What can Vulkan* do for you?

Jason Ekstrand - Embedded Linux Conference - February 22, 2017

What is the Vulkan* API?

Vulkan is a new 3-D rendering and compute api from Khronos, the same cross-industry group that maintains OpenGL

- Redesigned from the ground-up; It is *not* OpenGL++
- Designed for modern GPUs and software
- Designed for both desktop and embedded use-cases
- Will run on currently shipping (GL ES 3.1 class) hardware

Why do we need a new 3-D API?

- OpenGL* 1.0 was released by SGI in January of 1992
  - Based on the proprietary IRIS GL API
  - Heavily state-machine based
  - No real window system story
- OpenGL ES 1.0 was released in July of 2003
  - Based on OpenGL 1.4 but designed for embedded applications
  - Brought a unified EGL window system layer
- OpenGL ES 2.0 was released in March of 2007
  - Fully programmable pipeline (roughly equivalent to GL 3.0)
  - Not compatible with OpenGL ES 1.0/1.1
- OpenGL ES 3.2 was released in August of 2015

Why do we need a new 3-D API?

OpenGL* has done amazingly well over the last 25 years!

Not everything in OpenGL has stood the test of time:

- The OpenGL is API is a state machine
- OpenGL state is tied to a single on-screen context
- OpenGL hides *everything* the GPU is doing

This all made sense in 1992!

*Other names and brands may be claimed as the property of others.

Why do we need a new 3-D API?

Much has changed since 1992:

- Multithreading is now common-place
  - A state machine based on a singleton context doesn’t thread well
- Off-screen rendering is a thing
  - Why do I need to talk to X11 to get a context?
- GPU hardware is much more standardized
  - You don’t ***need*** to hide everything
  - App developers ***don’t want*** you to hide everything

OpenGL* has adapted as well as it can

Why do we need a new 3-D API?

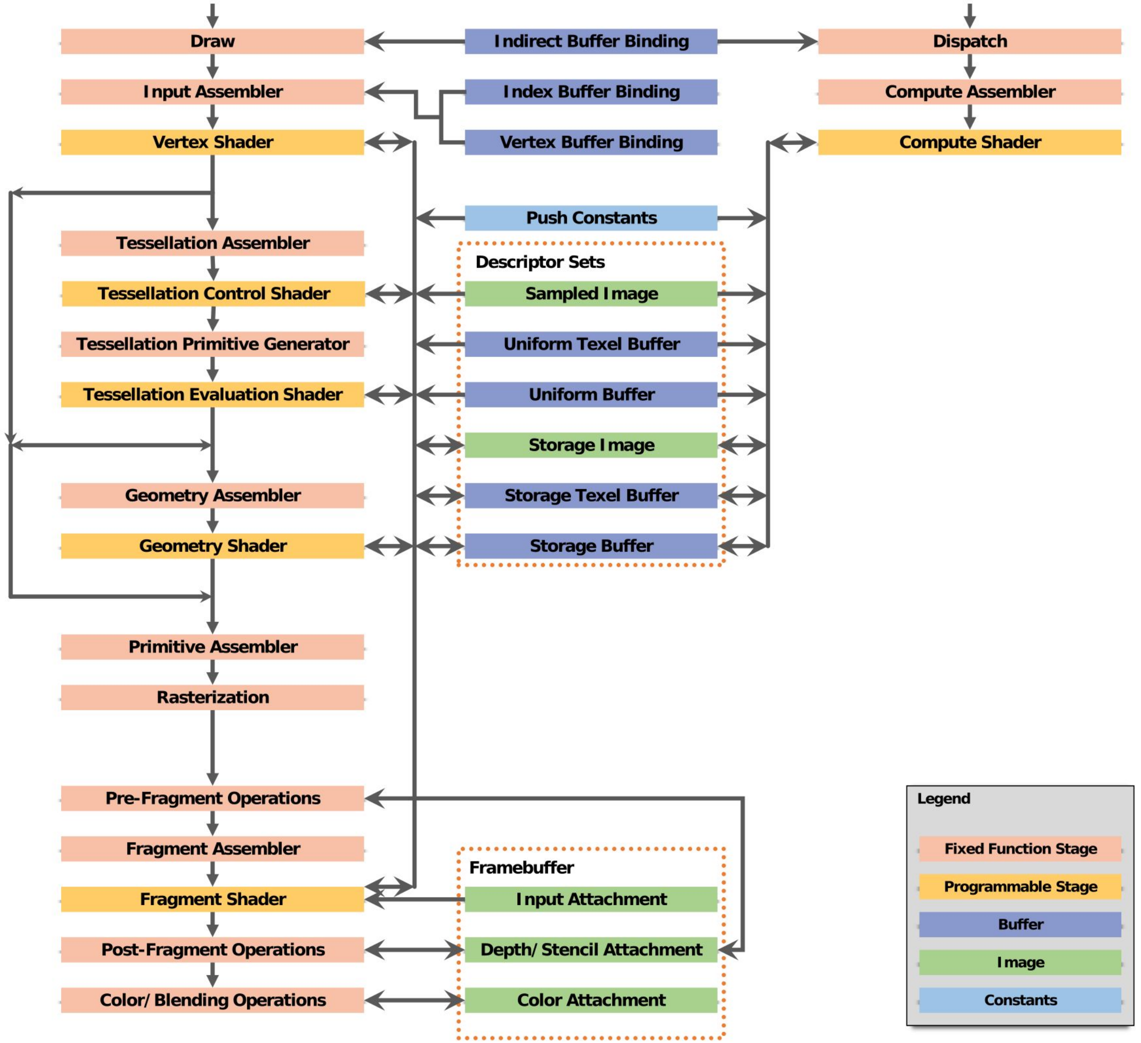
Vulkan* takes a different approach:

- Vulkan is an object-based API with no global state
  - All state concepts are localized to a command buffer
- WSI is an extension of Vulkan, not the other way round.
- Vulkan far more explicit about what the GPU is doing
  - Texture formats, memory management, and syncing are client-controlled
  - Enough is hidden to maintain cross-platform compatibility
- Vulkan drivers do no error checking!

What makes Vulkan* better?

We're going to focus on a four things:

- Pipelines
- Render passes
- Multithreading and synchronization
- Error handling (or the lack thereof)


Draw
Indirect Buffer Binding
Dispatch
Input Assembler
Index Buffer Binding
Compute Assembler
Vertex Shader
Vertex Buffer Binding
Compute Shader
Push Constants
Tessellation Assembler
Descriptor Sets
Tessellation Control Shader
Sampled Image
Tessellation Primitive Generator
Uniform Texel Buffer
Tessellation Evaluation Shader
Uniform Buffer
Storage Image
Geometry Assembler
Storage Texel Buffer
Geometry Shader
Storage Buffer
Primitive Assembler
Rasterization
Pre-Fragment Operations
Legend
Fragment Assembler
Framebuffer
Fixed Function Stage
Fragment Shader
Input Attachment
Programmable Stage
Buffer
Post-Fragment Operations
Depth/Stencil Attachment
Image
Color/Blending Operations
Color Attachment
Constants

Pipelines

Where do these come from?

Where do these go?

What happens between stages?

```
#version 450

layout(location=0) in vec4 a_vertex;
layout(location=1) in vec2 a_tex;
uniform mat4 u_matrix;
layout(location=0) out vec2 v_tex;

void main()
{
   v_tex = a_tex;
   gl_Position = u_matrix * a_vertex;
}
```

Pipelines

All of this is implementation-dependent!

Frequently, “fixed function” stages are implemented in shaders:

- Vertex fetch
- Color blending
- Alpha test
- And more…

All of the above are controlled by **state** not shader code.

Pipelines

So you're doing some rendering...

