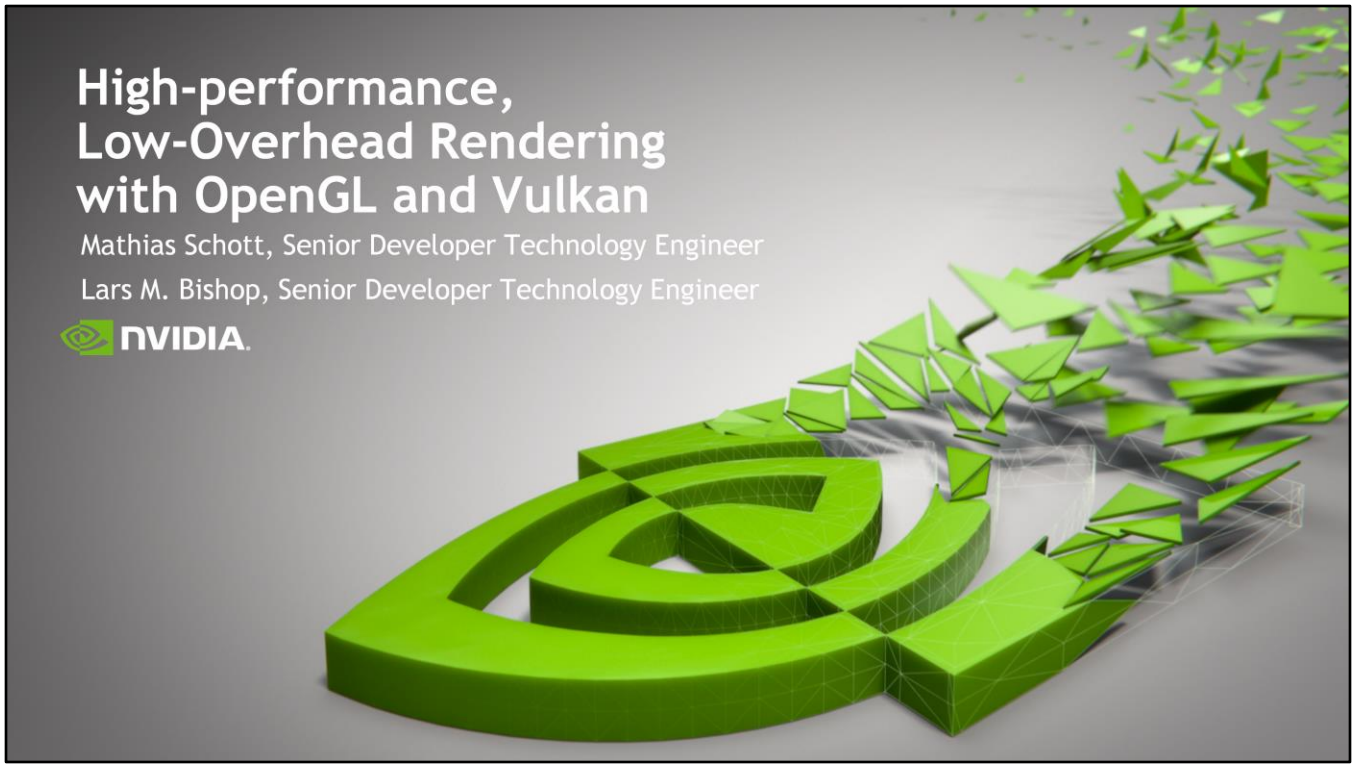
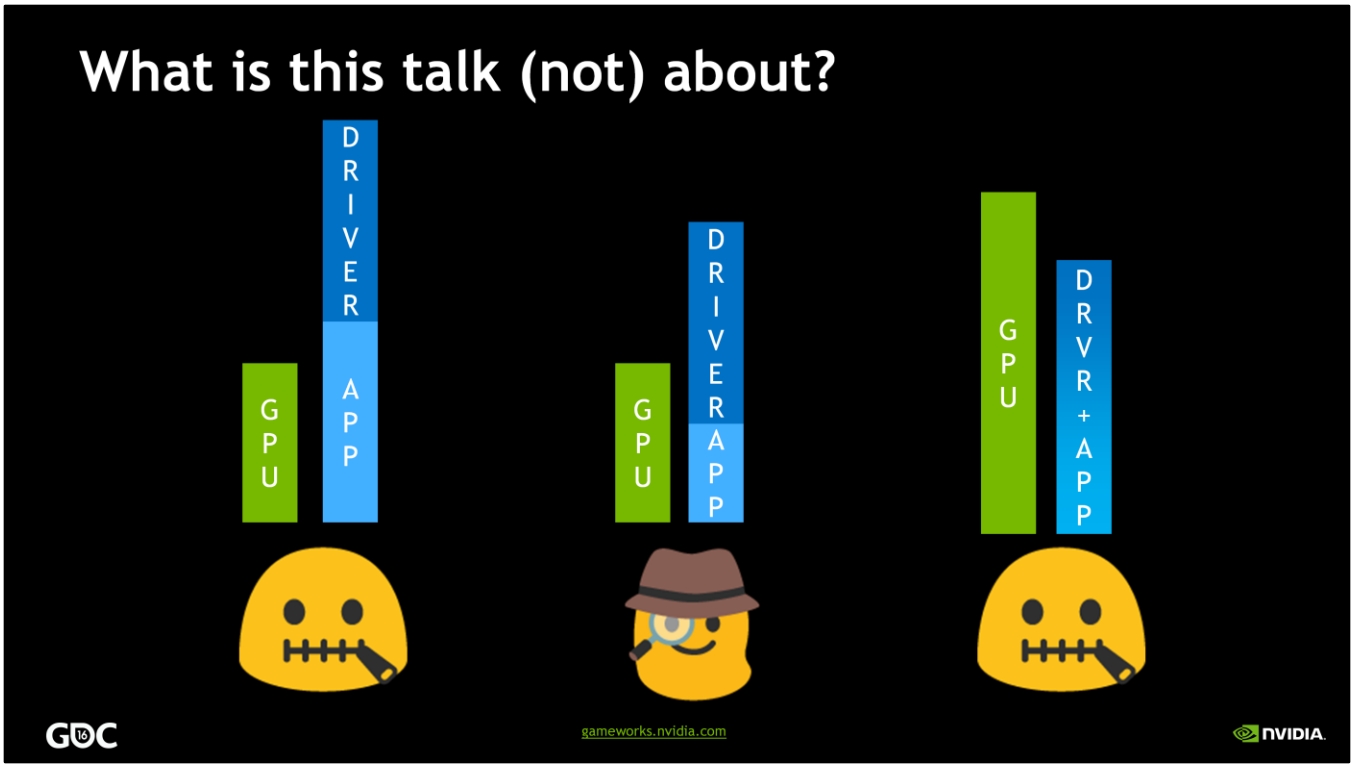


Welcome to our talk about High-performance, Low-Overhead Rendering with OpenGL and Vulkan.

Lars and I are with NVIDIAs developer technology teams. He's been focusing on mobile side things and I've been focusing on the desktop side of things.

But these days that difference seems to be blurry



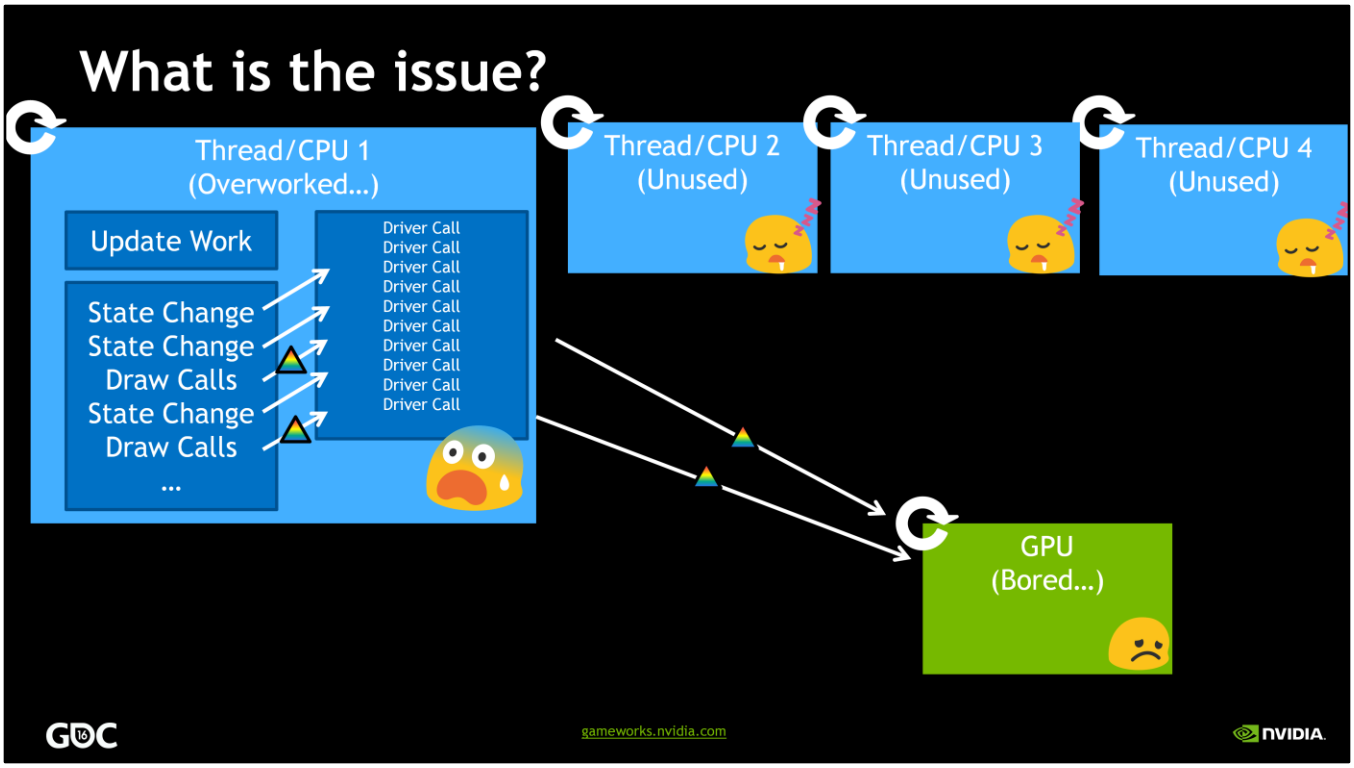
Applications typically fall somewhere on a spectrum between CPU and GPU limit

<CLICK>

We are NOT going to talk about cases where the application itself is CPU limited.
<CLICK>

We are also NOT going to talk about scenarios where the GPU is the liming factor
<CLICK>

However today we are going to investigate cases where the CPU cost of the driver is the limiting factor



Traditional APIs emerged in a single threaded world

One thread on one core traverses scene to collect objects to render

Issues many calls into driver to both change draw state actually render triangles

Saturating that core, thus not feeding the GPU

At the same time other threads are idle

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
       }
    }
  }
}
```


Most applications have a variant of this rendering loop where they iterate over

Sorting of the loops depends on natural frequencies in the scene description

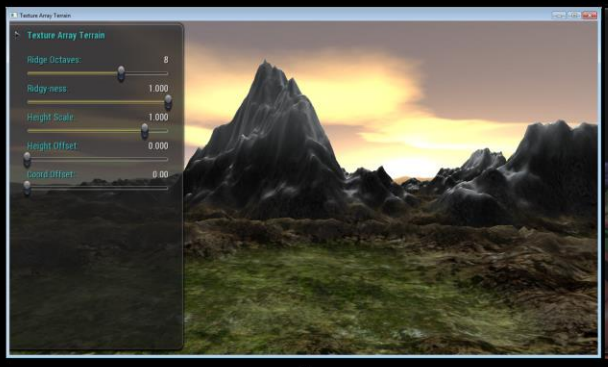
Bottlenecks in Rendering Loop


```
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
       }
    }
  }
}
```


The most frequent hot loop state changes are buffer, texture and shader bindings.

Their total CPU cost can in practice cause overall low visual complexity since it’s too expensive to change

More Triangles Help Increasing Complexity



Tessellation

Instancing

GPU technologies like tessellation are great for some forms of complexity

GPU instancing also helpful

But We Actually Want This

But compelling content is dense and heterogeneous as this example of a recently released game illustrates

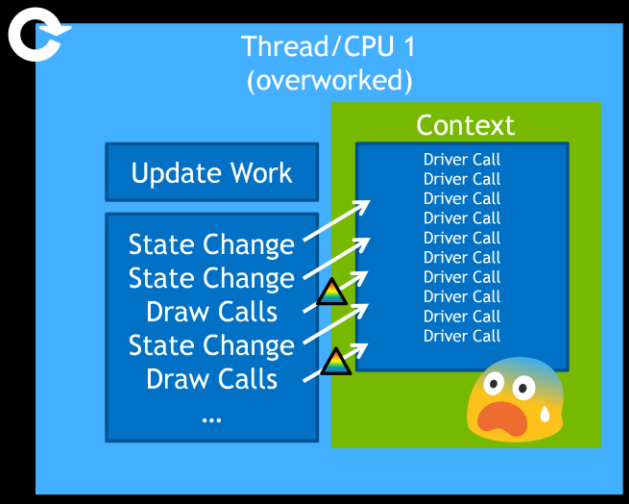
Diverse materials

Lots of independently animated characters with varied geometry

Developers need APIs that can handle complex rendering at a high rates!

Source: http://international.download.nvidia.com/geforce-com/international/images/assassins-creed-unity/assassins-creed-unity-screenshot-007.jpg

Traditional 3D APIs: Use “Heavy” Contexts

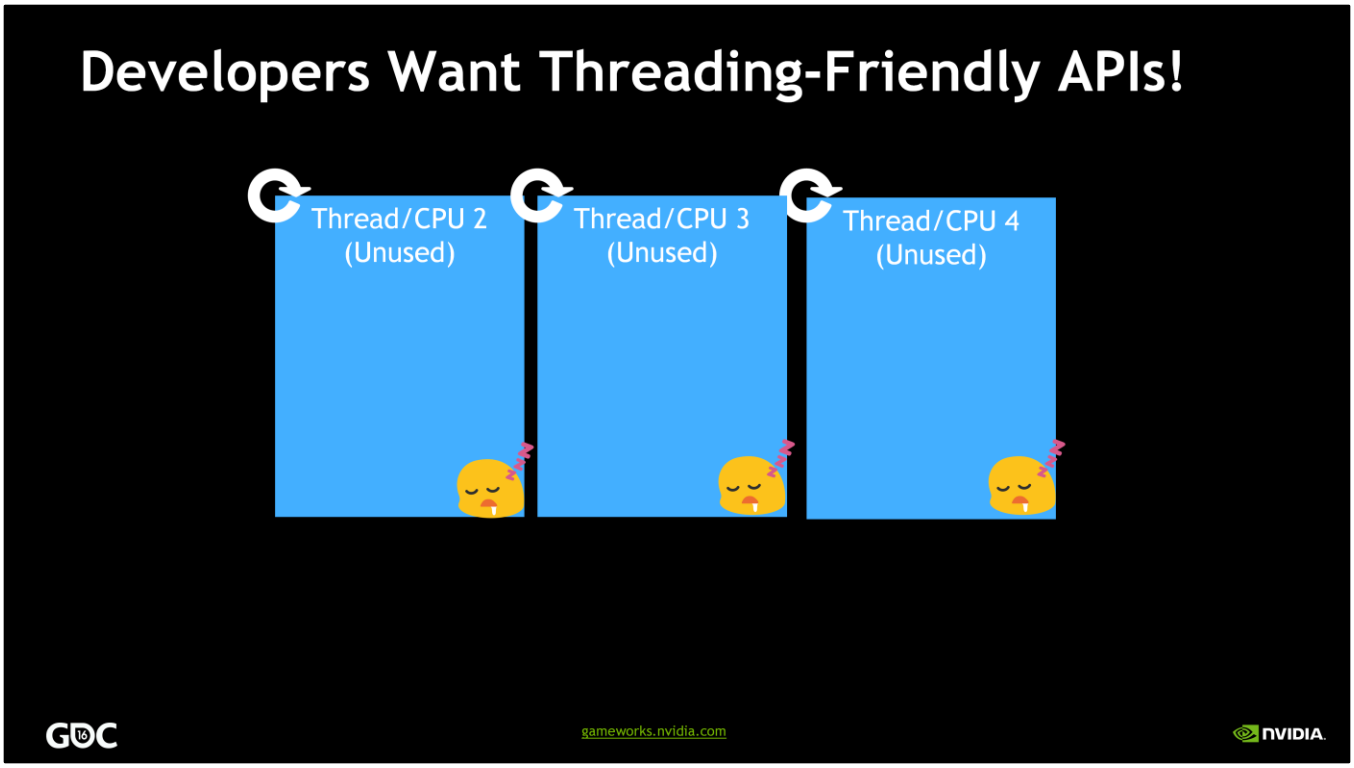


API calls made through a context bound to a thread

Expensive to change context of a thread

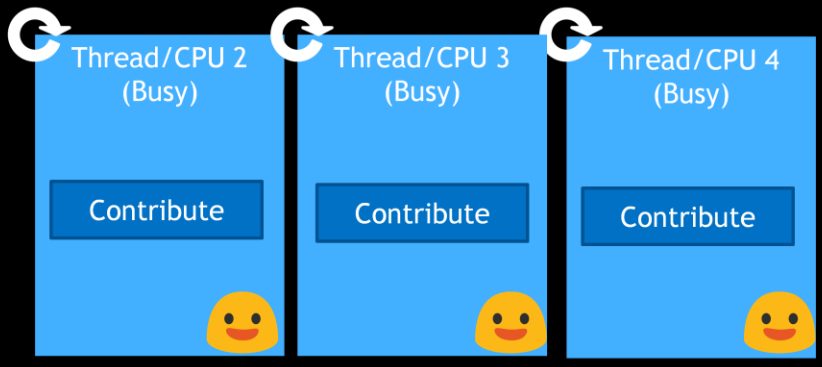
Tricky to share resources efficiently between contexts & threads

Not threading friendly



So all those idle cores do work and join the effort

Developers Want Threading-Friendly APIs!



So all those idle cores do work and join the effort

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




Examples of implicit operations

compiling shaders, downloading textures, downsampling

synchronization, validation & error checking

Unpredictable when and if those happen!

Symptoms such as large stalls on first draw call with a given…

shader, blend mode, vertex data layout, framebuffer attachment formats…

Ask any IHV and you'll likely get a different answer!

Developers want the ability schedule the expensive work explicitly on their schedule!

Updating OpenGL: "AZDO"

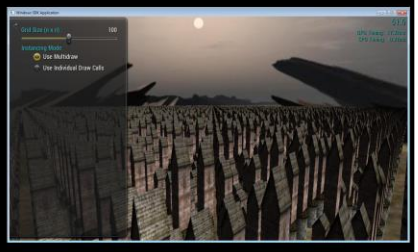
- Popular OpenGL extensions for **A**pproaching **Z**ero **D**river **O**verhead
  - Not a single, monolithic set
  - multiple extensions used for different aspects
- Improved dynamic data update model
  - OpenGL 4.3/GL_ARB_buffer_storage
  - glBufferStorage & glMapBuffer(GL_MAP_PERSISTENT_BIT)

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

Today's "AZDO" Focus

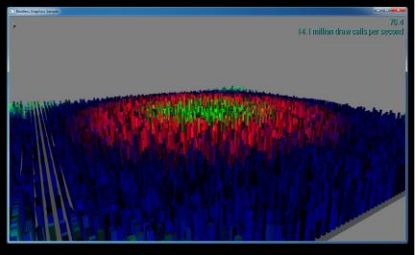
- More varied geometry per drawcall via "MultiDrawIndirect"

  - 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



GDC 16 gameworks.nvidia.com NVIDIA

More varied geometry per drawcall via "MultiDrawIndirect"
allows different shapes per individual drawcall

<click>

More varied materials per draw call via "bindless" resources

different textures, different material parameters

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);
```


