A (really too short) Introduction to Vulkan

by Ronell Sicat
Schedule

- **Monday**
  - What is Vulkan?
  - How to draw a triangle?
    - **Vulkan objects**
    - Code snippets
- **Wednesday**
  - Quiz
  - How to draw a triangle?
    - Line-by-line look at example code
    - Demo
  - More Vulkan demos and resources
  - Project ideas
What is Vulkan?

- Cross-platform graphics and compute API for GPUs by Khronos group
- Low-level API for better performance and more predictable behaviour
One API to rule them all...

Non-proprietary, royalty-free open standard ‘By the industry for the industry’
Portable across multiple platforms - desktop and mobile
Modern architecture | Low overhead | Multi-thread friendly
EXPLICIT GPU access for EFFICIENT, LOW-LATENCY, PREDICTABLE performance

Vulkan

Vulkan Porting Tools

Windows 10 | Windows 8 | Windows 7 | redhat | ubuntu | SteamOS | Nintendo Switch | Android | iOS | macOS
Anatomy of a graphics application (recap)
OpenGL: simplified pipeline model (recap)

**Vertices**
- **Vertex Processing**
  - glGen...
  - glVertex...

**Rasterization**

**Fragments**
- **Fragment Processing**
  - glBind...
  - glDraw...

**Geometric and image data**

**Application data and commands from the CPU**

**GPU Data Flow**

**Framebuffer**
How to draw a triangle using Vulkan?

- Create Vulkan objects (initial setup)
  - shader programs
  - buffer objects for loading data
  - and more...

- Render (loop)
  - store commands in command buffer (once)
  - submit command buffer to queue
Vulkan objects

- **Instance**
- **Physical Device**
- **Device**
  - **Device Memory**
  - **Descriptor Pool**
  - **Command Buffer**
  - **Command Pool**
- **Buffer**
- **Image**
- **Image View**
- **Descriptor Set**
  - **Descriptor Set Layout**
  - **Pipeline Layout**
- **Framebuffer**
- **Render Pass**
- **Shader Module**
- **const std::vector<Vertex> vertices = {**
  
  ```
  {{{0.0f, -0.5f}, {1.0f, 0.0f, 0.0f}},
  {{{0.5f, 0.5f}, {0.0f, 1.0f, 0.0f}},
  {{{-0.5f, 0.5f}, {0.0f, 0.0f, 1.0f}}}
  ```

vertex and fragment shaders
Object creation API convention

```cpp
VkXXXCreateInfo createInfo = {};  
createInfo.sType = VK_STRUCTURE_TYPE_XXX_CREATE_INFO;  
createInfo.pNext = nullptr;  
createInfo.foo = ...;  
createInfo.bar = ...;

VkXXX object;
if (vkCreateXXX(&createInfo, nullptr, &object) != VK_SUCCESS) {
    std::cerr << "failed to create object" << std::endl;
    return false;
}
```

```cpp
GLuint objectId;
glGenObject(1, &objectId);
```
- **VkInstance** is your application’s connection to the Vulkan loader that communicates with the driver
  - describe application
  - specify validation layers and API extensions

```cpp
VkApplicationInfo appInfo = {};
appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
appInfo.pApplicationName = "Hello Triangle";
appInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
appInfo.pEngineName = "No Engine";
appInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
appInfo.apiVersion = VK_API_VERSION_1_0;

VkInstanceCreateInfo createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
createInfo.pApplicationInfo = &appInfo;
createInfo.enabledLayerCount = static_cast<uint32_t>(validationLayers.size());
createInfo.ppEnabledLayerNames = validationLayers.data();
createInfo.enabledExtensionCount = static_cast<uint32_t>(requiredExtensions.size());
createInfo.ppEnabledExtensionNames = requiredExtensions.data();

if (vkCreateInstance(&createInfo, nullptr, &instance) != VK_SUCCESS) {
    throw std::runtime_error("failed to create instance!");
}
```
Validation layers

- Vulkan has limited error checking and debugging capabilities by default for high performance and low overhead - will often crash instead of returning error code!
- Validation layers are pieces of code that
  - can be inserted between API and driver for checks
  - can be completely disabled for zero overhead
  - are usually provided by SDK
- **VkPhysicalDevice** represents a specific Vulkan-compatible GPU
  - available GPUs can be enumerated using `vkEnumeratePhysicalDevices`
  - device properties (e.g., driver version, device limits) can be enumerated using `vkGetPhysicalDeviceProperties`
  - features (e.g., geometry shader, texture compression, etc.) can be enumerated using `vkGetPhysicalDeviceFeatures`
  - can enumerate supported memory types
- **VkDevice** (logical device) - main object that represents an initialized Vulkan device that is ready to create all other objects
  - specify queues to be requested (graphics, compute, transfer)
  - specify device extensions to enable, e.g.
    - VK_KHR_swapchain
    - VK_NV_mesh_shader

```cpp
VkDeviceQueueCreateInfo queueInfo = { VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO };
queueInfo.queueFamilyIndex = familyIndex;
queueInfo.queueCount = 1;

VkDeviceCreateInfo createInfo = { VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO };
createInfo.queueCreateInfoCount = 1;
createInfo.pQueueCreateInfos = &queueInfo;
createInfo.ppEnabledExtensionNames = extensions.data();
createInfo.enabledExtensionCount = uint32_t(extensions.size());

VkDevice device = 0;
VK_CHECK(vkCreateDevice(physicalDevice, &createInfo, 0, &device));
```
VkQueue is an object used for issuing commands to GPU for asynchronous execution

