VULKAN AND NVIDIA: THE ESSENTIALS

Tristan Lorach  Manager of Developer Technology Group, NVIDIA US

7/25/2016
ANALOGY ON GRAPHIC APIS

(getting ready for my 7 years old son’s questions on my job...)

Car Toy  Lego Kit  Derby Kit
Analogy
Different Valid Approaches

(booring...) (cool... Messes-up the bedroom) (adult supervision required!)

Fixed-function OpenGL

Modern AZDO OpenGL with Programmable Shaders

Vulkan
WHAT IS VULKAN?

...It’s a modern API

- Designed and maintained by Khronos Group
- Designed for high performance on rendering and compute
- [Extremely] low level: no more “baby-sitting” from our driver
  - Manage yourself memory, resource updates; batching; scheduling...
- [Extremely] verbose: Lots of structures to fill with parameters
- close to DX12 design...
- Opposite of OpenGL: Multi-threading friendly: Vulkan will especially shine if multi-threading used
- But still generic enough to work on many HW vendors & platforms
Beneficial Vulkan Scenarios

- Is your graphics work CPU bound?
  - yes
  - Can your graphics creation be parallelized?
    - yes
    - You’ll do whatever it takes to squeeze out Max perf.
    - yes
    - Your graphics platform is fixed
- yes
- Your graphics platform is fixed
  - yes
  - You put a premium on avoiding hitches
    - yes
    - You can manage your graphics resource allocations
      - yes
      - Vulkan friendly
Beneficial Vulkan Scenarios

start

Is your graphics work CPU bound? yes

Can your graphics creation be parallelized? yes

Tired with OpenGL (state-machine) or even D3D? 

Kinda... (it’s a Yes)

Want to learn new stuff? Spend lots of time coding? No sleep?

Alright... (Yes)

You put a premium on avoiding hitches yes

You can manage your graphics resource allocations yes

Vulkan friendly
Unlikely to Benefit
Scenarios to Reconsider Coding to Vulkan

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

Good News in any case: NVIDIA OpenGL driver is great and will always be there!
BIG PICTURE -OPENGL CASE

Application

OpenGL Commands

OpenGL resources

OpenGL Driver

Graphics pipeline States

Resources

Dependencies

Heap

memory

Element buffer (EBO)

Draw Indirect Buffer

Vertex Buffer (VBO)

Uniform Block

Texture Fetch

Image Load/Store

Atomic Counter

Shader Storage

FBO resources (Textures / RB)

Tr. Feedback buffer

GPU

Front-End (decoder)

Vertex Puller (IA)

Vertex Shader

TCS (Tessellation)

Tessellator

TES (Tessellation)

Geometry Shader

Transform Feedback

Rasterization

Fragment Shader

Per-Fragment Ops

Framebuffer
BIG PICTURE - VULKAN

Application

Cmd-buffers / queues

OpenGL

Driver

Fewer translation, Validation checks And internal mgt

Minimal memory management

memory

Element buffer (EBO)
Draw Indirect Buffer
Vertex Buffer (VBO)
Uniform Block
Texture Fetch
Image Load/Store
Atomic Counter
Shader Storage
FBO resources (Textures / RB)
Tr. Feedback buffer

GPU

Front-End (decoder)

Vertex Puller (IA)
Vertex Shader
TCS (Tessellation)
Tessellator
TES (Tessellation)
Geometry Shader
Transform Feedback
Rasterization
Fragment Shader
Per-Fragment Ops
Framebuffer

Pipeline

State

Rende
er Passes

Descriptor Sets

Resources

Heap

Dependencies

Cmd bundles

Push-Buffer (FIFO)
VULKAN OBJECTS: DEVICE

Instance ~⇔~ OpenGL Context

Instance-Layers
• Intercepting API calls for misc. purposes
• Many layers available (api-dump; core/std/parms validation; screenshot...)

Instance-Specific Extensions
• KHR_Surface (for Swap-chains)
• EXT_debug_report
• ...

Exposes some Devices...
HOW DOES IT LOOK?

Instance creation

```c
result = vkEnumerateInstanceLayerProperties(&count, NULL);
assert(result == VK_SUCCESS);
if(count > 0)
{
    instance_layers.resize(count);
    result = vkEnumerateInstanceLayerProperties(&count, &instance_layers[0]);
    //...
}
```

Layer properties

```c
result = vkEnumerateInstanceExtensionProperties(NULL, &count, NULL);
assert(result == VK_SUCCESS);
instance_extensions.resize(count);
extension_names.resize(count);
result = vkEnumerateInstanceExtensionProperties(NULL, &count, &instance_extensions[0]);
```

Extension properties
HOW DOES IT LOOK?

Instance creation

```c
appInfo.pApplicationName = "...";
appInfo.applicationVersion = 1;
appInfo.pEngineName = "...";
appInfo.engineVersion = 1;
appInfo.apiVersion = VK_MAKE_VERSION(1, 0, 0);
instanceInfo.flags = 0;
instanceInfo.pApplicationInfo = &appInfo;
instanceInfo.enabledLayerCount = instance_validation_layers_sz;
instanceInfo.ppEnabledLayerNames = instance_validation_layers_sz > 0 ? &instance_validation_layers[0] : NULL;
instanceInfo.enabledExtensionCount = (uint32_t)extension_names.size();
instanceInfo.ppEnabledExtensionNames = &extension_names[0];
result = vkCreateInstance(&instanceInfo, NULL, &m_instance);
```

Get Instance-Extension’s functions

```c
m_CreateDebugReportCallback = (PFN_vkCreateDebugReportCallbackEXT) vkGetInstanceProcAddr(m_instance, "vkCreateDebugReportCallbackEXT");
m_DestroyDebugReportCallback = (PFN_vkDestroyDebugReportCallbackEXT) vkGetInstanceProcAddr(m_instance, "vkDestroyDebugReportCallbackEXT");
```

e tc
VULKAN OBJECTS: DEVICE

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

Can have many ...

