Vulkan: the essentials

Tristan Lorach, March 17th 2016

NVIDIA

Analogy On Graphic APIs



Analogy

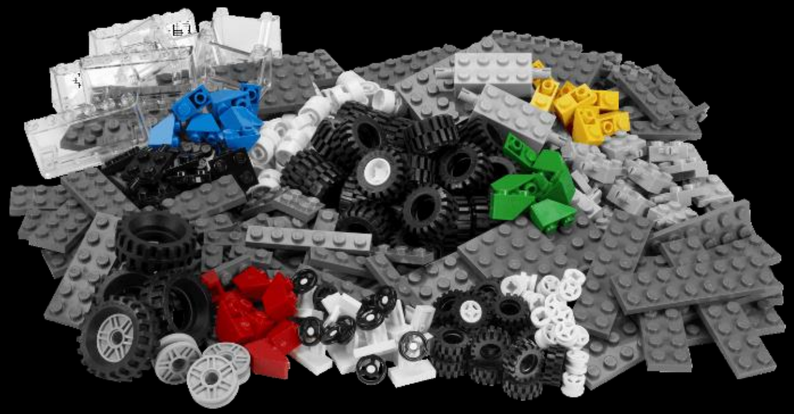
Fixed-function OpenGL

Pre-assembled toy car

fun out of the box,
not much room for customization

Analogy

Modern AZDO OpenGL with Programmable Shaders



LEGO Kit

you build it yourself,
comes with plenty of useful, pre-shaped pieces

Analogy

Vulkan

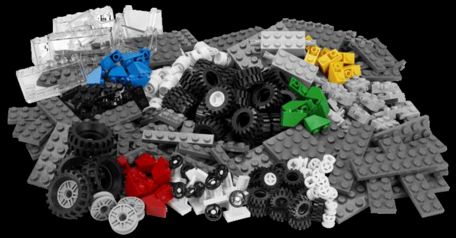
Pine Wood Derby Kit

you build it yourself to race from raw materials

power tools used to assemble, adult supervision highly recommended

Analogy

Different Valid Approaches

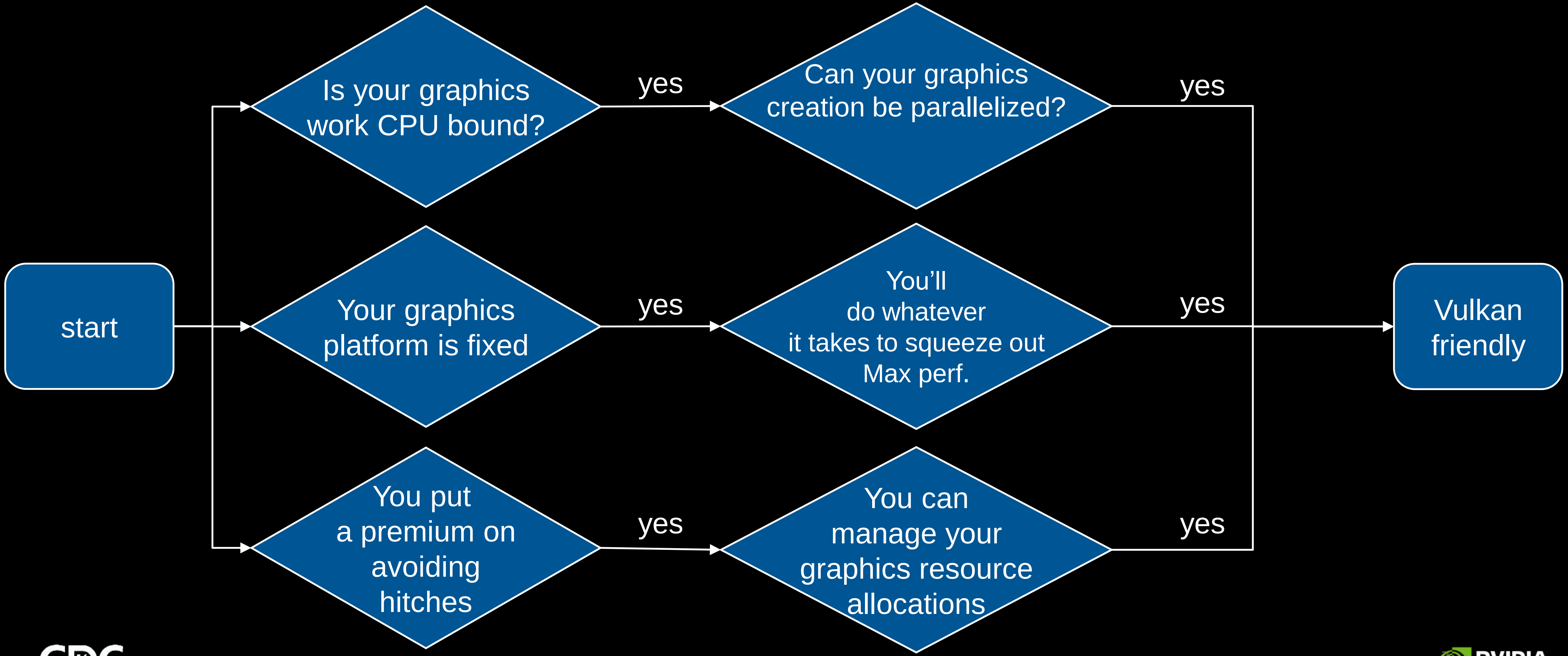


Fixed-function OpenGL

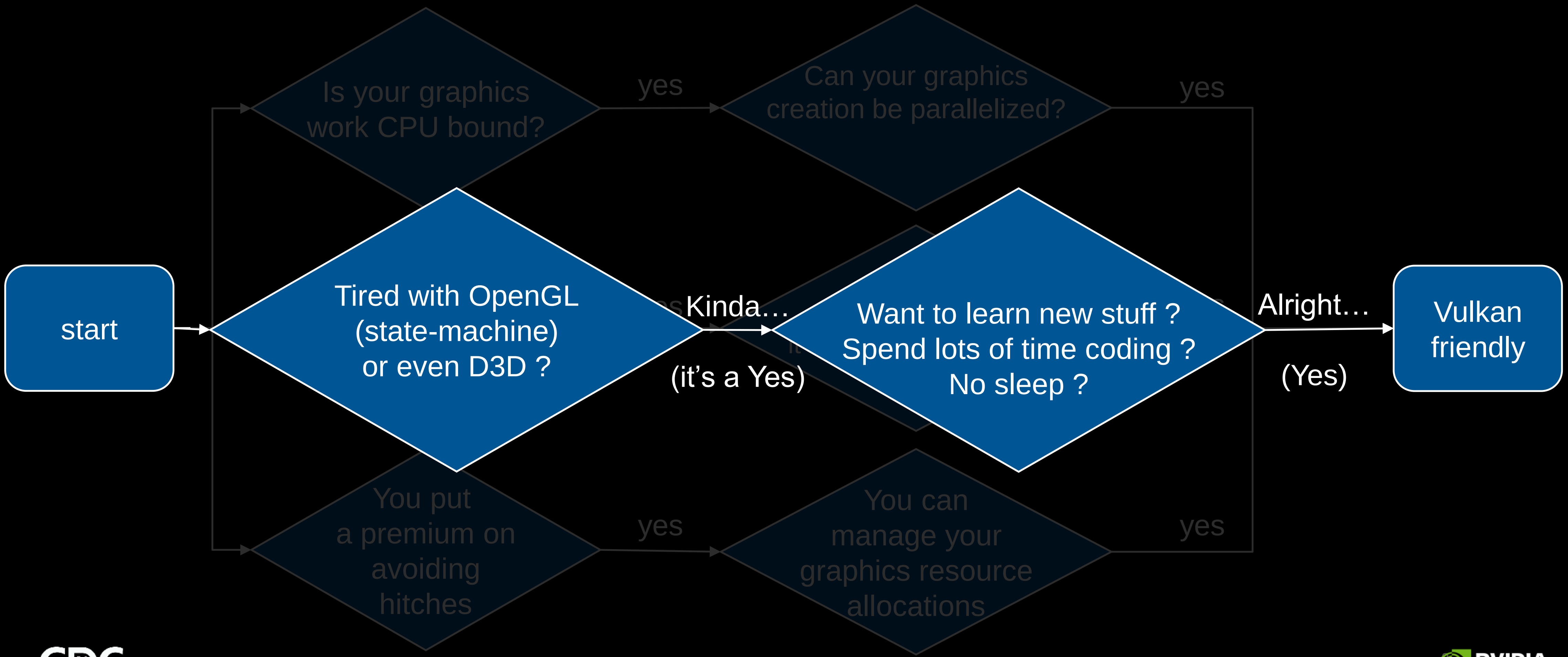
Modern AZDO OpenGL with Programmable Shaders

Vulkan

Beneficial Vulkan Scenarios



Beneficial Vulkan Scenarios



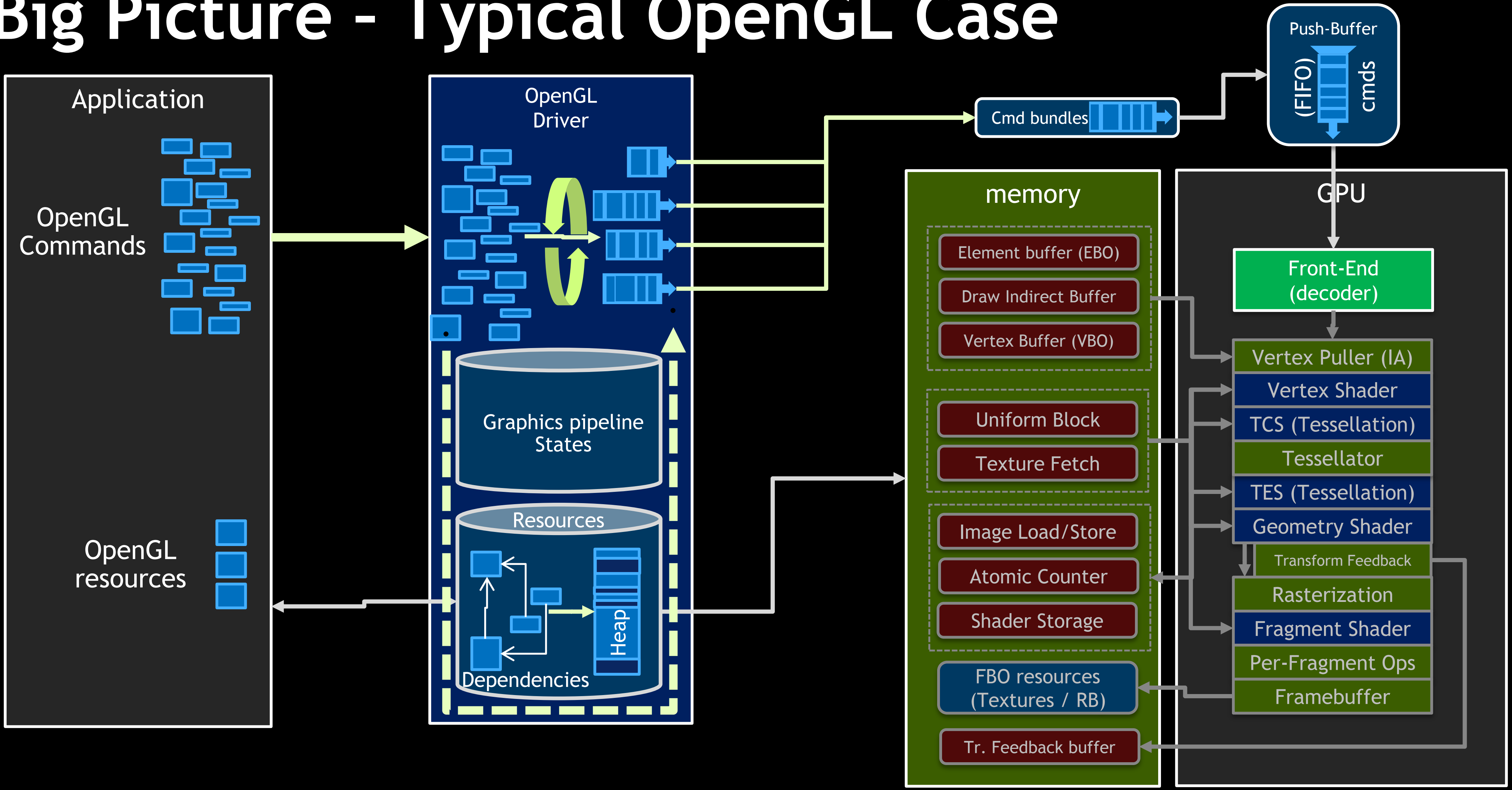
Unlikely to Benefit

Scenarios to Reconsider Coding to Vulkan

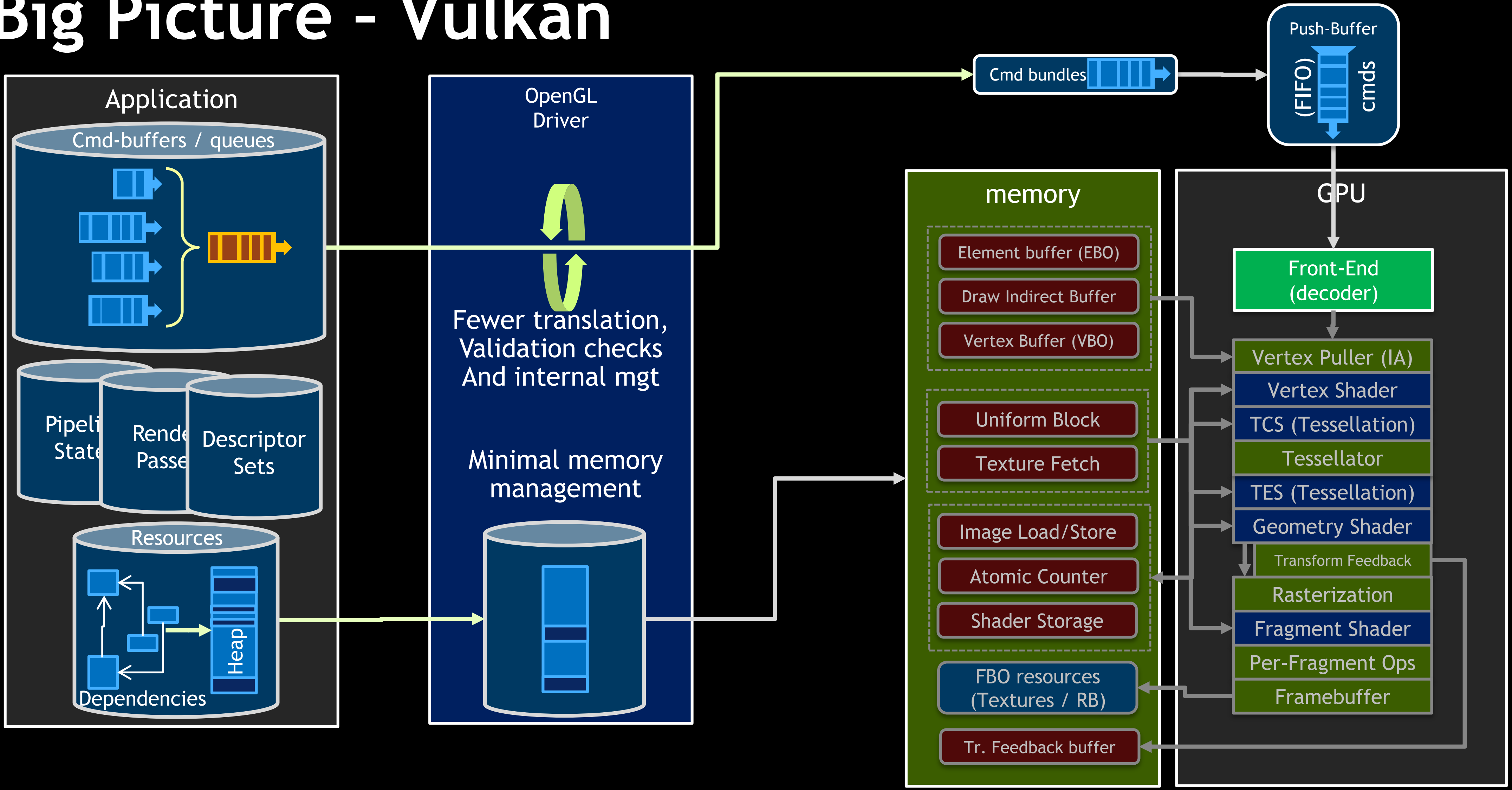
1. Need for compatibility to pre-Vulkan platforms
2. Heavily GPU-bound application
3. Heavily CPU-bound application due to non-graphics work
4. Single-threaded application, unlikely to change
5. App can target middle-ware engine, avoiding 3D graphics API dependencies
   - Consider using an engine targeting Vulkan, instead of dealing with Vulkan yourself

