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! The GeForce 1080ti cards are what made this course possible.

Third, thanks to Kathleen Mattson and the Khronos Group for the great laminated Vulkan Quick Reference Cards! (Look at those happy faces in the photo holding them.)
What Prompted the Move to 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?

<table>
<thead>
<tr>
<th>NVidia Titan V Specs vs. Titan Xp, 1080 Ti</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Titan V</strong></td>
</tr>
<tr>
<td>GPU</td>
</tr>
<tr>
<td>Transistor Count</td>
</tr>
<tr>
<td>Peak Performance</td>
</tr>
<tr>
<td><strong>CUDA Cores / Tensor Cores</strong></td>
</tr>
<tr>
<td>( \text{BPU} )</td>
</tr>
<tr>
<td>Core Clock</td>
</tr>
<tr>
<td>Boost Clock</td>
</tr>
<tr>
<td>PIPU TFLOPS</td>
</tr>
<tr>
<td>Memory Type</td>
</tr>
<tr>
<td>Memory Capacity</td>
</tr>
<tr>
<td>Memory Clock</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
</tr>
<tr>
<td>Total Power Budget ( (\text{TOP}) )</td>
</tr>
<tr>
<td>Power Connectors</td>
</tr>
<tr>
<td>Release Date</td>
</tr>
<tr>
<td>Reference Price</td>
</tr>
</tbody>
</table>

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

• Largely 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 – Vulkan can crash
• 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, texture functionality must be done in shaders.
• Shaders are pre-“half-compiled” outside of your application. The compilation process is then finished during the pipeline-building process.
The Basic OpenGL Computer Graphics Pipeline, OpenGL-style

Vertex, Normal, Color

ModelViewMatrix, ProjectionMatrix, ModelViewProjectionMatrix

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

gl_ModelViewMatrix, gl_ProjectionMatrix, gl_ModelViewProjectionMatrix

gl_Position, Per-vertex out variables

Vertex Shader

gl_ModelViewMatrix, gl_ProjectionMatrix, gl_ModelViewProjectionMatrix

Uniform Variables

Fragment Shader

Per-fragment in variables

Framebuffer

gl_FragColor

Per-fragment in variables

Uniform Variables

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

The pipeline consists of several stages:

1. **Vertex Shader**
   - **Per-vertex in variables**: gl_Position, Per-vertex out variables
   - **Uniform Variables**: Uniform Variables

2. **Rasterization**
   - **Output color(s)**: Framebuffer

3. **Fragment Shader**
   - **Per-fragment in variables**: Uniform Variables
   - **Out**: Output color(s)

A Complete API Redesign

<table>
<thead>
<tr>
<th>Original Architecture</th>
<th>Vulkan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Originally architected for graphics workstations with direct renderers and split memory</td>
<td>Matches architecture of modern platforms including mobile platforms with unified memory, tiled rendering</td>
</tr>
<tr>
<td>Driver does lots of work: state validation, dependency tracking, error checking. Limits and randomizes performance</td>
<td>Explicit API – the application has direct, predictable control over the operation of the GPU</td>
</tr>
<tr>
<td>Threading model doesn’t enable generation of graphics commands in parallel to command execution</td>
<td>Multi-core friendly with multiple command buffers that can be created in parallel</td>
</tr>
<tr>
<td>Syntax evolved over twenty years – complex API choices can obscure optimal performance path</td>
<td>Removing legacy requirements simplifies API design, reduces specification size and enables clear usage guidance</td>
</tr>
<tr>
<td>Shader language compiler built into driver. Only GLSL supported. Have to ship shader source</td>
<td>SPIR-V as compiler target simplifies driver and enables front-end language flexibility and reliability</td>
</tr>
<tr>
<td>Despite conformance testing, developers must often handle implementation variability between vendors</td>
<td>Simpler API, common language front-ends, more rigorous testing increase cross vendor functional/performance portability</td>
</tr>
</tbody>
</table>

Khronos Group
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

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

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

Vulkan Highlights: Command Buffers

- Graphics commands are sent to command buffers
- Think OpenCL…
- E.g., `vkCmdDoSomething(cmdBuffer, ...);`
- You can have as many simultaneous Command Buffers as you want
- Buffers are flushed when the application wants them flushed
- Each command buffer can be filled from a different thread (i.e., filling is thread-safe)
Vulkan Highlights: Pipelines

- In OpenGL, your "pipeline state" is 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 at once into a "pipeline state object" (PSO) and then invoke the entire PSO 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 states
- This is a good time to talk about how game companies view Vulkan...

```
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]:

```
result = vkEnumeratePhysicalDevices( Instance, &count, nullptr );
result = vkEnumeratePhysicalDevices( Instance, &count, physicalDevices );
```
Vulkan Code has a Distinct “Style”

```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, &MatrixBufferMemoryHandle );
result = vkBindBufferMemory( LogicalDevice, Buffer, MatrixBufferMemoryHandle, 0 );
```

Vulkan Quick Reference Card

Vulkan Highlights: a More Typical Block Diagram

Steps in Creating Graphics using Vulkan

1. Create the 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: Creating a Pipeline

Vulkan GPU Memory

- Your application allocates GPU memory for the objects it needs
- You map GPU memory to the CPU address space for access
- Your application is responsible for making sure what you put into that memory is actually in the right format, is the right size, has the right alignment, etc.

From the OpenGL Shader Storage Buffer notes:

```c
glGenBuffers(1, &posSSbo);
gBindBuffer(GL_SHADER_STORAGE_BUFFER, posSSbo);
gBufferData(GL_SHADER_STORAGE_BUFFER, NUM_PARTICLES * sizeof(struct pos), NULL, GL_STATIC_DRAW);
GLint bufMask = GL_MAP_WRITE_BIT | GL_MAP_INVALIDATE_BUFFER_BIT; // the invalidate makes a big difference when re-writing
struct pos *points = (struct pos *)glMapBufferRange(GL_SHADER_STORAGE_BUFFER, 0, NUM_PARTICLES * sizeof(struct pos), bufMask);
```
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
- Multiple render passes can be merged

Vulkan Compute Shaders

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

- Vulkan tries to run “flat out”
- Therefore, synchronization is the responsibility of the application
- Events can be set, polled, and waited for (much like OpenCL)
- Vulkan does not ever lock – that’s the 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
  ```
  #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 months – new shader languages are surely being developed
- OpenCL and OpenGL will be moving to SPIR-V as well

---

**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
Your Sample2017.zip File Contains This

<table>
<thead>
<tr>
<th>Name</th>
<th>Date modified</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>vt</td>
<td>12/27/2017 7:45 PM</td>
<td>File folder</td>
<td>149 KB</td>
</tr>
<tr>
<td>Debug</td>
<td>1/2/2018 13:03 PM</td>
<td>File folder</td>
<td>149 KB</td>
</tr>
<tr>
<td>gbu.h</td>
<td>10/16/2016 13:03 AM</td>
<td>C++ Header</td>
<td>130 KB</td>
</tr>
<tr>
<td>gbu.lib</td>
<td>6/15/2017 12:00 PM</td>
<td>Library</td>
<td>130 KB</td>
</tr>
<tr>
<td>gbu.lib.headers</td>
<td>10/16/2017 13:03 PM</td>
<td>Library</td>
<td>130 KB</td>
</tr>
<tr>
<td>gbu.lib.headers</td>
<td>6/15/2017 12:00 PM</td>
<td>Library</td>
<td>130 KB</td>
</tr>
<tr>
<td>sample.cpp</td>
<td>1/2/2018 13:03 PM</td>
<td>C++ Source</td>
<td>130 KB</td>
</tr>
<tr>
<td>Sample.cpio</td>
<td>12/27/2017 7:45 AM</td>
<td>Microsoft Visual Studio</td>
<td>2 KB</td>
</tr>
<tr>
<td>Sample.cpio.files</td>
<td>12/27/2017 7:45 AM</td>
<td>C++ Project</td>
<td>1 KB</td>
</tr>
<tr>
<td>simpleТИO.js</td>
<td>1/1/2018 15:00 AM</td>
<td>SPV File</td>
<td>1 KB</td>
</tr>
<tr>
<td>simple.log</td>
<td>1/1/2018 15:00 AM</td>
<td>SPV File</td>
<td>1 KB</td>
</tr>
<tr>
<td>simple.win.srt</td>
<td>1/1/2018 15:00 AM</td>
<td>SPV File</td>
<td>1 KB</td>
</tr>
<tr>
<td>simple.wts</td>
<td>1/1/2018 15:00 AM</td>
<td>SPV File</td>
<td>1 KB</td>
</tr>
<tr>
<td>SimplexData.cpp</td>
<td>1/2/2018 13:03 PM</td>
<td>C++ Source</td>
<td>1 KB</td>
</tr>
<tr>
<td>vk_priority.h</td>
<td>12/27/2017 7:45 AM</td>
<td>C++ Header</td>
<td>1 KB</td>
</tr>
<tr>
<td>vk_priority.h</td>
<td>12/27/2017 7:45 AM</td>
<td>C++ Header</td>
<td>1 KB</td>
</tr>
<tr>
<td>vk_platform.h</td>
<td>12/27/2017 7:45 AM</td>
<td>C++ Header</td>
<td>1 KB</td>
</tr>
<tr>
<td>vk_platform.h</td>
<td>12/27/2017 7:45 AM</td>
<td>C++ Header</td>
<td>1 KB</td>
</tr>
<tr>
<td>vk_platform.h</td>
<td>12/27/2017 7:45 AM</td>
<td>C++ Header</td>
<td>1 KB</td>
</tr>
<tr>
<td>vulkan.exe</td>
<td>12/27/2017 7:45 AM</td>
<td>C++ Header</td>
<td>1 KB</td>
</tr>
<tr>
<td>vulkan.exe</td>
<td>12/27/2017 7:45 AM</td>
<td>C++ Header</td>
<td>1 KB</td>
</tr>
<tr>
<td>vulkan.exe</td>
<td>12/27/2017 7:45 AM</td>
<td>C++ Header</td>
<td>1 KB</td>
</tr>
<tr>
<td>vulkan.exe</td>
<td>12/27/2017 7:45 AM</td>
<td>C++ Header</td>
<td>1 KB</td>
</tr>
<tr>
<td>VulkanDebug.bat</td>
<td>1/2/2018 13:03 PM</td>
<td>Text Document</td>
<td>1 KB</td>
</tr>
</tbody>
</table>