

**Entwicklung eines
plattformunabhängigen
Echtzeit-Renderers mit der Vulkan API
in C++**

PRAXISPROJEKT
ausgearbeitet von

Paul André Johne
11138534

vorgelegt an der
TECHNISCHEN HOCHSCHULE KÖLN
CAMPUS GUMMERSBACH
FAKULTÄT FÜR INFORMATIK UND
INGENIEURWISSENSCHAFTEN

im Studiengang
MEDIENINFORMATIK

Erster Prüfer/in: Prof. Dr. rer. nat. Wolfgang Konen
Technische Hochschule Köln

Zweiter Prüfer/in: Fabian Friederichs
Technische Universität Braunschweig

Gummersbach, 17. August 2023

Adressen: Paul André Johne
Weststraße 30
51643 Gummersbach
paul_andre.johne@th-koeln.de

Prof. Dr. rer. nat. Wolfgang Konen
Technische Hochschule Köln
Institut für Informatik
Steinmüllerallee 1
51643 Gummersbach
wolfgang.konen@th-koeln.de

Fabian Friederichs
Technische Universität Braunschweig
Institut für Computergraphik
Mühlenpfordtstraße 23
38106 Braunschweig
friederichs@cg.cs.tu-bs.de

Abstract

The practical project aims to create another entry point for working with the Vulkan API in order to go deeper into the individual concepts of the said API on basis of a given source code. For this purpose, the PJEngine was developed including the other libraries GLFW, GLM, Assimp and stb image into the Vulkan context to make the Vulkan API usables as a platform-independent real-time renderer.

Features such as mip-mapping, depth testing and MSAA were exemplary implemented with the PJEngine. The application provides a VSync and a performance mode with dynamic window sizing.

Kurzfassung

Ziel des Praxisprojektes war es einen weiteren Einstiegspunkt für das Arbeiten mit der Vulkan API zu schaffen, um auf Grundlage eines Quellcodes tiefergehend auf einzelnen Konzepte der besagten API einzugehen. Dazu wurde die PJEngine entwickelt, welche die weiteren Bibliotheken GLFW, GLM, Assimp und stb image in den Vulkan Kontext integriert, welche die Vulkan API als einen plattformunabhängige Echtzeitrenderer verwendbar machen.

Mit der PJEngine wurden Features wie das Mip-Mapping, ein Tiefentest und das MSAA implementiert. Die Anwendung bietet einen VSync- und einen Performance Modus mit dynamischer Fenstergröße.

Inhaltsverzeichnis

1 Einleitung	1
2 Softwarearchitektur der PJEngine	2
2.1 Ordnerstruktur	3
2.2 Softwarekompilierung mit CMake	4
3 Vulkan Konzepte anhand der PJEngine	5
3.1 Der Vulkan Loader und seine geschichtete Architektur	5
3.2 Von der VkApplicationInfo bis zum logischen VkDevice	8
3.3 Shadermodule	11
3.4 Synchronisationsprimitive von Vulkan	12
3.5 Swapchain	15
3.5.1 Rendertargets	17
3.6 Grafikpipeline und Renderpass	18
3.7 Command Buffer und Framebuffer	23
3.8 Shaderressourcen	25
3.8.1 Modelldaten	26
3.8.2 Descriptorsets	28
3.9 Aufnahme eines Renderbefehles und dessen Präsentation	30
3.10 Cleanup von Vulkan Ressourcen	32
4 Weitere PJEngine-relevante Bibliotheken	33
4.1 GLFW : Cross-Platform Fenstergenerierung	33
4.2 GLM : Bibliothek für Vektormathematik	33
4.3 Assimp : Der universelle Modelloader	34
4.3.1 stb image : Der universelle Textureloader	34
5 Resultate	35
6 Diskussion	39
6.1 Bedeutung der Arbeit	39
6.2 Schwierigkeiten bei der Umsetzung des Projektes	40
7 Fazit	41
Literatur	i
Eidesstattliche Erklärung	iii

Abbildungsverzeichnis

2.1	Softwarearchitektur der PJEngine [Joh23]	2
2.2	Ablaufsdiagramm der PJEngine [Joh23]	3
2.3	Ordnerstruktur der PJEngine [Joh23]	3
2.4	CMakeSettings.json der PJEngine [Joh23]	4
3.1	Architektur des Vulkan Loaders nach [Lun]	5
3.2	Instance Call Chain aus [Lun]	6
3.3	Device Call Chain aus [Lun]	6
3.4	Visualisierung der Erstellung von Call Chains	7
3.5	Visualisierung des Barrier-Primitives	14
3.6	Grafik aus [SK17, S. 227]	18
3.7	Aufbau des Renderpass des Renderers der PJEngine	20
3.8	Lebenszyklus von Command Buffer nach [Khr23, unter 6.1. Command Buffer Lifecycle]	24
3.9	Beispiel eines Descriptorsets nach [Bla20, unter 4. Descriptor Sets]	28
3.10	Arbeitablauf mit der Swapchain	30
5.1	Testmodell für die PJEngine (erstellt in Blender)	35
5.3	ausgewählte Mip-Map-Level der Testmodell-Textur auf der GPU	36
5.4	Integration des Tiefentest	37
5.5	Rendermodi mit eingeschaltetem ImGui	37
5.6	Auslastung der GPU/CPU je Modus	38

Listings

3.1	VkInstanceCreateInfo	8
3.2	Beispiel einer vkEnum Funktion	9
3.3	Ausschnitt einer VkDeviceQueueCreateInfo	10
3.4	VkDeviceCreateInfo der PJEngine	10
3.5	Erstellen eines VKDevice	10
3.6	Erhalten eines VkQueue Handles eines VkDevices	11
3.7	Generierung von Shadermodulen	11
3.8	Ausschnitt aus <code>createShaderModule</code>	11
3.9	Deklaration von Shaderstages	12
3.10	Erstellung der Semaphore	13
3.11	Erstellung von Fences	13
3.12	Ausschnitt der Erstellung einer Swapchain	15
3.13	Erhalten von VkImage und VkImageView von einer Swapchain	16
3.14	Datentyp PJImage mit default Werten	17
3.15	Ausschnitt aus <code>allocateVkImageMemory</code>	17
3.16	Ausschnitt aus <code>getMemoryTypeIndex</code>	17
3.17	Ausschnitt aus der Erstellung einer Grafikpipeline	19
3.18	Ausschnitt aus der Erstellung von VkAttachmentDescriptions	21
3.19	Subpass-Deklaration mit Referenzierung auf VkAttachmentDescriptions	21
3.20	Ausschnitt aus der Erstellung von VkSubpassDependencies	22
3.21	Ausschnitt aus der Erstellung eines Renderpass	23
3.22	Ausschnitt aus der Framebuffer-Erstellung für den Renderpass	24
3.23	Relevanter Teil des Vertex Shaders vom Renderer	25
3.24	Relevanter Teil des Fragment Shaders vom Renderer	25
3.25	Datentyp PJBuffer mit default Werten	27
3.26	Bereitstellung der Daten in Vulkan Objekte	27
3.27	Erstellung und Linken von Ressourcen für ein Descriptorset	29
3.28	Ausschnitt aus der Aufzeichnung eines Renderbefehles	31
4.1	PJModel Klasse	34

1 Einleitung

Die vorliegenden Ausarbeitung untersucht wie die Technologie der Vulkan API innerhalb eines CMake Projektes einsetzbar ist, um eine Anwendung von in Echtzeit generierten Bildern zu erstellen. Hierzu wird sich auf den Ansatz von [Joh23, s. Branch *Praxisprojekt*] bezogen.

Unter der Forschungsfrage „Wie kann mit bestehender Technologie eine Vulkan-gestützte Echtzeitanwendung zur Darstellung von 3D Objekten realisiert werden?“ richtet sich die Arbeit an eine Zielgruppe mit Erfahrung in der Softwareentwicklung und mit Vorkenntnissen im Bereich OpenGL.

Die Vulkan API ermöglicht es Kontrolle über vorhandene GPUs in der eigenen Anwendung zu erhalten und gilt nach [SK17, S. xxii] als logischer Nachfolger von OpenGL. Die einzelnen Befehle und Daten von Vulkan werden von entsprechenden Treibern für die eigentliche Hardware übersetzt [SK17, S. 2].

Vulkan ist verbesser als Alternativen wie OpenGL und DirectX11, da nun die eigene Anwendung Verantwortlichkeiten übernimmt, welche zuvor noch von einem Treiber behandelt wurden [SK17, S. xxii]. Der verbose Charakter von Vulkan liefert den GPU Herstellern eine Eindeutigkeit, sodass deren Treiber folglich nicht mehr auf Edge Cases anzupassen sind. Durch die erhöhte Programmierarbeit des Anwendungsentwicklers entstehen jedoch mehr potenzielle Fehlerquellen. Dem stehen mehr Kontrolle, wie beispielsweise die explizite Auswahl von Hardware-Ressourcen, und eine viel bessere Performance [SK17, S. 2] positiv entgegen. Zudem bieten Features wie der *Validation Layer* eine konkrete Form der Fehleranalyse von Code, welcher auf der GPU ausgeführt wird, was bei OpenGL nur mit vagen Vermutungen erfolgt. Vulkan Anwendungen haben die Möglichkeit Overhead, welcher beispielsweise durch Validierungsfunktionalitäten entsteht, modular ein- und auszuschalten, was in vorherigen APIs nicht möglich ist [Lun, unter Application Layer Usage]. Weitere Vorteile von Vulkan stellen die manuelle Speicherverwaltung, die manuelle Synchronisation zur Performanceoptimierung [Mun+21] und die Zusammenfassbarkeit von Befehlen auf der CPU in Command Buffer für die GPU dar.

In Kapitel [Softwarearchitektur der PJEngine](#) wird zunächst Umsetzungsentscheidung der Architektur begründet und verdeutlicht, wie der Code kompiliert wird. Das Kapitel [Vulkan Konzepte anhand der PJEngine](#) stellt den Kern der Ausarbeitung dar und geht näher auf die einzelnen verwendeten Konzepte der Vulkan API ein und verbindet diese mit Codeabschnitten aus [Joh23, s. Branch *Praxisprojekt*]. Prinzipiell können jegliche Konzepte der Vulkan API auch mit der offiziellen Spezifikation [Khr23] er-schlossen werden. Anschließend werden zusätzliche Bibliotheken, die Daten für die PJEngine bereitstellen in Kapitel [Weitere PJEngine-relevante Bibliotheken](#) flüchtig behandelt.

Als initiale Quelle für den eigenen Lernprozess gilt die Videoreihe von [Bro17, Video 1 bis 100], welche auf die Ausarbeitung von [Ove23] basiert.

2 Softwarearchitektur der PJEngine

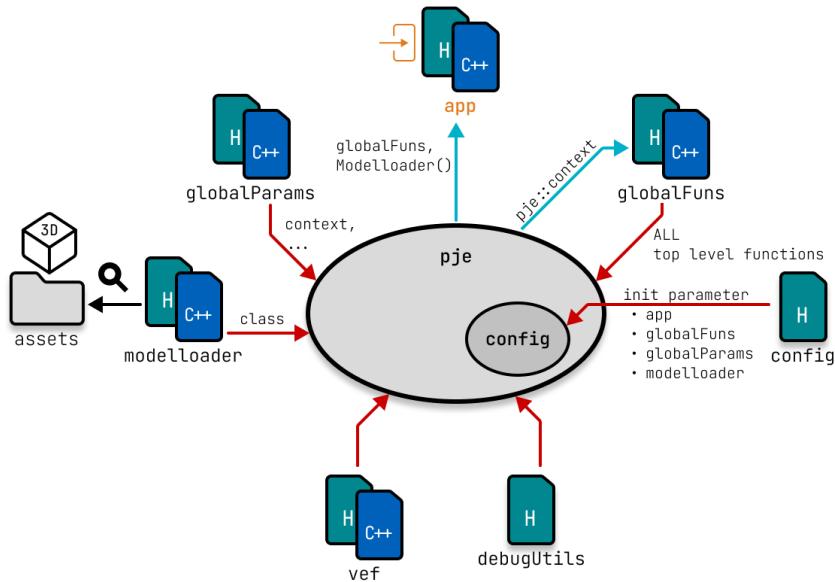


Abbildung 2.1: Softwarearchitektur der PJEngine [Joh23]

Das Projekt wurde als CMake Projekt in *Visual Studio* aufgesetzt und mit einem Fokus auf die Darstellung einzelner Vulkan Funktionalitäten entworfen. Um möglichst wenig Abstraktion durch die eigene Implementation zu generieren, verfolgt das Projekt einen prozeduralen Charakter. Abbildung 2.1 zeigt, dass sämtliche Dateien an einem *Namespace*¹ namens *pje* mitwirken. Es werden globale Variablen, *Macros*², Funktionen und Klassen bereitgestellt, welche von anderen Teilen (*Source* und *Header Dateien*) verwendet werden.

Die globale Variable *context* vom Typ *Context* beeinhaltet unter anderem jegliche *Handle-Variablen* zu Vulkan Objekten, die zur Programmausführung an bestimmter Stelle erstellt und zum späteren Zeitpunkt an anderer Stellen benötigt werden.

Der *Subnamespace* *pje::config* dient als Schnittstelle für den Benutzer, um vor der Kompilierung noch einige Einstellungen an dem Renderer vorzunehmen.

Der gesamte Renderer wird aufseiten der CPU nur auf einem *Thread* ausgeführt. Dieser befindet sich während der Aufführung des Programms hauptsächlich in einer Schleife, in welcher ein Renderbefehl für den nächsten zu rendernden Frame bestimmt wird und anschließend an die GPU versendet wird.

¹Scope in C++ für einzelne Identifier wie Variablen und Funktionen

²C++ Identifier für einen Wert, der an geeigneter Stelle eingesetzt wird

2 Softwarearchitektur der PJEngine

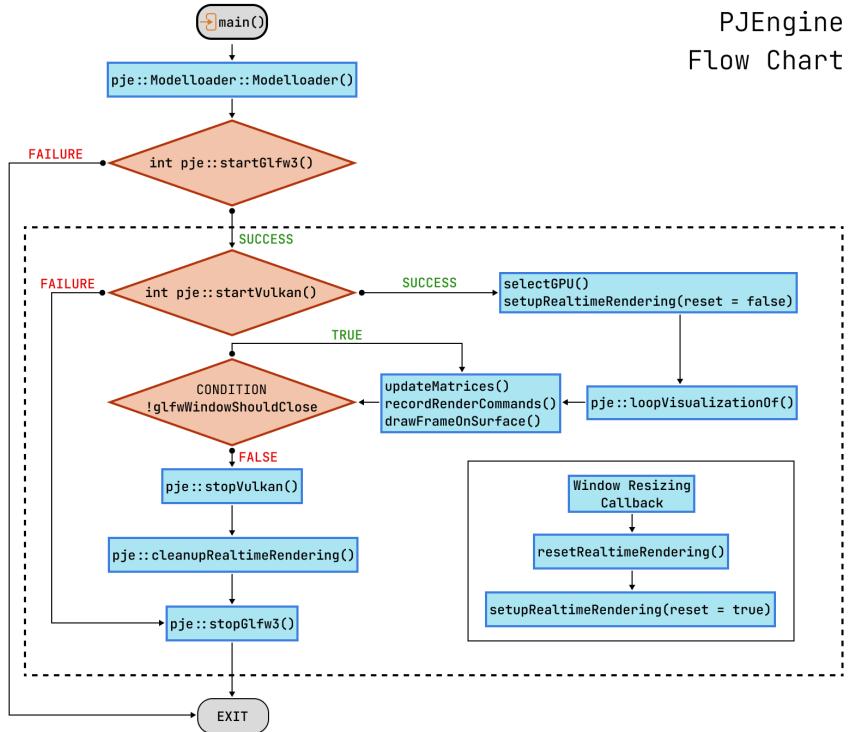


Abbildung 2.2: Ablaufsdiagramm der PJEngine [Joh23]

2.1 Ordnerstruktur

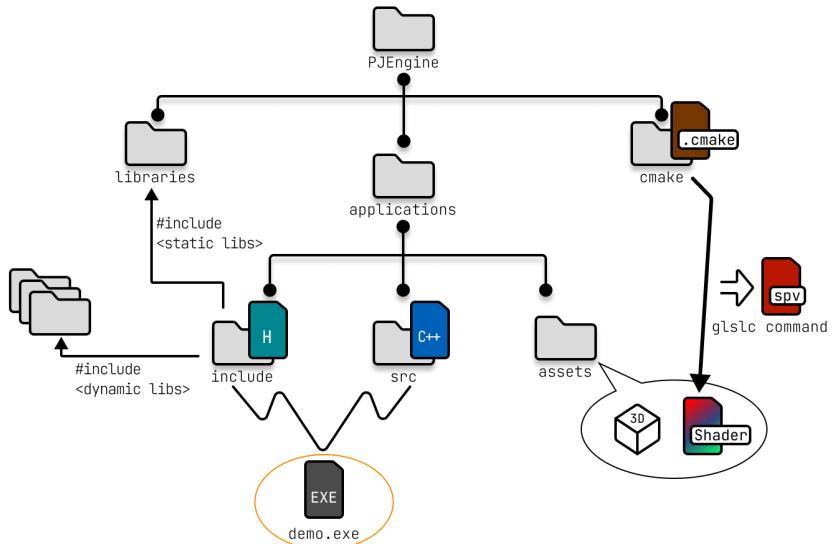


Abbildung 2.3: Ordnerstruktur der PJEngine [Joh23]

Um die einzelnen Komponenten (Bibliotheken, Quellcode, CMake-Funktionen und Assets) des CMake Projektes logisch zu trennen, wurde die Struktur, die in Abbildung 2.3 zu erkennen ist, gewählt. Damit ein „CMake Generator“[Inc, unter cmake-generators(7)] das Projekt auf allen Ebenen der Hierarchie erkennt, werden in den einzelnen Ordnern *CMakeLists.txt* Dateien mit CMake-Code benötigt, wobei jede *CMakeLists.txt* mit dem Befehl `add_subdirectory()` Unterverzeichnisse dem *Build* hinzufügt.

