AGENDA

Device Generated Commands
API Interop
VR in Vulkan
NSIGHT Support
VK_NVX_device_generated_commands
DEVICE GENERATED COMMANDS

**GPU creates its own work** (drawcalls and compute)

Define the work-load **in-pipeline, in-frame**

**Reduce latency** as no CPU roundtrip is required (VR!)

Use any GPU accessible resources to drive decision making (zbuffer etc.)

Select level of detail, cull by occlusion, classify work into different state usage, ...
DEVICE GENERATED COMMANDS

OpenGL Examples

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

ARB_draw_indirect to classify how particles are drawn (point, mesh, tessellation)

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

ARB_multi_draw_indirect / NV_command_list to do shader-based occlusion culling

Reverse angle & bboxes of culled Model courtesy of PGO Automobiles
EVOLUTION

**Draw Indirect:** Typically change # primitives, # instances

**Multi Draw Indirect:** Multiple draw calls with different index/vertex offsets

**GL_NV_command_list & DX12 ExecuteIndirect:** Change shader input bindings for each draw

**VK_NVX_device_generated_commands**
Change shader (pipeline state) per draw call

```cpp
DrawElements {
    GLuint     indexCount;
    GLuint     instanceCount;
    GLuint     firstIndex;
    GLuint     baseVertex;
    GLuint     baseInstance;
}
```

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

```cpp
DescriptorSetToken {
    GLuint     objectTableIndex;
    GLuint8[]  offsets[];
}
```
CPU-driven state setup is for worst-case distribution of indirect work
May yield lots of needless state setup (imagine 100s of potentially-used Pipelines)
NEW VULKAN ABILITY

GPU classifies items with state assignment

Optionally preserve ordering or provide permutation

Draw Indirects with State

Compact stream without unnecessary state setup or data overfetching

Grouping by state is still recommended
PIPELINE CHANGES

Add command-related work on the GPU to be more efficient at the actual tasks

Make use of shader specialization (less dynamic branching, more aggressive compile-time optimizations...)

Shader level of detail

Partition & organize work by shader permutation or usage pattern
STATELESS DESIGN

- CPU Commands
- Device-Generated Commands
- CPU Commands

CPU-provided state is inherited

State Access

bind  bind  draw
bind  bind  draw

Stateful within single command sequence
Modified state is undefined for subsequent sequences or CPU commands
OVERVIEW

Sequence & CPU Arguments

**VkIndirectCommandsLayout**
- **BindVertexBuffer** (binding)
- **Draw**

GPU-Written Arguments

- **VkCmdBindVertexBuffer** (binding, Buffer C, 256)
- **VkCmdDraw(32)**
- **VkCmdBind..**
- **VkCmdDraw**

Resources

**VkObjectTable**
- [0] Buffer A
- [1] Buffer B
- [2] Buffer C

Reserved CommandBuffer Space

```
uint32[]
2,256  0,0
```
Define a stateless sequence of commands as `VkIndirectCommandsLayout`

Register Vulkan resources (VkBuffer, VkDescriptorSet, VkPipeline) in `VkObjectTable` at developer-managed index

Fill & modify `VkBuffers` with command arguments and object table indices for many sequences

Use `VkCmdReserveSpaceForCommands` to allocate command buffer space

Generate the commands from token buffer content via `VkCmdProcessCommands`

Execute via `VkCmdExecuteCommands`
SEPARATE GENERATION & EXECUTION

Record an array of command sequences into the reserved space

Generate & Execute as single action is also supported

Reuse commands, or reuse reserved space for another generation
OBJECT TABLE

ObjectTable behaves similar to DescriptorPool

Do not delete it, nor modify resource indices that may be in-flight
CommandBuffer reservation depends on ObjectTable's state

Use only those resources, that were registered at reservation time
## INDIRECT COMMANDS

