VULKAN OVERVIEW

Piers Daniell, January 19, 2016

AGENDA

What is Vulkan?

Hello Triangle

Release plans

WHAT IS VULKAN?

VULKAN REQUIREMENTS

Leading Edge Graphics Functionality
Equivalent to OpenGL in V1.0

General Purpose Compute
Graphics AND compute queues in a single API

Precompiled Shaders
Enables language flexibility - including C++ Programming (future)

FUNCTIONALITY

Multi-core Efficient
Multi-threading friendly

Low Driver Overhead
Thinner, simpler driver reduces CPU bottleneck

PERFORMANCE

Same API for mobile, desktop, console and embedded
Defined 'Feature Sets' per platform
No need for 'Vulkan ES'

Explicit API
Direct control over GPU.
Simpler driver gives less surprises and vendor differences

Streamlined API
Easier to implement and test for cross-vendor consistency

PORTABILITY

NEXT GENERATION GPU APIS

DirectX 12

Only Windows 10

Only Apple

Vulkan™

Cross Platform

Any OpenGL ES 3.1/4.X GPU

Windows 7 · Windows 8 · Windows 10

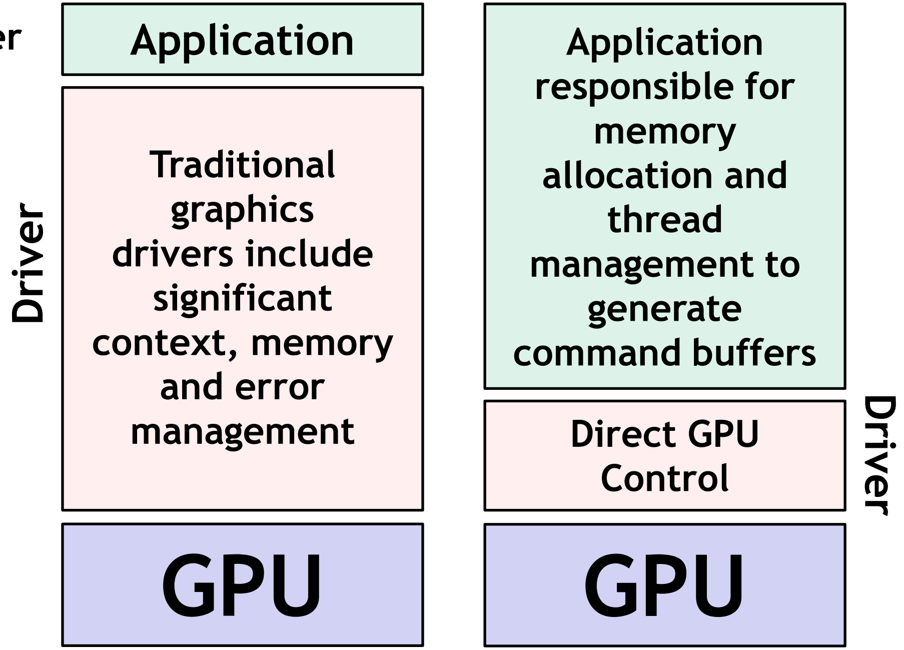
SteamOS · ubuntu · redhat · TIZEN

VULKAN EXPLICIT GPU CONTROL

Complex drivers lead to driver overhead and cross vendor unpredictability

Error management is always active

Driver compiles full shading language source



Simpler drivers for low-overhead efficiency and cross vendor consistency

Layered architecture so validation and debug layers can be loaded only when needed

Run-time only has to ingest SPIR-V intermediate language

THE POWER OF A THREE LAYER ECOSYSTEM

Applications can use Vulkan directly for maximum flexibility and control

Application uses utility libraries to speed development

Utility libraries and layers

Application

Games Engines fully optimized over Vulkan

Vulkan™

Developers can choose at which level to use the Vulkan Ecosystem

The industry's leading games and engine vendors are participating in the Vulkan working group

OXIDE

LUCASFILM Ltd

BLIZZARD ENTERTAINMENT

EA

unity

EPIC GAMES

VALVE

Rich Area for Innovation

- Many utilities and layers will be in open source
- Layers to ease transition from OpenGL
- Domain specific flexibility

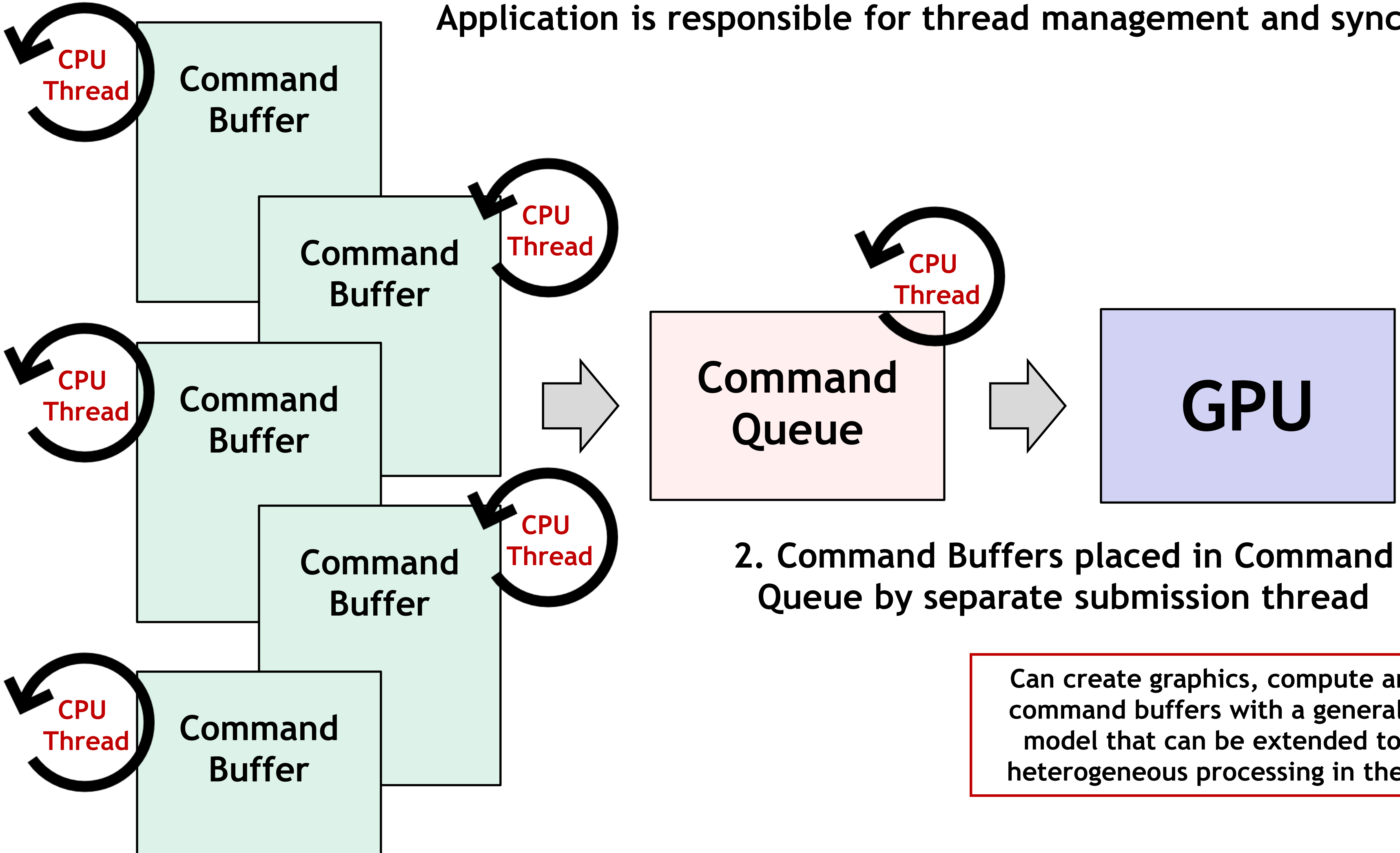
The same ecosystem dynamic as WebGL

A widely pervasive, powerful, flexible foundation layer enables diverse middleware tools and libraries

