Optimizing Roblox: Vulkan Best Practices for Mobile Developers

Jose Emilio Munoz-Lopez (Senior Software Engineer, Arm)
Arseny Kapoulkine (Technical Fellow, Roblox)

Agenda

- Introduction
- Vulkan GPU best practices
  - Load/Store operations
  - Vulkan subpasses
  - Pipeline barriers
  - MSAA
- Roblox CPU optimizations
  - Command buffer management
  - Render passes
  - Pipeline state
  - Descriptor management
- Further reading

Arm: The Mobile Game Industry's Architecture of Choice

23+ bn

Arm-based Cortex CPUs shipped in 2019*

70%

of the world's population uses Arm technology

Over 1bn

Arm-based Mali GPUs shipped in 2019

1000+

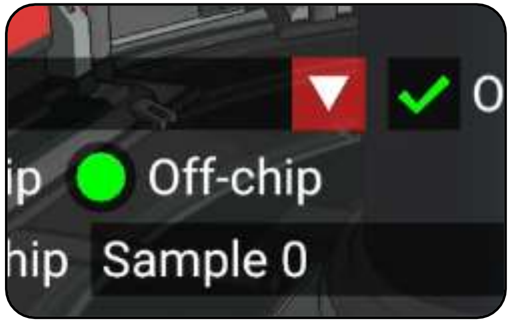
Ecosystem Partners

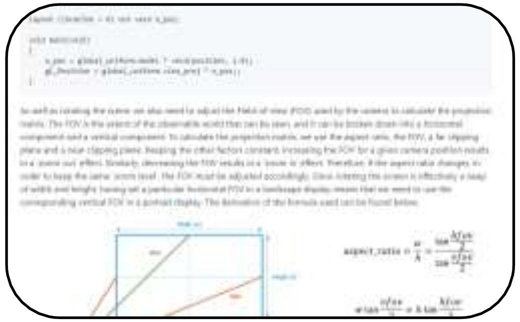
---

*As reported to Arm in 2019 calendar by partners

Vulkan Samples

Vulkan samples





Run samples

Analyze counters

Read tutorials

Experiment on a mobile-optimized, multi-platform framework

https://github.com/KhronosGroup/Vulkan-Samples

Pipeline Barriers

GPU: Mali-G76

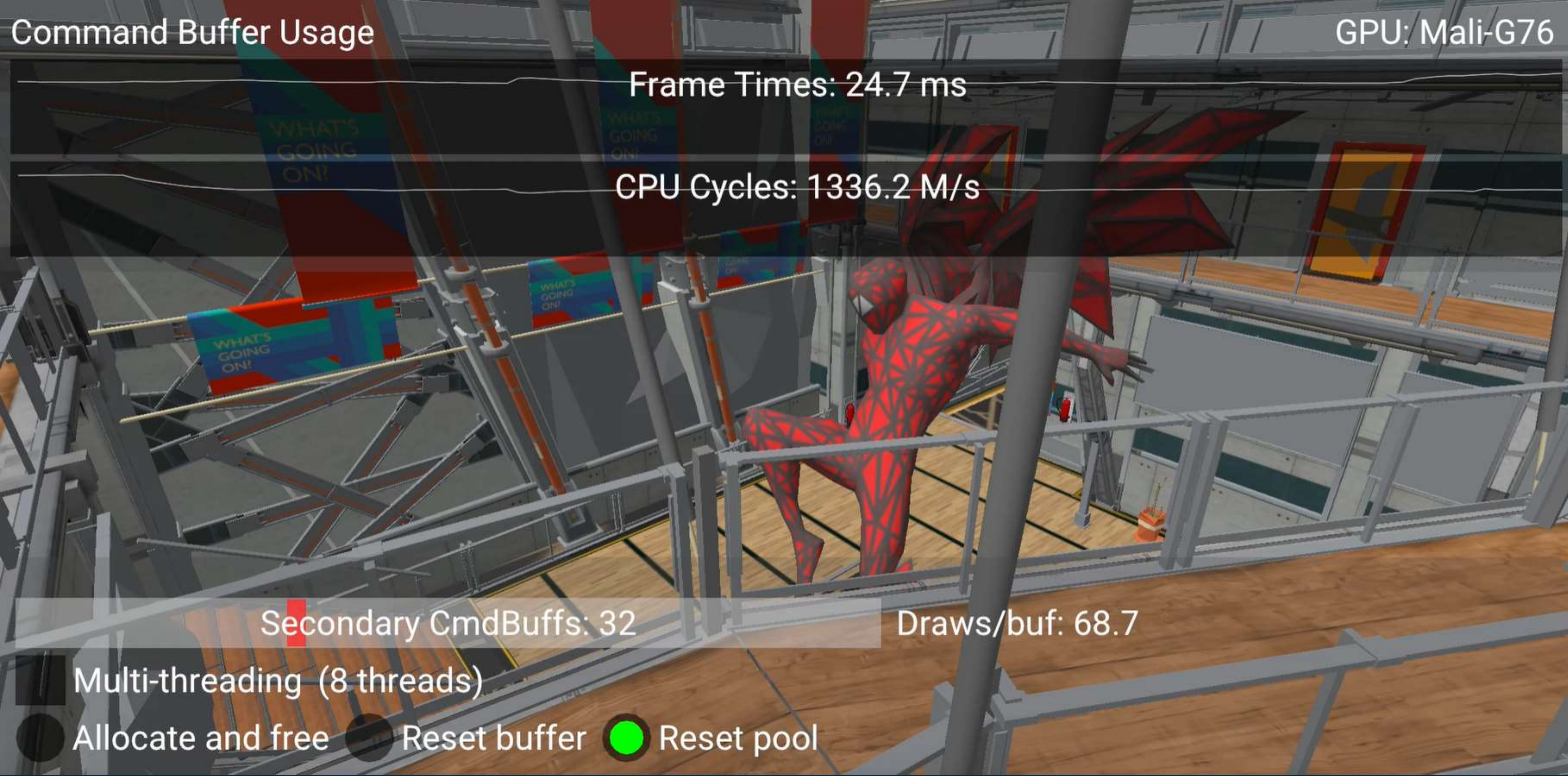
Frame Times: 16.7 ms

Vertex Compute Cycles: 98.0 M/s

Fragment Cycles: 667.3 M/s

Pipeline barrier stages:

○ Bottom to top ○ Frag to vert ● Frag to frag



Command Buffer Usage

GPU: Mali-G76

Frame Times: 24.7 ms

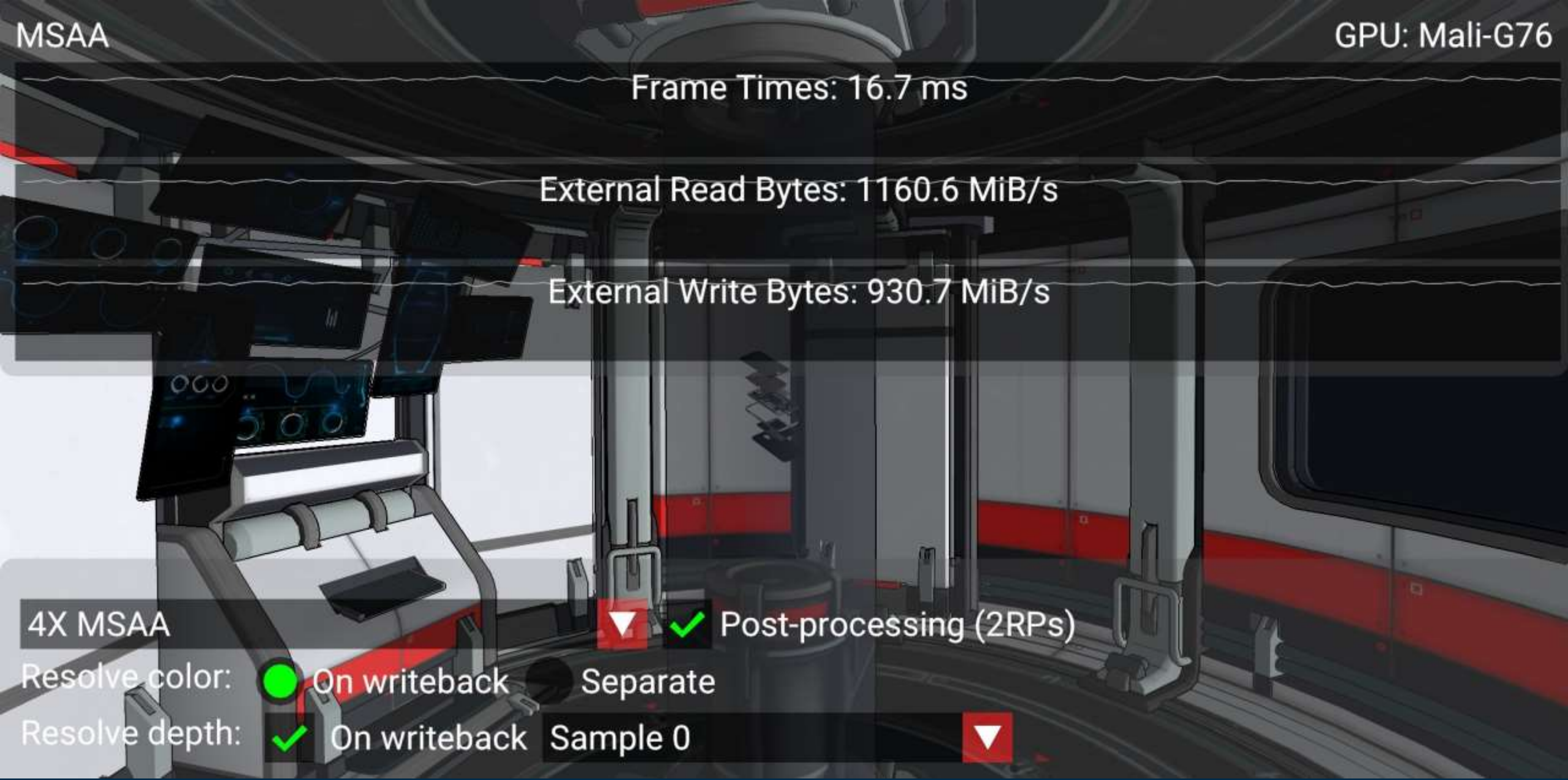
CPU Cycles: 1336.2 M/s

Secondary CmdBuffs: 32

Draws/buf: 68.7

☐ Multi-threading (8 threads)

○ Allocate and free ○ Reset buffer ● Reset pool



MSAA

GPU: Mali-G76

Frame Times: 16.7 ms

External Read Bytes: 1160.6 MiB/s

External Write Bytes: 930.7 MiB/s

4X MSAA ▼ ☑ Post-processing (2RPs)

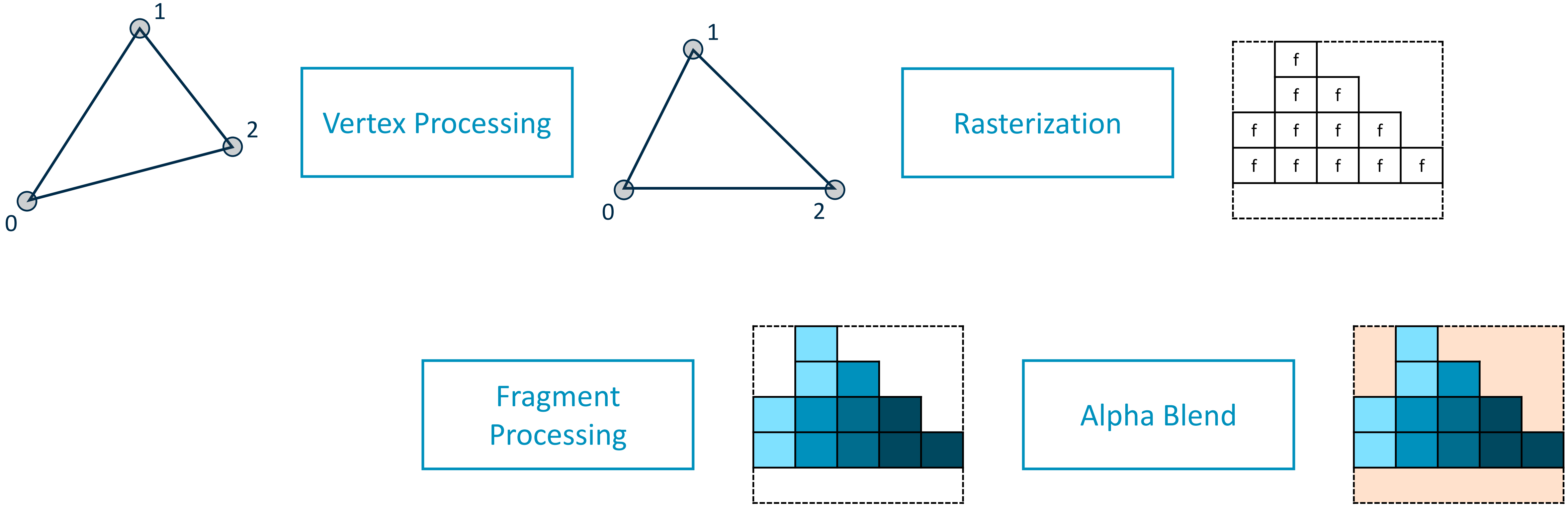
Resolve color: ● On writeback ○ Separate

Resolve depth: ☑ On writeback Sample 0 ▼

Framework

- Platform independent (Android, Linux, Mac and Windows)
- Maintains a close relationship with Vulkan objects
- Runtime GLSL shader variant generation + shader reflection (Khronos' SPIRV-Cross)
  - Automate creation of Vulkan objects:
    - VkRenderPass
    - VkFramebuffer
    - VkPipelineLayout
    - VkDescriptorSetLayout
- Load 3D models (glTF 2.0)
  - Internal scene graph