<table>
<thead>
<tr>
<th>VK INDIRECT COMMANDS TOKEN</th>
<th>EQUIVALENT COMMAND &amp; GPU-WRITTEN ARGUMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>_PIPELINE_NVX</td>
<td>vkCmdBindPipeline(... pipeline)</td>
</tr>
<tr>
<td>_DESCRIPTOR_SET_NVX</td>
<td>vkCmdBindDescriptorSets(... descrSet, offsets)</td>
</tr>
<tr>
<td>_INDEX_BUFFER_NVX</td>
<td>vkCmdBindIndexBuffer(... buffer, offset)</td>
</tr>
<tr>
<td>_VERTEX_BUFFER_NVX</td>
<td>vkCmdBindVertextBuffer (... buffer, offset)</td>
</tr>
<tr>
<td>_PUSH_CONSTANT_NVX</td>
<td>vkCmdPushConstants(... data)</td>
</tr>
<tr>
<td>_DRAW_INDEXED_NVX</td>
<td>vkCmdDrawIndexed( <em>all</em> )</td>
</tr>
<tr>
<td>_DRAW_NVX</td>
<td>VkCmdDraw( <em>all</em> )</td>
</tr>
<tr>
<td>_DISPATCH_NVX</td>
<td>VkCmdDispatch( <em>all</em> )</td>
</tr>
</tbody>
</table>
MULTIPLE INPUT STREAMS

Command Sequences

| 0 Command A | 0 Command B | 0 Command C | 1 | 1 | 1 |

Traditional approaches used single interleaved stream (array of structures AoS)

| Buffer | 0 | 0 | 0 | 1 | 1 | 1 |

VK extension uses input streams (SoA), allows individual re-use and efficient updates on input

| Buffer | 0 | 1 |
| Buffer | 0 | 1 |
| Buffer | 0 | 1 |

Common Input Rate

| Buffer | 0 | 1 |
| Buffer | 0,1 |
| Buffer | 0,1,.. |

Individual Input Rate
## Flexible Sequencing

<table>
<thead>
<tr>
<th>Ordered Sequences</th>
<th>Unordered / Subset</th>
<th>Custom Subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7</td>
<td>3 2 0 1</td>
<td>2 5 1 4</td>
</tr>
</tbody>
</table>

- **Default monotonic order of command sequences**
- **Allow impl.-dependent ordering (incoherent)**
- **Provide sequence indices as additional GPU buffer**

**CPU Argument**
- Number of sequences by CPU: 8

**Buffer**
- Actual number provided by GPU Buffer: 4

**Buffer**
- 2 5 1 4
TEST BENCHMARK

200,000 Drawcalls (few triangles/lines)

45,000 Pipeline switches (lines vs triangles)

6 Tokens:
- Pipeline
- DescriptorSet (1 ubo + 1 offset)
- DescriptorSet (1 ubo + 1 offset)
- VertexBuffer + 1 offset
- IndexBuffer + 1 offset
- DrawIndexed

https://github.com/nvpro-samples/gl_vk_threaded_cadscene/blob/master/doc/vulkan_nvdevicegenerated.md
# TEST BENCHMARK

<table>
<thead>
<tr>
<th>Test Scenario</th>
<th>Generate</th>
<th>Execute</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>200 000 DRAWCALLS</strong>&lt;br&gt;<strong>45 000 PSO CHANGES</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Driver (CPU 1 thread)</td>
<td>8.74 ms (async, on CPU)</td>
<td>14.74 ms</td>
</tr>
<tr>
<td>Device Gen. Cmds</td>
<td>0.35 ms</td>
<td>8.12 ms</td>
</tr>
<tr>
<td><strong>100 000 DRAWCALLS</strong>&lt;br&gt;<strong>NO PSO</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Driver (CPU 1 thread)</td>
<td>3.8 ms (async, on CPU)</td>
<td>1.8 ms</td>
</tr>
<tr>
<td>Device Gen. Cmds</td>
<td>0.20 ms</td>
<td>1.8 ms</td>
</tr>
</tbody>
</table>

Test benchmark is very simplified scenario, your milage will vary
NVIDIA IMPLEMENTATION

Currently experimental extension, feedback welcome (design, performance etc.)

