A summary

Mateus Surrage Reis
history + context
Why, when and how it was made
history + context
Why, when and how it was made

A quick tour of Vulkan +
The ‘feeling’ of programming with it
Starting from the late 80s, The market was overflowing with video card makers.
Starting from the late 80s, the market was overflowing with video card makers.
Software devs had to adjust for each hardware developer, and they all did things slightly differently. There had to be another way!
Software devs had to adjust for each hardware developer, and they all did things slightly differently. There had to be another way! Eventually, someone did something about it:
Some standards tussling later…
Hardware interfacing
Windowing/input
The point was: Centralization Standardization Simplification
The point was:
Centralization
Standardization
Simplification
Timeline

1992
OpenGL 1 released
Timeline

1992
OpenGL 1 released

1993
Doom released
1992
OpenGL 1 released

1993
Doom released
Timeline

1992
OpenGL 1 released

1993
Doom released

Also I’m born
Timeline

1992
OpenGL 1 released

1993
Doom released

1996
Toy Story
Direct3D

Also I’m born
Timeline

1992
OpenGL 1 released

1993
Doom released

1996
Toy Story Direct3D

2004
OpenGL 2.0
Now with shaders!

Also I’m born
Timeline

1992
OpenGL 1 released

1993
Doom released

1996
Toy Story
Direct3D

2004
OpenGL 2.0
Now with shaders!

The field was rapidly maturing, and the industry’s needs were piling up as well.

OGL steadily gets more and more features

Geometry Shader
Framebuffer Objects
Tessellation
GPU compute shaders
Buffer Textures
Uniform locations
Etc etc
Timeline

1992
OpenGL 1 released

1993
Doom released

1996
Toy Story
Direct3D

2009
Avatar
Borderlands

2004
OpenGL 2.0
Now with shaders!

Also I’m born

OGL steadily gets more and more features
Timeline

1992
OpenGL 1 released

1993
Doom released

Also I'm born

1996
Toy Story
Direct3D

2004
OpenGL 2.0

2009
Avatar
Borderlands

Now with shaders!
Timeline

1992
OpenGL 1 released

1993
Doom released

1996
Toy Story
Direct3D

OGL steadily gets more and more features

2004
OpenGL 2.0
Now with shaders!

2009
Avatar
Borderlands

2015
Vulkan!
(glNext)

Also I’m born
The logical endpoint of the past decade of development. Performance was now the biggest priority.
DirectX12 and Metal follow the same general idea
That’s all for the history.
Now we go into a bit of code
Now we go into a bit of code

