# Vulkan core concepts

## Basics

### Definition

Vulkan is a low level, cross platform, graphics and computing API. It is supported by Valve through a group called LunarG.

- Low level: Access almost every GPU process individually. It willl be verbose! This will allow optimizations.
- Graphics : offline and online rendering
- Computing : GPGPU (General Programming...)

This course is an introduction to the Vulkan API. It will teach you the concepts and make you render but won't give you production ready code. It won't either focus on memory management. For a more advanced and production oriented course, please see VkGuide.

By the way, if you need additional information about the concepts studied here, please report to the Vulkan Tutorial.

### Installation

Download the SDK on https://vulkan.lunarg.com/ and install it.

You can find samples at https://github.com/LunarG/VulkanSamples and https://vulkan.lunarg.com/doc/view/1.3.239.0/windows/samples_index.html

Download GLFW precompiled binaries, 64 bits version for windows : https://www.glfw.org/download.html . It will be more compatible for most systems. Unzip the folder and rename it GLFW.

Also download and unzip the latest version of GLM from https://glm.g-truc.net/. Unzip it and rename the *first* glm folder to GLM. You can put both GLM and GLFW libs into a `externals` folder.

## Visual Studio config

Create a new empty C++ Visual Studio project, called VulkanApp, in the same folder as your `externals` folder.

In the project properties, modify a C++/General/Additional includes with :

```
$(SolutionDir)/../externals/GLM
$(SolutionDir)/../externals/GLFW/include
C:\VulkanSDK\1.3.239.1\Include
```

...depending on your Vulkan version.

You can now:

```
#include <GLFW/glfw3.h>
#include <glm/glm.hpp>
```

Let's go to Linker/General/Additional Library Directories and add the right GLFW version for your Visual Studio.

```
$(SolutionDir)/../externals/GLFW/lib-vc2019
C:\VulkanSDK\1.3.239.1\Lib
```

In Linker/Input/Addition Dependencies:

```
vulkan-1.lib
glfw3.lib
```

## Test

Test vulkan and glfw are working with:

```
#define GLFW_INCLUDE_VULKAN
#include <GLFW/glfw3.h>

#define GLM_FORCE_RADIANS
#define GLM_FORCE_DEPTH_ZERO_TO_ONE
#include <glm/glm.hpp>
#include <glm/mat4x4.hpp>

#include <iostream>
using std::cout;
using std::endl;

int main() {
	glfwInit();
	glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
	GLFWwindow* window = glfwCreateWindow(800, 600, "Test", nullptr, nullptr);

	uint32_t extensionCount = 0;
	vkEnumerateInstanceExtensionProperties(nullptr, &extensionCount, nullptr);
	cout << "Extension count: " << extensionCount << endl;

	while (!glfwWindowShouldClose(window)) {
		glfwPollEvents();
	}

	glfwDestroyWindow(window);
	glfwTerminate();
	return 0;
}
```

Your computer supports vulkan if the displayed count is superior to zero.

# Instances, devices and validation

## Concepts

A vulkan Instance is a reference to the vulkan context. It defines the vulkan version and its capabilities. When starting a vulkan application, we create a vulkan instance and enumerate all

the available Physical Devices (physical GPU). Then we create a Logical Device that will handle the reste of the work. Instances are rarely used beyond.

The logical device will serve as an interface to the physical devices, to their memory (for data) and queues - for FIFO commands, different queues with different uses. A type of queues is called a queue family. There can have multiple queues in each queue family. Example: graphics queue family, compute queue family, transfer queue family (move memory inside the GPU). Queue families can be combinations of these queues. When we enumerate physical devices, we need to check the device has the queue family.

To create a logical device, that will be our main tool, we follow this order of operations:

- Define queue families and number of queues we want to assign to the logical device from the physical device
- Define the features you want to enable on the L.D. (geometry shader, anisotropy...)
- Define the extensions the device will use. Extensions are a way to adress different systems. Ex: you have extensions to interact with windows on different OS. GLFW will load the right extensions for us.

It is possible to create multiple logical devices.

Validation layers. By default vulkan do not validate code and crash when meeting a fatal error. This is a necessity for optimization. If we want to check for errors, we have to enable multiple validation layers. Each layer can check different functions. Ex: VK_LAYER_LUNARG_standard_validation will be our main all-round validation layer. Note that validation layers are like extensions and are not build in the vulkan core. Showing errors from validation errors is itself an extension.

# Skeletton

Empty your Main.cpp file and create a new VulkanRenderer class.

`VulkanRenderer.h`

```
#pragma once
#define GLFW_INCLUDE_VULKAN
#include <GLFW/glfw3.h>

#include <stdexcept>
#include <vector>

class VulkanRenderer
{
public:
        VulkanRenderer();
        ~VulkanRenderer();

        int init(GLFWwindow* windowP);

private:
        GLFWwindow* window;
        vk::Instance instance;

        void createInstance();
};
```

`VulkanRenderer.cpp`

```
#include "VulkanRenderer.h"

int VulkanRenderer::init(GLFWwindow* windowP)
{
	window = windowP;
	try
	{
		createInstance();
	}
	catch (const std::runtime_error& e)
	{
		printf("ERROR: %s\n", e.what());
		return EXIT_FAILURE;
	}

	return EXIT_SUCCESS;
}

void VulkanRenderer::createInstance()
{
}
```

Use this empty renderer in the main.

`Main.cpp`

```
#define GLFW_INCLUDE_VULKAN
#include <GLFW/glfw3.h>
#include <stdexcept>
#include <vector>

#include <string>
using std::string;

#include "VulkanRenderer.h"


GLFWwindow* window = nullptr;
VulkanRenderer vulkanRenderer;

void initWindow(string wName = "Vulkan", const int width = 800, const int height = 600)
{
	// Initialize GLFW
	glfwInit();
	glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);		// Glfw won't work with opengl
	glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);

	window = glfwCreateWindow(width, height, wName.c_str(), nullptr, nullptr);
}

void clean()
{
	glfwDestroyWindow(window);
	glfwTerminate();
}

int main()
{
	initWindow();
	if(vulkanRenderer.init(window) == EXIT_FAILURE) return EXIT_FAILURE;

	while (!glfwWindowShouldClose(window))
	{
		glfwPollEvents();
	}
```

```
        clean();
        return 0;
}
```

# Instance

## Create the instance

To setup the instance, we will create an appInfo struct and a create info struct, then use this info in a function that runs the vulkan instance.

Note that the info struct are initialized with a list-initialize ( `{}` ). This will set the struct's fields to the default values.

