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 overflow
video card

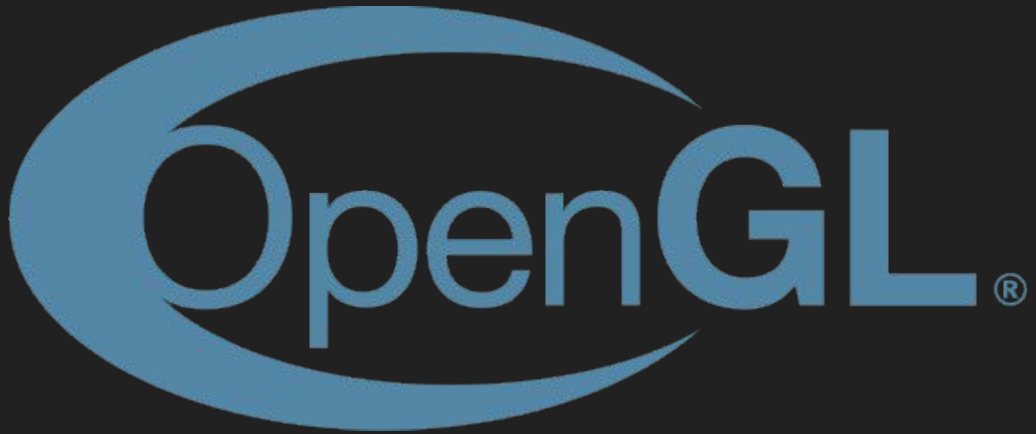
Software devs had to adjust for each hardware developer, and they all did things slightly differently.
There had to be another way!

Software devs had to adjust for each hardware developer, and they all did things slightly differently.

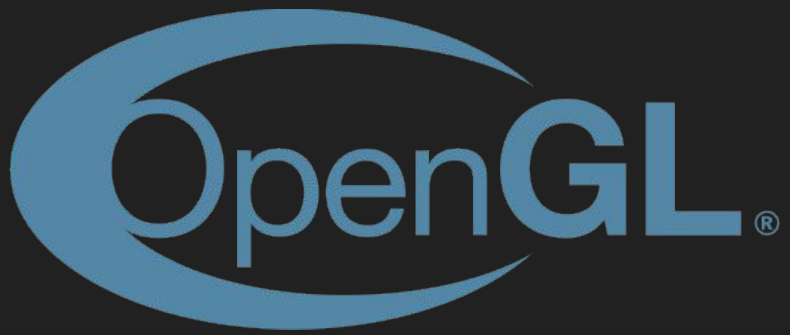
There had to be another way!

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

```c
#include <stdio.h>
#include <GL/glut.h>

void display(void)
{
 glClear( GL_COLOR_BUFFER_BIT);
 glColor3f(0.0, 1.0, 0.0);
 glBegin(GL_POLYGON);
  glVertex3f(2.0, 4.0, 0.0);
  glVertex3f(8.0, 4.0, 0.0);
  glVertex3f(8.0, 6.0, 0.0);
  glVertex3f(2.0, 6.0, 0.0);
 glEnd();
 glFlush();
}

int main(int argc, char **argv)
{
 printf("hello world\n");
 glutInit(&argc, argv);
 glutInitDisplayMode ( GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);

 glutInitWindowPosition(100,100);
 glutInitWindowSize(300,300);
 glutCreateWindow ("square");

 glClearColor(0.0, 0.0, 0.0, 0.0);          // black backgrour
 glMatrixMode(GL_PROJECTION);               // setup viewing p
 glLoadIdentity();                            // start with ic
rix
 glOrtho(0.0, 10.0, 0.0, 10.0, -1.0, 1.0);   // setup a 10x16
 world

 glutDisplayFunc(display);
 glutMainLoop();

 return 0;
}
```

Timeline

1992
OpenGL 1
released

Timeline

1992
OpenGL 1
released

1993
Doom released

1992
OpenGl
release

19
Doom r

Timeline

1992
OpenGL 1
released

1993
Doom released

Also I'm born

Timeline

1992
OpenGL 1
released

1996
Toy Story
Direct3D

1993
Doom released

Also I'm born

Timeline

1992
OpenGL 1
released

1996
Toy Story
Direct3D

1993
Doom released

Also I'm born

2004
OpenGL 2.0
Now with
shaders!

Timeline

1992
OpenGL 1
released

1996
Toy Story
Direct3D

1993
Doom released

Also I'm born

2004
OpenGL 2.0
Now with
shaders!

OGL steadily
gets more and
more features

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

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

Timeline

1992
OpenGL 1
released

1996
Toy Story
Direct3D

OGL steadily
gets more and
more features

1993
Doom released

Also I'm born

2004
OpenGL 2.0
Now with
shaders!

2009
Avatar
Borderlands

1

Ope
rel

Doc

Also I’m born

Timeline

1992
OpenGL 1
released

1996
Toy Story
Direct3D

OGL steadily
gets more and
more features

1993
Doom released

Also I'm born

2004
OpenGL 2.0
Now with
shaders!

2009
Avatar
Borderlands

2015
Vulkan!
(glNext)

The logical endpoint of the past decade~ of development. Performance was now the biggest priority.

