Lecture 3:

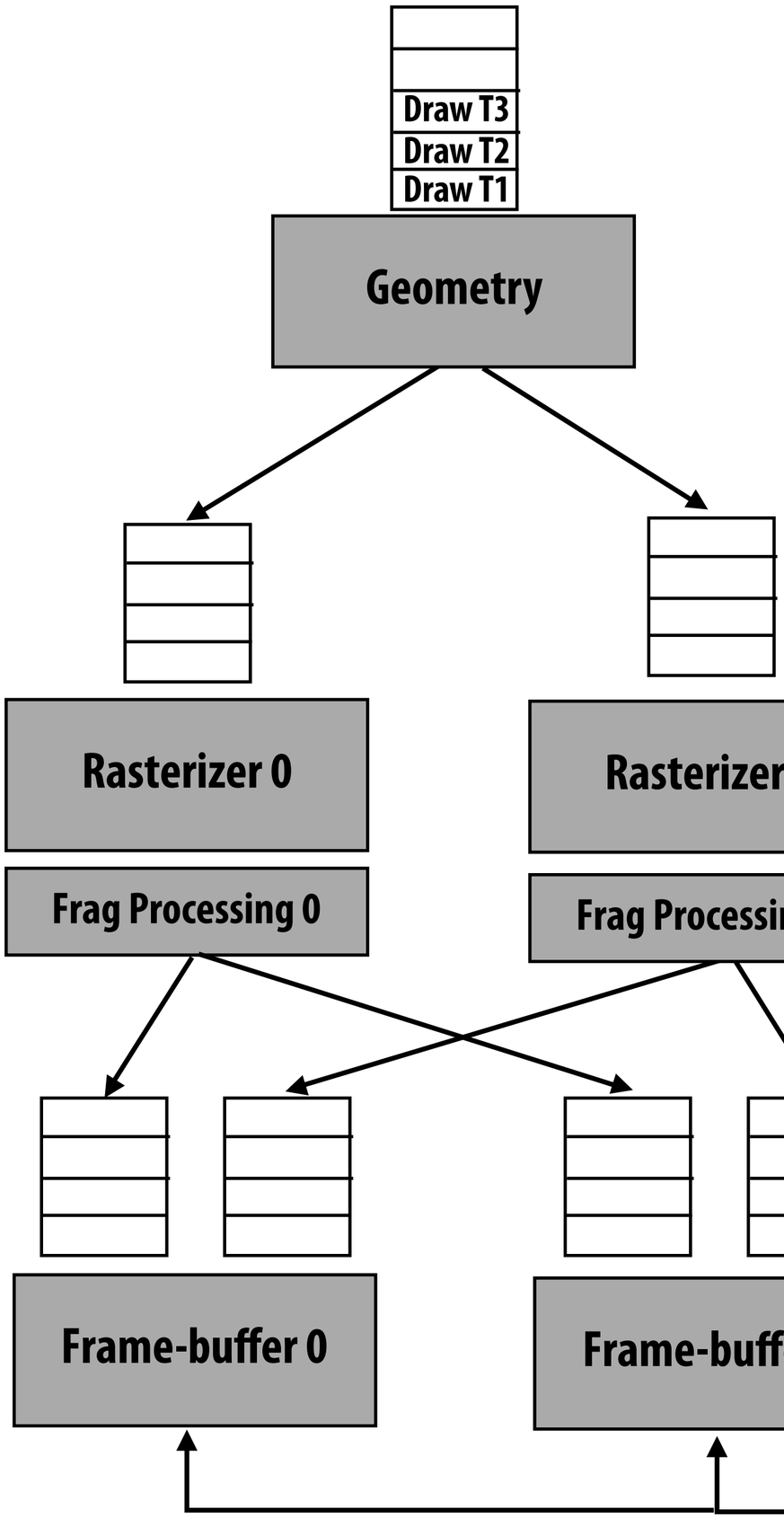
Sort-Everywhere Pipeline Scheduling and Geometry Processing

Visual Computing Systems
CMU 15-869, Fall 2014

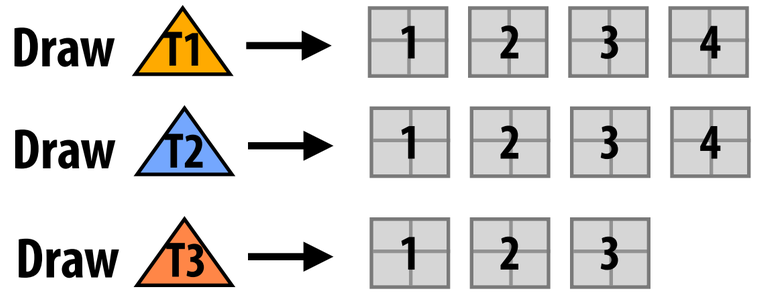
Scheduling a sort-everywhere graphics pipeline

The following figures follows the design of Pomegranate [Eldridge et al.], but use stage names consistent with modern graphics pipeline.

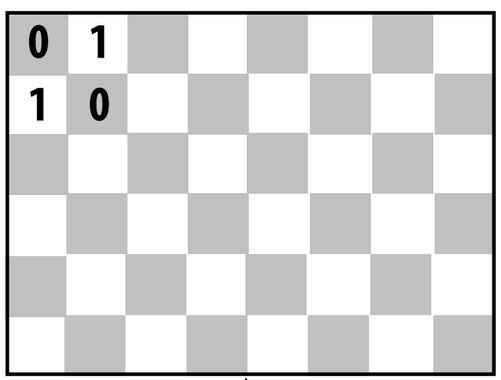
Starting state: draw commands enqueued for pipeline



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

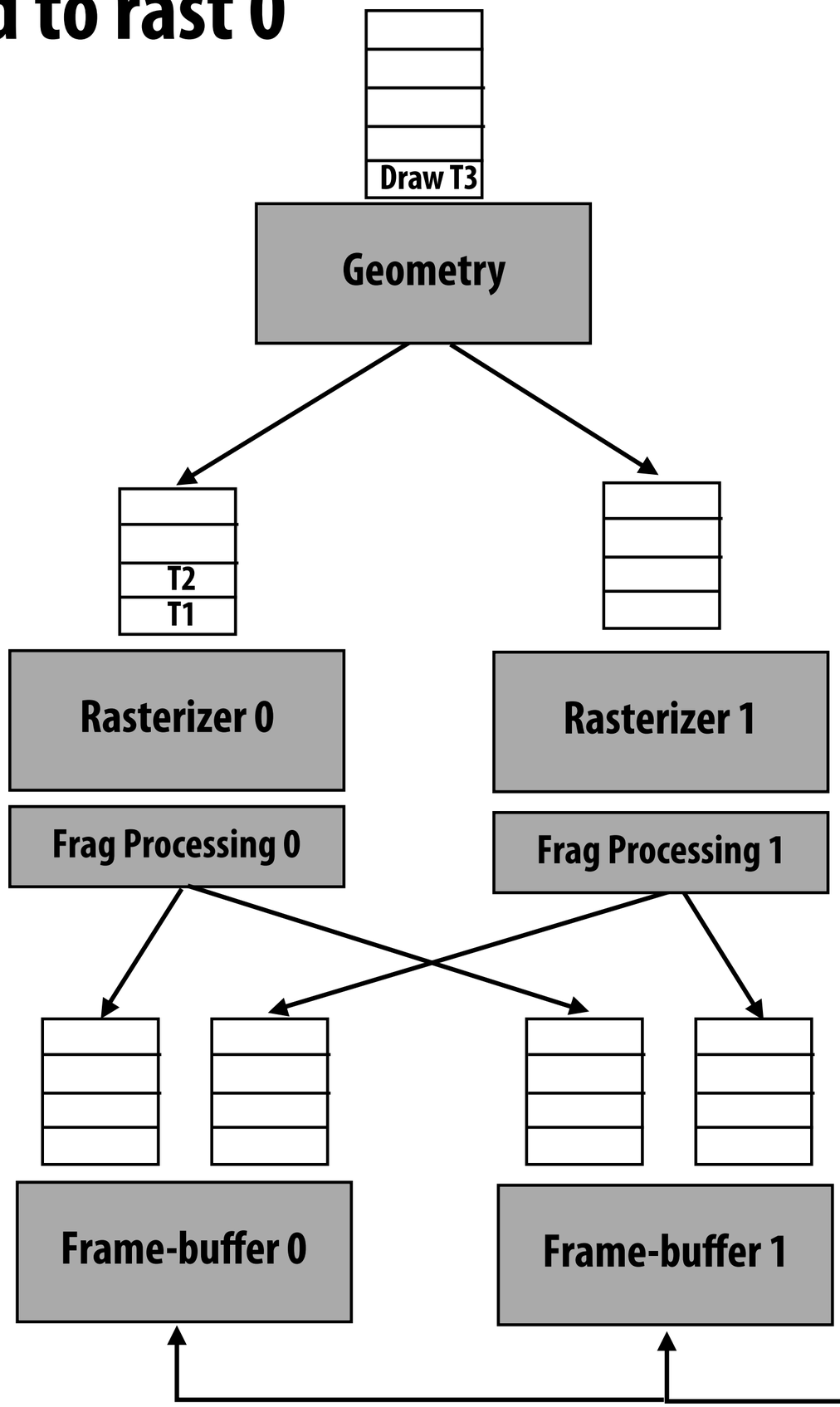


Assume batch size is 2 for assignment to rasterizers.

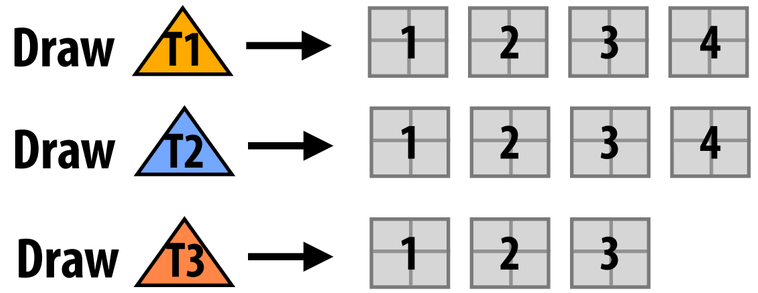


Interleaved render target

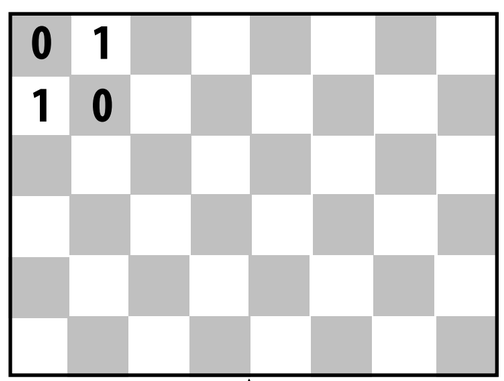
After geometry processing, first two processed triangles assigned to rast 0



Input:



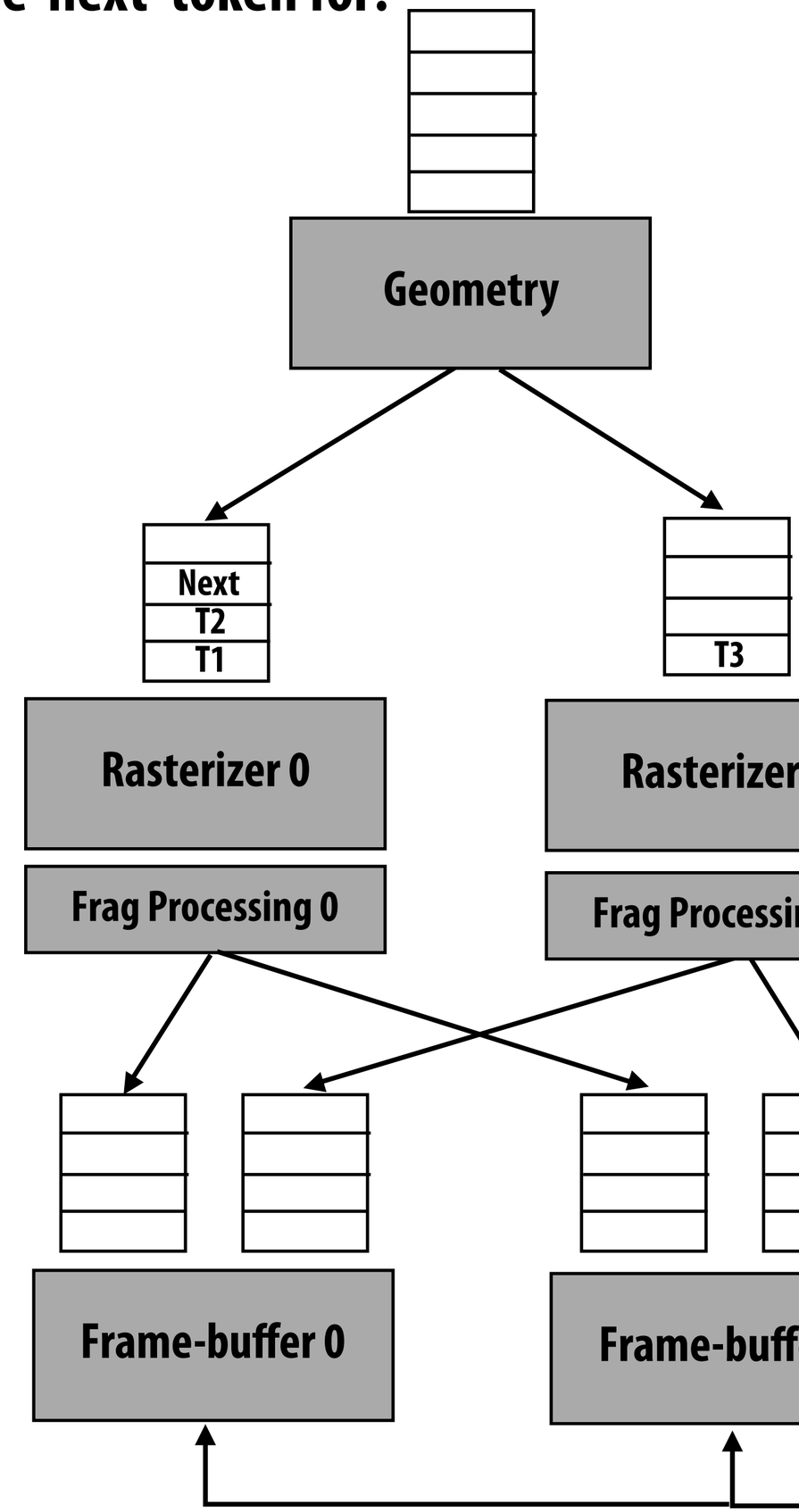
Assume batch size is 2 for assignment to rasterizers.



Interleaved render target

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

Q. What is the 'next' token for?

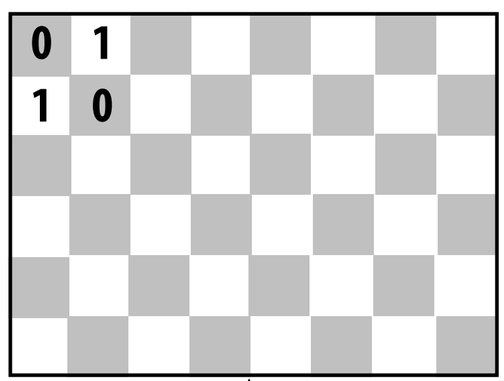


Input:

Draw T1 → 1 2 3 4

Draw T2 → 1 2 3 4

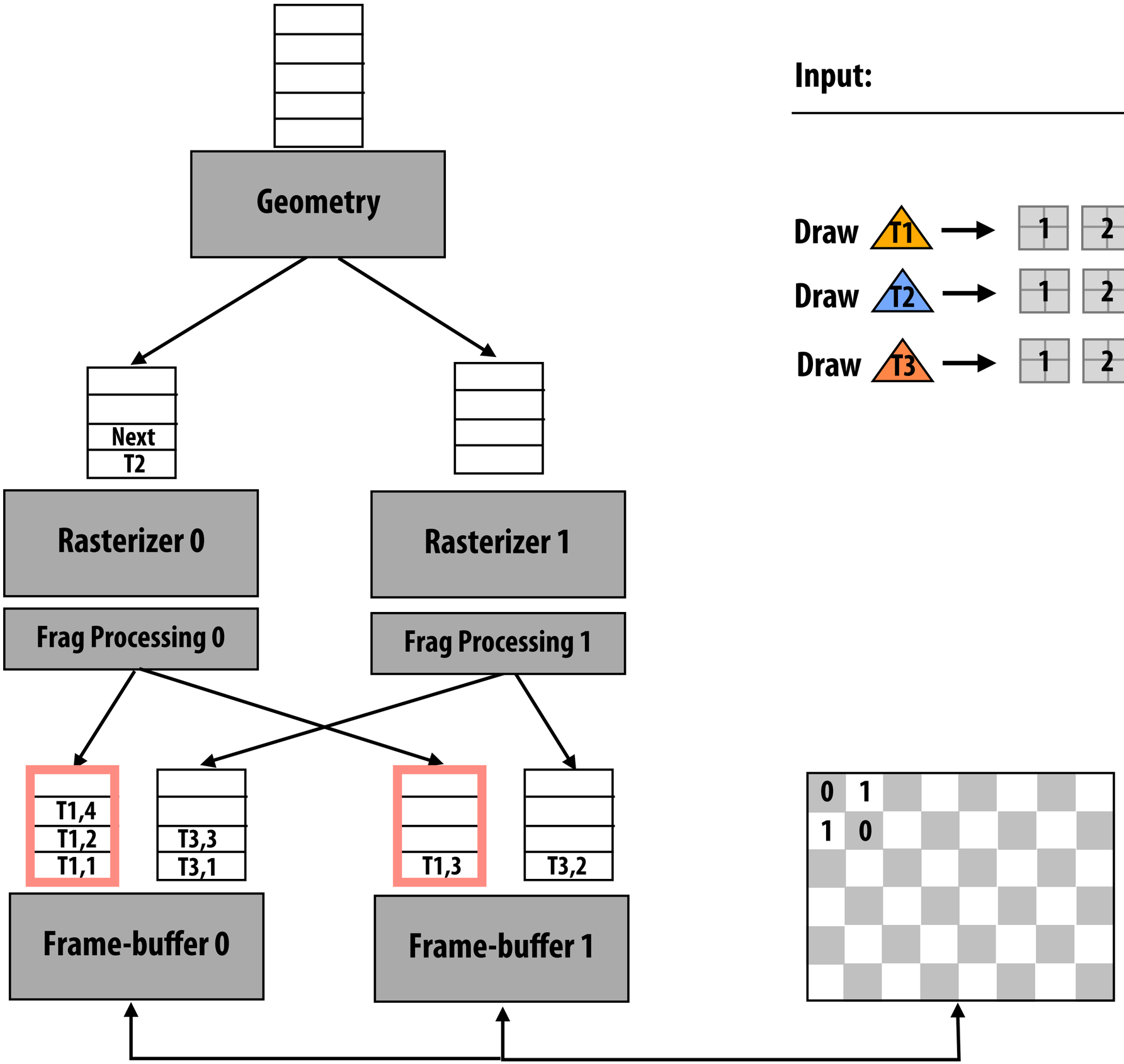
Draw T3 → 1 2 3



Interleaved render target

Rast 0 and rast 1 can process T1 and T3 simultaneously

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



Input:

Draw T1 → 1 2 3 4

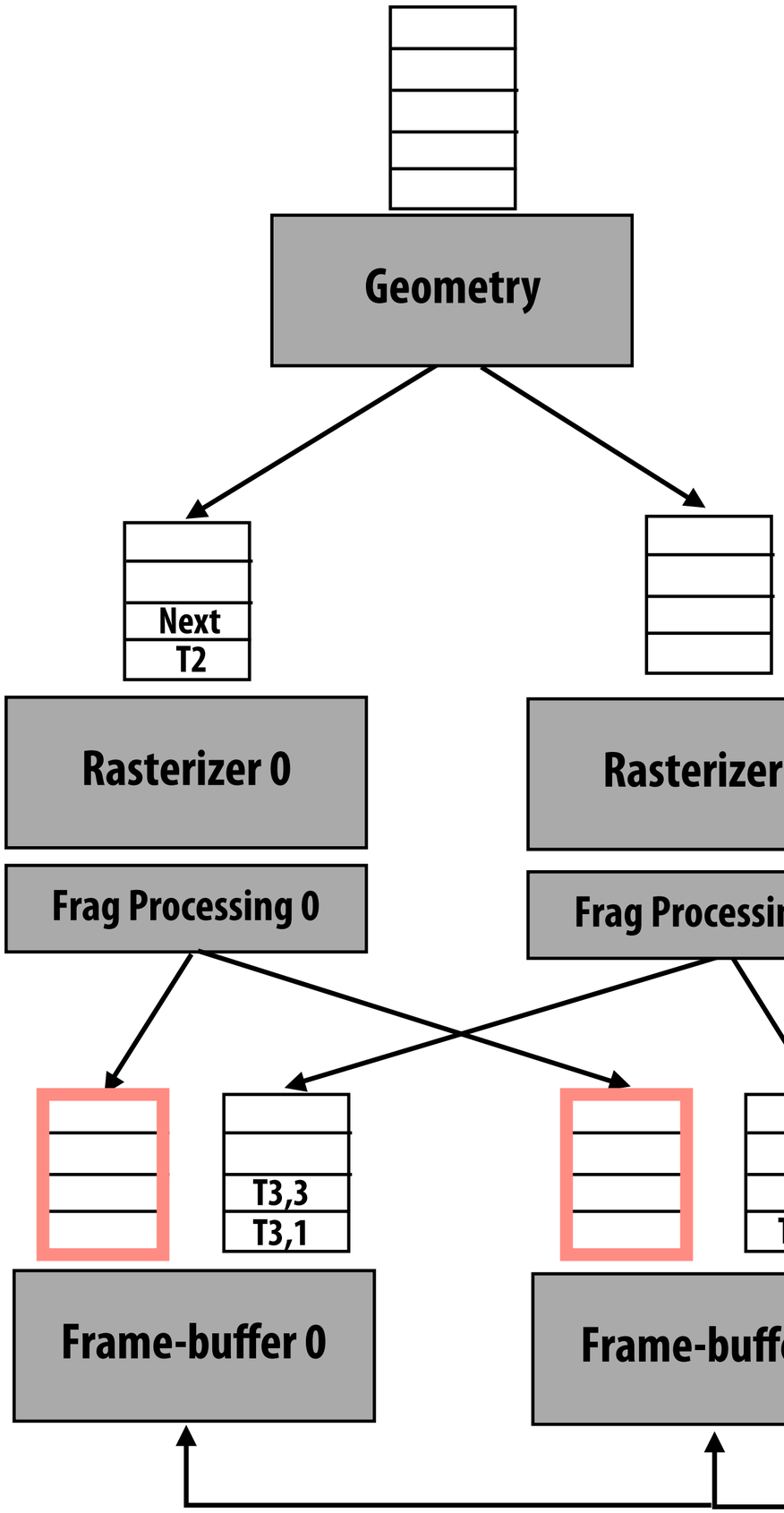
Draw T2 → 1 2 3 4

Draw T3 → 1 2 3

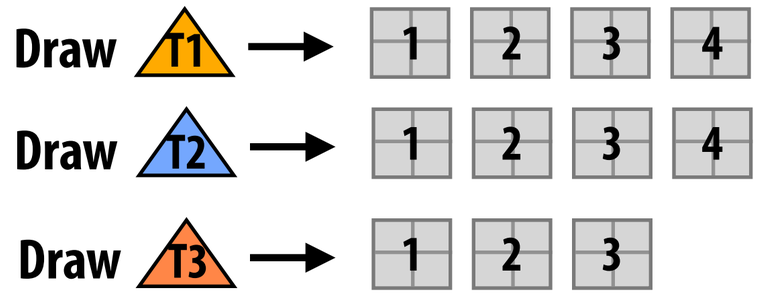
Interleaved render target

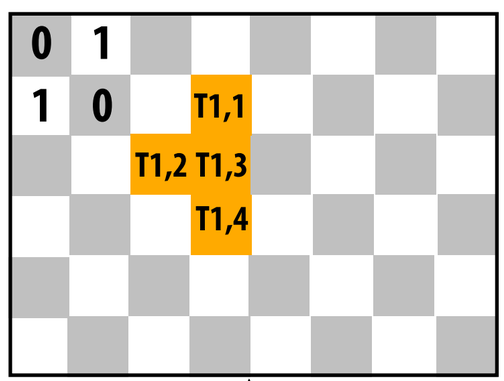
FB 0 and FB 1 can simultaneously process fragments from rast 0

(Notice updates to frame buffer)



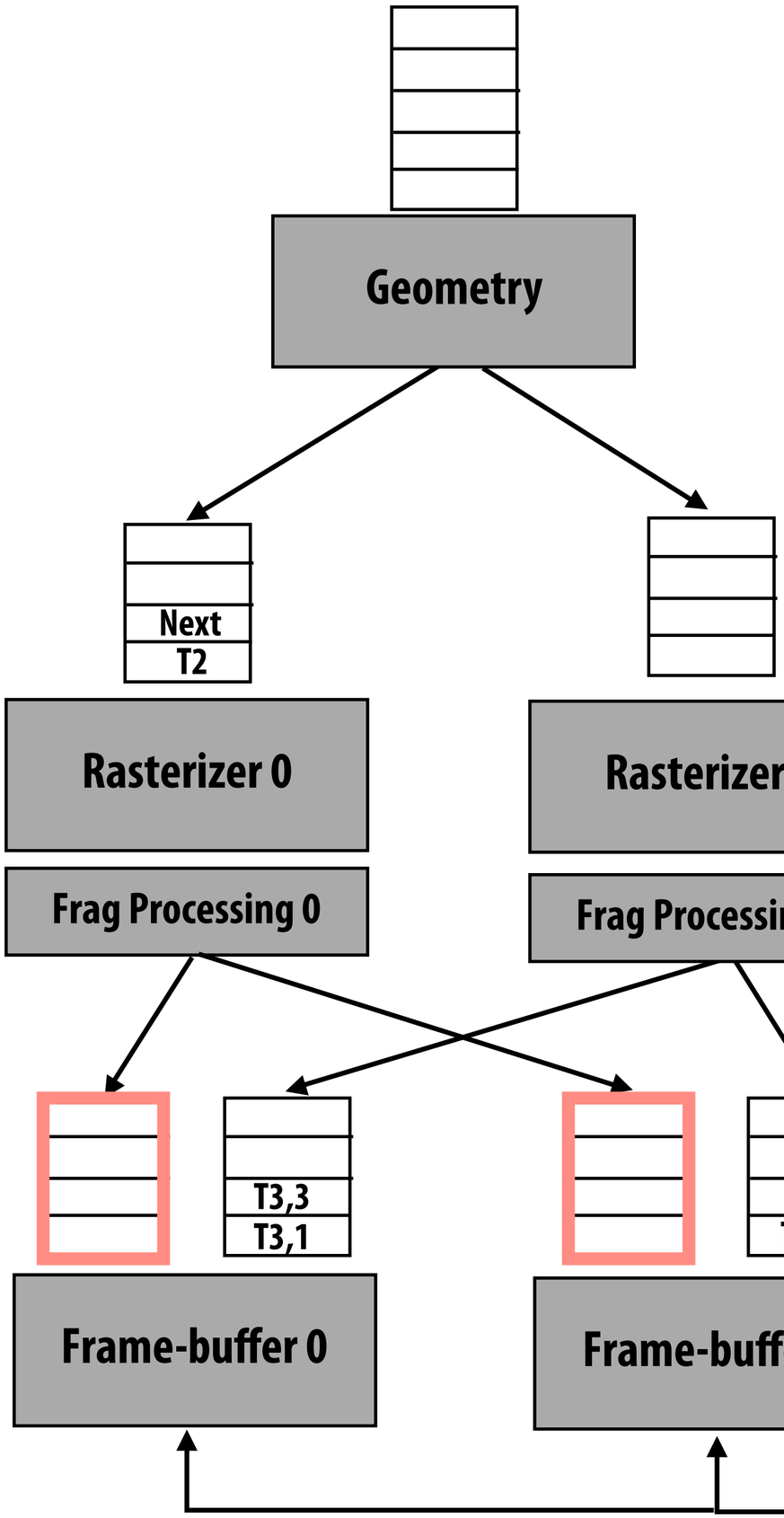
Input:



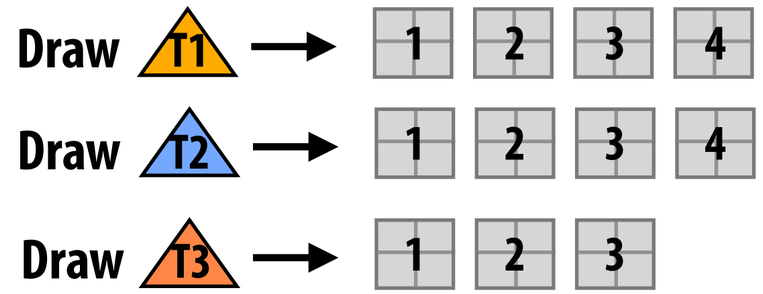


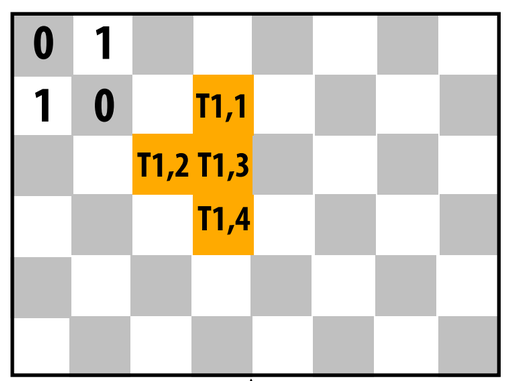
Interleaved render target

Fragments from T3 cannot be processed yet. Why?