NVIDIA®

VULKAN MULTI-THREADING EFFICIENCY

1. Multiple threads can construct Command Buffers in parallel
Application is responsible for thread management and synch



2. Command Buffers placed in Command Queue by separate submission thread

Can create graphics, compute and DMA command buffers with a general queue model that can be extended to more heterogeneous processing in the future

SPIR-V Transforms the Language Ecosystem

- **First multi-API, intermediate language for parallel compute and graphics**
  - Native representation for Vulkan shader and OpenCL kernel source languages
  - https://www.khronos.org/registry/spir-v/papers/WhitePaper.pdf
- **Cross vendor intermediate representation**
  - Language front-ends can easily access multiple hardware run-times
  - Acceleration hardware can leverage multiple language front-ends
  - Encourages tools for program analysis and optimization in SPIR form

Multiple Developer Advantages

Same front-end compiler for multiple platforms
Reduces runtime kernel compilation time
Don't have to ship shader/kernel source code
Drivers are simpler and more reliable

Standard
Portable
Intermediate
Representation

VULKAN WORKING GROUP

- **Participants come from all segments of the graphics industry**
  - Including an unprecedented level of participation from game engine ISVs

Working Group Participants

HELLO TRIANGLE

HELLO TRIANGLE

Quick tour of the API

Launch driver and create display

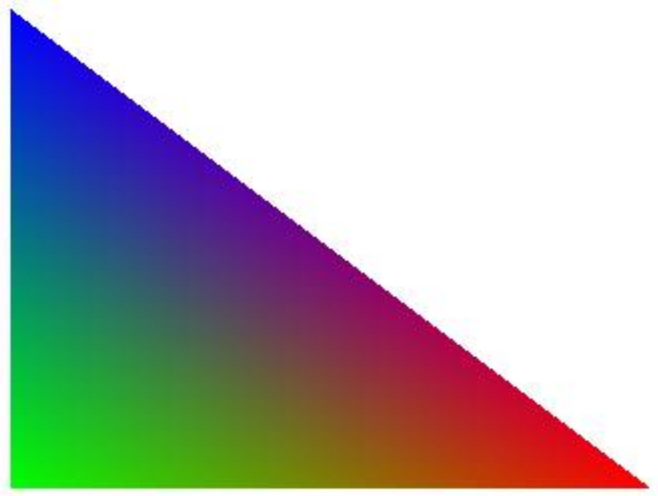
Set up resources

Set up the 3D pipe

- Shaders
- State

Record commands

Submit commands



VULKAN LOADER

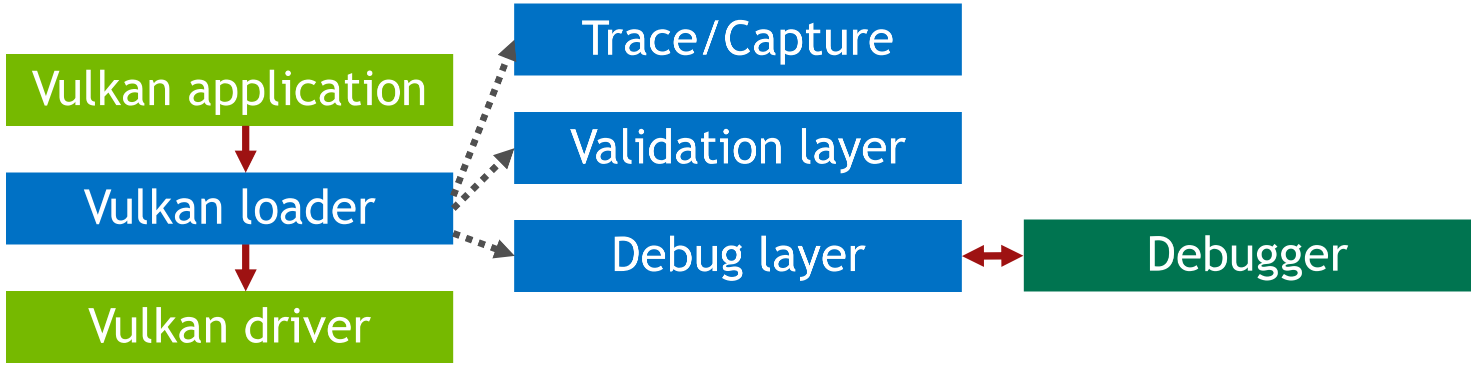
Part of the Vulkan ecosystem

Khronos provided open-source loader

Finds driver and dispatches API calls

Supports injectable layers

- Validation, debug, tracing, capture, etc.



VULKAN WINDOW SYSTEM INTEGRATION

WSI for short

Khronos defined Vulkan extensions

Creates presentation surfaces for window or display

Acquires presentable images

Application renders to presentable image and enqueues the presentation

Supported across wide variety of windowing systems

- Wayland, X, Windows, etc.

