Vulkan - Der Weg zum Dreieck

Peter Maximilian Kain

8. April 2020

Inhaltsverzeichnis

1 Einführung **2**
1.1 Die Vorteile von Vulkan 3
1.2 Die folgenden Kapitel 3
1.3 Abhängigkeiten 3
1.4 Was wird nicht besprochen 3

2 Interagieren mit der Grafikkarte **4**
2.1 Erstellen einer VkInstance 4
2.2 Erstellen eines physical devices 5
2.3 Erstellen eines logical devices 5

3 Validation Layers **7**
3.1 Hinzufügen von Validation Layers 7

4 Die Swapchain **8**

5 Frame und Command Puffer **9**

6 Die Grafikpipeline **10**

7 Das Dreieck **11**

Kapitel 1

Einführung

Diese Arbeit handelt von Vulkan, der neuesten Grafik-API von der Khronos Group, erschienen im Jahr 2016. Die Khronos Group ist unter anderem bekannt für OpenGL, ebenfalls einer Grafik-API. Vulkan dient nicht als Ersatz für OpenGL, sondern vielmehr als zweite Option. Warum braucht man eine zweite Option?
In OpenGL hatte man zunächst den Immediate Mode, mit dem einem der Großteil der Arbeit abgenommen wurde, man ziemlich schnell entwickeln konnte, aber alles unter einem Einbußen an Performance. Performance ist für Grafikanwendungen jedoch ein äußerst wichtiger Faktor, da sie eine Grenze darstellt, die manche Ideen nicht umsetzbar macht. Mit OpenGL 3.3+, auch als "Modern OpenGL"bezeichnet, wurde diese Performancegrenze angehoben. Man hat mehr Kontrolle über Geschehnisse, die vorher alle im Hintergrund passierten und hat so eine höhere Performance.
Mit Vulkan wurde diese Grenze noch weiter angehoben, aber da Modern OpenGL noch immer brauchbar, und für Entwickler viel einfacher ist, ist Vulkan, wie der Name schon sagt, ein eigenes Produkt. Vulkan soll den Overhead von Modern OpenGL noch weiter reduzieren und so ressourcenschonender und noch performanter sein. Dafür ist die API äußerst verbos und man muss einen Großteil selbst konfigurieren. Der Vorteil dabei ist, dass man als OpenGL Entwickler noch einmal sieht, was denn alles im Hintergrund für einen gemacht wird.

1.1 Die Vorteile von Vulkan

Der Hauptkonkurrent zu Vulkan ist momentan Microsoft mit DirectX 12. Um gegen so einen großen Konkurrenten bestehen zu können, braucht es klare Vorteile:
Vulkan ist plattformunabhängiger als DirectX 12. DirectX 12 unterstützt Windows 10 und XBox One X, während Vulkan Windows 7, 8 und 10, Linux, MacOS, Android und iOS untertützt. Auch ist Vulkan im Gegensatz zu DirectX 12 quelloffen. Da ich persönlich auch einen OpenGL Hintergrund habe und mein Projekt auf Linux laufen sollte, war meine Wahl einfach.

1.2 Die folgenden Kapitel

Die folgenden Kapitel sollen Meilensteine am Weg zum ersten Dreieck darstellen. In OpenGL ist das erste Dreieck eine Art Hello World, in Vulkan stellt das Dreieck weit mehr Arbeit dar, jedoch hat man zu dem Zeitpunkt die Idee Vulkans verstanden und das langwierige, aber notwendige Setup hinter einem.
Jedes Beispiel im src Ordner ist ausführbar und beinhaltet so weit wie möglich nur den Code, den man braucht, um das jeweilige Ziel (Name der Datei) zu erfüllen. Die Vorgehensweise ist dabei hauptsächlich übernommen aus dem Vulkan Tutorial.

1.3 Abhängigkeiten

Die Beispiele haben folgende Abhängigkeiten: GLFW 3.3.2 und Vulkan. GLFW3 wird hauptsächlich verwendet, um plattformunabhängig Fenster, und spezifisch für Vulkan, einen Surface zu erstellen. In surface.cpp ist die Vorgehensweise der Erstellung von einem Surface als Kommentar verfasst, aber GLFW3 macht einem die Arbeit leichter.
Wozu man Vulkan braucht, sollte selbstverständlich sein. Spezifisch habe ich das Vulkan SDK von LunarG verwendet, weil es empfohlen wird und nicht nur die Notwendigkeiten für Vulkan (Bibliothek und Header), sondern auch Testprogramme und Compiler von GLSL auf SPIR-V (wie später besprochen wird) beinhaltet.

1.4 Was wird nicht besprochen

Diese Arbeit soll keine Dokumentation von Vulkan darstellen (die wäre viel zu umfangreich und im gegebenen Zeitrahmen nicht möglich), sondern vielmehr die Vorgehensweise behandeln, wie man zu den einzelnen Meilensteinen gelangt. Das beinhaltet beispielsweise: Welche Informationen man braucht, um ein Objekt zu erstellen. Jedoch nicht: Eine detaillierte Beschreibung jedes Attributs eines structs, welches man ausfüllen muss, um ein Objekt zu erstellen.