VkIndirectCommandsLayout generates internal compute shader

Compute shader stitches the command buffer from data stored in the VkObjectTable

Implements redundant state filter within local workgroup

Reserved command buffer space has to be allocated for worst-case scenario
Previous 200,000 drawcall example reserved ~35 and generated ~15 megs

Global memory used internally to stitch command buffer

```c
struct ObjectTable {
  uint pipelinesCount;
  uint descriptorsetsCount;
  uint vertexbuffersCount;
  uint indexbuffersCount;
  uint pushconstantCount;
  uint pipelinesetsCount;
  ResourcePipeline* pipelines;
  ResourceDescriptorSet* descriptorsets;
  ResourceVertexBuffer* vertexbuffers;
  ResourceIndexBuffer* indexbuffers;
  ResourcePushConstant* pushconstants;
  ResourcePipelineSet* pipelinesets;
  uint* rawPipelines;
  uint* rawDescriptorsets;
  uint* rawVertexbuffers;
  uint* rawIndexbuffers;
  uint* rawPushconstants;
  uint* rawPipelinesets;
  uvec2* pipelinediffs;
  uint* rawPipelinediffs;
};

struct GeneratingTask {
  uint maxSequences;
  uvec4 sequenceRawSizes;
  uint* outputBuffer;
  uint* inputBuffers[MAX_INPUTS];
  ...
};
```

Variable GPU command sizes per object

Reserved size for worst-case
CONCLUSION

GPU-generating will get slower with divergent resource usage
Still important to group by state, helps both CPU and GPU

CPU-generating is asynchronous to device, may not add to frame-time
GPU-generating is on device, best used to save work, not to offload work
CROSS API INTEROP
CROSS API INTEROP

Generic framework lead by Khronos

Share device memory & synchronization primitives across APIs and processes

Created in context of Vulkan, but not exclusive to it

Vulkan, OpenGL, DirectX (11, 12), others may follow
EXTERNAL MEMORY
VK_KHX_external_memory (& friends)

New extensions to share memory objects across APIs

VkMemoryAllocateInfo was extended

VkImportMemory*Platform*HandleInfoKHX to reference memory owned by other instances of the same device

VkExportMemory*Platform*HandleInfoKHX to make memory accessible to other instances

VkGetMemory*Platform*KHX to query platform handle
EXTERNAL MEMORY

VK_KHX_external_memory (& friends)

Resource owning instance/API
Vulkan/DX/

Resource shared instance/API
Vulkan/GL/DX/

Memory Allocation
Buffer
Image

Memory Allocation
Buffer
Image

Export
Native Handle
Import

Memory offsets for resources are provided by original instance
EXTERNAL SYNCHRONIZATION

VK_KHX_external_semaphore (& friends)

Same principle as with memory

Allows sharing device synchronization primitives

Control command flow and dependencies on the same device

API/Instance A
Vulkan/GL/DX/…

Command Stream

Semaphore

Native Handle

API/Instance B
Vulkan/GL/DX/…

Semaphore

Command Stream
CROSS API INTEROP

May allow adding Vulkan (or other APIs) to host applications not designed for it

OpenGL extension to import Vulkan memory is in progress (but not to export from it)

Synchronization across (or within) APIs should not be very frequent (Frankenstein API usage)
VULKAN VR
NVIDIA VRWORKS
Comprehensive SDK for VR Developers