1. This is all hello world tutorial code and
2. Don’t bother trying to follow line by line
A very common idiom

```cpp
VkShaderModule createShaderModule(const std::vector<char>& code) {

    VkShaderModuleCreateInfo createInfo{};
    createInfo.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
    createInfo.codeSize = code.size();
    createInfo.pCode = reinterpret_cast<const uint32_t*>(code.data());
    VkShaderModule shaderModule;

    if (vkCreateShaderModule(device, &createInfo, nullptr, &shaderModule) != VK_SUCCESS)
        throw std::runtime_error("failed to create shader module!");
    return shaderModule;
}
```
A very common idiom

```cpp
VkShaderModule createShaderModule(const std::vector<char>& code) {

VkShaderModuleCreateInfo createInfo{};
createInfo.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
createInfo.codeSize = code.size();
createInfo.pCode = reinterpret_cast<const uint32_t*>(code.data());
VkShaderModule shaderModule;

if (vkCreateShaderModule(device, &createInfo, nullptr, &shaderModule) != VK_SUCCESS)
    throw std::runtime_error("failed to create shader module!");
return shaderModule;
```
A very common idiom
A very common idiom

```cpp
VkShaderModule createShaderModule(const std::vector<char>& code) {
    VkShaderModuleCreateInfo createInfo{};
    createInfo.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
    createInfo.codeSize = code.size();
    createInfo.pCode = reinterpret_cast<const uint32_t*>(code.data());
    VkShaderModule shaderModule;

    if (vkCreateShaderModule(device, &createInfo, nullptr, &shaderModule) != VK_SUCCESS) {
        throw std::runtime_error("failed to create shader module!");
    }
    return shaderModule;
}
```
A very common idiom

```cpp
VkShaderModule createShaderModule(const std::vector<char>& code) {
    VkShaderModuleCreateInfo createInfo{};
    createInfo.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
    createInfo.codeSize = code.size();
    createInfo.pCode = reinterpret_cast<const uint32_t*>(code.data());
    VkShaderModule shaderModule;
    if (vkCreateShaderModule(device, &createInfo, nullptr, &shaderModule) != VK_SUCCESS)
        throw std::runtime_error("failed to create shader module!");
    return shaderModule;
}
```
Another one

```cpp
uint32_t deviceCount = 0;
vkEnumeratePhysicalDevices(instance, &deviceCount, nullptr);
if (deviceCount == 0) {
    throw std::runtime_error("failed to find GPUs with Vulkan support!");
}
std::vector<VkPhysicalDevice> devices(deviceCount);
vkEnumeratePhysicalDevices(instance, &deviceCount, devices.data());
```
Another one

```cpp
uint32_t deviceCount = 0;
vkEnumeratePhysicalDevices(instance, &deviceCount, nullptr);
if (deviceCount == 0) {
    throw std::runtime_error("failed to find GPUs with Vulkan support!");
}
std::vector<VkPhysicalDevice> devices(deviceCount);
vkEnumeratePhysicalDevices(instance, &deviceCount, devices.data());
```
Another one

```c++
uint32_t deviceCount = 0;
vkEnumeratePhysicalDevices(instance, &deviceCount, nullptr);
if (deviceCount == 0) {
    throw std::runtime_error("failed to find GPUs with Vulkan support!");
}
std::vector<VkPhysicalDevice> devices(deviceCount);
vkEnumeratePhysicalDevices(instance, &deviceCount, devices.data());
```
| No global state. Everything is held in objects. State is localized. | One big “context” global. |
What do I mean by ‘state is localized’?
What do I mean by ‘state is localized’?

// Turn on depth test, so the objects can // be drawn in front of the objects far
setEnabled(GL_DEPTH_TEST);
// Enable back-face culling
setEnabled(GL_CULL_FACE);
cullFace(GL_BACK);
// Clear our background to black
setColor(0.0f, 0.0f, 0.0f, 1.0f);
What do I mean by ‘state is localized’?
What do I mean by ‘state is localized’?
What do I mean by ‘state is localized’?
What do I mean by ‘state is localized’?
<table>
<thead>
<tr>
<th>No global state. Everything is held in objects. State is localized.</th>
<th>One big “context” global.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multithreading is allowed.</td>
<td>API calls must be made from one thread.</td>
</tr>
</tbody>
</table>
Threads

Old way
Current Frame

Core 1
Game Job  Physics Job  Sim Job  Sim Job  Sim Job

Core 2
Al Job  Physics Job  Sim Job

Core 3
Al Job  Physics Job  Sim Job  Sim Job

Core 4
Game Job  Al Job  Al Job  Sim Job  Sim Job

Core 5
Game Job  Game Job  Game Job  Sim Job  Sim Job

Core 6
Graphics Prev Frame

Dead time

Graphics (Opaque, in driver)

???GPU Fence, or CPU wait???
Threads
<table>
<thead>
<tr>
<th>No global state. Everything is held in objects. State is localized.</th>
<th>One big “context” global.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multithreading is allowed.</td>
<td>API calls must be made from one thread.</td>
</tr>
<tr>
<td>Synchronization and memory layouts are explicit and your job</td>
<td>Synchronization and memory layouts are usually hidden</td>
</tr>
</tbody>
</table>
Synchronization

Things Vulkan does aren’t ordered by default…
So we use synchronization primitives
Synchronization

Things vulkan does aren’t ordered by default…
So we use synchronization primitives

Semaphores
Fences
Events
Barriers
void drawFrame() {
    // device, fencecount, &fencearray, wait for all/any, timeout
    vkWaitForFences(device, 1, &inFlightFence, VK_TRUE, UINT64_MAX);
    vkResetFences(device, 1, &inFlightFence);

    uint32_t imageIndex;
    // device, swapchain, timeout (ns), semaphore,
    vkAcquireNextImageKHR(device, swapChain, UINT64_MAX, imgAvailableSMPH, VK_IN rub
dex);
    // buffer, vkcommandbufferresetflag bit
    vkResetCommandBuffer(cmdBuffer, 0);
    recordCommandBuffer(cmdBuffer, imageIndex);