OpenGL / D3D

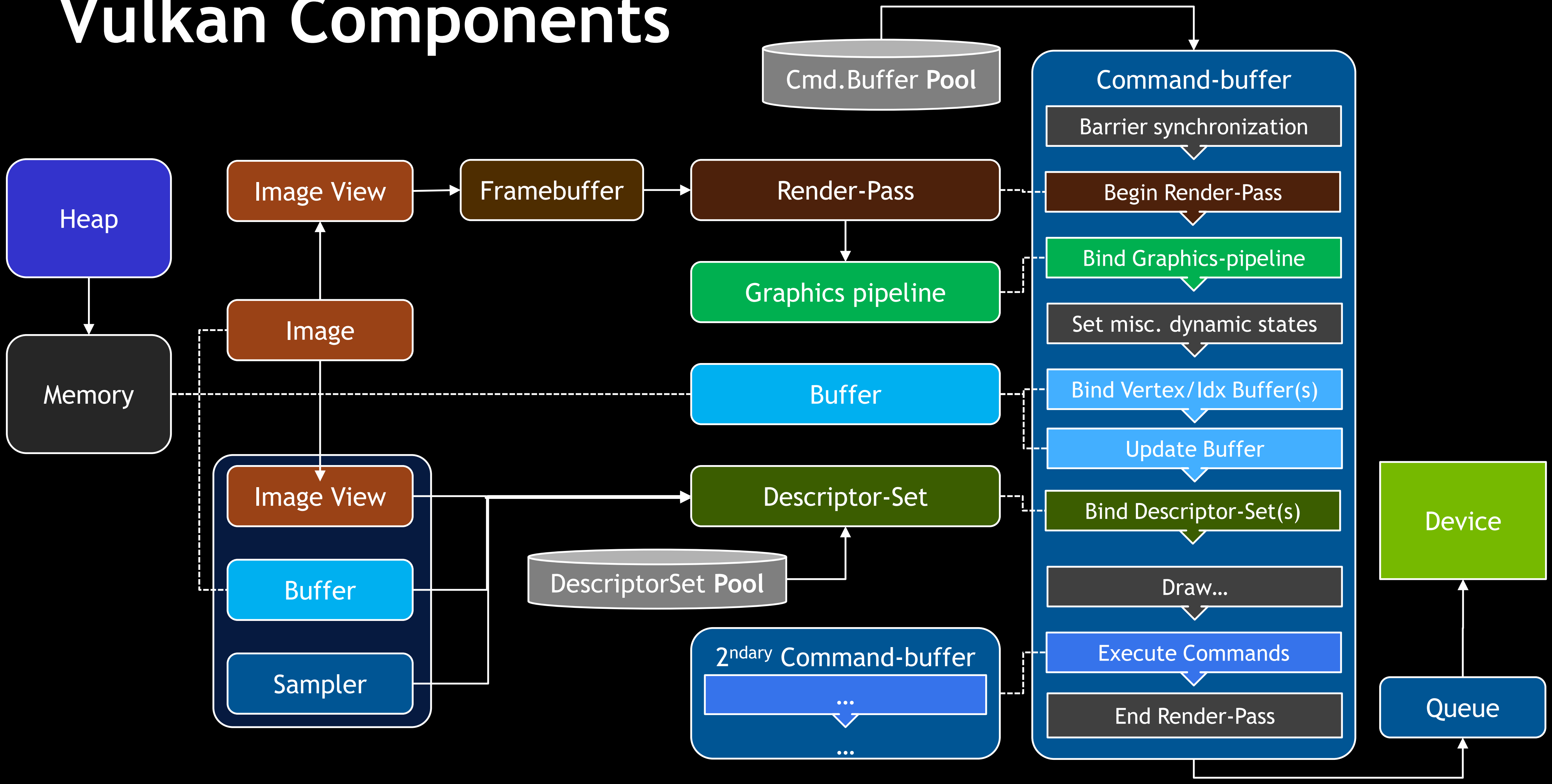
Big Picture - Typical OpenGL Case



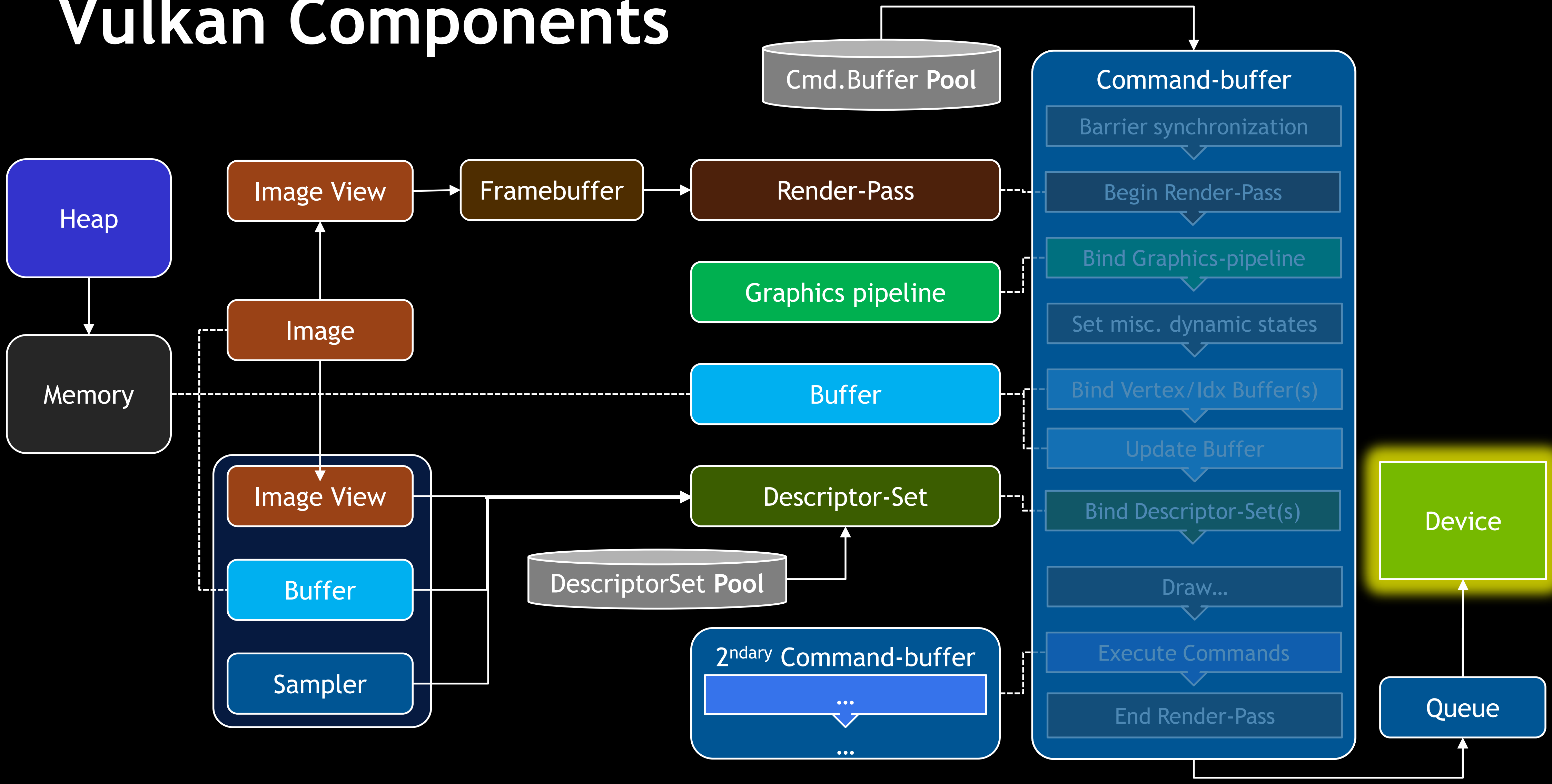
Big Picture - Vulkan



Vulkan Components



Vulkan Components



Vulkan Objects: Device

VkPhysicalDevice

- Capabilities
- Memory Management
- Queues
- Objects
  - Buffers
  - Images
  - Sync Primitives

Can have many …

Device

NVIDIA's Vulkan Capabilities

- Properties listed from Physical Device
- NVIDIA is almost full featured
  - Top to bottom: from GeForce, Quadro down to Tegra
- Check http://vulkan.gpuinfo.org/listreports.php

NVIDIA's Vulkan Capabilities

GeForce GTX 980

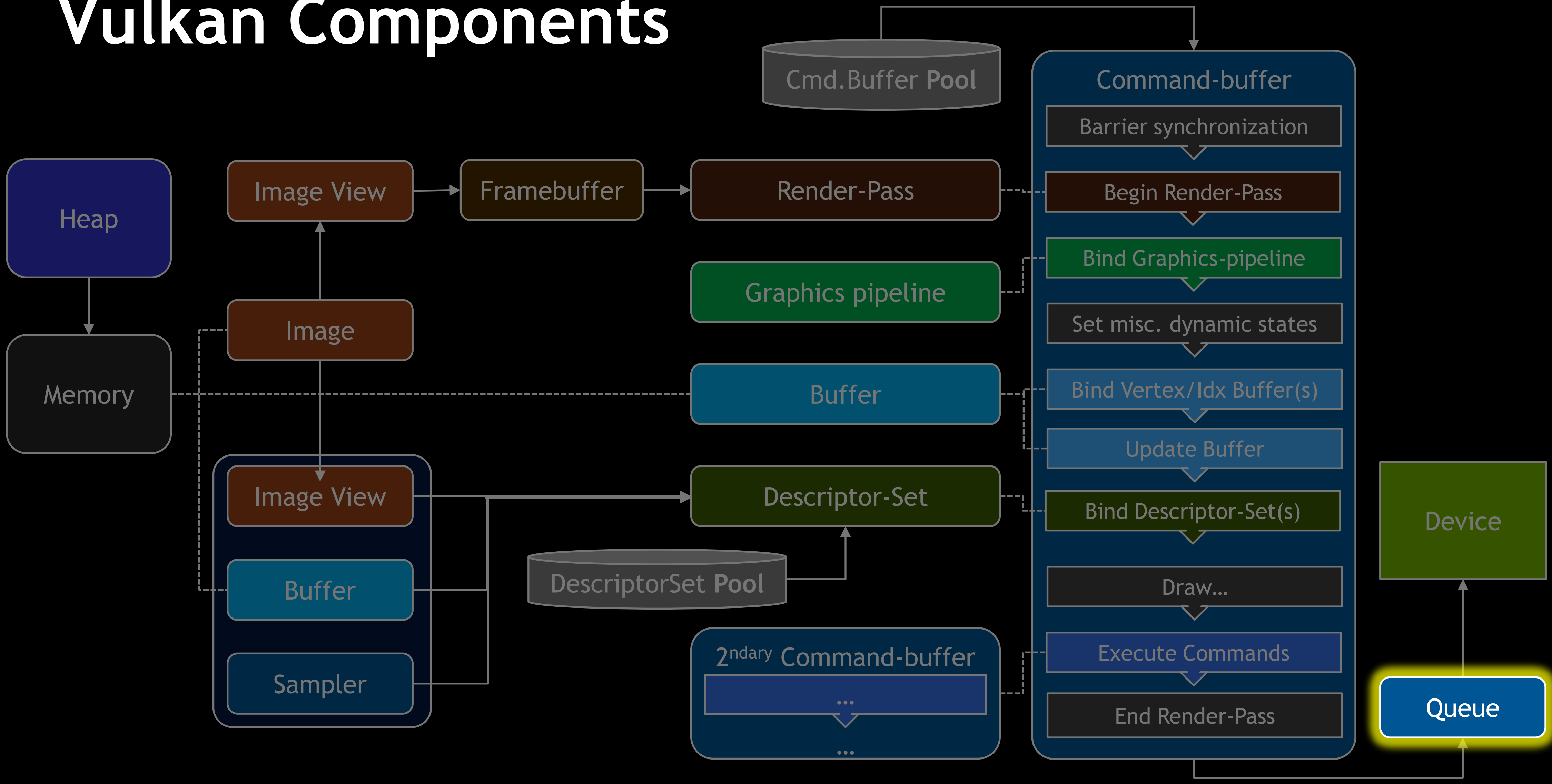
Feature	Value
alphaToOne	true
depthBiasClamp	true
depthBounds	true
depthClamp	true
drawIndirectFirstInstance	true
dualSrcBlend	true
fillModeNonSolid	true
fragmentStoresAndAtomics	true
fullDrawIndexUint32	true
geometryShader	true
imageCubeArray	true
independentBlend	true
inheritedQueries	true
largePoints	true
logicOp	true
multiDrawIndirect	true
multiViewport	true
occlusionQueryPrecise	true
pipelineStatisticsQuery	true
robustBufferAccess	true
sampleRateShading	true
samplerAnisotropy	true
shaderClipDistance	true
shaderCullDistance	true
shaderFloat64	true
shaderImageGatherExtended	true
shaderInt16	false
shaderInt64	true

shaderResourceMinLod	true
shaderResourceResidency	true
shader	
shader	
shader	
shader	
shader	
shader	
shader	
shader	
shader	
sparse	
sparse	
sparse	
sparse	
sparse	
sparse	
sparse	
sparse	
sparse	
tessella	
texture	
texture	
texture	
variabl	
vertexP	
wideLir	

Tegra X1 & K1

Feature	Report 3	Report 78
device	NVIDIA NVIDIA Tegra X1	NVIDIA NVIDIA Tegra K1
version	361.0.0 (1.0.2)	361.0.0 (1.0.2)
os	android 6.0 (arm)	android 6.0.1 (arm)
alphaToOne	true	true
depthBiasClamp	true	true
depthBounds	true	true
depthClamp	true	true
drawIndirectFirstInstance	true	true
dualSrcBlend	true	true
fillModeNonSolid	true	true
fragmentStoresAndAtomics	true	true
fullDrawIndexUint32	true	true
geometryShader	true	true
imageCubeArray	true	true
independentBlend	true	true
inheritedQueries	true	true
largePoints	true	true
logicOp	true	true
multiDrawIndirect	true	true
multiViewport	true	true
occlusionQueryPrecise	true	true
pipelineStatisticsQuery	true	true
robustBufferAccess	true	true
sampleRateShading	true	true
samplerAnisotropy	true	true
shaderClipDistance	true	true
shaderCullDistance	true	true
shaderFloat64	true	true
shaderImageGatherExtended	true	true
shaderInt16	false	false
shaderInt64	true	true
shaderResourceMinLod	true	false
shaderResourceResidency	true	false
shaderSampledImageArrayDynamicIndexing	true	true
shaderStorageBufferArrayDynamicIndexing	true	true
shaderStorageImageArrayDynamicIndexing	true	true
shaderStorageImageExtendedFormats	true	true
shaderStorageImageMultisample	true	true
shaderStorageImageReadWithoutFormat	true	false
shaderStorageImageWriteWithoutFormat	true	true
shaderTessellationAndGeometryPointSize	true	true
shaderUniformBufferArrayDynamicIndexing	true	true
sparseBinding	true	true
sparseResidency16Samples	true	false
sparseResidency2Samples	true	true
sparseResidency4Samples	true	true
sparseResidency8Samples	true	true
sparseResidencyAliased	true	true
sparseResidencyBuffer	true	true
sparseResidencyImage2D	true	true
sparseResidencyImage3D	true	true
tessellationShader	true	true
textureCompressionASTC_LDR	true	true
textureCompressionBC	true	true
textureCompressionETC2	true	true
variableMultisampleRate	true	true
vertexPipelineStoresAndAtomics	true	true
wideLines	true	true