HELLO TRIANGLE

Quick tour of the API

Launch driver and create display

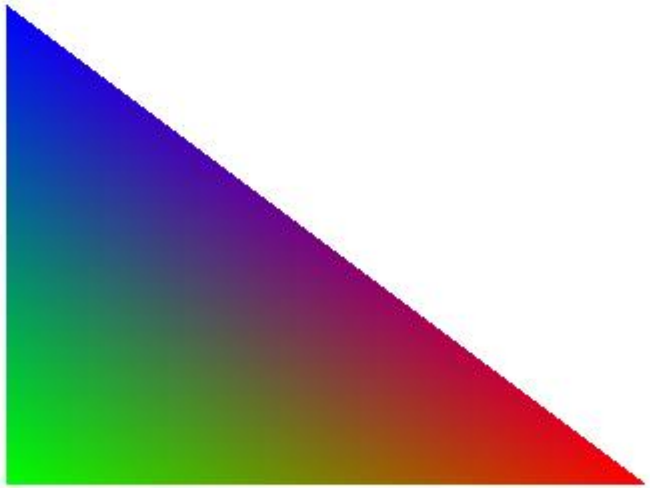
Set up resources

Set up the 3D pipe

- Shaders
- State

Record commands

Submit commands



Goals: explicit API, predictable performance

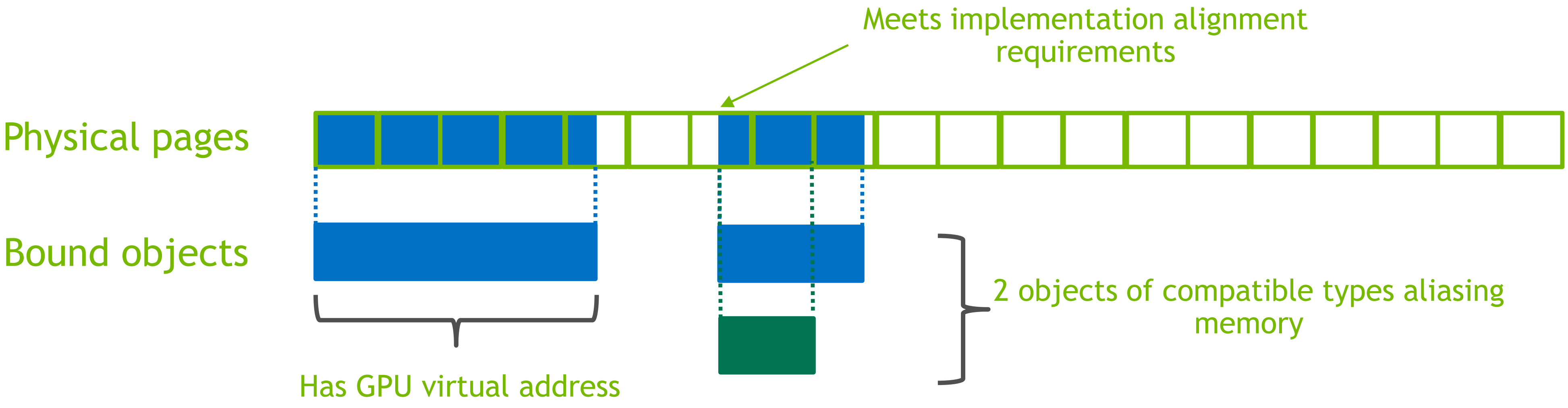
LOW-LEVEL MEMORY CONTROL

Console-like access to memory

Vulkan exposes several physical memory pools – device memory, host visible, etc.

Application binds buffer and image virtual memory to physical memory

Application is responsible for sub-allocation



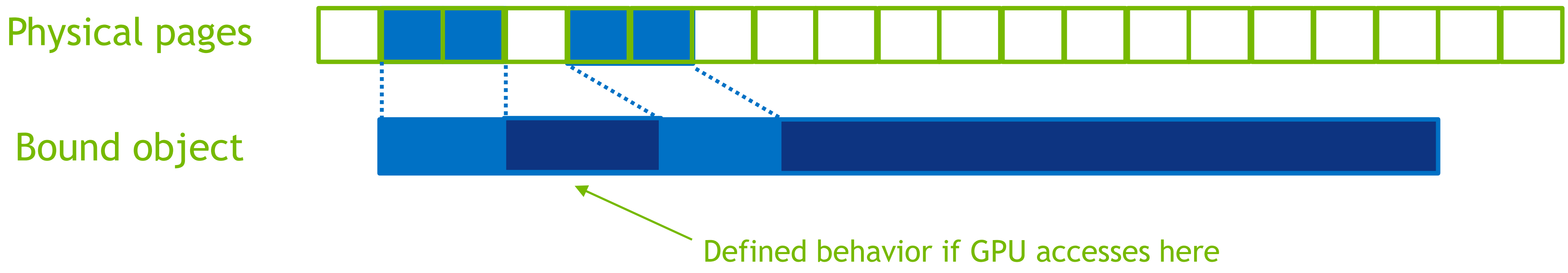
SPARSE MEMORY

More control over memory usage

Not all virtual memory has to be backed

Several feature levels of sparse memory supported

ARB_sparse_texture, EXT_sparse_texture2, etc.



RESOURCE MANAGEMENT

Populating buffers and images

Vulkan allows some resources to live in CPU-visible memory

Some resources can only live in high-bandwidth device-only memory

- Like specially formatted images for optimal access

Data must be copied between buffers

Copy can take place in 3D queue or transfer queue

Copies can be done asynchronously with other operations

- Streaming resources without hitching

POPULATING VIDMEM

Using staging buffers

Allocate CPU-visible staging buffers

- These can be reused

Get a pointer with vkMapMemory

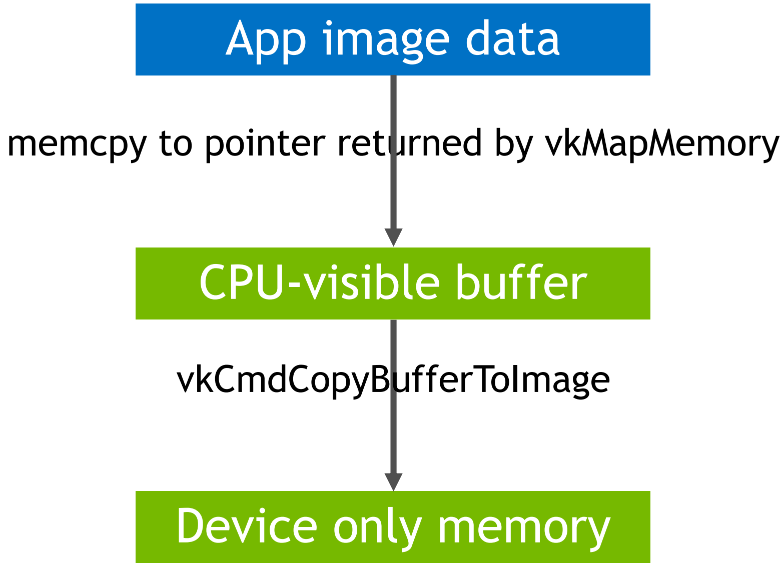
- Memory can remain mapped while in use

Copy from staging buffer to device memory

- Copy command is queued and runs async

Use vkFence for application to know when xfer is done

Use vkSemaphore for dependencies between command buffers



DESCRIPTOR SETS

Binding resources to shaders

Shader resources declared with binding slot number

```
layout(set = 1, binding = 3) uniform image2D myImage;
layout(set = 1, binding = 4) uniform sampler mySampler;
```

Descriptor sets allocated from a descriptor pool

Descriptor sets updated at any time when not in use

- Binds buffer, image and sampler resources to slots

Descriptor set bound to command buffer for use

- Activates the descriptor set for use by the next draw

MULTIPLE DESCRIPTOR SETS

Partitioning resources by frequency of update

Application can modify just the set of resources that are changing

Keep amount of resource binding changes as small as possible

Shader code

```
layout(set=0,binding=0) uniform { ... } sceneData;
layout(set=1,binding=0) uniform { ... } modelData;
layout(set=2,binding=0) uniform { ... } drawData;

void main() { }
```

Application code

```
foreach (scene) {
  vkCmdBindDescriptorSet(0, 3, {sceneResources,modelResources,drawResources});
  foreach (model) {
    vkCmdBindDescriptorSet(1, 2, {modelResources,drawResources});
    foreach (draw) {
      vkCmdBindDescriptorSet(2, 1, {drawResources});
      vkDraw();
    }
  }
}
```

