Lecture 2:

Parallelizing Graphics Pipeline Execution

(+ Basics of Characterizing a Rendering Workload)

Visual Computing Systems

CMU 15-869, Fall 2013

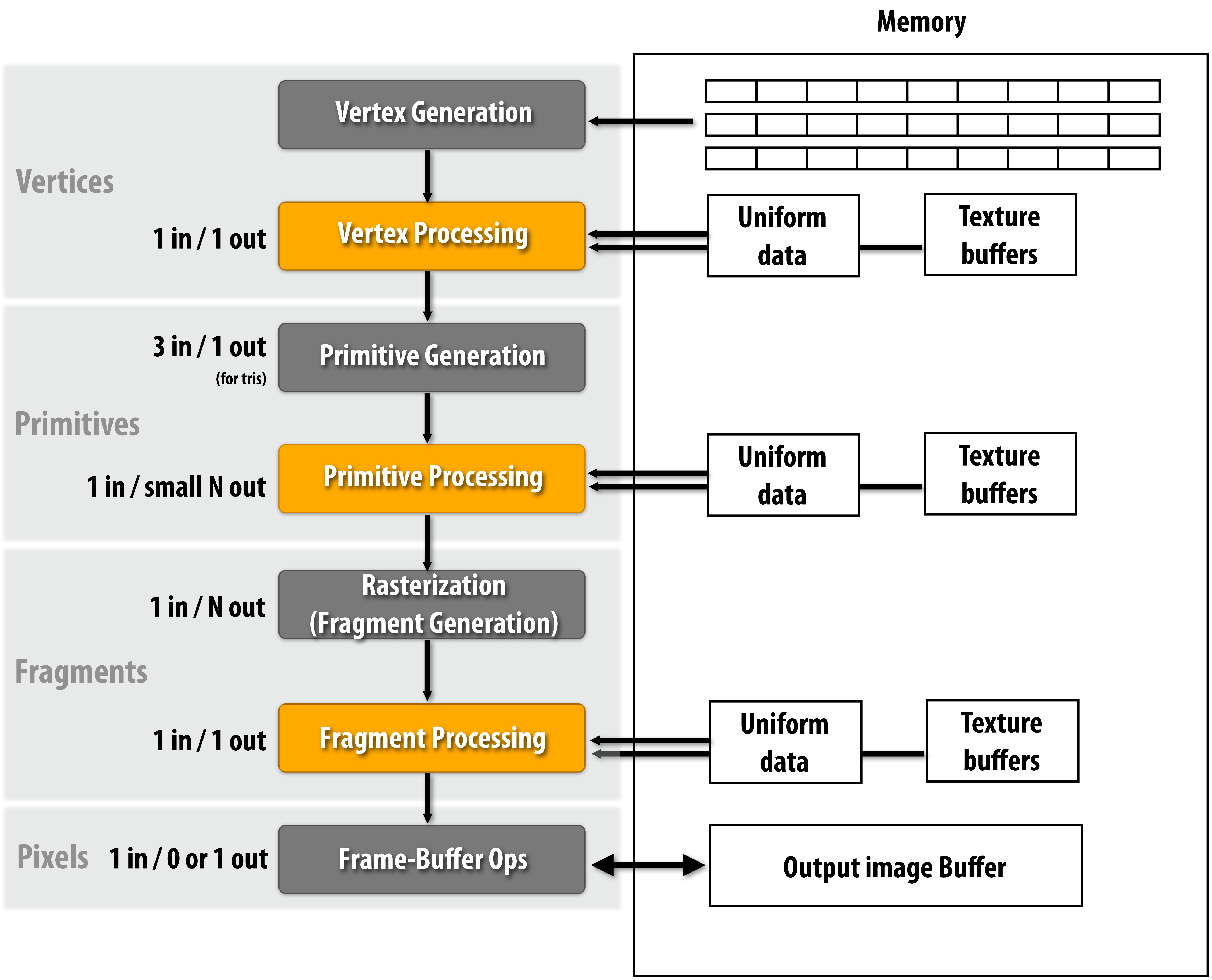
Today

- Finishing up from last time

- Brief discussion of graphics workload metrics

- Strategies for parallelizing the graphics pipeline

The graphics pipeline (last time)



Programming the pipeline (last time)

- **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 shader
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

Note: efficiently managing stage changes is a major challenge in implementations

A series of graphics pipeline commands

State change (set "red" shader)

Draw

State change (set "blue" shader)

Draw

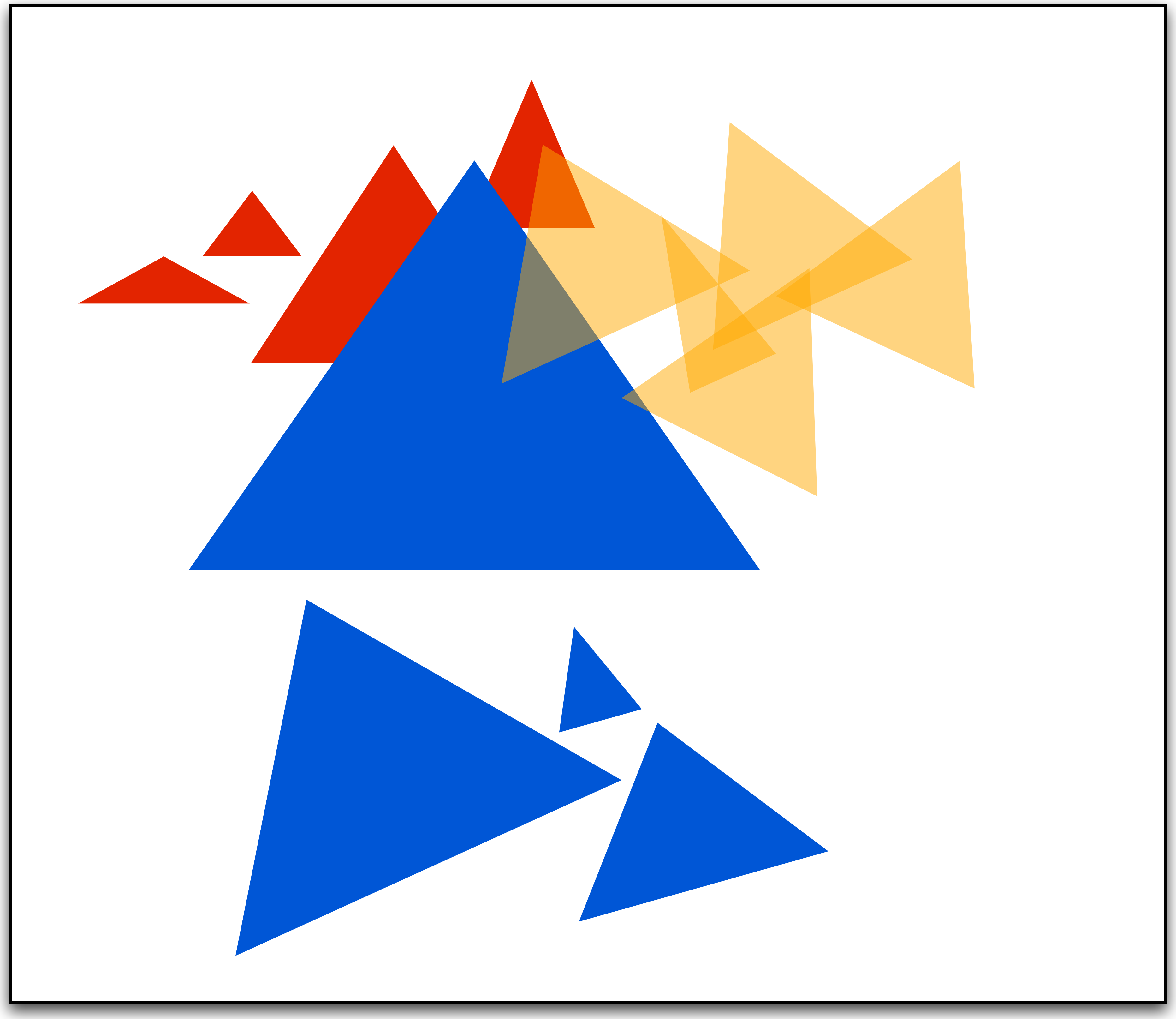
Draw

Draw

State change (change blend mode)

State change (set "yellow" shader

Draw



Using the pipeline to create feedback loops

- Issue draw commands ⟶ output image contents change

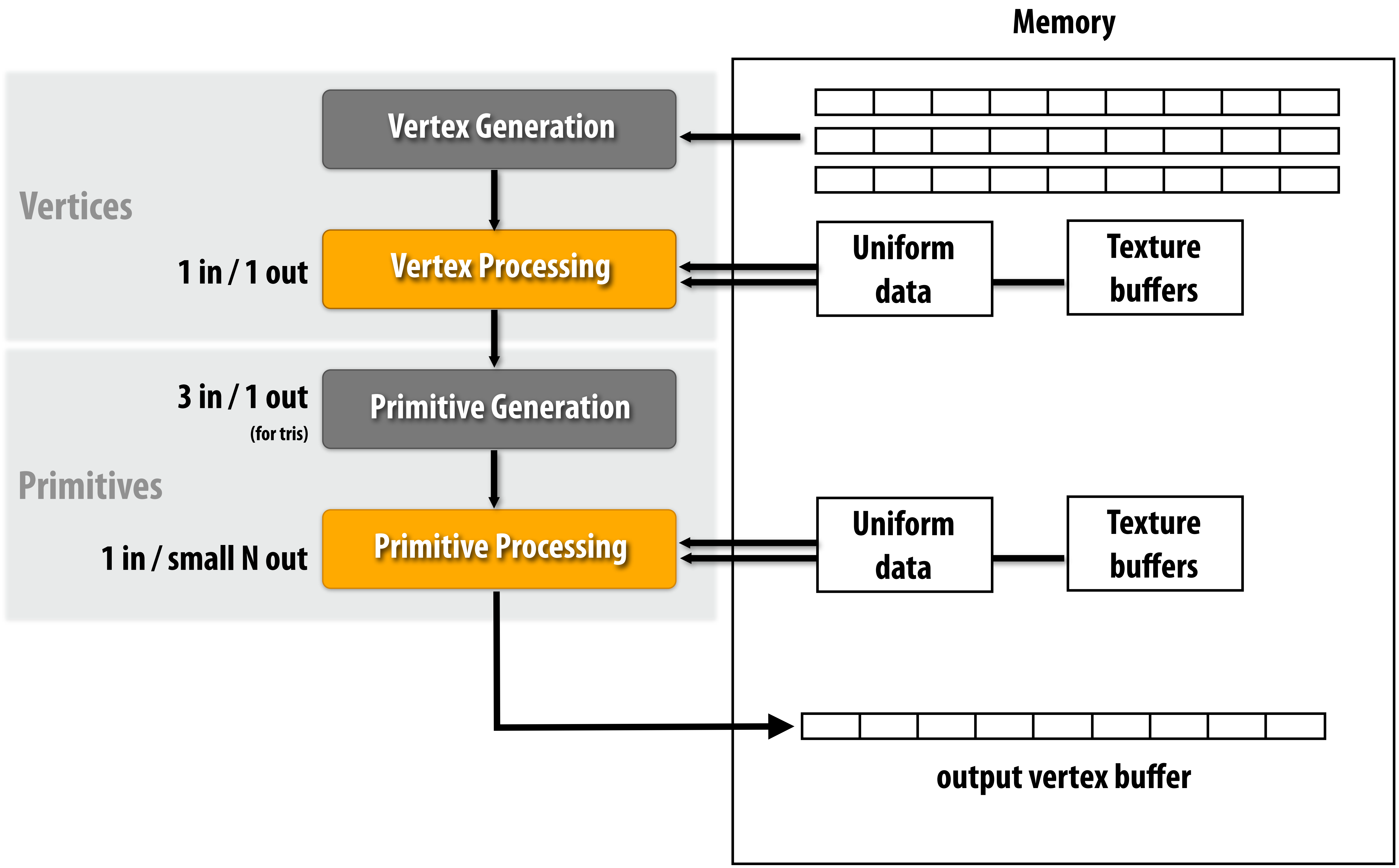
Command Type	Command
Draw	Draw using vertex buffer for object 5
Draw	Draw using vertex buffer for object 6
State change	Bind contents of output image as texture 1
Draw	Draw using vertex buffer for object 5
Draw	Draw using vertex buffer for object 6
	⋮

Key idea for:
shadows
environment mapping
post-processing effects

Modern games: 1000-1500 draw calls per frame
(source: Johan Andersson, DICE -- circa 1998)

Feedback loop: save intermediate geometry

- Issue draw commands ⟶ save intermediate geometry



Graphics pipeline characteristics

- **Level of abstraction**
  - Imperative abstraction, not declarative
    (Application says "draw these triangles, using this fragment shader, with depth testing on" rather than "draw a cow made of marble on a sunny day")
  - <u>Programmable</u> stages give large amount of application flexibility
    (e.g., to implement wide variety of materials and lighting techniques)
  - <u>Configurable</u> (but not programmable) pipeline structure: turn stages on and off, create feedback loops
  - Abstraction low enough to allow application to implement many techniques, but high enough to abstract over radically different GPU implementations

Orthogonality of abstractions

- **All vertices treated the same regardless of primitive type**
  - Vertex programs oblivious to primitive types
  - The same vertex program works for triangles and lines

- **All primitives are converted into fragments for per-pixel shading and frame-buffer operations**
  - Fragment programs oblivious to primitive type and the behavior of the vertex program *
  - Z-buffer is a common representation used to perform occlusion for any primitive that can be converted into fragments

* Almost oblivious. Vertex shader must make sure it passes along all inputs required by the fragment shader

What the pipeline DOES NOT do (non-goals)

- Pipeline has no concept of lights, materials, modeling transforms
  - Only vertices, primitives, fragments, pixels, and STATE
    (state examples: buffers, shaders, and config parameters)
  - Applications use these basic abstractions to implement lights, materials, etc.
- Pipeline has no concept of a scene
- No I/O or OS window management

Perspective from Kurt Akeley

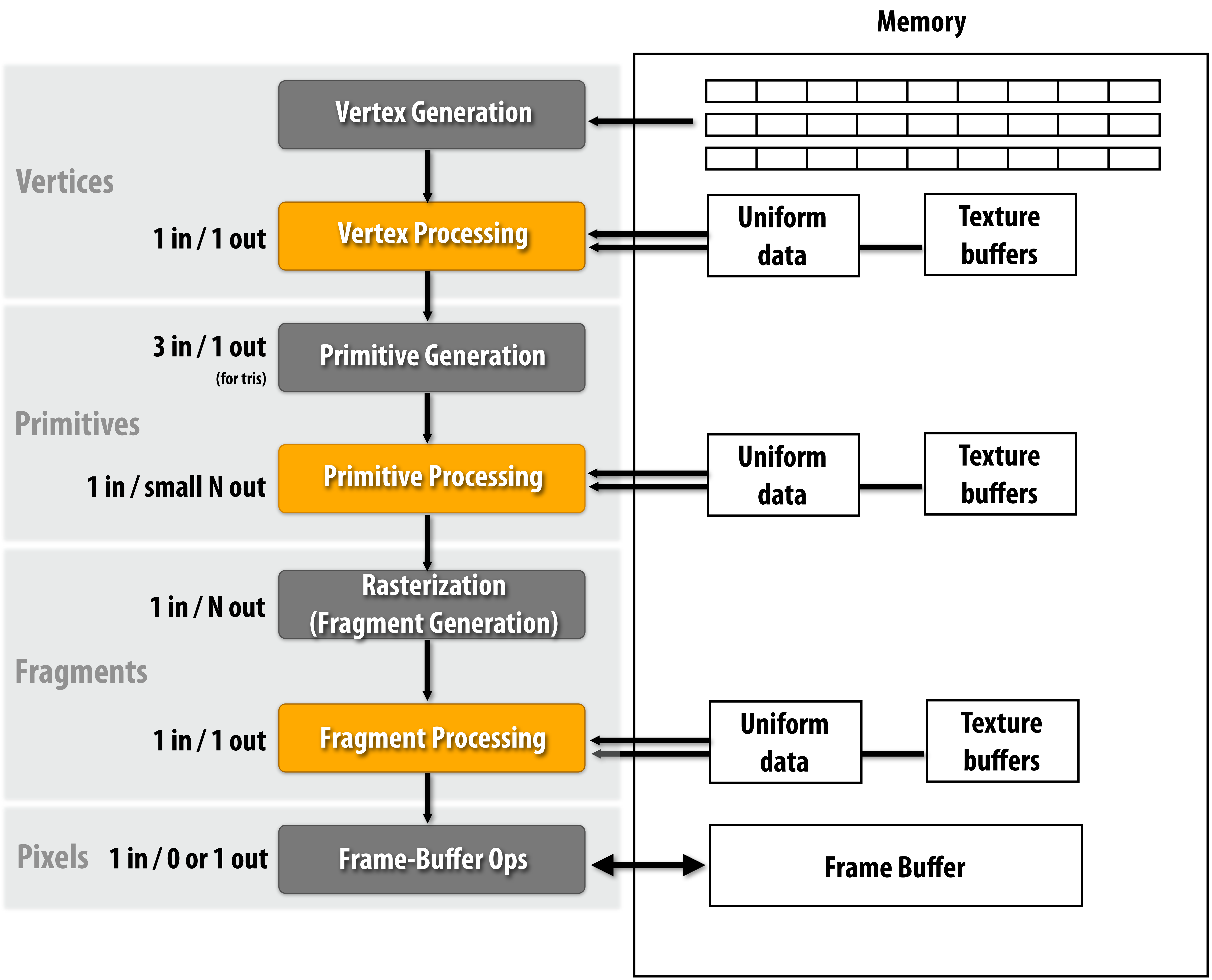
- Does the system meet original design goals, and then do much more than was originally imagined?
  - Simple, orthogonal concepts often yield an amplifier effect