Vulkan Components

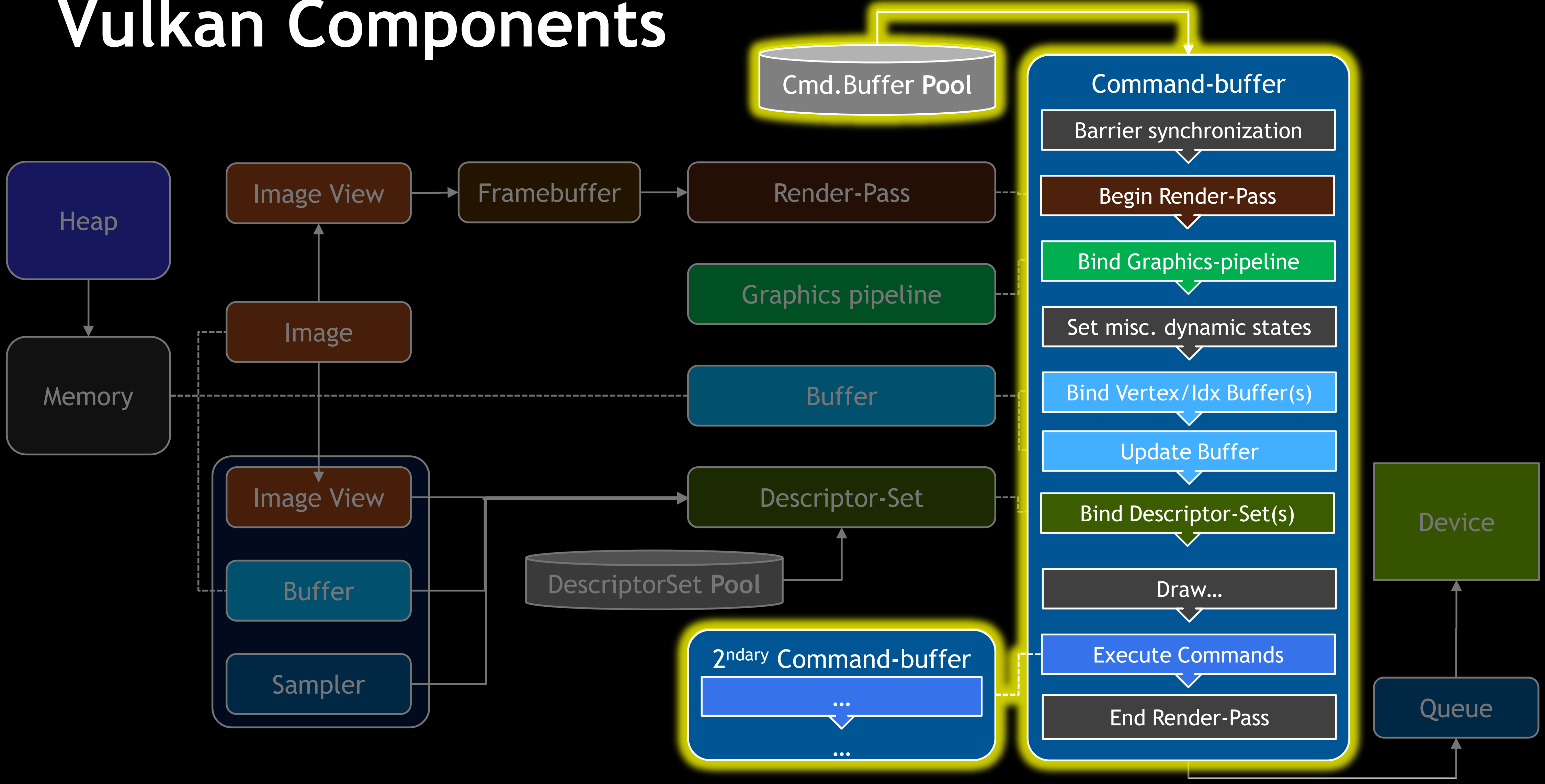


Queues

- Command queue was hidden in OpenGL Context… now explitly declared
  - Multiple threads can submit work to a queue (or queues)!
- Queues accept GPU work via CommandBuffer submissions
  - few operations available:, “submit work” and “wait for idle”
- Queue submissions can include sync primitives for the queue to:
  - ***Wait*** upon before processing the submitted work
  - ***Signal*** when the work in this submission is completed
- Queue “families” can accept different types of work, e.g.
- NVIDIA exposes 16 Queues
  - Only one type of queue for all the types of work

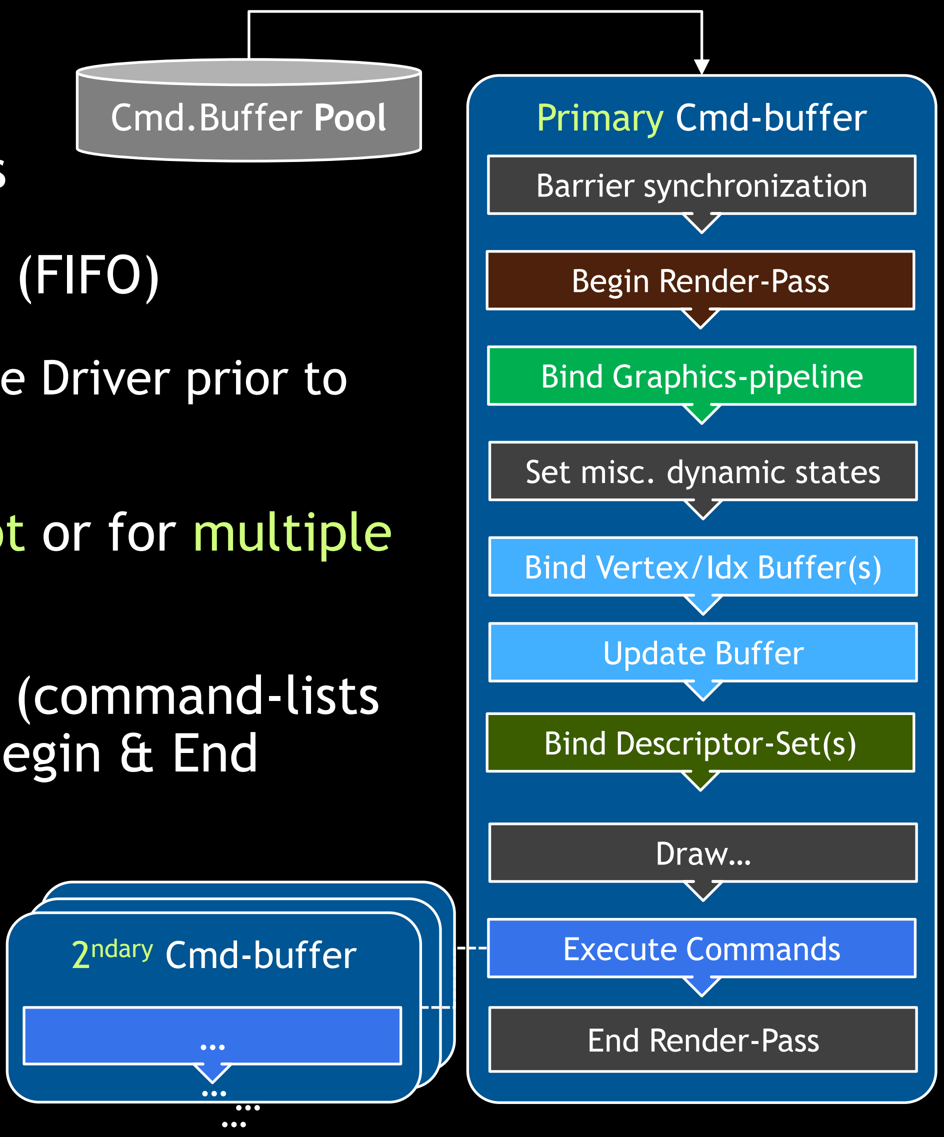
Queue

Vulkan Components



Command-Buffers

- Vulkan Rendering ➔ Command-Buffers
- Almost what GPU will get at Front-End (FIFO)
  - Minor translation & optimization from the Driver prior to sending to the GPU
- Each can be created either for one shot or for multiple frames/submissions
- Cannot create Graphic Work from GPU (command-lists can): API calls to vkCmd…() between Begin & End
- Multi-threading friendly !
- Primary Cmd-Buffer can call many 2ndary Cmd-Buffers



Command-Buffers: Update/Push Constants

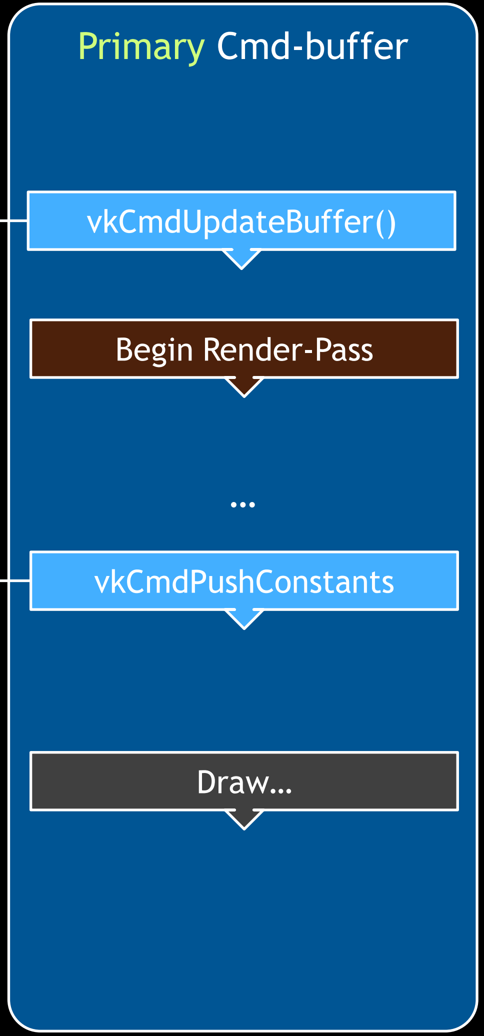
- 2 more ways to update constants/uniforms for Shaders from the Command-Buffer
  - Update-Buffer: prior to Render-Pass: can target any Buffer bound by Descriptor Sets

```
layout(set=0 , binding = 2 ) uniform MyBuffer {
   mat4 mW;
…
```

  - Push-Constants: targets a dedicated section in GLSL/SpirV

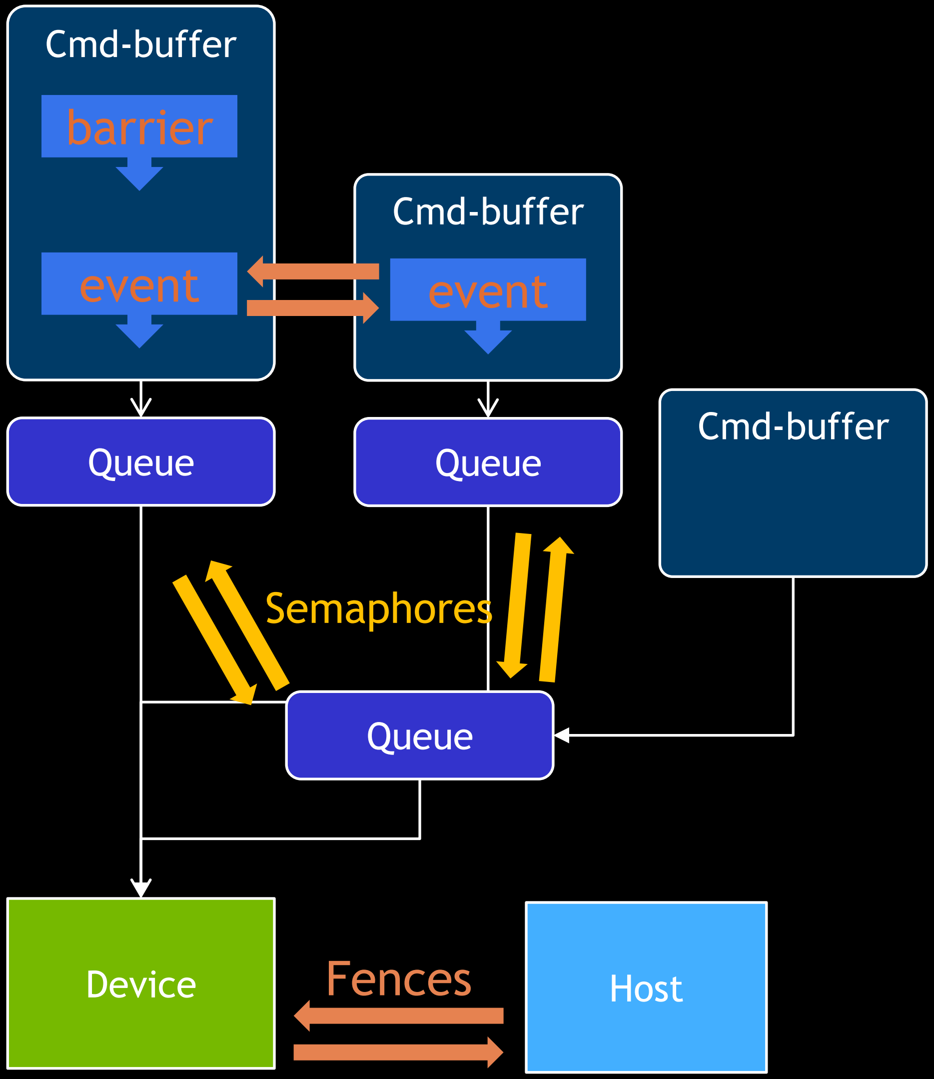
```
layout(push_constant) uniform objectBuffer {
   mat4 matrixObject;
   vec4 diffuse;
} object;
```

- New values appended “in-band”: in the Command-Buffer
- Efficient; but good for small amount of values

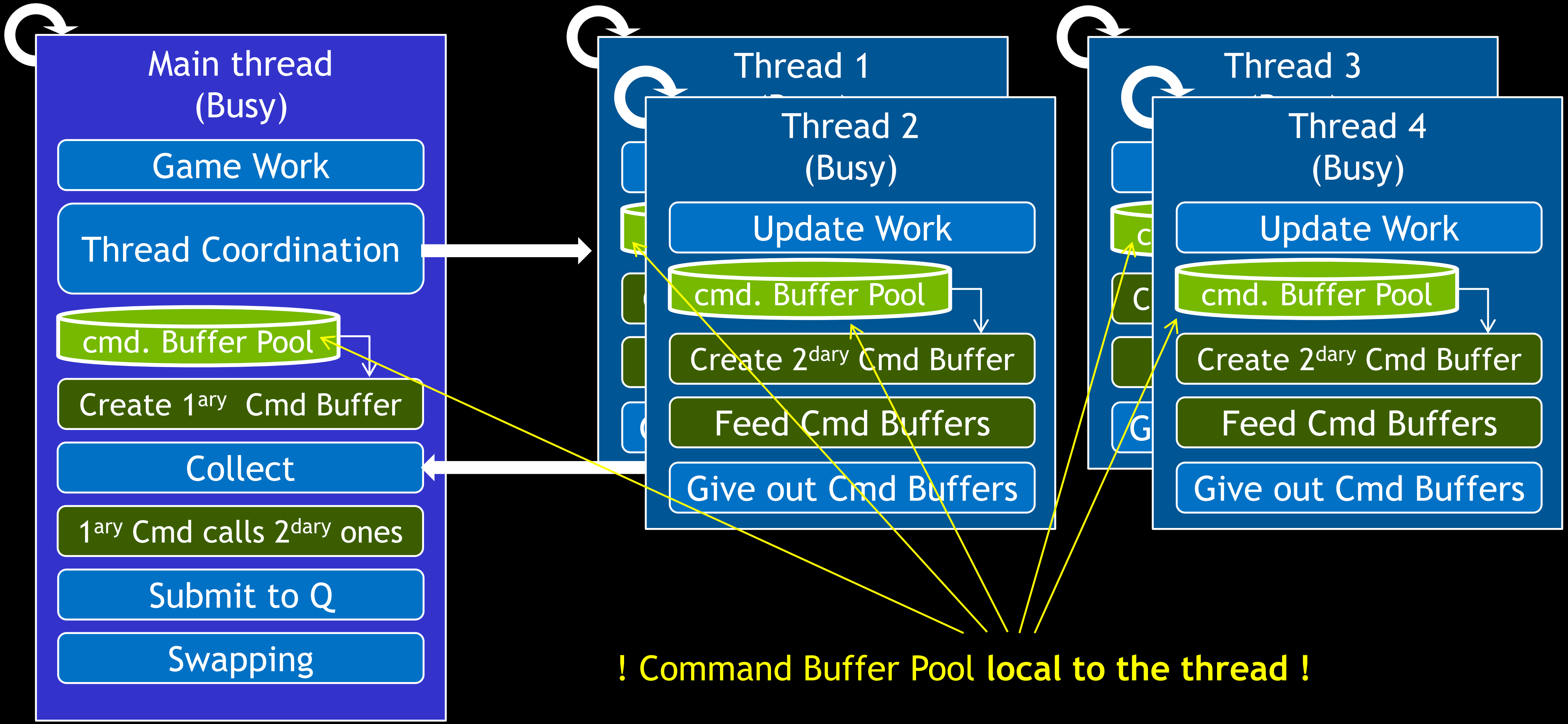


Synchronization

- semaphores
  - used to synchronize work across queues or across coarse-grained submissions to a single queue
- events and barriers
  - used to synchronize work within a command buffer or sequence of command buffers submitted to a single queue
- fences
  - used to synchronize work between the device and the host.