2.2 Softwarekomplilierung mit CMake

Zur Bewahrung der Plattformunabhängigkeit wurde sich für *CMake* als *Build System* entschieden. CMake kann hierbei als *Meta-Build System* gesehen werden, da es die *Build Files* eines anderen Build Systems generiert [Inc, unter cmake-generators(7)]. Ein Build System stellt dabei eine Ansammlung an Werkzeugen dar, die anhand der gegebenen Build File und dem *Source Code* plattformspezifischen Binärkode (auch *Build* genannt) erstellt.

Bekannte Build Systems sind unter anderem *Make* und *Ninja*. Für das Projekt wurde sich in der IDE Visual Studio für Ninja entschieden, um die ausführbare Datei *demo.exe* zu erhalten (s. Abbildung 2.4).

Der Shadercode wurde über eine, vom Betreuer Fabian Friederichs bereitgestellten, CMake Funktion zur *Compile-Zeit* in SPIR-V Dateien gewandelt. Die Datei mit der Funktionsdefinition befindet sich im `cmake` Ordner. Prinzipiell führt diese Funktion für alle Shaderdateien eine `glslc -o` *Command-Zeile* aus, welche SPIR-V generiert. SPIR-V ist die einzige offiziell unterstützte Shadersprache von Vulkan [SK17, S. 169].

Wie die Abstraktion des *Build Prozesses* mit CMake im Fall der PJEngine explizit umgesetzt wurde, ist anhand der Kommentare innerhalb der einzelnen *CMakeLists.txt* Dateien zu erfassen. Mehr Informationen zu den einzelnen CMake Befehlen sind in [Inc] zu finden.



Abbildung 2.4: CMakeSettings.json der PJEngine [Joh23]

3 Vulkan Konzepte anhand der PJEngine

Die Softwarearchitektur, wie sie im Kapitel [Softwarearchitektur der PJEngine](#) vorgestellt wurde, verfolgt die Absicht den Abstraktionsgrad möglichst gering zu halten, um durch seinen prozeduralen Charakter einen Fokus auf die Funktionalität der Vulkan API zu setzen. Die folgenden Abschnitte greifen die genutzten Konzepte der Vulkan API innerhalb der PJEngine [Joh23] in Reihenfolge des Programmablaufes auf.

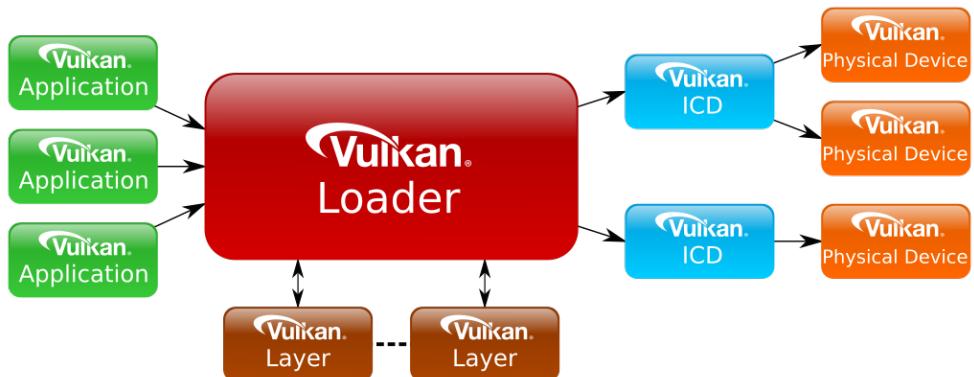


Abbildung 3.1: Architektur des Vulkan Loaders nach [Lun]

Wie in Abbildung 3.1 verdeutlicht, erfolgt der Zugriff auf Funktionen von Vulkan über den Vulkan Loader, welcher die Befehle an die ICDs, die *Installable Client Driver*, weiterleitet. Ein ICD ist ein Treiber eines GPU Herstellers, welcher die eigentliche Umsetzung des jeweiligen Befehls nach der Vulkan Spezifikation enthält. Dabei kommt es zu einer direkten und herstellerspezifischen Kommunikation mit der Hardware.

3.1 Der Vulkan Loader und seine geschichtete Architektur

Der Vulkan Loader agiert zwischen der eigenen Anwendung und den ICDs mit dem Ziel die gegebenen Funktionen der API unter Einbindung diverser Vulkan Layers und ICDs richtig zu verarbeiten [Lun, unter The Loader].

Nach [SK17, S. 23] stellen Vulkan Layers optionale Komponenten dar, die bestehende Funktionen abfangen und auf verschiedenste Weise mit diesen interagieren. Jedoch muss nicht jeder Layer alle Funktionen abfangen, sondern entscheidet selbst, welche Vulkan Funktionen für ihn von Relevanz sind [Lun, unter Layers]. Um einen Layer zu aktivieren, wird dessen Name dem Member `ppEnabledLayerNames` des struct¹ `VkInstanceCreateInfo` mitgeteilt [Lun, unter Application Layer Usage]. Mit dieser CreateInfo kann ein `VKInstance` Objekt erstellt werden, welches „per-application“

¹Schlüsselwort für zusammengesetzte Datentypen in C++

3 Vulkan Konzepte anhand der PJEngine

[Khr23, unter 4.2. Instances] Zustandsinformationen wie die konkreten Vulkan Layers hält. Damit ist die Instance die logisch äußerste Schicht einer Vulkan Anwendung.

Vulkan Objekte wie `VkInstance` werden von Vulkan in „dispatchable Objects“ und „nondispatchable Objects“ unterschieden. Die Bezugnahme auf jene erfolgt über *Handles*. Jede Funktion, außer welche dispatchable Objects erstellen (darunter z.B. `vkCreateInstance`), verlangt als ersten Parameter immer ein dispatchable Object [SK17, S. 18]. Aus der Dokumentation des Vulkan Loader [Lun, unter Instance Versus Device] geht hervor, dass Objekte entweder „Instance-related“ oder „Device-related“ sind. Einer Device-Funktion muss deshalb nicht immer als ersten Parameter ein dispatchable Object von `VkDevice` übergeben werden, sondern kann dispatchable Objects wie `VkQueue`, welche mit einem `VkDevice` assoziiert werden, ebenfalls als ersten Parameter erwarten.

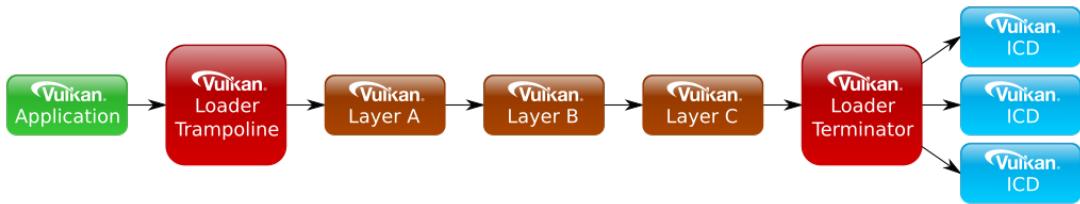


Abbildung 3.2: Instance Call Chain aus [Lun]

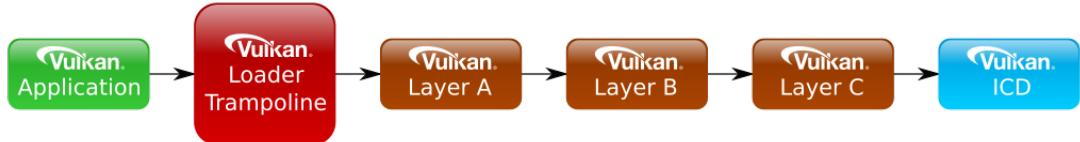


Abbildung 3.3: Device Call Chain aus [Lun]

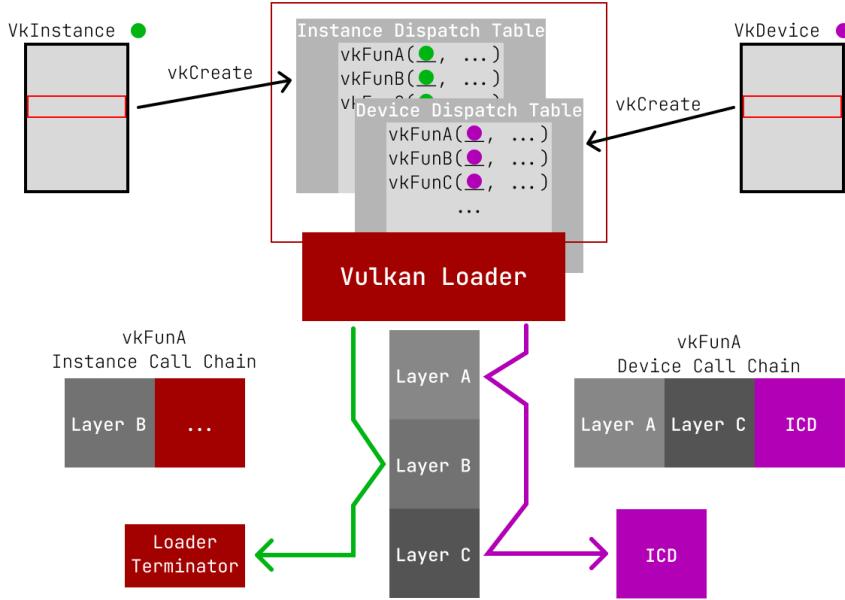


Abbildung 3.4: Visualisierung der Erstellung von Call Chains

Die Abbildungen 3.2 und 3.3 präsentieren mögliche Call Chains, die aus einer Instance oder Device Funktion generiert wurden. Bei der Erstellung der dispatchable Objects `VkInstance` und `VkDevice` werden vom Vulkan Loader aus allen Instance Funktionen bzw. Device Funktionen Instance bzw. Device Call Chains generiert [Lun, unter Dispatch Tables and Call Chains]. Diese Call Chain ist der jeweiligen `VkInstance` oder dem `VkDevice` zugeordnet und definiert, welche Layers und ICDs involviert sind, wenn jene Vulkan Funktion für ein bestimmtes `VkInstance`, `VkDevice` oder einer der beiden zugehörigen dispatchable Objects aufgerufen wird.

Die prinzipielle Reihenfolge der Layers in den Call Chains legt der Loader über die Deklarationsfolge in `ppEnabledLayerNames` von `VkInstanceCreateInfo` fest [Lun, unter Application Layer Usage]. Die Reihenfolge besitzt eine gewisse Form von Wichtigkeit, da, wie in [Lun, unter Overall Layer Ordering] erwähnt, manche Layers vom Verhalten vorheriger Layers beeinflusst werden.

Zur Verkettung der Elemente einer Call Chain werden intern *Dispatch Tables* vom Loader und den Layers bei der Erstellung von einem `VkInstance` bzw. `VkDevice` angelegt. Diese enthalten eine Liste an Funktionszeigern, die auf die Funktion des nächsten Elementes der Call Chain zeigen [Lun, unter Dispatch Tables and Call Chains]. Die Instance bzw. Device Dispatch Table, wie sie in Abbildung 3.4 zu erkennen ist, enthalten die Pointer auf das jeweilige erste Element einer Call Chain. Verwaltet wird diese Tabelle vom Vulkan Loader.

Die Layers erhalten Vorgaben für die Instance und Device Dispatch Table, sind aber in ihrer Umsetzung frei [Lun, unter Layer Dispatch Initialization]. Der Loader stellt unter `VkLayer*CreateInfo->u.pLayerInfo` die Reihenfolge der Layers den Layers zur Verfügung.

Abschließend lässt sich über den Vulkan Loader die Aussage treffen, dass sämtliche Aufgaben, welche dieser erfüllt, auch manuell getätigten werden können. Dazu müssen Dispatch Tables selbstständig mit `vkGetInstanceProcAddr()` bzw. `vkGetDeviceProcAddr()` aufgebaut werden, allerdings muss man dazu die verfügbaren Treiber und Geräte unterschiedlicher Hersteller in ihrer Funktionsweise kennen und deren Abstimmung aufeinander manuell durchführen. Portable Code ist auf diesem Weg in der Praxis nur sehr schwer zu erreichen.

3.2 Von der `VkApplicationInfo` bis zum logischen `VkDevice`

Zur Initialisierung der Vulkan Anwendung wird in der PJEngine die Funktion `startVulkan()` aufgerufen. Hierzu wird zunächst mit dem aus Sektion 3.1 erwähnten `VkInstanceCreateInfo` eine Instance mittels `vkCreateInstance()` erstellt.

Sämtlicher von Vulkan zur Verfügung gestellter Code wie Funktionen und Klassen sind mit einem `vk`, `Vk` bzw. `VK` vermerkt. Das Ausfüllen von Infoobjekten, wie in der folgenden Abbildung zu sehen, gehört zu den üblichen Aufgaben, welche beim Arbeiten mit Vulkan auftreten.

```
VkInstanceCreateInfo instanceInfo;
instanceInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
instanceInfo.flags = 0;
instanceInfo.pApplicationInfo = &appInfo;
instanceInfo.pNext = &validationFeatures;
instanceInfo.enabledLayerCount = static_cast<uint32_t>(
    usedInstanceLayers.size());
instanceInfo.ppEnabledLayerNames = usedInstanceLayers.data();
instanceInfo.enabledExtensionCount = static_cast<uint32_t>(
    usedInstanceExtensions.size());
instanceInfo.ppEnabledExtensionNames = usedInstanceExtensions.data();
```

Listing 3.1: `VkInstanceCreateInfo`

Das Infoobjekt `instanceInfo` in Listing 3.1 speichert neben den zu verwendenden Layers und Instance Extensions auch ein Pointer auf ein Infoobjekt (`appInfo`) vom Typ `VkApplicationInfo` mit weiteren Metadaten. Der `sType` teilt den Treiber mit, um was es sich bei dem jeweiligen Objekt handelt. Dieser Parameter kommt noch bei einigen der folgenden aufgegriffenen Vulkan Strukturen vor.

Extensions erweitern die Core API von Vulkan auf Instance oder Device Ebene. Sie definieren unter anderem neue Funktionen und Typen, welche dann als Teil der API im Instance bzw. Device Kontext gesehen werden [SK17, S. 27]. Zur Realisierung des Echtzeit- Renderers wird unter anderem auf Instance Ebene die `VK_KHR_surface` Extension benötigt, die in `usedInstanceExtensions` (s. Listing 3.1) namentlich Erwähnung findet. Diese Extension führt den `VkSurfaceKHR` Handle ein, das eine plattformspezifische Schnittstelle zur Darstellung der gerenderten Daten für die Präsentation bietet [SK17, S. 138]. Die PJEngine nutzt die GLFW Funktion `glfwCreateWindowSurface()` zur Generierung eines `VkSurfaceKHR`. Inwieweit GLFW in der PJEngine Anwendung findet, wird in [GLFW : Cross-Platform Fenstergenerierung](#) behandelt.