DirectX12 and Metal follow the same general idea

Vulkan®


AMD
arm
BROADCOM
arm
CINDER
Imagination
CRYTEK
CRYENGINE
Imagination
intel
NVIDIA.
intel
LUNARG
MoltenVK
The Forge
id TECH
Int
QUALCOMM
VeriSilicon
NVIDIA.
Sascha Willems
source
TORQUE
unity
UNREAL ENGINE
UX3D ENGINE
XENKO

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

```
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

```
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

```
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

```
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

```
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

```
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());
```

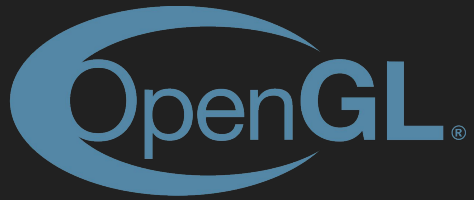
Another one

```
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());
```

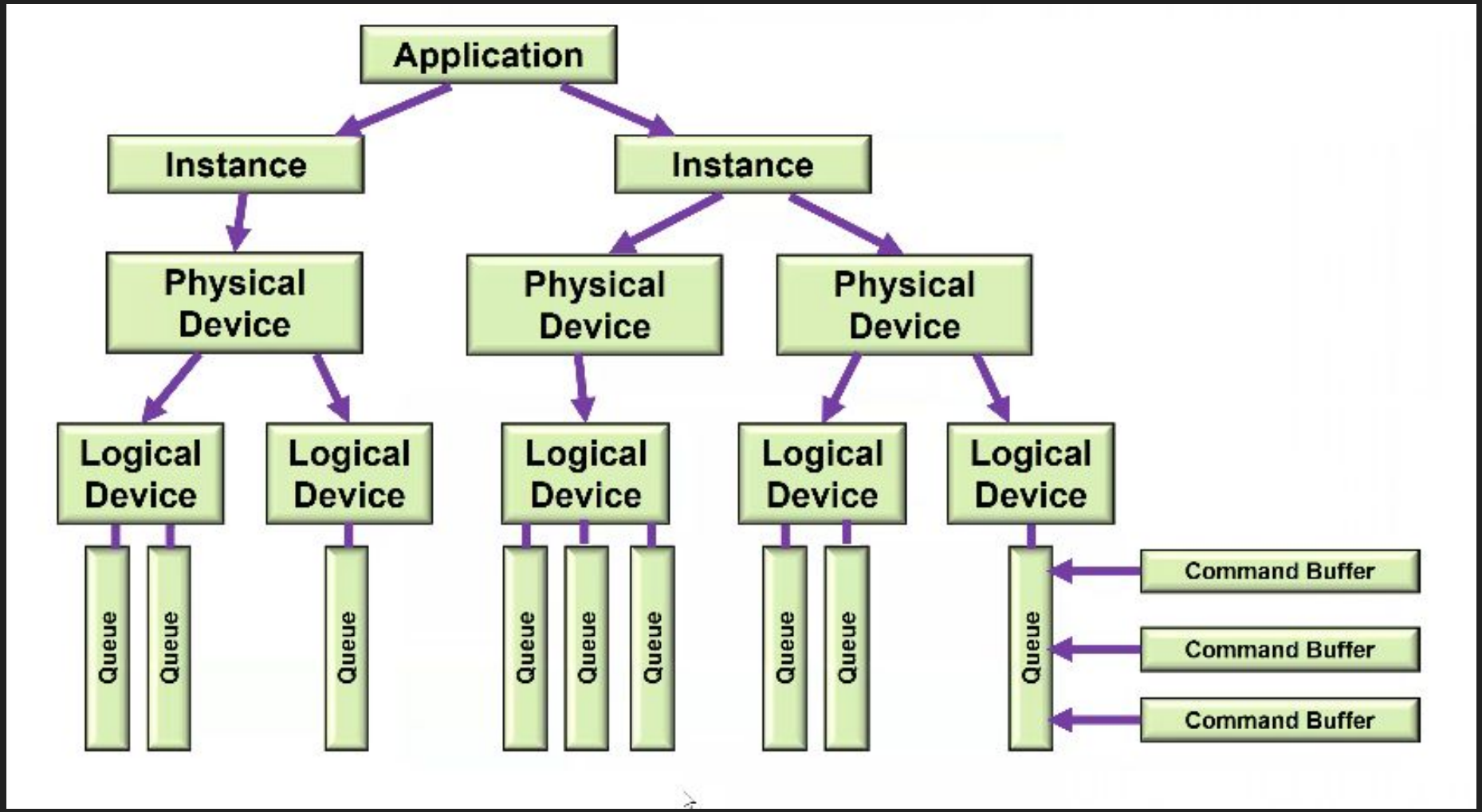
Another one

```
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());
```


Vulkan


OpenGL

Vulkan	OpenGL
No global state. Everything is held in objects. State is localized.	One big “context” global.


Application
Instance
Instance
Physical Device
Physical Device
Physical Device
Logical Device
Logical Device
Logical Device
Logical Device
Logical Device
Queue
Queue
Queue
Queue
Queue
Queue
Queue
Queue
Queue
Command Buffer
Command Buffer
Command Buffer

What do I mean by ‘state is localized’?

What do I mean by ‘state is localized’?

```
//Turn on depth test, so the objects c
//be drawn in front of the objects fur
glEnable(GL_DEPTH_TEST);
//Enable back-face culling
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
//Clear our background to black
glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
```

What do I mean by ‘state is localized’?

```
//Turn on depth test, so the objects c
//be drawn in front of the objects fur
glEnable(GL_DEPTH_TEST);
//Enable back-face culling
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
//Clear our background to black
glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
```

```
// rasterizer also does depth test, face cull, scissor test
// config all that
VkPipelineRasterizationStateCreateInfo rasterizer{};
rasterizer.sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_ST
rasterizer.depthClampEnable = VK_FALSE;
rasterizer.rasterizerDiscardEnable = VK_FALSE;
rasterizer.polygonMode = VK_POLYGON_MODE_FILL;
rasterizer.lineWidth = 1.0f;
rasterizer.cullMode = VK_CULL_MODE_BACK_BIT;
rasterizer.frontFace = VK_FRONT_FACE_CLOCKWISE;
rasterizer.depthBiasEnable = VK_FALSE;
rasterizer.depthBiasConstantFactor = 0.0f;
rasterizer.depthBiasClamp = 0.0f;
rasterizer.depthBiasSlopeFactor = 0.0f;
```

What do I mean by ‘state is localized’?