glDrawArrays/Elements

- Often called once per batch of geometry
- application side loop across all batchs

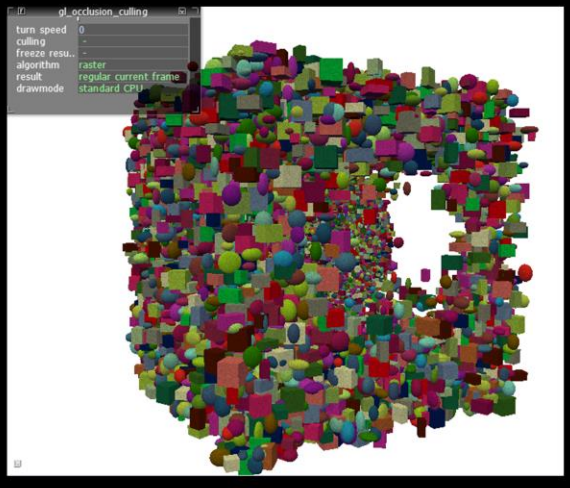
glMultiDrawArrays/Elements

roll loop across batches into a combine batches into single API call to reduce CPU cost

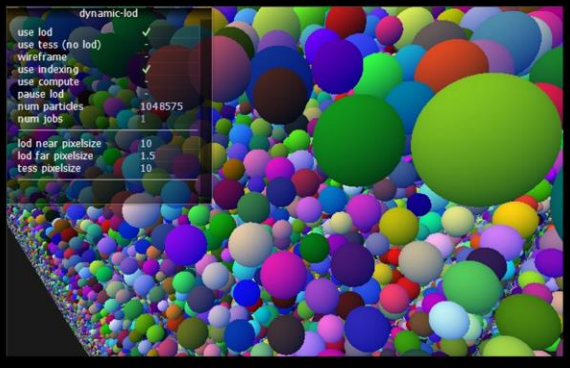
OpenGL 4.3 introduces glMultiDrawArrays/ElementsIndirect

- Source arguments from a buffer
- Transparent memory layout!

Transparent layout of “indirect” buffer…



GPU occlusion culling

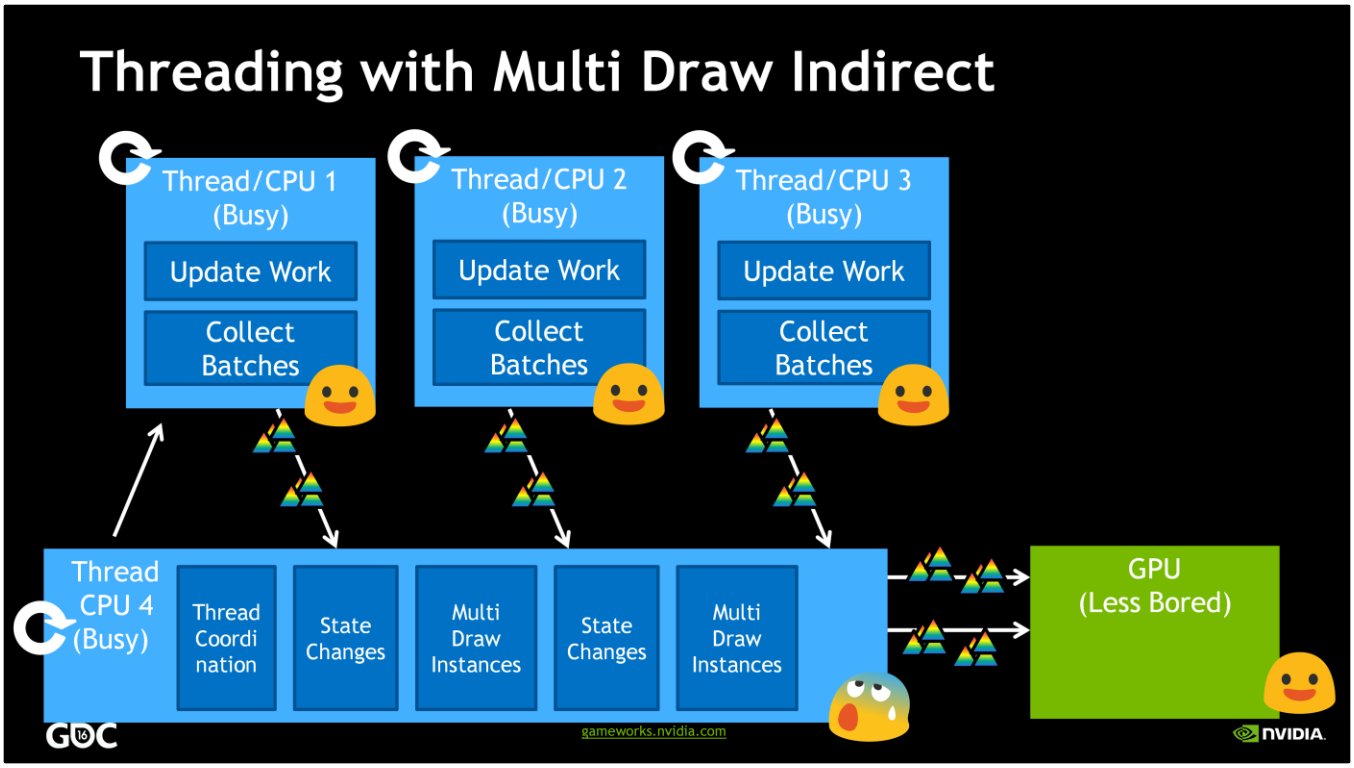


GPU dynamic level of detail

computation of draw call arguments on GPU
Interesting topic on it’s own

More relevant for today’s talk

On CPU multi-threading!



Worker threads traverse scene

- collect data for draw batches
- Either use inter-thread communication to “stitch” together
- Or use persistently mapped buffers

Main thread

executes draw calls and state changes

As a result you have fewer draw calls that render more triangles on the GPU

Multi Draw Indirect Limitations

- Cannot change vertex & index buffer bindings “inline”
  - pack index buffer (IB) and/or vertex buffer (VB)

- Cannot change
  - shaders
  - texture bindings
  - framebuffer object (FBO)
  - uniform buffer object (UBO)

Cannot change vertex & index buffer bindings “inline”

Fetch data from different parts of a (large) index buffer (IB) or vertex buffer (VB)

Cannot change

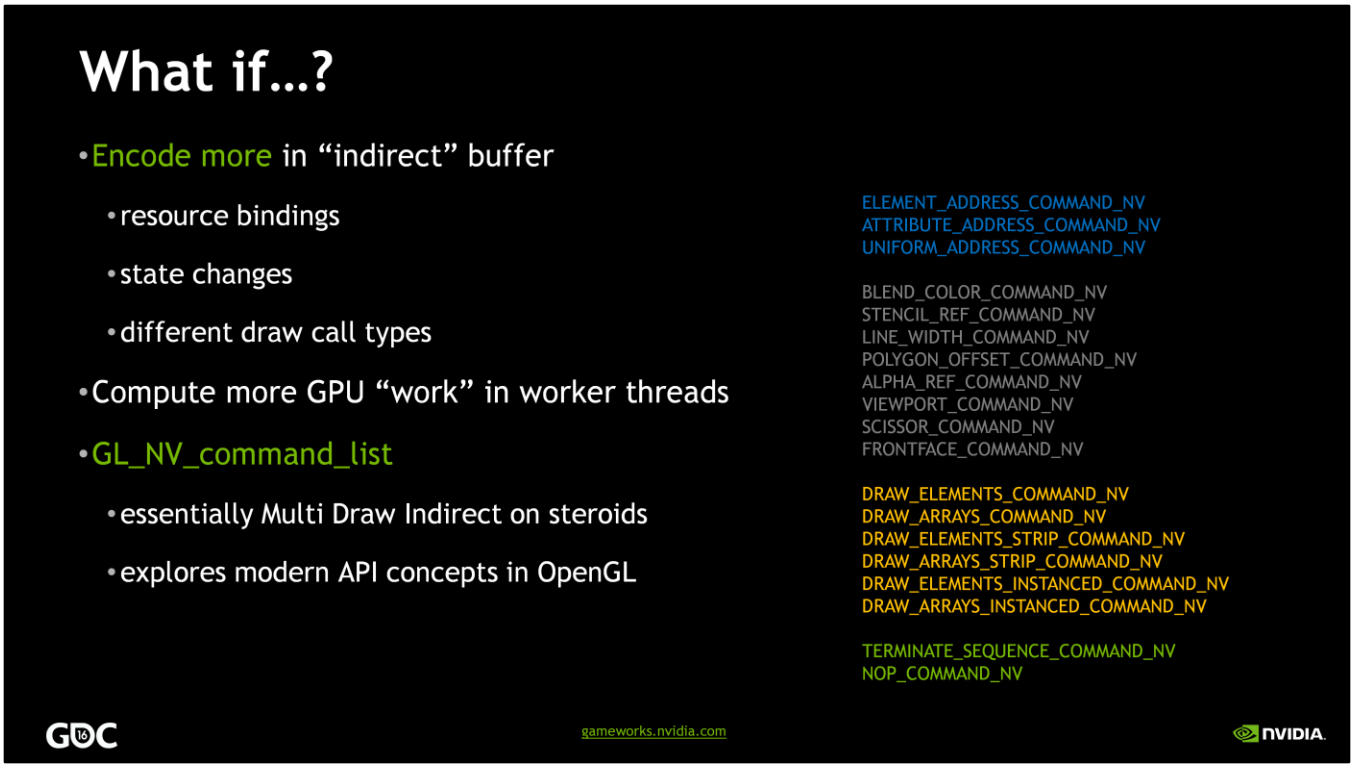
shaders

texture bindings

framebuffer object (FBO)

uniform buffer object (UBO)

Essentially all batches with different geometry get rendered with the same material but can have different geometry



… we could also encode the following in the “indirect” buffer?

- resource bindings
- state changes
- Various drawcall types

Then we could …

- compute more GPU “work” in the worker threads!

GL_NV_command_list does exactly that

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

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...)