In `usedInstanceLayers` aus Listing 3.1 wurde der `VK_LAYER_KHRONOS_validation` Layer namentlich an das Infoobjekt `instanceInfo` übergeben. Dieser ermöglicht ein

Debugging von Vulkan Funktionen. Mit `validationFeatures` wurde eine Extension für diesen Layer integriert, welche jedoch in dieser Arbeit keine weitere Relevanz hat. Manche Extensions benötigen zusätzlich zum Namen in `ppEnabledExtensionNames` ein Konfigurations-Struct, welches über den `pNext` Pointer übergeben wird. Jedes dieser Structs hat wiederum einen `pNext` Pointer. Somit können beliebig viele Zusatzinformationen angegeben werden. Diese *Struct-Chains* sind ein wiederkehrendes Design Element in der Vulkan API und bieten neben der Flexibilität einen Erweiterungsraum für zukünftiger API Versionen.

```
/* Ermittelt Anzahl vorhandener Layer */
uint32_t numberofInstanceLayers = 0;
vkEnumerateInstanceLayerProperties(&numberofInstanceLayers, nullptr);

/* Schreibt n vorhandene Layer in availableInstanceLayers */
auto availableInstanceLayers = vector<VkLayerProperties>(
    numberofInstanceLayers);
vkEnumerateInstanceLayerProperties(&numberofInstanceLayers,
    availableInstanceLayers.data());
```

Listing 3.2: Beispiel einer `vkEnum` Funktion

Zur Ermittlung von vorhandenen Layers, Extensions und GPUs wurden `vkEnum` Instance Funktionen wie in Listing 3.2 ausgeführt. Hierbei ruft der Terminator aus Abbildung 3.2 alle ICDs auf und sammelt die Resultate [Lun, unter Instance Call Chain Example]. Dadurch ist eine Liste an Layers usw. möglich.

Nach der Erstellung der Instance wurde mit der Liste an GPUs in `selectGPU()` und mit den Einstellungen aus der `config.h` Datei eine geeignete GPU mitsamt Queue Family für den Echtzeit-Renderer ausgewählt.

Queue Families fassen Queues, welche die abzuarbeitenden Befehle für die GPU enthalten, zu Gruppen mit identischen Eigenschaften zusammen [SK17, S. 12]. Die Queues sind physische Teile einer GPU. Zu welchen Berechnungen eine Queue fähig ist, wird anhand der `VkQueueFlags` einer Queue Family abgefragt. Eine Kombination aus den folgenden Flags ist möglich:

- `VK_QUEUE_GRAPHICS_BIT`
(grafikbezogene Anweisungen)
- `VK_QUEUE_COMPUTE_BIT`
(z.B. Ausführung von *Compute Shadern* ohne grafischen Output)
- `VK_QUEUE_TRANSFER_BIT`
(Transferoperationen von Daten zwischen Hostsystem und GPU)
- `VK_QUEUE_SPARSE_BINDING_BIT`
(Operationen im Zusammenhang mit *Sparse Resources* [Khr23, unter 33. Sparse Resources])

Die PJEngine benötigt zum Rendern unter anderem eine Queue einer Queue Family, die `VK_QUEUE_GRAPHICS_BIT` erfüllt, um grafische Operationen durchzuführen.

3 Vulkan Konzepte anhand der PJEngine

```
VkDeviceQueueCreateInfo deviceQueueInfo;
deviceQueueInfo.queueFamilyIndex = pje::context.choosenQueueFamily;
deviceQueueInfo.queueCount = 1;
```

Listing 3.3: Ausschnitt einer VkDeviceQueueCreateInfo

```
/* Device Features ab Vulkan 1.1 */
VkPhysicalDeviceFeatures2 coreDeviceFeature {
    VkStructureType::VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FEATURES_2
};

VkDeviceCreateInfo deviceInfo;
deviceInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
deviceInfo.pNext = &coreDeviceFeature;
deviceInfo.queueCreateInfoCount = 1;
deviceInfo.pQueueCreateInfos = &deviceQueueInfo;
deviceInfo.enabledLayerCount = 0;
deviceInfo.ppEnabledLayerNames = nullptr;
deviceInfo.enabledExtensionCount = static_cast<uint32_t>(
    deviceExtensions.size());
deviceInfo.ppEnabledExtensionNames = deviceExtensions.data();
deviceInfo.pEnabledFeatures = nullptr;
```

Listing 3.4: VkDeviceCreateInfo der PJEngine

Mit `deviceInfo` aus Listing 3.4 kann mittels `vkCreateDevice()` eine logische Referenz (`VkDevice`) auf ein physisches Gerät erstellt werden. Diese Softwareabstraktion auf ein physisches Gerät gestaltet sich spezifisch so, wie es die eigene Anwendung verlangt [SK17, S. 7]. Es sind mehrere unterschiedliche logische Referenzen auf dieselbe GPU möglich. Ein mit `deviceInfo` erstelltes `VkDevice` benötigt beispielweise laut Listing 3.3 eine einzelne Queue mit bestimmten Eigenschaften. Dies wird über `pQueueCreateInfos` mitgeteilt.

Das Listing 3.4 verdeutlicht noch mal, wie mithilfe von `pNext` die Struktur erweitert werden kann. Theoretisch genügt es über `pEnabledFeatures` direkt `physicalDeviceFeatures` vom Typ `VkPhysicalDeviceFeatures` bekanntzugeben. Jedoch besitzt es eine feste Anzahl an Features. Über ein `VkPhysicalDeviceFeatures2` ist dieses erweiterbar. Im Beispiel wurden allerdings keine neuen Device-Features verwendet.

```
/* Erstellter VkDevice Handle wird in logicalDevice gespeichert */
vkCreateDevice(
    pje::context.physicalDevices[pje::context.choosenPhysicalDevice],
    &deviceInfo,
    nullptr,
    &pje::context.logicalDevice
);
```

Listing 3.5: Erstellen eines VKDevice

```
/* Device Funktion : Speichert Handle in queueForPrototyping */
vkGetDeviceQueue(
    pje::context.logicalDevice,
    pje::context.choosenQueueFamily,
    0,
    &pje::context.queueForPrototyping
);
```

Listing 3.6: Erhalten eines VkQueue Handles eines VkDevices

3.3 Shadermodule

Nach der Erstellung des Devices können nun *Device-related* Objekte erzeugt werden, die dem Device des Renderers (context.logicalDevice) zugehörig sind. Das aus [Softwarearchitektur der PJEngine](#) bekannte Context context besitzt unter anderem alle globalen zuweisbaren Handlevariablen für die entstehenden Vulkan Objekte. Zunächst wurden die Handle der Shadermodule `VkShaderModule` generiert, welche SPIR-V Code einbetten.

Wie SPIR-V aus GLSL Code kompiliert wurde, kann unter [Softwarekomplilierung mit CMake](#) nachgelesen werden. SPIR-V wird nach [SK17, S. 166] als „binary intermediate representation of program code“ verstanden. Dies hat den Vorteil, dass der Interpretationsraum für Treiber eingeschränkt wird und dennoch eine Plattformunabhängigkeit ermöglicht wird.

```
std::vector<char> shaderBasicVert = pje::readSpirvFile("assets/shaders
/basic.vert.spv");
std::vector<char> shaderBasicFrag = pje::readSpirvFile("assets/shaders
/basic.frag.spv");

/* Einbettung des SPIR-V Codes in VkShaderModule */
pje::createShaderModule(shaderBasicVert, pje::context.
    shaderModuleBasicVert);
pje::createShaderModule(shaderBasicFrag, pje::context.
    shaderModuleBasicFrag);
```

Listing 3.7: Generierung von Shadermodulen

```
VkShaderModuleCreateInfo shaderModuleInfo;
shaderModuleInfo.codeSize = spvCode.size();
shaderModuleInfo.pCode = (uint32_t*)spvCode.data();

vkCreateShaderModule(pje::context.logicalDevice, &shaderModuleInfo,
    nullptr, &shaderModule);
```

Listing 3.8: Ausschnitt aus `createShaderModule`

Die mit `readSpirvFile()` (s. Listing 3.7) eingelesenen SPIR-V Shader werden an die programminterne Funktion `createShaderModule()` mitsamt einer vom context bereitgestellten `VkShaderModule` Handlevariable übergeben. Wie auch alle anderen Handlevariablen im Kontextobjekt context sind diese mit `VK_NULL_HANDLE` instanziert, was einem ungültigen Handle entspricht [[Khr23](#), unter Appendix G: API Boilerplate].

Um mit `vkCreateShaderModule()` aus Listing 3.8 ein Modul zu erhalten, wird wie gewohnt ein Infoobjekt (hier `VkShaderModuleCreateInfo`) verlangt. Die Member der Info speichern Pointer auf den im Arbeitsspeicher liegenden SPIR-V Code. Der *Char Pointer* hinter `spvCode.data()` wird auf einen 32-Bit Pointer gecastet, da jedes *SPIR-V Token* immer einem 32-Bit Wort entspricht [SK17, S. 173].

```
pje::addShaderModuleToShaderStages(pje::context.shaderModuleBasicVert,
                                   VK_SHADER_STAGE_VERTEX_BIT);
pje::addShaderModuleToShaderStages(pje::context.shaderModuleBasicFrag,
                                   VK_SHADER_STAGE_FRAGMENT_BIT);

/* Ausschnitt aus addShaderModuleToShaderStages() */
VkPipelineShaderStageCreateInfo shaderStageCreateInfo;

shaderStageCreateInfo.module = newModule;
shaderStageCreateInfo.stage = stageType;
shaderStageCreateInfo.pName = shaderEntryPoint;

/* Fügt neue Shaderstage der Liste an Shaderstages hinzu */
pje::context.shaderStageInfos.push_back(shaderStageCreateInfo);
```

Listing 3.9: Deklaration von Shaderstages

Damit Vulkan die Module später einer *Grafikpipeline* einbinden kann, müssen diese Handles über mehrere `VkPipelineShaderStageCreateInfo` structs deklariert werden. Listing 3.9 zeigt, dass neben der Deklaration mittels Membervariable (`*.module`) auch die Stage (`*.stage`), in der das jeweilige Modul greifen soll und der *Entry Point* (`*.pName`) in den jeweiligen Shader benötigt wird.

3.4 Synchronisationsprimitive von Vulkan

Die Grafikpipeline, welche einzelne Schritte bis zum fertig gerenderten Bild beschreibt, wird auf der GPU ausgeführt. Nicht nur die CPU und GPU agieren unabhängig voneinander, sondern auch die einzelnen Befehle werden nach dem Vulkan Designkonzept asynchron und parallel auf mehreren Queues abgearbeitet [SK17, S. 367]. Jedoch müssen bestimmte Aufgaben erledigt sein bevor andere Teile starten dürfen, sodass Synchronisationspunkte erforderlich sind.

Im Projektumfang wurden dazu die von Vulkan gebotenen Synchronisationsprimitive `VkSemaphore`, `VkFence` und Pipeline Barriers angewandt. Deren Einsatz wird in den folgenden Sktionen deutlich. Das `VkEvent` findet keine Anwendung.

```

/* Zur Signalisierung eines fertig verarbeiteten VkCommandBuffer */
VkSemaphoreCreateInfo semaphoreInfo;
semaphoreInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
semaphoreInfo.pNext = nullptr;
semaphoreInfo.flags = 0;

vkCreateSemaphore(
    pje::context.logicalDevice, &semaphoreInfo,
    nullptr, &pje::context.semaphoreSwapchainImageReceived
);
vkCreateSemaphore(
    pje::context.logicalDevice, &semaphoreInfo,
    nullptr, &pje::context.semaphoreRenderingFinished
);

```

Listing 3.10: Erstellung der Semaphore

```

/* Primitiv zur GPU - CPU Synchronisation */
VkFenceCreateInfo fenceInfo;
fenceInfo.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
fenceInfo.pNext = nullptr;

// 0 == unsignaled ; 1 == signaled
fenceInfo.flags = 0;
vkCreateFence(
    pje::context.logicalDevice, &fenceInfo,
    nullptr, &pje::context.fenceSetupTasks
);

fenceInfo.flags = 1;
vkCreateFence(
    pje::context.logicalDevice, &fenceInfo,
    nullptr, &pje::context.fenceRenderFinished
);

```

Listing 3.11: Erstellung von Fences

Die Listings 3.10 und 3.11 weisen alle Semaphore und Fences auf, die vom Renderer in der aktuellen Version gebraucht werden. Während Fences zur Synchronisation von CPU und GPU genutzt werden, können Abhängigkeiten zwischen einzelnen *Batches von Commands* (dazu mehr unter [Command Buffer und Framebuffer](#)) mithilfe des `VkSemaphore` Primitives definiert werden [Mun+21].

Bei einem Batch spricht man von einer Menge an Commands, die sich innerhalb eines *Command Buffers* befinden. Mit der Beendigung eines Batches von Commands werden diese Primitive signalisiert. Erst mit einem signalisierten Fence bzw. Semaphore wird ein *Thread* auf der CPU fortgesetzt oder eine Queue mit der Verarbeitung eines anderen Batches beginnen.

Für die *Inter-Command-Synchronisation* auf einer Queue werden *Pipeline Barriers* eingesetzt, die auch außerhalb des *Batch Scopes* gültig sind. Die Commands eines Command Buffers starten in der übergebenen Reihenfolge, aber die Beendigung einzelner Commands ist davon unabhängig. Da Abhängigkeiten untereinander möglich sind, werden *Barriers* und *Subpass Dependencies* definiert.

Subpass Dependencies sind Barriers, die explizit Abhängigkeiten zwischen *Render Subpasses* (s. [Grafikpipeline und Renderpass](#)) ausdrücken [Mun+21].

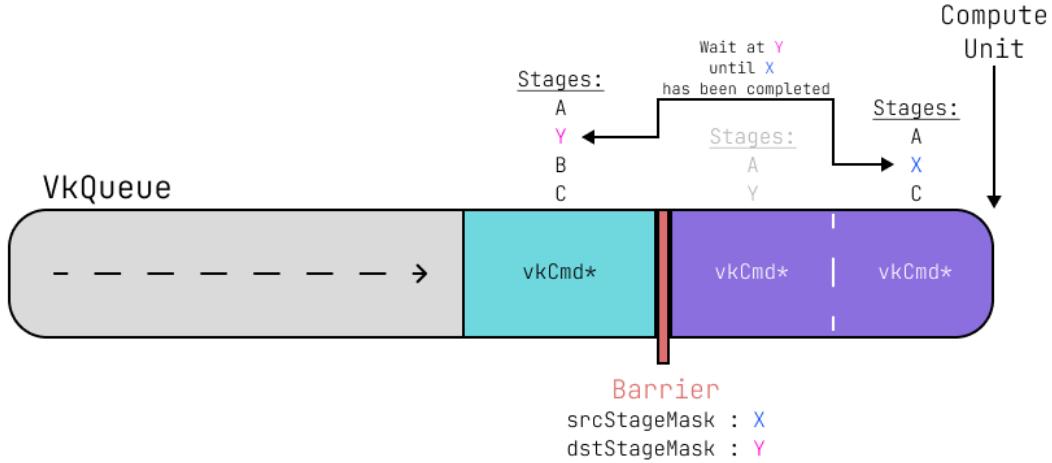


Abbildung 3.5: Visualisierung des Barrier-Primitives

Barriers dienen als Synchronisationselement bei Speicherzugriffen in den verschiedenen *Stages* der einzelnen Commands (`vkCmd*`) [SK17, S. 113]. Diese lassen sich in *Memory Barriers* und *Execution Barriers* unterscheiden, wobei Execution Barriers eine Untermenge von Memory Barriers darstellen.

Die verschiedenen Commands durchlaufen eine andere Kombination an Stages. Die Transfer-Stage `VK_PIPELINE_STAGE_TRANSFER_BIT` durchlaufen beispielsweise `vkCmdCopy*` Befehle [Khr23, unter 7.1.2 Pipeline Stages]. Eine Execution Barrier spezifiziert, welche Stage (`dstStageMask`) geblockt wird bis ein vorheriger Command eine bestimmte Stage (`srcStageMask`) beendet hat [Mun+21, unter Pipeline Barrier].

Die Speicherzugriffe in Stages verlaufen über Caches. Da VRAM relativ langsam ist und Daten im Cache Zwischenspeicher liegen bzw. liegen müssen, bedarf es eines *Cache Flushing* und *Cache Invalidation*. Das Flushing (`srcAccessMask`) bringt Daten vom Cache zum VRAM. Das Invalidating (`dstAccessMask`) eines Caches verlangt, dass neue Daten in den Cache geladen werden müssen [Mun+21, unter Memory Barriers]. Wann was geflusht sein muss, um in einer Stage, die unter `dstStageMask` vermerkt wurde, sichtbar und verfügbar zu sein, stellt die Memory Barrier als Bedingung.

