Acknowledgements

First of all, thanks to the inaugural class of 19 students who braved new, unrefined, and just-in-time course materials to take the first Vulkan class at Oregon State University – Winter Quarter, 2018. Thanks for your courage and patience!

Second, thanks to NVIDIA for all of their support!

Third, thanks to the Khronos Group for the great laminated Vulkan Quick Reference Cards! (Look at those happy faces in the photo holding them.)
History of Shaders

2004: OpenGL 2.0 / GLSL 1.10 includes Vertex and Fragment Shaders

2008: OpenGL 3.0 / GLSL 1.30 adds features left out before

2010: OpenGL 3.3 / GLSL 3.30 adds Geometry Shaders

2010: OpenGL 4.0 / GLSL 4.00 adds Tessellation Shaders

2012: OpenGL 4.3 / GLSL 4.30 adds Compute Shaders

2017: OpenGL 4.6 / GLSL 4.60

There is lots more detail at:

History of Shaders

2014: Khronos starts Vulkan effort

2016: Vulkan 1.0

2016: Vulkan 1.1

2020: Vulkan 1.2

There is lots more detail at:
https://en.wikipedia.org/wiki/Vulkan_(API)
Top Three Reasons that Prompted the Development of Vulkan

1. Performance
2. Performance
3. Performance

Vulkan is better at keeping the GPU busy than OpenGL is. OpenGL drivers need to do a lot of CPU work before handing work off to the GPU. Vulkan lets you get more power from the GPU card you already have.

This is especially important if you can hide the complexity of Vulkan from your customer base and just let them see the improved performance. Thus, Vulkan has had a lot of support and interest from game engine developers, 3rd party software vendors, etc.

As an aside, the Vulkan development effort was originally called “glNext”, which created the false impression that this was a replacement for OpenGL. It’s not.
Why is it so important to keep the GPU Busy?

**Nvidia Titan V Specs vs. Titan Xp, 1080 Ti**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Titan V</th>
<th>Tesla V100</th>
<th>Tesla P100</th>
<th>GTX 1080 Ti</th>
<th>GTX 1080</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>NVLink</td>
<td>NVLink</td>
<td>NVLink</td>
<td>NVLink</td>
<td>NVLink</td>
</tr>
<tr>
<td>Transistor Count</td>
<td>2.1 Tm</td>
<td>2.1 Tm</td>
<td>2.1 Tm</td>
<td>2.1 Tm</td>
<td>2.1 Tm</td>
</tr>
<tr>
<td>Fwidth</td>
<td>53.6 nm</td>
<td>53.6 nm</td>
<td>53.6 nm</td>
<td>53.6 nm</td>
<td>53.6 nm</td>
</tr>
<tr>
<td>GPU Convertor Speeds</td>
<td>8.6 Gbps</td>
<td>8.6 Gbps</td>
<td>8.6 Gbps</td>
<td>11 Gbps</td>
<td>11 Gbps</td>
</tr>
<tr>
<td>IBs</td>
<td>384</td>
<td>384</td>
<td>384</td>
<td>384</td>
<td>384</td>
</tr>
<tr>
<td>MRR</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Memory</td>
<td>12 GB</td>
<td>12 GB</td>
<td>12 GB</td>
<td>12 GB</td>
<td>12 GB</td>
</tr>
<tr>
<td>Memory Clock</td>
<td>900 MHz</td>
<td>900 MHz</td>
<td>900 MHz</td>
<td>900 MHz</td>
<td>900 MHz</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>28 GB/s</td>
<td>28 GB/s</td>
<td>28 GB/s</td>
<td>28 GB/s</td>
<td>28 GB/s</td>
</tr>
<tr>
<td>Total Power Budget</td>
<td>250 W</td>
<td>250 W</td>
<td>250 W</td>
<td>250 W</td>
<td>250 W</td>
</tr>
<tr>
<td>Power Connectors</td>
<td>1 x 6pin</td>
<td>1 x 6pin</td>
<td>1 x 6pin</td>
<td>1 x 6pin</td>
<td>1 x 6pin</td>
</tr>
<tr>
<td>Release</td>
<td>June 2017</td>
<td>June 2017</td>
<td>June 2017</td>
<td>June 2017</td>
<td>June 2017</td>
</tr>
</tbody>
</table>

The table above shows the specifications of the Titan V, Tesla V100, and Tesla P100 compared to the GTX 1080 Ti and GTX 1080 GPUs. The key differences are in the number of transistors, clock speeds, and power requirements. The Titan V is designed for high-performance computing tasks, while the Tesla V100 and P100 are optimized for data center applications. The GTX 1080 Ti and GTX 1080 are more suited for gaming and professional workstations. The diagram includes a comparison of the performance metrics such as clock speed, memory bandwidth, and power consumption. The Titan V is the most powerful among the four GPUs, with the highest clock speed and memory bandwidth. It is designed for tasks that require high computational power, such as deep learning and scientific simulations.
Who was the original Vulcan?