<table>
<thead>
<tr>
<th>GRAPHICS</th>
<th>HEADSET</th>
<th>TOUCH &amp; PHYSICS</th>
<th>AUDIO</th>
<th>VIDEO</th>
</tr>
</thead>
<tbody>
<tr>
<td>LENSMATCHED SHADING</td>
<td>CONTEXT PRIORITY</td>
<td>PHYSX</td>
<td>VRWORKS AUDIO</td>
<td>VRWORKS 360 VIDEO</td>
</tr>
<tr>
<td>SINGLEPASS STEREO</td>
<td>DIRECT MODE</td>
<td>WARP &amp; BLENDE</td>
<td></td>
<td>360</td>
</tr>
<tr>
<td>MULTIKS SHADING</td>
<td>FRONT BUFFER RENDERING</td>
<td>SYNCHRONIZATION</td>
<td></td>
<td>FOR VIDEO</td>
</tr>
<tr>
<td>VR SPLIT</td>
<td></td>
<td>GPU AFFINITY</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Vulkan®**
# NVIDIA VRWORKS

Comprehensive SDK for VR Developers

<table>
<thead>
<tr>
<th>GRAPHICS</th>
<th>HEADSET</th>
<th>TOUCH &amp; PHYSICS</th>
<th>AUDIO</th>
<th>VIDEO</th>
</tr>
</thead>
<tbody>
<tr>
<td>LENS MATCHED</td>
<td>CONTEXT</td>
<td>PHYSX</td>
<td>VRWORKS AUDIO</td>
<td>360 VIDEO</td>
</tr>
<tr>
<td>SHADING</td>
<td>PRIORITY</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SINGLE PASS</td>
<td>DIRECT MODE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STEREO</td>
<td>FRONT BUFFER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULTires</td>
<td>RENDERING</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHADING</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR SLI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
GRAPHICS PIPELINE

VR Workloads

124M Pix/s
N vertices
60 Hz

457M Pix/s
2N vertices
90 Hz

Preprocessing

Geometric Pipeline

Rasterization

Fragment Shader

Postprocessing
# NVIDIA VRWORKS

Comprehensive SDK for VR Developers

<table>
<thead>
<tr>
<th>GRAPHICS</th>
<th>HEADSET</th>
<th>TOUCH &amp; PHYSICS</th>
<th>AUDIO</th>
<th>VIDEO</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LENS MATCHED SHADING</strong></td>
<td><strong>CONTEXT PRIORITY</strong></td>
<td><strong>PHYSX</strong></td>
<td><strong>VRWORKS AUDIO</strong></td>
<td><strong>VRWORKS 360 VIDEO</strong></td>
</tr>
<tr>
<td><strong>SINGLE PASS STEREO</strong></td>
<td><strong>DIRECT MODE</strong></td>
<td><strong>PROFESSIONAL</strong></td>
<td></td>
<td><strong>GPU DIRECT FOR VIDEO</strong></td>
</tr>
<tr>
<td><strong>MULTIRES SHADING</strong></td>
<td><strong>FRONT BUFFER RENDERING</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>VR SLI</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SINGLE PASS STEREO

Traditional Rendering

Render eyes separately
Doubles CPU and GPU load
SINGLE PASS STEREO
Using SPS to improve rendering performance

Single Pass Stereo uses Simultaneous Multi-Projection architecture

Draw geometry only once

Vertex/Geometry stage runs once
Outputs two positions for left/right

Only rasterization is performed per-view

More Detail:
GTC2017 - S7578 - ACCELERATING YOUR VR APPLICATIONS WITH VRWORKS
SINGLE PASS STEREO

Vulkan

In Vulkan via `VK_NVX_multiview_per_view_attributes`

Requires `VK_KHX_multiview` and `VK_NV_viewport_array2` extensions

Check support using `vkGetPhysicalDeviceFeatures2KHR` with a
`VkPhysicalDeviceMultiviewPerViewAttributesPropertiesNVX` struct

Spec distinguishes between extension support in one or all components of position attribute

We only need support for the X component for VR
SINGLE PASS STEREO

Setup

Create layered texture image and view for rendering left and right simultaneously

Set up render pass with MultiView support

Broadcast rendering to both viewports

\[ \text{VkRenderPassMultiviewCreateInfoKHX::pViewMasks} \rightarrow 0b0011 \]

Hint to render both views concurrently, if possible

\[ \text{VkRenderPassMultiviewCreateInfoKHX::pCorrelationMasks} \rightarrow 0b0011 \]

