UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

COMPUTERVISUALISTIK

Raytracing mit Vulkan

Bachelorarbeit

Zur Erlangung des Grades Bachelor of Science (B.Sc.)
im Studiengang Computervisualistik

vorgelegt von

Jasper Grimmig

Erstgutachter:	Prof. Dr. Stefan Müller (Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter:	Kevin Keul (Institut für Computervisualistik, AG Computergraphik)

Koblenz, im März 2018

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden.	☐	☐

. . . . . .

(Ort, Datum) (Unterschrift)

Zusammenfassung

Der Schwerpunkt der vorliegenden Bachelorarbeit war die Entwicklung eines einfachen Raytracerprogrammes unter der Verwendung der Vulkan API, und das Einschätzen des Mehraufwandes im Vergleich zum Performancegewinn. Das Programm wird in dieser Arbeit vorgestellt. Die Vulkan Komponente des Programms wird detailliert erklärt. Anschließend wird das Programm mit einem, unter der Verwendung von OpenGL geschriebenen, ähnlichen Raytracerprogramm verglichen. Beide Programme verwenden dabei den gleichen Raytracer, der im Fragmentshader implementiert ist. Der Test ergibt, dass der mithilfe von Vulkan geschriebene Raytracer deutlich langsamer ist, als das zum Vergleich dienende OpenGL Programm.

Abstract

The focus of this bachelor thesis was the development of a simple raytracer application that uses the Vulkan API, and evaluate the additional effort in comparison to the gain in performance. The program is presented in this thesis. The Vulkan component of the application is described in detail. Following this is a comparison between the raytracer application and a similar application that uses OpenGL. Both programs use the same raytracer that is implemented in the fragmentshader. In the test the Vulkan application is much slower than the OpenGL application it is compared to.

Inhaltsverzeichnis

Einleitung

Mit Vulkan ist seit Frühjahr 2016 ein neues Graphik-API verfügbar, das gegenüber seinem logischen Vorgänger OpenGL eine flexiblere Programmierung und erhöhte Performance verspricht. Dem gegenüber steht allerdings ein deutlich höherer Programmieraufwand. In dieser Arbeit wird der im Rahmen dieser Arbeit geschriebene Vulkan Raytracer vorgestellt. Anschließend wird der Raytracer mit einem OpenGL Raytracer verglichen. Dabei wurde darauf geachtet, dass beide Programme über vergleichbare Eigenschaften verfügen. Die Implementation in Vulkan wird detailliert erklärt. Der Vergleich konzentriert sich dabei auf die Leistung der beiden Programme am Beispiel von Szenen mit unterschiedlicher Komplexität. Anschließend wird das Ergebnis mit Bezug auf den unterschiedlichen Programmieraufwand eingeschätzt.

1. Was ist Vulkan?

Vulkan ist ein low-level Application Programming Interface (API) der nächsten Generation für Graphik- und Compute-Hardware. Vergleichbare APIs sind Metal von Apple und Direct3D 12 von Microsoft. Das API wurde am 16.2.2016 von der Khronos Group veröffentlicht und gilt als logischer Nachfolger zu dem bereits über 20 Jahre alten API OpenGL, das ebenfalls von Khronos entwickelt wird. Vulkan wurde jedoch nicht zu dem Zweck entwickelt OpenGL zu ersetzen, beide APIs werden parallel weiterentwickelt. Vulkan ist als API eigenständig und besitzt keine Abwärtskompatibilität zu OpenGL. Genau wie OpenGL ist Vulkan hauptsächlich dafür gedacht, Computergraphiken mithilfe von Rasterisierung zu erzeugen, allerdings bietet Vulkan dabei eine höhere Flexibilität aber auch eine höhere Komplexität. Vulkan kann mithilfe von Compute Pipelines aber auch für nicht-Graphik Anwendungen verwendet werden, die von der hohen Parallelität einer GPU profitieren können. [SK16] [KHR03]

1.1 Worin unterscheidet sich Vulkan von OpenGL?

Vulkan verspricht vor allem eine höhere Performance und eine Portabilität auf mehr Systeme als OpenGL oder mit Vulkan vergleichbare APIs. Vulkan wird zurzeit von verschiedenen Versionen von Windows, Linux und Android unterstützt. Dabei gibt es kein spezielles Vulkan Subset für mobile Systeme wie es bei OpenGL mit OpenGL ES der Fall ist. Mit MoltenVK existiert auch eine Implementation die die Verwendung der Vulkan API auf iOS und macOS ermöglicht. Allerdings benötigt Vulkan im Vergleich zu OpenGL relativ neue Graphikhardware und auch entsprechend aktuelle Treiber. Die höhere Performance wird bei der Vulkan API durch einen, wie oben bereits erwähnt, normalerweise sehr viel höheren Programmieraufwand erreicht. Der Programmierer muss sich beispielsweise um viele Dinge kümmern, die bei OpenGL vom Treiber übernommen werden. Dazu gehören unter anderem die Auswahl der Hardware, Synchronisierung und Speichermanagement. Das Fehlermanagement fällt fast komplett weg. Vulkan führt standardmäßig nur

die nötigsten Fehlertests durch, was sich positiv auf die Performance auswirkt, da sich mit dem wegfallen der Tests der Overhead reduziert. Während der Entwicklung einer Vulkan Anwendung können Fehlertests aber über Layer aktiviert werden. Außerdem ist es, anders als bei OpenGL, in Vulkan möglich mehrere CPU Threads gleichzeitig zu verwenden. Das kann nützlich sein, wenn eine Anwendung auf der CPU Seite limitiert ist. Grundsätzlich ist es Aufgabe des Programmierers dafür zu sorgen, dass Vulkan performant arbeiten kann. Ein ineffizientes Vulkan Programm kann unter Umständen auch langsamer sein als ein vergleichbares effizientes OpenGL Programm. Vulkan hat allerdings auch Grenzen, wenn es fehlerfrei und effizient verwendet wird. Wenn ein OpenGL Programm beispielsweise durch eine ausgelastete GPU limitiert wird, ist das auch sehr wahrscheinlich bei Vulkan der Fall. [KHR03] Trotzdem ist auch eine effiziente Nutzung der GPU möglich. [WIT] [SK16]

1.2 SPIR-V

Shader werden für OpenGL in GLSL geschrieben. Vulkan unterstützt für Shader standardmäßig nur SPIR-V. SPIR-V ist eine plattformunabhängige Zwischensprache, die ebenfalls von der Khronos Group spezifiziert wurde. Die Sprache wird für verschiedene Shader und Kernel von Khronos APIs verwendet. SPIR-V bietet einige Vorteile gegenüber der Verwendung einer höheren Sprache wie GLSL. Allerdings ist es für Vulkan wahrscheinlich am wichtigsten, dass bei SPIR-V die ersten Schritte der Kompilierung bereits abgeschlossen sind. Dadurch sind SPIR-V Shader nicht so sehr von Treiber-Compilern abhängig wie es beispielsweise bei GLSL Shadern der Fall ist. Als Folge davon ist es möglich ein konsistenteres Verhalten der Shader und damit eine bessere Portabilität zwischen Systemen mit unterschiedlicher Hardware und verschiedenen Treibern sicherzustellen. Es ist für Vulkan nicht wichtig wie die SPIR-V Shader erzeugt werden, solange der resultierende Shader korrekt ist. Allerdings hat Khronos eine leicht abgeänderte Spezifikation von GLSL für Vulkan bereitgestellt, um sicherzustellen, dass es mindestens eine höhere Sprache gibt, in der Shader Module für Vulkan geschrieben werden können. Um SPIR-V aus GLSL zu erzeugen, kann beispielsweise das Programm GLSlang verwendet werden, dass ebenfalls von Khronos bereitgestellt wird. [SK16] [KOK] [KES]

2. Übersicht über Vulkan und das Raytracerprogramm

In diesem Kapitel wird das Raytracerprogramm beschrieben. Anschließend werden der Vulkan Anteil und die darin vorkommenden Konzepte erklärt.

2.1 Zusammenfassung des Programms

Das im Rahmen der vorliegenden Arbeit entwickelte Programm ist ein einfacher Raytracer, der stark an den OpenGL Raytracer aus dem CVK (Computer Visualistik Koblenz) Framework angelehnt ist. Das CVK wurde von der Arbeitsgruppe Computergraphik der Universität Koblenz-Landau entwickelt. Der Raytracer ist in der Lage Kugeln und Dreiecke mit verschiedenen Materialeigenschaften zu rendern. Das Beleuchtungsmodell entspricht der Phong-Beleuchtung. Die Sekundärstrahlen ermöglichen die Darstellung von Schatten, sowie von glasartigen und spiegelnden Materialien für Objekte. Der Raytracer verwendet keine Beschleunigungsdatenstruktur und der gesamte Raytracing Schritt findet vollständig im Fragmentshader statt.

Das Raytracerprogramm besteht im Wesentlichen aus drei Komponenten, dem Kontrollprogramm, der Szenenklasse, die die Informationen über die Szene speichert und dem eigentlichen Vulkan Programm, das für das Raytracing zuständig ist.

Das Kontrollprogramm ist dafür zuständig die Funktionen des Vulkanprogramms in der richtigen Reihenfolge aufzurufen und Szene und Kamera an das Vulkanprogramm zu übergeben.

Die Szeneklasse wird benutzt um die Informationen über die Szene zu kontrollieren. Alle Bestandteile der Szene werden mithilfe von Strukturen definiert. Informationen, die nur ein einziges mal pro Szene vorkommen wie die Hintergrundfarbe können mithilfe von Settern gesetzt werden. Szenenobjekte wie Lichtquellen und Kugeln können zu Listen hinzugefügt werden. Die Getter geben Pointer auf die Listen und Variablen zurück. Diese werden in dem Vulkan Programm benötigt um die Daten in die Buffer des Fragmentshaders zu kopieren. Darüber hinaus verfügt die Klasse über Funktionen die es ermöglichen CVK::Geometry Objekte in Dreiecke umzuwandeln und .objx Dateien zu laden. Beide Funktionen sind modifizierte Versionen von Funktionen aus der Klasse CVK_RT_SCENE des CVK.

Zusätzlich werden mehrere Klassen aus dem CVK verwendet. Wenn es notwendig war wurden diese Klassen so verändert, dass sie ohne OpenGL verwendet werden können. Die übernommenen Klassen sind CVK::Geometry, CVK::Camera sowie einige darauf aufbauende beziehungsweise dafür notwendige Klassen. Für die Camera Objekte existiert ebenfalls eine Funktion, die die Kamera in ein Frustum für den Raytracer umrechnet.

Das Vulkan Programm erstellt zunächst eine Vulkan-Instanz, um das API zu initialisieren. Mithilfe der Instanz können die Hardwarekomponenten des Host-Systems, die Vulkan unterstützen, erkannt und aufgelistet werden. Der Host ist normalerweise ein Prozessor, auf dem das Vulkan Programm läuft. Danach wird die Graphikkarte aus diesen Komponenten ausgewählt und ein Device Objekt erzeugt, das die logische Repräsentation der Graphikkarte darstellt. Diese Repräsentation wird benutzt, um Arbeiten auf der Graphikkarte auszuführen. Bei dem Erstellen der beiden Objekte werden Erweiterungen (Extensions) für das API geladen, die für die Darstellung der Renderingergebnisse in einem Fenster benötigt werden. Um das Fenstermanagement zu vereinfachen, wird die Bibliothek GLFW verwendet. Anschließend werden Storage und Uniform Buffer für die Shader erzeugt, die die Informationen zur Raytracingszene enthalten. Danach wird ein weiterer Buffer erstellt, der die Vertices für das Screen filling Quad enthält. Folgend wird ein Surface, das ein Fenster repräsentiert, und eine Swapchain erzeugt, die die Bilder enthält, die in dem Fenster abgebildet werden können. Die dafür notwendigen Funktionen sind Bestandteil der zuvor erwähnten Erweiterungen. Dann wird die Rendering Pipeline und deren Subpass erstellt. Als letztes wird ein Command Buffer aufgenommen, der die Anweisungen für die Graphikkarte enthält. Die enthaltenen Kommandos binden Ressourcen und Pipeline und definieren den Renderpass. Danach wird die Rendering Schleife gestartet. Anschließend werden Rendering und Anzeigen des Ergebnisses in der Schleife wiederholt bis das Fenster geschlossen wird.

2.2 Das Vulkanprogramm

Für die nachfolgende Erklärung des Vulkan Teils des Raytracer wurden insbesondere der Vulkan Programming Guide [SK16] und die Vulkan Spezifikation [KHR01][KHR02] als Quellen genutzt. Bei der Entwicklung des Programms wurde zusätzlich das Vulkan Tutorial von Alexander Overvoorde verwendet. [OVE]

Da Vulkan ein low-level API ist, sind bei den meisten Funktionsaufrufen sehr viele Einstellungen erforderlich. Diese Einstellungen werden zu großen Teilen in der Form von C Strukturen übergeben, was einen großen Teil des Schreibaufwandes in Vulkan ausmacht. Es gibt zwei Parameter, die bei fast jedem Struct dieser Art in Vulkan vorkommen, sType und pNext. sType hat normalerweise die Form VK_STRUCTURE_TYPE gefolgt von dem Namen der Struktur, wie unten in dem Beispiel zu sehen ist. Wie sType ist pNext ist ein Pointer, der auf eine extensionabhängige Struktur zeigen kann. Wenn keine entsprechende Extension verwendet wird, was in dem Raytracerprogramm nicht passiert, ist dieser Pointer grundsätzlich NULL. Diese Parameter werden allerdings normalerweise nur dann benötigt, wenn die Struktur direkt als Parameter an eine Funktion übergeben wird. Strukturen, die Komponenten von anderen Strukturen sind, können oft darauf verzichten.

Ein anderer Parameter der oft in Vulkan Strukturen vorkommt ist flags. Dieser Parameter erwartet einen Bitwert, der genauer Spezifiziert wie das, was durch

die Struktur beschrieben wird, genau verwendet wird. Das ist aber oft nur in speziellen Fällen notwendig und bei vielen Strukturen auch noch nicht möglich, da in der zum Zeitpunkt der Ausarbeitung aktuellen Vulkan Version nicht für alle Strukturen flags definiert worden sind. Die flags Komponenten sind in diesem Fall Reservierungen in der Spezifikation um eine spätere Nutzung zu erleichtern. Werden keine flags benutzt oder stehen keine zur Verfügung wird die Komponente auf 0 gesetzt.

Nachfolgend ist ein Beispiel zu sehen, wie Strukturen oft in Vulkan genutzt werden.

```
VkExampleInfo example = {};
example.sType = VK_STRUCTURE_TYPE_EXAMPLE_INFO;
example.pNext = NULL;
example.flags = 0;
example.exampleComponent = &variable;

vkDoExample(device, &exampleInfo);
```

Ein großer Teil der Vulkan Funktionen gibt einen VkResult Wert zurück, der davon abhängig ist ob die Funktion erfolgreich ausgeführt wurde. Es gibt sowohl für erfolgreiches als auch für nicht erfolgreiches ausführen mehrere Werte. VkResult wird für die Erklärung des Programms größtenteils ignoriert. Der Raytracer überprüft nach der Ausführung jeder Funktion die einen VkResult Wert zurückgibt, ob ein anderer Wert als VK_SUCCESS zurückgegeben wurde. Wenn ja wird ein Fehler geworfen und der entsprechende VkResult Wert wird ausgegeben. Dabei wird ignoriert, dass VK_SUCCESS nicht der einzige Wert für eine erfolgreiche Ausführung ist, da kein anderer Wert erwartet wird.

2.2.1 GLFW

Der Vulkan Raytracer benutzt die Open Source Bibliothek GLFW für das Fenstermanagement. GLFW wurde ursprünglich für OpenGL Anwendungen entwickelt, unterstützt aber mittlerweile auch Vulkan. Um GLFW mit Vulkan zu benutzen sind folgende Zeilen notwendig:

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

Wenn GLFW verwendet wird, ist es nicht mehr nötig vulkan.h zu den includes hinzuzufügen, da das automatisch mit GLFW passiert. Danach kann GLFW wie bei OpenGL verwendet werden, um ein Fenster zu erstellen.

```
glfwInit();
glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);

window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan
Raytracer", NULL, NULL);
```

Die Funktion `glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API)` informiert GLFW darüber, dass OpenGL nicht verwendet wird und kein OpenGL Kontext erstellt werden soll. GLFW benötigt außerdem mehrere Extensions, die auf Instanz- bzw. Deviceebene geladen werden müssen. Es werden in jedem Fall die Extension `VK_KHR_surface` und gegebenenfalls weitere plattformabhängige Extensions auf Instanzebene benötigt. Auf Deviceebene muss VK_KHR_swapchain geladen werden. [GVG] Die Extensions werden später genauer erklärt.

