Migrating from OpenGL to Vulkan

Mark Kilgard, January 19, 2016

About the Speaker

Who is this guy?

Mark Kilgard

Principal Graphics Software Engineer in Austin, Texas

Long-time OpenGL driver developer at NVIDIA

- Author and implementer of many OpenGL extensions

Collaborated on the development of Cg

- First commercial GPU shading language

Recently working on GPU-accelerated vector graphics

(Yes, and wrote GLUT in ages past)

Motivation for Talk

Coming from OpenGL, Preparing for Vulkan

What kinds of apps benefit from Vulkan?

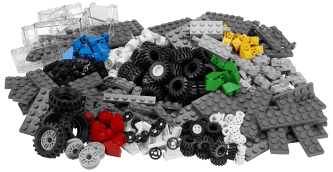
How to prepare your OpenGL code base to transition to Vulkan

How various common OpenGL usage scenarios are re-thought in Vulkan

Re-thinking your application structure for Vulkan

Analogy

Different Valid Approaches



Analogy

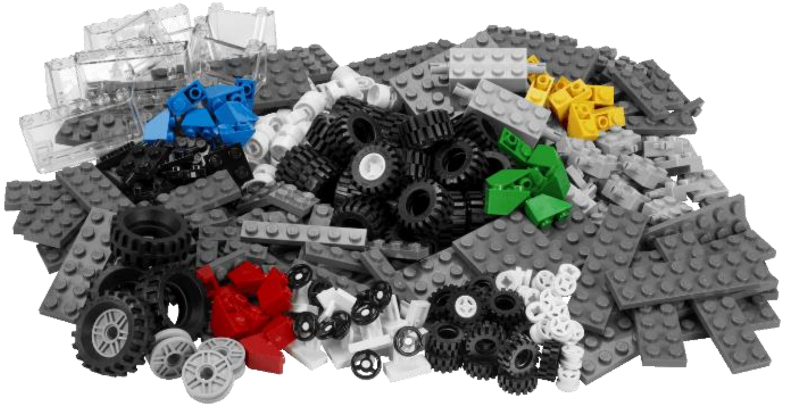
Fixed-function OpenGL

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

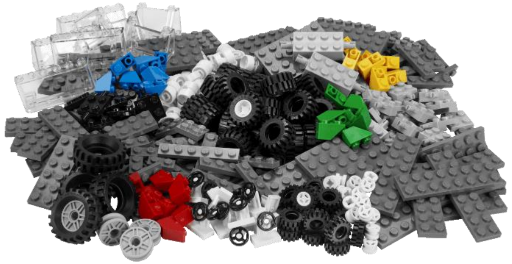
you build it yourself to race from raw materials

