Vulkan Tutorial

2016 Khronos Seoul DevU

SAMSUNG Electronics

Hyokuen Lee
Senior Graphics Engineer (hk75.lee@samsung.com)
Minwook Kim
Senior Graphics Engineer (minw83.kim@samsung.com)

Contents

- Introduction
- Development Environment
- Initialization
- Drawing a Triangle
- Drawing a Rectangle
- Rotation and 3D Projection
- Texture Mapping
- Standard Validation Layer

Introduction

Vulkan

- Low CPU overhead, cross-platform 3D graphics and compute API

OpenGL ES
OpenGL
Vulkan

Application
Single thread per context

High-level Driver Abstraction
Context management
Memory allocation
Full GLSL compiler
Error detection
Layered GPU Control

GPU

Application
Memory allocation
Thread management
Multi-threaded generation of command buffers
Multi-queue work submission

Language Front-end Compilers
Initially GLSL

SPIR
SPIR-V pre-compiled shaders

Thin Driver
Explicit GPU Control

Loadable debug and validation layers

GPU

Vulkan Features

Low CPU Overhead

- Low level API reducing CPU overhead

OpenGL ES
Vulkan
0.00 0.20 0.40 0.60 0.80 1.00 1.20
Driver Overhead (normalized CPU load)

GDC 2015 Khronos Vulkan Session

Vulkan Features

Multi-Core Efficiency

- Multi-threaded command submission to queue

Command Buffer

CPU Thread	cmd, cmd, cmd
CPU Thread	cmd, cmd, cmd, cmd
CPU Thread	
CPU Thread	cmd, cmd, cmd

Queue

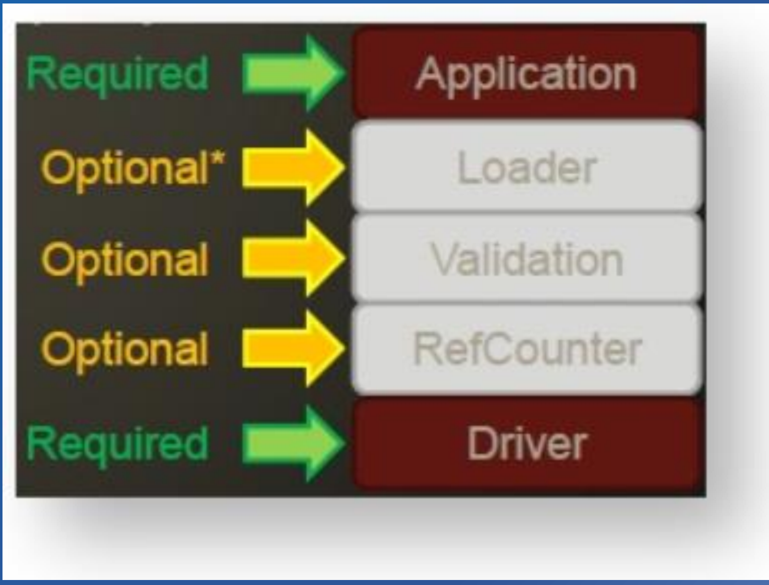
cmd, cmd, cmd, cmd, cmd

Vulkan Features

Layer Structure

- User can enable / add layers

(Vulkan API does not do basic error detection or dependency tracking for low-overhead)



Vulkan Features

SPIR-V

- Standard Portable Intermediate Representation
- Binary shading language
- Use pre-compiled shader (No need to compile shaders at run-time)

Development Environment

- Install Development Tools
- Build Vulkan SDK
- Visual Studio Configuration

Install Development Tools

- **Installation**

  - Vulkan SDK (https://vulkan.lunarg.com/app/download)

  - Cmake (https://cmake.org/download/)

  - Python 3 (https://www.python.org/downloads/)

  - GLM library (http://glm.g-truc.net/0.9.8/index.html)

  - Vulkan Graphic Driver
  : Nvdia (https://developer.nvidia.com/vulkan-driver)
  : AMD (http://www.amd.com/en-us/innovations/software-technologies/technologies-gaming/vulkan)

Build Vulkan SDK

- **Build Vulkan SDK (based on Visual Studio 2015, 64-bit computer)**

  - go to C:\VulkanSDK\1.0.17.0\glslang\build
    : cmake –G "Visual Studio 14 Win64" ..
    : build all Debug/Release x64

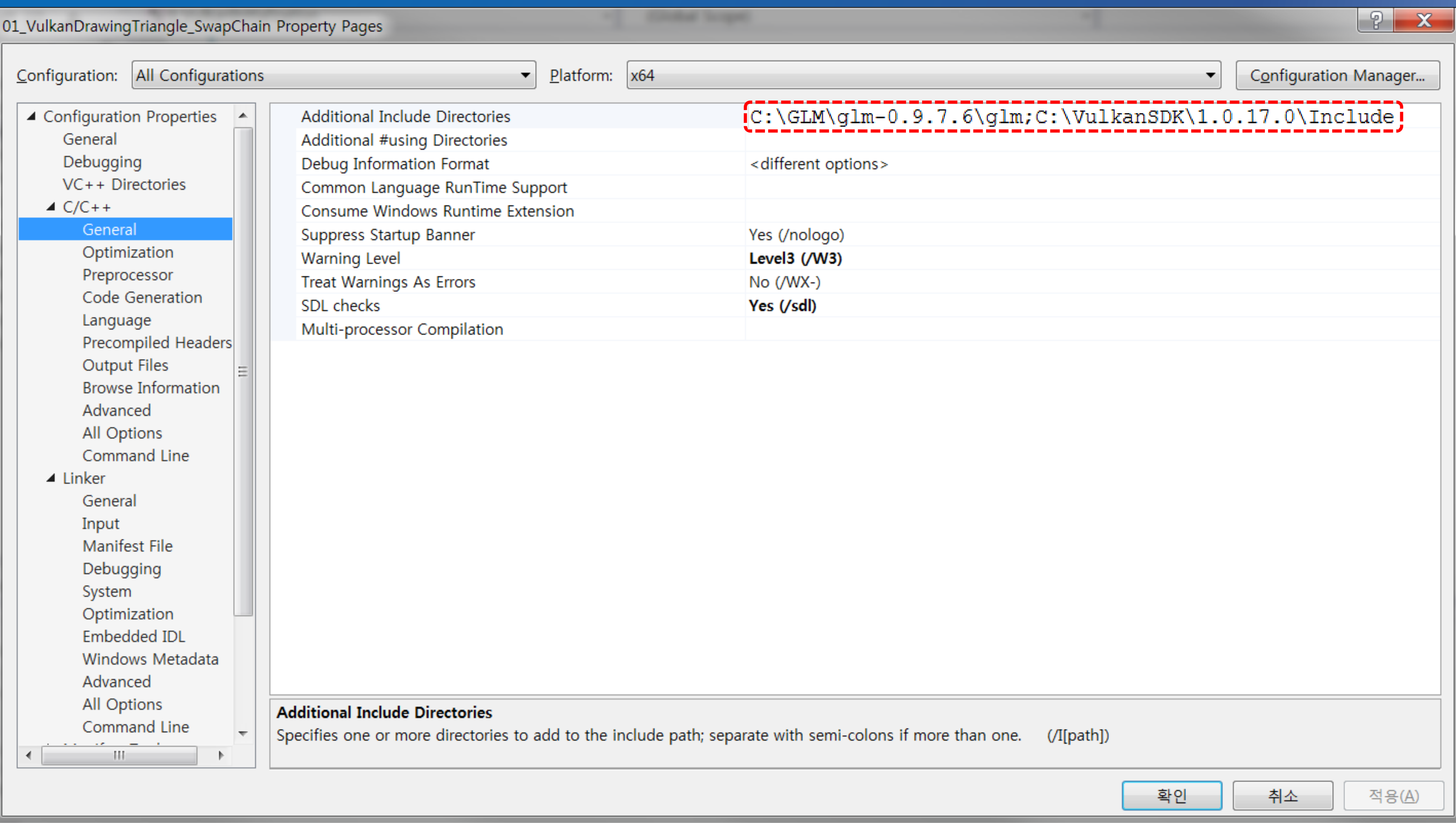
  - go to C:\VulkanSDK\1.0.17.0\spirv-tools\build
    : cmake –G "Visual Studio 14 Win64" ..
    : build all Debug/Release x64

  - go to C:\VulkanSDK\1.0.17.0\Samples\build
    : cmake –G "Visual Studio 14 Win64" ..
    : build all Debug/Release x64

Create the folder "build" in person

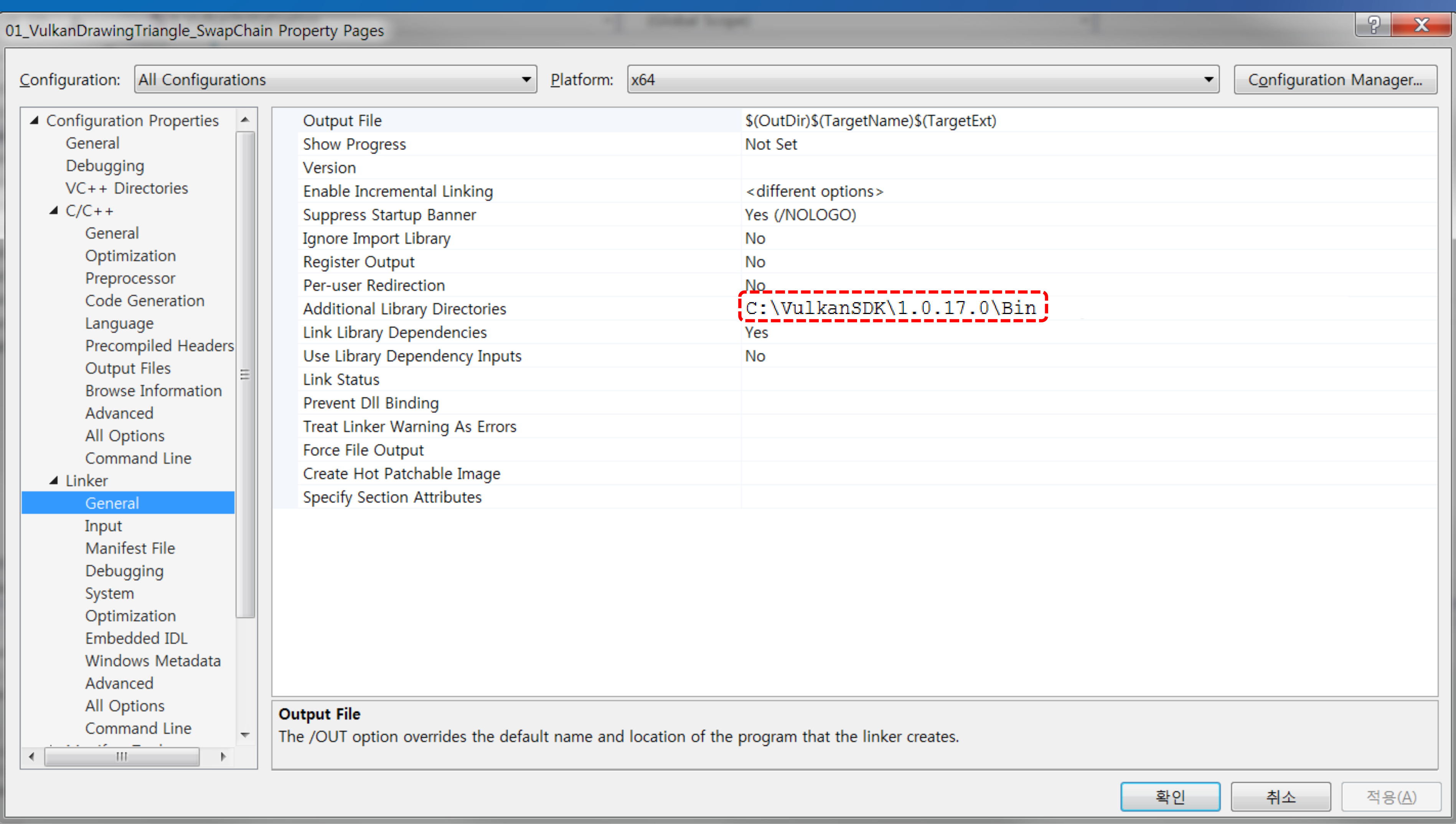
Visual Studio Configuration

- Set the Vulkan / GLM header path



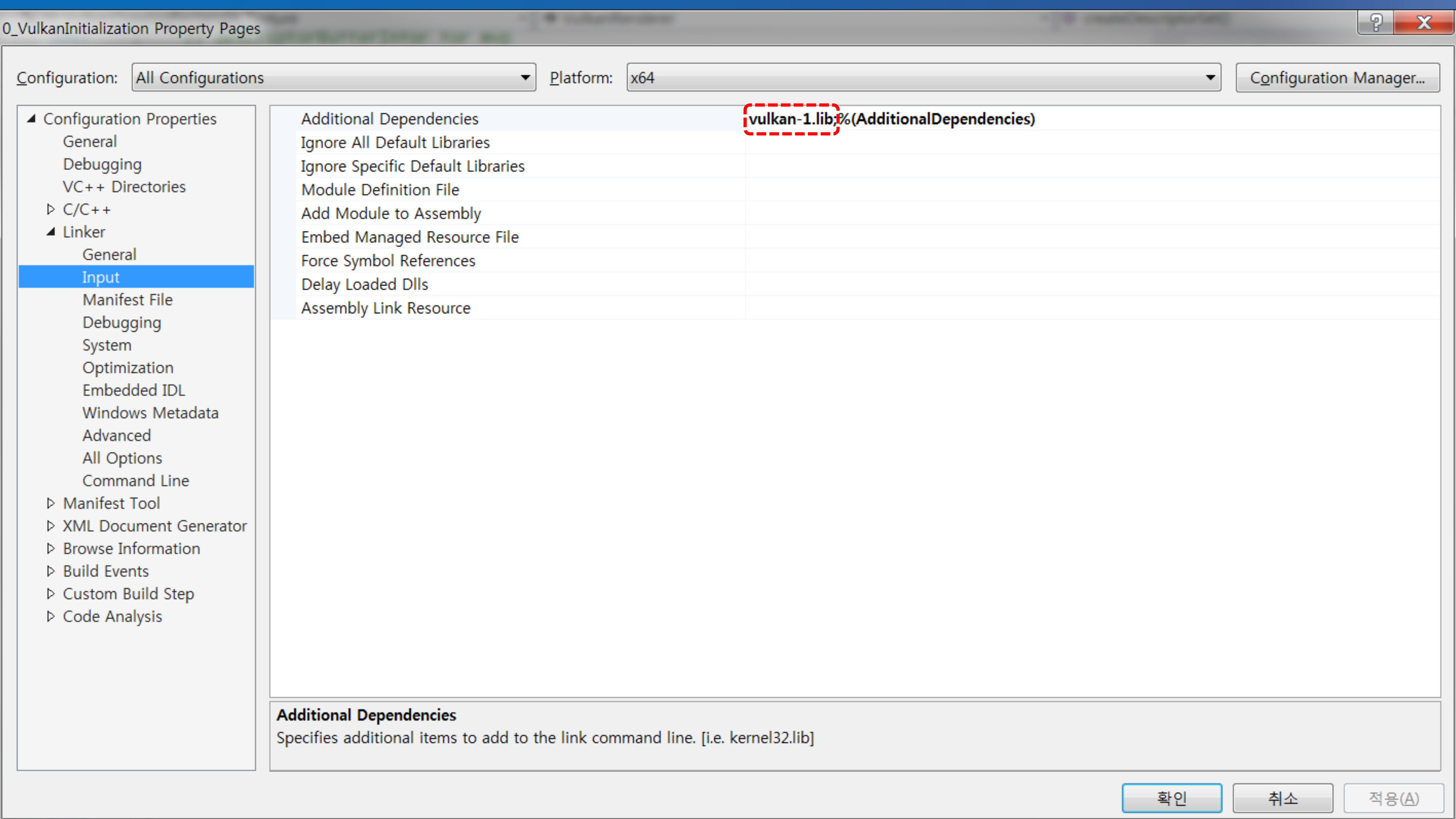
Visual Studio Configuration

- Set the Vulkan library path



Visual Studio Configuration

- Vulkan library



Initialization

- Instance
- Device (Physical Device / Logical Device)
- Queue / Queue Family

Vulkan.

API Naming Convention

Standard Prefixes

VK : Define / Vk : Type structure / vk : Function

p / PFN / pfn : Pointer, function pointer

vkCmd : Commands that will be stored in the command buffer

Extension

1) Type structure / Function

VkSurfaceFormatKHR / vkDestorySurfaceKHR( )

2) Define

VK_KHR_mirror_clamp_to_edge / VK_EXT_debug_marker

Instance

Instance

- Connection between vulkan and the application
- Including simple application information, instance layers and instance extensions

Creating an Instance

1) Enable an instance layer / extension
   - Check instance layer support
   - Check instance extension support
2) Create an instance
   - Set application information
   - Set instance layer and extension information

Instance

Create

```
VulkanRenderer::VulkanRenderer()
{
    createInstance();
    selectPhysicalDevice();
    createLogicalDevice();
}
```

Destroy

```
VulkanRenderer::~VulkanRenderer()
{
    destroyLogicalDevice();
    destroyInstance();
}
```

createInstance()

```
class VulkanRenderer
{
private:
    VkInstance  mInstance    = VK_NULL_HANDLE;
    std::vector<const char*>    mInstanceLayers;
    std::vector<const char*>    mInstanceExtensions;

    void createInstance();
    void destroyInstance();
}
```

```
void VulkanRenderer::createInstance()
{
    // Optional info
    VkApplicationInfo appInfo {};
    appInfo.sType                = VK_STRUCTURE_TYPE_APPLICATION_INFO;
    appInfo.apiVersion           = VK_MAKE_VERSION(1, 0, 0);
    appInfo.applicationVersion   = 0;
    appInfo.pApplicationName     = "Vulkan Tutorial";
//  appInfo.pNext                = ;     // point to extension info

    // Mandatory info
    VkInstanceCreateInfo instance_create_info{};
    instance_create_info.sType                     = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
    instance_create_info.pApplicationInfo          = &appInfo;
    instance_create_info.enabledLayerCount         = 0;
    instance_create_info.ppEnabledLayerNames       = nullptr;
    instance_create_info.enabledExtensionCount     = 0;
    instance_create_info.ppEnabledExtensionNames   = nullptr;
    instance_create_info.pNext                     = nullptr;

    checkError(vkCreateInstance(&instance_create_info, nullptr, &mInstance));
}
```


Vulkan

checkError()

```
void checkError(VkResult result)
{
    if (result < 0) {
        switch (result)
        {
        case VK_ERROR_OUT_OF_HOST_MEMORY:
            std::cout << "VK_ERROR_OUT_OF_HOST_MEMORY" << std::endl;
            break;
        case VK_ERROR_OUT_OF_DEVICE_MEMORY:
            std::cout << "VK_ERROR_OUT_OF_DEVICE_MEMORY" << std::endl;
            break;
        case VK_ERROR_INITIALIZATION_FAILED:
            std::cout << "VK_ERROR_INITIALIZATION_FAILED" << std::endl;
            break;
        case VK_ERROR_DEVICE_LOST:
            std::cout << "VK_ERROR_DEVICE_LOST" << std::endl;
            break;

        // ...

        default:
            break;
        }
        assert(0 && "Vulkan runtime error.");
    }
}
```

destroyInstance()

```
void VulkanRenderer::destroyInstance()
{
    vkDestroyInstance(mInstance, nullptr);
    mInstance = VK_NULL_HANDLE;
}
```

```
class VulkanRenderer
{
private:
    VkInstance  mInstance   = VK_NULL_HANDLE;
    std::vector<const char*>    mInstanceLayers;
    std::vector<const char*>    mInstanceExtensions;

    void createInstance();
    void destroyInstance();
}
```

Device

Physical Device

- Select physical devices through the instance
- "Physical device" means GPU in the system
- Multiple GPUs can be used in Vulkan

Logical Device

- Create logical devices through physical devices
- Logical connection between a Vulkan program and GPU
  (Main handle when using the Vulkan API)
- Multiple logical devices can be created through physical device

Queue / Queue Family

Queue

- Most operations (drawing, texturing, memory transfer, etc.) are encapsulated in a command buffer, which is submitted to a queue

Queue Family

- Different queue families for different combinations of queue capabilities
- Each queue family allows specific types of operation

  e.g.) Queue family 0 for drawing commands and compute commands,

  queue family 1 for memory transfer

Physical Device

Selecting Physical Device

- Pick a GPU
- Check a support of queue family for graphics commands

1) Enumerate physical devices (GPUs) available in the system
2) Check the graphics queue family support
   - Check VK_QUEUE_GRAPHICS_BIT flag
3) Check GPU properties and features

Physical Device

Create

```
VulkanRenderer::VulkanRenderer()
{
    createInstance();
    selectPhysicalDevice();
    createLogicalDevice();
}
```

Destroy

```
VulkanRenderer::~VulkanRenderer()
{
    destroyLogicalDevice();
    destroyInstance();
}
```

※ Destroy operation is not needed for physical device selection

selectPhysicalDevice()

```
class VulkanRenderer
{
private:
    VkPhysicalDevice           mGpu          = VK_NULL_HANDLE;
    VkPhysicalDeviceProperties mGpuProperties = {};
    VkPhysicalDeviceFeatures   mGpuFeatures   = {};

    void selectPhysicalDevice();
    bool isDeviceSuitable(VkPhysicalDevice gpu);
}
```

```
void VulkanRenderer::selectPhysicalDevice()
{
    // enumerate physical devices (GPUs)
    uint32_t gpuCount = 0;
    vkEnumeratePhysicalDevices(mInstance, &gpuCount, nullptr);
    std::vector<VkPhysicalDevice> gpuList(gpuCount);
    vkEnumeratePhysicalDevices(mInstance, &gpuCount, gpuList.data());

    for (const auto& device : gpuList) {
        if ( isDeviceSuitable(device) ) {
            mGpu = device;
            break;
        }
    }

    if ( mGpu == VK_NULL_HANDLE ) {
        std::cout << "Failed to find a suitable GPU!" << std::endl;
        std::exit(-1);
    }

    vkGetPhysicalDeviceProperties(mGpu, &mGpuProperties);
    vkGetPhysicalDeviceFeatures(mGpu, &mGpuFeatures);
}
```

Check the graphics queue family support

Vulkan

isDeviceSuitable()

```
bool VulkanRenderer::isDeviceSuitable(VkPhysicalDevice gpu)
{
    // 1. queue family supported
    VulkanQueueFamily queueFamily;
    queueFamily.findQueueFamilies(gpu);

    // 2. device extension supported

    // 3. swapchain supported (TODO)

    return queueFamily.isComplete();
}
```

Check the graphics queue family support

```
class VulkanRenderer
{
private:
    VkPhysicalDevice           mGpu           = VK_NULL_HANDLE;
    VkPhysicalDeviceProperties mGpuProperties  = {};
    VkPhysicalDeviceFeatures   mGpuFeatures    = {};

    void selectPhysicalDevice();
    bool isDeviceSuitable(VkPhysicalDevice gpu);
}
```


Vulkan.

findQueueFamilies() #1

See also : Present Queue Family extension version

```
void VulkanQueueFamily::findQueueFamilies(VkPhysicalDevice gpu)
{
    // enumerate Queue Family
    uint32_t familyCount = 0;
    vkGetPhysicalDeviceQueueFamilyProperties(gpu, &familyCount, nullptr);
    std::vector<VkQueueFamilyProperties> properties(familyCount);
    vkGetPhysicalDeviceQueueFamilyProperties(gpu, &familyCount, properties.data());

    for (uint32_t i = 0; i < familyCount; ++i) {
        // find graphics queue family index
        if (properties[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) {
            mIdxGraphicsFamily  = i;
        }

        if ( isComplete() ) {
            break;
        }
    }
}
```

Check the VK_QUEUE_GRAPHICS_BIT flag

```
class VulkanQueueFamily
{
public:
    uint32_t    mIdxGraphicsFamily  = -1;

    void findQueueFamilies(VkPhysicalDevice gpu);
    bool isComplete() {
        return ( mIdxGraphicsFamily >= 0 );
    }

};
```

Logical Device

Creating a Logical Device

- Logical connection between a Vulkan program and GPU
  (Main handle when using the Vulkan API)

- Creating a logical device
  1) Specify queues to use (create queue from queue family)
  2) Specify device extensions to use
  3) Specify device features to use
  4) Create a logical device

Logical Device

Create

```
VulkanRenderer::VulkanRenderer()
{
    createInstance();
    selectPhysicalDevice();
    createLogicalDevice();
}
```

Destroy

```
VulkanRenderer::~VulkanRenderer()
{
    destroyLogicalDevice();
    destroyInstance();
}
```

createLogicalDevice()

```
void VulkanRenderer::createLogicalDevice()
{
    VulkanQueueFamily queueFamily;
    queueFamily.findQueueFamilies(mGpu);

    std::vector<VkDeviceQueueCreateInfo> queueCreateInfoList;
    std::set<uint32_t> uniqueQueueFamiles = { queueFamily.mIdxGraphicsFamily };

    for ( int i : uniqueQueueFamiles ) {
        float priorities[]{ 1.0f };     // range : [0.0, 1.0]
        VkDeviceQueueCreateInfo queueCreateInfo{};
        queueCreateInfo.sType              = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
        queueCreateInfo.queueFamilyIndex   = i;
        queueCreateInfo.queueCount         = 1;
        queueCreateInfo.pQueuePriorities   = priorities;
        queueCreateInfoList.push_back( queueCreateInfo );
    }

    //VkPhysicalDeviceFeatures deviceFeatures = {};

    VkDeviceCreateInfo deviceCreateInfo{};
    deviceCreateInfo.sType                     = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
    deviceCreateInfo.queueCreateInfoCount      = (uint32_t)queueCreateInfoList.size();
    deviceCreateInfo.pQueueCreateInfos         = queueCreateInfoList.data();
    deviceCreateInfo.enabledLayerCount         = 0;
    deviceCreateInfo.ppEnabledLayerNames       = nullptr;
    deviceCreateInfo.enabledExtensionCount     = 0;
    deviceCreateInfo.ppEnabledExtensionNames   = nullptr;
    deviceCreateInfo.pEnabledFeatures          = &mGpuFeatures;

    // create logical device and queues
    checkError(vkCreateDevice(mGpu, &deviceCreateInfo, nullptr, &mDevice));

    // retrieve queue handle
    vkGetDeviceQueue( mDevice, queueFamily.mIdxGraphicsFamily, 0, &mGraphicsQueue );
}
```

```
class VulkanRenderer
{
private:
    VkDevice     mDevice = VK_NULL_HANDLE;
    VkPhysicalDeviceFeatures mGpuFeatures = {};

    void createLogicalDevice();
    void destroyLogicalDevice();
}
```

- Queues to use
- Device extensions to use
- Device features to use

Get the handle of the queue created during logical device creation

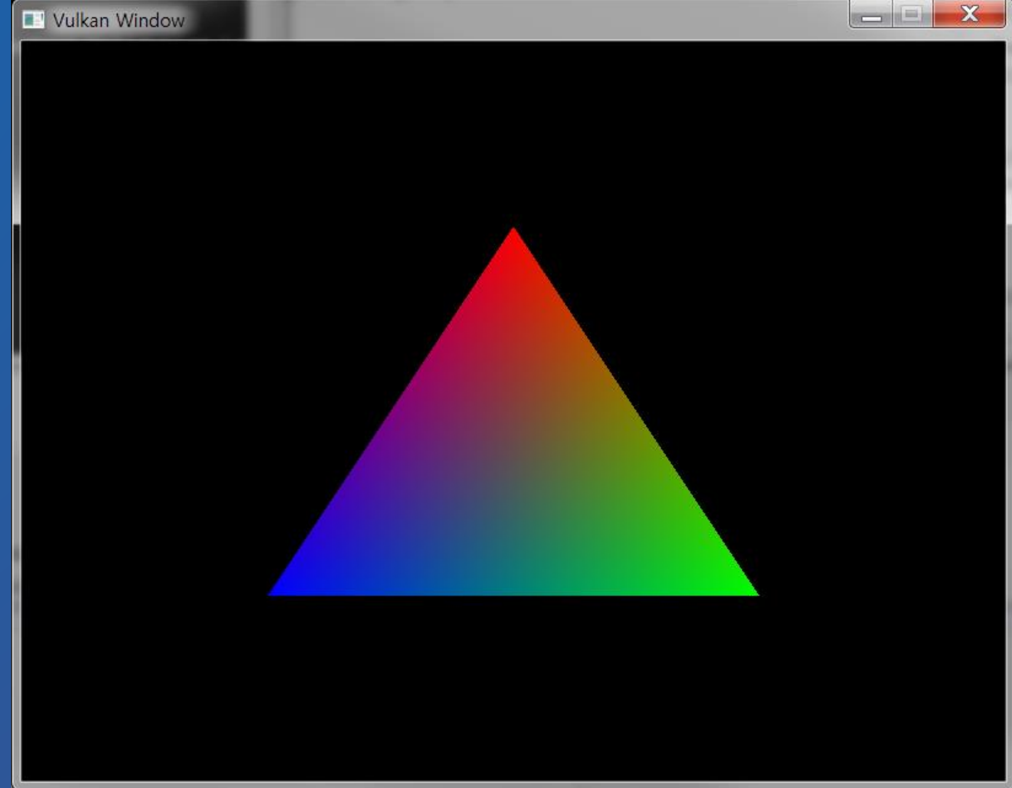
destroyLogicalDevice()

```
class VulkanRenderer
{
private:
    VkDevice     mDevice = VK_NULL_HANDLE;
    VkPhysicalDeviceFeatures mGpuFeatures = {};

    void createLogicalDevice();
    void destroyLogicalDevice();
}
```