Mittels *Bitwise Or* lassen sich in den vorher genannten Parametern mehrere Stages bzw. Accesses als Flag-Kombination definieren.

3.5 Swapchain

```

VkSwapchainCreateInfoKHR swapchainInfo;

// VkSurface, auf welchem Bilder präsentiert werden
swapchainInfo.surface = pje::context.surface;

// verfügbare Bilder in der Swapchain
swapchainInfo.minImageCount = pje::config::neededSurfaceImages;

// Daten liegen in einem VkFormat im Speicher
// und werden auf einen VkColorSpaceKHR gemappt
swapchainInfo.imageFormat = pje::config::outputFormat;
swapchainInfo.imageColorSpace = VK_COLOR_SPACE_SRGB_NONLINEAR_KHR;

if (pje::config::enableVSync)
    swapchainInfo.presentMode = VK_PRESENT_MODE_FIFO_KHR;
else
    swapchainInfo.presentMode = VK_PRESENT_MODE_IMMEDIATE_KHR;

vkCreateSwapchainKHR(
    pje::context.logicalDevice, &swapchainInfo,
    nullptr, &pje::context.swapchain
);

```

Listing 3.12: Ausschnitt der Erstellung einer Swapchain

Um auf einem `VkSurfaceKHR` (bekannt aus [Von der `VkApplicationInfo` bis zum logischen `VkDevice`](#)) einzelne Bilder zu präsentieren, werden spezielle `VkImages` benötigt, deren Daten für die Bildschirmdarstellung ausgelesen werden. `VkSwapchainKHR` fragt diese Bilder bei dem *Window System* des Hostsystems an [[SK17](#), S. 143]. Die Vulkan Anwendung kann verfügbare Bilder von der Swapchain anfragen, um in diese zu rendern und sie anschließend zurückzugeben, um sie von der „Presentation Engine“ [[Khr23](#), unter 34.5.3. Surface Presentation Mode Support] präsentieren zu lassen. Wie die PJEngine dies umsetzt, kann in [Aufnahme eines Renderbefehles und dessen Präsentation](#) nachgelesen werden.

Die Swapchain hält ein `VkPresentModeKHR` Objekt (`.presentMode`), welches die Synchronisationart mit dem Window System definiert und die Frequenz, mit welcher die Bilder dem Surface präsentiert werden, bestimmt [[SK17](#), S. 146]. Dazu gehören unter anderem:

- `VK_PRESENT_MODE_IMMEDIATE_KHR`
(zu präsentierende Bilder werden ohne Warten auf das Vertical Blanking dargestellt)
- `VK_PRESENT_MODE_MAILBOX_KHR`
(zu präsentierende Bilder werden als *pending* markiert und verworfen, falls vor dessen Darstellung nach dem Vertical Blank ein folgendes Bild präsentiert werden soll)

- **VK_PRESENT_MODE_FIFO_KHR**
(zu präsentierende Bilder werden aus einer internen Queue in FIFO Reihenfolge während den Vertical Blanks geholt und dargestellt)
- **VK_PRESENT_MODE_FIFO_RELAXED_KHR**
(zu präsentierende Bilder werden aus einer internen Queue auch außerhalb des Vertical Blanks geholt, falls gerade kein anderes Bild dargestellt wird)

Ein *Vertical Blank* meint die Zeit zwischen dem zuletzt abgeschlossenen dargestellten Frame und dem Präsentationsbeginn des nächsten Frames. Um ein *Screen Tearing* zu vermeiden, sollte das aktuell zu präsentierende Bild erst während eines Vertical Blanks getauscht werden. Vulkan bietet hierzu die Modi `VK_PRESENT_MODE_MAILBOX_KHR` und `VK_PRESENT_MODE_FIFO_KHR`, um diese Artefakte zu vermeiden.

Die PJEngine bietet über den boolschen Wert `pje::config::enableVSync` die Möglichkeit zwischen dem Performance Modus (`VK_PRESENT_MODE_IMMEDIATE_KHR`) und dem VSync Modus (`VK_PRESENT_MODE_FIFO_KHR`) zu wählen.

```
auto swapchainImages = vector<VkImage>(pje::context.
    numberofImagesInSwapchain);
vkGetSwapchainImagesKHR(
    pje::context.logicalDevice, pje::context.swapchain,
    &pje::context.numberofImagesInSwapchain, swapchainImages.data()
);

/* Ausschnitt aus dem Infoobjekt */
VkImageViewCreateInfo imageViewInfo;

imageViewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
imageViewInfo.format = pje::config::outputFormat;
imageViewInfo.components = VkComponentMapping{
    VK_COMPONENT_SWIZZLE_IDENTITY
};
imageViewInfo.subresourceRange = VkImageSubresourceRange{
    VK_IMAGE_ASPECT_COLOR_BIT, // Art der Daten im Bereich
    0,                        // baseMipLevel
    1,                        // levelCount
    0,                        // baseArrayLayer
    1                         // layerCount
};

/* Erstellt VkImageView fuer jedes VkImage in der Swapchain */
for (uint32_t i = 0; i < pje::context.numberofImagesInSwapchain; i++)
{
    imageViewInfo.image = swapchainImages[i];

    vkCreateImageView(
        pje::context.logicalDevice,
        &imageViewInfo,
        nullptr,
        &pje::context.swapchainImageViews[i]
    );
}
```

Listing 3.13: Erhalten von `VkImage` und `VkImageView` von einer Swapchain

Um die Handle der Swapchain Bilder zu erhalten, benötigt man die Funktionen `vkGetSwapchainImagesKHR()`. Den Zugriff auf die Bereiche eines `VkImage` erhält man über eine `VkImageView`.

3.5.1 Rendertargets

Rendertargets bezeichnen `VkImages`, auf welchen Resultate aus der Grafikpipeline gesichert werden. Neben den Bildern der Swapchain, welche die jeweiligen finalen Resultate eines Durchlaufes des *Renderpass* beinhalten, werden `VkImages` für das *MSAA* und den *Depth Buffer* benötigt. Mit dem *MSAA* `VkImage` werden für jeden Pixel/Fragment mehrere *Samples* gespeichert, um mit diesen einen Farbwert für das finale Resultat (*Resolve Image*) zu erhalten, welcher die Kanten weniger scharf wirken lässt. Mit den Tiefenwerten im Depth Buffer wird dafür gesorgt, dass sich nicht der zuletzt berechnete Farbwert automatisch im `VkImage` des *MSAA* bzw. *Resolve Image* befindet, sondern der Farbwert des Fragmentes, welcher am nächsten zur Kamera ist.

```
struct PJImage {
    VkImage           image      = VK_NULL_HANDLE;
    VkDeviceMemory   deviceMemory = VK_NULL_HANDLE;
    VkImageView       imageView   = VK_NULL_HANDLE;
    unsigned int      mipCount   = 0;
};
```

Listing 3.14: Datentyp PJImage mit default Werten

In `createDepthAndMSAA()` werden nacheinander `VkImage` und `VkImageView` für die globalen Variablen `pje::PJBuffer pje::rtMsaa` und `pje::PJBuffer pje::rtDepth` erstellt, die Handle zu diesen Vulkan Objekten besitzen.

Anders als bei der Swapchain reicht keine Anfrage von `VkImages`. Diese müssen nun selbst erstellt werden. Listing 3.14 zeigt, dass ein zusätzliches `VkDeviceMemory` benötigt wird. Nachdem mit den eigenen Funktionen `createVkImage()` und `allocateVkImageMemory()` beide Vulkan Objekte erstellt wurden, wird eine Verbindung mit `vkBindImageMemory()` geschaffen und anschließend die *Image View* erstellt.

```
VkMemoryRequirements memReq;
vkGetImageMemoryRequirements(pje::context.logicalDevice, image,
                            &memReq);

VkMemoryAllocateInfo memAllocInfo;
memAllocInfo.allocationSize = memReq.size;
memAllocInfo.memoryTypeIndex = getMemoryTypeIndex(memReq.
    memoryTypeBits, memoryFlags);

vkAllocateMemory(pje::context.logicalDevice, &memAllocInfo, nullptr,
                &memory);
```

Listing 3.15: Ausschnitt aus `allocateVkImageMemory`

```
VkPhysicalDeviceMemoryProperties memProps;

for (uint32_t i = 0; i < memProps.memoryTypeCount; i++) {
    if (((memoryTypeBits & (1 << i)) &&
        (memProps.memoryTypes[i].propertyFlags & flags) == flags) {
```

```

        return i;
    }
}
    
```

Listing 3.16: Ausschnitt aus getMemoryTypeIndex

Um in `allocateVkImageMemory` (Listing 3.15) über `vkAllocateMemory` Speicher für ein `VkImage` zu reservieren, muss ein geeigneter `VkMemoryType` mit `getMemoryTypeIndex()` ermittelt werden. Dazu wird eine Bitmaske `memReq.memoryTypeBits` herangezogen, dessen i -tes Bit auf 1 gesetzt ist, falls der jeweilige i -te `VkMemoryType` im Array `VkPhysicalDeviceMemoryProperties.memoryTypes` der GPU die Ressource unterstützt [Khr23, unter 12.8. Resource Memory Association]. Zudem sollen `VkMemoryPropertyFlags flags` vom `VkMemoryType` erfüllt werden (Listing 3.16), damit dieser alle erforderlich Eigenschaften erfüllt.

Es kann beispielsweise ein `VkMemoryType` mit Flag `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` mittels `vkMapMemory()` Daten vom Hostsystem erhalten [Khr23, unter 11.2.1. Device Memory Properties]. Dies wird noch mal in den Abschnitten [Modelldaten](#) und [Descriptorsets](#) aufgegriffen.

3.6 Grafikpipeline und Renderpass

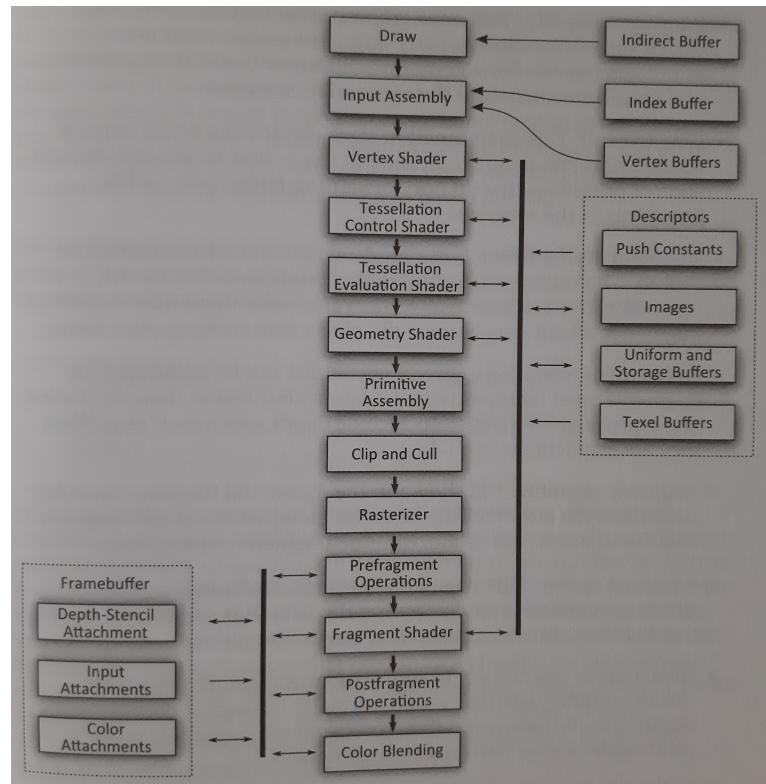


Abbildung 3.6: Grafik aus [SK17, S. 227]

Um mit den Daten, welche der GPU zur Verfügung stehen, einen einzelnen Frame zu rendern, müssen diverse Verarbeitungsschritte fließbandförmig durchgeführt werden. Die einzelnen *Stages* werden zu einer Grafikpipeline zusammengefasst. Wie bei einem Fließband werden je Stage Ressourcen (hier Daten) von außen reingegeben, aber auch die Resultate der vorherigen Stage verwendet. Abbildung 3.6 verdeutlicht in welcher Reihenfolge die einzelnen Teilaufgaben absolviert werden und in welchen Stages es zu einem Datenaustausch mit äußeren Daten kommt.

Das finale Resultat der Pipeline wird in den Framebuffer und damit in den angegebenen Rendertargets gespeichert. Die Berechnung des finalen Resultats findet im Fragment Shader und dessen folgenden „fixed-functions“ statt [SK17, S. 228]. Der Fragment Shader arbeite dabei einzeln mit den *rasterisierten Primitiven* aus dem Primitiv Assembly. Die Primitive beschreiben gruppierte Vertices aus vorherigen Stages. Rasterisierte Bereiche, die das aktuelle Primitiv nicht enthalten, werden nicht vom Fragment Shader verarbeitet.

Eine Pipeline besteht aus *programmable Stages* auch *fixed Functions*. Die programmable Stages beinhalten die Shadermodule, welche zuvor in **Shadermodule** erstellt wurden. Für die fixed Functions müssen der **VkGraphicsPipelineCreateInfo** Konfigurations-Infoobjekte zugewiesen werden.

```
VkGraphicsPipelineCreateInfo pipelineInfo;

// programmable states
pipelineInfo.stageCount = 2;
pipelineInfo.pStages = pje::context.shaderStageInfos.data();

// Beschreibung der zu bindenden Vertexdaten
pipelineInfo.pVertexInputState = &vertexInputInfo;

// einige fixed-functions
pipelineInfo.pViewportState = &viewportInfo;
pipelineInfo.pInputAssemblyState = &inputAssemblyInfo;
pipelineInfo.pRasterizationState = &rasterizationInfo;

// MSAA und Depth Test Konfiguration
pipelineInfo.pMultisampleState = &multisampleInfo;
pipelineInfo.pDepthStencilState = &depthStencilStateInfo;

// Deklaration von dynamic states
pipelineInfo.pDynamicState = &dynamicStateInfo;

// Deklaration der gebundenen Daten (Uniforms, Storage Buffer, etc.)
pipelineInfo.layout = pje::context.pipelineLayout;

// Renderpass - Zuordnung
pipelineInfo.renderPass = pje::context.renderPass;
pipelineInfo.subpass = 0;

vkCreateGraphicsPipelines(
    pje::context.logicalDevice,
    ...,
    &pipelineInfo,
    ...,
    &pje::context.pipeline
```

3 Vulkan Konzepte anhand der PJEngine

) ;

Listing 3.17: Ausschnitt aus der Erstellung einer Grafikpipeline

Zur Erstellung einer Grafikpipeline werden neben den Shadernmodulen, welche dem `.pStages` Parameter übergeben werden, auch Infoobjekte übergeben, welche das Sampling (`.pMultisampleState`) und einen möglichen Depth Test (`.pDepthStencilState`) in die Pipeline integrieren.

Bei den *dynamic States* handelt es sich um Eigenschaften einer Pipeline, welche auch nach der Erstellung veränderlich sein können. Die PJEngine nutzt dies, um den *Viewport* und die *Scissor* dynamisch festzulegen, was eine Veränderung der Fenstergröße mit gleichwertiger Auflösung zulässt ohne die Pipeline neu erstellen zu müssen.

Mittels `.layout` wird ein Pipelinelayout übergeben, welches die Layouts von Descriptorsets enthält, die in der Pipeline verwendet werden. Das Layout eines Descriptorsets deklariert die Daten, welche von außen hinzugegeben werden. Dies wird im Abschnitt [Descriptorsets](#) im Anwendungsfall noch mal verdeutlicht.

Um zu bestimmen, welcher *Subpass* eines *Renderpass* (`.renderPass`) die erstellte Pipeline verwendet, wird der entsprechende Index des Subpasses in `.subpass` angegeben.