Fill UBO with offsets for left and right eye
Calculate projection space position

\[
\text{proj\_pos} = (\text{proj} \ast \text{view} \ast \text{model} \ast \text{inPosition}).\text{xyz};
\]

Standard MultiView - specify once, may execute shader twice

\[
\text{gl\_Position} = \text{proj\_pos} + \text{UBO.offsets}[\text{gl\_ViewIndex}];
\]

With per-view attributes - also specify positions explicitly, execute shader only once

\[
\text{gl\_PositionPerViewNV[0]} = \text{proj\_pos} + \text{UBO.offsets}[0];
\]
\[
\text{gl\_PositionPerViewNV[1]} = \text{proj\_pos} + \text{UBO.offsets}[1];
\]
Single Pass Stereo brings benefits in geometry bound scenarios

Heavy fragment shaders will reduce scaling

**Graphics Pipeline**

**Single Pass Stereo Performance Results**

NVIDIA Quadro P6000, Scene with 17.6M faces, frame times in ms
# NVIDIA VRWORKS
Comprehensive SDK for VR Developers

<table>
<thead>
<tr>
<th>GRAPHICS</th>
<th>HEADSET</th>
<th>TOUCH &amp; PHYSICS</th>
<th>AUDIO</th>
<th>VIDEO</th>
</tr>
</thead>
<tbody>
<tr>
<td>LENS MATCHED SHADING</td>
<td>CONTEXT PRIORITY</td>
<td>PHYSX</td>
<td>VRWORKS AUDIO</td>
<td>VRWORKS 360 VIDEO</td>
</tr>
<tr>
<td>SINGLE PASS STEREO</td>
<td>DIRECT MODE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULTITRIP SHADING</td>
<td>FRONT BUFFER RENDERING</td>
<td></td>
<td>WARP &amp; BLENDE</td>
<td>GPU DIRECT FOR VIDEO</td>
</tr>
<tr>
<td>VR SLI</td>
<td></td>
<td></td>
<td>SYNCHRONIZATION</td>
<td></td>
</tr>
</tbody>
</table>
LENS MATCHED SHADING
Countering Lens Distortion

Displayed Image → Optics → User’s View
LENS MATCHED SHADING

Oversampling near the borders

Rendered Image

Displayed Image
LENS MATCHED SHADING

\[ w' = w + Ax + By \]
LENS MATCHED SHADING

Four Viewports

Original Image

LMS Image
LENSES MATCHED SHADING

Vulkan

In Vulkan via `VK_NV_clip_space_w_scaling` extension

Set up four viewports, rendering full resolution

Set scissors to each quadrant

`VkPipelineViewportWScalingStateCreateInfoNV`

W scaling parameters:

- Use the viewport struct / set on creation
- Dynamic state & `vkCmdSetViewportWScalingNV`
LENS MATCHED SHADING

Shaders

$\text{gl\_ViewportMask[0]}$ controls broadcasting of vertices and primitives

Inefficient - set mask in vertex shader

$$\text{gl\_ViewportMask[0]} = 15;$$

More efficient - filter in pass through geometry shader

Determine quadrant(s) for each primitive

Set bit(s) in $\text{gl\_ViewportMask[0]}$
LENS MATCHED SHADING

Scaling and Unscaling

HMD runtime can’t consume warped images yet, need to unscale before submit

\[
\text{scale} = \frac{1}{1 - w_x P'_x - w_y P'_y}
\]

\[
P' = \text{scale} \times P
\]

\[
\text{unscale} = \frac{1}{1 + w_x P_x + w_y P_y}
\]

\[
P = \text{unscale} \times P'
\]
LENS MATCHED SHADING
Scaling and Unscaling
LENS MATCHED SHADING
Wx = 0.4  Wy = 0.4  24.2ms -> 11.3ms
LENS MATCHED SHADING

$W_x = 1.0 \quad W_y = 1.0 \quad 24.2\text{ms} \rightarrow 5.9\text{ms}$
LENS MATCHED SHADING