HELLO TRIANGLE

Quick tour of the API

Launch driver and create display

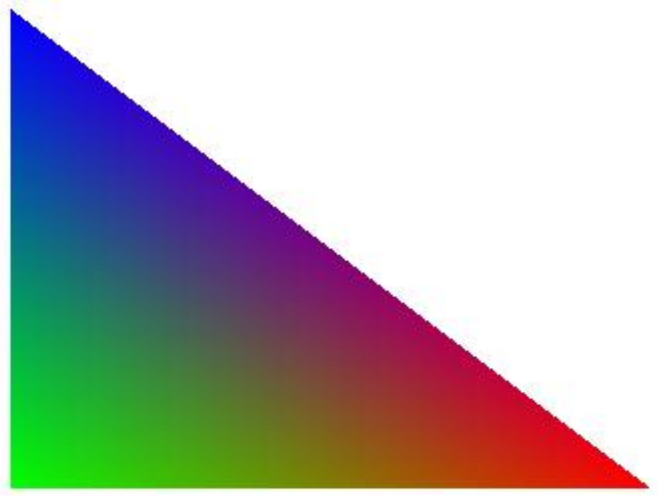
Set up resources

Set up the 3D pipe

- Shaders
- State

Record commands

Submit commands



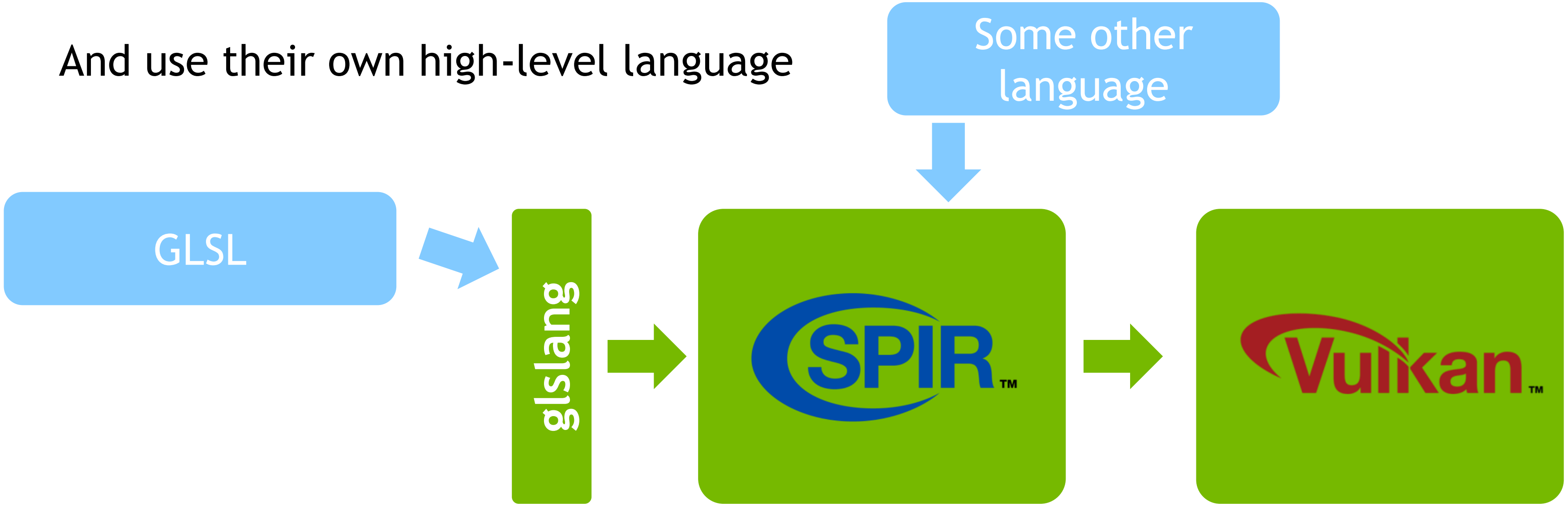
SPIR-V

For your content pipeline

Khronos supported open-source GLSL->SPIR-V compiler - glslang

ISVs can easily incorporate into their content pipeline

And use their own high-level language



VULKAN SHADER OBJECT

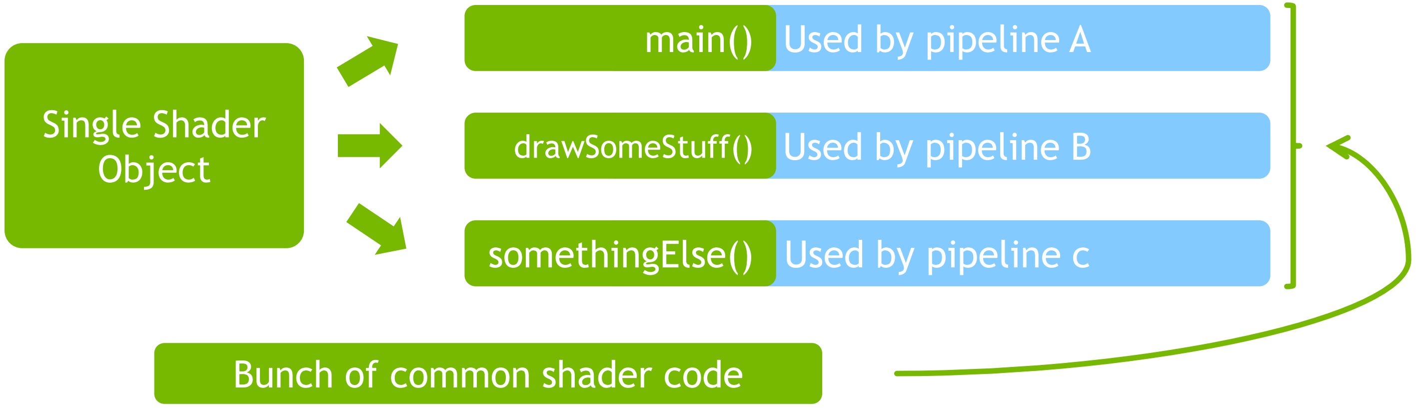
Compiling the SPIR-V

SPIR-V passed into the driver

Driver can compile everything except things that depend on pipeline state

Shader object can contain an uber-shader with multiple entry points

- Specific entry point used for pipeline instance



PIPELINE STATE OBJECT

Say goodbye to draw-time validation

Represents all static state for entire 3D pipeline

- Shaders, vertex input, rasterization, color blend, depth stencil, etc.

Created outside of the performance critical paths

Complete set of state for validation and final GPU shader instructions

- All state-based compilation done here – not at draw time

Can be cached for reuse

- Even across application instantiations

PIPELINE CACHE

Reusing previous work

Application can allocate and manage pipeline cache objects

Pipeline cache objects used with pipeline creation

- If the pipeline state already exists in the cache it is reused

Application can save cache to disk for reuse on next run

Using the Vulkan device UUID – can even stash in the cloud

PIPELINE LAYOUT

Using compatible pipelines

- Pipeline layout defines what kind of resource is in each binding slot
  - Images, Samplers, Buffers (UBO, SSBO)
- Different pipeline state objects can use the same layout
  - Which means shaders need to use the same layout
- Changing between compatible pipelines avoids having to rebind all descriptions
  - Or use lots of different descriptor sets

DYNAMIC STATE

State that can change easily

Dynamic state changes don’t affect the pipeline state

- Does not cause shader recompilation

Viewport, scissor, color blend constants, polygon offset, stencil masks and refs

Dynamic state changes are relatively lightweight

All other state has the potential to cause a shader recompile on some hardware

- So it belongs in the pipeline state object with the shaders

PUSH CONSTANTS

For high-frequency updates

Small shader-accessible high-speed uniform buffer

Up to 256 bytes in size

Can be updated at high-frequency – per draw for example

Use for per-draw indices or transform matrices, etc.