Device(s)

Cmd.Buffer Pool

Command-buffer
  - Barrier synchronization
  - Begin Render-Pass
  - Bind Graphics-pipeline
  - Set misc. dynamic states
  - Bind Vertex/Idx Buffer(s)
  - Update Buffer
  - Bind Descriptor-Set(s)
  - Draw...
  - Execute Commands
  - End Render-Pass

Queue

VkPhysicalDevice

Capabilities
Memory Management
Queues
Objects
Buffers
Images
Sync Primitives
NVIDIA’S VULKAN CAPABILITIES

Properties listed from Physical Device

NVIDIA is almost full featured

  Top to bottom: from GeForce, Quadro down to Tegra

Check [http://vulkan.gpuinfo.org/listreports.php](http://vulkan.gpuinfo.org/listreports.php)
# NVIDIA’S VULKAN CAPABILITIES

## GeForce GTX 980

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>alphaToOne</td>
<td>true</td>
</tr>
<tr>
<td>depthClamp</td>
<td>true</td>
</tr>
<tr>
<td>depthBounds</td>
<td>true</td>
</tr>
<tr>
<td>drawIndirectFirstInstance</td>
<td>true</td>
</tr>
<tr>
<td>dualSrcBlend</td>
<td>true</td>
</tr>
<tr>
<td>fillModelSolid</td>
<td>true</td>
</tr>
<tr>
<td>fragmentStoresAndAtomics</td>
<td>true</td>
</tr>
<tr>
<td>fullDrawIndexedUint32</td>
<td>true</td>
</tr>
<tr>
<td>geometryShader</td>
<td>true</td>
</tr>
<tr>
<td>imageCubeArray</td>
<td>true</td>
</tr>
<tr>
<td>independentBlend</td>
<td>true</td>
</tr>
<tr>
<td>inheritedQueries</td>
<td>true</td>
</tr>
<tr>
<td>largePoints</td>
<td>true</td>
</tr>
<tr>
<td>multDrawIndirect</td>
<td>true</td>
</tr>
<tr>
<td>multiViewport</td>
<td>true</td>
</tr>
<tr>
<td>occlusionQueryPrecise</td>
<td>true</td>
</tr>
<tr>
<td>pipelineStatisticsQuery</td>
<td>true</td>
</tr>
<tr>
<td>robustBufferAccess</td>
<td>true</td>
</tr>
<tr>
<td>sampleRateShading</td>
<td>true</td>
</tr>
<tr>
<td>samplerAnisotropy</td>
<td>true</td>
</tr>
<tr>
<td>shaderClipDistance</td>
<td>true</td>
</tr>
<tr>
<td>shaderGutDistance</td>
<td>true</td>
</tr>
<tr>
<td>shaderFloat64</td>
<td>true</td>
</tr>
<tr>
<td>shaderImageGatherExtended</td>
<td>false</td>
</tr>
<tr>
<td>shaderInt16</td>
<td>false</td>
</tr>
<tr>
<td>shaderInt46</td>
<td>false</td>
</tr>
<tr>
<td>shaderImageSampleResidency</td>
<td>true</td>
</tr>
<tr>
<td>shaderInt46</td>
<td>false</td>
</tr>
<tr>
<td>shaderInt64</td>
<td>false</td>
</tr>
</tbody>
</table>

## Tegra X1 & K1

<table>
<thead>
<tr>
<th>Feature</th>
<th>GeForce GTX 980</th>
<th>Tegra X1 &amp; K1</th>
<th>Report 3</th>
<th>Report 78</th>
</tr>
</thead>
<tbody>
<tr>
<td>alphaToOne</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>depthClamp</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>depthBounds</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>drawIndirectFirstInstance</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>dualSrcBlend</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>fillModelSolid</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>fragmentStoresAndAtomics</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>fullDrawIndexedUint32</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>geometryShader</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>imageCubeArray</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>independentBlend</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>inheritedQueries</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>largePoints</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>multDrawIndirect</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>multiViewport</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>occlusionQueryPrecise</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>pipelineStatisticsQuery</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>robustBufferAccess</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>sampleRateShading</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>samplerAnisotropy</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>shaderClipDistance</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>shaderGutDistance</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>shaderFloat64</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>shaderImageGatherExtended</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>shaderInt16</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>shaderInt46</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>shaderImageSampleResidency</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>shaderInt46</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>shaderInt64</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>shaderImageSampleResidency</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>shaderInt46</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

**Note:** The table above shows the Vulkan capabilities of NVIDIA’s hardware. The truth values indicate whether a specific feature is supported by the hardware. The table compares the GeForce GTX 980 with Tegra X1 & K1, highlighting differences in their support for various Vulkan features.
HOW DOES IT LOOK?
Device creation - enumerate physical devices; gather properties

```c
result = vkEnumeratePhysicalDevices(m_instance, &count, NULL);
physical_devices.resize(count);
result = vkEnumeratePhysicalDevices(m_instance, &count, &physical_devices[0]);
for(int j=0; j<physical_devices.size(); j++)
{
    vkGetPhysicalDeviceProperties(physical_devices[j], &m_gpu.properties);
    //...
    vkGetPhysicalDeviceMemoryProperties(physical_devices[j], &m_gpu.memoryProperties);
    //...
    result = vkEnumerateDeviceLayerProperties(physical_devices[j], &count, NULL);
    if(count > 0)
    {
        device_layers.resize(count);
        result = vkEnumerateDeviceLayerProperties(physical_devices[j], &count, &device_layers[0]);
        //...
    }
}
result = vkEnumerateDeviceExtensionProperties(physical_devices[j], NULL, &count, NULL);
device_extensions.resize(count);
result = vkEnumerateDeviceExtensionProperties(physical_devices[j], NULL, &count, &device_extensions[0]);
//...
```
queueInfo.queueFamilyIndex = queueFamilyIndex;
queueInfo.queueCount = m_gpu.queueProperties[queueFamilyIndex].queueCount;
deInfo.createDeviceInfoCount = 1;
deInfo.pCreateInfos = &queueInfo;
deInfo.enabledLayerCount = instance_validation_layers_sz;
deInfo.ppEnabledLayerNames = instance_validation_layers_sz > 0 ? &instance_validation_layers[0] : NULL;
deInfo.enabledExtensionCount = (uint32_t)extension_names.size();
deInfo.ppEnabledExtensionNames = &extension_names[0];
result = ::vkCreateDevice(m_gpu.device, &devInfo, NULL, &m_device);
Command queue was hidden in OpenGL Context... now explicitly declared

Multiple threads can submit work to a queue (or queues)!

