VK Tutorial - 1

Drawing a triangle(1)

Contents

- GLFW
- Basic logic / concept
- Validation layers
- Physical device
- Queue families
- Logical device

GLFW?

GLFW is an Open Source, multi-platform library for OpenGL, OpenGL ES and Vulkan development on the desktop. It provides a simple API for creating windows, contexts and surfaces, receiving input and events.

GLFW is written in C and has native support for Windows, OS X and many Unix-like systems using the X Window System, such as Linux and FreeBSD.

GLFW is licensed under the zlib/libpng license.

GLFW

```
void MainApp::InitWindow()
{
    glfwInit();

    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
    glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);

    m_pWindow = glfwCreateWindow(WIDTH, HEIGHT, "VkTest1", nullptr, nullptr);
}

void MainApp::MainLoop()
{
    while (!glfwWindowShouldClose(m_pWindow))
    {
        glfwPollEvents();
    }
}
```

Basic concept / logic

- 1. Declare some structure contains information about creation
- 2. Fill in the attributes
- 3. Call vkCreateSth(CreateInfo, AllocateInfo, Interface) function

Basic concept / logic

```
void MainApp::Run()
{
    InitWindow();
    InitVulkan();
    MainLoop();
}

void MainApp::InitVulkan()
{
    CreateInstance();
    SetupDebugCallback();

    CreateSurface();
    PickPhysicalDevice();
    CreateLogicalDevice();

    CreateSwapChain();
    CreateImageViews();

    CreateRenderPass();

    CreateGraphicsPipeline();

    CreateFramebuffers();
    CreateCommandPool();
}
```

Instance creation

```
void MainApp::CreateInstance()
{
    if (enableValidationLayers && !CheckValidationLayerSupport())
    {
        throw std::runtime_error("Validation layers requested, but not available");
    }

    VkApplicationInfo appInfo = {};

    appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
    appInfo.apiVersion = VK_API_VERSION_1_0;
    appInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
    appInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
    appInfo.pApplicationName = "VKTest1";
    appInfo.pEngineName = "No engine";
    appInfo.pNext = nullptr;

    VkInstanceCreateInfo createInfo = {};

    createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
    createInfo.pApplicationInfo = &appInfo;

    auto extensions = GetRequiredExtensions();
    createInfo.enabledExtensionCount = (uint32_t)extensions.size();
    createInfo.ppEnabledExtensionNames = extensions.data();

    if (enableValidationLayers)
    {
        createInfo.enabledLayerCount = (uint32_t)validationLayers.size();
        createInfo.ppEnabledLayerNames = validationLayers.data();
    }
    else createInfo.enabledLayerCount = 0;

    if (vkCreateInstance(&createInfo, nullptr, m_Instance.replace()))
    {
        throw std::runtime_error("Failed to create instance");
    }
}
```

Validation layers?

- Checking the values of parameters against the specification to detect misuse
- Tracking creation and destruction of objects to find resource leaks
- Checking thread safety by tracking the threads that calls originate from
- Logging every call and its parameters to the standard output
- Tracing Vulkan calls for profiling and replaying

Validation layers?

```cpp
bool MainApp::CheckValidationLayerSupport()
{
    uint32_t layerCount;
    vkEnumerateInstanceLayerProperties(&layerCount, nullptr);

    std::vector<VkLayerProperties> availableLayers(layerCount);
    vkEnumerateInstanceLayerProperties(&layerCount, availableLayers.data());

    for (const char* layerName : validationLayers)
    {
        bool layerFound = false;

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

        if (!layerFound) return false;
    }

    return true;
}
```

```cpp
const std::vector<const char*> validationLayers = { "VK_LAYER_LUNARG_standard_validation" };
```

Validation layers?

```
VkResult CreateDebugReportCallbackEXT(
    VkInstance instance,
    const VkDebugReportCallbackCreateInfoEXT* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkDebugReportCallbackEXT* pCallback)
{
    auto func = (PFN_vkCreateDebugReportCallbackEXT)vkGetInstanceProcAddr
                                                    (instance, "vkCreateDebugReportCallbackEXT");

    if (func != nullptr) return func(instance, pCreateInfo, pAllocator, pCallback);

    else return VK_ERROR_EXTENSION_NOT_PRESENT;
}

void DestroyDebugReportCallbackEXT(
    VkInstance instance,
    VkDebugReportCallbackEXT callback,
    const VkAllocationCallbacks* pAllocator
)
{
    auto func = (PFN_vkDestroyDebugReportCallbackEXT)vkGetInstanceProcAddr
                                                    (instance, "vkDestroyDebugReportCallbackEXT");

    if (func != nullptr) func(instance, callback, nullptr);
}
```