`VulkanRenderer.cpp`

```
void VulkanRenderer::createInstance()
{
	// Information about the application
	// This data is for developer convenience
	vk::ApplicationInfo appInfo {};
	// Name of the app
	appInfo.pApplicationName = "Vulkan App";
	// Version of the application
	appInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
	// Custom engine name
	appInfo.pEngineName = "No Engine";
	// Custom engine version
	appInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
	// Vulkan version (here 1.1)
	appInfo.apiVersion = VK_API_VERSION_1_1;



	// Everything we create will be created with a createInfo
	// Here, info about the vulkan creation
	vk::InstanceCreateInfo createInfo {};
	// createInfo.pNext                // Extended information
	// createInfo.flags                // Flags with bitfield
	// Application info from above
	createInfo.pApplicationInfo = &appInfo;

	// Setup extensions instance will use
	std::vector<const char*> instanceExtensions;
	uint32_t glfwExtensionsCount = 0;                // Glfw may require multiple extensions
	const char** glfwExtensions = glfwGetRequiredInstanceExtensions(&glfwExtensionsCount);
	for (size_t i = 0; i < glfwExtensionsCount; ++i)
	{
		instanceExtensions.push_back(glfwExtensions[i]);
	}

	createInfo.enabledExtensionCount = static_cast<uint32_t>(instanceExtensions.size());
	createInfo.ppEnabledExtensionNames = instanceExtensions.data();

	// Validation layers, for now not used
	// TODO : setup
```

```
        createInfo.enabledLayerCount = 0;
        createInfo.ppEnabledLayerNames = nullptr;

        // Finally create instance
        instance = vk::createInstance(createInfo);
}
```

Run the code.

## Check the extensions

There is a chance the above code crashes, if the extensions we want to use are not supported. So we will create a function to check this.

`VulkanRenderer.cpp`

```
...
bool VulkanRenderer::checkInstanceExtensionSupport(const vector<const char*>& checkExtensions)
{
	// Create the vector of extensions
	vector<vk::ExtensionProperties> extensions = vk::enumerateInstanceExtensionProperties();

	// Check if given extensions are in list of available extensions
	for (const auto& checkExtension : checkExtensions)
	{
		bool hasExtension = false;
		for (const auto& extension : extensions)
		{
			if (strcmp(checkExtension, extension.extensionName) == 0)
			{
				hasExtension = true;
				break;
			}
		}
		if (!hasExtension) return false;
	}

	return true;
}
...
```

We will use this function in the `createInstance` function:

```
...
        for (size_t i = 0; i < glfwExtensionsCount; ++i)
        {
                instanceExtensions.push_back(glfwExtensions[i]);
        }
        // HERE

        // Check extensions
        if (!checkInstanceExtensionSupport(instanceExtensions))
        {
                throw std::runtime_error("VkInstance does not support required extensions");
        }

        // END HERE
        createInfo.enabledExtensionCount = static_cast<uint32_t>(instanceExtensions.size());
        createInfo.ppEnabledExtensionNames = instanceExtensions.data();
...
```

## Cleaning

Because the instance creation allocates some memory, we have to clean it. We will create a ``VulkanRenderer::clean` function.

`VulkanRenderer.cpp`

```
void VulkanRenderer::clean()
{
        instance.destroy();
}
```

We will use it in the main.

`Main.cpp`

```
void clean()
{
	vulkanRenderer.clean();
	glfwDestroyWindow(window);
	glfwTerminate();
}
```

# Devices

As stated above, there are physical devices (GPUs) and we will create a logical device to be used by our program.

## Physical Device

First, we will store a struct that will hold both physical and logical device.

`VulkanRenderer.h`

```
	struct {
		vk::PhysicalDevice physicalDevice;
		vk::Device logicalDevice;
	} mainDevice;
```

Then we will create a `getPhysicalDevice`. We use this name because the physical device materialy exists (it is your GPU). This function will be called when we create the instance :

`VulkanRenderer.cpp`

```
...
int VulkanRenderer::init(GLFWwindow* windowP)
{
	window = windowP;
	try
	{
		createInstance();
		getPhysicalDevice();
	}
	}
	catch {
...
}
```

This function will enemerate the physical devices and store them, and check is the devices are suitable for our future uses. To do that we will need some new functions.

`VulkanRenderer.h`

```
...
private:
	...
	void getPhysicalDevice();
	bool checkDeviceSuitable(vk::PhysicalDevice device);
	QueueFamilyIndices getQueueFamilies(vk::PhysicalDevice device);
};
```

QueueFamilyIndices is a struct that will store for us the indices (locations) of queue families, if they exist. It will check if each family is valid. For now, we will be only interested by the graphics family. Create a new VulkanUtilises header file to declare this struct.

`VulkanUtilities.h`

```
struct QueueFamilyIndices
{
	int graphicsFamily = -1;		// Location of Graphics Queue Family

	bool isValid()
	{
		return graphicsFamily >= 0;
	}
};
```

Now let's write the logic for `getPhysicalDevice`.

`VulkanRenderer.cpp`

```
void VulkanRenderer::getPhysicalDevice()
{
	// Get available physical device
	vector<vk::PhysicalDevice> devices = instance.enumeratePhysicalDevices();

	// If no devices available
	if (devices.size() == 0)
	{
		throw std::runtime_error("Can't find any GPU that supports vulkan");
	}

	// Get device valid for what we want to do
	for (const auto& device : devices)
	{
		if (checkDeviceSuitable(device))
		{
			mainDevice.physicalDevice = device;
			break;
		}
	}
}
```

The `checkDeviceSuitable` will gather the info about the physical device and get its queue families indices. If those indices are valid, it will return true.

`VulkanRenderer.cpp`

```
bool VulkanRenderer::checkDeviceSuitable(vk::PhysicalDevice device)
{
	// Information about the device itself (ID, name, type, vendor, etc.)
	vk::PhysicalDeviceProperties deviceProperties = device.getProperties();

	// Information about what the device can do (geom shader, tesselation, wide lines...)
	vk::PhysicalDeviceFeatures deviceFeatures = device.getFeatures();

	// For now we do nothing with this info

	QueueFamilyIndices indices = getQueueFamilies(device);
	return indices.isValid();
}
```

For now the `getQueueFamilies` function will look for the presence of the graphics queue.

