GPU TECHNOLOGY CONFERENCE

April 4-7, 2016 | Silicon Valley

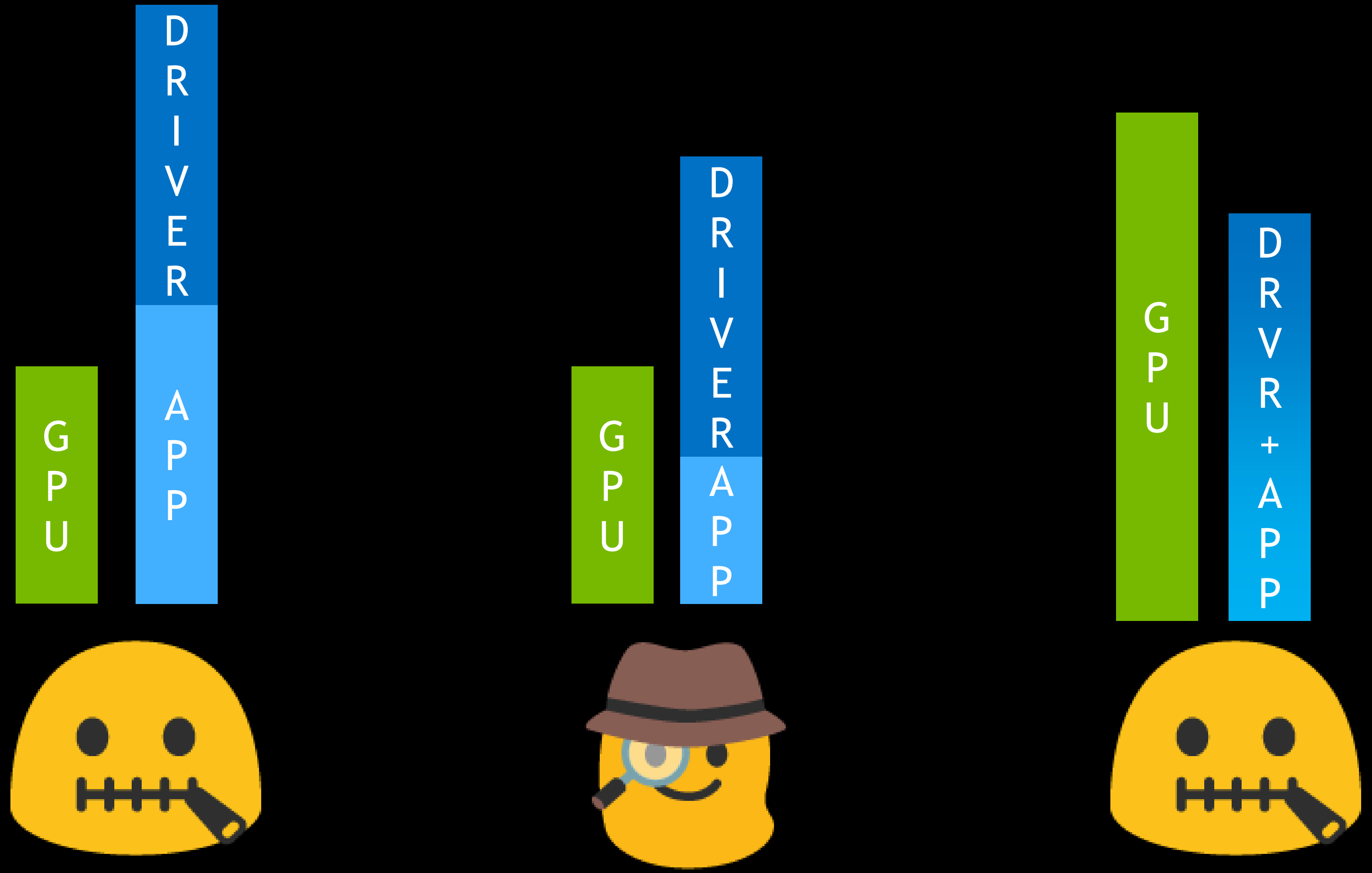
HIGH-PERFORMANCE, LOW-OVERHEAD RENDERING WITH OPENGL AND VULKAN

Edward Liu, April 4th 2016

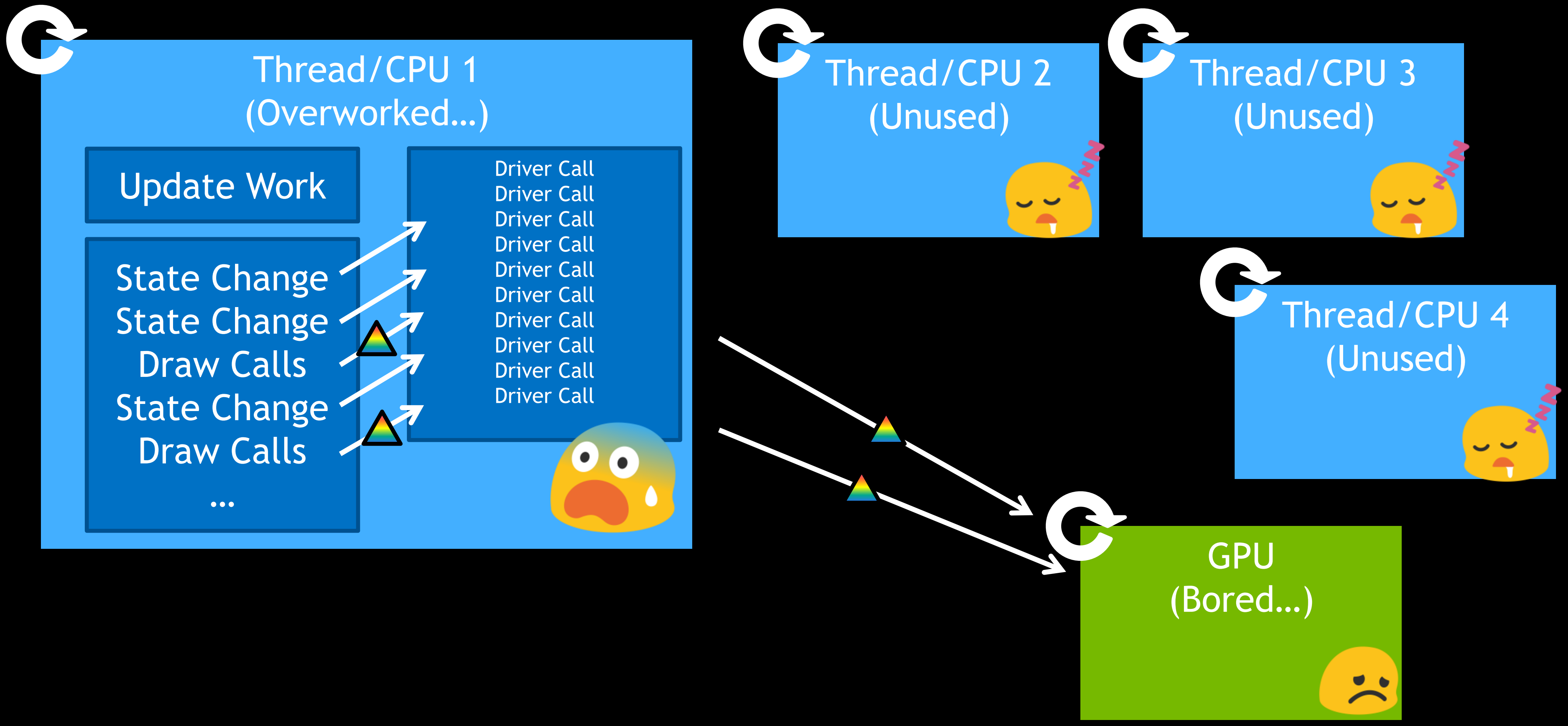
PRESENTED BY

NVIDIA.

What is this talk (not) about?



What is the issue?

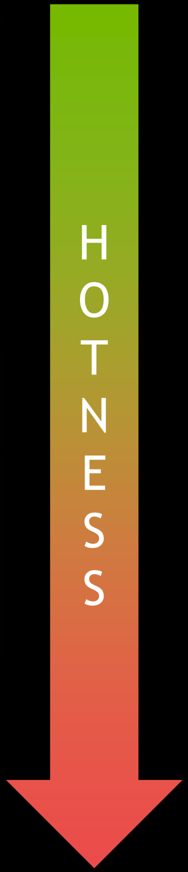


BOTTLENECKS IN RENDERING LOOP

HOTNESS

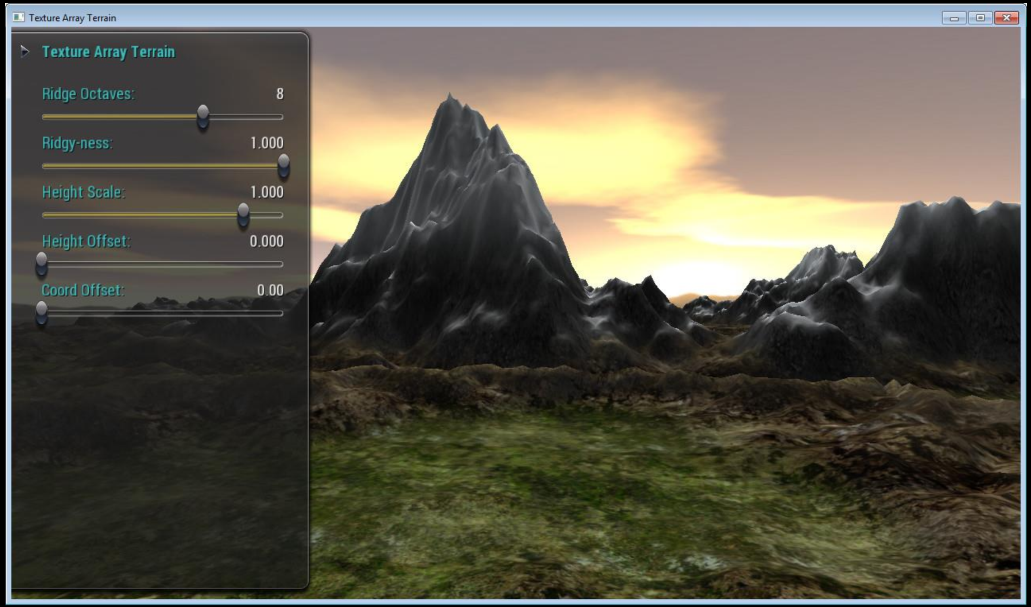
```
foreach render pass {
  set render pass state (e.g. framebuffer, blending, depth/stencil…)
  foreach shader {
    set shader state (e.g. shader, VS, PS…)
    foreach material {
       set material state (e.g. textures, uniforms)
       foreach object/geometry {
          set object/geometry state (e.g. vertex/index buffers, matrices)
          draw calls
       }
    }
  }
}
```

BOTTLENECKS IN RENDERING LOOP

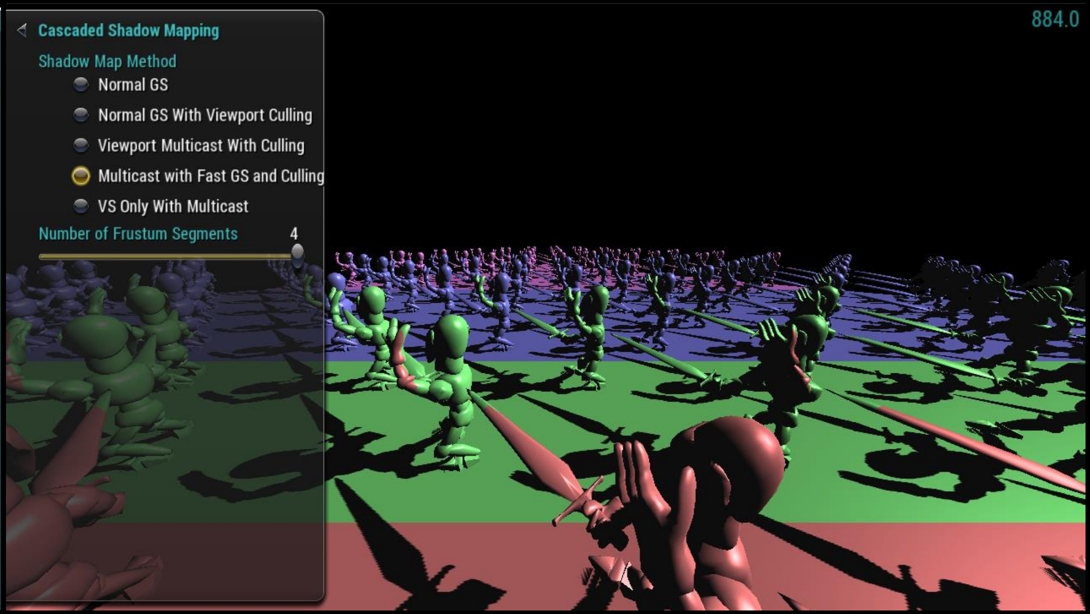



```
foreach render pass {
  set render pass state (e.g. framebuffer, blending, depth/stencil…)
  foreach shader {
    set shader state (e.g. shader, tessellation…)
    foreach material {
       set material state (e.g. textures, uniforms)
       foreach object/geometry {
          set object/geometry state (e.g. vertex/index buffers, matrices)
          draw calls
       }
    }
  }
}
```