- Often you've done a good job if neither system implementers nor system users are perfectly happy ;-)

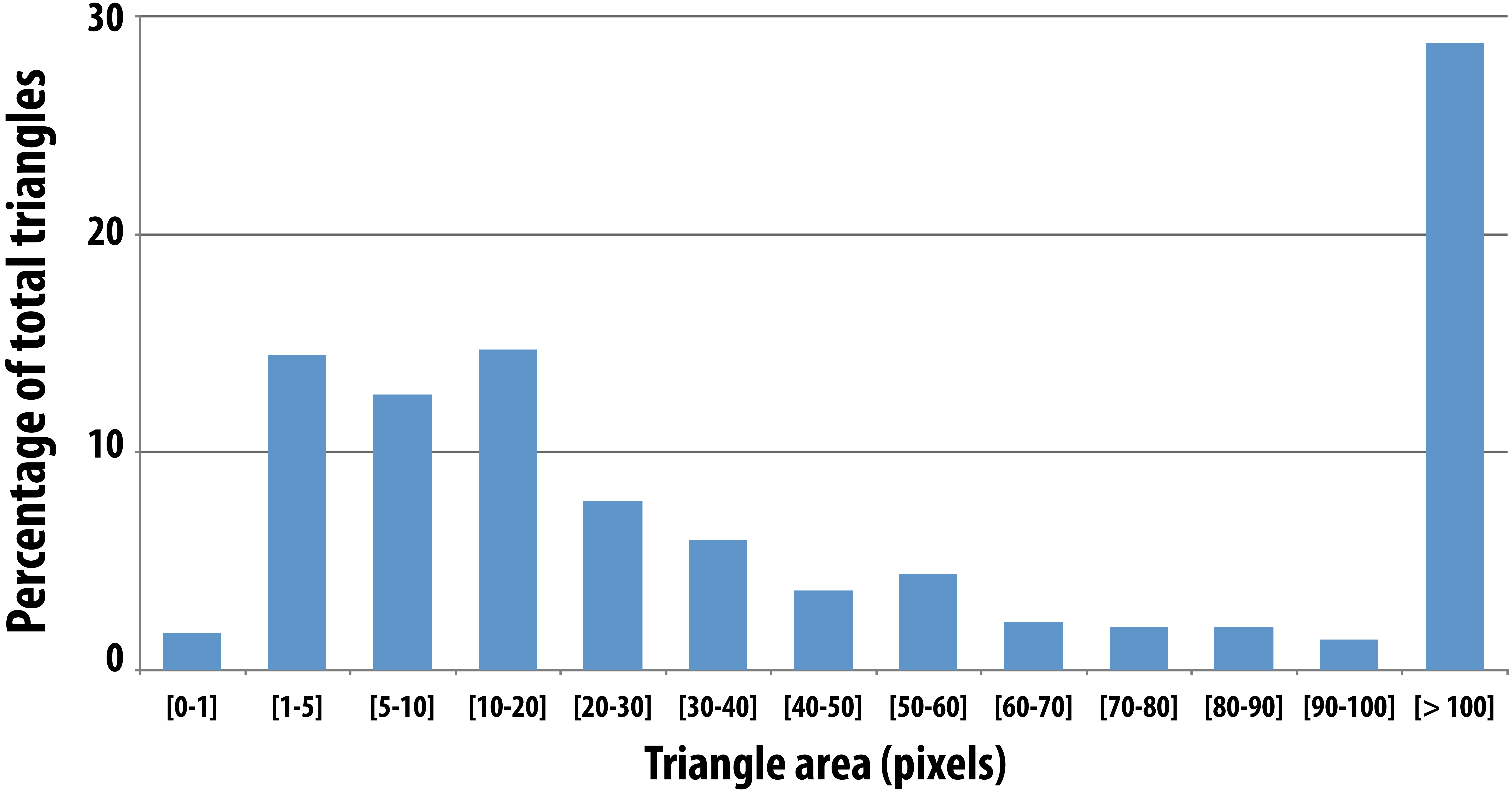
  (of course, you still have to meet design goals)

Analyzing a 3D Graphics Workload

Where is most of the work done?



Triangle size



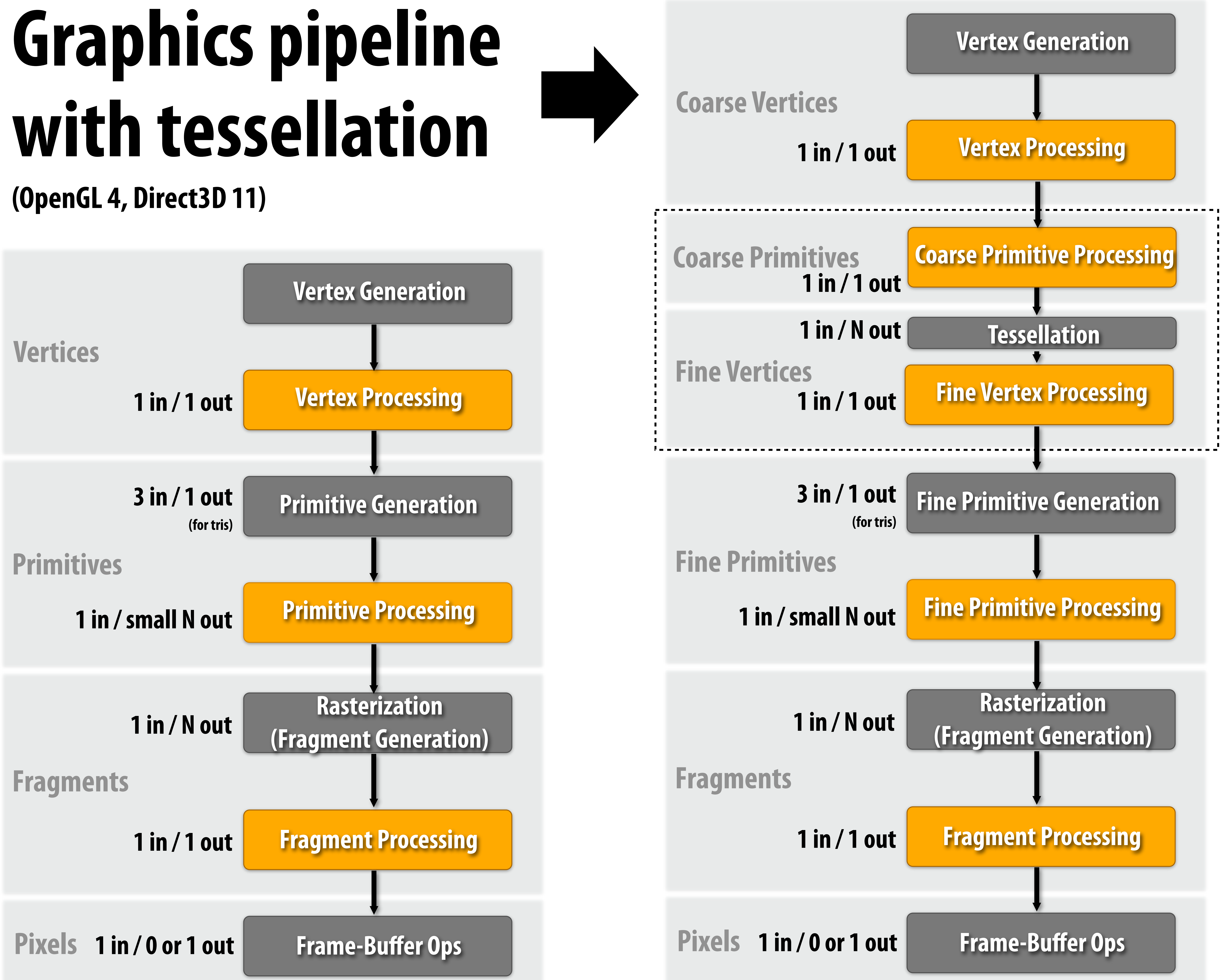
[source: NVIDIA]

Note: tessellation is triggering a reduction in triangle size

adidas
Carlsberg
Credit: Pro Evolution Soccer 2010 (Konami)

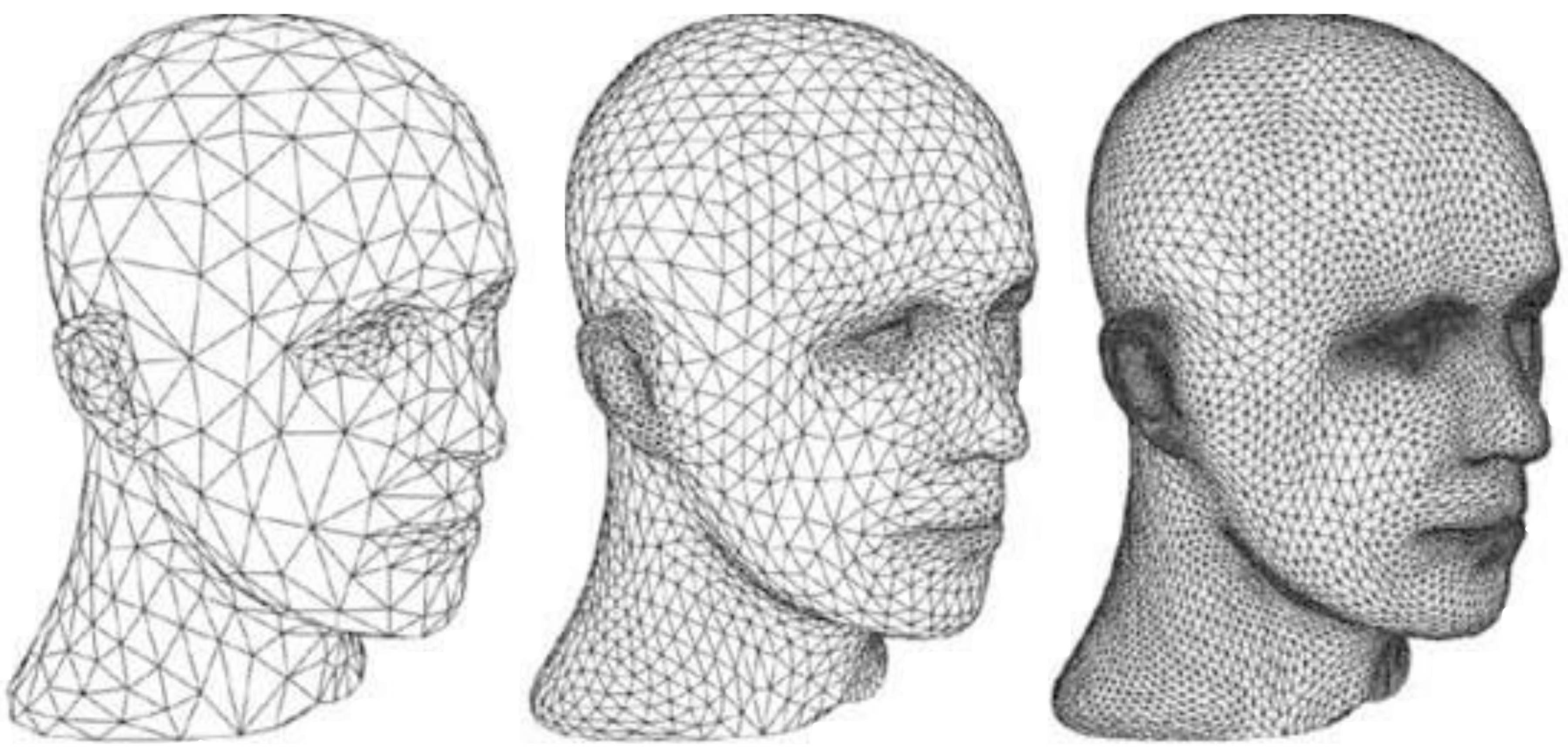
Graphics pipeline with tessellation

(OpenGL 4, Direct3D 11)



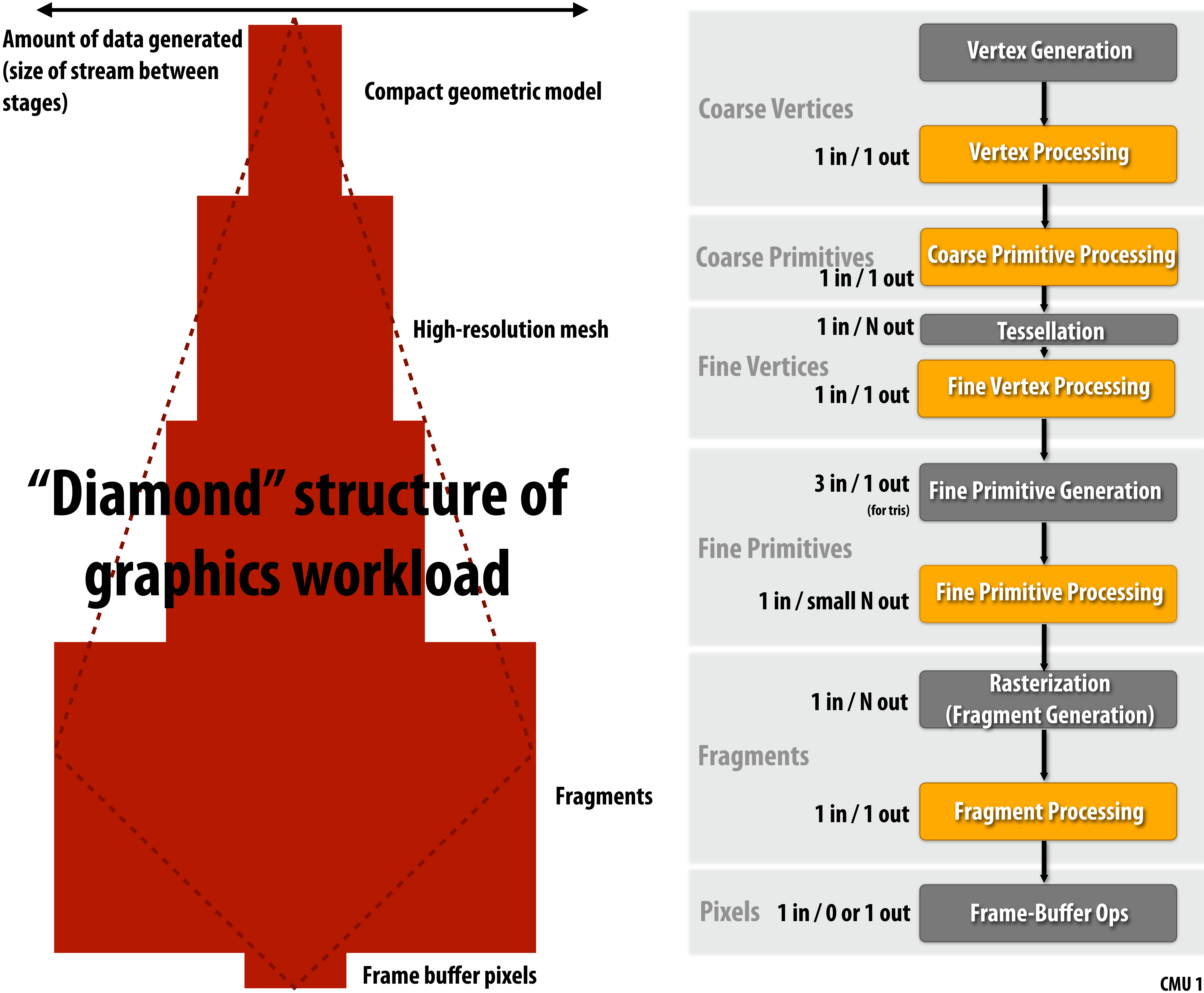
Tessellation

- **Generate fine triangle mesh from coarse mesh representation**



[Image credit: NVIDIA]

"Diamond" structure of graphics workload



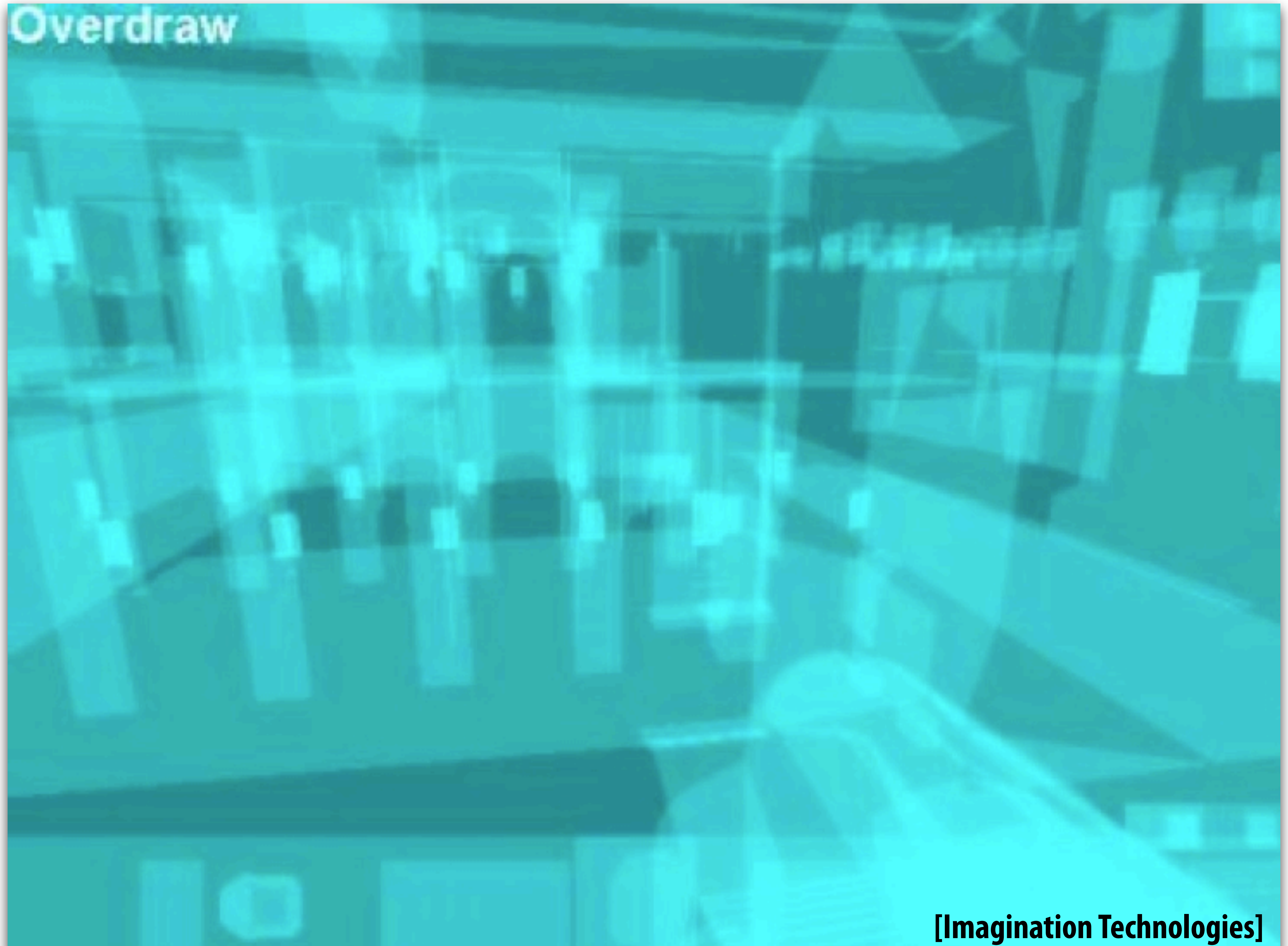
Key 3D graphics workload metrics

- Data amplification from stage to stage
  - Triangle size (amplification in rasterizer)
  - Expansion by geometry shader (if enabled)
  - Tessellation factor (if tessellation enabled)

- [Vertex/fragment] shader cost (how many instructions?)