HELLO TRIANGLE

Quick tour of the API

Launch driver and create display

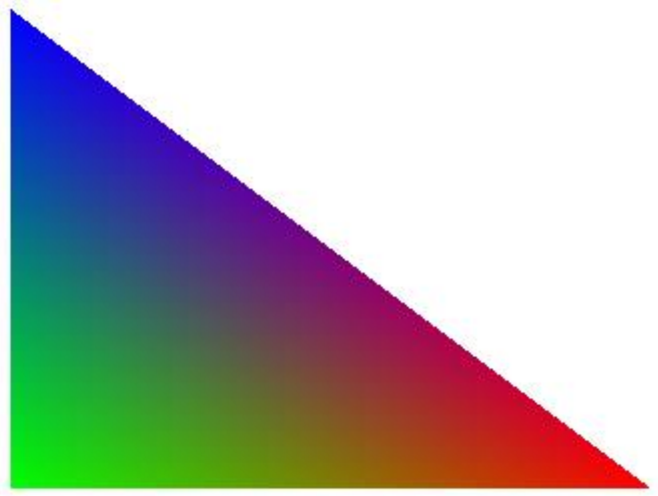
Set up resources

Set up the 3D pipe

- Shaders
- State

Record commands

Submit commands



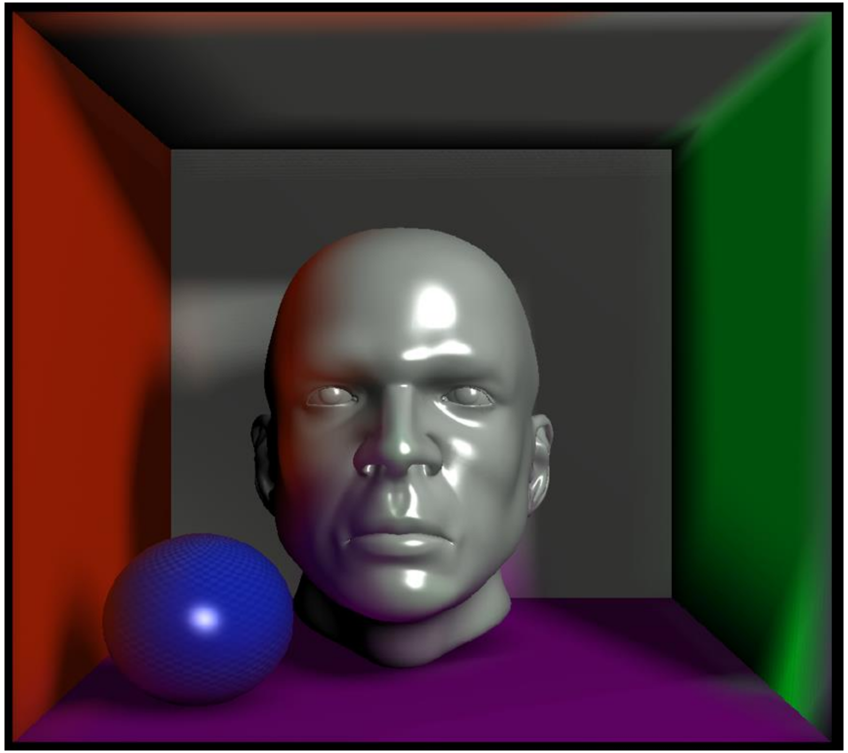
RENDERING

Multi-pass rendering

Consider a deferred renderer

- z-fill pass
- gBuffer pass
- Lighting pass

How would this work in GL on a tiled renderer



MULTI-PASS RENDERING

Tiled rendering

How would this work in GL on a tiled renderer

Pass 1

Bind depth attachment
Load each tile from FBO
Z-fill each tile
Store each tile to FBO
Repeat until done

Pass 2

Bind float attachment
Load each tile from FBO
Store geometry to tiles
Store each tile to FBO
Repeat until done

Pass 3

Bind gBuffer texture
Bind color attachment
Load each tile from FBO
Render scene to tiles
Store each tile to FBO
Repeat until done

Lots of bandwidth to and from the framebuffer!!

Of course it’s possible to do this in Vulkan as well. It’s just not a good idea.

MULTI-PASS RENDERING

Tiled Rendering

Vulkan uses a RenderPass object

For each tile

Load tiles for depth, gBuffer and color
Start subpass
Render z-fill
Start subpass
Store geometry in gBuffer
Specify gBuffer as input to final subpass
Start subpass
Render scene
Store depth,gBuffer and color back to FBO

All that slow and power hungry bandwidth is eliminated!

A render pass object can also handle a final multisample resolve

MULTI-PASS RENDERING

Tiled Rendering

CommandBuffer

SubPass A

Attachment Image(s)

SubPass B
(can depend on A)

Attachment Image(s)

Resolve…

Attachment Image(s)

Binds, Draws..

Also Secondary CmdBuffers

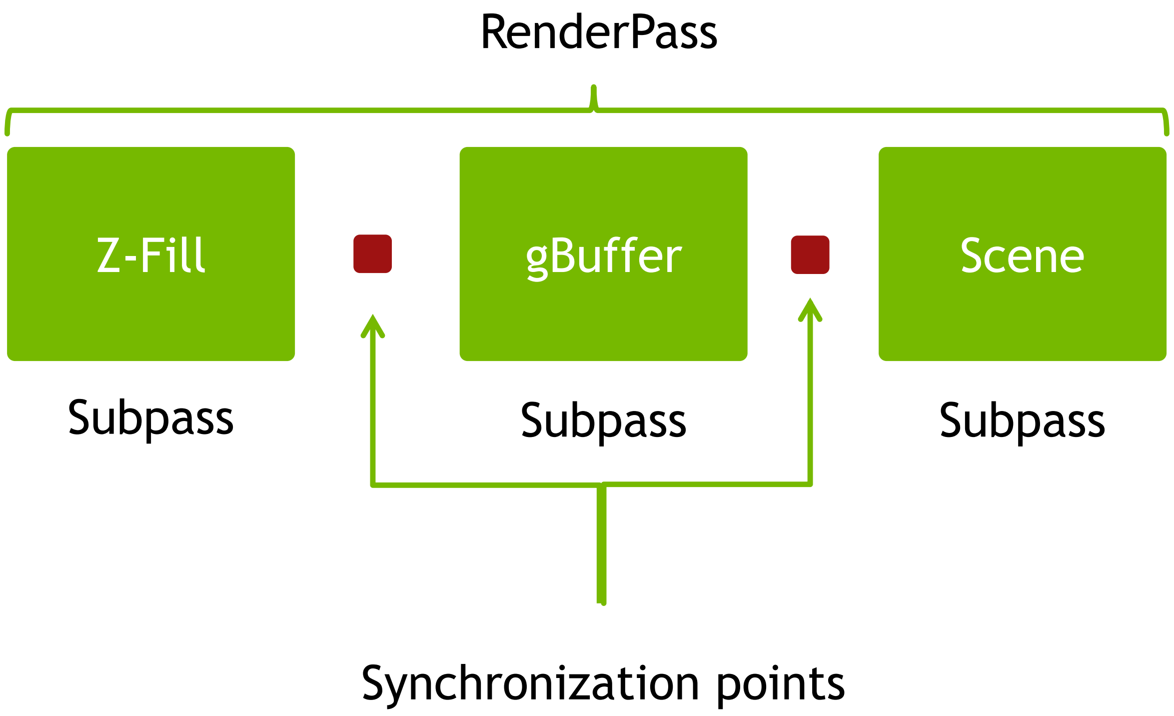
Binds Draws… | …

Make RenderTargets available to other CommandBuffer passes

Bound Pipelines must be associated with specific RenderPass config (A,B) and sub-pass at creation

