Rendering a triangle using Vulkan

Melvin Rother, melvin.rother@pm.me

Step one: Creating a Vulkan instance

- Creating a Vulkan Instance is the first step into using the Vulkan Rendering API
- Initializing a VkInstance object initializes the Vulkan library
- To create a Vulkan instance, a few steps are required:
  1. Create a VkApplicationInfo struct
  2. Initialize its members
     - sType refers to the type of a Vulkan struct -> every object in Vulkan requires this member to be filled
  3. The create a VkInstanceCreateInfo struct and initialize its members
     - Pass a reference to the VkApplicationStruct to the Create Infos pApplicationInfo member
  4. Call vkCreateInstance and pass it the VkInstanceCreateInfo, a pointer to your custom memory allocator if you use one and a pointer to the object supposed to store the instance handle to create the instance
- 1.1: Optional, but not really: Enabling the Vulkan validation layers

Step two: Selecting Physical devices

- To render something, we need a graphics card
  + Thus, we need to query which graphics cards are available and choose the one(s) which support the features we require -> in Vulkan, this can be achieved by using physical devices
- Listing the Graphics Cards is similar to listing the extensions
  1. Query the amount of graphics cards available
  2. Store the handles to all devices found in an array
  3. Loop over all of the devices found and check whether they support the operations you want to perform (queue families needed, does the device support presenting images to window surfaces etc.)

Step three: Queue Families

- In Vulkan, every operation needs to be submitted to a queue
- There are several different queues that allow for different operations
- Different devices support different queues -> one needs to check which ~~queue types~~ (so-called queue families) are supported by any given device
- Querying the supported queue families is similar to listing the supported extensions and available physical devices:
  1. Query the number of queue families supported by the device
  2. Use the number of supported queue families to retrieve the queue families' properties and store them in an array
  3. Check whether the queue families required are available

Step four: Creating Logical Devices

- Now that we have the physical devices and the queue families they support, we can create a Logical Device
- Physical devices represent the actual hardware we are using – the GPUs installed in the end users PC
- Logical Devices allow us to interface with the physical devices – they allow us to talk to the GPU
- To create a logical device, one needs to add the information about the queue families and device features one plans to use in the respective VkDeviceQueueCreateInfo and VkPhysicalDeviceFeatures structs and pass them to the VkDeviceCreateInfo struct used to create the logical device

Step five: How to interface with the windowing system?

- Windowing systems are system-dependent -> that is an issue because Vulkan is a system-independent rendering API
- The solution for this is window surfaces – abstract surface types that images can be presented to
- Their usage is system-independent, but their creation is system dependent as they need handles to the windows and other important details to ultimately present the images to
- To create a Windows-specific window surface, it is first required to access the native win32 window handling system and create a hwnd and a hinstance. Then a VkWin32SurfaceCreateInfoKHR can be created, to which the hwnd, the hinstance and some other basic info are passed. Then the create info, the Vulkan instance and a reference to the object supposed to store the surface can be passed to the vkCreateWin32SurfaceKHR() function.
- Or you can use GLFW and call the glfwCreateWindowSurface() -> that will instantly create a surface for any of the systems supported by GLFW
- Now we can also use the information about the queue families to create the presentation queue and the graphics queue

Step six: The Swap chain

- To render anything, the data we want to render to the screen needs to be stored somewhere -> In Vulkan (as in DX12), the swap chain is an infrastructure to do just that
- The swap chain is a queue of images that will be presented to the screen and is also the place for many important configurations like the surface format or the presentation format
- To create a swap chain, it is necessary first to query whether the physical device supports directly presenting images to a screen
- To use the swap chain, the correct extension needs to be enabled (VK_KHR_swapchain)
- The swap chain also needs to be compatible with the Window surface in use (it might not support the format of the surface, or the min/max width or height of an image may be incompatible between the surface and swap chain) -> the user needs to query whether the requirements are met
- Make sure to query whether the optimal settings for the Swap Chain you need (like source format etc.) are supported and if not write logic to choose another suitable option
  - One of the most important settings is the presentation mode which represents the conditions for showing the images to the screen.
  - There are four modes in Vulkan, it is important to make sure to choose the right one for the task