- Scene depth complexity
  - Determines number of Z/color buffer writes

Scene depth complexity



Very rough approximation: $TA = SD$

T = # triangles

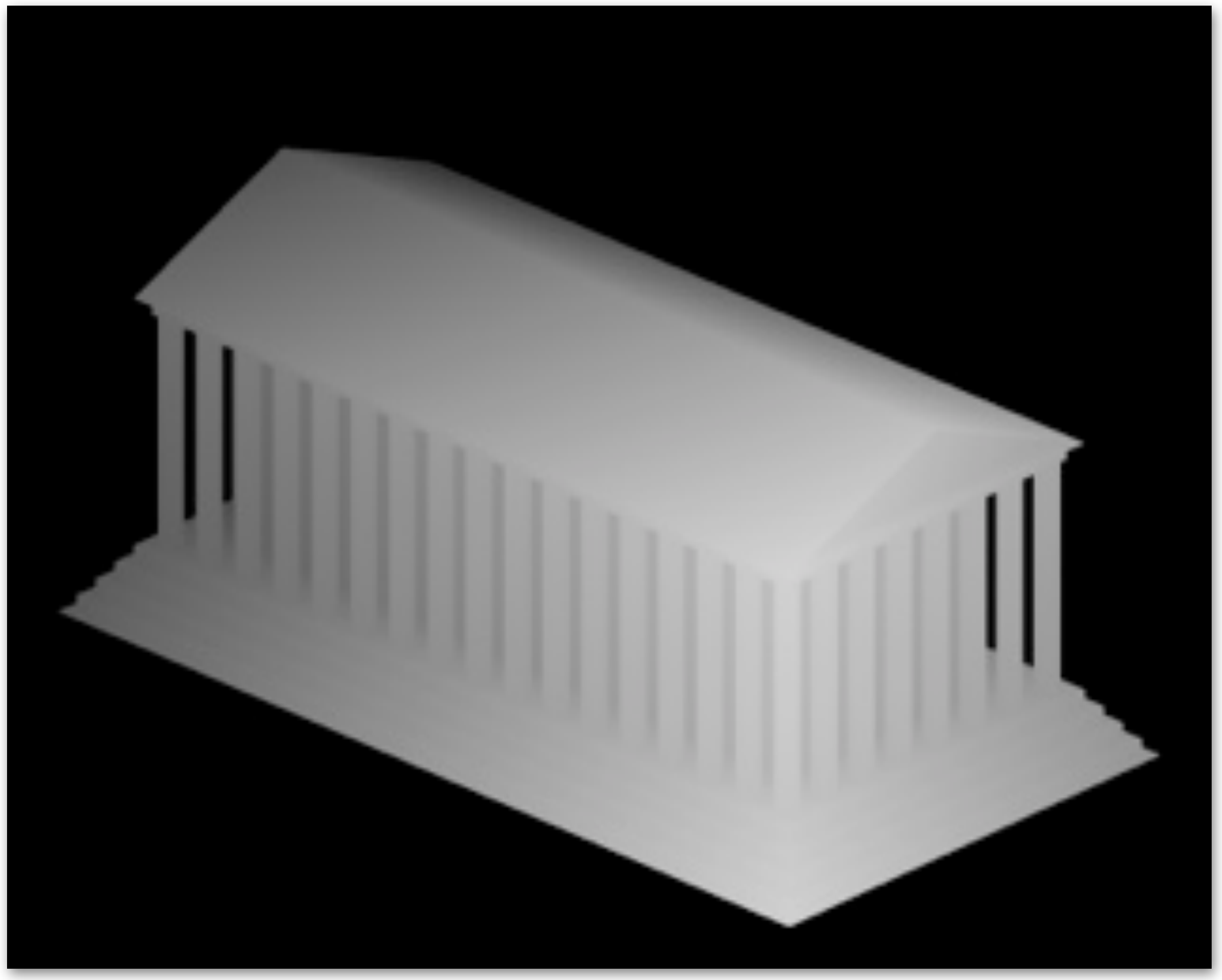
A = average triangle area

S = pixels on screen

D = average depth complexity

Graphics pipeline workload changes rapidly

- Triangle size is scene and frame dependent
  - Move far away from an object, triangles get smaller
  - Even object-dependent within a frame (characters: higher resolution meshes)
- Varying complexity of materials, different number of lights illuminating surfaces
  - No such thing as an "average" shader
  - Tens to several hundreds of instructions per shader
- Stages can be disabled
  - Shadow map creation = NULL fragment shader
  - Post-processing effects = no vertex work
- Recall: thousands of draw calls per frame



Example: rendering a "depth map" requires vertex shading but no fragment shading

Parallelizing the Graphics Pipeline

Select slides credit Kurt Akeley and Pat Hanrahan
(Stanford CS448 Spring 2007)

Reminder: requirements + workload challenges

- Immediate mode interface: pipeline accepts sequence of commands
  - Draw commands
  - State modification commands

- Processing of 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., triangle size)

- Ample opportunities for parallelism
  - Few dependencies (most notable: order, R-M-W frame-buffer update)

Parallelism and communication

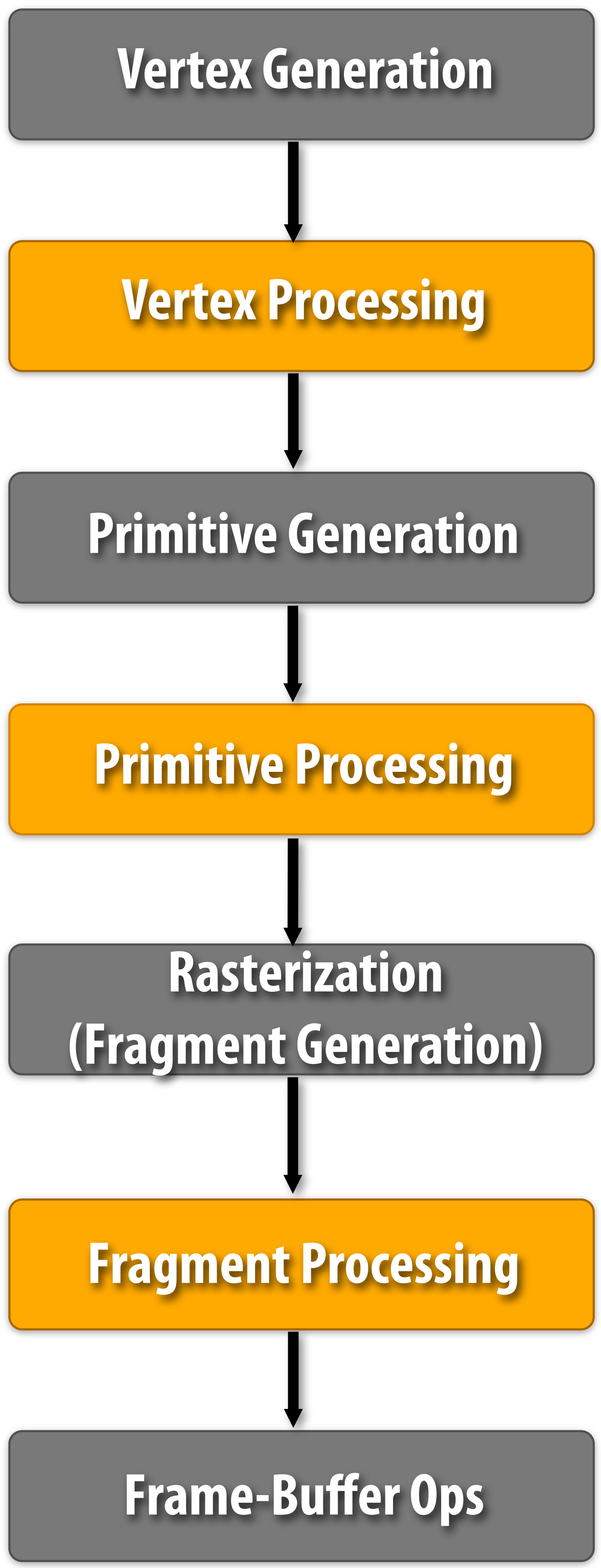
- Parallelism - using multiple execution units to process work in parallel
- Communication - parallel execution units must synchronize and communicate to cooperatively perform a rendering task
  - Communication between execution units
  - Communication between execution units and memory
- Big issues:
  - Correctness (preserving sequential semantics)
  - Achieving good workload balance (using all processors)
  - Minimizing communication/synchronization
  - Avoiding unnecessary work

Opportunities for parallelism in graphics

- **Data parallelism**
  - Simultaneously execute same operation on different data
  - Object space entities (vertices, primitives, etc.)
  - Image space entities (fragments, pixels)

- **Pipeline task parallelism**
  - Simultaneously execute different tasks on similar (or different) data
  - Vertex processing, rasterization, fragment processing

Simple parallelization (pipelined)

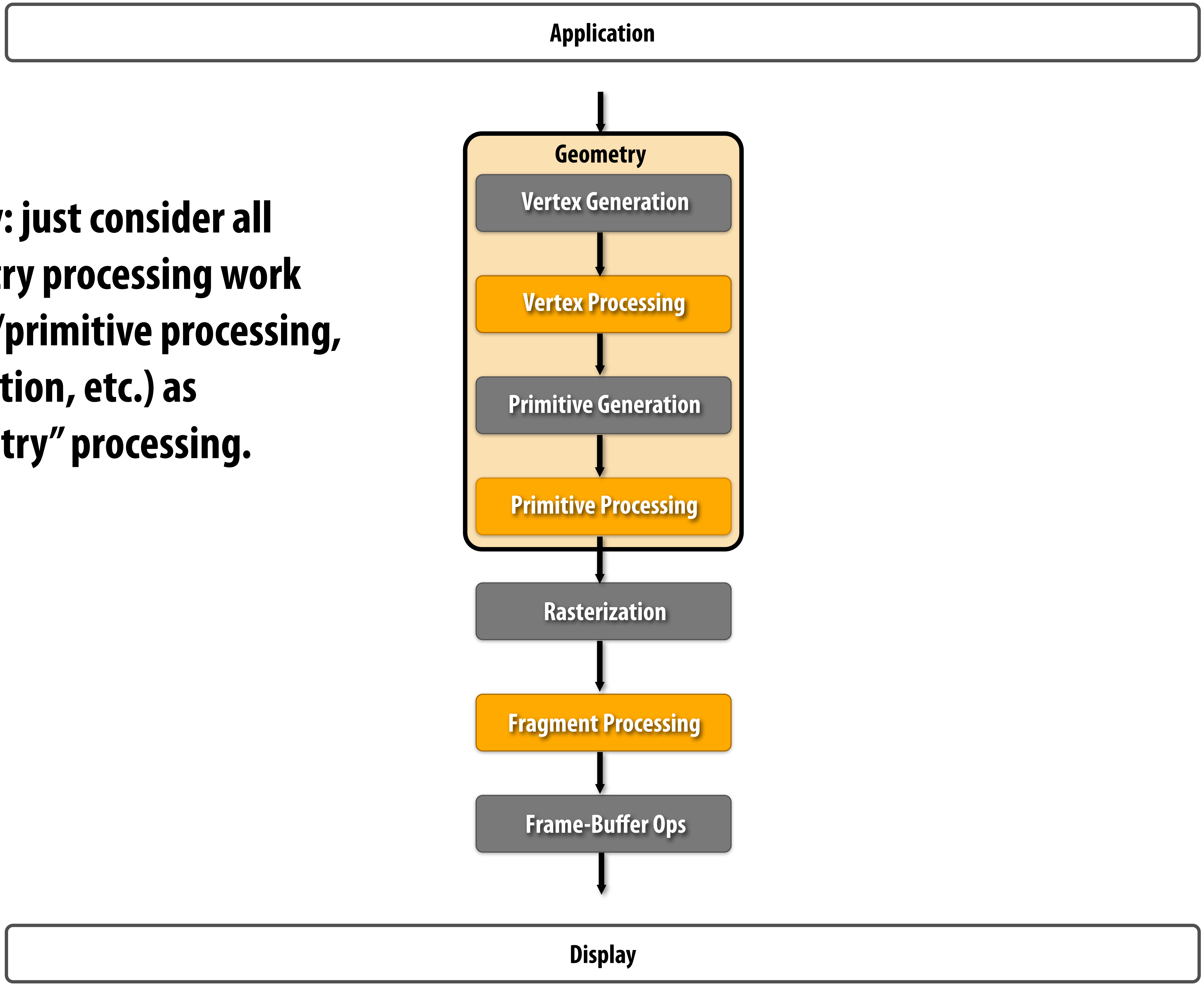


Separate hardware unit for each stage

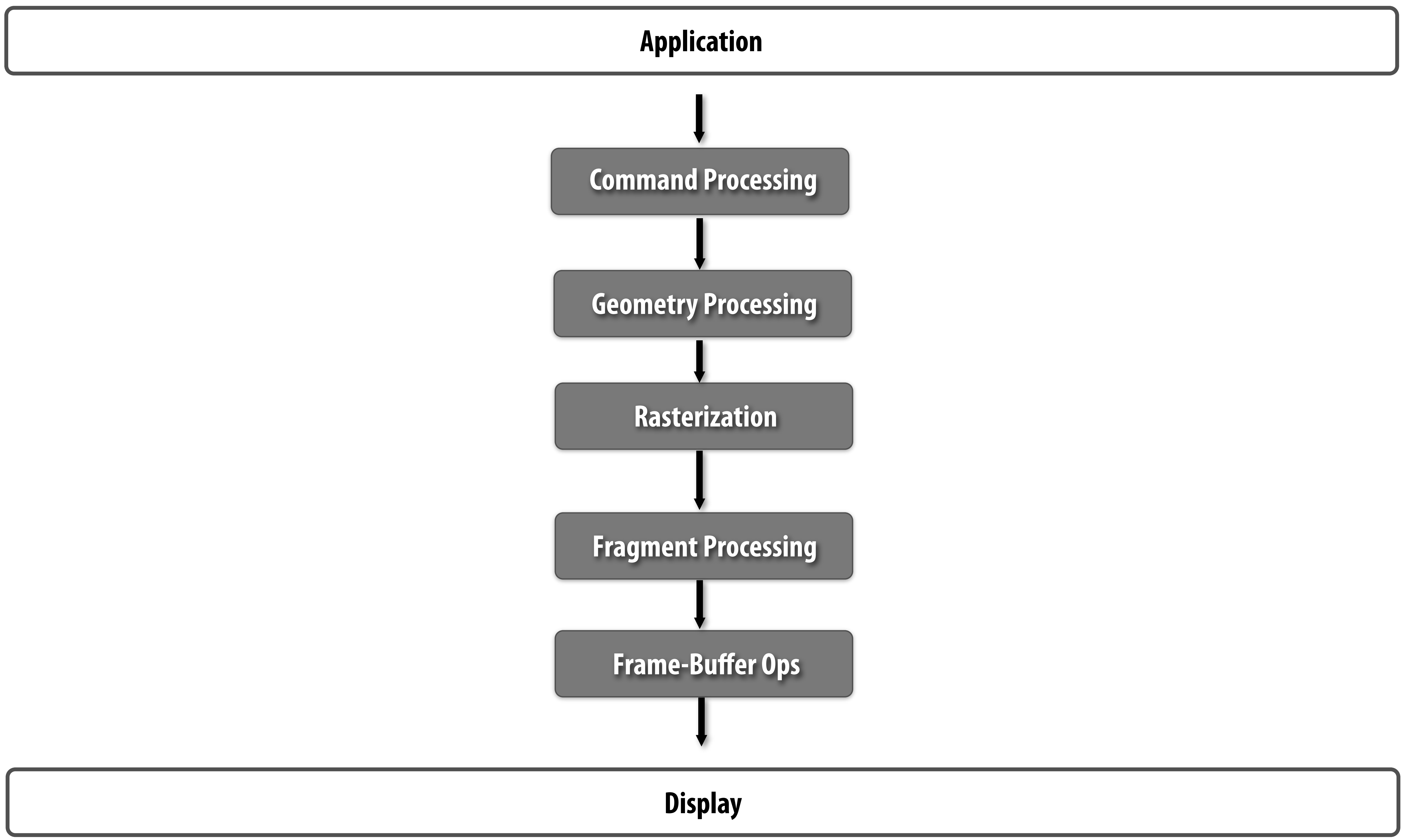
Speedup?

