Vulkan and its GLSL 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.
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.”

Why Name it after the God of the Forge?

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.

Who is the Khronos Group?
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

**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 Computer Graphics Pipeline, Vulkan-style

- **Per-vertex in variables**
- **Uniform Variables**
- **Vertex Shader**
- **Rasterization**
- **Per-fragment in variables**
- **Uniform Variables**
- **Fragment Shader**
- **Framebuffer**
- **Output color(s)**

### 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 “graphics pipeline data structure” (GPDS) and then invoke the entire GPDS 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 GPDSs

---

**Vulkan Highlights: Creating a Graphics Pipeline Data Structure**

VkPipelineCreateInfo

- **Shader stages**
  - VulkanCreatePipeline
  - PipelineState
  - Shader

- **InputAssembly State**
  - Topology
  - InputAssemblyState

- **Tesselation State**
  - TesselationState

- **Viewport State**
  - ViewportState

- **Rasterization State**
  - RasterizationState

- **MultiSample State**
  - MultiSampleState

- **ColorBlend State**
  - ColorBlendState

- **Dynamic State**
  - DynamicState

- **Pipeline layout**
  - PipelineLayout

- **RenderPass**
  - RenderPass

- **basePipelineHandle**
  - basePipelineHandle

- **basePipelineIndex**
  - basePipelineIndex

---

**Vulkan Highlights: Graphics Pipeline Data Structures**

- 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 “graphics pipeline data structure” (GPDS) and then invoke the entire GPDS 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 GPDSs
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: GLSL Differences from OpenGL

Detecting that a GLSL Shader is being used with Vulkan/SPIR-V:

- In the compiler, there is an automatic
  ```
  #define VULKAN 100
  ```

**Vertex and Instance indices:**

- Both are 0-based

**gl_FragColor:**

- In OpenGL, it broadcasts to all color attachments
- In Vulkan, it just broadcasts to color attachment location #0
- Best idea: don’t use it – explicitly declare out variables to have specific location numbers

**Descriptor Sets:**

```c
layout( set=0, binding=0 ) . . . ;
```

**Push Constants:**

```c
layout( push_constant ) . . . ;
```
Vulkan: Shaders’ use of Layouts for Uniform Variables

```cpp
layout( std140, set = 0, binding = 0 ) uniform matBuf
{
    mat4 uModelMatrix;
    mat4 uViewMatrix;
    mat4 uProjectionMatrix;
    mat3 uNormalMatrix;
} Matrices;

// non-opaque must be in a uniform block:
layout( std140, set = 1, binding = 0 ) uniform lightBuf
{
    vec4 uLightPos;
} Light;

layout( set = 2, binding = 0 ) uniform sampler2D uTexUnit;
```

All opaque (non-sampler) uniform variables must be in block buffers

Vulkan Shaders

- 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 program launch time (the driver does this)
- The SPIR-V spec has been public for years – new shader languages are surely being developed (HLSL is already there.)
- OpenCL and OpenGL have adopted 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
SPIR-V:
Standard Portable Intermediate Representation for Vulkan

```
```

Shaderfile extensions:
- .vert  Vertex
- .tesc  Tessellation Control
- .tese  Tessellation Evaluation
- .geom  Geometry
- .frag  Fragment
- .comp  Compute

(Can be overridden by the –S option)

- V     Compile for Vulkan
- G     Compile for OpenGL
- I     Directory(ies) to look in for #includes
- S     Specify stage rather than get it from shaderfile extension
- c     Print out the maximum sizes of various properties

---

The Famous Vulkan Bunny Button