2.2.2 VkInstance

Das erste Objekt das für eine Vulkan Anwendung erzeugt werden muss ist VkInstance. In Vulkan gibt es keinen global State, weshalb der State der Anwendung in der Instance festgehalten wird. Das Erzeugen des VkInstance Objekts initialisiert darüber hinaus die Vulkan Bibliothek. Die Funktion zur Erzeugung einer VkInstance ist

```
vkCreateInstance(&instanceCreateInfo, nullptr, &instance)
```

instanceCreateInfo ist ein Pointer auf eine VkInstanceCreateInfo Struktur, das die für die Erstellung notwendigen Parameter enthält. Der zweite Parameter ist ein optionaler Pointer, der benötigt wird um einen externen Speicherallokator zu verwenden. Das ist aber nur Sinnvoll wenn dieser Allokator besser geeignet ist als der interne Allokator von Vulkan. Wird wie im Raytracerprogramm kein eigener Allokator verwendet ist pAllocator NULL oder nullptr. Der letzte Parameter pInstance ist ein Pointer auf ein VkInstance Handle. Das Handle wird wie andere Objekte auch erstellt:

```
VkInstance instance;
```

In VulkanRaytracer ist das Handle der Instance eine private Variable der Klasse. Nachfolgend kann davon ausgegangen werden, dass alle Handle, die blau hervorgehoben sind, private Variablen sind.

Wie bereits erwähnt ist VkInstanceCreateInfo eine Struktur, das wie folgt befüllt wird:

```
	VkInstanceCreateInfo instanceCreateInfo = {};
	instanceCreateInfo.sType = 
VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
	instanceCreateInfo.pNext = nullptr;
	instanceCreateInfo.flags = 0;
	instanceCreateInfo.pApplicationInfo = &applicationInfo;
	instanceCreateInfo.enabledLayerCount = 
validationLayers.size();
	instanceCreateInfo.ppEnabledLayerNames = 
validationLayers.data();
	instanceCreateInfo.enabledExtensionCount = 
GlfwExtensionCount;
	instanceCreateInfo.ppEnabledExtensionNames = 
glfwExtensions;
```

pApplicationInfo ist ein Pointer auf eine weitere Struktur, die Informationen über die Anwendung enthält. Diese Struktur ist technisch gesehen optional, sollte aber verwendet werden.

```
	VkApplicationInfo applicationInfo = {};
	applicationInfo.sType = 
VK_STRUCTURE_TYPE_APPLICATION_INFO;
	applicationInfo.pNext = nullptr;
	applicationInfo.pApplicationName = "vulkan raytracer";
	applicationInfo.applicationVersion = 
VK_MAKE_VERSION(1,0,0);
	applicationInfo.pEngineName = "vulkan raytracer";
	applicationInfo.engineVersion  = VK_MAKE_VERSION(1,0,0);
	applicationInfo.apiVersion = VK_API_VERSION_1_0;
```

Die Komponenten pApplicationName und applicationVersion geben Namen und Version der Anwendung an, genauso geben pEngineName und enineVersion Namen und Version der Engine an. Beides kann gegebenenfalls vom Treiber erkannt werden, um Optimierungen vorzunehmen. Da der Raytracer allerdings weder eine bekannte Anwendung ist, noch eine bekannte Engine verwendet, sind die hier eigetragenen Informationen unwichtig. Die letzte Komponente apiVersion gibt an, welche Vulkan Version verfügbar sein muss um die Anwendung auszuführen. Damit ist allerdings nicht die Version gemeint, die aktuell verwendet wird, sondern die älteste noch mit dem Programm kompatible Version.

Die beiden Parameter enabledLayerCount und ppEnabledLayerNames beschreiben, wie viele und welche Layer von der Instanz geladen werden sollen. Während der Entwicklung des Programms wurde das Layer VK_LAYER_LUNARG_standard_validation verwendet. Während den

Performancetests werden allerdings keine Layer mehr verwendet. Das verwendete Layer ist Teil des LunarG SDK.

Wie bereits erwähnt erfordert GLFW die Nutzung von Vulkan Extensions. Die benötigten Informationen können mithilfe der folgenden Zeilen ermittelt werden.

```
uint32_t GlfwExtensionCount = 0;
auto glfwExtensions =
glfwGetRequiredInstanceExtensions(&GlfwExtensionCount);
```

GlfwExtensionCount wird mit der Anzahl der benötigten Informationen überschrieben und glfwExtensions enthält die Information um welche Extensions es sich handelt. Da die Surface Extensions die einzigen Extensions sind, die an dieser Stelle für den Raytracer benötigt werden, kann beides unverändert an die InstanceCreateInfo übergeben werden. Layer und Extensions sind allerdings nicht auf allen Systemen verfügbar, weshalb deren Verfügbarkeit erst überprüft werden sollte.

Vorhandene Layer können mithilfe folgenden Aufrufs überprüft werden.

```
	uint32_t layerCount = 0;
	vkEnumerateInstanceLayerProperties(&layerCount, nullptr);
	VkLayerProperties *layers = new
VkLayerProperties[layerCount];
	vkEnumerateInstanceLayerProperties(&layerCount, layers);
```

Die eigentliche Funktion vkEnumerateInstanceLayerProperties() wird zweimal aufgerufen. Der Grund dafür ist, dass diese Art von Funktion in Vulkan oft für zwei verschiedene Dinge genutzt werden kann, das Ermitteln der Anzahl von verfügbaren Ressourcen und das Ermitteln von deren Inhalt. Wird nur ein Pointer auf einen Integer, im Fall des Beispielcodes layerCount, und statt einem Array NULL übergeben, dann wird die Integervariable mit der Anzahl der gesuchten Informationen überschrieben. Im zweiten Fall muss ein Array übergeben werden. Das Array wird dann mit layerCount VkLayerProperties Strukturen gefüllt. Bei der zweiten Version bleibt layerCount unverändert, da erwartet wird, dass die richtige Größe angegeben wurde. Das Ermitteln der Extensions funktioniert analog.

```
	uint32_t extensionCount = 0;
	vkEnumerateInstanceExtensionProperties(nullptr,
&extensionCount, NULL);
	VkExtensionProperties *extensions = new
VkExtensionProperties[extensionCount];
	vkEnumerateInstanceExtensionProperties(nullptr,
&extensionCount, extensions);
```

Ein Unterschied zu vkEnumerateInstanceLayerProperties() ist, dass die Funktion vkEnumerateInstanceExtensionProperties() noch einen weiteren Parameter erwartet. Dabei handelt es sich um einen Pointer auf den Namen eines Layers oder NULL. Wird ein Name übergeben, werden die von diesem Layer bereitgestellten Extensions aufgelistet. Wenn nicht, werden nur die von Vulkan bereitgestellten Extensions angegeben.

2.2.3 VkDevice

Nachdem die Instanz erstellt wurde, können die verfügbaren physical Devices aufgelistet werden. Wie schon erwähnt repräsentieren Physical Devices die Hardware, auf der Vulkan läuft. Die Devices werden mit folgendem Code ermittelt:

```
	uint32_t PhysicalDeviceCount = 0;
	checkForError( vkEnumeratePhysicalDevices(instance, 
&PhysicalDeviceCount, nullptr) );

	physicalDeviceList.resize(PhysicalDeviceCount);

	checkForError( vkEnumeratePhysicalDevices(instance, 
&PhysicalDeviceCount, physicalDeviceList.data()) );
```

PhysicalDeviceList ist eine VkPhysicalDevice Vektorliste. Aus den ermittelten Devices muss mindestens eins ausgewählt werden, um ein logisches Device zu erstellen. Im Fall des Raytracers wird die Graphikkarte des Testrechners aus der Liste der Devices ausgewählt. Das logische Device wird mithilfe folgender Funktion erstellt:

```
vkCreateDevice(physicalDeviceList[physicalDeviceIndex], 
&deviceCreateInfo, nullptr, &device)
```

Wie bereits erwähnt ist physicalDeviceList die Liste der verfügbaren physicalDevices, physicalDeviceIndex ist der Index der Graphikkarte, für die das Device erstellt werden soll. DeviceCreateInfo ist ein Pointer auf eine Struktur.

```
	VkDeviceCreateInfo deviceCreateInfo = {};
	deviceCreateInfo.sType = 
VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
	deviceCreateInfo.pNext = NULL;
	deviceCreateInfo.flags = 0;
	deviceCreateInfo.queueCreateInfoCount = 1;
```

```
	deviceCreateInfo.pQueueCreateInfos = &deviceQueueCreateInfo;
	deviceCreateInfo.enabledLayerCount = 0;  //deprecated
	deviceCreateInfo.ppEnabledLayerNames = NULL; //deprecated
	deviceCreateInfo.enabledExtensionCount = deviceExtensions.size();
	deviceCreateInfo.ppEnabledExtensionNames = deviceExtensions.data();
	deviceCreateInfo.pEnabledFeatures = &usedFeatures;
```

Die VkDeviceCreateInfo Struktur enthält Informationen darüber, welche Extensions auf Deviceebene und welche Features des Devices geladen werden sollen. Die Einträge für Layer sind nicht mehr relevant und werden von Vulkan ignoriert, da Layer in Vulkan 1.0 nur auf Instanzebene aktiviert werden können. An dieser Stelle wird für den Raytracer die Extension VK_KHR_swapchain geladen. Diese Extension benötigt VK_KHR_surface auf Instanzebene, und ermöglicht es Renderingergebnisse auf einem VkSurface darzustellen. Features sind optionale Fähigkeiten des Device, und dementsprechend ist die Verfügbarkeit von der Hardware abhängig. Beispiele für Features sind Tesselation- und Geometryshader. Für den Raytracer müssen keine Features aktiviert werden. VkDeviceCreateInfo enthält eine zweite Struktur, dessen Zweck es ist, die zu verwendende Queuefamily zu beschreiben. Queuefamilies enthalten Queues, an die später Graphikkartenbefehle gesendet werden können. Da es möglich ist mehrere Queuefamilies zu benutzen, muss in queueCreateInfoCount zusätzlich noch die Anzahl der verwendeten VkDeviceQueueCreateInfo Strukturen angegeben werden. Wird mehr als eine Struktur verwendet, muss in pQueueCreateInfoCount eine Liste der Strukturen übergeben werden.

```
	VkDeviceQueueCreateInfo deviceQueueCreateInfo = {};
	deviceQueueCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
	deviceQueueCreateInfo.pNext = nullptr;
	deviceQueueCreateInfo.flags = 0;
	deviceQueueCreateInfo.queueFamilyIndex = queueFamilyIndex;
	deviceQueueCreateInfo.queueCount = 1;
	deviceQueueCreateInfo.pQueuePriorities = queuePriority;
```

QueueFamilyIndex ist ein Integerwert, der dem Index der Queuefamily entspricht. Für den Raytracer wird nur eine einzige Queue benutzt, die für das Rendering benötigt wird. Deshalb muss für die Queue das *VK_QUEUE_GRAPHICS_BIT* Flag gesetzt sein, das bedeutet, dass die Queue Graphikoperationen unterstützt. Die Informationen über die verfügbaren Queues können mit folgendem Code ermittelt werden:

```
        uint32_t queueFamiliesCount = 0;

vkGetPhysicalDeviceQueueFamilyProperties(physicalDevice[physica
lDeviceIndex], &queueFamiliesCount, NULL);
        VkQueueFamilyProperties *queueFamilyProperties = new
VkQueueFamilyProperties[queueFamiliesCount];

vkGetPhysicalDeviceQueueFamilyProperties(physicalDevice[physica
lDeviceIndex], &queueFamiliesCount, familyProperties);
```

VkQueueFamilyProperties ist eine Struktur, das Informationen über eine Queuefamily enthält. Der Index der Struktur entspricht dem Index der darin beschriebenen Queuefamily. QueuePriority ist ein Float Wert zwischen 1.0 und 0.0, wobei 1 die höchste Priorität darstellt. Die Priorität entscheidet darüber, wann welche Queue verwendet wird, wobei die Vulkanspezifikation kein bestimmtes Verhalten garantiert. Da der Raytracer nur eine Queue benötigt ist dieser Wert unwichtig und wird auf die höchste Priorität gesetzt.

Das Handle der Queue kann mithilfe folgender Funktion erzeugt werden.

```
        vkGetDeviceQueue(device, queueFamilyIndex, 0, &queue);
```

device ist das Device. Der zweite und dritte Parameter geben den Index der Queuefamily und den Index der Queue an. Queue ist ein Pointer auf ein VkQueue Handle, das mit dem neuen Handle überschrieben werden soll.

2.2.4 Create Swapchain

VkSurfaceKHR und VkSwapchainKHR repräsentieren in Vulkan das Fenster bzw. die Liste der Bildobjekte, die darauf angezeigt werden können. Die Präsentation von Bildern ist nicht in der Kernfunktionalität von Vulkan enthalten. Der Grund dafür ist, dass die Präsentation im allgemeinen vom Betriebssystem übernommen wird, und sich die dafür notwendigen plattformspezifischen Bibliotheken stark unterscheiden können. Gleichzeitig braucht nicht jede Vulkan Anwendung ein aktives Fenster, beispielsweise bei nicht graphikorientierten Compute-Anwendungen. VkSurfaceKHR und VkSwapchainKHR sind jeweils Teil der Extensions VK_KHR_surface und VK_KHR_swapchain. KHR steht hier für Khronos, da die Extensions von Khronos bereitgestellt werden. Da für den Raytracer GLFW verwendet wird, kann das VkSurfaceKHR Handle mithilfe des folgenden Aufrufs erstellt werden.

```
glfwCreateWindowSurface(instance, window, nullptr, &surfaceKHR)
```

instance ist die VkInstance, window ist das zuvor erstelle GLFW Fenster. Der genaue Code für die Erstellung eines Surface ist Plattformabhängig, GLFW nimmt dem Programmierer an dieser Stelle die Arbeit ab. Anschließend muss überprüft werden, ob das Device, das die Ergebnisse später präsentieren soll dazu in der Lage ist. Das passiert mit folgendem Aufruf.

```
vkGetPhysicalDeviceSurfaceSupportKHR(physicalDevice[physicalDev
iceIndex], 0, surfaceKHR, &surfaceSupport)
```

Wenn die Präsentation unterstützt wird, wird die Boolean Variable mit VK_TRUE überschrieben, wenn nicht dann mit VK_FALSE. Es bleibt dem Programmierer überlassen, wie mit dem Ergebnis umgegangen werden soll. Die Funktion überprüft immer nur ein einziges Device und eine dazugehörige Queuefamily. Jede Queuefamily, die später für die Präsentation genutzt werden soll muss einzeln überprüft werden.

```
vkCreateSwapchainKHR(device, &swapchainCreateInfo, nullptr,
&swapchain)
```

device ist da Device Handle. swapchainCreateInfo ist eine Struktur, die nachfolgend erklärt wird. Der dritte Parameter ist der ungenutzte pAllocator. swapchain ist das Swapchainhandle.

```
      VkSwapchainCreateInfoKHR swapchainCreateInfo = {};
      swapchainCreateInfo.sType =
VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
      swapchainCreateInfo.pNext = nullptr;
      swapchainCreateInfo.flags = 0;
      swapchainCreateInfo.surface = surfaceKHR;
      swapchainCreateInfo.minImageCount = 2;
      swapchainCreateInfo.imageFormat =
VK_FORMAT_B8G8R8A8_UNORM;
      swapchainCreateInfo.imageColorSpace =
VK_COLOR_SPACE_SRGB_NONLINEAR_KHR;
      swapchainCreateInfo.imageExtent = VkExtent2D{WIDTH,
HEIGHT};
      swapchainCreateInfo.imageArrayLayers = 1;
      swapchainCreateInfo.imageUsage =
VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
      swapchainCreateInfo.imageSharingMode =
VK_SHARING_MODE_EXCLUSIVE;
      swapchainCreateInfo.queueFamilyIndexCount = 0;
      swapchainCreateInfo.pQueueFamilyIndices = nullptr;
      swapchainCreateInfo.preTransform =
VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR;
```

```
        swapchainCreateInfo.compositeAlpha =
VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
        swapchainCreateInfo.presentMode =
VK_PRESENT_MODE_FIFO_KHR;
        swapchainCreateInfo.clipped = VK_TRUE;
        swapchainCreateInfo.oldSwapchain = VK_NULL_HANDLE;
```

Die Komponente surface ist das Surface, auf dem die Renderingergebnisse präsentiert werden sollen. minImageCount entspricht der Mindestanzahl von Bildern in der Swapchain, die von dem Programm benötigt werden. Die Zahl gibt das Minimum an, es ist auch möglich, dass die Swapchain nach der Erstellung mehr Images beinhaltet. ImageFormat ist das Format der Images in der Swapchain. ImageColorSpace ist der Farbraum der Images.

imageExtent ist eine Extent2D Struktur, die Höhe und Breite der Images angibt. Die Größe der Images sollte der aktuellen Größe des Surface entsprechen. Die Struktur sieht wie folgt aus.

```
VkExtent2D{
        uint32_t        width;
        uint32_t        height;
}
```

imageArrayLayers wird benötigt, um die Anzahl der Bilder für stereoskopische 3D Anwendungen zu definieren. Für nicht stereoskopische Anwendungen wie den Raytracer ist der Wert immer 1. imageUsage gibt an, wofür die Bilder in der Swapchain verwendet werden sollen. Im Fall des Raytracers werden Colorattachments benötigt.

imageSharingMode ist der Sharing Mode, der von den Images der Swapchain verwendet wird. VK_SHARING_MODE_EXCLUSIVE bedeutet, dass das Bild nicht von mehreren Queues gleichzeitig verwendet wird. pQueueFamilyIndices ist ein Array, das die Indices der Queuefamilies enthält, die Zugriff auf die Images der Swapchain haben sollen, wenn der Sharing mode VK_SHARING_MODE_CONCURRENT entspricht.