The graphics pipeline



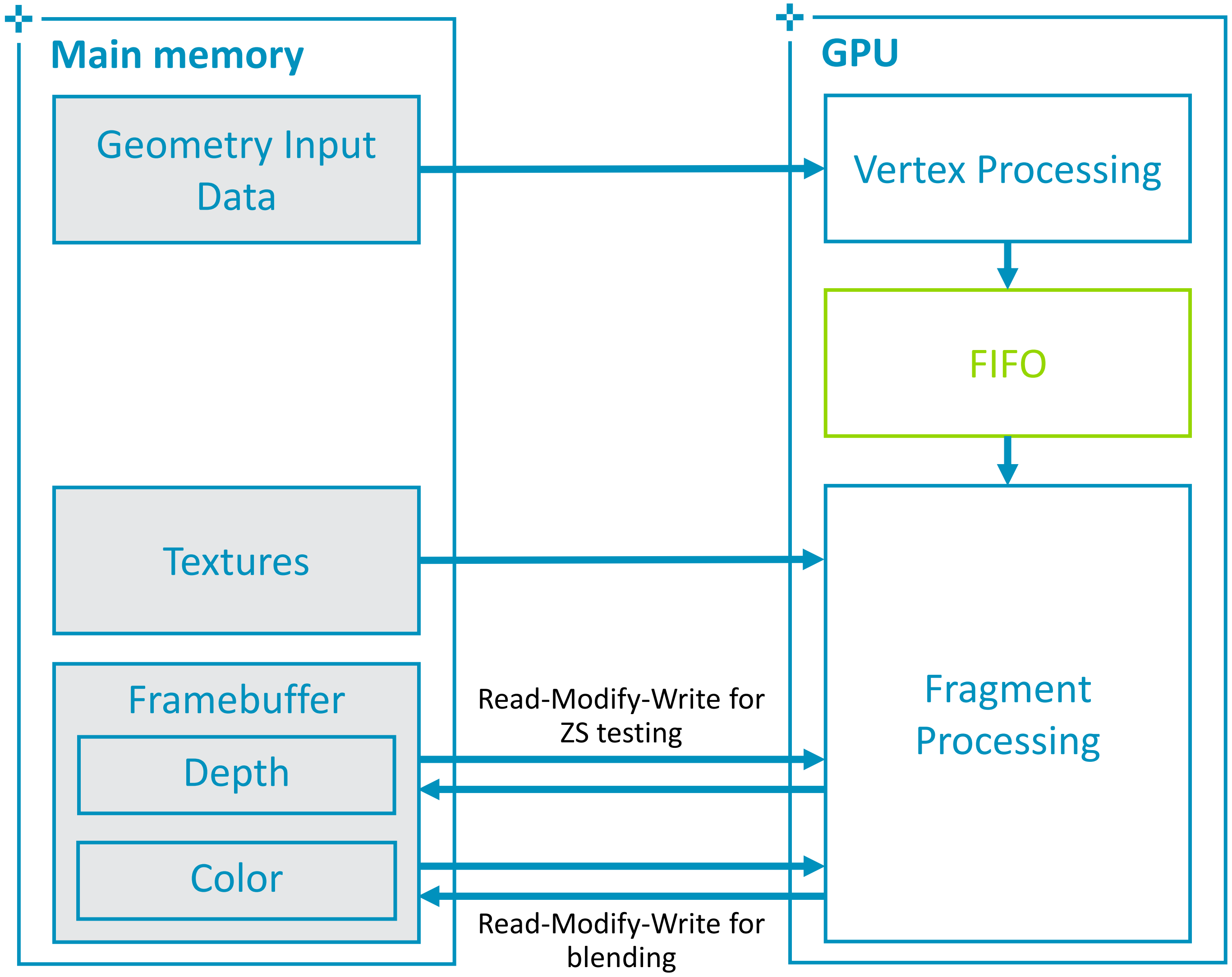
Vertex Processing

Rasterization

Fragment Processing

Alpha Blend

Immediate mode GPUs



Main memory

Geometry Input Data

Textures

Framebuffer

Depth

Color

GPU

Vertex Processing

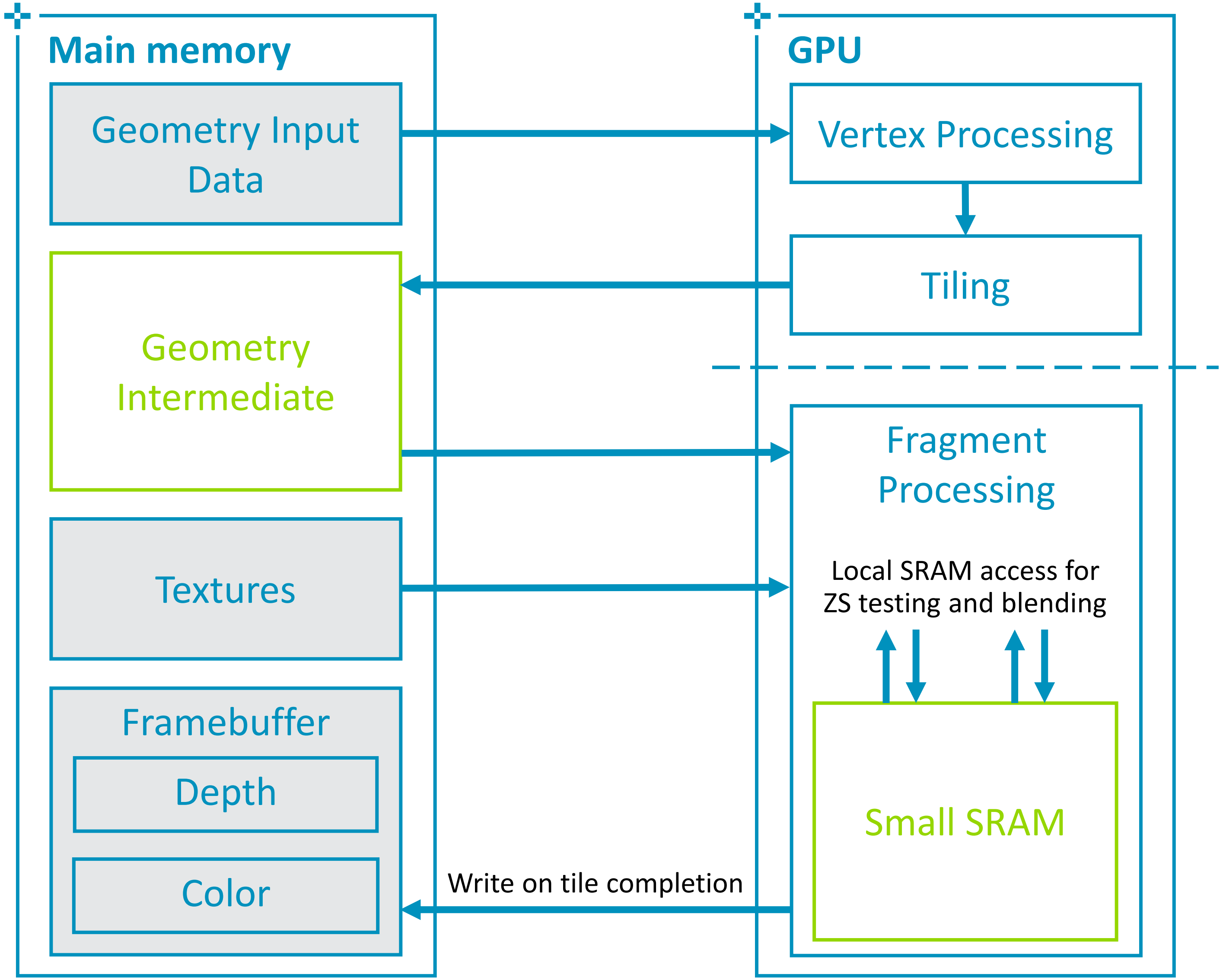
FIFO

Fragment Processing

Read-Modify-Write for ZS testing

Read-Modify-Write for blending

Tiled GPUs



Main memory

Geometry Input Data

Geometry Intermediate

Textures

Framebuffer

Depth

Color

GPU

Vertex Processing

Tiling

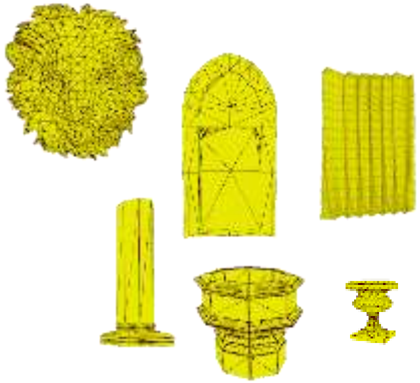
Fragment Processing

Local SRAM access for ZS testing and blending

Small SRAM

Write on tile completion

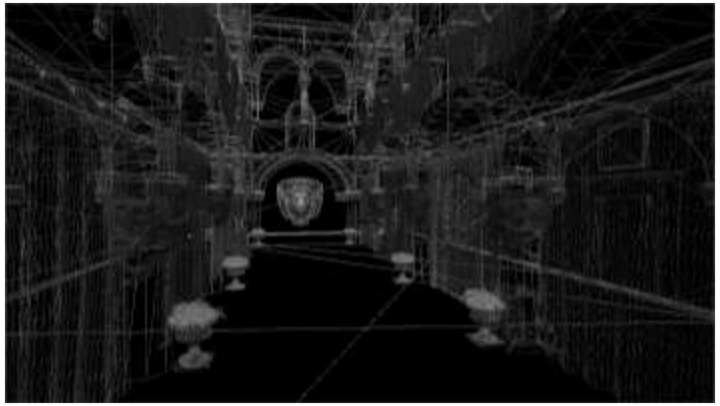
Renderpasses and subpasses



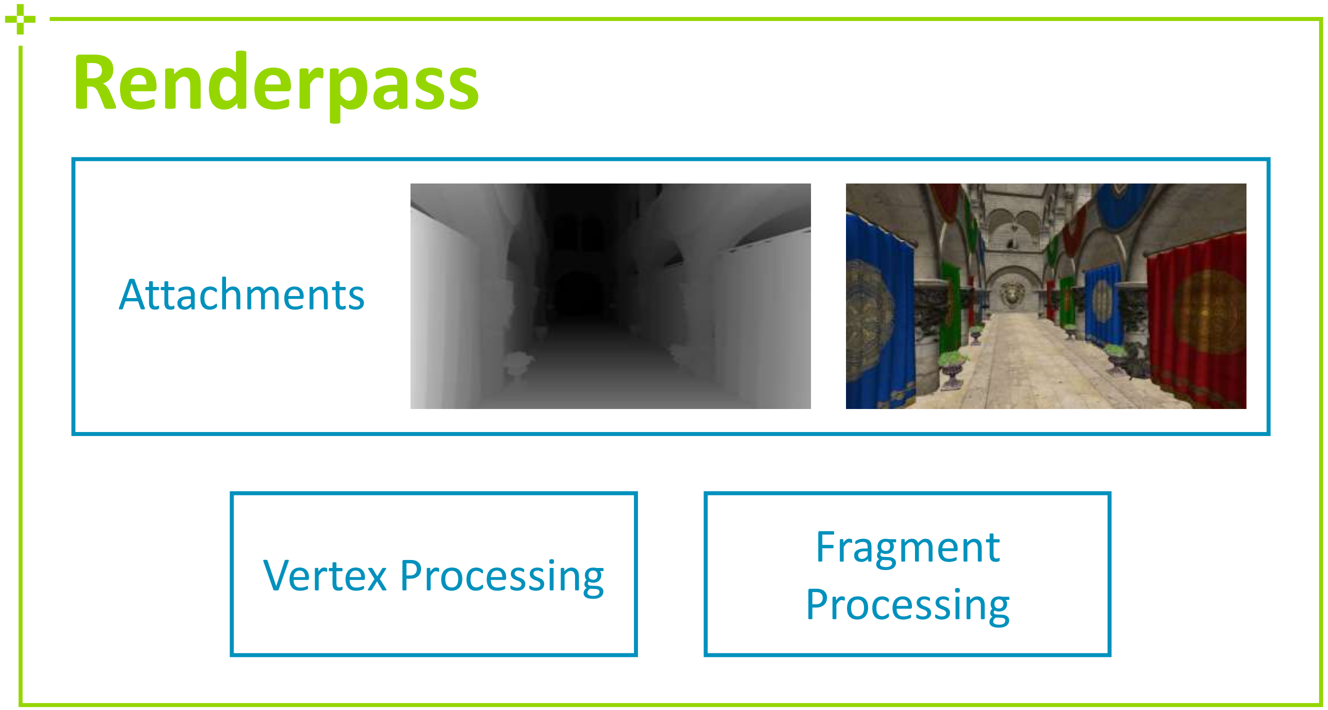
Vertex Processing

Rasterization

Fragment Processing



Renderpasses and subpasses



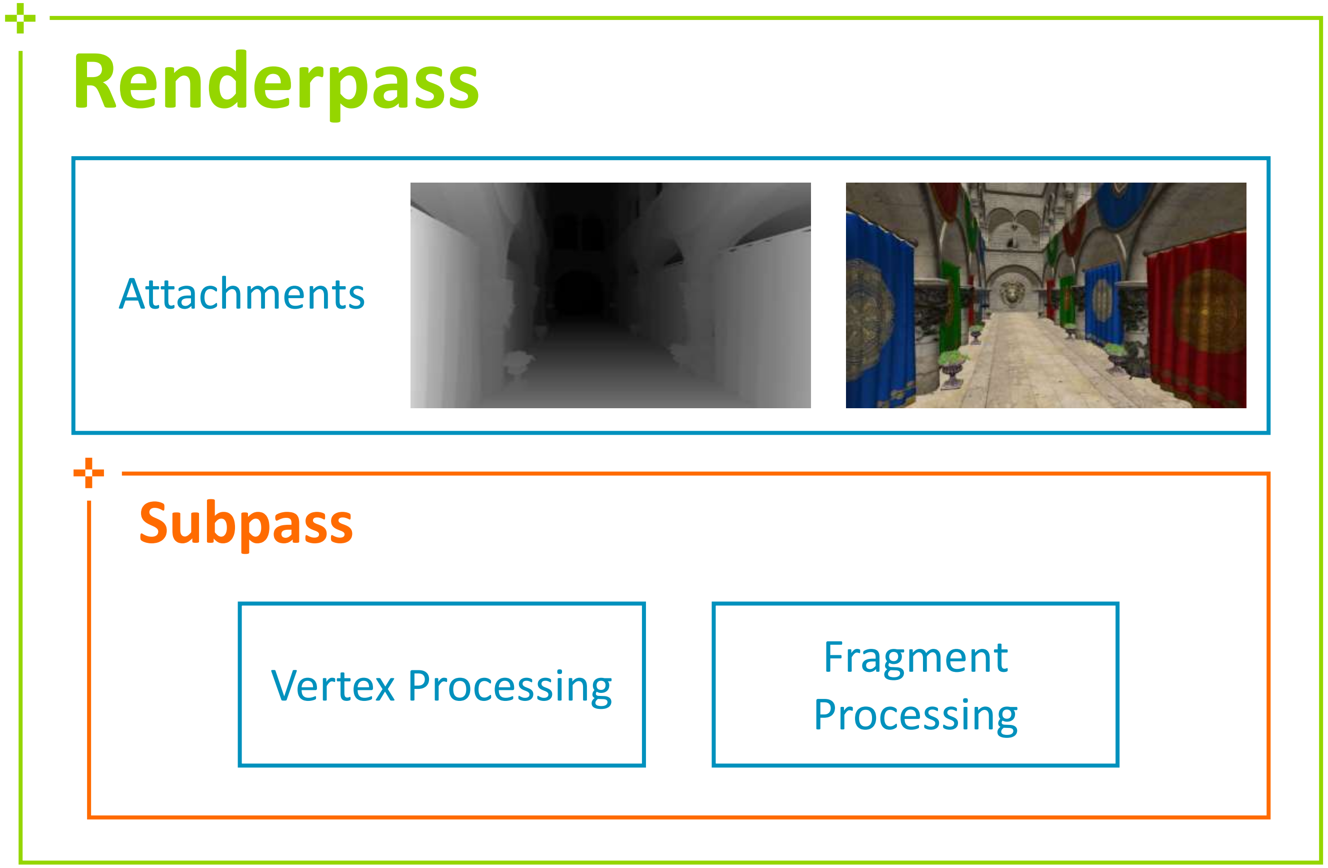
Renderpass

Attachments

Vertex Processing

Fragment Processing

Renderpasses and subpasses



Renderpass

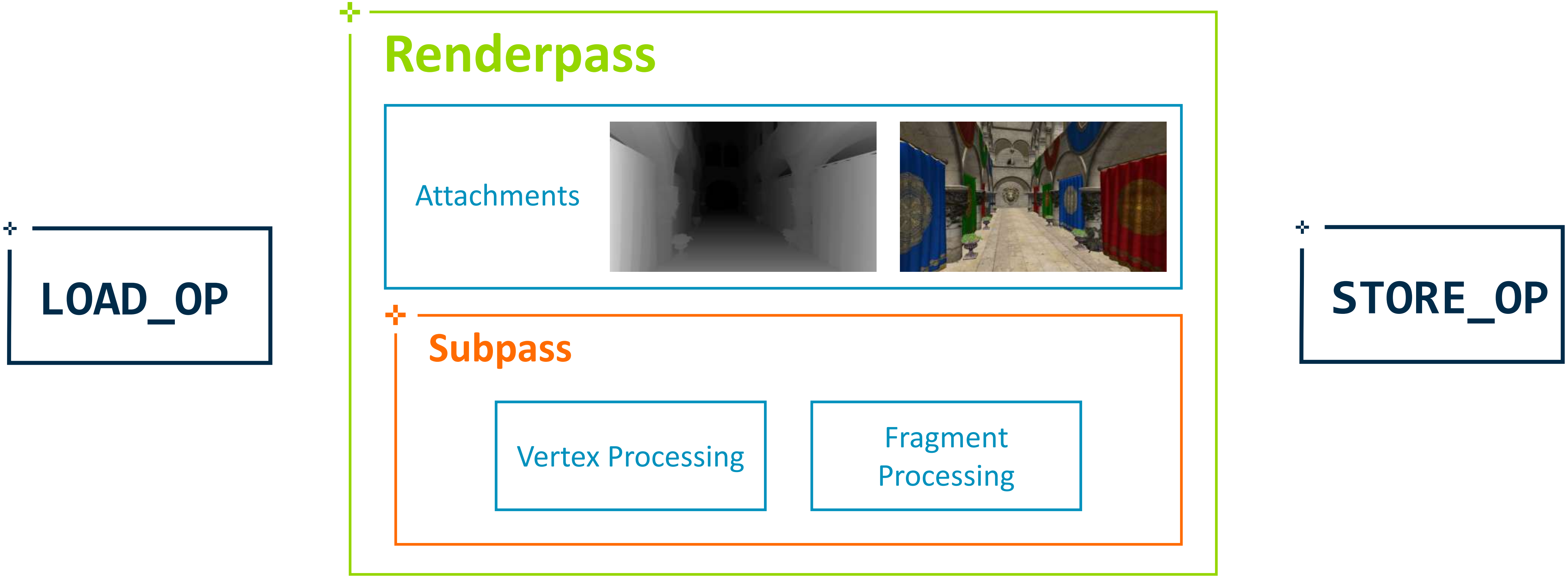
Attachments

Subpass

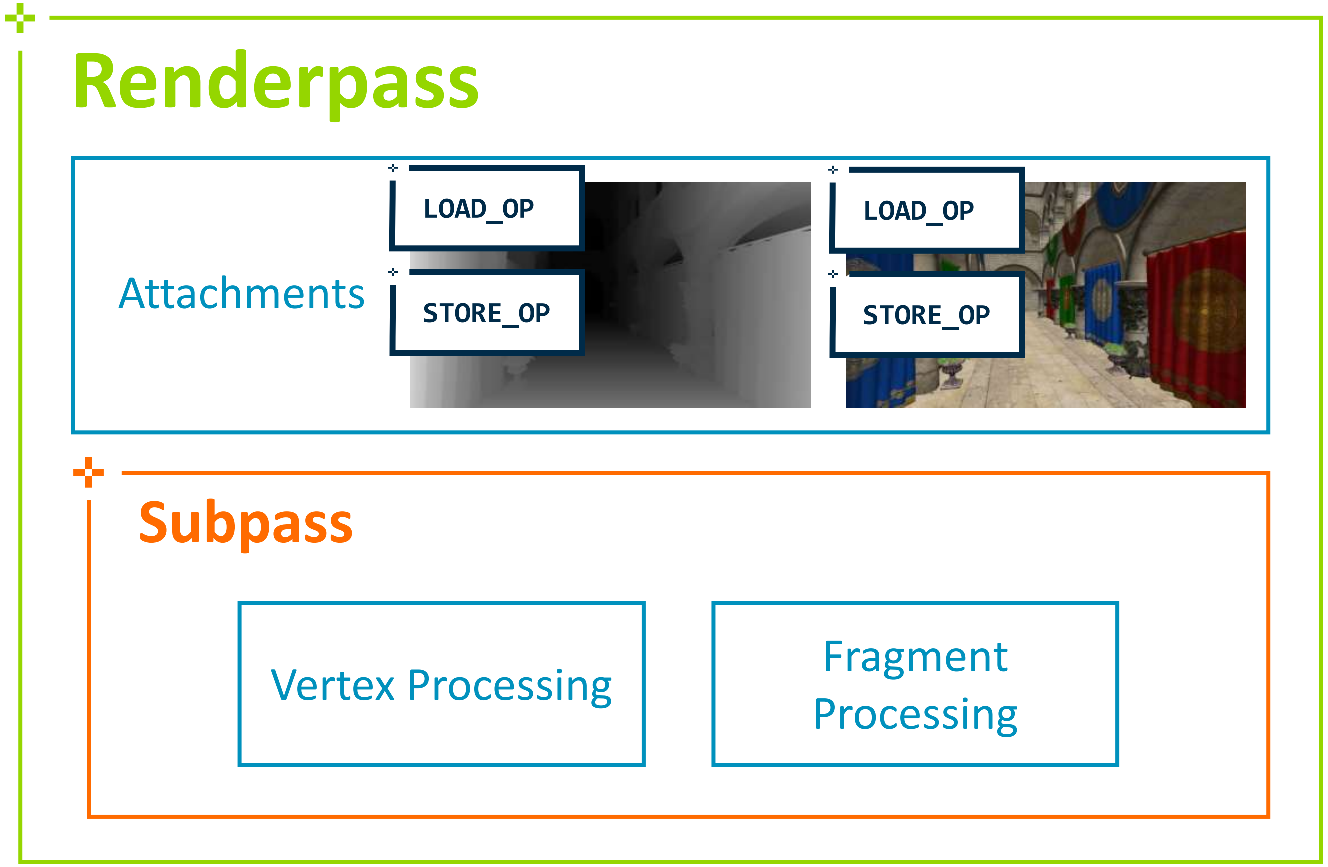
Vertex Processing

Fragment Processing

Renderpasses and subpasses



Renderpasses and subpasses



Renderpass

Attachments

`LOAD_OP`

`STORE_OP`

`LOAD_OP`

`STORE_OP`

Subpass

Vertex Processing

Fragment Processing

Load/Store operations

Load operations

- `loadOp` operations define how to initialize memory at the start of a render pass