Kapitel 2

Interagieren mit der Grafikkarte

Mit Vulkan startet man (fast) von Null. Bevor man mit Rendering beginnen kann, muss man zuerst eine Hardware auswählen, welches man zum Rendern verwenden will (physical device). Das Ziel dieses Kapitels ist es, eine Schnittstelle zu dieser Hardware (logical device) zu erstellen. Man müsste für jedes physische Gerät ein logisches Gerät erstellen, aber Multi-GPU Rendering behandelt dieses Kapitel nicht. Da wir auch nicht in tiefere Gebiete, wie Geometry Shaders, vordringen, ist die einzige Vorraussetzung eine Grafikkarte mit Vulkan Support.
Die Sections dieses Kapitels beschreiben die Beispiele:

1. vulkan_example
2. physical_devices_and_queue_families
3. logical_device

2.1 Erstellen einer VkInstance

Eine VkInstance dient quasi als Initialisierungspunkt von Vulkan, bei dem die Applikation Informationen über sich preisgeben kann. Außerdem dient die Instanz dazu, Informationen über physical devices zu bekommen. Fast alle Vulkan Objekte (gekennzeichnet durch Vk, im Gegensatz zu Funktionen vk) werden durch das Übernehmen von Informationen aus einem struct erstellt (Vk Objektname CreateInfo). So muss man, um eine Instanz zu erstellen, das struct VkInstanceCreateInfo ausfüllen. Um dieses struct zu erstellen, braucht man noch das struct VkApplicationInfo. Fast alle structs haben in Vulkan das Attribut sType. Dieses dient dazu, ein struct im Low-Level Bereich zu identifizieren und ist daher das erste Attribut in so einem struct. Ein sType ist zwar lang, aber klar definiert: VK_STRUCTURE_TYPE_NAME_OF_THE_STRUCTURE
Für VkApplicationInfo werden wir also VK_STRUCTURE_TYPE_APPLICATION_INFO angeben. Ansonsten enthält VkApplicationInfo Informationen über die Applikation, wie der Name, die Version, und die Version der verwendeten Vulkan API. Auch kann man den Namen und die Version einer Engine angeben, die man verwendet. Anhand dieser Informationen kann der Treiber die Applikation optimieren (bspw. man verwendet eine bekannte Engine).
VkInstanceCreateInfo verwendet dieses struct und fügt noch weitere Informationen hinzu,

wie die zu verwendeten Validation Layers (besprochen im nächsten Kapitel) und Extensions, die wir benötigen. Vulkan selbst ist wie ein Fundament zu sehen, auf dem man mit Extensions bauen kann. So muss man beispielsweise Extensions hinzufügen, damit Vulkan mit dem Fenster interagieren kann.
GLFW3 hat dafür die Funktion glfwGetRequiredInstanceExtensions, welche die notwendigen Extensions zurückliefert, die man dann angeben kann.
Anschließend kann man mit vkCreateInstance die Instanz erstellen. Vulkan bietet einem dabei die Möglichkeit, Callbacks für einen Allocator anzugeben, wenn man denn einen eigenen verwenden möchte. Wir werden das nicht, und geben für so einen Parameter nullptr an. Außerdem liefert eine vkCreate Funktion typischerweise ein VkResult zurück, welches man auf Werte überprüfen kann. Hat es den Wert VK_SUCCESS, war das Erstellen erfolgreich, jedoch gibt es noch andere Werte, wie z.B.: VK_ERROR_OUT_OF_HOST_MEMORY oder VK_ERROR_FEATURE_NOT_PRESENT. Letzterer Fehler wird in der nächsten Section mehr Sinn ergeben.

2.2 Erstellen eines physical devices

Eine Grafikanwendung hat typischerweise Anforderungen an Grafikkarten, wie zum Beispiel, dass die Grafikkarte ein spezielles Feature, wie Geometry Shaders, unterstützt, oder dass die Grafikkarte aufgrund der Rechenleistung dediziert ist. Die Wahl des richtigen physical devices behandelt diese Themen. Mit vkEnumeratePhysicalDevices bekommt man mithilfe der VkInstance die physical devices im System zurückgeliefert. Danach kann man mit vkGetPhysicalDevice(Properties/Features) bestimmte Properties und Features abfragen, die das physical device hat (Beispiel in Execute()). Man könnte dann die Features und Properties gewichten und das Gerät wählen, dass am besten diese Properties und Features unterstützt (im Idealfall alle gebrauchten).
Damit ist man jedoch noch nicht ganz fertig: Man muss auch überprüfen, ob das gewünschte physical device bestimmte Queues hat. Eine VkQueue wird verwendet, um Commands auszuführen (wird später besprochen). Für uns ist momentan wichtig, dass es eine Queue gibt, an die man Commands zum Rendern abgeben kann. Dafür überprüfen wir die Properties der queues auf dem physical device. Diese bekommen wir mithilfe von vkGetPhysicalDeviceQueueFamilyProperties. Die Art der Queue ist im Attribut queueFlags spezifiziert. Wir wollen, dass das VK_QUEUE_GRAPHICS_BIT gesetzt ist, also überprüfen wir das und speichern uns den Index der QueueFamily ab. Nachdem hoffentlich beide Anforderungen erfüllt sind, haben wir unser physical device gewählt! Jetzt müssen wir noch ein logical device erstellen, mit dem Vulkan mit dem physical device interagieren kann.