QueueFamilyIndexCount ist die Zahl dieser Indices. Wird wie beim Raytracer ein anderer Sharing Mode verwendet, werden die beiden Einträge ignoriert. PreTransform bestimmt, ob und wie die Images vor der Präsentation Transformiert werden sollen. Für den Raytracer wird hier das VK_SURFACE_TRANSFORM_IDENTITY_BIT verwendet. Dieses Bit bedeutet, dass das Bild angezeigt wird wie es ist. Da die Y-Achse im Vergleich zu OpenGL genau umgekehrt ist, wäre hier VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_BIT sinnvoller, da dieses Bit bedeutet, dass das Bild horizontal gespiegelt wird. Allerdings wird diese Transformation nicht von dem Testsystem unterstützt.

CompositeAlpha ist ein Bitwert, der den Alpha Compositing Mode bestimmt. VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR beduetet, dass der Alphawert des Attachments ignoriert, und immer als 1.0 gesehen wird. PresentMode bestimmt den Präsentationmodus, was bedeutet, dass darüber festgelegt wird, wie Präsentationsanfragen intern verarbeitet werden. Der Raytracer benutzt First-in-first-Out.

Clipped bestimmt, ob Renderingoperationen wegfallen dürfen, wenn die dazugehörigen Regionen des Bildes nicht sichtbar sind, beispielsweise wenn das Fenster von einem weiteren Fenster teilweise verdeckt wird. Dieser Wert ist für den Raytracer VK_TRUE. Es ist nur sinnvoll Clipping mit VK_FALSE zu deaktivieren, wenn die Renderingergebnisse gelesen werden sollen oder der Shader es erfordert, dass er für alle Pixel ausgeführt wird. Allerdings ist auch mit VK_TRUE nicht garantiert, dass Clipping ausgeführt wird. OldSwapchain ist die Swapchain, die aktuell mit dem Surface verwendet wird. Wenn diese Swapchain angegeben wird, können dadurch gegebenenfalls Ressourcen wiederverwendet werden. Gleichzeitig können Bilder die bereits von dieser Swapchain entnommen wurden weiterhin angezeigt werden. Da der Raytracer nur eine einzige Swapchain verwendet und es keine alte Swapchain gibt ist dieser Wert VK_NULL_HANDLE.

2.2.6 Image Views

Nachdem die Swapchain erstellt wurde, enthält diese mindestens minImageCount VkImage Objekte. Diese Bilder können allerdings nicht immer direkt verwendet werden, da dazu mehr Informationen benötigt werden als im Bild enthalten sind. Das ist beispielsweise der Fall, wenn die Bilder in einem Framebuffer verwendet werden sollen, wie es bei dem Raytracer der Fall ist. Um ein Bild zu verwenden, muss dafür ein VkImageView Objekt erstellt werden, das die fehlenden Informationen enthält und das eigentliche Bild referenziert. Als erstes werden dazu die VkImages in der Swapchain ermittelt.

```
      vkGetSwapchainImagesKHR(device, swapchain,
&swapchainImageCount, nullptr);
      VkImage *swapchainImages = new
VkImage[swapchainImageCount];
      vkGetSwapchainImagesKHR(device, swapchain,
&swapchainImageCount, swapchainImages);
```

Die Funktion vkGetSwapchainImageKHR funktioniert analog zu den anderen Funktionen dieser Art. Device- und Swapchainhandle geben an für welche Swapchain die Funktion ausgeführt werden soll, swapchainImageCount ist ein Pointer auf eine uint32_t Variable und swapchainImages ist das Array, das die VkImage Handle enthält. VkImages können auch manuell erstellt werden,

beispielsweise um diese als Shaderressourcen zu nutzen. Ein VkImageView kann mit der Funktion vkCreateImageView erzeugt werden.

```
	vkCreateImageView(device, &imageViewCreateInfo, nullptr,
&imageViews[i]);
```

Device ist das Devicehandle, imageViewCreateInfo ist ein Pointer auf eine VkImageViewCreateInfo Struktur, pAllocator ist für den Raytracer wie immer NULL und imageViews ist ein Pointer auf ein VkImageView Handle. Da für jedes Bild in der Swapchain ein ImageView benötigt wird, muss vkCreateImageView für alle Bilder ausgeführt werden. Im Raytracerprogramm werden die resultierenden ImageViews in einer privaten Variable auf Klassenebene gespeichert.

```
	VkImageViewCreateInfo imageViewCreateInfo = {};
	imageViewCreateInfo.sType =
VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
	imageViewCreateInfo.pNext = nullptr;
	imageViewCreateInfo.flags = 0;
	imageViewCreateInfo.image = swapchainImages[i];
	imageViewCreateInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
	imageViewCreateInfo.format = surfaceFormat;
	imageViewCreateInfo.components.r =
VK_COMPONENT_SWIZZLE_IDENTITY;
	imageViewCreateInfo.components.g =
VK_COMPONENT_SWIZZLE_IDENTITY;
	imageViewCreateInfo.components.b =
VK_COMPONENT_SWIZZLE_IDENTITY;
	imageViewCreateInfo.components.a =
VK_COMPONENT_SWIZZLE_IDENTITY;
	imageViewCreateInfo.subresourceRange.aspectMask =
VK_IMAGE_ASPECT_COLOR_BIT;
	imageViewCreateInfo.subresourceRange.baseMipLevel = 0;
	imageViewCreateInfo.subresourceRange.levelCount = 1;
	imageViewCreateInfo.subresourceRange.baseArrayLayer = 0;
	imageViewCreateInfo.subresourceRange.layerCount = 1;
```

VkImageViewCreateInfo enthält Informationen dazu wie die ImageView erstellt werden soll.

Die Komponente Image ist das VkImage Handle des Bildes, für das die View erstellt werden soll. ViewType ist die Art der View, die erstellt werden soll. Die verfügbaren ViewTypes sind in der VkImageViewType Enumeration enthalten. Dieser Typ muss mit dem Typ des VkImages kompatibel sein. Da die Bilder Teil der Swapchain sind und für den Framebuffer verwendet werden sollen, ist der Typ hier VK_IMAGE_VIEW_TYPE_2D, was einem zweidimensionalen Bild entspricht. Für andere Anwendungsfälle können aber

beispielsweise auch dreidimensionale Bilder oder Cubemaps als Typ angegeben werden. Die Komponente format gibt das neue Format des Bildes an. Dieses Format muss mit dem alten Format kompatibel sein, was im allgemeinen bedeutet, dass beide Formate dieselbe Anzahl an Bit pro Pixel verwenden müssen. Da für den Raytracer keine besonderen Ansprüche an das Format bestehen, wird dasselbe Format wie bei der Erstellung der Swapchain verwendet. Component ist eine Struktur, die angibt, welcher Kanal auf welchen Kanal gemappt werden soll.

```
VkComponentMapping componentMapping;
componentMapping.r = VK_COMPONENT_SWIZZLE_IDENTITY;
componentMapping.g = VK_COMPONENT_SWIZZLE_IDENTITY;
componentMapping.b = VK_COMPONENT_SWIZZLE_IDENTITY;
componentMapping.a = VK_COMPONENT_SWIZZLE_IDENTITY;
```

Die Komponenten r, g, b und a benötigen jeweils ein Element aus der VkComponentSwizzle Enumeration. Der Name der Komponente gibt den Kanal des ImageViews an der befüllt werden soll, während der Wert, auf den die Variable gesetzt wird, angibt aus welchem Kanal des VkImage die Daten kommen sollen. VK_COMPONENT_SWIZZLE_IDENTITY oder 0 gibt dabei an, dass der Quellkanal auch der Zielkanal ist.

Die Komponente subresourceRange der VkImageViewCreateInfo Struktur ist eine weitere Struktur. Das ImageView Bild kann ein Teilset des VkImage sein. Das genaue Set wird in der Struktur VkImageSubresourceRange definiert.

```
VkImageSubresourceRange imageSubresourceRange;
imageSubresourceRange.aspectMask =
VK_IMAGE_ASPECT_COLOR_BIT;
imageSubresourceRange.baseMipLevel = 0;
imageSubresourceRange.levelCount = 1;
imageSubresourceRange.baseArrayLayer = 0;
imageSubresourceRange.layerCount = 1;
```

Die Komponente aspectMask ist ein Bitfield das aus Elementen der Enumeration VkImageAspectFlagBits besteht. Beispielsweise könnte aus einem depth-stencil Bild nur die depth Komponente für die ImageView verwendet werden. Die Bilder der Swapchain sind Farbbilder weshalb VK_IMAGE_ASPECT_COLOR_BIT verwendet werden muss. Es werden keine weiteren Bits benötigt, da Farbbilder im Normalfall nur das color aspect besitzen. Die beiden Komponenten baseMipLevel und levelCount werden verwendet, wenn ImageView nur einen Teil der Mipmaps des Elternbildes übernehmen soll. BaseMipLevel gibt dabei an, an welcher Stelle der Mip-Kette ImageView anfängt, während levelCount die Anzahl der Mip-Level angibt. Wenn, wie bei dem Raytracer, keine Mipmaps verwendet werden, sind

diese Werte standardmäßig 0 für baseMipLevel und 1 für levelCount. BaseArrayLayer und layerCount werden analog benutzt, um ein Teilset der Array Layer zu definieren. Da das Elternbild kein Array Bild ist, sind die Standardwerte auch hier 0 und 1.

In dem Raytracerprogramm werden ImageViews für alle Images der Swapchain erzeugt.

2.2.7 Descriptorsets

Ein Descriptor repräsentiert in Vulkan eine Shader Ressource, wie beispielsweise einen Buffer oder Bilder. Descriptoren werden in Vulkan in Descriptor Sets zusammengefasst, deren Aufbau von einem Descriptor Set Layout bestimmt wird. Das Layout gibt an, welche Descriptoren in dem Set gespeichert werden können. Nachdem das Set erstellt wurde muss es an eine Pipeline gebunden werden. Descriptoren können sowohl von Compute- als auch von Graphikpipelines genutzt werden.

```
	VkDescriptorSetLayoutBinding descriptorSetLayoutBinding[]
= {

	{0, //binding
	VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, //descriptorType
	1, //descriptorCount
	VK_SHADER_STAGE_FRAGMENT_BIT, //stageFlags
	nullptr}, // pImmutableSamplers

	};

	VkDescriptorSetLayoutCreateInfo
descriptorSetLayoutCreateInfo = {};
	descriptorSetLayoutCreateInfo.sType =
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
	descriptorSetLayoutCreateInfo.pNext = nullptr;
	descriptorSetLayoutCreateInfo.flags = 0;
	descriptorSetLayoutCreateInfo.bindingCount = 9;
	descriptorSetLayoutCreateInfo.pBindings =
&descriptorSetLayoutBinding[0];

	vkCreateDescriptorSetLayout(device,
&descriptorSetLayoutCreateInfo, nullptr,
&descriptorSetLayout[0]);
```

Ein Descriptor Set Layout wird mithilfe der Funktion vkCreateDescriptorSetLayout erstellt. Das Device ist wie immer das Device,

das für die Erstellung des Objekts verwendet werden soll. pCreateInfo ist die für die Erstellung notwendige Struktur VkDescriptorSetLayoutBinding. pAllocator ist wie immer nullptr und pSetLayout ist das Set Layout Handle, das mit dem resultierenden Handle überschrieben wird.

Die Struktur VkDescriptorSetLayoutCreateInfo besitzt neben den üblichen Komponenten nur zwei weitere, bindingCount und pBindings. pBindings ist ein Array, das VkDescriptorSetLayoutBinding Strukturen enthält. BindingCount gibt dabei die Anzahl der Elemente des Arrays an.

VkDescriptorSetLayoutBinding beschreibt ein einzelnes Binding. Binding ist die Binding Nummer des Descriptors, dieselbe Nummer wird später im Shader verwendet, um auf die Ressource zuzugreifen. DescriptorType gibt an, um welche Art von Descriptor es sich handelt. Der Raytracer benutzt Storage- und Uniform Buffer Objekte. DescriptorCount ist die Anzahl der Descriptoren, die für dieses Binding erstellt werden sollen. Der Shader kann dann auf ein Array der Descriptoren zugreifen. Ist der Wert 0 kann nicht auf das Binding zugegriffen werden. stageFlags gibt an, in welcher ShaderStage auf den Descriptor zugegriffen werden kann. Der Raytracer benötigt die Ressourcen nur für den Fragment Shader, weshalb alle Bindings nur das VK_SHADER_STAGE_FRAGMENT_BIT gesetzt haben.

pImmutableSamplers wird verwendet, wenn der Descriptor einen Sampler definiert, und der Sampler bei der Erstellung des Layouts permanent daran gebunden werden soll. Sampler werden benutzt um Daten aus Bildern zu lesen. Sollen immutable Samplers verwendet werden, muss ein Array aus Sampler Handeln übergeben werden. Ist der Parameter nullptr kann der Sampler zu einem späteren Zeitpunkt dynamisch gebunden werden. Wird kein Sampler für das Binding definiert, wie bei dem Raytracer, sollte der Parameter ebenfalls nullptr sein.

```
	VkDescriptorPoolSize descriptorPoolSize[] = {{
				VK_DESCRIPTOR_TYPE_STORAGE_BUFFER,
				6}, //number of descriptors
				{VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,
				3}};

	VkDescriptorPoolCreateInfo descriptorPoolCreateInfo = {};
	descriptorPoolCreateInfo.sType =
VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
	descriptorPoolCreateInfo.pNext = nullptr;
	descriptorPoolCreateInfo.flags = 0;
	descriptorPoolCreateInfo.maxSets = 1;
	descriptorPoolCreateInfo.poolSizeCount = 2;
```

```
	descriptorPoolCreateInfo.pPoolSizes =
&descriptorPoolSize[0];

	vkCreateDescriptorPool(device, &descriptorPoolCreateInfo,
nullptr, &descriptorPool);
```

Descriptor Sets können nicht sofort erzeugt werden, sondern müssen aus einem Descriptor Pool allokiert werden. Descriptor Pools können mithilfe der Funktion vkCreateDescriptorPool() erzeugt werden. Deren Parameter funktionieren wie bei den meisten vkCreate* Funktionen, device ist das Device, das zur Erstellung des Pools verwendet werden soll, pCreateInfo ist ein Pointer auf eine VkDescriptorPoolCreateInfo, pAllocator ist NULL und pDescriptorPool ist ein Pointer auf ein Descriptor Pool Handle.

Die VkDescriptorPoolCreateInfo besitzt wie immer sType und pNext. Es ist ein Flag verfügbar, das allerdings für den Raytracer nicht benötigt wird. maxSets gibt an, wie viele Descriptorsets unabhängig von deren Größe höchstens aus dem Pool allokiert werden können. pPoolSizes ist ein Array der Länge poolSizeCount, das VkDescriptorPoolSize Strukturen enthält.

VkDescriptorPoolSize enthält Informationen darüber, welche und wie viele Descriptoren von dem Pool allokiert werden können.

```
	VkDescriptorSetAllocateInfo descriptorSetAllocateInfo={};
	descriptorSetAllocateInfo.sType =
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
	descriptorSetAllocateInfo.pNext = nullptr;
	descriptorSetAllocateInfo.descriptorPool=descriptorPool;
	descriptorSetAllocateInfo.descriptorSetCount = 1;
	descriptorSetAllocateInfo.pSetLayouts =
&descriptorSetLayout[0];

	vkAllocateDescriptorSets(device,
&descriptorSetAllocateInfo, &descriptorSet[0]);
```

Nachdem der Descriptor Pool erzeugt wurde, kann das Descriptorset allokiert werden. Das passiert mit der Funktion vkAllocateDescriptorSets(). Device ist für den Raytracer wie immer die Graphikkarte, pAllocateInfo ist eine VkDescriptorSetAllocateInfo Struktur und pDescriptorsets ist ein Descriptor Set Handle, das mit dem erzeugten Handle überschrieben wird.