LOAD_OP_LOAD

LOAD_OP_CLEAR

LOAD_OP_DONT_CARE

- Clear or invalidate each attachment at the start of a render pass using `LOAD_OP_CLEAR` or `LOAD_OP_DONT_CARE` on mobile
- Do not clear an attachment inside a render pass using `vkCmdClearAttachments()`

```
VkAttachmentDescription color_attachment = {};
color_attachment.format  = VK_FORMAT_B8G8R8A8_SRGB;
color_attachment.samples = VK_SAMPLE_COUNT_1_BIT;

color_attachment.loadOp  = VK_ATTACHMENT_LOAD_OP_CLEAR;
```

Store operations

- `storeOp` operations define what is written back to main memory at the end of a pass

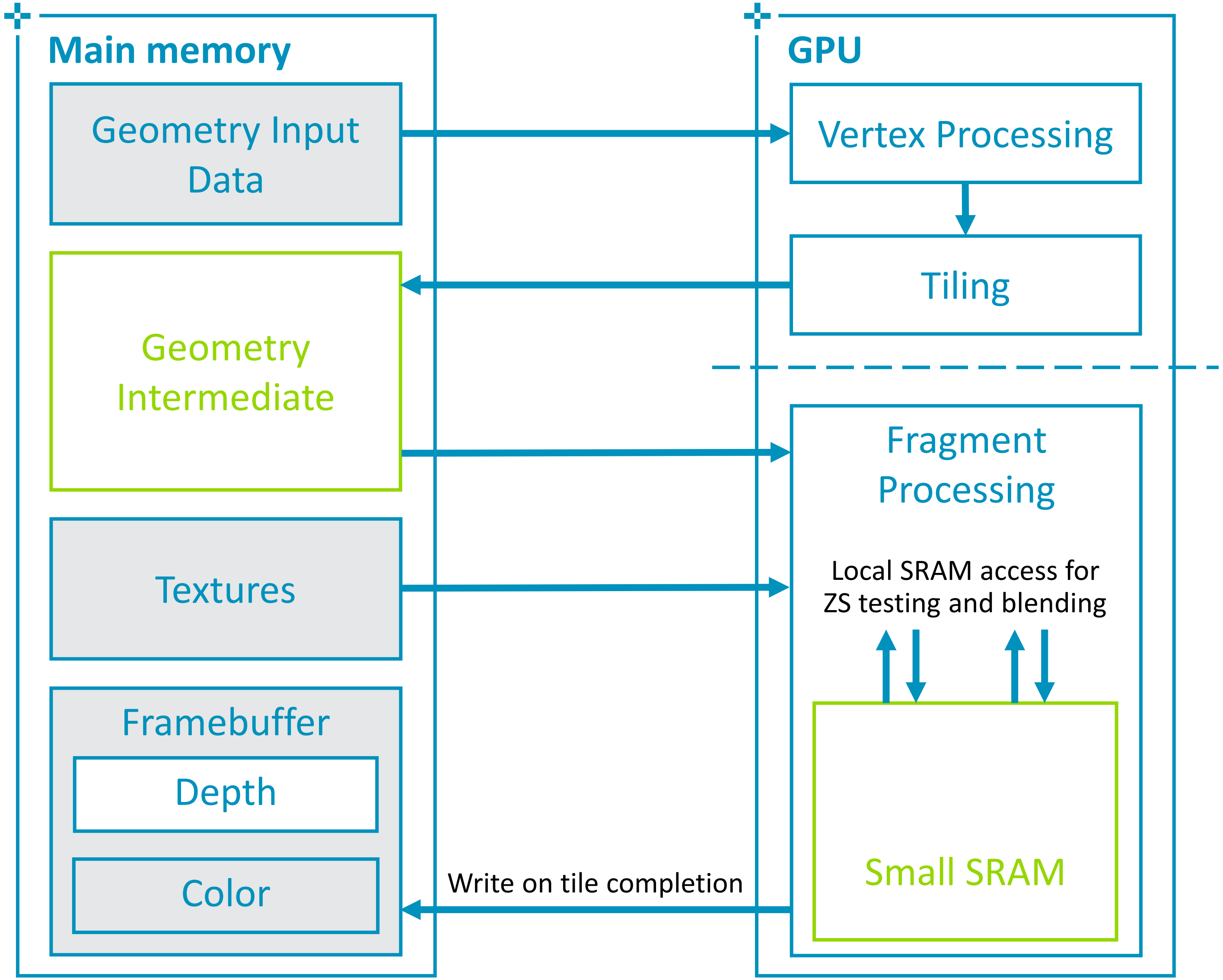
STORE_OP
_STORE

STORE_OP_
DONT_CARE

- If they are not going to be used further, ensure that the contents are invalidated at the end of the render pass using `STORE_OP_DONT_CARE` on mobile

```
VkAttachmentDescription depth_attachment = {};
depth_attachment.format  = VK_FORMAT_D32_SFLOAT;
depth_attachment.samples = VK_SAMPLE_COUNT_1_BIT;

depth_attachment.loadOp  = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
depth_attachment.storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
```

Tiled GPUs



Main memory

Geometry Input Data

Geometry Intermediate

Textures

Framebuffer

Depth

Color

Write on tile completion

GPU

Vertex Processing

Tiling

Fragment Processing

Local SRAM access for ZS testing and blending

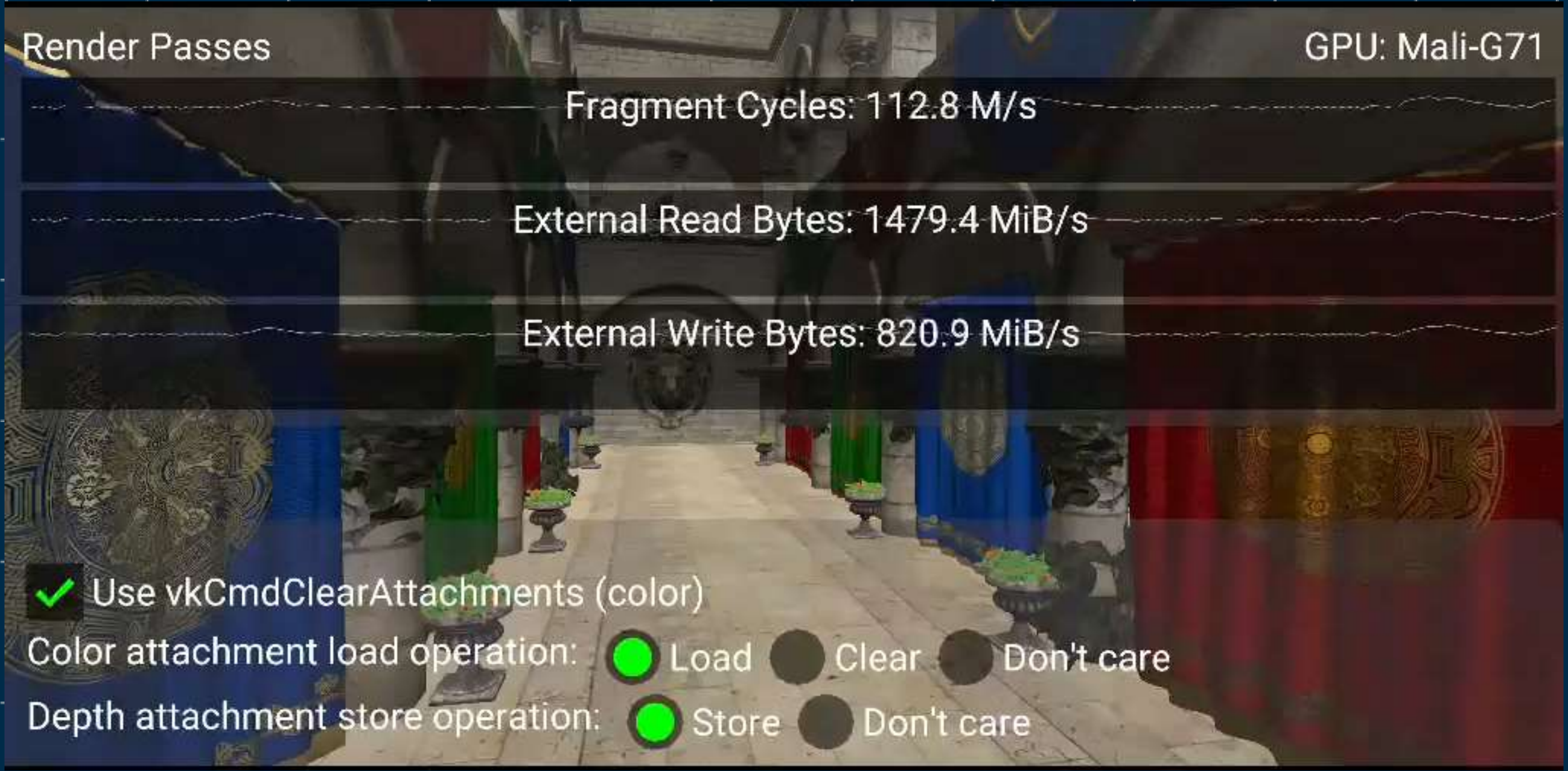
Small SRAM

Transient attachments

- Image usage flags: `TRANSIENT_ATTACHMENT`
- This tells the GPU that it can be used as a transient attachment
- It will only live for the duration of a single render-pass
- Additionally, it can be backed by `LAZILY_ALLOCATED` memory
- This way a tile-based renderer may avoid allocating external memory for it

```
VkImageCreateInfo image_info{VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO};
image_info.flags       = flags;
image_info.imageType   = type;
image_info.format      = format;
image_info.extent      = extent;
image_info.samples     = sample_count;
image_info.usage       = VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT;

VmaAllocation memory;
VmaAllocationCreateInfo memory_info{};
memory_info.usage = memory_usage;
memory_info.preferredFlags = VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT;

auto result = vmaCreateImage(device.get_memory_allocator(), &image_info, &memory_info, &handle, &memory, nullptr);
```

Load/Store operations sample



Render Passes

GPU: Mali-G71

Fragment Cycles: 112.8 M/s

External Read Bytes: 1479.4 MiB/s

External Write Bytes: 820.9 MiB/s

☑ Use vkCmdClearAttachments (color)

Color attachment load operation: ● Load ○ Clear ○ Don't care

Depth attachment store operation: ● Store ○ Don't care

**counters may be affected by screen recording and other factors such as framebuffer compression and transaction elimination*

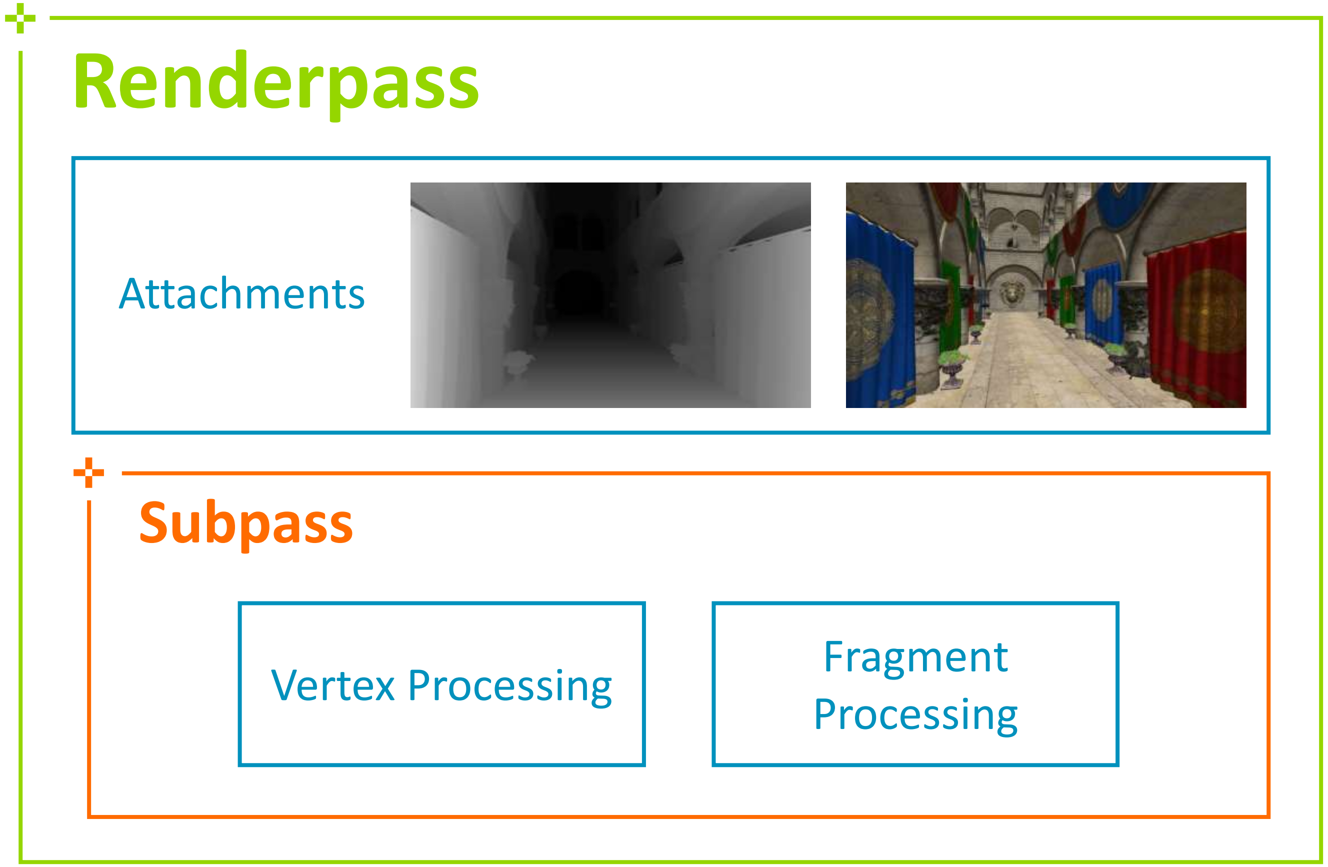
36%
reduction in external read bytes with LOAD_OP_CLEAR