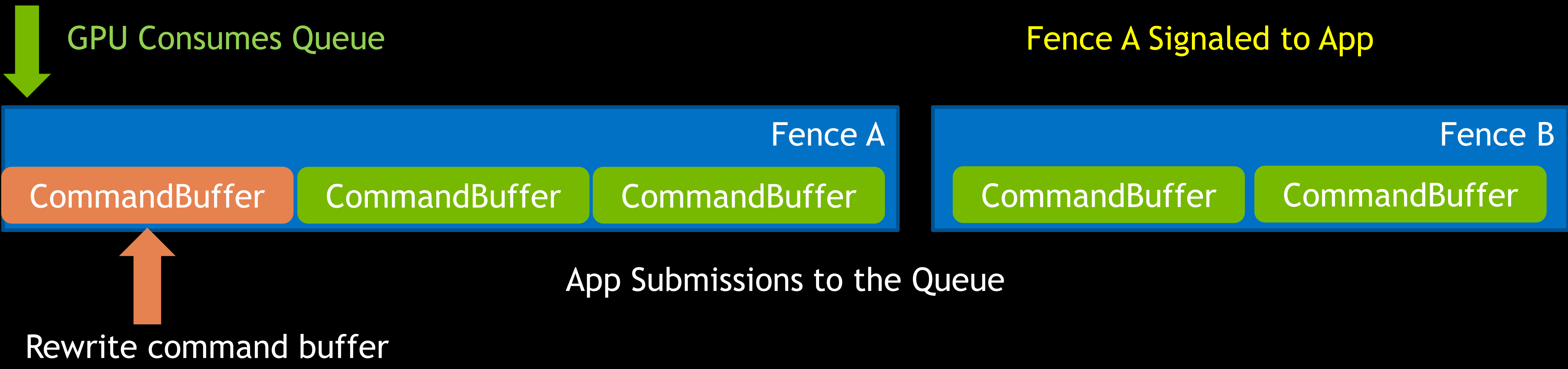
Command-Buffers and Multi-Threading



! Command Buffer Pool **local to the thread** !

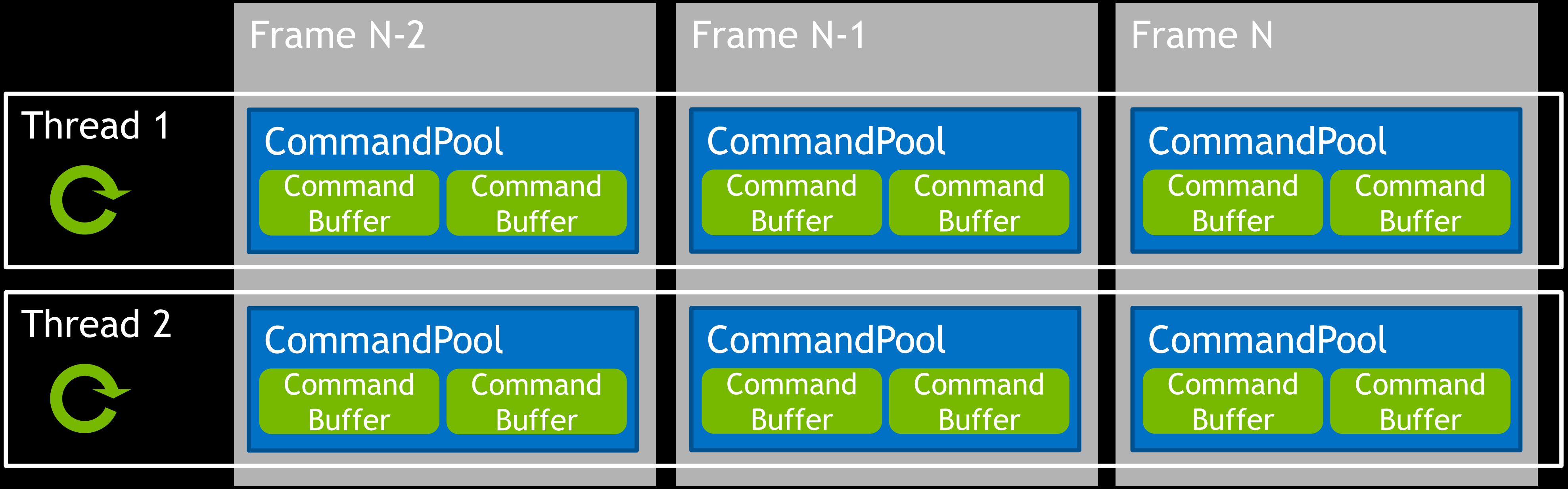
Command Buffer Thread Safety

- Must not recycle a CommandBuffer for rewriting until it is no longer in flight (In flight == GPU still consuming it on its side)
- But can't flush the queue each frame: would break parallelism !
- VkFences can be provided with a queue submission to test when a command buffer is ready to be recycled

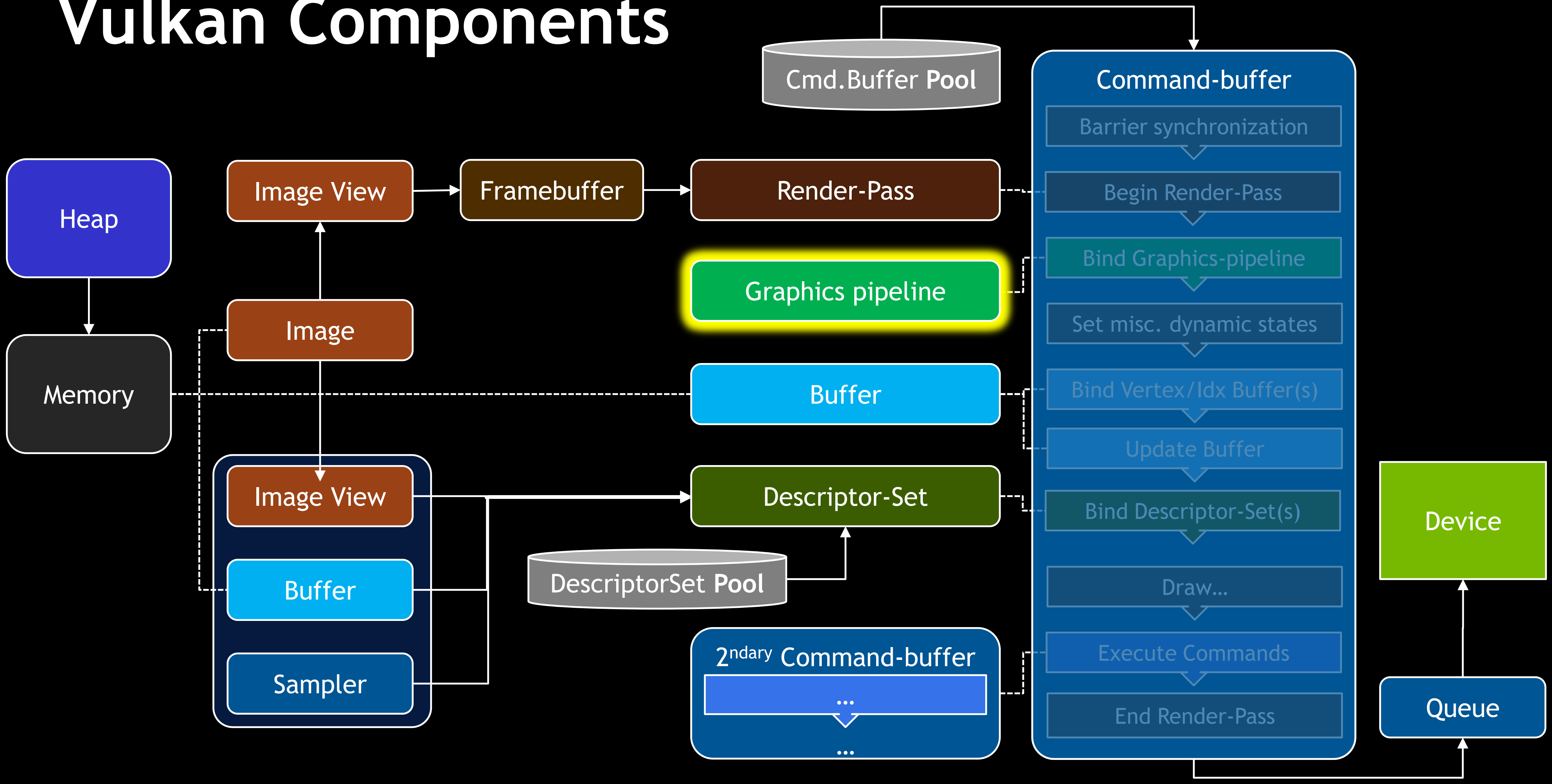


Threads And Command Pools

- Threads can have more than 1 Command Pool
  - Ring-buffer: One Command-Pool per Frame
- when that thread/frame is no longer in flight (Using Fences)
  - Faster to simply reset a pool

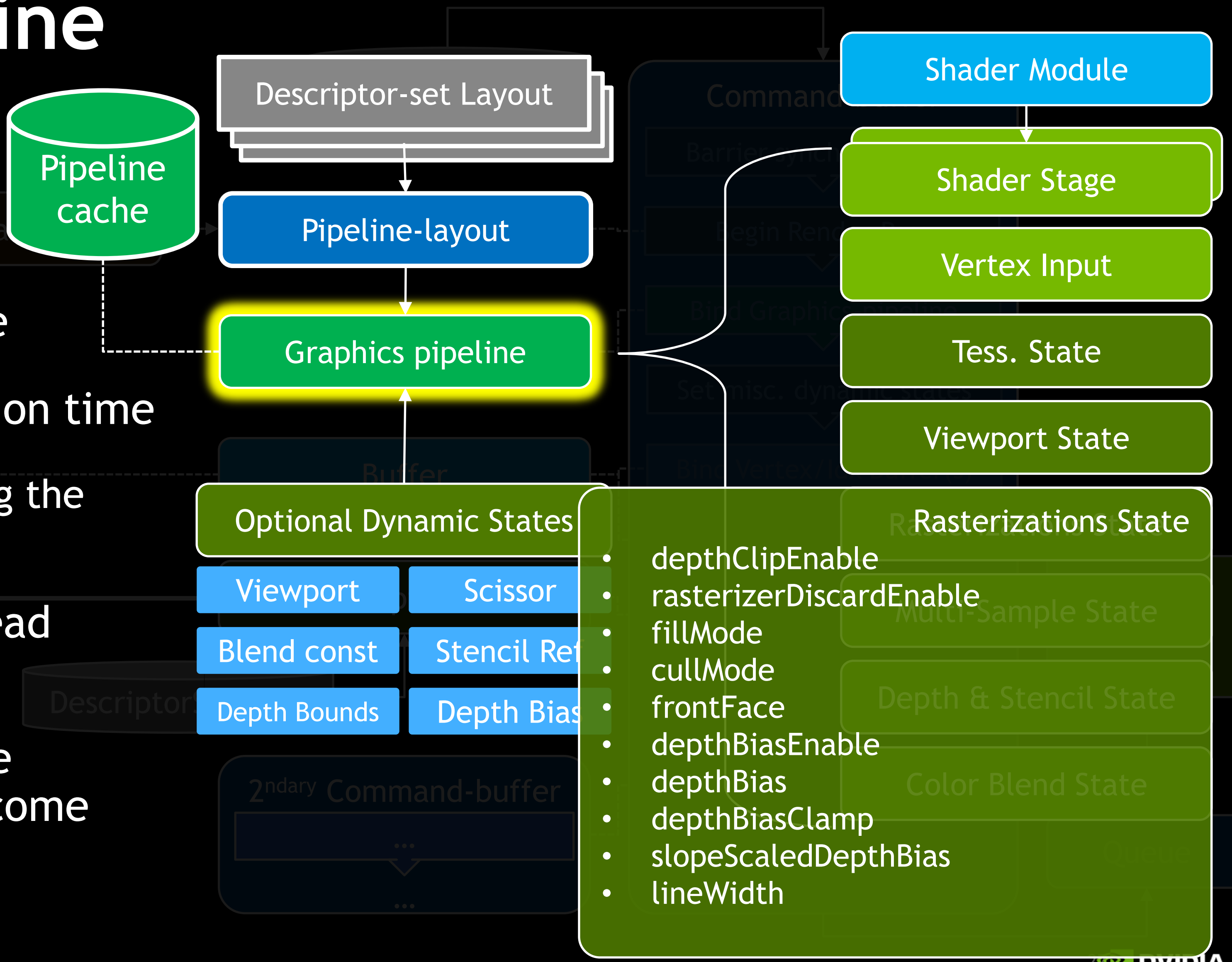


Vulkan Components



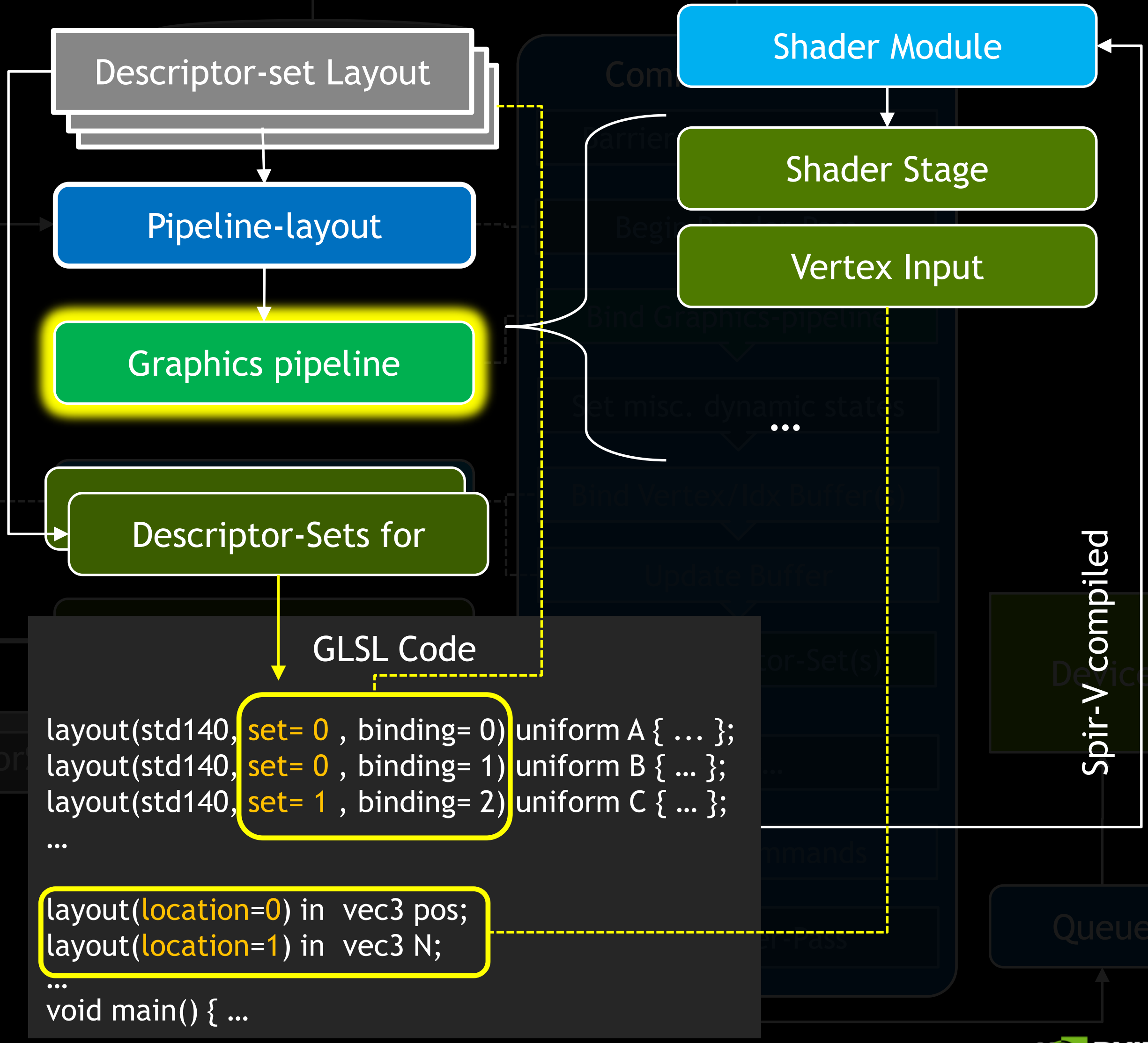
Graphics Pipeline

- Snapshot of all States
  - Including Shaders
- Pre-compiled & Immutable
- Ideally: done at Initialization time
  - Ok at render-time *if* using the Pipeline-Cache
- Prevents validation overhead during rendering loop
- Some Render-states can be excluded from it: they become “Dynamic” States