DescriptorPool ist das Handle des Descriptor Pools. DescriptorSetCount ist die Anzahl der Sets, die erzeugt werden soll, und in pSetLayouts werden entsprechend viele DescriptorSet Layout Handles übergeben.

```
        VkDescriptorBufferInfo descriptorBufferInfo;
        descriptorBufferInfo.buffer = storageBuffer[i];
        descriptorBufferInfo.offset = 0;
        descriptorBufferInfo.range = bufferSize[i];


        VkWriteDescriptorSet descriptorWrite;
        descriptorWrite.sType =
VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
        descriptorWrite.pNext = nullptr;
        descriptorWrite.dstSet = descriptorSet[0];
        descriptorWrite.dstBinding = i;
        descriptorWrite.dstArrayElement = 0;
        descriptorWrite.descriptorCount = 1;
        descriptorWrite.descriptorType =
VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
        descriptorWrite.pImageInfo = nullptr;
        descriptorWrite.pBufferInfo =&descriptorBufferInfo;
        descriptorWrite.pTexelBufferView = nullptr;
        vkUpdateDescriptorSets(device, 1, &descriptorWrite, 0,
nullptr);
```

Nachdem das Descriptorset allokiert wurde, besitzt es noch keine Informationen über die genauen Daten, die die Descriptoren enthalten sollen.

Das Handle des Sets, für das die Update-Funktion ausgeführt werden soll, wird in dstSet angegeben, das gewünsche Binding und der dazugehörige Array Index werden jeweils in dstBinding und dstArrayElement angegeben. DescriptorCount gibt die Zahl der Descriptoren an, für die das Update ausgeführt werden soll. DescriptorCount kann die Zahl der Descriptoren überschreiten, die ausgehend von dstArrayElement noch für das Binding vorhanden sind. In diesem Fall wird mit dem nächsten Binding (dstBinding+1) und Index 0 weitergearbeitet. Die letzen drei Parameter sind Arrays, die Informationen über die Descriptoren enthalten. Es kann immer nur jeweils einer der Parameter pImageInfo, pBufferInfo und pTexelBufferView verwendet werden. Welcher verwendet wird ist abhängig davon, welcher Descriptortyp in der Komponente descriptorType angegeben wird. Die anderen beiden Parameter sind nullptr. Für den Raytracer werden ausschließlich Storage- und Uniformbuffer verwendet, in beiden Fällen wird pBufferInfo benutzt.

pBufferInfo ist ein Array aus VkDescriptorBufferInfo Strukturen. Die Komponente Buffer ist ein VkBuffer Handle. Offset ist ein Offset in Byte ausgehend vom Anfang des Buffers, während Range die Länge in Byte ausgehend von Offset angibt. Damit ist es beispielsweise möglich einen Buffer für mehrere Descriptoren zu verwenden.

2.2.8 Buffer

Buffer sind in Vulkan Objekte, die verwendet werden, um Daten zu speichern. Im Raytracerprogramm werden Buffer verwendet, um die Vertices für das Screen Filling Quad und die Daten der Raytracing Szene zu speichern. Später werden die Buffer entsprechend ihres Inhalts an die Graphikpipeline gebunden. Ein Buffer Handle kann mit folgender Funktion erstellt werden.

```
	vkCreateBuffer(device, &bufferCreateInfo, nullptr,
&buffer);
```

device ist das Device das den Buffer erstellen soll. bufferCreateInfo ist ein Pointer auf eine VkBufferCreateInfo Struktur. Der dritte Parameter ist wie immer der ungenutzte pAllocator und buffer ist ein Pointer auf ein Buffer Handle. Die VkBufferCreateInfo Struktur sieht wie folgt aus.

```
	VkBufferCreateInfo bufferCreateInfo;
	bufferCreateInfo.sType =
VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
	bufferCreateInfo.pNext = nullptr;
	bufferCreateInfo.flags = 0;
	bufferCreateInfo.size = size; //VkDeviceSize
	bufferCreateInfo.usage = usageFlags;
	bufferCreateInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
	bufferCreateInfo.queueFamilyIndexCount = 0;
	bufferCreateInfo.pQueueFamilyIndices = nullptr;
```

Die Komponente size gibt die Größe des Buffers in Byte an. Usage ist ein Bit field, das aus Elementen der VkBufferUsageFlagsBit Enumeration besteht. Diese Bits geben an, wofür der Buffer verwendet wird. Im Raytracer werden jeweils die Bits für Vertex, Storage und Uniform Buffer verwendet. SharingMode gibt an, ob der Buffer ausschließlich von einem oder mehreren Command Buffern gleichzeitig verwendet werden soll. Der Raytracer hat nur einen Command Buffer, der später erklärt wird, weshalb Sharing Mode Exclusive verwendet wird. Die letzten beiden Parameter sind nur dann wichtig, wenn der Sharing Mode auf Concurrent gesetzt wird. In diesem Fall geben die beiden Komponenten an wie viele und welche Queue Families Zugriff auf den Buffer haben sollen.

Damit der Buffer benutzt werden kann, muss zuerst Speicher dafür allokiert werden. Dieser Speicher ist Device Memory, also Speicher auf den das Device Zugriff hat. Der Speicher kann mit folgender Funktion allokiert werden.

```
	vkAllocateMemory(device, &memoryAllocateInfo, nullptr,
&deviceMemory);
```

device is das Device, das den Speicher später nutzen soll. memoryAllocateInfo ist ein Pointer auf eine VkMemoryAllocateInfo Struktur, der dritte Parameter ist wieder pAllocator und die letzte Komponente ist ein VkDeviceMemory Handle. Die Info Struktur sieht wie folgt aus.

```
	VkMemoryAllocateInfo memoryAllocateInfo;
	memoryAllocateInfo.sType = 
VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
	memoryAllocateInfo.pNext = nullptr;
	memoryAllocateInfo.allocationSize = memRequirements.size;
	memoryAllocateInfo.memoryTypeIndex = 
findMemoryType(memRequirements.memoryTypeBits, 
memoryPropertyFlags);
```

AllocationSize ist die Größe des Speichers, der allokiert werden soll, in Byte. memoryTypeIndex ist der Index des benötigten Speichertyps im memoryType Array. Das Array wird von folgendem Aufruf zurückgegeben.

```
	VkPhysicalDeviceMemoryProperties memoryProperties;

vkGetPhysicalDeviceMemoryProperties(physicalDevice[physicalDevi
ceIndex], &memoryProperties);
```

memoryProperties ist eine Struktur, deren Komponente memoryType das Array enthält. Für den benötigten Speichertyp wir folgender Aufruf verwendet.

```
	VkMemoryRequirements memoryRequirements;
	vkGetBufferMemoryRequirements(device, buffer, 
&memoryRequirements);
```

Um den richtigen Index zu ermitteln, muss der Index des Arrays gefunden werden, der mit den memoryTypeBits des Buffers übereinstimmt.

Anschließend muss der Speicher an den Buffer gebunden werden.

```
	vkBindBufferMemory(device, buffer, deviceMemory, 0);
```

Device ist auch hier das Device. Buffer ist der Buffer, der mit dem Speicher assoziiert werden soll. DeviceMemory ist das Handle des Speichers. Der letzte Parameter ist der Offset. DeviceMemory kann größer sein als der Buffer, der damit verbunden wird. Darüber hinaus ist es möglich mehrere Ressourcen mit dem selben DeviceMemory zu verbinden, was auch empfohlen wird. Offset kann dabei benutzt werden, um Überschneidungen von Ressourcen zu vermeiden. Überschneidungen können aber auch genutzt werden, um unter bestimmten Bedingungen Speicherplatz zu sparen.

Nachdem Speicher allokiert wurde, kann dieser mit beliebigen Daten gefüllt werden. Es bleibt meistens dem Programmierer überlassen dafür zu sorgen, dass der Inhalt später richtig interpretiert wird. Der Speicher wird bei dem Raytracer auf folgende Art befüllt.

```
void* ppData;
vkMapMemory(device, deviceMemory, 0, size, 0, &ppData);
memcpy(ppData, data, size);
vkUnmapMemory(device, deviceMemory);
```

vkMapMemory() mappt Device Memory in den Adressraum des Hosts. Device ist das Device, das den Speicher besitzt. deviceMemory ist das Handle des Speichers. Die dritte Komponente ist der Offset vom Start des Speichers in Byte. Size gibt die Größe des Speichers in Byte an. Offset + Size sollte nicht größer sein als die gesamte Größe des Speichers. Flags ist für eine mögliche zukünftige Nutzung reserviert und wird deshalb auf 0 gesetzt. ppData ist der Pointer auf den Speicher im Addressraum des Hosts.

Memcpy ist keine Vulkan Funktion, kann aber benutzt werden um Daten in den Speicher zu schreiben. Data ist ein Pointer auf die Daten, die in den Device Speicher geschrieben werden sollen.

Nachdem der Pointer auf den Speicher nicht mehr benötigt wird, wird vkUnmapMemory() aufgerufen. Dadurch wird der Pointer unbrauchbar.

2.2.9 Renderpass

Für komplexe Graphikanwendungen werden oft mehrere Renderingschritte benötigt, von denen jeder einen Teil des finalen Bildes rendert.

Ein Renderpass Objekt kann mithilfe der Funktion vkCreateRenderPass() erzeugt werden.

```
vkCreateRenderPass(device, &renderPassCreateInfo,
nullptr, &renderPass);
```

Das Device ist das zuvor erstellte Device Handle, &renderPassCreateInfo ist ein Pointer auf eine VkRenderPassCreateInfo Struktur, anstelle eines Allocators wird wieder NULL übergeben und renderPass ist ein VkRenderPass Handle.

```
      VkRenderPassCreateInfo renderPassCreateInfo = {};
      renderPassCreateInfo.sType =
VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
      renderPassCreateInfo.pNext = nullptr;
      renderPassCreateInfo.flags = 0;
      renderPassCreateInfo.attachmentCount = 1;
      renderPassCreateInfo.pAttachments =
&attachmentDescription;
      renderPassCreateInfo.subpassCount = 1;
      renderPassCreateInfo.pSubpasses = &subpassDescription;
      renderPassCreateInfo.dependencyCount = 1;
      renderPassCreateInfo.pDependencies = &subpassDependency;
```

VkRenderPassCreateInfo enthält die Informationen, die den Renderpass definieren. Die Komponente pAttachments ist ein VkAttachmentDescription Array der Länge attachmentCount. pSubpasses und pDependencies funktionieren analog.

Die Pipeline für den Raytracer benötigt ausschließlich ein Colorattachment. Die verwendete Struktur sieht wie folgt aus.

```
      VkAttachmentDescription attachmentDescription = {};
      attachmentDescription.flags = 0;
      attachmentDescription.format = surfaceFormat;
      attachmentDescription.samples = VK_SAMPLE_COUNT_1_BIT;
      attachmentDescription.loadOp =
VK_ATTACHMENT_LOAD_OP_CLEAR;
      attachmentDescription.storeOp =
VK_ATTACHMENT_STORE_OP_STORE;
      attachmentDescription.stencilLoadOp =
VK_ATTACHMENT_LOAD_OP_DONT_CARE;
      attachmentDescription.stencilStoreOp =
VK_ATTACHMENT_STORE_OP_DONT_CARE;
      attachmentDescription.initialLayout =
VK_IMAGE_LAYOUT_UNDEFINED;
      attachmentDescription.finalLayout =
VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
```

VkAttachmentDescription enthält Informationen über ein einziges Attachment. Ein Attachment entspricht einem Bild, das in der Pipeline genutzt werden kann, um Daten darauf zu speichern oder davon zu lesen. Die erste Komponente format ist ein Element der Enumeration VkFormat. Dieses Format sollte mit dem Format des Bildes übereinstimmen, das als Attachment verwendet wird. Dabei handelt es sich bei dem Raytracer um die Bilder der Swapchain, die als Farb Attachment benutzt werden, weshalb hier dasselbe

Format verwendet wird. Samples gibt die Anzahl der Samples im Bild an und wird für Multisampling benutzt. Wird wie beim Raytracer kein Multisampling verwendet, ist der Standardwert VK_SAMPLE_COUNT_1_BIT. Die beiden Komponenten loadOp and storeOp geben an, wie mit dem Attachment vor und nach einem Renderpass umgegangen werden soll. LoadOp gibt an wie das Attachment geladen wird. Da der Raytracer keine bereits vorhandenen Farbinformationen benötigt und das gesamte Bild mit den Renderingergebnissen überschrieben wird, kann das Bild mit der Operation DONT_CARE geladen werden. Das bedeutet, dass der Inhalt des Attachments zu Beginn des Renderpasses unwichtig ist. StoreOp gibt an ob die Ergebnisse für eine spätere Nutzung gespeichert werden sollen. Da es sich hier um das Farbattachment handelt, das später auf dem Bildschirm angezeigt werden soll, ist dieser Wert VK_ATTACHMENT_STORE_OP_STORE, was bedeutet, dass die Ergebnisse gespeichert werden. Die Komponenten stencilLoadOp und stencilStoreOp werden Analog verwendet, um die Lade- und Speicheroperationen für den Stencilbuffer festzulegen. Das ist dann wichtig, wenn es sich bei dem Attachment um ein Depth-Stencil Attachment handelt. Wenn das der Fall ist, werden load- und storeOp für das Depth-Attachment genutzt. Da kein Stencil Attachment verwendet wird, werden beide Parameter von Vulkan ignoriert.

Die Komponenten initialLayout und finalLayout geben an in welchem Layout das Attachment am Anfang des Renderpasses zu erwarten ist und in welchem Layout das Attachment sein soll, wenn der Renderpass endet. Vulkan ändert das initiale Layout nicht automatisch, stattdessen sollte das Layout angegeben werden, in dem sich das Bild bereits befindet.

2.2.10 Framebuffer

Nachdem der Renderpass erstellt wurde kann der dazugehörige Framebuffer erzeugt werden. Der Framebuffer ist ein Set aus Attachments die von dem Renderpass verwendet werden.

```
vkCreateFramebuffer(device, &framebufferCreateInfo, nullptr,
&framebuffers[i]);
```

device ist das Device, framebufferCreateInfo ist die nachfolgend erklärte Struktur. Der dritte Parameter ist der optionale pAllocator. Framebuffers ist ein Framebuffer Handle.

```
      VkFramebufferCreateInfo  framebufferCreateInfo = {};
      framebufferCreateInfo.sType =
VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
      framebufferCreateInfo.pNext = nullptr;
      framebufferCreateInfo.flags = 0;
```

```
framebufferCreateInfo.renderPass = renderPass;
framebufferCreateInfo.attachmentCount = 1;
framebufferCreateInfo.pAttachments = &(imageViews[i]);
framebufferCreateInfo.width = WIDTH;
framebufferCreateInfo.height = HEIGHT;
framebufferCreateInfo.layers = 1;
```

Die Komponente renderPass ist der zuvor erstellte Renderpass. pAttachments ist ein Pointer auf ein Array der Länge attachmentCount, das die zuvor erstellten VkImageViews enthält, die die Bilder referenzieren auf die gezeichnet werden soll. Die Dimensionen des Framebuffers werden in width, height und layers angegeben. Da auf das gesamte Fenster gezeichnet werden soll, entsprechen diese Dimensionen bei dem Raytracer der Größe des Fensters.

Im Raytracerprogramm wird für jedes Bild der Swapchain ein Framebuffer erstellt.