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

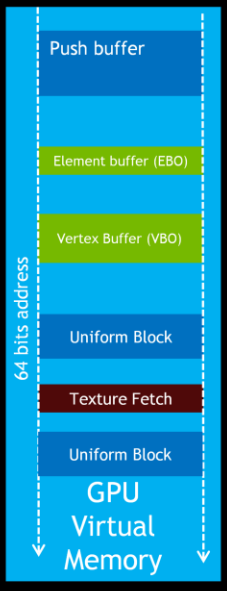
Allows pre-validation of state combinations, later reuse of objects is very fast

Execution either “interpreted” or “baked” via command list object to allow further optimization

Referencing Resources via “Bindless” GPU addresses

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**



•Work from native GPU pointers/handles
•conceptually similar to descriptors in Vulkan
•A lot less CPU work (memory hopping, validation...)
•Less locking when using threaded shared contexts
•Bindless Buffers
•Vertex & Global memory since Tesla Generation (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 );
}
```


a simple example how you could use bindless uniform buffers to update the transformation matrix of some objects to render

<CLICK>first, update the buffer content with the matrices

<CLICK>then make the buffer resident to the GPU in order guarantee that it has a valid GPU address

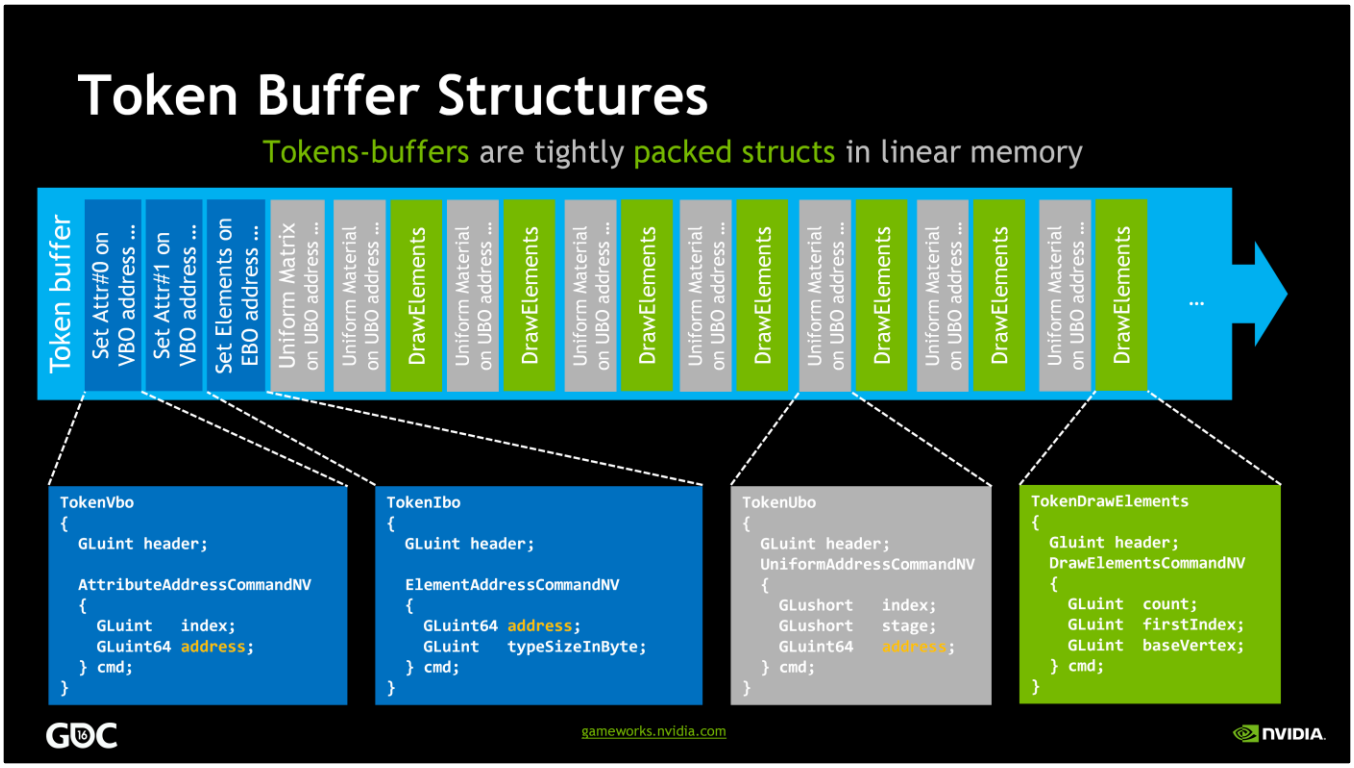
<CLICK>then retrieve the 64 bit GPU address and store it in a variable

<CLICK>enable bindless uniforms

<CLICK>loop over the objects

<CLICK>bind the buffer address instead of the buffer itself

<CLICK> this is what traditional API calls would look like



lets see how we can use bindless resource bindings in our token buffer which contains various structs tightly packed in memory

<CLICK>For example there is a token to change a uniform buffer binding

<CLICK>but here are also tokens for vertex data and uniform data

highlighted in yellow are changes to buffer bindings

<CLICK>last but not least, there are tokes for drawcalls similar to those in Multi Draw Indirect

D3D12 ExecuteIndirect quite similar to this

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

single API call to encapsulates majority of state + primitive type

immutable for more control over validation cost

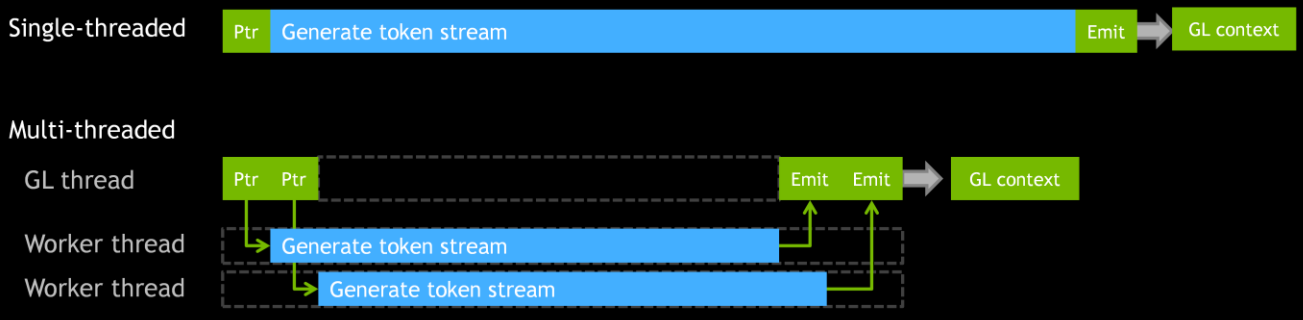
Resource bindings are not captured

bindless used instead

textures passed via UBO since there is no token for this

Threading and Command Lists

- Fill token buffers if reuse impossible



•single threaded case straightforward

get pointer, fill token buffer, emit to context to execute work

•<CLICK>Multi threaded case

•GL thread with context, for example 2 worker threads without a GL context

•<CLICK> Pass token buffer pointers to worker threads

•<CLICK>worker threads generate token stream

•<CLICK>emit on GL thread to context to execute work

•<CLICK> Handle state object captures in GL thread since they require a GL context which is one of the limitations

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

generally speaking command-List does NOT pretend to solve general OpenGL multi-threading

but allows partial multi threading possible, e.g the threaded token buffer generation

but state validation still single threaded since the state capture

State Object Capture must be handled in OpenGL context

but worker threads actually “know” the associated state for specific render workload

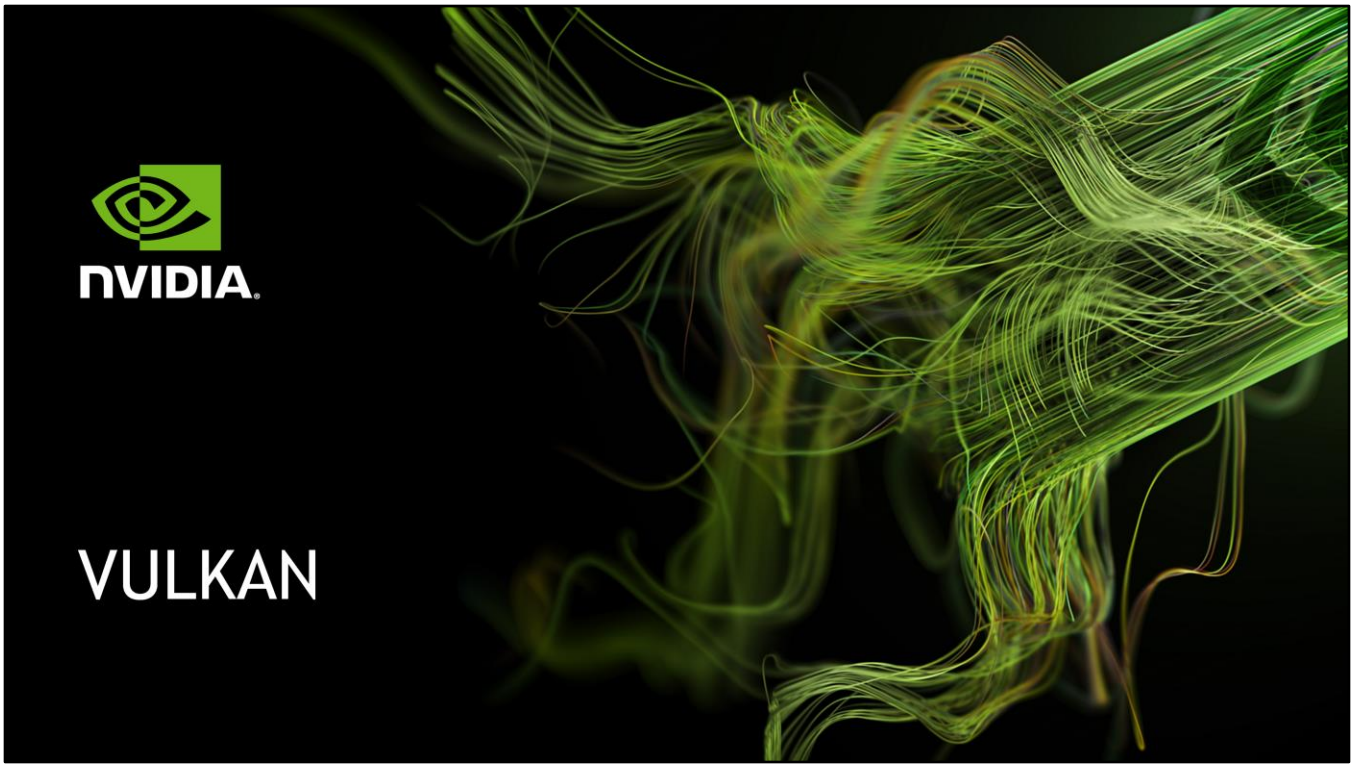
possible but tricky to have worker threads exchange data with GL thread who then does the state capture

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

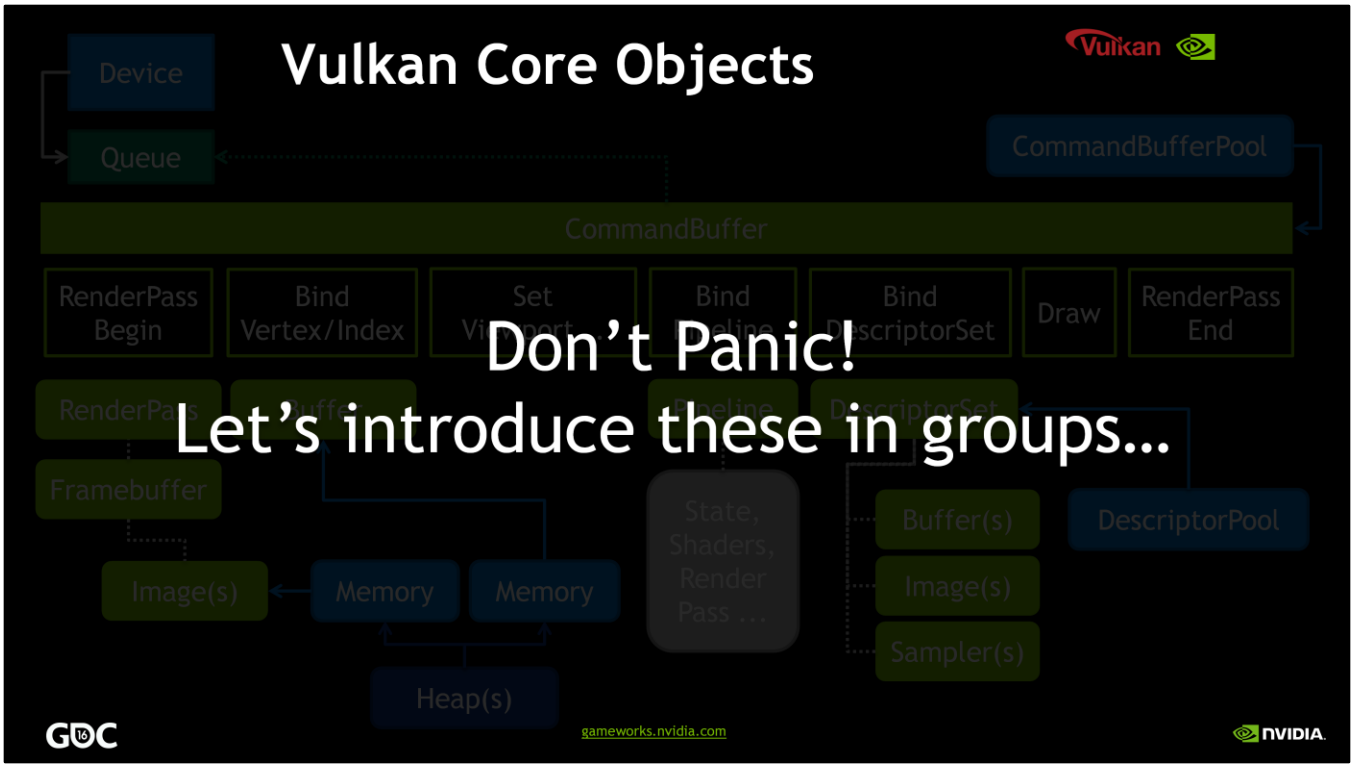


Good afternoon – I'm Lars Bishop, and like Mathias, I'm a developer technologies engineer at NVIDIA. I tend to focus on our SHIELD platforms. I'm here to provide a bit of an overview of Vulkan and how it fits into this continuum. Given the tech press coverage and internet discussions leading up to GDC, we likely don't need to say much more by way of introduction than the fact that Vulkan is a new, open, cross-platform 3D API that was launched by the Khronos Group in February. NVIDIA has been very active in the development of Vulkan, and we have drivers available today on Windows, Linux Desktop, Android and Linux for Tegra.

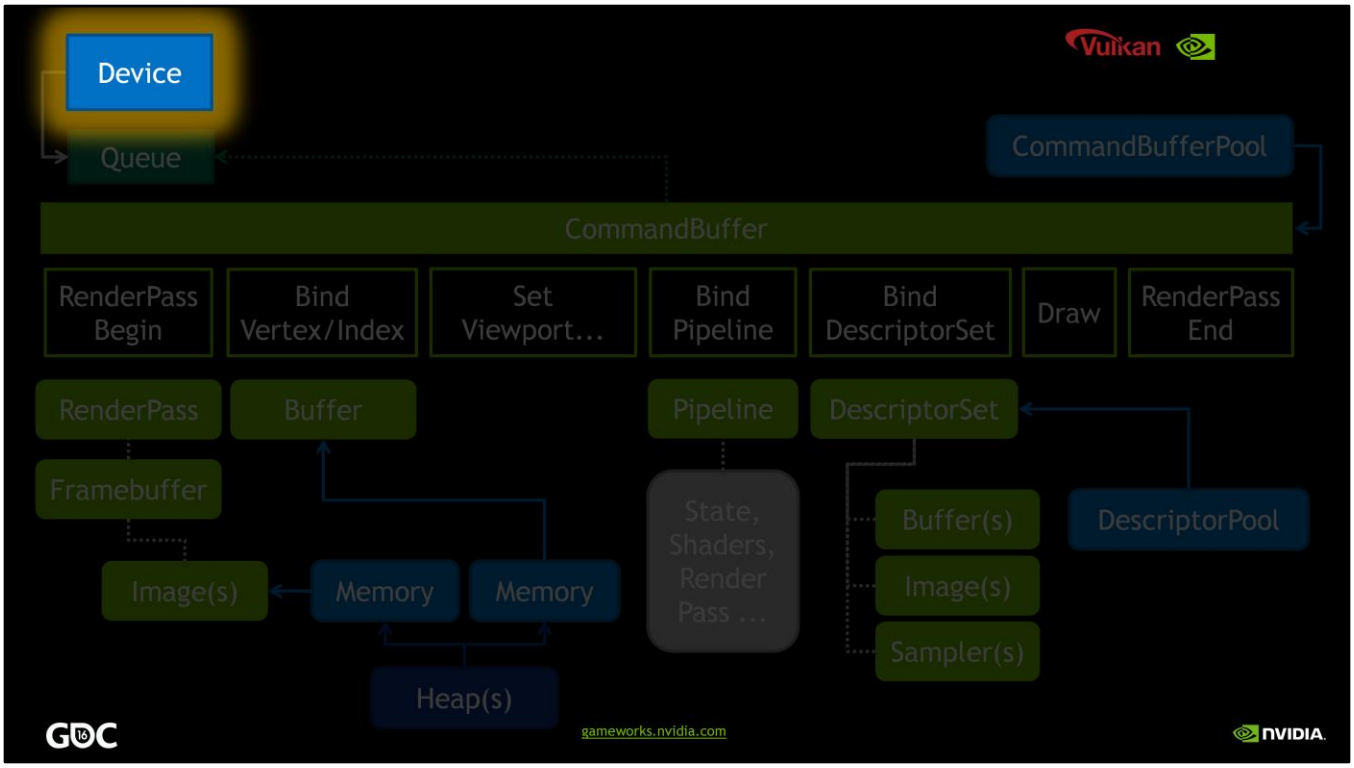
Vulkan Philosophies

- ***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