62%
reduction in external write bytes with STORE_OP_DONT_CARE

7%
reduction in fragment cycles with LOAD_OP_CLEAR

Subpasses

Renderpasses and subpasses



Renderpass

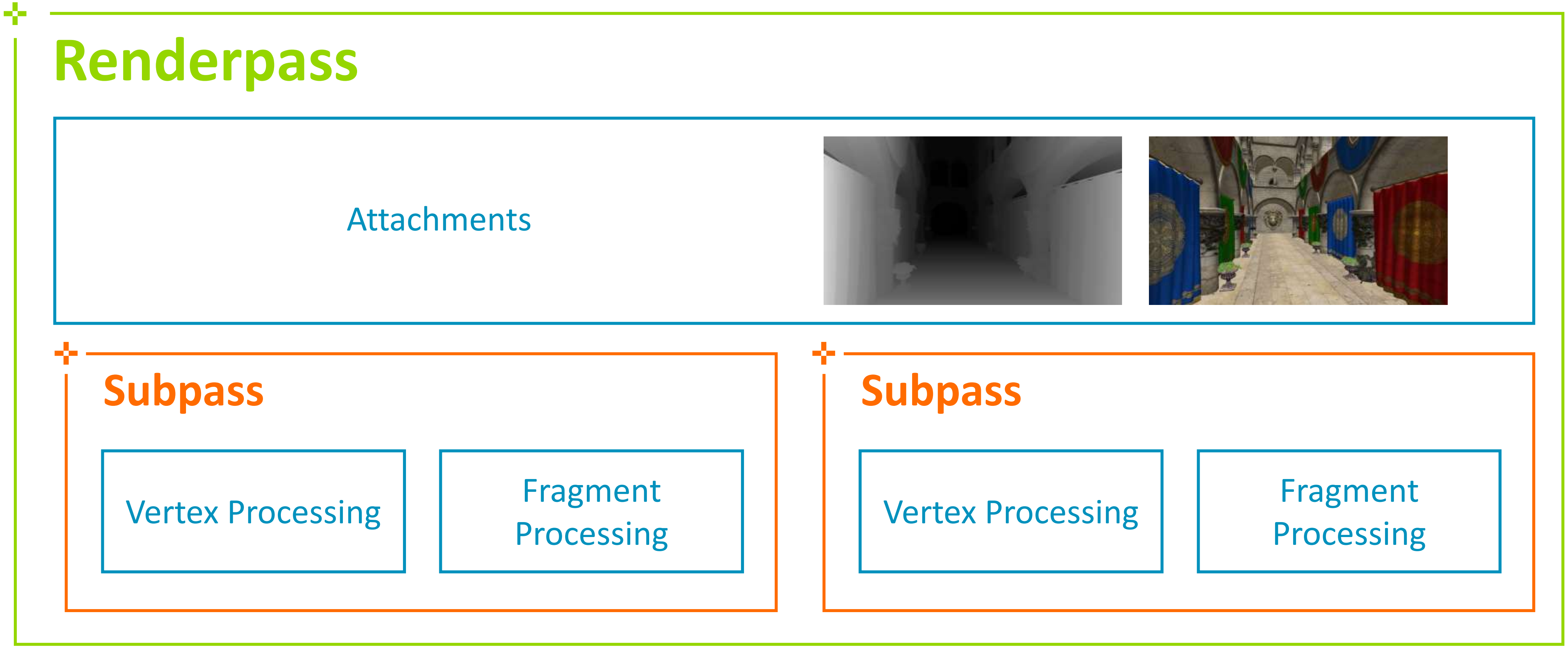
Attachments

Subpass

Vertex Processing

Fragment Processing

Renderpasses and subpasses



Renderpass

Attachments

Subpass

Vertex Processing

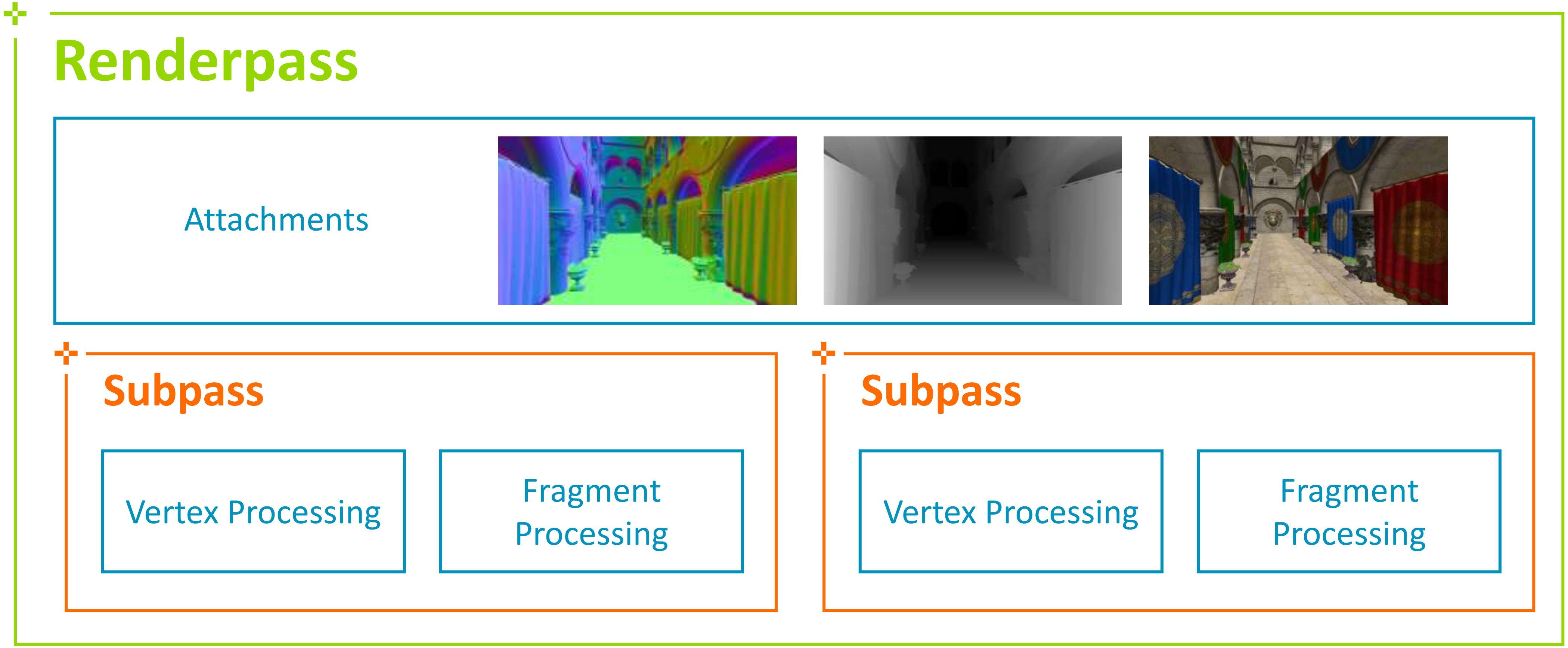
Fragment Processing

Subpass

Vertex Processing

Fragment Processing

Multipass deferred



Renderpass

Attachments

Subpass

Vertex Processing

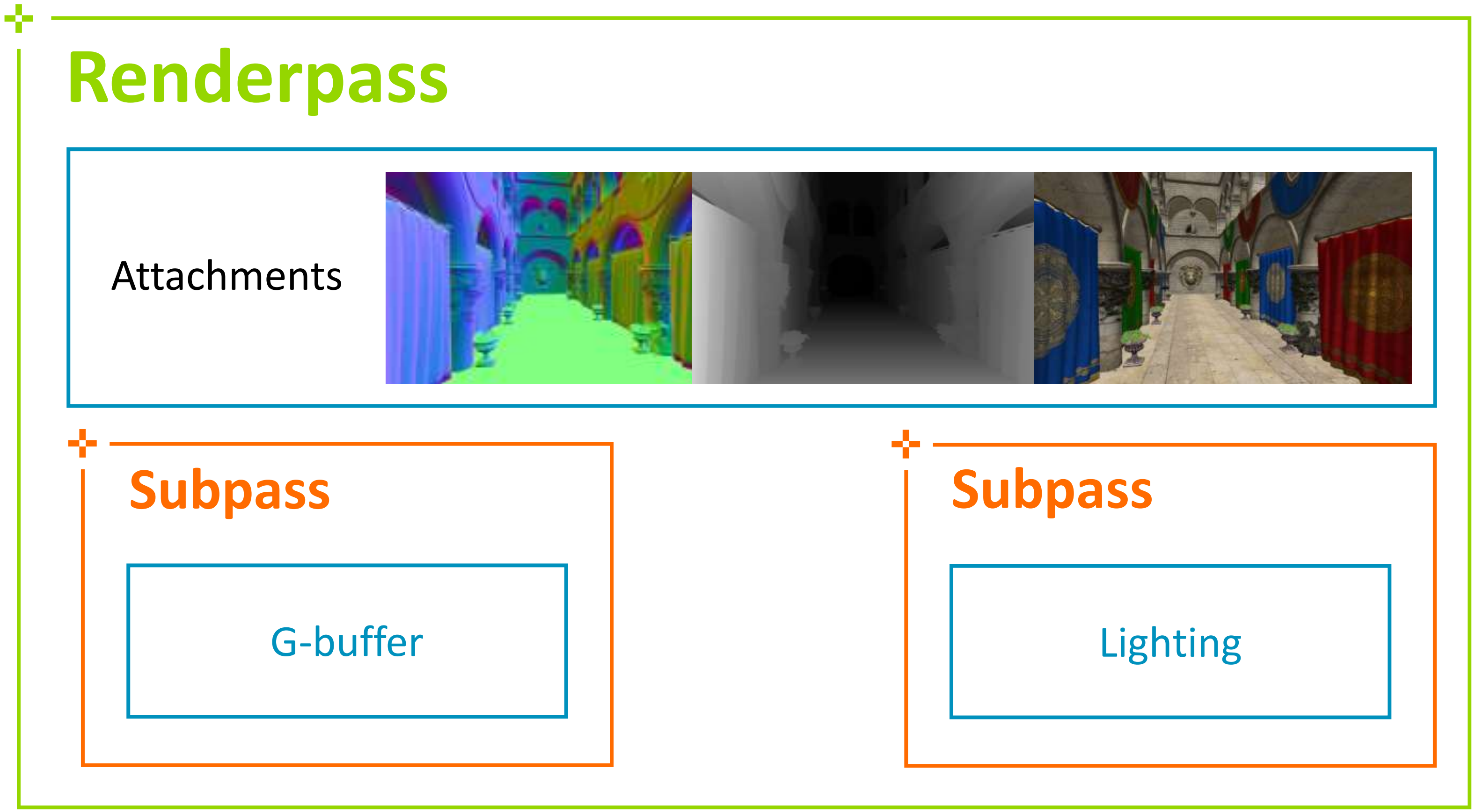
Fragment Processing

Subpass

Vertex Processing

Fragment Processing

Multipass deferred



Renderpass

Attachments

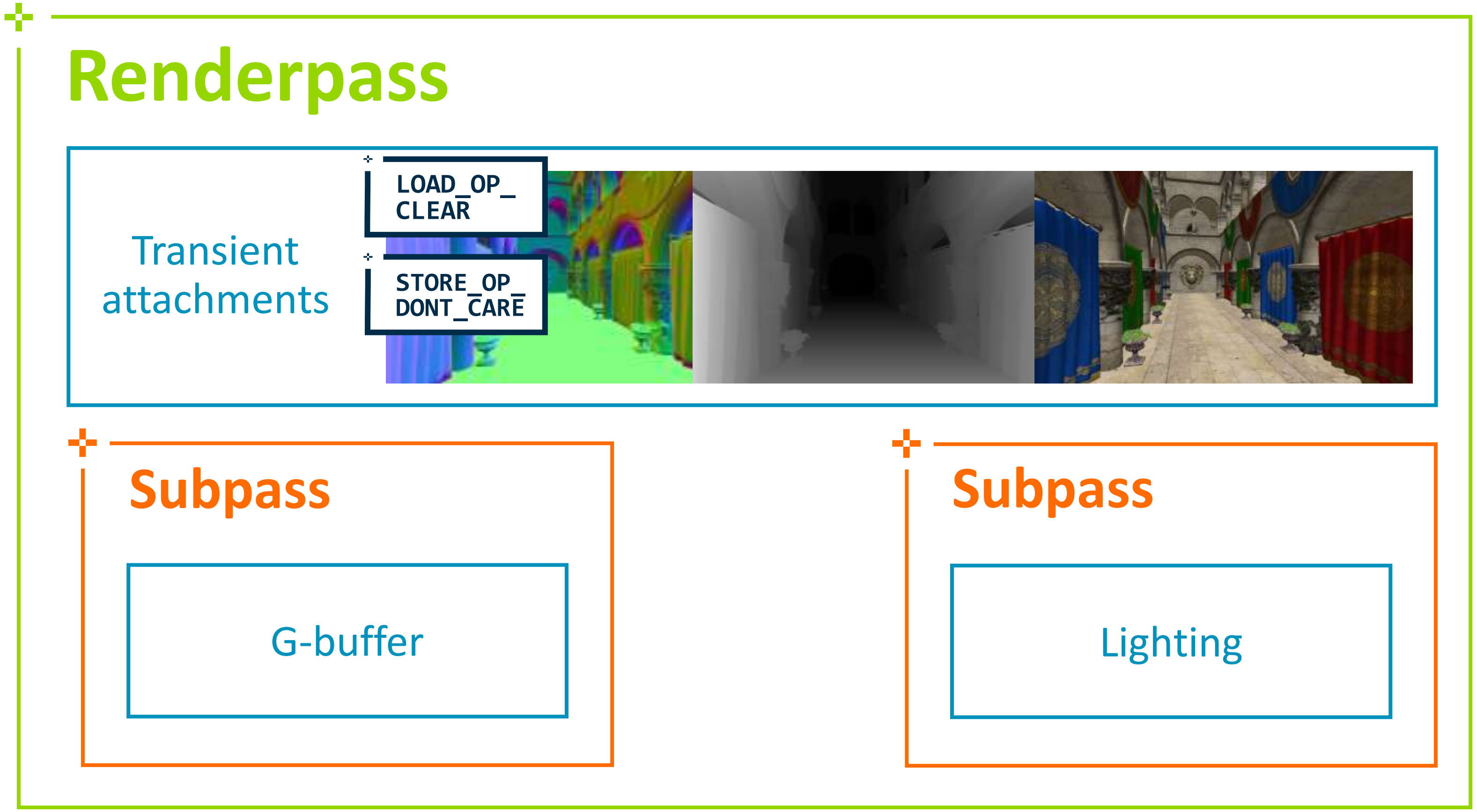
Subpass

G-buffer

Subpass

Lighting

Multipass deferred



Renderpass

Transient attachments

`LOAD_OP_CLEAR`

`STORE_OP_DONT_CARE`

Subpass

G-buffer

Subpass

Lighting

Subpass fusion

- We can express a per-pixel dependency between G-Buffer and Lighting subpasses
- Subpass information is known ahead of time (`VkRenderPass`)
- Driver can merge two or more sub-passes into one Renderpass if they have
  - `BY_REGION` dependencies
  - no external side effects which might prevent fusing
- `vkCmdNextSubpass()` essentially becomes a no-op
- The G-Buffer is transient, saving a lot of external memory bandwidth
- Special image type in SPIR-V, use `subpassInput` and `subpassLoad()` in GLSL
- Extension in GLES (PLS) now core functionality in Vulkan

```
VkSubpassDependency subpassDependency = {};
subpassDependency.srcSubpass = 0;
subpassDependency.dstSubpass = 1;

subpassDependency.dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT;
```

Subpasses sample

Render Subpasses

GPU: Mali-G76

Tiles: 1340.5 k/s

Fragment Jobs: 273/s

External Read Bytes: 1238.5 MiB/s

External Write Bytes: 940.7 MiB/s

Render technique: ○ Subpasses ● Renderpasses

Transient attachments: ● Enabled ○ Disabled

G-Buffer size: ● 128-bit ○ More

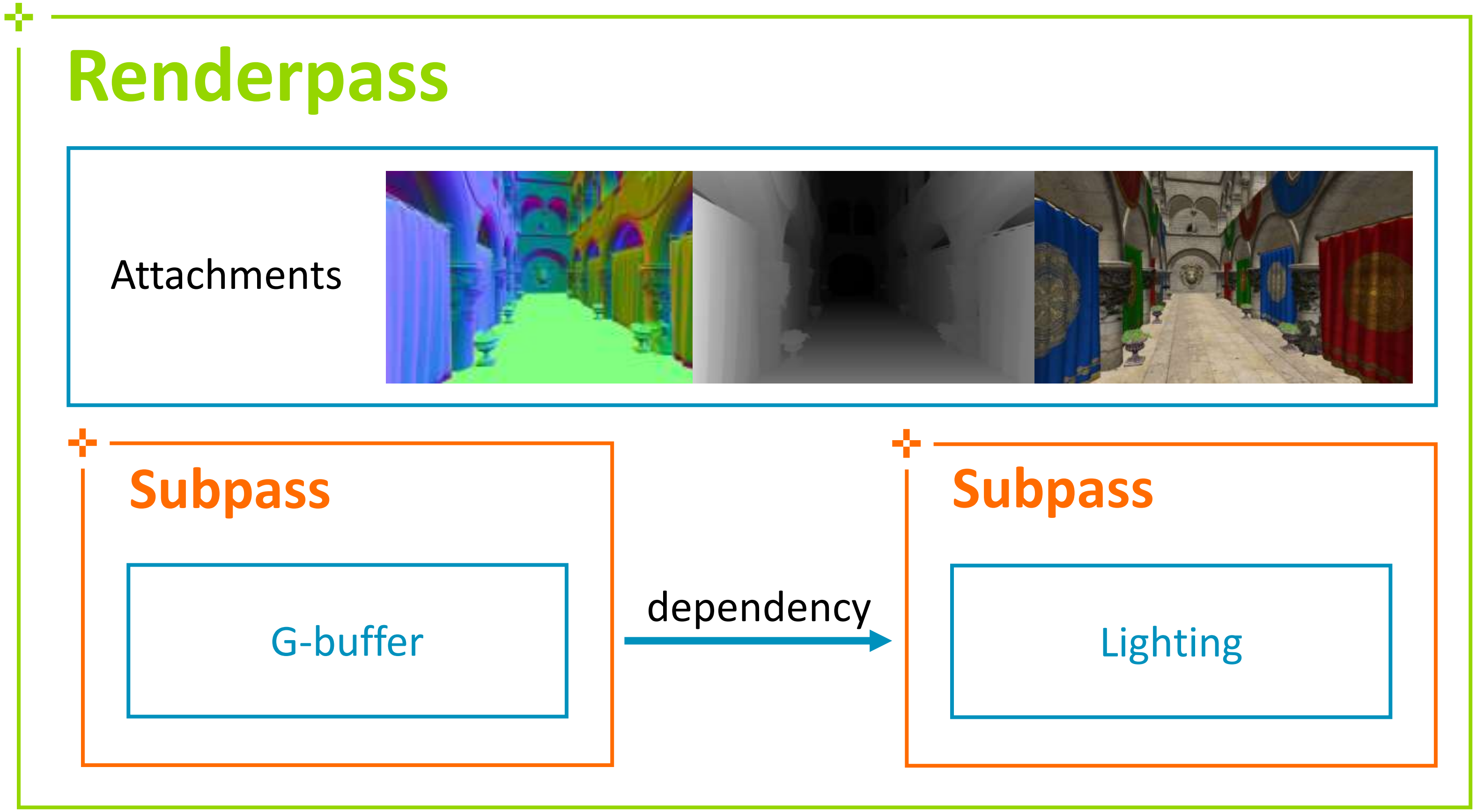
**counters may be affected by screen recording and other factors such as framebuffer compression and transaction elimination*