MULTI-PASS RENDERING

Dependencies



Dependencies exist between these subpasses

But these are on a per tile basis

Define these dependencies with the renderpass

Any tile who's dependencies are satisfied can continue

COMMAND BUFFERS AND POOLS

A place for the GPU commands

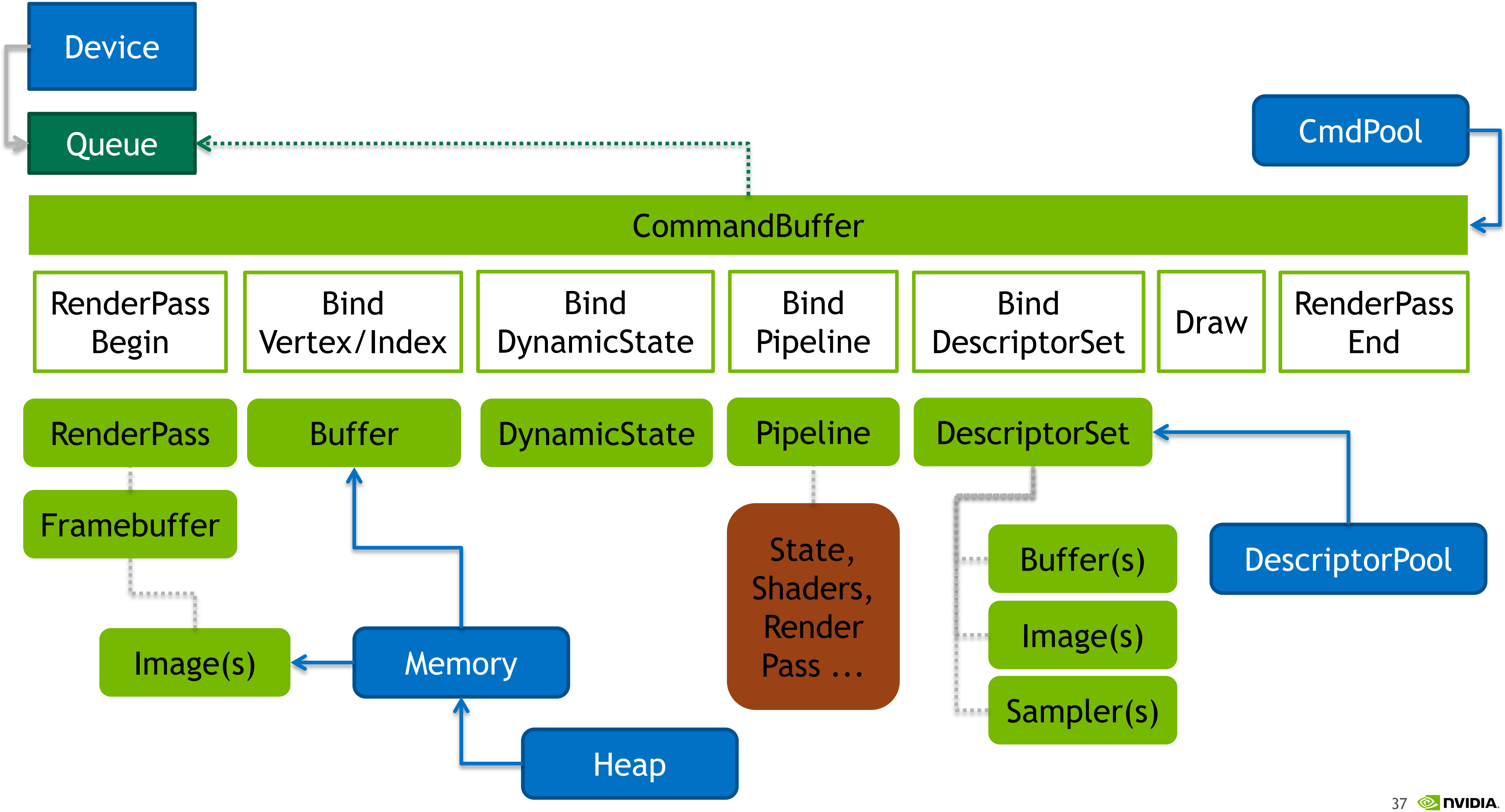
A command buffer is an opaque container of GPU commands

Command buffers are submitted to a queue for the GPU to schedule execution

Commands are adding when the command buffer is recorded

Memory for the command buffer is allocated from the command pool

Multiple command buffers can allocate from a command pool


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

COMMAND BUFFER PERFORMANCE

Command buffer recording needs to scale well

Recording command buffers is the most performance critical part

- But we have no idea how big command buffer will end up

Can record multiple command buffers simultaneously from multiple threads

Command pools ensure there is no lock contention

- True parallelism provides multi-core scalability

Command buffer can be reused, re-recorded or recycled after use

- Reuse previous allocations by the command pool

MULTI-THREADING

Maximizing parallel multi-CPU execution

- Vulkan is designed so all performance critical functions don’t take locks
  - Application is responsible for avoiding hazards
- Use different command buffer pools to allow multi-CPU command buffer recording
- Use different descriptor pools to allow multi-CPU descriptor set allocations
- Most resource creation functions take locks
  - But these are not on the performance path

COMPUTE

For all your general-purpose computational needs

- Uses a special compute pipeline
- Uses the same descriptor set mechanism as 3D
  - And has access to all the same resources
- Can be dispatched interleaved with render-passes
  - Or to own queue to execute in parallel

RESOURCE HAZARDS

Application managed

Resource use from different parts of the GPU may have read/write dependencies

- For example, will writes to framebuffer be seen later by image sampling

Application uses explicit barriers to resolve dependencies

- GPU may flush/invalidate caches so latest data is written/seen

Platform needs vary substantially

- Application expresses all resource dependencies for full cross-platform support

Application also manages resource lifetime

- Can’t destroy a resource until all uses of it have completed

AVOIDING HAZARDS

An example – sampling from modified image

Update an image with shader imageStore() calls

```
vkBindPipeline(cmd, pipelineUsesImageStore);
vkDraw(cmd);
```

Flush imageStore() cache and invalidate image sampling cache

```
vkPipelineBarrier(cmd, image, SHADER_WRITE, SHADER_READ);
```

Can now sample from the updated image

```
vkBindPipeline(cmd, pipelineSamplesFromImage);
vkDraw(cmd);
```

