Vulkan graphics pipeline

Pipeline and shaders concepts

The graphics pipeline is largely the same as in OpenGL. We will have to setup each stage of the pipeline ourselves. The rendering part is more complexe.

The pipeline stages are :

- Input assembler
- Vertex shader
- Tessellation
- Geometry shader
- Rasterization
- Fragment shader
- Color blending
- Then final send to frame buffer.

Unlike OpenGL, we won't load shaders from text and compile them. Instead we will pre-compile them to intermediate code in the **SPIR-V** (standard portable intermediate representaion - vulkan) format. This format can be compiled from GLSL. The LunarG tool glslangValidator.exe can perform the compilation. In our programs, we will load .spv binary files and create a shader module using Vulkan. The shader module will be passed to a create info struct, all those structs into a list that will be used by the pipeline create info.

Pipeline are statics, so we have to define them precisely. The settings we will have to set for the pipeline will be:

- **Vertex input** : defines layout and format of vertex input data
- **Input assembly** : how to assemble vertices to primitives (e.g. points, lines, triangles...)
- **Viewport and scissor** : how to fit output to image and crop it
- **Dynamic states**: set what CAN be changed at runtime in the static pipeline. E.g. viewport, when we want to change size of screen.
- **Rasterizer**: computation of fragment from primitives
- **Multisampling**: e.g. for anti-aliasing

- **Blending**: how to blend fragments at the end of the pipeline, e.g. for transparency
- **Depth stencil**: how to determine depth + stencil culling and writing

We also have to define the **pipeline layout**: data given directly to the pipeline for a single draw operation (as opposed to for each vertex or fragment). In OpenGL, this type of data was called uniform buffers, in Vulkan they are called **descriptor sets**. Descriptor sets have a specific location in memory. We can also pass **push constants** through the pipeline ; they are smaller values that are not held in a specific memory location.

A **Render Pass** is a larger operation that handles the execution and outputs of a pipeline. It can have multiple smaller subpasses, so draws can be combined together. Each subpass can use a different pipeline. A render pass can hold multiple **Attachments** to all the possible outputs (e.g. color output, depth output...). Subpasses inside a render pass reference attachments for each draw operation. Subpasses rely on strict ordering to ensure data is in the right format at the right time.

Let's think back to our swapchain. The swapchain image needs to be in a writable format at the stage of the pipeline's subpass that writes to the image. When we present this image, it will need to be in a presentable format. How to ensure the right format at the right time ? **Subpass dependencies** define stages in a pipeline where transitions need to occur. The transition is implicit: we do not state the transition that needs to take place. We say WHEN it should occur. The layout of each subpass will tell Vulkan how to transition.

Shader modules

Shader compilatio to SPIR-V

We will create shaders, compile them and send them to usage. Create a `shader` folder in your VS project folder and create a `shader.vert` and `shader.frag` files.

`shader.vert`

```
#version 450

// Output colors for vertex shader
layout(location = 0) out vec3 fragColor;

// Triangle vertex positions
// (will be put to vertex buffer later)
vec3 positions[3] = vec3[](
    vec3(0.0, -0.4, 0.0),
    vec3(0.4, 0.4, 0.0),
    vec3(-0.4, 0.4, 0.0)
);

// Colors
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], 1.0);
    fragColor = colors[gl_VertexIndex];
}
```

`shader.frag`

```
#version 450

// Connects to the output of vertex shader.
// Interpolated color from vertex shader.
layout(location = 0) in vec3 fragColor;

// Final output color, must have location
layout(location = 0) out vec4 outColor;

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

We want to execute C:\VulkanSDK[version]\Bin\glslangValidator.exe to compile those shaders to SPIR-V. We will create a batch file for that.

`compileShaders.bat`

```
C:/VulkanSDK/1.3.224.1/Bin/glslangValidator.exe -V shader.vert
C:/VulkanSDK/1.3.224.1/Bin/glslangValidator.exe -V shader.frag
pause
```

Execute the batch script to compile. you will eventually be signaled errors.

Using compiled shaders

First, we will need a utility function to read the shaders.

`VulkanUtilities.h`

```
static vector<char> readShaderFile(const string& filename)
{
	// Open shader file
	// spv files are binary data, put the pointer at the end of the file to get its size
	std::ifstream file { filename, std::ios::binary | std::ios::ate };
	if (!file.is_open())
	{
		throw std::runtime_error("Failed to open a file");
	}

	// -- Buffer preparation
	// Get the size through the position of the pointer
	size_t fileSize = (size_t)file.tellg();
	// Set file buffer to the file size
	vector<char> fileBuffer(fileSize);
	// Move in file to start of the file
	file.seekg(0);

	// Reading and closing
	file.read(fileBuffer.data(), fileSize);
	file.close();
	return fileBuffer;
}
```

We will now create a `createGraphicsPipeline` function to create the pipeline. Don't forget to call it in the initialization.

`VulkanRenderer.cpp`

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

Then we will create the basic structure for the pipeline creation. It starts with the load of shaders.

`VulkanRenderer.cpp`

```
void VulkanRenderer::createGraphicsPipeline()
{
	// Read shader code and format it through a shader module
	auto vertexShaderCode = readShaderFile("shaders/vert.spv");
	auto fragmentShaderCode = readShaderFile("shaders/frag.spv");
	vk::ShaderModule vertexShaderModule = createShaderModule(vertexShaderCode);
	vk::ShaderModule fragmentShaderModule = createShaderModule(fragmentShaderCode);


