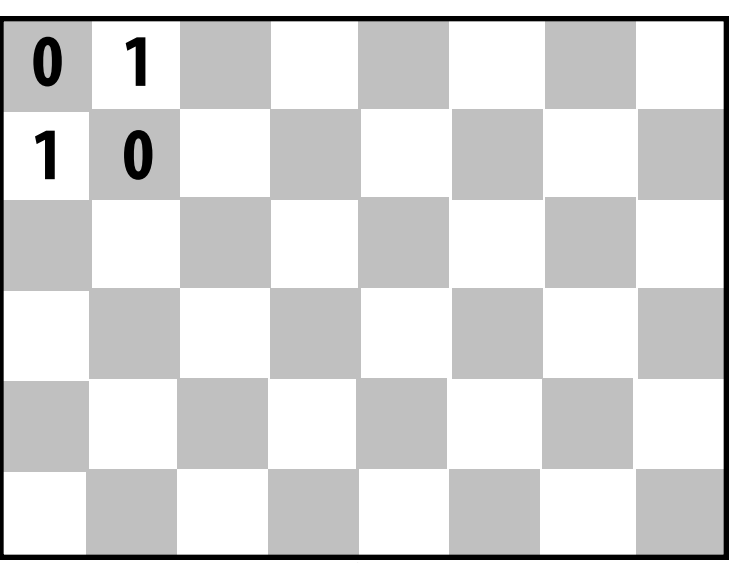
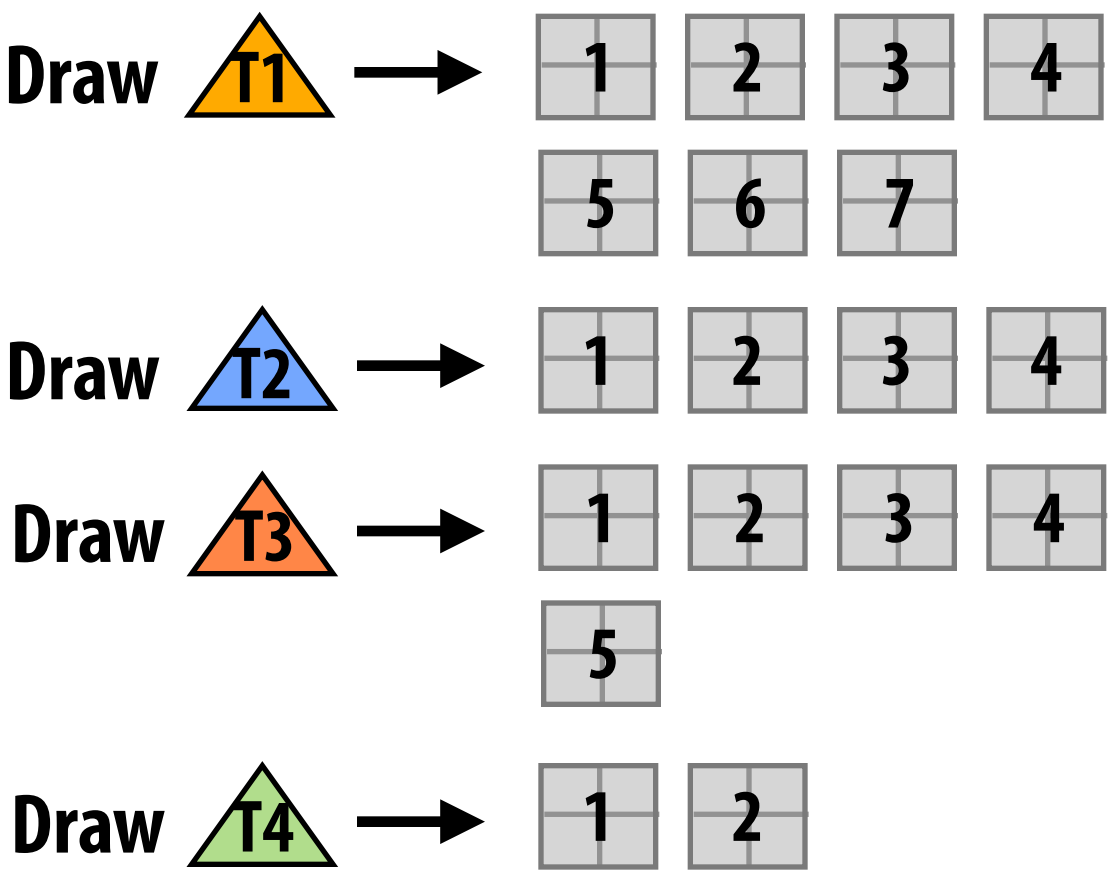
Interleaved
render target

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.])

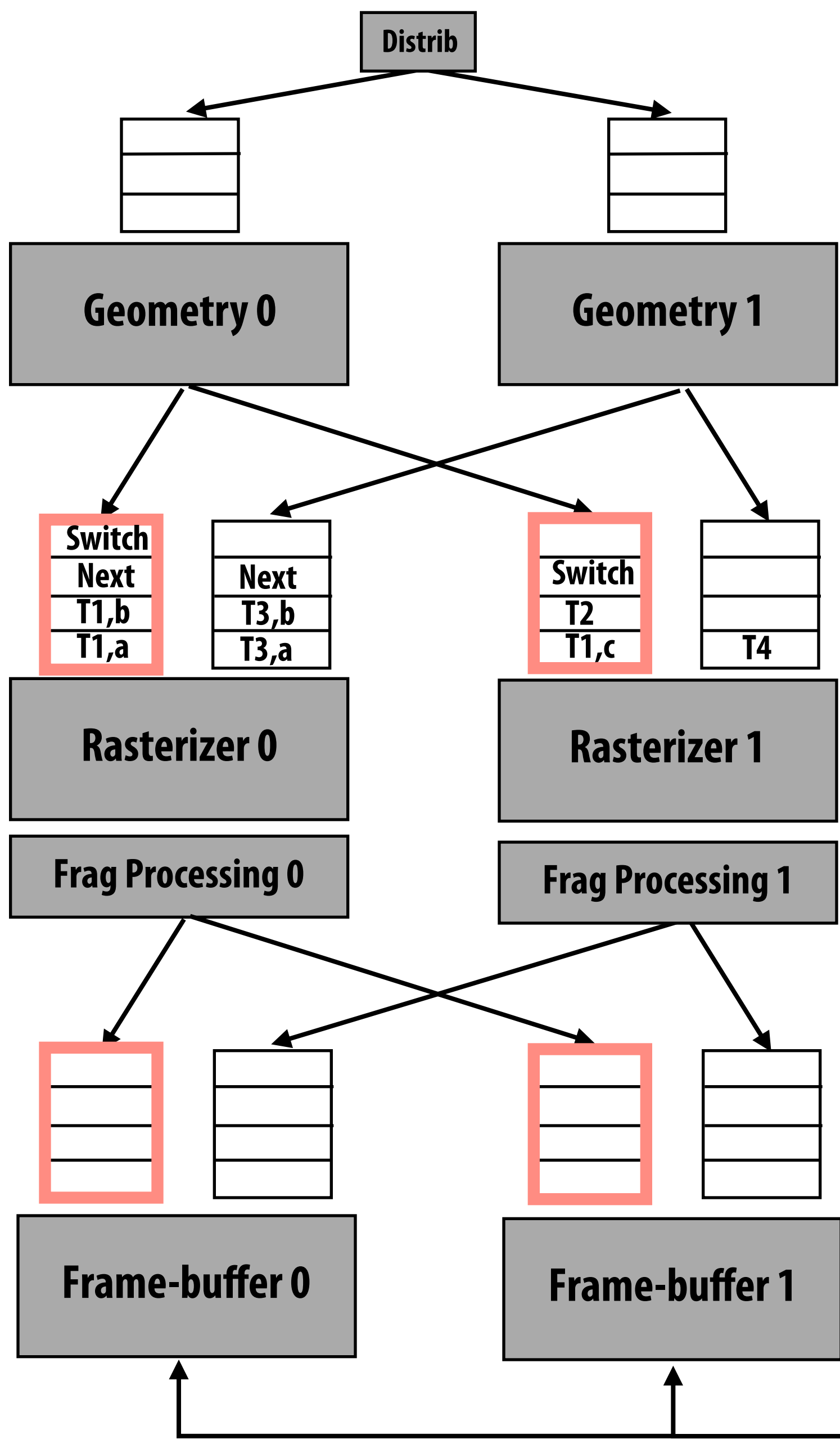


Input:

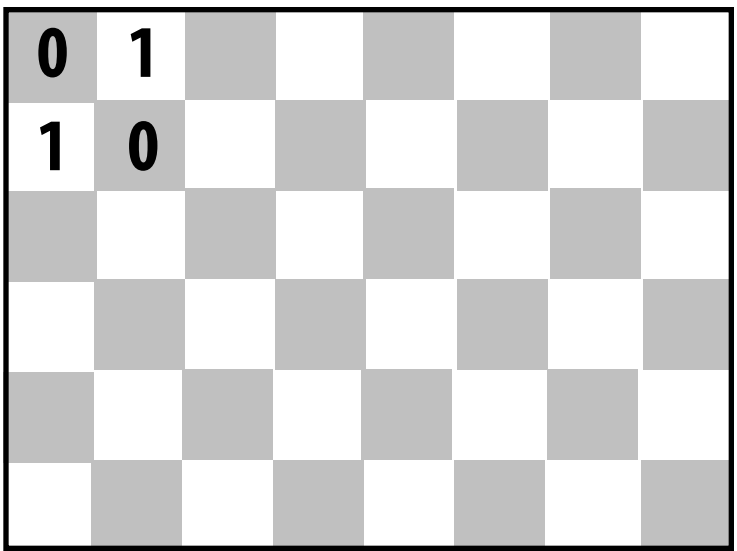
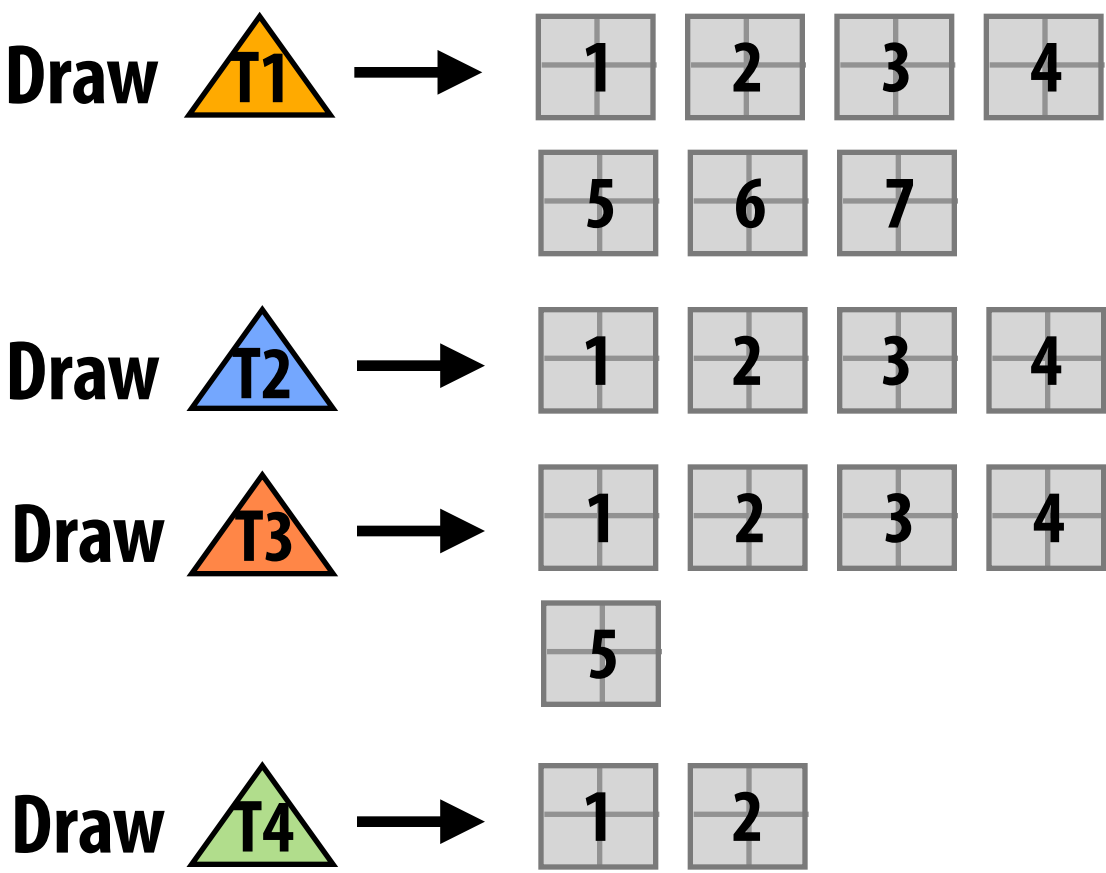


Interleaved
render target

Geom 0 broadcasts 'switch' token to rasterizers



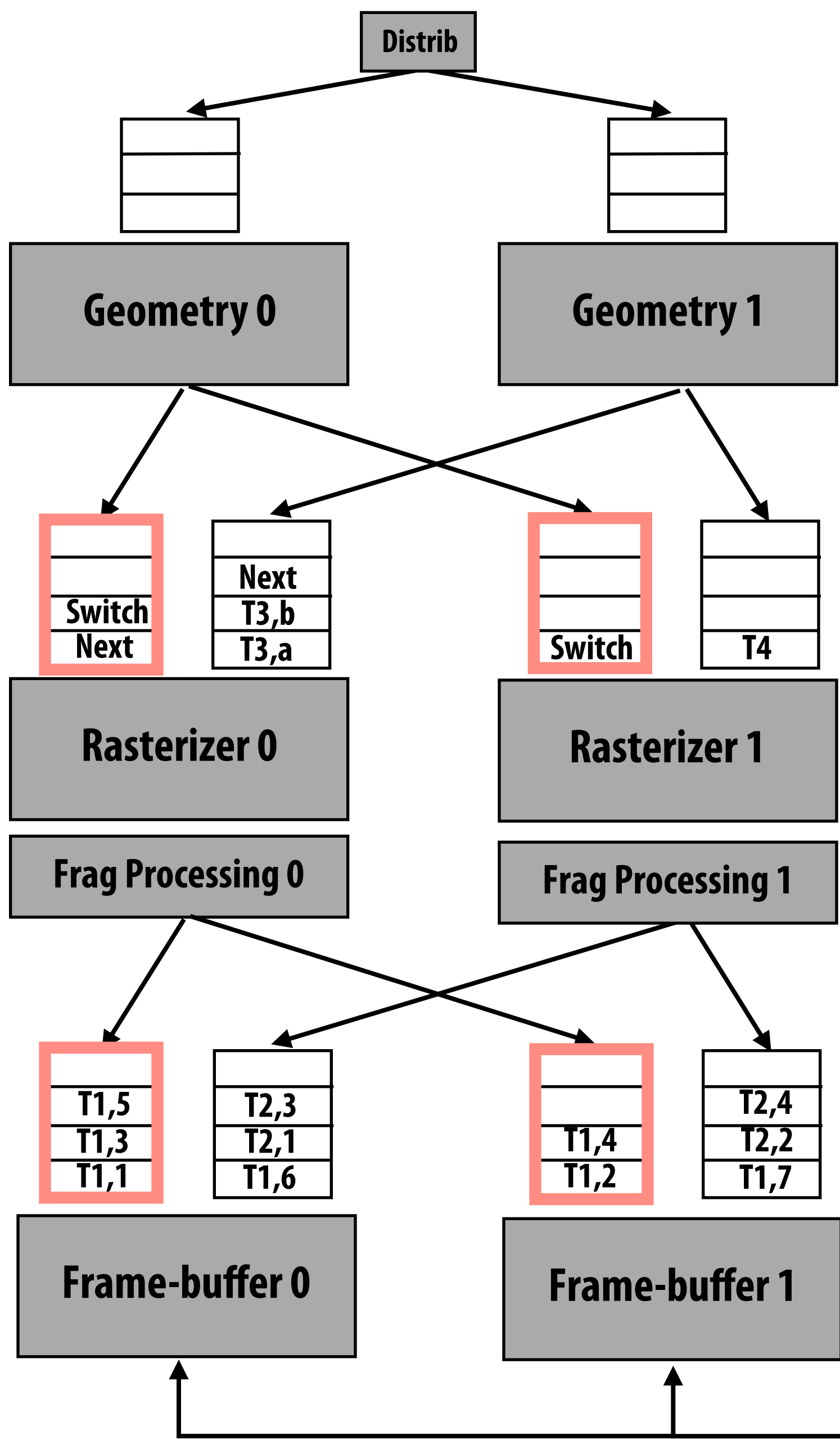
Input:



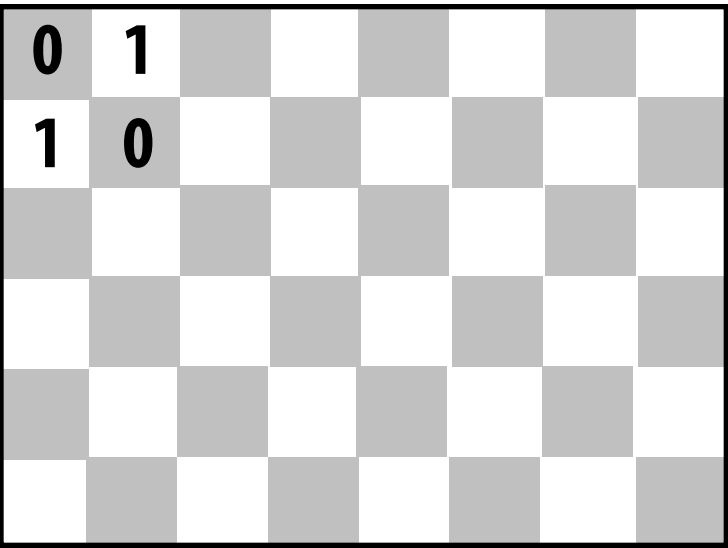
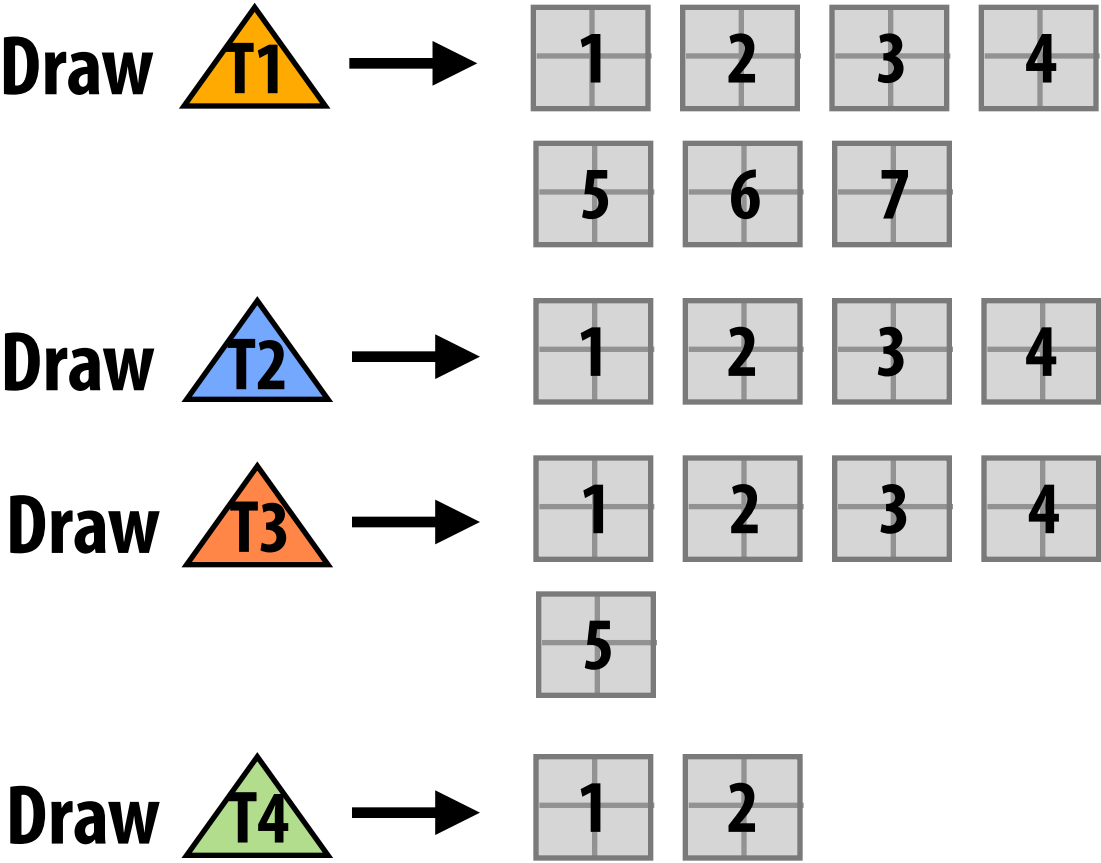
Interleaved
render target

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

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

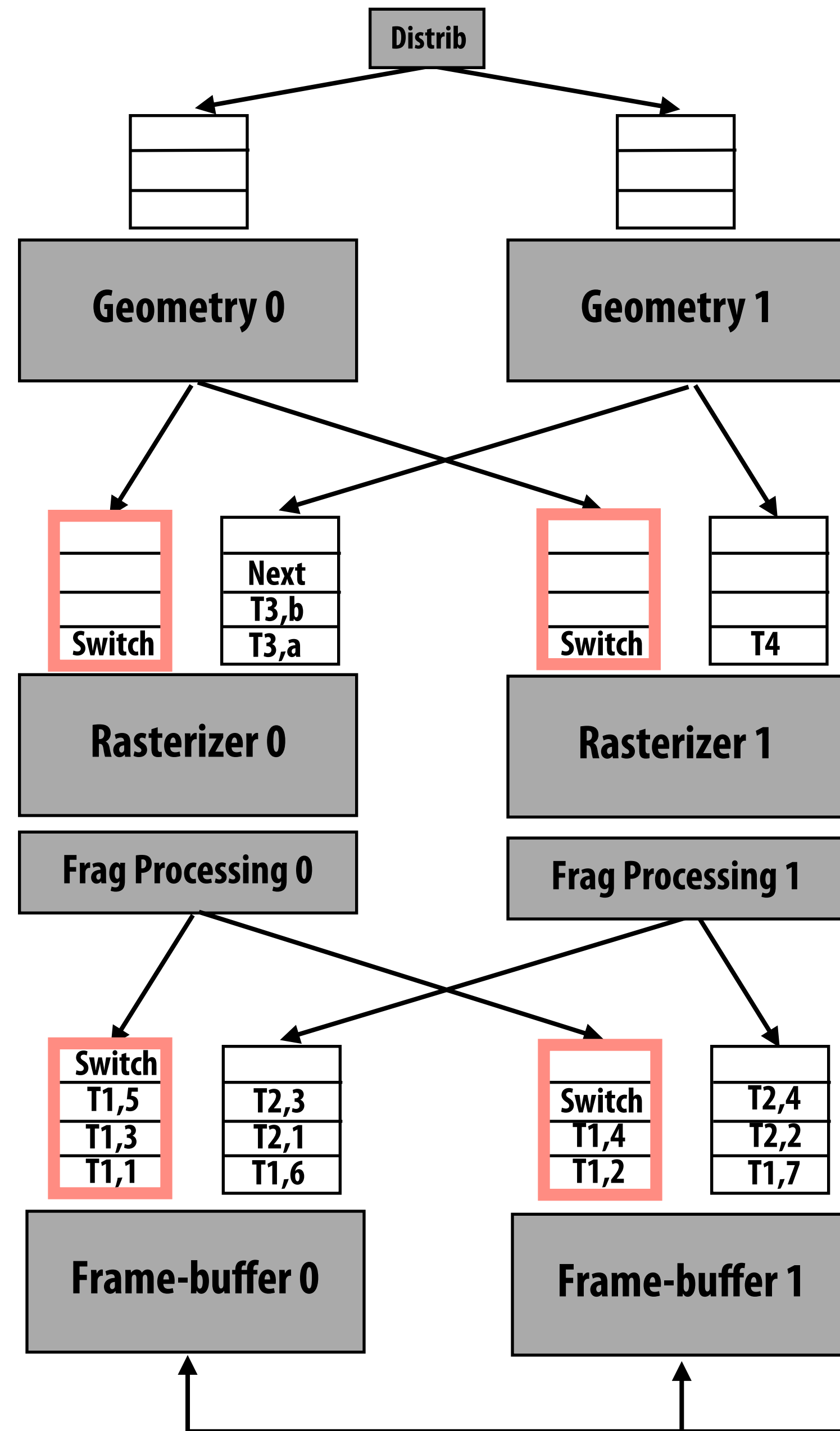


Input:

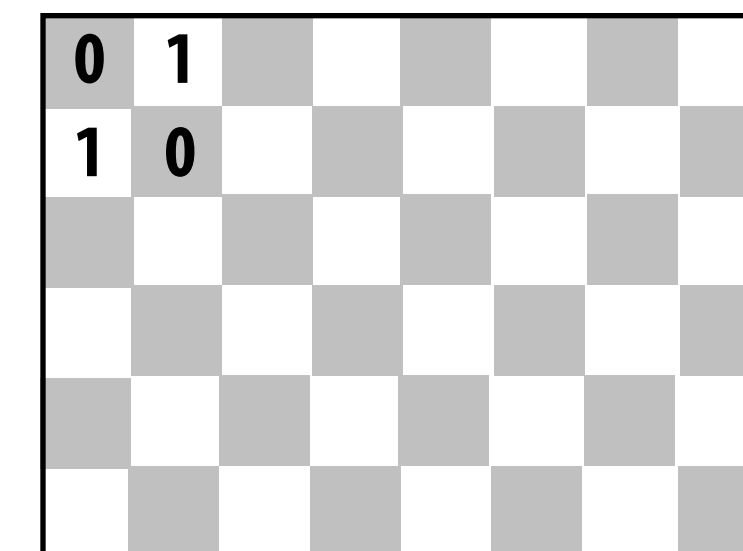
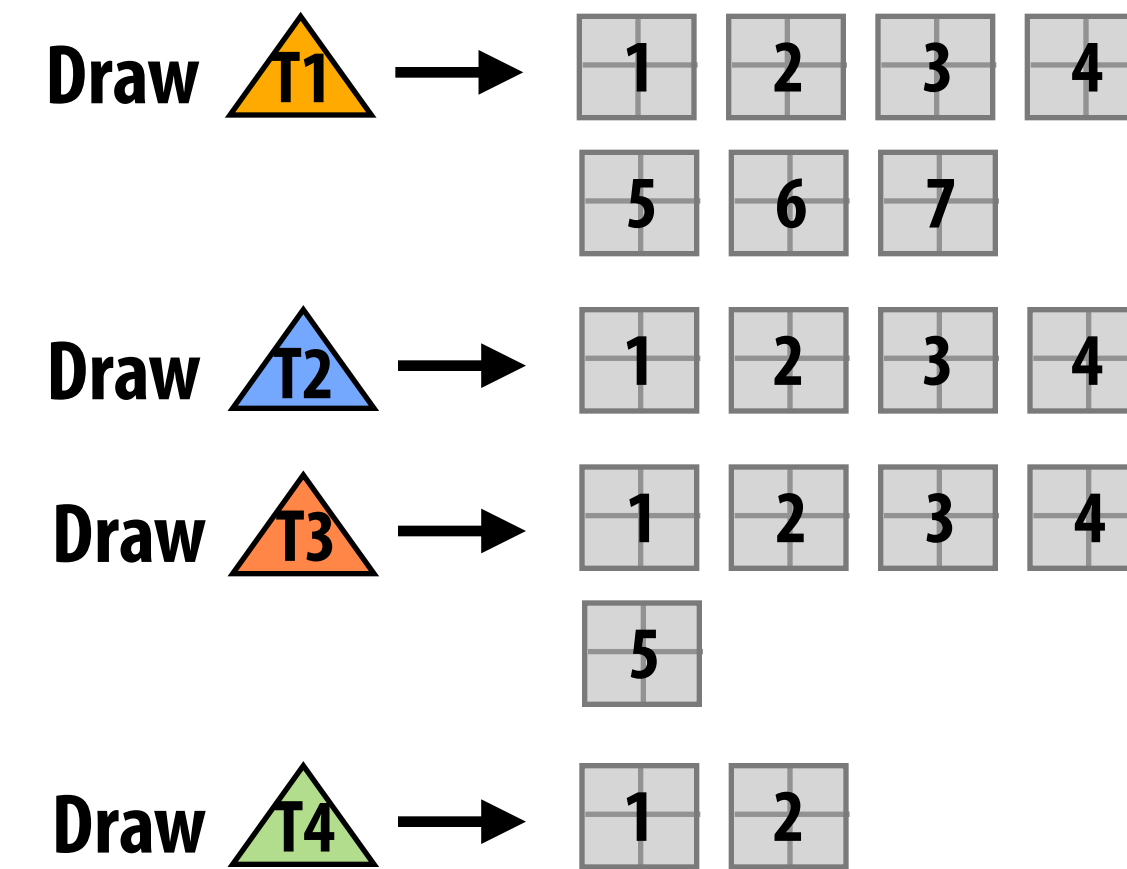


Interleaved
render target

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



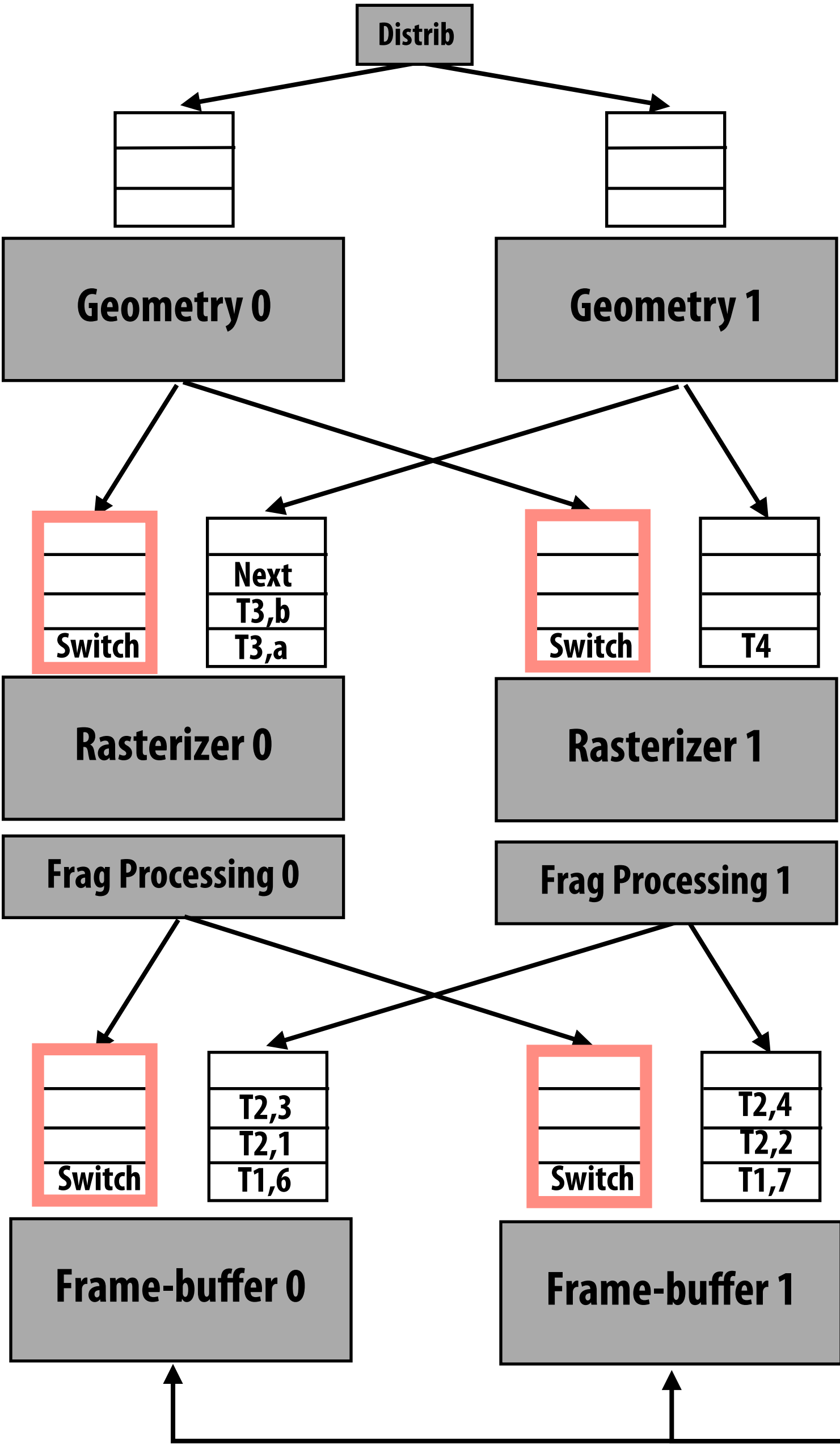
Input:



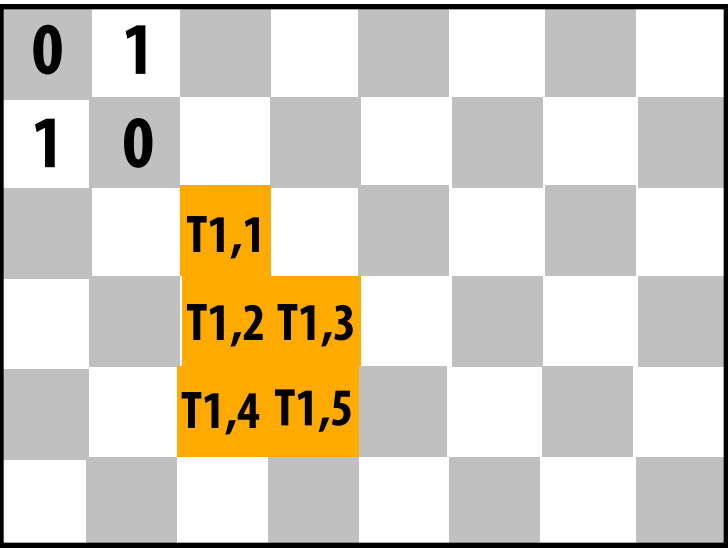
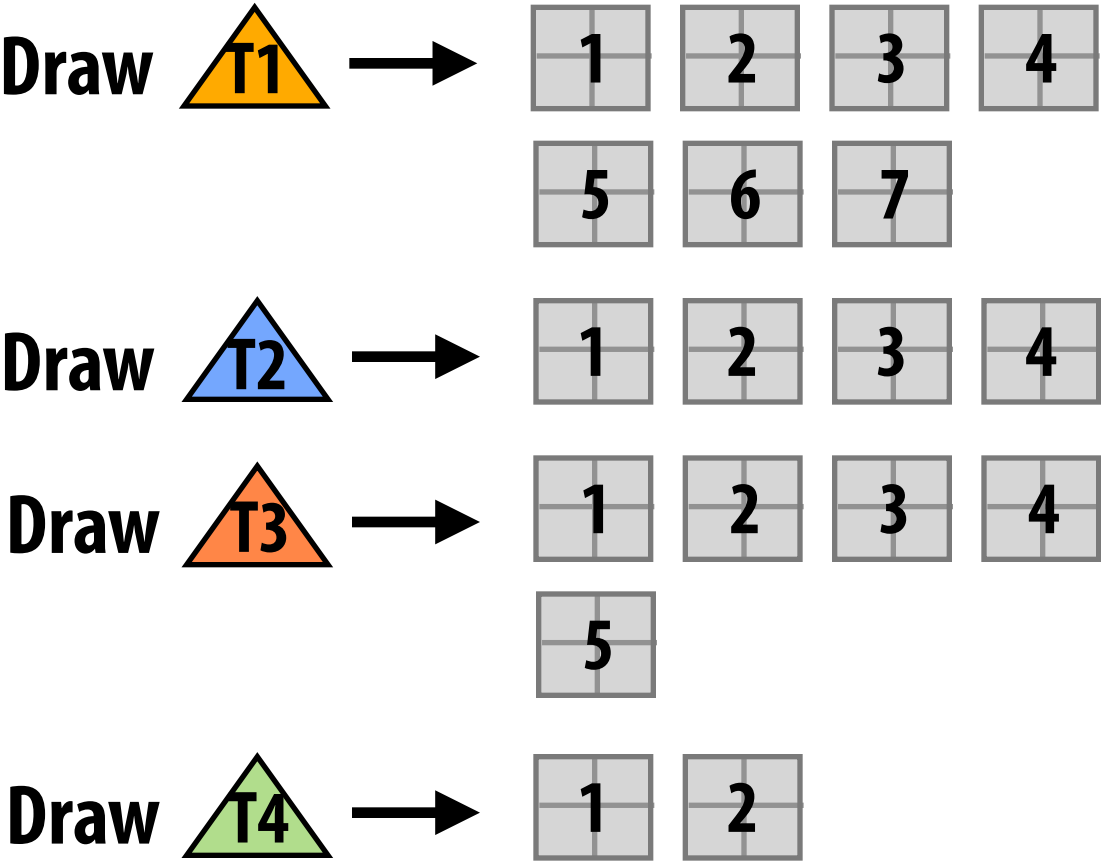
Interleaved
render target

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

(Notice updates to frame buffer)

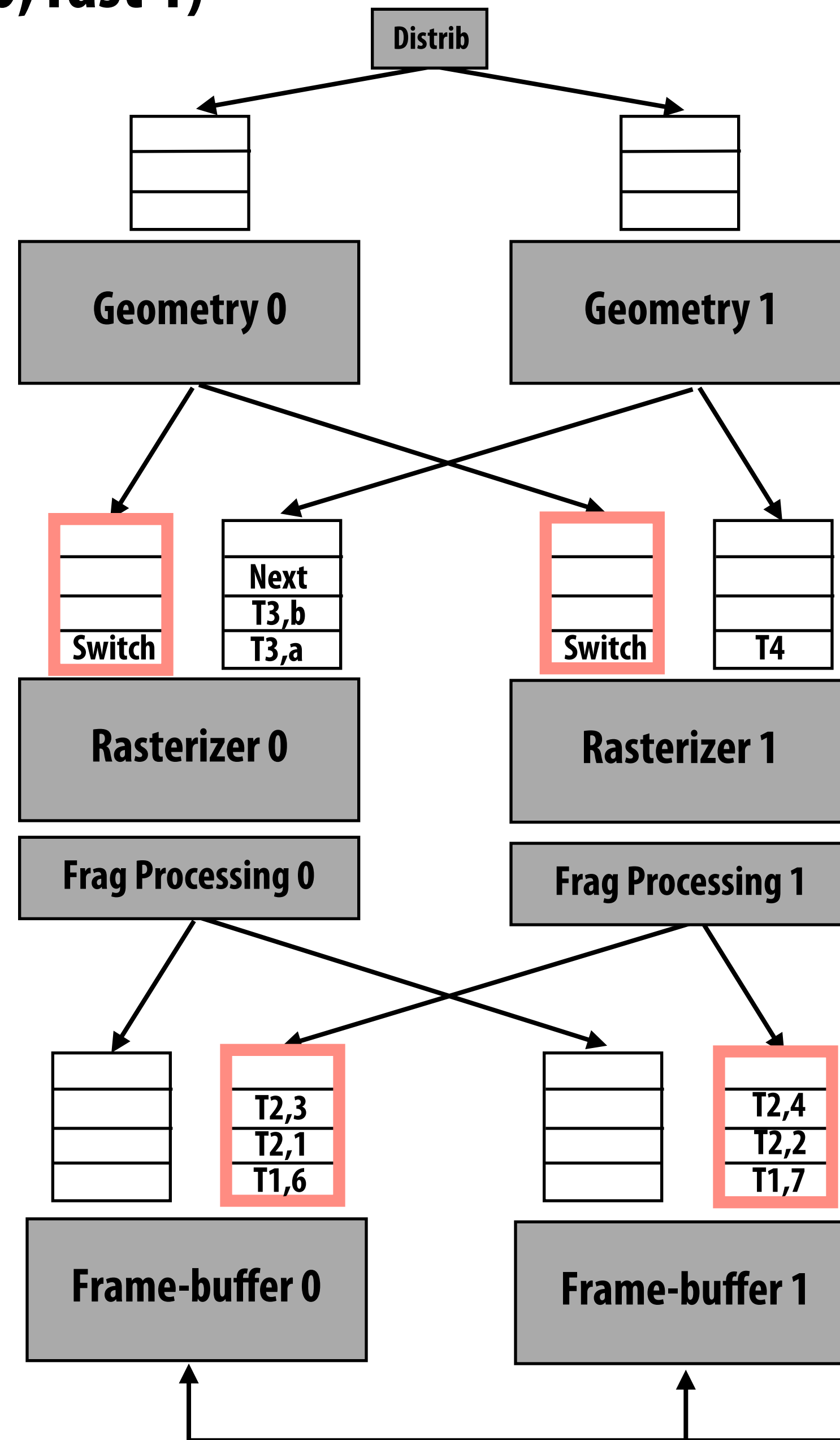


Input:

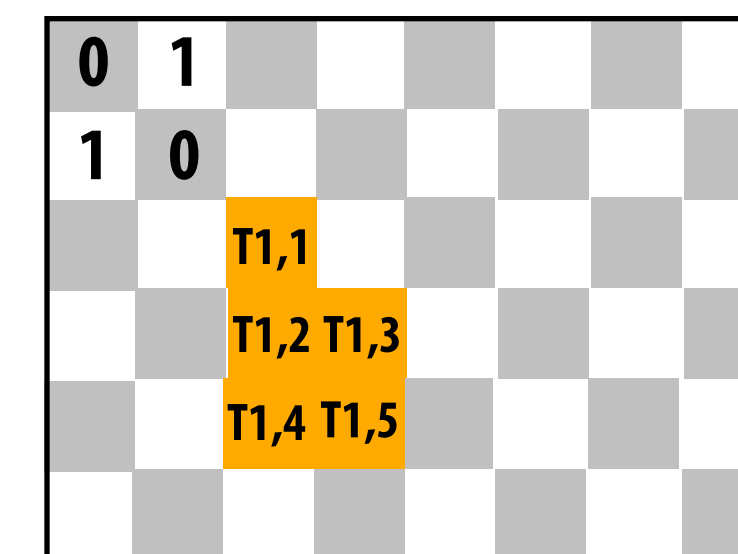
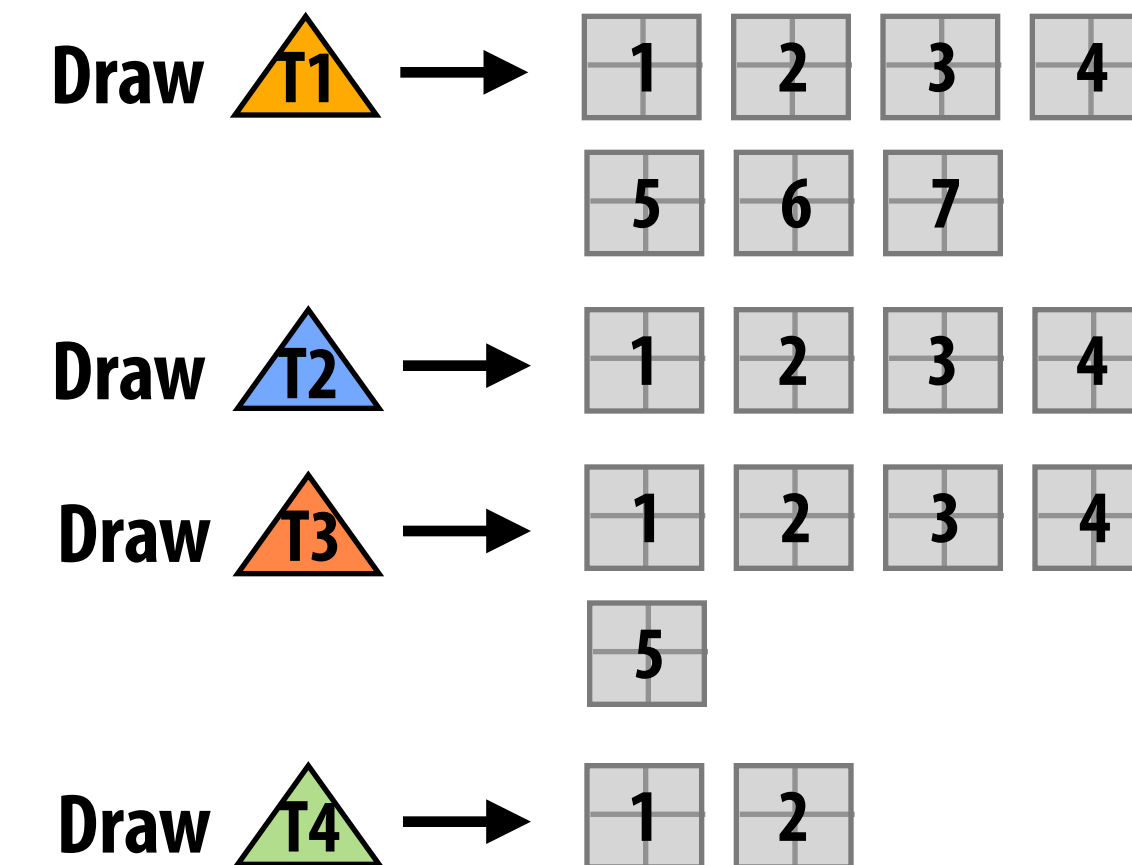


Interleaved
render target

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

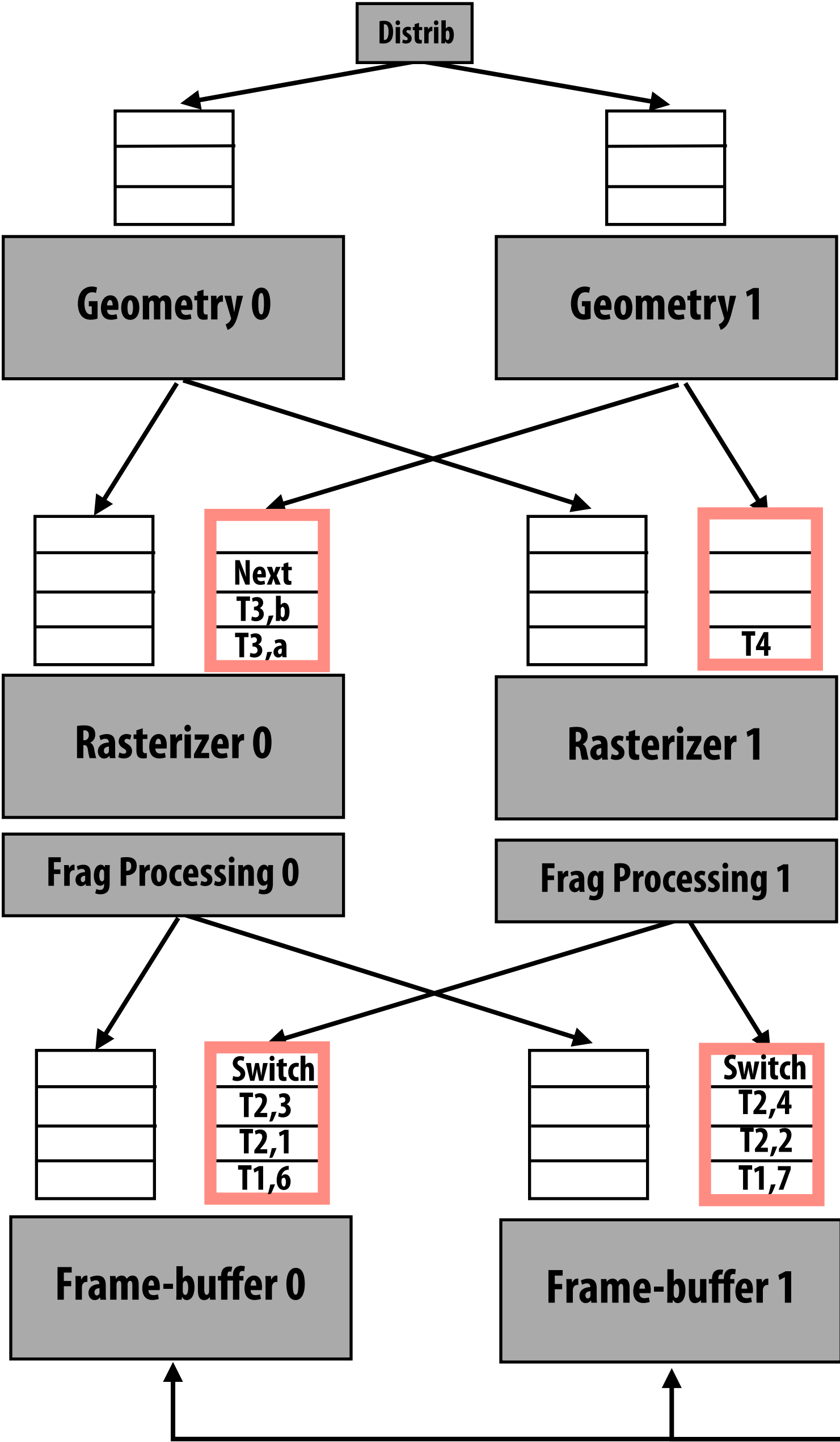


Input:

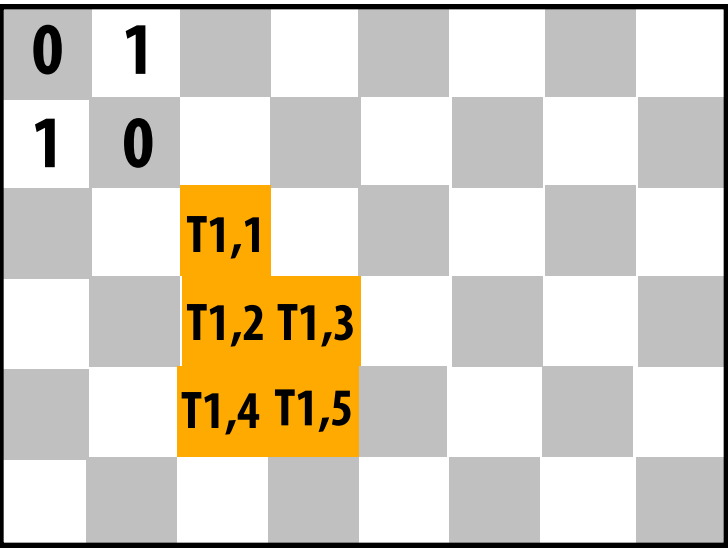
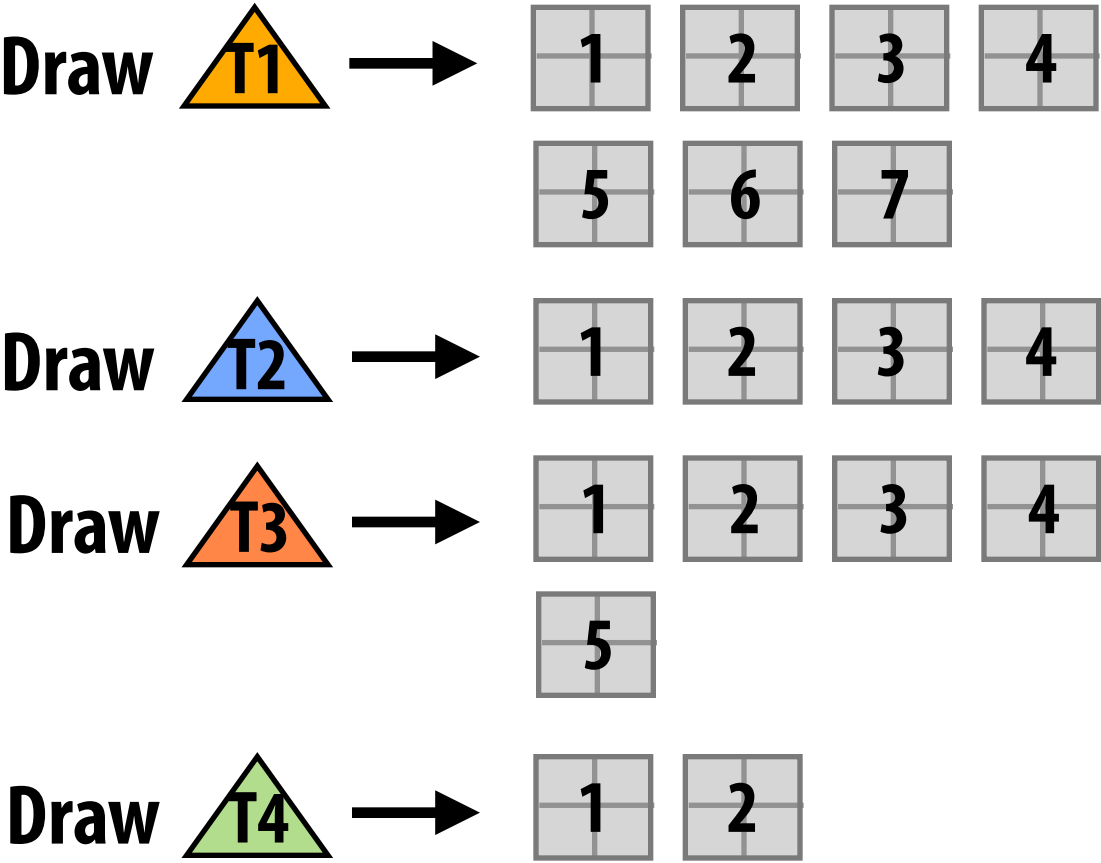