2.2.11 pSubpasses

VkSubpassDescription definiert einen Subpass. Für den Raytracer wird nur ein Subpass benötigt.

```
VkSubpassDescription subpassDescription = {};
subpassDescription.flags = 0;
subpassDescription.pipelineBindPoint =
VK_PIPELINE_BIND_POINT_GRAPHICS;
subpassDescription.inputAttachmentCount = 0;
subpassDescription.pInputAttachments = nullptr;
subpassDescription.colorAttachmentCount = 1;
subpassDescription.pColorAttachments =
&attachmentReference;
subpassDescription.pResolveAttachments = nullptr;
subpassDescription.pDepthStencilAttachment = nullptr;
subpassDescription.preserveAttachmentCount = 0;
subpassDescription.pPreserveAttachments = nullptr;
```

Die erste Komponente pipelineBindPoint gibt an ob der Subpass für eine Compute- oder Graphikpipeline definiert werden soll. In der Vulkan Version 1.0.71 werden Subpasses nur für die Graphikpipeline unterstützt. Die anderen Komponenten geben die verwendeten Attachments an.

pInputAttachments ist ein VkAttachmentReference Array der Länge inputAttachmentCount. Darin wird angegeben, von welchen Attachments während der Fragmentshader Stage gelesen werden kann. Wenn ein Attachment vom Shader gelesen werden soll, muss das entsprechende Attachment zusätzlich über Descriptorsets als Shaderressource an die Pipeline gebunden werden. Für den Raytracer werden keine Input Attachments benötigt.

pColorAttachments ist ein VkAttachmentReference Array der Länge colorAttachmentCount. Das Array spezifiziert, welche der Attachments des Renderpasses während des Subpasses als ColorAttachments benutzt werden sollen.

```
	VkAttachmentReference attachmentReference = {};
	attachmentReference.attachment = 0;
	attachmentReference.layout = 
VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
```

VkAttachmentReference wird für pColorAttachments, pResolveAttachments und pDepthStencilAttachment verwendet. Die Komponente attachment ist der Index des Attachment, für das die Referenz erstellt werden soll. Der Index entspricht dabei dem Index des Attachment in der Komponente pAttachments der VkRenderpassCreateInfo Struktur. Da es bei dem Raytracer nur ein Color Attachment gibt, ist der Index 0. layout gibt an, welches Layout das Attachment während des Subpasses haben soll. Für das einzige Attachment des Raytracers wird das Layout für Farbattachments benötigt.

pResolveAttachments ist ein optionales Array der Größe colorAttachmentCount. Das Array gibt die Farbattachments an, für die multisample resolve Operationen durgeführt werden sollen. Da der Raytracer kein Multisampling verwendet, ist die Komponente NULL.

pDepthStencilAttachment ist analog die Referenz zu dem depth-stencil Attachment. Im Gegensatz zu den anderen Komponenten kann es nur ein Attachment dieser Art in einem Subpass geben, weshalb keine Anzahl angegeben werden muss. Wie schon erwähnt hat die Pipeline des Raytracers kein depth-stencil Attachment.

Die letzte Komponente pPreserveAttachments ist ein Array der Länge preserveAttachmentCount, das die Indices der Attachments enthält, die nicht in dem Subpass verwendet werden, deren Inhalt aber während des Subpasses unverändert bleiben soll. Die Pipeline des Raytracers hat nur einen Subpass, weshalb auch diese Komponente ungenutzt bleibt.

Die letzte Struktur, die zur Erstellung des Renderpasses benötigt wird, ist VkSubpassDependency. Diese Struktur enthält Informationen über die Abhängigkeiten der Subpasses untereinander.

```
	VkSubpassDependency subpassDependency = {};
	subpassDependency.srcSubpass = VK_SUBPASS_EXTERNAL;
	subpassDependency.dstSubpass = 0;
	subpassDependency.srcStageMask = 
VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
	subpassDependency.dstStageMask = 
VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
	subpassDependency.srcAccessMask = 0;
	subpassDependency.dstAccessMask = 
VK_ACCESS_COLOR_ATTACHMENT_READ_BIT | 
VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
	subpassDependency.dependencyFlags = 0;
```

Die Komponenten srcSubpass und dst Subpass geben jeweils an, welcher Subpass Daten produziert, und welcher Subpass diese konsumiert. Beide Parameter werden jeweils auf einen Index des Arrays des Subpasses vom Renderpass gesetzt.

2.2.12 Pipeline Layout

Das Pipeline Layout wird benutzt um der Pipeline Zugriff auf die Ressourcen zu bieten, die durch Descriptoren und Push Konstanten bereitgestellt werden. Ein Pipeline Layout wird mit folgendem Aufruf erstellt.

```
	vkCreatePipelineLayout(device, &layoutInfo, nullptr, 
&pipelineLayout);
```

Die Parameter funktionieren wie bei den anderen Funktionen dieser Art auch.

```
	VkPipelineLayoutCreateInfo layoutInfo = {};
	layoutInfo.sType = 
VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
	layoutInfo.pNext = nullptr;
	layoutInfo.flags = 0;
	layoutInfo.setLayoutCount = 1; //1 buffer layout
	layoutInfo.pSetLayouts = &descriptorSetLayout[0];
	layoutInfo.pushConstantRangeCount = 0;
	layoutInfo.pPushConstantRanges = nullptr;
```

pSetLayouts ist ein Array der Länge setLayoutCount, das die Descriptorsets enthält, die von der Pipeline verwendet werden sollen. Analog ist pPushConstanRanges ein Array der Länge pushConstantRangeCount, das Push Constant Ranges enthält. Da in dem Raytracerprogramm keine Push Constant Ranges verwendet werden, ist die Komponente nullptr.

2.2.13 Pipeline

Graphikpipelines folgen in Vulkan einem ähnlichen Modell wie in OpenGL. Es gibt die gleichen Shader Stages, Vertex-, Tesselation Control- und Evaluation-, Geometry- und Fragment Shader, von denen bis auf den Vertexshader alle optional sind. Darüber hinaus erlaubt Vulkan die Konfiguration der Fixed-Function Stages.

```
	VkGraphicsPipelineCreateInfo pipelineCreateInfo = {};
	pipelineCreateInfo.sType = 
VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
	pipelineCreateInfo.pNext = nullptr;
	pipelineCreateInfo.flags = 0;
	pipelineCreateInfo.stageCount = 2;
	pipelineCreateInfo.pStages = shaderStages;
	pipelineCreateInfo.pVertexInputState = 
&vertexInputCreateInfo;
	pipelineCreateInfo.pInputAssemblyState = 
&inputAssemblyCreateInfo;
	pipelineCreateInfo.pTessellationState = nullptr;
	pipelineCreateInfo.pViewportState = 
&viewportStateCreateInfo;
	pipelineCreateInfo.pRasterizationState = 
&rasterizationCreateInfo;
	pipelineCreateInfo.pMultisampleState = 
&multisampleCreateInfo;
	pipelineCreateInfo.pDepthStencilState = nullptr;
	pipelineCreateInfo.pColorBlendState = 
&colorBlendCreateInfo;
	pipelineCreateInfo.pDynamicState = nullptr;
	pipelineCreateInfo.layout = pipelineLayout;
	pipelineCreateInfo.renderPass = renderPass;
	pipelineCreateInfo.subpass = 0;//index
    pipelineCreateInfo.basePipelineHandle = VK_NULL_HANDLE;
    pipelineCreateInfo.basePipelineIndex = -1;//nicht benutzt
```

basePipelineHandle und basePipelineIndex: Die beiden Komponenten werden benötigt, wenn eine abgeleitete Pipeline erzeugt werden soll. Ein Vorteil davon ist, dass das Erzeugen solcher Pipelines gegebenenfalls günstiger ist, da die

alte Pipeline als Startpunkt genutzt werden kann. Dabei wird davon ausgegangen, das beide Pipelines sich sehr ähnlich sind. Die Komponente basePipelineHandle wird dann verwendet, wenn die alte Pipeline bereits erzeugt wurde, während basePipelineIndex verwendet wird, wenn beide Pipelines gleichzeitig erzeugt werden. Zur Nutzung muss das Flag VK_PIPELINE_CREATE_DERIVATIVE_BIT in der Komponente flags gesetzt werden. Für den Raytracer wird nur eine Pipeline benötigt, weshalb beide Komponenten auf die von der Vulkan Spezifikation definierten, ungültigen Werte gesetzt werden.

StageCount, pStages: Die Komponente stageCount gibt an, wie viele Shader Stages die Pipeline verwenden soll, diese Zahl sollte der Zahl der Elemente im pStages Array entsprechen. Für eine Graphikpipeline wird mindestens ein Vertexshader benötigt und es kann in der Graphikpipeline kein Computeshader verwendet werden, da Compute- und Graphikpipelines in Vulkan verschiedene Objekte sind. Die Struktur für die Definition von Shader Stages sieht folgendermaßen aus:

```
	VkPipelineShaderStageCreateInfo shaderStageCreateInfoVert = {};
	shaderStageCreateInfoVert.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
	shaderStageCreateInfoVert.pNext = nullptr;
	shaderStageCreateInfoVert.flags = 0;
	shaderStageCreateInfoVert.stage = VK_SHADER_STAGE_VERTEX_BIT;
	shaderStageCreateInfoVert.module = shaderModuleVert;
	shaderStageCreateInfoVert.pName = "main";
	shaderStageCreateInfoVert.pSpecializationInfo = nullptr;
```

Wie bei allen anderen Strukturen dieser Art ist sType der Typ der Struktur und pNext ein Pointer auf eine Extension-spezifische Struktur oder NULL. Es sind in der verwendeten Vulkan Version keine Flags für diese Struktur verfügbar. Die Komponente stage ist ein VkShaderStageFlagBits Wert, der jeweils einer einzelnen Shader Stage entspricht. Der Raytracer verwendet einen Vertex- und einen Fragmentshader, die entsprechenden Makros sind VK_SHADER_STAGE_VERTEX_BIT und VK_SHADER_STAGE_FRAGMENT_BIT. Der Vertexshader ist für das Screen Filling Quad zuständig, und der Fragmentshader ist der eigentliche Raytracer. Die Komponente module ist ein VkShaderModule Objekt, das den Shader für die Shader Stage bereitstellt, das Objekt wird später erklärt. Die Komponente pName ist ein String, der den Namen der Funktion festlegt, die beim Eintritt in den Shader aufgerufen wird. pSpecialisationInfo kann genutzt werden, um Konstanten im SPIR-V Code zur Anwendungslaufzeit zu definieren, genauer gesagt zu dem Zeitpunkt, an dem die Pipeline erzeugt wird. Ist der Parameter nullptr bedeutet das, dass keine Spezialisation Constants benutzt werden.

Ein VkShaderModule Objekt kann erzeugt werden, indem folgende Funktion aufgerufen wird.

```
vkCreateShaderModule(device, &shaderCreateInfo, nullptr,
shaderModule)
```

```
	VkShaderModuleCreateInfo shaderCreateInfo = {};
	shaderCreateInfo.sType =
VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
	shaderCreateInfo.pNext = nullptr;
	shaderCreateInfo.flags = 0;
	shaderCreateInfo.codeSize = code.size();
	shaderCreateInfo.pCode = (uint32_t*)code.data();
```

Die Komponente codeSize gibt die Größe des Shadercodes in Byte an, während pCode auf die Speicheradresse des SPIR-V Codes zeigt, der von dem Shader verwendet werden soll. Der Code muss den Anforderungen der SPIR-V Spezifikation entsprechen. Für den Raytracer wird jeweils ein Shadermodul für Vertex- und Fragmentshader erzeugt.

2.2.13.1 Vertex Input State

Es gibt wie bereits erwähnt keine speziellen Vertex Buffer. Stattdessen kann jeder Buffer, für den das VK_BUFFER_USAGE_VERTEX_BUFFER_BIT bei der Erzeugung gesetzt war benutzt werden, um Vertex Daten zu speichern. Um einen Vertexbuffer zu nutzen, muss dieser später mithilfe eines Command Buffers als solcher an den Renderpass gebunden werden.

```
	VkPipelineVertexInputStateCreateInfo
vertexInputStateCreateInfo = {};
	vertexInputStateCreateInfo.sType =
VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
	vertexInputStateCreateInfo.pNext = nullptr;
	vertexInputStateCreateInfo.flags = 0;
	vertexInputStateCreateInfo.vertexBindingDescriptionCount
= 1;
	vertexInputStateCreateInfo.pVertexBindingDescriptions =
&vertexBindingDescriptions;

vertexInputStateCreateInfo.vertexAttributeDescriptionCount = 1;
	vertexInputStateCreateInfo.pVertexAttributeDescriptions =
&vertexAttributeDescriptions;
```

pVertexBindingDescriptions ist ein Pointer auf ein VkVertexInputBindingDescription Array der Länge vertexBindDescriptionCount. Jedes Element des Arrays definiert ein Vertex Binding , auf das später im Vertexshader zugegriffen werden kann. Die Struktur wurde für den Raytracer folgendermaßen gefüllt.

```
	VkVertexInputBindingDescription bindingDescription;
	bindingDescription.binding = 0;
	bindingDescription.stride = sizeof(Vertex);
	bindingDescription.inputRate = 
VK_VERTEX_INPUT_RATE_VERTEX;
```

Die Komponente binding ist der Index des Bindings, auf das der Vertexshader später zugreifen kann. Vulkan garantiert, dass mindestens 16 Vertex Buffer mit eigenem Binding gebunden werden können. Ein Device kann aber auch mehr unterstützen. Jedes dieser Bindings entspricht einem Buffer, der ein Array aus Strukturen enthält. Jede Struktur beschreibt die Eigenschaften eines einzelnen Vertex, wie Position oder Farbe. Der Programmierer ist für die Erstellung und den Inhalt dieser Struktur selbst zuständig. Die Struktur, die für die Vertices des Screen Filling Quads verwendet, wird beschreibt ausschließlich deren 2D Koordinaten und sieht folgendermaßen aus:

```
	struct Vertex{
		glm::vec2 pos;
	};
```

stride ist die Distanz zwischen dem Anfang jeder Struktur in Byte. Die Daten des Buffers müssen auf der Programmseite nicht als Struktur definiert sein, solange die Daten als solche interpretiert werden können. Da der Raytracer die Struktur Vertex verwendet, entspricht Stride der Größe der Struktur in Byte.

InputRate wird verwendet, um

pVertexAttributeDescriptions ist ein Array der Länge vertexAttributeDescriptionCount.

```
	VkVertexInputAttributeDescription attributeDescriptions = 
{};
	attributeDescriptions.location = 0;
	attributeDescriptions.binding = 0;
	attributeDescriptions.format = VK_FORMAT_R32G32_SFLOAT;
	attributeDescriptions.offset = offsetof(Vertex, pos);
```

location wird im Shader benutzt, um auf ein einzelnes Attribut des Vertex zuzugreifen. Der Raytracer hat nur die Position als Attribut, weshalb location 0 ist. Vulkan unterstützt die Nutzung von mindestens 16 Attributen pro Shader. Im Shader wird mit folgender Zeile auf die Daten zugegriffen.

```
layout(location = 0) in vec2 inPosition;
```

binding sollte einem zuvor in pVertexBindingDescription definierten Binding entsprechen, für das die Attribute Description verwendet werden soll. Im Raytracer wird nur Binding 0 spezifiziert, weshalb auch hier 0 angegeben wird.

Format gibt das Format des Attributs an. Da ein vec2 Objekt verwendet wird ist das Format VK_FORMAT_R32G32_SFLOAT.

Offset ist das Offset innerhalb der Struktur. Da es nur eine Komponente in der Vertex Struktur des Raytracers gibt, ist das Offset 0.

2.2.13.2 Input Assembly State

In der Input Assembly Phase werden die Vertices zu Primitiven zusammengesetzt.

```
VkPipelineInputAssemblyStateCreateInfo inputAssemblyCreateInfo = {};
inputAssemblyCreateInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
inputAssemblyCreateInfo.pNext = nullptr;
inputAssemblyCreateInfo.flags = 0;
inputAssemblyCreateInfo.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP;
inputAssemblyCreateInfo.primitiveRestartEnable = VK_FALSE;
```

Topology ist ein Element der VkPrimitiveTopology Enumeration. Die Topologie bestimmt wie in OpenGL wie die Vertexdaten zu Primitiven verbunden werden. Der Raytracer benutzt Triangle Strip. PrimitiveRestartEnable kann verwendet werden, um es Strip und Fan Topologien zu erlauben in einem einzigen Draw Aufruf neu gestartet zu werden. Nach einem Neustart beziehen sich die nachfolgenden Vertices nicht mehr auf die schon abgearbeiteten Vertices. Diese Option kann allerdings nur genutzt werden, wenn Indexed Draws verwendet werden. Da das bei dem Raytracer nicht der Fall ist und auch kein Neustart benötigt wird, wird Primitive Restart mit VK_FALSE deaktiviert.

2.2.13.3 Viewport State

In der Viewport Phase werden die Koordinaten der Vertices von normalisierten Device Koordinaten in Fensterkoordinaten umgewandelt.

```
	VkPipelineViewportStateCreateInfo viewportStateCreateInfo
= {};
	viewportStateCreateInfo.sType =
VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
	viewportStateCreateInfo.pNext = nullptr;
	viewportStateCreateInfo.flags = 0;
	viewportStateCreateInfo.viewportCount = 1;
	viewportStateCreateInfo.pViewports = &viewport;
	viewportStateCreateInfo.scissorCount = 1;
	viewportStateCreateInfo.pScissors = &scissor;
```

viewportCount definiert die Anzahl der Viewports, die in der Pipeline verwendet werden. pViewport ist ein Pointer auf ein VkViewport Array von der Länge viewportCount. Analog gibt scissorCount die Anzahl der Elemente im Array pScissors. Dieses Array besteht aus VkRect2D Strukturen. Scissors bestimmen den Teil des Fensters, der für das Rendering relevant ist, indem dieser Bereich aus dem Fenster ausgeschnitten wird. Für den Raytracer soll das gesamte Fenster verwendet werden, weshalb die Größe des Scissor Rechtecks auf die Größe des Fensters gesetzt wird, wodurch Scissor indirekt deaktiviert wird.

```
	VkViewport viewport = {};
	viewport.x = 0.0f;
	viewport.y = 0.0f;
	viewport.width = WIDTH;
	viewport.height = HEIGHT;
	viewport.minDepth = 0.0f;
	viewport.maxDepth = 1.0f;

VkRect2D scissor = {};
	scissor.offset = {0, 0};
	scissor.extent = {WIDTH, HEIGHT};
```

VkRect2D definiert ein Rechteck in einem Bild oder Framebuffer. Offset beschreibt dabei die position der linken oberen Ecke des Rechtecks und extent die Größe in x- und y-Richtung. Beides wird in der Zahl von Pixeln angegeben. Für den Raytracer wird VkRect2D verwendet, um Scissor für die

Viewport Stage zu definieren. VkOffset2D ist auch eine Struktur, die wie folgt aussieht.