Step six: The Swap chain

- To create a Swap Chain, a few steps are necessary
  1. Check whether swap chains are even supported (some graphics cards like the ones in servers do not support presenting images directly to the screen)
  2. Enable the VK_KHR_swapchain extension
  3. Query swap chain properties and check whether the Window Surface supports these properties (it might not support the min/max number of images the swap chain can hold, or it is not able to use the same colour space)
     - Make sure to check for surface formats, presentation modes and Surface Capabilities
  4. Create the VkSwapChainPresentInfoKHR and fill it with the details queried and other necessary configurations
     - Also make sure to decide how the Swap Chain is supposed to handle images shared between queues in case the graphics queue and present queue are not in the same queue family
     - If several queue families are supposed to own an image, use VK_SHARING_MODE_CONCURRENT. While using VK_SHARING_MODE_EXCLUSIVE is more performant, it requires switching ownership between queue families, which involves concepts a bit too big to tackle in this presentation
  5. Create the Swap Chain by calling vkCreateSwapChainKHR
     - Do not forget to destroy the Swap Chain when terminating the program!

Step six: The Swap chain

- One last step before we continue: We need to retrieve the handles to the images the swap chain contains
- You do this like you would when querying anything else in Vulkan:
  1. Query the number of images in the Swap Chain
  2. Use the number of images to retrieve their handles & store them

Step seven: Image Views

- To use any image, we first need to describe which part of and how to access the image
- To do so, Vulkan uses Image Views
- Thus, if we want to use the images in the swap chain as colour targets, we need to create individual Image Views for each image in the Swap Chain
- The steps to doing so are simple
  1. Iterate over each Image in the Swap Chain
  2. Create a VkImageViewCreateInfo in the loop and set the settings contained in it to whatever you need
  3. Call VkCreateImageView for each of them
  4. Do not forget to delete them when terminating the application
- Settings of Note:
  - viewType allows you to set how the image will be interpreted (1D, 2D, 3D Texture, Cubemap etc.)
  - subResourceRange describes the images purpose and which parts of it should be accessed (it allows, for example, to specify the mip levels)

Step eight: The Graphics Pipeline

- The graphics pipeline is one of the most important things when drawing anything
- It is a sequence of operations that receives the vertices and textures of any mesh we want to draw and outputs the pixels in the render targets
- In Vulkan, almost every stage of the render pipeline is mutable -> thus, we can disable any operation that is not needed
- This also means that we need to define every single combination of the pipeline we need beforehand
- To create a Graphics Pipeline, it is necessary to once again make a create info called VkGraphicsPipelineCreateInfo
- However, to actually fill this Pipeline with all the data it requires, a few more steps are required:

Step eight.1: Shaders & Shader Modules

- Shaders are small programs that run on the GPU and define how we interpret the input data and arrive at the pixels that are presented to the screen
- In Vulkan, shaders have to be defined in system-independent bytecode - which is also completely unreadable for humans
- Luckily the vulkan sdk provides ways to compile human-readable GLSL code to Vulkan bytecode
- After doing so, it is necessary to use the loaded shader bytecode to create Shader Modules that can be handed to the Graphics Pipeline
- Doing so is simple:
  1. Create a VkShaderModuleInfo and fill its members with both the raw bytecode data and the code size
  2. Call VkCreateShaderModule
- The bytecode will then be compiled and linked when the Graphics Pipeline is created
- It is also necessary to assign the shaders to a specific pipeline stage by creating a VkPipelineShaderStageCreateInfo as a part of the actual process of creating a Graphics Pipeline