Simplified pipeline

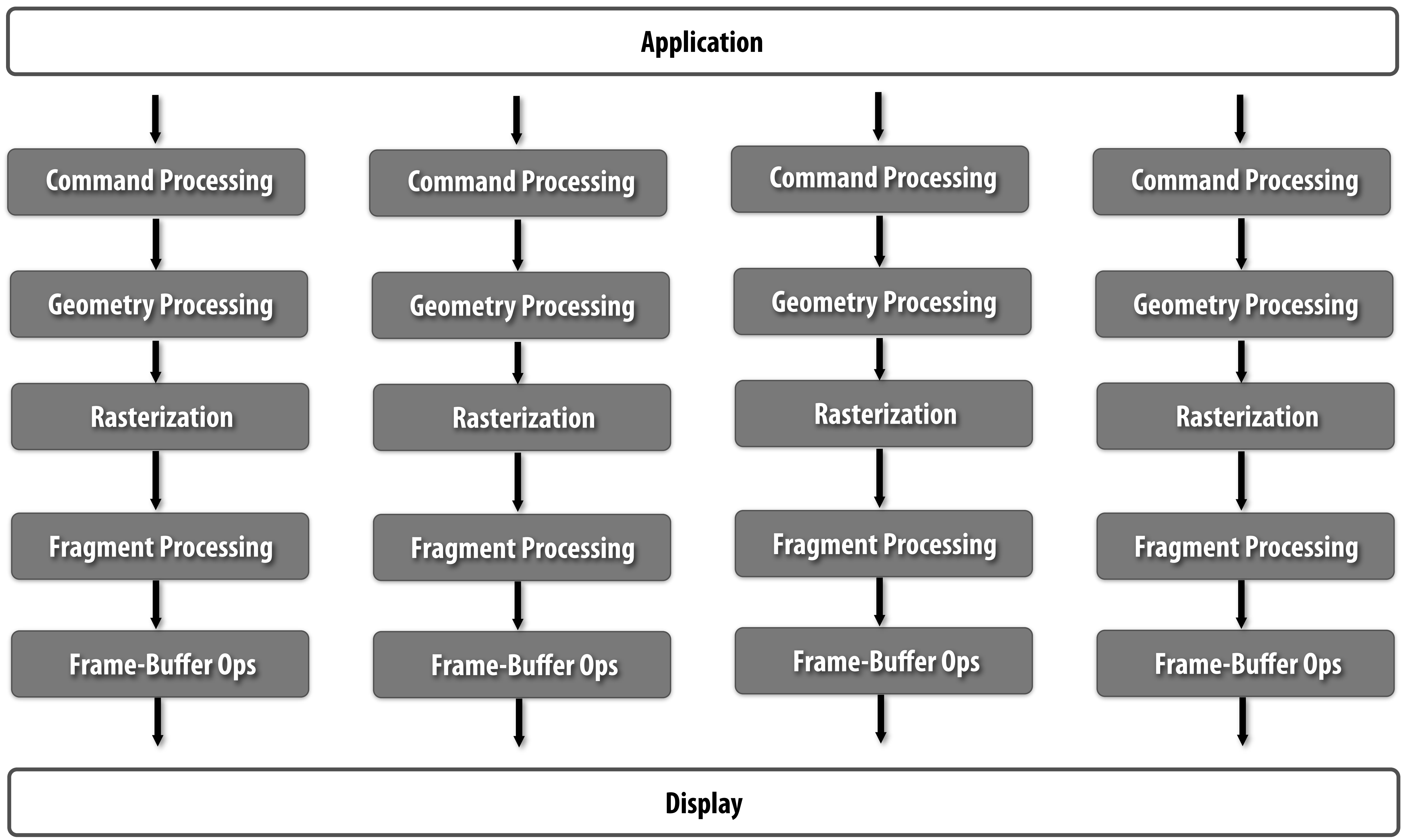
For now: just consider all geometry processing work (vertex/primitive processing, tessellation, etc.) as "geometry" processing.



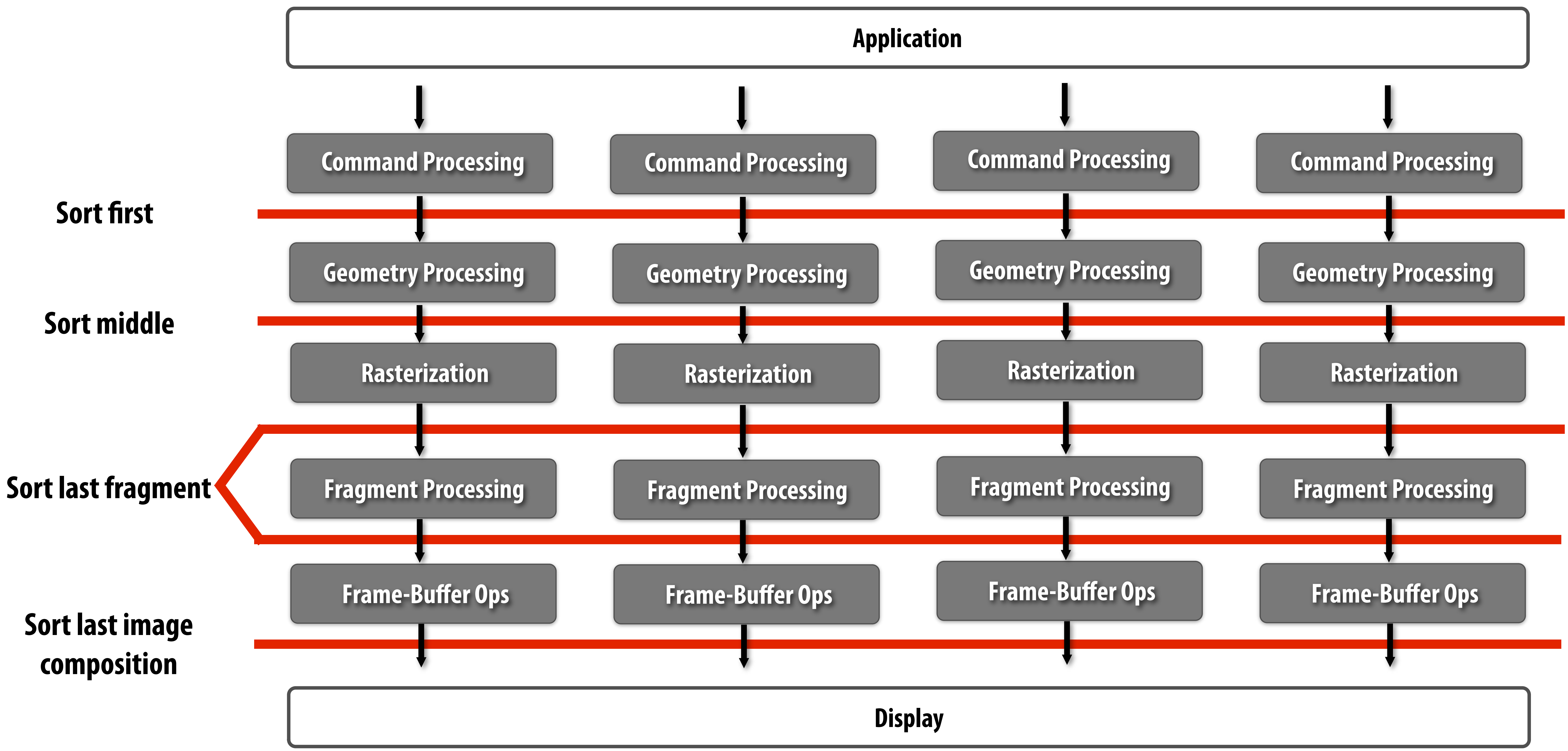
Simplified pipeline



Scaling "wide"

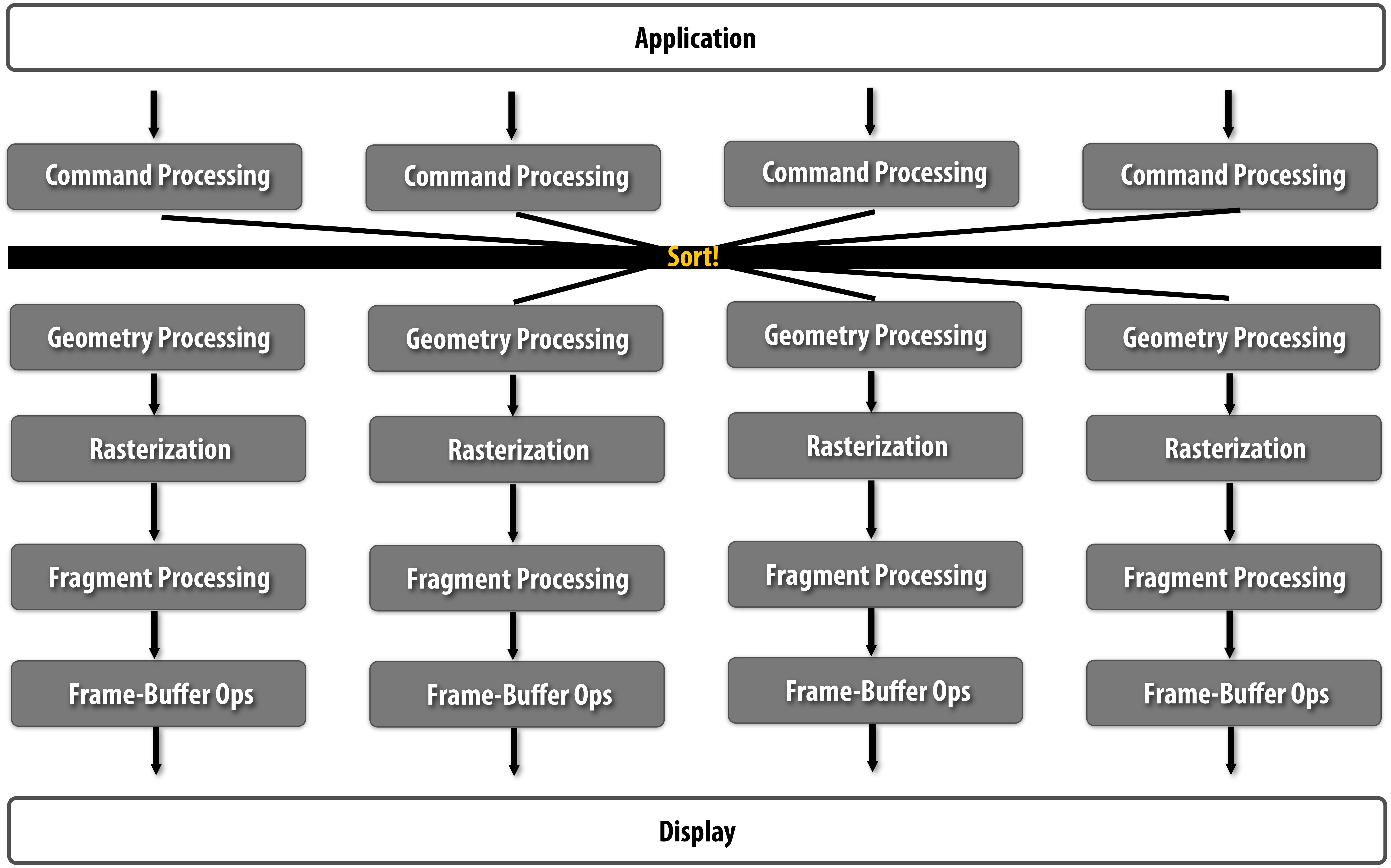


Sorting taxonomy



Sort first

Sort first

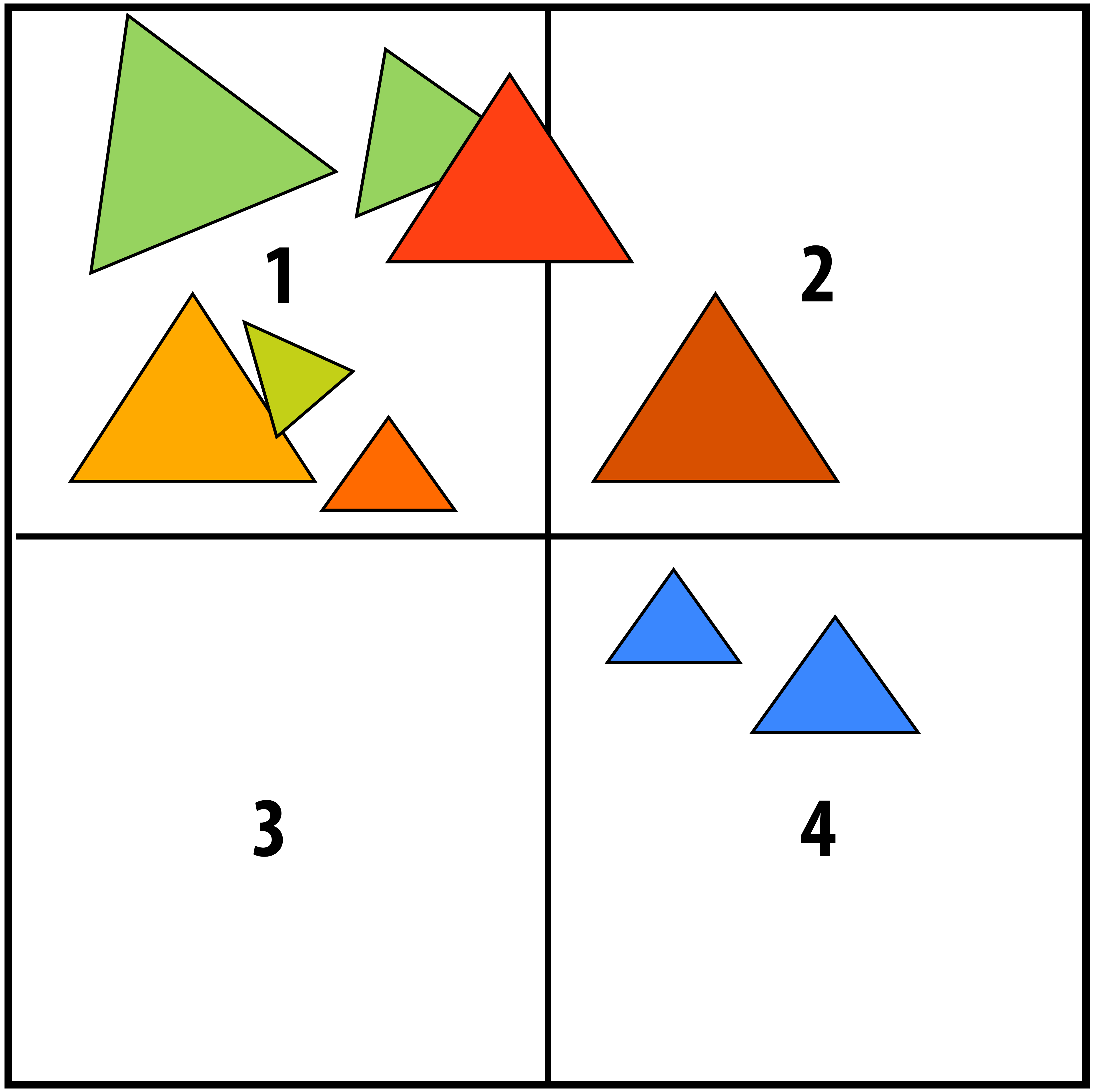


Assign each hardware pipeline a region of the render target

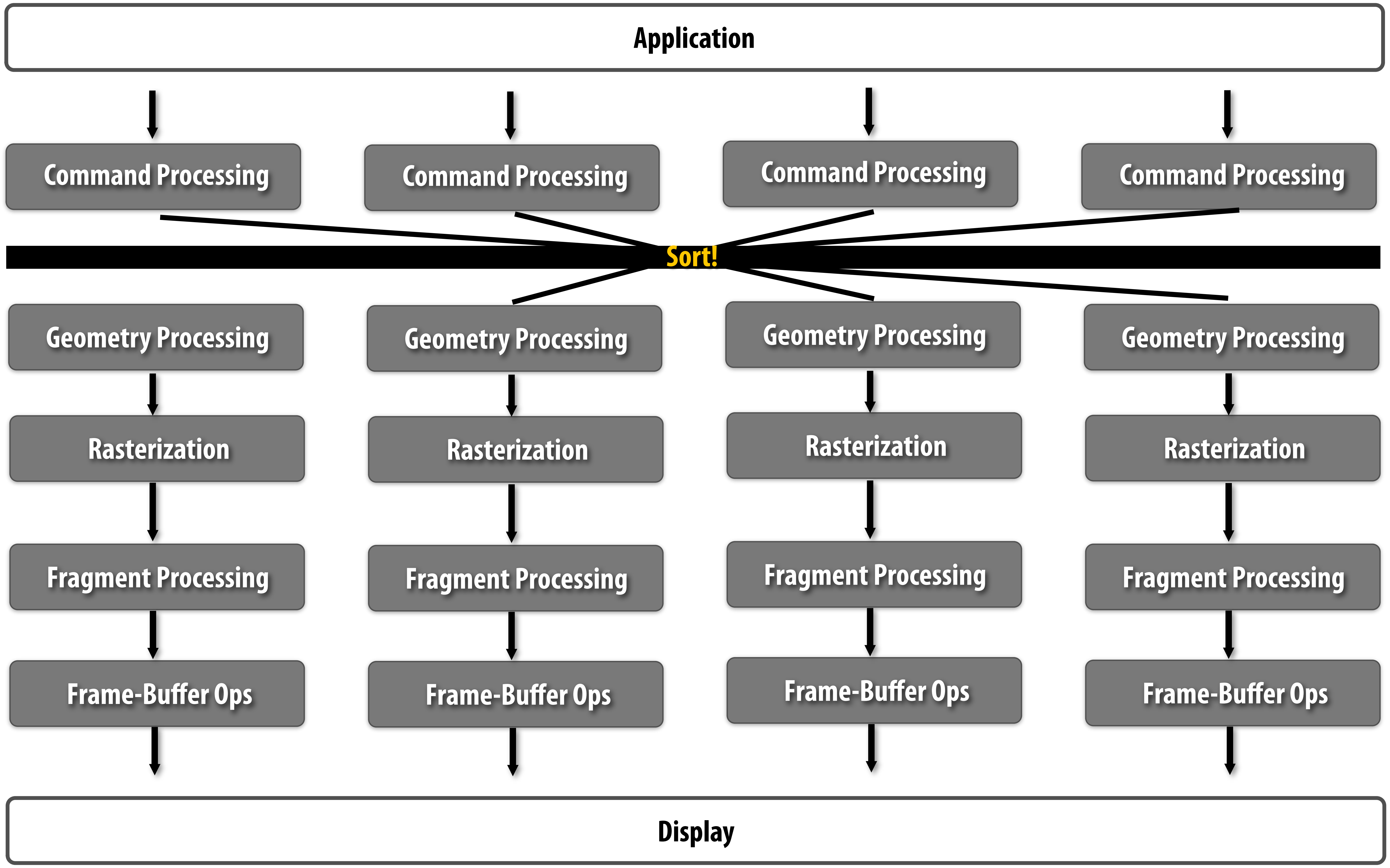
Do minimal amount of work to determine which region(s) input primitive overlaps