	// -- SHADER STAGE CREATION INFO --
	// Vertex stage creation info
	vk::PipelineShaderStageCreateInfo vertexShaderCreateInfo{};
	// Used to know which shader
	vertexShaderCreateInfo.stage = vk::ShaderStageFlagBits::eVertex;
	vertexShaderCreateInfo.module = vertexShaderModule;
	// Pointer to the start function in the shader
	vertexShaderCreateInfo.pName = "main";
	// Fragment stage creation info
	vk::PipelineShaderStageCreateInfo fragmentShaderCreateInfo{};
	fragmentShaderCreateInfo.stage = vk::ShaderStageFlagBits::eFragment;
	fragmentShaderCreateInfo.module = fragmentShaderModule;
	fragmentShaderCreateInfo.pName = "main";
	// Graphics pipeline requires an array of shader create info
	vk::PipelineShaderStageCreateInfo shaderStages[] {
		vertexShaderCreateInfo, fragmentShaderCreateInfo };

	// Create pipeline


	// Destroy shader modules
	mainDevice.logicalDevice.destroyShaderModule(fragmentShaderModule);
	mainDevice.logicalDevice.destroyShaderModule(vertexShaderModule);
}

VkShaderModule VulkanRenderer::createShaderModule(const vector<char>& code)
{
	vk::ShaderModuleCreateInfo shaderModuleCreateInfo{};
	shaderModuleCreateInfo.codeSize = code.size();
	// Conversion between pointer types with reinterpret_cast
```

```
        shaderModuleCreateInfo.pCode = reinterpret_cast<const uint32_t*>(code.data());

        vk::ShaderModule shaderModule =
                mainDevice.logicalDevice.createShaderModule(shaderModuleCreateInfo);
        return shaderModule;
}
```

Setup the pipeline

Various stages

We will now implement different stages of the pipeline. The comments in the code will explicit the use of those different stages. Some of them will be implemented later. We will stop at the render passes stage, that will need a specific paragraph. This code is meant to exist in the `createPipeline` function.

```
...
// Create pipeline

// -- VERTEX INPUT STAGE --
// TODO: Put in vertex description when resources created
vk::PipelineVertexInputStateCreateInfo vertexInputCreateInfo{};
vertexInputCreateInfo.vertexBindingDescriptionCount = 0;
// List of vertex binding desc. (data spacing, stride...)
vertexInputCreateInfo.pVertexBindingDescriptions = nullptr;
vertexInputCreateInfo.vertexAttributeDescriptionCount = 0;
// List of vertex attribute desc. (data format and where to bind to/from)
vertexInputCreateInfo.pVertexAttributeDescriptions = nullptr;


// -- INPUT ASSEMBLY --
vk::PipelineInputAssemblyStateCreateInfo inputAssemblyCreateInfo{};
// How to assemble vertices
inputAssemblyCreateInfo.topology = vk::PrimitiveTopology::eTriangleList;
// When you want to restart a primitive, e.g. with a strip
inputAssemblyCreateInfo.primitiveRestartEnable = VK_FALSE;


// -- VIEWPORT AND SCISSOR --
// Create a viewport info struct
vk::Viewport viewport{};
viewport.x = 0.0f;                // X start coordinate
viewport.y = 0.0f;                // Y start coordinate
viewport.width = (float)swapchainExtent.width;                // Width of viewport
viewport.height = (float)swapchainExtent.height;        // Height of viewport
viewport.minDepth = 0.0f;                                  // Min framebuffer depth
viewport.maxDepth = 1.0f;                                  // Max framebuffer depth

// Create a scissor info struct, everything outside is cut
vk::Rect2D scissor{};
scissor.offset = vk::Offset2D { 0, 0 };
scissor.extent = swapchainExtent;

vk::PipelineViewportStateCreateInfo viewportStateCreateInfo{};
viewportStateCreateInfo.viewportCount = 1;
```

```
viewportStateCreateInfo.pViewports = &viewport;
viewportStateCreateInfo.scissorCount = 1;
viewportStateCreateInfo.pScissors = &scissor;

// -- DYNAMIC STATE --
// This will be alterable, so you don't have to create an entire pipeline when you want to
// parameters. We won't use this feature, this is an example.
/*
vector<vk::DynamicState> dynamicStateEnables;
// Viewport can be resized in the command buffer
// with vkCmdSetViewport(commandBuffer, 0, 1, &newViewport);
dynamicStateEnables.push_back(vk::DynamicState::eViewport);
// Scissors can be resized in the command buffer
// with vkCmdSetScissor(commandBuffer, 0, 1, &newScissor);
dynamicStateEnables.push_back(vk::DynamicState::eScissor);

vk::PipelineDynamicStateCreateInfo dynamicStateCreateInfo{};
dynamicStateCreateInfo.dynamicStateCount =
        static_cast<uint32_t>(dynamicStateEnables.size());
dynamicStateCreateInfo.pDynamicStates = dynamicStateEnables.data();
*/