- Work is requested by filling command buffers and submitting them to the queue via `vkQueueSubmit`
- Multiple queues, e.g., graphics and compute, allow for asynchronous execution which can lead to performance improvements
- Developer is in charge of synchronization using fences and semaphores
- **VkCommandPool** - object used to allocate command buffers
- **VkCommandBuffer** - stores a list of commands to be executed by the device
  - commands start with `vkCmd`, e.g.,
    - `vkCmdBeginRenderPass`
    - `vkCmdBindPipeline`
    - `vkCmdBindDescriptorSets`
    - `vkCmdBindVertexBuffers`
    - `vkCmdBindIndexBuffer`
    - `vkCmdDraw`
    - `vkCmdEndRenderPass`
- **VkBuffer** - container for any binary data (e.g., vertex data, index data)
- **VkImage** - container for pixel data (e.g., texture, render target)
- Steps for creating buffers or images:
  - create buffer or image
  - allocate device memory (VkDeviceMemory)
  - bind them together using `vkBindBufferMemory` or `vkBindImageMemory`
- **VkDeviceMemory** - actual memory allocation which satisfy one or more type flags:
  - device local (GPU only; fastest)
  - host visible (CPU -> GPU)
  - host cached (GPU -> CPU)
  - host coherent (GPU <-> CPU)
- **VkBufferView** - describes a subset of buffer data (range and offset)
- **VkImageView** - describes a subset of image data (layers, mip levels)
- **VkDescriptorSet** - describes a set of buffers and images to be accessed by shaders.
- **VkDescriptorsetLayout** - template for descriptor set binding numbers and shader access flags.

Each **DescriptorSet** holds the references to actual resources.
Recap
- **VkRenderPass** - describes the number and type of attachments used in rendering
- Attachments (render targets) are images used as output from rendering
  - color attachment (R8G8B8A8_UNORM)
  - depth attachment (D16_UNORM)
- **VkRenderPass** - describes the number and type of attachments used in rendering
- Attachments (render targets) are images used as output from rendering
  - color attachment (R8G8B8A8_UNORM)
  - depth attachment (D16_UNORM)
- **VkFrameBuffer** - represents a link to actual images that can be used as attachments
- **VkPipeline** - represents the configuration of the whole graphics or compute pipeline (almost all parameters need to be set in advance for driver optimization)

- Specifies most of the graphics pipeline parameters such as:
  - primitive topology, e.g., point, triangle
  - raster front face (counter/clockwise)
  - raster cull mode (back/front face)
  - shader stages

- **VkPipelineLayout** - defines the layout of descriptors and push constants
Graphics pipeline

- ~150 lines of code!
- Tradeoff between complex setup and optimal performance.
- **VkShaderModule** - a buffer filled with shader code in SPIR-V format
- **SPIR-V** - open standard intermediate language for parallel compute and graphics; GLSL and HLSL shader code can be compiled to SPIR-V
- You can use your OpenGL-compatible shaders!
Example shaders

**Vertex shader**

```cpp
layout(location = 0) in vec2 inPosition;
layout(location = 1) in vec3 inColor;
layout(location = 0) out vec3 fragColor;

void main() {
    gl_Position = vec4(inPosition, 0.0, 1.0);
    fragColor = inColor;
}
```

**Fragment shader**

```cpp
layout(location = 0) out vec4 outColor;
layout(location = 0) in vec3 fragColor;

void main() {
    outColor = vec4(fragColor, 1.0);
}
```

```cpp
const std::vector<Vertex> vertices = {
    {{0.0f, -0.5f}, {1.0f, 0.0f, 0.0f}},
    {{0.5f, 0.5f}, {0.0f, 1.0f, 0.0f}},
    {{-0.5f, 0.5f}, {0.0f, 0.0f, 1.0f}}
};
```
Command buffer example (draw a triangle!)

```c
vkBeginCommandBuffer(...)
vkCmdBeginRenderPass(...)
vkCmdBindPipeline(...)
vkCmdBindVertexBuffers(...)
vkCmdBindIndexBuffer(...)
vkCmdBindDescriptorSets(...)
vkCmdDraw/Indexed(...)
vkCmdEndRenderPass(...)
vkEndCommandBuffer(...)

glBindObject(objectId);
glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, (void*)0);
```
Recap
Conclusion

● Vulkan is low-level
  ○ lots of control (and responsibility) given to developer
  ○ steep learning curve
  ○ lots of code
● Vulkan is very fast (if used properly)
  ○ 8 million cubes (1 FPS in OpenGL, 250 FPS in Vulkan)
● Usually need to do main setup once (if it works for a triangle, it will work with a bigger object/scene)
● What to use (OpenGL or Vulkan) depends on scenario
Next time we will look at actual code line by line...

https://github.com/Overv/VulkanTutorial

https://github.com/SaschaWillems/Vulkan
References

● https://vulkan-tutorial.com
● https://gpuopen.com/understanding-vulkan-objects/
● https://github.com/SaschaWillems/Vulkan
● https://github.com/Overv/VulkanTutorial
● https://github.com/vinjn/awesome-vulkan

Acknowledgements

● Peter Rautek
Thank you!