So let's start with some philosophies around Vulkan, and you'll see some themes that we've already discussed. Vulkan was designed to take most of these themes further. First, empower, rather than insulate the app developer. Ensure that the app can do as much as is reasonable from multiple threads. The driver should not try to synchronize everything internally. And then, reduce repeated work in the driver via explicit reuse of rendering work in the app. Don't make the driver try to find the reuse internally.



Okay – so here is Vulkan in terms of the objects and major relations. Got it? Good! Okay, in closing. <click> just kidding; let’s introduce these objects a few at a time and build things up.



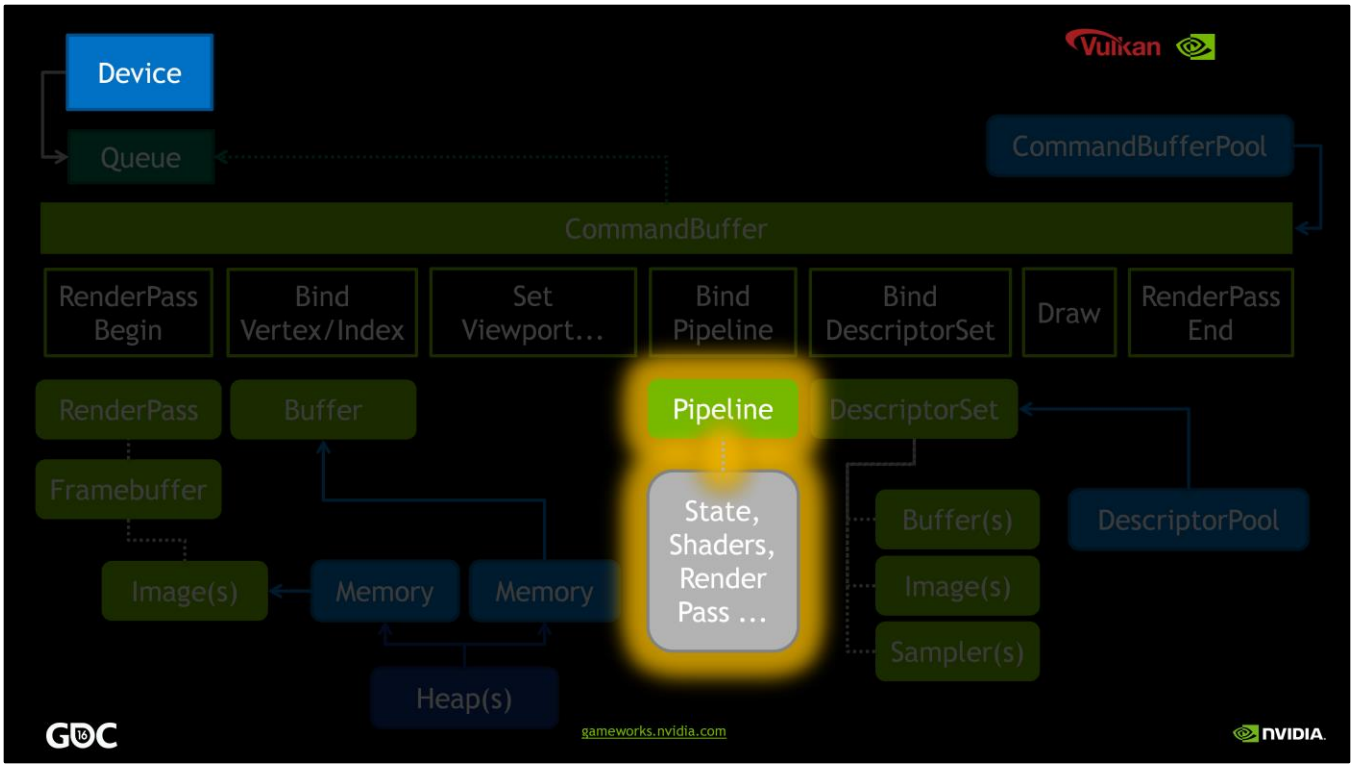
First, we have `the device object <click>. It's mainly a provider of resources.

Core Objects: Devices

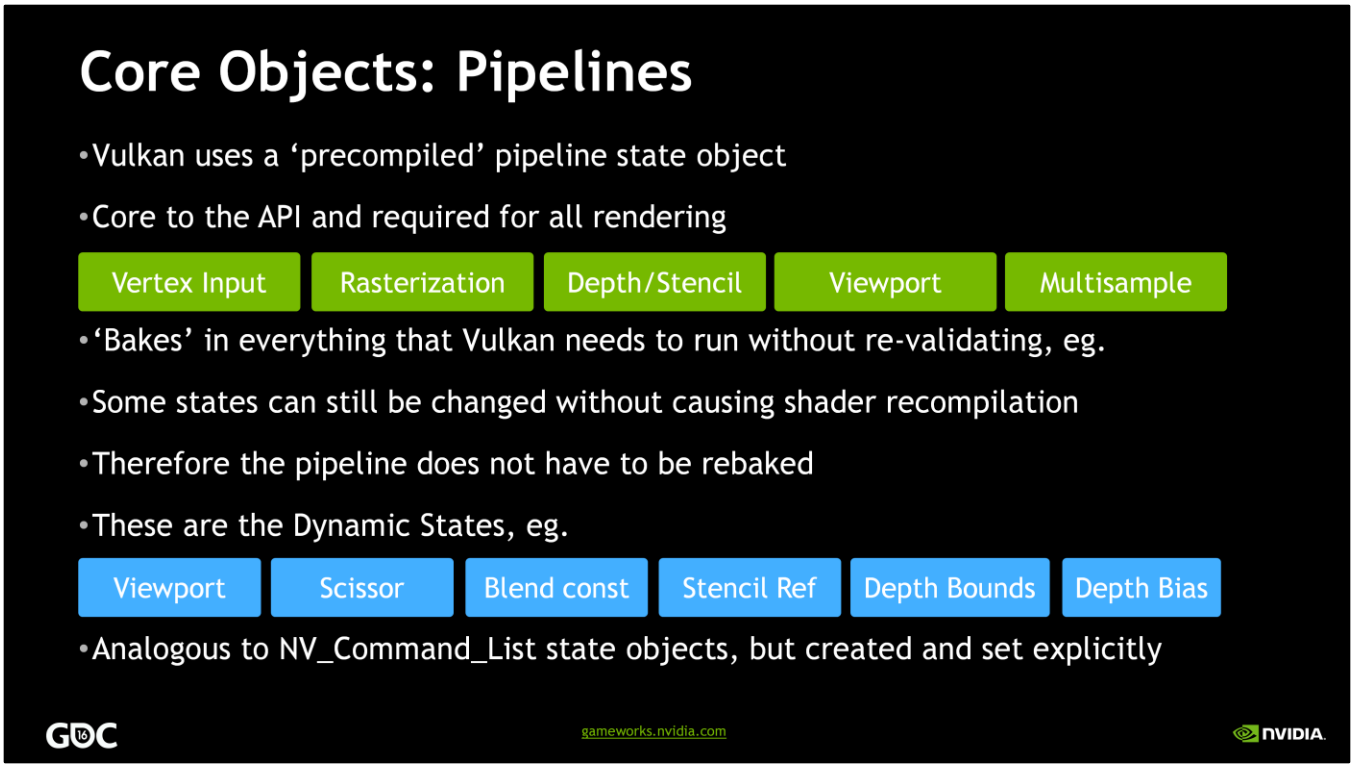
- You may have more than 1 Vulkan device on your system
  - A VkPhysicalDevice represents the actual hardware on the system.
  - Query Vulkan for its available VkPhysicalDevices
- VkDevice object "methods" include:
  - Getting Queues (used for all work submission)
  - Device memory management
  - Object management (buffers, images, sync primitives)

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

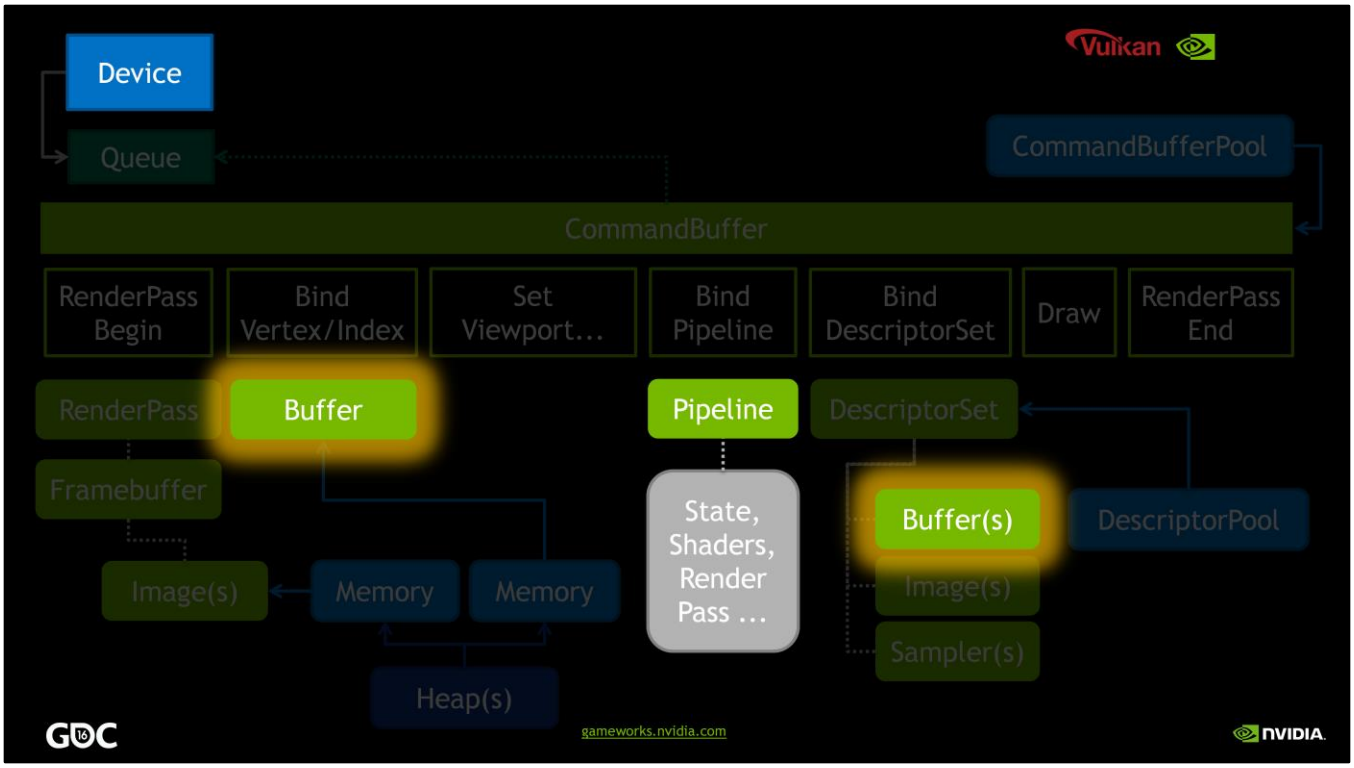
Since we often have more than one Vulkan-compatible GPU in a system, Vulkan has the concept of a physical device. We can query the available physical devices for their core properties and capabilities so we can choose between them. We use the chosen device to provide us with resources such as device memory-based objects. Also, the device provides us with a queue object that is our main interface with the GPU, along with synchronization objects for that queue.



Next, a much more complex object, the pipeline <click>. Pipelines represent an overall rendering pipeline and most of its configuration.



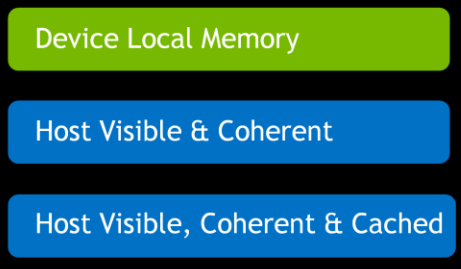
Pipelines are designed to be precompiled at setup time and include enough information in them that expensive operations like validation or shader recompilation should not be needed at render time. However, it also means that a lot of layouts and states cannot be changed within that pipeline. If you need to render with a different set of those states, you'll need another pipeline object. Note that these are similar in many ways to the N V command list state objects, only pipelines in vulkan are always created explicitly and directly, not collected from an implicit state capture.



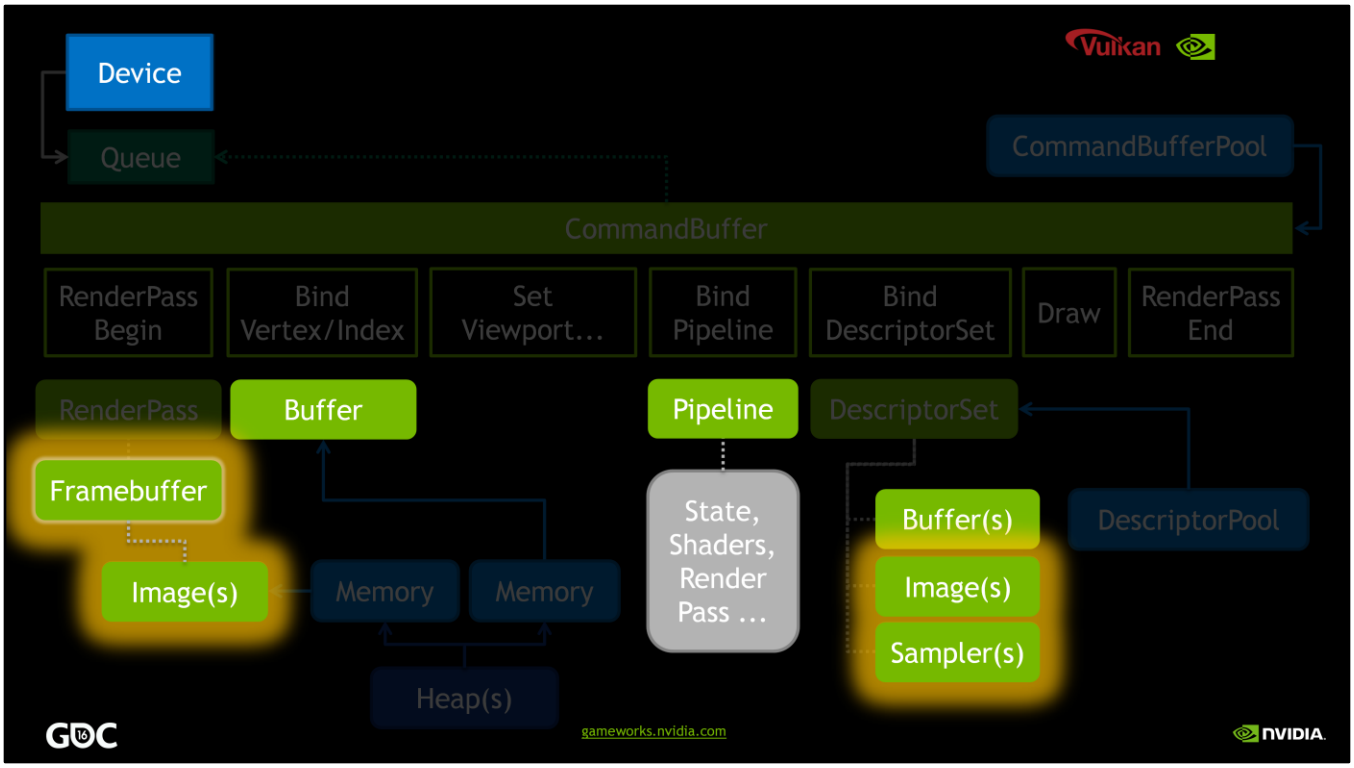
Of course, we don't bake everything into a pipeline; pipelines are reusable, and the next sets of objects we will cover can be bound dynamically to a pipeline. First, <click> buffers.

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



Buffers hold data similar to VBOs, UBOs and SSBOs that we are used to from GL; they are highly heterogeneous in Vulkan. But in Vulkan we have to pay careful attention to the type of memory in which a buffer is created. We have device local memory, and we have memory that is mappable and optionally cached by the CPU side. We’ll talk more about where we get this memory in a bit. But in Vulkan, memory type is a functional property, not just a hint to the driver.

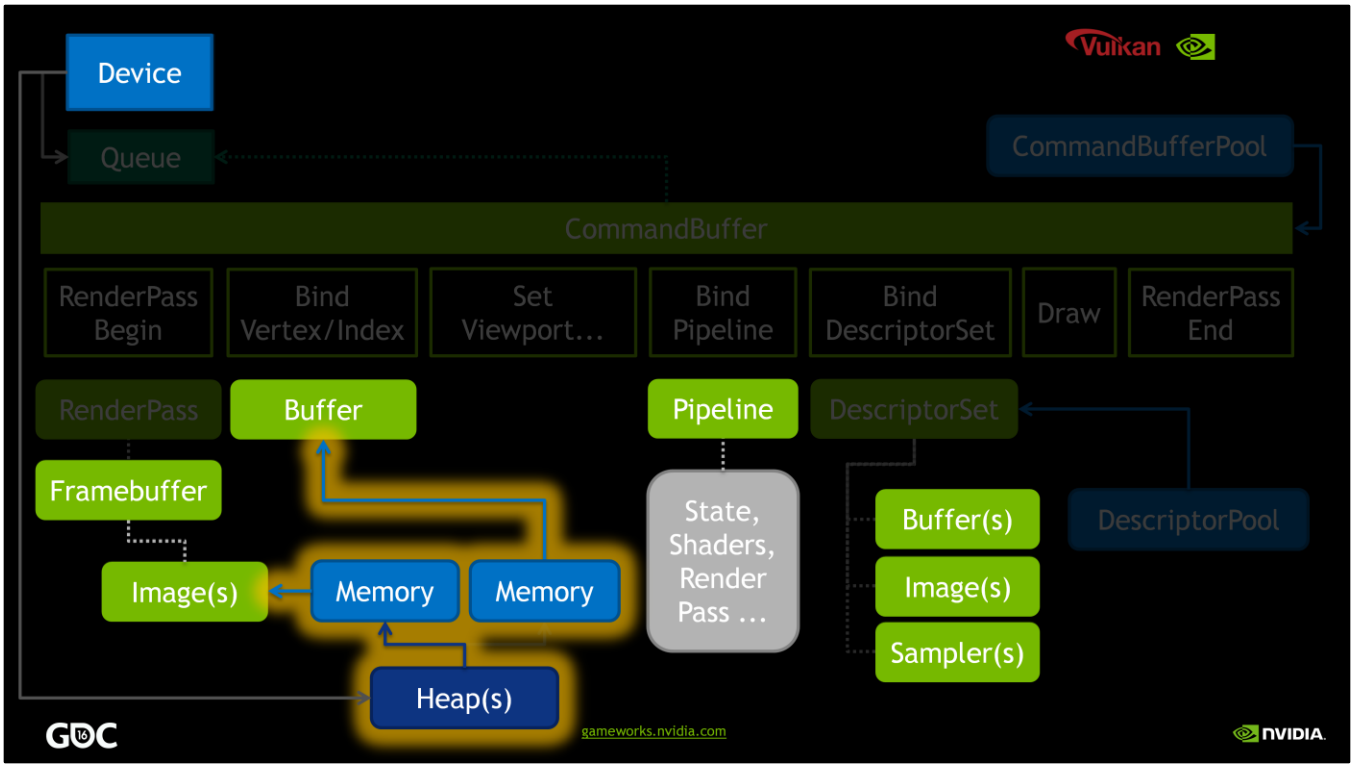


Images <click> are another familiar type of memory block

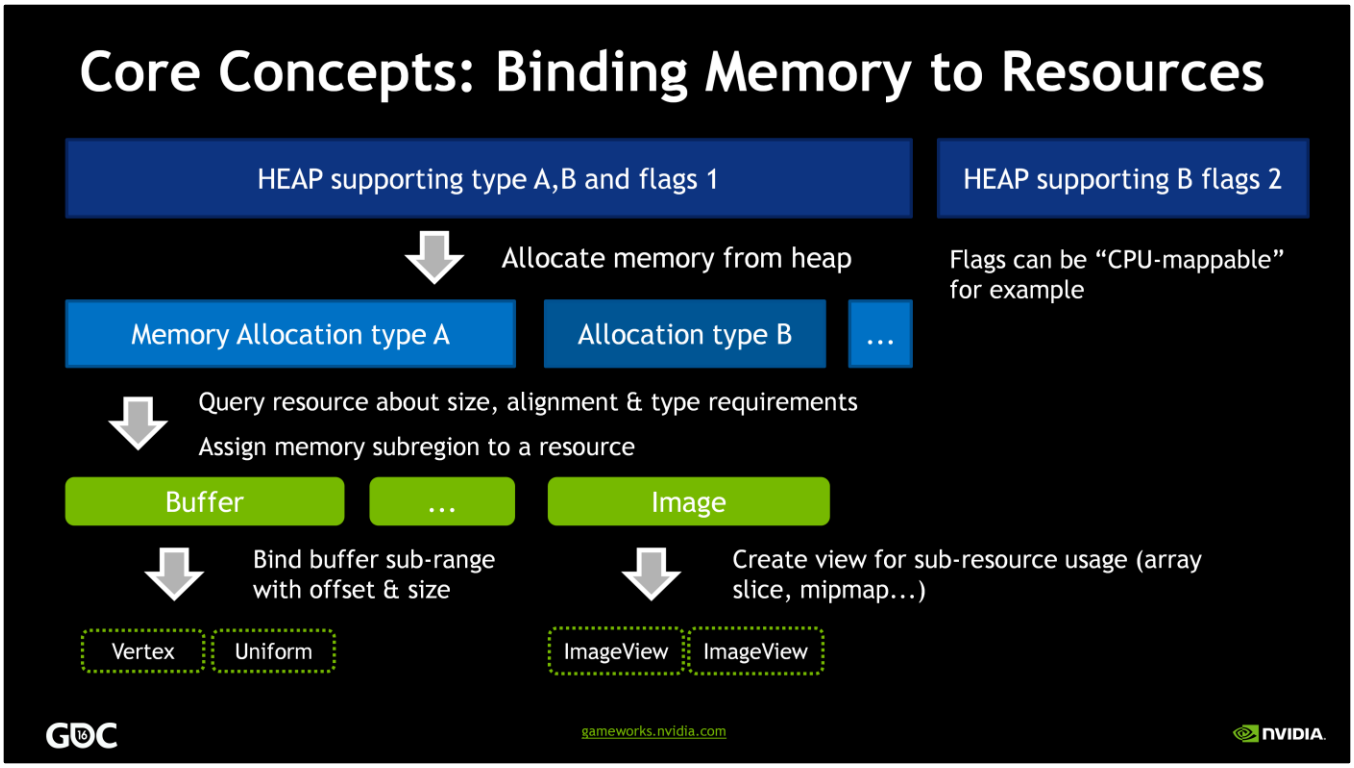
Core Objects: Images

- 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

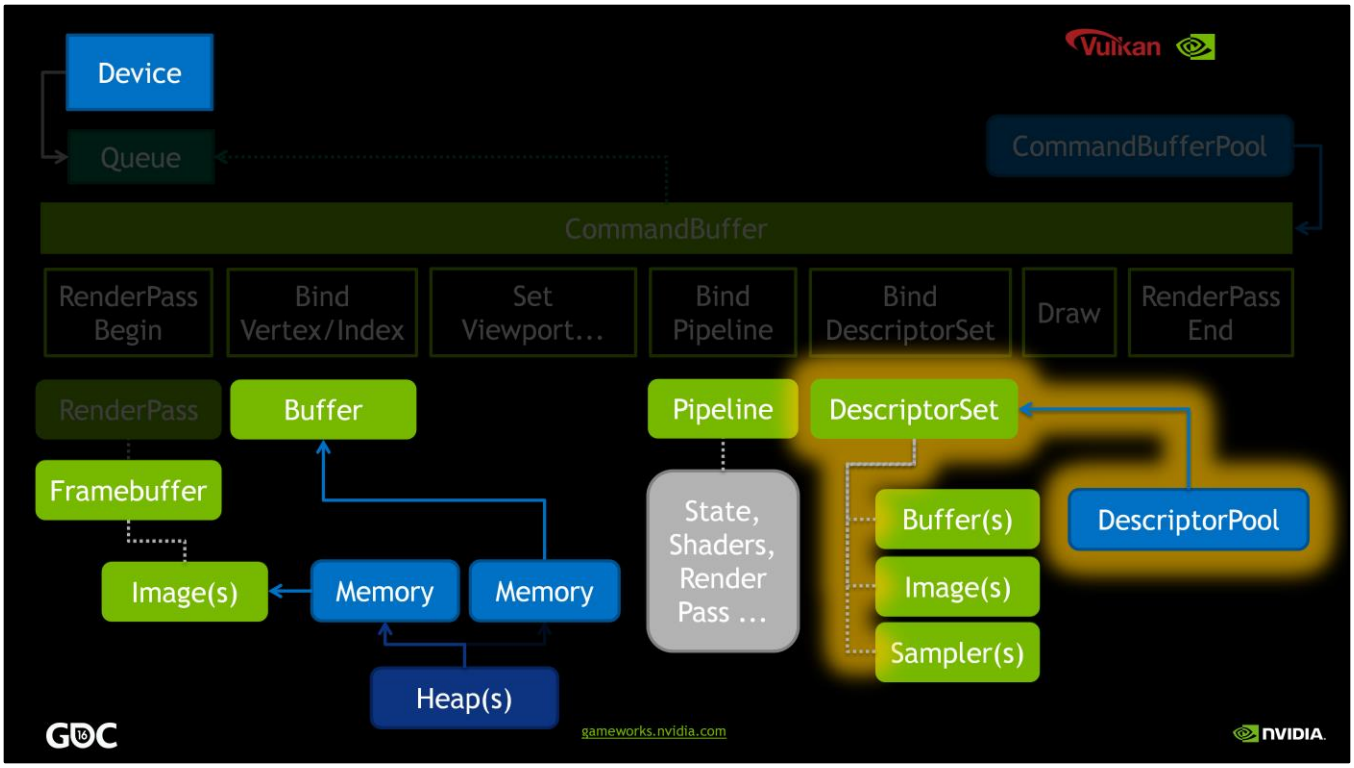
Images represent all manner of structured pixel-like arrays. And as with many other objects in Vulkan, the app specifies details of how it intends to use that object; some creation parameters control how the pixels are laid out in device memory. For example, tiled versus linear images are specified explicitly, and selection of a non-tiled layout for a texture can cause serious performance degradation. Note that images are not interpreted directly – this is the job of Image Views, which can be used to re-interpret the same image object in different manners.



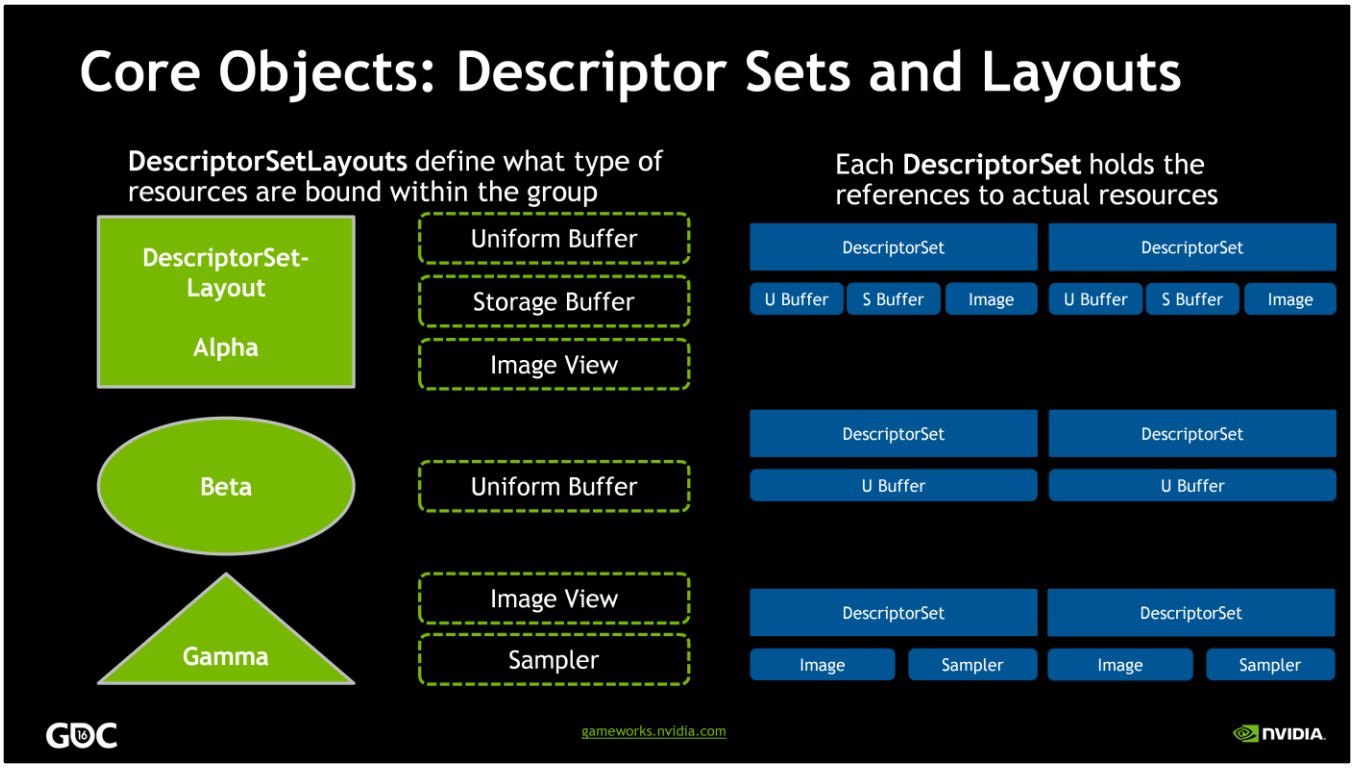
Of course, memory has to come from somewhere, and in Vulkan, it comes from <click> Heaps in a device.