Step eight.2: Fixed Functions

- Most Graphics APIs provide default states for most of the graphics pipeline
- In Vulkan, you need to be more explicit: you need to define these pipeline states clearly -> the advantage is that the user is aware of and can actively change what is happening in almost every state of the pipeline
- Most pipeline states are baked into immutable pipeline state objects and thus are not changeable except if the entire graphics pipeline is recreated at draw time
- A limited amount of a state can, however, be changed without recreating the entire pipeline -> so-called Dynamic States (e.g. size of the viewport, blend constants etc.)
- The Dynamic state is just one of the states that need to be defined. To create a graphics pipeline, the following need to be defined:
  + Dynamic State
  + Vertex Input State -> defining the format of the vertex data passed into the shader
  + Input Assembly State -> describes what kind of geometry will be drawn from the vertices & if primitive restart should be enabled
  + Viewport State -> describes the region of the framebuffer the output will be rendered to
  + Rasterizer State -> describes how the geometry shaped from the vertices is turned into fragments
  + Multisampling State -> configures multisampling (one of the ways to do Anti-Aliasing)
  + Depth and Stencil testing
  + Color blending
- The Pipeline Layout also needs to be described, that is which uniform values should be available in shaders

Step eight.3: Render Passes

- Render Passes tell Vulkan about the framebuffer attachments, colour and depth buffers, how to handle their contents and how many samples to use for each of them and more
- To finish the Render Pipeline and draw anything, Vulkan needs this information. Thus it is necessary to create a RenderPass providing all of that information
- To do so, we first need to create a colour buffer (or how many buffers we need) by creating a VkAttachmentDescription
- Then, we can describe the sub-passes. Sub-passes are subsequent rendering operations that depend on the contents of the previous render passes. A render pass can have several sub-passes.
- Every sub-pass references one or more of the attachments described using VKAttachmentDescription. To reference an attachment, create a VkAttachmentReference and feed it to the VkSubpassDescription you need to make for the sub-pass.
- When adding an attachment to a sub-pass, it will be stored in an array. The index of the attachment can then be directly referenced in the shaders using the layout(location = index) directive
- Creating the Render Pass is then trivial. Simply create a VkRenderPassCreateInfo struct, fill its members with the required information (including the attachments & sub-passes just created) and call vkCreateRenderPass

Step eight: The Graphics Pipeline

- With the Shader modules, the fixed functions and the render passes created, it is possible to create the actual graphics pipeline
- As with most objects in Vulkan, this is done by creating a struct describing the object; in this case, a VkGraphicsPipelineCreateInfo
- After assigning its members to Pipeline Layout, Fixed Function stages, Render Passes etc, call vkCreateGraphicsPipelines to generate the actual Pipeline Object
- The vkCreateGraphicsPipelines function has more parameters than the usual vkCreate functions, as it allows to create several pipelines at once
- The only thing left to do now is to destroy the pipeline when terminating the program using vkDestroyPipeline

Step nine: The Framebuffers

- Framebuffers represent a collection of the attachments used by a render pass instance
- Since the image used for attachments depends on the image the swap chain returns, it is required to create a frame buffer for each of these images
- Creating these framebuffers is relatively straightforward:
  1. Create a VkImageView for each of the Swap Chain Image Views
  2. Create a VkFrameBufferCreateInfo for each buffer you want to create and assign its members -> assign the pAttachments member to the corresponding VkImageView
  3. Call vkCreateFrameBuffer
- I suggest implementing these steps in a loop for simplicity

Step ten: The Command Buffer

- In Vulkan, commands like drawing operations are not directly executed
- Instead, all commands are recorded in command buffer objects
- This allows Vulkan to more efficiently process the commands as they are all submitted at the same time
- Command buffers are executed by submitting them on one of the device queues
- To create a Command Buffer, we first need to understand how to create a Command Pool

Step ten.1: Command Pools

- Command pools manage the memory used to store the buffers -> command buffers are allocated from them
- Command Pool creation is the same as with every Vulkan Object
- One of the most important members to set in the VkCommandPoolCreateInfo is the flags. There are two possible ones: VK_COMMAND_POOL_CREATE_TRANSIENT_BIT and VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT
- In our case, we want to be able to record a command buffer every frame; thus we need to be able to reset and rerecord it -> that is the behaviour VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT defines
- Every command pool can only create command buffers submitted to a single type of command queue -> as we want to draw with the command buffer in question, we have to hand the VkCommandPoolCreateInfo the indices to the graphics family we retrieved earlier
- The only thing left to do then is to call vkCreateCommandPool