45%
reduction in external read bytes

56%
reduction in external write bytes

Pipeline barriers

Multipass deferred



Renderpass

Attachments

Subpass

G-buffer

dependency

Subpass

Lighting

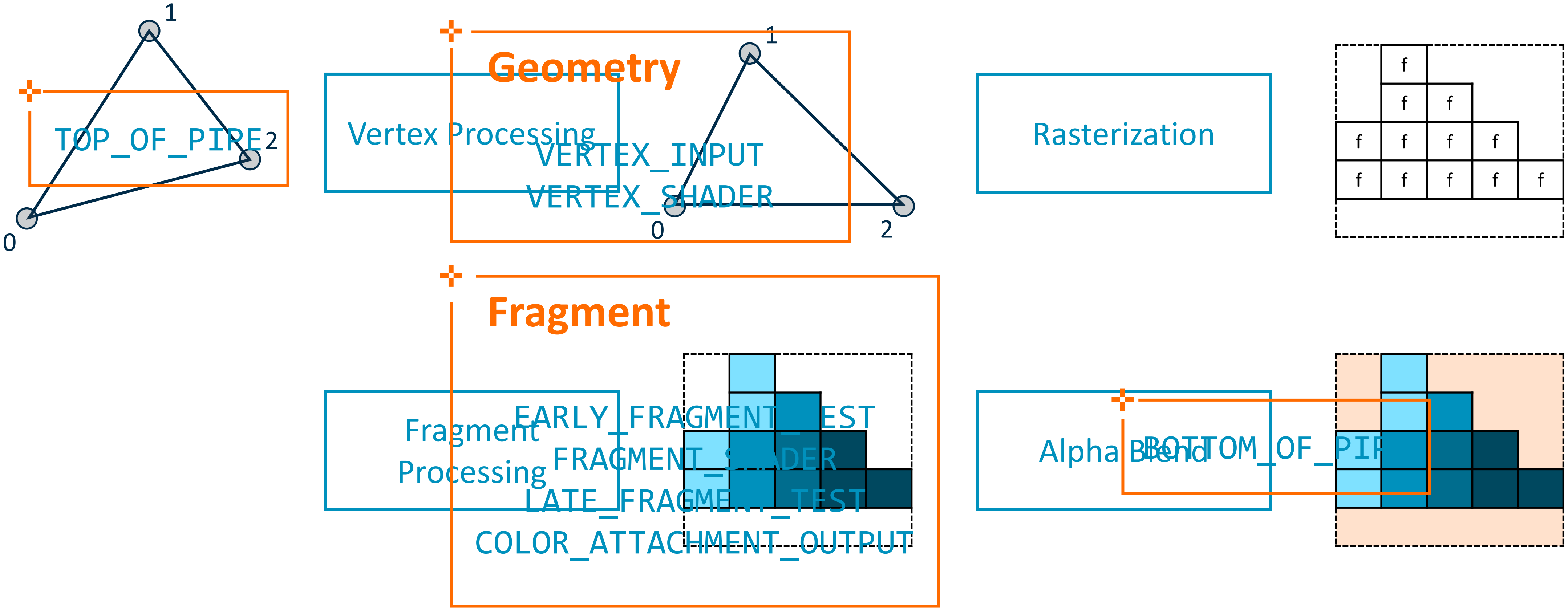
Pipeline stages

```
typedef enum VkPipelineStageFlagBits {
           VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT = 0x00000001,
           VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT = 0x00000002,
           VK_PIPELINE_STAGE_VERTEX_INPUT_BIT = 0x00000004,
           VK_PIPELINE_STAGE_VERTEX_SHADER_BIT = 0x00000008,
           VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT = 0x00000010,
           VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT = 0x00000020,
           VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT = 0x00000040,
           VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT = 0x00000080,
           VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT = 0x00000100,
           VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT = 0x00000200,
           VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT = 0x00000400,
           VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT = 0x00000800,
           VK_PIPELINE_STAGE_TRANSFER_BIT = 0x00001000,
           VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT = 0x00002000,
           VK_PIPELINE_STAGE_HOST_BIT = 0x00004000,
           VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT = 0x00008000,
           VK_PIPELINE_STAGE_ALL_COMMANDS_BIT = 0x00010000,
           VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT = 0x01000000,
           VK_PIPELINE_STAGE_CONDITIONAL_RENDERING_BIT_EXT = 0x00040000,
           VK_PIPELINE_STAGE_COMMAND_PROCESS_BIT_NVX = 0x00020000,
           VK_PIPELINE_STAGE_SHADING_RATE_IMAGE_BIT_NV = 0x00400000,
           VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_NV = 0x00200000,
           VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_NV = 0x02000000,
           VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV = 0x00080000,
           VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV = 0x00100000,
           VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT = 0x00800000,
           VK_PIPELINE_STAGE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkPipelineStageFlagBits;
```

Pipeline stages

```
typedef enum VkPipelineStageFlagBits {
           VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT = 0x00000001,
           VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT = 0x00000002,
           VK_PIPELINE_STAGE_VERTEX_INPUT_BIT = 0x00000004,
           VK_PIPELINE_STAGE_VERTEX_SHADER_BIT = 0x00000008,
           VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT = 0x00000010,
           VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT = 0x00000020,
           VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT = 0x00000040,
           VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT = 0x00000080,
           VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT = 0x00000100,
           VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT = 0x00000200,
           VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT = 0x00000400,
           VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT = 0x00000800,
           VK_PIPELINE_STAGE_TRANSFER_BIT = 0x00001000,
           VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT = 0x00002000,
           VK_PIPELINE_STAGE_HOST_BIT = 0x00004000,
           VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT = 0x00008000,
           VK_PIPELINE_STAGE_ALL_COMMANDS_BIT = 0x00010000,
           VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT = 0x01000000,
           VK_PIPELINE_STAGE_CONDITIONAL_RENDERING_BIT_EXT = 0x00040000,
           VK_PIPELINE_STAGE_COMMAND_PROCESS_BIT_NVX = 0x00020000,
           VK_PIPELINE_STAGE_SHADING_RATE_IMAGE_BIT_NV = 0x00400000,
           VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_NV = 0x00200000,
           VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_NV = 0x02000000,
           VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV = 0x00080000,
           VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV = 0x00100000,
           VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT = 0x00800000,
           VK_PIPELINE_STAGE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkPipelineStageFlagBits;
```

The graphics pipeline



TOP_OF_PIPE

Vertex Processing

Geometry

VERTEX_INPUT

VERTEX_SHADER

Rasterization

Fragment

Fragment Processing

EARLY_FRAGMENT_TEST

FRAGMENT_SHADER

LATE_FRAGMENT_TEST

COLOR_ATTACHMENT_OUTPUT

Alpha Blend

BOTTOM_OF_PIP

The graphics pipeline

TOP_OF_PIPE

Geometry

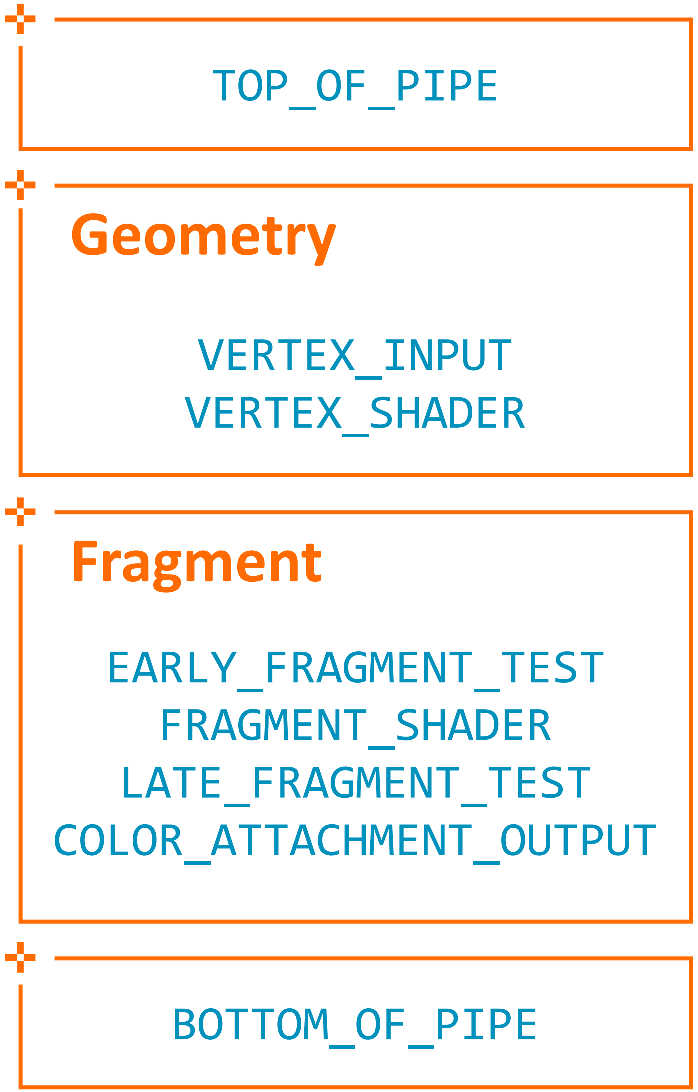
VERTEX_INPUT
VERTEX_SHADER

Fragment

EARLY_FRAGMENT_TEST
FRAGMENT_SHADER
LATE_FRAGMENT_TEST
COLOR_ATTACHMENT_OUTPUT

BOTTOM_OF_PIPE

Pipeline barriers



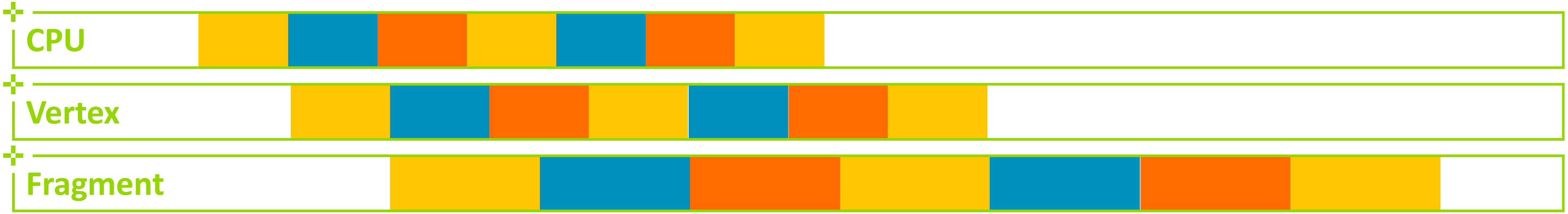
```
void vkCmdPipelineBarrier(
           VkCommandBuffer                  commandBuffer,
           VkPipelineStageFlags             srcStageMask,
           VkPipelineStageFlags             dstStageMask,
           VkDependencyFlags                dependencyFlags,
           uint32_t                         memoryBarrierCount,
           const VkMemoryBarrier*           pMemoryBarriers,
           uint32_t                         bufferMemoryBarrierCount,
           const VkBufferMemoryBarrier*     pBufferMemoryBarriers,
           uint32_t                         imageMemoryBarrierCount,
           const VkImageMemoryBarrier*      pImageMemoryBarriers
);
```

- A barrier splits the command stream in two
- It will synchronize everything before, and after the barrier
- `srcStageMask` specifies what we are waiting for
- `dstStageMask` specifies what stages will wait

Pipelining: avoid BOTTOM->TOP dependencies

```
vkCmdPipelineBarrier(
          command_buffer,
          VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT,
          VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT,
          0,
          0, nullptr,
          0, nullptr,
          1, &image_memory_barrier)
```

Pipelining: avoid BOTTOM->TOP dependencies



```
vkCmdPipelineBarrier(
           command_buffer,
           VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT,
           VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,
           0,
           0, nullptr,
           0, nullptr,
           1, &image_memory_barrier)
```

Pipeline barriers sample

Pipeline Barriers

GPU: Mali-G76

Frame Times: 19.0 ms

Vertex Compute Cycles: 73.9 M/s

Fragment Cycles: 605.7 M/s

Pipeline barrier stages:

● Bottom to top ● Frag to vert ● Frag to frag

**counters may be affected by screen recording. See this presentation by Samsung GameDev and Croteam*

Up to **56%**
reduction in frame time

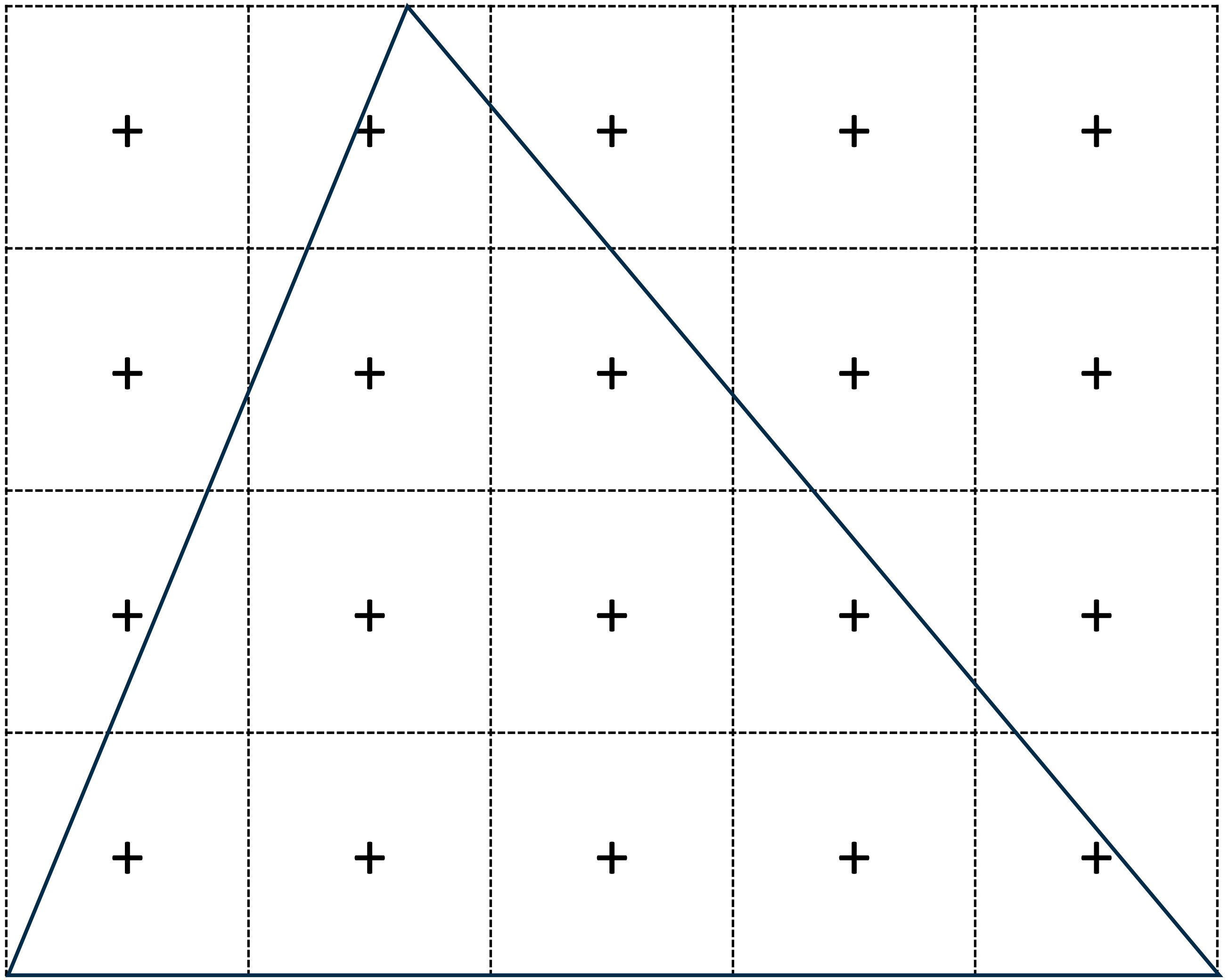
MSAA

Multisample anti-aliasing (MSAA)