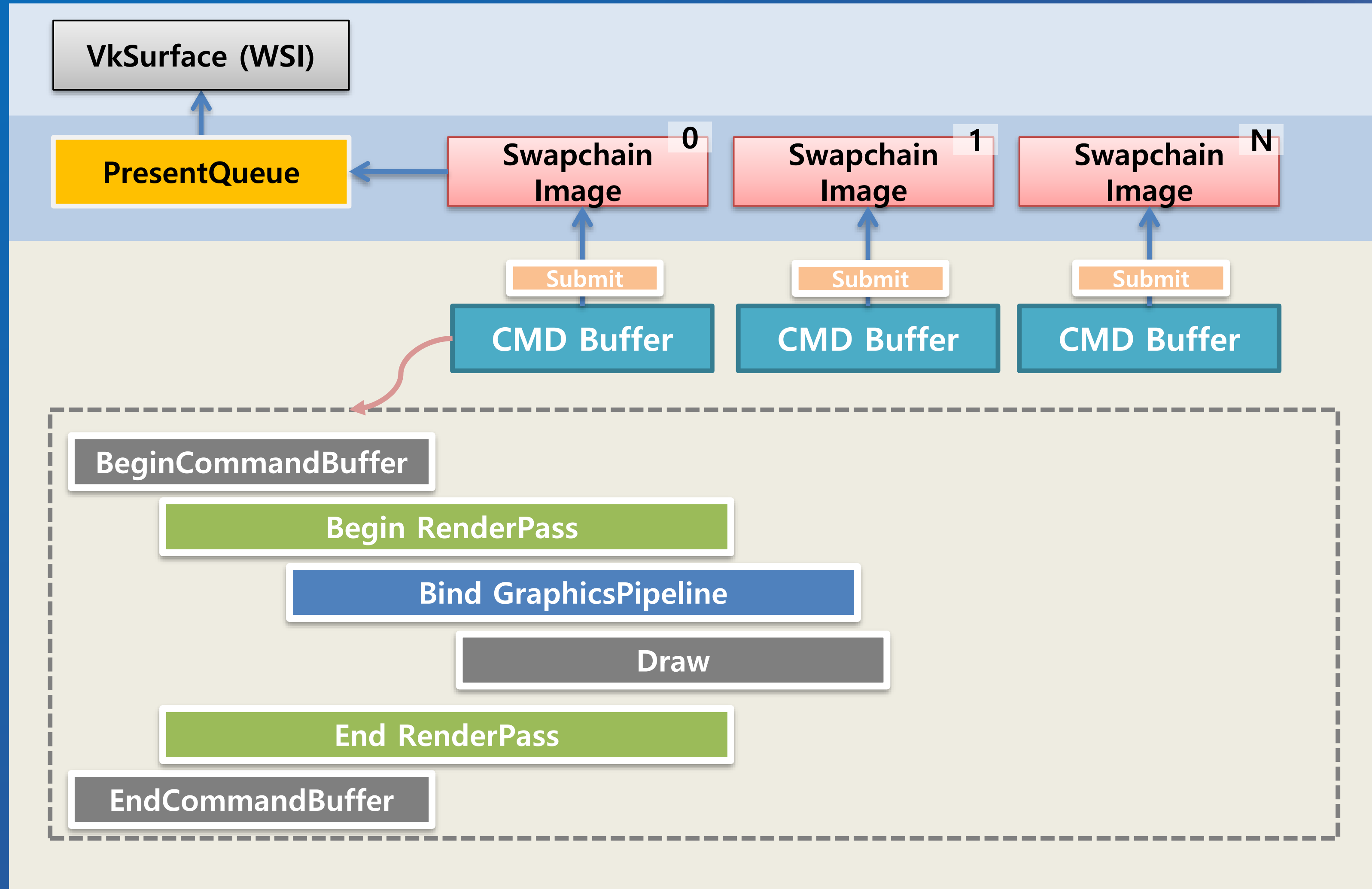
```
void VulkanRenderer::destroyLogicalDevice()
{
    vkDestroyDevice(mDevice, nullptr);  // clean up the logical device
    mDevice = VK_NULL_HANDLE;
}
```

Drawing a Triangle

- Window System / Surface
- Present Queue
- Swapchain / Framebuffer
- Command Buffer
- Render Pass
- Graphics Pipeline
- Shader (SPIR-V)
- Swapchain Recreation



Drawing a Triangle



Vulkan.

Window System / Surface

Window Surface

- Vulkan is platform agnostic API, each platform can not interface directly with the window system
- To present rendered screen, we need to use the WSI (Window System Integration) extension
- Also need to use rendering target surface fit with each platform

Device Extension for Presentation

- VK_KHR_surface
  : Vulkan WSI (Window System Integration)
- VK_KHR_win32_surface
  : Extension for using Win32 system platform Surface Window

※ VK_KHR_xcb_surface(Linux) / VK_KHR_android_surface(android)

Window System / Surface

Create

Destroy

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();
}
```

```
VulkanRenderer::~VulkanRenderer()
{
    destroyImageViews();
    destroySwapchain();

    destroyLogicalDevice();
    destroySurface();
    destroyInstance();
}
```

```
void VulkanRenderer::enableLayersAndExtensions()
{
    // extensions for grahpics
    mInstanceExtensions.push_back("VK_KHR_surface");
    mInstanceExtensions.push_back("VK_KHR_win32_surface");
    //mInstanceExtensions.push_back("VK_KHR_android_surface");    //Android Surface
    mDeviceExtensions.push_back( "VK_KHR_swapchain" );       // VK_KHR_swapchain
}
```

Creating the Surface

Creating Surface

- We need to create a VkSurfaceKHR surface
- Each platform needs a different function call to create a VkSurfaceKHR:

  Win32 : vkCreateWin32SurfaceKHR( )

  Android : vkCreateAndroidSurfaceKHR( )

  Linux : vkCreateXcbSurfaceKHR( )

```
static VkSurfaceKHR surface = VK_NULL_HANDLE;

if (surface == VK_NULL_HANDLE) {
    VkWin32SurfaceCreateInfoKHR create_info{};
    create_info.sType             = VK_STRUCTURE_TYPE_WIN32_SURFACE_CREATE_INFO_KHR;
    create_info.hinstance         = mInstance;
    create_info.hwnd              = mHwnd;

    checkError( vkCreateWin32SurfaceKHR(vulkanInstance, &create_info, nullptr, &surface) );
}
```

Creating the Surface

Create

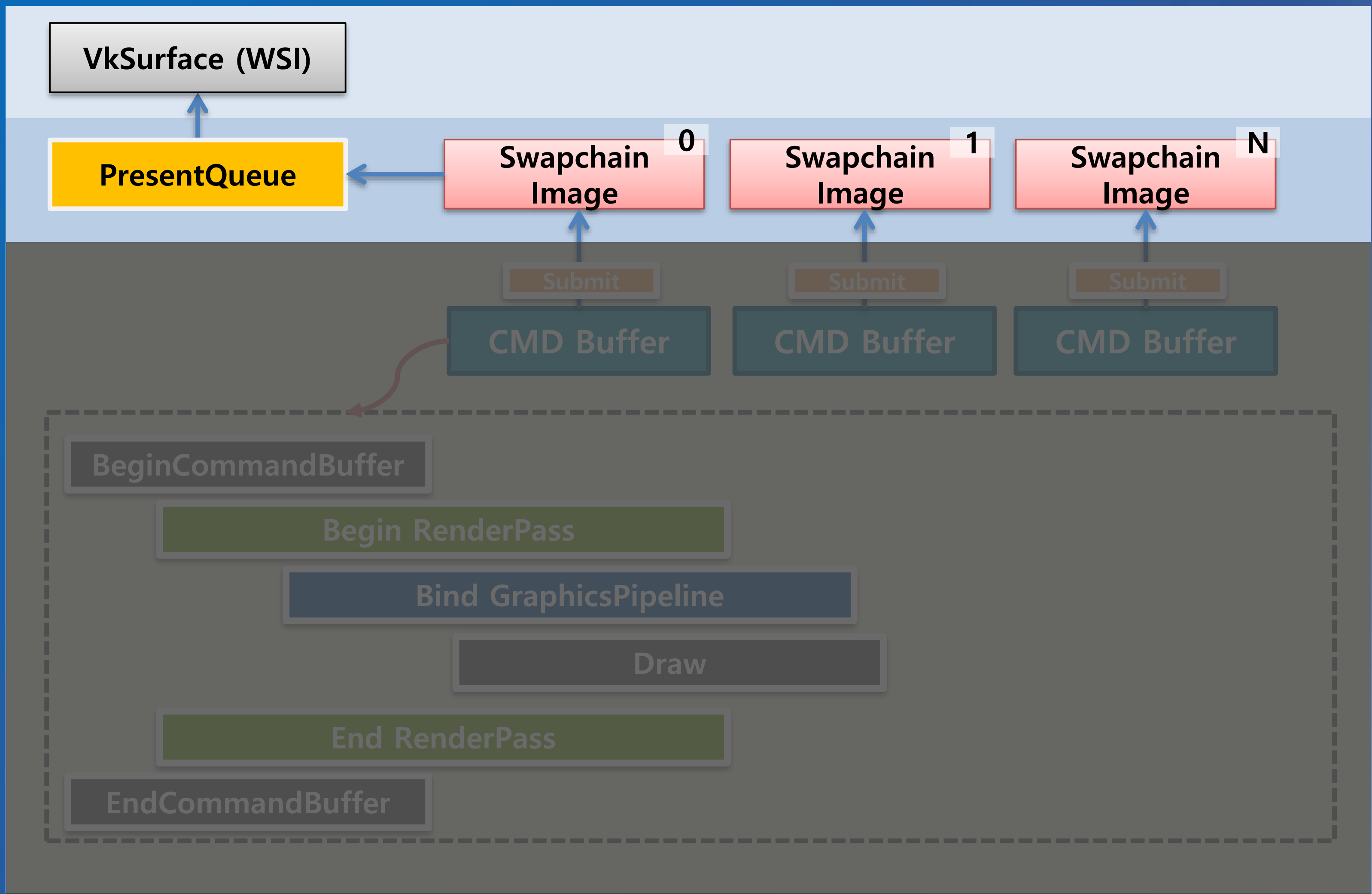
```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();
}
```

Destroy

```
VulkanRenderer::~VulkanRenderer()
{
    destroyImageViews();
    destroySwapchain();

    destroyLogicalDevice();
    destroySurface();
    destroyInstance();
}
```

```
void VulkanRenderer::destroySurface()
{
    vkDestroySurfaceKHR(mInstance, mSurface, nullptr);
}
```

Present Queue



Vulkan.

Present Queue

Present Queue

- Vulkan API uses a 'Present queue' to present a rendered screen to a Surface
- To present a rendered image to surface, we should submit to present queue

Present Queue Family

- Present queue families may or may not be the same as graphics queue families
- Check for present queue family using

vkGetPhysicalDeviceSurfaceSupportKHR( )

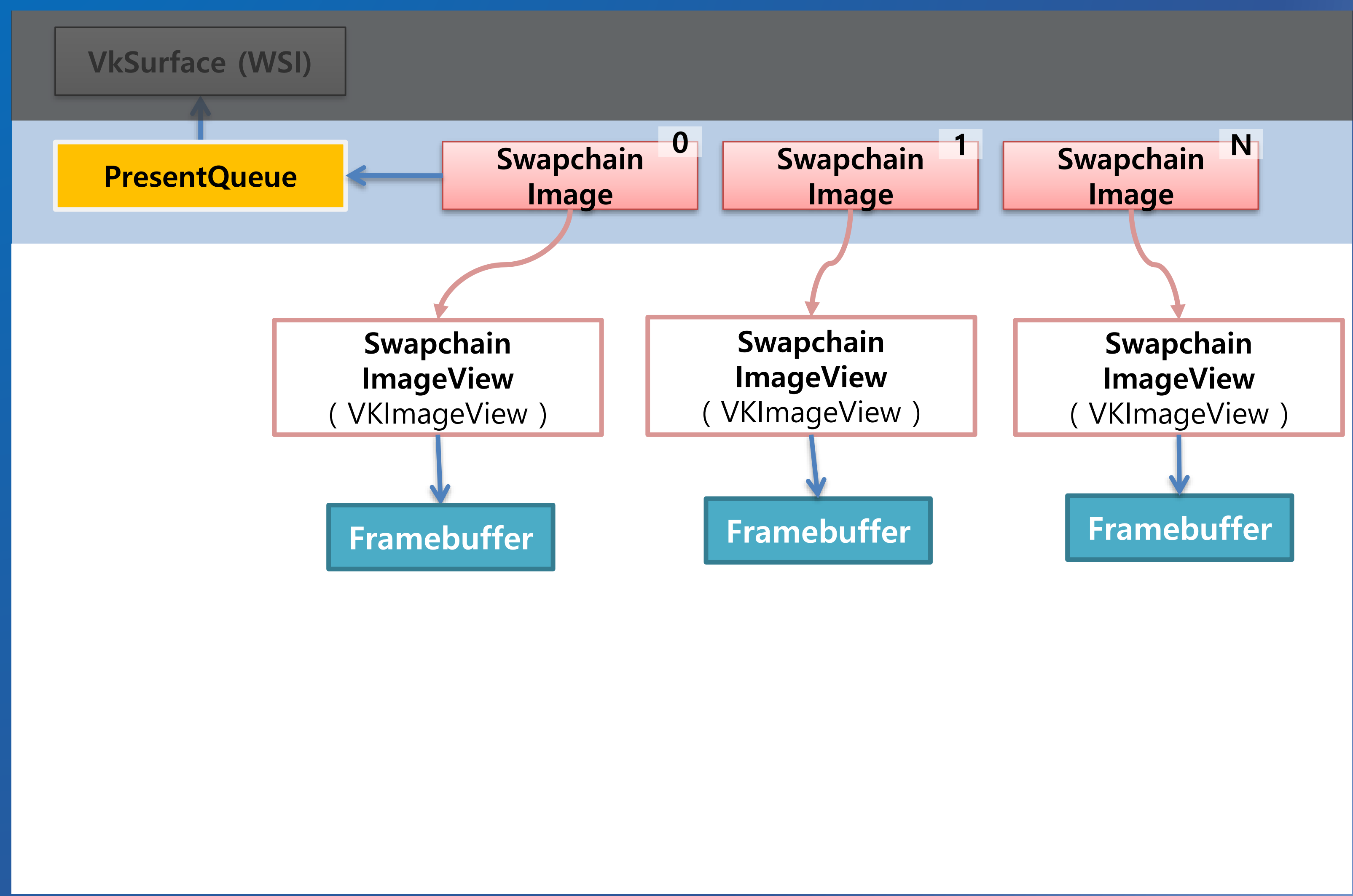
findQueueFamilies() #2

See also : Graphics Queue Family version

```
void VulkanQueueFamily::findQueueFamilies(VkPhysicalDevice gpu, VkSurfaceKHR surface)
{
    // enumerate Queue Family
    uint32_t familyCount = 0;
    vkGetPhysicalDeviceQueueFamilyProperties(gpu, &familyCount, nullptr);
    std::vector<VkQueueFamilyProperties> properties(familyCount);
    vkGetPhysicalDeviceQueueFamilyProperties(gpu, &familyCount, properties.data());

    for (uint32_t i = 0; i < familyCount; ++i) {
        // find graphics queue family index
        if (properties[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) {
            mIdxGraphicsFamily  = i;
        }

        // find present queue family index
        VkBool32 presentSupport = false;
        vkGetPhysicalDeviceSurfaceSupportKHR( gpu, i, surface, &presentSupport );
        if ( properties[i].queueCount > 0 && presentSupport ) {
            mIdxPresentFamily = i;
        }

        if ( isComplete() ) {
            break;
        }
    }
}
```

```
class VulkanQueueFamily
{
public:

    uint32_t    mIdxGraphicsFamily  = -1;
    uint32_t    mIdxPresentFamily   = -1;

    void findQueueFamilies(VkPhysicalDevice gpu, VkSurfaceKHR surface);
    bool isComplete() {
        return ( mIdxGraphicsFamily >= 0  && mIdxPresentFamily >= 0 );
    }

};
```

Swapchain



Vulkan.

Swapchain

Swapchain

- Collection of images that can be presented to the presentation engine

- Synchronize rendered image with the refresh rate of the screen

- Render to the image running drawing operation in graphics queue, and submit it to present queue

Swapchain

Swapchain Image

- Image resource obtained from the swapchain

Swapchain Image View

- Additional information for swapchain

e.g.) RGBA component, view type(2D/3D), surface format, mipmap, image array

Querying for Swapchain Support

Querying for Swapchain Support

- 3 additional information are needed to create swapchain

```
void SwapchainInfo::querySwapchainSupport(VkPhysicalDevice gpu, VkSurfaceKHR surface)
{
    // 1. surface capabilities
    vkGetPhysicalDeviceSurfaceCapabilitiesKHR( gpu, surface, &mCapabilities );

    // 2. surface format
    {
        uint32_t formatCount = 0;
        vkGetPhysicalDeviceSurfaceFormatsKHR( gpu, surface, &formatCount, nullptr );
        if ( formatCount == 0 ) {
            assert( 0 && "Surface formats missing." );
            std::exit(-1);
        }
        mSurfaceFormats.resize(formatCount);
        vkGetPhysicalDeviceSurfaceFormatsKHR( gpu, surface, &formatCount, mSurfaceFormats.data() );
    }

    // 3. presentation mode
    {
        uint32_t presentModeCount = 0;
        checkError( vkGetPhysicalDeviceSurfacePresentModesKHR( gpu, surface, &presentModeCount, nullptr ));
        mPresentModes.resize(presentModeCount);
        checkError( vkGetPhysicalDeviceSurfacePresentModesKHR( gpu, surface, &presentModeCount, mPresentModes.data() ));
    }
}
```

1. Surface capabilities

2. Surface format

3. Presentation Mode

Choosing Swapchain Support 1/3

Surface Capabilities (VkSurfaceCapabilitiesKHR)

- Use extent items in capabilities
- Example of surface capability

```
class SwapchainInfo
{
public:
    VkSurfaceCapabilitiesKHR            mCapabilities;
    std::vector<VkSurfaceFormatKHR>     mSurfaceFormats;
    std::vector<VkPresentModeKHR>       mPresentModes;
};
```

(mCapabilities).minImageCount	1
(mCapabilities).maxImageCount	16
(mCapabilities).currentExtent	{width=800 height=600 }
(mCapabilities).minImageExtent	{width=1 height=1 }
(mCapabilities).maxImageExtent	{width=800 height=600 }
(mCapabilities).maxImageArrayLayers	16
(mCapabilities).supportedTransforms	1
(mCapabilities).currentTransform	VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR (1)

```
actualExtent.width = MAX(mCapabilities.minImageExtent.width,
    MIN(mCapabilities.maxImageExtent.width, actualExtent.width));

actualExtent.height = MAX(mCapabilities.minImageExtent.height,
    MIN(mCapabilities.maxImageExtent.height, actualExtent.height));
```

Choosing Swapchain Support 2/3

Surface Format (VkSurfaceFormatKHR)

1) format (VkFormat)

- VK_FORMAT_B8G8R8A8_UNORM

2) colorSpace (VkColorSpaceKHR)

- VK_COLOR_SPACE_SRGB_NONLINEAR_KHR

```
class SwapchainInfo
{
public:
    VkSurfaceCapabilitiesKHR              mCapabilities;
    std::vector<VkSurfaceFormatKHR>       mSurfaceFormats;
    std::vector<VkPresentModeKHR>         mPresentModes;
};
```

```
VkSurfaceFormatKHR SwapchainInfo::chooseSwapchainFormat()
{
    for (const auto& i : mSurfaceFormats) {
        if (i.format == VK_FORMAT_B8G8R8A8_UNORM && i.colorSpace == VK_COLOR_SPACE_SRGB_NONLINEAR_KHR) {
            return i;
        }
    }

    return mSurfaceFormats[0];

}
```


Vulkan

Choosing Swapchain Support 3/3

Presentation Mode

- Setting timing to send present queue

1) VK_PRESENT_MODE_IMMEDIATE_KHR

2) VK_PRESENT_MODE_FIFO_KHR (wait when queue full)

3) VK_PRESENT_MODE_FIFO_RELAXED_KHR (no wait when queue empty)

4) VK_PRESENT_MODE_MAILBOX_KHR (no wait when queue full)

```
VkPresentModeKHR SwapchainInfo::chooseSwapchainPresentMode()
{
    for (const auto& i : mPresentModes) {
        if (i == VK_PRESENT_MODE_FIFO_KHR      ) {
            return i;
        }
    }
    return VK_PRESENT_MODE_FIFO_KHR;
}
```

```
class SwapchainInfo
{
public:
    VkSurfaceCapabilitiesKHR                mCapabilities;
    std::vector<VkSurfaceFormatKHR>         mSurfaceFormats;
    std::vector<VkPresentModeKHR>           mPresentModes;
};
```

Swapchain

Create

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();
}
```

Destroy

```
VulkanRenderer::~VulkanRenderer()
{
    destroyImageViews();
    destroySwapchain();

    destroyLogicalDevice();
    destroySurface();
    destroyInstance();
}
```

createSwapchain() #1

```
class VulkanRenderer
{
private:
    VkSurfaceCapabilitiesKHR        mSurfaceCapabilitiesKHR = {};
    VkSurfaceFormatKHR              mSurfaceFormatKHR = {};
    VkPresentModeKHR                mPresentModeKHR = {};
    VkExtent2D                      mSwapchainExtent;

    void createSwapchain();
    void destroySwapchain();
}

class SwapchainInfo
{
public:
    VkSurfaceCapabilitiesKHR               mCapabilities;
    std::vector<VkSurfaceFormatKHR>        mSurfaceFormats;
    std::vector<VkPresentModeKHR>          mPresentModes;

    VkSurfaceFormatKHR   chooseSwapchainFormat();
    VkPresentModeKHR     chooseSwapchainPresentMode();
    VkExtent2D           chooseSwapchainExtent();
};
```

```
void VulkanRenderer::createSwapchain()
{
    SwapchainInfo swapchainInfo;
    swapchainInfo.querySwapchainSupport(mGpu, mSurface);

    mSurfaceCapabilitiesKHR = swapchainInfo.mCapabilities;
    mSurfaceFormatKHR = swapchainInfo.chooseSwapchainFormat();
    mPresentModeKHR = swapchainInfo.chooseSwapchainPresentMode();
    mSwapchainExtent = swapchainInfo.chooseSwapchainExtent();

    mSwapchainImageCount = mSurfaceCapabilitiesKHR.minImageCount + 1;
    if (mSurfaceCapabilitiesKHR.maxImageCount > 0 &&
            mSwapchainImageCount > mSurfaceCapabilitiesKHR.maxImageCount) {
        mSwapchainImageCount = mSurfaceCapabilitiesKHR.maxImageCount;
    }
```

createSwapchain() #2

```
class VulkanRenderer
{
private:
    VkSwapchainKHR      mSwapchain;

    void createSwapchain();
    void destroySwapchain();
};
```

```
// create swapchain
VulkanQueueFamily queueFamily;
queueFamily.findQueueFamilies(mGpu, mSurface);
uint32_t queueFamilyIndices[] = { (uint32_t)queueFamily.mIdxGraphicsFamily, (uint32_t)queueFamily.mIdxPr

VkSwapchainCreateInfoKHR swapchain_create_info{};
swapchain_create_info.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
swapchain_create_info.surface = mSurface;                   // surface that swapchain tied to
swapchain_create_info.minImageCount = mSwapchainImageCount;     // desired image count (queue length)
swapchain_create_info.imageFormat = mSurfaceFormatKHR.format;
swapchain_create_info.imageColorSpace = mSurfaceFormatKHR.colorSpace;
swapchain_create_info.imageExtent = mSwapchainExtent;
swapchain_create_info.imageArrayLayers = 1;           // the amount of layers each image consists of
swapchain_create_info.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
```

```
swapchain_create_info.preTransform = mSurfaceCapabilitiesKHR.currentTransform;
swapchain_create_info.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;   // igonore alpha blending w
swapchain_create_info.presentMode = mPresentModeKHR;
swapchain_create_info.clipped = VK_TRUE;
swapchain_create_info.oldSwapchain = VK_NULL_HANDLE;    // related with swapchain recreation (window re

// create swapchain and swapchain images
checkError(vkCreateSwapchainKHR(mDevice, &swapchain_create_info, nullptr, &mSwapchain));
```

createSwapchain() #3

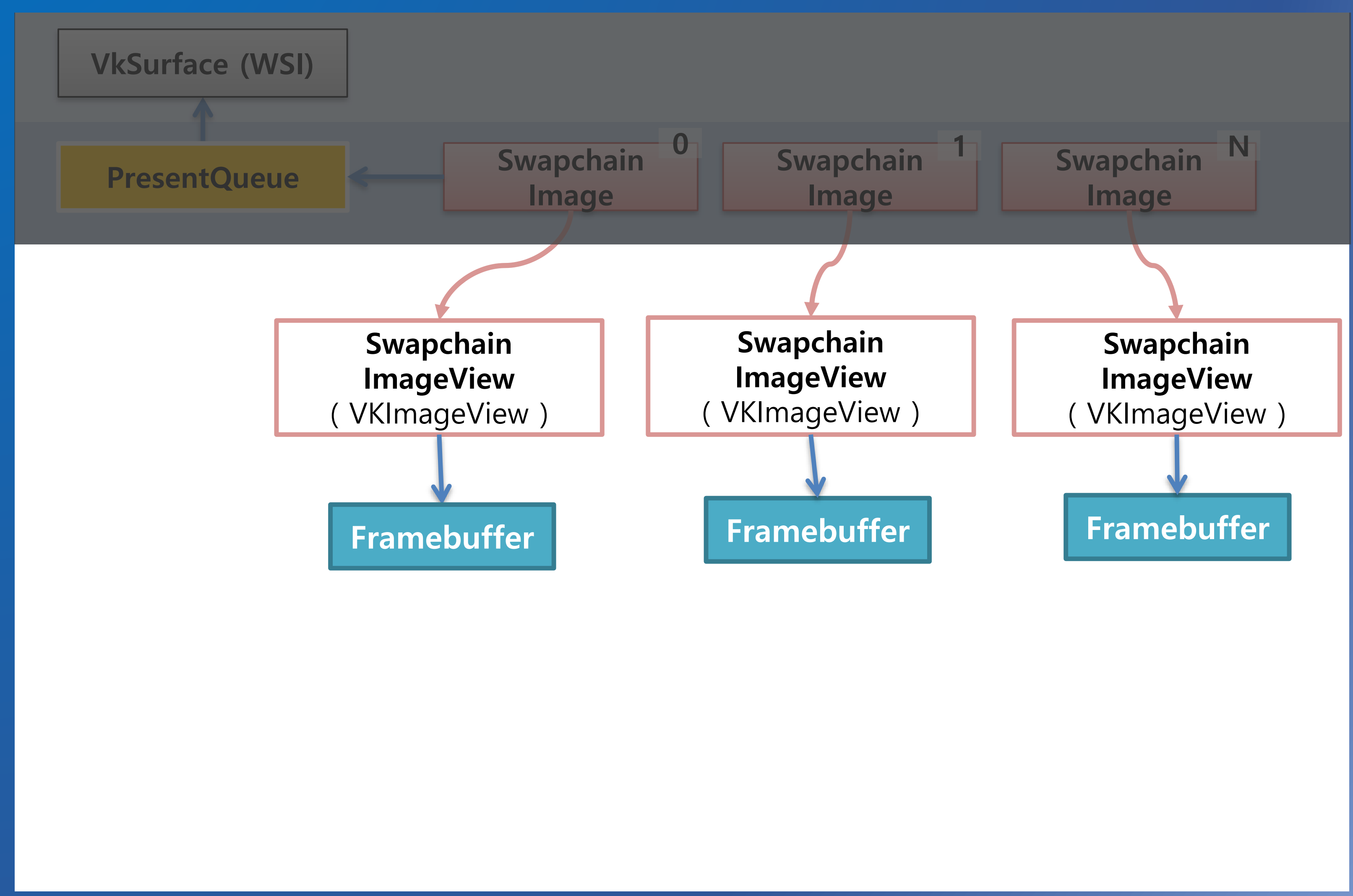
```
class VulkanRenderer
{
private:
    VkSwapchainKHR          mSwapchain;
    uint32_t                mSwapchainImageCount = 2;
    std::vector<VkImage>    mSwapchainImages;

    void createSurface();
    void destroySurface();
};
```

```
    // Retrieve the handle of swapchain image
    vkGetSwapchainImagesKHR(mDevice, mSwapchain, &mSwapchainImageCount, nullptr);
    mSwapchainImages.resize(mSwapchainImageCount);
    vkGetSwapchainImagesKHR(mDevice, mSwapchain, &mSwapchainImageCount, mSwapchainImages.data());
}
```

destroySwapchain()

```
void VulkanRenderer::destroySwapchain()
{
    // clean up the swapchain and swapchain images
    vkDestroySwapchainKHR(mDevice, mSwapchain, nullptr);
}
```

Swapchain Image View



Vulkan.