Step ten: The Command Buffer

- After creating the Command Pool, we can allocate the command buffer
- Doing so is the same process as we are already used to, but instead of being called VkCommandBufferCreateInfo, the struct we use is called VkCommandBufferAllocateInfo
- The most interesting member of that struct is the command buffer level. It can either be set to VK_COMMAND_BUFFER_LEVEL_PRIMARY or VK_COMMAND_BUFFER_LEVEL_SECONDARY
- Primary command buffers can be submitted to a queue for execution, while secondary command buffers cannot. However, secondary command buffers can be called from primary command buffers, while primary command buffers can not be called from other command buffers -> in our case, we need a primary command buffer
- After creating the VkCommandBufferAllocateInfo, we call vkAllocateCommandBuffer to actually allocate the Buffer

Step ten.2: Recording the Command Buffer

- After command buffer allocation, we can then record the command buffer (e.g. writing the commands we want to execute into the command buffer)
- This is again done through a struct: VkCommandBufferBeginInfo -> this denotes that we want to begin recording commands
- The flags parameter is again interesting as they allow for specifying how we will use the command buffer; however, for now, none of these flags is applicable
- Then, we can start a RenderPass by creating a VkRenderPassBeginInfo struct and submitting it to the vkCmdBeginRenderPass function
- We want to record three commands:
  1. Set the Viewport using vkCmdSetViewport
  2. Set the Scissor Rect using vkCmdSetScissor
  3. Draw using vkCmdDraw
- After recording them, we can end the Render Pass using vkCmdEndRenderPass and then end recording using vkEndCommandBuffer

Step eleven: Rendering & Presentation

- After creating & recording the Command Buffer, we can start working on rendering and then presenting our image
- At a high level, frames in Vulkan share common steps
  - Wait for the previous frame to finish
  - Retrieve image from the Swap Chain
  - Record Command Buffer that draws the scene onto that image
  - Submit the Command Buffer
  - Present the Swap Chain
- A lot of these steps are asynchronous -> which means that the functions will return before the action on the GPU has been completed
- That means that functions that depend on the other to be finished could start when the operation they depend on is, in fact, not yet finished -> bad
- Vulkan requires one to be very explicit with the synchronization of executions
- To order and synchronize the execution of commands in Vulkan, we have two prominent Options: Semaphores and Fences

Step eleven.1: Semaphores

- A semaphore is used to add order between queue operations (work submitted to a queue, either as Command buffers or from within a function)
- Semaphores can be used to order work inside the same queue and between different queues
- There are two kinds of Semaphores, binary and timeline (we will exclusively use binary Semaphores)
- Semaphores can be either signaled or unsignaled, but they begin their life unsignaled
- Because of that, we can use the Semaphore as a signal Semaphore in one operation and as a wait Semaphore in another. When the first operation is finished, the Semaphore will be signaled, and the waiting operation will know it can execute

Step eleven.2: Fences

- A fence is also used to synchronize the execution of operations, but it is used for ordering execution on the CPU (also known as the host)
- If the CPU needs to know when the GPU has finished something, using a Fence is usually the best option
- Fences are either signaled or unsignaled (as are Semaphores)
- Whenever work is submitted to be executed, a fence can be submitted alongside it, and when the work is finished the fence will be signaled
- The host can wait for the fence to be signaled, guaranteeing the work has been finished

Step eleven: Rendering & Presentation

- Knowing that we actively have to synchronize the execution of Vulkan commands and knowing we can use Semaphores and Fences to do so, we can start presenting our work to the screen
- To render our image is now almost trivial:
  1. First we have to wait for & reset our fences
  2. Then, we acquire an image from the Swap Chain using vkAcquireNextImageKHR
  3. Next, record & submit the Command Buffer
     - To submit the command buffer, create a VkSubmitInfo. Do not forget to add wait and signal semaphores to synchronise execution. Then call vkQueueSubmit
  4. After this, create a VkPresentInfoKHR object and pass it the semaphore you want to use to signal it, the Swap Chain and the image indices
  5. Finally, call vkQueuePresentKHR
  6. Watch the magic happen!

The End Result