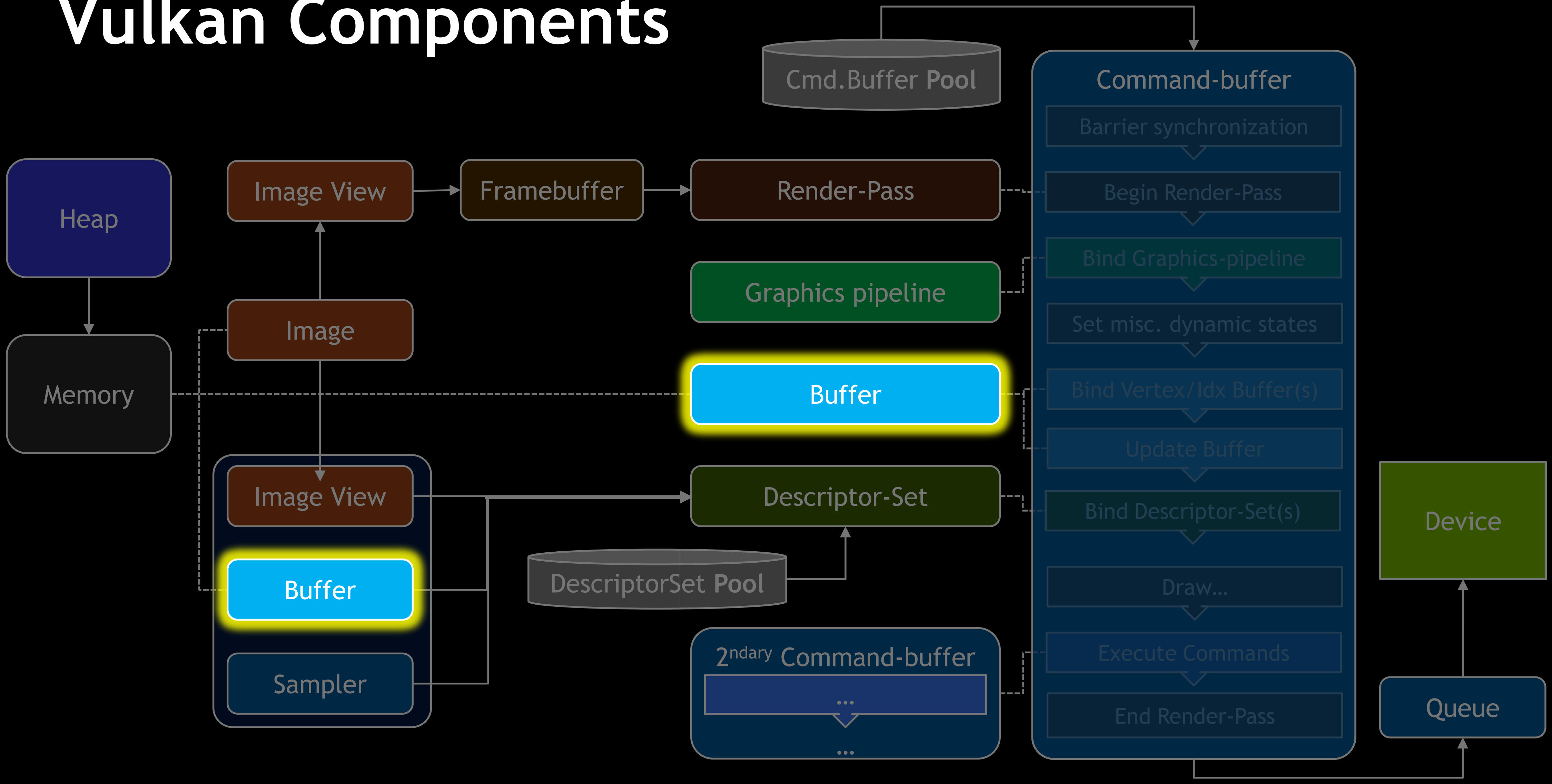


Graphics Pipeline

- Graphics Pipeline must be consistent with shaders
- No "introspection", so everything known & prepared in advance
- Vertex Input:
  - tells how Attributes: Locations are attached to which Vertex Buffer at which offset
- Pipeline Layout:
  - Tells how to map Sets and Bindings for the shaders at each stage (Vtx, Fragment, Geom…)



Vulkan Components

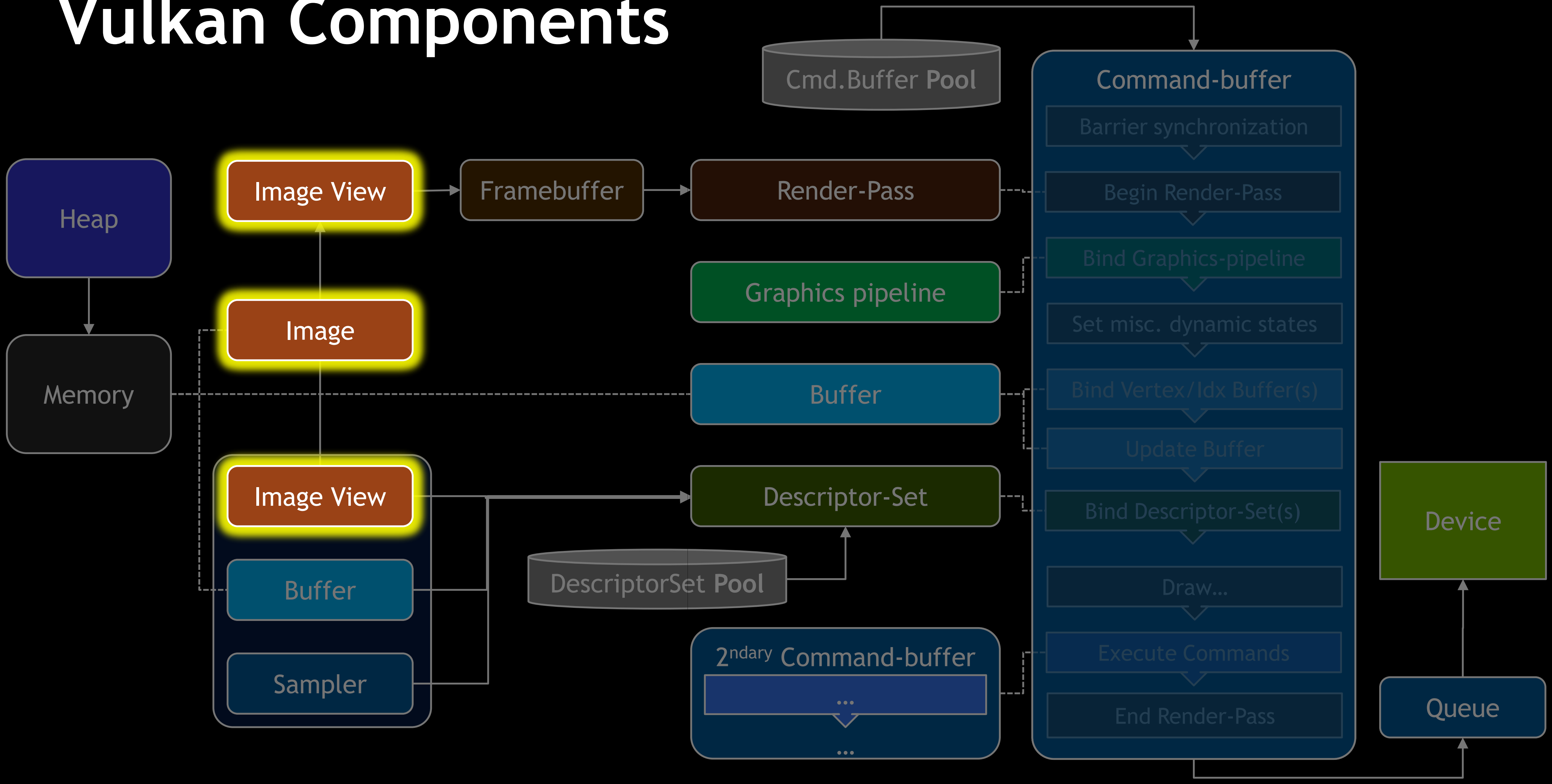


Buffers

- Highly Heterogenous. Most often used for:
  - Index/Vertex Buffers
  - Uniform Buffers (Matrices, material parameters…)
- Vulkan Object: Must be bound to some Device Memory
  - Can be CPU accessible memory (mappable)
  - Can be CPU cached
  - Can be GPU accessible only: need a “Staging Buffer” to write into it
    - But most Efficient

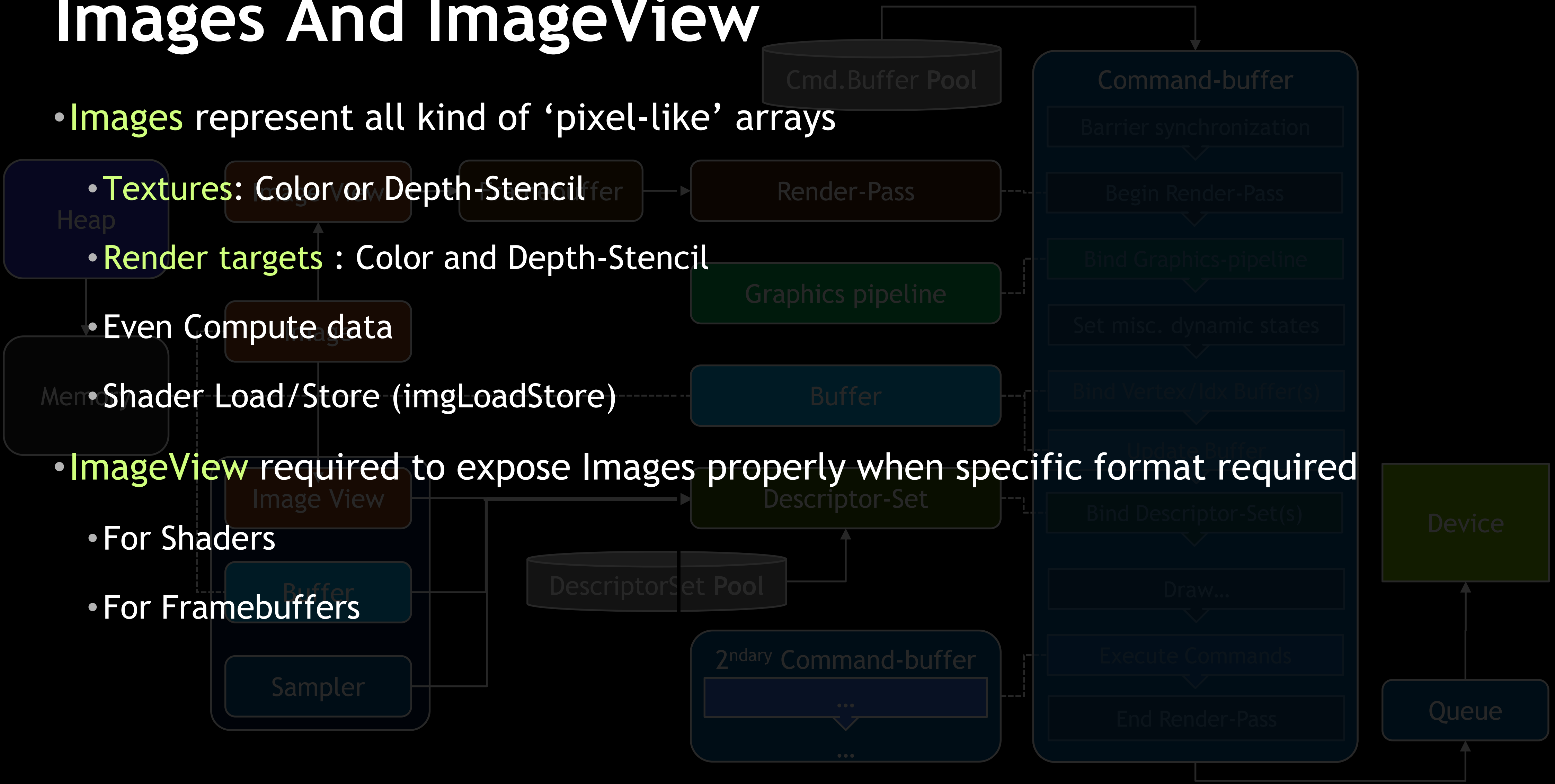
(More on Device Memory later…)