Queues accept GPU work via CommandBuffer submissions

Few operations available around Queues: “submit work” and “wait for idle”

Queue submissions can include sync primitives for the queue to:

- **Wait** upon before processing the submitted work
- **Signal** when the work in this submission is completed

Queue “families” can accept different types of work, e.g.

NVIDIA exposes 2 families: 1+16 Queues

16 for all available types of work

1 for transfer operations only (Copy Engine)
// at Init time (after Device creation...)
vkGetDeviceQueue(m_device, 0, 0, &m_queue);

::VkSubmitInfo submitInfo = { VK_STRUCTURE_TYPE_SUBMIT_INFO, NULL };
submitInfo.waitSemaphoreCount = 0;
submitInfo.pWaitSemaphores = NULL;
submitInfo.commandBufferCount = 1;
submitInfo.pCommandBuffers = &cmd;
submitInfo.signalSemaphoreCount = 0;
submitInfo.pSignalSemaphores = NULL;

CHECK(:,vkQueueSubmit(m_queue, 1, &submitInfo, VK_NULL_HANDLE));

eyeDeviceWaitIdle(m_device);
VULKAN COMPONENTS

Heap

Memory

Heap

Image View

Framebuffer

Render-Pass

Graphics pipeline

Buffer

Descriptor-Set

DescriptorSet Pool

2ndary Command-buffer

...
SYNCHRONIZATION

**events and barriers**

- used to synchronize work within a command buffer or sequence of command buffers submitted to a single queue

**semaphores**

- used to synchronize work across queues or across coarse-grained submissions to a single queue

**fences**

- used to synchronize work between the device and the host.
COMMAND-BUFFERS

Vulkan Rendering \(\rightarrow\) Command-Buffers

Close to what GPU will get at Front-End (FIFO)

Minor translation & optimization from the Driver prior to sending to the GPU

Each can be created either for one shot or for multiple frames/submissions

Cannot Cmd-Buffers from GPU (command-lists can): API calls to `vkCmd...()` between Begin & End

Multi-threading friendly (!)

Primary Cmd-Buffer can call many 2\textsuperscript{ndary} Cmd-Buffers
**COMMAND-BUFFERS AND MULTI-THREADING**

- **Main thread (Busy)**
  - Game Work
  - Thread Coordination
  - cmd. Buffer Pool
  - Create 1<sup>ary</sup> Cmd Buffer
  - Collect
  - 1<sup>ary</sup> Cmd calls 2<sup>ary</sup> ones
  - Submit to Q
  - Swapping

- **Thread 1 (Busy)**
  - Update Work
  - cmd. Buffer Pool
  - Create 2<sup>ary</sup> Cmd Buffer
  - Feed Cmd Buffers
  - Give out Cmd Buffers

- **Thread 2 (Busy)**
  - Update Work
  - cmd. Buffer Pool
  - Create 2<sup>ary</sup> Cmd Buffer
  - Feed Cmd Buffers
  - Give out Cmd Buffers

- **Thread 3 (Busy)**
  - Update Work
  - cmd. Buffer Pool
  - Create 2<sup>ary</sup> Cmd Buffer
  - Feed Cmd Buffers
  - Give out Cmd Buffers

- **Thread 4 (Busy)**
  - Update Work
  - cmd. Buffer Pool
  - Create 2<sup>ary</sup> Cmd Buffer
  - Feed Cmd Buffers
  - Give out Cmd Buffers

---

Command Buffer Pool local to the thread
To prevent conflicts in concurrent access
Must not recycle a `CommandBuffer` for rewriting until it is no longer in flight (in flight == GPU still consuming it on its side).

But we can’t flush the `queue` each frame: would break parallelism!

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

---

**COMMAND BUFFER THREAD SAFETY**

**App Submissions to the Queue**

1. CommandBuffer
2. CommandBuffer
3. CommandBuffer

**Fence A**

1. CommandBuffer
2. CommandBuffer
3. CommandBuffer

**Fence B**

1. CommandBuffer
2. CommandBuffer

**Fence A Signaled to App**

**Rewrite command buffer**

**GPU Consumes Queue**
Threads can have more than 1 Command Pool

**Ring-buffer**: One Command-Pool per Frame

When the frame is no longer in flight (Using Fences):

Simply reset the whole Pool

---

**Threads AND COMMAND POOLS**

<table>
<thead>
<tr>
<th>Frame N-2</th>
<th>Frame N-1</th>
<th>Frame N</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thread 1</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CommandPool</td>
<td>CommandPool</td>
<td>CommandPool</td>
</tr>
<tr>
<td>Command Buffer</td>
<td>Command Buffer</td>
<td>Command Buffer</td>
</tr>
<tr>
<td>Command Buffer</td>
<td>Command Buffer</td>
<td>Command Buffer</td>
</tr>
<tr>
<td><strong>Thread 2</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CommandPool</td>
<td>CommandPool</td>
<td>CommandPool</td>
</tr>
<tr>
<td>Command Buffer</td>
<td>Command Buffer</td>
<td>Command Buffer</td>
</tr>
<tr>
<td>Command Buffer</td>
<td>Command Buffer</td>
<td>Command Buffer</td>
</tr>
</tbody>
</table>
 HOW DOES IT LOOK ?  