Input:

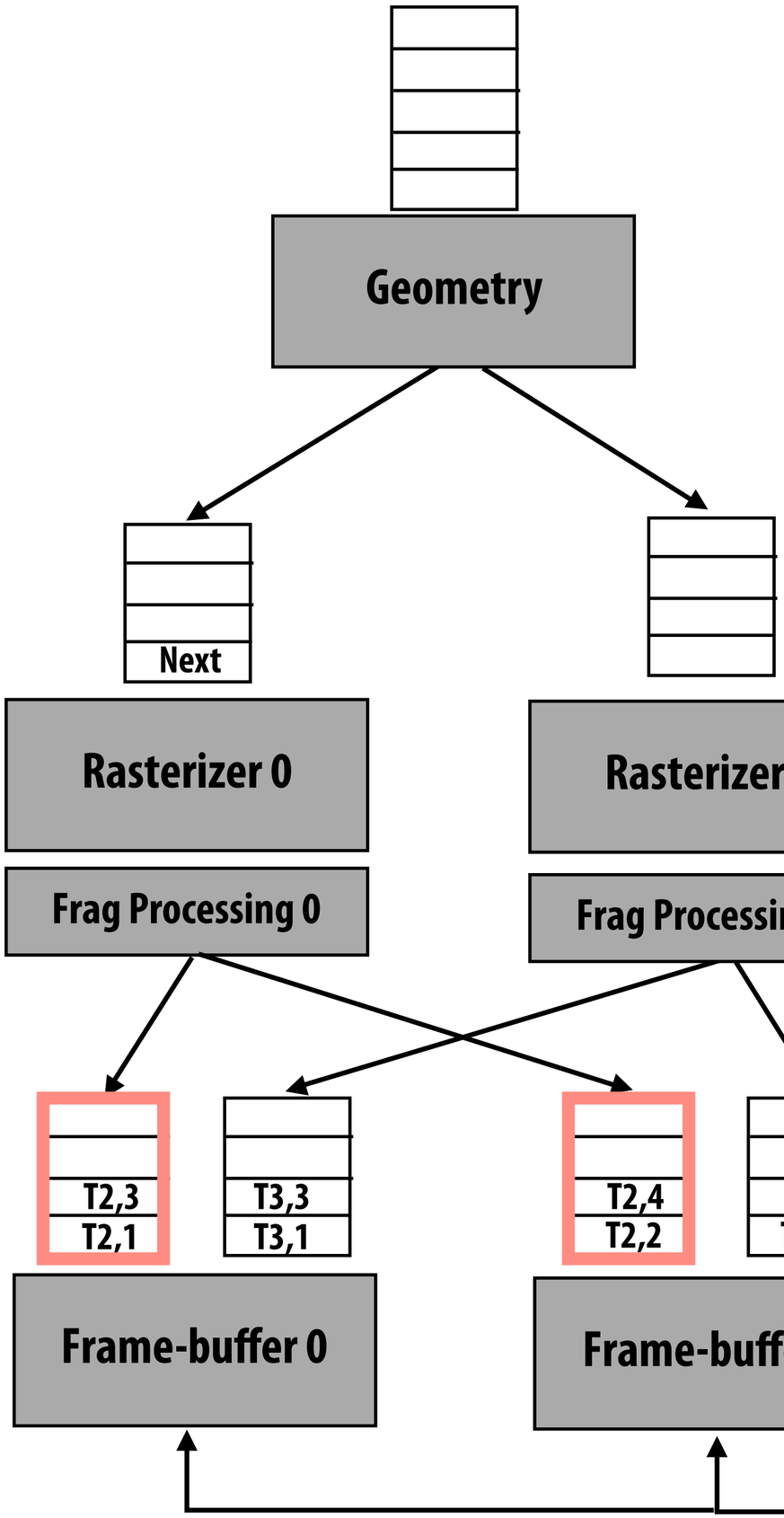




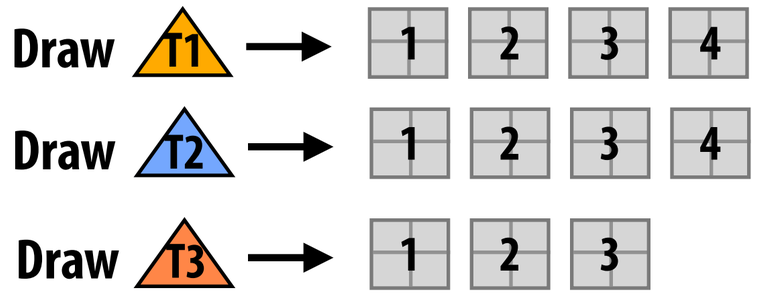
Interleaved render target

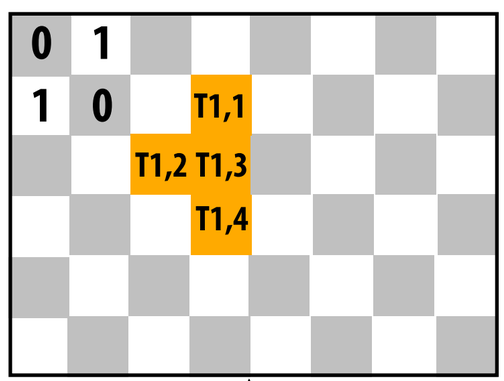
Rast 0 processes T2

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



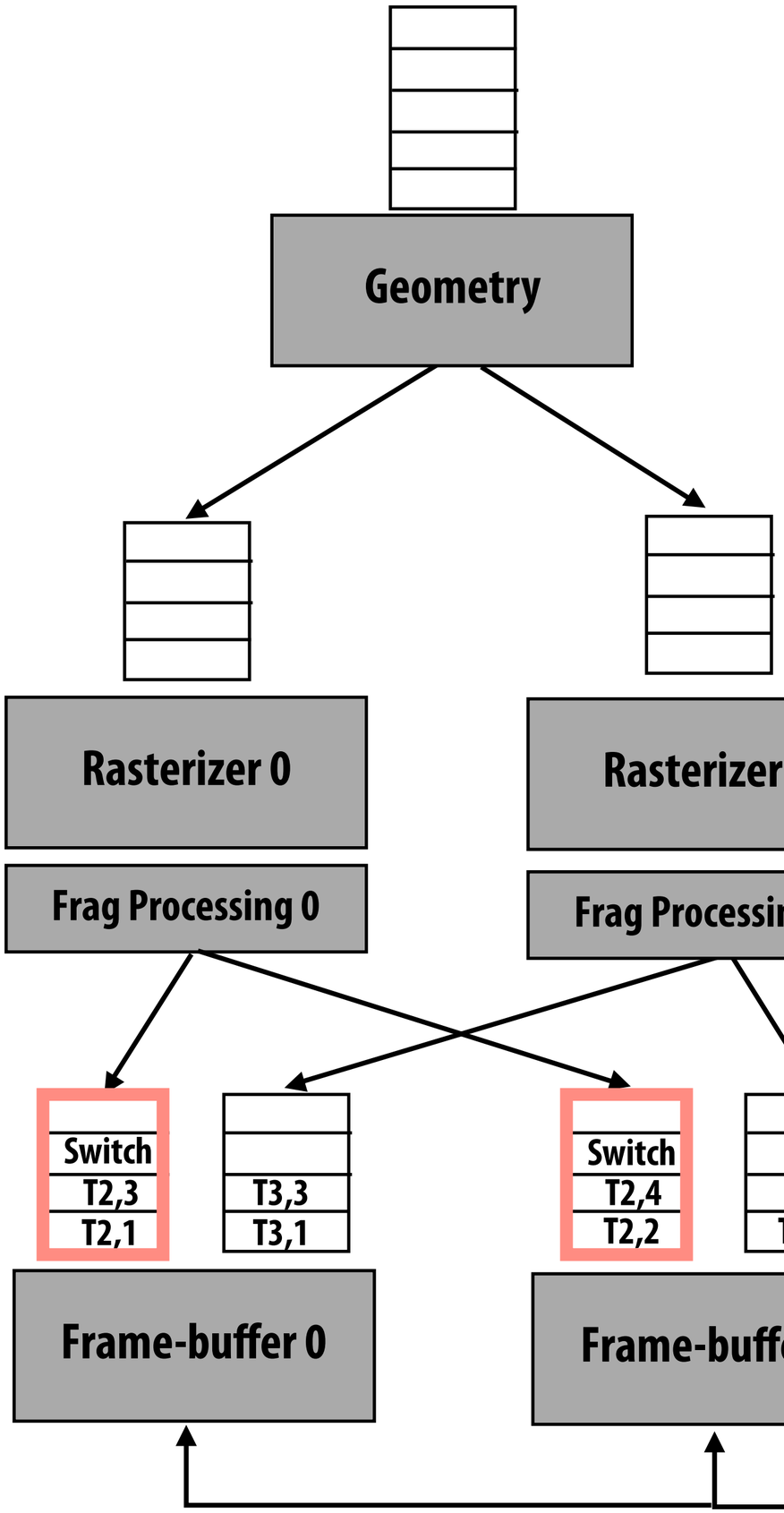
Input:



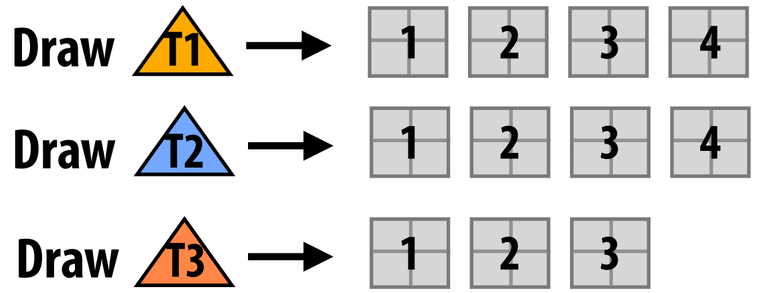


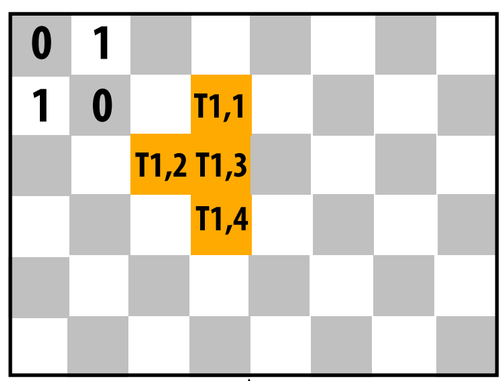
Interleaved render target

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



Input:

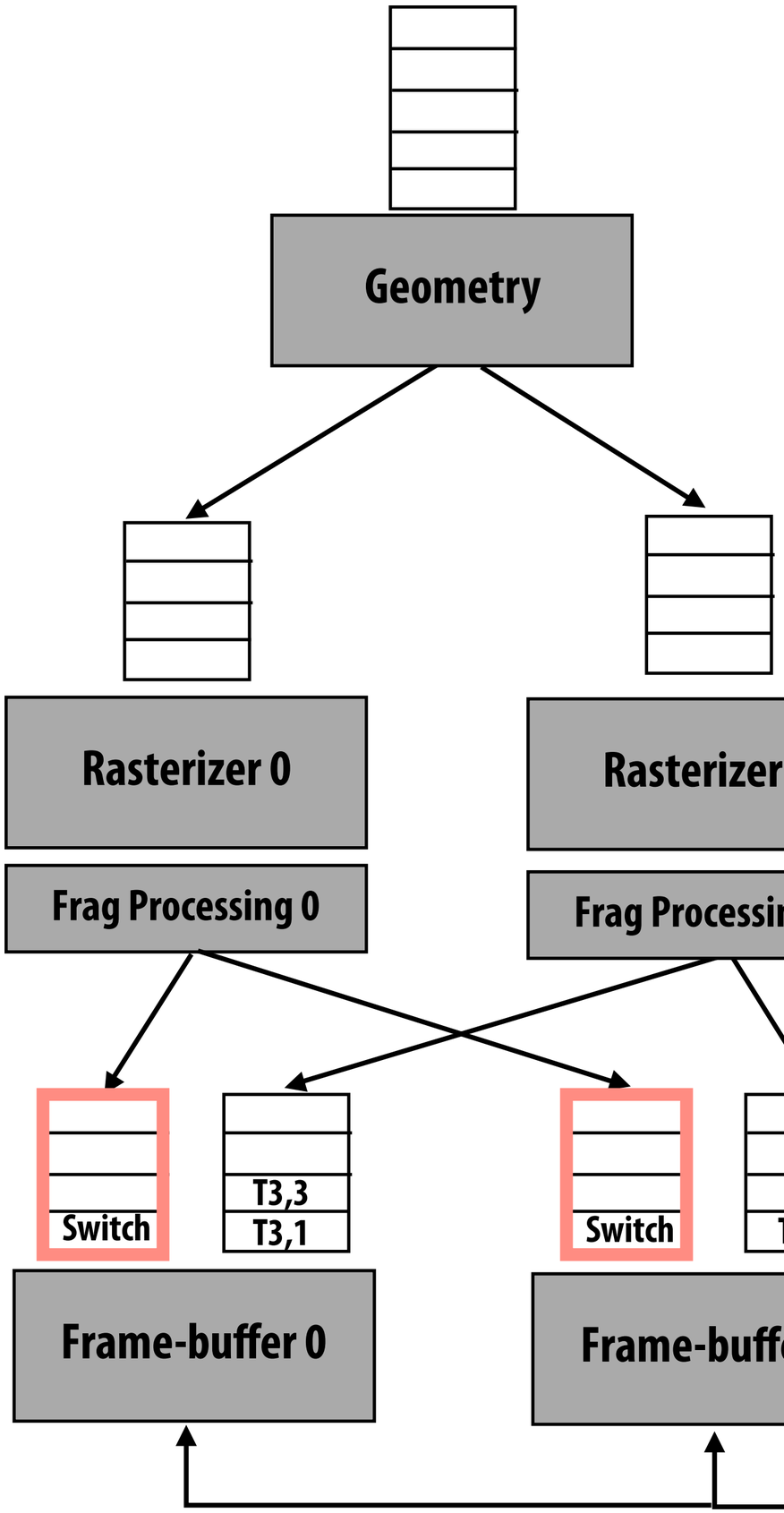




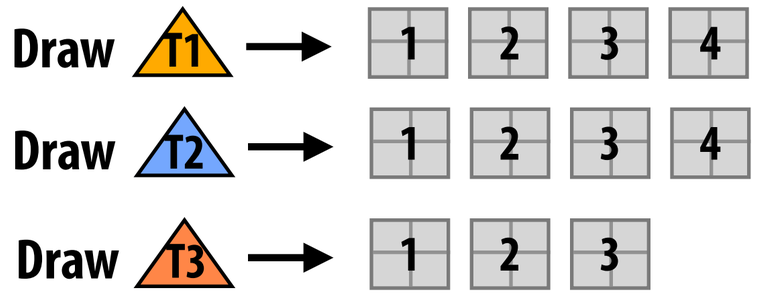
Interleaved render target

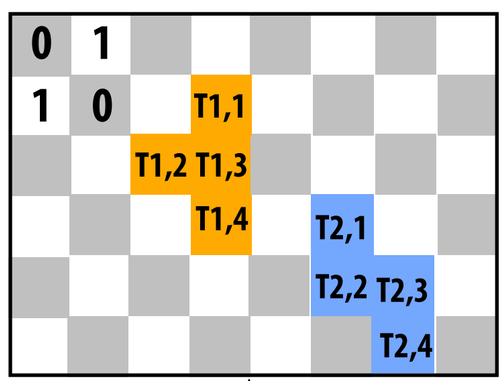
FB 0 and FB 1 can simultaneously process fragments from rast 0

(Notice updates to frame buffer)



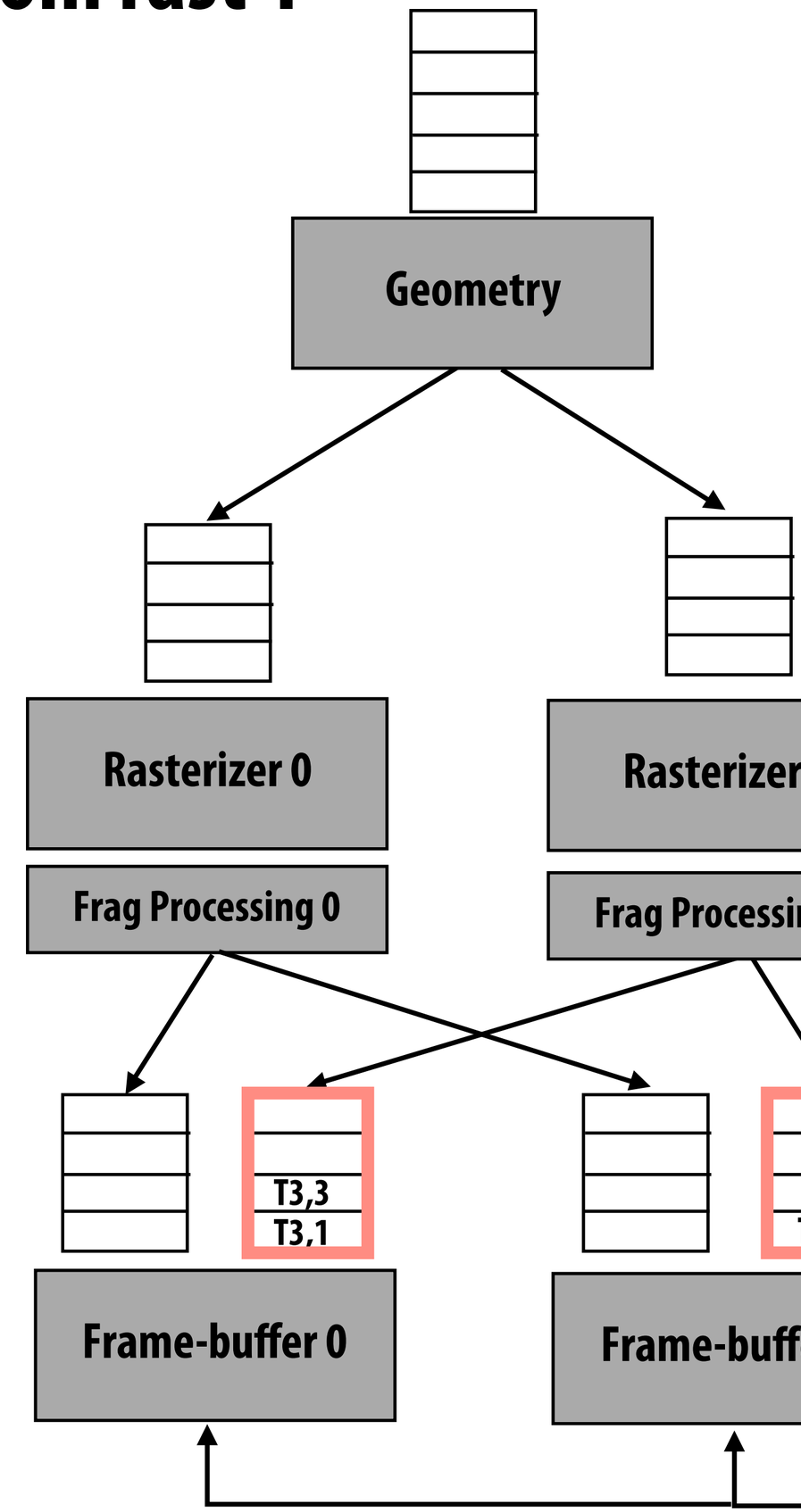
Input:



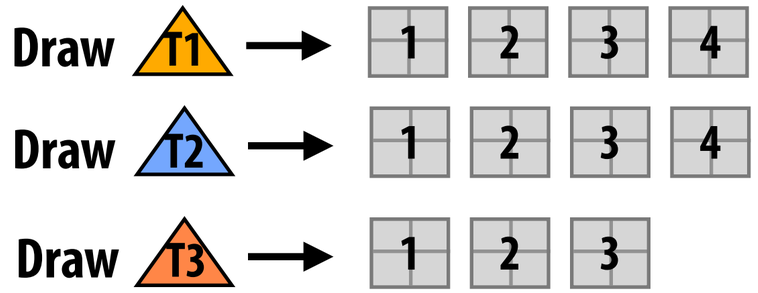


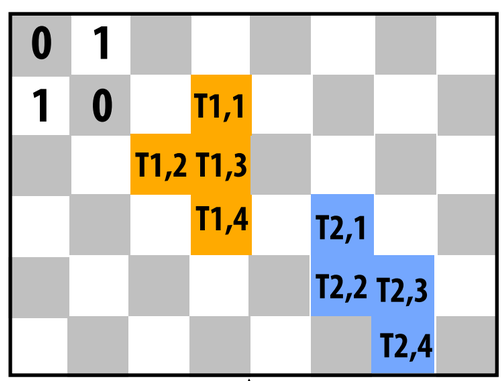
Interleaved render target

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



Input:

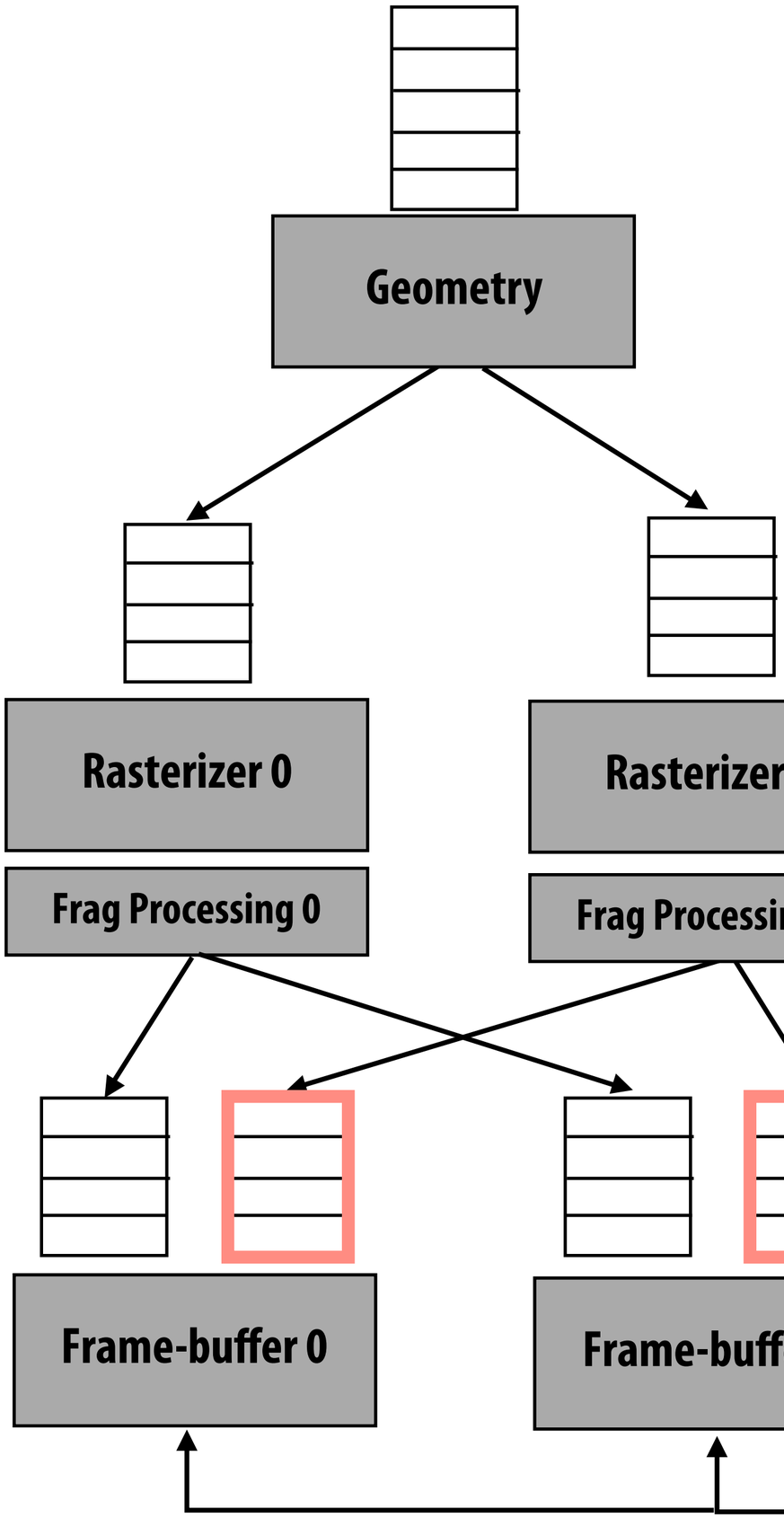




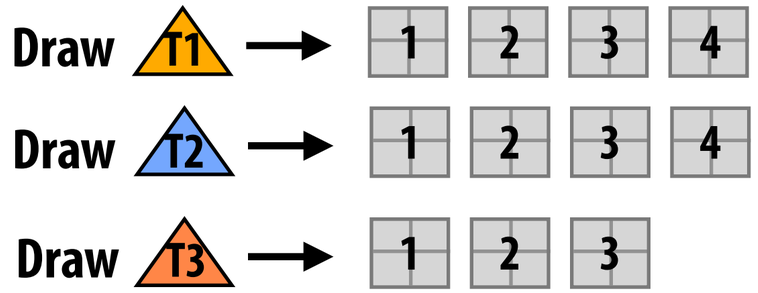
Interleaved render target

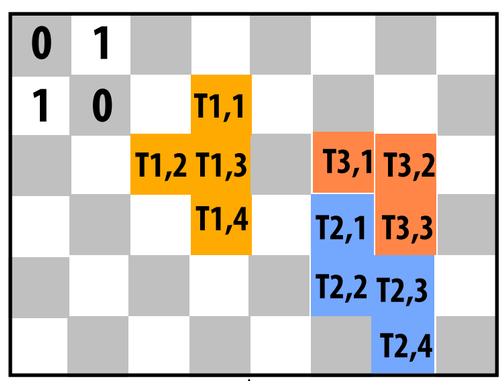
FB 0 and FB 1 can simultaneously process fragments from rast 1

(Notice updates to frame buffer)



Input:

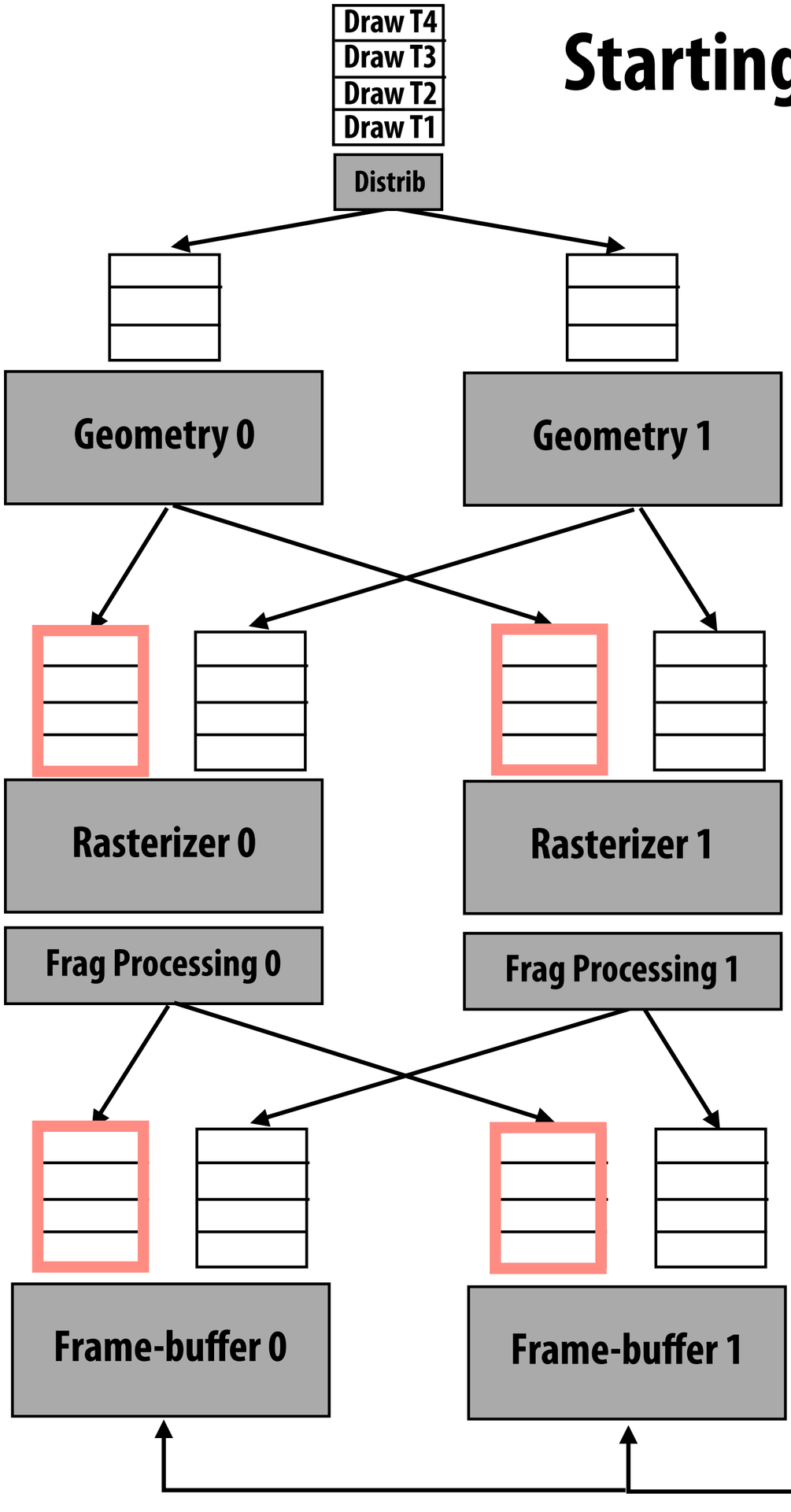




Interleaved render target

Extending to parallel geometry units

Starting state: commands enqueued



Input:

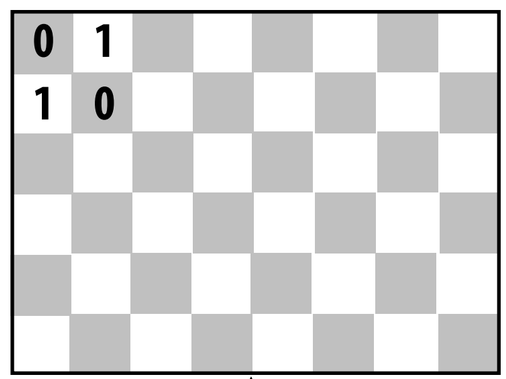
Draw T1 → 1 2 3 4 5 6 7

Draw T2 → 1 2 3 4

Draw T3 → 1 2 3 4 5

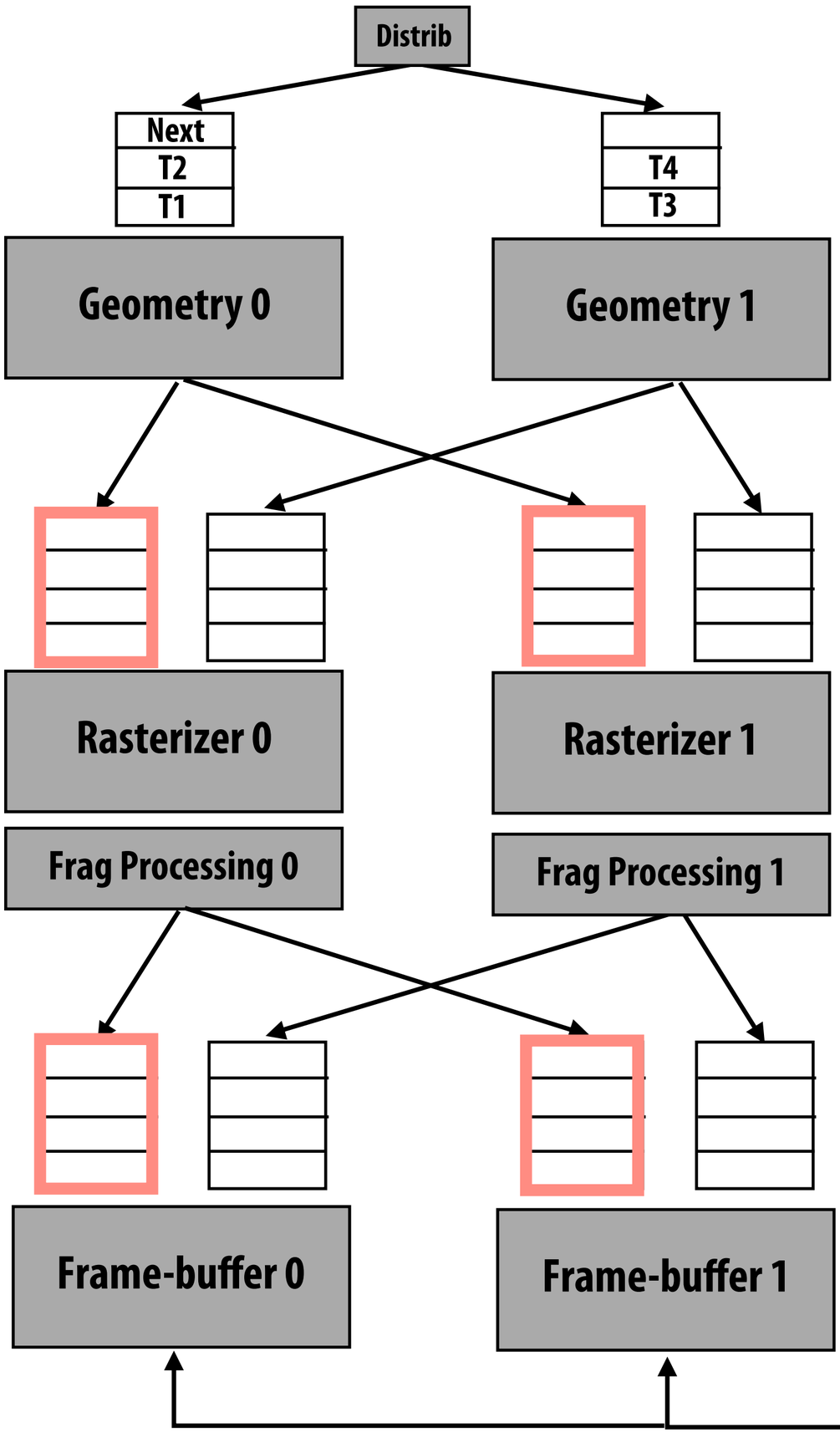
Draw T4 → 1 2

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)



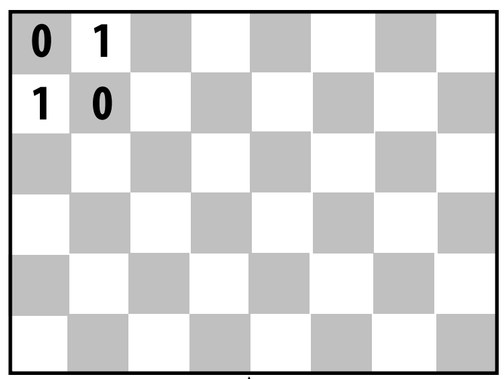
Input:

Draw T1 → 1 2 3 4 5 6 7

Draw T2 → 1 2 3 4

Draw T3 → 1 2 3 4 5

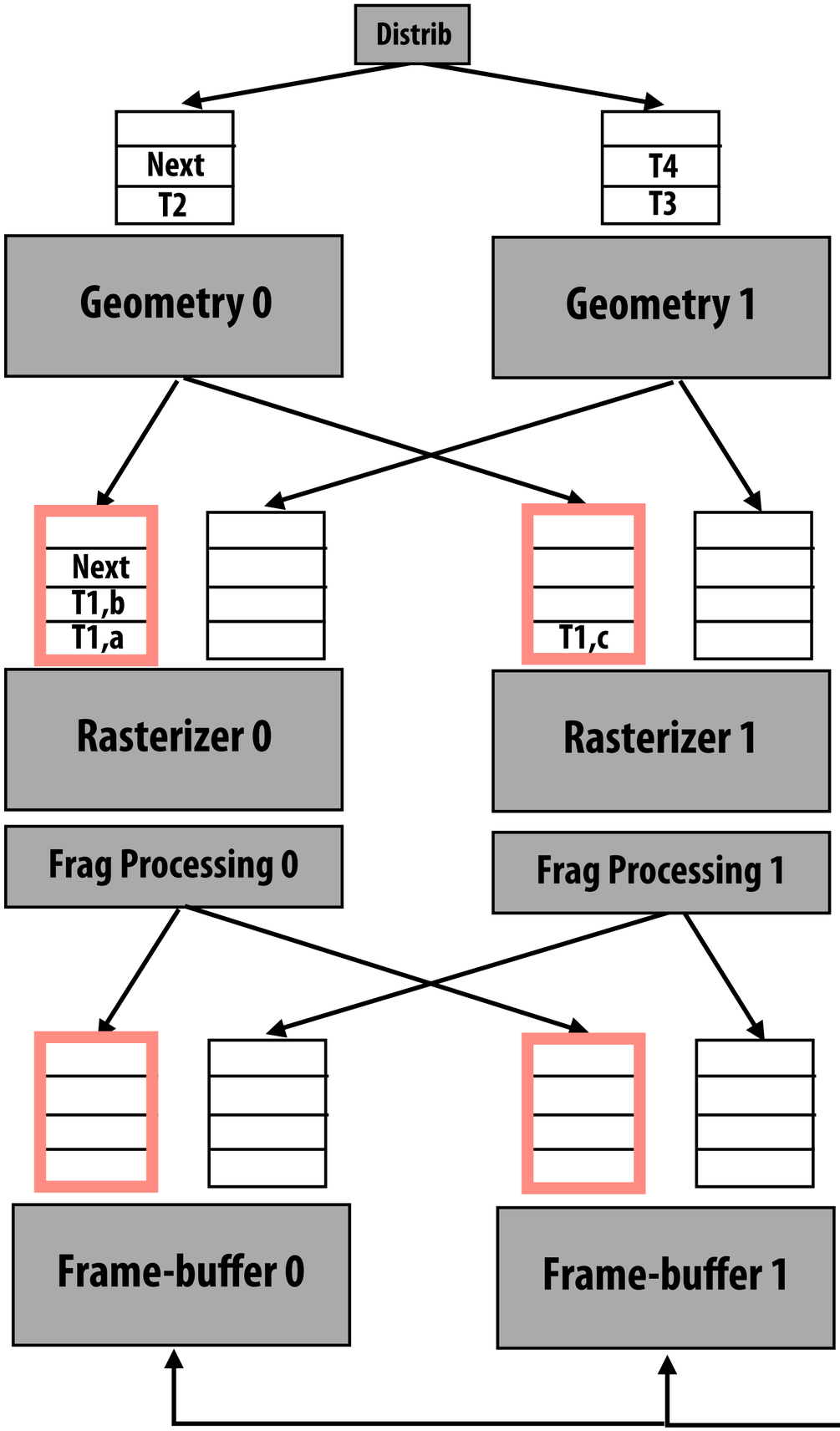
Draw T4 → 1 2



Interleaved render target

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.])



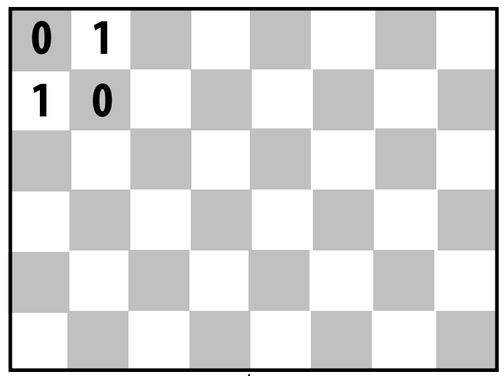
Input:

Draw T1 → 1 2 3 4 5 6 7

Draw T2 → 1 2 3 4

Draw T3 → 1 2 3 4 5

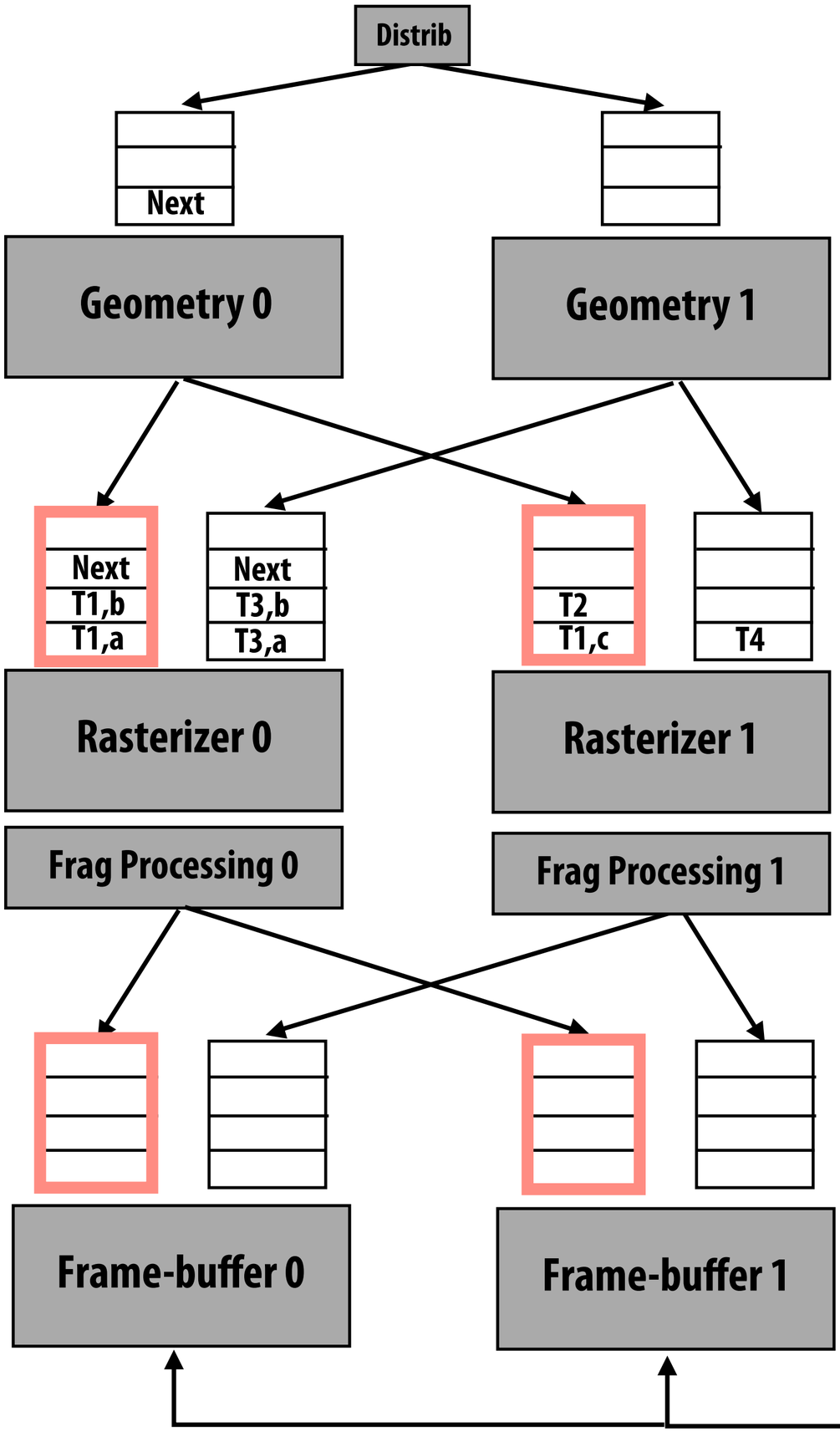
Draw T4 → 1 2



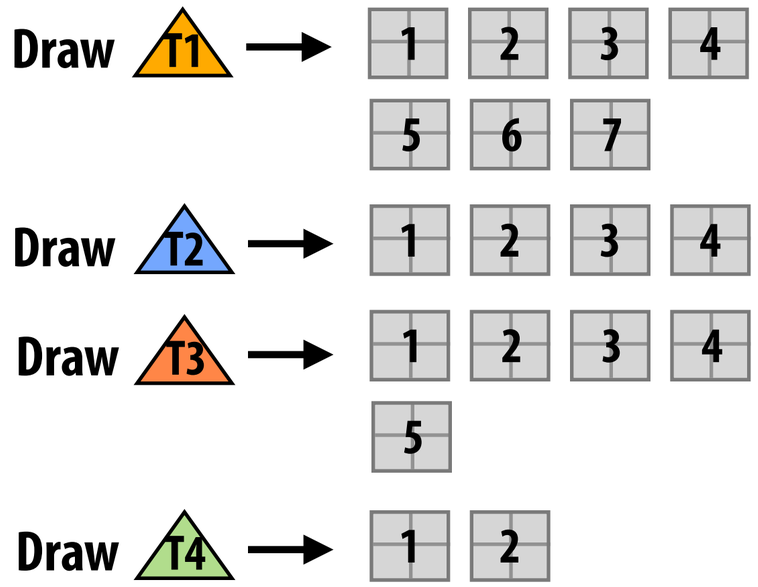
Interleaved render target

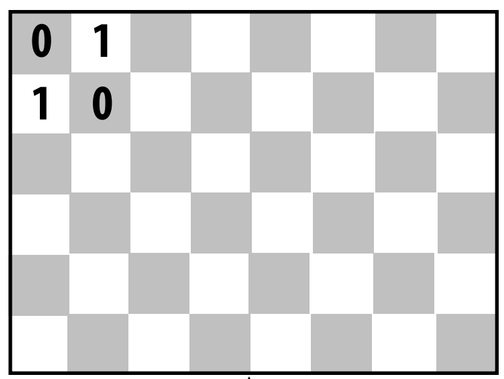
Geom 0 and geom 1 process triangles in parallel

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



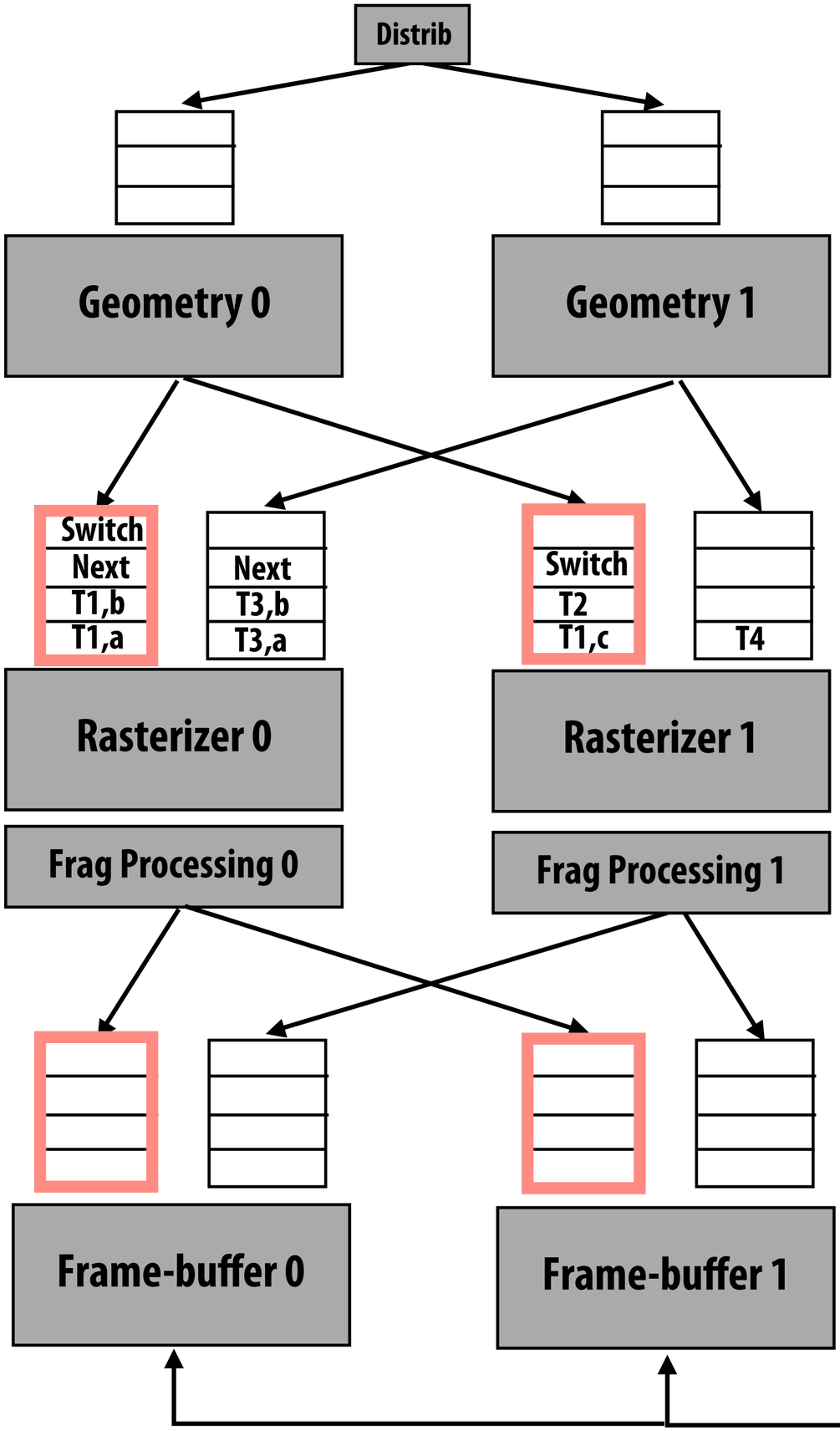
Input:





Interleaved render target

Geom 0 broadcasts 'next' token to rasterizers



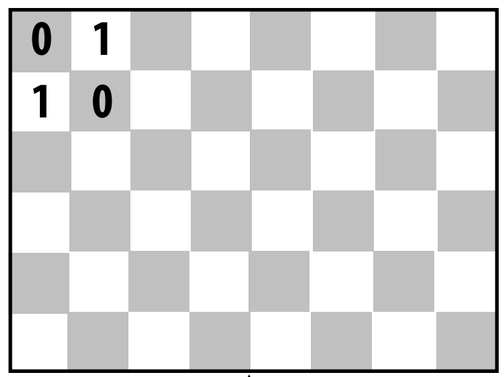
Input:

Draw T1 → 1 2 3 4 5 6 7

Draw T2 → 1 2 3 4

Draw T3 → 1 2 3 4 5

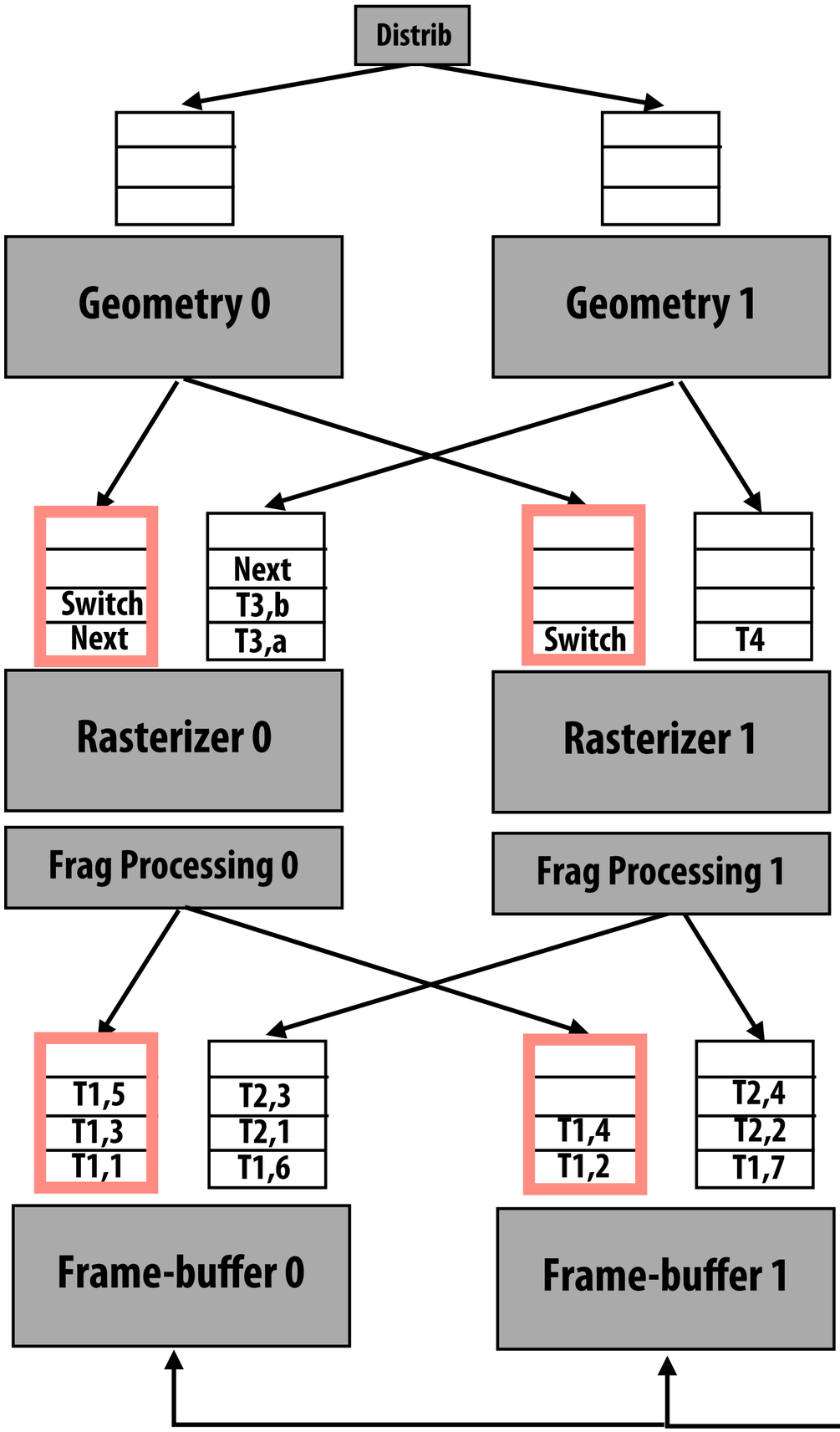
Draw T4 → 1 2



Interleaved render target

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

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



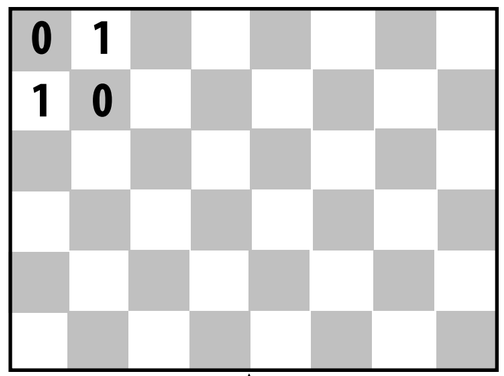
Input:

Draw T1 → 1 2 3 4 5 6 7

Draw T2 → 1 2 3 4

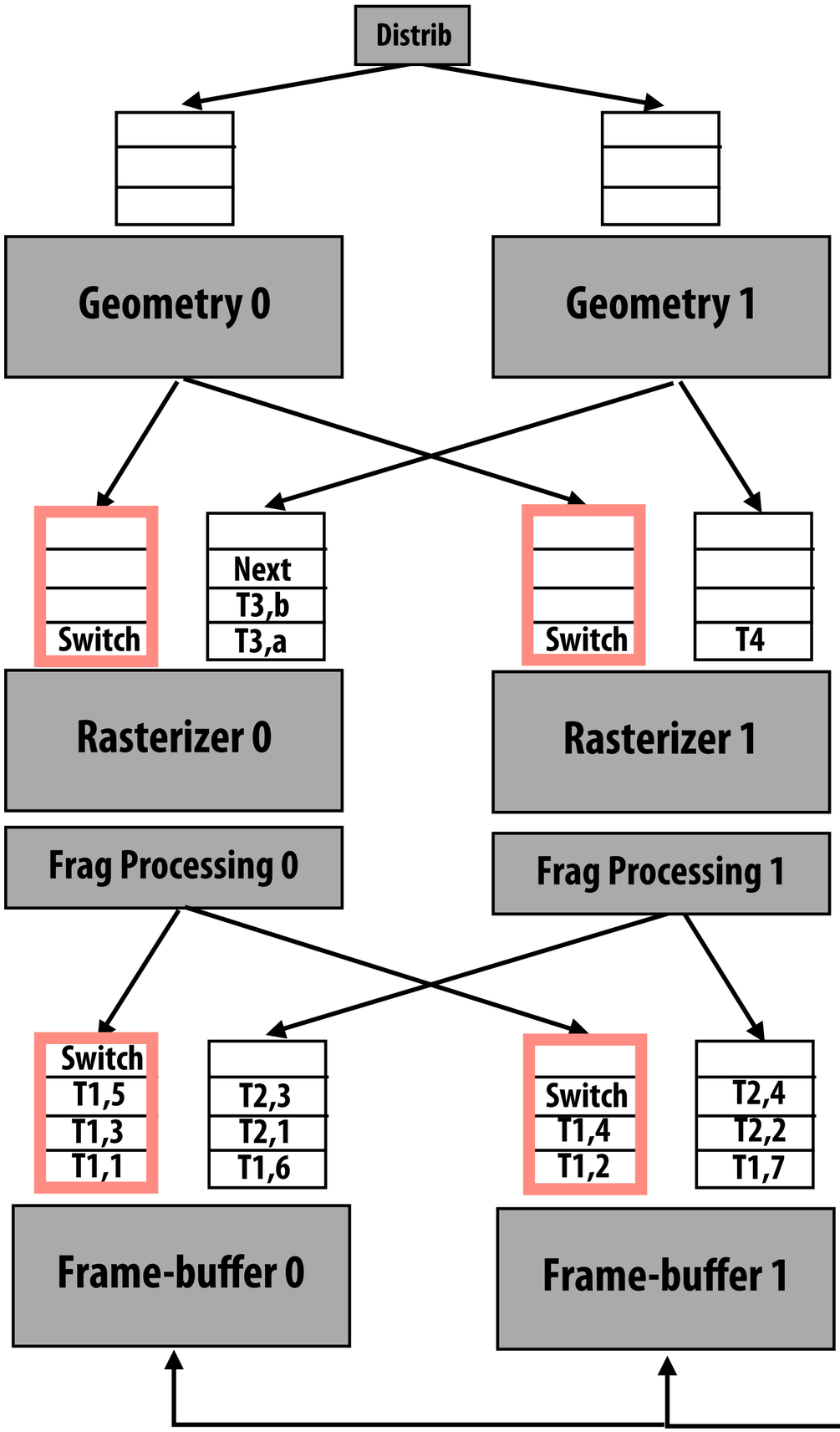
Draw T3 → 1 2 3 4 5

Draw T4 → 1 2



Interleaved render target

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



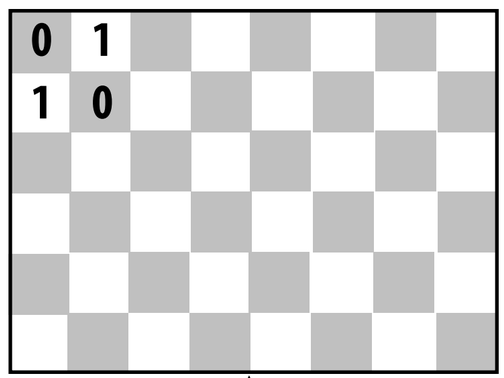
Input:

Draw T1 → 1 2 3 4 5 6 7

Draw T2 → 1 2 3 4

Draw T3 → 1 2 3 4 5

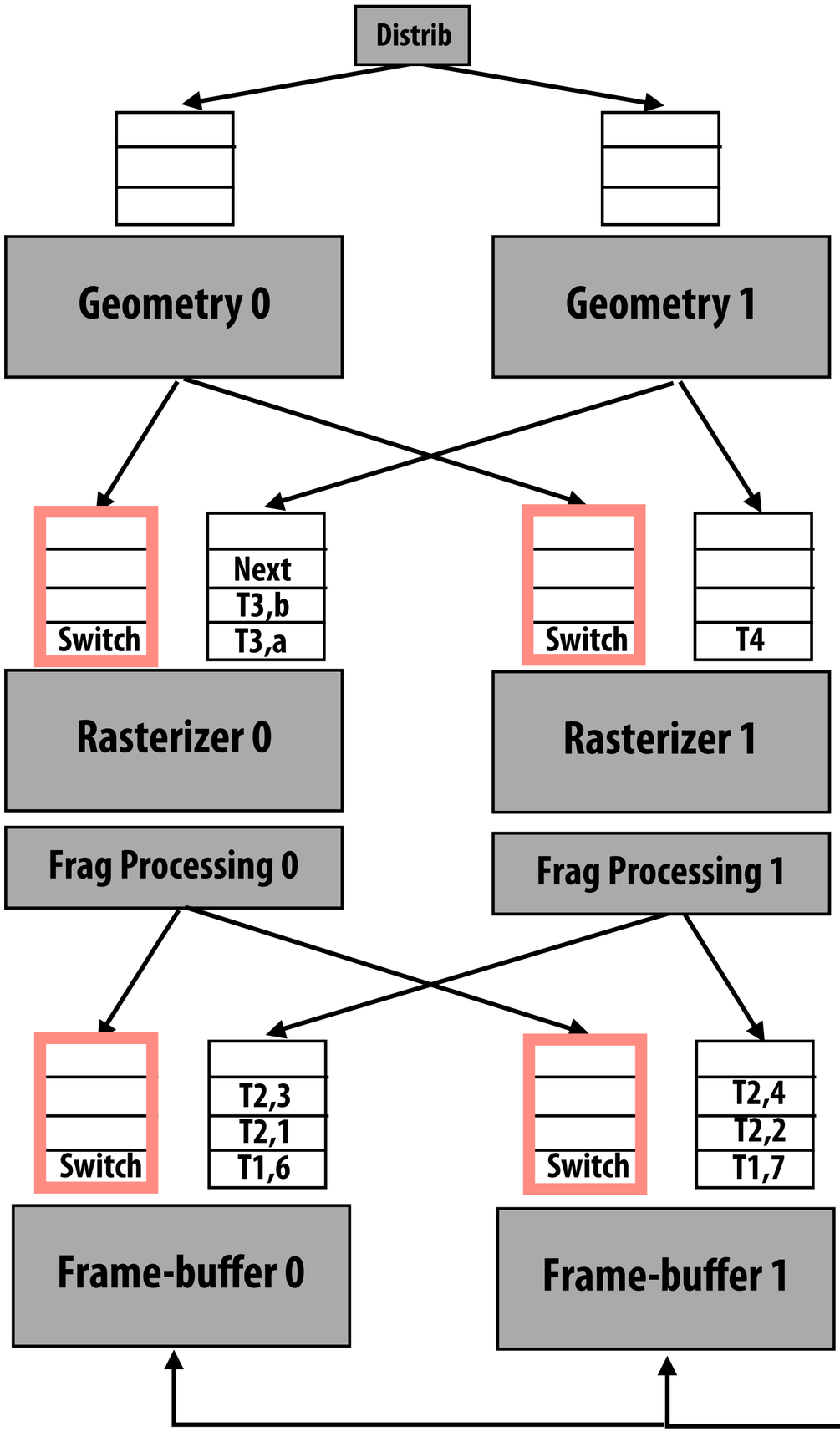
Draw T4 → 1 2



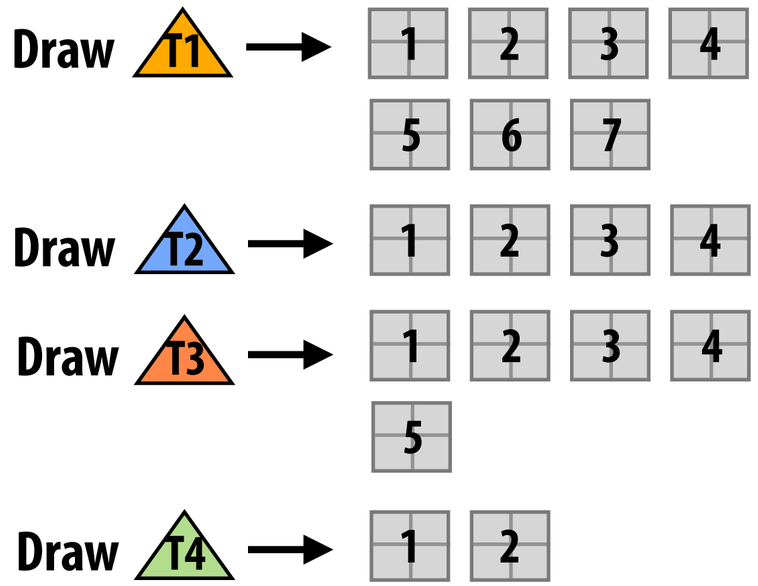
Interleaved render target

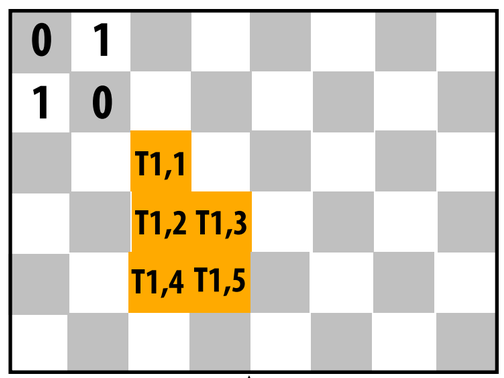
Frame-buffer units process frags from (geom 0, rast 0) in parallel

(Notice updates to frame buffer)



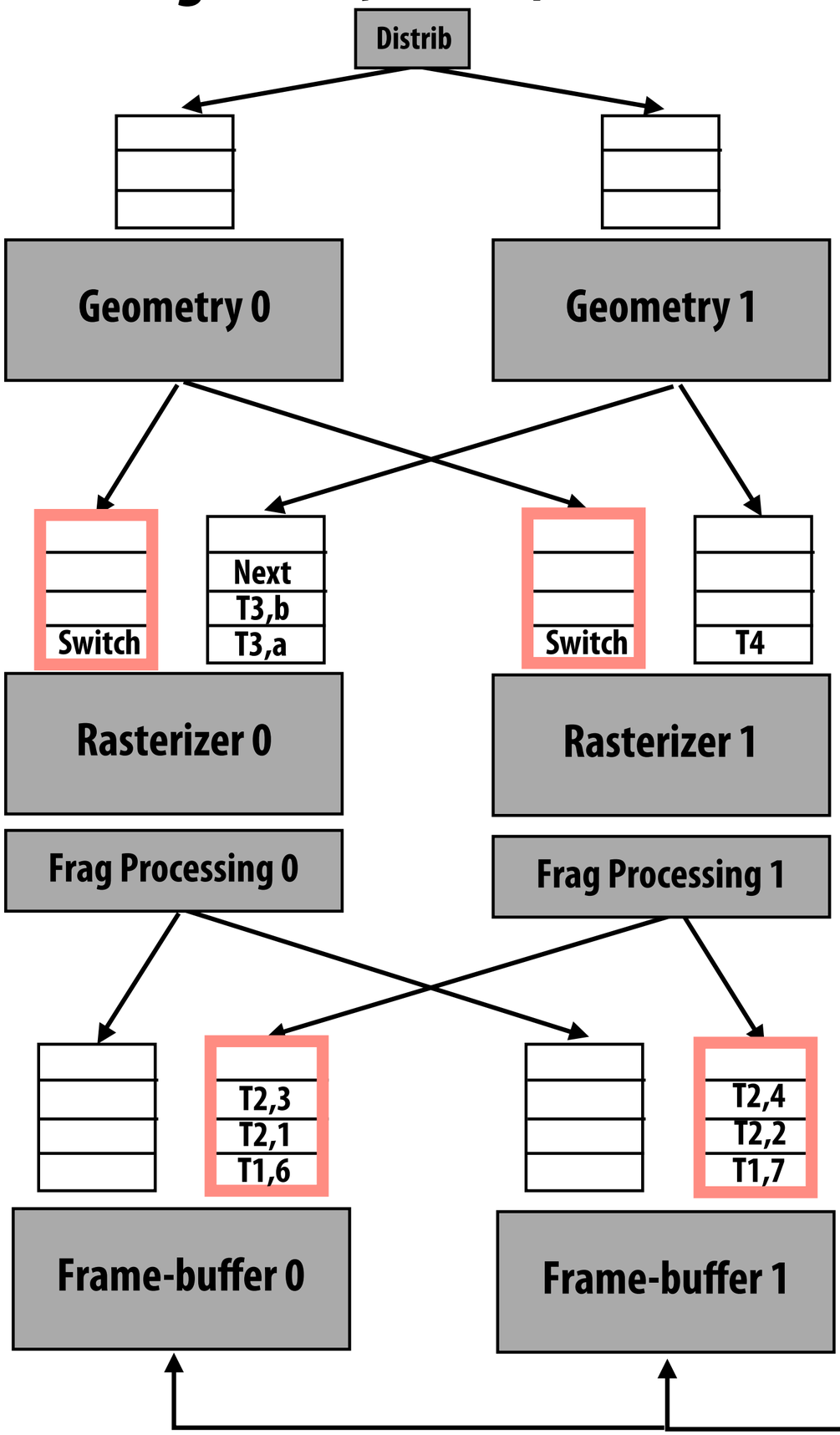
Input:



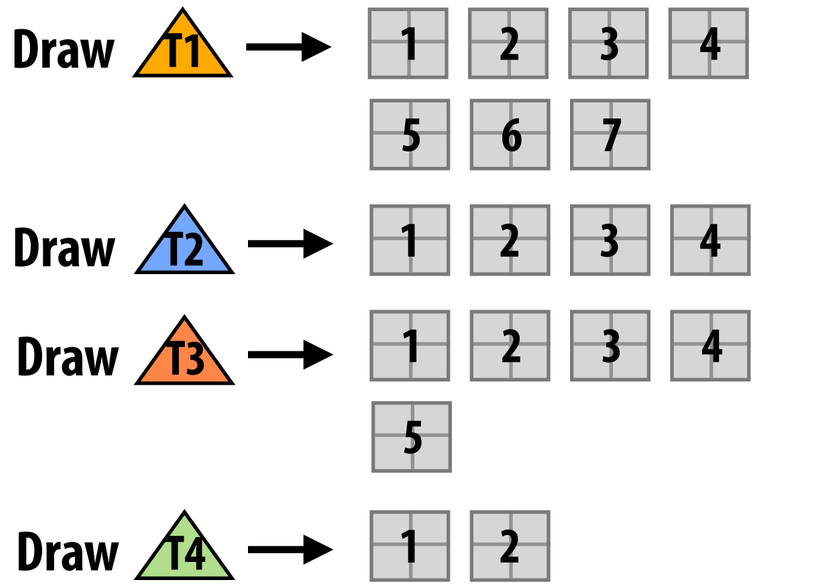


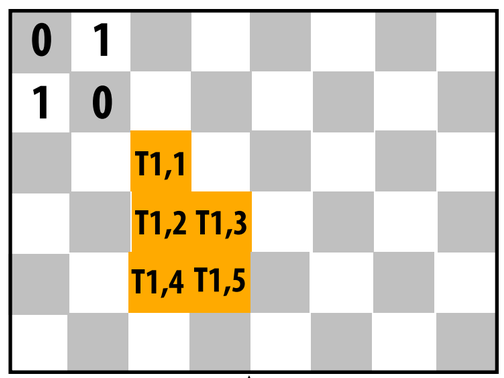
Interleaved render target

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



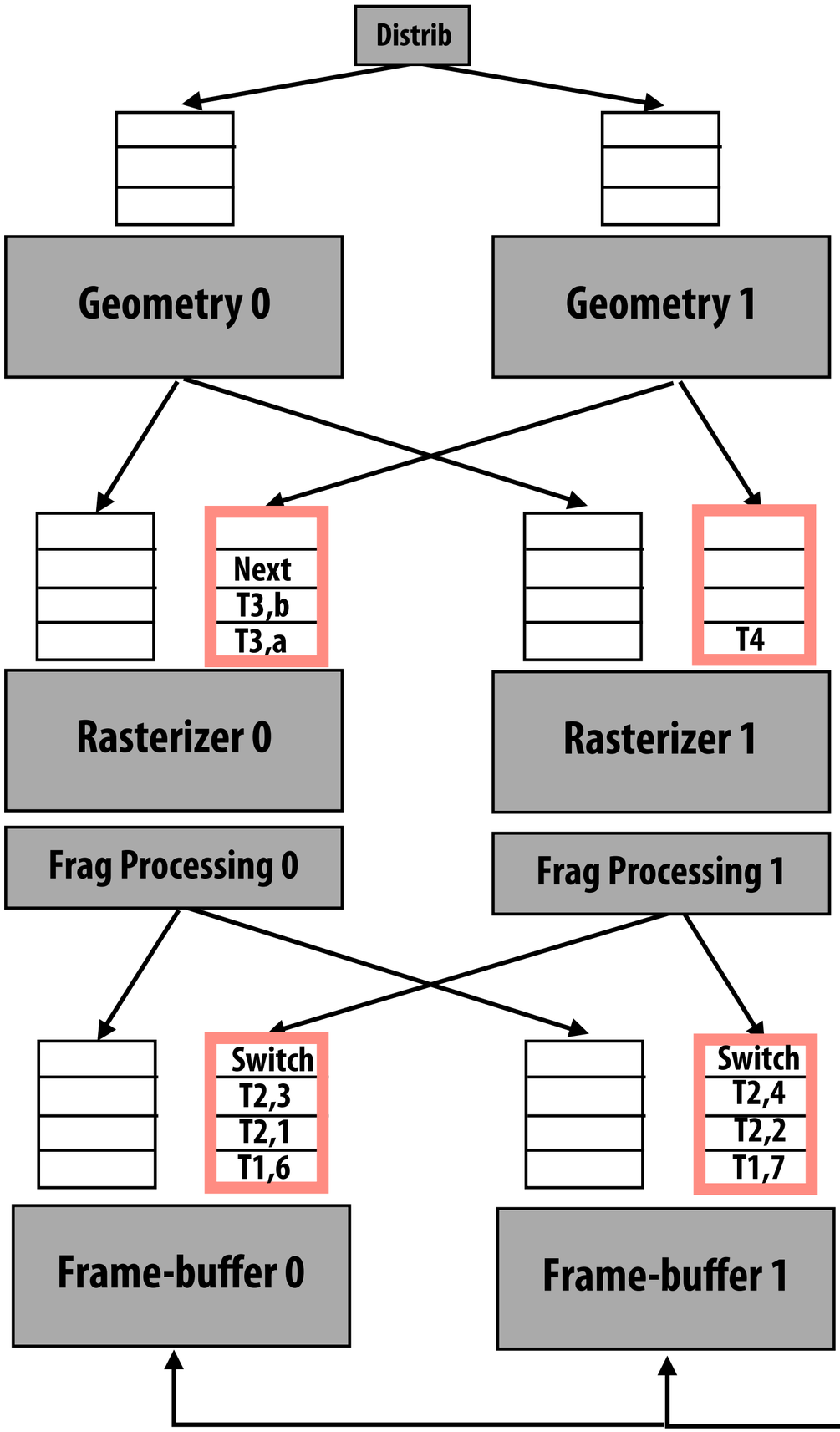
Input:

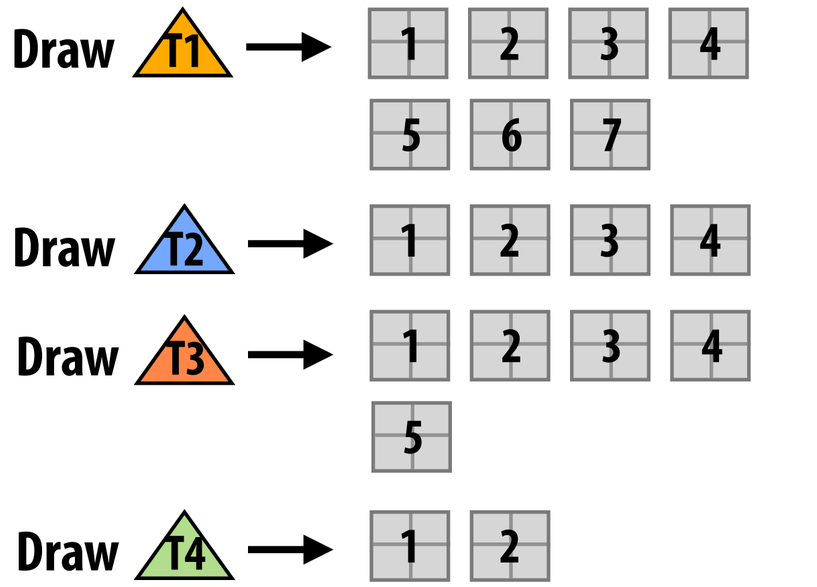


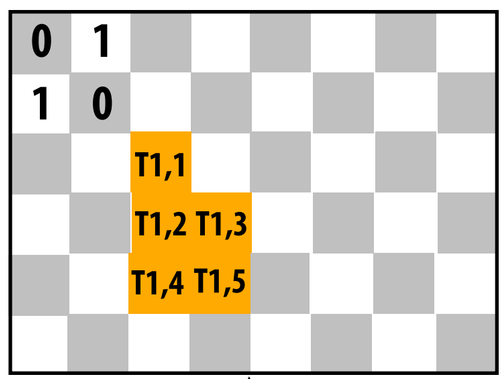


Interleaved render target

“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)



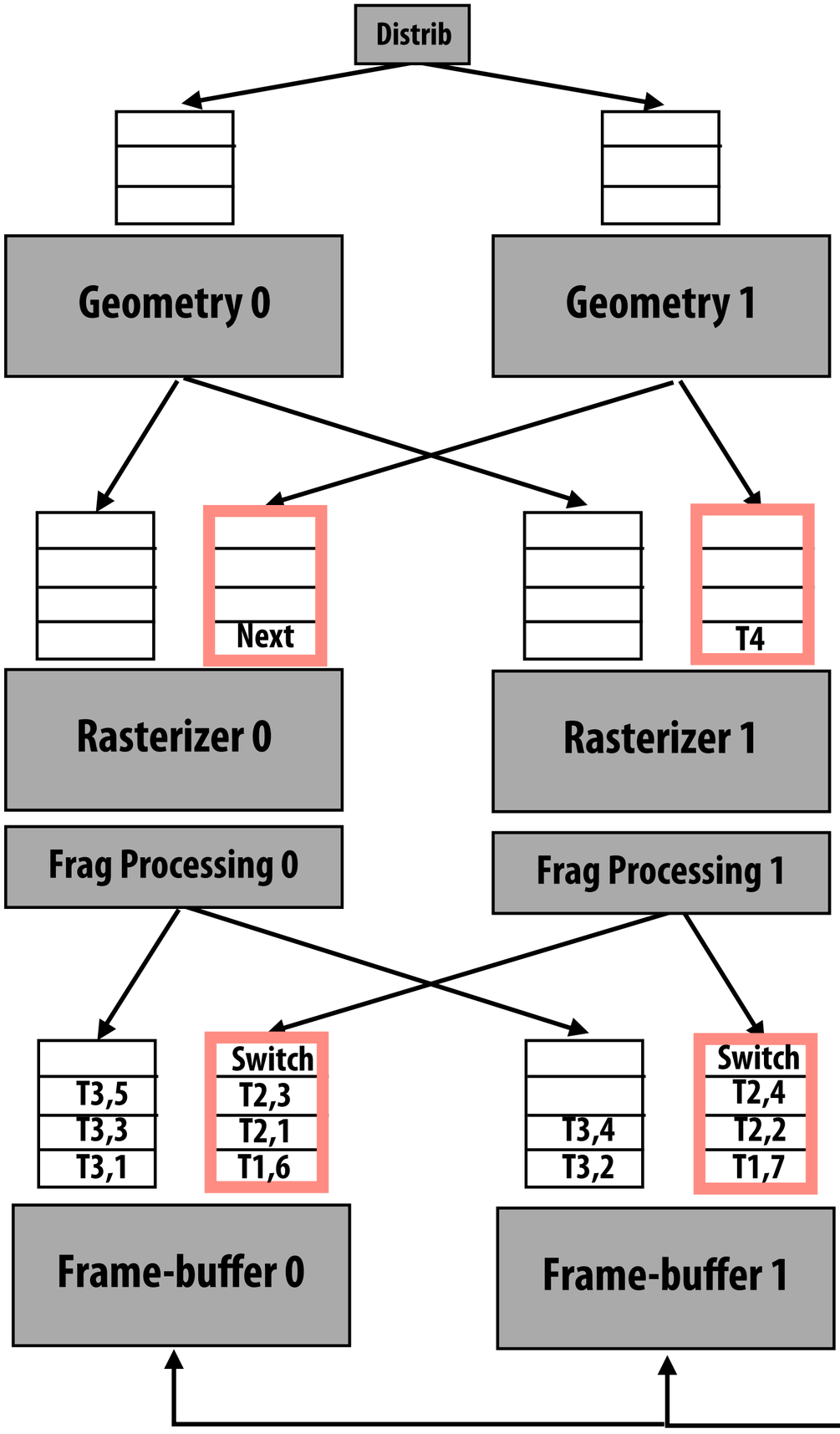




Interleaved render target

Rast 0 processes triangles from geom 1

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



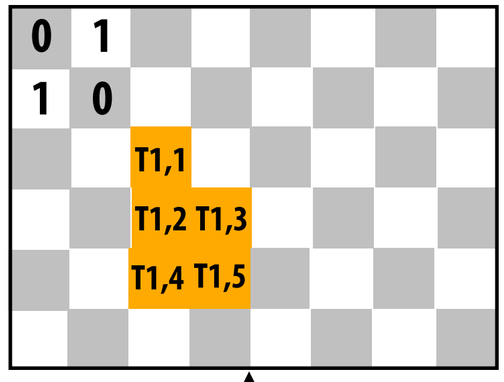
Input:

Draw T1 → 1 2 3 4 5 6 7

Draw T2 → 1 2 3 4

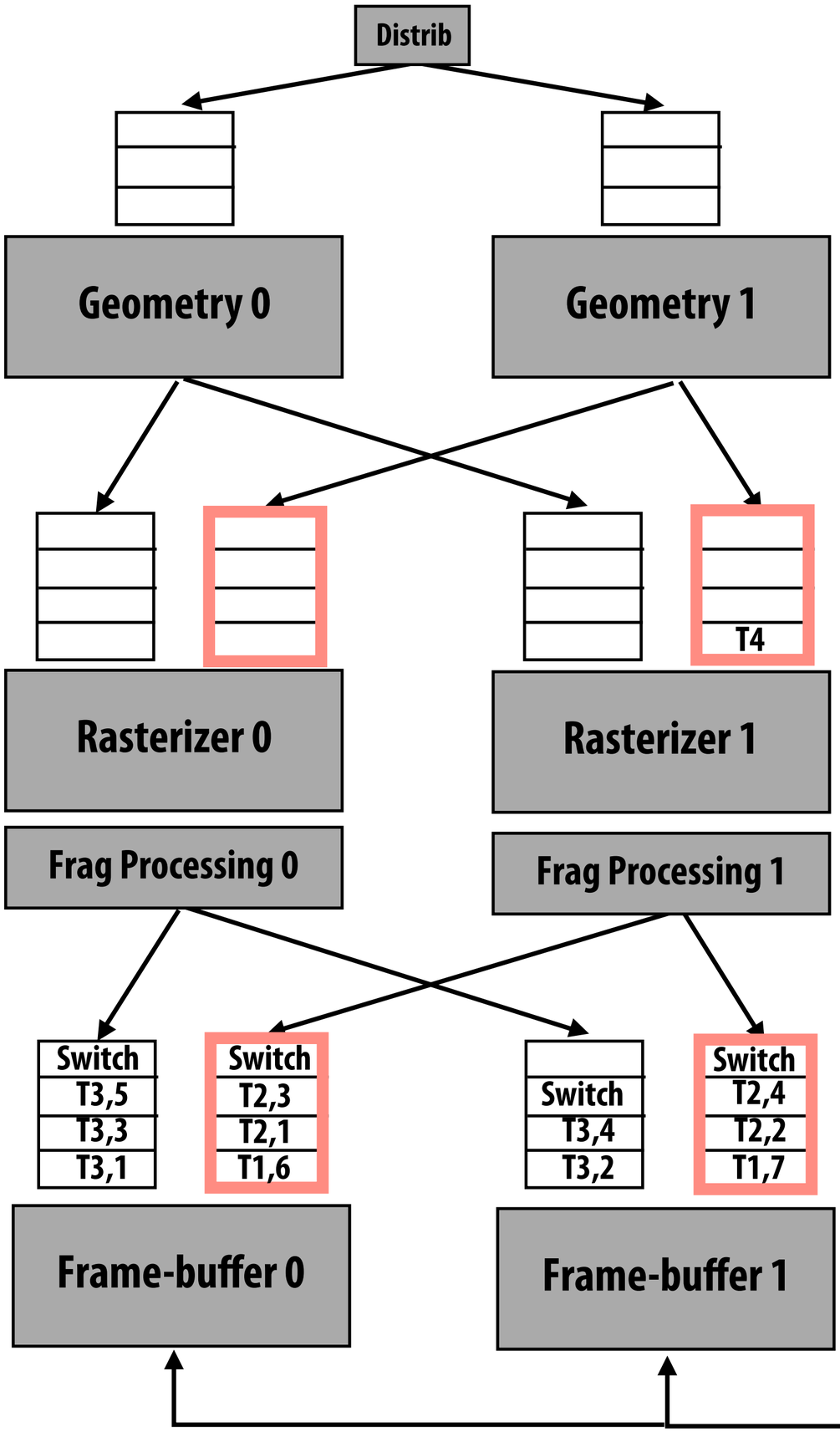
Draw T3 → 1 2 3 4 5

Draw T4 → 1 2



Interleaved render target

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



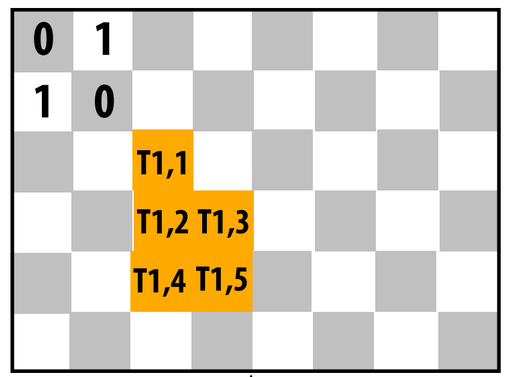
Input:

Draw T1 → 1 2 3 4 5 6 7

Draw T2 → 1 2 3 4

Draw T3 → 1 2 3 4 5

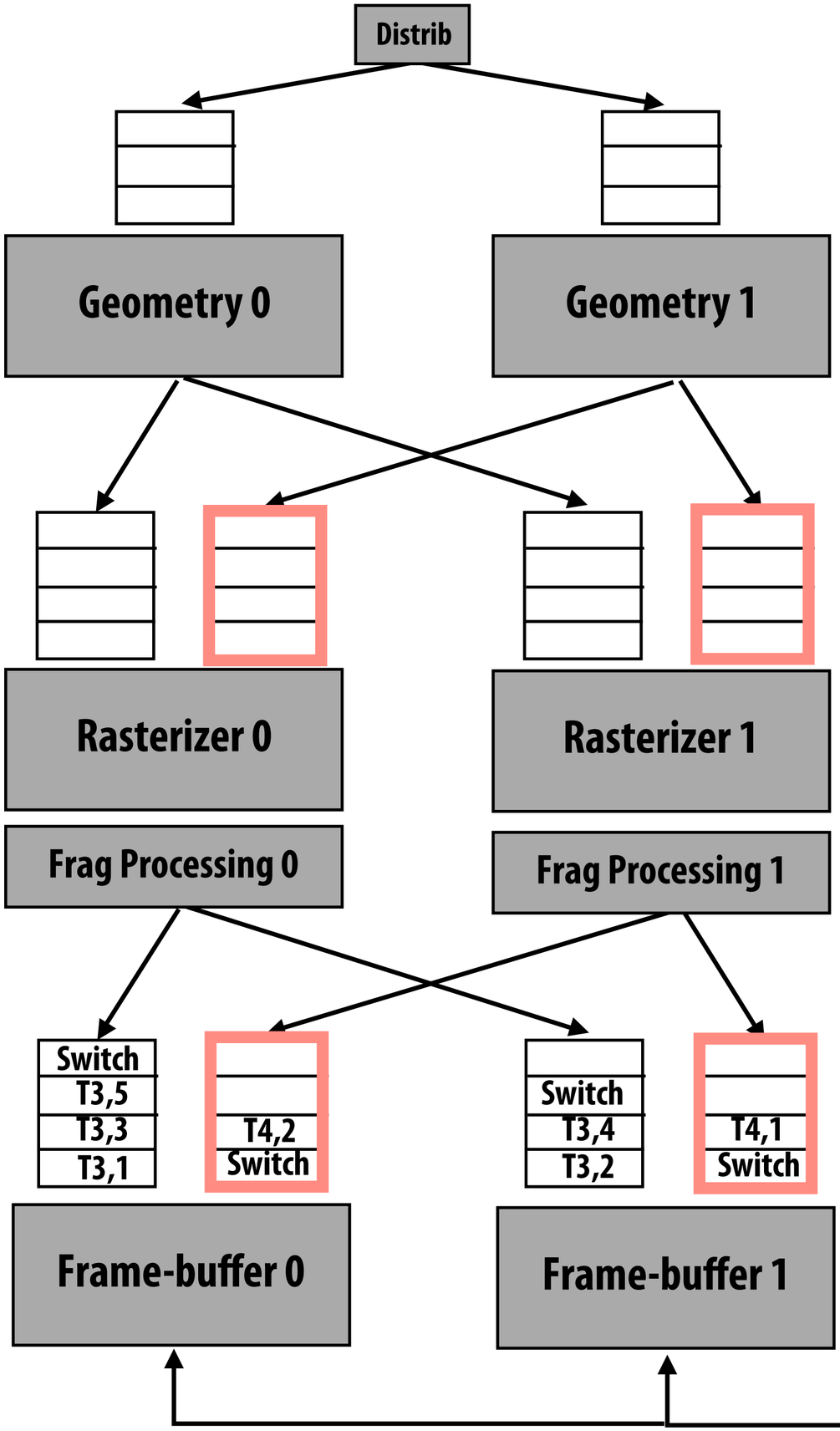
Draw T4 → 1 2



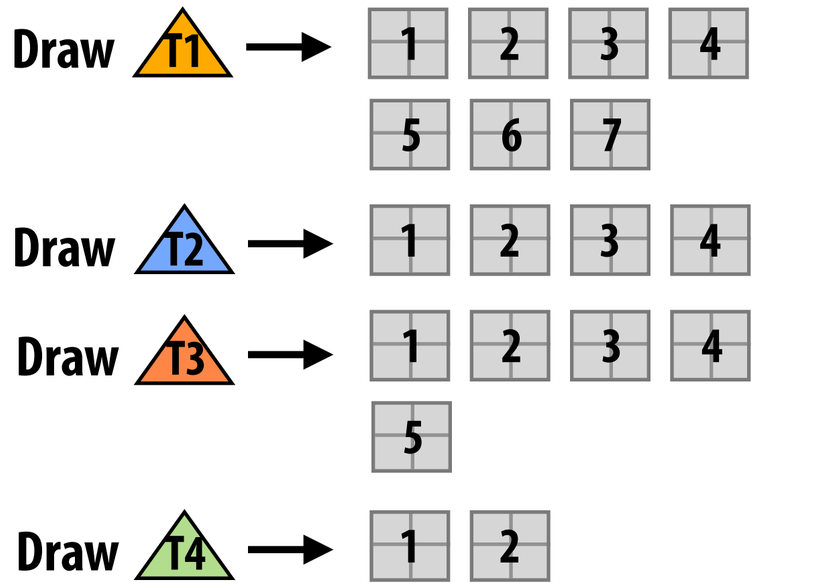
Interleaved render target

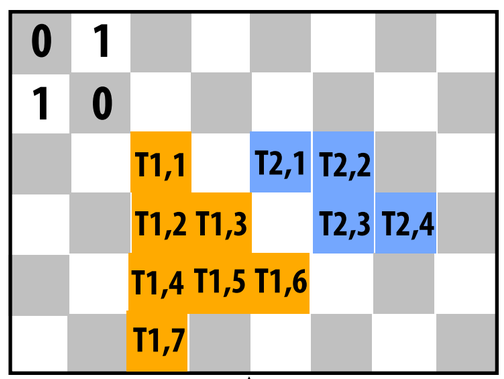
Frame-buffer units process frags from (geom 0, rast 1) in parallel

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



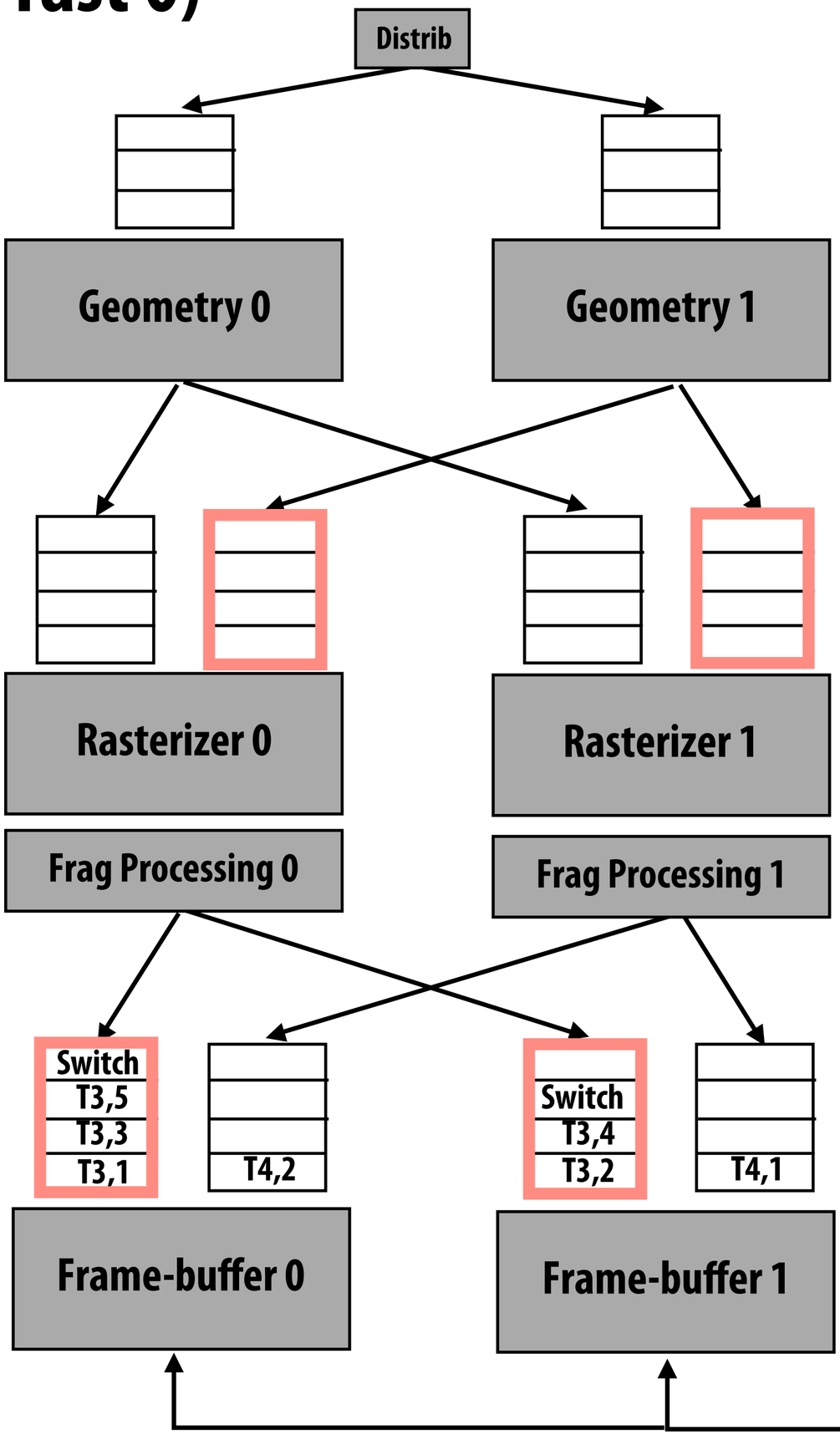
Input:



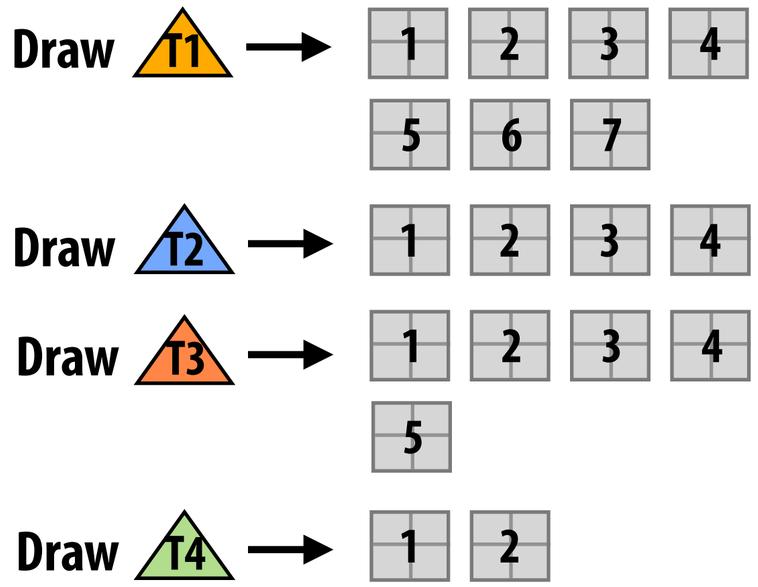


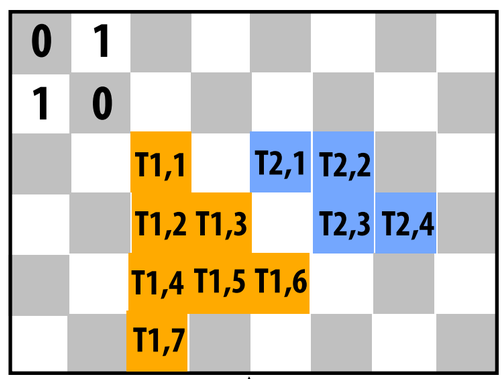
Interleaved render target

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



Input:

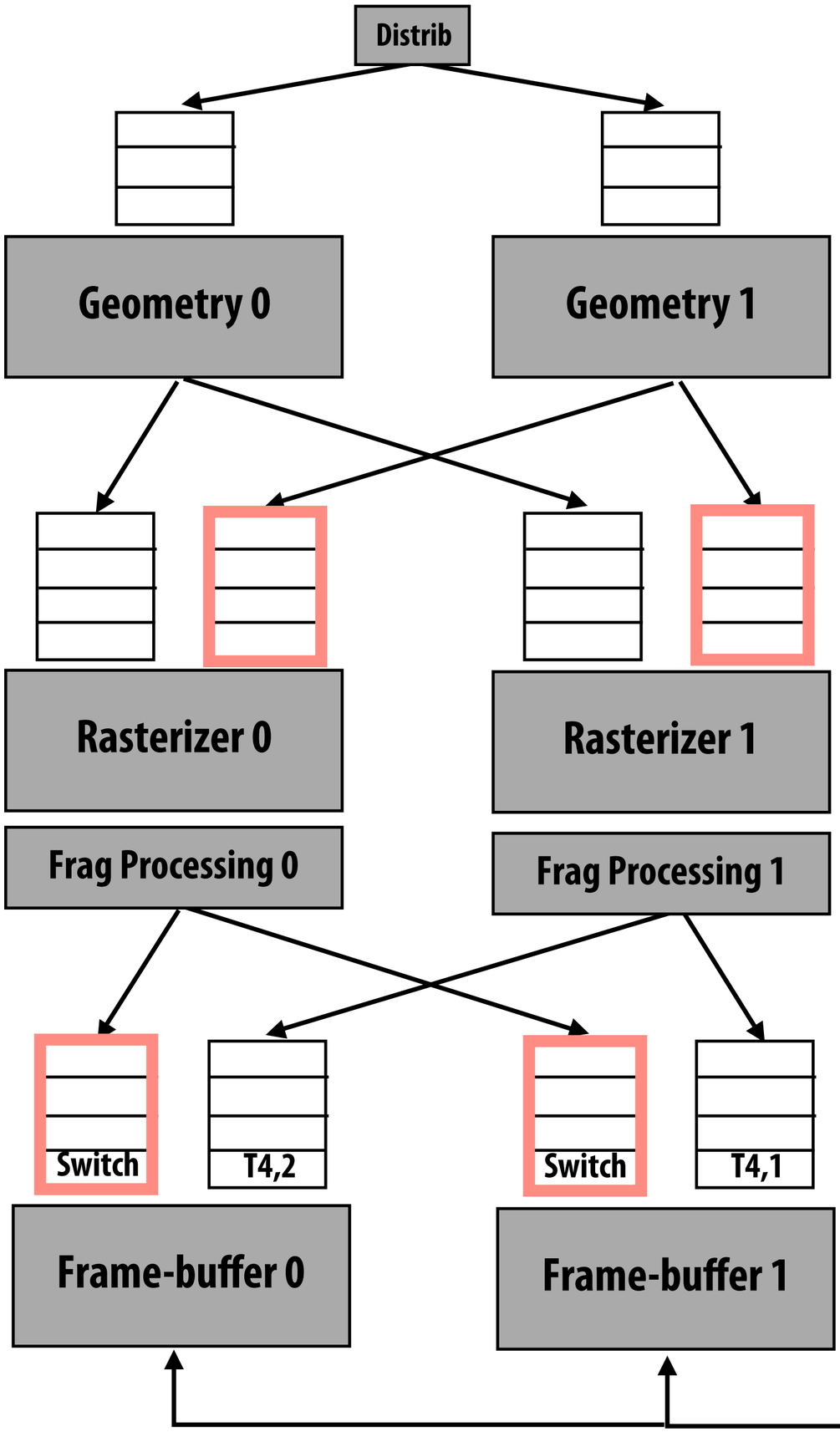




Interleaved render target

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

(Notice updates to frame buffer)



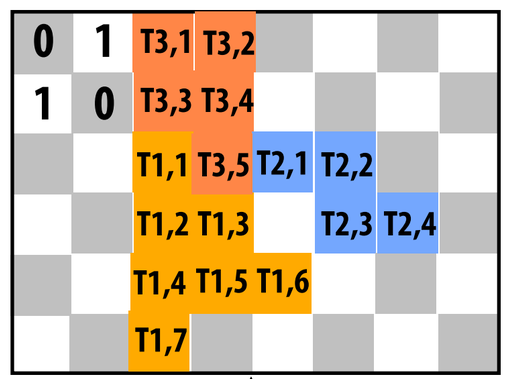
Input:

Draw T1 → 1 2 3 4 5 6 7

Draw T2 → 1 2 3 4

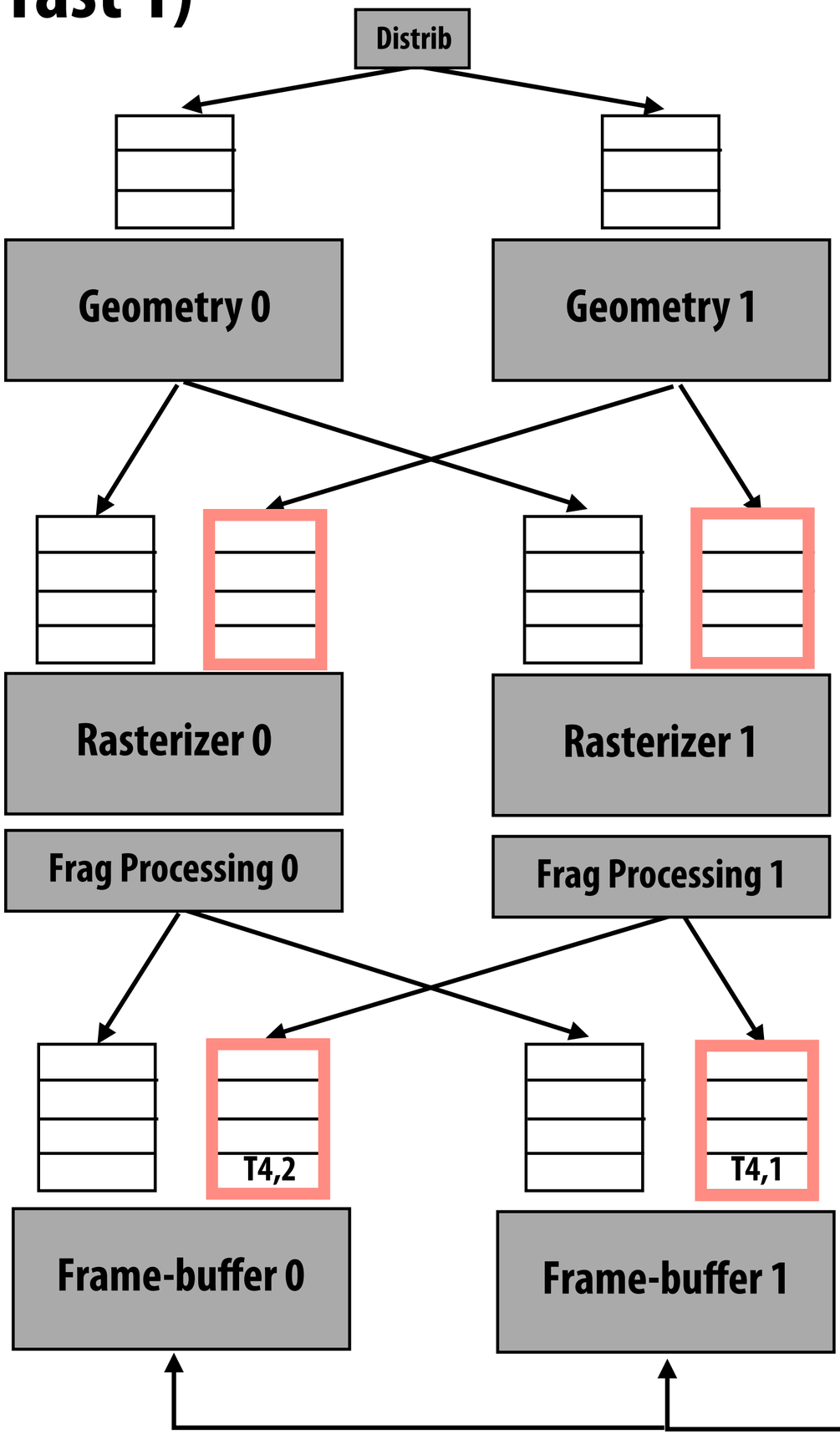
Draw T3 → 1 2 3 4 5

Draw T4 → 1 2



Interleaved render target

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



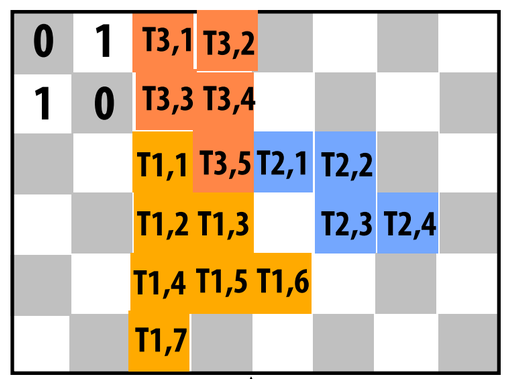
Input:

Draw T1 → 1 2 3 4 5 6 7

Draw T2 → 1 2 3 4

Draw T3 → 1 2 3 4 5

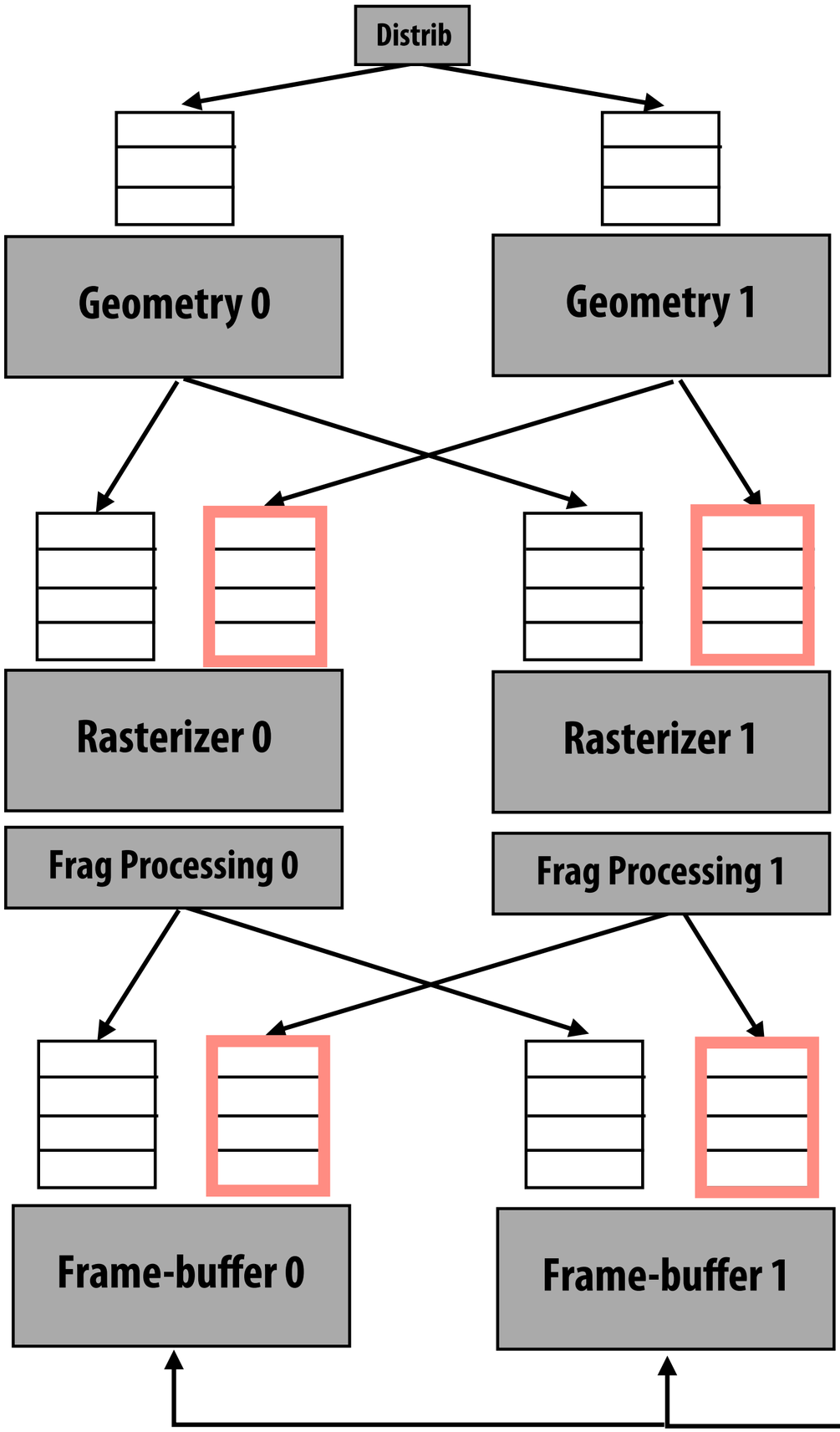
Draw T4 → 1 2



Interleaved render target

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

(Notice updates to frame buffer)



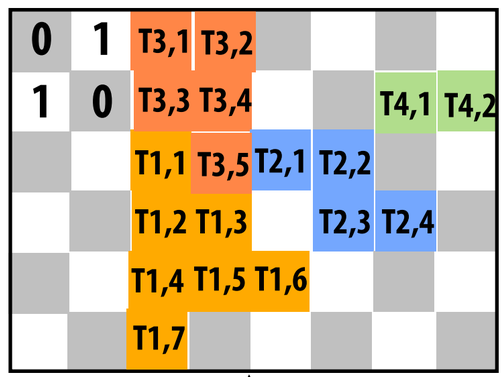
Input:

Draw T1 → 1 2 3 4 5 6 7

Draw T2 → 1 2 3 4

Draw T3 → 1 2 3 4 5

Draw T4 → 1 2



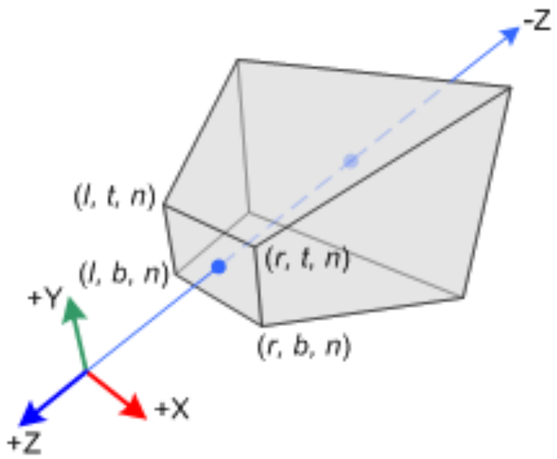
Interleaved render target

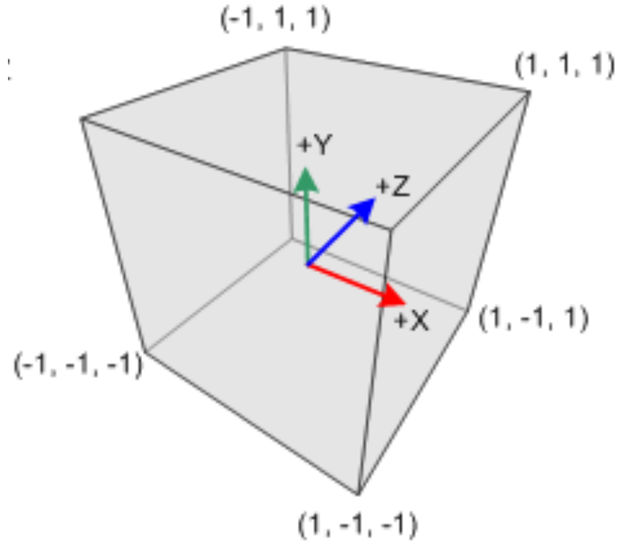
Geometry Processing Basics

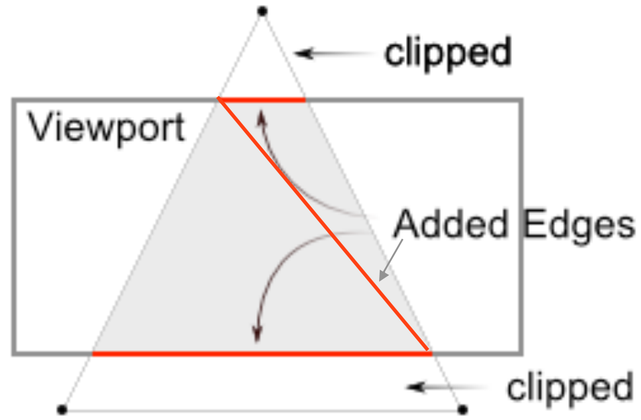
(Focus on design and implementation challenges of tessellation)

Quick review: coordinate spaces

- Modeling transform: compute vertex positions in world-space (“object-to-world” transform)
- View transform: compute vertex position in camera-space (“world-to-camera” transform)
- Projection transform: compute vertex position in homogeneous clip space (“camera-to-clip” transform)







Camera-space 3D view frustum:

- Camera at origin
- Camera looking down -Z axis
- All triangles intersecting with this volume will be drawn. All triangles outside the region will be discarded prior to rasterization (they are off-screen)

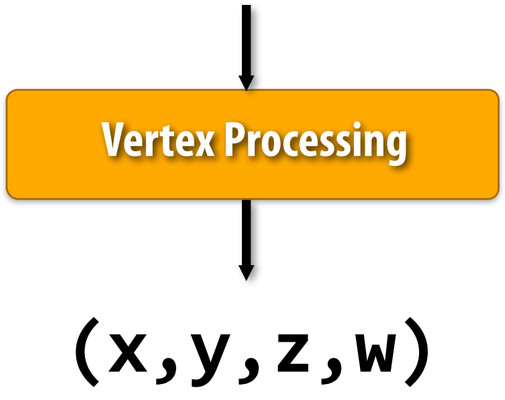
Canonical clip-space view frustum:

- (-1,-1,-1) to (1,1,1) in euclidian 3D space
- (-w,-w,-w, *) to (w,w,w, *) in homogeneous 3D space

Clipping example: triangle extends beyond view volume. Clipped to form two triangles entirely within volume.