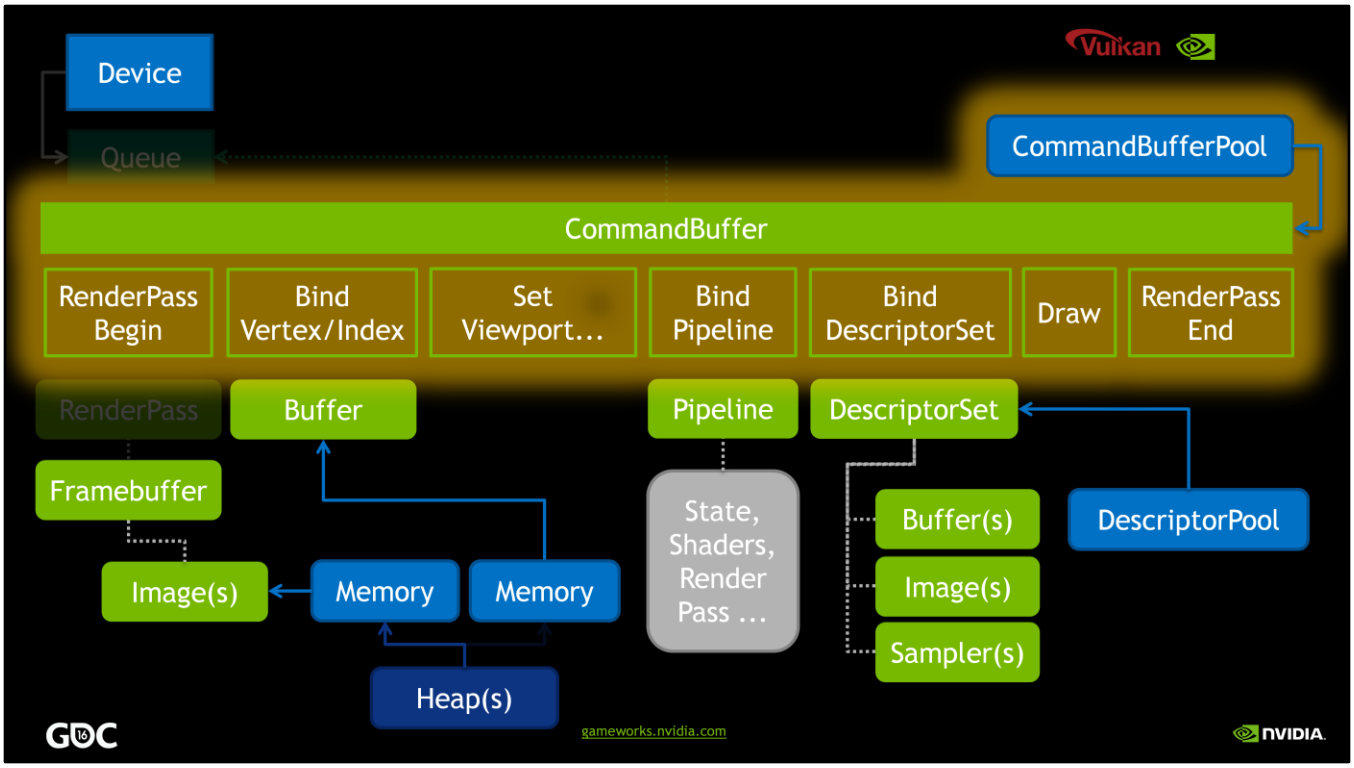
Memory in Vulkan is allocated from a SPECIFIC heap in a device. <click><click> Some heaps can generate multiple kinds of allocations, some only one, and this changes based on your platform. <click>Note that allocating memory from a heap in Vulkan is NOT a one to one mapping with Vulkan objects. <click> An allocation can and should contain <click> multiple objects; it's a basic Vulkan design philosophy



Descriptor sets <click> are how Vulkan allows an app to reuse a pipeline object by dynamically binding resources like vertex buffers, UBOs and images to that same pipeline.



Once again, Vulkan is designed to declare as much information as possible up front, so that there are no stalls at render time. In line with this, descriptor sets are allocated with fixed LAYOUTS, defined by the app. <click>Descriptor Set Layouts allow the app to declare the <click>number and type of each kind of resource that is referenced by a descriptor set OF that type, as well as how they are BOUND to indices within a shader. <click x4>Note that a pipeline can use multiple descriptor sets of different layouts. <click>So designing descriptor set layouts carefully for your app and engine is important. <click>And of course, for each layout, you'll likely have many instances of different descriptor sets. <click x5> Finally, note that while changing a buffer object BOUND into a descriptor set requires an UPDATE to the descriptor set object, changing the memory OFFSET of the binding within that buffer can be done very cheaply. More on that later.

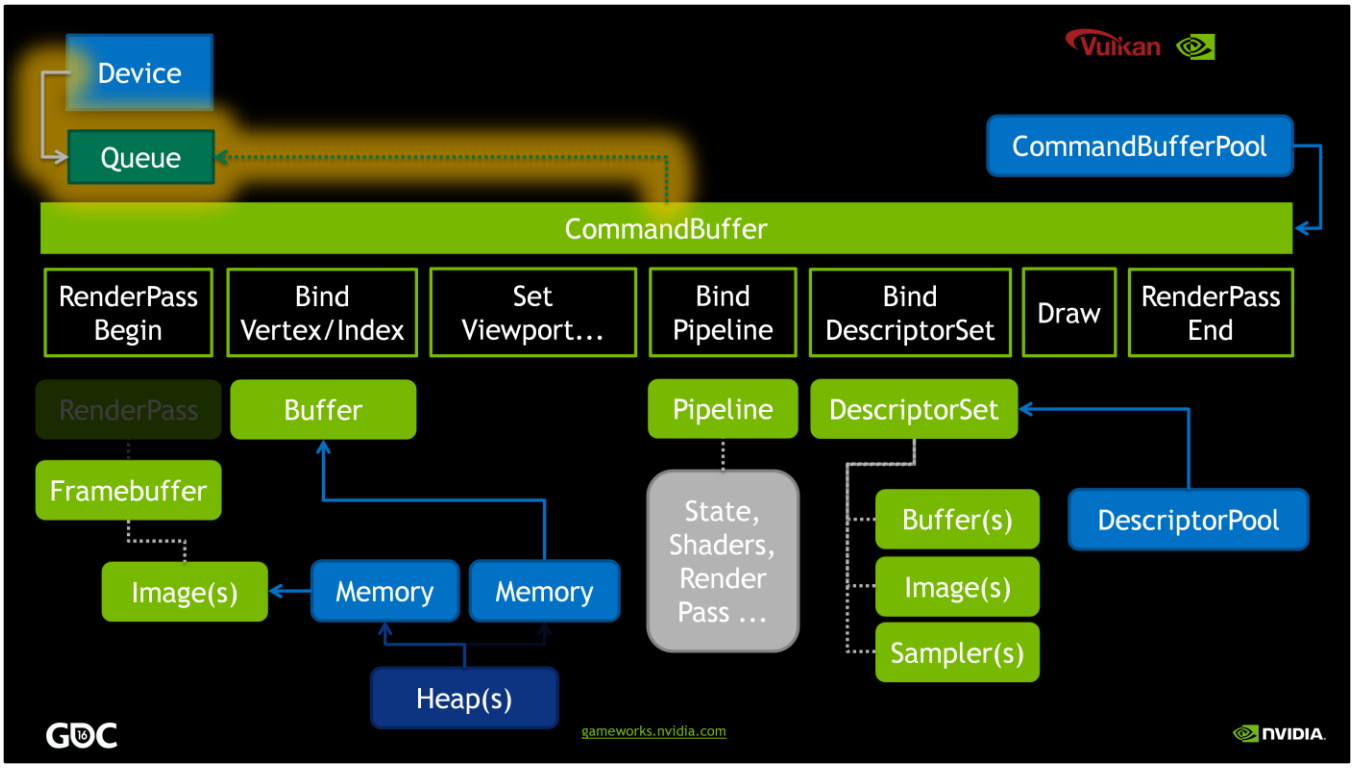


So far, we've discussed DATA having to do with rendering. What about ACTIONS? Real work. Well, that's what <click> Command Buffers are for.

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

So in vulkan, there are no functions to render IMMEDIATELY or directly to the device. All rendering, compute work and even most data transfer is via command buffers the apps build EXPLICITLY. Command buffers can be single-use OR cached by the app and resubmitted many times. And of course, Vulkan has FLAGS that let the app signal that intent for each buffer from the beginning. However, note that in order for the driver to be able to OPTIMIZE the buffers and avoid recompilation, basically NO rendering state is inherited across command buffers. So they EACH need to stand on their own. They need to BIND pipelines, BIND the desired descriptor sets and RENDER, all within that buffer.



So we're creating all of these resources, binding them, and making these big command buffers. So, you may be wondering "do I actually get to talk to the GPU at some point?" <click> That's what the queue is for.

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)

A queue object is the ONLY way to submit work to the GPU. They are EXPLICIT objects, so there's no hidden queue or synchronization going on INTERNAL to the driver, nor is there a CONTEXT to be bound to each thread. Queues are simple objects. The APIs are very simple. You SUBMIT work, and, if you need to, WAIT for idle. But where possible, you even avoid the latter. Semaphores allow you to internally sync operations WITHIN the queue with NO app intervention, and FENCES and EVENTS allow the app to know when batches of GPU work are complete. All of this put together also makes queue-based command buffers very thread friendly. And as mentioned, queues take not only graphics work, but also compute and memory transfer operations.

Vulkan Philosophies

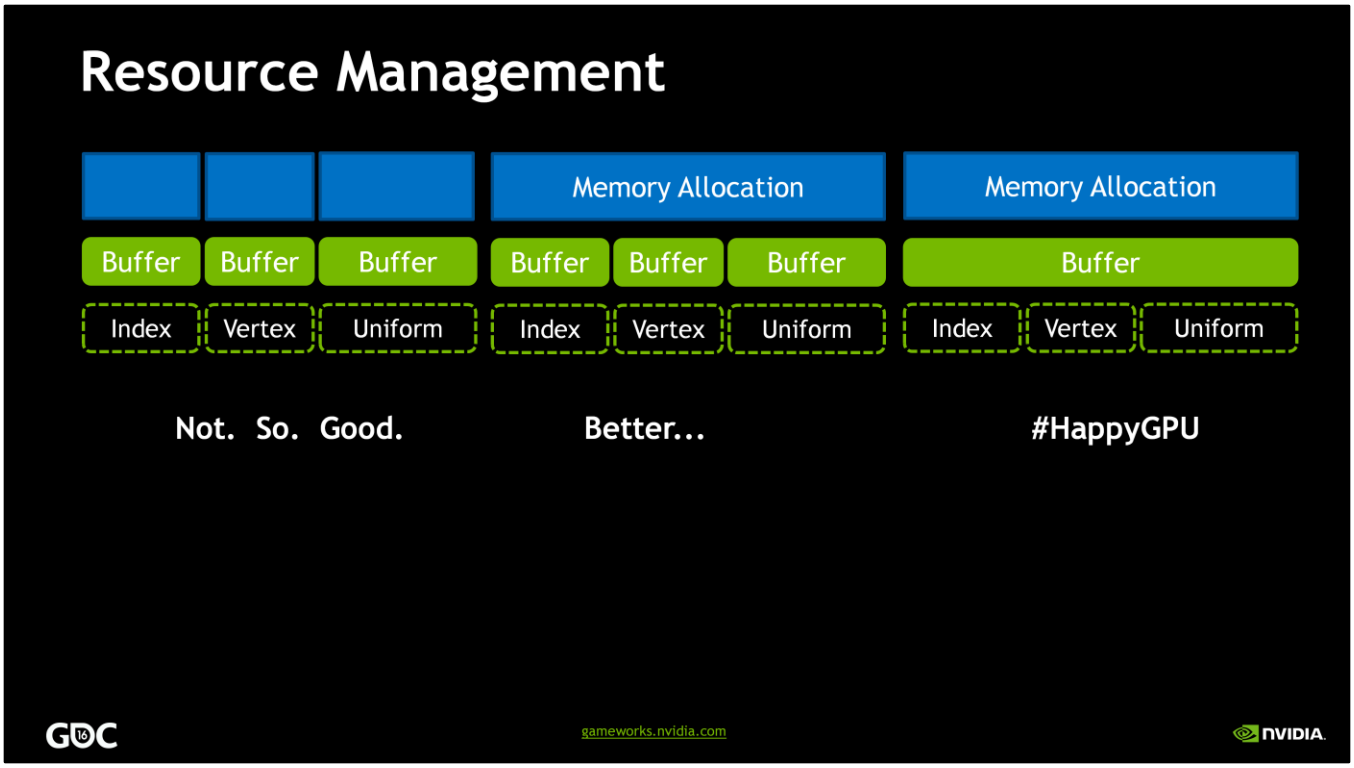
- ***Not*** specifically "the" core philosophies of Vulkan; just a few we want to highlight
- Take advantage of an ***application's high-level knowledge***
  - Do not require the driver to determine and optimize for "intent" implicitly
- Ensure that the API is ***thread-friendly*** and explicitly documented 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

Now that we've introduced the various players and made allusions to how they fit together, let's reiterate and discuss some of these core vulkan philosophies that we mentioned up top.

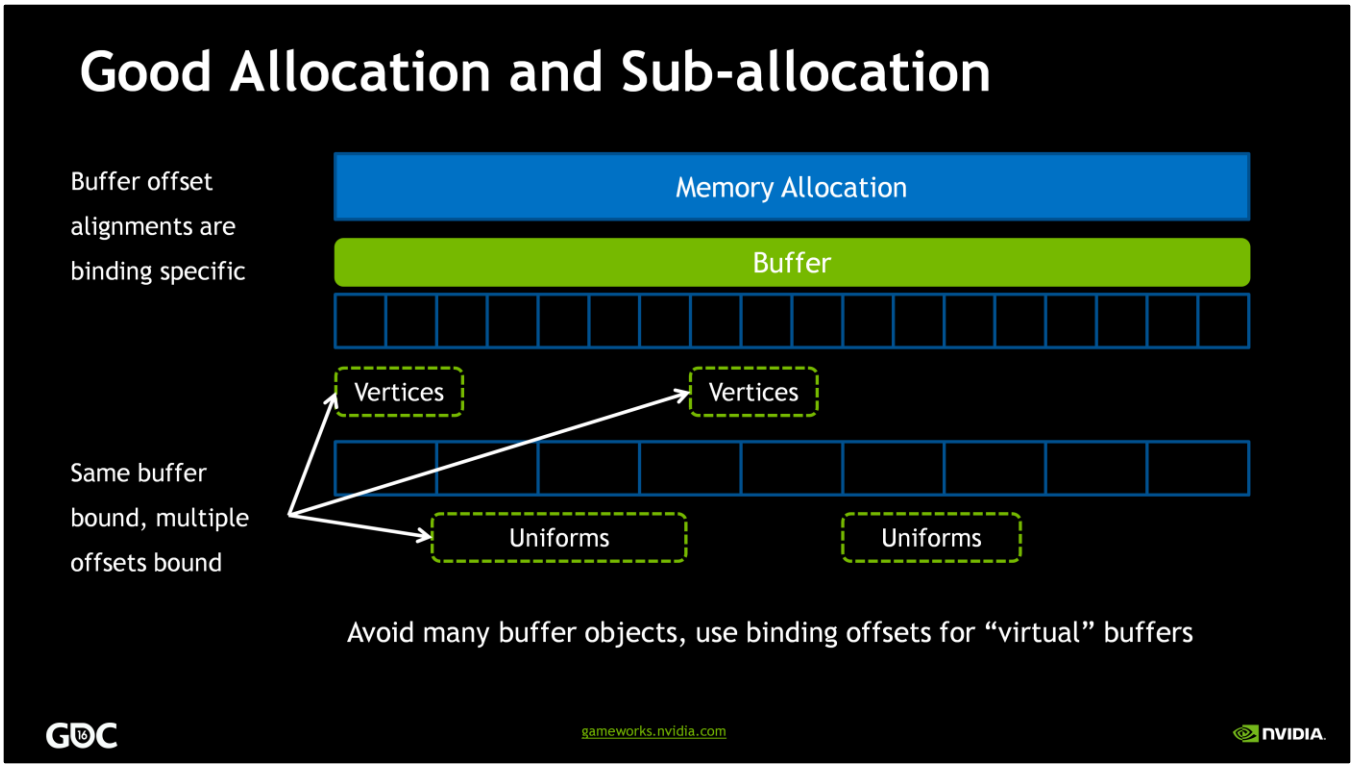
Vulkan Philpsophy: Exploit App knowledge