MORE TRIANGLES HELP INCREASING COMPLEXITY



Tessellation

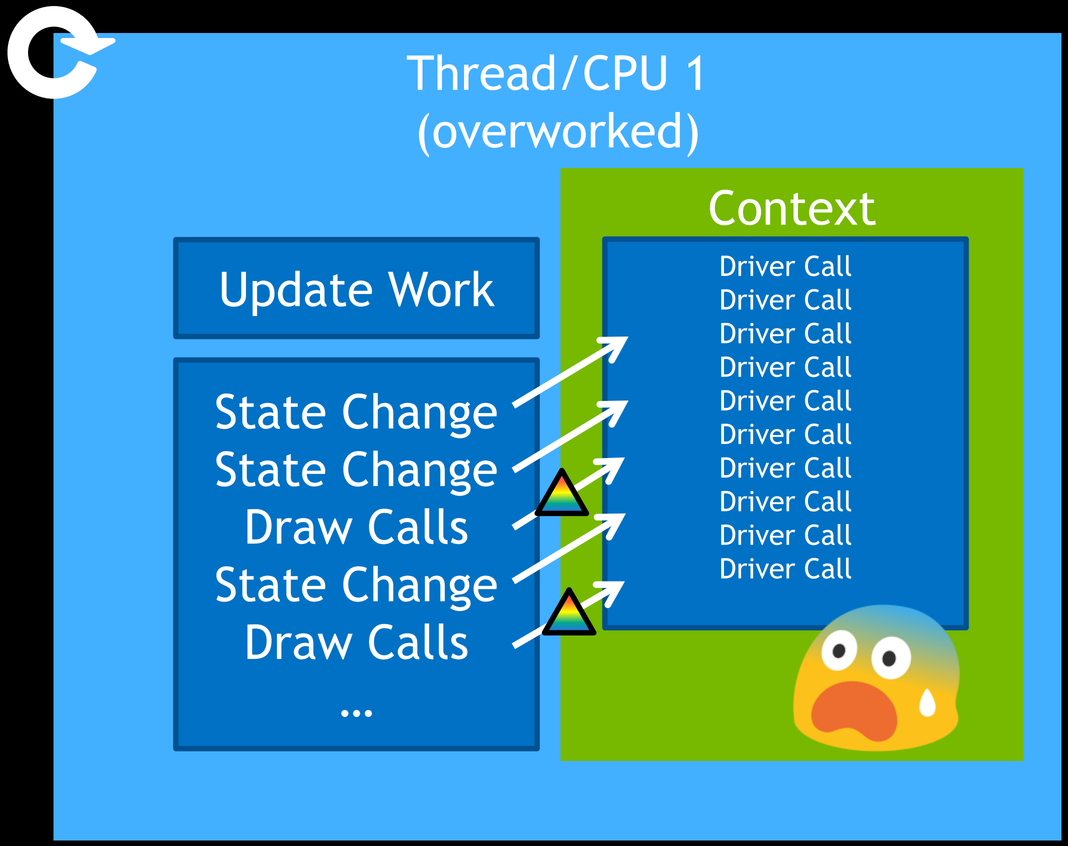


Instancing

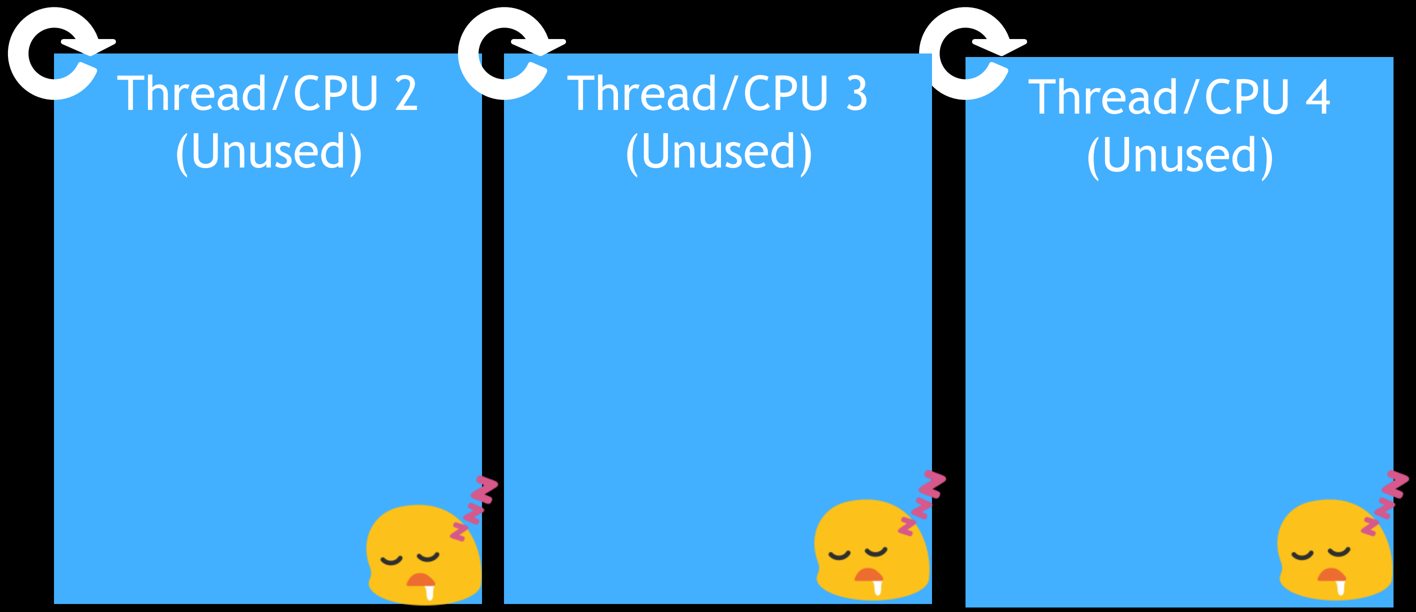
BUT WE ACTUALLY WANT THIS

Assassin’s Creed Unity, courtesy of Ubisoft

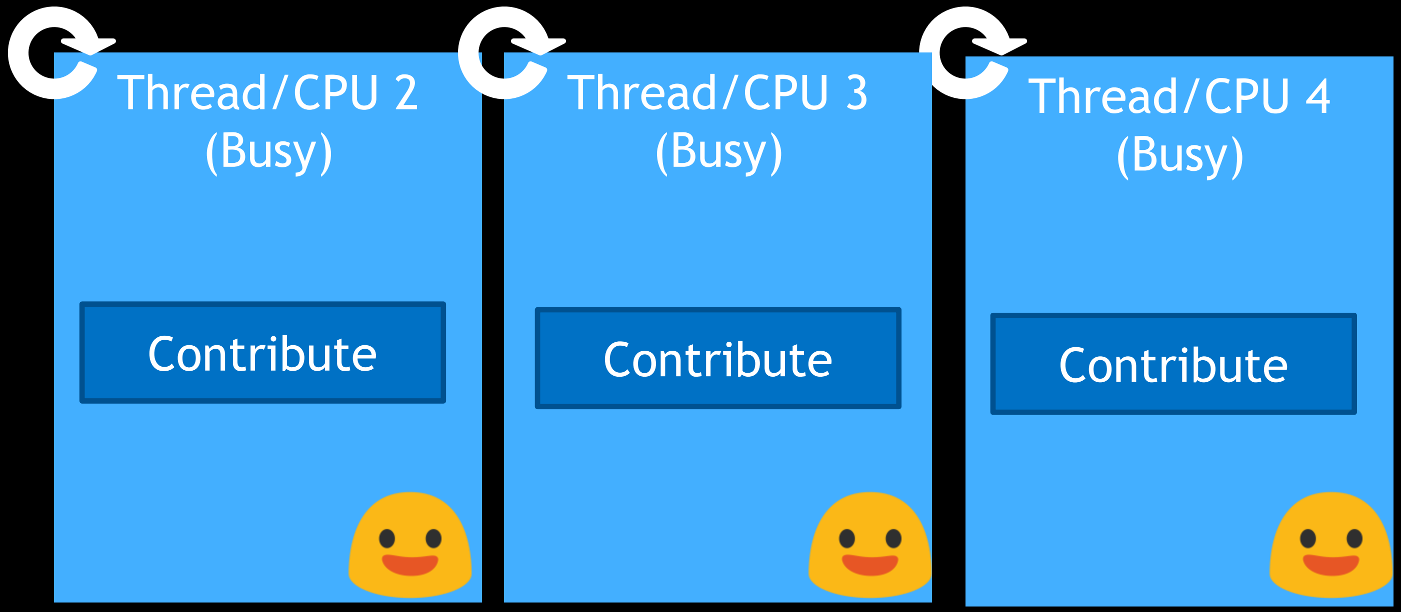
TRADITIONAL 3D APIS: USE "HEAVY" CONTEXTS



Developers Want Threading-Friendly APIs!



Developers Want Threading-Friendly APIs!



TRADITIONAL 3D APIS: PERFORM IMPLICIT WORK

Examples of implicit operations

- compiling shaders, downloading textures, downsampling
- synchronization, validation & error checking

Unpredictable!

Symptoms

- stalls when changing
  - shader, blend mode, vertex data layout, framebuffer attachment formats…

Developers want to explicitly schedule those

UPDATING OPENGL: “AZDO”

Popular OpenGL extensions for **A**pproaching **Z**ero **D**river **O**verhead

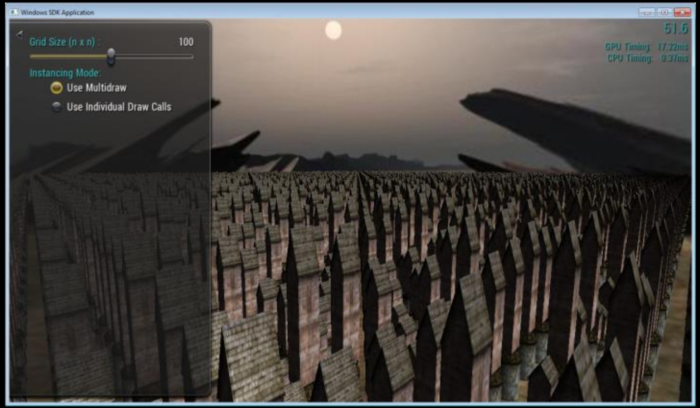
- Not a single, monolithic set
- multiple extensions used for different aspects

Improved dynamic data update model

- OpenGL 4.3/GL_ARB_buffer_storage
- glBufferStorage & glMapBuffer(GL_MAP_PERSISTENT_BIT)

TODAY'S "AZDO" FOCUS

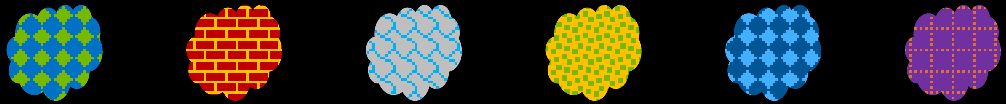
More varied geometry per drawcall via "MultiDrawIndirect"

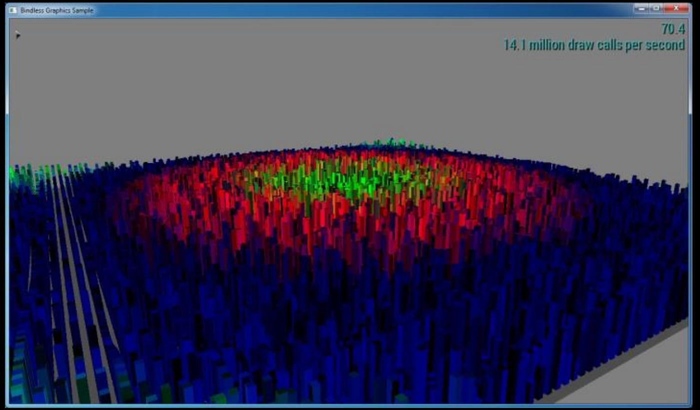


OpenGL 4.3/GL_ARB_multi_draw_indirect

glMultiDrawArraysIndirect & glMultiDrawElementsIndirect

More varied materials per draw call via "bindless" resources





GL_ARB_bindless_texture & GL_NV_bindless_texture

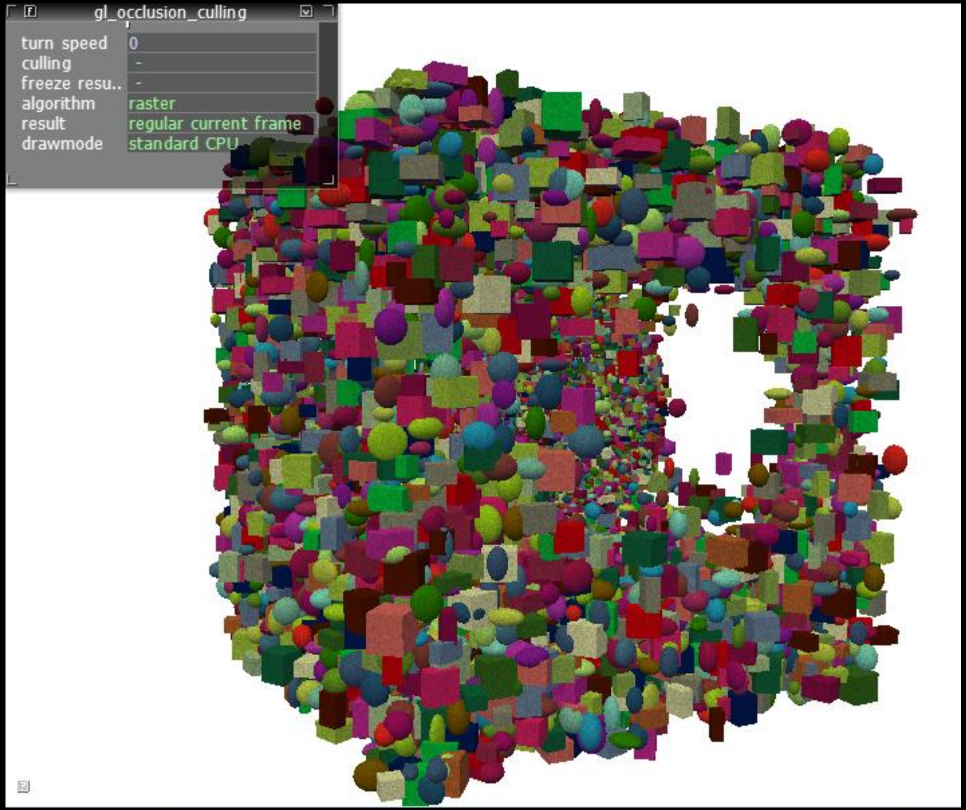
GL_NV_shader_buffer_load

GL_NV_{vertex|uniform}_buffer_unified_memory

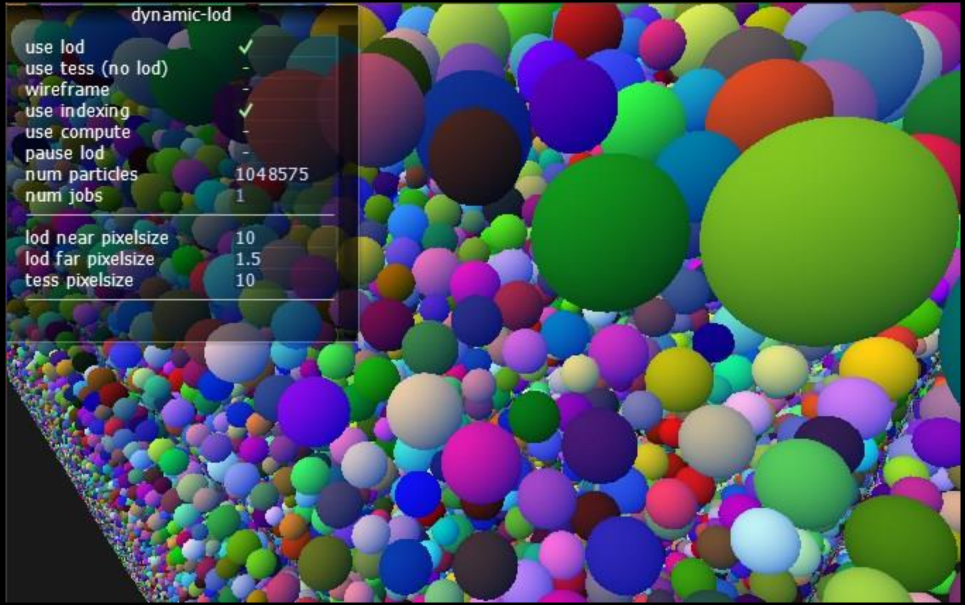
MULTI DRAW INDIRECT

```
for (d = 0; d < drawcount; ++d)
    glDrawArrays(    GL_TRIANGLES, first[d], count[d]);



glMultiDrawArrays(GL_TRIANGLES, first[], count[], GLsizei drawcount);

struct {
    uint  count;
    uint  instanceCount;
    uint  first;
    uint  baseInstance;
} DrawArraysIndirectCommand;

glMultiDrawArraysIndirect(GL_TRIANGLES, const void *indirect, drawcount, stride);
```