Command-Buffers

```cpp
m_perThreadData = new PerThreadData;
VkCommandPoolCreateInfo cmdPoolInfo = { VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO };
cmdPoolInfo.queueFamilyIndex = 0;
cmdPoolInfo.flags = VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT;
result = vkCreateCommandPool(nvk.m_device, &cmdPoolInfo, NULL, &m_perThreadData->m_cmdPoolStatic);

nvk.vkResetCommandPool(m_perThreadData->m_curCmdPoolDynamic, 0);
m_cmdScene = nvk.vkAllocateCommandBuffer(m_perThreadData->m_curCmdPoolDynamic, true);
nvk.vkBeginCommandBuffer(m_cmdScene, false, NVK::VkCommandBufferInheritanceInfo(renderPass, 0, framebuffer, VK_FALSE, 0, 0));
vCmdUpdateBuffer (m_cmdScene, m_matrix.buffer, 0, sizeof(g_globalMatrices), (uint32_t*)&g_globalMatrices);
vCmdBeginRenderPass(m_cmdScene, 
    NVK::VkRenderPassBeginInfo( 
        renderPass, framebuffer, viewRect, 
        NVK::VkClearValue(NVK::VkClearColorValue(0.0f, 0.0f, 0.0f, 1.0f)), 
        (NVK::VkClearDepthStencilValue(1.0, 0))));
vCmdSetViewport( m_cmdScene, 0, 1, NVK::VkViewport(0.0, 0.0, w, h, 0.0f, 1.0f) );
vCmdSetScissor( m_cmdScene, 0, 1, NVK::VkRect2D(0.0, 0.0, w, h ) );

vkCmdExecuteCommands(m_cmdScene, 1, &m_cmdBufferBgn); //...
vCmdEndRenderPass(m_cmdScene);
vCmdEndCommandBuffer(m_cmdScene);
```

**Note**: helpers (structs wrappers) to use Constructors & functors for compact data declaration
See NVK.h/cpp in https://github.com/nvpro-samples
GRAPHICS PIPELINE

Snapshot of all States

Including **Shaders**

Pre-compiled & Immutable

**Ideally**: done at Initialization time

Ok at render-time *if* using the **Pipeline-Cache**

Prevents validation overhead during rendering loop

Some Render-states can be excluded from it: they become “Dynamic” States

- **depthClipEnable**
- **rasterizerDiscardEnable**
- **fillMode**
- **cullMode**
- **frontFace**
- **depthBiasEnable**
- **depthBias**
- **depthBiasClamp**
- **slopeScaledDepthBias**
- **lineWidth**
GRAPHICS PIPELINE

Graphics Pipeline must be consistent with shaders

No “introspection”, so everything known & prepared in advance

Vertex Input:

- tells how Attributes: Locations are attached to which Vertex Buffer at which offset

Pipeline Layout:

- Tells how to map Sets and Bindings for the shaders at each stage (Vtx, Fragment, Geom...)

GLSL Code

```glsl
layout(std140, set=0, binding=0) uniform A { ... }
layout(std140, set=0, binding=1) uniform B { ... }
layout(std140, set=1, binding=2) uniform C { ... }
...
layout(location=0) in vec3 pos;
layout(location=1) in vec3 N;
... void main() { ... }
```
HOW DOES IT LOOK?

Graphics pipeline - setup

```cpp
m_descriptorSetLayouts[DSET_GLOBAL] = nvk.vkCreateDescriptorSetLayout(
    NVK::VkDescriptorSetLayoutCreateInfo(NVK::VkDescriptorSetLayoutBinding(
        BINDING_MATRIX, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, 1, VK_SHADER_STAGE_VERTEX_BIT) // BINDING_MATRIX
        (BINDING_CUBE_TEX, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, 1, VK_SHADER_STAGE_FRAGMENT_BIT) // BINDING_LIGHT
    )
);
```

```cpp
m_descriptorSetLayouts[DSET_OBJECT] = nvk.vkCreateDescriptorSetLayout(
    NVK::VkDescriptorSetLayoutCreateInfo(NVK::VkDescriptorSetLayoutBinding(
        BINDING_MATRIX_OBJ, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER
        (BINDING_MATERIAL, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER)
    )
);
```

```cpp
m_pipelineLayout = nvk.vkCreatePipelineLayout(
    NVK::VkPipelineDynamicStateCreateInfo dynamicStateCreateInfo(
        NVK::VkDynamicState
            (VK_DYNAMIC_STATE_VIEWPORT)
            (VK_DYNAMIC_STATE_SCISSOR)
            (VK_DYNAMIC_STATE_LINE_WIDTH)
            (VK_DYNAMIC_STATE_DEPTH_BIAS)
    );
```

```cpp
NVK::VkPipelineRasterizationStateCreateInfo vkPipelineRasterStateCreateInfo(
    VK_TRUE, /*depthClipEnable*/ VK_FALSE, //rasterizerDiscardEnable
    VK_POLYGON_MODE_FILL, /*fillMode*/ VK_CULL_MODE_NONE, //cullMode
    VK_FRONT_FACE_COUNTER_CLOCKWISE, //frontFace
    VK_TRUE, //depthBiasEnable
    0.0, /*depthBias*/ 0.0, //depthBiasClamp
    0.0, /*slopeScaledDepthBias*/ 1.0 /*lineWidth*/
);```
HOW DOES IT LOOK?

Graphics pipeline creation (compact notation)

```c
m_pipelineGrid = nvk.vkCreateGraphicsPipeline(NVK::VkGraphicsPipelineCreateInfo
(m.pipelineLayout, 0)
(NVK::VkPipelineVertexInputStateCreateInfo,
 NVK::VkVertexInputBindingDescription (0/*binding*/, sizeof(Attrs)*"stride"),
 NVK::VkVertexInputAttributeDescription (0/*location*/, 0/*binding*/, VK_FORMAT_R32G32B32_SFLOAT, 0/*offsetbytes*/)
(1/*location*/, 0/*binding*/, VK_FORMAT_R32G32B32_SFLOAT, 12/*offsetbytes*/))
(NVK::VkPipelineInputAssemblyStateCreateInfo(VK_PRIMITIVETopology TriangleList, VK_FALSE))
(NVK::VkPipelineShadingStageCreateInfo(VK_SHADER_STAGE_VERTEX_BIT, nvk.vkCreateShaderModule(spv_GLSL_bgdn_ert_c_str(), spv_GLSL_bgdn_ert_size()), "main")
(NVK::VkPipelineViewportStateCreateInfo)
(NVK::VkPipelineRasterizationStateCreateInfo)
(NVK::VkPipelineMultisampleStateCreateInfo)
(NVK::VkPipelineColorBlendStateCreateInfo)
(dynamicStateCreateInfo)
```