- 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

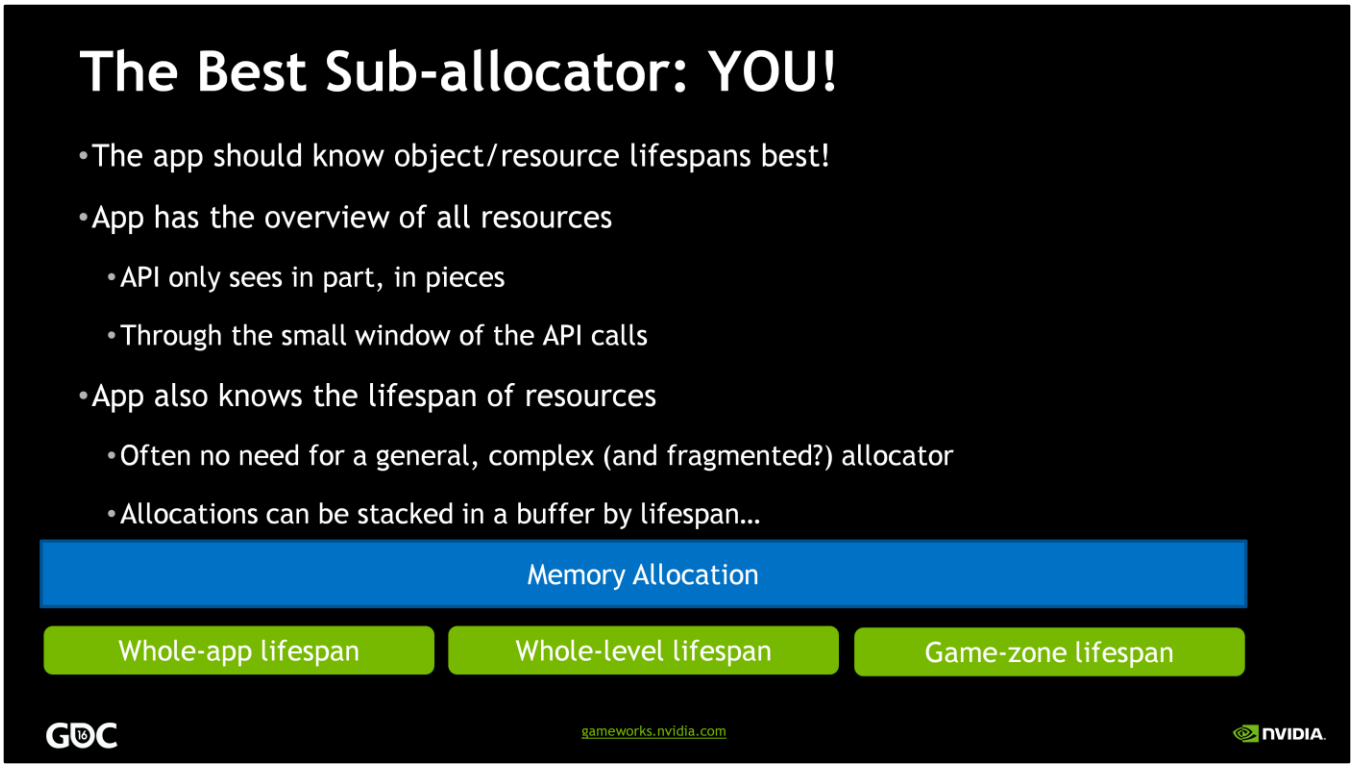
A knowledgeable app developer is a powerful creature. And many rendering APIs can only see into the mind of that developer through narrow slits of API functions. Vulkan seeks to remedy this; The FACT that MOST applications use a reasonably-sized, FIXED set of rendering pipeline structures is made explicit in Vulkan. Also, applications have a lot more CONTROL over and RESPONSIBILITY for things like object lifespan, thread synchronization, and CPU/GPU parallelism.



For example, managing resources. Vulkan makes it possible to create multiple buffer objects within a single memory allocation. The classic method often seen in GL code is that allocations <click>, buffer objects <click> and buffer object uses are <click> one to one allocations. This is clearly not optimal for memory allocation. <click>And it is NOT how things should be done in Vulkan. The next step up is <click> to use a single allocation from a Vulkan heap and create <click x3> multiple buffer objects from that. <click> Not great, but much better. <click>Vulkan makes it possible <click> to place index storage <click>, vertex storage <click> and even uniforms <click> into a single buffer object. <click>This is good, as it helps with both memory allocation and frequency of resource re-bindings.



Vulkan makes this sub-allocation of buffer objects easier by having offsets in itsbinding APIs. An app does not just bind a BUFFER; they bind a buffer AND an offset. In fact, that OFFSET can even be changed dynamically per draw call. So a buffer can be updated into the descriptor set, and then the lighter weight call that just binds the descriptor set and changes the offset can be used at higher frequency. So while changing the actual buffer object bound in a descriptor set requires updating the ENTIRE descriptor set<click>, specifying a new offset into that buffer <click>for a given binding <click>can be done cheaply when binding the <click>descriptor set.



As for the sub-allocator system, well, that's YOU. And this makes sense. One of the best ways to group allocations is object lifespan and usage. And the app should know best which objects share lifespans and binding patterns; <click>objects that live the entire app life, <click>objects that are used only in a given game level, and <click>dynamic objects that pop up and down per frame. So Vulkan lets apps do this sub-allocation of heap and buffer subsections themselves.

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

Threading is a topic that has always been complex in OpenGL, and Vulkan's design is a reaction to this and to the rise of multi-threaded games and 3D apps. The Vulkan spec is very detailed with respect to concurrency behavior of every function, and in general, that responsibility falls to the app. The API provides numerous sync primitives to make this work well. So lets discuss some of the common threaded rendering methods

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

The most common ways that apps thread their rendering behavior are <click>ONE, threading the updates of resources like vertices and uniforms and then <click>TWO, the "big ticket" case; actually generating rendering work efficiently from multiple threads. This latter case is where Vulkan shines.

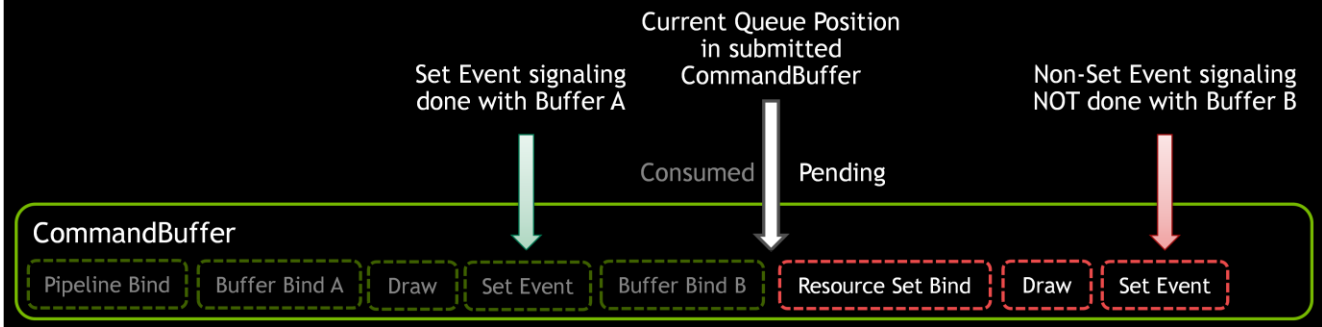
Threads: CPU Data Updates

- 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

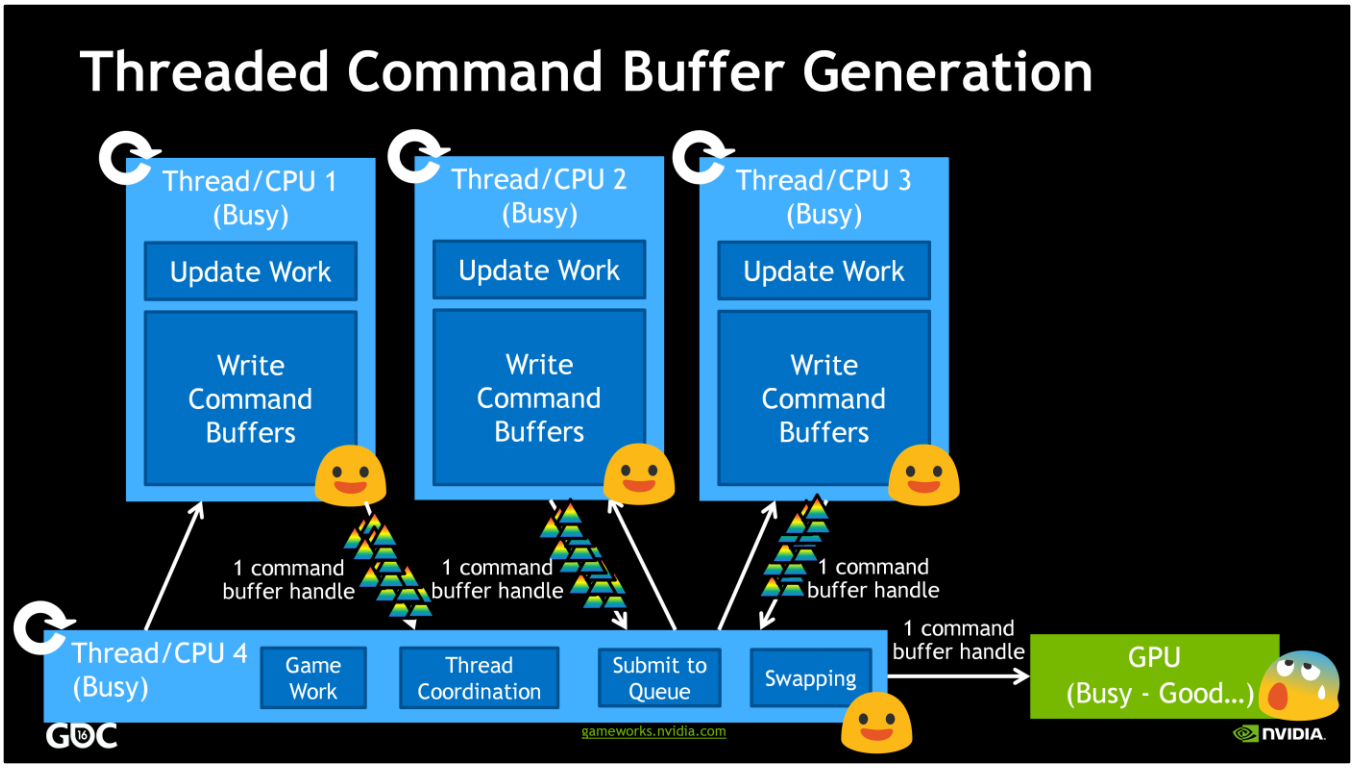
Vulkan has several methods of updating data from multiple threads. The first <click>is to use mapped buffers; depending on whether the memory was allocated as coherent or not, the app may need to flush the modified ranges explicitly. This form of threaded update is common for vertex buffers and UBOs. <click>There’s also the ability to fill staging buffers in a thread and then submit them to the queue for asynchronous D M A. This method is common for operations that require formatting or tiling conversion, like textures.

Threaded Data Updates: "Safety"

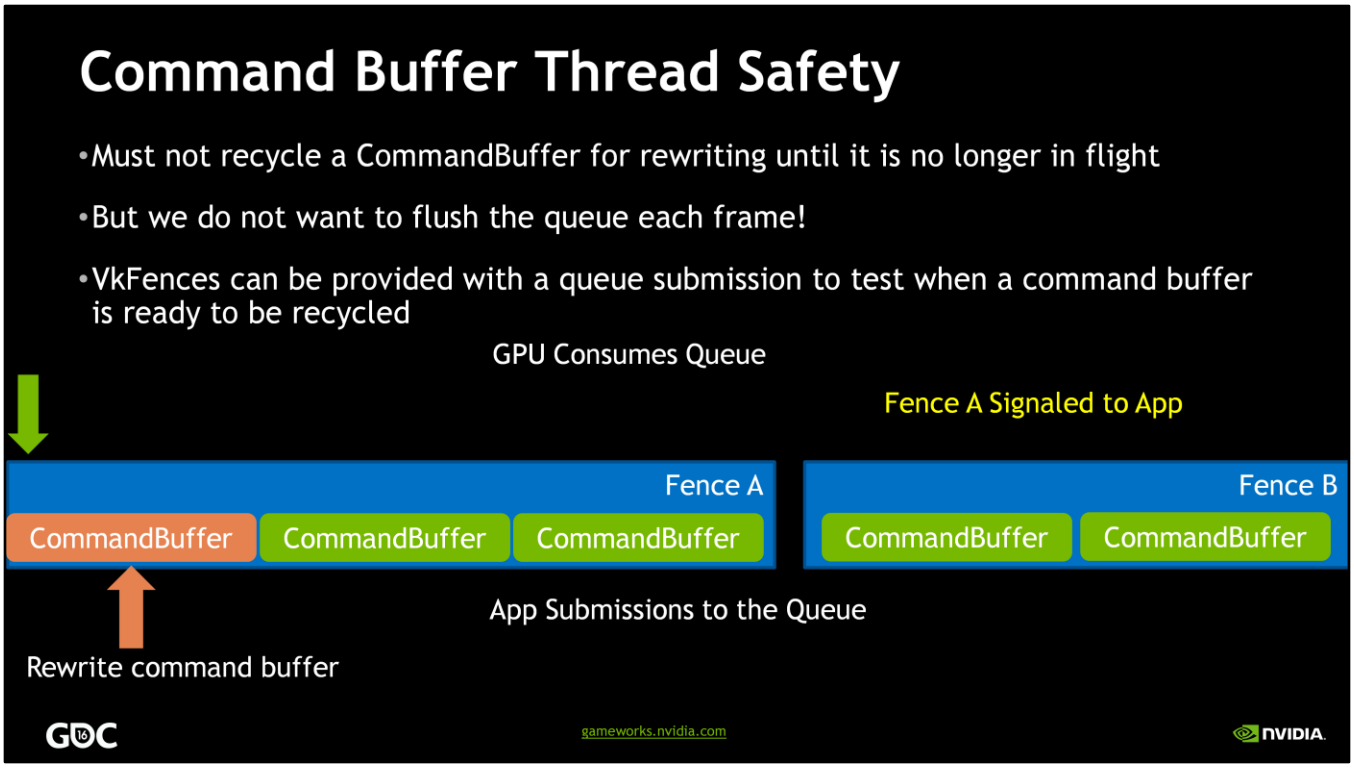
- Multiple frames will be in flight; cannot write to a single copy