Swapchain Image View

Create

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();
}
```

Destroy

```
VulkanRenderer::~VulkanRenderer()
{
    destroyImageViews();
    destroySwapchain();

    destroyLogicalDevice();
    destroySurface();
    destroyInstance();
}
```

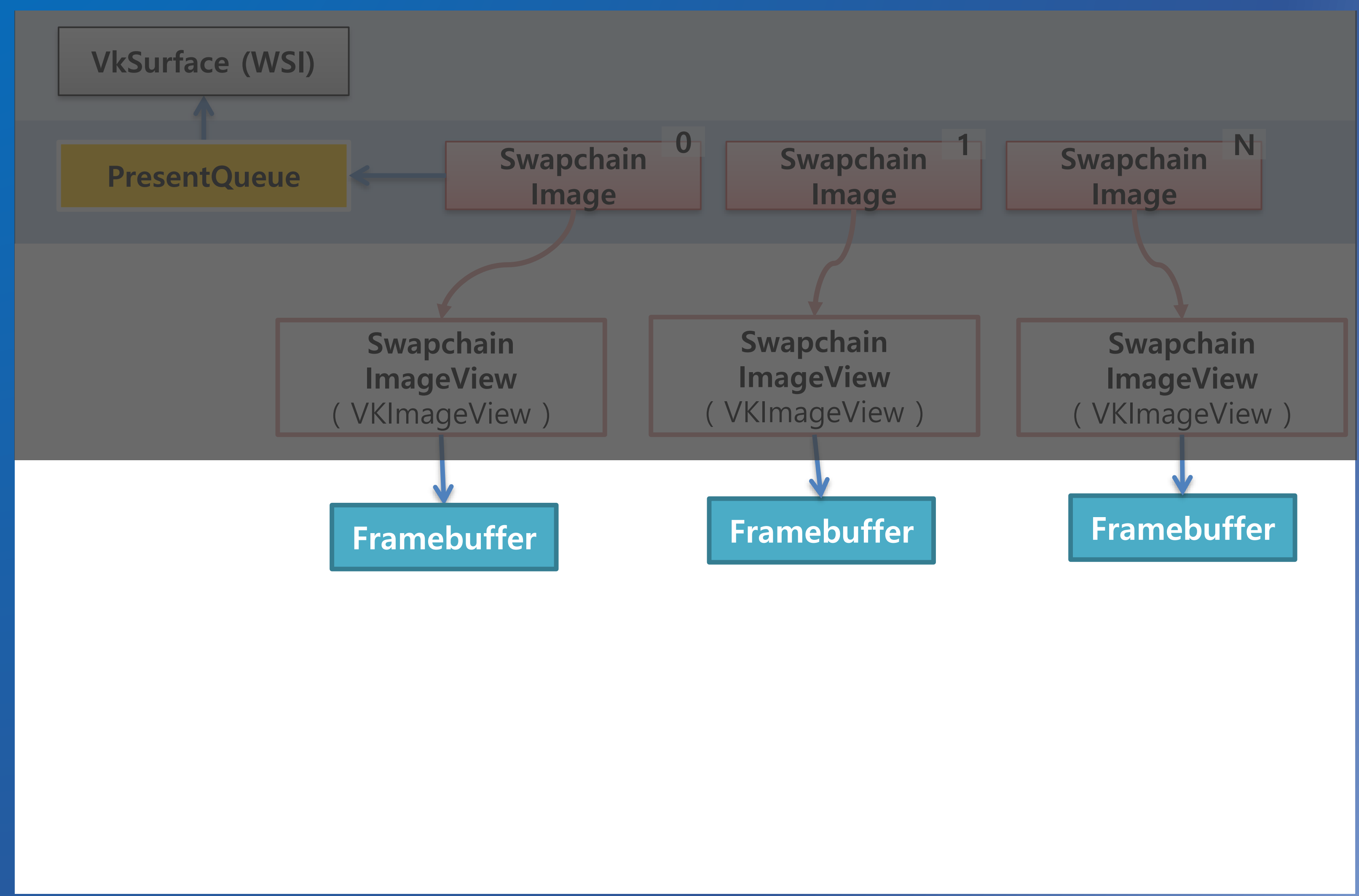
```
void VulkanRenderer::destroyImageViews()
{
    for (auto view : mSwapchainImageViews) {
        vkDestroyImageView(mDevice, view, nullptr);
    }
}
```

createImageViews()

```
class VulkanRenderer
{
private:
    VkSurfaceFormatKHR          mSurfaceFormatKHR = {};
    uint32_t                    mSwapchainImageCount = 2;
    std::vector<VkImage>        mSwapchainImages;
    std::vector<VkImageView>    mSwapchainImageViews;

    void createImageViews();
    void destroyImageViews();
};
```

```
void VulkanRenderer::createImageViews()
{

    mSwapchainImageViews.resize(mSwapchainImageCount);
    // create imageViews for each swapchain images
    for (uint32_t i = 0; i < mSwapchainImageCount; ++i) {
        VkImageViewCreateInfo image_view_create_info{};
        image_view_create_info.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
        image_view_create_info.image = mSwapchainImages[i];
        image_view_create_info.viewType = VK_IMAGE_VIEW_TYPE_2D;
        image_view_create_info.format = mSurfaceFormatKHR.format;       // swapchain image format (VkFormat)
        image_view_create_info.components.r = VK_COMPONENT_SWIZZLE_IDENTITY;
        image_view_create_info.components.g = VK_COMPONENT_SWIZZLE_IDENTITY;
        image_view_create_info.components.b = VK_COMPONENT_SWIZZLE_IDENTITY;
        image_view_create_info.components.a = VK_COMPONENT_SWIZZLE_IDENTITY;
        // image will be used as color targets without any mipmapping levels or multiple layers
        image_view_create_info.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
        image_view_create_info.subresourceRange.baseMipLevel = 0;
        image_view_create_info.subresourceRange.levelCount = 1;
        image_view_create_info.subresourceRange.baseArrayLayer = 0;
        image_view_create_info.subresourceRange.layerCount = 1; // ex) multiple layer needed for stereographic 3D app

        checkError(vkCreateImageView(mDevice, &image_view_create_info, nullptr, &mSwapchainImageViews[i]));
    }

}
```

Framebuffer



Framebuffer

Framebuffer

- Target buffer for color, depth, stencil target
- A frame buffer should be created fitting all swapchain image views

Creating Framebuffer

- Swapchain image view (color, depth, stencil image view)
- Render pass object that declared the framebuffer attachment type
- Number of attachments and attachment objects, extent information

Framebuffer

Create

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();

    createRenderPass();
    createFramebuffers();

    createCommandPool();
    createCommandBuffers();
    createGraphicsPipeline();
    recordCommandBuffers();
    createSemaphores();
}
```

Destroy

```
VulkanRenderer::~VulkanRenderer()
{
    destroySemaphores();
    destroyGraphicsPipeline();
    destroyCommandPool();

    destroyFramebuffers();
    destroyRenderPass();

    destroyImageViews();
    destroySwapchain();

    destroyLogicalDevice();
    destroySurface();
    destroyInstance();
}
```

createFramebuffers() / destroyFramebuffers()

```
class VulkanRenderer
{
private:
    VkRenderPass                mRenderPass;
    std::vector<VkImageView>    mSwapchainImageViews;
    VkExtent2D                  mSwapchainExtent;
    std::vector<VkFramebuffer>  mSwapchainFramebuffers;

    void createFramebuffers();
    void destroyFramebuffers();
};
```

```
void VulkanRenderer::createFramebuffers()
{
    mSwapchainFramebuffers.resize(mSwapchainImageViews.size() );

    for (size_t i = 0; i < mSwapchainImageViews.size(); ++i) {
        VkImageView attachments[] = {
            mSwapchainImageViews[i]
        };

        VkFramebufferCreateInfo framebufferInfo = {};
        framebufferInfo.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
        framebufferInfo.renderPass = mRenderPass;
        framebufferInfo.attachmentCount = 1;
        framebufferInfo.pAttachments = attachments;
        framebufferInfo.width = mSwapchainExtent.width;
        framebufferInfo.height = mSwapchainExtent.height;
        framebufferInfo.layers = 1;

        checkError(vkCreateFramebuffer(mDevice, &framebufferInfo, nullptr, &mSwapchainFramebuffers[i]));
    }
}

void VulkanRenderer::destroyFramebuffers()
{
    for (auto framebuffer : mSwapchainFramebuffers) {
        vkDestroyFramebuffer(mDevice, framebuffer, nullptr);
    }
}
```

Command Buffer

VkSurface (WSI)

PresentQueue

Swapchain Image 0

Swapchain Image 1

Swapchain Image N

Submit

Submit

Submit

CMD Buffer

CMD Buffer

CMD Buffer

BeginCommandBuffer

Begin RenderPass

Bind GraphicsPipeline

Draw

End RenderPass

EndCommandBuffer

Command Buffer

Command Buffers

- Vulkan commands are submitted to queues for execution
- Command buffer can be executed in multi-threaded command jobs
- Command buffers can be reused

Command Pools

- Manage memory for command buffer allocation
- Command buffers are allocated memory from a command pool

Command Pool

Create

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();

    createRenderPass();
    createFramebuffers();

    createCommandPool();
    createCommandBuffers();
    createGraphicsPipeline();
    recordCommandBuffers();
    createSemaphores();
}
```

Destroy

```
VulkanRenderer::~VulkanRenderer()
{
    destroySemaphores();
    destroyGraphicsPipeline();
    destroyCommandPool();

    destroyFramebuffers();
    destroyRenderPass();

    destroyImageViews();
    destroySwapchain();

    destroyLogicalDevice();
    destroySurface();
    destroyInstance();
}
```

createCommandPool() / destroyCommandPool()

```
class VulkanRenderer
{
private:
    VkCommandPool    mCommandPool;

    void createCommandPool();
    void destroyCommandPool();
};
```

```
void VulkanRenderer::createCommandPool()
{
    VulkanQueueFamily queueFamily;
    queueFamily.findQueueFamilies(mGpu, mSurface);

    VkCommandPoolCreateInfo poolInfo = {};
    poolInfo.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
    poolInfo.queueFamilyIndex = queueFamily.mIdxGraphicsFamily;
    poolInfo.flags = 0; // Optional

    checkError(vkCreateCommandPool(mDevice, &poolInfo, nullptr, &mCommandPool));
}

void VulkanRenderer::destroyCommandPool()
{
    vkDestroyCommandPool(mDevice, mCommandPool, nullptr);
}
```


Vulkan

Command Buffer

Create

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();

    createRenderPass();
    createFramebuffers();

    createCommandPool();
    createCommandBuffers();
    createGraphicsPipeline();
    recordCommandBuffers();
    createSemaphores();
}
```

Destroy

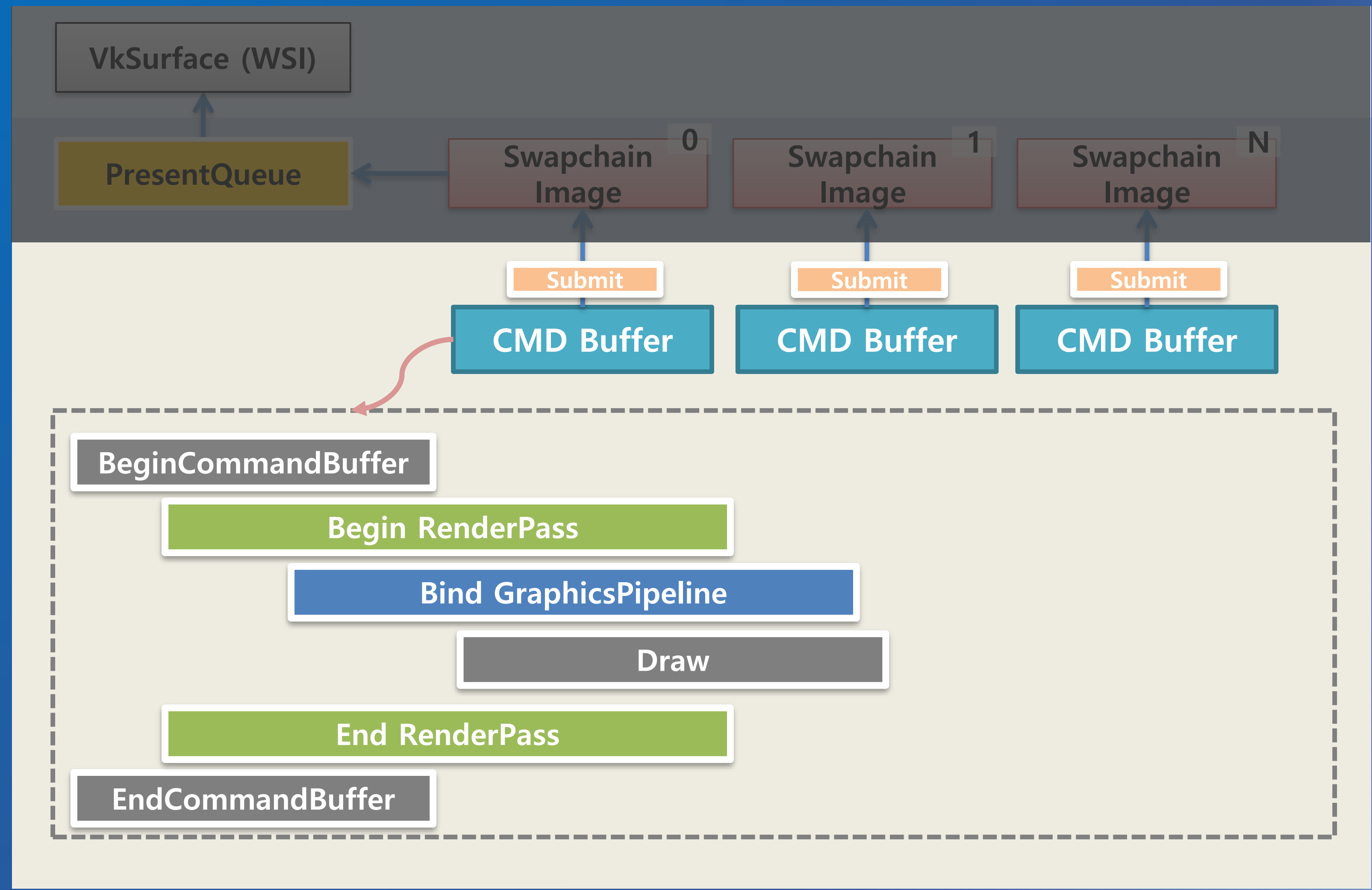
```
VulkanRenderer::~VulkanRenderer()
{
        destroySemaphores();
        destroyGraphicsPipeline();
        destroyCommandPool();

        destroyFramebuffers();
        destroyRenderPass();

        destroyImageViews();
        destroySwapchain();

        destroyLogicalDevice();
        destroySurface();
        destroyInstance();
}
```

※ Command buffers will automatically terminated upon command pool destruction

createCommandBuffers()

```
void VulkanRenderer::createCommandBuffers()
{
    // command buffers will be freed when their command pool is destroyed
    mCommandBuffers.resize(mSwapchainFramebuffers.size());

    // command buffer allocation
    VkCommandBufferAllocateInfo allocInfo = {};
    allocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
    allocInfo.commandPool = mCommandPool;
    allocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
    allocInfo.commandBufferCount = (uint32_t)mCommandBuffers.size();

    checkError(vkAllocateCommandBuffers(mDevice, &allocInfo, mCommandBuffers.data()));
```

Render Pass



Vulkan

Render Pass

Render Pass

- Specify framebuffer attachment type using for rendering

  : Framebuffer attachment information (color buffer, depth buffer, multisampling, etc.)

  : Subpass information (consecutive rendering)

Render Pass Generation

1) Attachment description

2) Subpass description / dependency

3) Render pass create info

4) Render pass

Render Pass

Create

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();

    createRenderPass();
    createFramebuffers();

    createCommandPool();
    createCommandBuffers();
    createGraphicsPipeline();
    recordCommandBuffers();
    createSemaphores();
}
```

Destroy

```
VulkanRenderer::~VulkanRenderer()
{
    destroySemaphores();
    destroyGraphicsPipeline();
    destroyCommandPool();

    destroyFramebuffers();
    destroyRenderPass();

    destroyImageViews();
    destroySwapchain();

    destroyLogicalDevice();
    destroySurface();
    destroyInstance();
}
```

createRenderPass() #1

```
class VulkanRenderer
{
private:
    VkRenderPass     mRenderPass;

    void createRenderPass();
    void destroyRenderPass();
}
```

```
void VulkanRenderer::createRenderPass()
{
    // (There can be multiple attachments)
    VkAttachmentDescription colorAttachment = {};
    colorAttachment.format = mSurfaceFormatKHR.format;          // swapchain image format
    colorAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
    colorAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;       // OP before rendering (color/depth)
    colorAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE;     // OP after rendering (color/depth)
    colorAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
    colorAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
    colorAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
    colorAttachment.finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;

    VkAttachmentReference colorAttachmentRef = {};
    colorAttachmentRef.attachment = 0;
    colorAttachmentRef.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
```

LOAD_OP_LOAD :
LOAD_OP_CLEAR :
LOAD_OP_DONT_CARE :

STORE_OP_STORE :
STORE_OP_DONT_CARE :

Vulkan

createRenderPass() #2

```
class VulkanRenderer
{
private:
    VkRenderPass     mRenderPass;

    void createRenderPass();
    void destroyRenderPass();
}
```

```
// (There can be multiple subpasses)
VkSubpassDescription subPass = {};
subPass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
subPass.colorAttachmentCount = 1;
subPass.pColorAttachments = &colorAttachmentRef;

VkRenderPassCreateInfo renderPassInfo = {};
renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
renderPassInfo.attachmentCount = 1;
renderPassInfo.pAttachments = &colorAttachment;
renderPassInfo.subpassCount = 1;
renderPassInfo.pSubpasses = &subPass;
renderPassInfo.dependencyCount = 0;
renderPassInfo.pDependencies = nullptr;

checkError(vkCreateRenderPass(mDevice, &renderPassInfo, nullptr, &mRenderPass));
```

```
void VulkanRenderer::destroyRenderPass()
{
    vkDestroyRenderPass(mDevice, mRenderPass, nullptr);
}
```

Graphics Pipeline

VkSurface (WSI)

PresentQueue

Swapchain Image 0

Swapchain Image 1

Swapchain Image N

Submit

Submit

Submit

CMD Buffer

CMD Buffer

CMD Buffer

BeginCommandBuffer

Begin RenderPass

Bind GraphicsPipeline

Draw

End RenderPass

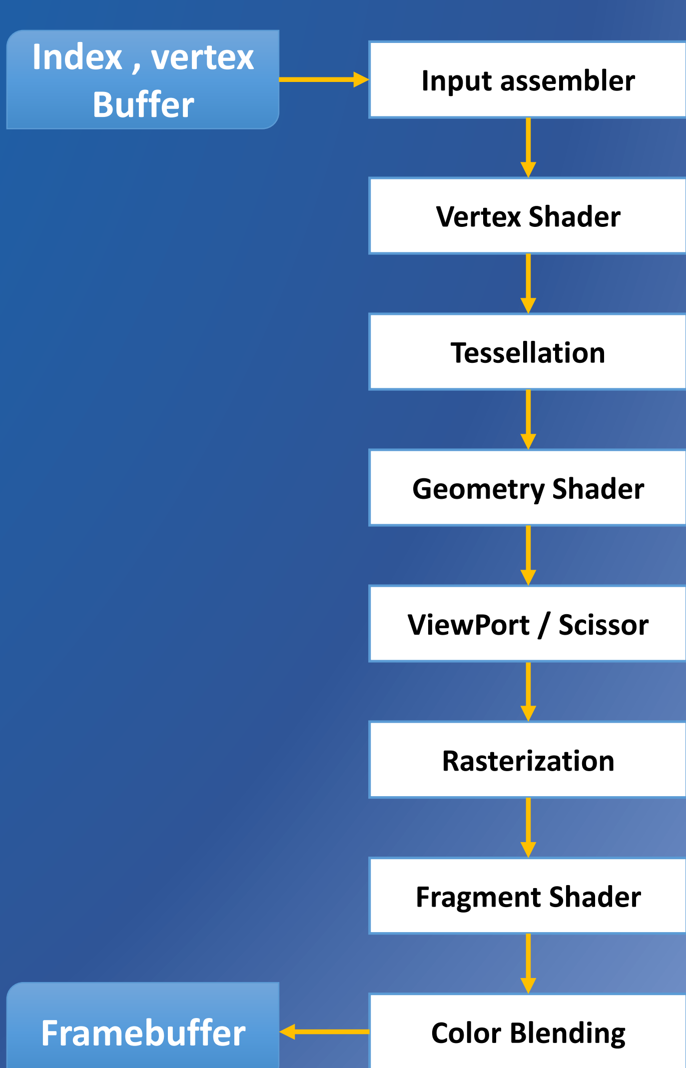
EndCommandBuffer

Graphics Pipeline

Graphics Pipeline

- Rasterizing 3D object to 2D image

- Vulkan explicitly defines each step of the graphics pipeline



Shader / SPIR-V

Vertex shader / Fragment shader

- Converting binary type of SPIR-V based on GLSL (glslang compiler)

SPIR-V

- Pre-compiled bytecode format
- Intermediate language for parallel compute and graphics
- GLSL can be compiled to SPIR-V using the Khronos GLSL open source compiler based on the GL_KHR_vulkan_glsl extension

Vertex Shader (shader.vert)

See also : Vertex buffer version

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

out gl_PerVertex {
    vec4    gl_Position;
};

layout(location = 0) out vec3 fragColor;

vec2 positions[3] = vec2[](
    vec2(0.0, -0.5),
    vec2(0.5, 0.5),
    vec2(-0.5, 0.5)
);

vec3 colors[3] = vec3[](
    vec3(1.0, 0.0, 0.0),
    vec3(0.0, 1.0, 0.0),
    vec3(0.0, 0.0, 1.0)
);

void main() {
    gl_Position = vec4(positions[gl_VertexIndex], 0.0, 1.0);
    fragColor = colors[gl_VertexIndex];
}
```

Vertex attributes

- coordinate
- color

Vulkan

Fragment Shader (shader.frag)

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(location = 0) in vec3 fragColor;

layout(location = 0) out vec4 outColor;

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

Shader Compile

C:/VulkanSDK/1.0.17.0/Bin32/glslangValidator.exe –V shader.vert shader.frag

=> vert.spv / frag.spv

createShaderModule()

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

    checkError(vkCreateShaderModule(mDevice, &create_info, nullptr, &shaderModule));
}
```

```
class VulkanRenderer
{
private:
    void createShaderModule(const std::vector<char>& code, VkShaderModule& shaderModule);
};
```

Graphics Pipeline

Create

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();

    createRenderPass();
    createFramebuffers();

    createCommandPool();
    createCommandBuffers();
    createGraphicsPipeline();
    recordCommandBuffers();
    createSemaphores();
}
```

Destroy

```
VulkanRenderer::~VulkanRenderer()
{
    destroySemaphores();
    destroyGraphicsPipeline();
    destroyCommandPool();

    destroyFramebuffers();
    destroyRenderPass();

    destroyImageViews();
    destroySwapchain();

    destroyLogicalDevice();
    destroySurface();
    destroyInstance();
}
```

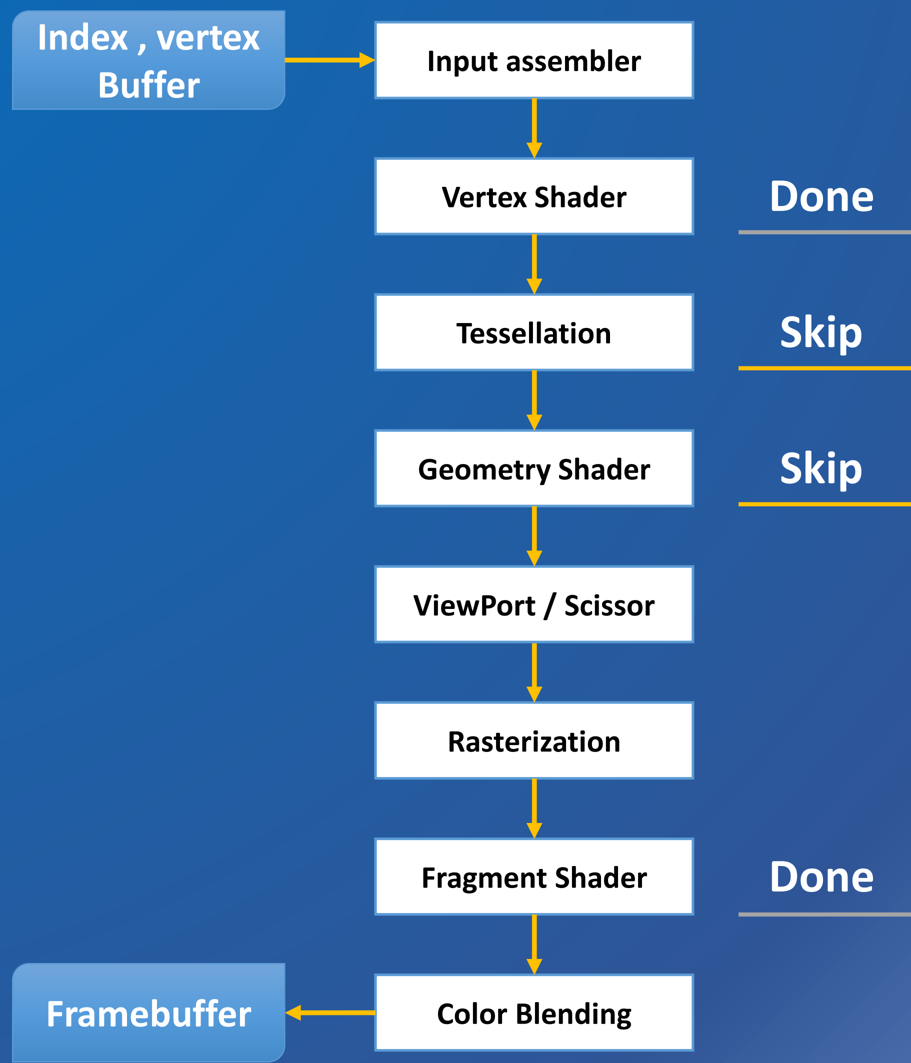
createGraphicsPipeline() #1

```
class VulkanRenderer
{
private:
    VkPipelineLayout            mPipelineLayout;
    VkPipeline                  mPipeline;

    void createGraphicsPipeline();
    void destroyGraphicsPipeline();
    void createShaderModule(const std::vector<char>& code, VkShaderModule& shaderModule);
};
```

```
void VulkanRenderer::createGraphicsPipeline()
{
    VkShaderModule vertShaderModule;
    VkShaderModule fragShaderModule;
    VkPipelineShaderStageCreateInfo vertShaderStageInfo = {};
    VkPipelineShaderStageCreateInfo fragShaderStageInfo = {};

    {
        auto vertShaderCode = ReadShader("shaders/vert.spv");
        auto fragShaderCode = ReadShader("shaders/frag.spv");

        createShaderModule(vertShaderCode, vertShaderModule);
        createShaderModule(fragShaderCode, fragShaderModule);

        vertShaderStageInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
        vertShaderStageInfo.stage = VK_SHADER_STAGE_VERTEX_BIT;
        vertShaderStageInfo.module = vertShaderModule;
        vertShaderStageInfo.pName = "main";
        vertShaderStageInfo.pSpecializationInfo = nullptr;  // values for shader constants

        fragShaderStageInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
        fragShaderStageInfo.stage = VK_SHADER_STAGE_FRAGMENT_BIT;
        fragShaderStageInfo.module = fragShaderModule;
        fragShaderStageInfo.pName = "main";
        fragShaderStageInfo.pSpecializationInfo = nullptr;  // values for shader constants
    }
    VkPipelineShaderStageCreateInfo shaderStages[] = { vertShaderStageInfo, fragShaderStageInfo };
```