Sort first work partitioning

(partition the primitives)

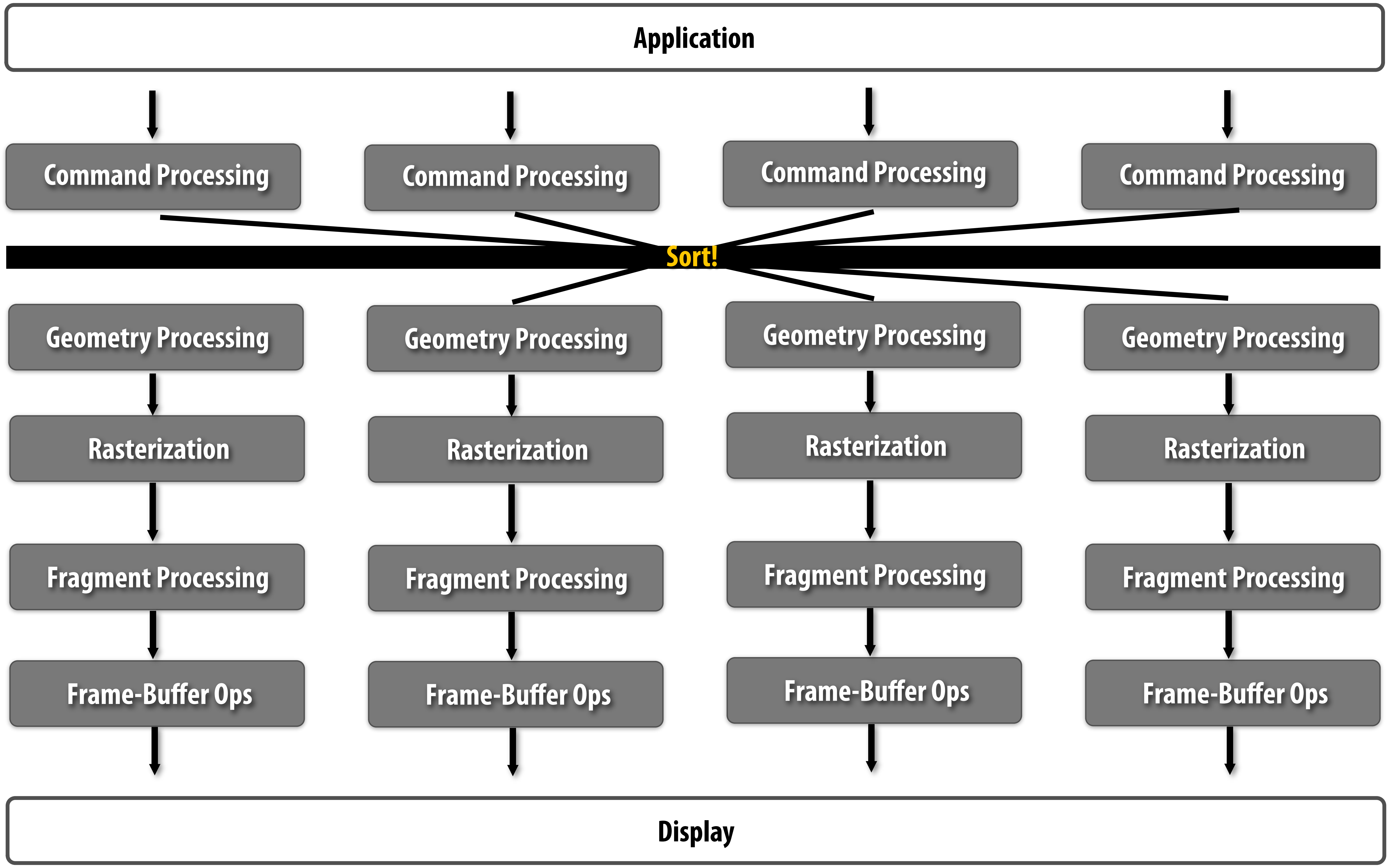


Sort first



- **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)
  - Easy early fine occlusion cull ("early z")

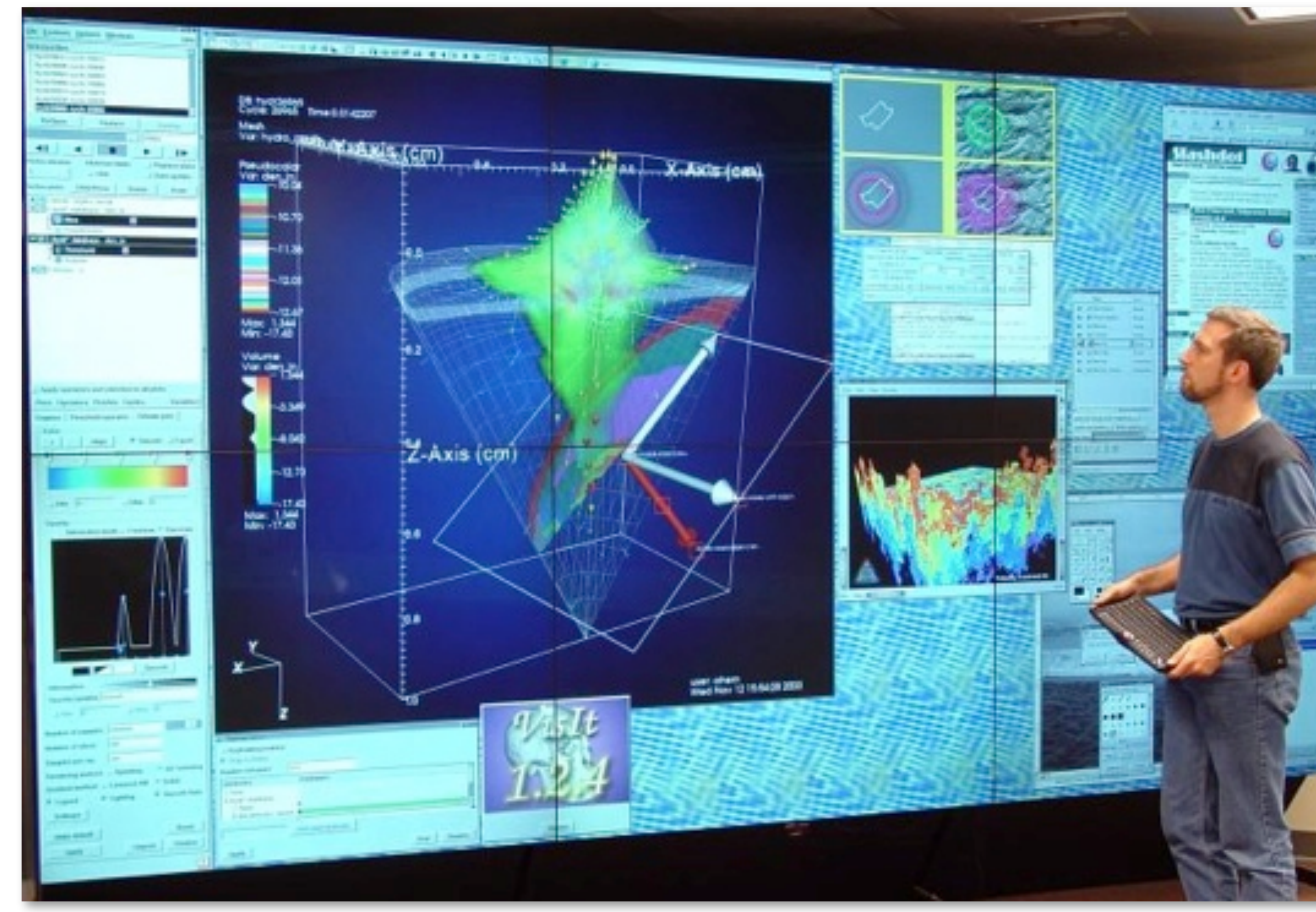
Sort first



- **Bad:**
  - Potential for workload imbalance (one part of screen contains most of scene)
  - Extra cost of triangle "pre-transformation" (do some vertex work twice)
  - "Tile spread": as screen tiles get smaller, primitives cover more tiles (duplicate geometry processing across the parallel pipelines)

Sort-first examples

- **WireGL/Chromium* (parallel rendering with a cluster of GPUs)**
  - **"Front-end" sorts primitives to machines**
  - **Each GPU is a full rendering pipeline**



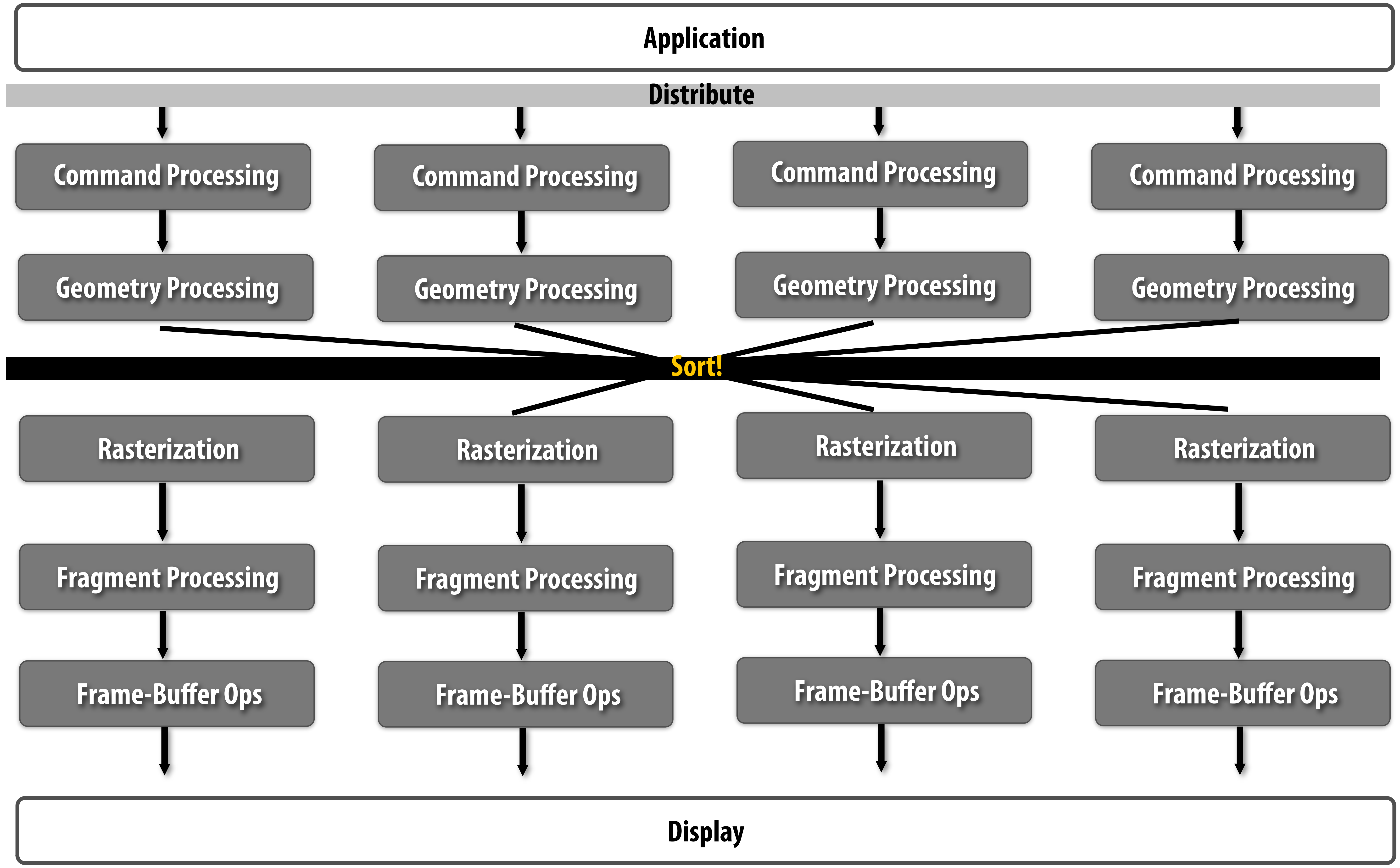
- **Pixar's RenderMan (implementation of REYES)**
  - **Multi-core software renderer**
  - **Sort surfaces into tiles prior to tessellation (sort the surfaces, not all the little "micropolygons")**

PIXAR's RenderMan

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

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

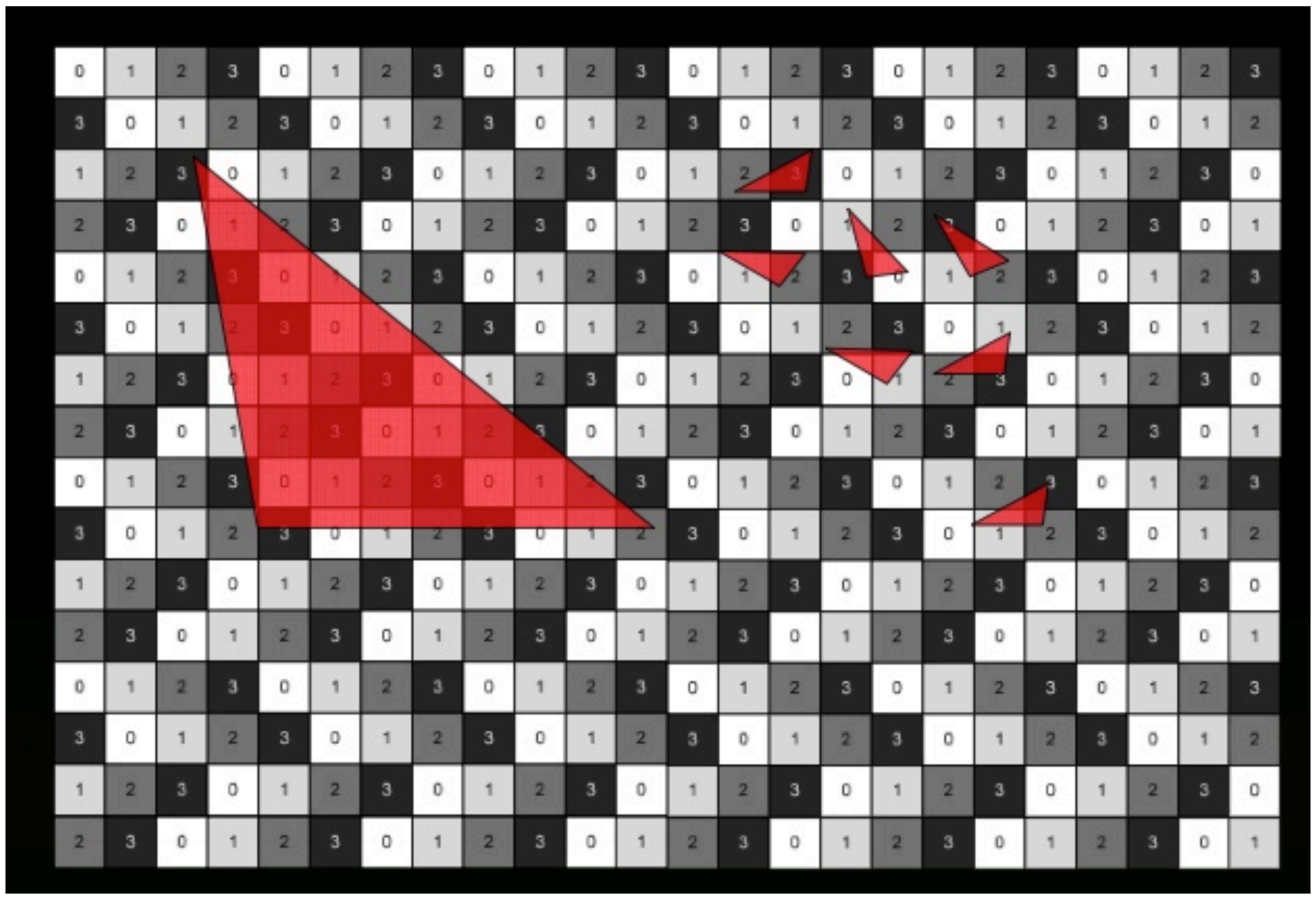
- Decrease chance of one rasterizer processing most of scene
- Most triangles overlap multiple screen regions (often overlap all)