    VkSubmitInfo submitInfo{};
    submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;

    // these define which semaphores to wait on before execution begins and what
    VkSemaphore waitSemaphores[] = {imgAvailableSMPH};
    VkPipelineStageFlags waitStages[] = {VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT}
    submitInfo.waitSemaphoreCount = 1;
    submitInfo.pWaitSemaphores = waitSemaphores;
    submitInfo.pWaitDstStageMask = waitStages;

    submitInfo.commandBufferCount = 1;
    submitInfo.pCommandBuffers = &cmdBuffer;
}

Things vulkan does aren't ordered by default…
So we use primitives
Semaphores
Fences
Events
Barriers
void drawFrame() {
  // device, fencecount, &fenceraay, wait for all/any, timeout
  vkWaitForFences(device, 1, &inFlightFence, VK_TRUE, UINT64_MAX);
  vkResetFences(device, 1, &inFlightFence);

  uint32_t imageIndex;
  // device, swapchain, timeout (ns), semaphore,
  vkAcquireNextImageKHR(device, swapChain, UINT64_MAX, imgAvailableSMPH, VK_NULL_HANDLE, &imageIndex);
  // buffer, vkcommandbufferresetflag bit
  vkResetCommandBuffer(cmdBuffer, 0);
  recordCommandBuffer(cmdBuffer, imageIndex);

  VkSubmitInfo submitInfo{};
  submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;

  // these define which semaphores to wait on before execution begins and what
  // to wait on
  // we wait until the image is available to write it, so we specify the color

  VkSemaphore waitSemaphores[] = {imgAvailableSMPH};
  VkPipelineStageFlags waitStages[] = {VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT};
  submitInfo.waitSemaphoreCount = 1;
  submitInfo.pWaitSemaphores = waitSemaphores;
  submitInfo.pWaitDstStageMask = waitStages;

  submitInfo.commandBufferCount = 1;
  submitInfo.pCommandBuffers = &cmdBuffer;
Things vulkan does aren't ordered by default…
So we use primitives:

- Semaphores
- Fences
- Events
- Barriers

```c
void drawFrame()
{
    // device, fencecount, &fencearray, wait for all/any, timeout
    vkWaitForFences(device, 1, &inFlightFence, VK_TRUE, UINT64_MAX);
    vkResetFences(device, 1, &inFlightFence);

    uint32_t imageIndex;
    // device, swapchain, timeout (ns), semaphore,
    vkAcquireNextImageKHR(device, swapChain, UINT64_MAX, imgAvailableSMPH, VK_TRUE, &imageIndex);
    // buffer, vkcommandbufferresetflag bit
    vkResetCommandBuffer(cmdBuffer, 0);
    recordCommandBuffer(cmdBuffer, imageIndex);

    VkSubmitInfo submitInfo{};
    submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;

    // these define which semaphores to wait on before execution begins and what to do when
    // we wait on
    // these define which semaphores to wait on before execution begins and what to do when
    VkSemaphore waitSemaphores[] = {imgAvailableSMPH};
    VkPipelineStageFlags waitStages[] = {VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT};
    submitInfo.waitSemaphoreCount = 1;
    submitInfo.pWaitSemaphores = waitSemaphores;
    submitInfo.pWaitDstStageMask = waitStages;

    submitInfo.commandBufferCount = 1;
    submitInfo.pCommandBuffers = &cmdBuffer;
}```
Things Vulkan does aren’t ordered by default…
So we use primitives:

- Semaphores
- Fences
- Events
- Barriers

```c
void drawFrame() {
    // device, fencecount, fencearray, wait for all/any, timeout
    vkWaitForFences(device, 1, &inFlightFence, VK_TRUE, UINT64_MAX);
    vkResetFences(device, 1, &inFlightFence);

    uint32_t imageIndex;
    // device, swapchain, timeout (ns), semaphore,
    vkAcquireNextImageKHR(device, swapChain, UINT64_MAX, VK_TRUE,
                           NULL, &imageIndex);
    // buffer, vkcommandbufferresetflag bit
    vkResetCommandBuffer(cmdBuffer, 0);
    recordCommandBuffer(cmdBuffer, imageIndex);