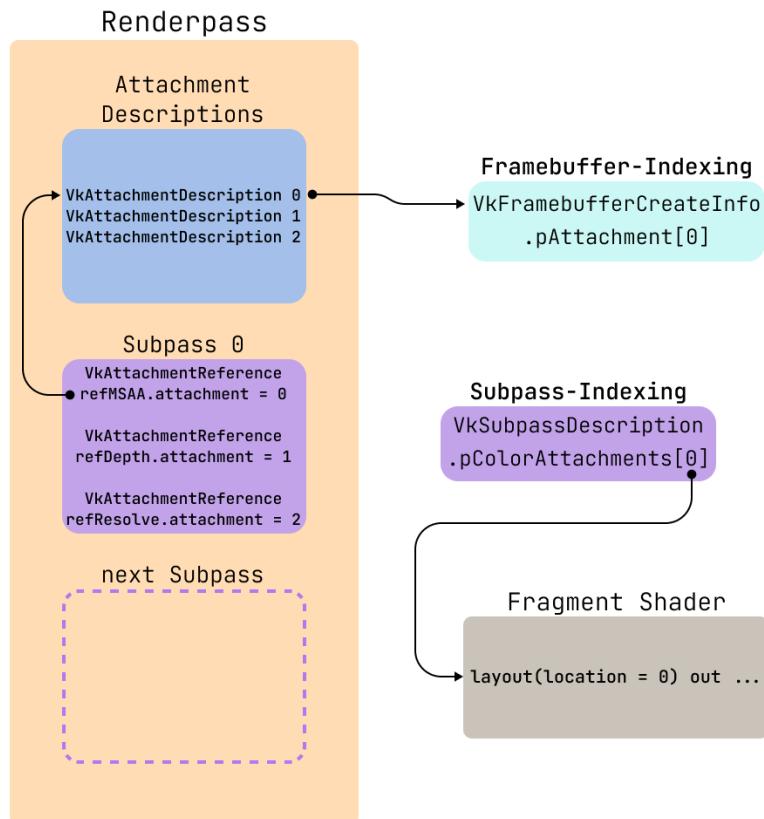


Abbildung 3.7: Aufbau des Renderpass des Renderers der PJEngine

Die einzelne Ausführung eines Renderbefehles umfasst die Komplexität eines einzelnen Renderpass. Dieser besteht aus einem oder mehreren Subpasses, die jeweils dem Durchlauf einer Grafikpipeline entsprechen. Um zu bestimmen, welcher Subpass eines Renderpass eine bestimmte Grafikpipeline verwendet, wird im `VkGraphicsPipelineCreateInfo.subpass` der jeweilige Index angegeben.

```
// Beispiel aus einer Attachment-Description
VkAttachmentDescription attachmentDescriptionDepth;
attachmentDescriptionDepth.format = pje::config::depthFormat;
attachmentDescriptionDepth.samples = pje::config::msaaFactor;

attachmentDescriptionDepth.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
attachmentDescriptionDepth.storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
attachmentDescriptionDepth.stencilLoadOp =
    VK_ATTACHMENT_LOAD_OP_DONT_CARE;
attachmentDescriptionDepth.stencilStoreOp =
    VK_ATTACHMENT_STORE_OP_DONT_CARE;

attachmentDescriptionDepth.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
attachmentDescriptionDepth.finalLayout =
    VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

// Attachment-Description-Array
array<VkAttachmentDescription, 3> attachmentDescriptions = {
    attachmentDescriptionMSAA, attachmentDescriptionDepth,
    attachmentDescriptionResolve
};
```

Listing 3.18: Ausschnitt aus der Erstellung von `VkAttachmentDescriptions`

Die einzelnen Bilder, welcher während eines Renderpasses als Input und/oder Output verwendet werden, werden mit `VkAttachmentDescriptionElementen` beschrieben [SK17, S. 231].

Der `VkFormat` Parameter (s. `.format` in Listing 3.18) muss mit dem Format des Bildes, welches als *Attachment* über den Framebuffer (genaueres unter [Command Buffer und Framebuffer](#)) genutzt wird, übereinstimmen. Über das Format lässt sich zudem bestimmen, ob eine *Stencil-Komponente* vorhanden ist. Ohne diese Komponente werden `.stencilLoadOp` und `.stencilStoreOp` ignoriert [Khr23, unter 8.2. Render Pass Creation].

Mit den *Load und Store Operationen* wird entschieden, was mit den Attachments zu Beginn und zum Ende eines Renderpass geschehen soll [SK17, S. 232]. Da sich alle Framebuffer der PJEngine dasselbe Depth und MSAA Rendertarget teilen, sollten zur Load Operation der Inhalt gecleared werden (`VK_ATTACHMENT_LOAD_OP_CLEAR`). Falls der Inhalt aus der Berechnung nach dem Durchlauf des Renderpass benötigt wird, wird die für die Store Operation `VK_ATTACHMENT_STORE_OP_STORE` gesetzt [SK17, S.232]. Dies wird im Beispiel für `attachmentDescriptionResolve` angewandt.

```
VkAttachmentReference attachmentRefMSAA;
attachmentRefMSAA.attachment = 0;
attachmentRefMSAA.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

VkAttachmentReference attachmentRefDepth;
attachmentRefDepth.attachment = 1;
```

```

attachmentRefDepth.layout =
    VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

VkAttachmentReference attachmentRefResolve;
attachmentRefResolve.attachment = 2;
attachmentRefResolve.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL
;

// Ausschnitt aus der Subpass-Description
VkSubpassDescription subpassDescription;
subpassDescription.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS
;
subpassDescription.colorAttachmentCount = 1;
subpassDescription.pColorAttachments = &attachmentRefMSAA;
subpassDescription.pDepthStencilAttachment = &attachmentRefDepth;
subpassDescription.pResolveAttachments = &attachmentRefResolve;

```

Listing 3.19: Subpass-Deklaration mit Referenzierung auf VkAttachmentDescriptions

Während der Renderpass eine Liste an `VkAttachmentDescription` zugewiesen bekommt, referenzieren die Subpasses des Renderpass diese Elemente über eine `VkAttachmentReference`, wie Abbildung 3.7 zeigt. Die `i-te` Referenz unter `.pColorAttachments` entspricht der einer möglichen Outputvariable mit `location = 0` [Khr23, unter 8.2. Render Pass Creation], wie Abbildung 3.7 ebenfalls darstellt.

```

VkSubpassDependency subpassDependencyDepth;

// entweder Index des Subpasses vom Renderpass
// oder EXTERNAL bei Subpasses anderer Renderpasses
subpassDependencyDepth.srcSubpass = VK_SUBPASS_EXTERNAL;
subpassDependencyDepth.dstSubpass = 0;

// Memory-Barrier Details
subpassDependencyDepth.srcStageMask =
    VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT |
    VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT;
subpassDependencyDepth.dstStageMask =
    VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT |
    VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT;
subpassDependencyDepth.srcAccessMask =
    VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;
subpassDependencyDepth.dstAccessMask =
    VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT;

// Dependency-Array
array<VkSubpassDependency, 2> subpassDependencies = {
    subpassDependencyDepth, subpassDependency
};

```

Listing 3.20: Ausschnitt aus der Erstellung von `VkSubpassDependencies`

Zur *Inter-Subpass-Synchronisation* stehen neben Barriers auch spezielle `VkSubpassDependency`Objekte zur Verfügung. In Listing 3.20 wird eine *Dependency* zwischen einem vorherigen Subpass eines anderen Renderpass (`VK_SUBPASS_EXTERNAL`) und dem Subpass des aktuellen Renderpass mit `Index = 0` aufgegriffen. Bei der Depen-

dencies im oberen Beispiel handelt es sich um eine Memory Barrier, da für die *Access-Mask*-Parameter andere Werte als VK_ACCESS_NONE gesetzt sind.

Mit den Parametern `.srcStageMask` und `.dstStageMask` wird die Execution Barrier festgelegt. Diese besagt im Beispiel, dass im `srcSubpass` zunächst die Stages zum Tiefentest zu durchlaufen sind bevor im `dstSubpass` die Stages zum Tiefentest starten dürfen. Die Stage VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT wird bereits während der *Load Operation* von `vkCmdBeginRenderPass()` durchlaufen [Khr23, unter 8.5. Render Pass Load Operation], da in den Attachments einer jeweiligen Ausführung des Renderpass das Depth Rendertarget vorhanden ist, welches sich alle Renderpasses in der PJEngine teilen.

Um *Race Conditions*¹ zwischen den Subpasses zu verhindert, wird eine Memory Barrier benötigt, auch wenn es zu keinem Caching kommt. Auch ohne konkrete Vorgaben zum Cachen von Ressourcen kann eine GPU Vulkan-konform sein. Um jede Möglichkeit abzudecken, wird eine Memory Barrier für das geteilte Depth Rendertarget gesetzt.

Erst mit einem durchgeführten *Flush* nach dem `.srcAccessMask`-Speicherzugriff, kann das Depth Rendertarget mit angewandten *Clear Value* für den Speicherzugriff von `.dstAccessMask` in der `.dstStageMask` gelesen werden (s. VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT). Falls dies nicht so bestimmt wurde, kann das scheinbar geclearte Depth Rendertarget vor dem Lesen alte Werte eines vorherigen Tiefentests enthalten.

Abschließend müssen sowohl die Dependencies und Subpasses als auch die einzelnen Attachment-Descriptions einem `VkRenderPassCreateInfo` zugewiesen werden, um einen Renderpass zu erstellen, welcher von einer Grafikpipeline im Parameter `VkGraphicsPipelineCreateInfo.renderPass` aufgegriffen wird.

```

VkRenderPassCreateInfo renderPassInfo;

// Liste der Attachment-Descriptions
renderPassInfo.attachmentCount = 3;
renderPassInfo.pAttachments = attachmentDescriptions.data();

// Liste der Subpasses in Ausfuehrungsreihenfolge
renderPassInfo.subpassCount = 1;
renderPassInfo.pSubpasses = &subpassDescription;

// Liste der Subpass-Dependencies
renderPassInfo.dependencyCount = 2;
renderPassInfo.pDependencies = subpassDependencies.data();

vkCreateRenderPass(pje::context.logicalDevice, &renderPassInfo, ...);

```

Listing 3.21: Ausschnitt aus der Erstellung eines Renderpass

3.7 Command Buffer und Framebuffer

¹Softwareanomalie zwischen parallel ablaufenden Einheiten von Aufgaben, welche sich durch geteilte Ressourcen gegenseitig beeinflussen können

3 Vulkan Konzepte anhand der PJEngine

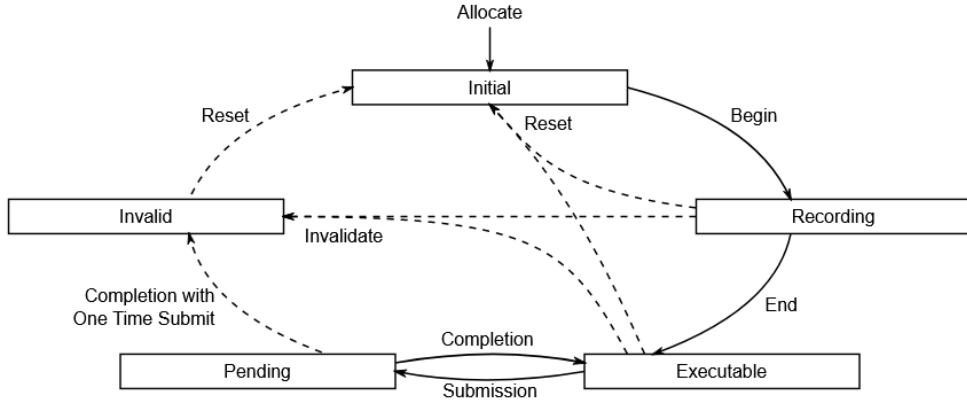


Abbildung 3.8: Lebenszyklus von Command Buffer nach [Khr23, unter 6.1. Command Buffer Lifecycle]

Um die einzelnen (`vkCmd*`()) Befehle auf eine `VkQueue` zu legen und somit von einer GPU verarbeiten zu lassen, werden Command Buffer benötigt. Diese Buffer werden mit einer Menge an Befehlen *recordet*, welche zusammen als *Batch* verstanden werden. Wie man einen Command Buffer recordet, zeigt Listing 3.28 beispielhaft.

Um einen Command Buffer zu erhalten, muss man diesen von einem `VkCommandPool` allokalieren. Dieser Pool bestimmt unter anderem, mit welcher Queue Family dessen Command Buffer arbeitet (`VkCommandPoolCreateInfo.queueFamilyIndex`).

Abbildung 3.8 zeigt die einzelnen States, in welchen sich ein Command Buffer befinden kann. Wenn ein Command Buffer über `vkQueueSubmit()` einer Queue übergeben wird, befindet sich dieser im *Pending State*. Erst mit der Beendigung aller recordeten Befehle wandert dieser in einen State, welcher für das nächste Recording in den *Initial State* zurücksetzbar ist. Dies kann mit `vkResetCommandBuffer()` oder indirekt über `vkBeginCommandBuffer()` geschehen.

```

VkFramebufferCreateInfo framebufferInfo;
framebufferInfo.renderPass = pje::context.renderPass;
framebufferInfo.attachmentCount = 3;

for (uint32_t i = 0; i < pje::context.number0fImagesInSwapchain; i++)
{
    array<VkImageView, 3> rendertargets{
        pje::rtMsaa.imageView,
        pje::rtDepth.imageView,
        pje::context.swapchainImageViews[i]
    };

    // VkAttachmentDescription des Renderpass beschreibt Rendertargets
    // des Framebuffers ; Reihenfolge muss eingehalten werden
    framebufferInfo.pAttachments = rendertargets.data();

    vkCreateFramebuffer(
        pje::context.logicalDevice,
        &framebufferInfo,
        nullptr,
        &pje::context.swapchainFramebuffers[i]
    );
}
  
```

```
) ;
}
```

Listing 3.22: Ausschnitt aus der Framebuffer-Erstellung für den Renderpass

Der Framebuffer dient als *Kommunikationsbuffer*, um festzulegen, welche Rendertargets (in diesem Kontext auch Attachments genannt) von einem Renderpass verwendet werden. Die Attachments werden über die `VkAttachmentDescription(s)` des jeweilige Renderpass vorher deklariert. Es ist zu beachten, dass die Reihenfolge der Attachments im Framebuffer der Reihenfolge der *Attachmentdescription* übereinstimmt.

Listing 3.22 zeigt, dass die PJEngine jeden Swapchain-Image einen Framebuffer erstellt, welche sich alle dasselbe Rendertarget für das MSAA und den Tiefentest teilen.

3.8 Shaderressourcen

```
* #version 450

/* ### IN ###
layout(location = 0) in vec3 pos;
layout(location = 1) in vec3 color;
layout(location = 2) in vec3 normal;
layout(location = 3) in vec2 texCoord;
layout(location = 4) in uvec2 boneRange;

layout(set = 0, binding = 0, std140) uniform UNIFORMS {
    mat4 mvp;
    mat4 modelMatrix;
    mat4 viewMatrix;
    mat4 projectionMatrix;
} uniforms;

/* ### OUT ###
layout(location = 0) out VOut {
    vec3 color;
    vec3 normal;
    vec2 texCoord;
} vOut;

/* ### ENTRY POINT ###
void main() {
    gl_Position = uniforms.mvp * vec4(pos.xyz, 1.0);
    vOut.color = color;
    vOut.normal = (uniforms.viewMatrix * transpose(inverse(uniforms.
        modelMatrix)) * vec4(normal, 0.0f)).xyz;
    vOut.texCoord = texCoord;
}
```

Listing 3.23: Relevanter Teil des Vertex Shaders vom Renderer

```
* #version 450

/* ### IN ###
layout(set = 0, binding = 3) uniform sampler2D texSampler;
```

```

layout(location = 0) in vIn {
    vec3 fragColor;
    vec3 fragNormal;
    vec2 fragTexCoord;
} vIn;

/* ### OUT ###
layout(location = 0) out vec4 color;

/* ### ENTRY POINT ###
void main() {
    // hier : VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT

    color = texture(texSampler, vIn.fragTexCoord);

    if (color.a == 0.0f) {
        discard;
    }

    vec3 n = normalize(vIn.fragNormal);
    float cosTheta = max(n.z, 0.0f);
    color = vec4(color.rgb * cosTheta, color.a);

    // hier : VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
}

// hier : VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT

```

Listing 3.24: Relevanter Teil des Fragment Shaders vom Renderer

In diesem Abschnitt wird verdeutlicht, wie die angesprochenen Shaderressourcen, die in Listing 3.23 und 3.24 verwendet werden, in dafür geeignete Vulkan Objekten gelagert werden. Listing 3.28 aus Abschnitt [Aufnahme eines Renderbefehles und dessen Präsentation](#) demonstriert, wie die Daten für eine ausgewählte Grafikpipeline anschließend gebunden werden.