// -- RASTERIZER --
vk::PipelineRasterizationStateCreateInfo rasterizerCreateInfo{};
// Treat elements beyond the far plane like being on the far place,
// needs a GPU device feature
rasterizerCreateInfo.depthClampEnable = VK_FALSE;
// Whether to discard data and skip rasterizer. When you want
// a pipeline without framebuffer.
rasterizerCreateInfo.rasterizerDiscardEnable = VK_FALSE;
// How to handle filling points between vertices. Here, considers things inside
// the polygon as a fragment. VK_POLYGON_MODE_LINE will consider element inside
// polygones being empty (no fragment). May require a device feature.
rasterizerCreateInfo.polygonMode = vk::PolygonMode::eFill;
// How thick should line be when drawn
rasterizerCreateInfo.lineWidth = 1.0f;
// Culling. Do not draw back of polygons
rasterizerCreateInfo.cullMode = vk::CullModeFlagBits::eBack;
```

```
// Widing to know the front face of a polygon
rasterizerCreateInfo.frontFace = vk::FrontFace::eClockwise;
// Whether to add a depth offset to fragments. Good for stopping
// "shadow acne" in shadow mapping. Is set, need to set 3 other values.
rasterizerCreateInfo.depthBiasEnable = VK_FALSE;


// -- MULTISAMPLING --
// Not for textures, only for edges
vk::PipelineMultisampleStateCreateInfo multisamplingCreateInfo{};
// Enable multisample shading or not
multisamplingCreateInfo.sampleShadingEnable = VK_FALSE;
// Number of samples to use per fragment
multisamplingCreateInfo.rasterizationSamples = vk::SampleCountFlagBits::e1;

// -- BLENDING --
// How to blend a new color being written to the fragment, with the old value
vk::PipelineColorBlendStateCreateInfo colorBlendingCreateInfo{};
// Alternative to usual blending calculation
colorBlendingCreateInfo.logicOpEnable = VK_FALSE;

// Enable blending and choose colors to apply blending to
vk::PipelineColorBlendAttachmentState colorBlendAttachment{};
colorBlendAttachment.colorWriteMask = vk::ColorComponentFlagBits::eR |
        vk::ColorComponentFlagBits::eG | vk::ColorComponentFlagBits::eB |
        vk::ColorComponentFlagBits::eA;
colorBlendAttachment.blendEnable = VK_TRUE;

// Blending equation:
// (srcColorBlendFactor * new color) colorBlendOp (dstColorBlendFactor * old color)
colorBlendAttachment.srcColorBlendFactor = vk::BlendFactor::eSrcAlpha;
colorBlendAttachment.dstColorBlendFactor = vk::BlendFactor::eOneMinusSrcAlpha;
colorBlendAttachment.colorBlendOp = vk::BlendOp::eAdd;

// Replace the old alpha with the new one: (1 * new alpha) + (0 * old alpha)
colorBlendAttachment.srcAlphaBlendFactor = vk::BlendFactor::eOne;
colorBlendAttachment.dstAlphaBlendFactor = vk::BlendFactor::eZero;
colorBlendAttachment.alphaBlendOp = vk::BlendOp::eAdd;
```

```
        colorBlendingCreateInfo.attachmentCount = 1;
        colorBlendingCreateInfo.pAttachments = &colorBlendAttachment;



        // -- PIPELINE LAYOUT --
        // TODO: apply future descriptorset layout
        vk::PipelineLayoutCreateInfo pipelineLayoutCreateInfo{};
        pipelineLayoutCreateInfo.setLayoutCount = 0;
        pipelineLayoutCreateInfo.pSetLayouts = nullptr;
        pipelineLayoutCreateInfo.pushConstantRangeCount = 0;
        pipelineLayoutCreateInfo.pPushConstantRanges = nullptr;

        // Create pipeline layout
        pipelineLayout = mainDevice.logicalDevice.createPipelineLayout(pipelineLayoutCreateInfo);



        // -- DEPTH STENCIL TESTING --
        // TODO: Set up depth stencil testing


        // -- PASSES --
        // Passes are composed of a sequence of subpasses that can pass data from one to another



        // Destroy shader modules
        ...
```

The layout part imply we create a specific variable, and clean its content when the rederer closes.

`VulkanRenderer.h`

```
        ...
        vk::PipelineLayout pipelineLayout;
        ...
```

`VulkanRenderer.cpp`

```
        ...
        void VulkanRenderer::clean()
        {
                mainDevice.logicalDevice.destroyPipelineLayout(pipelineLayout);
        ...
```

Render passes

We we now create a simple render pass with only one subpass. Explanations in commentaries.

First, we need to store (and destroy) a render pass object.

`VulkanRenderer.h`

```
        ...
        void createRenderPass();
        vk::RenderPass renderPass;
        ...
```

`VulkanRenderer.cpp`

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

void VulkanRenderer::clean()
{
	mainDevice.logicalDevice.destroyPipelineLayout(pipelineLayout);
	mainDevice.logicalDevice.destroyRenderPass(renderPass);
	...
}
```

Now we will implement the `createRenderPass` function:

`VulkanRenderer.cpp`

```
void VulkanRenderer::createRenderPass()
{
	vk::RenderPassCreateInfo renderPassCreateInfo{};

	// Attachement description : describe color buffer output, depth buffer output...
	// e.g. (location = 0) in the fragment shader is the first attachment
	vk::AttachmentDescription colorAttachment{};
	// Format to use for attachment
	colorAttachment.format = swapchainImageFormat;
	// Number of samples to write for multisampling
	colorAttachment.samples = vk::SampleCountFlagBits::e1;
	// What to do with attachement before renderer. Here, clear when we start the render pass.
	colorAttachment.loadOp = vk::AttachmentLoadOp::eClear;
	// What to do with attachement after renderer. Here, store the render pass.
	colorAttachment.storeOp = vk::AttachmentStoreOp::eStore;
	// What to do with stencil before renderer. Here, don't care, we don't use stencil.
	colorAttachment.stencilLoadOp = vk::AttachmentLoadOp::eDontCare;
	// What to do with stencil after renderer. Here, don't care, we don't use stencil.
	colorAttachment.stencilStoreOp = vk::AttachmentStoreOp::eDontCare;

	// Framebuffer images will be stored as an image, but image can have different layouts
	// to give optimal use for certain operations
	// Image data layout before render pass starts
	colorAttachment.initialLayout = vk::ImageLayout::eUndefined;
	// Image data layout after render pass
	colorAttachment.finalLayout = vk::ImageLayout::ePresentSrcKHR;

	renderPassCreateInfo.attachmentCount = 1;
	renderPassCreateInfo.pAttachments = &colorAttachment;

	// Attachment reference uses an attachment index that refers to index
	// in the attachement list passed to renderPassCreateInfo
	vk::AttachmentReference colorAttachmentReference{};
	colorAttachmentReference.attachment = 0;
	// Layout of the subpass (between initial and final layout)
	colorAttachmentReference.layout = vk::ImageLayout::eColorAttachmentOptimal;

	// Subpass description, will reference attachements
	vk::SubpassDescription subpass{};
```

```
        // Pipeline type the subpass will be bound to.
        // Could be compute pipeline, or nvidia raytracing...
        subpass.pipelineBindPoint = vk::PipelineBindPoint::eGraphics;
        subpass.colorAttachmentCount = 1;
        subpass.pColorAttachments = &colorAttachmentReference;

        renderPassCreateInfo.subpassCount = 1;
        renderPassCreateInfo.pSubpasses = &subpass;

        // Subpass dependencies: transitions between subpasses + from the last subpass to what
        // happens after. Need to determine when layout transitions occur using subpass
        // dependencies. Will define implicitly layout transitions.
        array<vk::SubpassDependency, 2> subpassDependencies;
        // -- From layout undefined to color attachment optimal
        // ---- Transition must happens after
        // External: from outside the subpasses
        subpassDependencies[0].srcSubpass = VK_SUBPASS_EXTERNAL;
        // Which stage of the pipeline has to happen before
        subpassDependencies[0].srcStageMask = vk::PipelineStageFlagBits::eBottomOfPipe;
        subpassDependencies[0].srcAccessMask = vk::AccessFlagBits::eMemoryRead;
        // ---- But must happens before
        // Conversion should happen before the first subpass starts
        subpassDependencies[0].dstSubpass = 0;
        subpassDependencies[0].dstStageMask = vk::PipelineStageFlagBits::eColorAttachmentOutput;
        // ...and before the color attachment attempts to read or write
        subpassDependencies[0].dstAccessMask =
                vk::AccessFlagBits::eMemoryRead | vk::AccessFlagBits::eMemoryWrite;
        subpassDependencies[0].dependencyFlags = {};        // No dependency flag

        // -- From layout color attachment optimal to image layout present
        // ---- Transition must happens after
        subpassDependencies[1].srcSubpass = 0;
        subpassDependencies[1].srcStageMask = vk::PipelineStageFlagBits::eColorAttachmentOutput;
        subpassDependencies[1].srcAccessMask =
                vk::AccessFlagBits::eMemoryRead | vk::AccessFlagBits::eMemoryWrite;
        // ---- But must happens before
        subpassDependencies[1].dstSubpass = VK_SUBPASS_EXTERNAL;
        subpassDependencies[1].dstStageMask = vk::PipelineStageFlagBits::eBottomOfPipe;
        subpassDependencies[1].dstAccessMask = vk::AccessFlagBits::eMemoryRead;
```

```
        subpassDependencies[1].dependencyFlags = vk::DependencyFlags();

        renderPassCreateInfo.dependencyCount = static_cast<uint32_t>(subpassDependencies.size());
        renderPassCreateInfo.pDependencies = subpassDependencies.data();

        renderPass = mainDevice.logicalDevice.createRenderPass(renderPassCreateInfo);
}
```

Creating the graphics pipeline

We will finally create and destroy a graphics pipeline:

`VulkanRenderer.h`

```
        ...
        vk::Pipeline graphicsPipeline;
        ...
```

`VulkanRenderer.cpp`

```
        ...
void VulkanRenderer::clean()
{
        mainDevice.logicalDevice.destroyPipeline(graphicsPipeline);
        ...
```

And set the creation instruction into our `createPipeline` function:

`VulkanRenderer.cpp`

```
void VulkanRenderer::createGraphicsPipeline()
{
	...
	// -- PASSES --
	// Passes are composed of a sequence of subpasses that can pass data from one to another

	// -- GRAPHICS PIPELINE CREATION --
	vk::GraphicsPipelineCreateInfo graphicsPipelineCreateInfo{};
	graphicsPipelineCreateInfo.stageCount = 2;
	graphicsPipelineCreateInfo.pStages = shaderStages;
	graphicsPipelineCreateInfo.pVertexInputState = &vertexInputCreateInfo;
	graphicsPipelineCreateInfo.pInputAssemblyState = &inputAssemblyCreateInfo;
	graphicsPipelineCreateInfo.pViewportState = &viewportStateCreateInfo;
	graphicsPipelineCreateInfo.pDynamicState = nullptr;
	graphicsPipelineCreateInfo.pRasterizationState = &rasterizerCreateInfo;
	graphicsPipelineCreateInfo.pMultisampleState = &multisamplingCreateInfo;
	graphicsPipelineCreateInfo.pColorBlendState = &colorBlendingCreateInfo;
	graphicsPipelineCreateInfo.pDepthStencilState = nullptr;
	graphicsPipelineCreateInfo.layout = pipelineLayout;
	// Renderpass description the pipeline is compatible with.
	// This pipeline will be used by the render pass.
	graphicsPipelineCreateInfo.renderPass = renderPass;
	// Subpass of render pass to use with pipeline. Usually one pipeline by subpass.
	graphicsPipelineCreateInfo.subpass = 0;
	// When you want to derivate a pipeline from an other pipeline OR
	graphicsPipelineCreateInfo.basePipelineHandle = VK_NULL_HANDLE;
	// Index of pipeline being created to derive from (in case of creating multiple at once)
	graphicsPipelineCreateInfo.basePipelineIndex = -1;

	// The handle is a cache when you want to save your pipeline to create an other later
	auto result = mainDevice.logicalDevice.createGraphicsPipeline(
		VK_NULL_HANDLE, graphicsPipelineCreateInfo);
	// We could have used createGraphicsPipelines to create multiple pipelines at once.
	if (result.result != vk::Result::eSuccess)
	{
		throw std::runtime_error("Cound not create a graphics pipeline");
	}
	graphicsPipeline = result.value;
```

```
        // Destroy shader modules
        mainDevice.logicalDevice.destroyShaderModule(fragmentShaderModule);
        mainDevice.logicalDevice.destroyShaderModule(vertexShaderModule);
}
```

Framebuffer and command buffers

Concepts

Framebuffer

The **Framebuffer** is a connection between an image (or images) and the render pass. We attach image to a framebuffer. The framebuffer is like a frame you put images on.

The render pass will outputs fragment data from a pipeline's execution to the images bound to the framebuffer's attachment. Be careful : the framebuffer images line up 1-to-1 with the attachments in the render pass : e.g. if the render pass output a color attachment at the top, the framebuffer will have a color attachment at the top, then if we have a depth attachment underneath, then the framebuffer will have to match and have a depth attachment.

Command buffer

Unlike OpenGL where we submit one command at a time, Vulkan works with pre-recording a group of command, then submitting them to a queue at once. This group is **the command buffer**.

Usually commands will be in the form of:

1. Start a render pass
2. Bind a pipeline
3. Bind vertex/index data
4. Bind descriptor sets and push constants
5. Draw

You can also submit a command to begin a new subpass, but you will have to bind a pipeline for this subpass.

We can't create buffers out of thin air. We have to allocate them from a **pool**. This pool is handled by Vulkan.

When the command buffer is set and ready, we have to submit it to the appropriate **queue**. In our case, we have a graphics queue, so our command buffer must be performing graphical operation, such as render passes. Queue will execute automatically.

For synchronisation (in order to avoid twice access on the same data for instance), we will use **Semaphores** and **Fences**.

A semaphore is a flag that say, when it is *signaled* (true), a resource is available to use. If it is *unsignaled*, the program must wait. Semaphores are used by the GPU itself : they only create synchronisation between GPU functions. We can set them up CPU side, but cannot control them. We will use semaphones :

- When an image has finished being rendered and is ready to present
- When an image has become available to draw onto (after presentation)

Fences are similar to semaphores, except we have the ability to unsignal a fence and wait on a fence CPU-side. This allow to have CPU-GPU synchronization. We will use those functions:

- `vkResetFences` : this will unsignal a fence until the GPU signales it again
- `vkWaitForFences` : block the CPU until the GPU signals the fence

We will use fences to ensure a "frame" is available, so we don't accidentally flood the queue with too many draw/present commands.

Framebuffer

Framebuffers are quite straightforward to create. We need a vector to store them and a create function:

`VulkanRenderer.h`

```
...
vector<vk::Framebuffer> swapchainFramebuffers;
void createFramebuffers();
...
```

We loop through swapchain images to create an associated framebuffer. This will ensure the 1-to-1 relationship with the attachments of the render pass.

`VulkanRenderer.cpp`

```
void VulkanRenderer::createFramebuffers()
{
	// Create one framebuffer for each swapchain image
	swapchainFramebuffers.resize(swapchainImages.size());
	for (size_t i = 0; i < swapchainFramebuffers.size(); ++i)
	{
		// Setup attachments
		array<vk::ImageView, 1> attachments{ swapchainImages[i].imageView };

		// Create info
		vk::FramebufferCreateInfo framebufferCreateInfo{};
		// Render pass layout the framebuffer will be used with
		framebufferCreateInfo.renderPass = renderPass;
		framebufferCreateInfo.attachmentCount = static_cast<uint32_t>(attachments.size());
		// List of attachments (1:1 with render pass, thanks to variable i)
		framebufferCreateInfo.pAttachments = attachments.data();
		framebufferCreateInfo.width = swapchainExtent.width;
		framebufferCreateInfo.height = swapchainExtent.height;
		// Framebuffer layers
		framebufferCreateInfo.layers = 1;

		swapchainFramebuffers[i] =
			mainDevice.logicalDevice.createFramebuffer(framebufferCreateInfo);
	}
}
```

Do not forget to use the create function and clean the framebuffers.