Vulkan

Graphics Pipeline



createGraphicsPipeline() #2

```
class VulkanRenderer
{
private:
    VkPipelineLayout            mPipelineLayout;
    VkPipeline                  mPipeline;

    void createGraphicsPipeline();
    void destroyGraphicsPipeline();
    void createShaderModule(const std::vector<char>& code, VkShaderModule& shaderModule);
};
```

```
// 1. vertex input stage[Fixed]
VkPipelineVertexInputStateCreateInfo vertexInputInfo = {};
{
    vertexInputInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
    vertexInputInfo.vertexBindingDescriptionCount = 0;
    vertexInputInfo.pVertexBindingDescriptions = nullptr;       // optional
    vertexInputInfo.vertexAttributeDescriptionCount = 0;
    vertexInputInfo.pVertexAttributeDescriptions = nullptr;     // optional
}

// 2. input assembly stage[Fixed]
VkPipelineInputAssemblyStateCreateInfo inputAssembly = {};
{
    inputAssembly.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
    inputAssembly.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
    inputAssembly.primitiveRestartEnable = VK_FALSE;
}

// 3. Vertext shader stage[Programmable]
// 4. Tessellation stage[Programmable] : optional
// 5. Geometry shader stage[Programmable] : optional
```


Vulkan

createGraphicsPipeline() #3

```
class VulkanRenderer
{
private:
    VkPipelineLayout            mPipelineLayout;
    VkPipeline                  mPipeline;

    void createGraphicsPipeline();
    void destroyGraphicsPipeline();
    void createShaderModule(const std::vector<char>& code, VkShaderModule& shaderModule);
};
```

```
// 6-1. vieports and scissors[Fixed]
VkViewport viewport = {};       // region of the framebuffer
VkRect2D scissor = {};
VkPipelineViewportStateCreateInfo viewportState = {};
{
    viewport.x = 0.0f;
    viewport.y = 0.0f;
    viewport.width = (float)mSwapchainExtent.width;
    viewport.height = (float)mSwapchainExtent.height;
    viewport.minDepth = 0.0f;
    viewport.maxDepth = 1.0f;

    scissor.offset = { 0, 0 };
    scissor.extent = mSwapchainExtent;

    viewportState.sType = VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
    viewportState.viewportCount = 1;
    viewportState.pViewports = &viewport;
    viewportState.scissorCount = 1;
    viewportState.pScissors = &scissor;
}
```


Vulkan

createGraphicsPipeline() #4

```
class VulkanRenderer
{
private:
    VkPipelineLayout            mPipelineLayout;
    VkPipeline                  mPipeline;

    void createGraphicsPipeline();
    void destroyGraphicsPipeline();
    void createShaderModule(const std::vector<char>& code, VkShaderModule& shaderModule);
};
```

```
// 6-2. Rasterizer stage[Fixed]
VkPipelineRasterizationStateCreateInfo rasterizer = {};
{
    rasterizer.sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
    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;          // optional
    rasterizer.depthBiasClamp = 0.0f;           // optional
    rasterizer.depthBiasSlopeFactor = 0.0f;         // optional
}

// 6-3. Multisampling stage[Fixed]
VkPipelineMultisampleStateCreateInfo multisampling = {};
{
    multisampling.sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
    multisampling.sampleShadingEnable = VK_FALSE;
    multisampling.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;
    multisampling.minSampleShading = 1.0f;          // optional
    multisampling.pSampleMask = nullptr;        // optional
    multisampling.alphaToCoverageEnable = VK_FALSE;     // optional
    multisampling.alphaToOneEnable = VK_FALSE;      // optional
}

// 6-4. Depth and stencil testing[Fixed]
//VkPipelineDepthStencilStateCreateInfo depthStencil = {};
```

createGraphicsPipeline() #5

```cpp
class VulkanRenderer
{
private:
    VkPipelineLayout            mPipelineLayout;
    VkPipeline                  mPipeline;

    void createGraphicsPipeline();
    void destroyGraphicsPipeline();
    void createShaderModule(const std::vector<char>& code, VkShaderModule& shaderModule);
};
```

```cpp
// 7. Fragment shader stage[Programmable]

// 8. Color blending[Fixed]
VkPipelineColorBlendAttachmentState colorBlendAttachment = {};  // configuration per attached framebuffer
VkPipelineColorBlendStateCreateInfo colorBlending = {};         // global color blending
{
    colorBlendAttachment.colorWriteMask = VK_COLOR_COMPONENT_R_BIT |
        VK_COLOR_COMPONENT_G_BIT |
        VK_COLOR_COMPONENT_B_BIT |
        VK_COLOR_COMPONENT_A_BIT;
    colorBlendAttachment.blendEnable = VK_FALSE;
    colorBlendAttachment.srcColorBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
    colorBlendAttachment.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
    colorBlendAttachment.colorBlendOp = VK_BLEND_OP_ADD;
    colorBlendAttachment.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE;
    colorBlendAttachment.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO;
    colorBlendAttachment.alphaBlendOp = VK_BLEND_OP_ADD;

    colorBlending.sType = VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
    colorBlending.logicOpEnable = VK_FALSE;
    colorBlending.logicOp = VK_LOGIC_OP_COPY;
    colorBlending.attachmentCount = 1;
    colorBlending.pAttachments = &colorBlendAttachment;
    colorBlending.blendConstants[0] = 0.0f;
    colorBlending.blendConstants[1] = 0.0f;
    colorBlending.blendConstants[2] = 0.0f;
    colorBlending.blendConstants[3] = 0.0f;
}
```

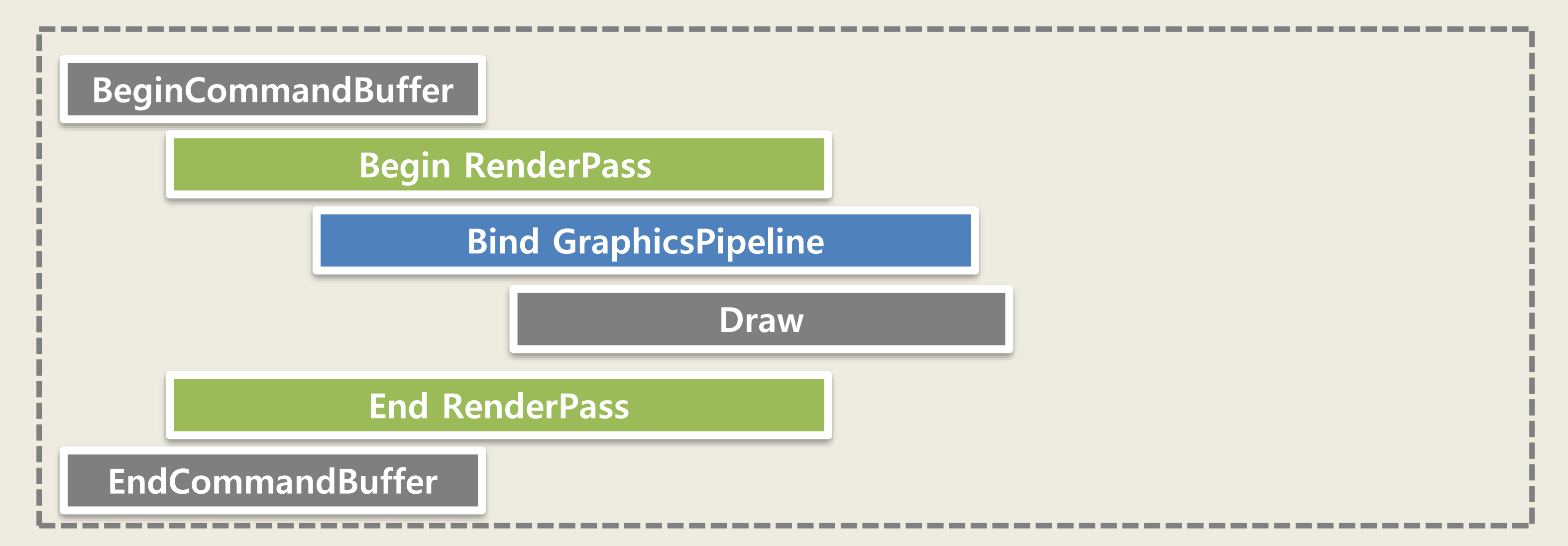

Vulkan

createGraphicsPipeline() #6

```cpp
class VulkanRenderer
{
private:
    VkPipelineLayout            mPipelineLayout;
    VkPipeline                  mPipeline;

    void createGraphicsPipeline();
    void destroyGraphicsPipeline();
    void createShaderModule(const std::vector<char>& code, VkShaderModule& shaderModule);
};
```

```cpp
    // Pipeline layout for passing uniform values to shaders
    VkPipelineLayoutCreateInfo pipelineLayoutInfo = {};
    {
        pipelineLayoutInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
        pipelineLayoutInfo.setLayoutCount = 0;
        pipelineLayoutInfo.pSetLayouts = nullptr;
        pipelineLayoutInfo.pushConstantRangeCount = 0;
        pipelineLayoutInfo.pPushConstantRanges = 0;

        checkError(vkCreatePipelineLayout(mDevice, &pipelineLayoutInfo, nullptr, &mPipelineLayout));
    }

    VkGraphicsPipelineCreateInfo pipelineInfo = {};
    pipelineInfo.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
    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
    pipelineInfo.pColorBlendState = &colorBlending;
    pipelineInfo.pDynamicState = nullptr; // Optional
    pipelineInfo.layout = mPipelineLayout;
    pipelineInfo.renderPass = mRenderPass;
    pipelineInfo.subpass = 0;
    pipelineInfo.basePipelineHandle = VK_NULL_HANDLE;    // deriving from an existing pipeline
    pipelineInfo.basePipelineIndex = -1; // Optional

    checkError(vkCreateGraphicsPipelines(mDevice, VK_NULL_HANDLE, 1, &pipelineInfo, nullptr, &mPipeline));

    // destory shader modules after graphics pipeline creation
    vkDestroyShaderModule(mDevice, fragShaderModule, nullptr);
    vkDestroyShaderModule(mDevice, vertShaderModule, nullptr);
}
```

Recording Command Buffer




```
// starting command buffer recording
for (size_t i = 0; i < mCommandBuffers.size(); i++) {
    VkCommandBufferBeginInfo beginInfo = {};
    beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
    beginInfo.flags = VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT;
    beginInfo.pInheritanceInfo = nullptr; // Optional

    checkError(vkBeginCommandBuffer(mCommandBuffers[i], &beginInfo));

    //
    // Render pass
    //

    checkError(vkEndCommandBuffer(mCommandBuffers[i]));
}
```

ONE_TIME_SUBMIT_BIT

RENDER_PASS_CONTINUE_BIT

SIMULTANEOUS_USE_BIT
Command buffer can be reused

recordingCommandBuffers()

```
class VulkanRenderer
{
private:
    std::vector<VkCommandBuffer>    mCommandBuffers;

    void createCommandBuffers();
};
```

```
//
// Render pass
//
{
    VkRenderPassBeginInfo renderPassInfo = {};
    renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
    renderPassInfo.renderPass = mRenderPass;
    renderPassInfo.framebuffer = mSwapchainFramebuffers[i]; // attachment to bind

    // size of render area
    renderPassInfo.renderArea.offset = { 0, 0 };
    renderPassInfo.renderArea.extent = mSwapchainExtent;

    VkClearValue clearColor = { 0.0f, 0.0f, 0.0f, 1.0f };
    renderPassInfo.clearValueCount = 1;
    renderPassInfo.pClearValues = &clearColor;

    vkCmdBeginRenderPass(mCommandBuffers[i], &renderPassInfo, VK_SUBPASS_CONTENTS_INLINE);

    vkCmdBindPipeline(mCommandBuffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS, mPipeline);

    vkCmdDraw(mCommandBuffers[i], 3, 1, 0, 0);

    vkCmdEndRenderPass(mCommandBuffers[i]);
}
```


Vulkan

Recording Command Buffer

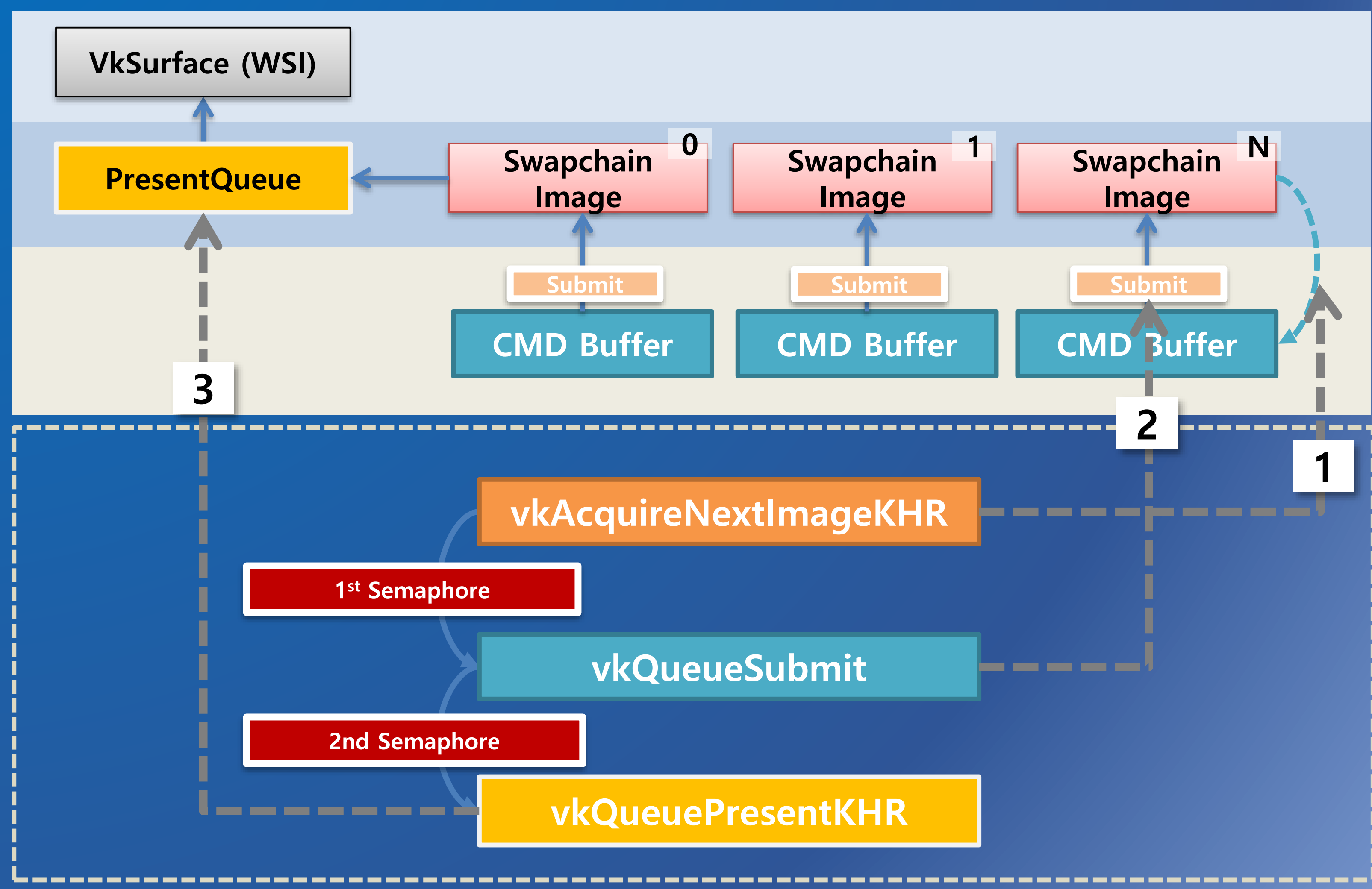
Create

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();

    createRenderPass();
    createFramebuffers();

    createCommandPool();
    createCommandBuffers();
    createGraphicsPipeline();
    recordCommandBuffers();
    createSemaphores();
}
```

Destroy

```
VulkanRenderer::~VulkanRenderer()
{
    destroySemaphores();
    destroyGraphicsPipeline();
    destroyCommandPool();

    destroyFramebuffers();
    destroyRenderPass();

    destroyImageViews();
    destroySwapchain();

    destroyLogicalDevice();
    destroySurface();
    destroyInstance();
}
```

Drawing Frame



Vulkan.

Semaphore

Synchronization

- Semaphore : Synchronize with each queue and command buffer sync
- Fence : Waiting for GPU ready on the CPU side (Fences are mainly designed to synchronize your application itself with rendering operation and can be used by the host to determine completion of execution of queue operations without GPU involvement)

Semaphore

- Usually using two type of semaphore for drawing

1) Getting swapchain images (waiting for rendering)

2) Returning signal when rendering finished

Semaphore

Create

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();

    createRenderPass();
    createFramebuffers();

    createCommandPool();
    createCommandBuffers();
    createGraphicsPipeline();
    recordCommandBuffers();
    createSemaphores();
}
```

Destroy

```
VulkanRenderer::~VulkanRenderer()
{
    destroySemaphores();
    destroyGraphicsPipeline();
    destroyCommandPool();

    destroyFramebuffers();
    destroyRenderPass();

    destroyImageViews();
    destroySwapchain();

    destroyLogicalDevice();
    destroySurface();
    destroyInstance();
}
```

createSemaphores() / destroySemaphores()

```
void VulkanRenderer::createSemaphores()
{
    VkSemaphoreCreateInfo semaphoreInfo = {};
    semaphoreInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;

    checkError(vkCreateSemaphore(mDevice, &semaphoreInfo, nullptr, &mSemaphore_Image_Available));
    checkError(vkCreateSemaphore(mDevice, &semaphoreInfo, nullptr, &mSemaphore_Render_Finished));

}

void VulkanRenderer::destroySemaphores()
{
    vkDestroySemaphore(mDevice, mSemaphore_Render_Finished, nullptr);
    vkDestroySemaphore(mDevice, mSemaphore_Image_Available, nullptr);
}
```

```
class VulkanRenderer
{
private:
    VkSemaphore     mSemaphore_Image_Available;
    VkSemaphore     mSemaphore_Render_Finished;

    void createSemaphores();
    void destroySemaphores();
};
```

drawFrame()

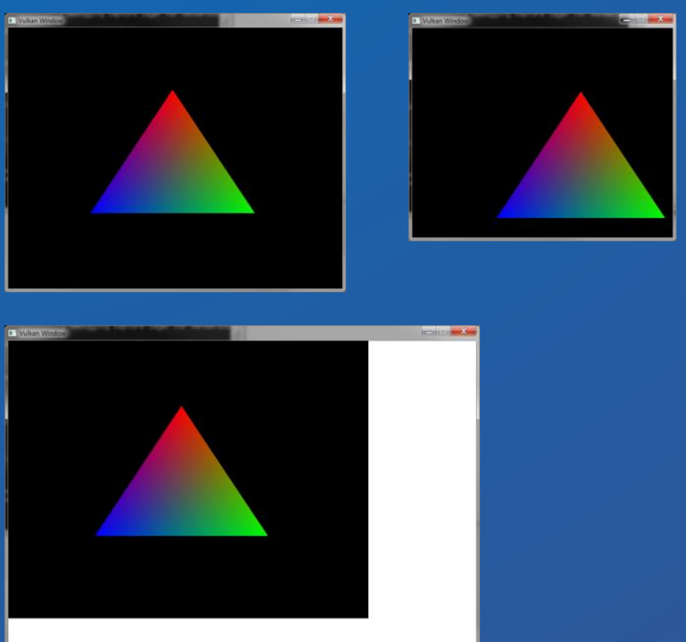
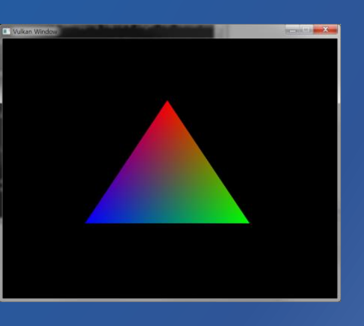
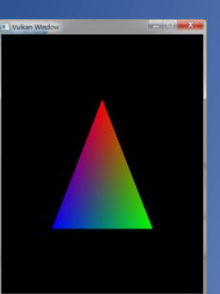
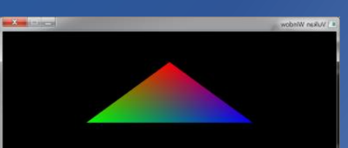
```
void VulkanRenderer::drawFrame()
{
    // 1. acquiring an image from the swapchain
    //    : this image is attached in the framebuffer
    uint32_t imageIndex;
    vkAcquireNextImageKHR(mDevice, mSwapchain, UINT64_MAX, mSemaphore_Image_Available, VK_NULL_HANDLE, &imageIndex);

    // 2. submitting the command buffer to the graphics queue
    VkSubmitInfo submitInfo = {};
    VkSemaphore waitSemaphores[] = { mSemaphore_Image_Available };
    VkSemaphore signalSemaphores[] = { mSemaphore_Render_Finished };
    VkPipelineStageFlags waitStages[] = { VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT };
    submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
    submitInfo.waitSemaphoreCount = 1;
    submitInfo.pWaitSemaphores = waitSemaphores;
    submitInfo.pWaitDstStageMask = waitStages;
    submitInfo.commandBufferCount = 1;
    submitInfo.pCommandBuffers = &mCommandBuffers[imageIndex];
    submitInfo.signalSemaphoreCount = 1;
    submitInfo.pSignalSemaphores = signalSemaphores;

    checkError(vkQueueSubmit(mGraphicsQueue, 1, &submitInfo, VK_NULL_HANDLE));

    // 3. return the image to the swapchain for presentation
    VkPresentInfoKHR presentInfo = {};
    VkSwapchainKHR swapchains[] = { mSwapchain };
    presentInfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
    presentInfo.waitSemaphoreCount = 1;
    presentInfo.pWaitSemaphores = signalSemaphores;
    presentInfo.swapchainCount = 1;
    presentInfo.pSwapchains = swapchains;
    presentInfo.pImageIndices = &imageIndex;
    presentInfo.pResults = nullptr;      // optional

    vkQueuePresentKHR(mPresentQueue, &presentInfo);
}
```

```
class VulkanRenderer
{
private:
    VkQueue mGraphicsQueue   = VK_NULL_HANDLE;
    VkQueue mPresentQueue    = VK_NULL_HANDLE;
    VkSwapchainKHR       mSwapchain;

    std::vector<VkCommandBuffer>    mCommandBuffers;
    VkSemaphore       mSemaphore_Image_Available;
    VkSemaphore       mSemaphore_Render_Finished;
public:
    void drawFrame();
};
```

Swapchain Recreation

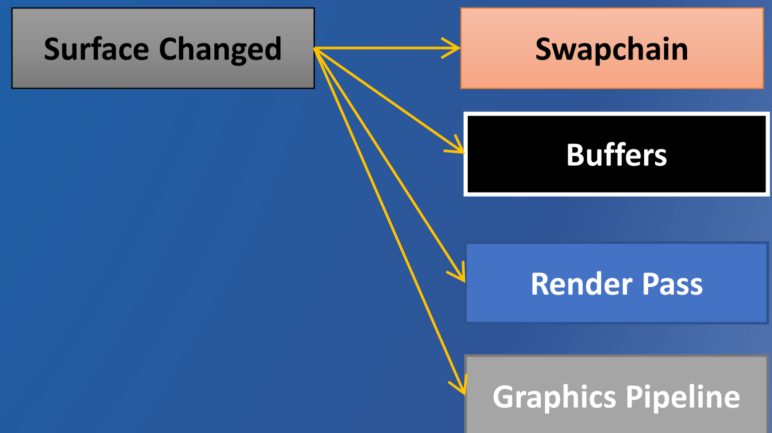
Swapchain Recreation

- If window surface size is changed, swapchain needs to be recreated
- Display rotation, pause/resume, scaling can also require swapchain recreation

Swapchain Recreation (Resize case)

Swapchain Dependency



- Window changed > Surface changed > Updating **swapchain**
- Surface format changed > Updating **render pass**

- Window/surface resolution changed > Updating **buffer** objects
  (framebuffer/depth buffer/command buffer)

- Viewport, scissor changed > Updating **graphics pipeline**

Vulkan

reInitSwapchain()

```
class VulkanRenderer
{
private:
    VkQueue          mGraphicsQueue = VK_NULL_HANDLE;
    VkQueue          mPresentQueue  = VK_NULL_HANDLE;
    VkSwapchainKHR   mSwapchain;

    std::vector<VkCommandBuffer>    mCommandBuffers;
    VkSemaphore      mSemaphore_Image_Available;
    VkSemaphore      mSemaphore_Render_Finished;

public:
    void drawFrame();
};
```

```
void VulkanRenderer::reInitSwapChain()
{
    if (mDevice) {
        vkDeviceWaitIdle(mDevice);
    }

    if (mSwapchain == VK_NULL_HANDLE) {
        return;
    }

    destroyFramebuffers();

    destroyGraphicsPiepline();
    destroyRenderPass();

    destroyImageViews();

    createSwapchain();
    createImageViews();        // <- swapchain image update

    createRenderPass();        // <- swapchain image format update
    createGraphicsPipeline();  // <- viewport, scissor update

    createFramebuffers();      // <- swapchain image update
    createCommandBuffers();    // <- swapchain image update
}
```

createSwapchain()

```
void VulkanRenderer::createSwapchain()
{
    SwapchainInfo swapchainInfo;
    // ...

    VkSwapchainKHR oldSwapchain = mSwapchain;
    VkSwapchainCreateInfoKHR swapchain_create_info{};
    // ...

    swapchain_create_info.oldSwapchain = oldSwapchain;

    // create swapchain and swapchain images
    VkSwapchainKHR newSwapchain;
    checkError(vkCreateSwapchainKHR(mDevice, &swapchain_create_info, nullptr, &newSwapchain));


    if (mSwapchain != VK_NULL_HANDLE) {
        vkDestroySwapchainKHR(mDevice, mSwapchain, nullptr);
    }
    mSwapchain = newSwapchain;


    // ...
}
```


Vulkan

Swapchain Recreation

Create

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();

    createRenderPass();
    createFramebuffers();

    createCommandPool();
    createCommandBuffers();
    createGraphicsPipeline();
    recordCommandBuffers();
    createSemaphores();
}
```

Destroy

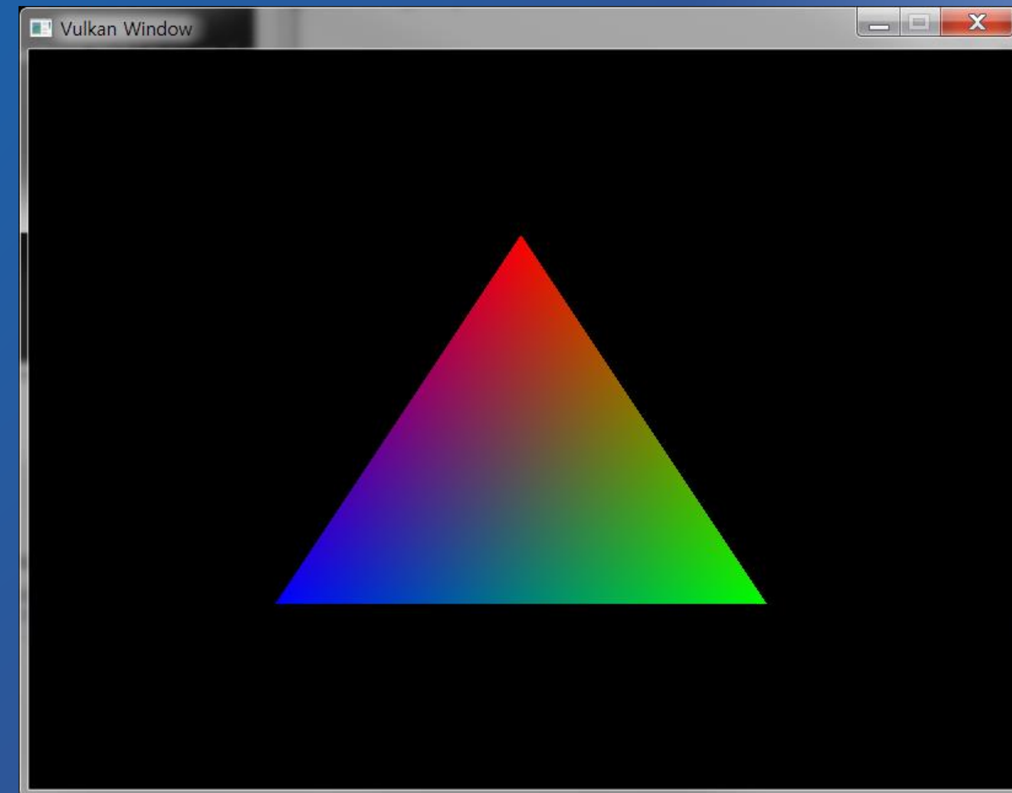
```
VulkanRenderer::~VulkanRenderer()
{
    destroySemaphores();
    destroyGraphicsPipeline();
    destroyCommandPool();

    destroyFramebuffers();
    destroyRenderPass();

    destroyImageViews();
    destroySwapchain();

    destroyLogicalDevice();
    destroySurface();
    destroyInstance();
}
```

Recreate