```
//Turn on depth test, so the
//be drawn in front of the ol
glEnable(GL_DEPTH_TEST);
//Enable back-face culling
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
//Clear our background to bla
glClearColor(0.0f, 0.0f, 0.0f
```

```
// rasterizer also does depth test, face cull, scissor test
// config all that
VkPipelineRasterizationStateCreateInfo rasterizer{};
rasterizer.sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_ST
rasterizer.depthClampEnable = VK_FALSE;
```

```
VkGraphicsPipelineCreateInfo pipelineInfo{};
pipelineInfo.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELI
pipelineInfo.stageCount = 2;
pipelineInfo.pStages = shaderStages;

pipelineInfo.pVertexInputState = &vertexInputInfo;
pipelineInfo.pInputAssemblyState = &inputAssembly;
pipelineInfo.pViewportState = &viewportState;
pipelineInfo.pRasterizationState = &rasterizer;
pipelineInfo.pMultisampleState = &multisampling;
pipelineInfo.pDepthStencilState = nullptr; // Optional
```

What do I mean by ‘state is localized’?

```
//Turn on depth test, so the
//be drawn in front of the ol
glEnable(GL_DEPTH_TEST);
/, if (vkCreateGraphicsPipelines
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
//Clear our background to bla
glClearColor(0.0f, 0.0f, 0.0f
```

```
// rasterizer also does depth test, face cull, scissor test
// config all that
VkPipelineRasterizationStateCreateInfo rasterizer{};
rasterizer.sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_ST
rasterizer.depthClampEnable = VK_FALSE;
```

```
VkGraphicsPipelineCreateInfo pipelineInfo{};
pipelineInfo.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELI
pipelineInfo.stageCount = 2;
device, VK_NULL_HANDLE, 1, &pipelineInfo, nullptr, &graphics

pipelineInfo.pVertexInputState = &vertexInputInfo;
pipelineInfo.pInputAssemblyState = &inputAssembly;
pipelineInfo.pViewportState = &viewportState;
pipelineInfo.pRasterizationState = &rasterizer;
pipelineInfo.pMultisampleState = &multisampling;
pipelineInfo.pDepthStencilState = nullptr; // Optional
```

What do I mean by ‘state is localized’?

```
// rasterizer also does depth test, face cull, scissor test
// config all that
VkPipelineRasterizationStateCreateInfo rasterizer{};
rasterizer.sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_ST
rasterizer.depthClampEnable = VK_FALSE;
```

```
//Turn on depth test, so the
//be drawn in front of the ol
glEnable(GL_DEPTH_TEST);
/, if (vkCreateGraphicsPipelines
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
//Clear our background to bla
glClearColor(0.0f, 0.0f, 0.0f
```

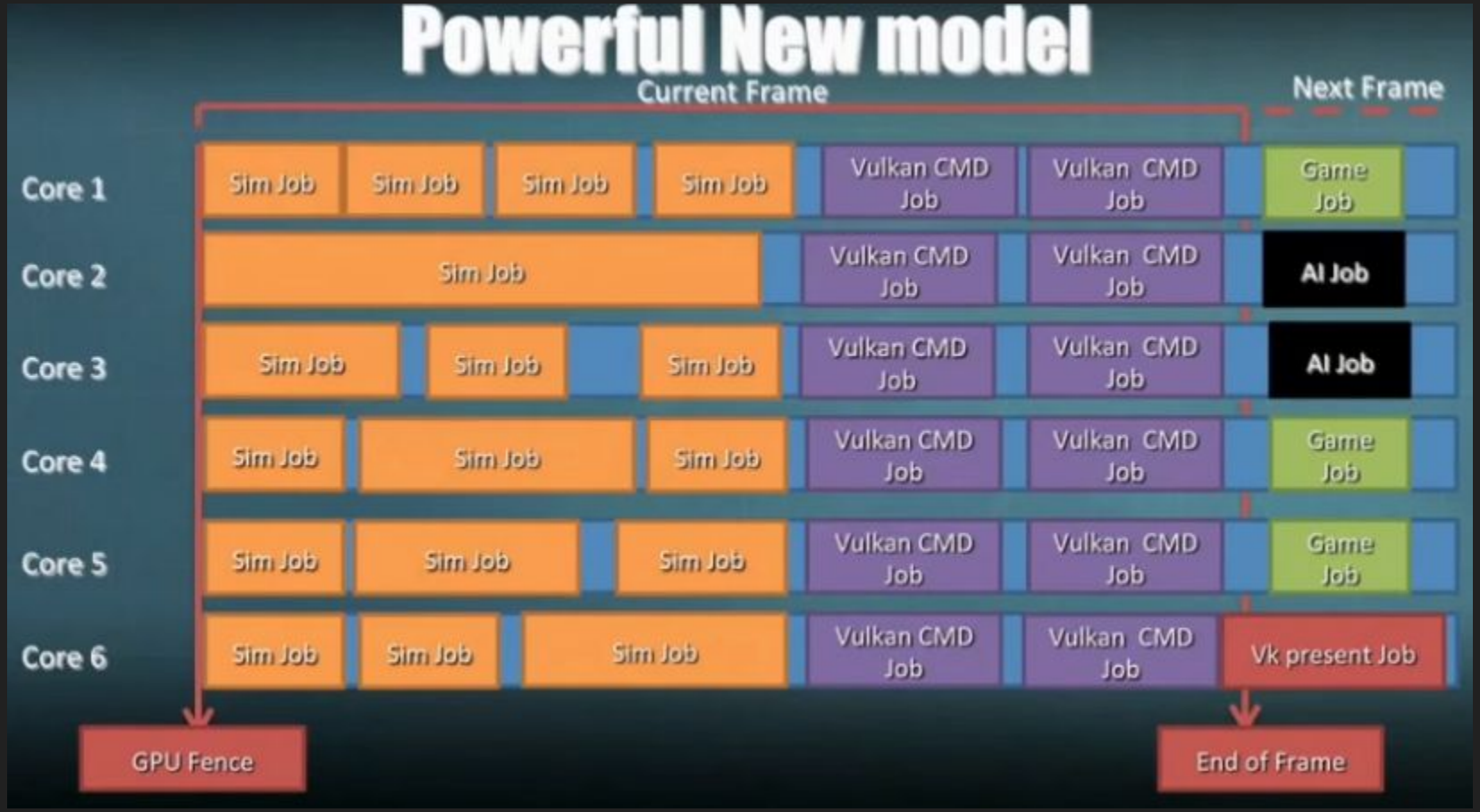
```
VkGraphicsPipelineCreateInfo pipelineInfo{};
pipelineInfo.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELI
pipelineInfo.stageCount = 2;
(device, VK_NULL_HANDLE, 1, &pipelineInfo, nullptr, &graphics