```c
m_pipelineCubeBgd = nvk.vkCreateGraphicsPipeline(NVK::VkGraphicsPipelineCreateInfo
(m.pipelineLayout, m.scenePass)
(NVK::VkPipelineVertexInputStateCreateInfo,
 NVK::VkVertexInputBindingDescription (0/*binding*/, sizeof(Attrs)*"stride"),
 NVK::VkVertexInputAttributeDescription (0/*location*/, 0/*binding*/, VK_FORMAT_R32G32B32_SFLOAT, 0/*offsetbytes*/)
(1/*location*/, 0/*binding*/, VK_FORMAT_R32G32B32_SFLOAT, 12/*offsetbytes*/))
(NVK::VkPipelineInputAssemblyStateCreateInfo(VK_PRIMITIVETopology TriangleList, VK_FALSE))
(NVK::VkPipelineShadingStageCreateInfo(VK_SHADER_STAGE_VERTEX_BIT, nvk.vkCreateShaderModule(spv_GLSL_bgdn_ert_c_str(), spv_GLSL_bgdn_ert_size()), "main")
(NVK::VkPipelineViewportStateCreateInfo)
(NVK::VkPipelineRasterizationStateCreateInfo)
(NVK::VkPipelineMultisampleStateCreateInfo)
(NVK::VkPipelineShadingStageCreateInfo(VK_SHADER_STAGE_FRAGMENT_BIT, nvk.vkCreateShaderModule(spv_GLSL_bgdn_frg_c_str(), spv_GLSL_bgdn_frg_size()), "main")
(NVK::VkPipelineColorBlendStateCreateInfo)
(NVK::VkPipelineDepthStencilStateCreateInfo)
(dynamStateCreateInfo)
```
BUFFERS

Highly Heterogenous. Most often used for:

- **Index/Vertex Buffers**
- **Uniform Buffers** (Matrices, material parameters...)

**Vulkan Object: Must be bound to some Device Memory**

- Can be **CPU accessible** memory (mappable)
- Can be **CPU cached**
- Can be **GPU accessible** only: need a “Staging Buffer” to write into it
- But most Efficient

(More on Device Memory later...)
COMMAND-BUFFERS: UPDATE/PUSH CONSTANTS

2 more ways to update constants/uniforms for Shaders from the Command-Buffer

**Update-Buffer**: prior to Render-Pass: can target any Buffer bound by Descriptor Sets

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

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

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

New values appended “in-band”: in the Command-Buffer

Efficient; but good for small amount of values
Buffers - copy data in a buffer

```
VkBuffer buffer; // may not be accessible from host
VkBufferCreateInfo bufferInfo(size, usage);
CHECK((::vkCreateBuffer(m_device, &bufferInfo, NULL, &buffer) ));
bufferMem = allocMemAndBindObject(buffer, memProps);
// Create staging buffer: accessible from host
VkBuffer bufferStage;
VkBufferCreateInfo bufferStageInfo(size, VK_BUFFER_USAGE_TRANSFER_SRC_BIT);
CHECK((::vkCreateBuffer(m_device, &bufferStageInfo, NULL, &bufferStage) ));
// Allocate and bind to the buffer
::VkDeviceMemory bufferStageMem;
bufferStageMem = allocMemAndBindObject(bufferStage, (::VkFlags)VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT);
vkMemcpy(bufferStageMem, data, size);
::VkCommandBuffer cmd;
cmd = vkAllocateCommandBuffer(cmdPool, true);
vkBeginCommandBuffer(cmd, true);
  vkCmdCopyBuffer(cmd, bufferStage, buffer, size, offset);
vkEndCommandBuffer(cmd);
::VkSubmitInfo submitInfo = { VK_STRUCTURE_TYPE_SUBMIT_INFO, NULL }; submitInfo.waitSemaphoreCount = 0;
submitInfo.pWaitSemaphores = NULL;
submitInfo.commandBufferCount = 1;
submitInfo.pCommandBuffers = &cmd;
submitInfo.signalSemaphoreCount = 0;
submitInfo.pSignalSemaphores = NULL;
CHECK((::vkQueueSubmit(m_queue, 1, &submitInfo, VK_NULL_HANDLE) ));
vkDeviceWaitIdle(m_device);
// release stuff
vkFreeCommandBuffer(cmdPool, cmd);
vkDestroyBuffer(bufferStage);
::vkFreeMemory(m_device, bufferStageMem, NULL);
```

**Create Buffer**

**Create staging buffer**

**Bind objects to some memory**

**Command buffer for copy**

**Enqueue command buffer**

**Note**: could be put with other graphic commands for better efficiency
VULKAN COMPONENTS

Heap

Memory

Device

Queue

Image View

Framebuffer

Render-Pass

Graphics pipeline

Buffer

Descriptor-Set

DescriptorSet Pool

2ndary Command-buffer

...
IMAGES AND IMAGEVIEW

Images represent all kind of ‘pixel-like’ arrays

Textures: Color or Depth-Stencil

Render targets: Color and Depth-Stencil

Even Compute data

Shader Load/Store (imgLoadStore)

ImageView required to expose Images properly when specific format required

For Shaders

For Framebuffers

Heap

Buffer

Cmd.Buffer Pool

Command-buffer

Barrier synchronization

Begin Render-Pass

Bind Graphics-pipeline

Set misc. dynamic states

Bind Vertex/Idx Buffer(s)

Update Buffer

Bind Descriptor-Set(s)

Draw...

Execute Commands

End Render-Pass

Device

Queue
HOW DOES IT LOOK?

Simple texture creation

Way more complex than OpenGL!

- Load image
- Create an Image (1D/2D/3D/Cube...)
- Create an Image-View
- Aggregate layers/mipmap layers info (offsets, sizes) in a structure (VkBufferImageCopy)
- Aggregate layers & mipmap data to contiguous memory

- Create staging buffer + bind memory + copy data in it
- Use command-buffer to copy to the image: layers and mipmaps
  - Layout transition of image for copy
  - vkCmdCopyBufferToImage
  - Layout transition of image for use by shader
- Enqueue command buffer and execute
HOW DOES IT LOOK?

Simple texture creation