Wx = 2.0  Wy = 2.0  24.2ms -> 3.3ms
LMS can improve performance of Raster / Fragment stage

Trade-off between quality and performance
# NVIDIA VRWORKS
Comprehensive SDK for VR Developers

<table>
<thead>
<tr>
<th>GRAPHICS</th>
<th>HEADSET</th>
<th>TOUCH &amp; PHYSICS</th>
<th>AUDIO</th>
</tr>
</thead>
<tbody>
<tr>
<td>LENSTMATCHED SHADING</td>
<td>CONTEXT PRIORITY</td>
<td>PHYSX</td>
<td>VRWORKS AUDIO</td>
</tr>
<tr>
<td>SINGLE PASS STEREO</td>
<td>DIRECT MODE</td>
<td>WARP &amp; BLEND</td>
<td>VRWORKS 360 VIDEO</td>
</tr>
<tr>
<td>MULTITEMS SHADING</td>
<td>FRONT BUFFER RENDERING</td>
<td>SYNCHRONIZATION</td>
<td>GPU DIRECT FOR VIDEO</td>
</tr>
<tr>
<td>VR SLI</td>
<td></td>
<td>GPU AFFINITY</td>
<td></td>
</tr>
</tbody>
</table>

54
VR SLI
Overview

Common HMD VR use case, realized through **VK_KHX_device_group** extension

1. Broadcast scene data, upload separate view data
2. Render left view @ GPU 0, right view @ GPU 1
3. Transfer right view @ GPU 1 to GPU 0 for HMD submit
VR SLI
Enumerate devices, create device group

Create VkInstance using \texttt{VK\_KHX\_device\_group\_creation}

Use \texttt{vkEnumeratePhysicalDeviceGroupsKHX} to enumerate device groups

Check that devices in a candidate group support \texttt{VK\_KHX\_device\_group}

Make sure the device group supports peer access via \texttt{vkGetDeviceGroupPeerMemoryFeaturesKHX}

Create logical VkDevice using \texttt{VkDeviceGroupDeviceCreateInfoKHX struct}
VR SLI
Prepare multi-GPU textures

Use `vkBindImageMemory2KHR` to bind memory to images across GPU boundaries.

No direct texture copies in VK,
Use bindings to access memory

```c
deviceIndices0[] = { 0, 1 };
deviceIndices1[] = { 1, 1 };
```

Make sure the formats match!
VR SLI

Data Upload

Upload data e.g. using `vkCmdUpdateBuffer` recorded in command buffer

Submit with a `VkDeviceGroupSubmitInfoKHX` struct, allowing device masks

Scene and other view independent data can be broadcast

View matrix and other view dependent uploads are limited to one GPU
Submit one command buffer for rendering on both GPUs

Use Image 0 as render target

Broadcasting is the default

Restrict rendering using

- Command Buffer Info
- Render Pass Info
- `vkCmdSetDeviceMaskKHX`
- Submit Infos
VR SLI
Texture Transfer

Texture transfer via `vkCmdCopyImage` or `vkCmdBlitImage` restricted to GPU 0

Transfer Image 0 and Image 1

Targets
- Swap Chain Image
- HMD textures
- Post-Process texture
VR SLI covers a wide variety of workloads

Perfect load balancing between left/right eye and two GPUs

Copy overhead and view independent workloads limit scaling
TRY IT OUT!