2.3 Erstellen eines logical devices

Bei der Erstellung des VkDevice (logical devices) legt man fest, wieviele Queues man für welche QueueFamilies erstellen will (im Normalfall braucht man nur eine Queue pro QueueFamily), und welche Features und physical device Extensions man verwenden will. Dafür benötigt man die structs VkDeviceQueueCreateInfo und VkDeviceCreateInfo.
In VkDeviceQueueCreateInfo gibt man an, wieviele Queues man für welche QueueFamily mit dem VkDevice erstellen will, sowie die Priority der Queues, welche am Ende das Scheduling betrifft. Für jede QueueFamily muss man so ein struct erstellen.
In VkDeviceCreateInfo gibt man die Adresse zu den VkDeviceQueueCreateInfo structs

an, sowie die Anzahl der VkDeviceQueueCreateInfo structs, wie es in C üblich ist. Wenn es nur ein struct ist, wie in diesem Fall, reicht die Adresse zu diesem und 1 für die Anzahl. Des weiteren braucht man die Adresse zu einem VkPhysicalDeviceFeatures struct, der die Information enthält, ob das Feature aktiv, oder nicht aktiv sein soll. Da wir keine besonderen Features verwenden, erstellen wir einen VkPhysicalDeviceFeatures struct, lassen alles mithilfe der Default-Initialization auf VK_FALSE und geben die Adresse von diesem an. Zu guter Letzt müssen wir noch die Extensions für das physical device angeben. Da wir für das physical device keine brauchen, lassen wir einfach den count auf 0.
Jetzt können wir mit vkCreateDevice ein logical device erstellen. Die verlangten Queues bekommen wir mit vkGetDeviceQueue. Da geben wir neben dem logical device den Index der QueueFamily an, sowie den Index der Queue (benötigt, wenn man mehr als eine Queue für die QueueFamily angegeben hat, bei uns ist der Index daher 0) und die Adresse einer VkQueue, wo man die Queue speichern will.
Nun haben wir ein logical device und können theoretisch schon anfangen, die Swap Chain zu definieren, jedoch handelt das nächste Kapitel erstmal von einer optionalen Möglichkeit zu Debuggen - den Validation Layers.

Kapitel 3

Validation Layers

Da Vulkan eine sehr verbose API ist, kann sich bei der Menge an Informationen, die man angeben kann, und muss, leicht ein Fehler einschleichen. Validation Layers dienen dazu, unter anderem solche Fehler aufzudecken und stellen quasi Sicherheitsschichten dar, die man durchdringen muss, damit man zur eigentlichen Funktionalität kommt. Klarerweise kann das nur einen Nachteil haben - Performance. Daher aktiviert man sie generell in einem Debug Build und im Release Build kann man sie ganz einfach nicht aktivieren. Der Vorteil dabei ist, dass man, auch wenn man sie aktiviert, man nur die aktivieren kann, die man braucht. Eine Standard Validation Layer ist VK_LAYER_KHRONOS_validation, welche ich im Laufe dieses Guides verwendet habe.
Noch mehr Informationen zu Validation Layers gibt es unter GPU Open
Die Section dieses Kapitels beschreibt das Beispiel:

1. validation_layers

3.1 Hinzufügen von Validation Layers

Wie wir schon erfahren haben, werden Validation Layers mit dem Erstellen einer VkInstance aktiviert. Das heißt, wir müssen, wie bei den Extensions die Anzahl und Namen der gewünschten Validation Layers im VkInstanceCreateInfo struct angeben. Dazu müssen wir erst einmal wissen, welche Validation Layers es überhaupt gibt. Mit vkEnumerateInstanceLayerProperties kann man diese Information bekommen. Im Sinne dieses Beispiels gibt es auch noch die Methode CheckIfLayerValid, die überprüft, ob die gewünschte Layer vorhanden ist, bevor sie einem Array hinzugefügt wird. Der Array und deren Größe werden dann der CreateInstance Methode vom ersten Beispiel mitgegeben und mit diesen Informationen sind die gewünschten Validation Layers aktiviert.
Will man sich mit diesen ein wenig spielen, enthält die Execute Methode ein kleines Programm, mit dem man verfügbare Layers mit $PRINT ausgeben kann, Layers hinzufügen kann, und mit $FINISH zuletzt eine VkInstance mit den angegebenen Layers erstellen kann. Wie schon erwähnt werden wir uns jedoch im weiteren Verlauf mit der Layer VK_LAYER_KHRONOS_validation begnügen.

Kapitel 4

Die Swapchain

Kapitel 5

Frame- und Commandbuffers

Kapitel 6

Die Grafikpipeline

Kapitel 7

Das Dreieck