pipelineInfo.pVertexInputState = &vertexInputInfo;
pipelineInfo.pInputAssemblyState = &inputAssembly;
pipelineInfo.pViewportState = &viewportState;
pipelineInfo.pRasterizationState = &rasterizer;
pipelineInfo.pMultisampleState = &multisampling;
pipelineInfo.pDepthStencilState = nullptr; // Optional
```

Vulkan	OpenGL
No global state. Everything is held in objects. State is localized.	One big “context” global.
Multithreading is allowed.	API calls must be made from one thread.

Threads


Old way
Current Frame
Dead time
Core 1
Game Job
Physics Job
Sim Job
Sim Job
Sim Job
Core 2
AI Job
Physics Job
Sim Job
Core 3
AI Job
Physics Job
Sim Job
Sim Job
Core 4
Game Job
AI Job
AI Job
Sim Job
Sim Job
Core 5
Game Job
Game Job
Game Job
Sim Job
Sim Job
Core 6
Graphics Prev Frame
Graphics (Opaque, in driver)
???GPU Fence, or CPU wait???

Threads


Powerful New model
Current Frame
Next Frame
Core 1
Core 2
Core 3
Core 4
Core 5
Core 6
Sim Job
Vulkan CMD Job
Game Job
AI Job
Vk present Job
GPU Fence
End of Frame

Vulkan	OpenGL
No global state. Everything is held in objects. State is localized.	One big “context” global.
Multithreading is allowed.	API calls must be made from one thread.
Synchronization and memory layouts are explicit and your job	Synchronization and memory layouts are usually hidden

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

Things vulkar
So we use pri

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_
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 wh
line
    // to wait on
    // we wait until the image is available to write it, so we specify the col

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

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

Things vulkar
So we use pri

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_
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 wh
line
    // to wait on
    // we wait until the image is available to write it, so we specify the col

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

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

Things vulkar
So we use pri

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_N
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 whi
line
    // to wait on
    // we wait until the image is available to write it, so we specify the colo

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

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

Things vulkar
So we use pri

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_N
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 whi
line
    // to wait on
    // we wait until the image is available to write it, so we specify the colo

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

    submitInfo.commandBufferCount = 1;
    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.memoryTypeBi
Y_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.memoryTypeBi
Y_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.memoryTypeBi
Y_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.memoryTypeBi
Y_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.memoryTypeBi
Y_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT);

VkDeviceMemory vertexBufferMemory;

vkAllocateMemory(device, &allocInfo, nullptr, &vertexBufferMemory)
vkBindBufferMemory(device, vertexBuffer, vertexBufferMemory, 0);
```

Vulkan	OpenGL
No global state. Everything is held in objects. State is localized.	One big “context” global.
Multithreading is allowed.	API calls must be made from one thread.
Synchronization and memory layouts are explicit and your job	Synchronization and memory layouts can be hidden
No error-checking is done. (Fast)	Extensive error-checking goes on internally. (Slow)

GL_STACK_OVERFLOW, 0x0503

Given when a stack pushing operation cannot be done because it would overflow the

GL_STACK_UNDERFLOW, 0x0504

Given when a stack popping operation cannot be done because the stack is already a

GL_OUT_OF_MEMORY, 0x0505

Given when performing an operation that can allocate memory, and the memory can OpenGL functions that return this error are undefined; it is allowable for partial execu in this circumstance.

GL_INVALID_FRAMEBUFFER_OPERATION, 0x0506

Given when doing anything that would attempt to read from or write/render to a fram

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[1], 0x8031

Part of the ARB_imaging extension.

Vulkan®

OpenGL®

```
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
```

GL_STACK_OVERFLOW, 0x0503

Given when a stack pushing operation cannot be done because it would overflow the

GL_STACK_UNDERFLOW, 0x0504

Given when a stack popping operation cannot be done because the stack is already a

GL_OUT_OF_MEMORY, 0x0505

Given when performing an operation that can allocate memory, and the memory can OpenGL functions that return this error are undefined; it is allowable for partial execu in this circumstance.

GL_INVALID_FRAMEBUFFER_OPERATION, 0x0506

Given when doing anything that would attempt to read from or write/render to a fram

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[1], 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

GL_STACK_UNDERFLOW, 0x0504

Given when a stack popping operation cannot be done because the stack is already a

GL_OUT_OF_MEMORY, 0x0505

Given when performing an operation that can allocate memory, and the memory can OpenGL functions that return this error are undefined; it is allowable for partial execu in this circumstance.

GL_INVALID_FRAMEBUFFER_OPERATION, 0x0506

Given when doing anything that would attempt to read from or write/render to a fram

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[1], 0x8031

Part of the ARB_imaging extension.

Those are the most important design points.