VRWorks SDK: https://developer.nvidia.com/vrworks

SPS: vk_stereo_view_rendering

LMS: vk_clip_space_w_scaling

VR SLI: vk_device_group

Extensions

www.khronos.org/registry/vulkan/specs/1.0-extensions/html/vkspec.html

KHX and NVX are experimental, feedback welcome!
VULKAN NSIGHT SUPPORT
NSIGHT + VULKAN
What is Nsight Visual Studio Edition

Understand CPU/GPU interaction

Explore and debug your frame as it is rendered

Profile your frame to understand hotspots and bottlenecks

Save your frame for targeted analysis and experimentation

Debug & profile VR applications

Leverage the Microsoft Visual Studio platform

New in 5.3: Vulkan 1.0.42 support, extensions, serialization, shader reflection, and descriptor view
NSIGHT & VULKAN

Scrubber

- Multi-queue / multi-thread
- State buckets & VK_EXT_debug_markers
- Synchronization
NSIGHT + VULKAN
API Inspector - All of the render state

- Pipeline
- Render Pass
- Framebuffer
- Input Assembly
- Shaders
  - SPIRV Decorations
  - Uniform Values
- Viewport
- Raster
- Pixel Ops.
- Misc.
NSIGHT + VULKAN
Device Memory

- Memory Objects
- Contained resources
- Raw memory
- Mini-map view
NSIGHT + VULKAN
Descriptor Sets

Pool information

All descriptor objects with usage counts

Associated resources

Selected resource information
NSIGHT + VULKAN
C/C++ Serialization - Challenges Solved

Portability

```c
typedef struct VkMemoryAllocateInfo {
    vkStructureType sType;
    const void* pNext;
    VkDeviceSize allocationSize;
    uint32_t memoryTypeIndex;
} VkMemoryAllocateInfo;
```

Trace api

- Convert trace into lightweight portable C/C++ project
- Maybe useful to experiment with the project rather than full application
- Supports original threads, queues etc.

Frame looping

*Where are my particles!?*
NSIGHT + VULKAN

Roadmap

Profiler & Performance Analysis
Android & Linux Support
Shader Editing
Sparse Texture Support
Improved Resource Barrier Visualization
Future Extensions & Core Releases
THANK YOU

Christoph Kubisch (ckubisch@nvidia.com, @pixeljetstream)
Ingo Esser (iesen@nvidia.com)
BACKUP
VkObjectTableCreateInfoNVX createInfo = {VK_STRUCTURE_TYPE_OBJECT_...};
createInfo.maxPipelineLayouts = 1;
createInfo.pObjectEntryTypes = {VK_OBJECT_ENTRY_PIPELINE_NVX,...};
createInfo.pObjectEntryCounts = {4,...};

vkCreateObjectTableNVX(m_device, &createInfo, NULL, &m_table.objectTable);

VkObjectTablePipelineEntryNVX entry = {VK_OBJECT_ENTRY_PIPELINE_NVX};
entry.pipeline = pipelines.usingShaderA;

vkRegisterObjectNVX(m_table.objectTable, (VkObjectTableEntryNVX*)&entry, developerChosenIndex);
INDIRECT COMMANDS

```c
VkIndirectCommandsLayoutTokenNVX input;
input.type = VK INDIRECT_COMMANDS_TOKEN_PIPELINE_NVX;
input.bindingUnit = 0;
input.dynamicCount = 0;
input.divisor = 1;
inputInfos.push_back(input);

input.type = VK_OBJECT_ENTRY_DESCRIPTOR_SET_NVX;
input.bindingUnit = 0;
input.dynamicCount = 1;
input.divisor = 1;
inputInfos.push_back(input);
...

vkCreateIndirectCommandsLayoutNVX(m_device, genCreateInfo, NULL, &m_genLayout);
```
vkCmdReserveSpaceForCommandsNVX(cmdSecondary, {resourceTable, indirectLayout, maxCount});

VkIndirectCommandsTokenNVX input;
inout.buffer = inputBuffer;
inout.type = VK_INDIRECT_COMMANDS_TOKEN_PIPELINE_NVX;
inout.offset = pipeOffset;
inout.push_back(input);

inout.type = VK_INDIRECT_COMMANDS_TOKEN_DESCRIPTOR_SET_NVX;
inout.offset = matrixOffset;
inout.push_back(input);

... 
vkCmdProcessCommandsNVX(cmdPrimary, {resourceTable, indirectLayout, 
inout.size(), inout.data(), count, cmdTarget, NULL, 0});