`VulkanRenderer.cpp`

```
QueueFamilyIndices VulkanRenderer::getQueueFamilies(vk::PhysicalDevice device)
{
	QueueFamilyIndices indices;
	vector<vk::QueueFamilyProperties> queueFamilies = device.getQueueFamilyProperties();

	// Go through each queue family and check it has at least one required type of queue
	int i = 0;
	for (const auto& queueFamily : queueFamilies)
	{
		// Check there is at least graphics queue
		if (queueFamily.queueCount > 0 &&
			queueFamily.queueFlags & vk::QueueFlagBits::eGraphics)
		{
			indices.graphicsFamily = i;
		}
		if (indices.isValid()) break;
		++i;
	}
	return indices;
}
```

Now, our `mainDevice.physicalDevice` is populated. We need to create the logical device.

## Logical device

Just add a private `createLogicalDevice` function in the `VulkanRenderer`.

`VulkanRenderer.cpp`

```
...
void VulkanRenderer::createLogicalDevice()
{
	QueueFamilyIndices indices = getQueueFamilies(mainDevice.physicalDevice);

	// Vector for queue creation information, and set for family indices.
	// A set will only keep one indice if they are the same.
	vector<vk::DeviceQueueCreateInfo> queueCreateInfos;
	set<int> queueFamilyIndices = { indices.graphicsFamily, indices.presentationFamily };

	// Queues the logical device needs to create and info to do so.
	for (int queueFamilyIndex : queueFamilyIndices)
	{
		vk::DeviceQueueCreateInfo queueCreateInfo {};
		queueCreateInfo.queueFamilyIndex = queueFamilyIndex;
		queueCreateInfo.queueCount = 1;
		float priority = 1.0f;
		// Vulkan needs to know how to handle multiple queues. It uses priorities.
		// 1 is the highest priority.
		queueCreateInfo.pQueuePriorities = &priority;

		queueCreateInfos.push_back(queueCreateInfo);
	}

	// Logical device creation
	vk::DeviceCreateInfo deviceCreateInfo {};
	// Queues info
	deviceCreateInfo.queueCreateInfoCount = static_cast<uint32_t>(queueCreateInfos.size());
	deviceCreateInfo.pQueueCreateInfos = queueCreateInfos.data();
	// Extensions info
	// Device extensions, different from instance extensions
	deviceCreateInfo.enabledExtensionCount = static_cast<uint32_t>(deviceExtensions.size());
	deviceCreateInfo.ppEnabledExtensionNames = deviceExtensions.data();
	// -- Validation layers are deprecated since Vulkan 1.1
	// Features
	// For now, no device features (tessellation etc.)
	vk::PhysicalDeviceFeatures deviceFeatures {};
	deviceCreateInfo.pEnabledFeatures = &deviceFeatures;
```

```
        // Create the logical device for the given physical device
        mainDevice.logicalDevice = mainDevice.physicalDevice.createDevice(deviceCreateInfo);
}
```

We need to update our `VulkanRenderer::clean` function to delete the logical device.

`VulkanRenderer.cpp`

```
void VulkanRenderer::clean()
{
        mainDevice.logicalDevice.destroy();
        instance.destroy();
}
```

## Access to the graphics queue

With the logical device, we have created a graphics queue. But we do not have access to this queue, and cannot clean it either.

`VulkanRenderer.h`

```
...
private:
        ...
        vk::Queue graphicsQueue;
...
```

`VulkanRenderer.cpp`

```
...
void VulkanRenderer::createLogicalDevice()
{
        ...
        // Ensure access to queues
        graphicsQueue = mainDevice.logicalDevice.getQueue(indices.graphicsFamily, 0);
}
```

# Validation layers

NB: this part of the course is a rewriting from the vulkan tutorial website.

## Introduction

The Vulkan API is designed around the idea of minimal driver overhead and one of the manifestations of that goal is that there is very limited error checking in the API by default. Even mistakes as simple as setting enumerations to incorrect values or passing null pointers to required parameters are generally not explicitly handled and will simply result in crashes or undefined behavior. Because Vulkan requires you to be very explicit about everything you're doing, it's easy to make many small mistakes like using a new GPU feature and forgetting to request it at logical device creation time.

However, that doesn't mean that these checks can't be added to the API. Vulkan introduces an elegant system for this known as validation layers. Validation layers are optional components that hook into Vulkan function calls to apply additional operations. Common operations in validation layers are:

- 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

These validation layers can be freely stacked to include all the debugging functionality that you're interested in. You can simply enable validation layers for debug builds and completely disable them for release builds, which gives you the best of both worlds!

Vulkan does not come with any validation layers built-in, but the LunarG Vulkan SDK provides a nice set of layers that check for common errors. They're also completely open source, so you can check which kind of mistakes they check for and contribute. Using the validation layers is the best way to avoid your application breaking on different drivers by accidentally relying on undefined behavior.

Validation layers can only be used if they have been installed onto the system. For example, the LunarG validation layers are only available on PCs with the Vulkan SDK installed.

There were formerly two different types of validation layers in Vulkan: instance and device specific. The idea was that instance layers would only check calls related to global Vulkan objects like instances, and device specific layers would only check calls related to a specific GPU. Device specific layers have now been deprecated, which means that instance validation layers apply to all Vulkan calls.

## Using validation layers

Let's first add two configuration variables to the vulkan renderer to specify the layers to enable and whether to enable them or not. I've chosen to base that value on whether the program is being compiled in debug mode or not. The NDEBUG macro is part of the C++ standard and means "not debug".

`VulkanRenderer.h`

```
class VulkanRenderer
{
public:
#ifdef NDEBUG
        static const bool enableValidationLayers = false;
#else
        static const bool enableValidationLayers = true;
#endif
        static const vector<const char*> validationLayers;
...
private:
        ...
        bool checkValidationLayerSupport()
        ...
```

`VulkanRenderer.cpp`

```
const vector<const char*> VulkanRenderer::validationLayers {
        "VK_LAYER_KHRONOS_validation"
};
...
```

We'll add a new function checkValidationLayerSupport that checks if all of the requested layers

are available. First list all of the available layers using the vkEnumerateInstanceLayerProperties function. Its usage is identical to that of vkEnumerateInstanceExtensionProperties.

```
bool VulkanRenderer::checkValidationLayerSupport()
{
	vector<vk::LayerProperties> availableLayers = vk::enumerateInstanceLayerProperties();

	// Check if all of the layers in validation layers exist in the available layers
	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;
}
```

We can now use this function in `createInstance` and setup the validation layers. We replace the previous incomplete code:

```
...
        // Validation layers
        if (enableValidationLayers && !checkValidationLayerSupport())
        {
                throw std::runtime_error("validation layers requested, but not available!");
        }
        if (enableValidationLayers)
        {
                createInfo.enabledLayerCount = static_cast<uint32_t>(validationLayers.size());
                createInfo.ppEnabledLayerNames = validationLayers.data();
        }
        else
        {
                createInfo.enabledLayerCount = 0;
                createInfo.ppEnabledLayerNames = nullptr;
        }
...
```

## Message callback

The validation layers will print debug messages to the standard output by default, but we can also handle them ourselves by providing an explicit callback in our program. This will also allow you to decide which kind of messages you would like to see, because not all are necessarily (fatal) errors. If you don't want to do that right now then you may skip to the last section in this chapter.