TRANSPARENT LAYOUT OF “INDIRECT” BUFFER…

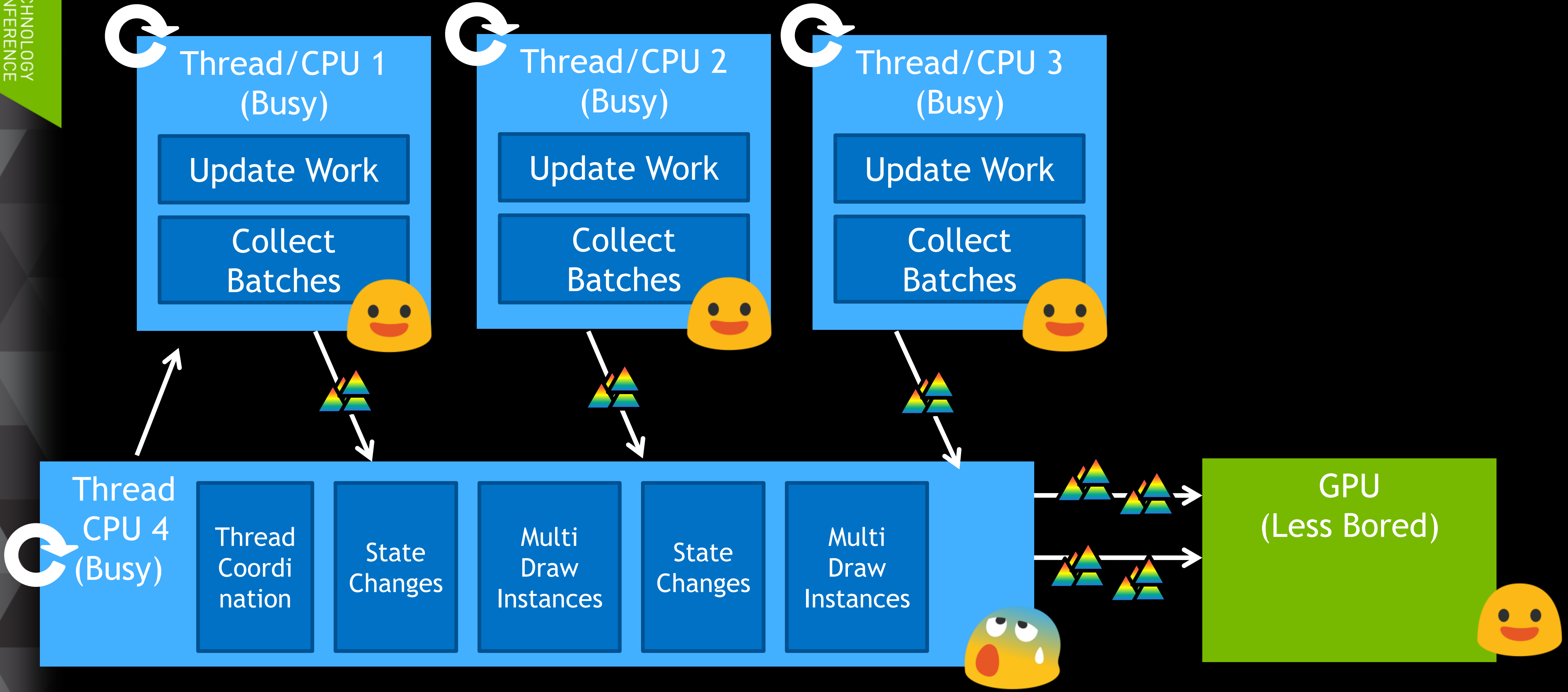


GPU occlusion culling



GPU dynamic level of detail

THREADING WITH MULTI DRAW INDIRECT



MULTI DRAW INDIRECT LIMITATIONS

Cannot change vertex & index buffer bindings “inline”

- pack index buffer (IB) and/or vertex buffer (VB)

vertices_0	vertices_1	vertices_2	vertices_3
indices_0	indices_1	indices_2	indices_3

Cannot change

- shaders
- texture bindings
- framebuffer object (FBO)
- uniform buffer object (UBO)

What if…?

Encode more in “indirect” buffer

- resource bindings
- state changes
- different draw call types

Compute more GPU “work” in worker threads

GL_NV_command_list

- essentially Multi Draw Indirect on steroids
- explores modern API concepts in OpenGL

ELEMENT_ADDRESS_COMMAND_NV
ATTRIBUTE_ADDRESS_COMMAND_NV
UNIFORM_ADDRESS_COMMAND_NV

BLEND_COLOR_COMMAND_NV
STENCIL_REF_COMMAND_NV
LINE_WIDTH_COMMAND_NV
POLYGON_OFFSET_COMMAND_NV
ALPHA_REF_COMMAND_NV
VIEWPORT_COMMAND_NV
SCISSOR_COMMAND_NV
FRONTFACE_COMMAND_NV

DRAW_ELEMENTS_COMMAND_NV
DRAW_ARRAYS_COMMAND_NV
DRAW_ELEMENTS_STRIP_COMMAND_NV
DRAW_ARRAYS_STRIP_COMMAND_NV
DRAW_ELEMENTS_INSTANCED_COMMAND_NV
DRAW_ARRAYS_INSTANCED_COMMAND_NV

TERMINATE_SEQUENCE_COMMAND_NV
NOP_COMMAND_NV

GL_NV_command_list CONCEPTS

Tokenized Rendering

- Some state changes and all draw commands are encoded into binary data stream
- Binary stream layout **transparent** to GPU and CPU!

State Objects

- Whole OpenGL States (program, blending...) captured as an object
- Allows pre-validation + fast reuse

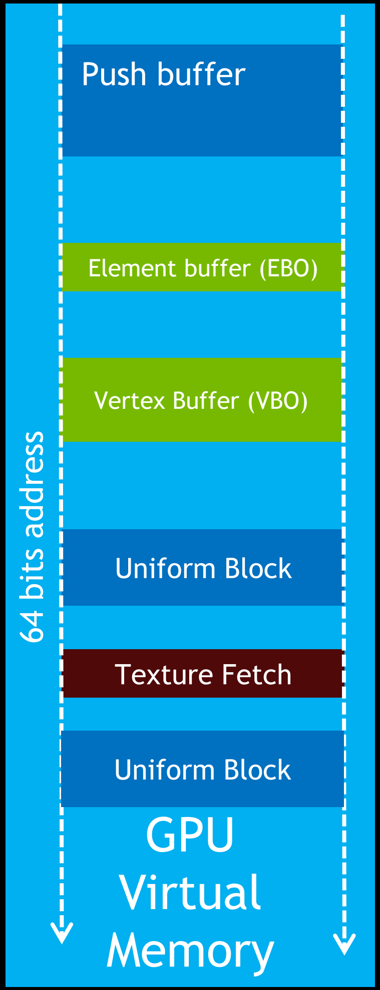
Execution either “interpreted” or “baked” via command list object

Referencing Resources via “Bindless” GPU addresses

- content can still be modified (matrices, vertices...)

REFERENCING RESOURCES WITH "BINDLESS"

- Work from native GPU pointers/handles
    - less CPU work, less locking
    - flexible data structures on GPU
- Bindless Buffers
    - Vertex & Global memory since Tesla (2008+)
- Bindless Textures
    - Since Kepler (2012+)
- Bindless Constants (UBO)
- **Bindless plays a central role for Command-List**

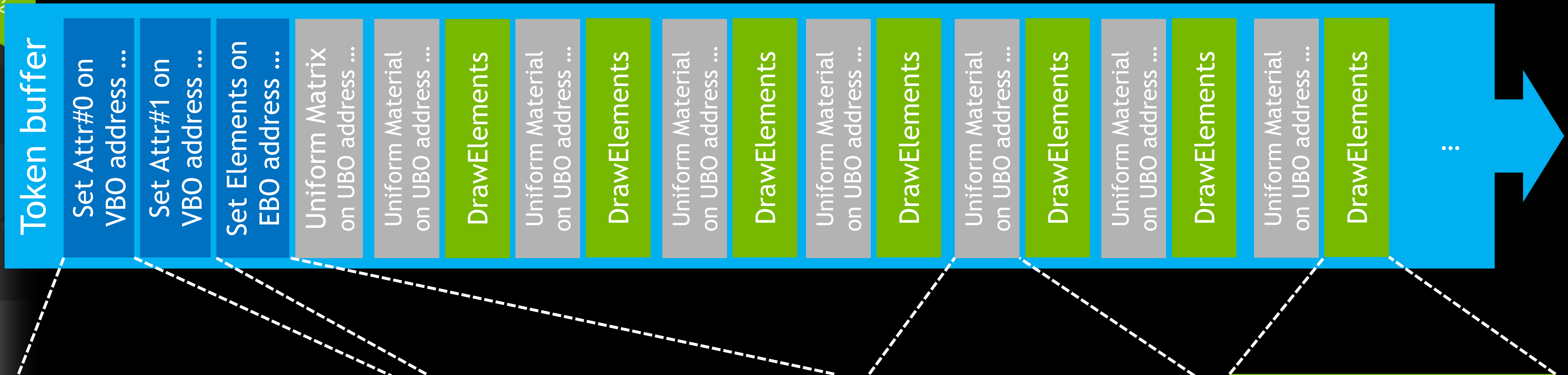


EXAMPLE ON USING BINDLESS UBO

```
UpdateBufferContent( bufferId );

glMakeNamedBufferResidentNV( bufferId, READ);

GLuint64 bufferAddr;
glGetNamedBufferParameteri64v( bufferId, BUFFER_GPU_ADDRESS_NV, &bufferAddr );

glEnableClientState( UNIFORM_BUFFER_UNIFIED_NV );

foreach (obj in scene) {
    ...
    // glBindBufferRange  ( UNIFORM_BUFFER_OBJECT, 0, bufferId, obj.matrixOffset,
    maSize );
    glBufferAddressRangeNV( UNIFORM_BUFFER_ADDRESS_NV, 0, bufferAddr +
    obj.matrixOffset, maSize );
}
```

TOKEN BUFFER STRUCTURES

Tokens-buffers are tightly packed structs in linear memory




```
TokenVbo
{
  GLuint header;

  AttributeAddressCommandNV
  {
    GLuint   index;
    GLuint64 address;
  } cmd;
}
```

```
TokenIbo
{
  GLuint header;

  ElementAddressCommandNV
  {
    GLuint64 address;
    GLuint   typeSizeInByte;
  } cmd;
}
```

```
TokenUbo
{
  GLuint header;
  UniformAddressCommandNV
  {
    GLushort   index;
    GLushort   stage;
    GLuint64   address;
  } cmd;
}
```

```
TokenDrawElements
{
  Gluint header;
  DrawElementsCommandNV
  {
    GLuint  count;
    GLuint  firstIndex;
    GLuint  baseVertex;
  } cmd;
}
```

PRECOMPILED STATE OBJECTS

```
Gluint stateObject;

glStateCaptureNV (stateobject, GL_TRIANGLES );
```

Majority of state + primitive type

framebuffer formats, shader, blend mode, depth ...)

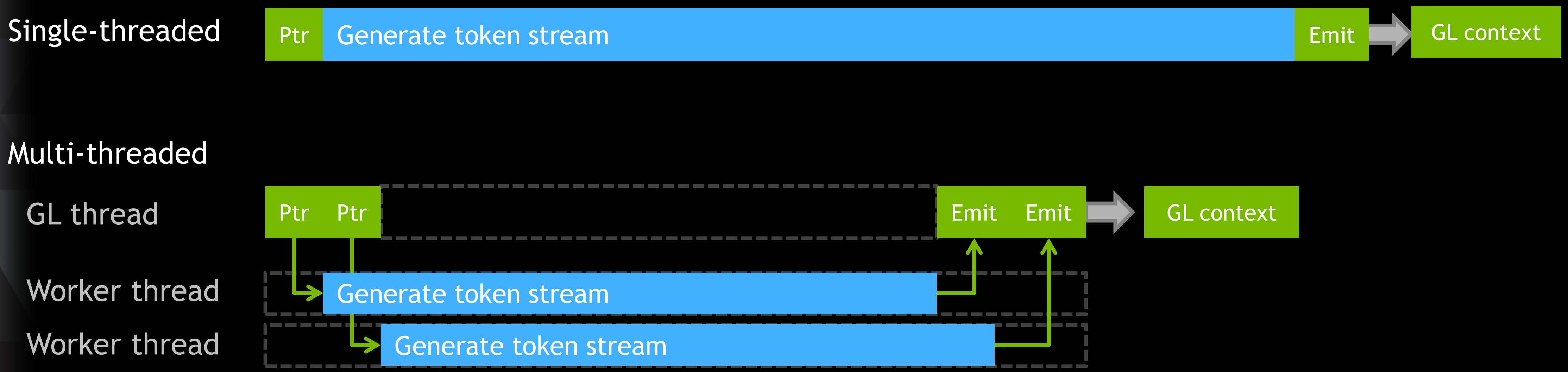
Immutable

„Bindless“ for resource

Note: texture GPU addresses also passed via UBO

THREADING AND COMMAND LISTS

Fill token buffers if reuse impossible



COMMAND LIST LIMITATIONS

- Command-List does NOT pretend to solve general OpenGL multi-threading
  - allows partially multi-threaded work creation
- single-threaded state validation
  - State Object Capture must be handled in OpenGL context
  - but worker threads “know” state for render workload

OPENGL RESOURCES (1/2)

Sample Code

- https://github.com/nvpro-samples/gl_occlusion_culling
- https://github.com/nvpro-samples/gl_dynamic_lod
- https://github.com/nvpro-samples/gl_vk_threaded_cadscene

Presentations

- http://on-demand.gputechconf.com/gtc/2015/presentation/S5135-Christoph-Kubisch-Pierre-Boudier.pdf (command list and culling)
- http://on-demand.gputechconf.com/siggraph/2014/presentation/SG4117-OpenGL-Scene-Rendering-Techniques.pdf (which gives a run down on optimizing the hot loop)
- http://en.slideshare.net/tlorach/opengl-nvidia-commandlistapproaching-zerodriveroverhead

OPENGL RESOURCES (2/2)

Extension Specifications

https://www.opengl.org/registry/specs/ARB/multi_draw_indirect.txt

https://www.opengl.org/registry/specs/ARB/buffer_storage.txt

https://www.opengl.org/registry/specs/ARB/bindless_texture.txt

https://www.opengl.org/registry/specs/NV/bindless_texture.txt

https://www.opengl.org/registry/specs/NV/shader_buffer_load.txt

https://www.opengl.org/registry/specs/NV/uniform_buffer_unified_memory.txt

https://www.opengl.org/registry/specs/NV/vertex_buffer_unified_memory.txt

https://www.opengl.org/registry/specs/NV/command_list.txt

VULKAN

VULKAN PHILOSOPHIES

Not specifically "the" core philosophies of Vulkan; just a few we want to highlight

Take advantage of an **application's high-level knowledge**

- Do not require the driver to determine and optimize for "intent" implicitly

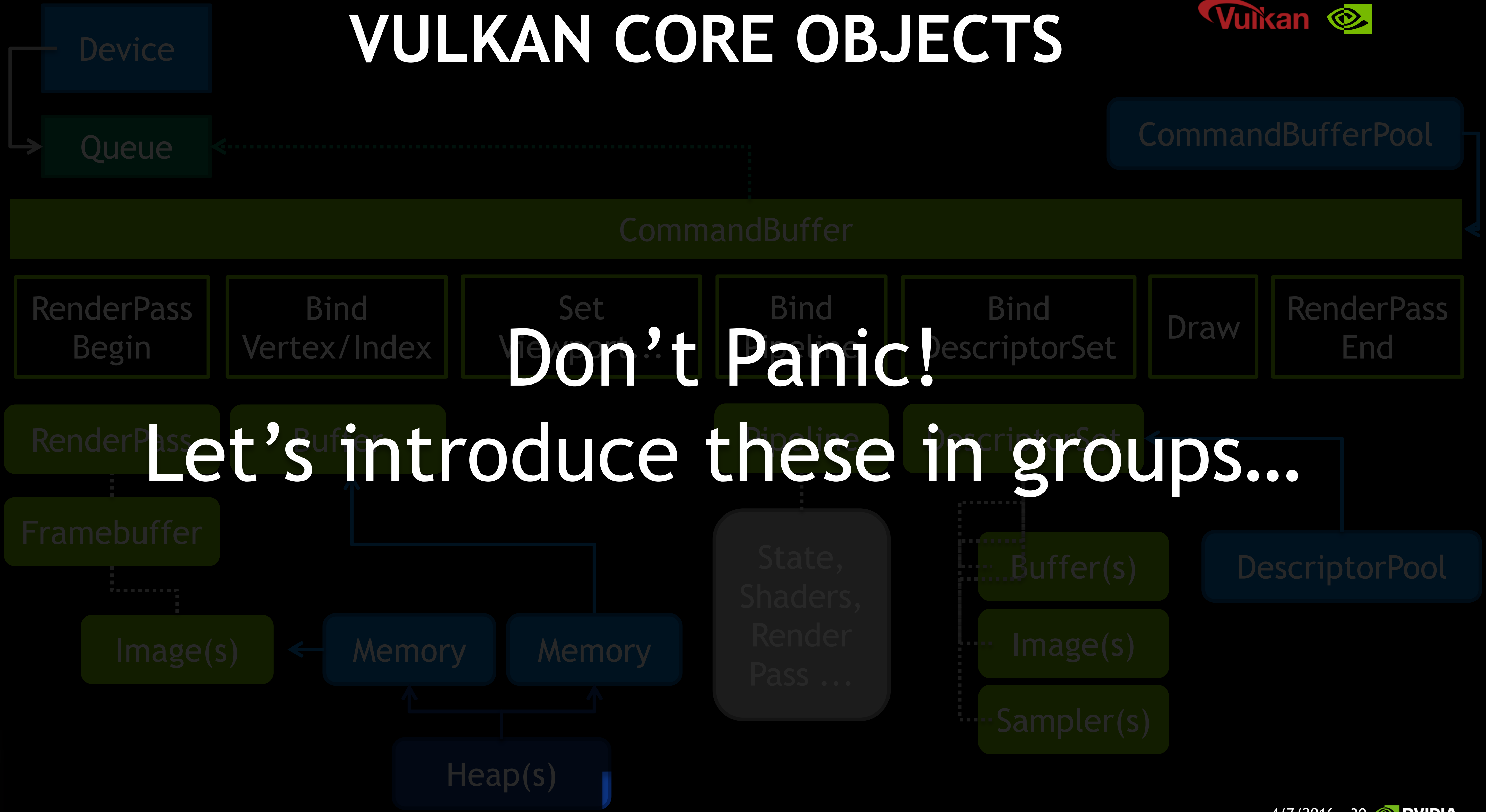
Ensure that the API is **thread-friendly** and explicitly documented for app threading