VkOffset2D{
    int32_t x;
    int32_t y;
}

2.2.13.4 Rasterization State

Während der Rasterisierung werden die Primitive, die durch Vertices repräsentiert werden, in Fragmente umgewandelt. Die dazugehörige Struktur legt fest wie dabei vogegangen werden soll.

```
    VkPipelineRasterizationStateCreateInfo
rasterizationStateCreateInfo = {};
    rasterizationStateCreateInfo.sType =
VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
    rasterizationStateCreateInfo.pNext = nullptr;
    rasterizationStateCreateInfo.flags = 0;
    rasterizationStateCreateInfo.depthClampEnable = VK_FALSE;
    rasterizationStateCreateInfo.rasterizerDiscardEnable =
VK_FALSE;
    rasterizationStateCreateInfo.polygonMode =
VK_POLYGON_MODE_FILL;
    rasterizationStateCreateInfo.cullMode =
VK_CULL_MODE_BACK_BIT;
    rasterizationStateCreateInfo.frontFace =
VK_FRONT_FACE_CLOCKWISE;
    rasterizationStateCreateInfo.depthBiasEnable = VK_FALSE;
    rasterizationStateCreateInfo.depthBiasConstantFactor =
0.0f;
    rasterizationStateCreateInfo.depthBiasClamp = 0.0f;
    rasterizationStateCreateInfo.depthBiasSlopeFactor = 0.0f;
    rasterizationStateCreateInfo.lineWidth = 1.0f;
```

VkPipelineRasterisationStateCreateInfo enthält die Parameter für die Rasterisierung.

DepthClampEnable bestimmt, ob Fragmente, die durch Near- oder Far Clipping wegfallen würden, stattdessen auf die Near- oder Far Clipping Ebene projiziert werden. Depth Clamp wird für den Raytracer nicht benötigt und ist deshalb deaktiviert. RasterizerDiscardEnable kann benutzt werden, um die Rasterisierung komplett auszuschalten. Das macht dann Sinn, wenn keine Fragmente benötigt werden.

PolygonMode bestimmt wie die Primitive gezeichnet werden sollen. Dabei sind Punkte, Linien und Flächen verfügbar. Da der Raytracer ein Screen Filling Quad benötigt, müssen die Primitive als Flächen gezeichnet werden. cullMode wird verwendet, um festzulegen ob und welche Primitive wegfallen sollen. Dabei stehen Front-, Backface oder beides zur Auswahl. Sollen keine Primitive wegfallen ist der Wert 0. Front Primitive zeigen zur Kamera, Back Primitive zeigen davon weg. In der Komponente Frontface wird festgelegt welche Primitive in welche Richtung zeigen. Bei dem Raytracer zeigen die Primitive nach vorne, deren Vertices im Uhrzeigersinn angeordnet sind.

Die DepthBias* Komponenten können benutzt werden, um das DepthBias Feature zu kontrollieren. Dadurch können Fragmente vor dem Tiefentest ein Tiefenoffset bekommen. Das soll dafür sorgen, dass Fragmente nicht auf dieselbe Tiefe fallen. Diese Funktion wird für den Raytracer nicht benötigt und ist deshalb deaktiviert.

LineWidth gibt die breite der Linien an, die gezeichnet werden, wenn polygonMode VK_POLYGON_MODE_LINE ist. Wenn nicht explizit etwas anderes benötigt wird, ist der Standardwert 1.0.

2.2.13.5 Multisample State

Die Multisampling wird für das Antialiasing von Primitiven verwendet.

```
	VkPipelineMultisampleStateCreateInfo multisampleCreateInfo = {};
	multisampleCreateInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
	multisampleCreateInfo.pNext = nullptr;
	multisampleCreateInfo.flags = 0;
	multisampleCreateInfo.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;
	multisampleCreateInfo.sampleShadingEnable = VK_FALSE;
	multisampleCreateInfo.minSampleShading = 1.0f;
	multisampleCreateInfo.pSampleMask = nullptr;
	multisampleCreateInfo.alphaToCoverageEnable = VK_FALSE;
	multisampleCreateInfo.alphaToOneEnable = VK_FALSE;
```

rasterizationSamples gibt an, wie viele Samples pro Pixel verwendet werden. Da Multisampling für den Raytracer nicht benötigt wird sind sämtliche Multisample Funktionen entweder deaktiviert, oderauf einem neutralen Wert eingestellt. Trotzdem ist die Struktur notwendig, wenn Rasterisierung verwendet wird.

2.2.13.6 Color Blend State

In der Color Blend Phase werden Fragmente in die Color Attachments geschrieben.

```
	VkPipelineColorBlendStateCreateInfo colorBlendCreateInfo
= {};
	colorBlendCreateInfo.sType =
VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
	colorBlendCreateInfo.pNext = nullptr;
	colorBlendCreateInfo.flags = 0;
	colorBlendCreateInfo.logicOpEnable = VK_FALSE;
	colorBlendCreateInfo.logicOp = VK_LOGIC_OP_NO_OP;
	colorBlendCreateInfo.attachmentCount = 1;
	colorBlendCreateInfo.pAttachments =
&colorBlendAttachmentState;
//	colorBlendCreateInfo.blendConstants[0] = 0.0f;
//	colorBlendCreateInfo.blendConstants[1] = 0.0f;
//	colorBlendCreateInfo.blendConstants[2] = 0.0f;
//	colorBlendCreateInfo.blendConstants[3] = 0.0f;
```

logicOpEnable legt fest, ob logische Operationen zwischen den neuen Werten aus dem Fragmentshader und dem Inhalt der Attachments ausgeführt werden sollen. LogicOp gibt an, welche Operation verwendet werden soll. Da die Ergebnisse des Fragmentshaders des Raytracers die alten Ergebnisse einfach nur überschreiben sollen, ist diese Funktion deaktiviert. pAttachments ist ein Array der Länge attachmentCount, das aus VkPipelineColorBlendAttachmentState Strukturen besteht. Darin werden die Blendingeigenschaften jedes ColorAttachments beschrieben. BlendConstants werden für den Raytracer nicht verwendet.

```
	VkPipelineColorBlendAttachmentState
colorBlendAttachmentState = {};
	colorBlendAttachmentState.blendEnable = VK_FALSE;
	colorBlendAttachmentState.srcColorBlendFactor =
VK_BLEND_FACTOR_SRC_ALPHA;
	colorBlendAttachmentState.dstColorBlendFactor =
VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
	colorBlendAttachmentState.colorBlendOp = VK_BLEND_OP_ADD;
	colorBlendAttachmentState.srcAlphaBlendFactor =
VK_BLEND_FACTOR_ONE;
	colorBlendAttachmentState.dstAlphaBlendFactor =
VK_BLEND_FACTOR_ZERO;
	colorBlendAttachmentState.alphaBlendOp = VK_BLEND_OP_ADD;
```

```
      colorBlendAttachmentState.colorWriteMask =
VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT |
VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT;
```

Die erste Komponente blendEnable gibt an, ob Blending für das dazugehörige Attachment angewendet werden soll. Das ist für den Raytracer allerdings nicht nötig.

Die nächsten drei Komponenten beziehen sich auf einander. src* (Source) bezieht sich auf den Output des Fragmentshaders und dst* (Destination) auf den Inhalt des Farb Attachments. Die beiden Komponenten *ColorBlendFactor geben jeweils an, mit welchem Wert die Farben aus Source und Destination multipliziert werden, bevor die daraus resultierenden Ergebnisse mit der in colorBlendOp spezifizierten Operation kombiniert werden. Die entsprechenden Parameter für die Alpha Werte funktionieren analog.

Die letzte Komponente colorWriteMask gibt an, in welche Kanäle des Attachments geschrieben werden soll. Für den Raytracer werden alle Farbkanäle benötigt.

2.2.14 Command Buffer

Command Buffer werden benutzt, um Kommandos aufzunehmen, die dann später zur Ausführung an eine Queue gesendet werden können. Es gibt Primäre und Sekundäre Commandbuffer. Primäre Commandbuffer werden zur Ausführung direkt an eine Queue geschickt während Sekundäre Commandbuffer von Primären ausgeführt werden können. Der Draw Befehl, der das Rendering einleitet, ist ein Beispiel für ein verfügbares Kommando.

Der Command Buffer des Raytracers bindet die Pipeline, den VertexBuffer und die Descriptoren und führt anschließend den Draw Befehl aus.

```
      VkCommandPoolCreateInfo commandPoolCreateInfo = {};
      commandPoolCreateInfo.sType =
VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
      commandPoolCreateInfo.pNext = nullptr;
      commandPoolCreateInfo.flags = 0;
      commandPoolCreateInfo.queueFamilyIndex =
queueFamilyIndex;

      vkCreateCommandPool(device, &commandPoolCreateInfo,
nullptr, &commandPool);
```

queueFamilyIndex ist der Index der Queue Family des Device für die der Command Pool erzeugt werden soll.

Nachdem der Pool erstellt wurde, können daraus Command Buffer allokiert werden.

```
VkCommandBufferAllocateInfo commandBufferAllocateInfo = {};
	commandBufferAllocateInfo.sType = 
VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
	commandBufferAllocateInfo.pNext = nullptr;
	commandBufferAllocateInfo.commandPool = commandPool;
	commandBufferAllocateInfo.level = 
VK_COMMAND_BUFFER_LEVEL_PRIMARY;
	commandBufferAllocateInfo.commandBufferCount = 
swapchainImageCount;

	commandBuffers = new 
VkCommandBuffer[swapchainImageCount];
	vkAllocateCommandBuffers(device, 
&commandBufferAllocateInfo, commandBuffers);
```

commandPool ist das zuvor erstellte Command Pool Handle. Level gibt an, ob es sich bei dem Commandbuffer um einen primären oder sekundären Buffer handelt. Für den Raytracer wird nur ein Primärer Buffer erstellt. Die letzte Komponente CommandBufferCount gibt die Anzahl der Command Buffer an, die allokiert werden sollen.

Um einen Command Buffer zu benutzen, muss dieser mit Kommandos für das Device gefüllt werden. Dazu werden die Kommandos aufgenommen. Das passiert mithilfe von Command Funktionen. Um die Aufnahme zu starten, muss zuerst vkBeginCommandBuffer() ausgeführt werden.

```
vkBeginCommandBuffer(commandBuffers[i], 
&commandBufferBeginInfo) );
```

commandBuffers ist das zuvor erstellte Command Buffer Handle. CommandBufferBeginInfo ist die dazugehörige Struktur, die die Parameter enthält.

```
	VkCommandBufferBeginInfo commandBufferBeginInfo = {};
	commandBufferBeginInfo.sType = 
VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
	commandBufferBeginInfo.pNext = nullptr;
	commandBufferBeginInfo.flags = 0;
	commandBufferBeginInfo.pInheritanceInfo = nullptr;
```

pInheritanceInfo wird dann benötigt, wenn ein sekundärer Command Buffer aufgenommen wird. In diesem Fall wird in der Komponente angegeben welcher State des primären Command Buffers von dem sekundären Command Buffer geerbt werden soll. Da es sich bei dem Command Buffer des Raytracers um einen primären Buffer handelt, wird diese Komponente ignoriert.

Durch das Ausführen von vkCmd* Funktionen werden Kommandos in den Command Buffer geschrieben. Diese Kommandos können später auf einem Device ausgeführt werden.

Zuerst wird der Renderpass gestartet. Die dazugehörige Funktion sieht wie folgt aus.

```
	vkCmdBeginRenderPass(commandBuffers[i], 
&renderpassBeginInfo, VK_SUBPASS_CONTENTS_INLINE);
```

Der erste Parameter ist der Command Buffer in den das Kommando aufgenommen werden soll. RenderpassBeginInfo ist ein Pointer auf die Struktur, die einen Großteil der Parameter für die Funktion enthält. Der letzte Parameter gibt an, dass die die Kommandos in den Primären Command Buffer aufgenommen werden, und keine sekundären Buffer in dem Subpass ausgeführt werden dürfen.

```
	VkRenderPassBeginInfo renderpassBeginInfo = {};
	renderpassBeginInfo.sType = 
VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
	renderpassBeginInfo.pNext = nullptr;
	renderpassBeginInfo.renderPass = renderPass;
	renderpassBeginInfo.framebuffer = framebuffers[i];
	renderpassBeginInfo.renderArea.offset = {0, 0};
	renderpassBeginInfo.renderArea.extent = {WIDTH, HEIGHT};
	renderpassBeginInfo.clearValueCount = 0;
	renderpassBeginInfo.pClearValues = nullptr;
```

renderPass ist das Handle des Renderpass, für den die Kommandos aufgenommen werden. framebuffer ist das Handle des Framebuffers, in den die Ergebnisse des Rendering geschrieben werden sollen. RenderArea kann genutzt werden, um das Rendering nur für einen Teil des Bildes auszuführen. Dazu wird eine VkRect2D Struktur benutzt, die schon für Scissor im Viewport State der Pipeline erklärt wurde. Für den Raytracer wird das gesamte Bild verwendet. pClearValues ist ein Array der Länge clearValueCount, das VkClearValue Unions enthält. VkClearValue enthält eine Farbe und einen Tiefenwert. Das Array wird dazu benutzt um die Clear Werte für Attachments anzugeben, wenn für die Attachments die Ladeoperation VK_ATTACHMENT_LOAD_OP_CLEAR benutzt wird. Da das

einzigeAttachment des Raytracers die Operation *DONT_CARE benutzt sind die beiden Parameter unwichtig.

Danach werden für den Commandbuffer des Raytracers folgende Kommandos ausgeführt.

Das Binden der Pipeline an den Renderpass:

```
	vkCmdBindPipeline(commandBuffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS, pipeline);
```

Das Binden der Vertexbuffer an den Renderpass:

```
	VkDeviceSize offsets[] = {0};
	vkCmdBindVertexBuffers(commandBuffers[i], 0, 1, &vertexBuffer, offsets);
```

Binden der Descriptorsets:

```
	vkCmdBindDescriptorSets(commandBuffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS, pipelineLayout, 0, 1, &descriptorSet[0], 0, nullptr);
```

Der Draw Befehl:

```
	vkCmdDraw(commandBuffers[i], (uint32_t) VKRT::vertices.size(), 1, 0, 0);
```

Als letztes werden der Renderpass und die Command Buffer Aufnahme beendet.

```
	vkCmdEndRenderPass(commandBuffers[i]);

	vkEndCommandBuffer(commandBuffers[i]);
```

Im Vulkanprogramm wird für jeden Framebuffer, und damit für jedes Bild in der Swapchain, ein Commandbuffer aufgenommen.

2.2.15 Renderloop

Damit ist das Programm fast vollständig. Als letztes wird das Rendering in einer Schleife ausgeführt.

Semaphoren sind eines der Synchronisationprimitive die Vulkan bereitstellt. Semaphoren werden in Vulkan benutzt um den Besitz von Ressourcen zwischen den Queues eines Device zu kontrollieren. Im Raytracerprogramm werden sie verwendet um sicherzustellen, dass die einzelnen Schritte der Renderschleife abgeschlossen sind, bevor der jeweils nächste Schritt ausgeführt wird.

```
	VkSemaphoreCreateInfo semaphoreCreateInfo = {};
	semaphoreCreateInfo.sType = 
VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
	semaphoreCreateInfo.pNext = nullptr;
	semaphoreCreateInfo.flags = 0;

	vkCreateSemaphore(device, &semaphoreCreateInfo, nullptr, 
&semaphoreImageAvailable);
```

Ein Semaphore kann mit der Funktion vkCreateSemaphore erzeugt werden. Device ist das Device Handle. semaphoreCreateInfo ist ein Pointer auf eine Struktur, die im Fall der Semaphoren ausschließlich die drei üblichen Parameter besitzt. Der dritte Parameter ist der ungenutzte pAllocator und semaphoreImageAvailable ist ein Pointer auf das Semaphore Handle, das überschrieben werden soll.