**Interleaved
render target**

“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)



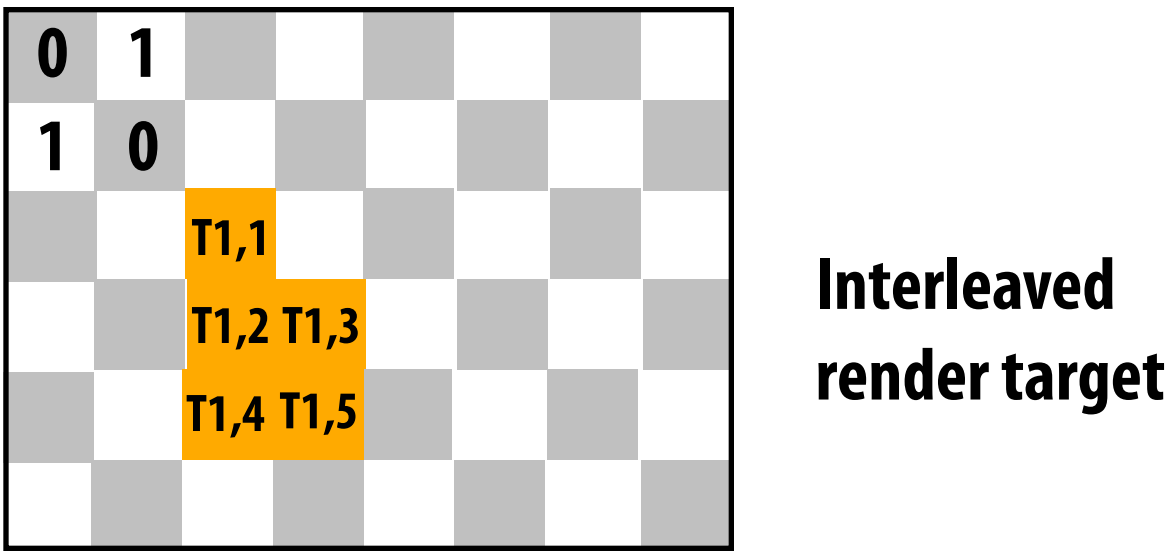
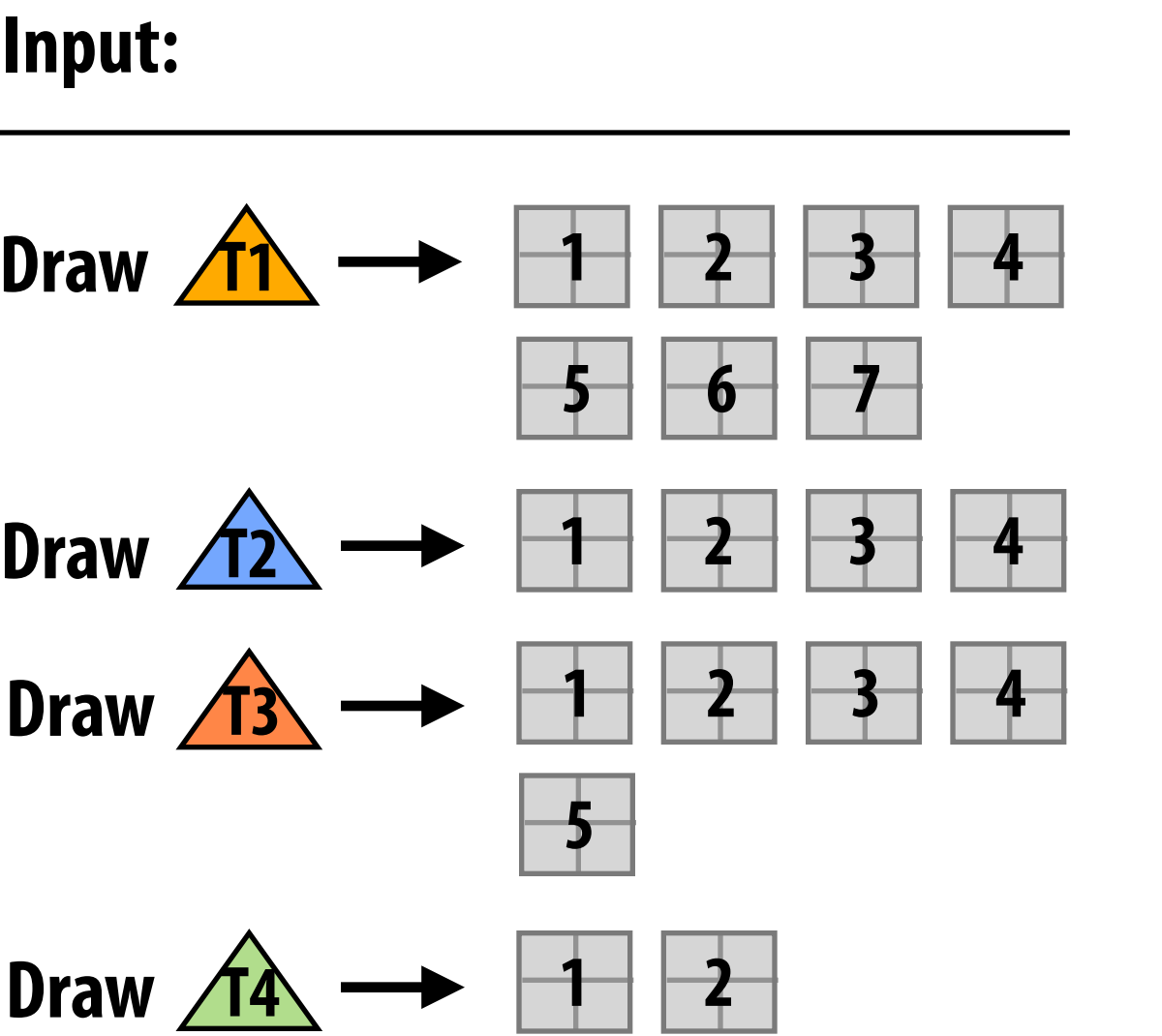
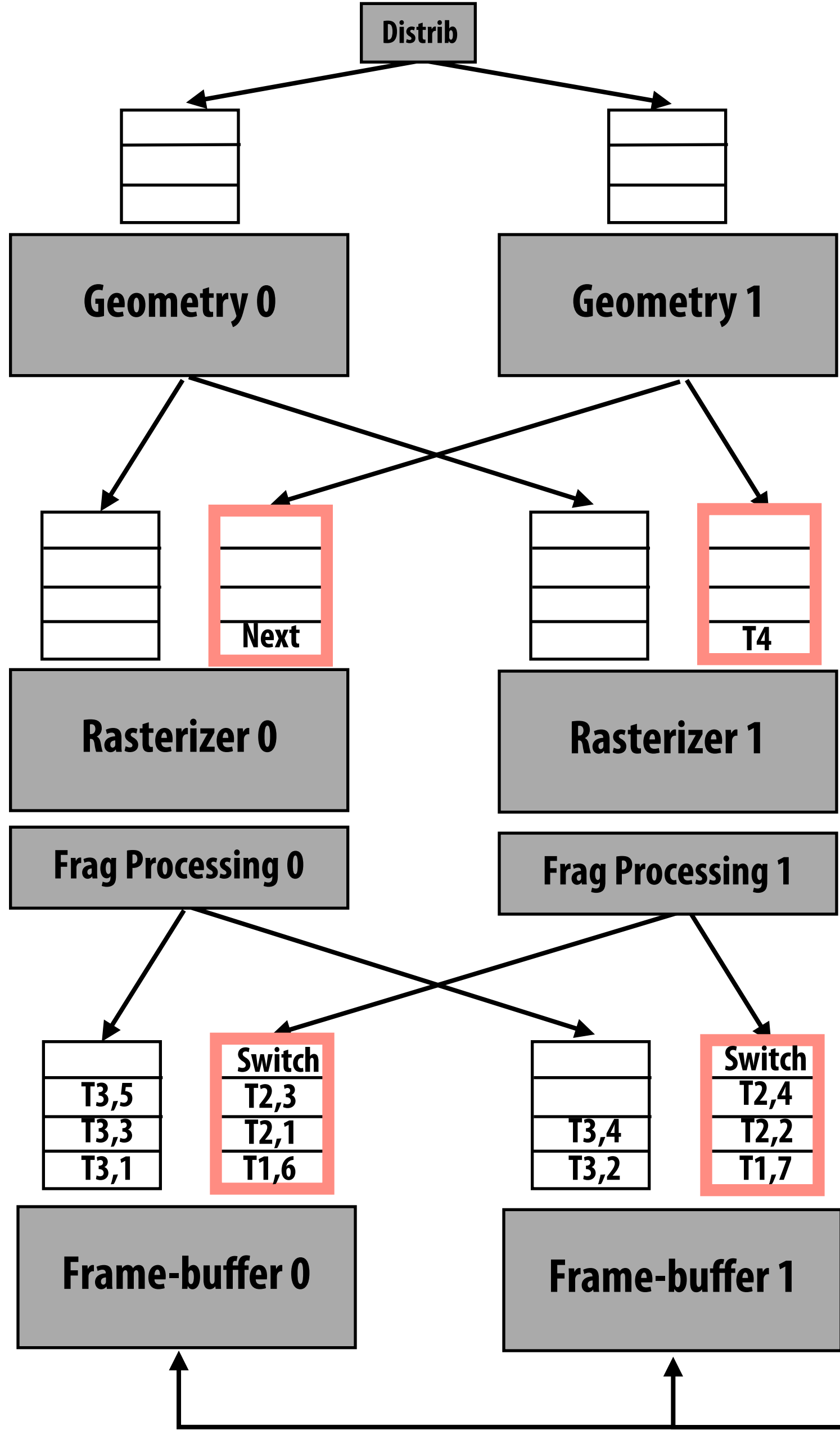
Input:



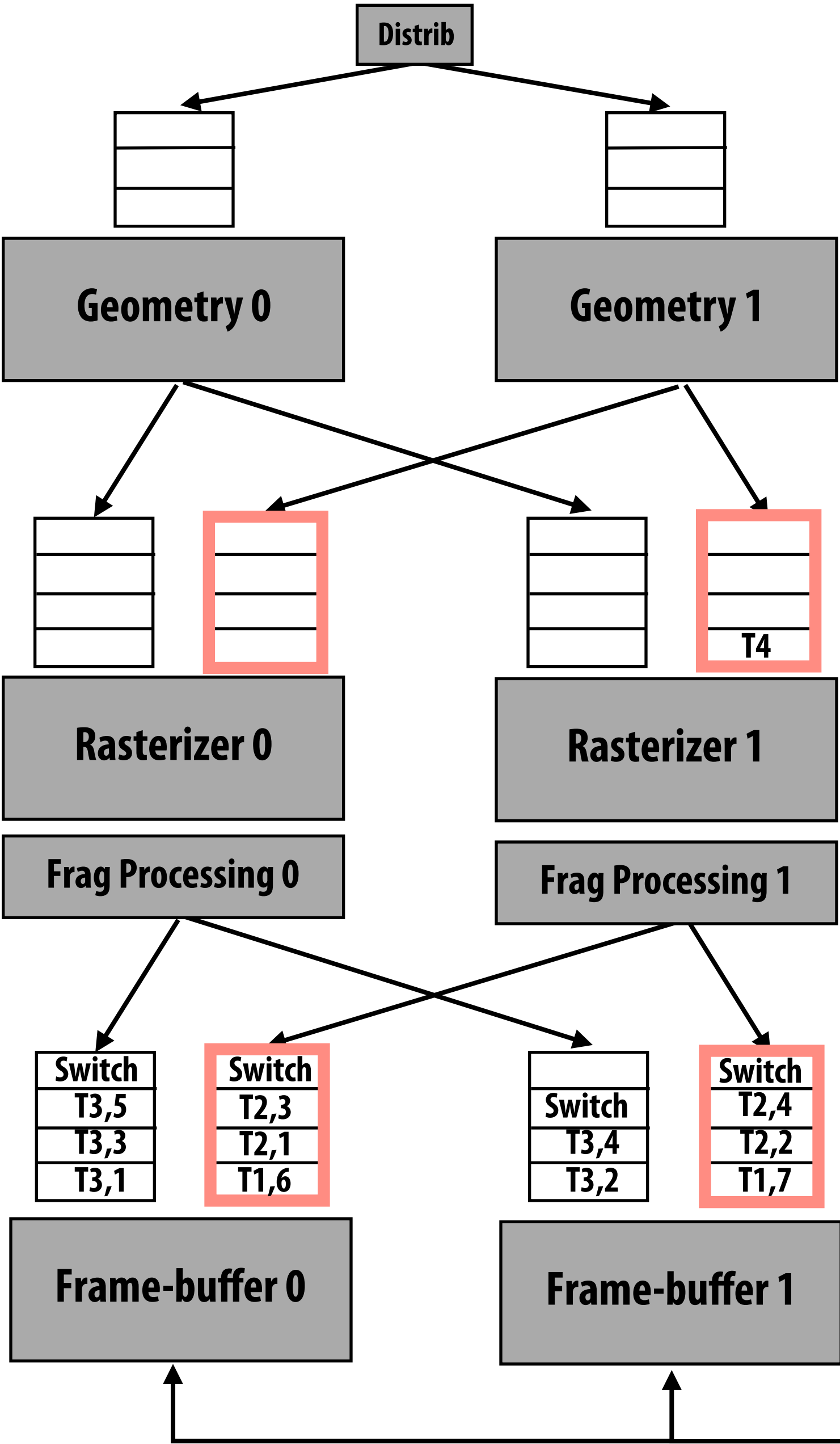
Interleaved
render target

Rast 0 processes triangles from geom 1

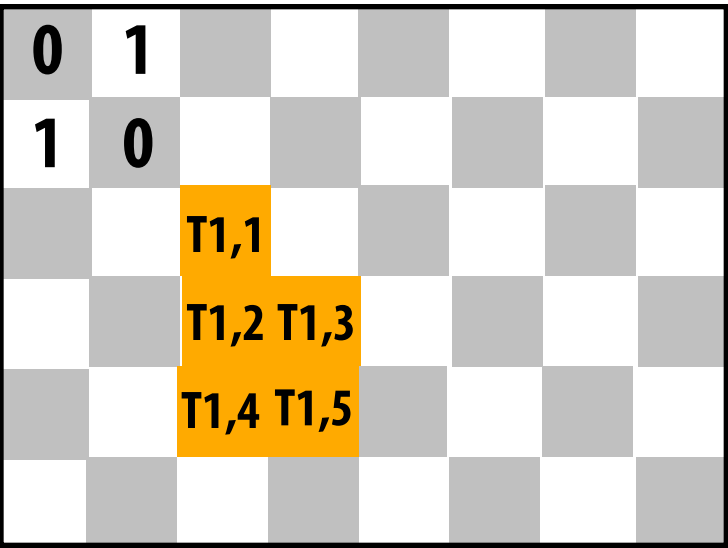
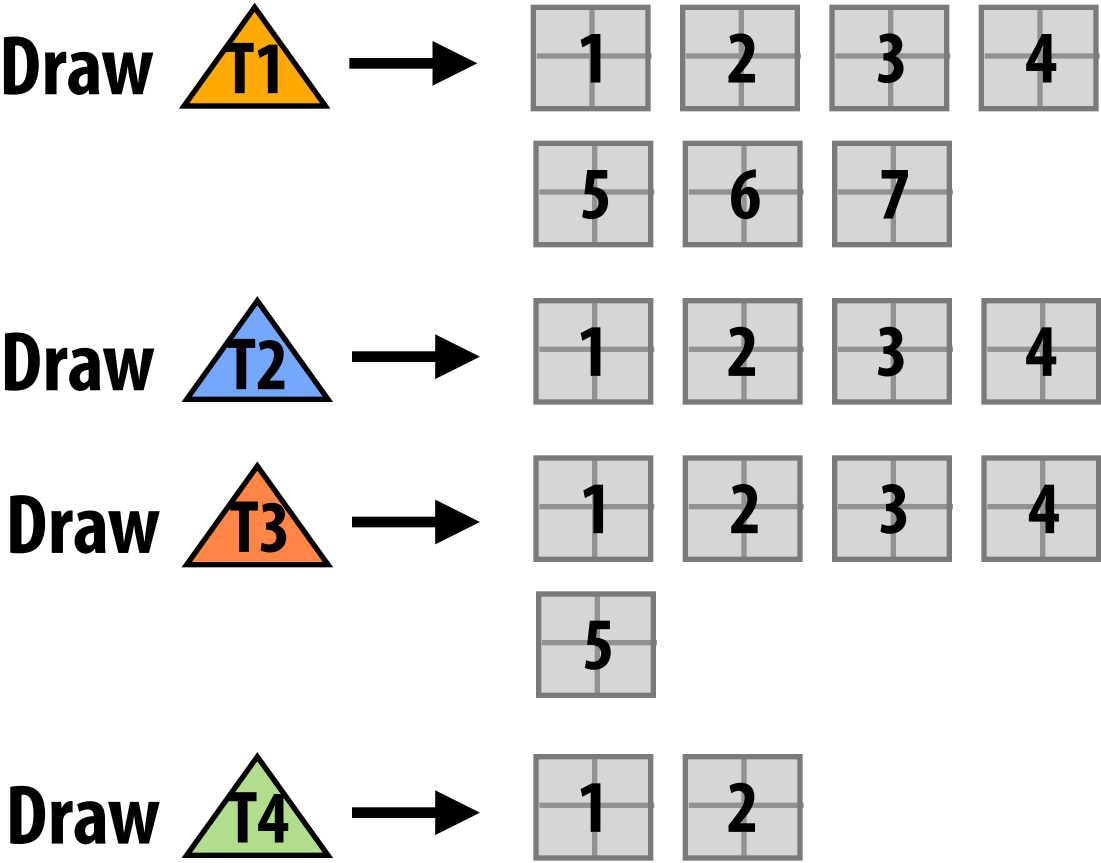
(Note rast 1 has work to do, but cannot make progress because its output queues are full)



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



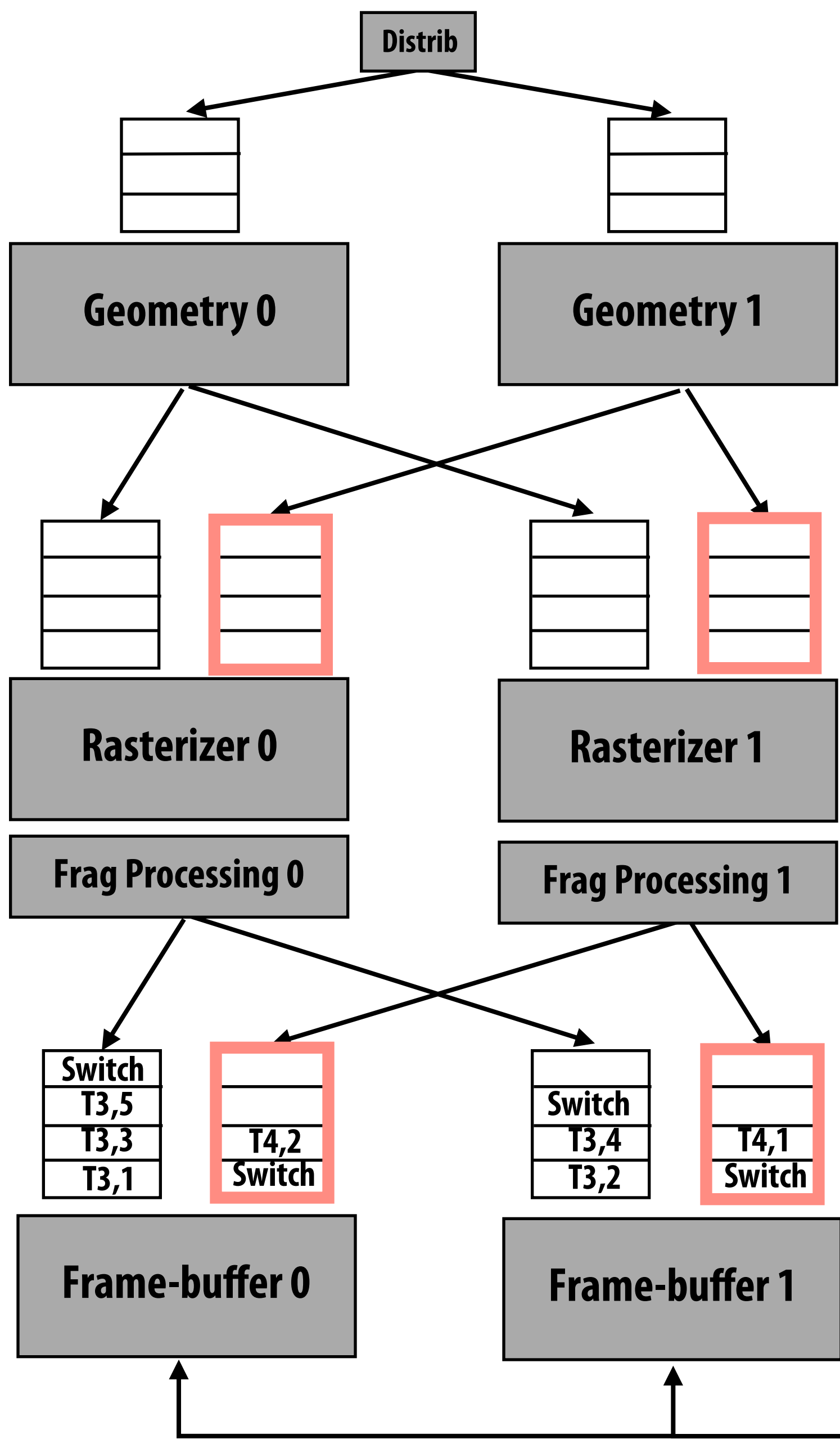
Input:



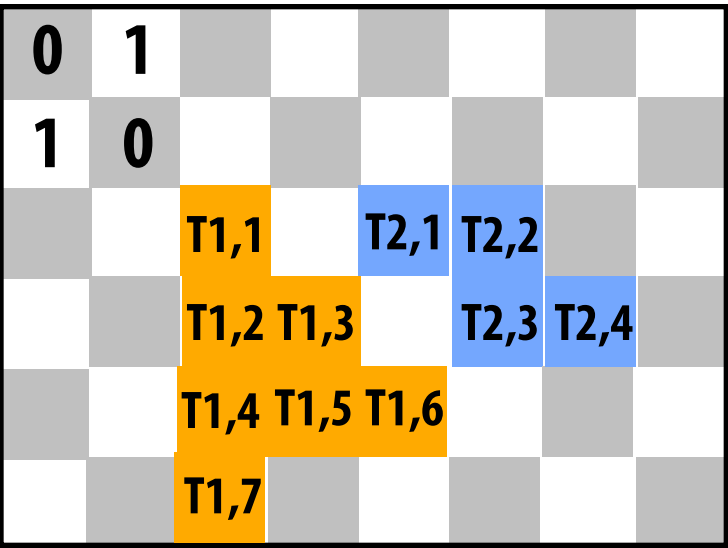
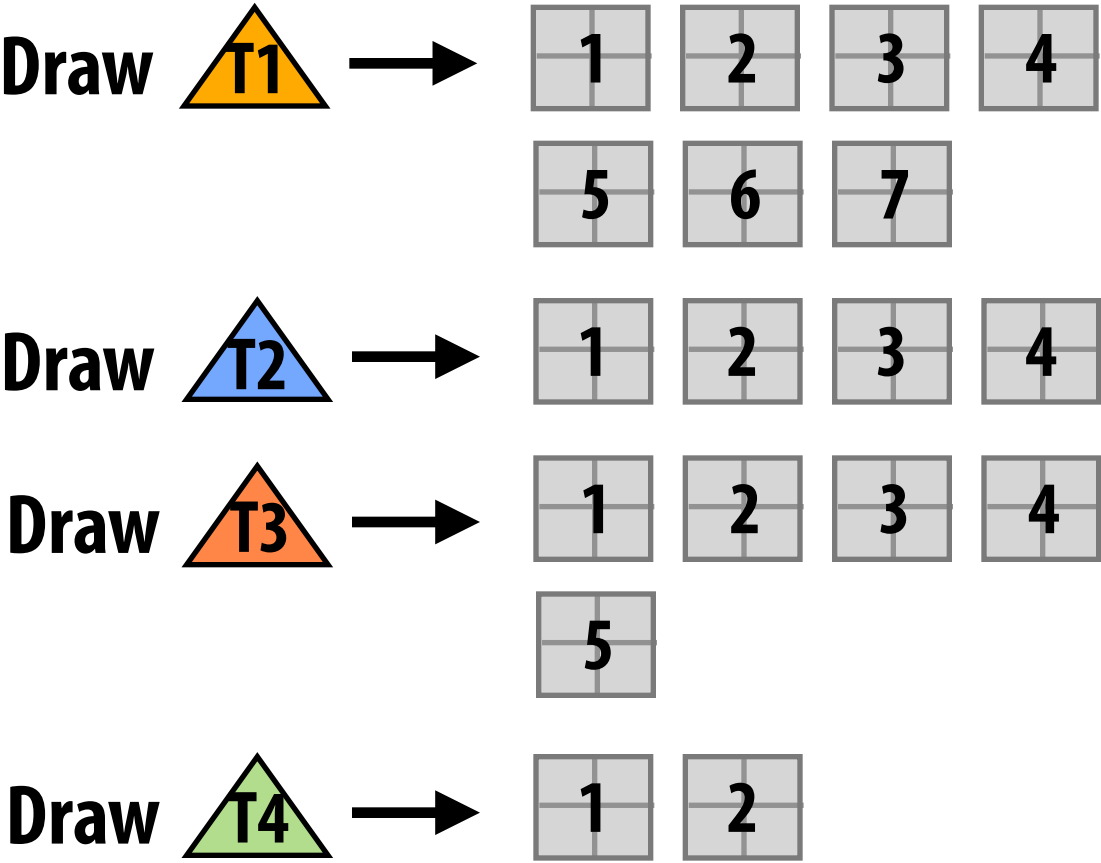
Interleaved
render target

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)

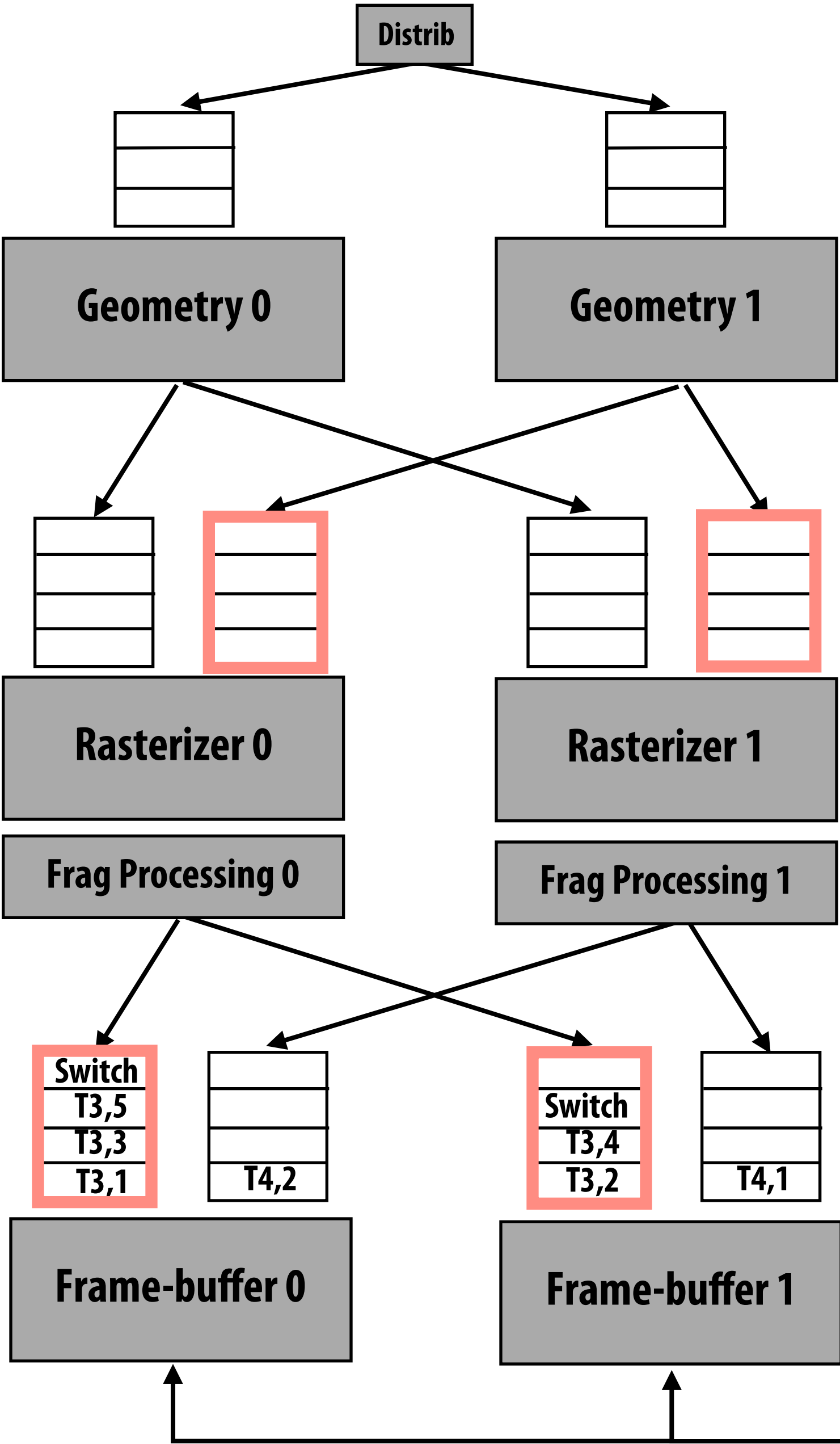


Input:

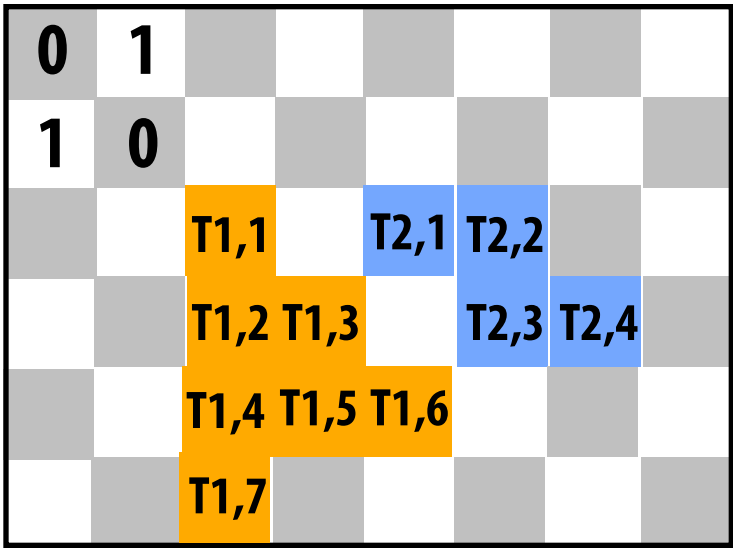
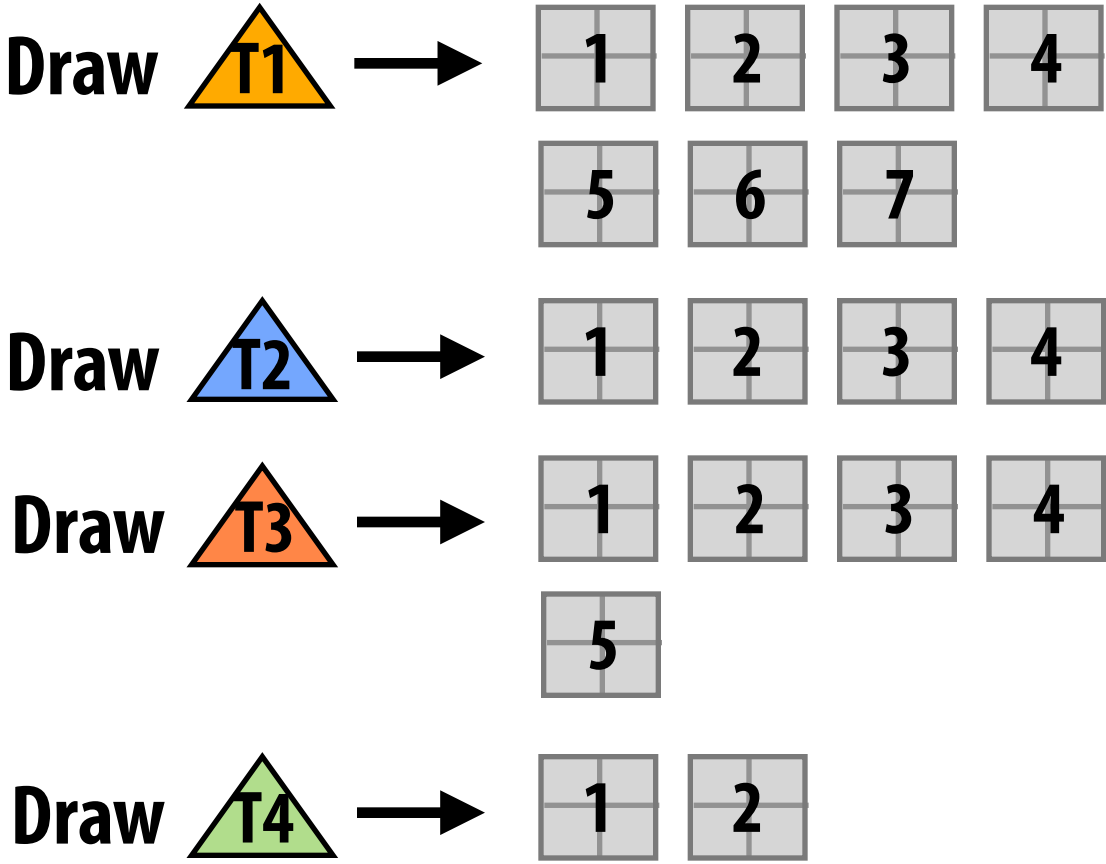