Interleaved mapping

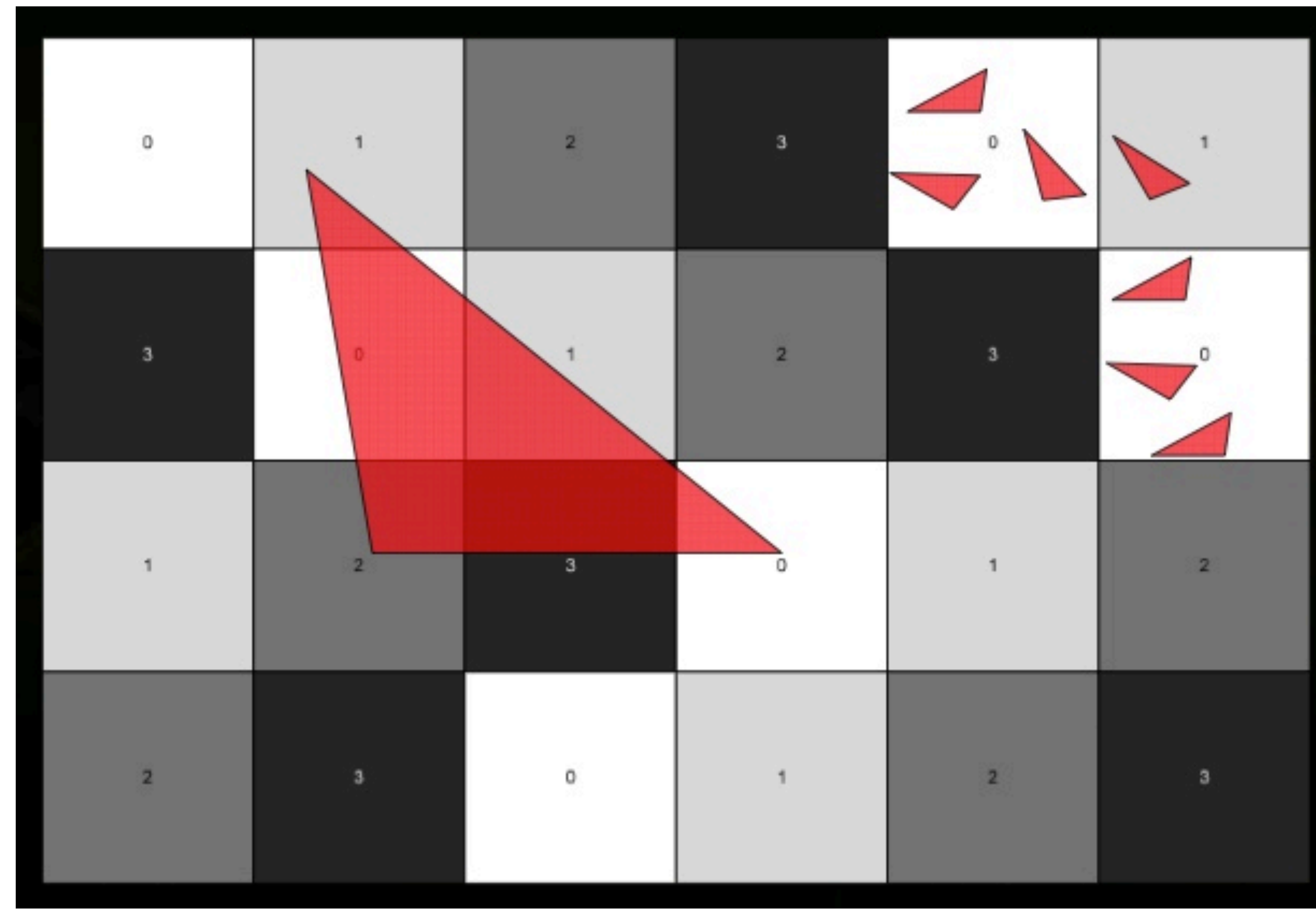
Tiled mapping

Interleaving in NVIDIA Fermi

Fine granularity interleaving



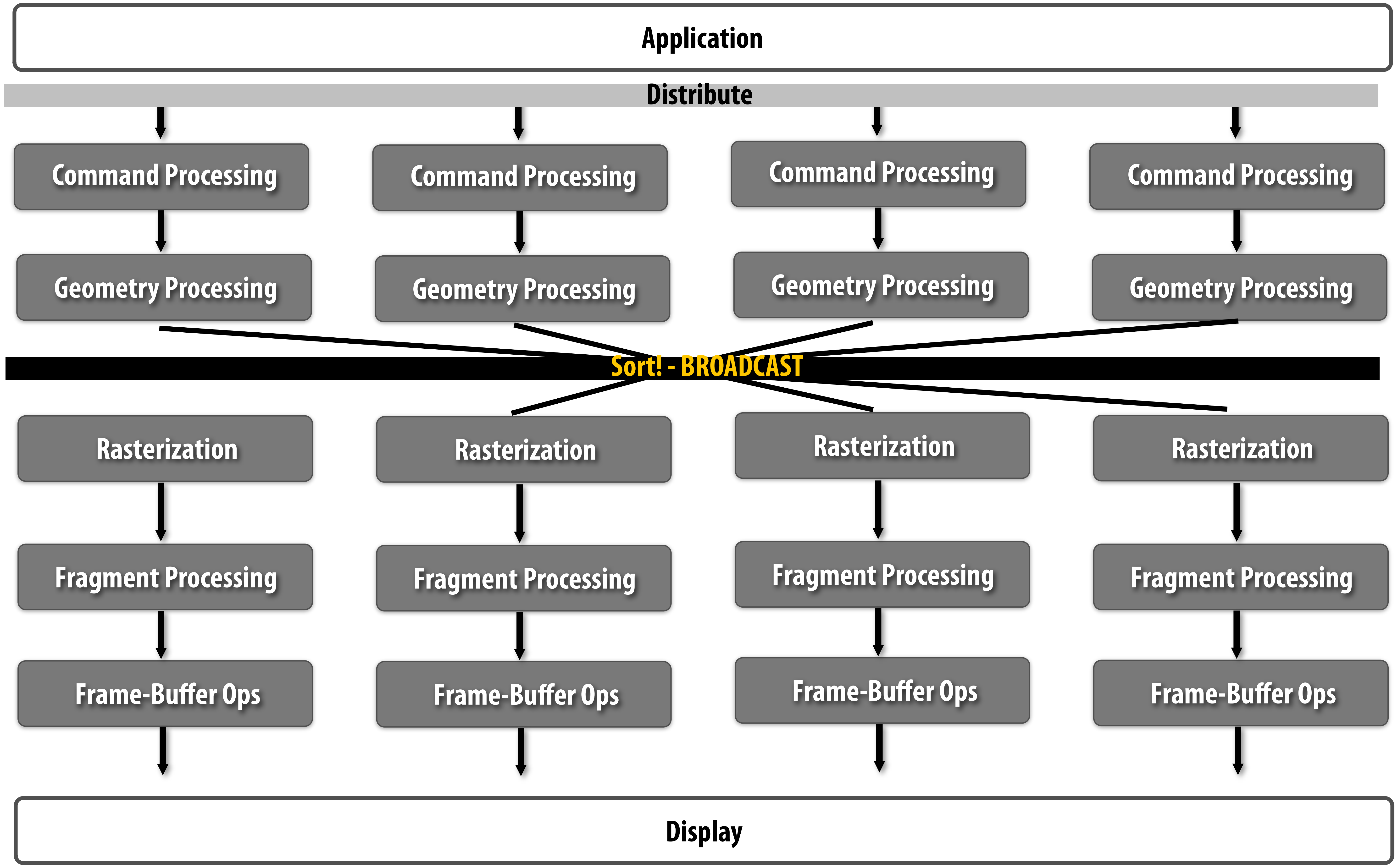
Coarse granularity interleaving



Notice anything interesting about these patterns?

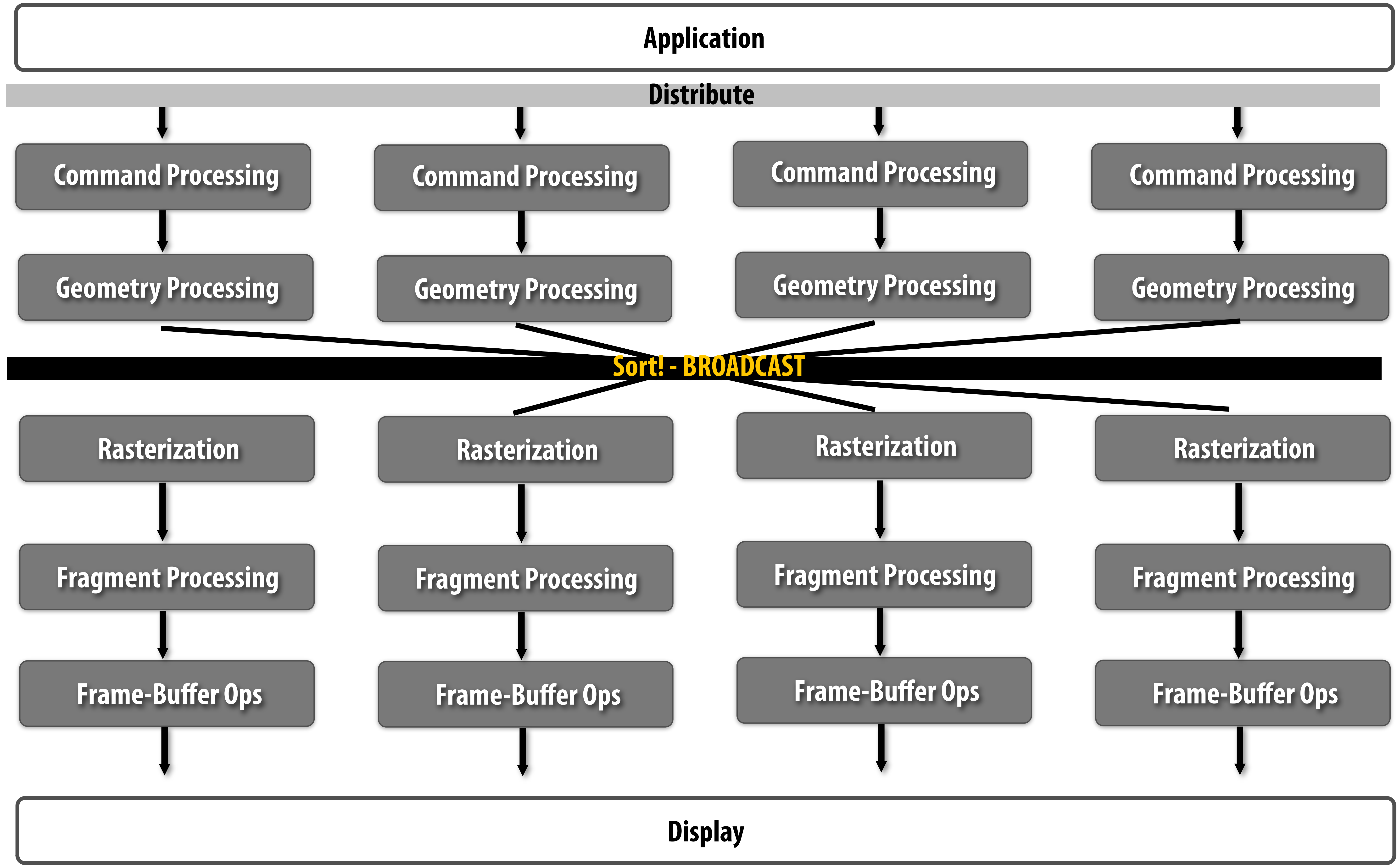
[Image source: NVIDIA]

Sort middle interleaved



- **Good:**
  - Workload balance: both for geometry work AND onto rasterizers (due to interleaving)
  - Computation scaling
  - Easy fine early occlusion cull
  - Does not duplicate geometry processing for each overlapped screen region

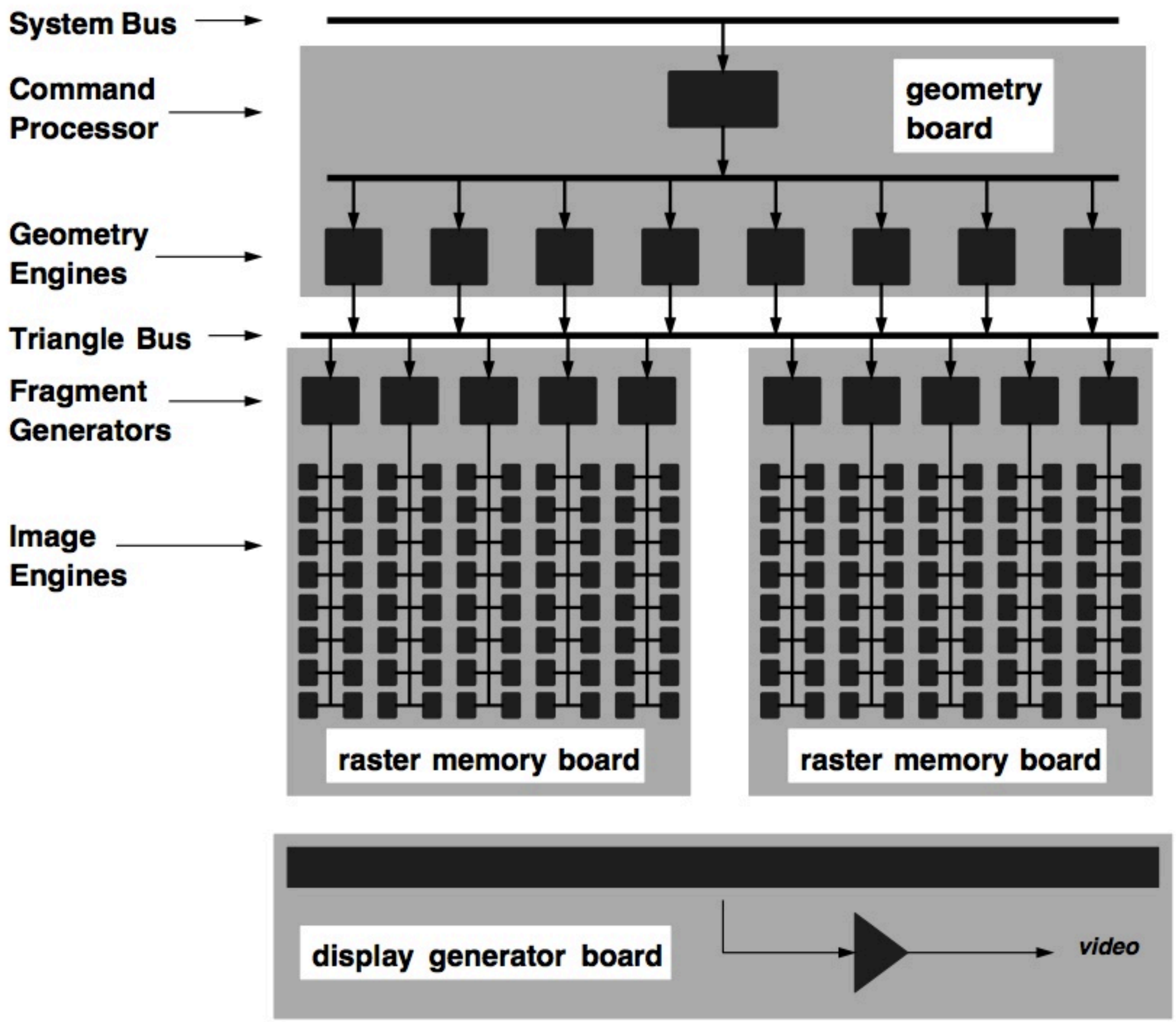
Sort middle interleaved



- **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

SGI RealityEngine [Akeley 93]