- Really multi-***regioning***; use regions in a single buffer for different frames
- VkEvents can be placed in a command buffer after the last use of a copy



It is up to the app to ensure that buffers are updated "safely". So – what's a safe buffer update? Well, multiple frames will generally be in flight at once. So an app will need to keep multiple buffer copies and access them round-robin. Vulkan Events can be used to tell when a draw call that uses a given buffer has completed in the GPU. By placing a set event call in the command buffer after the draw call, the app can wait on that event before recycling that copy of the buffer. In the best case, you'll have enough copies of the buffer so that you never ACTUALLY wait on the events. But that will be dependent on the target system performance.

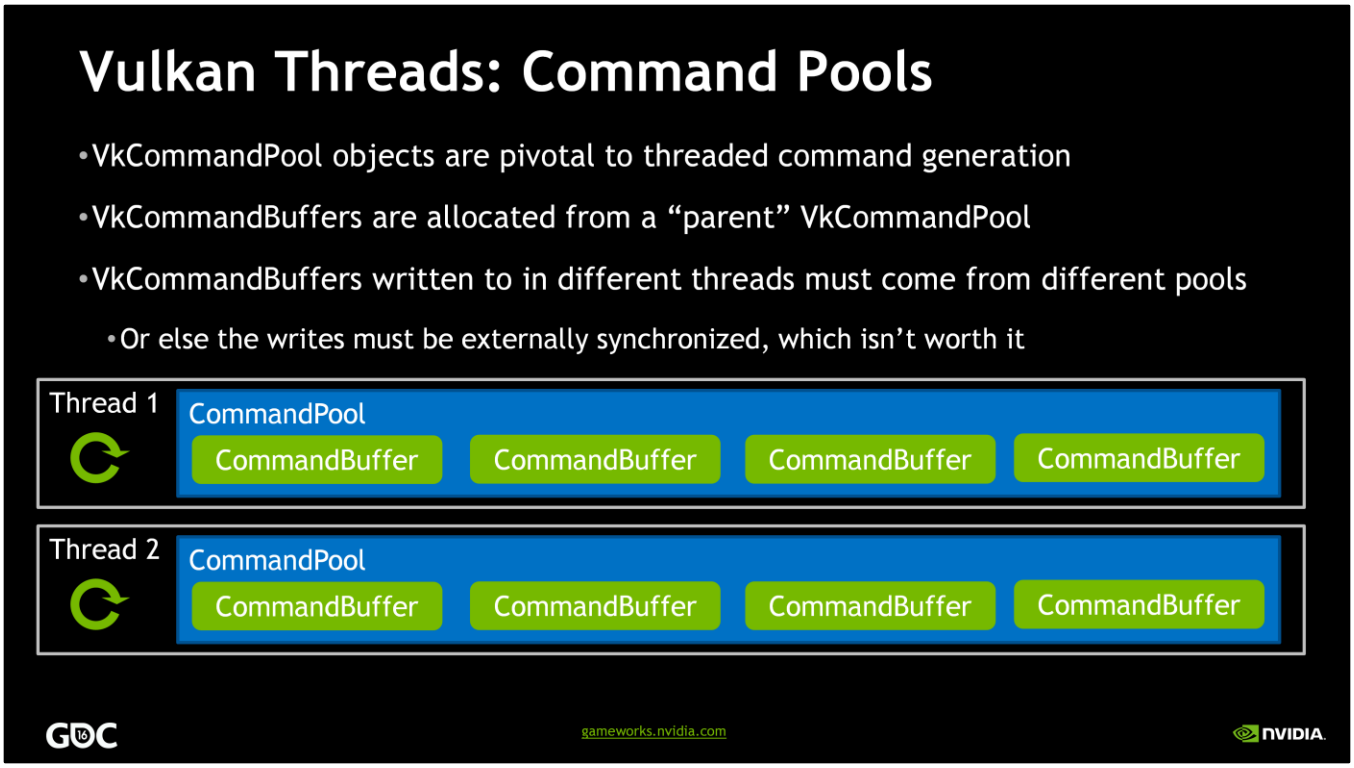


The next type of threading is even more tempting for advanced applications. Using several threads to generate command buffers that render independent parts of the scene. This can be a big win with complex scenes. The diagram here lays out the case. In Vulkan, the app can do their <click>update work AND <click>generate command buffers in simultaneous threads, without having to synchronize between them. At the end, the threads can either <click>pass the command buffer handles back to the main thread for queue submission, or can submit via per-thread queues; note that queue submissions are not free, so the performance is likely better passing the handles and submitting them together from the main thread. All of which leads to a ton of rendering <click>generated by busy threads to keep the <click>GPU busy with big batches of real work<click>.



Once again, we need to think about queue safety. Luckily, Vulkan makes this case even more pre-fabricated. The app can create its command buffers <click x3> and then submits a group of them to the queue <click> The Vulkan submit to queue function takes an optional fence as a parameter. The app can continue creating and submitting buffers in parallel <click x6>. That fence will be signaled <click> when the command buffers represented in that submit are complete and can be reused. The app just needs to wait on that fence before rewriting the command buffer.<click>

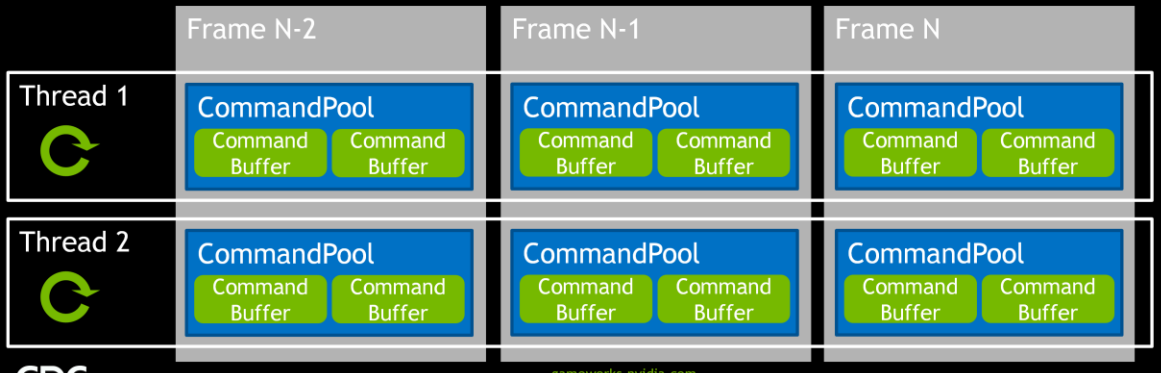
To see this in action, visit the NVIDIA booth on the expo floor and you can see several examples of threaded rendering with vulkan in action. Specifically, our Fish demo – it is a cross-platform, cross-API demo that shows threading of data updates in OpenGL ES and Vulkan and command buffer generation with Vulkan.



As mentioned, we need a round-robin of command buffers per thread and can only reuse them when they are not in-flight. However, there’s another threading-related item to consider. Command Pool objects are parent objects used to create and manage their child command buffers. Command buffer operations cause work to happen in the command buffer’s parent command pool, such as memory allocation. And any work in a command pool must be synchronized across threads. So the app likely wants to have a Command Pool per thread and associate all of its command buffers with that pool, so that there is no need to synchronize. Each thread can do command buffer operations at will.

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:

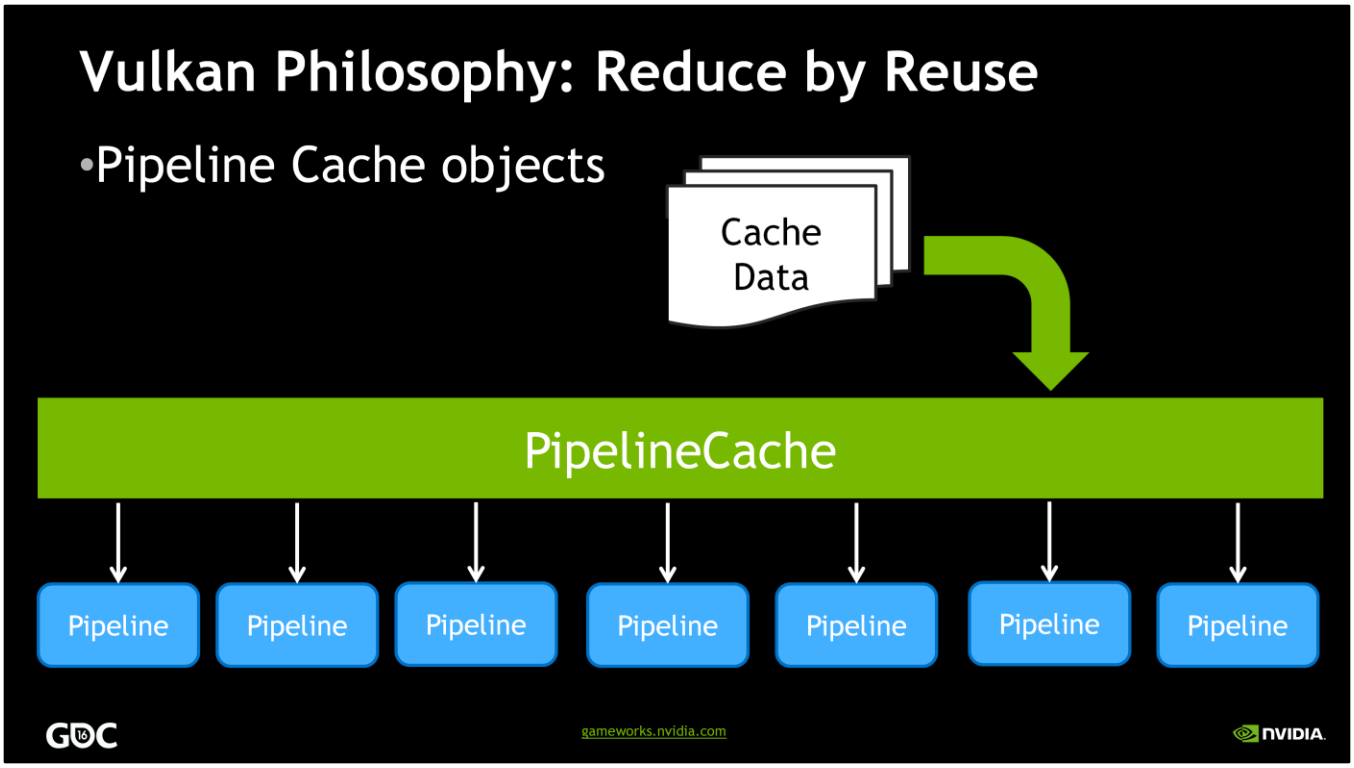


Note that command pools can be reset in bulk, which is a fast and clean way to reset all of the pool's command buffers and reclaim the memory. So if you have multiple command buffers per frame, per thread, then it may even make sense to have a command pool per thread-frame pair. That way, when all of the command buffers for a given thread/frame pair are no longer in flight, you can reset the entire command pool en masse. This can be fast and help avoid fragmentation at the same time.

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