To set up a callback in the program to handle messages and the associated details, we have to set up a debug messenger with a callback using the `VK_EXT_debug_utils` extension.

We will use the C API (and not the CPP API) so you can discover how it works. In this specific case, it will also simplify a bit the code.

We'll first create a `getRequiredExtensions` function that will return the required list of extensions based on whether validation layers are enabled or not:

`VulkanRenderer.cpp`

```
        uint32_t glfwExtensionCount = 0;
        const char** glfwExtensions;
        glfwExtensions = glfwGetRequiredInstanceExtensions(&glfwExtensionCount);

        vector<const char*> extensions(glfwExtensions, glfwExtensions + glfwExtensionCount);

        if (enableValidationLayers) {
                extensions.push_back(VK_EXT_DEBUG_UTILS_EXTENSION_NAME);
        }

        return extensions;
```

The extensions specified by GLFW are always required, but the debug messenger extension is conditionally added. Note that we've used the VK_EXT_DEBUG_UTILS_EXTENSION_NAME macro here which is equal to the literal string "VK_EXT_debug_utils". Using this macro lets you avoid typos.

We can replace `glfwGetRequiredInstanceExtensions` in `createInstance` by this new function.

```
...
        // Setup extensions instance will use
        vector<const char*> instanceExtensions = getRequiredExtensions();

        // Check instance extensions
...
```

Run the code to check everything is fine.

Now let's see what a debug callback function looks like. Add a new static member function called `debugCallback` with the `PFN_vkDebugUtilsMessengerCallbackEXT` prototype. The `VKAPI_ATTR` and `VKAPI_CALL` ensure that the function has the right signature for Vulkan to call it. We will define this callback with the other utilities.

`VulkanUtilities.h`

```
static VKAPI_ATTR VkBool32 VKAPI_CALL debugCallback(
        VkDebugUtilsMessageSeverityFlagBitsEXT messageSeverity,
        VkDebugUtilsMessageTypeFlagsEXT messageType,
        const VkDebugUtilsMessengerCallbackDataEXT* pCallbackData,
        void* pUserData)
{
        std::cerr << "validation layer: " << pCallbackData->pMessage << std::endl;
        return VK_FALSE;
}
```

The first parameter specifies the severity of the message, which is one of the following flags:

- `VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT`: Diagnostic message
- `VK_DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT`: Informational message like the creation of a resource
- `VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT`: Message about behavior that is not necessarily an error, but very likely a bug in your application
- `VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT`: Message about behavior that is invalid and may cause crashes

The values of this enumeration are set up in such a way that you can use a comparison operation to check if a message is equal or worse compared to some level of severity.

The `messageType` parameter can have the following values:

- `VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT`: Some event has happened that is unrelated to the specification or performance
- `VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT`: Something has happened that violates the specification or indicates a possible mistake
- `VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT`: Potential non-optimal use of Vulkan

The `pCallbackData` parameter refers to a `VkDebugUtilsMessengerCallbackDataEXT` struct containing the details of the message itself, with the most important members being:

- `pMessage`: The debug message as a null-terminated string
- `pObjects`: Array of Vulkan object handles related to the message
- `objectCount`: Number of objects in array

Finally, the `pUserData` parameter contains a pointer that was specified during the setup of the callback and allows you to pass your own data to it.

The callback returns a boolean that indicates if the Vulkan call that triggered the validation layer message should be aborted. If the callback returns true, then the call is aborted with the `VK_ERROR_VALIDATION_FAILED_EXT` error. This is normally only used to test the validation layers themselves, so you should always return `VK_FALSE`.

All that remains now is telling Vulkan about the callback function. Perhaps somewhat surprisingly, even the debug callback in Vulkan is managed with a handle that needs to be explicitly created and destroyed. Such a callback is part of a debug messenger and you can have as many of them as you want. Add a class member for this handle:

`VulkanRenderer.h`

```
...
private:
        ...
        VkDebugUtilsMessengerEXT debugMessenger;
        ...
```

Now add a function `setupDebugMessenger` to be called from `init` right after `createInstance`:

`VulkanRenderer.cpp`

```
int VulkanRenderer::init(GLFWwindow* windowP)
{
        window = windowP;
        try
        {
                createInstance();
                setupDebugMessenger();
                ...
```

We'll need to fill in a `VkDebugUtilsMessengerCreateInfoEXT` structure with details about the messenger and its callback. This struct should be passed to the `vkCreateDebugUtilsMessengerEXT` function to create the VkDebugUtilsMessengerEXT object. Unfortunately, because this function is an extension function, it is not automatically loaded. We

have to look up its address ourselves using `vkGetInstanceProcAddr`. We're going to create our own proxy function that handles this in the background.

`VulkanRenderer.cpp`

```
VkResult VulkanRenderer::createDebugUtilsMessengerEXT(
        VkInstance instance,
        const VkDebugUtilsMessengerCreateInfoEXT* pCreateInfo,
        const VkAllocationCallbacks* pAllocator,
        VkDebugUtilsMessengerEXT* pDebugMessenger)
{
        auto func = (PFN_vkCreateDebugUtilsMessengerEXT)vkGetInstanceProcAddr(
                instance, "vkCreateDebugUtilsMessengerEXT");
        if (func != nullptr)
        {
                return func(instance, pCreateInfo, pAllocator, pDebugMessenger);
        }
        else
        {
                return VK_ERROR_EXTENSION_NOT_PRESENT;
        }
}
```

The `vkGetInstanceProcAddr` function will return nullptr if the function couldn't be loaded. The second to last parameter is again the optional allocator callback that we set to `nullptr`, other than that the parameters are fairly straightforward. Since the debug messenger is specific to our Vulkan instance and its layers, it needs to be explicitly specified as first argument.

The `VkDebugUtilsMessengerEXT` object also needs to be cleaned up with a call to `vkDestroyDebugUtilsMessengerEXT`. Similarly to `vkCreateDebugUtilsMessengerEXT` the function needs to be explicitly loaded.