```
void VulkanRenderer::reInitSwapChain()
{
    if (mDevice) {
        vkDeviceWaitIdle(mDevice);
    }

    if (mSwapchain == VK_NULL_HANDLE) {
        return;
    }

    destroyGraphicsPipeline();

    destroyFramebuffers();
    destroyRenderPass();

    destroyImageViews();

    createSwapchain();
    createImageViews();

    createRenderPass();
    createFramebuffers();

    createCommandBuffers();
    createGraphicsPipeline();
    recordCommandBuffers();
}
```

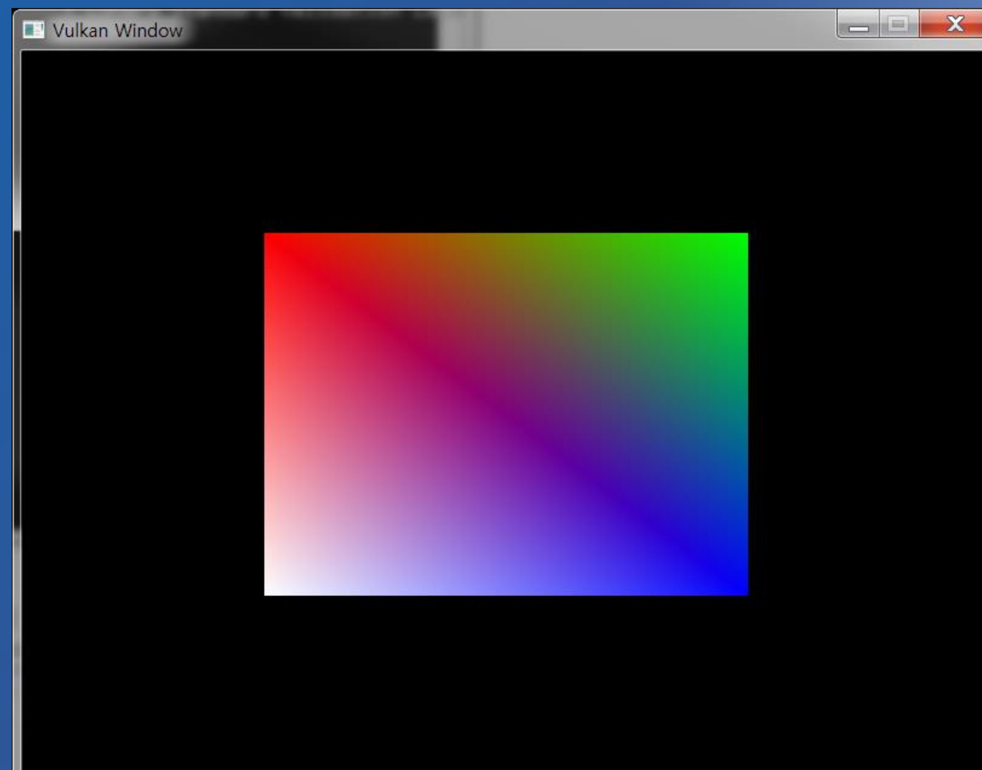
Drawing a Triangle

- Window System / Surface
- Present Queue
- Swapchain / Framebuffer
- Command Buffer
- Render Pass
- Graphics Pipeline
- Shader (SPIR-V)
- Swapchain Recreation



Drawing a Rectangle

- Vertex Buffer
- Staging Buffer
- Index Buffer



Vertex Buffer

Vertex Buffer

- Vertex shader get vertex attribute input using vertex buffer
  (coordinates, color, texture coordinates, etc.)

Sequence using Vertex Buffer

- Modify the vertex shader
- Define a vertex data
- Set the vertex binding description
- Set the vertex attribute description
- Set the vertex binding / attribute description in vulkan graphics pipeline
- Create the vertex buffer
- Draw using the vertex buffer

Update the Vertex Shader

Vertex Shader

- Vertex attribute : get per-vertex data from the program (vertex buffer)
- Use the "in" keyword to get attributes from vertex buffer
- Compile shader again after updating vertex shader

Vertex Shader (shader.vert)

See also : Fixed version

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(location = 0) in vec2 inPosition;
layout(location = 1) in vec3 inColor;

layout(location = 0) out vec3 fragColor;

out gl_PerVertex {
    vec4    gl_Position;
};

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

Vertex attributes

- coordinate
- color

Vulkan.

Define the Vertex Data

Vertex Data

- Use GLM library to use linear algebra (vector, matrix type, etc.)

※ GLM supports C++ compatible vector types (vec2, vec3, etc.)

Vulkan.

Define the Vertex data

```
#include <glm/glm.hpp>
#include <vector>

struct Vertex {
    glm::vec2   pos;
    glm::vec3   color;
};

// array of vertex data
const std::vector<Vertex> gVertices = {
    { {0.0f, -0.5f}, {1.0f, 0.0f, 0.0f} },  // coordinate, color
    { {0.5f, 0.5f}, {0.0f, 1.0f, 0.0f} },
    { {-0.5f, 0.5f}, {0.0f, 0.0f, 1.0f} }
};
```

Vertex Binding/Attribute Description

Vertex Binding Description

- One unit of information in the data array

  (Instance unit in the case of instance rendering)

- VkVertexInputBindingDescription

Vertex Attribute Description

- Specify binding index, location, format, offset information of vertex data
- VkVertexInputAttributeDescription

Vertex Binding/Attribute Description

```
struct Vertex {
    glm::vec2   pos;
    glm::vec3   color;

    static VkVertexInputBindingDescription getBindingDescription() {
        VkVertexInputBindingDescription bindingDescription = {};
        bindingDescription.binding      = 0;      // index of the binding in the array of bindings
        bindingDescription.stride       = sizeof(Vertex);
        bindingDescription.inputRate    = VK_VERTEX_INPUT_RATE_VERTEX;

        return bindingDescription;
    }

    static std::array<VkVertexInputAttributeDescription, 2> getAttributeDescriptions() {
        std::array<VkVertexInputAttributeDescription, 2> attributeDescriptions = {};

        attributeDescriptions[0].binding    = 0;    // index of the binding to get per-vertex data
        attributeDescriptions[0].location   = 0;    // location directive of the input in the vertex shader
        attributeDescriptions[0].format     = VK_FORMAT_R32G32_SFLOAT;      // vec2
        attributeDescriptions[0].offset     = offsetof(Vertex, pos);

        attributeDescriptions[1].binding    = 0;
        attributeDescriptions[1].location   = 1;
        attributeDescriptions[1].format     = VK_FORMAT_R32G32B32_SFLOAT;   // vec3
        attributeDescriptions[1].offset     = offsetof(Vertex, color);

        return attributeDescriptions;
    }
};
```

Define the Vertex Data

Create

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();

    createRenderPass();
    createFramebuffers();

    createCommandPool();
    createCommandBuffers();
    createDescriptorSetLayout();
    createGraphicsPipeline();

    createVertexBuffer();

    recordCommandBuffers();
    createSemaphores();
}
```

Destroy

```
VulkanRenderer::~VulkanRenderer()
{
    destroySemaphores();

    destroyVertexBuffer();

    destroyGraphicsPipeline();
    destroyDescriptorSetLayout();
    destroyCommandPool();

    destroyFramebuffers();
    destroyRenderPass();

    destroyImageViews();
    destroySwapchain();

    destroyLogicalDevice();
    destroySurface();
    destroyInstance();
}
```

createGraphicsPipeline()

```
void VulkanRenderer::createGraphicsPipeline()
{
    // ...

    // 1. vertex input stage[Fixed]
    VkPipelineVertexInputStateCreateInfo vertexInputInfo = {};
    auto bindingDescription = Vertex::getBindingDescription();
    auto attributeDescriptions = Vertex::getAttributeDescriptions();
    {
        vertexInputInfo.sType                           = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
        vertexInputInfo.vertexBindingDescriptionCount   = 1;
        vertexInputInfo.vertexAttributeDescriptionCount = (uint32_t)attributeDescriptions.size();
        vertexInputInfo.pVertexBindingDescriptions      = &bindingDescription;
        vertexInputInfo.pVertexAttributeDescriptions    = attributeDescriptions.data();
    }

    // ...
}
```

Create the Vertex Buffer

Vulkan Buffer

- Can store any data, GPU memory area
- User needs to allocate memory explicitly

Vertex Buffer Creation

1) Create the vertex buffer
2) Check the memory requirement
3) Allocate the memory (CPU accessible)
4) Binding the memory to the vertex buffer
5) Copy the vertex data to the vertex buffer

Create the Vertex Buffer

Create

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();

    createRenderPass();
    createFramebuffers();

    createCommandPool();
    createCommandBuffers();
    createDescriptorSetLayout();
    createGraphicsPipeline();

    createVertexBuffer();

    recordCommandBuffers();
    createSemaphores();
}
```

Destroy

```
VulkanRenderer::~VulkanRenderer()
{
    destroySemaphores();

    destroyVertexBuffer();

    destroyGraphicsPipeline();
    destroyDescriptorSetLayout();
    destroyCommandPool();

    destroyFramebuffers();
    destroyRenderPass();

    destroyImageViews();
    destroySwapchain();

    destroyLogicalDevice();
    destroySurface();
    destroyInstance();
}
```


Vulkan

createVertexBuffer()

```
class VulkanRenderer
{
private:
    VkBuffer         mVertexBuffer;
    VkDeviceMemory   mVertexBufferMem;

    void createVertexBuffer();
    void destroyVertexBuffer();
};
```

```
void VulkanRenderer::createVertexBuffer()
{
    // 1. create vertex buffer (GPU buffer)
    VkBufferCreateInfo bufferInfo = {};
    bufferInfo.sType           = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    bufferInfo.size            = sizeof(gVertices[0]) * gVertices.size();
    bufferInfo.usage           = VK_BUFFER_USAGE_VERTEX_BUFFER_BIT;
    bufferInfo.sharingMode     = VK_SHARING_MODE_EXCLUSIVE;    // used by graphics queue only

    checkError( vkCreateBuffer(mDevice, &bufferInfo, nullptr, &mVertexBuffer) );

    // 2. retrieve memory rquirement of vertex buffer
    VkMemoryRequirements memRequirements;
    vkGetBufferMemoryRequirements(mDevice, mVertexBuffer, &memRequirements);

    // 3. allocate memory for vertext buffer
    VkMemoryAllocateInfo allocInfo = {};
    allocInfo.sType              = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    allocInfo.allocationSize     = memRequirements.size;
    allocInfo.memoryTypeIndex    = findMemoryType(mGpu, memRequirements.memoryTypeBits,
                                     VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT);

    checkError( vkAllocateMemory(mDevice, &allocInfo, nullptr, &mVertexBufferMem) );

    // 4. binding memory with vertex buffer
    checkError( vkBindBufferMemory(mDevice, mVertexBuffer, mVertexBufferMem, 0) );

    // 5. copy vertex data(CPU mem) to the buffer(GPU buffer)
    void* data;
    checkError( vkMapMemory(mDevice, mVertexBufferMem, 0, bufferInfo.size, 0, &data) );
        memcpy(data, gVertices.data(), (size_t)bufferInfo.size);
    vkUnmapMemory(mDevice, mVertexBufferMem);
}
```

findMemoryType()

```
uint32_t findMemoryType(VkPhysicalDevice gpu, uint32_t typeFilter, VkMemoryPropertyFlags properties)
{
    VkPhysicalDeviceMemoryProperties memProperties;
    vkGetPhysicalDeviceMemoryProperties(gpu, &memProperties);

    for (uint32_t i = 0; i < memProperties.memoryTypeCount ; ++i) {
        if ( typeFilter & (1 << i) && (memProperties.memoryTypes[i].propertyFlags & properties) == properties ) {
            return i;
        }
    }

    assert(0 && "Failed to find suitable memory type!");
    std::exit(-1);
}
```

destroyVertexBuffer()

```
class VulkanRenderer
{
private:
    VkBuffer        mVertexBuffer;
    VkDeviceMemory  mVertexBufferMem;

    void createVertexBuffer();
    void destroyVertexBuffer();
};
```

```
void VulkanRenderer::destroyVertexBuffer()
{
    vkFreeMemory(mDevice, mVertexBufferMem, nullptr);
    vkDestroyBuffer(mDevice, mVertexBuffer, nullptr);
}
```

Drawing using the Vertex Buffer

Drawing with the Vulkan Buffer

- Bind the vertex buffer before running the drawing command

Drawing using the Vertex Buffer

Create

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();

    createRenderPass();
    createFramebuffers();

    createCommandPool();
    createCommandBuffers();
    createDescriptorSetLayout();
    createGraphicsPipeline();

    createVertexBuffer();

    recordCommandBuffers();
    createSemaphores();
}
```

Destroy

```
VulkanRenderer::~VulkanRenderer()
{
    destroySemaphores();

    destroyVertexBuffer();

    destroyGraphicsPipeline();
    destroyDescriptorSetLayout();
    destroyCommandPool();

    destroyFramebuffers();
    destroyRenderPass();

    destroyImageViews();
    destroySwapchain();

    destroyLogicalDevice();
    destroySurface();
    destroyInstance();
}
```

recordCommandBuffers()

```
vkCmdBeginRenderPass(mCommandBuffers[i], &renderPassInfo, VK_SUBPASS_CONTENTS_INLINE);

    vkCmdBindPipeline(mCommandBuffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS, mPipeline);

    // binding the vertex buffer
    VkBuffer vertexBuffers[] = {mVertexBuffer};
    VkDeviceSize offsets[] = {0};
    vkCmdBindVertexBuffers( mCommandBuffers[i], 0, 1, vertexBuffers, offsets );

    // drawing using vertex buffer
    vkCmdDraw( mCommandBuffers[i], gVertices.size(), 1, 0, 0 );

vkCmdEndRenderPass(mCommandBuffers[i]);
```

Staging Buffer

Memory Optimization

- CPU-accessible vertex buffer is not an optimized memory type
- Optimized memory type need to have

VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT flag (potentially not CPU accessible)

Use the Staging Buffer

1) Create the staging buffer (GPU memory, CPU accessible)
2) Copy vertex data to staging buffer
3) Copy staging buffer to final vertex buffer (GPU memory, CPU not accessible)

createBuffer() / destroyBuffer()

```
void createBuffer(VkPhysicalDevice gpu, VkDevice device, VkDeviceSize size,
    VkBufferUsageFlags usage, VkMemoryPropertyFlags properties, VkBuffer& buffer, VkDeviceMemory& bufferMemory)
{
    // create vertex buffer (GPU buffer)
    VkBufferCreateInfo bufferInfo = {};
    bufferInfo.sType        = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    bufferInfo.size         = size;
    bufferInfo.usage        = usage;
    bufferInfo.sharingMode  = VK_SHARING_MODE_EXCLUSIVE;    // used by graphics queue only

    checkError( vkCreateBuffer(device, &bufferInfo, nullptr, &buffer) );

    // retrieve memory rquirement of vertex buffer
    VkMemoryRequirements memRequirements;
    vkGetBufferMemoryRequirements(device, buffer, &memRequirements);

    // allocate memory for vertext buffer
    VkMemoryAllocateInfo allocInfo = {};
    allocInfo.sType             = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    allocInfo.allocationSize    = memRequirements.size;
    allocInfo.memoryTypeIndex   = findMemoryType(gpu, memRequirements.memoryTypeBits, properties);

    checkError( vkAllocateMemory(device, &allocInfo, nullptr, &bufferMemory) );

    // binding memory with vertex buffer
    checkError( vkBindBufferMemory(device, buffer, bufferMemory, 0) );
}
```

```
void destroyBuffer(VkDevice device, VkBuffer buffer, VkDeviceMemory bufferMemory)
{
    vkFreeMemory(device, bufferMemory, nullptr);
    vkDestroyBuffer(device, buffer, nullptr);
}
```


Vulkan

copyBuffer()

```
void copyBuffer(VkDevice device, VkCommandPool cmdPool, VkQueue queue, VkBuffer srcBuffer, VkBuffer dstBuffer, VkDeviceSize size)
{
    // command buffer allocation
    VkCommandBufferAllocateInfo allocInfo = {};
    allocInfo.sType                 = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
    allocInfo.level                 = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
    allocInfo.commandPool           = cmdPool;
    allocInfo.commandBufferCount    = 1;

    VkCommandBuffer commandBuffer;
    checkError( vkAllocateCommandBuffers(device, &allocInfo, &commandBuffer) );

    VkCommandBufferBeginInfo beginInfo = {};
    beginInfo.sType             = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
    beginInfo.flags             = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;

    checkError( vkBeginCommandBuffer(commandBuffer, &beginInfo) );

        VkBufferCopy copyRegion = {};
        copyRegion.srcOffset    = 0;
        copyRegion.dstOffset    = 0;
        copyRegion.size         = size;
        vkCmdCopyBuffer(commandBuffer, srcBuffer, dstBuffer, 1, &copyRegion);

    checkError( vkEndCommandBuffer(commandBuffer) );

    VkSubmitInfo submitInfo = {};
    submitInfo.sType                = VK_STRUCTURE_TYPE_SUBMIT_INFO;
    submitInfo.commandBufferCount   = 1;
    submitInfo.pCommandBuffers      = &commandBuffer;

    checkError( vkQueueSubmit(queue, 1, &submitInfo, VK_NULL_HANDLE) );
    checkError( vkQueueWaitIdle(queue) );

    vkFreeCommandBuffers(device, cmdPool, 1, &commandBuffer);
}
```

Create a one time command buffer

Copy the buffer

Submit a command buffer/
Free the command buffer

Vulkan

createVertexBuffer()

```
class VulkanRenderer
{
private:
    VkBuffer          mVertexBuffer;
    VkDeviceMemory    mVertexBufferMem;

    void createVertexBuffer();
    void destroyVertexBuffer();
};
```

```
void VulkanRenderer::createVertexBuffer()
{
    VkDeviceSize bufferSize = sizeof(gVertices[0]) * gVertices.size();

    // 1. staging buffer (GPU mem : CPU accessible)
    VkBuffer stagingBuffer;
    VkDeviceMemory stagingBufferMemory;
    createBuffer(
        mGpu, mDevice,
        bufferSize,
        VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
        VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
        stagingBuffer,
        stagingBufferMemory
    );

    // 2. copy vertex data(CPU mem) to the staging buffer(temporary buffer)
    void* data;
    checkError( vkMapMemory(mDevice, stagingBufferMemory, 0, bufferSize, 0, &data) );
        memcpy(data, gVertices.data(), (size_t)bufferSize);
    vkUnmapMemory(mDevice, stagingBufferMemory);

    // 3. local device buffer (GPU mem : CPU not accessible)
    createBuffer(
        mGpu, mDevice,
        bufferSize,
        VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_VERTEX_BUFFER_BIT,
        VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
        mVertexBuffer,
        mVertexBufferMem
    );

    // 4. copy staging buffer to local buffer
    copyBuffer( mDevice, mCommandPool, mGraphicsQueue, stagingBuffer, mVertexBuffer, bufferSize );

    destroyBuffer( mDevice, stagingBuffer, stagingBufferMemory );
}
```

Index Buffer

Index Data

- If drawing with vertex data only, there can be a lot of vertex data duplication
- The solution is to use index data

V0 V1
V5
V4 V3 V2

Vertex buffer = { (V0, V1, V2), (V3, V4, V5) }

V0 V1
V3 V2

Vertex buffer = { V0, V1, V2, V3 }

Index buffer = { (0, 1, 2), (2, 3, 0) }

Define the Index data

```
// array of vertex data
const std::vector<Vertex> gVertices = {
    { {-0.5f, -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} },
    { {-0.5f, 0.5f}, {1.0f, 1.0f, 1.0f} }
};

// array of index data
const std::vector<uint16_t> gIndices = {
    0, 1, 2, 2, 3, 0
};
```

Index Buffer

Create

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();

    createRenderPass();
    createFramebuffers();

    createCommandPool();
    createCommandBuffers();
    createDescriptorSetLayout();
    createGraphicsPipeline();

    createVertexBuffer();
    createIndexBuffer();

    recordCommandBuffers();
    createSemaphores();
}
```

Destroy

```
VulkanRenderer::~VulkanRenderer()
{
    destroySemaphores();

    destroyIndexBuffer();
    destroyVertexBuffer();

    destroyGraphicsPipeline();
    destroyDescriptorSetLayout();
    destroyCommandPool();

    destroyFramebuffers();
    destroyRenderPass();

    destroyImageViews();
    destroySwapchain();

    destroyLogicalDevice();
    destroySurface();
    destroyInstance();
}
```

createIndexBuffer()

```
class VulkanRenderer
{
private:
    VkBuffer        mIndexBuffer;
    VkDeviceMemory  mIndexBufferMem;

    void createIndexBuffer();
    void destroyIndexBuffer();
};
```

```
void VulkanRenderer::createIndexBuffer()
{
    VkDeviceSize bufferSize = sizeof(gIndices[0]) * gIndices.size();

    // staging buffer (CPU accessible)
    VkBuffer stagingBuffer;
    VkDeviceMemory stagingBufferMemory;
    createBuffer(
        mGpu, mDevice,
        bufferSize,
        VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
        VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
        stagingBuffer,
        stagingBufferMemory
    );

    // copy vertex data(CPU mem) to the staging buffer(temporary buffer)
    void* data;
    checkError( vkMapMemory(mDevice, stagingBufferMemory, 0, bufferSize, 0, &data) );
        memcpy(data, gIndices.data(), (size_t)bufferSize);
    vkUnmapMemory(mDevice, stagingBufferMemory);

    // local device buffer (GPU)
    createBuffer(
        mGpu, mDevice,
        bufferSize,
        VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_VERTEX_BUFFER_BIT,
        VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
        mIndexBuffer,
        mIndexBufferMem
    );

    copyBuffer( mDevice, mCommandPool, mGraphicsQueue, stagingBuffer, mIndexBuffer, bufferSize );

    destroyBuffer( mDevice, stagingBuffer, stagingBufferMemory);
}
```

destroyIndexBuffer()

```
class VulkanRenderer
{
private:
    VkBuffer        mIndexBuffer;
    VkDeviceMemory  mIndexBufferMem;

    void createIndexBuffer();
    void destroyIndexBuffer();
};
```

```
void VulkanRenderer::destroyIndexBuffer()
{
    destroyBuffer(mDevice, mIndexBuffer, mIndexBufferMem);
}
```

Drawing using the Index Buffer

Create

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();

    createRenderPass();
    createFramebuffers();

    createCommandPool();
    createCommandBuffers();
    createDescriptorSetLayout();
    createGraphicsPipeline();

    createVertexBuffer();
    createIndexBuffer();

    recordCommandBuffers();
    createSemaphores();
}
```

Destroy

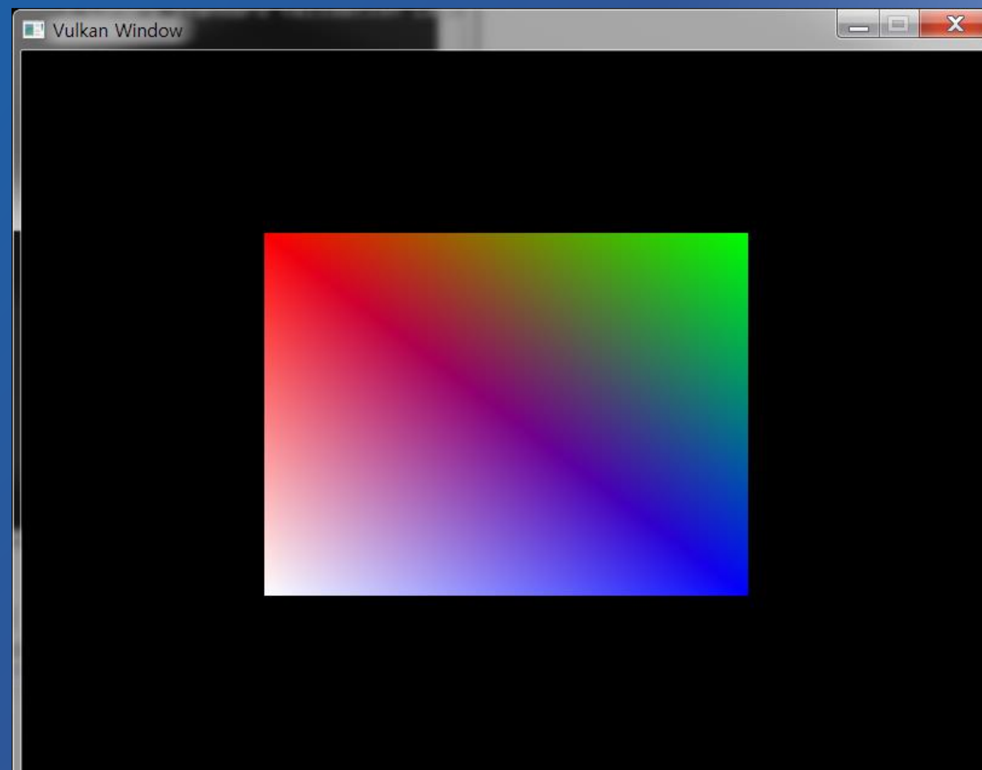
```
VulkanRenderer::~VulkanRenderer()
{
    destroySemaphores();

    destroyIndexBuffer();
    destroyVertexBuffer();

    destroyGraphicsPipeline();
    destroyDescriptorSetLayout();
    destroyCommandPool();

    destroyFramebuffers();
    destroyRenderPass();

    destroyImageViews();
    destroySwapchain();

    destroyLogicalDevice();
    destroySurface();
    destroyInstance();
}
```

recordCommandBuffers()

```
vkCmdBeginRenderPass(mCommandBuffers[i], &renderPassInfo, VK_SUBPASS_CONTENTS_INLINE);

    vkCmdBindPipeline(mCommandBuffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS, mPipeline);

    // binding the vertex buffer
    VkBuffer vertexBuffers[] = {mVertexBuffer};
    VkDeviceSize offsets[] = {0};
    vkCmdBindVertexBuffers( mCommandBuffers[i], 0, 1, vertexBuffers, offsets );

    // binidng the index buffer
    vkCmdBindIndexBuffer( mCommandBuffers[i], mIndexBuffer, 0, VK_INDEX_TYPE_UINT16 );

    // Drawing triangle using vertex buffer
    //vkCmdDraw( _command_buffers[i], gVertices.size(), 1, 0, 0 );

    // Drawing rectangle using index buffer
    vkCmdDrawIndexed( mCommandBuffers[i], (uint32_t)gIndices.size(), 1, 0, 0, 0 );

vkCmdEndRenderPass(mCommandBuffers[i]);
```

Drawing a Rectangle

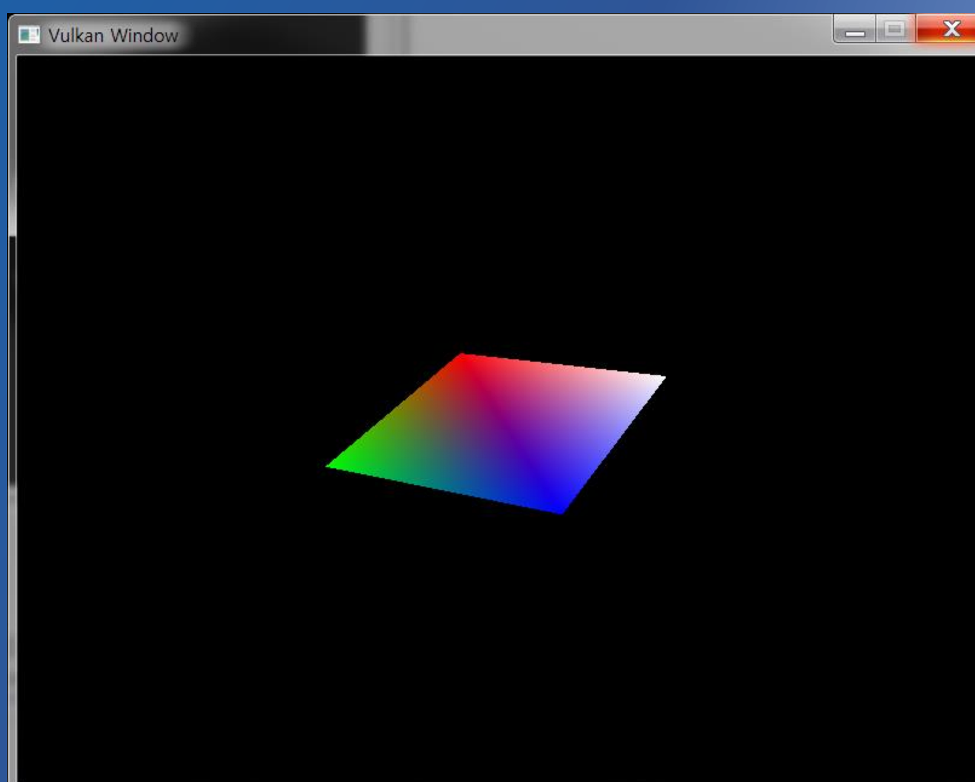
- Vertex Buffer
- Staging Buffer
- Index Buffer



Vulkan.