power tools used to assemble, adult supervision highly recommended

Analogy

Different Valid Approaches

Fixed-function OpenGL

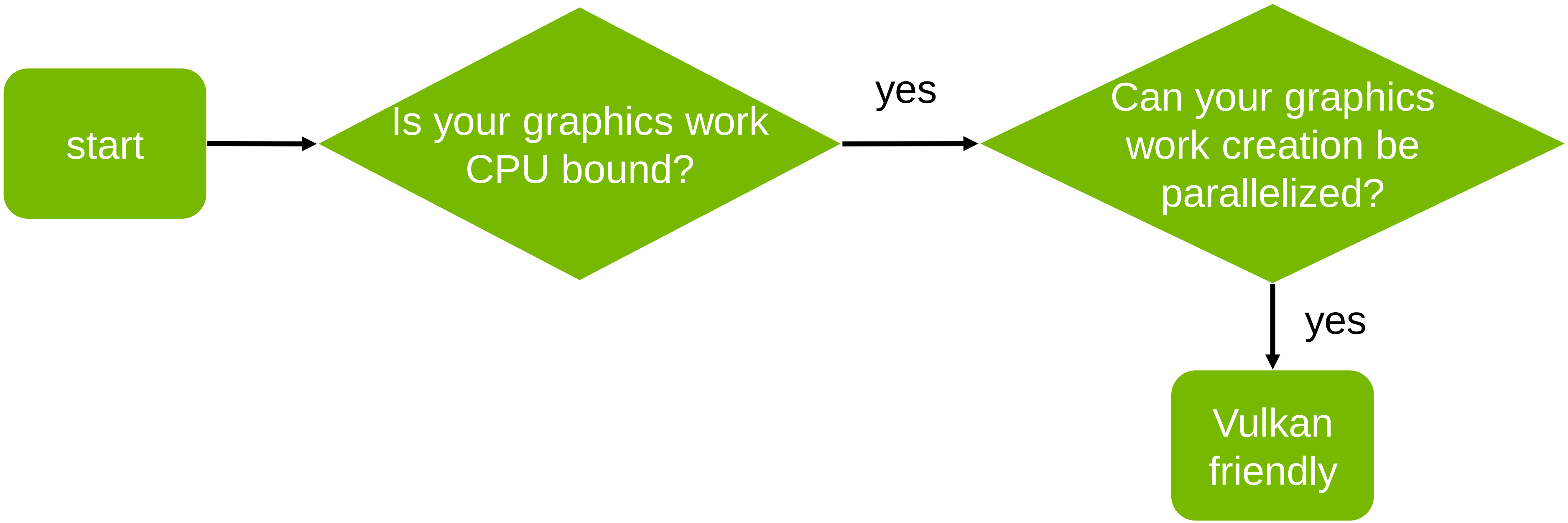


Modern AZDO OpenGL with Programmable Shaders

Vulkan

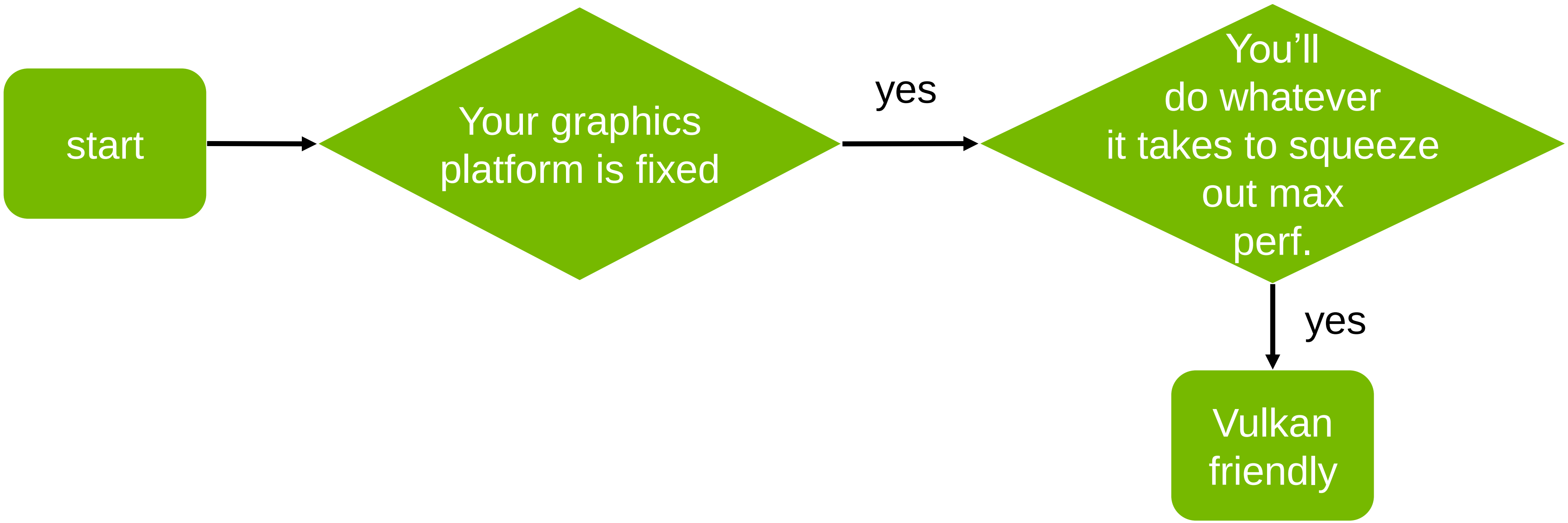
Beneficial Vulkan Scenarios

Has Parallelizable CPU-bound Graphics Work



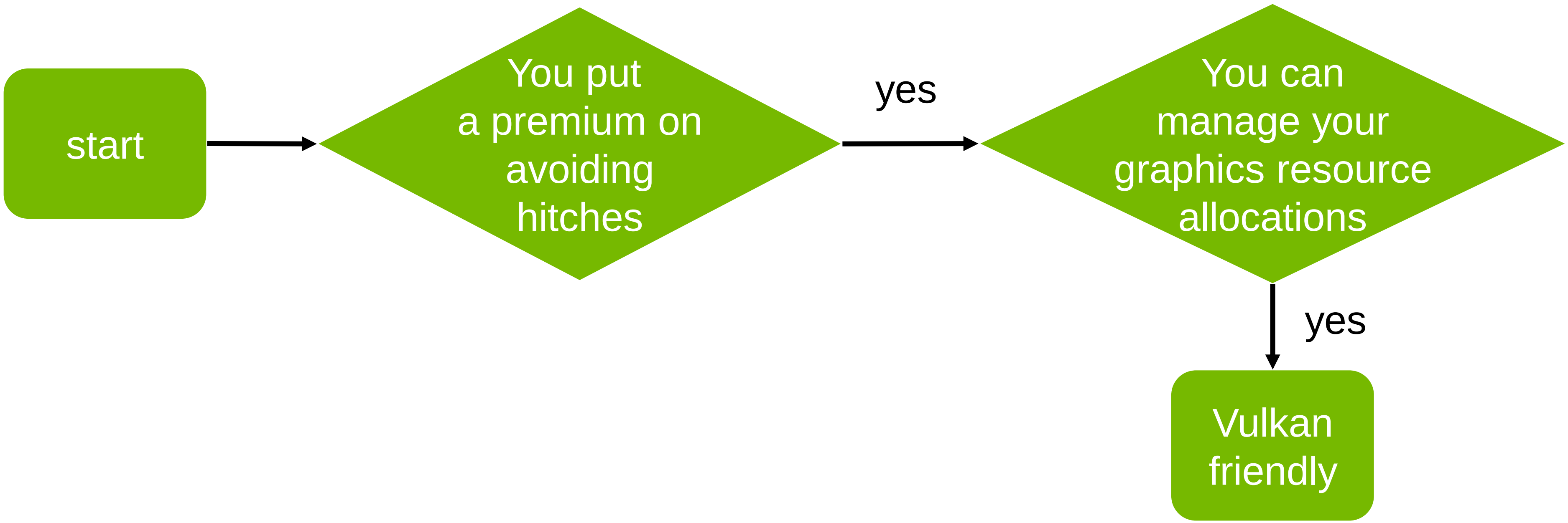
Beneficial Vulkan Scenarios

Maximizing a Graphics Platform Budget



Beneficial Vulkan Scenarios

Managing Predictable Performance, Free of Hitching



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 direct 3D graphics API dependencies
    - Consider using an engine targeting Vulkan, instead of dealing with Vulkan yourself

First Steps Migrating to Vulkan

Modernize Your OpenGL

Eliminate fixed-function

Source all geometry from vertex buffer objects (VBOs) with vertex arrays

Use all programmable GLSL shaders with layout() qualifiers

Consider using samplers

Do all rendering into framebuffer objects (FBOs)

Stay within the non-deprecated subset (e.g. no GL_QUADS, for now…)

Think about better batching & classify all your render states

Avoid depending on OpenGL context state

All pretty sensible advice, even if you stick with OpenGL

Next Steps Migrating to Vulkan

Modernize Your OpenGL

Think about how your application would handle losing the GPU

- Similar to ARB_robustness

Profile your application, understand what portions are CPU and GPU bound

- Vulkan most benefits apps bottlenecked on graphics work creation and driver validation
- Is that your app?