Vulkan Components

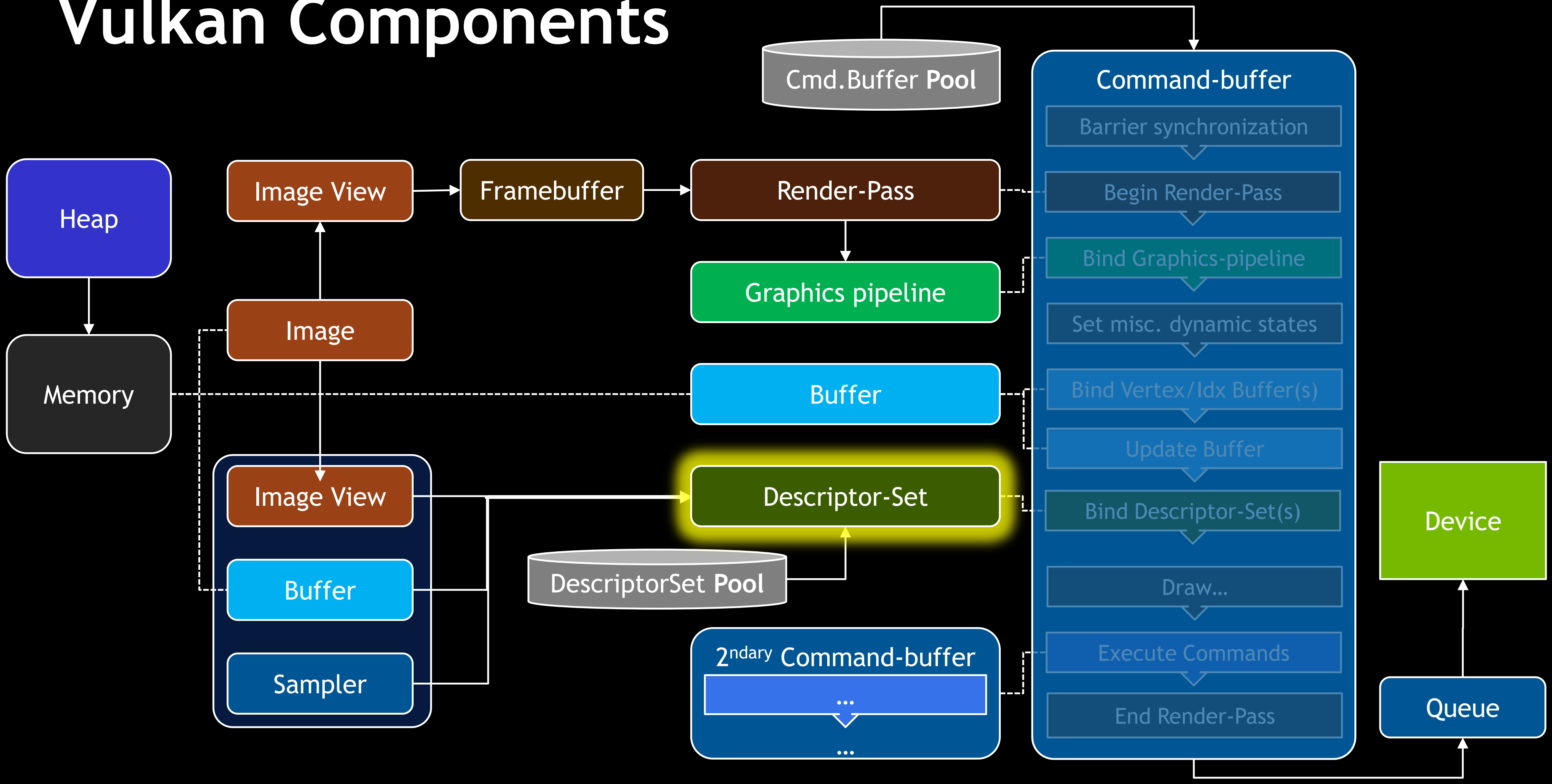


Images And ImageView

- Images represent all kind of 'pixel-like' arrays
  - Textures: Color or Depth-Stencil
  - Render targets : Color and Depth-Stencil
  - Even Compute data
  - Shader Load/Store (imgLoadStore)
- ImageView required to expose Images properly when specific format required
  - For Shaders
  - For Framebuffers

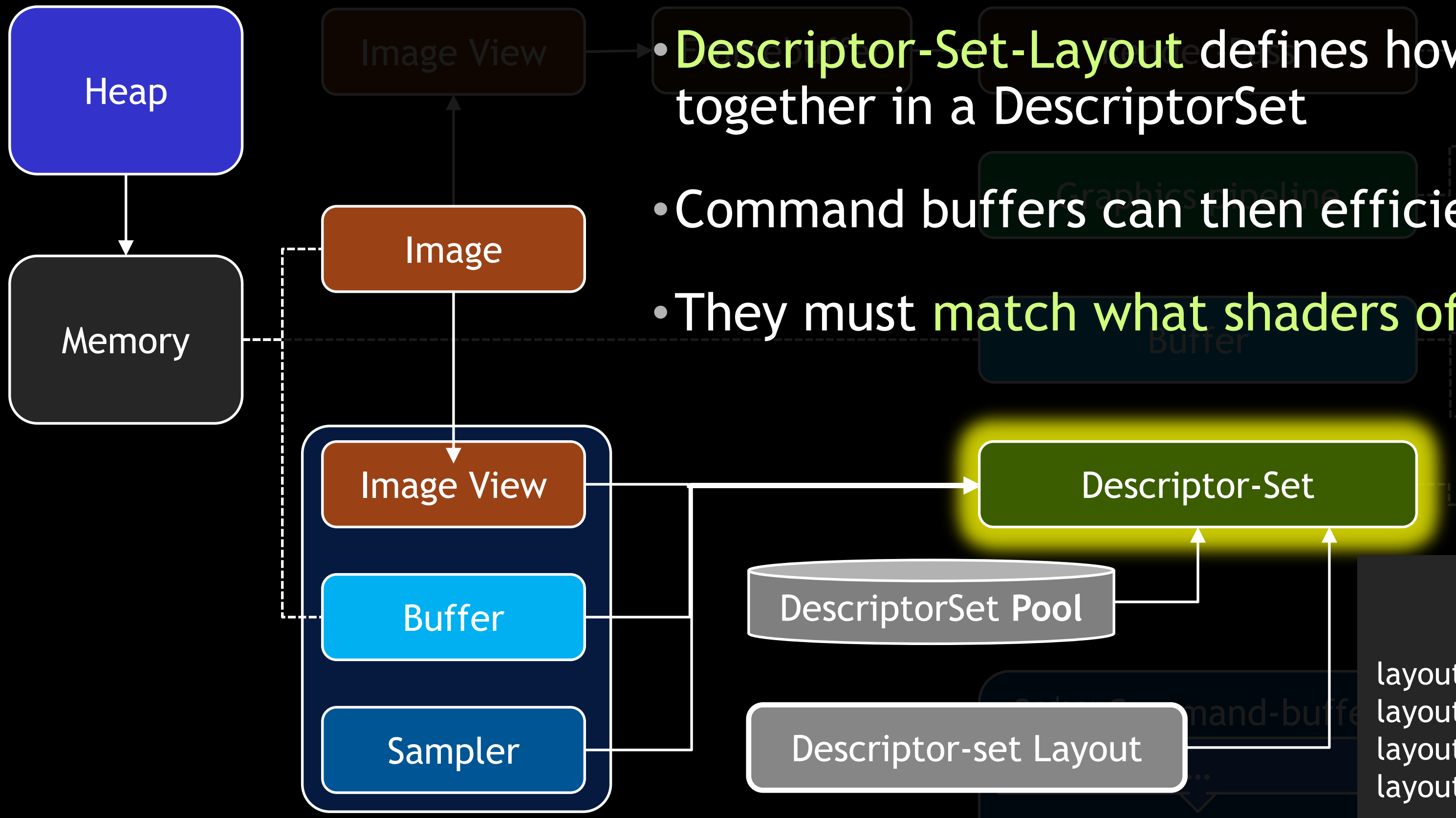


Vulkan Components



Descriptor-Set

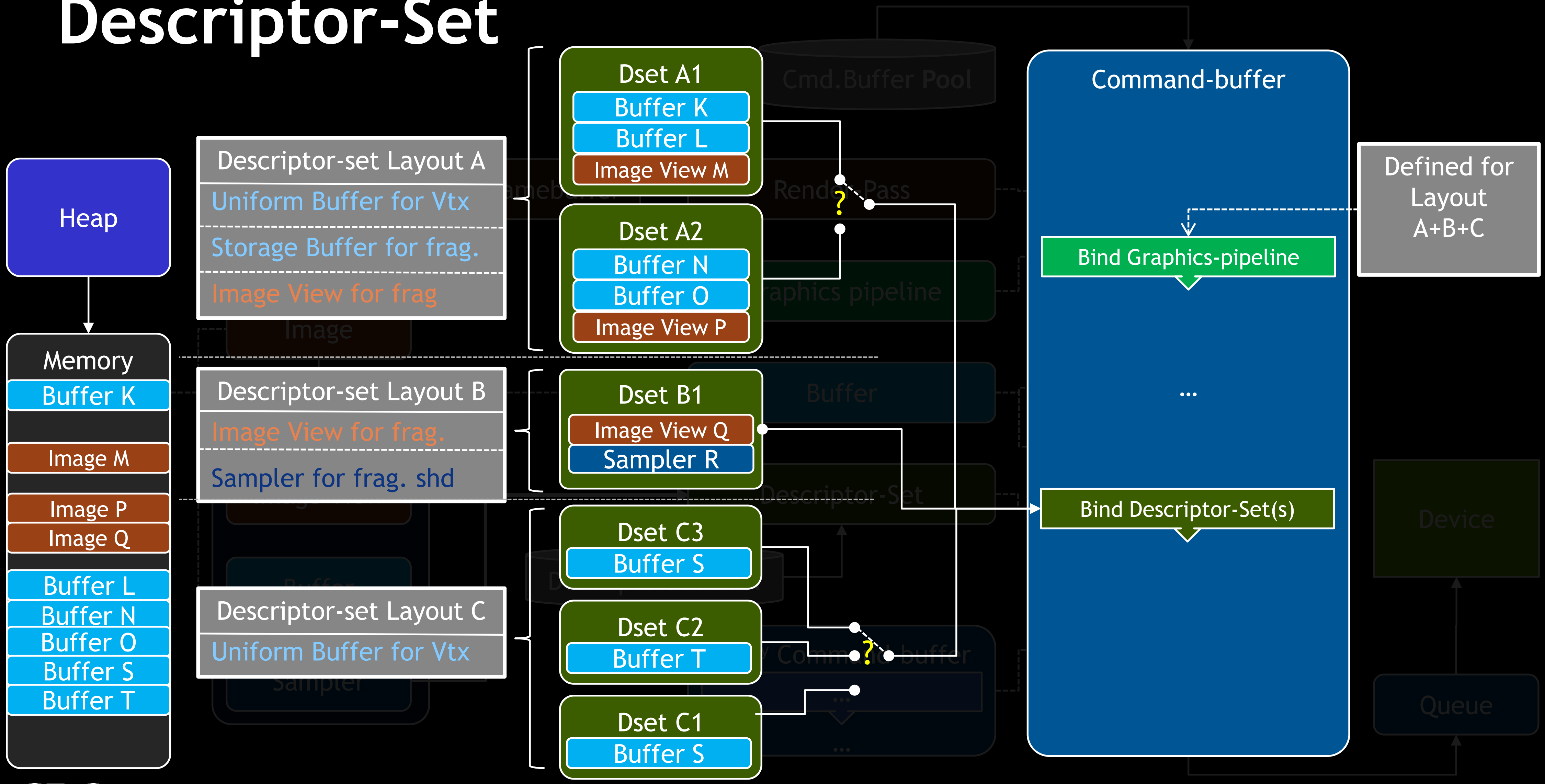
- Each DescriptorSet holds **references** to some resources
- **Descriptor-Set-Layout** defines how resources must be put together in a DescriptorSet
- Command buffers can then efficiently bind any or them
- They must **match what shaders of each stage expect** !



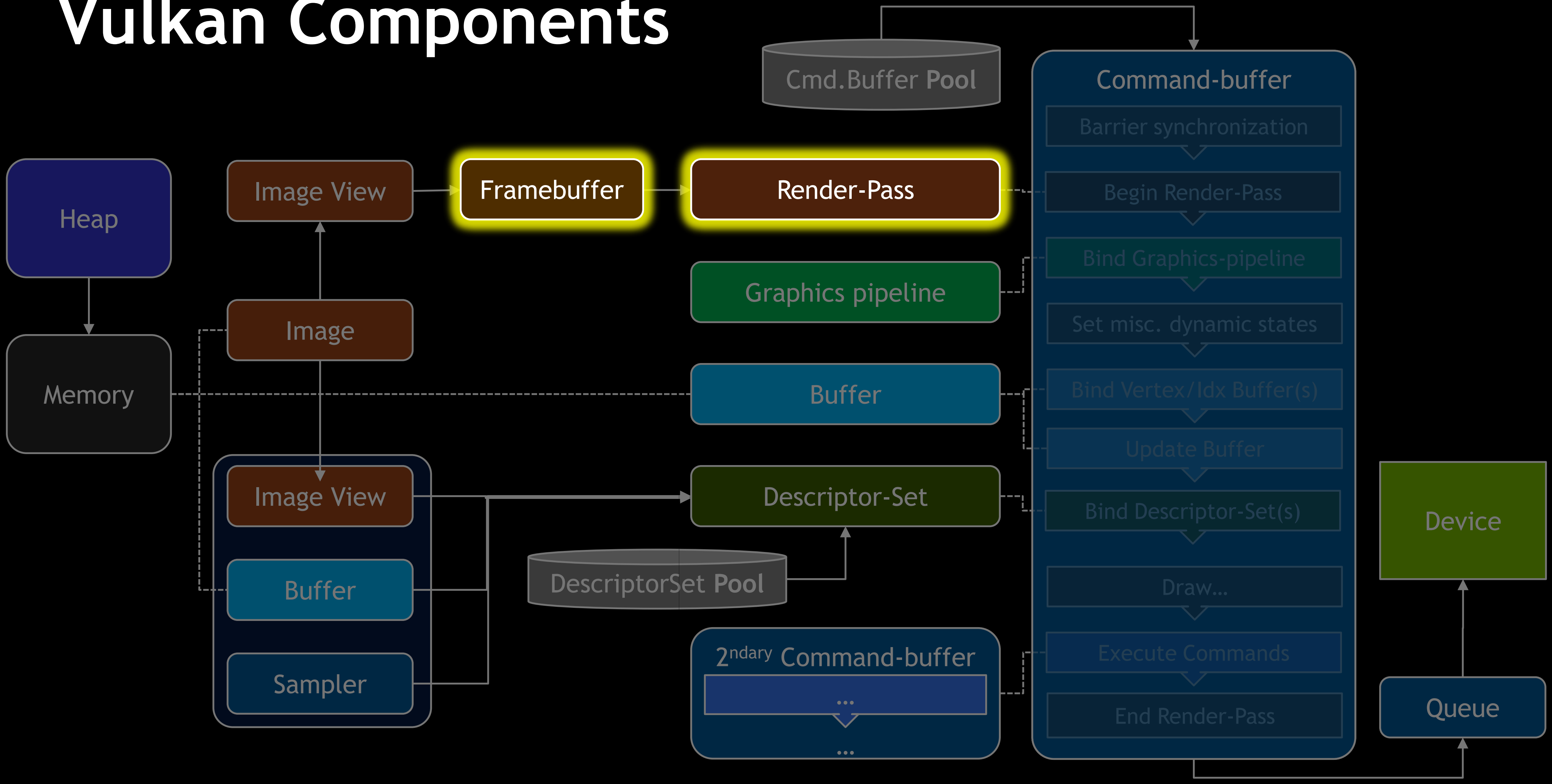
GLSL Code

```
layout(std140, set= 0 , binding= 0) uniform A { … };
layout(std140, set= 0 , binding= 1) uniform B { … };
layout(std140, set= 1 , binding= 2) uniform C { … };
layout(set=0, binding=3) uniform sampler2D tex;
…
void main() { …
```