- Place the synchronization responsibility upon the app to allow higher-level sync

Reduce by **explicit re-use**

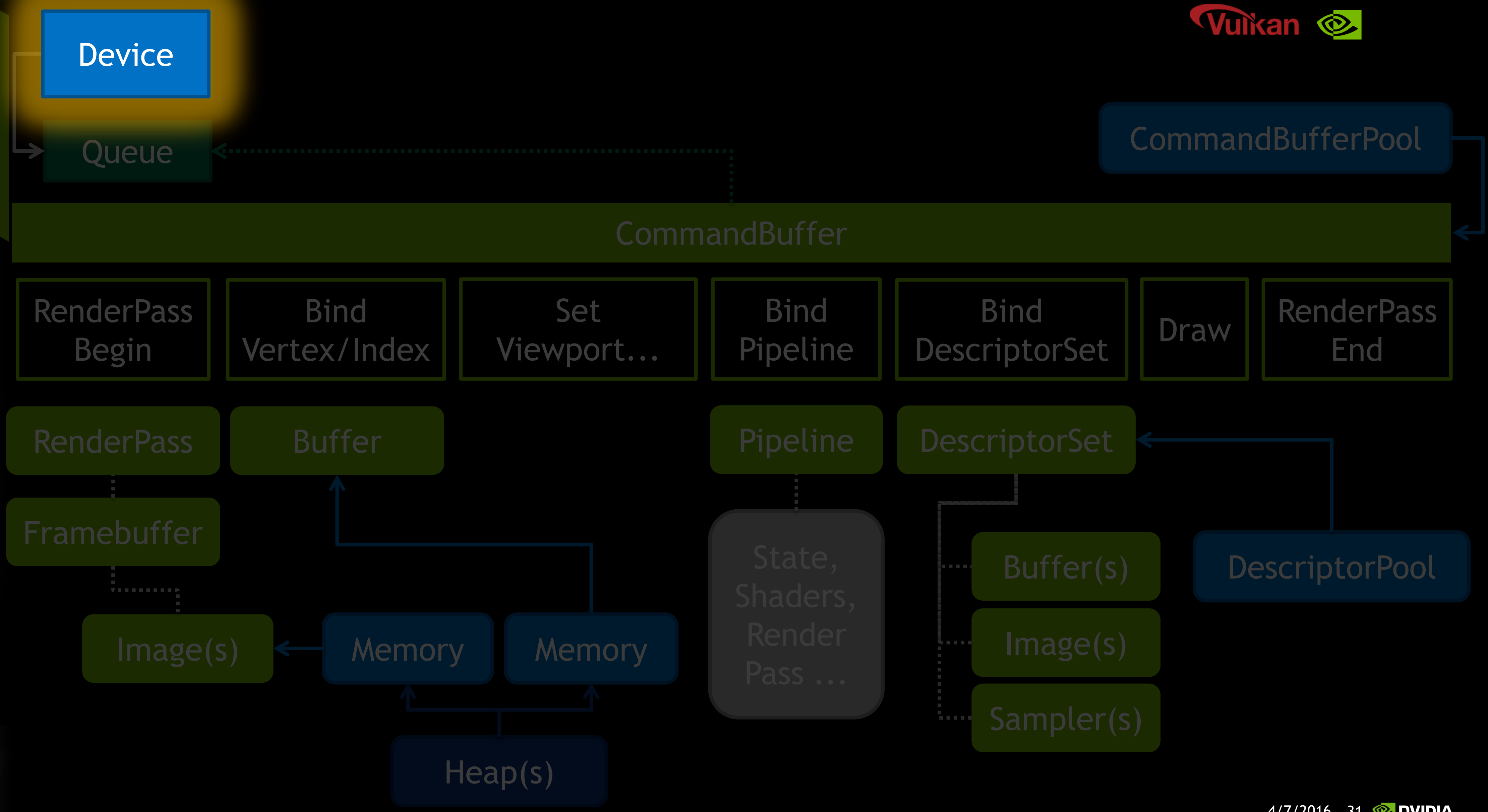
- Make explicit as many cases of resource/state/command reuse as possible

VULKAN CORE OBJECTS



Don't Panic!

Let's introduce these in groups...


Device
Queue
CommandBufferPool
CommandBuffer
RenderPass Begin
Bind Vertex/Index
Set Viewport...
Bind Pipeline
Bind DescriptorSet
Draw
RenderPass End
RenderPass
Buffer
Pipeline
DescriptorSet
Framebuffer
State, Shaders, Render Pass ...
Buffer(s)
DescriptorPool
Image(s)
Memory
Memory
Image(s)
Sampler(s)
Heap(s)

CORE OBJECTS: DEVICES

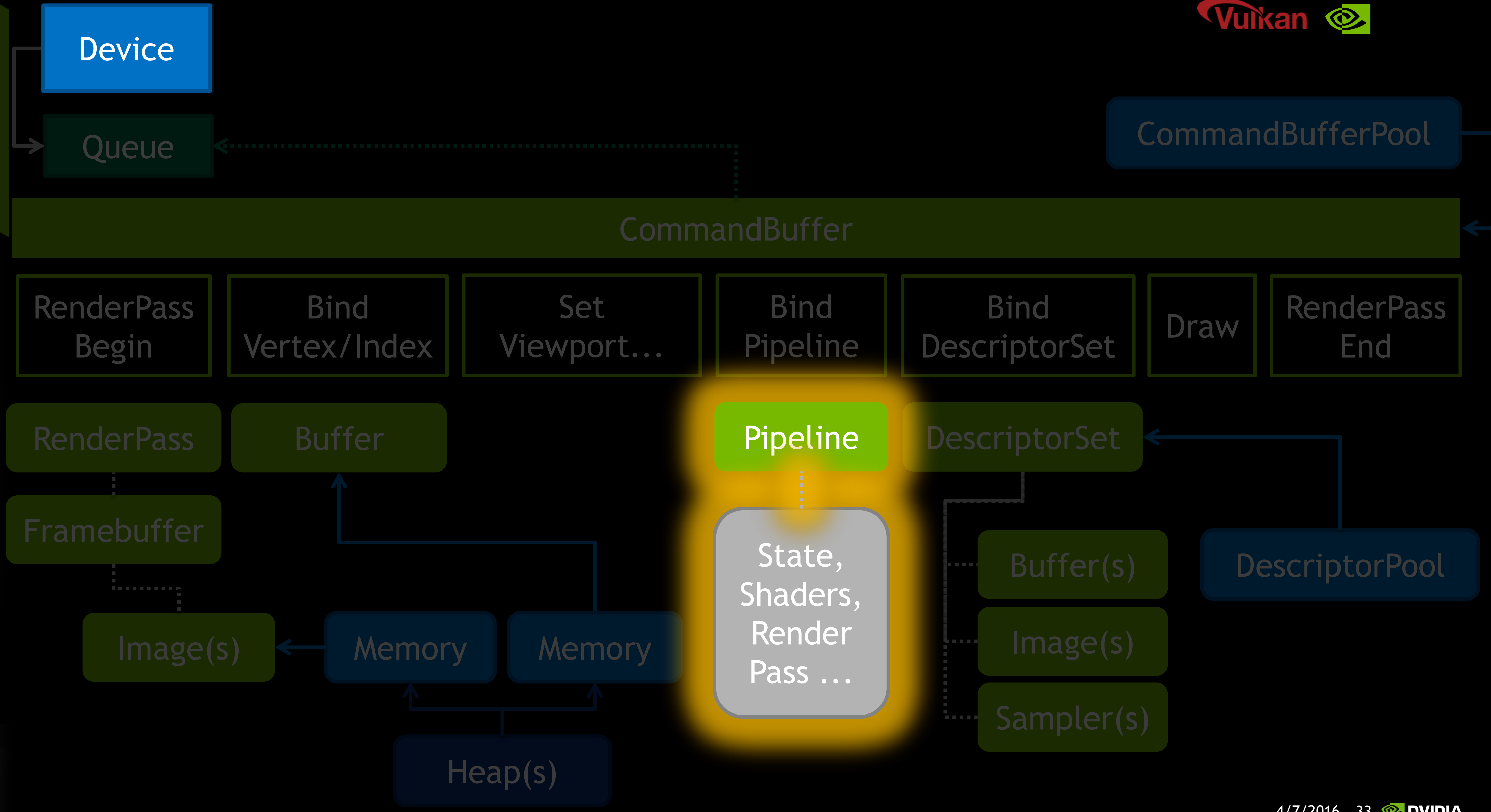
You may have more than 1 Vulkan device on your system

- A VkPhysicalDevice represents the actual hardware on the system.
- Query Vulkan for its available VkPhysicalDevices

VkPhysicalDevice
- Capabilities
- Memory
- Queues
- Buffer Objects
- Images
- Sync Primitives

VkDevice object "methods" include:

- Getting Queues (used for all work submission)
- Device memory management
- Object management (buffers, images, sync primitives)


Device
Queue
CommandBufferPool
CommandBuffer
RenderPass Begin
Bind Vertex/Index
Set Viewport...
Bind Pipeline
Bind DescriptorSet
Draw
RenderPass End
RenderPass
Buffer
Pipeline
DescriptorSet
Framebuffer
State, Shaders, Render Pass ...
Buffer(s)
DescriptorPool
Image(s)
Memory
Memory
Image(s)
Sampler(s)
Heap(s)

CORE OBJECTS: PIPELINES

Vulkan uses a ‘precompiled’ pipeline state object

Core to the API and required for all rendering

Vertex Input	Rasterization	Depth/Stencil	Viewport	Multisample

‘Bakes’ in everything that Vulkan needs to run without re-validating, eg.

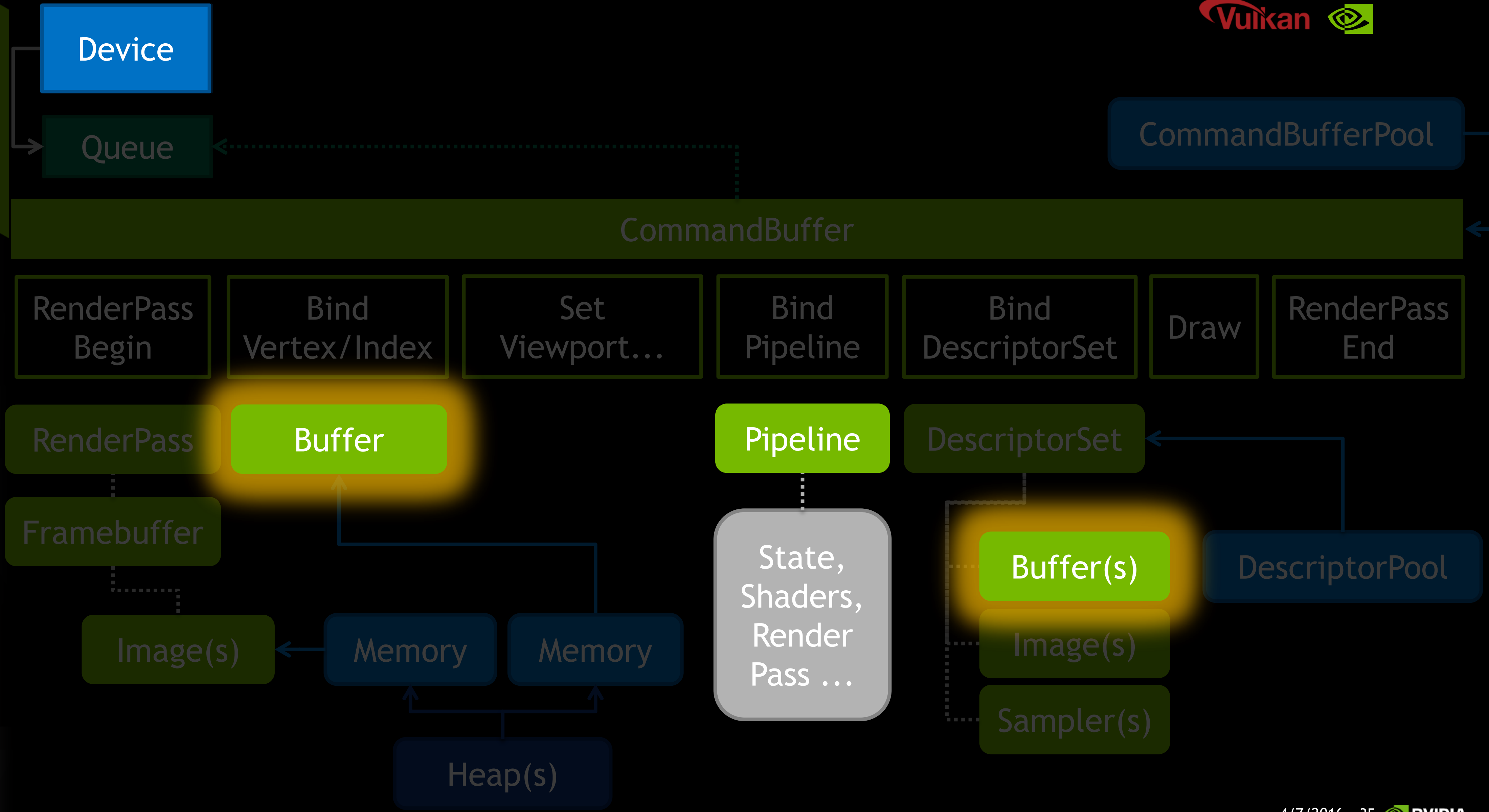
Some states can still be changed without causing shader recompilation

Therefore the pipeline does not have to be rebaked

These are the Dynamic States, eg.