    VkSubmitInfo submitInfo;
    submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;

    // these define which semaphores to wait on before execution begins and which
    // to wait on
    // we wait until the image is available to write it, so we specify the color
    VkSemaphore waitSemaphores[] = {imgAvailableSMPH};
    VkPipelineStages waitStages[] = {VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT};

    submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
    submitInfo.pWaitSemaphores = waitSemaphores;
    submitInfo.pWaitDstStageMask = &waitStages;
    submitInfo.pCommandBuffers = &cmdBuffer;
```
Memory?

It’s complicated…
VkMemoryRequirements memRequirements;
vkGetBufferMemoryRequirements(device, vertexBuffer, &memRequirements);

VkPhysicalDeviceMemoryProperties memProperties;
vkGetPhysicalDeviceMemoryProperties(physicalDevice, &memProperties);
vkMemoryAllocateInfo allocInfo{};

allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
allocInfo.allocationSize = memRequirements.size;
allocInfo.memoryTypeIndex = findMemoryType(memRequirements.memoryTypeBits, VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT);

VkDeviceMemory vertexBufferMemory;

vkAllocateMemory(device, &allocInfo, nullptr, &vertexBufferMemory)
vkBindBufferMemory(device, vertexBuffer, vertexBufferMemory, 0);
VkMemoryRequirements memRequirements;
vkGetBufferMemoryRequirements(device, vertexBuffer, &memRequirements);

VkPhysicalDeviceMemoryProperties memProperties;
vkGetPhysicalDeviceMemoryProperties(physicalDevice, &memProperties);
VkMemoryAllocateInfo allocInfo{};

allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
allocInfo.allocationSize = memRequirements.size;
allocInfo.memoryTypeIndex = findMemoryType(memRequirements.memoryTypeBits)
  | VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT);

VkDeviceMemory vertexBufferMemory;

vkAllocateMemory(device, &allocInfo, nullptr, &vertexBufferMemory)
vkBindBufferMemory(device, vertexBuffer, vertexBufferMemory, 0);
VkMemoryRequirements memRequirements;
vkGetBufferMemoryRequirements(device, vertexBuffer, &memRequirements);

VkPhysicalDeviceMemoryProperties memProperties;
vkGetPhysicalDeviceMemoryProperties(physicalDevice, &memProperties);
VkMemoryAllocateInfo allocInfo{};

allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
allocInfo.allocationSize = memRequirements.size;
allocInfo.memoryTypeIndex = findMemoryType(memRequirements.memoryTypeBits)
| VK_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT);

VkDeviceMemory vertexBufferMemory;

vkAllocateMemory(device, &allocInfo, nullptr, &vertexBufferMemory)
vkBindBufferMemory(device, vertexBuffer, vertexBufferMemory, 0);
VkMemoryRequirements memRequirements;
vkGetBufferMemoryRequirements(device, vertexBuffer, &memRequirements);
VkPhysicalDeviceMemoryProperties memProperties;
vkGetPhysicalDeviceMemoryProperties(physicalDevice, &memProperties);
VkMemoryAllocateInfo allocInfo{};

allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
allocInfo.allocationSize = memRequirements.size;
allocInfo.memoryTypeIndex = findMemoryType(memRequirements.memoryTypeBits, VK_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT);

VkDeviceMemory vertexBufferMemory;

vkAllocateMemory(device, &allocInfo, nullptr, &vertexBufferMemory)
vkBindBufferMemory(device, vertexBuffer, vertexBufferMemory, 0);
Memory?
It's complicated…

```cpp
VkMemoryRequirements memRequirements;
vkGetBufferMemoryRequirements(device, vertexBuffer, &memRequirements);

VkPhysicalDeviceMemoryProperties memProperties;
vkGetPhysicalDeviceMemoryProperties(physicalDevice, &memProperties);
VkMemoryAllocateInfo allocInfo{};

allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
allocInfo.allocationSize = memRequirements.size;
allocInfo.memoryTypeIndex = findMemoryType(memRequirements.memoryTypeBits
    .HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT);