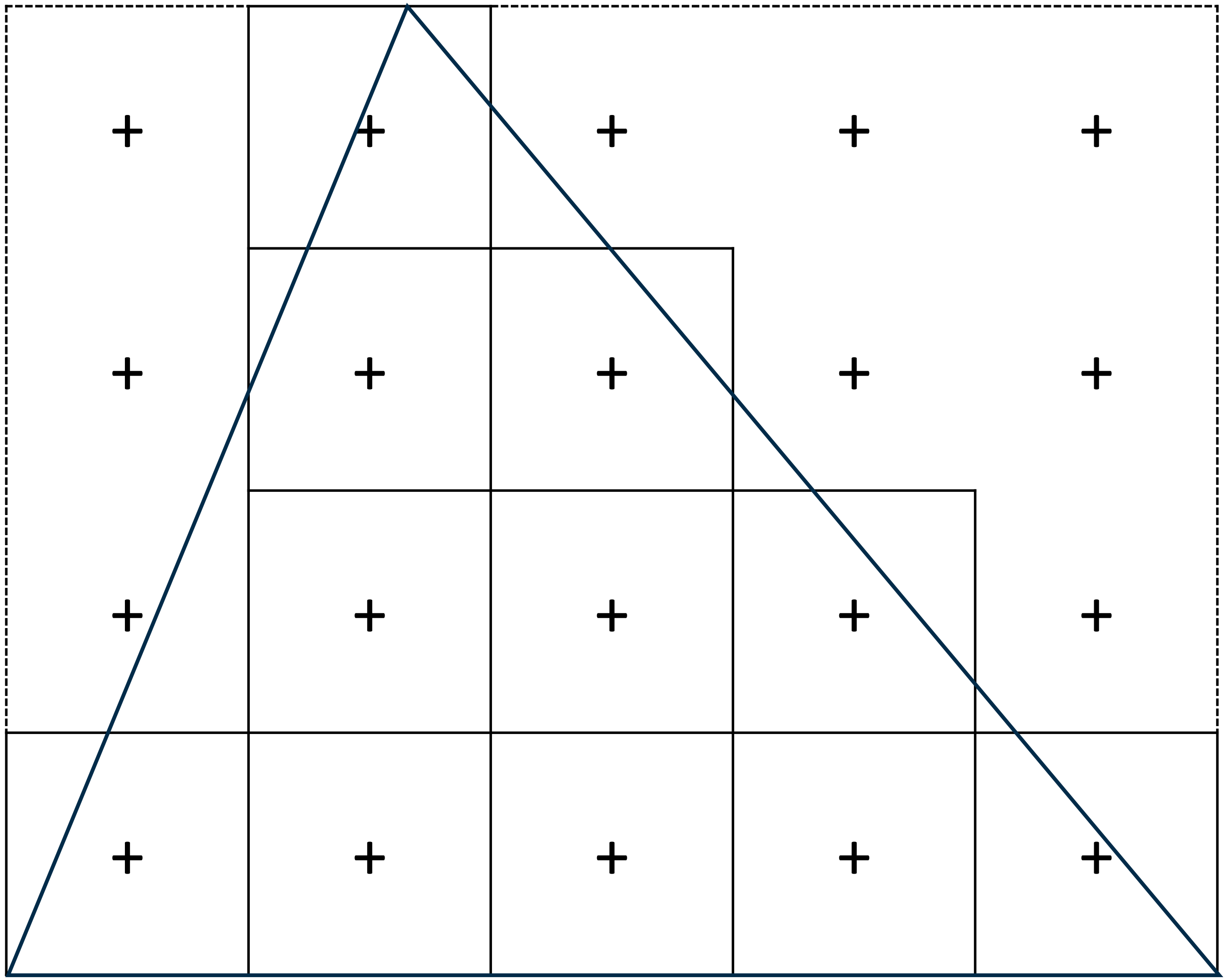
Multisample anti-aliasing (MSAA)

Multisample anti-aliasing (MSAA)