```c
// Code snippet for texture creation
```
Each DescriptorSet holds *references* to some resources. *Descriptor-Set-Layout* defines how resources must be put together in a DescriptorSet.

Command buffers can then efficiently bind any or them. They must *match what shaders of each stage expect!*
HOW DOES IT LOOK?

Descriptor Set - initialization

```c
m_descriptorSetLayouts[DSET_GLOBAL] = nvk.vkCreateDescriptorSetLayout(
    NVK::VkDescriptorSetLayoutCreateInfo(NVK::VkDescriptorSetLayoutBinding
        (BINDING_MATRIX, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, 1, VK_SHADER_STAGE_VERTEX_BIT) // BINDING_MATRIX
        (BINDING_CUBETEX, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, 1, VK_SHADER_STAGE_FRAGMENT_BIT)
    )
);

m_descPool = nvk.vkCreateDescriptorPool(NVK::VkDescriptorPoolCreateInfo(
    10/*maxSets*/, NVK::VkDescriptorPoolSize(VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, 5)
        (VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC, 5)
        (VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, 5))
);

nvk.vkAllocateDescriptorSets(NVK::VkDescriptorSetAllocateInfo
    (m_descPool, 1, m_descriptorSetLayouts), &m_descriptorSetGlobal);

NVK::VkDescriptorBufferInfo descBuffer = NVK::VkDescriptorBufferInfo(m_matrix.buffer, 0, m_matrix.Sz);
nvk.vkUpdateDescriptorSets(NVK::VkWriteDescriptorSet
    (m_descriptorSetGlobal, BINDING_MATRIX, 0, descBuffer, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER)
    (m_descriptorSetGlobal, BINDING_CUBETEX, 0, cubeTextureSamplerAndView, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER)
);
```
HOW DOES IT LOOK ?
Descriptor Set - using it

```c
nvk.vkBeginCommandBuffer(m_cmdBufferGrid, false,
    NVK::VkCommandBufferInheritanceInfo(renderPass, 0, framebuffer, 0/*occlusionQueryEnable*/,
    0/*queryFlags*/, 0/*pipelineStatistics*/));
{
    vkCmdBindPipeline(m_cmdBufferGrid, VK_PIPELINE_BIND_POINT_GRAPHICS, m_pipelineGrid);
    vkCmdSetDepthBias(m_cmdBufferGrid, 0.0f, 0.0f, 0.0f); // offset raster
    vkCmdSetLineWidth(m_cmdBufferGrid, lineWidth);
    VkDeviceSize vboffsets[1] = {0};
    vkCmdBindVertexBuffers(m_cmdBufferGrid, 0, 1, &m_gridBuffer.buffer, vboffsets);
    
    vkCmdBindDescriptorSets(m_cmdBufferGrid, VK_PIPELINE_BIND_POINT_GRAPHICS, m_pipelineLayout,
        DSET_GLOBAL, 1, &m_descriptorSetGlobal, 0, NULL);

    vkCmdDraw(m_cmdBufferGrid, GRIDDEF * 4, 1, 0, 0);
}

vkEndCommandBuffer(m_cmdBufferGrid);
```
DESRIPTOR SETS AND MULTI-THREADING

Main thread
- Game Work
- Thread Coordination
- Descriptor Pool
- Allocate Desc. Sets
- Update resources in Desc. Sets
- ... more Vulkan work ...
- synchronize
- Swapping

Thread 1
- Update Work
- Descriptor Pool
- Allocate Desc. Sets
- Update resources in Desc. Sets
- ... more Vulkan work ...

Thread 2
- Update Work
- Descriptor Pool
- Allocate Desc. Sets
- Update resources in Desc. Sets
- ... more Vulkan work ...

Thread 1
- Update Work
- Descriptor Pool
- Allocate Desc. Sets
- Update resources in Desc. Sets
- ... more Vulkan work ...

Thread 2
- Update Work
- Descriptor Pool
- Allocate Desc. Sets
- Update resources in Desc. Sets
- ... more Vulkan work ...

! Descriptor Pool local to the thread!
**VULKAN COMPONENTS**

Can use many if compatibles

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

Renderpass Creation ('compacted' notation for Vulkan structures)

```c
NVK::VkRenderPassCreateInfo rpinfo = NVK::VkRenderPassCreateInfo(
    NVK::VkAttachmentDescription(
        VK_FORMAT_R8G8B8A8_UNORM, (VkSampleCountFlagBits)MSAA,
        VK_ATTACHMENT_LOAD_OP_CLEAR, VK_ATTACHMENT_STORE_OP_STORE,
        VK_ATTACHMENT_LOAD_OP_DONT_CARE, VK_ATTACHMENT_STORE_OP_DONT_CARE,
        VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL
    ),
    VK_FORMAT_D24_UNORM_S8_UINT, (VkSampleCountFlagBits)MSAA,
    VK_ATTACHMENT_LOAD_OP_CLEAR, VK_ATTACHMENT_STORE_OP_STORE,
    VK_ATTACHMENT_LOAD_OP_CLEAR, VK_ATTACHMENT_STORE_OP_STORE,
    VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL
),
    VK_FORMAT_R8G8B8A8_UNORM, (VkSampleCountFlagBits)1,
    VK_ATTACHMENT_LOAD_OP_DONT_CARE, VK_ATTACHMENT_STORE_OP_DONT_CARE,
    VK_ATTACHMENT_LOAD_OP_DONT_CARE, VK_ATTACHMENT_STORE_OP_DONT_CARE,
    VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL
),
    // Easy way
    NVK::VkSubpassDescription(
        VK_PIPELINE_BIND_POINT_GRAPHICS,
        NVK::VkAttachmentReference(),
        NVK::VkAttachmentReference(0/*attachment*/), VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL/*layout*/),
        NVK::VkAttachmentReference(1/*attachment*/), VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL/*layout*/),
        NVK::VkAttachmentReference(2/*attachment*/), VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL/*layout*/),
        NVK::VkAttachmentReference(1/*attachment*/), VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL/*layout*/),
        NVK::uInt32Array(),
    0
),
    NVK::VkSubpassDependency(/*NONE*/),
);