Sort-middle interleaved design

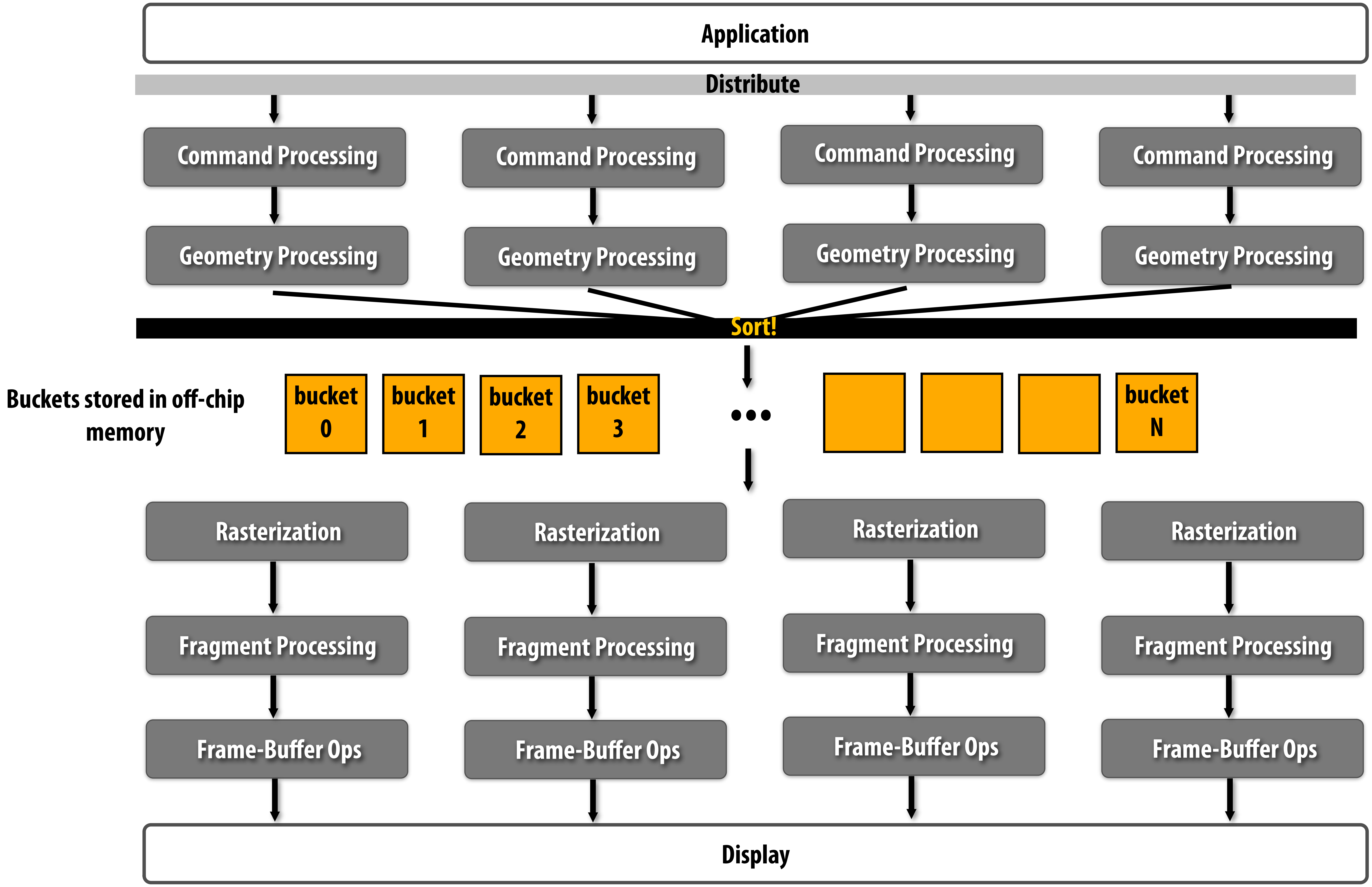


Sort middle tiled

- Sort does not require broadcast
  - Point-to-point communication
  - Better bandwidth scaling
  - Less duplicated triangle setup

- Risks workload imbalance among rasterizers
  - NVIDIA term: "camping" -- when a triangle falls entirely within a tile mapped to one rasterizer, causing imbalance

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)

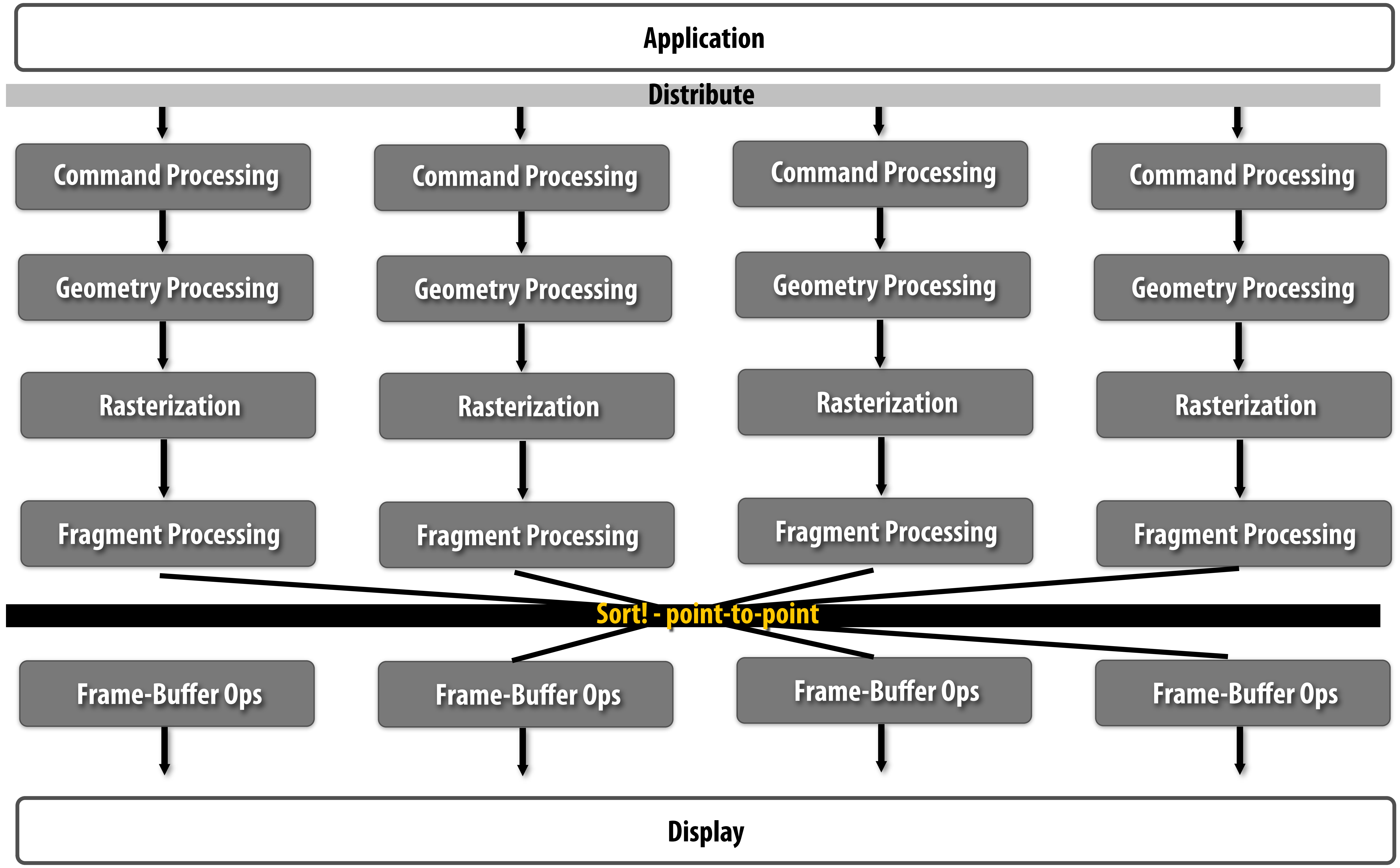
After all geometry complete, rasterizers process buckets (think: work queue of buckets)

Sort middle tiled (chunked)

- Two phase approach:
  - Phase 1: place triangles into buckets
  - Phase 2: rasterize contents of buckets (independently for each bucket)
- Requires off-chip storage of triangle lists for each bucket
- Good:
  - Sort requires point-to-point traffic (assuming each triangle only touches a few buckets)
  - Good load balance (distribute buckets onto rasterizers)
  - Low bandwidth requirements (why?)
- Recent examples:
  - Intel Larrabee
  - NVIDIA CUDA software rasterizer
  - Many mobile GPUs (ARM MALI, Imagination)

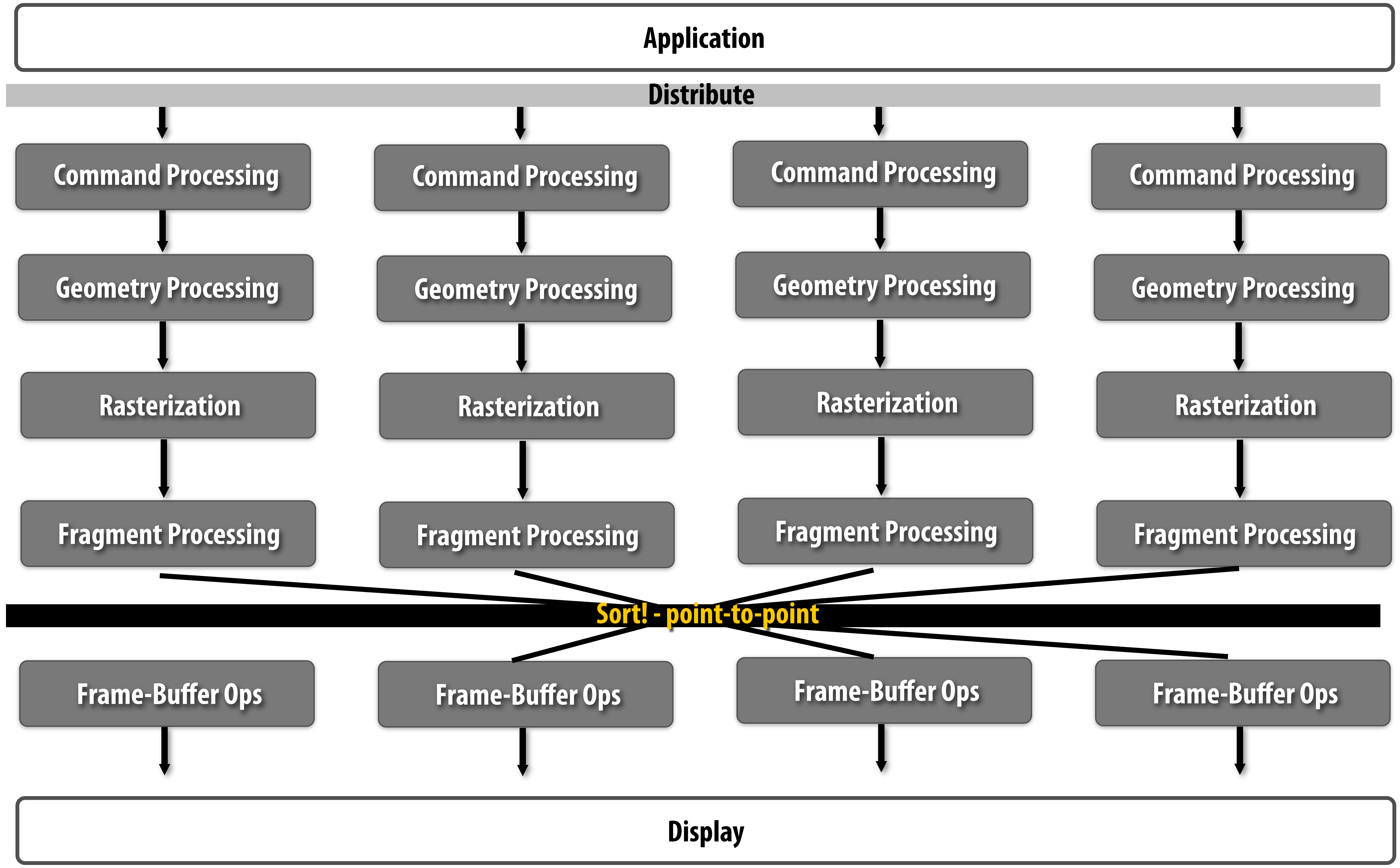
Sort last

Sort last fragment



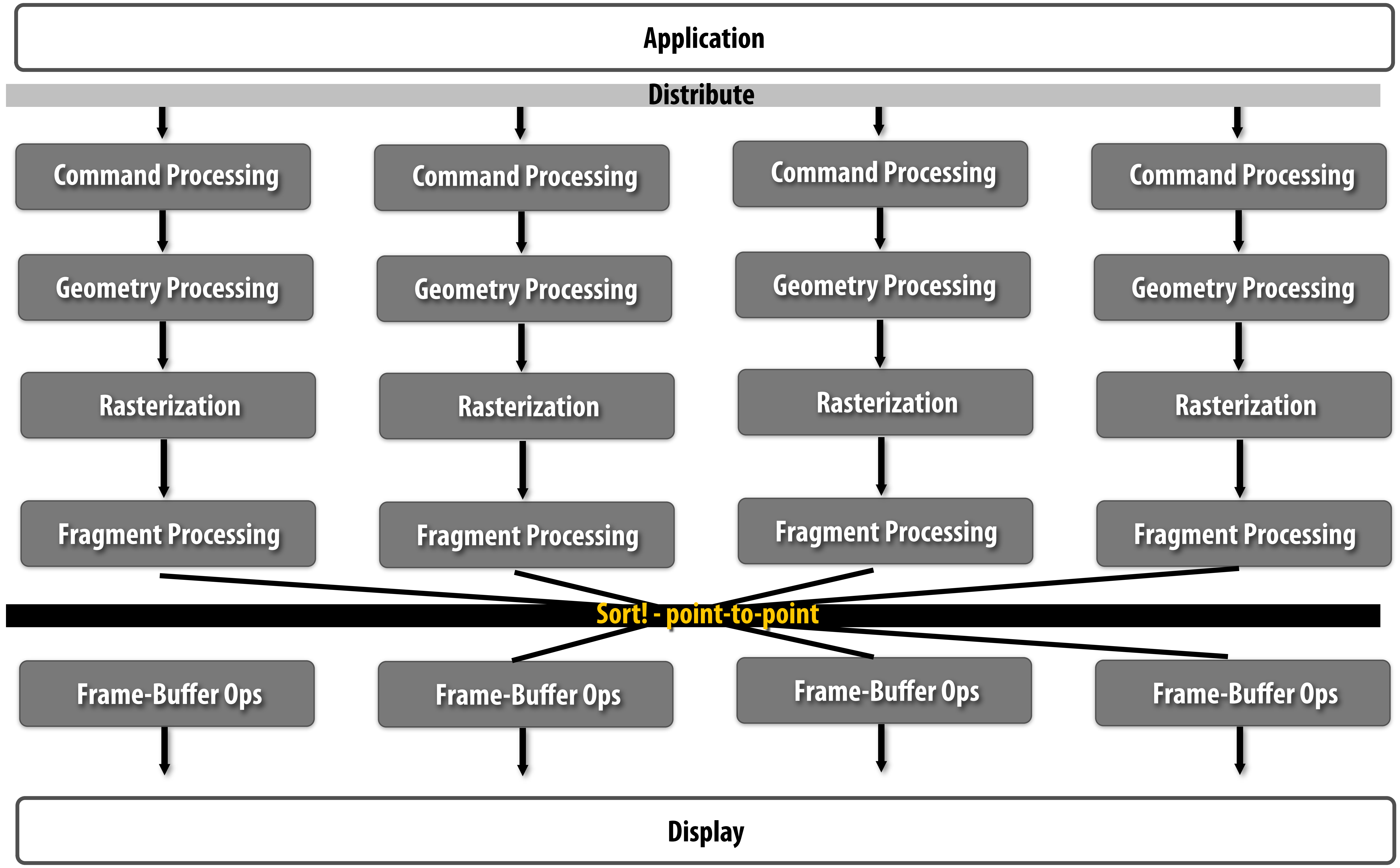
Distribute primitives to top of pipelines (e.g., round robin)
Sort after fragment processing based on (x,y) position of fragment

Sort last fragment



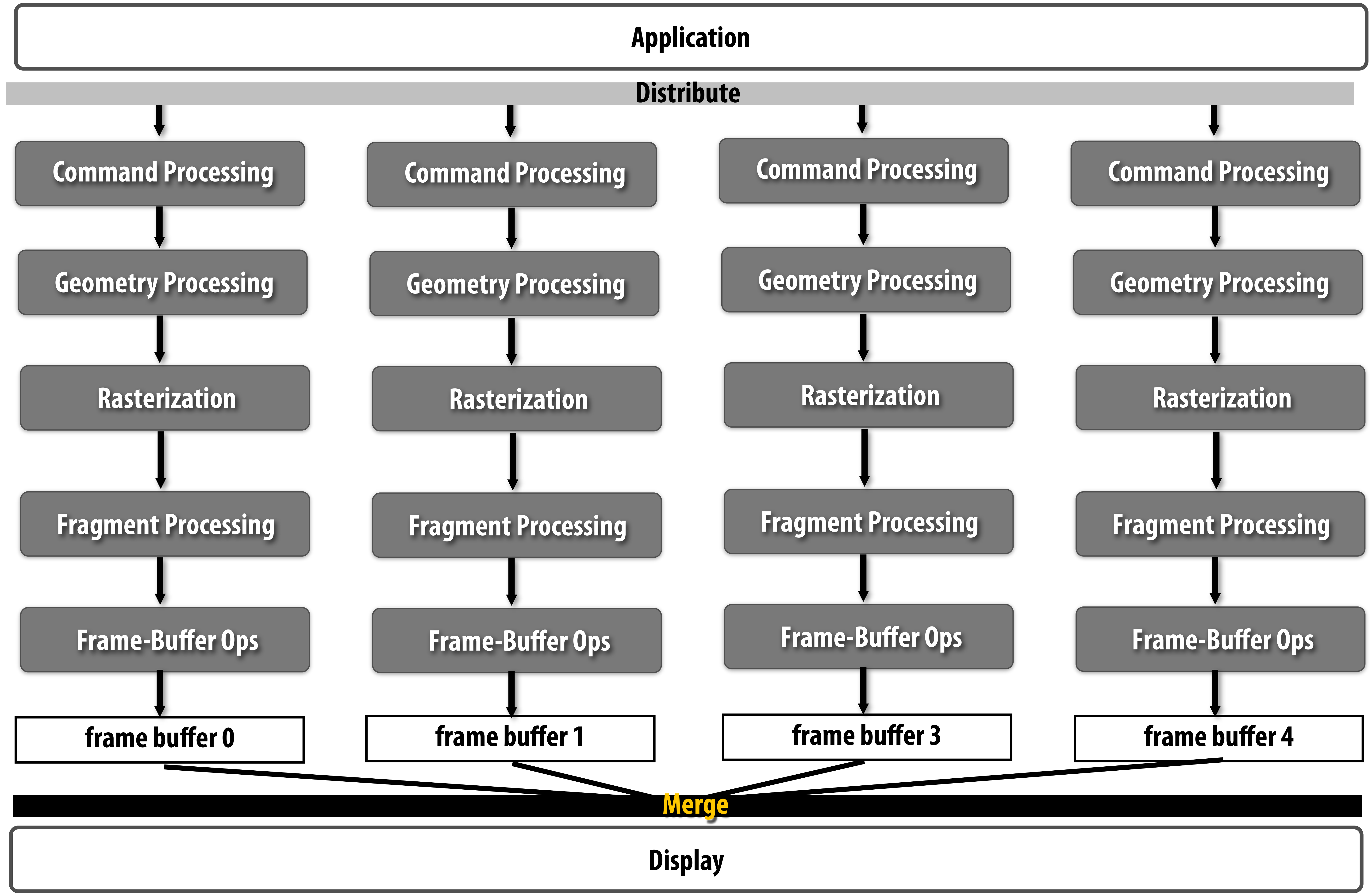
- **Good:**
  - No redundant work (geometry processing or in rasterizers)
  - Point-to-point communication during sort
  - Interleaved pixel mapping results in good workload balance for frame-buffer ops