HELLO TRIANGLE

Quick tour of the API

Launch driver and create display

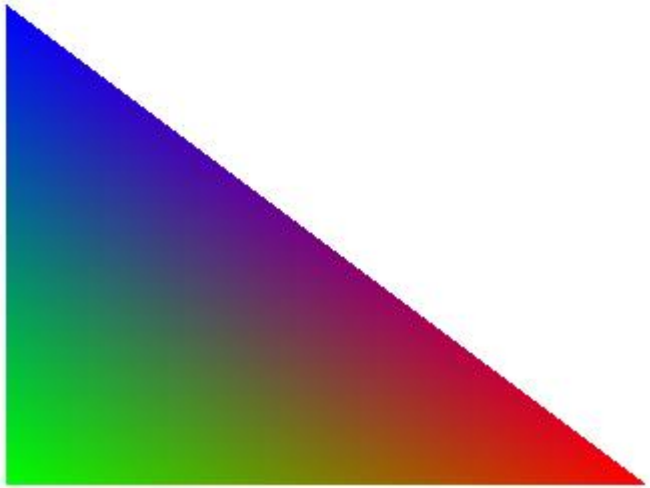
Set up resources

Set up the 3D pipe

- Shaders
- State

Record commands

Submit commands



QUEUE SUBMISSION

Scheduling the commands in the GPU

Implementation can expose multiple queues

- 3D, compute, transfer or universal

Queue submission should be cheap

Queue execution is asynchronous

App uses VkFence to know when work is done

App can use VkSemaphore to synchronize dependencies between queues

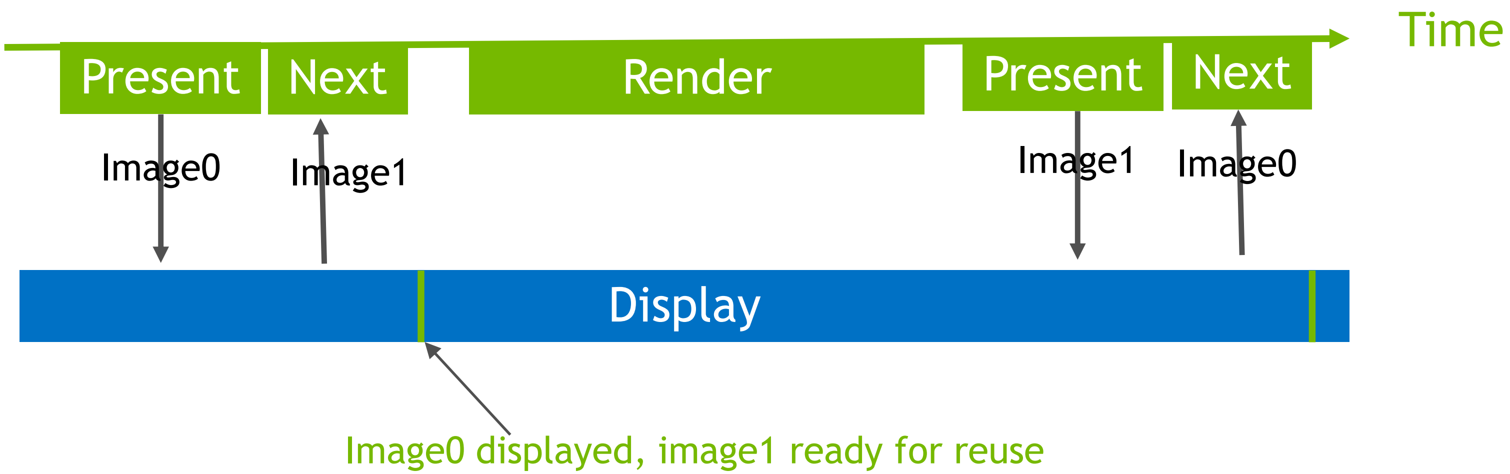
PRESENTATION

Using the WSI extension

The final presentable image is queued for presentation

Presentation happens asynchronously

After present is queued application picks up next available image to render to



GOOD PRACTICES

Use Vulkan well on NVIDIA GPUs

Perform your own sub-allocation from larger VkDeviceMemory allocations

- Reduces allocation overhead and “hitching”

Use **optimal** image tiling whenever possible

- Linear tiling is very limited on NVIDIA GPUs – 2D-only, no mipmaps, no arrays

Using dynamic UBOs and SSBOs to reduce descriptor set updates

On NVIDIA GPUs image layouts are irrelevant

- Just leave images in the VK_IMAGE_LAYOUT_GENERAL layout

PERFORMANCE

Putting it all together

MODE	GPU TIME	CPU TIME
gl uncached	4.1	7.8
vk uncached cmd 1 thread	1.7	1.5
vk uncached cmd 2 threads	1.7	0.8

From csfthreaded sample app with 44k drawcalls and high-frequency UBO and vertex buffer binding updates

NVIDIA VULKAN RELEASE PLANS

WHY IS IT IMPORTANT TO NVIDIA?

It’s open

API is designed to be extensible

- We can easily expose new GPU features

No single vendor or platform owner controls the API

Scales from low-power mobile to high-performance desktop

Can be used on any platform

It’s fast!

WHAT ABOUT OPENGL?

OpenGL is also very important to NVIDIA

- OpenGL and OpenGL ES will remain vital
  - Together have largest 3D API market share
  - Applications – games, design, medical, science, education, film-production, etc.
- OpenGL improvements just last year
  - OpenGL ES 3.2
  - 13+ New ARB extensions: ARB_post_depth_coverage, ARB_fragment_shader_interlock, ARB_texture_filter_minmax, ARB_sample_locations, ARB_shader_viewport_layer_array, ARB_sparse_texture2, ARB_sparse_texture_clamp, ARB_gpu_shader_int64, ARB_shader_clock, ARB_shader_ballot, ARB_ES3_2_compatibility, ARB_parallel_shader_compile, ARB_shader_atomic_counter_ops

OPENGL VS VULKAN

Solving 3D in different ways

- OpenGL higher-level API, easier to teach and prototype with
    - Many things handled automatically
- OpenGL can be used efficiently and obtain great single-threaded performance
    - Use multi-draw, bindless, persistently mapped buffers, PBO, etc.
- Vulkan’s ace is its ability to scale across multiple CPU threads
    - Can be used with almost no lock contention on the performance critical path

VULKAN ON NVIDIA GPUS

Fully featured

Vulkan is one API for all GPUs

Vulkan API supports optional features and extensions

Supports multiple vendors and hardware

- From ES 3.1 level hardware to GL 4.5 and beyond

NVIDIA implementation fully featured

- From Tegra K1 through GeForce GTX TITAN X

Write once run everywhere

VULKAN RELEASE DAY

Coming real soon now

Exact release date still Khronos confidential – but it's real soon

NVIDIA will release public developer drivers for Windows and Linux

Shield Tablet and Shield Android TV OTA updates will support Vulkan

Vulkan to be included in Windows and Linux r364 UDA consumer drivers by April

VULKAN GPU SUPPORT

ARCHITECTURE	GPUS
Kepler	GeForce 600 and 700 series Quadro Kxxx series Tegra K1
Maxwell	GeForce 900 series and TITAN X Quadro Mxxx series Tegra X1
Pascal	TBD

VULKAN FEATURE SUPPORT

FEATURE	KEPLER	MAXWELL
OpenGL ES 3.1 level features	Yes	Yes
OpenGL 4.5 level features	Yes	Yes
Sparse memory	Partial	Yes
ETC2, ASTC texture compression	Tegra	Tegra

RELEASE PLANS

What’s in our first release?

Fully conformant Vulkan implementation

- All basic optimizations implemented

Basic GL interop and GLSL support

- To help ease porting existing code and shaders

cfsthreaded sample app – source code and documentation

NVIDIA Dev-tech material

- Blog posts, samples, frameworks, wrappers, talks, conference sessions, support, etc.