Innerhalb der Schleife wird zuerst die Funktion vkAcquireNextImageKHR() ausgeführt.

```
vkAcquireNextImageKHR(device, swapchain, 
std::numeric_limits<uint64_t>::max(), semaphoreImageAvailable, 
VK_NULL_HANDLE, &imageIndex);
```

Diese Funktion ist Teil der Swapchain Extension und gibt den Index des nächsten verfügbaren Bildes der Swapchain in der Variable imageIndex zurück. Device und swapchain sind die Handle von device und swapchain, Der dritte Parameter gibt in Nanosekunden an, wie lange die Funktion warten soll wenn kein Bild verfügbar ist. SemaphoreImageAvailable ist ein Semaphore das signalisiert werden soll wenn ein Bild verfügbar ist. Im fünften Parameter kann ein Fence angegeben werden, der signalisiert werden soll. Da kein Fence verwendet wird, wird stattdessen VK_NULL_HANDLE übergeben.

Nachdem der Index des Bildes ermittelt wurde, kann dieses für das Rendering benutzt werden. Das Rendering wird gestartet, indem der Command Buffer von dem Device ausgeführt wird.

```
vkQueueSubmit(queue, 1, &submitInfo, VK_NULL_HANDLE);
```

Queue ist das Handle der Queue. SubmitInfo ist ein VkSubmitInfo Array, dessen Länge im zweiten Parameter angegeben wird. Der letzte Parameter ist entweder VK_NULL_HANDLE oder ein Fence.

```
	VkPipelineStageFlags waitStages = 
{VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT};

	VkSubmitInfo submitInfo = {};
	submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
	submitInfo.pNext = nullptr;
	submitInfo.waitSemaphoreCount = 1;
	submitInfo.pWaitSemaphores = &semaphoreImageAvailable;
	submitInfo.pWaitDstStageMask = &waitStages;
	submitInfo.commandBufferCount = 1;
	submitInfo.pCommandBuffers = &commandBuffers[imageIndex];
	submitInfo.signalSemaphoreCount = 1;
	submitInfo.pSignalSemaphores = &semaphoreRenderingDone;
```

pWaitSemaphores ist ein Array der Länge waitSemaphoreCount, das VkSemaphore enthält, auf die die Command Buffer warten sollen. pWaitDstStageMask ist ein Array das ebenfalls die Länge waitSemaphoreCount hat. Darin werden die Pipeline Stages angegeben, an denen auf die Semaphoren gewartet werden soll. Der Raytracer muss warten sobald das Bild der Swapchain benötigt wird. pCommandBuffers ist ein Array der Länge commandBufferCount, das alle Command Buffer enthält, die in diesem Aufruf von der Queue abgearbeitet werden sollen. pSignalSemaphore ist ein Array der Länge signalSemaphoreCount, das Semaphoren enthält, die signalisiert werden sollen, sobald die Command Buffer abgearbeitet wurden.

Als letztes muss das fertige Bild auf dem Bildschirm angezeigt werden. Dazu wird die Funktion QueuePresentKHR verwendet.

```
	vkQueuePresentKHR(queue, &presentInfo);
```

vkQueuePresentKHR ist ebenfalls Teil der Swapchain Exension. Queue ist die Queue, die verwendet werden soll um die Präsentation durchzuführen. PresentInfo ist die Struktur, die die nötigen Parameter enthält.

```
VkPresentInfoKHR presentInfo = {};
presentInfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
presentInfo.pNext = nullptr;
presentInfo.waitSemaphoreCount = 1;
presentInfo.pWaitSemaphores = &semaphoreRenderingDone;
presentInfo.swapchainCount = 1;
presentInfo.pSwapchains = &swapchain;
presentInfo.pImageIndices = &imageIndex;
presentInfo.pResults = nullptr;
```

pWaitSemaphores ist ein Array der Länge waitSemaphoreCount, das VkSemaphore enthält, auf die die Präsentation warten soll. Für die Präsentation muss das Rendering abgeschlossen sein. pSwapchains ist ein Array der Länge swapchainCount, das die Swapchains enthält auf die Präsentiert werden soll. Für den Raytracer wird nur eine Swapchain verwendet. pImageIndices ist ein Array, das für jede angegebene Swapchain den Index des Bildes auf das präsentiert werden soll enthält. Die letzte Komponente pResults kann ebenfalls ein Array der Länge swapchainCount sein. Die Elemente des Arrays werden nach der Ausführung der Funktion für jede Swapchain mit VkResult Werten überschrieben. Wenn wie beim Raytracer keine VkResults für jede einzelne Swapchain benötigt werden, kann diese Komponente auf NULL gesetzt werden.

2.2.16 Beenden von Vulkan

Um eine Vulkan Anwendung korrekt zu beenden müssen alle Vulkan Objekte gelöscht werden, die noch nicht gelöscht wurden. Für den Raytracer bedeutet das, dass die meisten der zuvor erstellten Handle zerstört werden müssen. Einige Vulkan Objekte werden Implizit zerstört, beispielsweise werden die physical Device Handles zerstört wenn die Instance zerstört wird. Für jedes Handle gibt es eine eigene Funktion, die dafür zuständig ist. Grundsätzlich ist es Sinnvoll Handle zu zerstören sobald diese nicht mehr benötigt werden, allerdings benötigt das Raytracerprogramm sämtliche Ressourcen bis zum Ende des Programms. Wenn ein Handle verwendet wurde um ein anderes Handle zu erzeugen, dann muss das erzeugte Handle zuerst zerstört werden. Beispielsweise muss das Device vor der Instance zerstört werden. Die unten zu sehende Funktion wird zum Zerstören der Instance verwendet. instance ist das Handle der Instance, der zweite Parameter ist pAllocator, der nur verwendet wird, wenn auch zum Erstellen der Instance ein externer Allokator verwendet wird.

```
vkDestroyInstance(instance, nullptr);
```

Die anderen Funktionen funktionieren analog, allerdings benötigen einige weitere Parameter, wie das Device, das verwendet wurde um das Handle zu erstellen.

2.3 Der Fragmentshader und der Raytracer

Um GLSL für das Erzeugen von SPIR-V zu können, muss der GLSL Shader mit folgenden Zeilen beginnen:

```
#version 450 core
#extension GL_ARB_seperate_shader_objects : enable
```

Bei dem Raytracer handelt es sich um eine an Vulkan angepasste Version des rekursiven Raytracers, der Teil des CVK ist. Wie bereits erwähnt wird keine Beschleunigungsdatenstruktur verwendet. Der originale Raytracer kann neben einfachen Materialien auch Texturen verwenden, um die Farbe der Objekte in der Szene zu bestimmen. Diese Funktion wurde für den Vulkan Raytracer entfernt, da die dazu verwendeten GLSL Extensions nicht mit Vulkan kompatibel sind. Außerdem wurden die Uniform Variablen durch einen Uniform Buffer ersetzt, da in Vulkan keine Uniform Variablen in der Form wie es sie in OpenGL gibt verfügbar sind.

Als erstes wird aus dem Frustum und der Fensterkoordinate ein Strahl berechnet. Das Frustum ist über einen Augpunkt, die linke obere Ecke der Fläche, die den Bildschirm in der Szene repräsentiert, und die Größe der Schritte in x- und y-Richtung definiert. Danach wird getestet, ob dieser Strahl eines der Objekte der Szene schneidet. Dabei werden zuerst die Variablen für Gewicht und Tiefe des Rekursionsschritt initialisiert. Die Tiefe gibt an, um den wievielten Rekursionschritt es sich handelt. Gewicht bestimmt zu welchem Anteil Farben zu der Fragmentfarbe addiert werden und ob rekursive Strahlen weiter verfolgt werden sollen. Dann startet die Schleife zur Berechnung der Fragmentfarbe. Gibt es einen Schnittpunkt wird die Farbe des Objekts an dieser Stelle berechnet, und das Ergebnis wird gewichtet zu der Output Farbe addiert. Dabei werden alle Lichtquellen für die Beleuchtung und alle Objekte für mögliche Verschattung beachtet. Anschließend werden die Sekundärstrahlen, sofern es diese für das Objekt gibt, berechnet. Mithilfe des Gewichts und der Tiefe wird bestimmt, ob die Sekundärstrahlen weiter verfolgt werden sollen. Wird ein Strahl weiter verfolgt, wiederholt sich die Schleife. Andernfalls endet sie. Wird kein Schnittpunkt gefunden, wird statt einer Objektfarbe die Hintergrundfarbe zur Fragmentfarbe Addiert. Endet die Schleife wird die Summe der von den Strahlen gesammelten Farben als Fragmentfarbe ausgegeben und das Shaderprogramm endet.

2.4 Zusammenfassung der OpenGL Version des Raytracerprogramms

Der OpenGL Raytracer besteht anders als der Vulkan Raytracer aus zwei Komponenten, dem Kontrollprogramm und der Szenenklasse, sowie einigen angepassten Klassen aus dem CVK. Die OpenGL Version verwendet die gleichen CVK Klassen wie Vulkan. Allerdings wird noch zusätzlich die Klasse CVK_ShaderSet benutzt. Die Szenenklasse der OpenGL Version erbt die Funktionalität von ShaderSet und besitzt, anders als die Szenenklasse der Vulkan Version, keine Getter Funktionen. Stattdessen sind Funktionen vorhanden, die die Shaderressourcen direkt binden. Alles andere passiert direkt im Kontrollprogramm.

3. Vergleich der Performance der Raytracer

In diesem Kapitel wird die Performance des Vulkan Raytracers mit der des OpenGL Raytracers verglichen. Dazu werden verschiedene Szenen in beiden Raytracern gerendert. Zum Vergleich wird die Zeit, die ein Durchlauf der Renderschleife benötigt, gemessen.

Am Anfang der Renderschleife wird in beiden Programmen mithilfe von `glfwGetTime()` die Zeit gemessen. Um sicherzustellen, das die gesamte Arbeit auf der Graphikkarte abgeschlossen wurde, wird in dem OpenGL Programm die Funktion `glFinish()` aufgerufen. In dem Vulkan Programm werden die beiden Funktionen `vkDeviceWaitIdle(device)` und `vkQueueWaitIdle(queue)` verwendet. Diese beiden Funktionen warten, bis Device und Queue keine ausstehenden Aufgaben mehr haben. Anschließend wird die aktuelle Zeit erneut gemessen, um die in der Schleife vergangene Zeit daraus zu berechnen.

Für die einzelnen Szenen werden jeweils 60 Schleifendurchläufe abgewartet. Aus den sich daraus ergebenden Messwerten wird für jeden Workload der Mittelwert und Median ermittelt. Dazu werden die letzten 30 Messwerte verwendet.

Nachfolgend werden die Szenen beschrieben und die gemessenen Ergebnisse vorgestellt. Die jeweiligen Szenen sind abgebildet. Die Szenen des Vulkan Raytracers sind gegenüber denen des OpenGL Raytracers horizontal gespiegelt, da die y-Achse wie schon erwähnt bei Vulkan invertiert ist.

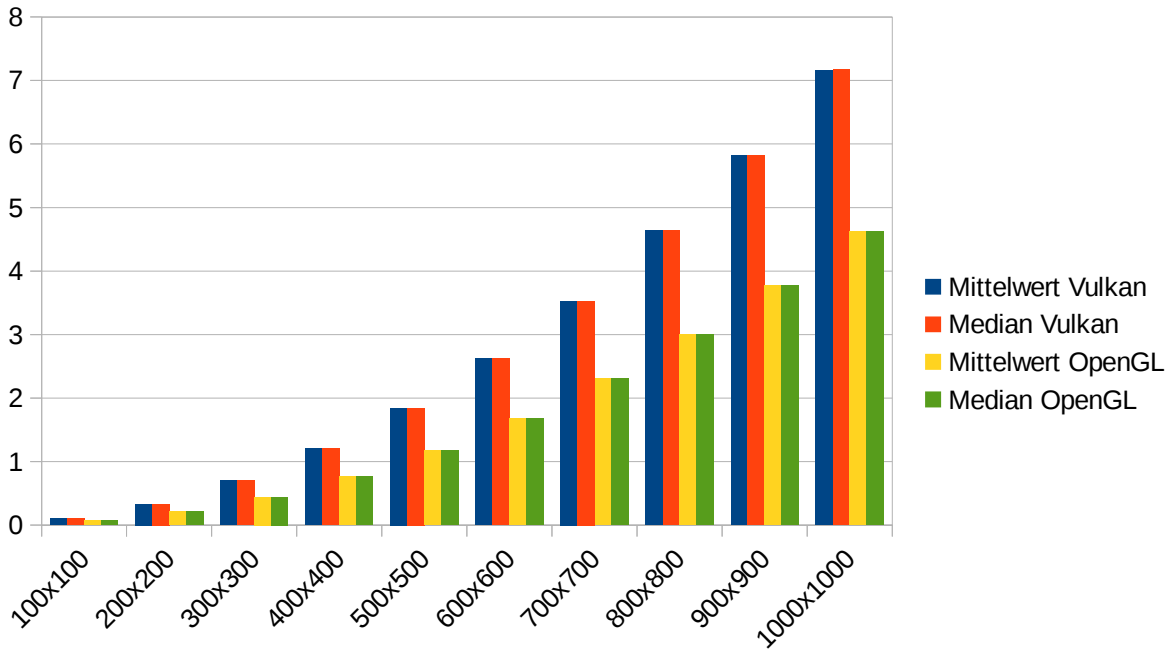
3.1 Szene 1: Teapot

Die Teapot Szene enthält den Teapot, der aus 19200 Dreiecken besteht, und eine Kugel, die nicht auf dem Bild zu sehen ist. Zusätzlich gibt es drei Lichtquellen. Die Positionen und Eigenschaften aller Objekte sind in den Szenen beider Raytracer identisch.

3.1 Szene 1: Teapot Vergleich

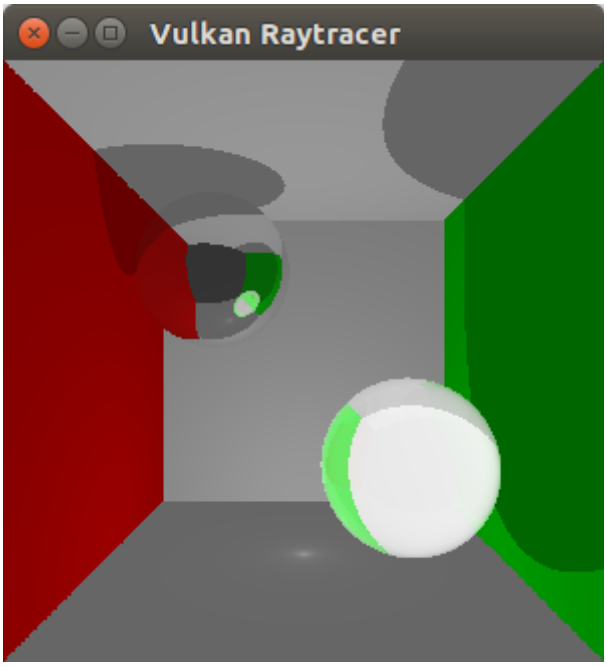
Die gemessene Zeit für einen Schleifendurchlauf ist für das Vulkanprogramm deutlich höher als für die OpenGL Version. OpenGL benötigt nur zwischen 62 bis 68 Prozent der Zeit, die die Vulkan Version benötigt. Daraus lässt sich schließen, dass die Bildgröße beide Programme in einem ähnlichen Verhältnis beeinflusst.

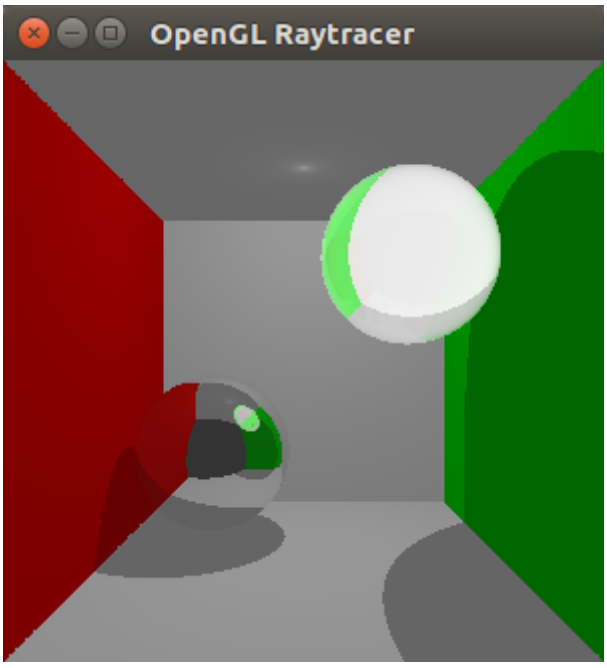
Auflösung	Mittelwert Vulkan	Median Vulkan	Mittelwert OpenGL	Median OpenGL
100x100	0.10543822	0.1055295	0.0713465867	0.0707357
200x200	0.3328785667	0.333995	0.2176877333	0.2155585
300x300	0.7072104333	0.7069605	0.4405426	0.4407045
400x400	1.2012016667	1.20292	0.7704225	0.770545
500x500	1.8444706667	1.843485	1.178893	1.17875
600x600	2.615302	2.61568	1.685576	1.686065
700x700	3.5264053333	3.527415	2.3112303333	2.31007
800x800	4.634234	4.6338	3.006991	3.006775
900x900	5.8252436667	5.822515	3.7747406667	3.773785
1000x1000	7.1625706667	7.1684	4.6215676667	4.619585



3.2 Szene 2: Box

Die Box Szene besteht aus 30 Dreiecken und zwei Kugeln sowie einer Lichtquelle. Eine der Kugeln verwendet ein spiegelndes Material, die andere ein Glasmaterial.