`VulkanRenderer.cpp`

```
...
int VulkanRenderer::init(GLFWwindow* windowP)
{
	...
	try
	{
		...
		createFramebuffers();
	}
	catch (const std::runtime_error& e)
	...
}

void VulkanRenderer::clean()
{
	for (auto framebuffer : swapchainFramebuffers)
	{
		mainDevice.logicalDevice.destroyFramebuffer(framebuffer);
	}
	...
```

Graphics command buffer

Command pool

`VulkanRenderer.h`

```
	...
	vk::CommandPool graphicsCommandPool;
	void createGraphicsCommandPool();
	...
```

`VulkanRenderer.cpp`

```
void VulkanRenderer::createGraphicsCommandPool()
{
	QueueFamilyIndices queueFamilyIndices = getQueueFamilies(mainDevice.physicalDevice);

	VkCommandPoolCreateInfo poolInfo{};
	poolInfo.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
	// Queue family type that buffers from this command pool will use
	poolInfo.queueFamilyIndex = queueFamilyIndices.graphicsFamily;

	graphicsCommandPool = mainDevice.logicalDevice.createCommandPool(poolInfo);
;}
```

Create an clean code:

`VulkanRenderer.cpp`

```
...
int VulkanRenderer::init(GLFWwindow* windowP)
{
	...
	try
	{
		...
		createGraphicsCommandPool();
	}
	catch (const std::runtime_error& e)
	...
}

void VulkanRenderer::clean()
{
	mainDevice.logicalDevice.destroyCommandPool(graphicsCommandPool);
	...
```

Command buffer

Because we have a pool, we won't need to create the command buffers. We just have to allocate them into the pool. As usual we will need a function and a vector to store the allocated

command buffers:

`VulkanRenderer.h`

```
        ...
        vector<vk::CommandBuffer> commandBuffers;
        void createGraphicsCommandBuffers();
        ...
```

`VulkanRenderer.cpp`

```
void VulkanRenderer::createGraphicsCommandBuffers()
{
        // Create one command buffer for each framebuffer
        commandBuffers.resize(swapchainFramebuffers.size());

        vk::CommandBufferAllocateInfo commandBufferAllocInfo{};
        // We are using a pool
        commandBufferAllocInfo.commandPool = graphicsCommandPool;
        commandBufferAllocInfo.commandBufferCount = static_cast<uint32_t>(commandBuffers.size());
        // Primary means the command buffer will submit directly to a queue.
        // Secondary cannot be called by a queue, but by an other primary command
        // buffer, via vkCmdExecuteCommands.
        commandBufferAllocInfo.level = vk::CommandBufferLevel::ePrimary;

        commandBuffers = mainDevice.logicalDevice.allocateCommandBuffers(commandBufferAllocInfo);
}
```

Recording commands

Commands in Vulkan, like drawing operations and memory transfers, are not executed directly using function calls. You have to record all of the operations you want to perform in command buffer objects. The advantage of this is that all of the hard work of setting up the drawing commands can be done in advance and in multiple threads. After that, you just have to tell Vulkan to execute the commands in the main loop.

We will create a `recordCommands` function:

`VulkanRenderer.cpp`

```
int VulkanRenderer::init(GLFWwindow* windowP)
{
	window = windowP;
	try
	{
		...
		recordCommands();
	}
	catch (const std::runtime_error& e)
	...
}
...
void VulkanRenderer::recordCommands()
{
	// How to begin each command buffer
	vk::CommandBufferBeginInfo commandBufferBeginInfo{};
	// Buffer can be resubmited when it has already been submited
	commandBufferBeginInfo.flags = vk::CommandBufferUsageFlagBits::eSimultaneousUse;

	// Information about how to being a render pass (only for graphical apps)
	vk::RenderPassBeginInfo renderPassBeginInfo{};
	// Render pass to begin
	renderPassBeginInfo.renderPass = renderPass;
	// Start point of render pass in pixel
	renderPassBeginInfo.renderArea.offset = vk::Offset2D { 0, 0 };
	// Size of region to run render pass on
	renderPassBeginInfo.renderArea.extent = swapchainExtent;

	vk::ClearValue clearValues{};
	std::array<float, 4> colors { 0.6f, 0.65f, 0.4f, 1.0f };
	clearValues.color = vk::ClearColorValue{ colors };
	renderPassBeginInfo.pClearValues = &clearValues;
	renderPassBeginInfo.clearValueCount = 1;

	for (size_t i = 0; i < commandBuffers.size(); ++i)
	{
		// Because 1-to-1 relationship
		renderPassBeginInfo.framebuffer = swapchainFramebuffers[i];
```

```
                // Start recording commands to command buffer
                commandBuffers[i].begin(commandBufferBeginInfo);

                // Begin render pass
                // All draw commands inline (no secondary command buffers)
                commandBuffers[i].beginRenderPass(
                        renderPassBeginInfo, vk::SubpassContents::eInline);

                // Bind pipeline to be used in render pass,
                // you could switch pipelines for different subpasses
                commandBuffers[i].bindPipeline(
                        vk::PipelineBindPoint::eGraphics, graphicsPipeline);

                // Execute pipeline
                // Draw 3 vertices, 1 instance, with no offset. Instance allow you
                // to draw several instances with one draw call.
                commandBuffers[i].draw(3, 1, 0, 0);

                // End render pass
                commandBuffers[i].endRenderPass();

                // Stop recordind to command buffer
                commandBuffers[i].end();
        }
}
```

The Cmd part of the functions' names mean this operation can be recorded.