Die PJEngine nutzt grundlegend die Vulkan Objekte `VkBuffer` und `VkImage` als Schnittstelle, um Vulkan Daten bereitzustellen oder von Vulkan in Form von Rendertargets zu erhalten. Da beide Handle noch zusätzliche Vulkan Objekte wie das `VkDeviceMemory` benötigten, wurden *structs* definiert, um notwendige Objekte zu *verkapseln*. Diese structs heißen `PJImage` und `PJBuffer`. Das `PJImage` ist bereits aus Listing 3.14 bekannt. Wie ein `PJImage` bzw. `PJBuffer` befüllt wird, wird in Unterabschnitt [Modelldaten](#) aufgeschlüsselt.

Anders als die Vertex- und Indexdaten, welche über eine `VkPipelineVertexInputStateCreateInfo` für eine Grafikpipeline deklariert werden, werden die übrigen Daten über ein *Descriptorlayout* der Grafikpipeline bekannt gegeben. Die Deklaration beinhaltet noch nicht die Bindung der Daten. Das Binden erfolgt über einen Befehl, der mithilfe eines *Command Buffers* auf eine `VkQueue` gelangt.

Unterabschnitt [Descriptorsets](#) zeigt, wie man ein *Descriptorset* für ein bestehendes *Descriptorlayout* erstellt.

3.8.1 Modelldaten

```
struct PJBuffer {
    VkBuffer           buffer      = VK_NULL_HANDLE;
    VkDeviceMemory     deviceMemory = VK_NULL_HANDLE;
    VkDeviceSize        size        = 0;
    VkMemoryPropertyFlags flags      = 0;
};
```

Listing 3.25: Datentyp PJBuffer mit default Werten

Bezüglich der Inputdaten wurden *PJBuffers* für die Vertex-, Index- und Uniformdaten und ein *PJImage* für die Albedotextur erstellt. Die globalen Variablen werden mit den default Werten aus Listing 3.25 und 3.14 instanziert. Um einen tatsächlichen Handle mit zugehörigen Speicher (s. *VkDeviceMemory*) zu erhalten, werden die Funktionen `sendPJModelToVRAM()`, `uploadTextureToVRAM()` und `createUniformBuffer()` für die bestehenden Variablen aufgerufen.

```
/* Allokation eines VkBuffer fuer Vertex- und Indexdaten */
sendPJModelToVRAM(
    pje::vertexBuffer, pje::indexBuffer, pje::loadedModels[pje::config::selectedPJModel]
);

/* Allokation eines PJImage fuer die unkomprimierte Albedotextur */
uploadTextureToVRAM(
    pje::texAlbedo, pje::loadedModels[pje::config::selectedPJModel], 0,
    true
);

/* Allokation eines kohärenten VkBuffer fuer die anzuwendenden
   Matrizen */
createUniformBuffer(
    pje::uniformsBuffer, sizeof(pje::Uniforms), "uniformsBuffer"
);
```

Listing 3.26: Bereitstellung der Daten in Vulkan Objekte

Sowohl `sendPJModelToVRAM()` als auch `uploadTextureToVRAM()` erstellen für den eingebetteten *VkBuffer* oder *VkImage* eines jeweiligen *PJBuffer* oder *PJImage* einen validen Handle. Das dazugehörige *VkMemory*, welches die eigentlichen Rohdaten halten wird, wird mit einem `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT`-Property-Flag instanziert, um den effizientesten „device access“ [Khr23, unter 11.2. Device Memory] zu ermöglichen. Mit der Angabe des Flags wird *Memory* allokiert, der die genannten Flags erfüllt (s. Listing 3.15). Bei Speicher mit diesem alleinigen Flag handelt es sich nicht direkt um Speicher, welcher von der CPU direkt zugreifbar ist.

Um die Daten von der CPU auf die GPU zu verschieben, wird ein *Staging Buffer* (ein zusätzlicher *PJBuffer*) benötigt, der die Flags `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` und `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT` erfüllt. Über den Staging Buffer kann Speicher von der CPU auf die GPU gemappt werden und anschließend über einem Command Buffer Befehl in den eigentlich dafür vorgesehenen lokalen Speicher mit `vkCmdCopyBuffer()` bzw. `vkCmdCopyBufferToImage()` kopiert werden.

Mit `createUniformBuffer()` wird ein *Uniform*¹ erstellt, welcher ebenfalls wie der Staging Buffer kohärenten Speicher besitzt. Dies wird benötigt, um für jeden neuen Frame die aktuellen Matrizen für die Position des zu rendernden Objektes anzupassen.

Zusätzlich zum erstellten `VkImage` wird ein universeller `VkSampler` erstellt, um die Bildaten im Shader lesen zu können. Der Sampler wird als Shaderressourcen über ein Descriptorset einer Grafikpipeline zur Verfügung gestellt.

3.8.2 Descriptorsets

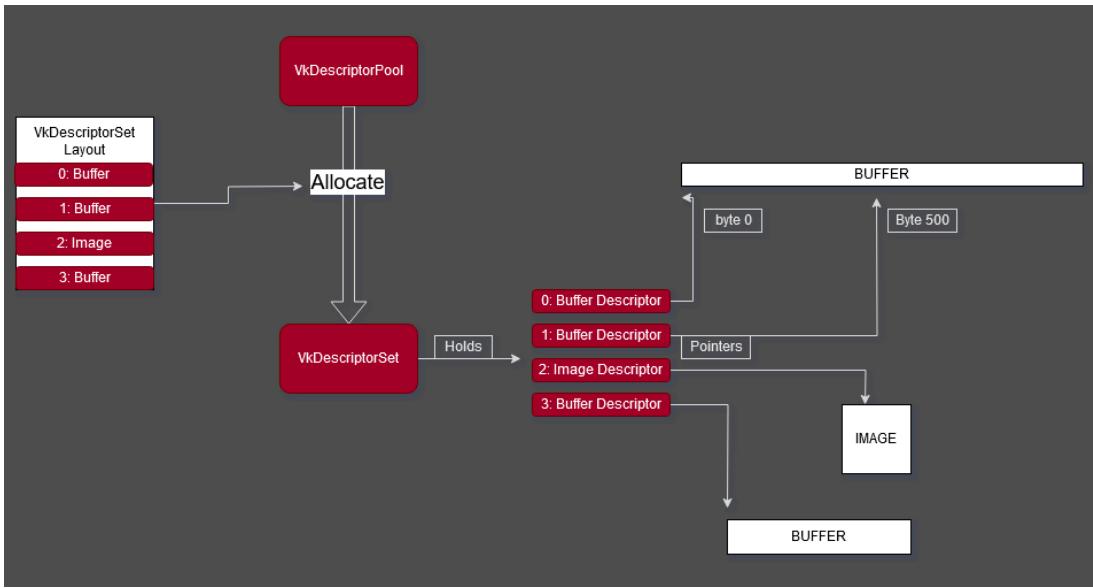


Abbildung 3.9: Beispiel eines Descriptorsets nach [Bla20, unter 4. Descriptor Sets]

Neben den eigentlichen Vertexdaten benötigen die Shaders der PJEngine weitere Ressourcen, welche über ein Descriptorset angesprochen werden. `layout(set = 0, binding = 3)` aus Listing 3.24 referenziert beispielsweise die 4. Ressource im 0. Set, welche für die Pipeline gebunden wurde.

Ein Descriptorset basiert immer auf einem `VkDescriptorSetLayout`, welches einer Pipeline vor dessen Erstellung mitgegeben wird, sofern diese ein Descriptorset mit dem bestimmten Layout benötigt. Hierzu wurde die Funktion `createDescriptorSetLayout()` aufgerufen, um `pje::context.descriptorSetLayout` das Layout für die Shaderressourcen zu bestimmen.

Um ein layout-konformes Descriptorset allozieren zu können, muss ein `VkDescriptorPool` vorhanden sein, welcher für dessen einzelne Sets mindestens den Umfang des Layouts genügen muss. Zudem muss deklariert sein, wie viele Sets von diesem Pool maximal allozierbar sind. Ein einzelnes Binding eines Layouts kann auch mehrere Element (*Descriptoren*) enthalten, die dann als Array im Shadercode angesprochen werden (`VkDescriptorSetLayoutBinding.descriptorCount`).

¹Storage Buffer, welcher in einem Pipelinedurchlauf unveränderlich ist

```

/* Erstellung eines Descriptorsets */
VkDescriptorSetAllocateInfo allocateInfo;
allocateInfo.descriptorPool = pje::context.descriptorPool;
allocateInfo.descriptorCount = 1;
allocateInfo.pSetLayouts = &pje::context.descriptorsetLayout;

vkAllocateDescriptorSets(
    pje::context.logicalDevice, &allocateInfo, &aDescriptorSet
);

/* aus : linkDescriptorSetToBuffer() */
VkDescriptorBufferInfo bufferInfo;

// eigentliche Daten
bufferInfo.buffer = pjBuffer.buffer;
bufferInfo.range = VK_WHOLE_SIZE;

VkWriteDescriptorSet updateBuffer;
updateBuffer.dstSet = aDescriptorSet;
updateBuffer.pBufferInfo = &bufferInfo;

// welches Binding und wie viele Elemente davon
updateBuffer.dstBinding = dstBinding;
updateBuffer.dstArrayElement = 0;
updateBuffer.descriptorCount = 1;

vkUpdateDescriptorSets(
    pje::context.logicalDevice, 1, &updateBuffer, 0, nullptr
);

/* aus : linkDescriptorSetToImage() */
VkDescriptorImageInfo imageInfo;

// eigentliche Daten ueber VkImageView und VkSampler
imageInfo.sampler = sampler;
imageInfo.imageView = pjImage.imageView;
imageInfo.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;

VkWriteDescriptorSet updateImage;
updateImage.dstSet = aDescriptorSet;
updateImage.pImageInfo = &imageInfo;

updateImage.dstBinding = dstBinding;
updateImage.dstArrayElement = 0;
updateImage.descriptorCount = 1;

vkUpdateDescriptorSets(
    pje::context.logicalDevice, 1, &updateImage, 0, nullptr
);

```

Listing 3.27: Erstellung und Linken von Ressourcen für ein Descriptorset

Nach der erfolgreichen Erstellung eines `VkDescriptorSet` können Vulkan Objekte, die bereits Daten für die Shader enthalten, mit diesem gelinkt werden. Dies geschieht über ein Binding des jeweiligen Sets. Durch die Verlinkung sind die Ressourcen im Shader zugreifbar.

3.9 Aufnahme eines Renderbefehles und dessen Präsentation

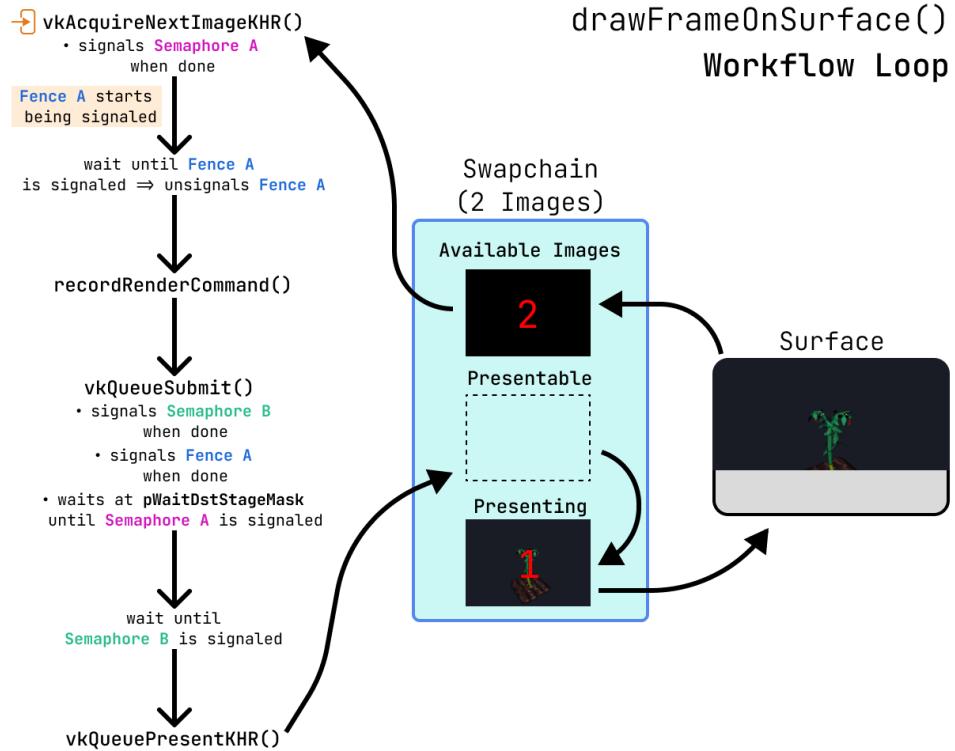


Abbildung 3.10: Arbeitablauf mit der Swapchain

Nachdem die Anwendung sämtliche Setup-Funktionen durchlaufen hat, wird `loop-Visualization()` ausgeführt, welche bis zur Schließung des GLFW Fensters eine `while` Schleife ausführt, bei der mit `glfwPollEvents()` *Events* und *Callbacks* verarbeitet werden, die bis zu diesem Zeitpunkt in die *Event Queue* eingegangen sind [Org22b, unter `glfwPollEvents`]. Nach der Abfrage der Events werden noch die Funktionen `updateMatrices()` zur Aktualisierung der Matrizen für das darzustellende Objekt und `drawFrameOnSurface()` für die Aufnahme und Ausführung der Renderanweisung und der Darstellung des Renderresultats aufgerufen.

Die obere Abbildung 3.10 visualisiert im Groben die Funktion `drawFrameOnSurface()`. Zunächst wird mit `vkAcquireNextImageKHR()` der *Image-Index* des nächsten verfügbaren Swapchain-Rendertargets geholt. Der erwartete `timeout` Parameter wurde auf 0 gesetzt, um ein „nonblocking behavior“ der Vulkan Anwendung zu erzielen [SK17, S. 151]. Somit können folgende Aufgaben bis zum Zeitpunkt, an dem das Render-target benötigt wird, bereits bearbeitet werden. In der Abbildung ist hierfür *Semaphore A* zuständig. Erst mit dessen Signalisierung wird mit der Pipeline-Stage begonnen, die im Parameter `pWaitDstStageMask` von `vkQueueSubmit()` gesetzt wurde. Mit `vkQueueSubmit()` werden über ein `VkSubmitInfo` ein oder mehrere Command Buffer an eine `VkQueue` geschickt, welche die aufgezeichneten Befehle auf dessen Queue legt, um diese von Compute Units verarbeiten zu lassen (s. Abbildung

3 Vulkan Konzepte anhand der PJEngine

3.5). Da die Funktion direkt nach dem *Scheduling* zur Vulkan Anwendung zurückkehrt [SK17, S. 108], wird mit *Semaphore B* mit der Ausführung von `vkQueuePresentKHR()` gewartet und mit *Fence A* mit der Aufzeichnung des nächsten Renderbefehles. Mit `vkQueuePresentKHR()` werden Rendertargets einer Swapchain über den Image-Index zur Presentation mit der Presentation Engine freigegeben.

```
/* recordRenderCommand(pje::PJModel& renderable, uint32 imageIndex) */

VkCommandBufferBeginInfo commandBufferBeginInfo;
commandBufferBeginInfo.flags =
    VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;

VkRenderPassBeginInfo renderPassBeginInfo;
renderPassBeginInfo.renderPass = pje::context.renderPass;
renderPassBeginInfo.clearValueCount = 3;
renderPassBeginInfo.pClearValues = clearValues.data();
renderPassBeginInfo.framebuffer = pje::context.swapchainFramebuffers[
    imageIndex];

/* START RECORDING */
vkBeginCommandBuffer(pje::context.commandBuffers[imageIndex],
    &commandBufferBeginInfo);
vkCmdBeginRenderPass(pje::context.commandBuffers[imageIndex],
    &renderPassBeginInfo, ...);

// welche Pipeline soll genutzt werden
vkCmdBindPipeline(pje::context.commandBuffers[imageIndex], ...,
    pje::context.pipeline);

// Setzen der VkDynamicState(s)
vkCmdSetViewport(pje::context.commandBuffers[imageIndex], ...);
vkCmdSetScissor(pje::context.commandBuffers[imageIndex], ...);

// welches Descriptorset mit zugeteilten Buffern soll verwendet werden
// das Pipelinelayout muss das Descriptorlayout unterstuetzen
vkCmdBindDescriptorSets(
    pje::context.commandBuffers[imageIndex],
    ...,
    pje::context.pipelineLayout,
    ...,
    &pje::context.descriptorSet,
    ...
);

vkCmdBindVertexBuffers(pje::context.commandBuffers[imageIndex],
    0, 1, &pje::vertexBuffer.buffer, offsets.data());
vkCmdBindIndexBuffer(pje::context.commandBuffers[imageIndex],
    pje::indexBuffer.buffer, 0, VK_INDEX_TYPE_UINT32);

for (const auto& mesh : renderable.m_meshes)
    vkCmdDrawIndexed(...);

/* END RECORDING */
vkCmdEndRenderPass(pje::context.commandBuffers[imageIndex]);
```

```
vkEndCommandBuffer(pje::context.commandBuffers[imageIndex]);
```

Listing 3.28: Ausschnitt aus der Aufzeichnung eines Renderbefehles

Das obere Listing 3.28 beschreibt die Arbeit, welche auf eine `VkQueue` gelegt wird, um einen einzelnen Frame zu rendern. Der Command Buffer, welcher bereits aus [Command Buffer und Framebuffer](#) bekannt sein sollte, wird zwischen einem `vkBeginCommandBuffer()` und `vkEndCommandBuffer()` recordet. Da die Command Buffers hinter `context.commandBuffers` von einem Pool stammen, welcher das Flag `VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT` besitzt, können die Buffer indirekt über `vkBeginCommandBuffer()` in den *initial state* zurückgesetzt werden [[Khr23](#), unter 6.2 Command Pools].