Interleaved
render target

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



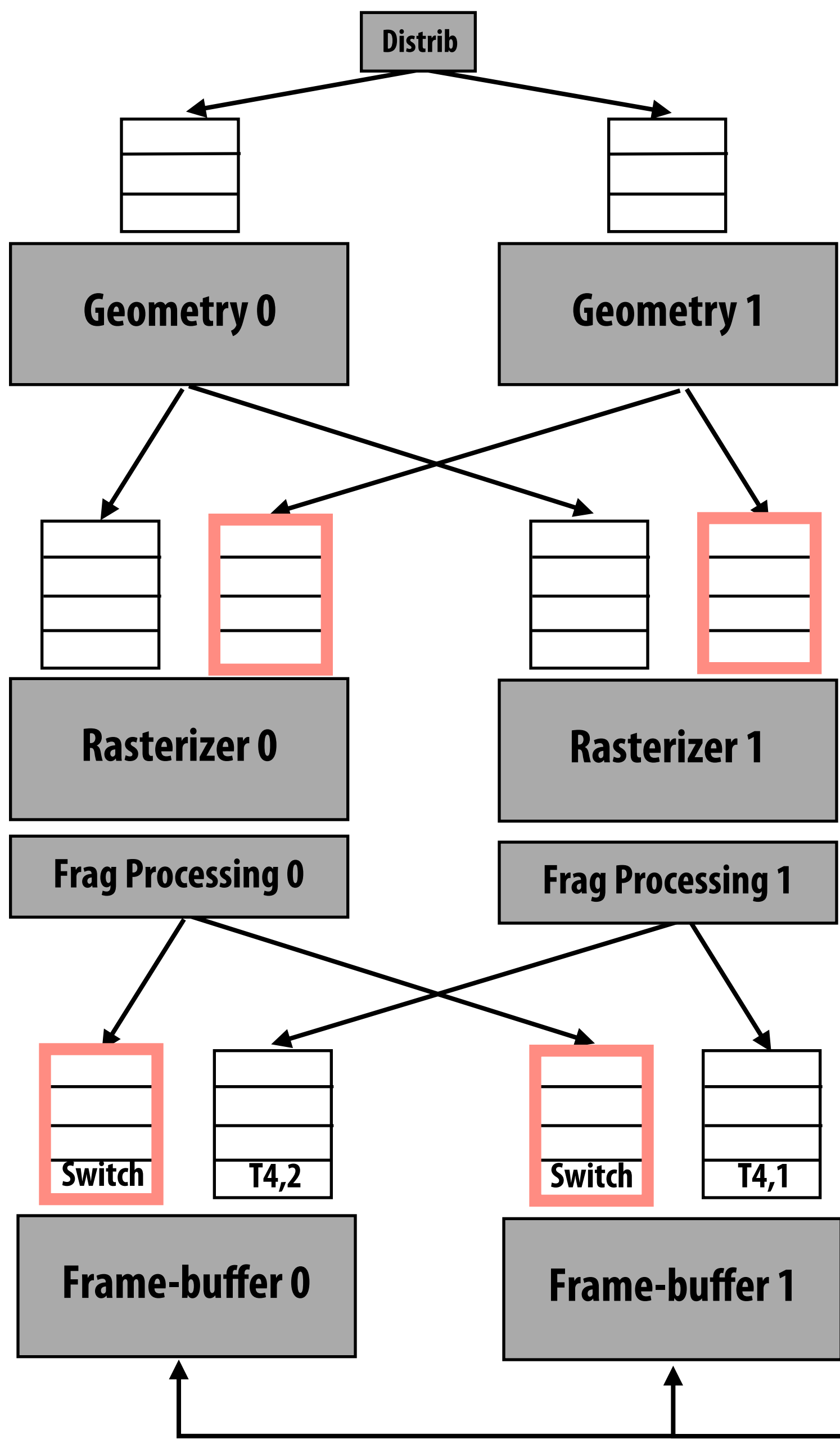
Input:



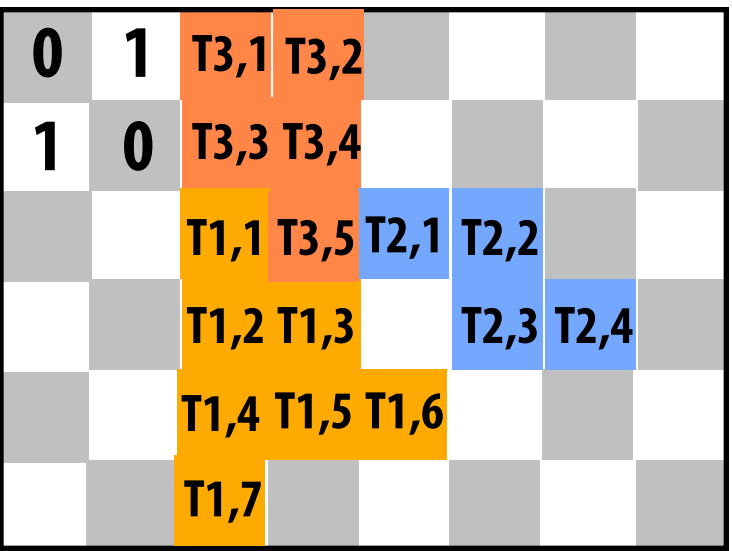
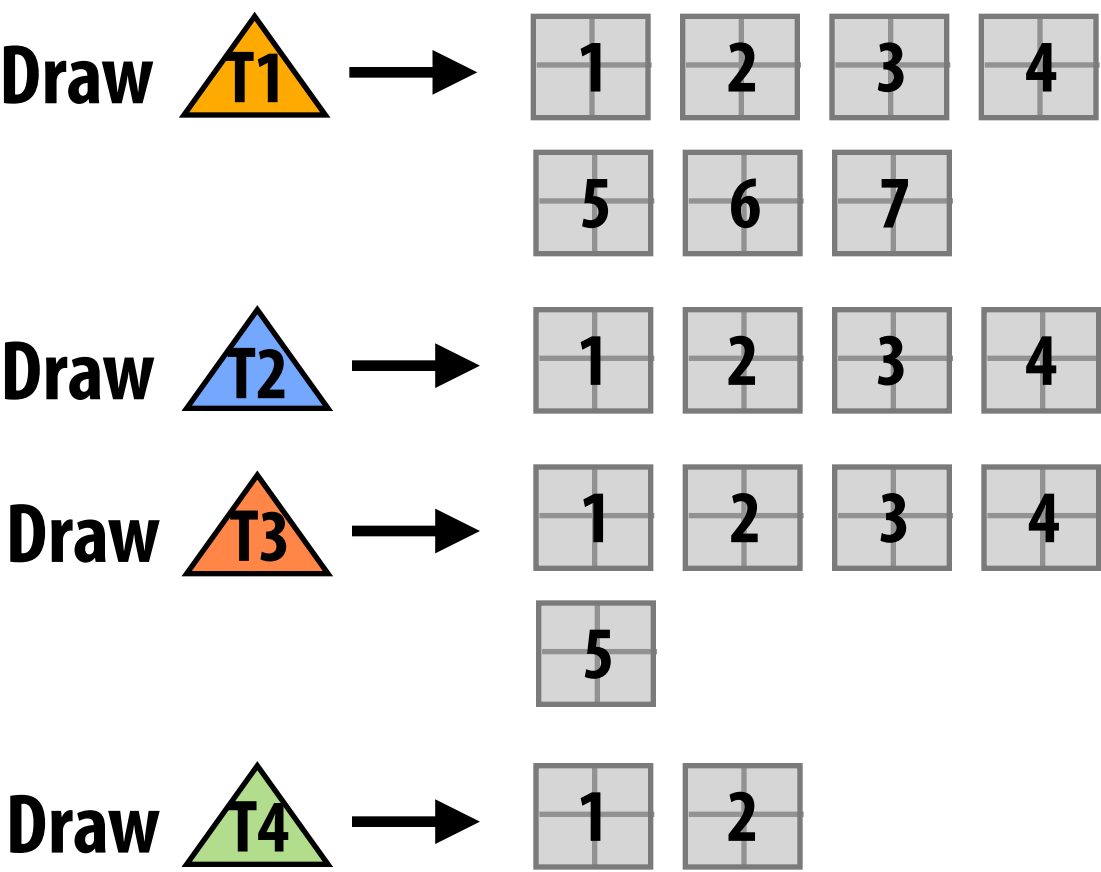
Interleaved
render target

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

(Notice updates to frame buffer)

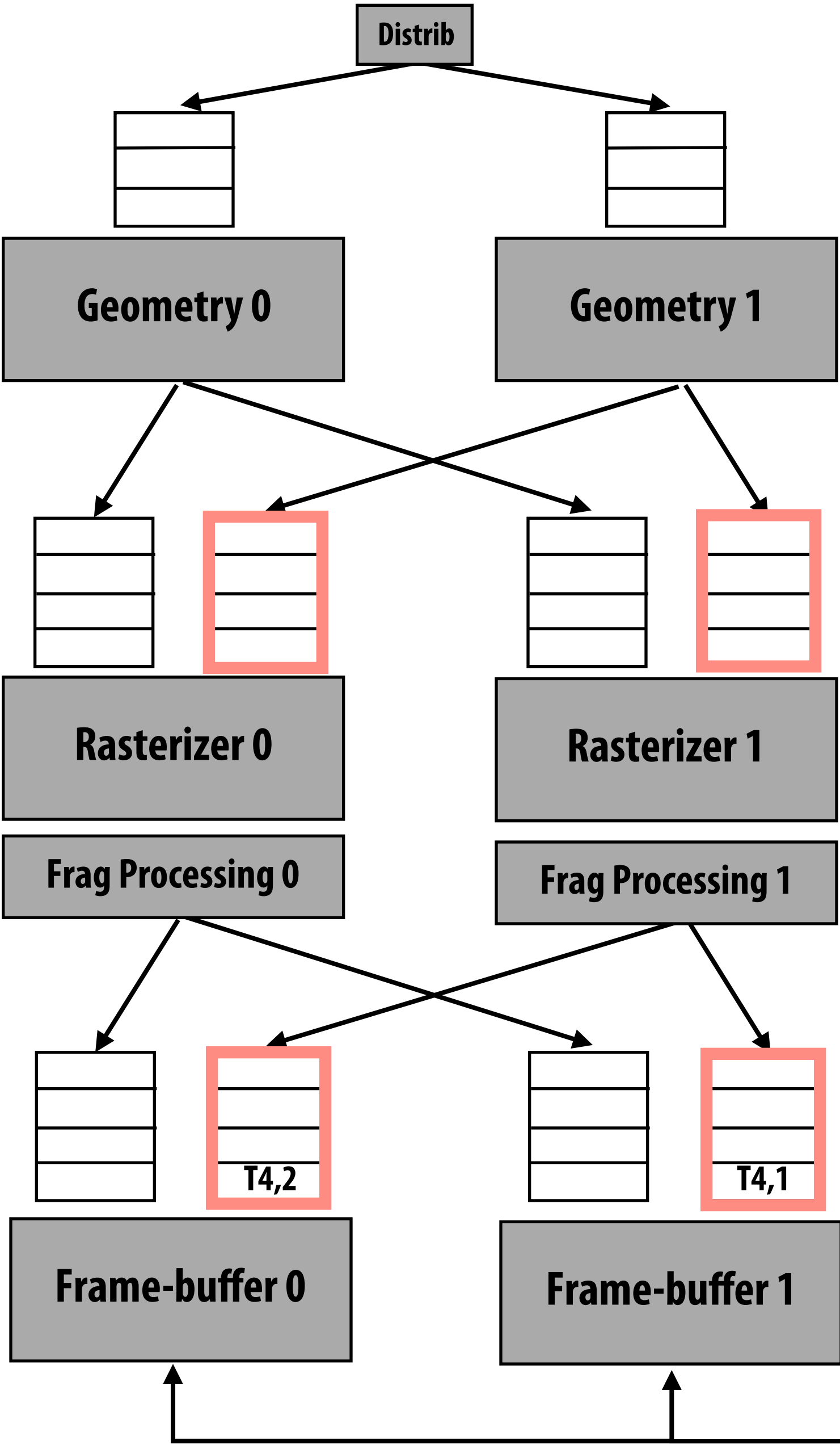


Input:

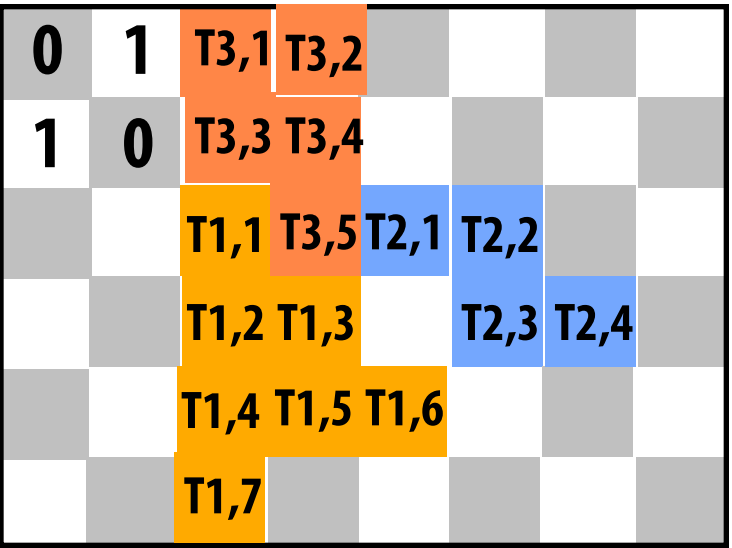
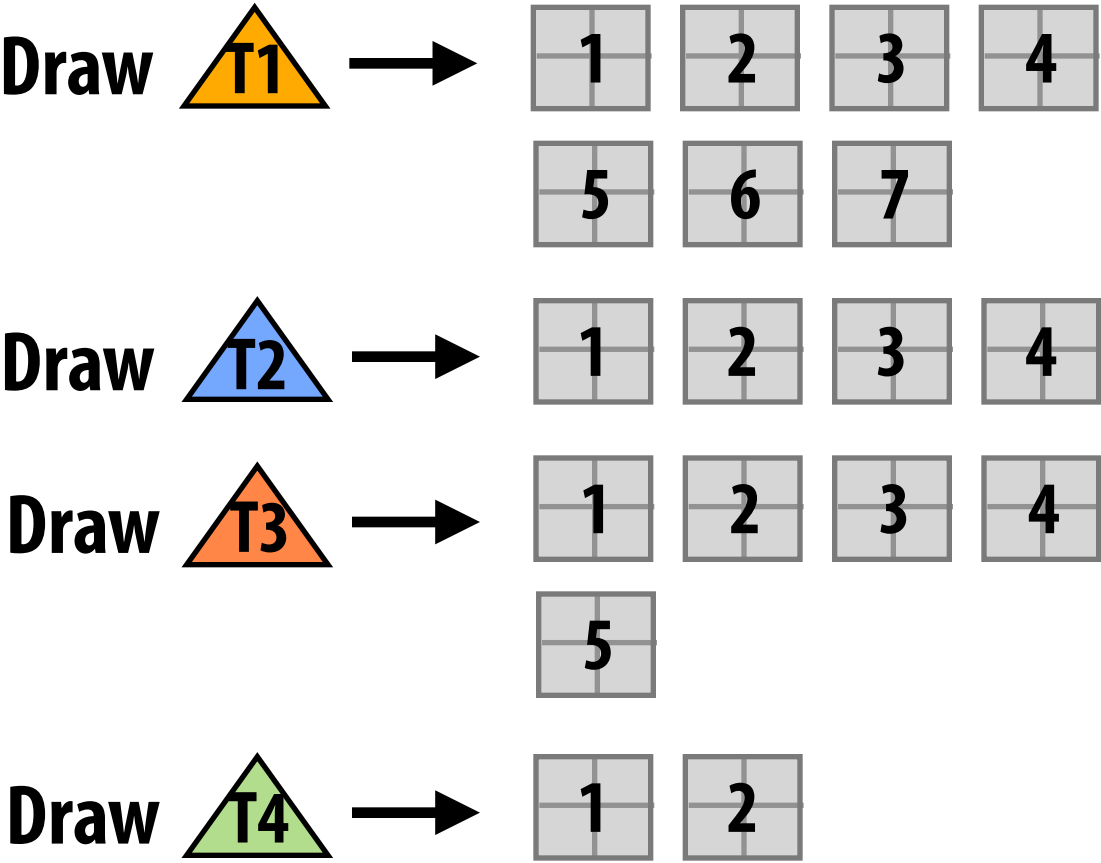