You cou call `glDrawArrays` and the driver:

1. Examines the currently bound shaders
2. Examines various bits of context state
3. Decides it needs to spend 100ms compiling a new shader

You just missed vblank and your app visibly stutters

Pipelines

Vulkan's* solution: The `VkPipeline` object:

- A monolithic object describing the entire pipeline
- Contains shaders for all stages (vertex, fragment, etc.)
- Contains linkage information
  - Vertex input layout
  - Render target formats
  - Resource descriptor layouts (textures, UBOs, etc.)
- Contains most of the pipeline state
  - Color blending
  - Depth and stencil tests

Pipelines

Isn’t this far less flexible than the state model?

- More data must be provided up-front
- Many pipelines must be created per-shader because of state

Yes, but it comes with several advantages:

- A pipeline contains everything needed to compile shaders
- Common data can be shared via a `VkPipelineCache`
- A `VkPipelineCache` can be easily serialized and written to disk

Pipelines

Pipelines bring predictability to the API:

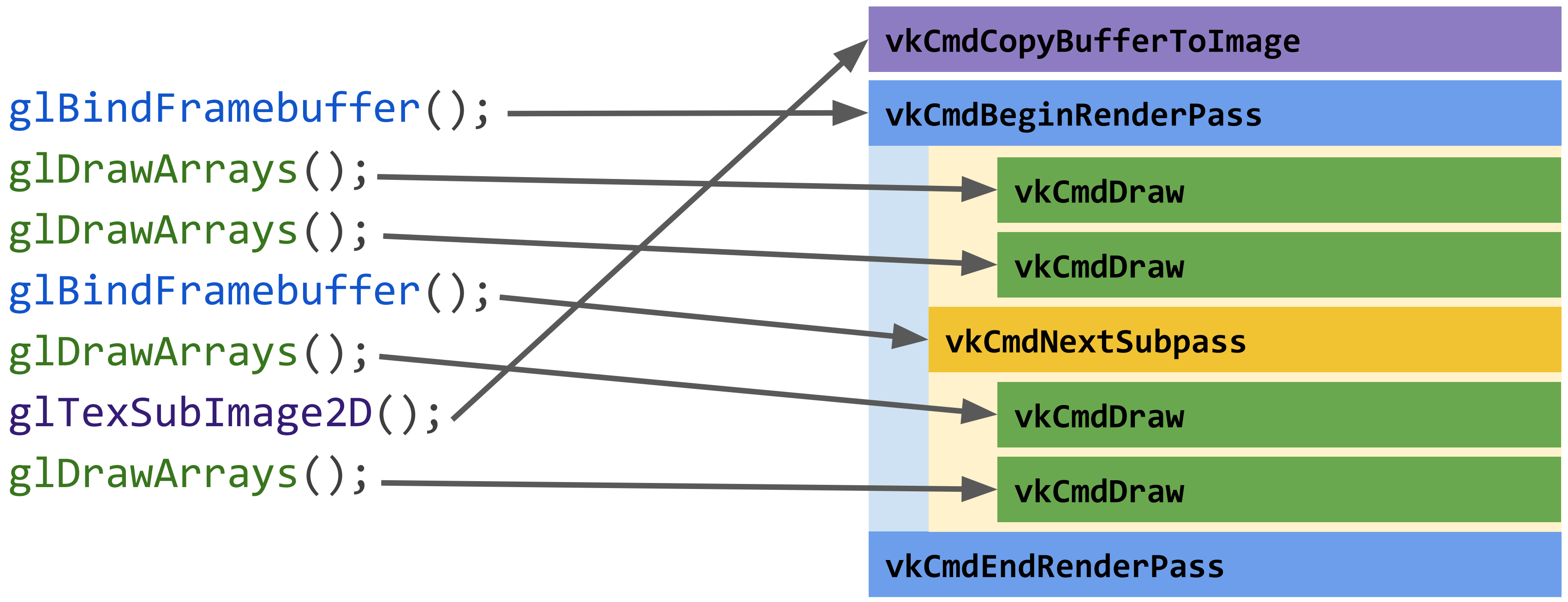
- All shader compilation happens in `vkCreateGraphicsPipelines`
- Drivers have less work to do at draw time
- Using `VkPipelineCache` serialization can almost completely remove shader compilation from application start-up time

Render passes

Render passes are a concept fairly unique to Vulkan*:

- Structures rendering into passes and subpasses
    - Each subpass has its own render targets
    - Render target information is declared up-front
    - Dependencies between subpasses are explicit
- Forces the application to render “nicely”
- Provides extra information to the implementation

Render passes



Render passes

Why require this structure?