Create another proxy function right below `createDebugUtilsMessengerEXT` :

`VulkanRenderer.cpp`

```
...
void VulkanRenderer::destroyDebugUtilsMessengerEXT(VkInstance instance,
VkDebugUtilsMessengerEXT debugMessenger, const VkAllocationCallbacks* pAllocator)
{
	auto func = (PFN_vkDestroyDebugUtilsMessengerEXT)vkGetInstanceProcAddr(
		instance,
		"vkDestroyDebugUtilsMessengerEXT"
	);
	if (func != nullptr)
	{
		func(instance, debugMessenger, pAllocator);
	}
}
```

We can call this function in the `VulkanRenderer::clean` function.

`VulkanRenderer.cpp`

```
void VulkanRenderer::clean()
{
	if (enableValidationLayers)
	{
		destroyDebugUtilsMessengerEXT(instance, debugMessenger, nullptr);
	}
	mainDevice.logicalDevice.destroy();
	instance.destroy();
}
```

## Setting up the debug messenger

Now we need to fill the `setupDebugMessenger` function. We will configure the debug message with a specific createInfo, set through a `populateDebugMessengerCreateInfo` function:

`VulkanRenderer.cpp`

```
void VulkanRenderer::populateDebugMessengerCreateInfo(
        vk::DebugUtilsMessengerCreateInfoEXT& createInfo)
{
        createInfo.messageSeverity = vk::DebugUtilsMessageSeverityFlagBitsEXT::eWarning |
                vk::DebugUtilsMessageSeverityFlagBitsEXT::eError;
        createInfo.messageType = vk::DebugUtilsMessageTypeFlagBitsEXT::eGeneral |
                vk::DebugUtilsMessageTypeFlagBitsEXT::eValidation |
                vk::DebugUtilsMessageTypeFlagBitsEXT::ePerformance;
        createInfo.pfnUserCallback = debugCallback;
}
```

We can now write setupDebugMessager:

`VulkanRenderer.cpp`

```
void VulkanRenderer::setupDebugMessenger()
{
        if (!enableValidationLayers) return;

        vk::DebugUtilsMessengerCreateInfoEXT createInfo;
        populateDebugMessengerCreateInfo(createInfo);

        if (instance.createDebugUtilsMessengerEXT(&createInfo, nullptr, &debugMessenger)
                != vk::Result::eSuccess)
        {
                throw std::runtime_error("Failed to set up debug messenger.");
        }
}
```

Test the application to check it works.

## Debugging Instance creation and destruction

Although we've now added debugging with validation layers to the program we're not covering everything quite yet. The `vkCreateDebugUtilsMessengerEXT` call requires a valid instance to have been created and `vkDestroyDebugUtilsMessengerEXT` must be called before the instance is destroyed. This currently leaves us unable to debug any issues in the `vkCreateInstance` and `vkDestroyInstance` calls.

However, there is a way to create a separate debug utils messenger specifically for those two function calls. It requires you to simply pass a pointer to a `VkDebugUtilsMessengerCreateInfoEXT` struct in the `pNext` extension field of `VkInstanceCreateInfo`.

`VulkanRenderer.cpp`

```
...
void VulkanRenderer::createInstance()
{
	...
	// Validation layers
	VkDebugUtilsMessengerCreateInfoEXT debugCreateInfo; // NEW
	if (enableValidationLayers && !checkValidationLayerSupport())
	{
		throw std::runtime_error("validation layers requested, but not available!");
	}
	if (enableValidationLayers)
	{
		createInfo.enabledLayerCount = static_cast<uint32_t>(validationLayers.size());
		createInfo.ppEnabledLayerNames = validationLayers.data();
		populateDebugMessengerCreateInfo(debugCreateInfo); // NEW
		createInfo.pNext = (VkDebugUtilsMessengerCreateInfoEXT*)&debugCreateInfo; // NEW
	}
	else
	{
		createInfo.enabledLayerCount = 0;
		createInfo.ppEnabledLayerNames = nullptr;
	}
	...
```

The `debugCreateInfo` variable is placed outside the `if` statement to ensure that it is not destroyed before the `vkCreateInstance` call. By creating an additional debug messenger this way it will automatically be used during `vkCreateInstance` and `vkDestroyInstance` and cleaned up after that.

If you want to test the logging, comment out `destroyDebugUtilsMessengerEXT(instance, debugMessenger, nullptr);` in the

`VulkanRenderer::clean` function. Note that you may have messages about layer manifestos. This comes from a graphics driver to be updated.

# Surfaces, Image Views and the Swapchain

## Concepts

Swapchains and Images are extensions of Vulkan.

A Swapchain handles the retrieval of images to be swapped and images display. It is used in double or triple buffering.

Surfaces are an interface between the window and an vulkan image in the swapchain. Surface will be created specifically for the window of the system we're using. GLFW will handle it for us.

In order to present a new swapchain image to our surface, we need a queue, the presentation queue. Being a presentation queue is a feature, not a type. The graphics queue IS a presentation queue.

The swapchain is an extension. It is a group of images that can be drawn, then presented when ready. The downside of it is it requires a lot of synchronisation. 3 steps to create a swapchain :

- Get the surface capabilities, e.g. image size.
- Get the surface format, e.g. RGB.
- Presentation mode : The order and timing of images being presented to the surface.

The presentation mode are the order and timing of images being presented to the surface. There can be only one image present at the surface to be drawn on the screen. There are 4 presentation modes, with 2 of them leading to tearing:

- vk::PresentModeKHR::eImmediate : the surface image will be replaced immediately after the next image to be displayed has finished rendering. Causes tearing.
- vk::PresentModeKHR::eMailbox : images ready to present are added to a queue of size 1. The surface uses this image at Vertical Blank Interval. No tearing. If a new

image is sent to the queue, it replaces the image currently in the queue. This is what is used with triple-buffering.
- vk::PresentModeKHR::eFifo : works the same, but the size is configurable and we present the images in order. No tearing. If the queue is full, the program will wait. When the queue is empty, it redraws the same image.
- vk::PresentModeKHR::eFifoRelaxed : same as above, except when the queue is empty, it switch to immediate mode and draw the incomplete image that will be added to the queue. Causes tearing.

When the swapchain is created, it will automatically create a set of images to be used by the swapchain. We need access to those images, therefore the swapchain can be queried and an array of images can be returned. The problem is those images are raw data and cannot be directly used.