Adopt common features available in both OpenGL and Vulkan first in OpenGL

- Proving out tessellation or multi-draw-indirect probably easier first in your stable OpenGL code base

Again all pretty sensible advice, even if you stick with OpenGL

Thinking about Vulkan vs. OpenGL

OpenGL Has a Well-established Approach

OpenGL, largely understood in terms of

- Its API, functions for commands and queries
- And how that API interacts with the OpenGL state machine

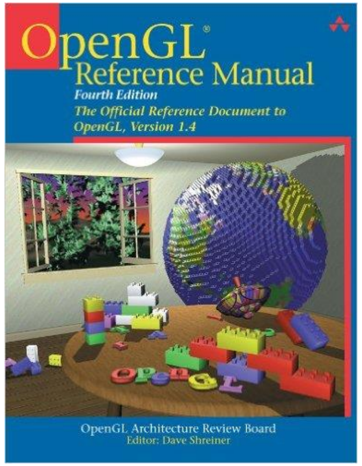
OpenGL has lots of implicit synchronization

Errors handled largely by ignoring command

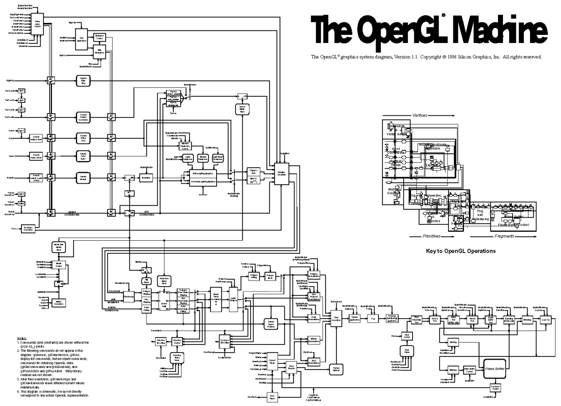
OpenGL API manages various objects

- But allocation of underlying device resources largely handled by driver

Originally client-server



+



Thinking about Vulkan vs. OpenGL

Vulkan Plays By Different Rules

Vulkan constructs and submits work for a graphics device

- Idealized graphics + compute device

Requires application to maintain valid Vulkan usage for proper operation

- Not expected to behave correctly in face of errors, justified for performance

Instead of updating state machine, Vulkan is about establishing working relationships between objects

- Pre-validates operation of actions

Explicit API

- Explicit memory management
- Explicit synchronization
- Explicit queuing of work
- Explicit management of buffer state with render passes

Not client-server, explicitly depends on shared resources and memory

Truly Transitioning to Vulkan

Vulkan Done Right Rethinks Entire Graphics Rendering

Much more graphics resource responsibility for the application

You need to allocate from large device memory chunk

You become responsible for proper explicit synchronization

- Fences, Barriers, Semaphores
- Barriers are probably the hardest to appreciate

Everything has to be structured as pipeline state objects (PSOs)

Understand render passes

Think how parallel command buffer generation would operate

- You become responsible for multi-threaded exclusion of your Vulkan objects

Common Graphics Tasks

Managing Predictable Performance, Free of Hitching

Loading a mipmapped texture

Loading a vertex buffer object for rendering

Choosing a shader representation and loading shaders

Initiating compute work

Managing a memory sub-allocator

Thinking about render passes

Loading a Texture

The OpenGL View

Well traveled path via OpenGL 3.0

```
glBindTexture(GL_TEXTURE_2D, texture_name);

glTexImage2D(GL_TEXTURE_2D, 0, GL_SRGB8, width, height,
  /*border*/0, GL_UNSIGNED_BYTE, GL_RGBA, pixels);

glGenerateMipmap(GL_TEXTURE_2D);
```

Does more than you think it does

Logical Operations to Load a Texture

1. Create host driver objects corresponding to texture/sampler/image/view/layout
2. Copy call's image to staging memory accessible to host+device
3. (OpenGL might do a format conversion and pixel transfer)
4. Allocates device memory for texture image for texturing
5. Copy image from host+device memory to device memory for texturing
6. Allocate device resources for sampler
7. Generate mipmap levels

Various Vulkan Objects “inside” a Texture

Building Up the OpenGL Texture Object from Vulkan Concepts

No such thing as “VkTexture”

Instead

- Texture state in Vulkan = VkDescriptorImageInfo
  - Combines: VkSampler + VkImageView + VkImageLayout
  - Sampling state + Image state + Current image layout
- Texture binding in Vulkan = part of VkDescriptorSetLayoutBinding
  - Contained with VkDescriptorSetLayout