Rotation and 3D Projection

- Resource Descriptor
- Descriptor Set / Descriptor Set Layout
- Uniform Buffer Object (UBO)



Resource Descriptor

Resource Descriptor

- Specify the resource in vulkan
- Shader can access resources(buffer, image, etc.) using resource descriptor

Typical Resource Descriptor

- UBO (Uniform Buffer Object)
  : Update values in rendering time without modifying the shader
  e.g.) transformation matrix(vertex shader)
- Texture image

Descriptor Set / Descriptor Set Layout

Descriptor Set Layout

- Specify resource type to access in pipeline

 (binding number, pipeline stage information, etc.)

Descriptor Pool

- Have the number of descriptor

- Descriptor sets are created from a descriptor pool

Descriptor Set

- Specify the actual resources bound to the resource descriptor

- Descriptor sets are created from a descriptor set layout and descriptor pool

Descriptor Set / Descriptor Set Layout

Using Descriptor

1) Before graphics pipeline creation : Create the descriptor set layout
2) Graphics pipeline creation time : Specify the descriptor set layout
3) After graphics pipeline creation : Create the uniform buffer object , descriptor pool and descriptor set
4) Rendering time : Descriptor binding

Resource Descriptor

Create

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();

    createRenderPass();
    createFramebuffers();

    createCommandPool();
    createCommandBuffers();
    createDescriptorSetLayout();
    createGraphicsPipeline();

    createVertexBuffer();
    createIndexBuffer();

    createStagingUniformBuffer();
    createUniformBuffer();
    createDescriptorPool();
    createDescriptorSet();

    recordCommandBuffers();
    createSemaphores();
}
```

- createDescriptorSetLayout() → Create the descriptor set layout
- createGraphicsPipeline() → Specify the descriptor set layout
- createStagingUniformBuffer() … createDescriptorSet() → Create the Uniform Buffer Object; Create the Descriptor pool/descriptor set
- recordCommandBuffers() → Bind the descriptor set

Update the Vertex Shader

Vertex Shader

- Add uniform in vertex shader (MVP matrix)
- Apply MVP transformation in gl_Position

  (Model-View-Projection : Rotation and 3D projection)

Vertex Shader (shader.vert)

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(set = 0, binding = 0) uniform UniformBufferObject {
    mat4 model;
    mat4 view;
    mat4 proj;
} ubo;

layout(location = 0) in vec2 inPosition;
layout(location = 1) in vec3 inColor;

layout(location = 0) out vec3 fragColor;

out gl_PerVertex {
    vec4    gl_Position;
};

void main() {
    gl_Position =  ubo.proj * ubo.view * ubo.model * vec4(inPosition, 0.0, 1.0);
    fragColor = inColor;
}
```

Uniform (MVP matrix)

(binding = 0) is used as index in descriptor set layout

MVP transformation

Vulkan

Define the Uniform Data

Uniform Data

- User GLM library to use uniform data (matrix type) in vertex shader

Use the Uniform data

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

struct UniformBufferObject {
    glm::mat4 model;
    glm::mat4 view;
    glm::mat4 proj;
};
```

Create the Descriptor Set Layout

Descriptor Set Layout

- Meta-information of resources

e.g.) binding information, resource type(uniform, texture, etc.), usage stage (pipeline stage)

Descriptor Set Layout Creation

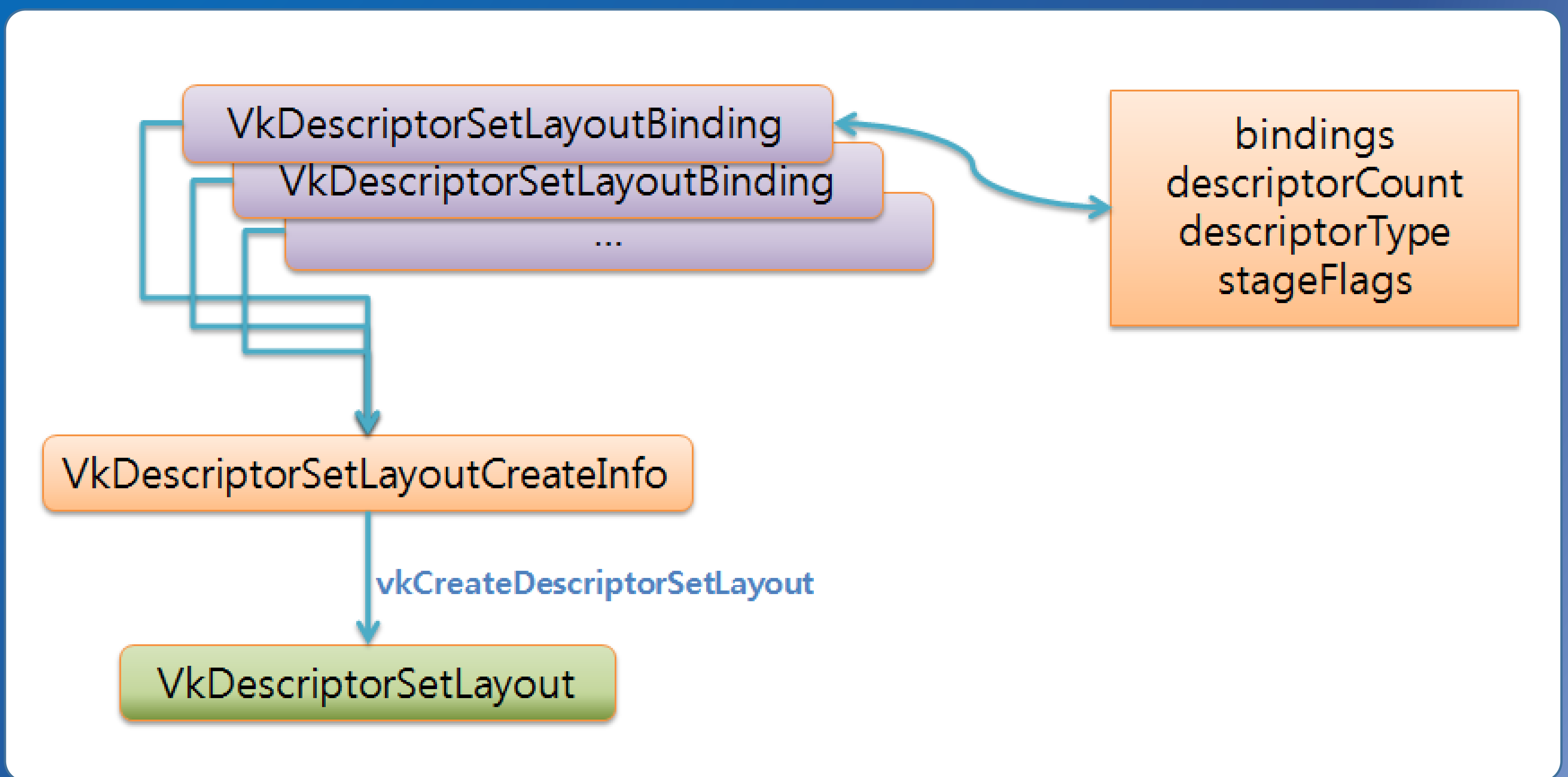
1) Set the descriptor set layout binding information for each resource type (VkDescriptorSetLayoutBinding)

2) Set descriptor set layout create information to bind more than one descriptor set layout binding information (VkDescriptorSetLayoutCreateInfo)

3) Create the descriptor set layout

4) Set the descriptor set layout information in the graphics pipeline

Create the Descriptor Set Layout



Create the Descriptor Set Layout

Create

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();

    createRenderPass();
    createFramebuffers();

    createCommandPool();
    createCommandBuffers();
    createDescriptorSetLayout();
    createGraphicsPipeline();

    createVertexBuffer();
    createIndexBuffer();

    createStagingUniformBuffer();
    createUniformBuffer();
    createDescriptorPool();
    createDescriptorSet();

    recordCommandBuffers();
    createSemaphores();
}
```

Destroy

```
VulkanRenderer::~VulkanRenderer()
{
    destroySemaphores();

    destroyDescriptorPool();
    destoryUniformBuffer();
    destroyStagingUniformBuffer();

    destroyIndexBuffer();
    destroyVertexBuffer();

    destroyGraphicsPipeline();
    destroyDescriptorSetLayout();
    destroyCommandPool();

    destroyFramebuffers();
    destroyRenderPass();

    destroyImageViews();
    destroySwapchain();

    destroyLogicalDevice();
    destroySurface();
    destroyInstance();
}
```

createDescriptorSetLayout()
destroyDescriptorSetLayout()

```
class VulkanRenderer
{
private:
    VkDescriptorSetLayout   mDescriptorSetLayout;

    void createDescriptorSetLayout();
    void destroyDescriptorSetLayout();
};
```

```
void VulkanRenderer::createDescriptorSetLayout() {
    VkDescriptorSetLayoutBinding mvpLayoutBinding = {};
    mvpLayoutBinding.binding = 0;                                          // binding number in shader stages
    mvpLayoutBinding.descriptorCount = 1;                                  // number of dectiptors containded
    mvpLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;   // VkDescriptorType which is blow
    mvpLayoutBinding.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;              // which pipeline shader stages can access

    VkDescriptorSetLayoutCreateInfo createInfo = {};
    createInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
    createInfo.bindingCount = 1;                                           // count of VkDescriptorSetLayoutBindings
    createInfo.pBindings = &mvpLayoutBinding;                              // VkDescriptorSetLayoutBinding pointer

    checkError( vkCreateDescriptorSetLayout(mDevice, &createInfo, nullptr, &mDescriptorSetLayout) );
}

void VulkanRenderer::destroyDescriptorSetLayout() {
    vkDestroyDescriptorSetLayout(mDevice, mDescriptorSetLayout, nullptr);
}
```

Specify the Descriptor Set Layout

Create

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();

    createRenderPass();
    createFramebuffers();

    createCommandPool();
    createCommandBuffers();
    createDescriptorSetLayout();
    createGraphicsPipeline();

    createVertexBuffer();
    createIndexBuffer();

    createStagingUniformBuffer();
    createUniformBuffer();
    createDescriptorPool();
    createDescriptorSet();

    recordCommandBuffers();
    createSemaphores();
}
```

Destroy

```
VulkanRenderer::~VulkanRenderer()
{
    destroySemaphores();

    destroyDescriptorPool();
    destoryUniformBuffer();
    destroyStagingUniformBuffer();

    destroyIndexBuffer();
    destroyVertexBuffer();

    destroyGraphicsPipeline();
    destroyDescriptorSetLayout();
    destroyCommandPool();

    destroyFramebuffers();
    destroyRenderPass();

    destroyImageViews();
    destroySwapchain();

    destroyLogicalDevice();
    destroySurface();
    destroyInstance();
}
```

createGraphicsPipeline()

```
class VulkanRenderer
{
private:
    VkPipelineLayout        mPipelineLayout;
    VkPipeline              mPipeline;
    VkDescriptorSetLayout   mDescriptorSetLayout;

    void createGraphicsPipeline();
};
```

```
// Pipeline layout for passing uniform values to shaders
VkPipelineLayoutCreateInfo pipelineLayoutInfo = {};
{
    pipelineLayoutInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
    pipelineLayoutInfo.setLayoutCount = 1;
    pipelineLayoutInfo.pSetLayouts = &mDescriptorSetLayout;
    pipelineLayoutInfo.pushConstantRangeCount = 0;
    pipelineLayoutInfo.pPushConstantRanges = 0;

    checkError(vkCreatePipelineLayout(mDevice, &pipelineLayoutInfo, nullptr, &mPipelineLayout));
}

VkGraphicsPipelineCreateInfo pipelineInfo = {};
// ...
pipelineInfo.layout = mPipelineLayout;
// ...

checkError(vkCreateGraphicsPipelines(mDevice, VK_NULL_HANDLE, 1, &pipelineInfo, nullptr, &mPipeline));
```

Create the Uniform Buffer Object

Uniform Buffer Object (UBO)

- Uniform information (MVP transformation) can be updated every frame
- Use the Vulkan buffer (VkBuffer) to deliver uniform information from CPU to GPU
- Uniform buffer creation process is similar to vertex buffer creation

▸ In every frame, copy uniform data to the UBO so the shader can access it

Create the Uniform Buffer Object

Create

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();

    createRenderPass();
    createFramebuffers();

    createCommandPool();
    createCommandBuffers();
    createDescriptorSetLayout();
    createGraphicsPipeline();

    createVertexBuffer();
    createIndexBuffer();

    createStagingUniformBuffer();
    createUniformBuffer();
    createDescriptorPool();
    createDescriptorSet();

    recordCommandBuffers();
    createSemaphores();
}
```

Destroy

```
VulkanRenderer::~VulkanRenderer()
{
    destroySemaphores();

    destroyDescriptorPool();
    destoryUniformBuffer();
    destroyStagingUniformBuffer();

    destroyIndexBuffer();
    destroyVertexBuffer();

    destroyGraphicsPipeline();
    destroyDescriptorSetLayout();
    destroyCommandPool();

    destroyFramebuffers();
    destroyRenderPass();

    destroyImageViews();
    destroySwapchain();

    destroyLogicalDevice();
    destroySurface();
    destroyInstance();
}
```

createStagingUniformBuffer()
destroyStagingUniformBuffer()

```
class VulkanRenderer
{
    VkBuffer          mStagingUniformBuffer;
    VkDeviceMemory    mStagingUniformBufferMem;
    VkBuffer          mUniformBuffer;
    VkDeviceMemory    mUniformBufferMem;

    void createStagingUniformBuffer();
    void destroyStagingUniformBuffer();
    void createUniformBuffer();
    void destoryUniformBuffer();
    void updateUniformBuffer();
};
```

```
void VulkanRenderer::createStagingUniformBuffer() {
    VkDeviceSize bufferSize = sizeof(UniformBufferObject);

    // staging buffer (GPU mem : CPU accessible)
    createBuffer(
        mGpu, mDevice,
        bufferSize,
        VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
        VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
        mStagingUniformBuffer,
        mStagingUniformBufferMem
    );
}

void VulkanRenderer::destroyStagingUniformBuffer() {
    destroyBuffer(mDevice, mStagingUniformBuffer, mStagingUniformBufferMem);
}
```

createUniformBuffer()
destroyUniformBuffer()

```
class VulkanRenderer
{
    VkBuffer         mStagingUniformBuffer;
    VkDeviceMemory   mStagingUniformBufferMem;
    VkBuffer         mUniformBuffer;
    VkDeviceMemory   mUniformBufferMem;

    void createStagingUniformBuffer();
    void destroyStagingUniformBuffer();
    void createUniformBuffer();
    void destoryUniformBuffer();
    void updateUniformBuffer();
};
```

```
void VulkanRenderer::createUniformBuffer() {
    VkDeviceSize bufferSize = sizeof(UniformBufferObject);

    // local device buffer (GPU mem : CPU not accessible)
    createBuffer(
        mGpu, mDevice,
        bufferSize,
        VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT,
        VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
        mUniformBuffer,
        mUniformBufferMem
    );
}

void VulkanRenderer::destoryUniformBuffer() {
    destroyBuffer(mDevice, mUniformBuffer, mUniformBufferMem);
}
```


Vulkan

updateUniformBuffer()

Call this every frame before calling the drawFrame()

```
class VulkanRenderer
{
    VkBuffer         mStagingUniformBuffer;
    VkDeviceMemory   mStagingUniformBufferMem;
    VkBuffer         mUniformBuffer;
    VkDeviceMemory   mUniformBufferMem;

    void createStagingUniformBuffer();
    void destroyStagingUniformBuffer();
    void createUniformBuffer();
    void destoryUniformBuffer();
    void updateUniformBuffer();
};
```

```
void VulkanRenderer::updateUniformBuffer() {
    VkDeviceSize bufferSize = sizeof(UniformBufferObject);

    static float angle = 0.0f;
    angle += 0.05f;

    UniformBufferObject ubo = {};
    ubo.model = glm::rotate(glm::mat4(), angle / 180.0f * 3.14f * 0.5f, glm::vec3(0.6f, 0.0f, 1.0f));
    ubo.view = glm::lookAt(
                glm::vec3(2.0f, 2.0f, 2.0f),    // eye position
                glm::vec3(0.0f, 0.0f, 0.0f),    // center position
                glm::vec3(0.0f, 0.0f, 1.0f)     // up vector
    );
    ubo.proj = glm::perspective(
                glm::radians(45.0f),
                mSwapchainExtent.width / (float)mSwapchainExtent.height,    // aspect ratio
                0.1f,    // near plane
                10.0f    // far plane
    );
    ubo.proj[1][1] *= -1;    // invert Y-coordinate (OpenGL -> Vulkan)

    // copy vertex data(CPU mem) to the staging buffer(temporary buffer)
    void* data;
    checkError( vkMapMemory(mDevice, mStagingUniformBufferMem, 0, bufferSize, 0, &data) );
    memcpy(data, &ubo, bufferSize);
    vkUnmapMemory(mDevice, mStagingUniformBufferMem);

    // copy staging buffer to local buffer
    copyBuffer(mDevice, mCommandPool, mGraphicsQueue, mStagingUniformBuffer, mUniformBuffer, bufferSize);
}
```

createGraphicsPipeline()

```
// 6-2. Rasterizer stage[Fixed]
VkPipelineRasterizationStateCreateInfo rasterizer = {};
{
    rasterizer.sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
    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.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
    rasterizer.depthBiasEnable = VK_FALSE;
    rasterizer.depthBiasConstantFactor = 0.0f;          // optional
    rasterizer.depthBiasClamp = 0.0f;           // optional
    rasterizer.depthBiasSlopeFactor = 0.0f;           // optional
}
```

Invert Y-coordinate

Vulkan

Create the Descriptor Pool

Descriptor Pool

- Create the descriptor pool prior to creating the descriptor set
- Descriptor pools are created for each resource type with some information (descriptor type, the number of descriptors, etc.)

Vulkan.

Create the Descriptor Pool

Create

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();

    createRenderPass();
    createFramebuffers();

    createCommandPool();
    createCommandBuffers();
    createDescriptorSetLayout();
    createGraphicsPipeline();

    createVertexBuffer();
    createIndexBuffer();

    createStagingUniformBuffer();
    createUniformBuffer();
    createDescriptorPool();
    createDescriptorSet();

    recordCommandBuffers();
    createSemaphores();
}
```

Destroy

```
VulkanRenderer::~VulkanRenderer()
{
    destroySemaphores();

    destroyDescriptorPool();
    destoryUniformBuffer();
    destroyStagingUniformBuffer();

    destroyIndexBuffer();
    destroyVertexBuffer();

    destroyGraphicsPipeline();
    destroyDescriptorSetLayout();
    destroyCommandPool();

    destroyFramebuffers();
    destroyRenderPass();

    destroyImageViews();
    destroySwapchain();

    destroyLogicalDevice();
    destroySurface();
    destroyInstance();
}
```

createDescriptorPool()
destroyDescriptorPool()

```
class VulkanRenderer
{
private:
    VkDescriptorPool            mDescriptorPool;

    void createDescriptorPool();
    void destroyDescriptorPool();
};
```

```
void VulkanRenderer::createDescriptorPool() {
    VkDescriptorPoolSize descriptorPoolSize = {};
    descriptorPoolSize.type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;    // descritor type
    descriptorPoolSize.descriptorCount = 1;                         // descritor count

    VkDescriptorPoolCreateInfo createInfo = {};
    createInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
    createInfo.poolSizeCount = 1;                      // count of VkDescriptorPoolSize
    createInfo.pPoolSizes = &descriptorPoolSize;       // VkDescriptorPoolSize pointer
    createInfo.maxSets = 1;                            // maximum number of descriptorset, which be allocated in pool

    checkError( vkCreateDescriptorPool(mDevice, &createInfo, nullptr, &mDescriptorPool) );
}

void VulkanRenderer::destroyDescriptorPool() {
    vkDestroyDescriptorPool(mDevice, mDescriptorPool, nullptr);
}
```

Create the Descriptor Set

Descriptor Set

- The descriptor set is created from the descriptor set layout and descriptor pool
- Allocate the buffer after creating the descriptor set

VkDescriptorPool → VkDescriptorSetAllocateInfo ← VkDescriptorSetLayout

VkDescriptorSetAllocateInfo → (vkAllocateDescriptorSets) → VkDescriptorSet

VkBuffer → VkDescriptorBufferInfo

VkDescriptorBufferInfo + VkDescriptorSet → (vkUpdateDescriptorSets) → VkWriteDescriptorSet

Create the Descriptor Set

Create

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();

    createRenderPass();
    createFramebuffers();

    createCommandPool();
    createCommandBuffers();
    createDescriptorSetLayout();
    createGraphicsPipeline();

    createVertexBuffer();
    createIndexBuffer();

    createStagingUniformBuffer();
    createUniformBuffer();
    createDescriptorPool();
    createDescriptorSet();

    recordCommandBuffers();
    createSemaphores();
}
```

Destroy

```
VulkanRenderer::~VulkanRenderer()
{
    destroySemaphores();

    destroyDescriptorPool();
    destoryUniformBuffer();
    destroyStagingUniformBuffer();

    destroyIndexBuffer();
    destroyVertexBuffer();

    destroyGraphicsPipeline();
    destroyDescriptorSetLayout();
    destroyCommandPool();

    destroyFramebuffers();
    destroyRenderPass();

    destroyImageViews();
    destroySwapchain();

    destroyLogicalDevice();
    destroySurface();
    destroyInstance();
}
```

※ Descriptor sets are destroyed when destroying the descriptor pool

createDescriptorSet()

```
class VulkanRenderer
{
private:
    VkBuffer                mUniformBuffer;
    VkDescriptorSetLayout   mDescriptorSetLayout;
    VkDescriptorPool        mDescriptorPool;
    VkDescriptorSet         mDescriptorSet;

    void createDescriptorSet();
};
```

```
void VulkanRenderer::createDescriptorSet() {
    VkDescriptorSetAllocateInfo allocateInfo = {};
    allocateInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
    allocateInfo.descriptorPool = mDescriptorPool;                    // VkDescriptorPool
    allocateInfo.descriptorSetCount = 1;                              // count of VkDescriptorSet
    allocateInfo.pSetLayouts = &mDescriptorSetLayout;                 // VkDescriptorSetLayout pointer

    checkError(vkAllocateDescriptorSets(mDevice, &allocateInfo, &mDescriptorSet));

    VkDescriptorBufferInfo bufferInfo = {};
    bufferInfo.buffer = mUniformBuffer;                   // VkBuffer
    bufferInfo.offset = 0;                                // VkBuffer Offset
    bufferInfo.range = sizeof(UniformBufferObject);       // VkBuffer Size

    VkWriteDescriptorSet writeDescriptorSet = {};
    writeDescriptorSet.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
    writeDescriptorSet.dstSet = mDescriptorSet;                                // VkDescriptorSet
    writeDescriptorSet.dstBinding = 0;                                         // binding number in shader
    writeDescriptorSet.dstArrayElement = 0;                                    // start element in array
    writeDescriptorSet.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;     // descriptor type
    writeDescriptorSet.descriptorCount = 1;                                    // descriptor count
    writeDescriptorSet.pBufferInfo = &bufferInfo;                              // VkDescritporBufferInfo

    vkUpdateDescriptorSets(mDevice, 1, &writeDescriptorSet, 0, nullptr);
}
```

Descriptor Set Binding

Descriptor Set Binding

- Bind the descriptor set at rendering time
- Submit binding command in the command buffer, before submitting drawing commands

Bind the Descriptor Set

Create

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();

    createRenderPass();
    createFramebuffers();

    createCommandPool();
    createCommandBuffers();
    createDescriptorSetLayout();
    createGraphicsPipeline();

    createVertexBuffer();
    createIndexBuffer();

    createStagingUniformBuffer();
    createUniformBuffer();
    createDescriptorPool();
    createDescriptorSet();

    recordCommandBuffers();
    createSemaphores();
}
```

Destroy

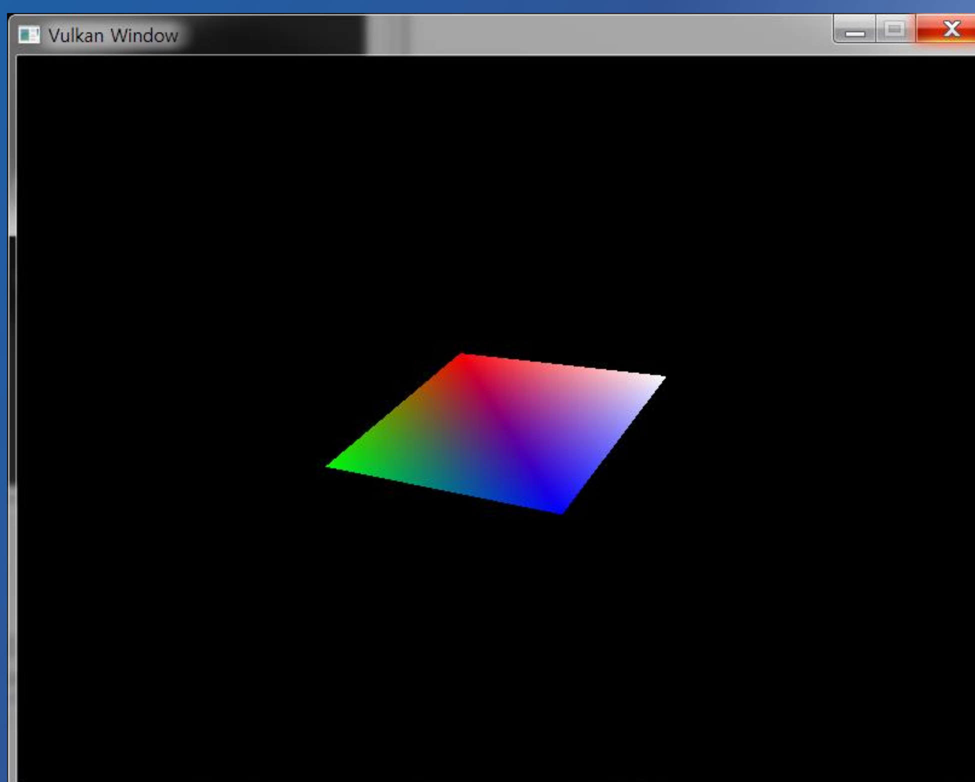
```
VulkanRenderer::~VulkanRenderer()
{
    destroySemaphores();

    destroyDescriptorPool();
    destoryUniformBuffer();
    destroyStagingUniformBuffer();

    destroyIndexBuffer();
    destroyVertexBuffer();

    destroyGraphicsPipeline();
    destroyDescriptorSetLayout();
    destroyCommandPool();

    destroyFramebuffers();
    destroyRenderPass();

    destroyImageViews();
    destroySwapchain();

    destroyLogicalDevice();
    destroySurface();
    destroyInstance();
}
```

recordCommandBuffers()

```
vkCmdBeginRenderPass(mCommandBuffers[i], &renderPassInfo, VK_SUBPASS_CONTENTS_INLINE);

    vkCmdBindPipeline(mCommandBuffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS, mPipeline);

    // binding the vertex buffer
    VkBuffer vertexBuffers[] = {mVertexBuffer};
    VkDeviceSize offsets[] = {0};
    vkCmdBindVertexBuffers( mCommandBuffers[i], 0, 1, vertexBuffers, offsets );