m_scenePass = nvk.vkCreateRenderPass(rpinfo);
```
Framebuffer Creation and use

```cpp
m_colorImage[i].img = nvk.createImage2D(width, height, m_colorImage[i].imgMem, VK_FORMAT_R8G8B8A8_UNORM, VK_SAMP
m_colorImage[i].imgView = nvk.vkCreateImageView(NVK::VkImageViewCreateInfo(
    m_colorImage[i].img, /*image*/ VK_IMAGE_VIEW_TYPE_2D, /*viewType*/ VK_FORMAT_R8G8B8A8_UNORM, /*
    NVK::VkComponentMapping(/*channels*/ NVK::VkImageSubresourceRange() /*subresourceRange*/));

// ...

m_framebuffers[i] = nvk.vkCreateFramebuffer(
    NVK::VkFramebufferCreateInfo(
        m_scenePass, //renderPass
        width, height, 1, //width, height, layers
        (m_colorImageMS[i].imgView),
        (m_DSTImageMS[i].imgView),
        (m_colorImage[i].imgView)
    );

nvk.vkBeginCommandBuffer(m_cmdScene, false,
    NVK::VkCommandBufferInheritanceInfo(renderPass, 0, framebuffer, VK_FALSE, 0, 0));

vkCmdBeginRenderPass(m_cmdScene,
    NVK::VkRenderPassBeginInfo(
        renderPass, framebuffer, viewRect,
        NVK::VkClearValue(NVK::VkClearColorValue(0.0f, 0.0f, 0.0f, 1.0f))
        (NVK::VkClearDepthStencilValue(1.0, 0)), VK_SUBPASS_CONTENTS_INLINE);

vkCmdSetViewport(m_cmdScene, 0, 1,
    NVK::VkViewport(0.0, 0.0, w, h, 0.0f, 1.0f);

vkCmdSetScissor(m_cmdScene, 0, 1,
    NVK::VkRect2D(0.0, 0.0, w, h));
```
VULKAN COMPONENTS

Heap 1
Heap 2

Memory (Vid)

Memory (Sys)

Image View
Framebuffer
Render-Pass
Graphics pipeline
Buffer
Descriptor-Set
DescriptorSet Pool

Image

2ndary Command-buffer
...
...

Command-buffer
Barrier synchronization
Begin Render-Pass
Bind Graphics-pipeline
Set misc. dynamic states
Bind Vertex/Idx Buffer(s)
Update Buffer
Bind Descriptor-Set(s)
Draw...
Execute Commands
End Render-Pass

Device

Queue

Cmd.Buffer Pool

Heap 2

Heap 1

Memory (Vid)

Memory (Sys)
MEMORY ⇔ VULKAN OBJECTS

Vulkan Objects **referring to buffer(s) of data** need binding to memory

- Vertex/Index Buffers; Uniform Buffers; Images/Textures...

Vulkan Device exposes various **Memory Heaps** - Example:

- **heap 0**: size: 12,288 Mb (Video Memory of my K6000)
- **heap 1**: size: 17,911 Mb (System Memory of my PC)

And various Memory Types from these Heaps. Example:

<table>
<thead>
<tr>
<th>Mem. Type</th>
<th>Heap</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 (sys.mem)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0 (Video)</td>
<td>DEVICE_LOCAL</td>
</tr>
<tr>
<td>2</td>
<td>1 (sys.mem)</td>
<td>HOST_VISIBLE</td>
</tr>
<tr>
<td>3</td>
<td>1 (sys.mem)</td>
<td>HOST_VISIBLE</td>
</tr>
</tbody>
</table>

**Tegra**: Adds one more: HOST_VISIBLE “NON-Coherent”
RESOURCE MANAGEMENT

Allocation and Sub allocation

Allocate memory type from heap

Allocation Type A
- Image
- Cube Image

Allocation Type B
- Buffer

Query Vulkan Object about size, alignment & type requirements
Assign memory subregion to a resource (allows aliasing)

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

HEAP supporting A,B

HEAP supporting B
RESOURCES MANAGEMENT

Not. So. Good.

Better...

#HappyGPU
Bind same buffer with Offsets > 0
1 buffer can have many types of data
Binding Memory to Objects - Simple use-case

Mem properties previously gathered at init time

```c
vkGetPhysicalDeviceMemoryProperties(physical_devices[j], &m_gpu.memoryProperties);

vkGetBufferMemoryRequirements(m_device, obj, &memReqs);

// Find an available memory type that satisfies the requested properties.
uint32_t memoryTypeIndex;
for (memoryTypeIndex = 0; memoryTypeIndex < m_gpu.memoryProperties.memoryTypeCount; ++memoryTypeIndex) {
    if ((memReqs.memoryTypeBits & (1 << memoryTypeIndex)) &&
        (m_gpu.memoryProperties.memoryTypes[memoryTypeIndex].propertyFlags & memProps) == memProps)
        break;
}

::VkMemoryAllocateInfo memInfo = { VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO, 0, memReqs.size, memoryTypeIndex };
result = vkAllocateMemory(m_device, &memInfo, NULL, &deviceMem);
result = vkBindBufferMemory(obj, deviceMem, 0);
```
Vulkan uses SPIR-V passed directly to the driver

Can be compiled from GLSL Via glslang or LunarG’s glslangValidator; Google ShaderC

theoretically other languages could be compiled to Spir-V...

Libraries available to compile GLSL to Spir-V from the application

NVIDIA allows to compile GLSL directly

NVIDIA VK_NV_glsd_shader: Vulkan reads GLSL directly
SHADERS

Multiple entry points can be defined in a single Spir-V shader-module

Prevents redundant code: shader module used by many Graphics-Pipelines

Specialization Constants: early setup of constants for shaders in given Graphics-Pipeline

Allows sharing snippets of code: easier to share common shader code

Warning: Current GLSL ➔ Spir-V compilers
Don’t support this feature, yet
But part of the API & Spir-V
Will happen soon
WSI manages the ownership of images via a swap chain
One image is presented while the other is rendered to
WSI is a Vulkan Extension
NVIDIA OPENGL ⇔ VULKAN INTEROP

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

GL_NV_draw_vulkan_image

Initialize...