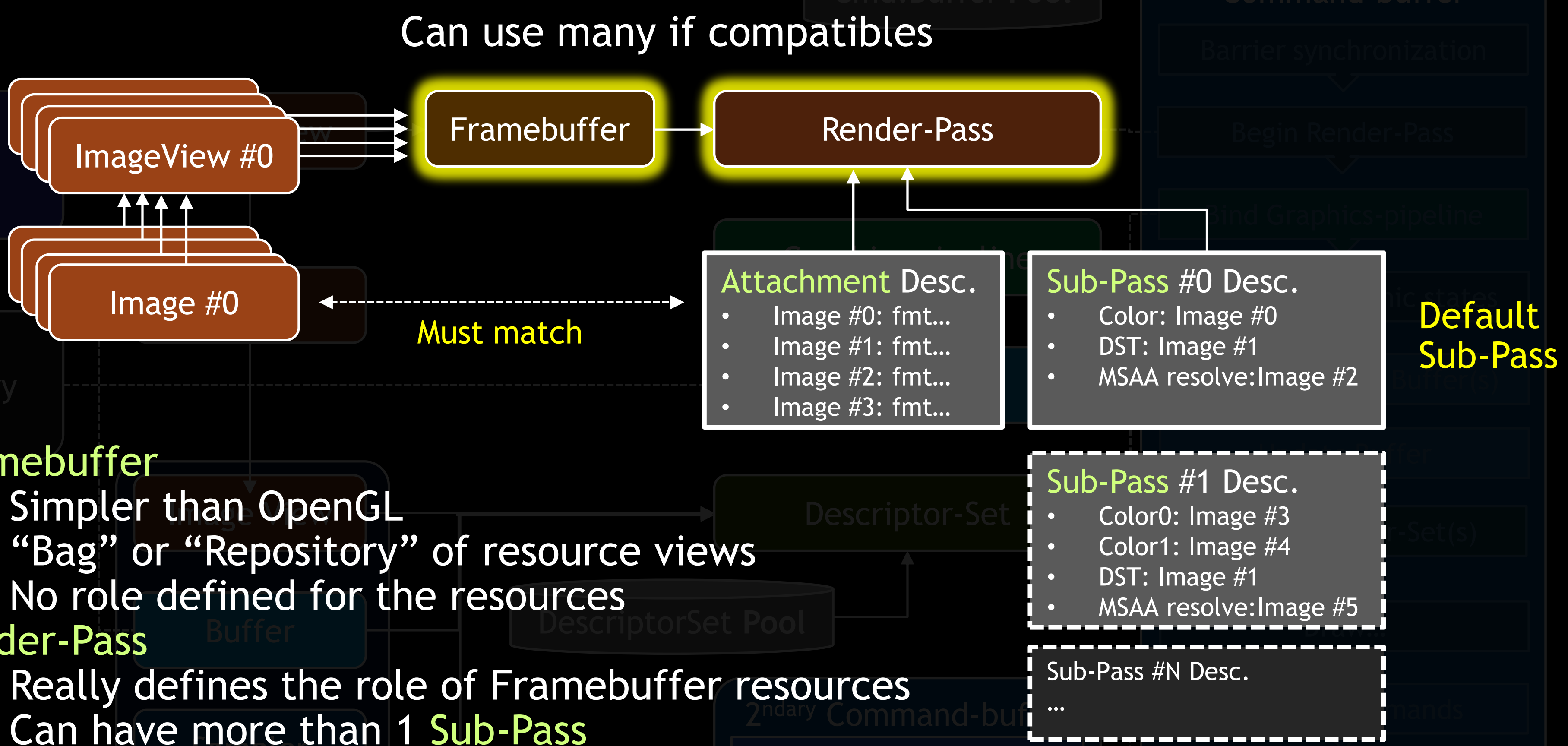
Descriptor-Set



Vulkan Components

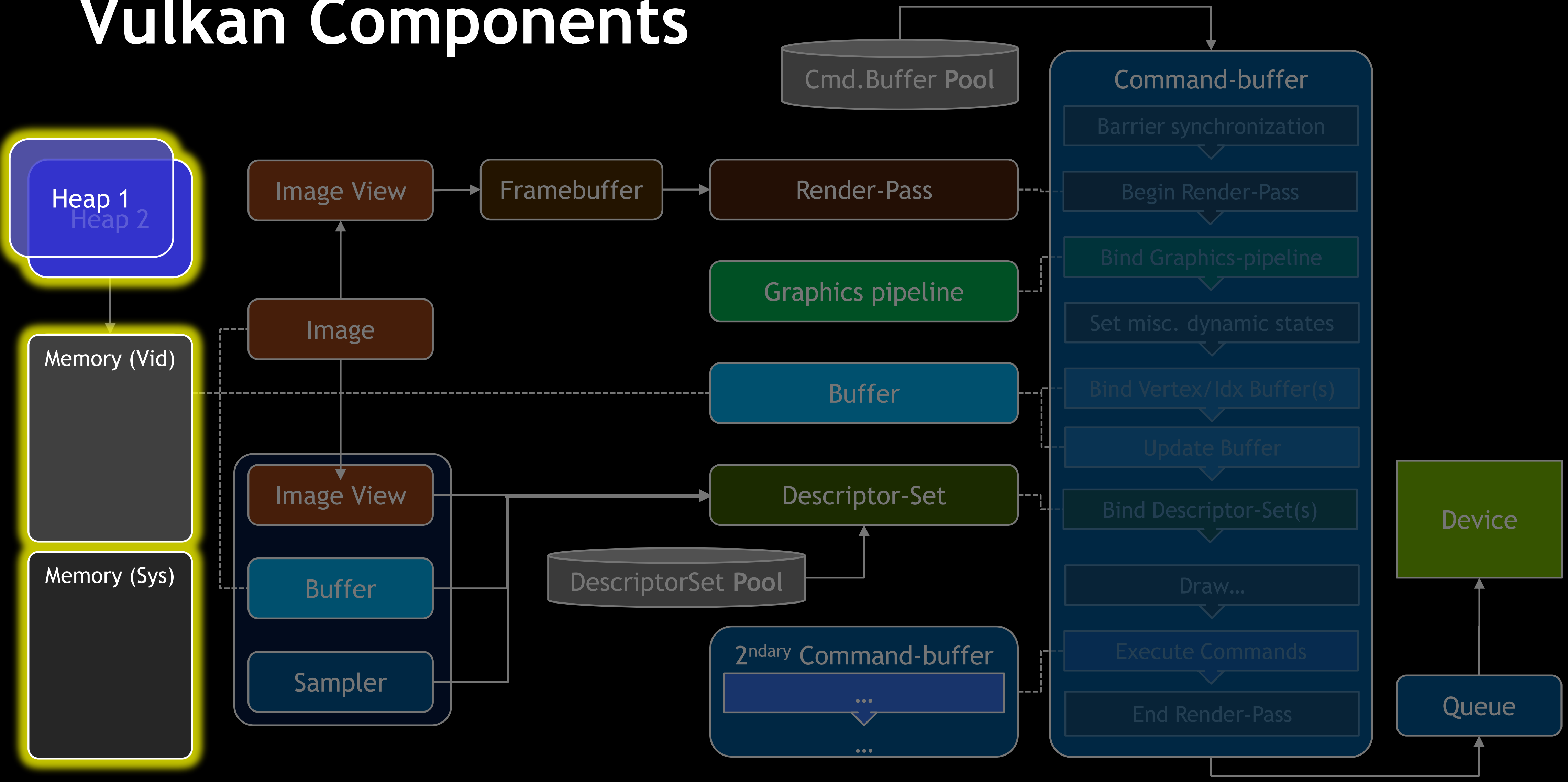


Vulkan Components



- Framebuffer
  - Simpler than OpenGL
  - “Bag” or “Repository” of resource views
  - No role defined for the resources
- Render-Pass
  - Really defines the role of Framebuffer resources
  - Can have more than 1 Sub-Pass
  - Each Sub-Passes defines which Framebuffer resource to use
  - invented for Tilers Arch

Vulkan Components



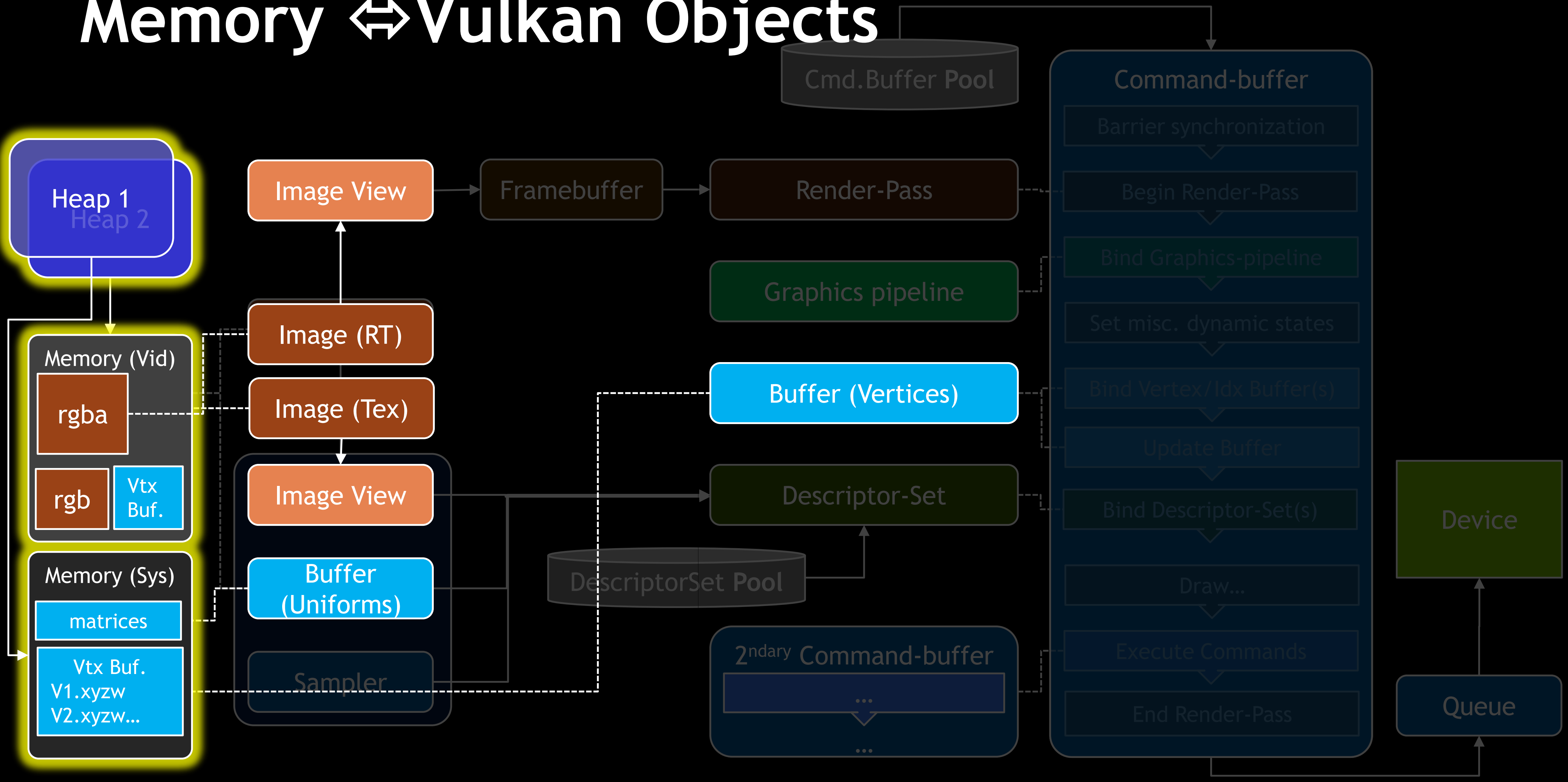
Memory ⇔Vulkan Objects

- Vulkan Objects *referring to buffer(s) of data* need binding to memory
  - Vertex/Index Buffers; Uniform Buffers; Images/Textures…
- Vulkan Device exposes various Memory Heaps - Example:
  - heap 0: size:12,288Mb (Video Memory of my K6000)
  - heap 1: size:17,911Mb (System Memory of my PC)
- And various Memory Types from these Heaps. Example:

Mem.Type	Heap	Flags
0	1 (sys.mem)	-
1	0 (Video)	DEVICE_LOCAL
2	1 (sys.mem)	HOST_VISIBLE | HOST_COHERENT
3	1 (sys.mem)	HOST_VISIBLE | HOST_COHERENT | HOST_CACHED

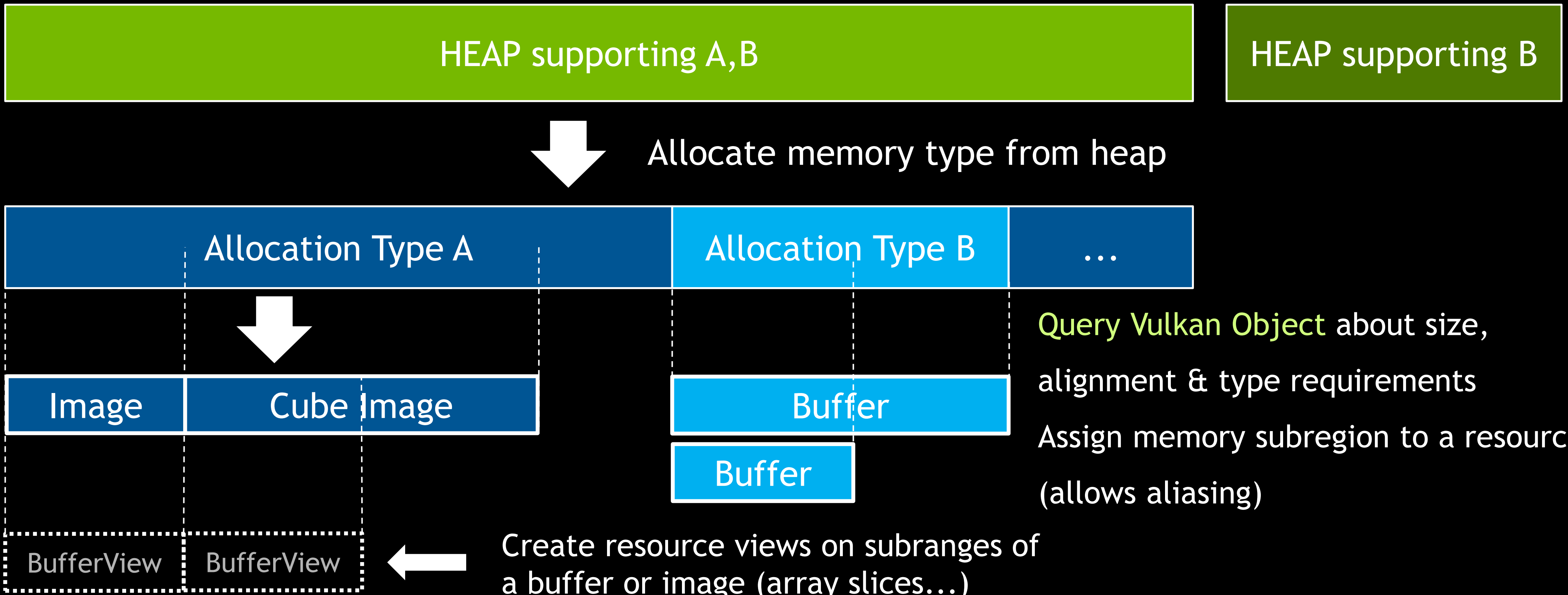
Tegra: Adds one more:
HOST_VISIBLE “NON-Coherent”

Memory ⇔ Vulkan Objects



Resource management

Allocation and Sub allocation



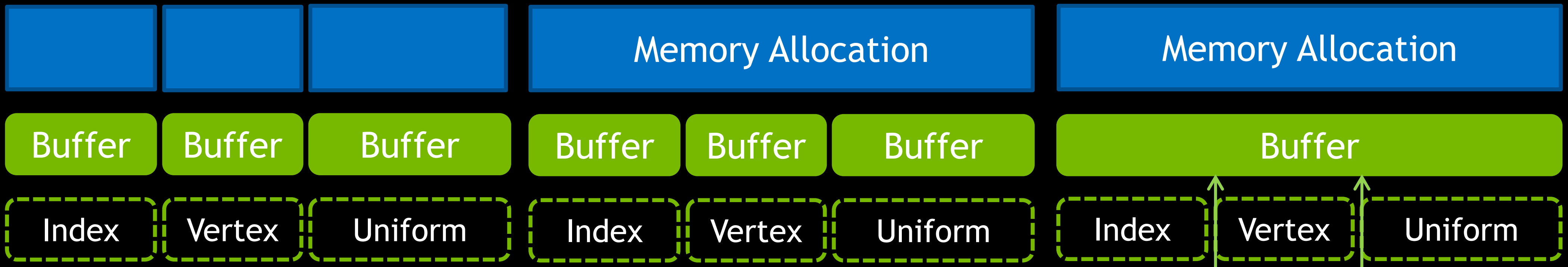
Allocate memory type from heap

Query Vulkan Object about size, alignment & type requirements

Assign memory subregion to a resource (allows aliasing)

Create resource views on subranges of a buffer or image (array slices...)

Resource Management



Not. So. Good.

Better...