Interleaved
render target

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



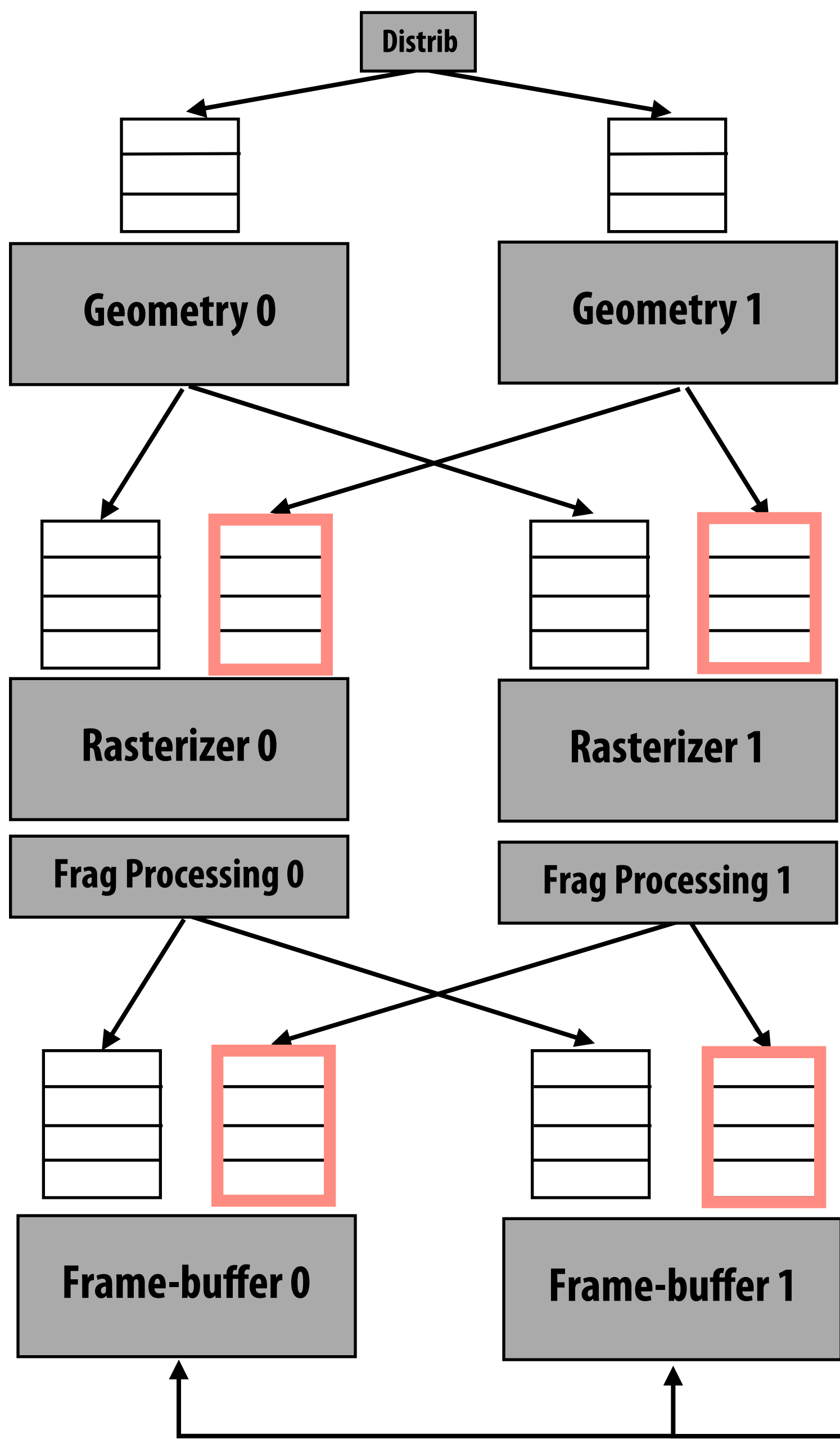
Input:



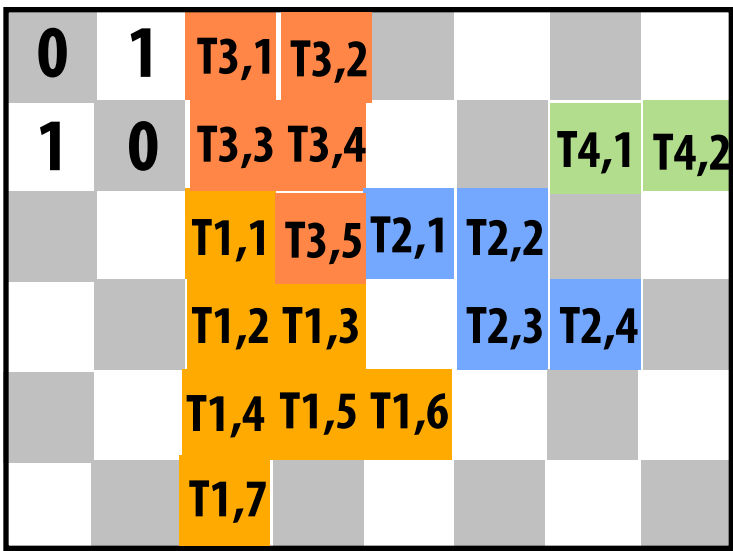
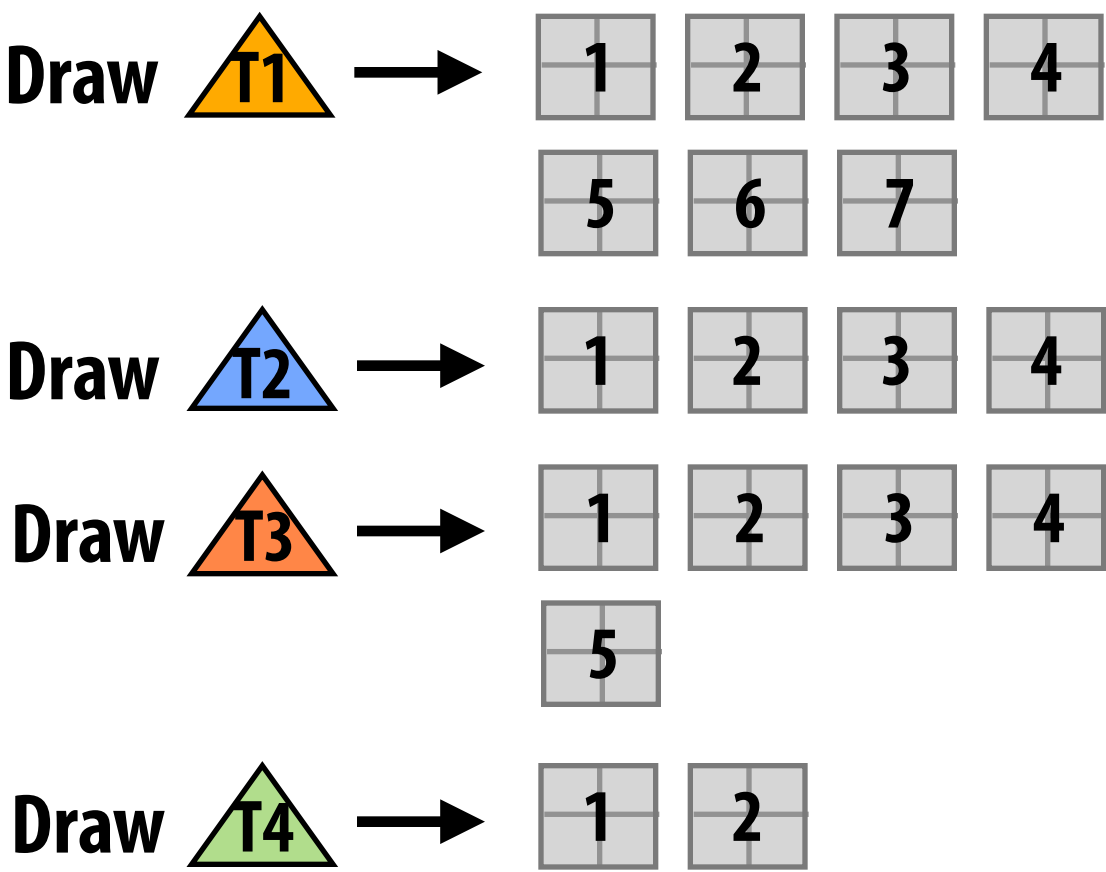
Interleaved
render target

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

(Notice updates to frame buffer)



Input:



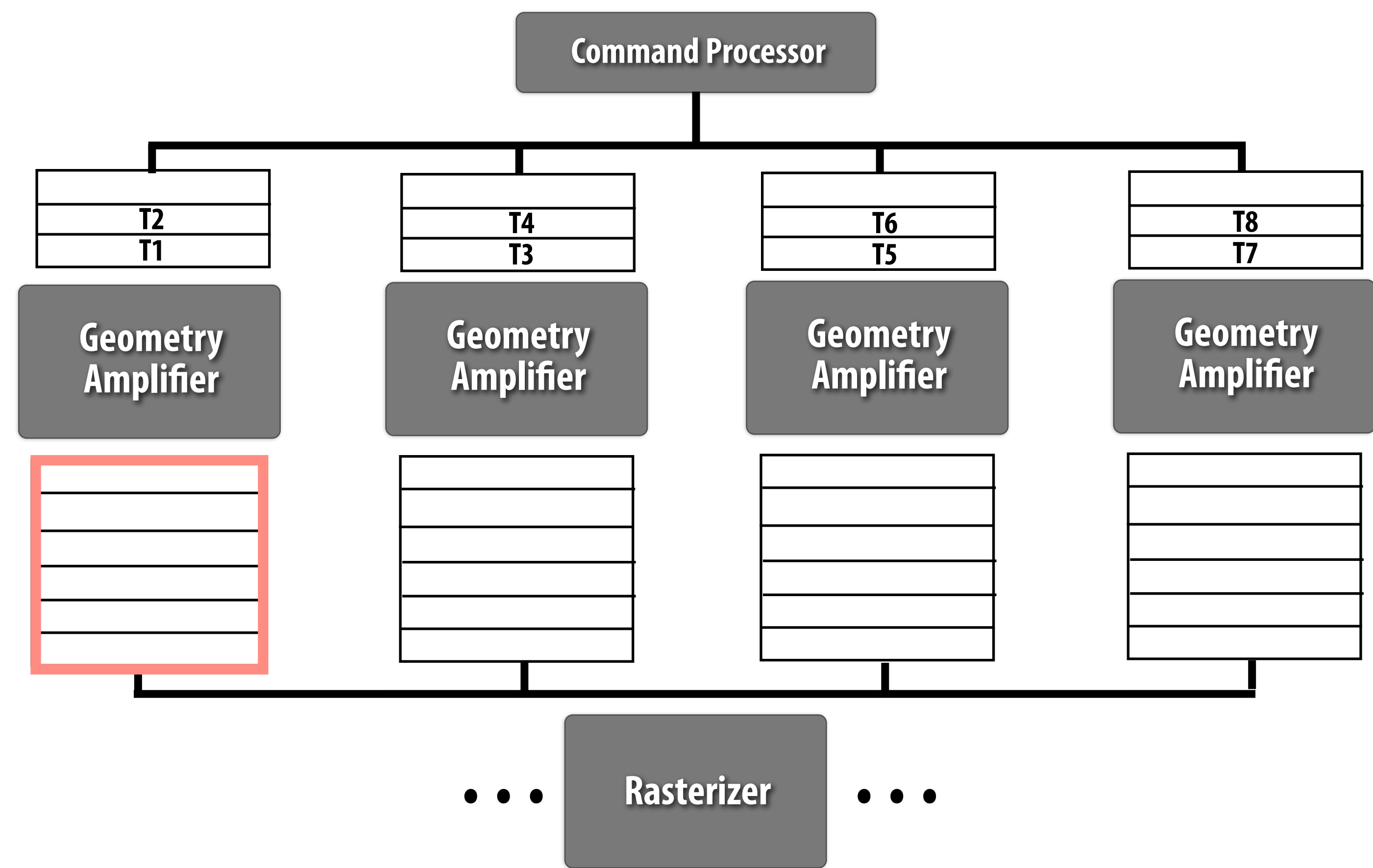
Interleaved
render target

Parallel scheduling with data amplification

Geometry amplification

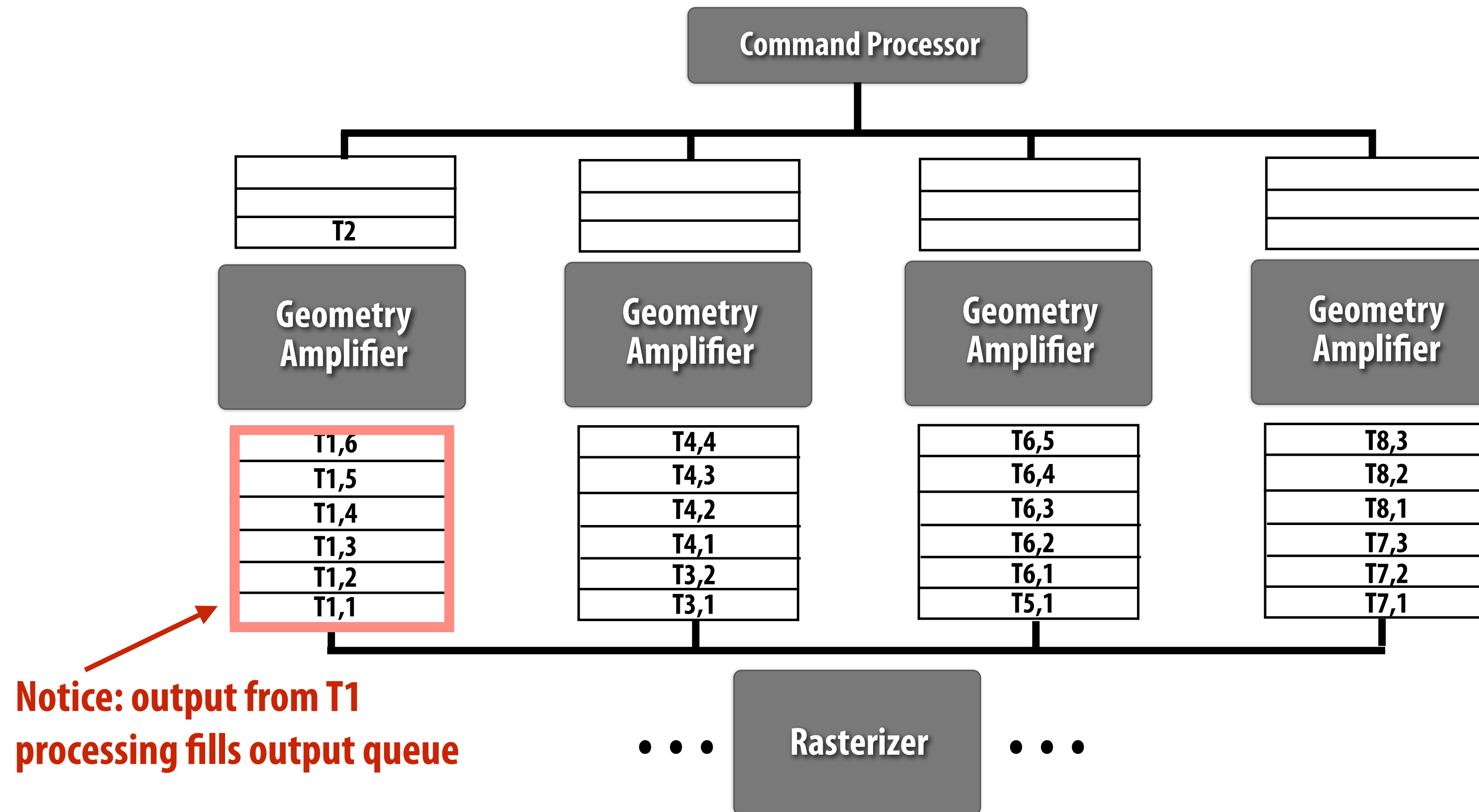
- **Consider examples of one-to-many stage behavior during geometry processing in the graphics pipeline:**
 - **Clipping amplifies geometry (clipping can result in multiple output primitives)**
 - **Tessellation: pipeline permits thousands of vertices to be generated from a single base primitive (challenging to maintain highly parallel execution)**
 - **Primitive processing (“geometry shader”) outputs up to 1024 floats worth of vertices per input primitive**

Thought experiment



Assume round-robin distribution of eight primitives to geometry pipelines, one rasterizer unit.