Now, a few extras:

Steps in Creating Graphics using Vulkan

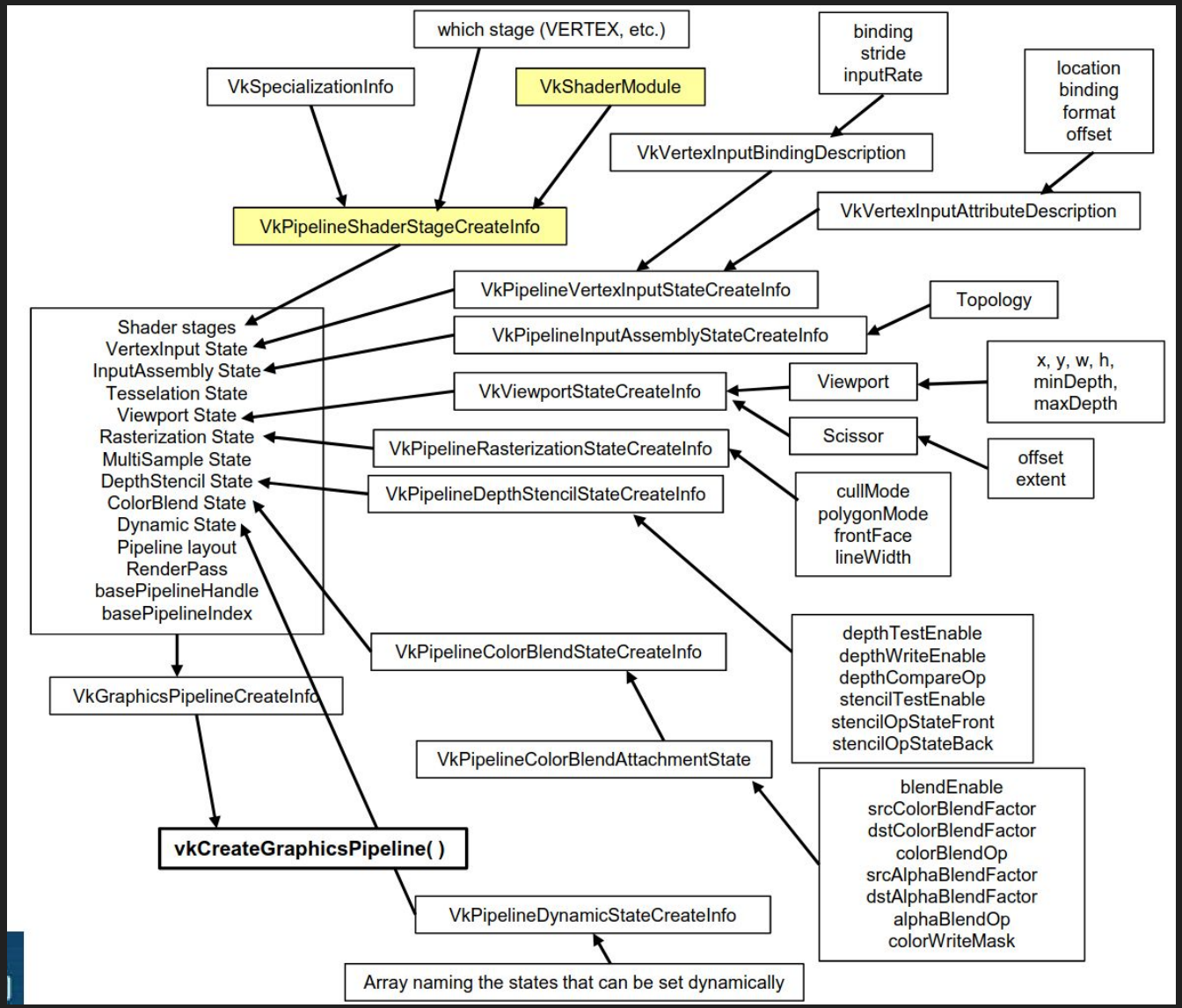
1. Create the Vulkan Instance
2. Setup the Debug Callbacks
3. Create the Surface
4. List the Physical Devices
5. Pick the right Physical Device
6. Create the Logical Device
7. Create the Uniform Variable Buffers
8. Create the Vertex Data Buffers
9. Create the texture sampler
10. Create the texture images
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- …