Viewport	Scissor	Blend const	Stencil Ref	Depth Bounds	Depth Bias

Analogous to NV_Command_List state objects, but created and set explicitly


Device
Queue
CommandBufferPool
CommandBuffer
RenderPass Begin
Bind Vertex/Index
Set Viewport…
Bind Pipeline
Bind DescriptorSet
Draw
RenderPass End
RenderPass
Buffer
Pipeline
DescriptorSet
Framebuffer
State, Shaders, Render Pass …
Buffer(s)
DescriptorPool
Image(s)
Memory
Memory
Image(s)
Sampler(s)
Heap(s)

CORE OBJECTS: BUFFERS

Contain per-vertex, per-instance or uniform-level data

(Highly) Heterogeneous

- More on this later

Multiple memory types:

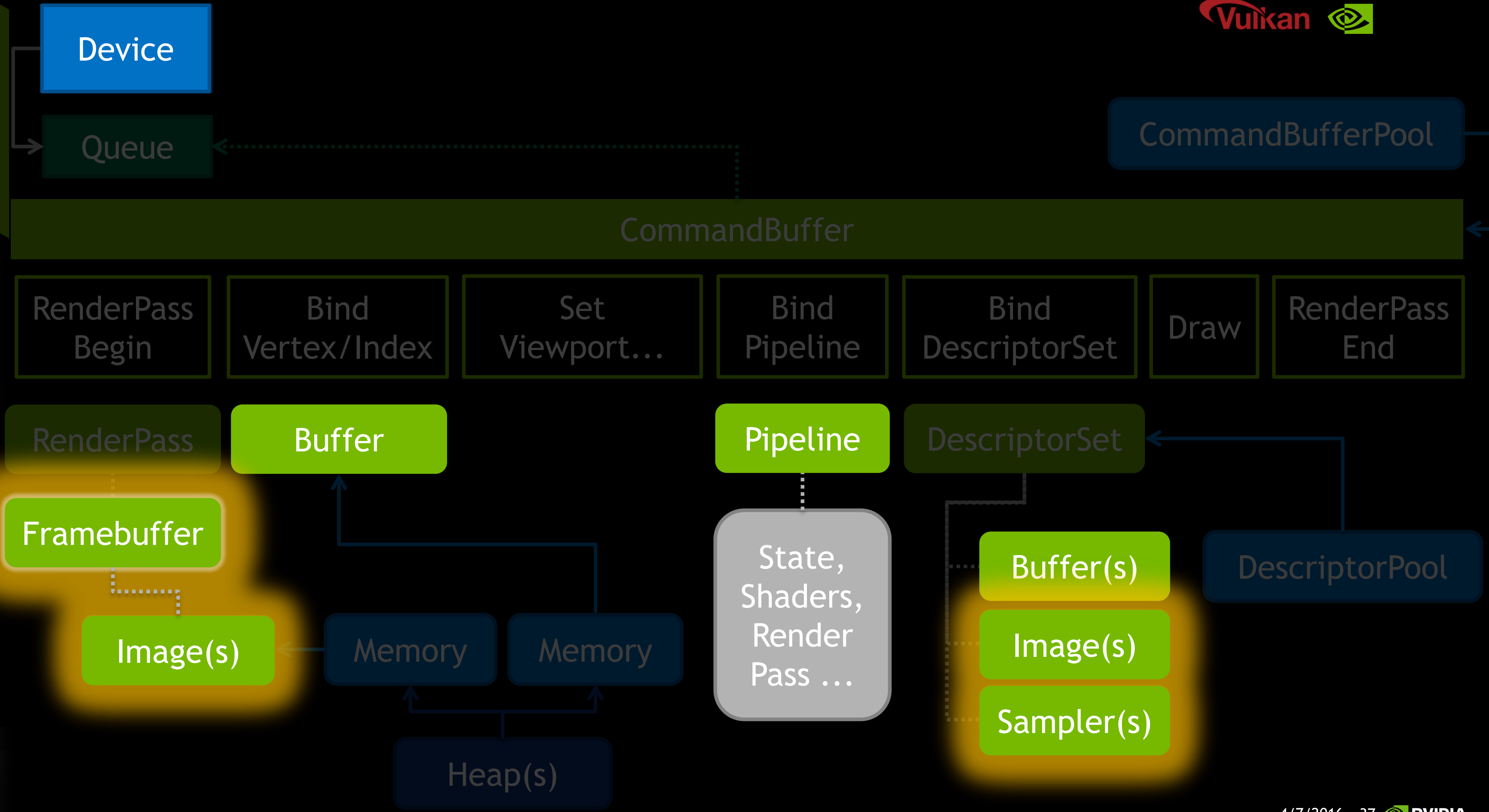
- May or may not be CPU accessible (mappable)
- May or may not be CPU cached
- Buffer Views allow a buffer to be accessed from shaders

More on “where does memory come from” later

Device Local Memory

Host Visible & Coherent

Host Visible, Coherent & Cached


Device
Queue
CommandBufferPool
CommandBuffer
RenderPass Begin
Bind Vertex/Index
Set Viewport...
Bind Pipeline
Bind DescriptorSet
Draw
RenderPass End
RenderPass
Buffer
Pipeline
DescriptorSet
Framebuffer
State, Shaders, Render Pass ...
Buffer(s)
DescriptorPool
Image(s)
Memory
Memory
Image(s)
Sampler(s)
Heap(s)

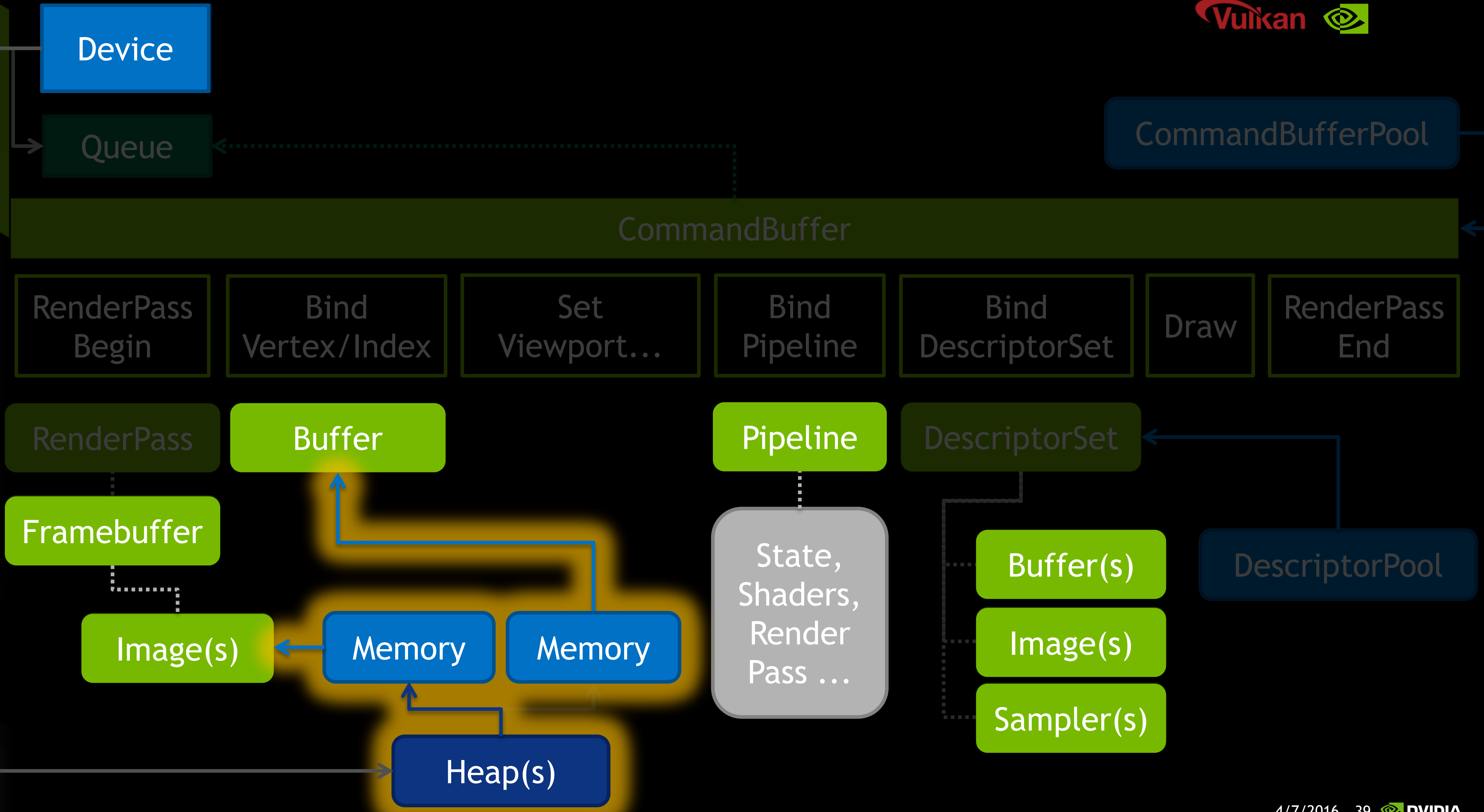
CORE OBJECTS: IMAGES

Represent pixel arrays:

- Textures
- Rendering targets
- Depth targets/textures
- Compute data
- General shader load/store (imgLoadStore)
- Pay careful attention to creation parameters, esp. tiling – big performance implications

Accessed indirectly via Views (and Samplers) to interpret for (re)use:

- Shader read
- Rendertarget, etc


Device
Queue
CommandBufferPool
CommandBuffer
RenderPass Begin
Bind Vertex/Index
Set Viewport…
Bind Pipeline
Bind DescriptorSet
Draw
RenderPass End
RenderPass
Buffer
Pipeline
DescriptorSet
Framebuffer
State, Shaders, Render Pass …
Buffer(s)
DescriptorPool
Image(s)
Memory
Memory
Image(s)
Sampler(s)
Heap(s)

CORE CONCEPTS: BINDING MEMORY TO RESOURCES

HEAP supporting type A,B and flags 1

HEAP supporting B flags 2

Allocate memory from heap

Flags can be “CPU-mappable” for example

Memory Allocation type A

Allocation type B

...

Query resource about size, alignment & type requirements

Assign memory subregion to a resource

Buffer

...

Image

Bind buffer sub-range with offset & size

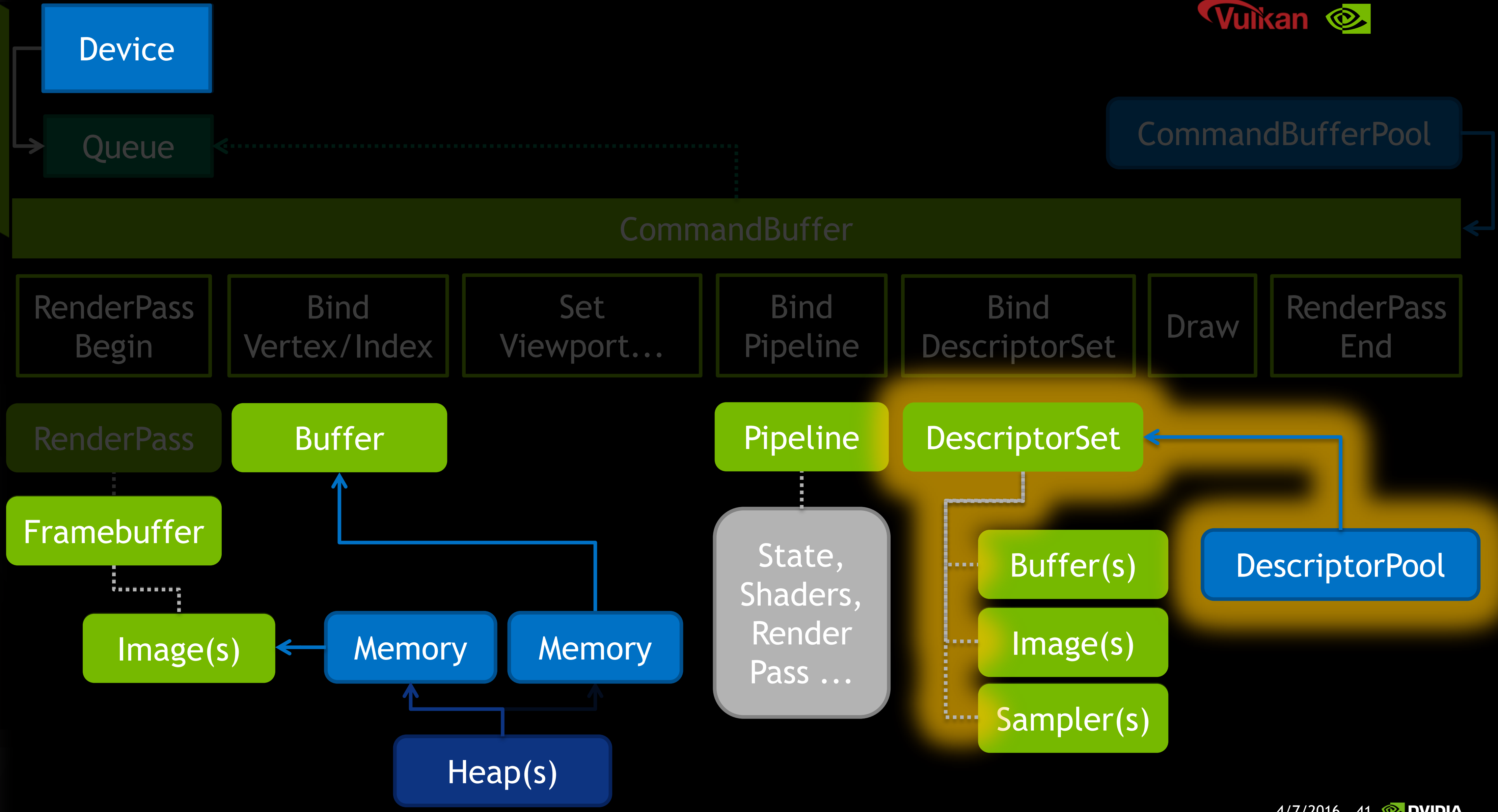
Create view for sub-resource usage (array slice, mipmap...)