VkDeviceMemory vertexBufferMemory;

vkAllocateMemory(device, &allocInfo, nullptr, &vertexBufferMemory)
vkBindBufferMemory(device, vertexBuffer, vertexBufferMemory, 0);
```
<table>
<thead>
<tr>
<th>No global state. Everything is held in objects. State is localized.</th>
<th>One big “context” global.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multithreading is allowed.</td>
<td>API calls must be made from one thread.</td>
</tr>
<tr>
<td>Synchronization and memory layouts are explicit and your job</td>
<td>Synchronization and memory layouts can be hidden</td>
</tr>
<tr>
<td>No error-checking is done. (Fast)</td>
<td>Extensive error-checking goes on internally. (Slow)</td>
</tr>
</tbody>
</table>
GL_STACK_OVERFLOW, 0x0503
Given when a stack pushing operation cannot be done because it would overflow the stack.

GL_STACK_UNDERFLOW, 0x0504
Given when a stack popping operation cannot be done because the stack is already empty.

GL_OUT_OF_MEMORY, 0x0505
Given when performing an operation that can allocate memory, and the memory cannot be allocated. OpenGL functions that return this error are undefined; it is allowable for partial execution to be performed in this circumstance.

GL_INVALID_FRAMEBUFFER_OPERATION, 0x0506
Given when doing anything that would attempt to read from or write/render to a framebuffer other than the current one.

GL_CONTEXT_LOST, 0x0507 (with OpenGL 4.5 or ARB_KHR_robustness)
Given if the OpenGL context has been lost, due to a graphics card reset.

GL_TABLE_TOO_LARGE, 0x8031
Part of the ARB_imaging extension.
GL_STACK_OVERFLOW, 0x0503
Given when a stack pushing operation cannot be done because it would overflow the stack.

GL_STACK_UNDERFLOW, 0x0504
Given when a stack popping operation cannot be done because the stack is already at the bottom.

GL_OUT_OF_MEMORY, 0x0505
Given when performing an operation that can allocate memory, and the memory cannot be allocated. OpenGL functions that return this error are undefined; it is allowable for partial execution to take place in this circumstance.

GL_INVALID_FRAMEBUFFER_OPERATION, 0x0506
Given when doing anything that would attempt to read from or write/render to a framebuffer with invalid resources.

GL_CONTEXT_LOST, 0x0507 (with OpenGL 4.5 or ARB_KHR_robustness)
Given if the OpenGL context has been lost, due to a graphics card reset.

GL_TABLE_TOO_LARGE, 0x8031
Part of the ARB_imaging extension.
Removable validation layers

**GL_STACK_OVERFLOW, 0x0503**
Given when a stack pushing operation cannot be done because it would overflow the stack.

**GL_STACK_UNDERFLOW, 0x0504**
Given when a stack popping operation cannot be done because the stack is already empty.

**GL_OUT_OF_MEMORY, 0x0505**
Given when performing an operation that can allocate memory, and the memory cannot be allocated. OpenGL functions that return this error are undefined; it is allowable for partial execution in this circumstance.

**GL_INVALID_FRAMEBUFFER_OPERATION, 0x0506**
Given when doing anything that would attempt to read from or write/render to a framebuffer that is not a valid framebuffer.

**GL_CONTEXT_LOST, 0x0507 (with OpenGL 4.5 or ARB_KHR_robustness)***
Given if the OpenGL context has been lost, due to a graphics card reset.

**GL_TABLE_TOO_LARGE**, 0x8031
Part of the ARB_imaging extension.
Those are the most important design points.

Now, a few extras:
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- ...
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- …
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- …
Shader Compiler
(inside graphics driver)
Shader Compiler (inside graphics driver)
Shader code

FRONTEND DEALS WITH THE LANGUAGE

BACKEND GENERATES MACHINE CODE

GPU Machine code

GPU exec

GPU
Shader code → SPIR-V COMPILER → BACKEND GENERATES MACHINE CODE → GPU Machine code → GPU exec
Why bother?

- GPU vendors don’t have to write their own parsers.
- Uniformity.
- You can support multiple shader languages with less effort.
- Some optimizations can be done out of the application.
SPIR-V enables a rich ecosystem of languages and compilers to target low-level APIs such as Vulkan and OpenCL, including deployment flexibility: e.g., running OpenCL kernels on Vulkan.
That's all
Thanks