Why Do Some People Want to Replace OpenGL?

• OpenGL is good for high-level users who care more about hiding the details of the process than they do about performance.

• OpenGL has a variety of ways to do things – some more efficient than others

• All of OpenGL’s methods keep you at arm’s length from the details.

• OpenGL protects you from the consequences of making mistakes.

• OpenGL often has to guess at what you are intending to do in order to implement structures in the best way.

• OpenGL funnels all graphics operators through a single thread.
What's the General Idea of Vulkan?

- Power-performance users need an API that lets them do as many graphics operations as close to the hardware as possible with very little overhead.
- Developers take responsibility for protecting themselves from making catastrophic mistakes.

Vulkan Features

- Direct access to GPU memory – put what you want where you want it. Access it and interpret it as you want it.
- Multiple command buffers to access multiple features of the graphics hardware – pipeline, memory, shaders, etc.
- Allow multiple threads to be filling those command buffers simultaneously.
- High efficiency, much simpler, more reliable drivers.
- GLSL stays about the same.
- GLSL shaders get pre-compiled into an intermediate format. At runtime, the driver finishes compiling that intermediate format.
- Starting with version 2.1, OpenCL uses that same intermediate format.
- Lots of other “languages” could too.
Vulkan!

Playing "Where's Waldo" with Khronos Membership
Vulkan

- Largely derived from AMD’s Mantle API
- Also heavily influenced by 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

A Complete API Redesign

<table>
<thead>
<tr>
<th>Originally architected for graphics workstations with direct renderers and split memory</th>
<th>Matches architecture of modern platforms including mobile platforms with unified memory, tiled rendering</th>
</tr>
</thead>
<tbody>
<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>
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

Vulkan Code is Very “User-supplied Information Rich”

```c
VkSubmitInfo submit_info = {
    .sType = VK_STRUCTURE_TYPE_SUBMIT_INFO,
    .pNext = NULL,
    .waitSemaphoreCount = 0,
    .pWaitSemaphores = NULL,
    .pWaitDstStageMask = NULL,
    .commandBufferCount = 1,
    .pCommandBuffers = cmd_bufs,
    .signalSemaphoreCount = 0,
    .pSignalSemaphores = NULL
};

err = vkQueueSubmit( queue, 1, &submit_info, nullFence );
```
Vulkan 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 they are full or when the application wants them flushed
- Each command buffer can be filled from a different thread (i.e., filling is thread-safe)

Vulkan Pipelines

- In OpenGL, your “pipeline state” is whatever your current state is: 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 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…
Vulkan GPU Memory

- Your application allocates GPU memory for the objects it needs
- You map 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, etc.

From the Shader Storage Buffer notes:

```c
glGenBuffers( 1, &posSSbo );
glBindBuffer( 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

- Synchronization is the responsibility of the application
- Events can be set, polled, and waited for (much like OpenCL)
- Vulkan does not ever block – 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

Vulkan Shader Compiling

- You pre-compile your shaders with an external compiler
- Your shaders get turned into an intermediate form known as SPIR-V
- SPIR-V gets turned into fully-compiled code at runtime
- SPIR-V spec has been public for months – new shader languages are surely being developed
- OpenCL will be moving to SPIR-V as well

Advantages:
1. Game vendors don’t need to ship their shader source
2. This guarantees a common front-end syntax (sort of)
So What Do We All Do Now?

- I don’t see Vulkan replacing OpenGL ever
- However, I wonder if Khronos will become less and less excited about adding new extensions to OpenGL
- And, I also wonder if vendors will become less and less excited about improving OpenGL drivers
- If I ever teach a Vulkan class at OSU, I see it as a one-term standalone course, not part of another OpenGL-based course

So What Do We All Do Now?

This is what I think the model of the immediate future is:

You → Engine → Application

Open GL

You

Vulkan

Application