Das `VkRenderPassBeginInfo` Infoobjekt besitzt den Parameter `pClearValues`, um für die Rendertargets hinter dem verwendeten Framebuffer die *Clear Values* zu definieren. Die Reihenfolge muss hierbei der Reihung der Rendertargets im Framebuffer entsprechen. Je nachdem welcher Wert in `VkAttachmentDescription.loadOp` für das jeweilige Rendertarget gesetzt wurde, wird während `vkCmdBeginRenderPass()` der Clear Value für die einzelnen Pixel angewandt.

Nachdem eine Pipeline mit `vkCmdBindPipeline()` ausgewählt wurde, müssen noch die genutzten Ressourcen, die bei der Pipelineerstellung deklariert wurden (s. [Grafikpipeline und Renderpass](#)), gebunden werden. Mit `vkCmdBindDescriptorSets` wird ein Descriptorset gesetzt, dessen Layout im Pipelinelayout übergeben wurde. `vkCmdBindVertexBuffers()` bindet die eigentlichen Vertexdaten, die einer `VkVertexInputBindingDescription` der Pipeline (im Beispiel der 0. Description) entsprechen sollten.

Der eigentliche Renderbefehl wird mit einem `vkCmdDraw` Befehl eingeleitet. Die Darstellung des Renderresultats kann anschließend mit einem `vkQueuePresentKHR()`, wie in Abbildung 3.10, präsentiert werden.

3.10 Cleanup von Vulkan Ressourcen

Wenn das Programmende erreicht wurde, sollte eine manuelle Freigabe der Ressourcen, welche von Vulkan Objekten belegt wurden, stattfinden. Um diese Objekte zu entfernen, werden die zugehörigen Handle an von Vulkan bereitgestellte Funktionen (`vkDestroy*` und `vkFreeMemory()`) übergeben. In der PJEngine sind diese Funktionen in `stopVulkan()`, `resetRealtimeRendering()` und `cleanupRealtimeRendering()` implementiert. Wann die genannten Funktionen durchlaufen werden, kann aus Abbildung 2.2 entnommen werden. Daraus lässt sich folgern, dass einige Ressourcen auch beim *Resizen* des GLFW Fensters freigegeben werden. Hierbei handelt es sich um die Swapchain und sämtliche Rendertargets mitsamt deren Speicher, deren Größenausmaß der vorherigen Fenstergröße entspricht.

Generell gilt es als *Good Practice*, dass jegliche Arbeit auf dem entsprechenden `VkDevice` terminiert und alle Threads aufseiten der CPU, die mit zu löschen Vulkan Objekten arbeiten, diese Arbeit beenden. Zudem sollten die Vulkan Objekte in umgekehrter Reihenfolge ihrer Erstellung entfernt werden [[SK17](#), S. 31]. Um sicherzugehen, dass zu diesem Zeitpunkt keine Arbeit für die Vulkan Anwendung auf dem Device ausgeführt wird, kann `vkDeviceWaitIdle()` davor ausgeführt werden.

4 Weitere PJEngine-relevante Bibliotheken

Damit die PJEngine Resultate von Rendervorgängen von Vulkan auch auf einem Ziel-system anzeigen kann, benötigt die Vulkan API neben Ressourcen wie 3D-Objekte auch ein zur Verfügung gestelltes `VkSurfaceKHR`, welches der aktuellen Plattform (z.B. Microsoft Windows) entspricht. Zur Erfüllung dieser Anforderungen wurden weitere Bibliotheken dem Projekt hinzugefügt.

Während *GLM*, *Assimp* und *stb image* für die Bereitstellung der Ressourcen verwendet wurden, wird das Surface mithilfe von *GLFW* erzeugt.

4.1 GLFW : Cross-Platform Fenstergenerierung

Bevor mit `glfwCreateWindowSurface()` ein Surface erstellbar ist, wird ein `GLFWwindow` Objekt gebraucht. Dieses wird mit `glfwCreateWindow()` in der Funktion `pje::startGlfw3()` erstellt.

Zusätzlich zur Fenstererstellung wird ein *Callback* für die nachträgliche Änderung der Fenstergröße zur Laufzeit mit `glfwSetWindowSizeCallback()` festgelegt. Die Abfrage von Events, welche Callbacks auslösen, erfolgt während `glfwPollEvents()` und wird verlangt, um das *Window System* des Hostsystems darüber zu informieren, dass die Anwendung noch lebendig ist.

Weitere Informationen zur Bibliothek lassen sich unter [Org22a] und [Org22b] finden.

4.2 GLM : Bibliothek für Vektormathematik

GLM ist eine Bibliothek für das mathematische Teilgebiet der linearen Algebra und bietet erforderliche Matrizen, welche in der Computergrafik eingesetzt werden, um Objekte korrekt aus einer Perspektive im Raum zu zeichnen. *GLM* ([Cre]) liefert Funktionen, um Matrizen in einem Datenformat zu erhalten, welche in dieser Form von *GLSL* Shadercode verwendet wird. Die erstellten Matrizen werden über die globale Variable `pje::uniforms` auf den `PJBuffer pje::uniformsBuffer` als Ressource dem Shader zur Verfügung gestellt.

4.3 Assimp : Der universelle Modelloader

```

class PJModel {
public:
    std::vector<PJMesh> m_meshes;
    std::vector<std::vector<unsigned char>> m_uncompressedTextures;
    std::vector<TextureInfo> m_textureInfos;
    std::string m_modelPath;
    bool m_centered;

    void prepareTexture(
        const aiTexture* compressedRaw,
        size_t dstIndex,
        uint8_t desiredChannels = STBI_rgb_alpha,
        bool newEntry = true
    );
};

```

Listing 4.1: PJModel Klasse

Mit *Assimp* können unterschiedliche Dateiformate für 3D-Modelle in den Arbeitsspeicher geladen werden. Die PJEngine nutzt diese Schnittstelle, um Instanzen der PJModel-Klasse mit Daten für die Vulkan API zu befüllen.

Die Programmlogik von Assimp wurde in eine Klasse namens **Modelloader** gebündelt. Im *Constructor* der Klasse werden alle Dateien im *assets*-Ordner auf *FBX* bzw. *OBJ* Dateien überprüft. Die gefundenen FBX und OBJ Dateien werden mit der Methode *loadModel()* geladen, formatiert und der Instanzvariable *m_models* zugeordnet.

Zur Formatierung gehört unter anderem die Umwandlung der *Transformationsmatrizen* in *Column Major*¹, damit die Daten in der richtigen Reihenfolge bereitliegen, um von GLM für weitere Operationen verwendet zu werden. Assimp arbeitet selbst mit *Row Major* [ass, unter 2.5. Data Structures].

4.3.1 stb image : Der universelle Textureloader

Die über Assimp mit Modeldaten befüllten PJModel-Objekte besitzen eine *Instanzvariable* für die unkomprimierten Texturen. Da Assimp keine Garantie für Texturen in einem unkomprimierten Format gewährt und die eingebettete Textur im FBX-Testmodell eine komprimierte PNG-Datei ist, werden die geladenen Texturen mit der *stb image* Bibliothek ([Bar]) im Arbeitsspeicher dekomprimiert. Die PJEngine implementiert die Funktionen der Bibliothek innerhalb der Methode *PJModel::prepareTexture()*.

¹Daten liegen Spaltenweise im Speicher

5 Resultate

Die vorliegenden Resultate wurden auf Windows 10 mit einer NVIDIA GeForce RTX 2060 GPU und einer Intel Core i7-9750H CPU erzielt. Die Anwendung *demo.exe* wurde mit dem Ninja Build System in Visual Studio kompiliert. Die Berechnung auf der CPU geschieht auf einem Kern. Zusätzliche *Kindprozesse* und Threads wurden im Programmablauf nicht angelegt. Die Anwendung wurde auf einem 144 Hertz Monitor ausgeführt.

Mit dem Renderer wurden folgende Features mit der Vulkan API umgesetzt:

- Laden von FBX und OBJ Dateien, welche im *assets* Ordner hinterlegt wurden
- Entpacken und Dekomprimierung von in FBX Dateien eingebetteten Texturen
- Echtzeitdarstellung eines geladenen 3D Modells
(gerendert auf Swapchain Rendertarget mit eigenem geladenen Shader)
- Render-Features:
 - dynamische Fenstergröße mit angepasster Renderauflösung
 - Verwendung von Texturen für das Fragmentshading
 - Mip-Mapping von Texturen
 - einschaltbare anisotrope Filterung
 - anpassbares MSAA (Multisample Anti-Aliasing)
 - einschaltbarer Tiefentest
 - Auswahl zwischen einem VSync und einem Performance Modus

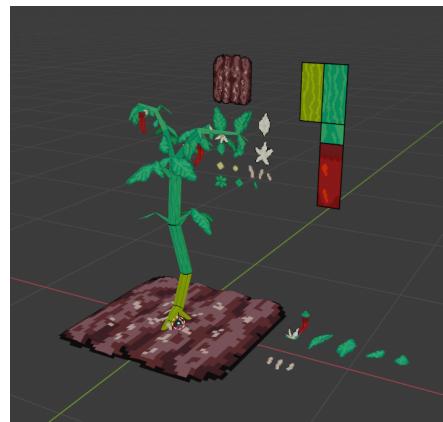


Abbildung 5.1: Testmodell für die PJEngine (erstellt in Blender)

5 Resultate

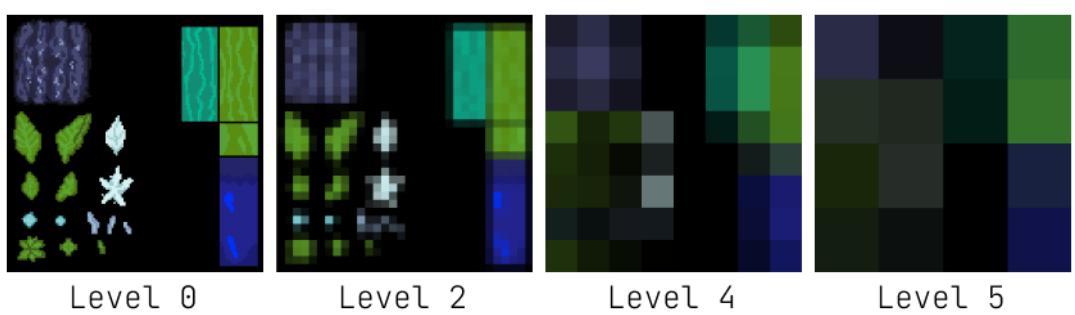
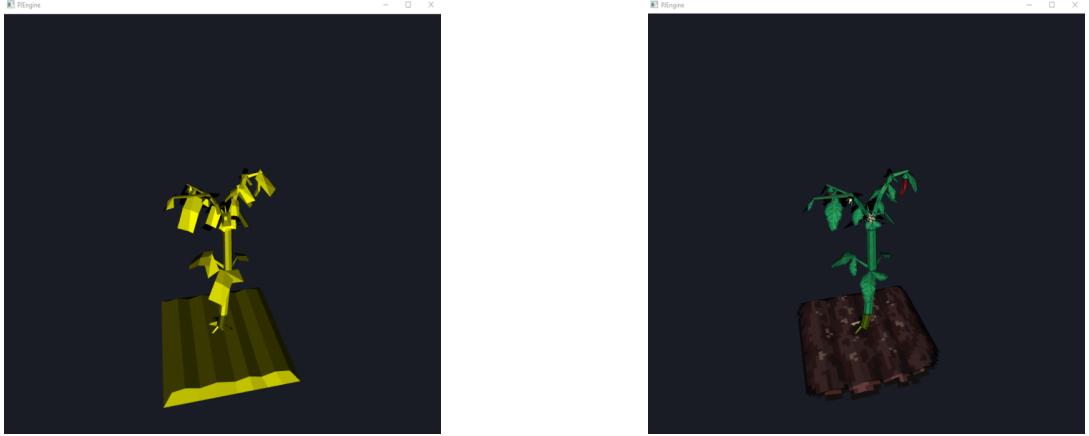


Abbildung 5.3: ausgewählte Mip-Map-Level der Testmodell-Textur auf der GPU

Um das 3D Modell mit einer Albedotextur zu rendern, werden die Texturkoordinaten je Fragment verwendet. Je nach Perspektive des Modells kann es zu bemerkbaren Sprüngen zwischen den Texturkoordinaten von Fragmenten kommen. Mit mehreren Auflösungsstufen derselben Textur s. 5.3 kann dies angegangen werden.

Der Sampler bestimmt pro Fragment das Mip-Map-Level bzw. das LOD^1 anhand der Ableitung der 2D Texturkoordinate \mathbf{u} und \mathbf{v} bezüglich der Screen Space Koordinate \mathbf{x} und \mathbf{y} ($\frac{du}{dx}, \frac{dv}{dx}, \frac{du}{dy}, \frac{dv}{dy}$) in Kombination mit weiteren festgelegten Parametern.

Zur Bestimmung des LOD für ein Fragment werden unter Vulkan die Gleichungen aus [Khr23, unter 16.5.7. Scale Factor Operation, LOD Operation and Image Level(s) Selection] von einem VkSampler befolgt. Zunächst werden die Skalierungsfaktoren ρ_x und ρ_y über die Beträge der Ableitungen nach x und y berechnet. Anschließend wird mit $\rho_{max} = max(\rho_x, \rho_y)$ der Skalierungsfaktor festgelegt, welcher bei der Ermittlung des von $\lambda_{base}(x, y)$ verwendet werden kann.

$$\lambda_{base}(x, y) = \begin{cases} shaderOp.Lod & \text{optionaler SPIR-V Operator} \\ \log_2 \frac{\rho_{max}}{\eta} & \text{ansonsten} \end{cases}$$

$$\lambda'(x, y) = \lambda_{base} + clamp(bias_{shader} + bias_{shaderOP}, -bias_{maxSamplerLod}, bias_{maxSamplerLod})$$

¹Level of Detail

5 Resultate

Mit λ' und weiteren festgelegten Parametern kann der LOD Parameter λ bestimmt werden, um damit das Mip-Map-Level für das jeweilige Fragment festzulegen.

$\eta = \min(\rho_x/\rho_y, \max_{Aniso})$ in $\log_2 \frac{\rho_{max}}{\eta}$ beschreibt ein Anisotropieverhältnis zu ρ_{max} , welches den zu ermittelnden LOD Parameter bei eingeschalteter anisotropischer Filterung beeinflusst.