```c
void Initialize() {
    glWaitVkSemaphoreNV = (PFNGLWAITVKSEMPAPHORENVPROC)NVPWindow::sysGetProcAddress("glWaitVkSemaphoreNV");
    glSignalVkSemaphoreNV = (PFNGLSIGNALKSEMPAPHORENVPROC)NVPWindow::sysGetProcAddress("glSignalVkSemaphoreNV");
    glSignalVkFenceNV = (PFNGLSIGNALKVFENCEENVPROC)NVPWindow::sysGetProcAddress("glSignalVkFenceNV");
    glDrawVkImageNV = (PFNGLDRAWVKIMAGENVPROC)NVPWindow::sysGetProcAddress("glDrawVkImageNV");
    if(glDrawVkImageNV == NULL) {
        LOGE("couldn't find entry points to blit Vulkan to OpenGL back-buffer (glDrawVkImageNV...)");
        nvk.DestroyDevice();
        m_bValid = false;
        return false;
    }
    VkSemaphoreCreateInfo semCreateInfo = { VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO ];
    mSemOpenGLReadDone = nvk.vkCreateSemaphore();
    mSemVKRenderingDone = nvk.vkCreateSemaphore();
```
HOW DOES IT LOOK?

GL_NV_draw_vulkan_image

```cpp
nvk.vkQueueSubmit( NVK::VkSubmitInfo(
    1, &m_semOpenGLReadDone, // pWaitSemaphores
    NULL,
    1, &m_cmdScene, // pCommandBuffers
    1, &m_semVKRenderingDone // pSignalSemaphores
),
    VK_NULL_HANDLE);
```

Submit commands to Queue with semaphores

Backbuffer Blit

```cpp
void RendererVk::blitToBackbuffer()
{
    float w = m_viewRect.extent.width;
    float h = m_viewRect.extent.height;

    // Wait for the queue of our VK rendering to signal m_semVKRenderingDone so we know the image is ready
    glWaitVkSemaphoreNV((GLuint64)m_semVKRenderingDone);

    // Blit the image
    glDrawVkImageNV((GLuint64)m_colorImage[m_currentImage].img, 0, 0, w, h, 0, 0, 1, 1, 0);

    // Signal m_semOpenGLReadDone to tell the VK rendering queue that it can render the next one
    glSignalVkSemaphoreNV((GLuint64)m_semOpenGLReadDone);
}
```
PRE-REQUISITES TO WORK WITH VULKAN

Lunar-G (http://lunarg.com/)

Vulkan Loader (+Source code)

Tools: Spir-V compiler for GLSL code and other libraries

Layers: intermediate code invoked by Vulkan API functions to help debug

Vulkan Includes

Drivers:

GeForce Experience

https://developer.nvidia.com/vulkan-driver

NVIDIA resources: https://developer.nvidia.com/Vulkan
RECAP’ ON NVIDIA-SPECIFIC FEATURES

Compatible GPUs for Vulkan: Kepler and Higher; Shield Tablet; Shield Android TV

VK_NV_glsl_shader : GLSL can be directly sent to Vulkan

VK_NV_dedicated_allocation : more efficient memory usage

GL_NV_draw_vulkan_image can replace WSI

16 Queues. All available for all kind of use; 1 Queue for Copy-Engine only

3 frames (max) in flight with WSI

All Host memories are “Coherent” (except one for Tegra)

Layout transitions don’t exist in our HW (VK_IMAGE_LAYOUT_GENERAL)

Linear-Tiling only for 2D non-mipmapped textures… please avoid (bad performance)

Shaders never need re-compilation due to states in Graphics-pipeline
NSIGHT FOR VULKAN
RECAP’ ON VULKAN PHILOSOPHY

Validate as much as possible up-front (DescriptorSets; Pipelines...)

The driver doesn’t waste time on figuring-out how to set things-up

Reuse existing patterns of Graphics-Pipelines: cached pipelines

Know your application: Taylor Vulkan design according to it

Know your memory usage: You are in charge of optimal sub-allocations

Explicit multi-threading for graphics: Application’s responsibility

Explicit Resource updates: Either through [non]Coherent buffers; or Queue-Based DMA transfers
FEW WORDS ON VKCPP PROJECT
C++11 to the rescue

- Open-Source Project of a C++11 overlay for Vulkan: became Khronos-official (!)
- Simplify Vulkan usage by
  - reducing risk of errors, i.e. type safety, automatic initialization of sType, ...
  - Reduce #lines of written code, i.e. constructors, initializer lists for arrays, ...
  - Add utility functions for common tasks (suballocators, resource tracking, ...)

VKCPP PROJECT
Two C++ based layers

Autogenerated 'low-level' layer using vulkan.xml
- Type safety
- Syntactic sugar
- Lightweight layer; Keeps you closer to the real Vulkan

Hand-coded 'high level' layer
- Reduce code complexity
- Exception safety, resource lifetime tracking, ...
- Closer dependency with VkCpp internal implementations
NATIVE VULKAN VS. VKCPP CODE

Native Vulkan: ~750 lines

vkCPP: ~200 lines
REFERENCES

Vulkan info from NVIDIA:

https://developer.nvidia.com/Vulkan

Samples + Source code in OpenGL and Vulkan:

https://github.com/nvpro-samples

Other:

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

HTTPS://DEVELOPER.NVIDIA.COM/DESIGNWORKS

TLORACH@NVIDIA.COM