Validation layers?

```
static VKAPI_ATTR VkBool32 VKAPI_CALL debugCallback(
    VkDebugReportFlagsEXT flags,
    VkDebugReportObjectTypeEXT objType,
    uint64_t obj,
    size_t location,
    int32_t code,
    const char* layerPrefix,
    const char* msg,
    void* userData
)
{
    std::cerr << "Validation layer : " << msg << std::endl;

    return VK_FALSE;
}
```

Physical device

```
void MainApp::PickPhysicalDevice()
{
    uint32_t deviceCnt = 0;
    vkEnumeratePhysicalDevices(m_Instance, &deviceCnt, nullptr);

    if (deviceCnt == 0) throw std::runtime_error("Failed to find GPUs with Vulkan support");

    std::vector<VkPhysicalDevice> devices(deviceCnt);
    vkEnumeratePhysicalDevices(m_Instance, &deviceCnt, devices.data());

    for (const auto& device : devices)
    {
        if (isDeviceSuitable(device))
        {
            m_PhysicalDevice = device;
            break;
        }
    }

    if (m_PhysicalDevice == VK_NULL_HANDLE) throw std::runtime_error("Failed to find a suitable GPU");
}
```

Queue families

- There are different types of queues that originate from different *queue families*
- Each family of queues allows only a subset of commands
- For example, there could be a queue family that only allows processing of compute commands or one that only allows memory transfer related commands.

Queue families

```
struct QueueFamilyIndices
{
    int graphicsFamily = -1;
    int presentFamily = -1;

    bool isComplete() { return graphicsFamily >= 0 && presentFamily >= 0; }
};
```

Queue families

```
QueueFamilyIndices MainApp::FindQueueFamilies(VkPhysicalDevice device)
{
    QueueFamilyIndices indices;

    uint32_t queueFamilyCnt = 0;
    vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCnt, nullptr);

    std::vector<VkQueueFamilyProperties> queueFamilies(queueFamilyCnt);
    vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCnt, queueFamilies.data());

    int i = 0;

    for (const auto& queueFamily : queueFamilies)
    {
        if (queueFamily.queueCount > 0 && (queueFamily.queueFlags & VK_QUEUE_GRAPHICS_BIT))
        {
            indices.graphicsFamily = i;
        }

        VkBool32 presentSupport = false;
        vkGetPhysicalDeviceSurfaceSupportKHR(device, i, m_Surface, &presentSupport);

        if (queueFamily.queueCount > 0 && presentSupport) indices.presentFamily = i;

        if (indices.isComplete()) break;

        i++;
    }

    return indices;
}
```

Logical device

```
const std::vector<const char*> DeviceExtensions = { VK_KHR_SWAPCHAIN_EXTENSION_NAME };
```

```
void MainApp::CreateLogicalDevice()
{
    QueueFamilyIndices indices = FindQueueFamilies(m_PhysicalDevice);

    std::vector<VkDeviceQueueCreateInfo> queueCreateInfos;
    std::set<int> uniqueQueueFamilies = { indices.graphicsFamily, indices.presentFamily };

    float queuePriority = 1.0f;

    for (int queueFamily : uniqueQueueFamilies)
    {
        VkDeviceQueueCreateInfo queueCreateInfo = {};

        queueCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
        queueCreateInfo.pQueuePriorities = &queuePriority;
        queueCreateInfo.queueCount = 1;
        queueCreateInfo.queueFamilyIndex = queueFamily;

        queueCreateInfos.push_back(queueCreateInfo);
    }
```

Logical device

```
    VkPhysicalDeviceFeatures deviceFeatures = {};

    VkDeviceCreateInfo createInfo = {};

    createInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;

    createInfo.enabledExtensionCount = (uint32_t)DeviceExtensions.size();
    createInfo.ppEnabledExtensionNames = DeviceExtensions.data();

    createInfo.pEnabledFeatures = &deviceFeatures;

    createInfo.pQueueCreateInfos = queueCreateInfos.data();
    createInfo.queueCreateInfoCount = (uint32_t)queueCreateInfos.size();

    if (enableValidationLayers)
    {
        createInfo.enabledLayerCount = (uint32_t)validationLayers.size();
        createInfo.ppEnabledLayerNames = validationLayers.data();
    }

    else createInfo.enabledExtensionCount = 0;

    if (vkCreateDevice(m_PhysicalDevice, &createInfo, nullptr, m_Device.replace()) != VK_SUCCESS)
    {
        throw std::runtime_error("Failed to create logical device");
    }

    vkGetDeviceQueue(m_Device, indices.graphicsFamily, 0, &m_Queue);
    vkGetDeviceQueue(m_Device, indices.presentFamily, 0, &m_PresentQueue);
}
```

Todo

- Swap chain
- Image views
- Render pass
- Graphics pipeline

Todo

- Frame buffers
- Command pool/ buffers
- Semaphore