- Changing framebuffers can be expensive
- Copy operations (texture uploads etc.) may implicitly require changing framebuffers
- Improves parallelism by removing pixel dependencies
  - An entire render pass can be run one pixel at a time
  - Tiling architectures split rendering into small chunks
- Reduces driver “guesswork”

Multithreading and synchronization

Vulkan* is object-based, not state-based:

- Most objects are immutable
- The only stateful object is the command buffer
- Command buffers can be built in parallel
- The only synchronization point is `vkQueueSubmit`
- Command buffers may even execute in parallel

Multithreading and synchronization

Synchronization is handled by the client:

- Client must synchronize around `vkQueueSubmit`
- Synchronization between GPU and GPU or CPU and GPU is done using fences and semaphores
- Client is responsible for ensuring GPU resources remain alive so long as the GPU is using them.

Error handling

Many APIs do “lazy” error handling

OpenGL* is a state machine

- Non-fatal errors must leave the context in a known state
- Non-fatal OpenGL errors do not change state
- Most OpenGL API misuse is non-fatal
- OpenGL drivers do a lot of up-front error checking
- For well-behaved apps, this is all wasted CPU cycles

Error handling

Vulkan* drivers don’t handle errors:

- Any API misuse may result in a crash or worse
- Invalid synchronization may result in GPU hangs

A set of API validation layers is provided by Khronos:

- Perform an extensive set of API valid usage checks
- Provides costly “deep validation” checks

Validation can be used during development and removed for release

What makes Vulkan* better?

Vulkan is designed to be light-weight and low-overhead:

- Pipelines give more predictable performance and faster load times
- Render passes provide structure and avoids driver guess-work
- Vulkan natively multithreads
- No CPU cycles are wasted on pointless run-time error checks

Don’t waste valuable CPU cycles on driver overhead!

Status of Vulkan* and open-source

Vulkan was released on Feb. 16, 2016

- Four day-one conformant implementations:
  - Imagination
  - Intel
  - NVIDIA
  - Qualcom
- Intel had a conformant open-source Linux* driver on day 1!
- Tools, tests, and validation layers released open-source
- Two day-one AAA game titles: *Dota 2* and *The Talos Principle*

*Other names and brands may be claimed as the property of others.

Status of Vulkan* and open-source

	Linux	Source	Git history	Community
Vulkan spec	✓	✓	✗	✗
Intel Linux driver	✓	✓	✓	✓
Other drivers	✓	✗	✗	✗
Vulkan Loader	✓	✓	✓	✓
SPIR-V Tools	✓	✓	✓	✓
Vulkan conformance tests	✓	✓	✓	75%
Vulkan validation layers	✓	✓	✓	✓

Status of Vulkan* and open-source

Much has happened in the last year:

- Seven conformant implementations:
  - AMD, ARM, Imagination, Intel, NVIDIA, Qualcomm, VeriSilicon
- Intel still has the only conformant open-source implementation
- Validation layers and other tools are much better
- *Doom* has joined the list of AAA titles
- Many game engines are porting to Vulkan
  - CryEngine, id Tech 4, Serious 4, Source 2, Unity 5, Unreal 4, Xenko, ...

Status of Vulkan* and open-source

The open-source community has embraced Vulkan:

- Many open-source Vulkan demos
- Community-developed, open-source radeon driver
- Open-source games/engines
  - vkQuake, Intrinsic, Xenko, ...
- Open-source N64 and PS1 emulators using Vulkan compute
- Open-source D3D9 over Vulkan implementation
- Open-source libraries and tools
  - Renderdoc, VKTS, ...
- ...

intel®