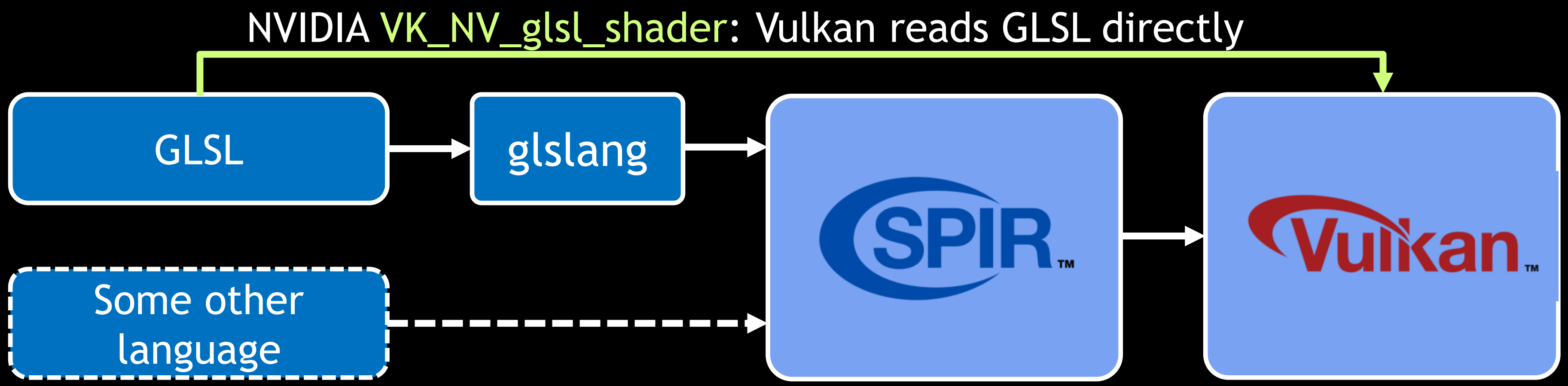
#HappyGPU

Bind same buffer with Offsets > 0

1 buffer can have many types of data

Shaders

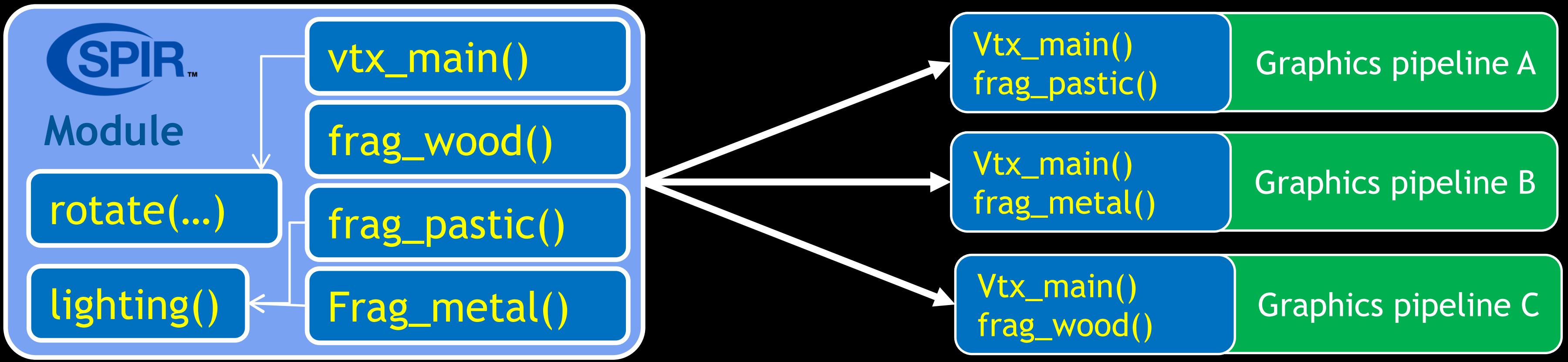
- Vulkan uses SPIR-V passed directly to the driver
  - Can be compiled from GLSL Via glslang or LunarG's glslangValidator; Google ShaderC
  - theoretically other languages could be compiled to Spir-V…
  - Libraries available to compile GLSL to Spir-V from the application
- NVIDIA allows to compile GLSL directly



Shaders

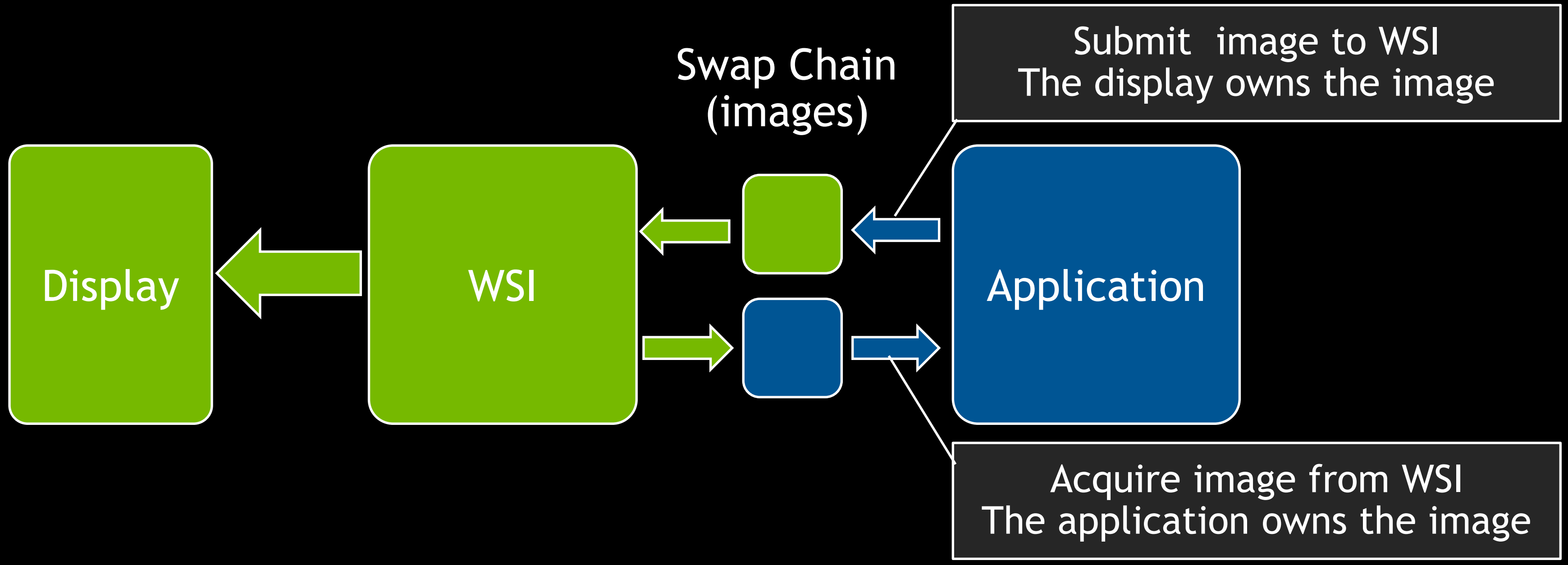
- Multiple entry points can be defined in a signle Spir-V shader-module
- Prevents redundant code: shader module used by many Graphics-Pipelines
- Allows sharing snippets of code
- Easier to share common shader code

Warning: Current GLSL → Spir-V compilers
Don't support this feature, yet
But part of the API & Spir-V
Will happen soon



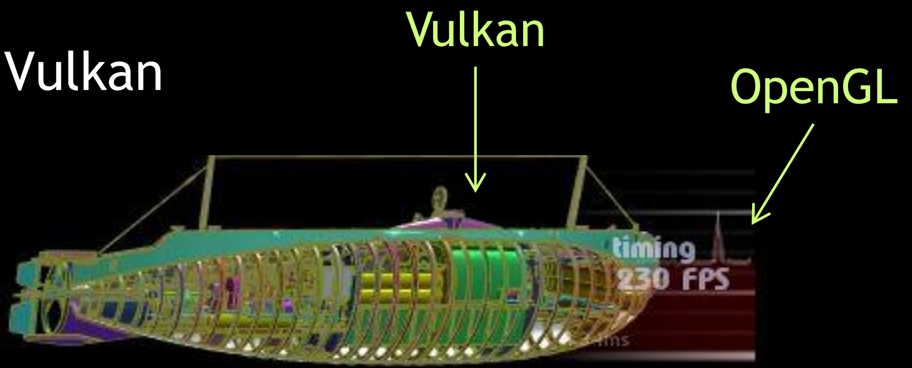
Vulkan Window System Integration (WSI)

- WSI manages the ownership of images via a swap chain
- One image is presented while the other is rendered to
- WSI is a Vulkan Extension



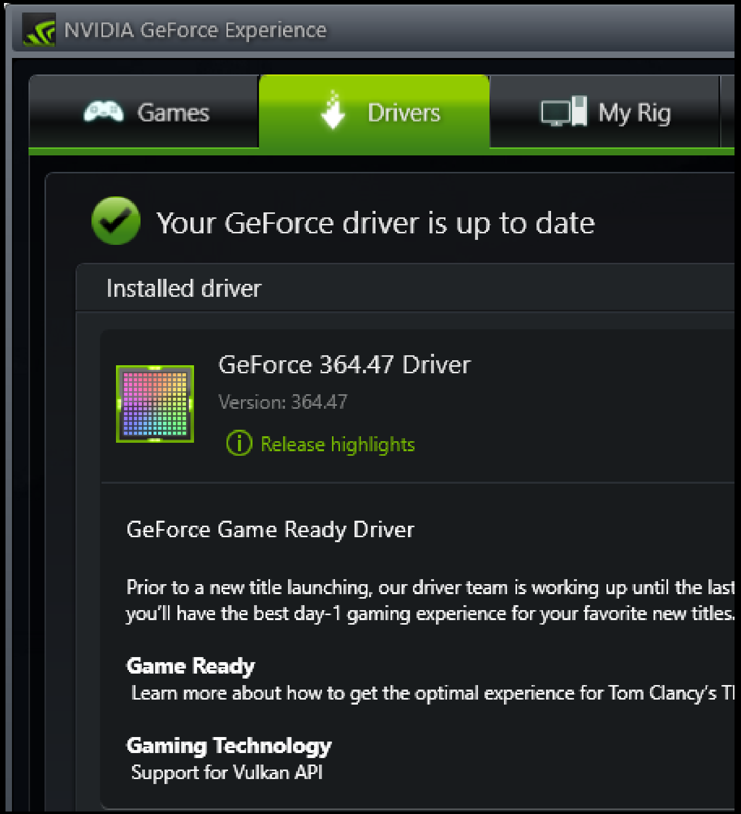
NVIDIA OpenGL ⇔Vulkan Interop

- Alternative to WSI: GL_NV_draw_vulkan_image
- Create an OpenGL Context and all the usual things
- Create Vulkan Device
- Rendering Loop involves both OpenGL and Vulkan
  - Blit the Vulkan image to OpenGL backbuffer: glDrawVkImageNV
  - Extra care on synchronization (Semaphores)
- Bonus: Mix OpenGL rendering (UI overlay…) with Vulkan
  - Allows smooth transition in projects



Pre-requisites to work with Vulkan

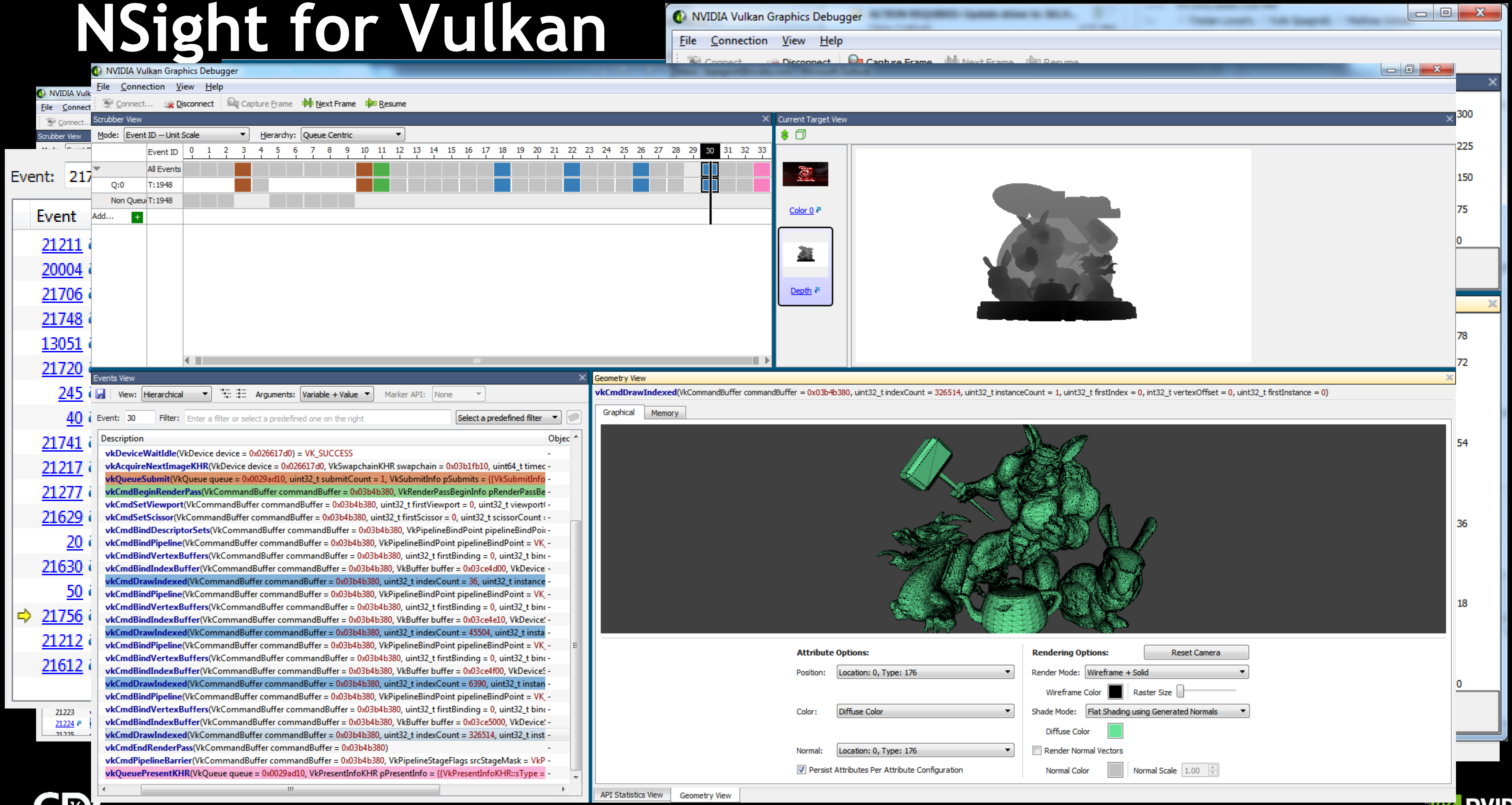
- Lunar-G (http://lunarg.com/)
  - Vulkan Loader (+Source code)
  - Tools: Spir-V compiler for GLSL code and other libraries
  - Layers: intermediate code invoked by Vulkan API functions to help debug
  - Vulkan Includes
- Drivers:
  - GeForce Experience (latest is 364.51 for a fix)
  - https://developer.nvidia.com/vulkan-driver
- NVIDIA resources: https://developer.nvidia.com/Vulkan



Recap' On NVIDIA-Specific Features

- Compatible GPUs for Vulkan: Kepler and Higher; Shield Tablet; Shield Android TV
- GLSL can be directly sent to Vulkan
- GL_NV_draw_vulkan_image can replace WSI
- 16 Queues. All available for any kind of use
- 2 frames in flight with WSI
- All Host memories are "Coherent" (except one for Tegra)
- Layout transitions don't exist in our HW (VK_IMAGE_LAYOUT_GENERAL)
- Linear-Tiling only for 2D non-mipmapped textures
- Shaders never need re-compilation due to states in Graphics-pipeline

NSight for Vulkan



Recap' on Vulkan Philosophy

- Validate as much as possible up-front (DescriptorSets; Pipelines…)
  - The driver doesn't waste time on figuring-out how to set things-up
- Reuse existing patterns of Graphics-Pipelines: cached pipelines
- Know your application: Taylor Vulkan design according to it
- Know your memory usage: You are in charge of optimal sub-allocations
- Explicit multi-threading for graphics: Application's responsibility
- Explicit Resource updates: Either through [non]Coherent buffers; or Queue-Based DMA transfers

Thank you !

Feedback welcome: tlorach@nvidia.com

Vulkan info from NVIDIA:

- https://developer.nvidia.com/Vulkan
- https://developer.nvidia.com/vulkan-graphics-api-here

Samples + Source code in OpenGL and Vulkan:

- https://github.com/nvpro-samples

Other:

- https://gameworks.nvidia.com
- https://developer.nvidia.com/designworks
- http://vulkan.gpuinfo.org/listreports.php