From WikiPedia:

“Vulcan is the god of fire including the fire of volcanoes, metalworking, and the forge in ancient Roman religion and myth. Vulcan is often depicted with a blacksmith's hammer. The Vulcanalia was the annual festival held August 23 in his honor. His Greek counterpart is Hephaestus, the god of fire and smithery. In Etruscan religion, he is identified with Sethlans. Vulcan belongs to the most ancient stage of Roman religion: Varro, the ancient Roman scholar and writer, citing the Annales Maximi, records that king Titus Tatius dedicated altars to a series of deities among which Vulcan is mentioned.”

The Khronos Group, Inc. is a non-profit member-funded industry consortium, focused on the creation of open standard, royalty-free application programming interfaces (APIs) for authoring and accelerated playback of dynamic media on a wide variety of platforms and devices. Khronos members may contribute to the development of Khronos API specifications, vote at various stages before public deployment, and accelerate delivery of their platforms and applications through early access to specification drafts and conformance tests.

Playing “Where’s Waldo” with Khronos Membership
Who’s Been Specifically Working on Vulkan?

Vulkan

• Originally derived from AMD’s Mantle API
• Also heavily influenced by Apple’s Metal API and Microsoft’s DirectX 12
• Goal: much less driver complexity and overhead than OpenGL has
• Goal: much less user hand-holding
• Goal: higher single-threaded performance than OpenGL can deliver
• Goal: able to do multithreaded graphics
• Goal: able to handle tiled rendering
Vulkan Differences from OpenGL

- More low-level information must be provided (by you!) in the application, rather than the driver
- Screen coordinate system is Y-down
- No “current state”, at least not one maintained by the driver
- All of the things that we have talked about being deprecated in OpenGL are really deprecated in Vulkan: built-in pipeline transformations, begin-end, fixed-function, etc.
- You must manage your own transformations.
- All transformation, color and texture functionality must be done in shaders.
- Shaders are pre-“half-compiled” outside of your application. The compilation process is then finished during the runtime pipeline-building process.

The Basic OpenGL Computer Graphics Pipeline, OpenGL-style

Vertex, Normal, Color

MC → WC → EC → EC

Model Transform → View Transform → Per-vertex Lighting → Projection Transform

Framebuffer

Fragment Processing, Texturing, Per-fragment Lighting

MC = Model Vertex Coordinates
WC = World Vertex Coordinates
EC = Eye Vertex Coordinates
The Basic Computer Graphics Pipeline, Shader-style

- **gl_Vertex, gl_Normal, gl_Color** Per-vertex in variables
- **gl_ModelViewMatrix, gl_ProjectionMatrix, gl_ModelViewProjectionMatrix** Uniform Variables

**Vertex Shader**

- **MC** Model Transform
- **WC** World Transform
- **EC** Eye Transform

**Fragment Shader**

- **gl_Position, Per-vertex out variables**
- **Framebuffer**

**Uniform Variables**

- **gl_ModelViewMatrix**, **gl_ProjectionMatrix**, **gl_ModelViewProjectionMatrix**

MC = Model Vertex Coordinates
WC = World Vertex Coordinates
EC = Eye Vertex Coordinates

---

The Basic Computer Graphics Pipeline, Vulkan-style

- **Per-vertex in variables**
- **Uniform Variables**

**Vertex Shader**

- **Framebuffer**

**Fragment Shader**

- **Output color(s)**
- **Per-fragment in variables**

**Uniform Variables**

- **gl_ModelViewMatrix**, **gl_ProjectionMatrix**, **gl_ModelViewProjectionMatrix**

---
Moving part of the driver into the application

Complex drivers lead to
driver overhead and
cross vendor
unpredictability
Error management is
always active
Driver processes full
shading language source
Separate APIs for
desktop and mobile
markets

OpenGL

Vulkan

Application

Traditional
graphics
drivers include
significant
context, memory
and error
management

Application

responsible for
memory
allocation and
thread
management to
generate
command buffers

Direct GPU
Control

GPU

GPU

Simpler drivers for low-
overhead efficiency and
cross vendor portability
Layered architecture so
validation and debug
layers can be unloaded
when not needed
Run-time only has to
ingest SPIR-V
intermediate language
Unified API for mobile,
desktop, console and
embedded platforms

Khronos Group

Vulkan Highlights: Command Buffers

- Graphics commands are sent to command buffers
- E.g., `vkCmdDoSomething(cmdBuffer, ...);`
- You can have as many simultaneous Command Buffers as you want
- Buffers are flushed to Queues when the application wants them to be flushed
- Each command buffer can be filled from a different thread
In OpenGL, your “pipeline state” is the combination of whatever your current graphics attributes are: color, transformations, textures, shaders, etc.

Changing the state on-the-fly one item at-a-time is very expensive.