Image credit: http://www.songho.ca/opengl/gl_projectionmatrix.html

Per vertex operations: application-specified vertex shader outputs vertex positions in clip space



Vertex positions emitted by vertex processing (or the geometry shader, if enabled) are represented in homogeneous, clip-space coordinates.

Homogeneous vertex v = `(x, y, z, w)`

Vertex position in euclidian space: `(x/w, y/w, z/w)`

Vertex is within the canonical clip-space view frustum if:

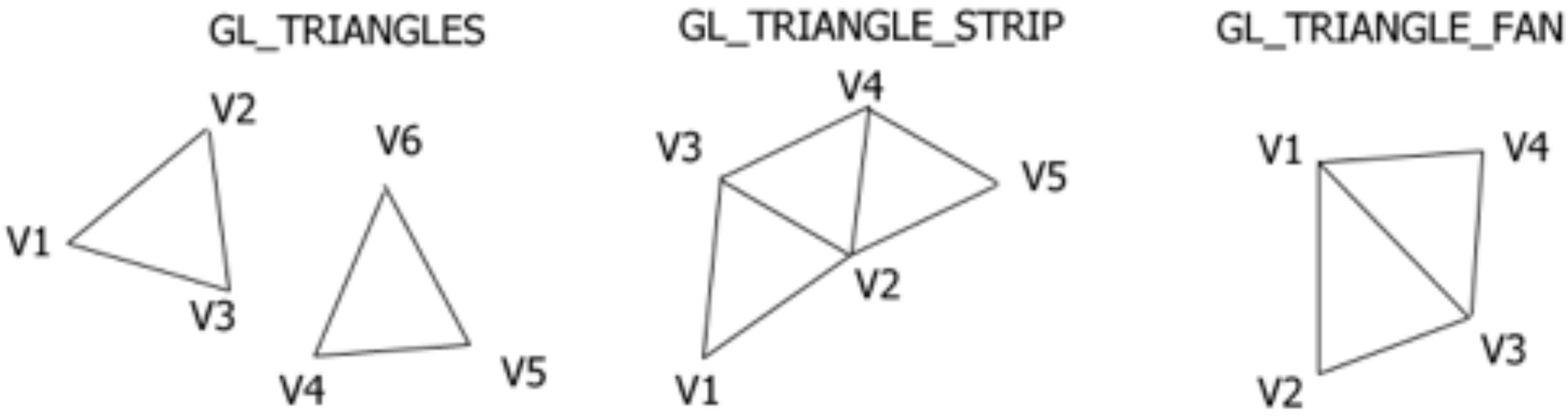
```
-w ≤ x ≤ w
-w ≤ y ≤ w
-w ≤ z ≤ w
```

Fixed-function, per-primitive operations

Given triangle's input vertices (with positions in homogeneous clip space):

1. **"Assemble" (a.k.a. group) vertices into primitives**
2. **Discard primitive if it is entirely outside the view frustum**
3. **Clip primitive against view frustum (if clipping is necessary)**
4. **For each resulting primitive**
   - **Divide homogeneous x,y,z coordinates by w (coordinates now in [-1,1] range)**
   - **Apply viewport transform to compute screen position of vertex (coordinates now in the range [0,screen_width])**
   - **Discard back-facing primitives [optional, depends on rasterizer configuration]**

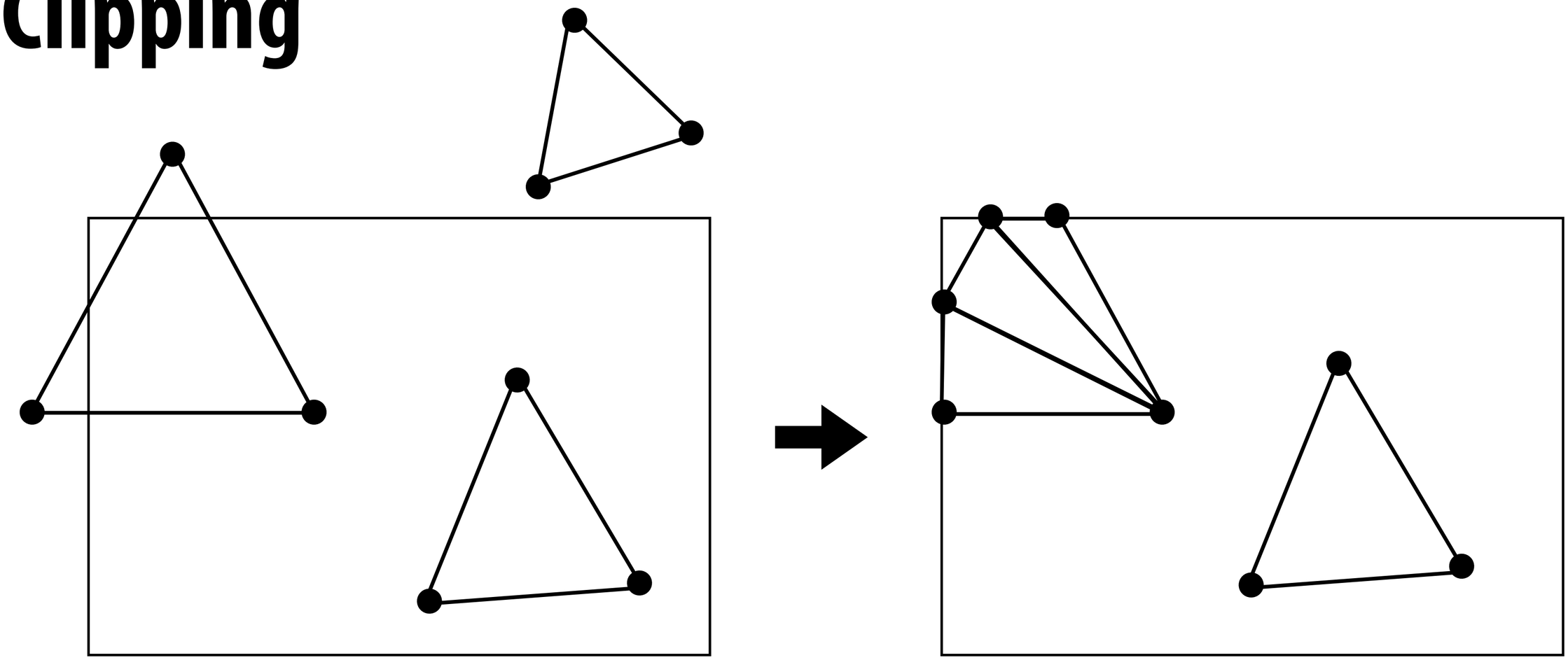
Assembling vertices into primitives



How to assemble vertices into primitives is pipeline state (specified as an argument to the pipeline's draw command)

Notice: vertices are processed independently, but get grouped into primitives (this is a "join" in fork-join parallelism terminology)

Clipping



- May generate new vertices/primitives, or eliminate vertices/primitives
- Data-dependent computation
  - Variable amount of work per primitive
  - Variable control flow per primitive

Why clipping?

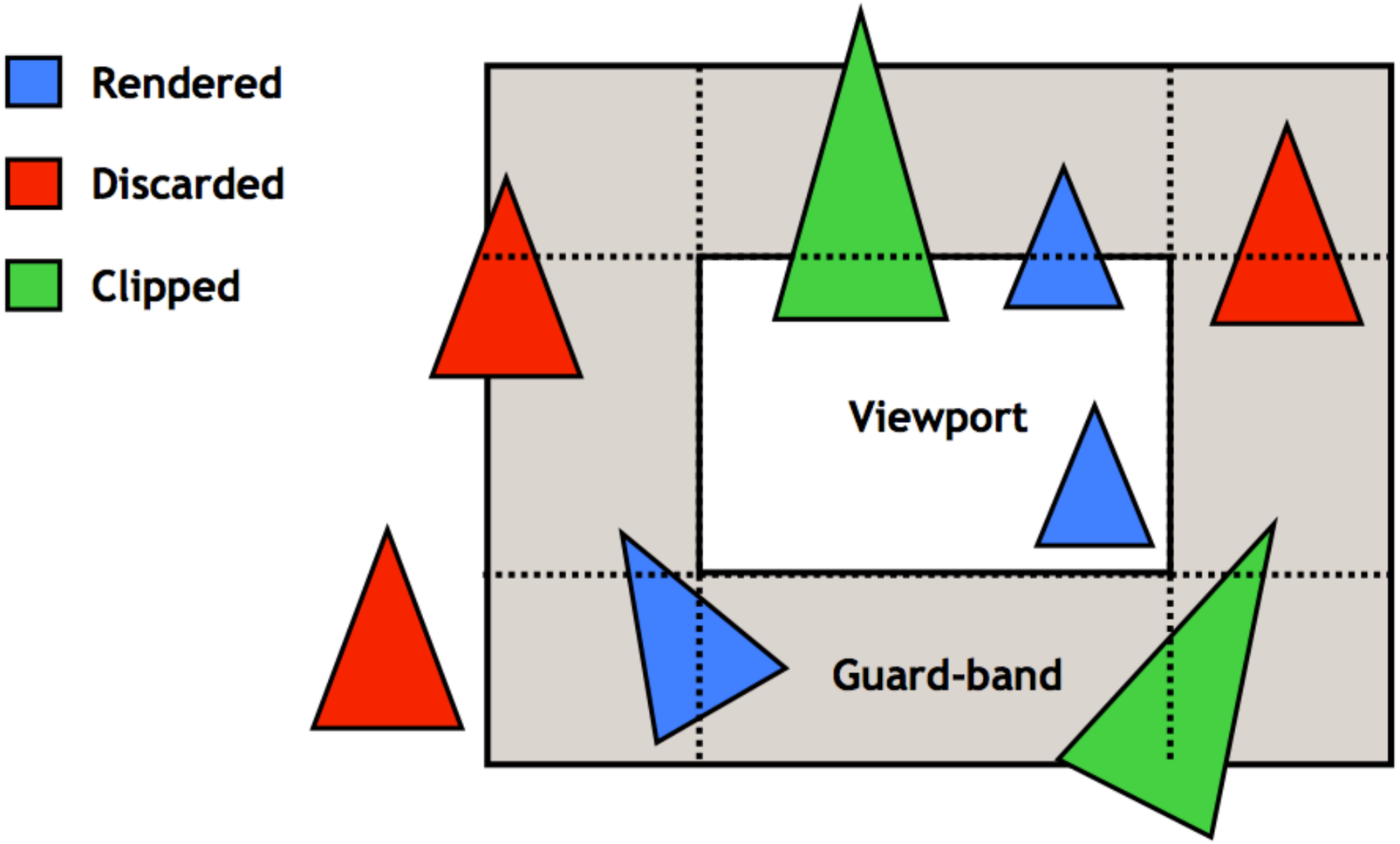
- Avoid downstream processing that will not contribute to image (rasterization, fragment processing)

- Establish invariants for emitted primitives
  - Can safely divide vertex clip space *xyz* coordinates by *w* after clipping
  - Bounds vertex positions (allowing system to perform subsequent rasterization operations using lower precision arithmetic)

Guard-band clipping

[RealityEngine, Akeley 93]

- Idea: clip against wider X-Y plane clip bounds than canonical view volume
- Significantly reduces number of triangles that must undergo clipping -- reduces variance in per-primitive clipping work
- Cost is that primitives no longer guaranteed to be fully on screen
  - Rasterizer must not generate off-screen fragments
  - Increased precision required during rasterization



Back-face culling

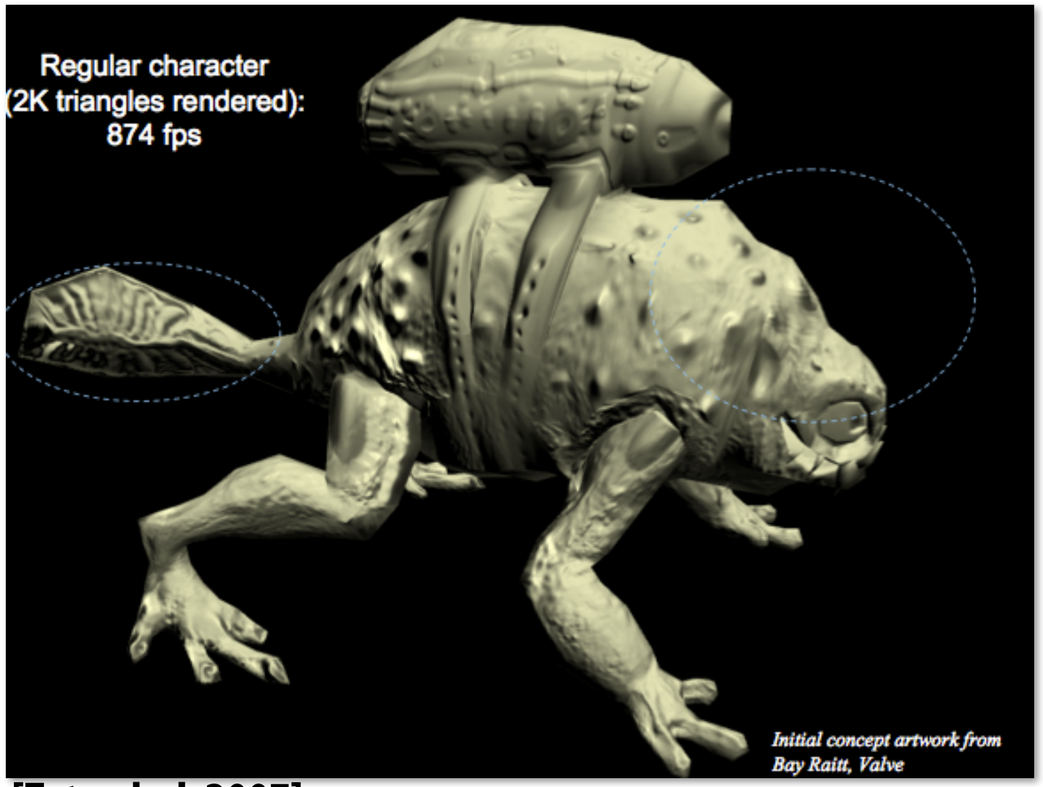
- Sign of triangle area indicates if triangle is facing toward or away from camera
- Pipeline may discard primitive as a result of this test
  - For closed meshes, eliminates ~ 1/2 of triangles
    (these triangles would be occluded by other triangles anyway)

$$\textit{Triangle area} = \frac{(x0y1 - x1y0) + (x1y2 - x2y1) + (x2y0 - x0y2)}{2}$$

GPU Tessellation Pipeline

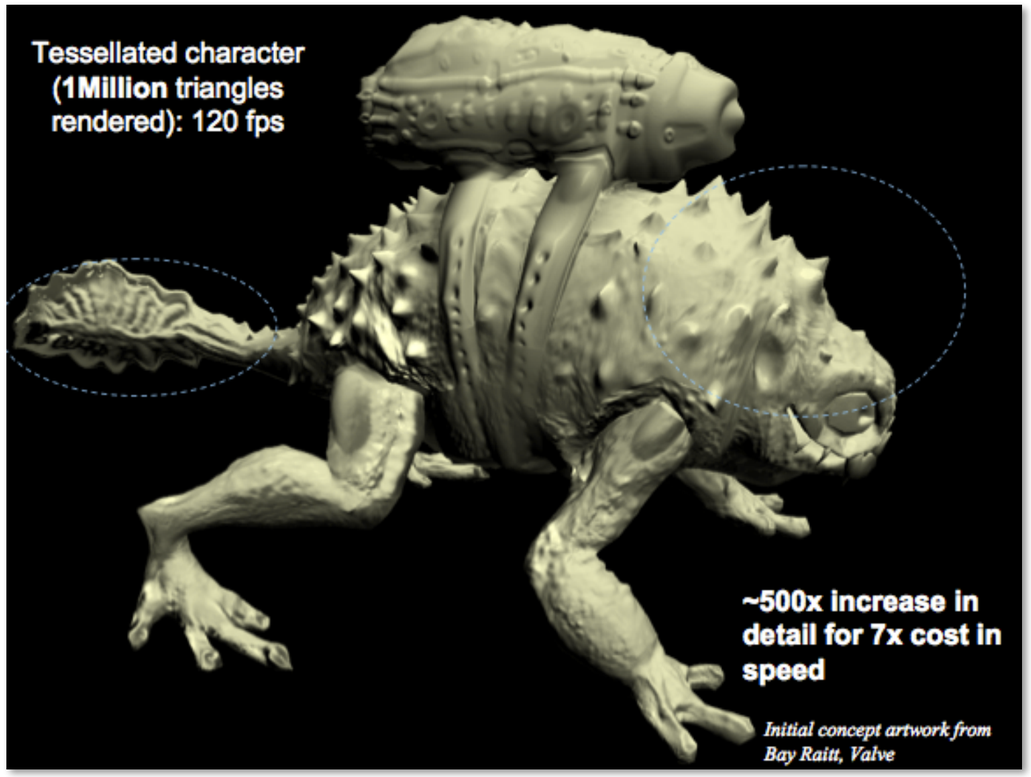
Graphics pipeline support for tessellation

No tessellation



[Tatarchuk 2007]

Tessellated surfaces with displacement mapping



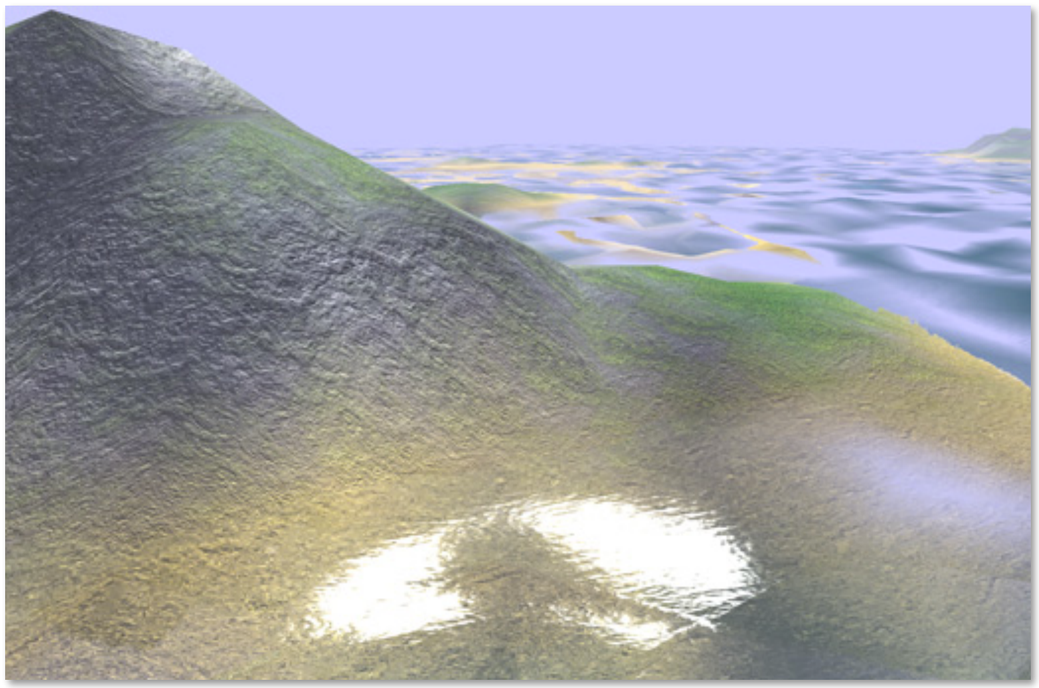
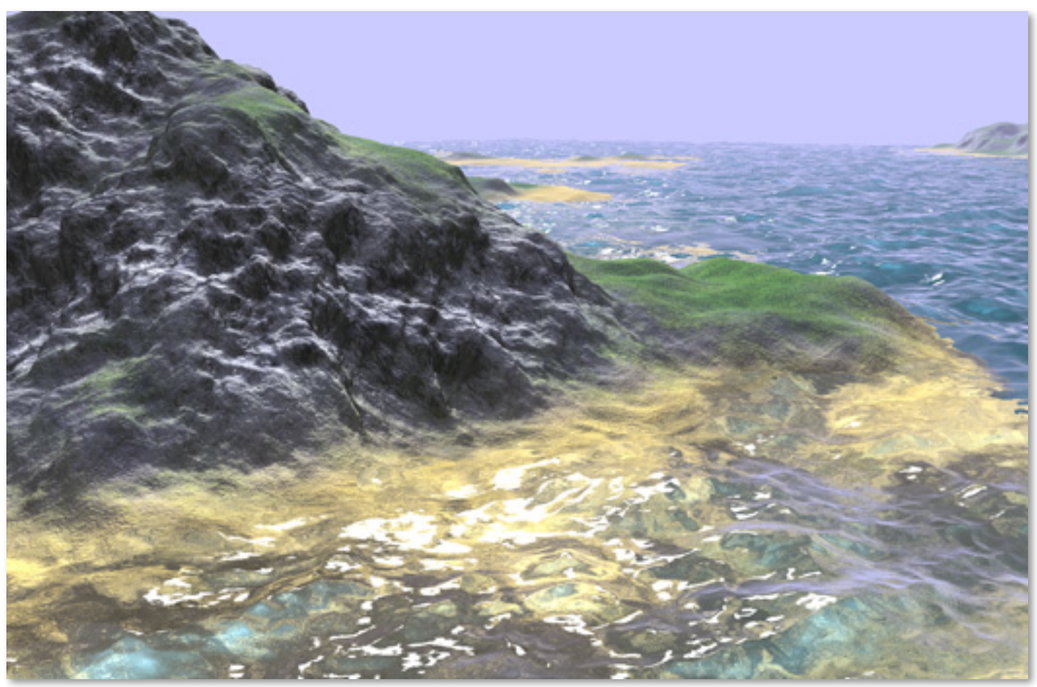


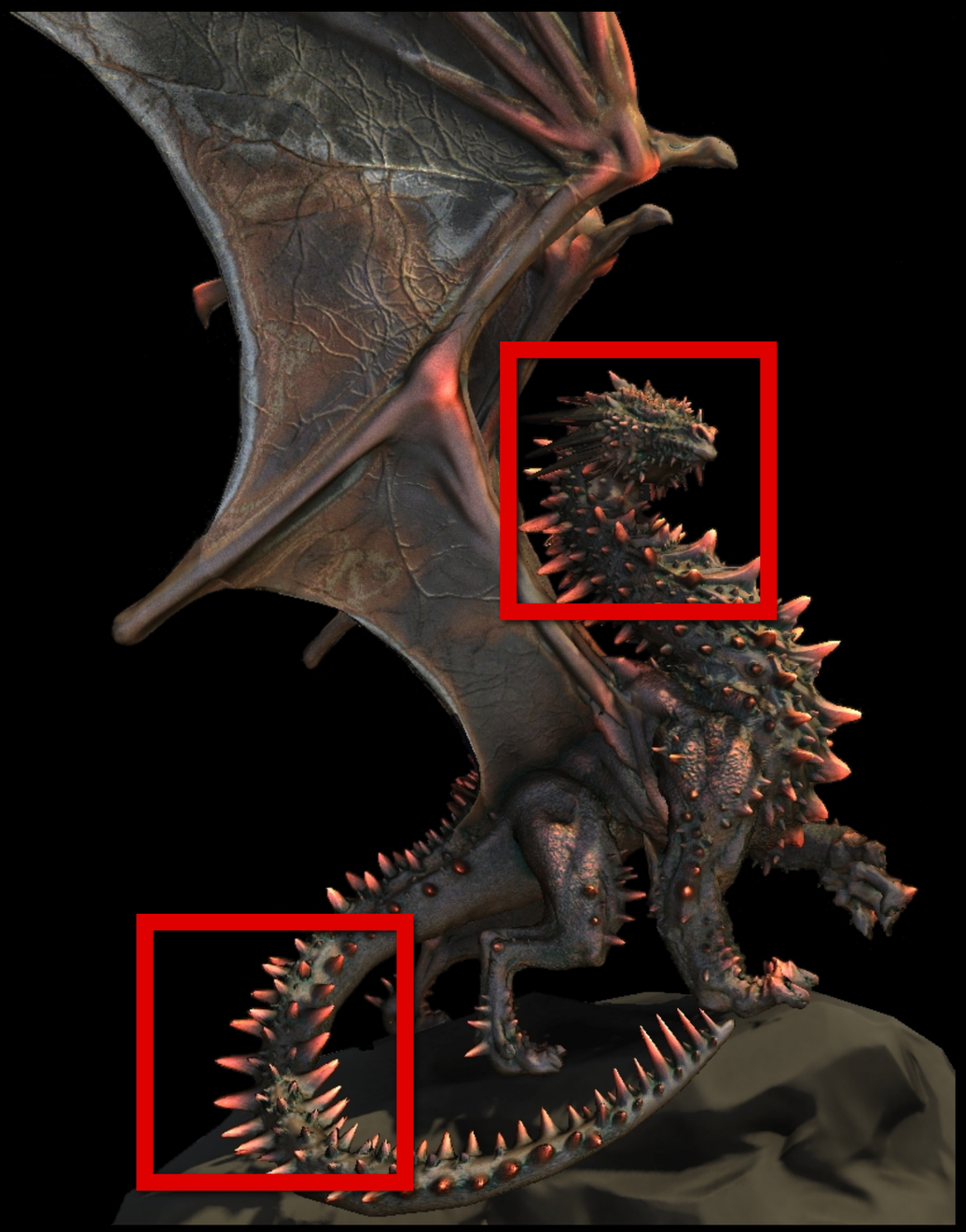
Image credit: 3D Guru (from NVIDIA)



No tessellation

Image credit: Unigine Engine, copyright Unigine Corp.