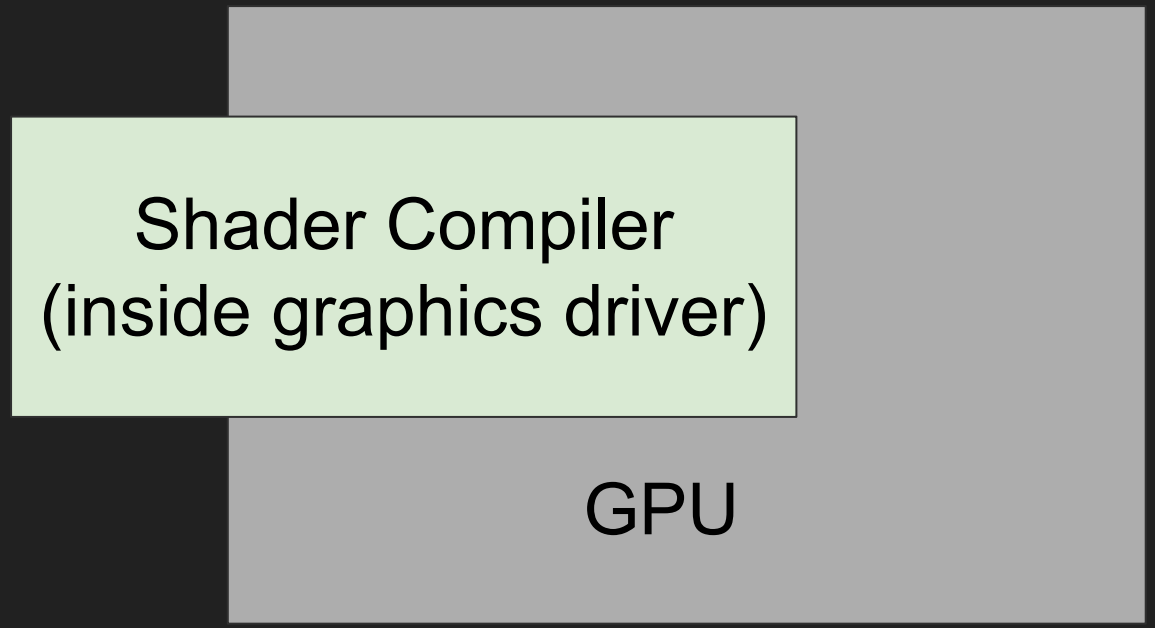
Steps in Creating Graphics using Vulkan

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- …

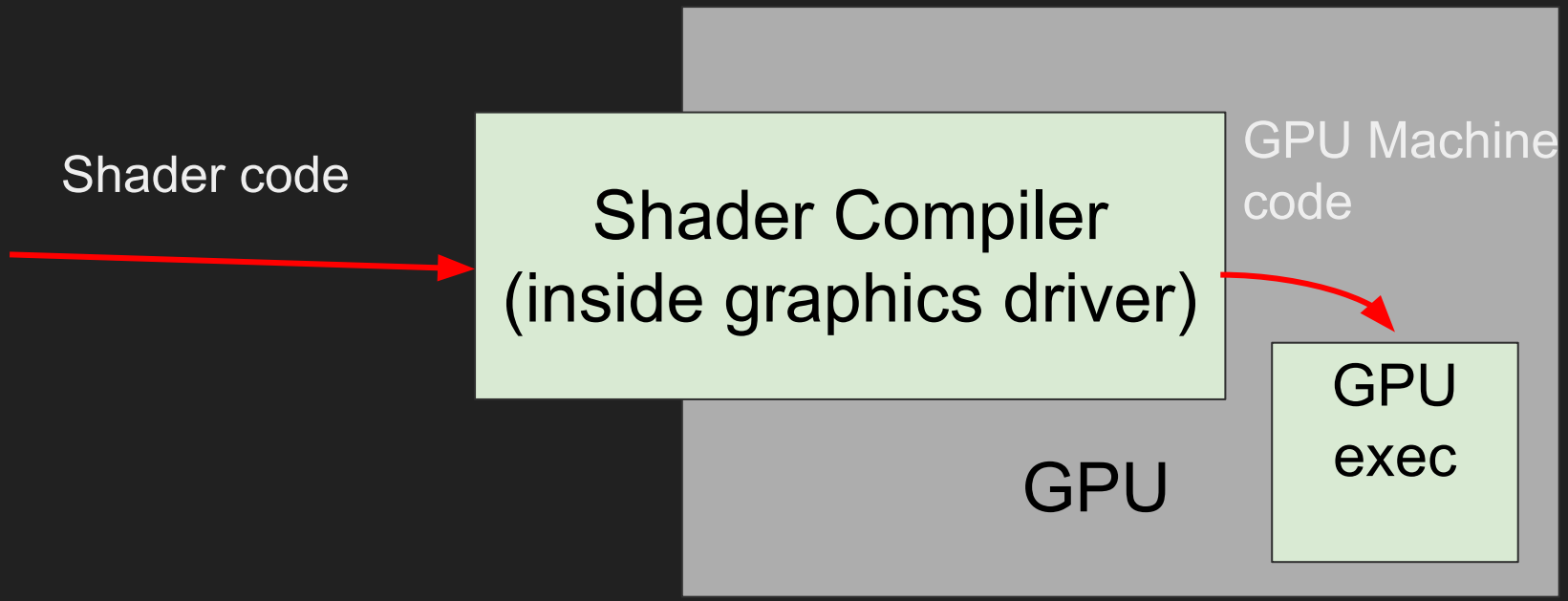
Steps in Creating Graphics using Vulkan

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- …

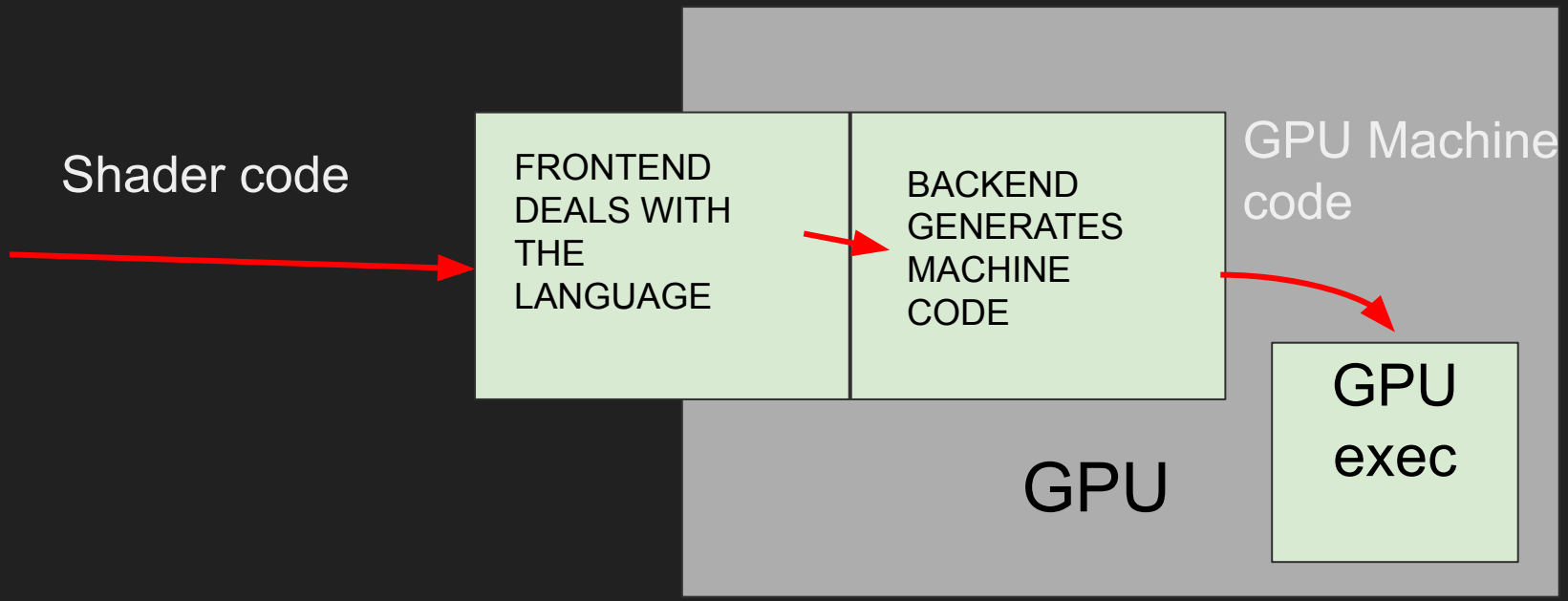

which stage (VERTEX, etc.)
binding
stride
inputRate
location
binding
format
offset
VkSpecializationInfo
VkShaderModule
VkVertexInputBindingDescription
VkVertexInputAttributeDescription
VkPipelineShaderStageCreateInfo
VkPipelineVertexInputStateCreateInfo
Topology
Shader stages
VertexInput State
InputAssembly State
Tesselation State
Viewport State
Rasterization State
MultiSample State
DepthStencil State
ColorBlend State
Dynamic State
Pipeline layout
RenderPass
basePipelineHandle
basePipelineIndex
VkPipelineInputAssemblyStateCreateInfo