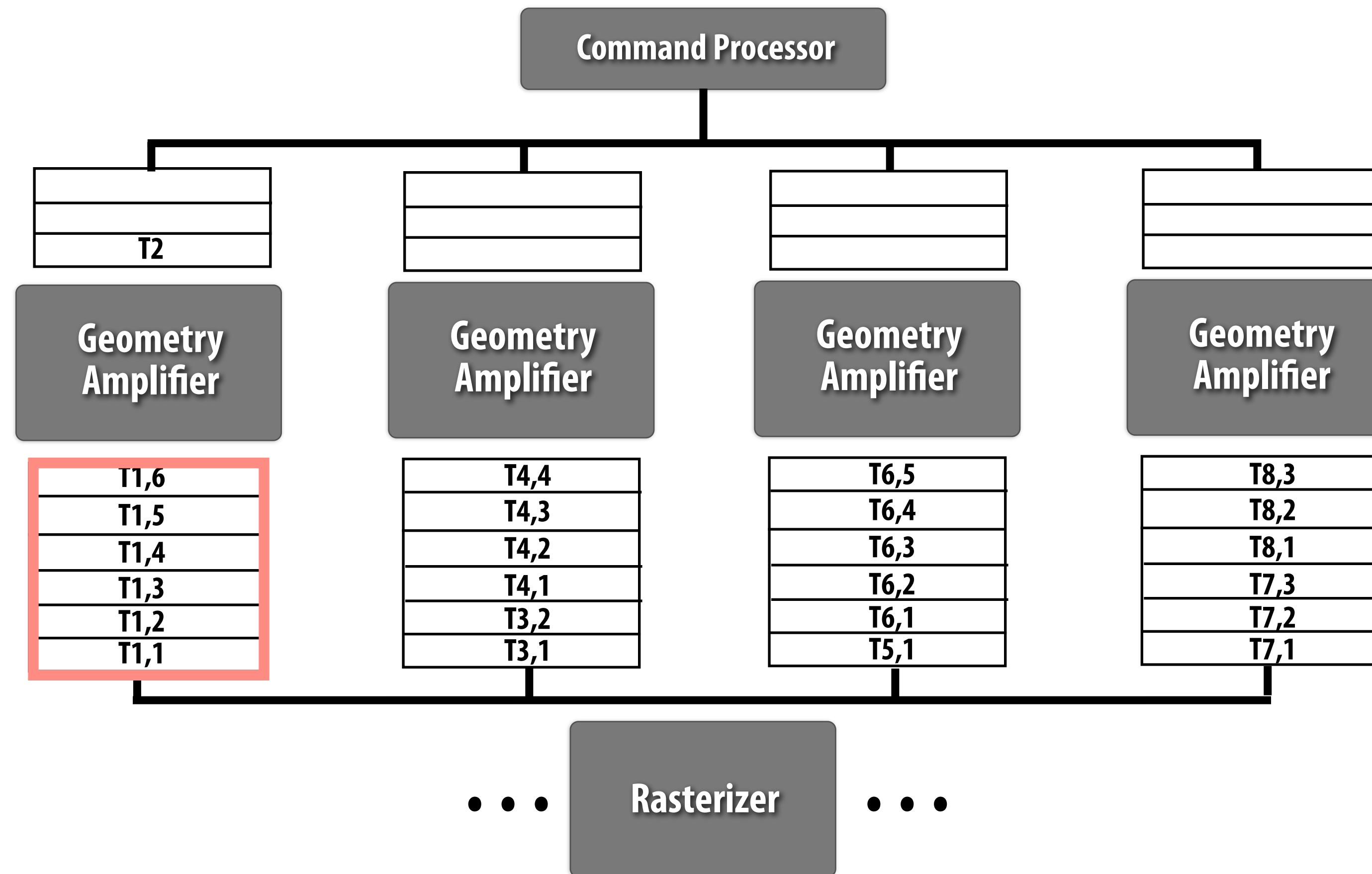
Consider case of large amplification when processing T1



Result: one geometry unit (the one producing outputs from T1) is feeding the entire downstream pipeline

- **Serialization of geometry processing: other geometry units are stalled because their output queues are full (they cannot be drained until all work from T1 is completed)**
- **Underutilization of rest of chip: unlikely that one geometry producer is fast enough to produce pipeline work at a rate that fills resources of rest of GPU.**

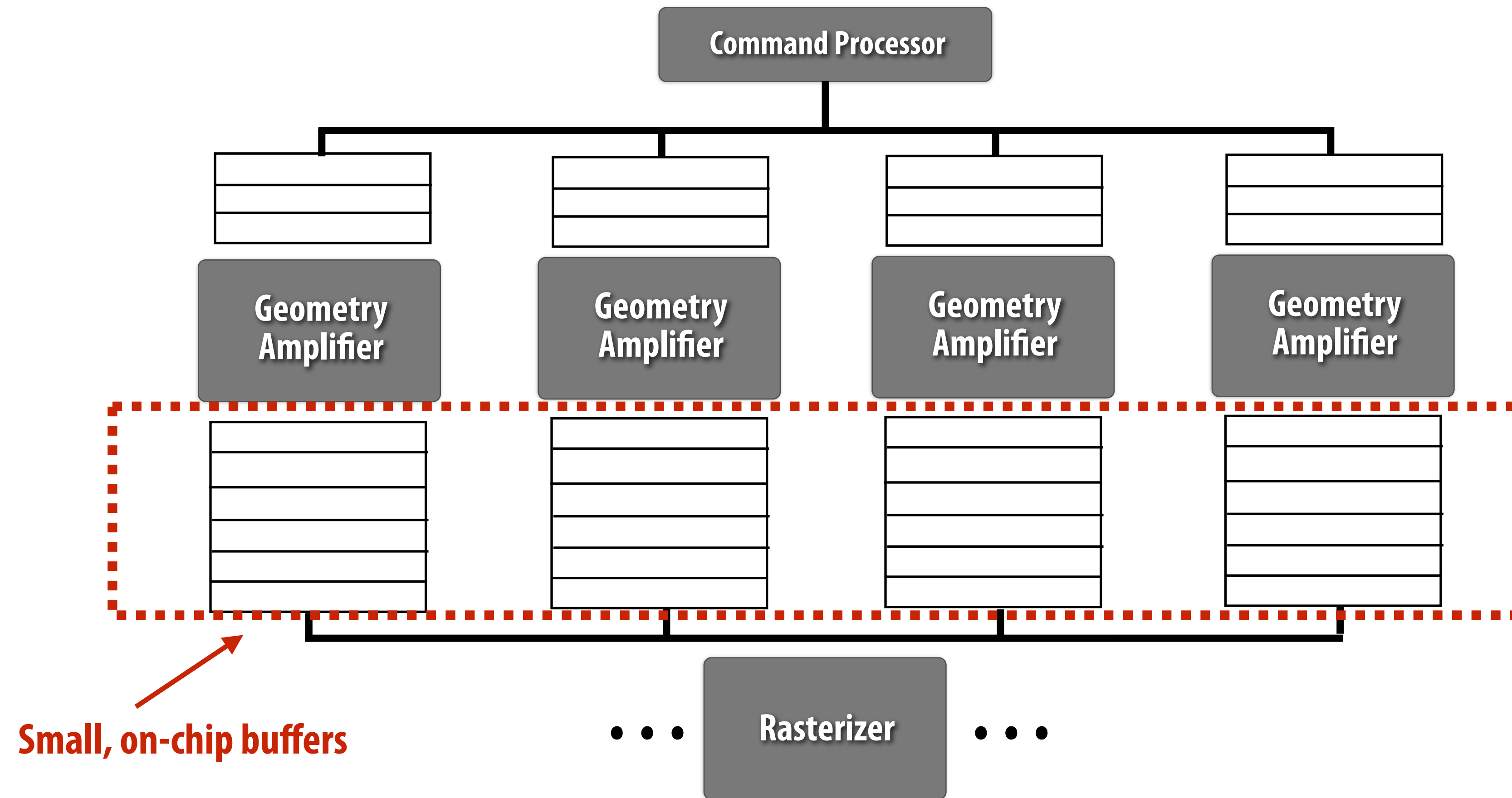
Thought experiment: design a scheduling strategy for this case



1. Design a solution that is performant when the expected amount of data amplification is low?
2. Design a solution this is performant when the expected amount of data amplification is high
3. What about a solution that works well for both?

The ideal solution always executes with maximum parallelism (no stalls), and with maximal locality (units read and write to fixed size, on-chip inter-stage buffers), and (of course) preserves order.

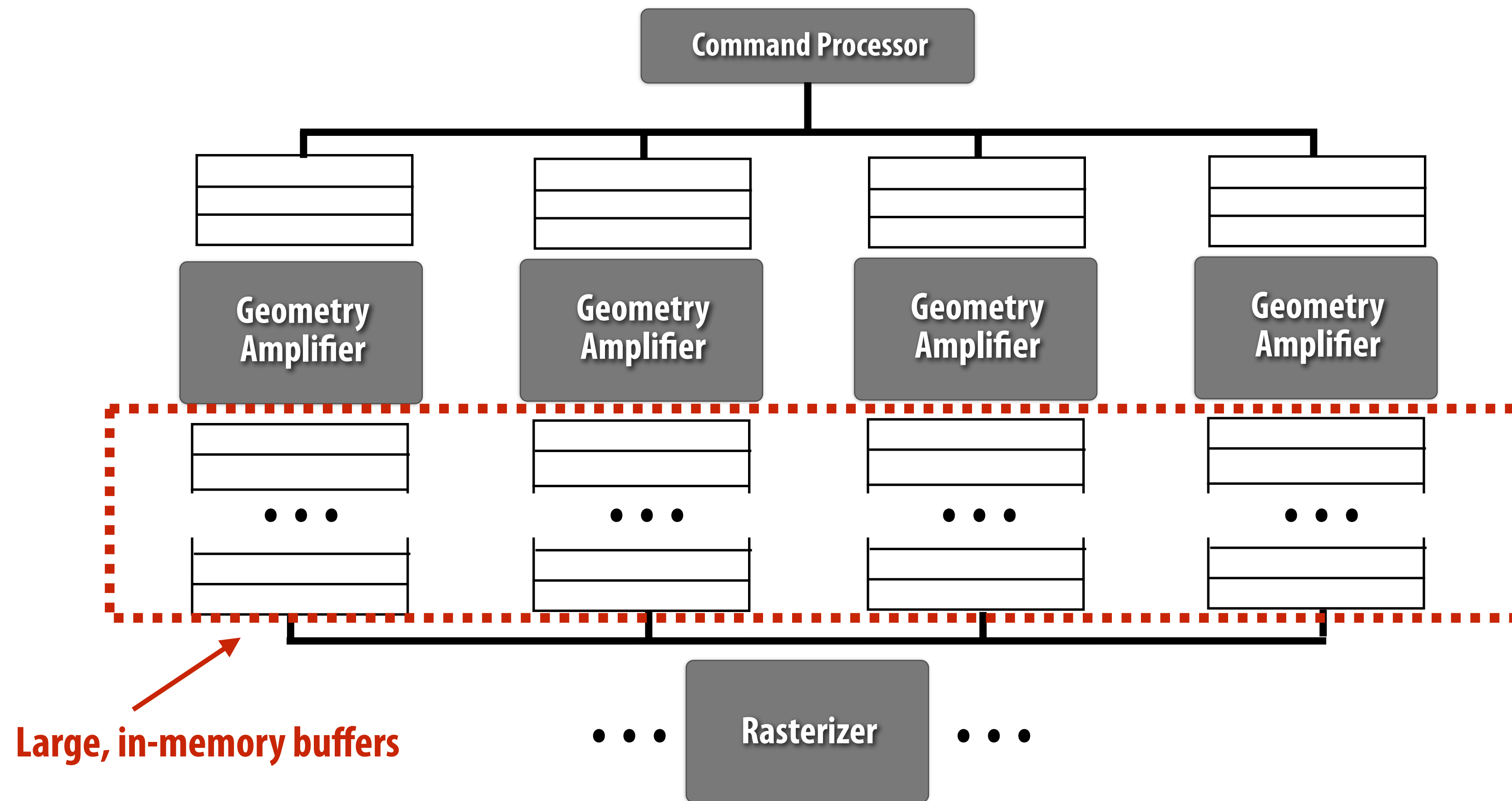
Implementation 1: fixed on-chip storage



Approach 1: make on-chip buffers big enough to handle common cases, but tolerate stalls

- Run fast for low amplification (never move output queue data off chip)
- Run very slow under high amplification (serialization of processing due to blocked units). Bad performance cliff.

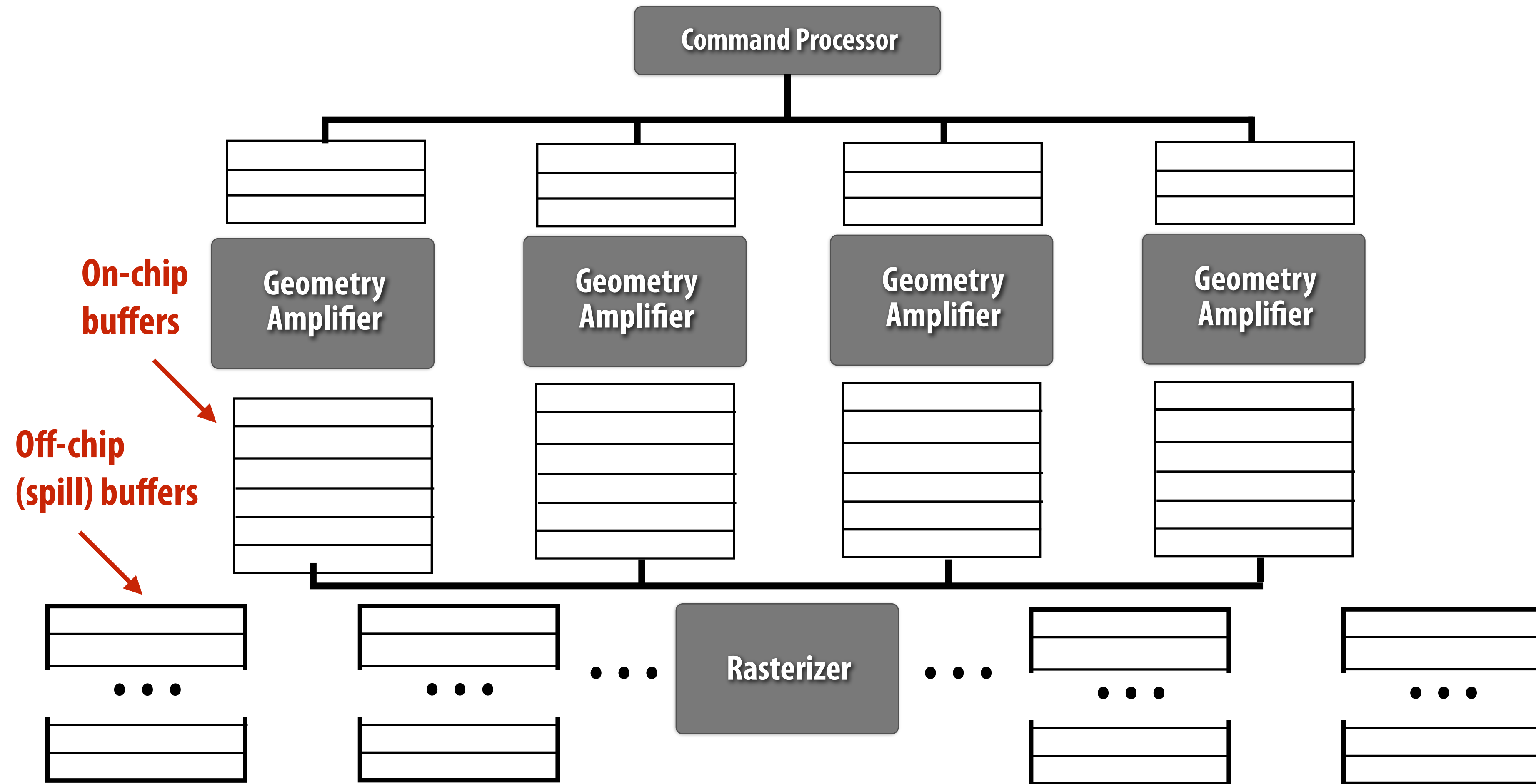
Implementation 2: worst-case allocation



Approach 2: never block geometry unit: allocate worst-case space in off-chip buffers (stored in DRAM)

- **Run slower for low amplification (data goes off chip then read back in by rasterizers)**
- **No performance cliff for high amplification (still maximum parallelism, data still goes off chip)**
- **What is overall worst-case buffer allocation if the four geometry units above are Direct3D 11 geometry shaders?**

Implementation 3: hybrid



Hybrid approach: allocate output buffers on chip, but spill to off-chip, worst-case size buffers under high amplification

- Run fast for low amplification (high parallelism, no memory traffic)
- Less of performance cliff for high amplification (high parallelism, but incurs more memory traffic)

Summary

- **Graphics pipeline: abstract machine for executing graphics commands**
 - **Draw triangles**
 - **Change state**
- **Mapping execution of sequence of these commands to a parallel GPU is a complex scheduling problem**
 - **Scheduling done by “graphics mode” execution of the GPU**