GLSL IN VULKAN

To help with rapid development

Use GLSL directly when creating Vulkan shader modules

Implements KHR_vulkan_glsl extension

Developer convenience

Not intended to “replace” SPIR-V for shipping apps

OPENGL INTEROP

To ease porting existing apps

- OpenGL and Vulkan can be used together
- OpenGL extension to draw Vulkan image to GL framebuffer
  - glDrawVulkanImageNV
- Synchronize OpenGL and Vulkan
- cfsthreaded sample app demonstrates this
  - Sample app made available at release

CSFTHREADED

Sample app

Renders CAD models

Uses OpenGL and Vulkan together

Demonstrates several rendering techniques

- Simple GL
- NV_command_list
- Single-threaded Vulkan
- Multi-threaded Vulkan
- Various buffer updating techniques

ERROR CHECKING

Last safety net

- Vulkan spec requires minimal error checking in driver
  - Results are undefined with bad inputs or usage
  - May cause VK_ERROR_DEVICE_LOST
- NVIDIA Vulkan driver retains some “cheap” error checking
  - Mostly on vkCreate calls
  - Checks bad parameters
  - Reports invalid shaders

VALIDATION LAYER TODOS

Vital for Vulkan success

The Vulkan API is not easy to use for first-timers

There are no safety nets provided by base implementations

Validation layer is vital to Vulkan’s success

Current validation layer is far from complete

All our responsibility to improve the validation layer

- Report bugs (to LunarG and soon via GitHub issues)
- Fix and improve implementation directly

MISSING API FEATURES

Stuff in OpenGL that didn’t make version 1.0

Transform feedback

Conditional rendering

Multi-GPU

Specifying the instanced array divisor

Shader subroutines

MULTI-GPU

Working together to do more

Ability to synchronize GPUs with shared semaphores

Ability to share memory

Ability to do peer-peer transfers

Khronos goal to support both homogeneous and heterogeneous multi-GPU

VULKAN API IMPROVEMENTS

We can do better

State inheritance

More dynamic state

Remove PSO-framebuffer dependency for better PSO reuse

Remove secondary command buffer-framebuffer dependency

- So command buffers can be used with different framebuffers

DYNAMIC STATE

Things we can easily make dynamic

Primitive topology – point, lines, triangles, etc

Polygon mode – fill, line, point

Cull mode – none, front, back, front+back

Front face – ccw, cw

Depth stencil state – depthWrite, depthCompareOp, etc.

Blend state – color and alpha blend factor and ops

VULKAN INTEROP

Playing nice with other APIs

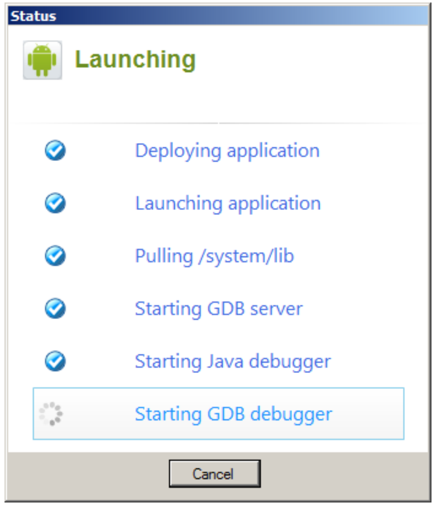
OpenGL interop – beyond the basic

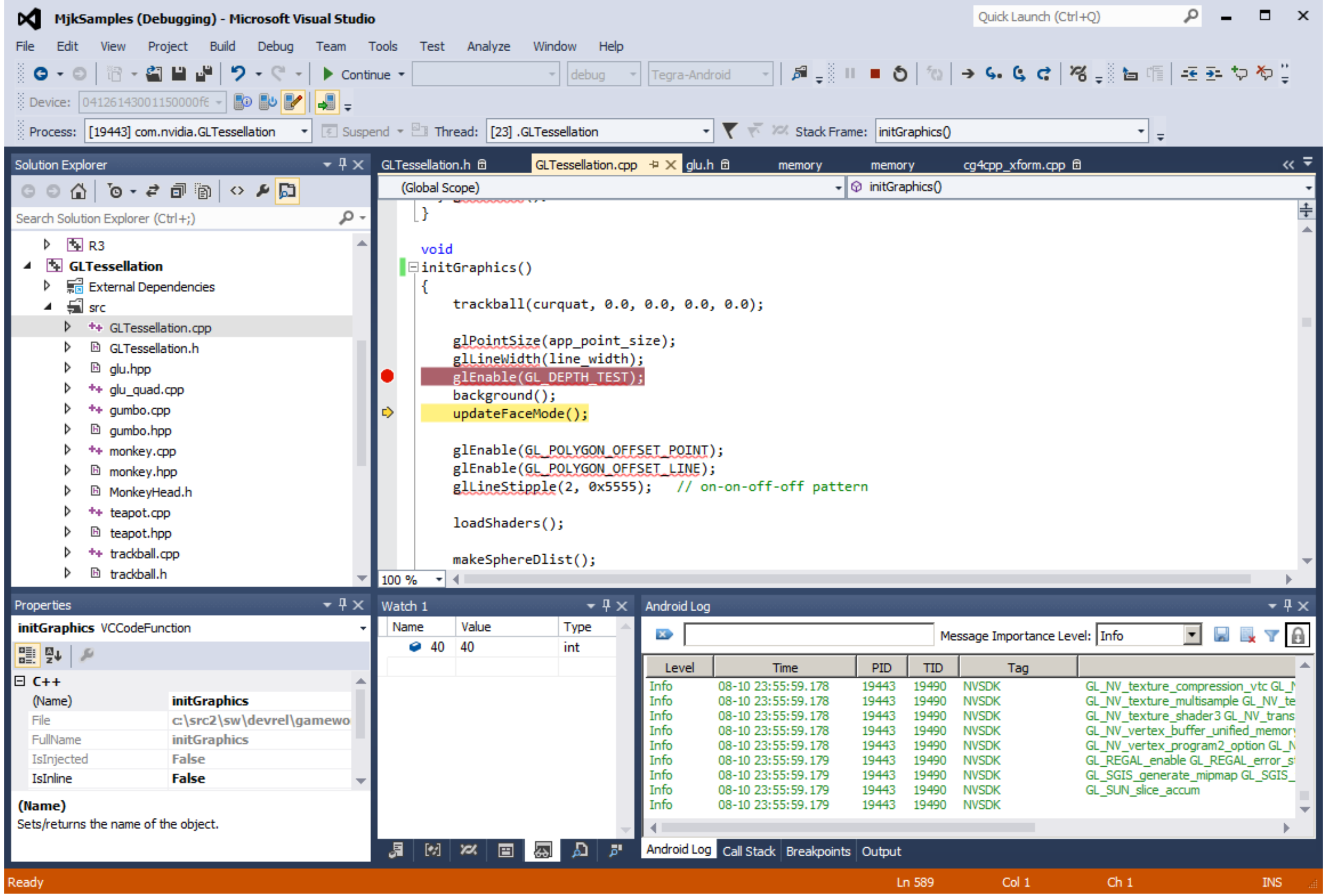
CUDA interop

DirectX interop

GAMEWORKS FRAMEWORK

Build, deploy and debug Android code right from Visual Studio





CONCLUSION

Over to you

We’re super-excited about Vulkan

Can’t wait to see what you do with it!

GO

NVIDIA