x, y, w, h,
minDepth,
maxDepth
Viewport
VkViewportStateCreateInfo
Scissor
offset
extent
VkPipelineRasterizationStateCreateInfo
VkPipelineDepthStencilStateCreateInfo
cullMode
polygonMode
frontFace
lineWidth
depthTestEnable
depthWriteEnable
depthCompareOp
stencilTestEnable
stencilOpStateFront
stencilOpStateBack
VkPipelineColorBlendStateCreateInfo
VkGraphicsPipelineCreateInfo
VkPipelineColorBlendAttachmentState
blendEnable
srcColorBlendFactor
dstColorBlendFactor
colorBlendOp
srcAlphaBlendFactor
dstAlphaBlendFactor
alphaBlendOp
colorWriteMask
vkCreateGraphicsPipeline( )
VkPipelineDynamicStateCreateInfo
Array naming the states that can be set dynamically

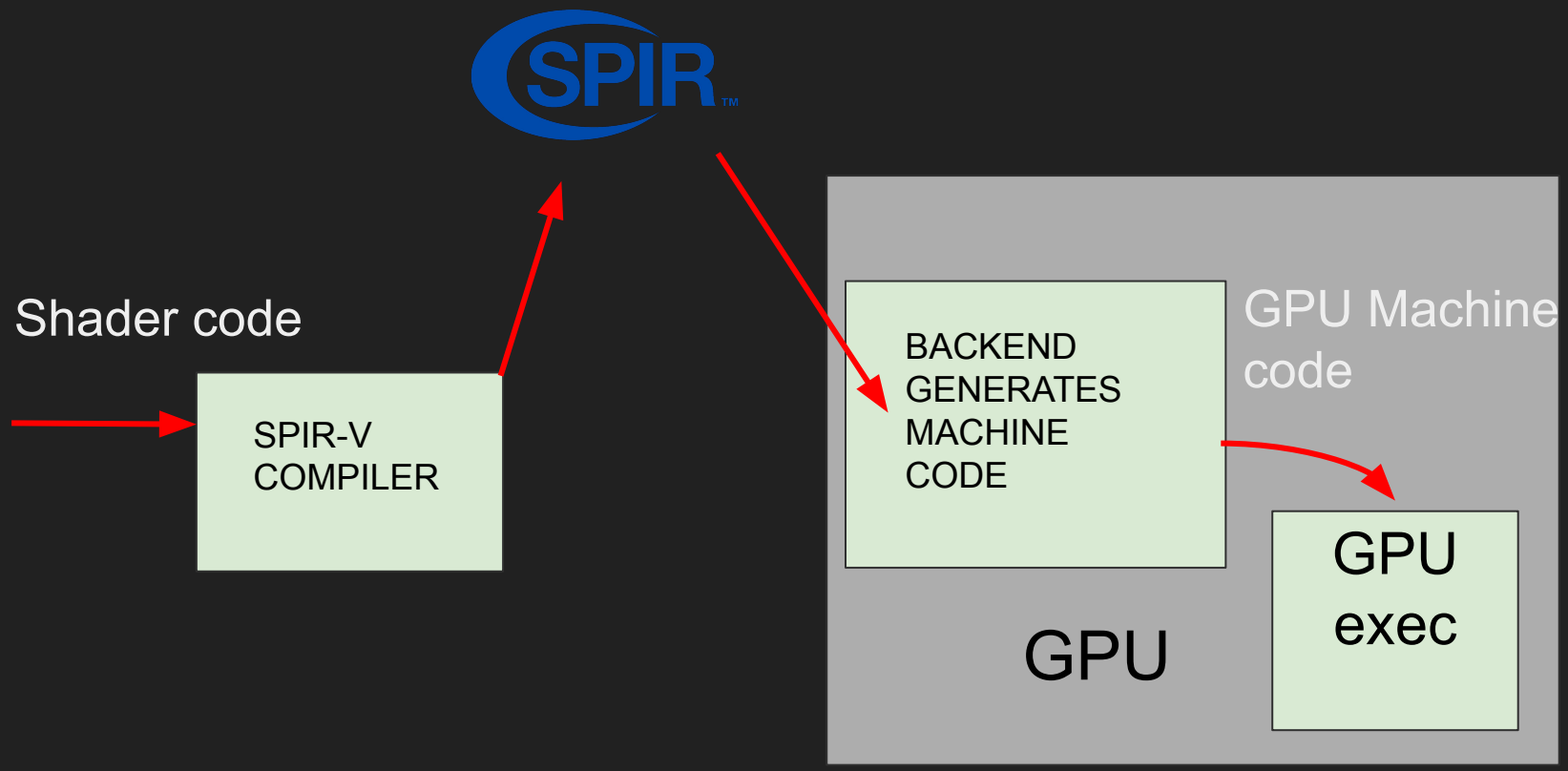

Shader Compiler
(inside graphics driver)
GPU


SPIR™


Shader code
Shader Compiler
(inside graphics driver)
GPU Machine
code
GPU
exec
GPU