Tessellated surfaces with displacement mapping



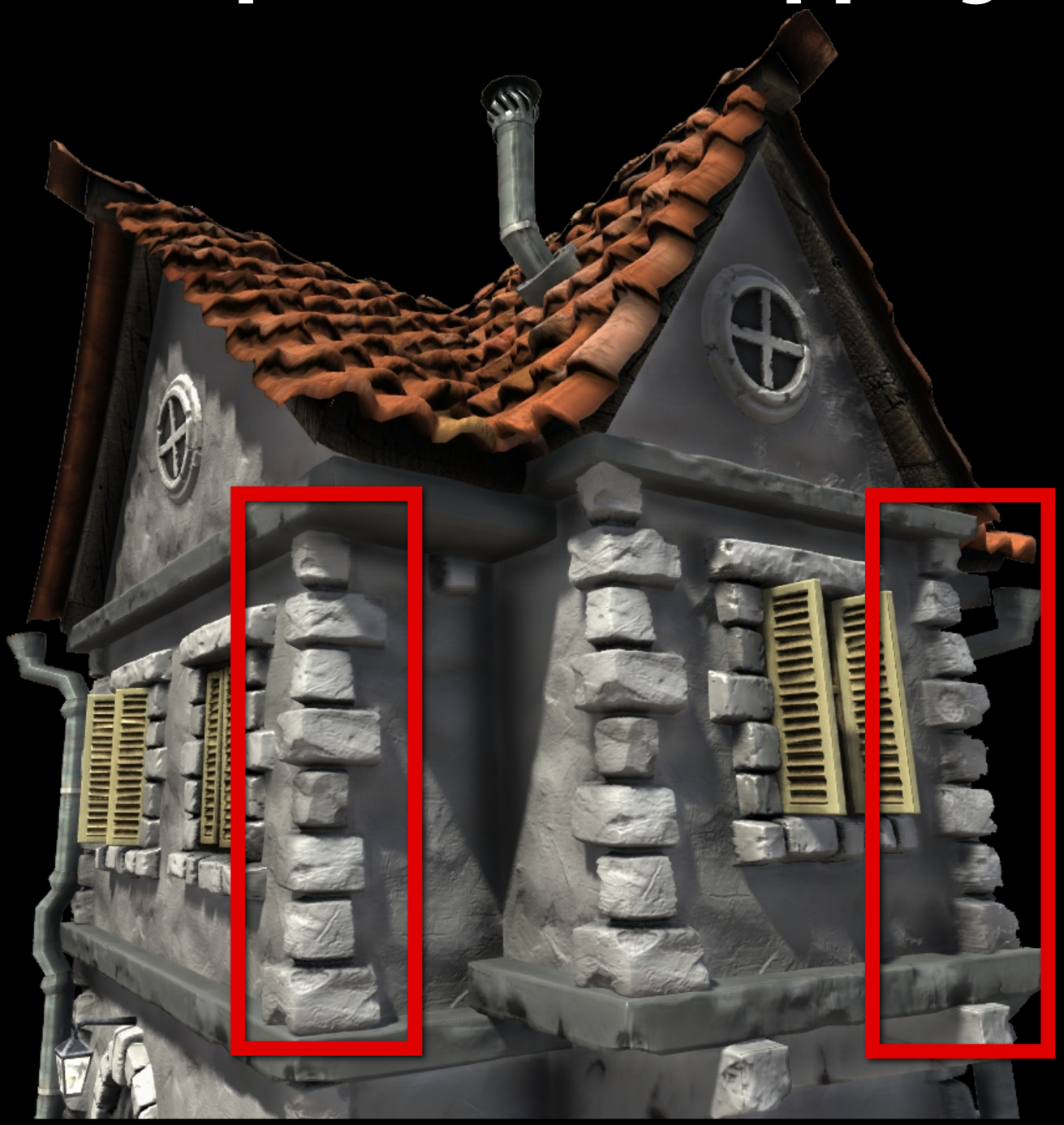
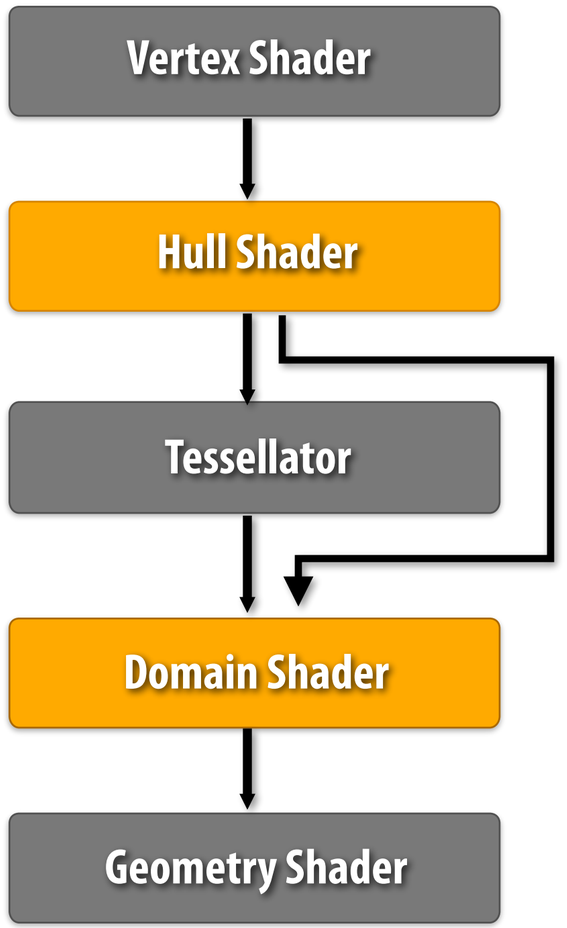


Image credit: Unigine Engine, copyright Unigine Corp.

Modern GPU tessellation [Moreton 01]

- **Motivations:**
  - Reduce footprint (do not store high-resolution triangles in memory)
  - For animated surfaces: reduce CPU-GPU bandwidth (do not transfer high resolution mesh over PCIe bus to GPU each frame)
  - Animate/skin/compute physics using course resolution mesh, but render high resolution mesh

Note: D3D11 Stage Naming
(not canonical stage names)



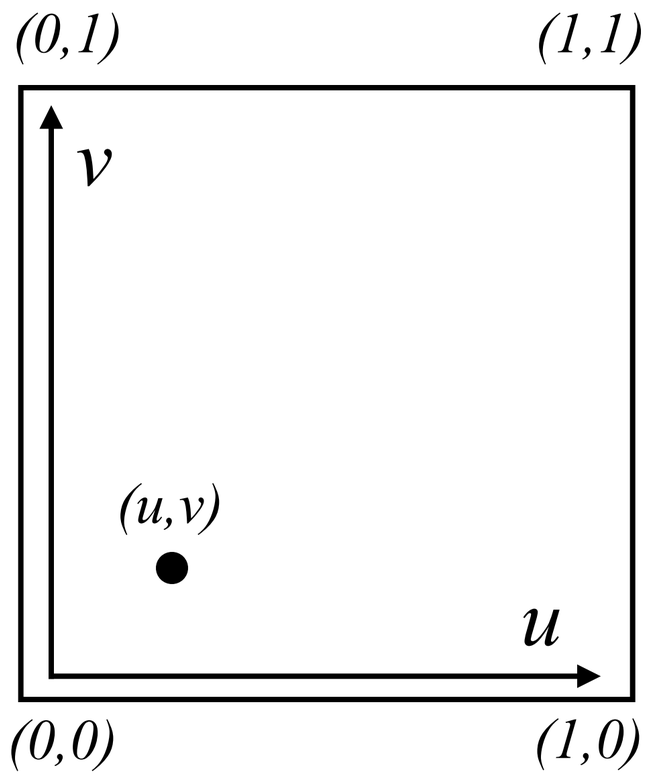
- **Design:**
  - Hybrid programmable / non-programmable implementation using three new pipeline stages
  - Supports only parametric surfaces:

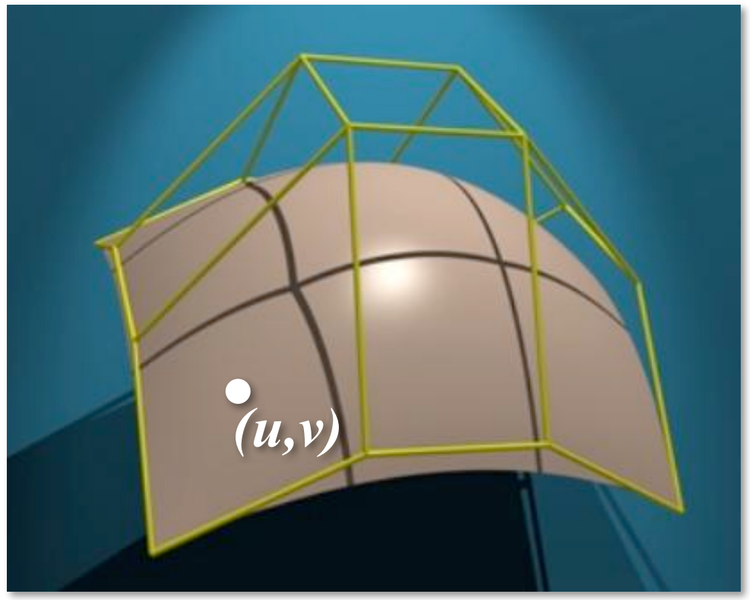
    surface must support direct evaluation of $position(u,v)$

Pipeline tessellation requires parametric surfaces

3D surface position is a function of 2D parametric coordinate (u,v)

Below: $[0,1]^2$ parametric domain



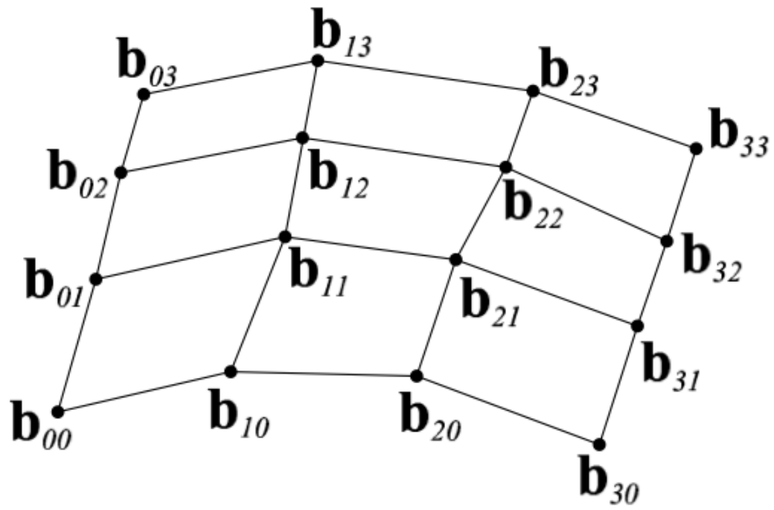


$f(u,v)=(x,y,z)$

Image credit: http://csis.pace.edu/~marchese/CG/Lect10/cg_10.htm

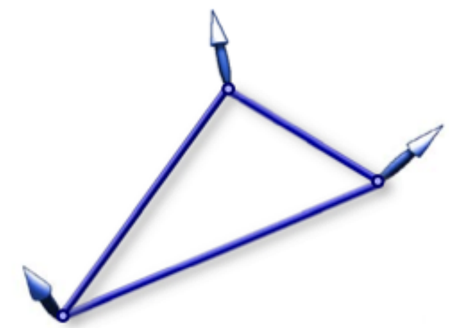
Example parametric surfaces

Bicubic patch, defined by 16 control points (quadrilateral domain)



See "Approximating Catmull-Clark Subdivision Surfaces With Bicubic Patches", Loop et al. 2008

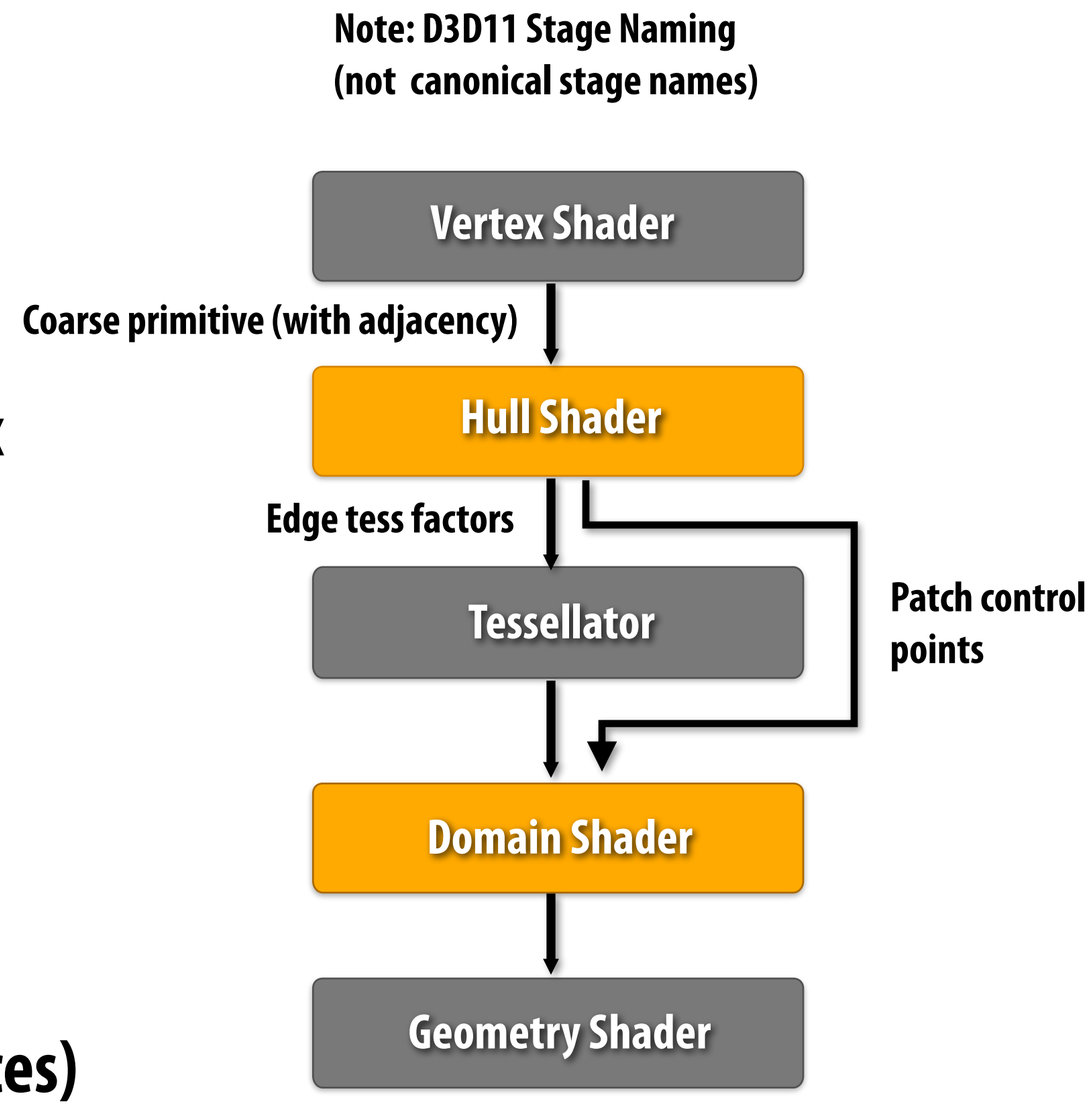
PN Triangles, 3 vertices + 3 normals (defines bezier patch on triangular domain)



See "Curved PN Triangles", Vlachos et al. 2008

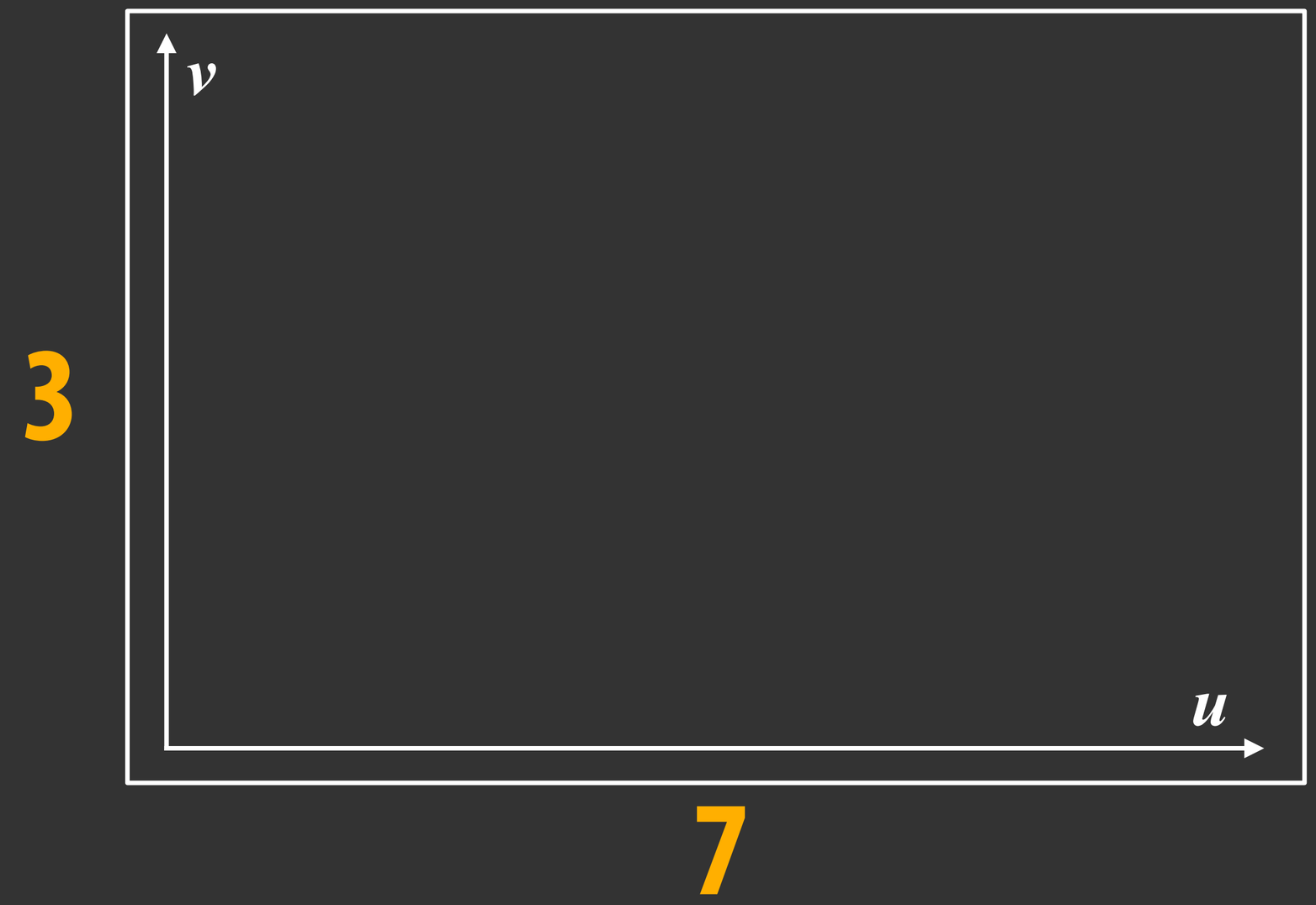
Three-stage tessellation pipeline

- **Stage 1: hull shader**
  - Programmable stage
  - Input: one "surface patch" primitive: grouped vertices after traditional vertex processing
  - Output: new "patch" primitive
  - Tessellation factor for each edge of parametric domain
  - Control points for parametric surface (computed from input primitive vertices)



Hull shader produces edge tessellation rates

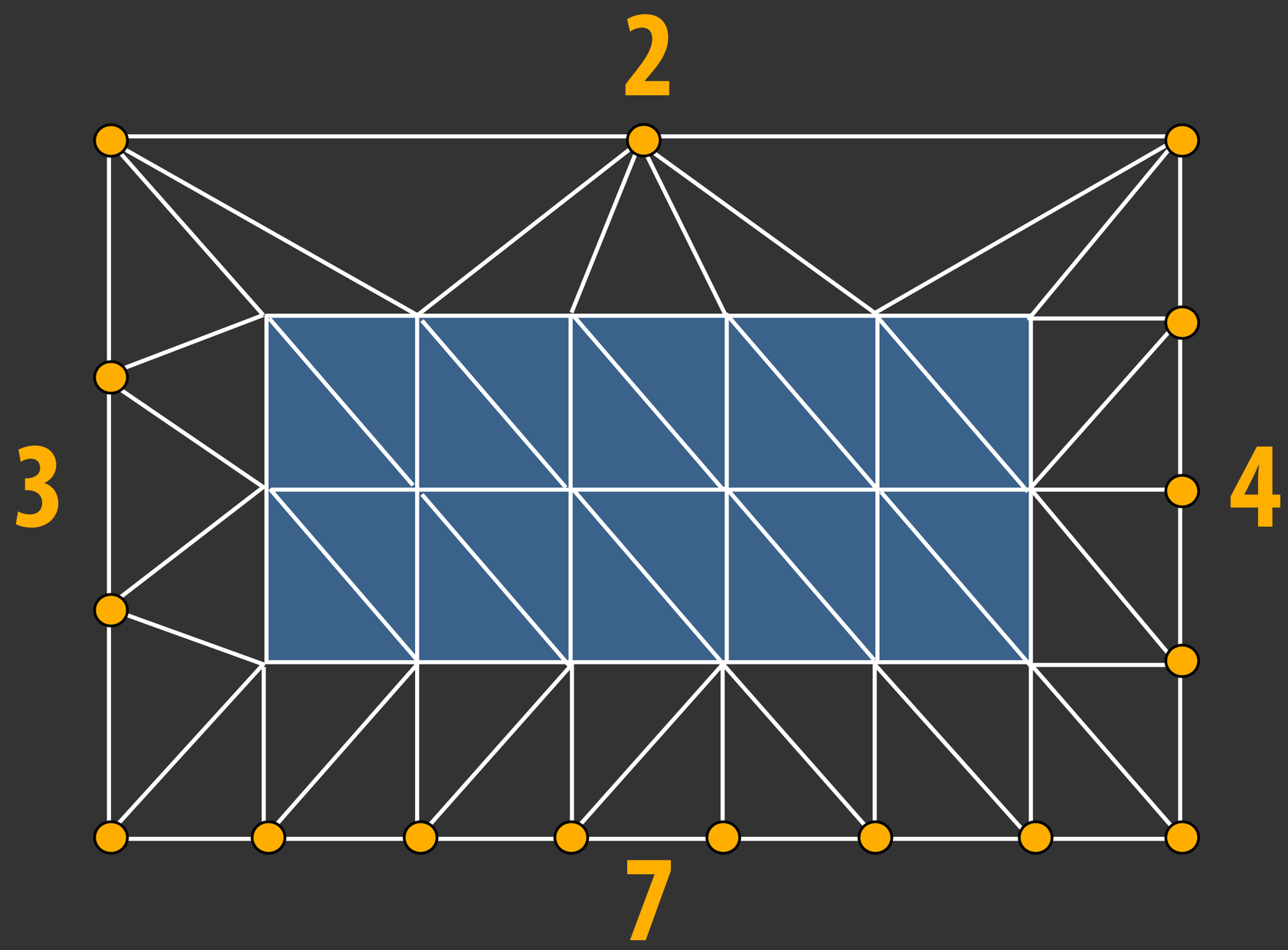
Based on estimate of parametric surface position (e.g, larger surfaces on screen yield higher tessellation rates)



Stage 2: fixed-function tessellation stage

Input: edge tessellation constraints for a patch

Output: (almost) uniform mesh topology meeting constraints

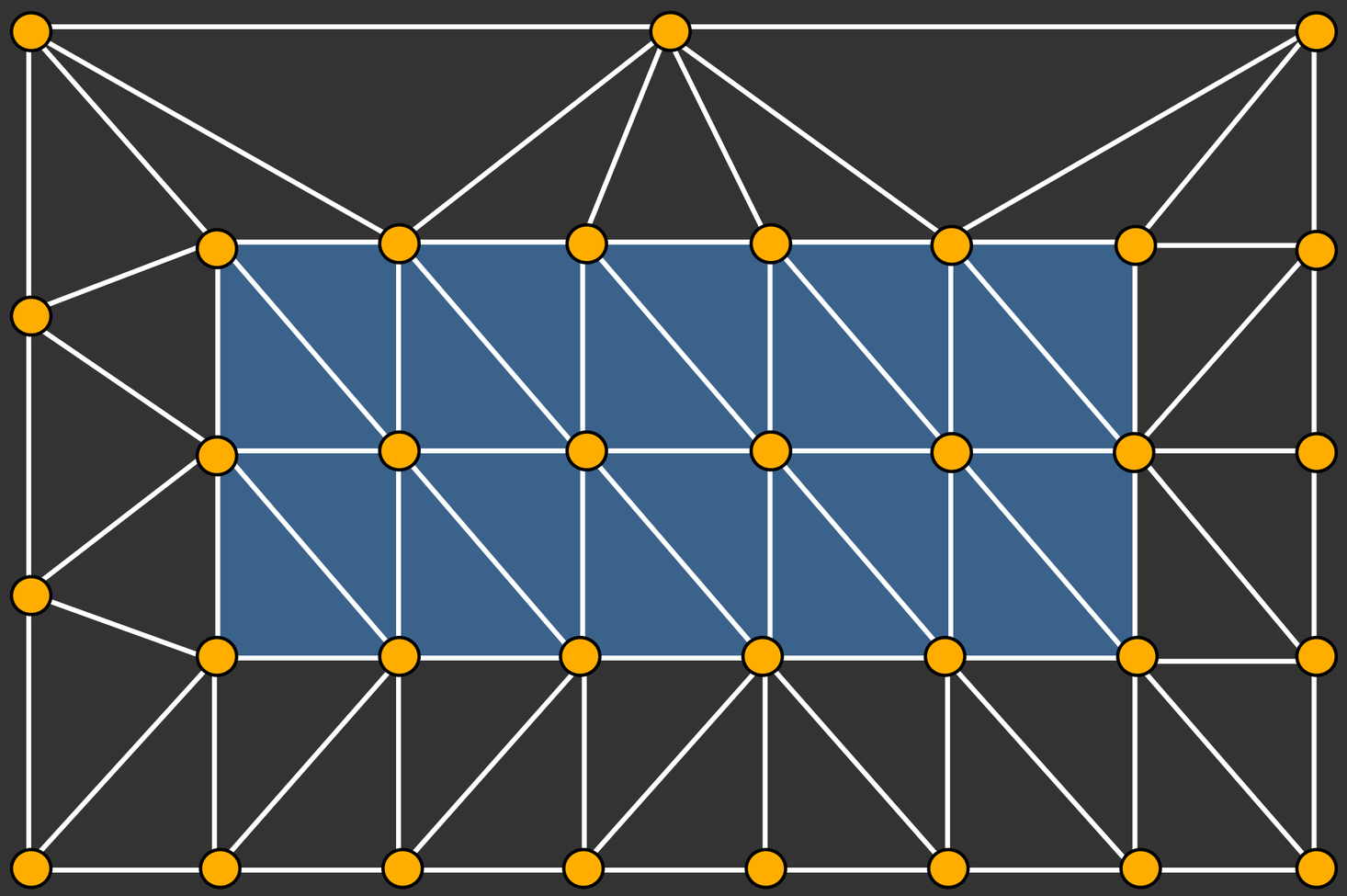


[Moreton 01]

Stage 3: domain shader (“fine vertex” shader)