Vertex

Uniform

ImageView

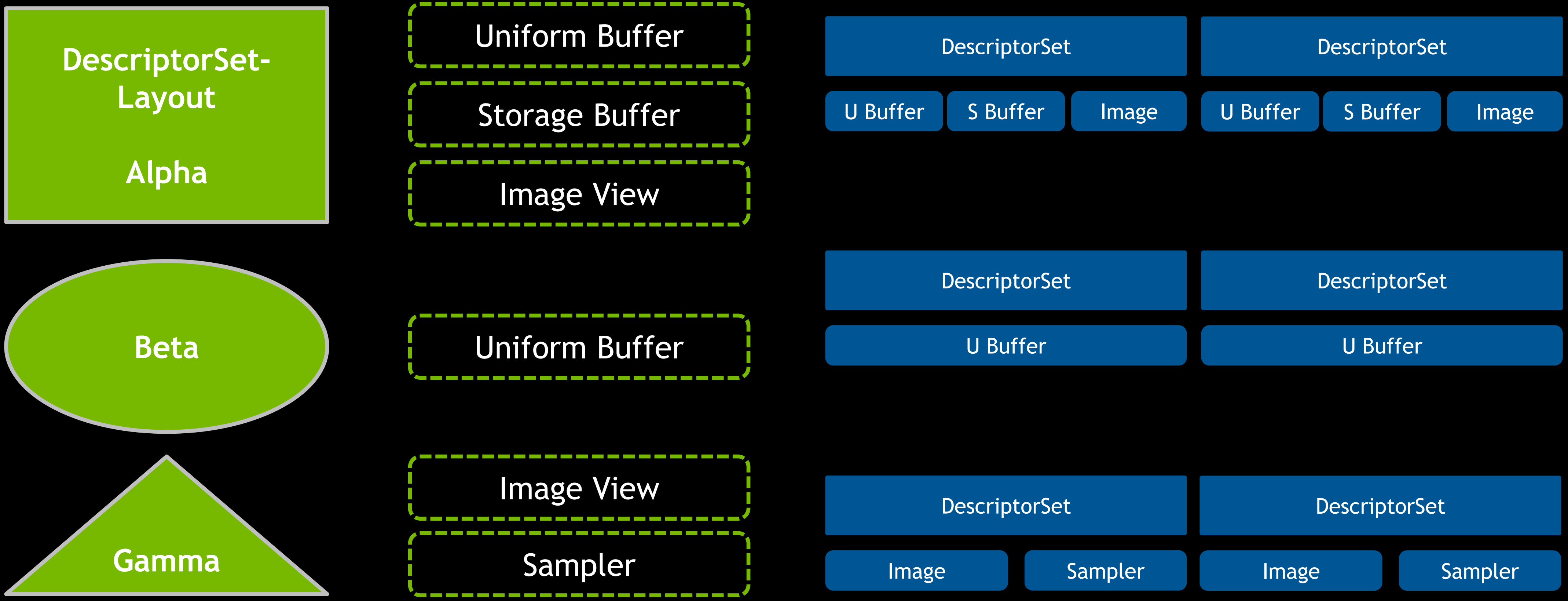
ImageView

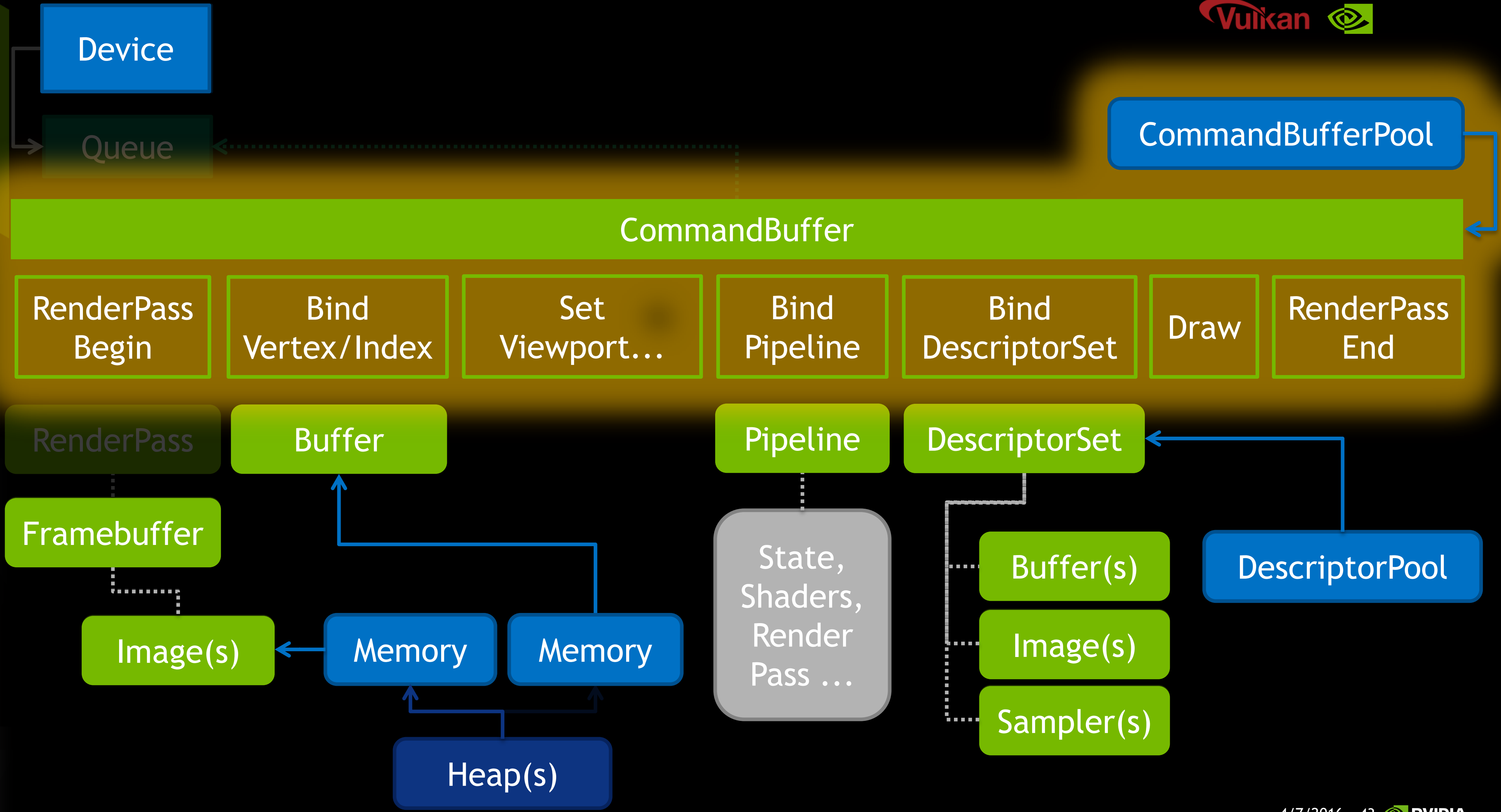

Device
Queue
CommandBufferPool
CommandBuffer
RenderPass Begin
Bind Vertex/Index
Set Viewport...
Bind Pipeline
Bind DescriptorSet
Draw
RenderPass End
RenderPass
Buffer
Pipeline
DescriptorSet
Framebuffer
State, Shaders, Render Pass ...
Buffer(s)
DescriptorPool
Image(s)
Memory
Memory
Image(s)
Sampler(s)
Heap(s)

CORE OBJECTS: DESCRIPTOR SETS AND LAYOUTS

DescriptorSetLayouts define what type of resources are bound within the group

Each **DescriptorSet** holds the references to actual resources




Device
Queue
CommandBufferPool
CommandBuffer
RenderPass Begin
Bind Vertex/Index
Set Viewport...
Bind Pipeline
Bind DescriptorSet
Draw
RenderPass End
RenderPass
Buffer
Pipeline
DescriptorSet
Framebuffer
State, Shaders, Render Pass ...
Buffer(s)
DescriptorPool
Image(s)
Memory
Memory
Image(s)
Sampler(s)
Heap(s)

CORE OBJECTS: COMMAND BUFFERS

All Vulkan rendering is through command buffers

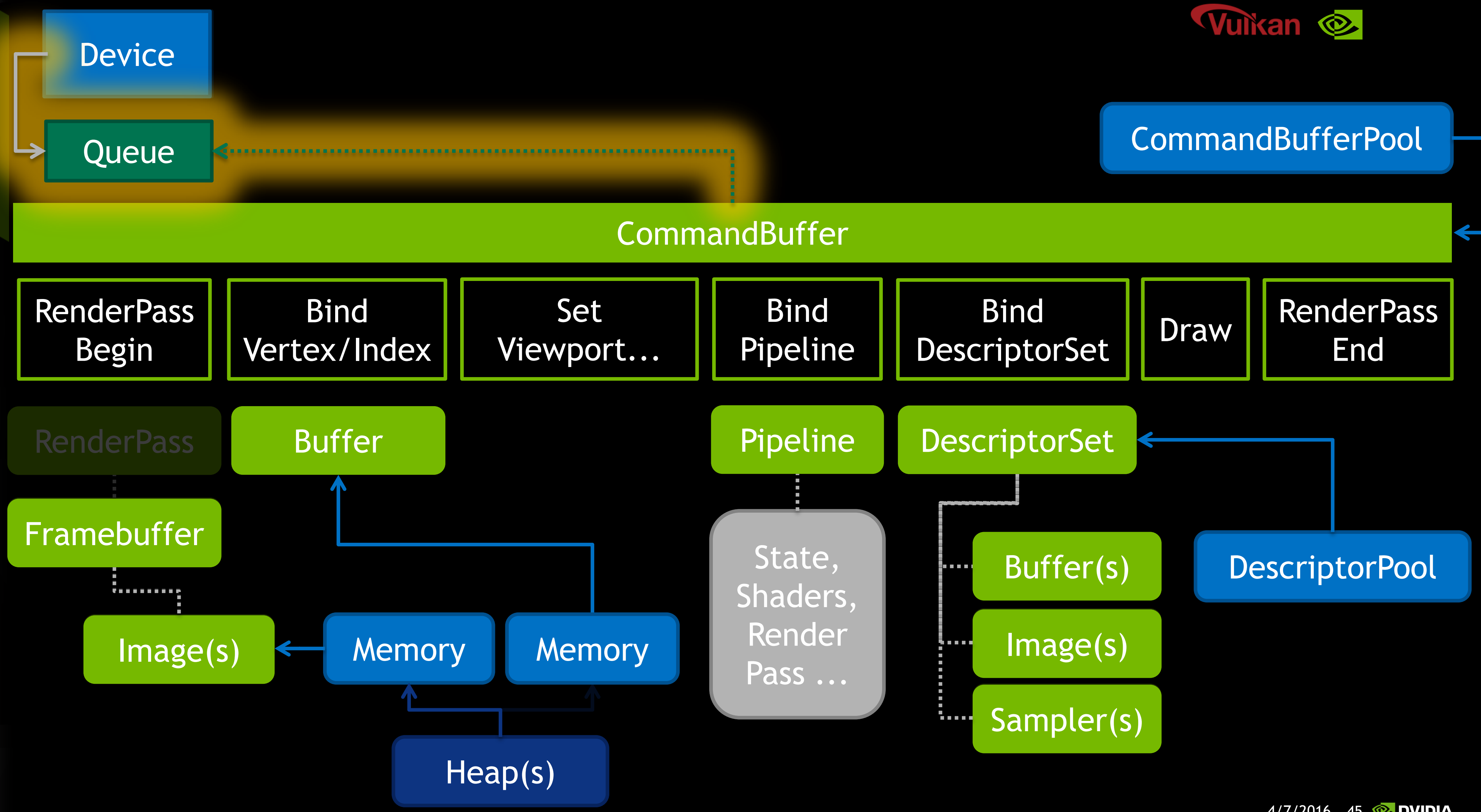
Can be single-use or multi-submission

- Driver can optimize the buffer accordingly

IMPORTANT: No state is inherited across command buffers!

NV_command_lists are similar, and provide a subset of this functionality in GL

- Extension allows GPU-written commands, but is less CPU thread-friendly


Device
Queue
CommandBufferPool
CommandBuffer
RenderPass Begin
Bind Vertex/Index
Set Viewport...
Bind Pipeline
Bind DescriptorSet
Draw
RenderPass End
RenderPass
Buffer
Pipeline
DescriptorSet
Framebuffer
State, Shaders, Render Pass ...
Buffer(s)
DescriptorPool
Image(s)
Memory
Memory
Image(s)
Sampler(s)
Heap(s)

CORE OBJECTS: QUEUES

- Makes explicit the command queue that is implicitly in a context in GL
  - Multiple threads can submit work to a queue (or queues)!
  - No need to “bind a context” in order to submit work
- Queues accept GPU work via CommandBuffer submissions
  - Queues have extremely few operations: in essence, “submit work” and “wait for idle”
- Queue work 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.
    - All forms of work in a single queue
    - One form of work in a queue (e.g. DMA/memory transfer-only queue)

VULKAN PHILOSOPHIES

Not specifically "the" core philosophies of Vulkan; just a few we want to highlight

Take advantage of an **application's high-level knowledge**

- Do not require the driver to determine and optimize for "intent" implicitly

Ensure that the API is **thread-friendly** and explicitly documented for app threading

- Place the synchronization responsibility upon the app to allow higher-level sync

Reduce by **explicit re-use**

- Make explicit as many cases of resource/state/command reuse as possible

VULKAN PHILPSOPHY: EXPLOIT APP KNOWLEDGE

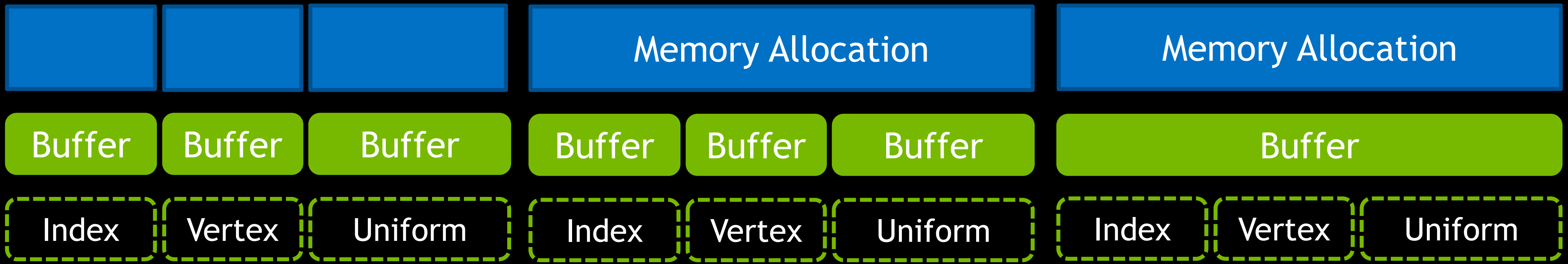
The application has high-level knowledge that the API sees only in pieces

Vulkan seeks to make it possible for the app to use this knowledge

But also requires the app take responsibility for it

- E.g life span of memory allocations is generally known by the app
- An app can usually synchronize threads at a higher level than per driver call
- Apps know what they plan to re-use later

RESOURCE MANAGEMENT



Not. So. Good.

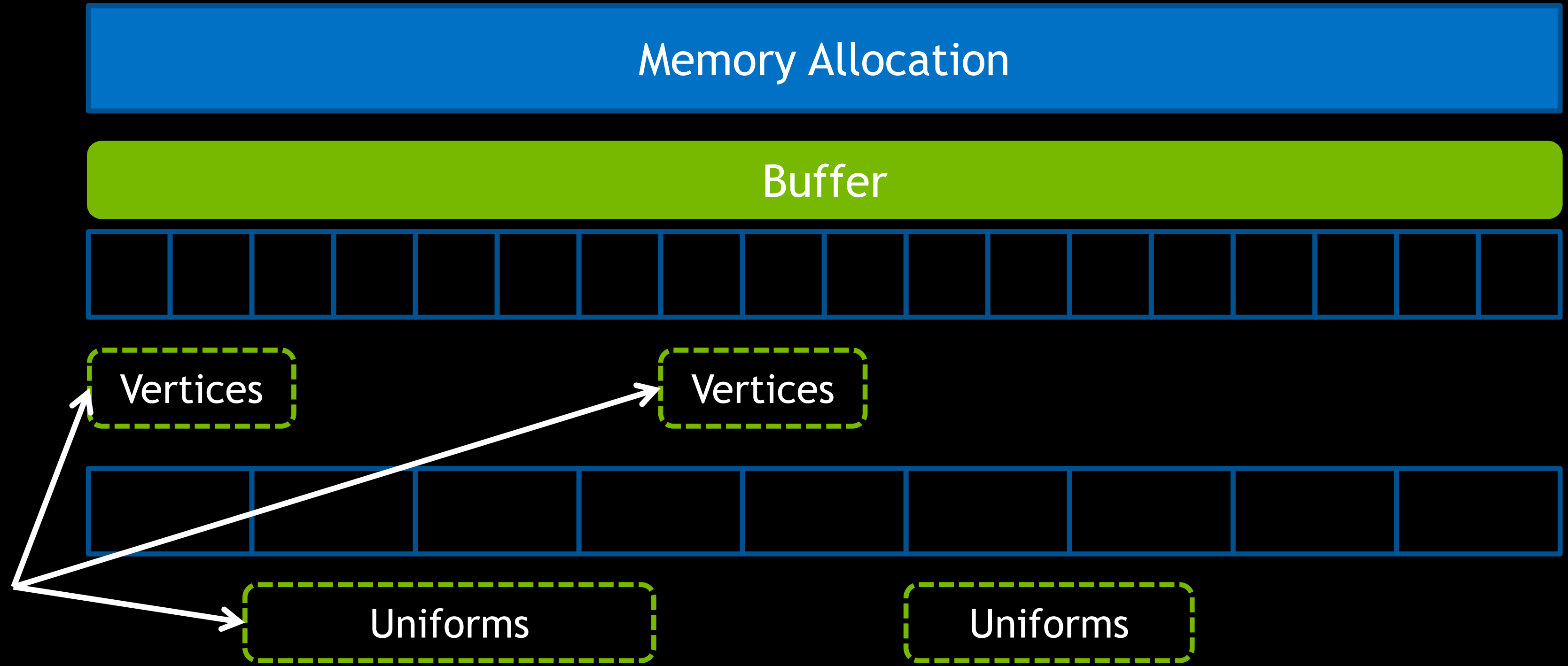
Better…

#HappyGPU

GOOD ALLOCATION AND SUB-ALLOCATION

Buffer offset alignments are binding specific

Same buffer bound, multiple offsets bound



Avoid many buffer objects, use binding offsets for “virtual” buffers

THE BEST SUB-ALLOCATOR: YOU!