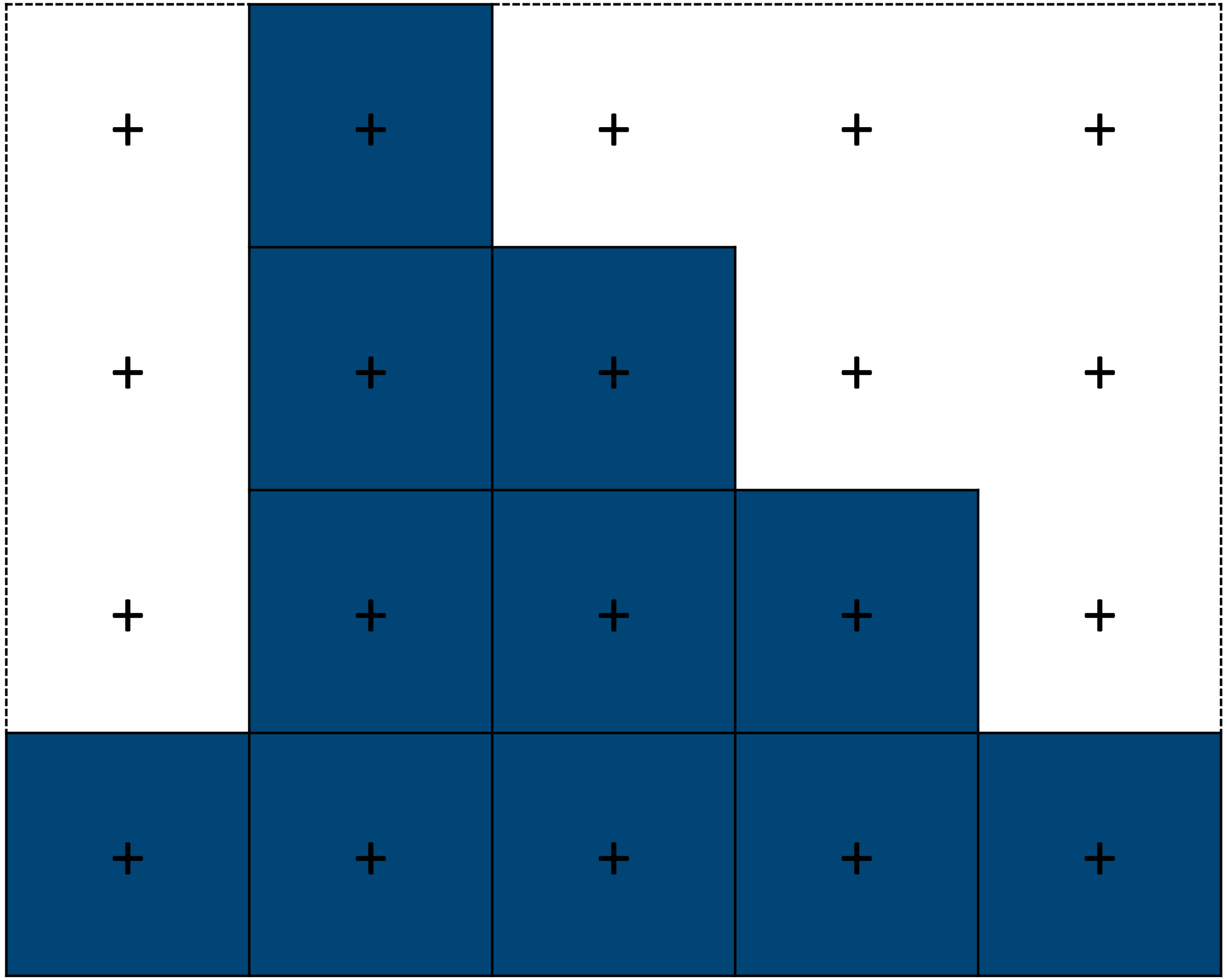
No MSAA



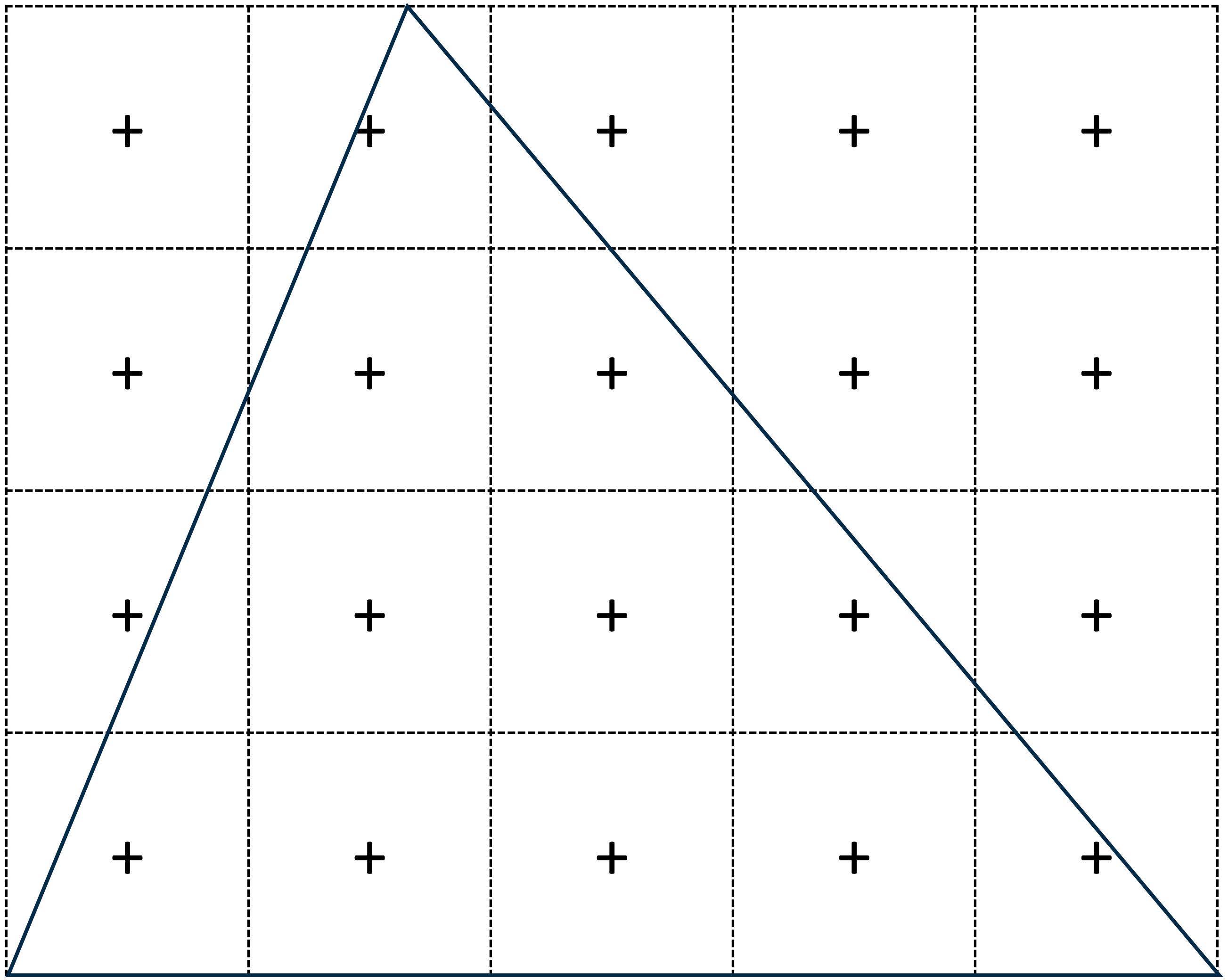
No MSAA



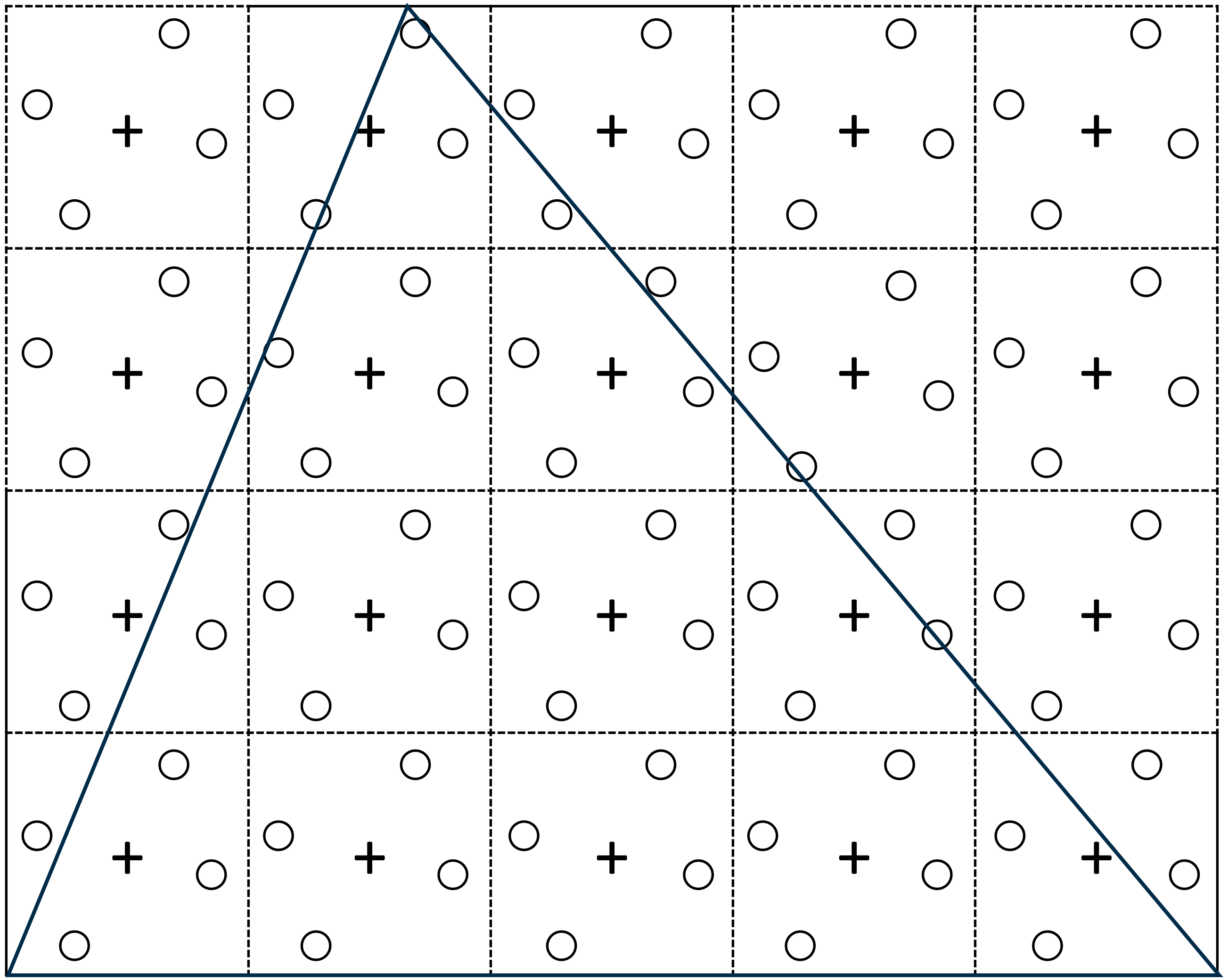
No MSAA



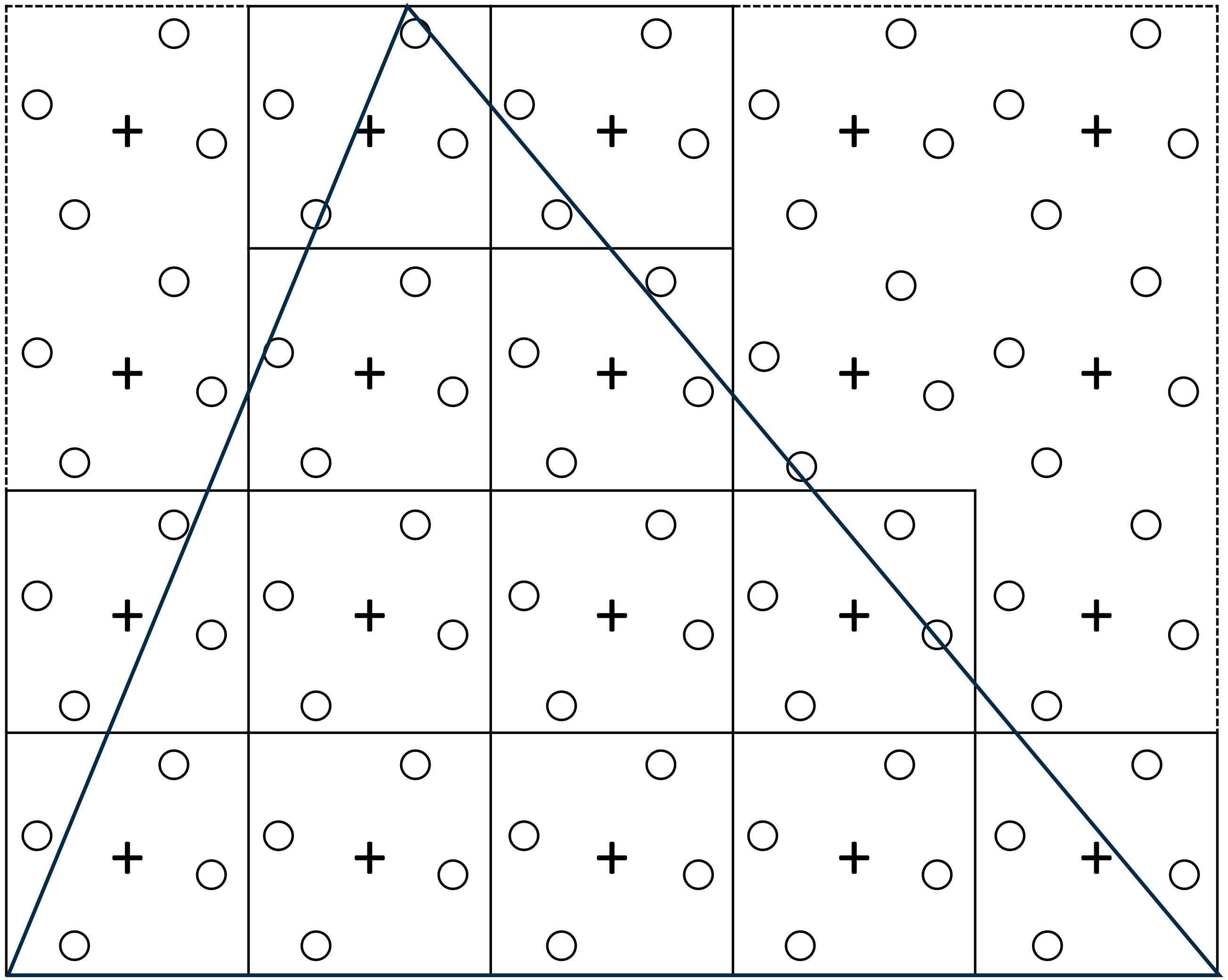
MSAA



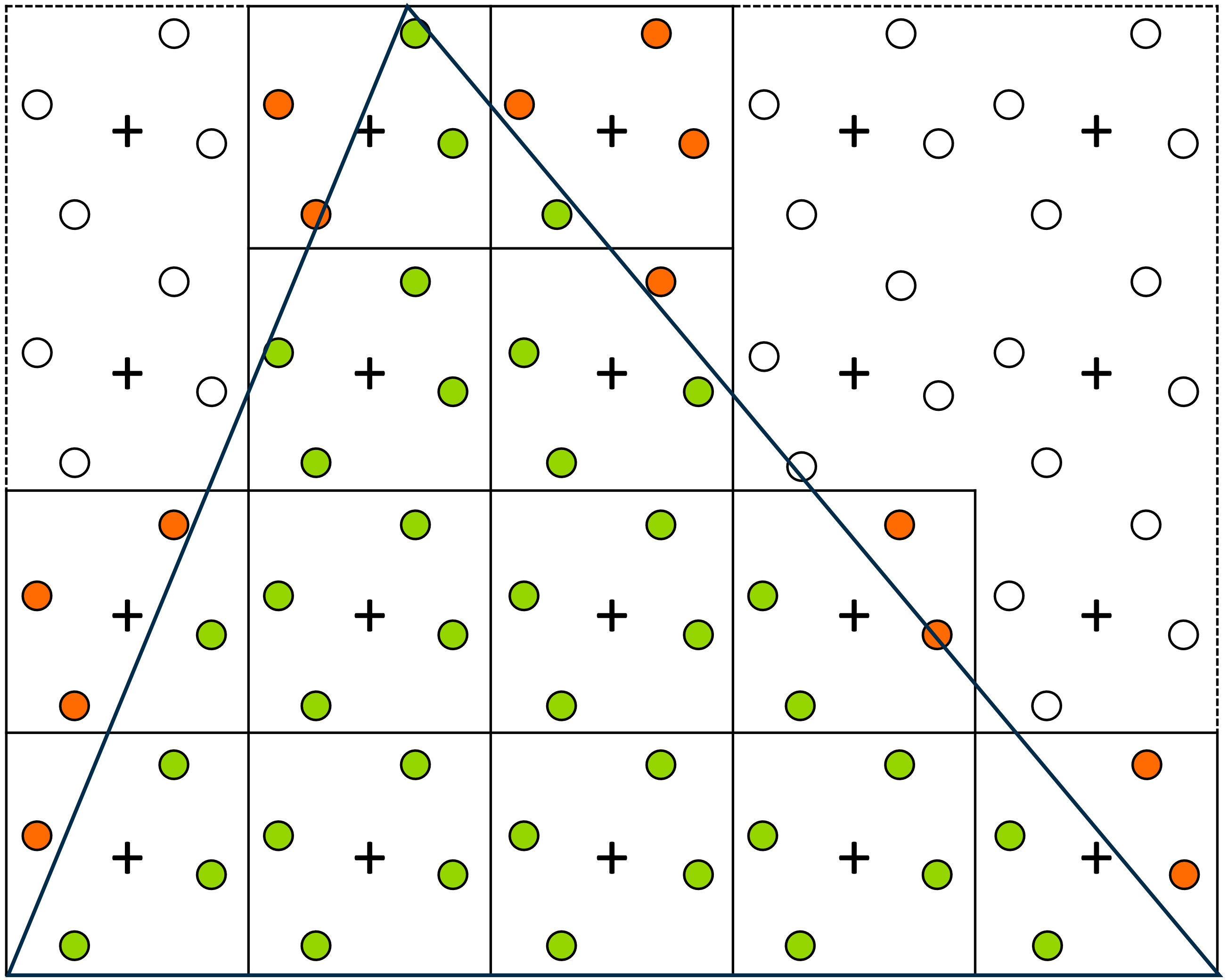
MSAA



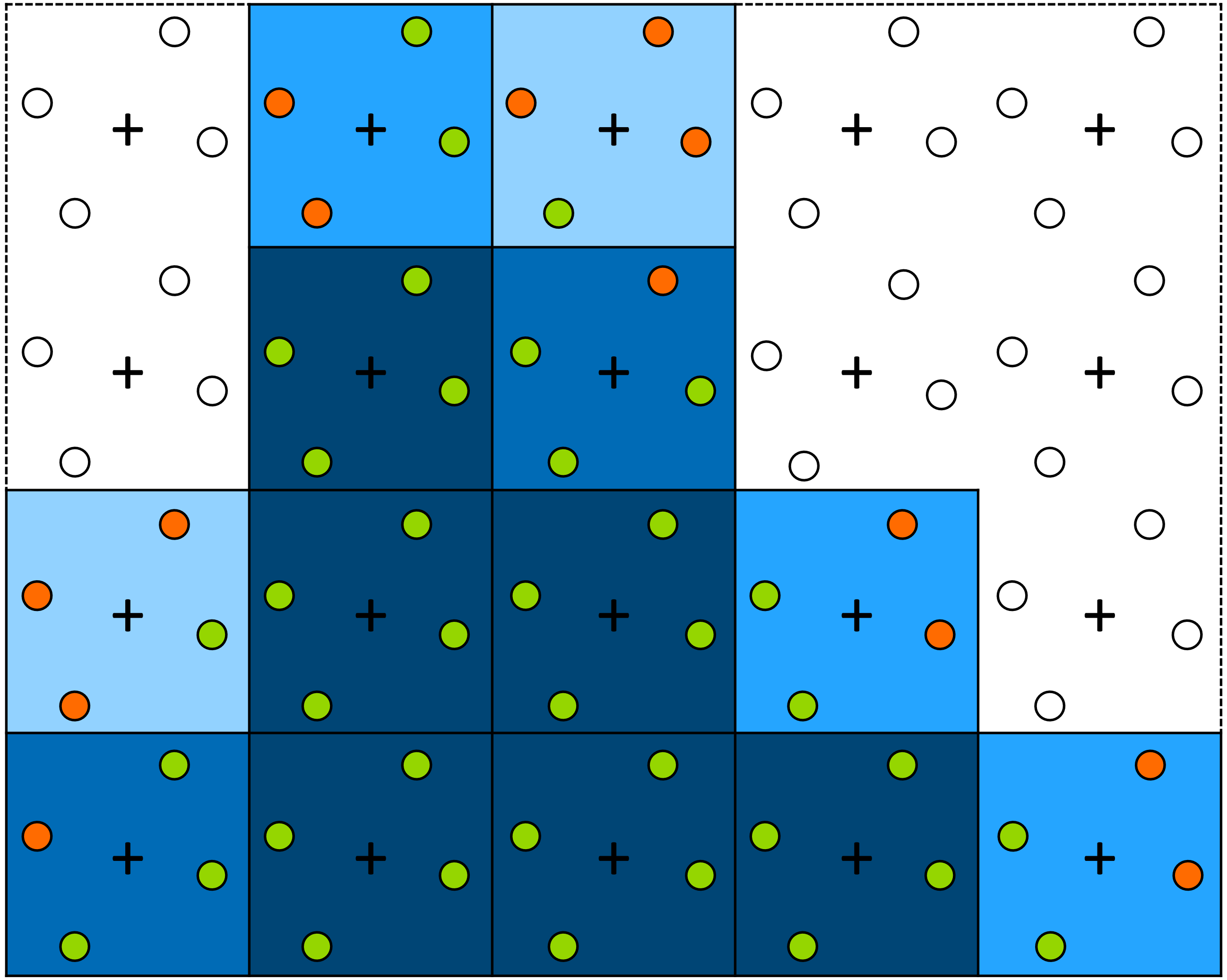
MSAA



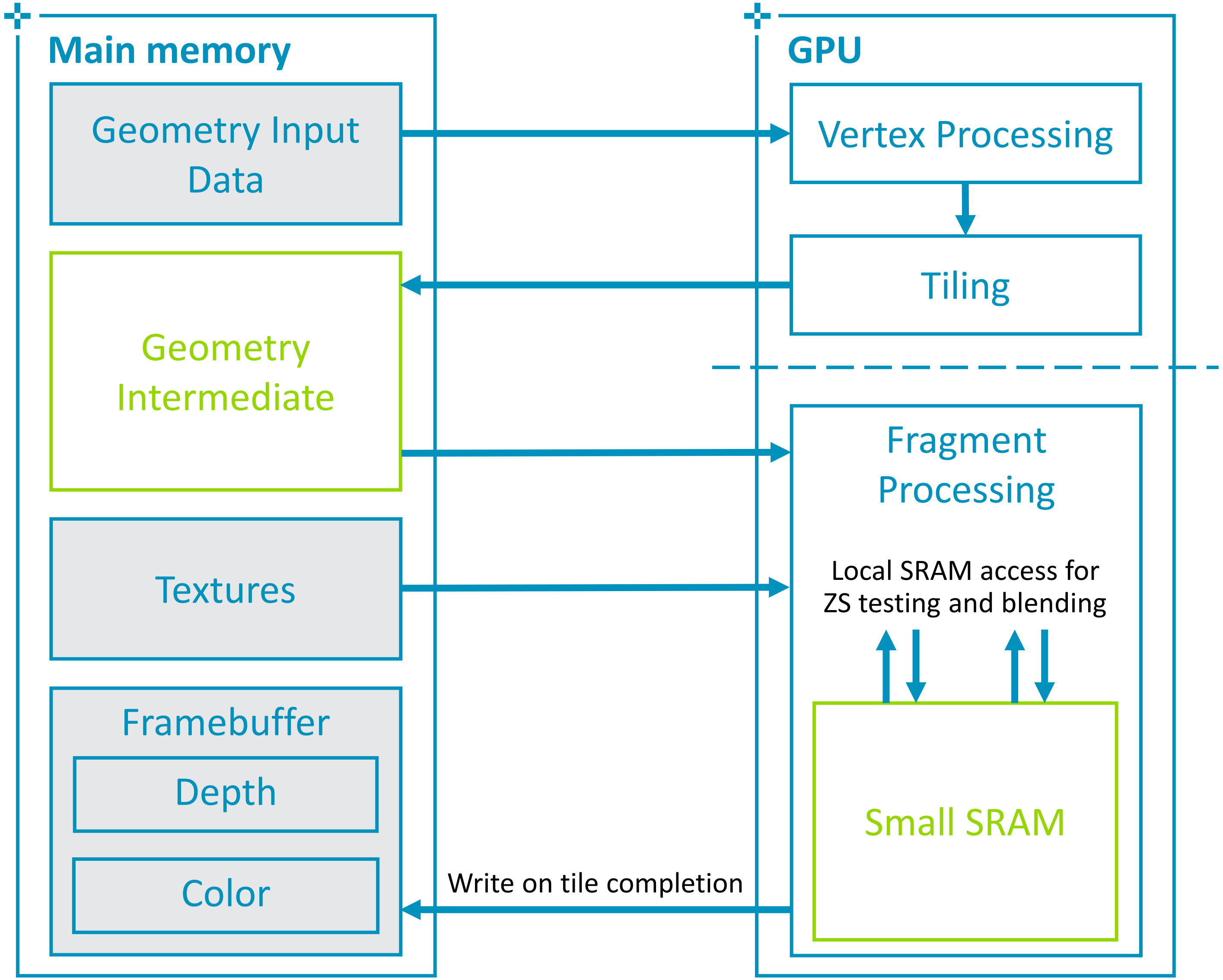
MSAA



MSAA



Tiled GPUs



Main memory

Geometry Input Data

Geometry Intermediate

Textures

Framebuffer

Depth

Color

GPU

Vertex Processing

Tiling

Fragment Processing

Local SRAM access for ZS testing and blending

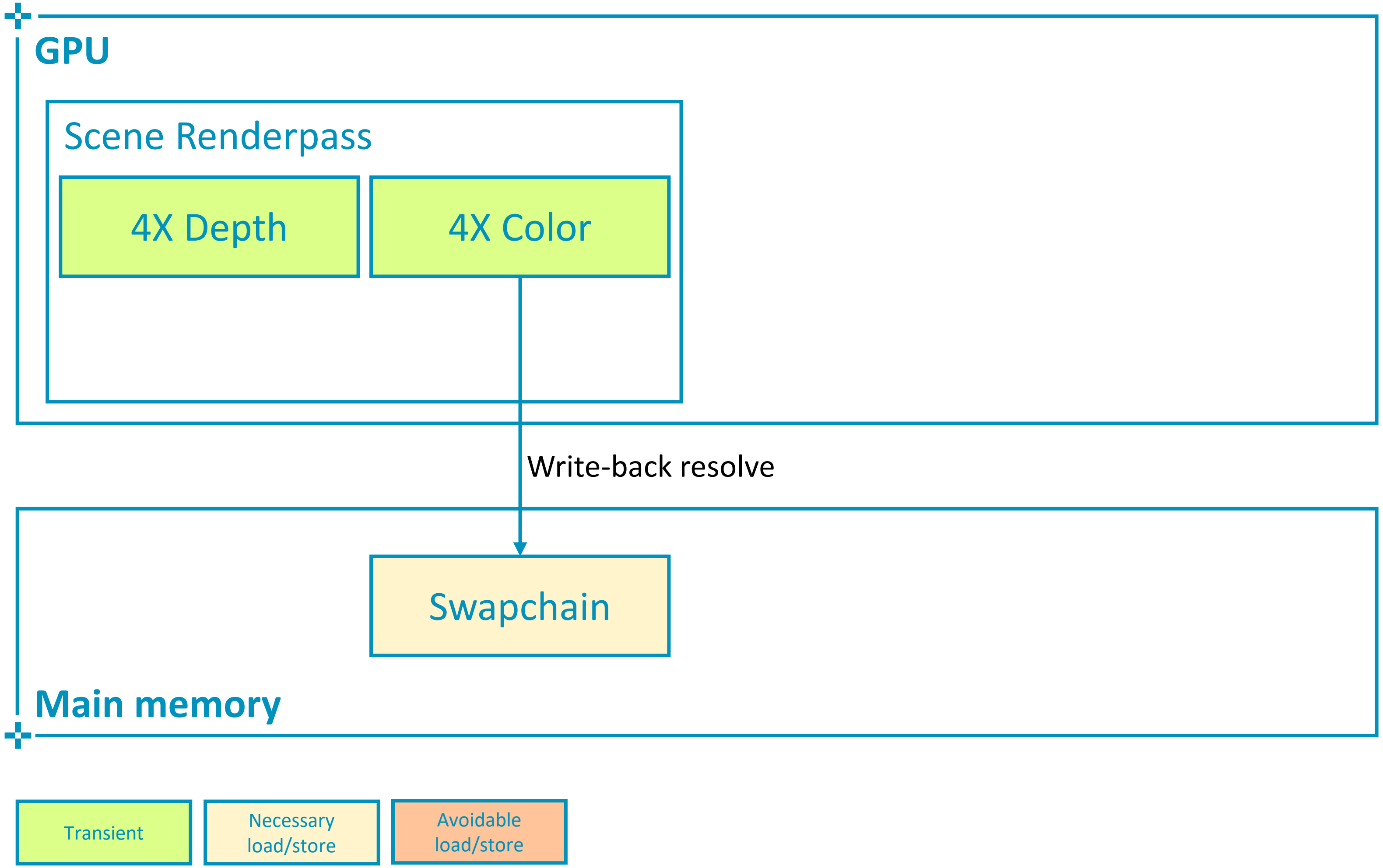
Small SRAM

Write on tile completion

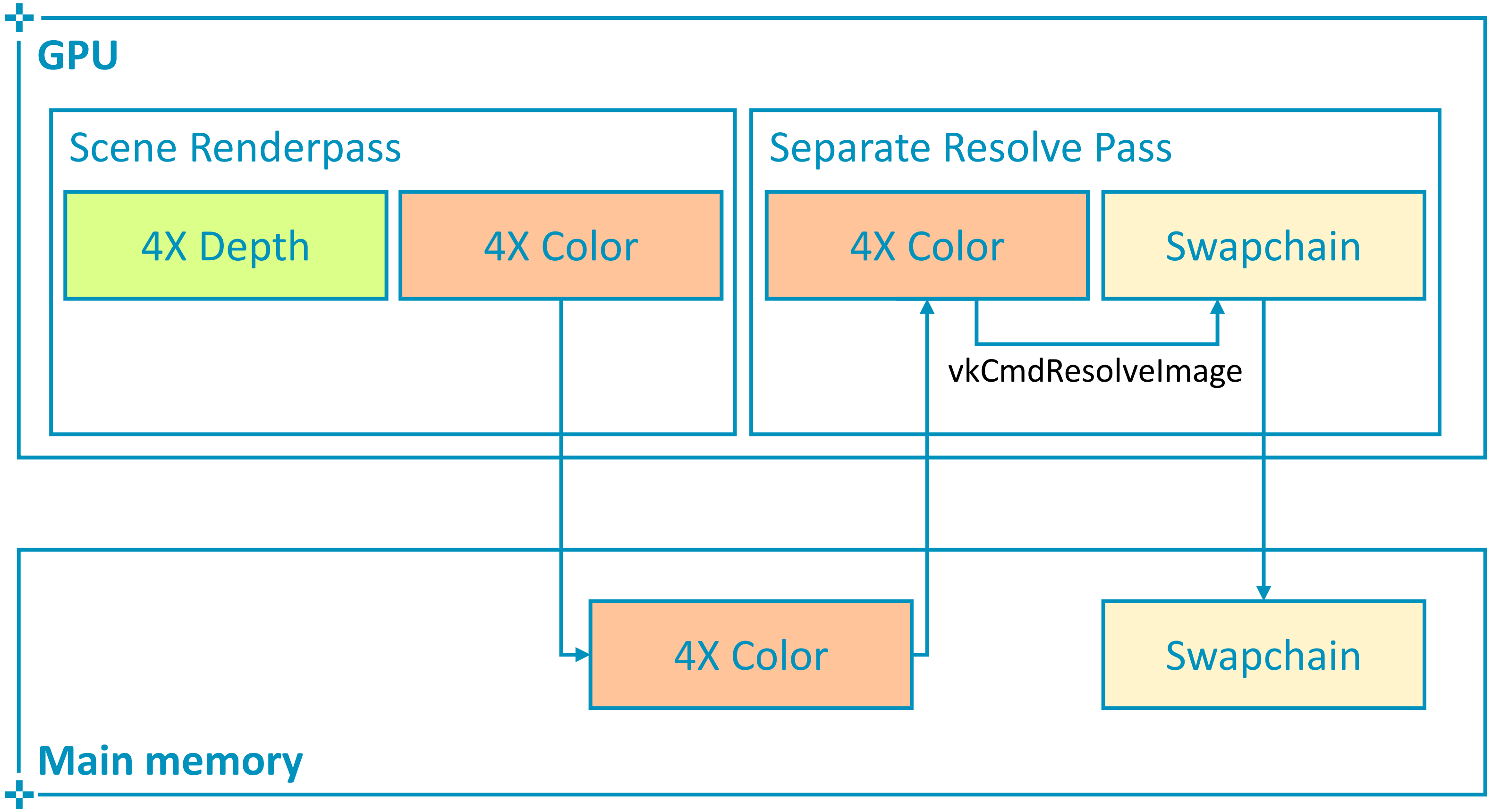
Resolve attachments

- Multisampled image may be transient
  - `loadOp = LOAD_OP_CLEAR`
  - `storeOp = STORE_OP_DONT_CARE`
  - Use `LAZILY_ALLOCATED` memory
- Use `pResolveAttachments` in a subpass to automatically resolve a multisampled color buffer into a single-sampled color buffer
- Avoid `vkCmdResolveImage()`: this has a significant negative impact on bandwidth and performance
- With `VK_KHR_depth_stencil_resolve` (core in Vulkan 1.2) the depth attachment may also be resolved in a similar fashion

Resolve on tile writeback (best practice)



Resolve in a separate pass (avoid!)