    // binidng the index buffer
    vkCmdBindIndexBuffer( mCommandBuffers[i], mIndexBuffer, 0, VK_INDEX_TYPE_UINT16 );

    vkCmdBindDescriptorSets(
        mCommandBuffers[i],
        VK_PIPELINE_BIND_POINT_GRAPHICS,
        mPipelineLayout,
        0,                  // first set
        1,                  // descriptor set count
        &mDescriptorSet,
        0,                  // dynamic offset count
        nullptr             // dynamic offset
    );

    // Drawing rectangle using index buffer
    vkCmdDrawIndexed( mCommandBuffers[i], (uint32_t)gIndices.size(), 1, 0, 0, 0 );

vkCmdEndRenderPass(mCommandBuffers[i]);
```

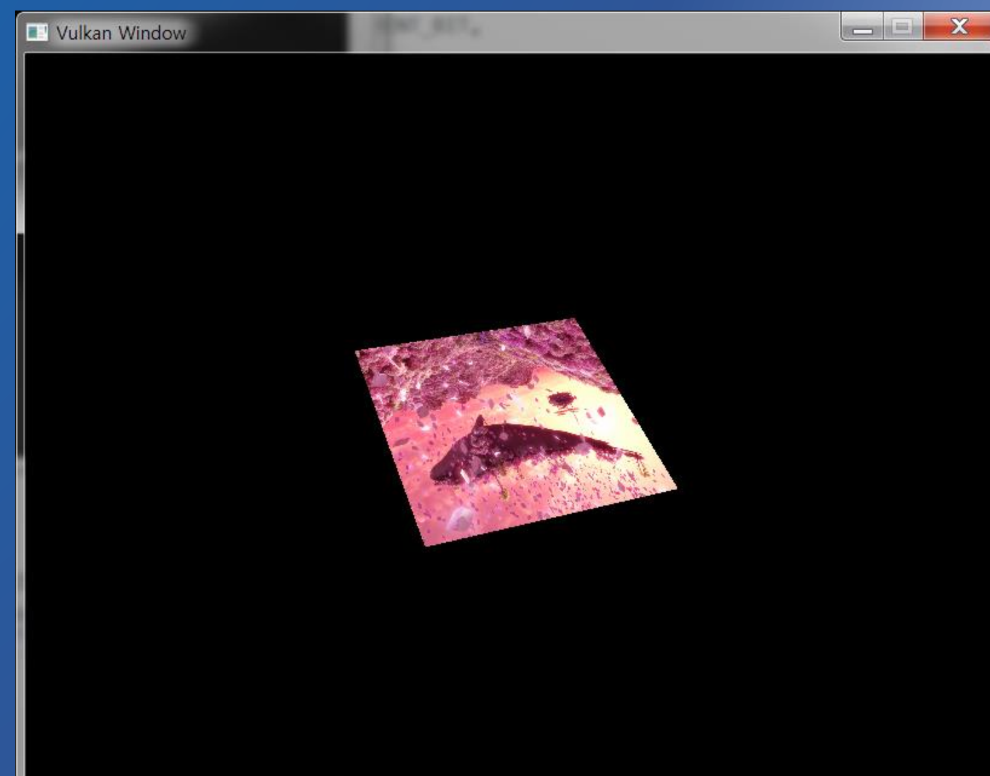
Rotation and 3D Projection

- Resource Descriptor
- Descriptor Set / Descriptor Set Layout
- Uniform Buffer Object (UBO)



Texture Mapping

- Texture Image
- Sampler
- Texture Descriptor Set



Update the Fragment Shader

Fragment Shader

- Fragment shader gets per-fragment data from the program
- Deliver sampler and texture coordinates to shader

Vulkan.

Fragment Shader (shader.frag)

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(location = 0) in vec3 fragColor;
layout(location = 1) in vec2 fragTexCoord;

layout(set = 0, binding = 1) uniform sampler2D texSampler;

layout(location = 0) out vec4 outColor;

void main() {
    //outColor = vec4(fragColor, 1.0);
    outColor = texture(texSampler, fragTexCoord);
}
```

Define the Texture Data

Vertex Data

- Add texture coordinates on vertex data

Define the Texture data

```
// array of vertex data
// { {x, y}, {r, g, b}, {coordination_x, coordination_y} }
const std::vector<Vertex> gVertices = {
    { {-0.5f, -0.5f}, {1.0f, 0.0f, 0.0f}, {0.0f, 0.0f} },
    { {0.5f, -0.5f},  {0.0f, 1.0f, 0.0f}, {1.0f, 0.0f} },
    { {0.5f, 0.5f},   {0.0f, 0.0f, 1.0f}, {1.0f, 1.0f} },
    { {-0.5f, 0.5f},  {1.0f, 1.0f, 1.0f}, {0.0f, 1.0f} }
};
```


Vulkan.

Texture Image

Create the Texture Image

- Texture image process is similar with vertex buffer creation
  : VkImage → Image object handle
  : VkDeviceMemory → Memory object that has actual image data

Texture Image Creation Process

1) Read image data from image file
2) Create the texture image object
3) Copy image data to the texture image object

Texture Image

Texture Image vs. Vertex buffer

Texture image creation	Vertex buffer creation
Create Staging VkImage object	Create Staging VkBuffer object
Query VkMemoryRequirements	Query VkMemoryRequirements
Allocate VkDeviceMemory	Allocate VkDeviceMemory
Bind VkDeviceMemory to VkImage	Bind VkDeviceMemory to VkBuffer
Map VkDeviceMemory to void* type data	Map VkDeviceMemory to void* type data
Copy Texture Image data using memcpy	Copy Vertex data using memcpy
Copy Staging VkImage to actual local VkImage	Copy Staging VkBuffer to actual local VkBuffer

createImage()

```
void createImage(VkPhysicalDevice gpu, VkDevice device, uint32_t width, uint32_t height, VkFormat format, VkImageTiling tiling,
    VkImageUsageFlags usage, VkMemoryPropertyFlags properties, VkImage& image, VkDeviceMemory& imageMemory)
{
    VkImageCreateInfo imageInfo = {};
    imageInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    imageInfo.imageType = VK_IMAGE_TYPE_2D;
    imageInfo.format = format;
    imageInfo.extent.width = width;
    imageInfo.extent.height = height;
    imageInfo.extent.depth = 1;
    imageInfo.mipLevels = 1;
    imageInfo.arrayLayers = 1;
    imageInfo.samples = VK_SAMPLE_COUNT_1_BIT;
    imageInfo.tiling = tiling;
    imageInfo.usage = usage;
    imageInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    imageInfo.initialLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;

    checkError( vkCreateImage(device, &imageInfo, nullptr, &image) );

    VkMemoryRequirements memRequirements;
    vkGetImageMemoryRequirements(device, image, &memRequirements);

    VkMemoryAllocateInfo allocInfo = {};
    allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    allocInfo.allocationSize = memRequirements.size;
    allocInfo.memoryTypeIndex = findMemoryType(gpu, memRequirements.memoryTypeBits, properties);

    checkError( vkAllocateMemory(device, &allocInfo, nullptr, &imageMemory) );

    checkError( vkBindImageMemory(device, image, imageMemory, 0) );
}
```


Vulkan

destroyImage()

```
void destoryImage(VkDevice device, VkImage image, VkDeviceMemory bufferMemory)
{
    vkFreeMemory(device, bufferMemory, nullptr);
    vkDestroyImage(device, image, nullptr);
}
```

copyImage()

```
void copyImage(VkDevice device, VkCommandPool cmdPool, VkQueue queue, VkImage srcImage,
            VkImage dstImage, uint32_t width, uint32_t height)
{
    VkCommandBuffer commandBuffer = beginSingleCommandBuffer(device, cmdPool);

    VkImageSubresourceLayers subResource = {};
    subResource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    subResource.baseArrayLayer = 0;
    subResource.mipLevel = 0;
    subResource.layerCount = 1;

    VkImageCopy region = {};
    region.srcSubresource = subResource;
    region.dstSubresource = subResource;
    region.srcOffset = { 0, 0, 0 };
    region.dstOffset = { 0, 0, 0 };
    region.extent.width = width;
    region.extent.height = height;
    region.extent.depth = 1;

    vkCmdCopyImage(
        commandBuffer,
        srcImage, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL,
        dstImage, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,
        1, &region
    );

    endSingleCommandBuffer(device, cmdPool, queue, commandBuffer);
}
```

beginSingleCommandBuffer() / endSingleCommandBuffer()

```
VkCommandBuffer beginSingleCommandBuffer(VkDevice device, VkCommandPool cmdPool)
{
    // command buffer allocation
    VkCommandBufferAllocateInfo allocInfo = {};
    allocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
    allocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
    allocInfo.commandPool = cmdPool;
    allocInfo.commandBufferCount = 1;

    VkCommandBuffer commandBuffer;
    checkError( vkAllocateCommandBuffers(device, &allocInfo, &commandBuffer) );

    VkCommandBufferBeginInfo beginInfo = {};
    beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
    beginInfo.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;

    checkError( vkBeginCommandBuffer(commandBuffer, &beginInfo) );

    return commandBuffer;
}

void endSingleCommandBuffer(VkDevice device, VkCommandPool cmdPool, VkQueue queue, VkCommandBuffer commandBuffer)
{
    checkError( vkEndCommandBuffer(commandBuffer) );

    VkSubmitInfo submitInfo = {};
    submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
    submitInfo.commandBufferCount = 1;
    submitInfo.pCommandBuffers = &commandBuffer;

    checkError( vkQueueSubmit(queue, 1, &submitInfo, VK_NULL_HANDLE) );
    checkError( vkQueueWaitIdle(queue) );

    vkFreeCommandBuffers(device, cmdPool, 1, &commandBuffer);
}
```

Texture Image

Image Layout

- User optimal layout for Vulkan image object according to usage
- User barrier object to synchronize image layout transition

Pipeline barrier

- Use the resource read/write synchronization

-Use image layout transition, queue family ownership transfer synchronization in VK_SHARING_MODE_EXCLUSIVE mode

- Image memory barrier (VkImageMemoryBarrier) : image layout transition

  Buffer memory barrier (VkBufferMemoryBarrier) : buffer synchronization

transitionImageLayout() #1

```
void transitionImageLayout(VkDevice device, VkCommandPool cmdPool, VkQueue queue, VkImage image,
    VkImageLayout oldLayout, VkImageLayout newLayout)
{
    VkCommandBuffer commandBuffer = beginSingleCommandBuffer(device, cmdPool);

    VkImageMemoryBarrier barrier = {};
    barrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
    barrier.oldLayout = oldLayout;
    barrier.newLayout = newLayout;
    barrier.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    barrier.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    barrier.image = image;

    if (newLayout == VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL) {
        barrier.subresourceRange.aspectMask = VK_IMAGE_ASPECT_DEPTH_BIT;
    }
    else {
        barrier.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    }

    barrier.subresourceRange.baseMipLevel = 0;
    barrier.subresourceRange.levelCount = 1;
    barrier.subresourceRange.baseArrayLayer = 0;
    barrier.subresourceRange.layerCount = 1;
```

Layout transition

Queue family ownership transfer

Vulkan

transitionImageLayout() #2

```
    if (oldLayout == VK_IMAGE_LAYOUT_PREINITIALIZED && newLayout == VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL) {
        barrier.srcAccessMask = 0;
        barrier.dstAccessMask = VK_ACCESS_TRANSFER_READ_BIT;
    }
    else if (oldLayout == VK_IMAGE_LAYOUT_PREINITIALIZED && newLayout == VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL) {
        barrier.srcAccessMask = 0;
        barrier.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
    }
    else if (oldLayout == VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL && newLayout == VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL) {
        barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
        barrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT | VK_ACCESS_INPUT_ATTACHMENT_READ_BIT;
    }
    else if (oldLayout == VK_IMAGE_LAYOUT_UNDEFINED && newLayout == VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL) {
        barrier.srcAccessMask = 0;
        barrier.dstAccessMask = VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT | VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;
    }
    else {
        throw std::invalid_argument("unsupported layout transition!");
    }

    vkCmdPipelineBarrier(
        commandBuffer,
        VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT, VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT,
        0,
        0, nullptr,
        0, nullptr,
        1, &barrier
    );

    endSingleTimeCommands(device, cmdPool, queue, commandBuffer);
}
```


Vulkan

Texture Image

Create

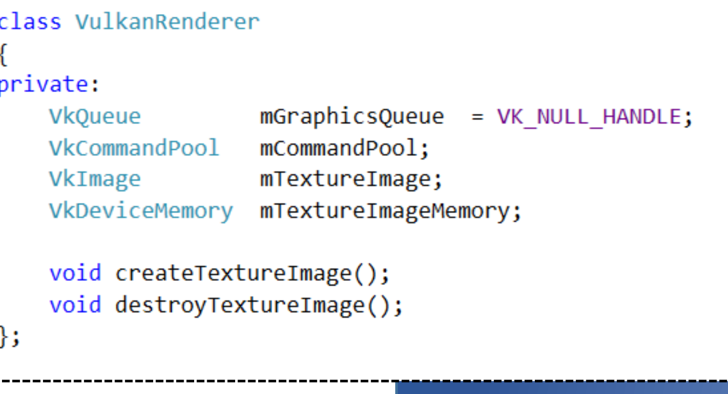
```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();

    createRenderPass();
    createFramebuffers();

    createCommandPool();
    createCommandBuffers();
    createDescriptorSetLayout();
    createGraphicsPipeline();

    createVertexBuffer();
    createIndexBuffer();

    createStagingUniformBuffer();
    createUniformBuffer();
    createTextureImage();
    createTextureImageView();
    createTextureSampler();
    createDescriptorPool();
    createDescriptorSet();

    recordCommandBuffers();
    createSemaphores();
}
```

Destroy

```
VulkanRenderer::~VulkanRenderer()
{
    destroySemaphores();

    destroyDescriptorPool();
    destroyTextureSampler();
    destroyTextureImageView();
    destroyTextureImage();
    destoryUniformBuffer();
    destroyStagingUniformBuffer();

    destroyIndexBuffer();
    destroyVertexBuffer();

    destroyGraphicsPipeline();
    destroyDescriptorSetLayout();
    destroyCommandPool();

    destroyFramebuffers();
    destroyRenderPass();

    destroyImageViews();
    destroySwapchain();

    destroyLogicalDevice();
    destroySurface();
    destroyInstance();
}
```

createTextureImage() #1

```
void VulkanRenderer::createTextureImage()
{
    // 1. read texture image data
    FILE *fp;
    if (fopen_s(&fp, "image.bmp", "rb") != 0) {
        return;
    }
    BITMAPFILEHEADER bfh;
    BITMAPINFOHEADER bih;
    fread(&bfh, 1, sizeof(BITMAPFILEHEADER), fp);   // read file header
    fread(&bih, 1, sizeof(BITMAPINFOHEADER), fp);   // read info header
    fseek(fp, bfh.bfOffBits, SEEK_SET);             // find start byte

    int texWidth = bih.biWidth;
    int texHeight = bih.biHeight;
    VkDeviceSize imageSize = texWidth * texHeight * 4;  // image size for rgba

    unsigned char *pImageData = new unsigned char[imageSize];
    unsigned char *tempPtr = pImageData;
    for (int y = 0; y < texHeight; y++) {
        for (int x = 0; x < texWidth; x++) {
            fread(&(tempPtr[2]), 1, 1, fp);
            fread(&(tempPtr[1]), 1, 1, fp);
            fread(&(tempPtr[0]), 1, 1, fp);
            tempPtr[3] = 255;
            tempPtr += 4;
        }
    }
```



```
class VulkanRenderer
{
private:
    VkQueue          mGraphicsQueue  = VK_NULL_HANDLE;
    VkCommandPool    mCommandPool;
    VkImage          mTextureImage;
    VkDeviceMemory   mTextureImageMemory;

    void createTextureImage();
    void destroyTextureImage();
};
```

createTextureImage() #2

```
class VulkanRenderer
{
private:
    VkQueue           mGraphicsQueue   = VK_NULL_HANDLE;
    VkCommandPool     mCommandPool;
    VkImage           mTextureImage;
    VkDeviceMemory    mTextureImageMemory;

    void createTextureImage();
    void destroyTextureImage();
};
```

```
// 2. staging image buffer (CPU accessible)
VkImage stagingImage;
VkDeviceMemory stagingImageMemory;
createImage(
    mGpu, mDevice,
    texWidth, texHeight,
    VK_FORMAT_R8G8B8A8_UNORM,
    VK_IMAGE_TILING_LINEAR,
    VK_IMAGE_USAGE_TRANSFER_SRC_BIT,
    VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
    stagingImage,
    stagingImageMemory);

// 3. copy bitmap data to VkDeviceMemory
void* data;
vkMapMemory(mDevice, stagingImageMemory, 0, imageSize, 0, &data);
memcpy(data, pImageData, (size_t)imageSize);
vkUnmapMemory(mDevice, stagingImageMemory);

// close file
fclose(fp);
delete[] pImageData;
```


Vulkan

createTextureImage() #3

```
class VulkanRenderer
{
private:
    VkQueue           mGraphicsQueue  = VK_NULL_HANDLE;
    VkCommandPool     mCommandPool;
    VkImage           mTextureImage;
    VkDeviceMemory    mTextureImageMemory;

    void createTextureImage();
    void destroyTextureImage();
};
```

```
    // 4. final image buffer
    createImage(
        mGpu, mDevice,
        texWidth, texHeight,
        VK_FORMAT_R8G8B8A8_UNORM,
        VK_IMAGE_TILING_OPTIMAL,
        VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT,
        VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
        mTextureImage,
        mTextureImageMemory);

    // 5. copy staging image buffer to final image buffer
    transitionImageLayout(mDevice, mCommandPool, mGraphicsQueue, stagingImage,
                          VK_IMAGE_LAYOUT_PREINITIALIZED, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL);
    transitionImageLayout(mDevice, mCommandPool, mGraphicsQueue, mTextureImage,
                          VK_IMAGE_LAYOUT_PREINITIALIZED, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL);
    copyImage(mDevice, mCommandPool, mGraphicsQueue, stagingImage, mTextureImage, texWidth, texHeight);

    // destroy staging VkImage
    vkDestroyImage(mDevice, stagingImage, nullptr);
    vkFreeMemory(mDevice, stagingImageMemory, nullptr);

    transitionImageLayout(mDevice, mCommandPool, mGraphicsQueue, mTextureImage,
                          VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL);
}
```

Texture Image View

Image View

- Have the texture image object and additional information

  (Texture view type, format, mipmap, texture array, etc.)

- Use texture image view as main handle instead of texture image

Vulkan.

Texture Image View

Create

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();

    createRenderPass();
    createFramebuffers();

    createCommandPool();
    createCommandBuffers();
    createDescriptorSetLayout();
    createGraphicsPipeline();

    createVertexBuffer();
    createIndexBuffer();

    createStagingUniformBuffer();
    createUniformBuffer();
    createTextureImage();
    createTextureImageView();
    createTextureSampler();
    createDescriptorPool();
    createDescriptorSet();

    recordCommandBuffers();
    createSemaphores();
}
```

Destroy

```
VulkanRenderer::~VulkanRenderer()
{
    destroySemaphores();

    destroyDescriptorPool();
    destroyTextureSampler();
    destroyTextureImageView();
    destroyTextureImage();
    destoryUniformBuffer();
    destroyStagingUniformBuffer();

    destroyIndexBuffer();
    destroyVertexBuffer();

    destroyGraphicsPipeline();
    destroyDescriptorSetLayout();
    destroyCommandPool();

    destroyFramebuffers();
    destroyRenderPass();

    destroyImageViews();
    destroySwapchain();

    destroyLogicalDevice();
    destroySurface();
    destroyInstance();
}
```

createTextureImageView()
destroyTextureImageViews()

```
class VulkanRenderer
{
private:
    VkImage          mTextureImage;
    VkDeviceMemory   mTextureImageMemory;
    VkImageView      mTextureImageView;

    void createTextureImageView();
    void destroyTextureImageView();
};
```

```
void VulkanRenderer::createTextureImageView()
{
    createImageView(mDevice, mTextureImage, VK_FORMAT_R8G8B8A8_UNORM, mTextureImageView);
}

void VulkanRenderer::destroyTextureImageView()
{
    vkDestroyImageView(mDevice, mTextureImageView, nullptr);
}
```

createImageView()

```
void createImageView(VkDevice device, VkImage image, VkFormat format, VkImageView & imageView)
{
    VkImageViewCreateInfo viewInfo = {};
    viewInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
    viewInfo.image = image;
    viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
    viewInfo.format = format;
    viewInfo.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    viewInfo.subresourceRange.baseMipLevel = 0;
    viewInfo.subresourceRange.levelCount = 1;
    viewInfo.subresourceRange.baseArrayLayer = 0;
    viewInfo.subresourceRange.layerCount = 1;

    checkError( vkCreateImageView(device, &viewInfo, nullptr, &imageView) );
}
```

Sampler

Sampler

- Shader does not sample a texture image directly, but accesses it through a sampler
- Samplers support texture filter, mipmap, wrap mode

Sampler

Create

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();

    createRenderPass();
    createFramebuffers();

    createCommandPool();
    createCommandBuffers();
    createDescriptorSetLayout();
    createGraphicsPipeline();

    createVertexBuffer();
    createIndexBuffer();

    createStagingUniformBuffer();
    createUniformBuffer();
    createTextureImage();
    createTextureImageView();
    createTextureSampler();
    createDescriptorPool();
    createDescriptorSet();

    recordCommandBuffers();
    createSemaphores();
}
```

Destroy

```
VulkanRenderer::~VulkanRenderer()
{
    destroySemaphores();

    destroyDescriptorPool();
    destroyTextureSampler();
    destroyTextureImageView();
    destroyTextureImage();
    destoryUniformBuffer();
    destroyStagingUniformBuffer();

    destroyIndexBuffer();
    destroyVertexBuffer();

    destroyGraphicsPipeline();
    destroyDescriptorSetLayout();
    destroyCommandPool();

    destroyFramebuffers();
    destroyRenderPass();

    destroyImageViews();
    destroySwapchain();

    destroyLogicalDevice();
    destroySurface();
    destroyInstance();
}
```

createTextureSampler()
destroyTextureSampler()

```
class VulkanRenderer
{
private:
    VkSampler    mTextureSampler;

    void createTextureSampler();
    void destroyTextureSampler();
};
```

```
void VulkanRenderer::createTextureSampler() {
    VkSamplerCreateInfo createInfo = {};
    createInfo.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
    createInfo.magFilter = VK_FILTER_LINEAR;
    createInfo.minFilter = VK_FILTER_LINEAR;
    createInfo.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
    createInfo.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
    createInfo.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
    createInfo.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
    createInfo.anisotropyEnable = VK_TRUE;
    createInfo.maxAnisotropy = 16;
    createInfo.compareEnable = VK_FALSE;
    createInfo.compareOp = VK_COMPARE_OP_ALWAYS;
    createInfo.borderColor = VK_BORDER_COLOR_INT_OPAQUE_BLACK;
    createInfo.unnormalizedCoordinates = VK_FALSE;

    checkError( vkCreateSampler(mDevice, &createInfo, nullptr, &mTextureSampler) );
}

void VulkanRenderer::destroyTextureSampler()
{
    vkDestroySampler(mDevice, mTextureSampler, nullptr);
}
```


Vulkan

Texture Descriptor Set

Texture Descriptor Set

- Use descriptor sets to deliver texture image views and samplers to a shader

Using Descriptor

1) Before graphics pipeline creation : Create the descriptor set layout

2) Graphics pipeline creation time : Specify the descriptor set layout

3) After graphics pipeline creation : Create the texture image view, sampler, descriptor pool and descriptor set

4) Rendering time : Descriptor binding

Resource Descriptor

Create

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();

    createRenderPass();
    createFramebuffers();

    createCommandPool();
    createCommandBuffers();
    createDescriptorSetLayout();
    createGraphicsPipeline();

    createVertexBuffer();
    createIndexBuffer();

    createStagingUniformBuffer();
    createUniformBuffer();
    createTextureImage();
    createTextureImageView();
    createTextureSampler();
    createDescriptorPool();
    createDescriptorSet();

    recordCommandBuffers();
    createSemaphores();
}
```

- createDescriptorSetLayout() → Create the descriptor set layout
- createGraphicsPipeline() → Specify the descriptor set layout
- createStagingUniformBuffer() … createDescriptorSet() → Create the texture image view / sampler; Create the Descriptor pool / descriptor set
- recordCommandBuffers() → Bind the descriptor set

createDescriptorSetLayout()

```
class VulkanRenderer
{
private:
    VkDescriptorSetLayout    mDescriptorSetLayout;

    void createDescriptorSetLayout();
    void destroyDescriptorSetLayout();
};
```

```
void VulkanRenderer::createDescriptorSetLayout() {

    std::array<VkDescriptorSetLayoutBinding, 2> descriptorSetLayoutBindings = {};

    // binding for mvp matrix
    descriptorSetLayoutBindings[0].binding = 0;
    descriptorSetLayoutBindings[0].descriptorCount = 1;
    descriptorSetLayoutBindings[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    descriptorSetLayoutBindings[0].stageFlags = VK_SHADER_STAGE_VERTEX_BIT;

    // binding for texture
    descriptorSetLayoutBindings[1].binding = 1;
    descriptorSetLayoutBindings[1].descriptorCount = 1;
    descriptorSetLayoutBindings[1].descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
    descriptorSetLayoutBindings[1].pImmutableSamplers = nullptr;
    descriptorSetLayoutBindings[1].stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;

    VkDescriptorSetLayoutCreateInfo createInfo = {};
    createInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
    createInfo.bindingCount = (uint32_t)descriptorSetLayoutBindings.size();
    createInfo.pBindings = descriptorSetLayoutBindings.data();

    checkError( vkCreateDescriptorSetLayout(mDevice, &createInfo, nullptr, &mDescriptorSetLayout) );
}
```

createGraphicsPipeline()

```
class VulkanRenderer
{
private:
    VkPipelineLayout        mPipelineLayout;
    VkPipeline              mPipeline;
    VkDescriptorSetLayout   mDescriptorSetLayout;

    void createGraphicsPipeline();
};
```

```
// Pipeline layout for passing uniform values to shaders
VkPipelineLayoutCreateInfo pipelineLayoutInfo = {};
{
    pipelineLayoutInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
    pipelineLayoutInfo.setLayoutCount = 1;
    pipelineLayoutInfo.pSetLayouts = &mDescriptorSetLayout;
    pipelineLayoutInfo.pushConstantRangeCount = 0;
    pipelineLayoutInfo.pPushConstantRanges = 0;

    checkError(vkCreatePipelineLayout(mDevice, &pipelineLayoutInfo, nullptr, &mPipelineLayout));
}

VkGraphicsPipelineCreateInfo pipelineInfo = {};
// ...
pipelineInfo.layout = mPipelineLayout;
// ...

checkError(vkCreateGraphicsPipelines(mDevice, VK_NULL_HANDLE, 1, &pipelineInfo, nullptr, &mPipeline));
```

Create the Descriptor Pool

Descriptor Pool

- Add texture image descriptor types to the descriptor pool

Create the Descriptor Pool

Create

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();

    createRenderPass();
    createFramebuffers();

    createCommandPool();
    createCommandBuffers();
    createDescriptorSetLayout();
    createGraphicsPipeline();

    createVertexBuffer();
    createIndexBuffer();

    createStagingUniformBuffer();
    createUniformBuffer();
    createTextureImage();
    createTextureImageView();
    createTextureSampler();
    createDescriptorPool();
    createDescriptorSet();

    recordCommandBuffers();
    createSemaphores();
}
```

Destroy

```
VulkanRenderer::~VulkanRenderer()
{
    destroySemaphores();

    destroyDescriptorPool();
    destroyTextureSampler();
    destroyTextureImageView();
    destroyTextureImage();
    destoryUniformBuffer();
    destroyStagingUniformBuffer();

    destroyIndexBuffer();
    destroyVertexBuffer();

    destroyGraphicsPipeline();
    destroyDescriptorSetLayout();
    destroyCommandPool();

    destroyFramebuffers();
    destroyRenderPass();

    destroyImageViews();
    destroySwapchain();

    destroyLogicalDevice();
    destroySurface();
    destroyInstance();
}
```

createDescriptorPool()