Sort last fragment



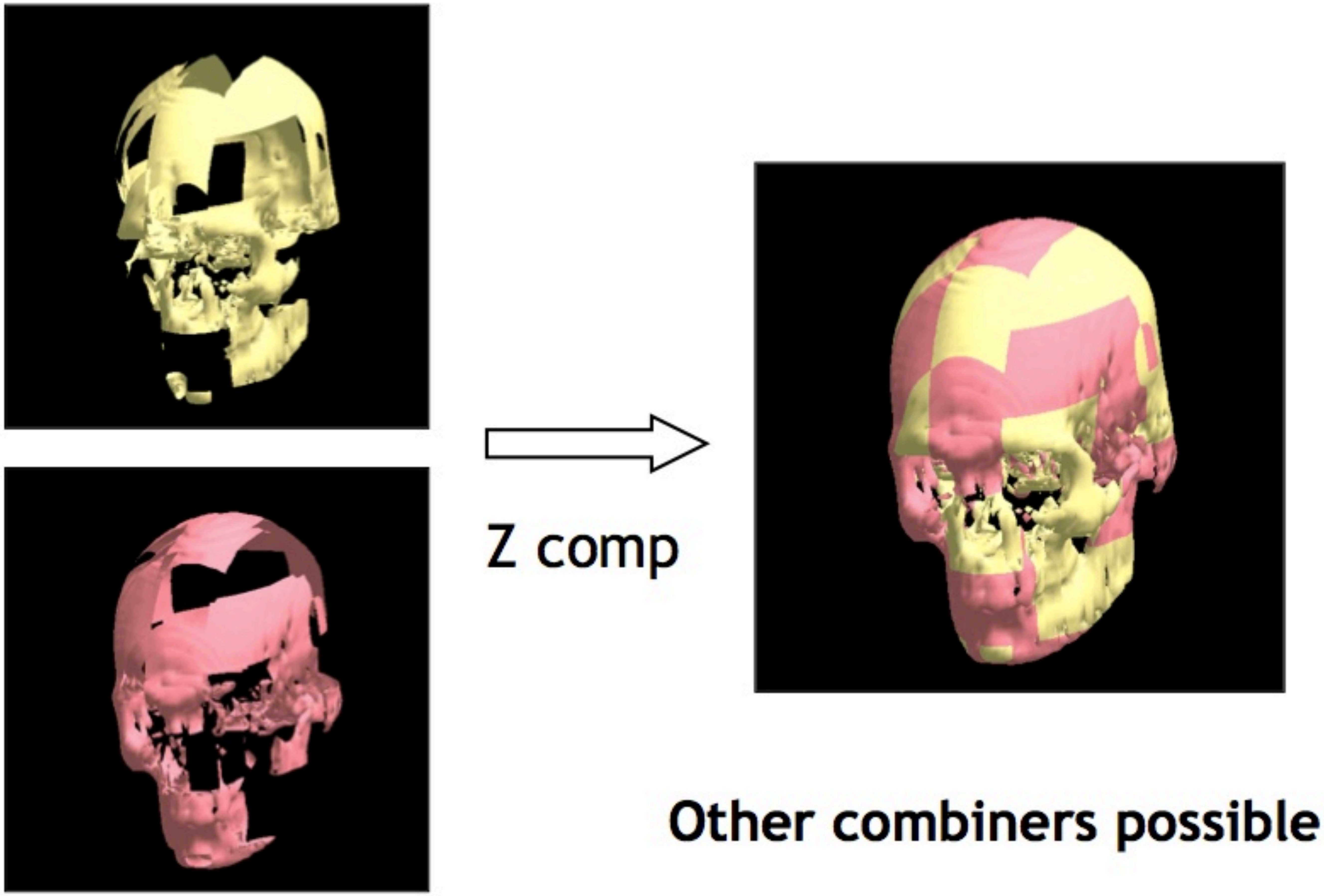
- **Bad:**
  - Workload imbalance due to primitives of varying size
  - Bandwidth scaling: many more fragments than triangles
  - Hard to implement early occlusion cull (more bandwidth challenges)

Sort last image composition



Each pipeline renders some part of the frame (color buffer + depth buffer)
Combine the color buffers, according to depth into the final image

Sort last image composition



Sort last image composition

- Cannot maintain sequential semantics

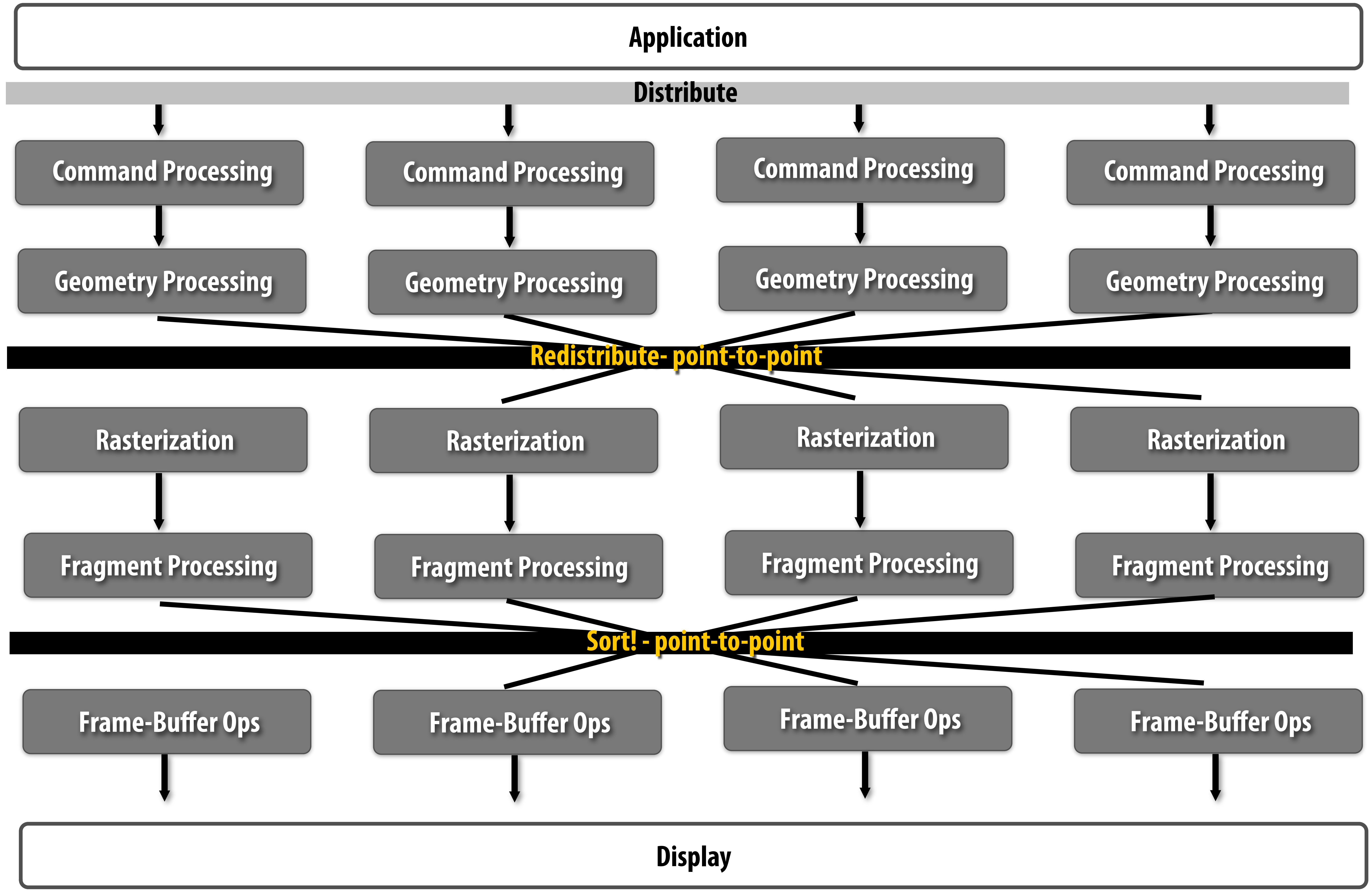
- Simple: N separate rendering pipelines
  - Can use off-the-shelf GPUs to build a massive rendering system
  - Coarse-grained communication

- Similar load imbalance problems as sort-last fragment

- Bandwidth requirements compared to sort-last fragment depend on scene depth complexity

Sort everywhere

Pomegranate [Eldridge 00]



Distribute primitives to top of pipelines

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

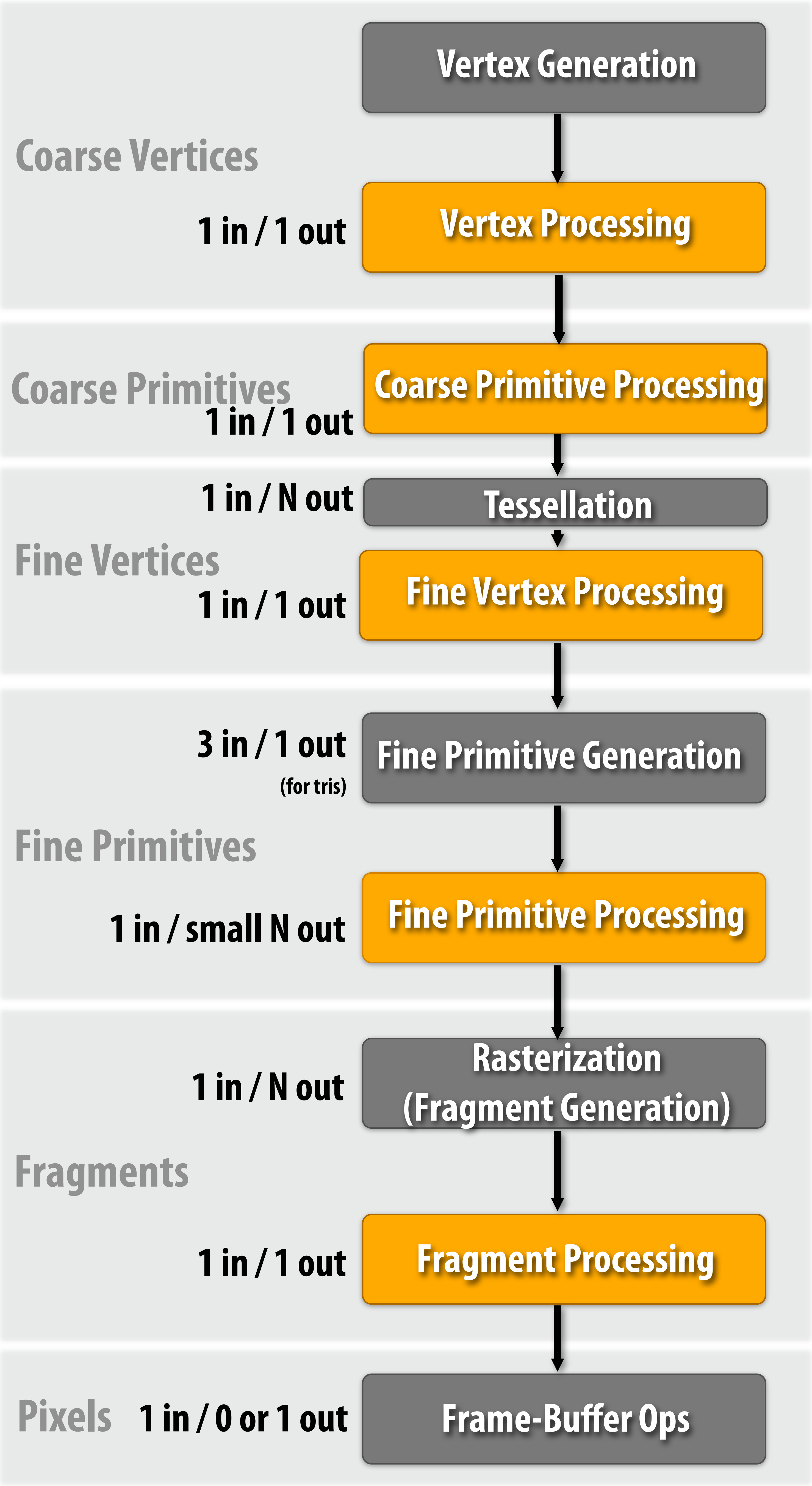
Sort after fragment processing based on (x,y) position of fragment

Recall: modern OpenGL 4/Direct3D 11 pipeline

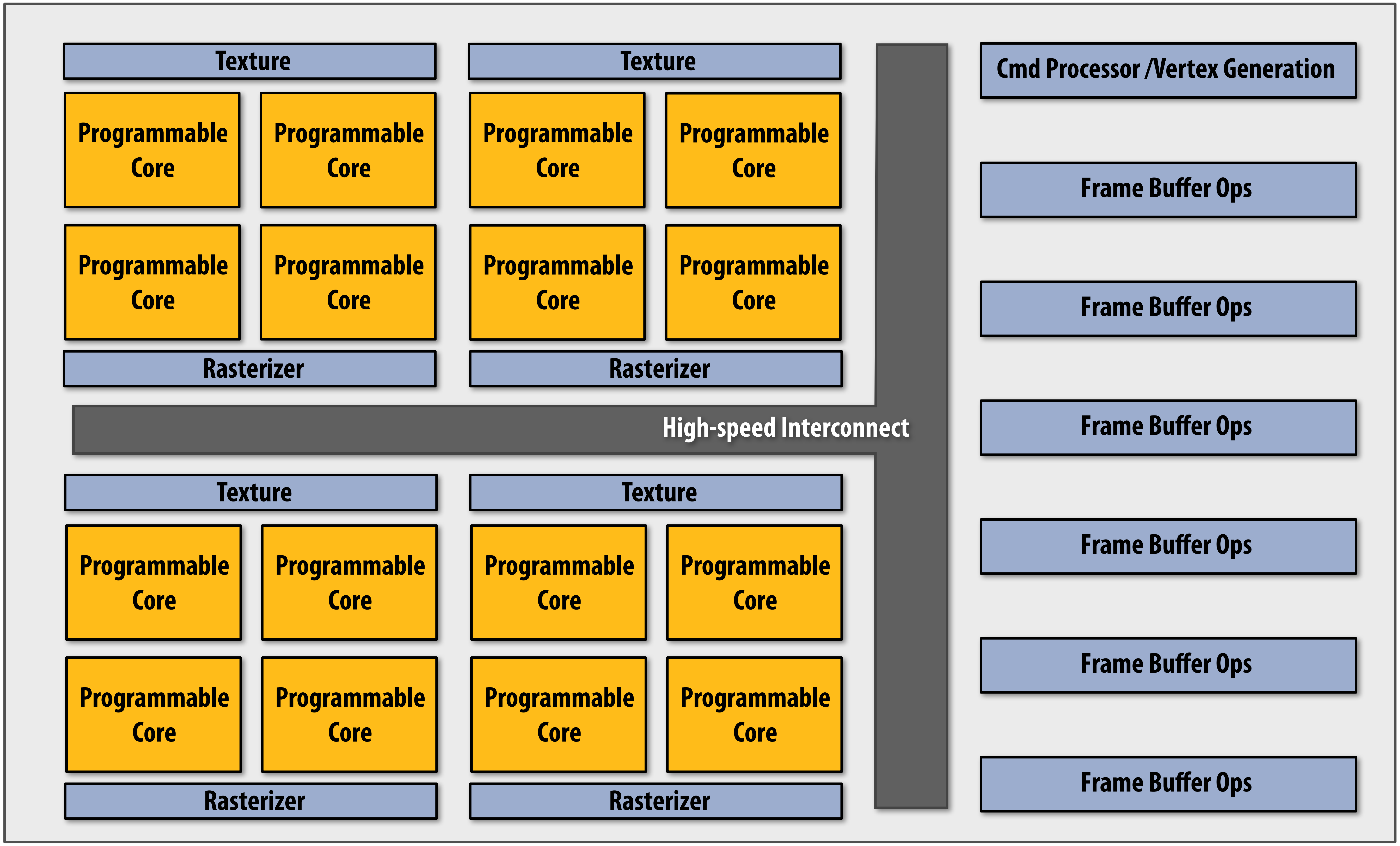
Five programmable stages

Including tessellation

Programmable stages feature data-dependent control flow in shaders (unpredictable per vertex/per fragment run-time)



Modern GPUs



Hardware is a heterogeneous collection of resources

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

Must keep programmable cores busy: sort everywhere

Readings

- Eldridge et al. Pomegranate: A Fully Scalable Graphics Architecture. SIGGRAPH 2000

- Molnar et al. A Sorting Classification of Parallel Rendering. IEEE Graphics and Applications 1994