An ImageView can be created to interface with an Image. It describes how to read an Image (2D or 3D adresses, format...) and what part of the image to read (colour channels, mip levels, etc.).

## The Surface

Create a surface member, a presentation queue member and a `createSurface` function:

`VulkanRenderer.f`

```
...
        vk::SurfaceKHR surface;
        vk::Queue presentationQueue;
        ...
        vk::SurfaceKHR createSurface();
...
```

Call this last function in the initialization:

`VulkanRenderer.h`

```
...
	try
	{
		createInstance();
		setupDebugMessenger();
		surface = createSurface(); // Before physical device
		getPhysicalDevice();
		createLogicalDevice();
	}
...
```

`VulkanRenderer.cpp`

```
void VulkanRenderer::createSurface()
{
	// Create a surface relatively to our window
	VkSurfaceKHR _surface;

	VkResult result = glfwCreateWindowSurface(instance, window, nullptr, &_surface);
	if (result != VK_SUCCESS)
	{
		throw std::runtime_error("Failed to create a vulkan surface.");
	}

	return vk::SurfaceKHR(_surface);
}
```

Update the `clean` function:

```
void VulkanRenderer::clean()
{
	instance.destroySurfaceKHR(surface);
	...
}
```

We now want to check the presentation family is working. We stert by updating the `QueueFamilyIndices` struct:

`VulkanUtilities.h`

```
struct QueueFamilyIndices
{
	int graphicsFamily = -1;	// Location of Graphics Queue Family
	int presentationFamily = -1; // Location of Presentation Queue Family

	bool isValid()
	{
		return graphicsFamily >= 0 && presentationFamily >= 0;
	}
};
```

Now we update the `getQueueFamilies` function:

`VulkanRenderer.cpp`

```
QueueFamilyIndices VulkanRenderer::getQueueFamilies(VkPhysicalDevice device)
{
	...
	// Go through each queue family and check it has at least one required type of queue
	int i = 0;
	for (const auto& queueFamily : queueFamilies)
	{
		// Check there is at least graphics queue
		if (queueFamily.queueCount > 0
			&& queueFamily.queueFlags & vk::QueueFlagBits::eGraphics)
		{
			indices.graphicsFamily = i;
		}

		// Check if queue family support presentation
		VkBool32 presentationSupport = device.getSurfaceSupportKHR(
			static_cast<uint32_t>(indices.graphicsFamily),
			surface
		);
		if (queueFamily.queueCount > 0 && presentationSupport)
		{
			indices.presentationFamily = i;
		}

		if (indices.isValid()) break;
		++i;
	}
	return indices;
}
```

Because we will now support multiple queues - at least the graphics and the presentation queue, we need to update the `createLogicalDevice` function.

`VulkanRenderer.cpp`

```
void VulkanRenderer::createLogicalDevice()
{
	QueueFamilyIndices indices = getQueueFamilies(mainDevice.physicalDevice);

	// Vector for queue creation information, and set for family indices.
	// A set will only keep one indice if they are the same.
	vector<vk::DeviceQueueCreateInfo> queueCreateInfos;
	set<int> queueFamilyIndices = { indices.graphicsFamily, indices.presentationFamily };

	// Queues the logical device needs to create and info to do so.
	for (int queueFamilyIndex : queueFamilyIndices)
	{
		vk::DeviceQueueCreateInfo queueCreateInfo {};
		queueCreateInfo.queueFamilyIndex = queueFamilyIndex;
		queueCreateInfo.queueCount = 1;
		float priority = 1.0f;
		// Vulkan needs to know how to handle multiple queues. It uses priorities.
		// 1 is the highest priority.
		queueCreateInfo.pQueuePriorities = &priority;

		queueCreateInfos.push_back(queueCreateInfo);
	}

	// Logical device creation
	vk::DeviceCreateInfo deviceCreateInfo {};
	// Queues info
	deviceCreateInfo.queueCreateInfoCount = static_cast<uint32_t>(queueCreateInfos.size());
	deviceCreateInfo.pQueueCreateInfos = queueCreateInfos.data();
	// Extensions info
	// Device extensions, different from instance extensions
	deviceCreateInfo.enabledExtensionCount = static_cast<uint32_t>(deviceExtensions.size());
	deviceCreateInfo.ppEnabledExtensionNames = deviceExtensions.data();
	// -- Validation layers are deprecated since Vulkan 1.1
	// Features
	// For now, no device features (tessellation etc.)
	vk::PhysicalDeviceFeatures deviceFeatures {};
	deviceCreateInfo.pEnabledFeatures = &deviceFeatures;

	// Create the logical device for the given physical device
```

```
        mainDevice.logicalDevice = mainDevice.physicalDevice.createDevice(deviceCreateInfo);

        // Ensure access to queues
        graphicsQueue = mainDevice.logicalDevice.getQueue(indices.graphicsFamily, 0);
        presentationQueue = mainDevice.logicalDevice.getQueue(indices.presentationFamily, 0);
}
```

We want to check the physical device supports extensions. In our case, we want the swapchain extension to be supported.

We list the extensions to support in VulkanUtilities:

`VulkanUtilities.cpp`

```
const vector<const char*> deviceExtensions
{
        VK_KHR_SWAPCHAIN_EXTENSION_NAME
};
...
```

Create a `checkDeviceExtensionSupport` function to check extensions are supported:

`VulkanRenderer.cpp`

```
bool VulkanRenderer::checkDeviceExtensionSupport(vk::PhysicalDevice device)
{
	vector<vk::ExtensionProperties> extensions = device.enumerateDeviceExtensionProperties();

	for (const auto& deviceExtension : deviceExtensions)
	{
		bool hasExtension = false;
		for (const auto& extension : extensions)
		{
			if (strcmp(deviceExtension, extension.extensionName) == 0)
			{
				hasExtension = true;
				break;
			}
		}

		if (!hasExtension) return false;
	}

	return true;
}
```

We will consider our device suitable if extensions are supported. So let's update `checkDeviceSuitable`:

`VulkanRenderer.cpp`

```
bool VulkanRenderer::checkDeviceSuitable(vk::PhysicalDevice device)
{
	...
	QueueFamilyIndices indices = getQueueFamilies(device);
	bool extensionSupported = checkDeviceExtensionSupport(device);

	return indices.isValid() && extensionSupported;
}
```

Now we are sure the physical devices support the swapchain extension, we can set the enabled extensions in the logical device:

`VulkanRenderer.cpp`

```
void VulkanRenderer::createLogicalDevice()
{
	...
	// Extensions info
	// Device extensions, different from instance extensions
	deviceCreateInfo.enabledExtensionCount = static_cast<uint32_t>(deviceExtensions.size());
	deviceCreateInfo.ppEnabledExtensionNames = deviceExtensions.data();
	...
}
```

# The Swapchain

## Get the surface info to configure the swapchain

We now need to make sure we will create a swapchain that will be compatible with the surface we have. So we need to get the surface information to see what is supported, and create the surface in function of this information.