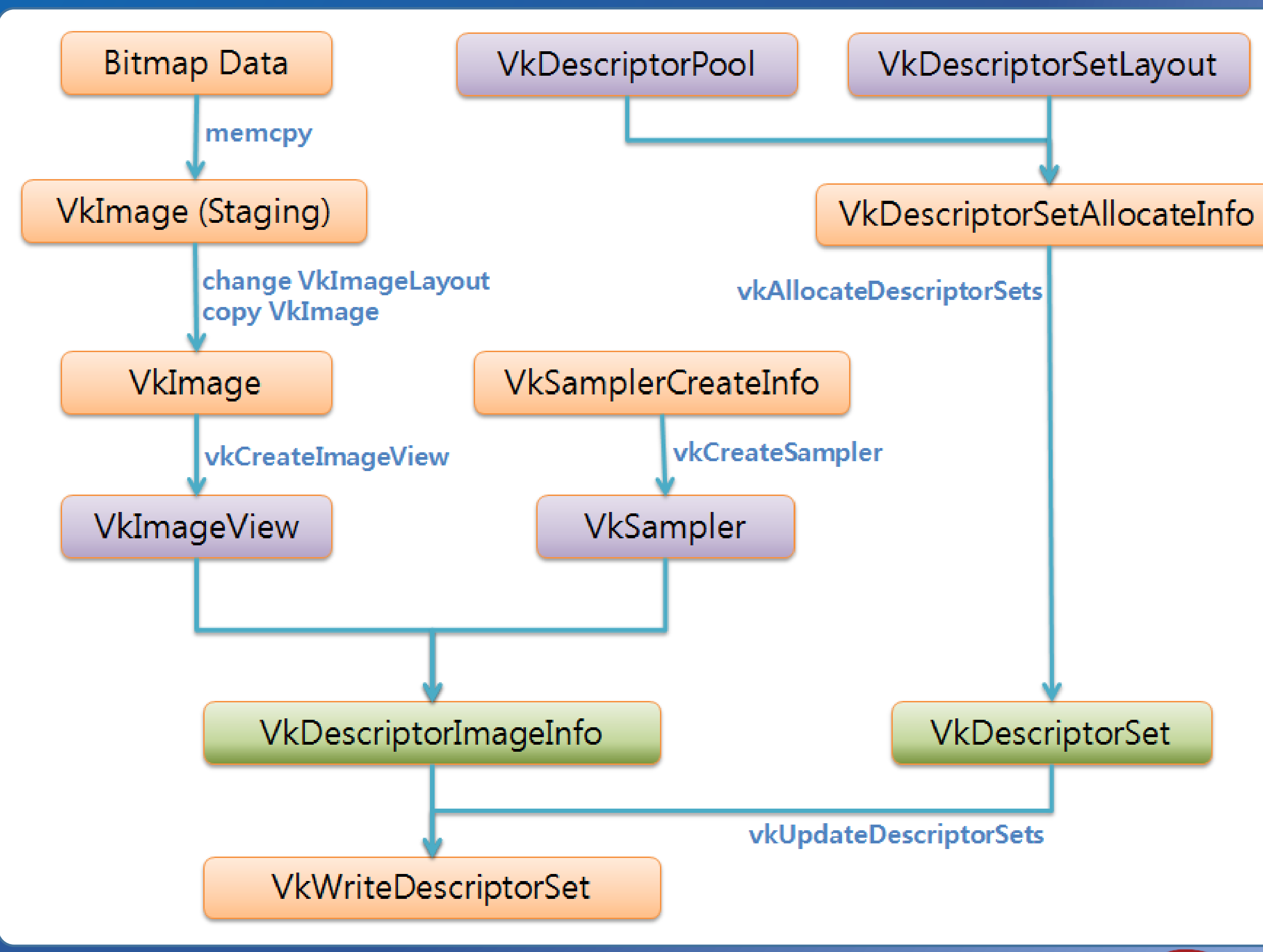
```
class VulkanRenderer
{
private:
    VkDescriptorPool            mDescriptorPool;

    void createDescriptorPool();
    void destroyDescriptorPool();
};
```

```
void VulkanRenderer::createDescriptorPool()
{
    std::array<VkDescriptorPoolSize, 2> descritorPoolSizes = {};
    descritorPoolSizes[0].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;         // for mvp uniform
    descritorPoolSizes[0].descriptorCount = 1;
    descritorPoolSizes[1].type = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER; // for texture
    descritorPoolSizes[1].descriptorCount = 1;

    VkDescriptorPoolCreateInfo createInfo = {};
    createInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
    createInfo.poolSizeCount = (uint32_t)descritorPoolSizes.size();
    createInfo.pPoolSizes = descritorPoolSizes.data();
    createInfo.maxSets = 1;

    checkError( vkCreateDescriptorPool(mDevice, &createInfo, nullptr, &mDescriptorPool) );
}
```

Create the Descriptor Set

Descriptor Set

- Create the texture type descriptor set



Vulkan

Create the Descriptor Set

Create

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();

    createRenderPass();
    createFramebuffers();

    createCommandPool();
    createCommandBuffers();
    createDescriptorSetLayout();
    createGraphicsPipeline();

    createVertexBuffer();
    createIndexBuffer();

    createStagingUniformBuffer();
    createUniformBuffer();
    createTextureImage();
    createTextureImageView();
    createTextureSampler();
    createDescriptorPool();
    createDescriptorSet();

    recordCommandBuffers();
    createSemaphores();
}
```

Destroy

```
VulkanRenderer::~VulkanRenderer()
{
    destroySemaphores();

    destroyDescriptorPool();
    destroyTextureSampler();
    destroyTextureImageView();
    destroyTextureImage();
    destoryUniformBuffer();
    destroyStagingUniformBuffer();

    destroyIndexBuffer();
    destroyVertexBuffer();

    destroyGraphicsPipeline();
    destroyDescriptorSetLayout();
    destroyCommandPool();

    destroyFramebuffers();
    destroyRenderPass();

    destroyImageViews();
    destroySwapchain();

    destroyLogicalDevice();
    destroySurface();
    destroyInstance();
}
```

※ Descriptor sets are destroyed automatically when destroying the descriptor pool

createDescriptorSet()

```
class VulkanRenderer
{
private:
    VkBuffer                mUniformBuffer;
    VkDescriptorSetLayout   mDescriptorSetLayout;
    VkDescriptorPool        mDescriptorPool;
    VkDescriptorSet         mDescriptorSet;

    void createDescriptorSet();
};
```

```
std::array<VkWriteDescriptorSet, 2> writeDescriptorSets = {};

// descriptorBufferInfor for mvp
VkDescriptorBufferInfo bufferInfo = {};
bufferInfo.buffer = mUniformBuffer;
bufferInfo.offset = 0;
bufferInfo.range = sizeof(UniformBufferObject);

writeDescriptorSets[0].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
writeDescriptorSets[0].dstSet = mDescriptorSet;
writeDescriptorSets[0].dstBinding = 0;
writeDescriptorSets[0].dstArrayElement = 0;
writeDescriptorSets[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
writeDescriptorSets[0].descriptorCount = 1;
writeDescriptorSets[0].pBufferInfo = &bufferInfo;

// descriptorBufferInfor for texture
VkDescriptorImageInfo imageInfo = {};
imageInfo.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
imageInfo.imageView = mTextureImageView;
imageInfo.sampler = mTextureSampler;

writeDescriptorSets[1].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
writeDescriptorSets[1].dstSet = mDescriptorSet;
writeDescriptorSets[1].dstBinding = 1;
writeDescriptorSets[1].dstArrayElement = 0;
writeDescriptorSets[1].descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
writeDescriptorSets[1].descriptorCount = 1;
writeDescriptorSets[1].pImageInfo = &imageInfo;

vkUpdateDescriptorSets(mDevice, (uint32_t)writeDescriptorSets.size(), writeDescriptorSets.data(), 0, nullptr);
```


Vulkan

Descriptor Set Binding

Descriptor Set Binding

- Bind the descriptor set at rendering time
- Submit binding command in command buffer, before submitting drawing commands

Bind the Descriptor Set

Create

```
VulkanRenderer::VulkanRenderer()
{
    enableLayersAndExtensions();

    createInstance();
    createSurface();
    selectPhysicalDevice();
    createLogicalDevice();

    createSwapchain();
    createImageViews();

    createRenderPass();
    createFramebuffers();

    createCommandPool();
    createCommandBuffers();
    createDescriptorSetLayout();
    createGraphicsPipeline();

    createVertexBuffer();
    createIndexBuffer();

    createStagingUniformBuffer();
    createUniformBuffer();
    createTextureImage();
    createTextureImageView();
    createTextureSampler();
    createDescriptorPool();
    createDescriptorSet();

    recordCommandBuffers();
    createSemaphores();
}
```

Destroy

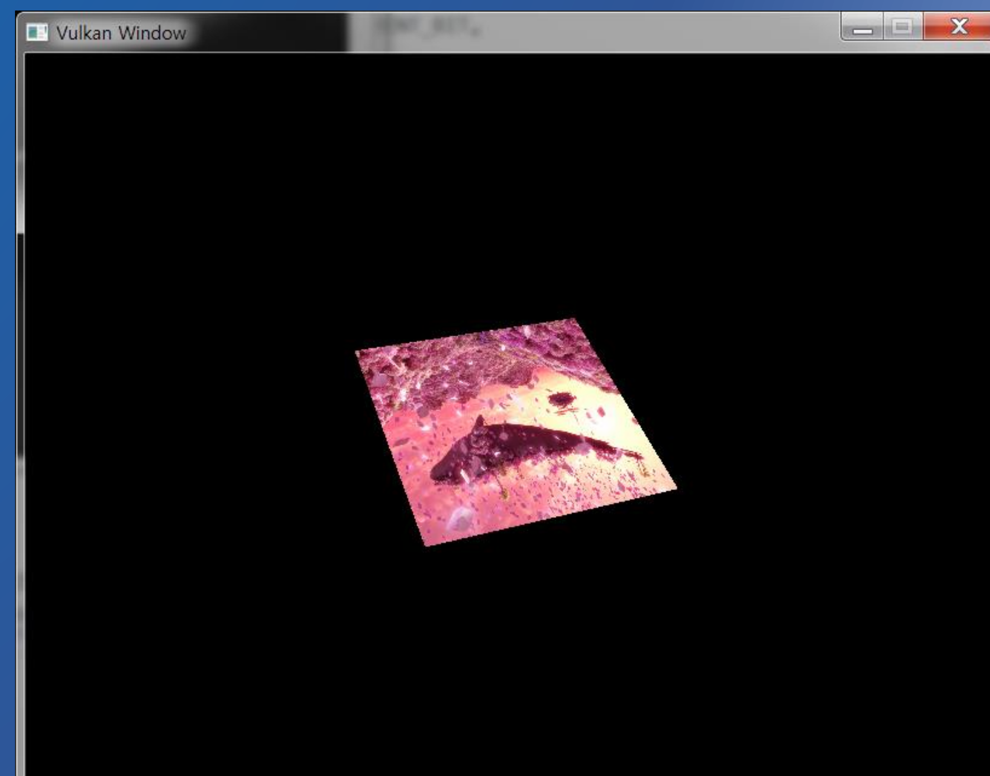
```
VulkanRenderer::~VulkanRenderer()
{
    destroySemaphores();

    destroyDescriptorPool();
    destroyTextureSampler();
    destroyTextureImageView();
    destroyTextureImage();
    destoryUniformBuffer();
    destroyStagingUniformBuffer();

    destroyIndexBuffer();
    destroyVertexBuffer();

    destroyGraphicsPipeline();
    destroyDescriptorSetLayout();
    destroyCommandPool();

    destroyFramebuffers();
    destroyRenderPass();

    destroyImageViews();
    destroySwapchain();

    destroyLogicalDevice();
    destroySurface();
    destroyInstance();
}
```

recordCommandBuffers()

```
vkCmdBeginRenderPass(mCommandBuffers[i], &renderPassInfo, VK_SUBPASS_CONTENTS_INLINE);

    vkCmdBindPipeline(mCommandBuffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS, mPipeline);

    // binding the vertex buffer
    VkBuffer vertexBuffers[] = {mVertexBuffer};
    VkDeviceSize offsets[] = {0};
    vkCmdBindVertexBuffers( mCommandBuffers[i], 0, 1, vertexBuffers, offsets );

    // binidng the index buffer
    vkCmdBindIndexBuffer( mCommandBuffers[i], mIndexBuffer, 0, VK_INDEX_TYPE_UINT16 );

    vkCmdBindDescriptorSets(
        mCommandBuffers[i],
        VK_PIPELINE_BIND_POINT_GRAPHICS,
        mPipelineLayout,
        0,                  // first set
        1,                  // descriptor set count
        &mDescriptorSet,
        0,                  // dynamic offset count
        nullptr             // dynamic offset
    );

    // Drawing rectangle using index buffer
    vkCmdDrawIndexed( mCommandBuffers[i], (uint32_t)gIndices.size(), 1, 0, 0, 0 );

vkCmdEndRenderPass(mCommandBuffers[i]);
```

Texture Mapping

- Texture Image
- Sampler
- Texture Descriptor Set



Enable the Standard Validation Layer

Vulkan Design Concepts

- Minimal driver overhead
- User control most thing explicitly
- Very limited error checking

Validation Layer

- Support various functions and services by hooking Vulkan API
- In general, enable validation layer in debug mode and disable in release mode

Vulkan API Hooking Example

```
// vkCreateInstance in Validation Layer
// This will be called if validation layer is enabled
VkResult vkCreateInstance(const VkInstanceCreateInfo* pCreateInfo, const VkAllocationCallbacks* pAllocator, VkInstance* instance)
{
    // check parameters
    if (pCreateInfo == nullptr || instance == nullptr) {
        log("Null pointer passed to required parameter!");
        return VK_ERROR_INITIALIZATION_FAILED;
    }

    // call original vkCreateInstance()
    VkResult result = real_vkCreateInstance(pCreateInfo, pAllocator, instance);

    // check return values if needed
    if (result != VK_SUCCESS) {
        ...
    }

    return result;
}
```

Check error before calling the final vkCreateInstance( )

Check result value after calling the final vkCreateInstance( )

Standard Validation Layer

Validation Layers	Description
"VK_LAYER_LUNARG_standard_validation"	All of the standard validation layers (listed below)
"VK_LAYER_GOOGLE_threading"	Check multithreading of API calls for validity
"VK_LAYER_LUNARG_parameter_validation"	Check the input parameters to API calls for validity
"VK_LAYER_LUNARG_object_tracker"	Track object creation, use, and destruction. As objects are created they are stored in a map. As objects are used the layer verifies they exist in the map, flagging errors for unknown objects.
"VK_LAYER_LUNARG_core_validation"	Core_validation includes tracking object bindings, memory hazards, and memory object lifetimes. It also validates several other hazard-related issues related to command buffers, fences, and memory mapping. Additionally core_validation include shader validation
"VK_LAYER_LUNARG_image"	The image layer is intended to validate image parameters, formats, and correct use
"VK_LAYER_LUNARG_swapchain"	Check that WSI(Window System Integration) extensions are being used correctly
"VK_LAYER_GOOGLE_unique_objects"	The Vulkan specification allows objects that have non-unique handles

- The LunarG Vulkan SDK supports standard validation layer
- Users can develop customized validation layers

Enable the Standard Validation Layer

Create

```
VulkanRenderer::VulkanRenderer()
{
    setupValidationLayers();

    createInstance();
    createDebugReportCallback();
    selectPhysicalDevice();
    createLogicalDevice();
}
```

Set validation layer information before creating instance

→ This information is used when creating the instance

Destroy

```
VulkanRenderer::~VulkanRenderer()
{
    destroyLogicalDevice();
    detroyDebugReportCallback();
    destroyInstance();
}
```

setupValidationLayers()

```
void VulkanRenderer::setupValidationLayers()
{
    // enable validation layers
    mInstanceLayers.push_back("VK_LAYER_LUNARG_standard_validation");
//  mInstanceLayers.push_back("VK_LAYER_LUNARG_threading");
//  mInstanceLayers.push_back("VK_LAYER_GOOGLE_threading");
//  mInstanceLayers.push_back("VK_LAYER_LUNARG_draw_state");
//  mInstanceLayers.push_back("VK_LAYER_LUNARG_image");
//  mInstanceLayers.push_back("VK_LAYER_LUNARG_mem_tracker");
//  mInstanceLayers.push_back("VK_LAYER_LUNARG_object_tracker");
//  mInstanceLayers.push_back("VK_LAYER_LUNARG_param_tracker");

    // Device layer was deprecated
//  mDeviceLayers.push_back("VK_LAYER_LUNARG_standard_validation");
//  mDeviceLayers.push_back("VK_LAYER_LUNARG_threading");
//  mDeviceLayers.push_back("VK_LAYER_GOOGLE_threading");
//  mDeviceLayers.push_back("VK_LAYER_LUNARG_draw_state");
//  mDeviceLayers.push_back("VK_LAYER_LUNARG_image");
//  mDeviceLayers.push_back("VK_LAYER_LUNARG_mem_tracker");
//  mDeviceLayers.push_back("VK_LAYER_LUNARG_object_tracker");
//  mDeviceLayers.push_back("VK_LAYER_LUNARG_param_tracker");

    // enable callback extension for validation layer
    mInstanceExtensions.push_back(VK_EXT_DEBUG_REPORT_EXTENSION_NAME);

    // debug report callback for validation layer
    mDebugReportCreateInfo.sType        = VK_STRUCTURE_TYPE_DEBUG_REPORT_CREATE_INFO_EXT;
    mDebugReportCreateInfo.pfnCallback  = (PFN_vkDebugReportCallbackEXT) VulkanDebugCallback;
    mDebugReportCreateInfo.flags        =
//      VK_DEBUG_REPORT_INFORMATION_BIT_EXT |
        VK_DEBUG_REPORT_WARNING_BIT_EXT |
        VK_DEBUG_REPORT_PERFORMANCE_WARNING_BIT_EXT |
        VK_DEBUG_REPORT_ERROR_BIT_EXT |
//      VK_DEBUG_REPORT_DEBUG_BIT_EXT |
        0;
}
```

Used when creating the instance

- instance layer
- instance extension
- debug report callback information

Device Layer : deprecated (SDK ver. 1.0.13.0)

VK_EXT_debug_report

User-defined callback function

```
class VulkanRenderer
{
private:
    std::vector<const char*>    mInstanceLayers;
    std::vector<const char*>    mInstanceExtensions;

    VkDebugReportCallbackEXT            mDebugReportCallback = VK_NULL_HANDLE;
    VkDebugReportCallbackCreateInfoEXT  mDebugReportCreateInfo {};
};
```

createInstance()

```
class VulkanRenderer
{
private:
    std::vector<const char*>     mInstanceLayers;
    std::vector<const char*>     mInstanceExtensions;

    VkDebugReportCallbackEXT             mDebugReportCallback = VK_NULL_HANDLE;
    VkDebugReportCallbackCreateInfoEXT   mDebugReportCreateInfo {};
};
```

```
void VulkanRenderer::createInstance()
{
    // check instance layer support
    if ( !checkInstanceLayerSupport(mInstanceLayers) ) {
        assert(0 && "Not available Instance Layer!");
        std::exit(-1);
    }

    // check instance extension support
    if ( !checkInstanceExtensionSupport(mInstanceExtensions) ) {
        assert(0 && "Not available Instance Extension!");
        std::exit(-1);
    }

    // Optional info
    VkApplicationInfo appInfo {};
    appInfo.sType                  = VK_STRUCTURE_TYPE_APPLICATION_INFO;
    appInfo.apiVersion             = VK_MAKE_VERSION(1, 0, 17);
    appInfo.applicationVersion     = VK_MAKE_VERSION(0, 1, 0);
    appInfo.pApplicationName       = "Vulkan Tutorial";
//  appInfo.pNext                  = ;     // point to extension info

    // Mandatory info
    VkInstanceCreateInfo instance_create_info{};
    instance_create_info.sType                    = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
    instance_create_info.pApplicationInfo         = &appInfo;
    instance_create_info.enabledLayerCount        = (uint32_t)mInstanceLayers.size();
    instance_create_info.ppEnabledLayerNames      = mInstanceLayers.data();
    instance_create_info.enabledExtensionCount    = (uint32_t)mInstanceExtensions.size();
    instance_create_info.ppEnabledExtensionNames  = mInstanceExtensions.data();
    instance_create_info.pNext                    = &mDebugReportCreateInfo;

    checkError(vkCreateInstance(&instance_create_info, nullptr, &mInstance));
}
```

Set below information at setupValidationLayers( )

- instance layer
- instance extension
- debug report callback information

Vulkan.

checkInstanceLayerSupport()

```
bool checkInstanceLayerSupport(const std::vector<const char*>& instanceLayers)
{
    // query available instance layers
    uint32_t layerCount = 0;
    vkEnumerateInstanceLayerProperties(&layerCount, nullptr);
    std::vector<VkLayerProperties> properties(layerCount);
    vkEnumerateInstanceLayerProperties(&layerCount, properties.data());

    for (const auto &i : instanceLayers) {
        bool layerFound = false;

        for (const auto &j : properties) {
            if ( strcmp(i, j.layerName) == 0 ) {
                layerFound = true;
                break;
            }
        }
    }

    return true;
}
```

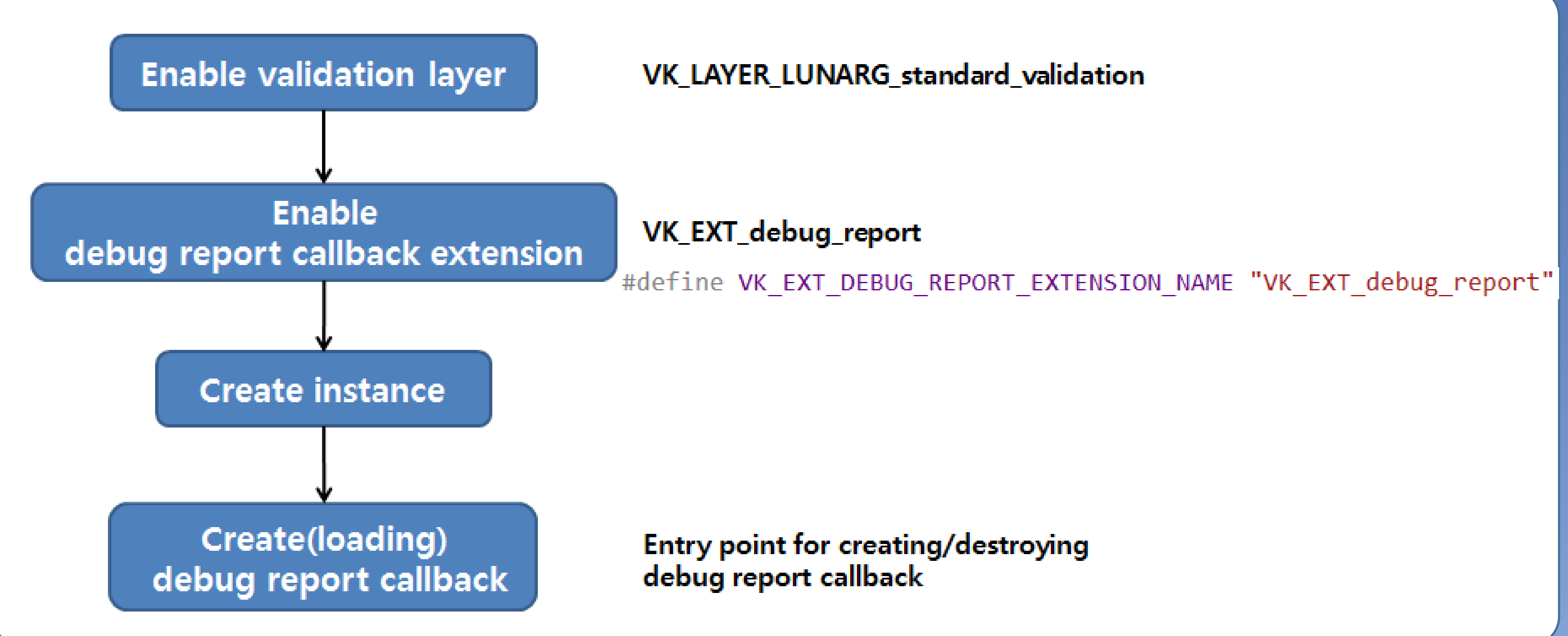
checkInstanceExtensionSupport()

```
bool checkInstanceExtensionSupport(const std::vector<const char*>& instanceExtensions)
{
    // query instance extensions
    uint32_t extensionCount = 0;
    vkEnumerateInstanceExtensionProperties(nullptr, &extensionCount, nullptr);
    std::vector<VkExtensionProperties> extensions(extensionCount);
    vkEnumerateInstanceExtensionProperties(nullptr, &extensionCount, extensions.data());

    for (const auto &i : instanceExtensions) {
        bool extensionFound = false;

        for (const auto &j : extensions) {
            if ( strcmp(i, j.extensionName) == 0 ) {
                extensionFound = true;
                break;
            }
        }
    }

    return true;
}
```

Set the Debug Report Callback

Debug Report Callback

- Define an error report callback that will be called when error detected in validation layer



Create the Debug Report Callback

Create

```
VulkanRenderer::VulkanRenderer()
{
    setupValidationLayers();

    createInstance();
    createDebugReportCallback();
    selectPhysicalDevice();
    createLogicalDevice();
}
```

Destroy

```
VulkanRenderer::~VulkanRenderer()
{
    destroyLogicalDevice();
    detroyDebugReportCallback();
    destroyInstance();
}
```

createDebugReportCallback() / destroyDebugReportCallback()

```
PFN_vkCreateDebugReportCallbackEXT  fvkCreateDebugReportCallbackEXT = nullptr;
PFN_vkDestroyDebugReportCallbackEXT fvkDestroyDebugReportCallbackEXT = nullptr;

void VulkanRenderer::createDebugReportCallback()
{
    fvkCreateDebugReportCallbackEXT = (PFN_vkCreateDebugReportCallbackEXT)vkGetInstanceProcAddr(mInstance, "vkCreateDebugReportCallbackEXT");
    fvkDestroyDebugReportCallbackEXT = (PFN_vkDestroyDebugReportCallbackEXT)vkGetInstanceProcAddr(mInstance, "vkDestroyDebugReportCallbackEXT")

    if (fvkCreateDebugReportCallbackEXT == nullptr || fvkDestroyDebugReportCallbackEXT == nullptr) {
        assert(0 && "Failed to fetch debug function pointers.");
        std::exit(-1);
    }

    // create extension object(_debug_report)
    checkError( fvkCreateDebugReportCallbackEXT(mInstance, &mDebugReportCreateInfo, nullptr, &mDebugReportCallback) );
}

void VulkanRenderer::detroyDebugReportCallback()
{
    fvkDestroyDebugReportCallbackEXT(mInstance, mDebugReportCallback, nullptr);
    mDebugReportCallback = VK_NULL_HANDLE;
}
```

- Error report callbacks need to be created/destroyed explicitly
- Use vkCreateDebugReportCallbackEXT( ) and vkDestoryDebugReportCallbackEXT( ) functions
- These functions are not loaded automatically because those are extensions. Get function pointer using vkGetInstanceProcAddr( )

VulkanDebugCallback()

Set in VkDebugReportCallbackCreateInfoEXT

```
/*
*   debug report callback for validation layer
*   prototype : PFN_vkDebugReportCallbackEXT
*/
VKAPI_ATTR VkBool32 VKAPI_CALL
VulkanDebugCallback(
    VkDebugReportFlagsEXT       flags,          // type of message
    VkDebugReportObjectTypeEXT  objType,        // type of object (subject of the message)
    uint64_t                    srcObj,     // object (VkPhysicalDevice, ... etc)
    size_t                      location,
    int32_t                     msgCode,
    const char*                 layerPrefix,
    const char*                 msg,            // message
    void*                       userData        // your own data
    )
{
    if (flags & VK_DEBUG_REPORT_INFORMATION_BIT_EXT) {
        std::cout << "[INFO | ";
    }
    if (flags & VK_DEBUG_REPORT_WARNING_BIT_EXT) {
        std::cout << "[WARNING | ";
    }
    if (flags & VK_DEBUG_REPORT_PERFORMANCE_WARNING_BIT_EXT) {
        std::cout << "[PERFORMANCE | ";
    }
    if (flags & VK_DEBUG_REPORT_ERROR_BIT_EXT) {
        std::cout << "[ERROR | ";
    }
    if (flags & VK_DEBUG_REPORT_DEBUG_BIT_EXT) {
        std::cout << "[DEBUG | ";
    }
    std::cout << layerPrefix << "] ";
    std::cout << msg << std::endl;

    return false;
}
```

Vulkan

Appendix

- **Vulkan API Specification**
  https://www.khronos.org/registry/vulkan/specs/1.0/xhtml/vkspec.html
- **Building sample project in Vulkan SDK**
  https://www.youtube.com/watch?v=wHt5wcxIPcE (Tutorial 0)
- **Vulkan Validation Layers**
  http://gpuopen.com/using-the-vulkan-validation-layers/