OpenGL textures are opaque, so lacks an exposed image layout

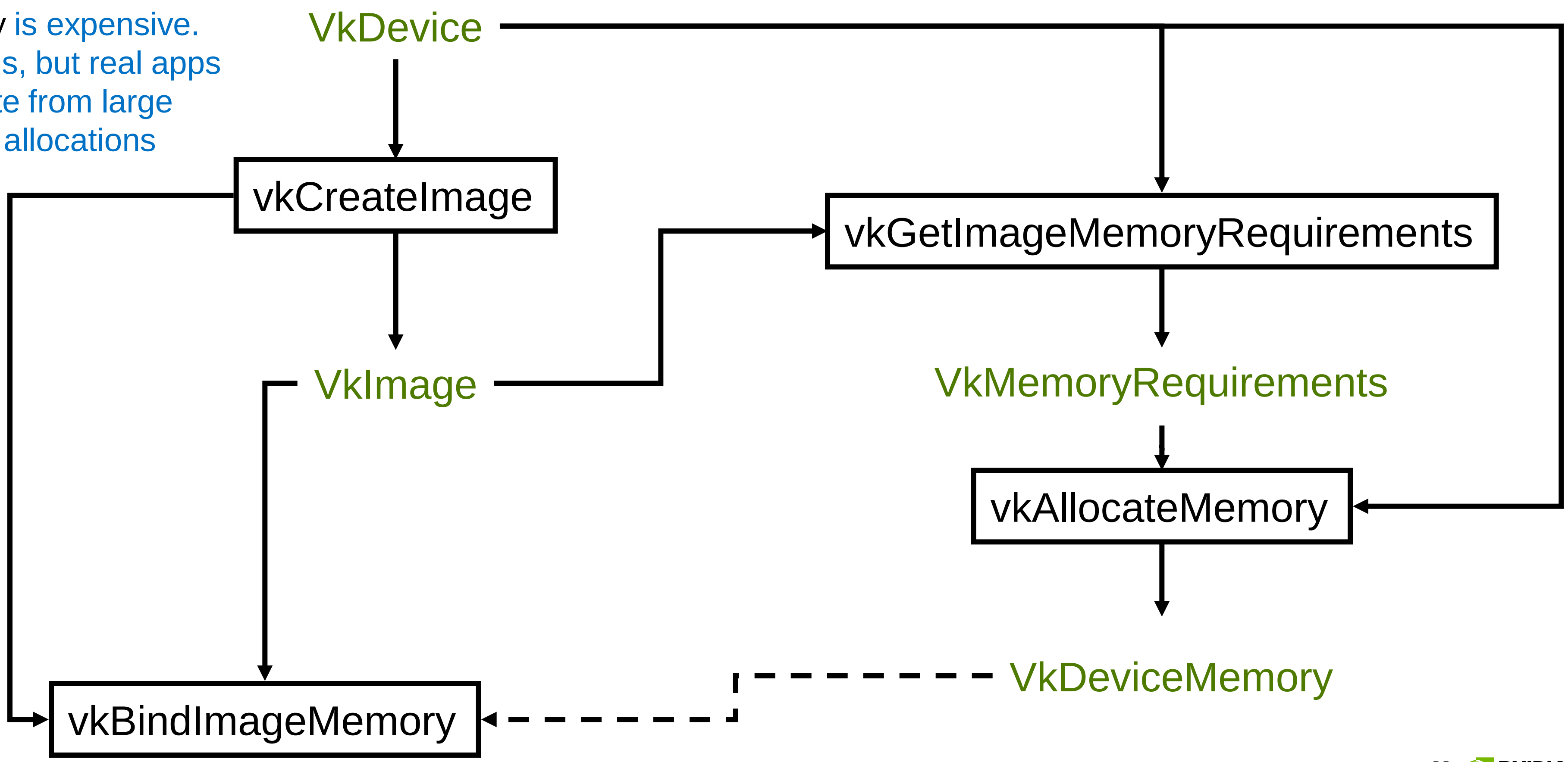
Allocating Image Memory for Texture

With Vulkan

Naïve approach!

vkAllocateMemory is expensive.
Demos may do this, but real apps should sub-allocate from large VkDeviceMemory allocations

See next slide…



Sub-allocating Image Memory for Texture

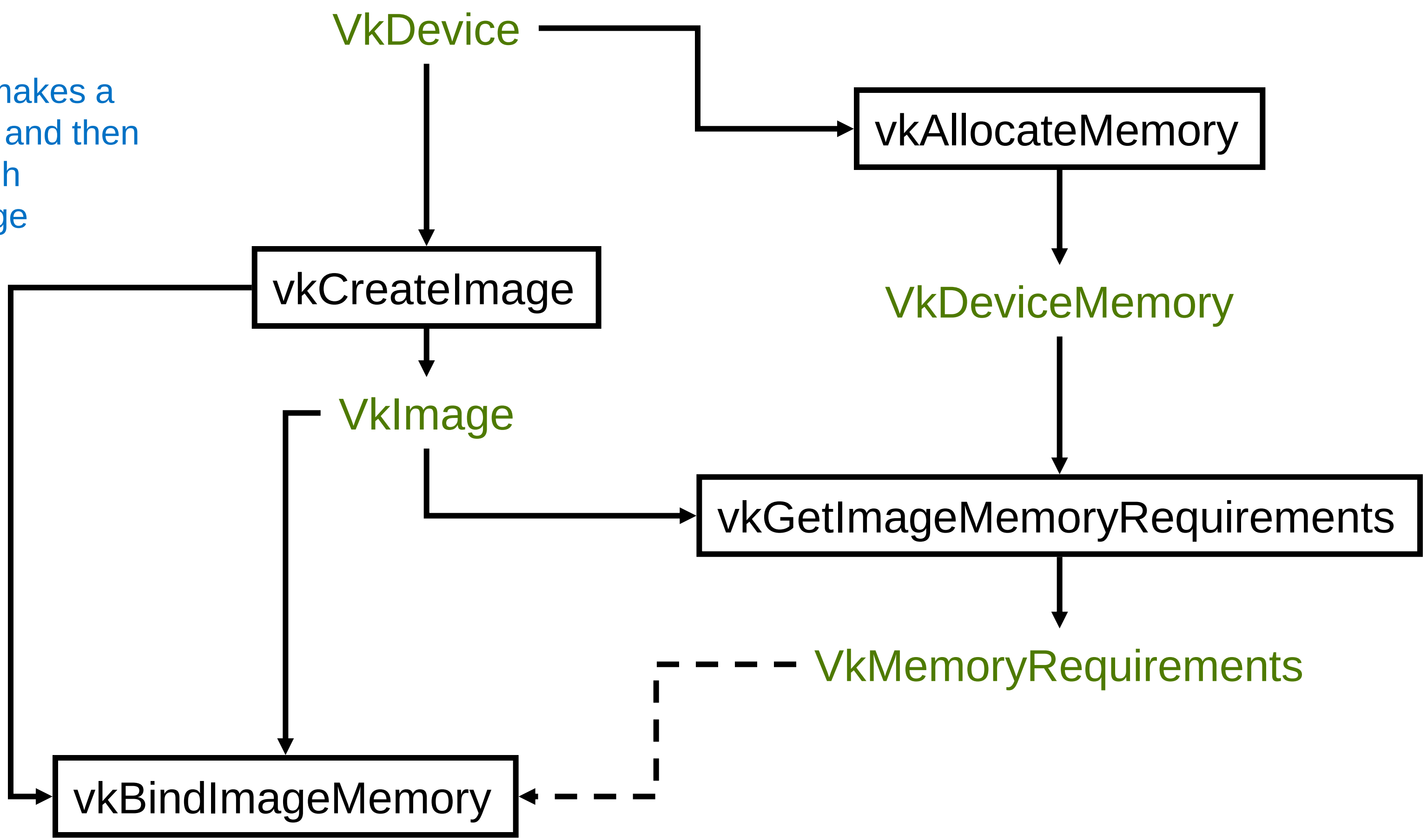
One large device memory allocation, assigned to multiple images