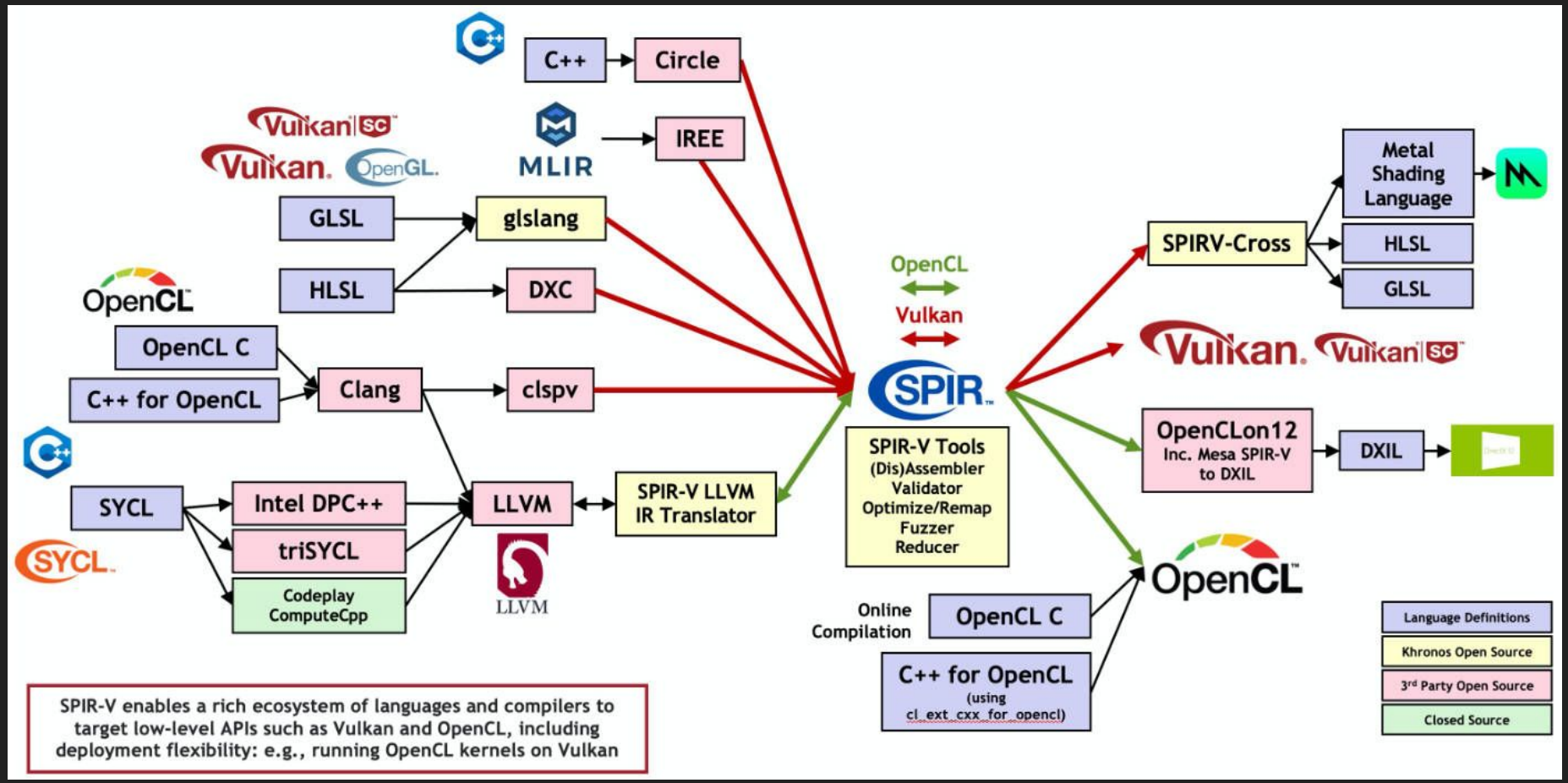

SPIR™


Shader code
FRONTEND DEALS WITH THE LANGUAGE
BACKEND GENERATES MACHINE CODE
GPU Machine code
GPU exec
GPU


SPIR
Shader code
SPIR-V
COMPILER
BACKEND
GENERATES
MACHINE
CODE
GPU Machine
code
GPU
exec
GPU

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.


C++
Circle
MLIR
IREE
GLSL
glslang
HLSL
DXC
OpenCL C
C++ for OpenCL
Clang
clspv
SYCL
Intel DPC++
triSYCL
Codeplay ComputeCpp
LLVM
SPIR-V LLVM IR Translator
OpenCL
Vulkan
SPIR
SPIR-V Tools
(Dis)Assembler
Validator
Optimize/Remap
Fuzzer
Reducer
SPIRV-Cross
Metal Shading Language
HLSL
GLSL
OpenCLon12
Inc. Mesa SPIR-V to DXIL
DXIL
Online Compilation
OpenCL C
C++ for OpenCL
(using cl_ext_cxx_for_opencl)
Language Definitions
Khronos Open Source
3rd Party Open Source
Closed Source
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