Drawing and synchronisation

Create a public `draw` function in the VulkanRenderer and call it in the main. Here is what we are going to do:

`VulkanRenderer.cpp`

```
void VulkanRenderer::draw()
{
	// 1. Get next available image to draw and set a semaphore to signal
	// when we're finished with the image.

	// 2. Submit command buffer to queue for execution, make sure it waits
	// for the image to be signaled as available before drawing, and
	// signals when it has finished rendering.

	// 3. Present image to screen when it has signalled finished rendering

}
```

Synchronisation

We will create the two semaphore and set the synchronisation in a dedicated function:

`VulkanRenderer.h`

```
	...
	vk::Semaphore imageAvailable;
	vk::Semaphore renderFinished;

	void createSynchronisation();
	...
```

`VulkanRenderer.cpp`

```
int VulkanRenderer::init(GLFWwindow* windowP)
{
	window = windowP;
	try
	{
		...
		createSynchronisation();
	}
	catch (const std::runtime_error& e)
	...
}

void VulkanRenderer::clean()
{
		mainDevice.logicalDevice.destroySemaphore(renderFinished);
		mainDevice.logicalDevice.destroySemaphore(imageAvailable);
	...
}
...
void VulkanRenderer::createSynchronisation()
{
	// Semaphore creation info
	vk::SemaphoreCreateInfo semaphoreCreateInfo{}; // That's all !

	imageAvailable = mainDevice.logicalDevice.createSemaphore(semaphoreCreateInfo);
	renderFinished = mainDevice.logicalDevice.createSemaphore(semaphoreCreateInfo);
}
```

Our first draw

We can implement the draw function after a thousand line of codes.

`VulkanRenderer.cpp`

```
void VulkanRenderer::draw()
{
	// 1. Get next available image to draw and set a semaphore to signal
	// when we're finished with the image.
	uint32_t imageToBeDrawnIndex = (mainDevice.logicalDevice.acquireNextImageKHR(swapchain,
		std::numeric_limits<uint32_t>::max(), imageAvailable, VK_NULL_HANDLE)).value;

	// 2. Submit command buffer to queue for execution, make sure it waits
	// for the image to be signaled as available before drawing, and
	// signals when it has finished rendering.
	vk::SubmitInfo submitInfo{};
	submitInfo.waitSemaphoreCount = 1;
	submitInfo.pWaitSemaphores = &imageAvailable;
	// Keep doing command buffer until imageAvailable is true
	vk::PipelineStageFlags waitStages[]{ vk::PipelineStageFlagBits::eColorAttachmentOutput };
	// Stages to check semaphores at
	submitInfo.pWaitDstStageMask = waitStages;
	submitInfo.commandBufferCount = 1;
	// Command buffer to submit
	submitInfo.pCommandBuffers = &commandBuffers[imageToBeDrawnIndex];
	// Semaphores to signal when command buffer finishes
	submitInfo.signalSemaphoreCount = 1;
	submitInfo.pSignalSemaphores = &renderFinished[currentFrame];

	graphicsQueue.submit(submitInfo, VK_NULL_HANDLE);

	// 3. Present image to screen when it has signalled finished rendering
	vk::PresentInfoKHR presentInfo{};
	presentInfo.waitSemaphoreCount = 1;
	presentInfo.pWaitSemaphores = &renderFinished;
	presentInfo.swapchainCount = 1;
	// Swapchains to present to
	presentInfo.pSwapchains = &swapchain;
	// Index of images in swapchains to present
	presentInfo.pImageIndices = &imageToBeDrawnIndex;

	presentationQueue.presentKHR(presentInfo);
}
```

Do not worry though: this preparation was long because we needed to setup everything. Further code will be shorter.

Solving errors

Solving validation errors

We have a lot of validation errors when we close our application. This comes from the fact the device is is doing operations when we want to quit. We will add a new command to the clean up code so that we'll wait for the device to be idle.

`VulkanRenderer.cpp`

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

Solving memory leak

If you watch the memory consuption of your vulkan app in the program manager, you'll see that it eats 100kB of memory every few seconds. There are two reasons:

1. We use the same semaphores for everything. Images are computed in parallel, and maybe if image 2 is ready before, it will hit the imageAvailable semaphore, that will allow to take image 1 (which is still being calculated). We need to ensure we have a semaphore for each indivisual image (command buffer actually) so we avoid this mix up.
2. We never set a limit to the amount of thing we submit to the queue. The time one object is drawing, maybe five objects can be added to the queue. Queue will grow endlessly.