Proper approach!

vkAllocateMemory makes a large allocation with and then sub-allocates enough memory for the image

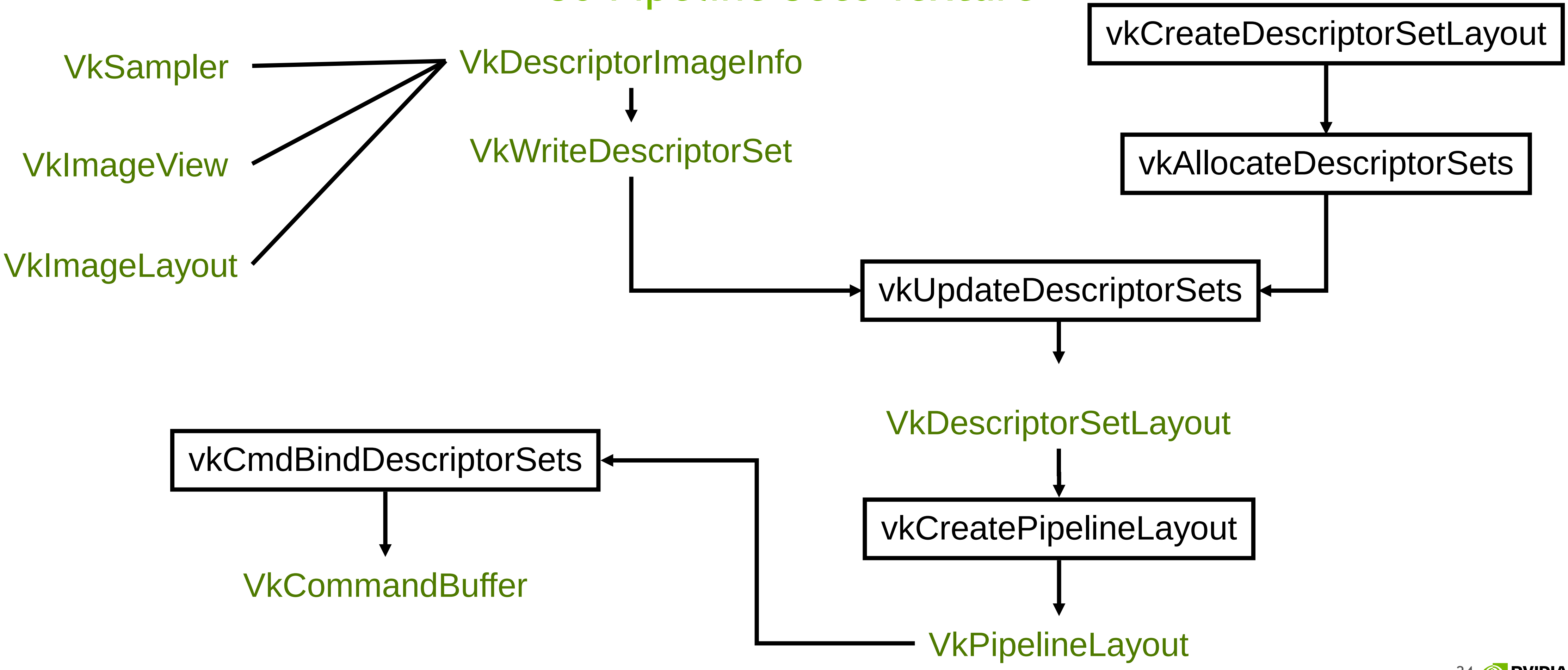
So who writes the sub-allocator?

You do!