First, create a new struct to store swapchains:

`VulkanUtilities.cpp`

```
struct SwapchainDetails {
	// What the surface is capable of displaying, e.g. image size/extent
	vk::SurfaceCapabilitiesKHR surfaceCapabilities;
	// Vector of the image formats, e.g. RGBA
	vector<vk::SurfaceFormatKHR> formats;
	// Vector of presentation modes
	vector<vk::PresentModeKHR> presentationModes;
};
```

Now create a function to populate this struct:

`VulkanRenderer.cpp`

```
SwapchainDetails VulkanRenderer::getSwapchainDetails(vk::PhysicalDevice device)
{
	SwapchainDetails swapchainDetails;
	// Capabilities
	swapchainDetails.surfaceCapabilities = device.getSurfaceCapabilitiesKHR(surface);
	// Formats
	swapchainDetails.formats = device.getSurfaceFormatsKHR(surface);
	// Presentation modes
	swapchainDetails.presentationModes = device.getSurfacePresentModesKHR(surface);

	return swapchainDetails;
}
```

It is possible the Swapchain is not working, and thus there will be no available format or presentation modes. We will update checkDeviceSuitable to take this case into account.

`VulkanRenderer.cpp`

```
bool VulkanRenderer::checkDeviceSuitable(vk::PhysicalDevice device)
{
	...
	QueueFamilyIndices indices = getQueueFamilies(device);
	bool extensionSupported = checkDeviceExtensionSupport(device);

	bool swapchainValid = false;
	if (extensionSupported)
	{
		SwapchainDetails swapchainDetails = getSwapchainDetails(device);
		swapchainValid = !swapchainDetails.presentationModes.empty()
				&& !swapchainDetails.formats.empty();
	}

	return indices.isValid() && extensionSupported && swapchainValid;
}
```

## Create the Swapchain

We will use a `createSwapchain` function and we need a `vk::SwapchainKHR swapchain` member.

First, we will pick what we consider best settings from the physical device.

`VulkanRenderer.cpp`

```
void VulkanRenderer::createSwapchain()
{
  // We will pick best settings for the swapchain
  SwapchainDetails swapchainDetails = getSwapchainDetails(mainDevice.physicalDevice);
  vk::SurfaceFormatKHR surfaceFormat = chooseBestSurfaceFormat(swapchainDetails.formats);
  vk::PresentModeKHR presentationMode = chooseBestPresentationMode(swapchainDetails.presentationMo
  vk::Extent2D extent = chooseSwapExtent(swapchainDetails.surfaceCapabilities);

  // To be continued
}
```

We create the associated functions:

`VulkanRenderer.cpp`

```
vk::SurfaceFormatKHR VulkanRenderer::chooseBestSurfaceFormat(
        const vector<vk::SurfaceFormatKHR>& formats)
{
        // We will use RGBA 32bits normalized and SRGG non linear colorspace
        if (formats.size() == 1 && formats[0].format == vk::Format::eUndefined)
        {
                // All formats available by convention
                return { vk::Format::eR8G8B8A8Unorm, vk::ColorSpaceKHR::eSrgbNonlinear };
        }

        for (auto& format : formats)
        {
                if (format.format == vk::Format::eR8G8B8A8Unorm
                                && format.colorSpace == vk::ColorSpaceKHR::eSrgbNonlinear)
                {
                        return format;
                }
        }

        // Return first format if we have not our chosen format
        return formats[0];
}

vk::PresentModeKHR VulkanRenderer::chooseBestPresentationMode(
        const vector<vk::PresentModeKHR>& presentationModes)
{
        // We will use mail box presentation mode
        for (const auto& presentationMode : presentationModes)
        {
                if (presentationMode == vk::PresentModeKHR::eMailbox)
                {
                        return presentationMode;
                }
        }

        // Part of the Vulkan spec, so have to be available
        return vk::PresentModeKHR::eFifo;
}
```

```
vk::Extent2D VulkanRenderer::chooseSwapExtent(
	const vk::SurfaceCapabilitiesKHR& surfaceCapabilities)
{
	// Rigid extents
	if (surfaceCapabilities.currentExtent.width != std::numeric_limits<uint32_t>::max())
	{
		return surfaceCapabilities.currentExtent;
	}
	// Extents can vary
	else
	{
		// Create new extent using window size
		int width, height;
		glfwGetFramebufferSize(window, &width, &height);
		vk::Extent2D newExtent {};
		newExtent.width = static_cast<uint32_t>(width);
		newExtent.height = static_cast<uint32_t>(height);

		// Sarface also defines max and min, so make sure we are within boundaries
		newExtent.width = std::max(surfaceCapabilities.minImageExtent.width,
			std::min(surfaceCapabilities.maxImageExtent.width, newExtent.width));
		newExtent.height = std::max(surfaceCapabilities.minImageExtent.height,
			std::min(surfaceCapabilities.maxImageExtent.height, newExtent.height));
		return newExtent;
	}
}
```

Now, we can setup the swapchain's create info:

`VulkanRenderer.cpp`

```
void VulkanRenderer::createSwapchain()
{
...
        // Setup the swap chain info
        vk::SwapchainCreateInfoKHR swapchainCreateInfo {};
        swapchainCreateInfo.surface = surface;
        swapchainCreateInfo.imageFormat = surfaceFormat.format;
        swapchainCreateInfo.imageColorSpace = surfaceFormat.colorSpace;
        swapchainCreateInfo.presentMode = presentationMode;
        swapchainCreateInfo.imageExtent = extent;
        // Minimal number of image in our swapchain. We will use one
        // more than the minimum to enable triple-buffering.
        uint32_t imageCount = swapchainDetails.surfaceCapabilities.minImageCount + 1;
        if (swapchainDetails.surfaceCapabilities.maxImageCount > 0 // Not limitless
                && swapchainDetails.surfaceCapabilities.maxImageCount < imageCount)
        {
                imageCount = swapchainDetails.surfaceCapabilities.maxImageCount;
        }
        swapchainCreateInfo.minImageCount = imageCount;
        // Number of layers for each image in swapchain
        swapchainCreateInfo.imageArrayLayers = 1;
        // What attachment go with the image (e.g. depth, stencil...). Here, just color.
        swapchainCreateInfo.imageUsage = vk::ImageUsageFlagBits::eColorAttachment;
        // Transform to perform on swapchain images
        swapchainCreateInfo.preTransform = swapchainDetails.surfaceCapabilities.currentTransform;
        // Handles blending with other windows. Here we don't blend.
        swapchainCreateInfo.compositeAlpha = vk::CompositeAlphaFlagBitsKHR::eOpaque;
        // Whether to clip parts of the image not in view (e.g. when an other window overlaps)
        swapchainCreateInfo.clipped = VK_TRUE;

        // To be continued
}
```

Images in our swapchain will be used by the graphics queue then by the presentation queue. We have to define if images will be exclusively used by a queue, are shared between both. Here, we can remember we are actually using the same queue, so we don't need sharing.