Descriptor pools are analogous to the command pools, but for descriptor sets. Since dynamically created objects may require new descriptor sets to be created from within a thread, it is important to consider a descriptor pool for a given thread. Also, like command pools, descriptor pools can be reset en masse. So creating descriptor pools based on lifespan (say, a per-game level pool) can allow for trivial, fast and clean deletion of lots of descriptor sets when no longer needed.



The final vulkan philosophy we mentioned was reduce by reuse. And most of those topics we've already covered as we went along. Pipeline objects are not only in and of themselves trivially reusable without having to re-generate a set of states over and over again, there are even additional optimizations possible. For example, Vulkan supports explicit pipeline cache objects<click>. If one is created and then passed in when creating pipeline objects<click x 7>, the cache can accumulate driver-specific information about a set of pipelines. This opaque data can be retrieved by the application <click>and saved<click>. On subsequent runs of the app <click x2>, the data can be loaded and passed to Vulkan<click>, and the driver may be able to greatly limit the cost of creating that same pipelines objects <click x7>.

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

This eyechart covers some of the tradeoffs of the various API sets. Nothing shocking after the discussions here. AZDO, command list and Vulkan each provide improved single-threaded performance options. Real re-use of created draw work is the domain of command lists and vulkan. And for multi-threading, vulkan is likely to be best. But of course, there's tradeoffs in developer workload.

Beneficial Vulkan Scenarios

1. Has parallelizable CPU-bound graphics work
   - Vulkan's CommandBuffer and Queue system make it possible to efficiently spread the CPU rendering workload
2. Looking to maximize a graphics platform budget
   - Direct management of allocations and resources help on limited platforms
3. Looking for predictable performance, desire to be free of hitching
   - Precompilation of state, Pipeline structure avoids runtime shader recompilation and state cache updates

In terms of Vulkan, if you are CPU bound on graphics work, <click>looking to maximize a known, tight platform resource budget, or <click>very focused on a lack of hitching, Vulkan is a strong consideration.

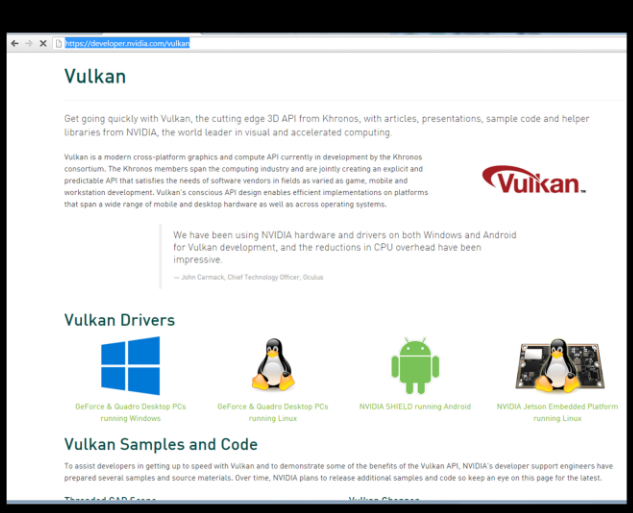
Cases Unlikely to Benefit from Vulkan

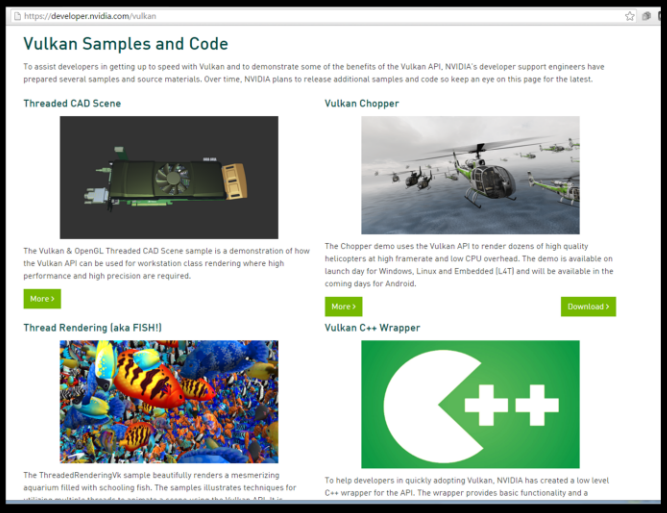
1. Need for compatibility to pre-Vulkan platforms
2. Heavily GPU-bound application
3. Heavily CPU-bound application due to non-graphics work
4. Single-threaded application, unlikely to change to multithreaded
5. App targets middleware engine, little-to-no app-level 3D graphics API calls
   - Consider using an engine targeting Vulkan
6. App is late in development and cannot risk changing 3D APIs

However, if you need a wide range of platform support immediately, Vulkan is still new. <click>If you are heavily GPU bound, Vulkan is unlikely to help (note that we're talking about ACTUALLY GPU bound – use your tools to make sure it isn't just being driver-bound, where Vulkan could help). <click>If your app is bound on non-rendering CPU work, well, put your optimization time elsewhere… <click>And if you really want the most from Vulkan, being a threaded app or easily threadable helps. <click>Finally, there are other development concerns, like your use of rendering middleware, and <click>where you are in the development process.

Vulkan Resources

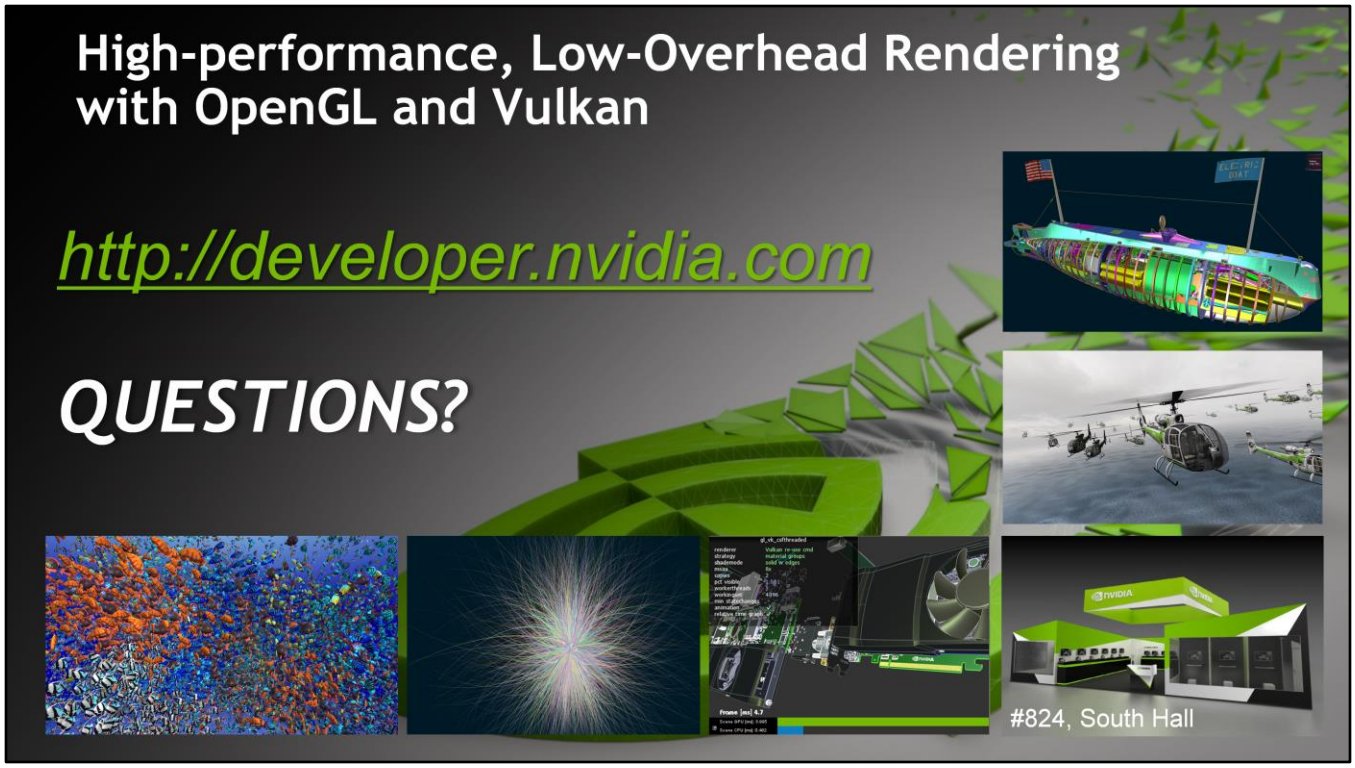
http://developer.nvidia.com/vulkan





GDC gameworks.nvidia.com NVIDIA

There is so much more information available for developers on Vulkan. Here, we present a set of Vulkan resources from NVIDIA developer technologies that are targeted at all levels of developer experience. In closing, we hope we've shown that numerous options exist for app developers looking to get the most from their 3D games and applications on today's rendering APIs. But there are tradeoffs to consider and decisions to be taken carefully when deciding between OpenGL, extended openGL, and Vulkan.



Thanks for joining Mathias and myself today; please visit our booth in the south hall and see Vulkan in action on NVIDIA platforms. We'll close out with some time for questions.