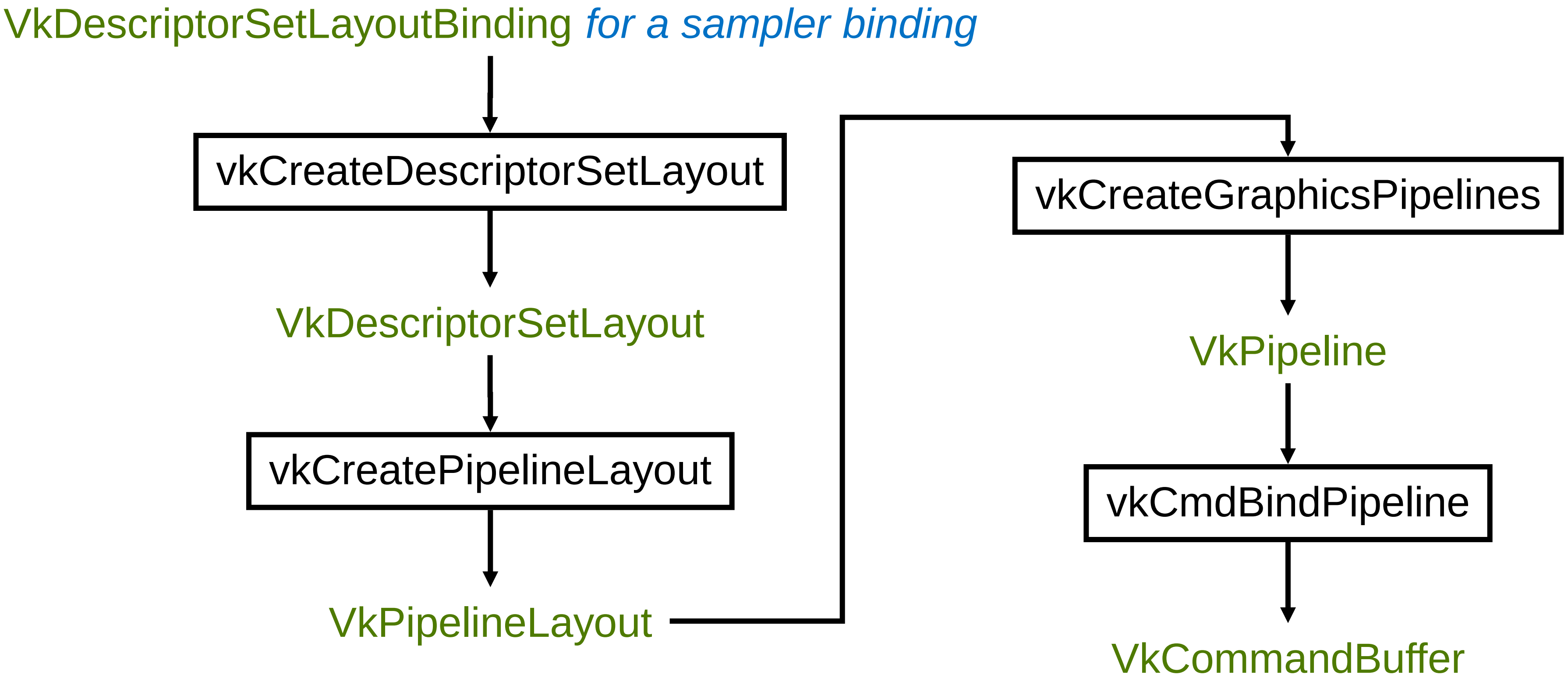
Binding Descriptor Sets for a Texture

So Pipeline Sees Texture



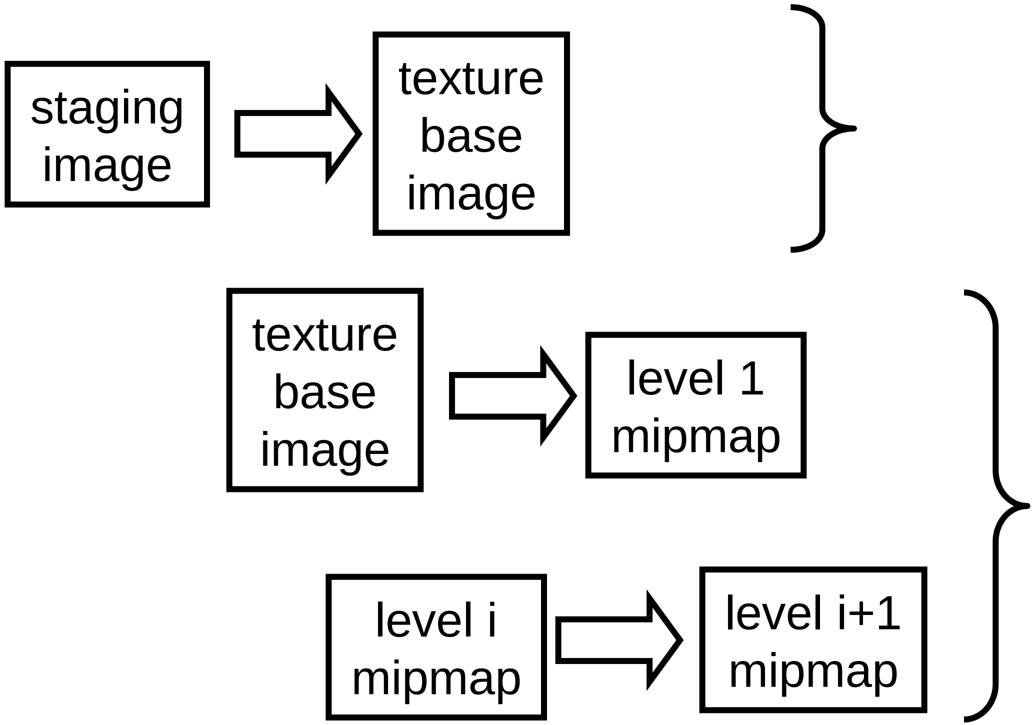
Establishing Sampler Bindings for a Pipeline Object

Vulkan Pipelines Need to Know How They Will Bind Samplers



Base Level Specification + Mipmap Generation

Vulkan Command Buffer Orchestrates Blit + Downsamples



VkCommandBuffer

vkCmdBlitImage

vkCmdPipelineBarrier

for each upper mipmap level

vkCmdBlitImage

vkCmdPipelineBarrier

Binding to a Vertex Array Object (VAO) and Rendering from Vertex Arrays

The OpenGL View

Well traveled path via OpenGL 3.0

```
glBindVertexArray(vertex_array_object);

glDrawElements(GL_TRIANGLES, count, GL_UNSIGNED_INT, indices);
```