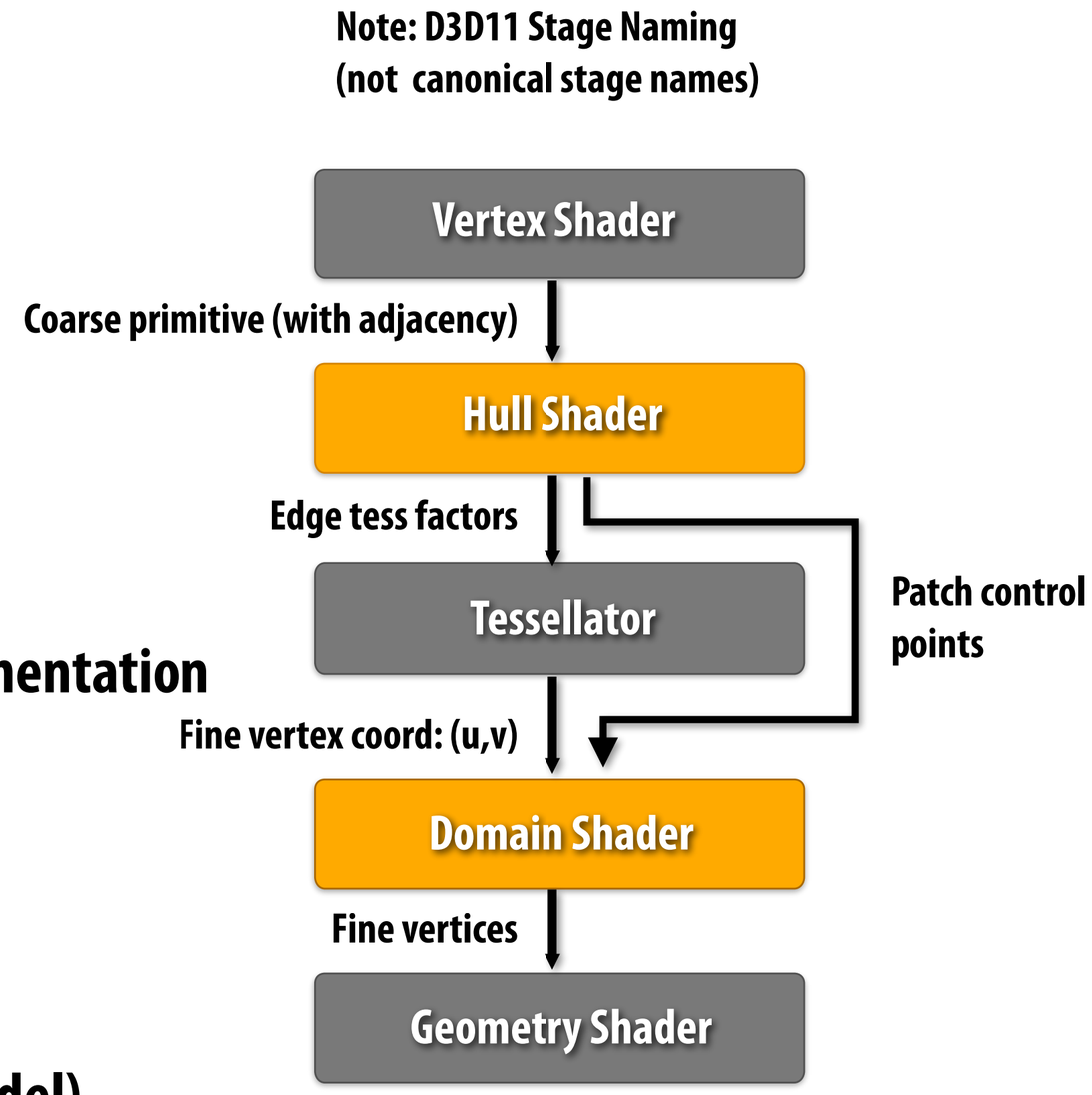
Input: control points (from hull shader) and stream of parametric vertex locations (u,v) from tessellator

Output: position of vertex at parametric coordinate: $position(u,v)$

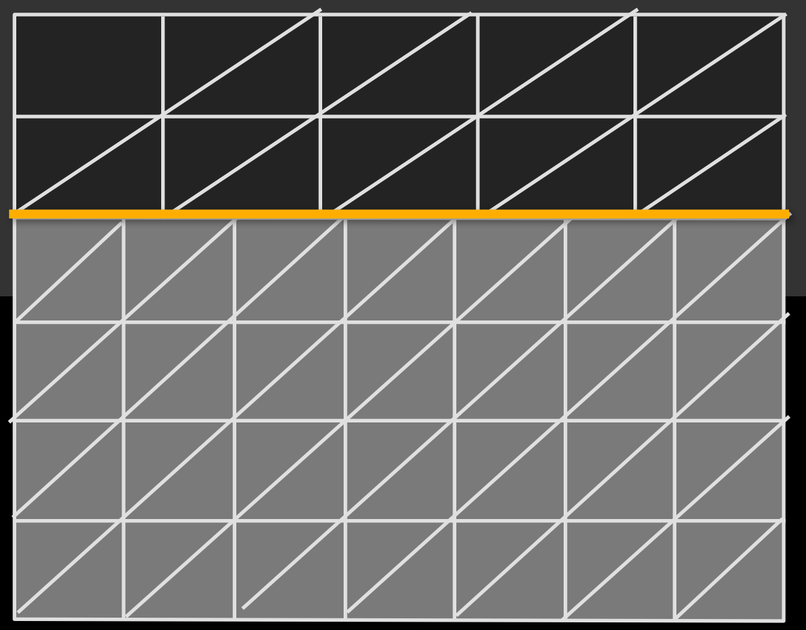


Modern GPU tessellation pipeline

- **Hull shader (programmable)**
  - Coarse primitive granularity
  - Data-parallel across coarse primitives
  - Large working set per invocation (typically a primitive + one-ring)
- **Tessellator (fixed-function)**
  - Surface agnostic, fixed-function hardware implementation
  - Irregular control flow
- **Domain shader (programmable)**
  - Fine-mesh-vertex granularity
  - Data-parallel (preserves shader programming model)
  - Direct evaluation of surface at specified domain coordinate



Parallel tessellation challenge: avoid cracks!



(parametric domain)

Hull shader must carefully compute edge tess factors to avoid cracks
(See suggested readings on website)

Modern GPU tessellation summary

- **Enables adaptive level-of-detail, high-resolution meshes in games**

- **Heterogeneous abstraction: co-design of algorithm and hardware**
  - **Retain flexibility: surface type is programmable, but it must be parametric**
  - **Minimize new complexity: extension of existing shader programming model**
  - **Retain performance:**
    - **Parallelism: data-parallel evaluation of fine vertex positions**
    - **Fixed-function HW algorithm for generating points (not data parallel algorithm)**

- **Implementation challenges**
  - **Application developer: avoiding cracks (requires consistent edge rate evaluation, this is tricky in floating point math)**
  - **GPU implementor: managing large data amplification… while maintaining parallelism, locality, and order**

Parallel Scheduling of Data Amplification

Geometry amplification

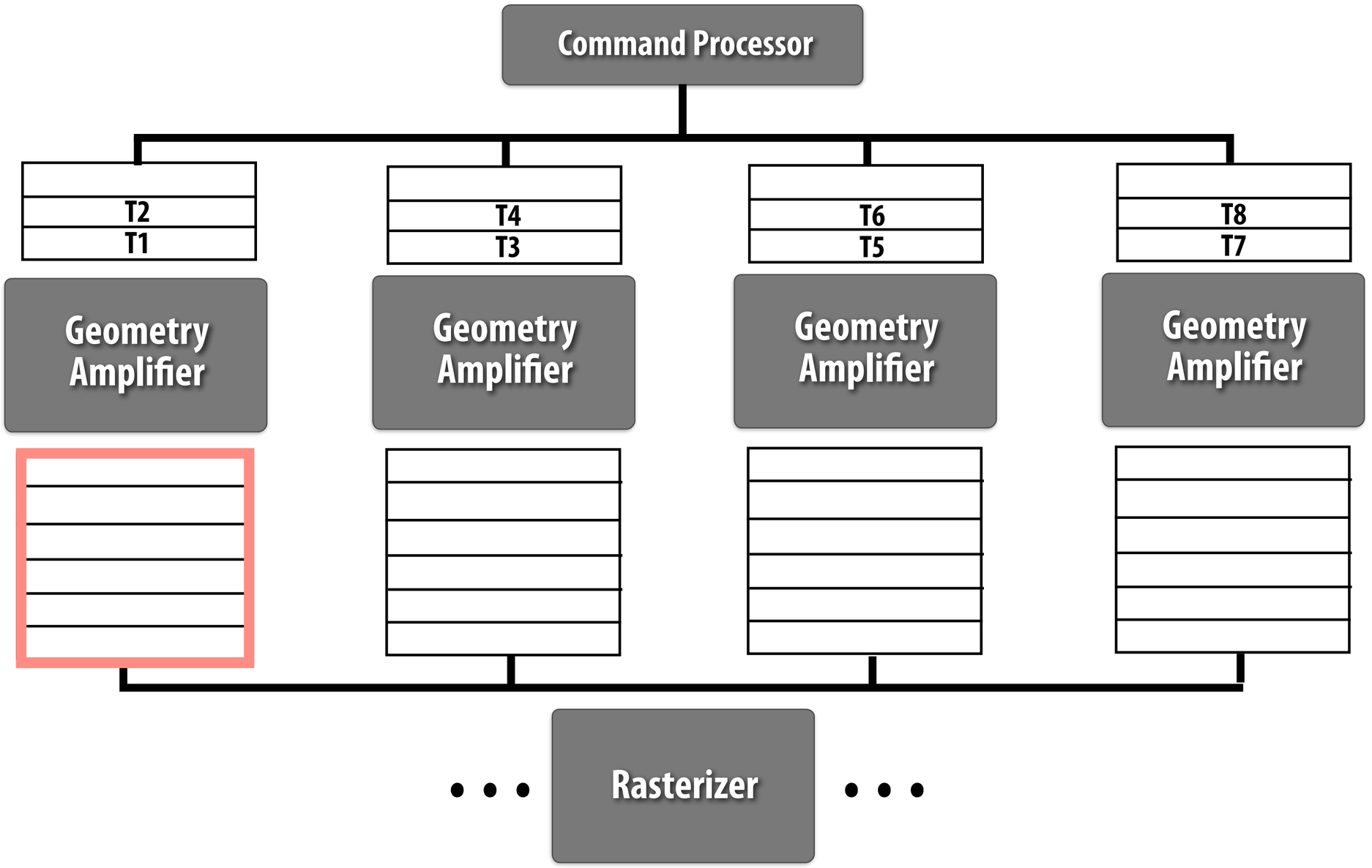
- Examples of one-to-many stage behavior during geometry processing in the graphics pipeline

  - 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)

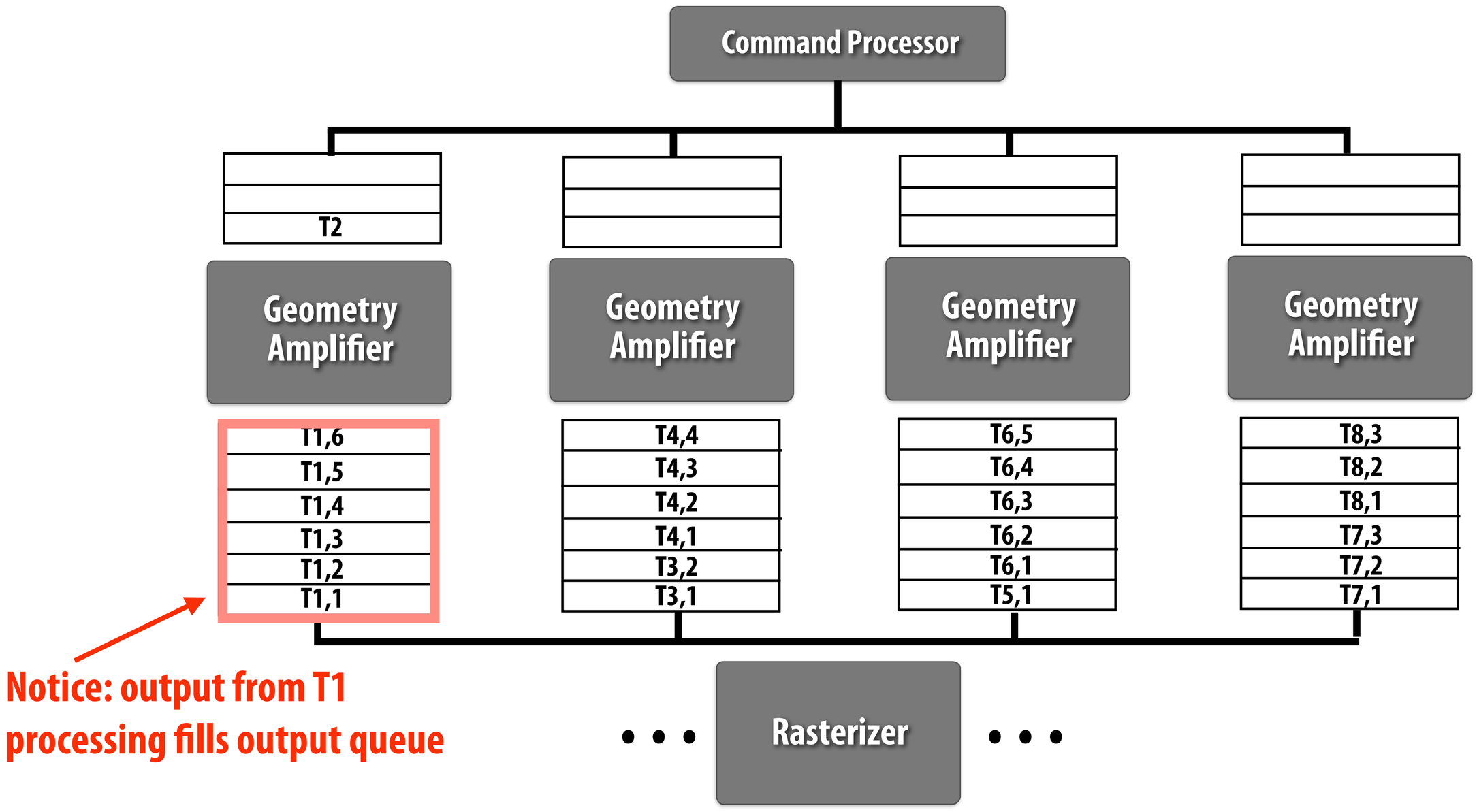
  - Geometry shader: output up to 1024 floats worth of vertices per input primitive

Thought experiment



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

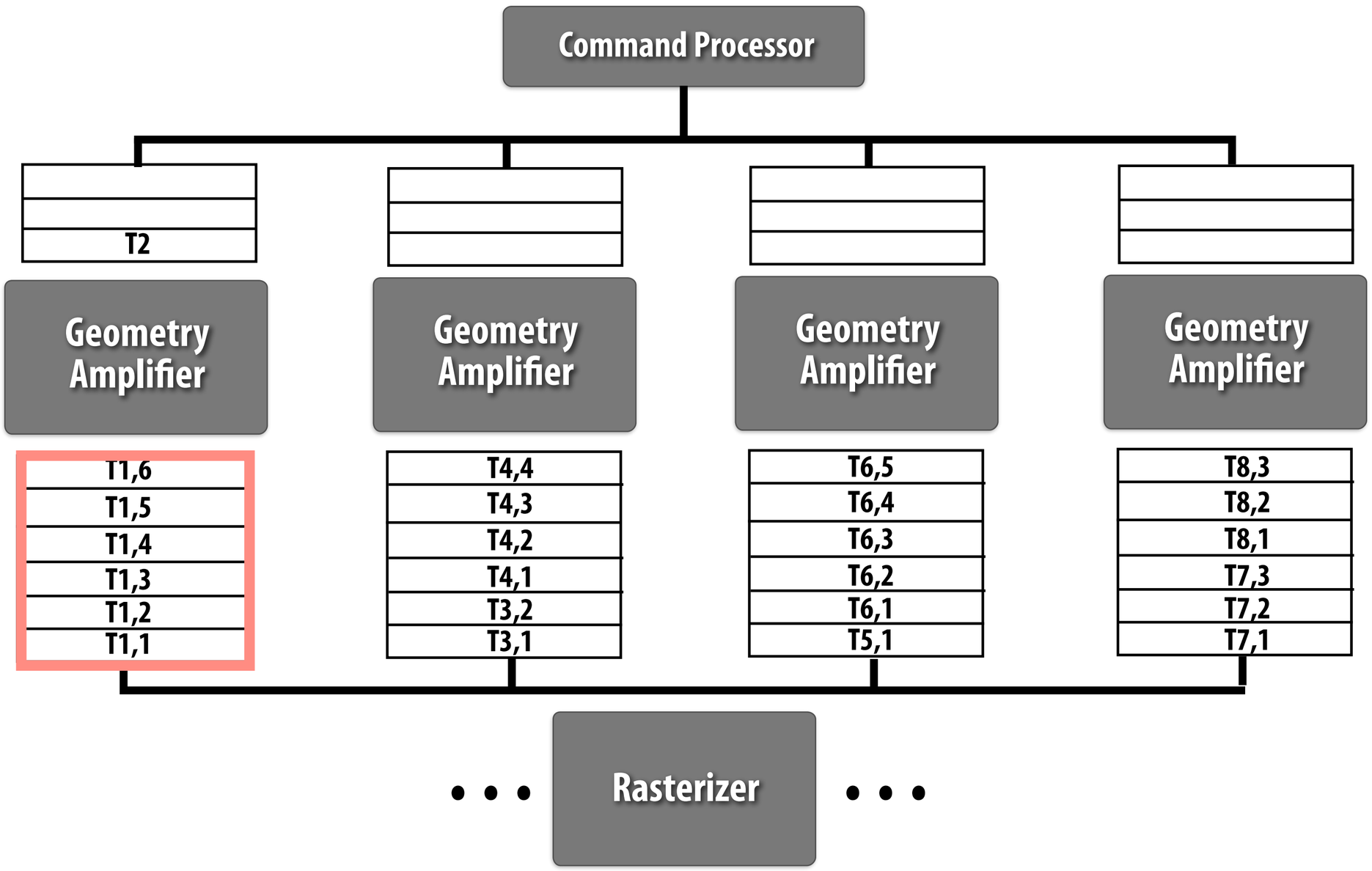
Consider case of large amplification when processing T1



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.**

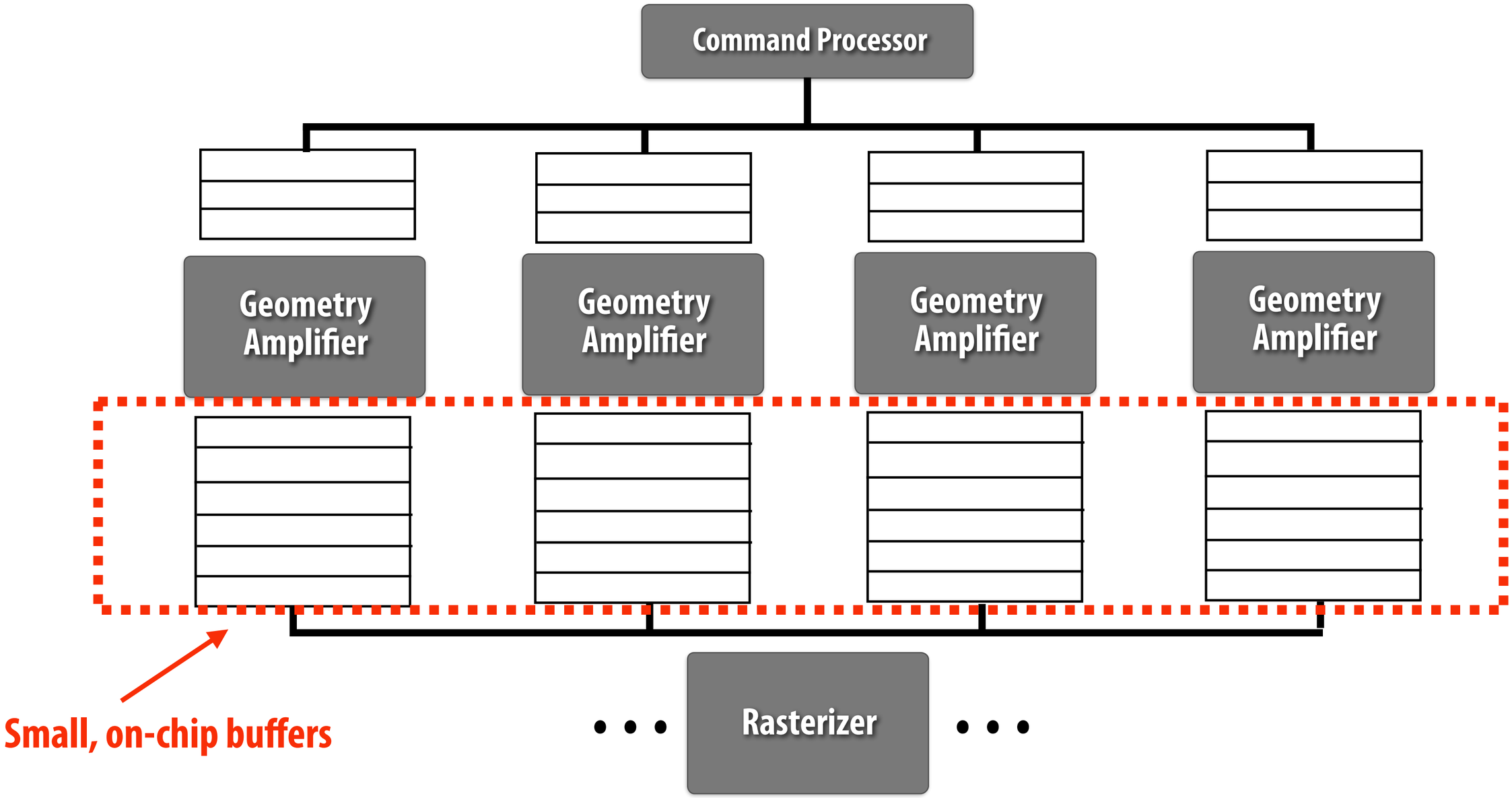
Thought experiment: (class homework) design a scheduling strategy for this case



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.

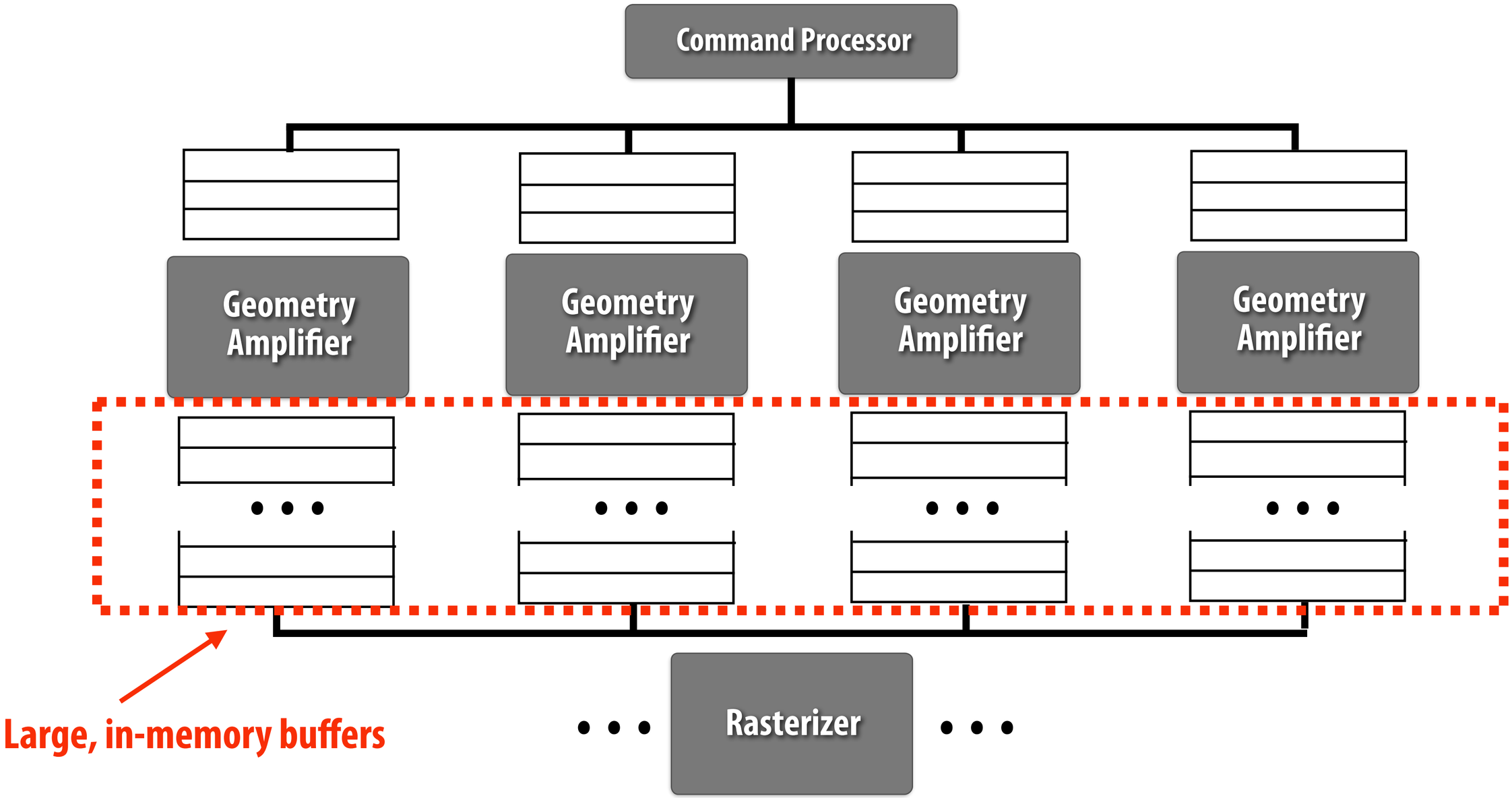
Implementation 1: fixed on-chip storage



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.**

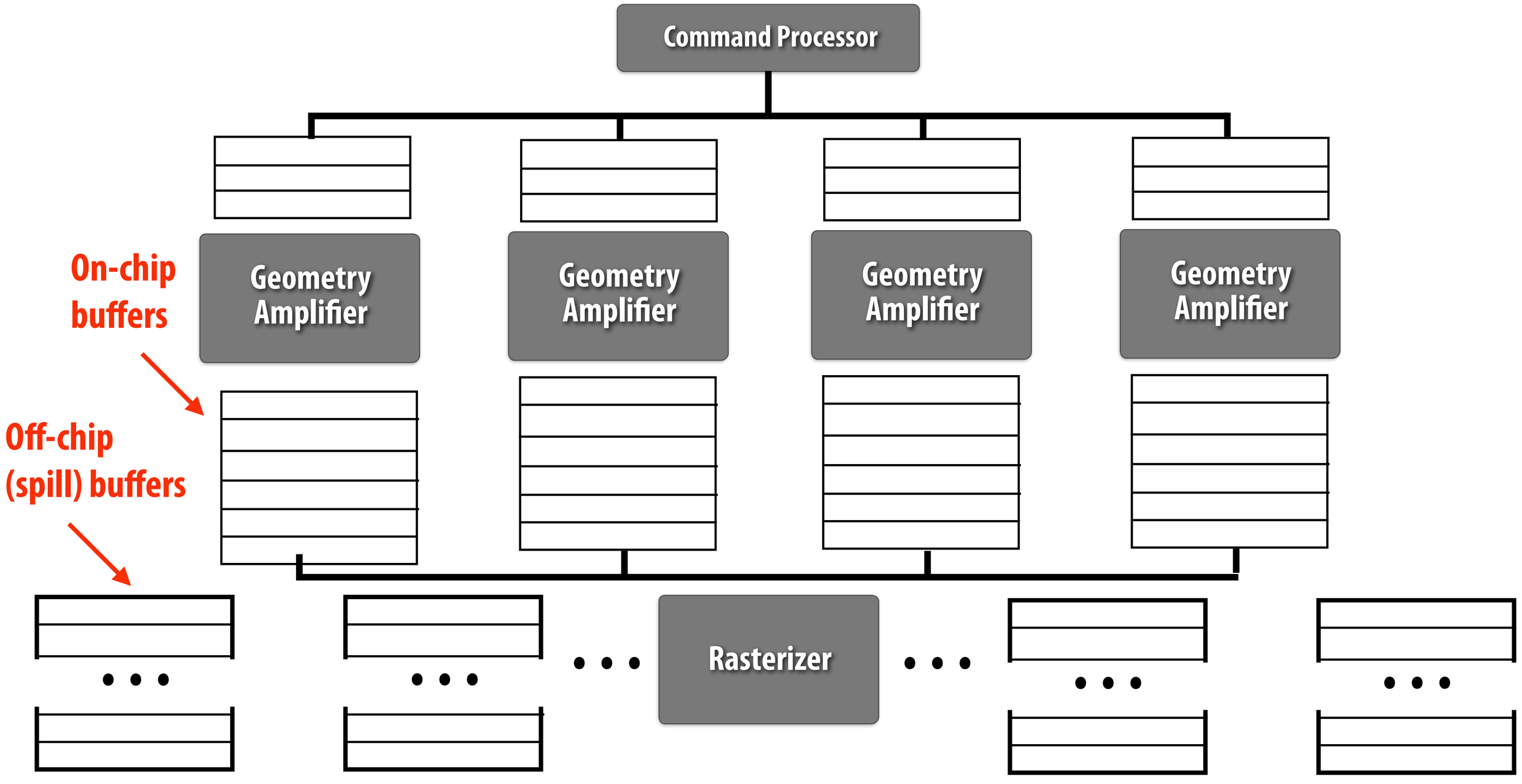
Implementation 2: worst-case allocation



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)**

Readings

- A. R. Smith, *A Pixel is Not a Little Square, a Pixel is Not a Little Square, a Pixel is Not a Little Square!* (and a Voxel is Not a Little Cube) Microsoft Technical Memo, 1995 (to prep for next class)