We will also setup last details and create the swapchain.

`VulkanRenderer.cpp`

```
void VulkanRenderer::createSwapchain()
{
	...
	// Queue management
	QueueFamilyIndices indices = getQueueFamilies(mainDevice.physicalDevice);
	uint32_t queueFamilyIndices[]{ (uint32_t)indices.graphicsFamily,
		(uint32_t)indices.presentationFamily };
	// If graphics and presentation families are different, share images between them
	if (indices.graphicsFamily != indices.presentationFamily)
	{
		swapchainCreateInfo.imageSharingMode = vk::SharingMode::eConcurrent;
		swapchainCreateInfo.queueFamilyIndexCount = 2;
		swapchainCreateInfo.pQueueFamilyIndices = queueFamilyIndices;
	}
	else
	{
		swapchainCreateInfo.imageSharingMode = vk::SharingMode::eExclusive;
		swapchainCreateInfo.queueFamilyIndexCount = 0;
		swapchainCreateInfo.pQueueFamilyIndices = nullptr;
	}

	// When you want to pass old swapchain responsibilities when destroying it,
	// e.g. when you want to resize window, use this
	swapchainCreateInfo.oldSwapchain = VK_NULL_HANDLE;

	// Create swapchain
	swapchain = mainDevice.logicalDevice.createSwapchainKHR(swapchainCreateInfo);
}
```

We also need to create the swapchain at initialization and delete it when the program closes.

`VulkanRenderer.cpp`

```
...
int VulkanRenderer::init(GLFWwindow* windowP)
{
	window = windowP;
	try
	{
		createInstance();
		setupDebugMessenger();
		createSurface();
		getPhysicalDevice();
		createLogicalDevice();
		createSwapchain();
	}
	...
}

void VulkanRenderer::clean()
{
	mainDevice.logicalDevice.destroySwapchainKHR(swapchain);
	instance.destroySurfaceKHR(surface);
	if (enableValidationLayers)
	{
		destroyDebugUtilsMessengerEXT(instance, debugMessenger, nullptr);
	}
	mainDevice.logicalDevice.destroy();
	instance.destroy();
}
```

By the way, we want to store the swapchain image format and extents.

`VulkanRenderer.h`

```
	vk::Format swapchainImageFormat;
	vk::Extent2D swapchainExtent;
```

Add this at the end of the `createSwapchain` function:

```
...
        // Store for later use
        swapchainImageFormat = surfaceFormat.format;
        swapchainExtent = extent;
```

## Image Views

Now we want to get the images out of the swapchain. The same manner we get the physical device and create the logical device, we will get the swapchain Image and create the ImageView, that will allow us to handle the image.

Create a new struct in the utilities:

`VulkanUtilities.h`

```
struct SwapchainImage
{
        vk::Image image;
        vk::ImageView imageView;
};
```

Also create a vector of `SwapchainImage` in the header:

`VulkanRenderer.h`

```
        vector<SwapchainImage> swapchainImages;
```

Add the following at the end of the `createSwapchain` function:

`VulkanRenderer.cpp`

```
void VulkanRenderer::createSwapchain()
{
	...
	// Get the swapchain images
	vector<vk::Image> images = mainDevice.logicalDevice.getSwapchainImagesKHR(swapchain);
	for (VkImage image : images)	// We are using handles, not values
	{
		SwapchainImage swapchainImage {};
		swapchainImage.image = image;

		// Create image view
		swapchainImage.imageView = createImageView(image, swapchainImageFormat,
			vk::ImageAspectFlagBits::eColor);

		swapchainImages.push_back(swapchainImage);
	}
}
```

We need a new `createImageView` function:

`VulkanRenderer.cpp`

```
vk::ImageView VulkanRenderer::createImageView(
        vk::Image image, vk::Format format, vk::ImageAspectFlagBits aspectFlags)
{
        vk::ImageViewCreateInfo viewCreateInfo {};
        viewCreateInfo.image = image;
        // Other formats can be used for cubemaps etc.
        viewCreateInfo.viewType = vk::ImageViewType::e2D;
        // Can be used for depth for instance
        viewCreateInfo.format = format;
        // Swizzle used to remap color values. Here we keep the same.
        viewCreateInfo.components.r = vk::ComponentSwizzle::eIdentity;
        viewCreateInfo.components.g = vk::ComponentSwizzle::eIdentity;
        viewCreateInfo.components.b = vk::ComponentSwizzle::eIdentity;
        viewCreateInfo.components.a = vk::ComponentSwizzle::eIdentity;

        // Subresources allow the view to view only a part of an image
        // Here we want to see the image under the aspect of colors
        viewCreateInfo.subresourceRange.aspectMask = aspectFlags;
        // Start mipmap level to view from
        viewCreateInfo.subresourceRange.baseMipLevel = 0;
        // Number of mipmap level to view
        viewCreateInfo.subresourceRange.levelCount = 1;
        // Start array level to view from
        viewCreateInfo.subresourceRange.baseArrayLayer = 0;
        // Number of array levels to view
        viewCreateInfo.subresourceRange.layerCount = 1;

        // Create image view
        vk::ImageView imageView = mainDevice.logicalDevice.createImageView(viewCreateInfo);
        return imageView;
}
```

Do not forget to clean the created image views:

`VulkanRenderer.cpp`

```
void VulkanRenderer::clean()
{
        for (auto image : swapchainImages)
        {
                mainDevice.logicalDevice.destroyImageView(image.imageView);
        }
        ...
```

That's all for the basics ! In the next lesson we will setup the pipeline and finally get an image on our screen. If you want additional or comparative information about this lesson, please use the Vulkan Tutorial.