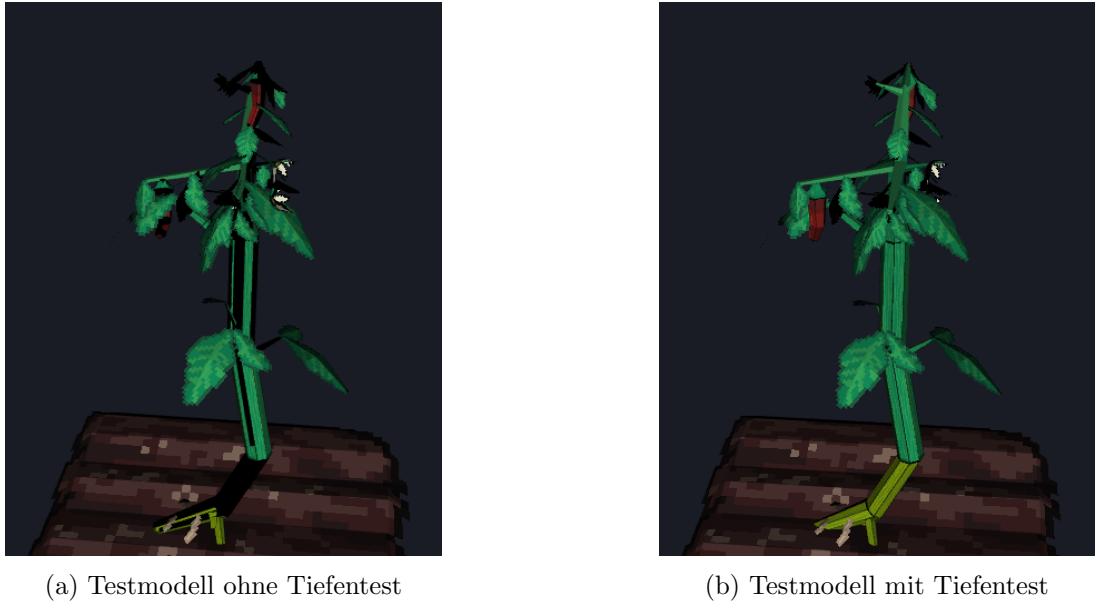


Abbildung 5.4: Integration des Tiefentest

Die Integration eines Rendertargets für den Tiefentest verbessert die Darstellung der Oberfläche des 3D Modells. Ohne Tiefentest sind neben schwarzen Artefakten auch einige Dreiecke perspektivisch vor andere Dreiecke verschoben worden. Die perspektivisch richtige Darstellung wurde mit einem vorhandenen Tiefentest erzielt.

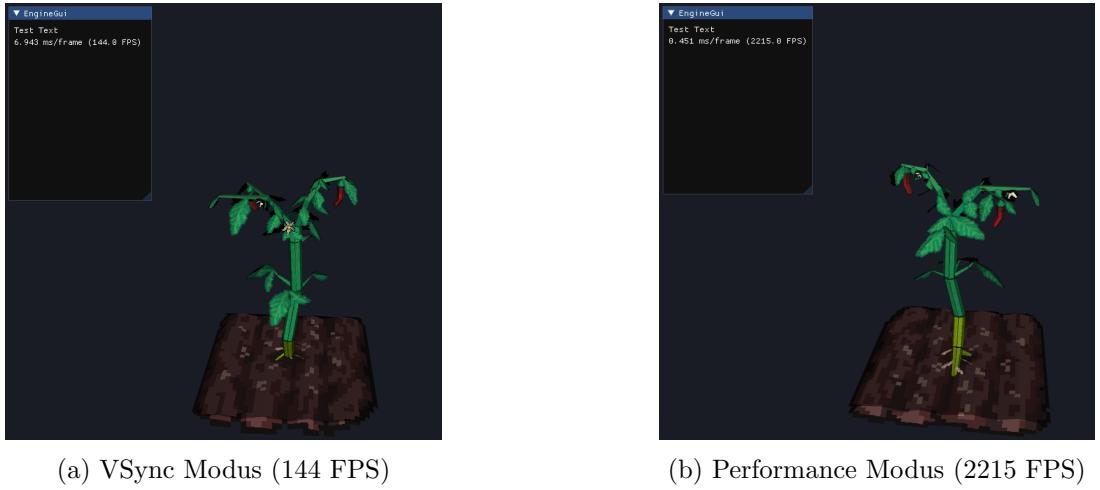


Abbildung 5.5: Rendermodi mit eingeschaltetem ImGui

5 Resultate

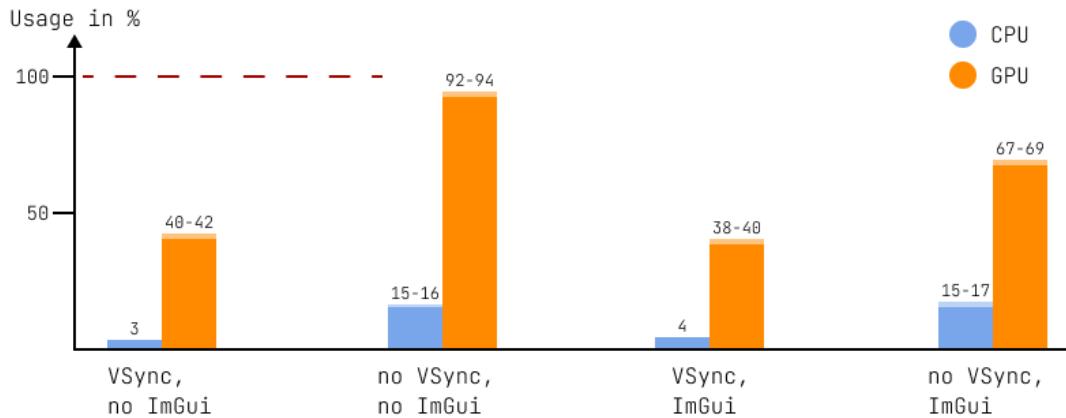


Abbildung 5.6: Auslastung der GPU/CPU je Modus

Je Modus konnten unterschiedliche Auslastungs- und Performanzdaten erzielt werden.

Die Integration von ImGui, um die *FPS*¹ live anzeigen zu lassen, weisen nur eine minimale erhöhte Auslastung der CPU auf. Der Performance Modus vollrichtet die Befehle eines einzelnen Frames mit einer 15.38194-fachen Outputrate zum VSync Modus während für beide ImGui aktiviert wurde. Bemerklich ist die reduzierte Auslastung der GPU unter ImGui ohne VSync bei gleicher Auslastung der CPU.

¹Frames per Second

6 Diskussion

Die in Abbildung 5.6 aufgezeigten Daten verdeutlichen die Auslastungsunterschiede zwischen einem limitierten und einem unlimitierten Rendervorgangs.

Die CPU-Auslastung von 15 – 16% im unlimitierten Modus lässt sich auf den einen, der Anwendung zur Verfügung gestellten Kern zurückführen. Das Testsystem selbst besitzt 6 physische Kerne. Die Prozentzahl beschreibt somit die vollständige Auslastung des gewählten Kerns.

Während bei dem Testdurchlauf (no VSync, no ImGui) die Auslastung der GPU beinahe das Maximum erzielen konnte, liegt diese im Testdurchlauf (no VSync, ImGui) nur bei 67 – 69%, da sich der Anteil der zu verrichtenden Arbeit auf der CPU mit ImGui erhöht hat und somit weniger Renderbefehle im gleichen Zeitraum an die GPU geschickt wurden. Der maximale Renderoutput aus Abbildung 5.5 (b) kann somit signifikant über 2215FPS liegen. Beim VSync Modus wurden 144 FPS erzielt, da die Anwendung auf einem 144 Hertz Monitor ausgeführt wurde, wobei 144 mal die Sekunde ein Vertical Blank durchgeführt wurde.

Die Entstehung der schwarzen Artefakte und den perspektivisch verschobenen Dreiecken (s. Abbildungen 5.4) geht aus der Reihenfolge der gerenderten Dreiecke hervor. Dabei wurde für jedes Fragment der zuletzt berechnete Wert behalten. Mit einem vorhandenen Tiefentest wird der Tiefenwert des zuletzt berechnete Farbwertes bezüglich des zu diesem Zeitpunkt vorhandenen Tiefenwertes für das jeweilige Fragment verglichen. Das Resultat ist eine perspektivisch korrekte Darstellung des zu rendernden 3D Modells.

Die in Abbildung 5.3 dargestellten Texturen entsprechen nicht den `rgba`-Farbwerten, welche tatsächlich für die jeweiligen Fragmente verwendet wurden. Die vermeintlich veränderten Farbwerte der Textur resultieren aus dem Speicherformat der Textur auf der GPU. Die einzelnen Farbkanäle werden in `bgra` anstatt `rgba` Reihenfolge gespeichert. Die Texturen aus der Abbildung interpretieren die Farbkanäle als `rgba`, während die PJEngine die `bgra` Reihenfolge der Farbkanäle für den Shadercode berücksichtigt. Bevor die PJEngine die Albedotextur für die Festlegung des Farbwertes herangezogen hat, wurde ein beliebiger gelber Farbwert für die einzelnen Fragmente gewählt.

6.1 Bedeutung der Arbeit

Die PJEngine wurde mit der Absicht entwickelt für den Leser einen möglichst schnell, aber auch einen etwas tiefergehenderen Einstieg als bereits vorhandene Quellen (s. [Bro17], [Ove23] und [Bla20]) zu bieten. Besonders bei der Recherche von tiefergehenderen Inhalten stellt die Vulkandokumentation [Khr23] wegen seines stark verteilten Charakter bezüglich der Inhalte keine einsteigerfreunde Quelle dar. Die Ausarbeitung dient somit als effizienzorientierter Einstiegspunkt in das komplexe Thema der Vulkan API.

6.2 Schwierigkeiten bei der Umsetzung des Projektes

Zum Startpunkt des Praxisprojektes wurden sämtliche verwendete Technologien und die Programmiersprache C++ vom Autor zum ersten Mal angewandt. Bei der Aneignung der Vulkan API sind besonders die in [Bedeutung der Arbeit](#) angemerckten Probleme aufgetreten, welche die Ausarbeitung mit Kapitel [Vulkan Konzepte anhand der PJEngine](#) versucht für weitere Vulkan-API-Einsteiger zu minimieren. Um einen präzisen Text für die Ausarbeitung verfassen zu können, mussten eine Vielzahl von Verständnisproblemen mit dem Zweitbetreuer im Dialog gelöst werden. Besonders beim Thema Synchronisation war eine intensive Betreuung von Nöten. Die eigen, in Rücksprache mit dem Zweitbetreuer erstellen Abbildungen unterstützten den eigenen Verständnisprozess.

Explizite Probleme sind bei der Vielzahl an aufgetretenden Problemen mit Vulkan während der Bearbeitungsphase nicht zu konkretisieren. Bei der Arbeit mit Assimp kam es zu Problemen bei der Extrahierung von Vertexdaten aus der FBX Datei und bei der Extrahierung der eingebetteten Textur.

Assimp verfolgt eine komplexe Datenhierarchie für die geladenen Objekte. Dabei mussten rekursiv die einzelnen Ebenen der Hierarchie durchlaufen werden, um sämtliche Daten nacheinander dem eigenen `PJModel` Objekt zuzuordnen.

Da die Transformationsmatrizen eines einzelnen Datenknotens von Assimp in *Row Major*¹ gespeichert werden, aber für Vulkan in *Column Major* benötigt werden, musste diese mit der Methode `Modelloader::convertToColumnMajor()` umgewandelt werden.

Bei geladenen FBX Dateien ist aufgefallen, dass die Modellskallierung sichtbar überdimensional war. Dies liegt am FBX Format selbst. Zur Lösung des Problems wurden die einzelnen Positionsdaten der Vertices mit einem Faktor von 0.01 angepasst, was zu den Renderresultaten in Abbildung [5.5](#) führt.

Das Laden einer eingebetteten Textur wird von Assimp ohne Berücksichtigung des Datenformates der eingebetteten Textur durchgeführt. Da die Textur des Testmodells ein komprimiertes Datenformat enthält, musste diese erst noch dekomprimiert werden, um auch in der PJEngine verwendbar zu sein. Dazu wurde die stb image Bibliothek herangezogen und mit der Methode `PJModel::prepareTexture()` implementiert.

¹Daten liegen Zeilenweise im Speicher

7 Fazit

Die Ausarbeitung dient mitsamt des Quellcodes als Grundlage für den Verständnisaufbau beim mit der Vulkan API in Kombination mit den weiteren ausgewählten Technologien GLFW, GLM, Assimp und stb image. Mit dieser Ausarbeitung wurde eine kollektive Quelle für tiefergehendere Vulkaninhalte geschaffen. Die Arbeit wurde mit der Philosophie geschrieben, dass Erklärungen auf purer Anwendungsebene wie in bereits vorhandenen tutorialartigen Quellen zu oberflächlich für die praxisorientierte Anwendung mit einer verbosen Bibliothek wie der Vulkan API.

Mit der PJEngine kann, wie in [Resultate](#) aufgelistet, ein beliebiges 3D Modell mit eingebetteter Textur visualisiert werden. Für die Darstellung wurden Features wie das Mip-Mapping, ein Tiefentest und das MSAA umgesetzt. Die Anwendung wird im VSync und Performance Modus mit dynamischer Fenstergröße angeboten.

Die Arbeit mit einer verbosen Technologien wie Vulkan war spürbar durch die schiere Quantität an Code, welcher geschrieben werden musste, um ein einfaches Dreieck in Echtzeit zu rendern. Jedoch flachte die zukommende Menge nach einem bestehenden Gerüst stark ab. Die Einarbeitung in die einzelnen Konzepte von Vulkan war sehr zeitintensiv. Besonders die in Vulkan integrierten Synchronisationselemente verlangten einen Großteil der Gesamtrecherche und ein genaues Coden war von Nöten. Wie eine Vulkan-gestützte Echtzeitanwendung mit bestehenden Technologien realisiert werden kann, wurde über die Ausarbeitung hinweg anhand der eigenen Herangehensweise mit der PJEngine demonstriert.

Literatur

- [ass] assimp. *Asset-Importer-Lib.* zuletzt aufgerufen am 2. August 2023. URL: https://assimp-docs.readthedocs.io/_/downloads/en/v5.0.2/pdf/.
- [Bar] S. Barrett. *stb.* zuletzt aufgerufen am 28. Februar 2023. URL: <https://github.com/nothings/stb>.
- [Bla20] V. Blanco. *Vulkan Guide.* zuletzt aufgerufen am 24. November 2022. 2020. URL: <https://vkguide.dev/>.
- [Bro17] Brotcrunsher. *Vulkan in C++.* zuletzt aufgerufen am 24. November 2022. Jan. 2017. URL: <https://www.youtube.com/watch?v=mzVFHEmnRLg&list=PL58qjcU5nk8uH9mmlASm4SFy1yuPzDAHO>.
- [Cre] G.-T. Creation. *glm.* zuletzt aufgerufen am 31. Juli 2023. URL: <https://github.com/g-truc/glm>.
- [Inc] K. Inc. *CMake Reference Documentation: Introduction.* zuletzt aufgerufen am 27. Juli 2023. URL: <https://cmake.org/cmake/help/latest/index.html>.
- [Joh23] P. Johne. *PJEngine : A Platform-Independent Real-Time Renderer Using The Vulkan SDK.* zuletzt aufgerufen am 23. Juli 2023. 2023. URL: <https://github.com/Paul-Johne/PJEngine>.
- [Khr23] Khronos. *Vulkan 1.3.250 - A Specification (with all registered Vulkan extensions).* zuletzt aufgerufen am 11. Mai 2022. Jan. 2023. URL: <https://registry.khronos.org/vulkan/specs/1.3-extensions/html/index.html>.
- [Lun] LunarG. *Architecture of the Vulkan Loader Interfaces.* zuletzt aufgerufen am 28. April 2023. URL: https://vulkan.lunarg.com/doc/view/1.2.189.0/linux/loader_and_layer_interface.html.
- [Mun+21] R. Mun, J. Zulauf, J. Gebben und J.-H. Fredriksen. *Understanding Vulkan Synchronization.* zuletzt aufgerufen am 25. Mai 2023. März 2021. URL: <https://www.khronos.org/blog/understanding-vulkan-synchronization>.
- [Org22a] G. Organisation. *Input Guide.* zuletzt aufgerufen am 7. Juli 2023. 2022. URL: https://www.glfw.org/docs/latest/input_guide.html.
- [Org22b] G. Organisation. *Window reference.* zuletzt aufgerufen am 7. Juli 2023. 2022. URL: https://www.glfw.org/docs/latest/window_guide.html.
- [Ove23] A. Overvoorde. *Vulkan Tutorial.* zuletzt aufgerufen am 6. April 2023. Apr. 2023. URL: https://vulkan-tutorial.com/resources/vulkan_tutorial_en.pdf.

Literatur

- [SK17] G. Sellers und J. M. Kessenich. *Vulkan programming guide: the official guide to learning Vulkan*. Boston: Addison-Wesley, 2017.

Eidesstattliche Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben.

Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Ort, Datum

Unterschrift