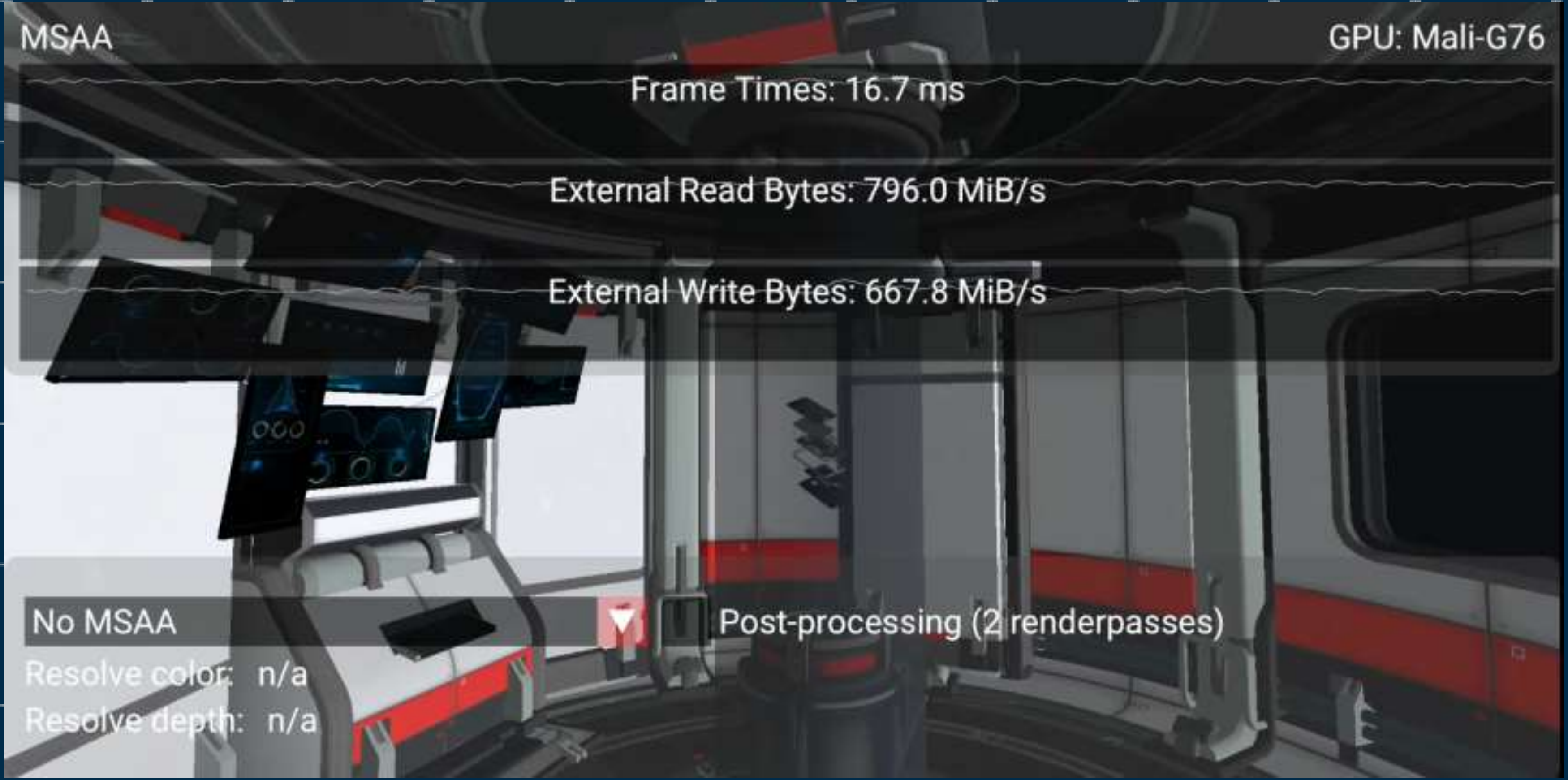


🟩 Transient

🟨 Necessary load/store

🟧 Avoidable load/store

MSAA sample



MSAA

GPU: Mali-G76

Frame Times: 16.7 ms

External Read Bytes: 796.0 MiB/s

External Write Bytes: 667.8 MiB/s

Resolve color: n/a

Resolve depth: n/a

**counters may be affected by screen recording and other factors such as framebuffer compression and transaction elimination*

261%

reduction in external read bytes

440%

reduction in external write bytes

Optimizing Roblox

Reducing CPU overhead of render dispatch

What is Roblox?

- Online multiplayer game creation platform
  - 100M+ MAU, 2.5M+ CCU
  - Windows, macOS, iOS, **Android**, Xbox One
  - Direct3D 9, Direct3D 11, OpenGL 2/3, OpenGL ES 2/3, Metal, **Vulkan**
- All content is user generated
  - A lot of content creators with a lot of varied content!
  - We don’t police quality or performance
  - Optimizing engine makes all content run better
- Historically geometry and draw call heavy
  - Levels are often built from basic primitives


Show Friends
Show Players
HP 542/542
63620
28
VIP
28
542/542
NEW!
Cosmetics
Inventory
Play

Optimizing draw call dispatch

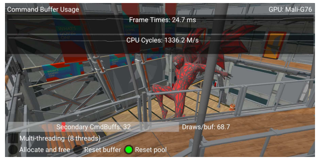
- Vulkan is implemented on top of a common rendering interface
  - How can we get maximum performance with reasonable effort?
- Focus on steady state performance
  - Cache everything that is easy to cache
  - Assume regular frame structure
- Minimize abstraction overhead
  - Find a compromise between ease of use and performance
  - Optimize the implementation as much as possible
- Threading-friendly implementation
  - Allow each thread to record draw calls in isolation

Optimizing draw call dispatch

```
// 1. Command buffer management
DeviceContext* ctx = device->createCommandBuffer();

PassClear passClear;
passClear.mask = Framebuffer::Mask_Color0;

// 2. Render passes
ctx->beginPass(fb, 0, Framebuffer::Mask_Color0, &passClear);

// 3. Pipeline state
ctx->bindProgram(program.get());

// 4. Descriptor management
ctx->bindBuffer(0, globalDataBuffer.get());
ctx->bindBufferData(1, &params, sizeof(params));
ctx->bindTexture(0, lightMap, SamplerState::Filter_Linear);

// 5. General optimizations
ctx->draw(geometry, Geometry::Primitive_Triangles, 0, count);

ctx->endPass();

device->commitCommandBuffer(ctx);
```

Command buffer management

Arm sample
[Allocation and management of command buffers]


Command Buffer Usage
GPU: Mali-G76
Frame Times: 24.7 ms
CPU Cycles: 1336.2 M/s
Secondary CmdBuffs: 32
Draws/buf: 68.7
☐ Multi-threading (8 threads)
○ Allocate and free ○ Reset buffer ● Reset pool

- Deceptively simple…
  - createCommandBuffer() => vkAllocateCommandBuffers
  - commitCommandBuffer() => vkQueueSubmit
- … but actually complicated
  - Each thread needs a separate VkCommandPool to allocate from
  - VkCommandPool can not be used if command buffers allocated from it are in flight
  - vkAllocateCommandBuffers is not free
  - vkFreeCommandBuffers doesn’t always recycle command memory
  - vkQueueSubmit can be expensive

Command buffer management

- Pool of command pools
  - createCommandBuffer() steals a VkCommandPool (or creates one) under a critical section
  - We never free command buffers, and reuse allocated ones
- Batched command buffer submissions
  - commitCommandBuffer() adds the command buffer to frame list and returns the pool
  - A single vkQueueSubmit at the end of the frame with submitCount = 1
- Command pool recycling
  - After recording a frame we remove all pools with pending command buffers from global pool
  - After a frame has completed, we put all pools back into global pool
  - Don’t forget to run vkResetCommandPool!
    - This automatically resets all allocated command buffers and puts them into ready state

Render passes

Arm samples
Load/store actions
Layout transitions
Pipeline barriers
Subpasses

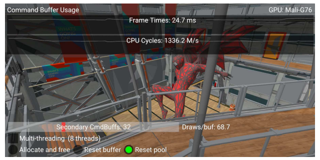
- Many complex topics!
  - Load/store actions
  - Image layout transitions
  - Pipeline barriers
- No global view of the frame
  - Immediate mode command submission
  - Each thread records commands in isolation
- We specify all information in beginPass() precisely
  - A full set of textures to render to (color/depth)
  - Which framebuffer textures need to be loaded from memory?
  - Which framebuffer textures need to be stored to memory?
  - Which framebuffer textures need to be cleared with what initial data?
  - Do we need to do MSAA resolve in endPass() and if so, where?

Render passes

- Lazily create / cache VkRenderPass / VkFramebuffer
  - This includes load/store actions, image layout transitions, barriers and resolve!
  - Load/store actions are specified explicitly, the rest is inferred
- Inferring image layout transitions
  - No concept of “current” resource state – not threading-friendly
  - Instead, “default” resource state – for each resource, what state is it in between passes?
    - For textures with shader access this is SHADER_READ_ONLY
    - For textures without shader access this is COLOR_ATTACHMENT_OPTIMAL (or DEPTH)
    - For read/write textures this is GENERAL
  - All image layout transitions are performed at the pass boundary – no in-pass synchronization!
  - All image layout transitions are guided by load/store masks
    - An image that is not loaded is transitioned from UNDEFINED to COLOR_ATTACHMENT
    - An image that is not stored is kept in COLOR_ATTACHMENT (or DEPTH_ATTACHMENT)
- Inferring pipeline barriers
  - Default to VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
  - Extra annotations required to read textures from vertex/compute

Pipeline state

Arm sample
Pipeline cache



Command Buffer Usage
GPU: Mali-G76
Frame Times: 24.7 ms
CPU Cycles: 1336.2 M/s
Secondary CmdBuffs: 32
Draws/buf: 68.7
Multi-threading (8 threads)
○ Allocate and free ○ Reset buffer ● Reset pool

- Lazily create / cache VkPipeline objects
  - To fix frame stalls, use VkPipelineCache serialized to disk and cache pre-warming*
- Automatically filter redundant binds – cheap!
- Use lock-free read / locked write cache for pipeline states
  - Two hash tables from Key to State: read-only and read-write
  - Read-write table gets merged into the read-only table at the end of the frame

```
// do we have the key in readMap? thread-safe since readMap is only
// ever written to from flush() that runs exclusively to all tasks
if (V* rv = readMap.find(key)) {
    return rv;
}
mutex.lock();
// do we have the key in writeMap? thread-safe since we locked mutex
if (V* wv = writeMap.find(key)) {
    V value = *wv;
    mutex.unlock();
    return value;
}
// create the cache entry and add it to writeMap
```

Descriptor management

Arm sample
Descriptor
management

- Slot-based binding model
- This should look familiar and yet it’s not
  - Coupled textures and samplers (OpenGL ☹)
  - Only two namespaces, buffers and textures
    - No per-stage namespaces (constant buffer #3 is bound to the entire pipeline)
    - No difference between constant buffers and shader storage buffers
    - No difference between read-write (UAV) slots and read slots
  - An option to specify constant buffer data
  - Works surprisingly well for Metal/Vulkan!

```
void bindBuffer(unsigned int slot, Buffer* buffer);
void bindBufferRw(unsigned int slot, Buffer* buffer);
void bindBufferData(unsigned int slot, const void* data, unsigned int size);
void bindTexture(unsigned int slot, Texture* texture, SamplerState state);
void bindTextureRw(unsigned int slot, Texture* texture);
```

- Descriptor set layouts configured from shader reflection metadata
  - Validate compatibility between stages, e.g. uniform buffer #5 must be uniform in VS & FS
  - Note that we use at most 2 sets (buffers & textures)!
- Before each draw/dispatch we lazily allocate/update descriptor sets

Descriptor management: configuring pools

- A pool per shader pipeline object
  - We know the number of textures/buffers each pipeline uses, can configure pools optimally
  - E.g. shadow map opaque pipeline: 1024 sets, 0 textures, 2*1024 buffers
  - E.g. scene opaque pipeline: 1024 sets, 8*1024 textures, 3*1024 buffers
  - A lot of space wasted on rarely used pipelines (postfx), more expensive to switch pipelines
  - Very hard to manage across multiple threads
- One type of pool, configured using worst-case descriptor count
  - E.g. one VkDescriptorPool has 1024 sets, 16*1024 textures, 8*1024 buffers
  - Simple – just one type of pool!
  - A lot of space wasted because the ratio of sets:textures:buffers varies
- Settled on one type of pool, configured for “average” usecase
  - sets:textures:buffers ratios determined by collecting data on typical levels
  - Simple, little space wasted in common case
  - Non-trivial space savings – tens of megabytes on moderate levels

Descriptor management: allocate / update / bind

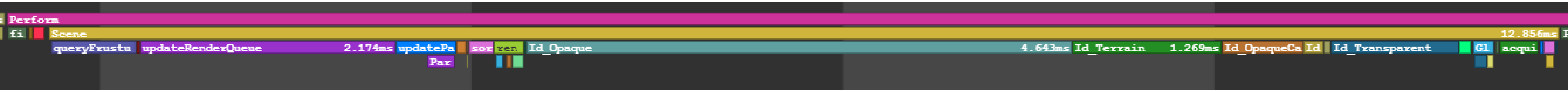
- Allocating the sets: a pool of pools
  - If the current pool has space, allocate a descriptor set in this pool (free-threaded)
  - Otherwise, get a pool out of the global “pool of pools” (requires a lock)
  - Recycle pools at the end of the frame (never free descriptor sets, vkResetDescriptorPool instead)
- Lazy update / bind
  - Only update the set with changes (e.g. texture-only changes only need to update one set)
  - Often don’t need to rebind sets when pipeline changes (~50% fewer buffer descriptor updates)
  - Do not use descriptor set copying for partial updates!
  - Descriptor templates from Vulkan 1.1 reduce CPU cost further
- Constant data update
  - Most of our per-frame constant data is small and dynamic
  - We sub-allocate it from a large buffer in bindBufferData()
  - Instead of allocating a new buffer descriptor every time, use pDynamicOffsets

General optimizations

- The driver is much slimmer than a typical GL driver
  - This surfaces things that were trivial/unnoticeable before!
- Don’t call vk* functions unless you need to
  - Cache everything that’s easy to cache, filter redundant state bindings
- Aggressively eliminate cache misses
  - Reduce allocations and indirections in your abstraction
  - E.g. we use GeometryVulkan that is similar to OpenGL VAO – struct with all geometry state
- Call most functions through pointers obtained via vkGetDeviceProcAddr
  - volk (github.com/zeux/volk) loader does this for us; a few % wins on some drivers

Results

- Seeing 2x-3x CPU performance gains across all vendors vs GLES
  - End-to-end render frame, real-world contents
- Mobile test level @ 840 draw calls, single core
  - 2.4 GHz Cortex-A73, Mali-G72
  - GLES: 38 ms 😱
  - Vulkan single-core: 13 ms
- Good multi-core scaling as well!
  - Beware big vs LITTLE



References

Further reading

- *Synchronization Examples* by Tobias Hector
- *Yet another blog explaining Vulkan synchronization* by Hans-Kristian Arntzen
- *GPU Framebuffer Memory: Understanding Tiling*, Samsung GameDev
- *Writing an efficient Vulkan renderer*, GPU Zen 2, by Arseny Kapoulkine
- *Vulkan Guide*, Khronos Group

Vulkan samples

https://github.com/KhronosGroup/Vulkan-Samples

Sascha Willems

- API examples
  - Compute shader N-body simulation
  - Dynamic uniform buffers
  - High Dynamic Range rendering
  - Instanced mesh rendering
  - Dynamic terrain tessellation
  - Texture loading and display
  - Runtime mipmap generation
- Extension samples
  - `VK_EXT_conservative_rasterization`
  - `VK_KHR_push_descriptor`
  - `VK_NV_ray_tracing`

Arm

- Performance samples with tutorials
  - AFBC
  - Command buffer management
  - Constant data
  - Descriptor and buffer management
  - Impact of vkDeviceWaitIdle()
  - Layout transitions
  - Load/store operations
  - MSAA
  - Multi-threaded command buffer recording
  - N-buffering and presentation modes
  - Pipeline barriers
  - Pipeline cache
  - Pre-rotation
  - Specialization constants
  - Subpass merging and G-buffer size

Thank You
Danke
Merci
谢谢
ありがとう
Gracias
Kiitos
감사합니다
धन्यवाद
شكرًا
ধন্যবাদ
תודה