The app should know object/resource lifespans best!

App has the overview of all resources

- API only sees in part, in pieces
- Through the small window of the API calls

App also knows the lifespan of resources

- Often no need for a general, complex (and fragmented?) allocator
- Allocations can be stacked in a buffer by lifespan…

Memory Allocation

Whole-app lifespan	Whole-level lifespan	Game-zone lifespan

VULKAN PHILOSOPHY: EXPLICIT THREADABILITY

- Vulkan was created from the ground up to be thread-friendly
  - A huge amount of the spec details the thread-safety and consequences of calls
  - But all of the responsibility falls on the app – which is good!
- Threading at the app level continues to rise in popularity
  - Apps want to generate rendering work from multiple threads
  - Spread validation and submission costs across multiple threads
  - Apps can often handle object/access synchronization at a higher level than a driver

VULKAN AND THREADS

Common threading cases in Vulkan:

- Threaded updates of resources (Buffers)
  - CPU vertex data or instance data animations (e.g. morphing)
  - CPU uniform buffer data updates (e.g. transform updates)
- Threaded rendering / draw calls
  - Generation of command buffers in multiple threads

THREADS: CPU DATA UPDATES

Vulkan exposes multiple methods of updating data from different threads:

Unsynchronized, host visible, mapped buffers

- Coherent buffers, which may be mapped and written without any explicit flushing
- Non-coherent, which may be mapped and written, but must be flushed explicitly

Queue-based DMA transfers

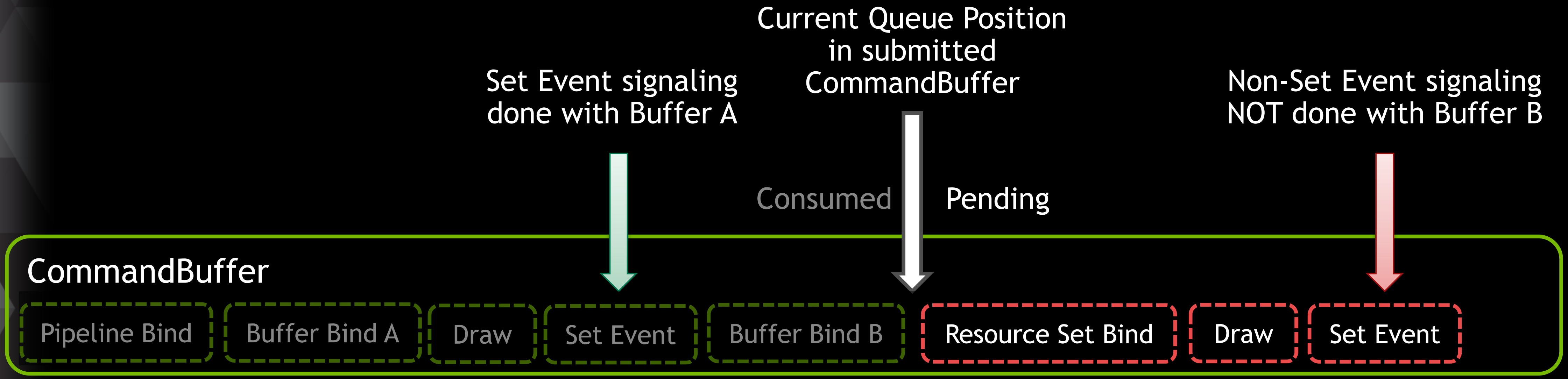
- Host-visible “staging” buffers can be filled as above
- Then data can be transferred to non-host-visible buffers via copy commands
- Which are placed in command buffers and submitted to DMA-supporting queues

THREADED DATA UPDATES: “SAFETY”

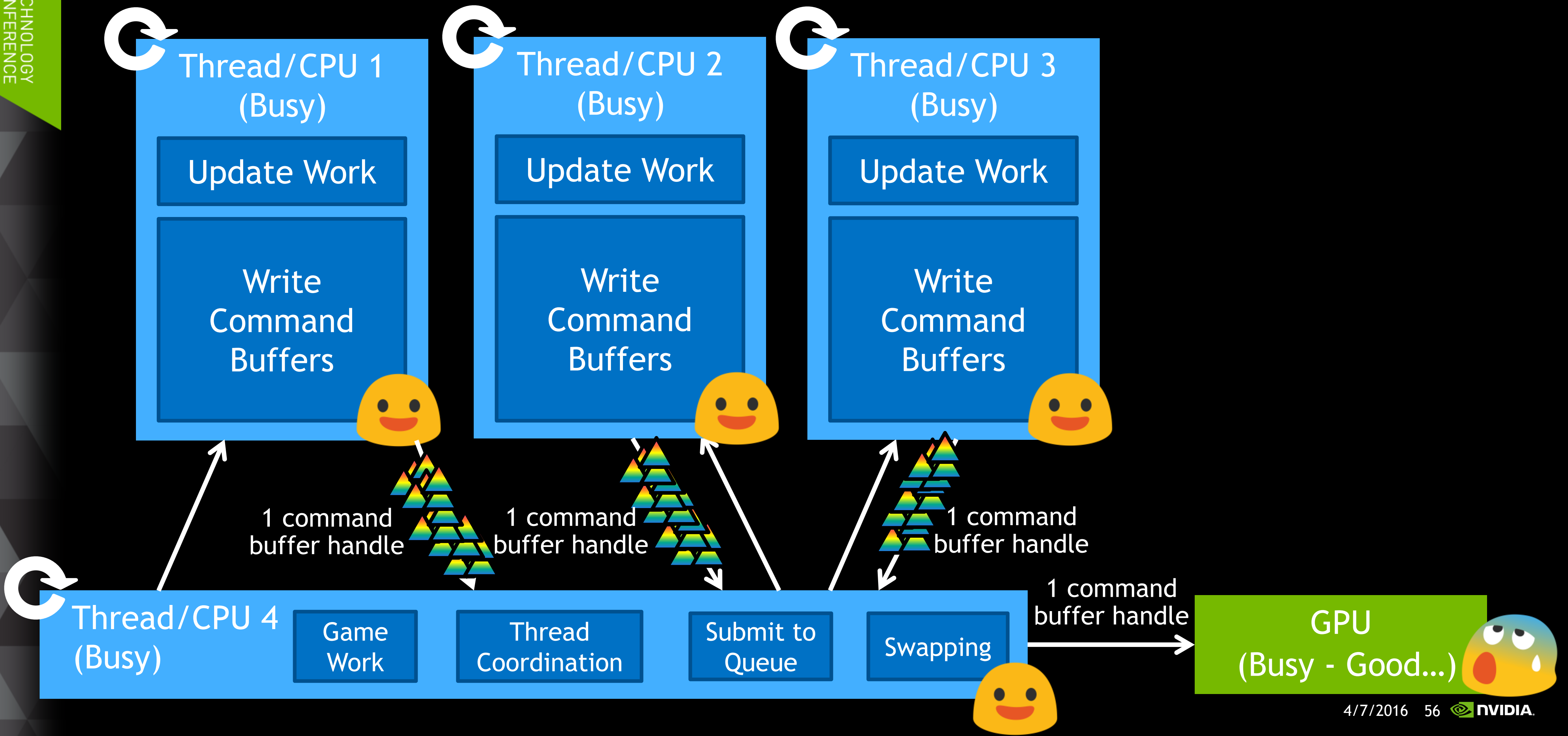
Multiple frames will be in flight; cannot write to a single copy

Really multi-**regioning**; use regions in a single buffer for different frames

VkEvents can be placed in a command buffer after the last use of a copy



THREADED COMMAND BUFFER GENERATION

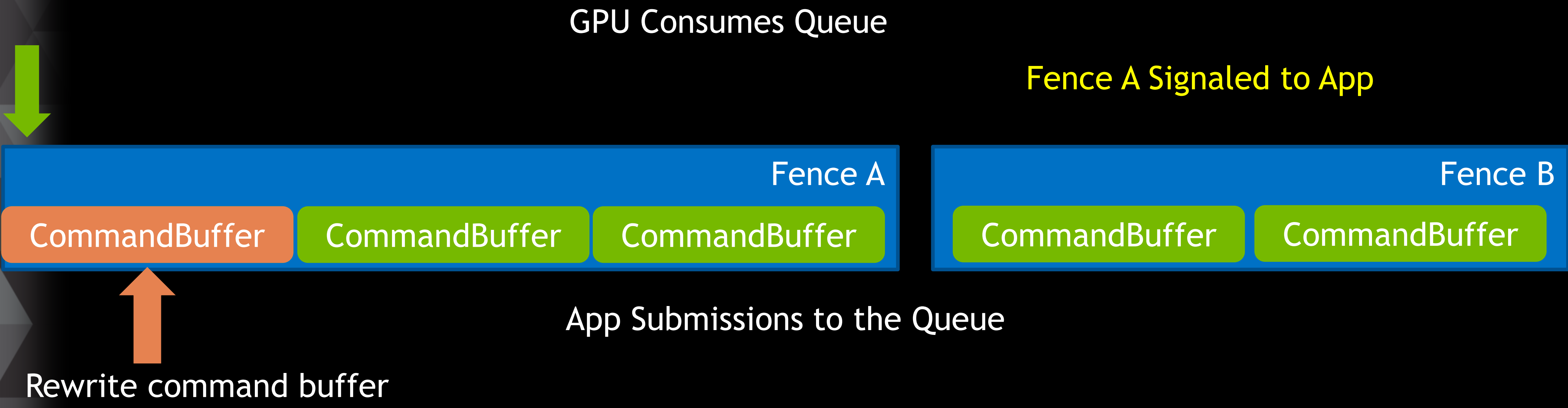


COMMAND BUFFER THREAD SAFETY

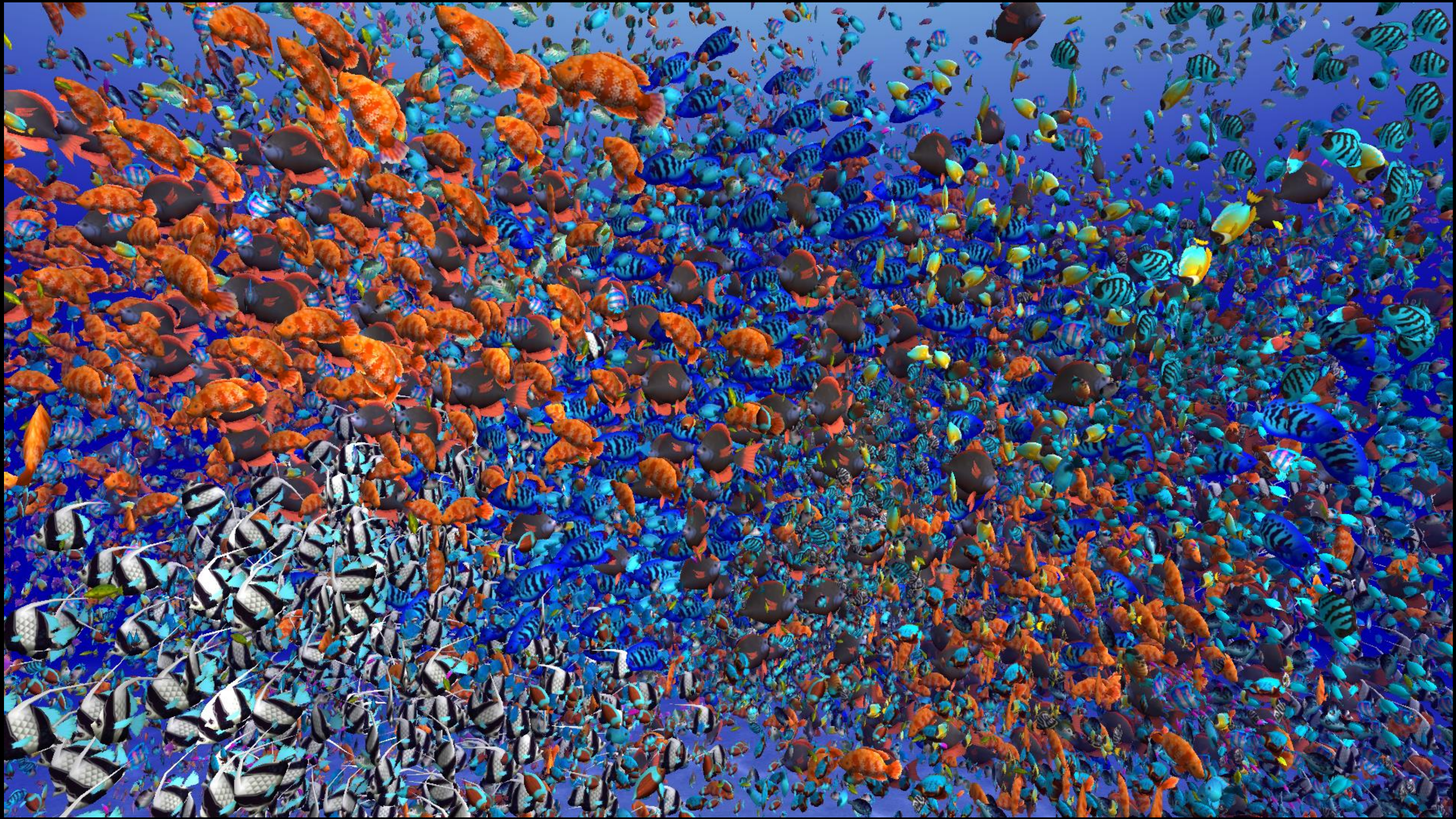
Must not recycle a CommandBuffer for rewriting until it is no longer in flight

But we do not want to flush the queue each frame!

VkFences can be provided with a queue submission to test when a command buffer is ready to be recycled



THREADED RENDERING: FISH!