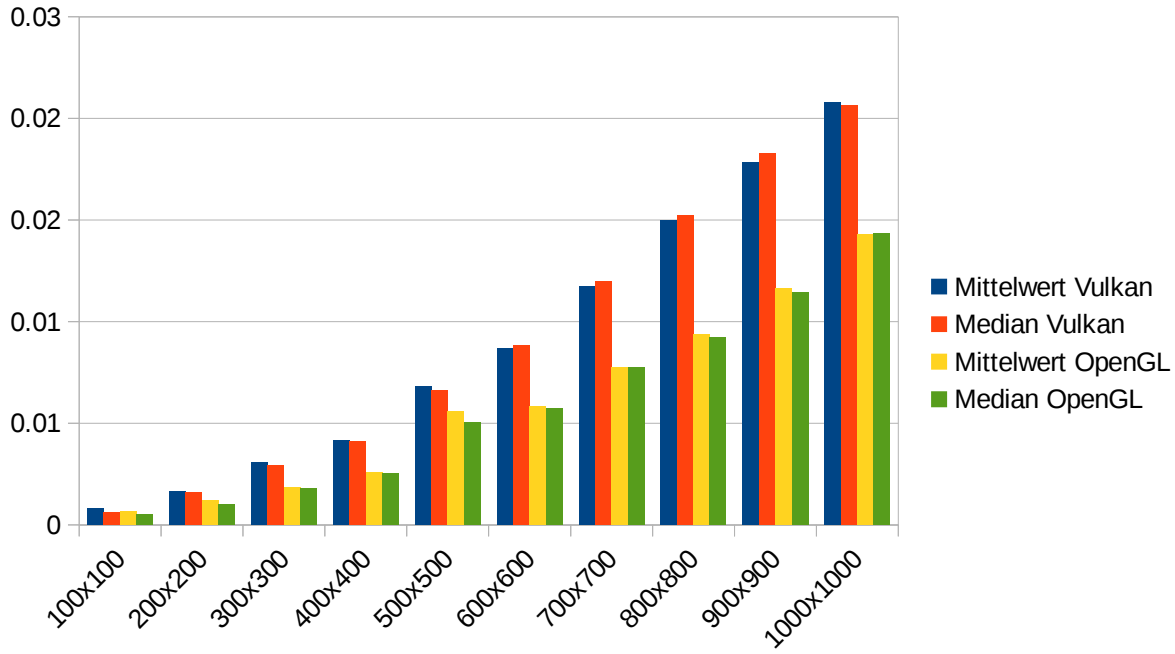




3.2 Szene 2: Box Vergleich

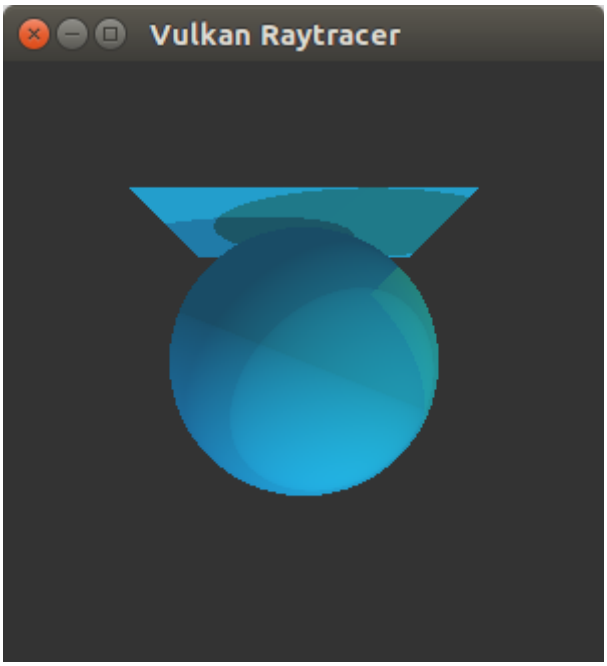
Für die Box Szene wurden ebenfalls bessere Ergebnisse für OpenGL gemessen. Die Zeit, die OpenGL benötigt liegt zwischen 60 und 82 Prozent der Zeit, die Vulkan benötigt. Bei dieser Szene sind die Ergebnisse nicht so eindeutig wie bei der Teapot Szene, allerdings ist das vermutlich auf die höhere Störungsanfälligkeit bei den kleineren Messwerten zurückzuführen. Bei 7 der 10 Messwerte benötigt OpenGL nur 60 bis 69 Prozent der Zeit, die Vulkan benötigt.

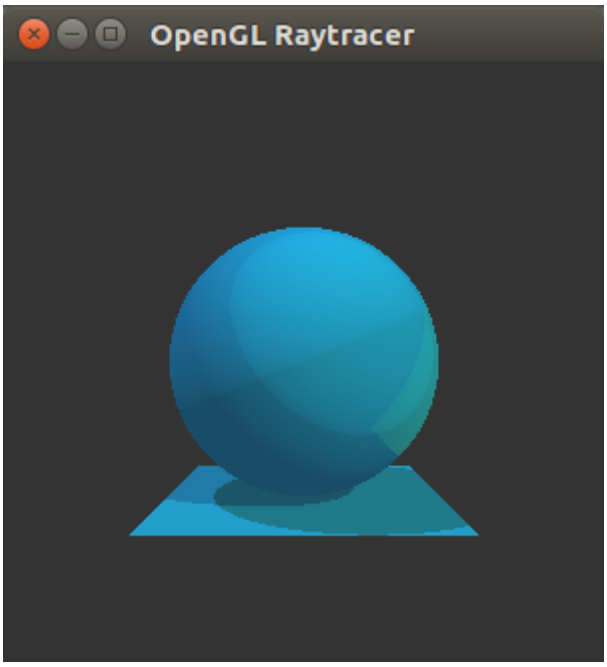
Auflösung	Mittelwert Vulkan	Median Vulkan	Mittelwert OpenGL	Median OpenGL
100x100	0.0008317141	0.0006292255	0.0006694513	0.0005306215
200x200	0.0016253277	0.00159365	0.0011869789	0.001021175
300x300	0.0030897473	0.00291111	0.001846019	0.001798115
400x400	0.0041518897	0.004126635	0.0025868197	0.00253419
500x500	0.0068049883	0.00662194	0.0055786577	0.00504247
600x600	0.0087059543	0.00881805	0.005827543	0.005724385
700x700	0.0117271633	0.01199465	0.0077521387	0.007757235
800x800	0.0149933633	0.01521265	0.0093580527	0.00923921
900x900	0.0178406767	0.0182829	0.01165269	0.0114558
1000x1000	0.0207804233	0.02062525	0.01428352	0.01436395



3.3 Szene 3: Sphere List

Die Sphere List Szene enthält 6 Dreiecke, sowie eine Kugel. Der Unterschied zu den anderen Szenen ist, dass nicht die Größe des Fensters verändert wird, sondern die Anzahl der Objekte in der Szene. Dabei befinden sich die zusätzlichen Objekte, bei denen es sich um identische Kugeln handelt, innerhalb der sichtbaren Kugel. Das Fenster hat immer die Größe von 300x300 Pixeln.

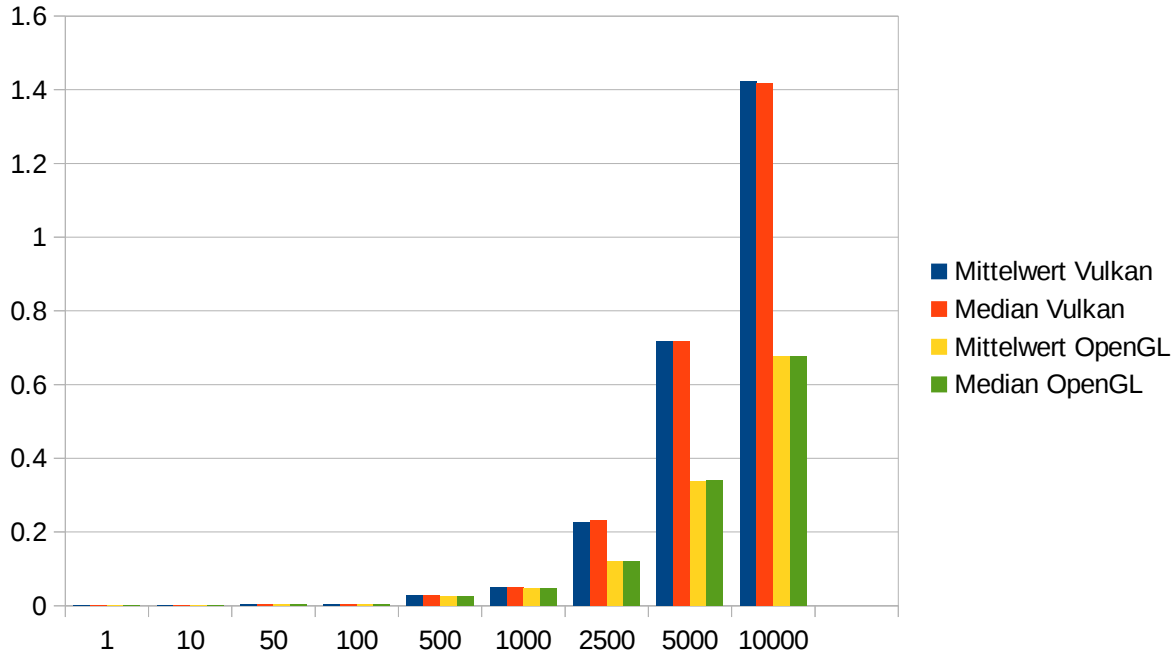




3.3 Szene 3: Sphere List Vergleich

Das Vulkanprogramm hat auch für die Sphere List Szene größtenteils schlechtere Ergebnisse erzielt als die OpenGL Version. Für die Szenen mit einer, 10 und 50 zusätzlichen Kugeln ist die gemessene Zeit, die OpenGL benötigt, zwischen ungefähr 10 bis 25 Prozent geringer. Interessant ist, insbesondere unter Betrachtung der Ergebnisse der anderen Szenen, dass die Renderzeit der beiden Programme für 100, 500 und 1000 Kugeln sehr ähnlich ist, und im Fall von 100 Kugeln sogar eine geringere Zeit für Vulkan gemessen wurde. Allerdings benötigt der Vulkan Raytracer bei 2500 und mehr Kugeln doppelt so viel Zeit wie der OpenGL Raytracer.

Anzahl der Kugeln	Mittelwert Vulkan	Median Vulkan	Mittelwert OpenGL	Median OpenGL
1	0.0006303075	0.0005746165	0.0005094952	0.0004479535
10	0.0010421857	0.000997497	0.0009230848	0.000842557
50	0.003140918	0.00297927	0.0023895687	0.00227506
100	0.004341595	0.004277065	0.0047414317	0.004829015
500	0.02755348	0.027261	0.0258586533	0.02588175
1000	0.0490867833	0.048556	0.04801034	0.04779735
2500	0.2270002667	0.232047	0.1202058	0.120451
5000	0.7167208667	0.7181225	0.3376599667	0.339126
10000	1.424089	1.416195	0.6754984667	0.6764515



3.4 Einschätzung der Ergebnisse

Die Tests haben gezeigt, dass die OpenGL Version des Raytracers effizienter Arbeitet als die Vulkan Version. Die ersten beiden Tests legen nahe, dass die Größe des Bildes im Verhältnis zur Leistung des Vulkan Programms im Vergleich zur OpenGL Version keine große Rolle spielt. Das bedeutet, dass sich vor allem die Arbeit pro Pixel negativ auf die Leistung von Vulkan auswirkt, und nicht die Anzahl an Aufgaben in Form von Pixelberechnungen. Gleichzeitig scheint es einen Leistungsbereich zu geben, in dem der Vulkan Raytracer effektiver ist als der OpenGL Raytracer.

4. Beurteilung von Aufwand gegenüber Performance

In Bezug auf den Aufwand im Vergleich zum Performancegewinn kann man sagen, dass der Aufwand sich im Fall dieses Raytracers nicht lohnt. Das schreiben des Vulkanprogramms ist deutlich aufwändiger als das der OpenGL Version, trotzdem ist die Performance in den meisten Fällen deutlich schlechter ausgefallen. Natürlich bedeutet das nicht, dass sich Vulkan niemals lohnt. Viele der grundlegenden Konzepte mit denen Vulkan seine potentiell bessere Performance erreicht sind in dem Raytracerprogramm nicht zur Anwendung gekommen. Beispielsweise wurde nur ein Prozessorkern verwendet, weil mehr für den Raytracer nicht notwendig sind. Hat man aber ein Anwendung die an dieser Stelle profitieren kann, ist es fast sicher, dass mit Vulkan eine bessere Performance erzielt werden kann. Es ist auch nicht unwahrscheinlich, das auch das Raytracerprogramm optimiert werden kann, so dass es eine vergleichbare oder bessere Leistung als die OpenGL Version erbringt. Allerdings würde das einen noch größeren Aufwand bedeuten. Um zu beantworten, ob sich der Mehraufwand lohnt, sollte man sich die Frage stellen, was das Ziel ist. Vulkan eignet sich nicht für simple Programme, die keine besonderen Anforderungen an die Performance stellen, oder für Programme die diese Anforderungen auch problemlos mit einem einfacheren API erreichen.[WIT] In diesem Fall ist es meistens sinnvoller ein API wie OpenGL zu verwenden. Ist die Performance allerdings wichtig, und ist das genaue Verhalten der Anwendung bekannt, bietet sich die Verwendung von Vulkan an.

5. Fazit

In dieser Arbeit wurde ein Raytracerprogramm vorgestellt, das die Vulkan API nutzt. Dabei wurden die programmiertechnischen Grundlagen von Vulkan an dem Beispiel des Programms erläutert. Durch die Erklärung wurde außerdem die hohe Komplexität der API aufgezeigt. Anhand des Vergleichs mit der OpenGL Version des Raytracers konnte man sehen, das Vulkan nicht automatisch bessere Ergebnisse liefert als eine API wie OpenGL.

6. Quellenverzeichnis

Literatur:

[SK16] Sellers, Graham; Kessenich, John: Vulkan Programming Guide. Addison-Wesley Education Publishers Inc, New Jersey, 2016. ISBN 978-0-13-446454-1

Internetquellen:

[KHR01] Vulkan 1.1.71 – A Specification (with all registered Vulkan extensions)

URL: https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#VkSurfaceTransformFlagBitsKHR

Autor: Khronos Vulkan Working Group

Zuletzt Aufgerufen: 29.3.18

[KHR02] Vulkan 1.0.71 – A Specification

URL: https://www.khronos.org/registry/vulkan/specs/1.0/pdf/vkspec.pdf

Autor: Khronos Vulkan Working Group

Zuletzt Aufgerufen: 29.3.18

[OVE] Vulkan Tutorial

URL: https://vulkan-tutorial.com/Introduction

Autor: Alexander Overvoorde

Zuletzt Aufgerufen: 29.3.18

[KHR03] Vulkan FAQ

URL: https://github.com/KhronosGroup/Khronosdotorg/blob/master/api/vulkan/faq.md#vulkan-where-how-when

Autor: Khronos Group

Zuletzt Aufgerufen: 29.3.18

[GVG] GLFW Vulkan Guide

URL: http://www.glfw.org/docs/latest/vulkan_guide.html

Autor: Unbekannt

Zuletzt Aufgerufen: 29.3.18

[WIT] The Most Common Vulkan Mistakes
URL: http://32ipi028l5q82yhj72224m8j-wpengine.netdna-ssl.com/wp-content/uploads/2016/05/Most-common-mistakes-in-Vulkan-apps.pdf
Autor: Dominik Witczak
Zuletzt Aufgerufen: 29.3.18

[KOK] SPIR-V Specification 29.3
https://www.khronos.org/registry/spir-v/specs/unified1/SPIRV.pdf
Autor: John Kessenich, Boaz Ouriel, Raun Krisch
Zuletzt Aufgerufen: 29.3.18