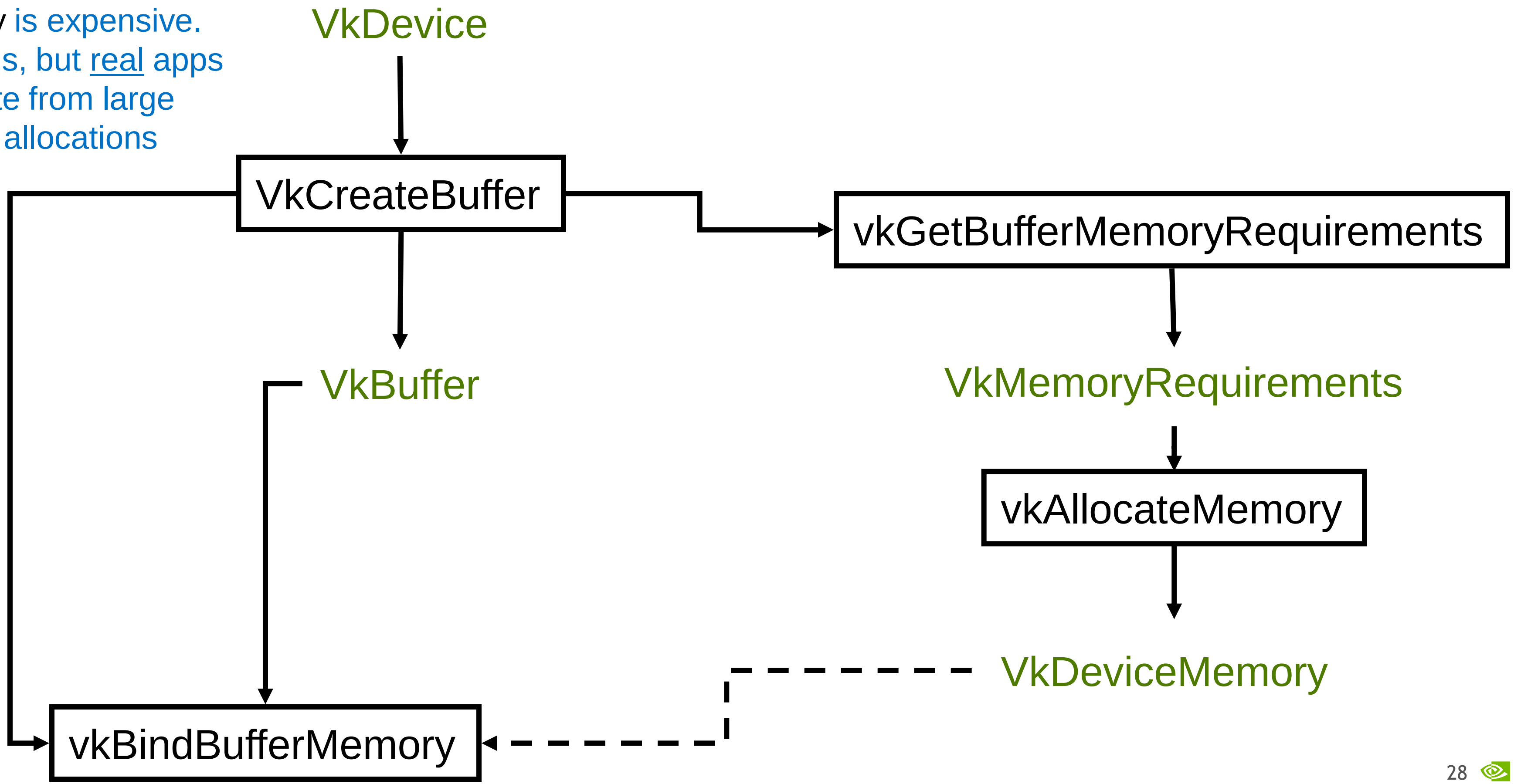
Again, does more than you think it does

Allocating Buffer Memory for VBO

With Vulkan

Naïve approach!

vkAllocateMemory is expensive.
Demos may do this, but <u>real</u> apps should sub-allocate from large VkDeviceMemory allocations



Binding Vertex State To A Pipeline

So Pipeline Sees Vertex Input State

VkVertexInputBindingDescription

VkVertexInputAttributeDescription

VkPipelineVertexInputStateCreateInfo

VkGraphicsPipelineCreateInfo

VkPipelineShaderStageCreateInfo
VkPipelineInputAssemblyStateCreateInfo
VkPipelineViewportStateCreateInfo
VkPipelineRasterizationStateCreateInfo
VkPipelineMultisampleStateCreateInfo
VkPipelineDepthStencilStateCreateInfo
VkPipelineColorBlendStateCreateInfo
VkPipelineDynamicStateCreateInfo

vkCreateGraphicsPipelines

VkGraphicsPipeline

Binding Vertex Buffer For Drawing

Buffer Binding Is Performed With The Command Queue

vkCmdBeginRenderPass

↓

vkCmdBindDescriptorSets

Begin Drawing

↓

vkCmdBindPipeline

For the vertex input state

↓

vkCmdBindVertexBuffers

For the vertex data

↓

vkCmdDrawIndexed

Draw

↓

vkCmdEndRenderPass

End Drawing

Big Features, Ready for Vulkan

Prototype in OpenGL now, Enable in Vulkan next

Compute shaders

Tessellation shaders

Geometry shaders

Sparse resources (textures and buffers)

In OpenGL 4.5 today

Vulkan has the same big features, just different API

Mostly the same GLSL can be used in either case

Thinking about render passes

How do rendering operations affect the retained framebuffer?

- OpenGL just has commands to draw primitives
  - No explicit notion of a render pass in OpenGL API
- Vulkan does have render passes, VkRenderPass
  - Includes notion of sub-passes
  - Designed to facilitate tiling architectures
    - Bounds the lifetime of intermediate framebuffer results
    - Allows iterating over subpasses (chunking) within a render pass on a per tile basis
- In most cases, you won’t need to deal with subpasses

Render Pass

What does a Vulkan Render Pass Provide?