First, change semaphores to vector of semaphors and add a constant for the maximum number of frame draws:

`VulkanRenderer.h`

```
void VulkanRenderer::clean()
{
	...
	vector<vk::Semaphore> imageAvailable;
	vector<vk::Semaphore> renderFinished;
	const int MAX_FRAME_DRAWS = 2;
	// Should be less than the number of swapchain images, which is 3 (could cause bugs)
	...
}
```

Update `createSynchronisation` and the `clean` function:

`VulkanRenderer.cpp`

```
void VulkanRenderer::clean()
{
	mainDevice.logicalDevice.waitIdle();

	for (size_t i = 0; i < MAX_FRAME_DRAWS; ++i)
	{
		mainDevice.logicalDevice.destroySemaphore(renderFinished[i]);
		mainDevice.logicalDevice.destroySemaphore(imageAvailable[i]);
	}
	...
}

...
void VulkanRenderer::createSynchronisation()
{
	imageAvailable.resize(MAX_FRAME_DRAWS);
	renderFinished.resize(MAX_FRAME_DRAWS);

	vk::SemaphoreCreateInfo semaphoreCreateInfo{}; // That's all !

	for (size_t i = 0; i < MAX_FRAME_DRAWS; ++i)
	{
		imageAvailable[i] = mainDevice.logicalDevice.createSemaphore(semaphoreCreateInfo);
		renderFinished[i] = mainDevice.logicalDevice.createSemaphore(semaphoreCreateInfo);
	}
}
```

Now we need the track the current frame we are drawing. Create a `currentFrame` variable:

`VulkanRenderer.h`

```
int currentFrame = 0;
```

We will use this variable in the draw function, to call the right semaphore and to update it.

`VulkanRenderer.cpp`

```
void VulkanRenderer::draw()
{
	// 1. Get next available image to draw and set a semaphore to signal
	// when we're finished with the image.
	uint32_t imageToBeDrawnIndex = (mainDevice.logicalDevice.acquireNextImageKHR(swapchain,
		std::numeric_limits<uint32_t>::max(), imageAvailable[currentFrame], VK_NULL_HANDLE

	// 2. Submit command buffer to queue for execution, make sure it waits
	// for the image to be signaled as available before drawing, and
	// signals when it has finished rendering.
	...
	submitInfo.pWaitSemaphores = &imageAvailable[currentFrame];
	...
	// Semaphores to signal when command buffer finishes
	submitInfo.pSignalSemaphores = &renderFinished[currentFrame];
	...

	// 3. Present image to screen when it has signalled finished rendering
	...
	presentInfo.pWaitSemaphores = &renderFinished[currentFrame];
	...

	currentFrame = (currentFrame + 1) % MAX_FRAME_DRAWS;
}
```

This solves the first problem, but we are still filling the queues and memory is still leaking. We will use a fence so that the CPU do not continuously submit command buffers to the graphics queue.

`VulkanRenderer.h`

```
vector<VkFence> drawFences;
```

Update `clean` and `createSynchronisation`:

`VulkanRenderer.cpp`

```
void VulkanRenderer::clean()
{
	mainDevice.logicalDevice.waitIdle();

	for (size_t i = 0; i < MAX_FRAME_DRAWS; ++i)
	{
		mainDevice.logicalDevice.destroySemaphore(renderFinished[i]);
		mainDevice.logicalDevice.destroySemaphore(imageAvailable[i]);
		mainDevice.logicalDevice.destroyFence(drawFences[i]);
	}
	...
}
...
void VulkanRenderer::createSynchronisation()
{
	imageAvailable.resize(MAX_FRAME_DRAWS);
	renderFinished.resize(MAX_FRAME_DRAWS);
	drawFences.resize(MAX_FRAME_DRAWS);

	// Semaphore creation info
	vk::SemaphoreCreateInfo semaphoreCreateInfo{}; // That's all !

	// Fence creation info
	vk::FenceCreateInfo fenceCreateInfo{};
	// Fence starts open
	fenceCreateInfo.flags = vk::FenceCreateFlagBits::eSignaled;

	for (size_t i = 0; i < MAX_FRAME_DRAWS; ++i)
	{
		imageAvailable[i] = mainDevice.logicalDevice.createSemaphore(semaphoreCreateInfo);
		renderFinished[i] = mainDevice.logicalDevice.createSemaphore(semaphoreCreateInfo);
		drawFences[i] = mainDevice.logicalDevice.createFence(fenceCreateInfo);
	}
}
```

Now use the fence in the draw function to signal when the image has finished drawing.

`VulkanRenderer.cpp`

```
void VulkanRenderer::draw()
{
	// 0. Freeze code until the drawFences[currentFrame] is open
	mainDevice.logicalDevice.waitForFences(drawFences[currentFrame], VK_TRUE,
		std::numeric_limits<uint32_t>::max());
	// When passing the fence, we close it behind us
	mainDevice.logicalDevice.resetFences(drawFences[currentFrame]);


	// 1. Get next available image to draw and set a semaphore to signal
	// when we're finished with the image.
	uint32_t imageToBeDrawnIndex = (mainDevice.logicalDevice.acquireNextImageKHR(
		swapchain,
		std::numeric_limits<uint32_t>::max(),
		imageAvailable[currentFrame],
		VK_NULL_HANDLE)
	).value;


	// 2. Submit command buffer to queue for execution, make sure it waits
	// for the image to be signaled as available before drawing, and
	// signals when it has finished rendering.
	...
	graphicsQueue.submit(submitInfo, drawFences[currentFrame]);
	...
}
```

We now have gotten rid of our memory leak. We can still update the code.

In `recordCommands`, because we now fence the images to be drawn, we no longer need the commandBufferBeginInfo flags. Comment it out:

`VulkanRenderer.cpp`

```
void VulkanRenderer::recordCommands()
{
	// How to begin each command buffer
	vk::CommandBufferBeginInfo commandBufferBeginInfo{};
	// Buffer can be resubmited when it has already been submited
	//commandBufferBeginInfo.flags = vk::CommandBufferUsageFlagBits::eSimultaneousUse;
	...
```

Now, our first triangle is finally ready. Congratulations 😄