Vulkan forces you to set all your state variables at once into a “pipeline state object” (PSO) data structure and then invoke the entire PSO at once whenever you want to use that state combination.

Think of the pipeline state as being immutable.

Potentially, you could have thousands of these pre-prepared pipeline state objects.
uint32_t count;
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT (VkPhysicalDevice *)nullptr );

VkPhysicalDevice * physicalDevices = new VkPhysicalDevice[ count ];
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT physicalDevices );

This way of querying information is a recurring OpenCL and Vulkan pattern (get used to it):

<table>
<thead>
<tr>
<th>How many total there are</th>
<th>Where to put them</th>
</tr>
</thead>
<tbody>
<tr>
<td>result = vkEnumeratePhysicalDevices( Instance, &amp;count, nullptr );</td>
<td></td>
</tr>
<tr>
<td>result = vkEnumeratePhysicalDevices( Instance, &amp;count, physicalDevices );</td>
<td></td>
</tr>
</tbody>
</table>

Vulkan Code has a Distinct “Style” of Setting Information in structs and then Passing that Information as a pointer-to-the-struct

```c
VkBufferCreateInfo vbci;
vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
vbci.pNext = nullptr;
vbci.flags = 0;
vbci.size = << buffer size in bytes >>
vbci.usage = VK_USAGE_UNIFORM_BUFFER_BIT;
vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
vbci.queueFamilyIndexCount = 0;
vbci.pQueueFamilyIndices = nullptr;

VK_RESULT result = vkCreateBuffer ( LogicalDevice, IN &vbci, PALLOCATOR, OUT &Buffer );

VkMemoryRequirements vmr;
result = vkGetBufferMemoryRequirements( LogicalDevice, Buffer, OUT &vmr ); // fills vmr

VkMemoryAllocateInfo vmai;
vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
vmai.pNext = nullptr;
vmai.flags = 0;
vmai.allocationSize = vmr.size;
vmai.memoryTypeIndex = 0;

result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &MatrixBufferMemoryHandle );
result = vkBindBufferMemory( LogicalDevice, Buffer, MatrixBufferMemoryHandle, 0 );
```
Vulkan Quick Reference Card – I Recommend you Print This!


Vulkan Quick Reference Card

Steps in Creating Graphics using Vulkan

1. Create the Vulkan Instance
2. Setup the Debug Callbacks
3. Create the Surface
4. List the Physical Devices
5. Pick the right Physical Device
6. Create the Logical Device
7. Create the Uniform Variable Buffers
8. Create the Vertex Data Buffers
9. Create the texture sampler
10. Create the texture images
11. Create the Swap Chain
12. Create the Depth and Stencil Images
13. Create the RenderPass
14. Create the Framebuffer(s)
15. Create the Descriptor Set Pool
16. Create the Command Buffer Pool
17. Create the Command Buffer(s)
18. Read the shaders
19. Create the Descriptor Set Layouts
20. Create and populate the Descriptor Sets
21. Create the Graphics Pipeline(s)
22. Update-Render-Update-Render- ...

Vulkan GPU Memory

- Your application allocates GPU memory for the objects it needs
- To write and read that GPU memory, you map that memory to the CPU address space
- Your application is responsible for making sure that what you put into that memory is actually in the right format, is the right size, has the right alignment, etc.
Vulkan Render Passes

- Drawing is done inside a render pass
- Each render pass contains what framebuffer attachments to use
- Each render pass is told what to do when it begins and ends

Vulkan Compute Shaders

- Compute pipelines are allowed, but they are treated as something special (just like OpenGL treats them)
- Compute passes are launched through dispatches
- Compute command buffers can be run asynchronously
Vulkan Synchronization

- Synchronization is the responsibility of the application
- Events can be set, polled, and waited for (much like OpenCL)
- Vulkan itself does not ever lock – that’s your application’s job
- Threads can concurrently read from the same object
- Threads can concurrently write to different objects

Vulkan Shaders

- GLSL is the same as before … almost
- For places it’s not, an implied
  \[\texttt{#define VULKAN 100}\]
  is automatically supplied by the compiler
- You pre-compile your shaders with an external compiler
- Your shaders get turned into an intermediate form known as SPIR-V (Standard Portable Intermediate Representation for Vulkan)
- SPIR-V gets turned into fully-compiled code at runtime
- The SPIR-V spec has been public for years – new shader languages are surely being developed
- OpenCL and OpenGL have adopted SPIR-V as well

GLSL Source  \(\rightarrow\) External GLSL Compiler  \(\rightarrow\) SPIR-V  \(\rightarrow\) Compiler in driver  \(\rightarrow\) Vendor-specific code

Advantages:
1. Software vendors don’t need to ship their shader source
2. Software can launch faster because half of the compilation has already taken place
3. This guarantees a common front-end syntax
4. This allows for other language front-ends
The "19" refers to the version of Visual Studio, not the year of development.