Describes the list attachments the render pass involves

Each attachment can specify

- How the attachment state is initialized (loaded, cleared, dont-care)
- How the attach state is stored (store, or dont-care)
  - Don't-care allows framebuffer intermediates to be discarded
  - E.g. depth buffer not needed after the render pass
- How the layout of attachment could change

Sub-pass dependencies indicate involvement of attachments within a subpass

Render Passes

Monolithic or Could Have Sub-passes

vkCmdBeginRenderPass

vkCmdEndRenderPass

Simple render pass,
no subpasses

vkCmdBeginRenderPass

vkCmdNextSubpass

vkCmdNextSubpass

vkCmdEndRenderPass

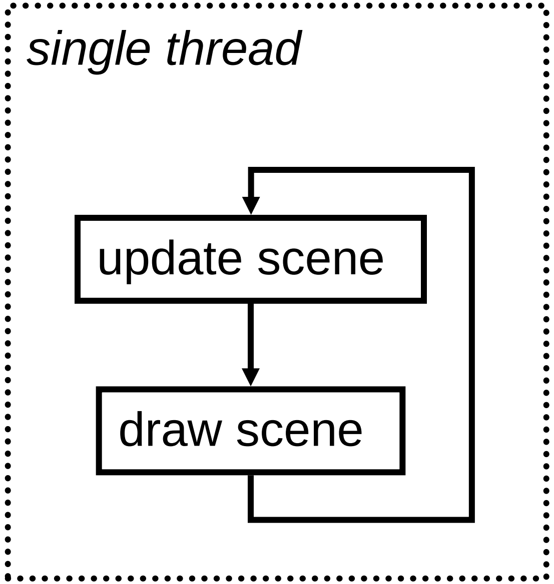
Complex render pass,
with multiple subpasses

Each subpass has its own description and dependencies

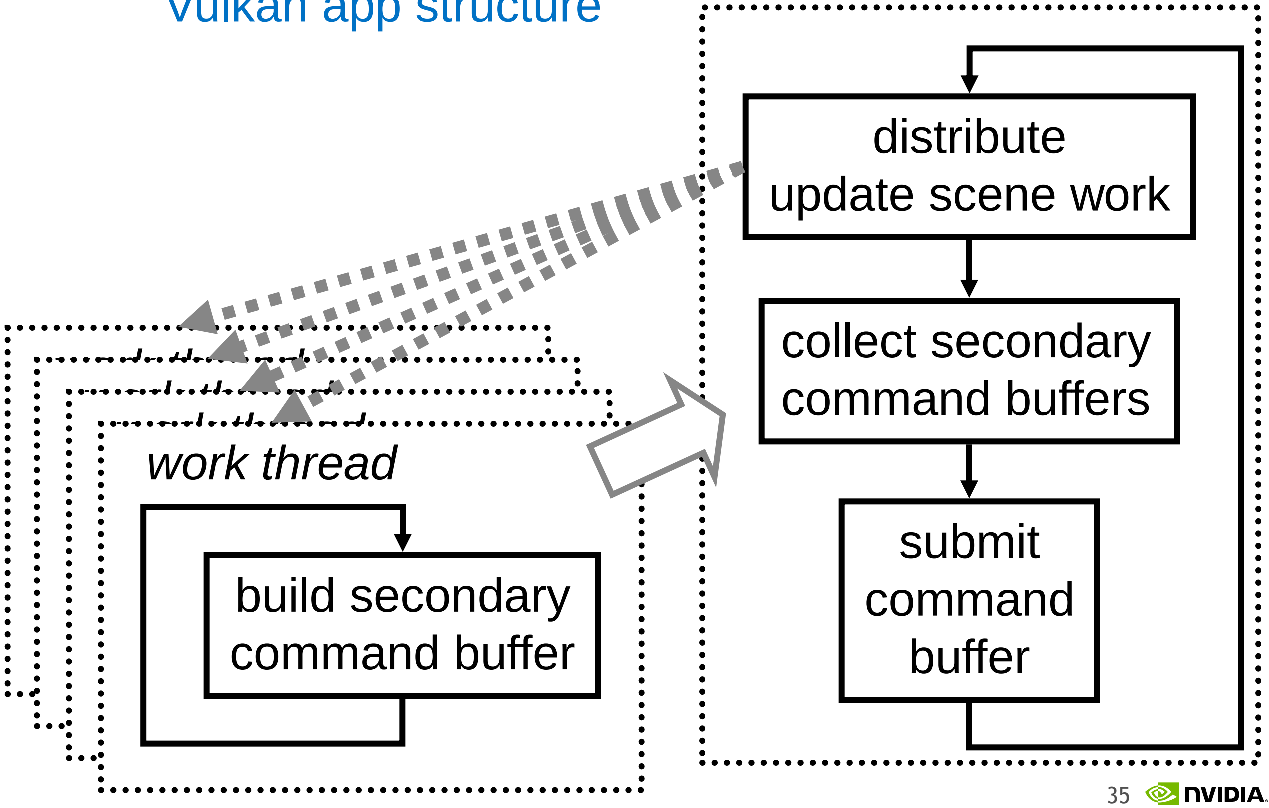
Vulkan Application Structure

Parallel Command Buffer Generation

Traditional single-threaded
OpenGL app structure



Possible multi-threaded
Vulkan app structure



Conclusions

Get Ready for Vulkan

Vulkan is a radical departure from OpenGL

Modernizing your OpenGL code base is definitely good for moving to Vulkan

But it will take more work than that!

Vulkan’s explicitness makes simple operations like a texture bind quite involved

Think about multi-threaded command buffer creation

NVIDIA