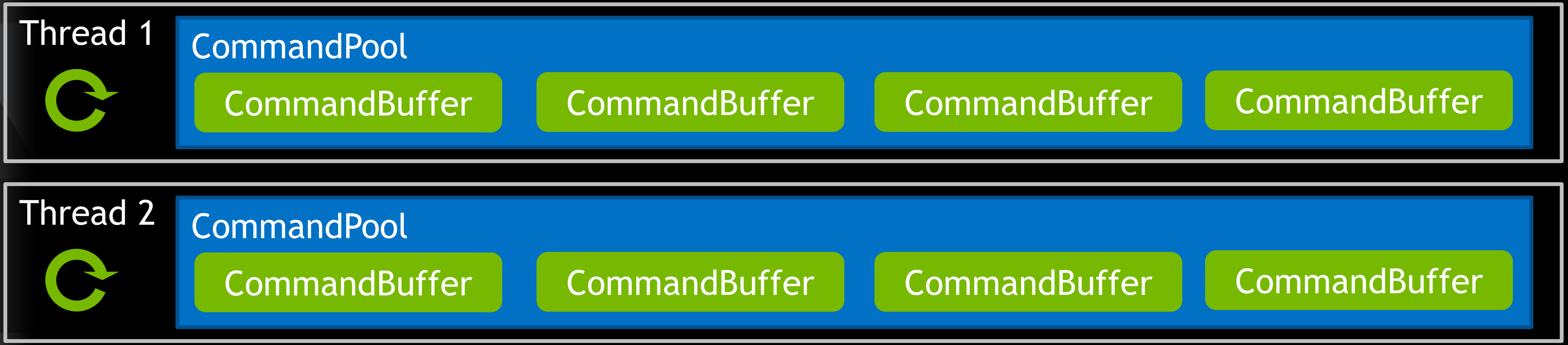
VULKAN THREADS: COMMAND POOLS

VkCommandPool objects are pivotal to threaded command generation

VkCommandBuffers are allocated from a “parent” VkCommandPool

VkCommandBuffers written to in different threads must come from different pools

- Or else the writes must be externally synchronized, which isn’t worth it



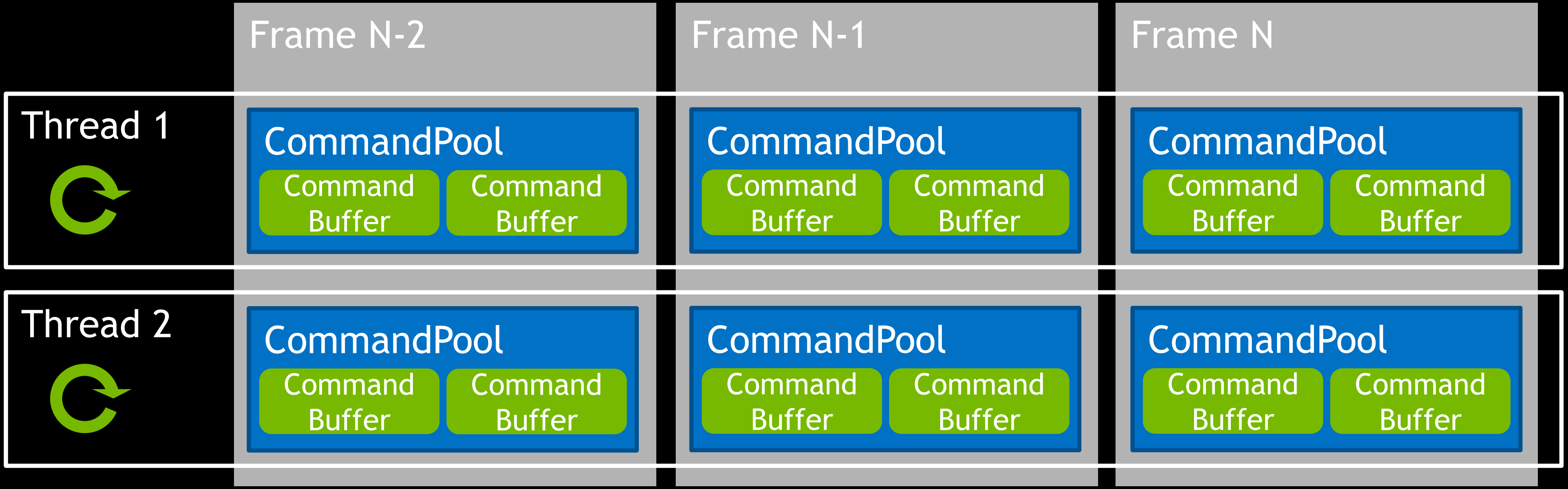
THREADS: COMMAND POOLS

Need to have multiple command buffers per thread

- Cannot reuse a command buffer until it is no longer in flight

And threads may have multiple, independent buffers per frame

Faster to simply reset a pool when that thread/frame is no longer in flight:

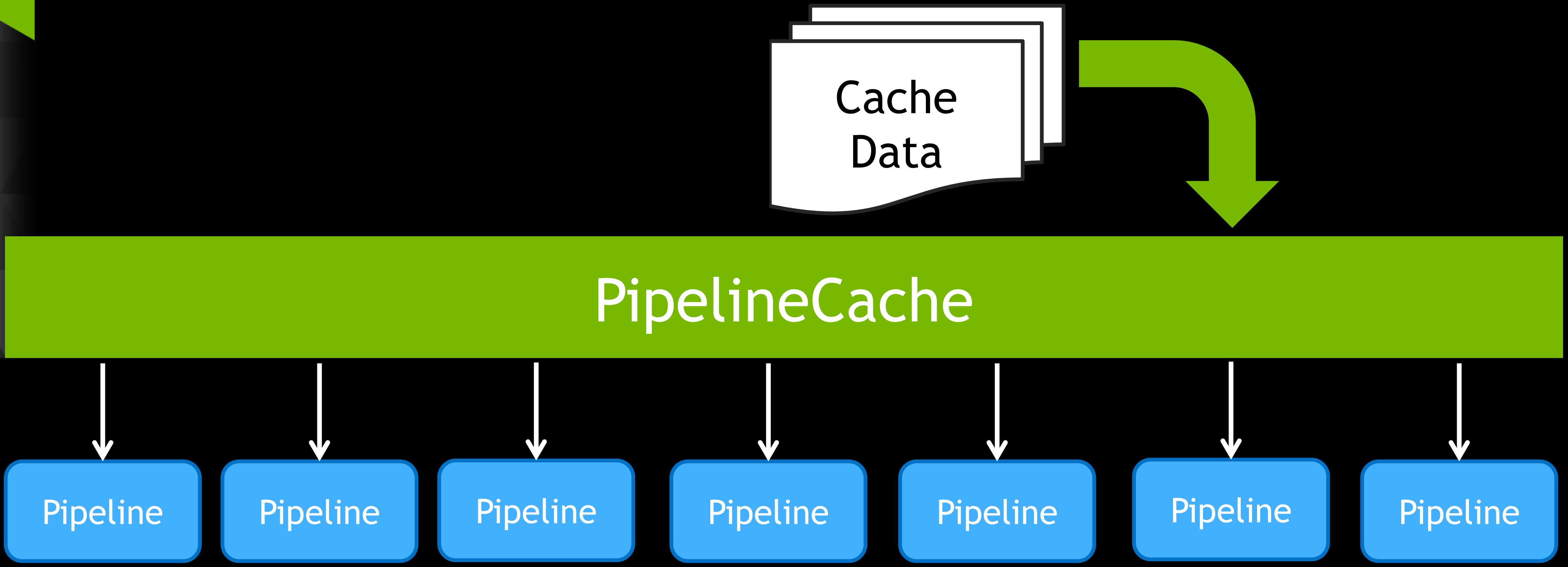


THREADS: DESCRIPTOR POOLS

- VkDescriptorPool objects may be needed for threaded object state generation
  - E.g. dynamically thread-generated rendered objects
- Pools can hold multiple types of VkDescriptorSet
  - E.g. sampler, uniform buffer, etc
  - Max number of each type specified at pool creation
- VkDescriptorSets are allocated from a “parent” VkDescriptorPool
- VkDescriptors written to in different threads must come from different pools

VULKAN PHILOSOPHY: REDUCE BY REUSE

Pipeline Cache objects



GPU TECHNOLOGY CONFERENCE

OVERVIEW: GL, AZDO, AND VULKAN

Issue	Naïve GL	AZDO	NV command list	Vulkan
Deterministic state validation/pre-compilation	no	no	Yes	Yes
Improved single thread performance	no	Yes	Yes	Yes
Multi-threaded work creation	no	partial	partial	yes
Multi-threaded work submission (to driver)	no	no	no	yes
GPU based work creation	no	partial	yes	partial (MDI)
Ability to re-use created work		partial	yes	yes
Multi-threaded resource updates	no	Yes	Yes	Yes
Effort	low	high	Medium-high	Significant rewrite

BENEFICIAL VULKAN SCENARIOS

Has parallelizable CPU-bound graphics work

- Vulkan's CommandBuffer and Queue system make it possible to efficiently spread the CPU rendering workload

Looking to maximize a graphics platform budget

- Direct management of allocations and resources help on limited platforms

Looking for predictable performance, desire to be free of hitching

- Precompilation of state, Pipeline structure avoids runtime shader recompilation and state cache updates

CASES UNLIKELY TO BENEFIT FROM VULKAN

Need for compatibility to pre-Vulkan platforms

Heavily GPU-bound application

Heavily CPU-bound application due to non-graphics work

Single-threaded application, unlikely to change to multithreaded

App targets middleware engine, little-to-no app-level 3D graphics API calls

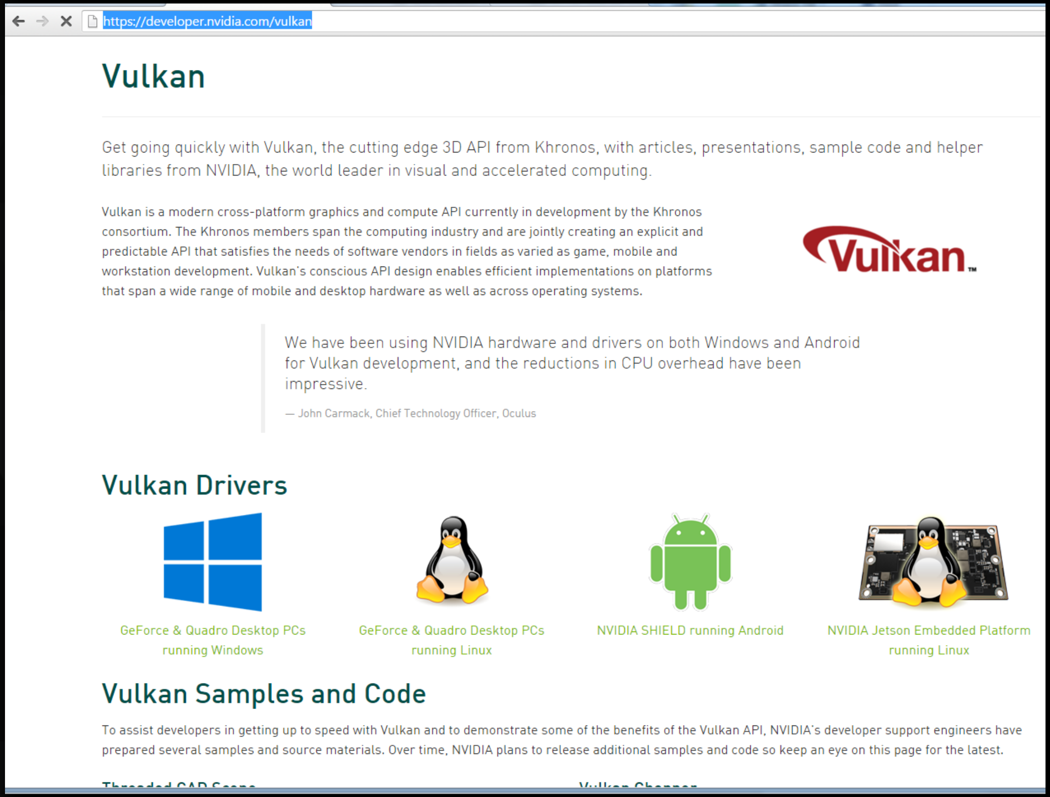
- Consider using an engine targeting Vulkan

App is late in development and cannot risk changing 3D APIs

GPU TECHNOLOGY CONFERENCE

VULKAN RESOURCES

http://developer.nvidia.com/vulkan



https://developer.nvidia.com/vulkan

Vulkan

Get going quickly with Vulkan, the cutting edge 3D API from Khronos, with articles, presentations, sample code and helper libraries from NVIDIA, the world leader in visual and accelerated computing.

Vulkan is a modern cross-platform graphics and compute API currently in development by the Khronos consortium. The Khronos members span the computing industry and are jointly creating an explicit and predictable API that satisfies the needs of software vendors in fields as varied as game, mobile and workstation development. Vulkan's conscious API design enables efficient implementations on platforms that span a wide range of mobile and desktop hardware as well as across operating systems.

> We have been using NVIDIA hardware and drivers on both Windows and Android for Vulkan development, and the reductions in CPU overhead have been impressive.
>
> — John Carmack, Chief Technology Officer, Oculus

Vulkan Drivers

GeForce & Quadro Desktop PCs running Windows

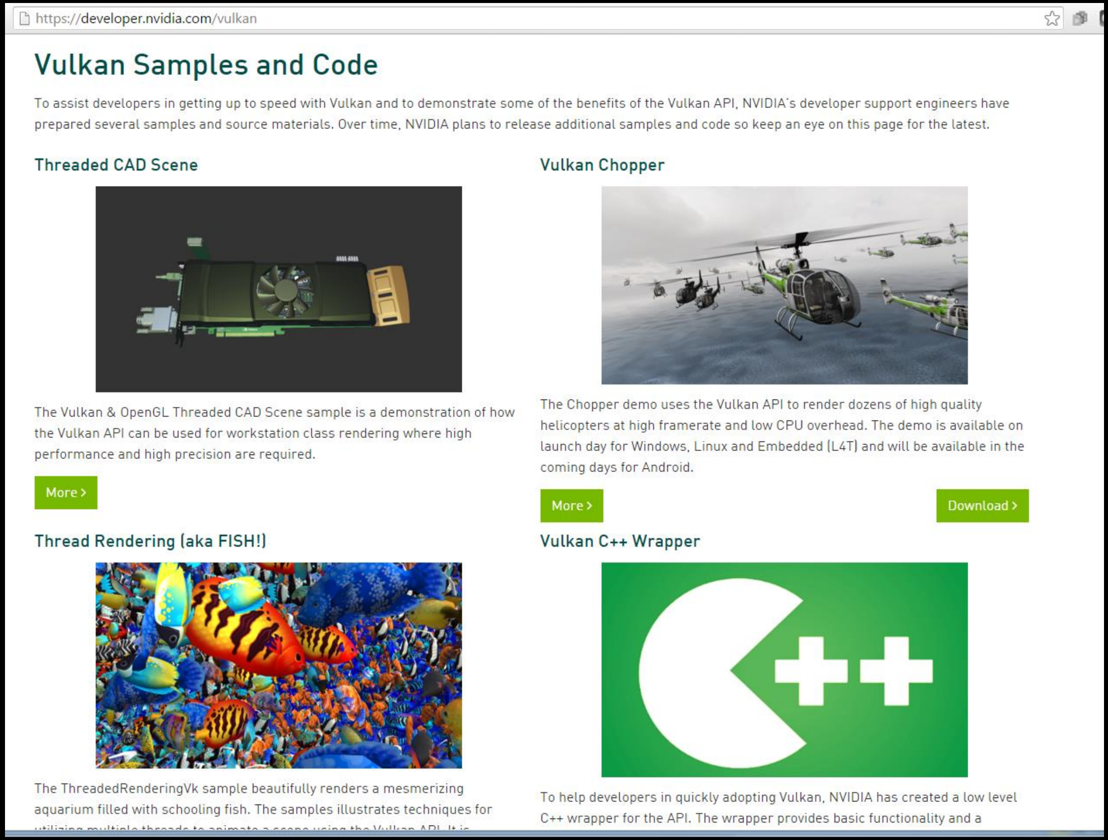
GeForce & Quadro Desktop PCs running Linux

NVIDIA SHIELD running Android

NVIDIA Jetson Embedded Platform running Linux

Vulkan Samples and Code

To assist developers in getting up to speed with Vulkan and to demonstrate some of the benefits of the Vulkan API, NVIDIA's developer support engineers have prepared several samples and source materials. Over time, NVIDIA plans to release additional samples and code so keep an eye on this page for the latest.



https://developer.nvidia.com/vulkan

Vulkan Samples and Code

To assist developers in getting up to speed with Vulkan and to demonstrate some of the benefits of the Vulkan API, NVIDIA's developer support engineers have prepared several samples and source materials. Over time, NVIDIA plans to release additional samples and code so keep an eye on this page for the latest.

Threaded CAD Scene

The Vulkan & OpenGL Threaded CAD Scene sample is a demonstration of how the Vulkan API can be used for workstation class rendering where high performance and high precision are required.

More >

Vulkan Chopper

The Chopper demo uses the Vulkan API to render dozens of high quality helicopters at high framerate and low CPU overhead. The demo is available on launch day for Windows, Linux and Embedded (L4T) and will be available in the coming days for Android.

More > Download >

Thread Rendering (aka FISH!)

The ThreadedRenderingVk sample beautifully renders a mesmerizing aquarium filled with schooling fish. The samples illustrates techniques for

Vulkan C++ Wrapper

To help developers in quickly adopting Vulkan, NVIDIA has created a low level C++ wrapper for the API. The wrapper provides basic functionality and a

GPU TECHNOLOGY CONFERENCE

April 4-7, 2016 | Silicon Valley

THANK YOU

JOIN THE CONVERSATION

#GTC16

JOIN THE NVIDIA DEVELOPER PROGRAM AT developer.nvidia.com/join

PRESENTED BY NVIDIA